

# 12

# C Data Structures



*Much that I bound, I could not free;  
Much that I freed returned to me.*

—Lee Wilson Dodd

*‘Will you walk a little faster?’ said  
a whiting to a snail,  
‘There’s a porpoise close behind us,  
and he’s treading on my tail.*

—Lewis Carroll



*There is always room at the top.*

—Daniel Webster

*Push on — keep moving*

—Thomas Morton

*I think that I shall never see  
A poem lovely as a tree.*

—Joyce Kilmer



# OBJECTIVES

In this chapter you will learn:

- To allocate and free memory dynamically for data objects.
- To form linked data structures using pointers, self-referential structures and recursion.
- To create and manipulate linked lists, queues, stacks and binary trees.
- Various important applications of linked data structures.



# Outline

- 12.1 Introduction**
- 12.2 Self-Referential Structures**
- 12.3 Dynamic Memory Allocation**
- 12.4 Linked Lists**
- 12.5 Stacks**
- 12.6 Queues**
- 12.7 Trees**



# 12.1 Introduction

- **Dynamic data structures**
  - Data structures that grow and shrink during execution
- **Linked lists**
  - Allow insertions and removals anywhere
- **Stacks**
  - Allow insertions and removals only at top of stack
- **Queues**
  - Allow insertions at the back and removals from the front
- **Binary trees**
  - High-speed searching and sorting of data and efficient elimination of duplicate data items



# 12.2 Self-Referential Structures

- **Self-referential structures**

- Structure that contains a pointer to a structure of the same type
- Can be linked together to form useful data structures such as lists, queues, stacks and trees
- Terminated with a NULL pointer (0)

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

- **nextPtr**

- Points to an object of type **node**
- Referred to as a **link**
  - Ties one node to another node



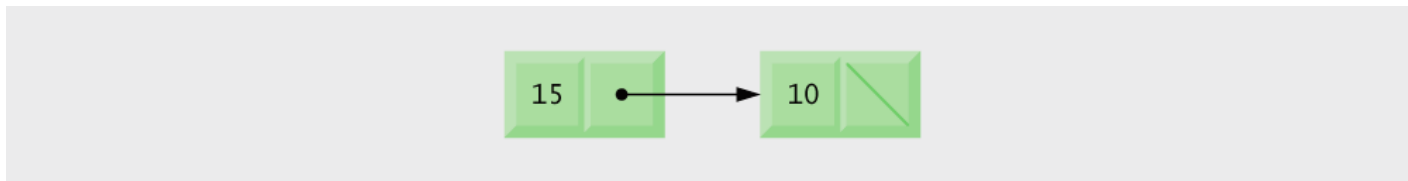
# Common Programming Error 12.1

---

**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

- **Dynamic memory allocation**
  - Obtain and release memory during execution
- **malloc**
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available, returns `NULL`
  - Example

```
newPtr = malloc( sizeof( struct node ) );
```
- **free**
  - Deallocates memory allocated by `malloc`
  - Takes a pointer as an argument
  - `free ( newPtr );`



## 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).**



# Common Programming Error 12.2

---

**Assuming that the size of a structure is simply the sum of the sizes of its members is a logic error.**



# Good Programming Practice 12.1

---

**Use the `sizeof` operator to determine the size of a structure.**



# Error-Prevention Tip 12.1

---

**When using `malloc`, test for a NULL pointer return value. Print an error message if the requested memory is not allocated.**



# Common Programming Error 12.3

---

**Not returning dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely. This is sometimes called a “memory leak.”**



## Good Programming Practice 12.2

---

**When memory that was dynamically allocated is no longer needed, use `free` to return the memory to the system immediately.**





# Common Programming Error 12.4

---

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



# Common Programming Error 12.5

---

**Referring to memory that has been freed is an error.**



# 12.4 Linked Lists

- **Linked list**
  - **Linear collection of self-referential class objects, called nodes**
  - **Connected by pointer links**
  - **Accessed via a pointer to the first node of the list**
  - **Subsequent nodes are accessed via the link-pointer member of the current node**
  - **Link pointer in the last node is set to NULL to mark the list's end**
- **Use a linked list instead of an array when**
  - **You have an unpredictable number of data elements**
  - **Your list needs to be sorted quickly**



## 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.**



## 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.**

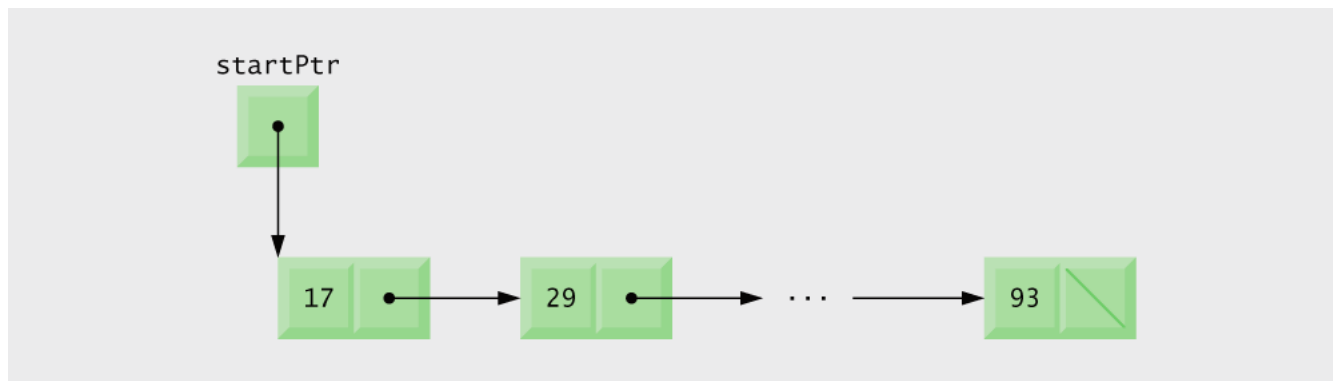


## Performance Tip 12.3

---

**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.**





**Fig. 12.2** | Linked list graphical representation.



## 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.**





## Outline

Each node in the list contains a data element and a pointer to the next node

fig12\_03.c

(1 of 8)

```

1  /* Fig. 12.3: fig12_03.c
2     Operating and maintaining a list */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* self-referential structure */
7  struct listNode {
8     char data; /* each listNode contains a character */
9     struct listNode *nextPtr; /* pointer to next node */
10 }; /* end structure listNode */
11
12 typedef struct listNode ListNode; /* synonym for struct listNode */
13 typedef ListNode *ListNodePtr; /* synonym for ListNode* */
14
15 /* prototypes */
16 void insert( ListNodePtr *sPtr, char value );
17 char delete( ListNodePtr *sPtr, char value );
18 int isEmpty( ListNodePtr sPtr );
19 void printList( ListNodePtr currentPtr );
20 void instructions( void );
21
22 int main( void )
23 {
24     ListNodePtr startPtr = NULL; /* initially there are no nodes */
25     int choice; /* user's choice */
26     char item; /* char entered by user */
27
28     instructions(); /* display the menu */
29     printf( "? " );
30     scanf( "%d", &choice );

