<u>EITANBROWN 346816549</u>
<u>YONI RUBIN 648831051</u>

Question 1 Suggest a data structure A that contains distinct natural numbers that supports the following operations that run in the worst time complexity of O(log(n)).
Explain your choice.
Insert (A,x) – Inserts x into A
Delete (A,x) – Deletes x from A
CountInRange(A,a,b) – Returns the amount of numbers in the range [a,b].

Balanced Binary Search Tree (that has the height as a variable stored in each node)
Insert would be O(log(n)) since at worst it's inserting into the bottom
Delete would be O(log(n)) since at worst it's deleting from the bottom

CountInRange if the current node is <= than a, go to the right (if equal add 1 to the count). If the current node is >= than b, go to the left (if equal add 1 to the count).
If at node N we are greater than a, all of the left children in a's right subtree will be greater than a as well, so we don't need to check them. Therefore we will keep going along the right path of node N's right subtree until we reach b (if ever). If we reach a value greater than b, we will go to the left until we are back into the range.

```
{Basic
function CountInRange(node, a, b):
if node is null: return 0
count = 0
if node.key >= a and node.key <= b: count += 1
if node.key > a: count += CountInRange(node.left, a, b)
 if node.key < b: count += CountInRange(node.right, a, b)
return count
}
```

Question 2 Suggest a data structure for storing whole numbers that supports the operations of a normal stack (push, pop, top, isEmpty) as well as an additional operation GetMax. Getmax returns the maximal value without removing it from the data structure. For Example: After inserting the following values 5, 4, 3, 6 and removing one element the data structure will contain 5,4,3 and the operation top will return 3 and the operation GetMax will return 5. All the operations must run in constant time – O(1) Explain your suggested data structure and describe how you would implement each of the operations.

```
Create MainStack
 Create MaxStack
// Push an element x onto the stack
Function push(x):
   MainStack.push(x)  // Push x onto MainStack
   If MaxStack.isEmpty() OR x >= MaxStack.top():
// Push x onto MaxStack
      MaxStack.push(x)
   Else:       MaxStack.push(MaxStack.top())  // Repeat the current max value
—------------------------------------------------------------------------------------------- O(1)
// Pop the top element from the stack
Function pop():
   If MainStack.isEmpty():
      Return
   MainStack.pop()
   MaxStack.pop()
—-------------------------------------------------------------------------------------O(1)
Function top():
   If MainStack.isEmpty():
      Return None
//Top element of stack
   Return MainStack.top()
—--------------------------------------------------------------------------------------------O(1)
// Check if the stack is empty
Function isEmpty():
   Return MainStack.isEmpty()  // Return whether MainStack is empty
—--------------------------------------------------------------------------------------------O(1)
// Get the maximum value in the stack without removing it
Function GetMax():
   If MaxStack.isEmpty():
      Return None
   Return MaxStack.top()
—--------------------------------------------------------------------------------------------O(1)
```

Question 3
We want to construct a data structure that supports the following functions:
Init(A) – The function accepts an array A containing n distinct numbers and builds the data structure. Note: The function is only called once. The runtime complexity must be Θ(n).
Avg(i,j) – The function accepts two indexes i,j (1 <= i <= j <= n) and returns the average of the elements A[i], A[i+1],…,A[j]. The runtime complexity must be O(1). Implement the functions in pseudo-code and explain the run time complexity of each function. If you use additional data structures, what is the additional memory needed (as a function of the size of the data) needed for the algorithm?

Plan: add value from I to J into value and for every index that gets added count++;
Once we hit J then divide by count and return Avg;

```
//GLOBAL – for a prefix sum array
P=[] {Additional memory of O(n)}
function Init(A) — Θ(n).
   n = length(A)
   P = array of size (n + 1)
   P[0] = 0
   for i from 1 to n
      P[i] = P[i-1] + A[i-1]
   end for
end function
—-
function Avg(i, j) → O(1)
   sum = P[j] - P[i-1]
   count = j - i + 1
   average = sum / count
   return average
end function
```

Question 4 Write an algorithm that accepts an array containing n distinct whole positive numbers and a whole number k, (k ≤ n) and returns the sum of the k largest numbers in the array. For example: Given the array {8,14,7,12,42} and k =3 the algorithm will return 68 (42+14+12). You can use any data structure or algorithm taught in the course without the need to implement them. Your Algorithm needs to run in O(n log k) time.

```
Function SumLagrestNumbers(Arr, N, K)

//Min heap size K
createMinHeap(K)


//We loop thru first k elements
For i =0 to k-1
insert(heap, Arr[i])
---------------------------------------
//We loop through the remaining elements to compare them to numbers we have and since min
heap mean the root is smallest value in the whole heap we compare it to that
For i = k to n-1
        If Arr[i] > root(heap)
        Remove Root(heap)
        Insert(heap, Arr[i])
---------------------------------------
//Now we know the largest number so add to a sum var and return it
Sum = 0;
While(!heap.Isempty)
Sum = Sum + Remove Root(heap)
---------------------------------------
Return Sum
```

