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 = 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?

**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.

**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)*

**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 $\text{max}$ value but with $\text{min} = x.key$. 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.
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 \(O((n + m)\lg(n))\) 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.
Solution. Let us denote the maximum 'lateness' of any given ask as $L$. There are multiple greedy strategies which we may think of when we encounter this problem.

**Idea 1:** We could run the tasks in order of increasing length ($t_i$). However we notice quickly that this ordering completely ignores deadlines and will not minimize lateness.

**Idea 2:** We could run tasks in increasing order of $d_i - t_i$. In other words, run the tasks in increasing order of 'slack time', the time the task must be started by in order to be completed by the deadline. However, this doesn’t succeed either. Consider task 1 with $t_1 = 1$, $d_1 = 2$, and task 2 with $t_2 = 10$, $d_2 = 10$. We end up running tasks with low slack time earlier, which can cause tasks with early deadlines to be late.

**Idea 3:** Another simple idea is to run the tasks in increasing order of their deadline. This greedy choice turns out to minimize $L$, despite its simplicity. *But why does this work?* It seems like this choice is bound to fail on some input. In order to prove this, we will begin with an optimal solution, $O$ and modify it without changing the fact that it is optimal until it matches the scheduling produced by our algorithm.

**Claim 1:** There is an optimal scheduling with no idle time:

First, we’ll establish that an optimal scheduling $O$ has no 'idle time'. Idle time is a period in our schedule such that no task is currently being worked on. If there was idle time, later tasks could simply be moved earlier by the duration of the idle time, either maintaining or decreasing the total lateness of our schedule, meaning our schedule wouldn’t have been optimal to begin with.

**Claim 2:** All schedules with no inversions and no idle time have the same max lateness:

We say that a schedule $S$ has an inversion if a job $i$ with deadline $d_i$ is scheduled before another job $j$ with earlier deadline $d_j < d_i$. We note that our algorithm produces a schedule with no inversions.

Now, let’s prove this claim. If two different schedules $S_1$ and $S_2$ have neither inversions nor idle time, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical deadlines are scheduled. Let’s consider such a deadline $d$ shared by multiple jobs. In both $S_1$ and $S_2$, the jobs with deadline $d$ are all schedule consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines). Among the jobs with deadline $d$, the last one has the greatest lateness, and this lateness does not depend on the order of the jobs.

**Claim 3:** There is an optimal schedule with no inversions and no idle time:

To prove this claim, we will need to prove the following lemmas.

**Lemma 1:** If $O$ has an inversion, then there must be two adjacent tasks $i$ and $j$, $i$ scheduled before $j$, such that $d_j < d_i$. The proof of this is simple. Let’s consider each task in consecutive pairs in our optimal schedule. If no such pair existed, then the schedule would be in strictly non-decreasing order of deadlines, meaning it had no inversions to begin with.

**Lemma 2:** After swapping consecutive jobs $i$ and $j$ that create an inversion, we see that we get a schedule with one less inversion. This follows directly from lemma 1. Suppose $O$ has at least one inversion, and let $(i, j)$ be the inversion between consecutively scheduled tasks we showed must exist above. If we swap the scheduling of the tasks $i$ and $j$ to create $O'$, we eliminate an inversion from our new schedule, and don’t create any new ones.

**Lemma 3:** Let $O$ be an optimal scheduling with an inversion and let $O'$ be the resulting schedule of swapping consecutive jobs $i$ and $j$ that create the inversion, then $O'$ has a maximum lateness no larger than that of $O$.

Consider each task $r$ to have a start and finish time, $s(r)$ and $f(r)$ respectively. Let the lateness of task $r$ in $O$ be $L_r$, and let $L = \max_{r} (L_r)$, the maximum lateness among all tasks in $O$. Similarly call the corresponding values from $O'$ $s'(r)$, $f'(r)$, $l'_r$, and $L'$. Consider again the two adjacent, inverted tasks, $i$ and $j$. The finish time of $j$ before the swap is equal to the finish time of $i$ after the swap, so all other jobs besides $i$ and $j$ do not have their lateness changed. Because $j$ was moved earlier in the schedule, its lateness does not increase, so the only task which can possibly become more late is $i$. Task $i$’s lateness in $O'$ becomes: $l'_i = f'(i) - d_i = f(j) - d_j$. However, task $i$ can’t possibly become more late in $O'$ than task $j$ was in $O$, since by our inversion assumption, $d_i > d_j$, so $l'_i = f(j) - d_i < f(j) - d_j = l_j'$. Since the only lateness that could possibly be changed between $O$ and $O'$ did not increase, this implies that the total lateness did not increase from swapping an inversion.
From lemmas 1, 2, and 3, we can directly see that we can start with any optimal scheduling $O$, take all of the inversions in $O$ and swap consecutive inverted jobs repeatedly to find an optimal schedule $O'$ with no inversions and no idle time.

Claim 4: The schedule created by our greedy algorithm has optimal maximum lateness $L$. This implies directly from claim 3. We can simply take an optimal scheduling, swap all inversions out, maintaining optimality, and obtain an optimal schedule with no inversions. We see that if a schedule has no inversions, then jobs in the schedule are in order of non-decreasing deadline. This means that we can obtain a optimal schedule by creating a schedule based on the increasing order of each task’s deadlines. Thus, the scheduling produced by our algorithm is optimal by the exchange argument.