Chapter 12
C Data Structures

C How to Program, 8/e, GE

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 such as single-subscripted arrays, double-subscripted arrays and `structs`.
- This chapter introduces dynamic data structures with sizes that grow and shrink at execution time.
  - Linked lists are collections of data items "lined up in a row"—insertions and deletions are made *anywhere* in a linked list.
  - Stacks are important in compilers and operating systems—insertions and deletions are made *only at one end* of a stack—its top.

12.1 Introduction (Cont.)

  - Queues represent waiting lines; insertions are made *only at the back* (also referred to as the tail) of a queue and deletions are made *only from the front* (also referred to as the head) of a queue.
  - Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language.
- Each of these data structures has many other interesting applications.

12.1 Introduction (Cont.)

- We'll discuss each of the major types of data structures and implement programs that create and manipulate them.
- In the next part of the book—the introduction to C++ and object-oriented programming—we'll study data abstraction.
- This technique will enable us to build these data structures in a dramatically different manner designed for producing software that's much easier to maintain and reuse.

12.2 Self-Referential Structures

- Recall that a *self-referential structure* contains a pointer member that points to a structure of the *same* structure type.
- For example, the definition
  - ```
    struct node {
       int data;
       struct node *nextPtr;
    };
    ```

  defines a type, `struct node`.
- A structure of type `struct node` has two members—integer member `data` and pointer member `nextPtr`.
    ```

12.2 Self-Referential Structures (Cont.)

- Member `nextPtr` points to a structure of type `struct node`—a structure of the *same* type as the one being declared here, hence the term "*self-referential structure*."
- Member `nextPtr` is referred to as a link—i.e., it can be used to "tie" a structure of type `struct node` to another structure of the same type.
- Self-referential structures can be *linked* together to form useful data structures such as lists, queues, stacks and trees.

12.2 Self-Referential Structures (Cont.)

- Figure 12.1 illustrates two self-referential structure objects linked together to form a list.
- A slash—representing a NULL pointer—is placed in the link member of the second self-referential structure to indicate that the link does not point to another structure.
- [*Note*: The slash is only for illustration purposes; it does not correspond to the backslash character in C.]
- A NULL pointer normally indicates the end of a data structure just as the null character indicates the end of a string.

Common Programming Error 12.1

Not setting the link in the last node of a list to `NULL` *can lead to runtime errors.*

Fig. 12.1 | Self-referential structures linked together.

12.3 Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation—the ability for a program to *obtain more memory space at execution time* to hold new nodes, and to *release space no longer needed*.
- Functions `malloc` and `free`, and operator `sizeof`, are essential to dynamic memory allocation.

12.3 Dynamic Memory Allocation (Cont.)

- Function `malloc` takes as an argument the number of bytes to be allocated and returns a pointer of type `void *` (pointer to void) to the allocated memory.
- As you recall, a `void *` pointer may be assigned to a variable of *any* pointer type.
- Function `malloc` is normally used with the `sizeof` operator.

12.3 Dynamic Memory Allocation (Cont.)

- For example, the statement

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

  evaluates `sizeof(struct node)` to determine the size in bytes of a structure of type `struct node`, *allocates a new area in memory* of that number of bytes and stores a pointer to the allocated memory in variable `newPtr`.
- The allocated memory is *not* initialized.
- If no memory is available, `malloc` returns `NULL`.

12.3 Dynamic Memory Allocation (Cont.)

- Function `free` *deallocates* memory—i.e., the memory is *returned* to the system so that it can be reallocated in the future.
- To *free* memory dynamically allocated by the preceding `malloc` call, use the statement
  - `free(newPtr);`
- C also provides functions `calloc` and `realloc` for creating and modifying *dynamic arrays*.

Portability Tip 12.1

A structure's size is not necessarily the sum of the sizes of its members. This is so because of various machine-dependent boundary alignment requirements (see Chapter 10).

Error-Prevention Tip 12.1

When using `malloc`*, test for a* `NULL` *pointer return value, which indicates that the memory was* not *allocated.*

Common Programming Error 12.2

Not `free`ing 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."

Error-Prevention Tip 12.2

When memory that was dynamically allocated is no longer needed, use `free` *to return the memory to the system immediately. Then set the pointer to* `NULL` *to eliminate the possibility that the program could refer to memory that's been reclaimed and which may have already been allocated for another purpose.*

Common Programming Error 12.3

Freeing memory not allocated dynamically with `malloc` *is an error.*

Common Programming Error 12.4

Referring to memory that has been freed is an error that typically results in the program crashing.

12.4 Linked Lists

- A linked list is a linear collection of self-referential structures, called nodes, connected by pointer links—hence, the term "linked" list.
- A linked list is accessed via a pointer to the first node of the list.
- 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.

12.4 Linked Lists (Cont.)

- A node can contain data of *any* type including other `struct` objects.
- Stacks and queues are also linear data structures
  - Constrained versions of linked lists
- Trees are *nonlinear* data structures.
- Lists of data can be stored in arrays, but linked lists provide several advantages.
- A linked list is appropriate when the number of data elements is *unpredictable*.

12.4 Linked Lists (Cont.)

- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of an array created at compile time, however, cannot be altered.
- Arrays can become full.
- Linked lists become full only when the system has *insufficient memory* to satisfy dynamic storage allocation requests.

Performance Tip 12.1

An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations.

12.4 Linked Lists (Cont.)

- Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.

Performance Tip 12.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

Performance Tip 12.3

The elements of an array are stored contiguously in memory. This allows immediate access to any array element because the address of any element 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.

12.4 Linked Lists (Cont.)

- Linked-list nodes are normally *not* stored contiguously in memory.
- Logically, however, the nodes of a linked list *appear* to be contiguous.
- Figure 12.2 illustrates a linked list with several nodes.

Performance Tip 12.4

Using dynamic memory allocation (instead of arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that the pointers take up space, and that dynamic memory allocation incurs the overhead of function calls.

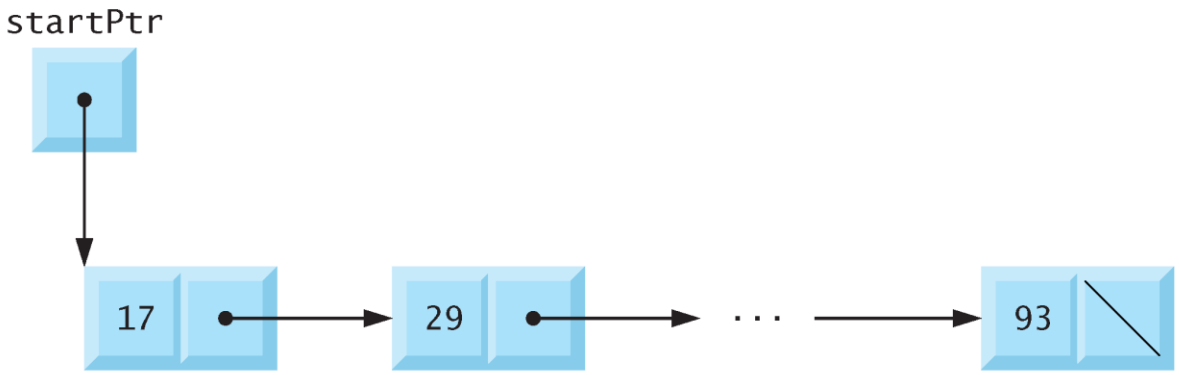

Fig. 12.2 | Linked-list graphical representation.

12.4 Linked Lists (Cont.)

- Figure 12.3 (output shown in Fig. 12.4) manipulates a list of characters.
- You can insert a character in the list in alphabetical order (function `insert`) or to delete a character from the list (function `delete`).

```c
// Fig. 12.3: fig12_03.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)
{
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 1 of 8.)

```
24      ListNodePtr startPtr = NULL; // initially there are no nodes
25      char item; // char entered by user
26
27      instructions(); // display the menu
28      printf("%s", "? ");
29      unsigned int choice; // user's choice
30      scanf("%u", &choice);
31
32      // loop while user does not choose 3
33      while (choice != 3) {
34
35         switch (choice) {
36            case 1:
37               printf("%s", "Enter a character: ");
38               scanf("\n%c", &item);
39               insert(&startPtr, item); // insert item in list
40               printList(startPtr);
41               break;
42            case 2: // delete an element
43               // if list is not empty
44               if (!isEmpty(startPtr)) {
45                  printf("%s", "Enter character to be deleted: ");
46                  scanf("\n%c", &item);
47
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 2 of 8.)

