**Homework 02 -- Questions 1 - 5**

1. What is binary tree and binary search tree? Discuss their similarities and differences.

A binary tree is a tree that satisfies the following two properties: Each internal node of the tree (nodes with at least one child) has at most two children, and the children of a node are an ordered pair, where one is the left child and the other is the right child (in a proper binary tree, each internal node has exactly two children). A binary search tree has these properties as well, but it has an additional property. In a binary search tree, the value stored at any node is always greater than the value of its left child and is always less than the value of its right child. Thus, it is possible for the left child of a node containing 8 to contain 9 in a binary tree, but this is not possible in a binary search tree.

2. How can we traverse the nodes of a tree? In which scenario we should use which traversal mechanism?

We can traverse the nodes of a tree in three different ways: preorder, postorder, and inorder traversal. In preorder traversal, each node of the tree is visited and then its children are visited by recursively calling the preorder traversal for each child of the parent node. This method of traversal is useful for printing a structured document, where each piece of information should be printed within its respective category or subcategory.

In postorder traversal, first each child of a node is visited by recursively calling the postorder traversal method, and then the parent node is visited. In essence, descendant nodes are always visited before their parents, and the root of the tree will always be the last node visited. This method is useful for any case where we want to retrieve the external items before the internal items, since it always visits the left subtree of a node, then the right subtree of the node, and finally the node itself. Computing space used by files in a directory, deleting a tree, or getting the postfix expression of an arithmetic tree are all cases where we would use postorder traversal.

In inorder traversal, each node is visited after its left subtree and before its right subtree. That is, if a left subtree exists, the inorder method is recursively called on the left child, then the node itself is visited, and finally, if a right subtree exists, the inorder method is recursively called on the right child. Because inorder traversal relies on defined left and right subtrees, it is only useable for binary trees. Inorder traversal is most useful for binary search trees, as it retrieves all of the nodes in increasing order.

3. What are the advantages of binary search over a linear search?

A binary search’s advantage over a linear search comes from its runtime advantage. In the worst case scenario, a linear search has 0(n) runtime complexity, whereas a binary search has 0(log(n)) runtime complexity. This is because a binary search eliminates half of the searchable elements for every operation. The search method visits the middle element and compares it to the target. If it is greater than the target, the right half of the remaining elements is removed, and vice versa if it is less than the target. This makes the binary search much faster than the linear search, although it only works on sorted data.

4. What is a priority-queue? Discuss some use cases of priority-queue.

A priority-queue is list which stores a collection of entries. An entry is defined as a pair of elements where the first element is a key and the second element is a value. Each entry’s level of priority in the priority-queue is determined by its key, and thus the keys must be comparable objects (the item with the “lowest” key is considered highest priority). Depending on the implementation of the priority-queue, entries can either be sorted when they are added to the collection so that the first value in the collection always has the lowest key (insertion sort), or the entry with the lowest key can be found when an item needs to be removed, in which case the list remains unsorted. Alternatively, the priority-queue can be implemented using a heap, in which case the lowest key is stored at the root of the heap.

Priority-queues are useful for any situation in which the elements stored in a list need to be removed in a particular order based on the precedence of each element. An auction, for example, needs to keep track of each bidder while ensuring that the top bidder has priority for receiving the item. Another example where a priority-queue might be useful is a guest list where there are VIP slots. The priority-queue would ensure that VIPs have precedence over non-VIPs.

5. What is a heap? What is the advantage of the heap over a stack? What is the time complexity to get the minimum item from min-heap?

A heap is a binary tree with nodes storing keys (comparable values) such that every internal node that is not the root always has a key that is greater in value than that of its parent (meaning the child node is lower priority). It is also a complete binary tree, meaning that at the depth (h – 1) = i in the tree (the depth just before the furthest depth) there are 2i nodes, and of the nodes at depth i, if any are external, they are to the right of the internal nodes. Essentially, this means that a heap is filled out row by row from left to right when key are inserted, and the last node is the rightmost node at the height of the tree (maximum depth).

The heap has several advantages over the stack. The stack, being linear, does not benefit from the hierarchical structure of a tree like the heap does. Thus a stack cannot achieve 0(log(n)) runtime complexity for certain operations involved in sorting and moving values. The heap is also a more flexible data structure than the stack, as stack memory needs to be stored in a continuous block of memory space, while the nodes of a heap can be stored anywhere in memory. This means that while a stack can run into a stack overflow error, a heap’s only constraint is the amount of memory in the computer itself.

The removeMin() method for a heap runs in 0(log(n)) time complexity. The first step of this method is to swap the positions of the root node with the last node in the tree. Since the heap is implemented using an array, this is an 0(1) operation. Then, the method removes the last node from the heap, which is also an 0(1) operation. Finally, the heap must restore its order using the downheap support method, which continually swaps nodes downward in the tree as long as the key of the current node is greater than the key of its child. This final operation has a runtime of 0(log(n)) because the height of the heap is equal to log(n) (n is the number of nodes). Thus, the entire method has 0(log(n)) runtime complexity.

**Programming Problems (6 – 10)**

Question 6:

Discussion:

The program HW2Question6.java contains the code for this problem. The algorithm for returning the index of the target number in the array “nums” (or -1 if the target does not exist) uses an ArrayList and a binary search tree (we can use a binary search algorithm since the list is sorted in ascending order). The ArrayList is used solely for the functionality of the “indexOf” method, which allows us to easily return the index of the target in the array if it is found. The binary search tree is what allows us to search for the target in the array efficiently.

The binary search tree takes advantage of the fact that the value of the node to the left of a parent node is less than that of the parent, and the value of the node to the right of a parent node is greater than that of the parent. This allows the search for the target integer to have a runtime of 0(log(n)), as opposed to 0(n) for a sequential search. Adding each node to the tree has 0(n) runtime since adding each item to the binary search tree in the proper location requires recursion and Master’s Theorem gives us 0(n) (Source: Time Complexity). The space complexity is 0(n) since each item needs a spot in both the ArrayList and the binary search tree.

Verification:

We will verify the program using an array of integer values and several different target values, including values not in the array. The program should output the index of the target if it exists, or -1 if it does not.

int[] nums = {-1, 0, 3, 5, 9, 12}

Input: 9

Output: 4

Input: 0

Output: 1

Input: 12

Output: 5

Input: 6

Output: -1

Input: -500

Output: -1

Question 7:

Discussion:

The program HW2Question7.java contains the code for this problem. To simplify the problem, we create an Interval class as a blueprint for the meeting time intervals. This class implements the Comparable interface, allowing us to compare intervals. The comparison works as follows: If a meeting falls entirely before another meeting, it is considered “less-than” the other meeting. If a meeting falls entirely after another meeting, it is considered “greater-than” the other meeting. If two meetings overlap at all, they are considered “equal.”

These comparison results allow us to leverage the functionality of the binary search tree to determine the necessary number of conference rooms. Each conference room is represented by a binary search tree. When attempting to add a meeting to the schedule using the addMeeting() method, we attempt to add the meeting to each binary search tree in our list of conference rooms. My implementation of the binary search tree throws an IllegalArgumentException whenever the element being added to the tree is “equal” to an existing element in the tree (compareTo() returns 0). Thus, we can use a try-catch statement to detect this error and determine if a meeting cannot fit in a particular conference room. When a meeting does not fit in any existing conference rooms (or there are no conference rooms), the addMeeting() method creates a new conference room and adds the meeting to it. The return value of this method is 0 if the meeting fits in an existing room and 1 if a new room is required. Thus, we can take an ArrayList of Intervals and run each through the addMeeting() method, adding the result of the method to the room counter after each iteration to obtain the minimum number of rooms required.

The runtime complexity of this algorithm is 0(n2). Adding a new node to the binary search tree uses recursion and achieves 0(n) runtime by master’s theorem. Since we run this for each interval in the list of intervals, the total runtime complexity is 0(n2). The space complexity would be 0(n), since each of the ArrayLists and the binary search trees have a space complexity of 0(n).

Verification:

We will verify the program using several test cases of input intervals that have overlapping and non-overlapping times. The program should print a message whenever a meeting does not fit in a particular room and then provide us with the total number of rooms needed.

Input: meetingTimes = {{7,10}, {2,4}, {0, 30}, {5, 10}, {15, 20}}

Output: Meeting [0, 30] does not fit in room 0

Meeting [5, 10] does not fit in room 0

Meeting [5, 10] does not fit in room 1

Rooms required: 3

Input: meetingTimes = {{0,500}, {501,600}}

Output: Rooms required: 1

Question 8:

Discussion:

The program HW2Question8.java contains the code for this problem. In order to sort the elements in order from largest to smallest, we use a priority queue with a reversed comparator such that the largest key value is the highest priority. We then add each integer in the array to the queue and then remove the largest elements before the kth largest using a for loop, and then print the kth largest element by removing it from the queue.

