SECTION 6.5

6.5 Self-referential Structures

Suppose we want to handle the more general problem of counting the occurrences of *all* the words in some input. Since the list of words isn't known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each word as it arrives, to see if it's already been seen; the program would take too long. (More precisely, its running time is likely to grow quadratically with the number of input words.) How can we organize the data to cope efficiently with a list of arbitrary words?

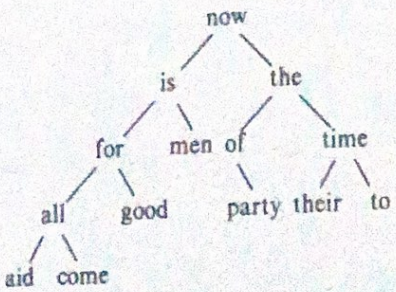
One solution is to keep the set of words seen so far sorted at all times, by placing each word into its proper position in the order as it arrives. This shouldn't be done by shifting words in a linear array, though—that also takes too long. Instead we will use a data structure called a *binary tree*.

The tree contains one "node" per distinct word; each node contains

a pointer to the text of the word
a count of the number of occurrences
a pointer to the left child node
a pointer to the right child node

No node may have more than two children; it might have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words that are lexicographically less than the word at the node, and the right subtree contains only words that are greater. This is the tree for the sentence "now is the time for all good men to come to the aid of their party", as built by inserting each word as it is encountered:



To find out whether a new word is already in the tree, start at the root and compare the new word to the word stored at that node. If they match, the question is answered affirmatively. If the new word is less than the tree word, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new word is not in the tree, and in fact the empty slot is the proper place to add the new word. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is conveniently represented as a structure with four components:

```
struct tnode {          /* the tree node: */
    char *word;             /* points to the text */
    int count;              /* number of occurrences */
    struct tnode *left;     /* left child */
    struct tnode *right;    /* right child */
};
```

This recursive declaration of a node might look chancy, but it's correct. It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares `left` to be a pointer to a `tnode`, not a `tnode` itself.

Occasionally, one needs a variation of self-referential structures: two structures that refer to each other. The way to handle this is:

```
struct t {
    ...
    struct s *p;    /* p points to an s */
};
struct s {
    ...
    struct t *q;    /* q points to a t */
};
```

The code for the whole program is surprisingly small, given a handful of supporting routines like `getword` that we have already written. The main routine reads words with `getword` and installs them in the tree with `addtree`.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* word frequency count */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

The function `addtree` is recursive. A word is presented by `main` to the top level (the root) of the tree. At each stage, that word is compared to the word already stored at the node, and is percolated down to either the left or right subtree by a recursive call to `addtree`. Eventually the word either matches something already in the tree (in which case the count is incremented), or a null pointer is encountered, indicating that a node must be created and added to the tree. If a new node is created, `addtree` returns a pointer to it, which is installed in the parent node.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree:  add a node with w, at or below p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {     /* a new word has arrived */
        p = talloc();    /* make a new node */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;      /* repeated word */
    else if (cond < 0)   /* less than into left subtree */
        p->left = addtree(p->left, w);
    else                 /* greater than into right subtree */
        p->right = addtree(p->right, w);
    return p;
}
```

Storage for the new node is fetched by a routine `talloc`, which returns a pointer to a free space suitable for holding a tree node, and the new word is copied to a hidden place by `strdup`. (We will discuss these routines in a moment.) The count is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. We have (unwisely) omitted error checking on the values returned by `strdup` and `talloc`.

`treeprint` prints the tree in sorted order; at each node, it prints the left subtree (all the words less than this word), then the word itself, then the right subtree (all the words greater). If you feel shaky about how recursion works, simulate `treeprint` as it operates on the tree shown above.

```
/* treeprint:  in-order print of tree p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

A practical note: if the tree becomes "unbalanced" because the words don't arrive in random order, the running time of the program can grow too much. As a worst case, if the words are already in order, this program does an expensive simulation of linear search. There are generalizations of the binary tree that do not suffer from this worst-case behavior, but we will not describe them here.

