**Sorted Array to Balanced BST**

The goal is to identify the middle element of the array and make it the tree's root, after which the same operation is carried out on the left subarray to find the root's left child and the right subarray to find the root's right child.

Follow the steps mentioned below to implement the approach:

* Set The middle element of the array as root.
* Recursively do the same for the left half and right half.
  + Get the middle of the left half and make it the left child of the root created in step 1.
  + Get the middle of the right half and make it the right child of the root created in step 1.
* Print the preorder of the tree.

Below is the implementation of the above approach in C++ language:

#include<bits/stdc++.h>

using namespace std;

// Creating A Binary Tree node

class TreeNode {

public:

int data;

TreeNode\* left;

TreeNode\* right;

};

TreeNode\* newNode(int data);

// A function to construct a Balanced Binary Search Tree from a sorted array.

TreeNode\* sortedArrayToBST(int arr[], int start, int end) {

//Creating Base Case

if (start > end)

return NULL;

// Get the middle element and make it root.

int mid = (start + end)/2;

TreeNode \*root = newNode(arr[mid]);

// Construct the left subtree recursively and make it left child of root.

root->left = sortedArrayToBST(arr, start, mid - 1);

// Construct the right subtree recursively and make it right child of root.

root->right = sortedArrayToBST(arr, mid + 1, end);

return root;

}

// Function that allocates a new node with the given data and NULL left and right pointers.

TreeNode\* newNode(int data) {

TreeNode\* node = new TreeNode();

node->data = data;

node->left = NULL;

node->right = NULL;

return node;

}

// Function to print preorder traversal of BST.

void preOrder (TreeNode\* node) {

if (node == NULL)

return;

cout << node->data << " ";

preOrder(node->left);

preOrder(node->right);

}

// Main Code

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

// Convert List to BST.

TreeNode \*root = sortedArrayToBST(arr, 0, n-1);

cout << "PreOrder Traversal of constructed BST"<<endl;

preOrder(root);

return 0;

}

**Output:**

PreOrder Traversal of constructed BST

4 2 1 3 6 5 7

**Deleting a node in BST**

// C++ program to demonstrate delete operation in binary search tree.

#include <bits/stdc++.h>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

// Function to create a new BST node.

struct node\* newNode(int item) {

struct node\* temp = (struct node\*) malloc (sizeof(struct node));

temp->key = item;

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

return temp;

}

// Function to do inorder traversal of BST.

void inorder(struct node\* root) {

if (root != NULL) {

inorder(root->left);

cout << root->key<<” “;

inorder(root->right);

}

}

// A utility function to insert a new node with given key in BST.

struct node\* insert (struct node\* node, int key) {

// If the tree is empty, return a new node.

if (node == NULL)

return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

// Return the (unchanged) node pointer.

return node;

}

// Given a non-empty binary search tree, return the node with minimum key value found in that tree. Note that the entire tree does not need to be searched.

struct node\* minValueNode(struct node\* node)

{

struct node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current && current->left != NULL)

current = current->left;

return current;

}

// Given a binary search tree and a key, this function deletes the key and returns the new root.

struct node\* deleteNode(struct node\* root, int key) {

// Base case

if (root == NULL)

return root;

// If the key to be deleted is smaller than the root's key, then it lies in left subtree.

if (key < root->key)

root->left = deleteNode(root->left, key);

// If the key to be deleted is greater than the root's key, then it lies in right subtree.

else if (key > root->key)

root->right = deleteNode(root->right, key);

// If key is same as root's key, then This is the node to be deleted.

else {

// node has no child

if (root->left==NULL and root->right==NULL)

return NULL;

// node with only one child or no child

else if (root->left == NULL) {

struct node\* temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL) {

struct node\* temp = root->left;

free(root);

return temp;

}

// node with two children: Get the inorder successor (smallest in the right subtree).

struct node\* temp = minValueNode(root->right);

// Copy the inorder successor's content to this node

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

// Main Code

int main()

{

struct node\* root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout << "Inorder traversal of the given tree \n";

inorder(root);

cout << "\nDelete 20\n";

root = deleteNode(root, 20);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 30\n";

root = deleteNode(root, 30);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 50\n";

root = deleteNode(root, 50);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

return 0;

}

**Output:**

Inorder traversal of the given tree

20 30 40 50 60 70 80

Delete 20

Inorder traversal of the modified tree

30 40 50 60 70 80

Delete 30

Inorder traversal of the modified tree

40 50 60 70 80

Delete 50

Inorder traversal of the modified tree

40 60 70 80

**Deleting in Array**

// C++ program to remove a given element from an array

#include<bits/stdc++.h>

using namespace std;

// This function removes an element x from arr[] and returns new size after removal (size is reduced only when x is present in arr[].

int deleteElement(int arr[], int n, int x) {

int i;

for (i=0; i<n; i++) {

if (arr[i] == x)

break;

}

// If x found in array

if (i < n) {

// reduce size of array and move all elements on space ahead

n = n - 1;

for (int j=i; j<n; j++)

arr[j] = arr[j+1];

}

return n;

}

// Main Code

int main()

{

int arr[] = {11, 15, 6, 8, 9, 10};

int n = sizeof(arr)/sizeof(arr[0]);

int x = 6;

// Delete x from arr[].

n = deleteElement(arr, n, x);

cout << "Modified array is \n";

for (int i=0; i<n; i++)

cout << arr[i] << " ";

return 0;

}

**Output:**

Modified array is

11 15 8 9 10

**Space complexity of both structures**

1. Deleting an element fom an array: space complexity is O(1).
2. Deleting an node from BST: space complexity is O(n).

**SUBMITTED BY:**

**Lakshay Batra**

**21104031**

**Electrical**