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 operations that we will focus on are Search, Minimum, Maximum, Successor, and Predecessor. Note that the pseudocode for these operations can be found in Chapter 12 of CLRS.

- **SEARCH**(x, k): In this operation, we want to search for the element with key k in the BST rooted at node x. This is done by starting at node x and doing the following: If the key of the current node is equal to k, return the current element. If the key of the current node is greater than k, move to the left child of the current node. Otherwise, move to the right child of the current node. We continue this process until we find the element we are looking for or reach a leaf node and realize that the element is not in our BST.

- **MINIMUM**(x): In this operation, we want to find the element with the smallest key in the BST rooted at x. Due to the BST property, we can do this by beginning at x and following the left child pointers until we reach a leaf node u. Then we know that u will be the node with the smallest key.

- **MAXIMUM**(x): In this operation, we want to find the element with the largest key in the BST rooted at x. In this case, we begin at x and follow the right child pointers until we reach a leaf node u. Then u is the node with the largest key in the tree.
SUCCESSOR(x): This operation returns the successor of the given node x. That is, it returns the node with the smallest key that is greater than the key of x. We can break this into two cases.

Case 1: x has a right child y. In this case, we return MINIMUM(y), which will return the smallest element in the subtree rooted at y. We know that the subtree rooted at y only contains elements with keys greater than that of x, so we return the minimum element from that subtree.

Case 2: x does not have a right child. In this case, we begin at x and follow parent pointers until the current node is a left child of its parent, at which point we return that parent node. This is essentially finding the first node y such that x is the maximum element in the left subtree of y.

PREDECESSOR(x): This operation returns the predecessor of the given node x. That is, it returns the node with the greatest key that is smaller than the key of x. This is totally symmetrical to the Successor operation. As such, we can break it into two cases:

Case 1: x has a left child y. In this case, we return MAXIMUM(y), which will return the largest element in the subtree rooted at y. We know that the subtree rooted at y only contains elements with keys less than that of x, so we return the maximum element from that subtree.

Case 2: x does not have a left child. In this case, we begin at x and follow parent pointers until the current node is a right child of its parent, at which point we return that parent node. This is essentially finding the first node y such that x is the minimum element in the right subtree of y.

Note that all of these operations take $O(h)$ time, where $h$ is the height of the BST.

BST Insertion and Deletion

Now we will cover the techniques used to insert and delete elements in general binary search trees. We will not focus on how to implement insertion and deletion in balanced BSTs, as that will be covered in lecture this week.

INSERT(x, z): This operation inserts element z in the BST rooted at x. Luckily, maintaining the BST property as we insert z is fairly simple to do. We simply search for where z should be in the tree, and then insert it into the tree at that position if it is not already there. That is, we begin at x and compare its key to z’s key. Then we move to the left or right child accordingly and continue until we reach an empty/null child pointer where z should be placed.

DELETE(x): This operation deletes element x from the BST. Deletion is a bit more tricky than insertion, as there are several cases that we need to consider.

Case 1: x has no children. In this case, x is a leaf node, and we can delete it by simply modifying its parent to replace x with null as its child.

Case 2: x has exactly one child y. In this case, we delete x by letting y take the place of x. That is, we will modify the parent of x to replace x with y as its child.

Case 3: x has two children. This is the trickiest case. We can maintain the BST property by replacing x with its successor in the tree. So we find the node $y = \text{SUCCESSOR}(x)$. Since x has two children, it must have a right child, so y must be the minimum element from the right subtree of x. Now we call DELETE(y) to delete y from its original position, and finally we remove x from the tree by replacing it with y. How do we know that our recursive call to DELETE won’t also end up in this same case?

Note that both insertion and deletion take $O(h)$ time, where $h$ is the height of the BST.

BST Traversal

Sometimes it is useful to visit the nodes of a binary search tree in a particular order. This can be achieved with tree traversal algorithms. The three traversal algorithms that we will discuss are Inorder traversal, Preorder traversal, and Postorder traversal. These traversals are implemented as follows:
- **Inorder Traversal**$(x)$: Recursively visit the left subtree of $x$, then visit $x$, and then recursively visit the right subtree of $x$. So the ordering is (Left, Root, Right).

- **Preorder Traversal**$(x)$: Visit $x$, then recursively visit the left subtree of $x$, and then recursively visit the right subtree of $x$. So the ordering is (Root, Left, Right).

- **Postorder Traversal**$(x)$: Recursively visit the left subtree of $x$, then recursively visit the right subtree of $x$, and then visit $x$. So the ordering is (Left, Right, Root).

As you will see later, the Inorder traversal algorithm is particularly useful when using BSTs. Also note that these algorithms visit each element in the BST exactly once and do a constant amount of work at each element. So in a BST containing $n$ elements, these algorithms will run in $O(n)$ time.

**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 $O(lg n)$ 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.

![Tree rotations](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 tree labeled with node heights](image)

Figure 3: An example of an AVL tree labeled with node heights

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?

**Solution.** We can output the elements in sorted order by simply performing an Inorder traversal on the BST and outputting the elements as we visit them. This traversal will take just $O(n)$ time.

With this idea, we can design a new sorting algorithm called BSTSort, which sorts $n$ elements by constructing a BST containing those elements and then performs an inorder walk in the BST. Now remember that any comparison based sorting algorithm must run in $\Omega(n \log n)$ time in the worst case. This means that BST construction must also run in $\Omega(n \log n)$ worst case time. Otherwise, if BST construction ran in $o(n \log n)$ time, then BSTSort could sort in $o(n \log n)$ time because BST construction would take $o(n \log n)$ time and inorder traversal takes $O(n)$ time.
Problems

Problem 1
Design an algorithm to decide if a given binary tree is a valid binary search tree.

Solution. This problem is a bit trickier than it may at first seem. The first thing that may come to mind is to simply visit every node in the tree and check that the current node’s key is greater than the key of its left child and less than the key of its right child. However, this algorithm does not work on the following tree:

```
  3
 / \
2   5
 / \
1   4
```

We can see that every node is in the correct order in relation to its immediate children, but 4 should not be in the left subtree of 3. So we will have to refine our algorithm a bit.

We want to use a similar algorithm, but we need a way of knowing the valid values that can appear in the current subtree of the BST. So we can define a recursive algorithm that begins at the root of the tree and works down. Each recursive call takes in a node to visit, and a minimum and maximum value that define the range of values that can appear in the subtree rooted at that node. We begin at the root \( x \) and set \( \min = -\infty \) and \( \max = \infty \). Now we check the keys of the children of \( x \) to make sure that they are in the correct order in comparison to \( x \) and ensure that the keys are in the range strictly between \( \min \) and \( \max \). If these conditions are not met, then return false. Otherwise, recursively call this algorithm on the left child of \( x \) with the same \( \min \) value but with \( \max = x.key \). We also recursively call the algorithm on the right child of \( x \) with the same \( \max \) value but with \( \min = x.key \). If this procedure visits the entire tree without returning false, then the binary tree is indeed a BST.

Another solution, which is particularly elegant, is to simply perform an inorder traversal on the given tree and check that the elements are visited in sorted order. If they are not in sorted order, then the tree is not a BST. Otherwise, it is a BST. This gives us an easy \( O(n) \) time algorithm to check if a binary tree is a binary search tree.
Problem 2

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

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

Solution. First, rotate the root right

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

Then, rotate the root right again

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

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.)

Solution. Suppose we want to transform the tree \(T_1\) into \(T_2\). Our first objective is to transform \(T_1\) into a right-going chain. We can do this by strategically applying right rotations to \(T_1\). Consider the chain of nodes in \(T_1\) that results from beginning at the root and following right child pointers until you reach a null element. We will call this our partial right-going chain. Note that the partial chain must contain at least 1 element because the root itself must be part of the chain. We will construct the full right-going chain by applying right rotations in \(T_1\) that add a node to our partial chain each time (think about how this works). Note that each rotation will add 1 to the length of the partial chain. Since the partial chain starts with length at least 1 and there are \(n\) elements in the tree, we will need at most \(n - 1\) right rotations to construct the full right-going chain.

Now, by the same reasoning, we can also convert \(T_2\) to the same right-going chain by applying at most \(n - 1\) right rotations. But note that any right rotation can be reversed by a corresponding left rotation. In other words, the sequence of operations used to convert \(T_2\) to a right-going chain can be inverted to convert a right-going chain to \(T_2\). Thus, we can convert \(T_1\) to \(T_2\) by converting \(T_1\) to a right-going chain and then applying the sequence of rotations that convert the right-going chain to \(T_2\). This will require at most \(2(n - 1) = O(n)\) rotations.