

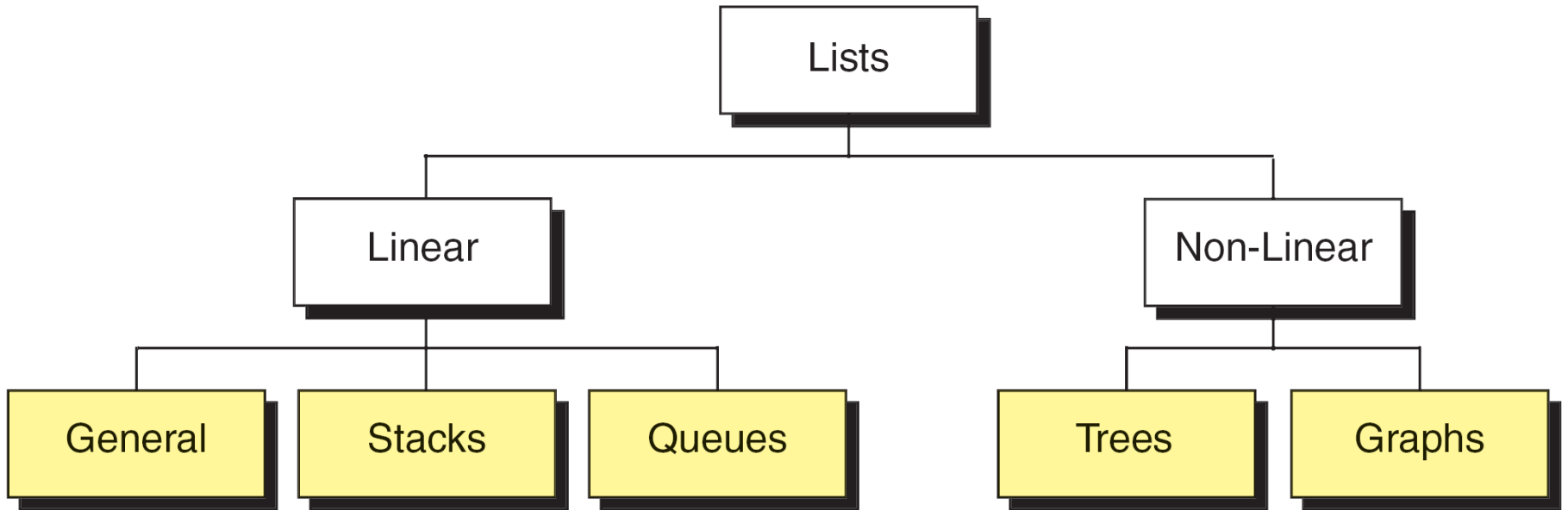
# *Chapter 15*

## *Lists*

### **Objectives**

---

- To introduce the basic concepts of linked lists
- To introduce the basic concepts of stacks
- To introduce the basic concepts of queues
- To introduce the basic concepts of tree structures
- To introduce the basic concepts of graph structures



---

**FIGURE 15-1 Lists**

---

# 15-1 List Implementations

*The C language does not provide any list structures or implementations. When we need them, we must provide the structures and functions for them. Traditionally, two data types, arrays and pointers, are used for their implementation.*

*Topics discussed in this section:*

**Array Implementation**

**Linked List Implementation**

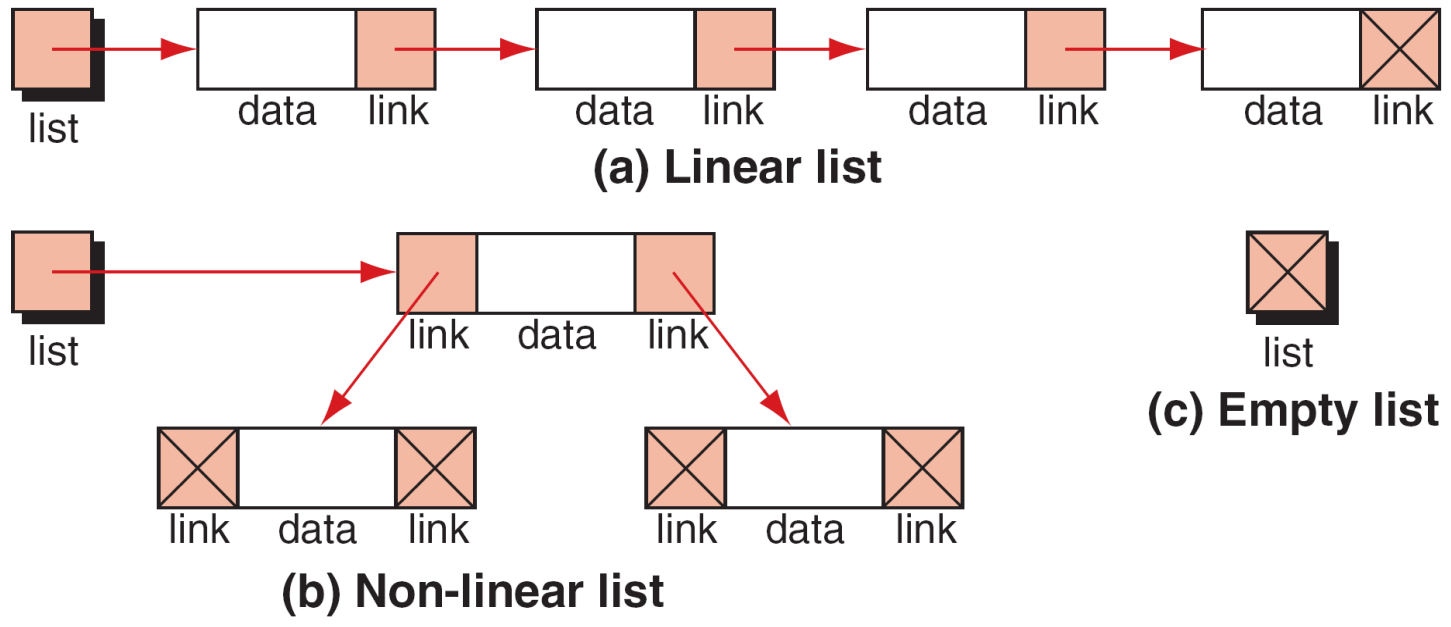
**Pointers to Linked Lists**

# Array Implementation

- How?
- Pros and cons?

# Linked List Implementation

- Linked list
  - An ordered collection of data in which each element contains the location of the next element or elements
- A node contains data and links
- Linear and non-linear structures
- Pros and cons over the array?

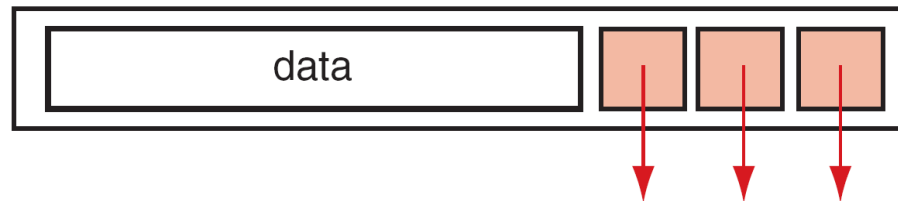


**FIGURE 15-2** Linked Lists

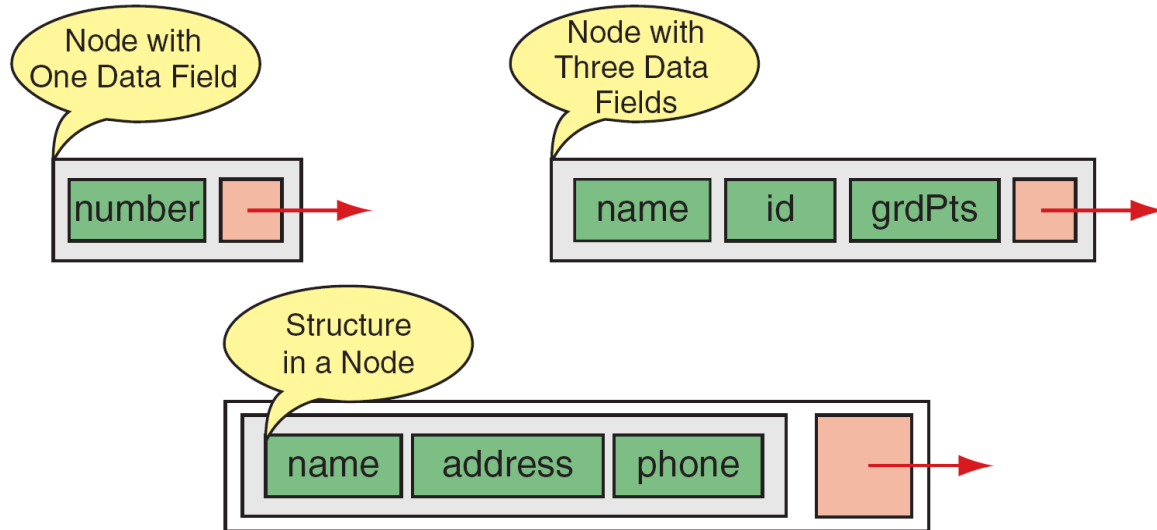
**(a) Node in a linear list**



**(b) Node in a non-linear list**



**FIGURE 15-3** Nodes



**FIGURE 15-4** Linked List Node Structures

# 15-2 General Linear Lists

*A general linear list is a list in which operations, such as retrievals, insertions, changes, and deletions, can be done anywhere in the list, that is, at the beginning, in the middle, or at the end of the list..*

*Topics discussed in this section:*

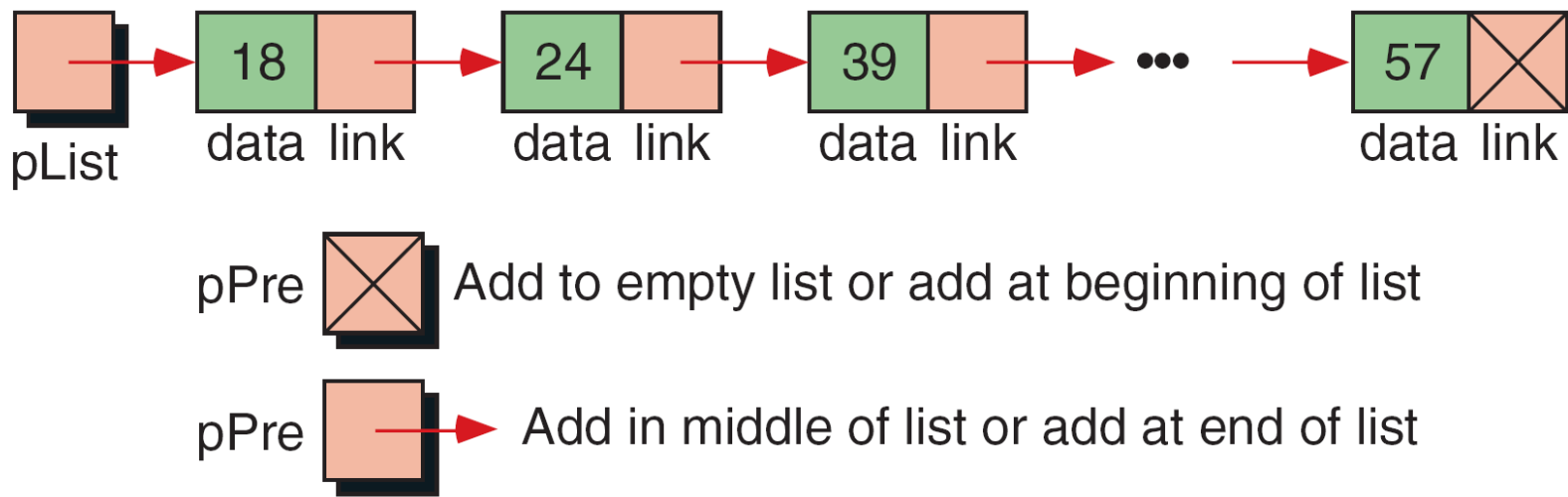
**Insert a Node**

**Delete a Node**

**Locating Data in Linear Lists**

**Traversing Linear Lists**

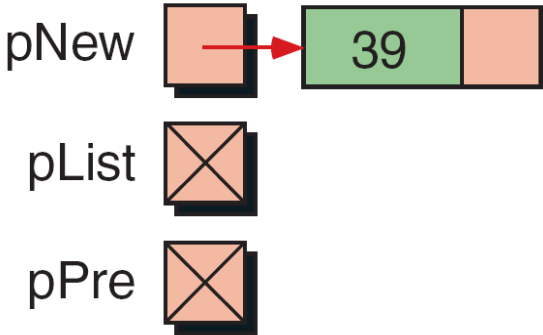
**Building a Linear List**



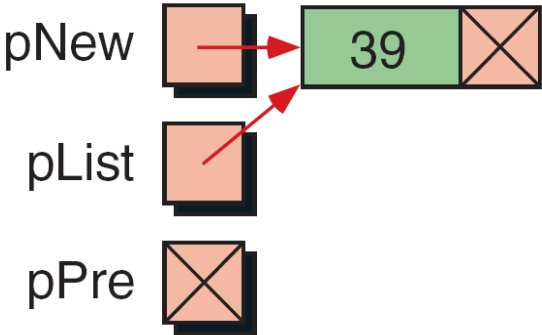
**FIGURE 15-5** Pointer Combinations for Insert

```
pNew->link = pList ;  
pList      = pNew ;
```

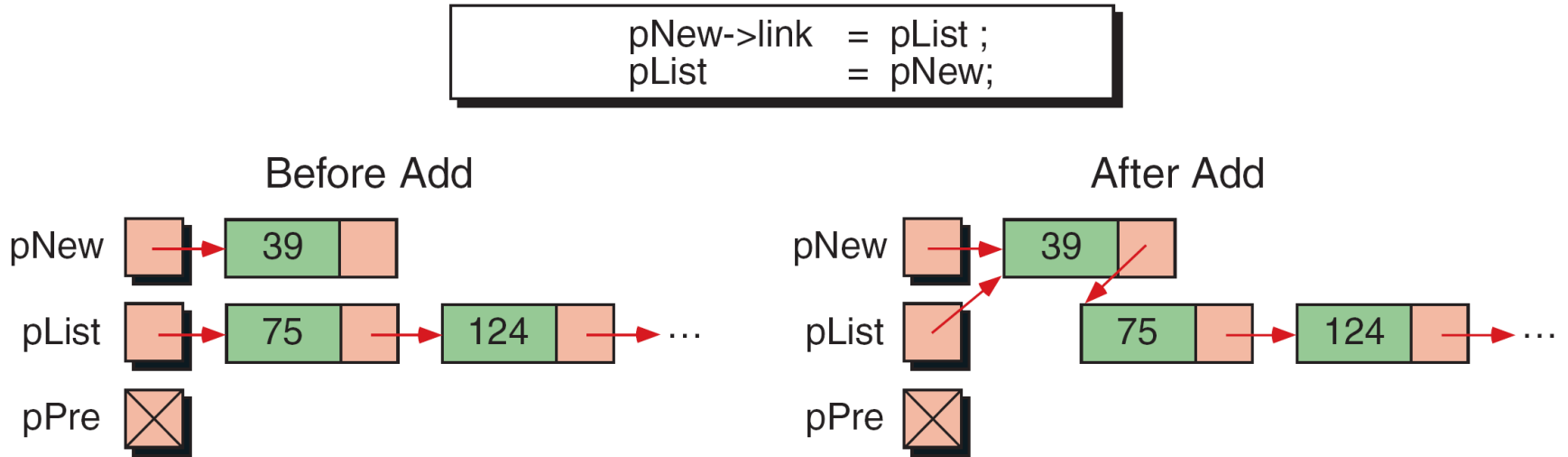
Before Add



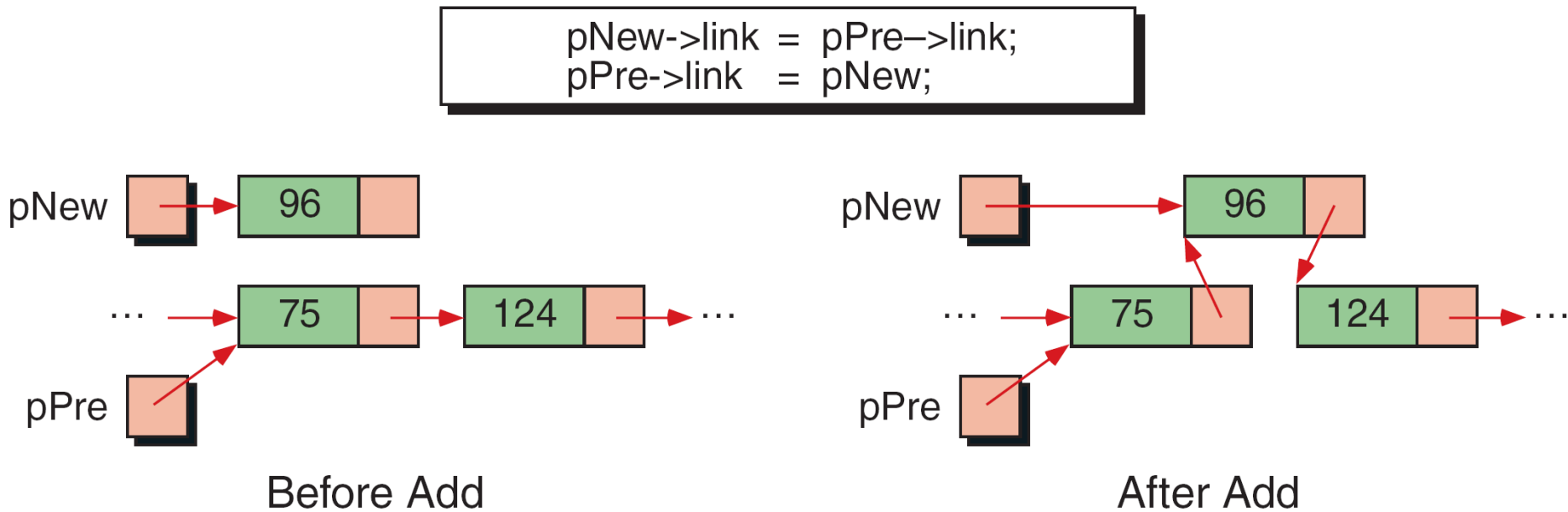
After Add



**FIGURE 15-6** Insert Node to Empty List

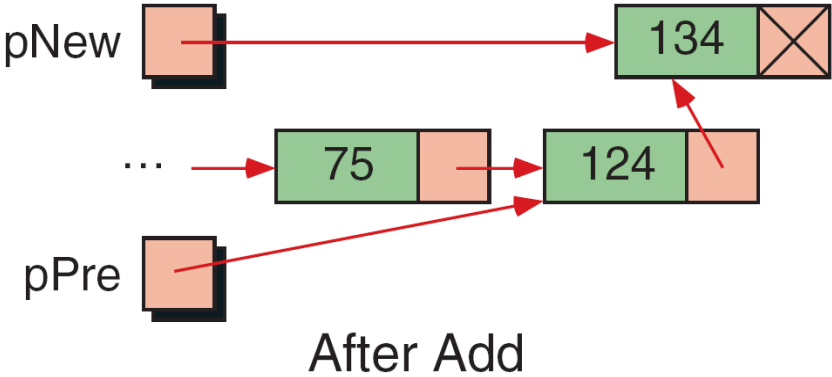
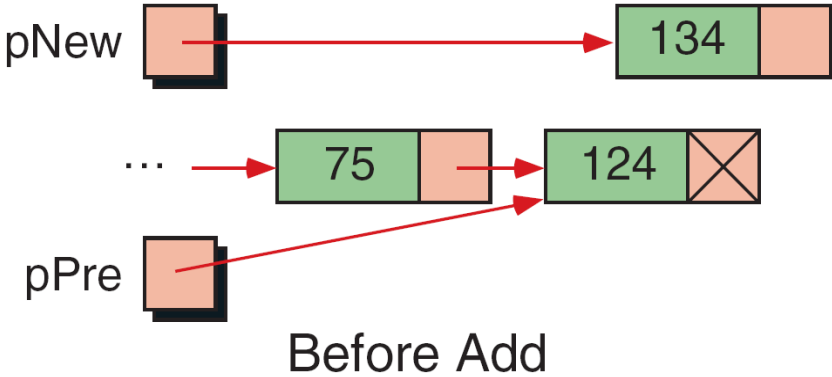


