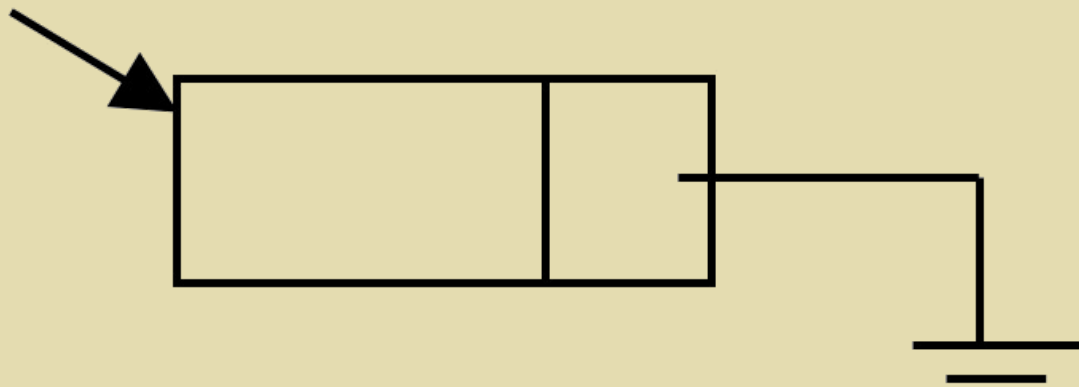


header



null

FIRST BOOK FOR DATA STRUCTURES & ALGORITHMS IN JAVA

VISHNU KOTRAJARAS

First Book for Data
Structures & Algorithms in
Java

Vishnu Kotrajaras

Vishnu Kotrajaras (วิชณู โคตรจรัส)

First Book for Data Structures & Algorithms in Java / Vishnu Kotrajaras (วิชณู โคตรจรัส)

1. โครงสร้างข้อมูลและอัลกอริทึม
2. ภาษาโปรแกรม จาวา

พิมพ์ครั้งที่ 1 จำนวน 100 เล่ม พ.ศ. 2561

สงวนลิขสิทธิ์ตาม พ.ร.บ. ลิขสิทธิ์ พ.ศ. 2537/2540

โดย วิชณู โคตรจรัส

การผลิตและลอกเลียนตำราเล่มนี้ไม่ว่ารูปแบบใดทั้งสิ้น
ต้องได้รับอนุญาตเป็นลายลักษณ์อักษรจากเจ้าของลิขสิทธิ์

จัดพิมพ์โดย

วิชณู โคตรจรัส

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

จุฬาลงกรณ์มหาวิทยาลัย

พญาไท กรุงเทพฯ 10330

ออกแบบปก: วิชณู โคตรจรัส

ออกแบบรูปเล่ม: วิชณู โคตรจรัส

ภาพประกอบ: คามิน กลยุทสกุล

พิมพ์ที่ ห้างหุ้นส่วนจำกัด ภาณุธร โทรศัพท์ 081-9245642, 086-4265269

26/114 ถนนพระรามที่ 2 แขวงบางมด กรุงเทพฯ

Vishnu Kotrajaras

First Book for Data Structures & Algorithms in Java / Vishnu Kotrajaras

1. Data Structures and Algorithms
2. Java Programming Language

First Edition, 100 copies, 2018.

Copyright 2018 by Vishnu Kotrajaras.

No part of this publication may be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, or scanning, without prior permission in writing from the author.

Printed in Thailand by

Vishnu Kotrajaras

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Payathai, Bangkok 10330

Cover Design: Vishnu Kotrajaras

Print Format Design: Vishnu Kotrajaras

Illustration: Kamin Kolyutsakul

Printed at Panuthorn L.P. Tel. 081-9245642, 086-4265269

26/114 Rama 2 Road, Bang Mod, Bangkok, Thailand

To my wife for her love and support.

Preface

Data structures is one of the fundamental subjects students in Computer Science and Computer Engineering have to learn to master. Usually, this subject is taught after students passed at least one basic programming course. The author has been involved with this subject for over two decades, being a student, a teaching assistant, and eventually a professor who teaches the subject. Students who had problems with this subject were either unable to draw pictures to help them understand various concepts, or unable to translate from pictures to code. This textbook, developed from class notes the author wrote for Fundamental Data Structure and Algorithm course offered to students in the International School of Engineering, Chulalongkorn University, tries to present concepts with many pictures, together with code explanations, to help students overcome those problems. The author hopes this textbook helps improve readers' understanding and enjoyment for the subject.

Table of Contents

CHAPTER 1 : DATA STRUCTURES AND OUR PROGRAM.....	1
What is a Data Structure?	1
This book's organization.....	6
CHAPTER 2 : INTRODUCTION TO ALGORITHM ANALYSIS	11
Running Time Analysis	11
Asymptotic Notation.....	20
<i>Asymptotic Notations and Nested Loop</i>	22
<i>Asymptotic Runtime and Consecutive Operations</i>	25
<i>Asymptotic Runtime and Conditional Operations</i>	26
<i>Asymptotic Runtime and Recursion</i>	27
<i>Asymptotic Runtime in Logarithmic Form.....</i>	28
<i>Asymptotic Runtime and Its Application in Choosing</i> <i>Implementation.....</i>	38
<i>Best-Case, Worst-Case, and Average Case Runtime</i>	41
Beyond Big-Theta and Big-O.....	44
CHAPTER 3 : LIST	49
List and Its Operations.....	49
Implementing a List with Array.....	49
Implementing a List with Linked List	58
Doubly-linked list.....	80
Sparse Table	97
Skip List.....	100
CHAPTER 4 : STACK.....	109
Stack Operations.....	109
Notable uses of Stack	110
<i>Bracket Pairing</i>	110
<i>Handling Data for Method Calls.....</i>	114
<i>Postfix Calculation.....</i>	119
<i>Transforming Infix to Postfix Form</i>	123
Implementing a Stack with Array	137
Implementing a Stack with Linked List.....	141

CHAPTER 5 : QUEUE.....	155
Queue Operations.....	155
Implementing a Queue with Array.....	158
Implementing a Queue with Linked List.....	166
Double-Ended Queue.....	172
Implementing a Double-Ended Queue with Array.....	174
Double-Ended Queue implemented with Linked List.....	177
Application of Queue: Radix Sort.....	180
CHAPTER 6 : BINARY TREE.....	195
Interesting properties of A Binary Tree.....	200
Binary Search Tree.....	204
Binary Search Tree Implementation.....	206
Recursive Implementation of Binary Search Tree.....	233
Recursive Tree Traversal.....	241
Breadth-First Tree Traversal.....	244
CHAPTER 7 : HASH TABLE.....	255
Designing A Hash Function.....	257
<i>Transforming our key.....</i>	<i>257</i>
<i>Making our integer more widely distributed.....</i>	<i>259</i>
<i>Transforming our value into array index.....</i>	<i>261</i>
Separate Chaining Hash Table.....	262
<i>Implementation of Separate Chaining Hash Table.....</i>	<i>263</i>
Runtime Analysis of Separate Chaining Hash Table.....	269
Open Addressing Hash Table.....	271
<i>Linear Probing.....</i>	<i>271</i>
<i>Quadratic Probing.....</i>	<i>277</i>
<i>Double Hashing.....</i>	<i>281</i>
<i>Implementation of Open Addressing Hash Table.....</i>	<i>282</i>
Separate Chaining VS Open Addressing.....	289
CHAPTER 8 : SORTING.....	293
Bubble Sort.....	293
Selection Sort.....	295
Insertion Sort.....	299

Merge Sort.....	301
<i>Array Splitting</i>	301
<i>Sorting Each Portion</i>	303
<i>Merging Two Sorted Portions</i>	304
<i>Implementation and Runtime of Merge Sort</i>	305
Quick Sort	310
<i>Choosing a Pivot</i>	312
<i>Partitioning</i>	313
<i>Implementation and Runtime of Quick Sort</i>	316
Bucket Sort.....	324
CHAPTER 9 : PRIORITY QUEUE.....	331
Implementation Choices	333
Linked List Implementation of Priority Queue.....	334
Heap	338
Heap Implementation and Runtime Analysis.....	338
Priority Queue Application: Data Compression.....	350
CHAPTER 10 : AVL TREE.....	365
Rebalancing the Tree.....	367
Implementation of AVL Tree.....	371
<i>Node Implementation</i>	371
<i>Iterator Implementation</i>	374
<i>Tree Implementation</i>	374
BIBLIOGRAPHY.....	388
INDEX.....	390

List of Figures

Figure 1-1: Linked list storing 5 integers.	3
Figure 1-2: Binary Search Tree that stores 5 integers.	3
Figure 1-3: Maxheap storing 5 integers.....	4
Figure 1-4: Following pointers to the last data in our linked list.	5
Figure 1-5: Book organization.....	9
Figure 2-1: Code calculating an average value from a given array.	11
Figure 2-2: Running time for each growth rate.	15
Figure 2-3: Code 1.....	16
Figure 2-4: Code 2.....	16
Figure 2-5: Growth rates of codes in Figure 2-3 and Figure 2-4.....	17
Figure 2-6: Big-Theta definition shown by graph.	21
Figure 2-7: Program with a conditional exit.	23
Figure 2-8: Growth rate of code with a conditional exit.	24
Figure 2-9: Code with consecutive loops.	25
Figure 2-10: Code with alternative paths of execution.	26
Figure 2-11: Code with recursive calls.	27
Figure 2-12: Iterative version of code in Figure 2-11.....	28
Figure 2-13: Starting condition for our binary search example.....	29
Figure 2-14: Finding the middle data for the 1 st time in binary search.	30
Figure 2-15: Finding the middle data of the right half of the array in our binary search.	30
Figure 2-16: The required data found by our binary search.....	31
Figure 2-17: Code for binary search on an array.	32
Figure 2-18: Program that finds the greatest common divisor.....	34
Figure 2-19: Values of each variable in each iteration of the code in Figure 2-18.....	35
Figure 2-20: Proof of claim - If $a > b$ then $(a \% b) < a/2$	36
Figure 2-21: Code for calculating x^n	38
Figure 2-22: Code for calculating x^n , written recursively.	39
Figure 2-23: Calculating the largest gap between 2 values in a given array, the exhaustive approach.	40
Figure 2-24: Calculating the largest gap between 2 values in a given array, using the maximum and minimum value.	40
Figure 2-25: Finding the position of x in an array.....	42
Figure 3-1: List implementation using array.	50

Figure 3-2: Method <i>insert</i> for linked list implemented using array.	52
Figure 3-3: Inserting new data into array.	53
Figure 3-4: Method <i>remove</i> of List implemented by array.	54
Figure 3-5: Removing data from an array.	55
Figure 3-6: Method <i>head</i> , <i>tail</i> , and <i>append</i>	56
Figure 3-7: Linked list structure concept.	58
Figure 3-8: Removing data from a linked list.	59
Figure 3-9: Inserting data into a linked list.	60
Figure 3-10: Inserting new data into the first position.	61
Figure 3-11: A linked list with a header node.	61
Figure 3-12: An empty linked list with a header node.	62
Figure 3-13: Implementation of a node that stores an integer.	62
Figure 3-14: A node created from ListNode <code>a = new ListNode(5);</code>	63
Figure 3-15: Simple markers <i>a</i> , <i>b</i> , and <i>c</i> at positions of interest.	63
Figure 3-16: interface <i>Iterator</i>	65
Figure 3-17: Iterator for Linked List implementation.	66
Figure 3-18: List iterator focusing on list header.	66
Figure 3-19: State of List iterator that method <i>hasNext</i> returns false.	67
Figure 3-20: The working of method <i>next</i>	68
Figure 3-21: Linked List implementation (<i>constructor</i> , <i>find</i> , and <i>findKth</i>).	69
Figure 3-22: Execution steps of method <i>find</i>	70
Figure 3-23: <i>insert</i> method of <i>LinkedList</i>	71
Figure 3-24: Inserting a new value at the start of the list (after the header).	72
Figure 3-25: Inserting after the last data in a list.	72
Figure 3-26: <i>remove</i> method of <i>LinkedList</i>	73
Figure 3-27: Status of variables when <i>findPrevious</i> is called on an empty list.	74
Figure 3-28: Status of variables in each step of execution when <i>findPrevious</i> is called on a list that stores <i>value</i>	75
Figure 3-29: Final status of variables when <i>findPrevious</i> is called on a list that does not store <i>value</i>	76
Figure 3-30: Method <i>head</i> and <i>tail</i> of <i>LinkedList</i>	77
Figure 3-31: Method <i>append</i> of <i>LinkedList</i>	78
Figure 3-32: Example of a circular doubly-linked list.	80
Figure 3-33: Code of a node of a doubly-linked list.	81

Figure 3-34: A DListNode created by its default constructor.....	81
Figure 3-35: Iterator that can traverse a data structure in two directions.	82
Figure 3-36: Bi-directional linked list iterator.....	83
Figure 3-37: List iterator creation on a doubly-linked list.	84
Figure 3-38: Circular doubly-linked list variables, constructor, and small utility methods.....	85
Figure 3-39: Making an empty list with method <i>makeEmpty</i>	86
Figure 3-40: Method <i>find</i> of circular doubly-linked list.	87
Figure 3-41: Method <i>findKth</i> of circular doubly-linked list.	88
Figure 3-42: Method <i>insert</i> of circular doubly-linked list.	89
Figure 3-43: Execution steps of insert for doubly-linked list.....	91
Figure 3-44: Method <i>remove</i> of doubly-linked list.	92
Figure 3-45: Method <i>findPrevious</i> and <i>remove(Iterator p)</i> of doubly-linked list.....	93
Figure 3-46: The working of <i>findPrevious</i> for doubly-linked list.	94
Figure 3-47: How <i>remove(Iterator p)</i> operates in doubly-linked list.....	95
Figure 3-48: Code for removing data at a specified position.	96
Figure 3-49: Two-dimensional array representing games and players' progresses.....	98
Figure 3-50: Linked list implementation of a sparse table.....	99
Figure 3-51: A skip list (shown with 1 direction pointers only in order to avoid confusion).....	100
Figure 3-52: Doubly-linked skip list with 5 data.	101
Figure 3-53: Sample code for a skip list node.	103
Figure 4-1: Stack with 3 data inside.....	109
Figure 4-2: Pushing data <i>d</i> onto a stack.....	110
Figure 4-3: Popping data out of a stack.....	110
Figure 4-4: Processing brackets, with excess closing brackets.	111
Figure 4-5: Processing brackets, with incorrect type pairing.	112
Figure 4-6: Processing brackets, with excess opening bracket.	113
Figure 4-7: Example method calls.....	114
Figure 4-8: Pseudocode for Infix to Postfix Transformation.	126
Figure 4-9: Pseudocode for Infix to Postfix Transformation, after adding inside-outside stack priorities.....	133
Figure 4-10: Stack Operations (interface).....	136

Figure 4-11: Stack implemented by array (fields, constructors, get, set).	137
Figure 4-12: <i>isEmpty()</i> , <i>isFull()</i> , and <i>makeEmpty()</i> of stack implemented with array.....	138
Figure 4-13: Code of method <i>top</i> , for stack implemented with array..	139
Figure 4-14: <i>top</i> and <i>bottom</i> of stack of size 3 (implemented with array).	139
Figure 4-15: Code of method <i>pop</i> , for stack implemented with array.	140
Figure 4-16: Popping data from a stack implemented with array.	140
Figure 4-17: Code of method <i>push</i> , for stack implemented with array.	141
Figure 4-18: Pushing data onto stack implemented with array.....	141
Figure 4-19: Linked list used as stack.	142
Figure 4-20: Code for stack implemented with circular doubly-linked list (fields and constructors).	143
Figure 4-21: <i>isEmpty()</i> , <i>isFull()</i> , and <i>makeEmpty()</i> for stack implemented with circular doubly-linked list.....	144
Figure 4-22: <i>top()</i> for stack implemented with circular doubly-linked list.	144
Figure 4-23: <i>pop()</i> for stack implemented with circular doubly-linked list.	145
Figure 4-24: Removing the top of stack in linked list implementation.	145
Figure 4-25: Method <i>push</i> for stack implemented with circular doubly- linked list.....	146
Figure 4-26: Pushing new data onto stack implemented with circular doubly-linked list.	146
Figure 5-1: Queueing for services.....	155
Figure 5-2: Dequeueing the first data from a queue.	156
Figure 5-3: Enqueueing a new data.....	157
Figure 5-4: Interface for queue storing integer data.	157
Figure 5-5: Dequeue for array implementation.....	159
Figure 5-6: Enqueue for array implementation.....	160
Figure 5-7: Incrementing <i>front</i> that goes back to the first array slot when dequeueing.	161
Figure 5-8: <i>front+size</i> that goes back to the first array slot when enqueueing.	162

Figure 5-9: fields, constructors, and methods that check for <i>size</i> in the array implementation of queue.	163
Figure 5-10: Code for <i>front()</i> in array implementation of queue.	164
Figure 5-11: Code for <i>back()</i> in array implementation of queue.	164
Figure 5-12: Code for <i>removeFirst()</i> in array implementation of queue.	165
Figure 5-13: Code for method <i>insertLast</i> in array implementation of queue.	166
Figure 5-14: Using a circular doubly-linked list to represent a queue.	167
Figure 5-15: Code for field, constructors, <i>isEmpty()</i> , <i>isFull()</i> , <i>size()</i> of linked list implementation of queue.	168
Figure 5-16: Code for <i>front()</i> of linked list implementation of queue.	168
Figure 5-17: Code for <i>back()</i> of linked list implementation of queue.	169
Figure 5-18: Identifying the last data in linked list implementation of queue.	169
Figure 5-19: Code for <i>removeFirst()</i> of linked list implementation of queue.	170
Figure 5-20: Removing the first data in linked list implementation of queue.	171
Figure 5-21: Code for <i>insertLast()</i> of linked list implementation of queue.	171
Figure 5-22: Adding a new data to linked list implementation of queue.	172
Figure 5-23: Illustrated concept of <i>removeLast()</i>	173
Figure 5-24: Illustrated concept of <i>insertFirst(data)</i>	173
Figure 5-25: Java interface for double-ended queue.	174
Figure 5-26: Double-ended queue implementation using array.	174
Figure 5-27: Reducing <i>size</i> without changing <i>front</i> in array implementation of double-ended queue.	175
Figure 5-28: Operations inside <i>insertFirst(77)</i> for array implementation of double-ended queue.	176
Figure 5-29: Linked list implementation of double-ended queue.	177
Figure 5-30: Operations inside <i>removeLast()</i> for linked list implementation of double-ended queue.	178
Figure 5-31: Operations of <i>insertFirst(9)</i> for linked list implementation of double-ended queue.	179

Figure 5-32: Step a), getting numbers into queues, when the least significant digit is the sorting identifier.	181
Figure 5-33: Step b), getting all numbers back to the array.....	182
Figure 5-34: Step a), when using the second digit from the right as a sorting identifier.	182
Figure 5-35: Step b), when using the second digit from the right as a sorting identifier.	183
Figure 5-36: Step a), when using the third digit from the right as a sorting identifier.	183
Figure 5-37: Step b), when using the third digit from the right as a sorting identifier.	184
Figure 5-38: Radix sort implementation (part 1).	184
Figure 5-39: Radix sort implementation (part 2).	185
Figure 6-1: A Binary Tree.....	195
Figure 6-2: Node levels in a tree.....	198
Figure 6-3: A perfectly balanced tree.	198
Figure 6-4: Examples of non-complete/complete binary trees.....	199
Figure 6-5: Searching for the number 4 in a Binary Search Tree.....	205
Figure 6-6: A node implementation concept of a binary tree.....	206
Figure 6-7: Tree from Figure 6-5, utilizing our implementation idea..	207
Figure 6-8: Code for binary search tree node.	208
Figure 6-9: Creating a node by using one parameter constructor.	208
Figure 6-10: Node visiting sequence.....	209
Figure 6-11: Code for tree iterator field and constructor.....	210
Figure 6-12: Method <i>hasNext</i> of class <i>TreeIterator</i>	210
Figure 6-13: Immediate parent contains a larger value.	212
Figure 6-14: Movement of <i>p</i> and <i>temp</i> when the larger value is in some ancestor node.....	212
Figure 6-15: Movement of <i>p</i> and <i>temp</i> when a node with larger value does not exist.	213
Figure 6-16: Method <i>hasPrevious</i> of class <i>TreeIterator</i>	214
Figure 6-17: Code for method <i>next</i> of class <i>TreeIterator</i>	215
Figure 6-18: Finding node <i>Z</i> , with a value just larger than <i>X</i> , when our current node, <i>X</i> , has another node as its <i>right</i>	215
Figure 6-19: Code for method <i>previous</i> of class <i>TreeIterator</i>	217
Figure 6-20: Code for method <i>set</i> of class <i>TreeIterator</i>	217
Figure 6-21: Structure of a binary search tree (implementation).	218

Figure 6-22: Implementation of method <i>findMin</i> for a binary search tree.	219
Figure 6-23: Code for method <i>find</i> of binary search tree.	220
Figure 6-24: Adding new data, <i>v</i> , to an empty binary search tree.	222
Figure 6-25: Adding 6 to a binary search tree that does not originally store 6.	223
Figure 6-26: Code for inserting value <i>v</i> into a binary search tree.	224
Figure 6-27: Removing <i>v</i> when <i>v</i> is in a root with no children.	227
Figure 6-28: Removing <i>v</i> when <i>v</i> is in a node (not a root) with no children.....	227
Figure 6-29: Removing <i>v</i> when <i>v</i> is in a root with right child but no left child.	228
Figure 6-30: Removing <i>v</i> when the node, <i>n</i> , that stores <i>v</i> has only its right child, it is not the tree's root, and <i>n</i> stores a larger value than its parent.....	228
Figure 6-31: Removing <i>v</i> when the node, <i>n</i> , that stores <i>v</i> has only its right child, it is not the tree's root, and <i>n</i> stores a smaller value than its parent.....	229
Figure 6-32: Removing <i>v</i> when the node, <i>n</i> , that stores <i>v</i> has both left and right child.....	230
Figure 6-33: Code for method <i>remove</i> of a binary search tree (part 1).	231
Figure 6-34: Code for method <i>remove</i> of a binary search tree (part 2).	232
Figure 6-35: Instance variables and simple methods of a recursive binary search tree.	234
Figure 6-36: Method <i>findMin</i> of class <i>BSTRecursive</i>	234
Figure 6-37: Method <i>find</i> of class <i>BSTRecursive</i>	235
Figure 6-38: Method <i>insert</i> of class <i>BSTRecursive</i>	236
Figure 6-39: The tree is not modified properly if <i>n.right</i> is not used to store the result of <i>insert</i>	237
Figure 6-40: Incorrect use of method <i>insert</i>	238
Figure 6-41: How the code in Figure 6-40 works.	238
Figure 6-42: Correction of code from Figure 6-40.	239
Figure 6-43: Code for method <i>remove</i> of class <i>BSTRecursive</i>	240
Figure 6-44: Tree for use with all traversal examples.	241
Figure 6-45: code for preorder and inorder printing of data in a tree.	243
Figure 6-46: Search sequence by level of a tree.	244
Figure 6-47: Putting <i>root</i> into <i>thisLevel</i> queue.	245

Figure 6-48: Removing a node from <i>thisLevel</i> queue and put its <i>left</i> and <i>right</i> nodes into <i>nextLevel</i> queue.	245
Figure 6-49: Removing all nodes from <i>nextLevel</i> queue and putting them in <i>thisLevel</i> queue.	246
Figure 6-50: The queues after nodes with 1 and 8 are removed.	246
Figure 6-51: Removing all nodes from <i>nextLevel</i> queue and putting them in <i>thisLevel</i> queue for the 2 nd time.	246
Figure 7-1: A hash table example.	256
Figure 7-2: A function that transforms a string into integer.	258
Figure 7-3: A separate chaining hash table.	263
Figure 7-4: Fields, constructors, and utility methods for separate chaining hash table.	264
Figure 7-5: Method <i>hash</i> of separate chaining hash table.	266
Figure 7-6: Method <i>find</i> of separate chaining hash table.	267
Figure 7-7: Method <i>add</i> and <i>rehash</i> of separate chaining hash table. ...	268
Figure 7-8: Method <i>remove</i> of separate chaining hash table.	269
Figure 7-9: Putting 1, 11, and 3 into a hash table of size 7, where $hashx = x \% array\ size$	272
Figure 7-10: Putting 8 in a hash table from Figure 7-9.	273
Figure 7-11: Putting 9 in a hash table from Figure 7-10.	273
Figure 7-12: Removing 3 and then trying to search for 9, where $hashx = x \% array\ size$	274
Figure 7-13: Lazy deletion prevents premature stopping while searching for data.	276
Figure 7-14: Adding 8 and 9 to a quadratic probing hash table that already has 1, 3, and 11, with $hashx = x \% array\ size$	277
Figure 7-15: Adding 8, 15, 22 into a quadratic probing hash table that already has 1, with $hashx = x \% array\ size$	279
Figure 7-16: Adding 8, 15, 22 into a double hashing hash table that already has 1, with $hashx = x \% array\ size$ and $hash2x = 3 - (x \% 3)$	283
Figure 7-17: Fields, constructors, and utility methods for open addressing hash table.	284
Figure 7-18: Fields, constructors, hash functions, and method <i>find</i> of a double hashing implementation.	286
Figure 7-19: Method <i>add</i> of a double hashing implementation.	288
Figure 7-20: Method <i>rehash</i> of a double hashing implementation.	289

Figure 7-21: Method remove of a double hashing hash table.	289
Figure 8-1: Bubble Sort.	294
Figure 8-2: Code for bubble sort algorithm.	295
Figure 8-3: Selection sort.	297
Figure 8-4: Code for selection sort.	298
Figure 8-5: Insertion sort.	300
Figure 8-6: Code for insertion sort.	301
Figure 8-7: Merge sort concept.	302
Figure 8-8: Splitting array into 2 portions for sorting.	303
Figure 8-9: Combining 2 sorted arrays.	306
Figure 8-10: Code for merge sort.	307
Figure 8-11: Code for combining 2 sorted arrays into one.	307
Figure 8-12: Quick sort concept.	311
Figure 8-13: Bad pivot selection.	312
Figure 8-14: Code for median of 3.	313
Figure 8-15: Partitioning example (part 1).	315
Figure 8-16: Partitioning example (part 2).	316
Figure 8-17: Code for quick sort.	317
Figure 9-1: Priority queue operations.	332
Figure 9-2: Priority queue implemented by linked list (part 1).	335
Figure 9-3: Priority queue implemented by linked list (part 2).	336
Figure 9-4: A min heap example.	339
Figure 9-5: Array representation of heap in Figure 9-4.	339
Figure 9-6: Code for constructor, <i>isEmpty()</i> , and <i>size()</i> of class <i>Heap</i>	340
Figure 9-7: Each step for adding 30 into a heap, showing both the tree version and its array implementation.	342
Figure 9-8: Code for method <i>add</i> of class <i>Heap</i>	343
Figure 9-9: Code for method <i>top</i> of class <i>Heap</i>	345
Figure 9-10: Removing a root without using a special algorithm, destroying a complete binary tree structure.	345
Figure 9-11: Each step for removing the most important value from a heap, showing both the tree version and its array implementation. ...	347
Figure 9-12: Code for method <i>pop</i> of class <i>Heap</i>	348
Figure 9-13: A possible encoding for character 'a' to 'e' in a text file. .	352
Figure 9-14: Another possible tree for encoding 'a' to 'e'.	353
Figure 9-15: An Example of order of nodes to be retrieved from priority queue that stores nodes of Huffman tree.	355

Figure 9-16: Example of Huffman tree creation (first iteration).	356
Figure 9-17: Example of Huffman tree creation (second iteration).	357
Figure 9-18: Example of Huffman tree creation (third iteration).	358
Figure 9-19: Example of Huffman tree creation (fourth iteration).	359
Figure 10-1: Examples of AVL trees.	366
Figure 10-2: Examples of non-AVL Trees.	367
Figure 10-3: Rebalance, 1 st possible case.	368
Figure 10-4: Rebalance, 2 nd possible case.	369
Figure 10-5: Rebalance, 3 rd possible case.	370
Figure 10-6: Rebalance, 4 th case.	372
Figure 10-7: Code for a node of AVL tree.	373
Figure 10-8: Code for AVL Tree (part 1).	375
Figure 10-9: Code for AVL Tree (part 2).	376
Figure 10-10: Code for <i>rotateLeftChild</i> and <i>rotateRightChild</i>	377
Figure 10-11: Detailed operation of method <i>rotateLeftChild</i>	378
Figure 10-12: Code for method <i>rebalance</i>	379
Figure 10-13: Code for method <i>insert</i> of AVL tree.	380
Figure 10-14: Code for method <i>remove</i> of AVL tree.	381

List of Tables

Table 3-1: List operations	49
Table 3-2: Asymptotic runtime comparisons on operations of array and linked list.....	79
Table 4-1: Various stages of stack for storing methods data when code in Figure 4-7 is executed (part 1).	116
Table 4-2: Various stages of stack for storing methods data when code in Figure 4-7 is executed (part 2).	117
Table 4-3: Various stages of stack for storing methods data when code in Figure 4-7 is executed (part 3).	118
Table 4-4: Expressions and their corresponding postfix form.....	119
Table 4-5: Postfix calculation of 2+3.	120
Table 4-6: Calculation of $7\ 4\ 3\ * -$ (infix form is $7-4*3$).	121
Table 4-7: Calculation of $7\ 8\ 9\ * + 5 + 10\ *$ (infix form is $(7+(8*9)+5)*10$).	122
Table 4-8: Transforming 2+3 to its postfix counterpart.....	124
Table 4-9: Transforming $7-4*3$ to its postfix counterpart.	124
Table 4-10: Transforming $7+5-10$ to its postfix counterpart.....	125
Table 4-11: Operator Priority.	128
Table 4-12: Transforming $10*(5-2)$ into postfix form.....	129
Table 4-13: Transforming an expression with nested brackets into its postfix form (Part 1).	130
Table 4-14: Transforming an expression with nested brackets into its postfix form (Part 2).	131
Table 4-15: Transforming an expression with nested brackets into its postfix form (Part 3).	132
Table 4-16: Incorrect postfix transformation due to priorities forcing left association.....	134
Table 4-17: Operator Priority, with right associative operator '^'.....	135
Table 4-18: Correct postfix transformation after fixing right associative operator.	135
Table 7-1: Separate Chaining and Open Addressing Comparison.	290
Table 9-1: Average runtime for method <i>add</i> , <i>top</i> , and <i>pop</i> in linked list implementation and heap implementation of priority queue.	350

Chapter 1 : Data Structures and Our Program

Data structures are essential for coding. Programmers who know various types of data structures have more tools to choose from when they do their coding and they can program more efficient codes. But what is a data structure anyway?

What is a Data Structure?

A data structure is a piece of data that can store more than one instance of other data. Usually, all data stored in a data structure have the same type, but this is not an absolute requirement.

A data structure you probably already know is array. We can use array to store several information of the same data type. For example, you may declare:

```
int a = new int[5];
```

as an array that provides 5 slots for storing integer data.

With array, we can work on several data of the same type. So, if we know how to use array, we also know the basics of how to work with a data structure.

What does this book going to tell you about data structures? Well, array is just one kind of data structures. We need to know about other data structures too because they have different uses (and you will see). This book assumes that you are familiar with basic structured programming, including the use of array.

Data structures used in programming today are based on what are taught in this book. Therefore, once you have learned data structures from this book, you will have an easier time understanding other people's codes, as well as be able to apply these data structures in your own programs. You will not need to reinvent the wheel. Many programming languages have versions of these basic data structures available in their libraries. Understanding the basic data structures will also help you understand those built-in data structures.

In a program, you may be able to choose a data structure to store your data. Sometimes it does not really matter which data structure you choose. But many times, it matters! One data structure may allow faster data retrieval than others. Let's look at an example.

Let's say we want to store 5 integers. There are several data structures that can be used:

- Of course, our first data structure is an array.
- The second possible choice is a linked list, which is usually organized as a sequence of slots linked with one another using special links called "pointers". Figure 1-1 shows an example linked list that we can start access its contents through its first pointer "p" (don't worry too much about it. We will be covering it in detail in its own chapter).

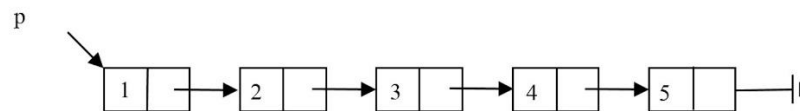


Figure 1-1: Linked list storing 5 integers.

- Our third possible choice is storing data in a tree, as shown in Figure 1-2. The tree shown here is a binary search tree (again, a chapter will be dedicated to it).

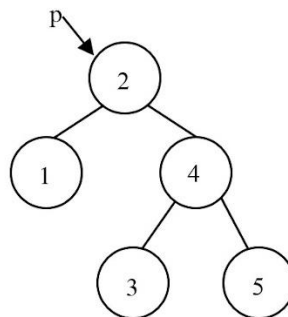


Figure 1-2: Binary Search Tree that stores 5 integers.

- Another data structure that can be used is a heap, which is a type of trees. Figure 1-3 shows a maxheap, which stores larger data at the top.

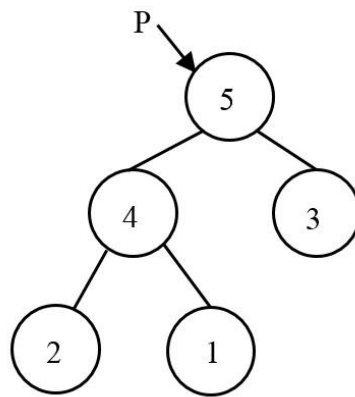


Figure 1-3: Maxheap storing 5 integers.

So, which one do we choose in our implementation?



Of course, it depends on how we want to use the data. If our program makes use of the smallest data much more often than others, then the choice would be the linked list because the smallest data can be accessed directly from pointer “p”. A sorted array can easily access its smallest data too, but if the smallest data is going to be regularly deleted, it will need a lot of time, especially in a large array, moving other data to the left.

But what if our program will be using the largest data most of the time? To reach our largest data in the linked list, we would have to follow the pointer “p” for several steps, wasting a lot of time especially if the list is long (see Figure 1-4).

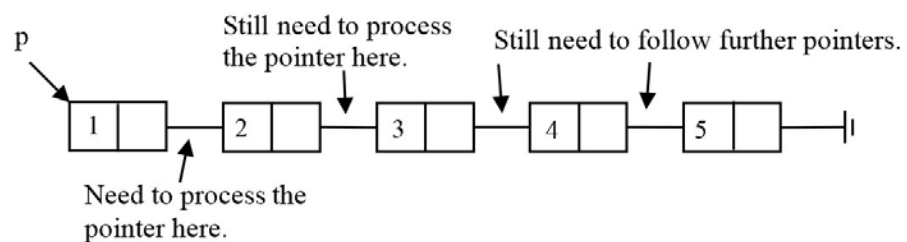


Figure 1-4: Following pointers to the last data in our linked list.

To reach the largest data in our binary search tree (Figure 1-2), we follow a right pointer from each node until we can follow no further. The number of data nodes needed to be investigated is lower than when we search our linked list.

It is even faster for our heap since the maximum value is already at the data node pointed to by “p”.

As you can see, choosing a different data structure affects the speed of your program. That is why we need to know the basics.

This book’s organization

This book is intended to give you information you need for an introduction to data structures and algorithms course. All example codes in this book are in Java language but the codes are organized such that they can easily be applied in other programming languages. The chapters in this book are as follows:

- **Chapter 2** introduces common terms used in data structures and the analysis of algorithms, such as asymptotic runtime.
- **Chapter 3** takes you through our first new data structure: **Linked list**, including its implementation.
- **Chapter 4** covers the concept and implementation of **stack**.
- **Chapter 5** establishes the concept and walks you through how to implement **queue**.

- **Chapter 6** is all about **binary tree** and **binary search tree**. These are the first non-sequential data storage that you will encounter.
- **Chapter 7** is all about **hash table**, a data structure that allows data retrieval to take constant time on average.
- **Chapter 8** covers sorting algorithms and their complexity. The algorithms introduced in this chapter operate on arrays, but can be applied on data structures as well.
- **Chapter 9** gets you familiar with **priority queue** and its major implementation, **heap**.
- **Chapter 10** introduces an **AVL tree**, which is one of the approaches we can use to maintain a balanced binary search tree.

All readers should read chapter 2 first in order to get to know the terms used throughout this book.

Linked list, stack, and queue are used to store data sequence. They only differ in ways data can be accessed. Stack and queue can be thought of as linked lists with special restrictions. Building blocks we used to implement a linked list can be used to implement stacks and queues too. Therefore, **it is recommended that chapter 3 be read before chapter 4 and chapter 5.**

Both our linked list and binary search tree (including AVL tree) implementations need pointers. Linked list pointers are easier to understand therefore **it is recommended that you read chapter 3 before chapter 6.**

Hash table implementations use both linked list and array. Therefore, **to fully understand chapter 7, it is recommended that you read chapter 3 first.**

Chapter 8 can be read independently of other chapters (except chapter 2).

Heap in chapter 9 uses the concept of binary tree but the actual implementation uses an array. **It is recommended that you read chapter 6 (the binary tree part) before chapter 9.**

Chapter 10 should be read after chapter 6 since AVL tree is a special form of binary search tree.

Figure 1-5 shows our book's organization.

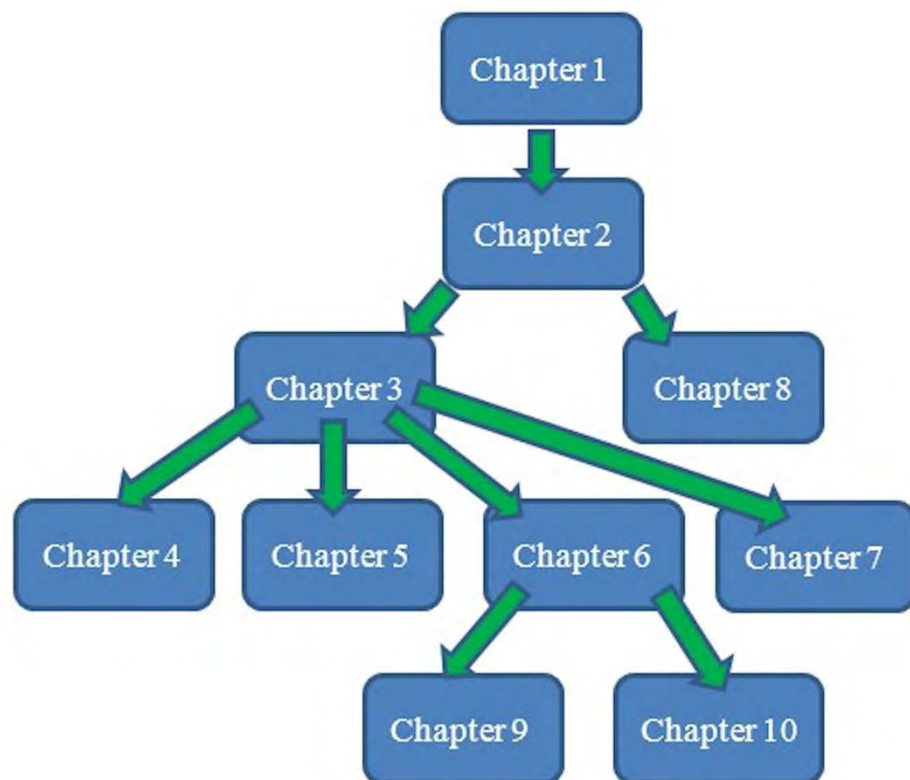


Figure 1-5: Book organization.

Without further ado, let's start!



Chapter 2 : Introduction to Algorithm Analysis

In this chapter, we will be looking at how we can estimate the speed of our programs. We will also see notations commonly used among programmers when referring to program speed.

Running Time Analysis

Let's analyze the code in Figure 2-1 that calculates the average value of all data in a given array.

```
double average(int[] a)
{
1:     int n = a.length;
2:     double tempSum = 0;
3:     for (int i=0;i<n;i++)
4:         tempSum += a[i];
5:     return tempSum/n;
}
```

Figure 2-1: Code calculating an average value from a given array.

If we are to estimate the running time of the above code, we have the following alternatives:

- Estimate the running time of each component of the code and add them all up.
- Choose a representative statement of the code and estimate the running time of that statement only.

Let us try estimating the running time of code components and adding them up. Let's look at the code (Figure 2-1) line-by-line.

Line 1: Variable declaration and assignment. Let a variable declaration consumes 1 unit of time and an assignment consumes 1 unit of time also. Therefore, this line of code takes 2 units of time to run.

Line 2: Variable declaration and assignment also. The estimated time is also 2 units.

Line 3: This involves quite a few operations:

- Declaration and initialization of variable "i". Both are performed only once. So, the time is 2 units (1 for declaration and the other unit is for initialization).
- Conditional testing of "i". The first test takes place when the value of "i" is 0. The last test takes place when the value of "i" is equal to n. If we let each conditional testing consume 1 unit of time, this part of the program consumes $n+1$ units of time.
- Increment the value of "i" by 1. This part of the code is executed every time before starting the next iteration. The first time takes place when the value of "i" is 0. The last time this code gets executed is when the value of "i" is $n-1$. If an increment operation takes 1 unit of time, then all increments take n units of time.

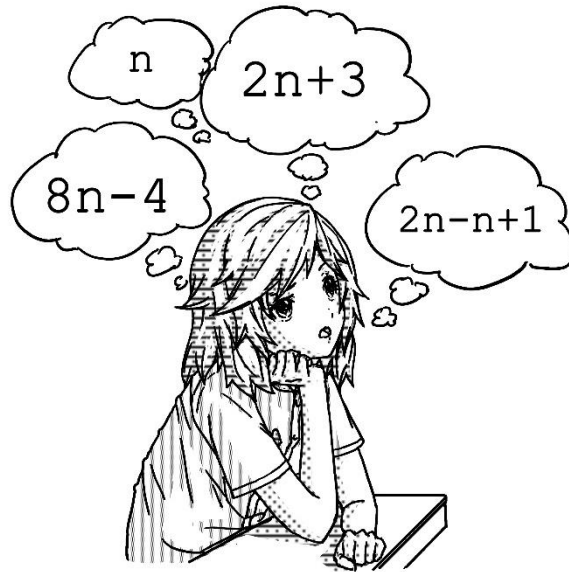
From the analysis above, the total time (our estimated unit time) for the execution of line 3 is $2+(n+1)+n$, which is equal to $2n+3$.

Line 4 contains an addition and an assignment. If each action takes 1 unit of time, we therefore have 2 units. But that is not all. This line of code is within a loop, which iterates n times. Therefore, the total unit time is $2n$.

Line 5 contains a division and a return statement. If each takes 1 unit of time, then overall it takes 2 units of time to execute.

If we add the estimated running time of all 5 lines together, we get $2+2+(2n+3)+2n+2$, which is $4n+9$ units of time, where n is the size of our input array.

This method of running time estimation can be used to compare estimated running time of 2 programs. However, it is obviously impractical because we need to work out estimated time for every line of code. A better method is discussed below.



Now, let's try the method that chooses a representative statement of the code and estimates the running time of that statement only.

Let's choose line 4 as the program's representative since it runs in a loop and therefore contributes significantly to the running time. Line 3 runs with the same loop but looks much more complex so it is not chosen. With line 4 chosen, we have the estimated running time of $2n$. You can see that it is much easier to obtain compared to the first method.

This running time, in terms of n , grows with the size of data. The growth pattern is called a **growth rate**.

Programs can have different growth rates such as n , n^2 , $\log n$, 2^n depending on how you write them (they may have nested loops or they may be able to eliminate data by half at each iteration, etc.). The larger the growth rate, the longer the program runs, especially when you have larger data size. Growth rate is more important than actual running time when it comes to comparing program performances.

Here is how the running time of each growth rate looks like as the data size grows. The x-axis indicates data size, while the y-axis indicates the running time.

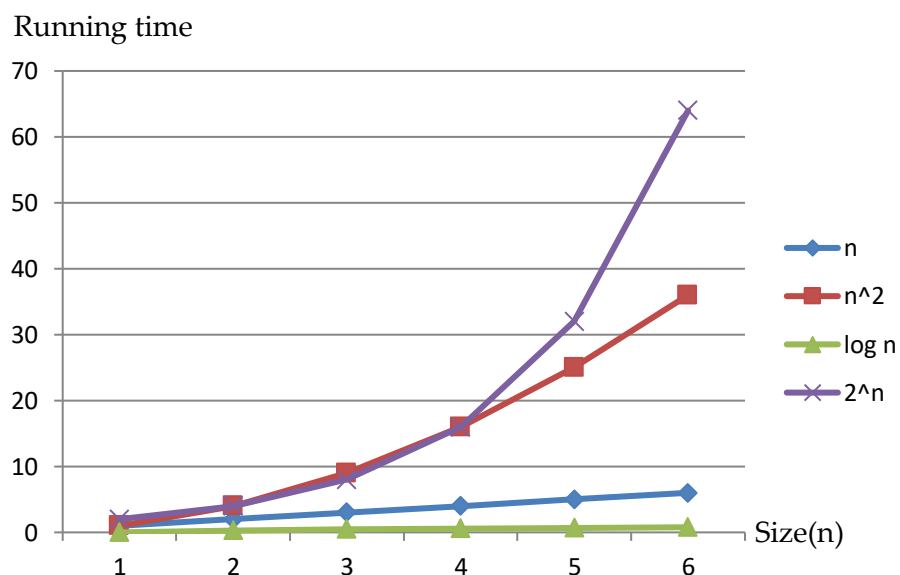


Figure 2-2: Running time for each growth rate.

You can see that for large n , the running time for each growth rate can be compared as follows:

$$\log n < n < n^2 < 2^n$$

Therefore, knowing a program's growth rate means we know its performance. We can also compare programs performances by comparing their growth rates.

Let's look at the following Code 1 and Code 2 (Figure 2-3 and Figure 2-4) for two different programs.

```
for(int i=1; i<=n; i++){
    for(int j=1; j<= n; j++){
        x= x+1;
    }
}
```

Figure 2-3: Code 1.

```
for(int i=1; i<=n; i++){
    for(int j=5; j<= n; j++){
        x= x+1;
    }
}
```

Figure 2-4: Code 2.

If we use the line that has $x= x+1$ as our code representative, Code 1 will take the following time to run:

$$\sum_{i=1}^n \sum_{j=1}^n 1 = n^2 \quad (1)$$

And Code 2 will take the following time to run:

$$\sum_{i=1}^n \sum_{j=5}^n 1 = n(n - 4) = n^2 - 4n \quad (2)$$

Although their running time is different, both codes have the same growth rate. Let's look at their running time as n (x-axis) increases.

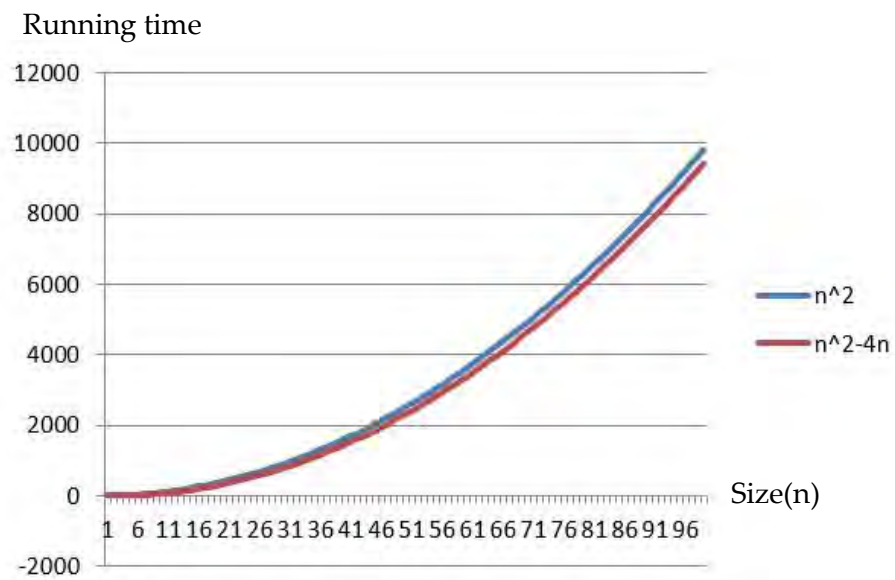


Figure 2-5: Growth rates of codes in Figure 2-3 and Figure 2-4.

It is straightforward to see that their running time (y-axis) increase with the same rate. Therefore, we can regard their performances to be equal.

But comparing 2 programs' growth rates may not be as simple as in the above example. The following definition helps you do the comparison without even having to draw a graph.

Definition 2-1: Faster / Slower growth rates

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ means $f(n)$ grows slower than $g(n)$.

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ means $f(n)$ grows faster than $g(n)$.

$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \text{constant value}$ means $f(n)$ grows with the same rate as $g(n)$, or $f(n) \in \Theta(g(n))$. Θ is one of asymptotic notations. We will look at asymptotic notations more closely in the next topic.

Instead of comparing graphs, the values of functions when n is large can be compared.

With this definition, the following running times have the same growth rate (look at their most significant term!):

$$n^2, 0.001n^2, 3n^2 - 10n, 500n^2 + 5000n + 50000$$

And the following growth rates are shown, from slow to fast, according to the above definition:

$$\log n, n, n \log n, n^2, n^3, 2^n, n^n$$

The definition helps us compare growth rates in a more difficult case, such as comparing $\log n$ and \sqrt{n} . By just looking at them, it is not obvious which one is slower. Let's try to compare them using the definition. Let $f(n) = \log n$ and $g(n) = \sqrt{n}$.

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} \\ &= \lim_{n \rightarrow \infty} \frac{\ln n}{\ln 10 \sqrt{n}} \\ &= \frac{1}{\ln 10} \lim_{n \rightarrow \infty} \frac{\ln n}{\sqrt{n}} \\ &= \frac{1}{\ln 10} \lim_{n \rightarrow \infty} \frac{1/n}{1/(2\sqrt{n})} \\ &= \frac{1}{\ln 10} \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} \\ &= 0 \end{aligned}$$

Therefore $\log n$ grows slower than $n^{0.5}$. In fact, we can use this definition to show that $\forall c, k > 0$, $(\log n)^c$ always grow slower than n^k . Because $\frac{\log n}{\sqrt{n}}$ is a factor of $\frac{(\log n)^c}{n^k}$, hence $\lim_{n \rightarrow \infty} \frac{(\log n)^c}{n^k} = 0$.

There are some common notations used by computing people when they talk about running times of programs. These notations are collectively called asymptotic notations. For a computer science/ engineering student, it is crucial to understand these to communicate effectively with your co-workers and supervisors.

Asymptotic Notation

These notations are used to display growth rates. In this book, we will mainly focus on two most often used notations: the big-Theta and big-O.

Definition 2-2: Big-Theta, or Θ

$\Theta(g(n))$ is a set of functions that grow with the same rate as $g(n)$. We can define it mathematically using limit, as done in the last section. There is an alternative definition also. Let's have a look at the alternative definition.

$$\Theta(g(n)) = \{f(n) | \exists_{c_1} \exists_{c_2} \exists_{n_0} c_1 > 0 \wedge c_2 > 0 \wedge n_0 \geq 0 \wedge c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ where } n \geq n_0\}$$

Basically, the definition says that $f(n) \in \Theta(g(n))$ if and only if $f(n)$ is within the bound of $c_1 g(n)$ and $c_2 g(n)$ for all n greater than a certain value.

Time for an example! Let our program's running time be $f(n) = 5n^2 + 10n + 18$. Let $g(n) = n^2$. If we set c_1 to 1 and c_2 to 8, the graph of their values is shown in Figure 2-6.

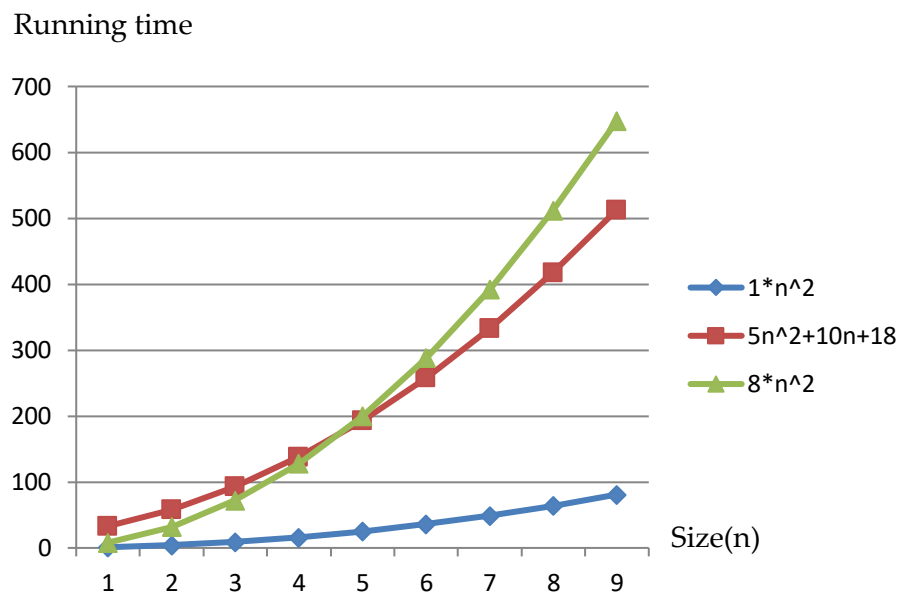


Figure 2-6: Big-Theta definition shown by graph.

The values of $f(n)$ eventually lie between $1 \cdot n^2$ and $8 \cdot n^2$. Therefore $f(n)$ has the same growth rate as n^2 , or $f(n) \in \Theta(n^2)$, according to Definition 2-2. There are many possible values for c_1 and c_2 that make $f(n) \in \Theta(n^2)$. Just finding one pair that works is enough.

Another definition that you will encounter a lot is big-O.

Definition 2-3: Big-O, or O

$O(g(n))$ is a set of functions that grow slower, or with the same rate as $g(n)$. Its formal definition is given below.

$$O(g(n)) = \{f(n) | \exists c \exists n_0 c > 0 \wedge n_0 \geq 0 \wedge f(n) \leq cg(n), \text{ where } n \geq n_0\}$$

Basically, the definition says that $f(n) \in O(g(n))$ if and only if $f(n)$ has lesser or equal value to $cg(n)$ for all n greater than a certain value.

Looking at this definition and the previous example, we can see that $f(n) = 5n^2 + 10n + 18 \in O(n^2)$ since $f(n) \leq 8n^2$ for $n = 5$ onwards. This means programs that satisfy $\theta(g(n))$ also satisfy $O(g(n))$ (but not vice versa).

Asymptotic Notations and Nested Loop

You may be wondering about how these notations get used in real programs. Let us recall the earlier example code from Figure 2-3 (the code is shown below for your convenience).

```
for(int i=1; i<=n; i++){
    for(int j=1; j<= n; j++){
        x= x+1;
    }
}
```

We already know that the running time is $n^2 - 4n$ and its growth rate is the same as n^2 .

Now, knowing asymptotic notations, we can write down the code's performance in terms of asymptotic runtime, that is $\theta(n^2)$. The running time also satisfies $O(n^2)$.

Seeing a program, one can look at its representative statement (the one that gets run most often. For example, a loop or a recursive call) and write down its estimated runtime in asymptotic form. It is fast, convenient, and easily comparable with other programs (and mathematically usable too!).

Now, let's have a look at a slightly modified code in Figure 2-7. The code has a conditional exit.

```
for(int i=1; i<=n; i++){
    for(int j=5; j<= n; j++){
        if(f(i)) return;
        x= x+1; // a representative statement
    }
}
```

Figure 2-7: Program with a conditional exit.

The number of times a representative statement gets to run can be from 0 to $n^2 - 4n$, depending on the value of $f(i)$. A graph showing the runtime growth of this code is shown in Figure 2-8 (one possibility is shown).

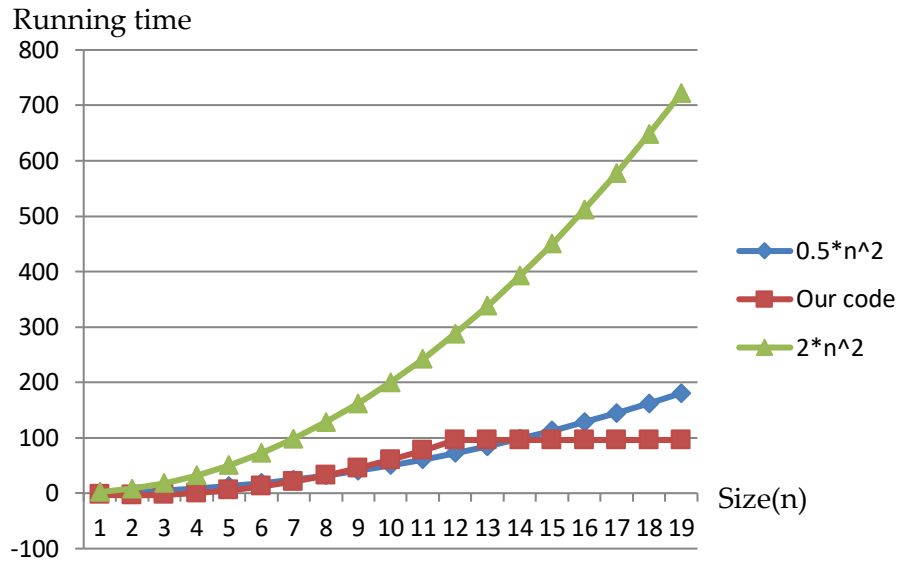


Figure 2-8: Growth rate of code with a conditional exit.

From Figure 2-8, there is a point where the running time no longer grows with n . So, its runtime no longer satisfies $\theta(n^2)$. It still satisfies $O(n^2)$ though. Many programs have exit conditions like this and their runtime hence must be indicated using big-O rather than big-Theta.

Here is a definition for an asymptotic runtime of nested loops (the definition works for big-O and big-Theta):

Definition 2-4: Asymptotic runtime of nested loops.

If loop T_1 has its asymptotic runtime = $O(f(n))$ and loop T_2 has its asymptotic runtime = $O(g(n))$, then a nested loop formed from these two loops has its asymptotic runtime = $O(f(n) * g(n))$.

Asymptotic Runtime and Consecutive Operations

You have already seen that for a nested loop, its asymptotic runtime is the multiplication of each layer's runtime. What about programs with consecutive chunks of instructions, for example:

```
for(int i =1; i<=n; i++) //first loop
    y = y*2;
}

for(int i=1; i<=n; i++){ //second loop
    for(int j=5; j<= n; j++){
        if(f(i)) break;
        x= x+1;
    }
}
```

Figure 2-9: Code with consecutive loops.

From Figure 2-9, the first loop has its asymptotic runtime equals to $\theta(n)$ while the second loop has its asymptotic runtime equals to $O(n^2)$.

The total runtime is $\theta(n) + O(n^2)$. But when n becomes large, the runtime growth from $O(n^2)$ will totally dominate the runtime growth from $\theta(n)$. Therefore, the asymptotic runtime that we can write down is just $O(n^2)$.

Definition 2-5: Asymptotic runtime for consecutive operations (the definition works for big-O and big-Theta).

If code segment T_1 has its asymptotic runtime = $O(f(n))$ and code segment T_2 has its asymptotic runtime = $O(g(n))$, then a code segment formed from by putting T_1 followed by T_2 has its asymptotic runtime = $\max(O(f(n)), O(g(n)))$.

Asymptotic Runtime and Conditional Operations

What about alternative statements? What should we use as their asymptotic runtime? Let's look at the following example together:

```
1:   if (condition)
2:       Statement1
3:   Else
4:       Statement2
```

Figure 2-10: Code with alternative paths of execution.

This code in Figure 2-10 either executes Statement 1 or Statement 2, never both. Since we do not know which statement will get executed at runtime, the running time we should assume should be the worst-case scenario, that is, the most time-consuming statement.

Asymptotic Runtime and Recursion

A recursive program is a program or method that keeps calling itself. For each successive call, its input size reduces until a condition where it will not call itself again is satisfied.

Let us analyze recursive code in Figure 2-11:

```
1:  mymethod (int n){
2:      if (n == 1) {
3:          return 1;
4:      } else {
5:          return 2*mymethod(n - 1) + 1;
6:      }
7:  }
```

Figure 2-11: Code with recursive calls.

From the code, the input is originally n . It reduces by 1 each time *mymethod* is called. This is similar to executing a loop for about n times. Therefore, our asymptotic runtime for this code is $\theta(n)$.

In short, we analyze how many times the method is called repeatedly and use that number as our asymptotic runtime.

Indeed, if we transform this program into its iterative counterpart (its code is shown in Figure 2-12), its asymptotic runtime is still $\theta(n)$.

```
1:  mymethod (int n) {
2:      int result = 1;
3:      int i = n;
4:      while (i > 1) {
5:          result = 2 * result + 1;
6:          i = i-1;
7:      }
8:      return result;
9:  }
```

Figure 2-12: Iterative version of code in Figure 2-11.

Asymptotic Runtime in Logarithmic Form

Sometimes our programs run in logarithmic time. This usually happens when we can spend a constant time to divide a problem into equal parts (reading input data does not count because it is already $\theta(n)$).

Let's check out an example. Let's say we have an array that stores n positive integers from index 0 to $n-1$, and the elements are sorted from small to large. We want to find an index of a given value, x . If x is not in the array, our algorithm should return -1.

To find the index, we could straightforwardly start searching from the first element of the array and stop when we find x . But x can be anywhere, from the first element to the last, even not in the array. Therefore, the asymptotic runtime is $O(n)$.

But we know that the elements are sorted, so we can use a faster algorithm. This algorithm is called binary search. We start by looking at the middle element of the array. If it is less than x , it means x , if it is in the array at all, is on the right half of the array. On the other hand, if the middle value is more than x , we know that x , if it is in the array, is in the left half of the array. Once we know which half of the array to search, we can search that half of the array by starting with the middle element of that half, and so on.

Below (Figure 2-13) is an array we want to work on (of course we normally do not know all array contents).

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

Figure 2-13: Starting condition for our binary search example.

If we want to see if the value 7 is in this array, searching the array from the first element will require us to look into 7 array slots. Using binary search, however, only requires us to look into 3 array slots (we use integer division when calculating the index of the middle

element). The middle element we find is at index $(0+7)/2$, which is 3 (0 is the index of the first slot, while 7 is the index of the last slot.). The stored value at that array slot is 4 (see Figure 2-14).

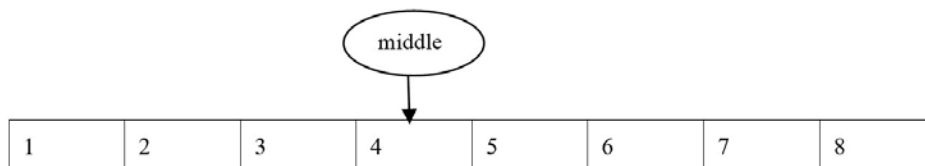


Figure 2-14: Finding the middle data for the 1st time in binary search.

Since our array is sorted from small to large, we immediately know that the value 7 must be on the right half of the array. So, we start searching by looking at the middle element of that half, which has index $(4+7)/2$, which is 5 (the first slot of the half has index value equals to 4). The stored value at that array slot is 6, as shown Figure 2-15.

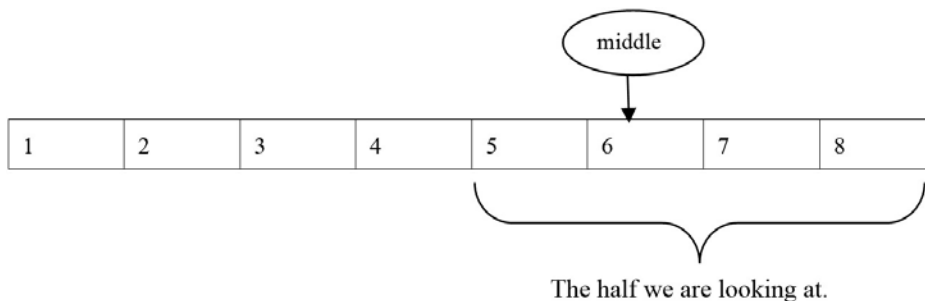


Figure 2-15: Finding the middle data of the right half of the array in our binary search.

Again, at this stage, we know that the value we are looking for (7) is on the right half of that array portion. So, we start searching by looking at the middle element of the portion, which has index $(6+7)/2$, which is 6 (the first slot of this portion has index value equals to 6). This time, we find the value we are looking for (see Figure 2-16). It can be seen that instead of looking at seven array slots, we only need to look at three of them, saving us half the time.

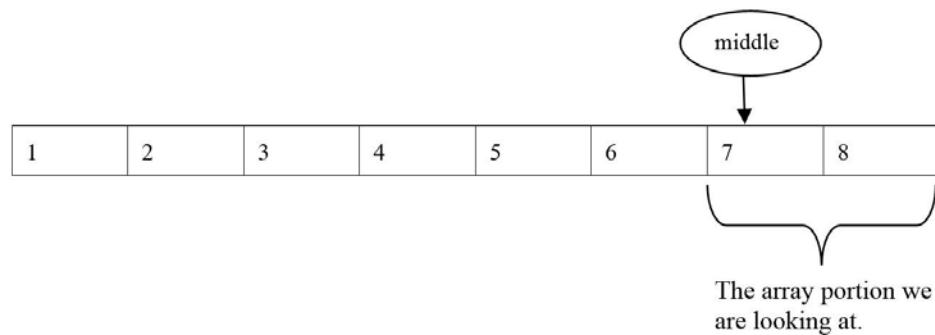


Figure 2-16: The required data found by our binary search.

Let us look at the program code for this binary search algorithm (see Figure 2-17).

The longest time that the code runs is when we cannot find the required number in the array. If the array size is n , the size of a portion that we need to investigate reduces by half each iteration.

```
1:  int binarySearch (int[] a, int x){
2:      int left =0, right =a.length - 1;
3:      while (left <=right) {
4:          int mid = (left + right)/2;
5:          if (a[mid] < x ) {
6:              left = mid + 1;
7:          } else if (a[mid] > x) {
8:              right = mid - 1;
9:          } else {
10:             return mid; //can exit early here.
11:          }
12:      }
13:      return -1;//number not found.
14:  }
```

Figure 2-17: Code for binary search on an array.

This means the number of iterations, i , is related to n in the following way:

$$n = 2^i + c, \text{ where } c \text{ is a constant.}$$

Applying logarithm on both sides, we get:

$$\log_2 n = \log_2 2^i + \log_2 c$$

$$\log_2 n = i + \log_2 c$$

$$i = \log_2 n - \log_2 c$$

In asymptotic form, the running time is therefore equal to $O(\log n)$ (big-Theta is not used because the program can exit early if the required data is found).

The reason we just use $\log n$ instead of $\log_2 n$ is because the base of the log is not important in its asymptotic form, as stated in the following definition.

Definition 2-6: Logarithmic asymptotic runtime

If a program has its running time equals to $f(n) = \log_a n$, then its asymptotic runtime is $O(\log_b n)$, where a and b are positive integers greater than 1.

The definition can be proven as follows:

Let the running time x , be logarithmic:

$$\log_a n = x \text{ and } \log_b n = y.$$

Hence, $n = a^x$ and $n = b^y$. Therefore, we get:

$$\ln n = x \ln a = y \ln b$$

$$x \ln a = y \ln b$$

$$\log_a n * \ln a = \log_b n * \ln b$$

$$\log_a n = \log_b n * \frac{\ln b}{\ln a}$$

$$\log_a n = \log_b n * c$$

$$\log_a n = O(\log_b n)$$

Therefore, the logarithmic asymptotic runtime can have so many possible bases, i.e. the bases do not matter.

Another example program that illustrates logarithmic asymptotic runtime is the program that finds the greatest common divisor (see Figure 2-18).

```
1:    long gcd (long m , long n) {
2:        while (n!=0) {
3:            long rem = m%n;
4:            m = n;
5:            n = rem;
6:        }
7:        return m;
8:    }
```

Figure 2-18: Program that finds the greatest common divisor.

From the program code, the value of n , which is the remainder, determines whether the program executes its next loop. How the remainder decreases will therefore decide our asymptotic runtime.

Let's run our program, with $m = 1974$ and $n = 1288$. The value of each variable in each loop is shown in Figure 2-19.

The value of n does decrease, but there seems to be no obvious pattern (Between the 1st and 2nd loop, it decreases only a little. But between the 2nd and the 3rd loop it seems to decrease a lot. And between the 3rd and the 4th loop, it decreases a little again).

1 st :	rem = 1974 mod 1288 = 686 m = 1288, n=686
2 nd :	rem = 1288 mod 686 = 602 m = 686, n = 602
3 rd :	rem = 686 mod 602 = 84 m = 602, n = 84
4 th :	rem = 602 mod 84 = 14 m = 84, n = 14
5 th :	rem = 84 mod 14 = 0 m = 14, n = 0

Figure 2-19: Values of each variable in each iteration of the code in Figure 2-18.

In order to determine the decrease speed of the remainder, we need to use the following fact:

$$\text{If } a > b \text{ then } (a \% b) < \frac{a}{2}$$

The above claim can be proven as follows:

- If $b \leq \frac{a}{2}$: since $a \% b < b$, therefore $(a \% b) < \frac{a}{2}$.
- If $b > \frac{a}{2}$: a/b will result in 1 and its remainder, which is $a - b$. Since we already know that $b > \frac{a}{2}$,

so the value of the remainder is $a - (> \frac{a}{2})$, which is less than $\frac{a}{2}$.

This proof is illustrated in Figure 2-20.

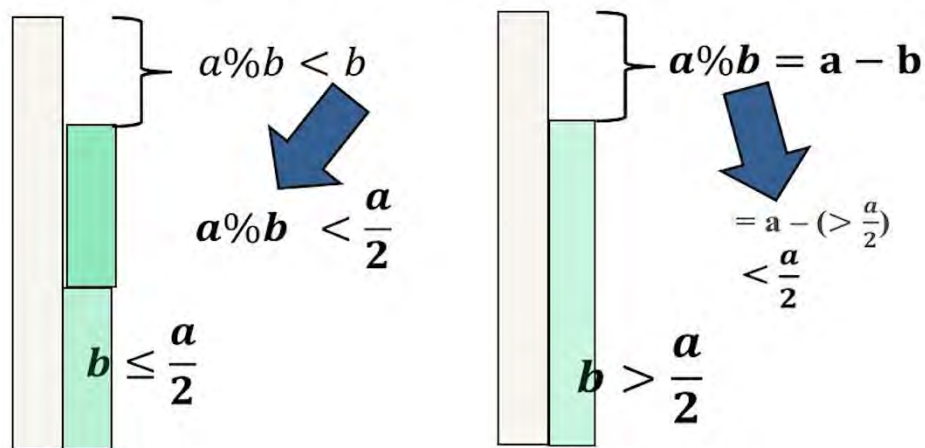


Figure 2-20: Proof of claim - If $a > b$ then $(a \% b) < \frac{a}{2}$.

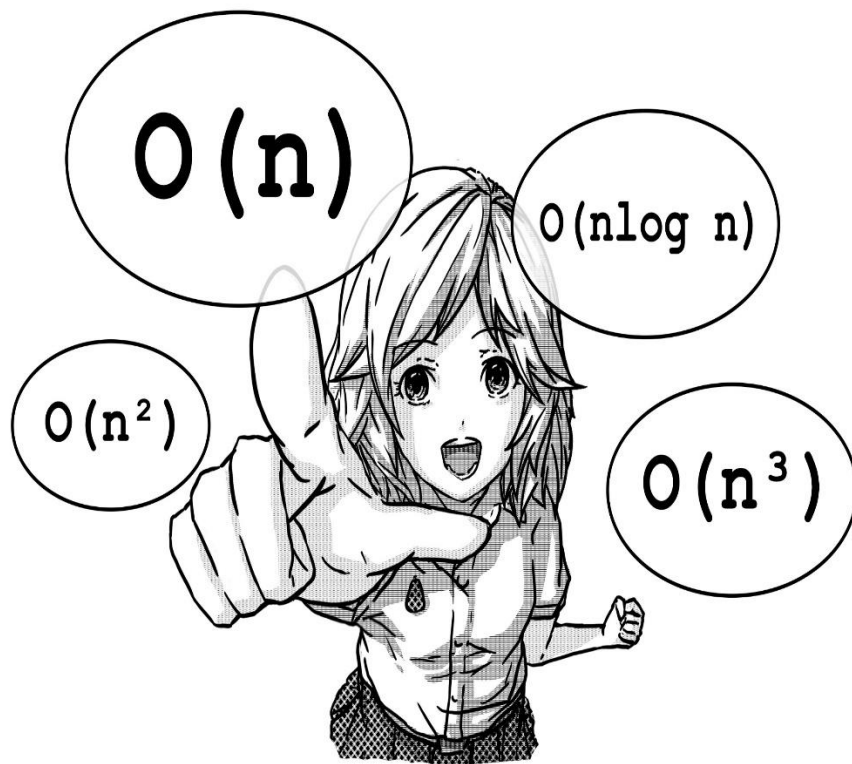
If we look at the value of each variable in each loop again, it can be seen that, starting from the third loop, the value of n in any of the loop comes from the modulo between the values of n in its previous two loops. For example, the value of n in the third loop (84) comes from the modulo between n in the first loop and n in the second loop.

It means that the value of n reduces by at least half in every two iterations. This is similar to binary search but takes twice as long so the number of iterations is in terms

of $2 * \log_2 n$. Thus, its asymptotic runtime is also $\theta(\log n)$.

Before we move on to a new topic, let us sort growth rates of programs from small to large:

$$c < \log n < \log^k n < n < n \log n < n^2 < n^3 < 2^n$$



Asymptotic Runtime and Its Application in Choosing Implementation

Different solutions for a problem often have different asymptotic runtime. We can analyze the runtime of each solution and pick the most efficient solution.

Let us look at an example together. If we are to write a program that calculates the value of x^n , where n is a positive integer, the code for the program can be straightforwardly written as shown in Figure 2-21 (we are not calling any pre-built function).

```
1: long power (long x, int n) {
2:     long result =1;
3:     for(int i=0;i<n;i++){
4:         result = result * x;
5:     }
6:     Return result;
7: }
```

Figure 2-21: Code for calculating x^n .

The above program executes its loop n times, therefore having its asymptotic runtime equals to $\theta(n)$.

This problem has a better solution, though. We can employ a divide and conquer approach as follows:

- If n is even, x^n comes from $(x * x)^{\frac{n}{2}}$.
- If n is odd, x^n comes from $(x * x)^{\frac{n}{2}} * x$, where $\frac{n}{2}$ uses integer division.

Divide and conquer is an algorithm that divides a problem into equal (or almost equal) portions. Let us look at the program code using this approach (Figure 2-22).

```
1: long power (long x, int n) {
2:     if (n==0)
3:         return 1;
4:     if (isEven (n))
5:         return power (x*x, n/2);
6:     else
7:         return power (x*x, n/2) *x;
8: }
```

Figure 2-22: Code for calculating x^n , written recursively.

Each time the method is called, n is reduced by half. This is very similar to the binary search problem. Until n reaches zero, the number of times method power is called is proportional to $\log_2 n$. Therefore, it has its asymptotic runtime equals to $\Theta(\log n)$.

By analyzing the asymptotic runtime of both programs, we can clearly pick the second implementation due to its smaller growth rate.

Another example is a program that calculates the largest gap between two values in an array. Below are two versions of this program. The first version calculates the difference between every value pair (Figure 2-23).

```
1:  int largestGap (int[] a){
2:      int largestGap = 0;
3:      for (int i = 0; i < a.length; i++) {
4:          for (int j = i+1; j < a.length; j++) {
5:              int dif = Math.abs(a[i] - a[j]);
6:              if (dif > largestGap) largestGap = dif;
7:          }
8:      }
9:      return largestGap;
10: }
```

Figure 2-23: Calculating the largest gap between 2 values in a given array, the exhaustive approach.

The other version finds the maximum and the minimum value, and then subtracts them (Figure 2-24).

```
1:  int largestGap (int[] a){
2:      int max = Integer.MIN_VALUE;
3:      int min = Integer.MAX_VALUE;
4:      for (int i = 0; i < a.length; i++) {
5:          if (a[i] > max) max = a[i];
6:          if (a[i] < min) min = a[i];
7:      }
8:      return max - min;
9:  }
```

Figure 2-24: Calculating the largest gap between 2 values in a given array, using the maximum and minimum value.

Let the array size be n . For the first version of the program, one loop is nested inside the other. The outer loop obviously iterates for n times. The running time of the inner loop varies according to the value of i . When i is 0, it runs $n-1$ times. When i is 1, it runs $n-2$ times, and so on. Therefore, the combined running time of both loops (from when i is 0 up to when i is $a.length-1$) is $(n-1) + (n-2) + (n-3) + \dots + 1 + 0 = \theta(n^2)$.

For the second version of the program, we only need one loop, running for n iterations, thus its asymptotic runtime is $\theta(n)$.

We can therefore choose to use the second implementation due to its slower growth rate.

Best-Case, Worst-Case, and Average Case Runtime

Best-case runtime is the fastest possible runtime for a program, when the input size is n . It has the lowest possible growth rate. Worst-case runtime is its opposite (we usually use the worst-case runtime as the upper bound for our asymptotic runtime).

What about average case runtime? How do we find its value? Well, the average case runtime is the average runtime of all possible runs of the program (when the input size is n). We can find this value from:

- Checking the number of inputs to the program.
- For each input, note down its runtime.
- Average case runtime =

$$\frac{\textit{the sum of all the runtime from each input}}{\textit{the number of possible inputs}}$$

- This is, however, based on an assumption that each input has equal probability of occurrences. If you

know that each input does not occur with equal probability, you must take that into account. This results in average case runtime =

$$\sum_i (\text{probability of input } i) * (\text{runtime for input } i)$$

Note that each probability value must be between (inclusive) 0 and 1. All probability values must add up to 1.

For an example, let us find best-case, worst-case, and average case runtime for a program that tries to find the index of value x in array size n . The program code is shown in Figure 2-25.

```
1:  int find (int x, int[] a) {
2:      for (int i = 0; i < a.length; i++) {
3:          if (a[i] == x) return i;
4:      }
5:      return -1;
6:  }
```

Figure 2-25: Finding the position of x in an array.

Best-case runtime takes place when x is in the first array slot. Hence the program enters its loop only once and returns immediately. Therefore, its runtime is constant ($\theta(1)$).

Worst-case runtime takes place when x is not in the array at all. The program enters the loop n times. Thus, the runtime is $\theta(n)$.

Asymptotic runtime is neither the worst-case nor the best case. Its value is $O(n)$ for this program.

For the average case, if x is in the array, it maybe in any array slot (each slot has equal chance to contain x). The case where x is not in the array also has equal chance compared to other possibilities.

Since there are n array slots, the probability of x in a slot is $\frac{1}{n+1}$. The probability for x not being in any slot is also $\frac{1}{n+1}$. The average runtime of this program =

$$\begin{aligned} & \frac{1}{n+1} (\text{runtime when } x \text{ is in the first slot}) + \\ & \frac{1}{n+1} (\text{runtime when } x \text{ is in the second slot}) + \dots + \\ & \frac{1}{n+1} (\text{runtime when } x \text{ is not in the array}) \\ &= \frac{1}{n+1} (1 + 2 + 3 + 4 + \dots + (n+1)) \\ &= \frac{(n+1) * (n+2)}{2(n+1)} \\ &= \frac{n+2}{2} \\ &= \theta(n) \end{aligned}$$

For this array search example, its average case runtime is equal to its worst-case runtime.

Beyond Big-Theta and Big-O

There are other asymptotic notation definitions. Let us check out some of them for completeness.

Definition 2-7: Big-Omega, or Ω

$\Omega(g(n))$ is a set of functions that grow not slower than $g(n)$.

$$\Omega(g(n)) = \{f(n) \mid \exists_c \exists_{n_0} c > 0 \wedge n_0 \geq 0 \wedge f(n) \geq cg(n), \text{ where } n \geq n_0\}$$

Basically, the definition says that $f(n) \in \Omega(g(n))$ if and only if $f(n)$ has greater or equal value to $cg(n)$ for all n greater than a certain value.

Definition 2-8: Little-O, or o

$o(g(n))$ is a set of functions that grow slower than $g(n)$.

$$o(g(n)) = \{f(n) \mid \exists_c \exists_{n_0} c > 0 \wedge n_0 \geq 0 \wedge f(n) < cg(n), \text{ where } n \geq n_0\}$$

Basically, $f(n) \in o(g(n))$ if and only if $f(n)$ has less value than $cg(n)$ for all n greater than a certain value. In other words:

$$\text{If } T(N) = O(p(N)) \text{ but } T(N) \neq \Theta(p(N)), \text{ then} \\ T(N) = o(p(N)).$$

Definition 2-9: Little-Omega, or ω

$\omega(g(n))$ is a set of functions that grow faster than $g(n)$.

$$\omega(g(n)) = \{f(n) | \exists c \exists n_0 c > 0 \wedge n_0 \geq 0 \wedge f(n) > cg(n), \text{ where } n \geq n_0\}$$

Exercises

1. Let $f(n) = 7n * \log 2n$ and $g(n) = n^2$. Find the value of n_0 , where $n_0 \leq n$, that satisfies $f(n) < g(n)$.
2. Show that, if $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$.
3. Show that $T(N) = O(f(N))$ if and only if $f(N) = \Omega(T(N))$.
4. Show that $7^{(n+1)}$ is $O(7^n)$.
5. Show that $n = O(n \log n)$.
6. Prove that $\log^k n = o(n)$ when k is a constant.

7. If $f(n) = 4n$ when n is an odd number, and $f(n) = n^2$ when n is an even number. Find the big-O of $f(n)$.
8. If there are n numbers. Write a program that finds the maximum and minimum values. The number of comparisons appeared in the program must not exceed $3n/2$ times.
9. Assume we have two programs. The first program has its worst-case running time = $230 n \log_2 n$. The other program has its worst-case running time = n^2 . Which value of n does the second program start to have its running time greater than the first program?
10. An equation, $p(x) = \sum_{i=0}^n a_i x^i$ can be re-written as:
 $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-1} + xa_n) \dots)))$. This is called Horner's method. Write a program to find the value of $p(x)$, given an array of a_i . Please also indicate the asymptotic runtime of your program.
11. Write a method *multiply*(*int x, int y*). You are only allowed to use the addition operator. What is the asymptotic runtime of your method.
12. Prove that $\sum_{i=1}^n (2i-1) = n^2$
13. Show that $\sum_{i=1}^n \lceil \log_2 i \rceil = \Theta(n \log n)$
14. A library fines us x dollars if we return books late for 1 day. The fine is multiplied by x each day. Write a method that calculates the fine on n^{th} day of late return.

15. Write a program to find the value of the minimum sum of a subsequence of integers in array a . What is the asymptotic runtime of this program?
16. Write a program that checks whether a positive integer n is a prime number. What is the asymptotic runtime of this program?
17. A program can process 300 input data in 0.25 second. How much data can it process in 5 minutes if:
- The program runs in $O(n \log n)$.
 - The program runs in $O(n^2)$.
18. Find asymptotic runtime of the following programs:

```

1:  sum=0;
2:  for (i=0; i<n; i++)
3:      for (j=0; j<n*n; j++)
4:          sum++;

```

```

1:  sum=0;
1:  for (i=0; i<n; i++)
2:      for (j=0; j<i; j++)
3:          sum++;

```

```

1:  sum=0;
2:  for (i=0; i<n; i++)
3:      for (j=0; j<i*i; j++)
4:          for (k=0; k<j; k++)
5:              sum++;

```

```

1:  sum=0;
2:  for (i=1; i<n; i++)
3:      for (j=1; j<i*i; j++)
4:          if (j%i == 0)
5:              for (k=0; k<j; k++)
6:                  sum++;

```

```
public void f(Object[] o){  
    for(int i=0; i<o.length; i++){  
        for(int j=1; j<o.length; j++){  
            ...  
        }  
    }  
}
```



Chapter 3 : List

In this chapter, we will look at list, its usage, and how we can implement it. We will mainly focus on an implementation called linked list.

List and Its Operations

A list is any structure that can store its data in sequence. Therefore, an array can be thought of as a list as well.

So, what can we do with a list? The table below shows possible operations that can be done:

Table 3-1: List operations

find	Find location of a given member.
insert	Insert a new member into a given position.
findKth	Give the k^{th} member of the list.
remove	Remove a specified member from the list.
head	Give the first member of the list.
tail	Give the list after removing the first member.
append	Attach 2 lists together.

Implementing a List with Array

As mentioned earlier, an array can be thought of as a list. What will happen if we use an array to implement a list?

Figure 3-1 to Figure 3-6 show the class that contains various methods listed in Table 3-1. For simplicity, all data are assumed to be integers.

```
1: public class ListFromArray {
2:     int[] theArray;
3:     static final int NUMBERSLOTS = 5;
4:     public ListFromArray() {
5:         theArray = new int[NUMBERSLOTS];
6:     }
7:
8:     public int find(int value) {
9:         if(theArray == null) return -1;
10:        for (int i = 0; i < theArray.length; i++) {
11:            if (theArray[i] == value)
12:                // exit immediately when
13:                // encounter the value.
14:                return i;
15:        }
16:        // return -1 if we do not find the value.
17:        return -1;
18:    }
19:
20:    public int findKth(int kthPosition) throws
21:    Exception{
22:        //throw exception if the array does
23:        //not exist or the position is negative or greater
24:        //than the last possible position, throw exception.
25:        if (theArray == null ||
26:            kthPosition < 0 ||
27:            kthPosition > theArray.length - 1)
28:            throw new Exception();
29:
30:        return theArray[kthPosition]; }
31:    }
32:    // the code is not finished.
```

$O(n)$

$\theta(1)$

Figure 3-1: List implementation using array.

From Figure 3-1, the list is implemented as a new class. Method *find* attempts to search our list (loop through our array), ending when it finds the given *value* (and returns the *value's* position). The method can end quickly if the

array is *null*, or if it can find *value* early. It can also look at every element in the array and do not find the required value at all. Therefore, its running time is $O(n)$, where n is the array size.

The method *findKth* attempts to find the k^{th} element stored in our list. It therefore accesses our array directly to find the k^{th} member (corresponding to the k^{th} member of the list). The method generates an exception if the given position is illegal for the given array. It can be seen that the method runs in constant time, $\theta(1)$, since it uses the array access operator to access the required array element instantly.

Figure 3-2 shows method *insert*. It attempts to put a new *value* into a given *position*. The method ends quickly if the array is initially empty or the position value is illegal (running time of this case is $\theta(1)$). Otherwise, it needs to loop through the array to expand the array size (this part takes $\theta(n)$). It also has to shift all elements from the *position* to the right (this part takes $O(n)$ since the number of shifting elements depends on *position*).

For example, let us insert 6 into the following array such that the array remains sorted (Figure 3-3). To keep the array sorted, 6 will have to be inserted into the slot after 5. Therefore 7 and 8 must be pushed to the right. For large arrays, there can be a lot of pushes.

```
1: public void insert(int value, int position) {
2:     // if we have empty array
3:     if (theArray == null) {
4:         if (position != 0) {
5:             return;
6:         } else {
7:             theArray = new int[1];
8:             theArray[0] = value;
9:             return;
10:        }
11:    }
12:
13:    // exit the method if the position is
14:    // negative or greater than the last possible
15:    // position.
16:    if (position < 0 || position > theArray.length - 1)
17:        return;
18:
19:    // copy contents into an expanded array
20:    //(must expand).
21:    int[] c = new int[theArray.length + 1];
22:    for (int i = 0; i < theArray.length; i++) {
23:        c[i] = theArray[i];
24:    }
25:    theArray = c;
26:
27:    // for each value from the specified position,
28:    // move the value to the right.
29:    for (int i = theArray.length - 1; i >= 0; i--) {
30:        if (i > position) {
31:            theArray[i] = theArray[i - 1];
32:        } else {
33:            theArray[i] = value;
34:            return; // return after the insertion.
35:        }
36:    }
37: } //the code is not finished.
```

$\theta(1)$

$\theta(n)$

$O(n)$

Figure 3-2: Method *insert* for linked list implemented using array.

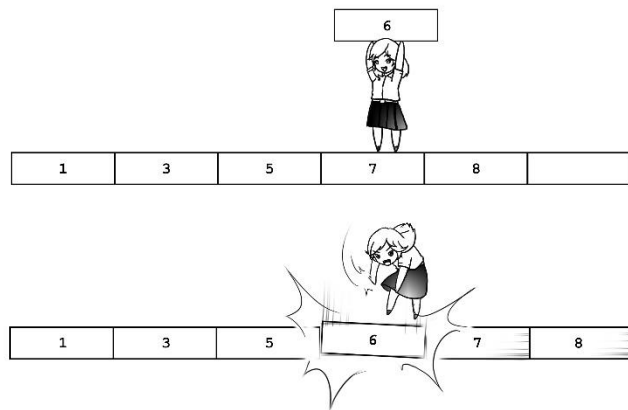


Figure 3-3: Inserting new data into array.

The running time of *insert* is $O(n)$. Although $\theta(n)$ is the most dominating term, the method can exit early regardless of the array size due to an illegal position value.

Figure 3-4 shows method *remove*. It tries to remove a specified value from our list. The program can end right away if the array is empty (nothing to be removed, $\theta(1)$). Otherwise, it has to loop in order to find *value* and copy all data on the right-hand side of the *value* in the array one place to the left ($O(n)$). It also needs to shrink the array by one slot, causing a need to copy almost the entire array ($\theta(n)$). The overall running time is $\theta(n)$. Conditional exits for this method do not count as premature exits because they are for cases where array

size is 0 or 1. Thus the method runtime is directly proportional to the array size. There is also a possibility that *value* is not in the array and the method will exit on line 14, but this case requires the whole array to be searched and therefore its performance is still $\theta(n)$.

```
1: public void remove(int value){
2:     int i;
3:     if (theArray == null) {           }  $\theta(1)$ 
4:         return;
5:     }
6:
7:     // search array for the value.
8:     for (i = 0; i < theArray.length; i++) { }  $\theta(n)$ 
9:         if (theArray[i] == value)
10:            break;
11:     }
12:
13:     if (i >= theArray.length) // if value not found. }  $\theta(1)$ 
14:         return;
15:
16:     // From the position of the value we want to
17:     //remove:
18:     // if the array has one element:
19:     if (theArray.length == 1) {       }  $\theta(1)$ 
20:         theArray = null;
21:         return;
22:     }
23:
24:     // if the array is not empty:
25:     for (; i < theArray.length - 1; i++) { }  $\theta(n)$ 
26:         // move value to the left.
27:         theArray[i] = theArray[i + 1];
28:     }
29: }
30:
31: // then shrink the array.
32: int[] c = new int[theArray.length - 1];
33: for (int j = 0; j < c.length; j++) { }  $\theta(n)$ 
34:     c[j] = theArray[j];
35: }
36: theArray = c;
37: } //the code is not finished.
```

Figure 3-4: Method *remove* of List implemented by array.

Consider an array in Figure 3-5, if we want to remove 5 from such array, the number 6, 7, and 8 will have to be copied to their left so that we can access all array slots from the beginning of the array without getting an undefined value. This is very time-consuming in a large array.

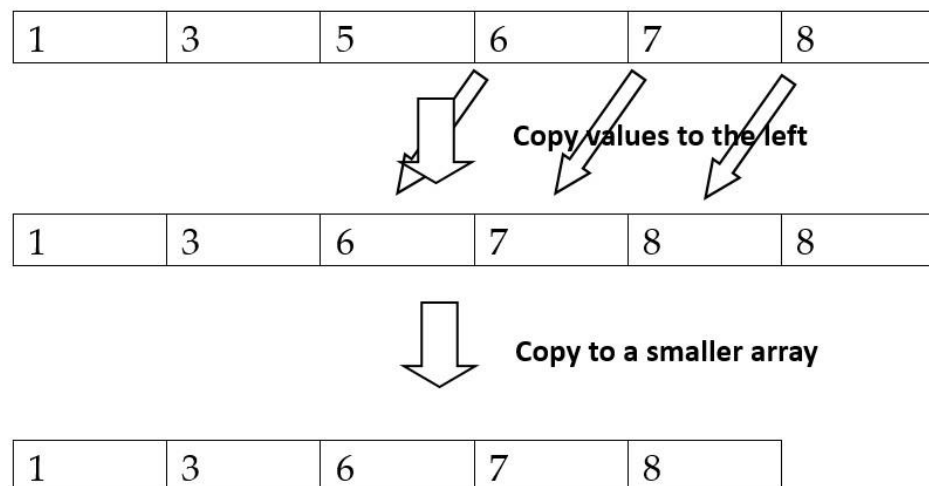


Figure 3-5: Removing data from an array.

Method *head*, *tail* and *append* are shown in Figure 3-6. Method *head* just tries to access the first member of the list, which is the first element of the array. Therefore, it takes constant time ($\Theta(1)$).

```
1: public int head() throws Exception{
2:     if(theArray == null)
3:         throw new Exception();
4:     return theArray[0];
5: }
6:
7: public ListFromArray tail() throws Exception{
8:     if(theArray == null)
9:         throw new Exception();
10:
11:     // create a copy because we don't want to
12:     // change the original.
13:     // Then just copy all except the first data.
14:     ListFromArray l2 = new ListFromArray();
15:     l2.theArray = new int[theArray.length-1];
16:     for (int i = 1; i < theArray.length; i++) {
17:         l2.theArray[i-1] = theArray[i];
18:     }
19:     return l2;
20: }
21:
22: public void append(ListFromArray b) {
23:     int thisLength = 0;
24:     int bLength = 0;
25:     int sumLength = 0;
26:     if (theArray != null) {
27:         thisLength = theArray.length;
28:     }
29:     if (b.theArray != null) {
30:         bLength = b.theArray.length;
31:     }
32:     sumLength = thisLength + bLength;
33:     if (sumLength == 0)
34:         return;
35:     int[] c = new int[sumLength];
36:     int cIndex = 0;
37:     for (int i = 0; i < thisLength; i++, cIndex++) {
38:         c[cIndex] = theArray[i];
39:     }
40:     for (int j = 0; j < bLength; j++, cIndex++) {
41:         c[cIndex] = b.theArray[j];
42:     }
43:     theArray = c;
44: }
45: } //end of class ListFromArray.
```

$\theta(n)$

$\theta(n)$

Figure 3-6: Method *head*, *tail*, and *append*.

Method *tail* ends immediately if the array is empty, or it has to copy the entire array except the first data. The conditional exit takes place when there is no data in the array. It will not be considered when we estimate the runtime since we only make our estimation from the case where the input size is n .

It basically means that the runtime always depends on array size, without any other factors to influence it. Therefore, *tail* takes $\Theta(n)$ to run.

Method *append* returns immediately ($\Theta(1)$) if both arrays we want to append do not have any content. Otherwise, contents from both arrays are copied into the resulting array, which takes $\Theta(n)$ if there are n total contents from the 2 original arrays. To sum up, the running time for *append* is $\Theta(n)$. The conditional exit does not count because it only happens when the arrays have no contents.

To summarize, *insert* and *remove* take some time to run because they need array elements to move around. Other methods such as *tail* and *append* take time because we need to copy data from the original array into a new array.

Speed improvement is impossible if a new copy of an array needs to be created. But for methods that just move data around, if we can find a data structure that does not

require elements to move around, we will save time. Indeed, such data structure exists. It is called a Linked List.



Implementing a List with Linked List

A linked list concept is shown in Figure 3-7.

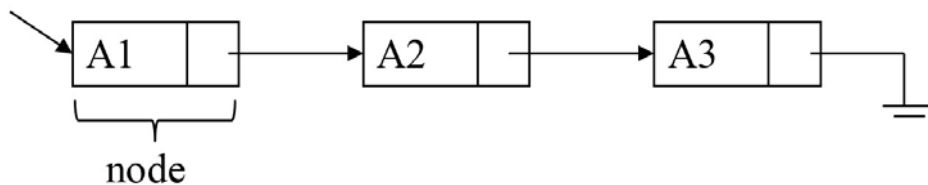


Figure 3-7: Linked list structure concept

A linked list consists of “nodes”. Each node stores a piece of data and a link to another node, thus storing data in

sequence. Each element in a list can be reached by following links from its first node.

Finding a specified value in a list will require searching from the very first node. Therefore, the running time is still $O(n)$. Method *findKth* can no longer make use of array index access so it has to count data from the beginning of a list, thus taking $O(k)$ to run (it can run in constant time if the value of k is illegal).

But for any functionalities that used to require array elements to move around, using linked lists eliminate such requirements. Let us see how this improvement is achieved. Figure 3-8 shows how to remove data A2 from a linked list that originally stores data A1, A2, and A3.

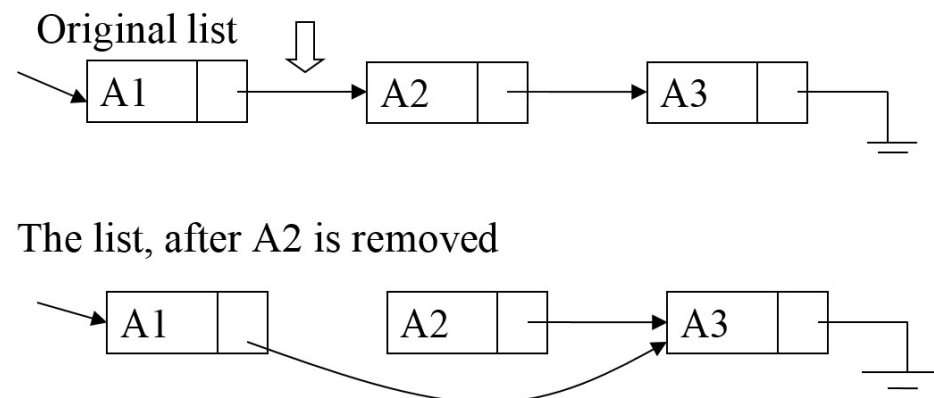


Figure 3-8: Removing data from a linked list.

Basically, we need just one change of link (a link can be called pointer, or reference). In Figure 3-8, the changed link is identified by a thick arrow. The link coming out of

the node that contains A2 does not have to be changed because A2 is no longer accessible, causing the link to become inaccessible as well (Java's garbage collector will clear it).

Inserting a new piece of data uses the same concept, as shown in Figure 3-9. A thick arrow is used to mark the link that needs to be changed when inserting X into the list. A new node containing X has to be created. Then, the link from the node that stores A2 to the node that stores A3 needs to be changed to point to the node that contains X. The link from the node that contains X is then set to point to the node that contains A3. Therefore, we achieve the effect of inserting X between nodes that store A2 and A3.

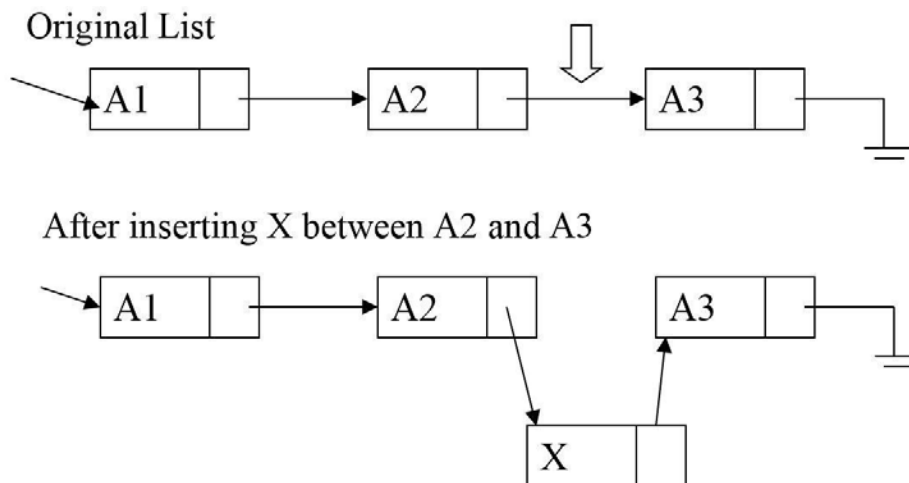


Figure 3-9: Inserting data into a linked list.

Note that inserting new data as a new first data in a list (Figure 3-10) requires different code to be written in the actual implementation. Since there is no node that can come before the newly created node (the new node stores X), we cannot write any code to change a link from a non-existing node.

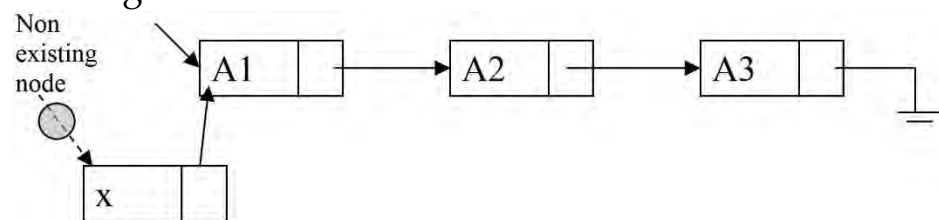


Figure 3-10: Inserting new data into the first position.

To avoid having to write a different code for this case, a header node (also called a dummy node) is introduced. The header node does not store any data. It is always in front of the list. With the addition of a header node, every node that stores data now has a node in front. Code is therefore the same. Figure 3-11 illustrates a linked list with header. Figure 3-12 shows an empty list with header. Yes, an empty list has a header too, but the pointer from its header node does not point to any other node. We call a pointer that does not point to anywhere a “null pointer”.

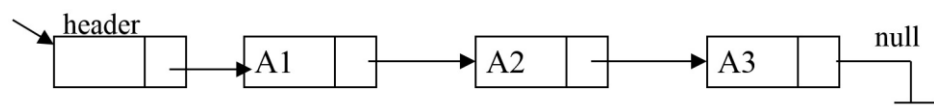


Figure 3-11: A linked list with a header node.

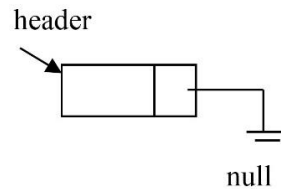


Figure 3-12: An empty linked list with a header node.

Now let us go through the implementation in detail. We will be looking at the following code portions respectively:

1. Node implementation.
2. Node marking implementation.
3. Linked list implementation.

Let us start with the implementation of a node. Figure 3-13 shows the code that defines a node that stores integer.

```
1:     class ListNode{
2:         int    data;
3:         ListNode nextNode;
4:
5:         // Constructors
6:         ListNode(int data){
7:             this(data, null);
8:         }
9:
10:        ListNode(int data, ListNode n){
11:            this.data = data;
12:            nextNode = n;
13:        }
14:    }
```

Figure 3-13: Implementation of a node that stores an integer.

A node constructed with statement:

```
ListNode a = new ListNode(5);
```

is shown in Figure 3-14.

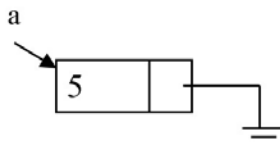


Figure 3-14: A node created from `ListNode a = new ListNode(5);`

Regarding node marking, a node we are interested in can be straightforwardly implemented because each node has a pointer to it, hence creating a pointer pointing to a node we are interested in should do the job. We can then traverse our linked list by following a link on each node. Figure 3-15 shows a linked list with 3 markers, two of them marking the same node.

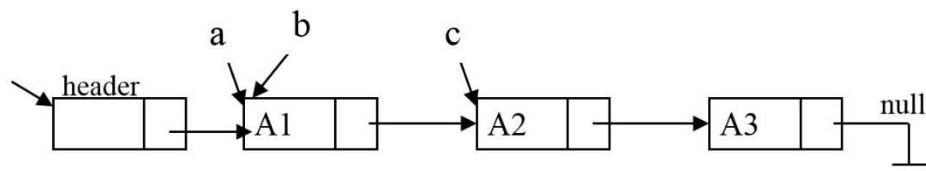


Figure 3-15: Simple markers a, b, and c at positions of interest.

However, traversing a list this way is not recommended. The reason is that there are usually many data structures that we need to provide for programmers, and each data structure is implemented differently. Thus, traversing

each data structure requires different implementation-specific methods, which is not very user-friendly. It is better to implement implementation-independent methods and have users call such methods on all data structures without having to go in-depth into each data structure implementation.

Here, we introduce iterator, one of the common implementation-independent approaches. An iterator for a data structure is an entity that marks and remembers the position of a single data. Once created, a programmer can instruct an iterator to mark the next data and operate on it with the following methods (regardless of implementation):

- *hasNext()*: checks if there is a next data in our data sequence.
- *next()*: returns the next data in the sequence.
- *set(Type value)*: replaces the last value returned by *next()* with *value*.

To make it implementation-independent, every possible implementation of each data structure should implement its own iterator that has the above functionalities. Thus, an iterator should be written as a Java interface. For our linked list, a linked list iterator can be created to implement the Iterator interface. Please note that Java already has Iterator and List Iterator interface that you can use. They are different from the implementation in this book. The implementation in this book focuses on

the fundamental knowledge needed to implement your own data structure from scratch, which is needed if readers want to customize a data structure of their own.

Figure 3-16 shows our iterator interface, simplified to use with data structures that store integers. In actual language libraries, data structures and iterators operate on more generic data types.

```
1: public interface Iterator{
2:     public boolean hasNext();
3:     public int next();
4:     public void set(int value);
5: }
```

Figure 3-16: interface Iterator.

Figure 3-17 shows our implementation of a list iterator class. Our linked list iterator implementation contains just one field, *currentNode*, that represents a node of which data has just been returned by method *next*. This is the node of interest. The list iterator constructor initializes this node to any given node from a linked list.

For example, if we are to initialize a linked list iterator to be ready for an iteration from the very first data in a linked list (assuming that the header node of that linked list is *header*), the following statement must be used:

```
ListIterator itr = new ListIterator(header);
```


Figure 3-18 shows the linked list and linked list iterator after the above statement was executed.

```
1: public ListIterator implements Iterator{
2:     ListNode currentNode;
3:
4:     public ListIterator(ListNode n){
5:         currentNode = n;
6:     }
7:
8:     public boolean hasNext(){
9:         return currentNode.nextNode != null;
10:    }
11:
12:    public int next() throws Exception{
13:        //Throw exception if the next data
14:        // does not exist.
15:        if(!hasNext())
16:            throw new NoSuchElementException();
17:        currentNode = currentNode.nextNode;
18:        return currentNode.data;
19:    }
20:
21:    public void set(int value){
22:        currentNode.data = value;
23:    }
24: }
```

Figure 3-17: Iterator for Linked List implementation.

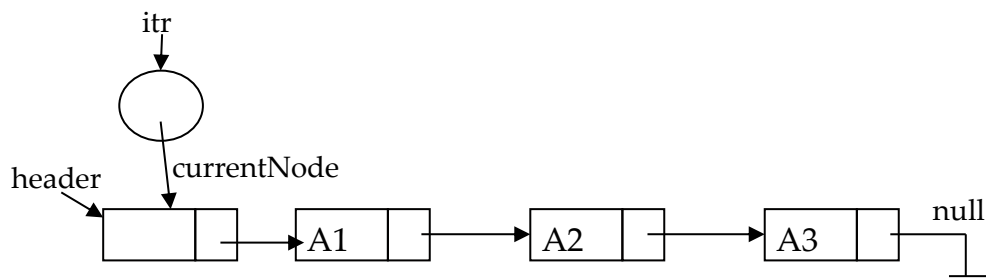


Figure 3-18: List iterator focusing on list header.

The method *hasNext* checks whether there is a node after the node of interest. For Figure 3-18, the method returns true since there is an actual node after *currentNode*. But for the list iterator in Figure 3-19, the method returns false because the node after *currentNode* does not exist.

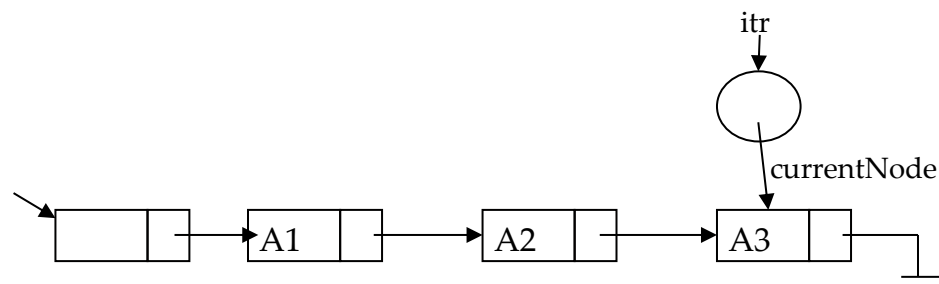


Figure 3-19: State of List iterator that method *hasNext* returns false.

Method *next* creates an exception if the next data does not exist. Otherwise, it moves *currentNode* by one position in the list and returns the data in the node it just moves to. Figure 3-20 shows an example of a list iterator status before and after method *next* is called.

The method *set* just straightforwardly changes the value of data in the node that we just focus on.

Now we are ready for our linked list implementation. Figure 3-21 displays the code for class `LinkedList`, its constructor, method *find* and method *findKth*.

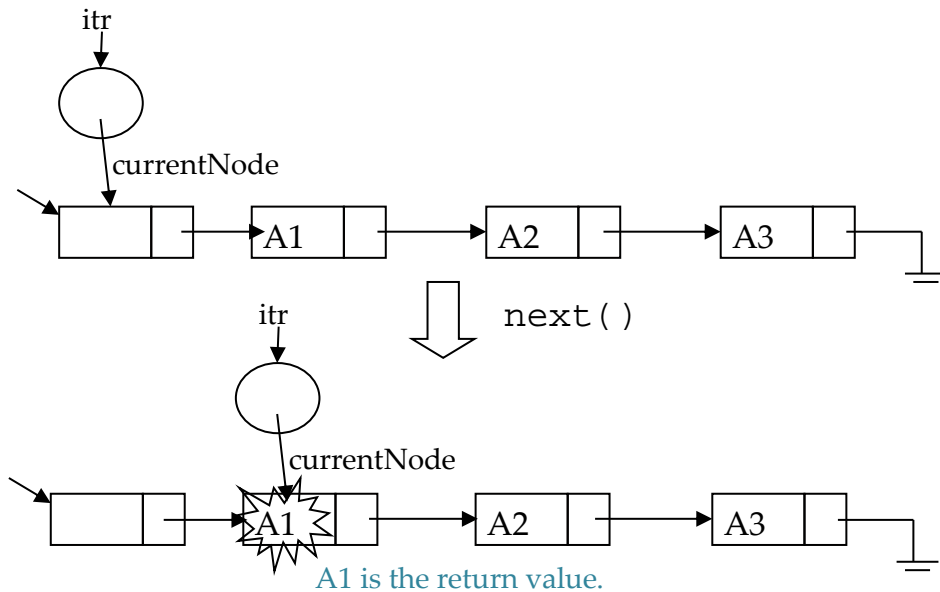


Figure 3-20: The working of method `next`.

The code for `find` works in the same way as its array version, that is, we need to search the data structure, one data at a time. The method ends immediately if the linked list is empty or if the value is found. Therefore, its running time is the same as its array counterpart, which is $O(n)$. An example run of this method, where the value we want to find is 5, is shown in Figure 3-22.

Method `findKth` suffers from the lack of direct positional access. Therefore, it requires a loop to look for the k^{th} data in the list (its code is similar to our code for `find`). But the loop can end as soon as the position is found. Thus, its asymptotic runtime is $O(n)$ (One could say the runtime is $O(k)$, but I prefer $O(n)$ because it directly informs us

about input size). This runtime is worse than its array counterpart, but the performance increase in *insert* and *remove* outweighs this drawback.

```
1: public class LinkedList {
2:     ListNode header;
3:     static int HEADER_MARKER = -9999999;
4:
5:     public LinkedList() {
6:         header = new ListNode(HEADER_MARKER);
7:     }
8:
9:     public int find(int value) throws Exception{
10:        Iterator itr = new ListIterator(header);
11:        int index = -1;
12:        while(itr.hasNext()){
13:            int v = itr.next();
14:            index++;
15:            if(v == value)
16:                //return the position of value.
17:                return index;
18:        }
19:        //return -1 if the value is not in the list.
20:        return -1;
21:    }
22:
23:    public int findKth(int kthPosition) throws Exception{
24:        //If the position number is negative (impossible)
25:        if (kthPosition < 0)
26:            throw new Exception
27:
28:        Iterator itr = new ListIterator(header);
29:        int index = -1;
30:        while(itr.hasNext()){
31:            int v = itr.next();
32:            index++;
33:            if(index == kthPosition)
34:                return v;
35:        }
36:        throw new Exception();
37:    }
38:    //This class continues in Figure 3-23.
```

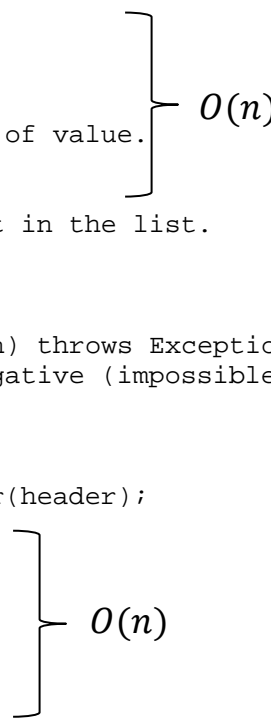


Figure 3-21: Linked List implementation (*constructor*, *find*, and *findKth*).

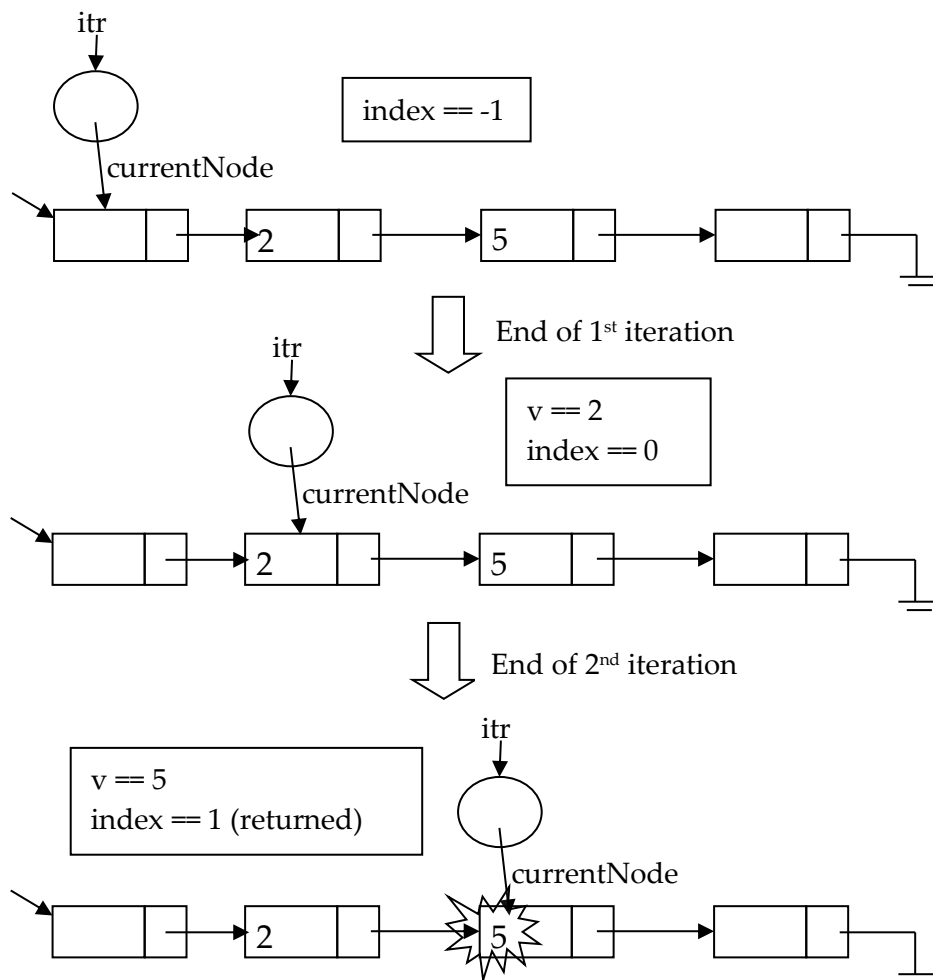


Figure 3-22: Execution steps of method *find*.

Method *insert* is shown in Figure 3-23. One of the parameters is different from its array counterpart. Instead of using a position number, we use an *Iterator* to indicate a position before the new data. We avoid using the position number because using it requires us to

iterate through the list, a very unnecessary time-consuming process. Using an *Iterator* to indicate a position allows our implementation to avoid any loop completely, thus reducing its asymptotic runtime to a constant value, $\theta(1)$. Please note that we have earlier seen the running time of *insert* for arrays, which is $O(n)$. A *ListNode* could also be used to indicate the position instead of an *Iterator*. For consistency, this book prioritizes *Iterator* when marking a position within a data structure.

```
1: public void insert(int value, Iterator p) throws Exception
2: {
3:     if (p == null || !(p instanceof ListIterator))
4:         throw new Exception();
5:     ListIterator p2 = (ListIterator)p;
6:     if(p2.currentNode == null) throw new Exception();
7:     ListNode n =
8:         new ListNode(value, p2.currentNode.nextNode);
9:     p2.currentNode.nextNode = n;
10: } // This class continues in Figure 3-26.
```

Figure 3-23: *insert* method of *LinkedList*.

Examples of what happens when method *insert* is called are shown in Figure 3-24 and Figure 3-25. They are slightly different from Figure 3-9 because the list has a header this time. The header node helps us add the new first data to the list without requiring any special-case coding.

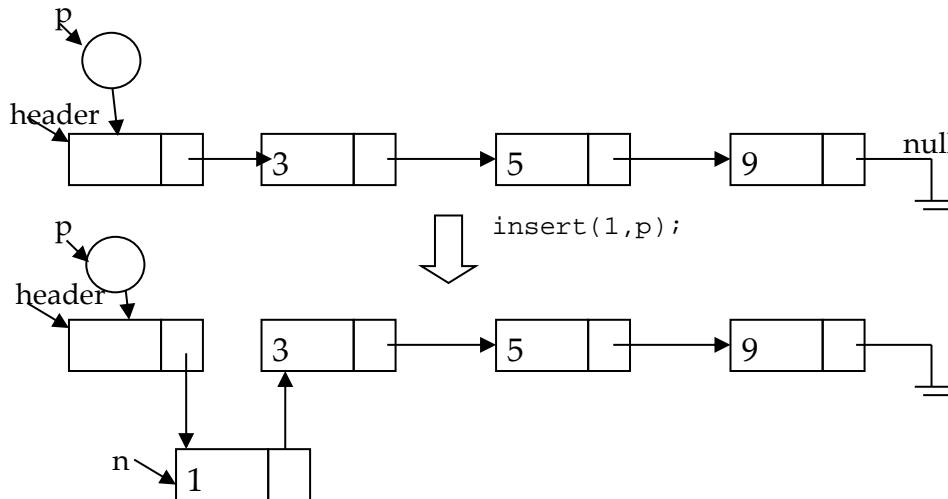


Figure 3-24: Inserting a new value at the start of the list (after the header).

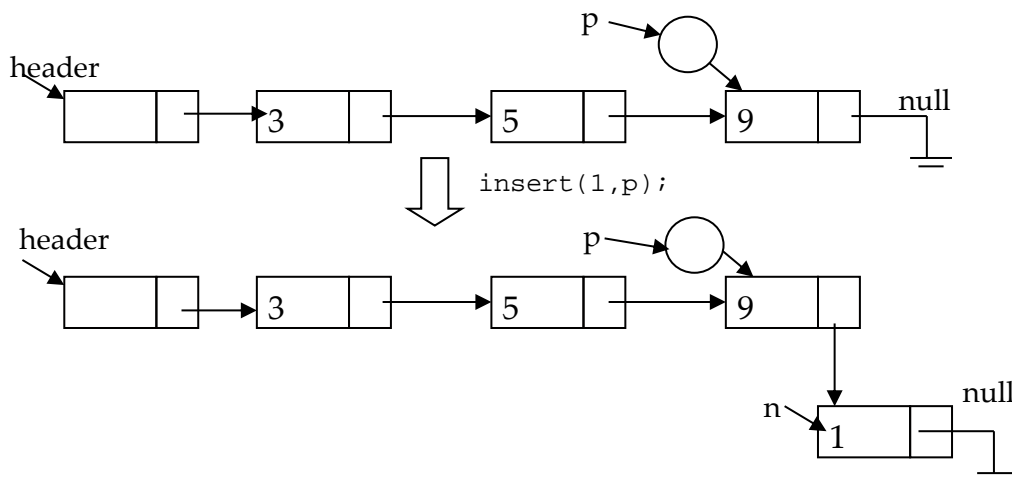


Figure 3-25: Inserting after the last data in a list.

The code for method *remove* is shown in Figure 3-26. It makes a call to method *findPrevious* and another *remove* which uses a position parameter.

Method *findPrevious* attempts to find a node just before the node that stores *value*. It returns null if no such node exists.

```

1:    public void remove(int value) throws Exception{
2:        Iterator p = findPrevious(value);
3:        if(p == null)
4:            return;
5:        remove(p);
6:    }
7:
8:    public Iterator findPrevious(int value) throws
9:    Exception{
10:        Iterator itr1 = new ListIterator(header);
11:        Iterator itr2 = new ListIterator(header);
12:        if(!itr2.hasNext())
13:            return null;
14:        int currentData = itr2.next();
15:        while(currentData != value && itr2.hasNext()){
16:            currentData = itr2.next();
17:            itr1.next();
18:        }
19:        if(currentData == value)
20:            return itr1;
21:        return null;
22:    }
23:
24:    public void remove(Iterator p){
25:        if(p == null || !(p instanceof ListIterator))
26:            return;
27:        ListIterator p2 = (ListIterator)p;
28:        if(p2.currentNode == null ||
29:            p2.currentNode.nextNode == null)
30:            return;
31:        p2.currentNode.nextNode =
32:            p2.currentNode.nextNode.nextNode;
33:    }
34:    // This class continues in Figure 3-31.

```

} $O(n)$

} $\theta(1)$

Figure 3-26: *remove* method of *LinkedList*.

First, two iterators are created focusing on *header*. Then we check if the list is an empty list by calling method *hasNext*. If the list is empty (method *hasNext* returns

false), null is returned right away (line 12-13 in Figure 3-26. Also, see Figure 3-27 for our drawing of this case). If not, we move the position of interest of *itr2* by one so that the position is ahead of the position of interest of *itr1* (see Figure 3-28). Then we move the positions of interest of both iterators until we find *value* or until there is no more data to work with. If *value* is found, we return the position of interest, *itr1*, which is the position before the node that stores *value* (see the lower part of Figure 3-28). Otherwise, *null* is returned because *value* surely does not exist in the list (line 21 in Figure 3-26). Figure 3-29 shows the final state of everything before the method returns, when *value* is not in the list.

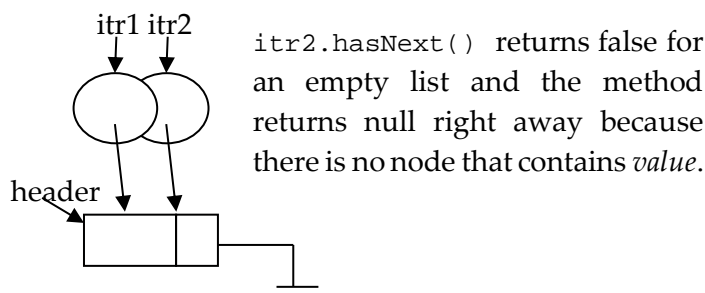


Figure 3-27: Status of variables when *findPrevious* is called on an empty list.

Method *remove* (line 24-33 of Figure 3-26) works just as illustrated in Figure 3-8.

Let us analyze the asymptotic runtime of the whole *remove* process. Since we have to search for *value*, a loop has to be employed (line 15-18 of Figure 3-26). The loop

can exit at any stage hence its asymptotic runtime is $O(n)$. Other parts of the program take constant time to run. Therefore, the overall running time of method *remove* is $O(n)$. This running time is equal to its array counterpart mainly because of the search requirement.

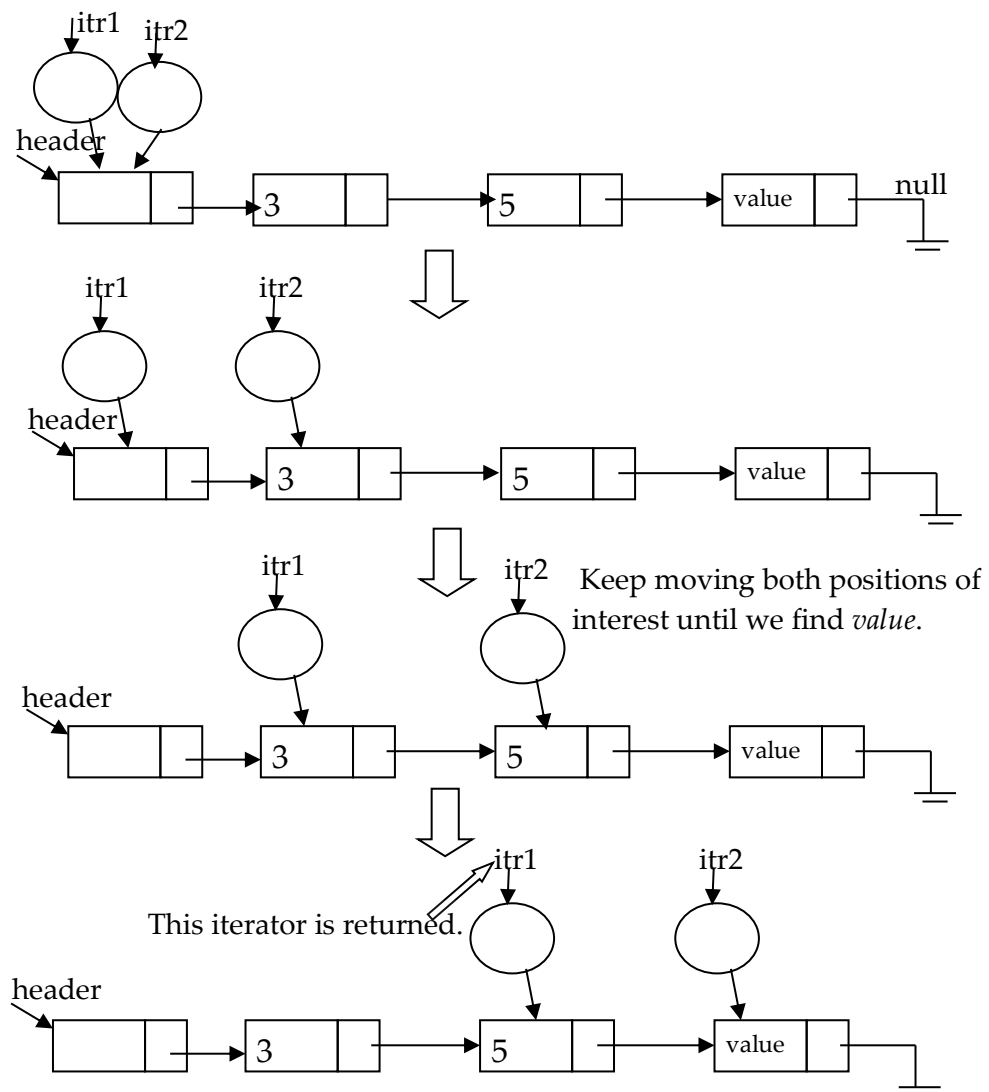


Figure 3-28: Status of variables in each step of execution when *findPrevious* is called on a list that stores *value*.

If we know the position of the data to be removed in advance, we can utilize method *remove* (line 24-33 of Figure 3-26) and remove the data in constant time. Note that in the array implementation, even though we may know the position of the data to be removed, other data to the right of the to-be-removed data must be shifted, causing the running time to still be $O(n)$.

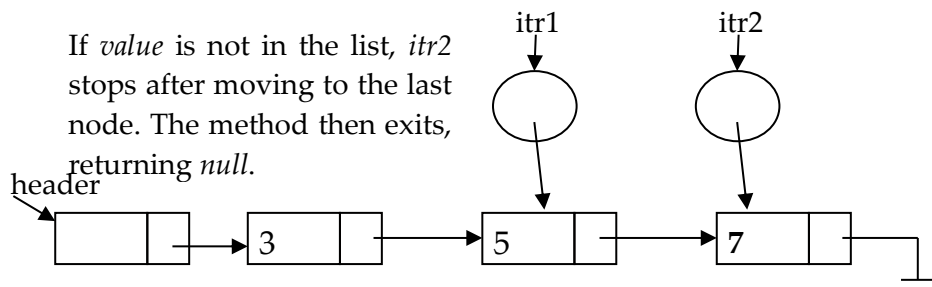


Figure 3-29: Final status of variables when *findPrevious* is called on a list that does not store *value*.

Method *head* and *tail* are shown in Figure 3-30. We introduce *isEmpty*, which checks whether the list does not store any data, and *makeEmpty*, which disconnects the link from the header to other nodes, effectively make the list become empty.

Method *head* is straightforward. It just tries to return data in the node next to *header*. So, it runs in constant time just like its array counterpart.

Method *tail* exits immediately if the list is empty. Otherwise it uses a loop to copy data (except the first

data) to a new list and return that list. The new list has to be created so that the original list is still preserved. The number of iterations is directly proportional to the size of the list. So, its asymptotic runtime is $\theta(n)$. Although the method can exit early, it does so due to the size of the list being zero.

```

1:   public boolean isEmpty(){
2:       return header.nextNode == null;
3:   }
4:
5:   public void makeEmpty(){
6:       header.nextNode = null;
7:   }
8:
9:   public int head() throws Exception{
10:      if(isEmpty())
11:          throw new Exception();
12:      return header.nextNode.data;
13:  }
14:
15:
16:  public LinkedList tail() throws Exception{
17:      if(isEmpty())
18:          throw new Exception();
19:
20:      // Now create a copy of the list
21:      // so that the original does not change.
22:      // Copy everything except the first data
23:      // to the new list.
24:      LinkedList list2 = new LinkedList();
25:      Iterator p1 = new
26:          ListIterator(header.nextNode);
27:      Iterator p2 = new ListIterator(list2.header);
28:      while(p1.hasNext()){
29:          int data = p1.next();
30:          list2.insert(data,p2);
31:          p2.next();
32:      }
33:      return list2;
34:  } //This class continues in Figure 3-31.

```

Figure 3-30: Method *head* and *tail* of *LinkedList*.

If we allow our original list to change, the method can simply remove the first data and take constant time. But here we do not want method *tail* to change our original list.

Method *append* (see Figure 3-31) enters loops n times, where n is the total amount of data from both lists. Therefore, its asymptotic runtime is $\theta(n)$, just like its array counterpart. This method can be made to run in constant time if we can mark the last node in our first list in advance, and connect the two lists together by changing just the end pointer from our first list (but doing it this way means any change made to one list will surely affect the other list). Please note that our *append* changes *this* list.

```
1: public void append(LinkedList list2) throws Exception{
2:     Iterator p1 = new ListIterator(header);
3:     Iterator p2 = new ListIterator(list2.header);
4:
5:     //move iterator to the end of our list.
6:     while(p1.hasNext())
7:         p1.next();
8:
9:     //then copy everything from list2 to our list.
10:    while(p2.hasNext()){
11:        insert(p2.next(),p1);
12:        p1.next();
13:    }
14: }// end of class LinkedList.
```

} $\theta(n)$

Figure 3-31: Method *append* of *LinkedList*.

From our implementation of linked list, the runtime can greatly be reduced from its array implementation counterpart when doing *insert* or *remove*, if the position before the node to be inserted or removed can be identified. This can greatly save time when our program has a lot of *insert* and *remove*. Table 3-2 summarizes asymptotic runtime comparisons of operations on an array and a linked list.

Table 3-2: Asymptotic runtime comparisons on operations of array and linked list.

operations	Array	Linked list
find	$O(n)$	$O(n)$
insert	$O(n)$	$\theta(1)$
findKth	$\theta(1)$	$O(n)$
remove	$\theta(n)$	$\theta(1)$
head	$\theta(1)$	$\theta(1)$
tail	$\theta(n)$	$\theta(n)$
append	$\theta(n)$	$\theta(n)$



Doubly-linked list

A doubly-linked list is very similar to the linked list we have seen in the previous section. However, its node stores one extra variable, *previousNode*, which is a link back to the node to its left. Having this extra link allows us to iterate through the list in both directions (obviously, the list iterator also needs to be expanded). Extra pointers mean we need to be more careful when updating them though.

Implementation can also be done such that a header is linked back to the node that stores the last data. This way, the list can be traversed in circle, in both directions. A doubly-linked list that can be traversed in circle is called a circular doubly-linked list. Figure 3-32 shows a circular doubly-linked list structure which stores 3, 6, and 4.

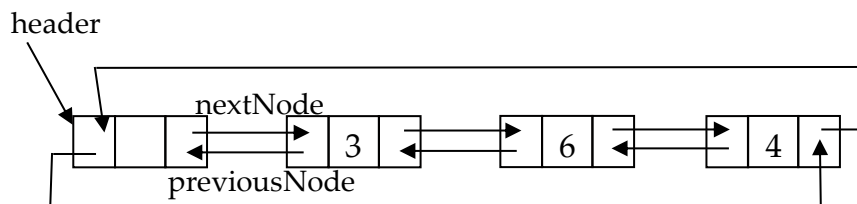


Figure 3-32: Example of a circular doubly-linked list.

Let us go through this list implementation. First of all, the structure of a *ListNode* needs to be changed to include a pointer back to a node that comes before it. Our extended class is called *DListNode*. The new pointer is

named *previousNode*. Figure 3-33 shows the code of *DListNode*.

```
1: class DListNode {
2:     int data;
3:     DListNode nextNode, previousNode;
4:
5:     // Constructors
6:     DListNode(int data) {
7:         this(data, null, null);
8:     }
9:
10:    DListNode(int theElement, DListNode n, DListNode p) {
11:        data = theElement;
12:        nextNode = n;
13:        previousNode = p;
14:    }
15: }
```

Figure 3-33: Code of a node of a doubly-linked list.

From Figure 3-33, with 2 pointers to a node before and a node after it, our *DListNode* object also has constructors that initialize the values of both pointers. Figure 3-34 shows a *DListNode* created by its default constructor (line 6-8 in Figure 3-33), with the code:

```
DListNode a = new DListNode(9);
```

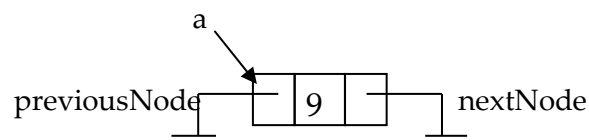


Figure 3-34: A *DListNode* created by its default constructor.

For node marking, our iterator has to be able to do a bit more. Thus, we add the following functions:

- *hasPrevious()*: returns true if there is a node prior to the currently interested node. Please note that since our list is circular, this will always be true. But this may not be true for some other implementations.
- *previous()*: returns the value currently marked, then move our iterator to the node before the current node. Please note that the returned value is obtained before the iterator is moved. Therefore, calling *next()* and *previous()* in sequence will return the same values.

Our modified iterator interface is shown in Figure 3-35. This iterator interface can be used with any data structure as long as the data structure allows 2-way traversal. Note that in Java we can write this new iterator interface by extending from the iterator in Figure 3-16. But here, the whole interface is re-written so that readers can clearly spot the differences between *next()* and *previous()*.

```
1: public interface Iterator {
2:     public boolean hasNext();
3:     public boolean hasPrevious();
4:
5:     // moves iterator to the next position,
6:     // then returns the value at that new position.
7:     public int next() throws Exception;
8:
9:     // returns the value at current position,
10:    // then moves the iterator back one position.
11:    public int previous() throws Exception;
12:
13:    public void set(int value);
14: }
```

Figure 3-35: Iterator that can traverse a data structure in two directions.

The code for our 2-way linked list iterator is shown in Figure 3-36. Instead of extending from the 1-directional version, the code is shown here in its entirety in order to emphasize the differences between *next()* and *previous()*.

```
1. public class DListIterator implements Iterator {
2.     DListNode currentNode; // interested position
3.     DListIterator(DListNode theNode) {
4.         currentNode = theNode;
5.     }
6.
7.     public boolean hasNext() {
8.         // always true for circular list.
9.         return currentNode.nextNode != null;
10.    }
11.
12.    public boolean hasPrevious() {
13.        // always true for circular list.
14.        return currentNode.previousNode != null;
15.    }
16.
17.    public int next() throws Exception {
18.        if (!hasNext())
19.            throw new NoSuchElementException();
20.        currentNode = currentNode.nextNode;
21.        return currentNode.data;
22.    }
23.
24.    public int previous() throws Exception{
25.        if (!hasPrevious())
26.            throw new NoSuchElementException();
27.        int data = currentNode.data;
28.        currentNode = currentNode.previousNode;
29.        return data;
30.    }
31.
32.    public void set(int value) {
33.        currentNode.data = value;
34.    }
35. }
```

Figure 3-36: Bi-directional linked list iterator.

Statement `DListIterator itr = new DListIterator(header);` creates an iterator pointing to header, as shown in Figure 3-37.

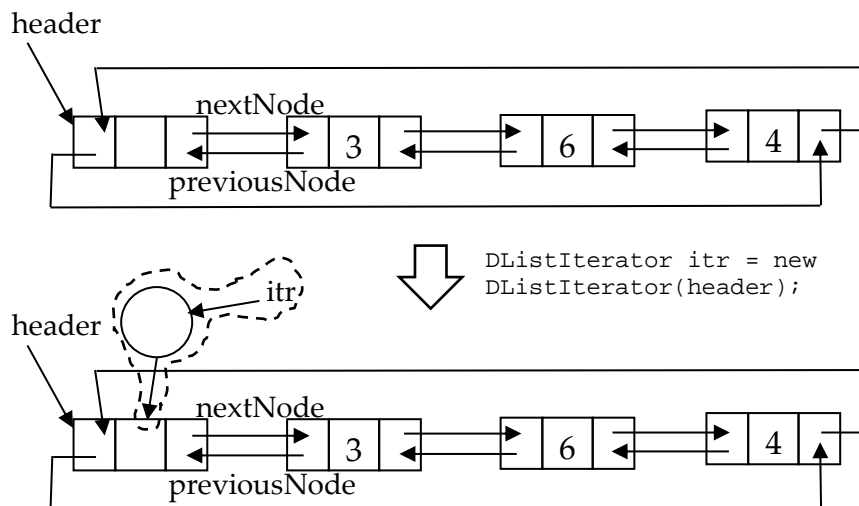


Figure 3-37: List iterator creation on a doubly-linked list.

The code for circular doubly-linked list can be written by extending from its singly-linked list counterpart, but due to a very different logic in list traversal, where there is no end of the list in this new circular implementation, all methods have to adapt this logic. This makes it difficult to write each method based on its existing counterpart. Therefore, the code shown in this section will be a complete rewrite, with differences highlighted in the code explanation.

Figure 3-38 shows our doubly-linked list class, *CDLinkedList*, with its variables, constructor, and method *isEmpty*, *makeEmpty*, and *size*.

```
1: public class CDLinkedList {
2:     DListNode header;
3:     int size;
4:     static final int HEADERVALUE = -9999999;
5:
6:     public CDLinkedList() {
7:         size =0;
8:         header = new DListNode(HEADERVALUE);
9:         makeEmpty(); //necessary, otherwise
10:                    // next/previous node will be null.
11:     }
12:
13:     public boolean isEmpty() {
14:         return header.nextNode == header;
15:     }
16:
17:     public void makeEmpty() {
18:         header.nextNode = header;
19:         header.previousNode = header;
20:     }
21:
22:     public int size(){
23:         return size;
24:     }
25:     //The class continues in Figure 3-40.
```

Figure 3-38: Circular doubly-linked list variables, constructor, and small utility methods.

For variables, we now have *size*. With it, now we can keep track of the number of data stored in the list without having to do any list traversal. We only need to update the value of *size* when adding and removing data from the list.

For the constructor (which creates an empty list), when creating the header, we **cannot** simply write:

```
header = new DListNode(HEADERVALUE,header,header);
```

The statement above will initialize *nextNode* and *previousNode* to *null* because the default value for *header* is *null* (the right-hand side of the assignment operator is executed before the left-hand side). In order to make *nextNode* and *previousNode* point to *header*, to create an empty list which is circular, we need to set their values after *header* is actually created (line 8-9, 17-20 of Figure 3-38). This process utilizes method *makeEmpty*, which can be used to reset any list back to an empty list.

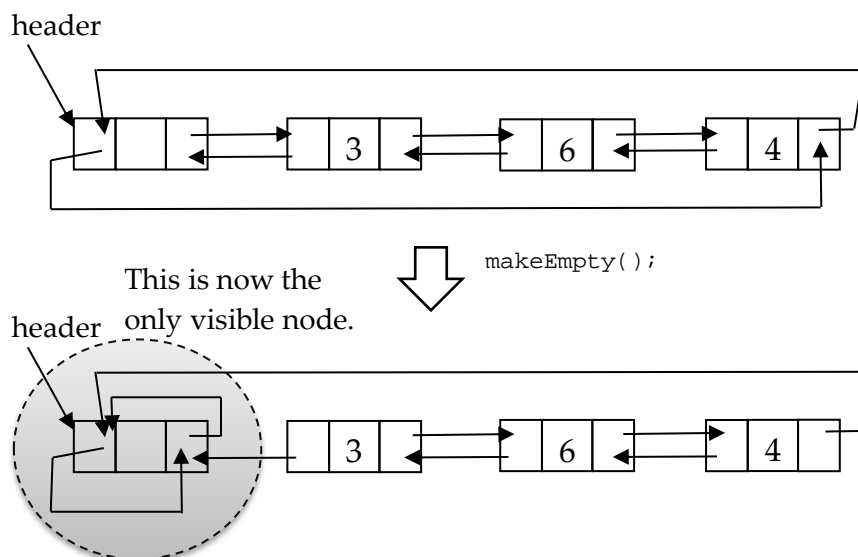


Figure 3-39: Making an empty list with method *makeEmpty*.

Figure 3-39 shows the working of method *makeEmpty*. It makes any generic list an empty list. All nodes except header are no longer accessible from any named variables and will be removed from memory by garbage collector.

The method *isEmpty* no longer checks for *null*. Since the list now goes in circle, *header* of an empty list has its *nextNode* and *previousNode* points to itself (see **Error! Reference source not found.**).

The implementation of the *find* method, that returns the position of *value* in the list (the first data is at position 0), is shown in Figure 3-40.

```
1: public int find(int value) throws Exception {
2:     Iterator itr = new DListIterator(header);
3:     int index = -1;
4:     while (itr.hasNext()) {
5:         int v = itr.next();
6:         index++;
7:         DListIterator itr2 = (DListIterator) itr;
8:         if (itr2.currentNode == header) //not found
9:             return -1;
10:        if (v == value)
11:            //return position of the value.
12:            return index;
13:    }
14:    return -1;
15: }
```

Figure 3-40: Method *find* of circular doubly-linked list.

The method is quite similar to its singly-linked list counterpart (see Figure 3-21 and Figure 3-22), but instead

of checking if the iterator has reached *null*, we check if the iterator has reached *header* instead (line 7-8 in Figure 3-40) because reaching *header* means we have looked at all data in our circular list. The performance of method *find* is $O(n)$, just like its singly-linked list counterpart because the list iterator may run through all data but the method can also exit early.

The implementation of method *findKth* is shown in Figure 3-41. It operates in almost the same way as its singly-linked list counterpart (its asymptotic runtime is also $O(n)$). The only difference is that it checks for *header* instead of *null* when determining whether the last data in the list has been processed (line 15 in Figure 3-41).

```
1:  public int findKth(int kthPosition) throws Exception{
2:      if (kthPosition < 0)
3:          throw new Exception();
4:          // exit the method if the position is
5:          // less than the first possible
6:          // position, throwing exception in the
7:          //process.
8:      Iterator itr = new DListIterator(header);
9:      int index = -1;
10:     while(itr.hasNext()){
11:         int v = itr.next();
12:         index++;
13:         DListIterator itr2;
14:         itr2 = (DListIterator) itr;
15:         if (itr2.currentNode == header)
16:             throw new Exception();
17:         if(index == kthPosition)
18:             return v;
19:     }
20:     throw new Exception();
21: }
```

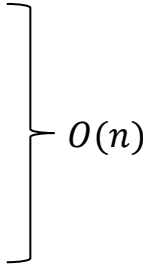


Figure 3-41: Method *findKth* of circular doubly-linked list.

The code for method *insert* is shown in Figure 3-42.

```
1: public void insert(int value, Iterator p) throws Exception
2: {
3:     if (p == null || !(p instanceof DListIterator))
4:         throw new Exception();
5:     DListIterator p2 = (DListIterator) p;
6:     if (p2.currentNode == null)
7:         throw new Exception();
8:
9:     DListIterator p3 = new
10:         DListIterator(p2.currentNode.nextNode);
11:     DListNode n;
12:     n = new DListNode(value, p3.currentNode, p2.currentNode);
13:     p2.currentNode.nextNode = n;
14:     p3.currentNode.previousNode = n;
15:     size++;
16: }
```

Figure 3-42: Method *insert* of circular doubly-linked list.

Method *insert* (Figure 3-42) tries to add *value* into our linked list, by putting the value at position just behind a node that is marked by *p*. First, if *p* has illegal value, then an exception is thrown since we cannot insert new data after an illegal position.

Illegal values of *p* are:

- *p* is *null* (line 3 in Figure 3-42).
- *p* is not an iterator that can navigate a doubly-linked list (line 3 in Figure 3-42).
- Value of *currentNode* stored in *p* is null, meaning *p* does not mark any position in the linked list (line 5-6 in Figure 3-42).

After we make sure that *p* is legal (in the code, *p* is now casted to *p2* to allow field access since *p* is just a normal

Iterator, not *DListIterator*), a new iterator ($p3$) is created to mark the node after the node marked by p . This new *DListIterator*, together with $p2$, are then used to create a new node with *value* inside. The pointers to/from the new node are also adjusted (see bottom half of Figure 3-43). The pointers adjustment does not need any loop, so the asymptotic runtime of this insert method is $\Theta(1)$.

Let us view an example in Figure 3-43. In the example, we insert 5 after the position that stores 3. Dotted line marked changes made in each step.

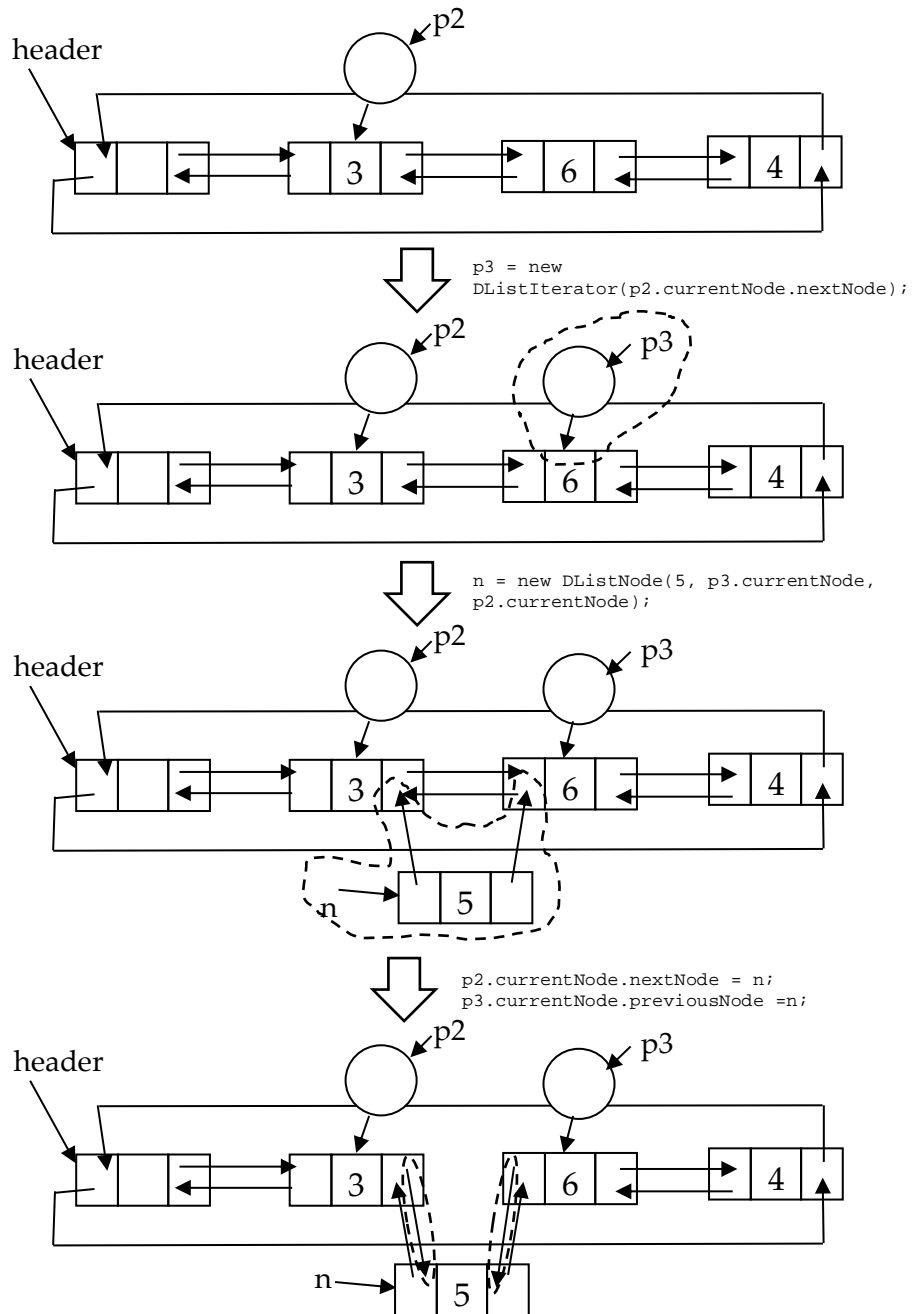


Figure 3-43: Execution steps of insert for doubly-linked list.

