Chapter 12
C Data Structures

C How to Program, 8/e

12.2 Self-Referential Structures

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

Fig. 12.1 | Self-referential structures linked together.

Dynamic Memory Allocation

- Functions `malloc` and `free`, and operator `sizeof`, are essential to dynamic memory allocation.

- `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
  - stores a pointer to the allocated memory in variable `newPtr`.

12.3 Dynamic Memory Allocation (Cont.)

- 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.

12.4 Linked Lists (Cont.)

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

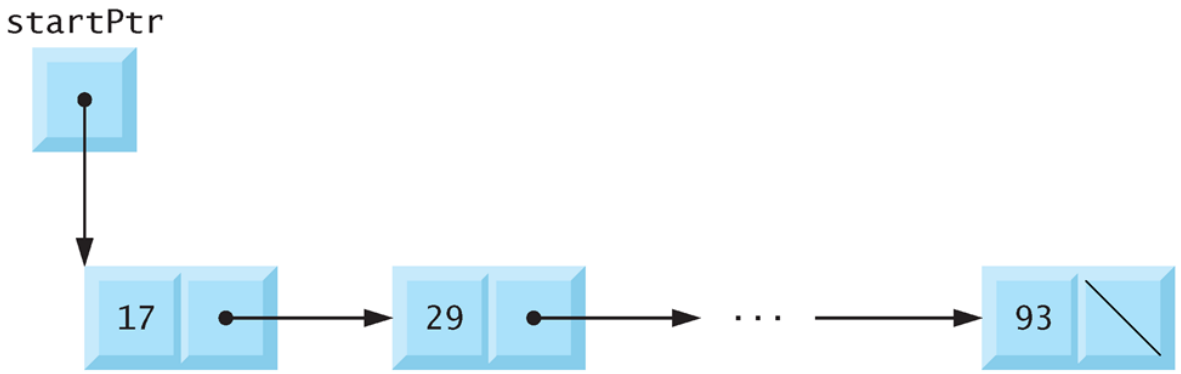
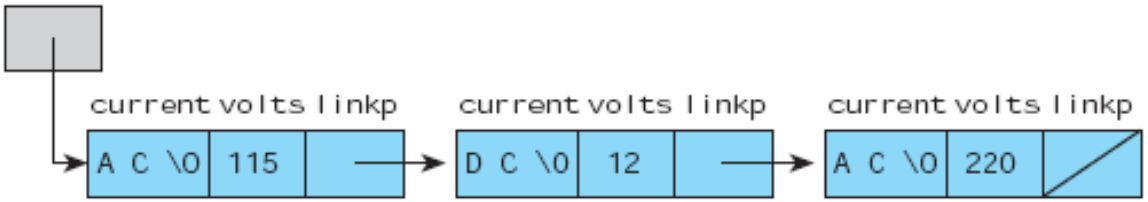


Fig. 12.2 | Linked-list graphical representation.

Linked List Example




```
1.  typedef  struct node_s{
2.          char  current[3];
3.          int  volts;
4.          struct  node_s  *linkp;
5.  } node_t;
```

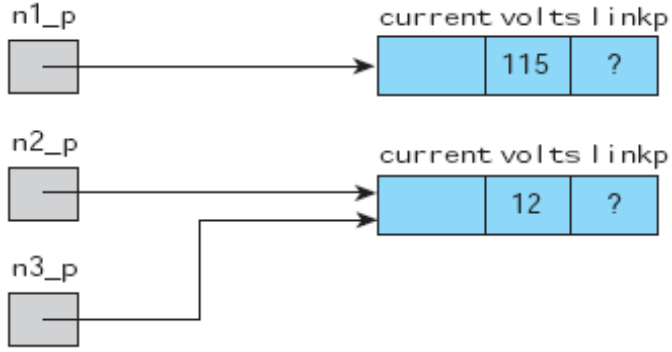
Dynamic Node Allocation

- node_t *n1_p, *n2_p, *n3_p;

- n1_p = (node_t *)malloc(sizeof (node_t));
- strcpy(n1_p->current, "AC");
- n1_p->volts = 115;

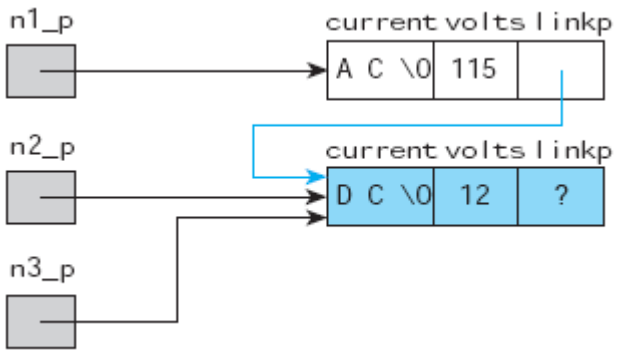
- n2_p = (node_t *)malloc(sizeof (node_t));
- strcpy(n2_p->current, "DC");
- n2_p->volts = 12;

