Binary Search Trees

A *binary search tree* is a binary tree where the nodes contain data. Each datum must include a *key*, a object of an ordered type. For example, if a datum is a record concerning a person, the key could be that person's social security number. The keys are in alphabetical order, meaning that during an inorder visitation of the nodes, the data are be ordered by their keys. In our discussion below, we assume keys are unique.

A binary search tree, or any binary tree, is a *recursive* structure, and it is best to use recursion to implement the operators.

Search. "Find" could mean to find the node whose datum has a given key K, or to simply report whether that node is in the tree. We start at the root. We proceed down the tree until we either find K, or establish that it is not in the tree. Let K' be the key of the current node. If K' = K, we are done. Otherwise, we continue at the left child of the current node if K ¡ K', or the right child if K ¿ K'; the search fails if the current node is empty. This search algorithm is essentially *binary search*, hence the name *binary search tree.*

Insert. If D is a datum which is not in a tree T, we use Find to return an empty leaf of T. We then create a new node whose datum is D, which we attach at that position.

Size and Height. The number of nodes and the height of a binary tree tree can easily be computed recursively. The size of the subtree rooted at a particulare node N is one plus the sizes of the subtrees rooted at the two children of N. The empty tree has size 0. The height of the subtree rooted at N is one plus the maximum height of the subtrees rooted at the two children of N. The empty tree has default height -1.

Visitation. There are four standard orders of visitation of a binary tree T, three of which can be easily implemented by recursion.

(a) Preorder, where we visit the root first, then the left subtree in preorder, then the right subtree in preorder.

(b) Inorder, where we visit left subtree of T in preorder, then visit the root, then the right subtree in inorder.

(c) Postorder, where we visit the left subtree of T is postorder, then the right subtree of T in postorder, then the root.

(d) Level order. The level (or depth) of a node of T is the length of the longest path from the root to that node. We frist visit the root, then nodes at leven 1, then nodes at level 2, and so forth. For each level, the nodes are visited in left-to-right order. I know of no convenient recursive algorithm for implementing level order visitation.

Deletion. Deletion is by far the trickiest of the operations on binary trees. There are a number of schemes for deleting a node while maintaining alphabetic order. One such algorithm is given below.

Let N be a non-empty node of a binary search tree T.

(a) If both subtrees of N are empty, simply delete N.

(b) If just one subtree of N is empty, replace N by its non-empty subtree.

(c) If both subtees of N are non-empty, find the node K whose key is the immediate successor of the key of N, in alphabetic order, then replace N by K. The alphabtic order of T will be maintained.

One way to implement step (c) is to copy the data from K into N, then delete K. However, it is possible to implement the step by simply shifting the pointers. If each node contains a large amount of data, that could be faster than copying data.

Self-Balancing Binary Search Trees. 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 the 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 data which contain a *key*, just as before, and also a *priority*, which is a randomly chosen object from an ordered set, typically an integer. Ideally, keys are distinct, but collisions are inevitable unless the number of possible keys much larger then the number of nodes.[1] However, if there are few collisions there is no significant loss of efficiency.

A treap has two orders, alphabetic order, as is normal for a binary search tree, and 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 a key 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 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* at Y modifies T as follows:
  - The left subtree of W is unchanged.
  - The right subtree of Y is unchanged.

[1]If there are n nodes and k possible keys, the probability that there are are no duplicate keys is roughly $e^{-n^2/2K}$.

