Binary Search Trees

The following C++ code implements a binary search tree type, along with standard operations for that type. Random integers are inserted, and the operations are tested.

```
int maxint(int x, int y)
 {
  if(x < y) return y;
  else return x;
 }

struct treenode;

typedef treenode*tree;

struct treenode
 {
  int item;
  tree left;
  tree right;
 };

tree root = NULL;

void insert(tree&t,int newitem)
 {
  if(t == NULL)
   {
    t = new treenode();
    t->item = newitem;
   }
  else if(newitem < t->item) insert(t->left,newitem);
  else insert(t->right,newitem);
 }

void inorder(tree t)
 {
  if(t)
   {
    inorder(t->left);
    cout << t->item << endl;
    inorder(t->right);
   }
 }
```

```
void preorder(tree t)
 {
  if(t)
   {
    cout << t->item << endl;
    preorder(t->left);
    preorder(t->right);
   }
 }

int hite(tree t) // height
  // empty tree has hite -1
 {
  if(t)
   return 1+maxint(hite(t->left),hite(t->right));
  else return -1;
 }

int numbr(tree t)
 {
  if(t)
   return 1 + numbr(t->left) + numbr(t->right);
  else return 0;
 }

void insertrandom(int n)
 {
  // inserts n random 2 digit numbers into the binary search tree
  srand(0);
  for(int i = 1; i < n; i++)
   insert(root,10+rand()%90);
 }

int main()
 {
  insertrandom(20);
  cout << "height = " << hite(root) << endl;
  cout << "number of nodes  = " << numbr(root) << endl;
  inorder(root);
  cout << "---------------" << endl;
  preorder(root);
  return 1;
 }
```

The *worst case* height of a binary search tree is $O(n)$. A binary search trees of size n is considered *balanced* if its height is $O(\log n)$. In an attempt to ensure that its height is $O(\log n)$, at least on the average, a *balancing scheme* may be used. Two forms of self-balancing binary search trees are *treaps* and *AVL trees*.

Treaps

A *treap* is a binary tree where each node has a *data item* and a *priority* The item contains the datum of interest, while the priority is randomly chosen, typically an integer. For maximum efficiency, priorities should be distinct.[1] A treap has two orders, alphabetic order, as is normal for a binary search tree, and max-heap order meaning that the priority of each node must be at least as great as the priority of any of its descendants.

Insertion or deletion of a node into a treap is done the same way as for a binary search tree. A priority is selected at random when in item is inserted. The new node could have higher priority that its parent; heap order is restored by recursive bottom-up rebalancing, consisting of a combination of *left rotations* and *right rotations*. Each rotation preserves the alphabetic ordering.

Rotations. We define two rotations, which we show in the figure below. Suppose W and Y are nodes of a binary tree T.

- If W is the left child of Y, a *right rotation* modifies T as follows:
  - The left subtree of W is unchanged.
  - The right subtree of Y is unchanged.
  - Y becomes the right child of W.
  - The previous right child of W becomes the left child of Y
- If Y is the right child of W, a *left rotation* modifies T as follows:
  - The left subtree of W is unchanged.
  - The right subtree of Y is unchanged.
  - W becomes the left child of Y.
  - The previous left child of Y becomes the right child of W

[1] If there are only a few collisions, there is no significant loss of efficiency.

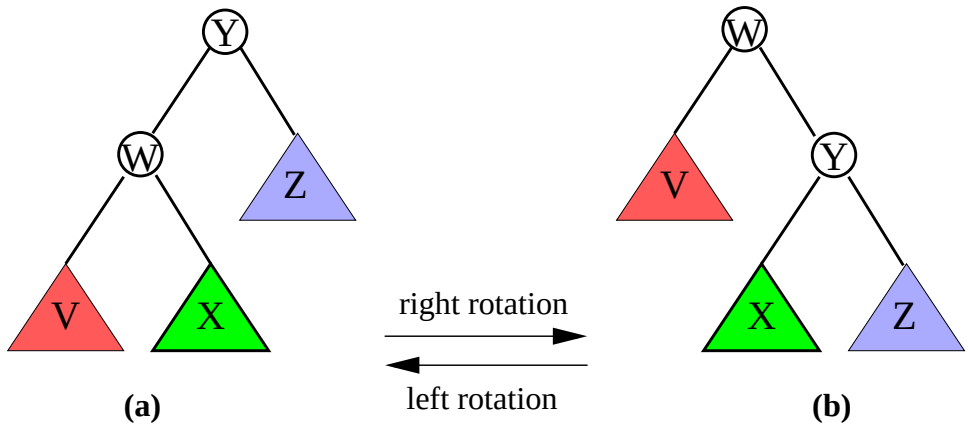

Figure 1 Right and Left Rotations.

