AIM: To study dictionary and perform different operations on it using binary search tree.

INDEX TERMS: class, objects, keyword, and methods, binary search tree

PROBLEM DEFINITION:

A Dictionary stores keywords & its meanings. Now provide facility for adding new keywords, deleting keywords, & updating values of any entry. Also provide facility to display whole data sorted in ascending/ Descending order, Also we have to find how many maximum comparisons may require for finding any keyword.

THEORY:

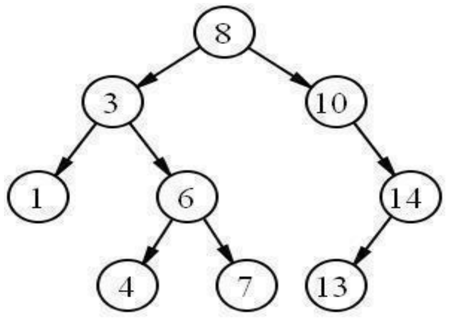
BINARY SEARCH TREE:

In computer science, a binary search tree (BST), which may sometimes also be called an ordered or sorted binary tree, is a node -based binary tree data structure.

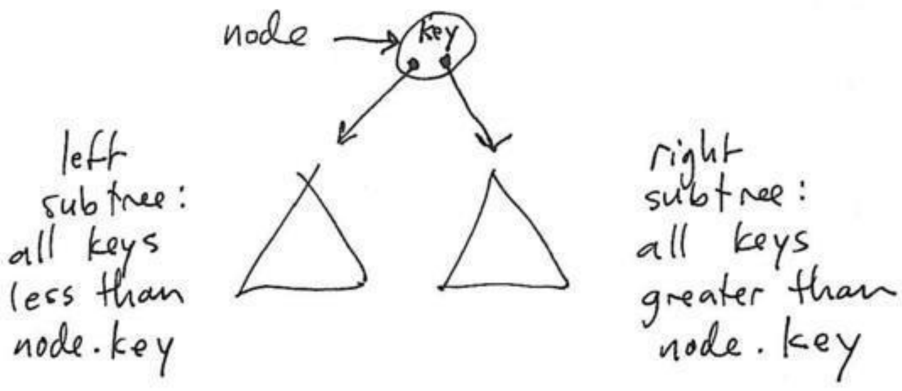
The Binary Search Tree which has the following properties:

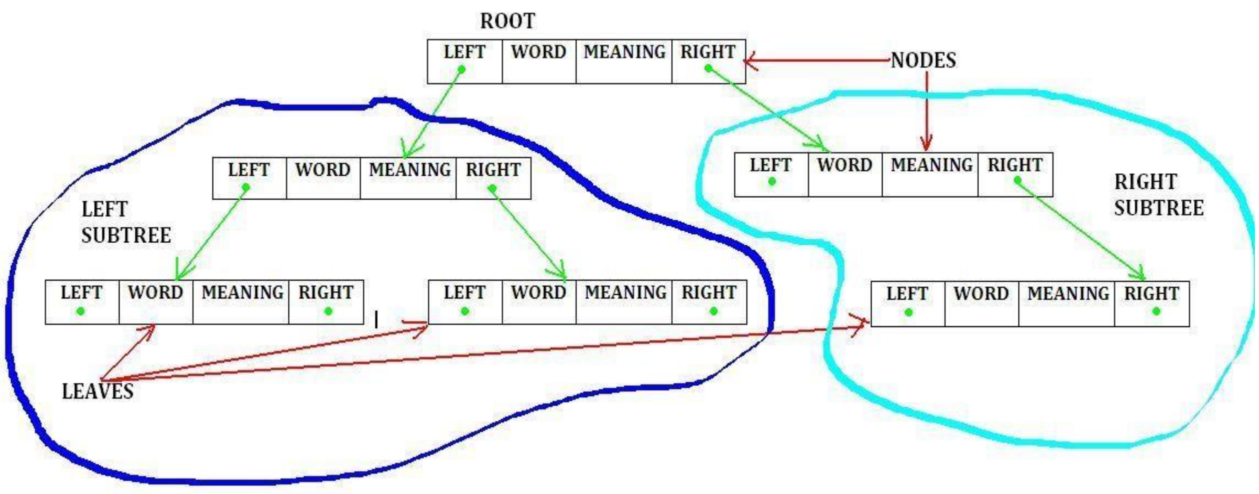
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

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



IMPLEMENTATION POINT OF VIEW:


node
key
left subtree: all keys less than node.key
right subtree: all keys greater than node.key