  - Y becomes the right child of W.
  - The previous right subtree of W becomes the left subtree 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 subtree of Y becomes the right subtree of W

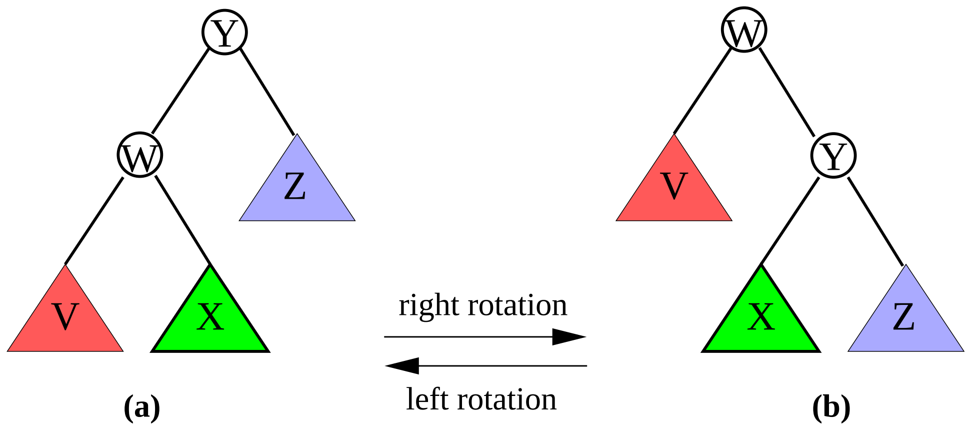

Figure 1 Right and Left Rotations.

Rotations after Insertion into a Treap. Insertion of a new node N into a treap T consists of the following steps.

(a) Insert N into T as is usual for binary search trees.

(b) If N is the root, or if the key of N is no greater than the key of its parent P, quit. Otherwise, proceed to step (c)

(c) If N is the left child of P, execute a right rotation at P, otherwise execute a left rotation at P.

(d) Return to step (b).

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.

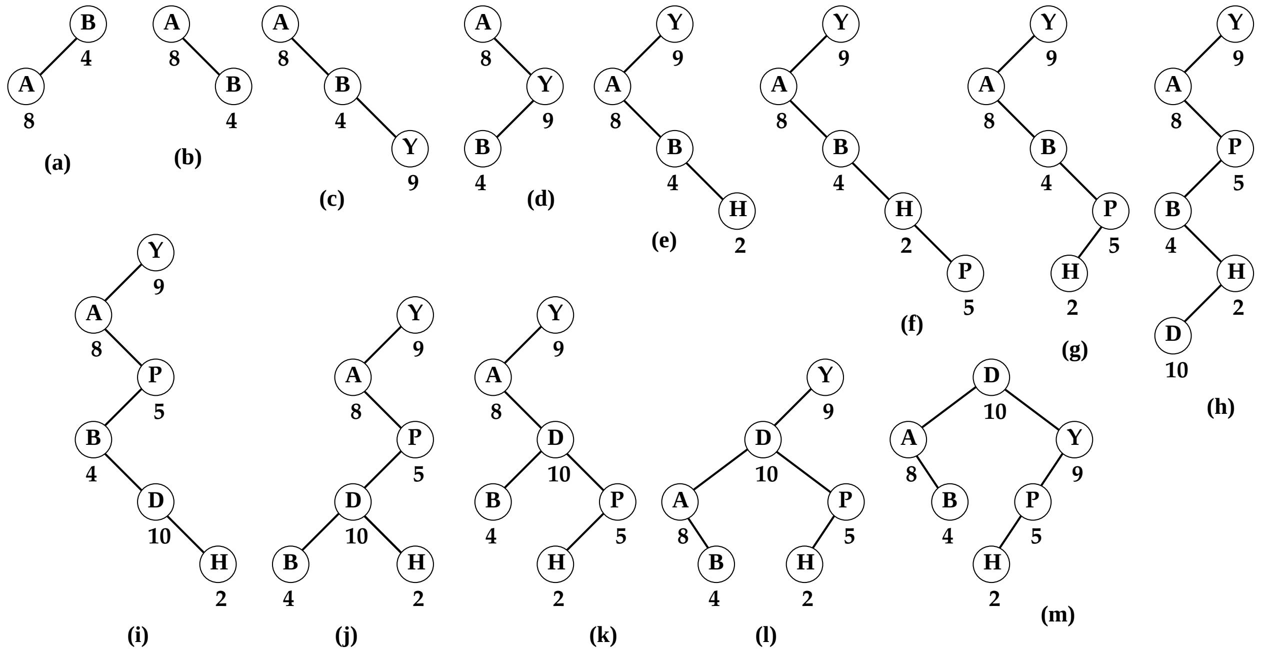

Figure 2: Sequence of Insertions into a Treap.

The steps shown in Figure 2 are as follows. Each time a key 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 $O(n)$.

Deletion from a Treap. We choose to delete from a treap using a different scheme than we used for binary search trees. Let N be a node of a treap T, and let K be the datum of node N. Our deletion algorithm at N will eliminate K in one or more recursive steps.

(a) If the left subtree of N is empty, replace N by its right subtree, and stop.

(b) If the right subtree of N is empty, replace N by its left subtree, and stop.

(c) If both subtrees are non-empty, let L and R be the left and right children of N. If the key of L is less than the key of R, execute left rotation at N. K will now be in the node at L, then recursively execute the deletion algorithm at L. Otherwise, execute right rotation at N, after which K will in R, then recursively execute the deletion algorithm at R.

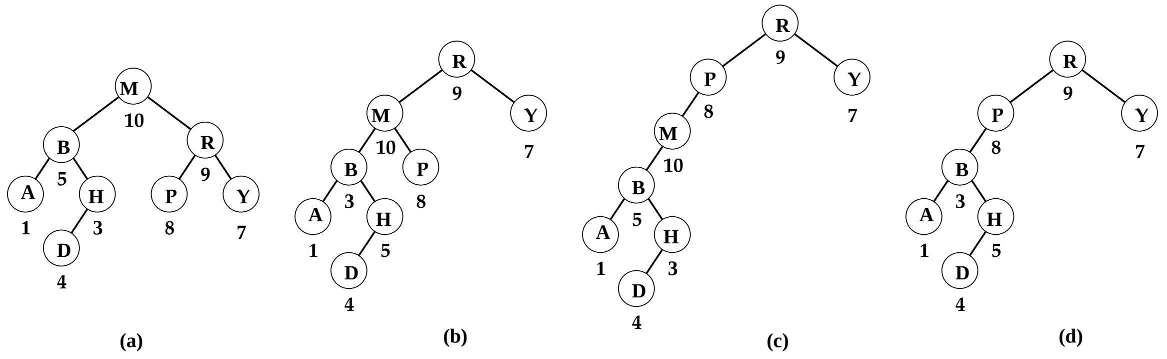

Figure 3: Deletion from a Treap

Figure 3 shows an execution of the algorithm to delete a node from a treap. The treap is shown before the deletion at (a). M is to be deleted. Since the left child of M has lower priority, we execute a left rotation at M, resulting in the treap shown in (b). At the next step, the left subtree of M has lower priority than the right child, hence we once again execute a left rotation at M, resulting in the treap shown in (c). Now, the right subtree of M is empty, hence we replace M by its left subtree, resulting in the treap shown in (c). Note that the heap condition is violated in both (b) and (c), but that doesn't matter, since M is destined for deletion.

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 height of the left subtree of X, minus the height of the right subtree 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 or

0; otherwise it is *unbalanced.* 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 right rotation, as shown in figure 4. A double right rotation at a node X consists of first a left rotation at the left child of X, followed by a right rotation at X.
4. Double left rotation, the mirror image of a double right 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, in our example, we let it be 2.. Let T be the left child of X. The balance factor at T must be 1, 0, or -1. If it is 0 or 1, a right rotation at X restores the AVL property. If it is -1, a double right rotation at X restores the AVL property. Figure 4 illustrates a double right rotation at the node X.

