Binary tree: (Tree is also a special type of graph)

Depth of a Node: The length of a path from root to particular node is depth of a node ------

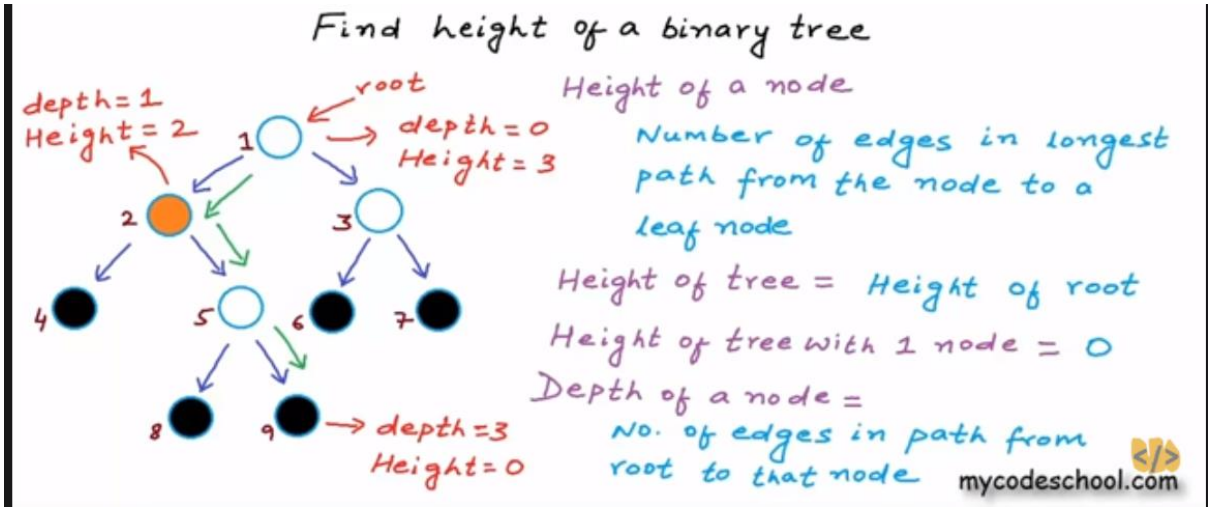
Max. Depth of a tree = Height of Binary tree = Height of root

Min. Depth:

```
int findmin(TreeNode* A){
   if(A==NULL) return INT_MAX;
   if(A->left==NULL && A->right==NULL) return 1;
   return 1+min(findmin(A->left), findmin(A->right));
```

Height of a node: length of a path from node to farthest leaf rooted from node.
[Height of a tree= height of a root]

- **Height of leaf**=0 or 1
- Height of NULL=-1 or 0




```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

int FindHeight(struct Node *root) {
    if(root == NULL)
        return -1;
    return max(FindHeight(root->left),FindHeight(root->right)) +1;
}
```

Proper binary tree: Child of any node are '0' or '2' only.

Complete binary:

- Each level is completely filled except possibly the last level(last level aligned to left).
- Tree of **N** node has floor[log2(N)] height.

Perfect binary: All levels are completely filled.

