Assignment No: A- 2

Aim: To study BST and operations on it.

Title: - Crete Binary Search Tree and perform basic operations on it

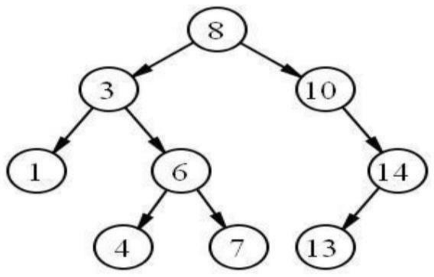
Problem Statement: Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree - i. Insert new node ii. Find number of nodes in longest path iii. Minimum data value found in the tree
iv. Change a tree so that the roles of the left and right pointers are swapped at every node
v. Search a value

Theory: **Binary Search Tree:** It is a binary Tree with following Properties.
- The left subtree of a node contains only nodes with keys less than the node's key.
- The ri ght subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Example:

Consider following binary search tree



OPERATIONS:
Operations on a binary search tree require comparisons between nodes.

1] Searching:
Searching a binary search tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method.
We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

2] Insertion:

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value.
In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root..

3] Deletion:
There are three possible cases to consider:

- Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.
- Deleting a node with one child: Remove the node and replace it with its child. Deleting a
- node with two children: Call the node to be deleted *N*. Do not delete *N*. Instead, choose **either its in-order successor node or its in-order predecessor node,** *R*. Replace the value of *N* with the value of *R*, then delete *R*.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

Algorithms:

Algorithm Create ( )
// This is used to Create Binary tree with no input and return nothing.

```
1.  {
2.  repeat
3.  {
4.  Read data element from user in „Value‟;
5.  Create new node;
6.  temp=new node;
7.  temp->data=value;
8.  temp->left=NULL;
9.  temp->right=NULL;
10. if(root is NULL) then        // Tree is Empty.
11. {
12. root=temp;
13. }
14. else   // Add new node into existing tree.
15. {
16. Insert(root,temp);
17. }
18. }until (false);
19. }
```

Algorithm Insert (root, temp)
// **This is used to add new node in existing tree pointed by root. It returns nothing.**

```
1.  {
2.  if(root is not NULL)
3.  {
4.  if( root->data > temp->data ) then        // Left Sub tree
5.  {
6.  if(temp->left is NULL) then    // left child is NULL
7.  temp->left=temp;
8.  else
9.  insert(temp->left, temp); // Check next left link
10. }
11. else
12. {
13. if(ch is Right side) then          // Right Sub tree
14. {
15. if(temp->right is NULL) then  // Right child is NULL
16. temp->right=temp;
17. else
18. insert(temp->right, temp); // Check next right link
19. }
20. }
21. else
22. Write (No tree Exist);
23. }
```

Algorithm Preorder (temp)
// **This is used to display the nodes in VLR way**

```
1. {
2. if(temp is not NULL) then
3. {
4. Write (temp->data);
5. Call Preorder(temp->lptr);
6. Call Preorder(temp->rptr);
7. }
8. }
```

Algorithm Inorder (temp)
// **This is used to display the nodes in LVR way**

```
1. {
2. if(temp is not NULL) then
3. {
```

```
4.      Call Inorder(temp->lptr);
5.      Write (temp->data);
6.      Call Inorder(temp->rptr);
7.      }
8.      }
```

Algorithm Postorder (temp)

// **This is used to display the nodes in LRV way**

```
1.      {
2.      if(temp is not NULL) then
3.      {
4.      Call Inorder(temp->lptr);
5.      Call Inorder(temp->rptr);
6.      Write (temp->data);
7.      }
8.      }
```

Algorithm Search (root, val, parent)

// This algorithm is used to Search an element in BST. It takes three parameters //namely root, value to be search and parent as input. It returns NULL of parent if //data not found or found respectively.

```
1.      {
2.      if(root is not NULL) then         // tree exist.
3.      {
4.      if(root->data > val) then         // search in Left subtree.
5.      search(root->llink,val,root);
6.      else if(root->data<val) then      // search in Right subtree
7.      search(root->rlink,val,root);
8.      if(root->data==val) then          // successful search
9.      {
10.     Write (Val is found successfully);
11.     return parent;
12.     }
13.     }
14.     else     // unsuccessful search
15.     return NULL;
16.     }
```

Algorithm Insert ()

// This algorithm is used to add new data into existing BST.

// This is unlike Insert in Create function, BST already exist. So read data from user //and call Insert (root, temp)

Algorithm Smallest(Node* root)
// This algorithm finds Minimum data value found in the tree

```
1. {
2. while (root->ln != NULL)
3. {
4. root = root->ln;
5. }
6. print root->key
7. }
```

Algorithm LongestPath(Node* root)
// This algorithm Find number of nodes in longest path

```
1.     {
2.     if(root==NULL)
3.     return 0;
4.     L = longestPath(root->ln);
5.     R = longestPath(root->rn);
6.     if(L>R)
7.     return (L+1);
8.     else return (R+1);
9.     }
```

Algorithm swapNodes(Node* root)
// This algorithm Change a tree so that the roles of the left and right pointers are swapped at every node

```
{
  1. Node* temp;
  2. if(root==NULL)
  3. return NULL;
  4. temp = root->ln;
  5. root->ln=root->rn;
  6. root->rn=temp;
  7. swapNodes(root->ln);
  8. swapNodes(root->rn);
}
```

Algorithm De le ( )
// This algorithm is used to delete an element from BST. It takes no input (explicitly) //and returns nothing

```
1.          {
2.          Read Data to be deleted in “val”;
3.          parent = Search(root,val,NULL);   // check element exist in BST or NOT.
```

```
4.      if( parent is NULL and root->data is not EQUAL to val)
5.      Write (“Value to be deleted doesn‟t exist. Cant Delete”)
6.      else
7.      {
8.      if(parent->llink->data = val)      // Data exist to left of parent
9.      {
10.     flag='l';
11.     temp=parent->llink;
12.     }
13.     else
14.     {
15.     flag='r';     // Data exist to right of parent
16.     temp=parent->rlink;
17.     }
18.     if( (temp->llink is not NULL && temp->rlink is not NULL)) // having two
        kids
19.     {
20.     if (root->data = val) then

21.     temp=root;
22.     temp1=temp->rlink;
23.     temp2=temp1;
24.     while (temp1->llink is not NULL)        // look for inorder successor.
25.     {
26.     temp2=temp1;
27.     temp1=temp1->llink;
28.     }
29.     temp->data=temp1->data;
30.     temp2->llink=NULL;
31.     delete(temp1);      // delete node
32.     }
33.     else if(temp->llink is not NULL && temp->rlink is NULL) // having left
        kid
34.     {
35.     if(flag = 'l')
36.     parent->llink=temp->llink;
37.     else
38.     parent->rlink=temp->llink;
39.     delete(temp);       // delete node
40.     }
41.     else if(temp->llink is NULL && temp->rlink is not NULL) // having right
        child
{
if(flag= 'l')
parent->llink=temp->rlink;
else
parent->rlink=temp->rlink;
delete(temp);          // delete node
}
else if(temp->llink is NULL && temp->rlink is NULL) // node is leaf
{
if(flag = 'l')
parent->llink=NULL;
else
parent->rlink=NULL;
delete(temp);          // delete node
}
Write(" val is deleted successfully");
}
}
```

<u>Frequently asked questions</u>

1. **What is Binary Search Tree (BST)?**
2. **Define the property of the Binary Search Tree?**
3. **What is the difference between the Binary Search tree & Binary Tree?**
4. **Comment on "Searching is Faster in Binary Search Tree than Binary tree" 5. What are the advantages of Binary Search Tree?**
6. **Why do we need a binary Search Tree?**
7. **How searching is perform in Binary Search tree?**
8. **Which traversal method gives data elements as sorted output?**
9. **How many different binary trees and binary search trees can be made from three Nodes that contain the key values 1, 2 & 3?**
10. **What is Self Balancing Binary Search Tree?**

Flowchart:

Conclusions: