**Aim:** Create and perform various operations on BST.

**Description:**

A Binary Search Tree (BST) is a type of data structure in computer science used to store data in a way that allows efficient access and retrieval. Each node in the tree has a value and two child nodes, left and right, which can each contain their own subtree.

Applications of BST include:

* Dictionary and Thesaurus: BST can be used to store words and their definitions, providing an efficient way to search for words.
* Spell Checker: BST can be used to store a list of words and efficiently check if a word is present or not.
* Priority Queue: BST can be used to implement priority queues, where the node with the highest priority is stored at the root.

Properties of BST:

1. Each node has a value that is larger than all values in its left subtree and smaller than all values in its right subtree.
2. The left and right subtrees are also BSTs.
3. The BST is efficient for searching, inserting, and deleting values.
4. The height of the BST is logarithmic in the number of nodes, which ensures that the operations are performed in an efficient manner.

The time complexity of various operations in a Binary Search Tree (BST) is as follows:

1. Searching: The time complexity of searching for a value in a BST is O(h), where h is the height of the tree. In the worst case, if the tree is unbalanced, h can be equal to the number of nodes, resulting in a time complexity of O(n). However, in the average case, the height of the tree is logarithmic in the number of nodes, resulting in a time complexity of O(log n).
2. Insertion: The time complexity of inserting a value in a BST is O(h), where h is the height of the tree. Similar to searching, in the worst case, h can be equal to the number of nodes, resulting in a time complexity of O(n). However, in the average case, the height of the tree is logarithmic in the number of nodes, resulting in a time complexity of O(log n).
3. Deletion: The time complexity of deleting a value in a BST is O(h), where h is the height of the tree. Similar to searching and insertion, in the worst case, h can be equal to the number of nodes, resulting in a time complexity of O(n). However, in the average case, the height of the tree is logarithmic in the number of nodes, resulting in a time complexity of O(log n).

Note: The time complexity of BST operations is dependent on the height of the tree and the distribution of values. In the average case, the height of the tree is logarithmic in the number of nodes, resulting in efficient operations. However, in the worst case, if the tree becomes unbalanced, the height can be equal to the number of nodes, resulting in operations with a time complexity of O(n).

**Node Of Binary Tree:**

class Node{

public:

int data;

Node \*leftChild = nullptr, \*rightChild = nullptr;

Node(int data):data(data){}

};

1. **Inserting Node into BST**

To insert a node into a Binary Search Tree (BST), follow these steps:

1. Start at the root node of the BST.
2. Compare the value of the new node with the value of the current node.
3. If the value of the new node is less than the value of the current node, move to the left child. If there is no left child, insert the new node as the left child.
4. If the value of the new node is greater than the value of the current node, move to the right child. If there is no right child, insert the new node as the right child.
5. Repeat steps 2-4 until the new node is inserted into the BST.

Node\* insertNode(int data, Node \*root){

if(root == nullptr) return nullptr;// if root is empty

else if(data < root->data)

root->leftChild = insertNode(data, root->leftChild);

else if(data > root->data)

root->rightChild = insertNode(data, root->rightChild);

return root;

}

1. **Deleting Node from BST**

Deletion in a Binary Search Tree (BST) is the process of removing a node from the tree while maintaining the BST property (i.e., left child nodes are smaller than the parent node and right child nodes are larger). There are three cases to consider when deleting a node in a BST:

* Leaf Node: If the node to be deleted is a leaf node, it can simply be removed without affecting the rest of the tree.
* Node with one child: If the node to be deleted has one child, the child node can take its place without violating the BST property.
* Node with two children: If the node to be deleted has two children, it must be replaced with its in-order successor (i.e., the node with the next smallest value in the tree) or its in-order predecessor (i.e., the node with the next largest value in the tree).

Node\* getMinValueNode(Node\* node) {

Node\* current = node;

while (current->leftChild != NULL) {

current = current->leftChild;

}

return current;

}

Node\* deleteNode(Node\* root, int key) {

if (root == NULL) {

return root;

}

if (key < root->data) {

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

} else if (key > root->data) {

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

} else {

if (root->leftChild == NULL) {

Node\* temp = root->rightChild;

delete root;

return temp;

} else if (root->rightChild == NULL) {

Node\* temp = root->leftChild;

delete root;

return temp;

}

Node\* temp = getMinValueNode(root->rightChild);

root->data = temp->data;

root->rightChild = deleteNode(root->rightChild, temp->data);

}

return root;

}

1. **Finding the max value in bst**

Node\* findMax(Node \*root){

if(root == nullptr)return nullptr;

else if(root->rightChild != nullptr) return findMin(root->rightChild);

return root;

}

1. **Finding the min value in bst**

Node\* findMin(Node \*root){

if(root == nullptr)return nullptr;

else if(root->leftChild != nullptr) return findMin(root->leftChild);

return root;

}

1. **Searching a value in bst**

bool search(Node \*root, int key){

if(root == nullptr)return false;

else if(key < root->data) return search(root->leftChild, key);

else if(key > root->data) return search(root->rightChild, key);

return true;

}

1. **Traversing in tree**

Tree traversal is the process of visiting all the nodes of a tree data structure in a specific order. There are three common tree traversal algorithms:

* In-Order Traversal: Visit the left child, then the current node, and then the right child. It is used to traverse a binary tree in sorted order.
* Pre-Order Traversal: Visit the current node, then the left child, and then the right child. It is used to create a copy of the tree.
* Post-Order Traversal: Visit the left child, then the right child, and then the current node. It is used to delete the tree or to get the postfix expression of an expression tree.

These traversals can be implemented using either recursive or iterative methods.

* 1. **inOrder**

void inOrder(Node \*root){

if(root == nullptr) return;

inOrder(root->leftChild);

cout << root->data << " ";

inOrder(root->rightChild);

}

* 1. **preOrder**

void preOrder(Node \*root){

if(root == nullptr) return;

cout << root->data << " ";

preOrder(root->leftChild);

preOrder(root->rightChild);

}

* 1. **postOrder**

void postOrder(Node \*root){

if(root == nullptr) return;

postOrder(root->leftChild);

postOrder(root->rightChild);

cout << root->data << " ";

}

1. **Count Nodes**

int countNode(Node \*root){

return (root == nullptr)?0:countNode(root->leftChild)+countNode(root->rightChild)+1;

}

void util(){

cout<<"operations=>{1 : add, 2 : delete, 3 : search}\n"<<endl;

cout<<"operations=>{4 : preorder, 5 : postorder, 6 : inorder, 7 : count}\n"<<endl;

cout<<"operations=>{8 : find min, 9 : find max}\n"<<endl;

Node \*BSTRoot = nullptr, \*temp=nullptr;

int choice, data;

loop:while(true){

cout << "Enter your choice : ";

cin >> choice;

switch(choice){

case 1:

cout << "Enter value : ";

cin >> data;

BSTRoot = insertNode(data, BSTRoot);

cout << "element added "<< data<<endl;

break;

case 2:

cout << "Enter value to delete : ";

cin >> data;

temp = deleteNode(BSTRoot, data);

if(temp == nullptr) cout << "element removed "<< data<<endl;

else cout << "element not found "<< data<<endl;

break;

case 3:

cout << "Enter value to search : ";

cin >> data;

if(search(BSTRoot, data))

cout << "element found "<<endl;

else

cout << "element not found"<<endl;

break;

case 4:

preOrder(BSTRoot);

cout<<endl;

break;

case 5:

postOrder(BSTRoot);

cout<<endl;

break;

case 6:

inOrder(BSTRoot);

cout<<endl;

break;

case 7:

cout << "total node "<<countNode(BSTRoot)<<endl;

break;

case 8:

temp = findMax(BSTRoot);

cout << "max value "<<temp->data<<endl;

break;

case 9:

temp = findMin(BSTRoot);

cout << "min value "<<temp->data<<endl;

break;

default:

return;

}

}

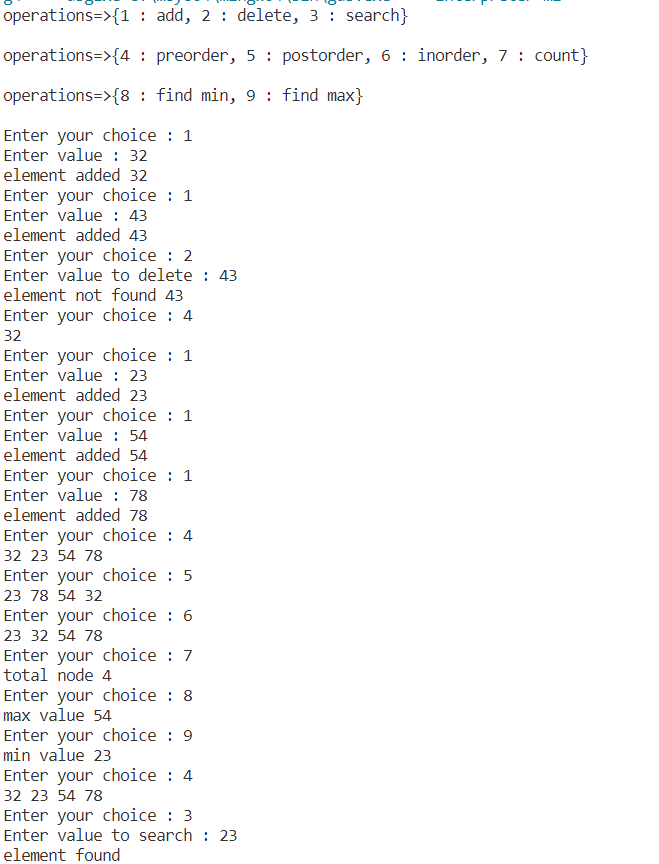
}

int main(int argc, char const \*argv[]){

util();

return 0;

}



**Conclusion:** I have learned about binary search tree.