BST is implemented using array as well as Linked organization. Linked structure of BST is shown in above figure. The figure consists of root, left subtree and right subtree. Each element of BST satisfies the following property,

So while inserting new element we compare with current root. If element is smaller than current root then it will go into its left subtree else will go in right subtree.

class node class BST

```
{
    public:
            char word[20];
            char meaning[20];
            node *lptr;
            node *rptr;
};
```

```
{
    public:
            node *root;
            void Create();
            void Insert(node*, node*);
             void Preorder(node*);
             void Inorder(node*);
             void Postorder(node*);
            void descending_order(node *);
            node* Search(node*,int,node*);
            BST( )
            {
             root =Null;
            }
            };
```

OPERATIONS ON DICTIONARY USING BINARY SEARCH TREE

1. **Construct DICTIONARY from input (Read key and meaning to create a dictionary) /* procedure Create */**
2. **Recursive Preorder traversal /* procedure Preorder */**
3. **Recursive Inorder traversal /* procedure Inorder */**
4. **Recursive Postorder traversal /* procedure Postorder */**
5. **Search in word in dictionary using Binary Search tree /* procedure Search */**
6. **Print a Dictionary in ascending order /* procedure Inorder */**
7. **Print a Dictionary in descending order /* procedure descending_order */**
8. **Exit**

Algorithm Create ( ) // this is used to Create Dictionary using BST with no input and return nothing.

```
1 .{
2.          repeat
3.          {
4.          //read the input in character array to create a DICTIONARY
5.          Create new node;
6.  temp=new node;
7.  Read data element from user in ‘key’;
8.  temp->word;
9.  Read ‘meaning’ of entered key;
10. temp->meaning;
11. temp->left=NULL;
12. temp->right=NULL;
13. if(root is NULL) then // Tree is Empty.
14. {
15. root=temp;
16. }
17. else // Add new node into existing tree.
18. {
19. Insert(root,temp);
20. }
21. }until (false);
22. }
```

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.  l1=strlen(root->word); // calculate length of key
5.  l2=strlen(temp->word); // calculate length of second key
6.  if(l1>l2)
7.  l=l2;
8.  else
9.  l=l1;
10. for(int i=0;i<l;i++)
11. {
12. if( root->word[i] > temp->word[i] ) then // Left Sub tree
13. {
14. if(temp->left is NULL) then // left child is NULL
15. {
16. temp->left=temp;
```

```
                break;
            }
            else
                insert(temp->left, temp); // Check next left link
        }
        else
        {
            if(temp->right is NULL) then // Right child is NULL
            {
                temp->right=temp;
                break;
            }
            else
                insert(temp->right, temp); // Check next right link
        }
    } // end of loop
}
else
    Write (No tree Exist);
}
```

//Recursive Preorder traversal

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

```
{
    if(temp is not NULL) then
    {
        Write (temp->data);
        Call Preorder(temp->lptr);
        Call Preorder(temp->rptr);
    }
}
```

//Recursive Inorder traversal

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

```
{
    if(temp is not NULL) then
    {
        Call Inorder(temp->lptr);
        Write (temp->data);
        Call Inorder(temp->rptr);
    }
}
```

//Recursive Postorder traversal

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.    }
```

Print a Dictionary in ascending order
Algorithm inorder (temp) // This algorithm is used to print existing dictionary in ascending order.

Print a Dictionary in descending order
//Recursive traversal method to print dictionary in descending order
Algorithm descending_order (temp) // This is used to display t he nodes in RVL way

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

Algorithm Search (root, temp, 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.     l1=strlen(root->word); // calculate length of key
5.     l2=strlen(temp->word); // calculate length of second key
6.     if(l1>l2)
7.     l=l2;
8.     else
9.     l=l1;
10.    for(int i=0;i<l;i++)
11.    {
12.    if(root->word[i] > temp->word[i]) then // search in Left subtree.
13.    search(root->llink,temp,root);
6.     else if(root->word[i]<temp->word[i]) then // search in Right subtree
search(root->rlink,temp,root);
if(strcmp(root->word==temp->word)==0) then // successful search
{
Write (Val is found successfully);
return parent;
}
}
else // unsuccessful search
return NULL;
}
```

Frequently asked questions:

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 co ntain the key values 1, 2 & 3?
10) What is Self Balancing Binary Search Tree?

CONCLUSION: Thus, We have studied and implemented the dictionary using Binary Search Tree and performed different operations on it.