**FIGURE 15-7** Insert Node at Beginning



**FIGURE 15-8** Insert Node in Middle

```
pNew->link = pPre->link;  
pPre->link = pNew;
```



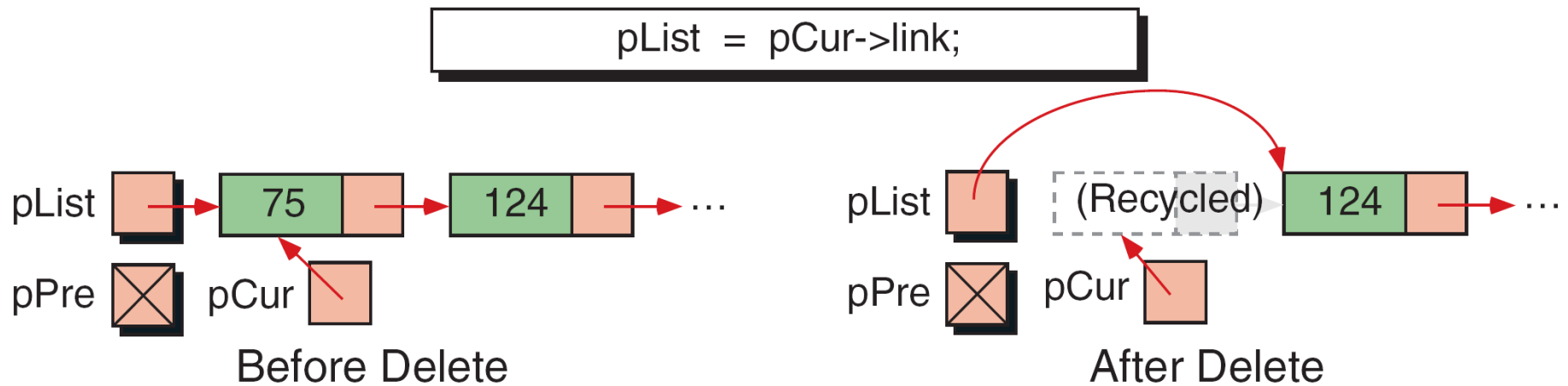
**FIGURE 15-9** Insert Node at End

## PROGRAM 15-1    Insert a Node

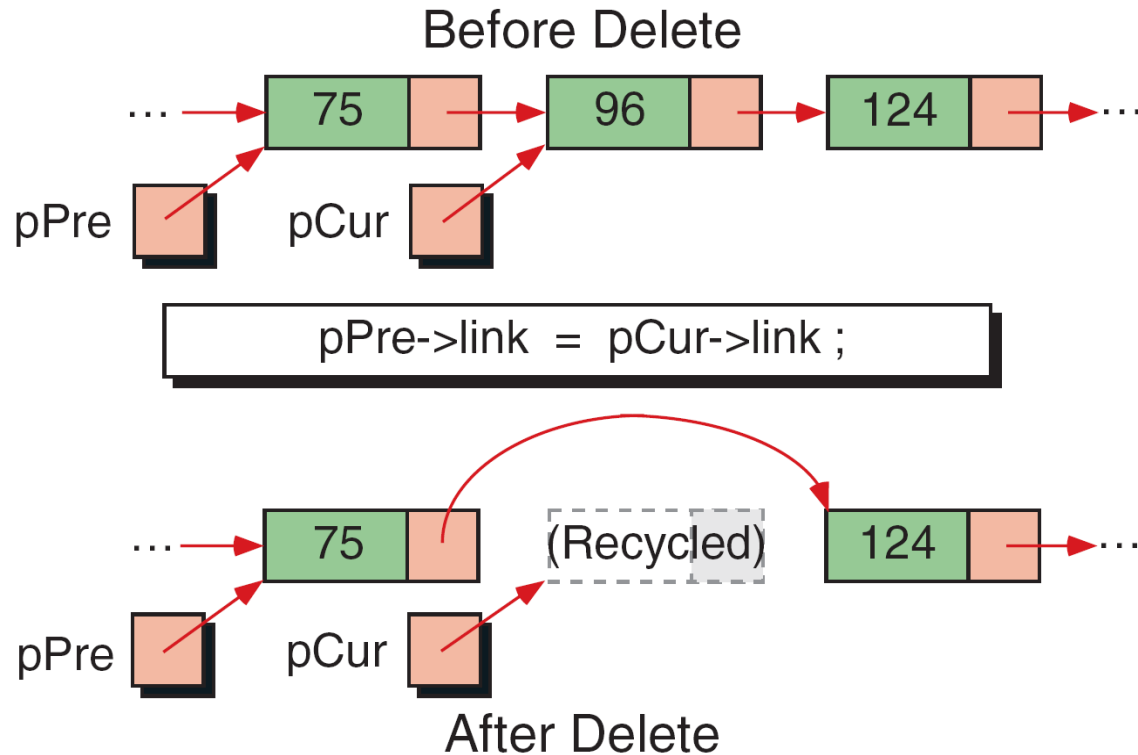
```
1  /* ===== insertNode =====
2     This function inserts a single node into a linear list.
3     Pre   pList is pointer to the list; may be null
4     pPre  points to new node's predecessor
5     item  contains data to be inserted
6     Post  returns the head pointer
7  */
8  NODE* insertNode (NODE* pList, NODE* pPre, DATA item)
9  {
10 // Local Declarations
11     NODE* pNew;
12
13 // Statements
14     if (!(pNew = (NODE*)malloc(sizeof(NODE))))
15         printf("\aMemory overflow in insert\n"),
16             exit (100);
17
```

## PROGRAM 15-1 Insert a Node

```
18     pNew->data = item;
19     if (pPre == NULL)
20     {
21         // Inserting before first node or to empty list
22         pNew->link = pList;
23         pList      = pNew;
24     } // if pPre
25     else
26     {
27         // Inserting in middle or at end
28         pNew->link = pPre->link;
29         pPre->link = pNew;
30     } // else
31     return pList;
32 } // insertNode
```



**FIGURE 15-10** Delete First Node



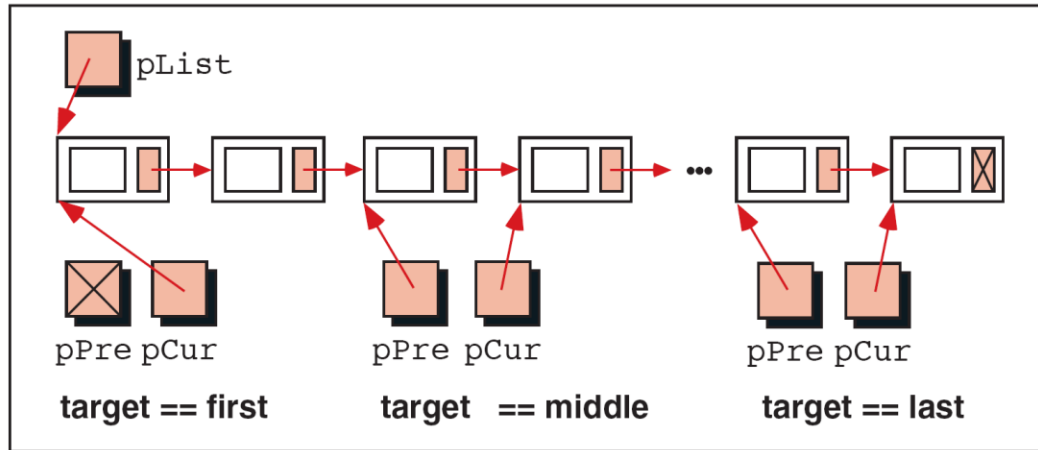
**FIGURE 15-11** Delete—General Case

## PROGRAM 15-2 Delete a Node

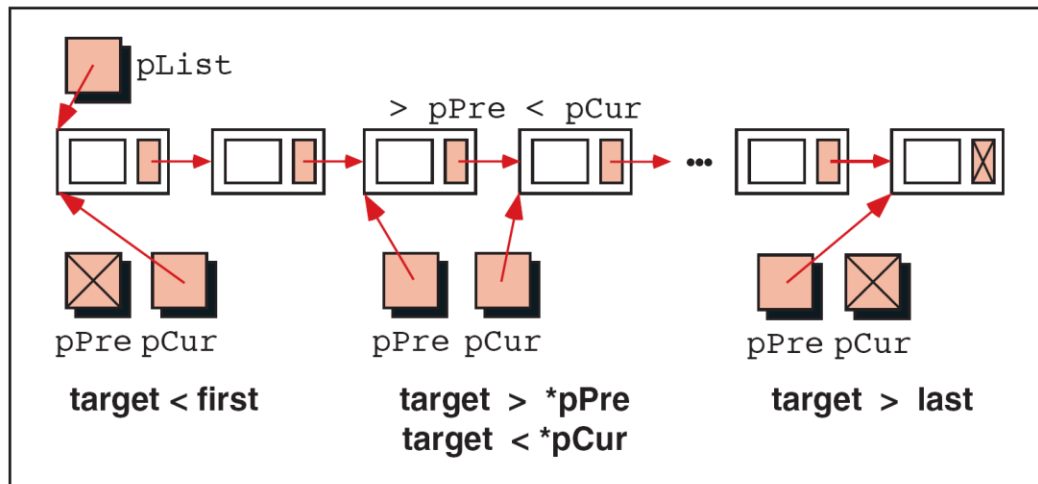
```
1  /* ===== deleteNode =====
2  This function deletes a single node from the link list.
3  Pre    pList is a pointer to the head of the list
4         pPre points to node before the delete node
5         pCur points to the node to be deleted
6  Post  deletes and recycles pCur
7         returns the head pointer
8  */
9  NODE* deleteNode (NODE* pList, NODE* pPre, NODE* pCur)
10 {
11 // Statements
12 if (pPre == NULL)
13     // Deleting first node
14     pList = pCur->link;
15 else
16     // Deleting other nodes
17     pPre->link = pCur->link;
18 free (pCur);
19 return pList;
20 } // deleteNode
```

Condition	pPre	pCur	Return
target < first node	<i>NULL</i>	first node	0
target == first node	<i>NULL</i>	first node	1
first < target < last	largest node < target	first node > target	0
target == middle node	node's predecessor	equal node	1
target == last node	last's predecessor	last node	1
target > last node	last node	<i>NULL</i>	0

**Table 15-1** Linear List Search Results



Successful Searches (Return *true*)



Unsuccessful Searches (Return *false*)

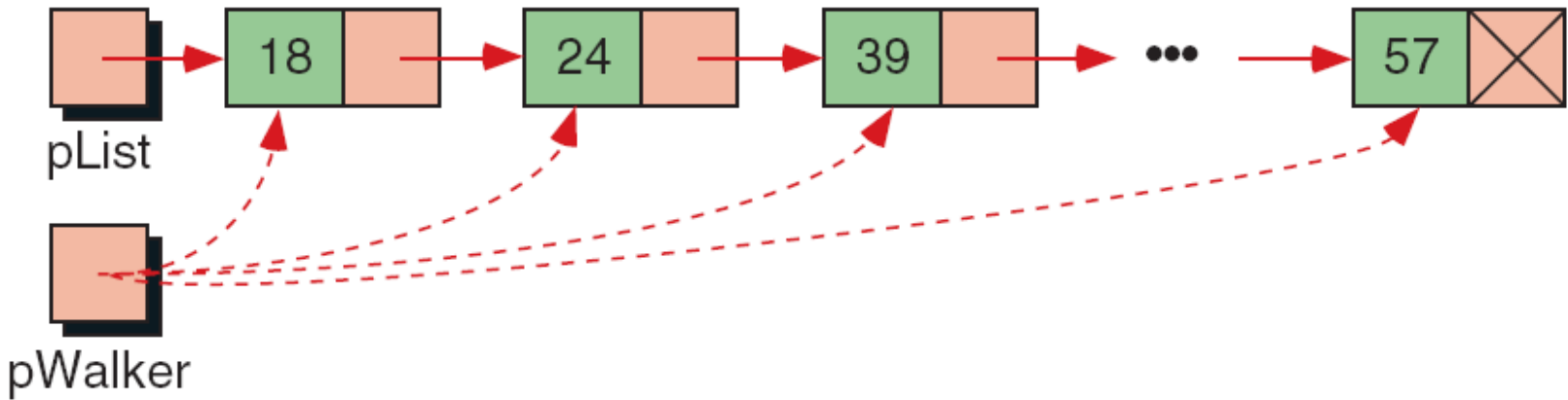
**FIGURE 15-12** Search Results

## PROGRAM 15-3 Search Linear List

```
1  /* ===== searchList =====
2  Given key value, finds the location of a node
3     Pre   pList points to a head node
4     pPre points to variable to receive pred
5     pCur points to variable for current node
6     target is key being sought
7     Post  pCur points to first node with >= key
8           -or- null if target > key of last node
9           pPre points to largest node < key
10          -or- null if target < key of first node
11          function returns true if found
12          false if not found
13 */
14 bool searchList (NODE*  pList, NODE**  pPre,
15                 NODE**  pCur,  KEY_TYPE target)
16 {
17 // Local Declarations
18     bool found = false;
19
```

## PROGRAM 15-3 Search Linear List

```
20 // Statements
21 *pPre = NULL;
22 *pCur = pList;
23
24 // start the search from beginning
25 while (*pCur != NULL && target > (*pCur)->data.key)
26     {
27         *pPre = *pCur;
28         *pCur = (*pCur)->link;
29     } // while
30
31 if (*pCur && target == (*pCur)->data.key)
32     found = true;
33 return found;
34 } // searchList
```



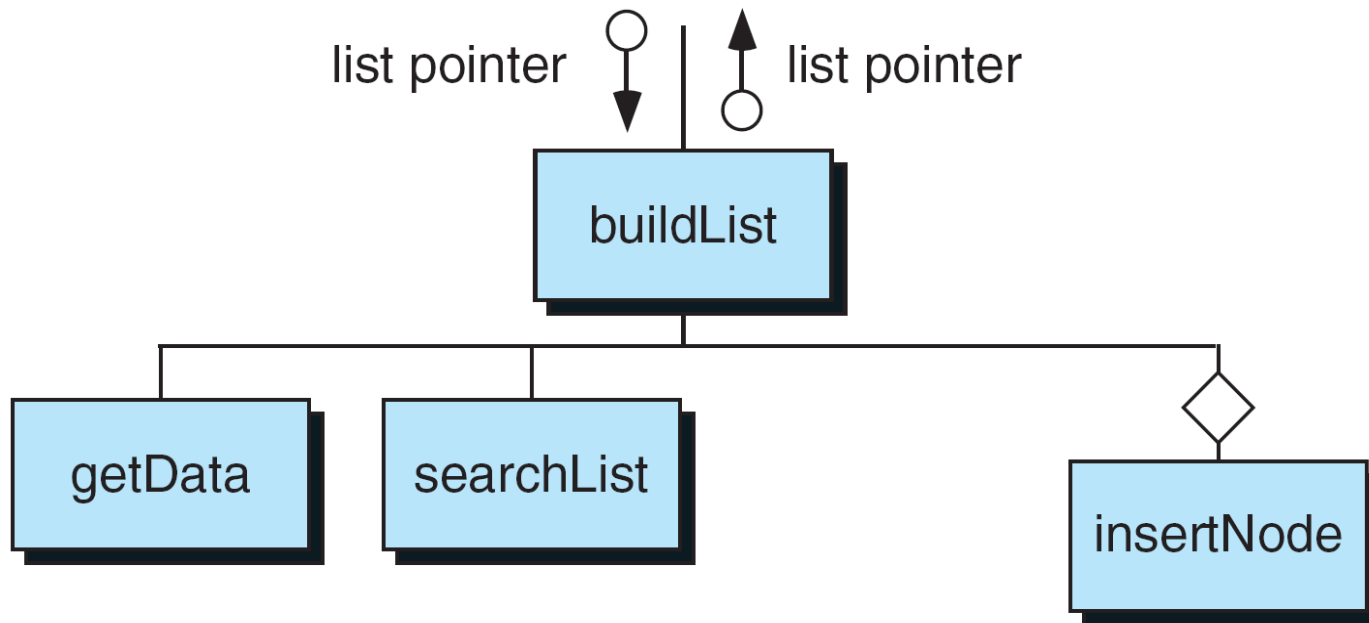
**FIGURE 15-13** Linear List Traversal

## PROGRAM 15-4 Print Linear List

```
1  /* Traverse and print a linear list.
2     Pre   pList is a valid linear list
3     Post  List has been printed
4  */
5  void printList (NODE* pList)
6  {
7  // Local Declarations
8     NODE* pWalker;
9
10 // Statements
11    pWalker = pList;
12    printf("List contains:\n");
13
14    while (pWalker)
15    {
16        printf("%3d ", pWalker->data.key);
17        pWalker = pWalker->link;
18    } // while
19    printf("\n");
20    return;
21 } // printList
```

