//Assignment no 3

#include <iostream>

using namespace std;

class node

{

public:

int data;

node\* left;

node\* right;

};

class bst

{

public:

node\* root;

int cnt;

bst()

{

root=NULL;

cnt=0;

}

void insert();

void inorder(node\* temp);

void preorder(node\* temp);

void smallest();

void mirror(node\* root);

void largest();

int search(int key);

int height(node\* root);

};

void bst::insert()

{

node \*newnode,\*temp;

newnode=new node();

newnode->left=NULL;

newnode->right=NULL;

cout<<"Insert data in new node:";

cin>>newnode->data;

if(root==NULL)

{

root=newnode;

cout<<"Node inserted successfully."<<endl;

}

else

{

int flag=0;

temp=root;

while(flag==0)

{

if(newnode->data < temp->data)

{

if(temp->left==NULL)

{

temp->left=newnode;

cout<<"Node inserted successfully."<<endl;

flag=1;

}

else

{

temp=temp->left;

}

}

else if(newnode->data > temp->data)

{

if(temp->right==NULL)

{

temp->right=newnode;

cout<<"Node inserted successfully."<<endl;

flag=1;

}

else

{

temp=temp->right;

}

}

else

{

cout<<"Data already exists."<<endl;

flag=1;

}

}

}

}

void bst::inorder(node\* temp)

{

if(temp!=NULL)

{

inorder(temp->left);

cout<<temp->data<<" ";

cnt++;

inorder(temp->right);

}

}

void bst::preorder(node\* temp)

{

if(temp!=NULL)

{

cout<<temp->data<<" ";

preorder(temp->left);

preorder(temp->right);

}

}

void bst::smallest()

{

node \*temp;;

temp=root;

while(temp->left!=NULL)

{

temp=temp->left;

}

cout<<"Smallest element is:"<<temp->data<<endl;

}

void bst::largest()

{

node \*temp;;

temp=root;

while(temp->right!=NULL)

{

temp=temp->right;

}

cout<<"Largest element is:"<<temp->data<<endl;

}

void bst::mirror(node\* root)

{

node\* temp;

if(root!=NULL)

{

temp=root->left;

root->left=root->right;

root->right=temp;

}

}

int bst::search(int key)

{

node\* temp;

temp=root;

while(1)

{

if(key<temp->data)

{

if(temp->left != NULL)

{

temp=temp->left;

}

else

return 0;

}

else if(key>temp->data)

{

if(temp->right != NULL)

{

temp=temp->right;

}

else

return 0;

}

else

{

return 1;

}

}

}

int bst::height(node\* root)

{

int rh,lh;

if(root==NULL)

return 0;

else if(root->left==NULL && root->right==NULL)

return 0;

rh=height(root->right);

lh=height(root->left);

if(rh>lh)

return (rh+1);

else

return (lh+1);

}

int main()

{

bst obj;

int ch;

do

{

cout<<endl;

cout<<"\n\*\*MENU\*\*";

cout<<"\n1.Insert";

cout<<"\n2.Display inorder";

cout<<"\n3.Display preorder";

cout<<"\n4.Display Smallest element";

cout<<"\n5.Display Largest element";

cout<<"\n6.Display Mirror";

cout<<"\n7.Search";

cout<<"\n8.Height";

cout<<"\n9.Exit";

cout<<"\nEnter your choice:";

cin>>ch;

switch(ch)

{

case 1:

{

obj.insert();

break;

}

case 2:

{

cout<<"Inorder:";

obj.inorder(obj.root);

break;

}

case 3:

{

cout<<"Preorder:";

obj.preorder(obj.root);

break;

}

case 4:

{

obj.smallest();

break;

}

case 5:

{

obj.largest();

break;

}

case 6:

{

obj.mirror(obj.root);

cout<<"Mirror:";

obj.inorder(obj.root);

break;

}

case 7:

{

int key;

cout<<"Enter the key to be searched:";

cin>>key;

int result=obj.search(key);

if(result==1)

{

cout<<"Element Found."<<endl;

}

else

{

cout<<"Element Not Found."<<endl;

}

break;

}

case 8:

{

int result=obj.height(obj.root);

cout<<"Height:"<<result<<endl;

break;

}

case 9:

{

cout<<"End of Program."<<endl;

break;

}

default:

{

cout<<"Invalid choice!!"<<endl;

}

}

}while(ch!=9);

return 0;

}

output:

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ g++ Bst.cpp

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ ./a.out

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:1

Insert data in new node:5

Node inserted successfully.

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:1

Insert data in new node:1

Node inserted successfully.

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:1

Insert data in new node:10

Node inserted successfully.

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:2

Inorder:1 5 10

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:3

Preorder:5 1 10

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:4

Smallest element is:1

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:5

Largest element is:10

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:6

Mirror:10 5 1

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:7

Enter the key to be searched:10

Element Not Found.

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:8

Height:1

\*\*MENU\*\*

1. Insert
2. Display inorder
3. Display preorder
4. Display Smallest element
5. Display Largest element
6. Display Mirror
7. Search
8. Height
9. Exit

Enter your choice:9

End of Program.

[gescoe@gescoe-OptiPlex-3010](mailto:gescoe@gescoe-OptiPlex-3010):~/Desktop/SE-A-55$

### Theory for Binary Search Tree (BST)

#### 1. ****Introduction****

A **Binary Search Tree (BST)** is a type of **binary tree** in which each node follows the **left child** being smaller and the **right child** being greater than the parent node. The structure of the BST helps in efficient searching, insertion, and deletion operations. The properties of BST make it a suitable data structure for dynamic ordered collections where data can be added, deleted, or searched quickly.

#### 2. ****Properties of BST****

* Every node has at most two children: a left child and a right child.
* The left subtree of a node contains only nodes with values smaller than the node's value.
* The right subtree of a node contains only nodes with values greater than the node's value.
* Both the left and right subtrees of each node are also binary search trees.

#### 3. ****Operations in BST****

##### ****a. Insertion****

Insertion in a BST starts from the root and follows these steps:

1. Compare the value to be inserted with the current node's data.
2. If the value is smaller, move to the left child.
3. If the value is larger, move to the right child.
4. Repeat the comparison until an empty spot (NULL) is found, where the new node is inserted.

**Time Complexity**: O(h), where **h** is the height of the tree. In the worst case (when the tree becomes a skewed tree), the height can be O(n), where **n** is the number of nodes.

##### ****b. Inorder Traversal****

Inorder traversal of a BST visits nodes in ascending order:

1. Visit the left subtree.
2. Visit the current node.
3. Visit the right subtree.

This traversal gives us the elements of the BST in sorted order.

**Time Complexity**: O(n), where **n** is the number of nodes in the tree.

##### ****c. Preorder Traversal****

Preorder traversal visits nodes in the order:

1. Visit the current node.
2. Visit the left subtree.
3. Visit the right subtree.

This can be useful for creating a copy of the tree or evaluating expressions stored in the tree.

**Time Complexity**: O(n), where **n** is the number of nodes in the tree.

##### ****d. Search****

To search for a value in a BST:

1. Start from the root node.
2. If the value is smaller than the current node, move to the left child.
3. If the value is greater, move to the right child.
4. Repeat the process until the value is found or a NULL node is encountered.

**Time Complexity**: O(h), where **h** is the height of the tree. In the worst case (skewed tree), the height can be O(n).

##### ****e. Find Smallest and Largest Element****

* The smallest element is the leftmost node in the BST (the node with no left child).
* The largest element is the rightmost node in the BST (the node with no right child).

**Time Complexity**: O(h), where **h** is the height of the tree. In the worst case, **h** can be O(n).

##### ****f. Mirror****

The mirror operation inverts the tree, swapping the left and right subtrees of every node.

**Time Complexity**: O(n), as every node must be visited and swapped.

##### ****g. Height****

The height of the tree is the length of the longest path from the root to a leaf node.

**Time Complexity**: O(n), as every node must be visited to find the maximum depth.

#### 4. ****Algorithm for Each Operation****

##### ****a. Insert Algorithm****

1. Create a new node with the given value.

2. If the tree is empty, set the new node as the root.

3. Otherwise, compare the value with the current node:

- If the value is smaller, move to the left child.

- If the value is greater, move to the right child.

4. Repeat step 3 until a NULL child is found, where the new node is inserted.

##### ****b. Inorder Traversal Algorithm****

1. If the current node is NULL, return.

2. Traverse the left subtree.

3. Visit the current node (print or store the value).

4. Traverse the right subtree.

##### ****c. Preorder Traversal Algorithm****

1. If the current node is NULL, return.

2. Visit the current node (print or store the value).

3. Traverse the left subtree.

4. Traverse the right subtree.

##### ****d. Search Algorithm****

1. Start from the root node.

2. If the value is smaller than the current node’s data, move to the left child.

3. If the value is greater, move to the right child.

4. If the value is equal to the current node’s data, return true.

5. If the node is NULL, return false.

##### ****e. Smallest Element Algorithm****

1. Start from the root node.

2. Keep moving to the left child until a NULL left child is encountered.

3. The current node is the smallest element.

##### ****f. Largest Element Algorithm****

1. Start from the root node.

2. Keep moving to the right child until a NULL right child is encountered.

3. The current node is the largest element.

##### ****g. Mirror Algorithm****

1. If the current node is NULL, return.

2. Swap the left and right children of the current node.

3. Recursively apply the mirror operation to the left and right subtrees.

##### ****h. Height Algorithm****

1. If the current node is NULL, return 0.

2. Recursively calculate the height of the left and right subtrees.

3. Return the maximum of the two heights plus 1 (for the current node).

#### 5. ****Time Complexity****

* **Insertion, Search, Find Min/Max**: O(h), where **h** is the height of the tree.
  + In the worst case (skewed tree), the height can be O(n), making these operations O(n).
* **Traversal (Inorder/Preorder)**: O(n), as all nodes are visited.
* **Mirror and Height Calculation**: O(n), as all nodes are processed.

#### 6. ****Advantages of BST****

* **Efficient Searching**: Due to its ordered nature, searching for an element is faster than in unsorted data structures.
* **Dynamic Size**: The size of the BST can dynamically grow or shrink as nodes are inserted or deleted.
* **Efficient Operations**: In a balanced BST, all operations (insert, search, delete) are O(log n), which is much faster than linear data structures.

#### 7. ****Disadvantages of BST****

* **Unbalanced Tree**: If the BST becomes unbalanced (like a linked list), the height can increase significantly, leading to poor performance (O(n) instead of O(log n)).
* **Complex Balancing**: Maintaining a balanced tree (such as in AVL trees or Red-Black trees) requires extra work and complexity.

#### 8. ****Conclusion****

A Binary Search Tree is a powerful data structure that supports fast search, insertion, and deletion. However, its performance depends on the tree's balance, so maintaining a balanced tree (e.g., using AVL or Red-Black trees) is important for achieving optimal performance.