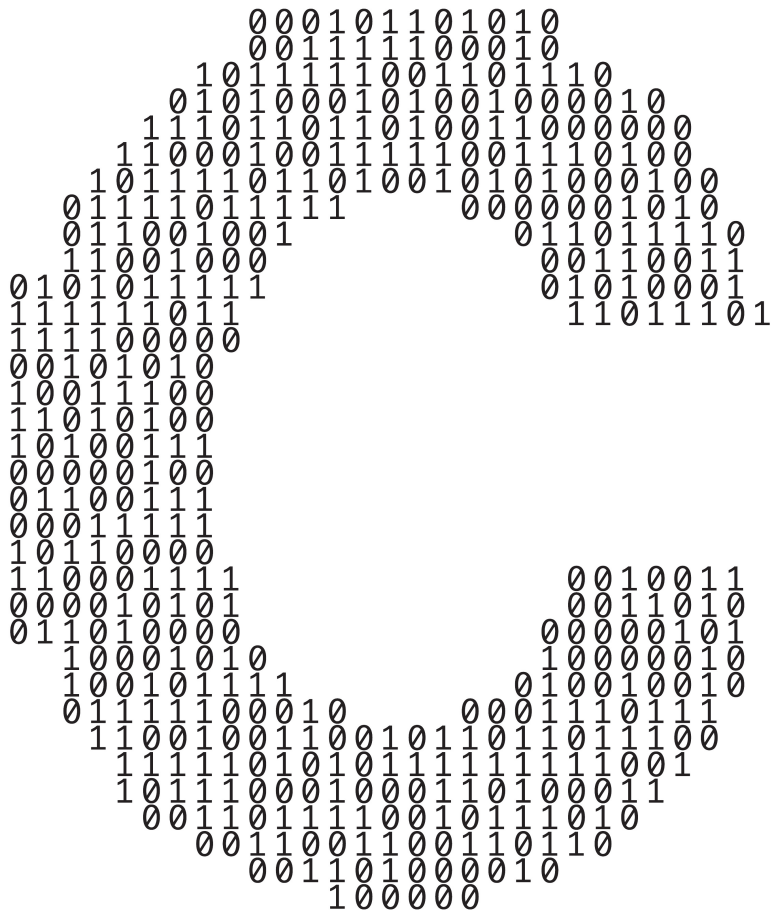
12

Data Structures



Objectives

In this chapter, you'll:

- Allocate and free memory dynamically for data objects.
- Form linked data structures using pointers, self-referential structures and recursion.
- Create and manipulate linked lists, queues, stacks and binary trees.
- Learn important applications of linked data structures.
- Study Secure C programming recommendations for pointers and dynamic memory allocation.
- Optionally build your own compiler in the exercises.

12.1 Introduction

We've studied fixed-size data structures, including one-dimensional arrays, two-dimensional arrays and `structs`. This chapter introduces **dynamic data structures** that can grow and shrink at execution time:

- **Linked lists** are collections of data items "lined up in a row." You can insert and delete items anywhere in a linked list.
- **Stacks** are important in compilers and operating systems. You can insert and delete items only at one end of a stack, known as its **top**.
- **Queues** represent waiting lines. You can insert only at the queue's back and delete only from its front. The back and front are known as the queue's **tail** and **head**.
- **Binary trees** facilitate high-speed searching and sorting of data, efficiently eliminating duplicate data items and compiling expressions into machine language.

Each of these data structures has many other interesting applications.

Optional Project: Building Your Own Compiler

We hope you'll attempt the optional major project described in the special section entitled Building Your Own Compiler at the end of the exercises. You've been using a compiler to translate your C programs to machine language so that you could execute them. In this project, you'll build your own compiler. It will read a file of statements written in a simple yet powerful, high-level language. Your compiler will translate these statements into a file of Simpletron Machine Language (SML) instructions. SML is the (Deitel-created) language you learned in Chapter 7's special section, Building Your Own Computer. Your Simpletron Simulator program will then execute the SML program produced by your compiler! This project enables you to exercise most of what you've learned in this book. The special section carefully walks you

through the high-level language's specifications and describes the algorithms for converting each high-level language statement into machine-language instructions. If you enjoy challenges, you might attempt the many enhancements to both the compiler and the Simpletron Simulator we suggest in the exercises.

Self Check

1 *(Fill-In)* Which data structure is described by "facilitates high-speed searching and sorting of data, efficiently eliminating duplicate data items and compiling expressions into machine language"? _____.
Answer: Binary tree.

2 *(Fill-In)* Which data structure is described by "a collection of data items lined up in a row—insertions and deletions are made anywhere in the data structure"? _____.
Answer: Linked list.

3 *(Fill-In)* Which dynamic data structure is described by "You can insert and delete items only at one end, the top"? _____.
Answer: Stack.

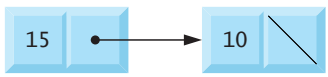
12.2 Self-Referential Structures

A **self-referential structure** contains a pointer member that points to a structure of the *same* structure type. For example, the following definition creates the type, `struct node`:

```
struct node {
   int data;
   struct node *nextPtr;
};
```

Our `struct node` has two members—integer member `data` and pointer member `nextPtr`. The `nextPtr` points to another `struct node`. This structure has the same type as the one we're defining, hence the term self-referential structure. Member `nextPtr` is a **link**—it can be used to link a `struct node` to another `struct node`. We link self-referential structure objects to form lists, queues, stacks and trees.

The following diagram illustrates two self-referential structure objects linked together to form a list:



The slash[1] in the last node represents a `NULL` pointer, which indicates that the node does not point to another node. A `NULL` pointer indicates the end of a data structure. Not setting the last node's link to `NULL` can lead to runtime errors. ERR

1. The slash is only for illustration purposes. It does not correspond to the C's backslash character.

Self Check

1 *(Fill-In)* What should replace `???` in the following code to make this a self-referential `struct`? _____.

```
struct node {
   int data;
   ??? *nextPtr;
};
```

Answer: `struct node`.

2 *(Fill-In)* A _____ pointer indicates the end of a data structure.
Answer: `NULL`.

3 *(Fill-In)* Self-referential structures can be _____ together to form data structures such as lists, queues, stacks and trees.
Answer: linked.

12.3 Dynamic Memory Management

Creating and maintaining dynamic data structures that grow and shrink at execution time requires **dynamic memory management**, which has two components:

- obtaining more memory at execution time to hold new nodes, and
- releasing memory no longer needed.

The function `malloc`, the function `free` and the operator `sizeof` are essential to dynamic memory management.

The `malloc` Function

To request memory at execution time, pass to the function `malloc` the number of bytes to allocate. If successful, `malloc` returns a `void *` pointer to the allocated memory. Recall that a `void *` pointer may be assigned to a variable of *any* pointer type.

Function `malloc` most commonly is used with `sizeof`. For example, the following statement determines a `struct node` object's size in bytes with `sizeof(struct node)`, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in `newPtr`:

```
newPtr = malloc(sizeof(struct node));
```

SE The memory is not guaranteed to be initialized, though many implementations initialize it for security. If no memory is available, `malloc` returns `NULL`. Always test for a `NULL` pointer before accessing the dynamically allocated memory to avoid runtime errors that might crash your program.

The `free` Function

When you no longer need a block of dynamically allocated memory, return it to the system immediately by calling the `free` function to deallocate the memory. This

returns it to the system for potential reallocation in the future. To free the memory from the preceding `malloc` call, use the statement

```
free(newPtr);
```

After deallocating memory, set the pointer to `NULL`. This prevents accidentally referring to that memory, which may have already been allocated for another purpose.

Not freeing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak." Referring to memory that has been freed is an error that typically causes a program to crash. Freeing memory that you did not allocate dynamically with `malloc` is an error.

ERR

ERR

Functions `calloc` and `realloc`

C also provides the functions `calloc` and `realloc` for creating and modifying the size of dynamic arrays. Section 15.8 discusses these functions.

Self Check

1 *(True/False)* Function `malloc` takes as an argument the number of bytes to be allocated and returns a `NULL` pointer.

Answer: *False*. Actually, `malloc` returns a `void *` pointer with the allocated memory's address or returns a `NULL` pointer if the memory could not be allocated.

2 *(Discussion)* Describe precisely what the following statement does:

```
newPtr = malloc(sizeof(struct node));
```

Answer: The statement evaluates `sizeof(struct node)` to determine the object's size in bytes, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in `newPtr`.

3 *(Discussion)* Write a statement that frees the memory that was dynamically allocated to `newPtr` by `malloc`.

Answer: `free(newPtr);`

4 *(Fill-In)* Not `free`ing dynamically allocated memory when it's no longer needed can cause the system to run out of memory. This is sometimes called a(n) _____.

Answer: memory leak.

12.4 Linked Lists

A **linked list** is a linear collection of self-referential `struct` objects, called **nodes**, connected by pointer links—hence the term "linked" list. You access a linked list via a pointer to its first node and access subsequent nodes via the nodes' pointer link members. You dynamically store data in a linked list by creating each node as necessary. A node may contain any type of data, including other `struct` objects. Stacks and queues are also **linear data structures**. You'll soon see that these are constrained versions of linked lists.

Arrays vs. Linked Lists

You can store lists of data in arrays, but linked lists provide several advantages:

- A linked list is appropriate when the number of data items is unpredictable. A linked list is dynamic, so its length can increase or decrease as necessary. Arrays are fixed-size data structures (though Section 15.8 shows how to dynamically allocate and reallocate arrays).

PERF

- An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Using linked lists and dynamic memory allocation for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that the pointers in a list's nodes require additional memory. Also, dynamic memory allocation incurs the overhead of function calls.
- Fixed-size arrays can become full. Linked lists become full only when the system has insufficient *memory* to satisfy dynamic storage-allocation requests.

PERF

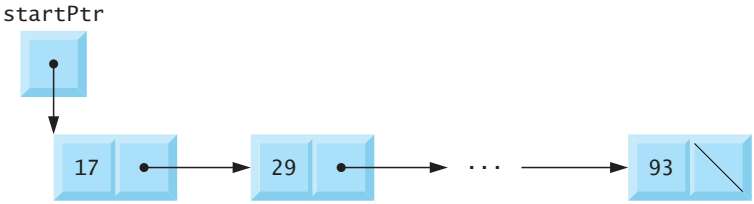
- Linked lists can be maintained in sorted order by inserting each new element at the appropriate point in the list. Inserting into and deleting from a sorted array can be time-consuming. All elements following the inserted or deleted element must be shifted appropriately.

Arrays Are Faster for Direct Element Access

PERF On the other hand, array elements are stored contiguously in memory. This allows immediate access to any array element—any element's address can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate access to their elements.

Illustrating a Linked List

Linked-list nodes are not guaranteed to be stored contiguously in memory. Logically, however, the nodes appear to be contiguous. The following diagram illustrates a linked list with several nodes.



Implementing a Linked List

Figure 12.1 manipulates a list of characters. You can insert a character in the list in alphabetical order (function `insert`) or delete a character from the list (function `delete`). A detailed discussion of the program follows. We've split this program's code for discussion purposes—the output is shown at the end of the first code table below.

```c
// fig12_01.c
// Inserting and deleting nodes in a list
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct listNode {
   char data; // each listNode contains a character
   struct listNode *nextPtr; // pointer to next node
};

typedef struct listNode ListNode; // synonym for struct listNode
typedef ListNode *ListNodePtr; // synonym for ListNode*

// prototypes
void insert(ListNodePtr *sPtr, char value);
char delete(ListNodePtr *sPtr, char value);
int isEmpty(ListNodePtr sPtr);
void printList(ListNodePtr currentPtr);
void instructions(void);

int main(void) {
   ListNodePtr startPtr = NULL; // initially there are no nodes
   char item = '\0'; // char entered by user

   instructions(); // display the menu
   printf("%s", "? ");
   int choice = 0; // user's choice
   scanf("%d", &choice);

   // loop while user does not choose 3
   while (choice != 3) {
      switch (choice) {
         case 1: // insert an element
            printf("%s", "Enter a character: ");
            scanf("\n%c", &item);
            insert(&startPtr, item); // insert item in list
            printList(startPtr);
            break;
         case 2: // delete an element
            if (!isEmpty(startPtr)) { // if list is not empty
               printf("%s", "Enter character to be deleted: ");
               scanf("\n%c", &item);

               // if character is found, remove it
               if (delete(&startPtr, item)) { // remove item
                  printf("%c deleted.\n", item);
                  printList(startPtr);
               }
               else {
                  printf("%c not found.\n\n", item);
               }
            }
```

Fig. 12.1 | Inserting and deleting nodes in a list. (Part 1 of 3.)

```
            else {
               puts("List is empty.\n");
            }

            break;
         default:
            puts("Invalid choice.\n");
            instructions();
            break;
      }

      printf("%s", "? ");
      scanf("%d", &choice);
   } // end while

   puts("End of run.");
}

// display program instructions to user
void instructions(void) {
   puts("Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.");
}
```

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A

The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.
```

Fig. 12.1 | Inserting and deleting nodes in a list. (Part 2 of 3.)

```
? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Fig. 12.1 | Inserting and deleting nodes in a list. (Part 3 of 3.)

Lines 7–10 define the self-referential structure `struct listNode`, which we use to build this example's linked list. Lines 12 and 13 define typedefs that we use to make the code more readable. The name `ListNode` represents a `struct listNode` object, and the name `ListNodePtr` represents a pointer to a `struct listNode` object. The `main` function enables you to insert characters in the list (lines 34–39), delete items from the list (lines 40–58) or terminate the program. Initially, `startPtr` (line 23) is set to `NULL` to indicate an empty list. The program's primary linked-list functions are `insert` (Section 12.4.1) and `delete` (Section 12.4.2).

