Data Structures

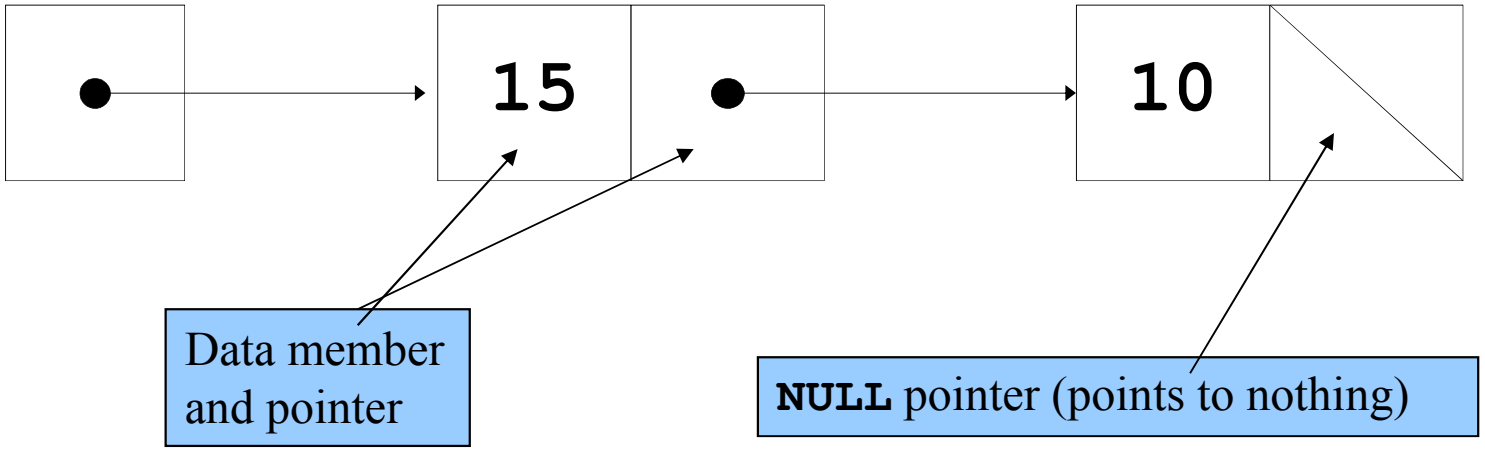
Outline

12.1 Introduction

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

12.2 Self-Referential Structures

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



12.2 Self-Referential Classes

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

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

12.3 Dynamic Memory Allocation

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

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

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

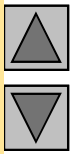
12.4 Linked Lists

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

12.4 Linked Lists

- Types of linked lists:
  - Singly linked list
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - Circular, singly linked
    - Pointer in the last node points back to the first node
  - Doubly linked list
    - Two “start pointers” – first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - Circular, doubly linked list
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

```c
/* Fig. 12.3: fig12_03.c
   Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

struct listNode {   /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void insert( ListNodePtr *, char );
char delete( ListNodePtr *, char );
int isEmpty( ListNodePtr );
void printList( ListNodePtr );
void instructions( void );

int main()
{
   ListNodePtr startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );
```



Outline

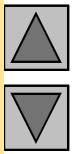
1. Define struct

1.1 Function prototypes

1.2 Initialize variables

2. Input choice

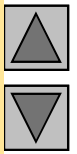
```
29
30    while ( choice != 3 ) {
31
32       switch ( choice ) {
33          case 1:
34             printf( "Enter a character: " );
35             scanf( "\n%c", &item );
36             insert( &startPtr, item );
37             printList( startPtr );
38             break;
39          case 2:
40             if ( !isEmpty( startPtr ) ) {
41                printf( "Enter character to be deleted: " );
42                scanf( "\n%c", &item );
43
44                if ( delete( &startPtr, item ) ) {
45                   printf( "%c deleted.\n", item );
46                   printList( startPtr );
47                }
48                else
49                   printf( "%c not found.\n\n", item );
50             }
51             else
52                printf( "List is empty.\n\n" );
53
54             break;
55          default:
56             printf( "Invalid choice.\n\n" );
57             instructions();
58             break;
59       }
```