```
48                  // if character is found, remove it
49                  if (delete(&startPtr, item)) { // remove item
50                     printf("%c deleted.\n", item);
51                     printList(startPtr);
52                  }
53                  else {
54                     printf("%c not found.\n\n", item);
55                  }
56               }
57               else {
58                  puts("List is empty.\n");
59               }
60
61               break;
62            default:
63               puts("Invalid choice.\n");
64               instructions();
65               break;
66         }
67
68         printf("%s", "? ");
69         scanf("%u", &choice);
70      }
71
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 3 of 8.)

```
72      puts("End of run.");
73   }
74
75   // display program instructions to user
76   void instructions(void)
77   {
78      puts("Enter your choice:\n"
79         "   1 to insert an element into the list.\n"
80         "   2 to delete an element from the list.\n"
81         "   3 to end.");
82   }
83
84   // insert a new value into the list in sorted order
85   void insert(ListNodePtr *sPtr, char value)
86   {
87      ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node
88
89      if (newPtr != NULL) { // is space available?
90         newPtr->data = value; // place value in node
91         newPtr->nextPtr = NULL; // node does not link to another node
92
93         ListNodePtr previousPtr = NULL;
94         ListNodePtr currentPtr = *sPtr;
95
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 4 of 8.)

```
96          // loop to find the correct location in the list
97          while (currentPtr != NULL && value > currentPtr->data) {
98             previousPtr = currentPtr; // walk to ...
99             currentPtr = currentPtr->nextPtr; // ... next node
100         }
101
102         // insert new node at beginning of list
103         if (previousPtr == NULL) {
104            newPtr->nextPtr = *sPtr;
105            *sPtr = newPtr;
106         }
107         else { // insert new node between previousPtr and currentPtr
108            previousPtr->nextPtr = newPtr;
109            newPtr->nextPtr = currentPtr;
110         }
111      }
112      else {
113         printf("%c not inserted. No memory available.\n", value);
114      }
115   }
116
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 5 of 8.)

```
117 // delete a list element
118 char delete(ListNodePtr *sPtr, char value)
119 {
120    // delete first node if a match is found
121    if (value == (*sPtr)->data) {
122       ListNodePtr tempPtr = *sPtr; // hold onto node being removed
123       *sPtr = (*sPtr)->nextPtr; // de-thread the node
124       free(tempPtr); // free the de-threaded node
125       return value;
126    }
127    else {
128       ListNodePtr previousPtr = *sPtr;
129       ListNodePtr currentPtr = (*sPtr)->nextPtr;
130
131       // loop to find the correct location in the list
132       while (currentPtr != NULL && currentPtr->data != value) {
133          previousPtr = currentPtr; // walk to ...
134          currentPtr = currentPtr->nextPtr; // ... next node
135       }
136
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 6 of 8.)

```
137         // delete node at currentPtr
138         if (currentPtr != NULL) {
139            ListNodePtr tempPtr = currentPtr;
140            previousPtr->nextPtr = currentPtr->nextPtr;
141            free(tempPtr);
142            return value;
143         }
144      }
145
146      return '\0';
147   }
148
149   // return 1 if the list is empty, 0 otherwise
150   int isEmpty(ListNodePtr sPtr)
151   {
152      return sPtr == NULL;
153   }
154
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 7 of 8.)

```
155 // print the list
156 void printList(ListNodePtr currentPtr)
157 {
158    // if list is empty
159    if (isEmpty(currentPtr)) {
160       puts("List is empty.\n");
161    }
162    else {
163       puts("The list is:");
164
165       // while not the end of the list
166       while (currentPtr != NULL) {
167          printf("%c --> ", currentPtr->data);
168          currentPtr = currentPtr->nextPtr;
169       }
170
171       puts("NULL\n");
172    }
173 }
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 8 of 8.)

