Balanced Binary Trees

Binary search trees provide O(log N) search times provided that the nodes are distributed in a reasonably "balanced" manner. Unfortunately, that is not always the case and performing a sequence of deletions and insertions can often exacerbate the problem.

When a BST becomes badly unbalanced, the search behavior can degenerate to that of a sorted linked list, O(N).

There are a number of strategies for dealing with this problem; most involve adding some sort of restructuring to the insert/delete algorithms.

That can be effective only if the restructuring reduces the average depth of a node from the root of the BST, and if the cost of the restructuring is, on average, O(log N).

We will examine one such restructuring algorithm…

AVL Trees

AVL tree*: a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1, and in which the left and right subtrees are themselves AVL trees.

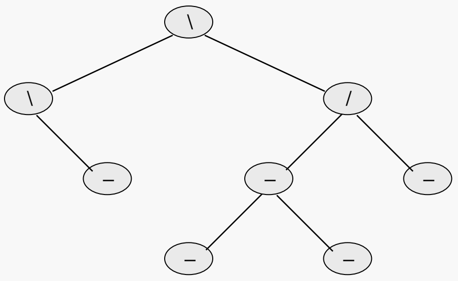
Each AVL tree node has an associated balance factor indicating the relative heights of its subtrees (left-higher, equal, right-higher). Normally, this adds one data element to each tree node and an enumerated type is used.

How effective is this? The height of an AVL tree with N nodes never exceeds 1.44 log N and is typically much closer to log N.

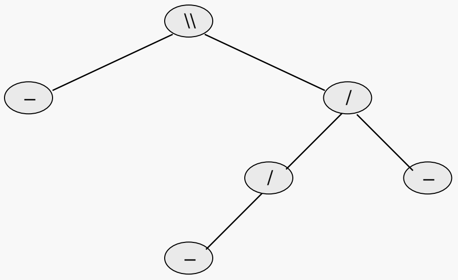
*G. M. Adelson-Velskii and E. M. Landis, 1962.

Examples

This is an AVL tree. . .

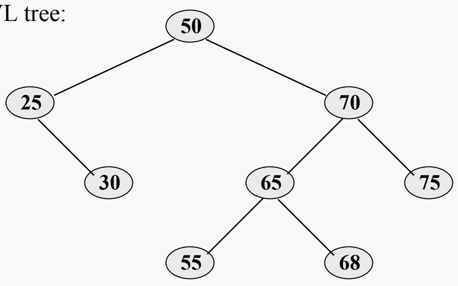


. . .and this is not.

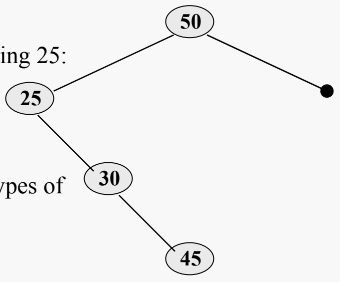


Unbalance from Insertion

Consider inserting the value 45 into the AVL tree:



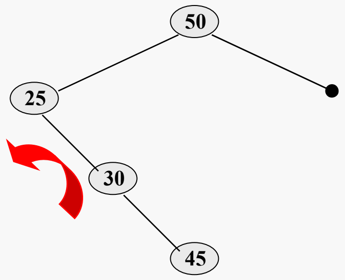
The result would be unbalanced at the node containing 25:



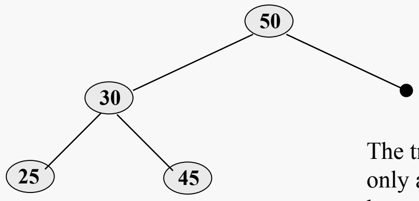
The unbalance is repaired by applying one of two types of "rotation" to the unbalanced subtree…

Rebalancing via Subtree Restructuring

The subtree rooted at 25 is right-higher.



