AVL (HEIGHT-BALANCED) TREES

In Chapter 19, you learned how to build and manipulate a binary search tree. The performance of the search algorithm on a binary search tree depends on how the binary tree is built. The shape of the binary search tree depends on the data set. If the data set is sorted, then the binary search tree is linear and so the search algorithm would not be efficient. On the other hand, if the tree is nicely built, then the search would be fast. In fact, the smaller the height of the tree, the faster the search. Therefore, we want the height of the binary search tree to be as small as possible. This section describes a special type of binary search tree, called the **AVL tree**, (also called the **height-balanced tree**), in which the resulting binary search tree is nearly balanced. AVL trees were introduced by the mathematicians G. M. Adeĺson-Veĺskiĭ and E. M. Landis in 1962 and are so named in their honor.

We begin with the following definition.

Definition: A **perfectly balanced** binary tree is a binary tree such that:

i. The heights of the left and right subtrees of the root are equal.
ii. The left and right subtrees of the root are perfectly balanced binary trees.

Figure AVL-1 shows a perfectly balanced binary tree.

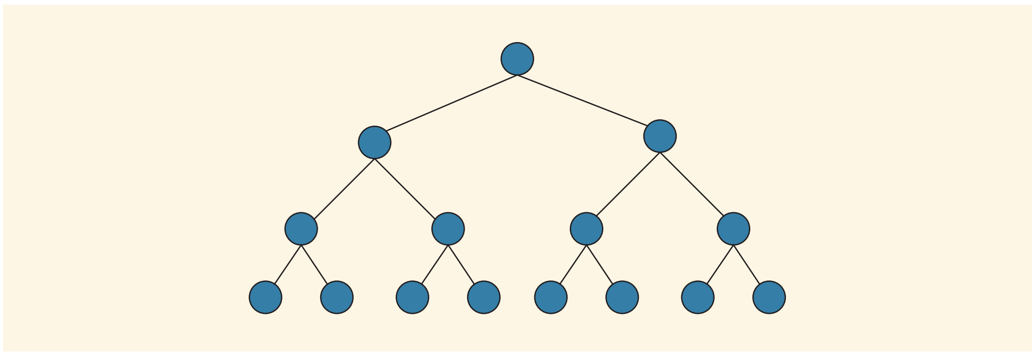

FIGURE AVL-1 Perfectly balanced binary tree

Let T be a binary tree and x be a node in T. If T is perfectly balanced, then from the definition of the perfectly balanced tree, it follows that the height of the left subtree of x is the same as the height of the right subtree of x.

It can be proved that if T is a perfectly balanced binary tree of height h, then the number of nodes in T is 2^h-1. From this it follows that if the number of items in the data set is not equal to 2^h-1, for some non-negative integer h, then we cannot construct a perfectly balanced binary tree. Moreover, perfectly balanced binary trees are a too stringent refinement.

Definition: An **AVL tree** (or **height-balanced tree**) is a binary search tree such that:

i. The height of the left and right subtrees of the root differ by, at most, 1.
ii. The left and right subtrees of the root are AVL trees.

Figures AVL-2 and AVL-3 give examples of AVL and non-AVL trees.

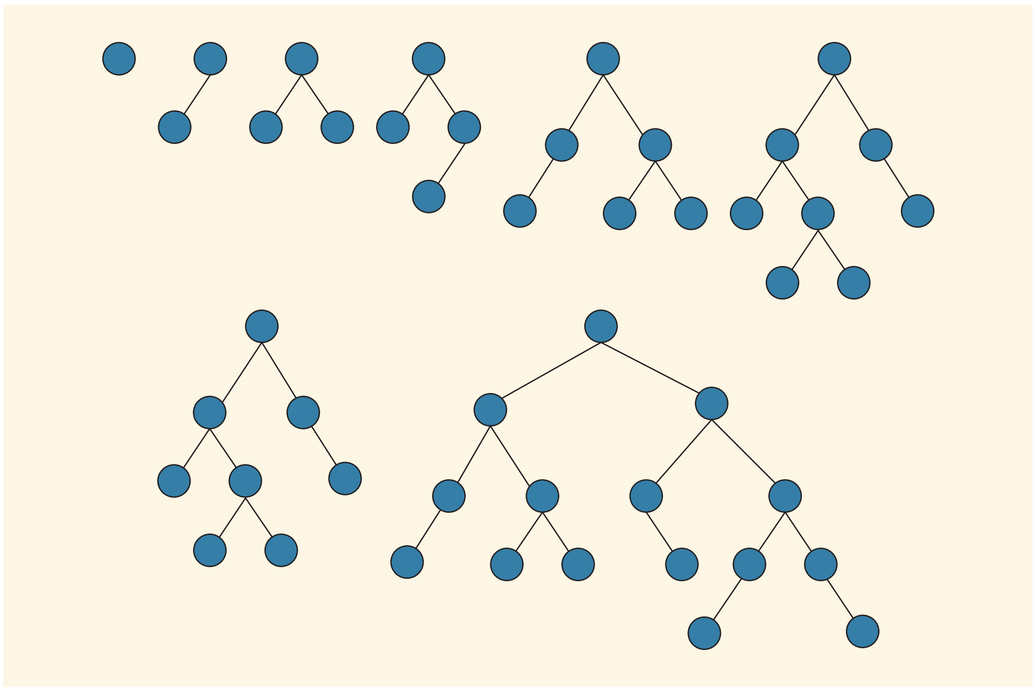

FIGURE AVL-2 AVL trees

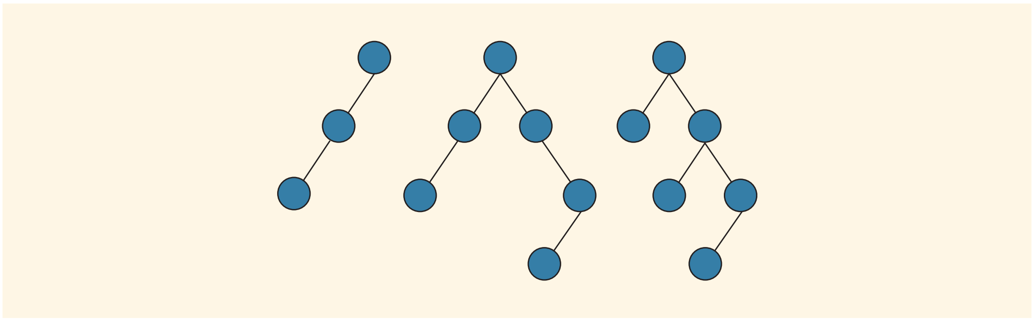

FIGURE AVL-3 Non-AVL trees

Let x be a node in a binary tree. Let x_l denote the height of the left subtree of x, and x_r denote the height of the right subtree of x.

Theorem: Let T be an AVL tree and x be a node in T. Then, $|x_r - x_l| \leq 1$, where $|x_r - x_l|$ denotes the absolute value of $x_r - x_l$.

Let x be a node in the AVL tree T.

1. If $x_l > x_r$, we say that x is **left high**. In this case, $x_l = x_r + 1$.
2. If $x_l = x_r$, we say that x is **equal high**.
3. If $x_r > x_l$, we say that x is **right high**. In this case, $x_r = x_l + 1$.

Definition: The **balance factor** of x, written $bf(x)$, is defined by $bf(x) = x_r - x_l$.

Let x be a node in the AVL tree T. Then:

1. If x is left high, then $bf(x) = -1$.
2. If x is equal high, then $bf(x) = 0$.
3. If x is right high, then $bf(x) = 1$.

Definition: Let x be a node in a binary tree. We say that the node x **violates the balance criteria** if $|x_r - x_l| > 1$, that is, if the height of the left and right subtrees of x differ by more than 1.

From the previous discussion, it follows that in addition to the data and pointers of the left and right subtrees, one more thing associated with each node x in the AVL tree T is the balance factor of x. Thus, every node must keep track of its balance factor. To make the algorithms efficient, we store the balance factor of each node in the node itself.

The following defines the node of an AVL tree:

```
   //Definition of the node
template <class elemType>
struct AVLNode
{
    elemType info;
    int bfactor; // balance factor
    AVLNode<elemType> *lLink;
    AVLNode<elemType> *rLink;
};
```

Because an AVL tree is a binary search tree, the search algorithm for an AVL tree is the same as the search algorithm for a binary search tree. Other operations on AVL trees, such as finding the height, determining the number of nodes, checking whether the tree is empty, tree traversal, and so on, can be implemented exactly the same way they are implemented on binary trees. However, item insertion and deletion operations on AVL trees are somewhat different than the ones discussed for binary search trees, because after inserting (or deleting) a node from an AVL tree, the resulting binary tree must be an AVL tree. Next, we describe these operations.

Insertion into AVL Trees

To insert an item in an AVL tree, first we search the tree and find the place where the new item is to be inserted. Because an AVL tree is a binary search tree, to find the place for the new item, we can search the AVL tree using a search algorithm similar to the one designed for binary search trees. If the item to be inserted is already in the tree, then the search ends at a nonempty subtree. Because duplicates are not allowed, in this case, we can output an appropriate error message. Suppose that the item to be inserted is not in the AVL tree. Then, the search ends at an empty subtree and we insert the item in that subtree. After inserting the new item in the tree, the resulting tree might not be an AVL tree. Thus, we must restore the tree's balance criteria. This is accomplished by traveling the same path (back to the root node) that was followed when the new item was inserted in the AVL tree. The nodes on this path (back to the root node) are visited and either their balance factors are changed, or we might have to reconstruct part of the tree. We illustrate these cases with the help of the following examples.

