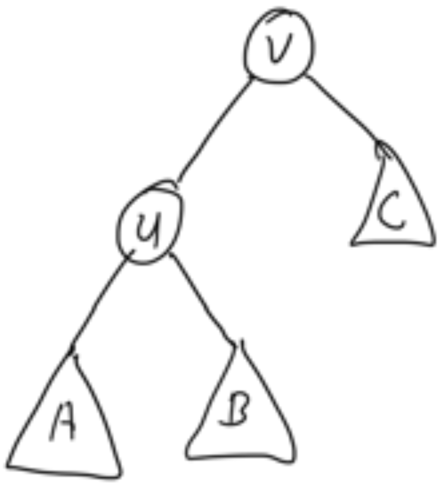
# FA HW 6

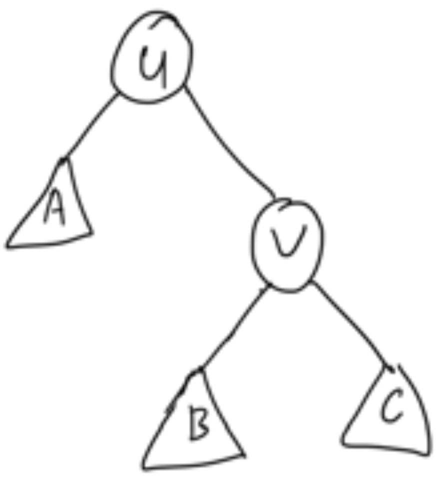
Problem 1.

1-a

Given,




Output of LeftRotate (T, u):



1. b

```
Left Rotate (T, u)
{
    v = u.p

    if v == v.p.left :
            v.p.left = u
            u.p = v.p

    if v == v.p.right :
            v.p.right = u
            u.p = v.p

    if v.p == Nil :
            u.p = Nil

    v.left = u.right

    u.right = v

    v.p = u

}
```

→ We assign v's parent as u's parent.

→ u's right sub tree is assigned as v's left subtree.

→ v is assigned as u's right subtree child

→ u is assigned v's parent.

→ The running time of this algorithm is O(1).

→ The number of operations required to rotate u does not grow with the size of the binary search tree T. The time complexity is constant.

$$\therefore \quad T(n) = O(1)$$

1.c

```
Rotate Delete (T, z)
{
    while (z.left != Nil) :
            Left Rotate (T, z.left)
    Transplant (T, z, z.right)
}
```

Statement 1: If z has no left child, Transplanting the right child of z with z will preserve the structure of the BST in the deletion of z.

lemma 2: If T is a binary search tree, a sub tree T' of T will also preserve the binary search tree properties. A violation of the properties in T' would mean a violation of the properties in T; but T is a binary search tree where the properties must hold for every node.

Proof 1:
If z is a left child of s,
r is the right child of z.

s.key ≥ r.key as r is to the left of s in the subtree rooted at s. The subtree rooted at r can therefore be placed to the left of s. This follows from lemma 2.

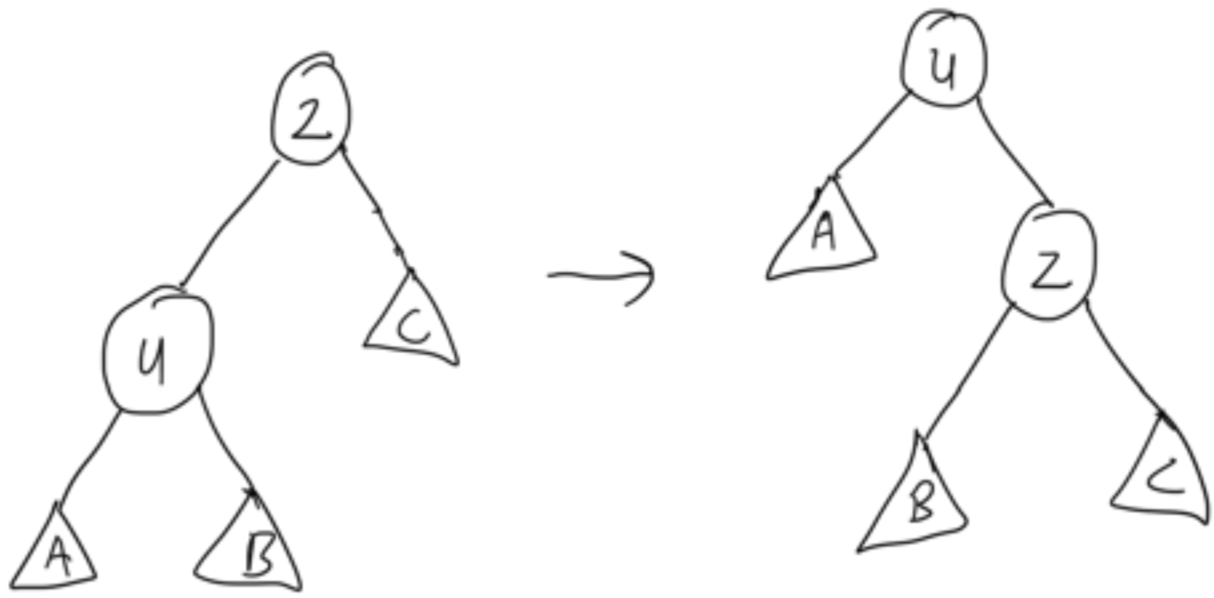
Similarly, we can show if z is the right child of s.

