เอกสารอ้างอิง

ภาษาไทย

สัมพันธ์ ระรื่นรมย์, การแบ่งคำไทยด้วยพจนานุกรม. โครงงานวิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย, 1991.

ภาษาต่างประเทศ

Appel, Andrew W., Jacobson, Guy J. The world's fastest Scrabble program. CACM 31 (May 1988):572-578.

Bach, Maurice J. The Design of the unix operating system. New Jersey:Prentice-Hall, 1986.

Deitel, Harvey M. An Introduction to Operation Systems. New York:Addison-Wesley, 1990.

Fredkin, Edward. Trie memory. CACM 3 (September 1960):490-499.

Liang, Franklin M. Word hy-phen-a-tion by com-pu-ter. doctoral thesis. Stanford University, 1983.

Loomis, Mary E.S. Data management and file structuress. New Jersey:Prentice-Hall, 1989.

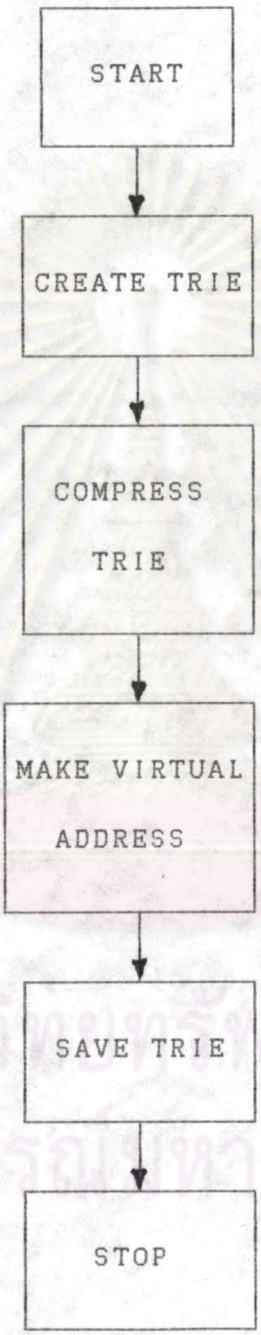
Quarterman, J.S. Silberschatz A. and Peterson,J.L. memory Management . Computer Survey 17 (December 1985):380-384.

Tanenbaum, Andrew S. Operating Systems design and implementation. New Jersey:Prentice-Hall, 1987.

