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.)

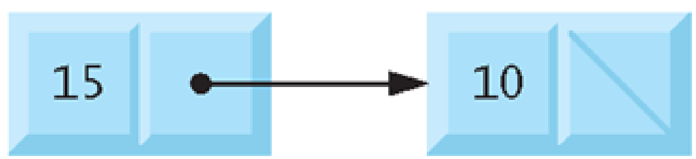
- Member *nextPtr* points to a structure of type struct node—a structure of the same type as the one being declared here, hence the term “self-referential structure.”

- 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.

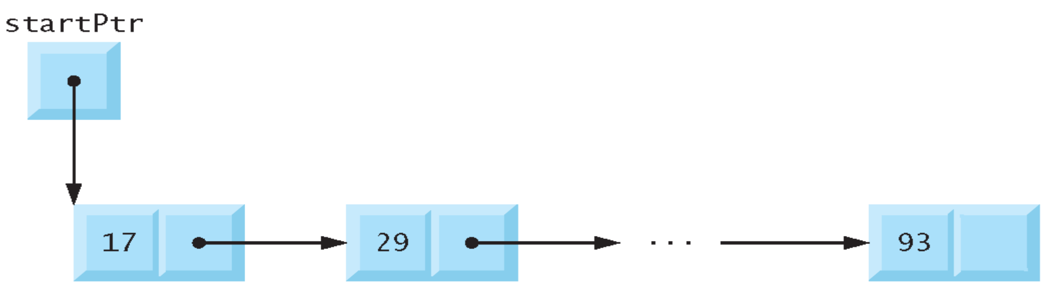
- A linked list is accessed via a pointer to the first node of the list. Subsequent nodes are accessed via the link pointer member stored in each node.

- 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.

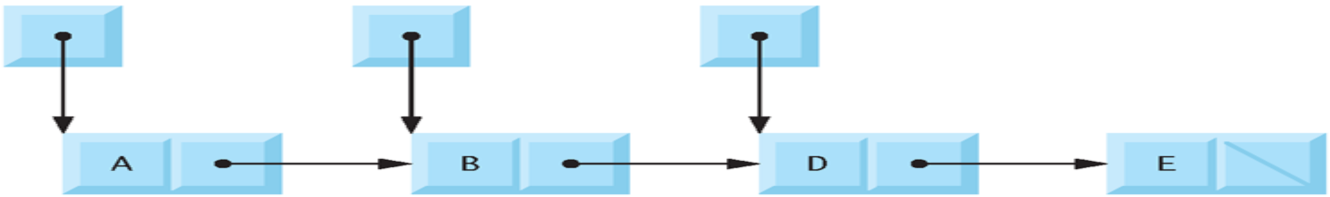
- Unlike arrays, linked-list nodes are normally not stored contiguously in memory. However, the nodes appear to be logically contiguous.



Linked-List Operations

```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
}; // end structure listNode

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 );

```

Linked-List Operations – Print List

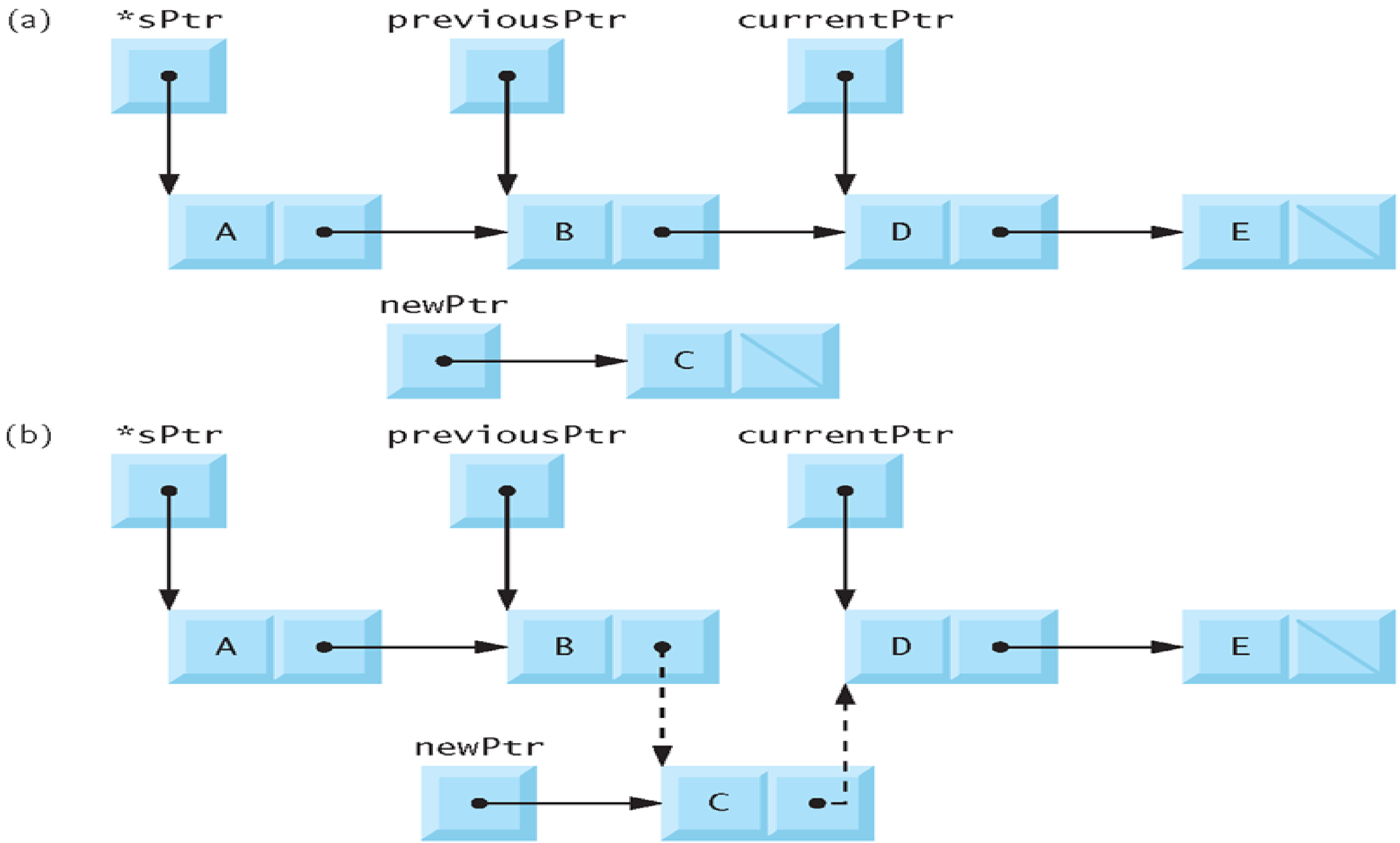



```c
// print the list
void printList( ListNodePtr currentPtr )
{
    // if list is empty
    if ( isEmpty( currentPtr ) ) {
        puts( "List is empty.\n" );
    } // end if
    else {
        puts( "The list is:" );

        // while not the end of the list
        while ( currentPtr != NULL ) {
            printf( "%c --> ", currentPtr->data );
            currentPtr = currentPtr->nextPtr;
        } // end while

        puts( "NULL\n" );
    } // end else
} // end function printList
```

Linked-List Operations – Insertion

```c
// insert a new value into the list in sorted order
void insert( ListNodePtr *sPtr, char value )
{
   ListNodePtr newPtr; // pointer to new node
   ListNodePtr previousPtr; // pointer to previous node in list
   ListNodePtr currentPtr; // pointer to current node in list

   newPtr = malloc( sizeof( ListNode ) ); // create node

   if ( newPtr != NULL ) { // is space available
      newPtr->data = value; // place value in node
      newPtr->nextPtr = NULL; // node does not link to another node

      previousPtr = NULL;
      currentPtr = *sPtr;

      // loop to find the correct location in the list
      while ( currentPtr != NULL && value > currentPtr->data ) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      } // end while

      // insert new node at beginning of list
      if ( previousPtr == NULL ) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      } // end if
      else { // insert new node between previousPtr and currentPtr
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      } // end else
   } // end if
   else {
      printf( "%c not inserted. No memory available.\n", value );
   } // end else
} // end function insert
```

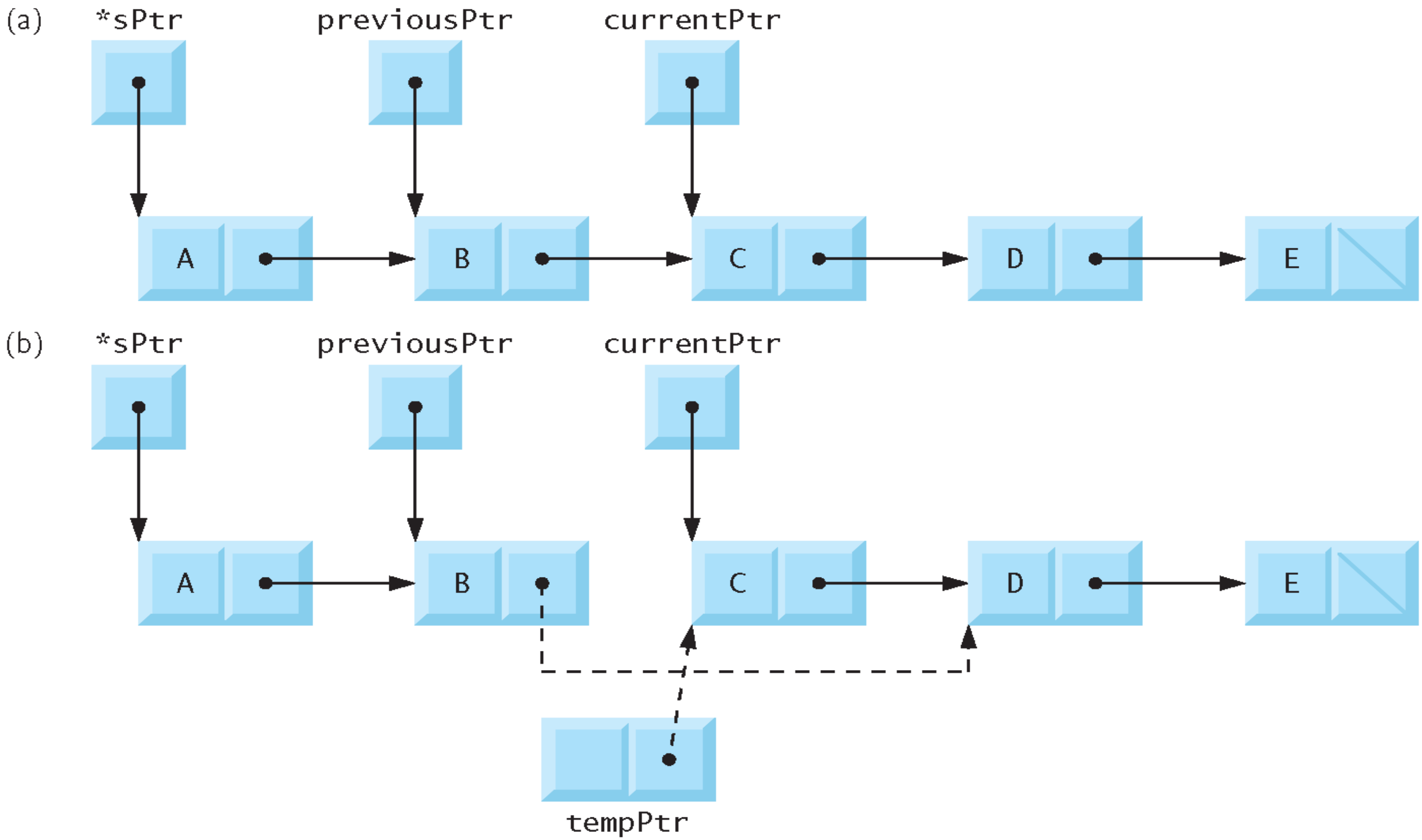
Linked-List Operations – Insertion



Linked-List Operations – Deletion

```
// delete a list element
char delete( ListNodePtr *sPtr, char value )
{
   ListNodePtr previousPtr; // pointer to previous node in list
   ListNodePtr currentPtr; // pointer to current node in list
   ListNodePtr tempPtr; // temporary node pointer

   // delete first node
   if ( value == ( *sPtr )->data ) {
      tempPtr = *sPtr; // hold onto node being removed
      *sPtr = ( *sPtr )->nextPtr; // de-thread the node
      free( tempPtr ); // free the de-threaded node
      return value;
   } // end if
   else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      // loop to find the correct location in the list
      while ( currentPtr != NULL && currentPtr->data != value ) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      } // end while

      // delete node at currentPtr
      if ( currentPtr != NULL ) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return value;
      } // end if
   } // end else

   return '\0';
} // end function delete

// return 1 if the list is empty, 0 otherwise
int isEmpty( ListNodePtr sPtr )
{
   return sPtr == NULL;
} // end function isEmpty
```

Linked-List Operations – Deletion



Linked-List Operation(cont.)

```c
int main( void )
{
   ListNodePtr startPtr = NULL; // initially there are no nodes
   unsigned int choice; // user's choice
   char item; // char entered by user

   instructions(); // display the menu
   printf( "%s", "? " );
   scanf( "%u", &choice );

   // loop while user does not choose 3
   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "%s", "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item ); // insert item in list
            printList( startPtr );
            break;
         case 2: // delete an element
            // if list is not empty
            if ( !isEmpty( startPtr ) ) {
               printf( "%s", "Enter character to be deleted: " );
               scanf( "\n%c", &item );
```

Linked-List Operation(cont.)

```
                    // if character is found, remove it
                    if ( delete( &startPtr, item ) ) { // remove item
                       printf( "%c deleted.\n", item );
                       printList( startPtr );
                    } // end if
                    else {
                       printf( "%c not found.\n\n", item );
                    } // end else
                 } // end if
                 else {
                    puts( "List is empty.\n" );
                 } // end else

                 break;
              default:
                 puts( "Invalid choice.\n" );
                 instructions();
                 break;
           } // end switch

           printf( "%s", "? " );
           scanf( "%u", &choice );
        } // end while

```

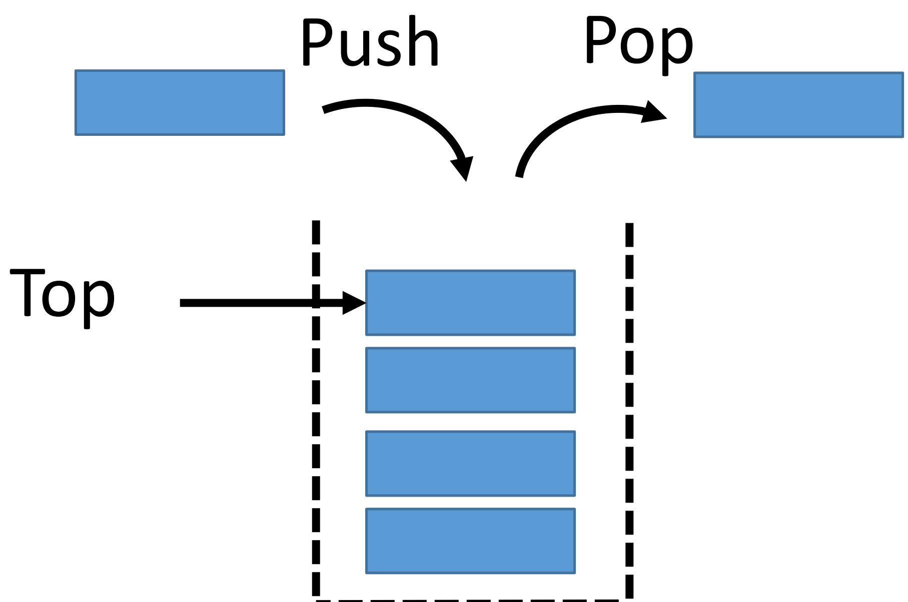
Linked-List Operation(cont.)

```
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

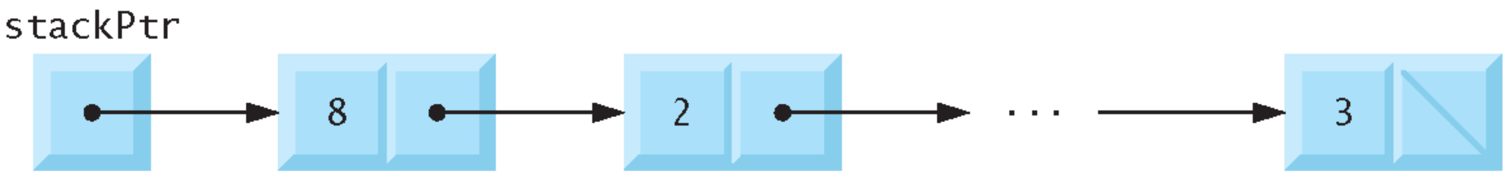
- A **Stack** can be considered a constrained version of the Linked list, as new nodes can only be added/removed from the **top** of a stack. For this reason, a stack is referred to as a **last-in, first-out** (**LIFO**) data structure.



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.)

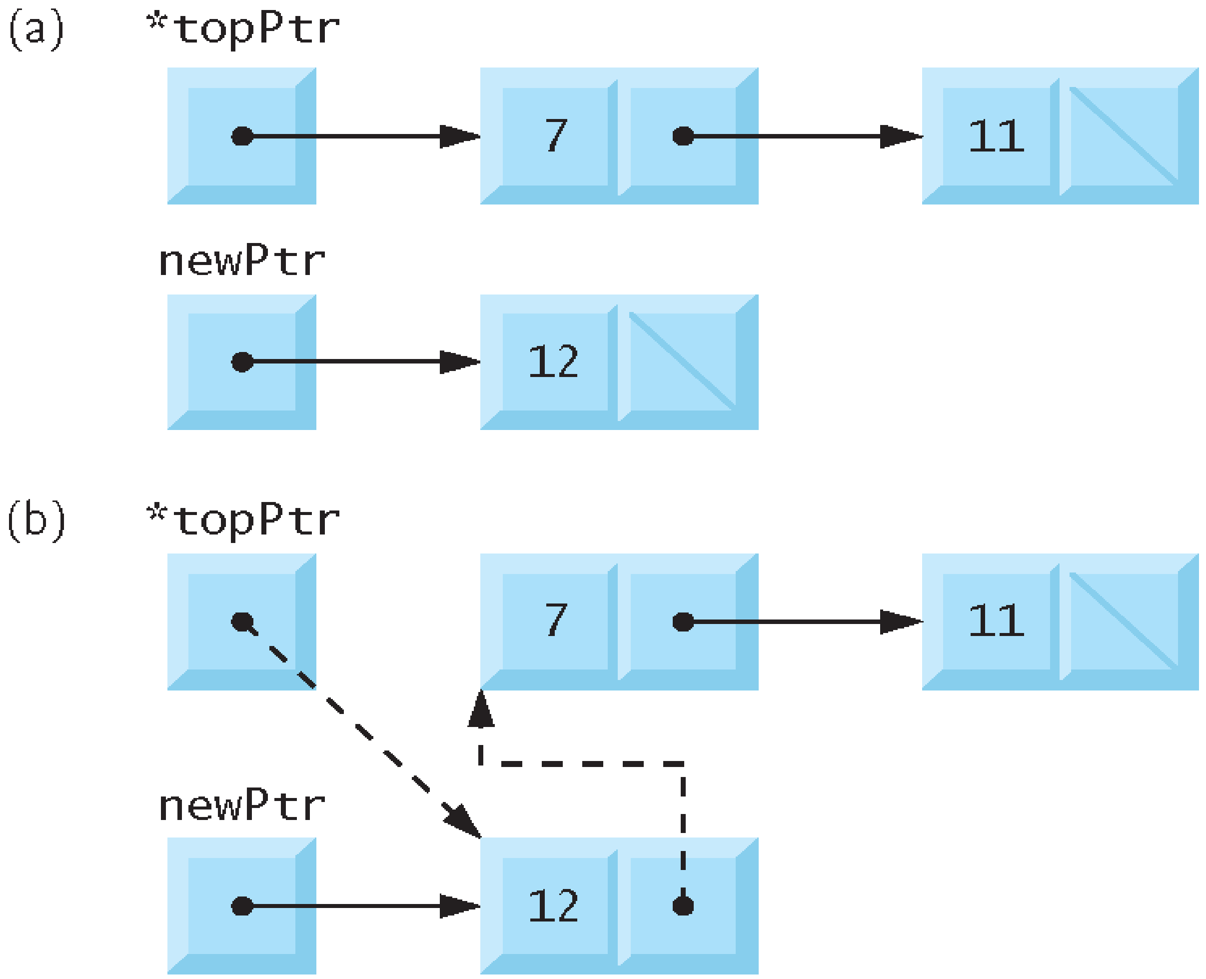
```c
// Fig. 12.8: fig12_08.c
// A simple stack program
#include <stdio.h>
#include <stdlib.h>

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

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 );
```

Stacks(cont.)

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

```c
// insert a node at the stack top
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr; // pointer to new node

   newPtr = malloc( sizeof( StackNode ) );

   // insert the node at stack top
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   } // end if
   else { // no space available
      printf( "%d not inserted. No memory available.\n", info );
   } // end else
} // end function push
```

Stacks(cont.)

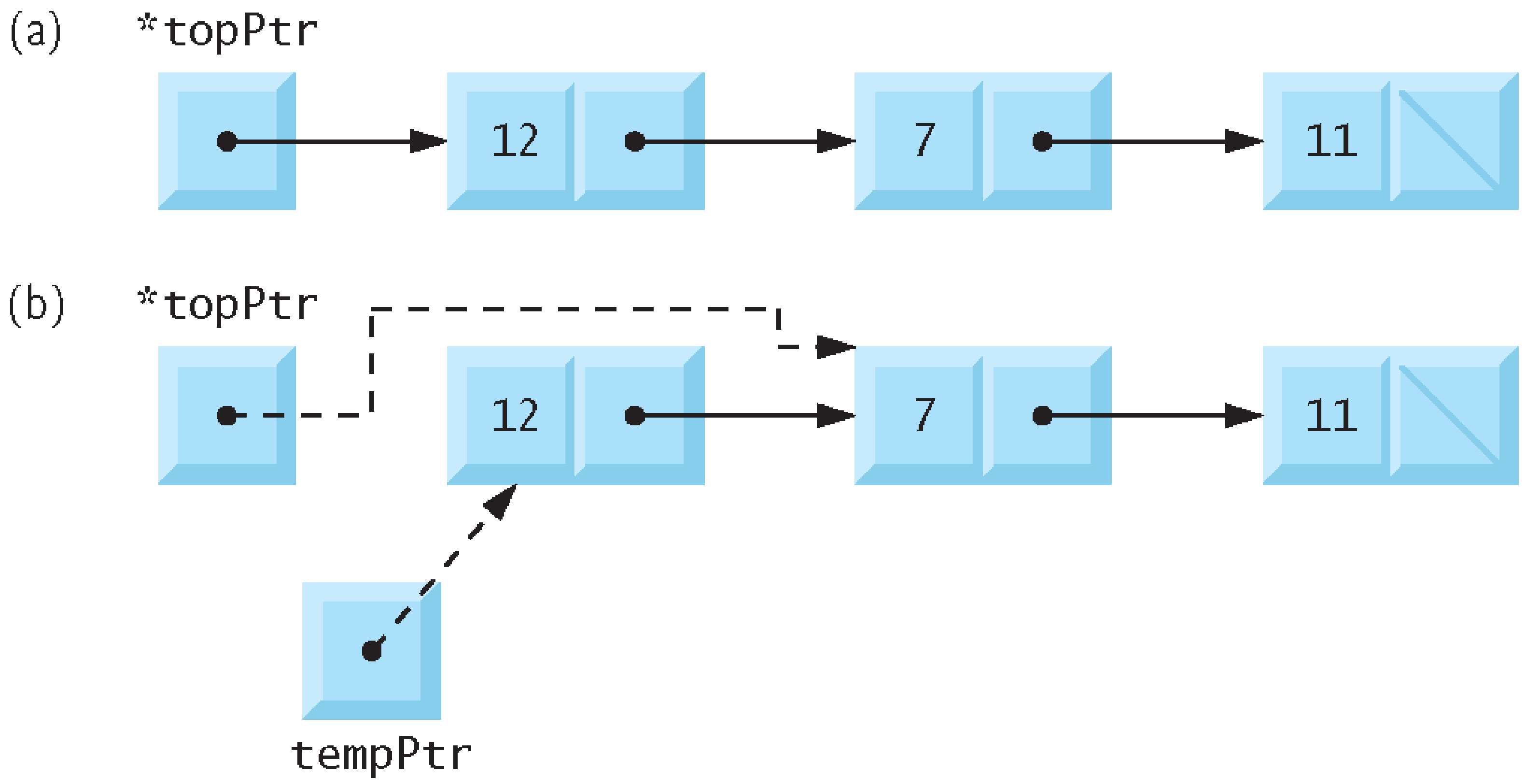


Stacks (cont.)

- 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.

```c
// remove a node from the stack top
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr; // temporary node pointer
   int popValue; // node value

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );
   return popValue;
} // 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.

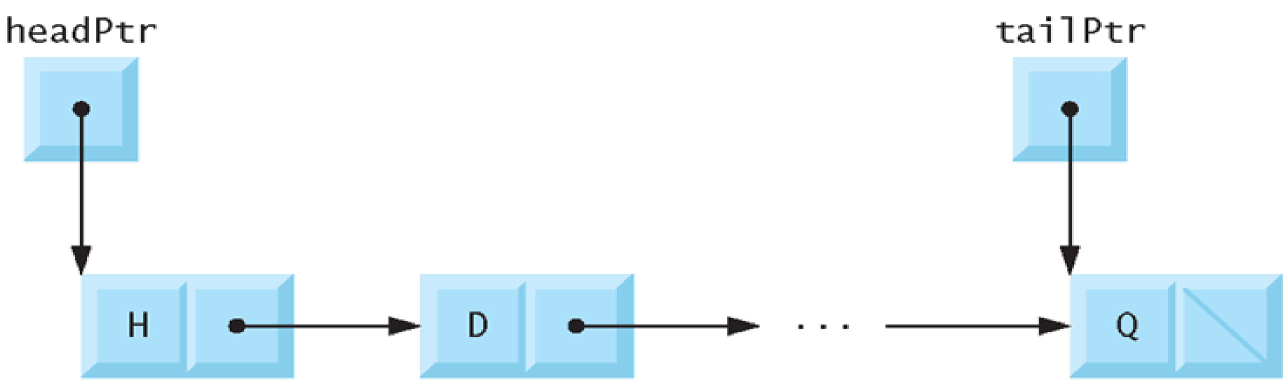
- Queue nodes are removed only from the **head** of the queue and are inserted only at the **tail** of the queue. For this reason, a queue is referred to as a **first-in, first-out** (**FIFO**) data structure.

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.



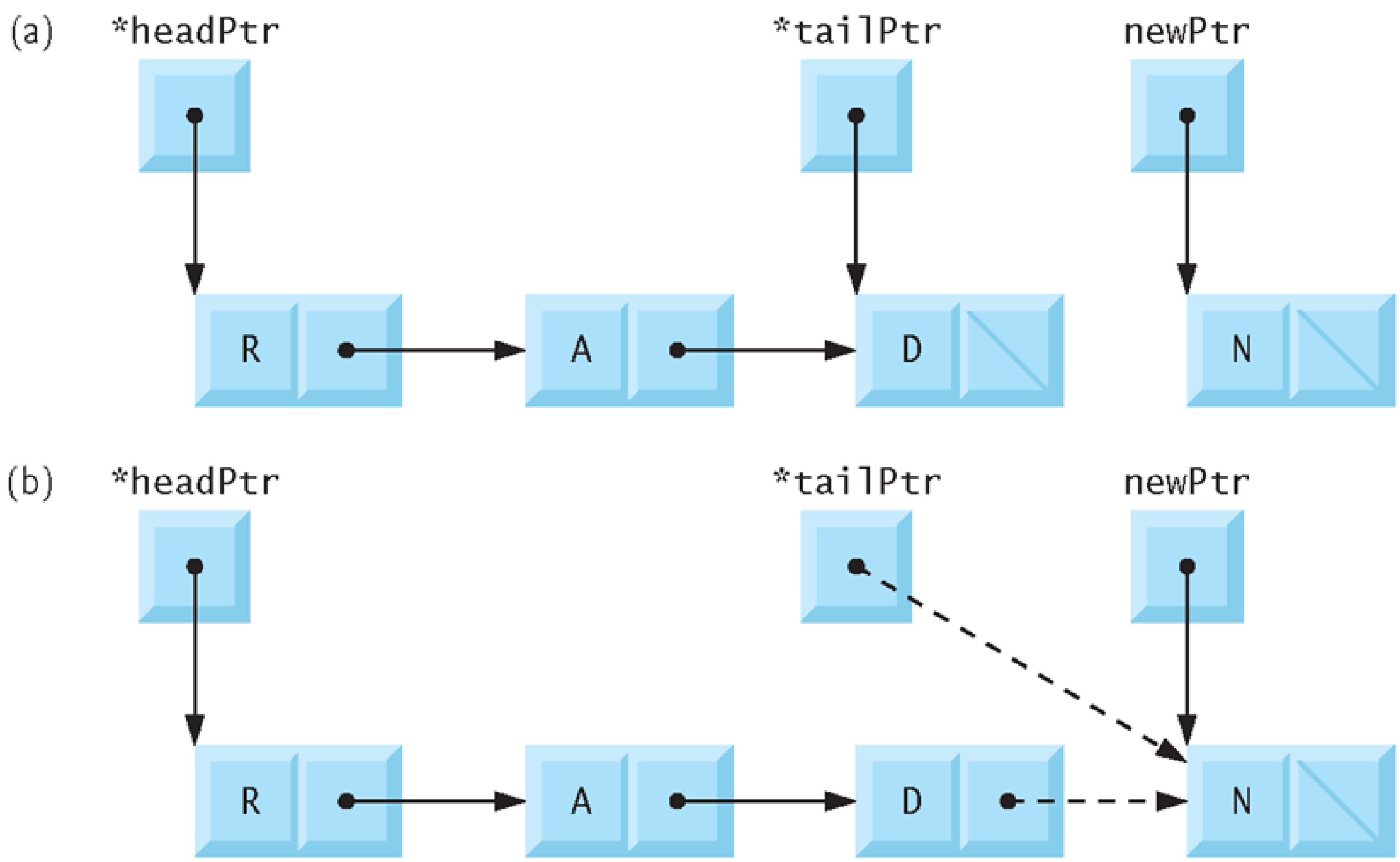
Queues(cont.)

```
// 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
}; // end structure queueNode

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 );

```

Queues(cont.)

```
// insert a node in at queue tail
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
   char value )
{
   QueueNodePtr newPtr; // pointer to new node

   newPtr = malloc( sizeof( QueueNode ) );

   if ( newPtr != NULL ) { // is space available
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      // if empty, insert node at head
      if ( isEmpty( *headPtr ) ) {
         *headPtr = newPtr;
      } // end if
      else {
         ( *tailPtr )->nextPtr = newPtr;
      } // end else

      *tailPtr = newPtr;
   } // end if
   else {
      printf( "%c not inserted. No memory available.\n", value );
   } // end else
} // end function enqueue
```

Queues(cont.)

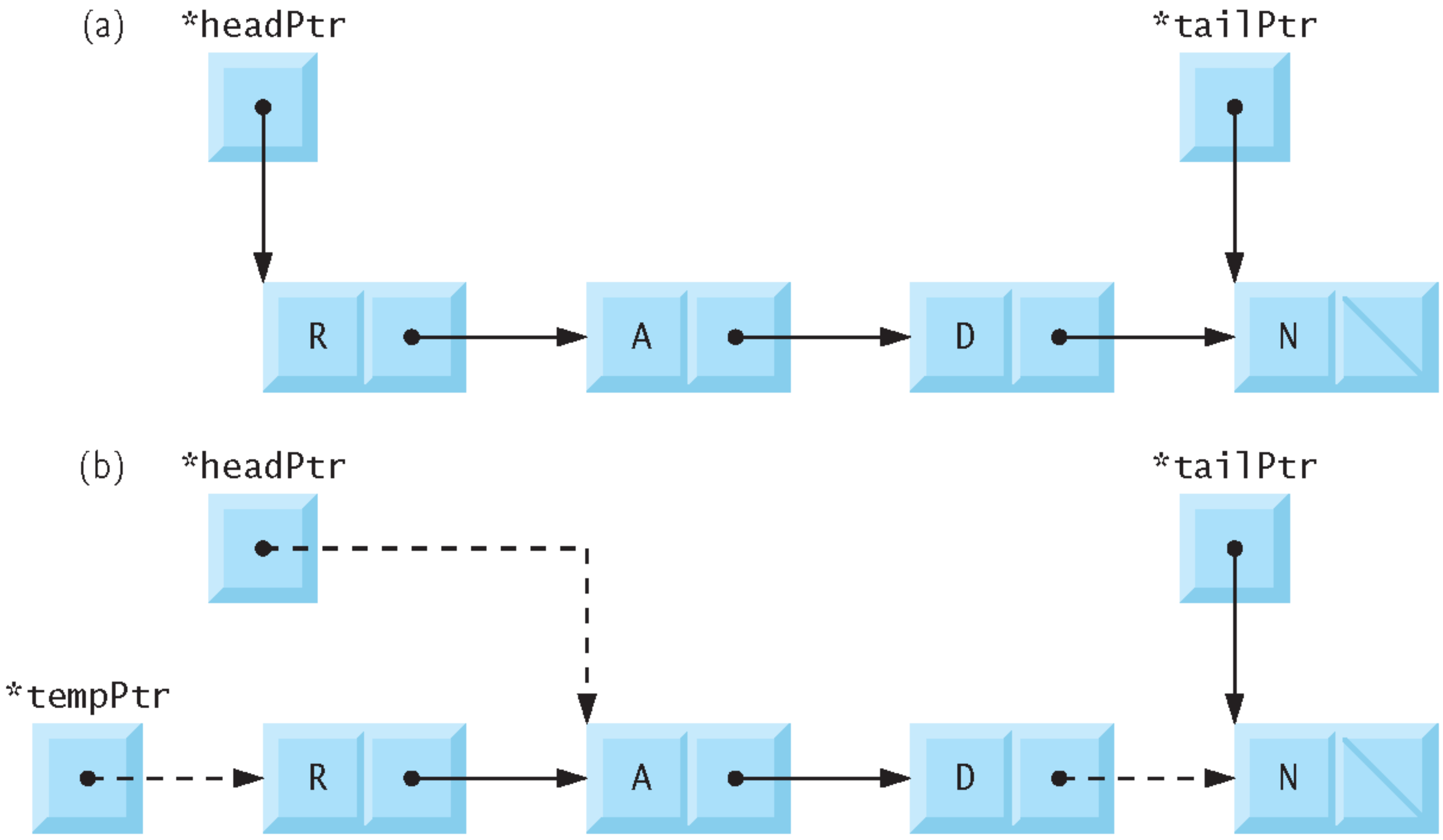


(a)
*headPtr
*tailPtr
newPtr
R
A
D
N
(b)
*headPtr
*tailPtr
newPtr
R
A
D
N

Queues(cont.)

```c

// remove node from queue head
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
{
   char value; // node value
   QueueNodePtr tempPtr; // temporary node pointer

   value = ( *headPtr )->data;
   tempPtr = *headPtr;
   *headPtr = ( *headPtr )->nextPtr;

   // if queue is empty
   if ( *headPtr == NULL ) {
      *tailPtr = NULL;
   } // end if

   free( tempPtr );
   return value;
} // end function dequeue

// return 1 if the queue is empty, 0 otherwise
int isEmpty( QueueNodePtr headPtr )
{
   return headPtr == NULL;
} // 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

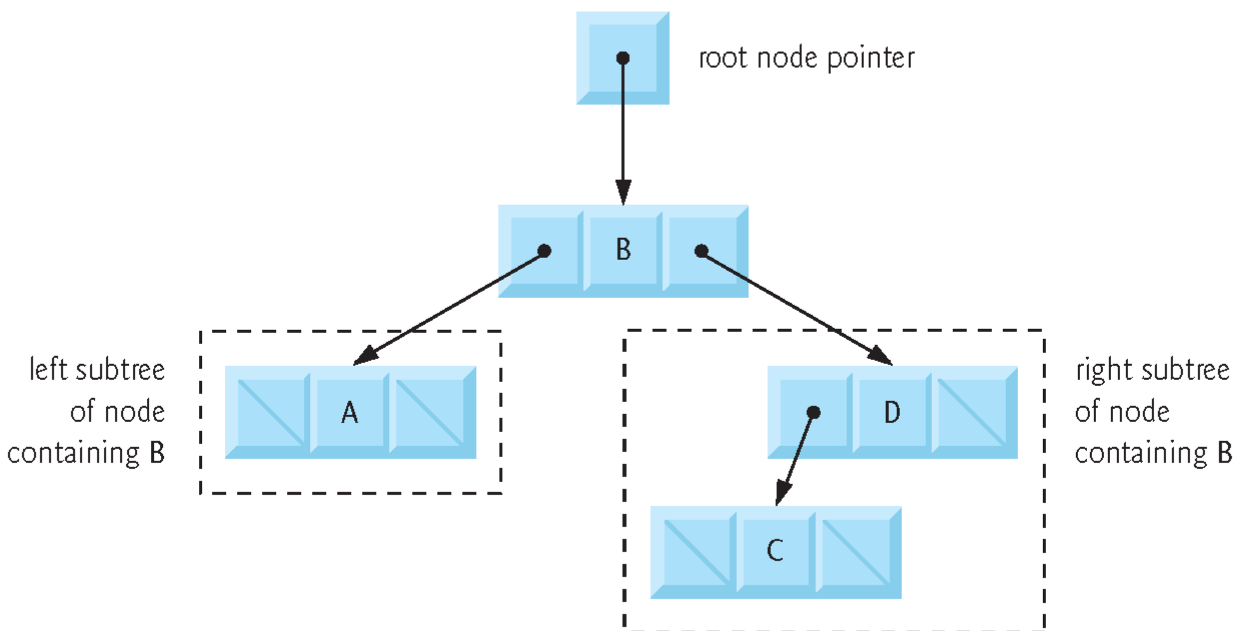
- Linked lists, stacks and queues are **linear data structures**. A **tree** is a nonlinear, two-dimensional data structure with special properties.

- 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.

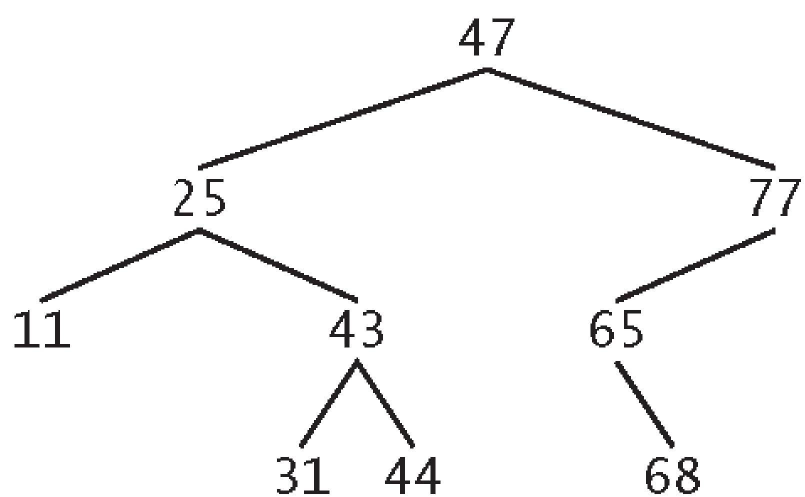
Trees(cont.)

- 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.

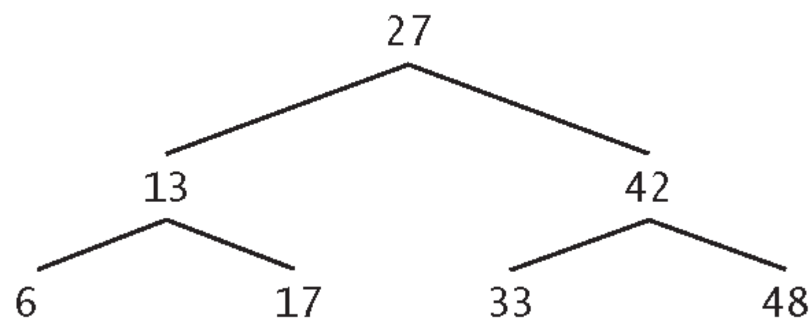


Trees(cont.)

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

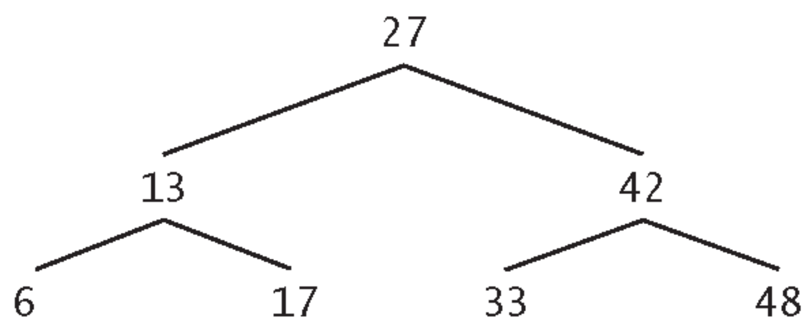
- The below BST is built from the following sequence of numbers.

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).

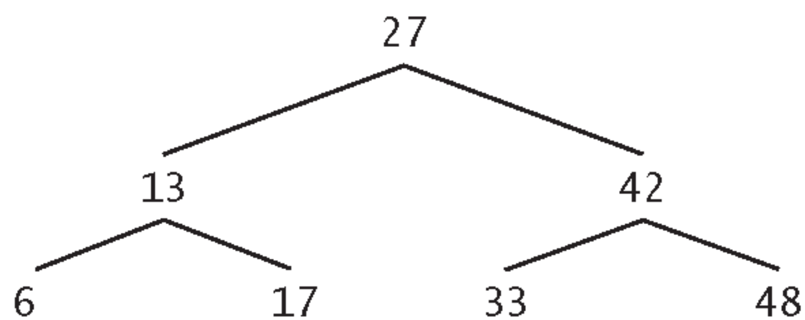


- So the inOrder (LDR) traversal of the tree is:

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).

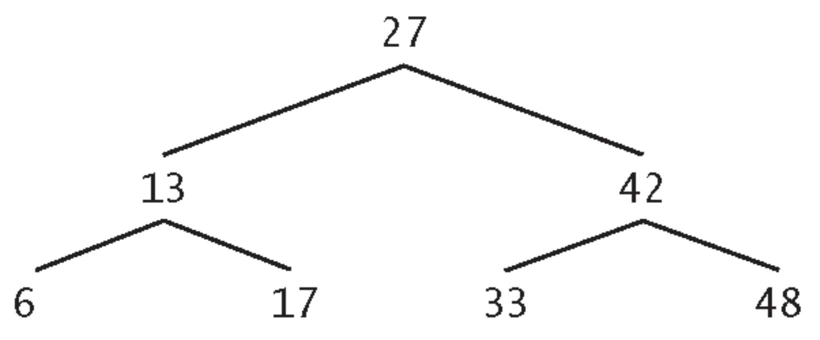


- So the preOrder (DLR) traversal of the tree is:

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

```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
}; // end structure treeNode

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 );
```

Tree Implementation-Insert Node

```
// insert node into tree
void insertNode( TreeNodePtr *treePtr, int value )
{
   // if tree is empty
   if ( *treePtr == NULL ) {
      *treePtr = malloc( sizeof( TreeNode ) );

      // if memory was allocated, then assign data
      if ( *treePtr != NULL ) {
         ( *treePtr )->data = value;
         ( *treePtr )->leftPtr = NULL;
         ( *treePtr )->rightPtr = NULL;
      } // end if
      else {
         printf( "%d not inserted. No memory available.\n", value );
      } // end else
   } // end if
   else { // tree is not empty
      // data to insert is less than data in current node
      if ( value < ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->leftPtr ), value );
      } // end if

      // data to insert is greater than data in current node
      else if ( value > ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->rightPtr ), value );
      } // end else if
      else { // duplicate data value ignored
         printf( "%s", "dup" );
      } // end else
   } // end else
} // end function insertNode
```

Tree Implementation-Traversal

```
// begin inorder traversal of tree
void inOrder( TreeNodePtr treePtr )
{
    // if tree is not empty, then traverse
    if ( treePtr != NULL ) {
        inOrder( treePtr->leftPtr );
        printf( "%3d", treePtr->data );
        inOrder( treePtr->rightPtr );
    } // end if
} // end function inOrder
```

```
// begin preorder traversal of tree
void preOrder( TreeNodePtr treePtr )
{
    // if tree is not empty, then traverse
    if ( treePtr != NULL ) {
        printf( "%3d", treePtr->data );
        preOrder( treePtr->leftPtr );
        preOrder( treePtr->rightPtr );
    } // end if
} // end function preOrder

// begin postorder traversal of tree
void postOrder( TreeNodePtr treePtr )
{
    // if tree is not empty, then traverse
    if ( treePtr != NULL ) {
        postOrder( treePtr->leftPtr );
        postOrder( treePtr->rightPtr );
        printf( "%3d", treePtr->data );
    } // end if
} // end function postOrder
```

Tree Implementation(cont.)

```
int main( void )
{
    unsigned int i; // counter to loop from 1-10
    int item; // variable to hold random values
    TreeNodePtr rootPtr = NULL; // tree initially empty

    srand( time( NULL ) );
    puts( "The numbers being placed in the tree are:" );

    // insert random values between 0 and 14 in the tree
    for ( i = 1; i <= 10; ++i ) {
        item = rand() % 15;
        printf( "%3d", item );
        insertNode( &rootPtr, item );
    } // end for

    // traverse the tree preOrder
    puts( "\n\nThe preOrder traversal is:" );
    preOrder( rootPtr );

    // traverse the tree inOrder
    puts( "\n\nThe inOrder traversal is:" );
    inOrder( rootPtr );

    // traverse the tree postOrder
    puts( "\n\nThe postOrder traversal
    postOrder( rootPtr );
} // 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
```