NOTE In Figures AVL-4 through AVL-15, for each node we show only the data stored in the node. Moreover, an equal sign (=) on the top of a node indicates that the balance factor of this node is 0; the less-than symbol (<) indicates that the balance factor of this node is −1; and the greater-than symbol (>) indicates that the balance factor of this node is 1.

Consider the AVL tree of Figure AVL-4.

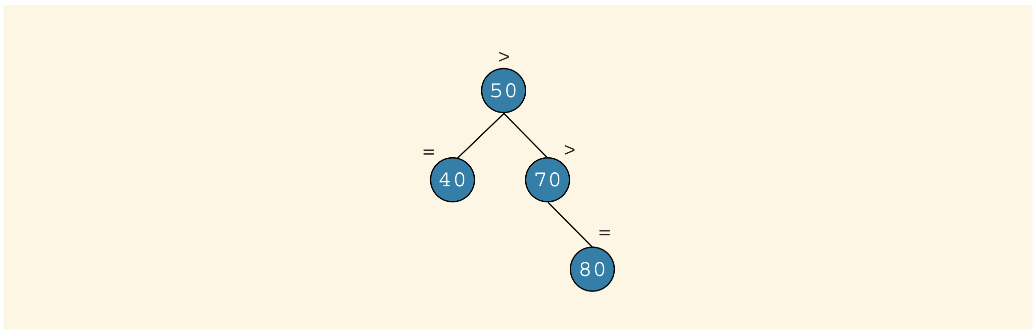

FIGURE AVL-4 AVL tree before inserting 90

Let us insert 90 into this AVL tree. We search the tree, starting at the root node, to find the place for 90. The dotted arrow shows the path traversed. We insert the node with info 90 and obtain the binary search tree of Figure AVL-5.

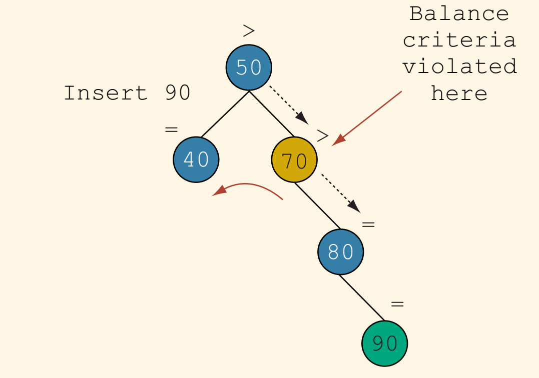

FIGURE AVL-5 Binary search tree of Figure AVL-4 after inserting 90; nodes other than 90 show their balance factors before insertion

The binary search tree of Figure AVL-5 is not an AVL tree. So, we backtrack and go to node 80. Prior to insertion, bf(80) was 0. Because the new node was inserted into the (empty) right subtree of 80, we change its balance factor to 1 (not shown in the figure). Now we go back to node 70. Prior to insertion, bf(70) was 1. After insertion, the height of the right subtree of 70 is increased; thus, we see that the subtree with the root node 70 is not an AVL tree. In this case, we reconstruct this subtree (this is called rotating the tree at root node 70). We thus obtain the AVL tree as shown in Figure AVL-6.

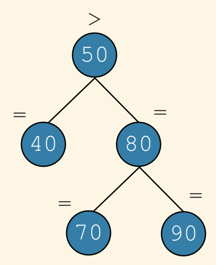

FIGURE AVL-6 AVL tree of Figure AVL-4 after inserting 90 and adjusting the balance factors

The binary search tree of Figure AVL-6 is an AVL tree.

Now consider the AVL tree of Figure AVL-7.

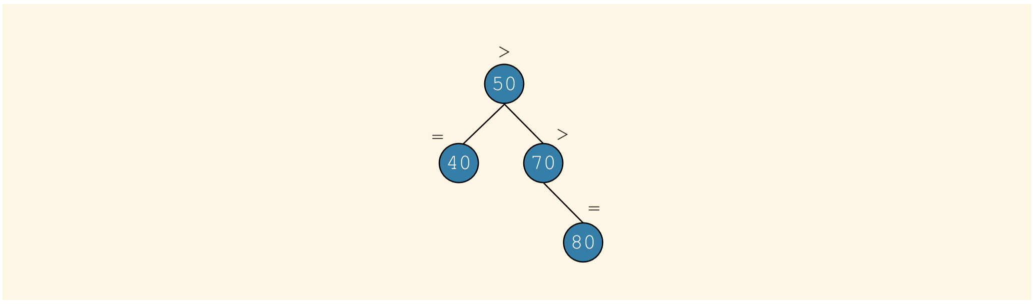

FIGURE AVL-7 AVL tree before inserting 75

Let us insert 75 into the AVL tree of Figure AVL-7.

As before, we search the tree starting at the root node. The dotted arrows show the path traversed. After inserting 75, the resulting binary search tree is as shown in Figure AVL-8.

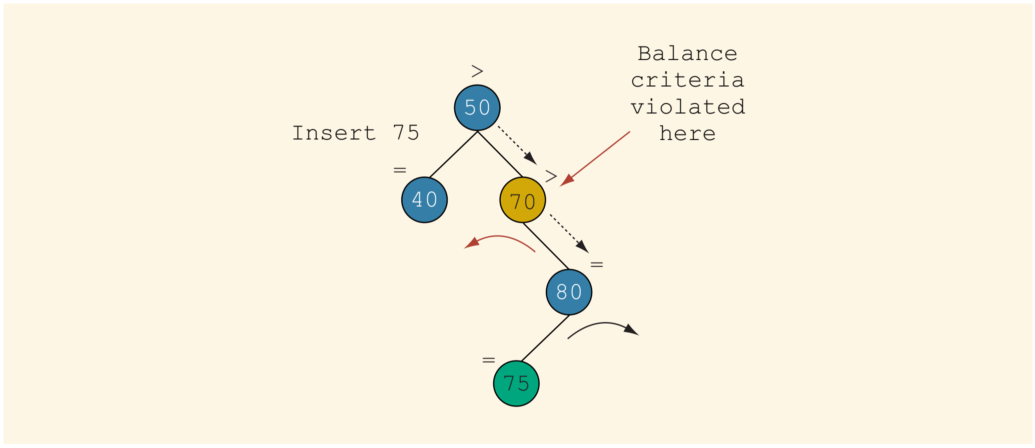

FIGURE AVL-8 Binary search tree of Figure AVL-7 after inserting 75; nodes other than 75 show their balance factors before insertion

After inserting 75, we backtrack. First, we go to node 80 and change its balance factor to -1. The subtree with the root node 80 is an AVL tree. Now we go back to 70. Clearly, the subtree with the root node 70 is not an AVL tree. So, we reconstruct this subtree. In this case, we first reconstruct the subtree at root node 80, and then reconstruct the subtree at root node 70 to obtain the binary search tree as shown in Figure AVL-9. (These constructions, that is, rotations, are explained in the next section, "AVL Tree Rotations.")

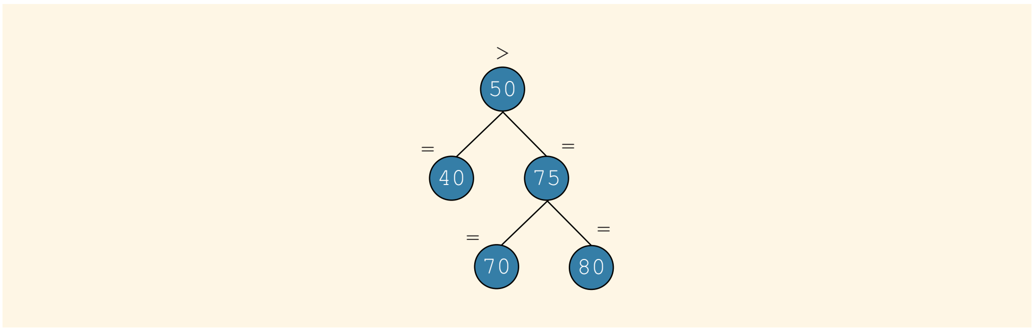

FIGURE AVL-9 AVL tree of Figure AVL-7 after inserting 75 and adjusting the balance factors

After reconstruction, the root of the constructed subtree is 75.

Notice that in Figures AVL-6 and AVL-9, after reconstructing the subtrees at the nodes, the subtrees no longer grew in height. At this point, to the remaining nodes on the path back to the root node of the tree, we usually send the message indicating that, overall, the tree did not gain any height. Thus, the remaining nodes on the path do not need to do anything.

Next, consider the AVL tree of Figure AVL-10.

