Tutorial
AVL TREES

Binary search trees are designed for efficient access to data. In some cases, however, a binary search tree is degenerate or "almost degenerate" with most of the n elements descending as a linked list in one of the subtrees of a node. The search efficiency of the tree becomes O(n). A complete binary tree is the other extreme. Nodes are uniformly distributed among left and right subtrees allowing the structure to store n elements in a tree of minimum height. The longest path is $\log_2 n + 1$ and the search efficiency is $O(\log_2 n)$. Intuitively, a degenerate binary tree is very "unbalanced" in the sense that all of the nodes lie in one of the subtrees of the root. On the other hand, a complete binary tree is "balanced" in the sense that nodes are equitably distributed between the two subtrees of a node. Ideally, all binary search trees would be complete trees. This is not possible with random data. The problem is the insert() and erase() algorithms that add or remove an element without any follow-up analysis to determine how the action affects the overall balance of the tree. To address this problem, researchers Adelson, Velskii and Landis defined the concept of *height-balance* for a node and developed new search tree insert and erase algorithms that would reorder the elements to maintain height-balance. The binary search trees with these new algorithms are called AVL trees after their creators. Besides the usual search-ordering of nodes it the tree, an AVL tree is height-balanced. By this we mean that for each node in the tree, the difference in height (depth) of its two subtrees is at most 1.

Figure 1 displays equivalent binary search and AVL trees that store array data. The first example uses array arrA, whose elements are in ascending order while the second example uses array arrB, whose elements are in random order.

```
arrA[5] = {1,2,3,4,5}      arrB[8] = {20,30,80,40,10,60,50,70}
```

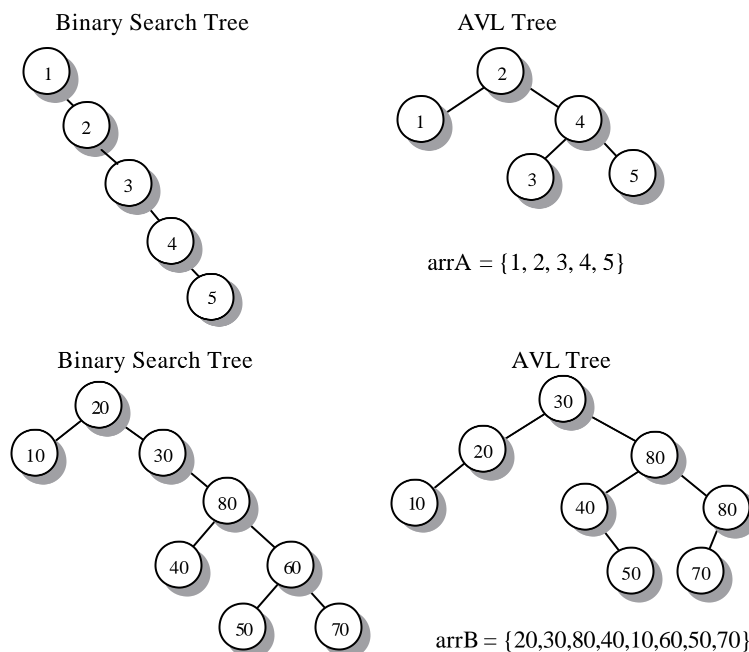


FIGURE 1
Equivalent Binary Search and AVL Trees

The binary search tree for array arrA has a height of 5, whereas the AVL tree has a height of 2. In general, the height of an AVL tree never exceeds $O(\log_2 n)$. This fact makes an AVL tree an efficient search container when rapid access to elements is demanded.

1 AVL Tree Nodes

AVL trees are modeled after binary search trees. The operations are identical although the action of the AVL tree insert() and erase() functions are quite different since they must preserve the balance feature of the tree. To maintain a measure of balance, we define an avlTreeNode object with the integer balanceFactor as an additional field.

avlTreeNode

The value of the field is the difference between the height of the right and left subtrees of the node.

```
balanceFactor = height(right-subtree) - height(left-subtree)
```

If balanceFactor is negative, the node is "heavy on the left" since the height of the left subtree is greater than the height of the right subtree. With balanceFactor positive, the node is "heavy on the right." A balanced node has balanceFactor = 0.

> An AVL tree is a binary search tree in which the balance-factor of each node is in the range -1 to 1.

Figure 2 describes three AVL trees with tags -1, 0, or +1 on each node to indicate its balance-factor (relative height of the left and right subtrees).

-1: height of the left subtree is one greater than the right subtree.
0: height of the left and right subtrees are equal.
+1: height of the right subtree is one greater than the left subtree.