We restructure the subtree, resulting in a balanced subtree:

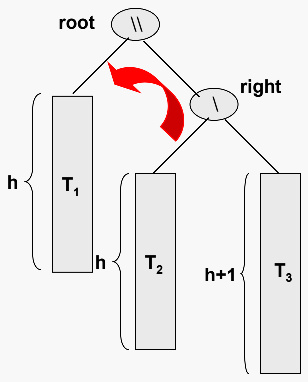


The transformation is relatively simple, requiring only a few operations, and results in a subtree that has equal balance.

AVL Insertion Case: right-higher

There are two unbalance cases to consider, each defined by the state of the subtree that just received a new node. For simplicity, assume for now that the insertion was to the right subtree (of the subtree).

Let `root` be the root of the newly unbalanced subtree, and suppose that its right subtree is now right-higher:



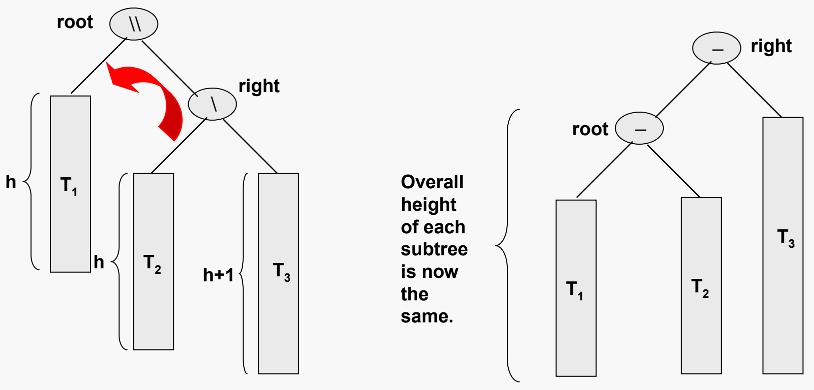
In this case, the subtree rooted at `right` was previously equally balanced (why?) and the subtree rooted at `root` was previously right-higher (why?).

The height labels follow from those observations.

Balance can be restored by "rotating" the values so that `right` becomes the subtree root node and `root` becomes the left child.

AVL Left Rotation

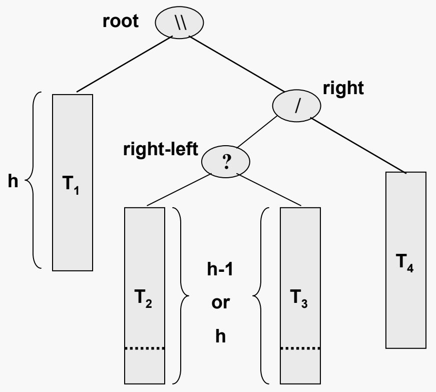
The manipulation just described is known as a “left rotation” and the result is:



That covers the case where the right subtree has become right-higher… the case where the left subtree has become left-higher is analogous and solved by a right rotation.

AVL Insertion Case: left-higher

Now suppose that the right subtree has become left-higher:



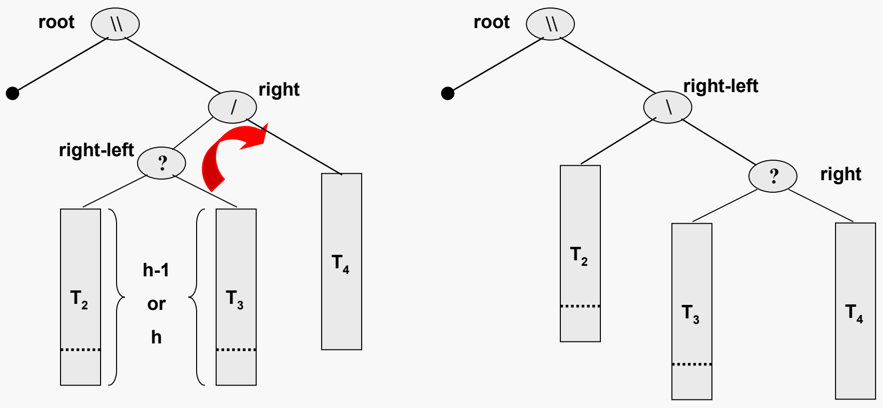
The insertion occurred in the left subtree of the right subtree of `root`.