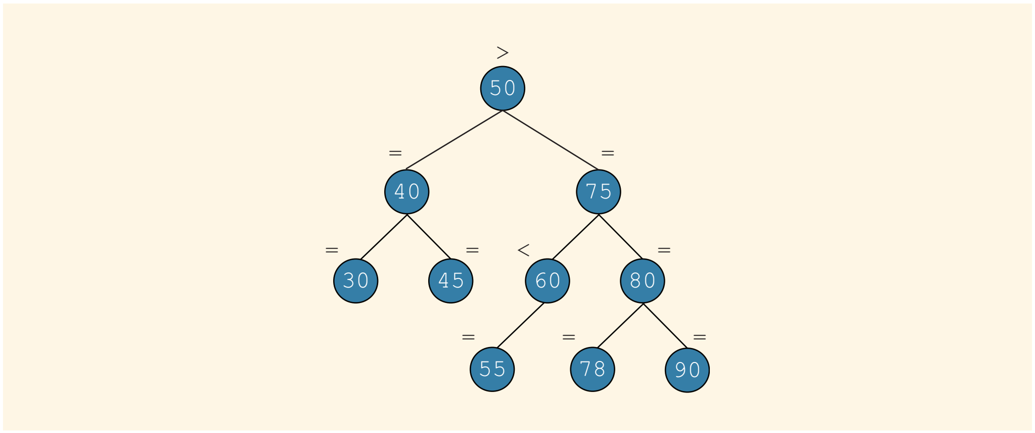

FIGURE AVL-10 AVL tree before inserting 95

Let us insert 95 into this AVL tree. We search the tree and insert 95, as shown in Figure AVL-11.

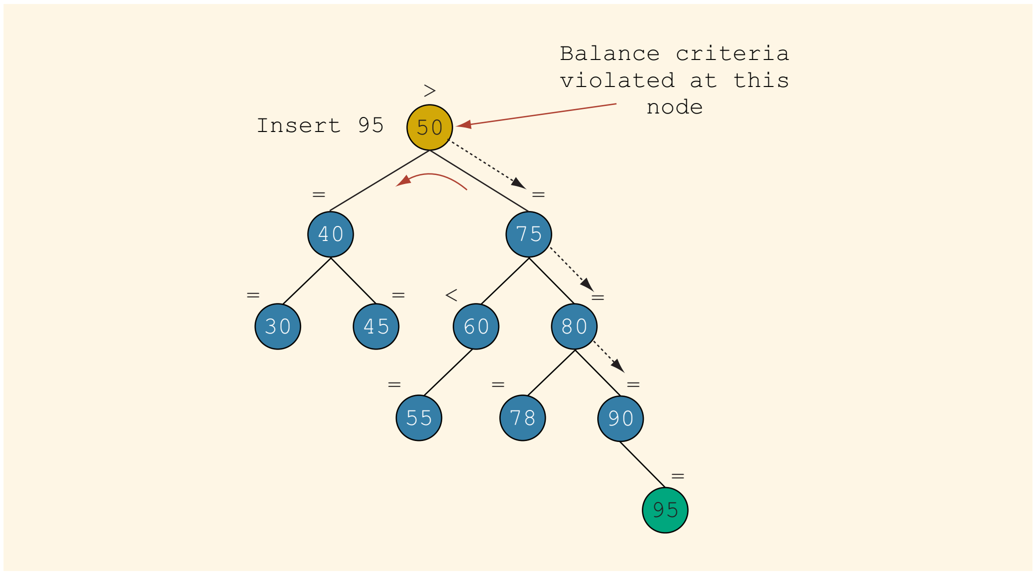

FIGURE AVL-11 Binary search tree of Figure AVL-10 after inserting 95; nodes other than 95 show their balance factors before insertion

After inserting 95, we see that the subtrees with the root nodes 90, 80, and 75 are still AVL trees. When backtracking the path, we simply adjust the balance factors of these nodes (if needed). However, when we backtrack to the root node, we discover that the tree at this node is no longer an AVL tree. Prior to insertion, `bf(50)` was 1, that is, its right subtree was higher than its left subtree. After insertion, the subtree grew in height, thus violating the balance criteria at 50. So, we reconstruct the binary search tree at node 50. In this case, the tree is reconstructed, as shown in Figure AVL-12.

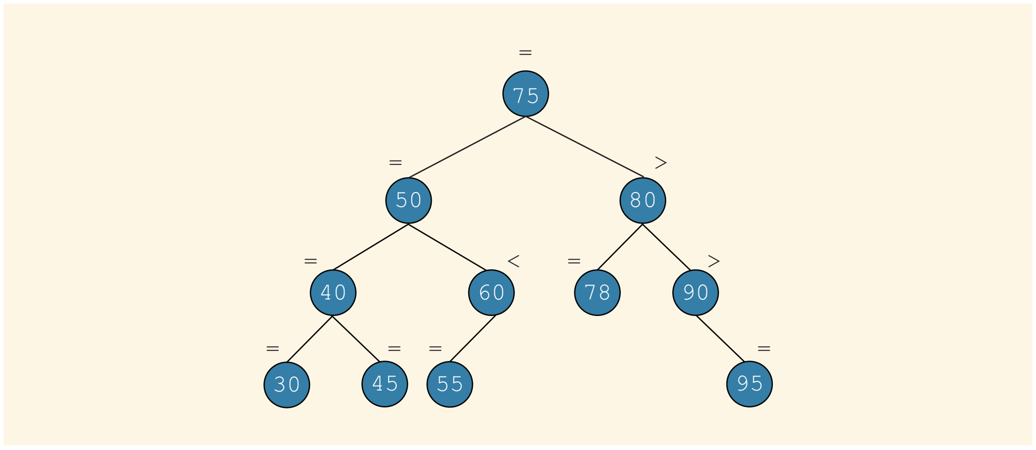

FIGURE AVL-12 AVL tree of Figure AVL-10 after inserting 95 and adjusting the balance factors

Before discussing the general algorithms for reconstructing (rotating) a subtree, let us consider one more case. Consider the AVL tree as shown in Figure AVL-13.

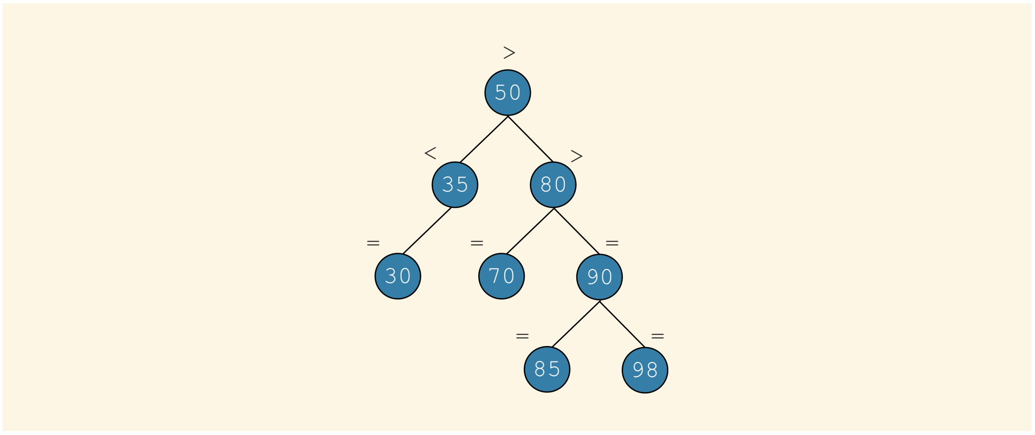

FIGURE AVL-13 AVL tree before inserting 88

Let us insert 88 into the tree of Figure AVL-13. Following the insertion procedure as described previously, we obtain the binary search tree as shown in Figure AVL-14.

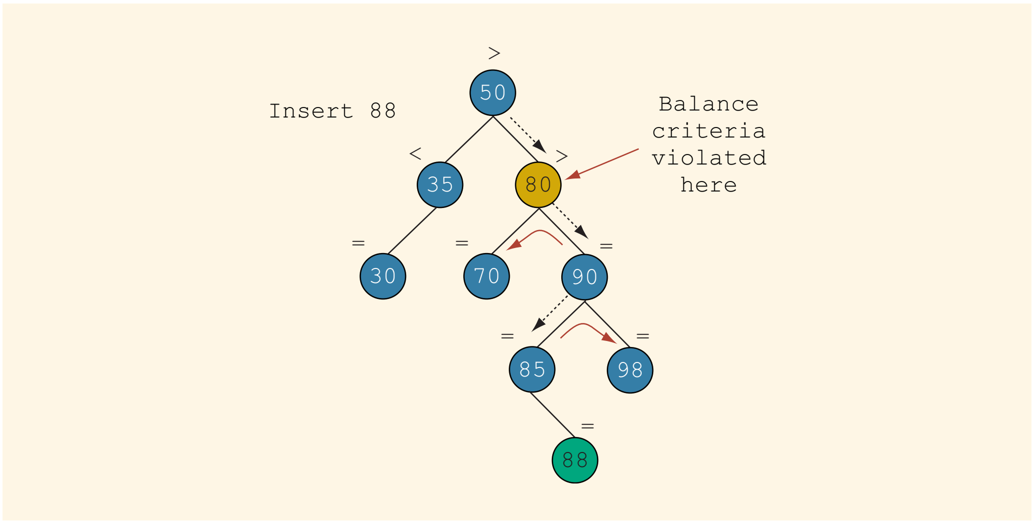

FIGURE AVL-14 Binary search tree of Figure AVL-13 after inserting 88; nodes other than 88 show their balance factors before insertion

As before, we now backtrack to the root node. We adjust the balance factors of nodes 85 and 90. When we visit node 80, we discover that at this node, we need to reconstruct the subtree. In this case, the subtree is reconstructed, as shown in Figure AVL-15.