```



Outline

fig12\_03.c

(2 of 8)

```

31  /* loop while user does not choose 3 */
32  while ( choice != 3 ) {
33
34
35      switch ( choice ) {
36
37          case 1:
38              printf( "Enter a character: " );
39              scanf( "\n%c", &item );
40              insert( &startPtr, item ); /* insert item in list */
41              printList( startPtr );
42              break;
43
44          case 2: /* delete an element */
45
46              /* if list is not empty */
47              if ( !isEmpty( startPtr ) ) {
48                  printf( "Enter character to be deleted: " );
49                  scanf( "\n%c", &item );
50
51                  /* if character is found, remove it*/
52                  if ( delete( &startPtr, item ) ) { /* remove item */
53                      printf( "%c deleted.\n", item );
54                      printList( startPtr );
55                  } /* end if */
56                  else {
57                      printf( "%c not found.\n\n", item );
58                  } /* end else */
59
60              } /* end if */

```

Function **insert** inserts data into the list

Function **delete** removes data from the list



## Outline

fig12\_03.c

(3 of 8)

```
61     else {
62         printf( "List is empty.\n\n" );
63     } /* end else */
64
65     break;
66
67     default:
68         printf( "Invalid choice.\n\n" );
69         instructions();
70         break;
71
72 } /* end switch */
73
74 printf( "? " );
75 scanf( "%d", &choice );
76 } /* end while */
77
78 printf( "End of run.\n" );
79
80 return 0; /* indicates successful termination */
81
82 } /* end main */
83
```



## Outline

fig12\_03.c

(4 of 8)

```

84 /* display program instructions to user */
85 void instructions( void )
86 {
87     printf( "Enter your choice:\n"
88           "  1 to insert an element into the list.\n"
89           "  2 to delete an element from the list.\n"
90           "  3 to end.\n" );
91 } /* end function instructions */
92
93 /* Insert a new value into the list in sorted order */
94 void insert( ListNodePtr *sPtr, char value )
95 {
96     ListNodePtr newPtr;      /* pointer to new node */
97     ListNodePtr previousPtr; /* pointer to previous node in list */
98     ListNodePtr currentPtr;  /* pointer to current node in list */
99
100     newPtr = malloc( sizeof( ListNode ) ); /* create node */
101
102     if ( newPtr != NULL ) { /* is space available */
103         newPtr->data = value; /* place value in node */
104         newPtr->nextPtr = NULL; /* node does not link to another node */
105
106         previousPtr = NULL;
107         currentPtr = *sPtr;
108
109         /* loop to find the correct location in the list */
110         while ( currentPtr != NULL && value > currentPtr->data ) {
111             previousPtr = currentPtr; /* walk to ... */
112             currentPtr = currentPtr->nextPtr; /* ... next node */
113         } /* end while */

```

To insert a node into the list, memory must first be allocated for that node

while loop searches for new node's place in the list



## Outline

```

114  /* insert new node at beginning of list */
115  if ( previousPtr == NULL ) {
116      newPtr->nextPtr = *sPtr;
117      *sPtr = newPtr;
118  } /* end if */
119  else { /* insert new node between previousPtr and currentPtr */
120      previousPtr->nextPtr = newPtr;
121      newPtr->nextPtr = currentPtr;
122  } /* end else */
123
124
125 } /* end if */
126 else {
127     printf( "%c not inserted. No memory available.\n", value );
128 } /* end else */
129
130 } /* end function insert */
131
132 /* Delete a list element */
133 char delete( ListNodePtr *sPtr, char value )
134 {
135     ListNodePtr previousPtr; /* pointer to previous node in list */
136     ListNodePtr currentPtr;  /* pointer to current node in list */
137     ListNodePtr tempPtr;     /* temporary node pointer */
138

```

If there are no nodes in the list, the new node becomes the “start” node

fig12\_03.c

(5 of 8)

Otherwise, the new node is inserted between two others (or at the end of the list) by changing pointers



## Outline

fig12\_03.c

(6 of 8)

```

139  /* delete first node */
140  if ( value == ( *sPtr )->data ) {
141      tempPtr = *sPtr; /* hold onto node being removed */
142      *sPtr = ( *sPtr )->nextPtr; /* de-thread the node */
143      free( tempPtr ); /* free the de-threaded node */
144      return value;
145  } /* end if */
146  else {
147      previousPtr = *sPtr;
148      currentPtr = ( *sPtr )->nextPtr;
149
150      /* loop to find the correct location in the list */
151      while ( currentPtr != NULL && currentPtr->data != value ) {
152          previousPtr = currentPtr;          /* walk to ... */
153          currentPtr = currentPtr->nextPtr; /* ... next node */
154      } /* end while */
155
156      /* delete node at currentPtr */
157      if ( currentPtr != NULL ) {
158          tempPtr = currentPtr;
159          previousPtr->nextPtr = currentPtr->nextPtr;
160          free( tempPtr );
161          return value;
162      } /* end if */

```

while loop searches for node's place in the list

Once the node is found, it is deleted by changing pointers and freeing the node's memory



## Outline

fig12\_03.c

(7 of 8)

```
163 } /* end else */
164 } /* end function delete */
165
166 return '\0';
167
168 } /* end function delete */
169
170 /* Return 1 if the list is empty, 0 otherwise */
171 int isEmpty( ListNodePtr sPtr )
172 {
173     return sPtr == NULL;
174
175 } /* end function isEmpty */
176
177 /* Print the list */
```

If the start node is **NULL**, there are no nodes in the list



## Outline

fig12\_03.c

(8 of 8)

```
178 void printList( ListNodePtr currentPtr )
179 {
180
181     /* if list is empty */
182     if ( currentPtr == NULL ) {
183         printf( "List is empty.\n\n" );
184     } /* end if */
185     else {
186         printf( "The list is:\n" );
187
188         /* while not the end of the list */
189         while ( currentPtr != NULL ) {
190             printf( "%c --> ", currentPtr->data );
191             currentPtr = currentPtr->nextPtr;
192         } /* end while */
193
194         printf( "NULL\n\n" );
195     } /* end else */
196
197 } /* end function printList */
```





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.

? 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

*(continued on next slide...)*



## Outline

? 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.

? 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.