The code for method *remove* updated for a doubly-linked list is shown in Figure 3-44, with its utility method *findPrevious* and *remove(Iterator p)* shown in Figure 3-45.

```
1: // remove the first instance of the given data.
2: public void remove(int value) throws Exception {
3:     Iterator p = findPrevious(value);
4:     if (p == null)
5:         return;
6:     remove(p);
7: }
```

Figure 3-44: Method *remove* of doubly-linked list.

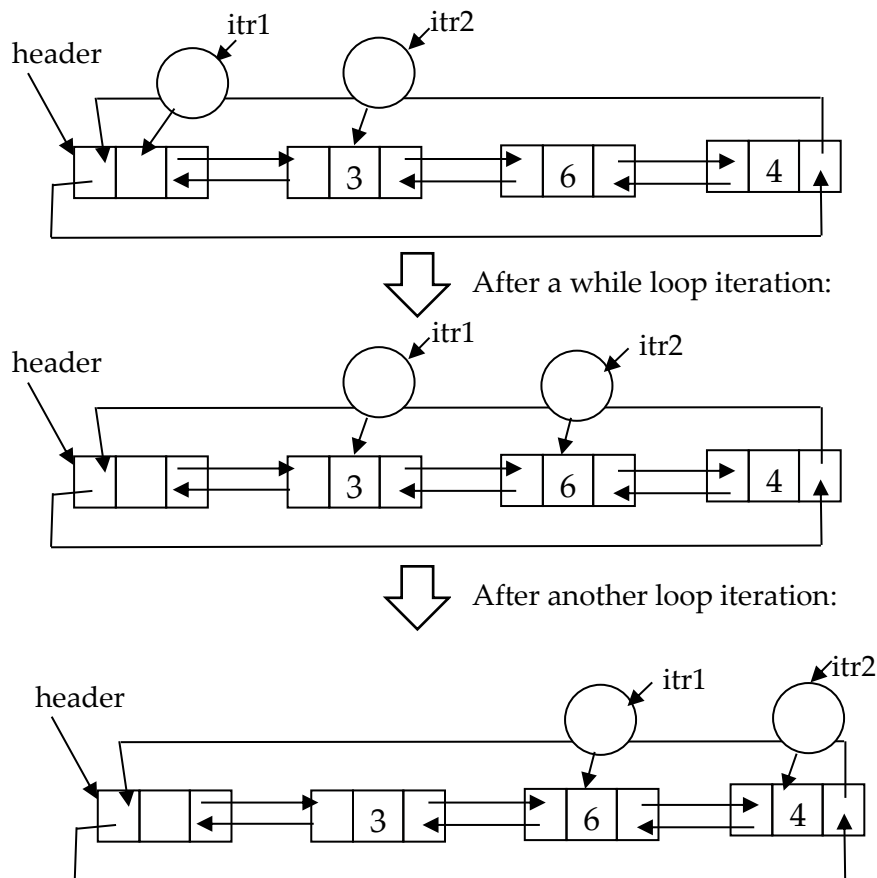
The method *remove* is almost the same as its singly-linked list counterpart. In fact, the code in Figure 3-44 is the same as its singly-linked list version. Its core concept does not change. That is, we must find the position prior to *value*, then remove the node at that position by changing pointers. Thus, calls to method *findPrevious* and *remove(Iterator p)* are still needed (their codes are in Figure 3-45). This time, however, method *findPrevious* needs to check for *header* instead of *null* to determine whether all data in the list are examined. Also, *remove(Iterator p)* needs to check for *header* instead of *null*. But since everything else does not change, (except one more *previousNode* pointer got updated) the asymptotic runtime of *remove(Iterator p)* is still $\Theta(1)$. The runtime for method *findPrevious* is also still $O(n)$. Thus, the overall runtime of the *remove* method in Figure 3-44 is $O(n)$, which is the same as its singly-linked list counterpart.

```
1: // Return iterator at position before the first
2: // position that stores value.
3: // If the value is not found, return null.
4: public Iterator findPrevious(int value) throws
5: Exception {
6:     if (isEmpty())
7:         return null;
8:     Iterator itr1 = new DListIterator(header);
9:     Iterator itr2 = new DListIterator(header);
10:    int currentData = itr2.next();
11:    while (currentData != value) {
12:        currentData = itr2.next();
13:        itr1.next();
14:        if (((DListIterator) itr2).currentNode ==
15:            header)
16:            return null;
17:    }
18:    if (currentData == value)
19:        return itr1;
20:    return null;
21: }
22:
23: //Remove content at position just after the given
24: // iterator. Skip header if found.
25: public void remove(Iterator p) {
26:     if (isEmpty())
27:         return;
28:     if (p == null || !(p instanceof DListIterator))
29:         return;
30:     DListIterator p2 = (DListIterator) p;
31:     if (p2.currentNode == null)
32:         return;
33:     if (p2.currentNode.nextNode == header)
34:         p2.currentNode = header;
35:     DListIterator p3;
36:     p3 = new
37:     DListIterator(p2.currentNode.nextNode.nextNode);
38:
39:     p2.currentNode.nextNode = p3.currentNode;
40:     p3.currentNode.previousNode = p2.currentNode;
41:     size--;
42: }
```

Figure 3-45: Method *findPrevious* and *remove(Iterator p)* of doubly-linked list.

Again, if a position is known in advance, data removal takes constant time.

An illustrated example of *findPrevious* code from Figure 3-45 is shown in Figure 3-46. Here we execute command *findPrevious(4)*. Figure 3-46 shows the execution after line 10.



The loop stops when *currentData == 4*. The method then returns *itr1*, the position before our value.

Figure 3-46: The working of *findPrevious* for doubly-linked list.

An example of `remove` with iterator parameter from Figure 3-45 is shown in Figure 3-47. Dotted lines show pointer changes. It can be seen that at the end, no pointer is connected to the node that stores 4, thus effectively rendering the node that contains 4 inaccessible from the rest of the list. The node that contains 4 will be cleaned up by a garbage collector (in languages without automatic memory management, you may have to remove the node by yourself, depending on the language).

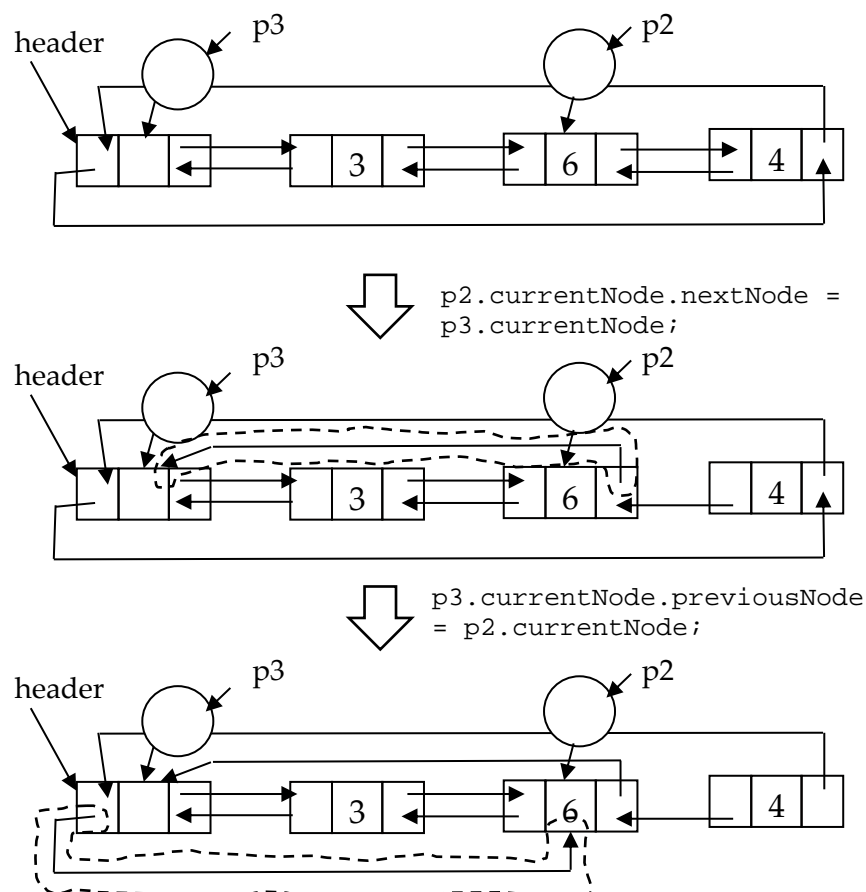


Figure 3-47: How `remove(Iterator p)` operates in doubly-linked list.

Our remove method from Figure 3-44 mainly removes the first instance of a specified value. The remove method in Figure 3-45 removes data behind a specified position. What if we want to remove data at a specified position? The code for this is shown in Figure 3-48.

```
1: // Remove data at position p.
2: // if p points to header or the list is empty, do
3: // nothing.
4: public void removeAt(Iterator p) throws Exception{
5:     if (isEmpty() || p == null
6:         || !(p instanceof DListIterator)
7:         || ((DListIterator) p).currentNode == null
8:         || ((DListIterator) p).currentNode ==
9:             header)
10:        return;
11:    DListIterator p2
12:        =(DListIterator)(findPrevious(p));
13:    remove(p2);
14: }
15:
16: //return iterator pointing to location before p.
17: public Iterator findPrevious(Iterator p) throws
18: Exception {
19:     if ((p == null)
20:         || !(p instanceof DListIterator)
21:         || ((DListIterator) p).currentNode == null)
22:        return null;
23:
24:    DListIterator p1 = ((DListIterator) p);
25:    DListIterator p2 = new
26:    DListIterator(p1.currentNode.previousNode);
27:    return p2;
28: }
```

Figure 3-48: Code for removing data at a specified position.

In *findPrevious(Iterator p)*, we can easily find a node in front of the node marked by *p* by just following its *previousNode* pointer (line 26 from Figure 3-48). Once that position is identified, a *remove(Iterator p)* can be called to simply remove a node behind *p*. The overall process of *removeAt* therefore does not require any loop operation. Thus, it runs in constant time, $\Theta(1)$.

Method *head*, *tail*, and *append* for a doubly-linked list can use codes almost exactly the same as its singly-linked list counterpart. In fact, the only difference is the list type. This is because those codes utilized method *next* and *insert* to add new data to the list. These two methods are also implemented for our doubly-linked list, so they already dealt with extra pointers manipulation for us. Thus, codes using these methods work perfectly with doubly-linked lists.

Sparse Table

Storing data which can be arranged in row-column format usually requires a 2-dimensional array. However, some data set has a lot of empty data. This means that there will be wasted array slots reserved in memory. This kind of data set is called a sparse table. One way to avoid wasting such memory reservation is to use a set of linked lists to store only necessary data.

Let's say we have data of all gamers who play games on Steam, together with their achievement percentage for each game. We want to arrange data such that:

- For each gamer, we must be able to find all the games he plays and his achievement for each game.
- For each game, we must be able to find all gamers who play the game.

A 2-dimensional array representation of such data is shown in Figure 3-49. There are many empty slots because a gamer does not play every game and each game is not played by every gamer.

	Ann	Ben	Cathy	Don	Jim
D.D.		20%			72%
MTG	25%		74%		21%
Orc					
Sky					99%
T2	11%	33%			

Figure 3-49: Two-dimensional array representing games and players' progresses.

A linked list that corresponds to data set from Figure 3-49 is shown in Figure 3-50.

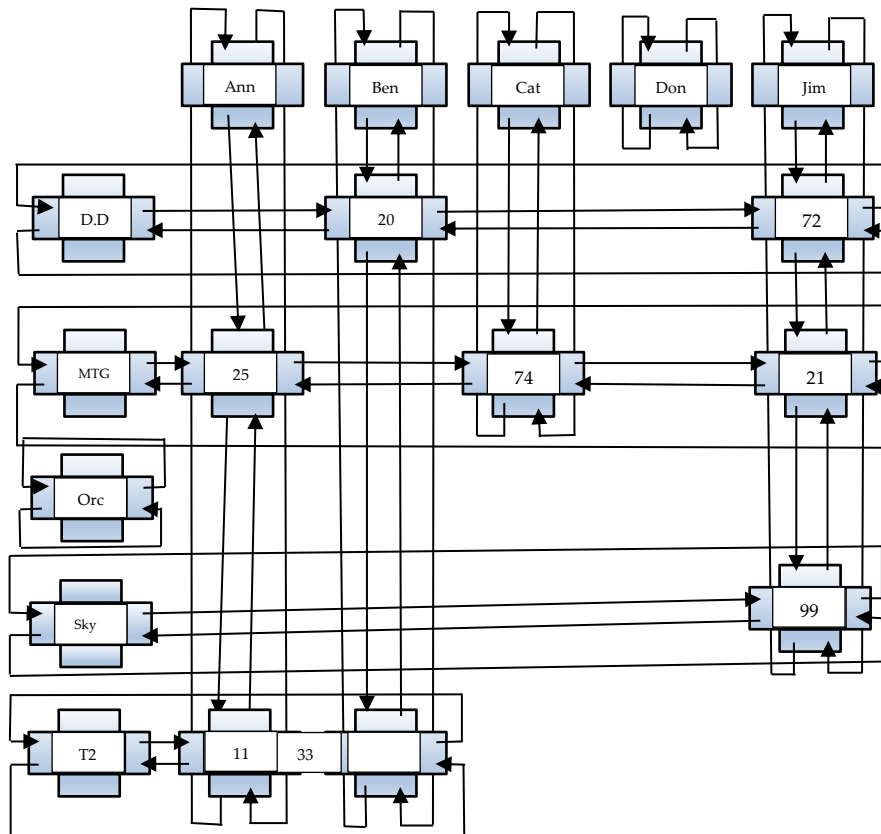


Figure 3-50: Linked list implementation of a sparse table.

The less data it has, the more memory saving we can achieve. Following links can still be slow, however. We can speed the search for game names and gamer names by adding direct links to game name and gamer name for each node.

Skip List

Searching for data in a linked list can be slow because we need to follow links through each and every node. One way to speed up the search is to implement a skip list. A node in a skip list can have more than one pointer, each one pointing to different nodes in the list (Figure 3-51).

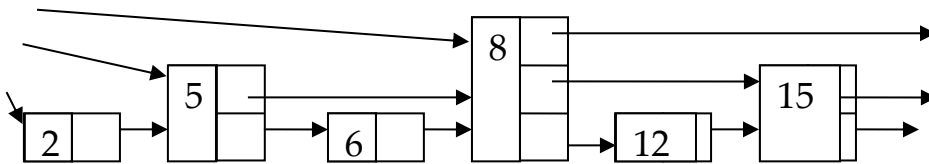


Figure 3-51: A skip list (shown with 1 direction pointers only in order to avoid confusion).

Hence in a sorted skip list, we can start by following the link that skips most data. Then if we go beyond the intended data, we can go back by one link and start searching again with a lower level link. In Figure 3-51, to find the number 15, we follow the link from the header to the node that stores number 8, but the link from the node that stores 8 goes to a node that stores a larger number than 15. We therefore start the search again from the node that contains 8, following links that skip fewer number of nodes. This time we successfully find 15.

Finding (or not finding) a required data can take $O(\log n)$ with a doubly-linked version of skip list. The skipping works the same way as a binary search.

Maintaining a skip list can be a problem though. If we want to maintain well distributed links for each level of links, we need to change all links when a data is inserted or removed from the list. This is impractical. Therefore, we resort to just maintain the number of links in each level when we add a new data.

Let's say we want to have 3 levels of links (level 0 to 2). level n^{th} starts at the $2^n - 1$ position (not counting header). Node level n links to 2^n $^{\text{th}}$ node to its right. A skip list with this scheme, with 5 data, is shown in Figure 3-52.

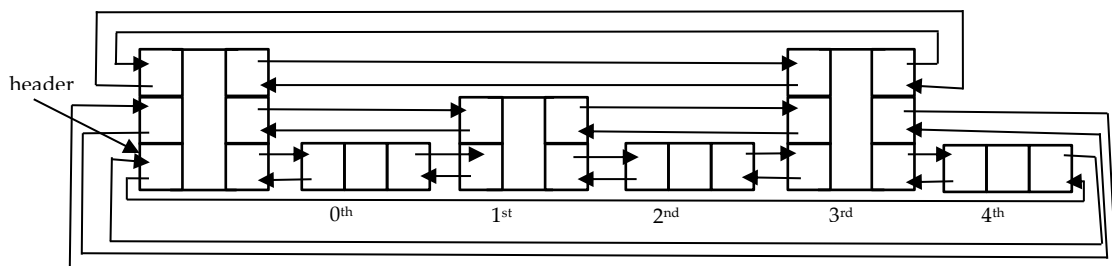


Figure 3-52: Doubly-linked skip list with 5 data.

If we are to expand the list in Figure 3-52 to have 20 nodes (not including header) while maintaining evenly distributed nodes of all level, we will have a list with the following nodes:

- Nodes level 0 (linked to its next node): 0th, 2nd, 4th, 6th, 8th, 10th, 12th, 14th, 16th, 18th node. There are 10 nodes in total.
- Nodes level 1 (linked to its next node and a node 2 places away): 1st, 5th, 9th, 13th, 17th node. There are 5 nodes in total.
- Nodes level 2 (linked to its next node, a node 2 places away, and a node 4 places away): 3rd, 7th, 11th, 15th, 19th node. There are 5 nodes in total.

The ratio of each node level from above is what we need to maintain when adding a new node (we do not change node type of any node when a node is removed because it is messy). We can achieve this by creating a random number between 1 and 20.

- If the number is between 1 and 10, we add node level 0.
- If the number is between 11 and 15, we add node level 1.
- If the number is between 16 and 20, we add node level 2.

In actual implementation, since there can be more than one next and previous pointers, you may want to implement array of pointers, as shown in a sample code in Figure 3-53.

```
1: class SKNode{
2:     int data;
3:     SKNode[] next;
4:     SKNode[] previous;
5: }
```

Figure 3-53: Sample code for a skip list node.

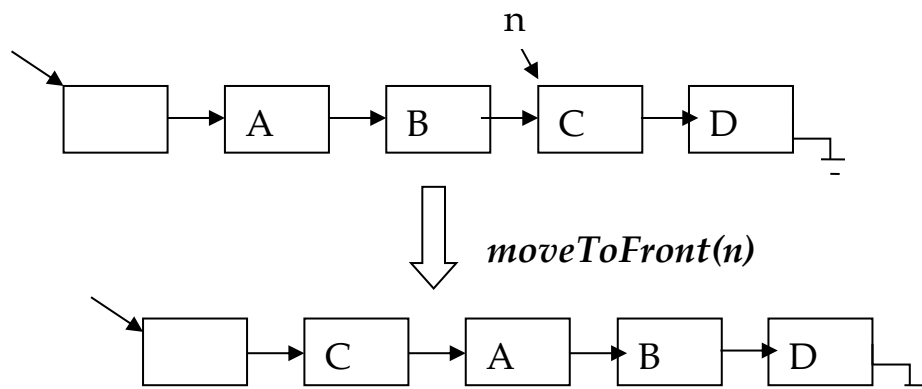
Exercises

1. Write method *public void insertAtFront(int x)* for class *CDLinkedList*. This method inserts a new list node, with data *x* inside, between header and other nodes.
2. Write method *public int removeAtLast()* for class *CDLinkedList*. This method removes the node at the last position from the list, and returns the data stored inside that node as the method's return value.
3. Write method *public CDLinkedList reverseList()* for class *CDLinkedList*. This method returns a new linked list that has all elements from **this** list, but the elements are arranged in reverse order. You are allowed to change **this** list.
4. Write method *public boolean isInFront(int x, int y)* for class *CDLinkedList*. This method returns true if *x* is stored in some node before *y* (when we search from left to right, starting from header). It returns false otherwise. If *x* or *y* is not in the list, this method returns false.
5. For class *LinkedList* (our singly-linked version), write method *public void setify()*. This method changes our

list by removing all duplicated data so that there is only one copy of each data.

6. For class *CDLinkedList*, write method *public void removeBefore(DListIterator p)*. This method removes a node before the node marked to by *p*, but it does not remove header. Do nothing if *p* is not valid.
7. For class *CDLinkedList*, write method *public void removeMin()*. This method removes the smallest data from the linked list.
8. For class *LinkedList*, write method *public void moveToFront(ListNode n)*. This method moves the content of *n* to the front of the list. Other contents' relative ordering remains unchanged.

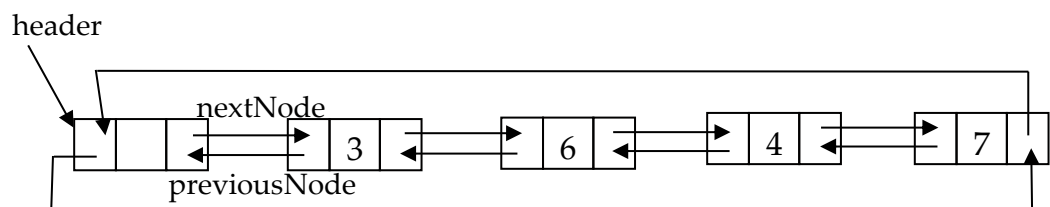
For example



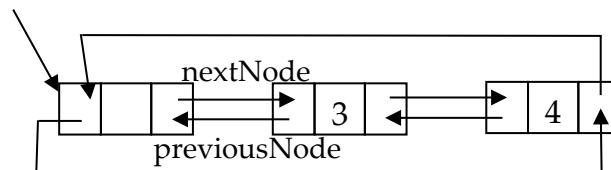
9. For class *CDLinkedList*, write method *public void clone(CDLinkedList in)*, which removed all data from this list and then copies all items from the input list in to itself. In this case if you change any data in the new

list, the data in the original input list must not be changed.

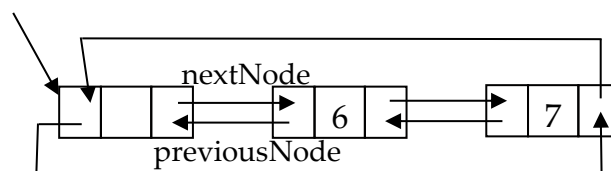
10. Write code for method `public CDLinkedList partition(int value)` of class `CDLinkedList`. This method removes all values greater than `value` from **this** list. The method returns a list containing all the removed values in order from left to right from the **this** list (or empty list if no value is removed). For example, if the original **this** list is:



then calling `partition(5)` will change **this** to:



And the returned list from the method will be:



11. Implement a polynomial (such as $5x^{36} - 3x^{17} + 8$) using class *LinkedList*. Write code to multiply 2 polynomials.
12. If the first element of a list has index $= 0$, explain how you would exchange the value of element with index number x and y in a list, provided that x and y are really legal positions in the given list (drawing can help your explanation). Write method ***public void swap(int x, int y)*** for class *LinkedList* that will perform such work.
13. Explain how to modify a list of numbers such that even numbers are in the front portion of the list and odd numbers are in the back portion (drawing can help). Write method ***public void evenOdds()*** for class *LinkedList* that performs such task. You cannot create another *LinkedList* or array.
14. For class *LinkedList*, write method ***public LinkedList specificElements(LinkedList C, int[] P)*** This method creates a new list by taking elements from C , as specified by their indices in P . For example, if P contains 1,3,4,6 the answer will be a linked list which has the 1st, 3rd, 4th, and 6th element from C . If a specified index does not exist in C , ignore it. P does not have to be sorted. Discuss the asymptotic runtime of your solution.
15. For class *CDLinkedList*, write method ***public CDLinkedList union(CDLinkedList a, CDLinkedList b)*** that creates a new list which is a result from the

union of a and b . a and b must remain unchanged. Only one copy of each data is allowed in a result list.

16. For class *CDLinkedList*, write method **public CDLinkedList intersect(CDLinkedList a, CDLinkedList b)** that creates a new list which is a result from the intersection of a and b . a and b must remain unchanged. Only one copy of each data is allowed in a result list.
17. For class *CDLinkedList*, write method **public CDLinkedList diff(CDLinkedList a, CDLinkedList b)** that creates a new list which has data which are in a but not in b . Only one copy of each data is allowed in a result list.
18. Illustrate how to swap two adjacent data in a *CDLinkedList* by only changing pointers.
19. For class *CDLinkedList*, write method **public void swapChunk(DListIterator start, DListIterator end, DListIterator p)**. This method changes our list by moving data from position $start$ to end (inclusive) to position in front of p . Assume that the list is not empty, all iterators actually point to positions in the list, $start$ is always to the left of end , and p is not a position between $start$ and end (inclusive).

Chapter 4 : Stack

A stack is a bucket. It stores data in layers. We can only insert and remove data from the top of the bucket. The way data can only be added or removed at the top is called LIFO (last in, first out). Figure 4-1 shows a stack:

- data *a*, *b* and *c* are stored inside.
- A new data will sit on top of *c*.
- *b* cannot be accessed unless *c* is removed first.
- *a* cannot be accessed unless *c* and *b* are removed.

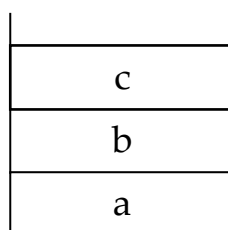


Figure 4-1: Stack with 3 data inside.

Stack Operations

Operations that we do with data on a stack are:

- Push: put a new data on top of the stack (Figure 4-2).
- Pop: remove the top most data (Figure 4-3).
- Top: read the data at the top without changing stack content.

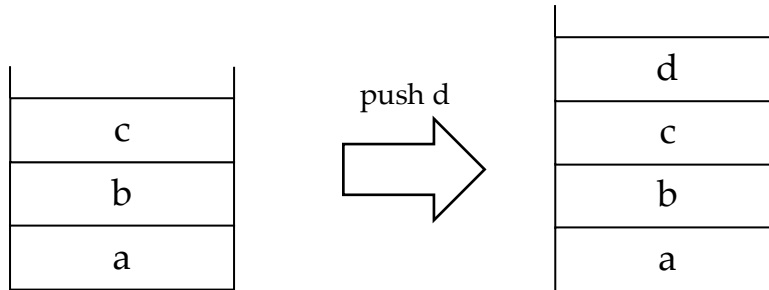


Figure 4-2: Pushing data d onto a stack.

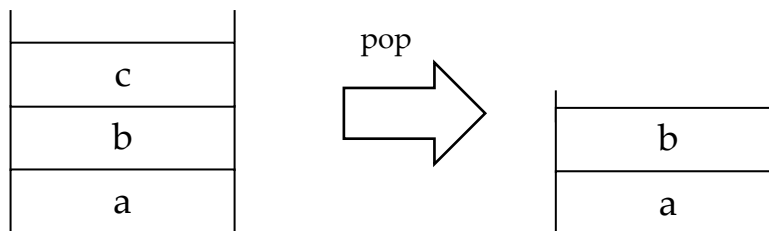


Figure 4-3: Popping data out of a stack.

Notable uses of Stack

Bracket Pairing

We can check to see if our program source code has the correct number/pairing of brackets. We do this by checking the source code character by character. Then:

- If we find an opening bracket, push it onto sack.
- If we find a closing bracket, pop data from stack.
 - If there is nothing in the stack to be popped, it means we have more closing brackets than opening brackets.

- If the popped bracket is not the same type as the closing bracket, we know we have an incorrect bracket type pairing.
- When the entire source code is read, if there are still opening brackets in the stack, we know we have too many opening brackets. Otherwise, the bracket pairings are correct.

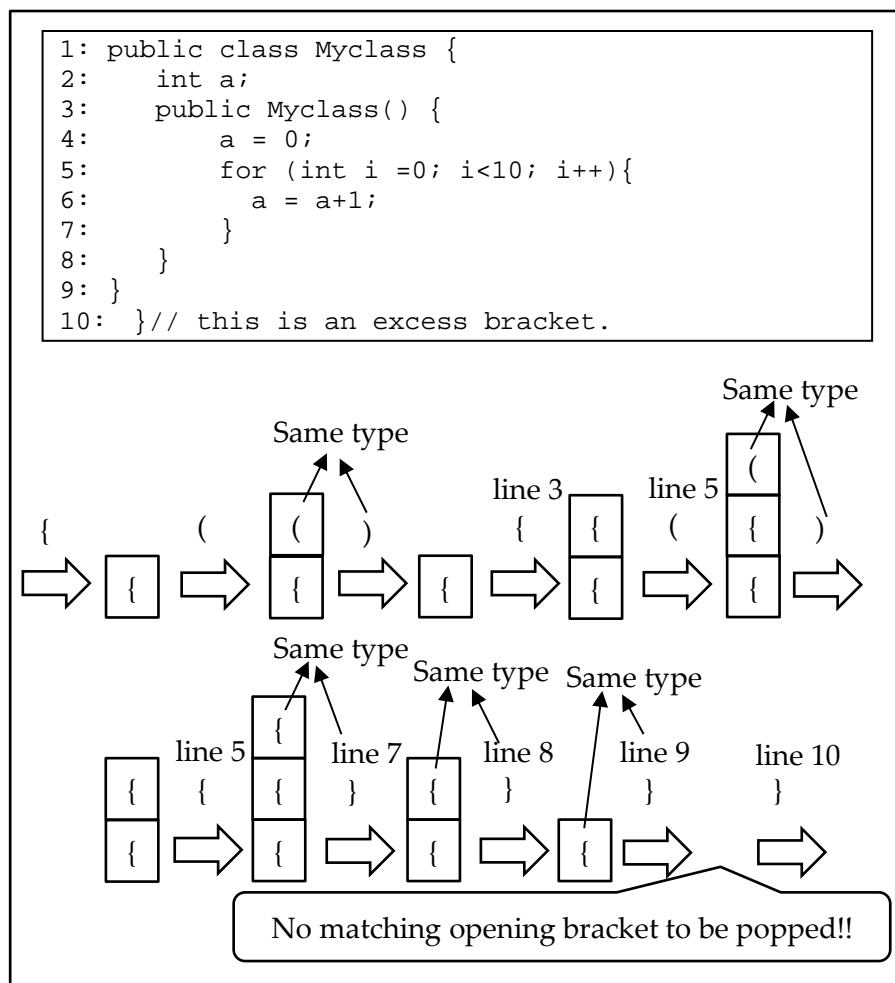


Figure 4-4: Processing brackets, with excess closing brackets.

Figure 4-4 shows each stage of data inside our stack after reading each bracket from a given code. Each thick arrow shows a bracket being read (lines of code is also given in most cases to help readers map the code and the picture in the figure).

The code has an excess closing bracket on line 10. This excess bracket is discovered when we try to pop an empty stack after reading that bracket.

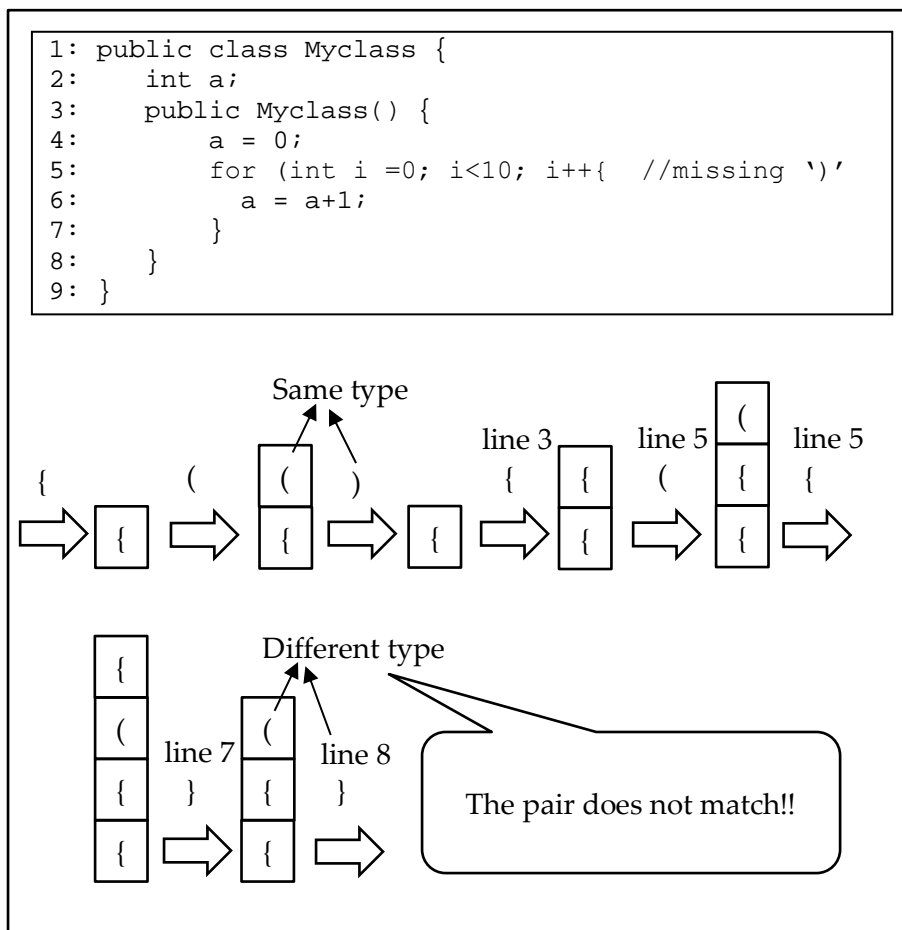


Figure 4-5: Processing brackets, with incorrect type pairing.

Figure 4-5 shows each stage of data inside our stack after reading each bracket, this time the code pairs different type of brackets because it is missing a bracket on line 5. When the process reads the closing curly bracket on line 8, before popping the stack, it checks the type of the bracket stored in the stack. And it discovers a different type of bracket. Therefore, the pairing is incorrect.

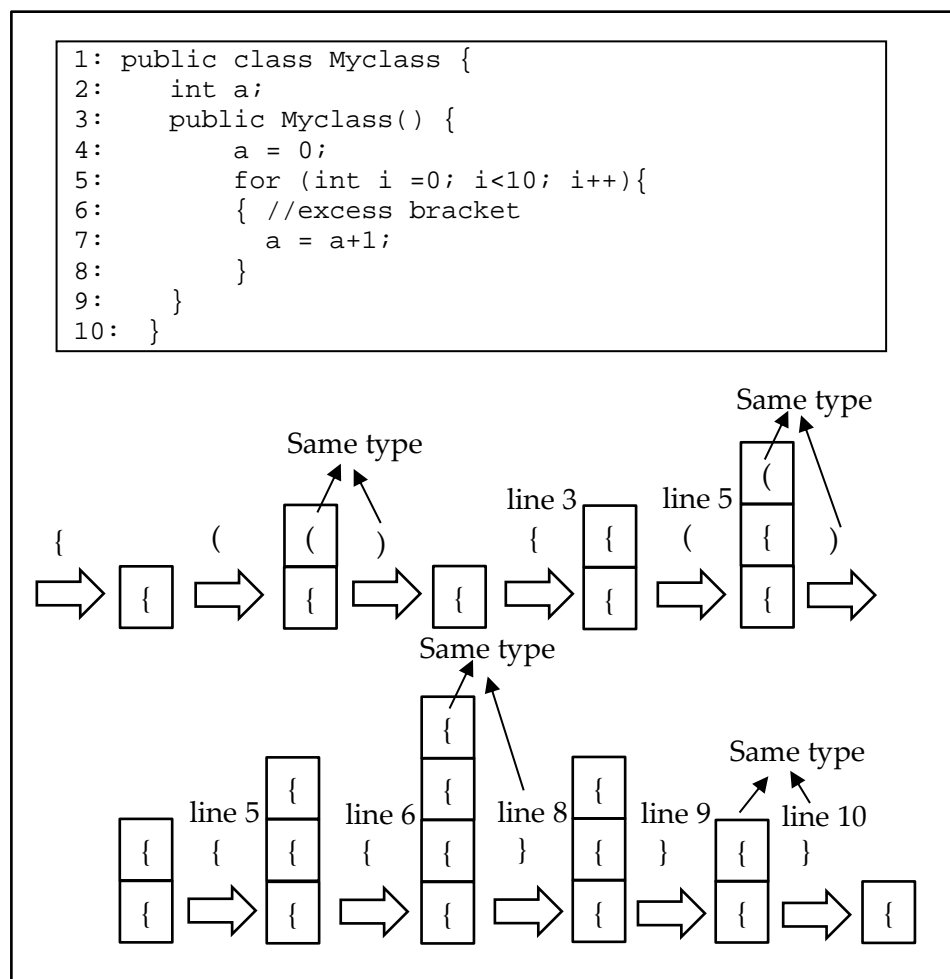


Figure 4-6: Processing brackets, with excess opening bracket.

Figure 4-6 shows each stage of data inside our stack after reading each bracket, this time the code has an extra opening bracket on line 6. When the process ends, the stack still has an opening bracket inside. Therefore, it means we run out of all closing brackets to pair with it. Hence it is an excess bracket.

Handling Data for Method Calls

In quite a few programming languages, including Java, a stack is used to store data of method calls (including nested calls). A data stored in this kind of stack is called a stack frame.

- When a method is called, a stack frame containing data specific to that method is created and pushed onto the stack.
- Data inside the top stack frame are visible to a language's runtime system (other globally visible data are also visible, but data from other method calls are not visible).
- When a method exits, a stack frame corresponding to that method is popped off the stack. Data that belong to that method are destroyed.

```
1:  public static void main(String[] args){
2:      int v = m1(2);
3:      System.out.print(v + m2(v));
4:  }
5:  public static int m1(int n){
6:      return n+5;
7:  }
8:  public static int m2(int i){
9:      i++;
10:     int v = i+m1(i);
11:     return v;
12: }
```

Figure 4-7: Example method calls.

Figure 4-7 shows an example of method calls. Table 4-1 to Table 4-3 shows what happens to the method stack when the code in Figure 4-7 is executed. The Execution Point column identifies the sequence of execution as the code gets run, while Stack Status shows what is on the stack at each execution point.

Each stack data (data for 1 method call) contains method parameters, local variables of the method, and a return address. A return address tells the runtime system where the execution should continue after a current method returns (For simplicity, we just use line number in Table 4-1 to Table 4-3).

Table 4-1: Various stages of stack for storing methods data when code in Figure 4-7 is executed (part 1).

Execution point	Stack status
Enter main.	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> args: null return address: system </div>
Enter m1(2) //line 2.	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> n: 2 return address: line 2 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> args: null return address: system </div>
return n+5 i.e. 2+5 (return value is kept by the system) and exit method.	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> args: null return address: system </div>
int v = the returned value.	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> v: 7 args: null return address: system </div>
Enter m2(7) //line 3.	<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> i: 7 return address: line 3 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 0 auto;"> v: 7 args: null return address: system </div>

Table 4-2: Various stages of stack for storing methods data when code in Figure 4-7 is executed (part 2).

Execution Point	Stack Status
i++	<div data-bbox="962 591 1307 745" style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> i: 8 return address: line 3 </div> <div data-bbox="962 745 1307 900" style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>
Enter m1(8) //line 10.	<div data-bbox="962 943 1307 1097" style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> n: 8 return address: line 10 </div> <div data-bbox="962 1097 1307 1252" style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> i: 8 return address: line 3 </div> <div data-bbox="962 1252 1307 1406" style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>
return n+5 i.e. 8+5 (return value is kept by the system) and exit method.	<div data-bbox="962 1447 1307 1601" style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> i: 8 return address: line 3 </div> <div data-bbox="962 1601 1307 1756" style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>

Table 4-3: Various stages of stack for storing methods data when code in Figure 4-7 is executed (part 3).

Execution Point	Stack Status
<p>int v = i + the returned value (this is not the same v as in the bottom of stack since it is created as part of the execution of m2)</p> <p>The value of i in the current stack is 8. And the return value is 13 from the last method call.</p>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> v: 21 i: 8 return address: line 3 </div> <div style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>
return v and exit method (return value (21) is kept by the system).	<div style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>
Enter print(v+ returned value).	<div style="border: 1px solid black; padding: 5px; margin-bottom: 5px;"> to print: 7+21 == 28 return address: line 3 </div> <div style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>
Exit print.	<div style="border: 1px solid black; padding: 5px;"> v: 7 args: null return address: system </div>
Exit main.	Nothing remains in the stack.

Postfix Calculation

A postfix expression is an arithmetic expression written by putting operands in front of their corresponding operator. Table 4-4 shows some expressions in their normal form (infix) and their corresponding postfix form.

Table 4-4: Expressions and their corresponding postfix form.

Infix Expression	Postfix Expression
$2+3$	$2\ 3\ +$
$5-3+2$	$5\ 3\ -\ 2\ +$
$7-4*3$	$7\ 4\ 3\ * \ -$
$(5-1)*3$	$5\ 1\ -\ 3\ *$
$((7+8)*9+5)*10$	$7\ 8\ +\ 9\ * \ 5\ +\ 10\ *$
$(7+(8*9)+5)*10$	$7\ 8\ 9\ * \ +\ 5\ +\ 10\ *$

It is easier for machines to evaluate postfix expression.

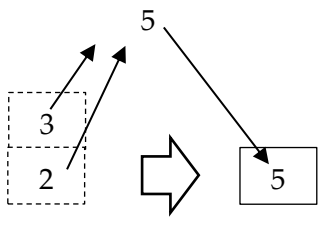
The machine evaluation of postfix expression, using a stack, is as follows:

- Read a token from a postfix notation.
 - If the token is actually a number, push that number onto the stack.
 - If the token is actually an operator, pop data off the stack to use with that operator (the number of data to pop depends on the number of parameters needed for that operator). After the calculation is completed, push a resulting data onto the stack.

- Repeat the above process until there is no more token to be read. At this point, a data on top of the stack is our calculated result.

Let us see an example. Let us perform the calculation $2+3$. Its postfix form is $2\ 3\ +$. Table 4-5 shows this operation step-by-step (top row to bottom row).

Table 4-5: Postfix calculation of $2+3$.

Token Read	Operation after Reading the Token	Stack Status
2	Push 2 onto stack.	<div style="border: 1px solid black; width: 40px; height: 30px; margin: 0 auto; display: flex; align-items: center; justify-content: center;">2</div>
3	Push 3 onto stack.	<div style="border: 1px solid black; width: 40px; height: 60px; margin: 0 auto; display: flex; flex-direction: column; align-items: center; justify-content: center;"> <div style="border: 1px solid black; width: 20px; height: 20px; margin-bottom: 5px;">3</div> <div style="border: 1px solid black; width: 20px; height: 20px;">2</div> </div>
+	Pop the top 2 data on stack to add them. Then put the result back on the stack.	

More complicated examples are shown in Table 4-6 and Table 4-7.

Now that you know how machines perform arithmetic calculations using a stack, you may wonder how machines get its arithmetic input in postfix form even

though we always input data into computers in infix form. The answer is – we have another algorithm, also using a stack, that can transform any infix arithmetic expression into its postfix counterpart.

Table 4-6: Calculation of $7\ 4\ 3\ * -$ (infix form is $7-4*3$).



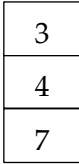
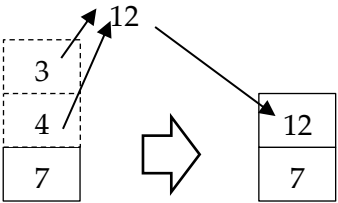
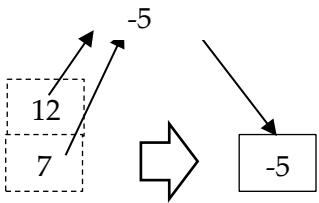

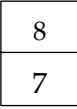
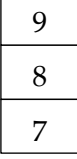
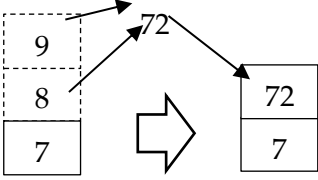
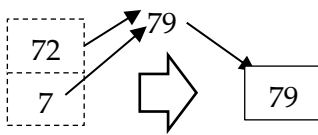
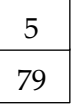
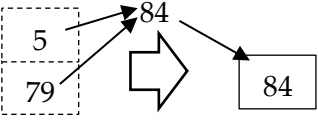
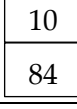
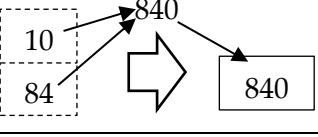
Token Read	Operation after Reading the Token	Stack Status
7	Push 7 onto stack.	
4	Push 4 onto stack.	
3	Push 3 onto stack.	
*	Pop the top 2 data on stack to multiply them. Then put the result back on the stack.	
-	Pop the top 2 data on stack to subtract them. Then put the result back on the stack.	

Table 4-7: Calculation of $7\ 8\ 9\ * + 5 + 10\ *$ (infix form is $(7+(8*9)+5)*10$).

Token Read	Operation after Reading the Token	Stack Status
7	Push 7 onto stack.	
8	Push 8 onto stack.	
9	Push 9 onto stack.	
*	Pop the top 2 data on stack to multiply them. Then put the result back on the stack.	
+	Pop the top 2 data on stack to add them. Then put the result back on the stack.	
5	Push 5 onto stack.	
+	Pop the top 2 data on stack to add them. Then put the result back on the stack.	
10	Push 10 onto stack.	
*	Pop the top 2 data on stack to multiply them. Then put the result back on the stack.	

Transforming Infix to Postfix Form

The algorithm is as follows:

- For each token:
 - If it is an operand, append it to the output.
 - If it is an operator
 - Pop operators on the stack and append them to the output if the current token has equal or less priority.
 - Otherwise, do not pop them.
 - Then push the new operator onto the stack.
- When finish reading the input:
 - pop all operators and append them to the output.

Table 4-8 to Table 4-10 show examples of infix to postfix transformations. Each table shows what happens when each token is read one by one (top row to bottom row).

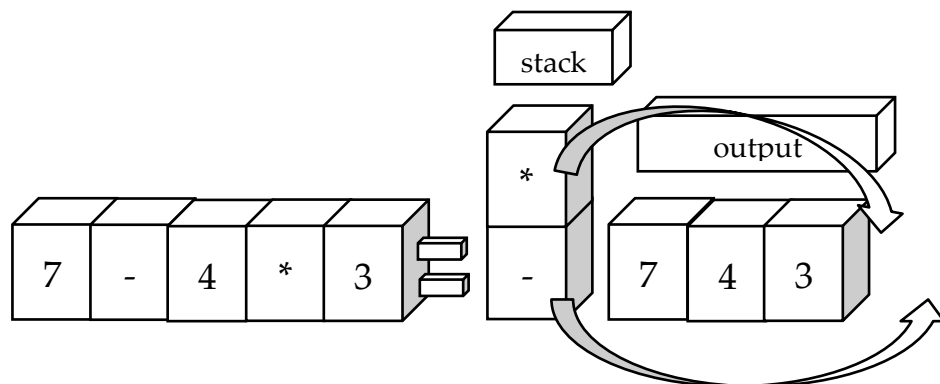


Table 4-8: Transforming 2+3 to its postfix counterpart.

Token Read	Operation after Reading the Token	Stack Status	Current Output
2	Append token to output.	empty	2
+	Push the operator onto stack.	+	2
3	Append token to output.	+	2 3
No more token	Pop all on stack and append to output.	empty	2 3 +

Table 4-9: Transforming 7-4*3 to its postfix counterpart.

Token Read	Operation after Reading the Token	Stack Status	Current Output
7	Append token to output.	empty	7
-	Push the operator onto stack.	-	7
4	Append token to output.	-	7 4
*	Do not pop anything because '*' has more priority than '-' on the stack. Push '*' onto the stack.	* -	7 4
3	Append token to output.	* -	7 4 3
No more token	Pop all on stack and append to output.	empty	7 4 3 * -

Table 4-10: Transforming 7+5-10 to its postfix counterpart.

Token Read	Operation after Reading the Token	Stack Status	Current Output
7	Append token to output.	empty	7
+	Push the operator onto stack.	+	7
5	Append token to output.	+	7 5
-	'-' has equal priority to '+' on the stack. Hence we pop '+' to the output, then push '-'.	-	7 5 +
10	Append token to output.	-	7 5 + 10
No more token	Pop all on stack and append to output.	empty	7 5 + 10 -

The algorithm can also be expressed as pseudocode, as seen in Figure 4-8. The pseudocode assumes the availability of function *isOperand*, which tests whether a token is an operand or an operator, and function *priority*, which outputs a priority value of a given token.

I hope the examples teach you some applications of stack in computing problems so that you understand why a data structure like stack exists. Before we move on to look at stack implementation, I would like to cover more details on this infix to postfix transformation.

```
1: String infix2postfix(String infix) {
2:     String[] tokenArray = infix.split("\\s");
3:     String postfix = ""; // our output string
4:     Stack s = new Stack();
5:     for (int i=0; i<tokenArray.length; i++) {
6:         String token = tokenArray[i];
7:         if(isOperand(token)){ //token is an operand
8:             postfix += token;
9:         } else { //token is an operator
10:            int pToken = priority(token);
11:            int pTop = priority(s.top());
12:            while(!s.isEmpty() && pToken <= pTop ){
13:                postfix += s.top();
14:                s.pop();
15:                pTop = priority(s.top());
16:            }
17:            s.push(token);
18:        }
19:    }
20:    while(!s.empty()) {
21:        postfix += s.top();
22:        s.pop();
23:    }
24:    return postfix;
25: }
```

Figure 4-8: Pseudocode for Infix to Postfix Transformation.

Our example so far did not address two common things found in arithmetic expressions. They are:

- brackets
- right associative operators

Let us look at brackets first. To group operations inside brackets, tokens between an opening bracket and its corresponding closing bracket should be processed just like they belong to a separate expression, independent of any part of the whole expression that comes before it or after it. We could use another stack to process the

bracketed expression separately, but it is also possible to use a single stack to do the job, as long as we make sure that:

- when an opening bracket is read, it is always pushed onto the stack.
- any operator (except the closing bracket) that follows the opening bracket does not cause the opening bracket to be popped from the stack.
- when a closing bracket is read, all data inside the stack down to the opening bracket are popped out and processed.

The above constraint ensures that a bracketed expression is treated as if it is processed on its own mini stack, without any interference to/from anything that comes before/after it.

- To make sure that an opening bracket is always pushed onto the stack, it must have the highest priority compared to any possible operator on top of the stack.
- To make sure that any operator (except the closing bracket) that follows the opening bracket does not cause the opening bracket to be popped from the stack, the opening bracket must have the lowest priority.
- To make sure that when a closing bracket is read, all data inside the stack down to the opening bracket are popped and processed, the closing

bracket must have the lowest priority, but higher priority than the opening bracket on the stack (so that it does not interfere with operators stored on the stack before the opening bracket. Then, we can detect the opening bracket and pop it from the stack ourselves).

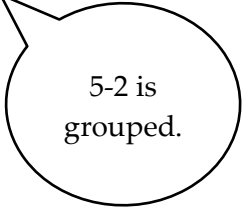
An opening bracket has the highest priority and the lowest priority at the same time! How can we make this work? The answer is: we create two priority values for each operator, one value is used when the operator is read from input, the other value is used when the operator is on the stack. Thus, an opening bracket can now have the highest priority when read from input, and the lowest priority when read from the stack. Table 4-11 shows priority values (when read from input and when read from the stack) of common arithmetic operators, including brackets.

Table 4-11: Operator Priority.

operator	+	-	*	/	()
Priority when read from input	3	3	5	5	9	1
Priority inside stack	3	3	5	5	0	None (never get stored)

To show how these priority values of brackets inside and outside the stack work, let us transform $10*(5-2)$ into its postfix form (see Table 4-12).

Table 4-12: Transforming $10*(5-2)$ into postfix form.

Token Read	Operation after Reading the Token	Stack Status	Current Output
10	Append token to output.	empty	10
*	Push the operator onto stack.	*	10
('(' from input has priority = 9, while '*' on the stack has priority = 5. Hence '(' is pushed onto the stack.	(*	10
5	Append token to output.	(*	10 5
-	'-' from input (priority = 3) has more priority than '(' on the stack (priority = 0). Hence we push '-' onto the stack.	- (*	10 5
2	Append token to output.	- (*	10 5 2
)	')' has lower priority than '-' on the stack but has higher priority than '(' on the stack. So '-' is popped to output. And now that '(' is on top of the stack, we pop it.	*	10 5 2 - 
No more token	Pop all on stack and append to output.	empty	10 5 2 - *

This way of priority arrangement keeps part of the expression in a bracket together. From Table 4-12, the subexpression (5-2) is guaranteed to become 5 2 – before getting processed with other subexpressions.

Let us try a more complicated example. This time we have nested brackets. The expression is $10 * ((7+8)*9)$. The transformation is shown in Table 4-13 to Table 4-15.

Table 4-13: Transforming an expression with nested brackets into its postfix form (Part 1).

Token Read	Operation after Reading the Token	Stack Status	Current Output
10	Append token to output.	empty	10
*	Push the operator onto stack.	*	10
('(' from input has priority = 9, while '*' on the stack has priority =5 . Hence '(' is pushed onto the stack.	(*	10
('(' from input has priority = 9, while '(' on the stack has priority =0. Hence the new '(' is also pushed onto the stack.	((*	10
7	Append token to output.	((*	10 7

Opening bracket always get pushed.

Table 4-14: Transforming an expression with nested brackets into its postfix form (Part 2).

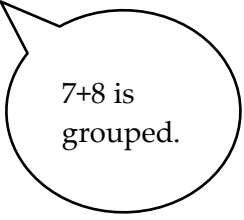
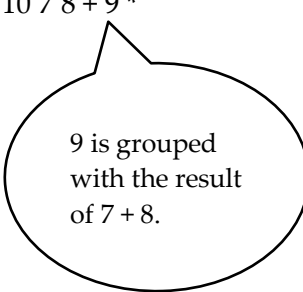
Token Read	Operation after Reading the Token	Stack Status	Current Output				
+	'+' from input (priority = 3) has more priority than '(' on the stack (priority = 0). Hence we push '+' onto the stack.	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>+</td></tr> <tr><td>(</td></tr> <tr><td>(</td></tr> <tr><td>*</td></tr> </table>	+	((*	10 7
+							
(
(
*							
8	Append token to output.	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>+</td></tr> <tr><td>(</td></tr> <tr><td>(</td></tr> <tr><td>*</td></tr> </table>	+	((*	10 7 8
+							
(
(
*							
)	')' has lower priority than '+' on the stack but has higher priority than '(' on the stack. So '+' is popped to output. And now that '(' is on top of the stack, we pop it.	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>(</td></tr> <tr><td>*</td></tr> </table>	(*	10 7 8 + 		
(
*							
*	'*' from input (priority = 5) has more priority than '(' on the stack (priority = 0). Hence we push '*' onto the stack.	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>*</td></tr> <tr><td>(</td></tr> <tr><td>*</td></tr> </table>	*	(*	10 7 8 +	
*							
(
*							
9	Append token to output.	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>*</td></tr> <tr><td>(</td></tr> <tr><td>*</td></tr> </table>	*	(*	10 7 8 + 9	
*							
(
*							

Table 4-15: Transforming an expression with nested brackets into its postfix form (Part 3).

Token Read	Operation after Reading the Token	Stack Status	Current Output
)	') ' has lower priority than ' * ' on the stack but has higher priority than ' (' on the stack. So ' * ' is popped to output. And now that ' (' is on top of the stack, we pop it.	<div style="border: 1px solid black; display: inline-block; padding: 2px;">*</div>	10 7 8 + 9 * 
No more token	Pop all on stack and append to output.	empty	10 7 8 + 9 * *

Our pseudocode for infix to postfix transformation, after adding inside-outside stack priorities, is shown in Figure 4-9. All changes from Figure 4-8 are indicated using bold texts. Now, finding priority values when data is read from input and when data is on the stack require different functions.

```

1:  String infix2postfix(String infix) {
2:      String[] tokenArray = infix.split("\\s");
3:      String postfix = ""; // our output string
4:      Stack s = new Stack();
5:      for (int i=0; i<tokenArray.length; i++) {
6:          String token = tokenArray[i];
7:          if(isOperand(token)){ //token is an operand
8:              postfix += token;
9:          } else { //token is an operator
10:             int pToken = outsidePriority(token);
11:             int pTop = insidePriority(s.top());
12:             while(!s.isEmpty() && pToken <= pTop ){
13:                 postfix += s.top();
14:                 s.pop();
15:                 pTop = insidePriority(s.top());
16:             }
17:             if(token == "(")
18:                 s.pop(); // pop "(" on top of stack
19:             else
20:                 s.push(token);
21:         }
22:     }
23:     while(!s.empty()) {
24:         postfix += s.top();
25:         s.pop();
26:     }
27:     return postfix;
28: }

```

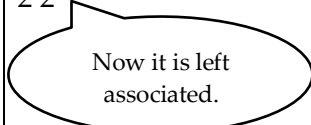
Figure 4-9: Pseudocode for Infix to Postfix Transformation, after adding inside-outside stack priorities.

The last remaining issue for this postfix transformation is right associative operators. Our algorithm currently does not support right associative operators. For example, 2^{2^3} (written as 2^{2^3}) should be $2\ 2\ 3\ \wedge\ \wedge$ in its postfix form because the rightmost operation must be carried out first. Our algorithm, so far, regards the same operator (apart from brackets) to have the same priority when read from input and when on stack. Therefore, the

postfix form of 2^2^3 by our algorithm will be $2 2 ^ 3 ^$, which is incorrect (see Table 4-16).

This can easily be fixed without any change to our code, however, by giving right associative operators higher priorities when it is outside the stack (being read from input), so that the same operator does not get popped out to our output earlier than it should.

Table 4-16: Incorrect postfix transformation due to priorities forcing left association.

Token Read	Operation after Reading the Token	Stack Status	Current Output
2	Append token to output.	empty	2
^	Push the operator onto stack.	<div style="border: 1px solid black; padding: 2px; display: inline-block;">^</div>	2
2	Append token to output.	<div style="border: 1px solid black; padding: 2px; display: inline-block;">^</div>	2 2
^	'^' has equal priority to '^' on the stack. Hence we pop '^' to the output, then push '^'.	<div style="border: 1px solid black; padding: 2px; display: inline-block;">^</div>	2 2 ^ 
3	Append token to output.	<div style="border: 1px solid black; padding: 2px; display: inline-block;">^</div>	2 2 ^ 3
No more token	Pop all on stack and append to output.	empty	2 2 ^ 3 ^

The corrected priority of '^' is shown in Table 4-17. The correct postfix transformation of 2^2^3 due to our fix is shown in Table 4-18.

Table 4-17: Operator Priority, with right associative operator '^'.

operator	+	-	*	/	()	^
Priority when read from input	3	3	5	5	9	1	8
Priority inside stack	3	3	5	5	0	None	7

Table 4-18: Correct postfix transformation after fixing right associative operator.

Token Read	Operation after Reading the Token	Stack Status	Current Output
2	Append token to output.	empty	2
^	Push the operator onto stack.	^	2
2	Append token to output.	^	2 2
^	'^' has more priority to '^' on the stack. Hence we push '^' onto stack.	^ ^	2 2
3	Append token to output.	^ ^	2 2 3
No more token	Pop all on stack and append to output.	empty	2 2 3 ^ ^

Now that we have seen some uses of stack, let us investigate how we can implement it. Figure 4-10 shows methods available from our implementation in this

chapter (our implementation is a stack that stores integer data). To summarize, we have the following methods:

- *isEmpty()*: returns true if our stack does not store any data. Otherwise, it returns false.
- *isFull()*: returns true if our stack reaches its maximum capacity. Otherwise, it returns false.
- *makeEmpty()*: gets rid of all data stored inside our stack.
- *top()*: returns data on top of our stack (the stack does not change). The method throws an exception if there is no data to be returned.
- *pop()*: removes data on top of our stack. The method throws an exception if there is no data to be popped.
- *push(int data)*: put a given data on top of our stack. the method throws an exception if the the push is somehow unsuccessful (caused by stack being full or from other reasons).

```
1: public interface MyStack {
2:     public boolean isEmpty();
3:     public boolean isFull();
4:     public void makeEmpty();
5:     public int top() throws Exception;
6:     public void pop() throws Exception;
7:     public void push(int data) throws Exception;
8: }
```

Figure 4-10: Stack Operations (interface).

Implementing a Stack with Array

Figure 4-11 to Figure 4-18 show our implementation, the class *StackArray*. We split the code into several figures so that we can explain each part of the code separately.

```
1:  public class StackArray implements MyStack{
2:      private int[] theArray;
3:      private int currentSize;
4:      private static final int DEFAULT_SIZE = 10;
5:
6:      public StackArray(){ // create an empty stack
7:          this(DEFAULT_SIZE);
8:      }
9:
10:     public StackArray(int intendedCapacity){
11:         theArray = new int[intendedCapacity];
12:         currentSize = 0;
13:     }
14:
15:     public int[] getTheArray() {
16:         return theArray;
17:     }
18:
19:     public void setTheArray(int[] theArray) {
20:         this.theArray = theArray;
21:     }
22:
23:     public int getCurrentSize() {
24:         return currentSize;
25:     }
26:
27:     public void setCurrentSize(int currentSize) {
28:         this.currentSize = currentSize;
29:     }
30:     //continued in Figure 4-12.
```

Figure 4-11: Stack implemented by array (fields, constructors, get, set).

Figure 4-11 shows fields, constructors, and get/set methods of our stack. Our stack contains an array

theArray to store data. It has variable *currentSize* to keep track of how many data are stored the stack.

Figure 4-12 shows method *isEmpty*, *isFull*, and *makeEmpty*. Method *isEmpty* and *isFull* can easily check *currentSize* to find out the number of data stored inside the stack, without having to check the array. Method *makeEmpty* works by resetting *theArray* and *currentSize*.

```
1:    public boolean isEmpty(){
2:        return currentSize ==0;
3:    }
4:
5:    public boolean isFull(){
6:        return currentSize == theArray.length;
7:    }
8:
9:    public void makeEmpty(){
10:        theArray = new int[DEFAULT_SIZE];
11:        currentSize =0;
12:    }
13: //continued in Figure 4-13.
```

Figure 4-12: *isEmpty()*, *isFull()*, and *makeEmpty()* of stack implemented with array.

Figure 4-13 shows method *top*, which returns the topmost value stored in our stack. From line 4, it can be seen that:

- Our top of stack is at position *currentSize-1* in the array.
- Data at position 0 in the array is considered to be on the bottom of stack.

```

1:    public int top() throws Exception{
2:        if(isEmpty())
3:            throw new Exception();
4:        return theArray[currentSize-1];
5:    }
6:    //continued in Figure 4-15.

```

Figure 4-13: Code of method *top*, for stack implemented with array.

Let us look at an example in Figure 4-14. Our stack, although has *theArray* that contains 5 data, has *currentSize* = 3. This means only data in slot 0, 1 and 2 are regarded as data on the stack. The data that is regarded as on the bottom of the stack is stored at position 0 in the array. The top data on the stack is at position $currentSize - 1 = 3 - 1 = 2$.

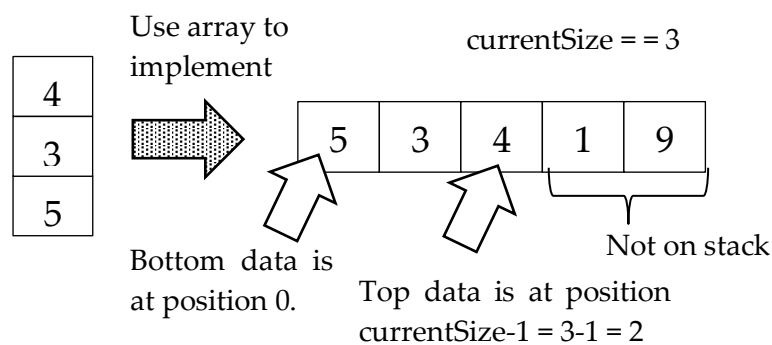


Figure 4-14: top and bottom of stack of size 3 (implemented with array).

Figure 4-15 shows the code for method *pop*. Its main operation is just decrementing *currentSize* by 1. Thus the top of stack changes position.

```

1: public void pop() throws Exception{
2:     if(isEmpty())
3:         throw new Exception();
4:     currentSize--;
5: }
6: //continued in Figure 4-17.

```

Figure 4-15: Code of method *pop*, for stack implemented with array.

Figure 4-16 shows the effect of *pop()* when applied to our stack implementation. Since *currentSize* is reduced, our top of stack (position *currentSize - 1*) is moved to the left of the previous top data on the stack. This means 4 is no longer regarded as on the stack.

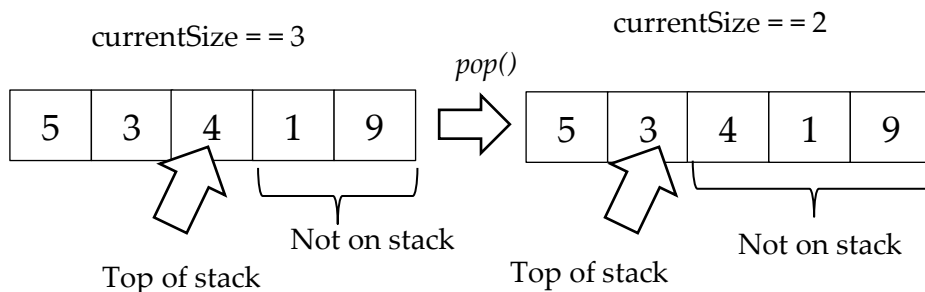


Figure 4-16: Popping data from a stack implemented with array.

Figure 4-17 shows code of method *push*. If the stack is already full (the array is full), we expand *theArray* to twice its original size. Then we simply overwrite data in the slot next to our top data and change *currentSize* so that the top data on the stack becomes that new data.

Figure 4-18 shows what happens when 7 is pushed onto a stack that originally stores 2 integers.

```

1:   public void push(int data) throws Exception{
2:       if(isFull())
3:           doubleCapacity();
4:       theArray[currentSize] = data;
5:       currentSize++;
6:   }
7:
8:   public void doubleCapacity(){
9:       //resize array to twice the original size
10:      int[] temp = new int[theArray.length*2];
11:      System.arraycopy(theArray, 0, temp, 0,
12:                      theArray.length);
13:      theArray = temp;
14:  }
15: } //end of class StackArray

```

Figure 4-17: Code of method *push*, for stack implemented with array.

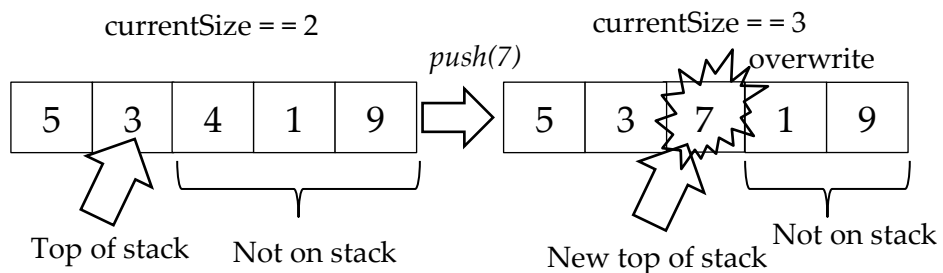


Figure 4-18: Pushing data onto stack implemented with array.

Implementing a Stack with Linked List

In this section, a circular doubly-linked list is used to store data as our stack. The first data of the list is regarded as the data on top of our stack, while the very last data in the list is considered to be at the bottom of our stack (see Figure 4-19). Therefore, stack operations

used in this implementation mainly involves adding/removing data from the node after *header*.

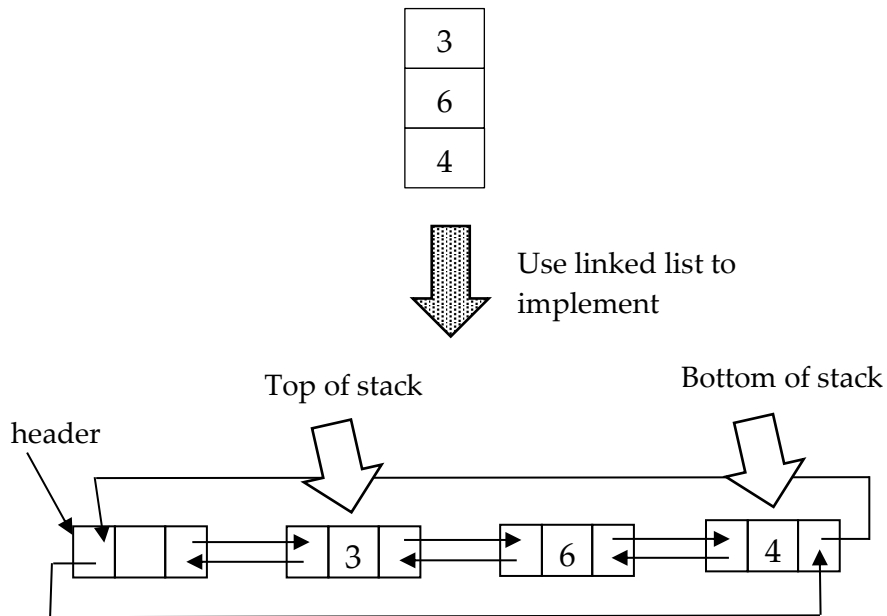


Figure 4-19: Linked list used as stack.

Our implementation, class *StackLinkedList*, is shown in Figure 4-20 to Figure 4-26.

Figure 4-20 shows field and constructors for this linked list implementation. We only have one field, *theList*, which is a circular doubly-linked list from chapter 3. This list will be used as our stack, as seen in Figure 4-19. There are two constructors. A default constructor (line 5-7) just creates an empty linked list. The other constructor is a copy constructor, which takes a linked list and copies each data from that list to *theList*. By copying data into

our stack, any change in *theList* does not affect the input list, and vice versa.

```
1:     public class StackLinkedList implements MyStack{
2:         private CDLinkedList theList;
3:
4:         // create an empty stack
5:     public StackLinkedList(){
6:         theList = new CDLinkedList();
7:     }
8:
9:     public StackLinkedList(CDLinkedList l) throws
10:    Exception {
11:         super();
12:         DListIterator iParam;
13:         iParam = new DListIterator(l.header);
14:         DListIterator iThis;
15:         iThis = new DListIterator(theList.header);
16:         while (iParam.hasNext()) {
17:             int v = iParam.next();
18:             if (iParam.currentNode == l.header)
19:                 return;
20:             theList.insert(v, iThis);
21:             iThis.next();
22:         }
23:     }
24: //continued in Figure 4-21.
```

Figure 4-20: Code for stack implemented with circular doubly-linked list (fields and constructors).

Figure 4-21 shows code for *isEmpty()*, *isFull()*, and *makeEmpty()* in this linked list implementation. Method *isEmpty* is performed by checking if the linked list is empty. Method *isFull* always returns false because there is no predefined space when we use a linked list. Method *makeEmpty* also makes use of *makeEmpty* of linked list. Mainly, we are just calling linked list methods.

```
1: public boolean isEmpty(){
2:     return theList.isEmpty();
3: }
4:
5: public boolean isFull(){
6:     return false;
7: }
8:
9: public void makeEmpty(){
10:    theList.makeEmpty();
11: }
12: //continued in Figure 4-22.
```

Figure 4-21: *isEmpty()*, *isFull()*, and *makeEmpty()* for stack implemented with circular doubly-linked list.

Figure 4-22 shows code for method *top* of this linked list implementation. The method throws exception if the stack is empty, since there is no data to return. Otherwise, it returns data stored in the node next to header, which we consider to be at the top of our stack (as illustrated in Figure 4-19, the returned data is 3).

```
1: public int top() throws Exception{
2:     if(isEmpty())
3:         throw new Exception();
4:     return theList.header.nextNode.data;
5: }
6: //continued in Figure 4-23.
```

Figure 4-22: *top()* for stack implemented with circular doubly-linked list.

Figure 4-23 shows our code for the linked list implementation of method *pop*. Again, the method throws an exception if there is no data on our stack. If there is a data on top of our stack, we pop it out by calling *remove* method of our linked list implementation to remove the node after *header* (see illustration in Figure

4-24), thus effectively removing our top data from the stack.

```

1: public void pop() throws Exception{
2:     if(isEmpty())
3:         throw new Exception();
4:     Iterator itr;
5:     itr = new DListIterator(theList.header)
6:     theList.remove(itr);
7: }
8: //code continued in Figure 4-25.

```

Figure 4-23: *pop()* for stack implemented with circular doubly-linked list.