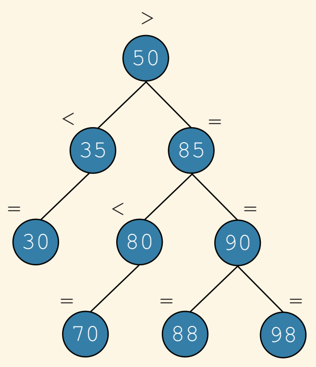

FIGURE AVL-15 AVL tree of Figure AVL-13 after inserting 88 and adjusting the balance factors

As before, after reconstructing the subtree, the entire tree is balanced. So, for the remaining nodes on the path back to the root node, we do not do anything.

The previous examples demonstrate that if part of the binary search tree requires reconstruction, then after reconstructing that part of the binary search tree, we can ignore the remaining nodes on the path back to the root node. Also, after inserting the node, the reconstruction can occur at any node on the path back to the root node.

AVL Tree Rotations

We now describe the reconstruction procedure, called **rotating** the tree. There are two types of rotations: **left rotation** and **right rotation**. Suppose that the rotation occurs at node *x*. If it is a left rotation, then certain nodes from the right subtree of *x* move to its left subtree; the root of the right subtree of *x* becomes the new root of the reconstructed subtree. Similarly, if it is a right rotation at *x*, certain nodes from the left subtree of *x* move to its right subtree; the root of the left subtree of *x* becomes the new root of the reconstructed subtree.

Case 1: Consider Figure AVL-16.

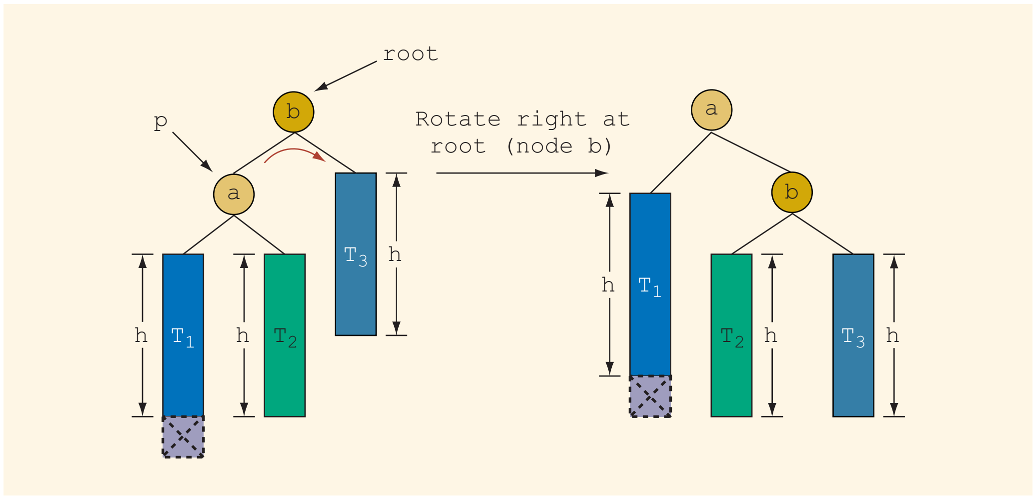

FIGURE AVL-16 Right rotation at *b*

In Figure AVL-16, subtrees T_1, T_2, and T_3 are of equal height, say, *h*. The dotted rectangle shows an item insertion in T_1, causing the height of the subtree T_1 to increase by 1. The subtree at node *a* is still an AVL tree, but the balance criteria is violated at the root node. We note the following in this tree. Because the tree is a binary search tree:

- Every key in T_1 is smaller than the key in node *a*.
- Every key in T_2 is larger than the key in node *a*.
- Every key in T_2 is smaller than the key in node *b*.

Therefore:

1. We make T_2 (the right subtree of node *a*) the left subtree of node *b*.
2. We make node *b* the right child of node *a*.
3. Node *a* becomes the root node of the reconstructed tree, as shown in Figure AVL-16.

Case 2: This case is a mirror image of Case 1. See Figure AVL-17.

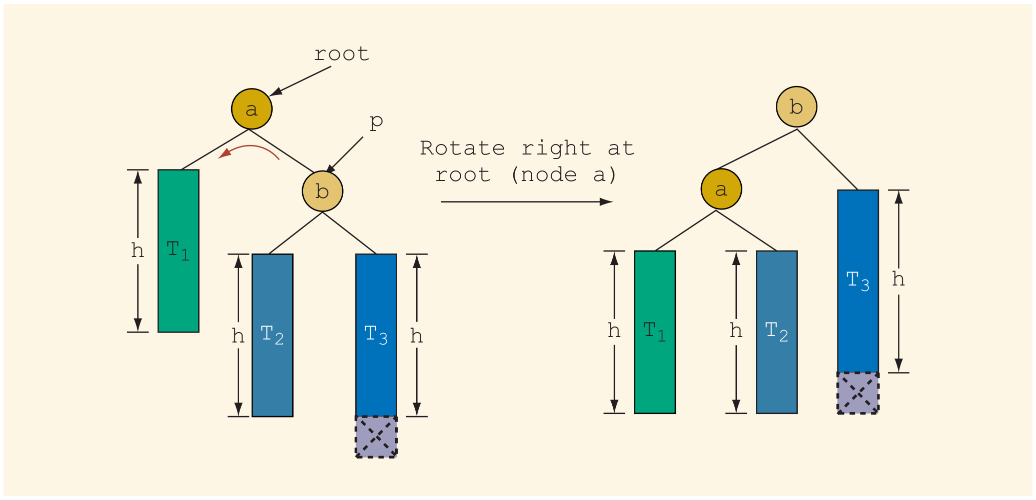

FIGURE AVL-17 Left rotation at *a*

Case 3: Consider Figure AVL-18.

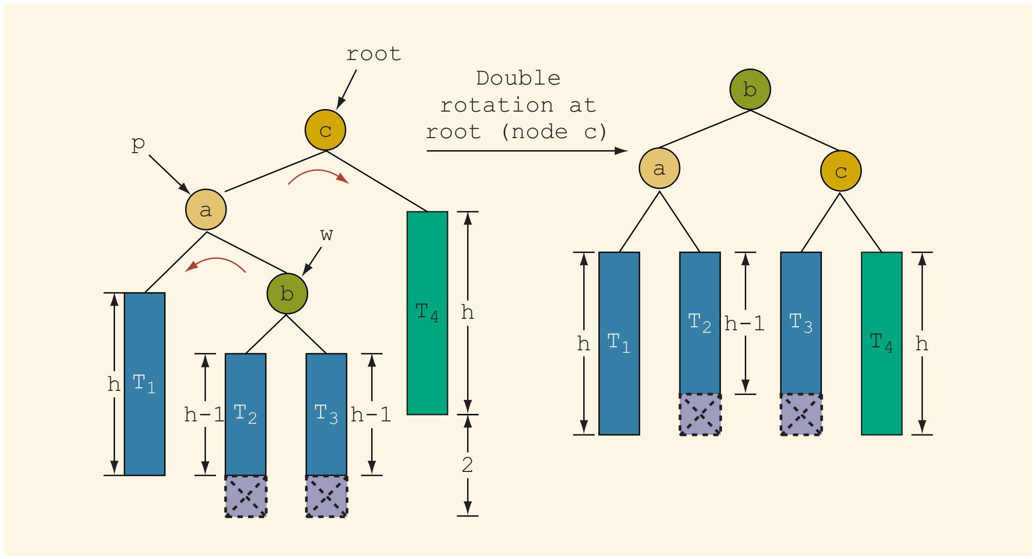

FIGURE AVL-18 Double rotation: first, rotate left at *a*, then rotate right at *c*. New item is inserted in either the subtree T_2 or T_3

In Figure AVL-18, the tree on the left is the tree prior to the reconstruction. The heights of the subtrees are shown in the figure. The dotted rectangle shows that a new item is inserted in the subtree, T_2 or T_3, causing the subtree to grow in height. We note the following (in the tree prior to reconstruction):

- All keys in T_3 are smaller than the key in node *c*.
- All keys in T_3 are larger than the key in node *b*.
- All keys in T_2 are smaller than the key in node *b*.
- All keys in T_2 are larger than the key in node *a*.
- After insertion, the subtrees with root nodes *a* and *b* are still AVL trees.
- The balance criteria is violated at the root node, *c*, of the tree.
- The balance factors of node *c*, $bf(c) = -1$, and node *a*, $bf(a) = 1$, are opposite.

This is an example of double rotation. One rotation is required at node *a* and another rotation is required at node *c*. If the balance factor of the node where the tree is to be reconstructed and the balance factor of the higher subtree are opposite, that node requires a double rotation. First, we rotate the tree at node *a* and then at node *c*. Now the tree at node *a* is right high, so we make a left rotation at *a*. Next, because the tree at node *c* is left high, we make a right rotation at *c*. Figure AVL-18 shows the resulting tree (which is to the right of the tree after insertion). Figure AVL-19, however, shows both rotations in sequence.