```
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. 12.4 | Sample output for the program of Fig. 12.3. (Part 1 of 2.)

```
? 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. 12.4 | Sample output for the program of Fig. 12.3. (Part 2 of 2.)

12.4 Linked Lists (Cont.)

- The primary functions of linked lists are `insert` and `delete`
- Function `isEmpty` is a predicate function—it *does not* alter the list in any way; rather it determines whether the list is empty (i.e., the pointer to the first node of the list is `NULL`).
- If the list is empty, `1` is returned; otherwise, `0` is returned.
- Function `printList` prints the list.

12.4 Linked Lists (Cont.)

- Characters are inserted in the list in *alphabetical order*.
- Function `insert` receives the address of the list and a character to be inserted.
- The list's address is necessary when a value is to be inserted at the *start* of the list.
- Providing the address enables the list (i.e., the pointer to the first node of the list) to be *modified* via a call by reference.
- Because the list itself is a pointer (to its first element), passing its address creates a pointer to a pointer (i.e., double indirection).
- This is a complex notion and requires careful programming.

12.4 Linked Lists (Cont.)

- Steps for inserting a character in the list (Fig. 12.5):
  - *Create a node:* call `malloc`, assign to `newPtr` the address of the allocated memory, assign the character to be inserted to `newPtr->data`, and assign `NULL` to `newPtr->nextPtr`
  - Initialize `previousPtr` to `NULL` and `currentPtr` to `*sPtr`—the pointer to the start of the list.
    - These pointers store the locations of the node *preceding* the insertion point and the node *after* the insertion point.
  - While `currentPtr` is not `NULL` and the value to be inserted is greater than `currentPtr->data`, assign `currentPtr` to `previousPtr` and advance `currentPtr` to the next node in the list
    - This locates the insertion point for the value.

12.4 Linked Lists (Cont.)

- If `previousPtr` is `NULL`, insert the new node as the first node in the list. Assign `*sPtr` to `newPtr->nextPtr` (the new node link points to the former first node) and assign `newPtr` to `*sPtr` (`*sPtr` points to the new node). Otherwise, if `previousPtr` is not `NULL`, the new node is inserted in place. 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).

Error-Prevention Tip 12.3

Assign `NULL` *to a new node's link member. Pointers should be initialized* before *they're used.*

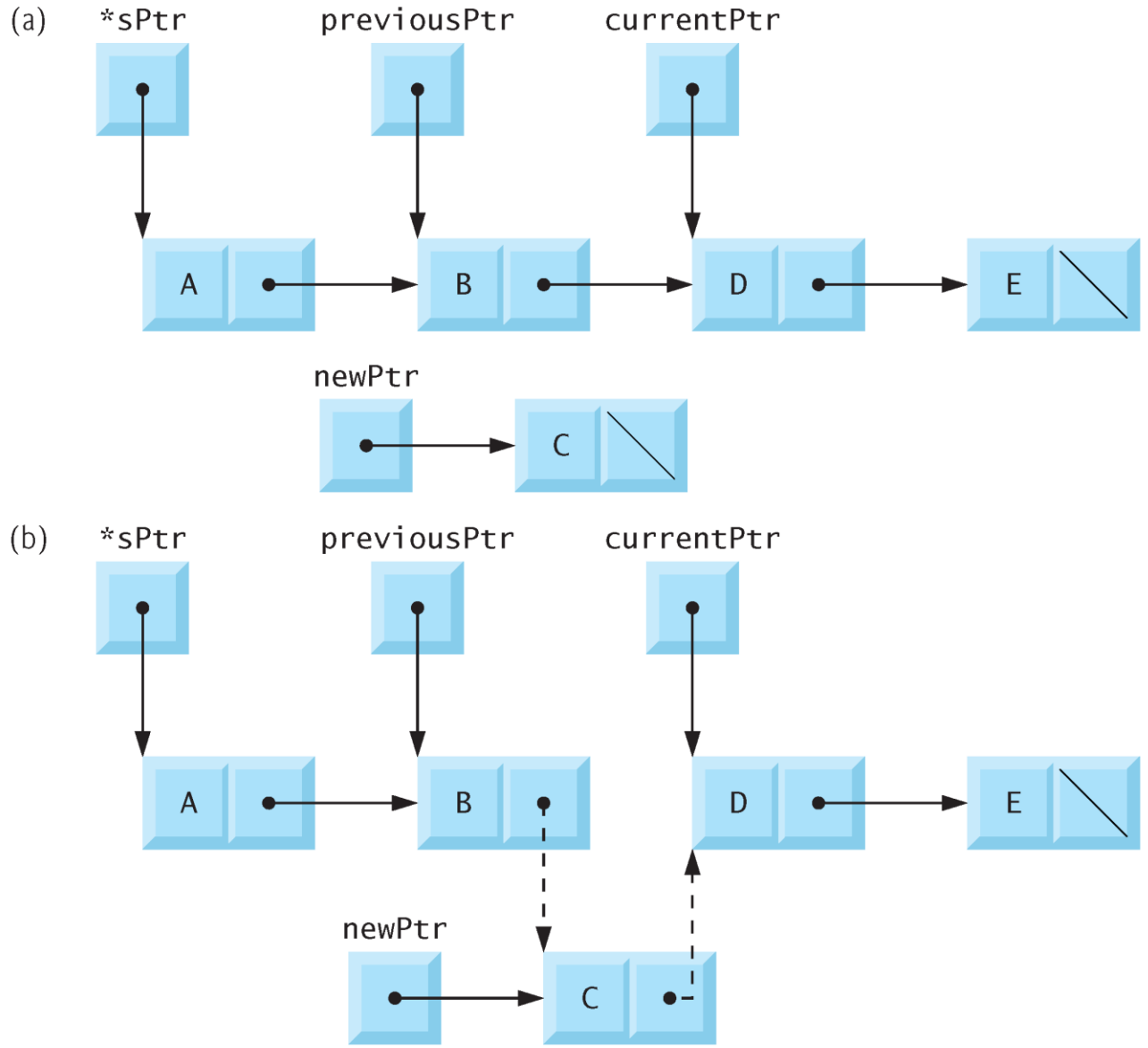

Fig. 12.5 | Inserting a node in order in a list.

12.4 Linked Lists (Cont.)

- Figure 12.5 illustrates the insertion of a node containing the character `'C'` into an ordered list.
- Part (a) of the figure shows the list and the new node just before the insertion.
- Part (b) of the figure shows the result of inserting the new node.
- The reassigned pointers are dotted arrows.
- For simplicity, we implemented function `insert` (and other similar functions in this chapter) with a `void` return type.
- It's possible that function `malloc` will *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.

12.4.2 Function `delete`

- Function `delete` receives the address of the pointer to the start of the list and a character to be deleted.
- Steps for deleting a character from the list:
  - If the character to be deleted matches the character in the first node of the list, assign `*sPtr` to `tempPtr` (`tempPtr` will be used to `free` the unneeded memory), assign `(*sPtr)->nextPtr` to `*sPtr` (`*sPtr` now points to the second node in the list), `free` the memory pointed to by `tempPtr`, and return the character that was deleted.
  - Otherwise, initialize `previousPtr` with `*sPtr` and initialize `currentPtr` with `(*sPtr)->nextPtr` to advance the second node.
  - While `currentPtr` is not `NULL` and the value to be deleted is not equal to `currentPtr->data`, assign `currentPtr` to `previousPtr`, and assign `currentPtr->nextPtr` to `currentPtr`. This locates the character to be deleted if it's contained in the list.

12.4.2 Function `delete` (Cont.)

- If `currentPtr` is not `NULL`, assign `currentPtr` to `tempPtr`, assign `currentPtr->nextPtr` to `previousPtr->nextPtr`, free the node pointed to by `tempPtr`, and return the character that was deleted from the list. If `currentPtr` is `NULL`, return the null character (`'\0'`) to signify that the character to be deleted was not found in the list.

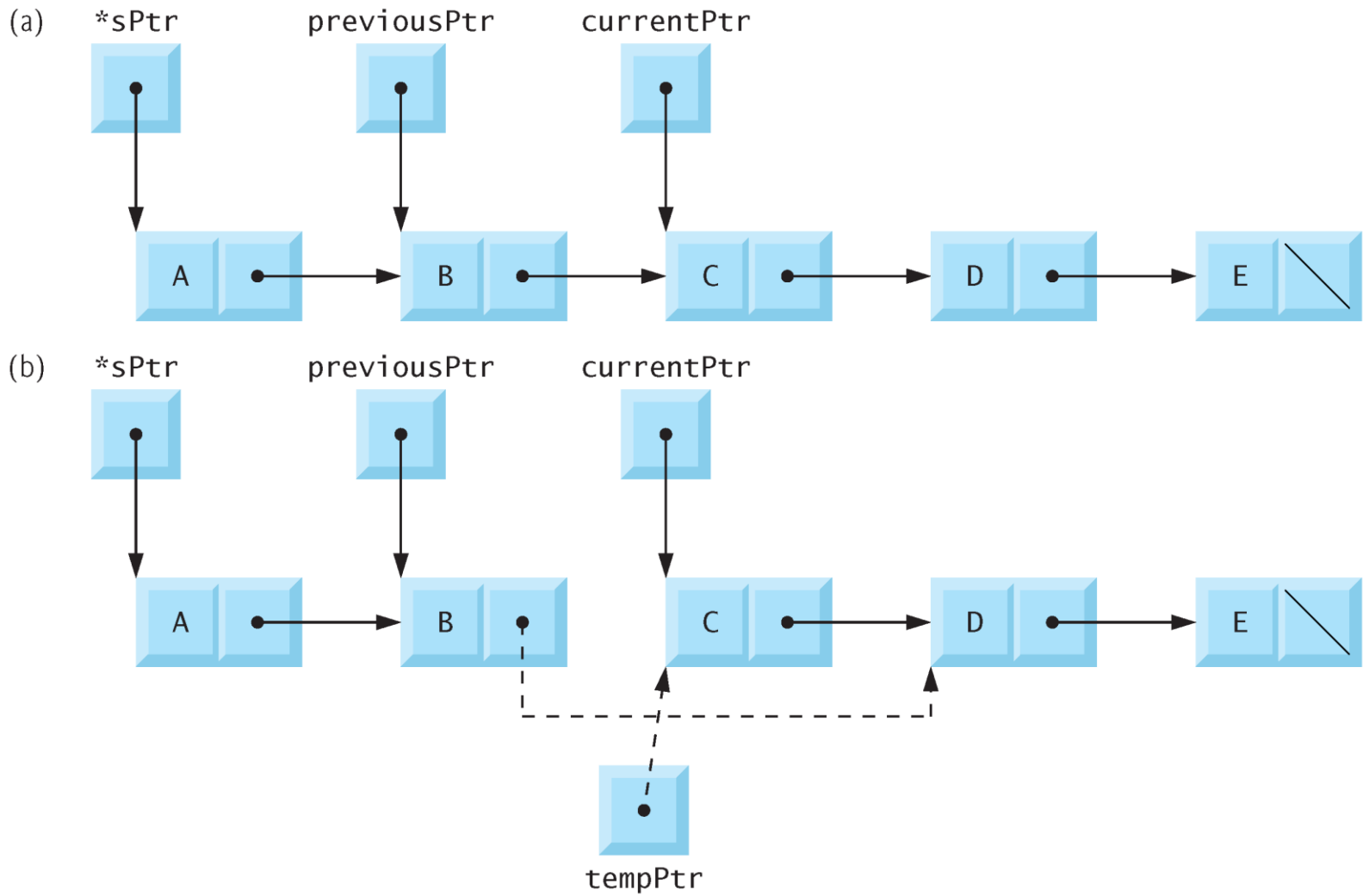

Fig. 12.6 | Deleting a node from a list.

12.4.2 Function `delete` (Cont.)

- Fig. 12.6 illustrates the deletion of a node from a linked list.
- Part (a) of the figure shows the linked list after the preceding insert operation.
- Part (b) shows the reassignment of the link element of `previousPtr` and the assignment of `currentPtr` to `tempPtr`.
- Pointer `tempPtr` is used to *free* the memory allocated to the node that stores `'C'`.
- Recall that we recommended setting a freed pointer to `NULL`.
- We do not do that in these two cases, because `tempPtr` is a local automatic variable and the function returns immediately.

12.4.3 Function `printList`

- Function `printList` receives a pointer to the start of the list as an argument and refers to the pointer as `currentPtr`.
- The function first determines whether the list is empty and, if so, prints "`List is empty.`" and terminates.
- Otherwise, it prints the data in the list

12.4 Linked Lists (Cont.)

- While `currentPtr` is not `NULL`, the value of `currentPtr->data` is printed by the function, and `currentPtr->nextPtr` is assigned 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*, and an error will occur.
- The printing algorithm is identical for linked lists, stacks and queues.

12.5 Stacks

- A stack can be implemented as a constrained version of a linked list.
- New nodes can be added to a stack and removed from a stack *only* at the *top*.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.
- A stack is referenced via a pointer to the top element of the stack.
- The link member in the last node of the stack is set to NULL to indicate the bottom of the stack.

12.5 Stacks (Cont.)

- Figure 12.7 illustrates a stack with several `nodes–stackPtr` points to the stack's top element.
- Stacks and linked lists are represented identically.
- The difference between stacks and linked lists is that insertions and deletions may occur *anywhere* in a linked list, but *only* at the *top* of a stack.

Common Programming Error 12.5

Not setting the link in the bottom node of a stack to `NULL` *can lead to runtime errors.*

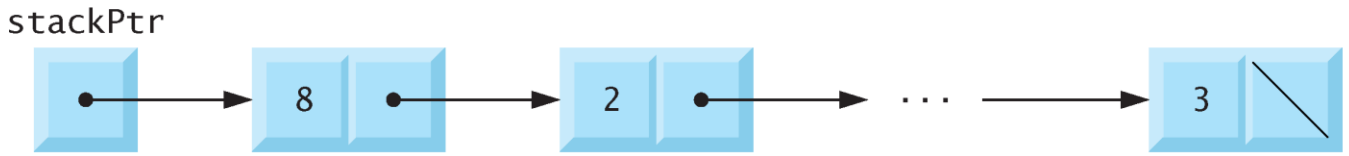

Fig. 12.7 | Stack graphical representation.

12.5 Stacks (Cont.)

- 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*.
- Figure 12.8 (output shown in Fig. 12.9) implements a simple stack of integers.
- The program provides three options: 1) push a value onto the stack (function `push`), 2) pop a value off the stack (function `pop`) and 3) terminate the program.