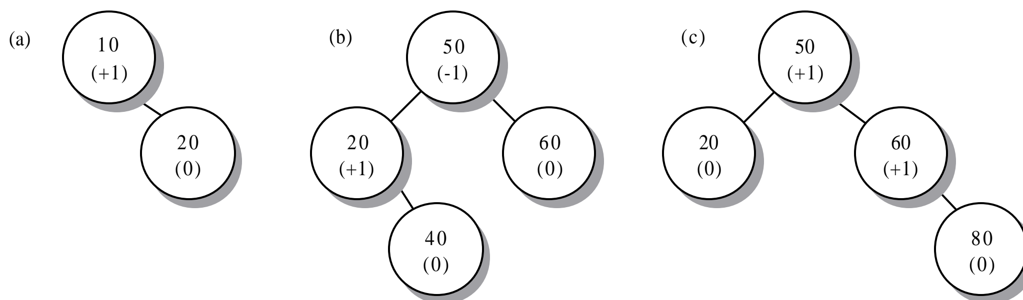


FIGURE 2
AVL trees with balance-factor of each node in parentheses

The avlTreeNode class is similar to the tnode class for binary search trees. Four public data members include the data value, the left and right pointers, and the balance factor. The constructor is used to create a node for an AVL tree and initialize the data members.

DECLARATION: avlTreeNode CLASS

```
// declares a tree node object for a binary tree
template <typename T>
class avlTreeNode
{
   public:
      // data in the node
      T nodeValue;

      // pointers to the left and right children of the node
      avlTreeNode<T> *left;
      avlTreeNode<T> *right;

      int balanceFactor;

      // CONSTRUCTOR
      avlTreeNode (const T& item, avlTreeNode<T> *lptr = NULL,
            avlTreeNode<T> *rptr = NULL, int bal = 0):
            nodeValue(item), left(lptr), right(rptr), balanceFactor(bal)
      {}
};
```

2 The avlTree Class

The avlTree class with the same programmer-interface used by the stree. The class has both constant and non-constant iterators, which can be used to scan the list of elements. A non-constant version is supplied so that a program can use the deference operator * to update the data value of a node. This feature can be used when the AVL trees stores records with a key and other data values and the update modifies only the data values. An attempt to update the key could destroy the structure of the tree.

Like BinSTree iterators, AVL iterators are forward iterators but with an important limitation. Since a tree may need to be rebalanced when an item is added or removed , the insert() and erase() operations invalidate all iterators. To be used again, the iterators must be reset to the start of the list with begin().

The following is a declaration of the programmer-interface for the avlTree class. We implement only the insert() method and do not discuss the erase() methods. The method clear() is added to remove all of the nodes in the tree. In the stree class, clear is executed by calling erase() with the iterator range [begin(), end()). The memory management functions are not included We will expand the declaration to include the private member functions when we discuss the implementation of the class.

DECLARATION: avlTree CLASS (Programmer-Interface

```
template <typename T>
class avlTree
{
      // CONSTRUCTORS, DESTRUCTOR, ASSIGNMENT

      // constructor. initialize root to NULL and size to 0
      avlTree();
      // constructor. insert n elements from range of T* pointers
```

```
    avlTree(T *first, T *last);

    // search for item. if found, return an iterator pointing
    // at it in the tree; otherwise, return end()
    iterator find(const T& item);

    // search for item. if found, return an iterator pointing
    // at it in the tree; otherwise, return end()
    const_iterator find(const T& item) const;

    // indicate whether the tree is empty
    int empty() const;
    // return the number of data items in the tree
    int size() const;

    // give a vertical display of the tree .
    void displayTree(int maxCharacters) const;

    // insert item into the tree
    //pair<iterator, bool> insert(const T& item);

    // insert a new node using the basic List operation and format
    pair<iterator, bool> insert(const T& item);

    // delete all the nodes in the tree
    void clear();

    // constant versions
    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;
};
```

The function displayTree() is similar to displayTree() in the stree class. It includes both the label and the balance-factor for each node in the format <label> : <balanceFactor>

EXAMPLE 1

The example illustrates the use of the avlTree operations.

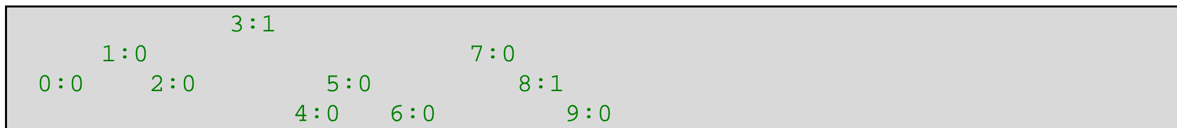
1. Declare an avlTree object avltreeA that stores integers and an object avlTreeB that stores real numbers with initial values from array arrB.

```
// avlTreeA is empty tree of integers
avlTree<int> avlTreeA;

// avlTreeB is a tree of reals with 6 initial values
double arrB[6] = {2.8, 3.9, -2.0, 4.9, 8.6, -12.8};
avlTree<double> avltreeB(arrB, arrB+6);
```