ภาคผนวก

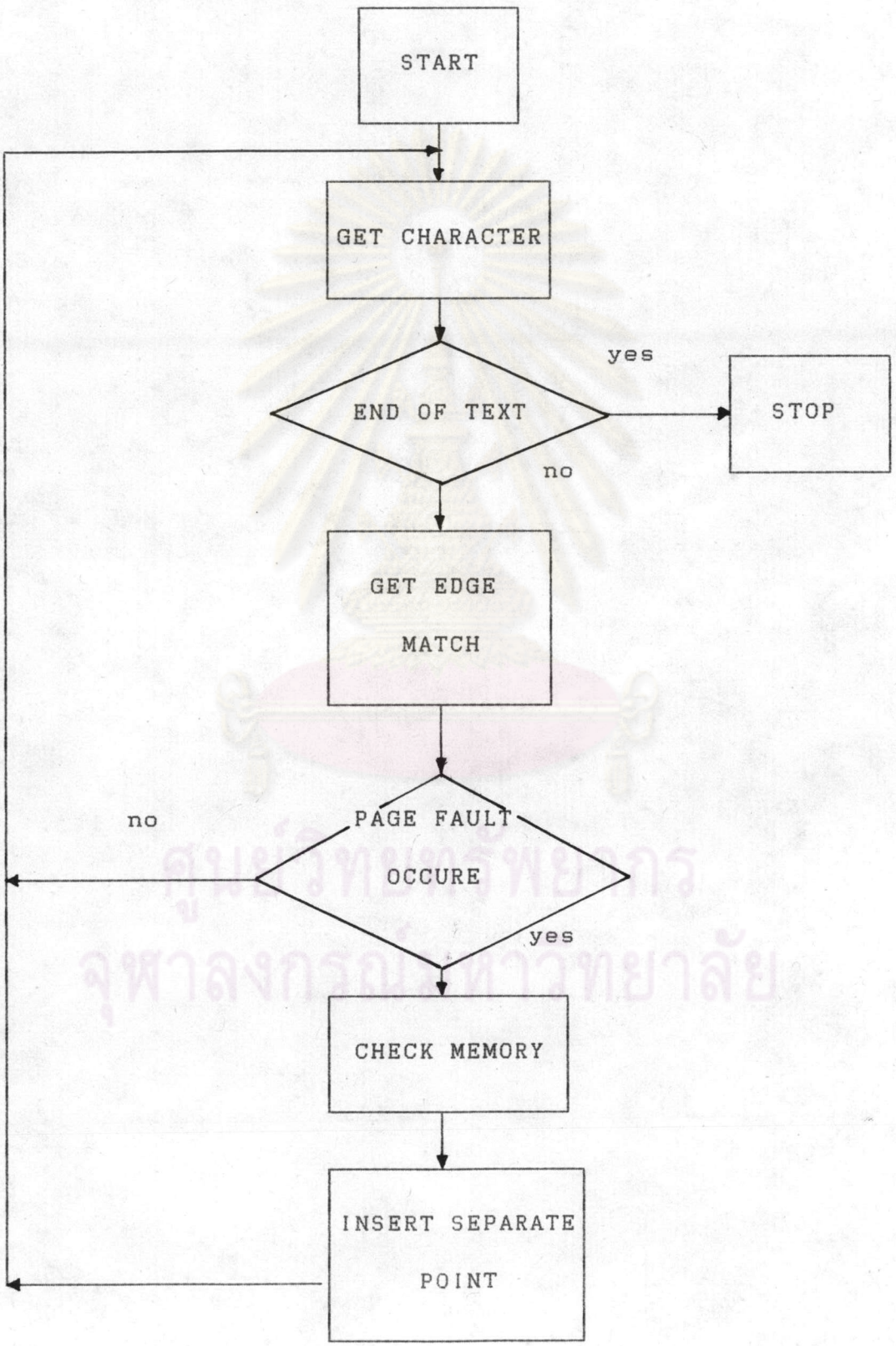
ภาคผนวก ก

ผังงานการสร้างพจนานุกรมเสมือน



ภาคผนวก ข

ผังงานการตัดคำภาษาไทย



โปรแกรมการสร้างพจนานุกรมเสมือน

```
/**
Trie  builder  :  Load  a word-list file and use it to construct a
                  Trie.
**/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <alloc.h>
#include <dir.h>
#include <process.h>
#include <conio.h>
#include <ctype.h>
#include "tctype.h"
#include "types.h"
#include "tsd.h"
#include "ltrie.h"

/* maximum number of edge */
#define MAX_EDGE60000u
#define MAX_EDGE_PER_NODE60
#define MAX_LEVEL40
```

```
word TmpNode[MAX_LEVEL][MAX_EDGE_PER_NODE];/* use in saveNode */
nat2 TmpLink[MAX_LEVEL][MAX_EDGE_PER_NODE];/* use in saveNode */

/*  NODETBL_SIZE, size of the hash table of nodes, must be a prime
*/
#define NODETBL_SIZE17317u
#define SAVETBL_SIZE20000
#define SIZE30000
#define REBUILD_SIZE60000

typedef struct St_map{
     Edge  Edg_map;
     nat2  page;
     nat2  offset;
     }St_map;

L_EdgePtr LnTrie;
int4 NumWord;
int4 NumByte;
nat2 NumEdge;/* number of edges in Trie */
nat2 NumNode;/* number of nodes in Trie */
int Height;/* height of Trie */
char Coding;/* character coding mode 6-bit, 7-bit or 12-bit */
word (*EncodeChar)(tchar **p);
void (*DecodeChar)(tchar **p, word chr);
nat2 huge* NodeTbl;/* hash table use in CompressTrie */
Edge Node;/* use in saveNode */
```

```
int SizeMatrix[MAX_LEVEL][MAX_EDGE_PER_NODE];/* use in LookTrie */
int SingleMatrix[3][MAX_LEVEL];
nat2 NumCompress;
int4 NumProbe;
int4 NumSearch;
Sv_Edge huge* Save;
int all_edge_no = 0;
nat2 NUM_PAGE = 0;
int f_no; /* file no of out.tri */

FILE *outfile;
FILE *fm;
FILE *mapfile;

char *def_extension( char *name, const char *ext )
{
     char *p=name;

     while( *p != 0 && *p != '.' ) p++; /* find extension
                                           position */
     if( *p == '.' ) return( name );
     *p++ = '.'; strcpy( p, ext ); /* add default extension */
     return( name );
}

word EncodeChar6(tchar **p)
{
```

```
        return *(*p)++;
}
/* table of combine-vowel */
tchar  combTbl[]  =  {  0x80,  0, 0, 0x84, 0x89, 0x8d, 0x91, 0x95,
                        0x9A };
word EncodeChar7(tchar **p)
{
      tchar c=*(*p)++;
      tchar c2;

      if(!isuprlwrvowel(c)) return c;
      c2=**p;
      if(!istonalgaran(c2)) return c;
      (*p)++;
      return combTbl[c-ั'']+c2-'';
}

word  envowelTbl[]  =  { 1<<7, 0, 0, 2<<7, 3<<7, 4<<7, 5<<7, 6<<7,
                         7<<7 };
word entonalTbl[] = { 1<<10, 2<<10, 3<<10, 4<<10, 5<<10, 6<<10 };
word EncodeChar12(tchar **p)
{
      word c=(*(*p)++)&0x7F;
      tchar c2=**p;

      if(isuprlwrvowel(c2)) {
      c|=envowelTbl[c2-ั'']; c2=*++(*p);
```

```
    }
    if(istonalgaran(c2) || c2=='ู') {
    c|=entonalTbl[c2-'ู']; (*p)++;
    }
    return c;
}

void DecodeChar6(tchar **p, word chr)
{
    *(*p)++=chr;
}
tchar  decombTbl[]  =
   { 0xD1, 0xD1, 0xD1, 0xD1, 0xD4, 0xD4, 0xD4, 0xD4,
     0xD4, 0xD5, 0xD5, 0xD5, 0xD5, 0xD6, 0xD6, 0xD6,
     0xD6, 0xD7, 0xD7, 0xD7, 0xD7, 0xD8, 0xD8, 0xD8,
     0xD8, 0xD8, 0xD9, 0xD9, 0xD9, 0xD9, 0xD9 };

void DecodeChar7(tchar **p, word chr)
{
    tchar c;
    if( !(chr>=0x80 && chr<=0x9E) ) *(*p)++ = chr;
    else {
        c=decombTbl[chr-0x80];
        *(*p)++=c;
        *(*p)++= '' + chr-combTbl[c-'ั'];
    }
}
```

```
tchar  devowelTbl[]  =  {  0,  0xD1, 0xD4, 0xD5, 0xD6, 0xD7, 0xD8,
                           0xD9 };
tchar detonalTbl[] = { 0, 0xE7, 0xE8, 0xE9, 0xEA, 0xEB, 0xEC };
void DecodeChar12(tchar **p, word chr)
{
     *(*p)++ = (chr&0x7F) | 0x80;
     chr>>=7;
     if( (chr&0x7) != 0 ) *(*p)++ = devowelTbl[chr&0x7];
     chr>>=3;
     if( (chr&0x7) != 0 ) *(*p)++ = detonalTbl[chr&0x7];
}

Edge trieread(triepage,trieoffset)
nat2 triepage;
nat2 trieoffset;
{
     nat4  addr, curpos;
     int numread;
     Edge Edge_trie;

     addr  =  (long)triepage  * (long)PAGESIZE +
              trieoffset*sizeof(Edge);
     fseek(outfile, addr, 0);
     if ((curpos = ftell(outfile)) == -1L)
     {
          perror("tell failed");
```

```
    }
    numread = fread(&Edge_trie, sizeof(Edge), 1, outfile);
    if (numread  < 1)
    {
        exit(1);
    }
    return(Edge_trie);
}
nat2 NUM_WORD=0;
void mapwrite(map_tbl)

MAP  map_tbl;
{
    long  addr, curpos;
    int wrt;
    int i;

    if(NUM_WORD == 0) fseek(mapfile, 4L, 0);
    if ((curpos = ftell(mapfile)) == -1L)
    {
        perror("tell failed");
    }
    wrt = fwrite(&map_tbl,sizeof(MAP),1,mapfile);
    if (wrt != 1)
    {
        printf(" Write ERROR\n ");
