Chapter 12: Searching: Binary Trees and Hash Tables

Exercises 12.1

1. 4, 6
2. 4, 1, 2, 3
3. 4, 6, 5
4. 4, 1, 0
5. 4, 6, 7, 8
6. template <typename ElementType>
   void linearSearch(ElementType x[], ElementType item, int capacity,
                     int n, bool & found, int & loc)
   
   /*---------------------------------------*/
   Linear search a list stored in an array x for an item, which is added
   at the end of the list, to improve performance.
   
   Precondition: n is the number of items in the array and is less
   than the array's capacity.
   Postcondition: found is true and loc is the position of item if the
   search is successful; otherwise found is false and loc == n.
   */
   {
      assert(n < capacity);
      x[n] = item;
      loc = 0;
      while (x[loc] != n)
         loc++;
      found = loc != n;
   }

7. template <typename ElementType>
   void linearSearch(ElementType x[], ElementType item, int n,
                      bool & found, int & loc)
   
   /*---------------------------------------*/
   Linear search an ordered list stored in an array x for an item.
   
   Precondition: n is the number of items stored in the array.
   Postcondition: found is true and loc is the position of item if the
   search is successful; otherwise found is false.
   */
   {
      loc = 0;
      while (loc < n && item > x[loc])
         loc++;
      found = (loc < n && x[loc] == item);
   }
8. template <typename ElementType>
   void recLinearSearch(ElementType x[], ElementType item, int n,
                         int start, bool & found, int & loc)
   /******************************************************************************/
   // Recursively linear search a list stored in an array x for an item.
   // Precondition: n is the number of items in the array and 0 <= start <= n. Initial call is with start = 0.
   // Postcondition: found is true and loc is the position of item if the search is successful; otherwise found is false and loc == n.
   /******************************************************************************/
   {
      loc = start;
      if (start == n)
         found = false;
      else if (item == x[start])
         found = true;
      else
         recLinearSearch(x, item, n, start + 1, found, loc);
   }

9. template <typename ElementType>
   void recLinkedLinearSearch(NodePointer first, ElementType item,
                                bool & found, NodePointer & locptr)
   /******************************************************************************/
   // Recursively linear search a linked list for an item.
   // Precondition: first points to first node in the linked list.
   // Postcondition: found is true and locptr positioned at item if the search is successful; otherwise found is false and loc == n.
   /******************************************************************************/
   {
      locptr = first;
      if (locptr == 0)
         found = false;
      else if (item == locptr->data)
         found = true;
      else
         recLinkedLinearSearch(first->next, item, found, locptr);
   }

10. The following solution uses an array to store the list so that its O(n) computing time caused by having to move array elements in the move-to-the-front operation is very visible. Using a vector, we could conceal this by using its insert() operation.

    template <typename ElementType>
    void moveFrontLinearSearch(ElementType x[], int size n, ElementType item,
                                 bool & found, int & loc)
    /******************************************************************************/
    // Self-organizing linear search of a list stored in an array x for an item.
    // Precondition: n is the number of items stored in the array.
Postcondition: found is true, loc is position of item, and item is moved to the front of the list if the search is successful; otherwise found is false and loc == n.

---

```cpp
{  
  found = false;
  loc = 0;
  while (!found && loc < n)  
  {  
    if (item == x[loc])
      found = true;
    else
      loc++;
  }
  if (found)   // Move to front of list
  {  
    for (int i = loc; i > 0; i--)
      x[i] = x[i - 1];
    x[0] = item;
  }
}
```

11.

```cpp
template <typename ElementType>
void moveFrontLinkedLinearSearch(NodePointer first,
        const ElementType & item,
        bool & found, NodePointer & locptr)
/*----------------------------------*/
Self-organizing linear search of a linked list for an item.

Precondition: first points to first node in the linked list.
Postcondition: found is true, locptr positioned at item, and item is moved to the front of the list if the search is successful;

{  
  found = false;
  locptr = first;
  NodePointer prev = 0;
  for (;;)
  {  
    if (found || locptr == 0) return;
    if (item == locptr->data)
      found = true;
    else
    {  
      prevptr = locptr;
      locptr = locptr->next;
    }
  }
  if (found && locptr != first)   // Move to front
  {  
    prev->next = locptr->next;
    locptr->next = first;
    first = locptr;
  }
}
```
12. In the solution for #10, instead of moving all items up to the location where the item is found, simply swap the found item with its predecessor.