This shows our validity of statement 1.

Statement 2:
At every iteration of our while loop, the

depth of the left subtree rooted at z will decrease by atleast 1.

Proof 2: Consider the Left Rotate call for some iteration i inside the while loop,



Based on the left rotate call,

Before call,

$$\text{Depth}(\text{Left sub tree}(z)) = 1\{u\} + \max(\text{Depth}(A), \text{Depth}(B))$$

After call,

$$\text{Depth}(\text{Left sub tree}(z)) = \text{Depth}(B)$$

$\therefore$ Statement 2 is valid.

→ The while loop will run for iterations in the order corresponding to the depth of the original left sub tree.

∴ After number of calls corresponding to the original depth of the left sub tree of 2, we get the conditions for statement 1.

Using the validity of statement 1, we use the transplant operation in order to get the final result.

The leftrotate and transform operations take constant time. The depth of the left subtree is $O(\log n)$ if the tree is balanced and $O(n)$ in general

$$\therefore \quad T(n) = O(n)$$

1.d

# Find inorder successor

```
Rotate Successor (T, u)

{

    if u.right.left == Nil :
            return u.right.key
```

```
    while (u.right.left != Nil):

            LeftRotate (T, u.right.left)

    return u.right.key
}
```

→ The inorder successor will be the minimum element on the right sub tree of u.
→ Given that u.right exists.

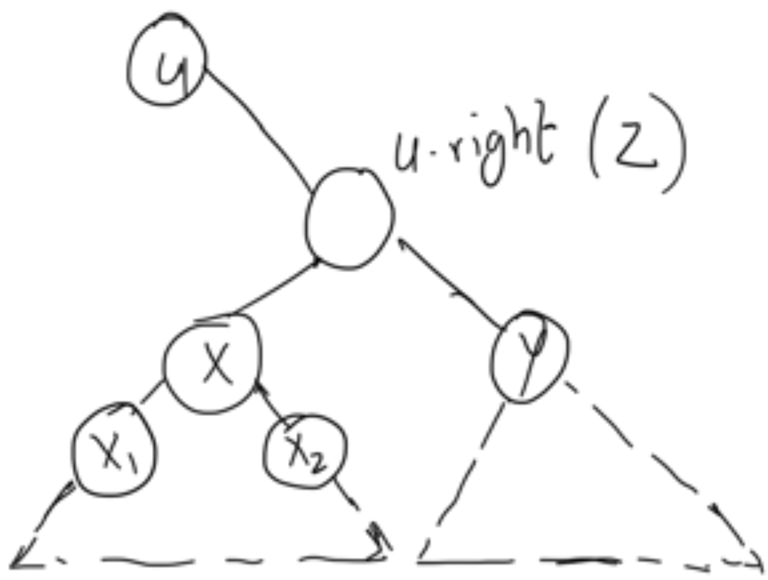
Statement: For the $i^{th}$ iteration of our while loop, the in order successor will lie in the left subtree of u.right and the distance of u.right to our in order successor will decrease by atleast one from iteration i to (i+1).

Boundary Case:
→ If there is a single element in u.right.left, left rotation will place that element at u.right and the original u.right will be the right child of this element.
→ The right child of this element is now our original right child of u. This element is the inorder successor of u as it is the smallest element on the right sub tree.

→ If the right child of u has no children, we return the value of the key as it is the successor of u by definition of inorder traversal.

Induction :

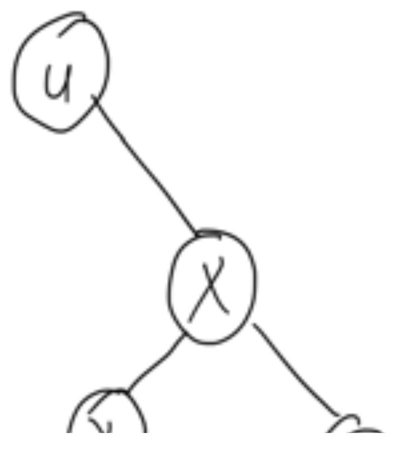
Consider the $i^{th}$ iteration configuration of T,

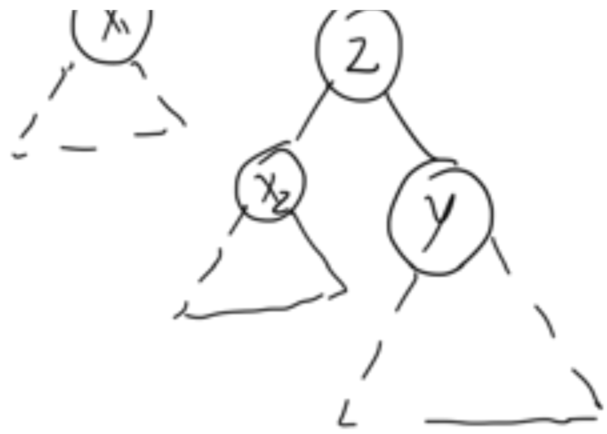



Since the inorder successor of u is the minimum element of the the right sub tree of u, it will lie in the left sub tree of u.right and will be present in subtree rooted at X.
let I be the inorder successor.

$$d(u.right, I) \le d(2, I) = d(x, I) + 1$$

After Left rotate,






We know from the last diagram,

$$\text{subtree}(X_1) < X < \text{subtree}(X_2)$$

I is the minimum element in the subtree X

$\therefore$ If subtree $(X_1)$.size $> 0$, I must lie in subtree $X_1$.

$$d(u.\text{right}, I) = d(X, I)$$

$\therefore$ Assuming that our statement is true for iteration $i$, moving from iteration $i$ to iteration $(i+1)$, we see that our statement holds. We have seen the boundary cases in the execution of our loop to be valid.