        exit(1);
```

```
    }
    return ;
}
MAP MapTbl;
tchar ps[40];
void mapTried(L_EdgePtr ep, St_map Edge_map, int l)
/* i=link l=no of char in word*/
{
    tchar *p;
    nat2  address,temp1,temp2;
    nat2 k;
    St_map E_map;
    nat2 i,m;
    nat2 num;
    MAP Map;
    L_EdgePtr te;

    for(;;) {
        temp1 = ((Edge_map.page << 8) & 0xFF00);
        temp2 = Edge_map.offset & 0x00FF;
        i = temp1 | temp2;
        MapTbl.link[l] = i;
        p=&ps[l];
        DecodeChar(&p, DicToTIS(L_CharOf(Edge_map.Edg_map)));
        if(IsTerminal(Edge_map.Edg_map))
        {
            *p=0;
```

```
        MapTbl.no = l+1;
        MapTbl.num = NUM_WORD + 1;
        for(k=0;k<=l;k++)
        {
             MapTbl.chr[k] = ps[k];
        }
        mapwrite(MapTbl);
        NUM_WORD++;
    }
    address = LinkOf(Edge_map.Edg_map);
    if(address != NIL)
    {
        E_map.page = ((address >> 8) & 0X00FF);
        E_map.offset = address & 0x00FF;
        E_map.Edg_map = trieread(E_map.page, E_map.offset);
        mapTried(L_EdgeAt(LnTrie,ep->link), E_map,
                 (int)(p-ps));
    }
    if(IsLastEdge(Edge_map.Edg_map))
    {
        break;
    }
    Edge_map.offset++;
    if(Edge_map.offset == EdgePerPage)
    {
        Edge_map.page++;
        Edge_map.offset = 0;
```

```
            }
            Edge_map.Edg_map    =  trieread(Edge_map.page,
                                                Edge_map.offset);
        }
    }

void MapTrie(FILE *fm)
/*
print the entire Trie
*/
{
    St_map Edge_map;
    nat2 num;
    mapfile=fm;
    Edge_map.page = 0;
    Edge_map.offset = 0;
    Edge_map.Edg_map = trieread(ROOT,ROOT);
    mapTried(L_EdgeAt(LnTrie,ROOT), Edge_map, 0);
    fseek(mapfile, 0L, 0);
    fwrite(&NUM_WORD, sizeof(nat2), 1, mapfile);
    NUM_PAGE++;
    fwrite(&NUM_PAGE, sizeof(int), 1, mapfile);
}

void printTried(L_EdgePtr ep, int l)
{
    tchar *p;
```

```
        for(;;) {
                p=&ps[1];
                DecodeChar(&p, L_CharOf(ep->chr));
                if(L_IsTerminal(ep->chr))
                {
                        *p=0;
                        fprintf(outfile, "%s\n", ps);
                }
                if(ep->link!=NIL)
                {
                        printTried(L_EdgeAt(LnTrie,ep->link), (int)(p-ps));
                }

                if(ep->next==NIL) break;
                ep=L_EdgeAt(LnTrie,ep->next);
        }
}

void PrintTrie(FILE *f)
/*
print the entire Trie
*/
{
        outfile=f;
        printTried(L_EdgeAt(LnTrie,ROOT), 0);
}
```

```
void clearVisit(L_EdgePtr ep)
{
      ep->chr&=~L_VISITED;
      for(;;) {
            if(ep->link!=NIL) clearVisit(L_EdgeAt(LnTrie,ep->link));
            if(ep->next==NIL) break;
            ep=L_EdgeAt(LnTrie,ep->next);
      }
}

void ClearVisited(void)
{
      clearVisit(L_EdgeAt(LnTrie,ROOT));
}

void countSingle(L_EdgePtr ep)
/*
count the single-edge node
*/
{
      int n=0;

      if(ep->chr&L_VISITED) return;
      ep->chr|=L_VISITED;

   if(ep->next==NIL) { /* found a single-edge node */
      for(;;) {
```

```
        if(ep->chr&L_MERGED)/* multi-parent single-edge node */
        {
            SingleMatrix[2][n]++;
            if(ep->link!=NIL)
                countSingle(L_EdgeAt(LnTrie,ep->link));
            return;
        }
        n++;
        if(ep->link==NIL) {
            SingleMatrix[0][n]++; return;
        }
        ep=L_EdgeAt(LnTrie,ep->link);
        ep->chr!=L_VISITED;
        if(ep->next!=NIL) {
            SingleMatrix[1][n]++; break;
        }
    }
  }
  for(;;) {
    if(ep->link!=NIL) countSingle(L_EdgeAt(LnTrie,ep->link));
    if(ep->next==NIL) break;
    ep=L_EdgeAt(LnTrie,ep->next);
  }
}
```

```
void lookTried(L_EdgePtr ep, int l)
/*
get the degree of node statistic
*/
{
      int n=0;

      if(ep->chr&L_VISITED) {
            ep->chr|=L_MERGED;
            return;
      }
      ep->chr|=L_VISITED;
      NumNode++;
      for(;;) {
            n++;
            if(ep->link!=NIL)    lookTried(L_EdgeAt(LnTrie,ep->link),
                                                        l+1);
            if(ep->next==NIL) break;
            ep=L_EdgeAt(LnTrie,ep->next);
      }
      NumEdge+=n;
      SizeMatrix[l][n]++;
}
```

```
void LookTrie(FILE *f) . .
/*
construct  the size matrix of Trie SizeMatrix[l][n]=number of node
at level l which has n edges.
*/
{
     int n, l;
     nat2 sum;

     NumEdge=0;
     NumNode=0;

     for(l=0; l<MAX_LEVEL; l++) {
          for(n=0; n<MAX_EDGE_PER_NODE; n++) SizeMatrix[l][n]=0;
