**Construct BST**

# Construct BST from given Preorder traversal

## Method - 1 O(n^2) time complexity

1. First element of preorder traversal is always root. We first construct the root.
2. Find the index of first element which is greater than root. Let the index be ‘i’.
   1. Left subtree: values between root and ‘i’
   2. Right subtree: values between ‘i+1’ and ‘n-1’
3. Divide given pre[] at index “i” and recur for left and right sub-trees.

**Time Complexity: O(n^2)**

#include <iostream>

using namespace std;

// A Binary Tree Node

struct Node {

int data;

struct Node \*left, \*right;

};

// Utility function to create a new tree node

Node\* newNode(int data) {

Node \*temp = new Node;

temp->data = data;

temp->left = temp->right = NULL;

return temp;

}

// function to print preorder traversal of the tree

void preorder\_traverse(Node \* arg\_root) {

if(NULL == arg\_root) {

return;

}

cout << (arg\_root -> data) << " ";

preorder\_traverse(arg\_root -> left);

preorder\_traverse(arg\_root -> right);

return;

}

// function to print preorder traversal of the tree

void inorder\_traverse(Node \* arg\_root) {

if(NULL == arg\_root) {

return;

}

inorder\_traverse(arg\_root -> left);

cout << (arg\_root -> data) << " ";

inorder\_traverse(arg\_root -> right);

return;

}

Node \* construct\_Tree\_Util(int \* pre, const int root\_index, const int left, const int right, const int size) {

Node \* root = NULL;

int left\_index = -1;

int right\_index = -1;

int temp\_index = -1;

//cout << "Root : " << root\_index << " Left: " << left << " Right: " << right;

if((root\_index >= size) || (left > right)) {

return NULL;

}

// make root by creating a new node

root = newNode(pre[root\_index]);

//cout << " Data: " << root -> data << endl;

// If the current subarry has only one element, no need to recur

if(left == right) {

return root;

}

left\_index = root\_index + 1;

// Search for the first element greater than root

right\_index = root\_index + 1;

while((right\_index < right) && (pre[right\_index] < pre[root\_index])) {

right\_index = right\_index + 1;

}

//cout << "left\_index : " << left\_index << " right\_index: " << right\_index << endl;

//divide preorder array in two parts Left subtree and right subtree

root -> left = construct\_Tree\_Util(pre, left\_index, left\_index, (right\_index - 1), size);

root -> right = construct\_Tree\_Util(pre, right\_index, right\_index, right, size);

return root;

}

Node \* construct\_Tree(int \* pre, const int size) {

return construct\_Tree\_Util(pre, 0, 0, (size - 1), size);

}

// Driver program to test above functions

int main(void) {

Node\* root = NULL;

int pre[] = {10, 5, 1, 7, 40, 50};

int size = sizeof(pre) / sizeof(pre[0]);

root = construct\_Tree(pre, size);

cout << "Preorder Traverse: ";

preorder\_traverse(root); cout << endl;

cout << "Inorder Traverse: ";

inorder\_traverse(root); cout << endl;

return 0;

}

Output:

Preorder Traverse: 10 5 1 7 40 50

Inorder Traverse: 1 5 7 10 40 50

# Method – 2 O(n) time complexity

The method is inspired from method to check a BST using min and max value.

1. The trick is to set a range {min .. max} for every node.
2. Initialize the range as {INT\_MIN .. INT\_MAX}. The first node will definitely be in range, so create root node.
   1. left subtree, set the range as {INT\_MIN …root->data}.
   2. right subtree, set the range as {root->data..max .. INT\_MAX}.
3. Recall for left and right subtree.

**Time Complexity: O(n)**

struct node\* constructTreeUtil( int pre[], int\* preIndex, int key, int min, int max, int size );

// The main function to construct BST from given preorder traversal.

// This function mainly uses constructTreeUtil()

struct node \*constructTree (int pre[], int size) {

int preIndex = 0;

return constructTreeUtil ( pre, &preIndex, pre[0], INT\_MIN, INT\_MAX, size );

}

// A recursive function to construct BST from pre[]. preIndex is used to keep track of index in pre[].

struct node\* constructTreeUtil( int pre[], int\* preIndex, int key, int min, int max, int size )

{

// Base case

if( \*preIndex >= size )

return NULL;

struct node\* root = NULL;

// If current element of pre[] is in range, then

// only it is part of current subtree

if((key > min) && (key < max)) {

// Allocate memory for root of this subtree and increment \*preIndex

root = newNode(key);

\*preIndex = \*preIndex + 1;

if (\*preIndex < size) {

// Contruct the subtree under root

// All nodes which are in range {min .. key} will go in left subtree

root->left = constructTreeUtil( pre, preIndex, pre[\*preIndex], min, key, size );

// All nodes which are in range {key..max} will go in right subtree

root->right = constructTreeUtil( pre, preIndex, pre[\*preIndex], key, max, size );

}

}

return root;

}

# Method - 3 Using Stack

**Algorithm:**

1. Create an empty stack.
2. Make the first value as root. Push it to the stack.
3. Keep on popping while the stack is not empty and the next value is greater than stack’s top value. Make this value as the right child of the last popped node Push the new node to the stack.
4. If the next value is less than the stack’s top value, make this value as the left child of the stack’s top node. Push the new node to the stack.
5. Repeat steps 2 and 3 until there are items remaining in pre[].

**Time Complexity: O(n)**

Every item is pushed and popped only once. So at most 2n push/pop operations are performed in the main loops of constructTree().

Therefore, time complexity is O(n).

Node \* construct\_Tree(int \* pre, const int size) {

stack <Node \*> st;

Node \* temp = NULL;

Node \* root = NULL;

int index = 0;

if(0 > size) {

return NULL;

}

index = 0;

root = newNode(pre[index]);

index = index + 1;

st.push(root);

while(index < size) {

temp = NULL;

while((!st.empty()) && (pre[index] > (st.top() -> data))) {

temp = st.top();

st.pop();

}

if(NULL != temp) {

temp -> right = newNode(pre[index]);

st.push(temp -> right);

}

else {

temp = st.top();

temp -> left = newNode(pre[index]);

st.push(temp -> left);

}

index++;

}

return root;

}

Output:

Preorder Traverse: 10 5 1 7 40 50

Inorder Traverse: 1 5 7 10 40 50

# END