Algorithms Laboratory (CS29203)
Lab Test - II
Department of CSE, IIT Kharagpur

2nd November 2023

Question-1 (30 points)

Consider a binary tree with n nodes. Let there is a non-negative weight (ω_i) associated with each node of the tree, and the total sum of weights of all nodes in the tree is equal to n, i.e. $\sum_{i=1}^{n} \omega_i = n$. Our goal is to make the weight of each node exactly equal to 1 by moving (or transferring) weights from one node to other. However there is some restriction on weight transfer among the nodes. In one move, we can choose two adjacent nodes and transfer 1 unit of weight. Hence the weight transfer can be performed from parent to child, or from child to parent.

Our ultimate goal is to find the **minimum** number of moves required so that each node has weight value **exactly** equal to 1.

You can assume that the input is taken in an array where the node weights are defined in level order from left to right and a missing child is denoted as '-1'. For example, consider the following array representing a binary tree: [3, 0, 0]. That is, the root node has weight 3, and both of its children have weight 0. In this case, the minimum number of moves required is 2 to make the weight of all nodes 1. From the root of the tree, we can perform one movement to transfer 1 unit weight to its left child, and 1 unit weight to its right child.

Important: You are **not** allowed to implement array based representation of trees (you have to use standard linked representation of trees).

Example 1:

```
(Input) [0, 3, 0]
(Output) 3
```

```
Explanation: From the left child of the root, we transfer 2 unit of weights to the root by making two moves.
Then, we transfer 1 unit weight from the root of the tree to the right child.
```

Example 2:

```
(Input) [1, 0, 0, -1, 3]
(Output) 4
```

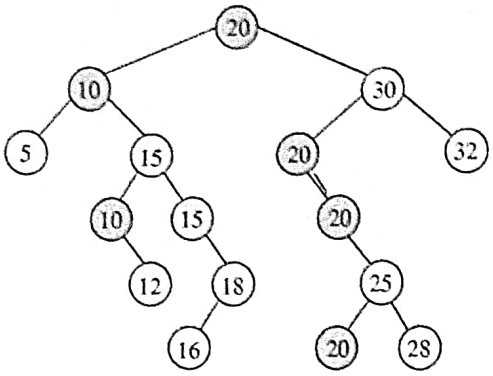
```
Explanation: Transfer 1 unit weight from the root of the tree to the right child. Then transfer
2 unit of weights from the right child of the left child of root (i.e. 2 moves), and finally transfer
1 unit of weight from the left child of the root to the root node.
```

Question-2 (40 points)

In this question, you will be dealing with Binary Search Tree (BST) with duplicate entries (*i.e.* storing same key value multiple times). For any node x, we will have:

Values at all nodes in the left subtree of x < Value at x ≤ Values at all nodes in the right subtree of x

This means that we always push repeated values to the right subtree of a current occurrence. An example of a BST with repeated values is shown in the figure below.



Implement the following functions on a BST T supporting duplicate entries. Assume that each node in the tree contains only a key value and two child pointers (and nothing else like parent pointers or any other information). We will let n be the number of nodes in T, and h the height of T.

- Search(T, x): This function should return the count of occurrences of x in T. The return value 0 means that x is not present in T. This function must run in O(h) time, independent of the number of occurrences of x in T.
- Insert(T, x): Irrespective of how many instances of x are already present in the tree, this function will insert a fresh occurrence of x in T. The standard BST insertion routine is followed with the only exception that the detection of the presence of x would not break the search loop. The loop will instead reach a suitable node q, and the new key value is inserted as a suitable child of q. At this point, the new inserted node must be a leaf in T. This function should also run in O(h) time.
- Delete(T, x): This function should delete all occurrences of x from T in a total of O(h) time. The function would first locate the topmost occurrence of x in T. If no such occurrence can be found, there is nothing to be done. Otherwise, the function will remember (a pointer to) this topmost occurrence. It will then make a single downward pass in order to delete all the remaining occurrences of x. By construction, all these occurrences lie in different levels, and have their left subtrees empty. So deleting the duplicate entries is easy (O(1) time per node). After the duplicate entries are removed, the function deletes the topmost occurrence (the only occurrence left now) as in the standard BST deletion procedure.

