BILKENT UNIVERSITY CS 202 COURSE HOMEWORK 2

NAME: Simay Uygur **ID:**22203328
SECTION: 1 **DATE:** 06.11.2024

Question 5

My implementation for the first question was filling an array of 5 elements with the name queryArray and holding the least five elements of the heap. The array is updated every time an insertion or deletion happens because the query method will not change any item in the queryArray. Because of that logic, everytime query method is called it only returns the elements of queryArray. This code is O(1) time considering only print lines for five elements and writing to the output file when necessary; it does not increase or decrease according to the size of the heap. My approach for only querying was to print the array's items in a for loop; with k elements, it becomes O(k) complexity (it may be assumed O(1) for small values). So, it is still efficient for larger values, but not O(1) asymptotically.

Also the method fillQuery that fills the queryArray every time after insertion and deletion has different time complexity. Its logic is taking the first element to the queryArray, then comparing the other two and taking the least one in those, and lastly, according to the current size, comparing elements until the fifth level elements and taking the least three. That approach compared $2^5 - 1$ elements, which is the maximum number of elements until the fifth level. Therefore, queryArray has five elements independent of the heap size, and time complexity will be O(31), which is asymptotically O(1).

Nevertheless, looking at and comparing more than five levels takes more time as we compare every node until that level k of the tree. If all nodes until level k are checked, a total of $2^k - 1$ elements will be checked. It is still applicable but not efficient to fill the queryArray for larger k because the heap does not have a sorted logic between its left and right nodes like bst.

In total, the overall complexity of methods that prepare the queryArray and print contents is O(k + $2^k - 1$) ≈ O(2^k). This complexity increases exponentially and does not apply to larger k values.

Question 6

In this homework I learned many tree structures such as max and min heaps, balanced binary search trees, AVL trees. All tree implementations have different advantages that fit the problem requirements and I learned specific features of those data structures. The tree's height makes most of the problems have small time complexity which a balanced binary search tree, AVL tree and heap has. This advantage stems from their structure because they are balanced and have logN height for total N elements. So addition and deletion operations take logN time at most.

However, finding a specific key in a heap is inefficient because it is not sorted like a binary search tree and we can't search in that fashion. Therefore, to not increase time complexity I implemented a balanced bst for the problem that requires searching for a specific key. However, heaps are powerful in retrieving the maximum or minimum element because they return the element in the first index of the heap's array if the heap is not empty. Therefore, for most of the questions I used that feature of heaps.

For the first question, without removing any element during querying makes the problem different. Heap doesn't hold the least five elements in its first five indexes (min-heap). A search or sorting is needed for the first five levels, since heap's root only contains the minimum element of its children and children's children. Therefore, I wrote 2 methods for querying, one named queryFiveElements() only prints and writes to the output file the contents of the queryArray[5] in a for loop. The other method, fillQuery traverses the heap's first thirty one elements and compares and changes the query's inside if a less element is encountered. First two elements can be obtained from the heap's three elements.So it takes asymptotically O(1) time for the both methods because it always traverses the first thirty one elements of the heap not without being affected by the heap size, O(31 + 5)≈O(1).

For the second question, I implemented a CardGame class containing the play() function, constructor, destructor, bst instance variables and merge sort function. I used a balanced binary search tree for this problem as searching is more efficient than heap and its length is minimized. CardGame constructor calls create the binary search tree and merge sorts of the given Bobo and Holosko card arrays in the constructBalancedBst(int* array, int size) function. This function adds the sorted array's elements recursively. It first adds the middle element of the array to minimize the tree's height. Then, continues to add mids of the right and left subarrays. This is important because, in the game, there must be a search that checks the maximum element of the opponent's deck that is less than the maximum of the player's deck. Playing optimally happens when, in his turn, the player takes their maximum in the bst and searches in the opponent's bst. A method called searchForLessBiggest(int key) is made for that logic. Because of the binary search tree's structure, while searching, going to the right or the left subtree or taking the root of that subtree depends on the given argument's key value. This search takes O(logN) time. Also, the bst class has methods that return the max element for the player to call a search function with that value after the value is retrieved and the compared cards are removed from both bsts. This loop continues until the end and returns both players' scores. Considering the time complexity, there are 2N cards, and each player has N cards so for playing N cards are discarded from both bsts. Each deletion from bst takes logN time maximum because the tree's length is fixed logN at construction with the sorted array. Also, merge sort takes NlogN time in every varying N. Constructing the bst is composed of adding to bst and binary searching for the middle element to add. Adding elements at most takes logN time for the least elements as the tree's height is logN, and binary search takes N time because, in total, it adds N elements but in a different way. So constructing bst takes O(NlogN) time, merge sort takes O(NlogN) time, and playing takes O(NlogN) time.

Retrieving the maximum in bst happens by going to the rightest leaf node in the bst which is the minimum element in the tree, at most it takes logN time because tree's height is logN maximum. Deleting the specific key in the opponent takes logN time, and deleting the maximum in the player's bst takes logN time. So, for N times, this process happens, each happening in O(logN) time and one after another. O(logN+logN+logN+logN)≈O(logN) for one turn, total time complexity becomes O(NlogN) for playing. Preparing the balanced bst also takes O(NlogN) time; adding those will not affect the time complexity. Therefore, the final complexity is O(NlogN).

For the third question, I implemented a PrefixArrPlay class and this class contains Heap and MaxHeap instance variables. I used heap in this problem because using a fixed heap with size M gives logM complexity which is wanted in the problem. Also, adding a key and deleting from the top of the heap take logM time at most because of the heapify() function. In my implementation solution is found while binary searching the index. Every time it calls the doesSatisfyConditon() method it constructs a min-heap for array A and max-heap for array B. It adds elements both of the heaps until the given index and copies heap to temporary arrays, and compares M elements from both arrays whether tempA's element is bigger than tempB's element at the same index. If it is true for M elements one solution is found and binary search for the solution moves to the middle of the left subarray. If not, binary search moves to the middle of the right subarray. Merge sort takes MlogM time (it is less than NlogM, so we can ignore), adding and deleting in doesSatisfyConditon() takes NlogM time at most if current index is at the end of the array. Searching solution takes logN time, overall complexity becomes O(logN(NlogM)) ≈ O(NlogNlogM).

For the fourth question, I implemented a SubArrPlay class which contains Heap and MaxHeap instance variables. I used heap in this problem because adding/deleting is efficient in this structure. Additionally, the best feature is retrieving the top element of the heap which has O(1) complexity and it enables an easy way of comparing the top elements to check whether the condition is satisfied. I constructed new heaps every time and added elements until the last index that we are searching. In one loop, I filled both of the heaps until the min(M, K), then I filled until max(M,K) and the right pointer was set to this value. Then the method starts to increase the right pointer and fill heaps until that right pointer. Then without changing the left pointer, the right index is increased and checked until it satisfies the condition. Until it reaches the end of the array, iteration is over for that specific left pointer. Everytime the minimum length is checked and updated if a less value is found. Then the left pointer is one incremented and this process repeated again. For N elements, the left pointer goes until the N-max(M,K) index. Each iteration inserts a new element to both of the heaps and deletes from the top of the heaps. This insertion and deletion only happens in maxHeapB, if B[rightPointer] <maxHeapB.getTopElement(), and in minHeapA, if A[rightPointer]> minHeapA.getTopElement(). Filling the heap until the right pointer is at the end of the array takes O(Nlog(max(M,K)) time, loose bound is O(NlogN) time. The outer loop increasing the left pointer takes O(N - max(M,K)) time and the loose bound is O(N). The overall complexity becomes O($N^2 logN$) in my solution.