## PROGRAM 15-5 Average Linear List

```
1  /* This function averages the values in a linear list.
2     Pre   pList is a pointer to a linear list
3     Post  list average is returned
4  */
5  double averageList (NODE* pList)
6  {
7  // Local Declarations
8     NODE* pWalker;
9     int   total;
10    int   count;
11
12 // Statements
13    total = count = 0;
14    pWalker = pList;
15    while (pWalker)
16        {
17        total += pWalker->data.key;
18        count++;
19        pWalker = pWalker->link;
20        } // while
21    return (double)total / count;
22 } // averageList
```



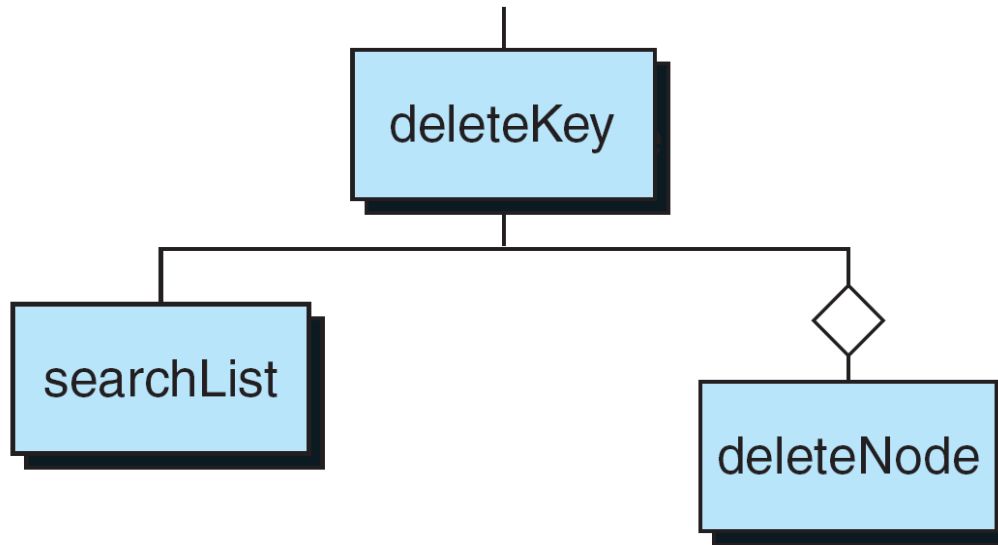
**FIGURE 15-14** Design for Inserting a Node in a List

## PROGRAM 15-6    Build List

```
1  /* ===== buildList =====
2  This program builds a key-sequenced linear list.
3     Pre   fileID is file that contains data for list
4     Post  list built
5           returns pointer to head of list
6  */
7  NODE* buildList (char* fileID)
8  {
9  // Local Declarations
10     DATA  data;
11     NODE*  pList;
12     NODE*  pPre;
13     NODE*  pCur;
14     FILE*  fpData;
15
16 // Statements
17     pList = NULL;
18
```

## PROGRAM 15-6 Build List

```
19     fpData = fopen(fileID, "r");
20     if (!fpData)
21     {
22         printf("Error opening file %s\n", fileID);
23         exit (210);
24     } // if open fail
25
26     while (getData (fpData, &data))
27     {
28         // Determine insert position
29         searchList (pList, &pPre, &pCur, data.key);
30         pList = insertNode(pList, pPre, data);
31     } // while
32     return pList;
33 } // buildList
```



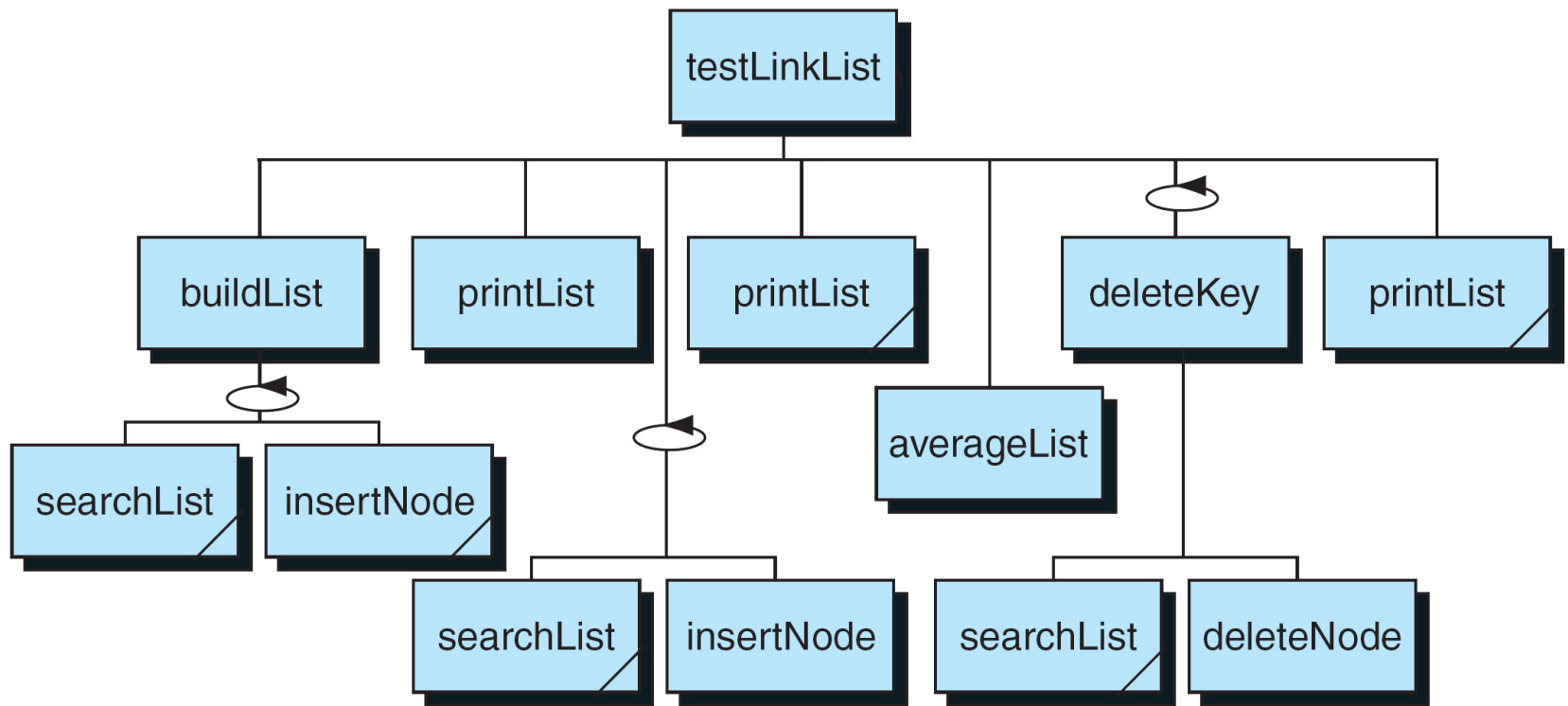
**FIGURE 15-15** Design for Remove Node

## PROGRAM 15-7 Delete Key

```
1  /* ===== deleteKey =====
2  Delete node from a linear list.
3  Pre   list is a pointer to the head of the list
4  Post  node has been deleted
5        -or- a warning message printed if not found
6        returns pointer to first node (pList)
7  */
8  NODE* deleteKey (NODE* pList)
9  {
10 // Local Declarations
11     int    key;
12     NODE*  pHead;
13     NODE*  pCur;
14     NODE*  pPre;
15
16 // Statements
17     printf("Enter key of node to be deleted: ");
```

## PROGRAM 15-7 Delete Key

```
18     scanf ("%d", &key);
19
20     if (!searchList(pList, &pPre, &pCur, key))
21         printf("%d is an invalid key\n", key);
22     else
23         pHead = deleteNode (pList, pPre, pCur);
24
25     return pHead;
26 } // deleteKey
```



**FIGURE 15-16** Link List Test Driver

## PROGRAM 15-8 Test Driver for Link List

```
1  /* Test driver for list functions.
2     Written by:
3     Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <stdbool.h>
8
9  // Global Declarations
10 typedef int KEY_TYPE;
11 typedef struct
12     {
13     KEY_TYPE key;
14     } DATA;
15 typedef struct nodeTag
16     {
17     DATA          data;
18     struct nodeTag* link;
19     } NODE;
20
```

## PROGRAM 15-8 Test Driver for Link List

```
21 // Function Declarations
22 NODE* insertNode (NODE* pList, NODE* pPre, DATA item);
23 NODE* deleteNode (NODE* List, NODE* pPre, NODE* pCur);
24 bool searchList (NODE* pList, NODE** pPre,
25                 NODE** pCur, KEY_TYPE target);
26 void printList (NODE* pList);
27 NODE* buildList (char* fileID);
28 NODE* deleteKey (NODE* pList);
29 bool getData (FILE* fpData, DATA* pData);
30
31 double averageList (NODE* pList);
32
33 int main (void)
34 {
35 // Local Declarations
36     NODE* pList;
37     NODE* pPre;
38     NODE* pCur;
39     DATA data;
40     double avrg;
41     char more;
42
43 // Statements
44     printf("Begin list test driver\n\n");
```

## PROGRAM 15-8 Test Driver for Link List

```
45
46 // Build List
47 pList = buildList("P15-LIST.DAT");
48 if (!pList)
49     {
50         printf("Error building chron file\a\n");
51         exit (100);
52     } // if
53 printList (pList);
54
55 printf("\nInsert data tests.\n");
56 printf("Enter key:                ");
57 scanf ("%d", &data.key);
58 do
59     {
60         if (searchList (pList, &pPre, &pCur, data.key))
61             printf("Key already in list. Not inserted\n");
62         else
63             pList = insertNode(pList, pPre, data);
64         printf("Enter key <-1> to stop: ");
65         scanf ("%d", &data.key);
66     } while (data.key != -1);
```

## PROGRAM 15-8 Test Driver for Link List

```
67     printList (pList);
68
69     avrg = averageList(pList);
70     printf("\nData average: %.1f\n", avrg);
71
72     printf("\nDelete data tests.\n");
73     do
74     {
75         pList = deleteKey (pList);
76         printf("Delete another <Y/N>: ");
77         scanf (" %c", &more);
78     } while (more == 'Y' || more == 'y');
79
80     printList (pList);
```

## PROGRAM 15-8 Test Driver for Link List

```
main_list_test_driver
```

```
List contains:
```

```
111 222 333 444 555 666 777
```

```
Insert data tests.
```

```
Enter key: 50
```

```
Enter key <-1> to stop: -1
```

```
List contains:
```

```
50 111 222 333 444 555 666 777
```

```
Data average: 394.8
```

```
Delete data tests.
```

```
Enter key of node to be deleted: 50
```

```
Delete another <Y/N>: n
```

```
List contains:
```

```
111 222 333 444 555 666 777
```

```
Tests complete.
```

# 15-3 Stacks

*A stack is a linear list in which all additions and deletions are restricted to one end, called the top. Stacks are known as the last in–first out (LIFO) data structure.*

*Topics discussed in this section:*

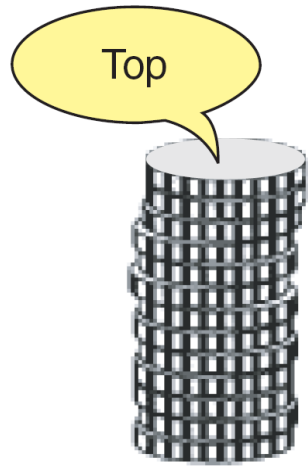
**Stack Structures**

**Stack Algorithms**

**Stack Demonstration**

## *Note*

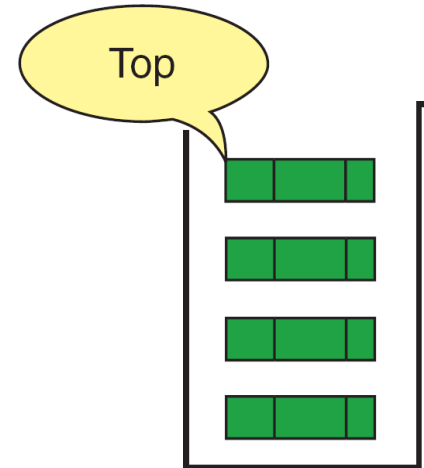
**A stack is a last in–first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.**



Stack of Coins



Stack of Books

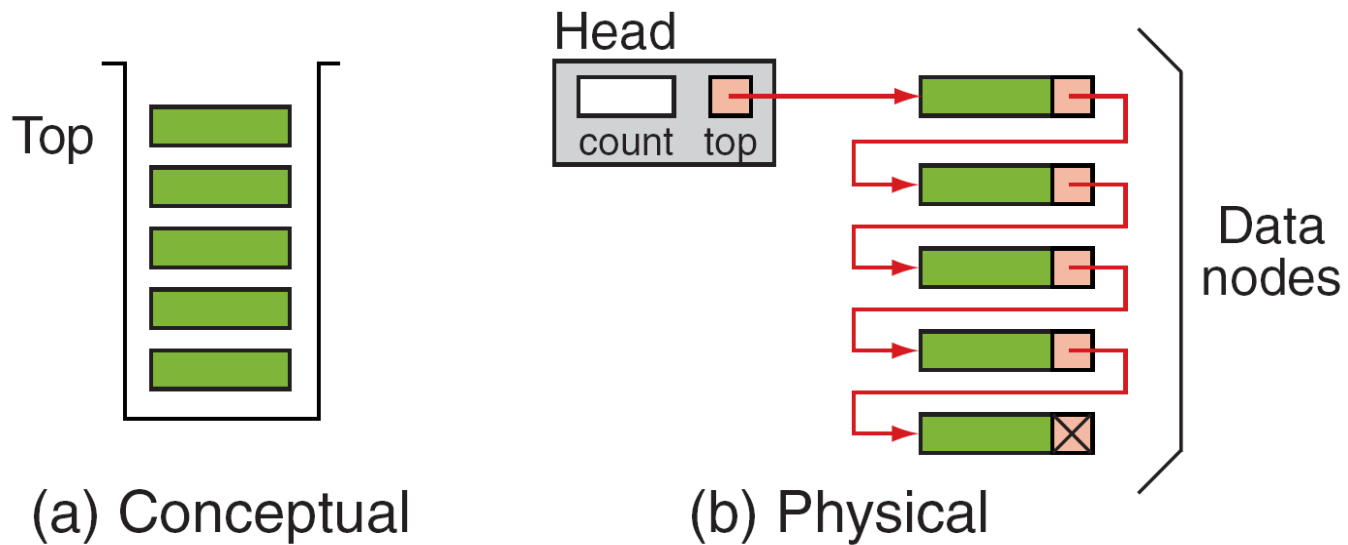


Computer Stack

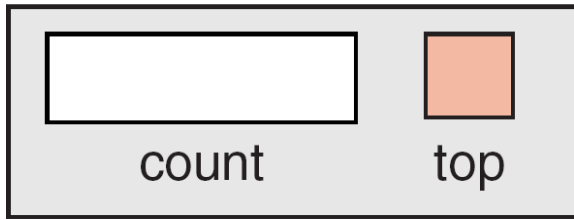
---

**FIGURE 15-17** Stack

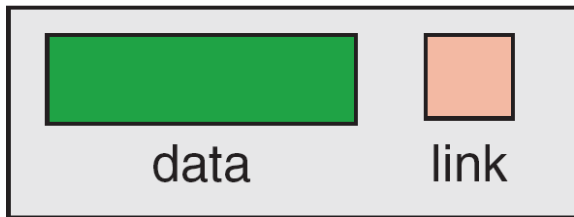
---



**FIGURE 15-18** Conceptual and Physical Stack Implementations



Stack Head Structure

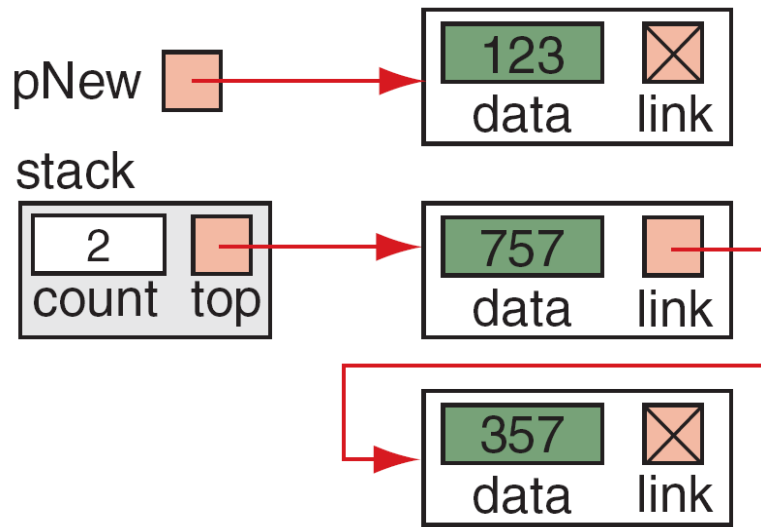


Stack Node Structure

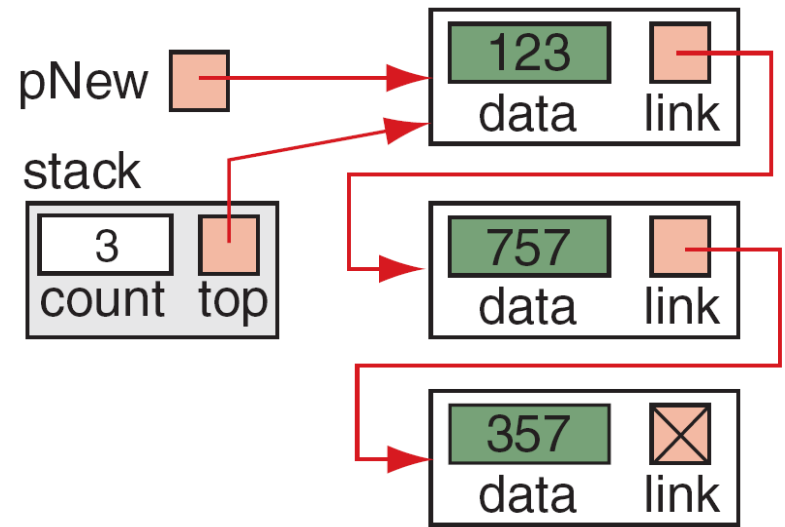
```
typedef struct
{
    int          count;
    struct node* top;
} STACK;

typedef struct node
{
    int          data;
    struct node* link;
} STACK_NODE;
```

