MIS 102: Computer Programming

Unit 10 – C Data Structures

Yihuang K. Kang, PhD

“Simplicity and elegance are unpopular because thy require hard work and discipline to achieve and education to be appreciated.”

– *Edsger W. Dijkstra*

Introduction

- We've been discussing that C structure can be used create complex and aggregate data type. This unit introduces popular uses of the structures—dynamic data structures that can grow or shrink at execution time.

- These techniques will enable us to build these data types in a dramatically different manner designed for producing software that's much easier to maintain and reuse.

Introduction(cont.)

- **Linked lists** are collections of data items “lined up in a row”—insertions and deletions are made anywhere in a linked list.
- **Stacks** are important in compilers and operating systems—insertions and deletions are made only at one end of a stack—its top.
- **Queues** represent waiting lines; insertions are made only at the back/tail of a queue and deletions are made only from the front/head of a queue.
- **Binary trees** facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language.

Self-Referential Structure Review

- Recall that a **self-referential structure** contains a pointer member that points to a structure of the same structure type. For example, the definition:

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

defines a type, *struct node*. This structure of type has two members—integer member *data* and pointer member *nextPtr*.

Self-Referential Structure Review(cont.)

- Member *nextPtr* points to a structure of type struct node—a structure of the same type as the one being declared here, hence the term "self-referential structure."

- Member *nextPtr* is referred to as a **link**—i.e., it can be used to "tie" a structure of type struct node to another structure of the same type.

- Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.

Self-Referential Structure Review(cont.)

- Here is an example. Objects linked together to form a list. Note that a slash is placed in this example to represent a NULL. The link of 2nd self-referential structure does not point to another structure.

- A NULL pointer normally indicates the end of a data structure just as the null character indicates the end of a string.

Dynamic Memory Allocation

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

- Functions *malloc*() and *free*(), and operator sizeof, are essential to dynamic memory allocation.

- One disadvantage of using dynamic memory allocation is that it incurs the overhead of function calls.

Dynamic Memory Allocation(cont.)

- Function *malloc*() takes as an argument the number of bytes to be allocated and returns a pointer of type **void *** (pointer to void) to the allocated memory.

- As you recall, a void * pointer may be assigned to a variable of any pointer type.

- Function *malloc*() is normally used with the *sizeof*().

Dynamic Memory Allocation(cont.)

- For example, the statement

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

evaluates *sizeof*(struct node) to determine the size in bytes of a structure of type struct node, allocates a new area in memory of that number of bytes and stores **a pointer to the allocated memory** in variable *newPtr*.

- Note that the allocated memory is not initialized. If no memory is available, *malloc*() returns NULL.

Dynamic Memory Allocation(cont.)

- Function *free*() de-allocates memory—i.e., the memory is returned to the system so that it can be reallocated in the future.
- To free memory dynamically allocated by the preceding *malloc*() call, use the statement

```
free( newPtr );
```

- Remember to free/return allocated memory when it's no longer needed. Otherwise the system may run out of memory prematurely, which is sometimes called "memory leak".

Linked Lists

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

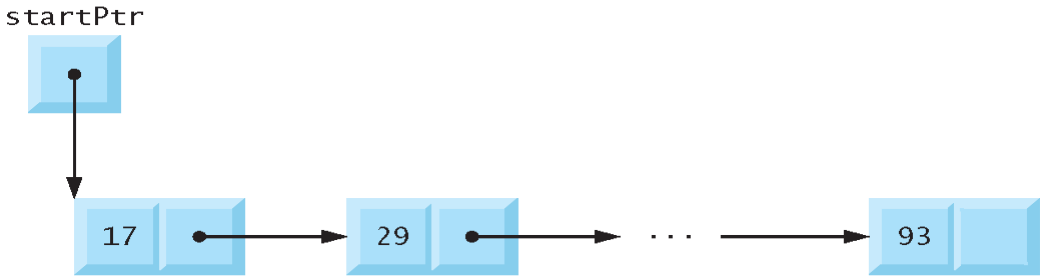
- A linked list is accessed via a pointer to the first node of the list. Subsequent nodes are accessed via the link pointer member stored in each node.

- By convention, the link pointer in the last node of a list is set to NULL to mark the end of the list. Data is stored in a linked list dynamically—each node is created as necessary.

Linked Lists(cont.)

- Lists of data can be stored in arrays, but linked lists provide several advantages. For example, Linked lists are dynamic, so the length of a list can increase or decrease as necessary. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable.

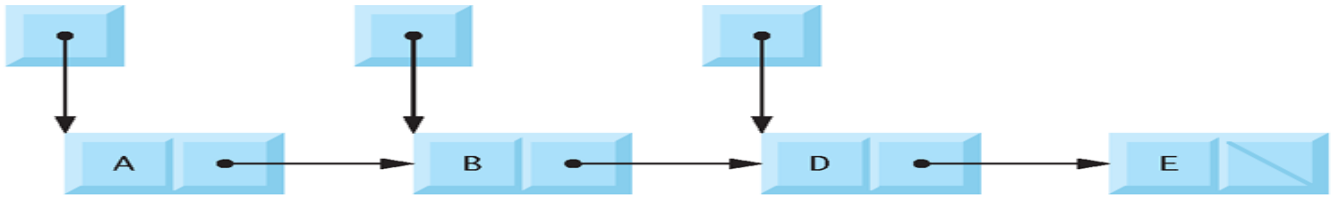
- Unlike arrays, linked-list nodes are normally not stored contiguously in memory. However, the nodes appear to be logically contiguous.



Linked-List Operations

```
1  // Fig. 12.3: fig12_03.c
2  // Inserting and deleting nodes in 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
```