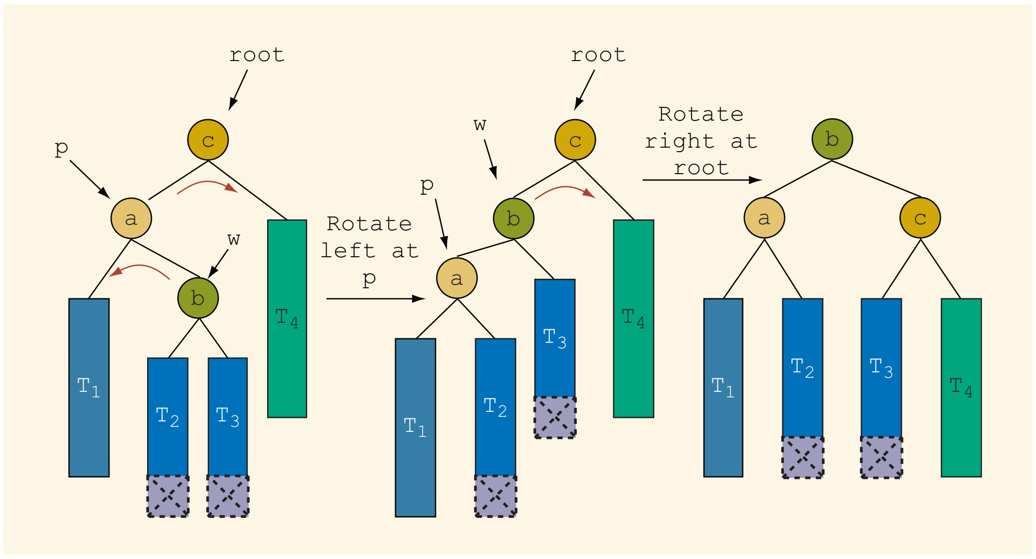

FIGURE AVL-19 Left rotation at *a* followed by a right rotation at *c*

Case 4: This is a mirror image of Case 3. We illustrate this with the help of Figure AVL-20.

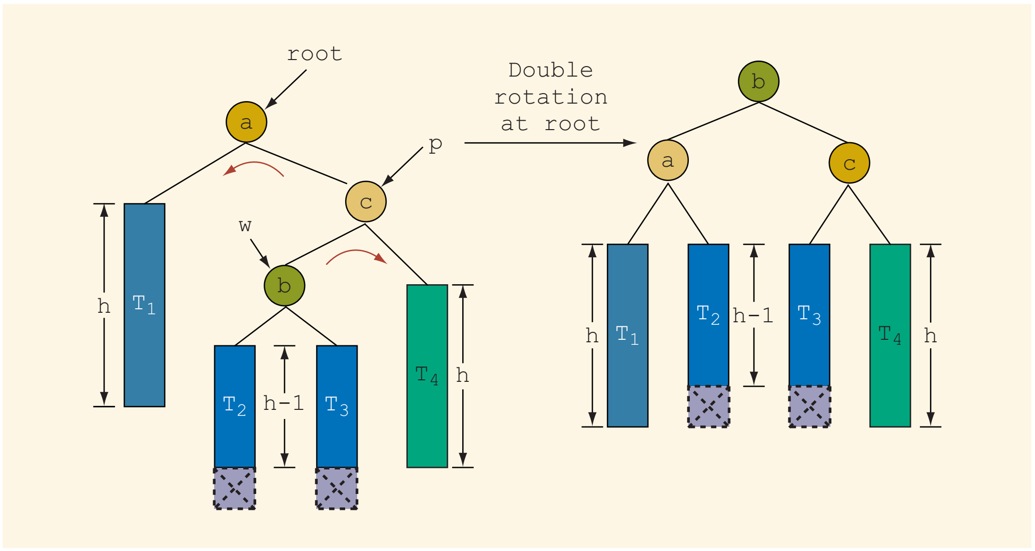

FIGURE AVL-20 Double rotation: first rotate right at *c*, then rotate left at *a*

Using these four cases, we now describe what type of rotation might be required at a node.

Suppose that the tree is to be reconstructed, by rotation, at node x. Then, the subtree with the root node x requires either a single or a double rotation.

1. Suppose that the balance factor of the node x and the balance factor of the root node of the higher subtree of x have the same sign, that is, both positive or both negative.

   a. If these balance factors are positive, make a single *left* rotation at x. (Prior to insertion, the right subtree of x was higher than its left subtree. The new item was inserted in the right subtree of x, causing the height of the right subtree to increase in height, which, in turn, violated the balance criteria at x.)

   b. If these balance factors are negative, make a single *right* rotation at x. (Prior to insertion, the left subtree of x was higher than its right subtree. The new item was inserted in the left subtree of x, causing the height of the left subtree to increase in height, which, in turn, violated the balance criteria at x.)

2. Suppose that the balance factor of the node x and the balance factor of the higher subtree of x are opposite in sign. To be specific, suppose that the balance factor of the node x prior to insertion was -1 and suppose that y is the root node of the left subtree of x. After insertion, the balance factor of the node y is 1. That is, after insertion, the right subtree of node y grew in height.

In this case, we require a *double* rotation at *x*. First, we make a left rotation at *y* (because *y* is right high). Then, we make a right rotation at *x*. The other case, which is a mirror image of this case, is handled similarly.

The following C++ functions implement the left and right rotations of a node. The pointer of the node requiring the rotation is passed as a parameter to the function:

```
template <class elemType>
void rotateToLeft(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;    //pointer to the root of the
                             //right subtree of root
    if (root == nullptr)
        cout << "Error in the tree" << endl;
    else if(root->rLink == nullptr)
        cout << "Error in the tree:"
             << " No right subtree to rotate." << endl;
    else
    {
        p = root->rLink;
        root->rLink = p->lLink; //the left subtree of p
                              //becomes the right subtree of root
        p->lLink = root;
        root = p; //make p the new root node
    }
}//rotateLeft

template <class elemType>
void rotateToRight(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;   //pointer to the root of
                            //the left subtree of root

    if (root == nullptr)
        cout << "Error in the tree" << endl;
    else if(root->lLink == nullptr)
        cout << "Error in the tree:"
             << " No left subtree to rotate." << endl;
    else
    {
        p = root->lLink;
        root->lLink = p->rLink; //the right subtree of p
                             //becomes the left subtree of root
        p->rLink = root;
        root = p; //make p the new root node
    }
}//end rotateRight
```

Now that we know how to implement both rotations, we next write the C++ functions, `balanceFromLeft` and `balanceFromRight`, which are used to reconstruct the tree at a particular node. The pointer of the node where the reconstruction occurs is passed as a parameter to this function. These functions use functions `rotateToLeft` and `rotateToRight` to reconstruct the tree and also adjust the balance factors of the nodes affected by the reconstruction. The function `balanceFromLeft` is called when the subtree is left double high and certain nodes need to be moved to the right subtree. The function `balanceFromRight` has similar conventions:

```
template <class elemType>
void balanceFromLeft(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;
    AVLNode<elemType> *w;

    p = root->lLink;   //p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToRight(root);
        break;
    case 0:
        cout << "Error: Cannot balance from the left." << endl;
        break;
    case 1:
        w = p->rLink;
        switch (w->bfactor)  //adjust the balance factors
        {
        case -1:
            root->bfactor = 1;
            p->bfactor = 0;
            break;
        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;
        case 1:
            root->bfactor = 0;
            p->bfactor = -1;
        }//end switch

        w->bfactor = 0;
        rotateToLeft(p);
        root->lLink = p;
        rotateToRight(root);
    }//end switch;
}//end balanceFromLeft
```

For the sake of completeness, we also give the definition of the function `balanceFromRight`:

```
template <class elemType>
void balanceFromRight(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;
    AVLNode<elemType> *w;

    p = root->rLink;   //p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        w = p->lLink;
        switch (w->bfactor) //adjust the balance factors
        {
        case -1:
            root->bfactor = 0;
            p->bfactor = 1;
            break;
        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;
        case 1:
            root->bfactor = -1;
            p->bfactor = 0;
        }//end switch

        w->bfactor = 0;
        rotateToRight(p);
        root->rLink = p;
        rotateToLeft(root);
        break;
    case 0:
        cout << "Error: Cannot balance from the left." << endl;
        break;
    case 1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToLeft(root);
    }//end switch;
}//end balanceFromRight
```

We now focus our attention on the function `insertIntoAVL`. The function `insertIntoAVL` inserts a new item into an AVL tree. The item to be inserted and the pointer of the root node of the AVL tree are passed as parameters to this function.

The following steps describe the function `insertIntoAVL`:

1. Create a node and assign the item to be inserted to the info field of this node.
2. Search the tree and find the place for the new node in the tree.

3. Insert the new node in the tree.
4. Backtrack the path, which was constructed to find the place for the new node in the tree, to the root node. If necessary, adjust the balance factors of the nodes, or reconstruct the tree at a node on the path.

