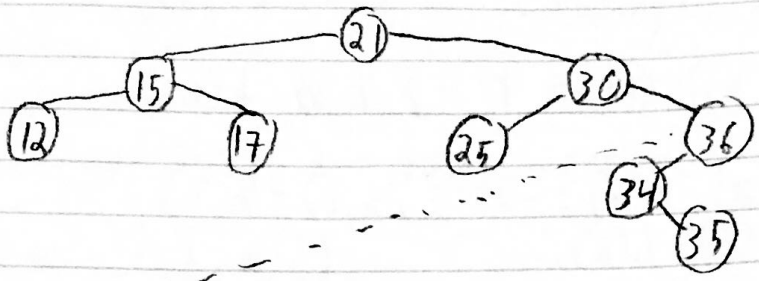
HW #3

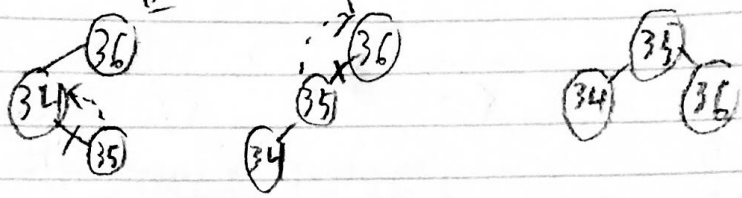
Recitation 5, 1-2 pm, TA: Marios Tsekitsidis

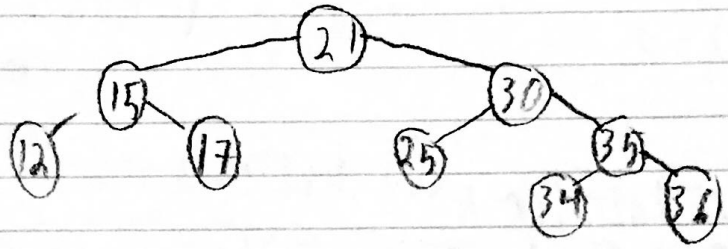
Christian Shinkle

1. a.

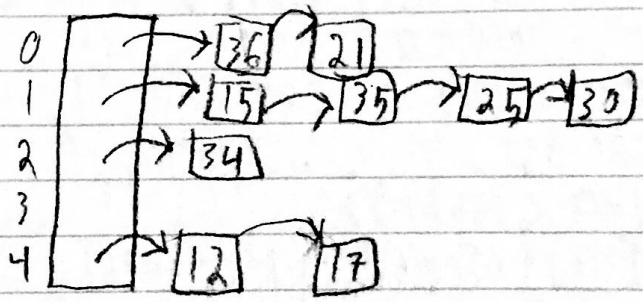


b.