Hence our proof holds.

$T(n)$ will be proportional to the size of the depth subtree of subtree of u.right.

$\therefore$ $T(n) = O(\log n)$ if the tree is balanced and $T(n) = O(n)$ in general.

i.e.

```
# Initialize_Rotate_Search (T)
{
    R' = T.root
    Rotate_Search (R', x)

}


RotateSearch (R, x)

{

  if R.key < x :
            Rotate Search ( R.right, x)

  if R.key > x :
            Rotate Search (R.left, x)


  if R.key == x :

          while (R.p != NiL):

                  if R == R.p.left :
                      Left Rotate (T, R)
```

else:
    Right Rotate (T, R)

}

- We first recursively search for the element with key $x$. If the current node key is greater than $x$, we search in the left sub tree. If the current node key is greater than $x$, search in the right sub tree.

- Once we find element R with key $x$, we repeatedly perform the rotation of that element with respect to it's parents until the element gets to the root.

- For each iteration of the while loop, the element R will be exchanged with it's parent preserving the structure of the BST, the height of R will increase by atleast 1 as it is floated up the tree. In the final iteration, the height of R will be maximum with it's parent as nil. It will become the root node.

- The search and recursive float up using rotate will be proportional to the depth of the tree.

$\therefore$ $T(n) = O(\log n)$ if the tree is balanced and $T(n) = O(n)$ in general.

Q 2.

2.a ## misinterpreted question, solved for given tree structure T next.

# A[1, ___, n], for unsorted A. We solve for sorted A as well.

# BST T

```
Initialize_Fill_Tree (A, T)
{
    start = 1
    end = n
    T.root = node ()
    Fill_tree ( T, T.root, A, 1, n)
}

FillTree (T, u, A, start, end)
  {
    if start < end :
```

$k = \text{Partition}(A, start, end)$

$u.key = A[k]$

$\text{Fill\_tree}(T, u.right, A, k+1, end)$

$\text{Fill\_tree}(T, u.left, A, 1, k-1)$

}

- Given array $A[1, \_, n]$.

- Consider the call to fill tree.
- When we call the partition function, elements $1 - (k-1)$ are less than $A[k]$. They are allocated in the left sub tree.
- Elements $(k+1, end)$ are greater than $A[k]$, they are allocated in the right sub tree.
- Hence our BST conditions hold for the first recursive call.
- Assuming they hold for call $i$, this would mean that until the $i^{th}$ level, the BST properties hold.
- For the sub arrays corresponding to level $(i+1)$, the element chosen to be the right child of an element in level $i$, is chosen from elements greater than the element at level $i$ after partitioning.
- Similarly, for the element which is to the left of the element in level $i$. This shows us that the BST properties hold for level $(i+1)$ leaves formed.

- Hence our argument by induction is valid.
- The partition function will be executed until the leaf nodes in our tree are reached as based on the initial if statement.

The running time of this algorithm will be analogous to the running time of quicksort.

$$T(n) = O(n^2) \text{ - worst case}$$
$$T(n) = \Theta(n \log n) \text{ - average case}$$

For sorted $A[1, \_, n]$, we just select the median element for u.key

$$T(n) = 2T(n/2) + \Theta(1)$$

Using master theorem,

$$f(n) = \Theta(1), \quad n^{\log_a b} = n^{\log_2 2} = n$$

$$T(n) = \Theta(n)$$

```
FillTree (T, u, A, start, end)
  {
    if start < end:

        k = A[⌈(end - start + 1)/2⌉]

        u.key = A[k]
```

```
        Fill_tree (T, u.right , A , k+1, end)
        Fill_tree (T, u.left , A , l, k-1)
    }
```

## 2.a redefined:

Given a tree structure T, we show that there is one way to fill the array A [1,…,n] into the tree T.

If we do an inorder traversal on T

```
Initialize_fill_tree (T, A)
{
    i = 1   # start
    u = T.root

    FillTree (T, u, A)

}
```

```
Fill tree (T, u, A)
{   if (u.left != Nil):
        Fill Tree (T, u.left, A)

    explored [u] = True
        u.key = i
        i = i+1

    if (u.right != Nil):
        Fill Tree (T, u.right, A).
}
```

- The i value given to u.key is assigned after i values to elements in the left subtree are assigned.
- The i values for the left subtree are less than the i values for u.

- The i value given to u.key is assigned before the i values are assigned to elements in the right sub tree.
- The i values for the right sub tree are greater than the i values for u.
- Thus for every node u, the binary search

tree property is maintained.

- As $i$ is incremented from 1 to $n$: the assignment of $i$ values to nodes is unique.

- Suppose the assignment of $i$ values to nodes is different from the one by this algorithm.