Linked-List Operations – Print List

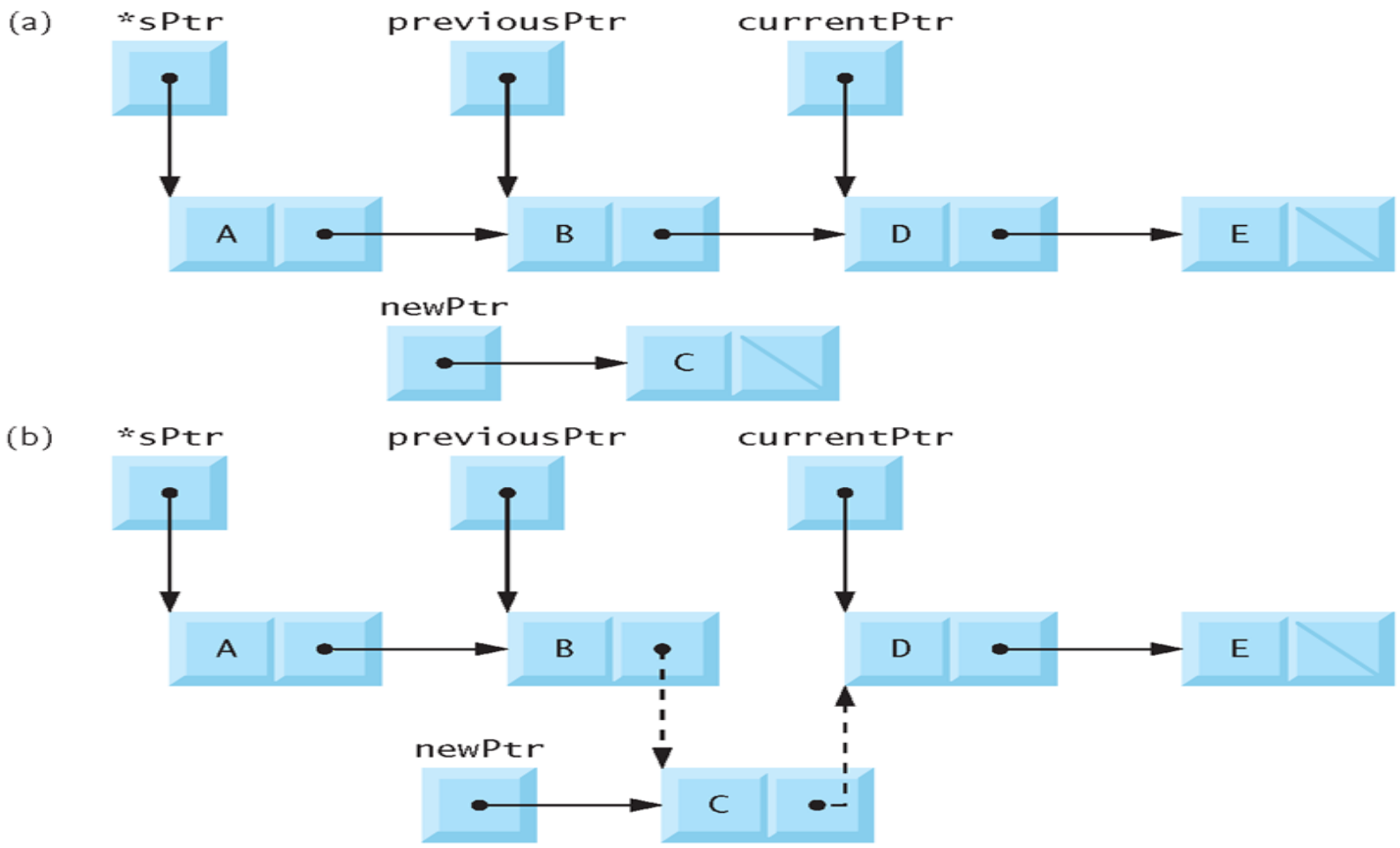



```
163  // print the list
164  void printList( ListNodePtr currentPtr )
165  {
166     // if list is empty
167     if ( isEmpty( currentPtr ) ) {
168        puts( "List is empty.\n" );
169     } // end if
170     else {
171        puts( "The list is:" );
172
173        // while not the end of the list
174        while ( currentPtr != NULL ) {
175           printf( "%c --> ", currentPtr->data );
176           currentPtr = currentPtr->nextPtr;
177        } // end while
178
179        puts( "NULL\n" );
180     } // end else
181  } // end function printList
```

