Question 1

Theorem 1. INSERT *operations do not invalidate a binary search tree.*

Proof. Consider the following pseudocode for insertion:

function INSERT(node N, value x)
    if $N.value < x \wedge N.RightChild = nil$ **then**
        $N.RightChild \leftarrow$ SINGLETON(X)
    else if $N.value < x \wedge R.RightChild \neq nil$ **then**
        INSERT$(N.RightChild, x)$
    else if $N.value > x \wedge N.LeftChild = nil$ **then**
        $N.LeftChild \leftarrow$ SINGLETON(X)
    else if $N.value > x \wedge N.LeftChild \neq nil$ **then**
        INSERT$(N.LeftChild, x)$
    end if
end function

Assume that the tree whose root is at N (which may be a subtree of some larger tree) is initially valid. I show that it is thus after any operation, by induction on the recursion in this function.

- Base case 1: If we enter the first case, then the tree whose root is at N initially contains no right subtree. After the operation, it's left subtree is unchanged, and the right subtree consists only of one node whose value is x. Since the left subtree is initially valid, and as it's values are less than $N.value$ (as we have assumed N to be initially valid), and since the right subtree is also valid (since it is a singleton), and all the values in the right subtree (i.e. x) are greater or equal to $N.value$, thus N remains valid after the operation.
- Base case 2: If we enter the third case, that N is valid after the operation can be shown with an argument symmetrical to the previous one.
- Inductive step 1: If we enter the second case, then, by inductive hypothesis, $N.RightChild$'s subtree will be valid after inserting x into it. Moreover, $N.LeftChild$'s subtree is valid by assumption, and the values in the left subtree are not greater than $N.value$. Finally, by assumption, all values other than x in N's right subtree are greater than $N.value$, and x also satisfies this property in this case. So overall, N's tree is valid after the operation.
- Inductive step 2: If we enter the fourth case, that N is valid after the operation can be shown with an argument symmetrical to the previous one.

□

Theorem 2. DELETE *operations do not invalidate a binary search tree.*

Proof. Consider the following pseudocode for deletion:

1: **function** DELETE(node N)
2: **if** $N.RightChild = nil \wedge N.LeftChild = nil$ **then**
3: Set N's father's link to N to nil.
4: **else if** $N.LeftChild = nil \wedge N.RightChild \neq nil$ **then**
5: Promote $N.RightChild$ in place of N.
6: **else if** $N.LeftChild \neq nil \wedge N.RightChild = nil$ **then**
7: Promote $N.LeftChild$ in place of N.
8: **else if** $N.LeftChild \neq nil \wedge N.RightChild \neq nil$ **then**
9: $M \leftarrow$ SUCCESSOR(N)
10: SWAP(N, M)
11: DELETE(N).
12: **end if**
13: **end function**

As before I show that, if applied on some node N from a valid tree, then DELETE results in a valid tree, by induction on the recursion present in this function. This time, I assume that DELETE indeed removes a node from a tree:

- Base case 1: Note that it can easily be shown by induction that, for each node $M \neq N$ in the tree, supposing that l_M is the set of values in the left subtree of M before the operation, l'_M is this set after the operation, and that r_M, r'_M are defined symmetrically for the right subtree, then $l_M \subseteq l'_M$ and $r_M \subseteq r'_M$. Now, since the tree is initially valid, we have that $\forall x \in l_M . x < N.value$ and $\forall x \in r_M . N.value < x$. Since $l_M \subseteq l'_M$ and $r_M \subseteq r'_M$, we have that $\forall x \in l'_M . x < N.value$ and $\forall x \in r'_M . N.value < x$. Thus the tree is valid after the operation.

- Base case 2: If we enter the second case, a proof similar to the previous one suffices.

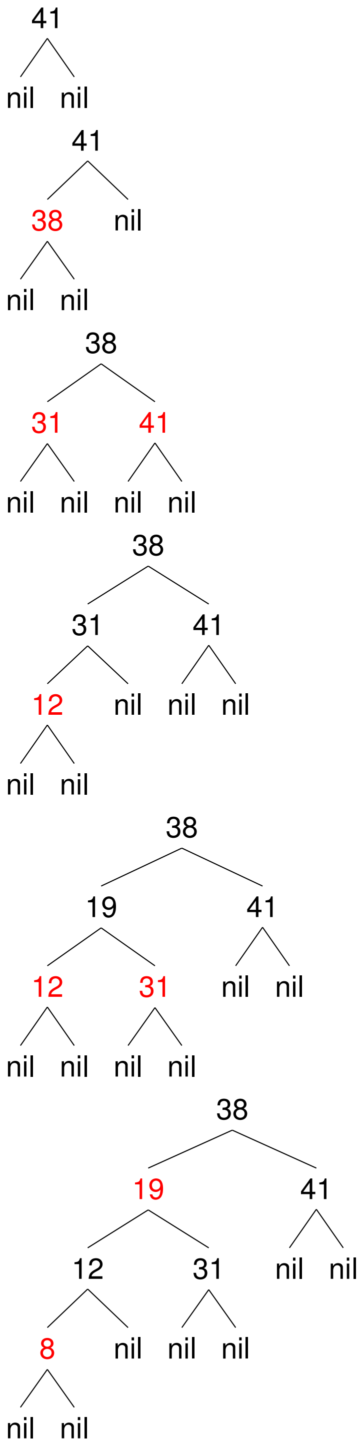
- Base case 3: If we enter the third case, a proof similar to the previous one suffices.

- Inductive step: If we enter the fourth case, then note that:
  - Step 9 does not modify the step.
  - Consider $V = \{v_1, ..., v_N, v_M, ..., v_n\}$, where $v_1 < ... < v_n$, $v_N = N.value$, $v_M = M.value$, the set of values in the tree (note that v_N and v_M are adjacent, as $M =$ SUCCESSOR(N)), and a relation $\prec \in V \times V$. Define $\prec$ to be the smallest total order that includes $v_1 \prec v_2 \prec ... \prec v_M \prec v_N \prec ... \prec v_n$. It is easy to see that, after step 10, the tree is valid w.r.t. $\prec$, although it is not so w.r.t. $<$.
  - Now, note that step 11 removes v_M. Moreover, by the inductive hypothesis, since the tree is initially valid w.r.t. $\prec$, it is also thus after this step. However, note that the tree does not include the value v_N, and that, by restricting $\prec$ to $\{v_1, ..., v_n\} \setminus \{v_N\}$, it becomes the same as $<$! This implies that the tree is also valid w.r.t. $<$, as desired.

$\square$

Question 2

The sequence of trees:

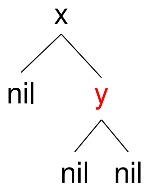


Question 3

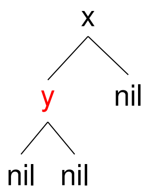
Theorem 3. *A red-black tree formed by inserting $n \geq 2$ nodes has at least one non-root red node.*

Proof. By induction on n:

- Base case: For $n = 2$, we note that any red-black tree formed by inserting two values $x \neq y$ has one of two different structures:
  - If $x < y$ then:

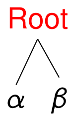


  - If $x > y$ then:



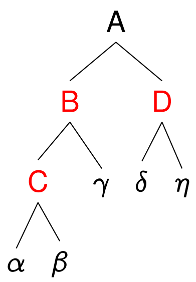
  Which obviously satisfy the claim.

- Inductive step: Note that inserting a node into a red-black tree is done by repeatedly applying various different transformations, according to different case logic. There are 4 different transformations:
  - Case 1:

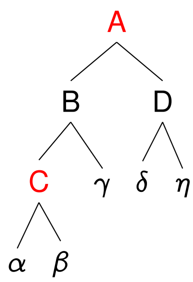


Becomes:

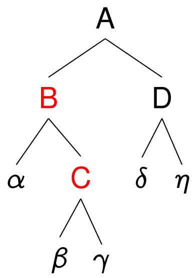
  - Case 2:



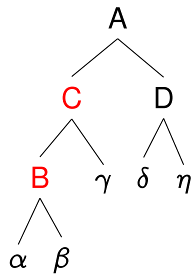
Becomes:



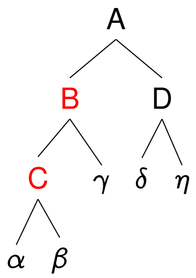
– Case 3:



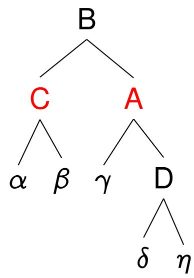
Becomes:



– Case 4:



Becomes:



But note that in all cases, if the tree initially has a non-root red node, then it also will have one after the transformation. This means that after any insertion, if the claim holds initially, it also holds at the end.

□

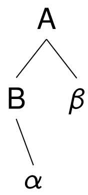
Question 4

Definition 1. *Say that a binary search tree is a "right-chain" if and only if all nodes in the tree lack a left child.*

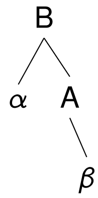
Theorem 4. *Any binary tree with n nodes, whose left subtree has l nodes, and whose left and right subtrees are right-chains, can be turned into a right chain with the same values in l rotations.*

Proof. By induction on l, for any n:

- Base case: For $l = 0$, the tree is already a right chain, so it can be "turned into" a right chain in $0 = l$ operations.
- Inductive step: For $l > 0$, note that performing a right rotation performed on the root will turn the tree into one whose root's left and right subtrees are right-chains, and whose left subtree has $l - 1$ nodes, viz:



Becomes:



(where α is a right chain of size $l - 1$ and β is also a right chain).
Now, by the inductive hypothesis, the resulting tree can be turned into a right-chain with the same elements using $l - 1$ rotations. Thus, overall, the initial tree can be turned into a right-chain with the same values using l operations.

□

Theorem 5. *Any binary tree with n nodes can be turned into a right-chain with the same values in at most n rotations.*

Proof. We prove this by induction on n:

- Base case: Note that all binary search trees with 1 node are already right-chains (as the sole node has neither left or right child). Thus they can be turned into right-chains with $0 \leq 1 = n$.
- Inductive step: For $n > 1$, suppose the root's left and right subtrees have l and r nodes respectively. First, apply the inductive hypothesis on the right subtree to turn it into a right-chain in $r - 1$ operations. Now, note that by applying a right rotation on the root, we maintain the property that the right subtree is a right-chain, and we reduce the left subtree's size by 1. This implies that by applying at most l further operations, we can turn the tree into a right-chain. Overall this implies that using $l + r - 1 \ leql + r + 1 = n$ operations we can turn the tree into a right-chain, as required.

□

Theorem 6. *A right-chain with n nodes can be turned into any binary search tree with the same values in at most n rotations.*

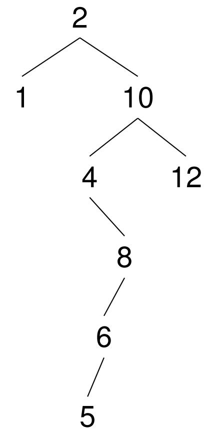
Proof. Use the sequence of rotations that would be used to transform the target binary search tree into a right-chain (which has n rotations at most, by the previous theorem), but reversed, and swapping right-rotations and left-rotations. □

Theorem 7. *Any binary tree with n nodes can be turned into any other binary tree with the same values in at most $2n$ rotations.*

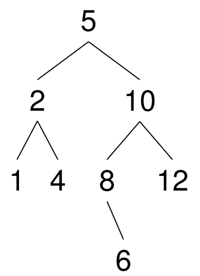
Proof. Use the previous two theorems to turn the tree first into a right-chain in n rotations, then into the target tree in another n rotations. $\square$

Question 5

After the first splay:

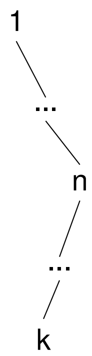


After the second splay:



Question 6

Take the definiton of right-chains from question 4, and define left-chains analogously. Now, consider a right-chain with n values, viz. $\{1, 2, ..., n\}$. Suppose we were to splay the node n with this alternate scheme. By simulating the rotations generated, we can see that, during the splay, the tree is of the following shape:



where k is some value in $\{1, 2, ..., n\}$. And the resulting tree after the splay is a left-chain with n values, with the splay costing $\Omega(n)$ operations. Note that then doing a splay on node 1 now leads again to a cost of $\Omega(n)$, and a right-chain with n values. Alternating these m times gives us a total cost of $\Omega(nm)$.