**FIGURE 15-19** Stack Data Structure



(a) Before



(b) After

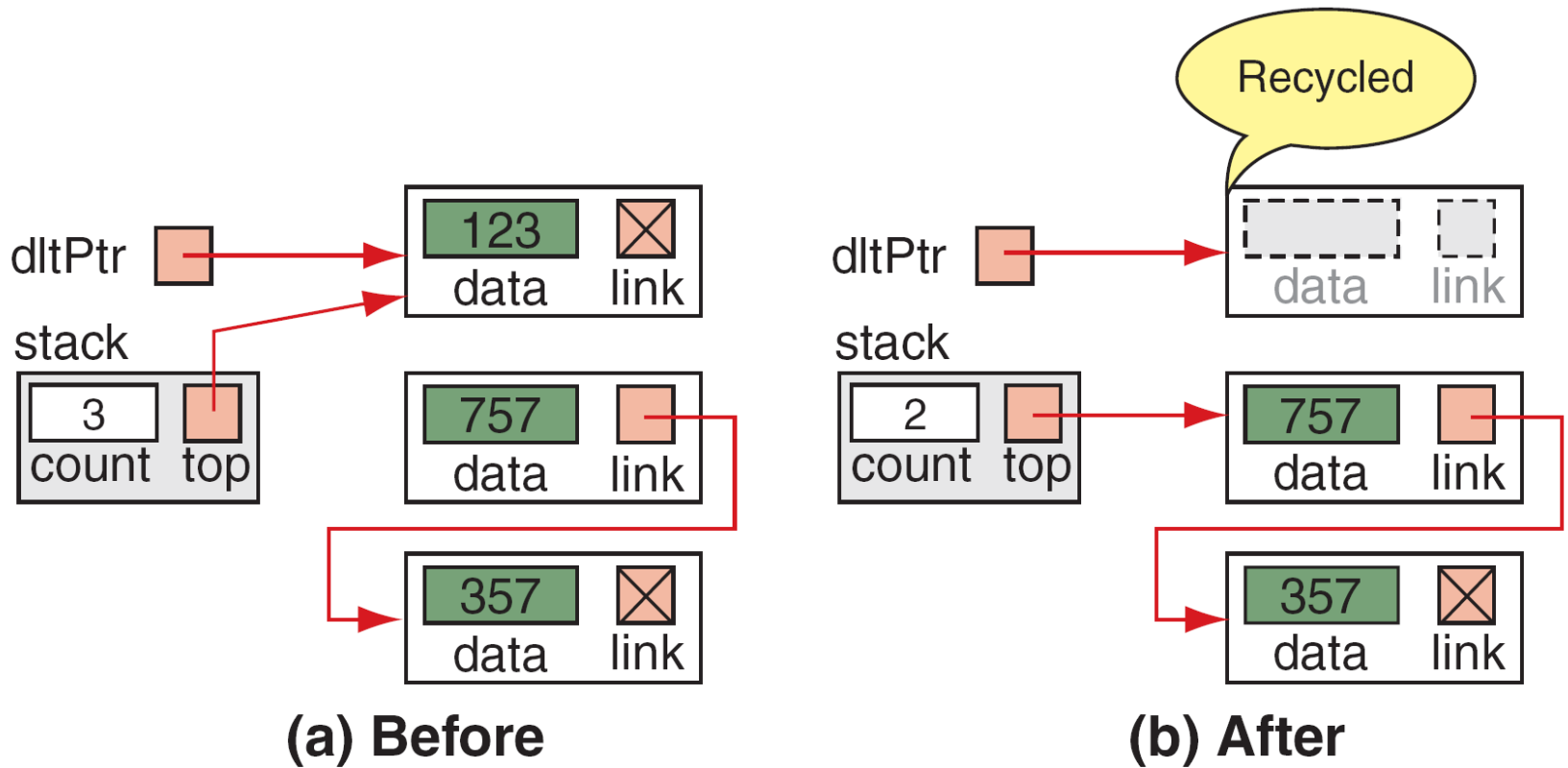
**FIGURE 15-20** Streams

## PROGRAM 15-9 Push Stack

```
1  /* ===== push =====
2  Inserts node into linked list stack.
3  Pre    pStack is pointer to valid stack header
4  Post   dataIn inserted
5  Return true  if successful
6         false if overflow
7  */
8  bool push (STACK* pStack, int dataIn)
9  {
10 // Local Declarations
11     STACK_NODE* pNew;
12     bool        success;
13
14 // Statements
15     pNew = (STACK_NODE*)malloc(sizeof (STACK_NODE));
16     if (!pNew)
17         success = false;
```

## PROGRAM 15-9 Push Stack

```
18     else
19     {
20         pNew->data = dataIn;
21         pNew->link = pStack->top;
22         pStack->top = pNew;
23         pStack->count++;
24         success = true;
25     } // else
26     return success;
27 } // push
```



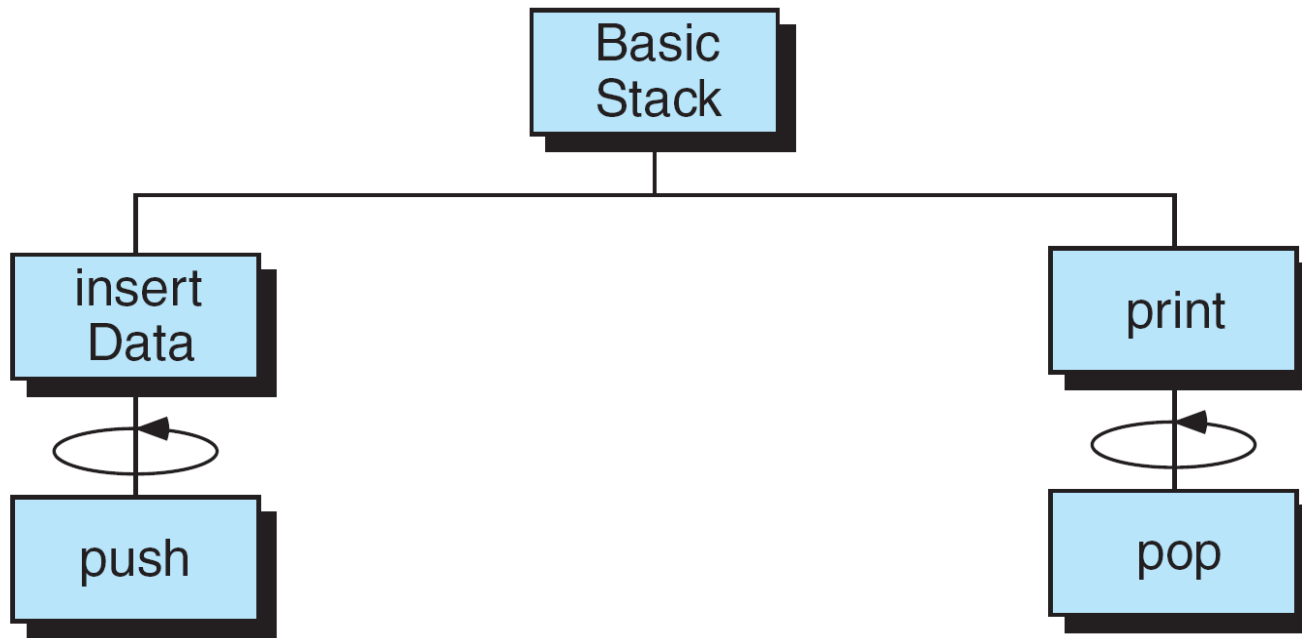
**FIGURE 15-21** Pop Stack Example

## PROGRAM 15-10 Pop Stack

```
1  /* ===== pop =====
2  Delete node from linked list stack.
3  Pre    pStackTop is pointer to valid stack
4  Post   dataOut contains deleted data
5  Return true  if successful
6         false if underflow
7  */
8  bool pop (STACK* pStack, int* dataOut)
9  {
10 // Local Declarations
11     STACK_NODE* pDlt;
12     bool        success;
13
14 // Statements
15     if (pStack->top)                // Test for Empty Stack
16     {
17         success = true;
18         *dataOut = pStack->top->data;
```

## PROGRAM 15-10 Pop Stack

```
20     pStack->top = (pStack->top)->link;
21     pStack->count--;
22     free (pDlt);
23     } // else
24 else
25     success = false;
26     return success;
27 }
```



**FIGURE 15-22** Design for Basic Stack Program

## PROGRAM 15-11 Simple Stack Application Program

```
1  /* This program is a test driver to demonstrate the
2     basic operation of the stack push and pop functions.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <stdbool.h>
9
10 // Global Declarations
11 typedef struct node
12     {
13     int          data;
14     struct node* link;
15     } STACK_NODE;
16
17 typedef struct
```

## PROGRAM 15-11 Simple Stack Application Program

```
18     {
19         int          count;
20         STACK_NODE* top;
21     } STACK;
22
23     // Function Declarations
24     void insertData (STACK* pStack);
25     void print      (STACK* pStack);
26     bool push       (STACK* pList, int  dataIn);
27     bool pop        (STACK* pList, int* dataOut);
28
29     int main (void)
30     {
31         // Local Declarations
32         STACK* pStack;
33
34         // Statements
35         printf("Beginning Simple Stack Program\n");
36
```

## PROGRAM 15-11 Simple Stack Application Program

```
37     pStack = malloc(sizeof(STACK));
38     if (!pStack)
39         printf("Error allocating stack"), exit(100);
40
41     pStack->top    = NULL;
42     pStack->count = 0;
43     insertData (pStack);
44     print      (pStack);
45
46     printf("\nEnd Simple Stack Program\n");
47     return 0;
48 } // main
```

### Results:

Beginning Simple Stack Program

Creating numbers: 854 763 123 532 82 632 33 426 228 90

Stack contained: 90 228 426 33 632 82 532 123 763 854

End Simple Stack Program

## PROGRAM 15-12    Insert Data

```
1  /* ===== insertData =====
2  This program creates random numbers and
3  inserts them into a linked list stack.
4     Pre  pStack is a pointer to first node
5     Post Stack has been created
6  */
7  void insertData (STACK* pStack)
8  {
9  // Local Declarations
10     int  numIn;
11     bool success;
12
13 // Statements
14     printf("Creating numbers: ");
15     for (int nodeCount = 0; nodeCount < 10; nodeCount++)
16     {
17         // Generate random number
18         numIn = rand() % 999;
19         printf("%4d", numIn);
20         success = push(pStack, numIn);
```

## PROGRAM 15-12    Insert Data

```
21         if (!success)
22         {
23             printf("Error 101: Out of Memory\n");
24             exit (101);
25         } // if
26     } // for
27     printf("\n");
28     return;
29 } // insertData
```

## PROGRAM 15-13    Print Stack

```
1  /* ===== print =====
2     This function prints a singly linked stack.
3     Pre      pStack is pointer to valid stack
4     Post    data in stack printed
5  */
6  void print (STACK* pStack)
7  {
8  // Local Declarations
9     int printData;
10
11 // Statements
12     printf("Stack contained: ");
13     while (pop(pStack, &printData))
14         printf("%4d", printData);
15     return;
16 } // print
```

# 15-4 Queues

*A queue is a linear list in which data can be inserted only at one end, called the rear, and deleted from the other end, called the front.*

*Topics discussed in this section:*

**Queue Operations**

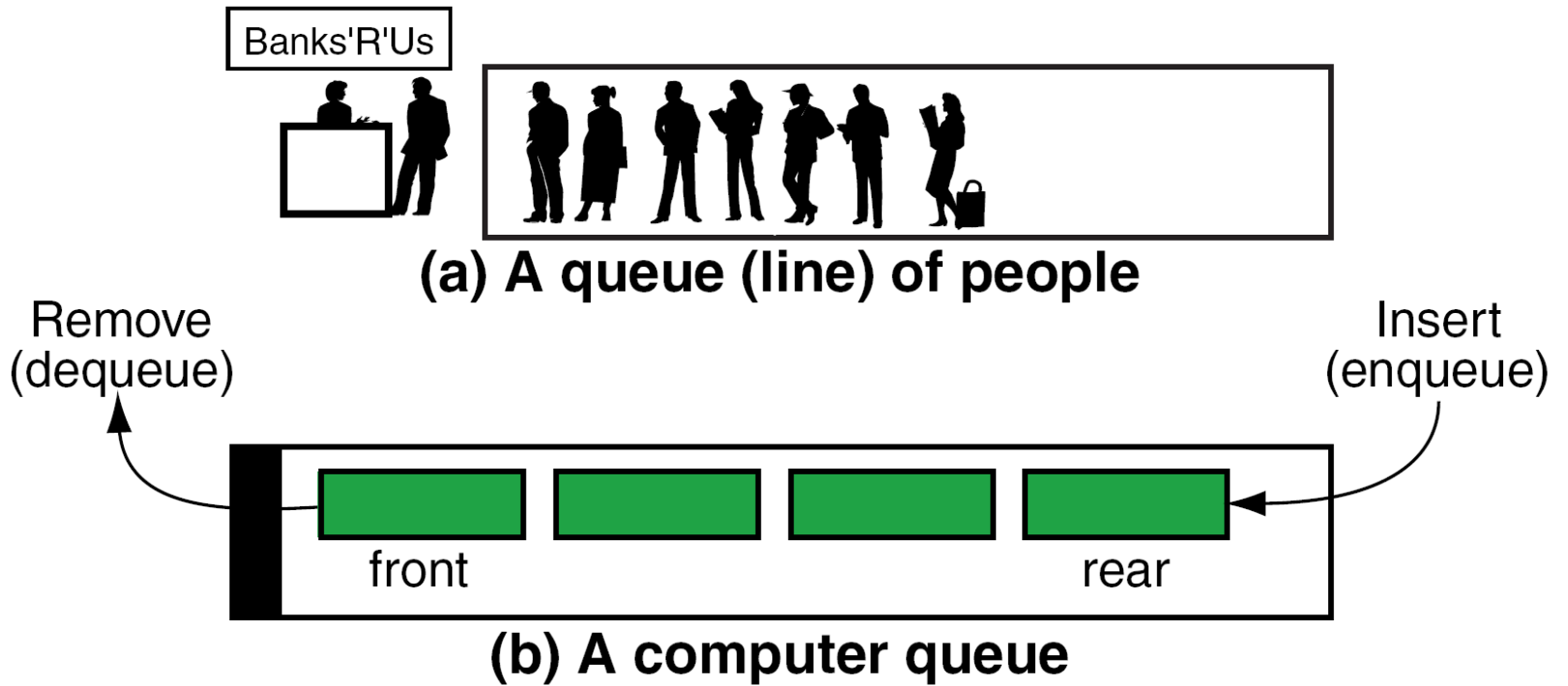
**Queue Linked List Design**

**Queue Functions**

**Queue Demonstration**

## *Note*

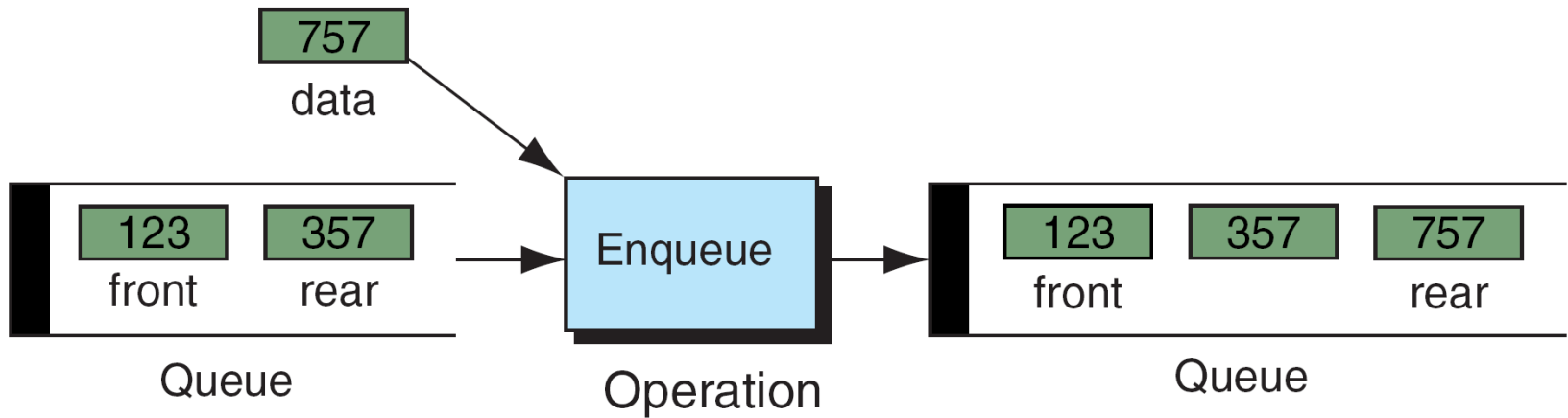
**A queue is a linear list in which data can be inserted at one end, called the rear, and deleted from the other end, called the front. It is a first in–first out (FIFO) restricted data structure.**



**FIGURE 15-23** Queue Concept

## *Note*

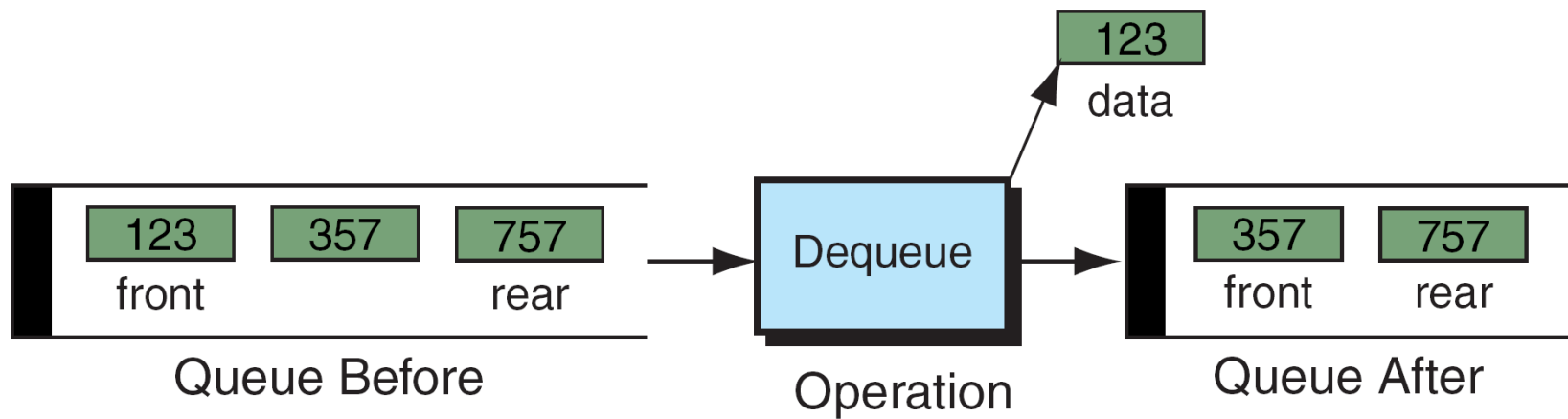
**Enqueue inserts an element at the rear of the queue.**



