12

C Data Structures

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**

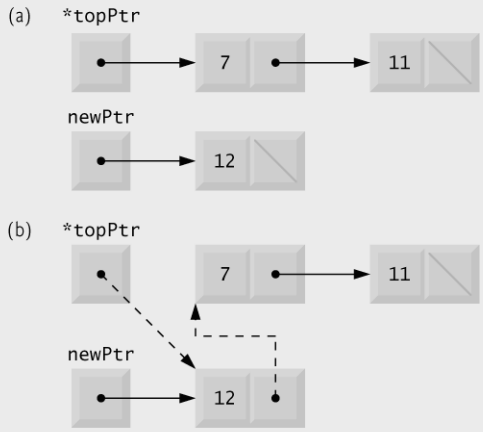


Fig. 12.10 | push operation.

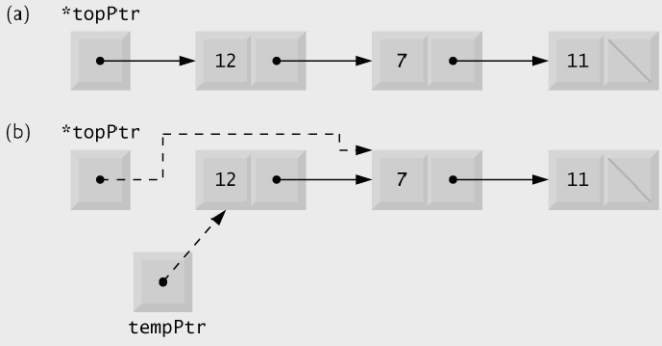


Fig. 12.11 | pop operation.

Common Programming Error 12.6

Not setting the link in the bottom node of a stack to `NULL` can lead to runtime errors.

Outline

fig12_08.c

(1 of 5)

Each node in the stack contains a data element and a pointer to the next node

```
1  /* Fig. 12.8: fig12_08.c
2     dynamic 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 );
21
22 /* function main begins program execution */
23 int main( void )
24 {
25    StackNodePtr stackPtr = NULL; /* points to stack top */
26    int choice; /* user's menu choice */
27    int value;  /* int input by user */
28
29    instructions(); /* display the menu */
30    printf( "? " );
```

Outline

fig12_08.c

(2 of 5)

```
31    scanf( "%d", &choice );
32
33    /* while user does not enter 3 */
34    while ( choice != 3 ) {
35
36       switch ( choice ) {
37
38          /* push value onto stack */
39          case 1:
40             printf( "Enter an integer: " );
41             scanf( "%d", &value );
42             push( &stackPtr, value );
43             printStack( stackPtr );
44             break;
45
46          /* pop value off stack */
47          case 2:
48
49             /* if stack is not empty */
50             if ( !isEmpty( stackPtr ) ) {
51                printf( "The popped value is %d.\n", pop( &stackPtr ) );
52             } /* end if */
53
54             printStack( stackPtr );
55             break;
56
57          default:
58             printf( "Invalid choice.\n\n" );
59             instructions();
60             break;
```

Outline

fig12_08.c

(3 of 5)

```
61
62       } /* end switch */
63
64       printf( "? " );
65       scanf( "%d", &choice );
66    } /* end while */
67
68    printf( "End of run.\n" );
69
70    return 0; /* indicates successful termination */
71
72 } /* end main */
73
74 /* display program instructions to user */
75 void instructions( void )
76 {
77    printf( "Enter choice:\n"
78       "1 to push a value on the stack\n"
79       "2 to pop a value off the stack\n"
80       "3 to end program\n" );
81 } /* end function instructions */
82
83 /* Insert a node at the stack top */
84 void push( StackNodePtr *topPtr, int info )
85 {
86    StackNodePtr newPtr; /* pointer to new node */
87
88    newPtr = malloc( sizeof( StackNode ) );
89
```

To insert a node into the stack, memory must first be allocated for that node

Outline

```
90    /* insert the node at stack top */
91    if ( newPtr != NULL ) {
92       newPtr->data = info;
93       newPtr->nextPtr = *topPtr;
94       *topPtr = newPtr;
95    } /* end if */
96    else { /* no space available */
97       printf( "%d not inserted. No memory available.\n", info );
98    } /* end else */
99
100 } /* end function push */
101
102 /* Remove a node from the stack top */
103 int pop( StackNodePtr *topPtr )
104 {
105   StackNodePtr tempPtr; /* temporary node pointer */
106   int popValue; /* node value */
107
108   tempPtr = *topPtr;
109   popValue = ( *topPtr )->data;
110   *topPtr = ( *topPtr )->nextPtr;
111   free( tempPtr );
112
113   return popValue;
114
115 } /* end function pop */
116
```

