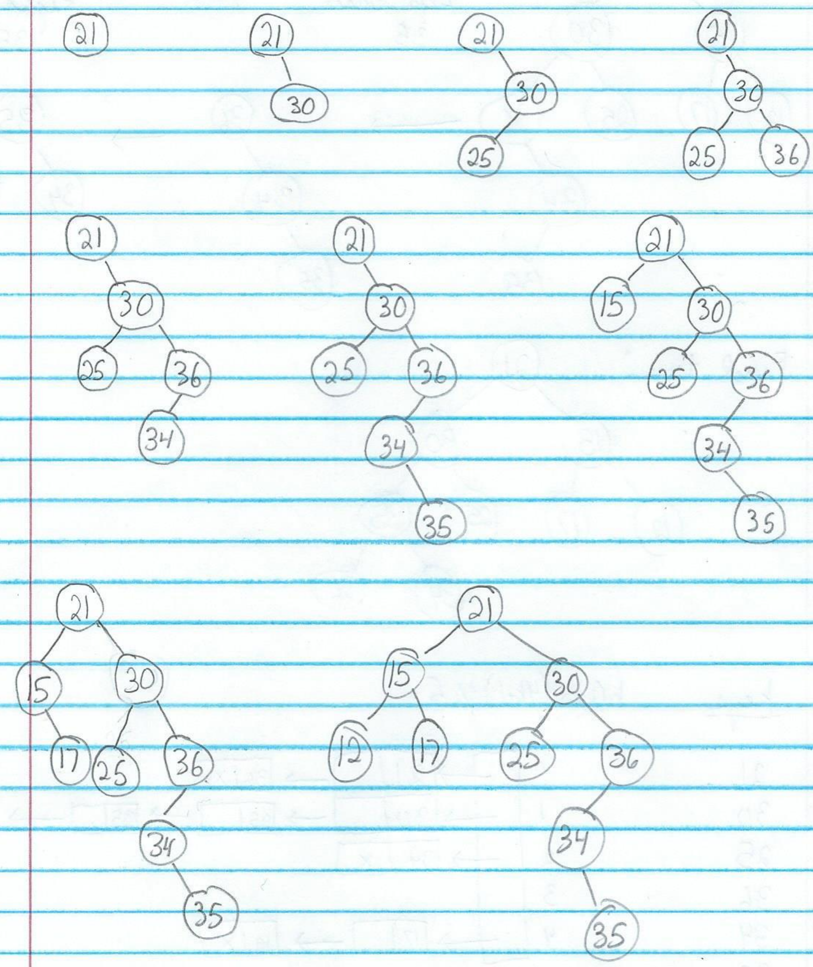
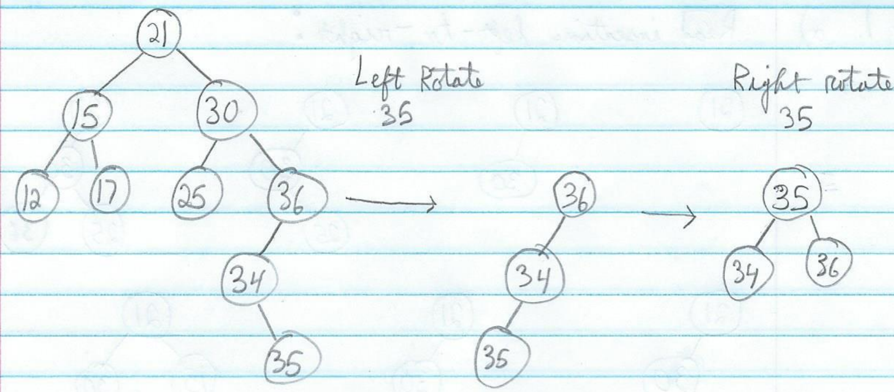
Thomas Haddy
499370896
Section 11
TA: Trent Muhr

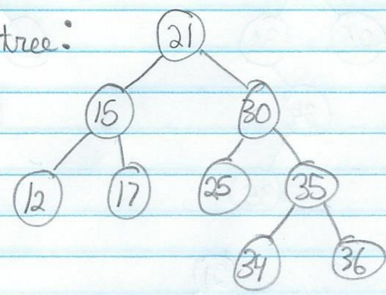
1. a) Read insertions left-to-right:



b)

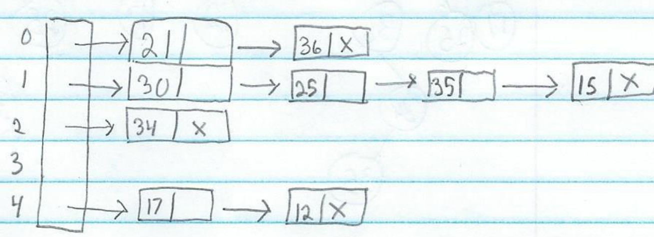


Final tree:

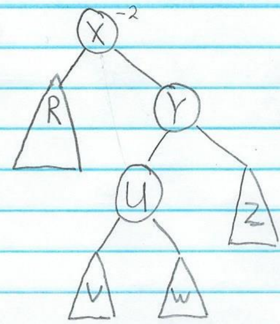


c) Keys $h(k) = (4k+1) \% 5$

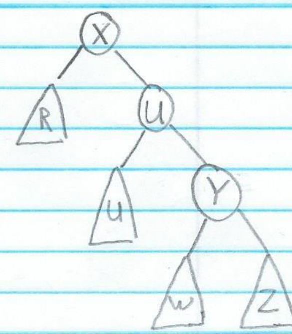
21
30
25
36
34
35
15
17
12



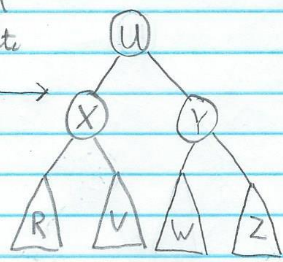
d) $B(U)=0$
$B(X)=-2$
$B(Y)=1$



Right rotate Y $\longrightarrow$



Left Rotate X $\longrightarrow$



$B(U)=0$
$B(X)=0$
$B(Y)=0$

Final Tree:

2.

So, this algorithm attempts to find an ‘a’ that is closest to k/2. Then it attempts to find a ‘b’ such that it is the closest value to k/2, which will always be either the predecessor or successor of ‘a’. Thus the algorithm has found an ‘a’, ‘b’ such that distance(a + b - k) is as close to 0 as possible.

```
Node closest = root;

distance(int a, int b):
        return absolute value of (a – b)

search(Node curr, int x):
        if (node is null)
                return
        if (distance(closest.val, x) < distance(curr.val, x))
                closest = curr
        search(curr.left, x)
        search(curr.right, x)

leftmostRightChild(Node curr, int x):
        if (curr has no children)
                return curr
        if (curr.right is null)
                return leftmostRightChild( curr.left, x)
        return leftmostRightChild( curr.right, x)

rightmostLeftChild(Node curr, int x):
        if (curr has no children)
                return curr
        if (curr.left is null)
                return leftmostRightChild(curr.right, x)
        return leftmostRightChild(curr.left, x)

leftmostRightParent(Node curr, int x):
        if (curr.val is greater than x)
                return curr
        return leftmostRightParent(curr.parent, x)

rightmostLeftParent(Node curr, int x):
        if (curr.value is less than x)
                return curr
        return rightmostLeftParent(curr.parent, x)

predecessor(Node curr):
        if (curr.left is null)
                return rightMostLeftParent(curr, curr.value)
        return rightmostLeftChild(curr, curr.val)
```

```
successor(Node curr):
	if (curr.right is null)
		return leftMostRightParent(curr, curr.val)
	return leftMostRightChild(curr, curr.val)

findMinDistance(Node root, int k):
	search(root, k/2)
	Node a = closest
	if (distance(closest.val, predecessor(a).val) > distance(closest.val, successor(a).val))
		Node b = predecessor(a)
	else
		Node b = successor(a)
	return a, b
```

The worst case of search if O(n) because if k is the right-most node, it must traverse the whole tree before finding it.

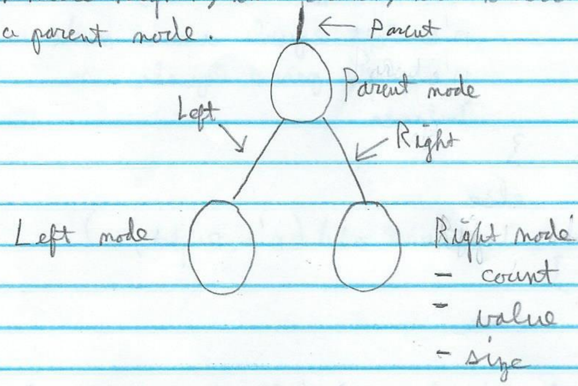
The worst case runtime of predecessor() is O(n) because it will either call rightMostLeftParent() or rightMostLeftChild() which in either case is O(n) because in the worst case it traverses the whole tree.

The worst case runtime of successor() is O(n) because it will either call leftMostRightParent() or leftMostRightChild() which in either case is O(n) because in the worst case it traverses the whole tree.

The worst case of findMinDistance() is O(n) because search is O(n) and it will either call successor() or predecessor() which are both O(n) so the entire algorithm has a worst case of O(n)

This algorithm is correct because the distance between a+b - k will become smaller as a and b approach k/2. The minimum distance possible for any a+b and k is a,b=k/2 since k/2+k/2=k and a, b will be as close as they can to k/2. This algorithm searches for the closest a to k/2 and then finds a’s successor or predecessor b closest to k/2. Therefore, the a, b nodes will be the two closest nodes to the minimum distance.