2. A loop initializes avlTreeA with integers 0 to 9. The tree is displayed with displayTree()

```
for (i = 1; i < 10; i++)
```

```
   avlTreeA.insert(i);
avlTreeA.displayTree(2);
```



```
                3:1
    1:0                     7:0
0:0    2:0          5:0          8:1
                4:0    6:0          9:0
```

3. Determine whether 8.6 and 1.6 are elements in avlTreeB.

```
avlTree<double>::iterator dblIter;

if ((dblIter = avlTreeB.find(8.6)) != avlTreeB.end())
   cout << "Element " << *dblIter
        << " is an element in AVL tree" << endl;

if ((dblIter = avlTreeB.find(1.6)) == avlTreeB.end())
   cout << "Element 1.6 is not an element in AVL tree"
        << endl;
```

```
Solution: Element 8.6 is an element in AVL tree
          Element 1.6 is not an element in AVL tree
```

3 Application: Updating Character Counts

An AVL tree is most effective in situations where the data statically resides in the tree and the application primarily searches for items and updates their value. A simple example declares objects of type charCount that contain a character and its frequency (count) as data members. The class has a constructor that with a char argument that initializes the data member and sets the count to 1. The class also provides overloaded versions of the comparison operators < and == along with overloaded version of the << operator. The comparison operators allow charCount to be used as the template type for avlTree objects and their implementation compares only at the char value of the operands. The << operator outputs the character and its count in the form <char>(<count>).

In the example, we count the number of occurrences of the letters 'a' to 'z' in the 25000+ word dictionary "words.dat". For each character in the dictionary, we search for the corresponding charCount object and update the count using the member function incCount().

DECLARATION: charCount CLASS

```
// object contains character and its frequency (count)
class charCount
{
   public:
      // initialize character and count with count = 1
      charCount(char ch);

      // overloaded comparison operators
      friend bool operator < (const charCount& a, const charCount& b);
      friend bool operator == (const charCount& a, const charCount& b);
```

```
      // overloaded output and increment operators
      friend ostream& operator << (ostream& ostr, const charCount& cc);

      // increment the count member of the object
      void incCount();
   private:
      char character;
      int count;
};
```

PROGRAM 1 COUNTING LETTERS

The program illustrates the AVL tree search efficiency by declaring an avlTree<charCount> object and an iterator. A loop inputs words in the dictionary "charct.dat" as strings. For each character in a word, we attempt to insert the corresponding charCount object in the tree. The insert() operation returns a pair. If the object is already in the tree, boolean value of the pair is false and a call to intCount() using the iterator part of the pair and the dereference operator * increments the count for the character. Otherwise, the charCount object enters the tree for the first time with count = 1. An iterator scans the elements in the tree and outputs the letters (in ascending order) and their counts with six items per line.

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <utility>
#include <string>
#include "d_avl.h"
#include "charCount.h"

using namespace std;

int main()
{
   // AVL tree and iterator for charCount objects
   avlTree<charCount> avlCharTree;
   avlTree<charCount>::iterator iter;
   pair<avlTree<charCount>::iterator,bool> p;

   // used for input and output
   string word;
   ifstream fin;
   int outputCount = 0, wdlen;

   // open the file
   fin.open("charct.dat");

   // extract words to end of file
   while (true)
```

```
   {
      fin >> word;
      if (!fin)
         break;

      // use a loop to extract each character from the word
      wdlen = word.length();
      for (i = 0; i < wdlen; i++)
      {
         // try to insert charCount object
         p = avlCharTree.insert(charCount(word[i]));
         // if already present, call incCount()
         if (p.second == false)
            (*(p.first)).incCount();
      }
   }

   // output routine with 6 entries per line
   for (iter = avlCharTree.begin(); iter != avlCharTree.end(); iter++)
   {
      // counter to identify 6 charCount objects per line
      outputCount++;
      cout << *iter;
      // after each multiple of 6 output statements,
      // insert a newline
      if(outputCount % 6 == 0)
         cout << endl;
   }
   cout << endl;

    return 0;
}
```

```
<Input: charct.dat>
peter piper picked a peck of pickled peppers
a peck of pickled peppers peter piper picked
if peter piper picked a peck of pickled peppers
where is the peck that peter piper picked


<Output>
a(  4)   c( 11)   d(  7)   e( 32)   f(  4)   h(  3)
i( 13)   k( 11)   l(  3)   o(  3)   p( 32)   r( 12)
s(  4)   t(  7)   w(  1)
```

4 Implementing the avlTree Class

AVL trees are special types of binary search trees in which each node satisfies the balance condition. The creation and maintenance of an AVL tree is the responsibility of the insert() operation. The algorithm must not only add an element but must reorder the nodes so that the balance can be maintained. The reordering is referred to as rebalancing the tree and the process is new code that is added to the implementation of the avlTree class.

The following is a listing of the private members of the class along with utility functions that are used to rebalance the tree.

DECLARATION: avlTree Class (Private Members)

