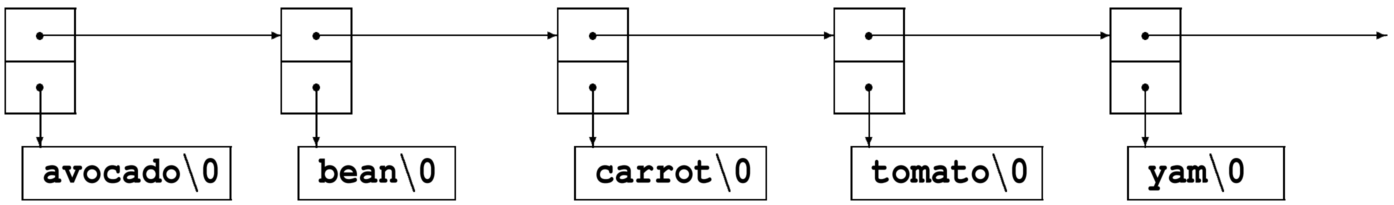
CHAPTER 19

LISTS AND TREES

A LINKED LIST OF STRINGS



lnodes.h

(Figure 19.2 – Page 473)

```
/*
 * Linked list node type and prototypes.
 */
#include <stddef.h>                    /* for NULL */
#include <stdlib.h>                    /* for malloc/free */

typedef struct listnode
{
  void *dataptr;                       /* ptr to node's data */
  struct listnode *nextptr;            /* ptr to next list node */
}
  LISTNODE;

LISTNODE *ln_create(void *dataptr, LISTNODE *nodeptr);
void ln_destroy(LISTNODE *nodeptr);
```

lnodes.c

(Figure 19.3 – Page 473)

```
/*
 * Manage linked list nodes.
 *   ln_create - create new list node.
 *   ln_destroy - free space used by existing list node.
 */
#include "lnodes.h"                          /* node type, prototypes */

LISTNODE *ln_create(void *dataptr, LISTNODE *nodeptr)
{
  LISTNODE *nptr = malloc(sizeof(LISTNODE));

  if (nptr != NULL)
  {
    nptr->dataptr = dataptr;                   /* points to data */
    nptr->nextptr = nodeptr;                   /* points to next node */
  }
  return nptr;
}

void ln_destroy(LISTNODE *nptr)
{
  free(nptr);                                /* destroy node, not data */
}
```

ptrfuncs.h

(Figure 19.4 – Page 474)

```
/*
 * Useful typedefs for pointers to functions that create, update,
 * compare, and access nodes in linked lists.
 */
#if !defined(POINTER_TYPEDEFS)
#define POINTER_TYPEDEFS

enum status {UPDATE, CREATE};     /* controls whether update function
                                     creates or updates list node */

typedef int  (*CMP_FPTR)(const void *, const void *);   /* comparison */
typedef void (*ACT_FPTR)(void *);                       /* action */
typedef void *(*UPD_FPTR)(void *, enum status);         /* update */

#endif
```

lists.h

(Figure 19.5 – Page 475)

```
/*
 * Definitions of list type and prototypes for list operations.
 */
#include "lnodes.h"                  /* list node and prototypes */
#include "ptrfuncs.h"                /* pointer to function typedefs */

typedef LISTNODE *LIST;              /* list is ptr to 1st list node */

LIST list_create(void);
void list_traverse(LIST l, ACT_FPTR action);
LIST list_insert(LIST l, void *dataptr, CMP_FPTR cmp);
void list_destroy(LIST l);
LIST list_update(LIST l, void *dataptr, CMP_FPTR cmp, UPD_FPTR upd);
```

lists1.c

(Figure 19.6 – Page 476)

