Homework 3
Com S 311
Due: Feb 16, 11:59PM

Late Submissions are NOT accepted. Solutions will be posted Feb 17.

There are 4 problems and each problem is worth 60 points. When asked to design and algorithm, please write *pseudo code*. If you write Java/C code, it will not be graded.

1. Let A be an array consisting of elements in the following order

$$21, 30, 25, 36, 34, 35, 15, 17, 12$$

   (a) Draw the binary search tree that is obtained by adding/inserting the elements (without balancing) in the order they appear in the array.

   (b) If the resulting tree is not balanced, perform balancing as per AVL tree discussed in lectures. Specify the steps needed to balance (left rotation/right rotation etc) and draw the resulting balanced tree.

   (c) Draw the hash table (using chain hashing) obtained by by adding the above elements using the hash function $(4x + 1)\%5$.

   (d) For a BST T and a node N in T, let $B(N)$ = height of left sub tree of N - height of right subtree of N. Note that $B(N)$ could be positive, negative or 0. Let T be the following BST: The root node is X, it has a left subtree named R and a right subtree rooted at Y. The node Y has a right subtree Z and a left subtree rooted at node U. The node U has two sub trees V and W. Suppose that $B(U) = 0$, $B(Y) = 1$ and $B(X) = -2$. Draw the tree obtained by balancing T. Draw the intermediate trees (if any) obtained while balancing. Once you balance the tree, what are $B(X), B(Y)$ and $B(U)$?

2. Suppose that you are given Binary Search Tree T and and integer k. Assume that nodes of T hold distinct integers. Give two integers a and b ($a \neq b$) in T let

$$Distance_{a,b} = |a + b - k|$$

   Give an algorithm that gets T and k as inputs and finds a pair $a, b \in T$ such that

$$Distance_{a,b} = \min\{Distance_{u,v} \mid u \in T, v \in T, u \neq v\}.$$

   Analyze the worst-case time complexity of your algorithm and justify the correctness.

   Ans. We need to find a pair of integers a and b in the Trees whose sum is closest to k. Basic Idea: Perform an in-order traversal and obtain a sorted array. Then this problem is similar to HW1, Problem 4. Below are the details. Consider the following algorithm:

- Input: Tree T, and an integer k.
- Perform in-order traversal of T and store the elements in a sorted array A (of length n). Assume
- $left = 0$, $right = n - 1$, $dist = \infty$. $index1, index2 = -1$
- while (left ¡ right) DO
  - $d = |A[left] + a[right] - k|$
  - if $d < diff$ then $diff = d$; $index1 = left$, $index2 = right$;
  - if $A[left] + a[right] < k$ increment $left$, else decrement $right$;
- Output

Recall that the time taken by the in-order traversal is $O(n)$. Note that every iteration of the loop takes $O(1)$ time. Initially the $right - left = n - 1$. When $left$ equals $right$ (equivalently the difference between $left$ and $right$ is zero) the algorithm stops. Note that during every iteration either $left$ is increased or $right$ is decreased. Thus after every iteration, the absolute value of difference between $left$ and $right$ is decremented by one. Thus after $O(n)$ iterations, the algorithm stops. Thus the total time taken by the above algorithm is $O(n)$.

We will now prove the correctness of the algorithm. We know by the in-order traversal property that A is a sorted array. Let x and y ($x < y$) be two indices in A whose sum is closest to k. (There might be multiple such indices, in such case fix one solution where x the smallest index.). We will show that there exists an iteration during which $left = x$ and $right = y$. If this happens, the value of $diff$ will be the $|a[x] + a[y] - k|$, $index1 = left$, $index2 = right$. After this iteration happens, note that the value of $diff$ never changes (and so $index1$ and $index2$ will also never change.

We consider two cases. Case 1: $a[x] + a[y] > k$; Say $a[x] + a[y] = k + T$ for some positive integers T. This means that T is the smallest difference between k and sum of any pairs of elements of the array. Initially the value of $left = 0$ and the value of $right = n - 1$. Since during every iteration, either $left$ is incremented or $right$ is decremented. One of the following will happen.