In this case, the left subtree of the right subtree (rooted at `right-left`) may be either left-higher or right-higher, but not balanced (why?).

Surprisingly (perhaps), this case is more difficult. The unbalance cannot be removed by performing a single left or right rotation.

AVL Double Rotation

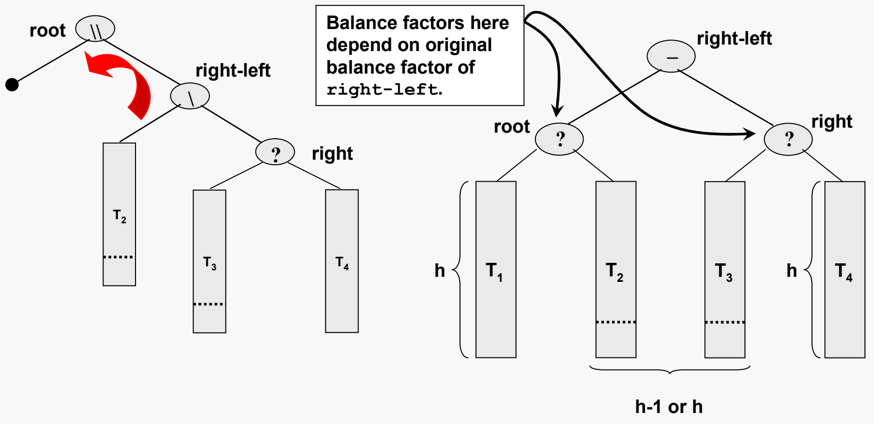
Applying a single right rotation to the subtree rooted at `right` produces…



…a subtree rooted at `right-left` that is now right-higher…

AVL Double Rotation

Now, applying a single left rotation to the subtree rooted at `root` produces…


root
\\
right-left
\
?
right
T2
T3
T4
Balance factors here depend on original balance factor of right-left.
right-left
–
root
?
right
?
h
T1
T2
T3
h
T4
h-1 or h

…a balanced subtree.

The case where the left subtree of `root` is right-higher is handled similarly (by a double rotation).

AVL Node Class Interface

An AVL tree implementation can be derived from the BinNodeT and BST classes seen earlier… however, the AVLNode class must add a data member for the balance factor, and a mutator member function for the balance factor:

```
enum BFactor {LEFTHI, RIGHTHI, EQUALHT, DBLRIGHTHI, DBLLEFTHI, BALUNKNOWN};

template <typename T> class AVLNode {
public:
   T            Element;
   AVLNode<T>* Left;
   AVLNode<T>* Right;
   AVLNode<T>* Parent;
   BFactor     Balance;

   AVLNode();
   AVLNode(const T& Elem, AVLNode<T>* L = NULL, AVLNode<T>* R = NULL,
                          AVLNode<T>* P = NULL);
   BFactor adjustBalance(BFactor Grew);
   bool    isLeaf() const;
   ~AVLNode();
};
```

Because of that, using deriving the AVLNode from the more general BSTNode creates more problems than it solves, and so inheritance is not used in this implementation.

AVL Tree Class Interface

The AVL tree public interface includes:

```
template <typename T> class AVL {
public:
   AVL();
   AVL(const AVL<T>& Source);
   AVL<T>& operator=(const AVL<T>& Source);
   ~AVL();

   T* const Insert(const T& Elem);
   bool     Delete(const T& Elem);
   T* const Find(const T& Elem);
   void Clear();

   void Display(ostream& Out) const;

   // Iterator declarations omitted to save space . . .

   IOiterator iFind(const T& D);   // return access to D

// continues . . .
```

AVL Tree Class Interface

The AVL tree private section includes:

```
// continues . . .

private:
   AVLNode<T>* Root;

   // private helper functions omitted . . .

   string BFtoString(BFactor BF) const;    // used in Display() fn

   void RightBalance(AVLNode<T>* sRoot);  // re-balance managers used by
   void LeftBalance(AVLNode<T>* sRoot);   //   used by insert/delete fns
   void RotateLeft(AVLNode<T>* sRoot);    // rotation functions used by
   void RotateRight(AVLNode<T>* sRoot);   //   used by managers
};
```