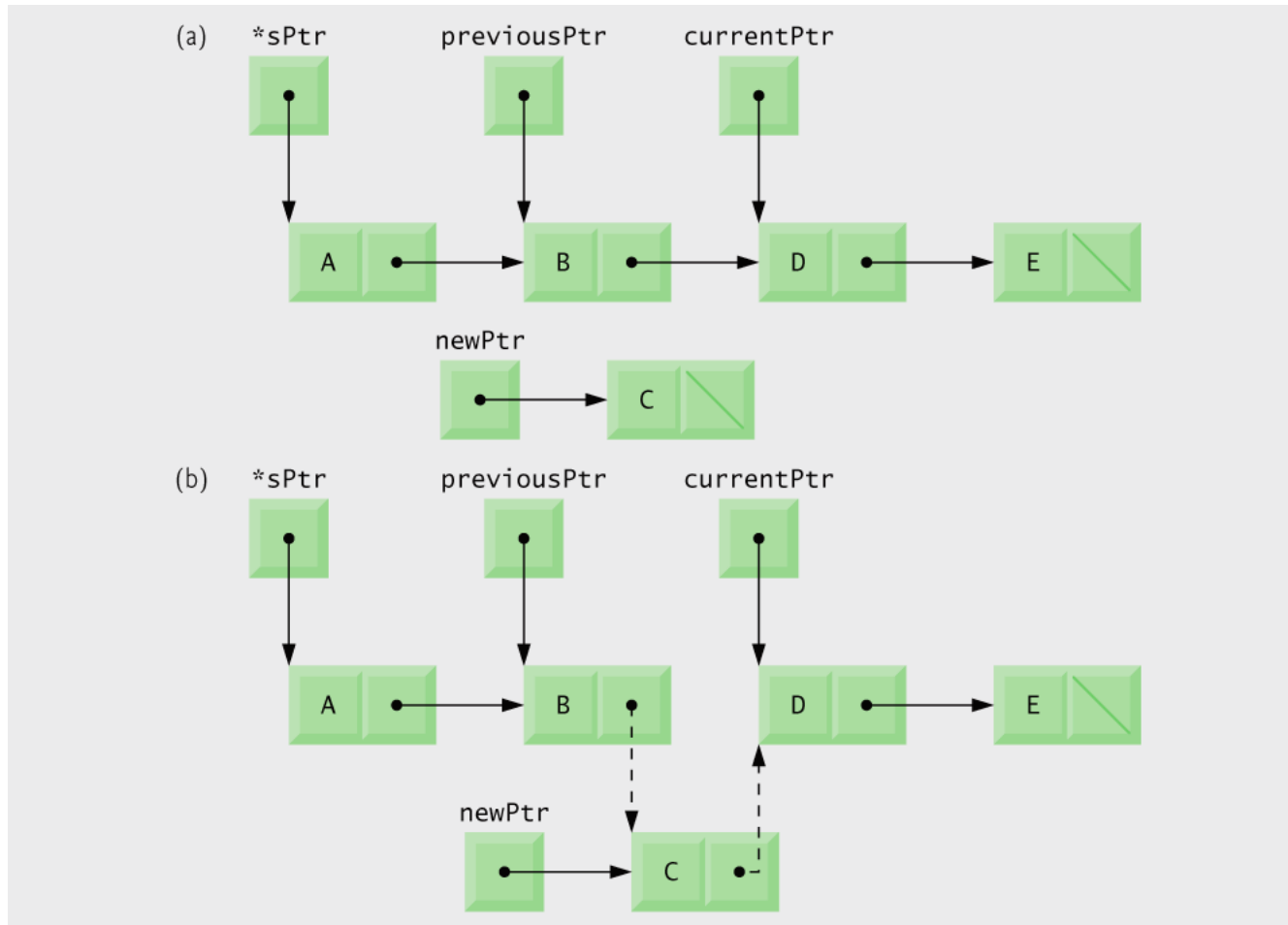


## Error-Prevention Tip 12.2

---

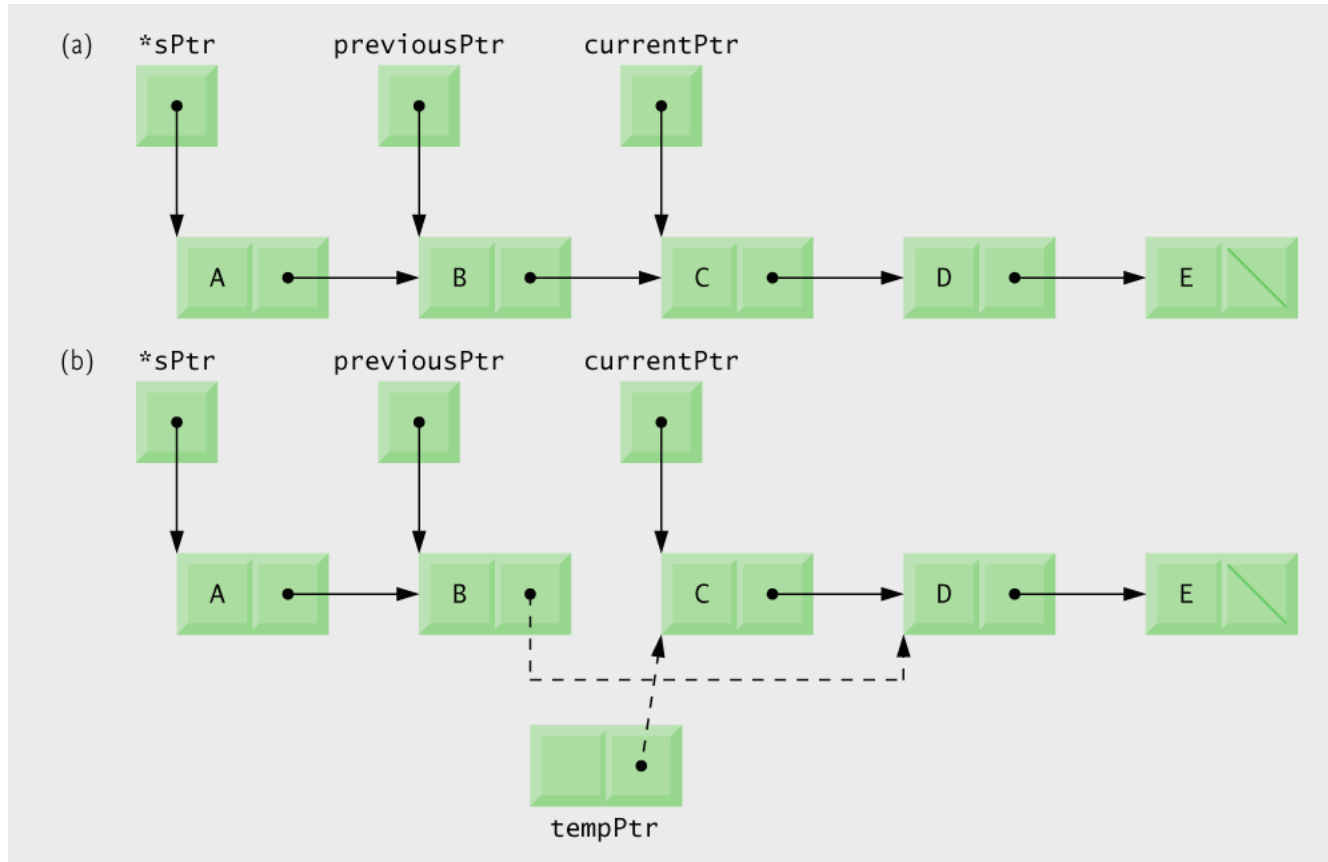
**Assign NULL to the link member of a new node. Pointers should be initialized before they are used.**





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





**Fig. 12.6** | Deleting a node from a list.

# 12.5 Stacks

## ■ Stack

- New nodes can be added and removed only at the top
- Similar to a pile of dishes
- Last-in, first-out (LIFO)
- Bottom of stack indicated by a link member to NULL
- Constrained version of a linked list

## ■ push

- Adds a new node to the top of the stack

## ■ pop

- Removes a node from the top
- Stores the popped value
- Returns true if pop was successful