     }
     for(l=0;  l<MAX_LEVEL;  l++) for(n=0; n<=2; n++)
     SingleMatrix[n][l]=0;
     fprintf(f, "L\\N : S\t");
     for(n=1;  n<MAX_EDGE_PER_NODE;  n++)
          fprintf(f, "%5d ", n);
     fprintf(f, "\n\n");
     lookTried(L_EdgeAt(LnTrie,ROOT), 0);
     for(l=0; l<MAX_LEVEL; l++) {
          sum=0;
          for(n=1;  n<MAX_EDGE_PER_NODE;  n++)
               sum+=SizeMatrix[l][n];
          fprintf(f, "%2d:%4u\t", l, sum);
```

```
        for(n=1; n<MAX_EDGE_PER_NODE; n++) {
            if(SizeMatrix[l][n]>0)
                fprintf(f,    "%5d    ", SizeMatrix[l][n]);
            else fprintf(f, "       ");
        }
        putc('\n', f);
}

fprintf(f, "\nS %5u\t", NumNode);
for(n=1; n<MAX_EDGE_PER_NODE; n++) {
        sum=0;
        for(l=0; l<MAX_LEVEL; l++) sum+=SizeMatrix[l][n];
        if(sum>0) fprintf(f, "%5u ", sum);
        else fprintf(f, "       ");
}
fprintf(f, "\n\%\t");
for(n=1; n<MAX_EDGE_PER_NODE; n++) {
        sum=0;
        for(l=0; l<MAX_LEVEL; l++) sum+=SizeMatrix[l][n];
        if(sum>0) fprintf(f, "%5.2f ", sum*100.0/NumNode);
        else fprintf(f, " ");
}

fprintf(f, "\n\nnumber of words : %ld\n", NumWord);
fprintf(f, "original size : %ld\n", NumByte);
fprintf(f, "Trie size : %ld\n", (long)NumEdge*sizeof(Edge));
fprintf(f, "number of edges     : %u\n", NumEdge);
```

```
    fprintf(f, "number of nodes     : %u\n", NumNode);
    fprintf(f, "height of Trie      : %d\n\n", Height);
    ClearVisited();

    fprintf(f, "TERM\t");
    for(l=0; l<MAX_LEVEL; l++) fprintf(f, "%5d ", l);
    fprintf(f, "\n\n");
    countSingle(L_EdgeAt(LnTrie,ROOT));
    for(n=0; n<=2; n++) {
        switch(n) {/* terminate with - */
            case 0 : fprintf(f, "LEAF\t"); break;/* LEAF */
            case  1  :  fprintf(f,  "MULTI\t"); break;
                                /* multi-edge node */
            case  2  :  fprintf(f, "MERGED\t"); break;
                                /* MERGED node */
        }
        for(l=0; l<MAX_LEVEL; l++) {
            if(SingleMatrix[n][l]!=0)
                fprintf(f,  "%5d  ", SingleMatrix[n][l]);
            else fprintf(f, "      ");
        }
        putc('\n', f);
    }
    ClearVisited();
}
```

```
bool AddWord(tchar *w)
/*
add a word into Trie.  preserving order in each node.
*/
{
     tchar *p=w;
     word c;
     nat2 e;
     L_Edge temp;
     L_EdgePtr ep=&temp;/* avoid dangerous pointer */
     L_EdgePtr tep;

     e= (NumEdge>0) ? ROOT : NIL;
     while( *p!=0 ) {
          c=EncodeChar(&p);
          if(e==NIL) {/* add new node with one edge */
               NumNode++;
               e=NumEdge++;
               if(NumEdge>=MAX_EDGE) {
                    ep->chr|=L_TERMINAL;
                    return FALSE;
               }
               ep->link=e;
               ep=L_EdgeAt(LnTrie,e);
               ep->chr=c; ep->link=NIL; ep->next=NIL;
          } else {
               ep=L_EdgeAt(LnTrie,e);/* search an existing node */
```

```
            if(c<L_CharOf(ep->chr)) {/* insert at head */
                e=NumEdge++;
                if(NumEdge>=MAX_EDGE) return FALSE;
                tep=L_EdgeAt(LnTrie,e);
                *tep=*ep;
                ep->chr=c; ep->link=NIL; ep->next=e;
            } else {
                while(c>L_CharOf(ep->chr)) {
                    tep=ep;
                    if(ep->next==NIL) break;
                    ep=L_EdgeAt(LnTrie,ep->next);
                }
                if(c!=L_CharOf(ep->chr)) {
                    e=NumEdge++;
                    if(NumEdge>=MAX_EDGE) return FALSE;
                    ep=L_EdgeAt(LnTrie,e);
                    ep->next=tep->next;
                    tep->next=e;
                    ep->chr=c; ep->link=NIL;
                }
            }
        }
        e=ep->link;
    }
    ep->chr!=L_TERMINAL;
    return TRUE;
}
```

```
void LoadTrie(FILE *f)
/*
Load Trie with words in a file.
*/
{
     char s[100], w[100];
     int l;
     NumByte=NumWord=NumNode=NumEdge=Height=0;
     while( fgets(s, 100, f)!=NULL ) {
          sscanf(s, "%s", w);
          l=strlen(s);
          NumWord++; NumByte+=l;
          if(l>Height) Height=l;
          if( NumWord%1000 == 0 ) printf("%ld\t%s\t", NumWord, w);
          if(!AddWord(w)) {
               printf("Trie full after %ld words\n", NumWord);
               exit(0);
          }
     }
     fclose(f);
     printf("\nnumber of words    : %ld\n", NumWord);
     printf("original size      : %ld\n", NumByte);
     printf("Trie size : %ld\n", (long)NumEdge*sizeof(Edge));