Outline

2.1 `switch` statement

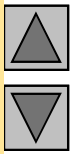
```
60
61          printf( "? " );
62          scanf( "%d", &choice );
63       }
64
65       printf( "End of run.\n" );
66       return 0;
67    }
68
69    /* Print the instructions */
70    void instructions( void )
71    {
72       printf( "Enter your choice:\n"
73               "   1 to insert an element into the list.\n"
74               "   2 to delete an element from the list.\n"
75               "   3 to end.\n" );
76    }
77
78    /* Insert a new value into the list in sorted order */
79    void insert( ListNodePtr *sPtr, char value )
80    {
81       ListNodePtr newPtr, previousPtr, currentPtr;
82
83       newPtr = malloc( sizeof( ListNode ) );
84
85       if ( newPtr != NULL ) {     /* is space available */
86          newPtr->data = value;
87          newPtr->nextPtr = NULL;
88
89          previousPtr = NULL;
90          currentPtr = *sPtr;
```



Outline

3. Function definitions

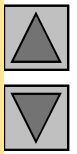
```
91
92       while ( currentPtr != NULL && value > currentPtr->data ) {
93          previousPtr = currentPtr;          /* walk to ...   */
94          currentPtr = currentPtr->nextPtr;  /* ... next node */
95       }
96
97       if ( previousPtr == NULL ) {
98          newPtr->nextPtr = *sPtr;
99          *sPtr = newPtr;
100      }
101      else {
102         previousPtr->nextPtr = newPtr;
103         newPtr->nextPtr = currentPtr;
104      }
105   }
106   else
107      printf( "%c not inserted. No memory available.\n", value );
108 }
109
110 /* Delete a list element */
111 char delete( ListNodePtr *sPtr, char value )
112 {
113   ListNodePtr previousPtr, currentPtr, tempPtr;
114
115   if ( value == ( *sPtr )->data ) {
116      tempPtr = *sPtr;
117      *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
118      free( tempPtr );             /* free the de-threaded node */
119      return value;
120   }
```



Outline

3. Function definitions

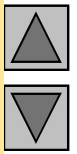
```
121   else {
122      previousPtr = *sPtr;
123      currentPtr = ( *sPtr )->nextPtr;
124
125      while ( currentPtr != NULL && currentPtr->data != value ) {
126         previousPtr = currentPtr;          /* walk to ...   */
127         currentPtr = currentPtr->nextPtr;  /* ... next node */
128      }
129
130      if ( currentPtr != NULL ) {
131         tempPtr = currentPtr;
132         previousPtr->nextPtr = currentPtr->nextPtr;
133         free( tempPtr );
134         return value;
135      }
136   }
137
138   return '\0';
139 }
140
141 /* Return 1 if the list is empty, 0 otherwise */
142 int isEmpty( ListNodePtr sPtr )
143 {
144   return sPtr == NULL;
145 }
146
147 /* Print the list */
148 void printList( ListNodePtr currentPtr )
149 {
150   if ( currentPtr == NULL )
```



Outline

3. Function definitions

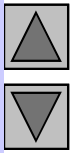
```
151      printf( "List is empty.\n\n" );
152   else {
153      printf( "The list is:\n" );
154
155      while ( currentPtr != NULL ) {
156         printf( "%c --> ", currentPtr->data );
157         currentPtr = currentPtr->nextPtr;
158      }
159
160      printf( "NULL\n\n" );
161   }
162}
```



Outline

3. Function definitions

```
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
```



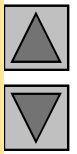
Outline

Program Output

12.5 Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to **`NULL`**
  - Constrained version of a linked list
- **`push`**
  - Adds a new node to the top of the stack
- **`pop`**
  - Removes a node from the top
  - Stores the popped value
  - Returns **`true`** if **`pop`** was successful

```
1  /* Fig. 12.8: fig12_08.c
2     dynamic stack program */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  struct stackNode {   /* self-referential structure */
7     int data;
8     struct stackNode *nextPtr;
9  };
10
11 typedef struct stackNode StackNode;
12 typedef StackNode *StackNodePtr;
13
14 void push( StackNodePtr *, int );
15 int pop( StackNodePtr * );
16 int isEmpty( StackNodePtr );
17 void printStack( StackNodePtr );
18 void instructions( void );
19
20 int main()
21 {
22    StackNodePtr stackPtr = NULL;  /* points to stack top */
23    int choice, value;
24
25    instructions();
26    printf( "? " );
27    scanf( "%d", &choice );
28
```



Outline

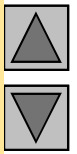
1. Define struct

1.1 Function definitions

1.2 Initialize variables

2. Input choice

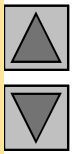
```
29    while ( choice != 3 ) {
30
31       switch ( choice ) {
32          case 1:      /* push value onto stack */
33             printf( "Enter an integer: " );
34             scanf( "%d", &value );
35             push( &stackPtr, value );
36             printStack( stackPtr );
37             break;
38          case 2:      /* pop value off stack */
39             if ( !isEmpty( stackPtr ) )
40                printf( "The popped value is %d.\n",
41                        pop( &stackPtr ) );
42
43             printStack( stackPtr );
44             break;
45          default:
46             printf( "Invalid choice.\n\n" );
47             instructions();
48             break;
49       }
50
51       printf( "? " );
52       scanf( "%d", &choice );
53    }
54
55    printf( "End of run.\n" );
56    return 0;
57 }
58
```