```
// constants to indicate the balance factor of a node
const int leftheavy = -1;
const int balanced = 0;
const int rightheavy = 1;

template <typename T>
class avlTree
{
   . . .
   private:
      // pointer to tree root
      avlTreeNode<T> *root;
      // number of elements in the tree
      int treeSize;

      // allocate a new tree node and return a pointer to it
      avlTreeNode<T> *getavlTreeNode(const T& item,
                     avlTreeNode<T> *lptr,avlTreeNode<T> *rptr);
      // used by copy constructor and assignment operator
      avlTreeNode<T> *copyTree(avlTreeNode<T> *t);
      // delete the storage occupied by a tree node
      void freeavlTreeNode(avlTreeNode<T> *p);
      // used by destructor, assignment operator and clear()
      void deleteTree(avlTreeNode<T> *t);

      // locate a node item and its parent in tree. used by find()
      avlTreeNode<T> *findNode(const T& item,
                  avlTreeNode<T>* & parent) const;

      // member functions to insert and erase a node
      void singleRotateLeft (avlTreeNode<T>* &p);
      void singleRotateRight (avlTreeNode<T>* &p);
      void doubleRotateLeft (avlTreeNode<T>* &p);
      void doubleRotateRight (avlTreeNode<T>* &p);
      void updateLeftTree (avlTreeNode<T>* &tree,
                      bool &reviseBalanceFactor);
      void updateRightTree (avlTreeNode<T>* &tree,
                        bool &reviseBalanceFactor);

      // class specific versions of the general insert and
};
```

5 The avlTree Insert Function

The implementation of insert() uses the recursive function AVLlnsert() to store the new element. We first give the code for insert() and then focus on the recursive function AVLlnsert(). The insert() function is passed an item of template type T as its argument. A check using find() determines whether the item is already in the tree. If so, insert()returns with a pair(iter, false) where iter is the iterator returned by find(). Otherwise, the function allocates a new node with item as the data value. It also declares an avlTreeNode pointer treeNode that is initially set to the AVL tree root along with the boolean flag reviseBalanceFactor. The local variables treeNode and reviseBalanceFactor are passed as reference arguments along with the new node to AVLInsert(). During the

recursive scan down a search path, treeNode identifies the root of each subtree. Since rebalancing may be needed, the root of the subtree may change and treeNode is updated. The return from insert() is a pair with an iterator referencing the new node and the boolean value true.

insert():

```
template <typename T>
pair<iterator, bool> insert(const T& item)
{
   avlTree<T>::iterator iter;
   // quietly return if item is already in the tree
   if ((iter = find(item)) != end() )
      return pair<iterator,bool>(iter,false);

   // declare AVL tree node pointers.
   avlTreeNode<T> *treeNode = root,*newNode;

   // flag used by AVLInsert to rebalance nodes
   bool reviseBalanceFactor = false;

   // get a new AVL tree node with empty pointer fields
   newNode = getavlTreeNode(item,NULL,NULL);

   // call recursive routine to actually insert the element
   avlInsert(treeNode, newNode, reviseBalanceFactor);