13. In the solution for #11, instead of changing links so node moves to front of list, simply swap the values stored in the nodes pointed to be prev and locptr.

14.

```cpp
template <class ElementType>
void interpolationSearch(ElementType x[], ElementType item, int n,
                          bool & found, int & loc)
{ /*--------------------------------------------------------------*/
    Linear interpolation of an ordered list stored in an array x for an item.

    Precondition: n is the number of items stored in the array.
    Postcondition: found is true and loc is the position of item if the
                  search is successful; otherwise found is false.
    *------------------------------------------------------------------*/
    int first = 0,
         last = n - 1;
    if (item < x[first] || item > x[last])
        found = false;
    else if (first == last && item == x[last])
    {  
        found = true;
        loc = first;
    }
    else
    {  
        found = false;
        while (!found && first < last)
        {  
            loc = first + 
                int( (item - x[first]) / (x[last] - x[first]) * (last - first) );
            if (item < x[loc])
                last = loc - 1;
            else if (item > x[loc])
                first = loc + 1;
            else  // item == x[loc]
                found = true;
        }
    }
}```
Exercises 12.2

1.

2. H, E, F, G

3. Yes

4. 4

5. A: 3, 2  B: 2, 1  C: 1, 1  D: 1, 0  E, F, G: 0, 0

6. Yes

7.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>array[i]</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
</tbody>
</table>

8. H, I, L, K, G

9. No. Next-to-last level is not completely filled in.

10. 5

11. No. For node E, height of left subtree is 0 and height of right subtree is 2

12.

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>array[i]</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L</td>
</tr>
</tbody>
</table>
Exercises 12.3

1. H, D, B, E, A, F, C, G
2. A, B, D, H, E, C, F, G
3. H, D, E, B, F, G, C, A
10. Preorder: - - A B C
    Postorder: A B - C -

11. Preorder: - - A B C
    Postorder: A B C - -

12. Preorder: / A - B - C - D - E F
    Postorder: A B C D E F - - - - /

13. Preorder: / - - - - A B C D E F
    Postorder: A B - C - D - E - F /

    Postorder: A B C + * D E F + / G H I J * / / *
Exercises 12.4

1.  

Preorder: A, C, R, E, S  
Inorder: A, C, E, R, S  
Postorder: E, S, R, C, A
2. 

```
  R
 /\  
A  C
  \ /
   E
```

Preorder:  R, A, C, E, S  
Inorder:   A, C, E, R, S  
Postorder: E, C, A, S, R

3. 

```
      C
    /\   /
  A  R  E
    \  /
     S
```

Preorder:  C, A, R, E, S  
Inorder:   A, C, E, R, S  
Postorder: A, E, S, R, C

4. 

```
    S
  /
C
  /
A
```

Inorder:   A, C, E, R, S  
Postorder: A, E, R, C, S
5. 


6. 

Preorder: new, const, bool, if, float, main, typedef
Inorder: bool, const, float, if, main, new, typedef
Postorder: bool, float, main, if, const, typedef, new

7. 

Preorder: break, operator, char, else, friend, return, switch
Inorder: break, char, else, friend, operator, return, switch
Postorder: friend, else, char, switch, return, operator, break
8.

```
8.  double
class    long
   int    namespace
      public
         new
```

Preorder:  double, class, long, int, namespace, public, new
Inorder:  class, double, int, long, namespace, new, public
Postorder:  class, int, new, public, namespace, long, double

9.

```
9.  while
    using
       static
          private
             enum
case
```

Preorder:  while, using, static, private, enum, case
Inorder:  case, enum, private, static, using, while
Postorder:  same as inorder
10.  

```
break
operator
if
typedef
else
case
for
do
double
return
while
true
unsigned
void
```

Preorder: break, operator, if, else, case, do, double, for, typedef, return, true, while, unsigned, void
Inorder: break, case, do, double, else, for, if, operator, return, true, typedef, unsigned, void, while
Postorder: double, do, case, for, else, if, true, return, void, unsigned, while, typedef, operator, break

11.  

```
struct
class
```

Preorder: struct, class
Inorder: class, struct
Postorder: same as inorder

12.  

13.  

