Chapter 15: Trees

Exercises 15.1

1. Morse is not immediately decodable because its codes do not have unique prefixes. For example, the given bit string 100001100 can be decoded as:
   10 000 110 0 ⇒ N S GE
   and also as
   100 00 11 00 ⇒ D I M I
   and as several other strings.

2. BAD
3. DEAD
4. CAD
5. BEADED

Note that for #6 and #7, several different codes are possible, depending on whether the nodes are placed in the left or the right subtrees.

6. int 10
   main 11
   while 000
   if 01
   for 001

7. a 111
   b 100
   c 1100
   d 1101
   e 0
   f 1010
   g 10110
   h 10111

8. 1010 0 0 1101 111 1101 0 111 1010 111 10110 0 1101 10111 111 10110
    F E E D A D E A F A G E D H A G

9. One possible solution is the following:
   case 000011
   class 011
   do 01001
   else 0101
   false 00010
   for 100
   goto 000010
   if 101
   int 110
   main 111
   static 010000
   struct 00000
   switch 010001
   true 00011
   while 001
Exercises 15.2

1.
2.

```
bool 0
    new
        return
            struct
                case
                return
                while

new -1
    bool 0
        return
            struct
                case
                return
                while

new -2
    bool -1
        return
            struct
                case
                return
                while

new 0
    bool 0
        return
do
    while

new -1
    bool 0
        return
            struct
                case
                return
                while

new 0
    bool 0
        return
            struct
                case
                return
                while

new 1
    bool -1
        return
            struct
                case
                return
                while

new 0
    case
        struct
            bool
            enum
            return
            while
```
3.

```
0 -> do -> long
     |     0
     v     v
-1 -> do -> long
     |     0
     v     v
-2 -> do -> long
     |     0
     v     v
0   -> do -> new
     |     0
     v     v
1   -> long
     |     0
     v     v
0   -> new
     |     0
     v     v
0   -> do
     |     0
     v     v
auto
     |     0
     v     v
0   -> long
     |     0
     v     v
1   -> auto
     |     0
     v     v
0   -> operator
     |     0
     v     v
0   -> long
     |     0
     v     v
0   -> new
     |     0
     v     v
auto
     |     0
     v     v
0   -> int
     |     0
     v     v
operator
     |     0
     v     v
0   -> namespace
     |     0
     v     v
operator
```
4.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned</td>
<td>short</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>long</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>short</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>short</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>short</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>char</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>long</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>unsigned</td>
</tr>
</tbody>
</table>
```

– 282 –
7.

8.
11.

Balanced Subtree Before Insertion

Unbalanced Subtree After Insertion

Rebalanced Subtree After Simple Left Rotation
Case 3: Balanced Subtree before Insertion

Unbalanced Subtree after Insertion
13.

**Case 1:** Balanced Subtree Before Insertion

-1

A

0

B

Unbalanced Subtree After Insertion

-2

A

+1

B

0

C

Balanced Subtree After Left-Right Rotation

A

C

B

0

C

0

B

**Case 2:** Balanced Subtree before Insertion

-1

A

0

B

0

C

\[ R(C) \]

\[ R(B) \]
Chapter 15

Balanced Subtree after Right-Left Rotation

Case 3:
Balanced Subtree before Insertion
template <typename ElementType>
class AVLTree
{
  public:
    /*** FUNCTION MEMBERS
    // Usual BST operations

  private:
    /*** Node */

  class AVLNode
  {
    short int balanceFactor;
    ElementType data
    AVLNode * left;
    AVLNode * right;
  }

  //-- AVLNode constructors
  AVLNode()
  {
    balanceFactor = 0;
    left = right = 0;
  }

  AVLNode(ElementType item)
  {
    balanceFactor = 0;
    data = item;
    left = right = 0;
  }
}
typedef AVLNode * AVLNodePointer;

/*** DATA MEMBERS ***/
AVLNodePointer myRoot;

The insert operation requires extensive modification from that for BSTs in general. Here is an algorithm:

/* Algorithm to insert an item item into an AVL tree */

1. If the BST is empty
   Create a node containing item with both left and right links null and balance factor = 0,
   and return from this algorithm.

   // Search for insertion point; ptrA keeps track of the most recent ancestor with balance factor
   // ±1 and ptrAParent points to the parent of ptrA. Pointer parent follows locPtr along the
   // search path.
2. Set pointers ptrAParent and parent to null, ptrA and locPtr to the root of the BST,
3. While (locPtr != 0) do the following:  // Search for insertion point for item
   If (locPtr->balanceFactor != 0)
      a. Set ptrA = locPtr and ptrAParent = parent.
      b. If (item < locPtr->data)  // go left
         Set parent = locPtr and locPtr = locPtr->left.
      Else  // go right
         Set parent = locPtr and locPtr = locPtr->right.
      Else  // item is in the tree
         Display message that item is in the tree and terminate the algorithm.
   End if.
   End if.
End while.

   // item is not in the tree. Insert it as a child of parent.
4. Get a node pointed to by newPtr and containing item in its data part, both left and right
   links set to null pointers, and balance factor = 0.
5. If (item < parent->data)  // insert as left child
   Set parent->left = newPtr.
Else  // insert as right child
   Set parent->right = newPtr.
End if.

   // Rebalance the tree by adjusting balance factors of nodes on the path from ptrA to parent;
   // all of these nodes have balance factor 0, which will change to ±1. direction is +1 or
   // according as item is inserted in the left subtree or the right subtree of ptrA.
6. If (item < ptrA->data)
   Set locPtr = ptrA->left, ptrB = p, and direction = +1,
Else
   Set locPtr = ptrA->right, ptrB = p, and direction = −1,
End if.

7. While (locPtr != newPtr) do the following:
   If (item < locPtr->data) // height of left subtree increases by 1
     Set locPtr<balanceFactor = +1 and locPtr = locPtr->left.
   Else // height of right subtree increases by 1
     Set locPtr<balanceFactor = −1 and locPtr = locPtr->right.
   End if.
End while.

// Check if tree is unbalanced
8. If (ptrA->balanceFactor == 0) // tree is still balanced
   Set ptrA->balanceFactor = d and terminate the algorithm.
9. If (ptrA->balanceFactor + d == 0) // tree is still balanced
   Set ptrA->balanceFactor = 0 and terminate the algorithm.

// Tree is unbalanced; perform suitable rotation
10. If (direction = +1) // left imbalance
    If (ptrB->balanceFactor = +1) // LL rotation
      a. Set ptrA->left = ptrB->right.
      b. Set ptrB->right = ptrA.
      c. ptrA->balanceFactor = ptrB->balanceFactor = 0.
    Else // LR rotation
      a. Set ptrC = ptrB->right, cptrLeft = c->left, cptrRight = c->right.
      b. Set ptrB->right = cptrLeft.
      c. Set ptrA->left = cptrRight,
      d. Set ptrC->left = ptrB.
      e. Set ptrC->right = ptrA.
      f. If (ptrC->balanceFactor == +1) // LR - case 2
         Set ptrA->balanceFactor = −1, ptrB->balanceFactor = 0.
      Else if (ptrC->balanceFactor == −1) // LR - case 3
         Set ptrB->balanceFactor = +1, ptrA->balanceFactor = 0.
      Else // LR - case 1
        i. Set ptrC->balanceFactor = 0.
        ii. Set ptrB = ptrC,
    End if.
End if.

Else
   // --- Right imbalance -- This is symmetric to the above for left imbalance.
   // ---
End if.
// Subtree with root ptrB has been rebalanced and is the new subtree of ptrAParent (for which the original // root was ptrA).

11. If (ptrAParent == 0)  
   Set BST's root equal to ptrB.  
   Else if (ptrA == ptrAParent->left)  
      Set ptrAParent->left == ptrB.  
   Else // ptrA = ptrAParent->right  
      Set ptrAParent->right = ptrB.  
   End if.

Deletion from an AVL tree is more difficult. If there are not many deletions, one might use lazy deletion: instead of deleting the node and modifying rebalancing the tree, we mark the node as deleted. Then, during operations that require searching the tree, we indicate that the item is not in the tree if the desired node is found but marked. This keeps the balance of a tree from being disturbed by deletions.