Because Step 4 requires us to backtrack the path to the root node, and in a binary tree we have links only from the parent to the children, the easiest way to implement the function `insertIntoAVL` is to use recursion. (Recall that recursion automatically takes care of the backtracking.) This is exactly what we do. The function `insertIntoAVL` also uses a `bool` member variable, `isTaller`, to indicate to the parent whether the subtree grew in height or not:

```
template <class elemType>
void insertIntoAVL(AVLNode<elemType>* &root,
                   AVLNode<elemType> *newNode,
                   bool& isTaller)
{
    if (root == nullptr)
    {
        root = newNode;
        isTaller = true;
    }
    else if(root->info == newNode->info)
        cout << "No duplicates are allowed." << endl;
    else if(root->info > newNode->info) //newItem goes in
                                         //the left subtree
    {
        insertIntoAVL(root->lLink, newNode, isTaller);

        if (isTaller)   //after insertion, the subtree grew
                        //in height
            switch (root->bfactor)
            {
            case -1:
                balanceFromLeft(root);
                isTaller = false;
                break;
            case 0:
                root->bfactor = -1;
                isTaller = true;
                break;
            case 1:
                root->bfactor = 0;
                isTaller = false;
            }//end switch
    }//end if
    else
    {
        insertIntoAVL(root->rLink, newNode, isTaller);
        if (isTaller)     //after insertion, the
                          //subtree grew in height
        switch (root->bfactor)
        {
        case -1:
            root->bfactor = 0;
            isTaller = false;
            break;
        case 0:
            root->bfactor = 1;
            isTaller = true;
            break;
        case 1:
            balanceFromRight(root);
            isTaller = false;
        }//end switch
    }//end else
}//insertIntoAVL
```

Next, we illustrate how the function `insertIntoAVL` works and build an AVL tree from scratch. Initially, the tree is empty. Each figure (Figures AVL-21 through AVL-29) shows the item to be inserted, as well as the balance factor of each node. An equal sign (=) on the top of a node indicates that the balance factor of this node is 0, the less-than symbol (<) indicates that the balance factor of this node is −1, and the greater-than symbol (>) indicates that the balance factor of this node is 1.

Initially, the AVL tree is empty. Let us insert 40 into the empty AVL tree (see Figure AVL-21).

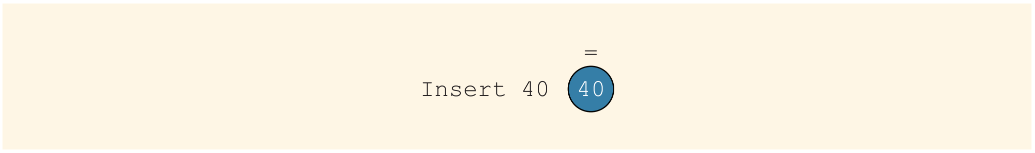

FIGURE AVL-21 AVL tree after inserting 40

Next, we insert 30 into the AVL tree (see Figure AVL-22).

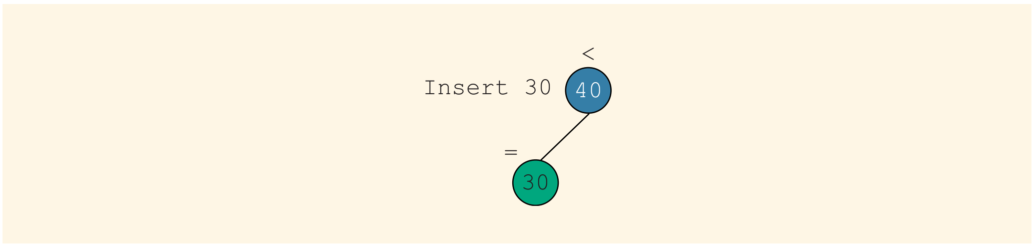

FIGURE AVL-22 AVL tree after inserting 30

Item 30 is inserted into the left subtree of node 40, causing the left subtree of 40 to grow in height. After insertion, the balance factor of node 40 is −1.

Next, we insert 20 into the AVL tree (see Figure AVL-23).

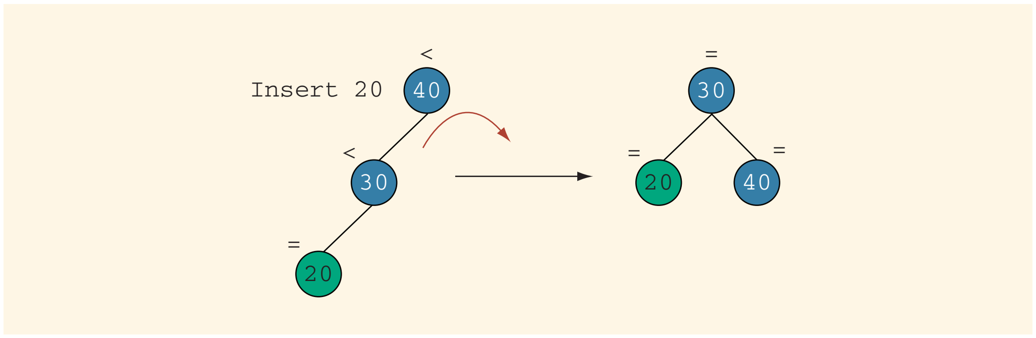

FIGURE AVL-23 AVL tree after inserting 20

The insertion of 20 violates the balance criteria at node 40. The tree is reconstructed at node 40 by making a single right rotation.

Next, we insert 60 into the AVL tree (see Figure AVL-24).

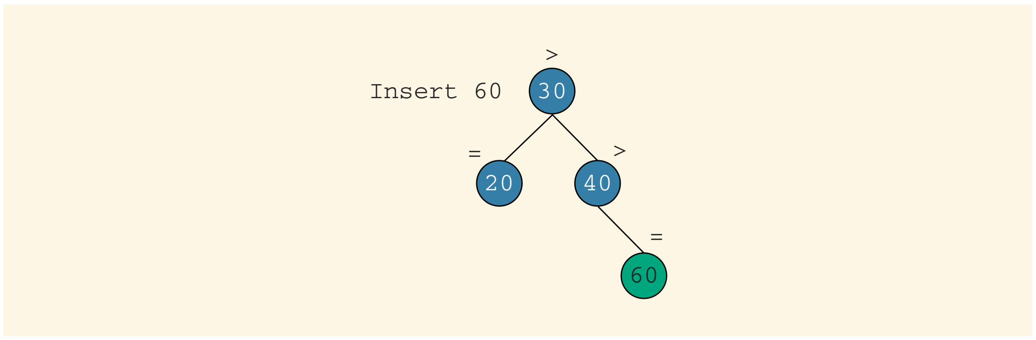

FIGURE AVL-24 AVL tree after inserting 60

The insertion of 60 does not require reconstruction; only the balance factor is adjusted at nodes 40 and 30.

Next, we insert 50 (see Figure AVL-25).

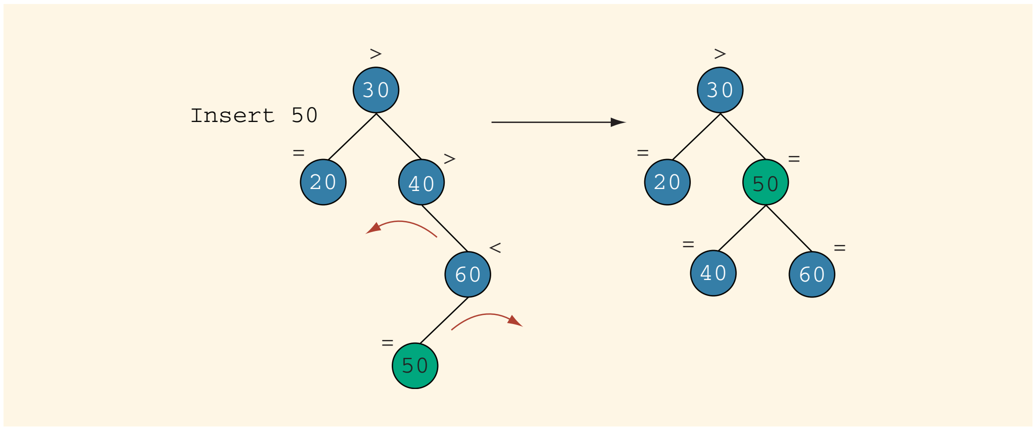

FIGURE AVL-25 AVL tree after inserting 50

The insertion of 50 requires the tree to be reconstructed at 40. Notice that a double rotation is made at node 40.

Next, we insert 80 (see Figure AVL-26).

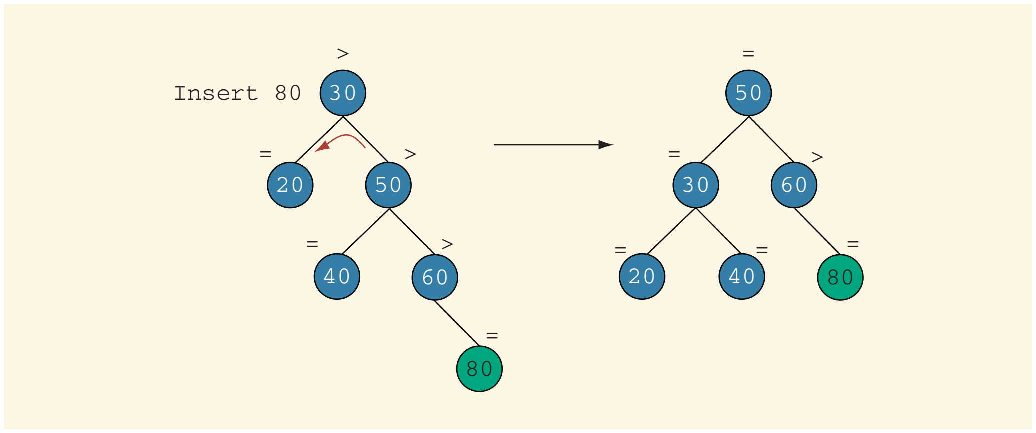

FIGURE AVL-26 AVL tree after inserting 80

The insertion of 80 requires the tree to be reconstructed at node 30. Next, we insert 15 (see Figure AVL-27).