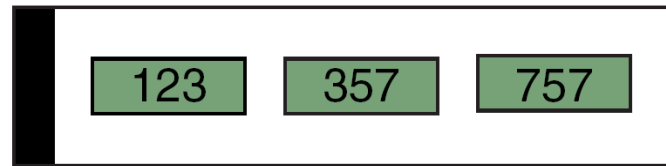
**FIGURE 15-24** Enqueue

## *Note*

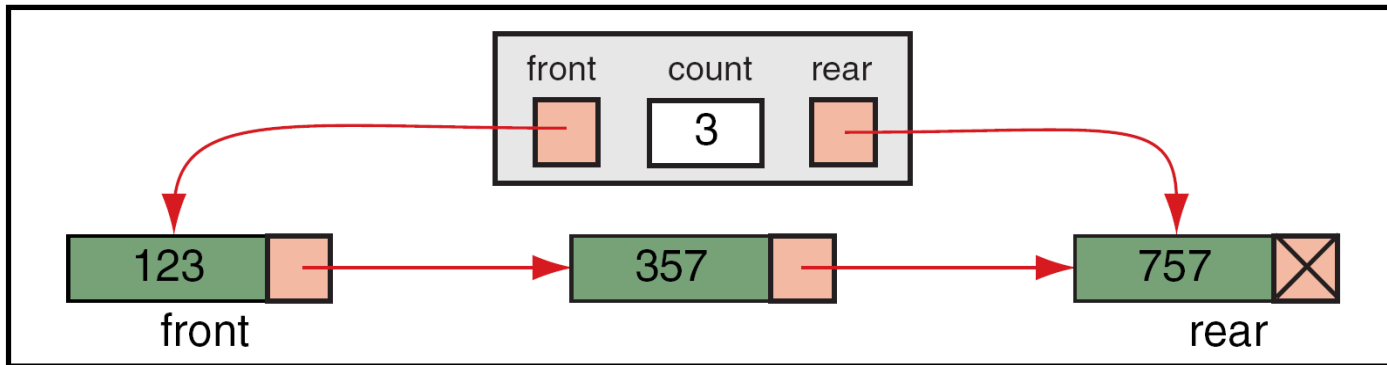
**Dequeue deletes an element at the front of the queue.**



**FIGURE 15-25** Dequeue

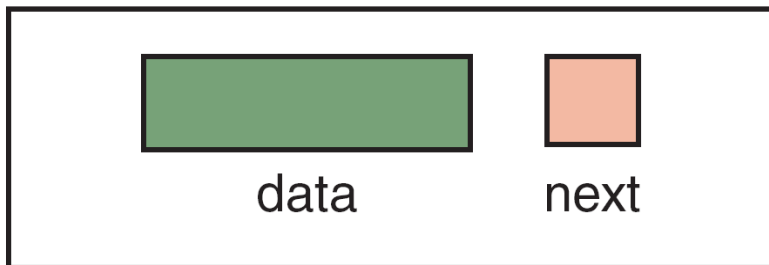


(a) Conceptual queue

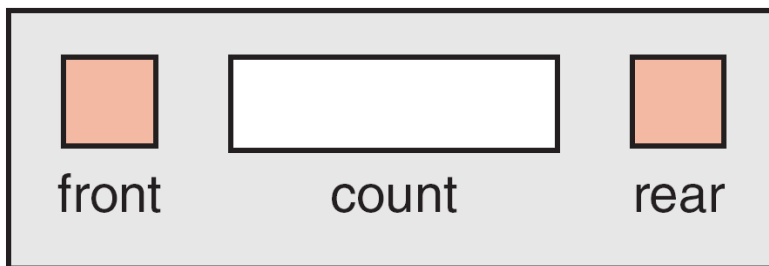


(b) Physical queue

**FIGURE 15-26** Conceptual and Physical Queue Implementations



**Node Structure**



**Head Structure**

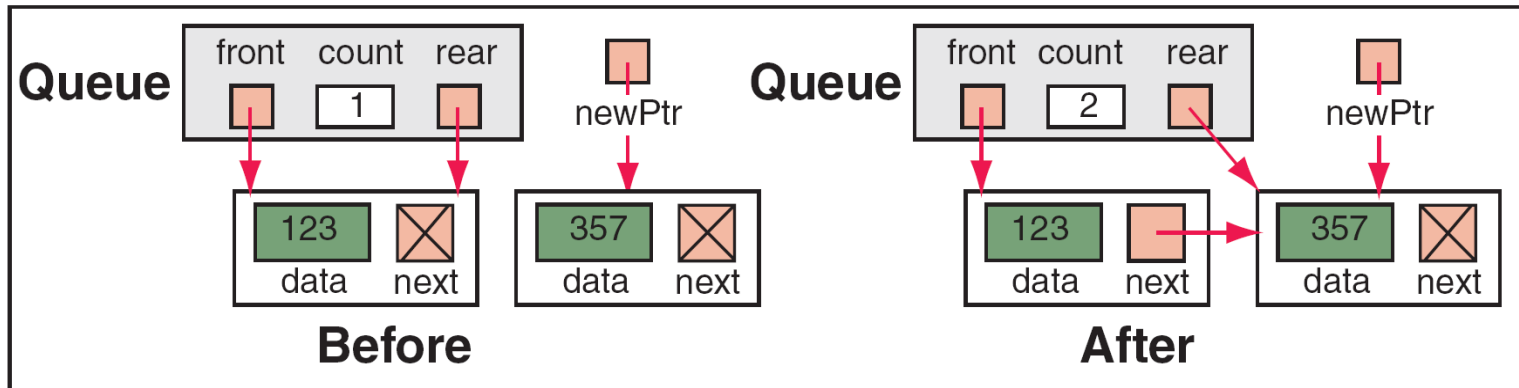
```
typedef struct node
{
    int data;
    struct node* next;
} QUEUE_NODE;
```

```
typedef struct
{
    QUEUE_NODE* front;
    int count;
    QUEUE_NODE* rear;
} QUEUE;
```

**FIGURE 15-27** Queue Data Structure



**(a) Case 1: Insert into Null Queue**



**(b) Case 2: Insert into Queue with Data**

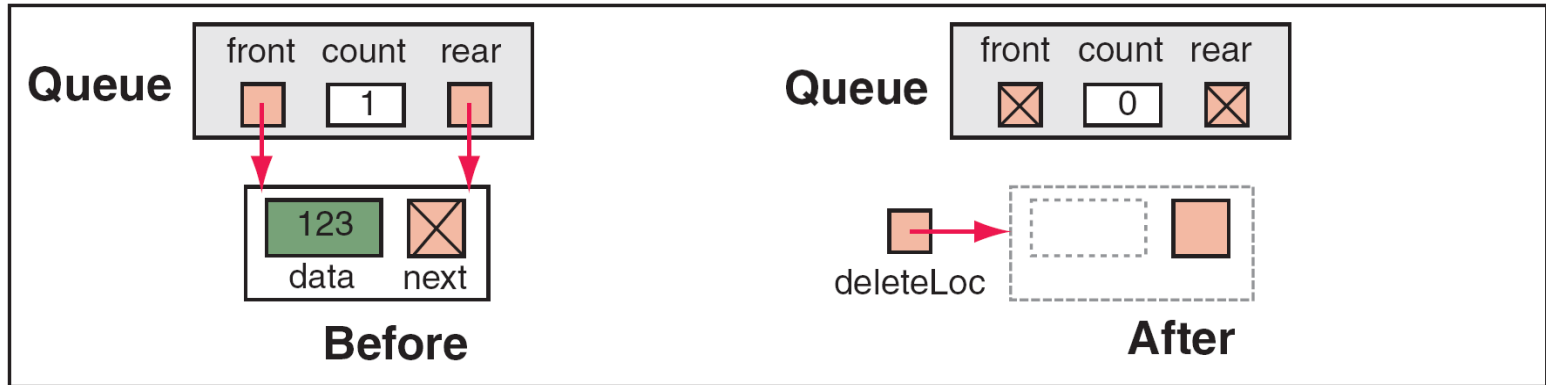
**FIGURE 15-28 Enqueue Example**

## PROGRAM 15-14 Enqueue

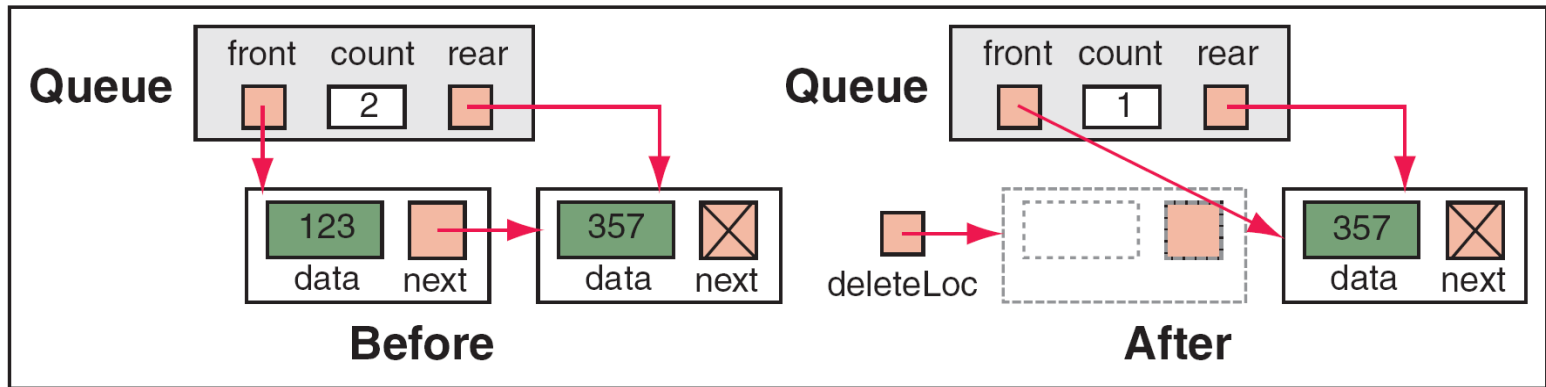
```
1  /* ===== enqueue =====
2     This algorithm inserts data into a queue.
3     Pre    queue is valid
4     Post   data have been inserted
5     Return true if successful, false if overflow
6  */
7  bool enqueue (QUEUE* queue, int dataIn)
8  {
9  // Local Declarations
10     QUEUE_NODE* newPtr;
11
12 // Statements
13     if (!(newPtr = malloc(sizeof(QUEUE_NODE))))
14         return false;
15
16     newPtr->data = dataIn;
17     newPtr->next = NULL;
18
```

## PROGRAM 15-14 Enqueue

```
19     if (queue->count == 0)
20         // Inserting into null queue
21         queue->front = newPtr;
22     else
23         queue->rear->next = newPtr;
24     (queue->count)++;
25     queue->rear = newPtr;
26     return true;
27 }
```



**(a) Case 1: Delete only item in queue**



**(b) Case 2: Delete item at front of queue**

**FIGURE 15-29 Dequeue Examples**

## PROGRAM 15-15    Dequeue

```
1  /* ===== dequeue =====
2     This algorithm deletes a node from the queue.
3     Pre     queue is pointer to queue head structure
4            dataOut is pointer to data being deleted
5     Post    Data pointer to queue front returned and
6            front element deleted and recycled.
7     Return  true if successful; false if underflow
8  */
9  bool dequeue (QUEUE* queue, int* dataOut)
10 {
11 // Local Declarations
12     QUEUE_NODE* deleteLoc;
13
14 // Statements
15     if (!queue->count)
16         return false;
17
```

## PROGRAM 15-15 Dequeue

```
18     *dataOut = queue->front->data;
19     deleteLoc = queue->front;
20     if (queue->count == 1)
21         // Deleting only item in queue
22         queue->rear = queue->front = NULL;
23     else
24         queue->front = queue->front->next;
25     (queue->count)--;
26     free (deleteLoc);
27
28     return true;
29 }
```

## PROGRAM 15-16 Simple Queue Demonstration

```
1  /* This program is a test driver to demonstrate the
2     basic operation of the enqueue and dequeue functions.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <stdbool.h>
10
11 // Global Declarations
12 typedef struct node
13     {
14     int          data;
15     struct node* next;
16     } QUEUE_NODE;
17
18 typedef struct
19     {
20     QUEUE_NODE* front;
```

## PROGRAM 15-16 Simple Queue Demonstration

```
21     int          count;
22     QUEUE_NODE* rear;
23     } QUEUE;
24
25     // Function Declarations
26     void insertData (QUEUE* pQueue);
27     void print      (QUEUE* pQueue);
28     bool enqueue   (QUEUE* pList, int dataIn);
29     bool dequeue   (QUEUE* pList, int* dataOut);
30
31     int main (void)
32     {
33     // Local Declarations
34     QUEUE* pQueue;
35
36     // Statements
37     printf("Beginning Simple Queue Program\n");
38
39     pQueue = malloc(sizeof(QUEUE));
40     if (!pQueue)
41         printf("Error allocating queue"), exit(100);
42
```

## PROGRAM 15-16 Simple Queue Demonstration

```
43     pQueue->front = NULL;
44     pQueue->count = 0;
45     pQueue->rear  = NULL;
46
47     insertData (pQueue);
48     print      (pQueue);
49
50     printf("\nEnd Simple Queue Program\n");
51     return 0;
52 } // main
```

### Results:

```
Beginning Simple Queue Program
Creating numbers:  854 763 123 532  82
Queue contained:  854 763 123 532  82
End Simple Queue Program
```

## PROGRAM 15-17    Insert Data

```
1  /* ===== insertData =====
2  This program creates random number data and
3  inserts them into a linked list queue.
4  Pre  pQueue is a pointer to first node
5  Post Queue created and filled
6  */
7  void insertData (QUEUE* pQueue)
8  {
9  // Local Declarations
10     int  numIn;
11     bool success;
12
13 // Statements
14     printf("Creating numbers: ");
15     for (int nodeCount = 0; nodeCount < 5; nodeCount++)
16     {
17         // Generate random number
18         numIn = rand() % 999;
19         printf("%4d", numIn);
```

## PROGRAM 15-17    Insert Data

```
20         success = enqueue(pQueue, numIn);
21         if (!success)
22             {
23                 printf("Error 101: Out of Memory\n");
24                 exit (101);
25             } // if
26         } // for
27     printf("\n");
28     return;
29 } // insertData
```

## PROGRAM 15-18 Print Queue

```
1  /* ===== print =====
2     This function prints a singly linked queue.
3     Pre     pQueue is pointer to valid queue
4     Post    data in queue printed
5  */
6  void print (QUEUE* pQueue)
7  {
8  // Local Declarations
9     int printData;
10
11 // Statements
12     printf("Queue contained: ");
13     while (dequeue(pQueue, &printData))
14         printf("%4d", printData);
15     return;
16 }
```

# 15-5 Trees

*Trees are used extensively in computer science to represent algebraic formulas; as an efficient method for searching large, dynamic lists; and for such diverse applications as artificial intelligence systems and encoding algorithms.*