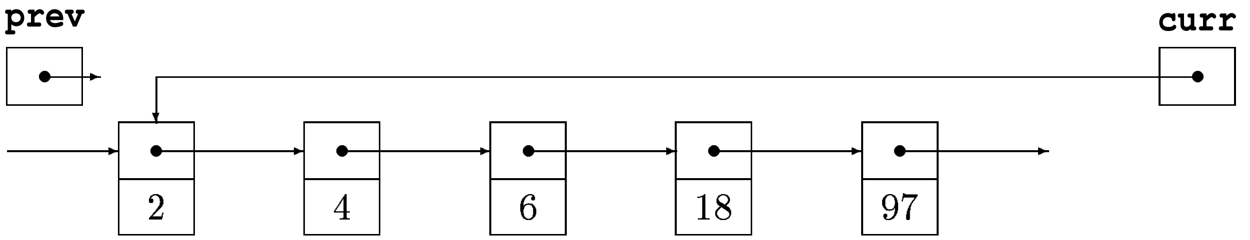
```
/*
 * Basic functions implementing list data type.
 *  list_create - create a new, empty list.
 *  list_traverse - do action on each list node's data.
 *  list_insert - insert new node in correct place in list.
 *  list_destroy - remove all list nodes.
 */
#include "lists.h"

LIST list_create(void)
{
  return NULL;
}

void list_traverse(LIST list, ACT_FPTR actfptr)
{
  LISTNODE *curr;                        /* ptr to current node */

  for (curr = list; curr != NULL; curr = curr->nextptr)
    actfptr(curr->dataptr);              /* do act to node's data */
}
```

lists1.c

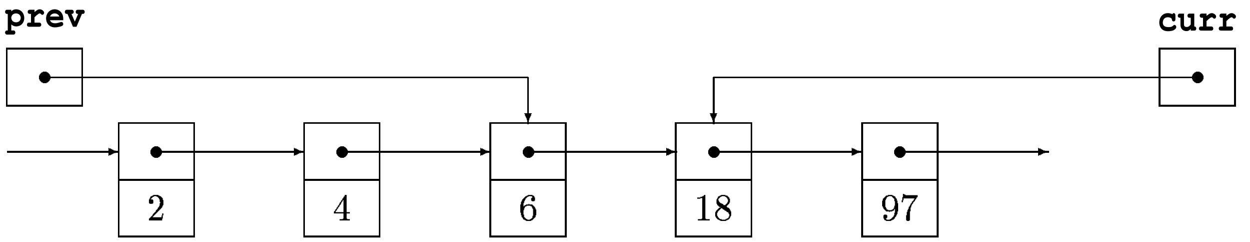
(Figure 19.6 – Page 476)

```
LIST list_insert(LIST list, void *target, CMP_FPTR cmpfptr)
{
  LISTNODE *nptr;                         /* ptr to new node */
  LISTNODE *curr = list;                  /* ptr to current node */
  LISTNODE *prev = NULL;                  /* ptr to previous node */

  for (; curr != NULL; prev = curr, curr = curr->nextptr)
    if (cmpfptr(curr->dataptr, target) > 0)
      break;                              /* found place for node */
  nptr = ln_create(target, curr);
  if (prev == NULL)
    list = nptr;                          /* insert at front */
  else
    prev->nextptr = nptr;                 /* insert in middle or end */
  return list;                            /* return updated list */
}

void list_destroy(LIST list)
{
  LISTNODE *curr = list;                  /* ptr to current node */
  LISTNODE *temp;                         /* ptr to next node */

  for ( ; curr != NULL; curr = temp)
  {
    temp = curr->nextptr;
    ln_destroy(curr);
  }
}
```

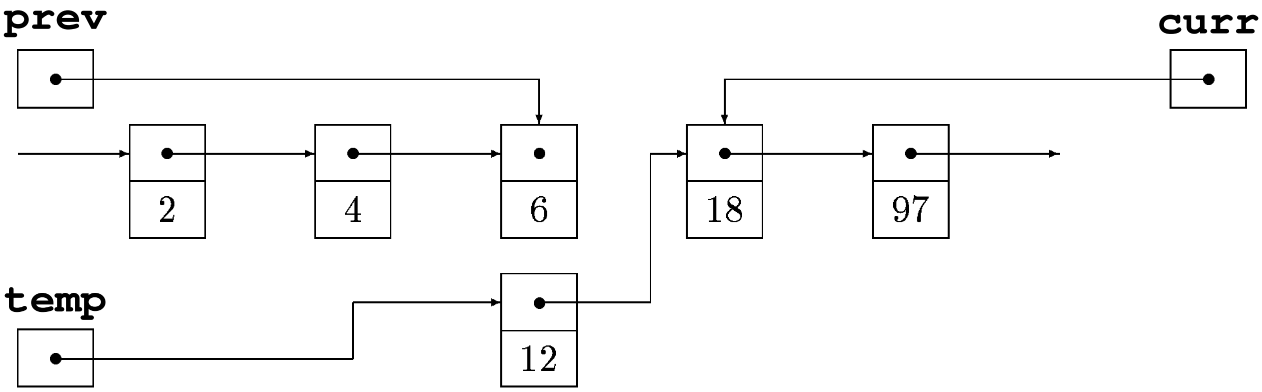
INSERTING INTO A LINKED LIST



(a) Before searching the list.