- n3_p = n2_p;



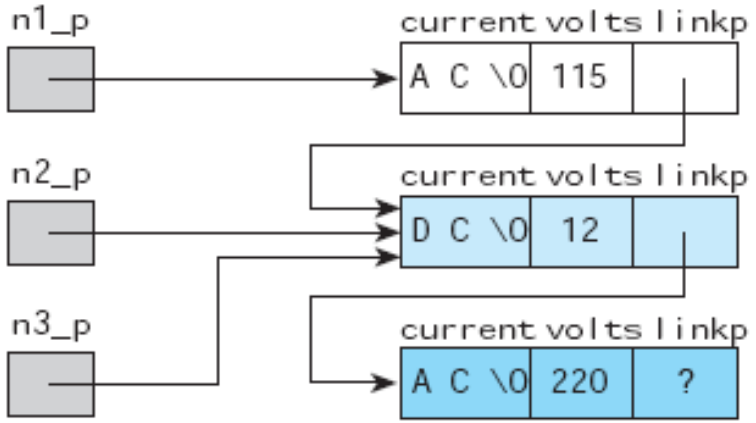
Connecting Nodes

- Pointer assignment
  - n1_p->linkp = n2_p;
- Now, 3 ways to access the 12 in the second node
  - n2_p->volts
  - n3_p->volts
  - n1_p->linkp->volts



Add the Third Node

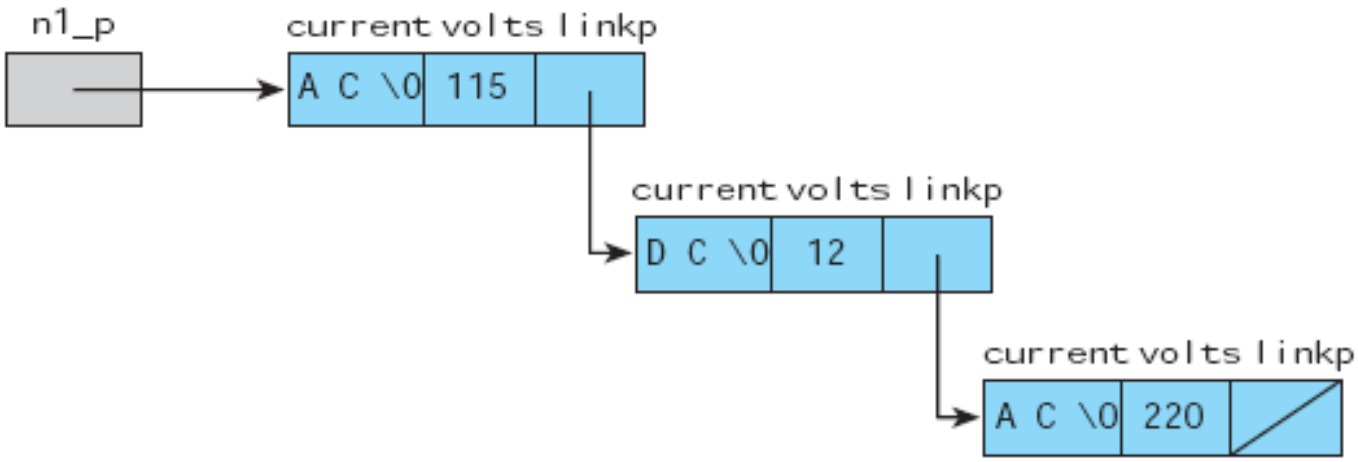
- n2_p->linkp = (node_t *)malloc(sizeof (node_t));
- strcpy(n2_p->linkp->current, "AC");
- n2_p->linkp->volts = 220;



- n2_p->linkp->linkp = NULL;
  - In C, the empty list is represented by the pointer NULL;