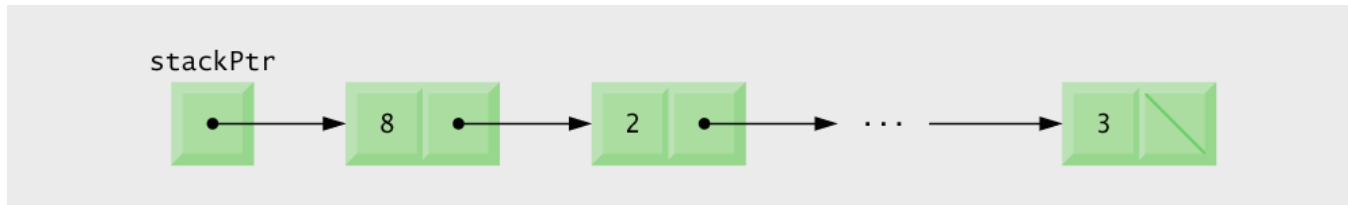


# Common Programming Error 12.6

---

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





**Fig. 12.7** | Stack graphical representation.





## Outline

fig12\_08.c

(1 of 5)

Each node in the stack contains a data element and a pointer to the next node

```

1  /* Fig. 12. 8: fig12_08.c
2     dynamic 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 }; /* end structure stackNode */
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
22 /* function main begins program execution */
23 int main( void )
24 {
25     StackNodePtr stackPtr = NULL; /* points to stack top */
26     int choice; /* user's menu choice */
27     int value; /* int input by user */
28
29     instructions(); /* display the menu */
30     printf( "? " );

```



## Outline

fig12\_08.c

(2 of 5)

```
31 scanf( "%d", &choice );
32
33 /* while user does not enter 3 */
34 while ( choice != 3 ) {
35
36     switch ( choice ) {
37
38         /* push value onto stack */
39         case 1:
40             printf( "Enter an integer: " );
41             scanf( "%d", &value );
42             push( &stackPtr, value );
43             printStack( stackPtr );
44             break;
45
46         /* pop value off stack */
47         case 2:
48
49             /* if stack is not empty */
50             if ( !isEmpty( stackPtr ) ) {
51                 printf( "The popped value is %d.\n", pop( &stackPtr ) );
52             } /* end if */
53
54             printStack( stackPtr );
55             break;
56
57         default:
58             printf( "Invalid choice.\n\n" );
59             instructions();
60             break;
```



## Outline

fig12\_08.c

(3 of 5)

```
61     } /* end switch */
62
63
64     printf( "? " );
65     scanf( "%d", &choice );
66 } /* end while */
67
68 printf( "End of run.\n" );
69
70 return 0; /* indicates successful termination */
71
72 } /* end main */
73
74 /* display program instructions to user */
75 void instructions( void )
76 {
77     printf( "Enter choice:\n"
78           "1 to push a value on the stack\n"
79           "2 to pop a value off the stack\n"
80           "3 to end program\n" );
81 } /* end function instructions */
82
83 /* Insert a node at the stack top */
84 void push( StackNodePtr *topPtr, int info )
85 {
86     StackNodePtr newPtr; /* pointer to new node */
87
88     newPtr = malloc( sizeof( StackNode ) );
89
```

To insert a node into the stack, memory must first be allocated for that node



## Outline

```

90  /* insert the node at stack top */
91  if ( newPtr != NULL ) {
92      newPtr->data = info;
93      newPtr->nextPtr = *topPtr;
94      *topPtr = newPtr;
95  } /* end if */
96  else { /* no space available */
97      printf( "%d not inserted. No memory available.\n", info );
98  } /* end else */
99
100 } /* end function push */
101
102 /* Remove a node from the stack top */
103 int pop( StackNodePtr *topPtr )
104 {
105     StackNodePtr tempPtr; /* temporary node pointer */
106     int popValue; /* node value */
107
108     tempPtr = *topPtr;
109     popValue = ( *topPtr )->data;
110     *topPtr = ( *topPtr )->nextPtr;
111     free( tempPtr );
112
113     return popValue;
114
115 } /* end function pop */
116

```

Stack nodes are always inserted at the top, so there is no need to search for the node's place

Inserted node becomes the new top

fig12\_08.c

(4 of 5)

Stack nodes are always removed from the top, so there is no need to search for the node's place

Second node becomes the new top

Free the memory of the popped node



## Outline

fig12\_08.c

(5 of 5)

```
117  /* Print the stack */
118  void printStack( StackNodePtr currentPtr )
119  {
120
121  /* if stack is empty */
122  if ( currentPtr == NULL ) {
123      printf( "The stack is empty.\n\n" );
124  } /* end if */
125  else {
126      printf( "The stack is:\n" );
127
128      /* while not the end of the stack */
129      while ( currentPtr != NULL ) {
130          printf( "%d --> ", currentPtr->data );
131          currentPtr = currentPtr->nextPtr;
132      } /* end while */
133
134      printf( "NULL\n\n" );
135  } /* end else */
136
137 } /* end function printList */
137 } /* end function printList */
138
139 /* Return 1 if the stack is empty, 0 otherwise */
140 int isEmpty( StackNodePtr topPtr )
141 {
142     return topPtr == NULL;
143
144 } /* end function isEmpty */
```



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

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

*(continued on next slide...)*



## Outline

? 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

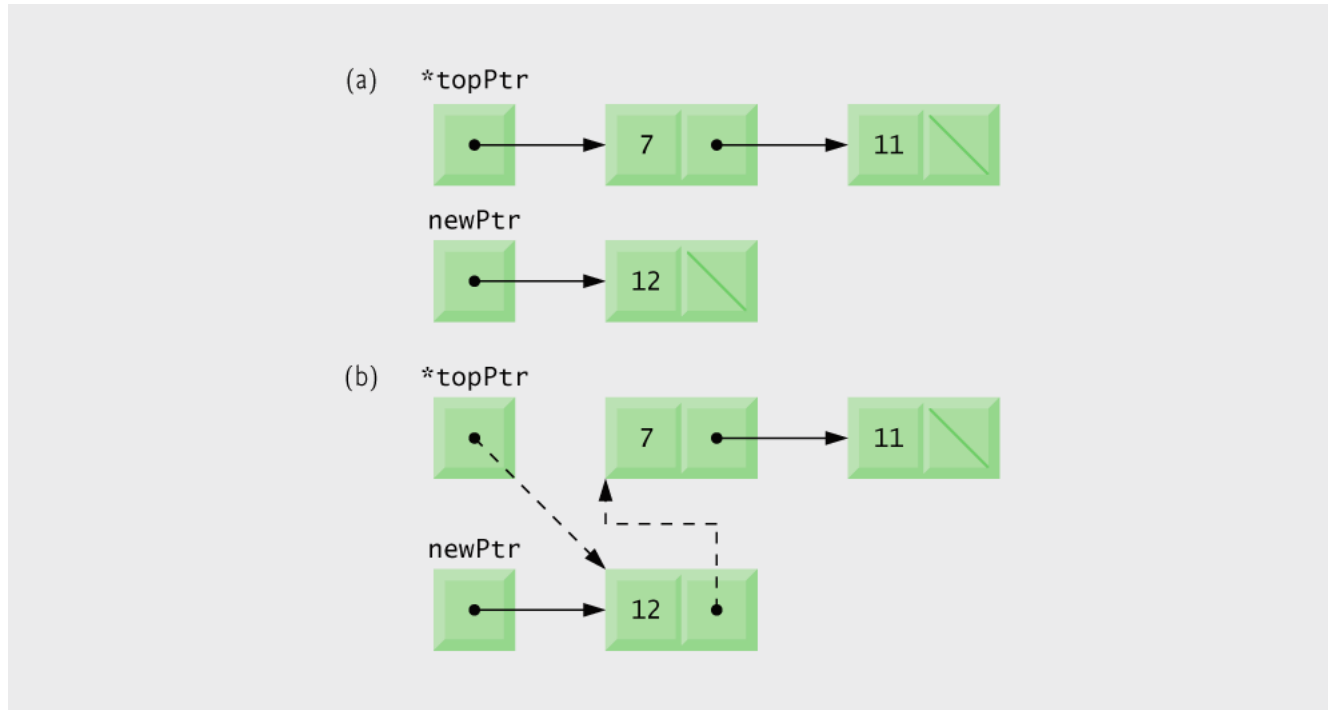
2 to pop a value off the stack

3 to end program

? 3

End of run.