AVL Left Rotation Implementation

```
template <typename T>
void AVL<T>::RotateLeft(AVLNode<T>* sRoot) {

   if ( (sRoot == NULL) || (sRoot->Right == NULL) )
      return;

   AVLNode<T>* Temp = new AVLNode<T>(sRoot->Element);

   Temp->Left  = sRoot->Left;
   sRoot->Left = Temp;
   Temp->Right = sRoot->Right->Left;
   AVLNode<T>* toDelete = sRoot->Right;
   sRoot->Element = toDelete->Element;
   sRoot->Right   = toDelete->Right;
   sRoot->Balance = toDelete->Balance;

   delete toDelete;
}
```

AVL Right Balance Implementation

The implementation uses two high-level functions to manage the rebalancing, one for when the imbalance is to the right and one for when it is to the left.

`RightBalance()` would be called when a condition of double-right-high has been detected at a node following an insertion:

```
template <typename T>
void AVL<T>::RightBalance(AVLNode<T>* sRoot) {

   if ( (sRoot == NULL) || (sRoot->Right == NULL) ) return;

   AVLNode<T>* rightSubTreeRoot = sRoot->Right;

   switch ( sRoot->Right->Balance ) {
   case RIGHTHI:  sRoot->Balance = EQUALHT;
                  rightSubTreeRoot->Balance = EQUALHT;
                  RotateLeft(sRoot);
                  break;

// continues . . .
```

AVL Right Balance Implementation

```
// continued . . .
   case LEFTHI:   AVLNode<T>* leftSubtreeRightSubTree =
                                    rightSubTreeRoot->Left;

                  switch ( leftSubtreeRightSubTree->Balance ) {
                       // code to reset balance factors . . .
                  };

                  leftSubtreeRightSubTree->Balance = EQUALHT;
                  RotateRight(rightSubTreeRoot);
                  RotateLeft(sRoot);
                  break;
   }
}
```

AVL Insert Implementation

As usual, insertion involves a relatively simple public "stub" and a recursive helper.

```
template <typename T>
T* const AVL<T>::Insert(const T& Elem) {

   bool Taller = false;

   return InsertHelper(Elem, Root, NULL, Taller);
}
```

```
template <typename T>
T* const AVL<T>::InsertHelper(const T& Elem, AVLNode<T>*& sRoot,
                              AVLNode<T>* Par, bool& Taller) {

   if ( sRoot == NULL ) {     // found the right location
      sRoot = new AVLNode<T>(Elem);
      sRoot->Parent = Par;
      Taller = true;
      return &(sRoot->Element);
   }
// continues . . .
```

This leaves compensating for the effect of the insertion to be handled when the recursion backs out.

AVL Insert Helper: Descent and Rebalance

```
// continued . . .

   if ( Elem < (sRoot->Element) ) {
      T* insertResult = InsertHelper(Elem, sRoot->Left, sRoot, Taller);
      if ( Taller ) { // left subtree just got taller
         BFactor sRootBalance = sRoot->adjustBalance(LEFTHI);
         if ( sRootBalance == EQUALHT ) {
            Taller = false;
         }
         else if ( sRootBalance == LEFTHI ) {
            Taller = true;
         }
         else if ( sRootBalance == DBLLEFTHI ) {
            Taller = false;
            LeftBalance( sRoot );
         }
      }
      return insertResult;
   }
   else {
// . . . continues to handle descent down right subtree . . .
```

Unbalance from Deletion

Deleting a node from an AVL tree can also create an imbalance that must be corrected.

The effects of deletion are potentially more complex than those of insertion.

The basic idea remains the same: delete the node, track changes in balance factors as the recursion backs out, and apply rotations as needed to restore AVL balance at each node along the path followed down during the deletion.

However, rebalancing after a deletion may require applying single or double rotations at more than one point along that path.