The runtime complexity for this algorithm is 0(n\*log(n)). The priority queue is implemented using a heap, and returning the minimum element from a heap has a runtime complexity of 0(log(n)). Since we must do this until we find the kth largest element, the total runtime complexity is 0(n\*log(n)). The space complexity of the algorithm is 0(n), since we only need to store each element from the array as a node in the heap.

Verification:

We will verify the algorithm using several test cases with large and small positive and negative integers, different k values, and arrays with duplicate values. The program should always return the value of the kth largest element.

Input: nums = {3,2,1,5,6,4}, k = 5

Output: 2

Input: nums = {-4000, 3000, 2, -1, 7000, 406, 23}, k = 3

Output: 406

Input: nums = {-40, 35, 2, -1, 78, 406, 78, 78, 23}, k = 7

Output: 2

Question 9:

Discussion:

The program HW2Question9.java contains the code for this problem. To find all of the values in the tree that are between the lower and upper bounds (inclusive), we run a for-loop from the lower bound to the upper bound and on each iteration call the search() method of the binary search tree. If it returns true, we add that value to the sum. This gives us a final sum of all the values between the bounds that exist in the tree.

The runtime complexity for this algorithm is 0(n\*log(n)). Searching for a value in the binary search tree has runtime 0(log(n)), and we do this for each value in the interval. The space complexity is 0(n) since we simply store all the values in the binary search tree.

Verification:

To verify this algorithm, we test several binary search trees with different upper and lower bounds. The program should correctly return the sum of all the values in the interval.

Input: root = {10, 5, 15, 3, 7, null, 18}, low = 7, high = 15

Output: Sum: 32

Input: root = {10, 5, 15, 3, 7, null, 18}, low = 1, high = 14

Output: Sum: 25

Input: root = {10, 5, 15, 3, 7, 13, 18, -3, 4, null, 8, 11}, low = 5, high = 13

Output: Sum: 54

Input: root = {10, 5, 15, 3, 7, 13, 18, -3, 4, null, 8, 11}, low = 15, high = 18

Output: Sum: 33

Question 10:

Discussion:

The program HW2Question10.java contains the code for this problem. To find a valid pair of integers in the array that add up to target, the program uses the method sumTarget(), which takes the list of numbers and the target as parameters. The method first stores all of the values in the list in a new binary search tree. Then, for each number in the list, it computes the difference between that number and the target and searches the binary search tree for that difference, ensuring that if it does exist, its index is not the same as the index of the original number (we cannot use the same number twice). If it finds a valid pair, it prints the indices of the two numbers that add up to the target. Otherwise, it informs the user that no such pair of numbers exists.

The runtime complexity of this algorithm is 0(n\*log(n)). The for-loop in the sumTarget() method runs once for each item in the list of numbers, and for each item, searching the binary tree has a runtime of 0(log(n)). Therefore, the total runtime complexity is 0(n\*log(n)). The space complexity of this algorithm is 0(n) since both the ArrayList and the binary search tree have 0(n) space complexity.

The only issue with this algorithm is that because it uses a binary search tree, our list of numbers cannot contain any duplicates. A different algorithm that would solve this problem would use the same outer for-loop over the list of numbers and still compute the difference between each number and the target, but instead of using a binary tree, we use the indexOf() method of the ArrayList. By checking that the index of the difference is not -1 in the list and that it is not equal to the index of the original number, we are able to find the pair of numbers even if there are duplicate values. This algorithm would have a runtime complexity of 0(n2), however, so I chose to implement the binary search tree solution.

Verification:

To verify this algorithm, we will use several test cases with positive and negative numbers, both large and small. The program should always return the indices of the two values that add up to the target.

Input: nums = {2,7,11,15}, target = 9

Output: [0,1]

Input: nums = {-3,6,23,4}, target = 20

Output: [0,2]

Input: nums = {-95712,5812,481923,-589183}, target = -107260

Output: [2,3]

Input: nums = {-95712,5812,0,-589183}, target = -107260

Output: No such pair exists.

Works Cited

“Time Complexity of Recursive Functions [Master Theorem].” *Algorithms to Go*, Yourbasic, <https://yourbasic.org/algorithms/time-complexity-recursive-functions/>. Accessed 13 March 2023.