Linked-List Operations – Insertion

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

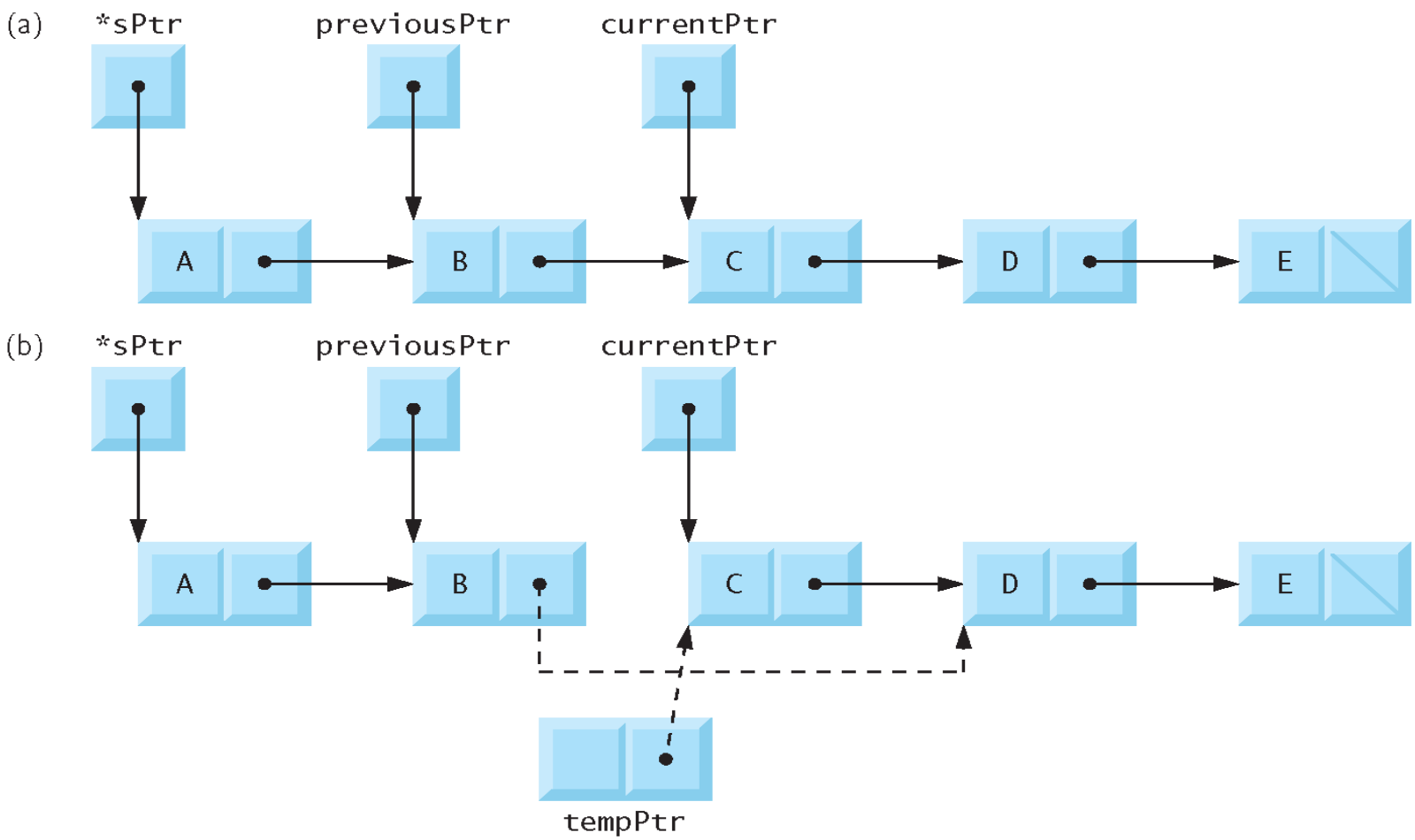
Linked-List Operations – Insertion



Linked-List Operations – Deletion

```
121 // delete a list element
122 char delete( ListNodePtr *sPtr, char value )
123 {
124    ListNodePtr previousPtr; // pointer to previous node in list
125    ListNodePtr currentPtr; // pointer to current node in list
126    ListNodePtr tempPtr; // temporary node pointer
127
128    // delete first node
129    if ( value == ( *sPtr )->data ) {
130       tempPtr = *sPtr; // hold onto node being removed
131       *sPtr = ( *sPtr )->nextPtr; // de-thread the node
132       free( tempPtr ); // free the de-threaded node
133       return value;
134    } // end if
135    else {
136       previousPtr = *sPtr;
137       currentPtr = ( *sPtr )->nextPtr;
138
139       // loop to find the correct location in the list
140       while ( currentPtr != NULL && currentPtr->data != value ) {
141          previousPtr = currentPtr; // walk to ...
142          currentPtr = currentPtr->nextPtr; // ... next node
143       } // end while
144
145        // delete node at currentPtr
146        if ( currentPtr != NULL ) {
147           tempPtr = currentPtr;
148           previousPtr->nextPtr = currentPtr->nextPtr;
149           free( tempPtr );
150           return value;
151        } // end if
152    } // end else
153
154    return '\0';
155 } // end function delete
156
157 // return 1 if the list is empty, 0 otherwise
158 int isEmpty( ListNodePtr sPtr )
159 {
160    return sPtr == NULL;
161 } // end function isEmpty
162
```

Linked-List Operations – Deletion



Linked-List Operation(cont.)

```
22  int main( void )
23  {
24     ListNodePtr startPtr = NULL; // initially there are no nodes
25     unsigned int choice; // user's choice
26     char item; // char entered by user
27
28     instructions(); // display the menu
29     printf( "%s", "? " );
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 );
```

Linked-List Operation(cont.)

```
47
48                   // if character is found, remove it
49                   if ( delete( &startPtr, item ) ) { // remove item
50                      printf( "%c deleted.\n", item );
51                      printList( startPtr );
52                   } // end if
53                   else {
54                      printf( "%c not found.\n\n", item );
55                   } // end else
56                } // end if
57                else {
58                   puts( "List is empty.\n" );
59                } // end else
60
61                break;
62             default:
63                puts( "Invalid choice.\n" );
64                instructions();
65                break;
66          } // end switch
67
68          printf( "%s", "? " );
69          scanf( "%u", &choice );
70       } // end while
71
```

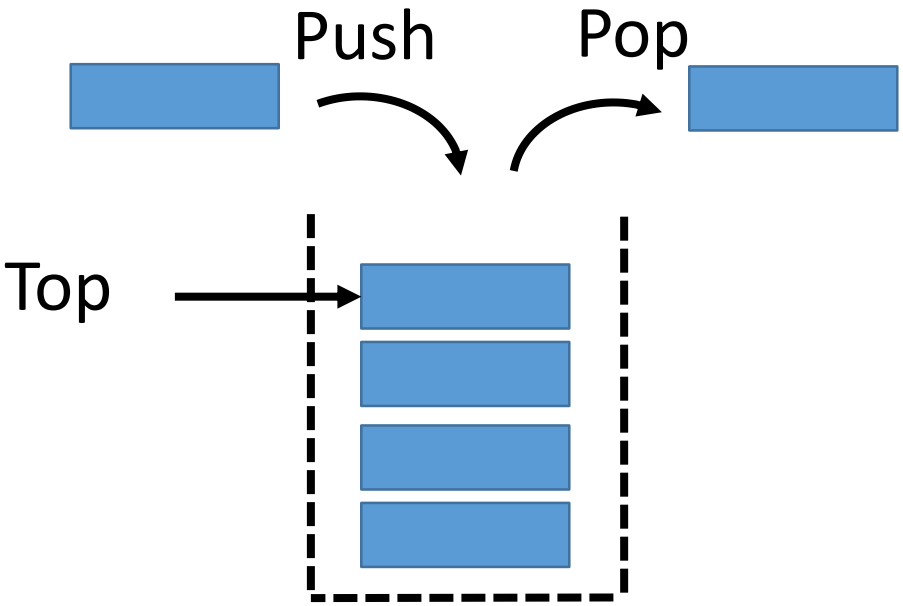
Linked-List Operation(cont.)

```
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

? 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.
```

Stacks

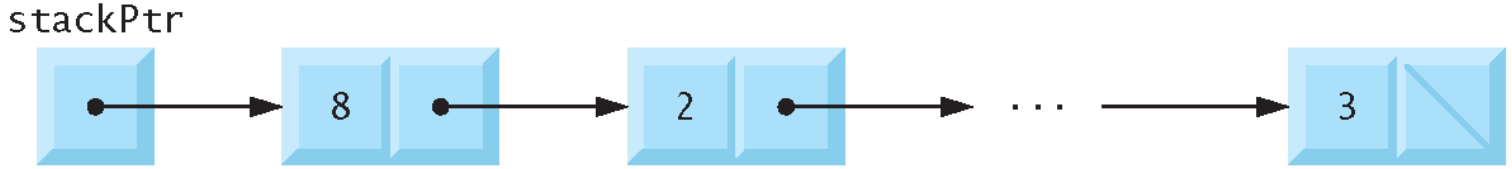
- A **Stack** can be considered a constrained version of the Linked list, as new nodes can only be added/removed from the **top** of a stack. For this reason, a stack is referred to as a **last-in, first-out** (**LIFO**) data structure.



Stacks(cont.)

- A stack is referenced via **a pointer to the top** element of the stack. The link member in the last node of the stack is set to NULL to indicate the bottom of the stack.

- Stacks and linked lists are represented identically. The difference between stacks and linked lists is that insertions and deletions may occur anywhere in a linked list, but only at the top of a stack.



Stacks(cont.)

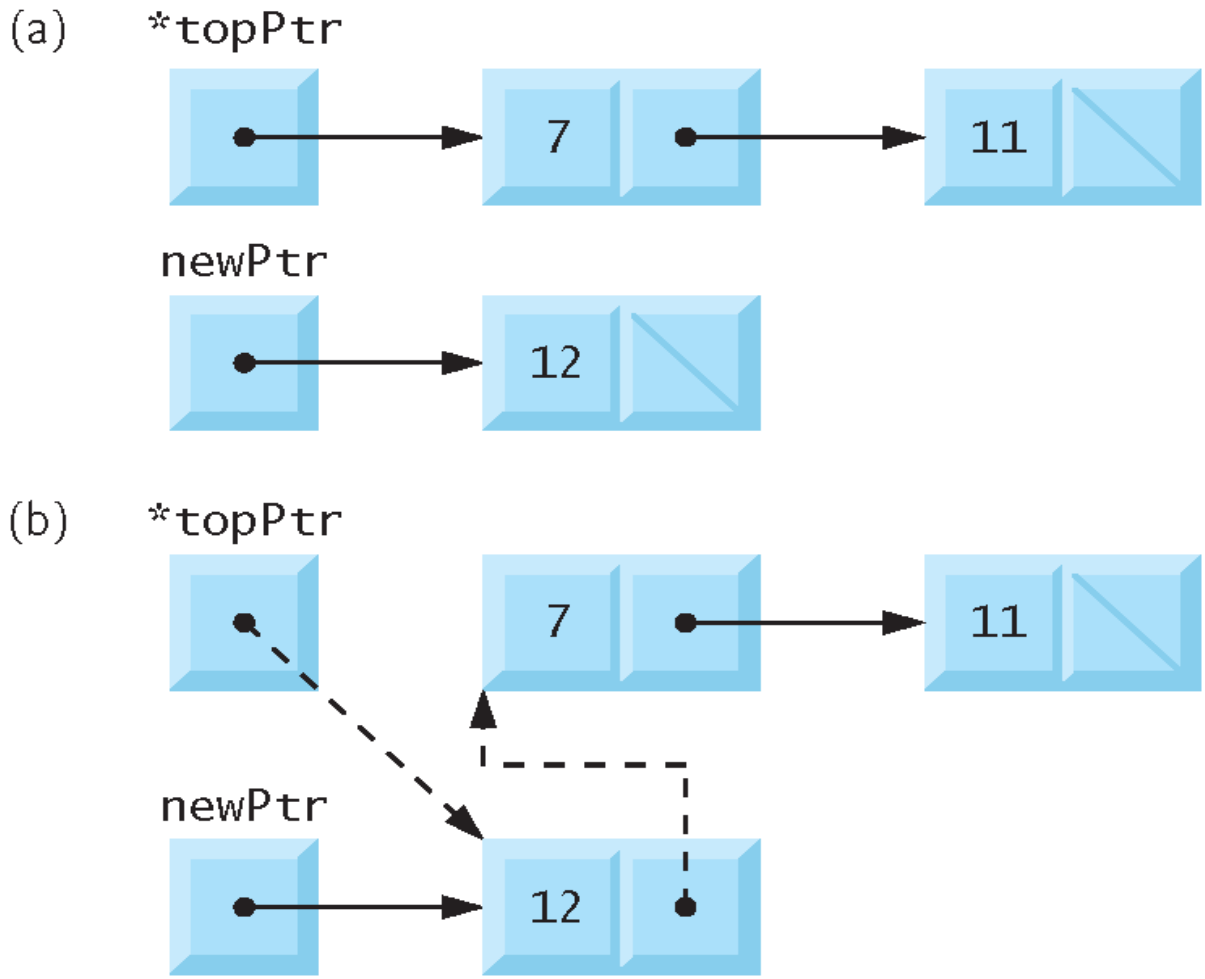
```
1  // Fig. 12.8: fig12_08.c
2  // A simple stack program
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  // self-referential structure
7  struct stackNode {
8     int data; // define data as an int
9     struct stackNode *nextPtr; // stackNode pointer
10 }; // end structure stackNode
11
12 typedef 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 );
```

Stacks(cont.)

- Function ***push*()** creates a new node and places it on top of the stack.

```
75  // insert a node at the stack top
76  void push( StackNodePtr *topPtr, int info )
77  {
78     StackNodePtr newPtr; // pointer to new node
79
80     newPtr = malloc( sizeof( StackNode ) );
81
82     // insert the node at stack top
83     if ( newPtr != NULL ) {
84        newPtr->data = info;
85        newPtr->nextPtr = *topPtr;
86        *topPtr = newPtr;
87     } // end if
88     else { // no space available
89        printf( "%d not inserted. No memory available.\n", info );
90     } // end else
91  } // end function push
```

Stacks(cont.)