12.4.1 Function `insert`

For this example, we insert characters in the list in alphabetical order. Function `insert` (lines 81–110) receives as an argument the *address of the pointer to the list's first node* and a character to insert. This enables `insert` to modify the caller's pointer to the list's first node to point to a new first node when a data item is placed at the front of the list. So we pass the *pointer* by reference. Passing a pointer's address creates a **pointer to a pointer**—this is sometimes called **double indirection**. This is a complex notion that requires careful programming.

```
80  // insert a new value into the list in sorted order
81  void insert(ListNodePtr *sPtr, char value) {
82     ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node
83
84     if (newPtr != NULL) { // is space available?
85        newPtr->data = value; // place value in node
86        newPtr->nextPtr = NULL; // node does not link to another node
87
88        ListNodePtr previousPtr = NULL;
89        ListNodePtr currentPtr = *sPtr;
90
91        // loop to find the correct location in the list
92        while (currentPtr != NULL && value > currentPtr->data) {
93           previousPtr = currentPtr; // walk to ...
94           currentPtr = currentPtr->nextPtr; // ... next node
95        }
96
97        // insert new node at beginning of list
98        if (previousPtr == NULL) {
99           newPtr->nextPtr = *sPtr;
100          *sPtr = newPtr;
101       }
102       else { // insert new node between previousPtr and currentPtr
103          previousPtr->nextPtr = newPtr;
104          newPtr->nextPtr = currentPtr;
105       }
106    }
107    else {
108       printf("%c not inserted. No memory available.\n", value);
109    }
110 }
111
```

The `insert` function performs the following steps:

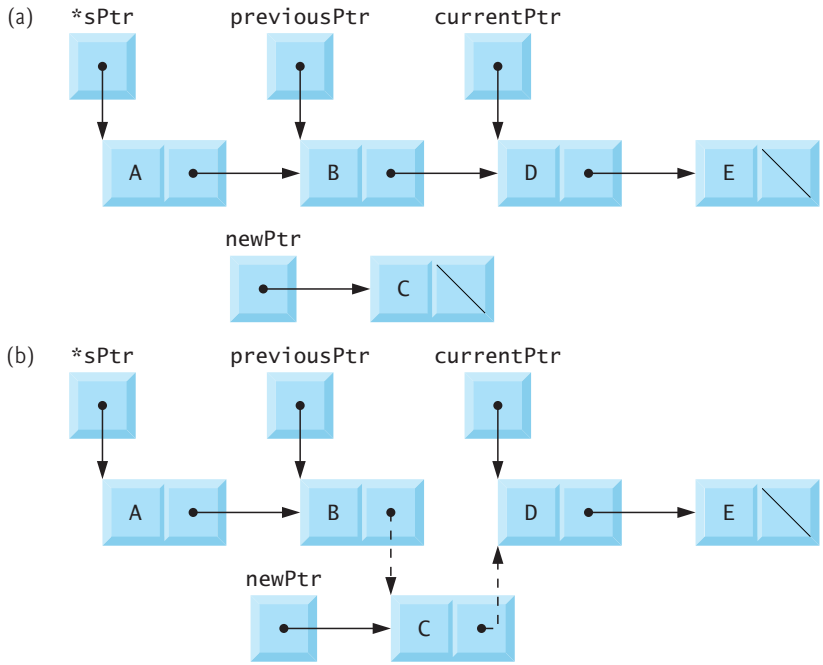
1. Call `malloc` to create a new node and assign `newPtr` the allocated memory's address (line 82).
2. If the memory was allocated, assign the character to insert to `newPtr->data` (line 85), and assign `NULL` to `newPtr->nextPtr` (line 86). Always assign `NULL` to a new node's link member. Pointers should be initialized before they're used.
3. We'll use pointers `previousPtr` and `currentPtr` to store the locations of the node *preceding* and *after* the insertion point, respectively. Initialize `previousPtr` to `NULL` (line 88) and `currentPtr` to `*sPtr` (line 89)—the first node's address.
4. Locate the new value's insertion point. While `currentPtr` is not `NULL` and the value to insert is greater than `currentPtr->data` (line 92), assign `currentPtr` to `previousPtr` (line 93), then advance `currentPtr` to the list's next node (line 94).
5. Insert the new value in the list. If `previousPtr` is `NULL` (line 98), insert the new node as the *first* in the list (lines 99–100). Assign `*sPtr` to `newPtr->nextPtr` (the new node's link points to the former first node), and assign `newPtr` to `*sPtr` so `startPtr` in `main` points to the new first node. Otherwise, insert the

new node in place (lines 103–104). Assign `newPtr` to `previousPtr->nextPtr` (the previous node points to the new node) and assign `currentPtr` to `newPtr->nextPtr` (the *new* node link points to the *current* node).

For simplicity, we implemented function `insert` (and other similar functions in this chapter) with a `void` return type. Function `malloc` may *fail* to allocate the requested memory. In this case, it would be better for our `insert` function to return a status that indicates whether the operation was successful.

Illustrating an Insert

The following diagram illustrates inserting a node containing `'C'` into an ordered list. Part (a) shows the list and the new node just before the insertion. Part (b) shows the result of inserting the new node. Dotted arrows represent the reassigned pointers.



12.4.2 Function `delete`

Function `delete` (lines 113–141) receives the address of the pointer to the list's first node and a character to delete.

```
112 // delete a list element
113 char delete(ListNodePtr *sPtr, char value) {
114    // delete first node if a match is found
115    if (value == (*sPtr)->data) {
116       ListNodePtr tempPtr = *sPtr; // hold onto node being removed
117       *sPtr = (*sPtr)->nextPtr; // de-thread the node
118       free(tempPtr); // free the de-threaded node
119       return value;
120    }
```

```
121    else {
122       ListNodePtr previousPtr = *sPtr;
123       ListNodePtr currentPtr = (*sPtr)->nextPtr;
124
125       // loop to find the correct location in the list
126       while (currentPtr != NULL && currentPtr->data != value) {
127          previousPtr = currentPtr; // walk to ...
128          currentPtr = currentPtr->nextPtr; // ... next node
129       }
130
131       // delete node at currentPtr
132       if (currentPtr != NULL) {
133          ListNodePtr tempPtr = currentPtr;
134          previousPtr->nextPtr = currentPtr->nextPtr;
135          free(tempPtr);
136          return value;
137       }
138    }
139
140    return '\0';
141 }
142
```

The `delete` function performs the following steps:

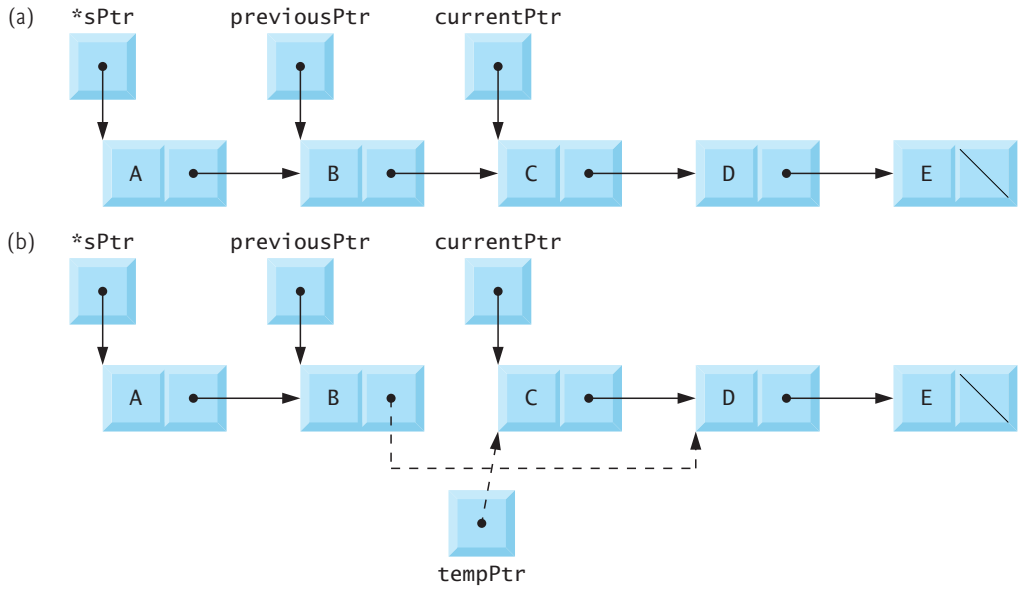
1. If the character to delete matches the first node's character (line 115), we must remove the first node. Assign `*sPtr` to `tempPtr`, which we'll use to `free` the node's memory. Assign `(*sPtr)->nextPtr` to `*sPtr`, so that `startPtr` in `main` now points to what was previously the list's *second* node. Call `free` to deallocate the memory pointed to by `tempPtr`. Return the character that was deleted.
2. Otherwise, initialize `previousPtr` with `*sPtr` and initialize `currentPtr` with `(*sPtr)->nextPtr` (lines 122–123) to advance to the second node.
3. Locate the character to delete. While `currentPtr` is not `NULL` and the value to delete is not equal to `currentPtr->data` (line 126), assign `currentPtr` to `previousPtr` (line 127) and assign `currentPtr->nextPtr` to `currentPtr` (line 128) to advance to the list's next node.
4. If `currentPtr` is not `NULL` (line 132), the character is in the list. Assign `currentPtr` to `tempPtr` (line 133), which we'll use to deallocate the node. Assign `currentPtr->nextPtr` to `previousPtr->nextPtr` (line 134) to connect the node before and the node after the one being removed. Free the node pointed to by `tempPtr` (line 135), then return the deleted character (line 136).

If nothing has been returned yet, line 140 returns the null character (`'\0'`) to signify that the character was not found in the list.

Illustrating a Delete

The following diagram illustrates deleting the node containing `'C'` from a linked list. Part (a) shows the linked list before the deletion. Part (b) shows the link reassignments. Pointer `tempPtr` is used to free the memory allocated to the node that stores

'C'. Note that in lines 118 and 135, we free tempPtr. Previously, we recommended setting a freed pointer to NULL. We do not do that here because tempPtr is a local automatic variable, and the function returns immediately after freeing the memory.



12.4.3 Functions isEmpty and printList

Function isEmpty (lines 144–146) is a predicate function—it does not alter the list in any way. Rather, isEmpty determines whether the list is empty—that is, the pointer to the first node is NULL. If the list is empty, isEmpty returns 1; otherwise, it returns 0.

```
143 // return 1 if the list is empty, 0 otherwise
144 int isEmpty(ListNodePtr sPtr) {
145    return sPtr == NULL;
146 }
147
148 // print the list
149 void printList(ListNodePtr currentPtr) {
150    // if list is empty
151    if (isEmpty(currentPtr)) {
152       puts("List is empty.\n");
153    }
154    else {
155       puts("The list is:");
156
157       // while not the end of the list
158       while (currentPtr != NULL) {
159          printf("%c --> ", currentPtr->data);
160          currentPtr = currentPtr->nextPtr;
161       }
162
163       puts("NULL\n");
164    }
165 }
```

Function `printList` (lines 149–165) prints a list. The function's `currentPtr` parameter receives a pointer to the list's first node. The function first determines whether the list is *empty* (lines 151–153) and, if so, prints `"List is empty."` and terminates. Otherwise, lines 155–163 print the list's data. While `currentPtr` is not `NULL`, line 159 prints the value in `currentPtr->data`, and line 160 assigns `currentPtr->nextPtr` to `currentPtr` to advance to the next node. If the link in the last node of the list is not `NULL`, the printing algorithm will try to print past the end of the list, which is a logic error. This printing algorithm is identical for linked lists, stacks and queues.

List-Based Recursion Exercises

Exercise 12.17 asks you to implement a recursive function that prints a list backward. Exercise 12.18 asks you to implement a recursive function that searches a linked list for a particular data item.

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) A linked list is a linear collection of self-referential structures, called nodes, connected by pointer links—hence, the term "linked" list.
b) A linked list is accessed via a pointer to the first node of the list.
c) Subsequent nodes in a linked list are accessed via the link pointer member stored in each node.
d) All of the above statements are *true*.

Answer: d.

2 *(True/False)* A linked list is appropriate when the number of data items to store in the data structure is unpredictable.
Answer: *True*.

3 *(True/False)* The elements in a linked list are stored contiguously in memory. This allows immediate access to any elements because its address can be calculated directly, based on its position relative to the beginning of the linked list. Arrays do not afford such immediate access to their elements.
Answer: *False*. Actually, the reverse is true. An array's elements are stored contiguously in memory and support immediate access by position. Linked lists do not afford such immediate access to their elements.

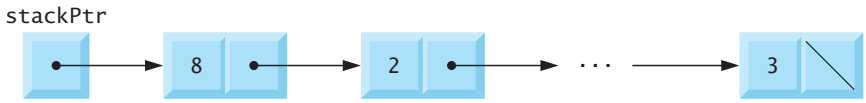
4 *(Fill-In)* Passing into a function the address of the pointer to a linked list's first node creates a pointer to a pointer. This is often called double _____.
Answer: indirection.

12.5 Stacks

A stack can be implemented as a constrained version of a linked list. You add new nodes and remove existing ones only at the top. For this reason, a stack is referred to

as a **last-in, first-out (LIFO)** data structure. You access a stack via a pointer to its top element. The link member in the stack's last node is set to `NULL` to indicate the stack's bottom. Not terminating the stack with `NULL` can lead to runtime errors.

The following diagram illustrates a stack with several nodes—`stackPtr` points to the stack's top element. We represent stacks and linked lists in these figures identically. The difference between them is that insertions and deletions may occur anywhere in a linked list, but only at the top of a stack.



Primary Stack Operations

A stack's primary functions are `push` and `pop`. Function `push` creates a new node and places it on top of the stack. Function `pop` removes a node from the stack's top, frees that node's memory and returns the popped value.