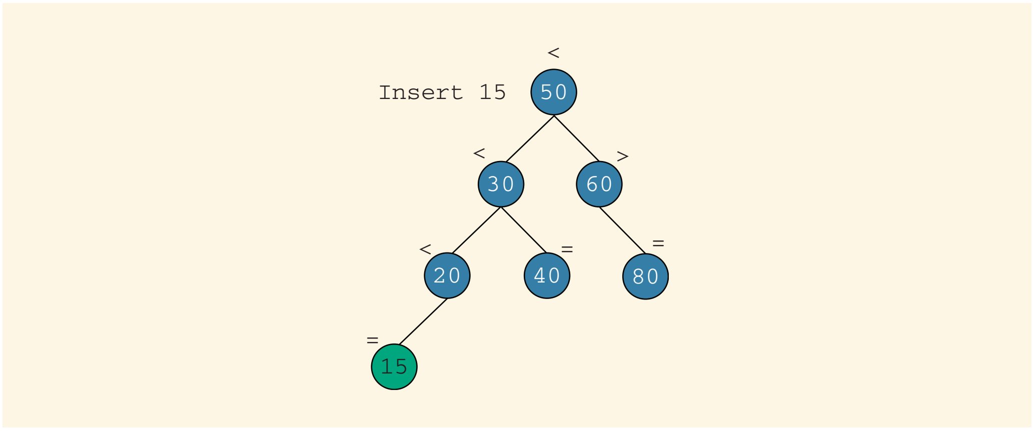

FIGURE AVL-27 AVL tree after inserting 15

The insertion of node 15 does not require any part of the tree to be reconstructed. We need only to adjust the balance factors of nodes 20, 30, and 50.

Next, we insert 28 (see Figure AVL-28).

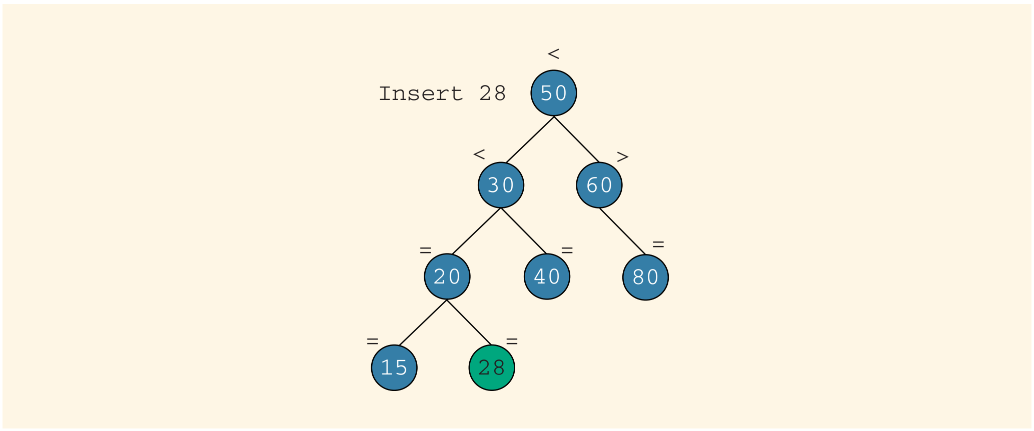

FIGURE AVL-28 AVL tree after inserting 28

The insertion of node 28 also does not require any part of the tree to be reconstructed. We need only to adjust the balance factor of node 20.

Next, we insert 25. The insertion of 25 requires a double rotation at node 30. Figure AVL-29 shows both rotations in sequence.

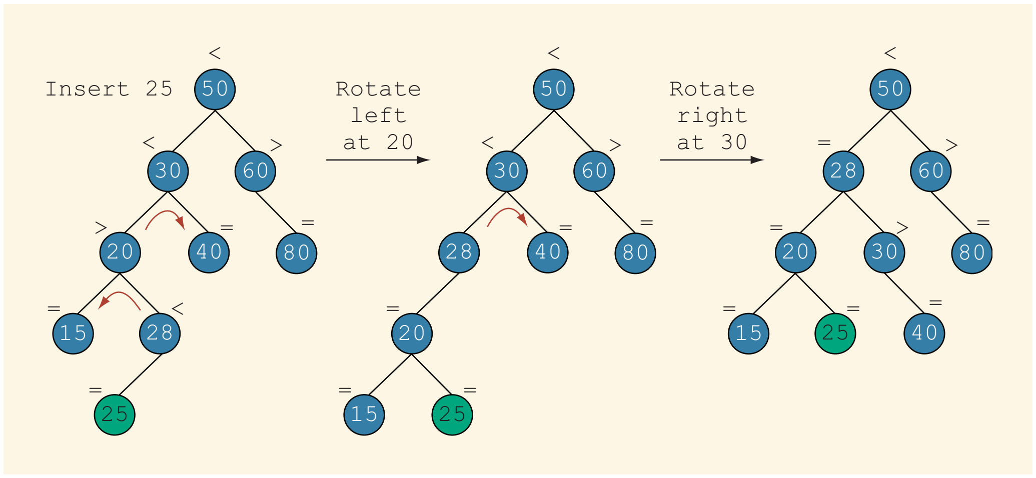

FIGURE AVL-29 AVL tree after inserting 25

In Figure AVL-29, the tree is first rotated left at node 20 and then right at node 30.

The following function creates a node, stores the info in the node, and calls the function `insertIntoAVL` to insert the new node into the AVL tree:

```
template <class elemType>
void insert(const elemType &newItem)
{
    bool isTaller = false;
    AVLNode<elemType> *newNode;

    newNode = new AVLNode<elemType>;
    newNode->info = newItem;
    newNode->bfactor = 0;
    newNode->lLink = nullptr;
    newNode->rLink = nullptr;

    insertIntoAVL(root, newNode, isTaller);
}
```

We leave it as an exercise for you to design the class to implement AVL trees as an ADT. (Notice that because the structure of the node of an AVL tree is different than the structure of the node of a binary tree as discussed in Chapter 19, you cannot use inheritance to derive the class to implement AVL trees from the `class binaryTreeType`.)

Deletion from AVL Trees

To delete an item from an AVL tree, first we find the node containing the item to be deleted. The following four cases arise:

Case 1: The node to be deleted is a leaf.
Case 2: The node to be deleted has no right child, that is, its right subtree is empty.

Case 3: The node to be deleted has no left child, that is, its left subtree is empty.
Case 4: The node to be deleted has a left child and a right child.

Cases 1 through 3 are easier to handle than Case 4. Let us first discuss Case 4.

Suppose that the node to be deleted, say, *x*, has a left and a right child. As in the case of deletion from a binary search tree, we reduce Case 4 to Case 2. That is, we find the immediate predecessor, say, *y*, of *x*. Then, the data of *y* is copied into *x* and now the node to be deleted is *y*. Clearly, *y* has no right child.

To delete the node, we adjust one of the links of the parent node. After deleting the node, the resulting tree may no longer be an AVL tree. As in the case of insertion into an AVL tree, we traverse the path (from the parent node) back to the root node. For each node on this path, sometimes we need to change only the balance factor, while other times the tree at a particular node is reconstructed. The following steps describe what to do at a node on the path back to the root node. (As in the case of insertion, we use the **`bool`** variable `shorter` to indicate whether the height of the subtree is reduced.) Let *p* be a node on the path back to the root node. We look at the current balance factor of *p*.

1. If the current balance factor of *p* is equal high, then the balance factor of *p* is changed according to whether the left subtree of *p* was shortened or the right subtree of *p* was shortened. The variable `shorter` is set to **`false`**.
2. Suppose that the balance factor of *p* is not equal and the taller subtree of *p* is shortened. The balance factor of *p* is changed to equal high, and the variable `shorter` is left as **`true`**.
3. Suppose that the balance factor of *p* is not equal high and the shorter subtree of *p* is shortened. Further suppose that *q* points to the root of the taller subtree of *p*.

    a. If the balance factor of *q* is equal high, a single rotation is required at *p* and `shorter` is set to **`false`**.
    b. If the balance factor of *q* is the same as *p*, a single rotation is required at *p* and `shorter` is set to **`true`**.
    c. Suppose that the balance factors of *p* and *q* are opposite. A double rotation is required at *p* (a single rotation at *q* and then a single rotation at *p*). We adjust the balance factors and set `shorter` to **`true`**.

We leave it as an exercise for you to develop and implement an algorithm to delete a node from an AVL tree.

Analysis: AVL Trees

Consider all the possible AVL trees of height *h*. Let T_h be an AVL tree of height *h* such that T_h has the fewest number of nodes. Let T_{hl} denote the left subtree of T_h and T_{hr} denote the right subtree of T_h. Then:

$$|T_h| = |T_{hl}| + |T_{hr}| + 1$$

where $|T_h|$ denotes the number of nodes in T_h.