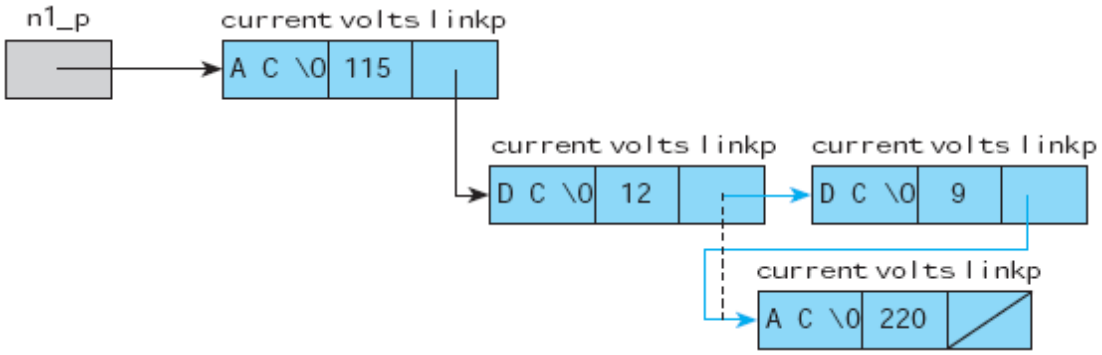
Three-Element Linked List

- List head: n1_p
  - points to the first list element

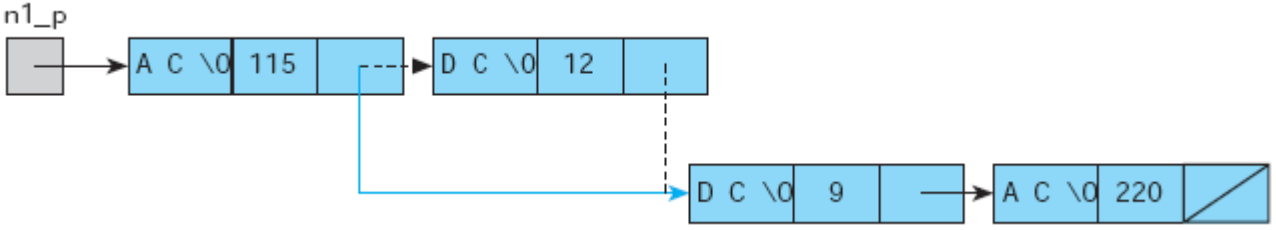


Advantage of Linked Lists

- Can be modified easily
  - Ex: insert a new node containing DC 9



- Easy to delete a list element



```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.)