Stack nodes are always inserted at the top, so there is no need to search for the node's place

Inserted node becomes the new top

Stack nodes are always removed from the top, so there is no need to search for the node's place

Second node becomes the new top

Free the memory of the popped node

fig12_08.c

(4 of 5)

Outline

```
117  /* Print the stack */
118  void printStack( StackNodePtr currentPtr )
119 {
120
121   /* if stack is empty */
122   if ( currentPtr == NULL ) {
123      printf( "The stack is empty.\n\n" );
124   } /* end if */
125   else {
126      printf( "The stack is:\n" );
127
128      /* while not the end of the stack */
129      while ( currentPtr != NULL ) {
130         printf( "%d --> ", currentPtr->data );
131         currentPtr = currentPtr->nextPtr;
132      } /* end while */
133
134      printf( "NULL\n\n" );
135   } /* end else */
136
137 } /* end function printList */
137 } /* end function printList */
138
139 /* Return 1 if the stack is empty, 0 otherwise */
140 int isEmpty( StackNodePtr topPtr )
141 {
142   return topPtr == NULL;
143
144 } /* end function isEmpty */
```

fig12_08.c

(5 of 5)

```
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


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
```

(continued on next slide…)

Outline

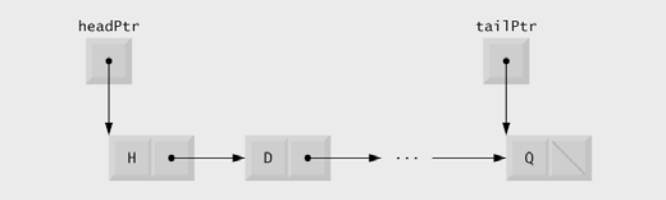
(continued from previous slide…)

```
? 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

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)**



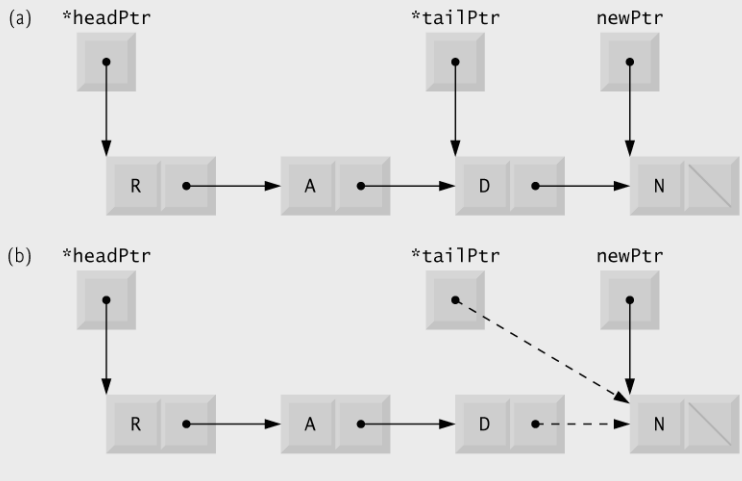


Fig. 12.15 | enqueue operation.

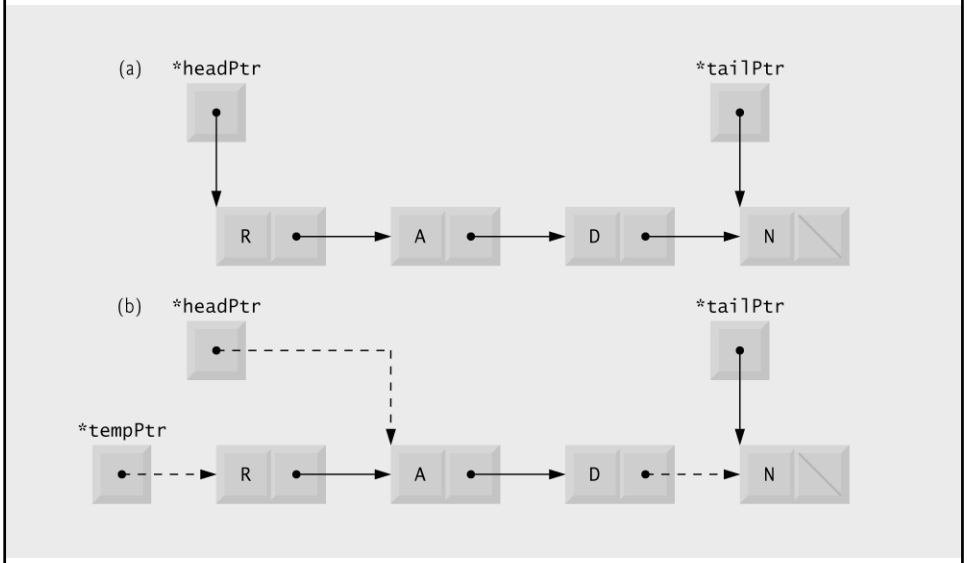


Fig. 12.16 | dequeue operation.

Common Programming Error 12.7

Not setting the link in the last node of a queue to `NULL` can lead to runtime errors.

Outline

fig12_13.c

(1 of 6)