     printf("number of edges    : %u\n", NumEdge);
     printf("number of nodes    : %u\n", NumNode);
     printf("height of Trie     : %d\n", Height);
}
```

```
void saveNode(Edge node)
{
      int w;
      nat2 link,page,offset;

      f_no = fileno(outfile);
      NumNode++;
      if( NumNode%5000 == 0 ) printf("%u\t", NumNode);
      w=fwrite(&node, sizeof(Edge), 1, outfile);
      if(w!=1) { perror("error in writing node"); exit(0); }
}

void MakeVirtualAddr(word tmp_node[],nat2 tmp_link[], int n)
{
      int i;
      nat2  page_no,offset_no;
      div_t   pg_offset;
      nat2 Rebuild;

      if(pe==0)
      {
            for (i=0;i<n;i++)
            {
                  Save[i].chr = tmp_node[i];
                  pg_offset = div(tmp_link[i],EdgePerPage);
                  if(   pg_offset.quot   >   NUM_PAGE)
```

```
                NUM_PAGE    = pg_offset.quot;
            page_no = pg_offset.quot << 8;
            offset_no = pg_offset.rem;
                                    /* Store Rebuild index */
            Rebuild = page_no | offset_no;
            Save[i].link = Rebuild;
        }
    }
    else
    {
        for (i=0;i<n;i++)
        {
            Save[all_edge_no].chr = tmp_node[i];
            pg_offset = div(tmp_link[i],EdgePerPage);
            if(   pg_offset.quot   >   NUM_PAGE)
                NUM_PAGE    = pg_offset.quot;
            page_no = pg_offset.quot << 8;
            offset_no = pg_offset.rem;
                                    /* Store Rebuild index */
            Rebuild  =  page_no  |  offset_no;
            Save[all_edge_no].link = Rebuild;
            all_edge_no++;
        }
    }
}
```

```
nat2 saveTrie(nat2 e, int l)

{
     L_Edge temp1, temp2;
     L_EdgePtr ep=&temp1;/* avoid dangerous pointer */
     L_EdgePtr nep=&temp2;/* avoid dangerous pointer */
     nat2 i;
     int  n=0;
     nat2 npe;
     nat2 last_edge;

     /* Put next in queue and mark it as visited */
     if(e==NIL) return NIL;
     nep=L_EdgeAt(LnTrie,e);
     if(nep->chr&L_VISITED)  return  nep->link;
                                   /* this node is visited */
     ep=nep;
     for(i=e; i!=NIL; i=ep->next, ep=L_EdgeAt(LnTrie,i)) {
          TmpNode[l][n] = ep->chr;
          TmpLink[l][n] = saveTrie(ep->link, l+1);
          n++;
     }
     TmpNode[l][n-1]|=LAST_EDGE;
     if(e==ROOT) pe=0;
     MakeVirtualAddr(TmpNode[l],TmpLink[l], n);
     nep->chr|=L_VISITED; /* remind that we've visited this node*/
     nep->link=pe;      /* and remember the position in file */
```

```
    npe=pe; pe+=n;
    return npe;
}

void SaveTrie(FILE *f)
/*
save packed Trie to a disk file
*/
{
    nat2 i;
    L_EdgePtr ep;
    nat2 Rebuild;
    nat2 link;

    outfile=f;
    NumNode=0;

    Save=(Sv_Edge huge*)farmalloc((long)SIZE*sizeof(Sv_Edge));
    if(Save==NULL) { puts("not enough memory"); exit(0); }
    for(i=ROOT; i!=NIL; i=ep->next)
    {
        ep = L_EdgeAt(LnTrie,i);
        pe++;
    }
    all_edge_no = all_edge_no + pe;
    saveTrie(ROOT, 0);
    for(i=0;i<all_edge_no;i++)
```

```
    {
        Node= MkEdge( TISToDic(L_CharOf((Save[i].chr))),
              Save[i].link ) | ( L_IsTerminal(Save[i].chr) ?
              TERMINAL : 0 );
        if(IsLastEdge(Save[i].chr))
        {
            Node |= LAST_EDGE;
        }
        saveNode(Node);
    }
    printf("NUM_PAGE = %d\n",NUM_PAGE);
    farfree((Sv_Edge far*)Save);
    printf("\n%u nodes saved\n", NumNode);
}


nat2 hash(nat2 e, int rl)
/*
Computing hash index of the node 'e' which has rev-len 'rl'.
Return the hash index
*/
{
    L_EdgePtr ep;
    nat2 i;
    int j;
    nat4 h=0;/* the hash index */
    nat4 c, p;
```

```
    for(i=e, j=0; i!=NIL; i=ep->next, j=(j+1)&3) {
        ep=L_EdgeAt(LnTrie,i);
        c=ep->chr; p=ep->link;
        h = (h>>9) | (h<<23); /* rotate right 9 bit */
        h^=  (c<<25)  | (c>>7) | (p<<9); /* c rotate right 7 bit
              | p<<9 */
    }
    h = (h+r1) % NODETBL_SIZE;
    return (nat2)h;
}

bool nodeMatch(nat2 e1, nat2 e2)
/*
Compare two node, return TRUE if match.
The edge in every node must be byte-sorted.
*/
{
    L_EdgePtr ep1=L_EdgeAt(LnTrie,e1), ep2=L_EdgeAt(LnTrie,e2);

    while( ep1->chr==ep2->chr && ep1->link==ep2->link ) {
        e1=ep1->next; e2=ep2->next;
        if(e1==NIL) {
            if(e2==NIL) return TRUE; else return FALSE;
        }
        if(e2==NIL) return FALSE;
        ep1=L_EdgeAt(LnTrie,e1); ep2=L_EdgeAt(LnTrie,e2);
```

```
    }
    return FALSE;
}

nat2 searchNode(nat2 e)
/*
Search for node point to by 'e' in NodeTbl.
