Chapter C3: Binary Search Trees

A **binary search tree** is used to store ordered data to allow efficient queries and updates.

C3.1 Binary Search Trees

A **binary search tree** is a binary tree with values at the nodes such that:

-left descendants are smaller, right descendants are bigger. (One can adapt this to allow repeated values.)

This assumes the data comes from a domain in which there is a **total order**: you can compare every pair of elements (and there is no inconsistency such as \( a < b < c < a \)).

In general, we could have a large object at each node, but the object are sorted with respect to a **key**. Here is an example:

```
  53
 / \   
31   57
 /   /   
12  34  56 69
 /   /   /   
 5  68  69 80
```

C3.2 Insertion and Deletion in BST

To find an element in a binary search tree, one compares it with the root. If it is larger, go right; if it is smaller, go left. And repeat. The following method returns `nullptr` if not found:

```c
find(key x) {
    Node *t=root;
    while( t!=nullptr && x!=t->key )
        t=(x<t->key ? t->left : t->right);
    return t;
}
```

Insertion is a similar process to searching, except you need a bit of look ahead.

To remove a value from a binary search tree, one first finds the node that is to be removed. The algorithm for removing a node \( x \) is divided into three cases:
• **Node \( x \) is a leaf.** Then just delete it.

• **Node \( x \) has only one child.** Then delete the node and do “adoption by grandparent” (get old parent of \( x \) to point to old child of \( x \)).

• **Node \( x \) has two children.** Then find the node \( y \) with the next-lowest value: go left, and then go repeatedly right (why does this work?). This node \( y \) cannot have a right child. So swap the values of nodes \( x \) and \( y \), and then delete the node \( y \) using one of the two previous cases.

The following picture shows a binary search tree and what happens if 11, 17, or 10 (assuming replace with next-lowest) is removed.

![Binary Search Tree Diagram]

All modification operations take time proportional to depth. In best case, the depth is \( O(\log n) \) (why?). But, the tree can become “lop-sided”—and so in worst case these operations are \( O(n) \).

### C3.3 Rotations

In order to talk about “self-balancing” trees, we need the concept of rotations. A rotation can be thought of as taking a parent-child link and swapping the roles. Here is a picture of a rotation of \( B \) with \( C \):

![Rotation Diagram]

### C3.4 AVL Trees

There are several ideas to force a binary search tree to be balanced. One popular version is a red-black tree. But here we consider **AVL trees**. These are named after their inventors: Adelson-Velsky and Landis.
The idea of “self-balancing” is to add some property, contract, or invariant that is maintained. The property cannot be too close to balanced, since that would probably entail complete rebuilding of the tree too often. The idea is to come up with a condition for “nearly balanced” and such that if the tree becomes too far from balanced, simple operations make it very nearly balanced.

The invariant for AVL trees is:

\[ \text{for each node, the height of its left and right subtrees cannot vary by more than 1.} \]

Each node keeps track of its height (that is, the height of the subtree rooted at it). This value can conceptually be updated after each insertion by going up the tree. (Only the ancestors of the new node can have changed.)

◊ Insert in AVL

We now consider how to perform the insert. As usual, there is a unique place in a binary search tree where a new node can be added while preserving the binary search tree property. So the new node is added there. If the result is a valid AVL tree, we are done. But what happens if it is not?

The tree was good just before the insertion. So the imbalance in the heights at any node can be at most 2. Consider the lowest node that is unbalanced. Let’s say it is \( g \). Assume that \( g \) has children \( c \) and \( k \), where node \( c \) has height 2 more than \( k \). Since the tree was good just before insertion, node \( c \) can have only one child that is too deep.

Now there are two possibilities. The first is that the deeper child of \( c \) is on the same side of \( c \) as \( c \) is of \( g \). Say the deeper child is \( b \), and \( b \) is the left child of \( c \) and \( c \) is left child of \( g \). Then rotation of the edge \( gc \) does the trick. The root of the subtree becomes \( c \), it becomes balanced, and its height is back to what it was before the latest insertion, so the whole tree is balanced.

Here is a picture of such a rotation. (You can think of node \( a \) as just having been inserted.)
The second possibility is that the deeper child of \( c \) is on the other side of \( c \) as \( c \) is of \( f \). Say the deeper child is \( e \), and \( e \) is the right child of \( c \). Then a “rotation” of the whole \( gce \) does the trick: \( e \) becomes parent of both \( c \) and \( g \). This can be programmed as two successive rotations—first \( ec \) then \( eg \)—and thus is often called a double rotation. The root of the subtree becomes \( e \), it becomes balanced, and its height is back to what it was before the latest insertion, so the whole tree is balanced.

Here is a picture of such a rotation. (You can think of node \( d \) as just having been inserted.)

\[
\begin{array}{c}
\text{Before} \\
\text{rotation:} \\
\text{After rotation:}
\end{array}
\]

\( 
\text{gce} \\
\text{becomes} \\
\text{eg} \\
\]

\( 
\text{e} \\
\text{becomes} \\
\text{parent} \\
\text{of both} \\
\text{c} \\
\text{and} \\
\text{g} \\
\]

\( 
\text{root} \\
\text{of the} \\
\text{subtree} \\
\text{becomes} \\
\text{e} \\
\text{it becomes} \\
\text{balanced} \\
\text{and its} \\
\text{height} \\
\text{is back} \\
\text{to what} \\
\text{it was} \\
\text{before} \\
\text{the latest} \\
\text{insertion,} \\
\text{so the} \\
\text{whole} \\
\text{tree} \\
\text{is balanced.}
\)

\text{Analysis}

The result is a running time of \( O(\log n) \). We claim! Okay: so one needs a theorem.

\textbf{Theorem.} Any AVL tree of height \( d \) has at least \( f_\text{d} \) nodes, where \( f_\text{d} \) is the \( d^{\text{th}} \) Fibonacci number.

\textbf{Proof.} The proof is by induction. If we consider a semi-balanced tree of height \( d \), then one of its subtrees has height \( d - 1 \) and the other one has height \( d - 2 \) or \( d - 1 \). The first has at least \( f_{d-1} \) nodes. The second has at least \( f_{d-2} \) nodes. And so the whole tree has at least \( f_{d-1} + f_{d-2} = f_d \) nodes.

Since \( f_d \) is exponential in \( d \), this means that the height is \( O(\log n) \). One can also support the \textit{delete} operation.

\textbf{Exercises}

1. Explain how one can add an int to each node that allows a binary search tree to offer the rank operation. This method should return the value of specified rank in time proportional to the depth of the tree.