Before we leave this example, it is also worth a brief digression on a problem related to storage allocators. Clearly it's desirable that there be only one storage allocator in a program, even though it allocates different kinds of objects. But if one allocator is to process requests for, say, pointers to chars and pointers to `struct tnodes`, two questions arise. First, how does it meet the requirement of most real machines that objects of certain types must satisfy alignment restrictions (for example, integers often must be located at even addresses)? Second, what declarations can cope with the fact that an allocator must necessarily return different kinds of pointers?

Alignment requirements can generally be satisfied easily, at the cost of some wasted space, by ensuring that the allocator always returns a pointer that meets *all* alignment restrictions. The `alloc` of Chapter 5 does not guarantee any particular alignment, so we will use the standard library function `malloc`, which does. In Chapter 8 we will show one way to implement `malloc`.

The question of the type declaration for a function like `malloc` is a vexing one for any language that takes its type-checking seriously. In C, the proper method is to declare that `malloc` returns a pointer to `void`, then explicitly coerce the pointer into the desired type with a cast. `malloc` and related routines are declared in the standard header `<stdlib.h>`. Thus `talloc` can be written as

```
#include <stdlib.h>

/* talloc:  make a tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

`strdup` merely copies the string given by its argument into a safe place, obtained by a call on `malloc`:


```
char *strdup(char *s)    /* make a dupli
{
    char *p;

    p = (char *) malloc(strlen(s)+1);
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

`malloc` returns `NULL` if no space is available; `str`
leaving error-handling to its caller.

Storage obtained by calling `malloc` may be
`free`; see Chapters 7 and 8.

Exercise 6-2. Write a program that reads a C prog
cal order each group of variable names that are id
ters, but different somewhere thereafter. Don't co
comments. Make 6 a parameter that can be set fr

Exercise 6-3. Write a cross-referencer that prints
ment, and, for each word, a list of the line numbe
noise words like "the," "and," and so on. □

Exercise 6-4. Write a program that prints the di
into decreasing order of frequency of occurren
count. □

6.6 Table Lookup

In this section we will write the innards of
trate more aspects of structures. This code is t
the symbol table management routines of a ma
example, consider the `#define` statement. W

```
#define  IN  1
```

is encountered, the name `IN` and the replace
Later, when the name `IN` appears in a statem

```
state = IN;
```

it must be replaced by `1`.

There are two routines that manipulate
`install(s,t)` records the name `s` and t
and `t` are just character strings. `lookup(`
returns a pointer to the place where it was f

The algorithm is a hash search—the inc

```
char *strdup(char *s)    /* make a duplicate of s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1);  /* +1 for '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

malloc returns NULL if no space is available; strdup passes that value on, leaving error-handling to its caller.

Storage obtained by calling malloc may be freed for re-use by calling free; see Chapters 7 and 8.

Exercise 6-2. Write a program that reads a C program and prints in alphabetical order each group of variable names that are identical in the first 6 characters, but different somewhere thereafter. Don't count words within strings and comments. Make 6 a parameter that can be set from the command line. □

Exercise 6-3. Write a cross-referencer that prints a list of all words in a document, and, for each word, a list of the line numbers on which it occurs. Remove noise words like "the," "and," and so on. □

Exercise 6-4. Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count. □

6.6 Table Lookup

In this section we will write the innards of a table-lookup package, to illustrate more aspects of structures. This code is typical of what might be found in the symbol table management routines of a macro processor or a compiler. For example, consider the #define statement. When a line like

```
#define  IN  1
```

is encountered, the name IN and the replacement text 1 are stored in a table. Later, when the name IN appears in a statement like

```
state = IN;
```

it must be replaced by 1.

There are two routines that manipulate the names and replacement texts. install(s,t) records the name s and the replacement text t in a table; s and t are just character strings. lookup(s) searches for s in the table, and returns a pointer to the place where it was found, or NULL if it wasn't there.

The algorithm is a hash search—the incoming name is converted into a small