If  already  existed  the edge found, else add it into NodeTbl and
return NIL.
*/
{
    nat2 h, d=1, e2;

    NumSearch++;
    h=hash(e, 0); /* don't use rev-len now */
    for(;;) {
        NumProbe++;
        e2=NodeTbl[h];
        if(e2!=NIL) {
            if(nodeMatch(e, NodeTbl[h])) return e2;
            h=h+d; d=d+2;
            if(h>=NODETBL_SIZE) h-=NODETBL_SIZE;
            if(d==NODETBL_SIZE)  { puts("hash table overflow");
                                   exit(0); }
        } else { /* not existed */
            NodeTbl[h]=e;
            return NIL;
```

```
        }
    }
}

nat2 compressTried(nat2 e)
/*
Compress the subtrie 'e'. Return new sub-trie (or old 'e' if
can't compress)
*/
{
    nat2 i;
    L_EdgePtr ep;
    if(e==NIL) return NIL;
    for(i=e; i!=NIL; i=ep->next) { /* compress its subtries */
        ep=L_EdgeAt(LnTrie,i);
        ep->link=compressTried(ep->link);
    }
    i=searchNode(e);
    if(i==NIL) return e; /* can't compress this node */
    NumCompress++;
    if(NumCompress%1000 == 0) printf("%u\t", NumCompress);
    return i; /* can compress */
}

void CompressTrie(void)
/*
Compress the entire Trie/
```

```
*/
{
        nat2 i;

        NodeTbl=(nat2 huge*)farmalloc((long)NODETBL_SIZE*
                                sizeof(nat2));
        if(NodeTbl==NULL) { puts("not enough memory"); exit(0); }
        for(i=0; i<NODETBL_SIZE; i++) NodeTbl[i]=NIL;
                                /*   init   the  hash table */
        NumSearch=NumProbe=NumCompress=0;
        compressTried(ROOT);
        NumNode-=NumCompress;
        printf("current number of nodes\t: %u\n", NumNode);
        printf("average  number of probe\t: %4.2f\n",
                (float)NumProbe/NumSearch);
        farfree((nat2 far*)NodeTbl);
}

void main(void)
{
        char s[100], c;
        FILE *f;
        int fm_no;
        nat2 i;

        LnTrie=(L_Edge huge*)farmalloc((long)MAX_EDGE*
                        sizeof(L_Edge));
```

```
if(LnTrie==NULL) { puts("not enough memory"); return; }
chdir("\\jume\\thai\\wordlist");
printf("load word list [CANCEL][.LST]: "); gets(s);
if(*s==0) { puts("cancel"); return; }
def_extension(s, "lst");
f=fopen(s, "r");
if(f==NULL) { puts("can't open word list file"); return; }
printf("character coding : 6-bit 7-bit 12-bit [6]: ");
c=toupper(getche()); puts("");
if(c!='6' && c!='7' && c!='1') c='6';
Coding=c;
switch(Coding) {
     case '6' : EncodeChar=EncodeChar6;
                DecodeChar=DecodeChar6;
                break;
     case '7' : EncodeChar=EncodeChar7;
                DecodeChar=DecodeChar7;
                break;
     case '1' : EncodeChar=EncodeChar12;
                DecodeChar=DecodeChar12;
                break;
}
puts("loading");
LoadTrie(f);

printf("\ncompress : Yes No [Yes]: ");
c=toupper(getche()); puts("");
```

```
if(c!='N') {
        puts("compressing");
        CompressTrie();
}

printf("\nmatrix [NO]: "); gets(s);
if(*s!=0 ) {
        f=fopen(s, "w");
        if(f==NULL) { puts("can't open matrix file"); return; }
        LookTrie(f);
}

printf("\noutput [NO]: "); gets(s);
if(*s!=0) {
        f=fopen(s, "w");
        if(f==NULL) { puts("can't open output file"); return; }
        PrintTrie(f);
}
printf("\nsave Trie [NO][.TRI]: "); gets(s);
if(*s!=0) {
        def_extension(s, "tri");
        printf("name of trie is %s\n",s);
        f=fopen(s,"a+b");
        f_no = fileno(f);
        printf("  File  number  of  out.tri  in main(f) = %u\n",
                f_no);
        if(f==NULL) { puts("can't open Trie file"); return; }
```

```
        puts("saving");
        SaveTrie(f);
        fclose(f);
}

printf("\nmap Trie [NO][.tbl]: "); gets(s);
if(*s!=0) {
        def_extension(s, "tbl");
        printf("name of map trie is %s\n",s);
        fm=fopen(s,"wb");
        fm_no = fileno(fm);
        printf("  File  number  of  map.tbl in main(fm) = %u\n",
                fm_no);
        if(fm==NULL) { puts("can't open Trie file"); return; }
        printf("\nOut Trie [.tri]: "); gets(s);
        def_extension(s, "tri");
        printf("name of trie is %s\n",s);
        outfile=fopen(s,"rb");
        f_no = fileno(outfile);
        printf("  File  number  of  out.tri  in main(f) = %u\n",
                f_no);
        if(outfile==NULL)   {   puts("can't  open  Trie  file");
                                return; }
        puts("maping");
        MapTrie(fm);
        fclose(fm);
        fclose(outfile);
```

```
    }
    farfree((L_Edge far*)LnTrie);
    chdir("\\");

}
```