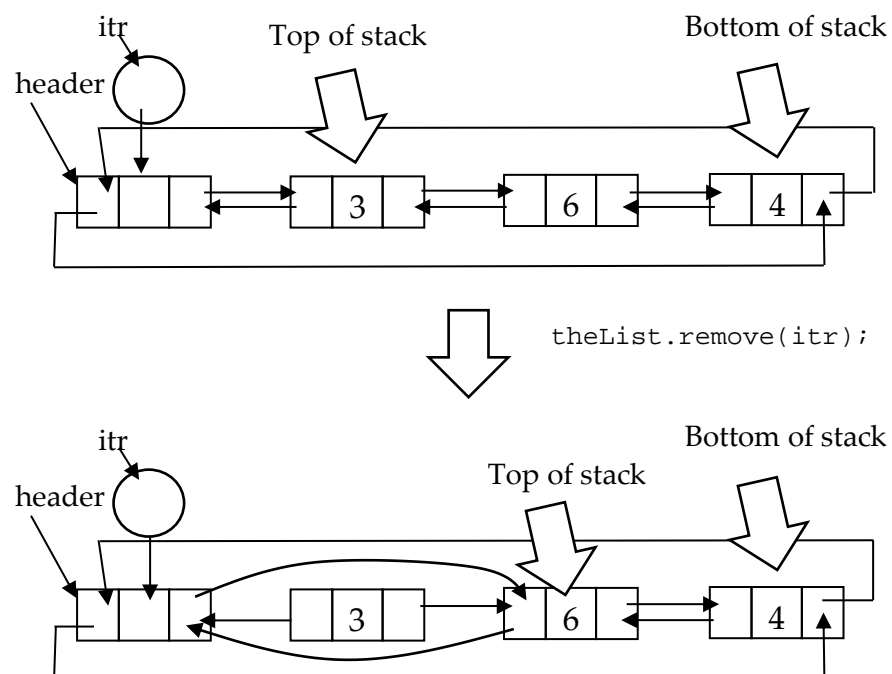


Figure 4-24: Removing the top of stack in linked list implementation.

Figure 4-25 shows code for method *push* in our linked list implementation. Similar to method *pop*, this method

mainly calls method of linked list. For this particular operation, it calls *insert* method of a linked list, to insert a new data next to *header* (illustrated in Figure 4-26).

```

1:     public void push(int data) throws Exception{
2:         Iterator itr;
3:         itr = new DListIterator(theList.header)
4:         theList.insert(data, itr);
5:     }
6: } // end of class StackLinkedList

```

Figure 4-25: Method *push* for stack implemented with circular doubly-linked list.

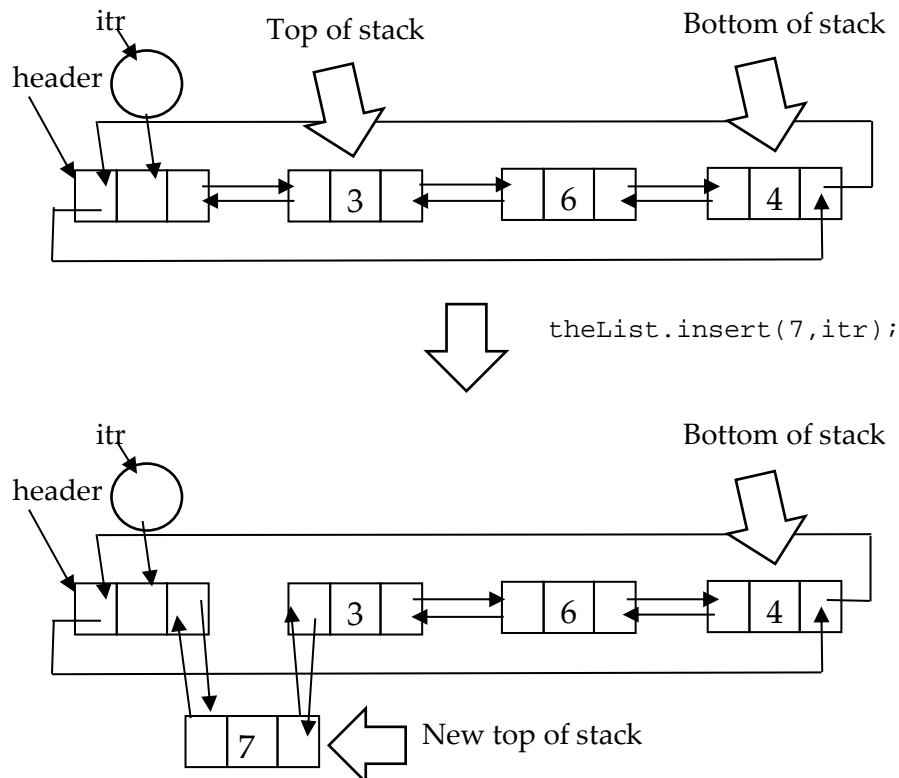


Figure 4-26: Pushing new data onto stack implemented with circular doubly-linked list.

There we are. We have covered stack's usage and its implementations. Now it is time for you to test your knowledge.

Exercises

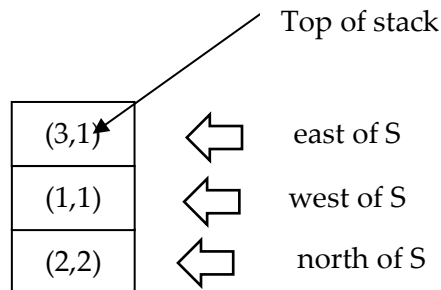
1. There is a maze

```
1 1 1 1 1 1
1 1 1 0 0 1
1 0 0 0 F 1
1 0 S 0 1 1
1 1 1 1 1 1
```

The number 1 represents wall and number 0 represents walkway. Let S be the starting point and F be the end point of travel (they are also walkways). We can systematically find a way from S to F by:

- recording the coordinate (X, Y) that we can travel to (at that moment in time) onto a stack. The order of storage is north, south, west, and east of the current position respectively (Let the coordinate of the most left-bottom number 1 be (0, 0)).

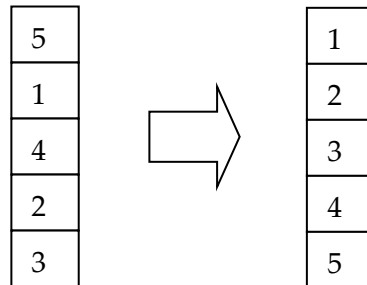
The original state of our stack will be:



- Walking is done by popping stack and then moving the current position to the popped coordinate. Then push the information of empty spaces surrounding that coordinate onto the stack (using north, south, west and east again) (we never push the coordinates that we have visited). We repeat this until we reach the destination. When the destination is reached, we do not push anything onto the stack.

What is the final stage of the stack?

2. Explain, step by step, how you can sort integers stored in a stack (after sorting, the smallest value must be at the top of the stack), using only one additional stack and two integer variables. You are not allowed to create array, linked list, or any data structure that can store a collection of values. A starting and ending state of an example stack are given below:



3. Write method:

public void addNoDuplicate (StackLinkedList s2) of class *StackLinkedList*. This method removes and pushes all contents (except for those that **this** already has) from *s2* into **this**. Contents that are duplicated must remain in *s2*, in their original ordering. **You are not allowed to create arrays, linked lists, trees and other kinds of data structures except *StackLinkedList***. Give the estimated running time of your implementation.

4. Assume we are using stack from class *Stack*, which has the code of all methods defined in the following Java interface (class *Stack* also has a working default constructor):

```
public interface MyStack {
    public boolean isEmpty();
    public boolean isFull();
    public void makeEmpty();

    //Return data on top of stack.
    //Throw exception if the stack is empty.
    public int top() throws Exception;
```

```
//Remove data on top of stack.
//Throw exception if the stack is empty.
public void pop() throws Exception;

//Add new data on top of stack.
//Throw exception if the operation is somehow
//unsuccessful.
public void push(int data) throws Exception;
}
```

We are using stack in our own class *TestStack*, which is:

```
Class TestStack{
    Stack s;
    public void removeDup(){
        // You have to write code for this method.
    }
}
```

You are to implement method *removeDup*, which removes duplicated data from *s*. For example, if the original data inside *s* is

1
4
3
1
3

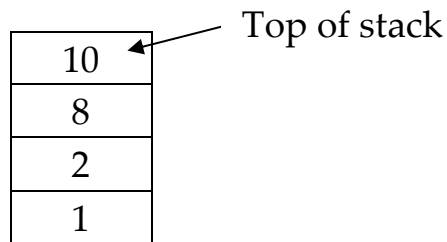
Then stack *s* after the method is called, is

1
4
3

Where 1, 4, and 3 can be in any order on the stack, i.e. order does not matter. In other words, method *removeDup* makes a **set** out of existing data on *s*.

- You do not know the internal workings of *s*, so you can only use methods provided by *MyStack* interface.
 - You are allowed to create primitive type variables.
 - You are not allowed to create non-primitive type variables, or any data structure, except *Stack(s)*.
- a. Explain, with illustrated example, the inner workings of *removeDup()*. Your explanation should be clear and step-by-step.
 - b. Write code for *removeDup()*.
5. For the same stack as in the previous question, write method *removeMin()*, which removes the smallest value (and any copies of it) from the stack. Other values must remain in their original order. You are only allowed to create primitive type variables and another stack.
6. For the same stack as in the previous question, write method *removeBottom()*, which removes the data at the bottom of the stack. Other values must remain in their original order. You are only allowed to create primitive type variables and another stack.

7. A palindrome is a sequence of integers (or letters) that reads the same left-to-right and right-to-left. For example, "abadacadaba", "1234321". Write your explanation on how to use a stack to check whether a given string is a palindrome.
8. Assume that values are always sorted from large to small in our stack, write the following methods of class *StackArray*:
 - *public void putIn(int x)*: this method adds number, *x*, into the stack. After this new value is added, the stack must remain sorted. For example, if *putIn(5)* is called by the following stack:



The resulting stack will be:



You are only allowed to create primitive variables and another stack. You are not allowed to create lists, arrays, or other data structures.

9. We have an infix expression $a*(b/c) - b*d$. If we use a stack to convert this expression to its postfix form, what will be left on the stack after we just read the last input? What is on the stack at that point in time?
10. Convert $a+(b-c)*(d-e)$ to its postfix form using stack. Illustrate this operation step by step.
11. Given a postfix expression $1\ 2\ 5\ *\ 4\ 2\ +\ +\ *\ 3\ -\ .$ Illustrate, step by step, how a resulting value can be evaluated using stack.
12. Write *factorial(int n)* that calculates a factorial of integer n , using only recursion. Draw what happens to stack frames of methods when *factorial(3)* is called from *main* method.
13. Compare asymptotic runtime of all stack methods from class *StackArray* and *StackLinkedList*.
14. Write method *int power(int x, int y)* for class *MyCalculation* (this is a newly created class), which calculates the value of x^y , using class *StackLinkedList*.

Chapter 5 : Queue

A queue is a data structure that stores a sequence of data. It is very much like a list, but it has extra restrictions:

- Remove (also called “dequeue”) can only be done on the first data of the sequence.
- A new data can be added (also called “enqueued”) to the queue only after the last data.

The way data can only be added or removed this way is called FIFO (first in, first out). It is similar to how people queue for services (see Figure 5-1).

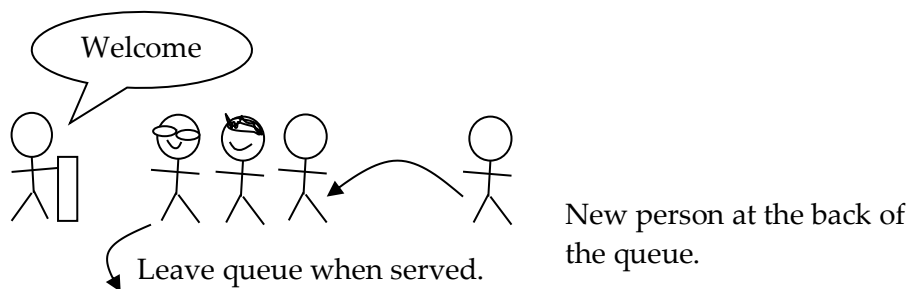


Figure 5-1: Queueing for services.

Queue Operations

Common operations that we do with a queue are as follows:

- *front()*: return the very first data in the queue.
- *back()*: return the very last data in the queue.

- *removeFirst()*: remove the first data from the queue. It returns that data. This is “dequeue” (illustrated in Figure 5-2).
- *insertLast(data)*: add new data following the current last data. This is “enqueue” (illustrated in Figure 5-3).
- *isEmpty()*: check if the queue stores no data. It returns true if the queue does not store any data and false otherwise.
- *isFull()*: check if the queue has no more space to store data.
- *size()*: return the number of data currently stored in the queue.

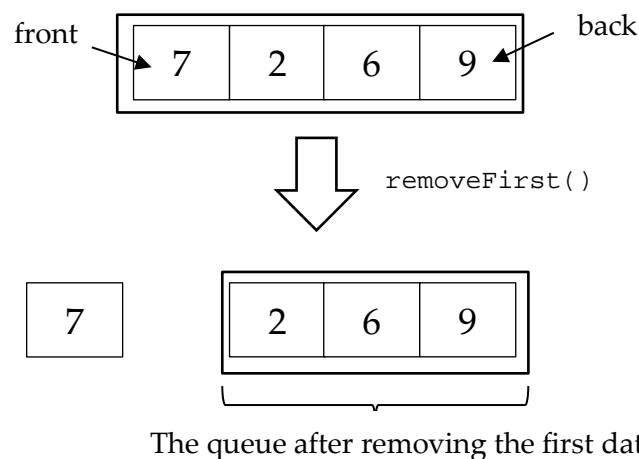
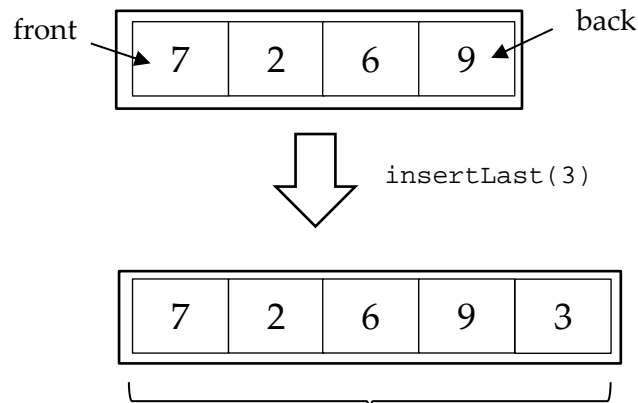


Figure 5-2: Dequeueing the first data from a queue.



The queue after adding a new data

Figure 5-3: Enqueueing a new data.

The operations can be put into a Java interface (named *MyQueue*) (see Figure 5-4). This queue is for integer.

```

1: public interface MyQueue {
2:
3:     //return the first data.
4:     public int front() throws Exception;
5:
6:     //return the last data.
7:     public int back() throws Exception;
8:
9:     //remove the first data (return its value too).
10:    public int removeFirst() throws Exception;
11:
12:    //insert new data after the last data.
13:    public void insertLast(int data) throws Exception;
14:
15:    //check if the queue is empty.
16:    public boolean isEmpty();
17:
18:    //check if the queue has no more space to store new
19:    //data.
20:    public boolean isFull();
21:
22:    //return the number of data currently stored in the
23:    //queue.
24:    public int size();
25: }

```

Figure 5-4: Interface for queue storing integer data.

Implementing a Queue with Array

In this section, we show our array implementation for a queue that stores integer data. Let us first go through the concept.

- We use array to store data.
- To manage enqueueing and dequeueing, we need the following variables:
 - *front*: an integer that is an index of the front data.
 - *size*: an integer indicating the number of currently stored data.

Let us name our array *theArray*. By having *front* and *size*, the following methods can easily be implemented:

- *front()*: we can just return *theArray[front]*.
- *isEmpty()*: we can simply check if *size* == 0. Our *theArray* will always have slots, so checking for *size* is the only way to find out if our queue is empty.
- *isFull()*: we can simply check whether *size* == *theArray.length*. This means there is absolutely no space left anywhere in *theArray*.
- *size()*: we can simply return the value of the variable *size*.
- *removeFirst()*: we can produce the effect of removing the first data from the queue by incrementing *front* by 1 (and, of course reducing *size* by 1). This removal is shown in Figure 5-5 when our queue has 3 data: 7, 2, and 6. Thus, the next time the front of the queue is to be accessed,

we get the data stored behind the original front data. In Figure 5-5, the new front data is 2. The queue after *removeFirst()* finishes contains 2 and 6. This implementation prevents us from having to shift array contents after a data is removed.

- *insertLast(int data)*: this is done by adding the new data after the last data (and incrementing *size* by 1). The last data is at position $front+size-1$ so the new data goes into position $front+size$. See Figure 5-6, which actually carries out its *insertLast* method right after the operation in Figure 5-5. In Figure 5-6, the original data sequence in the queue is 2 and 6. The data sequence after method *insertLast* executes is 2, 6, and 5.

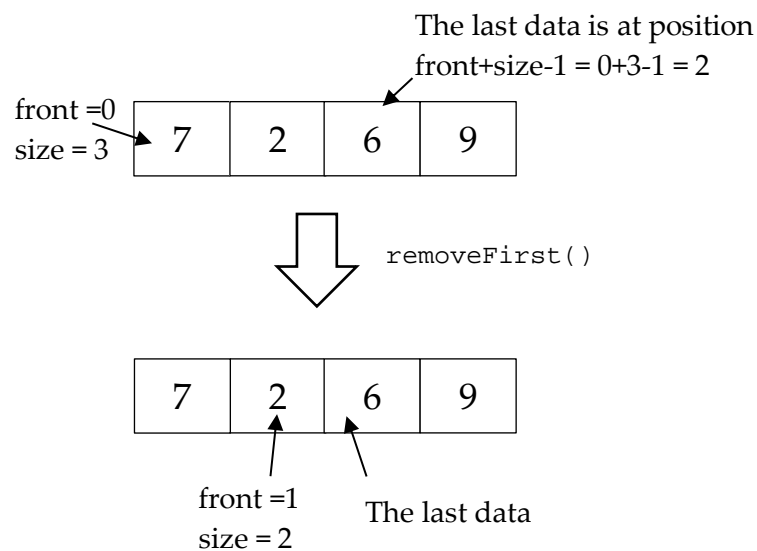


Figure 5-5: Dequeue for array implementation.

Method *removeFirst* and *insertLast*, although simple to implement, actually cause another problem. That is, when enqueue and/or dequeue are performed for some time, the last slot of our array will be occupied, while some slots at the front of the array are not used. This means we can no longer add another data because the position to add will exceed the rightmost array slot, even though there may be empty slots somewhere at the front of the array. In Figure 5-6, after method *insertLast* is executed, the last array slot is occupied by data 5, while the first slot is no longer used. If we want to enqueue a new data, it will have to be in position $front+size = 1+3 = 4$, which goes beyond the last array slot. So, it cannot be added, even though the first slot is available.

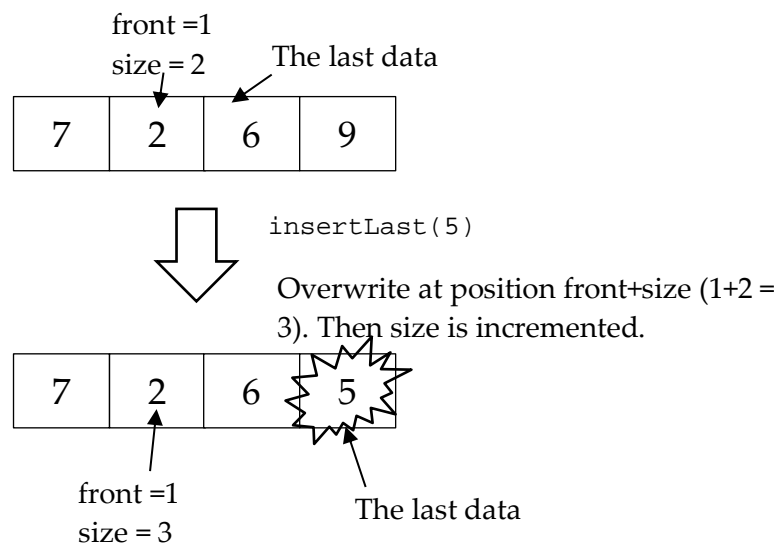


Figure 5-6: Enqueue for array implementation.

We can fix this problem by allowing array index to go from the back to the front of the array. Thus, method *removeFirst*, instead of having $front = front + 1$, will have $front = (front + 1) \% theArray.length$. An example of a call to method *removeFirst* is shown in Figure 5-7, where the original data sequence is 4, 2, 1 and the end data sequence is 2, 1.

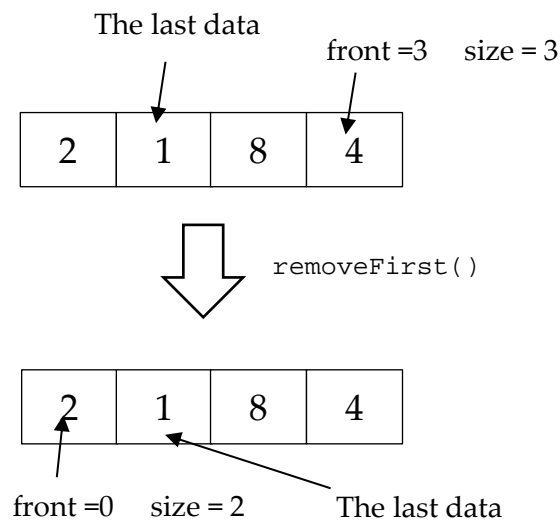


Figure 5-7: Incrementing *front* that goes back to the first array slot when dequeuing.

For method *insertLast*, instead of having a new data go into position $front + size$, we will have the new data go into position $(front + size) \% theArray.length$. In Figure 5-8, the data sequence is 2, 6, 5. The position to add the new data is $(1 + 3) \% 4 = 0$. Thus, the new data is put into the first slot, which is next to the position that stores 5 when we make

the array index go back to the start of the array. The data sequence then becomes 2, 6, 5, 1.

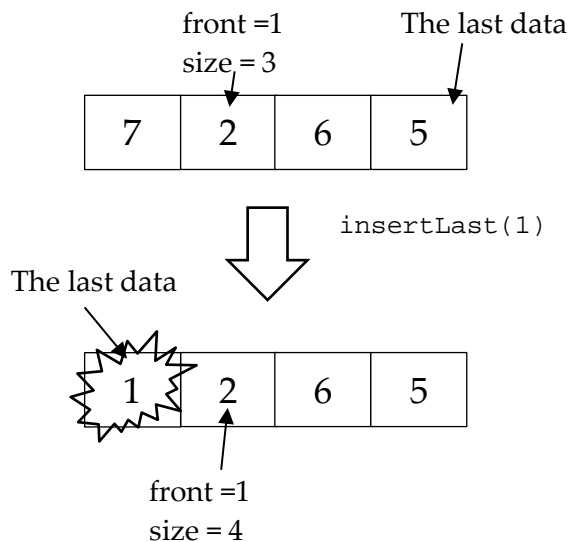


Figure 5-8: $front+size$ that goes back to the first array slot when enqueueing.

- *back()*: by making the index able to move from the last array slot back to the first array slot, the last position of data is at position $(front+size-1)\%theArray.length$. Method *back* simply returns data stored in that position.

The code for the class, fields, constructors, and methods that check for the value of *size* is shown in Figure 5-9, in class *QueueArray*. Although the constructors create an array (line 12), the array slots do not store any data for

our queue. That is why the value of *front* is set to -1 instead of 0 (line 14). If *front* is set to 0, it will mean there is one data in the queue, which is not true at the stage of queue creation.

```
1: public class QueueArray implements MyQueue {
2:     private int[] theArray;
3:     private int size; //number of currently stored data.
4:     private int front; //index of the first data.
5:     static final int DEFAULT_CAPACITY = 5;
6:
7:     public QueueArray() {
8:         this(DEFAULT_CAPACITY);
9:     }
10:
11:    public QueueArray(int capacity) {
12:        theArray = new int[capacity];
13:        size = 0;
14:        front = -1;
15:    }
16:    public boolean isEmpty() {
17:        return size == 0;
18:    }
19:    public boolean isFull() {
20:        return size == theArray.length;
21:    }
22:    public void makeEmpty() {
23:        size = 0;
24:        front = -1;
25:    }
26:    public int size() {
27:        return size;
28:    }
29:    // This class continues in Figure 5-10.
```

Figure 5-9: fields, constructors, and methods that check for *size* in the array implementation of queue.

All methods in Figure 5-9 do not have any loop in their codes, so each method has $\Theta(1)$ as its asymptotic runtime.

We have *makeEmpty()* (line 22-25) as an additional utility method. It basically resets *size* and *front* just like when we run a constructor, but it does use existing array.

The code for *front()* is shown in Figure 5-10. It can be seen that we can just return data in position *front*. But if the queue is empty, we will not be able to return any data. That is why an exception is thrown (the class *EmptyQueueException* just extends from class *Exception*). The asymptotic runtime of *front()* is $\theta(1)$ since there is no looping.

```
1: public int front() throws EmptyQueueException {
2:     if (isEmpty())
3:         throw new EmptyQueueException();
4:     return theArray[front];
5: }
6: //This class continues in Figure 5-11.
```

Figure 5-10: Code for *front()* in array implementation of queue.

The code for *back()* is shown in Figure 5-11. It returns data at position $(front+size-1)\%theArray.length$, as discussed earlier. It also needs to check if the queue is empty and throws an exception if so, since it will be impossible to return any value. The runtime of this method is $\theta(1)$ since there is no looping.

```
1: public int back() throws EmptyQueueException {
2:     if (isEmpty())
3:         throw new EmptyQueueException()
4:     return theArray[(front + size - 1) %
5:         theArray.length];
6: }
7: //This class continues in Figure 5-12.
```

Figure 5-11: Code for *back()* in array implementation of queue.

The code for *removeFirst()* is shown in Figure 5-12. Its illustrated operations are shown in Figure 5-5 and Figure 5-7. The method also needs to throw an exception if the queue is empty. It also needs to store the data at the front of the queue (line 5) in order to be able to return that data after *front* is incremented. Its execution does not involve any loop, therefore its asymptotic runtime is $\Theta(1)$.

```
1: public int removeFirst() throws EmptyQueueException{
2:     if (isEmpty())
3:         throw new EmptyQueueException();
4:     size--;
5:     int frontItem = thearray[front];
6:     front = (front + 1) % thearray.length;
7:     return frontItem;
8: }
9: //This class continues in Figure 5-13.
```

Figure 5-12: Code for *removeFirst()* in array implementation of queue.

The code for method *insertLast* is shown in Figure 5-13. Its illustrated operations are shown in Figure 5-6 and Figure 5-8. It simply overwrites data at position $(front+size)\%theArray.length$. But if the array is full, a new array of twice the original array size is created to replace the original (line 10-17). All data from the original array must be copied into a new array. Therefore, the copying process takes $\Theta(n)$, making method *insertLast* run in $O(n)$ since it either runs in constant time or in $\Theta(n)$.

```
1: public void insertLast(int data) throws
2:   EmptyQueueException {
3:     if (isFull())
4:       doubleCapacity();
5:     theArray[(front + size)%theArray.length] = data;
6:     size++;
7:   }
8:
9:   // resize array to twice its original size.
10:  public void doubleCapacity() {
11:    int[] temp = new int[theArray.length * 2];
12:    for (int i = 0; i < size; i++) {
13:      temp[i] = theArray[(front+i)%theArray.length];
14:    }
15:    theArray = temp;
16:    front = 0;
17:  }
18:  // end of class QueueArray.
```

Figure 5-13: Code for method *insertLast* in array implementation of queue.

Implementing a Queue with Linked List

In this section, a circular doubly-linked list is used to store data as our queue. The first data of the list is regarded as the first data in the queue, while the very last data in the list is considered to be the last data in the queue (see Figure 5-14). Therefore, our queue implementation using a linked list is just a linked list, but with restrictions forbidding any removal of data except the first, and any addition of new data except after the last data.

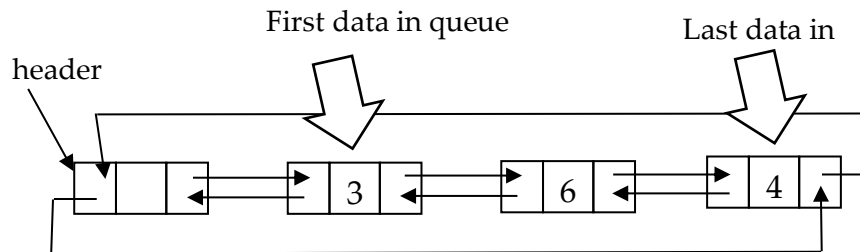


Figure 5-14: Using a circular doubly-linked list to represent a queue.

Figure 5-15 shows code for field, constructors, *isEmpty()*, *isFull()*, *size()* for this linked list implementation of a queue (the class name is *QueueLinkedList*). We only have one field, *theList*, which is a circular doubly-linked list from chapter 3. This list will be used as our queue. There are two constructors. A default constructor (line 4-6) just creates an empty linked list. The other constructor is a copy constructor, which takes a linked list and makes that list our data storage. Of course, we could do the same as in class *StackLinkedList* in chapter 3 and copy all data to a separate list. In this chapter, however, we opt for code simplicity. Method *isEmpty*, *isFull*, and *size* simply called their corresponding linked list methods in class *CDLinkedList*.

Figure 5-16 shows code for method *front* of our linked list implementation of queue. The method throws exception if the queue is empty (we check the list if it is empty), since there is no data to return. Otherwise, it returns data stored in the node next to *header*, using method *findKth* of

CDLinkedList to find data. The asymptotic runtime of this method is the asymptotic runtime of *findKth(0)*, which is $\Theta(1)$, since only the first iteration of the loop in *findKth(0)* is executed.

```
1: public class QueueLinkedList implements MyQueue {
2:     CDLinkedList theList;
3:
4:     public QueueLinkedList() {
5:         this(new CDLinkedList());
6:     }
7:
8:     public QueueLinkedList(CDLinkedList theList) {
9:         this.theList = theList;
10:    }
11:
12:    public boolean isEmpty() {
13:        return theList.isEmpty();
14:    }
15:
16:    public boolean isFull() {
17:        return theList.isFull();
18:    }
19:
20:    public int size() {
21:        return theList.size();
22:    }
23:    //continued in Figure 5-16.
```

Figure 5-15: Code for field, constructors, *isEmpty()*, *isFull()*, *size()* of linked list implementation of queue.

```
1: public int front() throws Exception {
2:     if (isEmpty())
3:         throw new EmptyQueueException();
4:     return theList.findKth(0);
5: }
6: // continued in Figure 5-17.
```

Figure 5-16: Code for *front()* of linked list implementation of queue.

The code for method *back* is shown in Figure 5-17. The method returns the last data in the queue, which is also the last data in the linked list that we use. The method throws an exception if the queue has no data.

```

1: public int back() throws EmptyQueueException {
2:     if (isEmpty())
3:         throw new EmptyQueueException();
4:     return theList.header.previousNode.data;
5: }
6: //continued in Figure 5-19.

```

Figure 5-17: Code for *back()* of linked list implementation of queue.

Figure 5-18 illustrates pointers from header to the last data (its corresponding code is at line 4 of Figure 5-17). There is no loop execution so the asymptotic runtime is $\theta(1)$.

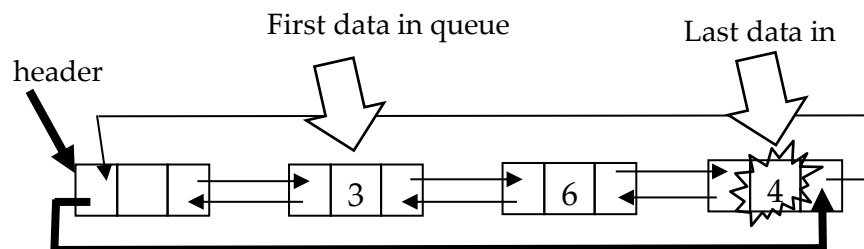


Figure 5-18: Identifying the last data in linked list implementation of queue.

Figure 5-19 shows our code for the linked list implementation of method *removeFirst*. Again, the method throws an exception if there is no data in our queue. If there is the first data, it is removed by calling method *removeAt* of our linked list implementation to

remove the node after *header*. Figure 5-20 shows what happens just before and after *removeAt(itr)*, from line 7 of Figure 5-19, is executed. The runtime of *removeFirst()* directly depends on the runtime of method *removeAt*, which is $\Theta(1)$.

```
1: public int removeFirst() throws Exception {
2:     if (isEmpty())
3:         throw new EmptyQueueException();
4:     DListIterator itr;
5:     itr = new DListIterator(theList.header);
6:     int data = itr.next();
7:     theList.removeAt(itr);
8:     return data;
9: }
10: // continued in Figure 5-21.
```

Figure 5-19: Code for *removeFirst()* of linked list implementation of queue.

Figure 5-21 shows code for method *insertLast* in our linked list implementation. Again, this method mainly calls a method of our linked list. For this particular operation, it calls *insert* method of a linked list, to insert a new data next to the last node. Figure 5-22 shows what happens just before and after *theList.insert(7, itr)*, at line 5 of the code in Figure 5-21, is executed. The asymptotic runtime of method *insertLast* directly depends on the runtime of *theList.insert(data, itr)*, which is $\Theta(1)$.

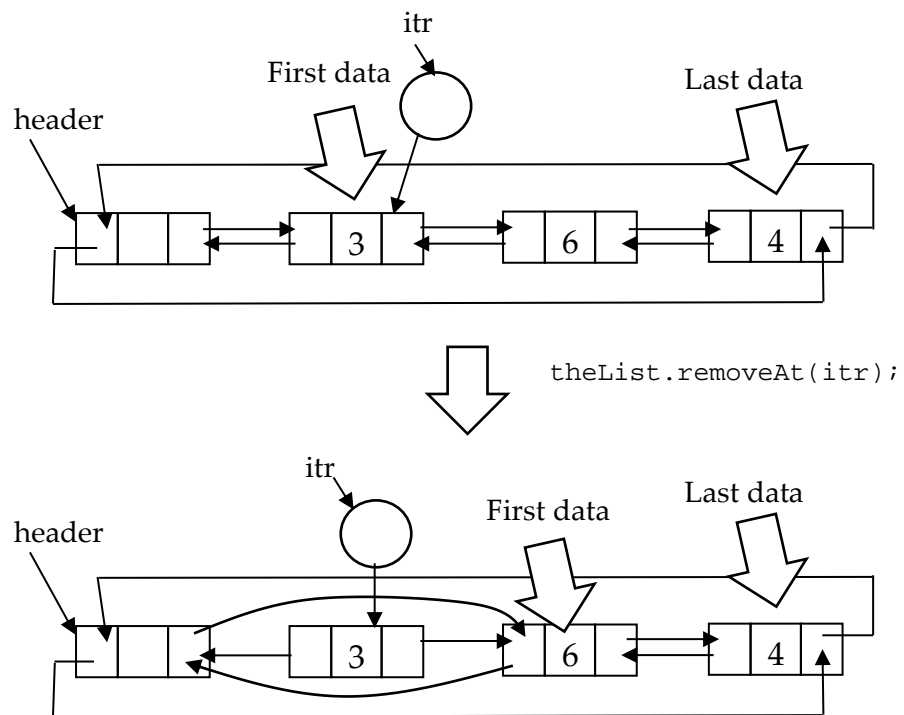


Figure 5-20: Removing the first data in linked list implementation of queue.

```

1: public void insertLast(int data) throws Exception {
2:     DListIterator itr;
3:     itr = new DListIterator(theList.header);
4:     itr.previous();
5:     theList.insert(data, itr);
6: }
7: } // end of class QueueLinkedList

```

Figure 5-21: Code for *insertLast()* of linked list implementation of queue.

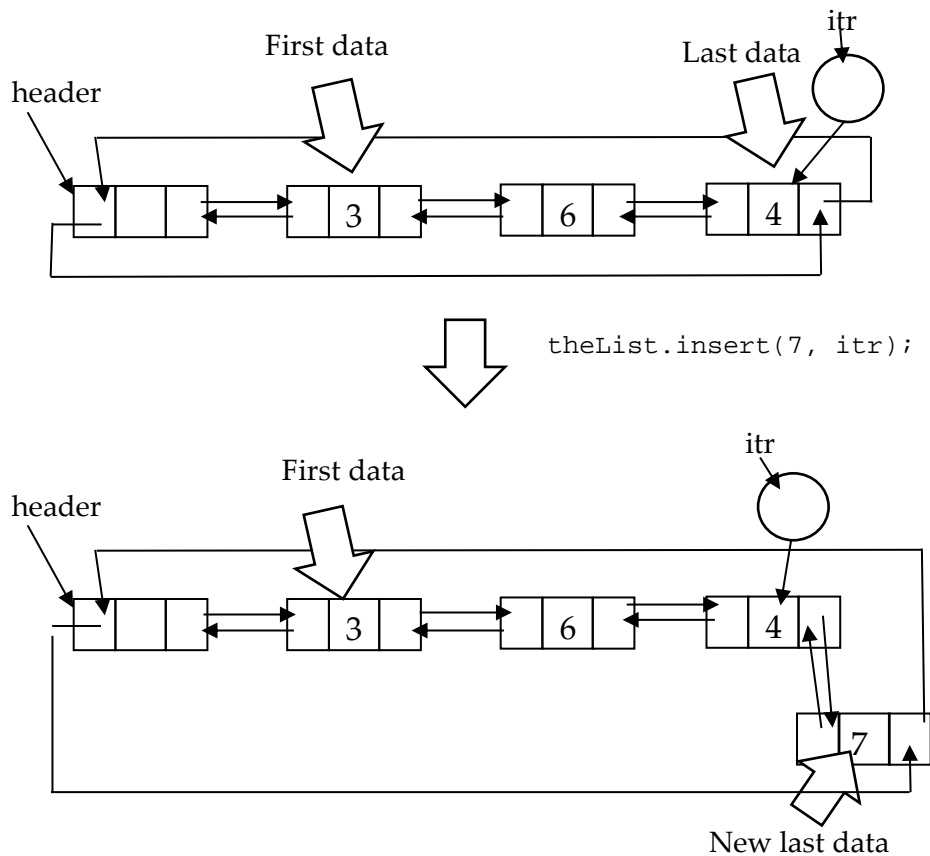


Figure 5-22: Adding a new data to linked list implementation of queue.

Double-Ended Queue

In a few programming languages, there is a queue-like data structure that is more flexible than the queue we saw in the previous section. This double-ended queue allows the following additional operations:

- *removeLast()*: remove the last data from a sequence of data stored in our queue. It also returns the removed data. Its concept is shown in Figure 5-23.
- *insertFirst(data)*: add a new data into the queue. This data becomes the first data in the sequence. Its concept is shown in Figure 5-24.

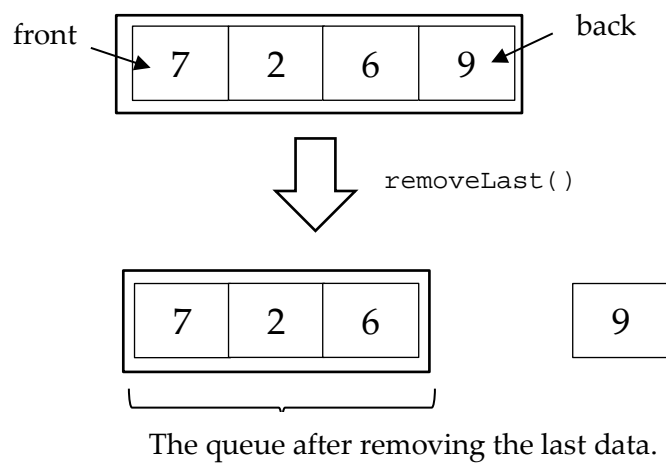


Figure 5-23: Illustrated concept of *removeLast()*.

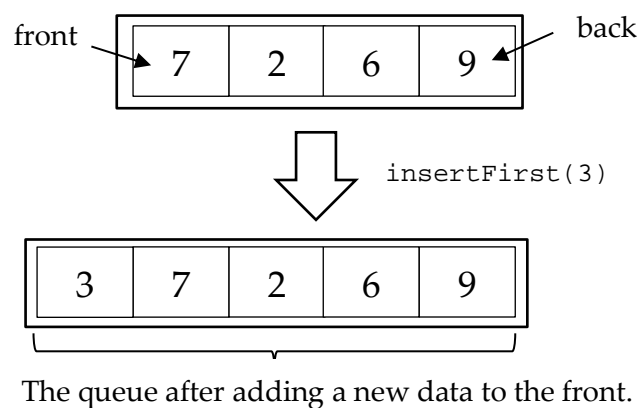


Figure 5-24: Illustrated concept of *insertFirst(data)*.

These operations can be shown as a Java interface in Figure 5-25. The interface presented here inherits from the interface in Figure 5-4. Basically, it is our existing queue, with two additional methods.

```
1: public interface DeQ extends MyQueue {
2:
3:     // remove the last data (return its value too).
4:     public int removeLast() throws Exception;
5:
6:     // insert new data as the first data.
7:     public void insertFirst(int data) throws
8:         Exception;
9: }
```

Figure 5-25: Java interface for double-ended queue.

Implementing a Double-Ended Queue with Array

Our implementation extends from *QueueArray* so that we only need to implement the two new functions (see Figure 5-26).

```
1: public class DeQArray extends QueueArray implements DeQ{
2:     public int removeLast() throws Exception {
3:         int data = back();
4:         size--; //change all fields to protected!
5:         return data;
6:     }
7:
8:     public void insertFirst(int data) throws Exception {
9:         if (isFull())
10:            doubleCapacity();
11:         front = front-1;
12:         if(front < 0)
13:             front = theArray.length-1;
14:         theArray[front] = data;
15:         size++;
16:     }
17: } //end of class DeQArray.
```

Figure 5-26: Double-ended queue implementation using array.

From Figure 5-26, *removeLast()* records data at the back of the queue, then simply reduces *size* by 1 before returning the recorded value. The effect of reducing *size* by 1, without changing *front*, is illustrated in Figure 5-27. The sequence of data stored in our queue is 4, 2, 1. The variable *size* determines the number of data considered to be in our queue, starting from data at position *front*. Reducing *size* means we consider smaller number of data from the position of *front* to be in our queue. Thus, we lose data at the back of the queue. The data sequence after the reduction of *size* is 4, 2.

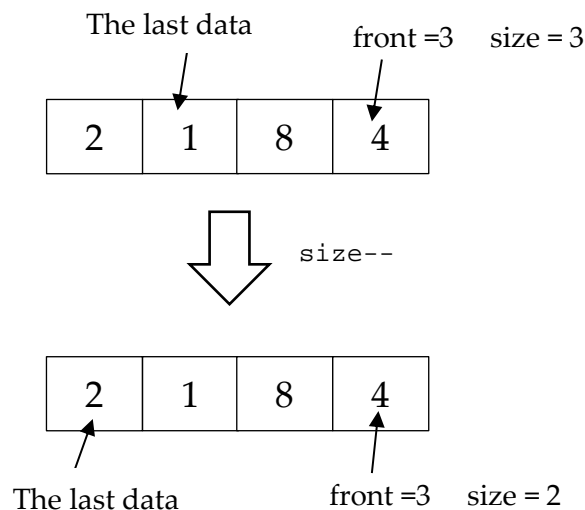


Figure 5-27: Reducing *size* without changing *front* in array implementation of double-ended queue.

The operation *removeLast()* does not require any loop. Therefore, its asymptotic runtime is $\Theta(1)$.

From Figure 5-26, method *insertFirst* reduces the value of *front* by 1 (it also changes *front* to identify the last array slot if its value becomes negative). Then it sets data at position of the new *front* to a given value, and increments *size*. This effectively adds a new data in front of the queue. Figure 5-28 illustrates line 11-14 from code in Figure 5-26, when *insertFirst(77)* is called on a queue with data 2, 1. The resulting data sequence is 77, 2, 1.

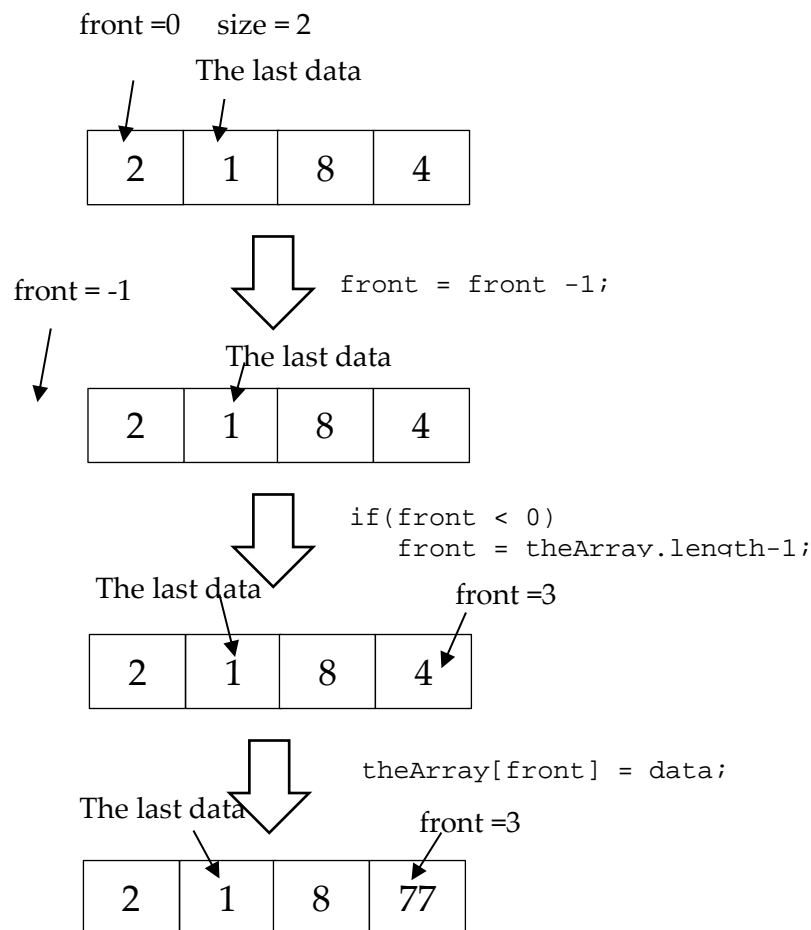


Figure 5-28: Operations inside *insertFirst(77)* for array implementation of double-ended queue.

Most of the time, method *insertFirst* does not require any loop execution. But occasionally, method *doubleCapacity* will have to be called to expand the array. This array resize causes the asymptotic runtime to be $O(n)$.

Double-Ended Queue implemented with Linked List

We can build on top of our existing linked list implementation. Thus, we only need to add method *removeLast* and *insertFirst*. Our implementation using a circular doubly-linked list is shown in Figure 5-29.

```
1: public class DeQLinkedList extends QueueLinkedList
2: implements DeQ {
3:
4:     public int removeLast() throws Exception {
5:         if (isEmpty())
6:             throw new EmptyQueueException();
7:         DListIterator itr = new DListIterator(theList.header);
8:         itr.previous();
9:         int data = itr.previous();
10:        theList.remove(itr);
11:        return data;
12:    }
13:
14:    public void insertFirst(int data) throws Exception {
15:        DListIterator itr = new DListIterator(theList.header);
16:        theList.insert(data, itr);
17:    }
18: } // end of class DeQLinkedList
```

Figure 5-29: Linked list implementation of double-ended queue.

From Figure 5-29, method *removeLast* throws an exception if the queue is empty. Otherwise, it creates a linked list iterator and moves the iterator to the left until

it identifies position before the last data. Then method *remove* of our circular doubly-linked list is called to remove the node after that position. The removed node is therefore the node that stores the last data.

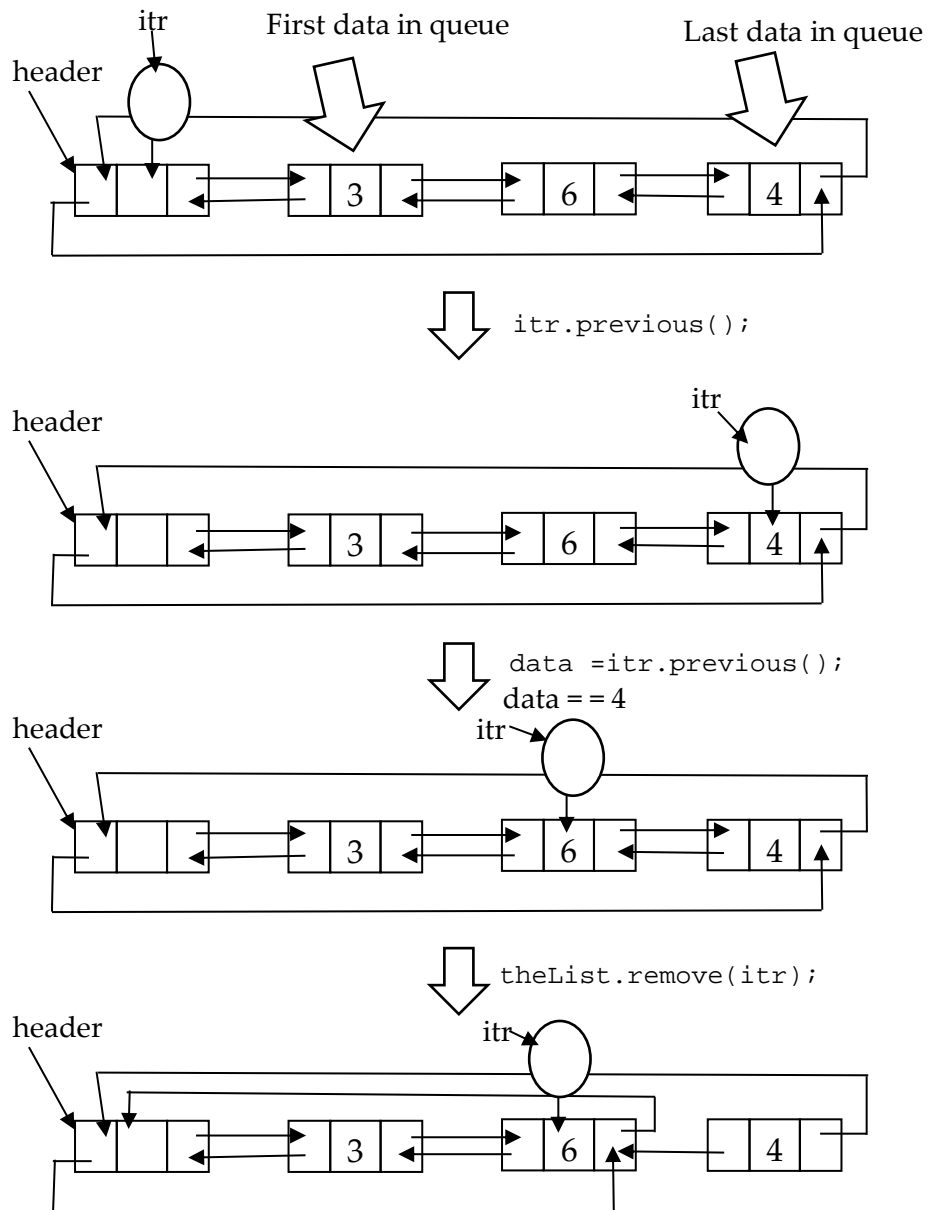


Figure 5-30: Operations inside `removeLast()` for linked list implementation of double-ended queue.

Once again, our implementation utilizes methods from linked list. The execution from line 8-10 in Figure 5-29 is illustrated in Figure 5-30. There is no loop so the runtime of *removeLast()* is $\theta(1)$.

From Figure 5-29, method *insertFirst* just adds a new node (containing new data) after *header*, by calling method *insert* of our linked list. The operation for *insertFirst(9)* is illustrated in Figure 5-31.

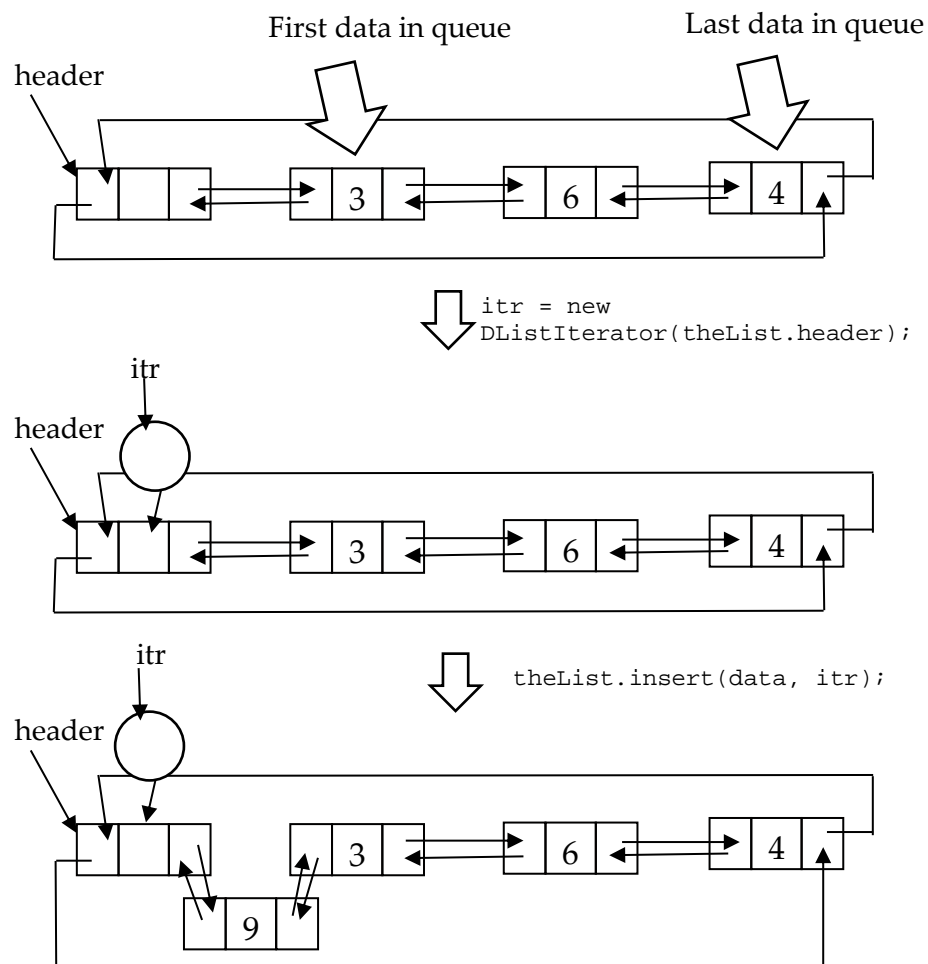


Figure 5-31: Operations of *insertFirst(9)* for linked list implementation of double-ended queue.

There is no loop, so the runtime is $\theta(1)$.

Application of Queue: Radix Sort

Apart from using queue(s) to simulate a FIFO data storage, there is a very interesting application to point out. That is, using queues to sort numbers with radix sort algorithm.

So, what is a radix sort? It is a method we can use to sort numbers. Let us have an array storing various numbers (let assume they are integers). What we do is as follows:

- 1) create 10 queues for storing numbers. The queues are labelled 0 to 9.
- 2) For each number, we use the value of its least significant digit as our "sorting identifier".
 - a) For each number in the array, look at its sorting identifier, then enqueue (method *insertLast*) that number into a queue with the same label as that sorting identifier.
 - b) For each queue, starting from the queue that has label '0', dequeue (method *removeFirst*) all numbers from that queue back to the array. Do it until all queues are empty.
 - c) Change the digit of the sorting identifier to the next significant digit, then repeat step a) to c) until there is no more possible sorting identifier.

Let us see an example. Let us sort 321, 521, 354, 324, 150 and 237 in an array. At the beginning, our sorting identifier is the value of the least significant digit. So, to do step a), we look at our array from the leftmost slot to the rightmost slot. Each number is put into a queue according to the sorting identifier. Thus, the number 321 and 521 go into queue 1. The number 354 and 324 go into queue 4. The number 150 goes into queue 0. The number 237 goes into queue 7 (see Figure 5-32).

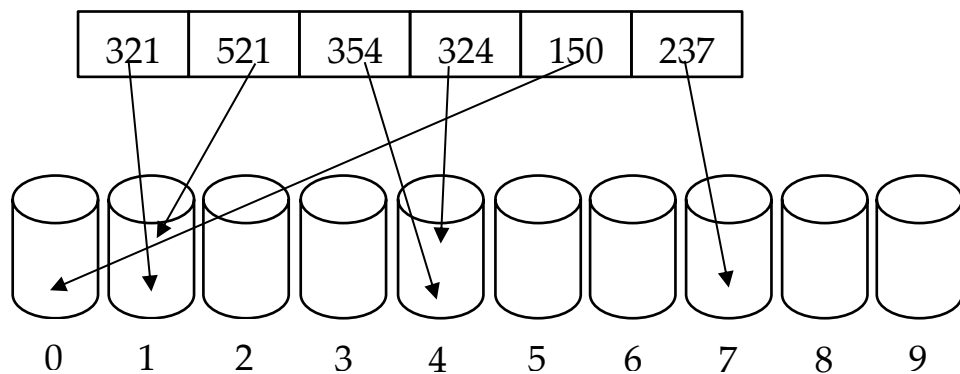


Figure 5-32: Step a), getting numbers into queues, when the least significant digit is the sorting identifier.

Now, to do step b), we start from queue 0. We dequeue all numbers from it back to the array. We do the same for queue 1, 2, 3, etc. (see Figure 5-33).

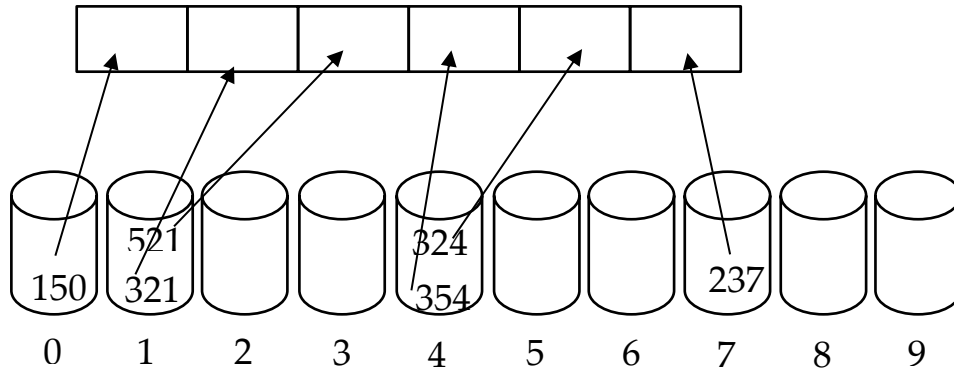


Figure 5-33: Step b), getting all numbers back to the array.

Now that all numbers are back in the array, we change the value of the sorting identifier to the value of the next significant digit. That is the number 150 will now have 5 as its sorting identifier. The number 321 will have 2 as its sorting identifier. Then we repeat step a) again with a new sorting identifier for each number (see Figure 5-34). Remember, we go through the array from left to right.

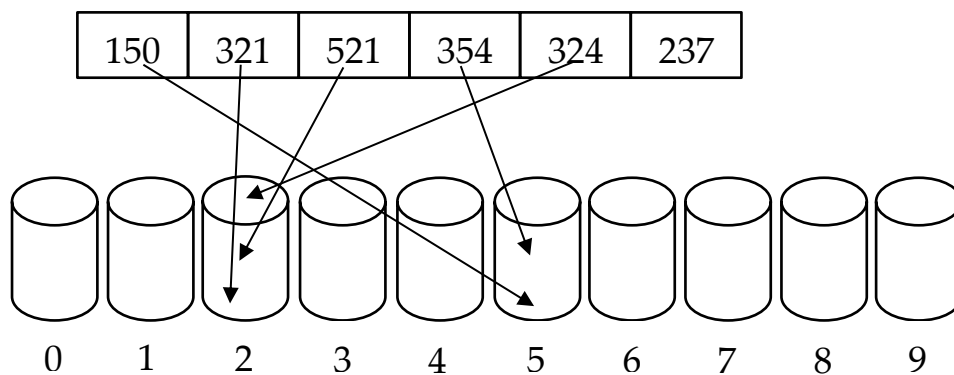


Figure 5-34: Step a), when using the second digit from the right as a sorting identifier.

And then step b) is carried out by doing *removeFirst()* on every queue, from left to right, and put the removed data back into our array (see Figure 5-35).

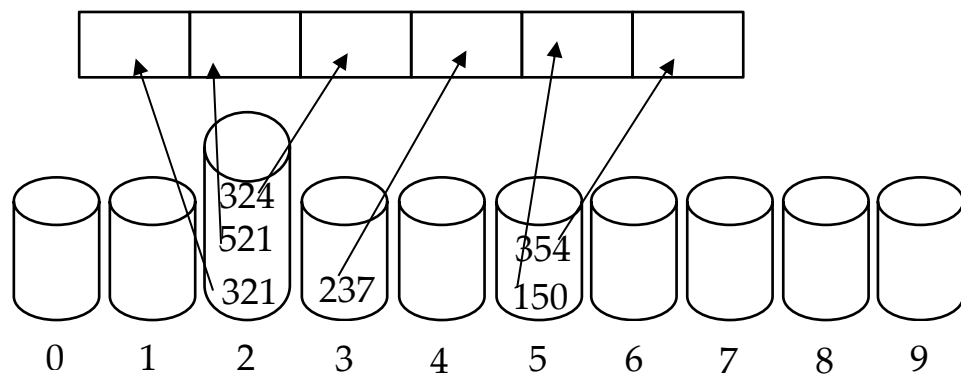


Figure 5-35: Step b), when using the second digit from the right as a sorting identifier.

Then, we change the sorting identifier for the last time (since all numbers in our example have the maximum digit number equal to 3). Now, the number 321 has 3 as its sorting identifier. The number 521 has 5 as its sorting identifier. And we carry out step a) (Figure 5-36) and step b) (Figure 5-37) as before.

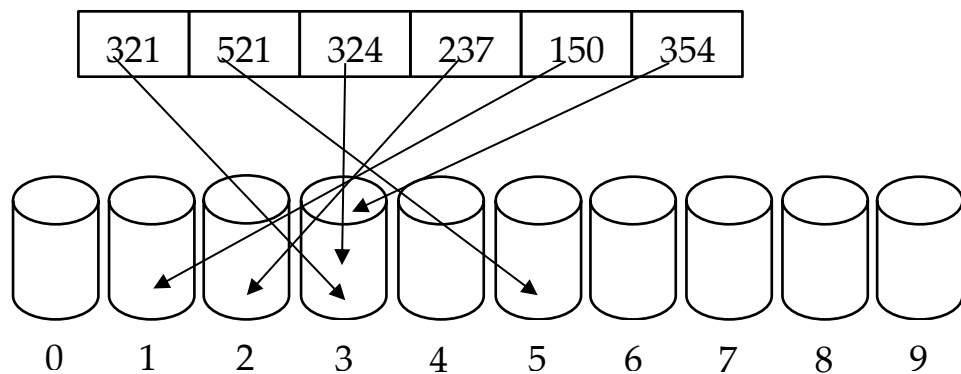


Figure 5-36: Step a), when using the third digit from the right as a sorting identifier.

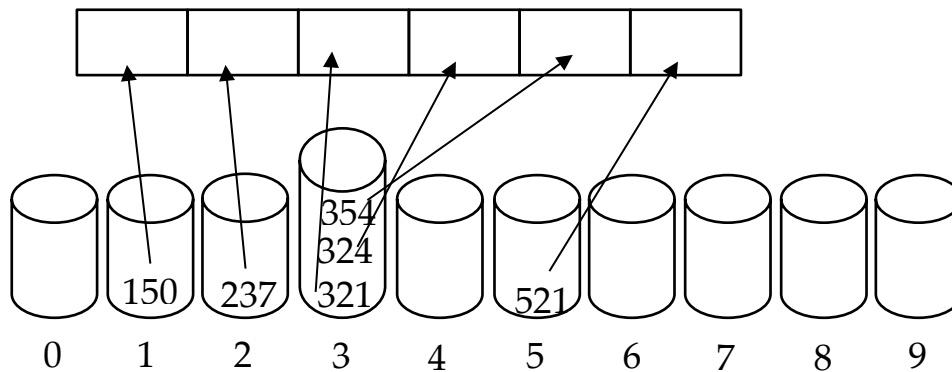


Figure 5-37: Step b), when using the third digit from the right as a sorting identifier.

After the actions carried out in Figure 5-37, all data in the array are sorted. The code for radix sort is shown in Figure 5-38 and Figure 5-39.

```

1: public class RadixSort {
2:   int[] theArray;
3:
4:   public RadixSort(int[] theArray) {
5:     this.theArray = theArray;
6:   }
7:
8:   // Return the kth digit of v.
9:   // The least significant digit is 0.
10:  public int getKthDigit(int v, int k) {
11:    for (int i = 0; i < k; i++)
12:      v /= 10;
13:    return v % 10;
14:  }
15:
16:  // Find the number of digits of a value v.
17:  public int numberOfDigit(int v) {
18:    int total = 1;
19:    while ((v / 10) > 0) {
20:      total++;
21:      v = v / 10;
22:    }
23:    return total;
24:  }
25:  //continued in Figure 5-39.

```

Figure 5-38: Radix sort implementation (part 1).

```
1: // Get the number of digits of
2: // the longest number in theArray.
3: public int maxDigit() {
4:     int maxDigit = 1;
5:     for (int i = 0; i < theArray.length; i++) {
6:         int n = numberOfDigit(theArray[i]);
7:         if (n > maxDigit)
8:             maxDigit = n;
9:     }
10:    return maxDigit;
11: }
12:
13: public void sort() throws Exception {
14:     int maxDigit = maxDigit();
15:     MyQueue[] allQueues = new MyQueue[10];
16:
17:     // initialize all 10 queues
18:     for (int i = 0; i < 10; i++)
19:         allQueues[i] = new QueueLinkedList();
20:
21:     // for each digit
22:     for (int k = 0; k < maxDigit; k++) {
23:         // for each data in array
24:         for (int i = 0; i < theArray.length; i++) {
25:             int value = theArray[i];
26:             MyQueue q = allQueues[getKthDigit(value, k)];
27:             q.insertLast(value);
28:         }
29:
30:         // index of array when we put data in from each
31:         // queue.
32:         int j = 0;
33:
34:         // for each queue
35:         for (int i = 0; i < 10; i++) {
36:             // empty each queue and output to theArray.
37:             while (!allQueues[i].isEmpty()) {
38:                 int data = allQueues[i].removeFirst();
39:                 theArray[j++] = data;
40:             }
41:         }
42:     } //end outer for
43: } //end method
44: } //end class
```

Figure 5-39: Radix sort implementation (part 2).

The main working of the code is in method *sort* in Figure 5-39. It uses the same sorting mechanism concept that has already been explained. For easier mapping of the concept, we summarize the code below:

- First, the maximum number of digits is calculated (line 14).
- Then all 10 queues are initialized (line 17-19).
- Then, starting from $k = 0$ (and ending when k reaches the maximum number of digit), we use k as the digit of our sorting identifier. For each k (line 22):
 - For each and every data in the array (line 24):
 - We identify the queue that the data must go to, by using its sorting identifier (line 26).
 - Then we use method *insertLast* to put the value into that queue (line 27).
 - For each queue, starting at queue 0 (line 35):
 - Until the queue is empty, we use method *removeFirst* to remove data from the queue and put it in our array (line 37-40). We put data in a different array slot each time.

Radix sort is very interesting because of its asymptotic runtime. Let us analyze the code of method *sort* together:

-
- The outer for loop (line 22) is only done MaxDigit number of times. This number is generally small, so it does not dominate the growth rate.
 - The first inner for loop (line 24):
 - Method *getKthDigit* (line 26) also works mainly on small value of k , so its runtime can be regarded as constant.
 - Method *insertLast* only takes $\theta(1)$ since we use a linked list here.
 - Therefore, the first inner for loop (line 24) only consumes time for the loop itself. Thus, it has growth rate = $\theta(n)$. Where n is the number of data.
 - The second inner for loop (line 35):
 - The most times the while loop gets run is $\theta(n)$, equal to the number of data.
 - *removeFirst* is only $\theta(1)$.
 - The second inner for loop itself runs only 10 times, so we can regard its growth rate to be $\theta(1)$.

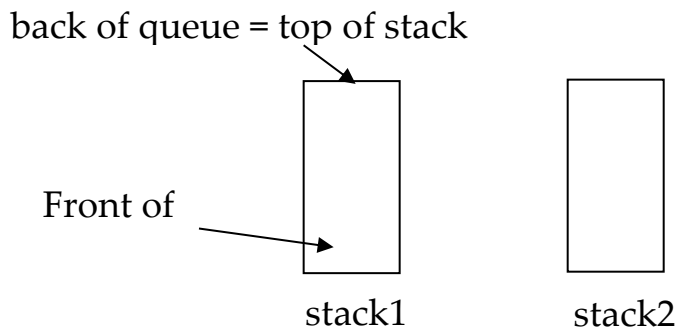
Thus, the growth rate of method *sort* is dominated by $\theta(n)$ of the first inner for loop and $\theta(n)$ of the while loop that follows. Therefore, the growth rate of the whole sort operation is $\theta(n) + \theta(n) = \theta(n)$.

Normally, if you are to write method *sort* that works on any number of data, you will need 2 for loops, each with its asymptotic runtime of $\theta(n)$. So, the runtime is

generally $\Theta(n^2)$. Therefore, the fact that radix sort has its runtime equal to $\Theta(n)$ means it is much faster than conventional methods.

Exercises

1. If we have existing *stack1* and *stack2*, we want to use *stack1* to represent a queue and use *stack2* for any bookkeeping (see a picture below). The data type for stack in this question is *MyStack* from chapter 4.



Explain how we can use these 2 stacks to implement method *insertLast* and *removeFirst*. Remember, stacks can only be manipulated by popping and pushing.

Write the code for your *insertLast* and *RemoveFirst*.

2. Assume we already have our own class *Q*, a queue that stores integers, and implement *MyQueue* interface of this chapter (assume all methods from *MyQueue* are implemented).

-
- a. Explain how we can manipulate the queue's content so that only even number remains (drawing can help). Write method *public void removeOdd()* of class *Q* that performs this task.
 - b. Write method *public void removeOddIndex()* of class *Q*. This method removes all data that are in odd positions from our queue. For example, if data in the queue are "a, b, c, d, e, f", this method will change the queue to "a, c, e" (the leftmost data is at position 0). **You must use only methods available for *MyQueue* and you are not allowed to create non-primitive variables.**
 - c. Write code for *public void moveBackToFront()* of class *Q*. This method moves the last integer stored in the queue to the front of the queue. Other stored integers remain unchanged. **You are not allowed to create any new array, linked list, stack, or queue.**
 - d. Explain how you can move integer x from anywhere in the queue to the front of the queue, without changing the ordering of other integers in the queue. **You are only allowed to create primitive variables and another queue. You are not allowed to create arrays, linked lists, stacks, trees and other kinds of data structures.** If x is not in the queue, do nothing. Following your description, write *public void moveToFront(int x)* for class *Q*. Give the

asymptotic runtime of your solution (Assume that each given method of Q in this question takes constant time to run).

- e. Write method *public void reverseQueue()* for class Q . This method reverses the ordering of elements in the queue. **You are only allowed to create primitive variables and a stack (any stack implementation from chapter 4 is fine). You are not allowed to create arrays, linked lists, trees and other kinds of data structures.** Assume that each given method of this question takes constant time to run, give the estimated runtime of your implementation.
- f. Write method *public Q merge(Queue q1, Queue q2)* for class Q . This method receives two queues, $q1$ and $q2$. Each of the queues has elements in sorted order (from small number to large number). The method creates a new queue that has all elements from $q1$ and $q2$. The new queue still has its elements in sorted order. You are allowed to destroy or change $q1$ and $q2$.
- g. In order to sort elements in a queue, you can do the following
 - i. Divide the elements in the queue in half (each half has equal, or almost equal numbers of elements).
 - ii. Put elements in the first half of the queue into a new queue.

- iii. Put elements in the last half of the queue into another new queue.
- iv. Sort the new queues.
- v. Combine the elements from the new queues to form the answer queue.

Write method `public Queue sortQueue()` for class `Q`. This method performs the above algorithm.

- h. Explain how you can put a new data x at position i in a queue (the leftmost data in your data sequence is at position 0). The data that used to be at position i (and all data stored after it) must move one position to the right. **You are not allowed to create any new data structure.** From your explanation, write method `public void jumpQueue(int x, int i)` of class `Q`.

3. Assume we have Class `Queue`, a double-ended queue that stores integers. This class **already implements** all methods defined in the following interfaces:

```
public interface MyQueue {
    //Return the first data.
    //Throw Exception if the queue is empty.
    public int front() throws Exception;

    //Return the last data.
    //Throw Exception if the queue is empty.
    public int back() throws Exception;

    //Remove the first data (return its value too).
    //Throw Exception if the queue is empty.
    public int removeFirst() throws Exception;
```

```
//Insert new data after the last data.
//Throw exception if the insert fails for some reason.
public void insertLast(int data) throws Exception;

//Check if the queue is empty.
public boolean isEmpty();

//Check if the queue has no more space to store new data.
public boolean isFull();

//Return the number of data currently stored in the queue.
public int size();
}
public interface DeQ extends MyQueue {
    // Remove the last data (return its value too).
    // Throw Exception if the queue is empty.
    public int removeLast() throws Exception;

    // Insert new data as the first data.
    // Throw Exception if the insert is not successful
    // for some unknown reason.
    public void insertFirst(int data) throws Exception;
}
```

We are using a double-ended queue in our own class TestQueue, which is:

```
Class TestQueue{
    Queue q;

    public int findValue(int i){
        //return a value at ith position in q
        //(the position number starts from 0).
        // This method is assumed to be completed and working with
        // runtime = O(n).
        // It assumes that the value of i must be from 0 to
        // size()-1.
        // Do not code this method!
    }

    public void swap(int p1, int p2){
        // You have to write code for this method.
    }
}
```

You are to implement method *swap*, which exchanges values stored in position *p1* and *p2* of *q*, where *p1* and *p2*'s value must be from 0 to *size()-1*. For example, if the original data inside *q* is: {1,2,3,4,5} and *swap(1,3)* or *swap(3,1)* is called, then the final data in the queue will be {1,4,3,2,5}. Please note that:

- You do not know the internal workings of *q*, so you can only use methods provided by the interfaces.
- You are allowed to create primitive type variables.
- You are not allowed to create non-primitive type variables, or any data structure.

- a. Write code for method *swap*.
- b. Draw pictures and explain your code. Do the explanation and drawings at the side of your code so that each part of the code is clearly explained.
- c. Analyze asymptotic runtime for each part of your code and give the overall asymptotic runtime of method *swap*.