```c
// Fig. 12.8: fig12_08.c
// A simple stack program
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct stackNode {
   int data; // define data as an int
   struct stackNode *nextPtr; // stackNode pointer
};

typedef struct stackNode StackNode; // synonym for struct stackNode
typedef StackNode *StackNodePtr; // synonym for StackNode*

// prototypes
void push(StackNodePtr *topPtr, int info);
int pop(StackNodePtr *topPtr);
int isEmpty(StackNodePtr topPtr);
void printStack(StackNodePtr currentPtr);
void instructions(void);

```

Fig. 12.8 | A simple stack program. (Part 1 of 7.)

```
22  // function main begins program execution
23  int main(void)
24  {
25     StackNodePtr stackPtr = NULL; // points to stack top
26     int value; // int input by user
27
28     instructions(); // display the menu
29     printf("%s", "? ");
30     unsigned int choice; // user's menu choice
31     scanf("%u", &choice);
32
33     // while user does not enter 3
34     while (choice != 3) {
35
36        switch (choice) {
37           // push value onto stack
38           case 1:
39              printf("%s", "Enter an integer: ");
40              scanf("%d", &value);
41              push(&stackPtr, value);
42              printStack(stackPtr);
43              break;
```

Fig. 12.8 | A simple stack program. (Part 2 of 7.)

```
44            // pop value off stack
45            case 2:
46               // if stack is not empty
47               if (!isEmpty(stackPtr)) {
48                  printf("The popped value is %d.\n", pop(&stackPtr));
49               }
50
51               printStack(stackPtr);
52               break;
53            default:
54               puts("Invalid choice.\n");
55               instructions();
56               break;
57         }
58
59         printf("%s", "? ");
60         scanf("%u", &choice);
61      }
62
63      puts("End of run.");
64   }
65
```

Fig. 12.8 | A simple stack program. (Part 3 of 7.)

```
66  // display program instructions to user
67  void instructions(void)
68  {
69     puts("Enter choice:\n"
70        "1 to push a value on the stack\n"
71        "2 to pop a value off the stack\n"
72        "3 to end program");
73  }
74
75  // insert a node at the stack top
76  void push(StackNodePtr *topPtr, int info)
77  {
78     StackNodePtr newPtr = malloc(sizeof(StackNode));
79
80     // insert the node at stack top
81     if (newPtr != NULL) {
82        newPtr->data = info;
83        newPtr->nextPtr = *topPtr;
84        *topPtr = newPtr;
85     }
86     else { // no space available
87        printf("%d not inserted. No memory available.\n", info);
88     }
89  }
```

Fig. 12.8 | A simple stack program. (Part 4 of 7.)

```
90
91   // remove a node from the stack top
92   int pop(StackNodePtr *topPtr)
93   {
94      StackNodePtr tempPtr = *topPtr;
95      int popValue = (*topPtr)->data;
96      *topPtr = (*topPtr)->nextPtr;
97      free(tempPtr);
98      return popValue;
99   }
100
```

Fig. 12.8 | A simple stack program. (Part 5 of 7.)

```
101 // print the stack
102 void printStack(StackNodePtr currentPtr)
103 {
104    // if stack is empty
105    if (currentPtr == NULL) {
106       puts("The stack is empty.\n");
107    }
108    else {
109       puts("The stack is:");
110
111       // while not the end of the stack
112       while (currentPtr != NULL) {
113          printf("%d --> ", currentPtr->data);
114          currentPtr = currentPtr->nextPtr;
115       }
116
117       puts("NULL\n");
118    }
119 }
120
```

Fig. 12.8 | A simple stack program. (Part 6 of 7.)

```
121 // return 1 if the stack is empty, 0 otherwise
122 int isEmpty(StackNodePtr topPtr)
123 {
124    return topPtr == NULL;
125 }
```

Fig. 12.8 | A simple stack program. (Part 7 of 7.)

```
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

? 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
```

Fig. 12.9 | Sample output from the program of Fig. 12.8. (Part 1 of 2.)

```
? 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. 12.9 | Sample output from the program of Fig. 12.8. (Part 2 of 2.)

12.5.1 Function push

- Function `push` places a new node at the top of the stack.
- The function consists of three steps:
  - Create a new node by calling `malloc` and assign the location of the allocated memory to `newPtr`.
  - Assign to `newPtr->data` the value to be placed on the stack and assign `*topPtr` (the stack top pointer) to `newPtr->nextPtr`—the link member of `newPtr` now points to the previous top node.
  - Assign `newPtr` to `*topPtr`—`*topPtr` now points to the new stack top.

12.5.1 Function push

- Manipulations involving `*topPtr` change the value of `stackPtr` in `main`.
- Figure 12.10 illustrates function `push`.
- Part (a) of the figure shows the stack and the new node before the `push` operation.
- The dotted arrows in part (b) illustrate Steps 2 and 3 of the `push` operation that enable the node containing `12` to become the new stack top.

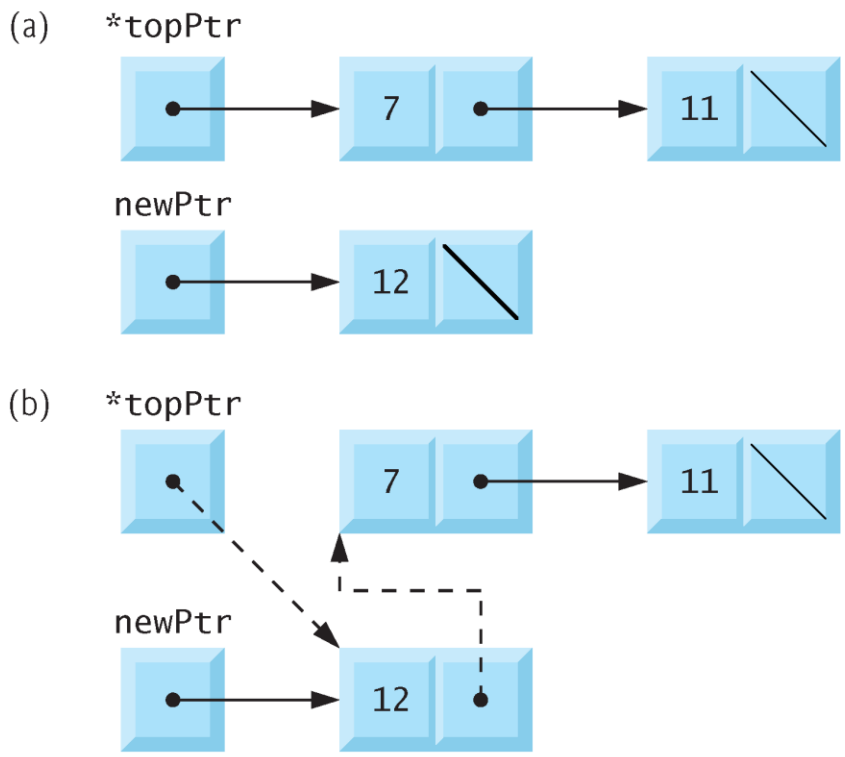

Fig. 12.10 | `push` operation.