(b) When we find the desired node.



(c) Halfway through inserting 12.

strlists.c

(Figure 19.8 – Page 479)

```
/*
 * Sort strings using generic linked lists.
 */
#include <stdio.h>
#include <string.h>
#include "lists.h"                          /* for list data type */

#define MAXLEN 80

int main()
{
  int  getline(char *line, int max);
  int  cmpstr(const void *ptr1, const void *ptr2);
  void printstr(void *ptr);
  char line[MAXLEN + 1];                    /* input line */
  char *ptr;                                /* ptr to input line copy */
  int  len;                                 /* length of input line */
  LIST sl = list_create();                  /* list of strings */

  while ((len = getline(line, MAXLEN)) != -1)
    if ((ptr = malloc(len + 1)) != NULL)
      sl = list_insert(sl, strcpy(ptr,line), cmpstr);
    else
    {
      printf("Out of memory reading line %s\n", line);
      break;
    }
  list_traverse(sl, printstr);              /* print elements */
  list_traverse(sl, free);                  /* free data */
  list_destroy(sl);                         /* free nodes */
  return 0;
}
```

strfuncs.c

(Figure 19.9 – Page 480)

```
/*
 * Comparison and printing functions.
 *   cmpstr - given generic pointers, compare strings.
 *   printstr - given generic pointer, print string.
 */
#include <stdio.h>
#include <string.h>

int cmpstr(const void *xptr, const void *yptr)
{
  return strcmp((const char *) xptr, (const char *) yptr);
}

void printstr(void *ptr)
{
  printf("%s\n", (const char *) ptr);
}
```

lists2.c

(Figure 19.10 – Page 482)

```
/*
 * Update a list item or create it if it's not already there.
 */
#include "lists.h"

LIST list_update(LIST list, void *target, CMP_FPTR cfptr, UPD_FPTR ufptr)
{
  LISTNODE *nptr;                       /* pointer to new node */
  LISTNODE *curr;                       /* pointer to current node */
  LISTNODE *prev = NULL;                /* pointer to previous node */
  int cmp;                              /* result of list comparison */

  for (curr = list; curr != NULL; curr = curr->nextptr)
  {
    if ((cmp = cfptr(target, curr->dataptr)) <= 0)
      break;
    prev = curr;
  }
  if (cmp == 0)                         /* found it in list; do update */
    curr->dataptr = ufptr(curr->dataptr, UPDATE);
  else
  {                                     /* not in list, so create */
    nptr = ln_create(ufptr(target, CREATE), curr);
    if (prev == NULL)
      list = nptr;                      /* insert at front */
    else
      prev->nextptr = nptr;             /* insert at middle or end */
  }
  return list;                          /* return updated list */
}
```

cntwords.c

(Figure 19.11 – Page 483)

```
/*
 * Prints number of times each word appears in input, sorted by word
 * (assumes one word per input line).
 */
#include <stdio.h>
#include <string.h>
#include "items.h"

int main()
{
  int getline(char *line, int max);      /* for reading input line */
  char line[MAXLEN + 1];                 /* input line */
  LIST sl = list_create();               /* list of strings */

  while (getline(line, MAXLEN) != -1)
    if (line[0] != '\0')
      sl = list_update(sl, line, cmpitem, upditem);
  list_traverse(sl, printitem);
  return 0;
}
```

items.h

(Figure 19.12 – Page 483)

```
/*
 * ITEM definition and prototypes.
 */
#include "lists.h"

#define  MAXLEN   80                   /* longest input line */

typedef struct item
{
  char wordtext[MAXLEN + 1];           /* text of word */
  int cnt;                             /* count of appearances */
} ITEM;

int  cmpitem(const void *ptr1, const void *ptr2);
void *upditem(void *ptr, enum status updtype);
void printitem(void *ptr);
```

