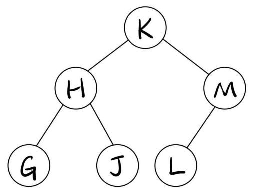
CSC265 Homework 2 | Fall 2021
Name: Haojun Qiu | Student Number: 1006900622 | Who I discussed with: Runshi Yang, Darcy Wang

---

Q1



Q2

```
INSERT($T, x$)
$P = $ NIL
$Q = $ NIL
$curr\text{-}p = P$
$curr\text{-}q = Q$
$rest = T.root$
while $rest \neq$ NIL
    if $rest.key < x.key$
        if $curr\text{-}p ==$ NIL
            $P = rest$
            $rest = rest.right$
            $P.right = $ NIL
        else
            $curr\text{-}p.right = rest$
            $rest = rest.right$
            $curr\text{-}p = curr\text{-}p.right$
            $curr\text{-}p.right = $ NIL
    else    // $rest.key > x.key$
        if $curr\text{-}q ==$ NIL
            $Q = rest$
            $rest = rest.left$
            $Q.left = $ NIL
        else
            $curr\text{-}q.left = rest$
            $rest = rest.left$
            $curr\text{-}q = curr\text{-}q.left$
            $curr\text{-}q.left = $ NIL
$T.root = x$
$x.left = P$
$x.right = Q$
```

Q3

I will prove a lemma first to facilitate proof below and also Q7. In below, I say a node x satisfies the BST property when if y is a node in the left subtree of x then $y.key \leq x.key$, and if y is a node in the right subtree of x then $y.key \geq x.key$. I say a (sub)tree satisfies the BST property when all nodes in it satisfy BST property.

Lemma 1. *Let T_L be a binary search (sub)tree rooted at node x with distinct keys, and T_R be a binary search (sub)tree rooted at node y with distinct keys that are greater than keys of nodes in T_L. Let tree T be a (sub)tree resulting from connecting T_L and T_R by setting the right attribute of the node with maximum key in T_L to be y, the root of T_R. Then, T satisfies the BST property.*

Proof. Notice that before conjunction, all nodes in T_L and T_R satisfy the BST property. After conjunction, in T, nodes whose descendants does not change will still preserve the BST property, otherwise its BST property can possibly be violated. Denote the node with maximum key in T_L as m, and root of T_L as r. The nodes in T that can possibly violate BST are who contain nodes in T_R as proper descendants in T, i.e., y's proper ancestor in T, or say m and m's proper ancestor. I encapsulate these nodes in T_L into a sequence $S = \langle m, p_1, p_2, \ldots, r\rangle$, where node comes right after itself in the sequence is its parent node (e.g., p_1 is the parent of m). It suffices to show that all nodes in S satisfies BST property. I argue that for all $i \in \{1, 2, \ldots, |S|-1\}$, $S[i+1].right = S[i]$, i.e., they are right child of their parent (except the root). If this is not the case, then there is $i \in \{1, 2, \ldots, |S|-1\}$ such that $S[i+1].right \neq S[i]$, i.e., $S[i+1].left = S[i]$, which implies m will be in the left subtree (or, left descendants) of $S[i+1]$, which contradicts to that T_L, in particular node $S[i+1]$, satisfies the BST property. Then, to complete the proof, using induction to show that $S[1]$, i.e., m satisfies the BST, and for all $i \in \{1, 2, \ldots, |S|-1\}$, if $S[i]$ satisfies the BST so does $S[i+1]$. This will imply all nodes in S satifies the BST property.

Let $i = 1$, $S[1] = m$, which is the node with maximum key in T_L, and its left subtree does not change so the keys of nodes in its left subtree still less than $m.key$. Further since its right subtree becomes T_R, in which all nodes have keys greater than all nodes in T_L, in particular m, so m satisfies the BST property. Let $i \in \{1, 2, \ldots, |S|-1\}$, and assume that $S[i]$ satisfies the BST property. I show that $S[i+1]$ satisfies BST property as well. As shown above we know that $S[i+1].right = S[i]$. We know that if l is a node in the left subtree of S, then $l.key < S[i+1].key$, which does not change as its left subtree is still the same as in T_L. Then it remains to show if l in right subtree of $S[i+1]$, i.e., in the subtree rooted at $S[i]$, then $S[i+1].key < a.key$.
Case 1: l is $S[i]$ or in the left subtree of $S[i]$. Then $m.key < l.key$ by the fact that T_L satisfies satisfies BST and postion of $S[i]$ and its left subtree does not change from T_L to T.
Case 2: l in the right subtree of $S[i]$. Then since $S[i]$ in T satisfies $S[i].key < l.key$ (by induction hypothesis that $S[i]$ satisfies BST), so $S[i+1].key < S[i].key < l.key$. So $S[i+1]$ satifies BST property, as wanted. ■

Then I prove my insertion algorithm works correctly.

Proof. Let T as the input to INSERTION(T, x) be a binary search tree have distinct keys, and x to be a node with arbitrary key that does not have the same key as any of nodes in T. Firstly, I define some loop invariant:

1. All nodes in the subtree rooted at P and all nodes in the subtree rooted at Q and all nodes in subtree rooted at *rest* together are all nodes in input tree T before any mutation.
2. The keys of all nodes in the subtree rooted at P are less than $x.key$, and $x.key$ is less than the keys of all nodes in the subtree rooted at Q.
3. The keys of all nodes in the subtree rooted at P are all less than the keys of all nodes in the subtree rooted at *rest*, and the keys of the nodes in the subtree rooted at *rest* are all less than the keys of all nodes in the subtree rooted at Q.
4. The subtrees rooted at P, and at Q, and at *rest* all satisfies the BST property. That is, let x be a node in the subtree rooted at P or the subtree rooted at Q or the subtree rooted at *rest*. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.
5. If $curr\text{-}p \neq$ NIL, then $curr\text{-}p$ is the node with largest key in the subtree rooted at P. If $curr\text{-}q \neq$ NIL, then $curr\text{-}q$ is the node with smallest key in the subtree rooted at Q.
6. The height of the tree rooted at *rest* decreases at least by 1 than the height of *rest* at the end of last iteration if exists (i.e., before the start of this iteration).

Base Case / Initialization: The invariant 1 - 6 above all hold before entering the first iteration of the loop.
(1) – (4) are all vacuously true since $P = Q =$ NIL before the loop starts, hence contains no node.
(5) is also vacuously true because initially, $curr\text{-}p =$ NIL and $curr\text{-}q =$ NIL.
(6) is vacuously true as well since there is no last iteration when we have not yet entered the loop.

Inductive Step / Loop Maintenance: Now, consider an arbitrary iteration, and assume that the loop invariant holds before this iteration, I will show that they will hold after this iteration. We denote the variable after (or say, at the end of) this iteration with a superscript $'$, i.e., P', Q' $rest'$ and etc, and use T to represents the input tree before any mutation to it. As in code we have two cases each with two sub-cases to verify.

- **Case 1**: *rest.key* $<$ *x.key.*
  - **Case 1.1**: *current-p* $=$ NIL.

    In this case line 8 evaluates to true, so lines 9 -11 get executed. We know that $Q' = Q$ as no mutation happen to Q and node in the subtree rooted at Q (as algorithm and (5) suggest, we change Q only if we mutate *curr-q* or Q itself, which is not the case). Also, $P =$ NIL in this case, we show by contradiction. If P is not NIL, as observed in line 9, it gets assigned to *rest* - its only assignment statement (and *rest* $\neq$ NIL by line 6). Then, if *curr-p* have not been updated before, *curr-p* $=$ *rest* $\neq$ NIL (as we make it aliases to P in line 3), and if *curr-p* have been updated, it will certainly be updated at line 15, and lines 13 to 15 tell that *curr-q* points to the node that *rest* points before *rest* been updated, which is not NIL by line 6. Hence, we reach a contradiction, as line 8 suggests that *curr-p* $=$ NIL. So, $P =$ NIL. We observe that P' now contains node *rest* and all the nodes in the subtree rooted at $rest.left$ (by lines 9 and 11).

    (1): By induction hypothesis, we have that all nodes in the subtree rooted at P and all nodes in the subtree rooted Q and all nodes in the subtree rooted at *rest* together are all nodes in T before any mutation. Together with $P =$ NIL and $Q = Q'$, we can conclude that all nodes in P' (that contains *rest* and all nodes in the subtree rooted at $rest.left$), $rest'$ (that contains all nodes in subtree rooted *rest.right* by line 10) and Q', together, are all nodes in T before any mutation to it.

    (2): All nodes in P' are the node that *rest* points to and all nodes in the subtree rooted at $rest.left$. We know *rest.key* $<$ *x.key* in this case. Also, by induction hypothesis of (4), all nodes in the subtree rooted at *rest* satisfies the BST property, in particlar, the keys in all the nodes in the subtree rooted at *rest.left* are less than the *rest.key*, thus less than *x.key*. It justifies that the keys of all nodes in the subtree rooted at P are less than *x.key*.

    Next, by induction hypothesis of (2), *x.key* is less than all keys of nodes in the subtree rooted at $Q' = Q$.

    (3): All nodes in P' are node that *rest* points to and all nodes in the subtree rooted at $rest.left$, whose keys are less than all the keys of nodes in $rest' =$ *rest.right*, by induction hypothesis of (4) that the subtree rooted at *rest* satisfies BST property. Next, by the induction hypothesis of (3), keys of all nodes rooted at *rest*, and thus keys of all nodes in the subtree rooted at $rest' =$ *rest.right*, are less than the keys of all nodes rooted at $Q = Q'$.

    (4): Firstly, by the induction hypothesis of (4) we know that the subtree rooted at P, Q, satisfies the BST property. The subtree rooted at P' is *rest* with its left subtree connected, hence still satisfies BST property. The subtree rooted at $rest'$ is the subtree rooted at *rest.right* which still satisfies BST property. The subtree rooted at $Q' = Q$ satisfies the BST property as well.

    (5): *curr-p* $=$ NIL so the first half of the statement is a vacuous truth. Assume *curr-q*$' \neq$ NIL. Since *curr-p* is not updated at this iteration, *curr-q*$' =$ *curr-q*, so by the induction hypothesis of (5) we have *curr-q*' $=$ *curr-q* is the node with smallest key in the subtree rooted at $Q' = Q$.

    (6) We want to show that the height of the subtree rooted at $rest' =$ *rest.right* is at least 1 less than the height of subtree rooted at *rest*. This is true since the height of the subtree rooted at *rest* $= \max\{$the height of the subtree rooted at $rest.left$, the height of the subtree rooted at *rest.right*$\} + 1$, which will be at least 1 greater than the height of the subtree rooted at $rest' =$ *rest.right*, as wanted.
  - **Case 1.2**: *current-p* $\neq$ NIL.

    In this case lines 13-16 are executed. Since *current-p* $\neq$ NIL, by the induction hypothesis of (5), *curr-p* is the node with the largest key in the tree rooted at P. Then, P' is the tree contains nodes in subtree rooted at P, and node *rest* points to, and all nodes in the subtree rooted at $rest.left$, with the structure of *rest* being connected to *curr-p.right* along with its left subtree (but not the right). And $rest' =$ *rest.right*, same as case 1.1. Also, $Q' = Q$ for the same reason stated in case 1.1.

    (1): From above observation, we know that all nodes in the subtree rooted at P and all nodes in the subtree rooted at *rest* are, all nodes in the subtree rooted at P' and all nodes in the subtree rooted at $rest'$. And also since $Q = Q'$, we know that all nodes in the subtrees rooted at P' and at Q' and at $rest'$ are all nodes in the subtrees rooted at P, Q and *rest*, which by the induction hypothesis of (1) are all nodes in input tree T before any mutation.

(2): By induction hypothesis of (2), The keys of all nodes in the subtree rooted at P are less than $x.key$. And in this case, $rest.key < x.key$. Further, by induction hypothesis of (5), the subtree rooted at *rest* satisfies BST property so the keys of all nodes in the subtree rooted at $rest.left$ are less than $rest.key$ hence are less than $x.key$. These are all nodes in the subtree rooted at P, so we showed the first half.

Next, by induction hypothesis of (2), $x.ley$ is less than all the keys of nodes in the subtree rooted at $Q' = Q$.

(3): We know that the keys of all nodes in the subtree rooted at P are all less than the kesy of all nodes in the subtree rooted at *rest* (by the induction hypothesis of (2)), thus the subtree rooted at $rest' = rest.right$. Also, we have $rest.key$ and the keys of the nodes in the subtree rooted at $rest.left$ are all less than all the keys of the nodes in the subtree rooted at $rest' = rest.right$, by the induction hypothesis of (4) that the subtree rooted at *rest* satisfies the BST property. These are all nodes in P' so we have shown the first half of the statement.

Next, by the induction hypothesis of (3) that keys of all nodes rooted at *rest*, and thus the keys of all nodes in the subtree rooted at $rest' = rest.right$, are less than the keys of all nodes rooted at $Q = Q'$.

(4): We know all nodes in P satisfies BST property. And *curr-p* is the node with largest key in P, also $curr - p.right$ is set to be *rest* with its left subtree, which is satisfies the BST property as well. So, by lemma 1 (replace T_L with P and T_R with *rest* together with its left subtree), subtree rooted at P' resulting from this conjunction still satisfies the BST property.
The subtree rooted at $rest' = rest.right$ satisfies the BST property since *rest* does so by the induction hypothesis of (4). The subtree rooted at Q satisfies the BST property by the fact that $Q' = Q$ and the induction hypothesis of (4).

(5): By lines 13 and 15, $curr\text{-}p' = rest$, which is the node with largest key in the subtree rooted at P' since firstly $rest.key$ is greater than the keys of all nodes in the subtree rooted at P (by the induction hypothesis of (3) as *rest* is a node in the subtree rooted at *rest*). Also, $rest.key$ are greater than the keys of all nodes in the subtree root at $rest.left$ (by the induction hypothesis of (4) that the subtree rooted at *rest* satisfies BST property, particularly the node *rest*.). Since these are all nodes in P except $curr\text{-}p' = rest$ itself, $curr\text{-}p'$ is the node with the largest key in the subtree rooted at P.

(6): The argument is the same is as case 1.1 (6).

- **Case 2**: $rest.key > x.key$.

  The argument will be exactly symmetric to case 1, switch keywords left and right, and *curr-p* and *curr-q*, and P and Q, and "<" and ">" (or, "less" and "greater").

We have shown that loop invariant holds at each iteration. Now, we turn to termination of the loop.

Termination: Let loop measure m to be the height of the *rest*. By the loop invariant (6), we know that it decreases at least by 1 at each iteration. Also, we know that the height of a tree can only be a natural number. So, values of m at each iteration forms a decreasing sequence of natural numbers, which would be finite. That indicates that loop will terminate. Also, the entire algorithm will terminate as well since the other lines certainly terminates.

Since the loop terminate we know that it runs a finite number of iteration. So, when it terminates, by the termination condition on line 6, we know that $rest = \text{NIL}$. Together with loop invariant (1), we know that the nodes in subtree rooted at P and the nodes in the subtree rooted at Q together are all nodes in the input tree T before any mutation. Hence, by lines 27-29 we know that after the algorithm terminates, (after performed insertion), T contains the nodes that were in T before the algorithm is called on it, plus the newly inserted node x. We remains to show this T is a binary search tree.

By loop invariant (4) the subtree rooted at P and the subtree rooted at Q satisfies the BST property, i.e., all nodes x in the subtree rooted at P or in the subtree rooted at Q satisfies that if y is a node in the left subtree of x then $y.key \leq x.key$ and if y is a node in the right substree of x they $y.key \geq x.key$. So, to show that T after insertion is completed is a binary search, we only left to show that $T.root$ satisfies that if y is a node in the left subtree of $T.root$ (i.e., y in P), then $y.key \leq T.root.key$, and if y is a node in the right subtree of $T.root$

(i.e., y in Q), then $y.key \geq T.root.key$. This follows directly from lines 27-29 and the loop invariant (2). This completes the proof. ■

Q4

Upper Bound: Let T as the input to INSERTION(T, x) be a binary search tree have distinct keys, and x to be a node with arbitrary key that does not have the same key as any of nodes in T. I will define and prove two new loop invariants for the loop in my insertion algorithm. Let $h(P)$, $h(Q)$, $h(rest)$, $h(T)$ and $h(T_i)$ denote the height of subtree rooted at P, and subtree rooted at Q, and subtree rooted at $rest$, and the tree T before insertion, and the tree T after insertion, respectively. Denote the number of edges on the simple path from P to *curr-p* as l.

- $l + h(rest) \leq h(T)$
- $h(P) \leq h(T)$

Initially, $l = 0$ since $curr\text{-}q = P =$ NIL and $h(rest) = h(T)$ so the equality follows, and $h(P) = 0 \leq h(T)$. Then, consider an arbitrary iteration, and assume that two invariant hold before this iteration. I will show that they holds at the end of this iteration. We denote the variable at the end of this iteration with a superscript $'$, same as in Q3. We know that l can increase by 1 or 0 (at each iteration, increase by 1 if line 15 is reached). And, $h(rest)$ decreases at least by 1 at each iteration by loop invariant (6) on Q3. Together, $l + h(rest)$ either not change or decrease, and combined with $l + h(rest) \leq h(T)$, we get $l' + h(rest)' \leq h(T)'$.

Next, we know that $h(P)$ can possibly increase only if we move *rest* along with its left subtree into the right of *curr-p* (or just set P to be it), by lines 13-16 or 9-11. By the first invariant, we know that $l + h(rest)$, the maximum number of edges from P to any new leaves of the tree will be less than or equal to $h(T) = h(T)'$. Since $h(P) \leq h(T)$, we know that the number of edges from P to any old leaves is also less than or equal to $h(T) = h(T)'$. Hence, by defnition, the heigh of subtree rooted at P, $h(P)$ will still be less than or equal to $h(T) = h(T)'$.

In similar way, define k to be the number of edges in the simple path from Q to *curr-q*, and we can verify that $h(Q) \leq h(T)$ (with the helper invariant $k + h(rest) \leq h(T)$). The argument would be symmetric.

Now, since we know the loop terminates, it terminates with $h(P) \leq h(T)$ and $h(Q) \leq h(T)$. Together with lines 27-29, we know that $h(T_i) = \max\{h(P), h(Q)\} + 1 \leq h(T) + 1$, that is, the height of the tree after insertion can be at most its height before insertion, plus 1. That is, the upper bound on the maximum height increase is 1.

Lower Bound: I will show the lower bound on the maximum height increase is 1. Firstly, we define an input family. Let $n \in \mathbb{N}$. Let input binary search tree T consists of n nodes, where nodes have distinct keys, and input x with its key greater than all the keys of nodes in T. Then, as partly explained in Q8, since $x.key$ less than all keys in original T, at each iteration subtree rooted at P will be appended and eventually restore the structure of the original input T, thus $h(P) = h(T)$. Then, in lines 27-29, subtree rooted at P is assigned to be the left subtree of x − the root of the tree after insertion, thus the height of entire tree after insertion $h(T_i)$, is $h(P) + 1 = h(T) + 1$.

Q5

We transplant and move the tree around after removing the node to be deleted.

When the deleted node x is the root of the tree T, we consider 4 cases. If both left and right child of x is empty, we just remove it and make entire T empty. If x has only one child (has left subtree or right subtree only), then we just make the left (or right) subtree of x the new tree, by modfiying $T.root$. If x has both left and right subtree, we make left child the new root of T and transplant entire right subtree of root to be the right subtree of the maximum-key node in the left subtree of original root.

When the deleted node x is not the root, the idea is similar. Still we have 4 cases, each with two subcases. Firstly we find the deleted node in the subtree, and keep track of the a parent pointer so that when we find the x in tree we have a pointer to its parent node. Then, after "search", when x has no child, we just make p points to NIL wherever x was in (left of p or right of p). If x has only one non-empty child/subtree, we make p points to the root of that subtree ($p.left$ or $p.right$ depends on x is left or right subtree of p). Lastly, if x has two subtrees, we move the right subtree of x to be the right subtree of the maximum-key node in left subtree of x, and have p connect to the root of the x's left subtree ($p.left$ or $p.right$ depends on x is left or right subtree of p).

Q6

```
DELETE($T, x$)
 1  if $T.root == x$
 2      if $x.left ==$ NIL and $x.right ==$ NIL
 3          $T.root =$ NIL
 4      elseif $x.left \neq$ NIL and $x.right ==$ NIL
 5          $T.root = x.left$
 6      elseif $x.left ==$ NIL and $x.right \neq$ NIL
 7          $T.root = x.right$
 8      else
 9          $m =$ TREE-MAXIMUM($x.left$)
10          $m.right = x.right$
11          $T.root = x.left$
12  else    // when $T.root \neq x$
13      $p =$ NIL
14      $s = T.root$
15      while $s \neq x$
16          $p = s$
17          if $x.key < s.key$
18              $s = s.left$
19          else
20              $s = s.right$
21      if $x.left ==$ NIL and $x.right ==$ NIL
22          if $x == p.left$
23              $p.left =$ NIL
24          else
25              $p.right =$ NIL
26      elseif $x.left \neq$ NIL and $x.right ==$ NIL
27          if $x == p.left$
28              $p.left = x.left$
29          else
30              $p.right = x.left$
31      elseif $x.left ==$ NIL and $x.right \neq$ NIL
32          if $x == p.left$
33              $p.left = x.right$
34          else
35              $p.right = x.right$
36      else
37          $m =$ TREE-MAXIMUM($x.left$)
38          $m.right = x.right$
39          if $x == p.left$
40              $p.left = x.left$
41          else
42              $p.right = x.left$
```

Q7

When the deleted node is the root, my algorithm from lines 1 - 11, only remove the root, so the left and right subtrees of x is preserved (if exists) and left or right child of x becomes the new root (if exists), and possibly two being connected when both exists, so resulting tree contains all the node before deletion except the deleted node x. It is binary search tree since firstly, if it has no child (both subtree empty), the resulting tree has root being assigned to NIL, representing an empty binary search tree. If x has only one non-empty subtree, then either left or right child of is moved the root of the tree, where we don't change the structure of the subtree tree so still satisfies BST property. If x has both subtree non empty. Then, our algorithm make right subtree of x to be the *right* of maximum-key node in left subtree of x, by lemma 7, since both left and right subtree of x satisfies BST property and keys of nodes in left subtree less than which of nodes in right subtree, the resulting tree is a BST.

When the deleted node is not the root of T. Lines 13 - 42 are executed. Lines 13 - 20 do the search for x, and when finds it, we also have p points to the parent node of x. This p will not be NIL since x is not the root so its parent can not be NIL.

- If x has both left and right being empty, in lines 22-25, we just set the position of x to be NIL ($p.left$ or $p.right$ depends on s is left or right child of its parent p). Only x is removed and it has no subtree so resulting tree contains all nodes from tree before deletion except x, and the binary search property is preserved.
- If x has only left subtree being non-empty, then lines 27-30 are executed. The $left$ or $right$ attribute of p is set to be $x.left$, so the resulting tree contains all nodes in original T except x. We know there is only local change to the structure of the subtree rooted at p, and BST property of this subtree is preserved since the left subtree of x is originally part of the p's left or right subtree (depends on x is left or right child). Also, there is no node has new or more descendants than before, so the BST property of other nodes in T other than any in the subtree rooted at p still preserved. Hence, the resulting tree is a bianry search tree.
- If x has only right subtree, lines 32-35 are executed and the resulting tree is also a binary search tree contains all nodes in T before deletion except x, the arugument would be symmetric to above case.
- If x has both left and right subtree being non-empty, my algorithm will transplant $x.right$ to be the right subtree of the maximum-key node in left subtree of x, and $x.left$ will becomes p's child ($p.right$ or $p.left$ depends on x's original position), hence we keep all nodes previously in the tree except x. Also, by lines 37-38 and lemma 1, the transplant resulting from setting the right subtree of maximum-key node in left subtree of x results that the subtree rooted at $x.left$ still satisfies BST property (since T, particularly node x satifies BST property before mutation, so the keys of all nodes in right subtree of x are greater which of left subtree). After $p.right$ or $p.left$ is being set to $x.left$, the entire tree is still a BST because firstly we preserve the BST property for the subtree rooted at P and there is no node has new or more descendants than before, so the BST property of other nodes in T other than any in the subtree rooted at p still preserved. Hence, the resulting tree is a binary search tree.

Q8

In the high level, it is because my delete algorithm splits the two subtrees, and form a clear "search path" so that when insert is called, the insert algorithm will restore the two subtrees I combined, and have them individually connect back to the inserted node as the root of T (which is the originally deleted root).

- When the root is the only node in the tree, i.e., has no subtree, after deletion $T.root$ is set to NIL. Then when we insert it using algorithm in part 2, the while loop condition checks to false before any iteration, and make $T.root = x$, restoring the tree.
- When the root has only left subtree being non-empty, the left subtree becomes the T after deletion. Since all nodes in it are less than the original nodes, $rest, key < x.key$ checks to true all iterations. And at each iteration, the nodes on the traversing path (roots of rest) at every iteration, along with its left subtree, will be appended to the right most node of P, and so at every iteration P and rest connected together is original left subtree (by setting $curr\text{-}p$.right $= rest$). So when loop stops, $rest =$ NIL is exhausted and P gets back to the original left subtree, hence in lines 27-29 of insertion algorithm, entire tree is restored by setting left subtree of the root to be P.
- When the root has only right subtree being non-empty, the argument would be symmetric to the second case, after deletion, we restore original right subtree as Q during insertion, which will become the right subtree of original root/inserted node, as needed.

- The last case is the root x has two non-empty subtrees. When insert, it actually do what case 2 do for a few iteration and then do what case 3 do for a few iteration. After deletion, we have the right subtree of original root x to be the right subtree of maximum-key node in left subtree of x. Then, we call the insert algorithm to insert x back to the tree. By our insertion algorithm, we know that when loops starts, it traverses the right most path of the original left subtree, from root to downwards. And as the reasoning in case 2, the continuous expansion will make P eventually get back to the left subtree of the original root (x). After these iteration, $rest$ refers to the root of original right subtree. Then, symmetrically, the algorithm will keep going to the left of the $rest$ as $x.key$ smaller than all keys in the original subtree, and will traverse the left most path of the original right tree, append node on the path along with its right subtree to Q. Since when doing deletion we do not change the structure of original right subtree locally we eventaully have Q restore the original right subtree of x. And at the end, by lines 27-29 in insertion algorithm, setting P and Q to be left and right subtrees of the root, we get back the tree before deletion!