Because T_h is an AVL tree of height h such that T_h has the fewest number of nodes, it follows that one of the subtrees of T_h is of height $h - 1$ and the other is of height $h - 2$. To be specific, suppose that T_{hl} is of height $h - 1$ and T_{hr} is of height $h - 2$. From the definition of T_h, it follows that T_{hl} is an AVL tree of height $h - 1$ such that T_{hl} has the fewest number of nodes among all AVL trees of height $h - 1$. Similarly, T_{hr} is an AVL tree of height $h - 2$ that has the fewest number of nodes among all AVL trees of height $h - 2$. Thus, T_{hl} is of the formT_{h-1} and T_{hr} is of the form T_{h-2}. Hence:

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1$$

Clearly:

$$\begin{aligned}|T_0| &= 1\\ |T_1| &= 2\end{aligned}$$

Let $F_{h+2} = |T_h| + 1$. Then:

$$\begin{aligned}F_{h+2} &= F_{h+1} + F_h\\ F_2 &= 2\\ F_3 &= 3.\end{aligned}$$

This is called a **Fibonacci sequence**. The solution to F_h is given by:

$$F_h \approx \frac{\phi^h}{\sqrt{5}}, \quad \text{where} \quad \phi = \frac{1+\sqrt{5}}{2}.$$

Hence:

$$|T_h| \approx \frac{\phi^{h+2}}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2}\right]^{h+2}$$

From this it can be concluded that:

$$h \approx (1.44)\log_2|T_h|$$

This implies that, in the worst case, the height of an AVL tree with n nodes is approximately $1.44\log_2 n = O(\log_2 n)$. Because the height of a perfectly balanced binary tree with n nodes is $\log_2 n$, it follows that, in the worst case, the time to manipulate an AVL tree is no more than 44% of the optimum time. However, in general, AVL trees are not as sparse as in the worst case. It can be shown that the average search time of an AVL tree is about 4% more than the optimum.

QUICK REVIEW

1. A perfectly balanced binary tree is a binary tree such that:
   i. The height of the left and right subtrees of the root are equal.
   ii. The left and right subtrees of the root are perfectly balanced binary trees.
2. An AVL (or height-balanced) tree is a binary search tree such that:
   i. The height of the left and right subtrees of the root differ by at most 1.
   ii. The left and right subtrees of the root are AVL trees.
3. Let x be a node in a binary tree. Then, x_l denotes the height of the left subtree of x and x_r denotes the height of the right subtree of x.
4. Let T be an AVL tree and x be a node in T. Then, $|x_r - x_l| \leq 1$, where $|x_r - x_l|$ denotes the absolute value of $x_r - x_l$.
5. Let x be a node in the AVL tree T.
   a. If $x_l > x_r$, we say that x is left high. In this case, $x_l = x_r + 1$.
   b. If $x_l = x_r$, we say that x is equal high.
   c. If $x_r > x_l$, we say that x is right high. In this case, $x_r = x_l + 1$.
6. The balance factor of x, written $bf(x)$, is defined as $bf(x) = x_r - x_l$.
7. Let x be a node in the AVL tree T. Then:
   a. If x is left high, then $bf(x) = -1$.
   b. If x is equal high, then $bf(x) = 0$.
   c. If x is right high, then $bf(x) = 1$.
8. Let x be a node in a binary tree. We say that node x violates the balance criteria if $|x_r - x_l| > 1$, that is, if the height of the left and right subtrees of x differ by more than 1.
9. Every node x in the AVL tree T, in addition to the data and references of the left and right subtrees, must keep track of its balance factor.
10. In an AVL tree, there are two types of rotations: left rotation and right rotation. Suppose that the rotation occurs at node x. If it is a left rotation, then certain nodes from the right subtree of x move to its left subtree; the root of the right subtree of x becomes the new root of the reconstructed subtree. Similarly, if it is a right rotation at x, certain nodes from the left subtree of x move to its right subtree; the root of the left subtree of x becomes the new root of the reconstructed subtree.

EXERCISES

1. Insert 100 in the AVL tree of Figure AVL-30. The resulting tree must be an AVL tree. What is the balance factor at the root node after the insertion?

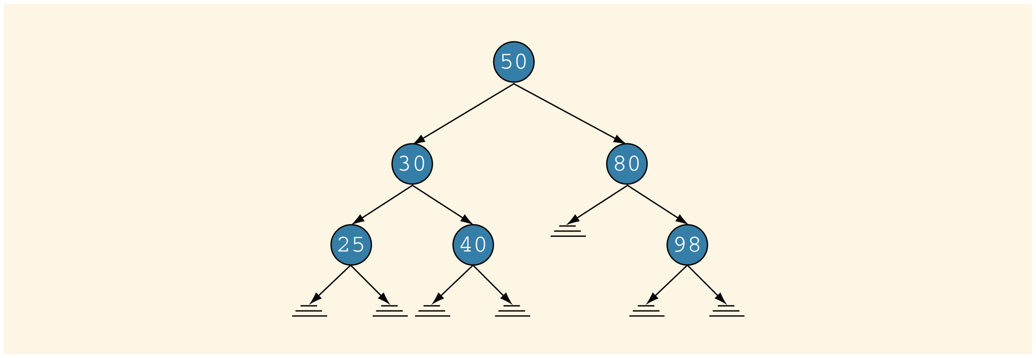

FIGURE AVL-30 AVL tree for Exercise 1

2. Insert 45 in the AVL tree of Figure AVL-31. The resulting tree must be an AVL tree. What is the balance factor at the root node after the insertion?

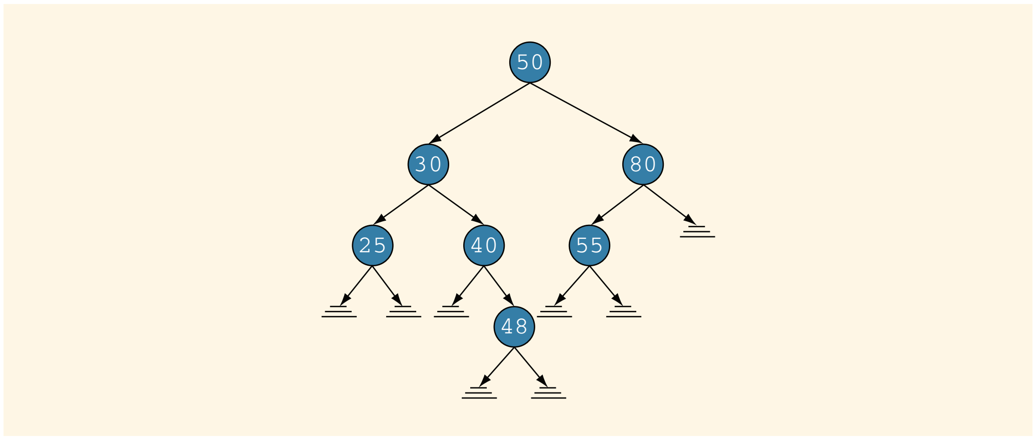

FIGURE AVL-31 AVL tree for Exercise 2

3. Insert 42 in the AVL tree of Figure AVL-32. The resulting tree must be an AVL tree. What is the balance factor at the root node after the insertion?

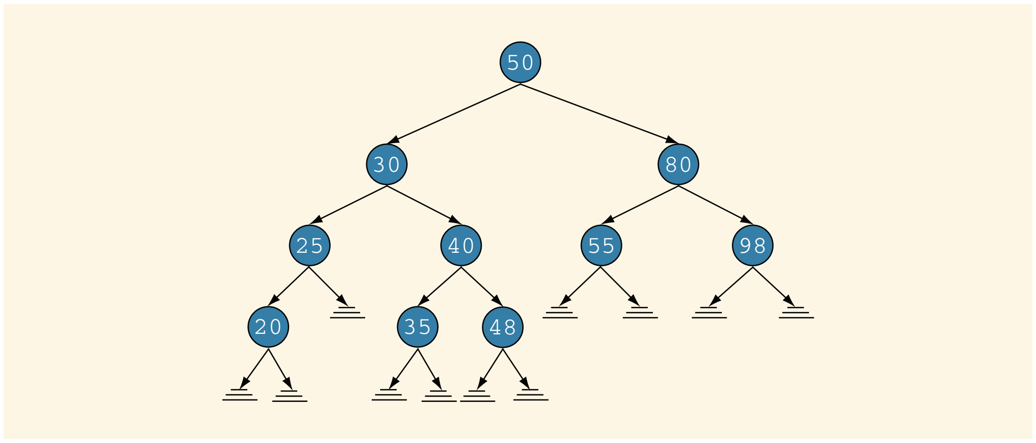

FIGURE AVL-32 AVL tree for Exercise 3

4. The following keys are inserted (in the order given) into an initially empty AVL tree. Show the AVL tree after each insertion.

   `24, 39, 31, 46, 48, 34, 19, 5, 29`

PROGRAMMING EXERCISES

1. a. Write the definition of the class that implements an AVL tree as an ADT. In addition to the functions to implement the operation to insert an item in an AVL tree, your class must, at least, contain the constructors, the function to overload the assignment operator, copy constructor, the function `copyTree`, and traversal algorithms. (You do not need to implement the delete operation.)
   b. Write the definitions of the constructors and the function of the class that you defined in (a).
   c. Write a program to test various operations of an AVL tree.