   // assign new values to data members root, size and current
   root = treeNode;
   treeSize++;
   return pair<iterator, bool> (iterator(newNode), true);
}
```

The adding of a new node is carried out by the recursive function avlInsert(). It traverses the left subtree if item is less than the node value and the right subtree if item is greater than the node value. The scan terminates at an empty subtree. The function has an argument called treeNode, which maintains a record of the current node in the scan, the new node to insert in the tree, and a flag called revisebalancefactor. As we scan the left or right subtree of a node, the flag notifies us if any balance factors in the subtree have been changed. If so, we must check that the AVL balance condition is valid at the node. If the insertion of the new node disrupts the equilibrium of the tree and distorts a balance factor, we must reestablish AVL balance.

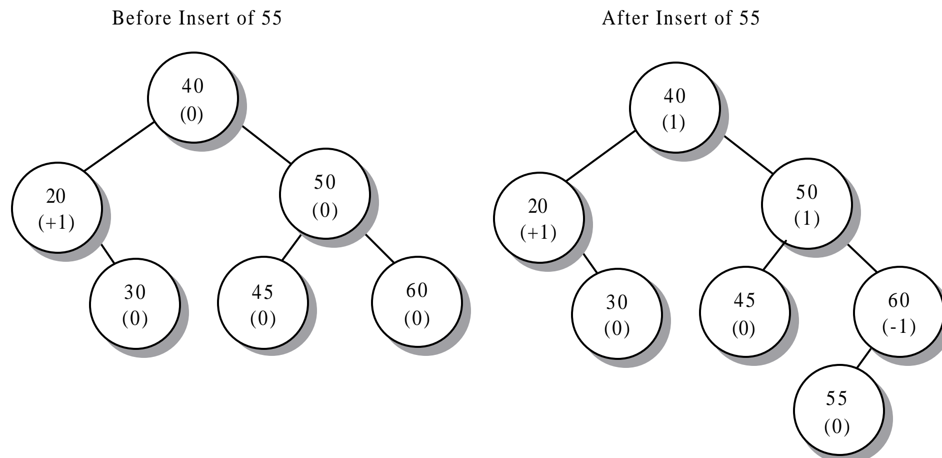
The avlInsert Algorithm

The avlInsert algorithm visits each node in the search path from the root to the new entry. Since the process is recursive, we have access to the nodes in reverse order and can update the balance factor in a parent node after learning the effect of adding the new node in one of its subtrees. At each node in the search path, we determine if an update is necessary. We are confronted with three possible situations. In two cases, the node maintains AVL balance and no rebalancing of subtrees is necessary. Only the balance factor of the node must be updated. The third case unbalances the tree and requires us to perform a single or double rotation of nodes to rebalance the tree.

Case 1: A node on the search path is initially balanced (balanceFactor = 0). After adding a new item in a subtree, the node becomes heavy on the left or the right, depending on which of its subtrees stores the new item. We update balancefactor to - 1 if the item is stored in the left subtree and 1 if stored in the right subtree.

For instance, assume 55 is the new value which must be added to the search path 40 - 50 - 60. Node 40 meets the condition of case 1. The node with value 55 is added in the right subtree of 40 and the balance factor must be updated to +1 which still satisfies the balance criteria. The balance factors are also updated during the recursive scan down the search path (Figure 3).

FIGURE 3
Node on Search Path Is Balanced



Case 2: A node on the path is weighted to the left or right subtree and the new item is stored in the other (lighter) subtree. The node then becomes balanced. For instance, the value 55 is added to the right (lighter) subtree of node 40 which is initially unbalanced ("heavy on the left"). After the insert, node 40 becomes balanced (Figure 4).

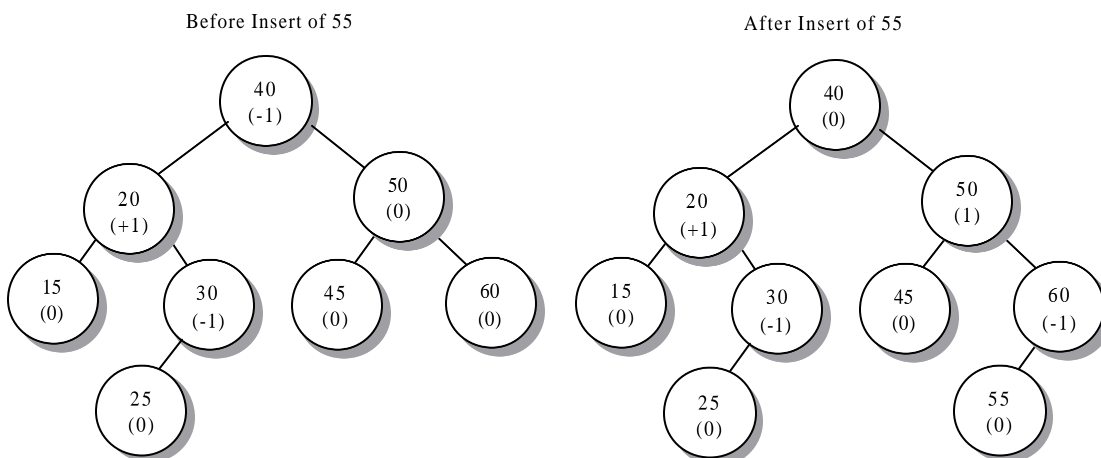


FIGURE 4
Node on Search Path Is Balanced by Insert

Case 3: A node on the path is weighted to a left or right subtree and the new item is positioned in the same (heavier) subtree. The resulting node violates the AVL balance condition since balanceFactor is not in the range -1 to 1. The algorithm directs us to rotate nodes to restore balance.

Figure 5 illustrates case 3. The trees become unbalanced to the left and are rebalanced with rotate right operations. The operations are symmetric when the tree becomes unbalanced to the right.

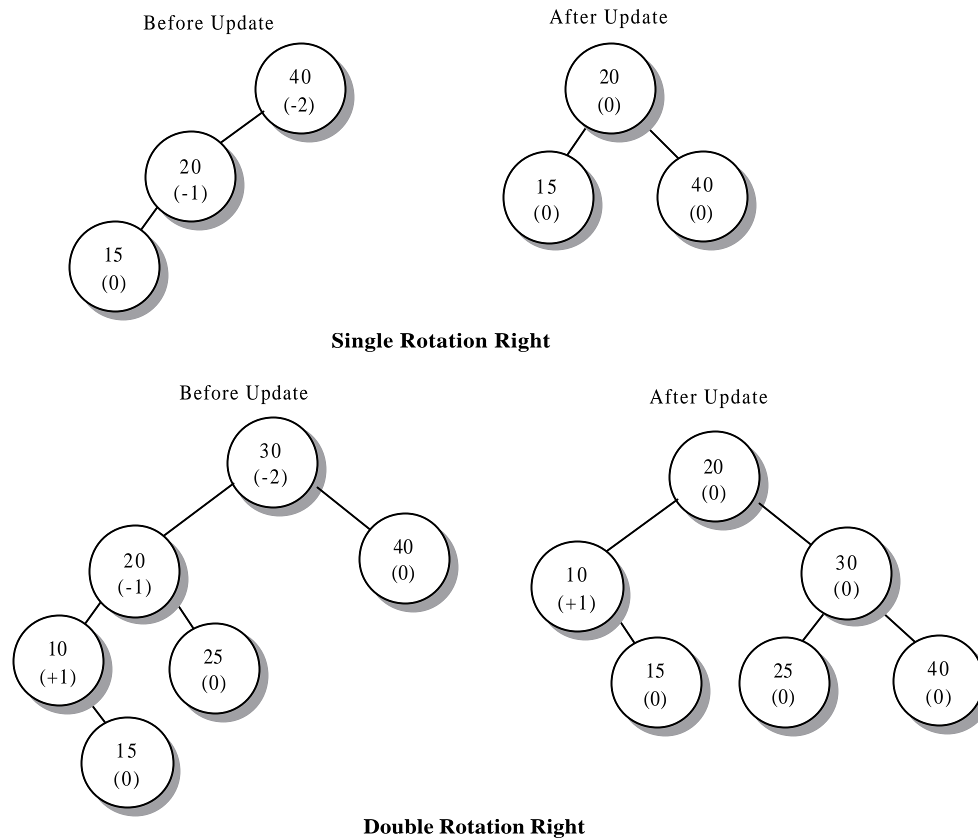


FIGURE 5
Unbalanced Trees

The avlInsert() Function

While traversing down the search path to insert the new item node, the recursive function identifies the three cases that were illustrated in the previous section. When case 3 occurs, the AVL balance condition is violated and we are forced to rebalance nodes. The operations are implemented by the functions updateLeftTree() and updateRightTree(). We first give the code for avlInsert().

avlInsert():