items.c
(Figure 19.13 – Page 484)

```
/*
 * Functions managing ITEM type.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "items.h"

void *upditem(void *target, enum status todo)   /* update/create item */
{
  ITEM *itemptr = target;

  if (todo == CREATE)                          /* not already there, create */
  {
    if ((itemptr = malloc(sizeof(ITEM))) == NULL)
      printf("Couldn't allocate space for item\n");
    else
    {
      strncpy(itemptr->wordtext, target, MAXLEN);
      itemptr->cnt = 1;
    }
  }
  else                                         /* already there, update */
    itemptr->cnt++;
  return itemptr;
}

int cmpitem(const void *xptr, const void *yptr)  /* compare items */
{
  return strcmp((const char *) xptr, ((ITEM *) yptr)->wordtext);
}

void printitem(void *ptr)                        /* print item */
{
  printf("%s %i\n", ((ITEM *) ptr)->wordtext, ((ITEM *) ptr)->cnt);
}
```

stacks.h

(Figure 19.14 – Page 486)

```
/*
 * Definitions of stack data type and prototypes.
 */
#include "lists.h"

typedef LISTNODE *STACK;

#define stack_create()   NULL              /* create an empty stack */
#define stack_empty(s)   ((s) == NULL)     /* is stack empty? */
#define stack_top(s)     ((s)->dataptr)    /* first item on stack */

STACK stack_push(STACK s, void *dataptr);
STACK stack_pop(STACK s);
```

stacks.c

(Figure 19.15 – Page 486)

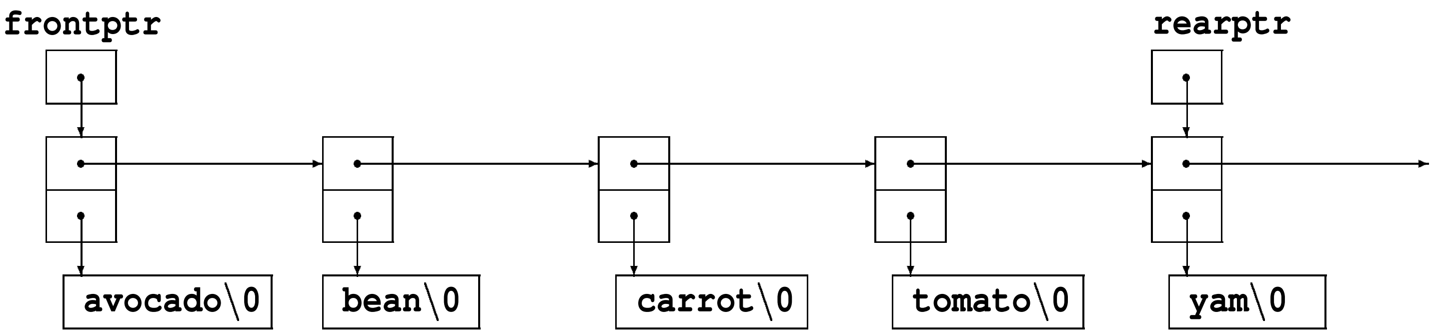
```
/*
 * Stack operations.
 */
#include "stacks.h"                        /* STACK type and prototypes */

STACK stack_push(STACK s, void *dptr)     /* add new item to stack */
{
  return ln_create(dptr, s);
}

STACK stack_pop(STACK s)                   /* delete top item from stack */
{
  LISTNODE *topptr    = s;
  LISTNODE *secondptr = topptr->nextptr;

  ln_destroy(topptr);
  return secondptr;
}
```

strrev.c

(Figure 19.16 – Page 487)

```
/*
 * Reverse input strings using a stack.
 */
#include <stdio.h>
#include <string.h>
#include "stacks.h"                /* STACK type and prototypes */

#define MAXLEN 80

int main()
{
  int   getline(char *buf, int len);
  char  line[MAXLEN + 1];              /* input line */
  char *ptr;                           /* copy of input line */
  STACK s = stack_create();            /* our stack */
  int len;

  while ((len = getline(line, MAXLEN)) != -1)
    if ((ptr = malloc(len + 1)) != NULL)
      s = stack_push(s, strcpy(ptr,line));
    else
    {
      printf("Out of memory processing line %s\n", line);
      break;
    }
  for (; !stack_empty(s); s = stack_pop(s))
  {
    printf("%s\n", ptr = stack_top(s));
    free(ptr);
  }
  return 0;
}
```