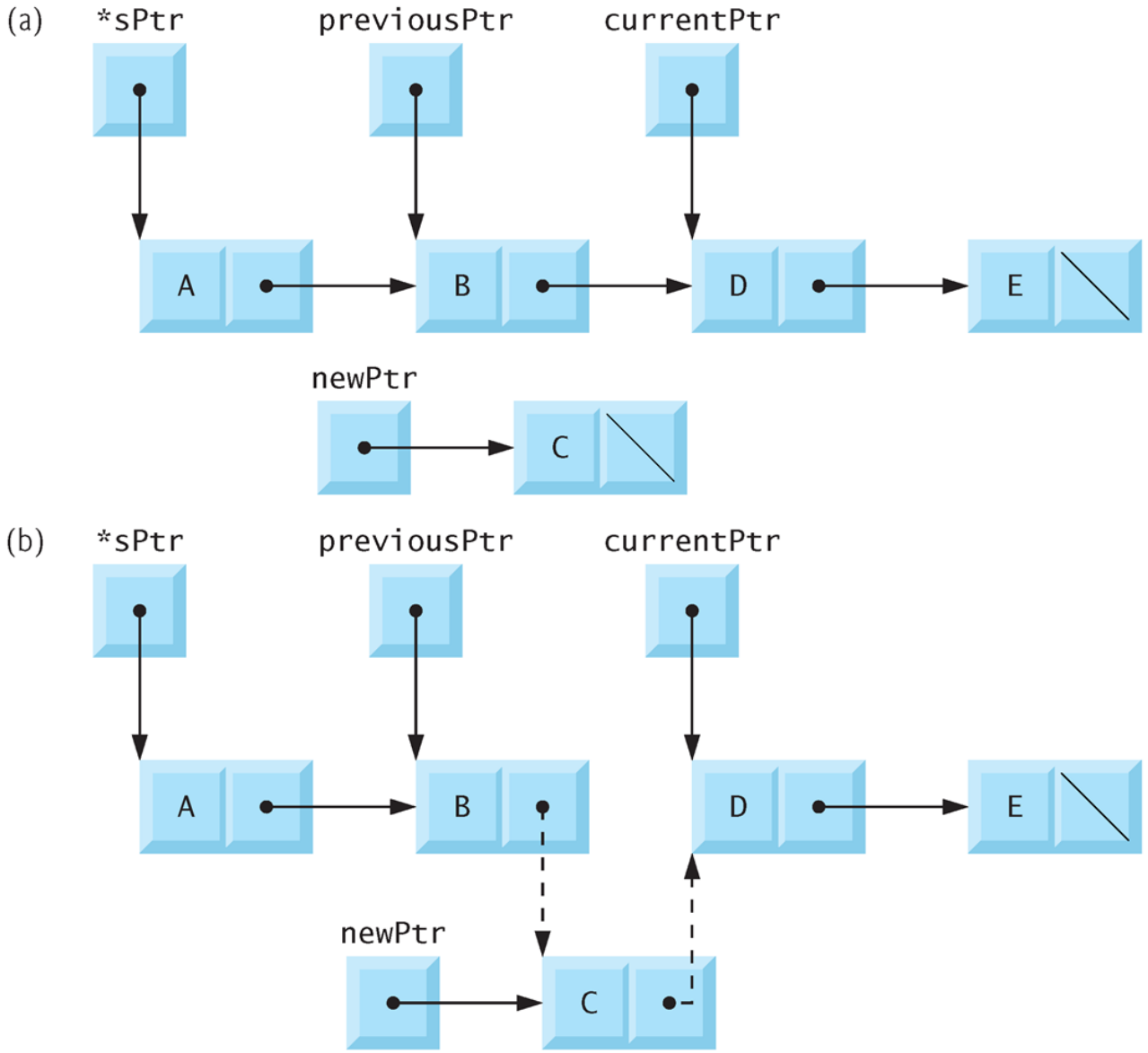


Fig. 12.5 | Inserting a node in order in a list.

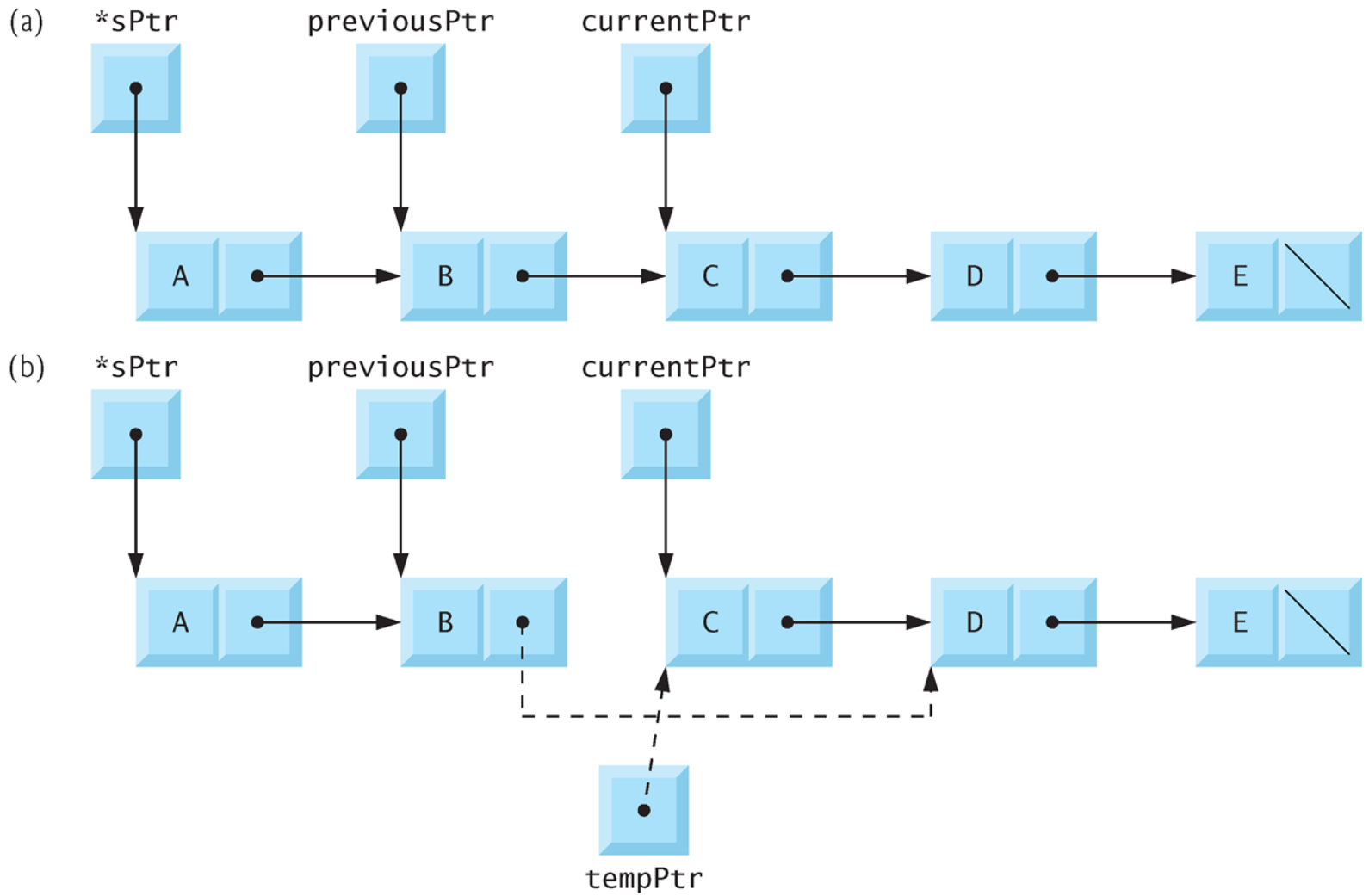


Fig. 12.6 | Deleting a node from a list.

12.5 Stacks

- A stack is 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.

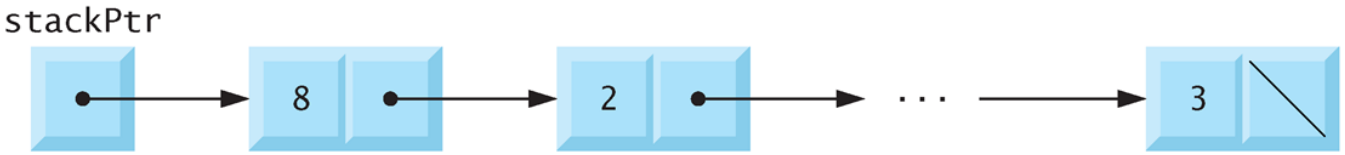


Fig. 12.7 | Stack graphical representation.

```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.)

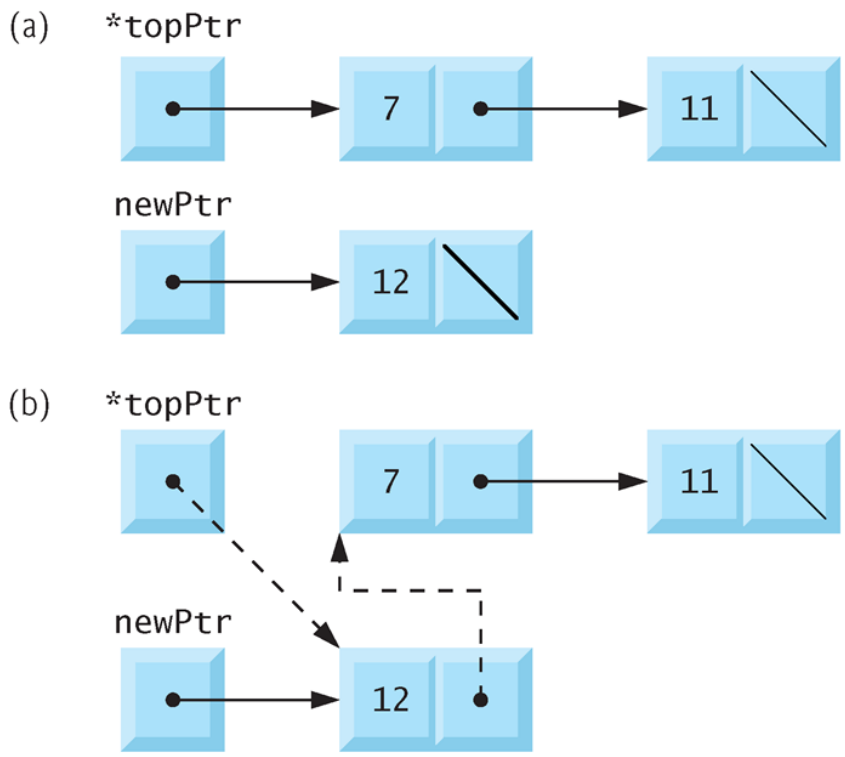


Fig. 12.10 | `push` operation.

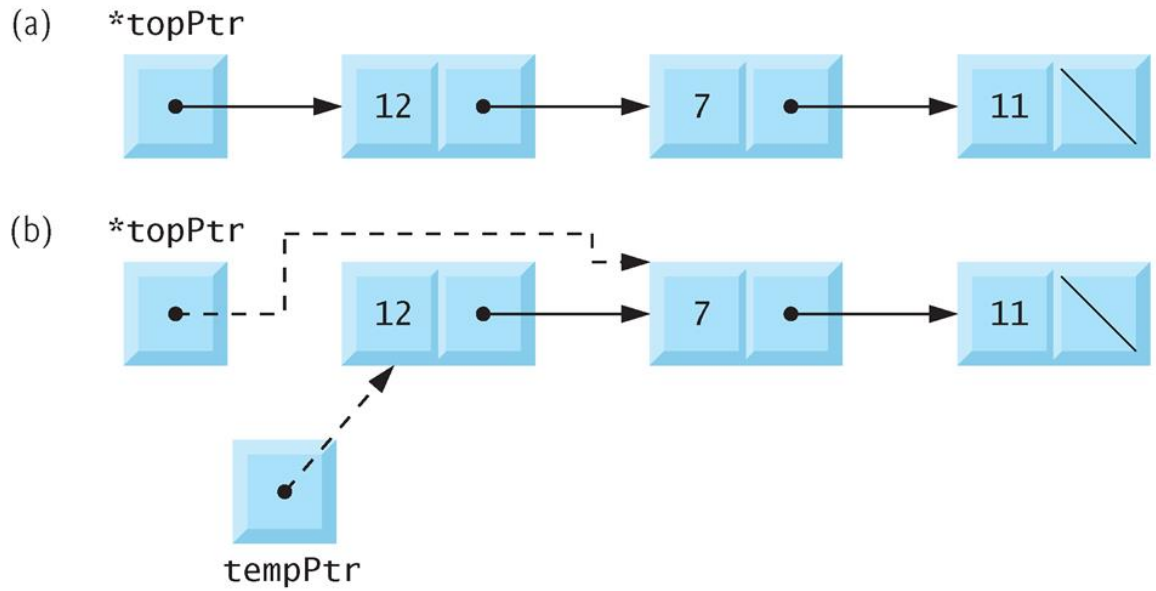


Fig. 12.11 | **pop** operation.