```
template <typename T>
void avlTree<T>::avlInsert(avlTreeNode<T>* & tree,
                 avlTreeNode<T>* newNode, bool& reviseBalanceFactor)
{
   // flag indicates change node's balanceFactor will occur
   bool rebalanceCurrNode;

   // scan reaches an empty tree; time to insert the new node
   if (tree == NULL)
   {
      // update the parent to point at newNode
```

```
   tree = newNode;

   // assign balanceFactor = 0 to new node
   tree->balanceFactor = balanced;

   // broadcast message; balanceFactor value is modified
   reviseBalanceFactor = true;
}

// recursively move left if new data < current data
else if (newNode->nodeValue < tree->nodeValue)
{
   avlInsert(tree->left,newNode,rebalanceCurrNode);
   // check if balanceFactor must be updated.
   if (rebalanceCurrNode)
   {
      // case 3: went left from node that is already heavy
      // on the left. violates AVL condition; rotatate
      if (tree->balanceFactor == leftheavy)
         updateLeftTree(tree,reviseBalanceFactor);

      // case 1: moving left from balanced node. resulting
      // node will be heavy on left
      else if (tree->balanceFactor == balanced)
      {
         tree->balanceFactor = leftheavy;
         reviseBalanceFactor = true;
      }
      // case 2: scanning left from node heavy on the
      // right. node will be balanced
      else
      {
         tree->balanceFactor = balanced;
         reviseBalanceFactor = true;
      }
   }
   else
      // no balancing occurs; do not ask previous nodes
      reviseBalanceFactor = false;
}

// otherwise recursively move right
else
{
   avlInsert(tree->right, newNode, rebalanceCurrNode);
   // check if balanceFactor must be updated.
   if (rebalanceCurrNode)
   {
      // case 2: node becomes balanced
      if (tree->balanceFactor == leftheavy)
      {
         // scanning right subtree. node heavy on left.
         // the node will become balanced
         tree->balanceFactor = balanced;
         reviseBalanceFactor = false;
      }
      // case 1: node is initially balanced
```

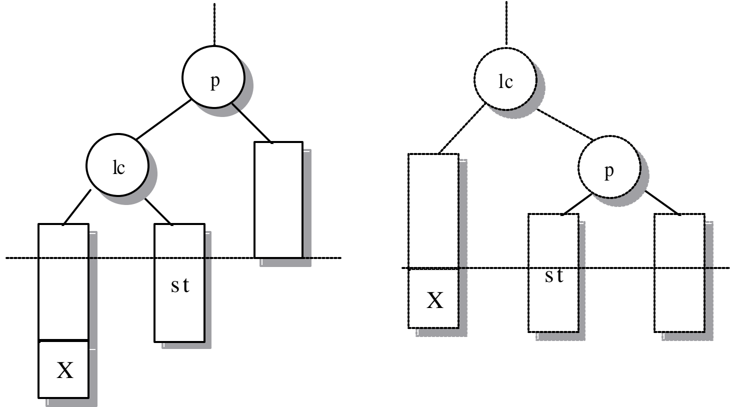
```
            else if (tree->balanceFactor == balanced)
            {
               // node is balanced; will become heavy on right
               tree->balanceFactor = rightheavy;
               reviseBalanceFactor = true;
            }
            else
               // case 3: need to update node
               // scanning right from a node already heavy on
               // the right. this violates the AVL condition
               // and rotations are needed.
               updateRightTree(tree, reviseBalanceFactor);
         }
         else
            reviseBalanceFactor = false;
      }
```

With case 3, avlInsert() uses updateLeftTree() and updateR!ghtTree() to carry out the rebalancing. These functions select the appropriate single or double rotation to balance a node and then set the flag reviseBalanceFactor to false to notify the parent that the subtree is balanced. We give the code for updateLeftTree() before illustrating the details for the rotations.

updateLeftTree():

```
   template <typename T>
   void avlTree<T>::updateLeftTree (avlTreeNode<T>* &p,
                                   bool &reviseBalanceFactor)
   {
      avlTreeNode<T> *lc;

      lc = p->left;              // left subtree is also heavy
      if (lc->balanceFactor == leftheavy)
      {
         singleRotateRight(p);    // need a single rotation
         reviseBalanceFactor = false;
      }
      // is right subtree heavy?
      else if (lc->balanceFactor == rightheavy)
      {
         // make a double rotation
         doubleRotateRight(p);
         // root is now balanced
         reviseBalanceFactor = false;
      }
   }
```

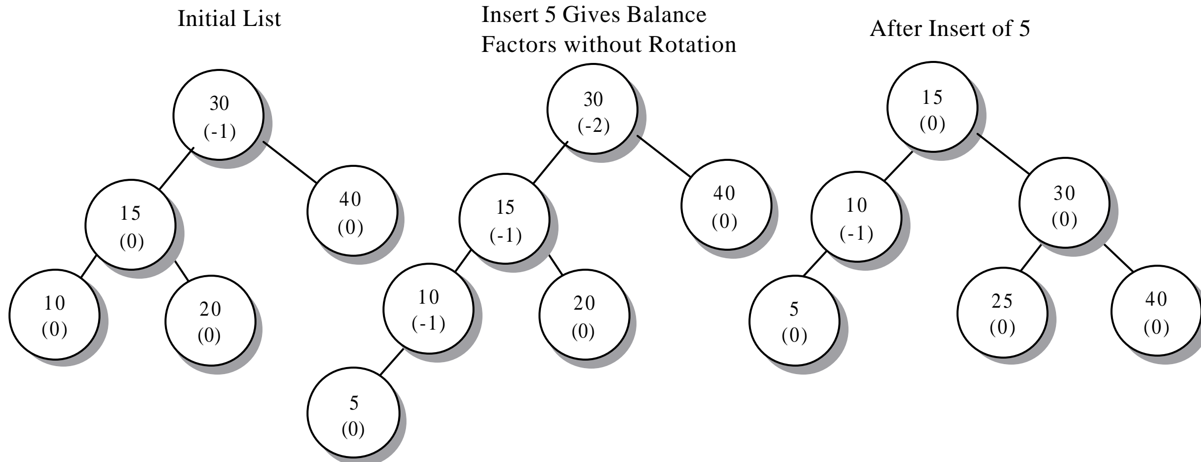
Rotations Rotations are necessary when the parent node P becomes unbalanced. A *single right rotation* occurs when both the parent node (P) and the left child (LC) become heavy on the left after inserting the node at position X. We rotate the nodes so that LC replaces the parent, which becomes a right child. In the process, we take the nodes in the right subtree of LC (ST) and attach them as a left subtree of P. This maintains the ordering since nodes in ST are greater than or equal to LC but less than P. The rotation balances both the parent and left child.



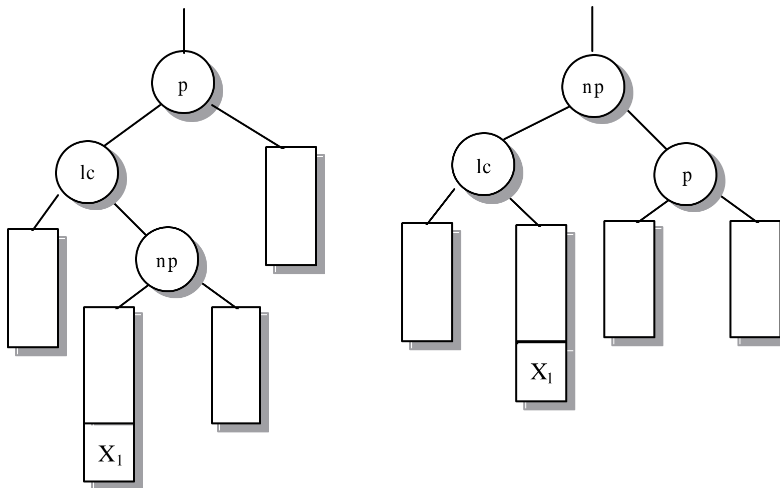
singleRotateRight():

```
// rotate clockwise about node p; make lc the new pivot
template <typename T>
void avlTree<T>::singleRotateRight (avlTreeNode<T>* & p)
{
   // the left subtree of p is heavy
   avlTreeNode<T> *lc;

   // assign the left subtree to lc
   lc = p->left;

   // update the balance factor for parent and left child
   p->balanceFactor = balanced;
   lc->balanceFactor = balanced;

   // any right subtree of lc must continue as right subtree
   // of lc. do this by making it a left subtree of p
   p->left = lc->right;

   // rotate p (larger node) into right subtree of lc
   // make lc the pivot node
   lc->right = p;
   p = lc;
}
```

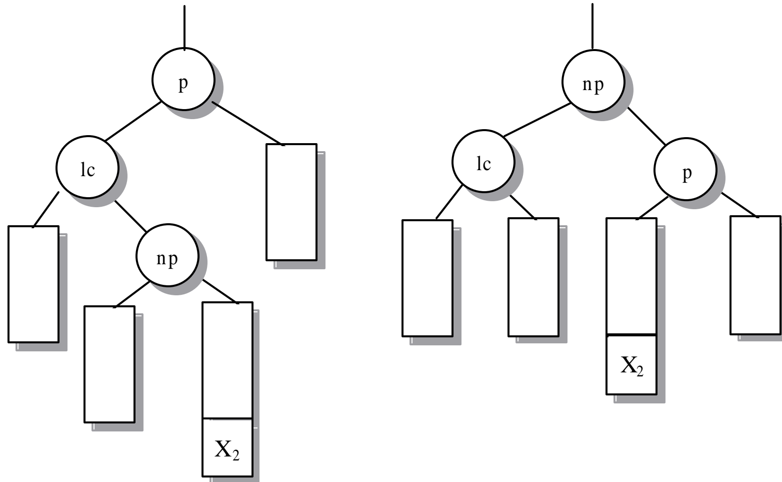
In the following AVL tree, an attempt to insert 5 causes node 30 to violate the AVL condition. At the same time, the left subtree of node 15 (LC) becomes heavy and we call singleRotateRight()to reorder the nodes. In the process, the parent node (node 30) becomes balanced and node 10 becomes heavy on the left.



A *double right rotation* occurs when the parent node (P) becomes heavy on the left and the left child (LC) become heavy on the right. NP is the root of the heavy right subtree of LC. We rotate the nodes so that NP replaces the parent node. In the following diagrams, we describe two cases where the new node is inserted as a child of NP. In each case, NP becomes the parent node and the original parent P rotates to the right subtree of NP.



In the first diagram, we see the shift of node X, after it is inserted in the left subtree of NP. The second diagram illustrates the repositioning of X, after its insertion in the right subtree of NP.



doubleRotateRight():

```
// double rotation right about node p
template <typename T>
void avlTree<T>::doubleRotateRight (avlTreeNode<T>* &p)
{
   // two subtrees that are rotated
   avlTreeNode<T> *lc, *np;

   // in the tree, node(lc) < nodes(np) < node(p)
   lc = p->left;         // lc is left child of parent
   np = lc->right;       // np is right child of lc

   // update balance factors for p, lc, and np
   if (np->balanceFactor == rightheavy)
   {
      p->balanceFactor = balanced;
      lc->balanceFactor = leftheavy;
   }
   else if (np->balanceFactor == balanced)
   {
      p->balanceFactor = balanced;
      lc->balanceFactor = balanced;
   }
   else
   {
      p->balanceFactor = rightheavy;
      lc->balanceFactor = balanced;
   }
   np->balanceFactor = balanced;

   // before np replaces the parent p, take care of subtrees
   // detach old children and attach new children
   lc->right = np->left;
   np->left = lc;
   p->left = np->right;
   np->right = p;
   p = np;
}
```

The following trees illustrate double rotation. An attempt to insert 25 unbalances the root node 50. In this case, node 20 (LC) has a heavy right subtree and a double rotation is required. The new parent node (NP) becomes node 40. The original parent rotates to the right subtree and attaches node 45, which also rotates from the left side of the tree.