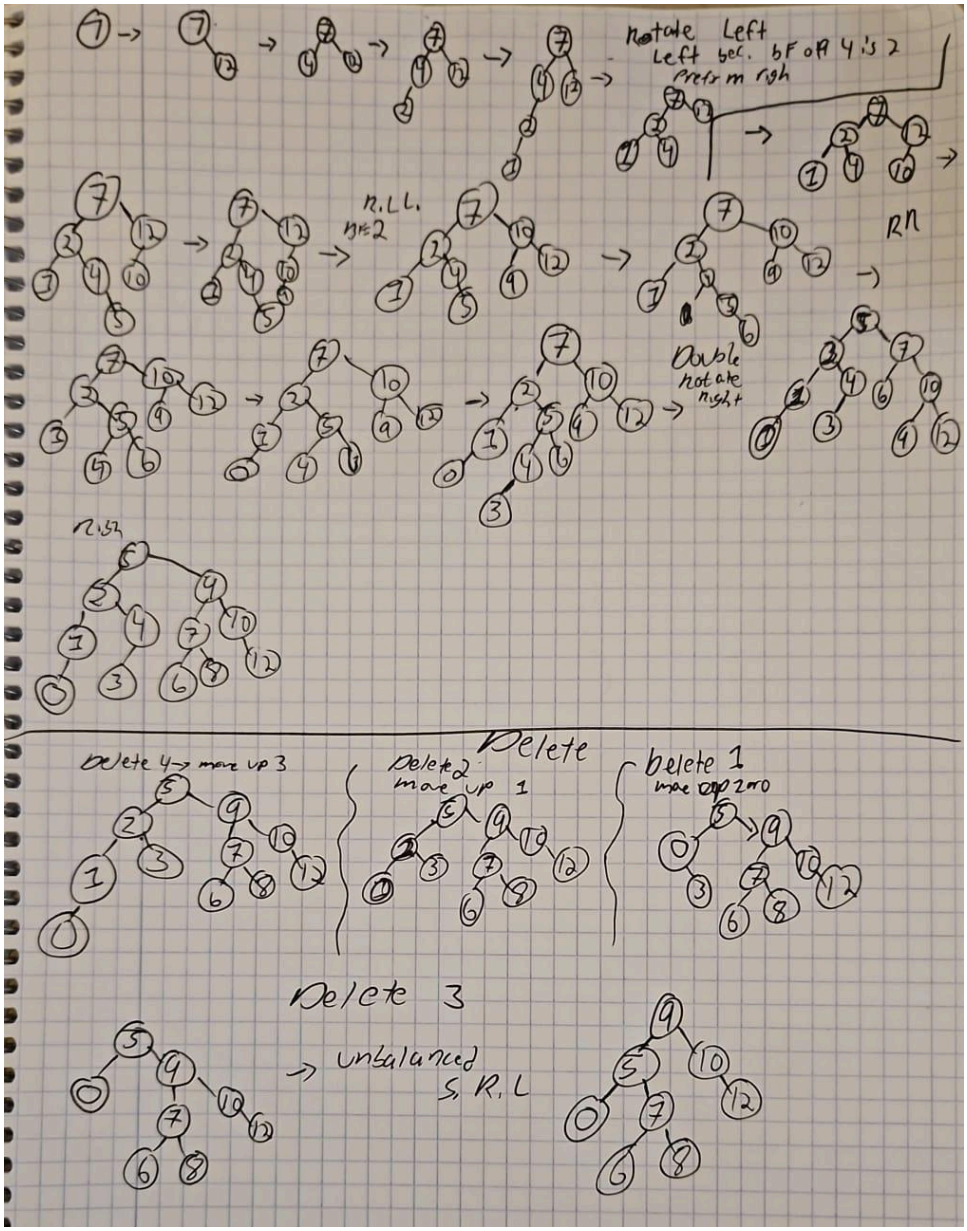
EITANBROWN 346816549
YONI RUBIN 648831051