Outline

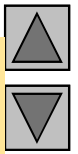
2.1 `switch` statement

```
59 /* Print the instructions */
60 void instructions( void )
61 {
62    printf( "Enter choice:\n"
63            "1 to push a value on the stack\n"
64            "2 to pop a value off the stack\n"
65            "3 to end program\n" );
66 }
67
68 /* Insert a node at the stack top */
69 void push( StackNodePtr *topPtr, int info )
70 {
71    StackNodePtr newPtr;
72
73    newPtr = malloc( sizeof( StackNode ) );
74    if ( newPtr != NULL ) {
75       newPtr->data = info;
76       newPtr->nextPtr = *topPtr;
77       *topPtr = newPtr;
78    }
79    else
80       printf( "%d not inserted. No memory available.\n",
81               info );
82 }
83
```



Outline

3. Function definitions

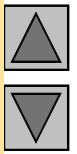


Outline

3. Function definitions

```
84 /* Remove a node from the stack top */
85 int pop( StackNodePtr *topPtr )
86 {
87    StackNodePtr tempPtr;
88    int popValue;
89
90    tempPtr = *topPtr;
91    popValue = ( *topPtr )->data;
92    *topPtr = ( *topPtr )->nextPtr;
93    free( tempPtr );
94    return popValue;
95 }
96
97 /* Print the stack */
98 void printStack( StackNodePtr currentPtr )
99 {
100   if ( currentPtr == NULL )
101      printf( "The stack is empty.\n\n" );
102   else {
103      printf( "The stack is:\n" );
104
105      while ( currentPtr != NULL ) {
106         printf( "%d --> ", currentPtr->data );
107         currentPtr = currentPtr->nextPtr;
108      }
109
110      printf( "NULL\n\n" );
111   }
112 }
113
```

```
114 /* Is the stack empty? */
115 int isEmpty( StackNodePtr topPtr )
116 {
117    return topPtr == NULL;
118 }
```



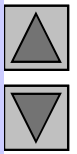
Outline

3. Function definitions

Program Output

```
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.
```



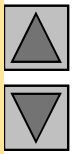
Outline

Program Output

12.6 Queues

- Queue
  - Similar to a supermarket checkout line
  - First-in, first-out (FIFO)
  - Nodes are removed only from the head
  - Nodes are inserted only at the tail
- Insert and remove operations
  - Enqueue (insert) and dequeue (remove)

```c
/* Fig. 12.13: fig12_13.c
   Operating and maintaining a queue */

#include <stdio.h>
#include <stdlib.h>

struct queueNode {   /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function prototypes */
void printQueue( QueueNodePtr );
int isEmpty( QueueNodePtr );
char dequeue( QueueNodePtr *, QueueNodePtr * );
void enqueue( QueueNodePtr *, QueueNodePtr *, char );
void instructions( void );

int main()
{
   QueueNodePtr headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );
```



Outline

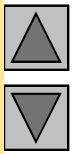
1. Define struct

1.1 Function prototypes

1.1 Initialize variables

2. Input choice

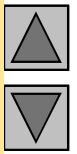
```
31
32    while ( choice != 3 ) {
33
34       switch( choice ) {
35
36          case 1:
37             printf( "Enter a character: " );
38             scanf( "\n%c", &item );
39             enqueue( &headPtr, &tailPtr, item );
40             printQueue( headPtr );
41             break;
42          case 2:
43             if ( !isEmpty( headPtr ) ) {
44                item = dequeue( &headPtr, &tailPtr );
45                printf( "%c has been dequeued.\n", item );
46             }
47
48             printQueue( headPtr );
49             break;
50
51          default:
52             printf( "Invalid choice.\n\n" );
53             instructions();
54             break;
55       }
56
57       printf( "? " );
58       scanf( "%d", &choice );
59    }
60
61    printf( "End of run.\n" );
62    return 0;
63 }
64
```