Implementing a Stack

Figure 12.2 implements a simple stack of integers. The program allows you to push a value onto the stack (function `push`), pop a value off the stack (function `pop`) and terminate the program. Lines 7–10 define `struct stackNode`, which we'll use to represent the stack's nodes. As in Fig. 12.1, we use `typedef`s (lines 12–13) to make the code more readable. Initially, `stackPtr` (line 23) is set to `NULL` to indicate an empty stack. Much of this application's logic is similar to Fig. 12.1, so we focus on the differences here.

```
1  // fig12_02.c
2  // A simple stack program
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  // self-referential structure
7  struct stackNode {
8     int data; // define data as an int
9     struct stackNode *nextPtr; // stackNode pointer
10 };
11
12 typedef struct stackNode StackNode; // synonym for struct stackNode
13 typedef StackNode *StackNodePtr; // synonym for StackNode*
14
15 // prototypes
16 void push(StackNodePtr *topPtr, int info);
17 int pop(StackNodePtr *topPtr);
18 int isEmpty(StackNodePtr topPtr);
19 void printStack(StackNodePtr currentPtr);
20 void instructions(void);
21
```

Fig. 12.2 | A simple stack program. (Part 1 of 4.)

```
22 int main(void) {
23    StackNodePtr stackPtr = NULL; // points to stack top
24    int value = 0; // int input by user
25
26    instructions(); // display the menu
27    printf("%s", "? ");
28    int choice = 0; // user's menu choice
29    scanf("%d", &choice);
30
31    // while user does not enter 3
32    while (choice != 3) {
33       switch (choice) {
34          case 1: // push value onto stack
35             printf("%s", "Enter an integer: ");
36             scanf("%d", &value);
37             push(&stackPtr, value);
38             printStack(stackPtr);
39             break;
40          case 2: // pop value off stack
41             // if stack is not empty
42             if (!isEmpty(stackPtr)) {
43                printf("The popped value is %d.\n", pop(&stackPtr));
44             }
45
46             printStack(stackPtr);
47             break;
48          default:
49             puts("Invalid choice.\n");
50             instructions();
51             break;
52       }
53
54       printf("%s", "? ");
55       scanf("%d", &choice);
56    }
57
58    puts("End of run.");
59 }
60
61 // display program instructions to user
62 void instructions(void) {
63    puts("Enter choice:\n"
64       "1 to push a value on the stack\n"
65       "2 to pop a value off the stack\n"
66       "3 to end program");
67 }
68
69 // insert a node at the stack top
70 void push(StackNodePtr *topPtr, int info) {
71    StackNodePtr newPtr = malloc(sizeof(StackNode));
72
```

Fig. 12.2 | A simple stack program. (Part 2 of 4.)

```
73     // insert the node at stack top
74     if (newPtr != NULL) {
75        newPtr->data = info;
76        newPtr->nextPtr = *topPtr;
77        *topPtr = newPtr;
78     }
79     else { // no space available
80        printf("%d not inserted. No memory available.\n", info);
81     }
82  }
83
84  // remove a node from the stack top
85  int pop(StackNodePtr *topPtr) {
86     StackNodePtr tempPtr = *topPtr;
87     int popValue = (*topPtr)->data;
88     *topPtr = (*topPtr)->nextPtr;
89     free(tempPtr);
90     return popValue;
91  }
92
93  // print the stack
94  void printStack(StackNodePtr currentPtr) {
95     if (currentPtr == NULL) { // if stack is empty
96        puts("The stack is empty.\n");
97     }
98     else {
99        puts("The stack is:");
100
101       while (currentPtr != NULL) { // while not the end of the stack
102          printf("%d --> ", currentPtr->data);
103          currentPtr = currentPtr->nextPtr;
104       }
105
106       puts("NULL\n");
107    }
108 }
109
110 // return 1 if the stack is empty, 0 otherwise
111 int isEmpty(StackNodePtr topPtr) {
112    return topPtr == NULL;
113 }
```

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1

Enter an integer: 5
The stack is:
5 --> NULL
```

Fig. 12.2 | A simple stack program. (Part 3 of 4.)

```
? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

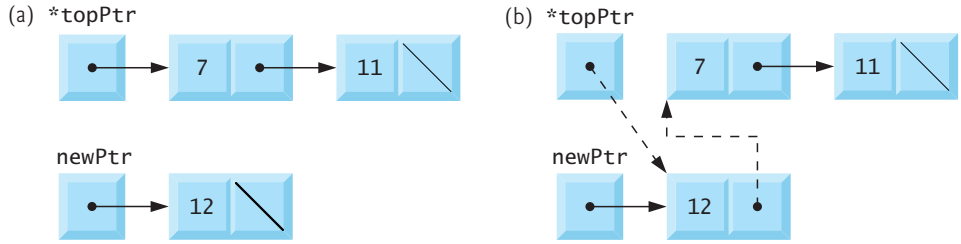
Fig. 12.2 | A simple stack program. (Part 4 of 4.)

12.5.1 Function push

Function `push` (lines 70–82) places a new node onto the stack using the following steps:

1. Call `malloc` to create a new node, then assign the allocated memory's address to `newPtr` (line 71).
2. Assign to `newPtr->data` the value to push onto the stack (line 75) and assign `*topPtr` (the pointer to the stack's top) to `newPtr->nextPtr` (line 76). The link member of the new top node now points to the previous top node.
3. Assign `newPtr` to `*topPtr` (line 77)—this modifies `stackPtr` in `main` to point to the new stack top.

The following diagram illustrates a `push` operation. Part (a) shows the stack and the new node before the `push` operation inserts the new node at the stack's top—`*topPtr` represents `stackPtr` in `main`. The dotted arrows in Part (b) illustrate *Steps 2* and *3* of the preceding discussion, which insert the node containing `12` at the top.

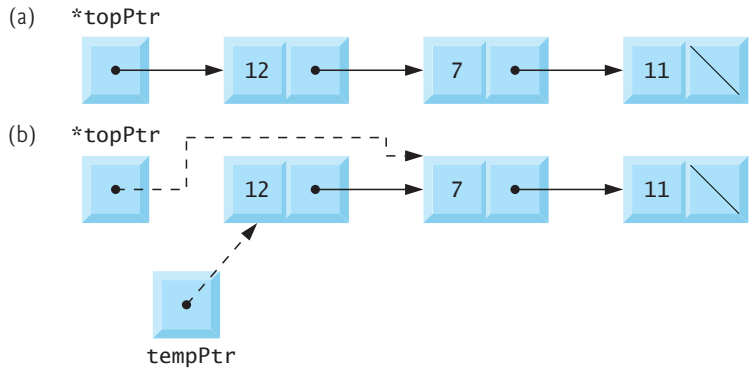


12.5.2 Function pop

Function `pop` (lines 85–91) removes the stack's top node. Function `main` determines whether the stack is empty before calling `pop`. The `pop` operation consists of five steps:

1. Assign `*topPtr` to `tempPtr` (line 86), which will be used to free the node's memory.
2. Assign `(*topPtr)->data` to `popValue` (line 87) to save the top node's value so we can return it.
3. Assign `(*topPtr)->nextPtr` to `*topPtr` (line 88) so that `stackPtr` in `main` now points to what was previously the stack's second element (or `NULL` if there were no other elements).
4. Free the memory pointed to by `tempPtr` (line 89).
5. Return `popValue` to the caller (line 90).

The following diagram illustrates a `pop` operation. Part (a) shows the stack before removing the node containing `12`—`*topPtr` represents `stackPtr` in `main`. Part (b) shows `tempPtr` pointing to the node being popped and `*topPtr` pointing to the new top node. Then we can `free` the memory to which `tempPtr` points.



12.5.3 Applications of Stacks

Stacks have many interesting applications. For example, whenever a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack (Section 5.7). In a series of function calls, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks contain the space created for automatic local variables on each invocation of a function. When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables are no longer known to the program. Stacks also are sometimes used by compilers in the process of evaluating expressions and generating machine-language code. The exercises explore several stack applications.

Self Check

1 ***(Multiple Choice)*** Which of the following statements is *false*?

a) A stack must be implemented as a constrained version of a linked list.
b) New nodes are added to and removed from a stack only at the top.
c) A stack is a last-in, first-out (LIFO) data structure.
d) A stack is referenced via a pointer to its top element.

Answer: (a) is *false*. Actually, a stack *can* be implemented as a constrained version of a linked list, but it need not be.

2 ***(Multiple Choice)*** Which of the following statements a), b) or c) is *false*?

a) When a function is called, it must know how to return to its caller, so the called function's return address is pushed onto the stack.
b) In a series of function calls, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller.
c) Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
d) All of the above statements are *true*.

Answer: (a) is *false*. Actually, the *caller's* return address is pushed onto the stack.

12.6 Queues

A **queue** is similar to a checkout line in a grocery store:

- The first person in line receives service first.
- Other customers enter the line only at the end and wait for service.

You remove queue nodes only from its **head** (front) and insert nodes only at its **tail** (back). For this reason, a queue is referred to as a **first-in, first-out (FIFO)** data structure. The insert and remove operations are known as `enqueue` (pronounced "en-cue") and `dequeue` (pronounced "dee-cue"), respectively.

Queue Applications

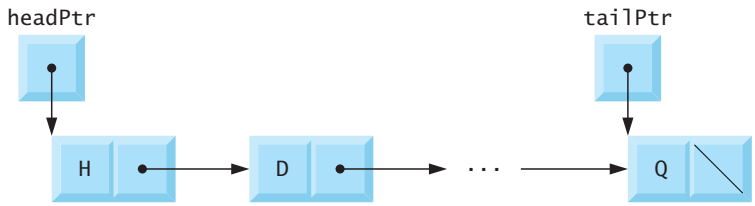
Queues have many applications in computer systems:

- For computers that have only a single processor, only one user at a time may be serviced. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queue as users receive service. The entry at the front of the queue is the next to receive service.

- Similarly, for today's multicore systems, there could be more programs running than there are processors. The programs not currently running are placed in a queue until a currently busy processor becomes available. In Appendix C, we discuss multithreading. When a program's work is divided into multiple threads capable of executing in parallel, there could be more threads than processors. The threads not currently running need to wait in a queue.
- Queues also support print spooling. An office may have only one printer. Many users can send documents to print at a given time. When the printer is busy, additional documents are spooled to memory or secondary storage, just as sewing thread is wrapped around a spool until it's needed. The documents wait in a queue until the printer becomes available.
- Information packets traveling over computer networks, like the Internet, also wait in queues. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to its final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

Illustrating a Queue

The following diagram illustrates a queue with several nodes. Note the separate pointers to the queue's head and tail. As with lists and stacks, not setting the link in the queue's last node to `NULL` can lead to logic errors.



Implementing a Queue

Figure 12.3 performs queue manipulations. The program provides options to insert a node in the queue (function **enqueue**), remove a node from the queue (function **dequeue**) and terminate the program. Lines 7–10 define `struct queueNode`, which we'll use to represent the queue's nodes. Again, we use `typedefs` (lines 12–13) to make the code more readable. Much of the logic in this application is similar to our list and stack examples, so we focus on the differences here. Initially, `headPtr` and `tailPtr` (lines 23–24) are both `NULL` to indicate an empty queue.

```
1  // fig12_03.c
2  // Operating and maintaining a queue
3  #include <stdio.h>
4  #include <stdlib.h>
5
```

Fig. 12.3 | Operating and maintaining a queue. (Part 1 of 5.)

```
// self-referential structure
struct queueNode {
   char data; // define data as a char
   struct queueNode *nextPtr; // queueNode pointer
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

// function prototypes
void printQueue(QueueNodePtr currentPtr);
int isEmpty(QueueNodePtr headPtr);
void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value);
char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
void instructions(void);

int main(void) {
   QueueNodePtr headPtr = NULL; // initialize headPtr
   QueueNodePtr tailPtr = NULL; // initialize tailPtr
   char item = '\0'; // char input by user

   instructions(); // display the menu
   printf("%s", "? ");
   int choice = 0; // user's menu choice
   scanf("%d", &choice);

   // while user does not enter 3
   while (choice != 3) {
      switch(choice) {
         case 1: // enqueue value
            printf("%s", "Enter a character: ");
            scanf("\n%c", &item);
            enqueue(&headPtr, &tailPtr, item);
            printQueue(headPtr);
            break;
         case 2: // dequeue value
            // if queue is not empty
            if (!isEmpty(headPtr)) {
               item = dequeue(&headPtr, &tailPtr);
               printf("%c has been dequeued.\n", item);
            }

            printQueue(headPtr);
            break;
         default:
            puts("Invalid choice.\n");
            instructions();
            break;
      }

      printf("%s", "? ");
      scanf("%d", &choice);
   }
```

Fig. 12.3 | Operating and maintaining a queue. (Part 2 of 5.)

```
59
60      puts("End of run.");
61   }
62
63   // display program instructions to user
64   void instructions(void) {
65      printf ("Enter your choice:\n"
66              "   1 to add an item to the queue\n"
67              "   2 to remove an item from the queue\n"
68              "   3 to end\n");
69   }
70
71   // insert a node at queue tail
72   void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value) {
73      QueueNodePtr newPtr = malloc(sizeof(QueueNode));
74
75      if (newPtr != NULL) { // is space available?
76         newPtr->data = value;
77         newPtr->nextPtr = NULL;
78
79         // if empty, insert node at head
80         if (isEmpty(*headPtr)) {
81            *headPtr = newPtr;
82         }
83         else {
84            (*tailPtr)->nextPtr = newPtr;
85         }
86
87         *tailPtr = newPtr;
88      }
89      else {
90         printf("%c not inserted. No memory available.\n", value);
91      }
92   }
93
94   // remove node from queue head
95   char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr) {
96      char value = (*headPtr)->data;
97      QueueNodePtr tempPtr = *headPtr;
98      *headPtr = (*headPtr)->nextPtr;
99
100     // if queue is empty
101     if (*headPtr == NULL) {
102        *tailPtr = NULL;
103     }
104
105     free(tempPtr);
106     return value;
107  }
108
```

Fig. 12.3 | Operating and maintaining a queue. (Part 3 of 5.)

```
// return 1 if the queue is empty, 0 otherwise
int isEmpty(QueueNodePtr headPtr) {
   return headPtr == NULL;
}

// print the queue
void printQueue(QueueNodePtr currentPtr) {
   if (currentPtr == NULL) { // if queue is empty
      puts("Queue is empty.\n");
   }
   else {
      puts("The queue is:");

      while (currentPtr != NULL) { // while not end of queue
         printf("%c --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      puts("NULL\n");
   }
}
```

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end

? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL
```

Fig. 12.3 | Operating and maintaining a queue. (Part 4 of 5.)

```
? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Fig. 12.3 | Operating and maintaining a queue. (Part 5 of 5.)

12.6.1 Function enqueue

Function `enqueue` (lines 72–92) receives three arguments:

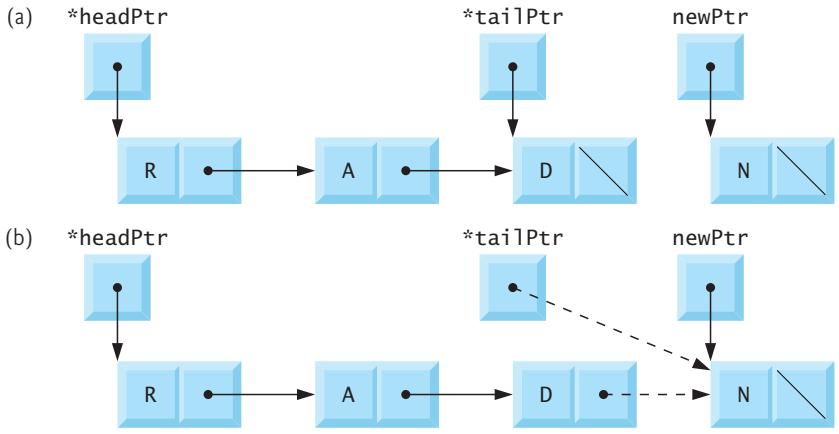
- the address of `headPtr`—the pointer to the queue's head,
- the address of `tailPtr`—the pointer to the queue's tail, and
- the value to insert.

