**Pseudocode:**

1. Insertion of node in a Binary Search Tree (BST):

This pseudocode assumes that the binary search tree has a "node" class with left and right pointers, and a "value" attribute. The new node is inserted in the correct position according to the value it holds. If the value is less than the value of the current node, it is inserted in the left subtree. If the value is greater than or equal to the value of the current node, it is inserted in the right subtree. The insertion process stops when the node is null, which means that the new node has been inserted into the tree.

PROCEDURE insert(node: Node, value: T)

IF node is null

node = create new Node with value T

ELSE IF value is less than node.value

node.left = insert(node.left, value)

ELSE

node.right = insert(node.right, value)

return node

1. Deletion of node in a Binary Search Tree (BST):

This pseudocode defines two procedures: "delete" and "find\_min". The "delete" procedure searches for the node with the target value in the tree, and removes it if it is found. The "find\_min" procedure finds the minimum value in the subtree rooted at the given node.

The delete procedure starts at the root node and compares the value of the node with the target value. If the value is less than the target value, it continues the search in the left subtree. If the value is greater than the target value, it continues the search in the right subtree. If the value is equal to the target value, it removes the node from the tree.

If the node to be deleted has no children, it is simply removed from the tree. If the node has one child, it is replaced with its child. If the node has two children, it is replaced with the minimum value in its right subtree, and the minimum value is then removed from the right subtree. This ensures that the binary search tree property is maintained.

PROCEDURE delete(node: Node, value: T)

IF node is null

return null

ELSE IF value is less than node.value

node.left = delete(node.left, value)

ELSE IF value is greater than node.value

node.right = delete(node.right, value)

ELSE

IF node.left is null

node = node.right

ELSE IF node.right is null

node = node.left

ELSE

min\_node = find\_min(node.right)

node.value = min\_node.value

node.right = delete(node.right, min\_node.value)

return node

PROCEDURE find\_min(node: Node)

WHILE node.left is not null

node = node.left

return node

1. Searching of node in a Binary Search Tree (BST):

This pseudocode assumes that the binary search tree has a "node" class with left and right pointers, and a "value" attribute. The search starts at the root node, and compares the value of the node with the target value. If the values are equal, the node is returned. If the target value is less than the node value, the search continues in the left subtree. If the target value is greater than the node value, the search continues in the right subtree. If the target value is not found in the tree, the search returns null.

PROCEDURE search(node: Node, value: T)

IF node is null

return null

ELSE IF value is equal to node.value

return node

ELSE IF value is less than node.value

return search(node.left, value)

ELSE

return search(node.right, value)

1. Traversal (Inorder) of a Binary Search Tree (BST):

This pseudocode assumes that the binary search tree has a "node" class with left and right pointers, and a "value" attribute. The inorder traversal starts at the root node, and visits the left subtree first, then the root node, and finally the right subtree. The value of each node is printed during the visit. The traversal stops when the node is null, which means that all nodes in the tree have been visited.

In an inorder traversal of a binary search tree, the nodes are visited in ascending order of their values. This is because the left subtree of a node always contains nodes with smaller values than the node, and the right subtree always contains nodes with larger values.

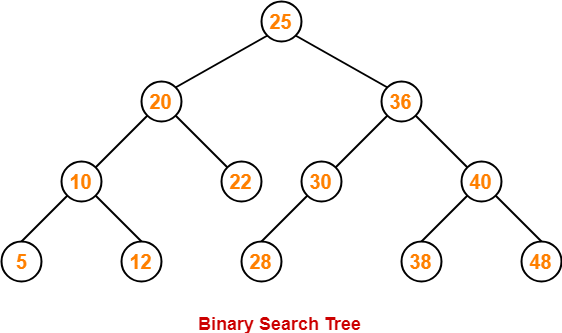
PROCEDURE inorder\_traversal(node: Node)

IF node is not null

inorder\_traversal(node.left)

print node.value

inorder\_traversal(node.right)



**Code:**

#include <iostream>

using namespace std;

class BST {

public:

int data;

BST \*left, \*right;

BST();

BST(int);

BST\* Insert(BST\*, int);

BST\* deleteNode(BST\* root, int key);

void Inorder(BST\*);

friend BST\* Search(BST\* root, int key);

friend void printnode(BST\* node);

};

BST ::BST()

: data(0)

, left(NULL)

, right(NULL)

{

}

BST ::BST(int value)

{

data = value;

left = right = NULL;

}

BST\* BST ::Insert(BST\* root, int value)

{

if (!root) {

return new BST(value);

}

if (value > root->data) {

root->right = Insert(root->right, value);

}

else if (value < root->data){

root->left = Insert(root->left, value);

}

return root;

}

void BST ::Inorder(BST\* root)

{

if (!root) {

return;

}

Inorder(root->left);

cout << root->data << endl;

Inorder(root->right);

}

BST\* Search(BST\* root, int key)

{

if (root==NULL || root->data == key)

{

cout<<"Data Found!\n";

return root;

}

if (root->data < key)

return Search(root->left, key);

return Search(root->right, key);

}

BST\* minValueNode(BST\* node)

{

BST\* current = node;

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

current = current->left;

return current;

}

BST\* BST::deleteNode(BST\* root, int key)

{

if (root == NULL)

return root;

if (key < root->data)

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

else if (key > root->data)

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

else {

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

return NULL;

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

BST\* temp = root->right;

free(root);

return temp;

}

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

BST\* temp = root->left;

free(root);

return temp;

}

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

root->data = temp->data;

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

}

return root;

}

int main()

{

BST b, \*root = NULL;

root = b.Insert(root, 50);

b.Insert(root, 30);

b.Insert(root, 20);

b.Insert(root, 40);

b.Insert(root, 70);

b.Insert(root, 60);

b.Insert(root, 80);

b.Inorder(root);

cout<<"\nDeleting node with data 20\n";

root=b.deleteNode(root, 20);

b.Inorder(root);

cout<<"\nSearching node with data 80\n";

Search(root,80);

return 0;

}

**Output Screenshot:**