3. My data structure D will be a BST with a node left, node right, int count, int value, int size, and a parent node.




```
add(Node n, int x) {

if (n value equals x)
    n count ++
    return
}
if (n value is less than x)
    if(n's right child is null)
        n's right child equals new node with val x
        n's right's parent equals n
        balance
    }
    else
        return add(n's right, x)
}
```

```
if (n's value is greater than x)
    if (n's left child is null)
        n's left equals a new node with val x
        n's left's parent equals n
        balance
    }
    else
        return add(n's right, x)
}
}
```

Analysis: For a balanced BST, the height = log n, where n is the size of the tree, So the add(x) percolates down to either add a new node or increase the count. So, at the worst case, it will take O(log n) time to add(x) because it percolates down to the height which is log n.

```
frequency(intx, node curr) {

    if (curr is null)
        return 0
    if (curr.val equals x)
        return curr.count
    if (curr.val is less than x)
        return frequency(curr.right, x)
    if (curr.val is greater than x)
        return frequency(curr.left, x)
}
```

Analysis: At the worst case, this function percolates down the height of the tree log n where n is the number of nodes in the tree. So the worst case is O(log n)

```
search(int x, node curr) {

    if (curr is null)
        return false
    if (curr.val is less than x)
        return search(curr.right, x)
    if (curr.vall is greater than x)
        return search(curr.left, x)
    return true
}
```

Analysis: At the worst case, search(x, curr) does not find a node and percolates down the height log n of the tree. So, it is O(log n)

```
order(int y, node root)
    node found= lookFor(y, root)
    if(found. val is greater than y)
        return sum(found, 1)
    return sum(found, 0)
}
lookFor(int y, node curr) {
    if(curr's right and left are null)
        return curr
    if(y is less than curr. val and curr has no left)
        return curr
    if(y is greater than curr. val and curr has no right)
        return curr
    if(y equals curr. val)
        return curr
    if(y less than curr. val)
        lookFor(y, curr. left)
    if(y greater than curr. val)
        lookFor(y, curr. right)
}
sum(node curr, int order)
    if(curr. parent. val is less than curr. val)
        return sum(curr. parent,    order)
    if(curr. parent is not null)
        return sum(curr. parent, order + 1 + curr. parent. right. size)
    return order
}
```

Analysis: Order uses 2 helper methods to achieve its goal, outputting the number of nodes greater than y. LookFor(y, curr) looks for a node by percolating down and finds the closest node value of y: so y or closest value > y. The second helper method sum(curr, order) percolates up the tree to the correct node and outputs its right subtrees size. Both methods are recursive.

lookFor(y, curr) runtime: at worst case, it goes down to the height of the tree: $h = \log n$ where n is number of nodes. So it's $O(\log n)$

sum(curr, order) runtime: at worst case, it goes up the height of tree: $h = \log n$. So it's $O(\log n)$

order(y, root) runtime: at worst case, it percolates down the whole height $\log n$ and percolates back up the tree's height $\log n$. So the worst case is $O(2\log n) =$

$$O(\log n)$$

4.

```
traverse (Node curr) {
    if curr is not null
        traverse (curr.left)
        if (curr.left and curr.right equals null)
            print leaf and rank
        rank++
        traverse (curr.right)
    }
}
```

Rank is a global variable initialized to 1.

Worst case Runtime: This algorithm in the worst case will go through every node in the tree of size n. So the worst case runtime is $O(n)$.

Proof: Prove the correctness of traverse i.e. $\forall n \in T$ where n is a node in BST T, all leaves get printed.

Base: n is a root. curr is not null. Then it prints n and its rank and exits ✓

I.H.: Assume curr is a leaf. Prove curr+1 is also a leaf and will be printed with its rank.

I.S.: By I.H., curr is a leaf on T. This leaf could be a left child, right child, or only a parent.

Case 1: leaf is left child.

So this leaf gets printed. Traverse(right) will be called and the curr+1 will be printed next as a leaf as long as its children are null. If the children are not null, it goes back up the call stack and traverses down the subtree's left most child until both children are null and updates rank. Then it prints curr+1 if it exists. If not, exits. ✓

Case 2: leaf is a right child.

So this leaf gets printed. It goes back up the call stack and traverses down the subtree's left most child until both children are null and updates rank. Then it prints curr+1 if it exists. If not, exits ✓

Case 3: leaf is only child.

This leaf gets printed. It goes back up the call stack and traverses down the subtree's left most child until both children are null and updates rank. Then it prints curr+1 if it exists. If not, program ends. ✓

Therefore since all of the cases hold true, the algorithm is correct and runs as specified.