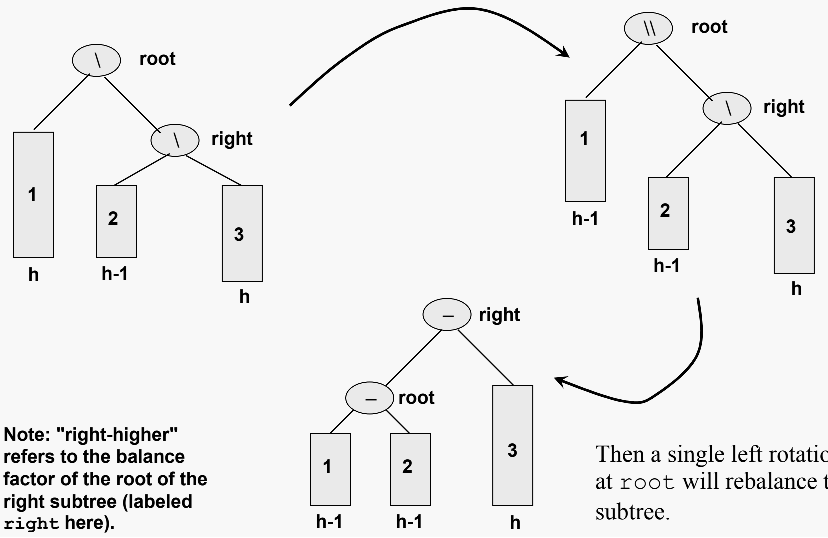
As usual, there are cases…

Here, we will make the following assumptions:

- the lowest imbalance occurs at the node `root` (a subtree root)
- the deletion occurred in the left subtree of `root`

AVL Deletion Case: right-higher

Suppose we have the subtree on the left prior to deletion and that on the right after deletion:

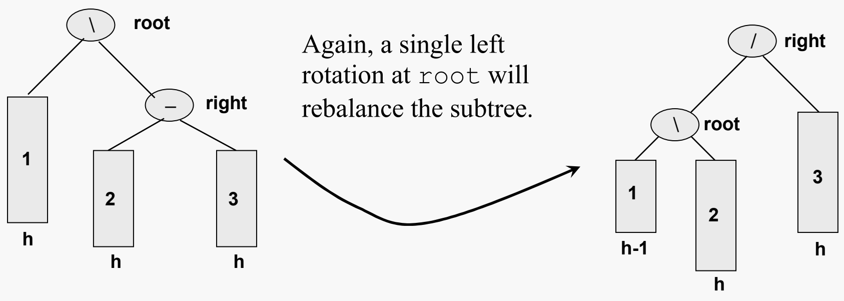


Note: "right-higher" refers to the balance factor of the root of the right subtree (labeled `right` here).

Then a single left rotation at `root` will rebalance the subtree.

AVL Deletion Case: equal-height

Suppose we the right subtree root has balance factor equal-height:

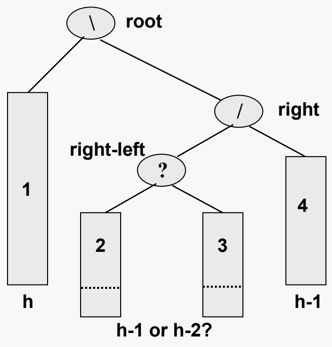


Again, a single left rotation at `root` will rebalance the subtree.

The difference is the resulting balance factor at the old subtree root node, `root`, which depends upon the original balance factor of the node `right`.