*Topics discussed in this section:*

**Basic Tree Concepts**

**Terminology**

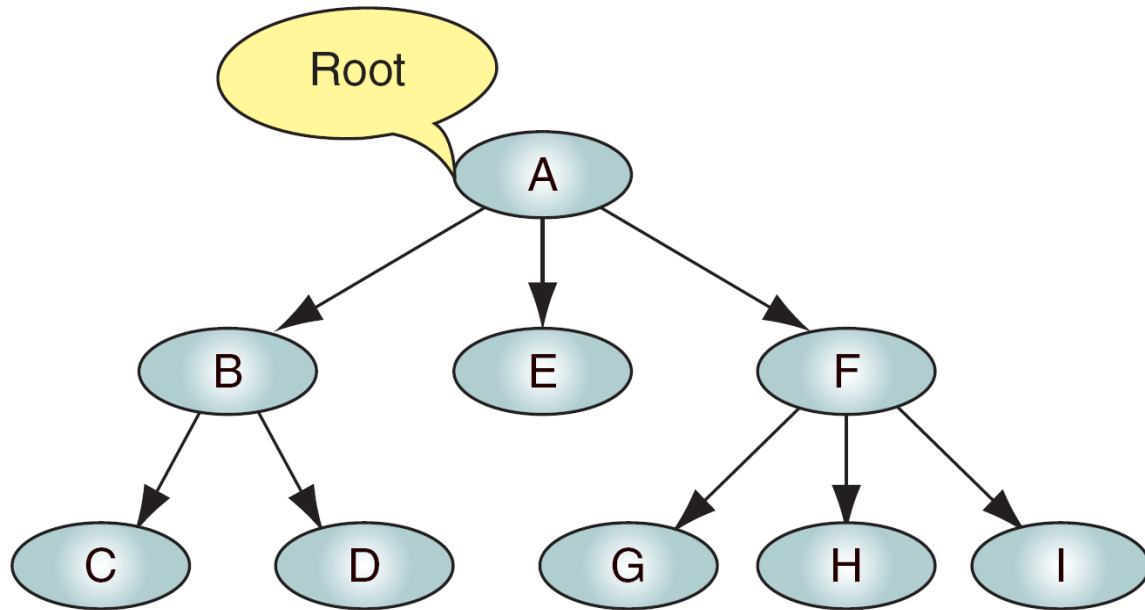
**Binary Trees**

**Binary Search Trees**

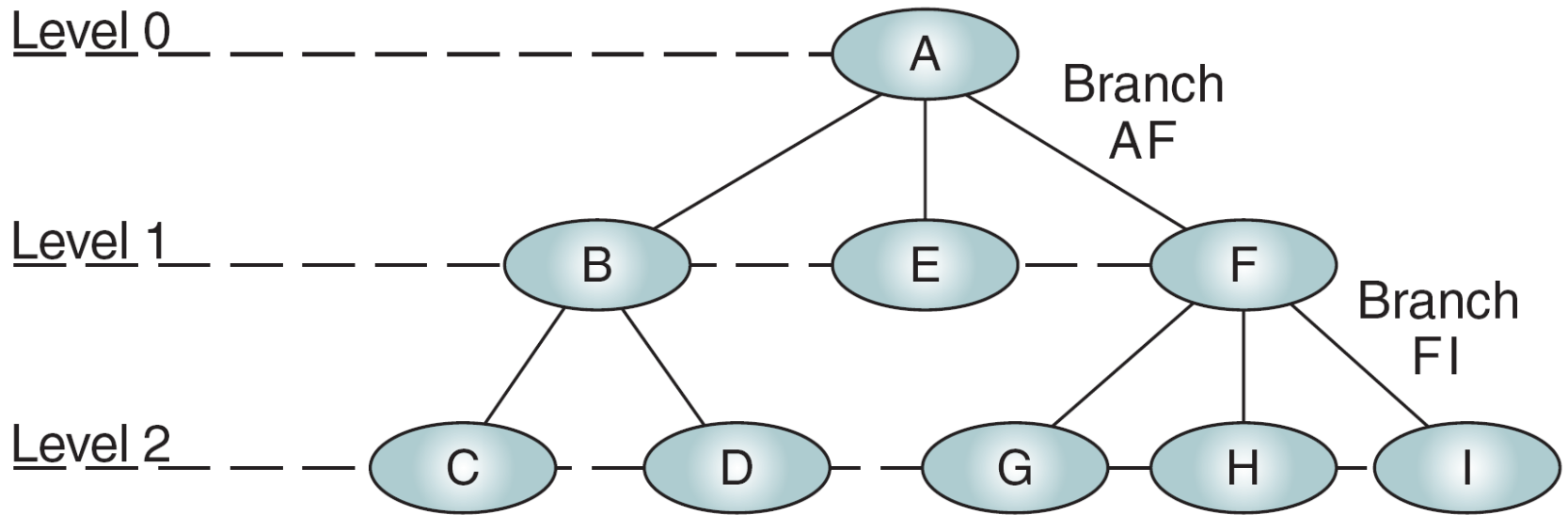
**Binary Tree Example**

## *Note*

**A tree consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches, that connect the nodes.**



**FIGURE 15-30** Tree



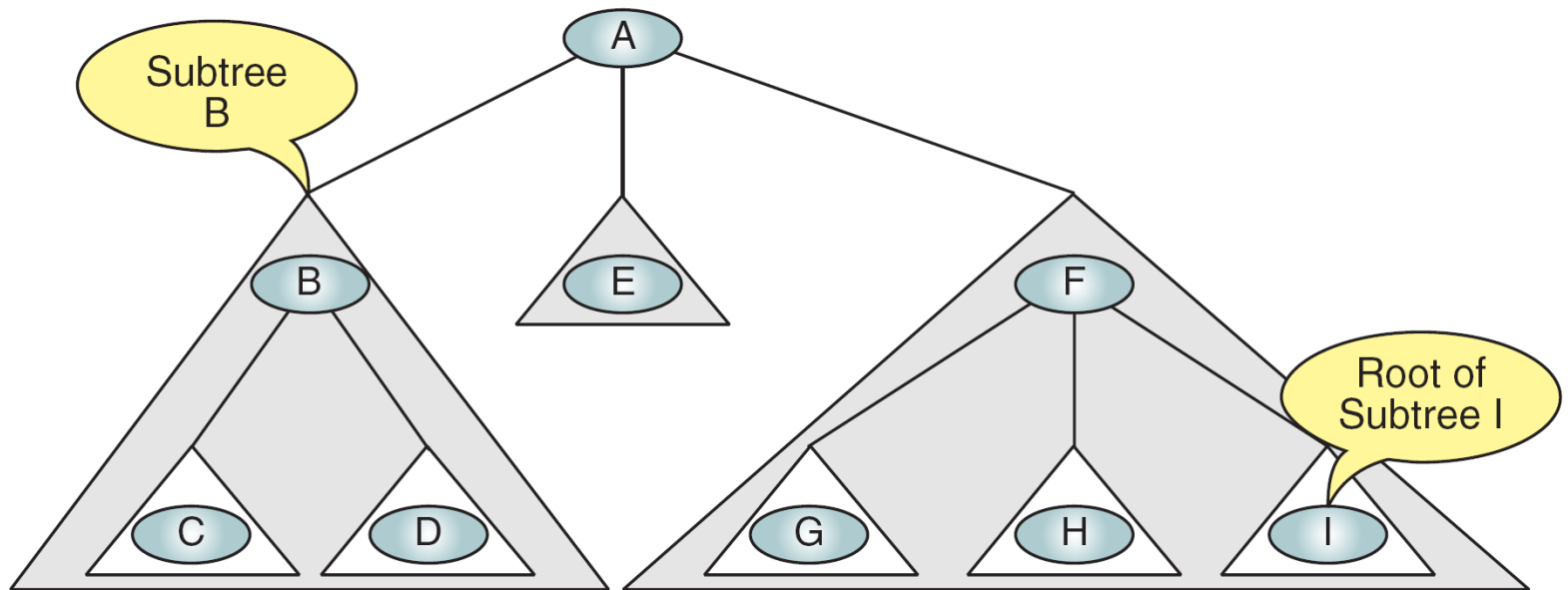
Root: A  
 Parents: A, B, F  
 Children: B, E, F, C, D, G, H, I

Siblings: {B, E, F}, {C, D}, {G, H, I}  
 Leaves: C, D, E, G, H, I  
 Internal nodes: B, F

**FIGURE 15-31** Tree Nomenclature

## *Note*

**The level of a node is its distance from the root. The height of a tree is the level of the leaf in the longest path from the root plus 1.**

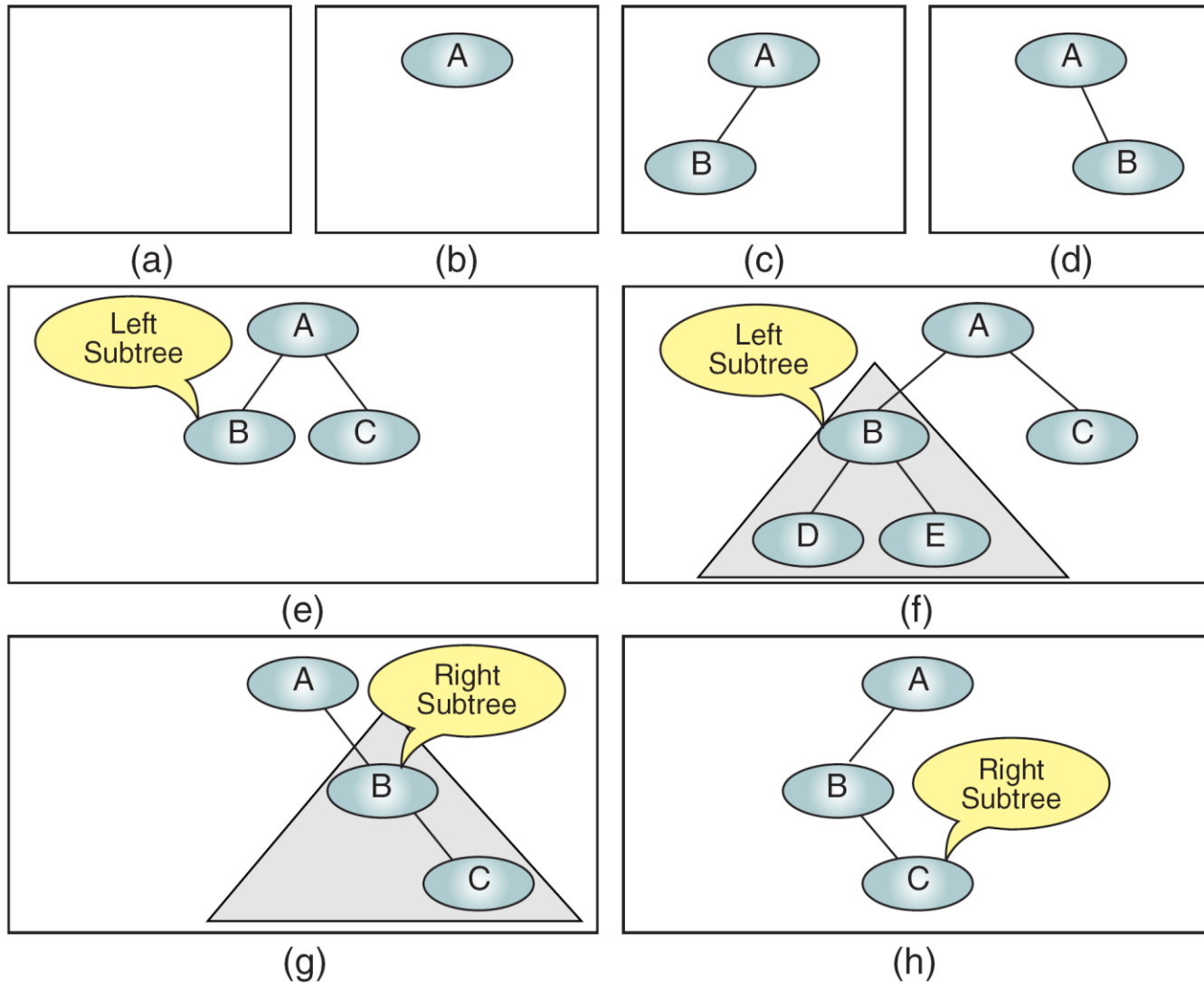


**FIGURE 15-32** Subtrees

## *Note*

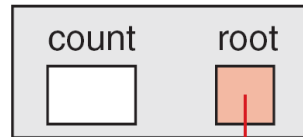
**A tree is a set of nodes that either:**

- 1. Is empty, or**
- 2. Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees**



**FIGURE 15-33** Collection of Binary Trees

BIN\_TREE



to tree

NODE



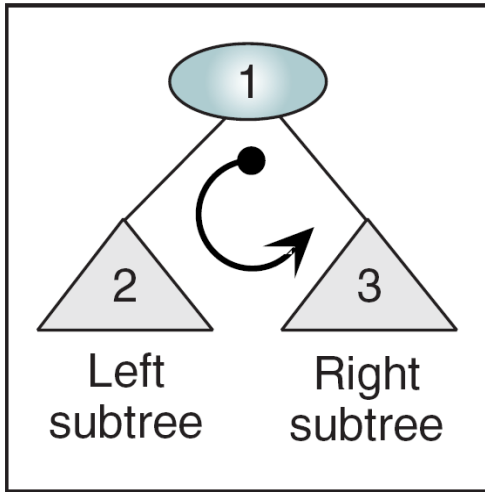
to left  
subtree

to right  
subtree

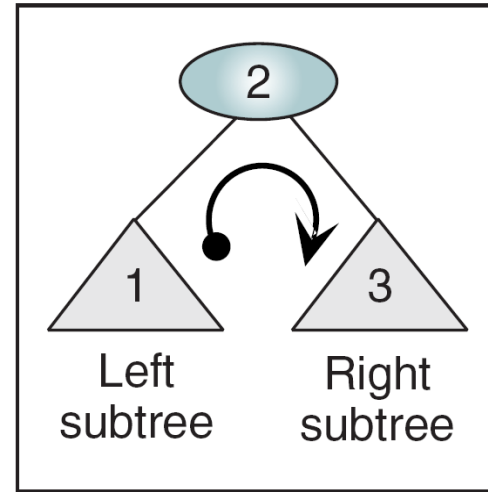
```
typedef struct
{
    int    count;
    NODE*  root;
} BIN_TREE;

typedef struct node
{
    int          data;
    struct node* left;
    struct node* right;
} NODE;
```

**FIGURE 15-34** Binary Tree Data Structure

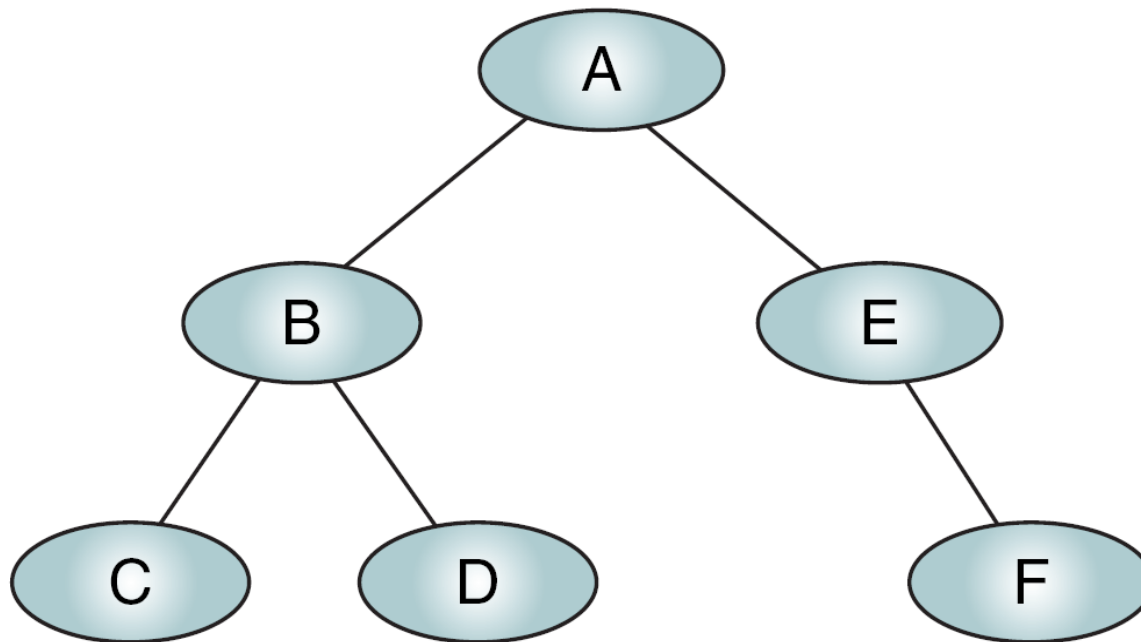


**(a) Preorder Traversal**



**(b) Inorder Traversal**

**FIGURE 15-35 Binary Tree Traversals**



---

**FIGURE 15-36** Binary Tree for Traversals

---

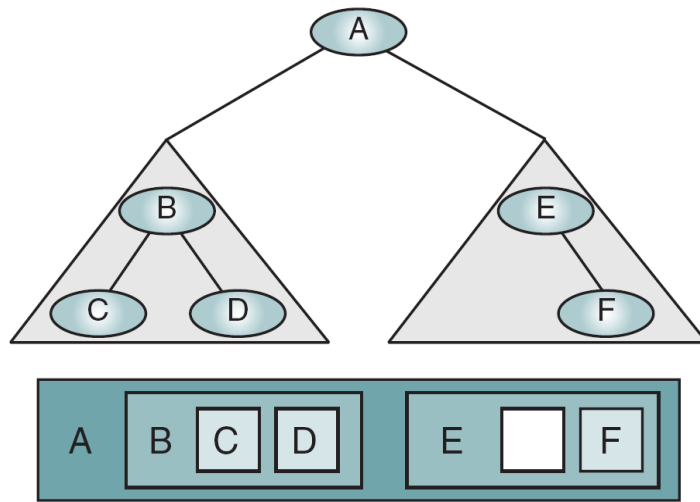
## *Note*

**In the preorder traversal, the root is processed first,  
before its subtrees.**

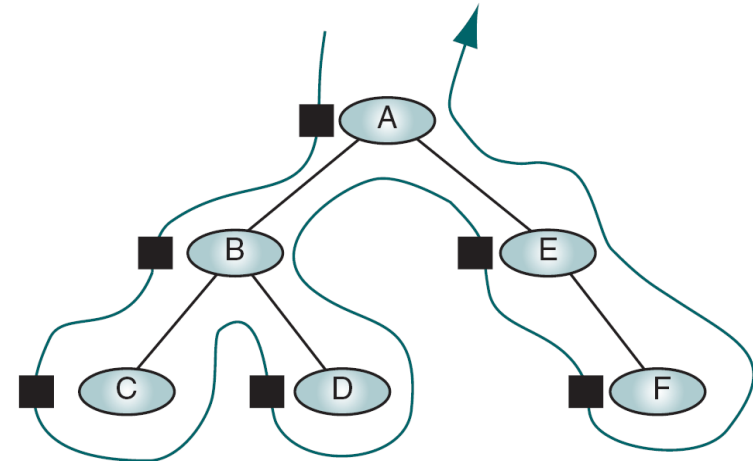
## PROGRAM 15-19

## Preorder Traversal of a Binary Tree

```
1  /* Traverse a binary tree and print its data (integers)
2     Pre root is entry node of a tree or subtree
3     Post each node has been printed
4  */
5  void preOrder (NODE* root)
6  {
7  // Statements
8     if (root)
9     {
10        printf("%4d", root->data);
11        preOrder (root->left);
12        preOrder (root->right);
13    } // if
14    return;
15 } // preOrder
```

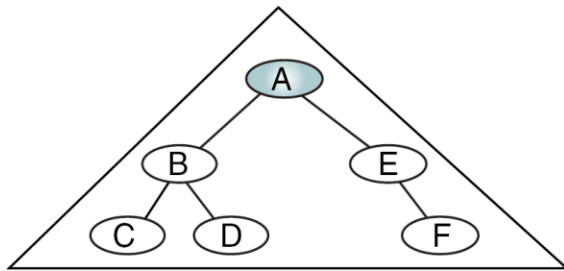


(a) Processing Order

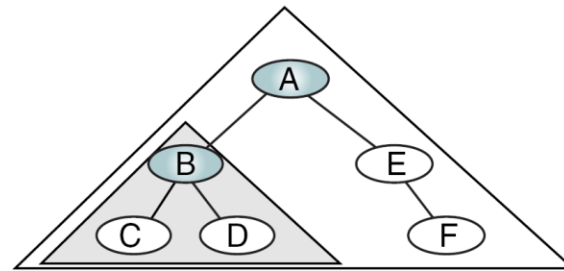


(b) "Walking" Order

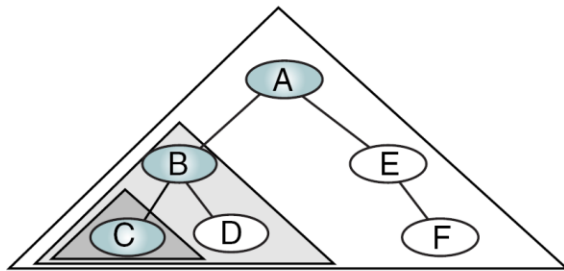
**FIGURE 15-37** Preorder Traversal—A B C D E F