Outline

2.1 `switch` statement

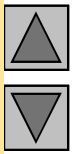
```
65 void instructions( void )
66 {
67    printf ( "Enter your choice:\n"
68             "   1 to add an item to the queue\n"
69             "   2 to remove an item from the queue\n"
70             "   3 to end\n" );
71 }
72
73 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
74               char value )
75 {
76    QueueNodePtr newPtr;
77
78    newPtr = malloc( sizeof( QueueNode ) );
79
80    if ( newPtr != NULL ) {
81       newPtr->data = value;
82       newPtr->nextPtr = NULL;
83
84       if ( isEmpty( *headPtr ) )
85          *headPtr = newPtr;
86       else
87          ( *tailPtr )->nextPtr = newPtr;
88
89       *tailPtr = newPtr;
90    }
91    else
92       printf( "%c not inserted. No memory available.\n",
93               value );
94 }
95
```



Outline

3. Function definitions

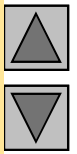
```
96 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
97 {
98    char value;
99    QueueNodePtr tempPtr;
100
101   value = ( *headPtr )->data;
102   tempPtr = *headPtr;
103   *headPtr = ( *headPtr )->nextPtr;
104
105   if ( *headPtr == NULL )
106      *tailPtr = NULL;
107
108   free( tempPtr );
109   return value;
110 }
111
112 int isEmpty( QueueNodePtr headPtr )
113 {
114   return headPtr == NULL;
115 }
116
117 void printQueue( QueueNodePtr currentPtr )
118 {
119   if ( currentPtr == NULL )
120      printf( "Queue is empty.\n\n" );
121   else {
122      printf( "The queue is:\n" );
```



Outline

3. Function definitions

```
123
124       while ( currentPtr != NULL ) {
125          printf( "%c --> ", currentPtr->data );
126          currentPtr = currentPtr->nextPtr;
127       }
128
129       printf( "NULL\n\n" );
130    }
131 }
```



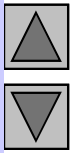
Outline

3. Function definitions

Program Output

```
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.
```



Outline

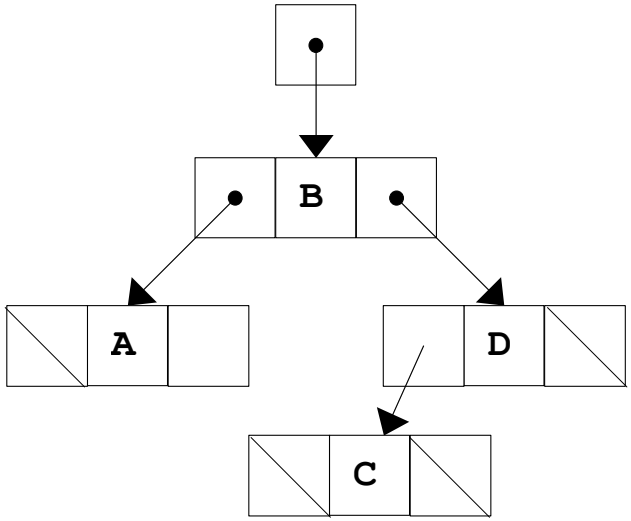
Program Output

12.7 Trees

- Tree nodes contain two or more links
  - All other data structures we have discussed only contain one
- Binary trees
  - All nodes contain two links
    - None, one, or both of which may be NULL
  - The root node is the first node in a tree.
  - Each link in the root node refers to a child
  - A node with no children is called a leaf node