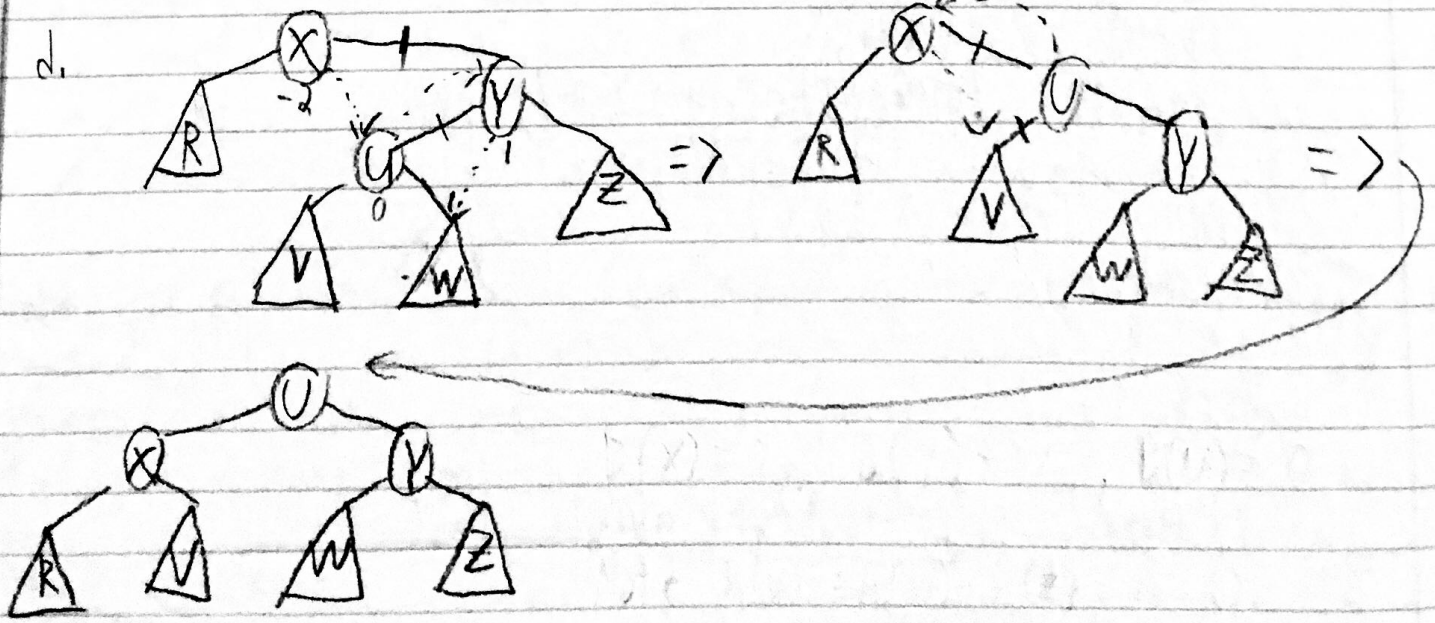




c. $(4 \cdot x + 1) \% 5$



d.



$h(A)$ = height of A

Before rotation: $B(u) = 0 \Rightarrow h'(v) = h(v)$

$B(y) = 1 \Rightarrow (1 + h(v)) - h(z) = 1 \Rightarrow h(v) = h(z)$

$B(x) = -2 \Rightarrow (2 + h(z)) - h(R) = -2 \Rightarrow h(z) = h(R)$

So, $h(v) = h(w) = h(z) = h(R)$.

After rotation:

$B(x) = h(R) - h(v) = 0$
$B(y) = h(w) - h(z) = 0$
$B(u) = (h(R)+1) - (h(w)+1) = 0$

2.

```
Input: BST T, Integer k
Array a = an array filled with an Inorder traversal of T;
bestLeft = left = 0;
bestRight = right = size of a - 1;
result = ∞;
while (left < right){
  if (|a[left] + a[right] - k| ≤ result) {
      bestLeft = left;
      bestRight = right;
      result = |a[left] + a[right] - k|;
  }
  if (a[left] + a[right] > k)
    right--;
  else
    left++;
  if (result == 0)
    return bestLeft, bestRight;
}
return bestLeft, bestRight;
```

(2 cont.)
For filling the Array a, use the following algorithm:

```
InorderHelper:
Input: BST T
Dynamically-sized Array a = empty array;
i=0;
InorderRec(T.root, a, i);
return a;

InorderRec:
Input: Node n, Array a, Integer i
if (n == null)
    return;
else {
    InorderRec(n.left, a, i);
    a[i] = n.data;
    i++;
    InorderRec(n.right, a, i);
}
```

Proof of correction by induction:
In order to prove correctness, we need to prove these two invariants hold: At the start of the k^{th} iteration,

- left < right
- If there exists indices i, j s.t. $|a[i]+a[j]-k| < result$, then $left \le i < j \le right$.

Note: the loop ends when left ≥ right. So left < right always holds.
Base case: let n = length of Array a. Then left = 0 and right = n-1. Note, for any values of a[0] and a[n-1], $|a[0]+a[n-1]-k| < \infty$. So result will equal this new value. Thus, if i, j exist s.t. $|a[i]+a[j]-k| < result$, then $left \le i < j \le right$.