15. For our doubly-linked list implementation of queue, extend it so that each data has a priority value. When adding new data to the queue, the data with more priority value is stored so that it is removed before data with lower priority values. Rewrite methods if necessary.

Chapter 6 : Binary Tree

A binary tree is a sequence of nodes, similar to a linked list. However, a node of a tree can have at most 2 next nodes. Thus, a linked list can be regarded as a form of binary tree that each node only has one next node. An illustrated example of a tree is shown in Figure 6-1.

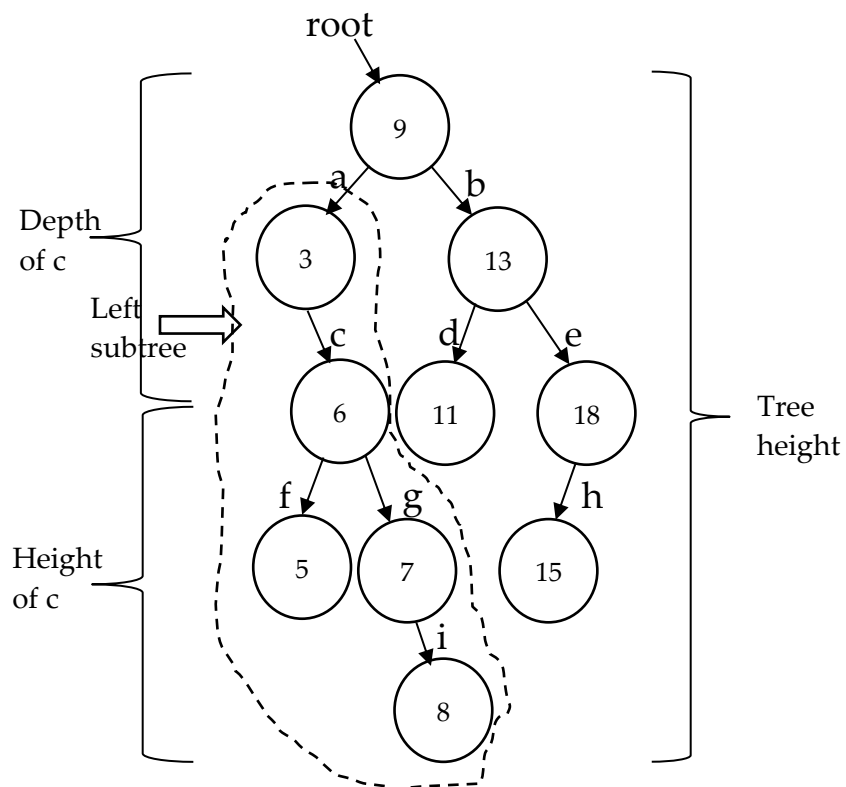


Figure 6-1: A Binary Tree.

In Figure 6-1, a tree is formed by connecting node a, b, c, ..., i together. Below are some terms we need to be familiar with regarding this data structure.

- A root is the very first node of a tree. From Figure 6-1, a root is the node that stores the number 9.
- For a tree, its root links to its left subtree and right subtree. From Figure 6-1, our tree has its left subtree being a tree that has node 'a' as its root. Similarly, its right subtree is a tree that has node 'b' as its root.
- A parent of node n is the node directly linked just above n. From Figure 6-1, node 'a' is a parent of node 'c'. Node 'c' is a parent of node 'f' and node 'g'.
- A child of node n is the node directly linked just below it. From Figure 6-1, node 'c' is a child of node 'a'. Node 'f' and node 'g' are children of node 'c'.
- An ancestor of node n is a node that can find only downward link(s) to n. From Figure 6-1, if we consider node 'g', we can see that root, node 'a', and node 'c' are ancestors of node 'g'. Node 'b' is not an ancestor of node 'g' because we cannot reach node 'g' from node 'b' with downward links alone.
- A leaf of a tree is the node that has no children. From Figure 6-1, node 'f', 'i', 'd' and 'h' are the leaves.
- The depth of node n is the largest number of links we can follow upwards from n (not including the root). From Figure 6-1, the depth of node 'c' is 2. The depth of node 'h' is 3.
- The height of node n is the largest number of links we can follow downwards from n (not including

itself). From Figure 6-1, the height of node 'c' is 2. The height of root is 4.

- The height of a tree is the height of its root. From Figure 6-1, the height of our tree is 4.
 - A tree that has only one node (that is, only a root) has its height equal to 0.
 - An empty tree, which is a tree with no node, has its height equal to -1.
 - The height of a tree can be calculated from the height of its tallest subtree +1. From Figure 6-1, our tallest subtree is the tree which has node 'a' as its root (and it has its height equal to 3). Therefore, our tree has height equal to $3+1 = 4$.
- Please note that there is only 1 path from a root to a node.

Nodes in a tree can be at different levels, as shown in Figure 6-2.

A **perfectly balanced** (or a perfect) **binary tree** looks like a complete triangle, as shown in Figure 6-3. A **full binary tree** (or strict binary tree) is a tree that each node either has 0 or 2 children.

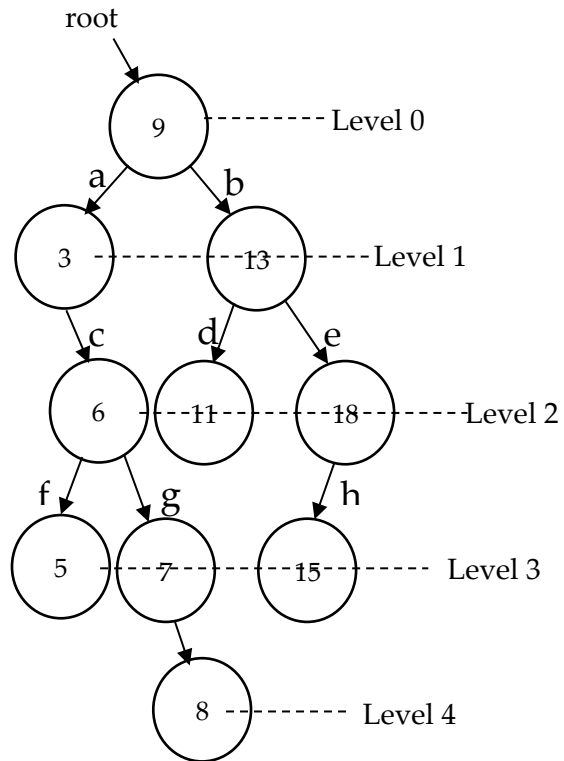


Figure 6-2: Node levels in a tree.

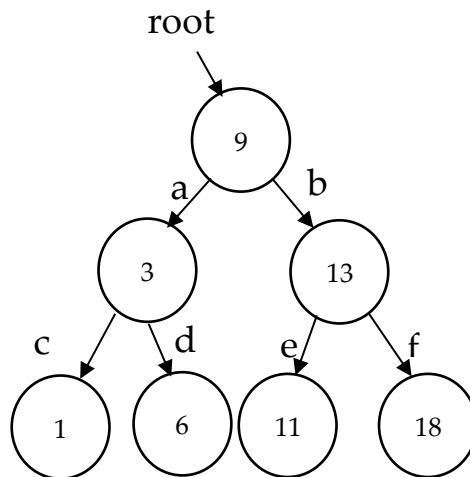


Figure 6-3: A perfectly balanced tree.

A **complete** binary tree is the tree filled up at least to the level before the highest level, from left to right. Once a node is missing, all other nodes in the same level (we look from left to right) and in the levels after that must not exist. Thus, a perfectly balanced tree is also a full and complete binary tree, but a complete (or full) binary tree does not have to be perfectly balanced.

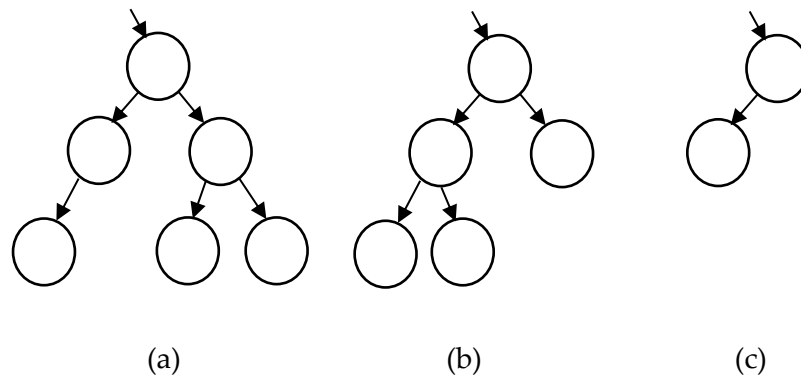


Figure 6-4: Examples of non-complete/complete binary trees.

The tree in Figure 6-4 (a) is not a complete binary tree because in the last level, nodes do not fill from left to right (one node is missing along the way). The tree in Figure 6-4 (b) and Figure 6-4 (c) are complete binary trees because when a node is missing, other nodes to the right and to other levels do not exist also.

Interesting properties of A Binary Tree

Let us define the followings:

- $leaves(t)$: the number of leaves in the tree t .
- $n(t)$: the number of nodes of tree t .
- $height(t)$: the height of t .
- $leftsubtree(t)$: the left subtree of t .
- $rightsubtree(t)$: the right subtree of t .
- $max(a,b)$: a maximum value from a and b .

For a non-empty binary tree, the following definitions are true:

Definition 6-1:

$$leaves(t) \leq \frac{n(t)+1}{2.0}$$

Definition 6-2:

$$\frac{n(t)+1}{2.0} \leq 2^{height(t)}$$

Definition 6-3:

$$\text{If } t \text{ is a full binary tree, then } leaves(t) = \frac{n(t)+1}{2.0}$$

Definition 6-4:

$$\text{If } leaves(t) = \frac{n(t)+1}{2.0}, \text{ then } t \text{ is a full binary tree.}$$

Definition 6-5:

If t is a perfect binary tree, then $\frac{n(t)+1}{2.0} = 2^{\text{height}(t)}$

Definition 6-6:

If $\frac{n(t)+1}{2.0} = 2^{\text{height}(t)}$, then t is a perfect binary tree.

These definitions can be proven using discrete mathematics. In our context, we are not interested in how they are proven, but on how they can be used. Here, we give a short proof for $\text{leaves}(t) \leq \frac{n(t)+1}{2.0}$:

We use mathematical induction. The base case is when our tree has only its root.

- Thus $\text{leaves}(t)$ is 1 because the root itself is the only leaf.
- The value of $\frac{n(t)+1}{2.0}$ is $\frac{1+1}{2.0}$, which is 1.
- Therefore $\text{leaves}(t) \leq \frac{n(t)+1}{2.0}$ is true for this base case.

Let the inductive case happens when our tree has height equal to h . Thus $\text{leaves}(t) \leq \frac{n(t)+1}{2.0}$ is true when the tree height is h .

Now, we must prove that when the tree height is $h+1$, $\text{leaves}(t) \leq \frac{n(t)+1}{2.0}$ still holds. So, for a tree of height $h+1$, the following is true:

$$\text{leaves}(t) = \text{leaves}(\text{leftsubtree}(t)) + \text{leaves}(\text{rightsubtree}(t))$$

Since t has height equal to $h+1$, its left subtree and right subtree must have their height less than $h+1$, meaning the property of the inductive case is true for them. Therefore, we can replace $\text{leaves}(\text{leftsubtree}(t))$ with $\frac{n(\text{leftsubtree}(t))+1}{2.0}$. The same can be done for the right subtree. Hence, we get:

$$\text{leaves}(t) \leq \frac{n(\text{leftsubtree}(t)) + 1}{2.0} + \frac{n(\text{rightsubtree}(t)) + 1}{2.0}$$

And since we know that:

$$n(t) = n(\text{leftsubtree}(t)) + n(\text{rightsubtree}(t)) + 1$$

We can substitute $n(t)$ into our previous equation and eventually get:

$$\text{leaves}(t) \leq \frac{n(t) + 1}{2.0}$$

This completes our proof.

These properties are very important for optimizing how data are stored. For example, let us look at definitions about height (Definition 6-2, Definition 6-5, and Definition 6-6). From these definitions, we can deduce that if data are well distributed such that the tree is a perfect (or almost perfect) binary tree, the height of the

tree will have the lowest value possible. This leads to the lowest possible search time when we want to find a data stored inside (we start our search from the root). This height is related to the number of nodes according to Definition 6-5 and Definition 6-6.

$$\frac{n(t) + 1}{2.0} = 2^{\text{height}(t)}$$

We can take log for the above equation and come up with:

$$\log_2 \frac{n(t) + 1}{2.0} = \log_2 2^{\text{height}(t)}$$

$$\log_2(n(t) + 1) - \log_2 2 = \text{height}(t)$$

$$\log_2(n(t) + 1) - 1 = \text{height}(t)$$

The number of stored data in a tree is equal to the number of nodes. Therefore, in a binary tree that has good distribution of data, if there is a way to choose which node (left or right) to go search for the data, the process of searching can take a number of steps equal to the height of the tree, which is directly proportional to the logarithm of the number of data. In other words, searching a binary tree can take as little time as $O(\log n)$, where n is the number of data in the tree.

With certain data storage policy, the above performance can be achieved. A tree with such policy is called a binary search tree, which is discussed in the next section.

Definitions regarding the number of leaves can also be useful. We will touch upon the use of these definitions when we discussed priority queues.

Binary Search Tree

A binary search tree is either an empty tree, or a binary tree with the following properties:

- For each node, every data in its left subtree has smaller (or equal) value than the data stored in that node.
- For each node, every data in its right subtree has larger (or equal) value than the data stored in that node.

Binary trees shown in Figure 6-1 to Figure 6-3 are binary search trees.

Searching for data is easy. Let us look at a binary search tree in Figure 6-5. Let us try to find number 4.

- When we look at the root, it stores the number 7, so 4 must be on the left side of the root. We therefore follow the link to node 'a'.

- Node 'a' stores 5. Due to the data arrangement of a binary search tree, our data is surely on the left side of 'a'. We therefore follow the link to node 'c'.
- Node 'c' stores 2. Our number, 4, must be to the right of node 'c'. So, we follow the link to node 'f' and eventually find 4 there.

As mentioned earlier, if data are distributed well, the searching time only depends on the height of the tree, which is $\log n$, where n is the number of data stored inside the tree.

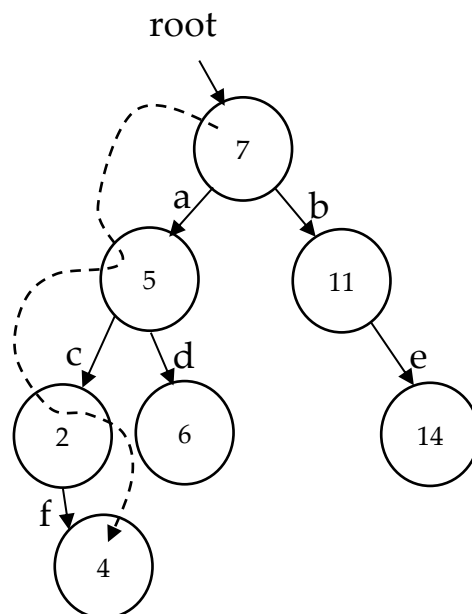


Figure 6-5: Searching for the number 4 in a Binary Search Tree.

Binary Search Tree Implementation

We define the followings:

- Node.
- Node marking and how to go through data.
- Binary Search Tree.

Let us start by defining the node that uses to store data. In our example code, data will be integer.

A node is a pointer to the space we store data. The following data are stored (Figure 6-6):

- A value (integer value in our example).
- A pointer to another node on the left.
- A pointer to another node on the right.
- A pointer to the node's parent node.

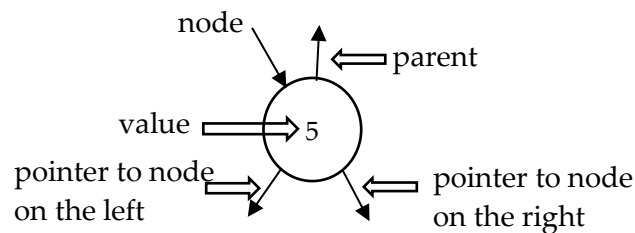


Figure 6-6: A node implementation concept of a binary tree.

Figure 6-7 shows how the tree from Figure 6-5 will look like under our implementation idea. If we focus at node 'a', we can see that:

- Its space stores the value 5.

- The pointer to the node on the left is 'c'.
- The pointer to the node on the right is 'd'.

The pointer to the parent node is *root* (we name the pointer according to its destination. Since that destination already has a name “*root*”, we use that name).

Parent of *root* is *null*. The pointer to the node on the left or on the right of a node can also be *null*.

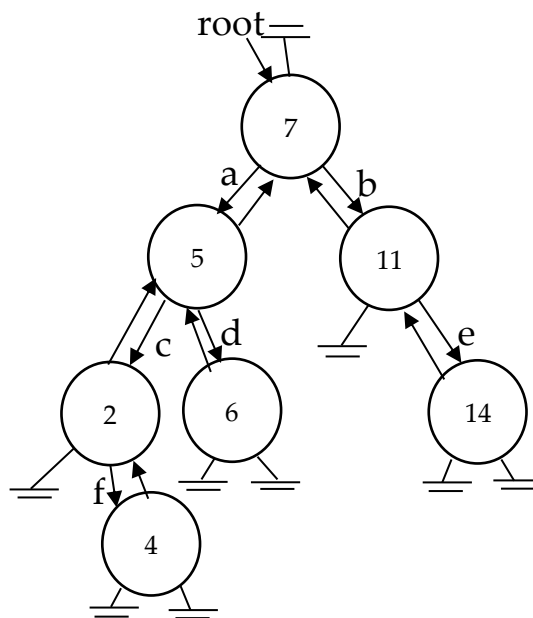


Figure 6-7: Tree from Figure 6-5, utilizing our implementation idea.

The code for our node implementation is shown in class *BSTNode* (Figure 6-8). A one parameter constructor

creates a node that simply stores data, but does not connect to left/right subtree and it does not have a parent (see Figure 6-9 for illustration).

```
1: public class BSTNode {
2:     int data; // value stored in the node.
3:     BSTNode left; //pointer to lower left BSTNode.
4:     BSTNode right; //pointer to lower right BSTNode.
5:     BSTNode parent; //pointer to the BSTNode above.
6:
7:     public BSTNode(int data){
8:         this(data,null,null,null);
9:     }
10:
11:     public BSTNode(int data, BSTNode left, BSTNode right,
12:         BSTNode parent) {
13:         this.data = data;
14:         this.left = left;
15:         this.right = right;
16:         this.parent = parent;
17:     }
18: }
```

Figure 6-8: Code for binary search tree node.

```
BSTNode b = new BSTNode(9);
```

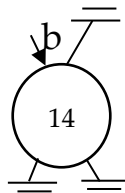


Figure 6-9: Creating a node by using one parameter constructor.

For node marking, we need to be able to determine a sequence for visiting all nodes in the tree, from the node that stores the smallest value to the node that stores the largest value. If we use the tree from Figure 6-7, the

sequence is shown in Figure 6-10. Round square labels indicate the visiting order for method *next* of our iterator.

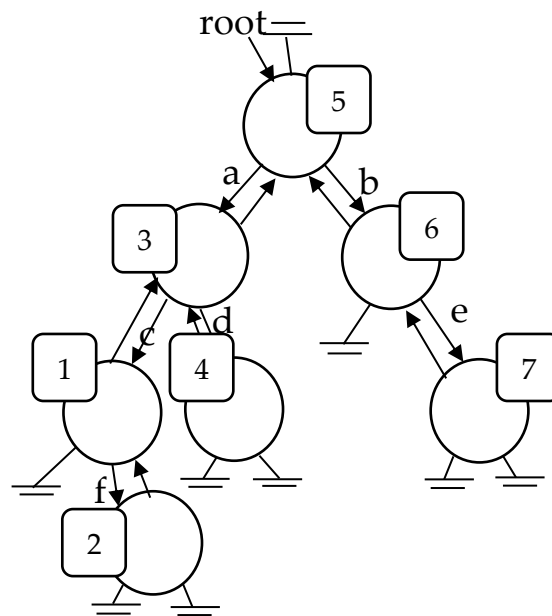


Figure 6-10: Node visiting sequence.

This is not quite straightforward. You can see that going from label 2 to 3 (or 4 to 5) requires more than one link traversal. With a higher tree, we will require even more link traversal. For the tree created from Figure 6-1, going from the node that stores 8 and the node that stores 9 requires 4 link traversals. Therefore, our implementation of a binary search tree iterator must support this.

Figure 6-11 shows how we implement a field and a constructor for our tree iterator (class *TreeIterator*). Our

iterator only marks a node and the constructor is straightforward.

```
1: public class TreeIterator implements Iterator {
2:     BSTNode currentNode;
3:
4:     public TreeIterator(BSTNode currentNode) {
5:         this.currentNode = currentNode;
6:     }
7: // continued in Figure 6-12.
```

Figure 6-11: Code for tree iterator field and constructor.

Figure 6-12 shows the implementation of method *hasNext*, which determines whether there is a next data (data with the next larger value from the current data) to look at.

```
1: public boolean hasNext() {
2:     BSTNode temp = currentNode;
3:     if (temp.right != null)
4:         return true;
5:     BSTNode p = temp.parent;
6:     while (p != null && p.right == temp) {
7:         temp = p;
8:         p = temp.parent;
9:     }
10:    if (p == null)
11:        return false;
12:    else
13:        return true;
14: }
15: // continued in Figure 6-16.
```

Figure 6-12: Method *hasNext* of class *TreeIterator*.

Method *hasNext* works as follows:

- If the currently marked node has another node to its right, then there definitely is a larger value to

visit via its *right* field. So, the method returns true right away (line 3-4 of Figure 6-12).

- As an example, consider the tree in Figure 6-10. If the node with label 3 is the currently marked node. It has the node with label 4 as its right field, so there is definitely a larger value to go visit. Thus, our method returns true for this case.
- If the currently marked node does not connect to any node by its right *field*, a value larger than the one stored in that node can be at one of its ancestor nodes. So, we need to follow the link up the tree until we find that ancestor node, or until there is no more link to follow (line 6-9 of Figure 6-12).
 - An example is shown in Figure 6-13. If *temp* is the node that contains value 5, its parent will be the node that contains value 6. It is the larger value we are looking for, so we know there is a next node. Indeed, our code (line 6 of Figure 6-12) evaluates that node *p.right* is not *temp* so the code does not even enter the loop. The value of *p* is not *null* so the code returns true.
 - Another example is shown in Figure 6-14. In this case, our current node *temp* stores value 8 (see the left-hand side of the figure). A node that stores a larger value is an ancestor quite further away. But our code will find it. Its loop will move *p* and *temp* up the tree until *p*

marks that node (see the right-hand side of the figure).

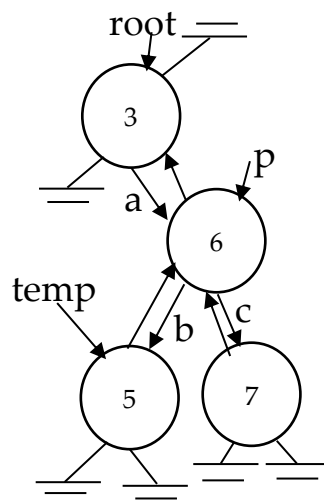


Figure 6-13: Immediate parent contains a larger value.

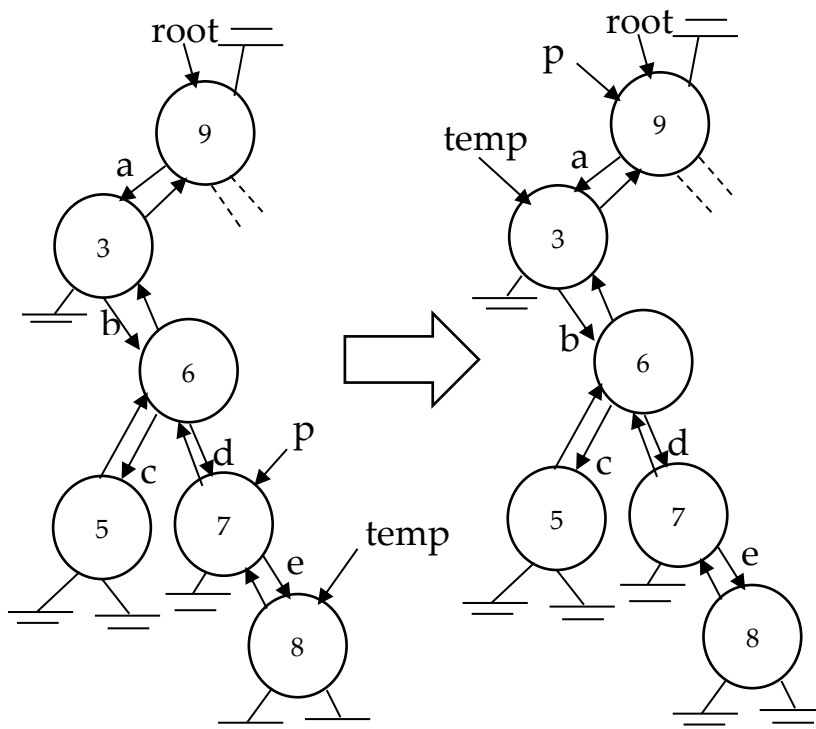


Figure 6-14: Movement of p and $temp$ when the larger value is in some ancestor node.

- It is also possible that a node that contains a larger value does not exist. In Figure 6-15 (left hand side of the figure), our current node stores 8. It is obvious that there exists no node that stores a larger value. So, there should not be a next node. Our code will loop to move p and $temp$ up the tree until p is *null* (see the right-hand side of the figure). The code will then return false.

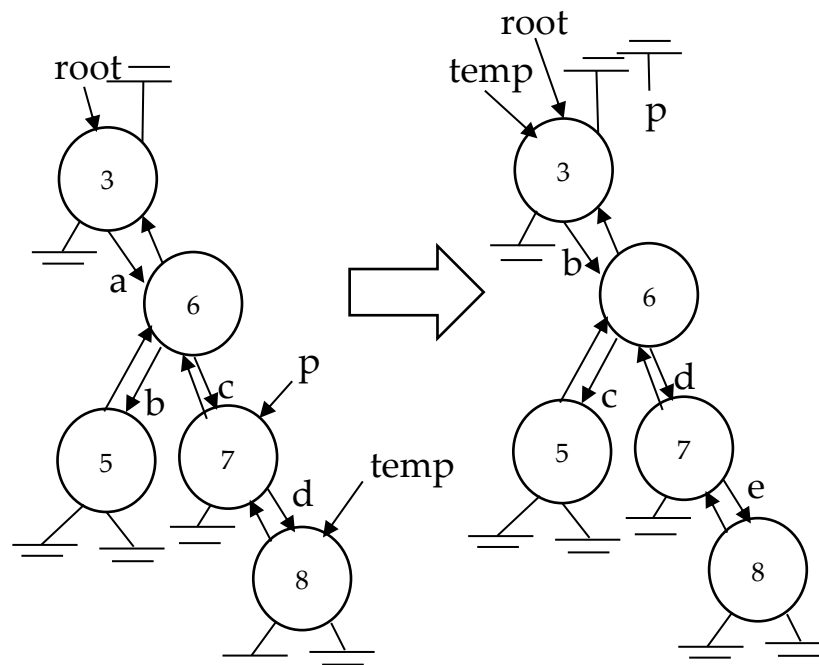


Figure 6-15: Movement of p and $temp$ when a node with larger value does not exist.

The code for method *hasPrevious* is shown in Figure 6-16. The code is the same as *hasNext*, except *right* is changed to *left* because the logic is the same.

```
1: public boolean hasPrevious() {
2:     BSTNode temp = currentNode;
3:     if (temp.left != null) {
4:         return true;
5:     }
6:
7:     BSTNode p = temp.parent;
8:     while (p != null && p.left == temp) {
9:         temp = p;
10:        p = temp.parent;
11:    }
12:    if (p == null)
13:        return false;
14:    else
15:        return true;
16: }
17: //continued in Figure 6-17.
```

Figure 6-16: Method *hasPrevious* of class *TreeIterator*.

Now, let us see the code for method *next*. It is shown in Figure 6-17. It uses the same logic as method *hasNext*. This time we not only have to determine if there is a next node that stores a larger value, but we also need to navigate to that very node and return the data stored there.

When our current node has another node as its *right* field. We navigate to *right*, then follow *left* until we can go no further. This way, we will always navigate to the node that stores data just larger than our current node. This

navigation is illustrated in Figure 6-18. Its corresponding code is in line 3-7 of Figure 6-17.

```

1:  public int next() throws Exception {
2:      BSTNode temp = currentNode;
3:      if (temp.right != null) {
4:          temp = temp.right;
5:          while (temp.left != null) {
6:              temp = temp.left;
7:          }
8:      } else {
9:          BSTNode p = temp.parent;
10:         while (p != null && p.right == temp) {
11:             temp = p;
12:             p = temp.parent;
13:         }
14:         temp = p;
15:     }
16:     if(temp == null) //hasNext() == false
17:         throw new NoSuchElementException();
18:     currentNode = temp;
19:     return currentNode.data;
20: }
21: //continued in Figure 6-19.

```

Figure 6-17: Code for method *next* of class *TreeIterator*.

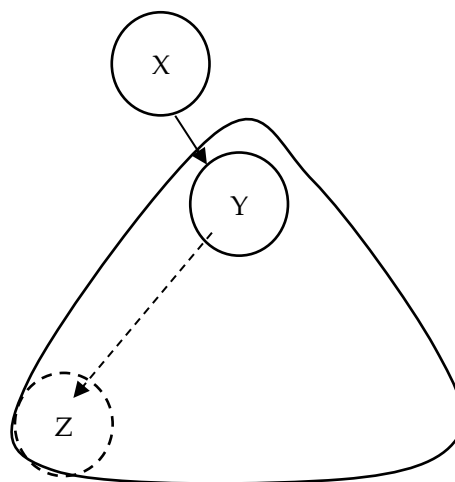


Figure 6-18: Finding node Z, with a value just larger than X, when our current node, X, has another node as its *right*.

When the *right* field of our current node does not connect to any node, we search the tree upwards, just like what we did with method *hasNext*, until we find the ancestor node that has a larger value, or until there is no more link to follow. This part of the code (line 10-13 of Figure 6-17) is the same as in method *hasNext*.

After finding the next node, method *next* then does some additional work from method *hasNext* by throwing an exception if the next node does not exist. Otherwise, it updates the current node and return the value stored inside the new current node. This is implemented in line 16-20 of Figure 6-17.

Method *previous* of our iterator (code shown in Figure 6-19) works the same way as *next()*. The code mainly switches *left* and *right*. But there is one important point to note. That is, the data returned by the method must be the data before *currentNode* is updated.

Lastly, we add method *set* to our tree iterator to allow the iterator to set the value stored in the current node (code shown in Figure 6-20).

```
1: public int previous() throws Exception {
2:     BSTNode temp = currentNode;
3:     int d = currentNode.data;
4:     if (temp.left != null) {
5:         temp = temp.left;
6:         while (temp.right != null) {
7:             temp = temp.right;
8:         }
9:     } else {
10:        BSTNode p = temp.parent;
11:        while (p != null && p.left == temp) {
12:            temp = p;
13:            p = temp.parent;
14:        }
15:        temp = p;
16:    }
17:    if(temp == null) //hasPrevious() == false
18:        throw new NoSuchElementException();
19:    currentNode = temp;
20:    return d;
21: }
22: //continued in Figure 6-20.
```

Figure 6-19: Code for method *previous* of class *TreeIterator*.

```
1: public void set(int value) {
2:     currentNode.data = value;
3: }
4: } //end of class TreeIterator.
```

Figure 6-20: Code for method *set* of class *TreeIterator*.

Now, let us look at the implementation of binary search tree. We will implement it as class *BST*, with the following methods:

- *findMin()*: return iterator that marks the node that stores the minimum value.

- *find(Type v)*: return an iterator that marks the node that stores the given *v*. If *v* is not in the tree, this method returns null.
- *insert(Type v)*: add *v* as a new data inside the tree. This method returns an iterator that focuses on the tree node that contains the new data *v*.
- *remove(Type v)*: remove a node that stores *v* from the tree.

The class structure of *BST* (without methods) is shown in Figure 6-21. Our binary search tree contains a root, which is a *BSTNode*, and a *size*, which indicates the number of data stored inside the tree.

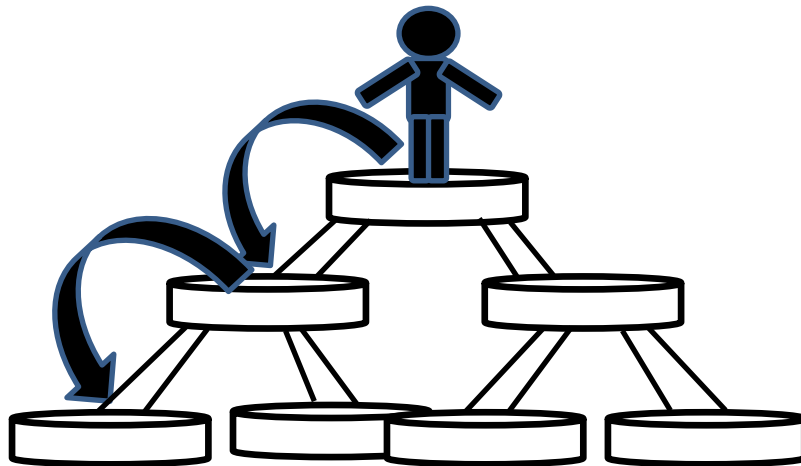
```
1: public class BST {
2:   BSTNode root;
3:   int size;
4:
5:   public BST(BSTNode root, int size) {
6:     this.root = root;
7:     this.size = size;
8:   }
9:
10: //continued in Figure 6-22.
```

Figure 6-21: Structure of a binary search tree (implementation).

Method *findMin* is shown in Figure 6-22. It starts its search at the tree root (creating a *temp* node there), then it tries to move *temp* down the tree (using its while loop) to the left as far as possible. The last possible value of *temp* then identifies the location of the smallest value in the tree.

```
1: public Iterator findMin() {
2:     BSTNode temp = root;
3:     if(temp == null)
4:         return null;
5:     while (temp.left != null) {
6:         temp = temp.left;
7:     }
8:     Iterator itr = new TreeIterator(temp);
9:     return itr;
10: }
11:
12: //continued in Figure 6-23.
```

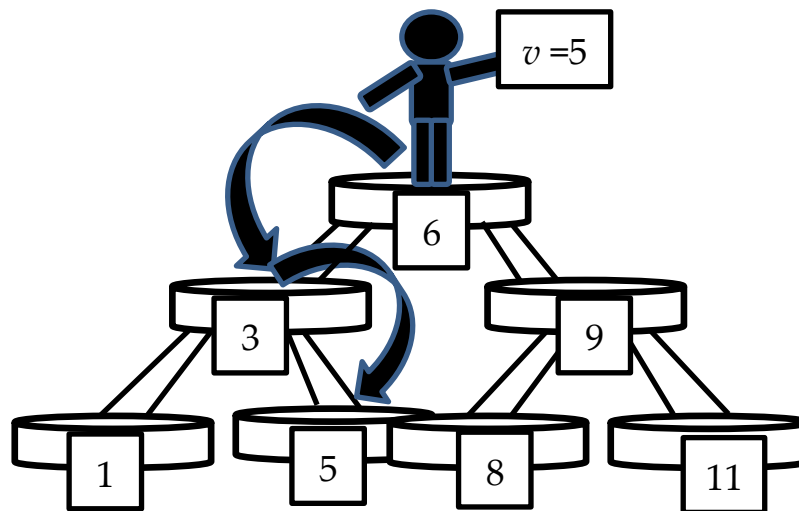
Figure 6-22: Implementation of method *findMin* for a binary search tree.



The code for method *find* is shown in Figure 6-23. It creates a *temp* node at the root and then tries to move *temp* down the tree (using its while loop) until *temp* reaches where the value *v* is stored, or until *temp* is *null* (which means *v* is not inside the tree). *temp* is then used to return the position of *v* inside the tree, or return *null* if *v* is not inside the tree.

```
1: public Iterator find(int v) {
2:     BSTNode temp = root;
3:     while (temp != null && temp.data != v) {
4:         if (v < temp.data) {
5:             temp = temp.left;
6:         } else {
7:             temp = temp.right;
8:         }
9:     }
10:    if (temp == null) // data not found
11:        return null;
12:    return new TreeIterator(temp);
13: }
14: //continued in Figure 6-26.
```

Figure 6-23: Code for method *find* of binary search tree.



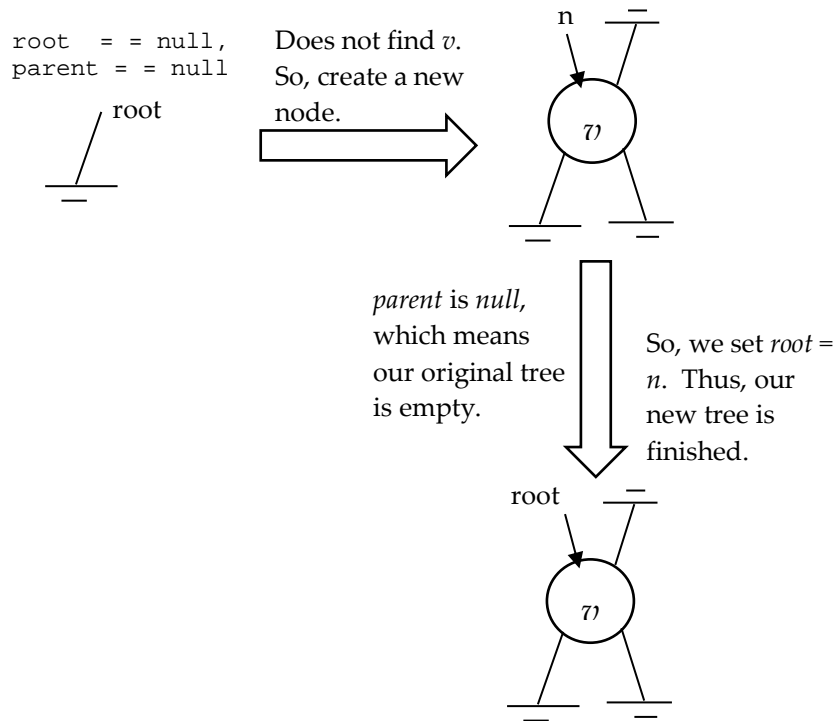
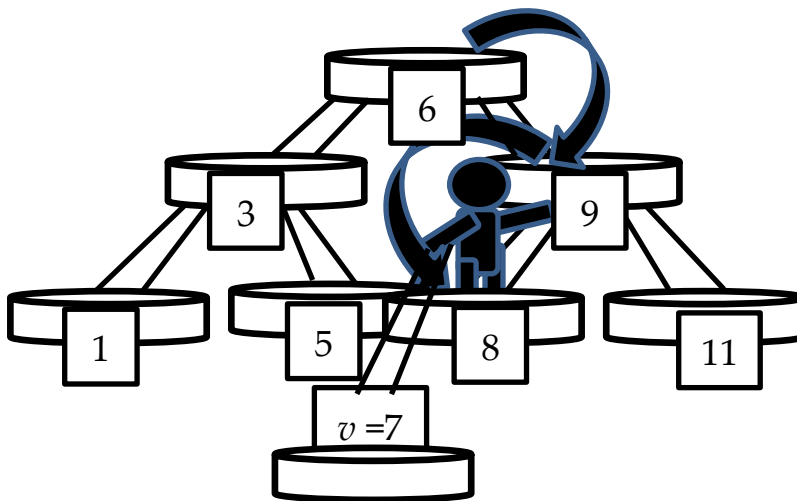
Now, let us look at method *insert*. To insert a new data, v , into a binary search tree, we do the followings:

- First, we try to find v in the tree. We can utilize the same code as method *find*, moving *temp* node down the tree (code is shown on line 8-16 of Figure 6-26).

But we will also have to keep track of the parent of *temp* in order to know where to add our new node that contains *v* later on.

- If *v* is found, we do nothing (in our implementation, we do not want to add duplicated data into the tree).
- If *v* is not found, we create a new node that contains *v* (code is shown on line 19 of Figure 6-26). The new node's *parent* is set to be the parent of *temp*, then:
 - If parent of *temp* is *null*: this is only possible when the tree has no node. So we set the root of the tree to be the new node (line 20-21 of Figure 6-26).
 - If *v* is less than the value stored inside the *parent* of *temp*, add the newly created node to the left of that parent node (line 22-23 of Figure 6-26).
 - If *v* is greater than the value stored inside the *parent* of *temp*, add the newly created node to the right of that parent node (line 24-25 of Figure 6-26).

An example showing what happens when adding *v* to an empty tree is shown in Figure 6-24. Another example, an addition of value 6 to a non-empty tree, is shown in Figure 6-25. The code for method *insert* is illustrated in Figure 6-26.

Figure 6-24: Adding new data, v , to an empty binary search tree.

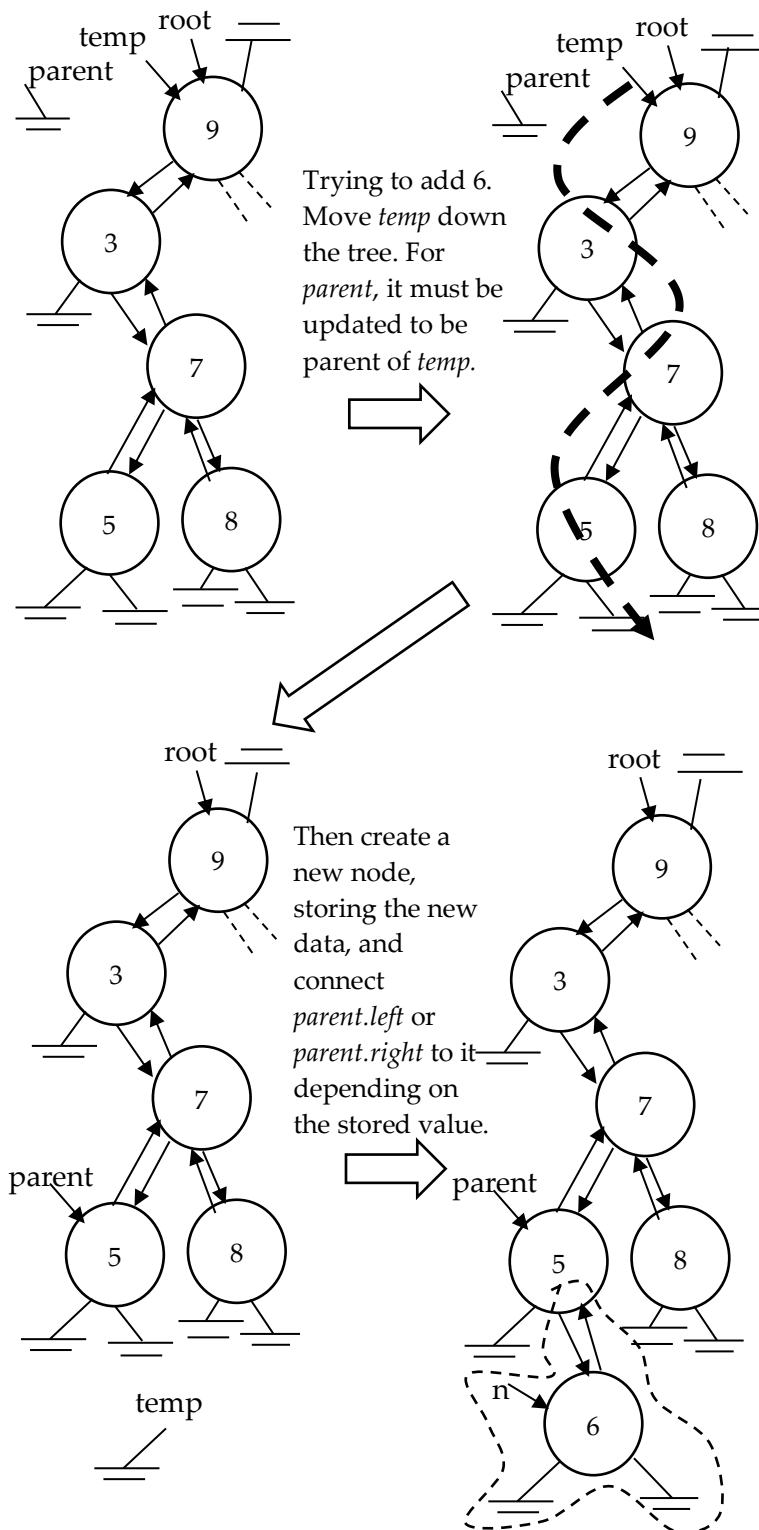


Figure 6-25: Adding 6 to a binary search tree that does not originally store 6.

```
1: public Iterator insert(int v) {
2:     BSTNode parent = null;
3:     BSTNode temp = root;
4:
5:     // This while loop is almost the same as in
6:     // method find, but it has an extra pointer
7:     // called parent.
8:     while (temp != null && temp.data != v) {
9:         if (v < temp.data) {
10:            parent = temp;
11:            temp = temp.left;
12:        } else {
13:            parent = temp;
14:            temp = temp.right;
15:        }
16:    }
17:
18:    if (temp == null) {
19:        BSTNode n = new BSTNode(v, null, null, parent);
20:        if(parent == null){
21:            root = n;
22:        } else if (v < parent.data) {
23:            parent.left = n;
24:        } else {
25:            parent.right = n;
26:        }
27:        size++;
28:        return new TreeIterator(n);
29:    } else {
30:        // we do nothing since
31:        // we don't want to add duplicated data.
32:        return null;
33:    }
34: }
35: //continued in Figure 6-33.
```

Figure 6-26: Code for inserting value v into a binary search tree.

Our next method is method *remove*. So how do we remove a value, v , stored in a binary search tree? Let us follow these steps:

- First, we try to locate where v is stored. We can use the same procedure as method *find*.

- If v is not stored in the tree, we end our execution since there is nothing to be removed from the tree.
- If v is stored in node n :
 - If n has no child node:
 - If n is the root of the tree, thus n is the only node of the tree, we can just remove it from the tree by setting $root$ to $null$ (see Figure 6-27 for an illustration and see the code on line 13 of Figure 6-33).
 - If there is a node above n , just cut the links to/from n from its $parent$ (see Figure 6-28 for an illustration and see the code on line 14-19 of Figure 6-33). The tree will no longer have access to node n . The code needs to test whether n is to the right or left of its $parent$ in order to cut the correct links.
 - If n has a right child, but no left child:
 - If n is the tree's root, make the root point to n 's right child. This will effectively bypass n . To completely disconnect n from the tree, $n.right$ then needs to be set to $null$ and the new root's parent will have to be disconnected

from n (see Figure 6-29 for an illustration and see the code on line 24-28 of Figure 6-33).

- If n is not the tree's root, and n stores a larger value than its parent, link $n.parent.right$ to $n.right$ instead of $n.parent$ to n (see Figure 6-30 for an illustration and see the code on line 29-34 of Figure 6-33).
- If n is not the tree's root, and n stores a smaller value than its parent, link $n.parent.left$ to $n.right$ instead of $n.parent$ to n (see Figure 6-31 for an illustration and see the code on line 35-40 of Figure 6-33).
- If n has a left child, but no right child, we perform the same tasks as when n has only its right child, but the tasks have to be done like in a mirror. The code is shown on line 44-65 of Figure 6-34.
- If n has both left and right child, we need to find the node that stores the smallest value in n 's right subtree (let us name that node x), then replace v in n with the value in x . After that, remove x . Removing x uses simple code

because we know x does not have its left child (see Figure 6-32 for an illustration and see code on line 66-81 of Figure 6-34).

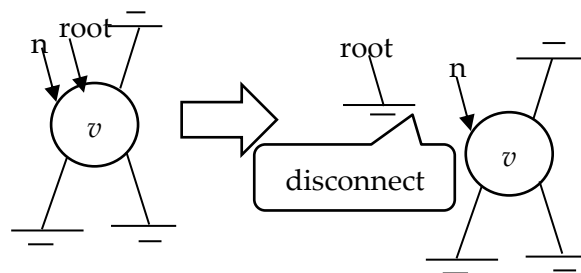


Figure 6-27: Removing v when v is in a root with no children.

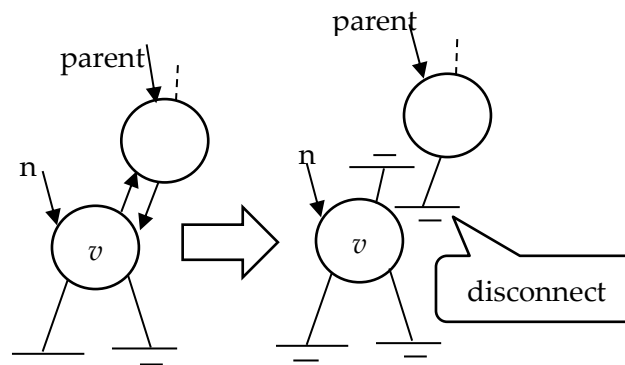


Figure 6-28: Removing v when v is in a node (not a root) with no children.

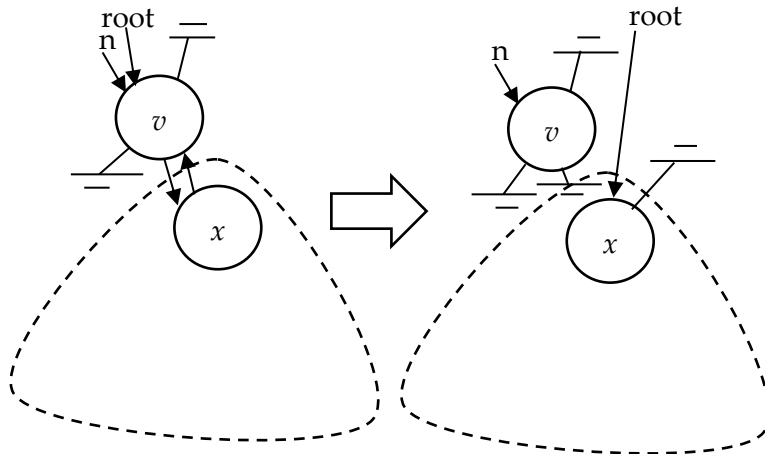


Figure 6-29: Removing v when v is in a root with right child but no left child.

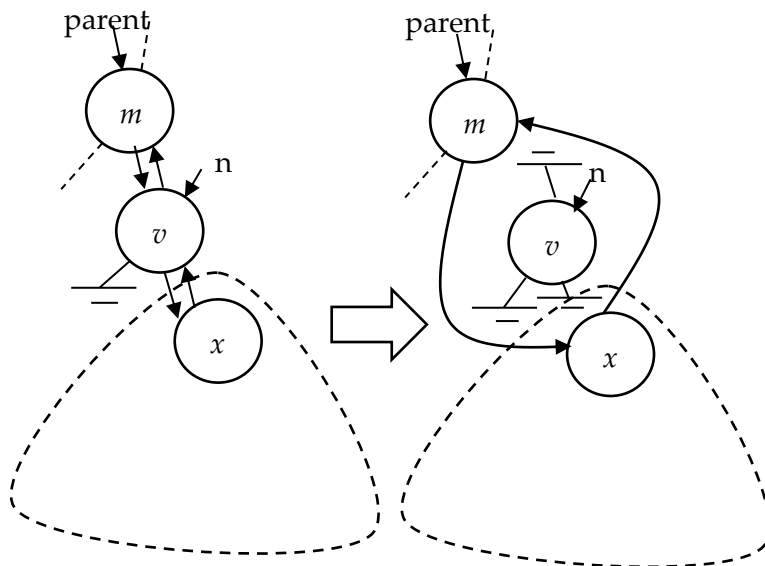


Figure 6-30: Removing v when the node, n , that stores v has only its right child, it is not the tree's root, and n stores a larger value than its parent.

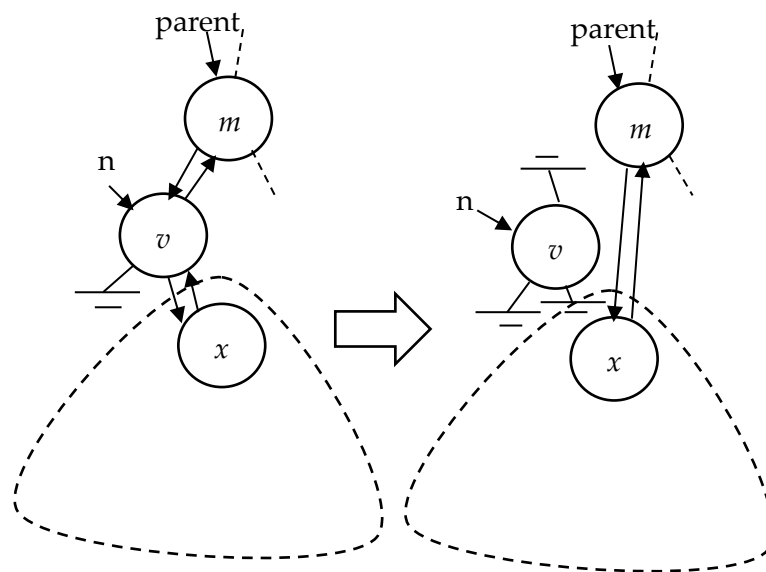


Figure 6-31: Removing v when the node, n , that stores v has only its right child, it is not the tree's root, and n stores a smaller value than its parent.

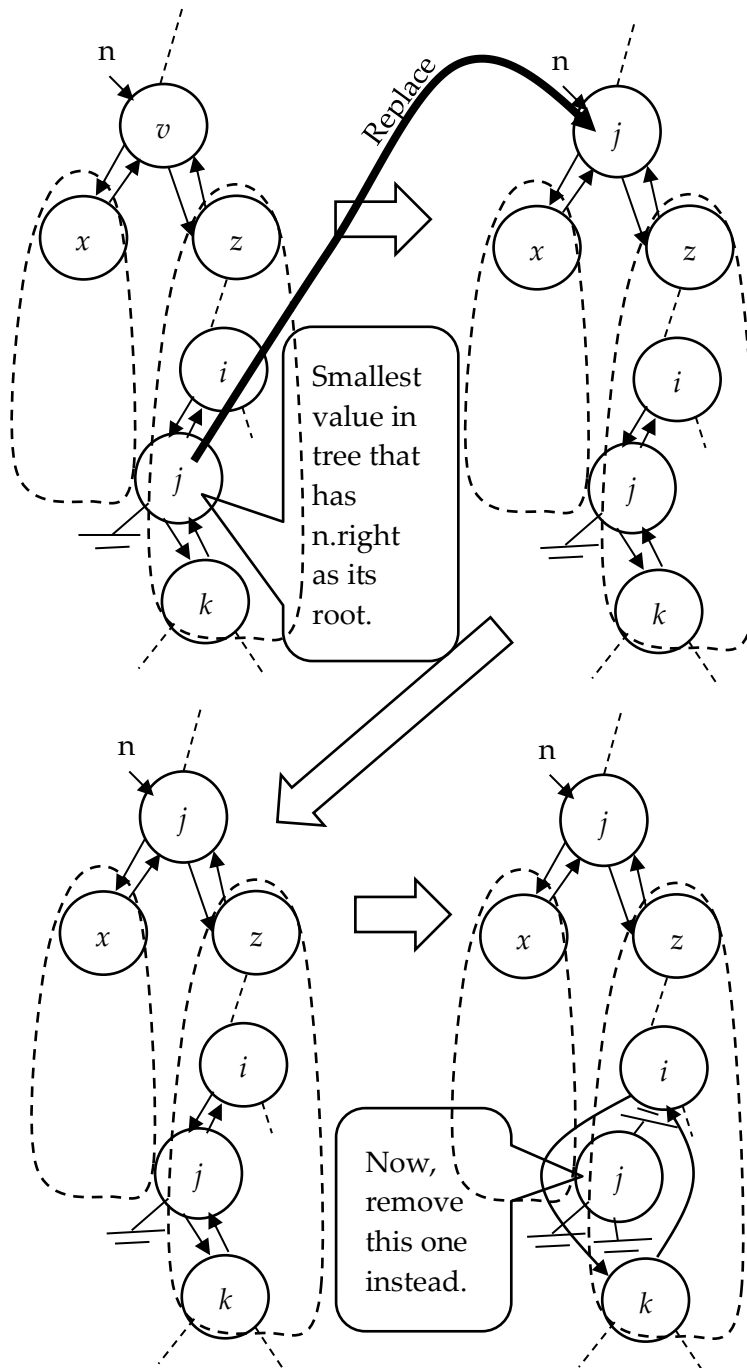


Figure 6-32: Removing v when the node, n , that stores v has both left and right child.

```
1: public void remove(int v) {
2:     BSTNode parent = null;
3:     BSTNode n = root;
4:     TreeIterator i = (TreeIterator) find(v);
5:     if (i == null) //not found, we can not remove it
6:         return;
7:     n = i.currentNode;
8:     parent = n.parent;
9:     size--;
10:    if (n.left == null && n.right == null) {
11:        //both subtrees are empty
12:        if (parent == null) {
13:            root = null;
14:        } else if (parent.left == n) {
15:            parent.left = null;
16:            n.parent = null;
17:        } else {
18:            parent.right = null;
19:            n.parent = null;
20:        }
21:    }
22:    else if (n.left == null && n.right != null){
23:        // only right child
24:        if (parent == null) {
25:            //the node to remove is a root
26:            root = n.right;
27:            root.parent = null;
28:            n.right = null;
29:        } else if (parent.right == n) {
30:            BSTNode q = n.right;
31:            q.parent = parent;
32:            parent.right = q;
33:            n.parent = null;
34:            n.right = null;
35:        } else { // parent.left == n
36:            BSTNode q = n.right;
37:            q.parent = parent;
38:            parent.left = q;
39:            n.parent = null;
40:            n.right = null;
41:        }
42:    }
43:    // This method continues in Figure 6-34.
```

Figure 6-33: Code for method *remove* of a binary search tree (part 1).

```
44:     else if (n.right == null && n.left != null) {
45:         if (parent == null) {
46:             root = n.left;
47:             root.parent = null;
48:             n.left = null;
49:         } else if (parent.right == n) {
50:             // a mirror image of line 35-40
51:             // from Figure 6-33
52:             BSTNode q = n.left;
53:             q.parent = parent;
54:             parent.right = q;
55:             n.parent = null;
56:             n.left = null;
57:         } else {
58:             // a mirror image of line 29-34
59:             // from Figure 6-33
60:             BSTNode q = n.left;
61:             q.parent = parent;
62:             parent.left = q;
63:             n.parent = null;
64:             n.left = null;
65:         }
66:     } else { // n has two subtrees
67:         BSTNode q = n.right;
68:         TreeIterator itr = findMin(q);
69:         BSTNode minInSubtree = itr.currentNode;
70:         n.data = minInSubtree.data;
71:         BSTNode parentOfMin = minInSubtree.parent;
72:         if (parentOfMin.left == minInSubtree) {
73:             parentOfMin.left = minInSubtree.right;
74:         } else { // min is the only node in the subtree
75:             parentOfMin.right = minInSubtree.right;
76:         }
77:         if (minInSubtree.right != null) {
78:             minInSubtree.right.parent = parentOfMin;
79:             minInSubtree.right = null;
80:         }
81:         minInSubtree.parent = null;
82:     }
83: } //end of method.
84: } //end of class BST.
```

Figure 6-34: Code for method *remove* of a binary search tree (part 2).

Recursive Implementation of Binary Search Tree

A recursive implementation of a binary search tree can be useful and is easier to understand for some students. In this section, the recursive implementation is shown.

For a node, we need not change anything, so we simply use class *BSTNode* (Figure 6-8).

For an iterator, we can also use class *TreeIterator* (Figure 6-11 to Figure 6-20).

What needs to be changed is class *BST*. The instance variables, constructor, and some simple methods remain unchanged, but other methods need to be changed to incorporate recursive tree traversals.

Let us name our recursive binary search tree class *BSTRecursive*. Let us break down our implementation into parts. First, the code for our instance variables and simple methods is shown in Figure 6-35.

Now, let us look at method *findMin* of this recursive implementation. The code is shown in Figure 6-36. For the method to be called recursively, we have to write a new method that takes a parameter. The parameter is the node that the method uses to start searching its tree. Method *findMin* will be recursively called, each time its parameter changes to go left down the tree. It stops when

there is no left branch to use as the method parameter. When this happens, our current method parameter is the node that stores the smallest data. This is just like using a loop, but we just change the method parameter as the method gets called instead.

```
1: public class BSTRecursive {
2:     BSTNode root;
3:     int size;
4:
5:     public BSTRecursive(BSTNode root, int size) {
6:         this.root = root;
7:         this.size = size;
8:     }
9:
10:    public boolean isEmpty() {
11:        return size == 0;
12:    }
13:
14:    public void makeEmpty() {
15:        root = null;
16:        size = 0;
17:    }
18:    // continued in Figure 6-36.
```

Figure 6-35: Instance variables and simple methods of a recursive binary search tree.

```
1: public Iterator findMin() {
2:     return findMin(root);
3: }
4:
5: public Iterator findMin(BSTNode n) {
6:     if (n == null)
7:         return null;
8:     if (n.left == null) {
9:         Iterator itr = new TreeIterator(n);
10:        return itr;
11:    }
12:    return findMin(n.left);
13: }
14: // continued in Figure 6-37.
```

Figure 6-36: Method *findMin* of class *BSTRecursive*.

Method *find* is also implemented with the same idea as its non-recursive version. Like method *findMin*, we need to write another method that can take an extra parameter, so that we can supply the starting point of our tree search for each call. Again, we change parameter value, just like changing the value in each loop, as each instance of the method gets called. The code is shown in Figure 6-37.

```
1:  public Iterator find(int v) {
2:      return find(v, root);
3:  }
4:
5:  public Iterator find(int v, BSTNode n) {
6:      if (n == null)
7:          return null;
8:      if (v == n.data)
9:          return new TreeIterator(n);
10:     if (v < n.data)
11:         return find(v, n.left);
12:     else
13:         return find(v, n.right);
14: }
15: // continued in Figure 6-38.
```

Figure 6-37: Method *find* of class *BSTRecursive*.

For method *insert*, it also uses the same idea as its iterative counterpart. That is, find the node that will be the parent of the node that stores the new data, then connect that node to the node with the new data.

Our code locates this parent node by updating the 4th parameter in each call to *insert*, until no more call is possible. The code is shown in Figure 6-38. The method

returns *BSTNode* instead of an iterator, however. We need to return *BSTNode* because we need to use *BSTNode* to update the tree branches when *insert* is repeatedly called. *TreeIterator* cannot help us with such update. The mentioned branch update is shown on line 13 and 15 of Figure 6-38. Instead of just calling *insert*, we update *n.left* and *n.right* to be the result of *insert*.

```
1: public BSTNode insert(int v) {
2:     root = insert(v, root, null);
3:     return root;
4: }
5:
6: // return the node n after v was added into the
7: // tree.
8: public BSTNode insert(int v, BSTNode n, BSTNode
9: parent) {
10:     if (n == null) {
11:         n = new BSTNode(v, null, null, parent);
12:         size++;
13:     } else if (v < n.data) {
14:         n.left = insert(v, n.left, n);
15:     } else if (v > n.data) {
16:         n.right = insert(v, n.right, n);
17:     }
18:     return n;
19: }
20: // continued in Figure 6-43.
```

Figure 6-38: Method *insert* of class *BSTRecursive*.

Let us focus on line 15 of Figure 6-38. On this line of code, we cannot just call *insert* without setting *n.right*. Let us see what happens if line 15 does not set *n.right*. Let us assume we have a tree with one node, containing value 1. If we call *insert(5)* on this tree, what happens is shown in Figure 6-39 .

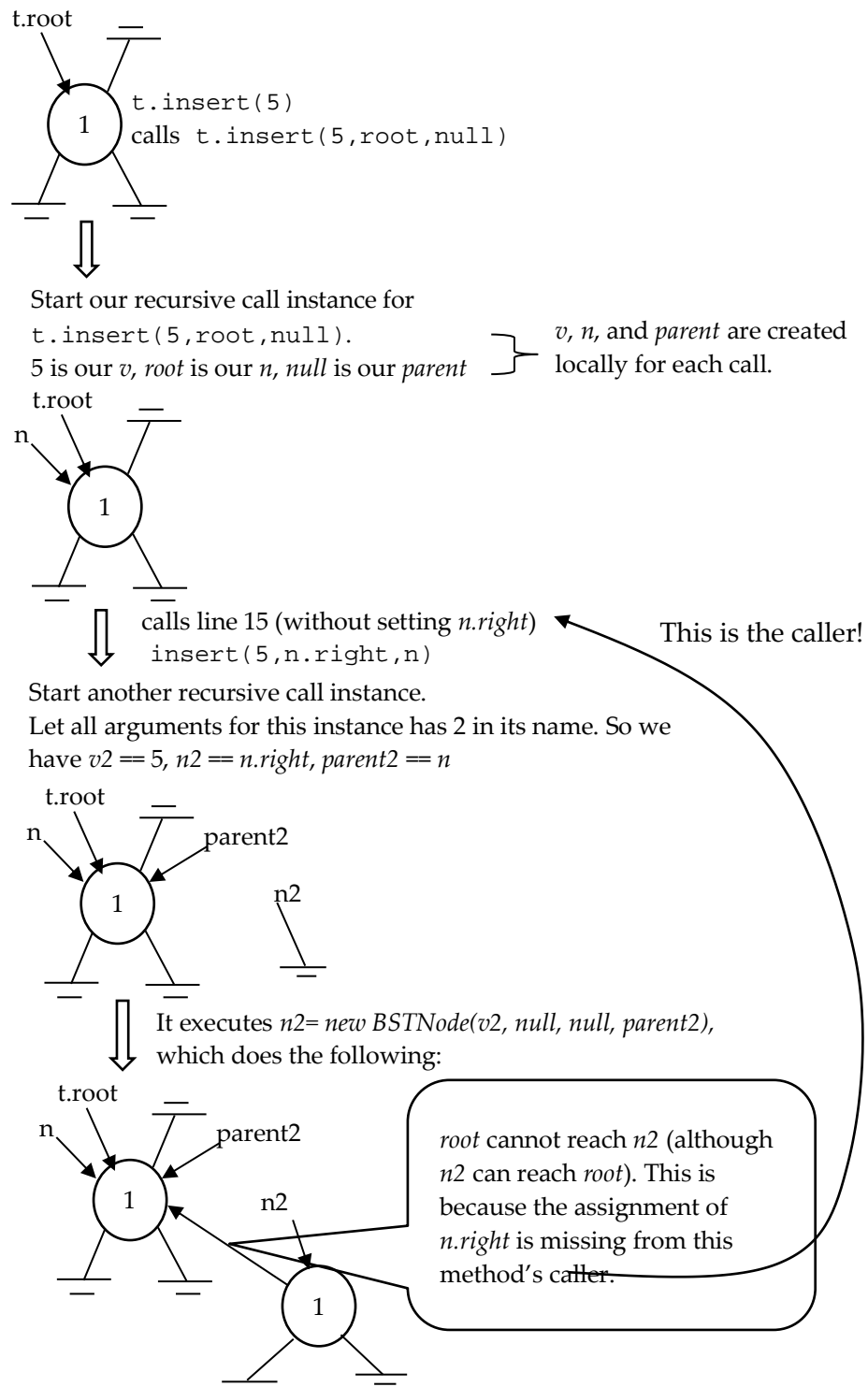


Figure 6-39: The tree is not modified properly if *n.right* is not used to store the result of *insert*.

A same type of mistake can cause problem in our main method, even with the assignment of *n.left* and *n.right* in Figure 6-38. For example, see the code for *main* method in Figure 6-40. The code first creates an empty tree, then it tries to add a new node with 1 inside.

```

1: public static void main(String[] args){
2:     BSTRecursive t = new BSTRecursive(null,0);
3:     t.insert(1);
4: }

```

Figure 6-40: Incorrect use of method *insert*.

How this code operate is shown in Figure 6-41.

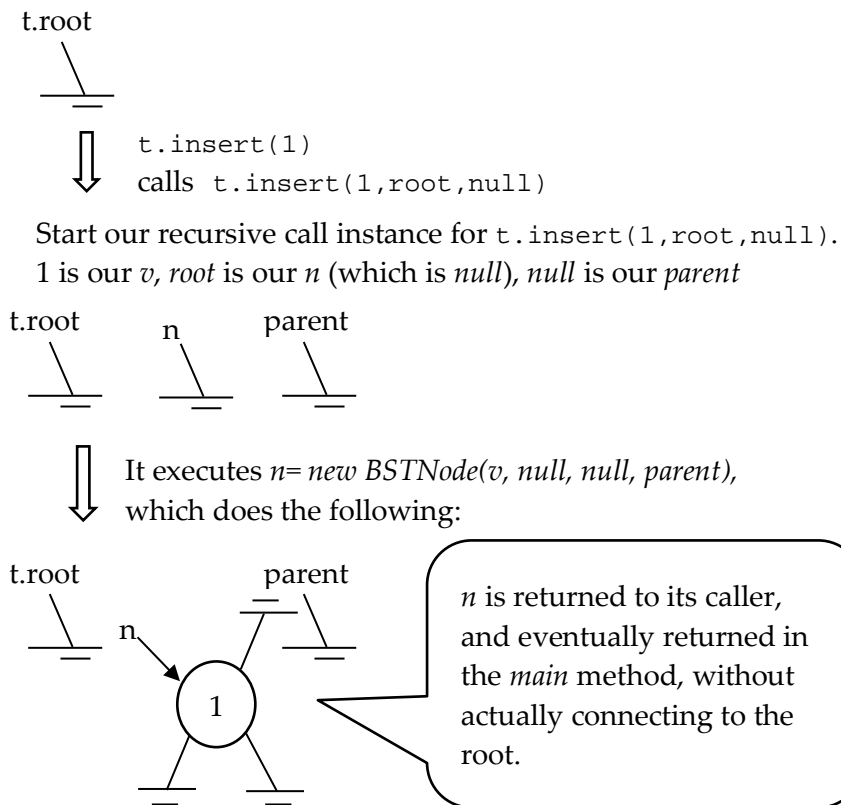


Figure 6-41: How the code in Figure 6-40 works.

The reason the code fails to work is that the original tree is not set to connect to the newly created node. Therefore, we need to set the connection from the original tree. A correct version of code in Figure 6-40 is shown in Figure 6-42. To sum up, when a part of a tree is changed due to recursive call(s), you must set the connection from the unchanged part of the tree to that new part.

```
1: public static void main(String[] args){
2:     BSTRecursive t = new BSTRecursive(null,0);
3:     t.root = t.insert(1);
4: }
```

Figure 6-42: Correction of code from Figure 6-40.

Now. Let us move on to method *remove*. The idea is exactly the same as our iterative version, and the code reflects the idea.

The code is shown in Figure 6-43. Similar to *insert*, it needs to set *n.left* and *n.right* (see line 12 and 14 of the code) in order to connect our tree to its changed part.

```
1: public BSTNode remove(int v) {
2:     return remove(v, root, null);
3: }
4:
5: // return the node n after v was removed from the
6: //tree
7: public BSTNode remove(int v, BSTNode n, BSTNode
8: parent) {
9:     if (n == null)
10:        ; // do nothing, there is nothing to be removed
11:     else if (v < n.data) {
12:         n.left = remove(v, n.left, n);
13:     } else if (v > n.data) {
14:         n.right = remove(v, n.right, n);
15:     } else {
16:         if (n.left == null && n.right == null) {
17:             n.parent = null; //disconnect from above
18:             n = null; //disconnect from below
19:             size--;
20:         } else if (n.left != null && n.right == null) {
21:             BSTNode n2 = n.left;
22:             n2.parent = parent;
23:             n.parent = null; //disconnect from above
24:             n.left = null; //disconnect from below
25:             n = n2; //change to the node below
26:             //to prepare for connection from parent
27:             size--;
28:         } else if (n.right != null && n.left == null) {
29:             BSTNode n2 = n.right;
30:             n2.parent = parent;
31:             n.parent = null; //disconnect from above
32:             n.right = null; //disconnect from below
33:             n = n2; //change to the node below
34:             //to prepare for connection from parent
35:             size--;
36:         } else {
37:             TreeIterator i;
38:             i = (TreeIterator) findMin(n.right);
39:             int minInRightSubtree = i.currentNode.data;
40:             n.data = minInRightSubtree;
41:             n.right = remove(minInRightSubtree, n.right, n);
42:         }
43:     }
44:     return n;
45: } // end of method.
46: } // end of class BSTRecursive.
```

Figure 6-43: Code for method *remove* of class *BSTRecursive*.

Recursive Tree Traversal

By visiting each the nodes recursively, we can go through every data in a tree (the tree does not have to be a binary search tree) in a few number of ways:

- Preorder traversal: we visit the root, then left subtree, then right subtree. Within a subtree, we visit its root first, then its left subtree, then its right subtree, and so on. An example tree (this one is a binary search tree) is shown in Figure 6-44. For simplicity, we only draw downward pointers. Let us name each node after a data stored inside it.

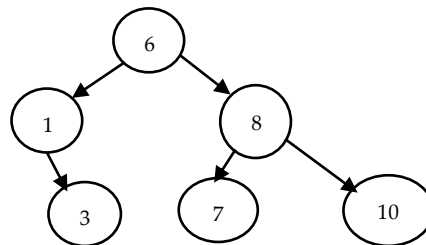


Figure 6-44: Tree for use with all traversal examples.

