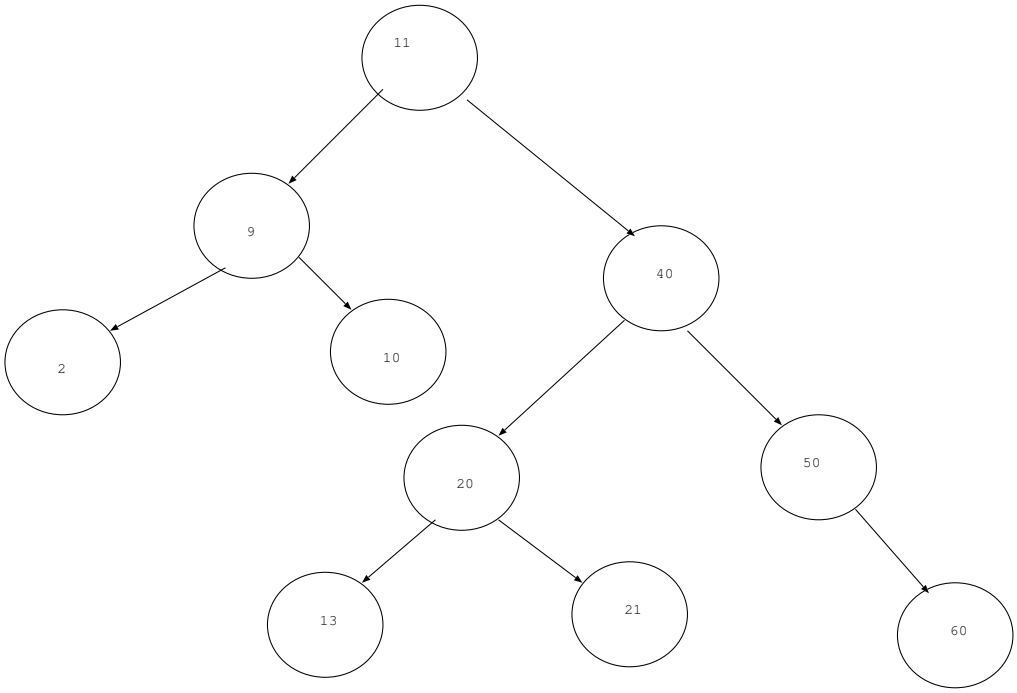
HW3

Shane Drafahl

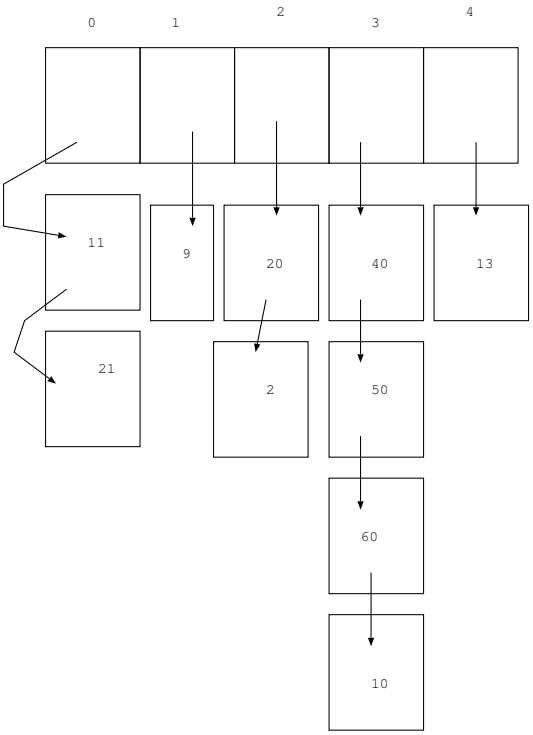
26 September,2017

1. (a)



(b). Every node in this tree follows the requirements to be an AVL tree. Every node has a difference of height for it children that is either -1,0,1.

(c). $(2x + 3) \bmod 5 \leq 4$ so we can assume the hashset only has a size of 4.



2. Consider that binary tree T is a perfectly balanced tree so each node must have 2 children or 0 children. The tree has $n = 2^{\ell} - 1$ distinct integers so the tree must have n nodes.

Lemma $n = 2^{h+1} - 1$ where h is the height of the binary tree.

Basis: Suppose a tree T' has only a single root node so h = 0. $1 = 2^1 - 1$.