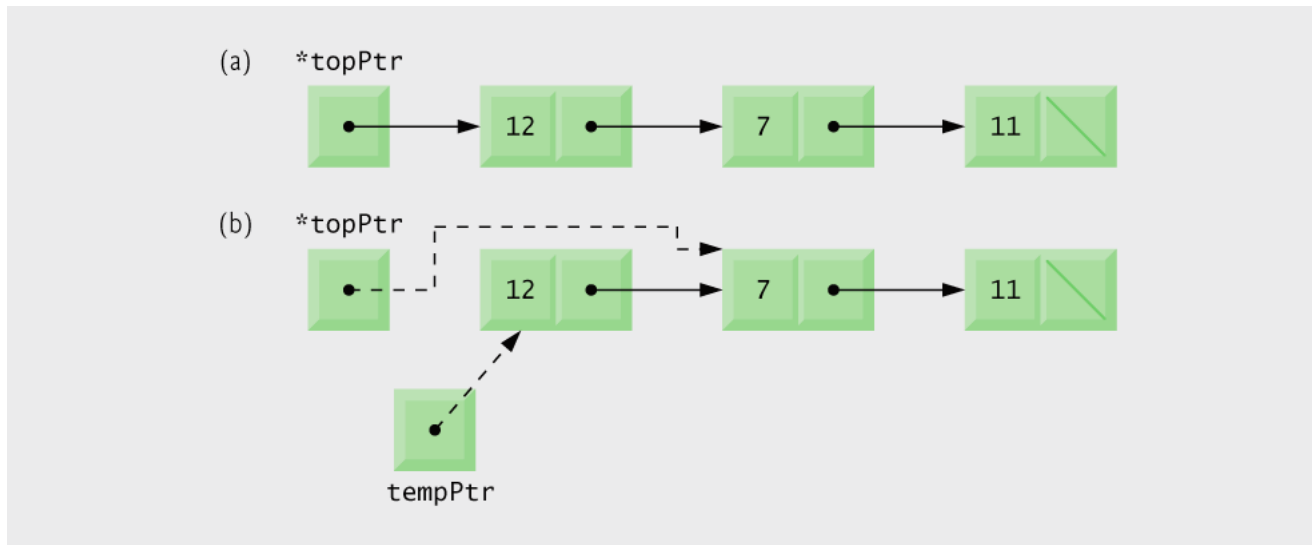




**Fig. 12.10** | push operation.







**Fig. 12.11** | pop operation.



# 12.6 Queues

## ■ Queue

- Similar to a supermarket checkout line
- First-in, first-out (FIFO)
- Nodes are removed only from the head
- Nodes are inserted only at the tail

## ■ Insert and remove operations

- Enqueue (insert) and dequeue (remove)

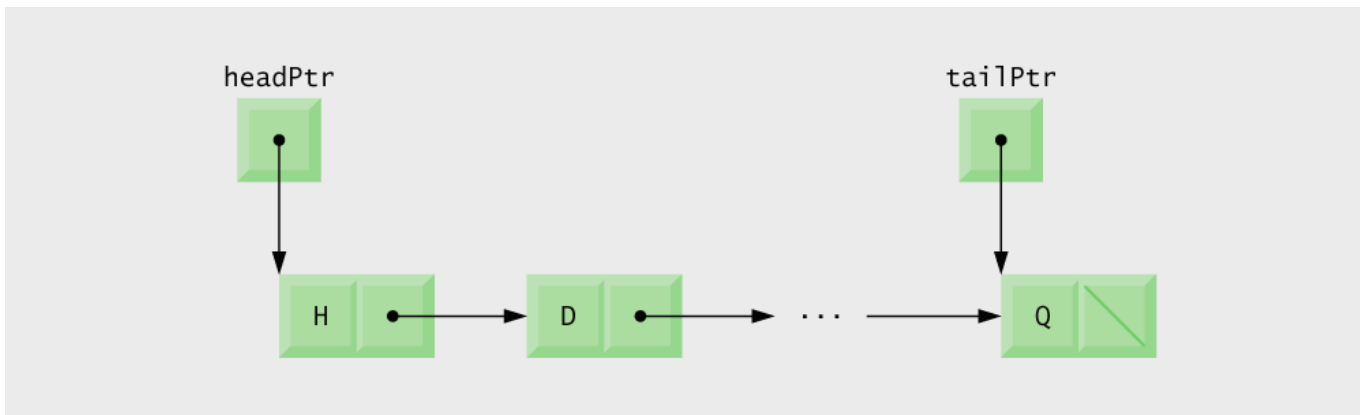


# Common Programming Error 12.7

---

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





**Fig. 12.12** | Queue graphical representation.



## Outline

fig12\_13.c

(1 of 6)

```

1  /* Fig. 12.13: fig12_13.c
2     Operating and maintaining a queue */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  /* self-referential structure */
8  struct queueNode {
9     char data; /* define data as a char */
10    struct queueNode *nextPtr; /* queueNode pointer */
11 }; /* end structure queueNode */
12
13 typedef struct queueNode QueueNode;
14 typedef QueueNode *QueueNodePtr;
15
16 /* function prototypes */
17 void printQueue( QueueNodePtr currentPtr );
18 int isEmpty( QueueNodePtr headPtr );
19 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
20 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
21             char value );
22 void instructions( void );
23
24 /* function main begins program execution */
25 int main( void )
26 {
27     QueueNodePtr headPtr = NULL; /* initialize headPtr */
28     QueueNodePtr tailPtr = NULL; /* initialize tailPtr */
29     int choice; /* user's menu choice */
30     char item; /* char input by user */

```

Each node in the queue contains a data element and a pointer to the next node

Note that unlike linked lists and stacks, queues keep track of the tail node as well as the head



## Outline

fig12\_13.c

(2 of 6)

```
31 instructions(); /* display the menu */
32 printf( "? " );
33 scanf( "%d", &choice );
34
35
36 /* while user does not enter 3 */
37 while ( choice != 3 ) {
38
39     switch( choice ) {
40
41         /* enqueue value */
42         case 1:
43             printf( "Enter a character: " );
44             scanf( "\n%c", &item );
45             enqueue( &headPtr, &tailPtr, item );
46             printQueue( headPtr );
47             break;
48
49         /* dequeue value */
50         case 2:
51
52             /* if queue is not empty */
53             if ( !isEmpty( headPtr ) ) {
54                 item = dequeue( &headPtr, &tailPtr );
55                 printf( "%c has been dequeued.\n", item );
56             } /* end if */
57
58             printQueue( headPtr );
59             break;
```