A preorder visit of the data will give us the following sequence of node visiting:

- 6, left subtree of 6, right subtree of 6

Expanding the sequence within the subtree traversal recursively, we get:

- 6, (1, left subtree of 1, right subtree of 1), (8, left subtree of 8, right subtree of 8)

- 6, (1, null, (3, left subtree of 3, right subtree of 3))(8, (7, left subtree of 7, right subtree of 7), (10, left subtree of 10, right subtree of 10))
 - 6, (1, null, (3, null, null))(8, (7, null, null), (10, null, null))
 - 6, 1, 3, 8, 7, 10
- Postorder traversal: we visit the left subtree, then right subtree, then root. Within a subtree, we visit its left subtree first, then its right subtree, then its root, and so on. Using the tree in Figure 6-44, our traversal sequence is as follows:
 - Left subtree of 6, right subtree of 6, 6
 - (left subtree of 1, right subtree of 1, 1), (left subtree of 8, right subtree of 8, 8), 6
 - (null, (left subtree of 3, right subtree of 3, 3), 1), ((left subtree of 7, right subtree of 7, 7), (left subtree of 10, right subtree of 10, 10), 8), 6
 - (null, (null, null, 3), 1), ((null, null, 7), (null, null, 10), 8), 6
 - 3, 1, 7, 10, 8, 6
- Inorder traversal: we visit the left subtree, then the root, then the right subtree. Within each subtree, we visit its left subtree, then its root, then its right subtree, and so on. Using the tree in Figure 6-44, our traversal sequence is as follows:
 - Left subtree of 6, 6, right subtree of 6

- (left subtree of 1, 1, right subtree of 1), 6, (left subtree of 8, 8, right subtree of 8)
- (null, 1, (left subtree of 3, 3, right subtree of 3)), 6, ((left subtree of 7, 7, right subtree of 7), 8, (left subtree of 10, 10, right subtree of 10))
- (null, 1, (null, 3, null)), 6, ((null, 7, null), 8, (null, 10, null))
- 1, 3, 6, 7, 8, 10

With a binary search tree, inorder traversal visits data from small to large.

The implementations of these traversals are straightforward. Figure 6-45 shows code (that can be part of class *BSTRecursive*) for printing all data from a tree using preorder traversal and inorder traversal.

```
1: public void preOrderPrint() {
2:     preOrderPrint(root);
3: }
4:
5: public void inOrderPrint() {
6:     inOrderPrint(root);
7: }
8:
9: public void preOrderPrint(BSTNode n){
10:     if (n == null)
11:         return;
12:     System.out.println(n.data);
13:     preOrderPrint(n.left);
14:     preOrderPrint(n.right);
15: }
16:
17: public void inOrderPrint(BSTNode n){
18:     if (n == null)
19:         return;
20:     inOrderPrint(n.left);
21:     System.out.println(n.data);
22:     inOrderPrint(n.right);
23: }
```

Figure 6-45: code for preorder and inorder printing of data in a tree.

Notice that although the code is simple, and we do not require the use of any iterator, an iterator is still useful if we want to mark the location for use later in other parts of the program.

Breadth-First Tree Traversal

All tree traversals that we have looked at so far do not provide any way for us to go through our data level-by-level. For example, with the tree in Figure 6-46, we may want to look through 6, 1, 8, 3, 7, 10 in sequence.

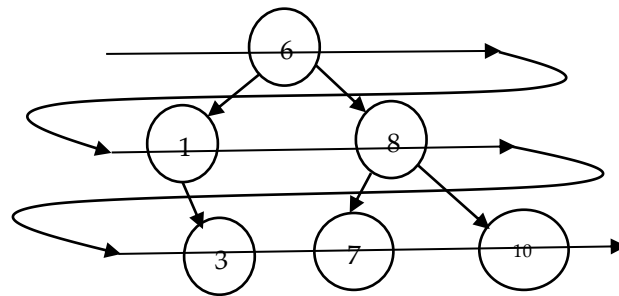


Figure 6-46: Search sequence by level of a tree.

In order to handle this special kind of tree traversal, we have to utilize another data structure, a queue. In fact, we will be using 2 queues to help us. The two queues are called:

- *thisLevel*: store nodes that are in the current level of our tree.
- *nextLevel*: store nodes that are in the next level of our tree.

Using the tree in Figure 6-46 as our example:

- When our traversal starts, both queues are empty.
- The root of the tree is then read. The node is then put into *thisLevel* (Figure 6-47).

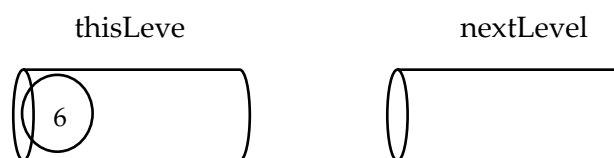


Figure 6-47: Putting *root* into *thisLevel* queue.

- Then, we remove a node from *thisLevel*. This is where we can read the node's data. We then put the node's *left* and *right* node into *nextLevel*. Repeat until *thisLevel* is empty. From our example, we only have one node to remove from *thisLevel* at this state. Therefore, our program reads data 6 from the node and the state of the queues after *thisLevel* becomes empty is shown in Figure 6-48.

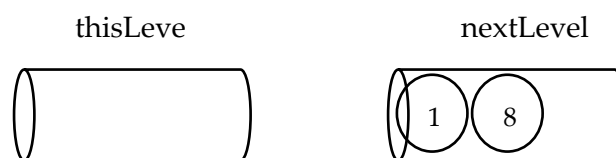


Figure 6-48: Removing a node from *thisLevel* queue and put its *left* and *right* nodes into *nextLevel* queue.

- At this state, remove all nodes from *nextLevel* and put them in *thisLevel* (shown in Figure 6-49).



Figure 6-49: Removing all nodes from *nextLevel* queue and putting them in *thisLevel* queue.

- Then we start our process again. We remove each node from *thisLevel* (and visit its data) and put each node's *left* and *right* node into *nextLevel* (not putting null though). From our example in Figure 6-49, the data that are visited are 1 and 8. The state of the queues after all the nodes are removed from *thisLevel* is shown in Figure 6-50.

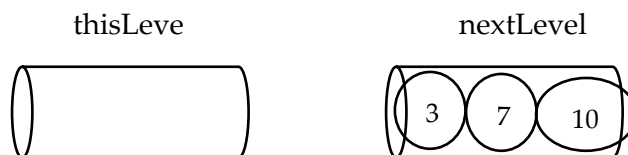


Figure 6-50: The queues after nodes with 1 and 8 are removed.

- At this state, remove all nodes from *nextLevel* and put them in *thisLevel*. This is shown in Figure 6-51.

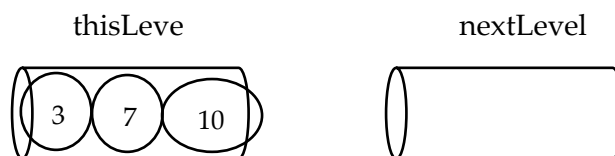


Figure 6-51: Removing all nodes from *nextLevel* queue and putting them in *thisLevel* queue for the 2nd time.

- We then start our process again by removing each node from *thisLevel* (and reading their information), and putting each node's *left* and *right* nodes into *nextLevel*. From our example, the data that are read are 3, 7, 10 respectively. There are no nodes to be put in *nextLevel*. Both queues are now empty and we end our traversal.

From the example, the sequence of read data is 6, 1, 8, 3, 7, 10. Which is from left to right, one level to the next.

We will leave the implementation of this breadth-first traversal as an exercise for readers.

Exercises

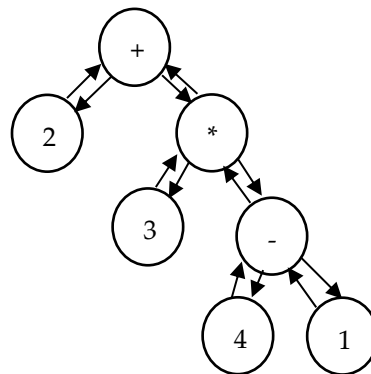
1. For class *BST*, write code for a method that performs the following task:
 - a. It receives a value *v*, as its method parameter.
 - b. The method returns an integer in the tree that is just lower than *v*, if such value exists.
2. For class *BST*, write code for a method that performs the following task:
 - a. It receives a value *v*, as its method parameter.
 - b. The method returns an integer in the tree that is just larger than *v*, if such value exists.
3. For class *BST*, write a non-recursive code for a method that checks whether our tree is a binary search tree.

4. For class *BSTRecursive*, write a recursive code for a method that checks whether our tree is a binary search tree.
5. For class *BSTNode* write code for method *public boolean isLeaf()*. This method tests whether the node is a leaf node.
6. For class *BSTRecursive*, write code for method *private int height(BSTNode n)*. This method calculates the height of a subtree in a recursive fashion. Note that:
 - A tree is one level higher than its highest subtree.
 - The height of an empty subtree is -1.
7. For class *BSTRecursive*, write code for method *public int maxNumNodes()*. This method calculates the maximum number of nodes that the tree can contain based on its current height.
8. For class *BST*, write the code for method:
 - a. *public int numNodes()*: this non-recursive method calculates the number of nodes in our tree.
 - b. *public int numLeaves()*: this non-recursive method count the number of leaves in the tree.
9. For class *BSTRecursive*, write the code for method:
 - a. *public int numNodes()*: this recursive method calculates the number of nodes in our tree.
 - b. *public int numLeaves()*: this recursive method count the number of leaves in the tree.

BSTNode findParent (BSTNode n, BSTNode d, BSTNode parent): this method finds the parent of a given node (in a subtree) without following parent link up from that node. It returns *null* if no parent can be found.

- *n* is the node that roots the subtree we want to work on.
- *d* is the node that we want to find its parent node.
- *parent* is our temporary node that will move down the tree. Its final value will be our return value.

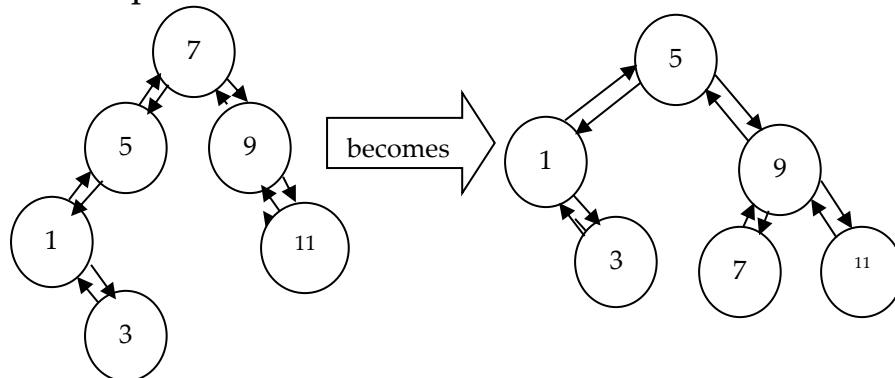
10. An expression tree is a tree that represents an arithmetic expression. It is not a binary search tree. For example, to represent $2+3*(4-1)$, we can use the following tree:



Each node now stores a String instead of an integer. Write a method *double evaluate(BSTNode n)*. This method receives a root of our expression tree and calculates the value of the expression.

11. Explain, with drawings, how you would generate the most balanced binary search tree possible from any given binary search tree. **The most balanced tree possible has all data filled from left to right, top to bottom, except at the last level.** It must still be a binary search tree. You are allowed to create any new data structures and objects.

For example:



Write code for method

public void constructMostBalanced() of class *BSTRecursive* according to your explanation. The method must reconstruct the tree from an existing tree so that the most balanced arrangement possible is obtained.

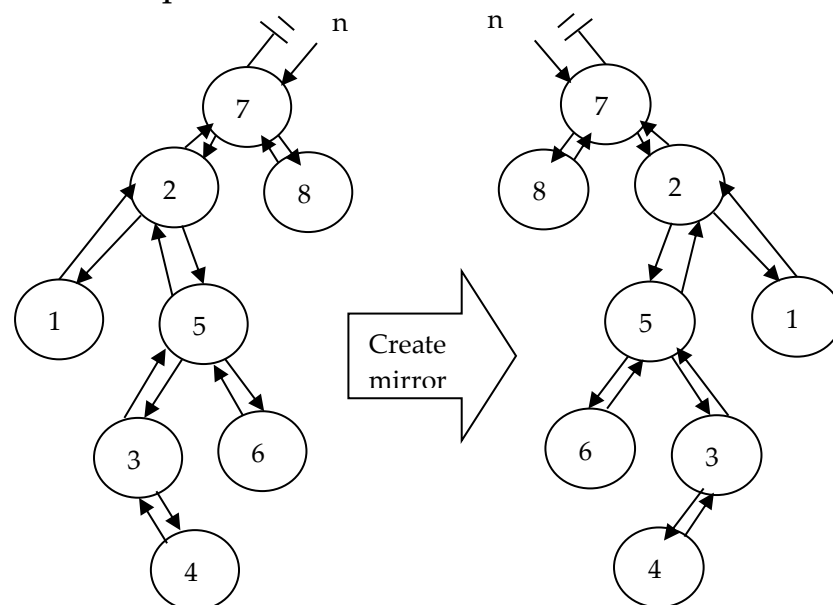
12. Write code for method *boolean sameData(BST t1, BST t2)* of a generic class that can access *BST* and related classes. This method returns true if *t1* and *t2* have the same data. It returns false otherwise. After

the method finishes its execution, $t1$ and $t2$ must not change.

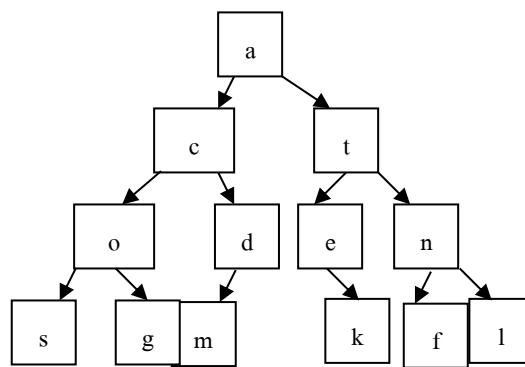
13. Write code for method *boolean same(BST t1, BST t2)* of a generic class that can access *BST* and its related classes. This method returns true if $t1$ and $t2$ have the same shape and the same content. It returns false otherwise. After the method finishes its execution, $t1$ and $t2$ must not change.

14. Write code for method *BSTNode createMirror(BSTNode n)* of class *BSTRecursive*. This method creates a completely new tree that looks to be a mirror image of a tree that has n as its root. It returns the root of the new tree. The new tree is a binary search tree, but data is arranged from large to small. The original tree must not be changed.

For example:



15. For a generic class that has access to class *BST* and all other related classes, write code for method ***public int average(BST t)***. This method calculates the average value from all nodes in the tree.
16. Let us look at the following binary tree:



What is the sequence of data when we traverse this tree with:

- Preorder traversal
 - Inorder traversal
 - Postorder traversal
17. Explain how we can convert a postfix expression to an expression tree. Write your pseudocode for this conversion.
18. For class *BST* and *BSTRecursive*, write code for method ***findMax()***. This method returns an iterator that marks the node that stores the maximum value. The code in class *BST* must be non-recursive, while the code in class *BSTRecursive* must be recursive.
19. For class *BST*, write code for method ***public void combine(BST t)***. This method combines *this* with a

binary search tree, t . The result is *this*, with all data from t also inside it. *this* must still be a binary search tree.

Chapter 7 : Hash Table

A hash table is basically an array that allows for fast data searching. If done right, we can find our data in constant time. Due to the way hash table data are arranged, there is no sorting of data. In fact, the only operations we will use on a hash table are:

- Find a data.
- Add a new data.
- Remove a data.

To find a data X (after that, we can try to add or remove X from our hash table), we do the followings:

- Use part of X as a “key”.
- Use the “key” as an input for our hash function. A hash function takes a key as its input, and return the array index, i , of a position that X should be in.
- Look at the array at position i .
 - If X is stored in that position, we have found it and we can decide to remove it from the array.
 - If X is not stored in that position, it means X does not exist in the array. We can then decide whether to put X in that array position.

An example of a hash table and its usage is shown in Figure 7-1 . Our hash table is an array that stores cars. A

car has many information, but to identify each car, we may only need some information. In our example, let us use brand, model, and color to identify a car in our array. We will use brand, model, and color as a key for our hash function when we want to find/add/remove a car from our array. Thus from Figure 7-1:

- $hash("Toyota", "yaris", "black")$ returns 1.
- $hash("Nissan", "sunny", "black")$ returns 3.
- $hash("Toyota", "camry", "black")$ returns 4.

To find a car, we call $hash(brand, model, color)$. An integer returned from $hash$ is the position of the car.

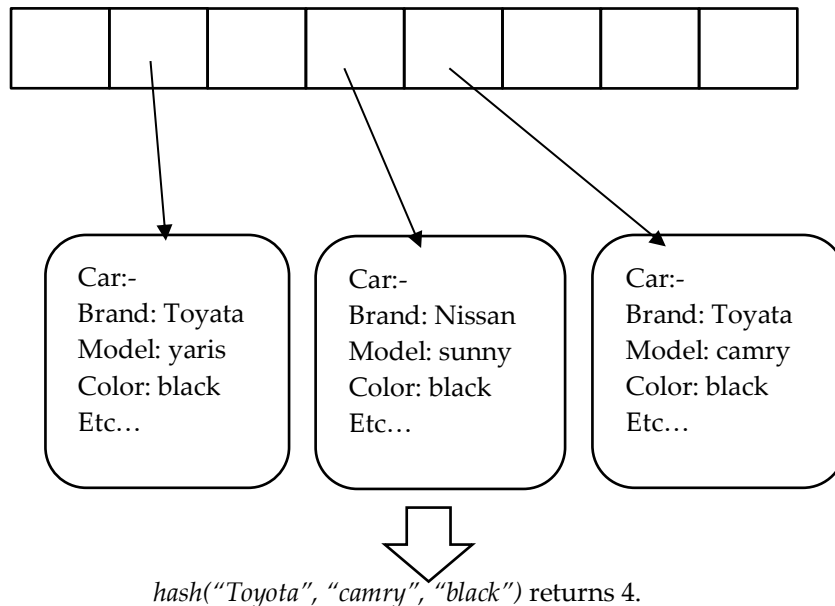


Figure 7-1: A hash table example.

Everything depends on our hash function. A hash function should:

- be fast to calculate. The runtime of a hash function directly dictates the speed that our data can be found.
- be appropriate for a given type of data (if calculated from data). Each type of data will certainly give us different keys.
- returns different values when supplied with different keys.

Designing A Hash Function

When designing a hash function, we try to make our hash function distribute its data all over the array. We generally use the following steps:

1. Transform our key into an integer.
2. Make our integer more widely distributed with some rules of transformation.
3. Transform the number we got in step 2 into array index.

Transforming our key

To transform our key into an integer, we can:

- convert it to integer using available methods.
- convert true to 1, false to 0.
- convert a string or strings to its ASCII values.
- combine integer conversions of various pieces of data into one integer.

Let us see an example. For this example, a key “john” can be converted to “j”+“o”+“h”+“n” = 106 + 97 + 118 + 110 = 431 according to each letter’s ASCII value.

But this may cause a problem or two.

- Since an ASCII value is at most 127, therefore the sum of 4 characters never exceeds $127 * 4 = 508$. So, even though we may have an array of size 10000, our data will only be stored in the first 508 array slots.
- “john” and “hojn” will be represented by the same integer value. The array slot calculated from “john” and “hojn” will therefore be the same.

A better approach is to regard each character as a digit in a base-26 number (there are 26 characters in English alphabets so we use base-26). Using this new approach, “john” gets converted to $106 * (26^3) + 97 * (26^2) + 118 * (26^1) + 110 * (26^0) = 1863056 + 65572 + 3068 + 110 = 1931806$. This is obviously more usable with a larger array. Also, “hojn” is converted to $118 * (26^3) + 97 * (26^2) + 106 * (26^1) + 110 * (26^0) = 2073968 + 65572 + 2756 + 110 = 2142406$. This gives us a different value. The code for this example is shown in Figure 7-2.

```
1: public static int f(String key) {
2:     int val =0;
3:     for(int i=0; i<key.length();i++)
4:         val = 26*val+key.charAt(i);
5:     return val;
6: }
```

Figure 7-2: A function that transforms a string into integer.

The code may seem a bit strange, but it is really calculating $(key.charAt(0) * 26 + key.charAt(1)) * 26 + key.charAt(2) \dots$

For simplicity, let us assume that our string has only 3 characters. So, the above expression can be rearranged to $key.charAt(0) * 26^2 + key.charAt(1) * 26^1 + key.charAt(0) * 26^0$, which is what we want to calculate. The code works for any number of characters. The reason we avoid method *power* in our code is that the method takes long time to run compared to our approach.

The execution of our example code may still take some time if we have a long key. We can fix this by selecting only a few characters.

Making our integer more widely distributed

Our key may already be an integer. But the integer may be too organized. This means two data have their integer representations too close to one another. In order to have the widest possible range of integers, so that we can use them as indices for our well-distributed array, we need to transform our integer.

There are several ways that we can generate more widely distributed integers from existing ones. In this book, three techniques are given.

The first technique is removing a too repetitive digit. For example, students in the same year have their ids as their unique identifiers. Student 1 and student 2 have the following ids: 5831380721 and 5830401221. These two numbers are too similar because they both have 583 at their front and 21 at their back. We can make better distributed numbers by removing 583 and 21. Thus both numbers are now 13807 and 04012.

The second technique is folding your integer. For example, if your integer is 5831380721, you split this number into 583, 1380 and 721. Then you just add them up, which results in $583 + 1380 + 721 = 2684$. Or if your integer is 5830401221, you get $583 + 0401 + 221 = 1205$. Any prior similarity between the two integers is gone. Apart from adding, you can also do something else that combines them, such as doing XOR operation and other bitwise operations.

The third technique is to divide our integer with a number and record its remainder as our transformed integer. Choosing the divisor is very important here.

- Do not choose a divisor which is a power of 10. Because the remainder will just be a few last digits (and will likely be the same for original integers with certain patterns). For example, if our original integers are 5831380721 and 5830401221, and we choose to divide each number by 100, their

remainders will be the same number, 21. This is not what we want.

- Do not choose a divisor which has a small common factor with its numerator. This is because the common factor will be a factor of the remainder as well. Hence our result will not be well distributed. As an example, let our original data be 100 and 200. If we use 55 as our divisor (value 5 is our common factor here), our result will be $100\%55 = 45$ and $200\%55 = 35$. Both the number 45 and 35 have 5 as their factor. They cannot be effectively used as positions in our array because other positions not divisible by 5 will simply not be used.

Transforming our value into array index

Once we have our integers from each data, these integers are likely to spread out. To use them as positions for our array, we simply need to make sure that their values do not fall out of possible array index range. The best way to handle this is to divide each integer with the array size and use the remainder as its corresponding array index value. But we have to be careful when using remainders, since the problem caused by the small common factor, as mentioned in the previous section, can arise. To prevent the small common factor problem, a prime number is often used as the array size.

If our hash function is well designed, the chance that 2 data will get the same position in our array is greatly

reduced. However, it is very hard to make a perfect hash function. Some data will end up being assigned to the same array position (this is called “collision”). When this happens, we have to organize these data in a systematic way to be able to store all of them in the array and be able to quickly find them later. There are two implementation ideas of a hash table that can deal with collision. They are called:

- Separate chaining
- Open addressing

Separate Chaining Hash Table

This idea can simply be explained as follows:

- Each of our array slots, instead of storing one data, stores a linked list of data instead.
- All data that have the same value from our hash function go into the same linked list.
- To find a data, use a hash function to find a linked list that stores the data, then search the data sequentially in that linked list.
- To add a new data, use a hash function to find a linked list that will (or already) store that data. Then search the list for that data.
 - If the list already has the data, we do nothing. There is no point for adding a duplicated data. A hash table has no use for duplicated copies of data, since it is only used to check whether a data is available.

- If the list does not store the data, we add the data in front of the list. Statistically, a new data is more likely to be accessed than older data. That is why we put it in front of the list.

A separate chaining hash table (implemented using circular doubly-linked lists) is illustrated in Figure 7-3. In this example, our array has 5 slots and the slots are shown vertically to allow linked lists to be drawn horizontally. Data 0, 5, and 10 are stored in the same linked list, indicating that our hash function put them in the same array slot.

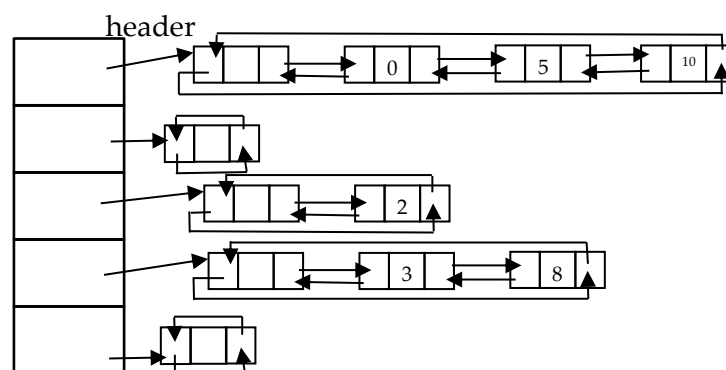


Figure 7-3: A separate chaining hash table.

Implementation of Separate Chaining Hash Table

In Java, every object can call method *hashCode*. This method maps the object's memory address to an integer value. This gives an evenly distributed integer. Our implementation makes use of this *hashCode* method. Our

code (class *SepChaining*) is shown in Figure 7-4, Figure 7-7 and Figure 7-8.

```
1: public class SepChaining{
2:     private static int DEFAULT_SIZE = 101;
3:     private static int MAXLOAD = 2;
4:     private CDLinkedList[] lists;
5:     private int currentSize =0;
6:
7:     public SepChaining(){
8:         this(DEFAULT_SIZE);
9:     }
10:
11:     public SepChaining(int size){
12:         int nextPrimeSize = nextPrime(size);
13:         lists = new CDLinkedList[nextPrimeSize];
14:         for(int i=0; i<lists.length; i++){
15:             lists[i] = new CDLinkedList();
16:         }
17:     }
18:
19:     private static boolean isPrime(int n){
20:         if(n == 2 || n == 3)
21:             return true;
22:         if(n == 1 || n % 2 == 0)
23:             return false;
24:         for(int i = 3; i*i <= n; i+= 2)
25:             if(n%i == 0)
26:                 return false;
27:         return true;
28:     }
29:
30:     private static int nextPrime(int n){
31:         if(n % 2 == 0)
32:             n++;
33:         for( ; !isPrime(n); n += 2 ){}
34:         return n;
35:     }
36:
37: // continued in Figure 7-7.
```

Figure 7-4: Fields, constructors, and utility methods for separate chaining hash table.

Figure 7-4 shows fields, constructors, and utility methods for our separate chaining hash table. Our implementation contains the following fields (line 2-5 in Figure 7-4):

- `DEFAULT_SIZE`: a default value for the number of array slots. This must be a prime number to avoid the small common factor problem.
- `MAXLOAD`: a maximum number of data that we want to store per list, on average. We do not want the average number of data (per list) to be long. This is because a list is searched sequentially. Longer list means longer search time. So, we need to check our current average number of data against `MAXLOAD`. If our value exceeds `MAXLOAD`, it means our lists are too long, we should make a larger array and redistribute our data into the new array (this is called rehash) so that lists within the new array are shorter.
- `lists`: an array that stores linked lists. This is our main data storage of our hash table. Our linked lists are circular doubly-link lists from chapter 3 (class *CDLinkedList*), but the code for class *CDLinkedList* and *DListIterator* has to be modified so that our list can store Objects, instead of integers (this modification is left for readers).
- `currentSize`: a total number of data stored in our hash table.

For constructors, we have a default one, which calls our main constructor. Our main constructor (line 11-17 in Figure 7-4) creates an array of linked list that has its number of slots equal to a prime number equal to or next to a given value (again, avoiding the small common factor problem). Once the slots are created, a list for each slot is created using the constructor of *CDLinkedList*.

To facilitate the generation of a given prime number, we have two utility methods, *isPrime* and *nextPrime* (line 19-35 of Figure 7-4).

Method *hash* (shown in Figure 7-5) returns the position that a given data is supposed to be stored in our array. It simply calls *hashCode()*, then makes sure the returned number is positive and the returned number is not outside possible positions.

```
1: public int hash(Object data){
2:     int hashValue = data.hashCode();
3:     int abs = Math.abs(hashValue);
4:     return abs%lists.length;
5: } //continued in Figure 7-7.
```

Figure 7-5: Method *hash* of separate chaining hash table.

Method *find* (shown in Figure 7-6) returns -1 if a given data is not stored in our hash table. Otherwise, it returns a non-negative value. This non-negative value is a position within a linked list that the data is stored (it is not the same value that we get from method *hash*).

```
1: public int find(Object data){
2:     int pos = hash(data);
3:     CDLinkedList theList = lists[pos];
4:     return theList.find(data);
5: } //continued in Figure 7-7.
```

Figure 7-6: Method *find* of separate chaining hash table.

Method *add* (shown in Figure 7-7) does the followings:

- It first identifies a linked list that may store our given data, by calculating its slot in our array (line 2-3 in Figure 7-7).
- Once the list is identified, it searches the list for the data using method *find* of linked list (line 4 in Figure 7-7).
 - If the data is not in the list, it adds the data in front of the list (line 4-8 in Figure 7-7), using method *insert* of *CDLinkedList*.
 - If the data is in the list, we do nothing since adding a duplicated data into a hash table does not help us with anything.
- Once the addition is done, we need to check if our current average number of data (per list) exceeds *MAXLOAD*. If so, we need to create a larger array (and new linked lists within the new array) and put all existing data from our array inside the new array. We achieve this by calling method *rehash* (also shown in Figure 7-7).

```
1: public void add(Object data){
2:     int pos = hash(data);
3:     CDLinkedList theList = lists[pos];
4:     if(theList.find(data) == -1){ // not found
5:         DListIterator itr =
6:             new DListIterator(lists[pos].header);
7:         lists[pos].insert(data, itr);
8:         currentSize++;
9:     }
10:    if(currentSize/lists.length >= MAXLOAD){
11:        rehash();
12:    }
13: }
14:
15: public void rehash(){
16:     CDLinkedList[] oldLists = lists;
17:     int newLength = nextPrime(2*lists.length);
18:     lists = new CDLinkedList[newLength];
19:     for(int i=0; i<lists.length; i++){
20:         lists[i] = new CDLinkedList();
21:     }
22:     for(int i=0; i<oldLists.length; i++){
23:         DListIterator itr;
24:         itr = new DListIterator(oldLists[i].header);
25:         while(itr.currentNode.nextNode !=
26:             oldLists[i].header){
27:             add(itr.next());
28:         }
29:     }
30: } //continued in Figure 7-8.
```

Figure 7-7: Method *add* and *rehash* of separate chaining hash table.

Please note that, for method *rehash* to work, we need to add new data using method *add*. We cannot simply copy data into new lists because a value returned from hashing each data is different due to the array changing its size.

Method *remove* (shown in Figure 7-8) does the followings:

```
1:  public void remove(Object data){
2:      int pos = hash(data);
3:      CDLinkedList theList = lists[pos];
4:      if(theList.find(data) != -1){ //data found
5:          theList.remove(data);
6:          currentSize--;
7:      }
8:  }
9: } //end of class SepChaining
```

Figure 7-8: Method *remove* of separate chaining hash table.

- It first identifies a linked list that may store our given data, by calculating its slot in our array (line 2-3 in Figure 7-8).
- Once the list is identified, it searches the list for the data using method *find* of linked list (line 4 in Figure 7-8).
 - If the data is not in the list, we do nothing.
 - If the data is in the list, we call method *remove* of *CDLinkedList* (line 5 in Figure 7-8).

Runtime Analysis of Separate Chaining Hash Table

First, we need to define a term that is commonly used when analyzing hash tables. It is called a load factor (or λ – pronounced lambda). A load factor is defined as follows:

Definition 7-1:

$$\text{load factor} = \frac{\text{total number of data}}{\text{table size}}$$

For a separate chaining hash table, a load factor is therefore an average length of each linked list.

The time spent when searching a separate chaining hash table can be calculated from:

$$\begin{aligned} \text{Search time} &= \text{time to do hashing} + \text{time to search a list} \\ &= \text{very small value} + \text{time to search a list} \\ &\cong \text{time to search a list} \end{aligned}$$

If the data that we want to find is not in our hash table at all, our search time is approximately equal to the time to search an entire list. This is directly proportional to an average list length, which is our load factor.

If our data is in one of the lists, on average, we will have to search half of that list. Assume that our hash function can distribute data evenly, each linked list in our hash table should be of equal size. The average search time is therefore directly proportional to half an average list length, which is half our load factor.

In both cases, the time for searching a data depends on the value of the load factor.

Open Addressing Hash Table

An open addressing hash table does not use any linked list. Instead, if a collision is found, we find a new slot for our data. We need a considerably larger array (compared to a separate chaining approach) because there are no lists to help store data.

There is a pattern for finding slot(s).

- If the data is x , we first check array slot $h_0(x)$.
- If the data collides with another existing data, we try slot $h_1(x)$ in the array.
- If the data still collides, we try slot $h_2(x)$, etc.

Where $h_i(x) = (\text{hash}(x) + f(i)) \% \text{array size}$
 $f(0) = 0$

There are several ways to define $f(i)$. In this chapter, 3 of them are discussed. They are:

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

In this approach, we have:

$$f(i) = i$$

As an example, let us store integer data into an empty hash table of size 7. Let $\text{hash}(x) = x \% \text{array size}$.

The data to put in our example hash table are 1, 11, 3, 8, 9.

Putting in 1, 11, and 3 are straightforward, since there is no collision. The process of putting these values inside the array is shown in Figure 7-9.

Putting 8 in the array (using $h_0(8)$), however, causes a collision with 1. So, we need to inspect a slot next to the slot that stores 1, that is slot number $h_1(8)$. Fortunately, that slot is empty. Hence, we put 8 in that slot (see Figure 7-10).

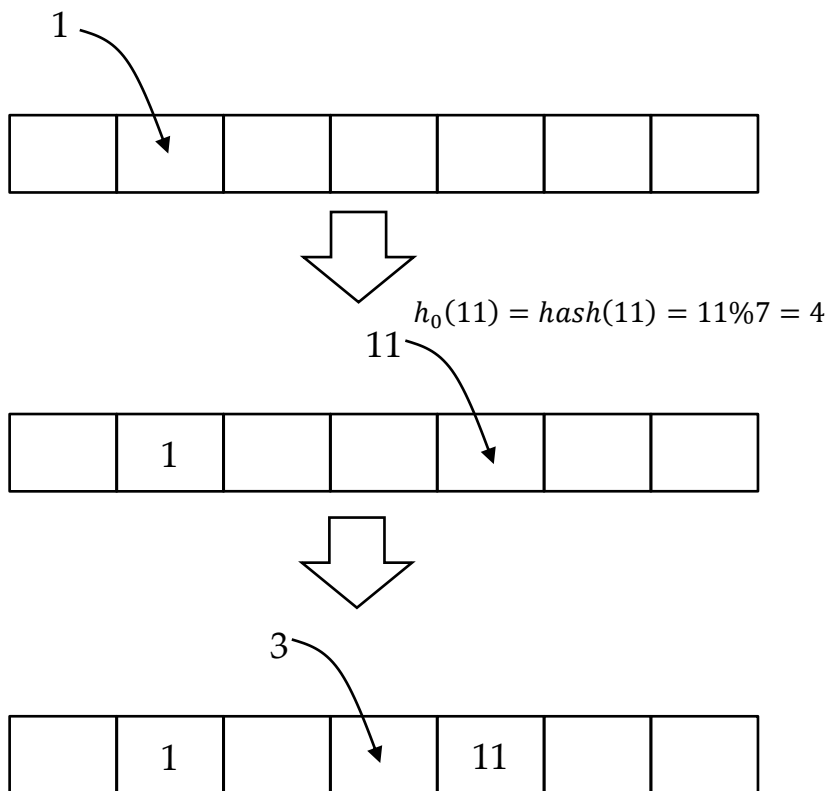


Figure 7-9: Putting 1, 11, and 3 into a hash table of size 7, where $\text{hash}(x) = x\% \text{array size}$.

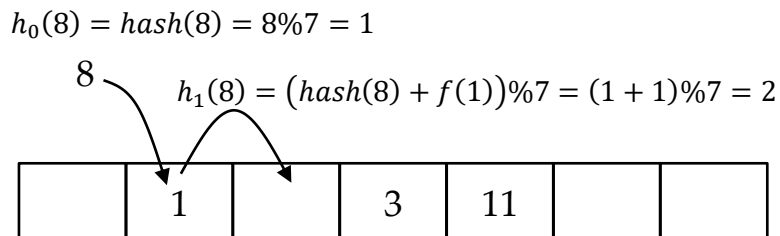


Figure 7-10: Putting 8 in a hash table from Figure 7-9.

Following 8, 9 is to be stored in the array. But using $h_0(9)$, 9 now collides with 8. So, what we do is look for the next slot, each one is at position $h_i(9)$, where i starts from 1, and keep looking until we find an empty slot.

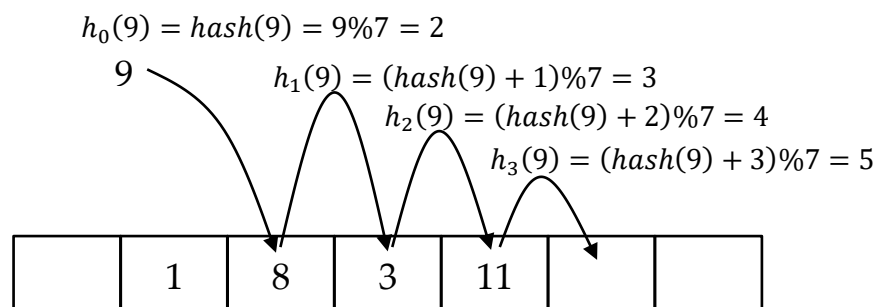


Figure 7-11: Putting 9 in a hash table from Figure 7-10.

Searching and removing data use this same procedure. To remove 9, we have to start from position $h_0(9)$, and work our way through each $h_i(9)$ until we find 9 or until we find an empty slot (which indicates that 9 is not in our hash table).

Removing data can cause a problem, however. Let us remove 3 and then try to search for 9, using a hash table in Figure 7-12.

It can be seen that after 3 is removed, the search for 9 will stop prematurely, at the position that used to store 3 (but now it is an empty slot). Our algorithm misinterprets that the array does not store 9 because it finds this empty slot during its search. **This premature stopping takes place in every open addressing approach discussed in this chapter.**

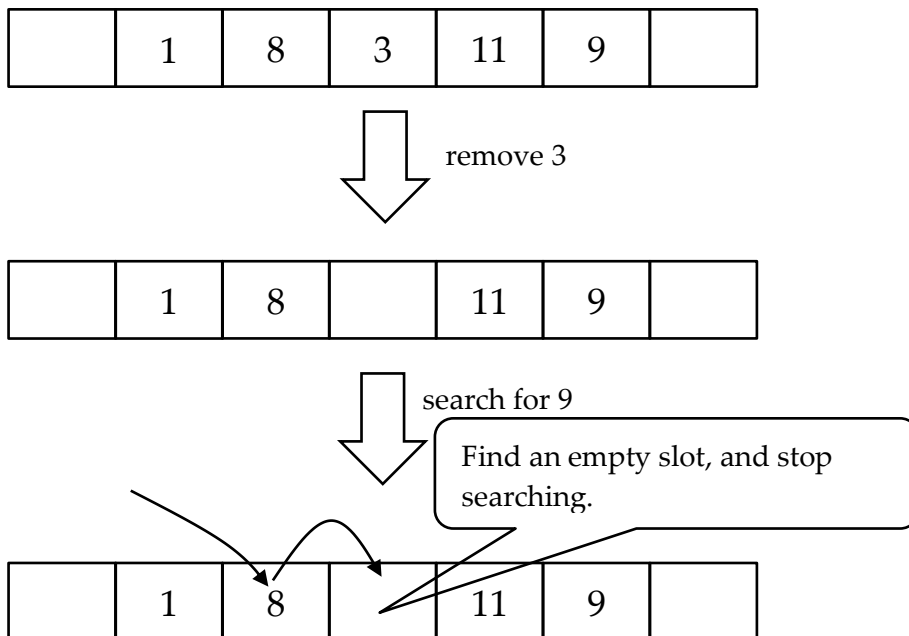


Figure 7-12: Removing 3 and then trying to search for 9, where $hash(x) = x \% array\ size$.

So, seeing an empty slot does not mean we should stop our search. When do we stop our search then? This can be handled by **lazy deletion**.

What is lazy deletion? Lazy deletion works as follows:

- When a data is to be deleted from an array slot, mark that array slot instead of deleting the data.
 - Marking can be done on the data, or a special kind of data can be inserted into that array slot to replace the original data.

Let us see lazy deletion in action in Figure 7-13, using the same array as in Figure 7-12. We try to do the same thing, that is, removing 3 and then searching for 9. This time, when a data is to be deleted, we replace it with a special data, DL.

It can be seen that DL prevents the search for 9 from stopping prematurely, since the search regards DL as a data.

Although using DL to replace the original data solves our problem, readers may wonder that we are wasting spaces that should be reclaimed for other data. But we cannot simply use this space right away when adding a new data. From Figure 7-13 (at the bottom), if we want to add 9, we will have to search beyond DL anyway because there is no guarantee that 9 is not stored anywhere further than DL in the array (and in this case

it was stored!). It is only when 9 is not found anywhere, that the first discovered DL slot can be used.

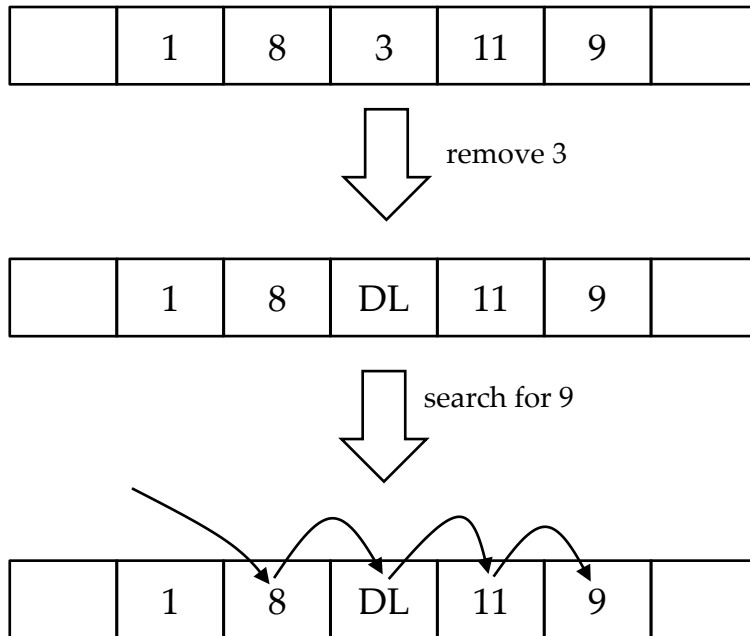


Figure 7-13: Lazy deletion prevents premature stopping while searching for data.

Apart from the deletion problem, linear probing has its own shortcoming. Readers may have already noticed from Figure 7-11 that when collisions take place near to one another, several consecutive array slots are very likely to be occupied. This can cause a problem when another collision takes place in one of these occupied slots. Because linear probing just searches one slot away each time, it can take a long time to find an empty array slot. This problem is called **primary clustering**.

Quadratic Probing

In this approach, we have:

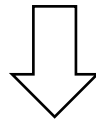
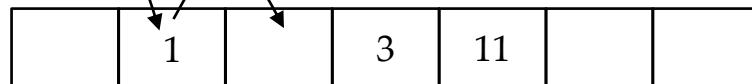
$$f(i) = i^2$$

An example is shown Figure 7-14. We store integer data into a hash table of size 7. Let $hash(x) = x \% array\ size$ and let the hash table originally store 1, 3, and 11. Then we try to add 8 and 9 to the hash table.

For each data, the more it collides, the more it is put further away. This prevents primary clustering.

$$h_0(8) = hash(8) = 8 \% 7 = 1$$

$$h_1(8) = (hash(8) + f(1)) \% 7 = (1 + 1^2) \% 7 = 2$$



$$h_0(9) = hash(9) = 9 \% 7 = 2$$

$$h_1(9) = (hash(9) + f(1)) \% 7 = (2 + 1^2) \% 7 = 3$$

$$h_2(9) = (hash(9) + f(2)) \% 7 = (2 + 2^2) \% 7 = 6$$

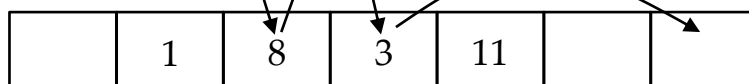


Figure 7-14: Adding 8 and 9 to a quadratic probing hash table that already has 1, 3, and 11, with $hash(x) = x \% array\ size$.

But a data that collides in the same slot as other several data before it will still have to go through the same calculation for each h_i and thus it can take some time to find the array slot that stores/will store the data. As an example, let us try to add 8, 15, 22 to a hash table with size 7, $hash(x) = x \% array\ size$. The table already has integer 1 stored inside. What happens is shown in Figure 7-15.

It can be seen that, to add 15, we need to repeat the operations done when adding 8. To Add 22, we need to repeat the operations done when adding 15. Thus, the more data that collide at the same slot, the longer it takes to search our hash table. This problem is called **secondary clustering**. It takes place because every data has the same calculation when avoiding the same array slot.

Another problem with quadratic probing hash table is that we may not be able to put our data into our array even though there are still some empty slots. Consider adding 29 to our array in Figure 7-15. You can see that its h_i always collide with existing data. This is even more likely to happen if the array size is not prime.

Therefore, a quadratic probing hash table needs to be larger than other types of hash tables in order to store the same amount of data.

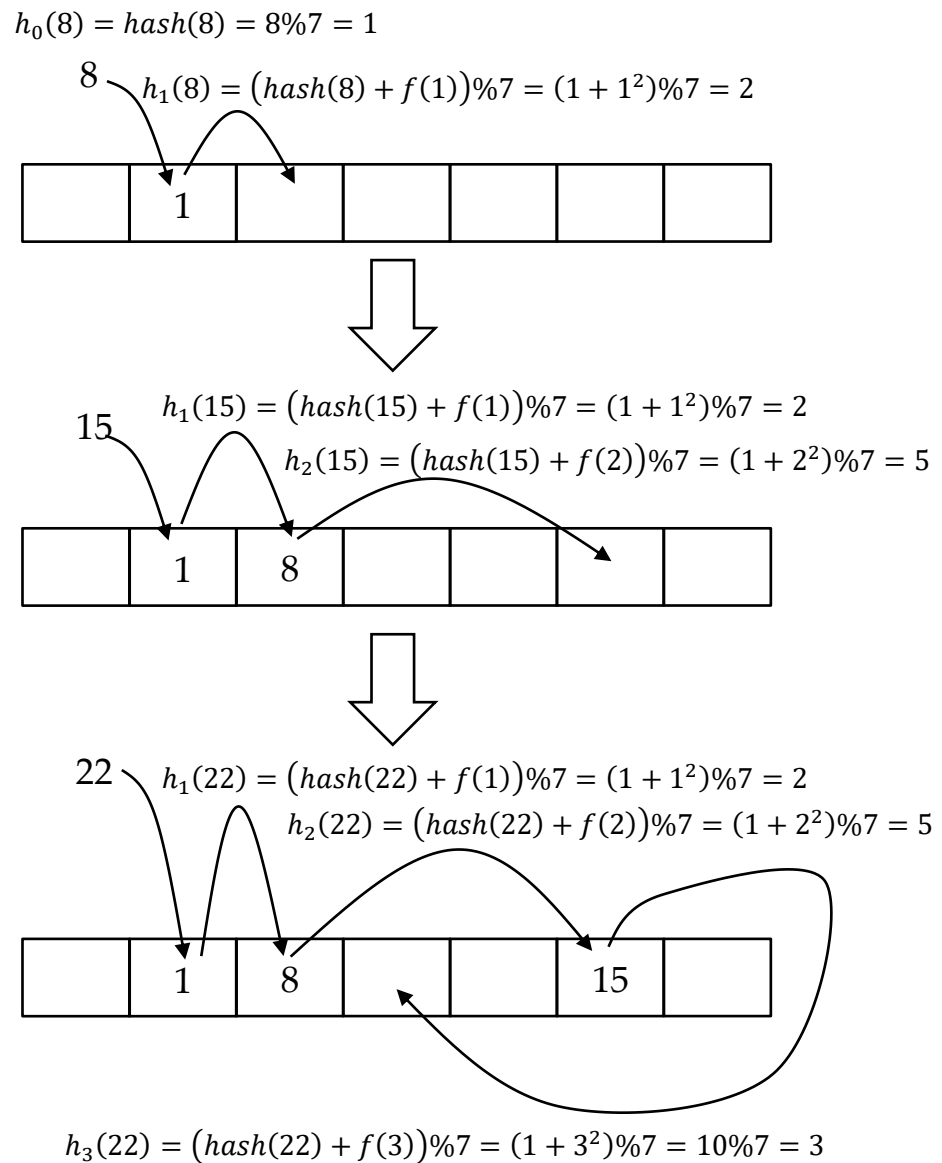


Figure 7-15: Adding 8, 15, 22 into a quadratic probing hash table that already has 1, with $\text{hash}(x) = x\% \text{array size}$.

How large do we need it to be? Here is a fact that can help us. It has been proven that if a quadratic probing hash table is not yet half full and the table size is prime, then we can always find a slot for a new data.

Let us look at the proof:

Let the array size be prime number larger than 3 and $0 \leq i, j \leq \lfloor \frac{\text{array size}}{2} \rfloor$ (this indicates that we still do not fill up to half of the array), where i and j are not equal (this represents different calculations).

Assume that position $(h(x) + i^2) \% \text{array size}$ collides with position $(h(x) + j^2) \% \text{array size}$.

Doing the math, we get:

$$\begin{aligned} h(x) + i^2 &= h(x) + j^2 && \% \text{ array size} \\ i^2 &= j^2 && \% \text{ array size} \\ (i - j)(i + j) &= 0 && \% \text{ array size} \end{aligned}$$

So, $((i - j)(i + j)) \% \text{array size} = 0$.

For the above statement to be true, one of the followings has to be true:

- $i - j = 0$
- $i + j = 0$
- $i - j = \text{array size}$

-
- $i + j = \text{array size}$
 - $((i - j)(i + j)) \% \text{array size} = 0$

The statement $i - j = 0$, or $i = j$ is impossible since we demand that i and j are from different calculations in the first place.

$i + j = 0$ is also impossible since they are both non-negative, and are not equal. So, they cannot both be 0.

$i - j = \text{array size}$ is impossible because both values are non-negative and the value of i is never larger than the array size.

$i + j = \text{array size}$ is also impossible because both values do not reach half of the array size at the same time.

$((i - j)(i + j)) \% \text{array size} = 0$ is impossible because the array size is prime.

So, our assumption that the two positions collide must be wrong! Therefore, they do not collide. Thus, if the array size is prime and not yet half full, we can find a position for our new data.

Double Hashing

This approach can avoid primary clustering and secondary clustering.

For this type of open addressing hash table:

$$f(i) = i * hash_2(x), \text{ where } x \text{ is our data.}$$

Using another hash function ($hash_2$) means that each data is likely to have a different pattern when avoiding collisions. Therefore, we can prevent both primary and secondary clusterings.

As an example, let us try to add 8, 15, 22 to a double hashing hash table with size 7, $hash(x) = x \% array \text{ size}$. Let $hash_2(x) = 3 - (x \% 3)$. The table already stores 1. This is shown in Figure 7-16.

From Figure 7-16, it can be seen that although 8, 15, and 22 collide with 1, all of them find different alternate positions. We thus avoid secondary clustering that was present in Figure 7-15.

Implementation of Open Addressing Hash Table

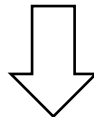
Fields, constructors, and utility methods of an open addressing hash table (the class name is *OpenAddressing*) is shown in Figure 7-17. Fields consist of:

- *DEFAULT_SIZE*: a default hash table size, which is prime.
- *DELETED*: a placeholder data to replace a deleted data. This is how we mark an array slot as deleted.

- *MAXFACTOR*: a default load factor that this table can tolerate. If a load factor exceeds this value, we need to do a rehash.
- *currentSize*: a number of data stored in our array.
- *array*: the array that stores our data.

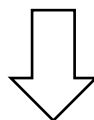
$$h_0(8) = \text{hash}(8) = 8\%7 = 1$$

$$h_1(8) = (\text{hash}(8) + f(1))\%7 = (1 + 1 * (3 - 8\%3))\%7 = 2$$



$$h_1(15) = (\text{hash}(15) + f(1))\%7 = (1 + 1 * (3 - 15\%3))\%7 = 4$$

15



$$h_1(22) = (\text{hash}(22) + f(1))\%7 = (1 + 1 * (3 - 22\%3))\%7 = 3$$

22

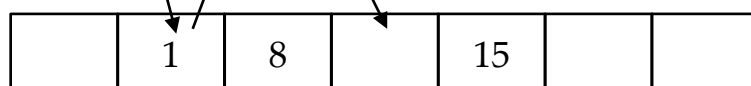


Figure 7-16: Adding 8, 15, 22 into a double hashing hash table that already has 1, with $\text{hash}(x) = x\% \text{array size}$ and $\text{hash}_2(x) = 3 - (x\%3)$.

```
1: public class OpenAddressing{
2:     private static int DEFAULT_SIZE = 101;
3:     Private static final Object DELETED = new
4:                                     Object();
5:     private static int MAXFACTOR = 0.5;
6:     private int currentSize =0;
7:     private Object[] array;
8:
9:     public OpenAddressing(){
10:         this(DEFAULT_SIZE);
11:     }
12:
13:     public OpenAddressing(int size){
14:         int nextPrimeSize = nextPrime(size);
15:         array = new Object[nextPrimeSize];
16:     }
17:
18:     private static boolean isPrime(int n){
19:         //Same code as in our
20:         //separate chaining hash table.
21:         // You can rewrite method isPrime and nextPrime
22:         // in a separate utility class.
23:         . . .
24:     }
25:
26:     private static int nextPrime(int n){
27:         // Same code as in our separate chaining hash
28:         // table.
29:         . . .
30:     }
31: } // end of class OpenAddressing.
```

Figure 7-17: Fields, constructors, and utility methods for open addressing hash table.

Constructors just create our array, making its size prime. Method *isPrime* and *nextPrime* are exactly the same as in Figure 7-4.

A double hashing hash table (shown from Figure 7-18 to Figure 7-21) is defined by extending from

OpenAddressing. We do not show the implementation of linear probing and quadratic probing in this book. Their codes are similar to a double hashing hash table except their function h_i .

Shown in Figure 7-18, our double hashing hash table (class `DoubleHashing`) has its own `MAXFACTOR` so that we can use our own default value. Its *hash* uses *hashCode*. *hash2* can use any function that produces unique number, but it must be different to *hash* because we do not want data that collide in the same array slot to use the same collision avoidance pattern.

It also has a new field, *occupiedSlots*. This field records the number of slots that store data or `DELETED` object. We use this field to determine whether to rehash (see method *add* in Figure 7-19).

Method *find* calculates $h_i(\text{data})$ until it finds our data, or finds an empty slot, or tries enough number of times. The calculation of each h_i , starting from $i = 0$, is carried out by a for loop (line 26-30 in Figure 7-18). The number of iterations is enough for us to look at every array slot because the array size is prime.

If you want to implement a linear probing or a quadratic probing hash table, simply change the code on line 30 in Figure 7-18 according to each type of hash table. Method *find* returns the position of our data (or the position of the

empty slot) if the data (or the empty slot) is found. Otherwise, it returns -1, which means we cannot find our data and there is no empty array slot.

```
1: class DoubleHashing extends OpenAddressing
2:     private static int MAXFACTOR = 0.75;
3:     private int occupiedSlots = 0;
4:
5:     public DoubleHashing(){
6:         this(DEFAULT_SIZE);
7:     }
8:
9:     public DoubleHashing(int size){
10:        super(size);
11:    }
12:
13:    public int hash(Object data){
14:        int hashValue = data.hashCode();
15:        int abs = Math.abs(hashValue);
16:        return abs%array.length;
17:    }
18:
19:    public int hash2(Object data){
20:        return //any unique number function different
21:            //from hash.
22:    }
23:
24:    public int find(Object data){
25:        int h = hash(data);
26:        int hash2Result = hash2(data);
27:        for(int i=0; i<currentSize; i++){
28:            if(array[h] == null || array[h].equals(data))
29:                return h;
30:            h = (h + hash2Result)%array.length;
31:        }
32:        return -1;
33:    }
34:
35:    //continued in Figure 7-19.
```

Figure 7-18: Fields, constructors, hash functions, and method *find* of a double hashing implementation.

For your information, the code on line 30 should be changed to $h = (h + 1) \% \text{array.length}$ for linear probing, and $h = (h + 2 * i - 1) \% \text{array.length}$ for quadratic probing.

Figure 7-19 shows method *add*. The first part of method *add* (line 6-14) is almost the same as method *find*, that is, we attempt to find our data through the calculation of h_i . The only major difference in this part is that in method *add*, if our search encounters a slot that is marked *DELETED*, we record this slot position. This slot position will be the position we add our new data. Reusing a *DELETED* slot helps save space.

After the attempt to find our data, if the data or an empty slot is not found, it means that our data is not in the array and the array is full somehow. So, we have to rehash and then attempt to add the data again (line 15-17).

Otherwise, if the data is found, we do nothing since there is no point adding a duplicated data.

But if the empty slot is found, we add the data to the array (add to the *DELETED* slot if its position was recorded earlier) (see line 20-25). Then we update *currentSize* and call method *rehash* if the current load factor exceeds our specified value (line 26-30).

```
1: public void add(Object data) throws Exception{
2:     int h = hash(data);
3:     int hash2Result = hash2(data);
4:     int emptySlotPosition = -1;
5:     int i;
6:     for(i=0; i<currentSize; i++){
7:         if(array[h] == null || array[h].equals(data))
8:             break;
9:         if(array[h] == DELETED &&
10:            emptySlotPosition == -1){
11:             emptySlotPosition = h;
12:         }
13:         h = (h + hash2Result)%array.length;
14:     }
15:     if(i >= currentSize){
16:         rehash();
17:         add(data);
18:     } else {
19:         if(array[h] == null){
20:             if(emptySlotPosition != -1){
21:                 array[emptySlotPosition] = data;
22:             } else{
23:                 array[h] = data;
24:                 occupiedSlots++;
25:             }
26:             currentSize++;
27:             double currentLoadFactor =
28:                 (double)(occupiedSlots/array.length);
29:             if(currentLoadFactor > MAXFACTOR)
30:                 rehash();
31:         }
32:     }
33: }
34: //continued in Figure 7-20.
```

Figure 7-19: Method *add* of a double hashing implementation.

Method *rehash* (see Figure 7-20) makes a new array that is larger, then adds all data (*DELETED* objects are not true data so they are not added) from our original array into the new array. The new additions have to be done using method *add* so that the correct position for each data can be determined by our hash function.

```
1:  public void rehash(){
2:      Object[] oldArray = array;
3:      array = new Object[nextPrime(array.length*2)];
4:      currentSize = 0;
5:      occupiedSlots = 0;
6:      for(int i=0; i<oldArray.length; i++){
7:          if(oldArray[i] != null &&
8:             oldArray[i]!=DELETED)
9:              add(oldArray[i]);
10:     }
11: }
12: //continued in Figure 7-21.
```

Figure 7-20: Method *rehash* of a double hashing implementation.

Method *remove* (see Figure 7-21) first attempts to find our data. If the data is found, we replace it with a *DELETED* object.

```
1:  public void remove(Object data){
2:      int index = find(data);
3:      if(index != -1 && array[index]!=null){
4:          array[index] = DELETED;
5:          currentSize--;
6:      }
7:  }
8: } // end of class DoubleHashing.
```

Figure 7-21: Method *remove* of a double hashing hash table.

Separate Chaining VS Open Addressing

Table 7-1 shows advantages and disadvantages from using both types of hash tables. It is up to readers to choose the type they believe to be suitable for their works.

Table 7-1: Separate Chaining and Open Addressing Comparison.

Separate Chaining	Open Addressing
Simple add and remove.	More complicated add and remove.
A lot of space required to store references.	Less space required, even though the load factor is small.
Collisions only affect data in the same linked list.	Collisions may affect an entire table.

Exercises

1. A hash table of size 11 with $hash(x) = x \% array.length$ uses double hashing. Index of a data when collision takes place is $hash(x) + f(i)$, where $f(i) = i * hash_2(x)$. The value of i starts at 1. Let $hash_2(x) = 5 - x \% 5$.

We use lazy deletion. A new data **does not** overwrite old data even though the old data may already be marked as *DELETED*.

The array currently has only one data:

				4						
--	--	--	--	---	--	--	--	--	--	--

What will happen if we sequentially do the following actions:

- Add 15
- Add 26
- Add 8
- Delete 26
- Add 19

Explain what happens in each step. Draw picture for each step too.

2. Given a double hashing hash table for integer data of size 13. Let $hash(x) = x \% array.length$ and $hash_2(x) = 7 - x \% 7$. **Show and explain** (step by step) what happens when 1, 5, 18, 8 are inserted into the hash table in order, using double hashing.
3. In open addressing hash table, explain the reason for the use of lazy deletion. Give example(s) too.
4. A hash table of size 13 with $hash(x) = x \% array.length$ uses quadratic probing with lazy deletion, where DELETED slot can be reused.

We sequentially do the following actions:

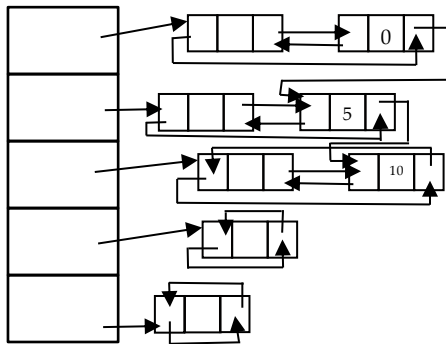
- Add 5
- Add 10
- Add 18
- Add 19
- Add 31
- Delete 18
- Add 32

Draw what happens to the hash table in each step.

5. A special kind of separate chaining hash table works as follows:
 - When a new data is added, if a linked list obtained using our hash function contains at least one data, but it does not store the new data,

we find a new empty position in our array (using linear probing) and add that new data in a node linked to that array slot. Then the linked list obtained from our hash function is linked to this newly created node.

For example, let us add 0, 5, 10 (all these collide in the first array slot) to this hash table respectively (size 5, $hash(x) = x \% array.length$). Below is our hash table after adding the three numbers. It can be seen that it is quite different from a normal separate chaining hash table. Our link on a linked list actually links to other linked lists.



Explain how to delete data from this kind of hash table. Write your code for method *find*, *add*, and *remove* using this table.

6. For a separate chaining hash table, what if we use ArrayLists instead of doubly-linked lists in our implementation? Write code for method *find*, *add*, and *remove* for this new implementation.

Chapter 8 : Sorting

In this chapter, we will be looking at various algorithms that can sort data (from small to large) stored in an array.

Bubble Sort

This is done by:

1. Comparing 2 adjacent values in an array, starting with the first two values. Swap the values if the value on the left is larger than the value on the right.
2. Then compare the next pair of values. The leftmost value maybe the one swapped from the previous comparison.
3. Once the last two values in the array are compared and/or swapped, start again with data in the first and second slot. Repeat.

An example is shown in Figure 8-1. But how many times do we need these swaps? Let us think of it this way:

- The number of swaps must be enough for moving the largest value from the leftmost array slot to the rightmost array slot.
- The number of swaps must also be enough for moving the smallest value from the rightmost array slot to the leftmost array slot.

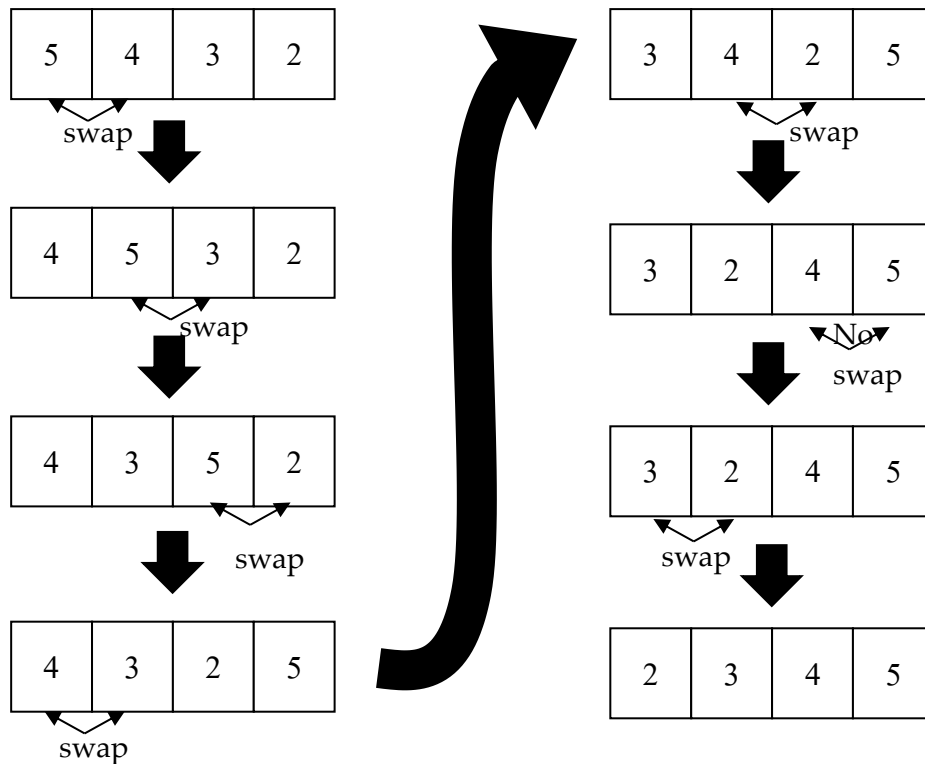


Figure 8-1: Bubble Sort.

When the array size is n . The first $n-1$ swaps are enough to move a value from the leftmost position to the rightmost position. But the value in the rightmost position can only move 1 position to the left. This is clearly shown in the left half of Figure 8-1, where the first $n-1$ swaps (n is 4) can move 5 from the leftmost array slot to the rightmost array slot, but 2 (originally at the rightmost slot) can only move one slot to the left.

In order to move the rightmost data all the way to the left of the array, we need to repeat the moves in the above

paragraph $n-1$ times. Therefore, the number of comparisons (and possibly, swaps) that we need is $(n - 1) \times (n - 1)$. Hence the estimated runtime of bubble sort is $\theta(n^2)$. Our code for bubble sort is shown in Figure 8-2. Please note that we write all sorting methods as static methods so they can be implemented in their own java classes.

```
1: public static void swap(int[] array, int a, int b){
2:     int temp = array[a];
3:     array[a]= array[b];
4:     array[b]= temp;
5: }
6:
7: public static void bubblesort(int[] array){
8:     for (int p = 1; p <=array.length-1; p++)
9:         for(int e =0; e <= array.length -2; e++)
10:            if(array[e] > array[e+1])
11:                swap(array, e, e +1);
12: }
```

Figure 8-2: Code for bubble sort algorithm.

The nested loop structure also confirms that $\theta(n^2)$ is our asymptotic runtime. The worst-case (requires the most number of swaps) running time for bubble sort takes place when the array is sorted from large to small. Figure 8-1 is also an example of the worst-case scenario.

Selection Sort

A selection sort algorithm can be described as follows:

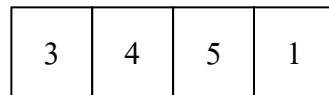
1. Store the index of the first array element in variable *maxindex*.

2. Check each array member one by one. If a member value is greater than $array[maxindex]$, change $maxindex$ to store the index of that member. Continue until all members (not including a sorted portion of the array) are checked.
3. Swap the last data that has not been sorted with $array[maxindex]$ (no swapping needed if both are the same member).
4. Then, consider all other data not yet in their correct position, repeat the above 3 steps. This means we gradually build up a sorted portion of our array from right to left. We repeat until all data are in the sorted array portion.

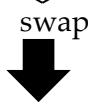
The algorithm is illustrated in Figure 8-3. In each step, $maxindex$ is updated to the position of the maximum unsorted value. Then that value (identified by $maxindex$), is swapped with the rightmost value in the unsorted portion of the array.

We only need to do a value swap when we find the largest unsorted value. Hence, selection sort is expected to be faster than bubble sort.

maxindex = 0, then updated to 2.



maxindex reset to 0, Sorted
then updated to 1. portion



maxindex reset
to 0, then Sorted
updated to 0. portion

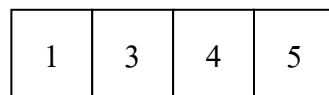
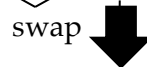
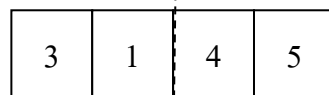


Figure 8-3: Selection sort.

The code for selection sort is given in Figure 8-4. The code has nested loops, each loop is estimated to run for at most about n times, where n is the array size. Therefore, the asymptotic runtime of selection sort is $\Theta(n^2)$. This is the same asymptotic runtime as bubble sort, even though selection sort is supposed to be faster.

The inner loop (line 6-8 in Figure 8-4) looks through every data in the unsorted portion of the array and stores the position of the maximum value amongst them.

The outer loop just swaps the maximum value with the rightmost unsorted value (line 11 in Figure 8-4). It also resets the value of *maxindex* before entering the inner loop.

```
1: public static void selectionSort(int[] a){
2:     int maxindex; //index of the largest value
3:     int unsorted;
4:     for(unsorted=a.length; unsorted > 1; unsorted--){
5:         maxindex = 0;
6:         for(int index= 1; index < unsorted; index++){
7:             if(array[maxindex] < array[index])
8:                 maxindex = index;
9:         }
10:        if(a[maxindex] != a[unsorted -1])
11:            swap(array, maxindex, unsorted -1);
12:    }
13: }
```

Figure 8-4: Code for selection sort.

The worst-case running time (maximum number of comparisons and maximum number of swaps) takes place when there is a swap for every round of comparisons, which happens when the data are almost sorted but the smallest value is the last data, such as 2,3,4,5,1.

Insertion Sort

This algorithm is as follows:

1. Split the array into 2 sides, left and right. The left side is considered sorted. Therefore, at the beginning, there is only one member in the left side.
2. Check the leftmost value on the right side. Store it in *temp* variable.
3. If the value of *temp* is smaller than the last member of the left side, put the value of *temp* in its correct place on the left side. To put the value of *temp* in its correct position on the left side, we need to keep moving other values to the right so that we have space to put our *temp* value.
4. Repeat the whole steps again. Each time, the left side (sorted side) will grow by 1. Repeat until all members are moved to the left side.

This algorithm is illustrated in Figure 8-5. The source code is shown in Figure 8-6. The inner loop (line 5-11) shifts data in the sorted array portion so that a correct position for the first data from an unsorted portion is ready. The outer loop makes sure that every data is examined and put in a prepared position.

The asymptotic runtime, from the code, is $\theta(n^2)$. The worst-case runtime takes place when there are maximum number of data shifting. That means, for each inner loop, all data must move. This happens when the array is initially sorted from large to small.

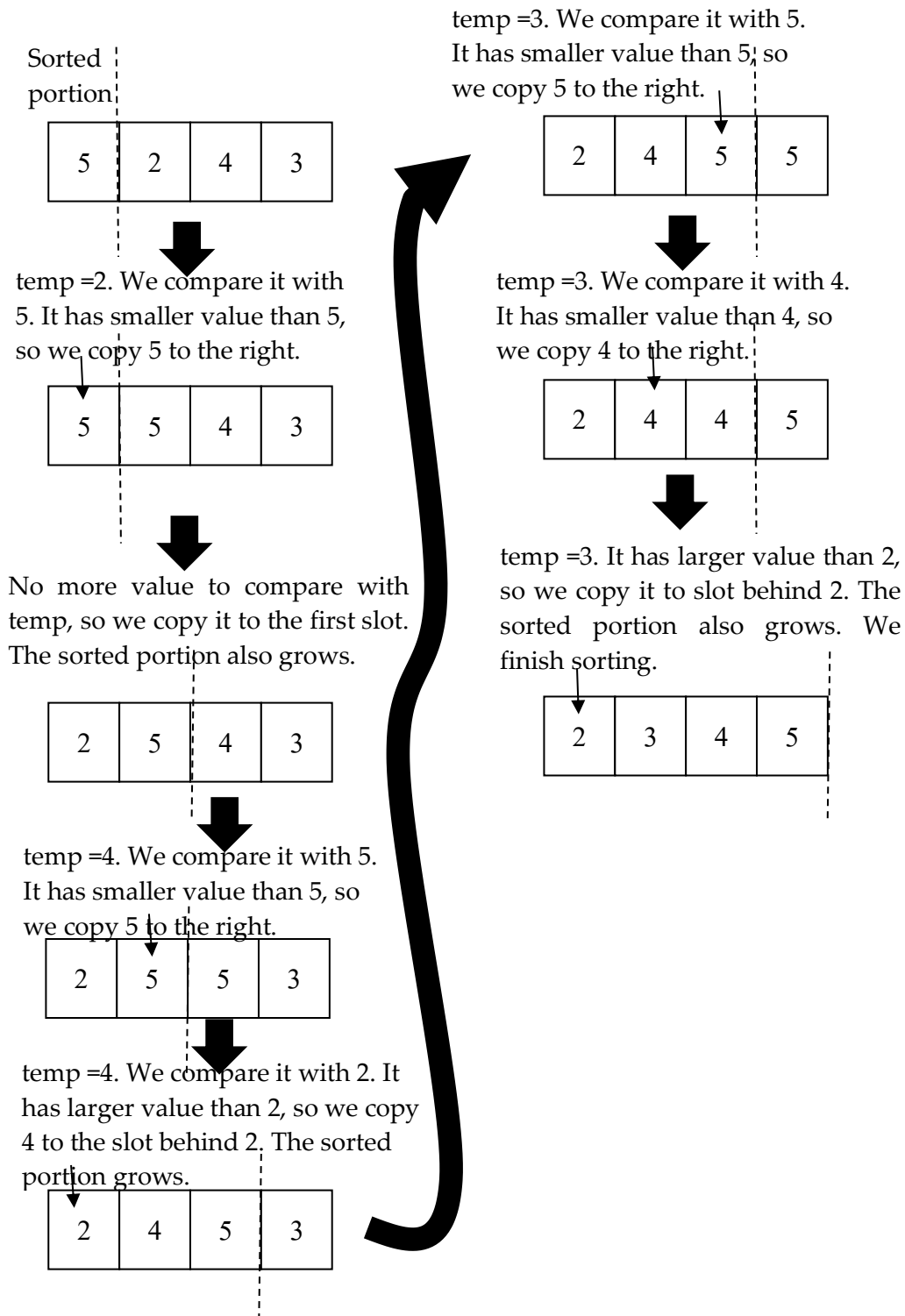


Figure 8-5: Insertion sort.

```
1: public static void insertionSort(int[] a){
2:     int index;
3:     for(int sorted = 1; sorted < a.length; sorted++){
4:         int temp = a[sorted];
5:         for(index = sorted; index >0; index--){
6:             if(temp< a[index-1]){
7:                 a[index] = a[index -1];
8:             } else{
9:                 break;
10:            }
11:        }
12:        a[index] = temp;
13:    }
14: }
```

Figure 8-6: Code for insertion sort.

