Divij Mehra 999705225
Edward Moore 999947919
Ramaneek Gill 1000005754
Seung Hoon Lee 996033956

CSC263 Assignment 1

1. a) The best case complexity for the algorithm occurs when every element in the array is 0. In this case, no matter what the size of the array is, the algorithm will make 2 accesses to the statement A[p] = 1. Once to the first while loop and once to the second loop. Thus, the algorithm has a best case run time of O(2) or constant time.

b) The worst case complexity for the algorithm occurs when every element in the array is 1 except for the last element which is a 0. The first while loop is accessed a total of floor(√n)times. The second while loop is also accessed a total of floor(√n)times [Even though the last element is 0, the second condition is still checked. This means the total is O(√n + √n) which is O(2√n).

c) Sample space S_n = {A_i) = [1,2, …..n], i = 0,1,2…n}
Average Case is E[tn], E[tn] = {R_i . Pr(A)i}
Probability distribution of Sn =Pr[(A_i)] = 1/(n+1)
i = x_1 + x_2 + ….x_n where n = A.length

R_i = km – 1 if 1 <= k <= n
R_i = 0 otherwise

= (km – 1)/(n+1) + 1/(n+1)
= (km)/(n+1)

---

2. We will use an augmented AVL tree to that implements MEDSETS. This will be done by having an instance variable that belongs to the AVL tree that keeps track of the median node and will update it if required after each insertion and extraction.
To do this we will be using the rank of each node to refer to it's position in a sorted 'list', rank relies on inorder traversal so the smaller the rank the smaller the key.

We will not be repeating code for the insertion and deletion functions for an AVL tree, instead we will highlight the changes that need to be made to the existing code.

```
Object AVL TREE NODE:
   //instance variables
   this.height //height of tree starting at this node and going down
```

```
    this.left //left child node
    this.right //right child node
    this.parent //parent node

    //Here is the augmented instance variables to a standard AVL tree node:
    this.rank; //rank of this node, this is the number of nodes
           //preceding x in an inorder tree walk plus 1 for itself
    this.median //pointer to the median node of the ENTIRE tree


Cases median has to be updated:


    median gets deleted/extracted:
        //Instead of DELETE simply over writing the node to be delete it just have it return it
        //this implementation is omitted because it is a simple temporary variable be return at the end of the DELETE function
        old_median = DELETE(M, M.median)
        //now the median gets deleted and the tree is rebalanced accordingly if needed

        //OS-SELECT will return a node with the given rank in O(log n)
        M.median = OS-SELECT(M, floor((height-1)/2)) //this function is defined in the textbook on page 361
        //new median is created

        return old_median

new node gets inserted:
        //Will not be repeating unnecessary code, only will be highlighting changes that need to be made

        if tree is empty: //instead of INSERT(M, x) simply inserting the first node we also assign the median node for the tree
            M = TreeNode(x)
            M.median = M

        //During the insertion process have the algorithm update the rank of the new node and nodes being affected by rotation
        //refer to the chapter 14 for a simple computation of updating the node's rank in constant time

        //now going to update the median node of the tree this is done RIGHT after the rotation is completed:

        if tree.size is even: //implies tree.size used to be odd
            //if tree is odd a new insertion will not change the rank for a 'new' median
            //ex. 8 nodes: new median rank = floor((8-1)/2) = 3
            //ex. 7 nodes: new median rank = floor((7-1)/2) = 3
            //As you can see if the tree size is odd before adding one new node the rank should not change
            //To check if you were inserting below or above the median node compare

            if inserting on right subtree:
               //rank of root (right node's parent) doesn't change since it's rank remains the same for
               //insertions in the right subtree
               if new_node.rank > M.median: //inserting after median
                  do nothing and return //median's rank is unaffected

               else: //inserting before median, implies median's rank was increased
                  M.median = M.median.parent //want to get the node with rank 1 less than M.median

            else: //inserting in the left subtree
               if new_node.rank < M.median: //implies M.median's rank was increased
                  M.median = M.median.left //need to get node with a rank 1 less than old M.median's rank
               else:
                  do nothing and return //if its above M.median this doesn't affect M.median's rank

        else:  //tree.size was even and is now odd
            //updated median's rank is old median's rank + 1

            if inserting on right subtree:
               if new_node.rank < M.median: //inserting after median
                  do nothing and return //median's rank was already increased no need for an update

               else: //inserting after median, implies median's rank was NOT increased
                  M.median = M.median.right //want to get the node with rank 1 less than M.median

            else: //inserting in the left subtree
               if new_node.rank > M.median: //implies M.median's rank was NOT increased
                  M.median = M.median.parent //need to get node with a rank 1 more than old M.median's rank
               else:
                  do nothing and return //if its below M.median then M.median's rank was increased, no need for update
```

```
MEDIAN(M):
   return M.median; //this is the pointer to the median node in the tree
```