Figure 2 shows the steps of inserting nodes A, Y, H, F, and D, in that order, into a treap which initially holds only B.

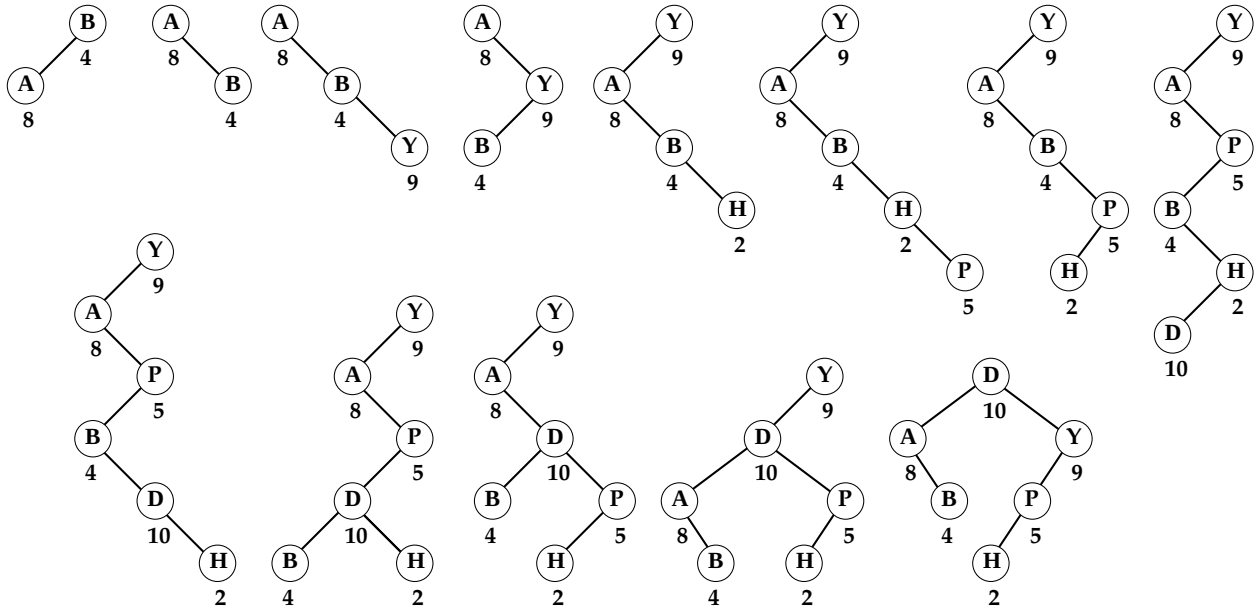

Figure 2: Sequence of Insertions into a Treap.

The steps shown in Figure 2 are as follows. Each time an item is inserted, a priority is chosen at random, and does not later change. Here are the steps illustrated in Figure 2.

1. Insert A with priority 8.
2. Right rotation at B. Heap order is restored.
3. Insert Y with priority 9.
4. Left rotation at B.
5. Left rotation at A. Heap order is restored.

6. Insert H with priority 2. Heap order is preserved.
7. Insert F with priority 5.
8. Left rotation at H. Heap order is restored.
9. Insert D with priority 10.
10. Right rotation at H.
11. Left rotation at B.
12. Right rotation at F.
13. Left rotation at A.
14. Right rotation at Y. Heap order is restored.

The height of a treap of n nodes is excpected to be $O(\log n)$, but in the worst case, the height is $O(n)$.

AVL Trees

A binary tree has the AVL property if the heights of any pair of sibling nodes differ by at most 1. An AVL tree is a self-balancing binary search tree, where the AVL property is maintained during both insertions and deletions by *rebalancing*.

Balance Factors. We define the *balance factor* at any node X of a binary tree to be the diffence between the heights of the left and right subtrees of X, where an empty tree is defined to have height -1. A binary tree has the AVL property if every node has balance factor -1, 0, or 1. If insertion or deletion causes an AVL tree to become unbalanced, one or more *rotations* restore the AVL property.

Four Rotations. To rebalance an AVL tree after an insertion or deletion, there are four kinds of rotations that might be needed:

1. Left rotation, as defined for treaps,
2. Right rotation, as defined for treaps,
3. Double left rotation, as shown in figure 3. A double left rotation at a node X consists of first a right rotation at the left child of X, followed by a left rotation at X.
4. Double right rotation, the mirror image of a double left rotation.

Suppose we insert a node into an AVL tree, and resulting tree is unbalanced, *i.e.,* no longer has the AVL property. Let X be the lowest node at which the tree is unbalanced: without loss of generality, the balance factor at X is 2. Let Y be the right child of X. The balance factor at Y

must be 1 or -1. If it is -1, a left rotation at X restores the AVL property. If it is 1, a double left rotation at X restores the AVL property. Figure 3 illustrates a double left rotation at the node U.