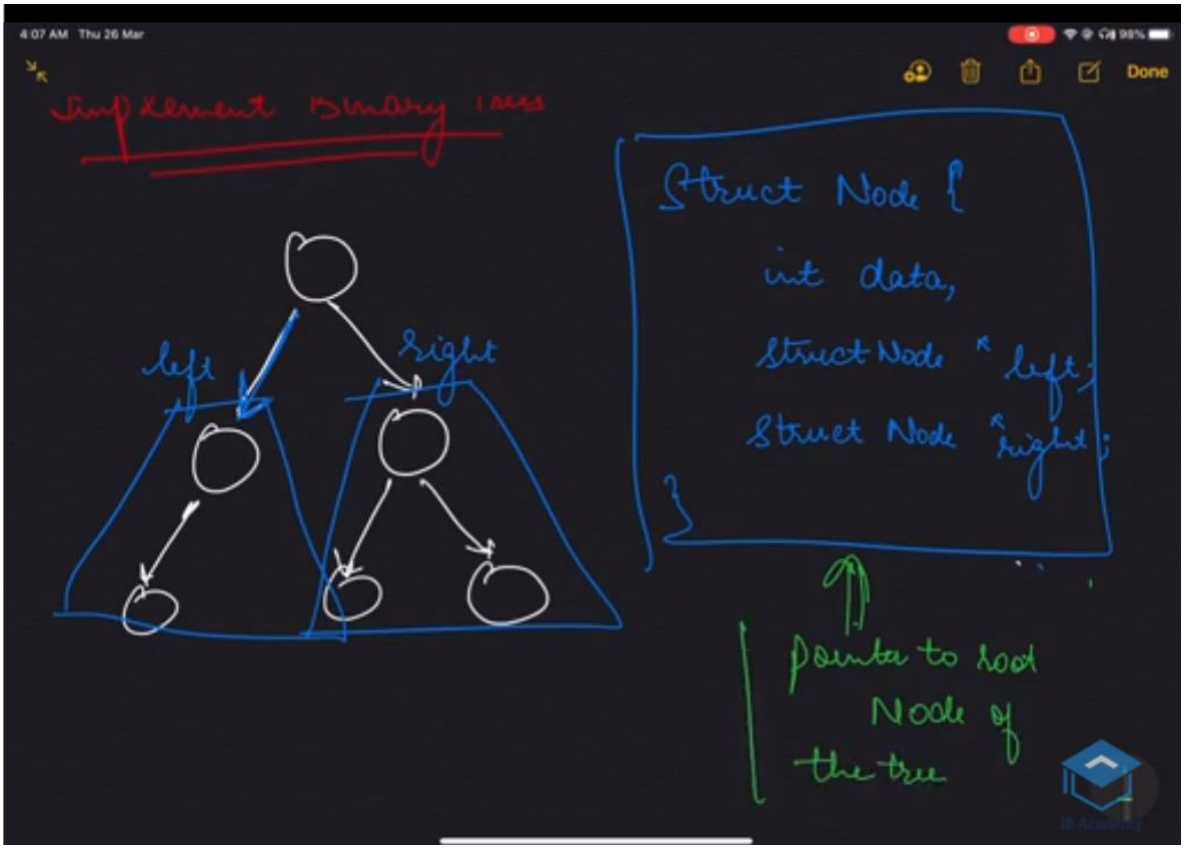
- tree of height h has **2^(h+1)-1** nodes. [0,1,2....h levels]==[2^0+2^1+2^2+....2^h] nodes.
- Tree of N node has **log2(N+1)** height

Balanced binary Tree(height balanced) Diff of heights of left and right subtree is not more than 1.

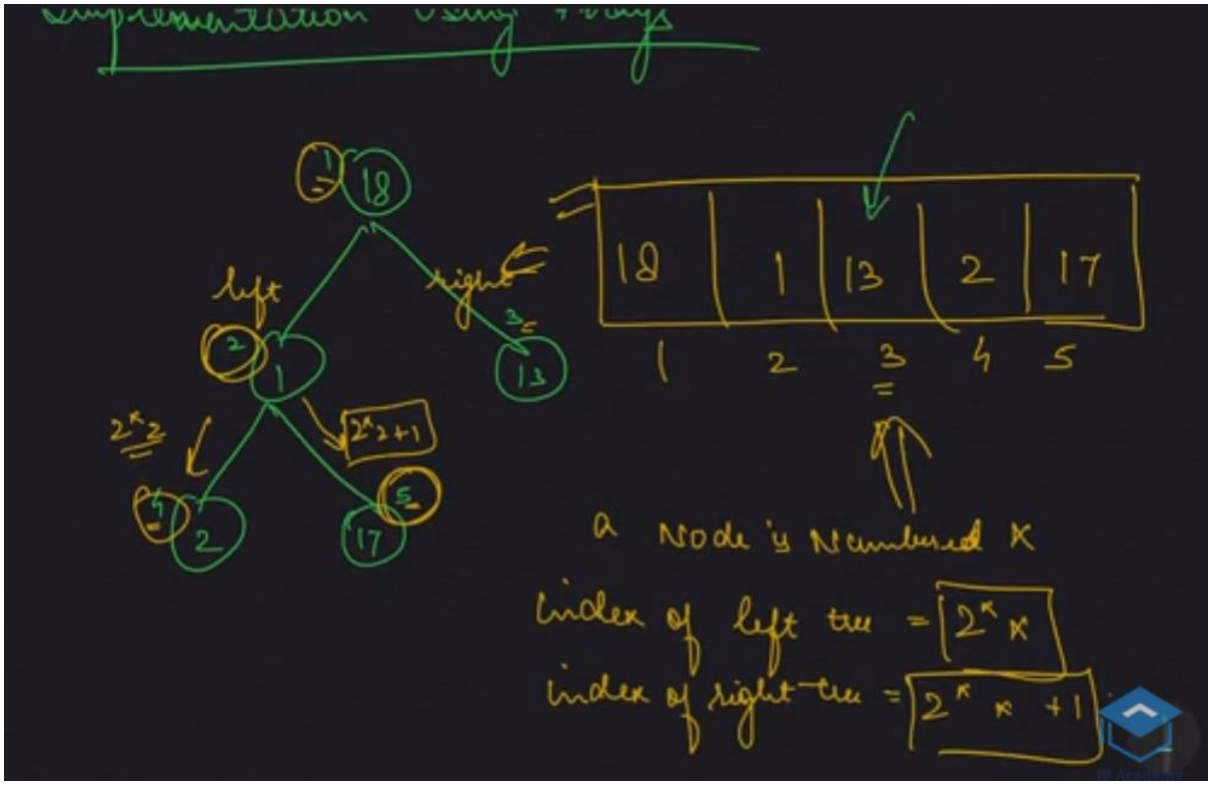
==========

TREE:

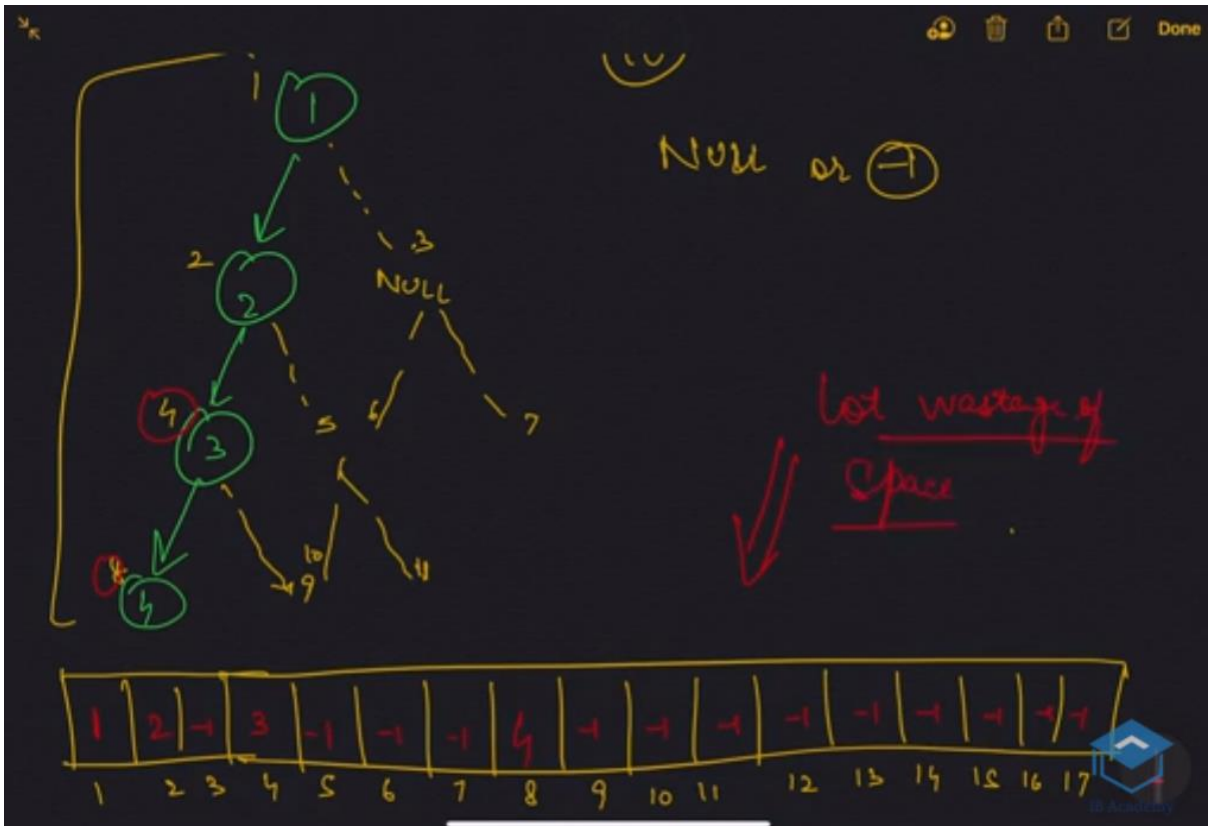
Using Linked list:



Using Arrays:



Array implementation is only used for balanced binary trees. Otherwise will consume lot of space
Eg. Skewed binary tree(all nodes act as linked list)



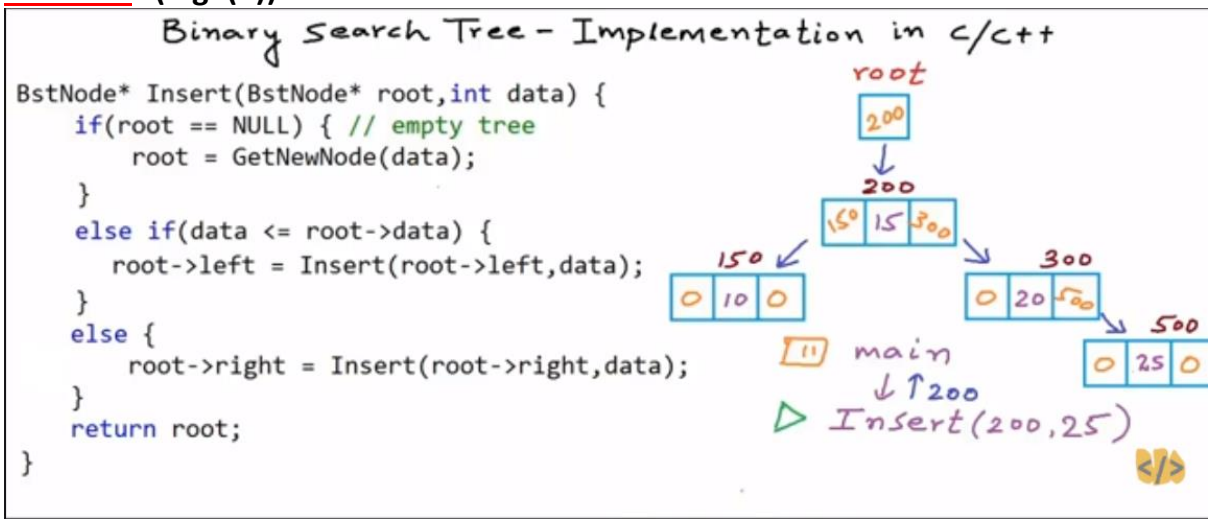
==========

BINARY Search tree:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

INSERT: O(log2(n))



SEARCH: O(log2(n))

```
TreeNode* search(TreeNode* A, int B){
    if(A==NULL) return NULL;

    if(A->val==B) return A;
    else if(A->val>B){
        return search(A->left,B);
    }
    else{
        return search(A->right,B);
    }
}
```

Function to find Node with min. Value in tree: O(log2(n))

```
//Function to find Node with minimum value in a BST
struct Node* FindMin(struct Node* root) {
    if(root == NULL) return NULL;
    while(root->left != NULL)
        root = root->left;
    return root;
}
```

==========

TRAVERSAL:

- **Breadth-first (**Level-order**)**: visit level wise, for any node we 1st visit its children than grand children.
    - Use queue for storing left and right node of current node. **T.C = O(n) S.C=O(n) avg./worst**

```
void LevelOrder(Node *root) {
    if(root == NULL) return;
    queue<Node*> Q;
    Q.push(root);
    //while there is at least one discovered node
    while(!Q.empty()) {
        Node* current = Q.front();
        cout<<current->data<<" ";
        if(current->left != NULL) Q.push(current->left);
        if(current->right != NULL) Q.push(current->right);
        Q.pop(); // removing the element at front
    }
}
```

mycod

```
void levelorder(TreeNode* A, vector<int> &ans){
  if(A==NULL) return;
  queue<TreeNode*> q;
  q.push(A);
  q.push(NULL);

  while(!q.empty()){
    TreeNode* curr= q.front();
    q.pop();
    if(curr!=NULL){
      ans.push_back(curr->val);
      if(curr->left!=NULL) q.push(curr->left);
      if(curr->right!=NULL) q.push(curr->right);
    }
    else if(!q.empty()){
      q.push(NULL);
    }
  }
  return;
}

vector<int> Solution::solve(TreeNode* A) {
  vector<int> ans;
  levelorder(A,ans);
  return ans;
}
```

==========

- **Depth-first**: we first complete the whole subtree of the child before going to next child.
    a. Pre-order
    b. In-order
    c. Post-order

IN-Order Pre-Order Post-order:

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1
Breadth First or Level Order Traversal : 1 2 3 4 5
Please see this post for Breadth First Traversal.

Inorder Traversal (Practice):

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal (Practice):

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal (Practice):

Preorder is reverse of Postorder if postorder is like right, left, root.

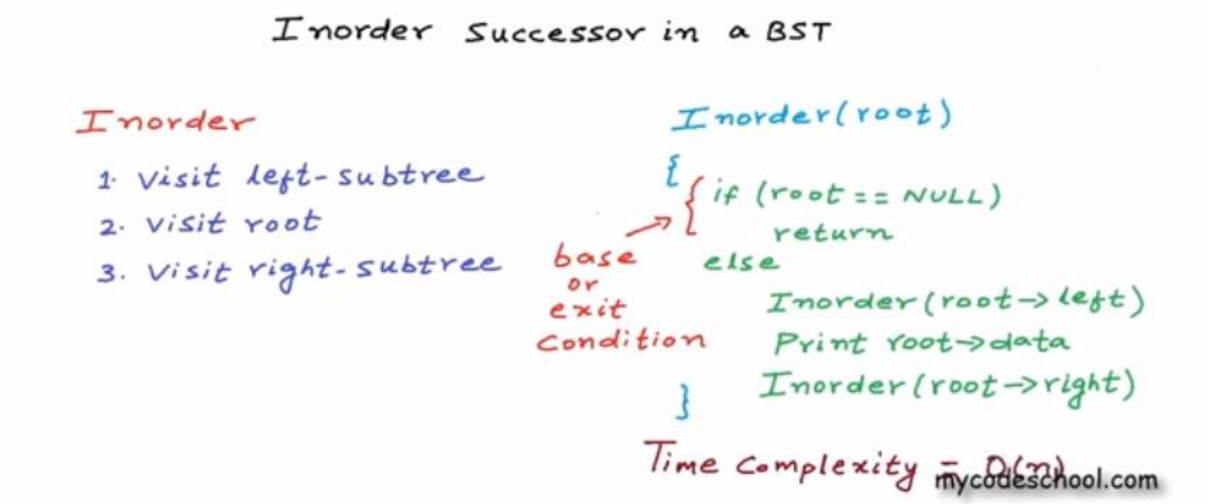
If we were given Postorder as right,left root, so its reverse is preorder and we would create tree left brach of tree first.
But here we are given left, right,root. Now to see it as a reverse of pre order we will first make right subtree instead of as usual left subtree.

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Uses of Postorder
Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

In-Order traversal: Give us sorted array.

Using recursion:

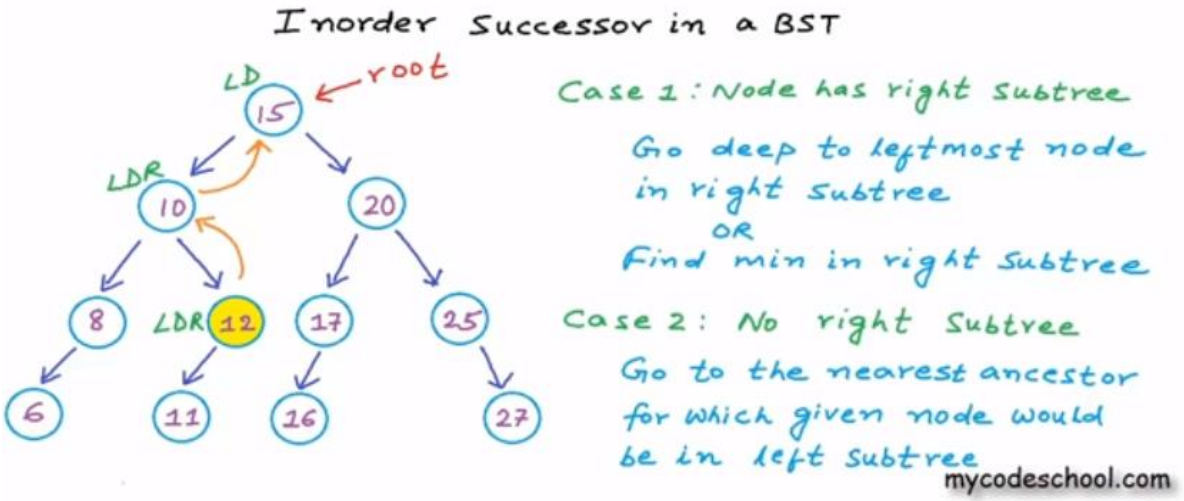


Using Stack:

```
void inOrder(struct Node *root)
{
    stack<Node *> s;
    Node *curr = root;

    while (curr != NULL || s.empty() == false)
    {
        /* Reach the left most Node of the
           curr Node */
        while (curr !=  NULL)
        {
            /* place pointer to a tree node on
               the stack before traversing
              the node's left subtree */
            s.push(curr);
            curr = curr->left;
        }

        /* Current must be NULL at this point */
        curr = s.top();
        s.pop();

        cout << curr->data << " ";

        /* we have visited the node and its
           left subtree.  Now, it's right
           subtree's turn */
        curr = curr->right;

    } /* end of while */
}
```

IN-order Successor:

1. Just do in-order traversal and find the next of that element-- **O(n)**
2. For **O(h)** h=height log2(n)



Code:

```
TreeNode* search(TreeNode* A, int B){
  if(A==NULL) return NULL;

  if(A->val==B) return A;
  else if(A->val > B){
    return search(A->left, B);
  }
  else{
    return search(A->right, B);
  }
}

TreeNode* findmin(TreeNode* A){
  if(A==NULL) return NULL;
  while(A->left!=NULL){
    A=A->left;
  }
  return A;
}

TreeNode* Solution::getSuccessor(TreeNode* A, int B) {
  TreeNode* current= search(A,B);
  if(current ==NULL ) return NULL;

  if(current->right !=NULL){
    return findmin(current->right);
  }

  else{
    //find its deepest ancestor to whom it is in left subtree.
    TreeNode* succesor = NULL;
    TreeNode* ancestor=A;
    while(current!=ancestor){
      if(current->val <ancestor->val){
        succesor=ancestor;
        ancestor=ancestor->left;
      }
      else{
        ancestor=ancestor->right;
      }
    }
    return succesor;
  }

}
```

Pre-Order traversal:

Using stack:

```
void preorder(TreeNode* A, vector<int> &ans){
  if(A==NULL) return;
  stack<TreeNode* > s;
  s.push(A);
  while(!s.empty()){
    TreeNode* a=s.top();
    ans.push_back(a->val);
    s.pop();

    if(a->right!=NULL) s.push(a->right);
    if(a->left!=NULL)  s.push(a->left);
  }

  return;
}

vector<int> Solution::preorderTraversal(TreeNode* A) {
  vector<int> ans;
  preorder(A, ans);
  return ans;
}
```

Using recursion:

```
void preorder(TreeNode* A, vector<int> &ans){
   if(A==NULL) return;
   else{
      ans.push_back(A->val);
      preorder(A->left, ans);
      preorder(A->right, ans);
   }

   return;
}

vector<int> Solution::preorderTraversal(TreeNode* A) {
   vector<int> ans;
   preorder(A, ans);
   return ans;
}
```

Post-Order Traversal:

Using recursion:

```
void postorder(TreeNode* A, vector<int> &ans){
   if(A==NULL) return;
   else{
      postorder(A->left, ans);
      postorder(A->right, ans);
      ans.push_back(A->val);
   }

   return;
}

vector<int> Solution::postorderTraversal(TreeNode* A) {
   vector<int> ans;
   postorder(A, ans);
   return ans;
}
```

Using 2-Stack:

```
void postorder(TreeNode* A, vector<int> &ans){
   if(A==NULL) return;
   stack<TreeNode*> s1, s2;
   s1.push(A);
   while(!s1.empty()){
      TreeNode* a=s1.top();
      s1.pop();
      s2.push(a);

      if(a->left!=NULL) s1.push(a->left);
      if(a->right!=NULL) s1.push(a->right);
```

```
    }

    while(!s2.empty()){
        ans.push_back(s2.top()->val);
        s2.pop();
    }
    return;
}

vector<int> Solution::postorderTraversal(TreeNode* A) {
    vector<int> ans;
    postorder(A, ans);
    return ans;
}
```

Using 1-stack:

```
void postorder(TreeNode* A, vector<int> &ans){
    if(A==NULL) return;
    stack<TreeNode*> s1;
    s1.push(A);
    while(!s1.empty()){
        TreeNode* a=s1.top();
        s1.pop();
        ans.push_back(a->val);

        if(a->left!=NULL) s1.push(a->left);
        if(a->right!=NULL) s1.push(a->right);
    }

    return;
}

vector<int> Solution::postorderTraversal(TreeNode* A) {
    vector<int> ans;
    postorder(A, ans);
    reverse(ans.begin(),ans.end());
    return ans;
}
```

```
/ Push nodes of a given binary search tree into a vector in sorted order
    vector<Node*> nodes;
    pushTreeNodes(root, nodes);
```

From <https://www.techiedelight.com/construct-height-balanced-bst-from-unbalanced-bst/>