REPRESENTING A QUEUE



qs.h
(Figure 19.18 – Page 488)

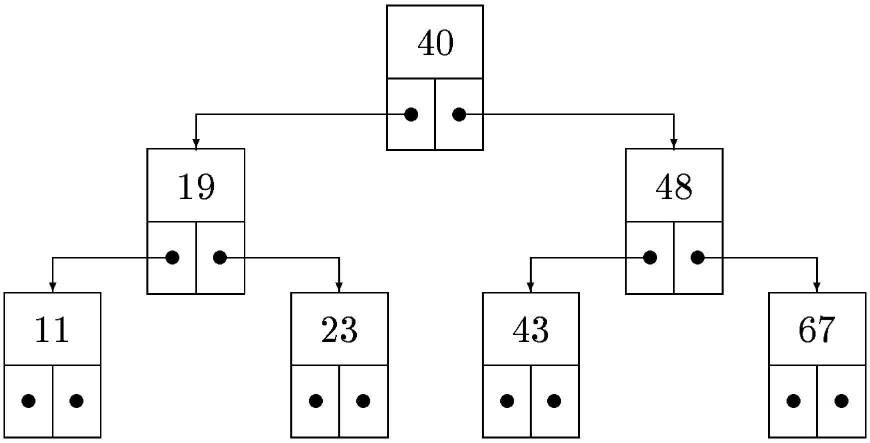
```
/*
 * Queue type and prototypes.
 */
#include "lists.h"                    /* list type and prototypes */

typedef struct queue
{
  LISTNODE *frontptr;                 /* pointer to first list node */
  LISTNODE *rearptr;                  /* pointer to last list node */
} *QUEUE;

QUEUE queue_create(void);
void  enqueue(QUEUE q, void *dataptr);
void  *dequeue(QUEUE q);

#define queue_empty(qptr) \
        ((qptr)->frontptr == NULL)    /* is queue empty? */
```

qs.c

(Figure 19.19 – Page 489)

```
/*
 * Queue-manipulating functions.
 */
#include "qs.h"                       /* QUEUE type and prototypes */

QUEUE queue_create(void)              /* create new, empty queue */
{
  QUEUE qptr = malloc(sizeof(struct queue));

  if (qptr != NULL)
    qptr->frontptr = qptr->rearptr = NULL;
  return qptr;
}

void enqueue(QUEUE qptr, void *ptr)   /* add item to rear of queue */
{
  LISTNODE *nptr = ln_create(ptr, NULL);

  if (qptr->frontptr == NULL)         /* empty queue */
    qptr->frontptr = nptr;
  else                                /* nonempty queue */
    qptr->rearptr->nextptr = nptr;
  qptr->rearptr = nptr;
}

void *dequeue(QUEUE qptr)             /* remove first item in queue */
{
  LISTNODE *nptr = qptr->frontptr;

  if ((qptr->frontptr = nptr->nextptr) == NULL)
    qptr->rearptr = NULL;             /* now queue empty */
  return nptr->dataptr;
}
```

A Binary Search Tree



tnodes.h

(Figure 19.21 – Page 490)

```
/*
 * Definitions and prototypes for a tree node.
 */
typedef struct treenode
{
  void            *dataptr;
  struct treenode *left;
  struct treenode *right;
}
  TREENODE;

TREENODE *tn_create(void *nodeptr);
void tn_destroy(TREENODE *nodeptr);
```

tnodes.c

(Figure 19.22 – Page 490)

```c
/*
 * Manage a tree node.
 *   tn_create - create a new tree node.
 *   tn_destroy - remove an existing tree node.
 */
#include <stdlib.h>
#include <stddef.h>
#include "tnodes.h"

TREENODE *tn_create(void *ptr)
{
  TREENODE *nptr = malloc(sizeof(TREENODE));

  nptr->dataptr = ptr;
  nptr->left = nptr->right = NULL;
  return nptr;
}

void tn_destroy(TREENODE *ptr)
{
  free(ptr);
}
```