```
1  /* Fig. 12.13: fig12_13.c
2     Operating and maintaining a queue */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  /* self-referential structure */
8  struct queueNode {
9     char data;                /* define data as a char */
10    struct queueNode *nextPtr; /* queueNode pointer */
11 }; /* end structure queueNode */
12
13 typedef struct queueNode QueueNode;
14 typedef QueueNode *QueueNodePtr;
15
16 /* function prototypes */
17 void printQueue( QueueNodePtr currentPtr );
18 int isEmpty( QueueNodePtr headPtr );
19 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
20 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
21               char value );
22 void instructions( void );
23
24 /* function main begins program execution */
25 int main( void )
26 {
27    QueueNodePtr headPtr = NULL; /* initialize headPtr */
28    QueueNodePtr tailPtr = NULL; /* initialize tailPtr */
29    int choice;                  /* user's menu choice */
30    char item;                   /* char input by user */
```

Each node in the queue contains a data element and a pointer to the next node

Note that unlike linked lists and stacks, queues keep track of the tail node as well as the head

Outline

fig12_13.c

(2 of 6)

```
31
32    instructions(); /* display the menu */
33    printf( "? " );
34    scanf( "%d", &choice );
35
36    /* while user does not enter 3 */
37    while ( choice != 3 ) {
38
39       switch( choice ) {
40
41          /* enqueue value */
42          case 1:
43             printf( "Enter a character: " );
44             scanf( "\n%c", &item );
45             enqueue( &headPtr, &tailPtr, item );
46             printQueue( headPtr );
47             break;
48
49          /* dequeue value */
50          case 2:
51
52             /* if queue is not empty */
53             if ( !isEmpty( headPtr ) ) {
54                item = dequeue( &headPtr, &tailPtr );
55                printf( "%c has been dequeued.\n", item );
56             } /* end if */
57
58             printQueue( headPtr );
59             break;
```

```
60
61          default:
62             printf( "Invalid choice.\n\n" );
63             instructions();
64             break;
65
66       } /* end switch */
67
68       printf( "? " );
69       scanf( "%d", &choice );
70    } /* end while */
71
72    printf( "End of run.\n" );
73
74    return 0; /* indicates successful termination */
75
76 } /* end main */
77
78 /* display program instructions to user */
79 void instructions( void )
80 {
81    printf ( "Enter your choice:\n"
82             "   1 to add an item to the queue\n"
83             "   2 to remove an item from the queue\n"
84             "   3 to end\n" );
85 } /* end function instructions */
```


Outline

fig12_13.c

(3 of 6)


```
86
87 /* insert a node a queue tail */
88 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
89               char value )
90 {
91    QueueNodePtr newPtr; /* pointer to new node */
92
93    newPtr = malloc( sizeof( QueueNode ) );
94
95    if ( newPtr != NULL ) { /* is space available */
96       newPtr->data = value;
97       newPtr->nextPtr = NULL;
98
99       /* if empty, insert node at head */
100      if ( isEmpty( *headPtr ) ) {
101         *headPtr = newPtr;
102      } /* end if */
103      else {
104         ( *tailPtr )->nextPtr = newPtr;
105      } /* end else */
106
107      *tailPtr = newPtr;
108   } /* end if */
109   else {
110      printf( "%c not inserted. No memory available.\n", value );
111   } /* end else */
112
113 } /* end function enqueue */
```


Outline

fig12_13.c

(4 of 6)

To insert a node into the queue, memory must first be allocated for that node

Queue nodes are always inserted at the tail, so there is no need to search for the node's place

If the queue is empty, the inserted node becomes the new head in addition to the new tail

Inserted node becomes the new tail

Outline

fig12_13.c

(5 of 6)