## Outline

fig12\_13.c

(3 of 6)

```
60     default:
61         printf( "Invalid choice.\n\n" );
62         instructions();
63         break;
64
65
66     } /* end switch */
67
68     printf( "? " );
69     scanf( "%d", &choice );
70 } /* end while */
71
72 printf( "End of run.\n" );
73
74 return 0; /* indicates successful termination */
75
76 } /* end main */
77
78 /* display program instructions to user */
79 void instructions( void )
80 {
81     printf ( "Enter your choice:\n"
82             "  1 to add an item to the queue\n"
83             "  2 to remove an item from the queue\n"
84             "  3 to end\n" );
85 } /* end function instructions */
```



## Outline

fi g12\_13. c

(4 of 6)

```

86 /* insert a node a queue tail */
87 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
88             char value )
89 {
90     QueueNodePtr newPtr; /* pointer to new node */
91
92     newPtr = malloc( sizeof( QueueNode ) );
93
94     if ( newPtr != NULL ) { /* is space available */
95         newPtr->data = value;
96         newPtr->nextPtr = NULL;
97
98         /* if empty, insert node at head */
99         if ( isEmpty( *headPtr ) ) {
100             *headPtr = newPtr;
101         } /* end if */
102         else {
103             ( *tailPtr )->nextPtr = newPtr;
104         } /* end else */
105
106         *tailPtr = newPtr;
107     } /* end if */
108     else {
109         printf( "%c not inserted. No memory available.\n", value );
110     } /* end else */
111 } /* end function enqueue */
112
113

```

To insert a node into the queue, memory must first be allocated for that node

Queue nodes are always inserted at the tail, so there is no need to search for the node's place

If the queue is empty, the inserted node becomes the new head in addition to the new tail

Inserted node becomes the new tail





## Outline

fig12\_13.c

(5 of 6)

```

114 /* remove node from queue head */
115 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
116 {
117     char value;           /* node value */
118     QueueNodePtr tempPtr; /* temporary node pointer */
119
120     value = ( *headPtr )->data;
121     tempPtr = *headPtr;
122     *headPtr = ( *headPtr )->nextPtr;
123
124     /* if queue is empty */
125     if ( *headPtr == NULL ) {
126         *tailPtr = NULL;
127     } /* end if */
128
129     free( tempPtr );
130
131     return value;
132 } /* end function dequeue */
133
134 /* Return 1 if the list is empty, 0 otherwise */
135 int isEmpty( QueueNodePtr headPtr )
136 {
137     return headPtr == NULL;
138 }
139
140 } /* end function isEmpty */

```

Queue nodes are always removed from the head, so there is no need to search for the node's place

Second node becomes the new head

If the removed node is the only node in the queue, it is the tail as well as the head of the queue, so **tailPtr** must be set to **NULL**

Free the memory of the removed node



## Outline

fig12\_13.c

(6 of 6)

```
142
143 /* Print the queue */
144 void printQueue( QueueNodePtr currentPtr )
145 {
146
147     /* if queue is empty */
148     if ( currentPtr == NULL ) {
149         printf( "Queue is empty.\n\n" );
150     } /* end if */
151     else {
152         printf( "The queue is:\n" );
153
154         /* while not end of queue */
155         while ( currentPtr != NULL ) {
156             printf( "%c --> ", currentPtr->data );
157             currentPtr = currentPtr->nextPtr;
158         } /* end while */
159
160         printf( "NULL\n\n" );
161     } /* end else */
162
163 } /* end function printQueue */
```