The algorithm proof is in the comments and relies on the understanding of waht rank means and how standard AVL insertions and deletions operate
with rotations.

The run time of MEDIAN(M) is constant since it is a simple return statement so it runs on O(1).

Run time of INSERT(M, x) is O(log n). This is because the augmented insertion process is virtually unchanged compared to the standard AVL INSERTION.
The ranks of affected nodes during the rotation are unaffected since they are simply computed from left subtree or parent node sizes which are stored
in the node as an instance variable so there is no need for a complex algorithm to do this. The update for the new median after insertion is done by the big block of if and else statements, this runs in constant time. The augmentations are all constant so the worst case run time of INSERT(M, x) is O(log n).

Run time of EXTRACT-MED(M) is slightly a bit more complex. The function's relies on the standard AVL delete, so we begin with O(log n). However the augmentation of this method calls another function OS-SELECT(M, rank), this also runs in O(log n). This is called exactly once in the algorithm so the total worst case running time is O(2log n). However since we are only interested in asymptotic notation the worst case running time of EXTRACT-MED(M) is just O(log n).

These functions are very efficient since the augmentation allows MEDIAN(M) to run in constant time while also keeping the insertion and extraction at the same asymptotic worst case running time as standard AVL trees which are already incredibly efficient to begin with.

3.
Operations are highlighted in green and new codes and terminologies are highlighted in yellow.

In addition to S.root and S.NIL, dictionary S has a new field S.minTree that stores a new type of node. That node contains the differences as its keys and corresponding pairs as values.

Also, each node stores has references to its parent predecessor, successor(PPS), and the minimum difference(minDif) between the node and one of PPS. PPS in each node will be referenced as nodes are inserted, and updated when deleted, or affected during rebalancing and then S.minTree is another AVL tree storing the smallest difference among differences of a node and its parent, predecessor, and successor for each node. Nodes in minTree store the minimum difference for a node as keys and the corresponding nodes, called “thePair”. Nodes are considered equal if they have the same pair of nodes and difference. Equal nodes will not be inserted to save storage by half. The maximum number of nodes for this tree is the number of nodes in the original AVL tree.

Operations

Helper method updatePPS(node) to update the parameter’s fields

```
updatePPS(node)
    // store the given node’s predecessor and successor and determine the minimum difference and which node has the minimum difference.

    predecessor = TREE-PREDECESSOR(node)

    successor = TREE-SUCCESSOR(node)

    if ( predecessor != NIL OR successor != NIL OR node.parent != NIL )
        minDif = minimum of difPredecessor, difSuccessor, and difParent
        minDifNode = one of the PSC nodes that has the minimum difference
    else
        minDif = NIL
        minDifNode = NIL

    return minDifNode
```

updatePPS takes O(log(n)), since finding predecessor and successor takes O(log(n)).