```
50
50

15
15

5
20
58
15

62
91
62

91

3
8
37
60

7
24

3
8
19
37
55
60

1
7
24
29
```
14. [Diagram of a tree with nodes labeled with numbers and arrows indicating the direction of traversal.]

15. [Diagram of a tree with nodes labeled with numbers and arrows indicating the direction of traversal.]

16. [Diagram of a tree with nodes labeled with numbers and arrows indicating the direction of traversal.]

17. 3, 5, 8, 15, 20, 24, 37, 50, 58, 60, 62, 91

18. 50, 15, 5, 3, 8, 20, 37, 24, 62, 58, 60, 91

19. 3, 8, 5, 24, 37, 20, 15, 60, 58, 91, 62, 50

20.  LLLU/3  
    RUU/5  
    RLU/8  
    RUUU/15  
    RLU/20  
    RLLU/24  
    RUU/37  
    RUUUU/50  
    RLLU/58  
    RLU/60  
    RUUU/62  
    RLU/91  
    RUUUU
21. template <typename DataType>
inline void BST<DataType>::displayPreOrder(ostream & out)
{ displayPreAux(myRoot); }

template <typename DataType>
void BST<DataType>::displayPreAux(BinNodePointer ptr, ostream & out)
{
if (ptr != 0)
{
    out << ptr->data;
    if (ptr->left != 0)
        out << setw(12) << ptr->left->data;
    else
        out << setw(12) << "-";
    if (ptr->right != 0)
        out << setw(12) << ptr->right->data;
    else
        out << setw(12) << "-";
    out << endl;
    displayPreAux(ptr->left, out);
    displayPreAux(ptr->right, out);
}

22. //-- Find level of item in a BST recursively; returns -1 if not found

template <typename DataType>
inline int BST<DataType>::recLevel(const DataType & item)
{ bool found;
    return recLevelAux(myRoot, item, found); }

template <typename DataType>
int BST<DataType>::recLevelAux(BinNodePointer ptr,
const DataType & item, bool & found)
{
if (ptr == 0)
{
    found = false;
    return -1;
}

int level;
if (item < ptr->data)
{
    level = 1 + recLevelAux(ptr->left, item, found);
    if (found) return level; else return -1;
}
else if (item > ptr->data)
{
    level = 1 + recLevelAux(ptr->right, item, found);
    if (found) return level; else return -1;
}
else
{
    found = true;
    return 0;
}
}

23.
//-- Find level of item in a BST iteratively; returns -1 if not found

template <typename DataType>
int BST<DataType>::level(const DataType & item)
{
    bool found = false;
    int count = 0;
    BST<DataType>::BinNodePointer ptr = myRoot;
    while (!found && ptr != 0)
    {
        if (item < ptr->data)
        {
            count ++;
            ptr = ptr->left;
        }
        else if (item > ptr->data)
        {
            count ++;
            ptr = ptr->right;
        }
        else
            found = true;
    }
    if (found)
        return count;
    else
        return -1;
}

24.
//-- Find height of a BST recursively

template <typename DataType>
inline int BST<DataType>::height()
{
    return heightAux(myRoot);
}

template <typename DataType>
int BST<DataType>::heightAux(BinNodePointer ptr)
{
    if (ptr == 0)
        return 0;
    // else
    int leftHeight = heightAux(ptr->left),
    rightHeight = heightAux(ptr->right);
    if (leftHeight > rightHeight)
        return 1 + leftHeight;
    else
        return 1 + rightHeight;
}
25. template <typename DataType>
   inline int BST<DataType>::leafCount()
   { return leafCountAux(myRoot); }

   template <typename DataType>
   int BST<DataType>::leafCountAux(BinNodePointer ptr)
   { if (ptr != 0)
     return 0;
   else if (ptr->left == 0 && ptr->right == 0)
     return 1;
   else
     return leafCountAux(ptr->left) + leafCountAux(ptr->right); }

26. /* Nonrecursive version of inorder() -- use a stack to retain
    addresses of nodes.  Output statement may be replaced with
    other appropriate action when visiting a node.
*/
#include <iostream>
#include <stack>     // or use Stack.h from the text
using namespace std;

   template <typename DataType>
   int BST<DataType>::inorder()
   { stack<DataType> s;
     BST<DataType>::BinNodePointer ptr;
     s.push(myRoot);

     while (!s.empty())
     { ptr = s.top();
       s.pop();

       if (ptr != 0)
       { s.push(ptr->right);
         s.push(ptr);
         s.push(ptr->left);
       }
     else if (!s.empty())
       { ptr = s.top();
         s.pop();
         cout << "Visiting node " << ptr->data << endl;
       }
     }
   }
27. /* Level-by-level traversal -- a queue is used to store
pointers to nodes. Output statement may be replaced with
other appropriate action when visiting a node. */
#include <iostream>
#include <queue>  // or use Queue.h from Chapter 8
using namespace std;

template <typename DataType>
void BST<DataType>::levelByLevel(ostream & out)
{
    queue<BST<DataType>::BinNodePointer> q;
    BST<DataType>::BinNodePointer ptr;
    q.push(myRoot);

    while (!q.empty())
    {
        ptr = q.front();
        q.pop();
        cout << "Visiting node " << ptr->data << endl;

        if (ptr->left != 0)
            q.push(ptr->left);
        if (ptr->right != 0)
            q.push(ptr->right);
    }
}

28. //--- Recursive version of remove()

void BST<DataType>::remove(const DataType & item)
{
    removeAux(item, myRoot);
}

void BST<DataType>::removeAux(const DataType & item,
    BST<DataType>::BinNodePointer & root)
{
    if (root == 0)                // empty BST -- item not found
    {
        cerr << "Item not in the BST\n"
        return;
    }
    //else recursively search for the node containing item
    //and remove it from the:
    if (item < root->data)        // left subtree
        deleteAux(item, root->left);
    else if (item > root->data)   // right subtree
        deleteAux(item, root->right);
else // item found -- delete node
{
    BST<DataType>::BinNodePointer ptr; // auxiliary pointer
    if (root->left == 0) // no left child
    {
        ptr = root->right;
        delete root;
        root = ptr;
    }
    else if (root->right == 0) // left child, but no right child
    {
        ptr = root->left;
        delete root;
        root = ptr;
    }
    else // 2 children
    {
        // find inorder successor
        ptr = root->right;
        while (ptr->left != 0)
        {
            ptr = ptr->left;
        }
        // Move contents of successor to the root of the subtree
        // being examined and delete the successor node.
        root->data = ptr->data;
        deleteAux(ptr->data, root->right);
    }
}

Exercises 12.6

1.

![Diagram of a binary search tree]
5. /* Inorder traversal of a right-threaded BST. Note that recursion is no longer needed. Also, the output statement may be replaced by appropriate processing of a node. */

#include <iostream>

template <typename DataType>
void BST<DataType>::nonrecInorder(ostream & out)
{
    BST<DataType>::BinNodePointer ptr = myRoot;
    while (ptr != 0)
    {
        while (ptr->left != 0)
            ptr = ptr->left;
        out << ptr->data << endl;

        while (ptr->right != 0 && ptr->rightThread)
        {
            ptr = ptr->right;
            out << ptr->data << endl;
        }

        ptr = ptr->right;
    }
}
7. /* Right-thread a BST. */