trees.h

(Figure 19.23 – Page 491)

```
/*
 * Definition of a tree and protoypes for its operations.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include "tnodes.h"                    /* tree node and prototypes */
#include "ptrfuncs.h"                  /* pointer to function defs */

typedef TREENODE *TREE;

#define tree_create()  NULL            /* create an empty tree */

TREE tree_insert(TREE t, void *dataptr, CMP_FPTR cmp);
void tree_traverse(TREE t, ACT_FPTR action);
void tree_destroy(TREE t);
TREE tree_update(TREE t, void *dataptr, CMP_FPTR cmp, UPD_FPTR upd);
```

trees1.c

(Figure 19.24 – Page 492)

```
/*
 * Manage a binary search tree.
 *   tree_insert - insert a new node in a tree.
 *   tree_destroy - remove all nodes in tree.
 *   tree_traverse - traverse tree in order, doing action.
 */
#include <stddef.h>
#include "trees.h"

TREE tree_insert(TREE root, void *target, CMP_FPTR cmpfptr)
{
  TREENODE *curr = root;                 /* current node in tree */
  TREENODE *prev = NULL;                 /* parent node in tree */
  TREENODE *temp = tn_create(target);    /* new node */
  int cmp;                               /* result of comparison */

  for (; curr != NULL; curr = cmp < 0 ? curr->left : curr->right)
  {
    cmp = cmpfptr(target, curr->dataptr);
    prev = curr;
  }
  if (prev == NULL)                      /* node is parent of tree */
    root = temp;
  else if (cmp < 0)
    prev->left = temp;                   /* left child */
  else
    prev->right = temp;                  /* right child */
  return root;
}
```

trees1.c

(Figure 19.24 – Page 492)

```
void tree_destroy(TREE root)
{
  if (root != NULL)
  {
    tree_destroy(root->left);
    tree_destroy(root->right);
    tn_destroy(root);
  }
}

void tree_traverse(TREE root, ACT_FPTR fptr)
{
  if (root != NULL)
  {
    tree_traverse(root->left, fptr);
    fptr(root->dataptr);
    tree_traverse(root->right, fptr);
  }
}
```

trees2.c

(Figure 19.25 – Page 493)

```
/*
 * A function to update a node or insert a new node.
 */
#include <stddef.h>
#include "trees.h"

TREE tree_update(TREE root, void *target, CMP_FPTR cfptr, UPD_FPTR ufptr)
{
  TREENODE *curr = root;              /* current tree node */
  TREENODE *prev = NULL;              /* parent tree node */
  TREENODE *temp;                     /* new tree node */
  int cmp;                            /* result of comparison */

  for (; curr != NULL; curr = cmp < 0 ? curr->left : curr->right)
  {
    if ((cmp = cfptr(target, curr->dataptr)) == 0)
    {                                 /* found match, so update */
      curr->dataptr = ufptr(curr->dataptr, UPDATE);
      return root;
    }
    prev = curr;
  }
  temp = tn_create(ufptr(target, CREATE));
  if (prev == NULL)                   /* insert new node, like before */
    root = temp;
  else if (cmp < 0)
    prev->left = temp;
  else
    prev->right = temp;
  return root;
}
```

tsort.c

(Figure 19.26 – Page 495)

```
/*
 * Insertion sort strings using trees.
 */
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include "trees.h"                       /* tree type and prototypes */

#define MAXLEN 80

int main()
{
  int  getline(char *line, int len);
  void printstr(void *ptr);
  int  cmpstr(const void *ptr1, const void *ptr2);
  char line[MAXLEN + 1];                 /* input line */
  char *ptr;                             /* ptr to input line copy */
  TREE st = tree_create();               /* tree of strings */
  int  len;

  while ((len = getline(line, MAXLEN)) != -1)
    if ((ptr = malloc(len + 1)) != NULL)
      st = tree_insert(st, strcpy(ptr,line), cmpstr);
    else
    {
      printf("Out of memory processing line %s\n", line);
      break;
    }
  tree_traverse(st, printstr);
  tree_traverse(st, free);              /* not really needed */
  tree_destroy(st);
  return 0;
}
```