The function performs the following steps:

1. Line 73 calls `malloc` to create a new node and assigns the allocated memory location to `newPtr`.
2. If the memory was allocated correctly, lines 76–77 assign the value to insert to `newPtr->data`, and assign `NULL` to `newPtr->nextPtr`.
3. If the queue is empty (line 80), line 81 assigns `newPtr` to `*headPtr` because the new node is both the queue's head and tail; otherwise, line 84 assigns `newPtr` to `(*tailPtr)->nextPtr` because the new node is the new tail node.
4. Line 87 assigns `newPtr` to `*tailPtr` to update the tail pointer to the new tail node.

Illustrating an enqueue Operation

The following diagram illustrates an `enqueue` operation. Part (a) shows `main`'s `headPtr` and `tailPtr` and `enqueue`'s new node before the operation. The dotted arrows in Part (b) illustrate *Steps 3* and *4* in the preceding discussion, which add the new node to the tail of a non-empty queue.



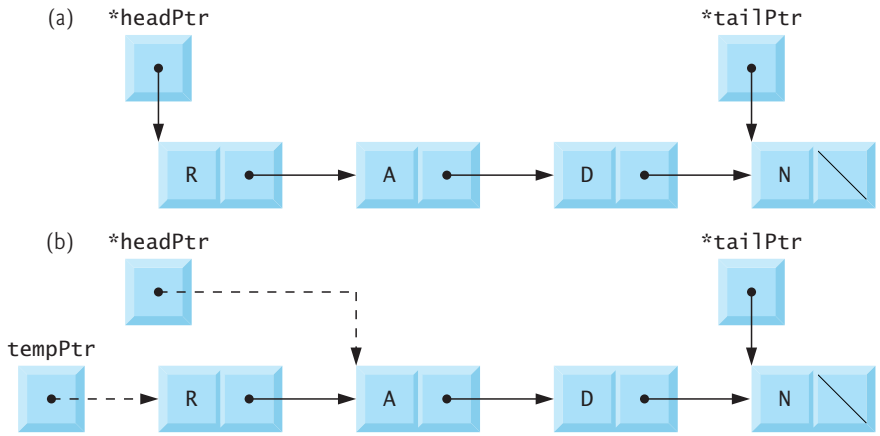
12.6.2 Function dequeue

Function `dequeue` (lines 95–107) receives as arguments the addresses of the queue's head and tail pointers and removes the queue's first node. The `dequeue` operation performs the following steps:

1. Line 96 assigns `(*headPtr)->data` to `value` to save the data being dequeued.
2. Line 97 assigns `*headPtr` to `tempPtr`, which will be used to `free` the memory.
3. Line 98 assigns `(*headPtr)->nextPtr` to `*headPtr` so that the queue's head pointer in `main` now points to the new head node.
4. Line 101 checks whether `*headPtr` is `NULL`. If so, line 102 assigns `NULL` to `*tailPtr` because the queue is now empty.
5. Line 105 frees the memory pointed to by `tempPtr`.
6. Line 106 return `value` to the caller.

Illustrating a dequeue Operation

The following diagram illustrates function `dequeue`. Part (a) shows the queue before the `dequeue` operation—`*headPtr` and `*tailPtr` in `dequeue` are used to modify `main`'s `headPtr` and `tailPtr`. Part (b) shows `tempPtr` pointing to the dequeued node, and `main`'s `headPtr` updated to point to the queue's new first node.



Self Check

1 *(Multiple Choice)* Which of the following statements about queues is *false*?

a) A queue is similar to a checkout line in a grocery store—the first person in line receives service first, and other customers enter the line at the end and wait for service.

b) Queue nodes are removed only from the queue's head (start) and inserted only at the queue's tail (end).

c) A queue is a first-in, last-out (FILO) data structure.

d) The insert and remove operations are known as enqueue (pronounced "en-cue") and dequeue (pronounced "dee-cue"), respectively.

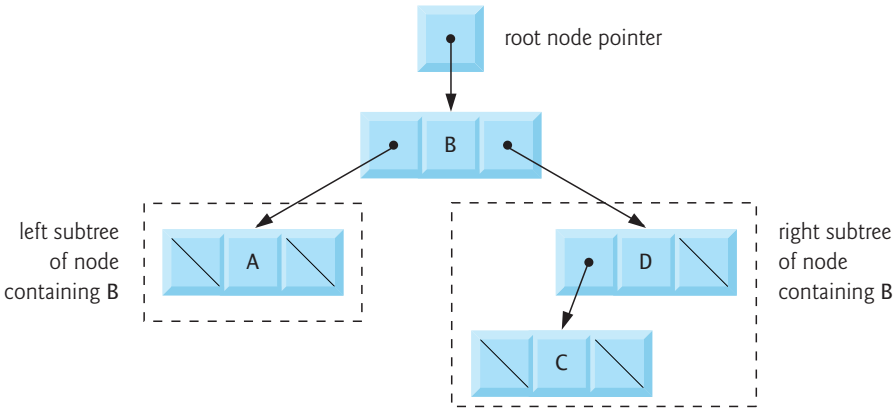
Answer: (c) is *false*. Actually, a queue is a first-in, first-out (FIFO) data structure.

2 *(Fill-In)* For today's _____ systems, there could be more programs running than there are processors, so the programs not currently running are placed in a queue until a currently busy processor becomes available.

Answer: multicore.

12.7 Trees

So far, we've presented linear data structures—linked lists, stacks and queues. A **tree** is a nonlinear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses **binary trees**—trees whose nodes contain two links, as in the following diagram:



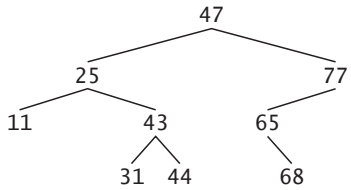
None, one, or both of the links in each node may be `NULL`. The **root node** is the first node in a tree. Each link in the root node refers to a **child**. The **left child** is the first node in the **left subtree**, and the **right child** is the first in the **right subtree**. A given node's children are called **siblings**. A node with no children is a **leaf node**. Not setting a leaf node's links to `NULL` can lead to runtime errors. Computer scientists typically draw trees with the root node at the top—exactly the opposite of trees in nature.

ERR

Binary Search Tree

This section presents a **binary search tree** containing unique values, which has the characteristic that the values in any *left* subtree are less than the value in its **parent**

node, and the values in any *right* subtree are greater than the value in its parent node. The figure below illustrates a binary search tree with nine values:



The shape of the binary search tree for a set of data can vary, based on the order in which you insert its values.

Implementing a Binary Search Tree

Figure 12.4 creates a binary search tree and traverses its nodes—that is, it visits each node in the tree to do something with the node values, like display them. We'll traverse the tree three ways—**inorder**, **preorder** and **postorder**. The program generates 10 random numbers and inserts each in the tree, except that we discard duplicate values. Lines 9–13 define `struct treeNode`, which we'll use to represent the tree's nodes. Again, we use `typedefs` (lines 15–16) to make the code more readable.

```
1  // fig12_04.c
2  // Creating and traversing a binary tree
3  // preorder, inorder, and postorder
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  // self-referential structure
9  struct treeNode {
10    struct treeNode *leftPtr; // pointer to left subtree
11    int data; // node value
12    struct treeNode *rightPtr; // pointer to right subtree
13 };
14
15 typedef struct treeNode TreeNode; // synonym for struct treeNode
16 typedef TreeNode *TreeNodePtr; // synonym for TreeNode *
17
18 // prototypes
19 void insertNode(TreeNodePtr *treePtr, int value);
20 void inOrder(TreeNodePtr treePtr);
21 void preOrder(TreeNodePtr treePtr);
22 void postOrder(TreeNodePtr treePtr);
23
24 int main(void) {
25    TreeNodePtr rootPtr = NULL; // tree initially empty
26
```

Fig. 12.4 | Creating and traversing a binary tree. (Part 1 of 3.)

```
27     srand(time(NULL));
28     puts("The numbers being placed in the tree are:");
29
30     // insert random values between 0 and 14 in the tree
31     for (int i = 1; i <= 10; ++i) {
32        int item = rand() % 15;
33        printf("%3d", item);
34        insertNode(&rootPtr, item);
35     }
36
37     // traverse the tree preOrder
38     puts("\n\nThe preOrder traversal is:");
39     preOrder(rootPtr);
40
41     // traverse the tree inOrder
42     puts("\n\nThe inOrder traversal is:");
43     inOrder(rootPtr);
44
45     // traverse the tree postOrder
46     puts("\n\nThe postOrder traversal is:");
47     postOrder(rootPtr);
48  }
49
50  // insert node into tree
51  void insertNode(TreeNodePtr *treePtr, int value) {
52     if (*treePtr == NULL) { // if tree is empty
53        *treePtr = malloc(sizeof(TreeNode));
54
55        if (*treePtr != NULL) { // if memory was allocated, then assign data
56           (*treePtr)->data = value;
57           (*treePtr)->leftPtr = NULL;
58           (*treePtr)->rightPtr = NULL;
59        }
60        else {
61           printf("%d not inserted. No memory available.\n", value);
62        }
63     }
64     else { // tree is not empty
65        if (value < (*treePtr)->data) { // value goes in left subtree
66           insertNode(&((*treePtr)->leftPtr), value);
67        }
68        else if (value > (*treePtr)->data) { // value goes in right subtree
69           insertNode(&((*treePtr)->rightPtr), value);
70        }
71        else { // duplicate data value ignored
72           printf("%s", "dup");
73        }
74     }
75  }
76
```

Fig. 12.4 | Creating and traversing a binary tree. (Part 2 of 3.)

```
77  // begin inorder traversal of tree
78  void inOrder(TreeNodePtr treePtr) {
79     // if tree is not empty, then traverse
80     if (treePtr != NULL) {
81        inOrder(treePtr->leftPtr);
82        printf("%3d", treePtr->data);
83        inOrder(treePtr->rightPtr);
84     }
85  }
86
87  // begin preorder traversal of tree
88  void preOrder(TreeNodePtr treePtr) {
89     // if tree is not empty, then traverse
90     if (treePtr != NULL) {
91        printf("%3d", treePtr->data);
92        preOrder(treePtr->leftPtr);
93        preOrder(treePtr->rightPtr);
94     }
95  }
96
97  // begin postorder traversal of tree
98  void postOrder(TreeNodePtr treePtr) {
99     // if tree is not empty, then traverse
100    if (treePtr != NULL) {
101       postOrder(treePtr->leftPtr);
102       postOrder(treePtr->rightPtr);
103       printf("%3d", treePtr->data);
104    }
105 }
```

```
The numbers being placed in the tree are:
  6  7  4 12  7dup  2  2dup  5  7dup 11

The preOrder traversal is:
  6  4  2  5  7 12 11

The inOrder traversal is:
  2  4  5  6  7 11 12

The postOrder traversal is:
  2  5  4 11 12  7  6
```

Fig. 12.4 | Creating and traversing a binary tree. (Part 3 of 3.)

12.7.1 Function `insertNode`

The functions in Fig. 12.4 that create a binary search tree and traverse it are recursive. Function `insertNode` (lines 51–75) receives as arguments the address of the pointer to the tree's root node and an integer to insert. Each new node in a binary search tree initially is inserted as a leaf node. The steps for inserting a new node are as follows:

1. If `*treePtr` is `NULL` (line 52), line 53 calls `malloc` to create a new leaf node and assigns the allocated memory to `*treePtr`. Lines 56 assigns to `(*treePtr)-`

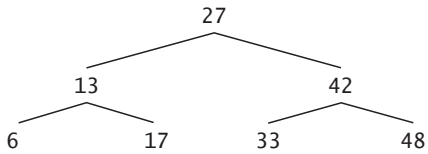
>data the integer to store. Lines 57–58 assign NULL to (*treePtr)->leftPtr and (*treePtr)->rightPtr. Then control returns to the caller—either main or a previous call to insertNode.

2. If *treePtr is not NULL and the value to insert is less than (*treePtr)->data, line 66 recursively calls insertNode with the address of (*treePtr)->leftPtr to insert the new node in the left subtree of the node pointed to by treePtr.
3. If the value to insert is greater than (*treePtr)-> data, line 69 recursively calls insertNode with the address of (*treePtr)->rightPtr to insert the node in the right subtree of the node pointed to by treePtr.

The recursive steps continue until insertNode finds a NULL pointer, then *Step 1* inserts the new node as a leaf node.

12.7.2 Traversals: Functions inOrder, preOrder and postOrder

Functions inOrder (lines 78–85), preOrder (lines 88–95) and postOrder (lines 98–105) each receive a pointer to a tree's root node and traverse the tree. Below we describe the traversals and show the result of applying each for the following tree:



inOrder Traversal

The steps for an inOrder traversal are:

1. Traverse the *left* subtree inOrder (line 81).
2. Process the value in the node (line 82).
3. Traverse the *right* subtree inOrder (line 83).

This traversal processes each node's value after processing the values in the left subtree. The inOrder traversal of the preceding tree is:

```
6 13 17 27 33 42 48
```

A binary search tree's inOrder traversal processes the nodes in ascending order. Creating a binary search tree actually sorts the data. So, this process is called a **binary tree sort**.

preOrder Traversal

The steps for a preOrder traversal are:

1. Process the value in the node (line 91).
2. Traverse the *left* subtree preOrder (line 92).
3. Traverse the *right* subtree preOrder (line 93).

This traversal processes each node's value as the node is visited. After processing the value, a `preOrder` traversal processes the left subtree's values, then the right subtree's values. The `preOrder` traversal of the preceding tree is:

```
27 13 6 17 42 33 48
```

`postOrder` Traversal

The steps for a `postOrder` traversal are:

1. Traverse the *left* subtree `postOrder` (line 101).
2. Traverse the *right* subtree `postOrder` (line 102).
3. Process the value in the node (line 103).