```cpp
template <typename DataType>
void BST<DataType>::rightThread()
{
    BST<DataType>::BinNodePointer inorderPred = 0;
    rightThreadAux(myRoot, inorderPred);
}
```

```cpp
template <typename DataType>
void BST<DataType>::rightThreadAux(BST<DataType>::BinNodePointer root,
                                   BST<DataType>::BinNodePointer & inorderPred)
{
    if (root != 0) {
        rightThreadAux(root->left, inorderPred);
        if (inorderPred != 0) {
            if (inorderPred->right == 0) {
                inorderPred->right = root;
                inorderPred->rightThread = true;
            }
            inorderPred = root;
            rightThreadAux(root->right, inorderPred);
        }
    }
}
```

8. Algorithm to thread a BST to facilitate preorder traversal involves replacing the null right links of leaves with their preorder successors. The trick is realizing that the preorder successor of node $x$ is the right child of the nearest ancestor — all ancestors are visited before a leaf — of $x$ such that this ancestor's right child is not an ancestor of $x$. The algorithm reduces to:

Perform a preorder traversal of the BST.
Whenever a leaf node (nil left and right links) is encountered:
  Replace the leaf's right link with a thread to its preorder successor.

9. 

![Diagram of a BST with threads](image-url)
10. There are no right threads for preorder traversal. The tree is simply the underlying BST shown in #3 without any thread replacements.

11. Again, there are no right threads for preorder traversal. The tree is simply the underlying BST shown in #4 without any thread replacements.

12. There are no right threads for preorder traversal. The tree is simply the underlying BST shown in #3 without any thread replacements.

13. There are no right threads for preorder traversal. The tree is simply the underlying BST shown in #4 without any thread replacements.
14. Algorithm to carry out a preorder traversal of a BST threaded as described in Exercise 8:

1. Set $ptr =$ root of BST.
2. Set boolean variable $done = (ptr$ is a null pointer).
3. While (!$done$) do the following:
   a. Visit the node pointed to by $ptr$.
   b. While ($ptr$-$>$left is not a null pointer) do the following:
      i. Set $ptr =$ $ptr$-$>$left.
      ii. Visit the node pointed to by $ptr$.
   c. If ($ptr$-$>$right is not a null pointer):
      i. Set $ptr =$ $ptr$-$>$right.
      ii. Visit the node pointed to by $ptr$.
   Else
      Set $done =$ true.

15. Algorithm to find preorder successor of node pointed to by $ptr$ in an inorder-threaded BST.

If ($ptr$-$>$left is not a null pointer):
   Set $preOrdSucc =$ $ptr$-$>$left.
Else do the following:
   a. Set $qptr =$ $ptr$.
   b. While ($qptr$-$>$right is a thread) do the following
      Set $qptr =$ $qptr$-$>$right.
   c. Set $preOrdSucc =$ $qptr$-$>$right.

16. Algorithm to carry out a preorder traversal of an inorder-threaded BST.

1. Set $ptr =$ root of BST.
2. While ($ptr$ is not a null pointer) do the following:
   a. Visit the node pointed to by $ptr$.
   b. Set $ptr =$ preorder successor of $ptr$ as determined by algorithm in Exercise 15.
End while.

17. Algorithm to insert a node into a right-threaded BST.

1. Get a new node pointed to by $temp$.
2. Set $temp$-$>$data =$item$ to be inserted into the BST, make $temp$-$>$left a null pointer, and set $temp$-$>$rightThread = false.
3. If the BST is empty:
   Set its root = $temp$ and make $temp$-$>$right a null pointer.
Else do the following:
   a. Set $done =$ false.
   b. Set $root =$ root of the BST.
c. While (!done) do the following:
   If (temp->data < root->data):
      If (root->left is a null pointer):
         (i) Set root->left = temp and set root->rightThread = false.
         (ii) Set temp->right = root and set temp->rightThread = true.
         (iii) Set done = true.
      Else
         Set root = root->left.
      End if.
   Else if (temp->data > root->data):
      If (root->right is a null pointer):
         (i) Set root->right = temp and set root->rightThread = false.
         (ii) Make temp->right a null pointer and set temp->rightThread = false.
         (iii) Set done = true.
      Else if (root->rightThread):
         (i) Set temp->right = root->right and temp->rightThread = true.
         (ii) Set root->right = temp and root->rightThread = false.
         (iii) Set done = true.
      Else
         Set root = root->right.
      End if
   Else
      Display a message that item is already in the BST.
   End if.
   End while.

18.