ภาคผนวก ง

โปรแกรมการตัดคำภาษาไทย

```
/*
thaisep.c
contain thaiSeparate() : the Thai phrase separation routine
*/
#include <stdio.h>
#include <stdlib.h>
#include <types.h>
#include <tctype.h>
#include <conio.h>
#include "tsd.h"
#define PAGE_RAM50
/* the maximum level we backtrack before we giveup */
#define MAX_BACKTRACK 2
/* use in check mode, the number of word we'll separate before we
sure that the start point is ok */
#define SATISFY_CHECK 2
#define PAGE_PER_ROW20
extern RamTblPtr RamPageTbl;
int addr_ram;

extern nat4 Num_access_dic;
```

```
extern nat4 Num_page_fault;

word GetEdgeMatch(Dictionary t,nat2 *n,tchar c,
     FILE *f,nat2 *Page_ram, nat2 *Num_page);

void check_mem(Dictionary t,nat2 *Page_ram, nat2 *Num_page);

int   ThaiSep(tchar   **sStart,  tchar  *sEnd,  Separation  sep[],
              Dictionary dic, bool checkMode , FILE *f,
              nat2 *Page_ram, nat2 *Num_page)
/**
Look for words in 's' thru 'sEnd' or until find a waste
using dictionary 'dic' and place the result in 'sep'
**/
{
     tchar *s=*sStart, c;
     nat2 temp;
     nat2 *n = &temp;/* a node in the dictionary */
     word chr,test;
     int level=0;/* current level in separation tree */
     int maxLevel=0;/* the farthest level we traverse to */
     tchar  *maxS=s;/*  the  farthest  position  in  the string we
                      traverse */
     Separation farSep[MAX_BACKTRACK+1];
                                /* room for farthest separation */
     int  numWord=0;/*  the  number  of  word (S_SEPARATION) we've
                     make */
```

```
int i;
int j;
int f_no;
bool error=FALSE;
nat2 number,link,page,offset;

sep[level].point=s;
if( s+1==sEnd || !istalpha(*(s+1)) ) {
     if(checkMode) return FALSE;
     sep[level].type=S_TSINGLE;
                /* single Thai alphabet, usaully be abbrev. */
     *sStart=s+1; return 1;
}
sep[level++].type=S_TALPHA;
              /* sentinel at head to prevent over backtrack */
farSep[0].point=s;
      /*  record  the point for the case where maxLevel==0 */

do {
     *n = ROOT;/* start at the root of the dictionary */

     for(;;) {
          c=*s++;
          c=TISToDic(c);
          chr = GetEdgeMatch(dic,n,c,f,Page_ram,Num_page);
            /* get the edge that match 'c' if there's one */
          if( c!=CharOf(chr) )
```

```
{
        break;
}

if(     IsTerminal(chr)     /*     &&     (s==sEnd     ||
                                !isunleadable(*s)) */ )
{
        sep[level].point=s;/*  may  make  a separation
                                here */
        sep[level++].type=S_CANDIDATE;
                        /*    add    the    point    as    a
                                candidate */
        if(s==sEnd || !istalpha(*s)) {/* finish */
             sep[level-1].type=S_SEPARATION;
               /*  flag  to caller their's no error */
             if(checkMode) return TRUE;
             *sStart=s; return level;
        }
}
if(s==sEnd || !istalpha(*s))
{
        break;/* str end w/o IsTerminal */
}
if(*n==NIL)
{
        break;
}
```

```
}
```