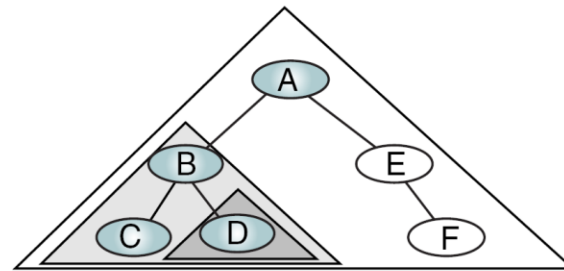
(a) Process Tree A



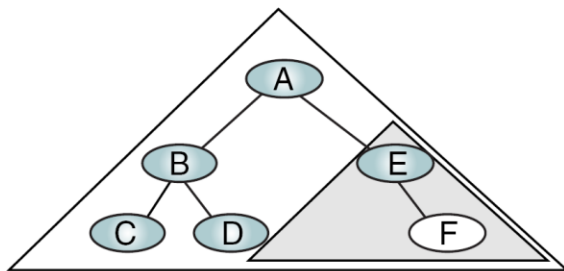
(b) Process Tree B



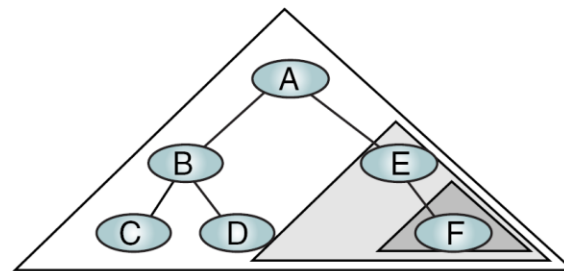
(c) Process Tree C



(d) Process Tree D



(e) Process Tree E

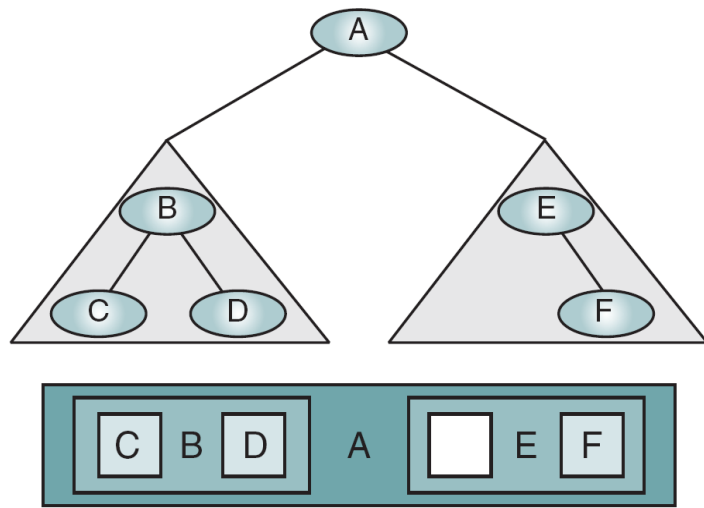


(f) Process Tree F

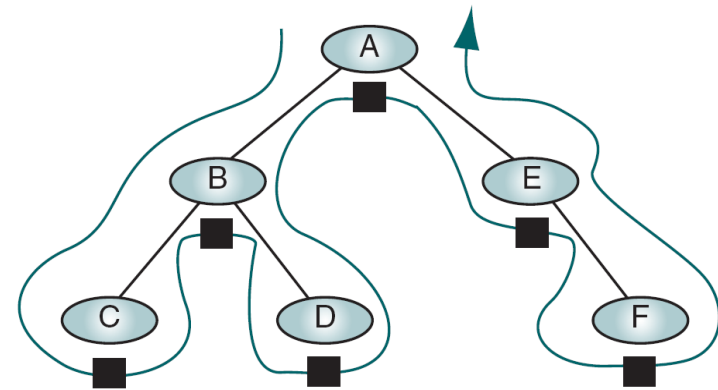
**FIGURE 15-38** Algorithmic Traversal of Binary Tree

## PROGRAM 15-20 Inorder Traversal of a Binary Tree

```
1  /* Traverse a binary tree and print its data (integers)
2     Pre  root is entry node of a tree or subtree
3     Post each node has been printed
4  */
5  void inOrder (NODE* root)
6  {
7  // Statements
8     if (root)
9     {
10        inOrder (root->left);
11        printf("%4d", root->data);
12        inOrder (root->right);
13    } // if
14    return;
15 }
```



(a) Processing Order



(b) "Walking" Order

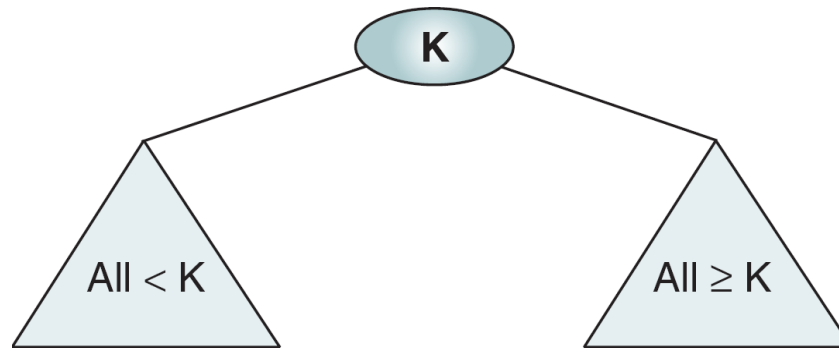
**FIGURE 15-39** Inorder Traversal—C B D A E F

## *Note*

**In the inorder traversal, the root is processed  
between its subtrees.**

## *Note*

**In a binary search tree, the left subtree contains key values less than the root, and the right subtree contains key values greater than or equal to the root.**



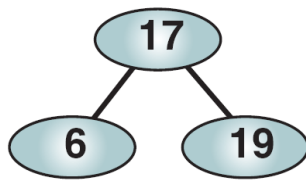
---

**FIGURE 15-40** Binary Search Tree

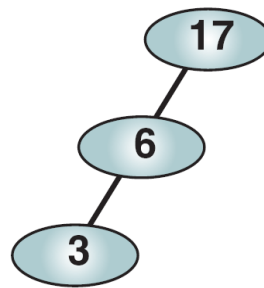
---



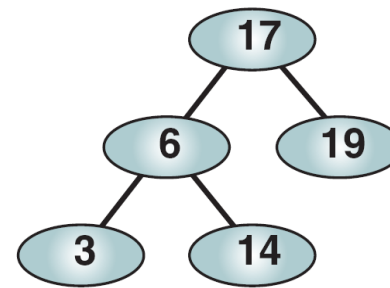
(a)



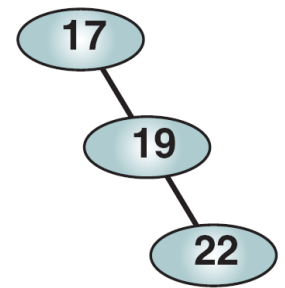
(b)



(c)

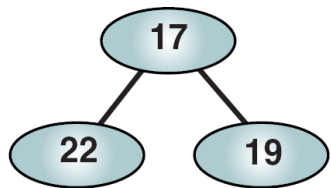


(d)

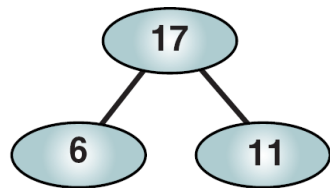


(e)

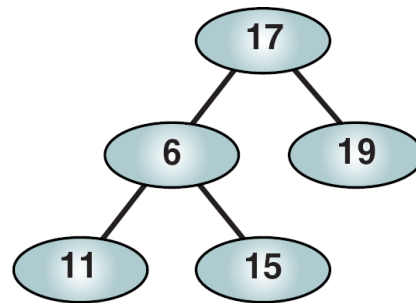
**FIGURE 15-41** Valid Binary Search Trees



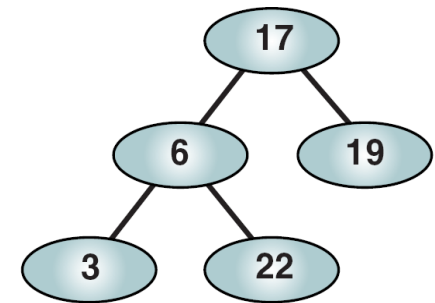
(a)



(b)



(c)

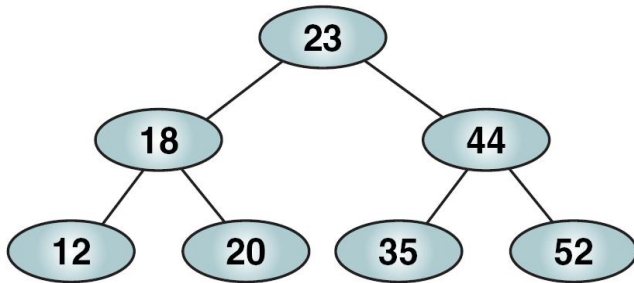


(d)

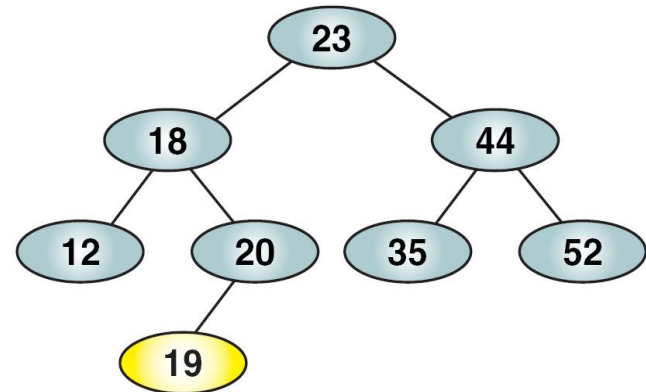
**FIGURE 15-42** Invalid Binary Search Trees

## *Note*

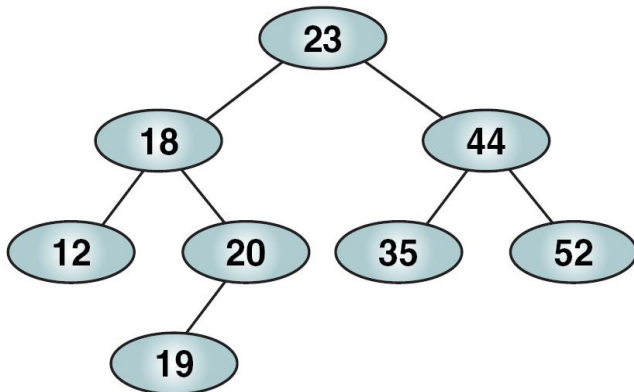
**All BST insertions take place at a leaf or a leaflike node.**



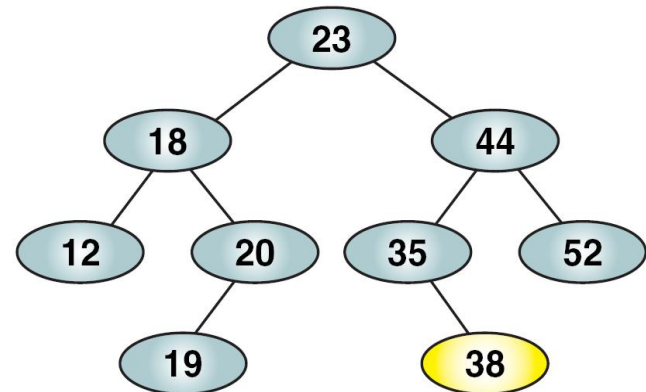
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38



(d) After inserting 38

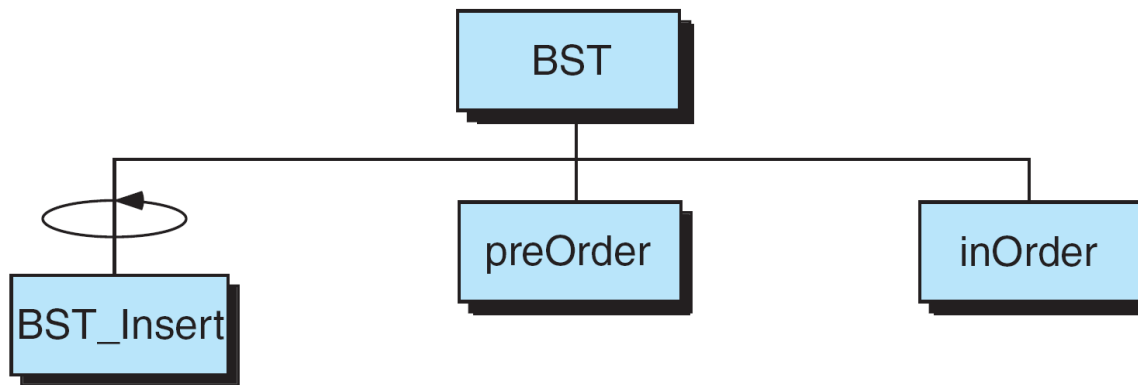
**FIGURE 15-43** BST Insertion

## PROGRAM 15-21 Binary Tree Insert Function

```
1  /* ===== BST_Insert =====
2  This function uses recursion to insert the new data
3  into a leaf node in the BST tree.
4  Pre    Application has called BST_Insert, which
5         passes root and data pointer
6  Post   Data have been inserted
7  Return pointer to [potentially] new root
8  */
9  NODE* BST_Insert (BST_TREE* tree,
10                  NODE* root,    int dataIn)
11  {
12  // Local Declarations
13  NODE* newPtr;
14
15  // Statements
16
17  if (!root)
18  {
19      // NULL tree -- create new node
20      newPtr = malloc(sizeof (NODE));
```

## PROGRAM 15-21 Binary Tree Insert Function

```
21     if (!newPtr)
22         printf("Overflow in Insert\n"), exit (100);
23     newPtr->data = dataIn;
24     newPtr->left = newPtr->right = NULL;
25     return newPtr;
26 } // if
27
28 // Locate null subtree for insertion
29 if (dataIn < root->data)
30     root->left = BST_Insert(tree, root->left,
31                             dataIn);
32 else
33     // new data >= root data
34     root->right = BST_Insert(tree, root->right,
35                             dataIn);
36 return root;
37 } // BST_Insert
```



**FIGURE 15-44** Binary Tree Example

## PROGRAM 15-22 Binary Tree Example

```
1  /* Demonstrate the binary search tree insert and
2     traversals.
3     Written by:
4     Date:
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  // Global Declarations
10 typedef struct node
11     {
12     int          data;
13     struct node* left;
14     struct node* right;
15     } NODE;
16
17 typedef struct
18     {
19     int    count;
20     NODE* root;
```

## PROGRAM 15-22 Binary Tree Example

```
21     } BST_TREE;
22
23     // Function Declarations
24     void preOrder (NODE* root);
25     void inOrder (NODE* root);
26     NODE* BST_Insert (BST_TREE* tree,
27                       NODE* root,      int data);
28
29     int main (void)
30     {
31     // Local Declarations
32         int      numIn;
33         BST_TREE tree;
34
35     // Statements
36         printf("Please enter a series of integers."
37              "\nEnter a negative number to stop\n");
38
39         tree.count = 0;
40         tree.root  = NULL;
```

## PROGRAM 15-22 Binary Tree Example

```
41     do
42         {
43             printf("Enter a number: ");
44             scanf("%d", &numIn);
45             if (numIn > 0)
46                 {
47                     tree.root = BST_Insert
48                         (&tree, tree.root, numIn);
49                     tree.count++;
50                 } // if
51         } while (numIn > 0);
52
53     printf("\nData in preOrder: ");
54     preOrder (tree.root);
55
56     printf("\n\nData in inOrder:  ");
57     inOrder (tree.root);
58
59     printf("\n\nEnd of BST Demonstration\n");
60     return 0;
61 } // main
```

## PROGRAM 15-22 Binary Tree Example

### Results

Please enter a series of integers.