12.5.2 Function pop

- Function `pop` removes a node from the top of the stack.
- Function `main` determines if the stack is empty before calling `pop`.
- The `pop` operation consists of five steps:
  - Assign `*topPtr` to `tempPtr`, which will be used to free the unneeded memory
  - Assign `(*topPtr)->data` to `popValue` to *save* the value in the top node
  - Assign `(*topPtr)->nextPtr` to `*topPtr` so `*topPtr` contains *address of the new top node*
  - *Free the memory* pointed to by `tempPtr`
  - *Return* `popValue` to the caller

12.5.2 Function pop (Cont.)

- Figure 12.11 illustrates function `pop`.
- Part (a) shows the stack *after* the previous `push` operation.
- Part (b) shows `tempPtr` pointing to the *first node* of the stack and `topPtr` pointing to the *second node* of the stack.
- Function `free` is used to *free the memory* pointed to by `tempPtr`.

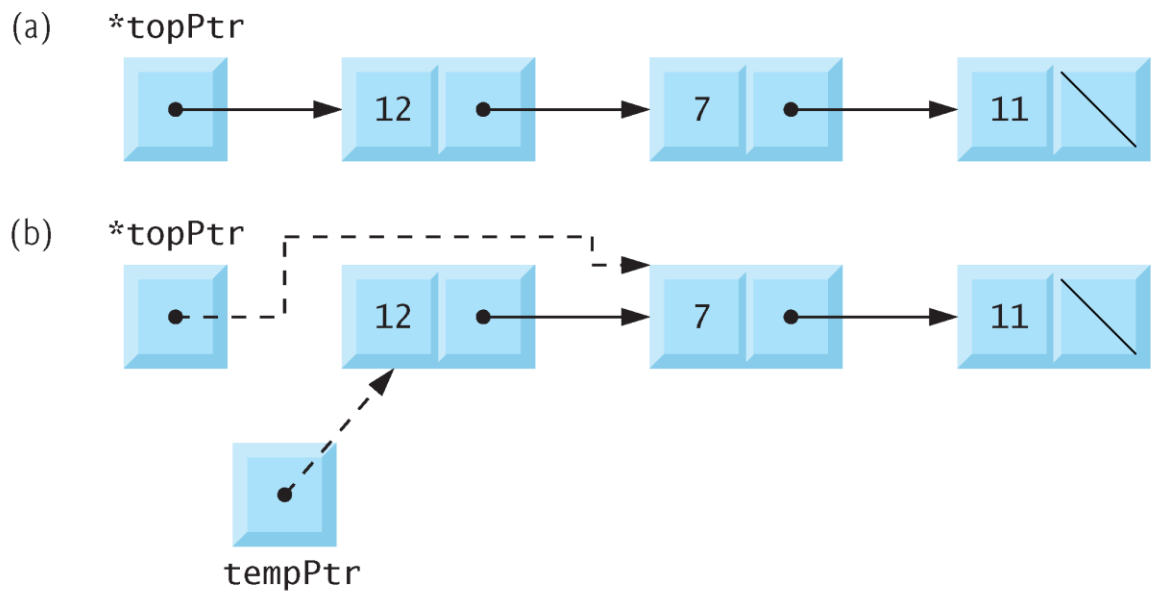

Fig. 12.11 | pop operation.

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.
- 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.

12.5.3 Applications of Stacks (Cont.)

- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- Stacks contain the space created for *automatic 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 no longer are known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

12.6 Queues

- Another common data structure is the queue.
- A queue is similar to a checkout line in a grocery store—the *first* person in line is *serviced first*, and other customers enter the line only at the *end* and *wait* to be serviced.
- Queue nodes are removed *only* from the head of the queue and are inserted *only* at the tail of the queue.
- 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 and dequeue, respectively.

12.6 Queues (Cont.)

- 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*.

12.6 Queues (Cont.)

- Queues are also used to support *print spooling*.
- A multiuser environment may have only a single printer.
- Many users may be generating outputs to be printed.
- If the printer is busy, other outputs may still be generated.
- These are spooled to disk where they wait in a *queue* until the printer becomes available.

12.6 Queues (Cont.)

- Information packets also wait in queues in computer networks.
- 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.
- Figure 12.12 illustrates a queue with several nodes.
- Note the pointers to the head of the queue and the tail of the queue.

Common Programming Error 12.6

Not setting the link in the last node of a queue to `NULL` *can lead to runtime errors.*

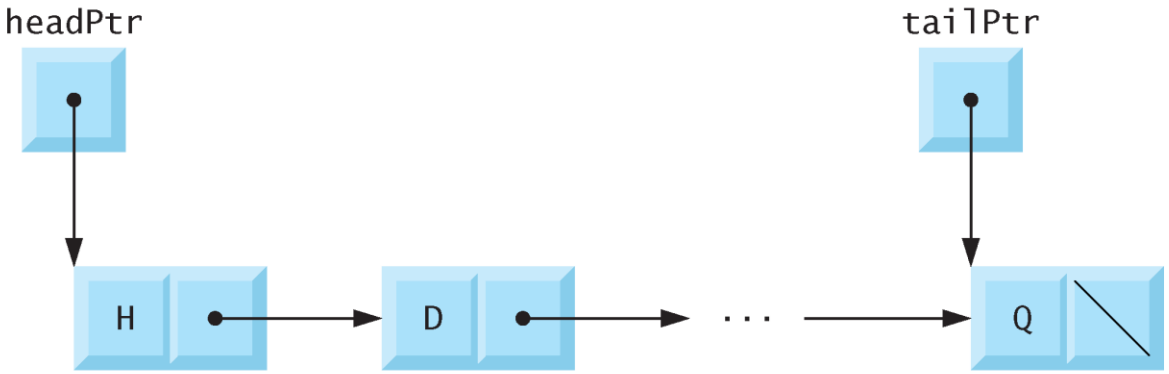

Fig. 12.12 | Queue graphical representation.

12.6 Queues (Cont.)

- Figure 12.13 (output in Fig. 12.14) performs queue manipulations.
- The program provides several options: *insert* a node in the queue (function enqueue), *remove* a node from the queue (function dequeue) and terminate the program.

```c
// Fig. 12.13: fig12_13.c
// Operating and maintaining a queue
#include <stdio.h>
#include <stdlib.h>

// 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);
char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value);
void instructions(void);

// function main begins program execution
int main(void)
{
```

Fig. 12.13 | Operating and maintaining a queue. (Part 1 of 7.)

```
25      QueueNodePtr headPtr = NULL; // initialize headPtr
26      QueueNodePtr tailPtr = NULL; // initialize tailPtr
27      char item; // char input by user
28
29      instructions(); // display the menu
30      printf("%s", "? ");
31      unsigned int choice; // user's menu choice
32      scanf("%u", &choice);
33
34      // while user does not enter 3
35      while (choice != 3) {
36
37         switch(choice) {
38            // enqueue value
39            case 1:
40               printf("%s", "Enter a character: ");
41               scanf("\n%c", &item);
42               enqueue(&headPtr, &tailPtr, item);
43               printQueue(headPtr);
44               break;
```

Fig. 12.13 | Operating and maintaining a queue. (Part 2 of 7.)

```
45            // dequeue value
46            case 2:
47               // if queue is not empty
48               if (!isEmpty(headPtr)) {
49                  item = dequeue(&headPtr, &tailPtr);
50                  printf("%c has been dequeued.\n", item);
51               }
52
53               printQueue(headPtr);
54               break;
55            default:
56               puts("Invalid choice.\n");
57               instructions();
58               break;
59         }
60
61         printf("%s", "? ");
62         scanf("%u", &choice);
63      }
64
65      puts("End of run.");
66   }
67
```

Fig. 12.13 | Operating and maintaining a queue. (Part 3 of 7.)

```
68  // display program instructions to user
69  void instructions(void)
70  {
71     printf ("Enter your choice:\n"
72             "   1 to add an item to the queue\n"
73             "   2 to remove an item from the queue\n"
74             "   3 to end\n");
75  }
76
```

Fig. 12.13 | Operating and maintaining a queue. (Part 4 of 7.)

```
77  // insert a node at queue tail
78  void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value)
79  {
80     QueueNodePtr newPtr = malloc(sizeof(QueueNode));
81
82     if (newPtr != NULL) { // is space available?
83        newPtr->data = value;
84        newPtr->nextPtr = NULL;
85
86        // if empty, insert node at head
87        if (isEmpty(*headPtr)) {
88           *headPtr = newPtr;
89        }
90        else {
91           (*tailPtr)->nextPtr = newPtr;
92        }
93
94        *tailPtr = newPtr;
95     }
96     else {
97        printf("%c not inserted. No memory available.\n", value);
98     }
99  }
100
```

Fig. 12.13 | Operating and maintaining a queue. (Part 5 of 7.)

```
101 // remove node from queue head
102 char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr)
103 {
104    char value = (*headPtr)->data;
105    QueueNodePtr tempPtr = *headPtr;
106    *headPtr = (*headPtr)->nextPtr;
107
108    // if queue is empty
109    if (*headPtr == NULL) {
110       *tailPtr = NULL;
111    }
112
113    free(tempPtr);
114    return value;
115 }
116
117 // return 1 if the queue is empty, 0 otherwise
118 int isEmpty(QueueNodePtr headPtr)
119 {
120    return headPtr == NULL;
121 }
122
```

Fig. 12.13 | Operating and maintaining a queue. (Part 6 of 7.)

```
123 // print the queue
124 void printQueue(QueueNodePtr currentPtr)
125 {
126    // if queue is empty
127    if (currentPtr == NULL) {
128       puts("Queue is empty.\n");
129    }
130    else {
131       puts("The queue is:");
132
133       // while not end of queue
134       while (currentPtr != NULL) {
135          printf("%c --> ", currentPtr->data);
136          currentPtr = currentPtr->nextPtr;
137       }
138
139       puts("NULL\n");
140    }
141 }
```

Fig. 12.13 | Operating and maintaining a queue. (Part 7 of 7.)

```
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
```

Fig. 12.14 | Sample output from the program in Fig. 12.13. (Part 1 of 2.)

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

? 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. 12.14 | Sample output from the program in Fig. 12.13. (Part 2 of 2.)

12.6.1 Function enqueue

- Function `enqueue` receives three arguments from `main`: the address of the *pointer* to the *head of the queue*, the *address* of the p*ointer to the tail of the queue* and the *value* to be inserted in the queue.

12.6 Queues (Cont.)

- The function consists of three steps:
  - To create a new node: Call `malloc`, assign the allocated memory location to `newPtr`, assign the value to be inserted in the queue to `newPtr->data` and assign `NULL` to `newPtr->nextPtr`
  - If the queue is empty, assign `newPtr` to `*headPtr`, because the new node will be both the head and tail of the queue; otherwise, assign pointer `newPtr` to `(*tailPtr)->nextPtr`, because the new node will be placed after the previous tail node.
  - Assign `newPtr` to `*tailPtr`, because the new node is the queue's tail.

12.6 Queues (Cont.)

- Figure 12.15 illustrates an enqueue operation.
- Part (a) shows the queue and the new node *before* the operation.
- The dotted arrows in part (b) illustrate *Steps 2* and *3* of function enqueue that enable a new node to be added to the *end* of a queue that is not empty.

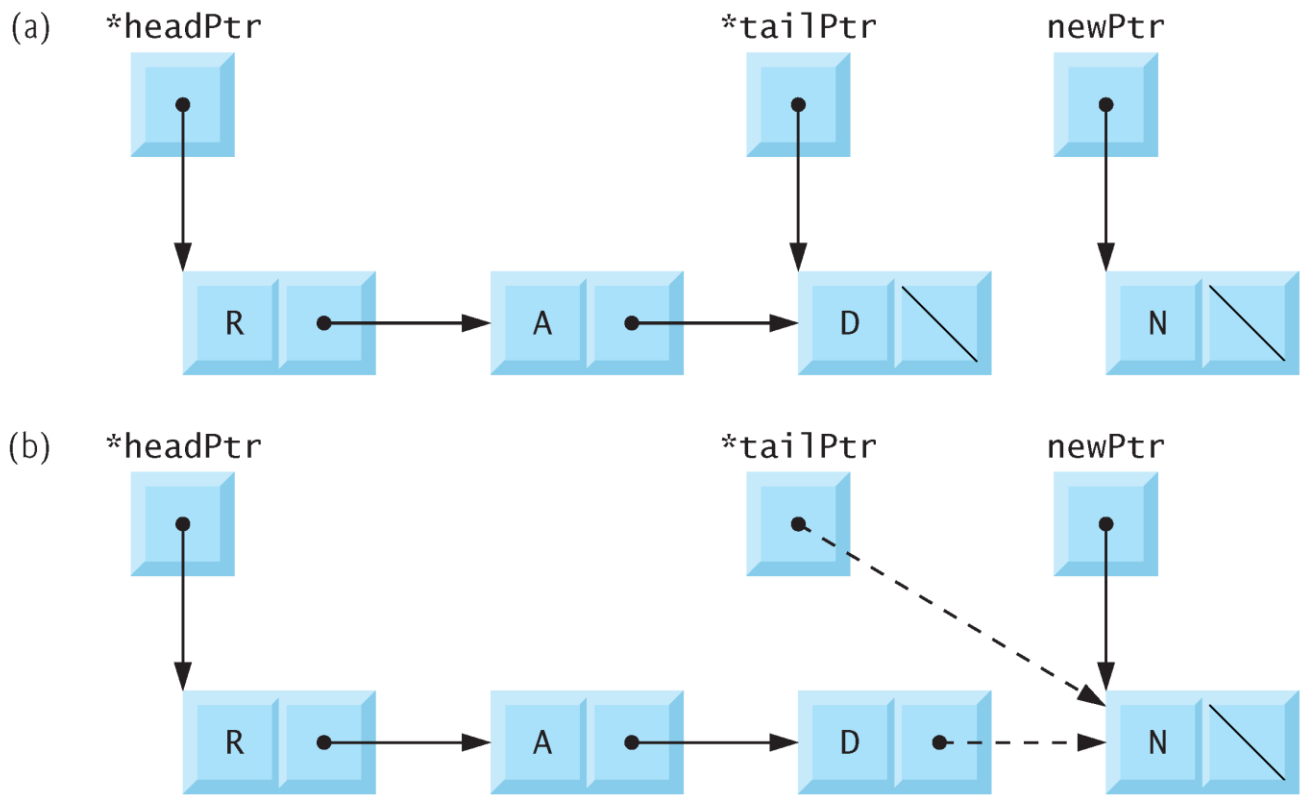

Fig. 12.15 | **enqueue** operation.

12.6.2 Function dequeue

- Function dequeue receives the *address* of the *pointer to the head of the queue* and the *address* of the *pointer to the tail of the queue* as arguments and removes the *first* node from the queue.

12.6.2 Function dequeue

- The `dequeue` operation consists of six steps:
  - Assign `(*headPtr)->data` to `value` to save the data
  - Assign `*headPtr` to `tempPtr`, which will be used to `free` the unneeded memory
  - Assign `(*headPtr)->nextPtr` to `*headPtr` so that `*headPtr` now points to the new first node in the queue
  - If `*headPtr` is `NULL`, assign `NULL` to `*tailPtr` because the queue is now empty.
  - Free the memory pointed to by `tempPtr`
  - Return `value` to the caller

12.6 Queues (Cont.)

- Figure 12.16 illustrates function `dequeue`.
- Part (a) shows the queue *after* the preceding `enqueue` operation.
- Part (b) shows `tempPtr` pointing to the *dequeued node*, and `headPtr` pointing to the new *first node* of the queue.
- Function `free` is used to *reclaim the memory* pointed to by `tempPtr`.