Question 1

a. Build an AVL tree from the following values. Draw the tree after each insertion and write which rotation you used when a rotation was needed. 7,12,4,2,1,10,5,9,6,0,3,8
b. Delete from the above tree the following values. Draw the tree after each deletion and write which rotation you used when a rotation was needed. 4,2,1,3



Question 2

Given: Two binary search trees, T1 containing n1 numbers and T2 containing n2 numbers. Write an algorithm that accepts T1 and T2 and creates an AVL tree containing the union set of the numbers in O(n1+n2) time.

```
H_Lt3 = height of left subtree
H_Rt3 = height of right subtree
BF(x) = H_Lt3-H_Rt3
//Helper
AVLTest(T)
// Update height of this ancestor node
node.height = 1 + max(getHeight(node.left), getHeight(node.right))
// Get the balance factor of this ancestor
node balanceFactor = getBalanceFactor(node)
// Left Left Case
if balanceFactor > 1 and value < node.left.value return rightRotate(node)
// Right Right Case
if balanceFactor < -1 and value > node.right.value return leftRotate(node)
// Left Right Case
if balanceFactor > 1 and value > node.left.value node.left = leftRotate(node.left) return
rightRotate(node)
// Right Left Case
if balanceFactor < -1 and value < node.right.value node.right = rightRotate(node.right) return
leftRotate(node)
return node
—----------------
AVLTree(T1,T2)
Newtree t3 = NULL
list1 = []
list2= []
//Place element of t1 in ascending order into list1
inorder_traversal(T1.root, list1) //O(n1)
//Place element of t1 in ascending order into list2
inorder_traversal(T2.root, list2) //O(n2)
for each value in list1 //runs n1 times
        newTree = insert(newTree, value) //O(1)
        AVLtest(newTree) //log(n1)
for each value in list2 //n2 times
        newTree = insert(newTree, value) //O(1)
        AVLtest(newTree) //O(log(n2))
```

//log because worst case scenario we need to go from the last leaf (all the way at the bottom) all the way to the root which is log(n)-1
return newTree

Total time is n1+n2+log(n1)+log(n2)
n1,n2→ ∞ the log functions are obsolete
Total time = O(n1+n2)

Question 3

a. What is the maximum number of nodes possible in a AVL tree of height h. Explain.
$2^{h+1}-1$
Same formula for a complete balanced tree

b. Given a BST can you transform it into a AVL tree by using only rotations? Explain.
Yes, it would just be a matter of how many operations you are willing to do
All rotating does is reformat the tree, it does not change any values

Question 4

How can you implement a queue using a priority queue?

Queue:

Data: PriorityQueue
InsertionOrder: 0

Enqueue(Queue, element):
//Insert element with current priority
Insert(Queue.data, element, Queue.insertionOrder)

//Increment insertion order
Queue.insertionOrder: Queue.insertionOrder + 1

Dequeue(Queue):
// Delete and return the element with the smallest priority
return DeleteMin(Queue.data)

Peek(Queue):
//Return the element with the smallest priority without removing it
return PeekMin(Queue.data)

Question 5

Suggest an algorithm that runs in O(n lg k) time to merge k sorted lists into one sorted list. n is the total number of elements in all the k lists together. (hint: use a minimum-heap)

```
function mergeKSortedLists(lists):
//will use min heap (aka PQ)
//Each element in the heap contains the value and the index of the list it comes from. This will help us keep track of which list the element came from so that we can fetch the next element from the same list in the code
  MH = createMinHeap()

  //Insert the first element of each list into the min-heap
  //Lists.size-1 is essentially k because it can change per users choice so k can be 10 or 52
  for i from 0 to lists.size - 1:
    if lists[i] is not empty:
      //first element in the i-list at index i
      insert(MH, (lists[i][0], i))

 //initialize the resulted list
  result = []

 //We Extract elements from the min-heap and build the result list
  while MH is not empty:
    (val, listIndex, EleInd) = extractMin(MH)
    result.append(val)

    //If there is a next element in the same list, insert it into the min-heap
    if EleInd + 1 < lists[listIndex].size:
      NextEl = lists[listIndex][EleInd + 1]
      insert(MH, (NextEl, listIndex, EleInd + 1))

      //print result list
      return result
```

