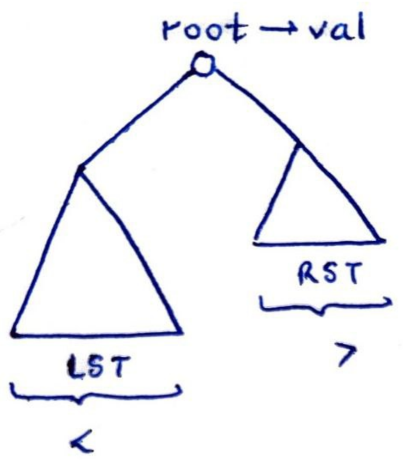
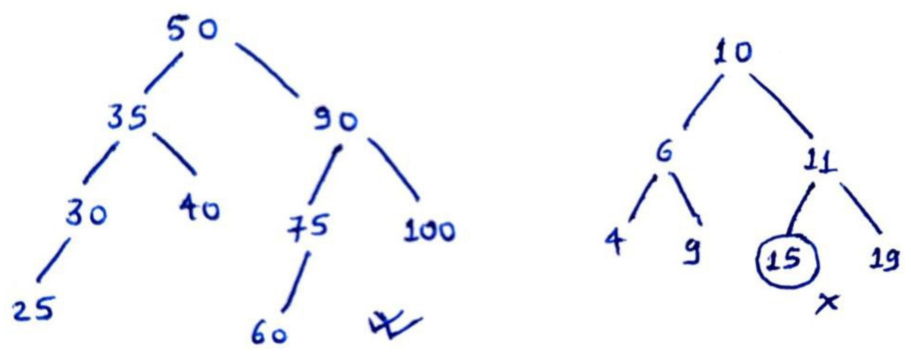
Programming Pathshala

M3 : Fundamental Data Structures

We have earlier faced the problem of random access to mid nodes while working with sorted linked lists. Binary Search tree is an attempt to solve that problem by arranging the elements of a linked list in such a way that they are sorted and the middle can be easily accessed.



Note: There is no equal sign in the binary search tree as we do not allow duplicate values by convention.



We create a height-balanced binary search tree to ease out searching, insertion and deletion.

Note: Height balanced tree - Height = log2N

Search in BST

We have been given a BST (root of the BST) and an integer K. Return the address of the node containing k otherwise return NULL.

As indicated in the diagram, there can be only three possibilities in a BST.

root->val == k //return address of the node
root->val < k //jump to RST
root->val > k //jump to LST

Iterative implementation -

```
Node* Search(Node* root, int k)

{

 Node* node = root;

 while(node!=NULL)
{
if(node->val==k) return node;

            if(node->val<k) return node=node->right;

            else return node=node->left;
}
return NULL;

}
```

Time complexity: **O(H)** //we are moving in such a way that we are following a particular root to leaf path

Space complexity: **O(1)**

Insert in BST

Given a BST with nodes containing integer values and an integer k. Insert a node with (val == k). Return the updated BST.

Approach:

One way is to keep k as the root and put all nodes with val<k in LST and val>k in RST. But there will be a lot of reordering in this case.

Since we know that while searching for an element that is not present in a BST, we ultimately end up reaching NULL. This NULL is actually the place for the key k.
Time complexity: **O(H)** //since we are taking a particular root to leaf path

Note:
It is important to remember the address of the parent as we have to attach the node to it.
Edge case: Input BST is empty.
=> Create a new node and return it.

Iterative Implementation -

```
Node* Insert(Node* root, int k){
if(!root) return new Node(k, NULL, NULL);

  Node* par = NULL;

  Node* node = root;

  while(node){

     par = node;
if(k>node->val)
node=node->right;
else
node=node->left;
}
if(k>par->val) par->right = new Node(k, NULL, NULL);
else par->left = new Node(k, NULL, NULL);
return root;
}
```

Time complexity: **O(H)**
Space complexity: **O(1)**

Delete in BST

Given a BST and an integer k. Delete the node containing value k and return the updated BST.