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

*(continued on next slide...)*



## Outline

? 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

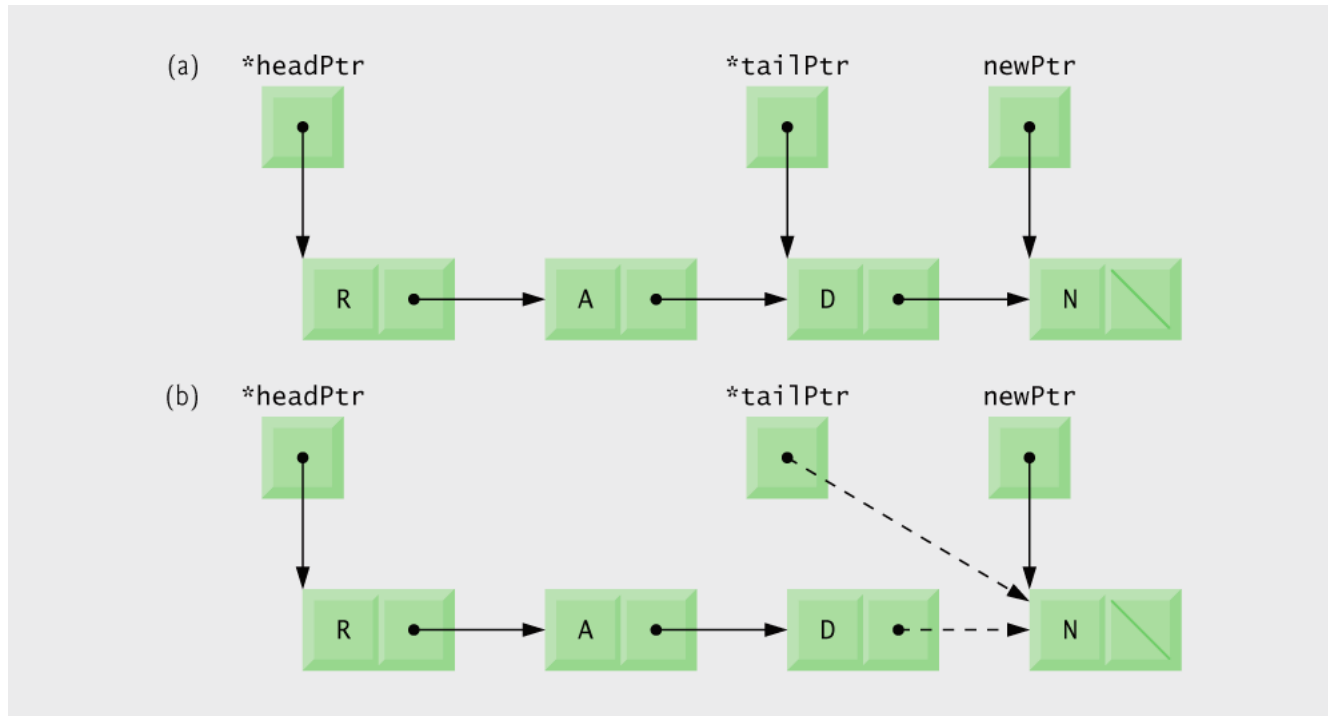
2 to remove an item from the queue

3 to end

? 3

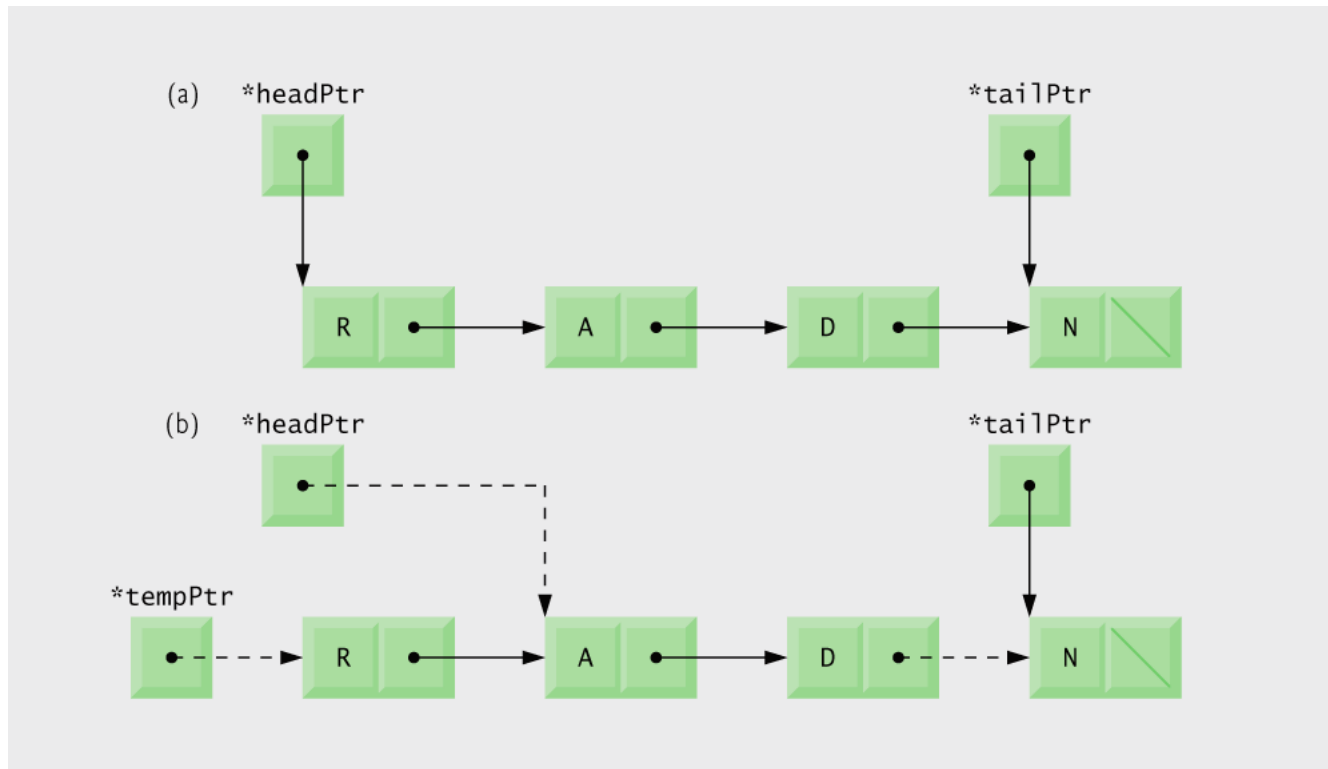
End of run.





**Fig. 12.15** | enqueue operation.





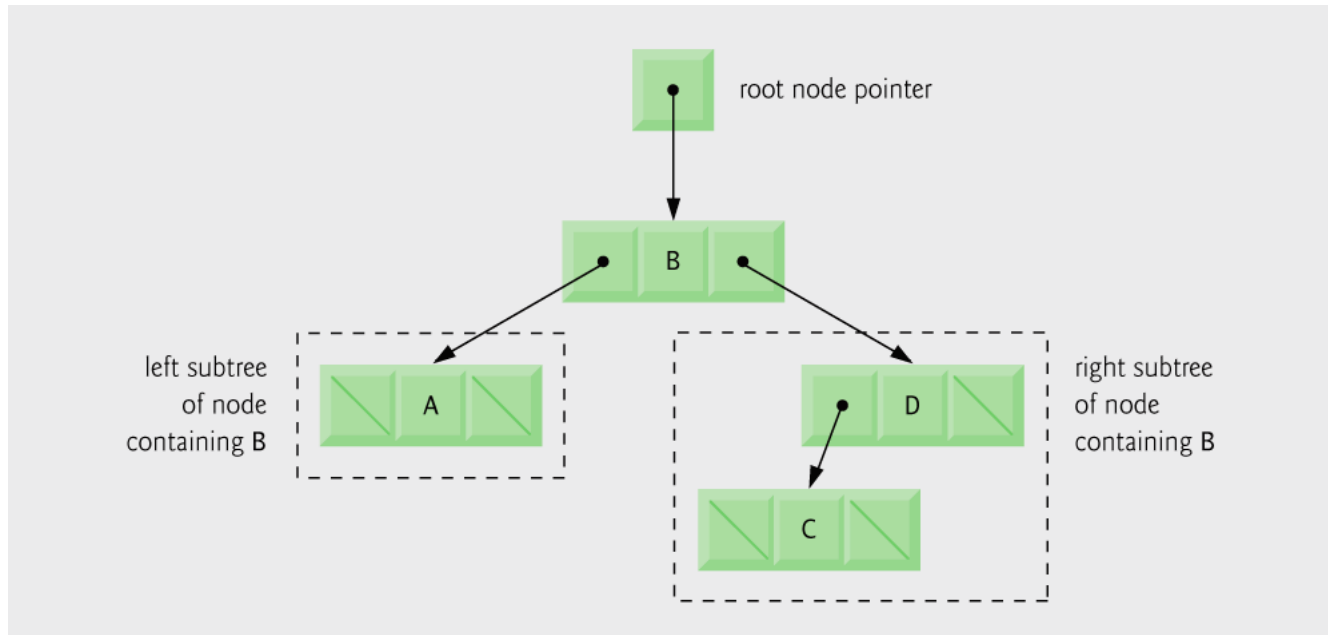
**Fig. 12.16** | dequeue operation.



# 12.7 Trees

- **Tree nodes contain two or more links**
  - All other data structures we have discussed only contain one
- **Binary trees**
  - All nodes 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
  - A node with no children is called a leaf node





**Fig. 12.17** | Binary tree graphical representation.





# Common Programming Error 12.8

---

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

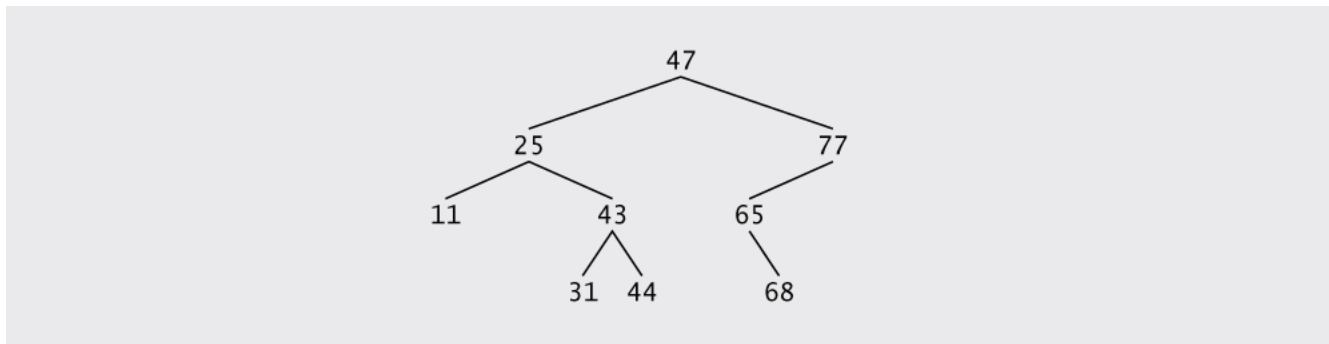


# 12.7 Trees

- **Binary search tree**

- Values in left subtree less than parent
- Values in right subtree greater than parent
- Facilitates duplicate elimination
- Fast searches - for a balanced tree, maximum of  $\log n$  comparisons