- Then there must be a swap for atleast one of the pairs of $i$ values assigned to nodes. But such a swap will violate the binary search tree properties; if an $i$ value of a node is swapped with an $i$ value of a node in the left subtree, That node in the left sub tree will have a greater key than our original node thus violating the search tree properties. Similarly, for a swap with an $i$ value of a node in the right sub tree.

- Hence for a given tree $T$, the assignment of $A[1, \_, n]$ is unique.

2.b.

Transform Balanced $(T, n)$

{ # Initialize $T'$ as BST.

```
    A = InorderTraversal(T)
    Initialize_Fill_Tree(A, T')
    return T'
}


FillTree(T, u, A, start, end)
  {
   if start < end:
        k = A[⌈(end - start + 1)/2⌉]
        u.key = A[k]
        Fill_tree(T, u.right, A, k+1, end)
        Fill_tree(T, u.left, A, 1, k-1)
   }


InorderTraversal(T, n)
{
  # Initialize A[1, —, n]
  i = 0
  Inorder(T.root)
```

}

```
Inorder(R)
{
  if R != Nil :
      Inorder (R.left)

      A[i] = R.key
      i = i+1

      Inorder (R.right)

}
```

- We know that an inorder traversal of a BST will produce a sorted array A of elements from 1 to n.

- For a sorted array, the median index will be the correct element to use as a key of a node for recursive calls. Since the sub trees contain the same number of elements. The corresponding recursive calls will give balanced sub trees themselves.

$$T(n) = 2T(n/2) + \theta(1)$$

Using master theorem,

$$T(n) = \Theta(n)$$

$T(n) = O(n)$

2.c

```
Is Iequivalent (T, T', n)
{
    A = InorderTraversal (T, n)
    A2 = Inorder Traversal (T', n)
    flag = 1
   r = T'.root
    i = 1
    Inorder Check (r, A)
     if flag == 0:
                return False
     else:
           return True
}
   Inorder Check (u, A)

   {
```

```
if u.left != Nil :
        Inorder Check (u.left, A)

    if (u.key != A[i]) :
        flag = 0
        i = i+1

    else :
        i = i+1

if u.right != Nil :
        Inorder Check (u.right, A)

}
```

- We use the inorder traversal function from our last problem.
- A [1, —, n] will contain the inorder traversal of T.
- The inorder checking of T' will correspond to the sorted order of elements in T' being checked with elements in A.
- If T, T' are equivalent, they contain the same set of elements. ∴ The sorted order of these elements must be equal.
- ∴ $A_1, A_2$ must have the same elements in that order.

Inorder traversal takes $\theta(n)$ time.

Inorder check takes $O(n)$ time

$\therefore T(n) = \Theta(n)$

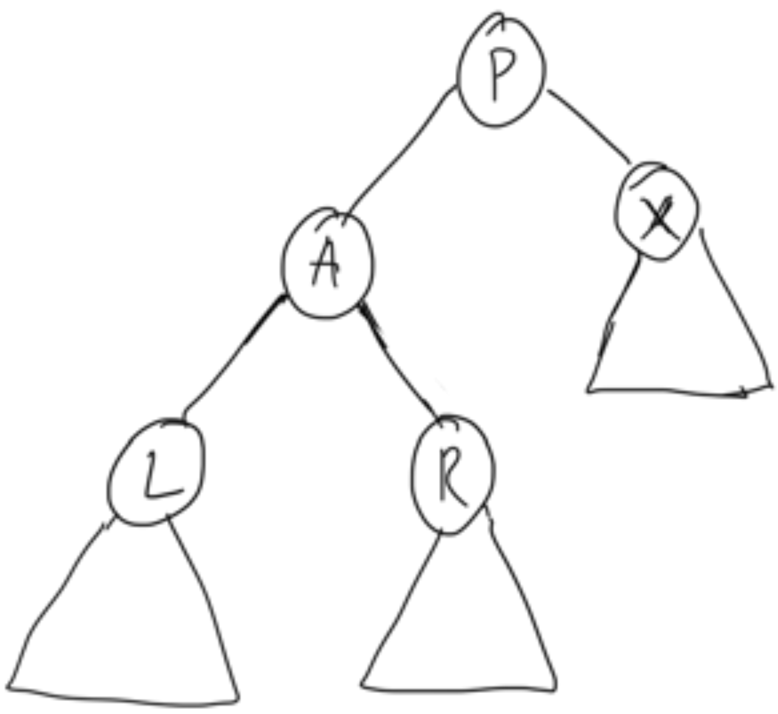
2.d.

```
Transform_init (T)
{
    R = T.root
    Transform Left (T, R)
}

Transform Left (T, A)
{
    while (A.right != Nil):
            rotate (T, A.right)

    Transform left (A.p)
```

Transform Left (A-left)

}

For some A, in a BST, consider the subtrees as follows,




For the final tree $T_2$,

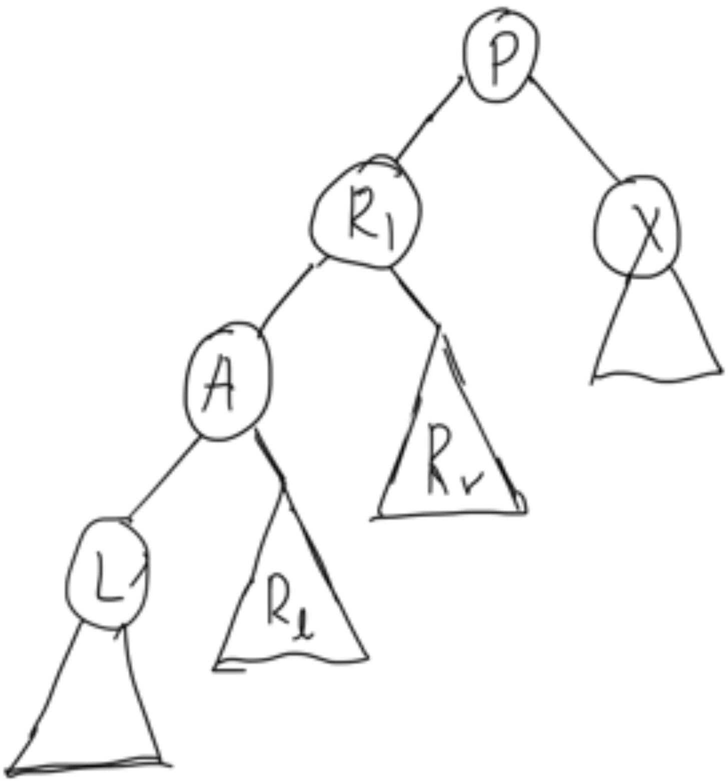
- Elements in R must be between A as a left child and P as the parent of the left child which is some element in R.

∴ A must have no right child.

- Elements in L must be below A as children.
- Elements in X must be above P.

We claim that after every call of the while loop for some node A, we put A in the correct position and the properties as required for the final 12 are maintained.

After the first right rotation,



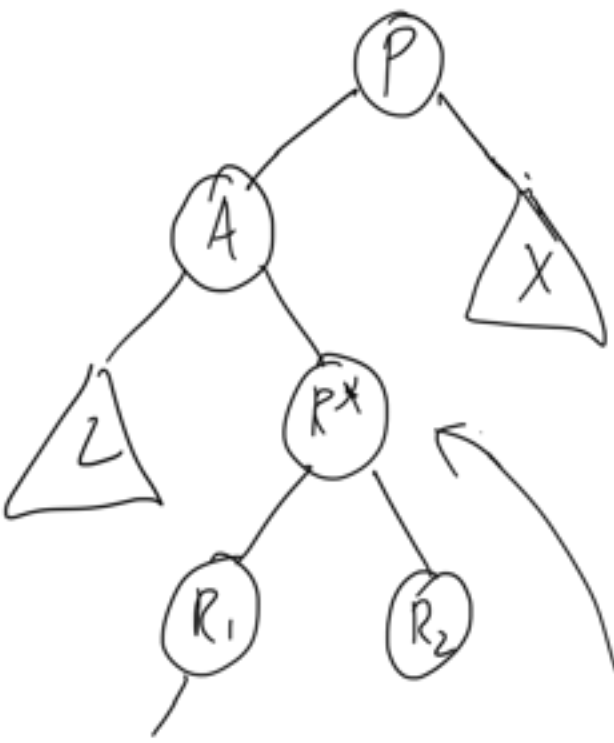

- For subsequent iterations, elements in $R_l$ will be placed through A and $R_1$ with none going between A and L. This is as per our right rotate function.

- Once A has no right children, elements above A are greater than A and elements below A are lesser than A. Thus A is at a correct position in the left sided tree.

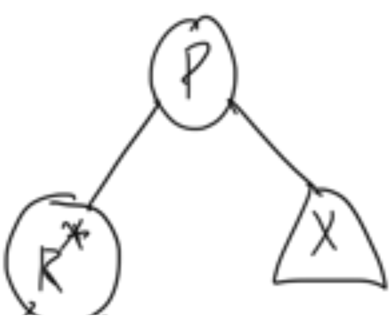
- For time complexity analysis, we show that for every while loop, the number of rotate calls is proportional to the 1 { right node } plus the number of nodes in the pre order traversal (left leaning nodes) beyond the right node.

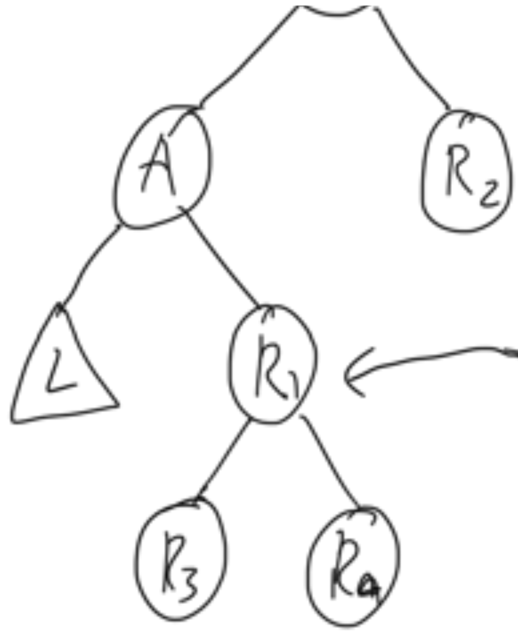
- This would mean that every element occurs uniquely in the while loop of some element. If an element occurs in the while loop of A, it will not occur in the while loop of any other element. The occurrence of this element corresponds to one rotate call.

Thus the total number of rotate calls will be proportional to the total number of elements in our BST which is n.



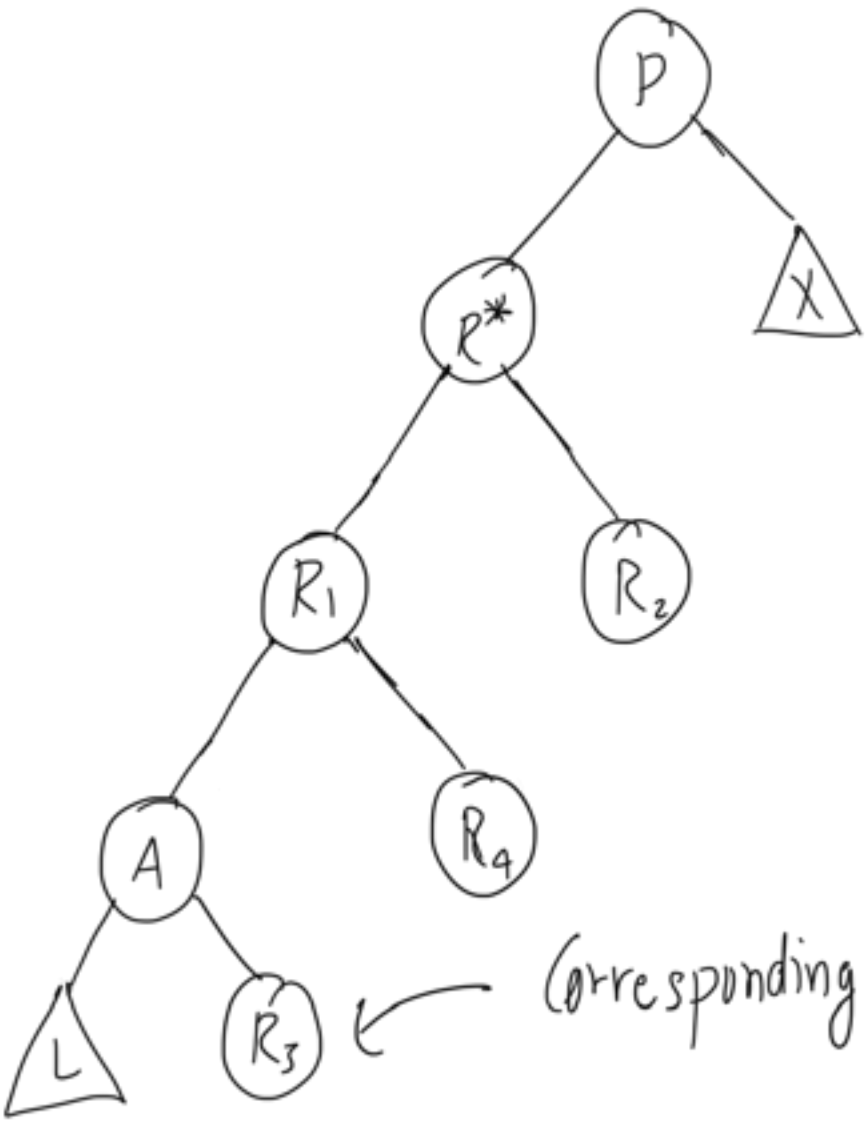

$\xrightarrow{LR}$ (Corresponding to A.right)



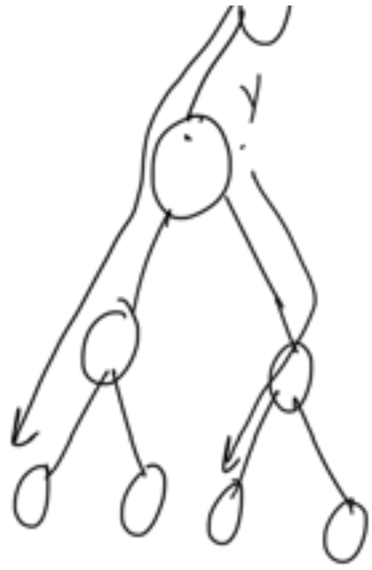



((Corresponding to A.right.left))

$\xrightarrow{LR}$




Corresponding to A.right.left.left





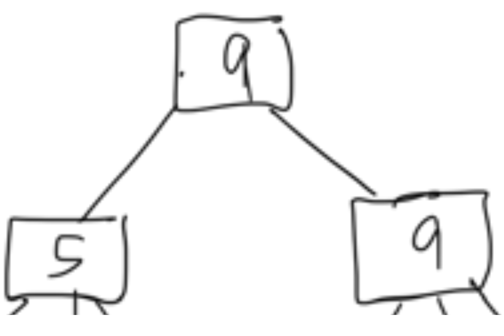
Since every node has one parent and so on. lie on a unique corresponding path. We could start with a node, trace a path along parents of left children until the child is a right child.

$$\therefore \quad T(n) = \theta(n)$$

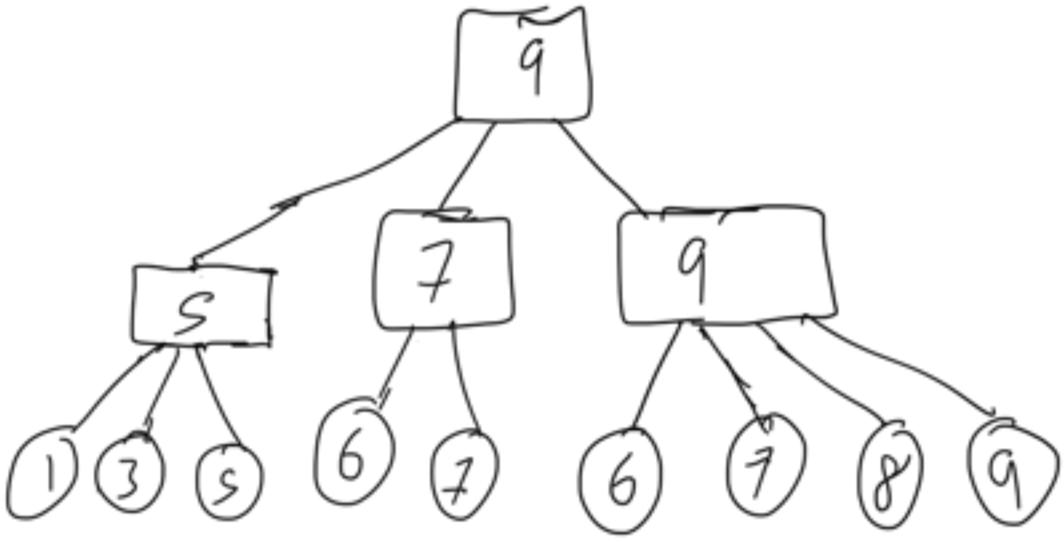
Q 3.

(a)

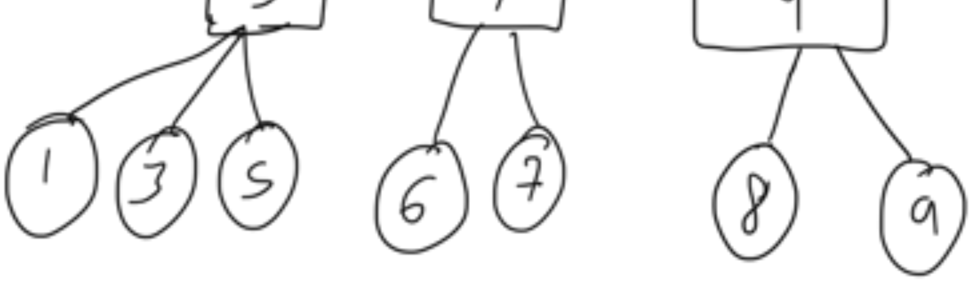
T: Given,



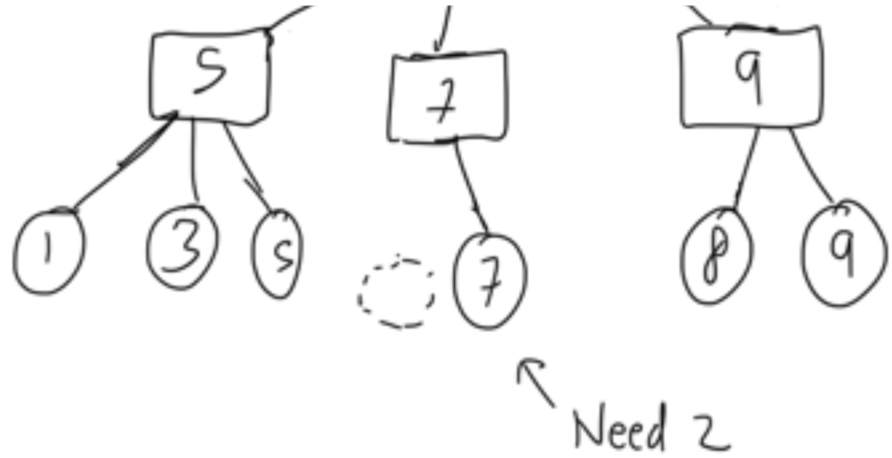
Insert (6)



split (2+2)

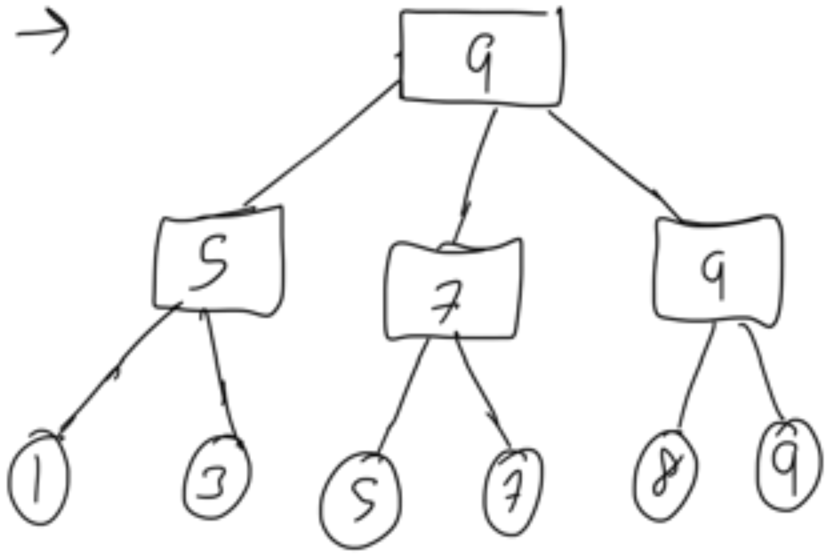


Delete (6)




Using siblings,

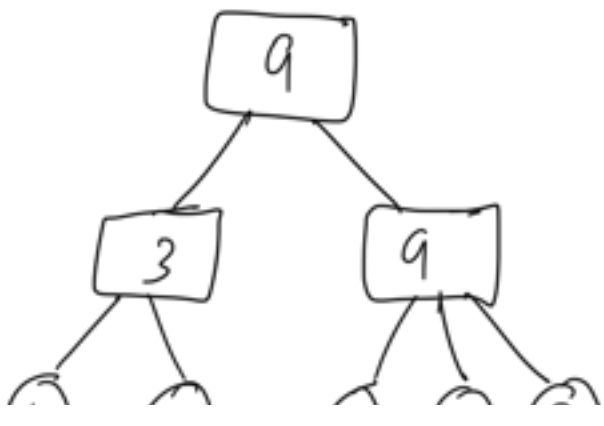
$T' \rightarrow$




$T'$ is different from $T$.

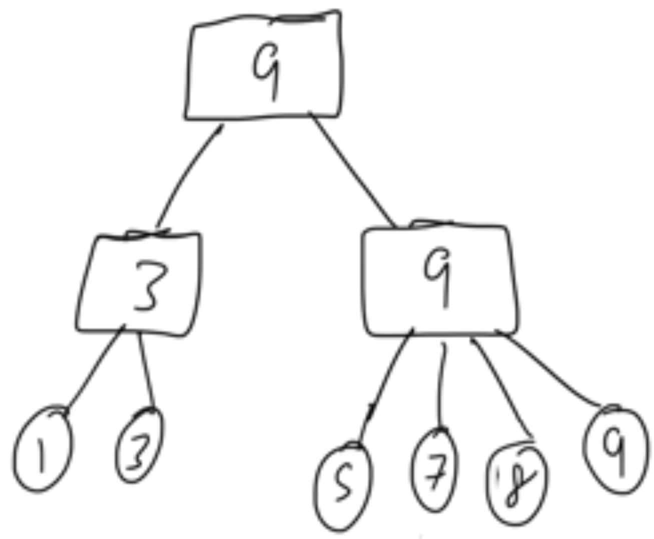
(b)

We delete (5),





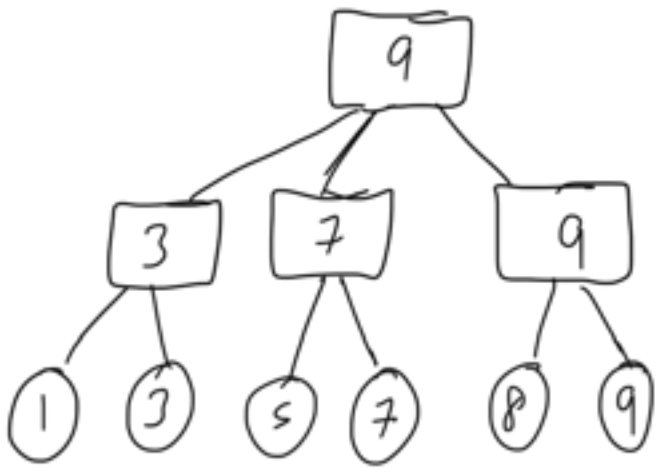
We Insert (5),





→ Split (2+2), Median goes up.

T'





We see that T' is not the same as T.

(c)

Given $S[1, \_\_, n]$

- In one pass over n, we find $O(\log n)$ distinct elements.

- Sorting the array of distinct elements, we construct a BST in $O(\log n \log(\log n))$
Since the array is sorted, Partition based recursion take $n \log n$ time. ie. $\log n(\log(\log n))$ time.
The height of this tree will be $O(\log(\log n))$ with number of terms being $O(\log n)$. The tree is balanced.

- We augment our created BST to store the number of occurences of each element in S.

- for each element we maintain a frequency count. In one pass over the array, we search for the element in S in the binary search tree and update the frequency count. Searching each element takes $O(\log(\log n))$ time and there are n such elements. This is the dominating term in our time complexity analysis. $(n \log \log n)$

- Once we build the frequency counts, we do an in order traversal and for every node, append it in our array number of times based on its frequency count. This is linear in $n$.

- We then get our sorted list.

$$\therefore T(n) = n \log(\log n)$$

(d)

- We use a 2-3 tree $T$ with the key being the book number.
- Associated with each book number, we keep data fields whose first element is the book price and second element is the book name.

For creating a book,

book.key = book_number

book.name = book_name

book.price = book_price

Insert (T, book, key = book_number)

For closing a book

Delete (T, key = book_number)

To add or subtract Price,

B = Search (T, key = book_number)

B. price = B. price + X

While insertion of books we maintain a variable max, name of most expensive book

```
B = Book()
if B.price > max :                      # O(1)
            most_expensive = B.name
Insert (T, key = book_number)  ← # O(log n)
```

Most expensive

```
{
    return most_expensive    ← θ(1)
}
```

To report the price of a given book number,

```
B = Search (T, key = book_number)

return B.price
```