Enter a negative number to stop

Enter a number: 45

Enter a number: 54

Enter a number: 23

Enter a number: 32

Enter a number: 3

Enter a number: -1

Data in preOrder: 45 23 3 32 54

Data in inOrder: 3 23 32 45 54

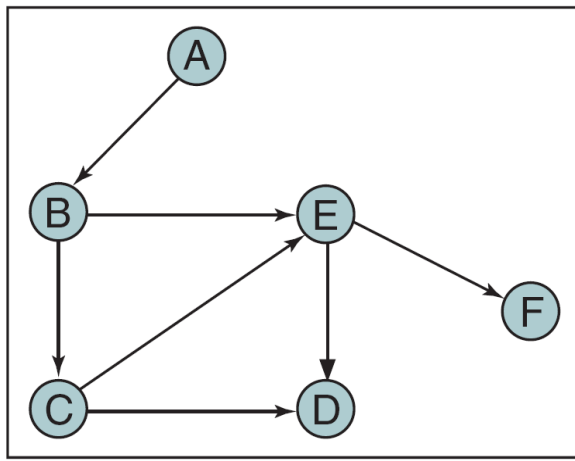
End of BST Demonstration

# 15-6 Graphs

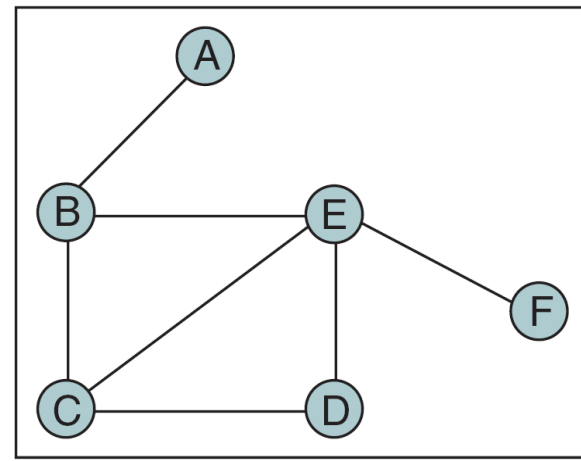
*A graph is a collection of nodes, called vertices, and a collection of segments, called lines, connecting pairs of vertices. In other words, a graph consists of two sets, a set of vertices and a set of lines.*

*Topics discussed in this section:*

**Graph Traversal**



(a) Directed Graph



(b) Undirected Graph

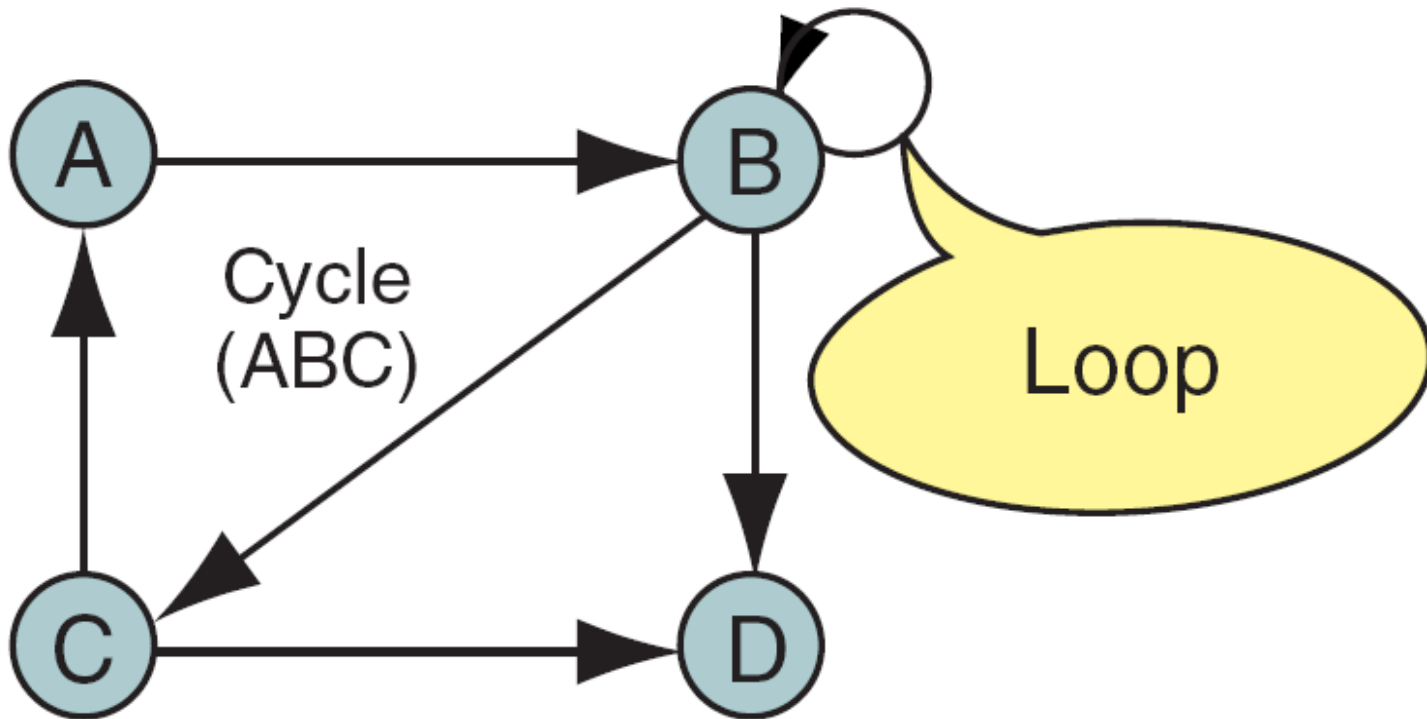
**FIGURE 15-45** Directed and Undirected Graphs

## *Note*

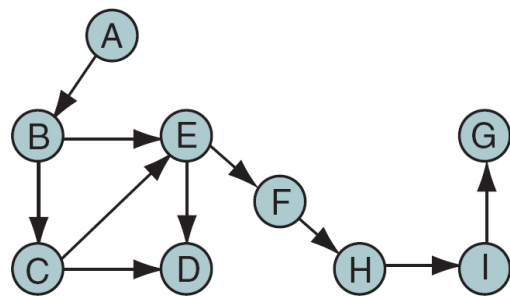
**Graphs may be directed or undirected. In a directed graph, each line, called an arc, has a direction indicating how it may be traversed. In an undirected graph, the line is known as an edge, and it may be traversed in either direction.**

## *Note*

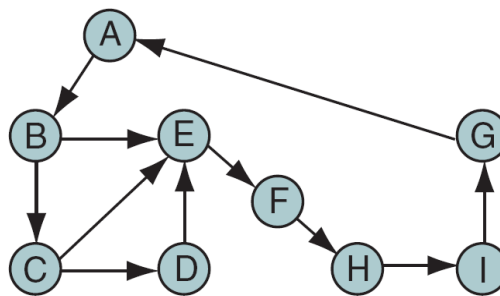
**A file is an external collection of related data treated as a unit.**



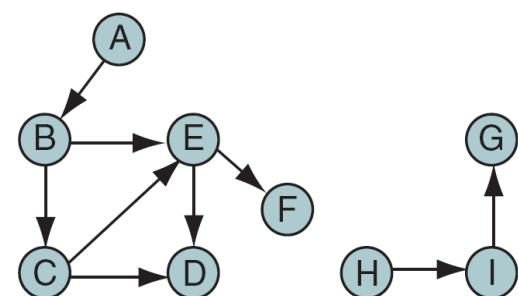
**FIGURE 15-46** Cycles and Loops



(a) Weakly Connected

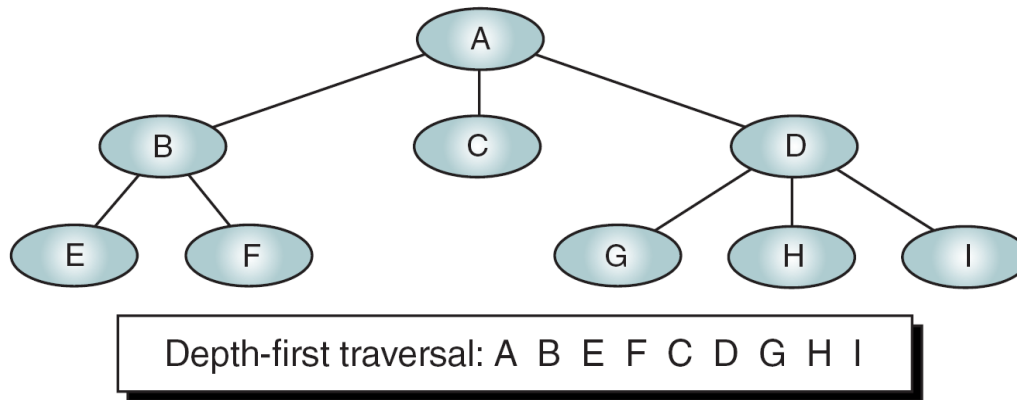


(b) Strongly Connected



(c) Disjoint Graph

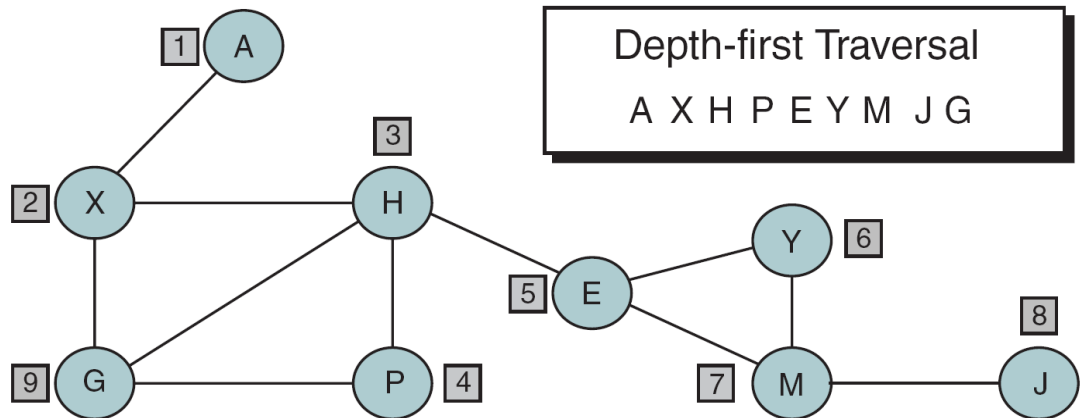
**FIGURE 15-47** Connected and Disjoint Graphs



**FIGURE 15-48** Depth-first Traversal of a Tree

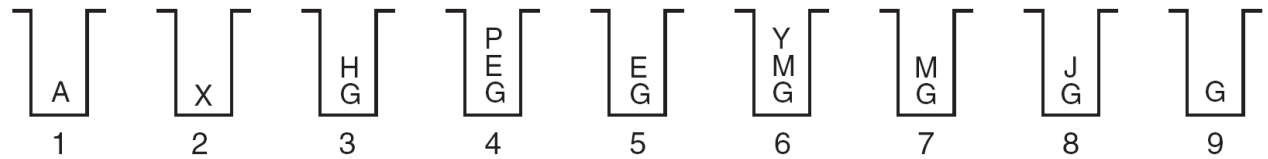
## *Note*

**In the depth-first traversal, all of a node's descendants are processed before moving to an adjacent node.**



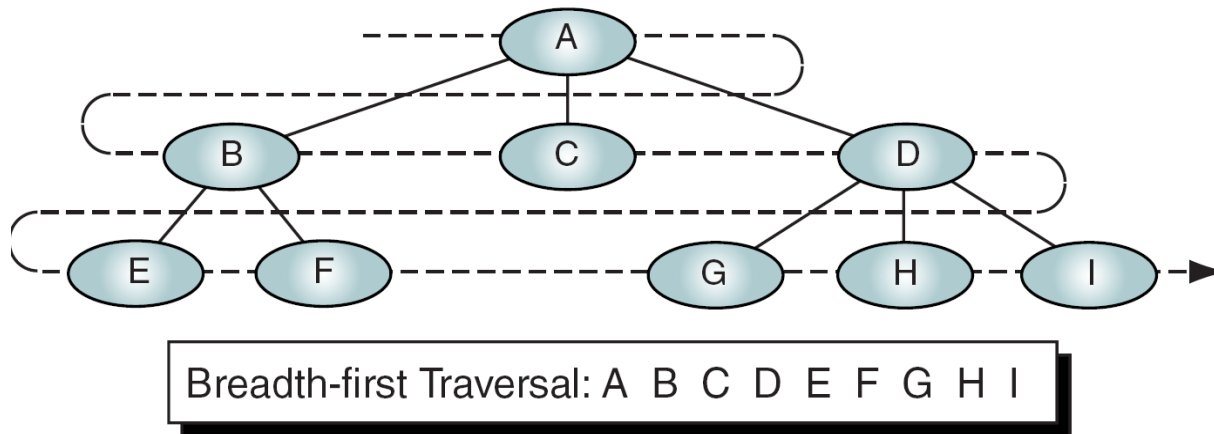
Depth-first Traversal  
 A X H P E Y M J G

**(a) Graph**

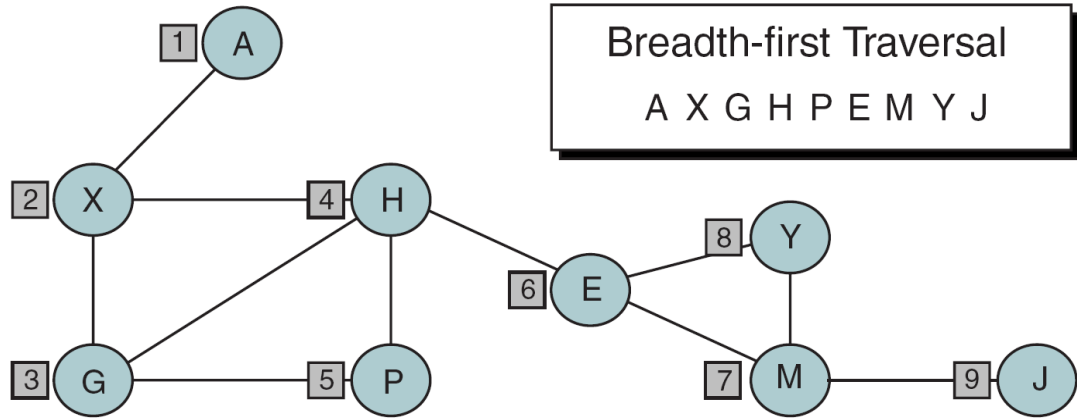


**(b) Stack Contents**

**FIGURE 15-49** Depth-first Traversal of a Graph



**FIGURE 15-50** Breadth-first Traversal of a Tree



Breadth-first Traversal  
A X G H P E M Y J

(a) Graph



(b) Queue contents

**FIGURE 15-51** Breadth-first Traversal of a Graph

## *Note*

**In the breadth-first traversal, all adjacent vertices are processed before processing the descendants of a vertex.**

# 15-7 Software Engineering

*Because lists are useful structures, programmers use them in many applications. Rather than rewrite their functions each time we need them, we can write functions once and put them in a library. The name given to a complete set of these functions is abstract data type (ADT).*

*Topics discussed in this section:*

**Atomic and Composite Data**

**Data Structure and Abstract Data Type**

**A Model for an Abstract Data Type**

**ADT Data Structure**

## ***Note***

### **Atomic Data Type**

- 1. A set of values.**
- 2. A set of operations on the values.**

## ***Note***

### **Data Structure**

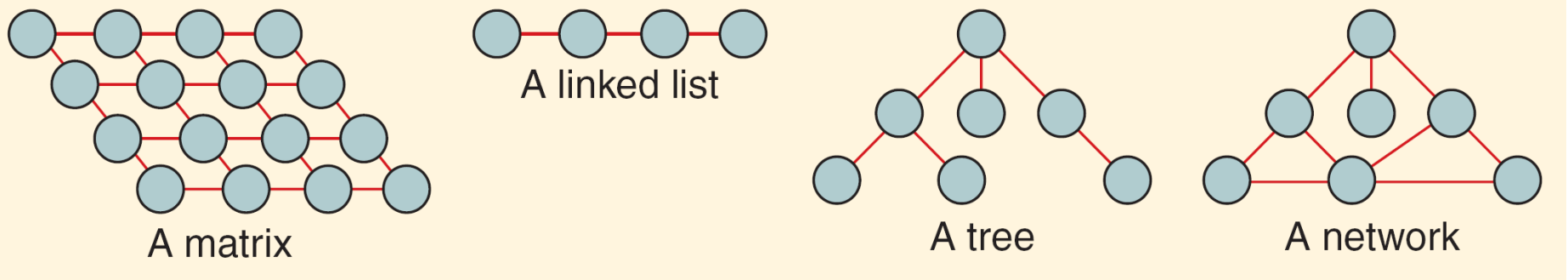
- 1. A combination of elements, each of which is either a data type or another data structure.**
- 2. A set of associations or relationships (structure) involving the combined elements.**

array	struct
A homogeneous combination of data structures.  Position association.	A heterogeneous combination of data structures.  No association.

**Table 15-2**    **Two Structures**

## *Note*

**In the concept of abstraction  
We know what a data type can do.  
How it is done is hidden.**

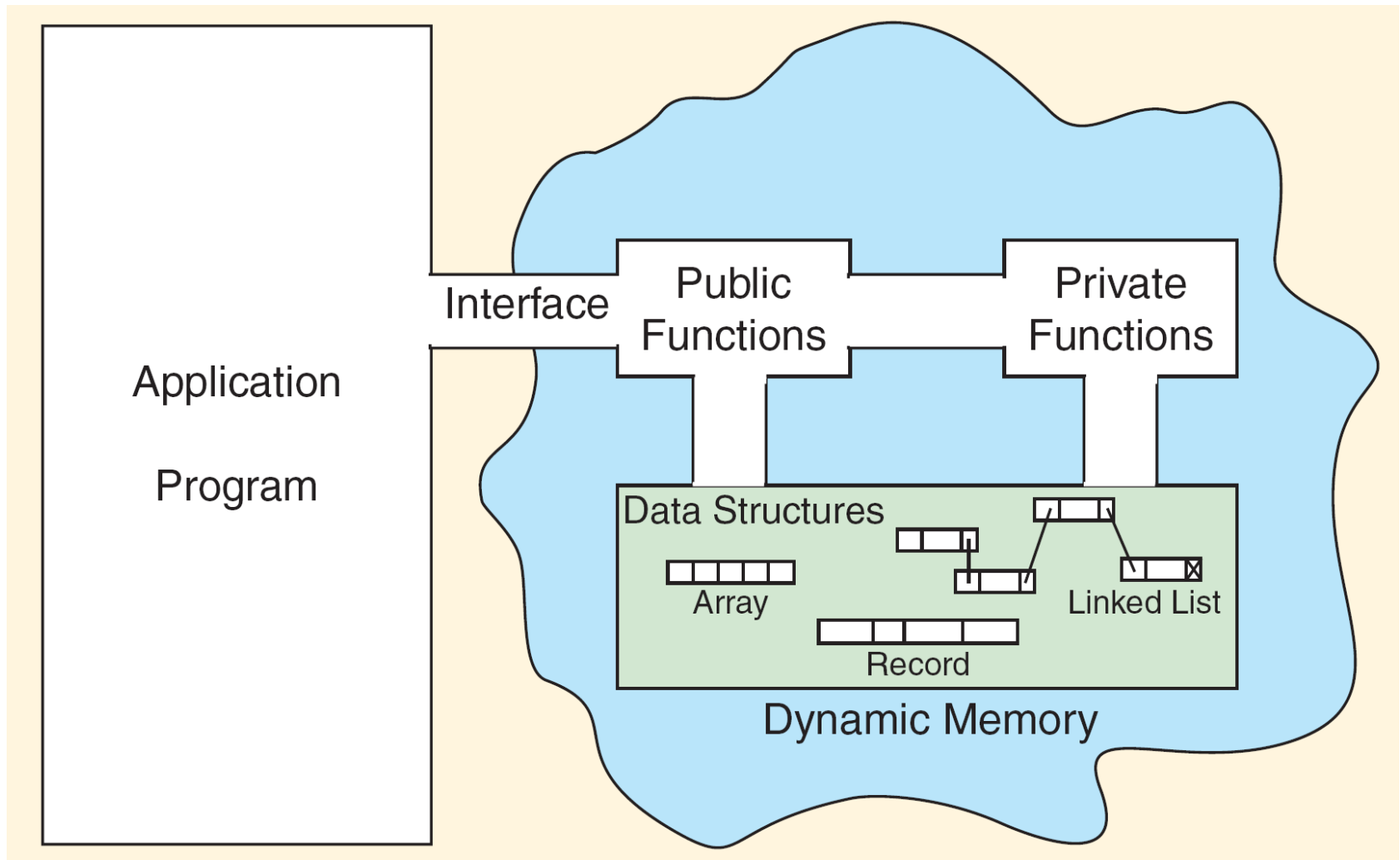


**FIGURE 15-52** Structures for Holding a List

## ***Note***

### **Abstract Data Type**

- 1. Declaration of data.**
- 2. Declaration of operations.**



**FIGURE 15-53** Abstract Data Type Model