{Inserting elements into a heap will take O(Log K) and since we do process N elements it will be O(N Log K)

Question 5 Suggest a data structure for saving both positive and negative integers. The structure must be able to contain more than one appearance of a number and support the following functions in the given time restraints.
a. Inserting a new number into the structure in O(log n) time.
b. Deleting a number in the structure in O(log n) time.
c. Searching for a number in the structure in O(log n) time.
d. Determine if in the structure there is a pair of opposite numbers (for example 3 and -3) in O(1) time.

BST where the root is 0, the left subtree is an AVL tree containing only negative numbers, the right subtree is also an AVL tree but containing only positive numbers. Like all balanced BSTs, insertion, deletion and searching, are all O(log(n)).
To determine if there is a pair, every node will have a section, aside from the numeric value, that will indicate if there is a pair or not as the tree is being built with the default state being false. While we are inserting +x into the right subtree, we will check if -x exists in the left sub tree. If it exists, both nodes will have their pair indicator set to true. This way when we want to check if a random node has a pair, we only need to check the section of that node that says true or false.

Question 6
In the relevant cell in the following table, write the most efficient run-time complexity for the worst-case scenario for performing the operations on the given data structure.

Operation needed → Given data structure ↓	Finding the 3rd largest ivalue	Inserting a new value	Finding a value
1. A single linked list sorted in non descending order	O(n)	O(n)	O(n)
2. Binary Tree	O(n)	O(n)	O(n)
3. Max-Heap	O(1)	O(log(n))	O(n)
4. AVL Tree	O(log(n))	O(log(n))	O(log(n))

Explain your answers

- Finding the 3rd largest ivalue
    - Single linked sorting non-descending

The last ‘index’ in the list is pointing to null, keep going down the list until next->next->next->null == true
This is O(n) because you are traversing down the entire list (technically O(n-1) :))

- Binary Tree

Store the first 3 values in auxiliary variables (O(1)). Transverse through the whole rest of the tree, when there is a value that is greater than one (or more) of the 3, replace the smallest of the 3 with the new value. After going through the whole tree (O(n)) and swapping the value, O(1), the smallest value of the 3 will be the 3rd largest element in the tree
O(n)

- Max-Heap

To find the third largest value in a max-heap, first look at the root (biggest number) and its two children. The second biggest number is the larger child of the root. The third biggest is either the smaller child of the root or one of the children of the second biggest number. *{because root access is 1, next to children is 1, checking children of second largest is at most 2 anyway so also 1}*
O(1)

- AVL Tree

Assuming that each node has a parent field, just go down the tree hugging the right side. When you reach the bottom (node.right==null) do parent of current node, then parent of that node (grandparent of the last node). This will be the 3rd largest value
Since this is an AVL tree it is as close to being balanced as can be making this O(log(n)) (or O(log(n)+2 :D)

- Inserting a new value
    - Single linked sorting non-descending

O(n), make a new node containing the value(s). Go down the list until you get to the place where the new node should be inserted. The new node will point to the node that the node that will be behind it is currently pointing to. Make the node that will be behind the new node point to the new node.
At worst you're inserting at the end of the list which is O(n)

-
    - Binary Tree

Since this is an binary tree not a BST, order doesn't matter, insert into the nearing open space
This is O(n) since at worst it is a straight line (at best the tree would be balanced making it O(log(n))

-
    - Max-Heap

This is O(log(n)) since max-heaps are always balanced.

-
    - AVL Tree

This is O(log(n)) since AVL trees are always balanced.

- Finding a value
    - Single linked sorting non-descending

O(n), simply transverse through the whole array until you find the value and return true, come across a value that is greater than the one you're searching for to return false, or get to the point where the current node you are checking is pointing to null indicating the end of the list meaning it's time to return false.
O(n)

-
    - Binary Tree

preOrder (root left right) transversal of the tree, O(n), if the value you're searching for is there return true, else continue the transversal.

-
    - Max-Heap

preOrder transversal of the tree. If you reach a value that is less than the value you're searching for, stop searching in that subtree.
O(n) since you might need to search the whole tree and not find it

-
    - AVL Tree

O(log(n)) it's a balanced tree, worst case is you're searching at the bottom level