AVL Deletion Case: left-higher

If the right subtree root was left-higher, we have the following situation:



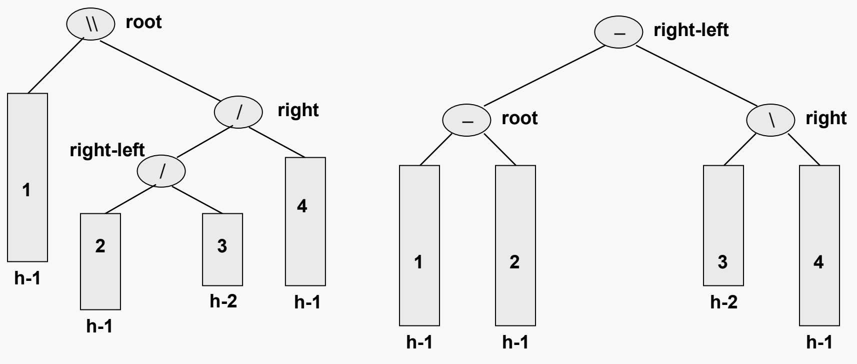
Deleting a node from the left subtree of `root` now will cause `root` to become double right higher.

As you should expect, the resulting imbalance can be cured by first applying a right rotation at the node `right`, and then applying a left rotation at the node `root`.

However, we must be careful because the balance factors will depend upon the original balance factor at the node labeled `right-left`…

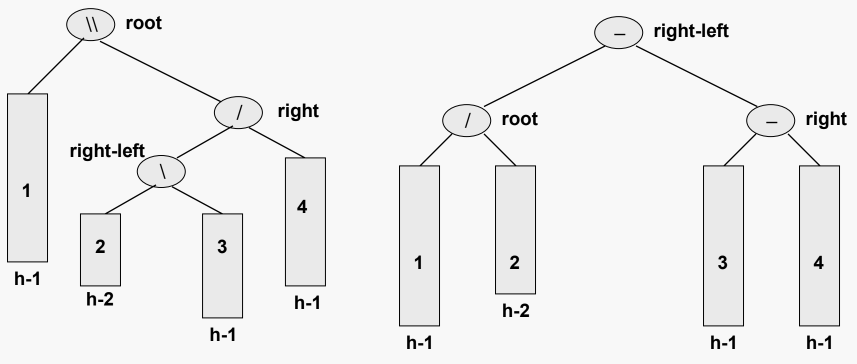
AVL Deletion Case: left-higher, left-higher

If the right-left subtree root was also left-higher, we obtain:



AVL Deletion Case: left-higher, right-higher

If the right-left subtree root was right-higher, we obtain:



And, finally, if the right-left subtree root was equal-height, we'd obtain a tree where all three of the labeled nodes have equal-height.

AVL Deletion Cases: Summary

We have considered a number of distinct deletion cases, assuming that the deletion occurred in the left subtree of the imbalanced node.

There are an equal number of entirely similar, symmetric cases for the assumption the deletion was in the right subtree of the imbalanced node.

Drawing diagrams helps…

This discussion also has some logical implications for how insertion is handled in an AVL tree. The determination of the balance factors in the tree, following the rotations, involves similar logic in both cases.

Implementing AVL Deletion

Turning the previous discussion into an implementation involves a considerable amount of work.

At the top level, the public delete function must take into account whether the deletion shortened either subtree of the tree root node. If that was the case, then it may be necessary to perform a rotation at the tree root itself.

Thus, the helper function(s) must be recursive and must indicate whether a subtree was shortened; this may be accomplished by use of a `bool` parameter, as was done for the insertion functions.

Deletion of the tree root node itself is a special case, and should take into account whether the tree root node had more than one subtree. If not, the root pointer should simply be retargeted to the appropriate subtree and no imbalance occurs.

If the tree root node has two subtrees, its value should be swapped with the minimum value in the right subtree and then the corresponding node should be deleted from the right subtree… which may lead to an imbalance that must be corrected.

Testing

In testing the AVL implementation, I found a few useful ideas:

- Test with small cases first, and vary them to cover all the different scenarios you can think of.
- Work out the solutions manually as well. If you don't understand how to solve the problem by hand, you'll NEVER program a correct solution.
- Test insertion first, thoroughly.
- Then build an initial, correct AVL tree and test deletion, thoroughly.
- Then test alternating combinations of insertions and deletions.
- Modify your tree print function to show the balance factors:

```
 c\
      d-
e\
         h-
      j-
         k-
   o/
      s-
```