This traversal processes each node's value after processing the values of the node's children in both the left and right subtrees. The `postOrder` traversal of the preceding tree is:

```
6 17 13 33 48 42 27
```

12.7.3 Duplicate Elimination

Binary search trees facilitate **duplicate elimination**. As you insert values to create the tree, a duplicate value will follow the same "go left" or "go right" decisions on each comparison as the original value did. So, the duplicate eventually will be compared with a node in the tree containing the same value. At that point, the duplicate value can be ignored.

12.7.4 Binary Tree Search

Searching for a value that matches a key also is fast. If the tree is tightly packed, each level contains about twice as many elements as the previous level. So, a binary search tree with n elements would have a maximum of $\log_2 n$ levels, and thus requires a *maximum* of $\log_2 n$ comparisons to find a match or to determine that no match exists. When searching a tightly packed 1,000-element binary search tree, no more than 10 comparisons need to be made because $2^{10} > 1{,}000$. When searching a tightly packed 1,000,000-element binary search tree, no more than 20 comparisons need to be made because $2^{20} > 1{,}000{,}000$.

12.7.5 Other Binary Tree Operations

In the exercises, we present algorithms for several other binary tree operations, such as printing a binary tree in a two-dimensional tree format and performing a level order traversal of a binary tree. The level order traversal visits a tree's nodes row-by-row starting at the root node level. The nodes on each level are visited from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, creating a tree of strings and determining a binary tree's number of levels.

Self Check

1 *(True/False)* Linked lists, stacks, queues and trees are linear data structures.
Answer: *False*. Actually, trees are nonlinear, two-dimensional data structures.

2 *(Fill-In)* Three popular ways to traverse a binary search tree are _____, preorder and postorder.
Answer: inorder.

3 *(Fill-In)* The process of creating a binary search tree actually _____ the data.
Answer: sorts.

4 *(True/False)* The shape of the binary search tree that corresponds to a set of data is independent of the order in which the values are inserted into the tree.
Answer: *False*. Actually, the shape of the binary search tree that corresponds to a set of data varies, based on the order in which the values are inserted into the tree.

5 *(True/False)* A node can be inserted only as a root node in a binary search tree.
Answer: *False*. Actually, a node can be inserted only as a *leaf* node in a binary search tree.

6 *(Fill-In)* A binary search tree facilitates _____. As you insert values to create the tree, identical values follow the same "go left" or "go right" decisions on each comparison as the original value did. An identical value eventually will be compared with a node in the tree containing the same value.
Answer: duplicate elimination.

7 *(True/False)* A tightly packed binary tree with n elements would have a maximum of about $\log_2 n$ levels. Searching such a binary tree requires a maximum of about $\log_2 n$ comparisons to find a match or to determine that no match exists. So, searching a (tightly packed) 1,000,000,000-element binary search tree requires no more than 20 comparisons.
Answer: *False*. Actually, this requires no more than 30 comparisons.

12.8 Secure C Programming

Chapter 8 of the *SEI CERT C Coding Standard* is dedicated to memory-management recommendations and rules—many apply this chapter's uses of pointers and dynamic memory management. For more information, visit `https://wiki.sei.cmu.edu/`.

- MEM01-C/MEM30-C: Pointers should always be initialized with `NULL` or the address of a valid item in memory. When you use `free` to deallocate dynamically allocated memory, the pointer passed to `free` is not assigned a new value, so it still points to the memory location where the dynamically allocated memory used to be. Using such a "dangling" pointer can lead to program crashes and security vulnerabilities. When you free dynamically allocated memory, immediately assign the pointer either `NULL` or a valid address. We chose not to do this for local pointer variables that immediately go out of scope after a call to `free`.

- MEM01-C: Undefined behavior occurs when you attempt to use `free` to deallocate dynamic memory that was already deallocated—this is known as a "double `free` vulnerability." To ensure that you don't attempt to deallocate the same memory more than once, immediately set a pointer to `NULL` after the call to `free`. Freeing a `NULL` pointer has no effect.
- ERR33-C: Most standard-library functions return values that you can check to determine whether the functions performed their tasks successfully. Function `malloc`, for example, returns `NULL` if it's unable to allocate the requested memory. You should always ensure that `malloc` did not return `NULL` before attempting to use the pointer that stores `malloc`'s return value.

Self Check

1 ***(True/False)*** Pointers should always be initialized to `NULL`.
Answer: *False*. Actually, pointers should always be initialized either to `NULL` or to the address of a valid item in memory.

2 ***(True/False)*** To avoid deallocating the same memory twice and causing undefined behavior, set the `freed` pointer to `NULL`.
Answer: *True*. Freeing a `NULL` pointer does not cause undefined behavior.

3 ***(Fill-In)*** If `malloc` can't fulfill a memory-allocation request, it returns _____. Always check for this value before using `malloc`'s pointer return.
Answer: `NULL`.

Summary

Section 12.1 Introduction

- Dynamic data structures (p. 650) grow and shrink at execution time.
- Linked lists (p. 650) are collections of data items "lined up in a row"—insertions and deletions are made anywhere in a linked list.
- With stacks (p. 650), insertions and deletions are made only at the top (p. 650).
- Queues (p. 650) represent waiting lines. Insertions are made at the back (the tail; p. 650). Deletions are made from the front (the head; p. 650).
- Binary trees facilitate high-speed searching and sorting of data, efficient duplicate elimination, representing file-system directories and compiling expressions into machine language.

Section 12.2 Self-Referential Structures

- A self-referential structure (p. 651) contains a pointer member that points to a structure of the same type.
- Self-referential structures can be linked together to form lists, queues, stacks and trees.
- A `NULL` pointer indicates the end of a data structure.

Section 12.3 Dynamic Memory Management

- Creating and maintaining dynamic data structures requires dynamic memory management (p. 652).

- Functions `malloc` and `free`, and operator `sizeof`, are essential to dynamic memory allocation.
- Function `malloc` (p. 652) receives the number of bytes to be allocated and returns a `void *` pointer to the allocated memory. A `void *` pointer may be assigned to a variable of any pointer type.
- If no memory is available, `malloc` returns `NULL`.
- Function `free` (p. 652) deallocates memory so that it can be reallocated in the future.
- C also provides functions `calloc` and `realloc` for creating and modifying dynamic arrays.

Section 12.4 Linked Lists

- A linked list is a linear collection of self-referential structures, called nodes (p. 653), connected by pointer links (p. 653).
- A linked list is accessed via a pointer to the first node. Subsequent nodes are accessed via the link pointer member stored in each node.
- By convention, the link pointer in the last node of a list is set to `NULL` to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.
- A node can contain data of any type, including other `struct` objects.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Linked-list nodes are not guaranteed to be stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous.

Section 12.5 Stacks

- A stack (p. 662) can be implemented as a constrained version of a linked list. New nodes are added to and removed from a stack only at the top.
- A stack is a last-in, first-out (LIFO; p. 663) data structure.
- The primary functions used to manipulate a stack are `push` and `pop`. Function `push` creates a new node and places it on top of the stack. Function `pop` removes a node from the top of the stack, frees the memory that was allocated to the popped node and returns the popped value.
- When you call a function, it must know how to return to its caller, so the caller's return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

Section 12.6 Queues

- Queue nodes are removed only from the head of the queue and inserted only at the tail of the queue—referred to as a first-in, first-out (FIFO; p. 668) data structure.
- The insert and remove operations for a queue are known as `enqueue` and `dequeue` (p. 669).

Section 12.7 Trees

- A tree (p. 675) is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links.
- Binary trees (p. 675) are trees whose nodes all contain two links.

- The root node (p. 675) is the first node in a tree. Each link in the root node of a binary tree refers to a child (p. 675). The left child (p. 675) is the first node in the left subtree (p. 675), and the right child (p. 675) is the first node in the right subtree (p. 675). A node's children are called siblings (p. 675).
- A node with no children is called a leaf node (p. 675).
- A binary search tree (with no duplicate node values; p. 675) has the characteristic that the values in any left subtree are less than the value in its parent node (p. 676), and the values in any right subtree are greater than the value in its parent node.
- A node can be inserted only as a leaf node in a binary search tree.
- The steps for an in-order traversal are: Traverse the left subtree in-order, process the value in the node, then traverse the right subtree in-order. The value in a node is not processed until the values in its left subtree are processed.
- The in-order traversal (p. 676) of a binary search tree processes the node values in ascending order. Creating a binary search tree actually sorts the data, so this process is called the binary tree sort (p. 679).
- The steps for a pre-order traversal (p. 676) are: Process the value in the node, traverse the left subtree pre-order, then traverse the right subtree pre-order. After this traversal processes a given node's value, it processes the node's left subtree values, then its right subtree values.
- The steps for a post-order traversal (p. 676) are: Traverse the left subtree post-order, traverse the right subtree post-order, then process the value in the node. The value in each node is not processed until the values of its children are processed.
- A binary search tree facilitates duplicate elimination (p. 680). As the tree is created, an attempt to insert a duplicate value will be recognized because a duplicate will follow the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the duplicate will eventually be compared with a node in the tree containing the same value. The duplicate value may simply be discarded at this point.
- Searching a binary tree for a value that matches a key is fast. In a tightly packed tree, each level contains about twice as many elements as the previous level. A binary search tree with n elements would have a maximum of $\log_2 n$ levels, and thus it requires a maximum of $\log_2 n$ comparisons to find a match or to determine that no match exists. Searching a tightly packed 1,000-element binary search tree requires no more than 10 comparisons because $2^{10} > 1{,}000$. Searching a tightly packed 1,000,000-element binary search tree requires no more than 20 comparisons because $2^{20} > 1{,}000{,}000$.

Self-Review Exercises

12.1 Fill-In the blanks in each of the following:

a) A self-_____ structure is used to form dynamic data structures.
b) Function _____ is used to dynamically allocate memory.
c) A(n) _____ is a specialized version of a linked list in which nodes can be inserted and deleted only from the start of the list.
d) Functions that look at a linked list but do not modify it are referred to as _____.
e) A queue is referred to as a(n) _____ data structure.
f) The pointer to the next node in a linked list is referred to as a(n) _____.
g) Function _____ is used to reclaim dynamically allocated memory.

h) A(n) _____ is a specialized version of a linked list in which nodes can be inserted only at the start of the list and deleted only from the end of the list.
i) A(n) _____ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
j) A stack is referred to as a(n) _____ data structure because the last node inserted is the first node removed.
k) The nodes of a(n) _____ tree contain two link members.
l) The first node of a tree is the _____ node.
m) Each link in a tree node points to a(n) _____ or _____ of that node.
n) A tree node that has no children is called a(n) _____ node.
o) The three traversal algorithms (covered in this chapter) for a binary tree are _____, _____ and _____.

12.2 *(Discussion)* What are the differences between a linked list and a stack?

12.3 *(Discussion)* What are the differences between a stack and a queue?

12.4 Write a statement or set of statements to accomplish each of the following. Assume that all the manipulations occur in `main` (therefore, no addresses of pointer variables are needed), and assume the following definitions:

```
struct gradeNode {
   char lastName[20];
   double grade;
   struct gradeNode *nextPtr;
};

typedef struct gradeNode GradeNode;
typedef GradeNode *GradeNodePtr;
```

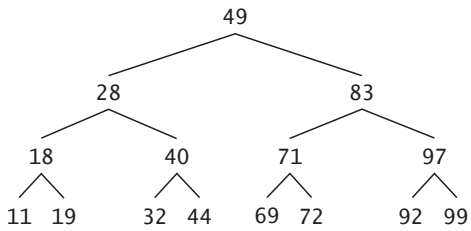
a) Create a pointer to the start of the list called `startPtr`. The list is empty.
b) Create a new node of type `GradeNode` that's pointed to by pointer `newPtr` of type `GradeNodePtr`. Assign the string `"Jones"` to member `lastName` and the value `91.5` to member `grade` (use `strcpy`). Provide any necessary declarations and statements.
c) Assume that the list pointed to by `startPtr` currently consists of two nodes—one containing `"Jones"` and one containing `"Smith"`. The nodes are in alphabetical order. Provide the statements necessary to insert nodes containing the following data for `lastName` and `grade`—be sure to insert the nodes in order:

```
"Adams"          85.0
"Thompson"       73.5
"Pritchard"      66.5
```

Use pointers `previousPtr`, `currentPtr` and `newPtr` to perform the insertions. State what `previousPtr` and `currentPtr` point to before each insertion. Assume `newPtr` always points to the new node and the new node has already been assigned the data.
d) Write a `while` loop that prints the data in each node of the list. Use pointer `currentPtr` to move along the list.

e) Write a `while` loop that deletes all the nodes in the list and frees the memory associated with each node. Use pointer `currentPtr` and pointer `tempPtr` to walk along the list and free memory, respectively.

12.5 ***(Binary Search Tree Traversals)*** Provide the inorder, preorder and postorder traversals of the following binary search tree.



Answers to Self-Review Exercises

12.1 a) referential. b) `malloc`. c) stack. d) predicates. e) FIFO. f) link. g) `free`. h) queue. i) tree. j) LIFO. k) binary. l) root. m) child, subtree. n) leaf. o) inorder, preorder, postorder.

12.2 It's possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. However, nodes in a stack may be inserted only at the top of the stack and removed only from the top of a stack.

12.3 A queue has pointers to both its head and its tail so that nodes may be inserted at the tail and deleted from the head. A stack has a single pointer to the top of the stack where both insertion and deletion of nodes is performed.

12.4 See the answers below:

a)
```
GradeNodePtr startPtr = NULL;
```

b)
```
GradeNodePtr newPtr;
newPtr = malloc(sizeof(GradeNode));
strcpy(newPtr->lastName, "Jones");
newPtr->grade = 91.5;
newPtr->nextPtr = NULL;
```

c) To insert `"Adams"`:
`previousPtr` is `NULL`, `currentPtr` points to the first element in the list.
```
newPtr->nextPtr = currentPtr;
startPtr = newPtr;
```
To insert `"Thompson"`:
`previousPtr` points to the last element in the list (containing `"Smith"`)
`currentPtr` is `NULL`.
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
To insert `"Pritchard"`:
`previousPtr` points to the node containing `"Jones"`
`currentPtr` points to the node containing `"Smith"`

```
   newPtr->nextPtr = currentPtr;
   previousPtr->nextPtr = newPtr;
d) currentPtr = startPtr;
   while (currentPtr != NULL) {
      printf("Lastname = %s\nGrade = %6.2f\n",
              currentPtr->lastName, currentPtr->grade);
      currentPtr = currentPtr->nextPtr;
   }
e) currentPtr = startPtr;
   while (currentPtr != NULL) {
      tempPtr = currentPtr;
      currentPtr = currentPtr->nextPtr;
      free(tempPtr);
   }
   startPtr = NULL;
```

12.5 The *inorder* traversal is: `11 18 19 28 32 40 44 49 69 71 72 83 92 97 99`
The *preorder* traversal is: `49 28 18 11 19 40 32 44 83 71 69 72 97 92 99`
The *postorder* traversal is: `11 19 18 32 44 40 28 69 72 71 92 99 97 83 49`

Exercises

12.6 ***(Creating a Linked List)*** Write a program that inserts 10 random integers into a linked list. After the creation of the linked list, print the integers in the linked list in the same order and the sum of all the elements in the linked list.

12.7 ***(Sorting a Linked List)*** Write a program that inserts 10 random integers into a linked list. After the creation of the linked list, sort the linked list in ascending order by manipulating the linked list pointers. [*Hint*: Use bubble sort or insertion sort.]

12.8 ***(Implement Stack Using a Linked List)*** Implement stack data structure using only a linked list. Test your implementation with examples.

12.9 ***(Implement Queue Using a Linked List)*** Implement queue data structure using only a linked list. Test your implementation with examples.

12.10 ***(Implement Stack Using Queue)*** Implement stack data structure using only queue data structure implemented in Exercise 12.8. Test your implementation with examples.

12.11 ***(Implement Queue Using Stack)*** Implement queue data structure using only stack data structure implemented in Exercise 12.9. Test your implementation with examples.

12.12 ***(Supermarket Simulation)*** Write a program that simulates a check-out line at a supermarket. The line is a queue. Customers arrive in random integer intervals of 1 to 4 minutes. Also, each customer is serviced in random integer intervals of 1 to 4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with balanced rates,

randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes) using the following algorithm:

1. Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.
2. At the first customer's arrival time:
   Determine customer's service time (random int 1–4);
   Begin servicing the customer;
   Schedule arrival time of next customer (random `int` 1–4 added to current time).
3. For each minute of the day:
   If the next customer arrives,Say so;
   Enqueue the customer;
   Schedule the arrival time of the next customer.
   If service was completed for the last customer,Say so;
   Dequeue next customer to be serviced;
   Determine customer's service completion time
   (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes and answer each of the following:

a) What's the maximum number of customers in the queue at any time?
b) What's the longest wait any one customer experiences?
c) What happens if the arrival interval is changed from 1 to 4 minutes to 1 to 3 minutes?

12.13 ***(Allowing Duplicates in a Binary Tree)*** Modify the program of Fig. 12.4 to allow the binary tree to contain duplicate values.

12.14 ***(Binary Search Tree of Strings)*** Write a program based on the program of Fig. 12.4 that inputs a line of text, tokenizes the sentence into separate words, inserts the words in a binary search tree, and prints the inorder, preorder, and postorder traversals of the tree.

Read the line of text into an array. Use `strtok` to tokenize the text. When a token is found, create a new node for the tree, assign the pointer returned by `strtok` to member `string` of the new node, and insert the node in the tree.

12.15 ***(Duplicate Elimination)*** We've seen that duplicate elimination is straightforward when creating a binary search tree. Describe how you would perform duplicate elimination using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

12.16 ***(Depth of a Binary Tree)*** Write a function `depth` that receives a binary tree and determines how many levels it has.

12.17 ***(Recursively Print a List Backward)*** Write a function `printListBackward` that recursively outputs the items in a list in reverse order. Use your function in a test program that creates a sorted list of integers and prints the list in reverse order.

12.18 ***(Recursively Search a List)*** Write a function `searchList` that recursively searches a linked list for a specified value. The function should return a pointer to the value if it's found; otherwise, `NULL` should be returned. Use your function in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

12.19 ***(Binary Tree Search)*** Write function `binaryTreeSearch` that attempts to locate a specified value in a binary search tree. The function should take as arguments a pointer to the root node of the binary tree and a search key to be located. If the node containing the search key is found, the function should return a pointer to that node; otherwise, the function should return a `NULL` pointer.

12.20 ***(Level-Order Binary Tree Traversal)*** The program of Fig. 12.4 illustrated three recursive methods of traversing a binary tree—inorder traversal, preorder traversal, and postorder traversal. This exercise presents the **level-order traversal** of a binary tree. This traversal processes the node values level-by-level from left-to-right starting at the root-node level. The level-order traversal is not a recursive algorithm. It uses the queue data structure to process the nodes in the correct order. The algorithm is as follows:

1. Insert the root node in the queue.
2. While there are nodes left in the queue,
   Get the next node in the queue.
   Print the node's value.
   If the pointer to the left child of the node is not `NULL`
      Insert the left child node in the queue.
   If the pointer to the right child of the node is not `NULL`
      Insert the right child node in the queue.

Write function `levelOrder` to perform a level order traversal of a binary tree. The function should take as an argument a pointer to the binary tree's root node. Modify the program of Fig. 12.4 to use this function. Compare the output from this function to the other traversals' outputs to see that it worked correctly. You'll need to modify and incorporate the queue-processing functions of Fig. 12.3 in this program.

12.21 ***(Printing Trees)*** Write a recursive function `outputTree` to display a binary tree. The function should output the tree row-by-row with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree in Exercise 12.5 is output as follows:

```
                99
          97
                92
     83
                72
          71
                69
49
                44
          40
                32
     28
                19
          18
                11
```

Note that the rightmost leaf node appears at the top of the output in the rightmost column, and the root node appears at the left of the output. Each column of output starts five spaces to the right of the previous column. Function `outputTree` should receive as arguments a pointer to the tree's root node and an integer `totalSpaces` representing the number of spaces preceding the value to output. This variable should start at zero so that the root node is output at the left of the screen. The function uses a modified inorder traversal to output the tree. The algorithm is as follows:

While the pointer to the current node is not `NULL`,
  Recursively call `outputTree` with the current node's right subtree and `totalSpaces + 5`.
  Use a `for` statement to count from `1` to `totalSpaces` and output spaces.
  Output the value in the current node.
  Recursively call `outputTree` with the current node's left subtree and `totalSpaces + 5`.

Special Section: Systems Software Case Study—Building Your Own Compiler

In Exercise 7.28, we introduced the made-up Simpletron Machine Language (SML). In Exercise 7.29, you used simulation to create the Simpletron computer—a *virtual machine*—to execute programs written in SML. In this challenge section, you'll build a compiler that converts programs written in **Simple**—a made-up, concise, high-level programming language—to SML. This section "ties" together key pieces of the programming process. You'll:

- **write** several Simple high-level language programs,
- **compile** the programs using the compiler you'll build, generating SML machine-language code into a file,

- **load** the SML machine-language code from that file into the Simpletron's memory, and
- **execute** the SML machine-language programs on the Simpletron virtual machine you built in Exercise 7.29.

This section consists of six exercises. The first two cover some key computer-science technology you'll need to implement your compiler. The third introduces the Simple high-level language with some completely coded examples and asks you to write several of your own Simple programs. The fourth guides you through building your compiler. The fifth introduces the crucial topic of compiler optimization—you'll modify your compiler to reduce the number of SML instructions it generates, which will make your SML programs more memory efficient and enable them to execute faster. The final exercise gives you an opportunity to modify your compiler to add more useful features.

12.22 ***(Infix-to-Postfix Converter)*** Compilers use stacks to help evaluate expressions and generate machine-language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of single-digit integer constants, operators and parentheses. You can easily modify the algorithms we present to work with multiple-digit integers and floating-point numbers as well.

People generally write expressions like `3 + 4` and `7 / 9` with the operator *between* its operands. This is called **infix notation**. Computers "prefer" **postfix notation**, in which the operator is written *to the right* of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`.

To evaluate an infix expression, some compilers

- first convert the expression to postfix notation, then
- evaluate the postfix version.

Each of these **stack-oriented algorithms** requires one left-to-right pass of the expression. In this exercise, you'll implement the **infix-to-postfix conversion algorithm**. In the next, you'll implement the **postfix-expression evaluation algorithm**.

Write a program that converts a valid infix arithmetic expression with single-digit integers such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression is

```
6 2 + 5 * 8 4 / -
```

Note that postfix expressions contain no parentheses. The program should read the expression into character array `infix` and use the stack functions implemented in this chapter to help create the postfix expression in character array `postfix`. The algorithm for creating a postfix expression is as follows:

1. Push a left parenthesis `'('` onto the stack.
2. Append a right parenthesis `')'` to the end of `infix`.
3. While the stack is not empty, read `infix` from left to right and do the following:
   If `infix`'s current character is a digit, copy it to the next element of `postfix`.
   If `infix`'s current character is a left parenthesis, push it onto the stack.
   If `infix`'s current character is an operator,
   Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in `postfix`.
   Push the current character in `infix` onto the stack.
   If `infix`'s current character is a right parenthesis,
   Pop operators from the top of the stack and insert them in `postfix` until a left parenthesis is at the top of the stack.
   Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division

The stack should be maintained with the following declarations:

```
struct stackNode {
   char data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;
```

The program should consist of `main` and eight other functions with the following function prototypes:

Function prototype	Description
`void convertToPostfix(char infix[], char postfix[]);`	Convert the infix expression to postfix notation.
`bool isOperator(char c);`	Return `true` if `c` is an operator; otherwise, return `false`. Recall that `bool`, `true` and `false` are defined in `stdbool.h`.
`int precedence(char operator1, char operator2);`	Return `-1`, `0` or `1` to indicate whether the precedence of `operator1` is less than, equal to, or greater than the precedence of `operator2`, respectively.
`void push(StackNodePtr *topPtr, char value);`	Push a value onto the stack.

Function prototype	Description
`char pop(StackNodePtr *topPtr);`	Pop a value off the stack and return that value.
`char stackTop(StackNodePtr topPtr);`	Return the stack's top value without popping the stack.
`bool isEmpty(StackNodePtr topPtr);`	Return `true` if the stack is empty (that is, `topPtr` is `NULL`); otherwise, return `false`.
`void printStack(StackNodePtr topPtr);`	Print the stack—this function traverses the linked list that implements the stack, but does not modify it.

12.23 ***(Postfix-Expression Evaluator)*** Write a program that evaluates a valid postfix expression such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of single digits and operators into a character array. Postfix expressions do not contain parentheses—they're eliminated during the infix-to-postfix conversion. The program should scan the postfix expression and evaluate it using the following algorithm and the stack functions implemented earlier in this chapter:

1. Append the null character (`'\0'`) to the end of the postfix expression. When the algorithm encounters this null character, the evaluation of the postfix expression is complete.
2. While the null character (`'\0'`) has not been encountered, read the expression from left to right:
   If the current character is a digit,
      Push its integer value onto the stack. The integer value of a digit character is its value in the computer's character set minus the value of the zero character (`'0'`) in the computer's character set.
   Otherwise, if the current character is an operator,
      Pop the stack's two top elements into variables `x` and `y`.
      Calculate `y` operator `x`.
      Push the calculation's result onto the stack.
3. When the null character (`'\0'`) is encountered in the expression, pop the stack's top value. This is the postfix expression's result.

This algorithm supports only binary arithmetic operators. So in *Step 2*, if the operator is `'/'`, the top of the stack is `2`, and the next element in the stack is `8`, you'd pop `2` into `x`, pop `8` into `y`, evaluate `8 / 2`, and push the result, `4`, back onto the stack. This applies to each binary arithmetic operator.

The arithmetic operations allowed in an expression are:

+	addition
-	subtraction
*	multiplication
/	division

The stack should be maintained with the following declarations:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;
```

The program should consist of `main` and six other functions with the following function prototypes:

Function prototype	Description
`int evaluatePostfixExpression(char *expr);`	Evaluate the postfix expression and return its result.
`int calculate(int op1, int op2, char operator);`	Evaluate the expression `op1 operator op2` and return its result.
`void push(StackNodePtr *topPtr, int value);`	Push a value onto the stack.
`int pop(StackNodePtr *topPtr);`	Pop a value off the stack and return that value.
`bool isEmpty(StackNodePtr topPtr);`	Return `true` if the stack is empty (that is, `topPtr` is `NULL`); otherwise, return `false`.
`void printStack(StackNodePtr topPtr);`	Print the stack—this function traverses the linked list that implements the stack, but does not modify it.

12.24 ***(The Simple Programming Language—Writing Simple Programs)*** Before building the compiler, let's discuss a simple yet powerful, high-level language similar to early versions of the BASIC programming language. We call the language *Simple*. Every Simple statement consists of a line number and a Simple instruction. Line numbers must appear in ascending order. Each instruction begins with one of the following Simple commands: **`rem`**, **`input`**, **`let`**, **`print`**, **`goto`**, **`if...goto`** or **`end`**, which we describe in the following table. Simple evaluates only integer expressions using the +, -, * and / operators. These operators have the same precedence as in C. Parentheses can change an expression's order of evaluation. Exercise 12.27 suggests enhance-

ments to the Simple compiler. Several of the suggested enhancements, such as adding floating-point capability, require modifications to the Simpletron virtual machine as well.

Command	Example statement	Description
`rem`	`50 rem this is a remark`	Text following the command `rem` is for documentation purposes only and is ignored—no SML code is generated.
`input`	`30 input x`	Display a question mark to prompt the user to enter a single integer. Read that integer from the keyboard and store the integer in `x`.
`let`	`80 let u = 4 * (j - 7))`	Assign to `u` the value of `4 * (j - 7)`. An arbitrarily complex **infix expression** can appear to the right of the equal sign.
`print`	`10 print w`	Display the value of single integer variable `w`.
`goto`	`70 goto 45`	Transfer program control to line `45`.
`if...goto`	`35 if i == z goto 80`	Compare `i` and `z` for equality and transfer control to line `80` if the condition is true; otherwise, continue execution with the next statement.
`end`	`99 end`	Terminate program execution.

Additional Simple Language Rules

Simple also has the following language rules:

- The Simple compiler **recognizes only lowercase letters**—all characters in a Simple program should be lowercase.
- A **variable name is a single letter**. Multi-character variable names are not allowed, so Simple programs should document their variables in `rem` statements.
- Simple uses **only `int` variables**.
- Simple **does not have variable declarations**—merely mentioning a variable name in a program declares the variable and initializes it to zero.
- Simple's syntax **does not allow string manipulation** (reading a string, writing a string, comparing strings, etc.).
- Simple uses the **conditional branch `if...goto` statement** and the **unconditional branch `goto` statement** to alter a program's flow of control. If the condition in the `if...goto` statement is true, control transfers to the specified line number. The following relational and equality operators `<`, `>`, `<=`, `>=`, `==` or `!=` are valid in an `if...goto` statement. Their precedence is the same as in C.

Our compiler assumes that Simple programs are entered correctly. Exercise 12.27 asks you to modify the compiler to perform **syntax-error checking**.

Simple Program Examples

Let's consider several Simple programs that demonstrate the language's features. The first (Fig. 12.5) reads two integers from the keyboard, stores the values in variables a and b, and computes and prints their sum (stored in variable c).