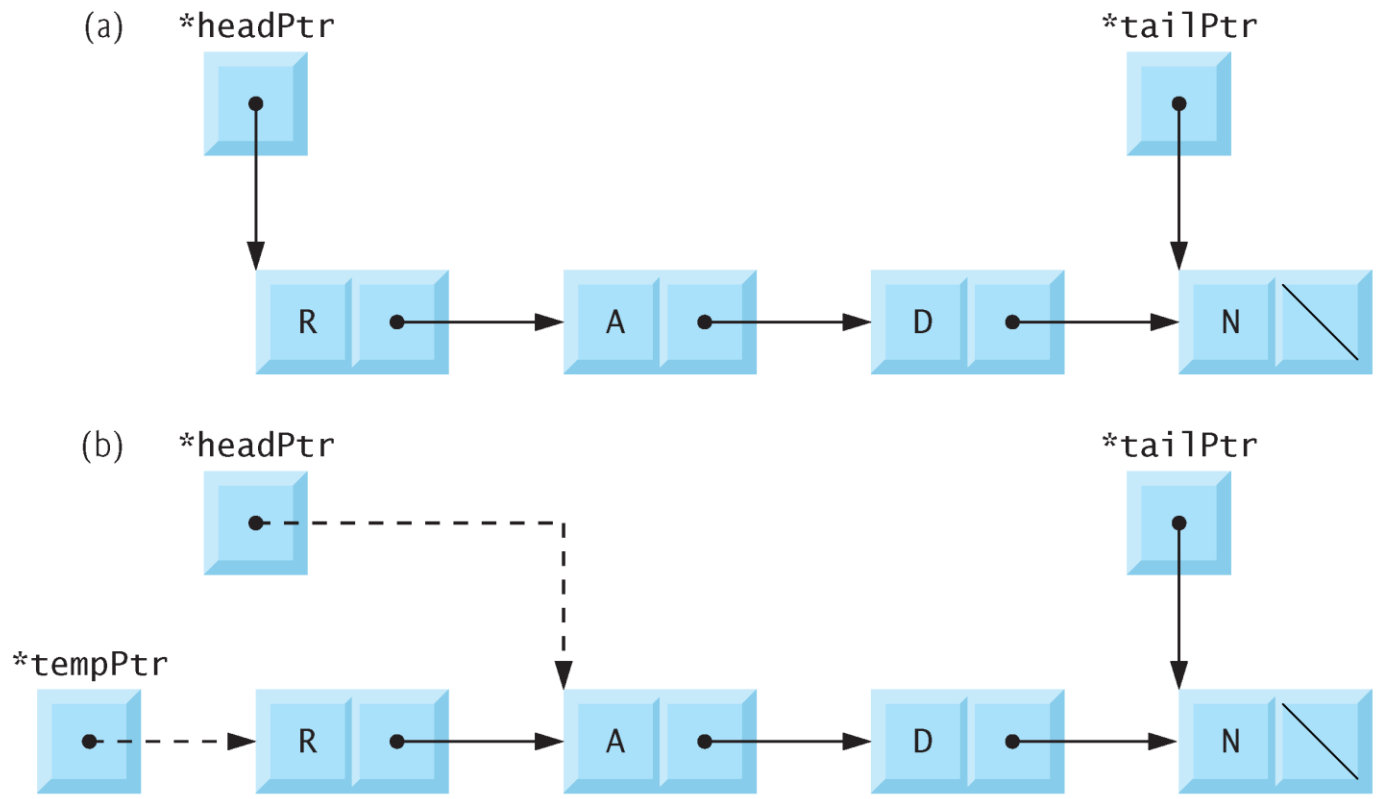

Fig. 12.16 | **dequeue** operation.

12.7 Trees

- Linked lists, stacks and queues are linear data structures.
- A tree is a *nonlinear, two-dimensional data structure* with special properties.
- Tree nodes contain *two or more* links.

12.7 Trees (Cont.)

- This section discusses binary trees (Fig. 12.17)—trees whose nodes all contain two links (none, one, or both of which 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 node in the right subtree.
- The children of a node are called siblings.
- A node with no children is called a leaf node.
- Computer scientists normally draw trees from the root node down—exactly the opposite of trees in nature.

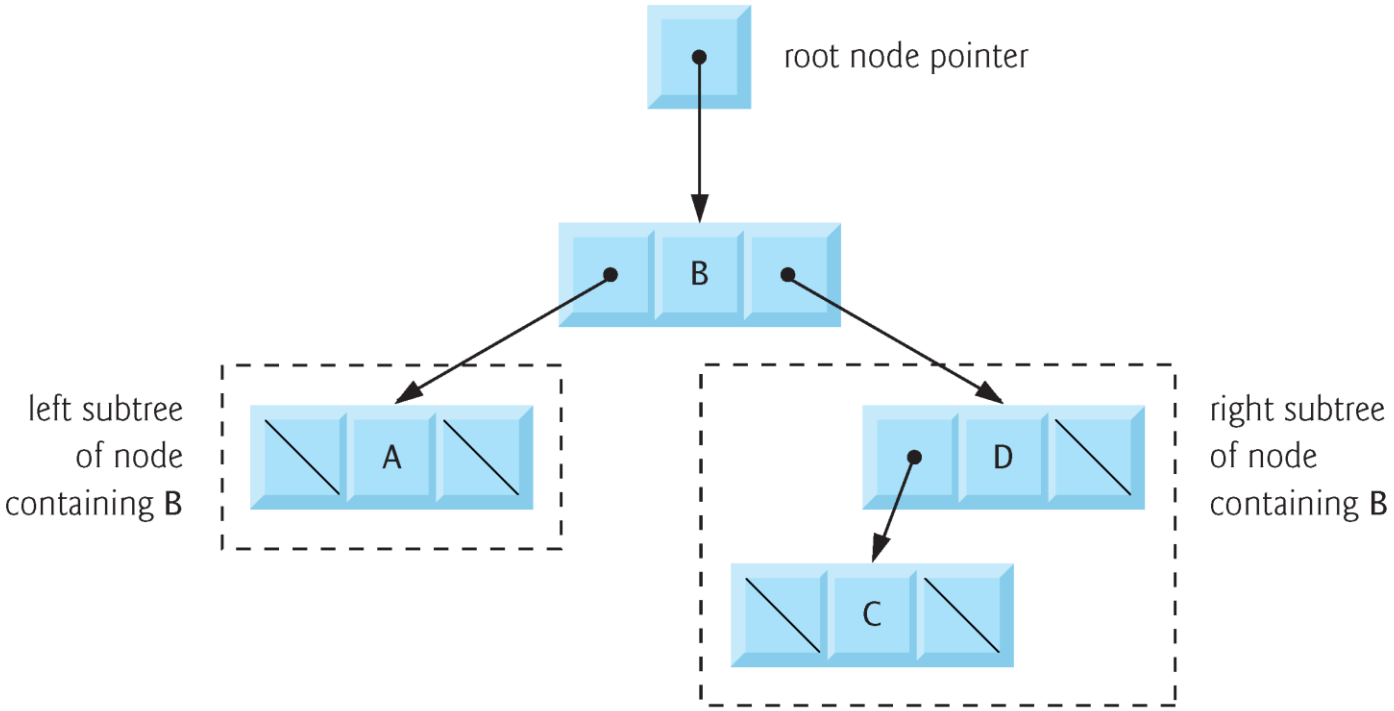

Fig. 12.17 | Binary tree graphical representation.

12.7 Trees (Cont.)

- In this section, a special binary tree called a binary search tree is created.
- A binary search tree (with no duplicate node values) 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.
- Figure 12.18 illustrates a binary search tree with 12 values.
- The shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

Common Programming Error 12.7

Not setting to `NULL` *the links in leaf nodes of a tree can lead to runtime errors.*

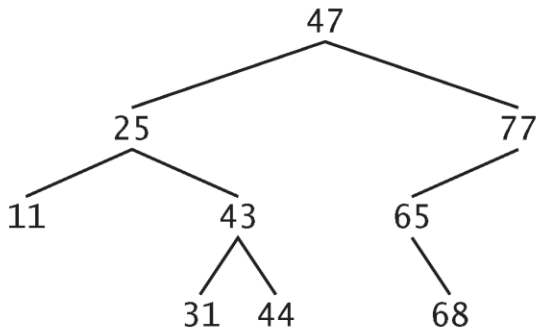

Fig. 12.18 | Binary search tree.