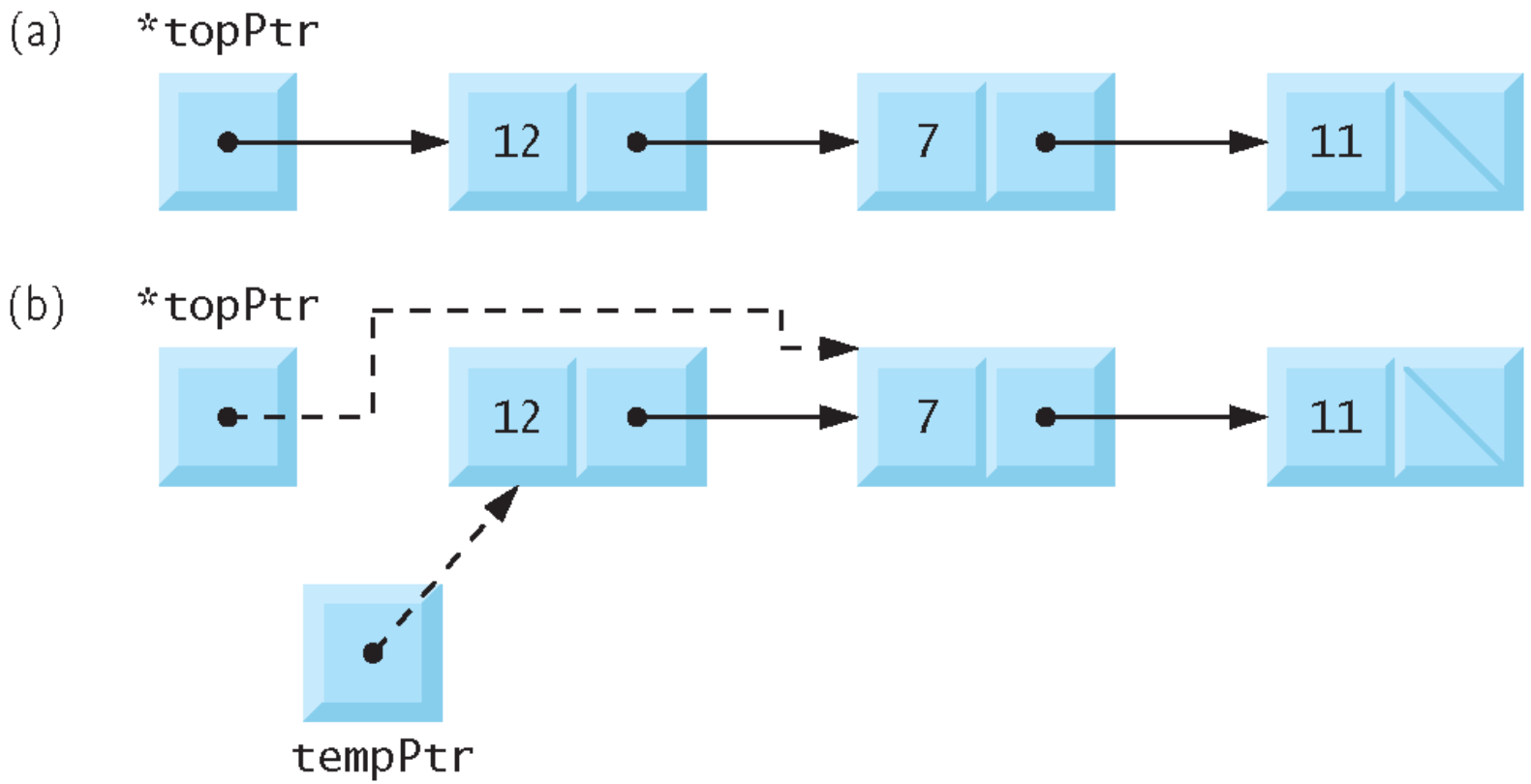


Stacks(cont.)

- Function ***pop()*** removes a node from the top of the stack, frees the memory that was allocated to the popped node and returns the popped value.

```
93   // remove a node from the stack top
94   int pop( StackNodePtr *topPtr )
95   {
96      StackNodePtr tempPtr; // temporary node pointer
97      int popValue; // node value
98
99      tempPtr = *topPtr;
100     popValue = ( *topPtr )->data;
101     *topPtr = ( *topPtr )->nextPtr;
102     free( tempPtr );
103     return popValue;
104  } // end function pop
```

Stacks(cont.)



Stacks(cont.)

```
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

? 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.
```

Application of Stacks

- Stacks have many interesting applications. For example, whenever a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack.

- If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each function can return to its caller.

- When the function returns to its caller, the space for that function's automatic variables is popped off the stack, and these variables no longer are known to the program.

Queues

- Another common data structure is the **queue**. A queue is similar to a checkout line in a store—the first person in line is serviced first, and other customers enter the line only at the end and wait to be serviced.

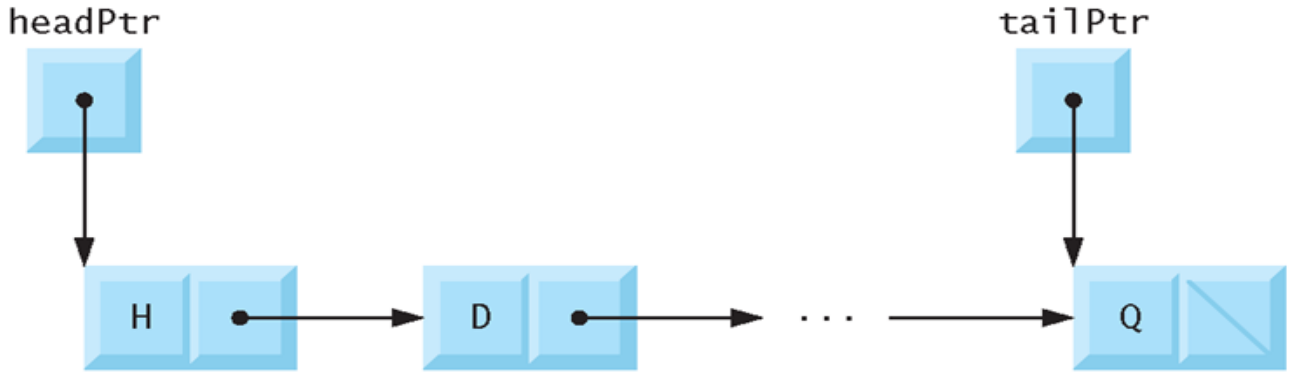
- Queue nodes are removed only from the **head** of the queue and are inserted only at the **tail** of the queue. For this reason, a queue is referred to as a **first-in, first-out** (**FIFO**) data structure.

The insert and remove operations are known as **enqueue** and **dequeue**, respectively.

Queues(cont.)

- Queues have many applications in computer systems. For example, queues are used to support print spooling. If a printer is busy, other outputs may still be generated. These are spooled to disk where they wait in a queue until the printer becomes available.

- Here is a graphical representation of Queues.