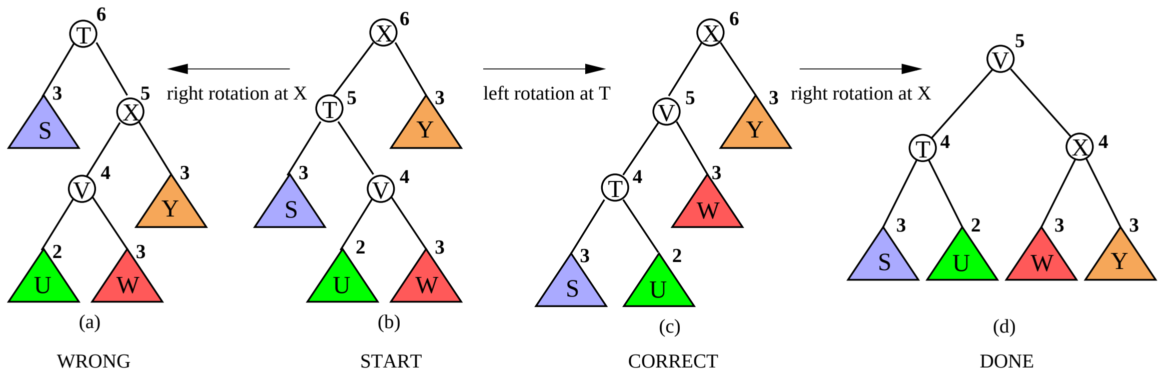

Figure 4: Double Right Rotation at X. The numbers are the heights of the subtrees. We start with the subtree shown in (b), which is unbalanced at X, with balance factor 2. If we execute a right rotation at X, we obtain the subtree shown in (a), which is still unbalanced. Instead, we execute a double right rotation at X, consisting of a left rotation at T, giving us the tree shown in (c), followed by a right rotation at X. The final subtree, shown in (d), is balanced.

Finally, in Figure 5, we show a sequence of insertions to an AVL tree, along with the balance restoring actions. The height of each node is indicated.

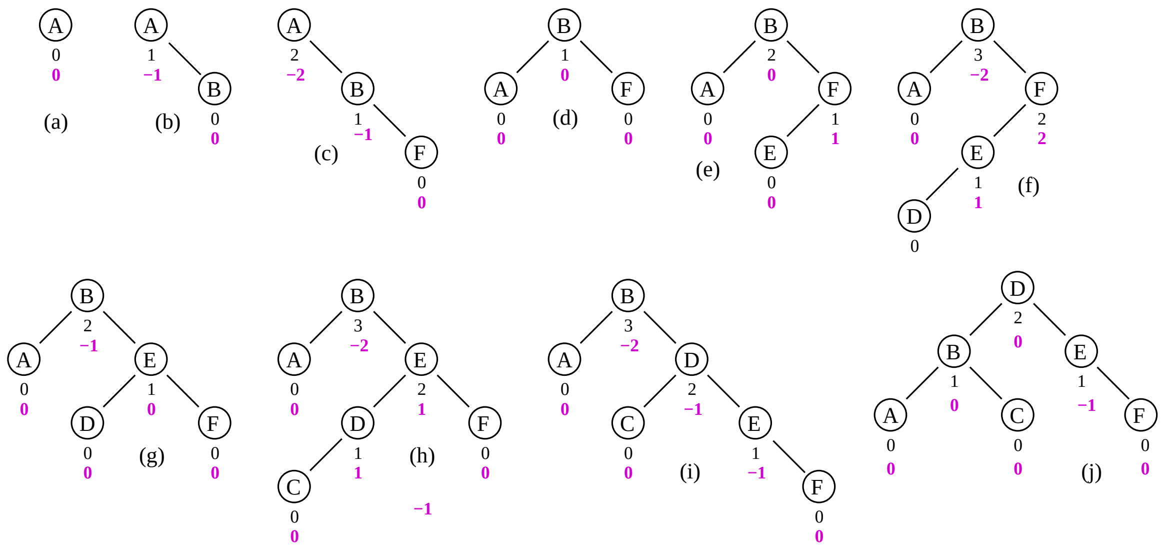

Figure 5: Sequence of Insertions to an AVL Tree with Rebalancing

Figure 5(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 now inserted at (e) and D at (f). The tree is 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 right 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 yielding (i) followed by a left rotation at B, restoring the AVL property at (j). 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`