12.7 Trees (Cont.)

- Figure 12.19 (output shown in Fig. 12.20) creates a binary search tree and *traverses* it three ways—inorder, preorder and postorder.
- The program generates 10 random numbers and inserts each in the tree, except that *duplicate* values are *discarded*.

```c
// Fig. 12.19: fig12_19.c
// Creating and traversing a binary tree
// preorder, inorder, and postorder
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// self-referential structure
struct treeNode {
   struct treeNode *leftPtr; // pointer to left subtree
   int data; // node value
   struct treeNode *rightPtr; // pointer to right subtree
};

typedef struct treeNode TreeNode; // synonym for struct treeNode
typedef TreeNode *TreeNodePtr; // synonym for TreeNode*

// prototypes
void insertNode(TreeNodePtr *treePtr, int value);
void inOrder(TreeNodePtr treePtr);
void preOrder(TreeNodePtr treePtr);
void postOrder(TreeNodePtr treePtr);

```

Fig. 12.19 | Creating and traversing a binary tree. (Part 1 of 6.)

```
24  // function main begins program execution
25  int main(void)
26  {
27     TreeNodePtr rootPtr = NULL; // tree initially empty
28
29     srand(time(NULL));
30     puts("The numbers being placed in the tree are:");
31
32     // insert random values between 0 and 14 in the tree
33     for (unsigned int i = 1; i <= 10; ++i) {
34        int item = rand() % 15;
35        printf("%3d", item);
36        insertNode(&rootPtr, item);
37     }
38
39     // traverse the tree preOrder
40     puts("\n\nThe preOrder traversal is:");
41     preOrder(rootPtr);
42
43     // traverse the tree inOrder
44     puts("\n\nThe inOrder traversal is:");
45     inOrder(rootPtr);
46
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 2 of 6.)

```
47      // traverse the tree postOrder
48      puts("\n\nThe postOrder traversal is:");
49      postOrder(rootPtr);
50   }
51
52   // insert node into tree
53   void insertNode(TreeNodePtr *treePtr, int value)
54   {
55      // if tree is empty
56      if (*treePtr == NULL) {
57         *treePtr = malloc(sizeof(TreeNode));
58
59         // if memory was allocated, then assign data
60         if (*treePtr != NULL) {
61            (*treePtr)->data = value;
62            (*treePtr)->leftPtr = NULL;
63            (*treePtr)->rightPtr = NULL;
64         }
65         else {
66            printf("%d not inserted. No memory available.\n", value);
67         }
68      }
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 3 of 6.)

```
69      else { // tree is not empty
70         // data to insert is less than data in current node
71         if (value < (*treePtr)->data) {
72            insertNode(&((*treePtr)->leftPtr), value);
73         }
74
75         // data to insert is greater than data in current node
76         else if (value > (*treePtr)->data) {
77            insertNode(&((*treePtr)->rightPtr), value);
78         }
79         else { // duplicate data value ignored
80            printf("%s", "dup");
81         }
82      }
83   }
84
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 4 of 6.)

```
85  // begin inorder traversal of tree
86  void inOrder(TreeNodePtr treePtr)
87  {
88     // if tree is not empty, then traverse
89     if (treePtr != NULL) {
90        inOrder(treePtr->leftPtr);
91        printf("%3d", treePtr->data);
92        inOrder(treePtr->rightPtr);
93     }
94  }
95
96  // begin preorder traversal of tree
97  void preOrder(TreeNodePtr treePtr)
98  {
99     // if tree is not empty, then traverse
100    if (treePtr != NULL) {
101       printf("%3d", treePtr->data);
102       preOrder(treePtr->leftPtr);
103       preOrder(treePtr->rightPtr);
104    }
105 }
106
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 5 of 6.)

```
107 // begin postorder traversal of tree
108 void postOrder(TreeNodePtr treePtr)
109 {
110    // if tree is not empty, then traverse
111    if (treePtr != NULL) {
112       postOrder(treePtr->leftPtr);
113       postOrder(treePtr->rightPtr);
114       printf("%3d", treePtr->data);
115    }
116 }
```

Fig. 12.19 | Creating and traversing a binary tree. (Part 6 of 6.)

```
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. 12.20 | Sample output from the program of Fig. 12.19.

12.7.1 Function `insertNode`

- The functions used in Fig. 12.19 to create a binary search tree and traverse the tree are recursive.
- Function `insertNode` receives the *address of the tree* and an *integer* to be stored in the tree as arguments.
- *A node can be inserted only as a leaf node in a binary search tree.*

12.7.1 Function `insertNode` (Cont.)

- The steps for inserting a node in a binary search tree are as follows:
  - If `*treePtr` is `NULL`, create a new node.
  - Call `malloc`, assign the allocated memory to `*treePtr`
  - Assign to `(*treePtr)->data` the integer to be stored
  - Assign to `(*treePtr)->leftPtr` and `(*treePtr)->rightPtr` the value `NULL`
  - Return control to the caller (either `main` or a previous call to `insertNode`)

12.7 Trees (Cont.)

- If `*treePtr` is not `NULL` and the value to be inserted is less than `(*treePtr)->data`, function `insertNode` is called with the address of `(*treePtr)->leftPtr` to insert the node in the left subtree of the node pointed to by `treePtr`.
- If the value to be inserted is *greater than* `(*treePtr)->data`, `insertNode` is called with the address of `(*treePtr)->rightPtr` to insert the node in the right subtree of the node pointed to by `treePtr`
- Otherwise, the *recursive steps* continue until a `NULL` pointer is found, then Step *1* is executed to *insert the new node.*

12.7.2 Traversals: Functions inOrder, preOrder and postOrder

- Functions `inOrder`, `preOrder` and `postOrder` each receive a *tree* (i.e., the *pointer to the root node of the tree*) and *traverse* the tree.
- The steps for an `inOrder` traversal are:
  - Traverse the left subtree `inOrder`.
  - Process the value in the node.
  - Traverse the right subtree `inOrder`.
- The value in a node is not processed until the values in its left subtree are processed.
- The `inOrder` traversal of the tree in Fig. 12.21 is:
  - `6 13 17 27 33 42 48`

12.7.2 Traversals: Functions inOrder, preOrder and postOrder

- The `inOrder` traversal of a binary search tree prints the node values in *ascending* order.
- The process of creating a binary search tree actually sorts the data—and thus this process is called the binary tree sort.

12.7.2 Traversals: Functions inOrder, preOrder and postOrder

The steps for a `preOrder` traversal are:

- Process the value in the node.
- Traverse the left subtree `preOrder`.
- Traverse the right subtree `preOrder`.

- The value in each node is processed as the node is visited.
- After the value in a given node is processed, the values in the left subtree are processed, then those in the *right* subtree are processed.

12.7.2 Traversals: Functions inOrder, preOrder and postOrder

- The `preOrder` traversal of the tree in Fig. 12.21 is:
  - `27 13 6 17 42 33 48`
- The steps for a `postOrder` traversal are:
  - Traverse the left subtree `postOrder`.
  - Traverse the right subtree `postOrder`.
  - Process the value in the node.
- The value in each node is not printed until the values of its children are printed.
- The `postOrder` traversal of the tree in Fig. 12.21 is:
  - `6 17 13 33 48 42 27`

12.7.3 Duplicate Elimination

- The binary search tree facilitates duplicate elimination.
- As the tree is being 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.

12.7.4 Binary Tree Search

- Searching a binary tree for a value that matches a key value is also 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 a maximum of $\log_2 n$ comparisons would have to be made either to find a match or to determine that no match exists.
- This means, for example, that when searching a (tightly packed) 1,000,000-element binary search tree, no more than 10 comparisons need to be made because $2^{20} > 1{,}000{,}000$.

12.7.5 Other Binary Tree Operations

- When searching a (tightly packed) 1,000,000 element binary search tree, no more than 20 comparisons need to be made because $2^{20} > 1{,}000{,}000$.
- The *level order traversal* of a binary tree visits the nodes of the tree *row-by-row* starting at the root node level.
- On each level of the tree, the nodes are visited from left to right.

Chapter 8 of the CERT Secure C Coding Standard

- Chapter 8 of the CERT Secure C Coding Standard is dedicated to memory-management recommendations and rules—many apply to the uses of pointers and dynamic-memory allocation presented in this chapter.
- For more information, visit

  `www.securecoding.cert.org`.

12.8 Secure C Programming (Cont.)

MEM01-C/MEM30-C:

- Pointers should not be left uninitialized.
- They should be assigned either `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.

12.8 Secure C Programming (Cont.)

- Using a pointer that's been freed can lead to program crashes and security vulnerabilities.
- When you free dynamically allocated memory, you should 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.

12.8 Secure C Programming (Cont.)

MEM31-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—attempting to free a NULL pointer has no effect.

12.8 Secure C Programming (Cont.)

MEM32-C:

- Function `malloc` 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.