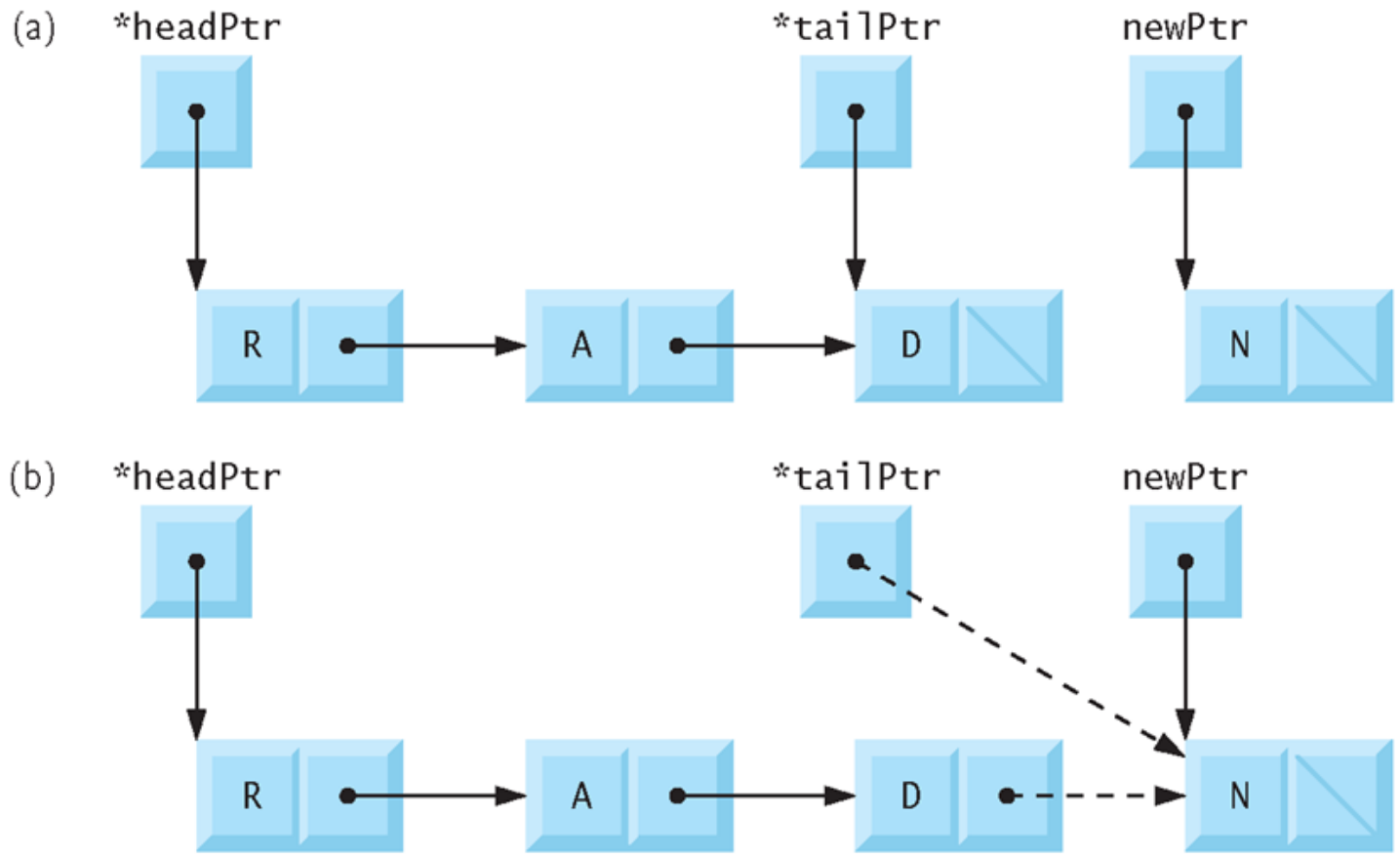
Queues(cont.)

```
1  // Fig. 12.13: fig12_13.c
2  // Operating and maintaining a queue
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  // self-referential structure
7  struct queueNode {
8     char data; // define data as a char
9     struct queueNode *nextPtr; // queueNode pointer
10 }; // end structure queueNode
11
12 typedef struct queueNode QueueNode;
13 typedef QueueNode *QueueNodePtr;
14
15 // function prototypes
16 void printQueue( QueueNodePtr currentPtr );
17 int isEmpty( QueueNodePtr headPtr );
18 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
19 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
20    char value );
21 void instructions( void );
22
```

Queues(cont.)

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

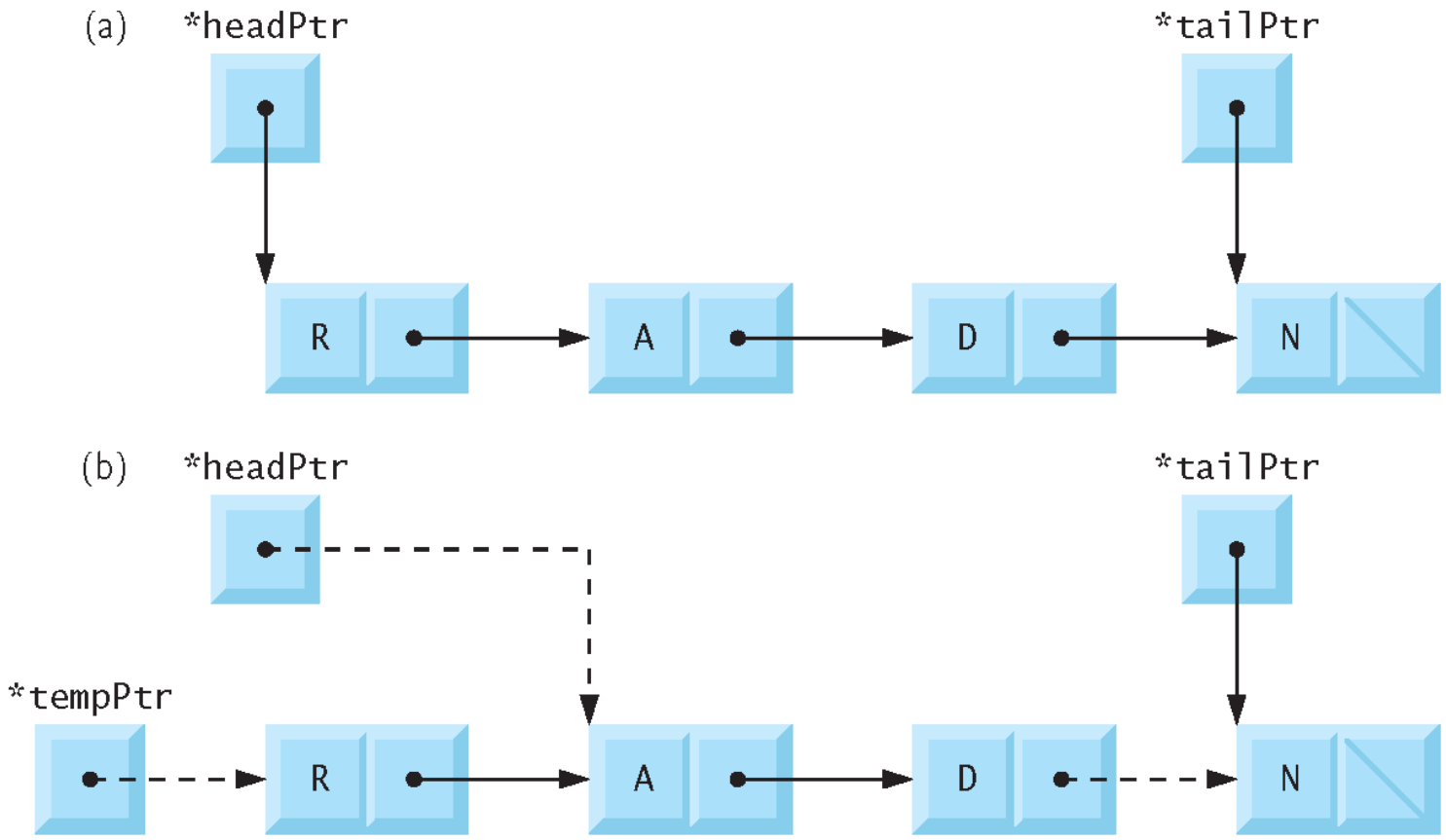
Queues(cont.)