How many ways can a node value be deleted? Is there only a single possibility or more?

Approach:

Brute Force - Store all the values of the BST in an array, remove k and create a new BST from the remaining values. It is not an efficient solution as it involves unnecessary operations.

Which types of nodes are easy to delete?
- Node with no children - leaf nodes
- Node with a single children

  To which part of the parent, should we attach the child of the deleted node?
  Check the child of the deleted node. If it is the right child, it should be attached to the right, otherwise to the left.
- Node with both the children
  - Here, if we have to delete 35, then we have to bring some other node at its place. Now, it can either be from LST or RST of the node.
  - If we bring a node from LST, then it has to be certainly larger than all the nodes in its left. Therefore, we can bring the largest node from the LST.
  - If we bring a node from RST, then it has to be certainly smaller than the nodes in its right. Therefore, we can bring the smallest node from the RST.
  - Largest node of LST = rightmost (also called inorder predecessor)
  - Smallest node of RST = leftmost
  - These nodes can either be a leaf or a single child node. In this lecture, we will bring the largest node of LST.

Special case - Node to be deleted is the root itself.
Methods required -
- getNodeAndParent
- isLeaf
- deleteLeaf
- hasSingleChild
- deleteSingleChildNode
- deleteDoubleChildNode
- deleteRoot

Time complexity: **O(H)**
- Figure out the node to be deleted: **O(H)**
- Find maximum in LST: **O(H)**
- Constant time pointer operations: **O(1)**

Note: Do not forget to delete the node from the memory using delete keyword.

Implementing Delete in BST

In this lecture, we will look at the implementation of all the methods discussed in the previous lecture to delete a node from a BST.

//To search the node to be deleted and its parent in the BST

```
pair<Node*, Node*> getNodeAndParent(Node* root, Int key){

Node* node = root;

Node* par = NULL;

while(node){

if(node->val==key)

break;

par = node;

if(key>node->val)

node=node->right;

else

node=node->left;

}
return {node, par};

}
```

//To check if a node is a leaf node

```
bool isLeaf(Node* node){

return(!node->left && !node->right);

}
```

//To delete a leaf node

```
Node* deleteLeaf(Node* node,  Node* par){

if(par->left==NULL)

par->left=NULL;

else

par->right=NULL;

return node;

}
```

//To check if a node is a single child node

bool hasSingleChild(Node* node){

return (node->left && !node->right) || (!node->left && node->right);

}

//To delete a single child node

```
Node* deleteSingleChildNode(Node* node, Node* par){

if(node->left){

if(par->right==node)

par->right=node->left;

else

par->left=node->left;

node->left=NULL;

}

else{

if(par->right==node)
par->right=node->right;

else

par->left=node->right;

node->right=NULL;

}

return node;

}
```

//To delete a double child node

```
Node* deleteDoubleChildNode(Node* node, Node* par){

Node* parent = node;

Node* pred = node->right; //Maximum in LST = Inorder predecessor

while(pred->right){

parent=pred;

pred=pred->right;

}

Node* n1;

if(isLeaf(pred))

n1=deleteLeaf(pred, parent);

else

n1=deleteSingleChildNode(pred, parent);

if(par && par->left==node) //checking par==NULL incase node to be deleted is root

par->left=n1;

else if(par)

par->right=n1;

n1->left=node->left;

n1->right=node->right;

return node;

}
```

//To delete root node

```
Node* deleteRoot(Node* root){

if(isLeaf(root)){

delete(root);

return NULL;
}

if(hasSingleChild(root)){

Node* ans = root->left?root->left:root->right;

delete(root);

return ans;

}

Node* ans = root->left;

while(ans->right){

ans-ans->right;

Node* node = deleteDoubleChildNode(root, NULL);

delete(node);

}

return ans;

}
```