Case A: $left$ becomes x before $right$ becomes y. This implies that $right > y$. Since $A[x] + A[y] > k$, and $right > y$, it holds that $A[x] + A[right] > k$. Since $left = x$, we have $A[left] + A[right] > k$. This means that $right$ is decremented and $left$ does not change. This the value of $right$ must keep on decrementing till it reaches y. Thus there is an iteration at which $left$ is x and $right$ is y.

Case 2: $right$ becomes y before $left$ becomes x. In this case $left < x$. We will argue that $right$ will not be decremented. For the sake of contradiction, assume that $right$ will be decremented. This means that the following must hold true: $a[left] + a[right] > k$; since $right = y$, this means that $A[left] + A[y] > k$. Note that $A[left] < A[x]$ (Since A is sorted). Thus $A[left] + A[y] < A[x] + A[y]$. Combining this with $A[left] + A[y] > k$, we have the following

$$k < A[left] + A[y] < A[x] + A[y]$$

This implies that $A[left] + A[y]$ is closer to k than $A[x] + A[y]$, and this is a contradiction. We

The proof for the case $a[x] + a[y] < k$ follows by symmetric arguments.

3. Design a data structure D that can store integers. The data structure should be ale to store a multi-set, i.e. an element can appear multiple times. The data structure should support the following operations in $O(\log n)$ time, where n is the number of distinct integers stored in D.

   - `add(x)`: Adds/inserts integer x into D. Even if x belongs to D, x should still be added.
   - `frequency(x)`: Number of times x appears in D.
   - `search(x)`: Returns true if x is in D.
   - `order(y)`: Number of elements in D that are larger than y. The element y may or may not be in D.

   If you are using BST and would like to balance the tree, you may simply write `balance` and assume that this operation will take $O(\log n)$ time. Analyze the worst-case run time of the operations.

   Ans. We will use an Augmented BST. Each node of the BST will contain the following the information: An integer x, number of times x appears in Tree, number of elements stored in the right subtree rooted at x. For a node N $N.data$ refers to the integer stored in N, $N.freq$ refers to the number of times the integer in node is appearing in the tree; and $N.rightCount$ refers to the number of elements in the right subtree of N. We initially start with empty tree.

   The method $add(x)$ works as follows:

   (a) $Current = root$;

   (b) Repeat

      i. if ($current.data == x$) $current.freq++$; and QUIT.

      ii. If ($current.data < x$), then

         - $current.rightCount++$;
         - If $current$ has no right child, create a node $\langle x, 1, 0\rangle$ and make it right child of $current$ and QUIT.
         - If $current$ has a right child, then $current = current.right$.

      iii. if ($current.data > x$) then

         - If $current$ has no left child, create a node $\langle x, 1, 0\rangle$ and make it left child of $current$ and QUIT.
         - If $current$ has a left child, then $current = current.left$

      iv. Balance

   We will first argue that the time taken is $O(h+logn)$ where h is the height of the tree. During each iteration, we either $QUIT$ the loop and or go one level down the tree. We will definitely quit the loop when we reach a leaf node. Thus the maximum number of times the loop can be performed is $O(h)$. Each iteration of the loop takes $O(1)$ and Balance takes $O(\log n)$ time. Thus the total time is $O(h + \log n)$.

   We will prove that the BST tree constructed will have the following properties:

   (a) For every Node N, $N.freq$ equals number of times $N.data$ is added to the tree.

(b) For every node N, $N.rightCount$ equals, the total number of elements that are added to the right subtree of N.

We will first prove by induction. Initially at the beginning, the tree is empty and so both the statements hold vacuously. Suppose the statements holds before the $add(x)$ operation. We will show that both the statements hold after the $add(x)$ operation. Suppose x has been added to the tree ℓ times, and let N be the node that contains x. By induction hypothesis, $N.freq = \ell$. During some iteration of *add*, the value of *current* becomes N and now we increment $N.freq$, and thus $N.freq$ equals $\ell + 1$. On the other hand, if x never has been added to the tree, then the above method created and adds a new node $\langle x, 1, 0$ to the tree. Note that $.freq$ of no other node has been changed. Thus Statement 1 holds, after the add operation. If $current.data < x$, then we are adding x to the right of the node *current*, and correctly $current.rightCount$ is incremented. If x is a leaf node in the tree, then it does not have a right subtree. This implies that Statement 2 also hold after the add operation.

`search(x)` works exactly as in search of BST.

(a) $Current = root$;

(b) Repeat

i. if ($current.data == x$) return True

ii. If ($current.data < x$), then

- If *current* has no right child, return False;
- If *current* has a right child, then $current = current.right$.

iii. if ($current.data > x$) then

- If *current* has no left child, return False;
- If *current* has a left child, then $current = current.left$.

and takes $O(h)$ time. Note that if x does not appear in the tree, this method never returns true. Suppose x appears in the tree. We will show that this method return True; We establish by showing that x is always in the subtree rooted at *current*. Initially this statement holds, as *current* equals *root*. Suppose that the statement holds at the start of ℓth iteration; I.e, x is in the subtree rooted at *current*. Consider the iteration. If $current.data$ equals x then the program correctly quits. If $current.data < x$, then it must be the case that x is in the right subtree of *current*. Note that in this case *current* indeed becomes $current.right$. If $current.data > x$, then x must be in the left sub tree of of *current*, and in this case *current* becomes $current.left$. Thus after the ℓth (and thus at the start of $(\ell + 1)$st iteration) x is in the subtree rooted at *current*.

`frequency(x)` works exactly as search; The time taken is $O(h)$. When if reach a node *current* for which $current.data = x$, we return $current.freq$; The correctness follows because of the property of BST that we established earlier.

`order(x).` Let us assume that x is the tree, and solve this first; Latter we will modify this.

(a) $current = root$

(b) $count = 0$

(c) Repeat

i. if ($current.data == x$) return $count + current.rightCount$;

ii. If ($current.data < x$), then $current = current.right$.

iii. if ($current.data > x$) then

- $count = current + current.rightCount + 1$;
- $current = current.left$.

Note that the time take is $O(h)$ as during each iteration, we go one level down the tree. We will establish the correctness by showing the following claim: At the start of every iteration consider the value of *count*. This equals the following quantity: Number of integers that are bigger than x and not in the tree rooted at *current*. This statement hols at the start of the first iteration, as at this time *current* equals *root*, *count* equals 0 and every number is in the subtree rooted at *current* (which is *root*.) Suppose that the statement holds at the start of ℓth iteration. Thus *count* equals the number of nodes in the tree that are bigger than x and not in the tree rooted at *current*. If $current.data == x$, now the additional nodes in the tree that are bigger than x are precisely all the nodes in the right subtree of *current*, and thus *count* correctly becomes $count + current.rightCount$. If $current.data < x$, then neither *current* not any node in $current.left$ are bigger than x. Now *current* becomes $current.right$. Thus *count* value still reflects all nodes in the tree that are bigger than x and not in subtree rooted at *current*. If $current.data > x$, then every node in $current.right$ and *current* are bigger than x. Since now *current* becomes $current.left$, the value of *count* is correctly incremented by $current.rightCont + 1$.

We can deal with the case x is not in tree by modifying the above process. Or another way is to add x to the tree, run the above procedure and delete x from tree.

Finally, note that the number of nodes in the tree equals number of distinct elements in the tree (Since when an element already present in the tree is added, we are not creating a new node). Thus the total height of the tree is $O(\log n)$ (as per out Balance assumption). Thus the time all the operations is $O(\log n)$.

4. For a set S consisting of (distinct) integers and an element $x \in S$, $rank(x)$ is the number of elements in S that are less than or equal to x. Given a Binary Search Tree T, give a recursive algorithm, that outputs all leaf nodes along with their ranks. Analyze the worst-case run time of your algorithm and prove its correctness. Your algorithm must be recursive, otherwise you will not receive any credit.

   Ans. Idea: Perform in-order traversal; while doing keep a counter that keeps tracks of number of nodes visited. When a leaf node is visited, output the leaf node along with the counter.

   Consider the following algorithm.

   (a) Input T whose root is r.

   (b) Global Variable $count = 0$.

   (c) Call $LeafRank(r)$

   The procedure $LeafRank(r)$ is as follows:

   (a) if r equals NULL, return;

   (b) LeafRank(r.left)

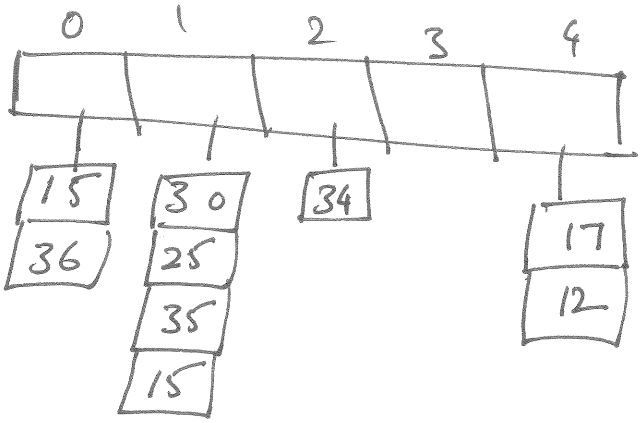
(c) $count++$; If r is leaf (can be checked by $r.right$ and $r.left$ are both NULL), output $\langle r, count$.

(d) LeafRank(r.right)

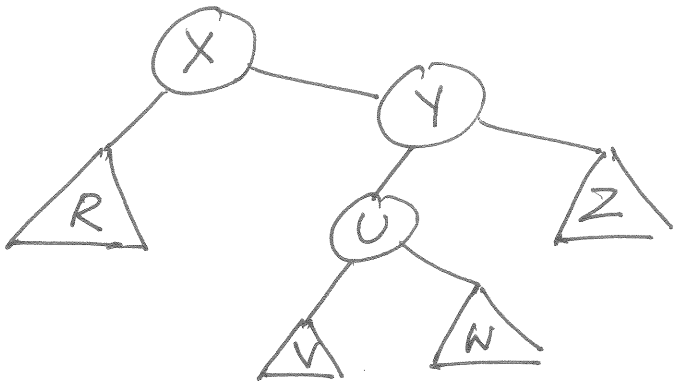
Suppose the tree rooted at r has ℓ nodes. Note that the total number of nodes in $r.right$ and $r.left$ is precisely $\ell - 1$. Thus time taken by the above procedure is $O(n)$.

We are performing in-order traversal which implies that we are visiting the nodes in the sorted order. Whenever we visit a node the counter is incremented exactly once. When we visit a node, we have visited all the node that are less than x. Thus when we visit a node x, the value of the counter is the number of elements that are less than x. Thus whenever we visit a leaf node the value of counter is all the elements that are less than the leaf node. Now the counter is incremented by 1 and the leaf node along with the rank is output.

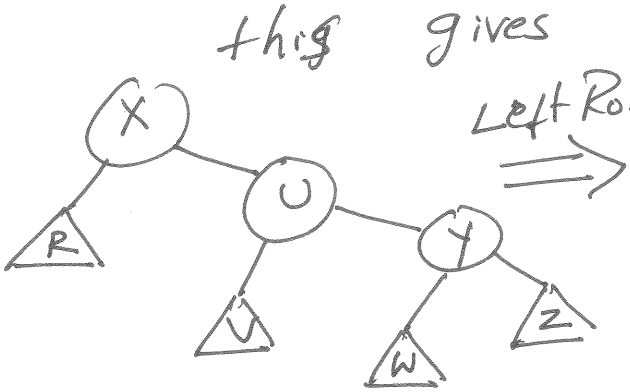
c)



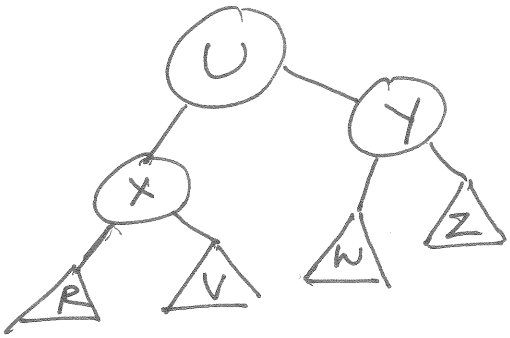
d) Original tree.



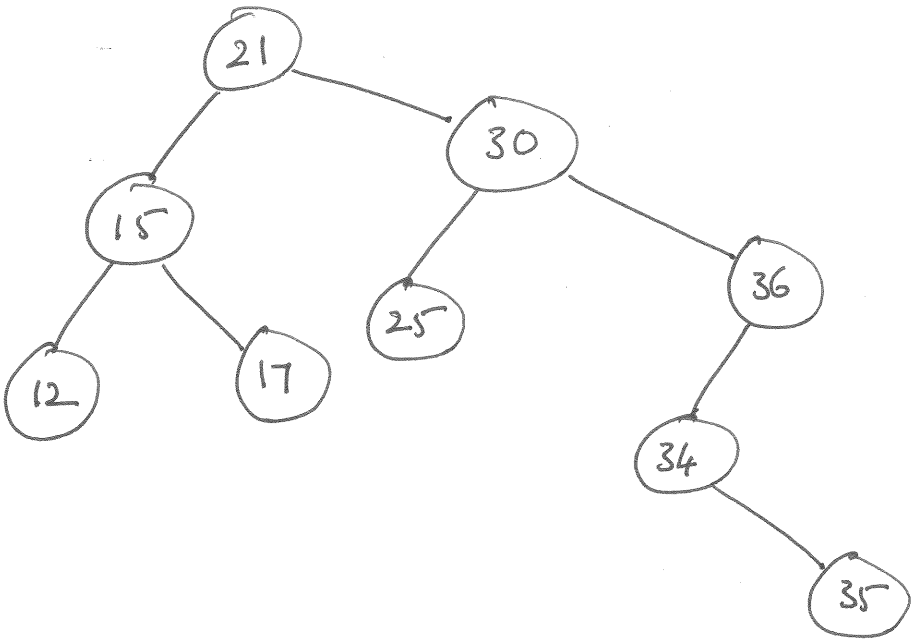
From B; it follows all trees R, V, W & Z have same height say k. Height of U is k+1, height of Y is k+2, height of X is k+3. We will first do RightRotate(Y) thig gives



LeftRotate(X)



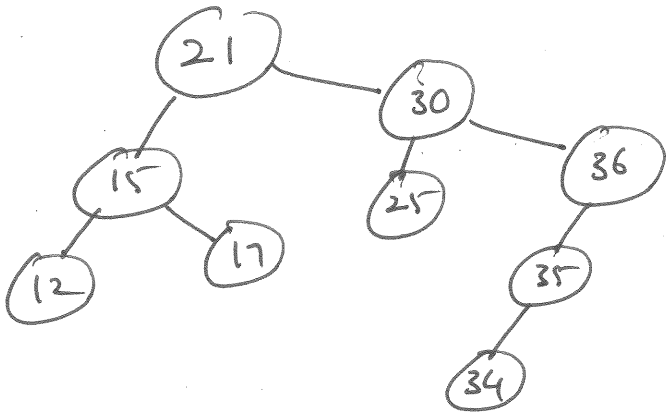
Now $B(U) = B(X) = B(Y) = 0$.



36 is not balanced.

We will first ~~Right~~ Left Rote (34).

this gives



Now do Right Rotate(36)