Batch: B1 **Roll No.: 1711072**

Experiment No. 5

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Binary Search Tree (BST)

Objective: Implementation of BST and Binary Tree Traversal Techniques.

Expected Outcome of Experiment:

CO	Outcome
CO3	Demonstrate searching and sorting methods

Books/ Journals/ Websites referred:

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein; (CLRS)
"Introduction to Algorithms", Third Edition, The MIT Press.

Abstract:-

DATA STRUCTURE:

1. int data //to store value of int type.
2. node* left //to point to the left node of the parent.
3. node* right //to point to the right node of the parent.

OPERATIONS:

1. **node* create(int data)**

   This function is used to create a new node each time the function is called.

2. **void insert(node* r, int data)**

   This function is used to insert element entered by user at appropriate position in the binary search tree.

3. **void inorder(node* parent)**

   This function is used to display the elements of the binary search tree in inorder form.

4. **void preorder(node* parent)**

   This function is used to display the elements of the binary search tree in preorder form.

5. **void postorder(node* parent)**

   This function is used to display the elements of the binary search tree in postorder form.

6. **node* search(node* root, int key)**

   This function is used to search for an element in the binary search tree.

7. **node* minimum(node* parent)**

   This function is used to find the smallest child for a particular parent.

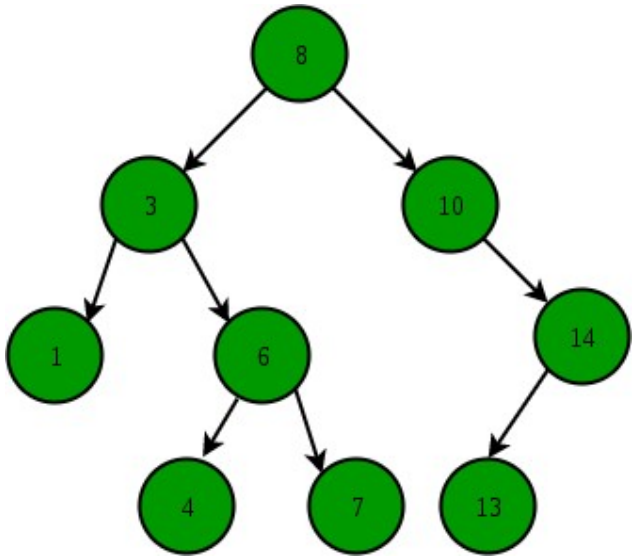
8. **node* deleteNode(node* parent, int key)**

   This function is used to delete an element from the binary search tree.

Related Theory: -

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key
To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Steps to search in below tree:
1. Start from root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If element to search is found anywhere, return true, else return false.

Insertion of node
A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

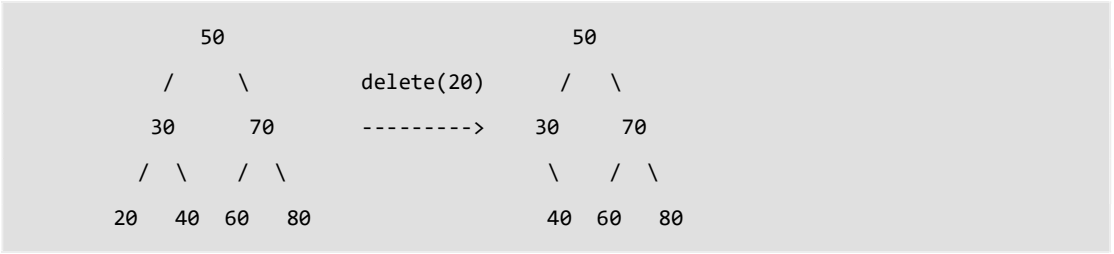
```
      100                                100
       /   \        Insert 40             /    \
     20     500    --------->           20     500
    /  \                               /  \
   10   30                            10   30
                                            \
                                            40
```

Time Complexity: The worst case time complexity of search and insert operations is O(h) where 'h' is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

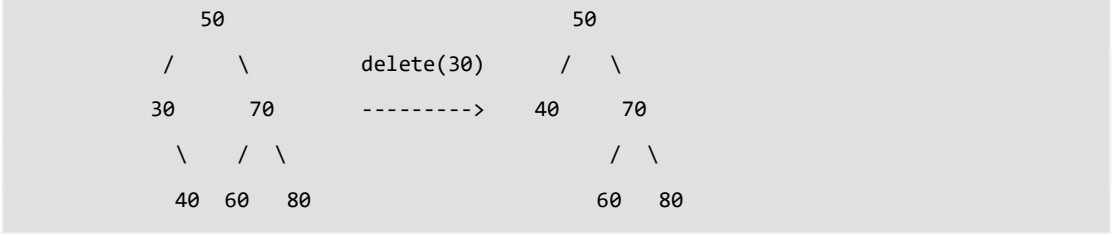
Delete a node:

When we delete a node, three possibilities arise.

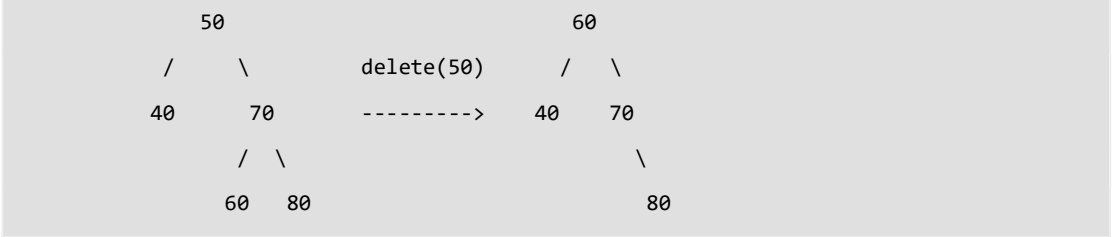
1) ***Node to be deleted is leaf:*** Simply remove from the tree.



2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Time Complexity: The worst case time complexity of delete operation is O(h) where 'h' is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n).

Some interesting facts about BST:

- Inorder traversal of BST always produces sorted output.
- We can construct a BST with only Preorder or Postorder or Level Order traversal. Note that we can always get inorder traversal by sorting the only given traversal.
- Number of unique BSTs with n distinct keys is Catalan Number.

Implementation Details:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
  int data;
  struct node* left;
  struct node* right;
}node;
node *root=NULL;
node* deleteNode(node*, int);

node* create(int data){
    node* temp;
    temp=(node*)malloc(sizeof(node));
    int key=data;
    temp->data=key;
    temp->left=temp->right=NULL;
    return temp;
}

void insert(node *r, int data){
  node* temp=create(data);
  if(r==NULL){
    //temp=create(data);
    root=temp;
    //return root;
  }
  else{
  if(data < r->data){
    if(r->left==NULL){
    //temp=create(data);
    r->left=temp;
    }
    else
      insert(r->left, data);
  }
```

```
  else if(data>r->data){
    if(r->right==NULL){
      //temp=create(data);
      r->right=temp;
    }
    else
      insert(r->right, data);
  }
}
}

void inorder(node *parent){
  if(parent==NULL)
    return;
  else if(parent!=NULL)
  {
    inorder(parent->left);
    printf("%d ", parent->data);
    inorder(parent->right);
  }
  else
  return;
}

void preorder(node *parent){
  if(parent==NULL)
    return;
  else if(parent!=NULL){
    printf(" %d", parent->data);
    preorder(parent->left);
    preorder(parent->right);
  }
}

void postorder(node* parent){
  if (parent==NULL)
    return;
  else if(parent!=NULL){
    postorder(parent->left);
    postorder(parent->right);
    printf(" %d", parent->data);
    }
}
```

```
node* search(node* root, int key){
  if(root==NULL){
    printf("No such element.");
    return root;
  }
  else if(root->data==key){
    printf("Element found!");
  }
  else if(root->data>key){
    return search(root->left, key);
  }
  else{
    return search(root->right, key);
  }
}

node* minimum(node* parent){
  if(parent->left==NULL)
    return parent;
  else
    return minimum(parent->left);
}

node* deleteNode(node* parent, int key){
  //node* temp=parent;
  if(parent==NULL)
    return NULL;
  else if(key<parent->data)
    parent->left=deleteNode(parent->left,key);
  else if(key>parent->data)
    parent->right=deleteNode(parent->right, key);
  else{
    if(parent->left==NULL && parent->right==NULL)
      parent=NULL;
    else if(parent->right==NULL)
      parent=parent->left;
    else if(parent->left==NULL)
      parent=parent->right;
    else{
      node* temp=minimum(parent->right);
      parent->data=temp->data;
      parent->right=deleteNode(parent->right, temp->data);
    }
  }
```

```
  return parent;
}

void main(){
  int choice=0;
  int data, key, element;
  while(choice!=7){
    printf("\n1. Insert\n2. Inorder traversal\n3. Postorder
Traversal\n4. Preorder Traversal\n5. Search\n6. Delete\n7.
Exit\nEnter a choice: ");
    scanf("%d", &choice);
    switch(choice){
      case 1:
        printf("Enter the data node: ");
        scanf("%d", &data);
        insert(root, data);
        break;
      case 2:
        printf("Inorder traversal: ");
        inorder(root);
        break;
      case 3:
        printf("Postorder Traversal: ");
        postorder(root);
        break;
      case 4:
        printf("Preorder Traversal: ");
        preorder(root);
        break;
      case 5:
        printf("Enter a number to search for: ");
        scanf("%d", &key);
        search(root,key);
        break;
      case 6:
        printf("Enter the number to delete: ");
        scanf("%d", &element);
        root=deleteNode(root,element);
        break;
      case 7:
        exit(1);
      default:
        printf("Enter valid choice.");
    }
```

```
  }
}
```

For verification, my code is available on:
https://repl.it/@ARGHYADEEPDAS/BST

OUTPUT SCREENS:

```
1. Insert
2. Inorder traversal
3. Postorder Traversal
4. Preorder Traversal
5. Search
6. Delete
7. Exit
Enter a choice:  1
Enter the data node:  20
Enter a choice:  1
Enter the data node:  10
Enter a choice:  1
Enter the data node:  30
Enter a choice:  1
Enter the data node:  15
Enter a choice:  1
Enter the data node:  25
Enter a choice:  1
Enter the data node:  8
```

```
1. Insert
2. Inorder traversal
3. Postorder Traversal
4. Preorder Traversal
5. Search
6. Delete
7. Exit
Enter a choice:  2
Inorder traversal: 8 10 15 20 25 30
Enter a choice:  3
Postorder Traversal:  8 15 10 25 30 20
Enter a choice:  4
Preorder Traversal:  20 10 8 15 30 25
```

```
Enter a choice:  5
Enter a number to search for:  15
Element found!
```

```
Enter a choice:  6
Enter the number to delete:  15
```

```
Enter a choice:  2
Inorder traversal: 8 10 20 25 30
Enter a choice:  5
Enter a number to search for:  15
No such element.
```

CONCLUSION:

The program ran successfully as we were able to implement binary search tree and perform various traversal techniques and implement other functionalities like search, delete and insert and run them for various test cases.