```
114
115 /* remove node from queue head */
116 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
117  {
118   char value;           /* node value */
119   QueueNodePtr tempPtr; /* temporary node pointer */
120
121   value = ( *headPtr )->data;
122   tempPtr = *headPtr;
123   *headPtr = ( *headPtr )->nextPtr;
124
125   /* if queue is empty */
126   if ( *headPtr == NULL ) {
127      *tailPtr = NULL;
128   } /* end if */
129
130   free( tempPtr );
131
132   return value;
133
134 } /* end function dequeue */
135
136 /* Return 1 if the list is empty, 0 otherwise */
137 int isEmpty( QueueNodePtr headPtr )
138 {
139   return headPtr == NULL;
140
141 } /* end function isEmpty */
```

Queue nodes are always removed from the head, so there is no need to search for the node's place

Second node becomes the new head

If the removed node is the only node in the queue, it is the tail as well as the head of the queue, so **`tailPtr`** must be set to **`NULL`**

Free the memory of the removed node

Outline

fig12_13.c

(6 of 6)

```
142
143 /* Print the queue */
144 void printQueue( QueueNodePtr currentPtr )
145 {
146
147   /* if queue is empty */
148   if ( currentPtr == NULL ) {
149      printf( "Queue is empty.\n\n" );
150   } /* end if */
151   else {
152      printf( "The queue is:\n" );
153
154    /* while not end of queue */
155      while ( currentPtr != NULL ) {
156         printf( "%c --> ", currentPtr->data );
157         currentPtr = currentPtr->nextPtr;
158      } /* end while */
159
160      printf( "NULL\n\n" );
161   } /* end else */
162
163 } /* end function printQueue */
```

```
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
```

(continued on next slide…)

Outline

(continued from previous slide…)


```
? 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

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**

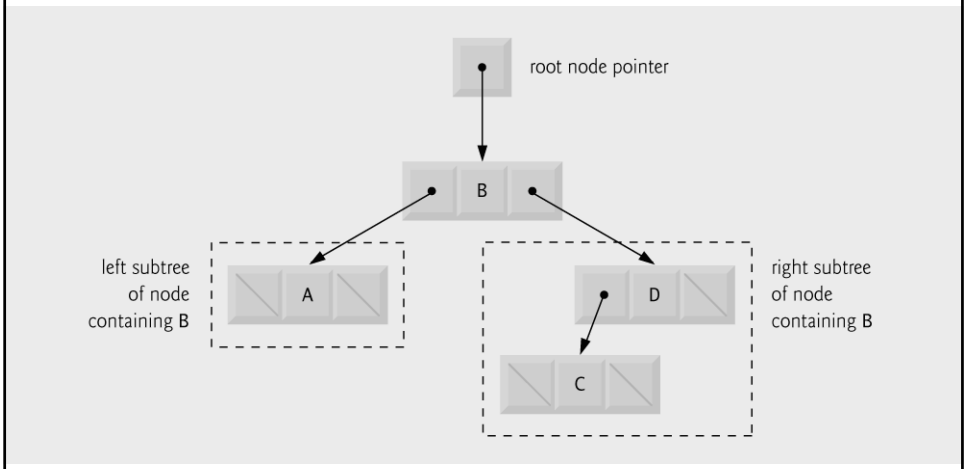


Fig. 12.17 | Binary tree graphical representation.

Common Programming Error 12.8

Not setting to `NULL` the links in leaf nodes of a tree can lead to runtime errors.

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 log n comparisons**

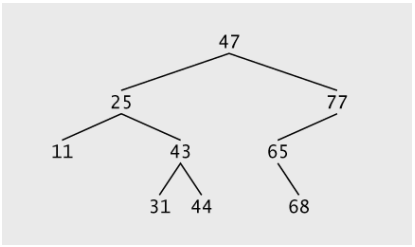
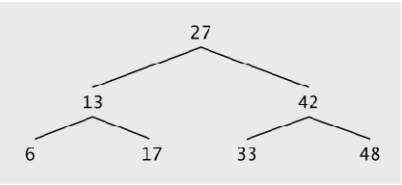


Fig. 12.18 | Binary search tree.

