ASSINGMENT NO.	**1**
TITLE	Binary Search Tree
PROBLEM STATEMENT /DEFINITION	A dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. 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. **OR** Beginning with an empty binary search tree. Construct the binary search tree by inserting the values in given order. After constructing binary search tree perform following operations 1) Insert a new node 2) Find numbers of node in longest path 3) Minimum and maximum data value found in tree 4) Change a tree so that the roles of the left and right pointers are swapped at every node 5)Search an element
OBJECTIVE	To understand practical implementation and usage of binary search tree for solving the problems.
OUTCOME	After successful completion of this assignment, students will be able to implement and use binary search tree for efficient solution to problems
S/W PACKAGES AND HARDWARE APPARATUS USED	• (64-bit)64-BIT Fedora 17 or latest 64-bit update of equivalent Open source OS • Programming Tools (64-bit) Latest Open source update of Eclipse Programming frame work
REFERENCES	1. E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 2. Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. and title 3. Problem definition 4. Learning Objective 5. Learning Outcome 6. Software / Hardware requirement 7. Concepts related Theory 8. Algorithms/ Pseudo Code 9. Class ADT 10. Test cases 11. Conclusion/Analysis

Prerequisites:

- Basic knowledge of linked list, its operations and its implementation, searching techniques
- Object oriented programming features

Learning Objectives:

- To understand practical implementation and usage of binary search tree for solving the problems

Learning Outcomes:

- After successful completion of this assignment, students will be able to implement and use binary search tree for efficient solution to problems

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 finding items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

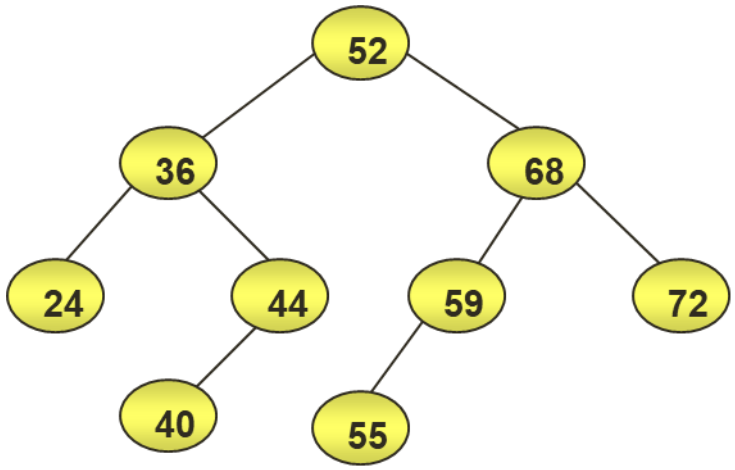
Binary search tree is a binary tree in which every node satisfies the following conditions:

- All values in the left subtree of a node are less than the value of the node.
- All values in the right subtree of a node are greater than the value of the node.
- Every node has a key and no two elements have same keys
- The left and right subtrees are also binary search tree.

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, the key is first compared with root node key if it is not equal then we see whether the key is less or more than root key, depending on that we need to decide whether we will move to left or right subtree and continue the same process until key is found.

Following is a pictorial representation of BST –



We observe that the root node key (52) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree. The same property is observed for every node.

Basic Operations:
Following are the basic operations of a tree −

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

Node:- Define a node having some data along with references to its left and right child nodes. Here data can be integer or string or any other type.
For dictionary the node consists of four fields (word, meaning, right pointer, left pointer)

ADT:

```
 class Node
  {
    private:
    string word,
    string meaning;
    Node *left,*right;
    public:
    Node(word, meaning);
    friend class BSTree;
 };
class BSTree
{
Node *root;
```

```
public:
//Binary search Tree operations
Node * search(word);
insert(word,meaning);
display_ascending();
display_descending();
update(oldword, new_meaning);
search(word);
deleteword(word);
}
```

Algorithms and Pseudo code:

Search Operation:

Whenever you search a word in a dictionary, your brain applies the mechanics of a binary search tree. To look the meaning of a word, you randomly open any page of the dictionary. Now depending upon the word to be searched, you turn the pages left or right. This is same as that of a binary search tree

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.

