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.

**Problems**

**Problem 1**
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?

**Problem 2**
Design an algorithm to decide if a given binary tree is a valid binary search tree.

*(Hint: use the solution from the previous question)*
A Return to Greedy Algorithms

We have seen many greedy algorithms throughout the course. They make seemingly exponential problems solvable in polynomial time. One example of this phenomena is Dijkstra’s Algorithm. Dijkstra’s algorithm finds the shortest path from a source vertex to a target vertex in $O((n + m)\log(n))$ time using a greedy choice of relaxing edges of a locally optimal shortest path node we find at the root our min heap. On the other hand, a very similar problem such as finding the longest path between a source node and a target node is NP-hard and only exponential time algorithms are known to solve the problem deterministically (refer to CIS 262 and CIS 320 for more on computational complexity theory).

As you can see, greedy algorithms are extremely powerful. Here is a brief review on the properties on greedy algorithms.

### Definition (Greedy Algorithms)

A greedy algorithm obtains an optimal solution to a problem by making the choice that seems ‘the best’ at the moment. It is a heuristic strategy that does not work all of the time, yet for certain problems, it produces an optimal solution.

### Greedy-choice Property

The key ingredient to greedy algorithms is the greedy-choice property. This properties states that we can assemble a globally optimal solution by making locally optimal choices. This means that when we are considering a choice in our problem, we will always make the choice that is the best in our current situation without considering any future problems that we may encounter.

You can think of this as a ‘bottoms up’ approach. Greedy algorithms will solve sub problems one by one, choosing what is best at the current iteration, until it finds a globally optimal solution for the entire problem. For any greedy algorithm to be valid, we need to show that a greedy choice at each step yields a globally optimal solution. We can do this with the exchange argument.

### Definition (The exchange argument)

We first examine some globally optimal solution to our problem. We call this optimal solution $O$. We want to show how to modify $O$’s construction to substitute our greedy choice for some other choice in the problem that results in a similar but smaller sub problem. If we can show that the optimal solution to our problem includes our greedy choice along with the same optimal solution to a smaller subproblem, then we can ensure our greedy solution is correct.

If you want to learn more about greedy algorithms, please read CLRS Chapter 16.1 and 16.2 for a more in depth analysis.

Test Your Understanding

**Problem 1. The Scheduling Problem**

Consider a set of tasks $i_1...i_n$ that need to be run on a single machine. Each task has a required time it takes to complete the task denoted as $t_i$ which must be contiguous, and a deadline time $d_i$ which task $i$ must be completed by. The machine can only run one task at once.

When a task is completed late, we consider the difference in completion time and its deadline to be that task’s ‘lateness’. The goal of our algorithm is to find some ordering to the tasks that minimizes the maximum ‘lateness’ of any given task.