```plaintext
int
   int
      int
         short
               int
                     char
                           short
```
19.
20. Two auxiliary algorithms:

*Algorithm to find inorder successor of a node pointed to by p:*

1. Set \( p_{\text{Succ}} = p \rightarrow \text{right} \).
2. if (!\( p \rightarrow \text{rightThread} \))
   
   While \( (p_{\text{Succ}} \rightarrow \text{left} != 0) \)
   
   Set \( p_{\text{Succ}} = p_{\text{Succ}} \rightarrow \text{left} \).
   
   End while.

   End if.

   // \( p_{\text{Succ}} \) points to inorder successor

*Algorithm to find inorder predecessor of a node pointed to by p:*

1. Set \( p_{\text{Pred}} = \) root of BST.
2. While \( (p_{\text{Pred}} \rightarrow \text{left} != 0) \)
   
   Set \( p_{\text{Pred}} = p_{\text{Pred}} \rightarrow \text{left} \).
   
   End while.
3. Set \( p_{\text{Succ}} = \) inorder successor of node pointed to by \( p_{\text{Pred}} \).
4. While \( (p_{\text{Succ}} != p) \)
   
   a. Set \( p_{\text{Pred}} = p_{\text{Succ}} \).
   
   b. Set \( p_{\text{Succ}} = \) inorder successor of node pointed to by \( p_{\text{Pred}} \).
   
   End while.

   // \( p_{\text{Pred}} \) points to inorder predecessor

*Algorithm to delete a node from a right-threaded BST, preserving right-threadedness:*

1. Use \( \text{search2}() \) operation to find node containing \( \text{item} \) to be removed and a pointer \( p \) to this
   
   node and pointer \( \text{parent} \) to its parent.

2. If \( \text{item} \) is not found
   
   Display an error message and terminate this algorithm.

   // Otherwise proceed with the following.

3. Let \( p_{\text{Succ}} \) point to inorder successor of node pointed to by \( p \).

4. If \( (p \rightarrow \text{left} != 0 \&\& p \rightarrow \text{right} != 0 \&\& !p \rightarrow \text{rightThread}) \) // Node has two children
   
   a. // Find parent of inorder successor
      
      If \( (p \rightarrow \text{right} == p_{\text{Succ}}) \)
      
      Set \( \text{parent} = p \).
      
      Else
      
      i. Set \( \text{parent} = p \rightarrow \text{right} \).
      
      ii. While \( (\text{parent} \rightarrow \text{left} != p_{\text{Succ}}) \)
           
          Set \( \text{parent} = \text{parent} \rightarrow \text{left} \).
           
          End while.

   End if.
c. Set $p = pSucc$. // We'll delete node pointed to by $pSucc$ -- it has 0 or 1 child

5. Let $pPred$ point to the inorder predecessor of node pointed to by $p$.

6. If ($pPred->right == p$)
   a. Set $pPred->right = pSucc$.
   b. If ($pSucc != 0$) set $pPred->rightThread = pSucc->rightThread$.
   Else do the following:
      a. Set $subtree = p->left$.
      b. If ($subtree == 0$) set $subtree = p->right$.
      c. If ($parent == 0$)
         Set BST's root = $subtree$.
      Else if ( $parent->left == subtree$)
         Set $parent->left = subtree$.
      Else
         Set $parent->right = subtree$ and $parent->rightThread = p->rightThread$.
   End if

7. Deallocate the node pointed to by $p$.

21.

22.
23. The fully-threaded three is the same as the right-threaded BST in Exercise 4; there are no left threads.

24. The fully-threaded three is the same as the right-threaded BST in Exercise 4; there are no left threads.

25.
26. Algorithm to fully-thread a BST:

Perform an inorder traversal of the BST.

— Whenever a node is visited, set pred = current node.

— Whenever a node x with a null right pointer is encountered, replace this right link with a thread to the inorder successor of x.

— Whenever a node x with a nil left pointer is encountered, replace this left link with a thread to pred.

27. Algorithm to insert a node into a fully-threaded BST:

1. Get a new node pointed to by temp.
2. Set temp->data = item to be inserted, temp->leftThread and temp->rightThread both false.
3. If the BST is empty:
   a. Set BST’s root = temp.
   b. Make temp->left and temp->right both null pointers.
   c. Terminate this algorithm.
// Otherwise do the following:
4.. Set done = false and root = BST’s root.
5. While (!done) do the following:
   a. If (temp->data < root->data)
      If (root->left is null)
         i. Set root->left = temp.
         ii. Set temp->right = root and temp->rightThread = true.
         iii. Make temp->left a null pointer and set temp->leftThread = false.
         iv. Set done = true.
      Else if (root->leftThread)
         i. Set temp->left = root->left and temp->leftThread = true.
         ii. Set temp->right = root and temp->rightThread = true.
         iii. Set root->left = temp and root->leftThread = false.
         iv. Set done = true.
      Else
         Set root = root->left.
   End if.
Else if (temp->data > root->data)
  If (root->right is null)
    i. Set root->right = temp.
    ii. Make temp->right a null pointer and temp->rightThread = false.
    iii. Set temp->left = root and temp->leftThread = true.
    iv. Set done = true.
  Else if (root->rightThread)
    i. Set temp->right = root->right and temp->rightThread = true.
    ii. Set root->right = temp and root->rightThread = false.
    iii. Set temp->left = root and temp->leftThread = true.
    iv. Set done = true.
  Else
    Set root = root->right.
  End if.
Else display a message that item is already in the BST.

28. Algorithm to find parent of node x in a fully-threaded BST:

1. Set ptr = x.
2. While (ptr != 0 && !ptr->leftThread)
   Set ptr = ptr->left.
   End while.
3. If (ptr != 0 && ptr->left->right != x)
   Set parent = ptr->left.
   Else:
   a. Set ptr = x.
   b. While (ptr != 0 && !ptr->rightThread)
      Set ptr = ptr->right.
      End while.
   c. If (ptr is a null pointer)
      Make parent a null pointer.
      Else
      Set parent = ptr->right.
      End if.
   End if.
   End if.
### Exercises 12.7

1. 

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>92</td>
</tr>
<tr>
<td>7</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>60</td>
</tr>
</tbody>
</table>

2. 

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>92</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>80</td>
</tr>
</tbody>
</table>

3. 

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>1</td>
<td>93</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>60</td>
</tr>
</tbody>
</table>
7. The following solution uses a `vector< list<ElementType> >` to store the hash table. Converting it to use an array of linked lists (for the linked list class considered in Chapter 6) is straightforward.

