Trees

Topics

1. Definition
2. Terminology
3. Types of Tree
4. Implementation
5. Problems & Solutions

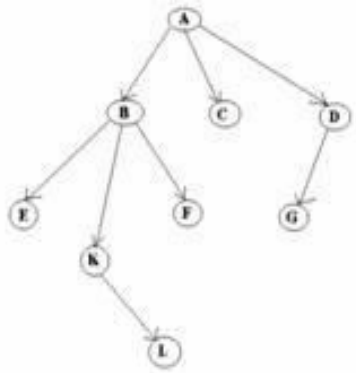
Tree

A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees.

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

A tree has following general properties:

1. One node is distinguished as a root;
2. Every node (exclude a root) is connected by a directed edge from exactly one other node; A direction is: parent -> children



A is a parent of B, C, D. B is called a child of A.
On the other hand, B is a parent of E, F, K.
In the above picture, the root has 3 subtrees.

Each node can have arbitrary number of children. Nodes with no children are called leaves, or external nodes. In the above picture, C, E, F, L, G are leaves. Nodes, which are not leaves, are called internal nodes. Internal nodes have at least one child.

Nodes with the same parent are called siblings. In the picture, B, C, D are called siblings. The depth of a node is the number of edges from the root to the node. The depth of K is 2. The height of a node is the number of edges from the node to the deepest leaf. The height of B is 2. The height of a tree is a height of a root.

Terminologies used in Trees

- Root – The top node in a tree.
- Child – A node directly connected to another node when moving away from the Root.
- Parent – The converse notion of a *child*.
- Siblings – Nodes with the same parent.
- Descendant – A node reachable by repeated proceeding from parent to child.
- Ancestor – A node reachable by repeated proceeding from child to parent.
- Leaf – A node with no children.
- Internal node – A node with at least one child
- External node – A node with no children.
- Degree – Number of sub trees of a node.
- Edge – Connection between one node to another.
- Path – A sequence of nodes and edges connecting a node with a descendant.
- Level – The level of a node is defined by 1 + (the number of connections between the node and the root).
- Height of node – The height of a node is the number of edges on the longest downward path between that node and a leaf.
- Height of tree – The height of a tree is the height of its root node.
- Depth – The depth of a node is the number of edges from the node to the tree's root node.
- Forest – A forest is a set of $n \geq 0$ disjoint trees.

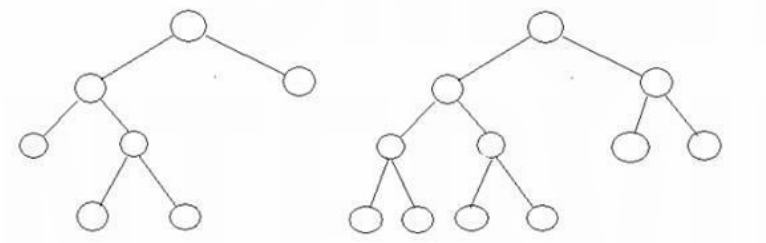
General Tree

A general tree is a tree where each node may have zero or more children (a binary tree is a specialized case of a general tree). General trees are used to model applications such as file systems.

Binary Trees

A tree where each node can have no more than two children is called a Binary Tree.

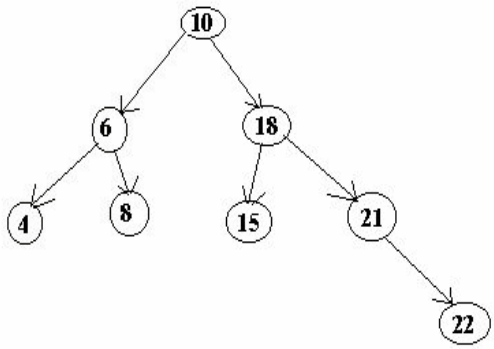
A binary tree in which each node has exactly zero or two children is called a full binary tree. In a full tree, there are no nodes with exactly one child.
A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2^h and $2^{(h+1)}$-1 nodes. Here are some examples:



Full Tree **Complete Tree**

Binary Search Trees

Given a binary tree, suppose we visit each node (recursively) as follows. We visit left child, then root and then the right child. For example, visiting the following tree



In the order defined above, this tree will produce the sequence {4, 6, 8, 10, 15, 18, 21, 22} which we call flat(T). A binary search tree (BST) is a tree, where flat(T) is an ordered sequence.
In other words, a binary search tree can be "searched" efficiently using this ordering property. A "balanced" binary search tree can be searched in O(log n) time, where n is the number of nodes in the tree.