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**



Outline

fig12_19.c

(1 of 5)

```
1  /* Fig. 12.19: fig12_19.c
2     Create a binary tree and traverse it
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 );
23
24 /* function main begins program execution */
25 int main( void )
26 {
27    int i; /* counter to loop from 1-10 */
28    int item; /* variable to hold random values */
29    TreeNodePtr rootPtr = NULL; /* tree initially empty */
30
```

Each node in the tree contains a data element and a pointer to the left and right child nodes

Outline

fig12_19.c

(2 of 5)

```
31    srand( time( NULL ) );
32    printf( "The numbers being placed in the tree are:\n" );
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    printf( "\n\nThe preOrder traversal is:\n" );
43    preOrder( rootPtr );
44
45    /* traverse the tree inOrder */
46    printf( "\n\nThe inOrder traversal is:\n" );
47    inOrder( rootPtr );
48
49    /* traverse the tree postOrder */
50    printf( "\n\nThe postOrder traversal is:\n" );
51    postOrder( rootPtr );
52
53    return 0; /* indicates successful termination */
54
55 } /* end main */
```

Outline

fig12_19.c

(3 of 5)

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

To insert a node into the tree, memory must first be allocated for that node

If the inserted node's data is less than the current node's, the program will attempt to insert the node at the current node's left child.

Outline

fig12_19.c

(4 of 5)

```
82
83       /* data to insert is greater than data in current node */
84       else if ( value > ( *treePtr )->data ) {
85          insertNode( &( *treePtr )->rightPtr ), value );
86       } /* end else if */
87       else { /* duplicate data value ignored */
88          printf( "dup" );
89       } /* end else */
90
91    } /* end else */
92
93 } /* end function insertNode */
94
95 /* begin inorder traversal of tree */
96 void inOrder( TreeNodePtr treePtr )
97 {
98
99    /* if tree is not empty then traverse */
100   if ( treePtr != NULL ) {
101      inOrder( treePtr->leftPtr );
102      printf( "%3d", treePtr->data );
103      inOrder( treePtr->rightPtr );
104   } /* end if */
105
106 } /* end function inOrder */
107
108 /* begin preorder traversal of tree */
109 void preOrder( TreeNodePtr treePtr )
110 {
111
```

If the inserted node's data is greater than the current node's, the program will attempt to insert the node at the current node's right child.

The inorder traversal calls an inorder traversal on the node's left child, then prints the node itself, then calls an inorder traversal on the right child.

Outline

fig12_19.c

(5 of 5)

```
112   /* if tree is not empty then traverse */
113   if ( treePtr != NULL ) {
114      printf( "%3d", treePtr->data );
115      preOrder( treePtr->leftPtr );
116      preOrder( treePtr->rightPtr );
117   } /* end if */
118
119 } /* end function preOrder */
120
121 /* begin postorder traversal of tree */
122 void postOrder( TreeNodePtr treePtr )
123 {
124
125   /* if tree is not empty then traverse */
126   if ( treePtr != NULL ) {
127      postOrder( treePtr->leftPtr );
128      postOrder( treePtr->rightPtr );
129      printf( "%3d", treePtr->data );
130   } /* end if */
131
132 } /* end function postOrder */
```

The preorder traversal prints the node itself, then calls a preorder traversal on the node's left child, then calls a preorder traversal on the right child.

The postorder traversal calls an postorder traversal on the node's left child, then calls an postorder traversal on the right child, then prints the node itself.

```
The numbers being placed in the tree are:
  6  7  4 12  7dup  2  2dup  5  7dup 11

The preOrder traversal is:
  6  4  2  5  7 12 11

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

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

Outline

Summary

- **Dynamic data structures are efficient to handle:**
  - **The problem with unpredictable input size (typical case in real)**
  - **The data that grow and shrink very often during execution**
- **Different problems require different data structures**
  - **Linked lists: put data in an array-like fashion**
    - **Easier to collect data**
  - **Stacks: last-in first-out (operation at top of stack)**
    - **Suitable for post-order (reversed) processing (like function call?)**
  - **Queues: first-in first-out (adding some delay?)**
    - **Suitable for waiting buffers**
  - **Trees: contain two or more links to next data**
    - **Efficient for searching and sorting of data**
- **Suitable data structure is the key to an efficient program**
  - **Many data structures are waiting for your investigation**