To search for a specific value, you need to perform the following steps:

Algorithm:

1. Make currentNode point to the root node
2. If currentNode is NULL:
    a. Display “Not Found”
    b. Exit
3. Compare the value to be searched with the value of currentNode. Depending on the result of the comparison, there can be three possibilities:
    a. If the value is equal to the value of currentNode:
        i. Display “Found”
        ii. Exit
    a. If the value is less than the value of currentNode:
        i. Make currentNode point to its left child
        ii. Go to step 2
    a. If the value is greater than the value of currentNode:
        i. Make currentNode point to its right child
        ii. Go to step 2

Pseudo Code:

```
Node* search(data){
        current = root;
        while(current->data != data){
                if(current != NULL) {
                        //go to left tree
                          if(current->data > data){
                                current = current->leftChild;
                        }//else go to right tree
                else {
                        current = current->rightChild;
                  }
                 //data not found
                 if(current == NULL){
                        print("Data not found")
                        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:

1. Allocate memory for the new node.
2. Assign value to the data field of new node.
3. Make the left and right child of the new node point to NULL.
4. Locate the node which will be the parent of the node to be inserted. Mark it as parent.
5. If parent is NULL (Tree is empty):
    a. Mark the new node as ROOT
    b. Exit
6. If the value in the data field of new node is less than that of parent:
    a. Make the left child of parent point to the new node
    b. Exit
7. If the value in the data field of the new node is greater than that of the parent:
    a. Make the right child of parent point to the new node
    b. Exit

Pseudo Code:

```
algorithm insert(data) {
```

```
 tempNode=new Node(data)  //allocate memory for new item datafield=data, rptr=lptr=NULL

  if(root == NULL) { // if tree is empty
    root = tempNode;
  } else {
    current = root;
    parent = NULL; //parent will point to parent node
    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:

Case I: Algorithm to delete a node with no child (leaf node):

1. Locate the node to be deleted. Mark it as currentNode and its parent as parent.

2. If currentNode is the root node: // **If parent is NULL**
    a. Make ROOT point to NULL.
    b. Go to step 5.
3. If currentNode is left child of parent:
    1. Make left child field of parent point to NULL.
    2. Go to step 5.
4. If currentNode is right child of parent:
    1. Make right child field of parent point to NULL.
    2. Go to step 5.
5. Release the memory for currentNode.

Case II: Algorithm to delete a node with one child:

1. Locate the node to be deleted. Mark it as currentNode and its parent as parent.
2. If currentNode has a left child:
    a. Mark the left child of currentNode as child.
    b. Go to step 4.
3. If currentNode has a right child:
    a. Mark the right child of currentNode as child.
    b. Go to step 4.
4. If currentNode is the root node:
    a. Mark child as root.
    b. Go to step 7.
5. If currentNode is the left child of parent:
    a. Make left child field of parent point to child.
    b. Go to step 7.
6. If currentNode is the right child of parent:
    a. Make right child field of parent point to child.
    b. Go to step 7.
7. Release the memory of currentNode.

Case III: Algorithm to delete a node with both child:

1. Locate the node to be deleted. Mark it as currentNode and its parent as parent.
2. Locate the inorder successor of currentNode. Mark it as Inorder_suc. Execute the following steps to locate Inorder_suc:
    a. Mark the right child of currentNode as Inorder_suc.
    b. Repeat until the left child of Inorder_suc becomes NULL:
        i. Make Inorder_suc point to its left child.
3. Replace the information held by currentNode with that of Inorder_suc.
4. If the node marked Inorder_suc is a leaf node:
    a. Delete the node marked Inorder_suc by using the algorithm for Case I.
5. If the node marked Inorder_suc has one child:
    a. Delete the node marked Inorder_suc by using the algorithm for Case II.

Pseudo code:

```
Algorithm deletion (int item)
{
   Node* parent = NULL;
   Node* cur = root;
```

```
   search(cur, item, parent);

   if (cur == NULL)
      return;

   if (cur->left == NULL && cur->right == NULL)
   {
      if (cur != root)
      {
         if (parent->left == cur)
            parent->left = NULL;
         else
            parent->right = NULL;
      }
      else
         root = NULL;

      free(cur);
   }
   else if (cur->left && cur->right)
   {
      Node* succ  = findMinimum(cur- >right);
       int val = succ->data;
       deletion(succ->data);
        cur->data = val;
   }
  else
   {
      Node* child = (cur->left)? Cur- >left: cur->right;

      if (cur != root)
      {
         if (cur == parent->left)
            parent->left = child;
         else
            parent->right = child;
      }

      else
         root = child;
      free(cur);
   }
}

Node* findMinimum(Node* cur)
{
```

```
   while(cur->left != NULL) {
      cur = cur->left;
   }
   return cur;
}
```

Conclusion:

- Students have understood and implemented the BST along with all its primitive operations.
- Understood and written the ADT for BST
- Able to use the BST for efficient solution in searching problems

Sample Questions:

1. What is a binary search tree?
2. How do you delete a node from a binary search tree?
3. How to handle duplicate nodes in a binary search tree?
4. Explain the difference between Binary Tree and Binary Search Tree with an example?
5. What are advantages and disadvantages of BST?
6. What is the difference between binary search and linear search?
7. What are time and space complexities of operations on BST?
8. What are different cases in delete operation of BST?