Important: You are **not** allowed to implement array based representation of trees (you have to use standard linked representation of trees).

Example 1:

```
(Input) 1, 1, 1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 7, 8, 8, 9, 9, 9, 9, 9, 10, 11, 11, 11, 12,
        13, 13, 14, 16, 16, 17, 18, 18, 19, 19, 19, 19

Search ( 12) =  1
Search ( 18) =  2
Search (  5) =  4
Search ( 12) =  1
Search (  6) =  0
Search ( 12) =  1
Search (  7) =  1
Search ( 17) =  1
Search ( 12) =  1
Search ( 20) =  0

Delete ( 17): 1, 1, 1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 7, 8, 8, 9, 9, 9, 9, 9, 10, 11, 11, 11, 12,
              13, 13, 14, 16, 16, 18, 18, 19, 19, 19, 19

Delete (9):   1, 1, 1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 7, 8, 8, 10, 11, 11, 11, 12,
              13, 13, 14, 16, 16, 18, 18, 19, 19, 19, 19

Search (9) =  0
```

Question-3 (30 points)

Given an array of N distinct integers your task is to find $k(<N)$ largest elements in the array. A naive way can be first sorting the array in ascending order and then take the last k elements. This will be $O(N \log N)$. Another way can be using the quicksort partitioning algorithm to find the pivot element at $(N-k)^{th}$ position and then taking everything to its right as the answer. This will give $O(N)$ average case and $O(N^2)$ worst case complexity. However, we would like you to perform the task using a min-heap. For this, the first step would be to push k elements in a min-heap. Then for each of the rest of the $N-k$ elements, you need to decide whether to push it to the min-heap, given that I would like to maintain the size of the heap always as k. Think what you need to do if you decide to push. The idea is, at the end the, k element min-heap will be our answer.
It will help if you work out an example first before jumping to code. Lets our array be [51, 43, 93, 18, 42, 99, 54, 2, 74]. Let $k = 3$. Try to see, if you push k elements from the array how does the min-heap look. Then for the rest of the elements, decide whether you need to push it to the heap keeping in mind that the heap size should remain k till the end. Then see if the final heap gives you the answer. I am providing an example run of how the program may come out to be.

```
Enter the value of k: 3
The original array: 51 43 93 18 42 99 54 2 74
3 largest elements: 74 99 93
```

Note that the output is not necessarily in sorted order. They are just the k largest elements which can be in any order. You can hardcode the input array (getting keys as input from the user is not necessary). If you need to take a maximum size/capacity of the heap, you can take it as 100. Some more example runs are given below.

```
Enter the value of k: 3
The original array: 89 500 22 -7 81 -19 34 21 222 54 68 -100 88
3 largest elements: 89 500 222

Enter the value of k: 5
The original array: 89 500 22 -7 81 -19 34 21 222 54 68 -100 88
5 largest elements: 81 88 89 500 222
```

You may use the following helper functions in your implementation, if required. Note, this assumes start index of the heap as 1.

```
int Root(){
    return 1;
}

int Parent(int n){
    return n/2;
}
```

```
int LeftChild(int n){
  return 2*n;
}

int RightChild(int n){
  return 2*n + 1;
}
```

```
bool HasParent(int n){
  return n!=Root();
}

bool IsNode(int n){
  return n<=size;
}
```

(Not for credit) Try to think what would be the runtime complexity of the approach you are taking in this problem.
Extra bit: There is a simpler way of doing this by creating a max-heap and then poping k times. If you think closely this will be $O(N) + O(k \log N)$. $O(N)$ for heapifying and $O(k \log N)$ for k pops. But we don't want you to do this way.