Merge Sort

So far, we have looked at sorting algorithms with performance $\theta(n^2)$. Merge sort is an algorithm that has better performance. Here is how the algorithm works:

1. Split a part of the array that we want to sort into two portions (with equal size).
2. Sort each portion with merge sort.
 - a. Each portion can be further divided. Hence, we have a recursion here.
3. Then combine the sorted portions.

The above concept is illustrated in Figure 8-7.

Array Splitting

In this part of the algorithm, we divide the data that we want to sort into two equal halves. We do not really need to create 2 new arrays. What we need to do is just

identifying array slots that are in the first half and the second half.

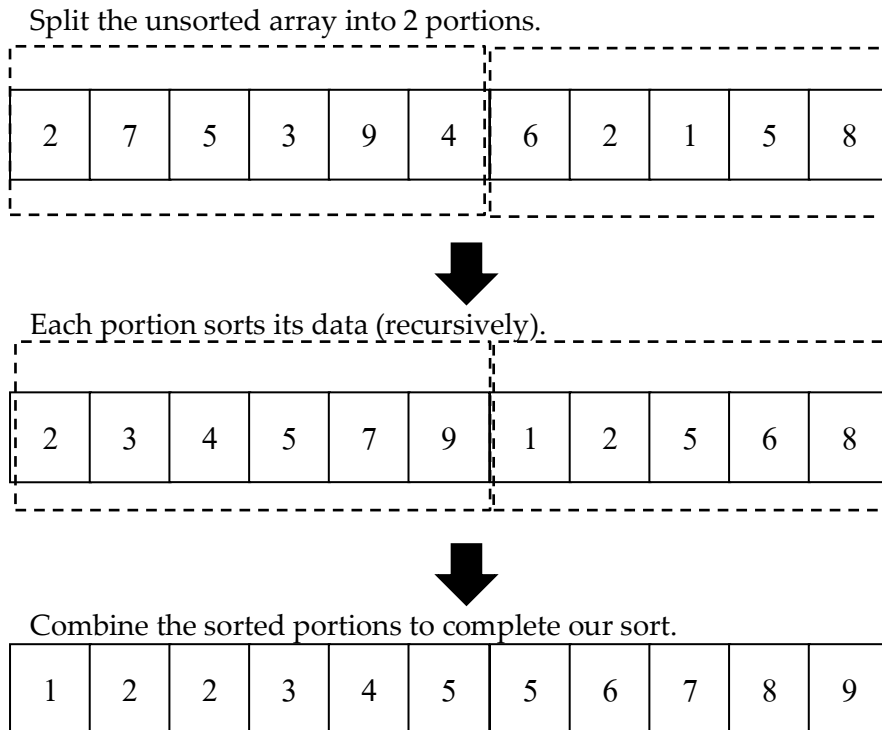


Figure 8-7: Merge sort concept.

For ease of implementation, we define the following variables:

- *left*: an integer indicating the leftmost position of the array portion that we want to sort.
- *right*: an integer indicating the rightmost position of the array portion that we want to sort.
- *center*: an integer indicating the position halfway between left and right.

The values of *left* and *right* can identify any portion of the array that we want to sort. We need to be able to identify any portion because we will be dividing the array recursively.

We only apply our algorithm to data stored between *left* and *right*. Figure 8-8 shows an array where we want to sort data from slot number 2 to slot number 7. Therefore $left = 2$, $right = 7$, $center = \frac{left+right}{2} = \frac{2+7}{2} = 4$. Consequently, the first array portion is from slot 2 to slot 4. The second array portion is from slot 5 to slot 7. This is how we identify the two portions.

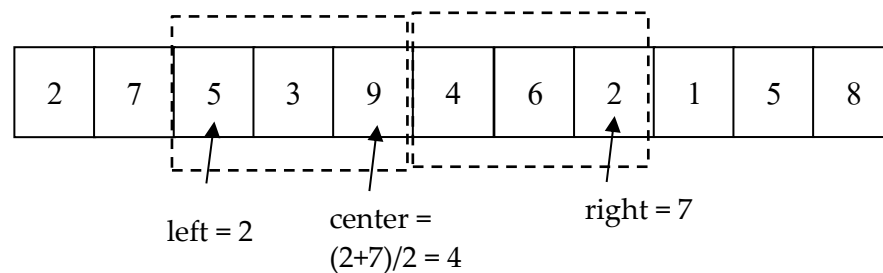


Figure 8-8: Splitting array into 2 portions for sorting.

Sorting Each Portion

We need to call merge sort recursively. Let our method *mergeSort* return a new array that contains a sorted sequence from index *left* to *right* of its input array.

The method looks like:

```
int[] mergeSort(int[] unsort, int left, int right)
```

Method *mergeSort* receives 3 parameters:

- *unsort*: our array to be sorted.
- *left*: the leftmost position of the data we want to sort.
- *right*: the rightmost position of the data we want to sort.

Using the array splitting mechanism mentioned previously, the first half of the array can be sorted by using :

- `int[] result1 = mergeSort(unsort, left, center);`

Similarly, the second half of the array can be sorted by using :

- `int[] result2 = mergeSort(unsort, center+1, right);`

Generally, *result1* and *result2* will be just about half the size of *unsort*.

If *left* == *right*, it means the part of the array we want to sort only contains 1 data. In such case, we do not need to call *mergeSort* recursively. We can simply return a one-slot array with that very data stored inside as our sorting result.

Merging Two Sorted Portions

After we get two sorted arrays (*result1* and *result2* from the previous section), we combine them into one sorted array with following algorithm:

1. Create integer i and j to mark the first slot of $result1$ and $result2$ respectively.
2. Create a result array (let us name it $answer$) that can hold all data from both $result1$ and $result2$. Also, create integer k to mark the first slot of $answer$.
3. Repeat until we look at all data in either $result1$ or $result2$:
 - a. Compare $result1[i]$ and $result2[j]$.
 - b. Copy the smaller value to $answer[k]$, then increment k and the index of the array that stores the value. For example, if $result1[i] < result2[j]$, then $answer[k++] = result1[i++]$. If compared values are equal, you can do the copying from either $result1$ or $result2$.
4. Copy all remaining contents from the array that we have not finished copying into the remaining slots of $answer$.

Figure 8-9 shows what happen when we combine $\{1,5,8,9\}$ with $\{2,4,6,7\}$.

Implementation and Runtime of Merge Sort

Code for merge sort is shown in Figure 8-10. Code for combining 2 sorted arrays is shown in Figure 8-11. To work out the asymptotic runtime of merge sort, we need to know the time used for combining 2 arrays first. From the code in Figure 8-11, the runtime for array combination is $\theta(n)$.

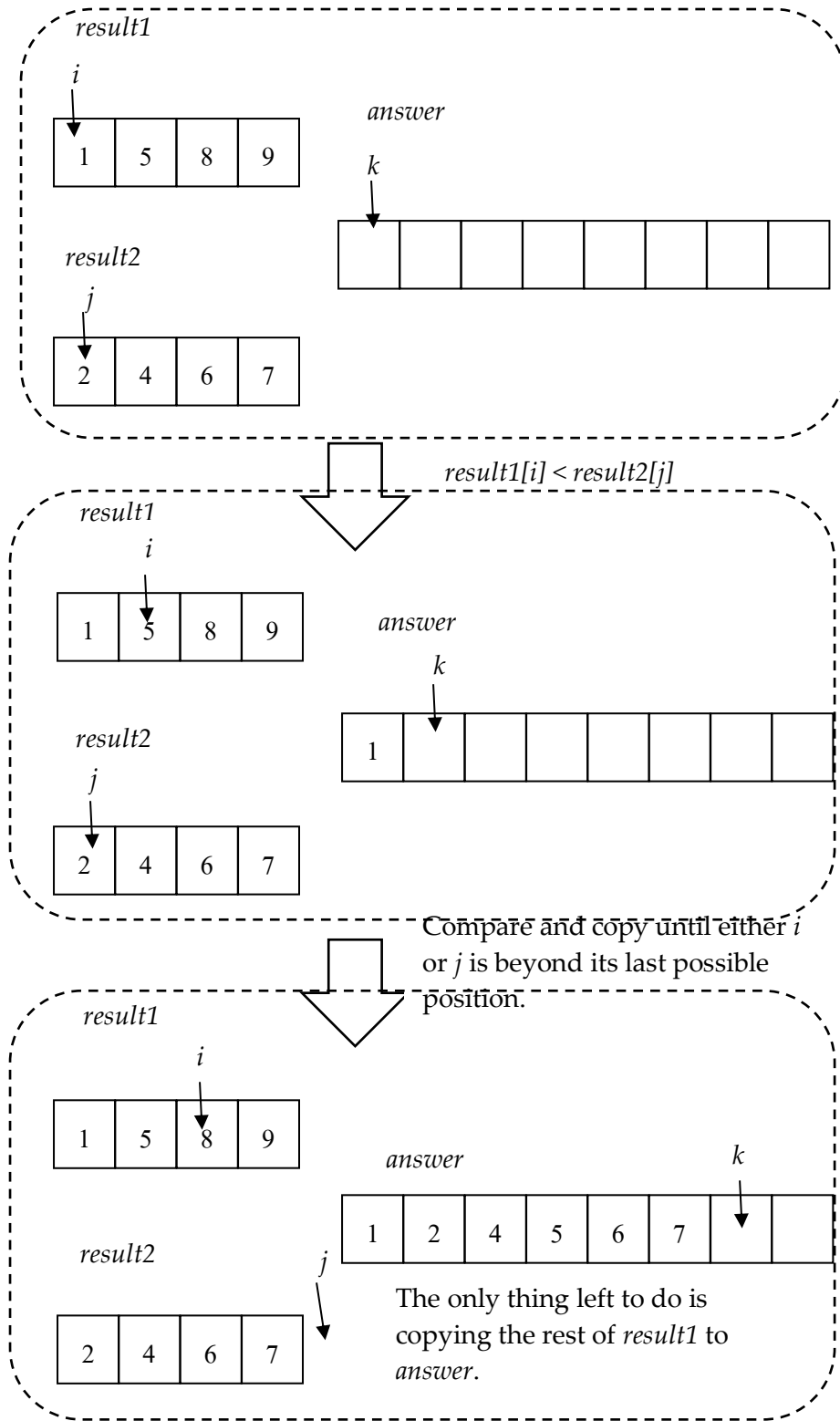


Figure 8-9: Combining 2 sorted arrays.

```
1: public static int[] mergeSort(int[] a, int
2: left, int right){
3:     if(left == right){//only 1 data to sort
4:         int[] x = new int[1];
5:         x[0] = a[left];
6:         return x;
7:     }
8:     else if(left<right){
9:         int center = (left+right)/2;
10:        int[] result1 = mergeSort(a,left,center);
11:        int[] result2 = mergeSort(a,center+1,right);
12:        return merge(result1,result2); //combine arrays
13:    }
14: }
```

Figure 8-10: Code for merge sort.

```
1: public static int[] merge(int[] a, int[] b){
2:     int aIndex = 0; int bIndex = 0; int cIndex = 0;
3:     int aLength = a.length;
4:     int bLength = b.length;
5:     int cLength = aLength + bLength;
6:     int[] c = new int[cLength];
7:
8:     // compare a and b then move a smaller value
9:     // into c until one array is spent.
10:    while((aIndex < aLength) && (bIndex < bLength)){
11:        if(a[aIndex]<=b[bIndex]){
12:            c[cIndex++] = a[aIndex++];
13:        }else{
14:            c[cIndex++] = b[bIndex++];
15:        }
16:    }
17:
18:    //copy the remaining data into c
19:    if(aIndex == aLength){ //if a is spent.
20:        while(bIndex<bLength){
21:            c[cIndex++] = b[bIndex++];
22:        }
23:    }else{ //if b is spent.
24:        while(aIndex<aLength){
25:            c[cIndex++] = a[aIndex++];
26:        }
27:    }
28:    return c;
29: }
```

Figure 8-11: Code for combining 2 sorted arrays into one.

Indeed, what the program in Figure 8-11 does is visiting each data once. Therefore, if the resulting array stores n data, the program simply traverses those data.

Now, let us try to estimate the runtime for merge sort. Let the time for executing method *mergesort* from Figure 8-10 be $T(n)$.

If there is only 1 data, our method (line 3-7 from Figure 8-10) runs in constant time. Therefore:

$$T(1) = \theta(1)$$

If there are many data, the time used is the sum of the time to sort the left portion, the time to sort the right portion, and the time to combine the sorted portions. For simplicity, we assume that the left and the right portion has equal number of data. Hence:

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + \theta(n)$$

We can divide both sides of the above equation by n . So, we get:

$$\frac{T(n)}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1$$

If we keep changing n to $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots$ we get the following sets of equations:

$$\frac{T(n)}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1$$

$$\frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} = \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + 1$$

$$\frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} = \frac{T\left(\frac{n}{8}\right)}{\frac{n}{8}} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

From observation, it can be seen that:

$$2^{\text{number of equations}} = n$$

Therefore:

$$\text{number of equations} = \log_2 n$$

If we add all of the above equations, most of the contents will cancel out, and we will get:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log_2 n$$

$$T(n) = n \times \theta(1) + n \log_2 n$$

Hence, $T(n) = \theta(n \log_2 n)$, which is faster than $\theta(n^2)$.

Quick Sort

Although merge sort has better performance than all the previous sorting algorithms, the algorithm needs to create a new array to store the sorted result. Quick sort uses similar principle, but it operates on the input array. Therefore, required space is reduced.

The quick sort algorithm (see Figure 8-12) is as follows:

1. If the input array has one data or less, that input array is our answer (we do not need to move any value inside the input array). The algorithm ends here in such case.
2. For a small input array (e.g. 20 data or less), other sorting algorithms are faster because the advantage of splitting the input array into 2 portions does not outweigh the overhead for partitioning. Hence, we just call other sorting methods such as insertion sort. The algorithm then ends here.

3. Choose a value in the input array. That value becomes our “pivot”.
4. “Partitioning” the input array by the following steps:
 - a. Moving all values that are less than our pivot to the left of the pivot.
 - b. Similarly, all values greater than the pivot should be moved to the right of the pivot.
 - c. For values equal to the pivot, distribute them evenly on both sides of the pivot.
5. At this stage, the pivot is in a correct position. We then call quick sort on parts of the array on both sides of the pivot.

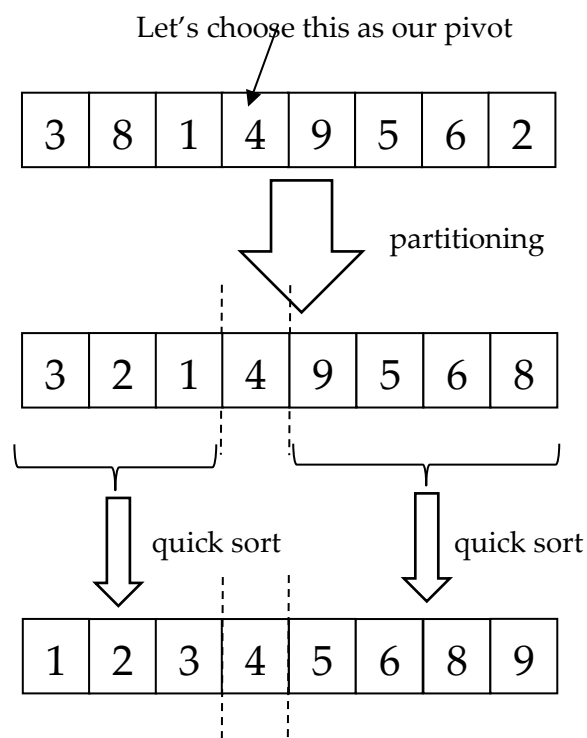


Figure 8-12: Quick sort concept.

Choosing a Pivot

An ideal pivot should be the median value amongst all values stored in the input array. This is so that partitioning can divide the array into 2 equal halves (the smaller the array can be reduced to, the faster quick sort recursive calls can finish).

But it is impractical to find a median because you will have to investigate all the stored values. You cannot simply choose the first or the last value as your pivot either because if the array (or its portion that you are working on) is already sorted, partitioning will just reduce the array size by one (because one of the two portions will be empty). This bad pivot selection is shown in Figure 8-13.

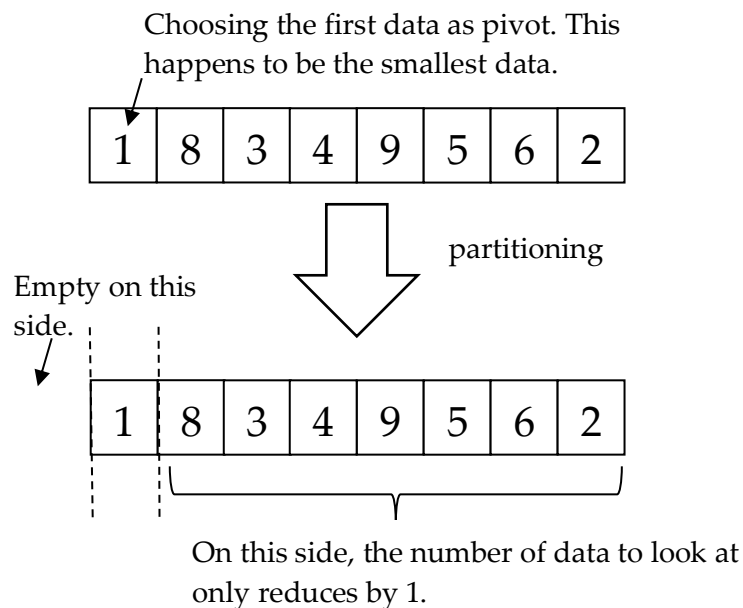


Figure 8-13: Bad pivot selection.

So how do we select a good pivot? Since we do not know how the data is arranged, choosing a pivot randomly is likely to get us even partitions. However, a random number is slow to generate.

To get a fast selection and a number that can generally divide the array in half, we will use a method that statistically works in most cases. The method is called “median of 3”. Basically, we find the median amongst the first data, the last data, and the middle data from our input array. The code for “median of 3” is shown in Figure 8-14 as method *pivotIndex*, which returns the position of our chosen pivot.

```
1:  static int pivotIndex(int[] a, int l, int r){
2:      int c = (l+r)/2;
3:      if((a[l]<=a[r] && a[l]>=a[c]) ||
4:         (a[l]>=a[r] && a[l]<=a[c]))
5:          return l;
6:      if((a[c]<=a[l] && a[c]>=a[r]) ||
7:         (a[c]>=a[l] && a[c]<=a[r]))
8:          return c;
9:      return r;
10: }
```

Figure 8-14: Code for median of 3.

Partitioning

Once a pivot is selected, we can partition our array (let us name our array “*a*”) using the following algorithm:

1. Get the pivot out of the way by swapping it with the last data.

2. Let i be the index of the first position and j be the index of the before-last position (the pivot is in the last position).
3. Keep incrementing i until $a[i] \geq \text{pivot value}$.
4. Keep decrementing j until $a[j] \leq \text{pivot value}$.
5. If i is on the left of j , swap $a[i]$ and $a[j]$. This is an attempt to move smaller value to the left and larger value to the right of the array. If i is not on the left of j , go to step 8.
6. Increment i by 1 and decrement j by 1. This is just avoiding the slots that we just swap their values.
7. Go to step 3.
8. Swap $a[i]$ with pivot. We will get the array with the pivot in its correct position. To the pivot's left will be the smaller values and to its right will be the larger values.

An example is illustrated in Figure 8-15, followed by Figure 8-16.

Using $a[i] \geq \text{pivot value}$ and $a[j] \leq \text{pivot value}$ in step 3 and 4 allows copies of our pivot to be evenly distributed on both sides of our partition. Any other conditions ($a[i] > \text{pivot value}$ and $a[j] < \text{pivot value}$, or $a[i] > \text{pivot value}$ and $a[j] \leq \text{pivot value}$) will simply distribute the copies in one side of the partition only.

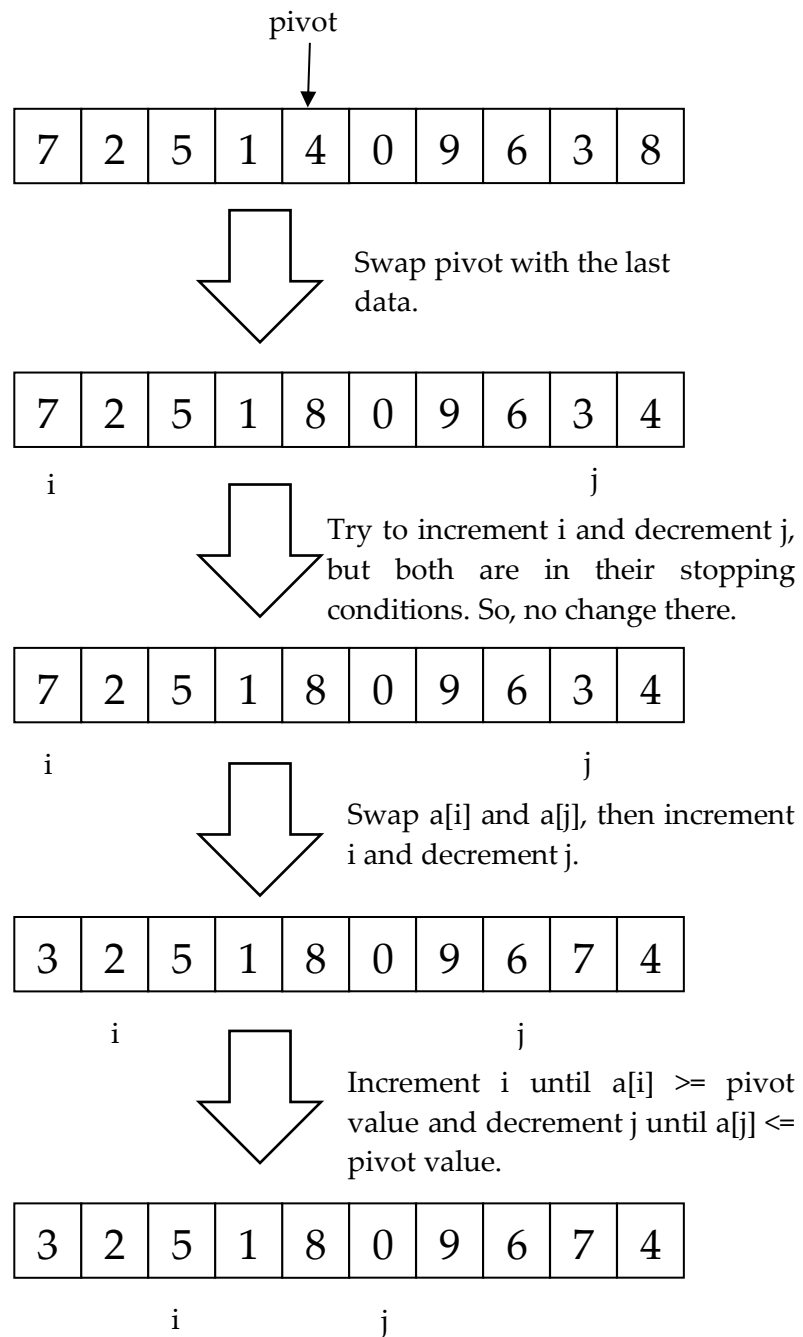


Figure 8-15: Partitioning example (part 1).

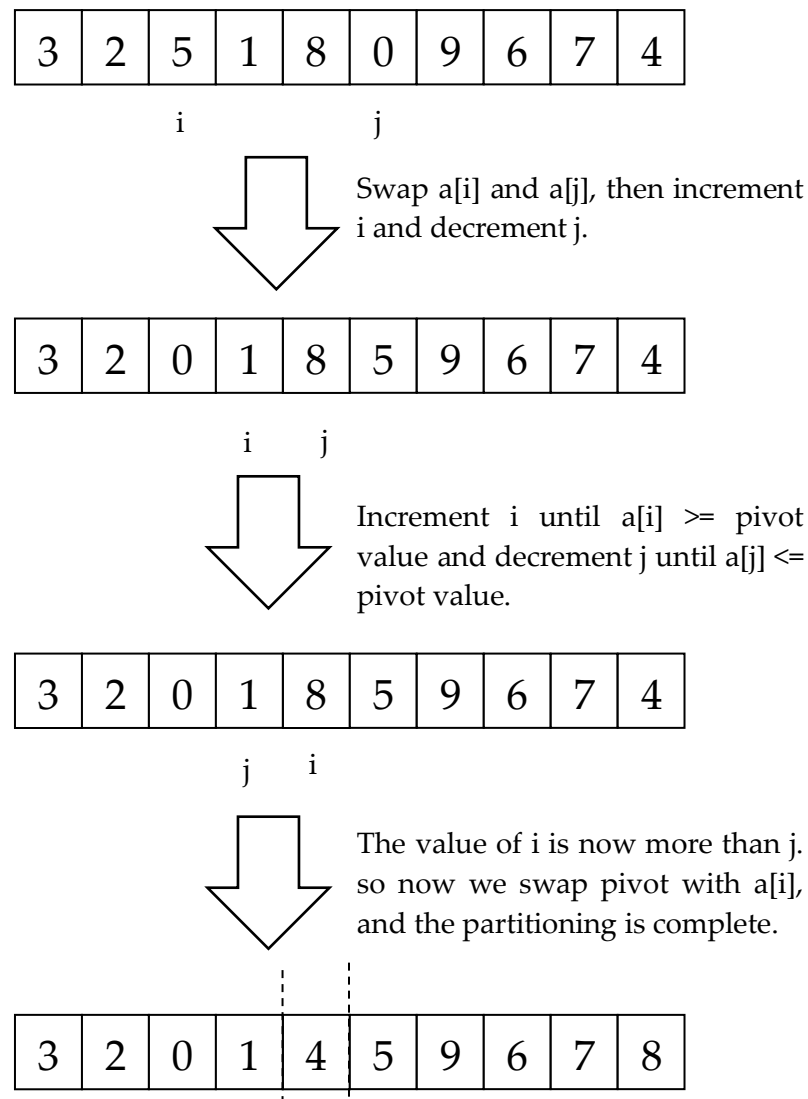


Figure 8-16: Partitioning example (part 2).

Implementation and Runtime of Quick Sort

Code for quick sort is given in Figure 8-17. It follows our algorithm in the previous section. The method receives the following inputs:

- $int[] a$: the input array.

- *int l*: the leftmost position of the input array that our sorting algorithm applies.
- *int r*: the rightmost position of the input array that our sorting algorithm applies.

The method performs quick sort from data at position *l* to *r* in the array. It changes the contents of the input array.

```
1:  static void quicksort(int[] a,int l, intr){
2:      if(l+CUTOFF>r){
3:          insertionSort(a,l,r);
4:      } else {
5:          //findpivot using median of 3.
6:          int pIndex=pivotIndex(a,l,r);
7:
8:          //get pivot out of the way.
9:          swap(a,pIndex,r);
10:         int pivot=a[r];
11:
12:         //start partitioning.
13:         int i=l, j=r-1;
14:         for( ; ; ){
15:             while(i<r && a[i]<pivot)i++;
16:             while(j>l && a[j]>pivot)j--;
17:             if(i<j){
18:                 swap(a,i,j);
19:                 i++;
20:                 j--;
21:             }else{
22:                 break;
23:             }
24:         } //end partitioning.
25:
26:         //swap pivot into its correct position.
27:         swap(a,i,r);
28:
29:         //quick sort on subarrays.
30:         quicksort(a,l,i-1);
31:         quicksort(a,i+1,r);
32:     }
33: }
```

Figure 8-17: Code for quick sort.

There are some parts of the code worth noting:

- *CUTOFF*: this value is the array length for when we choose to do another sorting algorithm instead of quick sort.
- *insertionSort(int[] a, int left, int right)*: this is an insertion sort that applies only from position *left* to *right* inclusively. The code for this method is not given in this book. It is recommended that readers modify the code from regular insertion sort.
- *swap(int[] a, int i, int j)*: this is a method that swaps value between position *i* and *j* in a given array. The code for this method is not given in this book. It is simple to implement, however.

To allow straightforward analysis of the runtime, we assume that random pivot selection is used and other sorting algorithms are not used when the array is small.

Let $T(n) = \text{runtime when the array length is } n$.

Therefore $T(0) = 1$ and $T(1) = 1$.

For other cases, the runtime is the sum of:

- time for pivot selection. If we use the median of 3 method, this time is constant and thus can be ignored.
- time for partitioning. This depends directly on the array size. Let the time be $c \times n$, where c is a constant and n is our array size.
- time for quick sorts on left and right subarrays.

If the left subarray has size equal to i , then

$$T(n) = T(i) + T(n - i - 1) + c \times n.$$

Let us first analyze the **worst-case runtime**. This case takes place when all our pivots happen to be the smallest value. In such situation, one subarray is always empty, the other's length is always reduced by 1 each time.

Therefore, $T(n) = T(n - 1) + T(0) + c \times n$.

With various n , we create the following set of equations:

$$\begin{aligned} T(n) &= T(n - 1) + T(0) + c \times n \\ T(n - 1) &= T(n - 2) + T(0) + c \times (n - 1) \\ T(n - 2) &= T(n - 3) + T(0) + c \times (n - 2) \\ &\dots \\ T(2) &= T(1) + T(0) + c \times 2 \end{aligned}$$

Adding all the equations above, we get:

$$T(n) = T(1) + (n - 1) \times T(0) + c \times (2 + 3 + 4 + \dots + n)$$

which simplifies to:

$$T(n) = T(1) + (n - 1) + c \sum_{i=2}^n i = \theta(n^2)$$

From our deduction, the worst-case runtime is similar to other sorting algorithms.

What about the best case? This happens when our pivot selection always divides the array in half. Our analysis for this is similar to merge sort.

The equation for $T(n)$ becomes:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c \times n$$

Dividing both sides by n , we get:

$$\frac{T(n)}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + c$$

Varying the value of n , we get the following set of equations:

$$\begin{aligned}\frac{T(n)}{n} &= \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + c \\ \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} &= \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + c \\ \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} &= \frac{T\left(\frac{n}{8}\right)}{\frac{n}{8}} + c \\ &\dots \\ \frac{T(2)}{2} &= \frac{T(1)}{1} + c\end{aligned}$$

Adding them all up (exactly like what we did with our analysis of merge sort), we get:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c \times \log_2 n$$

Multiplying both sides by n , we get:

$$T(n) = n + c \times n \times \log_2 n = \theta(n \log n)$$

Therefore, its best case performs at the same level as merge sort.

When sorting real life data, worst-case and best-case scenarios are unlikely to take place, however. The more useful runtime is the average case. For this, a subarray can have any size, from 0 to $n-1$ (a subarray cannot have size n because we are not counting the pivot).

For every subarray size to have equal chance of happening, each has a probability of $\frac{1}{n}$. Our equation for $T(n)$ becomes:

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{j=0}^{n-1} T(j) + \frac{1}{n} \sum_{j=0}^{n-1} T(j) + c \times n \\ &= \frac{2}{n} \sum_{j=0}^{n-1} T(j) + c \times n \end{aligned}$$

Multiplying both sides by n , we get:

$$n \times T(n) = 2 \sum_{j=0}^{n-1} T(j) + c \times n^2$$

Substituting n with $n-1$, we get another equation:

$$(n-1) \times T(n-1) = 2 \sum_{j=0}^{n-2} T(j) + c \times (n-1)^2$$

Subtracting the new equation from the previous one, we get:

$$n \times T(n) - (n-1) \times T(n-1) = 2 \times T(n-1) + 2cn - c$$

Ignoring a constant c , then move $(n-1) \times T(n-1)$ to the right-hand side, we get:

$$n \times T(n) = (n+1) \times T(n-1) + 2cn$$

Dividing both sides by $n(n+1)$, we get:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

Again, we can generate a set of equations:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

$$\dots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

When we add all equations in this set, we get:

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{n+1} \frac{1}{i}$$

where, $\sum_{i=1}^n \frac{1}{i}$ is a harmonic number. It is defined as:

$$\sum_{i=1}^n \frac{1}{i} = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon, \quad \text{where } n \geq 1, 0 < \varepsilon < \frac{1}{256n^6}, \gamma \approx 0.5772$$

Substituting the value of $\sum_{i=1}^n \frac{1}{i}$ into our $\frac{T(n)}{n+1}$ equation, we get:

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \left(\ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon - 1 - \frac{1}{2} \right)$$

It can be seen that the right-hand side is dominated by $\ln n$. Therefore, when we get rid of $n+1$ on the left-hand side, $T(n) = \theta(n \log n)$. So, in general, quick sort performs at the same level as merge sort.

Bucket Sort

Bucket sort is a type of sorting algorithms that generally performs well because we know where each data will go.

For example, putting each card in a 52-card deck on a table.

- We only need to prepare a space for each card.
- When we look at a card, just put it at its provided space.
- Therefore, picking a card means we know its position automatically.
- The running time is $\theta(n)$, which is the time to look at n cards.

A space for each card is called a bucket. For the above example, one bucket stores one data.

Let us look at another example. If we have n numbers in a range of 1 to m , where $n < m$, we can sort these numbers by:

- Creating an array, a , of size m . This will be a frequency array. Position 1 stores the frequency of value 1. Position m stores the frequency of value m .
- Each array slot initially stores 0.
- Read each number, for number k , we increment $a[k]$ by 1.
- When finish reading all the data, we will get a frequency of each number.

- We can then read the frequency array and construct our sorted data.
- The running time is $\theta(n)$ for constructing the frequency array and $\theta(m)$ for constructing our sorted result.

A bucket may store more than one distinct objects. As an example, consider the problem of sorting exam papers collected from 49 students:

- At collection time, an examiner can divide students into 5 groups (1-9,10-19,...,40-49).
- Within a group, we can use a sorting method such as insertion sort.
- After sorting within a group, simply put all groups in sequence.
- The running time depends on the method used to sort within buckets.

Bucket sort can be designed according to our knowledge of data. The more we know about the data, the better our sorting algorithm can be created.

Exercises

1. Use recursion to implement insertion sort.
2. Draw what happen when we perform a merge sort on array {4,78,3,34,1,45,7,8}.
3. Draw what happen when we perform a quick sort on array {4,78,3,34,1,45,7,8,10,20,15,24}.

4. In quick sort, instead of normal partitioning, we do the followings:
 - a) Compare all data with the leftmost data of the portion we want to sort. Count how many data are smaller than the leftmost data. Hence, we now know the correct position of the leftmost data.
 - b) Move the leftmost data to the position found in the previous step.

Is there anything wrong with this method of partitioning? If so, please show an example.

5. Show how you derive the asymptotic runtime of quicksort when data in the input array are already sorted. Do it for the case when our pivot is always the first data, and when our pivot is chosen randomly.
6. Show how you derive the asymptotic runtime of quicksort when data in the input array are already sorted from large to small. Do it for the case when our pivot is always the first data, and when our pivot is chosen randomly.
7. If we know that there are many copies of each data, in quick sort, we can partition our input array into 3 portions: data that are more than our pivot, data that are less than our pivot, and data that are equal to our pivot. Please rewrite the code for this new quick sort.
8. Compare insertion sort, quicksort and bucket sort. Explain their differences. In what circumstances will you choose one over others.
9. Write a special selection sort such that:

-
- For any two odd numbers, they are arranged from small to large.
 - For any two even numbers, they are arranged from small to large.
 - Odd numbers always come before even numbers.
10. We have a class *ListQuickSort* which will be used to implement quick sort for linked list.

```
public class ListQuickSort{
    CDLinkedList theList; //linked list from chapter 3
}
```

Implement the following methods. For each method state its asymptotic runtime.

DListIterator listIterator(int i)

This method returns a list iterator that focuses on position *i* in *thelist* (the first position on the left of the list has index = 0). If the position given by *i* is not in the list, the method returns an iterator that points to *null*.

DListIterator findPivot()

This method returns a list iterator that points to the median amongst the first, the middle, and the last element of the list. (throw exception if our list has less than 3 elements.)

public void swap(DListIterator i, DListIterator j)

This method swaps the positions of two elements (each one identified by a list iterator).

public void partition(DListIterator itr)

This method receives an iterator (that represents a pivot), then move all elements which are less than or equal to the pivot value to the left of the pivot and elements that are greater than or equal to the pivot value to the right of the pivot.

Write code for quick sort. Is there anything you need to make change?

- 11.If you want to sort an array of positive integers containing n numbers but you do not know the range of the values in the array, explain and write code for method:

public static int[] sort(int[] input)

This method sorts the array so that the big O is less than $O(n \log n)$. Discuss the big O of your code and any limitation your code has.

- 12.Write code for method:

void sortTwoArrays(int[] a1, int[] a2).

This method sorts two arrays. When finishes, both arrays must be sorted (from small to large numbers) and all numbers in $a1$ must be smaller than all numbers in $a2$. Assume that $a1$ and $a2$ are not null and you can call *void sort(int[] a)* that can sort values in a given array from small to large.

For example:

If $a1 = \{8,4,7,5,2\}$ and $a2 = \{6,10,3,0,11,1\}$. After the method is called, $a1$ will be $\{0,1,2,3,4\}$ and $a2$ will be $\{5,6,7,8,10,11\}$.

13. Write code for insertion sort that operates on a doubly-linked list. Write code as part of class *CDLinkedList* from chapter 3.
14. For class *BSTRecursive* in chapter 6, Write code for method:

int[] toSortedArray()

This method returns an array of elements contained in the tree, where:

- Elements in the returned array must be sorted from small to large.
- The tree must still contain all its contents when the method finishes its execution.

What is the asymptotic runtime of your code.

15. Write code for method:

public int[] sortFirstN(int[] a, int n)

This method sorts the first n integer in a from small to large. There is no change to other integers in the array. The method returns a modified array. Assume n is smaller than array size.

Chapter 9 : Priority Queue

A priority queue is a queue that accesses its elements according to their importance. For example, at a hospital, a person with broken back should be treated before a person with minor wounds, even though he arrives later.

In this book, small value is regarded as more important than large value. If two data have equal priority, we may regard the data that exists longer in the queue to be more important.

To compare data stored inside a priority queue, we can use Comparable interface or Comparator interface in the Java language.

In this book, Comparator interface is used. It has method:

compare(Object o1, Object o2)

o1's type has to be compatible with *o2*. This method returns a negative value if *o1* is less than *o2*, a positive value if *o1* is greater than *o2*, and it returns 0 otherwise.

Priority queue has the following operations:

- *size()*: returns the number of data currently stored in the priority queue.
- *isEmpty()*: returns true if our priority queue does not store any item, and false otherwise.

- *add(Object data)*: adds data to the priority queue.
- *top()*: returns the most important data (the smallest data).
- *pop()*: removes and returns the most important data.

These operations are defined in a Java interface in Figure 9-1.

```
1:  public interface PriorityQ {
2:      // Postcondition: return the number of
3:      // data in this priority queue.
4:      int size();
5:
6:      // Postcondition: return true if this priority
7:      // queue does not store any data, otherwise
8:      // return false.
9:      boolean isEmpty();
10:
11:     // Postcondition: element is
12:     // added to priority queue.
13:     void add(Object element) throws Exception;
14:
15:     // Throws NoSuchElementException if heap is
16:     // empty.
17:     // Postcondition: return the most important data.
18:     public Object top() throws Exception;
19:
20:     // Throws NoSuchElementException if heap is
21:     // empty.
22:     // Postcondition: remove and return the most
23:     // important data.
24:     public Object pop() throws Exception;
25: }
```

Figure 9-1: Priority queue operations.

Implementation Choices

There are quite a few possible implementations for a priority queue. Let us briefly investigate each possible implementation.

- normal queue: cannot be used because it does not have priority.
- Array of queues (each queue is used for each priority): if there are many possible values for priorities, the array will consume too much space because we have to reserve space in advance.
- Linked list of queues:
 - does not have the space problem like array of queues do, but instant access to each priority is eliminated. Searching for a priority will require the runtime of $O(p)$, where p is the number of priorities. If every data has a different priority, the time is $O(n)$.
 - $pop()$ and $top()$ has runtime = $\Theta(1)$. The first data in the first queue is always accessed first.
- *ArrayList*:
 - $add(data)$ takes time to find a position to add $data$. Finding the position takes $O(\log n)$ if a binary search is used. Furthermore, many data have to be shifted to the right to make space for the added value (takes $O(n)$). Therefore, adding a new data takes $O(n)$, which is not very good.

- *top()* takes $\theta(1)$ because we just return the first data in the *ArrayList*.
- *pop()* takes $\theta(n)$ because we have to shift all data after the leftmost data is removed.
- Linked list:
 - *top()* and *pop()* take $\theta(1)$ because we can return the front of the list right away and removing data in a linked list does not require shifting other data.
 - *add()* takes $O(n)$, it still needs to search the position to add, but does not need to shift other data when adding.
 - The performance is about equal to using linked list of queues.

Using a linked list seems to be a good implementation. Let us see the actual implementation using linked list (class *CDLinkedList* from chapter 3).

Linked List Implementation of Priority Queue

The linked list implementation is in class *PQDLinkedList* (shown in Figure 9-2 and Figure 9-3). Our implementation contains a comparator and a linked list to store data. Our linked list in this chapter is modified to store *Object* instead of *int*. Figure 9-2 shows fields, constructors, *size()*, *isEmpty()*, and method *compare*. Method *compare* checks if we have a comparator, if so, it

compares the two input data using that comparator (line 27). Otherwise, a *compareTo* method for the stored data is used (that class must implement Comparable interface).

This *compareTo* method is defined as follows:

- *data1.compareTo(data2)* returns a negative integer if *data1* is less than *data2*.
- *data1.compareTo(data2)* returns zero if *data1* is equal to *data2*.
- *data1.compareTo(data2)* returns a positive integer if *data1* is larger than *data2*.

```
1: public class PQDLinkedList implements PriorityQ {
2:     CDLinkedList list;
3:     Comparator comp;
4:
5:     public PQDLinkedList() {
6:         list = new CDLinkedList();
7:         comp = null;
8:     }
9:
10:    public PQDLinkedList(CDLinkedList l,Comparator c)
11:    {
12:        this.list = l;
13:        this.comp = c;
14:    }
15:
16:    public int size() {
17:        return list.size;
18:    }
19:
20:    public boolean isEmpty() {
21:        return list.isEmpty();
22:    }
23:
24:    protected int compare(Object d1, Object d2) {
25:        return (comp == null ?
26:            ((Comparable) d1).compareTo(d2) :
27:            comp.compare(d1, d2));
28:    }
29:    // this class continues in Figure 9-3.
```

Figure 9-2: Priority queue implemented by linked list (part 1).

Method *top*, *pop*, and *add* are shown in Figure 9-3.

Method *top* throws an exception if our list is empty. Otherwise, it returns a data stored in a node next to *header*. This is the first data. Method *top* runs in $\Theta(1)$ since there is no loop.

```
1:  public Object top() throws Exception {
2:      if (list.isEmpty())
3:          throw new Exception();
4:      return list.header.nextNode.data;
5:  }
6:
7:  public Object pop() throws Exception {
8:      if (list.isEmpty())
9:          throw new Exception();
10:     Object result = top();
11:     list.remove(new DListIterator(list.header));
12:     return result;
13: }
14:
15: public void add(Object d) throws Exception {
16:     if (list.isEmpty()) {
17:         DListIterator i =
18:             new DListIterator(list.header);
19:         list.insert(d, i);
20:     } else if (compare(d,
21:         list.header.previousNode.data) >= 0){
22:         DListIterator last =
23:             new DListIterator(list.header.previousNode);
24:         list.insert(d, last);
25:     } else {
26:         DListIterator itr =
27:             new DListIterator(list.header.previousNode);
28:         while (compare(d, itr.currentNode.data) < 0)
29:             itr.previous(); // back up one position
30:         list.insert(d, itr);
31:     }
32: } // end of class PQDLinkedList.
```

Figure 9-3: Priority queue implemented by linked list (part 2).

Method *pop* is similar. It throws an exception if our list is empty. Otherwise, it records data stored in the node after header, removes that node from our list, and returns the recorded value. Its runtime is also $\theta(1)$ since there is no loop.

Method *add* is used to add a new data into our priority queue. It receives the new data, d , as its input. It does the followings:

- If our list for data storage is empty (line 16-19): create a new node and store d inside that node. Then add the new node just behind the list's *header*. The runtime for this action is $\theta(1)$ since it does not involve any loop.
- If d is larger than or equal to the last data in our linked list (line 20-24): add a new node (with d inside) just after the last node of the list. The runtime for this part is also $\theta(1)$.
- Otherwise (line 26-30): we create an iterator pointing to the last node in the list. We keep moving our iterator to the left until d is larger than or equal to the data pointed to by our iterator. We then add a new node (with d inside) right after the node pointed to by our iterator. This process takes $O(n)$ since the loop may finish early or may go all the way through the entire list.

Heap

In this section, we will look at another implementation of priority queue that is more popular than using a linked list. This implementation is called heap.

A heap is a complete binary tree (a complete binary tree is defined in chapter 6) that has the following properties:

- It is an empty tree, or
- A tree that has its most important data in its root.
- Left and right subtrees must be heaps too (and so as their left and right subtrees, recursively).

Please note that a heap is not a binary search tree! A heap stores its data in a completely different manner.

The implementation in this text regards the most important data to be the smallest data (we call this kind of heap a min heap).

A min heap is shown in Figure 9-4. Every subtree stores its smallest value at its root.

Heap Implementation and Runtime Analysis

A complete binary tree can be represented by an array. We just need to traverse the tree from left to right down the trees' levels (breadth-first). Thus, the tree in Figure 9-4 is represented by an array in Figure 9-5.

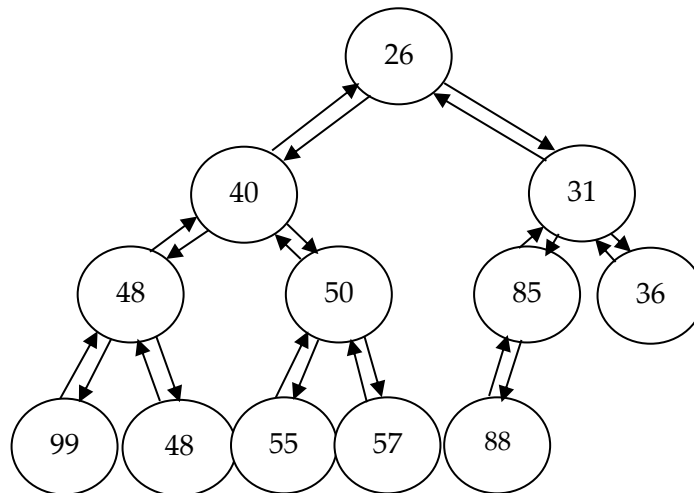


Figure 9-4: A min heap example.

26	40	31	48	50	85	36	99	48	55	57	88
----	----	----	----	----	----	----	----	----	----	----	----

Figure 9-5: Array representation of heap in Figure 9-4.

Although the representation is now an array (an array slot represents a node of the actual tree), we can still traverse to left child, right child, and parent of each node using the following rules:

- If our current data is in slot number i ,
 - the slot that represents slot i 's left child is slot $2*i+1$.
 - Similarly, the slot that represents slot i 's right child is slot $2*i+2$.
 - The slot that represents slot i 's parent is slot $\frac{i-1}{2}$.

Let us see an example. From Figure 9-5, value 50 is in the 4th slot:

- By the rules, the value at its left is in the $(2*4+1) = 9^{\text{th}}$ slot. That value is 55 (which is correct according to Figure 9-4).
- The value at 50's right is in the $(2*4+2) = 10^{\text{th}}$ slot. That value is 57 (which is correct according to Figure 9-4).
- The value at 50's parent is in the $\frac{4-1}{2} = 1^{\text{st}}$ slot. That value is 40 (which is correct according to Figure 9-4).

The code for class *Heap* is shown from Figure 9-6 to Figure 9-12.

```
1: public class Heap implements PriorityQ {
2:     Object[] mData;
3:     int size = 0;
4:
5:     public Heap() {
6:         final int DEFAULT_CAPACITY = 11;
7:         mData = new Comparable[DEFAULT_CAPACITY];
8:     }
9:
10:    public boolean isEmpty() {
11:        return size == 0;
12:    }
13:
14:    public int size() {
15:        return size;
16:    }
17:    // this class continues in Figure 9-8.
```

Figure 9-6: Code for constructor, *isEmpty()*, and *size()* of class *Heap*.

In Figure 9-6, fields, constructor, method *isEmpty* and method *size* are defined. Our heap simply stores its data in array *mData*. It has another field, *size*, for recording the number of data currently stored in the array. Method *isEmpty* and *size* simply use the field.

For method *add(Object data)*, we use the following algorithm:

1. We add *data* as the last data in the last tree level.
2. Then we swap *data* with the one in its parent node if *data* is smaller.
3. We keep swapping *data* up the tree until the tree is heap once more.

Moving *data* up the tree after putting it in is called “percolate up”. An example of what happens when adding 30 to a heap is shown in Figure 9-7.

First, 30 is added as the last data (in the last level) in our heap. Then it is compared with the data in its parent node, which is 48, causing 30 and 48 to be swapped. Then 30 is compared to the data in its parent node, which is now 40. They got swapped because 30 has smaller value (more important). After that, 30 is compared to 26 in its parent node, but 26 is smaller so 30 does not get swapped with 26. Our procedure then ends.

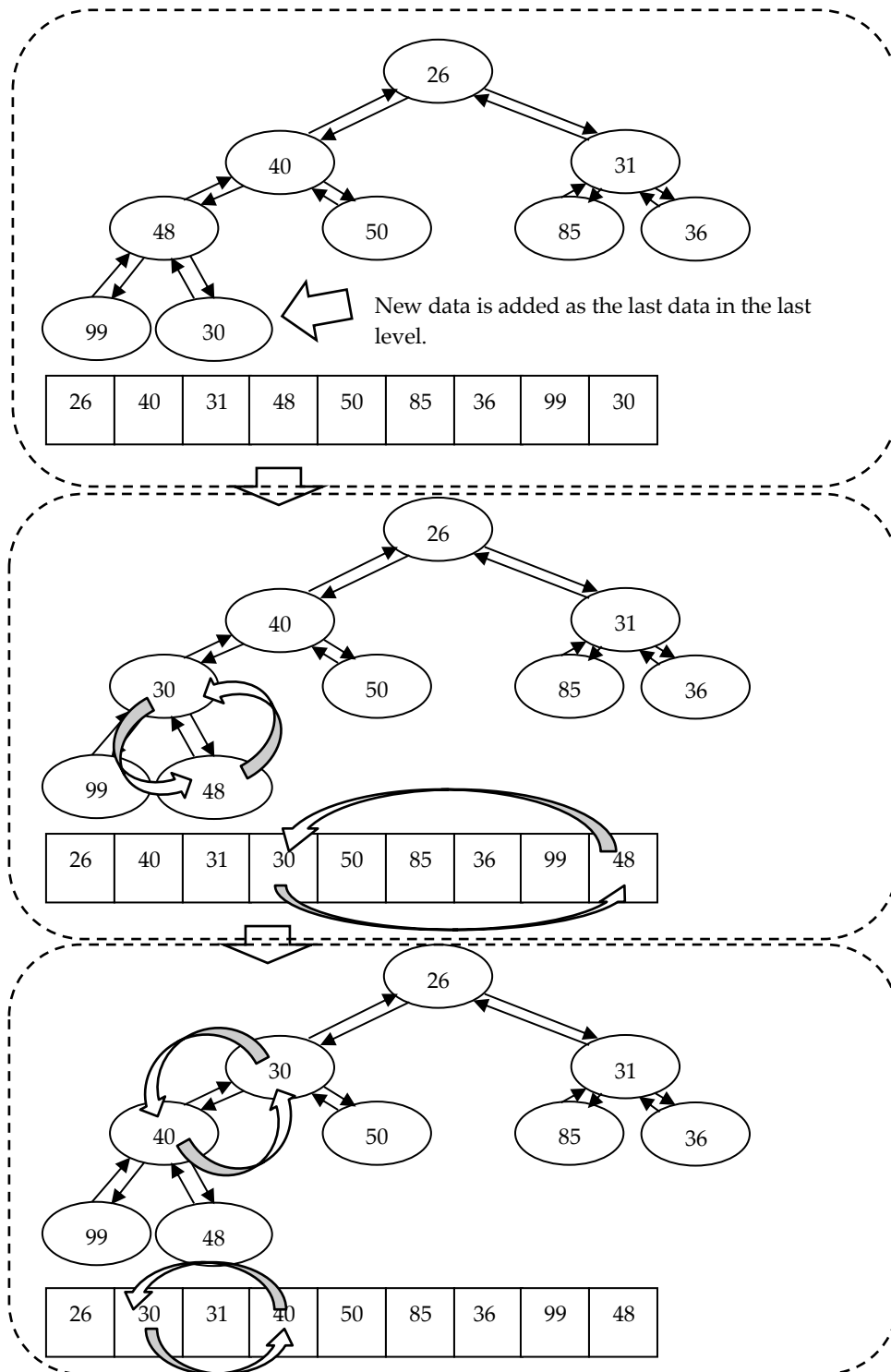


Figure 9-7: Each step for adding 30 into a heap, showing both the tree version and its array implementation.

```
1:  public void add(Object element) {
2:      if (++size == mData.length) {
3:          Object[] newHeap;
4:          newHeap = new Object[2 * mData.length];
5:          System.arraycopy(mData, 0, newHeap, 0, size);
6:          mData = newHeap;
7:      }
8:      mData[size - 1] = element;
9:      percolateUp();
10: }
11:
12: protected void percolateUp() {
13:     int parent;
14:     int child = size - 1;
15:     Comparable temp;
16:     while (child > 0) {
17:         parent = (child - 1) / 2;
18:         if (((Comparable)
19:             mData[parent]).compareTo(mData[child]) <= 0)
20:             break;
21:         temp = (Comparable) mData[parent];
22:         mData[parent] = mData[child];
23:         mData[child] = temp;
24:         child = parent;
25:     }
26: } //this class continues in Figure 9-9.
```

Figure 9-8: Code for method *add* of class *Heap*.

Although class *Heap* is defined to store data of type *Object*, examples in this chapter uses integer data for easier understanding.

The code for method *add* is shown in Figure 9-8. The worst-case runtime for method *percolateUp* takes place when the newly added data has to be moved up the entire height of our tree. Hence, we can write down the runtime of method *percolateUp* as $O(\log n)$, where n is the number of data. The asymptotic runtime for method *add*

is the worst-case runtime from method *percolateUp* (called on line 9 of Figure 9-8) and other parts of the code.

It turns out that the worst-case runtime for method *add* takes place when our array needs to be resized (line 2-7 of Figure 9-8). For such case, all data need to be copied to an expanded array, therefore the runtime is $O(n)$. This runtime is worse than the worst-case runtime of method *percolateUp* ($O(\log n)$). Therefore, the runtime for method *add* is $O(n)$, which is the same as the time for method *add* when using a linked list or other implementations.

The advantage of using heap over other implementations comes when we consider the average runtime. On average, a newly added value will be the value in the middle of existing values (if sorted from left to right). Since our heap is a complete binary tree, half of the heap's values are at its leaves. This means swapping the newly added value up the tree just once will get it to its correct position. Therefore, it takes constant time (on average) to add a new data into a heap.

Method *top* is shown in Figure 9-9. If there is no data, the method cannot return any value, so it throws an exception. Otherwise, the method returns the data stored inside the root of our heap, which is the first array slot in our implementation.

```
1: public Object top() throws Exception {
2:     if (size == 0)
3:         throw new Exception("Empty");
4:     return mData[0];
5: } //this class continues in Figure 9-12.
```

Figure 9-9: Code for method *top* of class *Heap*.

To pop a data out of our heap, we have to be careful because removing a node can destroy our complete binary tree structure (even though we can preserve the data ordering from the root), thus destroying our array representation of the heap, as shown in Figure 9-10.

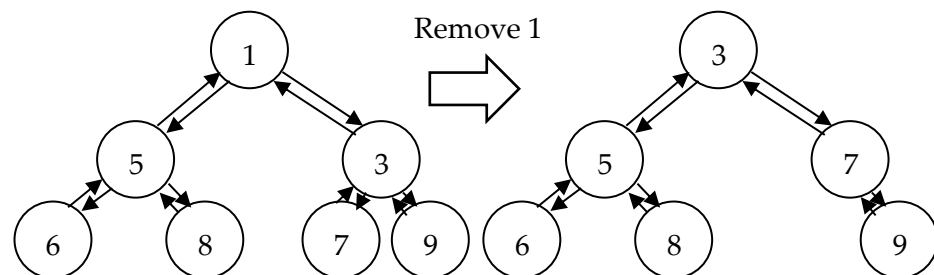


Figure 9-10: Removing a root without using a special algorithm, destroying a complete binary tree structure.

To preserve our complete binary tree, we must use the following algorithm when removing the most important value:

- Overwrite the value at the root with the last value from the last level of our complete binary tree.

- Swap the value at the root down the tree (swap with the most important child) until the tree becomes heap again.

Swapping our data down the tree is called “percolate down”. Figure 9-11 shows what happens when we remove 26 (the most important value) from the heap we obtained in Figure 9-7.

First, the data at the root (26) is replaced by the last data. Hence the data at the root now becomes 48. Then 48 is compared with data in its left and right child (30 and 31). The value 30 is the smallest of the two children values and it is smaller than 48 so it is swapped with 48. After this swap, 48 is then compared to data in its new left and right child (40 and 50). The value 40 is the smallest of the two and it is smaller than 48, so 48 is swapped with it. After that, 48 is then compared with data in its new left child (99) (no right child to compare since we consider it removed from our heap). This time, no swap occurs since 48 is already smaller than 99.

The code for method *pop* is shown in Figure 9-12.

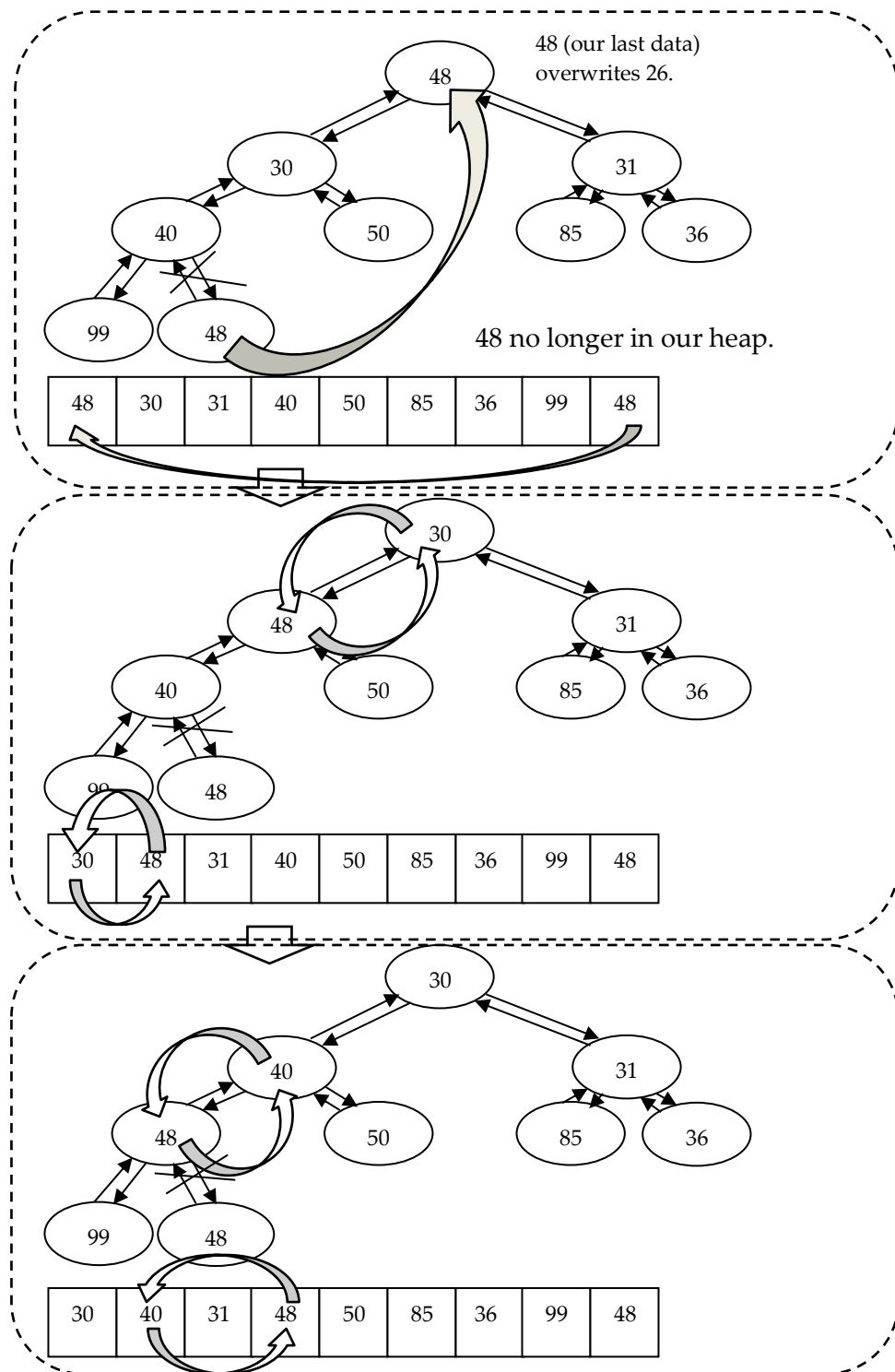


Figure 9-11: Each step for removing the most important value from a heap, showing both the tree version and its array implementation.

```
1: public Object pop() throws Exception {
2:     if (size == 0)
3:         throw new Exception("Priority queue empty.");
4:     Object minElem = mData[0];
5:     mData[0] = mData[size - 1];
6:     size--;
7:     percolateDown(0);
8:     return minElem;
9: }
10:
11: protected void percolateDown(int start) {
12:     int parent = start;
13:     int child = 2 * parent + 1;
14:     Object temp;
15:     while (child < size) {
16:         Comparable lVal = (Comparable)mData[child];
17:         if (child < size - 1)
18:             Comparable rVal = (Comparable)mData[child+1];
19:             if(lVal.compareTo(rVal) > 0)
20:                 child++;
21:
22:         Comparable pVal = (Comparable) mData[parent];
23:         if(pVal.compareTo(mData[child]) <= 0)
24:             break;
25:         temp = mData[child];
26:         mData[child] = mData[parent];
27:         mData[parent] = temp;
28:         parent = child;
29:         child = 2 * parent + 1;
30:     }
31: } //end of code for class Heap.
```

Figure 9-12: Code for method *pop* of class *Heap*.

From Figure 9-12, if the heap is empty, we throw an exception because there is nothing to return (line 2-3). Otherwise, we replace the data at the root with the last data (line 5), reduce the value of *size* (line 6) so that the last data is treated as no longer in the heap, then call method *percolateDown* to move the new value in the root down the tree, before returning the original data at the root.

Method *percolateDown* receives the position index of the root as its input. We then initialize 2 variables, *parent* and *child*. Variable *parent* indicates the position (in our array representation) of our to-be-moved-down-the-tree data. Variable *child* indicates the position of *parent*'s child that stores the more important value amongst the left child and the right child. Before every iteration, this is set to the position of the left child (line 13 and line 29).

The loop iteration is performed as long as *child* does not go beyond the last possible position in the last level of our tree (line 15). Inside the loop, *child* can change to indicate the right child of *parent* if the right child exists (line 17) and it stores a more important value than the left child (line 19). After *child* is updated, it is compared with the value stored in position *parent* (line 23). If position *parent* stores a more important value, we exit the method since there is no need to do any swap (line 24). Otherwise, values in position *parent* and *child* need to be swapped, and new *parent* and *child* are set to reflect the new position of our data that gets swapped down the tree (line 25-29).

The asymptotic runtime of method *pop* depends on the runtime of method *percolateDown*. The worst-case runtime for *percolateDown* takes place when we have to swap our data down the entire tree. Thus, it directly depends on the tree's height. If the number of data is n , our runtime for method *pop* is therefore $O(\log n)$.

For an average case, the new root data has value in the middle of all existing values. That means, half of the tree's data have greater values. But because we use a complete binary tree, half the data must be at its leaves. Method *percolateDown* will have to move the root data down to the level before last. So, it is almost like the worst case.

Table 9-1 summarizes the average runtime of each method for linked list implementation and heap implementation of a priority queue.

Table 9-1: Average runtime for method *add*, *top*, and *pop* in linked list implementation and heap implementation of priority queue.

Method	Linked list implementation	Heap implementation
<i>add</i>	$O(n)$	$\Theta(1)$
<i>top</i>	$\Theta(1)$	$\Theta(1)$
<i>pop</i>	$\Theta(1)$	$O(\log n)$

Priority Queue Application: Data Compression

Let us try to store a text file with 100,000 characters. Normally, we need 16 bits to represent a character. Therefore 100,000 characters need 1,600,000 bits.

We can reduce the number of bits by introducing our own encoding. For example, we can represent a character using only 3 bits:

- 'a' is 000
- 'b' is 001
- 'c' is 010
- 'd' is 011
- 'e' is 100

If our 100,000-character file contains only characters 'a' to 'e', we only need 300,000 bits to store the file.

Can the number of bits be reduced further? Yes, we can try the following encoding:

- 'a' is 0
- 'b' is 1
- 'c' is 00
- 'd' is 01
- 'e' is 10

With the new encoding, the number of bits is reduced even more, but a new problem arises. The problem is we can compress the file, but when we try to uncompress it, we can't get the original characters back because the decoded result is ambiguous.

For example, if our compressed file contains 001010. We can decode it as "aababa", or "cee", or "adae", or any other possible values. There is no way that we can be

sure we will get the original string (before compressed) back.

How do we encode and prevent ambiguity at the same time? We can do it by utilizing a binary tree. Drawing a binary tree with enough leaves, a leaf represents a character in our text file. A character encoding is indicated by a sequence of 0s and 1s marked on branches from root to the leaf. A left branch represents 0, a right branch represents 1. Figure 9-13 shows a possible encoding of 'a' to 'e', using a binary tree. For this encoding, 'a' is 010, 'b' is 11, 'c' is 00, 'd' is 10, and 'e' is 011 (follow the marked branches from root to each leaf).

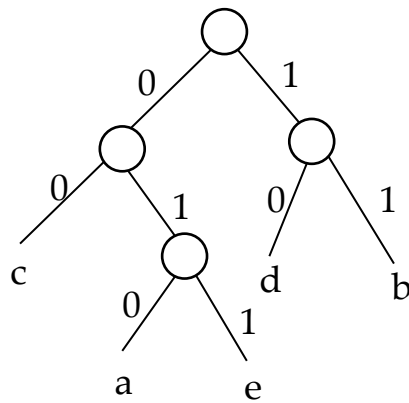


Figure 9-13: A possible encoding for character 'a' to 'e' in a text file.

With this encoding, there is no ambiguity. Bit string 001010 in our text file can only be interpreted as "cdd". No other strings are possible.

But there are more than one possible trees that can be constructed. Figure 9-14 shows another possible tree that we can use.

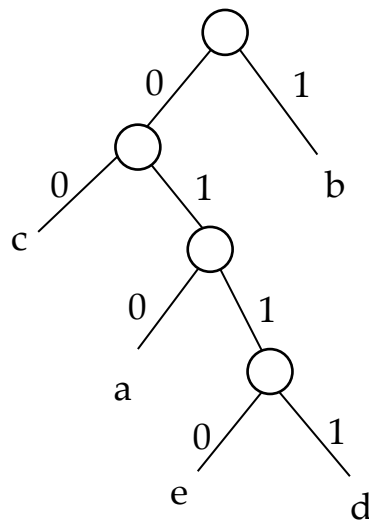


Figure 9-14: Another possible tree for encoding 'a' to 'e'.

So which tree is better? The answer is - it depends on a file we try to compress. Basically, if a character is very frequent in our file, we want that character to be represented using as few number of bits as possible in order to minimize the total number of bits for our compression.

That means, for each file to compress, we have to build a tree. Each file will have a different tree.

An optimum encoding tree for a file is called a **Huffman tree**. Please note that a Huffman tree is neither a binary search tree nor a heap. It is a tree that shows our encoding of data.

A Huffman tree can be constructed using a priority queue. First, we have to prepare our priority queue with the following algorithm:

1. Count the frequency of all characters in our to-be-compressed text file.
2. For each character, make a “node” of data that consists of:
 - a. That character
 - b. The frequency of that character
 - c. left (will later point to another node)
 - d. right (will later point to another node)
3. Put all the nodes (these will be nodes in our Huffman tree) of data we made into a priority queue. Our priority queue regards the node with the lowest frequency as the most important data.

Then our Huffman tree can be constructed using the following algorithm:

1. Remove two nodes with the smallest frequency from the priority queue.
2. Create a new node using:
 - a. the first removed node as its left branch.
 - b. the second removed node as its right branch.
 - c. The new node character is “”.

- d. The new node frequency is the sum of the frequency from both removed nodes.
3. Put the new node back into our priority queue.
4. Go back to do step 1 to 3 again until there is only one node left in our priority queue. That node is the root of our encoding tree (it will have already been linked with the rest of the tree).

Let us compress a text file with only characters 'a' to 'e'. Let the frequency count of each character be as follows:

- 'a' = 5000
- 'b' = 10000
- 'c' = 20000
- 'd' = 31000
- 'e' = 34000

Our priority queue constructed from these data stores the nodes to be retrieved in the order (from left to right) shown in Figure 9-15. We choose to represent our priority queue as a sequence of to-be-retrieved nodes in order to keep readers away from any particular priority queue implementation and focus on Huffman tree construction.

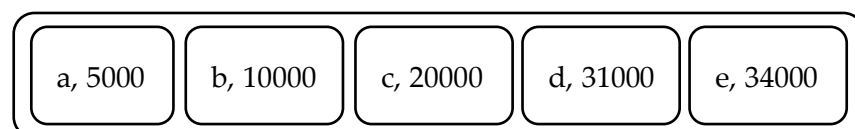


Figure 9-15: An Example of order of nodes to be retrieved from priority queue that stores nodes of Huffman tree.

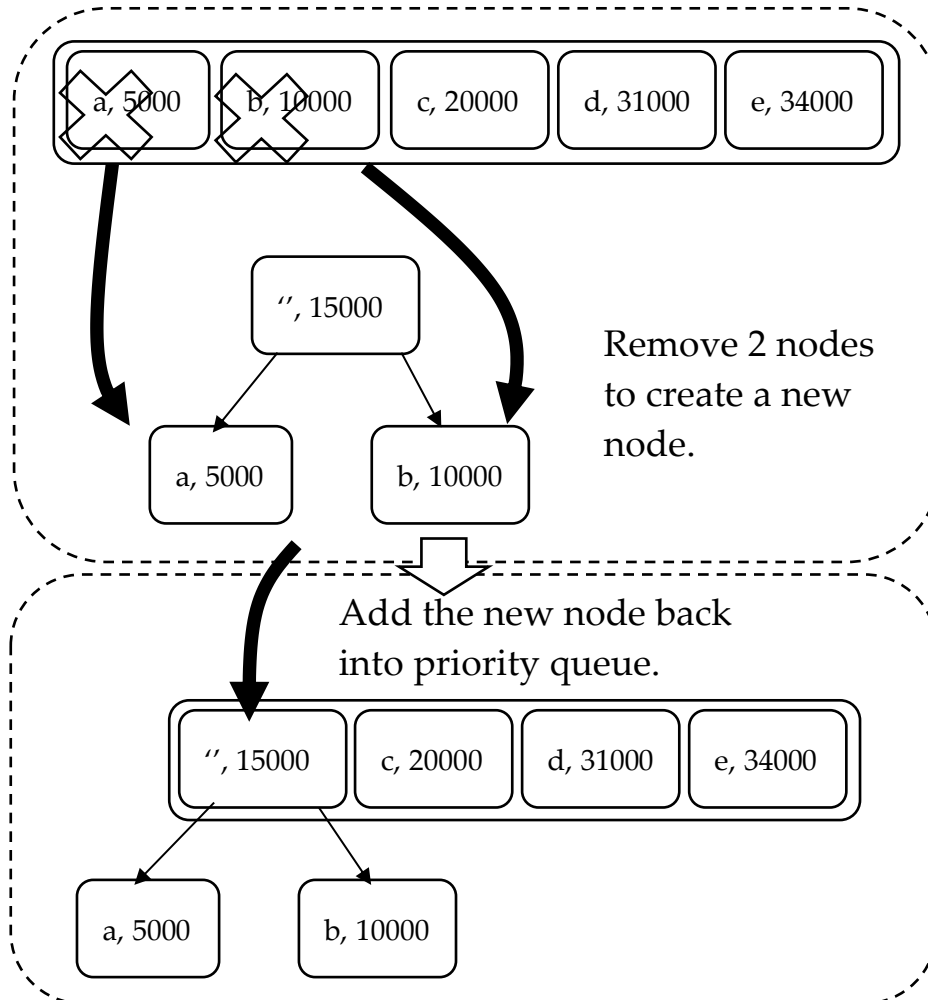


Figure 9-16: Example of Huffman tree creation (first iteration).

From our example text file, our Huffman tree construction is shown from Figure 9-16 to Figure 9-19. Each figure represents one iteration in our Huffman tree construction algorithm.