AVL-INSERT-M is modified to store PPS for new nodes as they are created and inserted and update minPair if the differences are small enough at the end of the code. AVL-INSERT also manages updating S.minTree.
Note that unmodified AVL-INSERT from the lecture note is used when inserting into S.minTree

```
AVL-INSERT-M(S.root, x)
    if root is NIL:  # found insertion point
        root = TreeNode(x, root.parent)  # initial height = 0
        closest = updatePPS(root)
        // minTreeNode stores the closest difference(key) and two corresponding nodes
        node = minTreeNode(root.minDif, (root, root.minDifNode)

        if ( root.minDif != NIL && root.minDifNode != NIL)
            // node’s key is minDif
            AVL-INSERT(S.minTree, node)
    elif x.key < root.item.key:
        root.left.parent = root
        root.left = AVL-INSERT-M(root.left, x)
        if root.left.height > root.right.height + 1:
            root = AVL-REBALANCE-TO-THE-RIGHT(root)
        else:  # no rebalancing, but height might have changed
            AVL-UPDATE-HEIGHT(root)
    elif x.key > root.item.key:
        root.right.parent = root
        root.right = AVL-INSERT-M(root.right, x)
        if root.right.height > root.left.height + 1:
            root = AVL-REBALANCE-TO-THE-LEFT(root)
        else:  # no rebalancing, but height might have changed
            AVL-UPDATE-HEIGHT(root)
    else:  # x.key == root.item.key
        root.item = x       # Just replace root's item with x, nothing else change

    oldMinDifNode = root.minDifNode
    root.minDifNode = updatePPS(root)
    updateMinTree(root, oldMinDifNode)

    return root
```

AVL-INSERT-M takes O (log n) as it makes multiple log(n) operations from using updatePPS and performing insert/delete in S.minTree, each of which take log(n). Therefore, AVL-INSERT-M takes c*log(n) where c is the sum of updatePPS and updating S.minTree.

updateMinTree is a helpmer method to update S.minTree. The method deletes the node that has old information and inserts a new node reflecting the changes made by insert, delete, or rebalance.

```
updateMinTree(node, oldMinDifNode)
    // if node's minDifNode has changed, delete that node in the minTree and insert a node with the new minDifNode
    if oldMinNode != node.minDifNode
        AVL-REMOVE(S.minTree, | oldMinDifNode – node |)
        newNode = minTreeNode(node.minDif, (node, node.minDifNode)
        AVL-INSERT(S.minTree, newNode)
```

updateMinTree can take O(log n) as it involves inserting and removing from an AVL tree.

AVL-ROTATE-TO-THELEFT is modified to save changes in predecessor, successor and minDif, minDifNode of a node if the node is affected rebalancing.

```
AVL-ROTATE-TO-THE-LEFT(parent)
    // Rearrange References
    child = parent.right
    parent.right = child.left
    child.left = parent
// save old info
    oldParentMinDifNode = parent.minDifNode
    oldParentMinDif = parent.minDif
    oldChildMinDifPair = child.minDifNode
    oldChildMinDif = child.minDif
// update PPL for rebalanced nodes
    parent.minDifNode = updatePPS(parent)
    child.minDifNode  = updatePPS(child)
// reflect this information in the minTree
    // if parent’s minDifNode has changed, delete that node in the minTree and insert a node with the new minDifNode
    if oldParentMinNode != parent.minDifNode
        AVL-REMOVE(S.minTree, | oldParentMinDif – parent.minDif |)
        AVL-INSERT(S.minTree, parent)
    if oldChildMinNode != child.minDifNode
        AVL-REMOVE(S.minTree, | oldChildMinDif – child.minDif |  )
        AVL-INSERT(S.minTree, child)

    // Update Heights
    AVL-UPDATE-HEGIHT(child.left)
    AVL-UPDATE-HEIGHT(child)
    Return child
```

The same change can be made to AVL-ROTATE-TO-THE-RIGHT(parent

AVL-ROTATE-TO-THE-RIGHT/LEFT can take O(log n) as it involves inserting and removing from an AVL tree and performing updatePPS.