Question 6

Given a max-heap of size n, implemented using an array, containing distinct values.
Parent: i/2||| Left 2i ||| Right 2i+1
a. In which indexes (indices , for non-americans 😊) of the array can the minimum value be found? Prove your answer.
In a max-heap, we already know that the min-value is generally the leaves since for all internal nodes, the parents must be greater than their children.
The leaf nodes start from index $\lfloor n/2 \rfloor + 1$ to n, where n is the size of the heap and n/2 is the parent and since one of the nodes after this index IS a leaf one of these values must be the min value

b. In which indexes of the array can the third smallest value be found? Prove your answer.
Without the values not completely straightforward we focus on the leaves and their immediate parents since they are out likely candidates and it would not be constrained to a specific index range but likely is found among the leaf nodes and potentially nodes in deeper levels of the heap since those are the smallest

c. In which indexes of the array can the fourth largest value be found? Prove your answer
The fourth largest value must be less than the top three largest values but greater than the remaining values. (no shizzbuckets)
In a max heap → the largest values typically found at root(1), and its two children (2,3) the next largest values are 2s children (4,5) and 3s children (6,7)
SO choices are either 4,5,6,7 and we would have to make an algorithm to compare those values to see which is > than the other values

Question 7

Perform the algorithm for bucket sort on the following array <0.33, 0.11, 0.53, 0.8, 0, 0.94, 0.2> Show the stages of the algorithm.

//step 1 - make them bucket
createBuckets() - since range between .1 and 1.0 we will make 10 buckets (although only 7 are needed/ should be used)

// Step 2: Create Buckets for i from 0 to n-1 → bucketIndex = floor(array[i] * k) append array[i] to buckets[bucketIndex]
.33 x 10 =3, .11 x 10=1, .53 x 10 = 5, 0.8 x 10 = 8, 0 x 10 = 0, .94 x 10 = 9, .2 x 10 = 2
Bucket 0: [0], Bucket 1: [0.11], Bucket 2: [0.2], Bucket 3: [0.33], Bucket 4: ([empty]), Bucket 5: [0.53], Bucket 6: ([empty]), Bucket 7: ([empty]), Bucket 8: [0.8], Bucket 9: [0.94]

// Step 3: Sort Each Bucket for i from 0 to k-1 sort buckets[i] using a suitable sorting algorithm like insertion sort
BUT IN OUR CASE WE HAVE 1 PER BUCKET SO NO NEED TO SORT AGAIN

// Step 4: Concatenate Buckets -> put them into the sorted array
Result: [0,0.11,0.2,0.33,0.53,0.8,0.94]

//return sorted array from step 4
return sortedArray

Question 8

Given 2 arrays of size n, A[1..n], B[1..n] containing natural numbers.

Given : $\forall i, 1 \leq A[i], B[i] \leq n^3$**.**

Describe how you can check if the values in array A are identical to the ones in array B, in θ**(n) time.**

```
//adjacent to counting sort
function IdenticalArrays(A, B, n)

// Step 1: Initialize the count array with offset to handle negative indices, which never happens so we have offset to make sure every value is qualified properly
   offset = n^3

//we will initialize the count to zero
   count = array of size 2 * n^3 + 1

// Step 2: Count occurrences in array A
   for i from 1 to n
      // Increment the count for the current element of A just like counting sort
      count[A[i] + offset] = count[A[i] + offset] + 1

   // Step 3: Count occurrences in array B
   for i from 1 to n
      // Decrement the count for the current element of B again just like counting sort and we decrement bec we assume same value as in A
      count[B[i] + offset] = count[B[i] + offset] - 1

   // Step 4: Check for mismatches
   for i from 0 to 2 * n^3
     // If any count is not zero, arrays are not identical
      if count[i] ≠ 0
         return false

  // If all counts are zero, arrays are identical
   return true
```