(a)
*headPtr
*tailPtr
newPtr
R
A
D
N
(b)
*headPtr
*tailPtr
newPtr
R
A
D
N

Queues(cont.)

```
104
105 // remove node from queue head
106 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
107 {
108    char value; // node value
109    QueueNodePtr tempPtr; // temporary node pointer
110
111    value = ( *headPtr )->data;
112    tempPtr = *headPtr;
113    *headPtr = ( *headPtr )->nextPtr;
114
115    // if queue is empty
116    if ( *headPtr == NULL ) {
117       *tailPtr = NULL;
118    } // end if
119
120    free( tempPtr );
121    return value;
122 } // end function dequeue
123
124 // return 1 if the queue is empty, 0 otherwise
125 int isEmpty( QueueNodePtr headPtr )
126 {
127    return headPtr == NULL;
128 } // end function isEmpty
```

Queues(cont.)



Queues(cont.)

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

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

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

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

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

? 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.
```

Trees

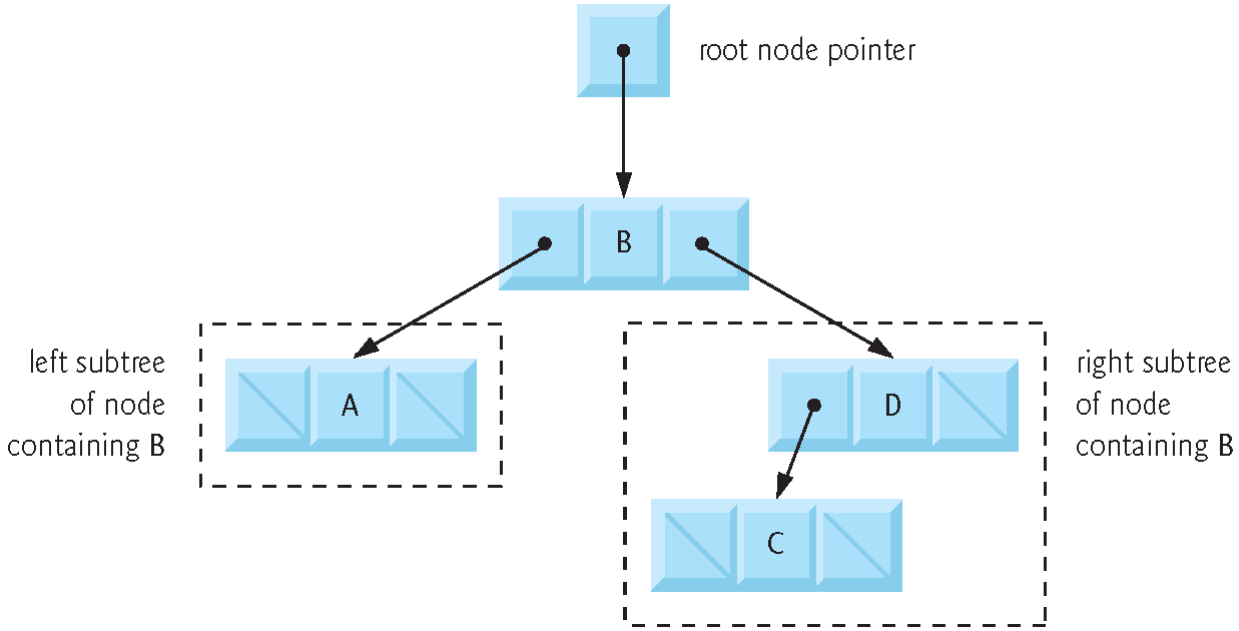
- Linked lists, stacks and queues are **linear data structures**. A **tree** is a nonlinear, two-dimensional data structure with special properties.

- A Tree node contains two or more links. Here, we only discuss **binary search trees**—trees whose nodes all contain two links (none, one, or both of which may be NULL).

- Computer scientists normally draw trees from the root node down—exactly the opposite of trees in nature.

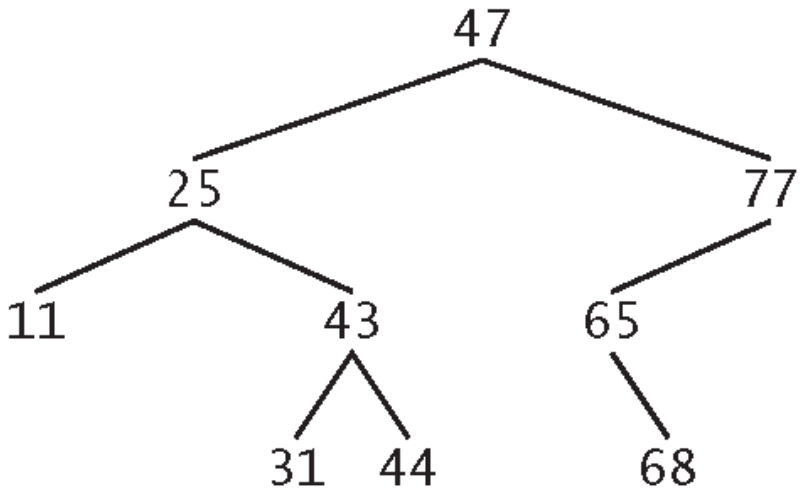
Trees(cont.)

- The **root** node is the first node in a tree. Each link in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree. The children of a node are called **siblings**. A node with no children is called a **leaf** node.



Trees(cont.)

- A **binary search tree (BST)** 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*. Below is an example of the tree.

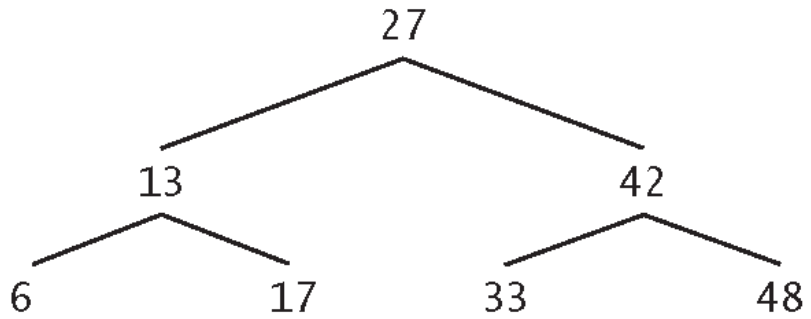


Trees(cont.)

- The following example shows how to create BST and traverse it in three ways—**inOrder**, **preOrder** and **postOrder**.

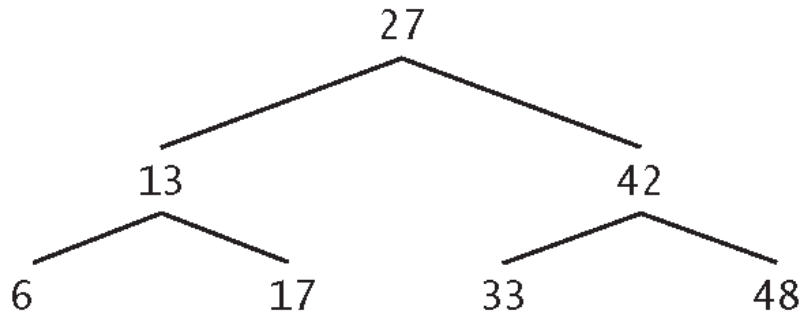
- The below BST is built from the following sequence of numbers.

27, 13, 42, 6, 17, 33, 48



Trees(cont.)

- The steps for an **inOrder** traversal of below BST are: traverse the left subtree inOrder (L), process the data value in the node (D), and then traverse the right subtree inOrder (R).

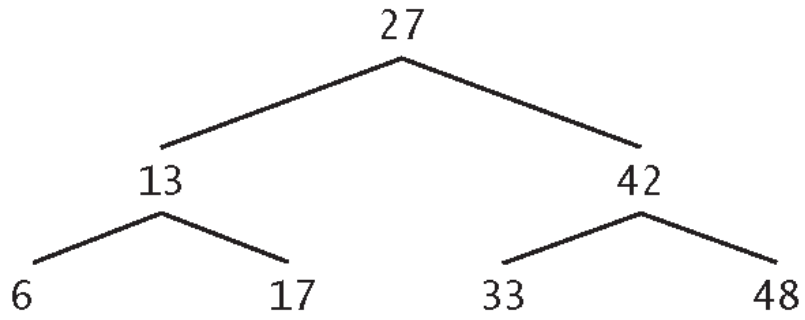


- So the inOrder (LDR) traversal of the tree is:

6 13 17 27 33 42 48

Trees(cont.)

- The steps for a **preOrder** traversal of below BST are: process the value in the node (D), traverse the left subtree preorder (L), and then traverse the right subtree preorder (R).

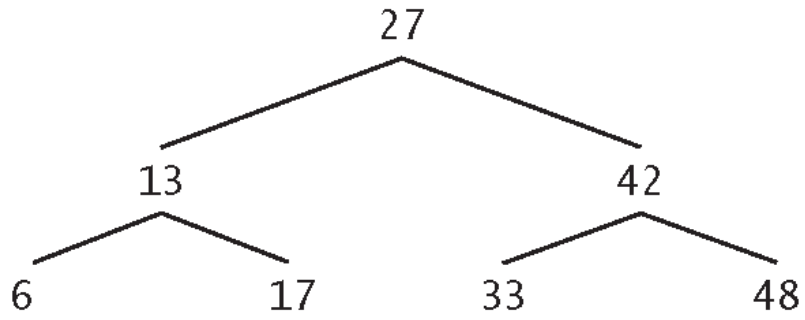


- So the preOrder (DLR) traversal of the tree is:

27 13 6 17 42 33 48

Trees(cont.)

- The steps for a **postOrder** traversal of below BST are: traverse the left subtree postOrder (L), traverse the right subtree postOrder (R), and then process the value in the node (D).



- So the postOrder (LRD) traversal of the tree is:

6 17 13 33 48 42 27

Binary Tree Search

- You may notice an advantage of using BST—once the binary search tree has been created, its elements can be retrieved by recursively in-order traversing the tree. It allows fast lookup as it keep the values in sorted order, so that lookup and other operations can use the principle of **binary search.**

- An BST with ***n*** elements would have a maximum of **log2(n**) levels, and thus a maximum of **log2(n**) comparisons would have to be made either to find a match or to determine that no match exists. It means, for example, that when searching a 1,000,000-element binary search tree, no more than 20 comparisons need to be made because $2^{20} > 1{,}000{,}000$.

Tree Implementation