**Fig. 12.18** | Binary search tree.



# 12.7 Trees

- **Tree traversals:**
  - **Inorder traversal – prints the node values in ascending order**
    1. **Traverse the left subtree with an inorder traversal**
    2. **Process the value in the node (i.e., print the node value)**
    3. **Traverse the right subtree with an inorder traversal**
  - **Preorder traversal**
    1. **Process the value in the node**
    2. **Traverse the left subtree with a preorder traversal**
    3. **Traverse the right subtree with a preorder traversal**
  - **Postorder traversal**
    1. **Traverse the left subtree with a postorder traversal**
    2. **Traverse the right subtree with a postorder traversal**
    3. **Process the value in the node**



## Outline

fig12\_19.c

(1 of 5)

```

1  /* Fig. 12.19: fig12_19.c
2     Create a binary tree and traverse it
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 }; /* end structure treeNode */
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 /* function main begins program execution */
25 int main( void )
26 {
27     int i; /* counter to loop from 1-10 */
28     int item; /* variable to hold random values */
29     TreeNodePtr rootPtr = NULL; /* tree initially empty */
30

```

Each node in the tree contains a data element and a pointer to the left and right child nodes



## Outline

fig12\_19.c

(2 of 5)

```
31 srand( time( NULL ) );
32 printf( "The numbers being placed in the tree are:\n" );
33
34 /* insert random values between 0 and 14 in the tree */
35 for ( i = 1; i <= 10; i++ ) {
36     item = rand() % 15;
37     printf( "%3d", item );
38     insertNode( &rootPtr, item );
39 } /* end for */
40
41 /* traverse the tree preOrder */
42 printf( "\n\nThe preOrder traversal is:\n" );
43 preOrder( rootPtr );
44
45 /* traverse the tree inOrder */
46 printf( "\n\nThe inOrder traversal is:\n" );
47 inOrder( rootPtr );
48
49 /* traverse the tree postOrder */
50 printf( "\n\nThe postOrder traversal is:\n" );
51 postOrder( rootPtr );
52
53 return 0; /* indicates successful termination */
54
55 } /* end main */
```



## Outline

fig12\_19.c

(3 of 5)

```
56 /* insert node into tree */
57 void insertNode( TreeNodePtr *treePtr, int value )
58 {
59     /* if tree is empty */
60     if ( *treePtr == NULL ) {
61         *treePtr = malloc( sizeof( TreeNode ) );
62
63         /* if memory was allocated then assign data */
64         if ( *treePtr != NULL ) {
65             ( *treePtr )->data = value;
66             ( *treePtr )->leftPtr = NULL;
67             ( *treePtr )->rightPtr = NULL;
68         } /* end if */
69     } else {
70         printf( "%d not inserted. No memory available.\n", value );
71     } /* end else */
72 } /* end if */
73 else { /* tree is not empty */
74
75     /* data to insert is less than data in current node */
76     if ( value < ( *treePtr )->data ) {
77         insertNode( &(amp; ( *treePtr )->leftPtr ), value );
78     } /* end if */
```

To insert a node into the tree, memory must first be allocated for that node

If the inserted node's data is less than the current node's, the program will attempt to insert the node at the current node's left child.



## Outline

fig12\_19.c

(4 of 5)

```

82      /* data to insert is greater than data in current node */
83      else if ( value > ( *treePtr )->data ) {
84          insertNode( &( ( *treePtr )->rightPtr ), value );
85      } /* end else if */
86      else { /* duplicate data value ignored */
87          printf( "dup" );
88      } /* end else */
89
90  } /* end else */
91
92
93 } /* end function insertNode */
94
95 /* begin inorder traversal of tree */
96 void inorder( TreeNodePtr treePtr )
97 {
98
99     /* if tree is not empty then traverse */
100    if ( treePtr != NULL ) {
101        inorder( treePtr->leftPtr );
102        printf( "%3d", treePtr->data );
103        inorder( treePtr->rightPtr );
104    } /* end if */
105
106 } /* end function inorder */
107
108 /* begin preorder traversal of tree */
109 void preOrder( TreeNodePtr treePtr )
110 {
111

```

If the inserted node's data is greater than the current node's, the program will attempt to insert the node at the current node's right child.

The inorder traversal calls an inorder traversal on the node's left child, then prints the node itself, then calls an inorder traversal on the right child.





## Outline

fig12\_19.c

(5 of 5)

```
112 /* if tree is not empty then traverse */
113 if ( treePtr != NULL ) {
114     printf( "%3d", treePtr->data );
115     preOrder( treePtr->leftPtr );
116     preOrder( treePtr->rightPtr );
117 } /* end if */
118
119 } /* end function preOrder */
120
121 /* begin postorder traversal of tree */
122 void postOrder( TreeNodePtr treePtr )
123 {
124
125     /* if tree is not empty then traverse */
126     if ( treePtr != NULL ) {
127         postOrder( treePtr->leftPtr );
128         postOrder( treePtr->rightPtr );
129         printf( "%3d", treePtr->data );
130     } /* end if */
131
132 } /* end function postOrder */
```

The preorder traversal prints the node itself, then calls a preorder traversal on the node's left child, then calls a preorder traversal on the right child.

The postorder traversal calls an postorder traversal on the node's left child, then calls an postorder traversal on the right child, then prints the node itself.



## Outline

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

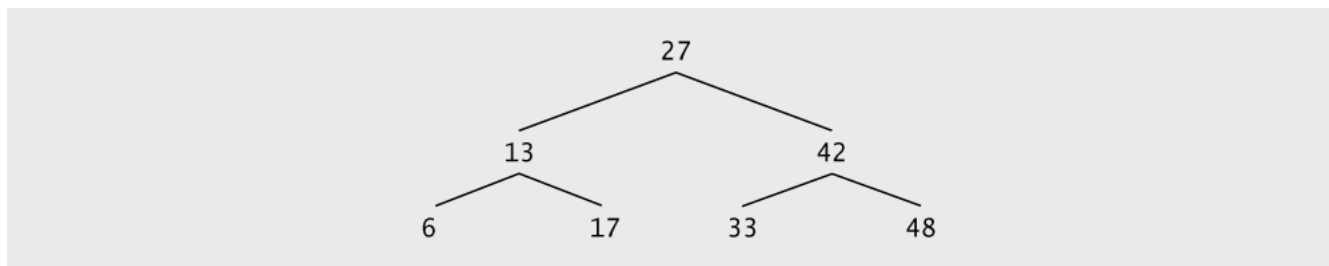
The inOrder traversal is:

2 4 5 6 7 11 12

The postOrder traversal is:

2 5 4 11 12 7 6





**Fig. 12.21** | Binary search tree with seven nodes.