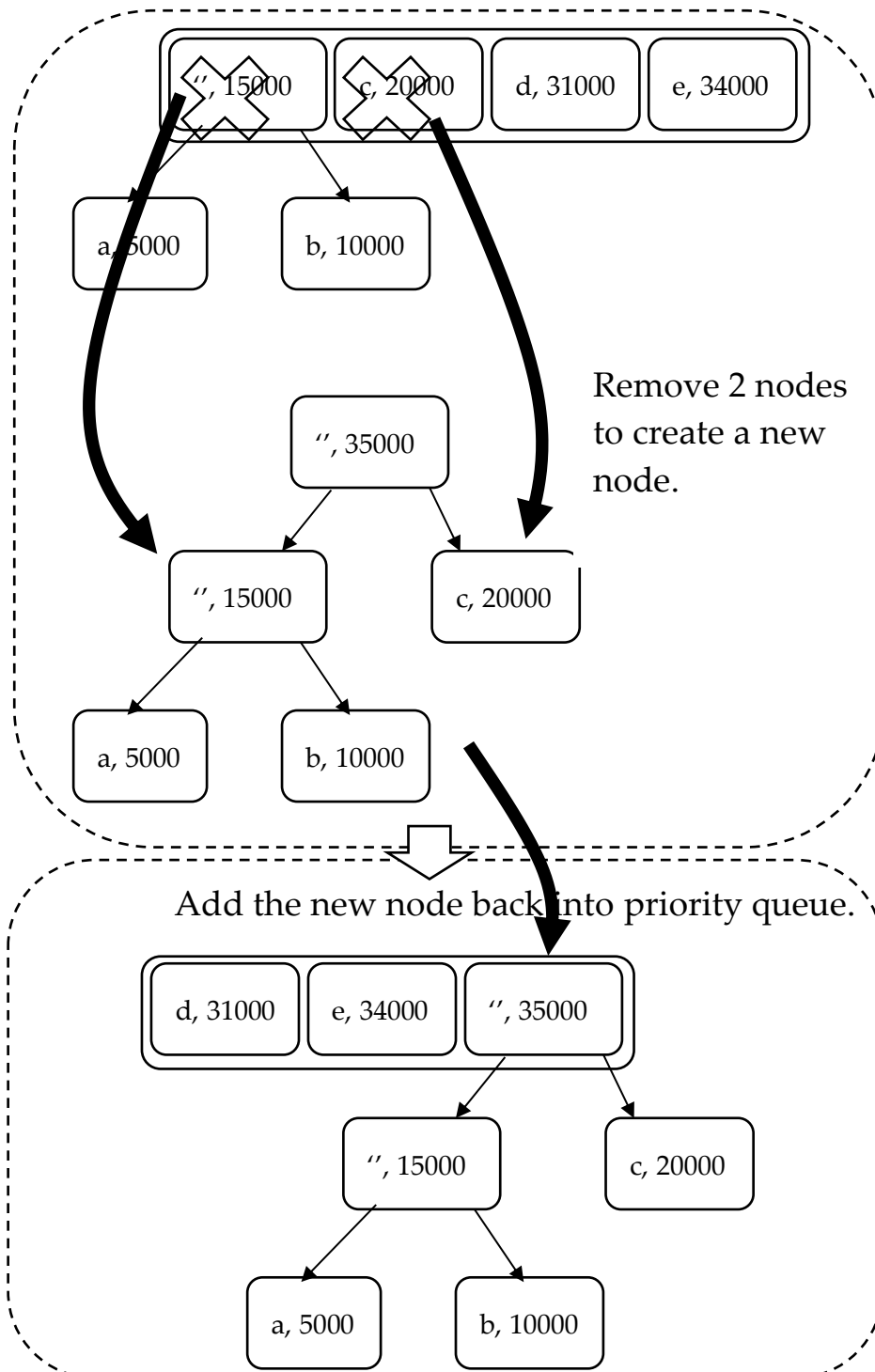


Figure 9-17: Example of Huffman tree creation (second iteration).

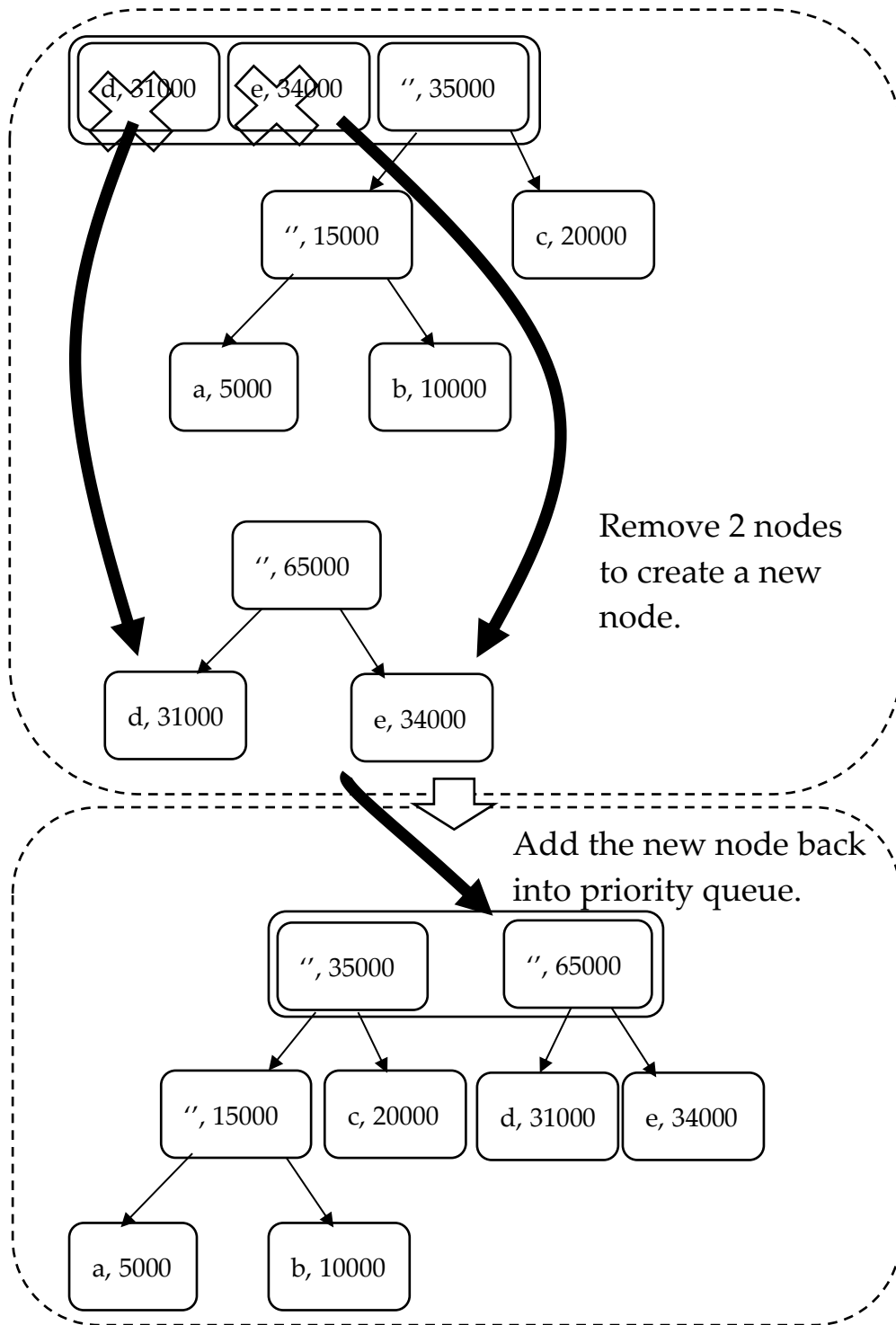


Figure 9-18: Example of Huffman tree creation (third iteration).

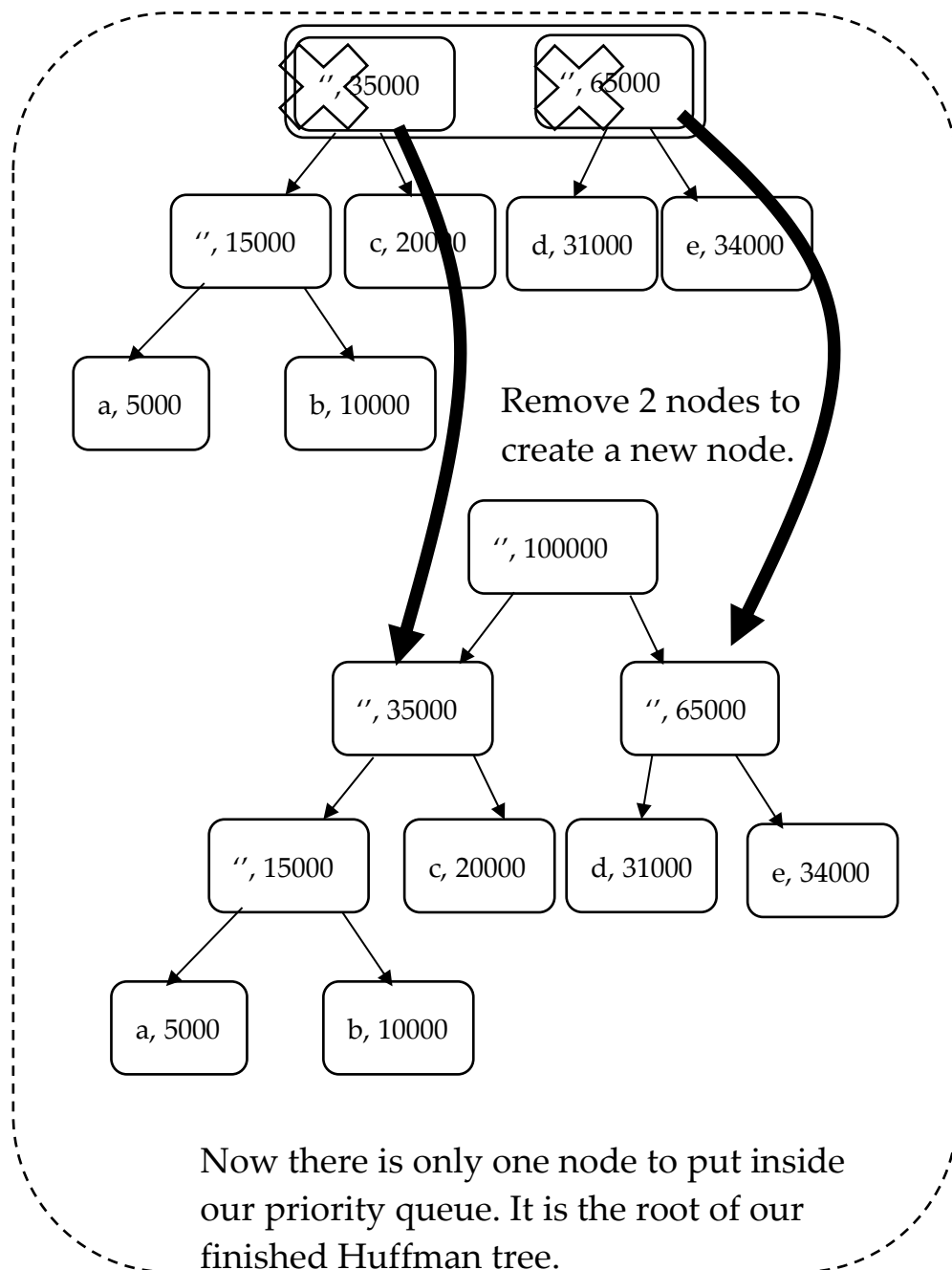


Figure 9-19: Example of Huffman tree creation (fourth iteration).

The code for Huffman tree construction is left as an exercise for readers.

Exercises

1. Define a new class for a node of a Huffman tree (let us call it class *HuffmanNode*). Then write a method that creates a Huffman tree from a given heap. The method returns a *HuffmanNode*.
2. Let your data (to be stored in a heap) be of type:

```
public class Student {
    String name; //name
    int mark; // score
    public Student(String n, int m){
        name = n; mark = m;
    }
}
```

Let class *StudentHeap* extend from class *Heap* (but students with more marked are popped first). Currently, *StudentHeap* does not have any method. Write the following methods in class *StudentHeap* :

- ***public Heap mergeHeap(Heap secondHeap)***: This method combines *secondHeap* of Student with our heap and return a new heap with all data from both heaps (for each data, a new copy of it must be created before being put in a new heap). **The array mData of this and secondHeap must not change.** What is the worst-case runtime of your code?
- ***public static boolean isAHeap(Heap h)***: This method tests heap of Student and returns true if

students with more marks are organized to be popped before students with less marks, i.e. a student has more score than all students below him/her in the heap (Assume that *h.size()* is correct but the data inside the heap's array may not be ordered correctly according to the definition of Heap).

- *public void changemark(String name, int newMark)*: Change the mark of any one student. Our student heap then must still preserve its heap properties.
3. we want to compress a text file that stores only alphabet a, b, c, d, e. where the frequency of each alphabet is as follows:
- a: 370
 - b: 80
 - c: 60
 - d: 150
 - e: 30

When putting these data inside a heap, the popping sequence of the heap (from left to right) is:

(e: 30), (c: 60), (b: 80), (d: 150), (a: 370)

Assume that you always use the first value taken from the heap as left branch of your construction of Huffman tree, and the second value from the heap as right branch, **draw a popping sequence** and **a partial Huffman tree**

for each step of removing 2 smallest-frequency values from the heap, until the Huffman tree is complete. Then after you have a complete Huffman tree, **write down a bit string that represents each alphabet.**

4. A heap of integer (smallest number is most important) is implemented using tree structure that has parent link.

```
class HeapNode{
    int data;
    HeapNode left, right, parent;

    public HeapNode(int v){
        data =v;
        left = null;
        right = null;
        parent = null;
    }
}

class Heap{
    HeapNode root;
}
```

A queue that can store *HeapNode* object is also available, with the following methods:

- *public Queue()*: constructor that creates an empty queue.
- *public void insertLast(HeapNode n)*: this method puts node *n* at the back of the queue.
- *public HeapNode removeFirst()*: this method removes the node stored in front of the queue. It returns the removed node.
- *public boolean isEmpty()*: this method returns true if the queue is empty, otherwise it returns false.

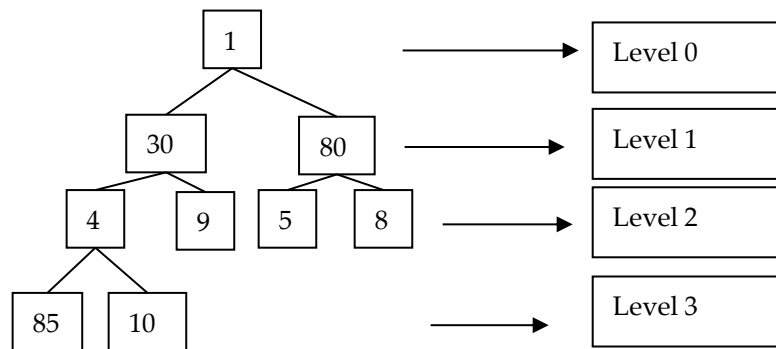
- *public HeapNode front()*: this method returns the node stored in front of the queue, without changing the queue.

Write code for the following methods of the above

Heap:

- *public void percolateUp(HeapNode n)*. This method moves the value stored in *n* up the tree of our Heap until the tree becomes heap again (the method is used to fix the tree after adding a new data).
 - *public void add(int v)*. This method adds new data, *v*, into our heap, then arranges the heap so that it retains all properties of heap.
 - *public int pop()*. This method removes the node that contains the smallest value from the heap, arranges the heap so that it retains all properties of heap, and returns the integer inside the removed node.
5. For class *Heap* in this chapter, write method *public void removeValue(Object value)*. This method removes specified value stored in our array implementation. The array after the removal must still have the quality of heap.
 6. Assume that our *Heap* contains at least 3 elements and the smallest value is the most important value. Write code for method *public void removeSecond()*. This method removes the value next to the smallest value from the heap. After the method finishes, the heap must still have all the qualities of heap.

7. For class *Heap* defined by array in this chapter, write method `public int calculateMaxIndex()`. This method returns the index of the maximum value in our array.
8. A min-max heap of integer looks like the following:



Even levels are like min heap (small value is more important). Odd levels are like max heap (greater value is more important).

A member in an even level must have smaller value than the value in its direct parent. For example, 8 and 5 are less than 80. Similarly, a member in an odd level must have larger value than the value in its direct parent. For example, 30 and 8 are greater than 4.

Write code for method `void add(int newnumber)` of this new data structure. This method adds a new number to the min-max heap. When the addition finishes, the heap must still be a min-max heap.

Chapter 10 : AVL Tree

For a binary search tree, the more it looks like a perfectly balanced tree, the less time for searching, since the tree height will be low. But we have no control over how a binary search tree will look like. It might look like a linked list, which gives us the worst possible search time.

An AVL tree (AVL is an abbreviation of Adelson-Velskii and Landis) is a binary search tree that has a rule controlling its height. The rule is as follows:

For each node, n , in an AVL tree, let the height of its left subtree be h_L , and the height of its right subtree be h_R . Then $|h_L - h_R| \leq 1$.

In other words, the difference of height between a left and right subtree must never go beyond 1. This means we are trying to make our tree as flat as possible prevent parts of the tree forming a linked list.

Examples of AVL trees are shown in Figure 10-1. Non-AVL Trees are shown in Figure 10-2. The nodes that fail the height condition are marked.

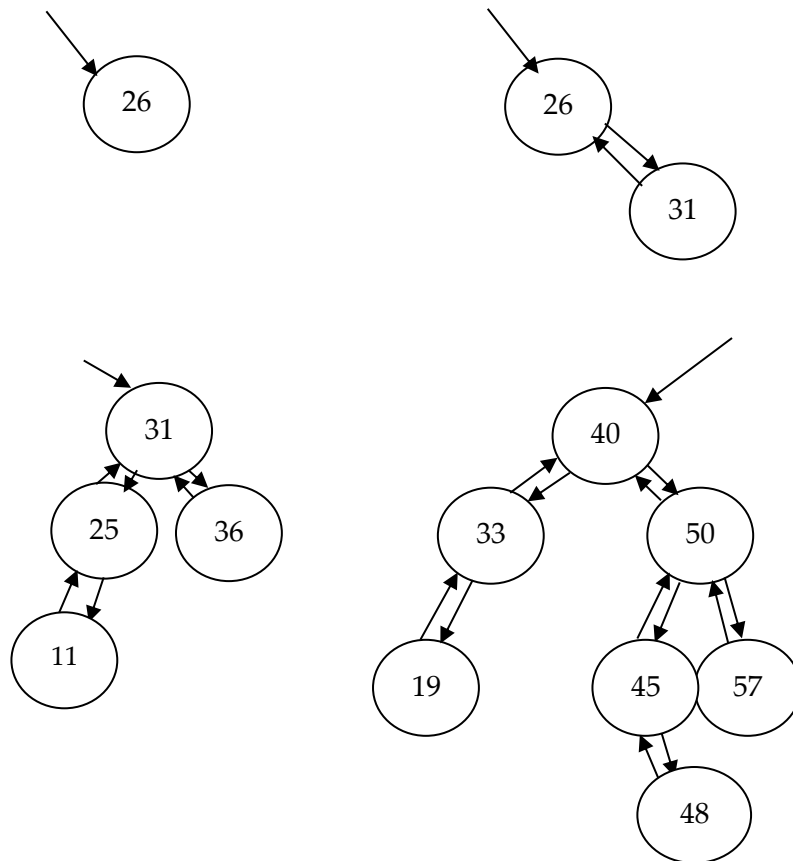


Figure 10-1: Examples of AVL trees.

With this condition, we are certain that our tree height is still in terms of $\log_2 n$.

The problem is how we can maintain this height condition after adding/removing data. What we actually need to do is: we add/remove just like what we do in binary search tree, but we **rebalance** the tree after the change.

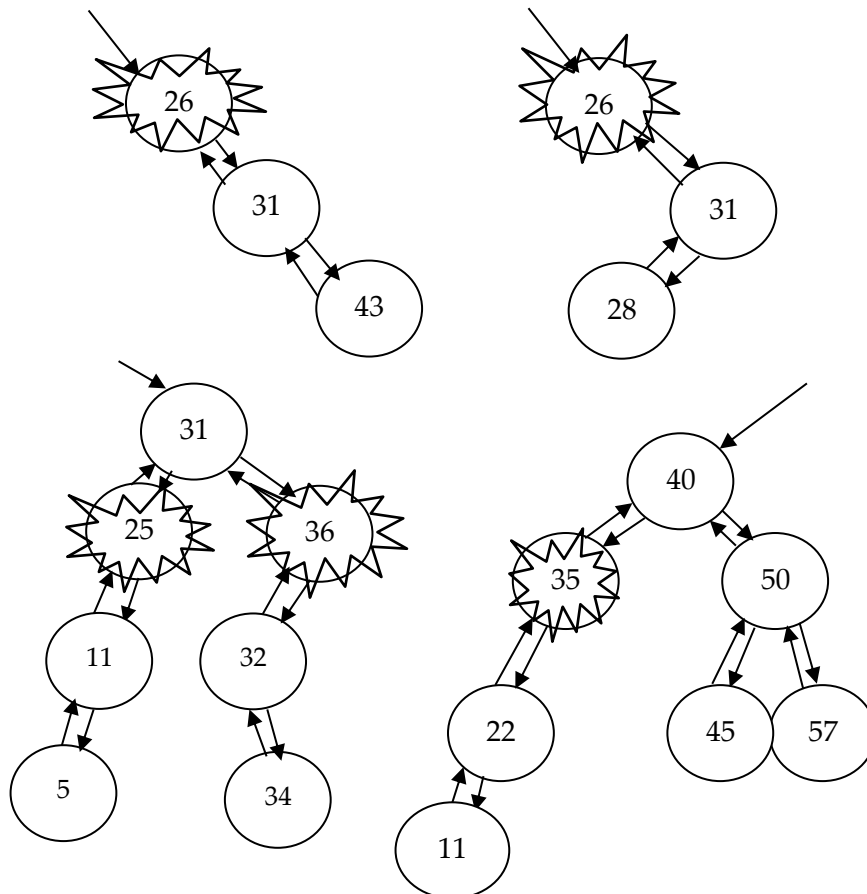


Figure 10-2: Examples of non-AVL Trees.

Rebalancing the Tree

There are only 4 possible cases that a node can become non-AVL after adding/removing data from/to an existing AVL tree. We will go through each case.

1. A node is heavy to its left (its $h_L - h_R = 2$, becoming non-AVL), and its left subtree is also heavy to its left (its $h_L - h_R = 1$).

This part of our tree can be made AVL again by rotating the left subtree up, as shown in Figure 10-3. The method that we will implement for this fix will be called *rotateLeftChild*. Rotating a node up a tree is not as difficult as readers may think. We just need to change pointers so that the part of our tree that needs fixing changes accordingly.

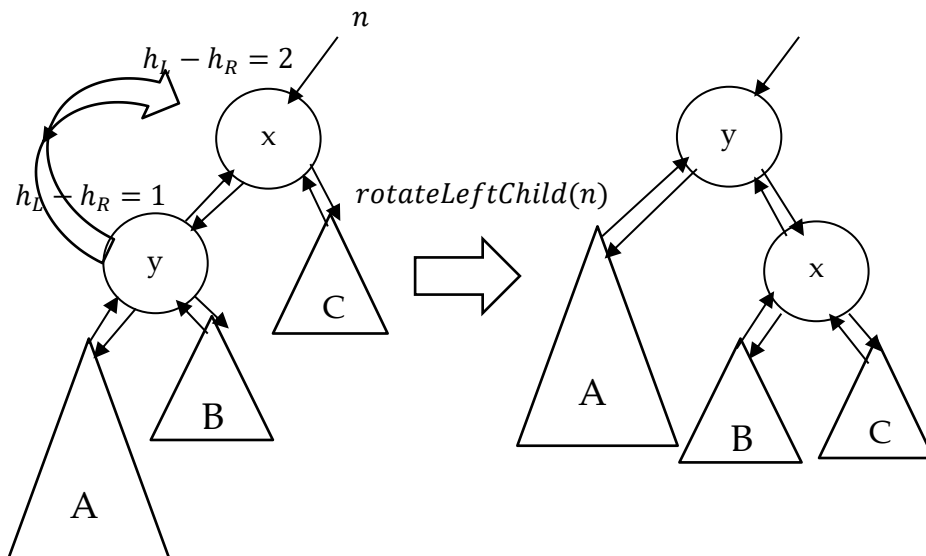


Figure 10-3: Rebalance, 1st possible case.

2. A node is heavy to its right (its $h_L - h_R = -2$, becoming non-AVL), and its right subtree is also heavy to its right (its $h_L - h_R = -1$). This part of our tree can be made AVL again by rotating the right subtree up, as shown in Figure 10-4. The method that we will implement for this fix will be called *rotateRightChild*.

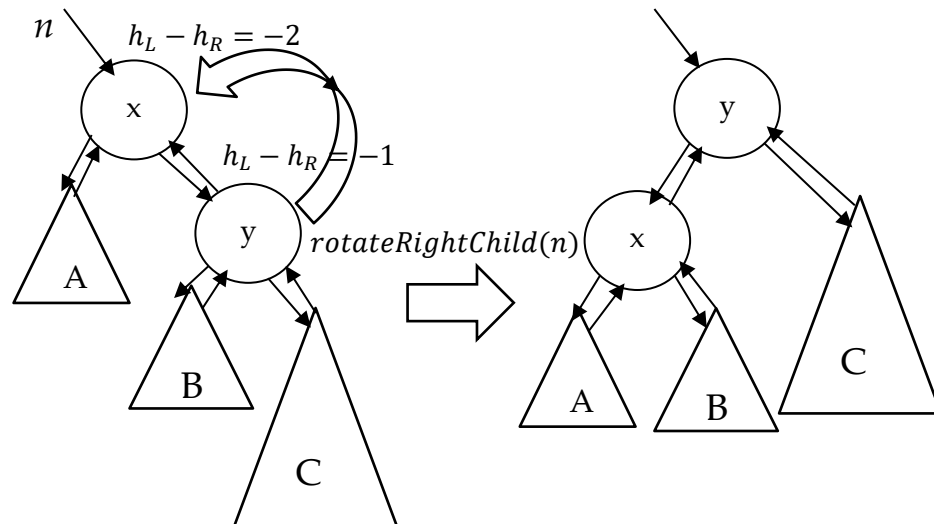
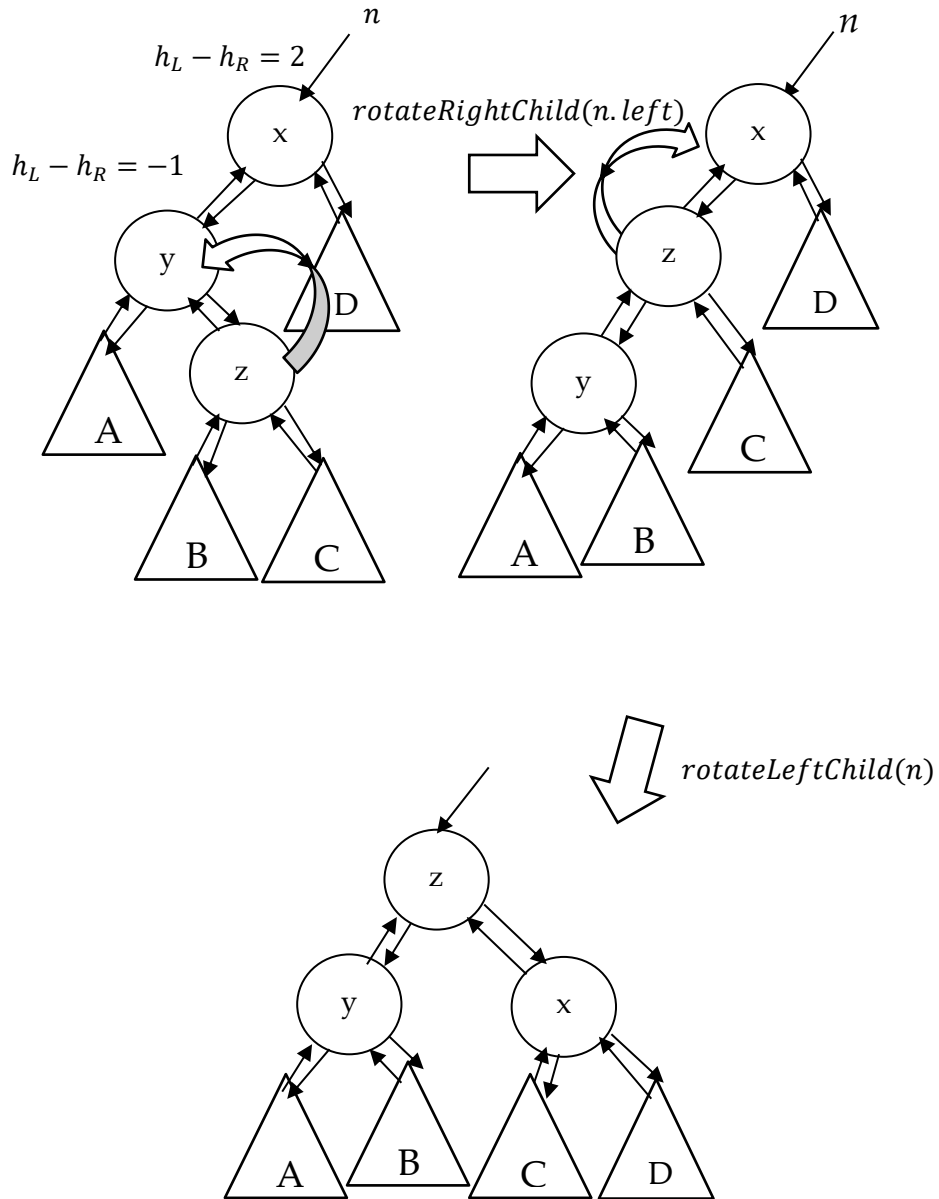


Figure 10-4: Rebalance, 2nd possible case.

3. A node is heavy to its left (its $h_L - h_R = 2$, becoming non-AVL), and its left subtree is heavy to its right (its $h_L - h_R = -1$). A rotation that fix this case is shown in Figure 10-5. One $rotateLeftChild(n)$ will not solve the problem because the longest branch (connecting to the node that stores z) does not move up even one level. For this case, the rotation must be done twice. First, $rotateRightChild(n.left)$, then $rotateLeftChild(n)$.

Figure 10-5: Rebalance, 3rd possible case.

4. A node is heavy to its right (its $h_L - h_R = -2$, becoming non-AVL), and its right subtree is heavy to its left (its $h_L - h_R = 1$). A rotation that fix this case is shown in Figure 10-6. One *rotateRightChild(n)* will not solve the problem because the longest branch (connecting to the node that stores z) does not move up even one level. For this case, the rotation has to be done twice. First, *rotateLeftChild(n.right)*, then *rotateRightChild(n)*.

Implementation of AVL Tree

The implementation is very similar to class BST. But we cannot extend from BSTNode because left, right, parent will then be BSTNode.

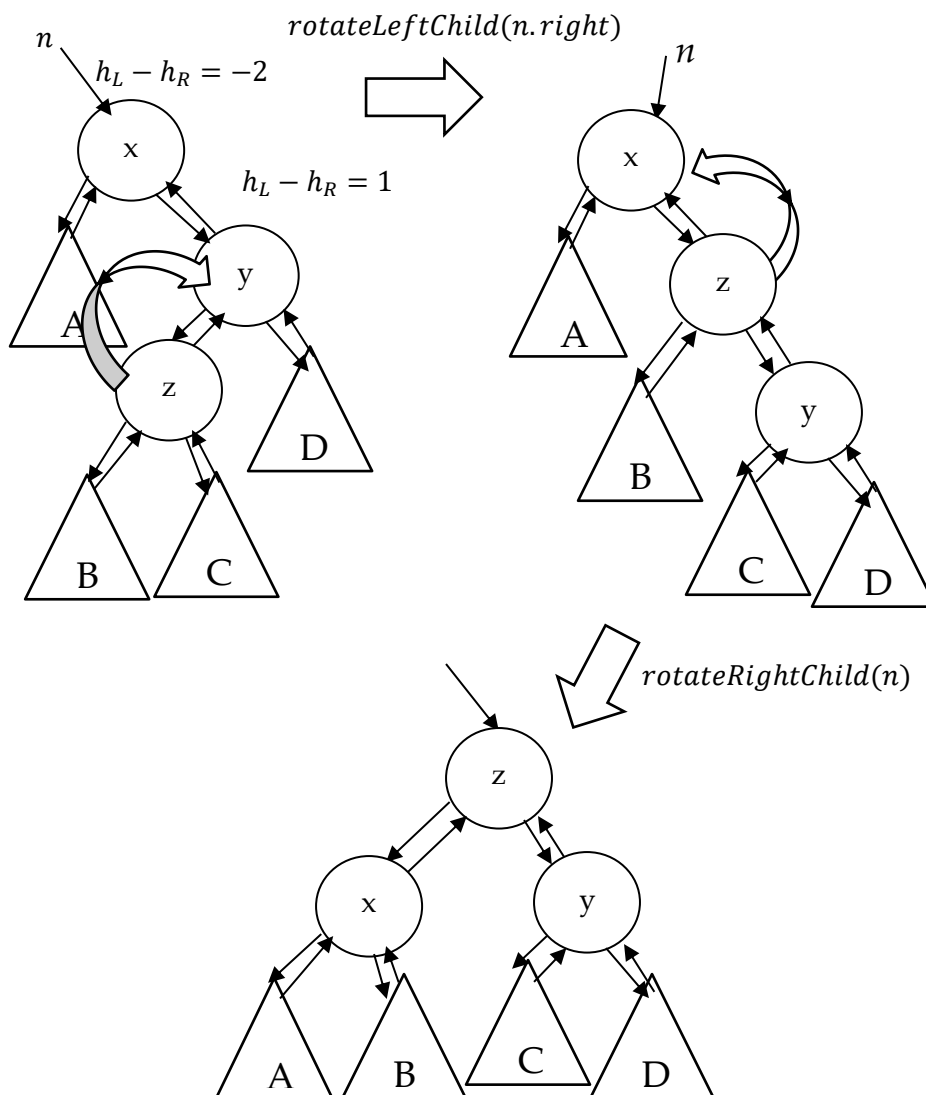
Node Implementation

First, we look at a node of an AVL tree. It is implemented by class *AVLNode*. It is almost the same as a node of a binary search tree, with one extra variable, *height*, which stores the height of the tree (counting down from the node).

The following methods are needed for a node:

- *int getHeight(AVLNode n)*: returns the height of a given node, n , by reading the value of variable *height*. We need to use a node as a parameter here because it allows the node to be *null*.

- *void updateHeight(AVLNode n)*: recalculates the height of n , and set the value of *height*, assuming that $n.left$ and $n.right$ have correct height.
- *int tiltDegree(AVLNode n)*: calculates $h_L - h_R$ of node n . This is necessary when we rebalance the tree.

Figure 10-6: Rebalance, 4th case.

The code for a node of an AVL tree is shown in Figure 10-7.

```
1:  public class AVLNode {
2:      int data;
3:      AVLNode left, right, parent;
4:      int height;
5:
6:      public AVLNode(int data) {
7:          this.data = data;
8:          left = null;
9:          right = null;
10:         parent = null;
11:         height = 0;
12:     }
13:
14:     public AVLNode(int data, AVLNode left, AVLNode
15:     right, AVLNode parent, int height) {
16:         this.data = data;
17:         this.left = left;
18:         this.right = right;
19:         this.parent = parent;
20:         this.height = height;
21:     }
22:
23:     public static int getHeight(AVLNode n) {
24:         return (n == null ? -1 : n.height);
25:     }
26:
27:     public static void updateHeight(AVLNode n) {
28:         if (n == null)
29:             return;
30:         int leftHeight = getHeight(n.left);
31:         int rightHeight = getHeight(n.right);
32:         n.height = 1 + (leftHeight < rightHeight ?
33:             rightHeight : leftHeight);
34:     }
35:
36:     public static int tiltDegree(AVLNode n) {
37:         if (n == null)
38:             return 0;
39:         return getHeight(n.left) - getHeight(n.right);
40:     }
41: }
```

Figure 10-7: Code for a node of AVL tree.

Iterator Implementation

Since an AVL tree is still a binary search tree, an iterator for an AVL tree works in the same manners as an iterator for a binary search tree. Therefore, the implementation is the same except all *BSTNode* get replaced by *AVLNode*. We call the class for this iterator *AVLTreeIterator*. The code for this iterator class is left as an exercise for readers.

Tree Implementation

An AVL tree class (*AVLTree*) contains root and size, just like our binary search tree, except the root is of type *AVLNode*.

Almost all methods remain the same as its binary search tree counterpart (but *AVLNode* replaces *BSTNode* and *AVLTreeIterator* replaces *TreeIterator*. See Figure 10-8 and Figure 10-9), except method *insert* and *remove*, where we need to rebalance the tree. There are also 3 new methods:

- *AVLNode rotateLeftChild(AVLNode n)*: this method rotates the left child of *n* (the root of our subtree we are working on) up the tree. It returns the new root of our subtree after the rotation finishes.
- *AVLNode rotateRightChild(AVLNode n)*: this method rotates the right child of *n* (the root of our subtree we are working on) up the tree. It returns the new root of our subtree after the rotation finishes.
- *AVLNode rebalance(AVLNode n)*: this method rebalances the subtree that has *n* as its root according to the 4 cases discussed in the last section

(page 367-371). It returns a new root of the subtree after the rebalance finishes.

```
1: public class AVLTree {
2:     AVLNode root;
3:     int size;
4:
5:     public AVLTree() {
6:         root = null;
7:         size = 0;
8:     }
9:
10:    public boolean isEmpty() {
11:        return size == 0;
12:    }
13:
14:    public void makeEmpty() {
15:        root = null;
16:        size = 0;
17:    }
18:
19:    public Iterator findMin() {
20:        return findMin(root);
21:    }
22:
23:    public Iterator findMin(AVLNode n) {
24:        if (n == null)
25:            return null;
26:        if (n.left == null) {
27:            Iterator itr = new AVLTreeIterator(n);
28:            return itr;
29:        }
30:        return findMin(n.left);
31:    }
32:    // continued in Figure 10-9.
```

Figure 10-8: Code for AVL Tree (part 1).

```
1: public Iterator findMax() {
2:     return findMax(root);
3: }
4:
5: public Iterator findMax(AVLNode n) {
6:     if (n == null)
7:         return null;
8:     if (n.right == null) {
9:         Iterator itr = new AVLTreeIterator(n);
10:        return itr;
11:    }
12:    return findMax(n.right);
13: }
14:
15: public Iterator find(int v) {
16:     return find(v, root);
17: }
18:
19: public Iterator find(int v, AVLNode n) {
20:     if (n == null)
21:         return null;
22:     if (v == n.data)
23:         return new AVLTreeIterator(n);
24:     if (v < n.data)
25:         return find(v, n.left);
26:     else
27:         return find(v, n.right);
28: }
29: //continued in Figure 10-13.
```

Figure 10-9: Code for AVL Tree (part 2).

Method *rotateLeftChild* updates the tree according to Figure 10-3. Method *rotateRightChild* updates the tree

according to Figure 10-4. Their codes are shown in Figure 10-10.

```
1:     public AVLNode rotateLeftChild(AVLNode n) {
2:         AVLNode l = n.left;
3:         AVLNode lr = n.left.right; // can be null
4:         n.left = lr;
5:         if (lr != null) {
6:             lr.parent = n;
7:         }
8:         l.right = n;
9:         l.parent = n.parent;
10:        n.parent = l;
11:        AVLNode.updateHeight(n);
12:        AVLNode.updateHeight(l);
13:        return l;
14:    }
15:
16:    public AVLNode rotateRightChild(AVLNode n) {
17:        AVLNode r = n.right;
18:        AVLNode rl = n.right.left; // can be null
19:        n.right = rl;
20:        if (rl != null) {
21:            rl.parent = n;
22:        }
23:        r.left = n;
24:        r.parent = n.parent;
25:        n.parent = r;
26:        AVLNode.updateHeight(n);
27:        AVLNode.updateHeight(r);
28:        return r;
29:    }
30:    // continued in Figure 10-13.
```

Figure 10-10: Code for *rotateLeftChild* and *rotateRightChild*.

How the code for method *rotateLeftChild* manipulates the tree is shown in Figure 10-11. For method *rotateRightChild*, it is the same except it is mirrored.

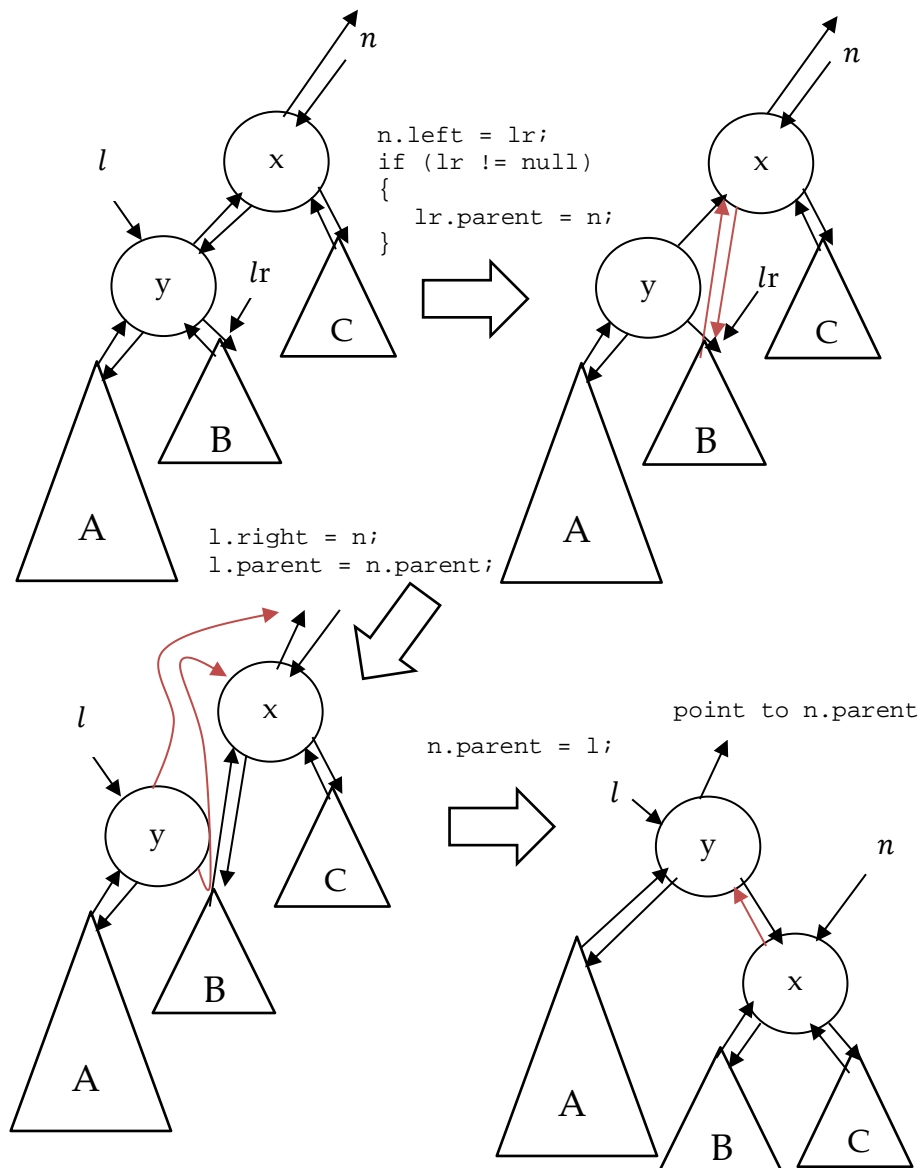


Figure 10-11: Detailed operation of method `rotateLeftChild`.

The code for method `rebalance` is shown in Figure 10-12. Method `rebalance` merges the 1st case and the 3rd case from

page 367-371 (code is shown on line 6-9 of Figure 10-12). It also merges the 2nd case and the 4th case (code is shown on line 11-14 of Figure 10-12).

```
1: public AVLNode rebalance(AVLNode n) {
2:     if (n == null)
3:         return n;
4:     int balance = AVLNode.tiltDegree(n);
5:     if (balance >= 2) {
6:         if (AVLNode.tiltDegree(n.left) <= -1)
7:             //3rd case
8:             n.left = rotateRightChild(n.left);
9:         n = rotateLeftChild(n); //1st case
10:    } else if (balance <= -2) {
11:        if (AVLNode.tiltDegree(n.right) >= 1)
12:            //4th case
13:            n.right = rotateLeftChild(n.right);
14:        n = rotateRightChild(n); //2nd case
15:    }
16:    AVLNode.updateHeight(n);
17:    return n;
18: } // continued in Figure 10-13.
```

Figure 10-12: Code for method *rebalance*.

The code for method *insert* is shown in Figure 10-13. The code is the same as its binary search tree counterpart (the recursive version) except *AVLNode* is used instead of

BSTNode, and method *rebalance* is called right at the end (on line 18) to fix any non-AVL nodes.

The code for method *remove* is shown in Figure 10-14. It works in the same way as its binary search tree counterpart. Similar to *insert*, it calls *rebalance* at the end.

```
1: public AVLNode insert(int v) {
2:     return insert(v, root, null);
3: }
4:
5: // n is the root of our subtree.
6: // this method returns the new root of the
7: // subtree after v is added to the tree.
8: public AVLNode insert(int v, AVLNode n, AVLNode
9: parent) {
10:     if (n == null) {
11:         n = new AVLNode(v, null, null, parent, 0);
12:         size++;
13:     } else if (v < n.data) {
14:         n.left = insert(v, n.left, n);
15:     } else if (v > n.data) {
16:         n.right = insert(v, n.right, n);
17:     }
18:     n = rebalance(n);
19:     return n;
20: }
21: //continued in Figure 10-14.
```

Figure 10-13: Code for method *insert* of AVL tree.

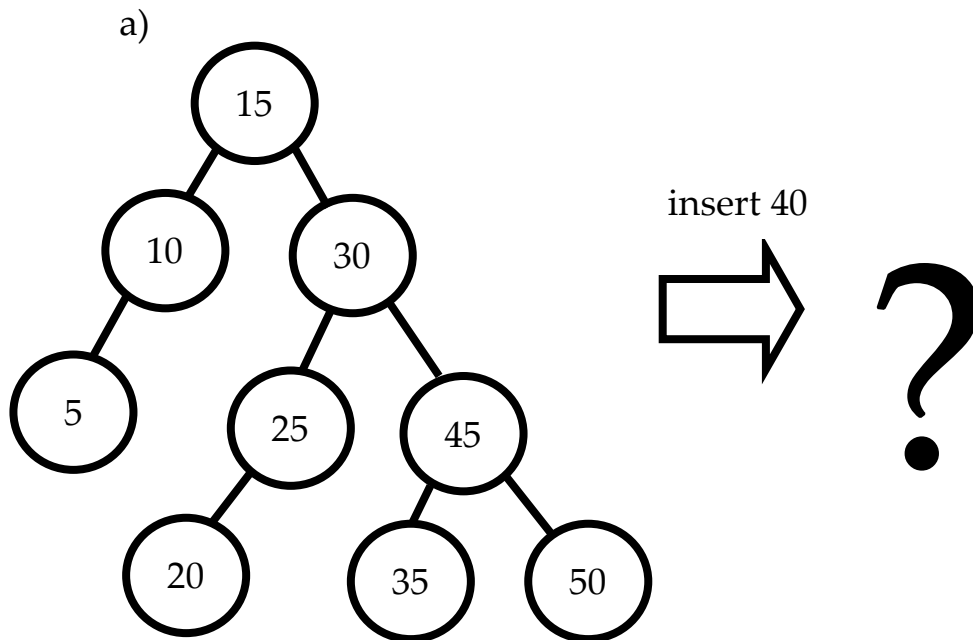
```
1: public AVLNode remove(int v) {
2:     return remove(v, root, null);
3: }
4:
5: public AVLNode remove(int v, AVLNode n, AVLNode
6: parent) {
7:     if (n == null)
8:         ; // do nothing (nothing to be removed)
9:     else if (v < n.data) {
10:        n.left = remove(v, n.left, n);
11:    } else if (v > n.data) {
12:        n.right = remove(v, n.right, n);
13:    } else {
14:        if (n.left == null && n.right == null) {
15:            n.parent = null;
16:            n = null;
17:            size--;
18:        } else if (n.left != null && n.right == null) {
19:            AVLNode n2 = n.left;
20:            n2.parent = parent;
21:            n.parent = null;
22:            n.left = null;
23:            n = n2;
24:            size--;
25:        } else if (n.right != null && n.left == null) {
26:            AVLNode n2 = n.right;
27:            n2.parent = parent;
28:            n.parent = null;
29:            n.right = null;
30:            n = n2;
31:            size--;
32:        } else {
33:            AVLTreeIterator i = (AVLTreeIterator)
34:                findMin(n.right);
35:            int minInRight = i.currentNode.data;
36:            n.data = minInRight;
37:            n.right = remove(minInRight, n.right, n);
38:        }
39:    }
40:    n = rebalance(n);
41:    return n;
42: }
43: // class AVLTree ends here.
```

Figure 10-14: Code for method *remove* of AVL tree.

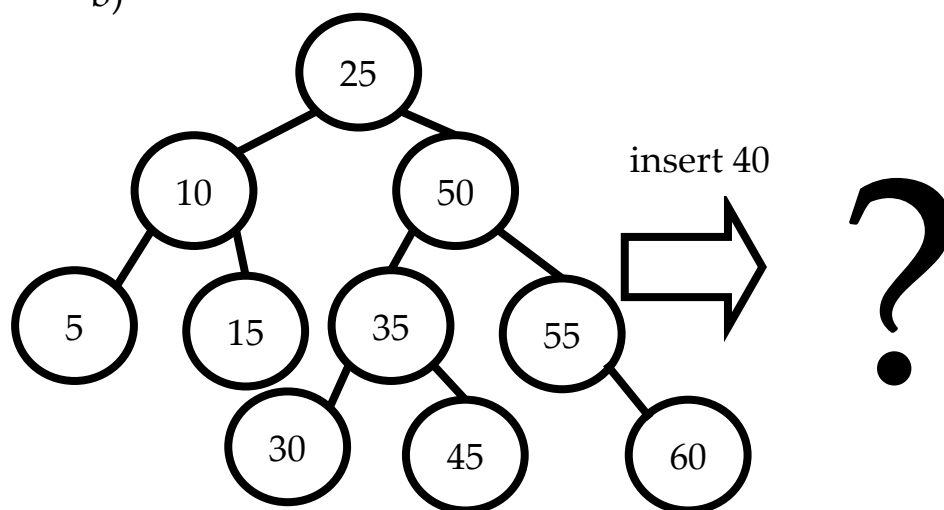
This is all we need for implementing an AVL tree. Please revise your knowledge with the exercises.

Exercises

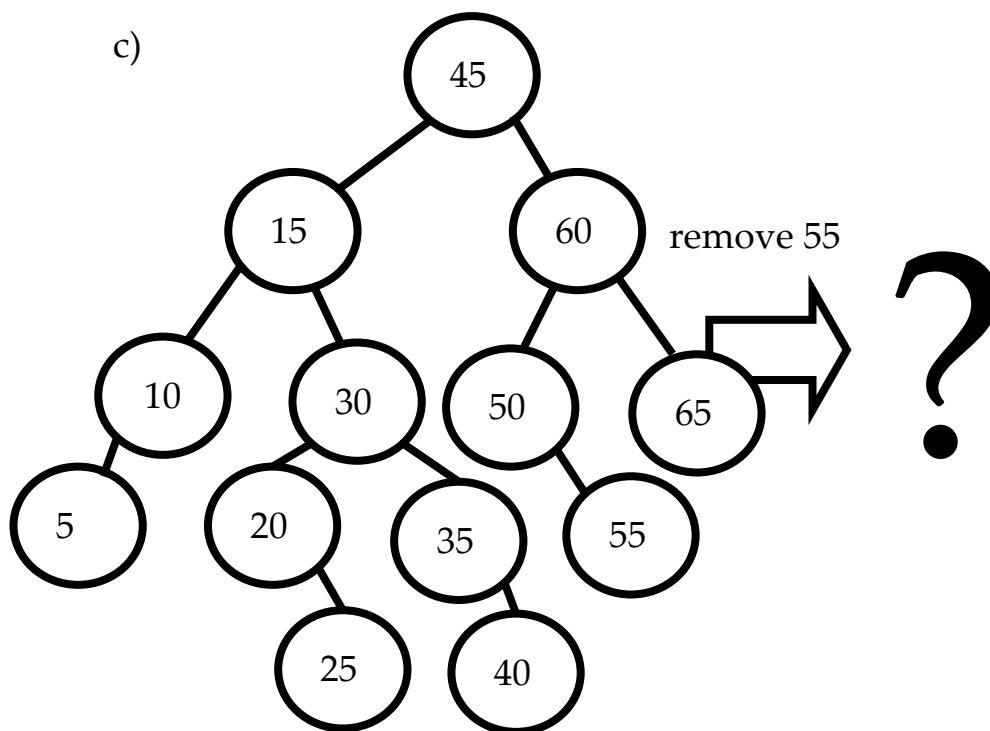
1. Draw the following AVL trees after a value is inserted or deleted. If a double rotation is used, draw each step of the rotation separately.

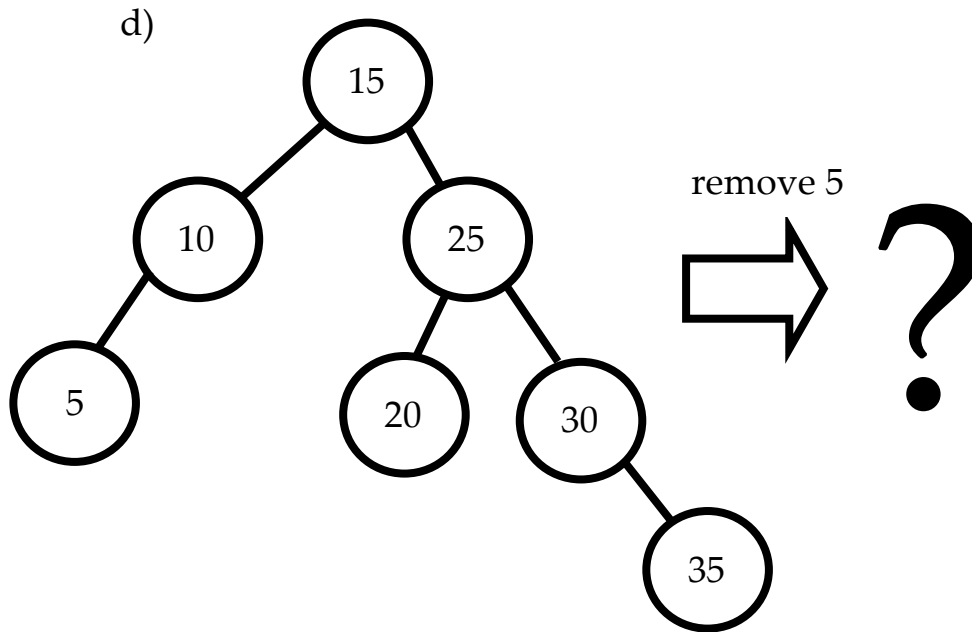


b)



c)





2. For class *BSTRecursive* in chapter 6, write code for method:

public BSTNode moveFurthestDown(BSTNode n).

This method regards n as the root of a subtree we are interested in. It rotates data in n down the longest path possible from n until that data is in a leaf. Note that the longest path changes for each rotation. The method returns a new root of this subtree.

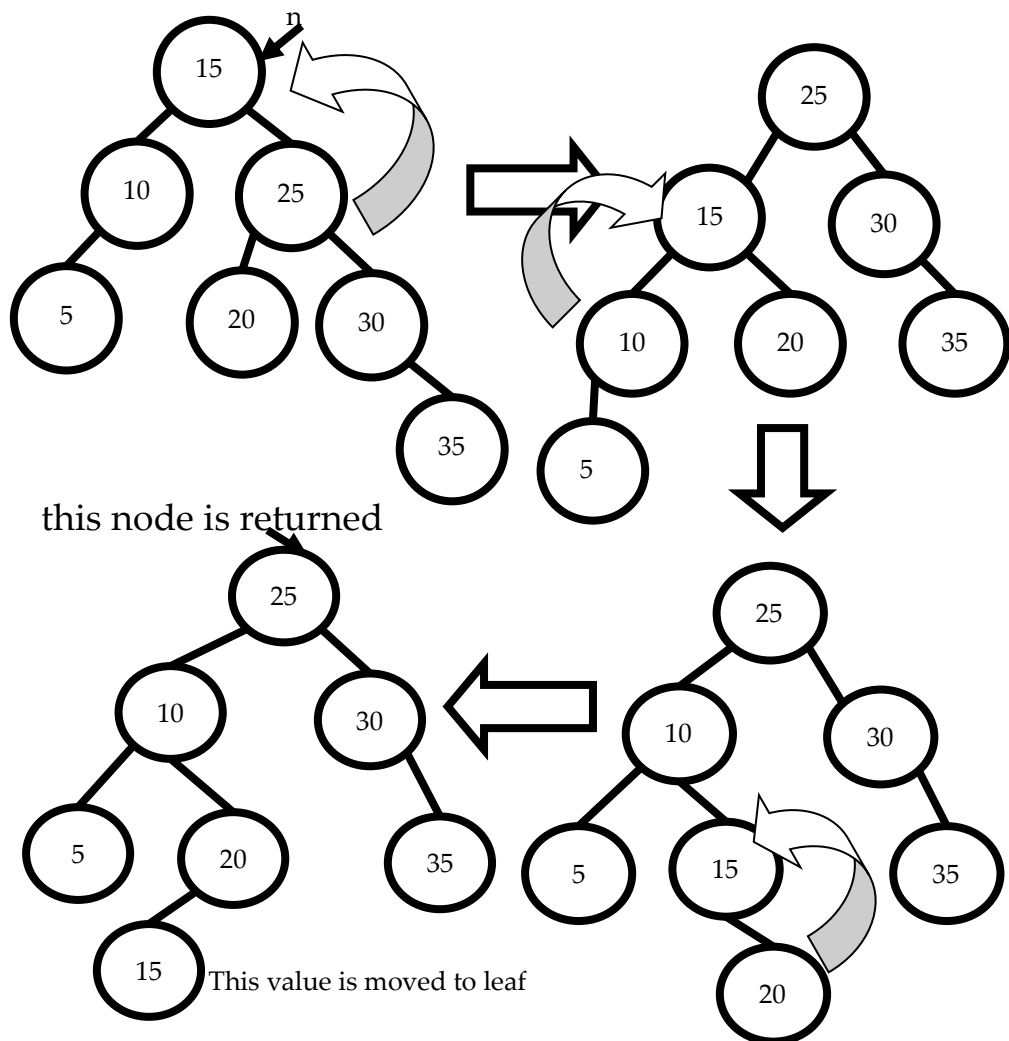
Hint:

- You can call *public BSTNode rotateRightChild(BSTNode n)* and *public BSTNode rotateLeftChild(BSTNode n)*. Both

methods work just like they do in an AVL tree.

- You can call *public int max(int a, int b)*. This method returns the maximum value among value *a* and *b*.
- A null node has its height equals to -1.

An example is shown below

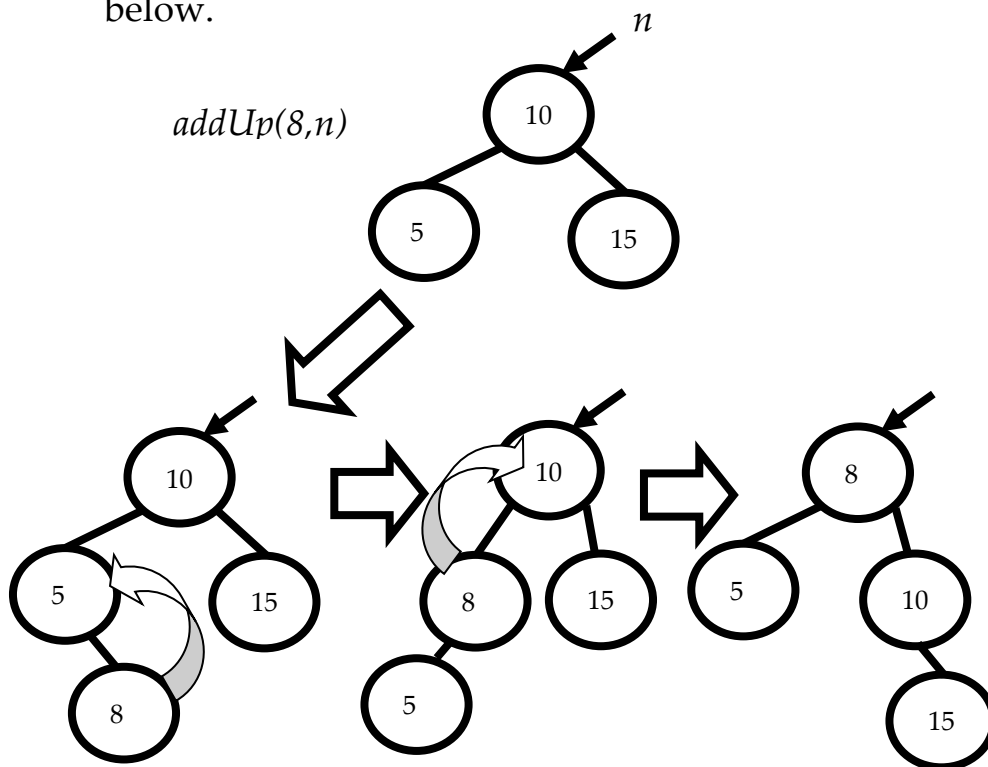


3. For class *AVLTree*, write code for method:
public boolean isAVL(AVLNode n)

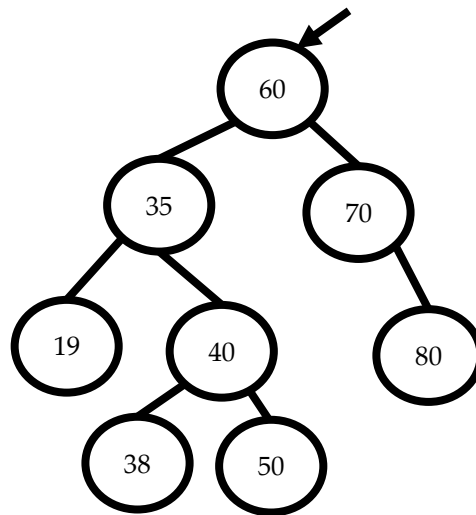
This method tests to see whether node n and all nodes below it satisfy the structural requirement of AVL tree.

4. For class *BST*, write code for method:
public BSTNode addUP(int num, BSTNode n)

This method adds num to part of the tree that has n as its root. After num is added, it must be at the root of that part of the tree (use rotation to move a newly added number up the tree). An example is shown below.



5. Draw, step by step, what happens when we add 10, 15, 20, 18, 35, 19 to an empty AVL tree.
6. An AVL Tree looks like:



- Draw, step by step, what happens when we delete 80.
7. For class *AVLTree*, write code for method:
public AVLTree merge(BST t)

This method combines all data in our AVL tree and in a binary search tree, *t*. It returns an AVL tree that has all data from both trees as a result.

Bibliography

- Bailey, D. (2002). *Java Structures: Data Structures in Java for the Principled Programmer (2nd Edition)*. McGraw-Hill.
- Burnette, E. (2005). *Eclipse IDE Pocket Guide: Using the Full-Featured IDE*. O'Reilly Media.
- Collins, W. J. (2011). *Data Structures and the Java Collections Framework (3rd Edition)*. Wiley.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms (3rd Edition)*. The MIT Press.
- cplusplus.com. (n.d.). *c++ Language Tutorial*. Retrieved 2018, from <http://www.cplusplus.com/doc/>
- Deitel, H. M., & Deitel, P. J. (2007). *Java How to Program (7th Edition)*. Prentice Hall.
- Drozdek, A. (2000). *Data Structures and Algorithms in Java*. Course Technology.
- Feldman, M. B. (1987). *Data Structures with Modula-2*. Prentice-Hall.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Java (6th Edition)*. Wiley.
- Horstmann, C. S. (2007). *Big Java (3rd Edition)*. Wiley.
- Horstmann, C. S. (2016). *Core Java Volume I--Fundamentals (10th Edition)*. Prentice Hall.
- Lemay, L., & Cadenhead, R. (2002). *Sam's Teach Yourself Java 2 in 21 Days (3rd Edition)*. Pearson Education.
- Oracle. (n.d.). *The Java Tutorial*. Retrieved 2018, from <https://docs.oracle.com/javase/tutorial/>
- Rosen, K. H. (2011). *Discrete Mathematics and Its Applications (7th Edition)*. McGraw-Hill Education.
- Sedgewick, R. (1983). *Algorithms*. Addison-Wesley.
- Sedgewick, R. (2002). *Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3rd Edition)*. Addison-Wesley Professional.
- Tutorialspoint. (n.d.). *c++ Tutorial*. Retrieved 2018, from <https://www.tutorialspoint.com/cplusplus/>
- Weiss, M. A. (2011). *Data Structures and Algorithm Analysis in Java*. Pearson.

Zakhour, S. B., Hommel, S., Royal, J., Rabinovitch, I., Risser, T., & Hoeber, M. (2006). *The Java Tutorial: A Short Course on the Basics (4th Edition)*. Addison-Wesley Professional.

สมชาย ประสิทธิ์จตุระกุล. (2552). *โครงสร้างข้อมูลฉบับวางจาวา*. สำนักพิมพ์แห่งจุฬาลงกรณ์มหาวิทยาลัย.

Index

A

Algorithm Analysis, 11
 analysis of algorithms, 6
 arithmetic calculations, 120
 array, 1, 3
 Asymptotic Notation, 20
 asymptotic runtime, 6
AVL tree, 7, 365
 AVLNode, 371
 getHeight, 371
 height, 371
 tiltDegree, 372
 updateHeight, 372
 AVLTree, 374
 insert, 379
 rebalance, 374, 378
 remove, 380
 rotateLeftChild, 374, 377
 rotateRightChild, 374, 377
 iterator, 374
 rebalance, 366
 rotateLeftChild, 368
 rotateRightChild, 368
 rotating, 368

B

Big-O, 21
 Big-Omega, 44
 Big-Theta, 20
 binary search, 29
 binary search tree, 3, 7, 204
 Breadth-First Tree Traversal, 244
 BST, 218
 find, 219
 findMin, 218
 insert, 220
 remove, 224
 BSTNode, 207
 BSTRecursive, 233
 find, 235
 findMin, 233
 insert, 235

 remove, 239
 Inorder, 242
 node, 206
 node marking, 208
 Postorder, 242
 Preorder, 241
 Recursive Tree Traversal, 241
 Treeliterator, 209
 hasNext, 210
 hasPrevious, 214
 next, 214
 previous, 216
 set, 216
binary tree, 7, 195
 ancestor, 196
 child, 196
 complete, 199
 depth of node, 196
 full, 197
 height, 197
 height of node, 196
 leaf, 196
 levels, 197
 parent, 196
 perfectly balanced, 197
 root, 196
 subtree, 196
 bubble sort, 295
 bucket sort, 324

C

complexity, 7
 Conditional Operations, 26
 asymptotic runtime, 26
 Consecutive Operations, 25
 asymptotic runtime, 26

D

data compression, 350
 dequeue, 155
 double-ended queue, 172
 insertFirst, 173, 176
 linked list implementation, 177

- insertFirst, 179
 - removeLast, 177
 - removeLast, 173, 175
 - doubly-link list
 - DListNode, 81
 - doubly-linked list, 80
 - CDLinkedList, 85
 - append, 97
 - find, 87
 - findKth, 88, 167
 - findPrevious, 92, 94, 97
 - head, 97
 - insert, 89, 170
 - isEmpty, 85, 87
 - makeEmpty, 85, 86
 - remove, 92, 95
 - removeAt, 97, 169
 - size, 85
 - tail, 97
 - circular, 80, 84
 - DListIterator, 84
 - hasPrevious, 82
 - next, 82
 - previous, 82
 - DListNode, 80
 - iterator, 82
 - previousNode, 80
 - doubly-linked list
 - CDLinkedList
 - garbage collector, 95
- E**
- enqueued, 155
- F**
- FIFO, 155
 - first in, first out, 155
- G**
- growth rate**, 14, 16, 18
- H**
- hash function, 257
 - SepChaining
 - DEFAULT_SIZE, 265
 - hash table**, 7, 255
 - double hashing, 281
 - DoubleHashing, 285
 - add, 287
 - find, 285
 - rehash, 288
 - remove, 289
 - lazy deletion, 275
 - linear probing, 271
 - load factor, 269
 - open addressing, 271
 - OpenAddressing, 282
 - primary clustering, 276
 - quadratic probing, 277
 - secondary clustering, 278
 - separate chaining, 262, 263
 - SepChaining, 264
 - add, 267
 - currentSize, 265
 - find, 266
 - hash, 266
 - lists, 265
 - MAXLOAD, 265
 - rehash, 267
 - remove, 269
 - hashCode*, 263
 - heap, 4, 7, 338
 - Huffman tree, 354, 355
- I**
- infix to postfix, 123
 - insertion sort, 299
 - isPrime, 266
- L**
- lazy deletion, 275
 - LIFO, 109
 - linked list, 3, 58
 - header node, 61
 - insert, 60
 - iterator, 64
 - hasNext, 64
 - next, 64
 - set, 64
 - iterator interface, 65
 - LinkedList, 67
 - append, 78
 - find, 68
 - findKth, 68

- findPrevious, 72
- head, 76
- insert, 70
- remove, 72
- tail, 76
- ListIterator, 66
 - hasNext(), 66
 - next(), 66
 - set(int value), 66
- ListNode, 63, 71
- node, 58
- node implementation, 62
- node marking, 63
- remove, 59
- Linked list**, 6
- list, 49
 - array, 49
 - append, 57
 - find, 50
 - findKth, 51
 - head, 55
 - insert, 51
 - remove, 53
 - tail, 57
 - operations, 49
- Little-O, 44
- Little-Omega, 45
- Logarithmic asymptotic runtime, 33
- Logarithmic Form, 28
 - greatest common divisor, 34
 - running time, 32

M

- median of 3, 313
- merge sort, 301

N

- Nested Loop, 22
 - asymptotic runtime, 24
 - exit conditions, 24
 - running time, 23
- nextPrime, 266
- null pointer, 61

P

- postfix, 119
- priority queue**, 7, 331

- data compression, 350
- heap, 338
 - complete binary tree, 338
 - pop, 345
- Heap, 340
 - add, 341
 - isEmpty, 341
 - percolateDown, 349
 - pop, 346
 - size, 341
 - top, 344
- Huffman tree, 354, 356
- implementations, 333
- linked list implementation, 334
- percolate down, 346
- percolate up, 341
- PQDLinkedList, 334
 - add, 337
 - compare, 334
 - isEmpty, 334
 - pop, 337
 - size, 334
 - top, 336

Q

- queue**, 6, 155
 - array implementation, 158
 - back, 162
 - front, 158
 - insertLast, 159, 161
 - isEmpty, 158
 - isFull, 158
 - removeFirst, 158, 161
 - size, 158
 - back, 155
 - constructors, 162
 - double-ended queue, 172
 - front, 155
 - insertLast, 156
 - isEmpty, 156
 - isFull, 156
 - MyQueue, 157
 - QueueArray, 162
 - back, 164
 - front, 164
 - insertLast, 165
 - makeEmpty, 164
 - QueueLinkedList, 167
 - back, 169

- front, 167
- insertLast, 170
- isEmpty, 167
- isFull, 167
- removeFirst, 169
- size, 167
- removeFirst, 156
- size, 156
- quick sort, 310
 - median of 3, 313
 - implementation, 316
 - partitioning, 311, 313
 - pivot, 311

R

- radix sort, 180, 184, 186
- recursive program, 27
 - asymptotic runtime, 28
- representative statement, 11, 14
- return address, 115
- right associative operators, 133
- running time, 11
- runtime, 41
 - average case, 41
 - best-case, 41
 - worst-case, 41

S

- selection sort, 295, 297
- skip list, 100, 101
- sorting, 293
 - bubble sort, 293
 - bucket sort, 324
 - insertion sort, 299
 - merge sort, 301
 - quick sort, 310
 - average case, 321
 - best case, 320
 - implementation, 316
 - median of 3, 313
 - partitioning, 311, 313
 - pivot, 311
 - worst-case runtime, 319

- selection sort, 295
- sorting algorithms, 7
- sparse table, 97
- speed, 11
- stack**, 6, 109
 - bracket pairing, 110
 - infix to postfix, 123, 132
 - bracket, 127
 - brackets, 126
 - priority values, 128, 132
 - right associative operators, 126, 133
 - isEmpty, 136
 - isFull, 136
 - makeEmpty, 136
 - method calls, 114
 - pop, 109, 136
 - postfix, 119
 - push, 109, 136
 - stack frame, 114
 - StackArray, 137
 - currentSize, 138
 - isEmpty, 138
 - isFull, 138
 - makeEmpty, 138
 - pop, 139
 - push, 140
 - theArray, 138
 - top, 138
 - StackLinkedList, 142
 - isEmpty, 143
 - isFull, 143
 - makeEmpty, 143
 - pop, 144
 - push, 145
 - top, 144
 - top, 109, 136

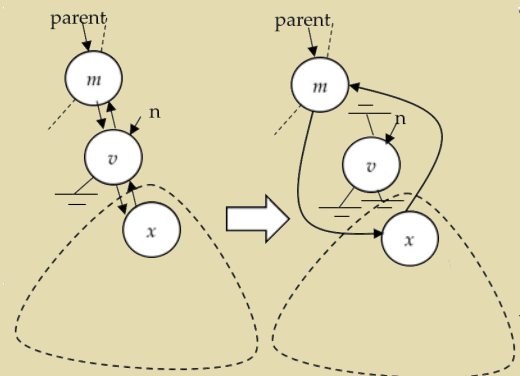
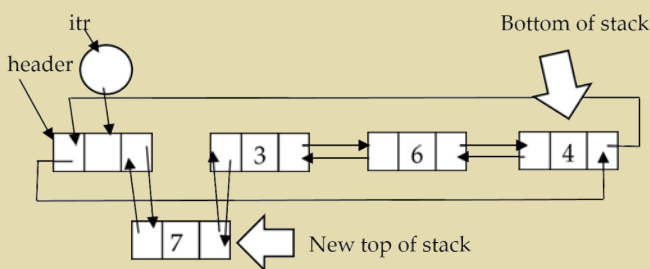
T

- tree, 3

U

- unit of time, 12

This book is intended for students who have learned the basics of programming and now want to expand their knowledge into data structures and algorithms. Intended for all levels of students, the book focuses on the most basic data structures and algorithms, with lots of illustrated examples.



maxindex reset
to 0, then
updated to 0. Sorted
portion