```
1  // Fig. 12.19: fig12_19.c
2  // Creating and traversing a binary tree
3  // preorder, inorder, and postorder
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  // self-referential structure
9  struct treeNode {
10    struct treeNode *leftPtr; // pointer to left subtree
11    int data; // node value
12    struct treeNode *rightPtr; // pointer to right subtree
13 }; // 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 );
```

Tree Implementation-Insert Node

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

Tree Implementation-Traversal

```
87  // begin inorder traversal of tree
88  void inOrder( TreeNodePtr treePtr )
89  {
90     // if tree is not empty, then traverse
91     if ( treePtr != NULL ) {
92        inOrder( treePtr->leftPtr );
93        printf( "%3d", treePtr->data );
94        inOrder( treePtr->rightPtr );
95     } // end if
96  } // end function inOrder
```

```
98  // begin preorder traversal of tree
99  void preOrder( TreeNodePtr treePtr )
100 {
101    // if tree is not empty, then traverse
102    if ( treePtr != NULL ) {
103       printf( "%3d", treePtr->data );
104       preOrder( treePtr->leftPtr );
105       preOrder( treePtr->rightPtr );
106    } // end if
107 } // end function preOrder
108
109 // begin postorder traversal of tree
110 void postOrder( TreeNodePtr treePtr )
111 {
112    // if tree is not empty, then traverse
113    if ( treePtr != NULL ) {
114       postOrder( treePtr->leftPtr );
115       postOrder( treePtr->rightPtr );
116       printf( "%3d", treePtr->data );
117    } // end if
118 } // end function postOrder
```

Tree Implementation(cont.)

```
25  int main( void )
26  {
27     unsigned int i; // counter to loop from 1-10
28     int item; // variable to hold random values
29     TreeNodePtr rootPtr = NULL; // tree initially empty
30
31     srand( time( NULL ) );
32     puts( "The numbers being placed in the tree are:" );
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     puts( "\n\nThe preOrder traversal is:" );
43     preOrder( rootPtr );
44
45     // traverse the tree inOrder
46     puts( "\n\nThe inOrder traversal is:" );
47     inOrder( rootPtr );
48
49      // traverse the tree postOrder
50      puts( "\n\nThe postOrder traversal
51      postOrder( rootPtr );
52  } // end main
```

Tree Implementation(cont.)

```
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
                 11 12
```

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

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