12.5 Stacks (Cont.)

- Stacks have many interesting applications.
  - 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.

- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

12.6 Queues

- Another common data structure is the queue.
- 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`.

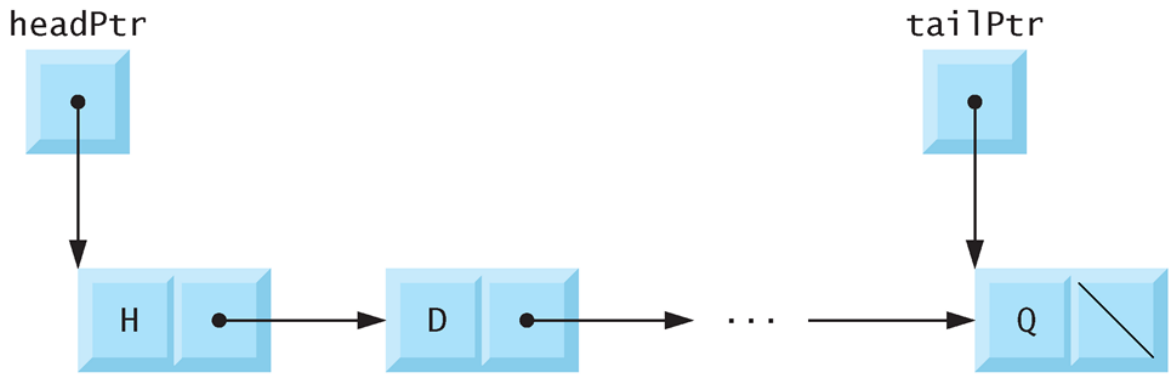


Fig. 12.12 | Queue graphical representation.

```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.)

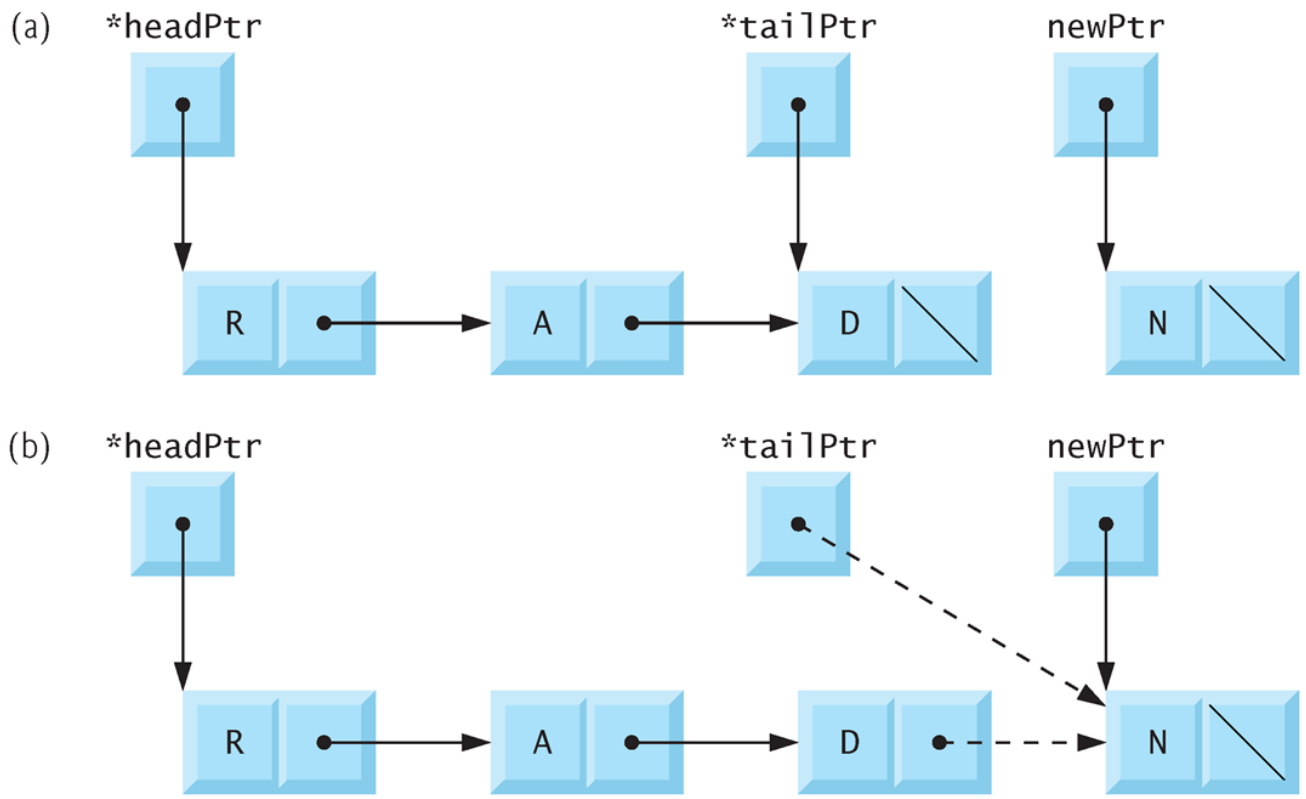


Fig. 12.15 | **enqueue** operation.