```cpp
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <list>
#include <algorithm>

template <typename ElementType>
class HashTable
{
public:
```

8. The following solution uses a `vector< list<ElementType> >` to store the hash table. Converting it to use an array of linked lists (for the linked list class considered in Chapter 6) is straightforward.

```cpp
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <list>
#include <algorithm>

template <typename ElementType>
class HashTable
{
public:
```
HashTable(int n = 0);
/*---------------------------------------
Constructor
Precondition: n >= 0 is the number of items to be stored in the table.
Postcondition: Hash table with n elements (default n = 0) has been constructed.
--------------------------------------*/

HashTable(const HashTable & original);
/*---------------------------------------
Copy Constructor
Precondition: A copy of HashTable object original is needed.
Postcondition: A copy of original has been constructe.
--------------------------------------*/

~HashTable();
/*---------------------------------------
Destructor
Precondition: This object must be destroyed.
Postcondition: All dynamically-allocated memory in object has been reclaimed.
--------------------------------------*/

void insert(ElementType item, int key);
/*------------------------------
Insert operation
Precondition: item is to be inserted into this hash table; key is an integer used by hash function to find its location.
Postcondition: item has been inserted into the hash table at the location determined by applying hash function to key.
------------------------------*/

int search(ElementType item, int key);
/*------------------------------
Search operation
Precondition: Hash table is to be searched for item; key is item's numeric key.
Postcondition: Location of item in table is returned, -1 if not found.
------------------------------*/

void display(ostream & out);
/*------------------------------
Output operation
Precondition: ostream out is open; << is defined for ElementType.
Postcondition: Contents of hash table have been output to out.
------------------------------*/
private:
    vector< list<ElementType> > myTable;
    //--- Hash function for a numeric key
    unsigned hash(int key);
    /*
     * Hash function
     * Precondition: None.
     * Postcondition: An index of allocation in the hash table is returned.
     ************************************************************************/

    //--- Definition of constructor
    template <typename ElementType>
    inline HashTable<ElementType>::HashTable(int n)
    {
        list<ElementType> emptyList;
        for (int i = 1; i <= n; i++)
            myTable.push_back(emptyList);
    }

    //--- Definition of copy constructor
    template <typename ElementType>
    inline HashTable<ElementType>::HashTable(
        const HashTable<ElementType> & original)
    {
        myTable = original.myTable;
    }

    //--- Definition of destructor
    template <typename ElementType>
    inline HashTable<ElementType>::~HashTable()
    { } // vector and list destructors take care of this

    //--- Definition of insert()
    template <typename ElementType>
    inline void HashTable<ElementType>::insert(ElementType item, int key)
    {
        if (search(item, key) < 0)
            myTable[hash(key)].push_front(item);
        else
            cerr << item << " already in table.\n";
    }

    //--- Definition of search()
    template <typename ElementType>
    int HashTable<ElementType>::search(ElementType item, int key)
    {
        int loc = hash(key);
        if (find(myTable[loc].begin(), myTable[loc].end(), item)
            != myTable[loc].end())
            return loc;
        else
            return -1;
    }
/--- Definition of display()
template <typename ElementType>
void HashTable<ElementType>::display(ostream & out)
{
    for (int i = 0; i < myTable.size(); i++)
    {
        out << setw(2) << i << " : ";
        list<ElementType>::iterator it;
        for (it = myTable[i].begin(); it != myTable[i].end(); it++)
            out << *it << " ";
        out << endl;
    }
}

//-- Definition of hash()
template <typename ElementType>
inline unsigned HashTable<ElementType>::hash(int number)
{
    const unsigned
    MULTIPLIER = 25173U,
    ADDEND = 13849U,
    MODULUS = 65536U;
    unsigned x = (MULTIPLIER * number + ADDEND) % MODULUS % myTable.size();
    return x;
}