A binary tree is a binary search tree (BST) if and only if an inorder traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently.

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

The basic operations that can be performed on binary search tree data structure are following:

1. Insert − insert an element in a tree/create a tree.
2. Search − search an element in a tree.
3. Delete – Delete an element from a tree

Traversal methods

Pre-order

1. Display the data part of the root (or current node).
2. Traverse the left subtree by recursively calling the pre-order function.
3. Traverse the right subtree by recursively calling the pre-order function.

In-order

1. Traverse the left subtree by recursively calling the in-order function.
2. Display the data part of the root (or current node).
3. Traverse the right subtree by recursively calling the in-order function.

Post-order

1. Traverse the left subtree by recursively calling the post-order function.
2. Traverse the right subtree by recursively calling the post-order function.
3. Display the data part of the root (or current node).

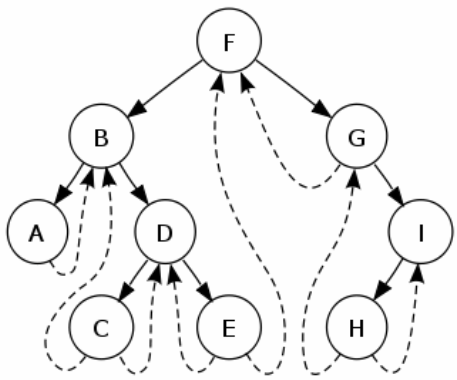
Code for Binary Search Tree

```
#include <iostream>
using namespace std;

typedef struct node{
    int data;
    struct node *left, *right;
} tree;

tree *createNode(int data) {
    tree *root = (tree *)(malloc)(sizeof(tree *));
    root->data = data;
    root->left = root->right = NULL;
    return root;
}

void inorder(tree *root) {
    if(root == NULL)
        return;
    inorder(root->left);
    cout<<root->data<<" ";
    inorder(root->right);
}

void preorder(tree *root) {
    if(root == NULL)
        return;
    cout<<root->data<<" ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(tree *root) {
    if(root == NULL)
        return;
    postorder(root->left);
    postorder(root->right);
    cout<<root->data<<" ";
}

int findMax(tree *root) {
    while(root->right!=NULL)
        root = root->right;
    return root->data;
}
```

```
tree *insert(int data, tree *root) {
    if(root == NULL)
        return createNode(data);
    if(root->data>data)
        root->left = insert(data, root->left);
    else
        root->right = insert(data, root->right);
    return root;
}

bool search(int data, tree *root) {
    if(root==NULL)
        return false;
    if(root->data == data)
        return true;
    if(root->data > data)
        return search(data, root->left);
    return search(data, root->right);
}

node *deleteNode(int data, tree *root) {
    if(root == NULL)
        return NULL;
    if(root->data > data)
        root->left = deleteNode(data, root->left);
    else if(root->data < data)
        root->right = deleteNode(data, root->right);
    else {
        if(root->left==NULL) {
            node *tmp = root->right;
            free(root);
            return tmp;
        } else if(root->right == NULL) {
            node *tmp = root->left;
            free(root);
            return tmp;
        } else {
            root->data = findMax(root->left);
            root->left = deleteNode(root->data, root->left);
        }
    }
    return root;
}
```

```
int main() {
    tree *root = createNode(4);
    insert(2, root);     insert(1, root);
    insert(3, root);     insert(6, root);
    insert(5, root);     insert(8, root);
    insert(7, root);     insert(10, root);
    insert(9, root);

    inorder(root);
    cout<<endl;
    preorder(root);
    cout<<endl;
    postorder(root); cout<<endl;
    cout<<search(5, root)<<endl;
    cout<<search(4, root)<<endl;
    cout<<search(12, root)<<endl;
    cout<<search(10, root)<<endl;
    cout<<search(-1, root)<<endl;

    root = deleteNode(10, root);
    root = deleteNode(4, root);
    root = deleteNode(5, root);
    root = deleteNode(-1, root);

    inorder(root);     cout<<endl;
    preorder(root);   cout<<endl;
    postorder(root);  cout<<endl;
}
```

Threaded Binary Tree



A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

InOrder Traversal of a Binary Tree using the concept Threaded Binary Tree

A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists). The idea of threaded binary trees is to perform inorder traversal without stack or recursion.

Algorithm

a. Initialize current as root
b. While current is not NULL
    a. If current does not have left child, print current's data and go to the right, i.e., current = current->right, else
    b. Make current as right child of the rightmost node in current's left subtree. Go to current's left child, i.e., current = current->left

Implementation

```
void MorrisTraversal(Node *root) {
   if(root == NULL)
       return;
   Node *current,*pre;
   current = root;
   while(current != NULL) {
      if(current->left == NULL) {
         printf(" %d ", current->data);
         current = current->right;
      }
      else {
          /* Find the inorder predecessor of current */
         pre = current->left;
         while(pre->right != NULL && pre->right != current)
            pre = pre->right;

         /* Make current as right child of its inorder predecessor */
         if(pre->right == NULL) {
            pre->right = current;
            current = current->left;
         }
         /* Revert the changes made in if part to restore the original tree i.e., fix the right
child of predecssor */
         else {
            pre->right = NULL;
            printf(" %d ",current->data);
            current = current->right;
         }
      }
   }
}
```

Problems

1. WAF to find the height of a binary tree.
   Solution

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

2. WAF to convert a binary tree into its mirror tree
   Solution

```
void mirror(node *root) {
   if(root == NULL)
      return ;
   node *tmp = root->left;
   root->left = root->right;
   root->right = tmp;
   mirror(root->left);
         mirror(root->right);
}
```

3. Check if a given Binary Tree is a Binary Search Tree or not.
   Solution: The trick is to write a utility helper function isBSTUtil(node* node, int min, int max) that traverses down the tree keeping track of the allowed values of min and max as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

4. Print Postorder traversal from given Inorder and Preorder traversals.
   Input: Inorder traversal = {4, 2, 5, 1, 3, 6}, Preorder traversal = {1, 2, 4, 5, 3, 6}
   Output: Postorder traversal = {4, 5, 2, 6, 3, 1}

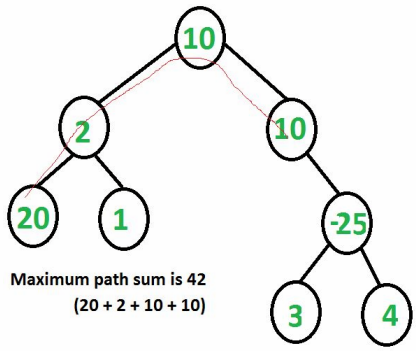
   Solution: Construct the tree and and then print Post-order traversal
   a. Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.
      i. Create a new tree node tNode with the data as picked element.
      ii. Find the picked element's index in Inorder. Let the index be inIndex.
      iii. Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
      iv. Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
      v. return tNode.

```
int search(int arr[], int strt, int end, int value) {
   for(int i = strt; i <= end; i++) {
      if(arr[i] == value)
         return i;
   }
}
```

```
node* buildTree(int in[], int pre[], int inStrt, int inEnd) {
   if(inStrt > inEnd)
      return NULL;
   static int preIndex = 0;
   node *tNode = createNode(pre[preIndex++]);
   if(inStrt == inEnd)
      return tNode;

   int inIndex = search(in, inStrt, inEnd, tNode->data);
   tNode->left = buildTree(in, pre, inStrt, inIndex-1);
   tNode->right = buildTree(in, pre, inIndex+1, inEnd);
   return tNode;
}
```

We can print the Post-order traversal without constructing the tree as well. Process left->Process right->Print Current.

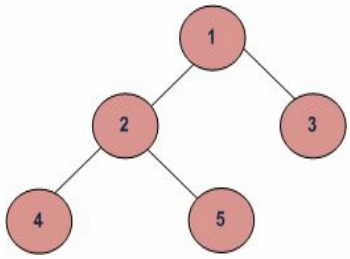
5. Find maximum non empty path sum in a binary tree.
   Solution
   a. For every node, find the maximum sum single path in the left subtree and in the right subtree.
   b. Check the highest sum path which you can form by including the current node.
   c. Update your global answer
   d. Return the maximum possible single path sum from using current node to the parent calling function.




```
int findMaxUtil(Node* root, int &res) {
   if (root == NULL)
      return 0;

   int l = findMaxUtil(root->left, res);
   int r = findMaxUtil(root->right, res);

   // Max path for parent call. This path must include only 1 child of root
   int max_single = max(max(l, r) + root->data, root->data);

   // max_top represents the sum when the current node is the root of the
   maxsum path and no ancestors of root are there in max sum path
   int max_top = max(max_single, l + r + root->data);
   res = max(res, max_top); // Store the Maximum Result.
   return max_single;
}
```

```
// Returns maximum path sum in tree with given root
int findMaxSum(Node *root) {
    int res = INT_MIN;
    findMaxUtil(root, res);
    return res;
}
```

6. Level order traversal of a tree.



Level order traversal of this tree is 1 2 3 4 5

Solution: Keep the elements in a queue data structure

```
void levelOrderTraversal(node *root) {
   if(root == NULL)
      return ;
   queue<node*> q;
   q.push(root);

   while(!q.empty()) {
      node *tmp = q.front();
      q.pop();
      cout<<tmp->data<<" ";
      if(tmp->left)
         q.push(tmp->left);
      if(tmp->right)
         q.push(tmp->right);
   }
}
```

7. Iterative Traversals of Binary Trees.
    a. **In-order**
        i. Create an empty stack S.
        ii. Initialize current node as root
        iii. Push the current node to S and set current = current->left until current is NULL
        iv. If current is NULL and stack is not empty then
        v. Pop the top item from stack.
        vi. Print the popped item, set current = popped_item->right
        vii. Go to step c.
        viii. If current is NULL and stack is empty then we are done.

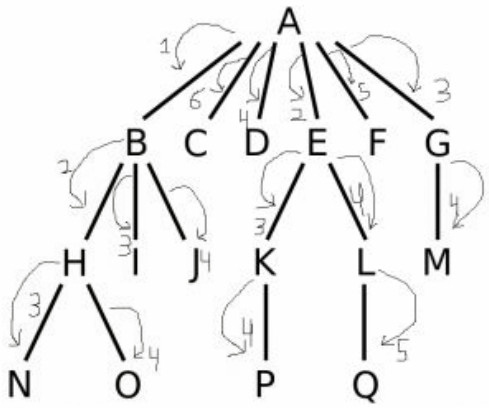
```
void inOrderIterative(node *root) {
   node *current = root;
   stack< node* > nodeStack;
   while (1) {
      while(current !=  NULL)  {
         nodeStack.push(current);
         current = current->left;
      }

      if (!nodeStack.empty()) {
         current = nodeStack.top();
         nodeStack.pop();
         cout<<(current->data)<<" ";
         current = current->right;
      }
      else break;
   }
}
```

b. **Pre-order**
   i. Create an empty stack nodeStack and push root node to stack.
   ii. Do following while nodeStack is not empty.
      1. Pop an item from stack and print it.
      2. Push right child of popped item to stack
      3. Push left child of popped item to stack

```
void preOrderIterative(node *root) {
   if (root == NULL)
      return;

   stack< node* > nodeStack;
   nodeStack.push(root);

   while(!nodeStack.empty()) {
      node *current = nodeStack.top();
      nodeStack.pop();
      cout<<(current->data)<<" ";
      if (current->right)
         nodeStack.push(current->right);
      if (current->left)
         nodeStack.push(current->left);
   }
}
```

c. **Post-order**
   i. Implement a variation of Pre-order traversal of the form – DRL [not DLR]
   ii. Instead of printing the elements in this order, push the elements to another stack as and when you print.
   iii. After exhausting the entire tree, you can print the second stack and you will get Post-order traversal of the tree – LRD [reverse of DRL].

8. Given a very large n-ary tree, where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one second). Find the minimum time required to pass the information to all nodes in the tree.



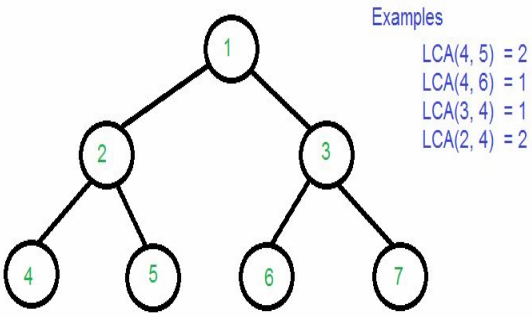
Minimum time for this tree is 6. The root A first passes information to B. Next, A passes information to E and B passes information to H and so on.

Solution

Let *minTime(A)* be the minimum iteration needed to pass info from node A to all the sub-trees of A. Let children(A) be the count of the number of children for node A.

a. For a node A, get minTime(B), where B is a child of A. Do this all the childs of the current node.
b. Compute minTime(A) of A, based on minTime(Bs)
   i. Sort the array minTime(B) in descending order.
   ii. minTime(A) = max(minTime(B_i) + i)
c. Base cases would be: if node is leaf, minTime = 0

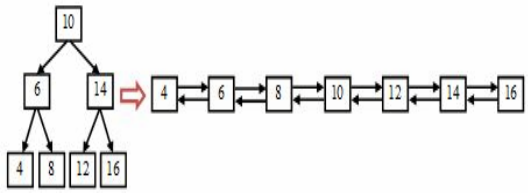
9. Given 2 nodes, WAF to find the Least Common Ancestor(LCA) in a Binary Tree



Solution: The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA, otherwise LCA lies in right subtree.

```
node *findLCA(node* root, int n1, int n2) {
   if (root == NULL)
      return NULL;
   if (root->data == n1 || root-> data == n2)
      return root;

   node *left_lca  = findLCA(root->left, n1, n2);
   node *right_lca = findLCA(root->right, n1, n2);
   if (left_lca && right_lca)
      return root;
   return (left_lca != NULL)? left_lca: right_lca;
}
```

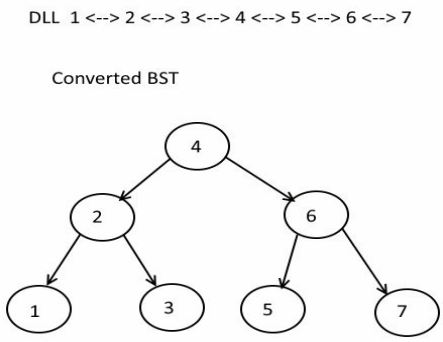
10. Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



Solution: The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say *prev*. For every visited node, make it next of *prev* and previous of this node as *prev*.

```
node *BinaryTree2DoubleLinkedList(node *root) {
   if (root == NULL) return NULL;
   static node* prev = NULL;

   node *head = BinaryTree2DoubleLinkedList(root->left);
   if (prev == NULL)
      head = root;
   else {
      root->left = prev;
      prev->right = root;
   }
   prev = root;
   BinaryTree2DoubleLinkedList(root->right);
   return head;
}
```

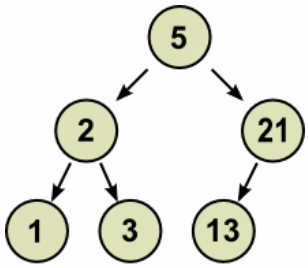
11. Given a Doubly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Doubly Linked List. The tree must be constructed in-place.



Solution: Let's construct the BST from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Doubly Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root. While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

```
int countNodes(node *head) {
   int count = 0;
   while(head) {
      head = head->right;
      count++;
   }
   return count;
}

node* sortedListToBSTRecur(node **head_ref, int n) {
   if (n <= 0)
      return NULL;

   node *left = sortedListToBSTRecur(head_ref, n/2);
   node *root = *head_ref;
   root->left = left;
   *head_ref = (*head_ref)->right;
   root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

   return root;
}

node* sortedListToBST(node *head) {
   int n = countNodes(head);
   return sortedListToBSTRecur(&head, n);
}
```

12. Sum of all the numbers that are formed from root to leaf paths.



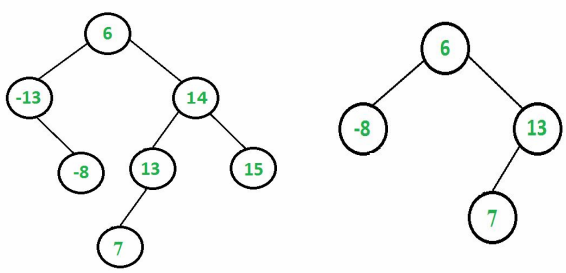
For example, the numbers formed from root to leaf in this tree are:
521, 523, 52113

Sum is: 521 + 523 + 52113 = **53157**

Solution: Do a top down traversal of the binary tree, passing the value of the number formed so far. When a encounter a new node, multiple your current value by 10^n, where n is the number of digits in current->data. As soon as you reach a NULL, add the current value to your answer.

```
int numDigits(int data) {
   int ret = 1;
   while(data) {
      ret *= 10;
      data /= 10;
   }
   return ret;
}

int pathSums(node *root) {
   return pathSumsUtil(root, 0);
}

int pathSumsUtil(node *root, int val) {
   if(root == NULL)
      return 0;
   val = val*numDigits(root->data) + root->data;
   if(root->left == NULL && root->right == NULL)
      return val;

   return pathSumsUtil(root->left, val) + pathSumsUtil(root->right, val);
}
```

13. Given a Binary Search Tree (BST) and a range [min, max], remove all keys which are outside the given range. The modified tree should also be BST. For example, consider the following BST and range [-10, 13].



Input **Output**

Solution: The idea is to fix the tree in Postorder fashion. When we visit a node, we make sure that its left and right sub-trees are already fixed. In case ai, we simply remove root and return right sub-tree as new root. In case aii, we remove root and return left sub-tree as new root.

There are two possible cases for every node:

a. Node's key is outside the given range. This case has two sub-cases.
    i. Node's key is smaller than the min value.
    ii. Node's key is greater that the max value.
b. Node's key is in range

14. Find a pair with given sum, S in a Balanced BST.
    Solution
    a. For every node x, search the other with value S-x : O(nlogn), S(1)
    b. Copy to an auxiliary array and use 2 pointer solution: O(n), S(n)
    c. Convert to sorted double LL: O(n) , S(1) - *Tree is modified !*
    d. Simultaneously perform inorder and reverse inorder traversal : O(n), S(1)

Notes

1. Handle duplicates in BST
    a. A Simple Solution is to allow same keys on right side
    b. A Better Solution is to augment every tree node to store count, along with regular fields like data, left and right pointers.

2. Index in SQL using B-Trees
    References:
    a. https://en.wikipedia.org/wiki/B-tree
    b. http://www.programmerinterview.com/index.php/database-sql/what-is-an-index/

3. C++ map/set are implemented using Balanced Binary Search Tree
    a. Complexity of Search/InsertDelete – O(logn)
    b. References:
        i. http://www.cplusplus.com/reference/map/map/
        ii. http://stackoverflow.com/questions/5288320/why-is-stdmap-implemented-as-a-red-black-tree
        iii. http://www.cplusplus.com/forum/general/56519/

4. Java TreeMap/TreeSet are implemented using Balanced Binary Search Tree
    a. Complexity of Search/InsertDelete – O(logn)
    b. Reference: http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html