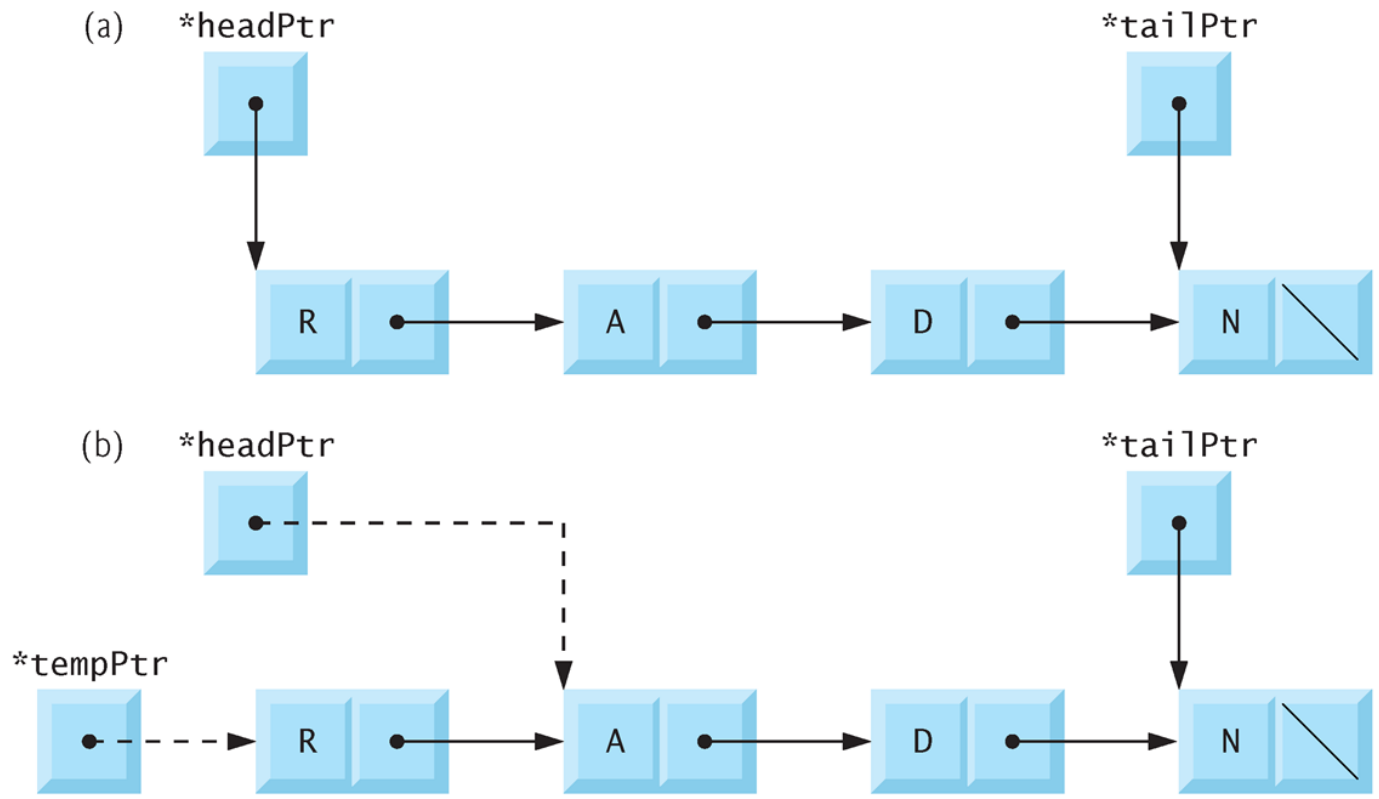


Fig. 12.16 | dequeue operation.

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.

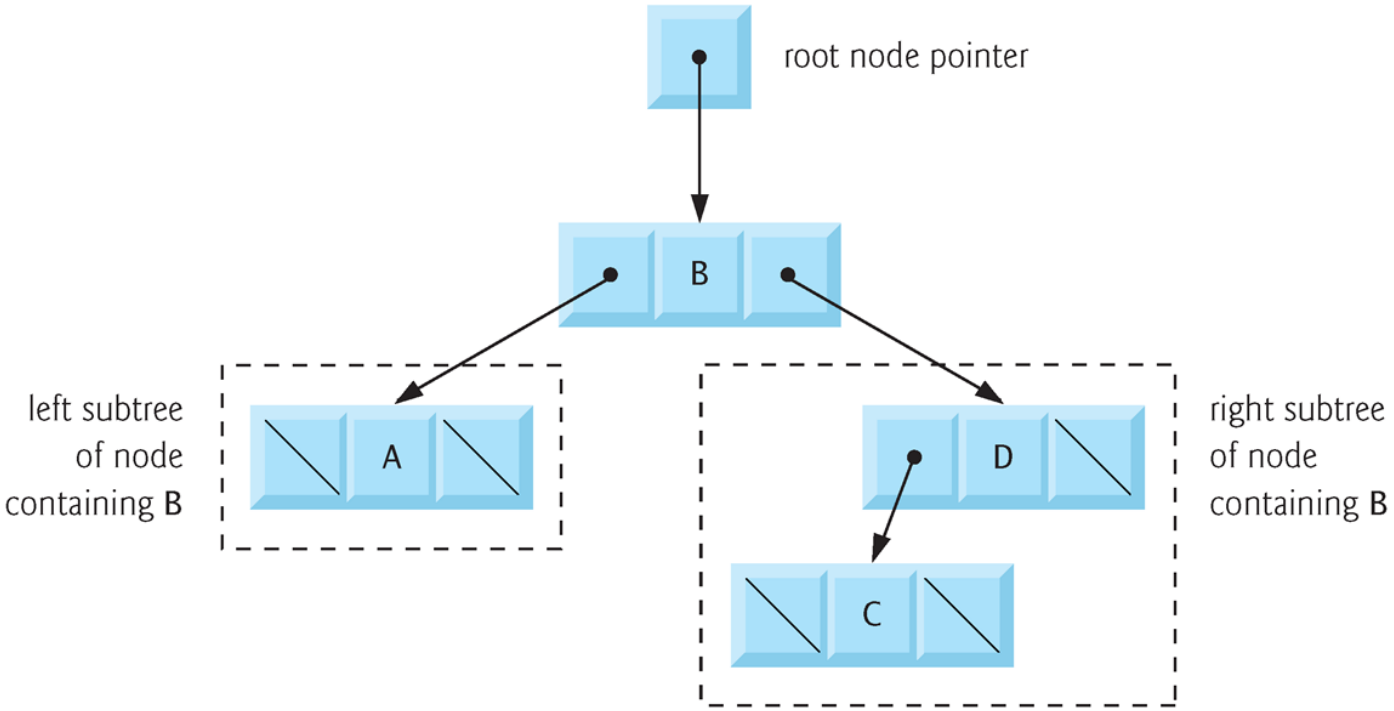


Fig. 12.17 | Binary tree graphical representation.

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.

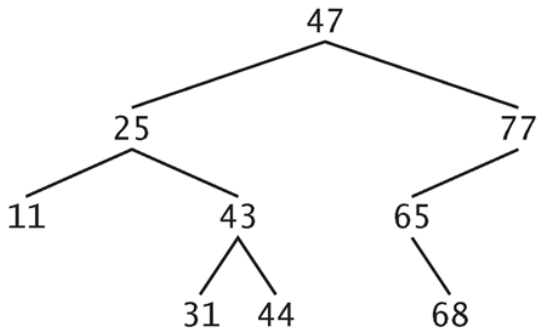


Fig. 12.18 | Binary search tree.

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 Trees (Cont.)

- 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`

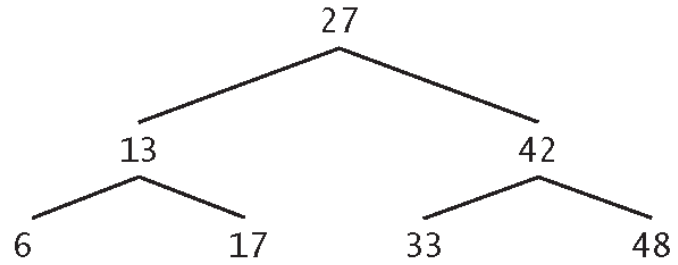


Fig. 12.21 | Binary search tree with seven nodes.

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 the values in the right subtree are processed.

12.7 Trees (Cont.)

- The `preOrder` traversal of the tree in Fig. 12.21 is:
  - `27 13 6 17 42 33 48`

12.7 Trees (Cont.)

- 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 Trees (Cont.)

- 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) 1000-element binary search tree, no more than 10 comparisons need to be made because $2^{10} > 1000$.

12.7 Trees (Cont.)

- 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.