```
10 rem   input two integers, then determine and print their sum
15 rem
20 rem   input the two integers
30 input a
40 input b
45 rem
50 rem   add integers and store result in c
60 let c = a + b
65 rem
70 rem   print the result
80 print c
90 rem   terminate program execution
99 end
```

Fig. 12.5 | Input two integers, then determine and print their sum.

The next program (Fig. 12.6) determines and prints the larger of two integers. The integers are input from the keyboard and stored in the variables s and t. The `if...goto` statement tests the condition `s >= t`. If the condition is true, control transfers to line 90, which displays s. Otherwise, the program displays t, then transfers control to the `end` statement in line 99, where the program terminates.

```
10 rem   input two integers, then determine and print the larger one
20 input s
30 input t
32 rem
35 rem   test if s is greater than or equal to t
40 if s >= t goto 90
45 rem
50 rem   t is greater than s, so print t
60 print t
70 goto 99
75 rem
80 rem   s is greater than or equal to t, so print s
90 print s
99 end
```

Fig. 12.6 | Input two integers, then determine and print the larger one.

Simple does not have repetition statements like C's `for`, `while` or `do...while`. However, you can simulate each of these using the `if...goto` and `goto` statements. Figure 12.7 uses a sentinel-controlled loop to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable j. If the value entered is the sentinel -9999, control transfers to line 99, where the program termi-

nates. Otherwise, `k` is assigned the square of `j`, `k` is output to the screen and control is passed to line `20`, where the next integer is input.

```
10 rem   calculate squares of integers until user enters -9999 sentinel to end
20 input j
23 rem
25 rem   test for sentinel value
30 if j == -9999 goto 99
33 rem
35 rem   calculate square of j and assign result to k
40 let k = j * j
50 print k
53 rem
55 rem   loop to get next j
60 goto 20
99 end
```

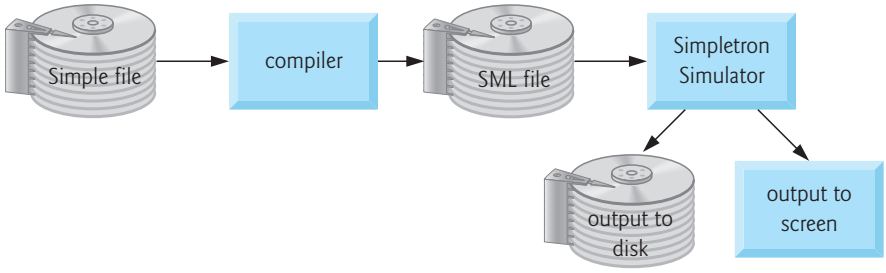
Fig. 12.7 | Calculate squares of integers until user enters `-9999` to end.

Write Your Own Simple Programs

Using the sample programs of Figs. 12.5–12.7 as your guide, write Simple programs to accomplish each of the following:

a) Input three integers, determine their average and print the result.
b) Use a sentinel-controlled loop to input 10 integers and compute and print their sum.
c) Use a counter-controlled loop to input seven integers, some positive and some negative, and compute and print their average.
d) Input a series of integers and determine and print the largest. The first integer input indicates how many numbers should be processed.
e) Input 10 integers and print the smallest.
f) Calculate and print the sum of the even integers from 2 to 30.
g) Calculate and print the product of the odd integers from 1 to 9.

12.25 ***(Building a Compiler; Prerequisite: Complete Exercises 7.28, 7.29, 12.22, 12.23 and 12.24)*** Now that we've presented the Simple language (Exercise 12.24), let's discuss how to build a Simple compiler. The following diagram summarizes the process for compiling a Simple program into SML, then executing it in the Simpletron simulator:



The compiler reads a file containing a Simple program, **compiles** it into **SML code**, then writes the SML one instruction per line to a text file. Next, the Simpletron simulator **loads** the SML file into the Simpletron's 100-element memory array, executes the program and outputs the results to the screen and to a file. We also send all screen outputs to a file to make it easy to print a hard copy.

The Simpletron simulator you developed in Exercise 7.29 takes its input from the keyboard, not a file. You must modify the Simpletron to read from a file so it can run the programs your Simple compiler produces.

The compiler performs **two passes** of a Simple program to convert it to SML:

- The **first pass** constructs a symbol table (discussed in detail below). The compiler stores in the symbol table every **line number**, **variable name** and **constant** of the Simple program. Each is stored with its **type** and its **location** in the final SML code. The **first pass** also produces the corresponding **SML instruction(s)** for each Simple statement. As you'll see, if the Simple program contains statements that transfer control to lines later in the program, the **first pass** results in an SML program containing some **incomplete instructions**.
- The **second pass** locates and **completes the unfinished instructions** and outputs the SML program to a file. The compiler's **first pass** code is much larger than its **second pass** code.

First Pass

The compiler begins by reading into memory the Simple program's first statement. The compiler separates the line into its individual tokens (i.e., "pieces" of a statement) for processing and compilation. You can use **function `strtok`** to do this. Recall that every statement begins with a **line number** followed by a **command**. As the compiler breaks the rest of the statement into tokens, if a token is a **line number**, a **variable**, or a **constant**, it's placed in the **symbol table**. A **line number** is placed in the **symbol table** only if it's the **first token** in a statement—you'll soon see what the compiler does with line numbers that are targets of conditional or unconditional branches.

The **`symbolTable`** is an **array** of **`tableEntry` structures** representing each **symbol** in the program. There's no restriction on the number of symbols that can appear in the program. So, the `symbolTable` could be large. Make the `symbolTable` a 200-element array for now. You can adjust its size once you have a working compiler.

The `tableEntry` structure is declared as follows:

```
struct tableEntry {
   int symbol;
   char type; // 'C' (constant), 'L' (line number), 'V' (variable)
   int location; // 00 to 99
};
```

Each `tableEntry` contains three members:

- **`symbol`** is an integer containing a variable's ASCII representation (again, **variable names** are **single characters**), a **line number**, or an integer **constant**.

- **type** is a character indicating the symbol's type—**'C' for a constant**, **'L' for a line number**, or **'V' for a variable**.
- **location** contains the Simpletron **memory location** (**00** to **99**) associated with the symbol. Simpletron memory is an array of 100 integers in which **SML instructions** and **data** are stored. For a **line number**, the **location** is the Simpletron **memory array** element at which the Simple statement's SML instructions begin. For a **variable** or **constant**, the **location** is the Simpletron memory array element that stores the **variable** or **constant**. **Variables** and **constants** are allocated from location 99 of the Simpletron's memory downward. The first variable or constant is stored in location 99, the next in location 98, and so on.

The **symbol table** plays an integral part in converting Simple programs to SML. We learned in Exercise 7.28 that an **SML instruction** is a signed **four-digit integer** that comprises two parts—the **operation code** and the **operand**. The operation code is determined by the Simple command. For example, the **Simple command input** corresponds to **SML operation code 10** (*read*), and the **Simple command print** corresponds to **SML operation code 11** (*write*). The **operand** is a **memory location** containing the **data** on which the **operation code** performs its task. For instance, the **operation code 10 reads a value from the keyboard** and **stores it in the memory location** specified by the **operand**. The compiler searches `symbolTable` to determine the **Simpletron memory location** for each **symbol** so the corresponding **location** can be used to **complete the SML instructions**.

Each Simple statement's compilation process is based on the particular command. For example, after the **line number** in a **rem statement** is inserted in the symbol table, the compiler ignores the statement's remainder—a `rem` statement is for documentation purposes only—no SML code is generated. The **input**, **print**, **goto** and **end** statements correspond to the SML *read*, *write*, *branch* (to a specific **location**) and *halt* **instructions**. The compiler converts statements containing these Simple commands directly to SML. A **goto statement** may initially contain an **unresolved reference** if the specified **line number** refers to a statement later in the Simple program file. This is called a **forward reference**.

When a **goto statement** is compiled with an **unresolved reference**, the SML instruction must be **flagged** to indicate that the **compiler's second pass** must complete the instruction. The flags are stored in 100-element array `int flags`, in which each element is initialized to `-1`. If the **memory location** to which a **line number** refers is **not yet known** (that is, it's not in the **symbol table**), its **line number** is stored in array `flags` in the element with the same subscript as the **incomplete instruction**. The **incomplete instruction**'s **operand** is set temporarily to `00`. For example, an **unconditional branch instruction** (making a **forward reference**) is left as `+4000` until the **compiler's second pass**, which we'll describe shortly.

Compiling an **if...goto** or **let statement** is more complicated than other statements—each produces more than one SML instruction. For an **if...goto statement**, the compiler produces code to test the condition and possibly branch to another line. The result of the branch could be an **unresolved forward reference**. Each Simple **rela-**

tional and equality operator can be simulated using SML's *branch zero* and *branch negative* instructions (or possibly a combination of both).

For a **`let` statement**, the compiler produces code to evaluate an **arbitrarily complex infix arithmetic expression** consisting of operators, single-letter integer variable names, integer constants and possibly parentheses. Expressions should separate each operand and operator with a space. Exercises 12.22–12.23 presented the **infix-to-postfix conversion algorithm** and the **postfix-evaluation algorithm** compilers use to evaluate expressions. Before building your compiler, you should complete those exercises. The compiler converts each expression from **infix notation** to **postfix notation**, then evaluates the **postfix expression**. As you'll see, the compiler actually generates machine-language instructions in the process of performing the postfix expression evaluation.

How is it that the compiler produces the **machine language** to evaluate an expression containing **variables**? The **postfix-evaluation algorithm** contains a "hook" that allows our compiler to **generate SML instructions rather than actually evaluating the expression**. To enable this "hook" in the compiler, the postfix-evaluation algorithm must be modified to:

- search the **symbol table** for each **symbol** it encounters (and possibly insert it),
- determine the symbol's corresponding **memory location**, and
- ***push the memory location instead of the symbol onto the stack***.

When an operator is encountered in the **postfix expression**, the stack's top two memory locations are popped, and SML for effecting the operation is produced using the **memory locations** as **operands**. Each subexpression's result is stored in a **temporary memory location** and **pushed back onto the stack**, so the postfix expression's evaluation can continue. When **postfix evaluation is complete**, the **result's memory location** is the **only location left on the stack**. This is popped, and SML instructions are generated to assign the result to the variable at the left of the `let` statement.

Second Pass

The compiler's second pass performs two tasks:

- **resolve any unresolved references**, and
- **output the SML code to a file**.

Resolution of each reference occurs as follows:

1. Search the `flags` array for an **unresolved reference** (i.e., an element with a value other than `-1`).
2. Locate the structure in array `symbolTable` containing the symbol stored in the `flags` array (be sure that the type of the symbol is **`'L'` for line number**).
3. Insert the memory location from structure member `location` into the instruction with the **unresolved reference** (remember that an instruction containing **an unresolved reference has operand `00`**).
4. Repeat *Steps 1–3* until the end of the `flags` array is reached.

After the resolution process is complete, the compiler outputs the SML code to a file with one SML instruction per line. The Simpletron can read this file and execute its instructions (after the **simulator is modified** to **read its input from a file**, of course).

A Complete Example

The following example illustrates a complete conversion of a Simple program to SML. Consider a Simple program that inputs an integer, sums the values from 1 to that integer and prints that sum. So, if the user enters 4, the program would calculate 1 + 2 + 3 + 4, which is 10 and print that value. Figure 12.8 shows the program and the SML instructions produced by the compiler's first pass. Figure 12.9 shows the symbol table constructed by the compiler's first pass. Figure 12.10 shows how the compiler allocates Simpletron memory downward from cell 99. In a moment, we'll do a step-by-step walkthrough showing precisely how the compiler creates these tables.

Simple program	SML location and instruction	Description
`5 rem  sum 1 to x`	*none*	`rem` ignored
`10 input x`	00 +1099	read `x` into location 99
`15 rem  check y == x`	*none*	`rem` ignored
`20 if y == x goto 60`	01 +2098	load `y` (location 98) into accumulator
	02 +3199	sub `x` (location 99) from accumulator
	03 +42**00**	branch zero to **unresolved location**
`25 rem  increment y`	*none*	`rem` ignored
`30 let y = y + 1`	04 +2098	load `y` (location 98) into accumulator
	05 +3097	add `1` (location 97) to accumulator
	06 +2196	**store in temporary location 96**
	07 +2096	**load from temporary location 96**
	08 +2198	store accumulator in `y` (location 98)
`35 rem  add y to total t`	*none*	`rem` ignored
`40 let t = t + y`	09 +2095	load `t` (location 95) into accumulator
	10 +3098	add `y` (location 98) to accumulator
	11 +2194	**store in temporary location 94**
	12 +2094	**load from temporary location 94**
	13 +2195	store accumulator in `t` (location 95)
`45 rem  loop to y == x test`	*none*	`rem` ignored
`50 goto 20`	14 +4001	branch to location 01
`55 rem  output result`	*none*	`rem` ignored
`60 print t`	15 +1195	output `t` (location 95) to screen
`99 end`	16 +4300	terminate execution

Fig. 12.8 | SML instructions produced after the compiler's first pass.

Symbol	Type	Location
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
Temporary 96 allocated		
35	L	09
40	L	09
't'	V	95
Temporary 94 allocated		
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Fig. 12.9 | Symbol table for program of Fig. 12.8.

Data counter	Value	Type
...		
93		**Next Simpletron memory cell to allocate**
94	*none*	*Temporary variable*
95	't'	Variable
96	*none*	*Temporary variable*
97	1	Constant
98	'y'	Variable
99	'x'	Variable

Fig. 12.10 | Compiler allocates Simpletron memory downward from the last cell of memory (99).

Most Simple statements convert directly to single SML instructions. The exceptions in this program are `rem` statements, the `if...goto` statement in line `20` and the `let` statements in lines 30 and 40. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in case the line number is referenced in a `goto` or an `if...goto` statement.

Line `20` of the program specifies that if the condition `y == x` is true, program control should transfer to line `60`. Because line `60` appears *later* in the program, the compiler's **first pass** has not yet placed `60` in the symbol table (line numbers are placed in

the symbol table only when they appear as the first token in a statement the compiler has processed). Therefore, it's not possible at this time to determine the operand of the SML branch-zero instruction at location `03` in the array of SML instructions. The compiler places `60` in location `03` of the **`flags` array** to indicate that the second pass will complete this instruction.

We must keep track of the next instruction location in the SML array because **there is not a one-to-one correspondence between Simple statements and SML instructions**. For example, the `if...goto` statement of line `20` compiles into *three* SML instructions. Each time an instruction is produced, we must increment the instruction counter to the next SML array location. The Simpletron's limited memory size could present a problem for Simple programs with many statements, variables and constants. It's conceivable that the compiler could run out of Simpletron memory. To test for this case, your program should contain a **data counter** to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the data counter's value, the SML array is full. In this case, the compilation process should terminate, and the compiler should display an "**out-of-memory**" **error message**.

Step-by-Step Explanation of the Compilation Process's First Pass

Let's walk through the compilation process for the Simple program in Fig. 12.8. The compiler reads the first line of the program:

```
5 rem sum 1 to x
```

The first token in the statement (the line number) is determined using `strtok` (Chapter 8 discussed C's string-manipulation functions). The token returned by `strtok` is converted to an integer using `atoi`, so the symbol `5` can be placed in the symbol table. If the symbol is not found, it's inserted in the symbol table. Since we're at the beginning of the program and this is the first line, no symbols are in the table yet. So, `5` is inserted into the symbol table as type `L` (line number) and assigned the first location in the SML memory array (`00`). Although this line is a remark, a space in the symbol table is allocated for the line number (in case it's referenced by a `goto` or an `if...goto`). **If a program branches to a `rem` statement's line number, control resumes with the first executable statement after the `rem`.** No SML instruction is generated for a `rem` statement, so the instruction counter is not incremented.

Next, the compiler tokenizes the statement

```
10 input x
```

The line number `10` is placed in the symbol table as type `L` and assigned the first location in the SML array (`00`)—a remark began the program, so the instruction counter is still `00`. The command `input` indicates that the next token is a variable (only a variable can appear as an argument in an `input` statement). Because `input` corresponds directly to an SML operation code, the compiler simply has to determine the location of `x` in the SML array. Symbol `x` is not found in the symbol table. So, it's inserted into the symbol table as the **ASCII representation of `x`**, given type `V`

(for variable), and assigned location 99 in the SML array. Data storage begins at 99 and is allocated downward—98, 97, and so on. SML code can now be generated for this statement. Operation code 10 (the SML *read* operation code) is multiplied by 100, and x's location (as determined in the symbol table) is added to it, which completes the instruction +1099. This is then stored in the SML array at location 00. The instruction counter is incremented by 1 because a single SML instruction was produced.

Next, the compiler tokenizes the statement

```
15 rem   check y == x
```

The symbol table is searched for line number 15, which is not found. The line number is inserted as type L and assigned the SML array's next location (01). Again, rem statements do not produce code, so the instruction counter is not incremented.

Next, the compiler tokenizes the statement

```
20 if y == x goto 60
```

Line number 20 is inserted in the symbol table and given type L with the next location in the SML array, 01. The command if indicates that a condition is to be evaluated. The variable y is not found in the symbol table, so it's inserted and given the type V and the SML location 98. Next, SML instructions are generated to evaluate the condition. There is no direct equivalent in SML for the if...goto, so it must be simulated by performing a calculation using x and y and branching based on the result. If y equals x, the result of subtracting x from y is zero. So, the SML ***branch-zero* instruction** can be used with the calculation result to simulate the if...goto statement.

The first step requires that y be loaded (from SML location 98) into the accumulator. This produces the instruction 01 +2098. Next, x is subtracted from the accumulator. This produces the instruction 02 +3199. The value in the accumulator may be zero, positive or negative. Since the **operator is** ==, we want to ***branch zero***. First, the symbol table is searched for the branch location (60), which is not found. So, 60 is placed in the flags array at location 03, and the instruction 03 +4200 is generated. We cannot add the branch location because we have not yet assigned a location to line 60 in the SML array—this location will be resolved later. The instruction counter is incremented to 04.

The compiler proceeds to the statement

```
25 rem   increment y
```

The line number 25 is inserted in the symbol table as type L and assigned SML location 04. The instruction counter is not incremented.

When the statement

```
30 let y = y + 1
```

is tokenized, the line number 30 is inserted in the symbol table as type L and assigned SML location 04. Command let indicates that the line is an assignment statement. First, all the symbols on the line are inserted in the symbol table (if they are not already there). The constant integer 1 is inserted as type C and assigned SML

location 97. Next, the right side of the assignment is **converted from infix to postfix notation**. Then the postfix expression (y 1 +) is evaluated. Symbol y is located in the symbol table, and its corresponding memory location, 98, is pushed onto the stack. Symbol 1 is also located in the symbol table, and its corresponding memory location, 97, is *pushed onto the stack*. When the operator + is encountered, the postfix evaluator *pops the stack* into the right operand of the operator and *pops the stack* again into the left operand of the operator, then produces the SML instructions

```
04 +2098    (load y)
05 +3097    (add 1)
```

The result of the expression is stored in a **temporary location in memory** (96) with instruction

```
06 +2196    (store temporary)
```

and the temporary location is *pushed onto the stack*. Now that the expression has been evaluated, the result must be stored in the `let` statement's variable y. So, the temporary location is loaded into the accumulator, and the accumulator is stored in y with the instructions

```
07 +2096    (load temporary)
08 +2198    (store y)
```

Notice that some of these SML instructions—storing the accumulator into temporary location 96, then immediately reloading the accumulator from location 96—appear to be **redundant**. Eliminating such redundancy is an example of **compiler optimization**, which we'll say more about shortly.

When the compiler tokenizes the statement

```
35 rem   add y to total
```

it inserts line number 35 in the symbol table as type L and assigns it location 09.

The following statement is similar to line 30:

```
40 let t = t + y
```

The variable t is inserted in the symbol table as type V and assigned SML location 95. The instructions follow the same logic and format as line 30, and the instructions 09 +2095, 10 +3098, 11 +2194, 12 +2094, and 13 +2195 are generated. The result of t + y is assigned to temporary location 94 before being assigned to t (95). The instructions in memory locations 11 and 12 also appear to be **redundant**. Again, we'll discuss this optimization issue shortly.

The statement

```
45 rem   loop to y == x test
```

is a remark, so line 45 is inserted in the symbol table as type L and assigned SML location 14.

The statement

```
50 goto 20
```

transfers control to line 20. Line number 50 is inserted in the symbol table as type L and assigned SML location 14. The **equivalent of goto in SML** is the ***unconditional branch*** **(40) instruction** that transfers control to a specific SML location. The compiler searches the symbol table for line 20 and finds that it corresponds to SML location 01. The operation code (40) is multiplied by 100, and location 01 is added to produce the instruction +4001 at location 14.

The statement

```
55 rem   output result
```

is a remark, so line 55 is inserted in the symbol table as type L and assigned SML location 15.

The statement

```
60 print t
```

is an output statement. Line number 60 is inserted in the symbol table as type L and assigned SML location 15. The **equivalent of print in SML** is **operation code 11** (***write***). Variable t's location is determined from the symbol table, then added to the result of multiplying the operation code by 100. This forms the instruction +1195 at location 15.

The statement

```
99 end
```

is the final line of the program. Line number 99 is stored in the symbol table as type L and assigned SML location 16. The end command produces the SML instruction +4300 (43 is *halt* in SML). This is written as the final instruction in the SML memory array. The *halt* instruction has no operand. Can you think of a useful reason to allow an operand for the *halt* instruction?

The Compilation Process's Second Pass

On the compiler's *second pass*, we begin by searching the **flags array** for values other than -1. Location 03 contains 60, so the compiler knows that instruction 03 is incomplete. The compiler completes the instruction by searching the symbol table for 60, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line 60 corresponds to SML location 15, so the completed instruction +4215 at location 03 is produced, replacing +4200. The Simple program has now been compiled successfully.

Building Your Compiler

To build the compiler, you'll have to perform each of the following tasks:

a) Modify the Simpletron simulator program you wrote in Exercise 7.29 to take its input from a file specified by the user (see Chapter 11). The simulator should also output its results to a file in the same format as the screen output.

b) Modify the infix-to-postfix evaluation algorithm of Exercise 12.22 to process **multi-digit integer operands** and **single-letter variable-name operands**. Standard library function `strtok` can be used to locate each constant and variable in an expression. Constants can be converted from strings to integers using standard-library function `atoi`. The postfix expression's data representation must be altered to support variable names and integer constants.

c) Modify the postfix evaluation algorithm to process **multi-digit-integer operands** and **single-letter variable-name operands**. The algorithm also should now implement the previously discussed "hook" so that it produces SML instructions rather than directly evaluating the expression. Standard-library function `strtok` can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using standard-library function `atoi`. The data representation of the postfix expression must be altered to support variable names and integer constants.

d) Build the compiler—incorporate *Part b* and *Part c* for evaluating expressions in `let` statements. Your program should contain a function that performs the compiler's ***first pass*** and one that performs its ***second pass***.

12.26 ***(Optimizing the Simple Compiler)*** When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, usually in triplets called **productions**. A production normally consists of three instructions such as ***load***, ***add*** and ***store***. For example, Fig. 12.11 shows five of the SML instructions produced while compiling the program in Fig. 12.8. The first three instructions are the production that adds 1 to `y`. Instructions `06` and `07` **store the accumulator value in temporary location `96`**, then **load the value from that location right back into the accumulator** so instruction `08` can store the value in location `98`. Often a production is followed by a load instruction for the same location that was just stored. This code can be *optimized* by eliminating the *store* instruction and the subsequent *load* instruction that operate on the same memory location. This optimization would decrease the SML program's "memory footprint" by 25% and improve its execution speed. Figure 12.12 shows the **optimized SML** for the program of Fig. 12.8. There are ***four fewer instructions in the optimized code***. Modify your compiler to perform the optimization you learned in this exercise.

```
04 +2098     (load)
05 +3097     (add)
06 +2196     (store)
07 +2096     (load)
08 +2198     (store)
```

Fig. 12.11 | Unoptimized code from the program of Fig. 12.8.

Simple program	SML location and instruction	Description
`5 rem sum 1 to x`	*none*	`rem` ignored
`10 input x`	`00 +1099`	read `x` into location `99`
`15 rem check y == x`	*none*	`rem` ignored
`20 if y == x goto 60`	`01 +2098`	load `y` (`98`) into accumulator
	`02 +3199`	sub `x` (`99`) from accumulator
	`03 +4211`	branch to location `11` if zero
`25 rem increment y`	*none*	`rem` ignored
`30 let y = y + 1`	`04 +2098`	load `y` into accumulator
	`05 +3097`	add `1` (`97`) to accumulator
	`06 +2198`	store accumulator in `y` (`98`)
`35 rem add y to total`	*none*	rem ignored
`40 let t = t + y`	`07 +2096`	load `t` from location (`96`)
	`08 +3098`	add `y` (`98`) to accumulator
	`09 +2196`	store accumulator in `t` (`96`)
`45 rem loop to y == x test`	*none*	`rem` ignored
`50 goto 20`	`10 +4001`	branch to location `01`
`55 rem output result`	*none*	`rem` ignored
`60 print t`	`11 +1196`	output `t` (`96`) to screen
`99 end`	`12 +4300`	terminate execution

Fig. 12.12 | Optimized code for the program of Fig. 12.8.

12.27 ***(Enhancing the Simple Compiler)*** Perform the following modifications to the Simple compiler. Some of these may also require modifications to the Simpletron Simulator program you wrote in Exercise 7.29. Many of these are quite challenging and could require substantial effort.

a) Modify the Simpletron's memory to have **1000 cells** (**000–999**). Modify the compiler to generate machine language appropriate for the 1000-element Simpletron memory.

b) **Allow the compiler to process floating-point values** in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.

c) **Add support for unary minus** to specify negative integer values.

d) Allow the **modulus operator** (**%**) to be used in `let` statements. Modify the Simpletron Machine Language to include a modulus instruction.

e) Allow **exponentiation** in a `let` statement **using ^ as the exponentiation operator**. Modify the Simpletron Machine Language to include an exponentiation instruction.

f) **Allow the compiler to recognize uppercase and lowercase letters** in Simple statements. So, `x` and `X` would be treated as different variables. No modifications to the Simpletron Simulator are required.

g) **Allow `input` statements to read values for multiple variables**, such as **`input x, y`**. No modifications to the Simpletron Simulator are required.

h) **Allow the compiler to output multiple values in a single `print` statement**, such as **`print a, b, c`**. This would output the variables' values, each separated from the next by one space. No modifications to the Simpletron Simulator are required.

i) Allow the `print` statement's operand to be an **infix expression**.

j) **Add syntax-checking** capabilities to the compiler so **error messages** are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron Simulator are required.

k) **Allow integer arrays**. No modifications to the Simpletron Simulator are required.

l) **Allow subroutines specified by the Simple commands `gosub` and `return`**. Command `gosub` passes program control to a subroutine, and command `return` passes control back to the statement after the `gosub`. This is **similar to a function call in C**. The same subroutine can be called from many `gosub`s distributed throughout a program. No modifications to the Simpletron Simulator are required.

m) **Allow repetition structures of the form**

```
for x = 2 to 10
   rem Simple statements
next
```

This `for` statement **loops from `2` to `10`** with a **default increment of `1`**. No modifications to the Simpletron Simulator are required.

n) **Allow repetition structures of the form**

```
for x = 2 to 10 step 2
   rem Simple statements
next
```

This **`for` statement loops from `2` to `10`** with an **increment of `2`**. The `next` line marks the end of the body of the `for` statement. No modifications to the Simpletron Simulator are required.

This page is intentionally left blank