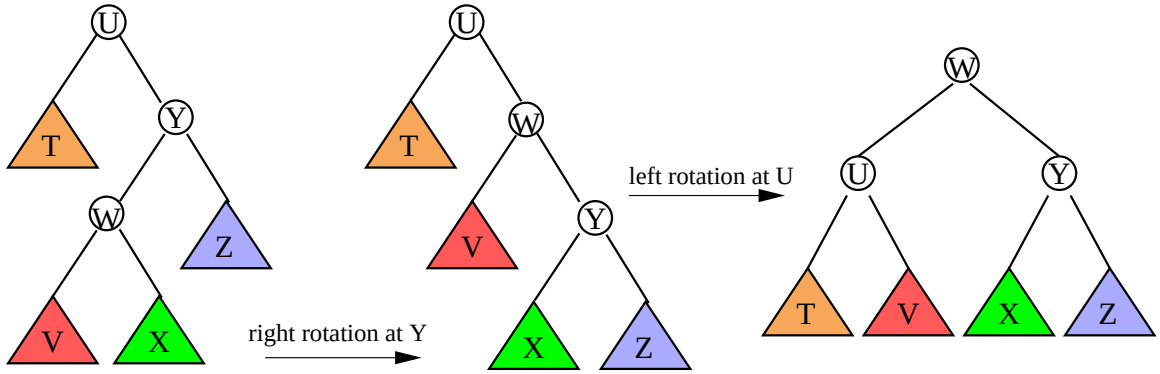

Figure 3: Double Left Rotation at U

Finally, in Figure 4, we show a sequence of insertions to an AVL tree, along with the balance restoring actions. The height of each node is indicated.

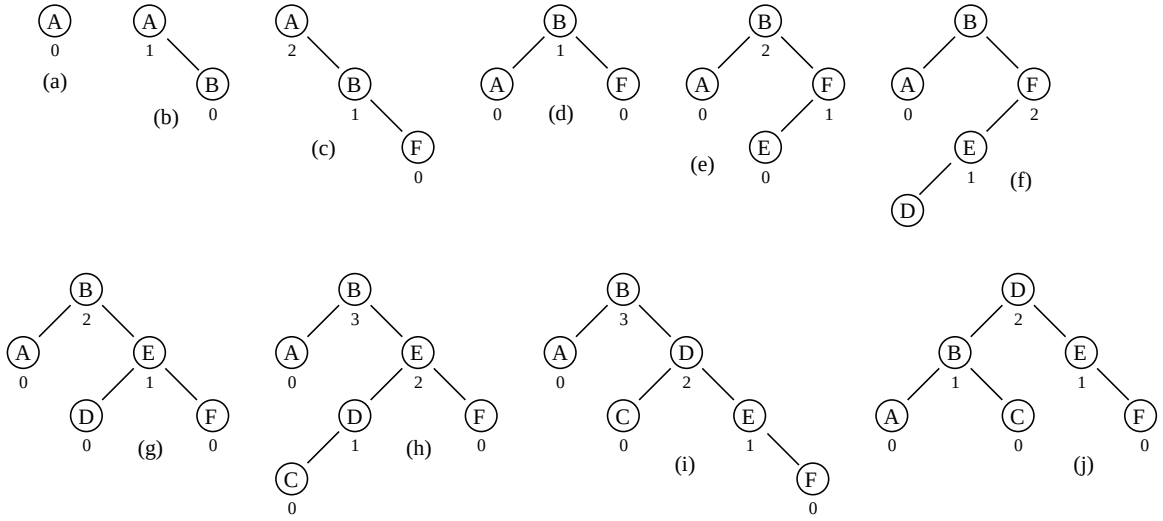

Figure 4: Sequence of Insertions to an AVL Tree with Rebalancing

Figure 4(a) shows an AVL tree of size 1, (b) shows the tree after insertion of B, and (c) shows the tree after insertion of F. The tree is now unbalanced at A. Left rotation at A yields (d). E is inserted at (e) and D at (f). The tree is now unbalanced at B and F. Since F is the lower of those two, we rebalance at F with a right rotation. At (h) we insert C, causing the balance factor of B to become 2. But since the balance factor is 1 at D, the left child of B, a left rotation at B will not suffice. Instead, we execute a double left rotation at B, consisting of a right rotation at E followed by a left rotation at B, restoring the AVL property. The worst case height of an AVL tree is $O(\log n)$, unlike the bad worst case behavior of treaps. I have reviewed several Youtube videos on the subject, and I recommend `https://www.youtube.com/watch?v=DB1HFCEdLxA`