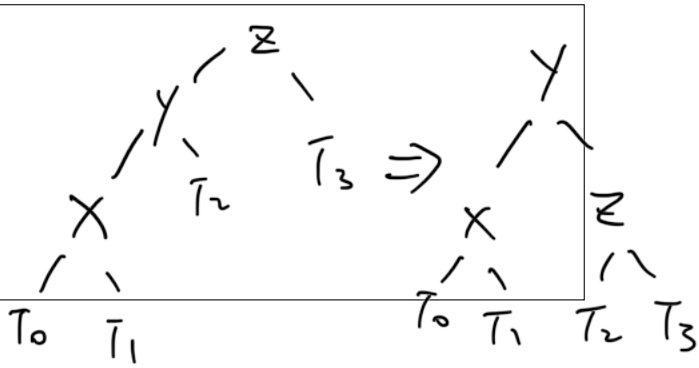
Warm-up

Problem 1. Give the full pseudocode for doing a trinode restructure at x, y, z where x is the left child of y and y is the left child of z.

Solution 1. Recall that x, y, z are internal nodes holding keys and due to their relation, $x.key < y.key < z.key$. First we identify the subtrees T_0, T_1, T_2, T_3 as in the picture of a single rotation from class (please refer to the picture from the slides). Then we update the left and right pointers of each node to reflect the updated tree structure.

1: $T_0 \leftarrow x.left$
2: $T_1 \leftarrow x.right$
3: $T_2 \leftarrow y.right$
4: $T_3 \leftarrow z.right$
5: $y.left, y.right \leftarrow x, z$
6: $x.left, x.right \leftarrow T_0, T_1$
7: $z.left, z.right \leftarrow T_2, T_3$



Problem 2. Consider the implementation of a binary search tree on n keys where the keys are stored in the internal nodes. Prove using induction that the height of the tree is at least $\log_2 n$.

Solution 2. Let $N(h)$ be the maximum number of keys that a tree of height h can have. Note that $N(1) = 1$ and $N(2) = 3$. For $h > 1$, we have $N(h) = 2N(h-1) + 1$ since we can take apart $N(h)$ into its left and right subtrees (each one no greater than $N(h-1)$) plus the root.

Using this inequality we can prove by induction that $N(h) = 2^h - 1$. The base case is $h = 1$, which is clearly true. For the inductive case, we note that $N(h) = 2N(h-1) + 1 = 2(2^{h-1} - 1) + 1 = 2^h - 1$, where the second inequality follows from our inductive hypothesis.

Taking the log on both sides of $N(h) = 2^h - 1$ we get $\log_2 N(h) < h$, which is precisely what we are after.

Problem 3. Consider the implementation of a binary search tree on n keys where the keys are stored in the internal nodes. Prove using induction that the number of external nodes is $n + 1$. Hypothesis

Solution 3. We give a proof by induction on n. Our base case is $n = 1$, which consists of a tree with one internal node and two external nodes. For the inductive case, consider any tree T with n keys. Let w be an internal node with two external children (there is always at least one) and consider the resulting tree after deleting w from the tree, which involves replacing w and its two external children with one external node. The resulting tree has $n - 1$ keys so by the induction hypothesis it has n external leaves. Inserting w back where it used to be, causes the tree to lose the leaf that replaced w, but it gains the two leaves of w. Hence, in total it gains one leaf, so the number of leaves in T is $n + 1$.

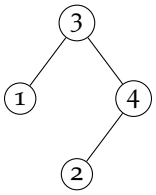
Problem solving

Problem 4. Consider the following algorithm for testing if a given binary tree has the binary search tree property.

```
1: function TEST-BST(T)
2:     for u ∈ T do
3:         if u.left ≠ nil and u.key < u.left.key then
4:             return False
5:         if u.right ≠ nil and u.key > u.right.key then
6:             return False
7:     return True
```

Either prove the correctness of this algorithm or provide a counter example where it fails to return the correct answer.

Solution 4. The algorithm is not correct as it returns True on the following tree lacking the binary search tree property.



all elements of right subtree should large than root node

Problem 5. Consider the following operation on a binary search tree: LARGEST() that returns the largest key in the tree.

Give an implementation that runs in $O(h)$ time, where h is the height of the tree.

Solution 5. Starting at the root, follow the right pointer until we reach a node u that has no right subtree and return the key of that node.

Since in a binary search tree we have that for every node u $u.left.key < u.key < u.right.key$, the above algorithm correctly finds the largest key in the tree.

The algorithm clearly runs in $O(h)$ time, as it takes $O(1)$ time per level of the tree it visits.

Problem 6. Consider the following operation on a binary search tree: SECOND-LARGEST() that returns the second largest key in the tree.

Give an implementation that runs in $O(h)$ time, where h is the height of the tree.

Solution 6. First we find the node w holding the largest key in the tree. If w has a left child, then the second largest key must be in this subtree, so we again search for the largest key in $w.left$.

Otherwise, the second largest key can be found at w's parent, provided that w has a parent, i.e., w is not the root of the tree. If w is the root of the tree and its left subtree is empty, then w is the only node in the tree and so the second-largest element is not defined and we can throw an exception.

The correctness follows from the binary search tree property. After finding the largest key, the second largest key is either its parent or the largest value in its left subtree. Since we know that every key in the left subtree of w is larger than the key of w's parent, the choice of the algorithm follows. If w has no left subtree, we know that there are no keys between w and its parent, as all keys in the left subtree of w's parent and those higher up in the tree are smaller than w's parent's key by the binary search tree property.

Since the algorithm takes $O(1)$ time per level of the tree it visits, the total running time is $O(h)$.

Problem 7. Consider the following operation on a binary search tree: MEDIAN() that returns the median element of the tree (the element at position $\lfloor n/2 \rfloor$ when considering all elements in sorted order).

Give an implementation that runs in $O(h)$, where h is the height of the tree. You are allowed to augment the insertion and delete routines to keep additional data at each node.

Solution 7. We can augment the nodes in our binary search trees with a size attribute. For a node u, $u.size$ is the number of nodes in the subtree rooted at u. When inserting a new key at some node w, we only need to increase by 1 the size of ancestors of w, which takes $O(h)$ time. Similarly, when we delete a node w with at least one external child, we only need to decrease by 1 the size of ancestors of w. Hence, the insertion and deletion times remain the same.

We implement a more powerful operation called POSITION(k), which returns the kth key (in sorted order) stored in the tree. When searching for the kth element at some node u, we check if $u.left.size \geq k$ in which case we know that the kth element is in the left subtree and we search for the kth key in $u.left$. Otherwise, if $u.left.size = k-1$, the kth key is at the root u. Finally, if $u.left.size < k-1$, then we know that the kth element is in the right subtree, so we search for the $(k - u.left.size - 1)$th key there, i.e., k minus the number of nodes smaller than the root of $u.right.key$.

The complexity is $O(h)$ because we do $O(1)$ work at each node and we only traverse one path from the root to the element we are searching for.

Problem 8. Consider the following binary search tree operation: REMOVE_ALL(k_1, k_2) that removes from the tree any key $k \in [k_1, k_2]$.

Give an implementation of this operation that runs in $O(h+s)$ where h is the height of the tree and s is the number of keys we are to remove.

Solution 8. We solve a simpler problem first; REMOVE-GREATER(w, k): remove all nodes with key at least a given key k from the tree rooted at w (the problem of removing nodes smaller than the given key is symmetric).

We do a search for k that takes us to a node u (which may hold k or may be an external node). Let v be any node on the path from u to the root that has a right child that is not on the path from u to the root. We can delete the nodes in those right subtrees right away since they are guaranteed to be greater than k. After that we may need to remove some of the nodes on the path from u to the root. Let v be a node in the path from u to the root. If u is a right descendant of v then there is no need to remove v as its key is smaller than k. If u is left descendant of v then after the first pruning step v has one external child so we can delete it in $O(1)$ time as we argued in class. Finally, we remove the bifurcation node u with the usual delete procedure.

Now to implement the REMOVE-ALL(k_1, k_2) operation, we perform the usual range search until we find the first node u such that $k_1 \leq u.key \leq k_2$. Then we call REMOVE-GREATER($u.left, k_1$)

and remove-smaller$(u.right, k_2)$. Finally, we delete u.

As we saw in class we can divide the nodes into boundary nodes (on the search path from the root to k_1 or k_2), inside nodes (those inside the range but not on the boundary), and outside nodes (those outside the range but not the boundary). The calls to remove-greater and remove-smaller run in $O(h+s)$ where h is the height of the tree and s is the number of deleted nodes, since the work done is proportional to the number of inside nodes, which is at most s, and the length of the search path, which is at most h. Finally, the removal of the bifurcation node u takes the usual $O(h)$ time. Hence, the overall complexity is $O(h+s)$.