//Driver method

```
Node* deleteNodeBST(Node* root, int key){

P = getNodeAndParent(root, key);

if(!P.first) //Node to be deleted doesn't exist

return root;

if(P.first==root)

return deleteRoot(root);

Node* node;

if(isLeaf(P.first))

node=deleteLeaf(P.first, P.second);

else if(hasSingleChild(P.first))

node=deleteSingleChildNode(P.first, P.second);

else

node=deleteDoubleChildNode(P.first, P.second);

delete(node);

return root;

}
```

K-th smallest Node in BST

Given a BST and an integer K. Find the Kth smallest value in BST. (K is one-indexed)

<>

Approach:

Brute Force - Traverse the tree to store the node values in an array. Sort the array to find the K-th smallest node in the BST.
<> preorder sorting
Time complexity: O(NlogN) //major time goes into sorting
Space complexity: O(N)
Can we optimize on time?
In the previous approach, we traversed the in a random manner. Can we do it in a better way?
Inorder traversal: LST | root->val | RST
We can use inorder traversal as it traverses the node values of a BST in a sorted fashion.
4, 6, 7, 9, 10, 15, 20
Time complexity: O(N)
Space complexity: O(N)

Can we optimize on space?
Considering the fact that inorder traversal takes nodes one after the another. Do we really need an array?
We can create a counter variable to count the order of the node value.
<>

```
int getKthVal(Node* root, int k){
int cnt=1, ans;
traverse(root, k, cnt, ans);
return ans;
}

void traverse(Node* root, int& k, int& cnt, int& ans){
if(!root) return;
traverse(root->left, k, cnt, ans);
if(cnt++==k){
ans=root->val;
return;
}
traverse(root->right, k, cnt, ans);
}
```

Time complexity: O(N)
Space complexity: O(H) //size of call stack

Check if a Binary Tree is BST - 1

Given a Binary Tree. Check if a Binary Search Tree.

<>

Approach:

For a given BST, its inorder traversal is sorted.
Its corollary - "If the inorder traversal is sorted then the binary tree is a BST" is also true.
Therefore, we can store the inorder traversal of the binary tree in an array and check if it is sorted or not.
<>
Time complexity: O(N)
Space complexity: O(N)

Can we do away with the use of arrays?
Yes, since we only need the previous and the current value for comparison.
<>

```
bool isBST(Node* root){
long prev = LONG_MIN;
bool ans = true;
traverse(root, prev, ans);
return ans;
}

void traverse(Node* root, long& prev, bool& ans){
if(!root) return;
traverse(root->left, prev, ans_;
if(prev<root->val)
 prev=root->val;
else{
 ans=false;
 return;
}
traverse(root->right, prev, ans);
}
```

Time complexity: O(N)
Space complexity: O(H)

Check if a Binary Tree is BST - 2

Given a binary tree check if it is a binary search tree without using inorder traversal.

Approach:

Will it work if we check the left and right child at every node?
node->val < node->right->val && node->val > node->left->val
No, it will not work
<>

The definition of binary search tree is not for left and right child but for left and right subtree. Thus, we have to check the LST and RST for each node.

But it will make it O(N2) time complexity as we have to individually check for each node. Can it be improved?
In such cases, we use the principle of returning from bottom to top. We can return the minimum and maximum value of nodes from bottom to top for each subtree.
<>
maxm = max(max1, max2, root->val)
minm = min(min1, min2, root->val)

<>
We can use a global flag variable (default value = true) and turn it to false as we encounter that the given tree is not a binary search tree.
Can we do it without using a global variable?
Along with the maximum and minimum node value from each subtree we can also return an isBST flag variable indicating if that subtree is a binary search tree or not.
<>
isBST = isBST1 && isBST2 && max1 < root->val && root->val < min2
maxm = max(root->val, max1, max2)
minm = min(root->val, min1, min2)

Termination condition: When we hit NULL, we need to return [true, LONG_MAX, LONG_MIN]

Note: Use LONG_MAX and LONG_MIN instead of INT_MAX and INT_MIN.

<>

```
pair<pair<long, long>, bool> check(Node* root){

 if(!root) return {{LONG_MAX, LONG_MIN}, true};
left = check(root->left);
right = check(root->right);
bool flag = left.second && right.second && root->val>left.first.second &&
right->val<right.first.first;
long minm = min({(long) root->val, left.first.first, right.first.first});
long maxm = max({(long) root->val, left.first.second, right.first.second});
return {{minm, maxm}, flag};

}

bool checkBST(Node* root){

P = check(root);return P.second;
}
```

Time complexity: **O(N)**
Space complexity: **O(H)**

Check if a Binary Tree is BST - 3

In this lecture we will learn a simple technique to check if a binary tree is a BST or not.

Approach:

Suppose we have to calculate the possible range of the different node values in a binary search tree. For eg.
<>
<>
<>
<>
<>
From the above examples we infer that if the range of root->val is (lb, up) and if we move towards left then it becomes (lb, ub') and (lb', ub) when we move towards the right.
Can we use this concept to check if a binary tree is BST or not?
<>

```
void check(Node* node, long lb, long ub, bool &ans){
if(!node) return;
if(node->val<=lb || node->val>=ub){
ans=false;
return;
}
check(node->left, lb, node->val, ans);
check(node->right, node->val, ub, ans);
}
```

Time complexity: **O(N)**
Space complexity: **O(H)**

Creating BST from Sorted Array

Given a sorted array int Arr[N]. Create a BST of the smallest height.

Note: Binary search tree of smallest height is also a Height balanced tree.

Approach:

Will height depend on root?
<>
Yes, it does. Thus we should choose the root closest to the center.
<>
Can the problem have multiple solutions? Yes, as evident from the example below.
<>
We can divide the array from mid each time to create a height balanced binary search tree.
<>

Note: It is preferable to terminate at NULL instead of terminating at the leaf edges.

```
Node* makeBST(vector<int>& nums, int i, int j){


 if(i>j) return NULL;
int m = (i+j)/2;
Node* root = new Node(nums[m],. NULL, NULL);
root->left = makeBST(nums, i, m-1);
root->right = makeBST(nums, m+1, j);
return root;


}
```

Time complexity: **O(N)**
Space complexity: **O(logN)** //In every iteration, the size of array is reduced to half.

Largest BST in a Binary Tree

Given a binary tree, find the size of the largest BST in it.

<>

Approach:

The basic brute force approach is to check the subtree starting at each node and compare the subtree length with the answer variable if it is a BST.

But this approach is of O(N2) time complexity. Can we optimize it?
Can we take help from the technique we previously used for checking if a given binary tree is BST or not?

We can use the principle of returning from bottom to top and return the isBST flag, max and min value of each subtree and `the node count in that subtree.`

```
[isBST, minm, maxm, cnt]
<>
Using this info -
- We can check if the subtree is BST or not
- Calculate the count of nodes (1+c1+c2) in that BST
- Find the answer to the largest BST in the binary tree = max(ans, 1+c1+c2)

<>

struct SubTreeDetails{
bool isBST;
long minm;
long maxm;
long cntNodes;
SubTreeDetails(bool flag, long minVal, long maxVal, long cnt){
isBST = flag;
minm = minVal;
maxm = maxVal;
cntNodes = cnt;
}
}

SubTreeDetails getLargestBST(Node* root, int& ans){
if(!root)
return newSubTreeDetails(true, LONG_MIN, LONG_MAX, 0);
SubTreeDetails* lst = getLargestBST(root->left, ans);
SubTreeDetails* rst = getLargestBST(root->right, ans);
bool isBST = false;
long cnt = 1+ lst->cntNodes + rst->cntNodes;
long minm = min({(long)root->val, lst->minm, rst->minm});
long maxm = max({(long)root->val, lst->maxm, rst->maxm});
if(lst->isBST && rst->isBST && root->val>lst->maxm && root->val<root->minm){
ans = max((long)ans, cnt);
isBST = true;
}
return new SubTreeDetails(isBST, minm, maxm, cnt);
}
```

Time complexity: **O(N)**
Space complexity: **O(H)**

Reconstructing a BST

Given the preorder traversal of a binary search tree, we have to construct the BST.

Approach:

In preorder traversal, roots come first, followed by its left subtree and then right subtree. Since it is a BST, the value of all the children in the left subtree will be less than the root and greater in the right subtree. We can use this fact to create BST.

Brute Force:

We first construct the root. Then we find the index of the first element, which is greater than the root. Let the index be 'i'. The values between root and 'i' will be part of the left subtree, and the values between 'i'(inclusive) and 'n-1' will be part of the right subtree. Divide given pre[] at index "i" and recur for left and right sub-trees.

Time Complexity: **O(N*N)**

Space Complexity: **O(N)**

Optimal Approach:

We can precompute NGE, i.e., the next greater element for every element in our array. If we already know the next greater element for the root, then we can create a division in O(1) time. Rest everything will remain the same.

```
TreeNode* makeBST(vector<int>& pre,int i, int j, vector<int>&NGE){

     if(i>j) return NULL;

     TreeNode* root= new Treenode(pre[i],NULL,NULL);

     root->left=makeBST(pre,i+1,NGE[i]-1,NGE);

     root->right=makeBST(pre,NGE[i],j,NGE);

     return root;

 }
```

Time Complexity: **O(N)**
Space Complexity: **O(N)**

BST Iterator - 1

Implement the BSTIterator class that represents an iterator over the in-order traversal of a binary search tree (BST):

BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.
boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false.
int next() returns the number at the pointer, then Moves the pointer to the right.

```
     class BSTIterator {

BSTIterator(TreeNode* root) {



}



int next() {
    }

    bool hasNext() {

    }
};
```

Approach:

The very first thing we need is inorder traversal of the tree. So when BSTIterator function is called, we will do inorder traversal and store values in a vector.

next() gives the current element and moves the pointer to the next element. We can maintain a pointer “curr” that will point to the current index in array. So when next() is called, we can just increase our pointer by 1 and return the element at index curr-1.

hasNext() checks if there is any next element in the inorder traversal or not. If curr is equal to the size of the array that means there doesn’t exist any next element else next element is present.

```
class BSTIterator {
    vector<int> nodes;
    int curr = 0;

    void inOrder(TreeNode* root) {
        if (!root)
            return;
        inOrder(root->left);
        nodes.push_back(root->val);
        inOrder(root->right);
    }
    BSTIterator(TreeNode* root) {
        inOrder(root);
    }

    int next() {
        curr++;
        return nodes[curr-1];
    }

    bool hasNext() {
        return curr != nodes.size();
    }
};
```

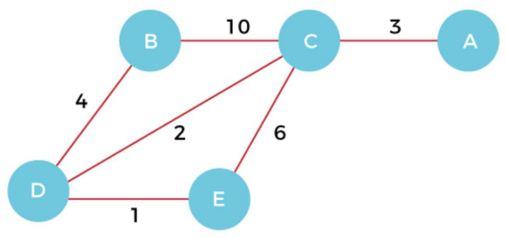
Time Complexity: **O(N)**

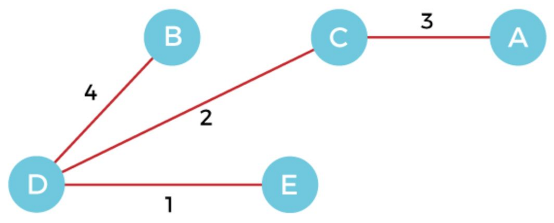
Space Complexity: **O(N)**

BST Iterator - 2

Optimized approach:

In the previous approach, we were using an array to store all the nodes. So the idea is to avoid storing all the nodes at once. Here we will use a stack. Basically, we will do in-order traversal Iteratively instead of recursively. The stack keeps track of at most the next "h" (height of tree) Elements for "next()" calls, and the top of the stack is the current minimum element. At every "next()" call, we need to refresh the stack by populating the stack with all the left nodes up to the leaf, starting from the right node of the current minimum node. Let's understand by an example:





We will push the left part of the tree into a stack which is 7, and then 3 is inserted. Now is next() is called 3 is stored as the top and then poped now we will push the right of 3, but it does not contain any, so we just return the top->val, i.e., 3. We will now pop() 7 from the stack and see if it has the right children. Yes, it has, so we push 15 and then 9 into the stack. Observe here that a time stack contains elements equal to the tree's height. Now 9 is poped, and 9 does contain any right child, so we move on to 15. If hasNext() is called, it would return true as traversal is still left, and the stack is also not empty. Now 15 is poped and checked if it has the right child; yes, it has, i.e., 20, so 20 is pushed inside the stack. Lastly, 20 is poped and returned; after that bool hasNext() is called, it will return false as there are no more elements or children inside the stack.

```
class BSTIterator {

    stack<TreeNode*> st;



    BSTIterator(TreeNode* root) {

        st.push(root);



        while(root->left)

        {

            st.push(root->left);

            root=root->left;

        }

    }



    int next() {

        TreeNode *root=st.top();

        st.pop();
        int ans=root->val;


        if(root->right)

        {

            root=root->right;

            st.push(root);

            while(root->left)

            {

                st.push(root->left);

                root=root->left;

            }

        }


        return ans;

    }


    bool hasNext() {

        return !st.empty();

    }

};
```

Time Complexity: **O(N)**

Space Complexity: **O(H)**

Pair Sum in BST

Given a Balanced Binary Search Tree and a target sum, check if there is a pair with a sum equal to the target sum or not.

Approach:

We can simply do an inorder traversal that will give nodes in sorted form. Then we can use two pointers approach to find the target sum in an array.

Time Complexity: **O(N)**

Space Complexity: **O(N)**

Space Optimized Approach:

The idea is first in place to convert BST to Doubly Linked List (DLL), then find the pair in sorted DLL in O(n) time. This solution takes O(n) time and O(Logn) extra space but modifies the given BST.

We can think of iterative inorder again. Iterative inorder will keep track of the next h(h is the height) elements, and the top element of the stack will be the least element. Can we use this element as a left pointer of two pointers approach in a sorted array? Yes, we can, but we need right pointer as well. So let's think of a traversal, the reverse of inorder traversal. It first prints the right subtree, then the root, and at last the left subtree. It will give nodes in decreasing order and we can use it as a right pointer of two pointers approach in a sorted array.

We will take two stacks, one for inorder traversal and the other for the reverse of inorder traversal. Suppose the top element of the first stack is x, and that of the second stack is y. If x+y>target, that means we need to decrease our right pointer, and if x+y<target, that means we need to increase our left pointer. In case they are equal, we found a pair equal to the target sum.

```
    bool findTarget(TreeNode* root, int target) {

        stack<TreeNode*>s1,s2;
        TreeNode* node= root;

        while(root){

            s1.push(root);

            root=root->left;

        }

        while(node){

            s2.push(node);

            node=node->right;

        }

        while(s1.top()->val != s2.top()->val){

            int x=s1.top()->val, y=s2.top()->val;

            if(x+y==target) return true;

            else if(x+y>target){

                node=s2.top()->left;

                s2.pop();

                while(node){

                    s2.push(node);

                    node=node->right;

                }

            }

            else{

                node=s1.top()->right;

                s1.pop();

                while(node){

                    s1.push(node);

                    node=node->left;

                }

            }

        }

        return false;

    }
```

Time Complexity: **O(N)**

Space Complexity: **O(H)**

LCA in BST

Given a Binary Search Tree and two values, n1 and n2, find the Lowest Common Ancestor (LCA).

Approach:

For the Binary search tree, while traversing the tree from top to bottom, the first node which lies in between the two numbers n1 and n2 is the LCA of the nodes, i.e., the first node n with the lowest depth which lies in between n1 and n2 (n1<=n<=n2) n1 < n2. So just recursively traverse the BST in; if the node's value is greater than both n1 and n2, then our LCA lies on the left side of the node; if it is smaller than both n1 and n2, then LCA lies on the right side. Otherwise, the root is LCA (assuming that both n1 and n2 are present in BST).

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {

    if(!root) return NULL;

    if ((root -> val > p -> val) && (root -> val > q -> val)) {

        return lowestCommonAncestor(root -> left, p, q);

    }

    if ((root -> val < p -> val) && (root -> val < q -> val)) {

        return lowestCommonAncestor(root -> right, p, q);

    }

    return root;

}
```

Time Complexity: **O(H)**
Space Complexity: **O(H)**

BST Recovery - 1

You are given a binary search tree (BST), where the values of exactly two nodes of the tree were swapped. Recover the tree without changing its structure.

Brute Force:

The inorder traversal of a BST produces a sorted array. So a simple method is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiliary array elements back into the BST, keeping the structure of the BST same.

Time Complexity: **O(N*logN)**

Space Complexity: **O(N)**

Optimized Approach:

We can solve this in O(n) time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

The swapped nodes are not adjacent in the inorder traversal of the BST.
For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5
If we observe carefully, during inorder traversal, we find node 7 is smaller than the previously visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (the current node). Finally, swap the two node's values.

The swapped nodes are adjacent in the inorder traversal of BST.
For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}. The inorder traversal of the given tree is 3 5 8 7 10 15 20 25. Unlike case 1, here, only one point exists where a node value is smaller than the previous node value. e.g., node 7 is smaller than node 8.

How to Solve?

We will maintain three-pointers, first, middle, last, and an integer variable count. When we find the first point where the current node value is smaller than the previous node value, we update the first with the previous node & the middle with the current node and increase the count to one. When we find the second point where the current node value is smaller than the previous node value, we update the last with the current node and increase the count to two. In the case of 2, the count will be 1. So, the last pointer will not be updated.

```
  void inorder(TreeNode* root, vector<TreeNode*>&nodes){

        if(!root) return;

        inorder(root->left,nodes);

        nodes.push_back(root);

        inorder(root->right,nodes);

    }

    void recoverTree(TreeNode* root) {
        vector<TreeNode*>nodes;

        inorder(root, nodes);

        TreeNode *n1=NULL, *n2=NULL, *n3=NULL;

        int count=0;

        for(int i=0;i<nodes.size()-1;i++){

            if(nodes[i]->val>nodes[i+1]->val){

                if(count==0){

                    count=1;

                    n1=nodes[i];

                    n2=nodes[i+1];

                }

                else{

                    count=2;

                    n3=nodes[i+1];

                    break;

                }

            }

        }

        if(count==1){

            int temp=n1->val;

            n1->val=n2->val;

            n2->val=temp;

        }

        else{

            int temp=n1->val;

            n1->val=n3->val;

            n3->val=temp;

        }

    }
```

Time Complexity: **O(N)**

Space Complexity: **O(N)**

BST Recovery - 2

Space Optimized Approach:

If you look closely, we were considering only adjacent pairs in the nodes array, which stores inorder traversal. So instead of storing all the nodes, we can limit our search to just two nodes. So we can store the predecessor of the current node. If the current node is smaller than its predecessor, then there is a swap at this position. Again we can have a count variable that will check if it has happened for the first time or the second time to cover two possible cases.

```
    TreeNode *n1,*n2,*n3,*prev=NULL;

    int count=0;
void inorder(TreeNode* root){

    if(!root) return;

    inorder(root->left);

    if(prev!=NULL && prev->val > root->val){

        if(count==0){

            n1=prev;

            n2=root;

            count =1;

        }

        else{

            count =2;

            n3=root;

        }

    }

    prev=root;

    inorder(root->right);



}

void recoverTree(TreeNode* root) {

    inorder(root);

    if(count==1){

        int temp=n1->val;

        n1->val=n2->val;

        n2->val=temp;

    }

    else{

        int temp=n1->val;

        n1->val=n3->val;

        n3->val=temp;

    }

}
```

Time Complexity: **O(N)**

Space Complexity: **O(1)**