(2 cont.)

Inductive Hypothesis: At the start of the m^{th} iterations,
- left < right
- If there exists i, j s.t. $|a[i] + a[j] - k| <$ result, then $left \le i < j \le right$.

Inductive Case: If result == 0 after m^{th} iteration, then the algorithm will terminate and return. Otherwise, let $left_m$ and $right_m$ be the value before the m^{th} iteration. There are two case: $a[left] + a[right] > k$ or $a[left] + a[right] < k$

Case 1: $a[left] + a[right] < k$.
Suppose that there exists $i < j$ s.t. $|a[i] + a[j] - k| <$ result.
By I.H, we have $left_m \le i < j \le right_m$. Since $a[left] + a[right] < k$, we argue $left_m \neq i$. Suppose that $left_m = i$. Then the max value of $a[i] + a[j]$ is at most $a[left_m] + a[right_m]$ since the array is sorted. However, we know $a[left_m] + a[right_m] < k$. Thus $left_m \neq i$. Using the I.H., we have $left_m < i$. In this case, left becomes left+1 and right remains unchanged. Thus, $left_{m+1} = left_m + 1$ and $right_{k+1} = right_k$. Since $left_k < i$, we conclude that $left_{k+1} = left_k + 1 \le i$. Thus, we have $left_{k+1} \le i < j \le right_{k+1}$.

Case 2: $a[left] + a[right] > k$
Suppose that there exsits $i < j$ s.t. $|a[i] + a[j] - k| <$ result.
By I.H., we have $left_m \le i < j \le right_m$. Since $a[left] + a[right] < k$, we argue $right_m \neq j$. Suppose that $right_m = j$. Then the min value of $a[i] + a[j]$ is at most $a[left_m] + a[right_m]$ since the array is sorted. However, we know $a[left_m] + a[right_m] > k$. Thus, $right_m \neq j$. Combining this with I.H, we have $right_m > j$. In this case right becomes right - 1 and left remains unchanged. Thus $right_{m+1} = right_m - 1$ and $left_{m+1} = left_m$. Since $right_k > j$, we conclude that $right_{k+1} = right_{k+1} = right_k - 1 \ge j$. Thus, we have $left_{m+1} \le i < j \le right_{m+1}$.

(2 cont.)
For the worst case runtime, we first analyze the runtime of Inorder traversal to build Array a. In the worst case, Inorder traversal will need to recursive find each node and add it's data to a dynamically-sized array. Adding to the array is a constant c_1 and the number of nodes = n. So, $\sum_{i=1}^{n} c_1 = c_1 n \in O(n)$.

For the while loop, the size of the array = number of nodes = n. In the worst case, every element of the array will need to be checked. For example, a[n-2] and a[n-1] may be the values we are searching for. Inside the loop, in this case, the 3 statements and three conditional checks will always be ran and this can be a constant c_2. So, $\sum_{i=1}^{n-1} c_2 = c_2(n-1) \in O(n)$.

This means our total runtime will be $O(n) + O(n) + c_3$, where c_3 is the constant time for intializing variables before the loop. Therefore, $O(n) + O(n) + c_3 \in O(n)$

3. The data structure I implemented is an augmented AVL BST with two fields added to the nodes: count, to track if multiple elements are in the BST and n_size, to track how many children are in the right subtree of that node. It is assumed that the root of the BST is a field of BST.

(3 cont.)

```
frequency(x);
 Input: x
 n = root;
 while (n is not null ) {
    if (n.data == x)
        return n.count;
    if (x > n.data) {
       n = n.right;
    }
    else {
       n = n.left;
    }
}
return 0;
```

Frequency

For worst case run time, note that this is an AVL BST so the height of the tree = $\log_2 n$, where n is the total number of nodes in the tree. The worst case for the algorithm is to check the height number of nodes and not find x. the content of the loop runs in constant time, so frequency $\in O(\log n)$.

