Introduction

Binary search trees (BSTs) are a concept that we have seen several times in the past. As a reminder, a binary search tree consists of nodes, each of which maintains a key value and pointers to a left and right child. Many BST implementations also maintain pointers to parent nodes. Now, the key characteristic of a binary search tree is that it maintains the following property:

**BST Property:** Any BST node’s key is greater than every key in its left subtree and less than every key in its right subtree.

Similar to hash maps and tries, binary search trees are useful data structures to efficiently map keys to values. In fact, the underlying data structure behind Java’s TreeMap implementation is a balanced binary search tree. In such a BST implementation, each node would contain a key-value pair, and the nodes would be arranged based on their keys.

![Figure 1: An example of a generic binary search tree](image)

Querying a BST

By utilizing the BST property, we can implement several operations that allow us to find specific elements in a binary search tree. The most common operations are:

1. **SEARCH**(x, k)
2. **INSERT**(x, k)
3. **DELETE**(x, k)

All of these operations take \( O(1) \) time for each level of the tree. So, if the input of the method happens to be in the lowest level of the tree, these operations will all take \( O(h) \) time, where \( h \) is the height of the tree.

Balanced BSTs: An Introduction to AVL Trees

As we have seen, many BST operations take time proportional to the height \( h \) of the tree. So if we can design a BST implementation that minimizes the height of the tree, then our BST operations will be much more efficient. This is the motivation behind balanced binary search tree implementations. In particular, balanced BSTs maintain the property that \( h = O(\lg n) \), where \( n \) is the number of nodes in the tree. This
means that our main BST operations will run in $O(\log n)$ time. Note that our BST querying operations do not change the structure of the BST, so they will remain the same for balanced BST implementations. The key changes occur in the insertion and deletion operations.

There are several balanced BST implementations, such as Red-Black trees and AVL trees. We will focus on AVL trees, but first we will discuss a concept that is common to both of these implementations: tree rotations.

**Tree Rotations**

Tree rotations are at the core of balanced BST implementations. A tree rotation is a constant time operation that changes the shape of a local area of a BST. There are two types of tree rotations, left rotations, and right rotations. Both involve making one child the new root, the root one of the children, and then swapping the “inner” subtrees of the two nodes that changed.

![Figure 2: Tree rotations in both directions](image)

By using tree rotations to preserve additional invariants for a BST, you can ensure it is balanced (or roughly balanced), and keep the favorable asymptotic running time of a BST in even the worst cases. The key difference between different balanced BST implementations is that they keep track of different invariants and values in order to determine when tree rotations are needed to restructure the tree.

**AVL Trees and the Height Balance Property**

AVL trees work by assigning a height value to every node in the tree. Leaf nodes are said to have height 1, and “null” children are said to have height 0. Then the height of every other node is 1 plus the greatest height of its two children. A BST is said to be an AVL tree if it satisfies the height balance property:

**Height Balance Property:** For every node in the BST, the heights of its children differ by at most 1.

AVL trees modify the insertion and deletion operations in order to maintain the height balance property. The exact details will be covered in lecture and in the lecture readings, but just know that these operations can be modified to still run in $O(h)$ time. And by maintaining the height balance property, AVL trees always have a height that is $O(\log n)$.

**Discussion**

Suppose we are given a BST containing $n$ distinct elements. How can we output all $n$ elements in sorted order in $O(n)$ time? What implications does this have on the running time of constructing a BST containing $n$ elements? More specifically, what must be the asymptotic lower bound on the worst case running time of BST construction?
Problems

Problem 1
Design an algorithm to decide if a given binary tree is a valid binary search tree.
Problem 2
What will the AVL tree below look like after 40 is inserted?
Problem 3
What will the AVL tree below look like after 40 is deleted?
Problem 4
Show that any arbitrary $n$-node binary search tree can be transformed into any other arbitrary $n$-node binary search tree using $O(n)$ rotations. Assume that both trees contain the same $n$ distinct elements. (Hint: First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain, which is a binary search tree where every internal node only has a right child.)

Problem 5
Perform a series of rotations to make 5 the root of the following unbalanced BST:

```
    10
   / \
  7   15
 /\  /\  
5 3 6 2
 /\ /\ /\  
2 4 1
```