When removing a node, the node is removed from S.tree.
Another node that has the removed node as its minDifNode must be updated.
The node containing the removed node in the minTree must be removed and a new node containing that node and its new minDifNode should be inserted into the minTree.
Update the minDif for pred, succ, parent of the removed node
Node A that has Node B as its minDifNode does not imply Node B has Node A as minDifNode.
Note that inserting into S.minTree uses AVL-REMOVE from the lecture notes.

```
AVL-REMOVE-M(S.root, x)
    oldLeft = root.left
    oldRight = root.right
    if root is NIL:
            pass  # nothing to remove
    elif x.key < root.item.key:
        ..........
    elif x.key > root.item.key:
        .........
    else:  # x.key == root.item.key            // FOUND THE NODE TO REMOVE


        // update the minTree to remove the minTreenode containing the removed node
            AVL-REMOVE(S.minTree, | root.minDifNode – root |)

        // save reference to the removed node
        removed = root


        if root.left is NIL or root.right is NIL:
            if root.left is NIL:
                root = root.right
            else:
                root = root.left

        else:
        # Select whether to replace root.item with its predecessor or
        # its successor, depending on the heights of subtrees.
                if root.left.height > root.right.height:
                    root.item, root.left = AVL-REMOVE-MAX(root.left)
                else:
                    root.item, root.right = AVL-REMOVE-MIN(root.right)
                # Height might have changed.
                AVL-UPDATE-HEIGHT(root)
// update the fields for the successor / predecessor which replaced the removed node
                oldMinDifNode = root.minDifNode
                updatePPS(root)
                updateMinTree(root, oldMinDifNode)

// update the minDif and minDifNode of the removed node's minDifNode. Updating field
        updatePPS(removed.minDifNode)
// update the minTree for the change caused by updating the removed node's minDifNode
        updateMinTree(removed.minDifNode, removed)

    updatePPS(root)
// update if needed
    updateMinTree(root, oldLeft)
    updateMinTree(root, oldRight)
    return root
```

AVL-REMOVE-M takes O (log n) as it makes multiple log(n) operations from using updatePPS in and performing insert/delete in S.minTree, each of which take log(n). Therefore, AVL-DELETE-M takes c*log(n) where c is the sum of updatePPS and updating S.minTree.

CONTAINS(S, x)

```
    Return AVL-SEARCH(S.root, x.key) == x
```

CONTAINS take O(log n) as it involves traversing the AVL tree.

CLOSEST(S)

```
    If (S.size < 2)
        Return NIL
    Else
```

```
// find the left most node with minimum difference as its key and return the two pairs related to it
        Return TREE-MINIMUM(S.minTree.root).thePair
```

CLOSEST takes O(log n) since it travels to the left most node.

---

4. a) Elements should have a rule for ordering to store them at the right place in the trees.
Elements in a standard hash table don't have to hold this property, because hash function determines where elements are stored using keys in the hash table so the order doesn't matter.

b) The worst case would be assigning all elements into the same AVL tree. Therefore, SEARCH, INSERT, and DELETE would take $\theta(\log n)$.

c) If $n > \#\ of\ occupied\ slot$, it means that collisions must have occurred
$n - \#\ of\ occupied\ slot$ is the number of keys that must have collided with other keys.
Therefore, $n - \#\ of\ occupied\ slot$ is the number of collisions.

$$E\ [\ Collisions\ ] = n - E\ [\ Occupied\ ]$$

The probability of a slot being occupied is

$$\frac{1}{m}$$

The probability of a slot not occupied is

$$1 - \frac{1}{m}$$

The probability of a slot not occupied after inserting n elements

$$\left(1 - \frac{1}{m}\right)^n$$

The probability of a slot occupied after inserting elements

$$[1 - \left(1 - \frac{1}{m}\right)^n\ ]$$

The expected number of occupied slots for a hash table of size m after inserting n elements

$$m[1 - \left(1 - \frac{1}{m}\right)^n\ ]$$

$$E\ [\ Collisions\ ] = n - E\ [\ Occupied\ ] = n - \ k[1 - \left(1 - \frac{1}{m}\right)^n\ ]$$

d) These two methods of implementation are very similar in comparison for their use in a hash table. An AVL tree would be more beneficial and efficient when a slot has a lot of elements hashing to that location. As this number becomes greater, it becomes easier to find a certain element due to the AVL's balancing features. In a linked list (although more efficient when the number of elements is small) it will take a longer amount of time to traverse through each node, as there is only one link to each “next” node. If all the slots had a small number of elements, the efficiency of both the linked list and the AVL tree would be relatively the same. However, a linked list would be better to use in the case where random numbers are being put into the hash table. An AVL requires its elements to be sorted, which would take longer than the linked list in the sense that it could just add the new element at the head and the previous head the “next” node. Also, only comparable elements can be stored in AVL trees, making linked-list the only option for unsortable elements.