(3 cont.)

```
Search (x)
Input: x
 n = root;
 while (n is not null) {
   if (n.data == x)
      return true;
   if (n.data < x)
      n = n.right;
   else
      n = n.left;
}
return false;
```

Search

for worst case run time, note that this is an AVL BST so the height of the tree is $= \log_2 n$, where n is the total number of nodes in the tree. The worst case for the algorithm is if x is not in the tree, so the loop runs height-number of times, the content of the loop runs in constant time, so Search $\in O(\log n)$

(3 cont.)

```
order (x)
Input: x
  n = root;
  while (true) {
     if (n.data == x)
        break;
     if (x > n.data) {
        if (n.right isn't null)
           n = n.right;
        else
           break;
     }
     else {
        if (n.left isn't null)
          n = n.left;
        else
          break;
     }
}
count = n.r_size;
while (n has a parent)
   if (n.par is left parent)
      count += (n.par.r_size + 1);
   n = n.par;
}


return count;
```

```
add (x)
  Input: x
  cur = root
  parent = null
  while (cur isn't null) {
      parent = cur;
      if (cur.data == x) {
          cur.count++;
          incParentsSubtrees (cur);
          terminate the algorithm;
      }
      if ( x > cur.data)
          cur = cur.right;
      else
          cur = cur.left;
  }
  create new node y with: {
      par = parent;
      data = x;
      r.size = 0;    left = null;
      count = 1;     right = null;
  }
  if (x < parent.data)
      parent.left = y;
  else
      parent.right = y;
  incParentsSubtrees(y);
  balance ();
```

```
incParentsSubtrees(n)

  Input: n
  while (n isn't null){
    if(n.par is a left parent)
        n.par.r.size++;
    n = n.par
  }
  terminate
```

(3 cont.) Add

For the worst case runtime, let h = max height of the tree and n = total number of nodes. The worst case will be when you traverse the tree to a leaf with max height. Then, when incParentSubtree is called, it will also have to traverse the max height of the tree in reverse. This means add's runtime = $\log(c_1 n) + \log(c_2 n) + c_3$ where c_1, c_2, c_3 are constant number of ops in side both the loops and the ops outside both loops. Therefore, add $(x) \in \log(n)$. Note $h = \log_2 n$ because the tree is an AVL BST.

Order

For the worst case, let h = max height of the tree and n = total number of nodes. The worst case will be when you traverse the tree to the right-most leaf, which is equal to h operations. Then, because the leaf is right-most, every ancestor of the leaf is a right ancestor. Therefore, the first while loop will run h times, the second loop will run h times as well, and C_1 constant operations will run in between the loops. Let C_2 and C_3 represent the constant operation in the first and second loop respectively. Therefore, Order = $C_1 + \log(c_2 n) + \log(c_3 n) \in O(\log n)$ Note $h = \log_2 n$ because the tree is an AVL BST.

4. rank (x)
Input; BST x; rankHelper (x.root, 0, false);

```
rankHelper (n, i, b)
 Input: Node n, Integer i, Boolean b

 if (n is a leaf){
    print (i+1);
    return i;
 }
 else {
  j = i
  if (n has left child)
     j += rankHelper (n.left, j, b);
  j ++j;

  if (n has right child)
     j += rankHelper (n.right , j, true );

  if (b)
     j -= i;

  return j;
}
```

Proof of correctness:
Note that rankHelper is just an inorder traversal that tracks the running total of nodes visited. The Boolean b keeps track whether nodes have been double-counted or not. If they have been, subtracting i from j will remove the double-counted nodes.

For the worst case runtime, every node will be visited at least once since it is an inorder traversal. The worst case will be if the tree is perfectly balanced because all internal nodes will be visited 3 times and all leaves will be visited only once. This means rankHelper $= c_1(\frac{n+1}{2}) + 3 \cdot c_2 \cdot (\frac{n-1}{2}) \in O(n)$, where c_1, c_2 are constants.