/* make a separation from the longest candidate or backtrack if needed */

```
level--;/* backtrack to the last level and see */
switch(sep[level].type) {
     case S_CANDIDATE :
       /* the longest (farthest) candidate point */
          numWord++;
          if(checkMode   &&  (numWord>=SATISFY_CHECK  ||
             level>=SATISFY_CHECK+1))
               return TRUE;/* check mode satisfy */
          sep[level].type=S_SEPARATION;
             /* promote it to be a separation point */
          s=sep[level++].point;/* move the string
                                  pointer back */
          break;
/* then loop back to begin searching the dictionary at root */
     case  S_TALPHA  :/*  found  the sentinel, mean sure
                          error */
          error=TRUE;
          break;

     case  S_SEPARATION  :/*  no  candidate  so  have to
                              backtrack */
```

```
            Show(sep+level);
            numWord--;
            if(    s>maxS|| (s==maxS && sep[level].point>
                   farSep[0].point) )
            {
                 maxS=s; maxLevel=level;
                      /* record the farthest traversal */
/* record the last (farthest) MAX_BACKTRACK+1 points */
for(i=0;  i<=MAX_BACKTRACK && level-i>=0; i++)
      farSep[i]=sep[level-i];
}
level--;/* backtrack more and search for a candidate */
while(   maxLevel-level<=MAX_BACKTRACK   &&
         sep[level].type==S_SEPARATION )
{
     level--; numWord--;
}
if(maxLevel-level>MAX_BACKTRACK||
   sep[level].type==S_TALPHA)
{
    error=TRUE;/* no more backtrack */
} else { /* so sep[level].type must be S_CANDIDATE */
     numWord++;
     sep[level].type=S_SEPARATION;/* promote */
     s=sep[level++].point;/* move pointer back */
}
break;
```

```
        }
        } while(!error);

      /* so there're some error, recover the farthest separation */
        if(checkMode) return FALSE;
        for(i=0; i<=MAX_BACKTRACK && maxLevel-i>=0; i++)
              sep[maxLevel-i]=farSep[i];
        sep[maxLevel].type=S_ERROR;
        *sStart=farSep[0].point; return maxLevel+1;
}
void display_insert(nat2 page)
{
        div_t row_col;
        nat2 row;
        nat2 col;

        gotoxy(2,20);
        textbackground(WHITE);
        textcolor(BLACK);
        cprintf("Insert page = "); /* display page to insert */
        gotoxy(16,20);
        textbackground(WHITE);
        textcolor(BLACK);
        cprintf("%u",page);

        row_col = div(page,PAGE_PER_ROW);
        row = (row_col.quot * 2) + 2;
```

```
    col = (row_col.rem * 2) + 10;
    gotoxy(col,row);
    textbackground(WHITE);
    textcolor(BLACK);
    cprintf(" r ");

}

void display_delete(nat2 k)
{

    div_t row_col;
    nat2 row;
    nat2 col;

    gotoxy(2,22);
    textbackground(WHITE);
    textcolor(BLACK);
    cprintf("delete page =    "); /* display page to delete */
    gotoxy(16,22);
    textbackground(WHITE);
     textcolor(BLACK);
    cprintf("%d",k);

    row_col = div(k,PAGE_PER_ROW);
    row = (row_col.quot * 2) + 2;
    col = (row_col.rem * 2) + 10;
```

```
        gotoxy(col,row);
        textbackground(BLACK);
        textcolor(BLACK);
        cprintf("    ");
}

word GetEdgeMatch(Dictionary t,nat2 *n,tchar c,
        FILE *f, nat2 *Page_ram, nat2 *Num_page)
{ .

        Edge Fdic;
        EdgePtr p,q;
        Edge e;
        char found=' ';
        int i,j,temp1,temp2,k,l;
        int indx,no,del_indx;
        nat2 sub_page_ram;
        long Address;
        long PageAddr;
        word chr;
        nat2 page,offset, ram_page,test_page,test_offset,test_link;
        Num_access_dic++;
        page = (( *n >> 8 ) & 0x00FF);
        offset = (*n) & 0x00FF;
        if(RamPageTbl[page].flag == 'r')
             found = 'y';
        for(k=0;k< *Num_page;k++)/* Update R_Bit */
```

```
{
        temp1 = RamPageTbl[k].R_Bit >> 1;/* shift right 1 bit */
        temp2 = temp1 & 0x7FFF;          /* make first bit to 0 */
        RamPageTbl[k].R_Bit = temp2;
}
if(found == 'y')/* Page is in ram */
{
        RamPageTbl[page].R_Bit = RamPageTbl[page].R_Bit|0x8000;
                                          /* Add 1 to first bit */
        p=&t[RamPageTbl[page].address + offset];
        while( c!=CharOf(e=*p++) && !IsLastEdge(e) );
                                            /* Find node in page */
        chr=ChrOf(e);
        *n=LinkOf(e);
        found = ' ';
        check_mem(t,Page_ram,Num_page);
        return(chr);
}
else/* Page isn't in ram */
{
        Num_page_fault++;
        page = (( *n >> 8 ) & 0x00FF);
        offset = (*n) & 0x00FF;
        display_insert(page);
        RamPageTbl[page].flag = 'r';
        RamPageTbl[page].address = (*Page_ram) * EdgePerPage;
        PageAddr    =   page;  /*  can't  use  unsinged  short  to
```

```
                                calculate */
        Address = (PageAddr * EdgePerPage) * 4;
        no = 0;
        do/* Read that page to ram */
        {
             fseek(f,Address,SEEK_SET);
             fread(&Fdic,sizeof(Edge),1,f);
             indx = (((*Page_ram) * EdgePerPage)+(no));
             t[indx] = Fdic;
             Address = Address + 4;
             no++;
         }while(no < EdgePerPage);
        (*Page_ram)++;
        p=&t[RamPageTbl[page].address + offset];
        temp1 = RamPageTbl[page].R_Bit >> 1;
                                        /* shift right  1 bit */
        temp2 = temp1 & 0x7FFF;         /* make first bit to 0 */
        temp1 = temp2 | 0x8000;         /* add 1 to first bit */
        RamPageTbl[page].R_Bit = temp1;
        while( c!=CharOf(e=*p++) && !IsLastEdge(e) );
        chr=ChrOf(e);
        *n=LinkOf(e);
        found = ' ';
        check_mem(t, Page_ram,Num_page);
        return(chr);
    }
}
```

```
void check_mem(Dictionary t,nat2 *Page_ram,nat2 *Num_page)

{
     int i, j,m;
     nat2 indx, del_indx,k,l,up_idx;
     EdgePtr q;
     nat2 number=0,temp;

     if(*Page_ram > PAGE_RAM)
     {
          for(k=2;k<*Num_page;k++)/* Find minimum r_bit &
                                     delete it */
          {
               if(RamPageTbl[k].flag == 'r')
               {
                    if(RamPageTbl[k].R_Bit < 0x3FFF)
                    {
                         display_delete(k);
                         indx = k;
                         del_indx = RamPageTbl[indx].address;
                         RamPageTbl[indx].flag = 'd';
                         RamPageTbl[indx].address = -1;
                         RamPageTbl[indx].R_Bit = 0x0000;
                         q = &t[del_indx+EdgePerPage];
                         if((del_indx)<((*Page_ram-1)*
                              EdgePerPage))
                              /* delete node that minimum r_bit */
```

```
                    {
                        temp = ((*Page_ram) * EdgePerPage)
                            - EdgePerPage;
                        for(j=del_indx;j< temp;j++)
                        {
                            t[j] = *q;
                            q++;
                        }
                        for(l=0;l<*Num_page;l++)
                        {
if((RamPageTbl[l].flag  ==  'r')  && (RamPageTbl[l].address >
    del_indx))
RamPageTbl[l].address = RamPageTbl[l].address - EdgePerPage;
                        }
                    }
                    number++;
                }
            }
        }/* for */
        *Page_ram = *Page_ram - number;
    }
}
```

ประวัติผู้เขียน

อรศรี สงวนชาติศรไกร เกิดเมื่อวันที่ 17 ธันวาคม พ.ศ. 2509 ที่ กรุงเทพมหานครฯ เป็นบุตรคนที่ 1 ในจำนวน 2 คนของครอบครัว สำเร็จการศึกษาระดับมัธยมปลาย จาก โรงเรียนสามเสนวิทยาลัย จากนั้นเข้าศึกษาระดับปริญญาตรีที่ภาควิชา สถิติ คณะ วิทยาศาสตร์ มหาวิทยาลัยเกษตรศาสตร์ สำเร็จการศึกษาเมื่อปี 2530 และศึกษาต่อระดับปริญญาโทในปี 2532 ที่ภาควิา วิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย