Name : Mufaddal Diwan
Class : SE-3
Roll no : 21340

ASSIGNMENT NO. 2

TITLE To write a program for implementing dictionary using binary search tree.

PROBLEM STATEMENT /DEFINITION A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry, assign a given tree into another tree(=). Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.

OBJECTIVE

- To understand Binary Search Tree implementation.
- To understand operation performed on tree such as addition, deletion, updating of keywords.
- To connect one tree to another tree
- To understand sorting of tree.

OUTCOME

- Dictionary implementation using Binary Search Tree
- Various operations performed on tree

S/W PACKAGES AND HARDWARE APPARATUS USED

- 64-bit Open source Linux or its derivative.
- Open Source C++ Programming tool like G++/GCC.

Concepts related Theory:

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items,

and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

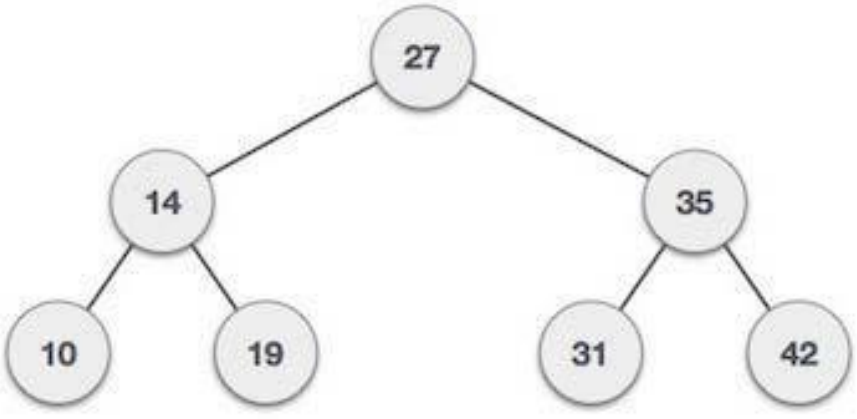
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent

node's key. Thus, BST divides all its sub-trees into two segments;

Representation:

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations:

Following are the basic operations of a tree −

- Search − Searches an element in a tree.
- Insert − Inserts an element in a tree.

Node:- Define a node having some data, references to its left and right child nodes.

Search Operation:

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm:

```
struct node* search(int data){
  struct node *current = root;
  printf("Visiting elements: ");
  while(current->data != data){
    if(current != NULL) {
      printf("%d ",current->data);
      //go to left tree
      if(current->data > data){
        current = current->leftChild;
      }//else go to right tree
      else {
        current = current->rightChild;
      }
```

```
      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }
  return current;
}
```

Insert Operation:

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;

struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;
  //if tree is empty
 if(root == NULL)
                    {
 root = tempNode;
```

```
} else { current
  = root; parent
  = NULL;
  while(1) {
    parent = current;
    //go to left of the
    tree if(data < parent-
    >data) {
      current = current-
      >leftChild; //insert to
      the left if(current ==
      NULL) {
        parent->leftChild =
        tempNode; return;
      }
    }//go to right of the
    tree else {
      current = current-
      >rightChild; //insert to
      the right if(current ==
      NULL) {
        parent->rightChild = tempNode;
        return;
      }
```

```
      }
    }
  }
}
```

Deletion:

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node 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. Copy the value of R to N, then recursively call delete on the original R until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of the first 2 cases.

Algorithm:

```
bool BinarySearchTree::remove(int value) {
    if (root == NULL)
        return false;
   else {
        if (root->getValue() == value) {
            BSTNode auxRoot(0);
            auxRoot.setLeftChild(root);
            BSTNode* removedNode = root->remove(value, &auxRoot);

            root = auxRoot.getLeft();
            if (removedNode != NULL) {
```

```
            delete removedNode;
            return true;
        } else
            return false;
    } else {
        BSTNode* removedNode = root->remove(value,
        NULL); if (removedNode != NULL) {
            delete removedNode;
            return true;
        } else
            return false;
       }
  }
}
 BSTNode* BSTNode::remove(int value, BSTNode
                          *parent) {
  if (value < this->value) {
      if (left != NULL)
          return left->remove(value, this);
       else
          return NULL;
  } else if (value > this-
      >value) { if (right !=
      NULL)
          return right->remove(value,
      this); else
          return NULL;
```

```
        } else {
            if (left != NULL && right != NULL) {
                this->value = right->minValue();
                return right->remove(this->value, this);
            } else if (parent->left == this) {
                parent->left = (left != NULL) ? left : right;
                return this;
            } else if (parent->right == this) {
                parent->right = (left != NULL) ? left : right;
                return this;
            }
        }
    }
    int BSTNode::minValue() {
        if (left == NULL)
            return value;
        else
            return left->minValue();}
```

Threaded Binary Tree:

Threaded binary tree is a binary tree variant that allows fast traversal: given a pointer to a node in a threaded tree, it is possible to cheaply find its in-order successor (and/or predecessor).

Binary trees, including (but not limited to) binary search trees and their variants, can be used to store a set of items in a particular order. For example, a binary search tree assumes data items are somehow ordered and maintain this ordering as part of their insertion and deletion algorithms. One useful operation on such a tree is traversal: visiting the items in the order in which they are stored (which matches the underlying ordering in the case of BST).

Algorithm:

1. For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.

2. Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.

3. Print the node and If node has right child then go to step 4 else go to step 5.

4. Make right child as current node.

5. if there is a thread node then make it the current node.

6. if all nodes have been printed then END else go to step 1.

Test-Cases

Description	Input	Output	Result
Create Dictionary	Key Value D 4 C 3 E 5 F 6 B 2		Pass
Display in Ascending order	-	Key Value B 2 C 3 D 4 E 5 F 6	Pass
Display in descending order	-	Key Value F 6 E 5 D 4 C 3	Pass

		B 2	
Update	Enter key: F Enter value: 9	Updated Successfully	Pass
Delete	Enter key: F Enter key: F	Deleted key: F, value: 9 Key F not found!	Pass

Conclusion : Thus we have successfully implement Binary Search Tree and performed all operations on it.