Inductive Hypothesis: Suppose that $n = 2^{h+1} - 1$ is true for tree T_1, T_2.

Recursion:

Using structural induction for T_1, T_2 returns the number of node for each tree $n = 2^{h+1} - 1$ where h is the height for either tree. Both trees need to have the same height or else the new binary tree might not be perfectly balanced . If we combine T_1 and T_2 and for order it to be a perfectly balanced tree we will add a single node N that will be the new root node that is a parent with the roots from T_1 and T_2. The height of the new tree 1 + h the number of nodes. The number of nodes in the new tree will be $2^{h+1} - 1 + 2^{h+1} - 1 + 1$ or it can be reduced to $2^{h+2} - 2 + 1$ ... $n = 2^{h+2} - 1$. Since $1 + h = h^{'}$ the new tree will have $2^{h^{'}+1} - 1$ nodes. So therefore for all perfectly balanced trees there are $2^{h+1} - 1$ nodes for its height h. QED

$n = 2^{\ell} - 1$ is the number of nodes in the tree so therefore ℓ = h + 1 where h is the height of the tree.

The algorithm is

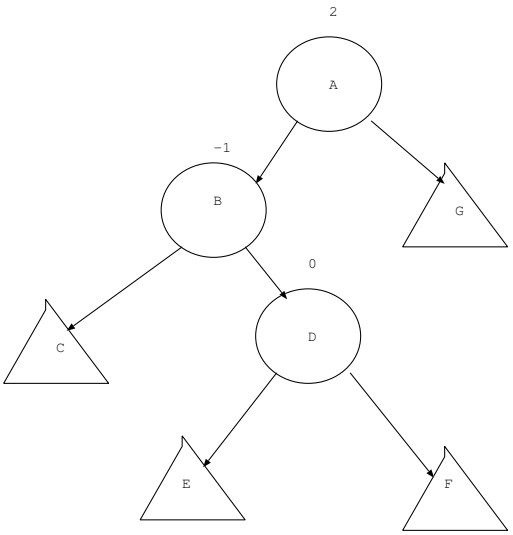
```
// T is a tree
// T.R is the root node
// T.R.L is the left child of the root
// T.R.R is the right child of the root

getElementSmallerThan(T) {
    return T.R.R
}
```

This algorithm is obviously O(1) this algorithm is correct because by the lemma there are $2^{h+1} - 1$ nodes in the tree and we want a value that is smaller than $2^{h-1} - 1$ nodes. If there are a nodes in the tree then we need

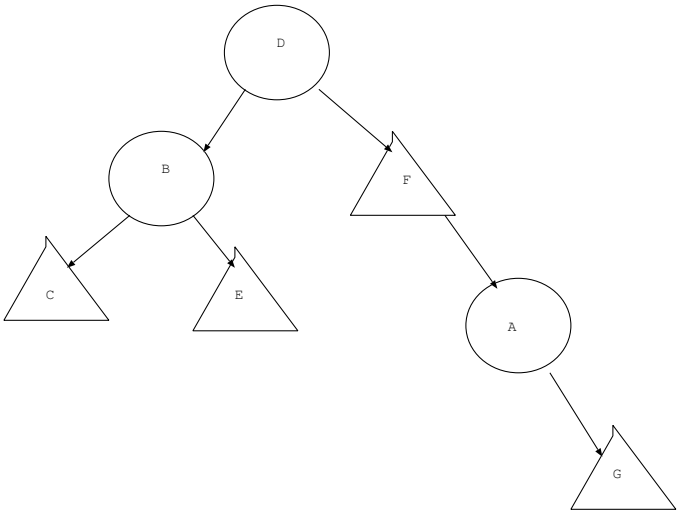
to find the node smaller than $\frac{a+1}{4} - 1$ nodes. Notice that $\frac{a+1}{4} - 1$ equals the number of nodes in the subtree of the right child of the right child of the root node. This makes sense because it should be about a fourth of the nodes it needs to be smaller than. For a tree of three nodes the right child has no children so $\frac{3+1}{4} - 1 = 0$ so in that case the right child of the root would be the largest value in the tree being smaller than 0 values in the tree. For larger trees almost a quarter of the entire tree is the subtree of the right child of the right child of the root. Meaning that the right child of the root is smaller than $2^{\ell-2} - 1$ or $n = 2^{h-1} - 1$ elements in the tree. QED

(b) We start with a tree that looks like

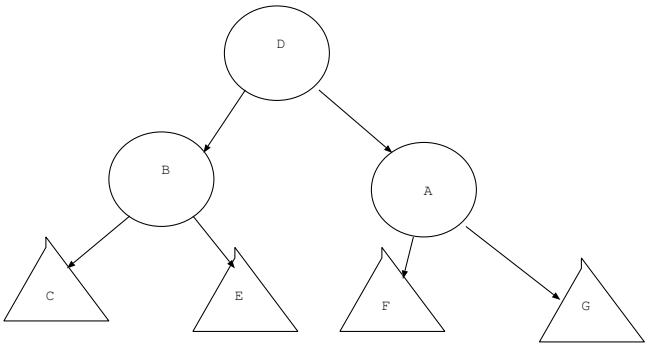


Suppose that tree C has a height of $h^{'}$, E and F has a height of h and G has a height of $h^{''}$. B has a balance factor of -1 so $(h^{'} + 1) - (h + 2) = -1$ so $h^{'} = h$. A has a balance factor of 2 so that means that $(h+3) - (h^{''} + 1) = 3$ so $h = h^{''}$.

So because the D is less than a and greater than B we will rotate the D.



This tree is still not balanced. So we will rotate a subtree to get.



The balance factor for D,B,A are 0 each. It is a perfectly balanced tree.

3. In order to do this I will create a tree of the right size. Since $2^h = l$ where the h is the height of the tree and l is the number of leaves .I just need to create a tree of that size and then insert the values into the leaf nodes. $2^{h+1} - 1$ also equals the number of nodes in the tree.

```
Node {
    value = 0,
    Node parent,
    Node leftChild,
    Node rightChild,
}

Tree {
    Node root,
    List leafNodes,
}

function createTree(List list) {
    Tree tree
    tree.root
    Node root
    root.parent = null
    tree.root = root
    height = log2(list.length)
    List leafNodes = [] // empty list that will hold nodes
    tree.leafNodes = leafNodes
    addLayerToTree(root, 0, height, leafNodes)
    addValuesToLeaves(list, leafNodes)
    return tree
}

function addLayerToTree(Node n, level, height,List leafs) {
    if(level != height) {
        Node left
        Node right
        n.leftChild = left
        n.rightChild = right
        addLayerToTree(n.leftChild, level + 1, height)
        addLayerToTree(n.rightChild, level + 1, height)
    } else {
        leafs.apend(n)
    }
}

function addValuesToLeaves(List values,List nodes) {
```

```
    for(int x=0;x<values.length;x++) {
        Node node = nodes.get(x)
        value = values.get(x)
        node.value = value
    }
}
```

This algorithm creates a binary tree to the correct height it would need to be for there to be at least a single leaf node node per value. Because of the order of recursion we know the list of nodes that is given to addValuesToLeaves() will be in order from left to right. To determine the runtime of the algorithm we first observe that the atomic lines dont matter so we will focus on the two functions. addValuesToLeaves() is $O(n)$ if n was equal to the number of items in the list. Now we will look at addLayerToTree, it is a recursive function so we must observe how many iterations it would be called in terms of n before it reaches the base case. The function goes untill the level != height. The height $log_2(n) = h$.The level increases by 1 every recursive call so this function is called $log_2(n) + 1$ times. So the runtime of the whole algorithm is $n + log_2(n) + 1 = O(n)$.

(b) For this problem we will use a binary search tree but then store the sum in its parent nodes. So we will modify the data structure in the first part. We need a function to assign the sum values to the non leaf nodes.

```
// new function
function assignSums(Node n) {
    if(n.parent !=null) {
        n.parent.value += n.value
        assignSums(n.parent)
    }
}

// modify this function
```

```
function createTree(List list) {
    Tree tree
    tree.root
    Node root
    root.parent = null
    tree.root = root
    height = log2(list.length)
    List leafNodes = [] // empty list that will hold nodes
    addLayerToTree(root, 0, height, leafNodes)
    addValuesToLeaves(list, leafNodes)
    for(int x=0;x<leafNodes.length;x++) {
        assignSums(leafNodes.get(x))
    }
    return tree
}

// one of the required functions
function increment(i , val) {
    Tree t // the tree the algorithm is applied to
    recursiveIncrament(t.leafNodes.get(i), val)
}

function recursiveIncrament(Node node, value) {
    node.value += value
    if(node.parent != null) {
        recursiveIncrament(node.parent, value)
    }
}

SS() {
    Tree t // the tree the algorithm is applied to
    return t.root.value
}
```