However, if many deletions will be done, then we can delete the node as for a BST and then rebalance the tree. A search of the Internet for AVL deletion will turn up several places where this deletion + rebalancing is described.

See, for example, http://www.cmcrossroads.com/bradapp/ftp/ for an AVLTree class.

Exercises 15.3
Chapter 15

7. 

8. 

9. 

10. 

11. 

12. 

13. 

14.
22. An algorithm to delete an element from a 2-3-4 tree must deal with the rebalancing that may be necessary after an element is removed. Recall that insertion sometimes results in a splitting of a node and that split may propagate up or down the tree, depending on whether one is performing a top-down insertion or a bottom-in insertion. In a like manner, deleting an element from a 2-3-4 tree may cause perturbations elsewhere in the tree. There are two main considerations:

(1) Deletion of an element in a 2-node necessitates the replacement of that value with one from its children.
(2) Deletion of a 3- or 4-node element may necessitate the merging of children (because in effect, the elements of a 3- or 4-node serve as separator values for their children).

Note that in the latter case, a merge may result in overflow, in which case, we actually need a merge-split operation: merge sibling nodes (causing overflow) and then split the resulting node.

To streamline the deletion process, we reduce the deletion of an arbitrary element to the deletion of an element that is in a leaf node. If the element to be deleted is in a 3-node or 4-node leaf, then after deletion, we have a 2-node or 3-node leaf. No restructuring is then required. To avoid propagation of restructuring, we need to ensure that the element to be deleted is in a 3-node or 4-node leaf before deletion. That is, restructuring is done as we search for the element to be deleted.

The restructuring process rests on the search for the item to be deleted as progressing through only 3-node and 4-nodes. If we move through a 2-node, then restructuring is required. The following steps detail the process. Assume that we are at a node $p$ and will next move to a child $q$.

- If $p$ is a leaf, the element to be deleted is either in this node or is not in the tree:
  - Delete it; no restructuring is needed.

- If $q$ is a 3-node or 4-node: Move to this node.

- If $q$ is a 2-node ($x$ & $y$) and its nearest sibling $r$ is also a 2-node with children $x$ and $y$:
  - If $p$ is a 2-node with children $q$ and $r$, merge $q$, $p$ and $r$ into a 4-node with children $x$, $y$, $v$, and $w$.
  - If $p$ is a 3-node (or 4-node), merge $q$, $p$, and $r$ into a 4-node with children $x$, $y$, $v$, and $w$ (as before), but this 4-node is a child of the 2-node (or 3-node) containing the remaining elements of $p$'s original node.

- If $q$ is a 2-node ($x$ & $y$) and its nearest sibling $r$ is also a 3-node (or 4-node):
  - Transform $q$ into a 3-node by taking the appropriate separator out of $p$ and one of its children out of $r$. 

-
23. An algorithm to delete a node from a red-black tree must deal with the rebalancing that may be necessary after an element is removed. As explored in detail in Exercise 22, restructuring may be confined if one ensures that the node to be deleted is a leaf node, in this case one with a red link from its parent. Again, rebalancing is done as we search for the element to be deleted. As in Exercise 22, the insertion transformations (rotations) are performed in reverse. However, here the cost is cheaper since color changes take care of most transformations.

24-25. Developing complete and correct classes for red-black trees and RB trees is a nontrivial task and can be assigned to students who want (or need) a challenge. Source code can be found by searching the Internet; for example, see http://www.cmcrossroads.com/bradapp/ftp/.

26. This is an interesting project to challenge students that is not as difficult as those in Exercises 24 and 25. They must decide how to have the user enter the structure of the original tree and store the data in a binary tree as described in the text and then modify the binary tree operations so they perform the desired tree operations.