SAMPLE I/O – xref

```
Balance             18
Generate            2
Interest            17
INTRATE             7 17 22
PRINCIPAL           6 15
PERIOD              8 19
Rate                17
Year                18
at                  13
accumulation        2
a                   2
balance             13 13 15 21 22 22 22
define              6 7 8
```

...

```
that                25
table               2
worked              25
with                6
while               19
year                8 12 12 13 16 19 21 23 23
```

xref.h

(Figure 19.28 – Page 496)

```
/*
 * Define cross-referencer tree items and prototypes.
 */
#include "trees.h"
#include "qs.h"

#define MAXWORD 20

typedef struct item                  /* WORD NODE */
{
  char wordtext[MAXWORD + 1];        /* value of the word */
  QUEUE lines;                       /* list of lines numbers */
} ITEM;

void *upditem(void *dataptr, enum status updtype);
int  cmpitem(const void *ptr1, const void *ptr2);
void printitem(void *dataptr);

extern char CurrentWord[];
extern int CurrentLineNo;
```

xref.c

(Figure 19.29 – Page 497)

```
/*
 * The cross-referencing program.
 */
#include <stdio.h>
#include <ctype.h>
#include "xref.h"

char CurrentWord[MAXWORD + 1];
int  CurrentLineNo = 0;

int main()
{
  int word_get(void);
  TREE words = tree_create();

  while (word_get() != 0)
    words = tree_update(words, CurrentWord, cmpitem, upditem);
  tree_traverse(words, printitem);
  return 0;
}
```

xref.c

(Figure 19.29 – Page 497)

```
/* Read next character, updating line count */

int getnextchar(void)
{
  static int lastc = '\n';
  int c = getchar();

  if (lastc == '\n')
    CurrentLineNo++;
  return lastc = c;
}

/* Read next word, storing into CurrentWord */

int word_get(void)
{
  int next;                                /* next character */
  char *ptr    = CurrentWord;              /* start of word */
  char *endptr = CurrentWord + MAXWORD;    /* end of word */

  while ((next = getnextchar()) != EOF && !isalpha(next))
     ;                                     /* skip non-letters */
  for (; next != EOF && isalpha(next); next = getnextchar())
    if (ptr < endptr)                      /* save up to max chars */
      *ptr++ = next;
  *ptr = '\0';
  return ptr - CurrentWord;                /* return chars in word */
}
```

utils.c

(Figure 19.30 – Page 498)

```
/*
 * The cross-referencer's comparison and update functions.
 */
#include "xref.h"

int cmpitem(const void *xptr, const void *yptr)   /* compare items */
{
  return strcmp((const char *) xptr, ((ITEM *) yptr)->wordtext);
}

void *upditem(void *ptr, enum status todo)        /* update item */
{
  ITEM *iptr = ptr;
  int *lptr = malloc(sizeof(int));                /* holds line # */

  if (todo == CREATE)
  {                                               /* new word */
    iptr = malloc(sizeof(ITEM));
    strcpy(iptr->wordtext, CurrentWord);
    iptr->lines = queue_create();
  }
  *lptr = CurrentLineNo;                          /* add line # */
  enqueue(iptr->lines, lptr);
  return iptr;
}

void printitem(void *ptr)
{                          /* write word and count on standard output */
  ITEM *iptr = ptr;                     /* item in current node */

  printf("%-*s", MAXWORD, iptr->wordtext);
  while (!queue_empty(iptr->lines))
    printf(" %i", * (int *) dequeue(iptr->lines));
  putchar('\n');
}
```

REMINDERS

- Don’t forget to make sure that all lists are properly **`NULL`** terminated.

CHAPTER SUMMARY

- By using structures, pointers, and dynamic allocation, we can implement data structures other than arrays.

- Linked lists consist of a list of nodes. Each node contains a pointer to a data item and a pointer to the next node in the list. The last node in the list contains a pointer to **`NULL`**.

- Binary trees consists of a set of nodes, each node containing a pointer to a data item and two pointers, one to a node containing a larger item, the other a pointer to a smaller item.

- A stack is a last-in, first-out data structure, easily implemented as a list in which all items are added and deleted to the front of the list.

- A queue is a first-in, first-out data structure, easily implemented as a list in which all items are added to the end and deleted from the front.