The datastructure is a tree structure that the leaf nodes hold the values and the shared parents are the sum of each of its child nodes. Similar to a fibonacci tree. The root node therefore holds the sum for all values in the tree. The time bounds for SS() = $O(1)$ and increment(i, val) adds a value

to the leaf node and then recursively travels up the tree untill it reaches the root of the tree. The height of the tree is the log of the number of leaf nodes so we know if n represents the number of leaf nodes that it would have take $log(n)$ steps to perform recursiveIncrament so therefore increment $= O(log(n))$.

4. This can be accomplished with a contigious unsorted array and a hashset with references to the inserted data input into D.To add we can just add the element to the end and if it runs out of space we will exponentialy increase the array size so that its unlikely that this will occure. We will do the same with the hashset as well. To add we will simply append a node to the end of the list and add a reference to the same node in the hashset. This would only take $O(1)$. When we search and delete we will use a quickselect algorithm to get the index of the node that is the kth largest element in the array. Quick select on average runs at $log(n)$ and when we get the value to delete it from the hashset it should only be $O(1)$. Once that is complete we just need to be able to search to see if somthing exists in the data structure. For this we will just need to search it via the hash so that is only $O(1)$.

```
D {
    occupied = 0
    size = 50
    List list = [size]
    List hash = [size]
}

Node {
    value,
    Node child,
}

function add(D structure, value) {
    occupied++
    if(occupied / size > .66) {
        newSize = structure.size^2
        List list = [newSize]
        List newHash = [newSize]
```

```
        for(int x=0;x<structure.size;x++) {
            Node n
            n = structure.list.get(x)
            hash = hashNumber(newSize, structure.list.get(x))
            list.append(n)
            if(newHash.get(hash)) {
                Node parent = newHash.get(hash)
                parent.child = n
            } else {
                newHash.set(hash, n)
            }
        }
        structure.size = newSize
        structure.list = list
    }
    Node n
    n.value = value
    structure.list.append(n)
    List hash = structure.hash
    hashNum = hashNumber(structure.size, value)
    Node searched = hash.get(hashNum)
    if(searched != null) {
        searched.child = n
    } else {
        hash.set(hashNum, n)
    }
}

function remove(D structure, k) {
    List list = structure.list
    lastIndex = structure.occupied - 1
    index = kthLargest(structure, 0, lastIndex, k)
    Node n = list.get(index)
    List hash = structure.hash
    hashN = hashNumber(structure.size, n.value)
    Node temp = hash.get(hashN)
    // removes it from the hashset
    while(true) {
        if(temp.value == n.value) {
```

```
            if(temp.child != null) {
                hash.set(hashN, temp)
            } else {
                hash.remove(hashN)
            }
            break;
        } else {
            if(temp.child != null) {
                temp = temp.child
            } else {
                return null
            }

        }
    }
    // remove it from the list
    list.remove(index)
}

function search(D structure, x) {
    List hash = structure.hash
    hash = hashNumber(structure.size, x)
    Node chain = hash.get(hash)
    while(true) {
        if(chain.value == x) {
            return true
        } else {
            chain = chain.child
            if(chain == null) {
                return false
            }
        }
    }
}

// quicksearch
function kthLargest(D structure, leftIndex, rightIndex, k) {

    List list = structure.list
    largest = structure.occupied - (k - 1)
```

```
    if (largest > 0 && largest <= rightIndex - l + 1) {

        pos = partition(structure, largest, list);

        if (pos-leftIndex == largest-1) {
            return pos;
        }

        if (pos-leftIndex > largest-1) {
            return kthSmallest(arr, l, pos-1, k);
        }

        return kthSmallest(arr, pos+1, r, largest-pos+l-1);
}
    return null
}

function partition(D structure, lastIndex, List list)
{
    x = arr[range],
    i = 0;
for (int j = i; j <= lastIndex - 1; j++){
if (arr[j] <= x) {
swap(&arr[i], &arr[j]);
i++;
}
}
    temp = list.get(i)
    list.set(i, list.get(lastIndex))
    list.set(lastIndex, temp)
return i;
}

function hashNumber(size,value) {
    A = (sqrt(5.0)-1.0)/2.0;
    return floor(size * (value * A - floor(value * A)));
}
```