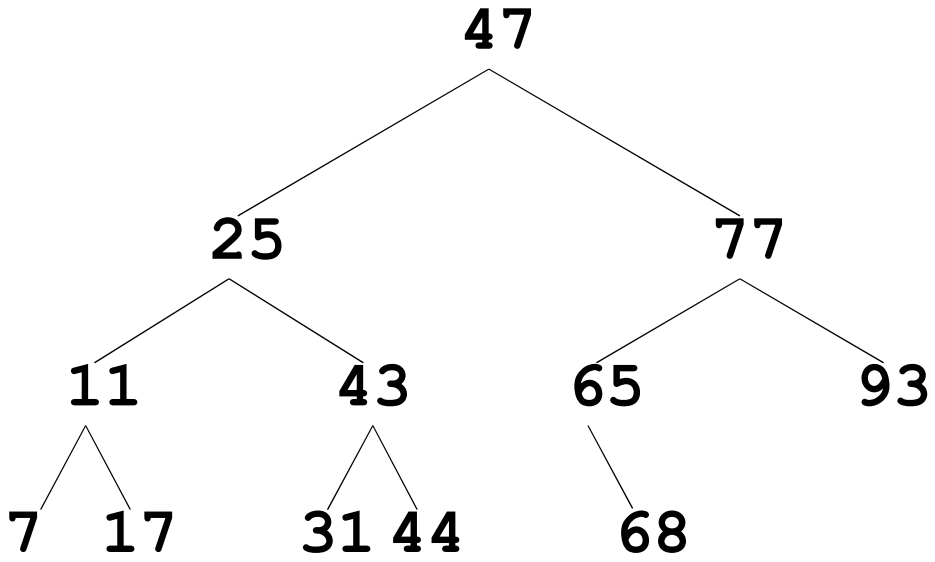
12.7 Trees

- Diagram of a binary tree



12.7 Trees

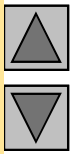
- Binary search tree
  - Values in left subtree less than parent
  - Values in right subtree greater than parent
  - Facilitates duplicate elimination
  - Fast searches - for a balanced tree, maximum of $\lceil \log_2 n \rceil$ comparisons



12.7 Trees

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

```c
/* Fig. 12.19: fig12_19.c
   Create a binary tree and traverse it
   preorder, inorder, and postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct treeNode {
   struct treeNode *leftPtr;
   int data;
   struct treeNode *rightPtr;
};

typedef struct treeNode TreeNode;
typedef TreeNode *TreeNodePtr;

void insertNode( TreeNodePtr *, int );
void inOrder( TreeNodePtr );
void preOrder( TreeNodePtr );
void postOrder( TreeNodePtr );

int main()
{
   int i, item;
   TreeNodePtr rootPtr = NULL;

   srand( time( NULL ) );

```



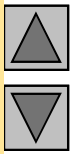
Outline

1. Define structure

1.1 Function prototypes

1.2 Initialize variables

```
29    /* insert random values between 1 and 15 in the tree */
30    printf( "The numbers being placed in the tree are:\n" );
31
32    for ( i = 1; i <= 10; i++ ) {
33       item = rand() % 15;
34       printf( "%3d", item );
35       insertNode( &rootPtr, item );
36    }
37
38    /* traverse the tree preOrder */
39    printf( "\n\nThe preOrder traversal is:\n" );
40    preOrder( rootPtr );
41
42    /* traverse the tree inOrder */
43    printf( "\n\nThe inOrder traversal is:\n" );
44    inOrder( rootPtr );
45
46    /* traverse the tree postOrder */
47    printf( "\n\nThe postOrder traversal is:\n" );
48    postOrder( rootPtr );
49
50    return 0;
51 }
52
53 void insertNode( TreeNodePtr *treePtr, int value )
54 {
55    if ( *treePtr == NULL ) {    /* *treePtr is NULL */
56       *treePtr = malloc( sizeof( TreeNode ) );
57
58       if ( *treePtr != NULL ) {
59          ( *treePtr )->data = value;
60          ( *treePtr )->leftPtr = NULL;
61          ( *treePtr )->rightPtr = NULL;
62       }
```



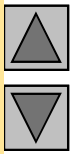
Outline

1.3 Insert random elements

2. Function calls

3. Function definitions

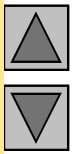
```
63       else
64          printf( "%d not inserted. No memory available.\n",
65                  value );
66    }
67    else
68       if ( value < ( *treePtr )->data )
69          insertNode( &( ( *treePtr )->leftPtr ), value );
70       else if ( value > ( *treePtr )->data )
71          insertNode( &( ( *treePtr )->rightPtr ), value );
72       else
73          printf( "dup" );
74 }
75
76 void inOrder( TreeNodePtr treePtr )
77 {
78    if ( treePtr != NULL ) {
79       inOrder( treePtr->leftPtr );
80       printf( "%3d", treePtr->data );
81       inOrder( treePtr->rightPtr );
82    }
83 }
84
85 void preOrder( TreeNodePtr treePtr )
86 {
87    if ( treePtr != NULL ) {
88       printf( "%3d", treePtr->data );
89       preOrder( treePtr->leftPtr );
90       preOrder( treePtr->rightPtr );
91    }
92 }
```



Outline

3. Function definitions

```
93
94 void postOrder( TreeNodePtr treePtr )
95 {
96    if ( treePtr != NULL ) {
97       postOrder( treePtr->leftPtr );
98       postOrder( treePtr->rightPtr );
99       printf( "%3d", treePtr->data );
100   }
101}
```



Outline

3. Function definitions

Program Output

```
The numbers being placed in the tree are:
  7   8   0   6 14   1   0dup 13   0dup   7dup

The preOrder traversal is:
  7   0   6   1   8 14 13

The inOrder traversal is:
  0   1   6   7   8 13 14

The postOrder traversal is:
  1   6   0 13 14   8   7
```