Experiment No. : 5

Title: Implementation of Binary Search Tree using Doubly Linked List (DLL)

Batch: B1 **Roll No.: 16010420133** **Experiment No.: 5**

Aim: Write a program for following operations on Binary Search Tree using Doubly Linked List (DLL).

1) Create empty BST,
2) Insert a new element on the BST
3) Search for an element on the BST
4) Delete an element from the BST
5) Display all elements of the BST

Resources needed: C/C++/JAVA editor and compiler

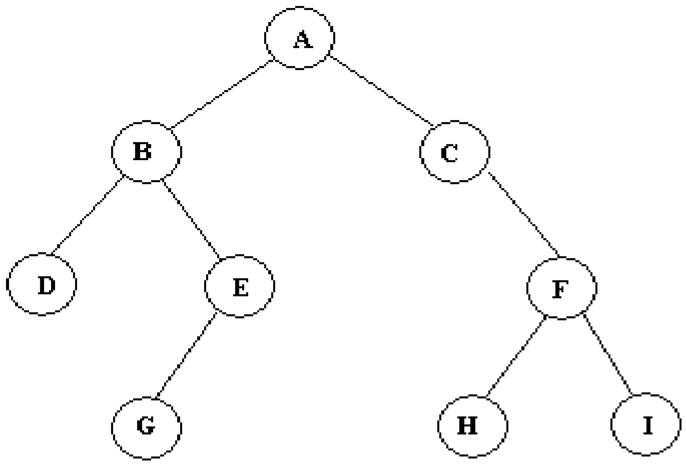
Theory

Binary Tree :-

A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

Example



Figure(a): Binary Tree Example

Traversals :

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds.

1. depth-first traversal
2. breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent.

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

Consider an example the following tree and its four traversals:

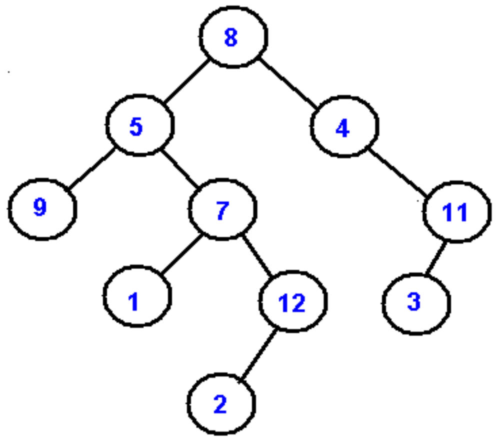
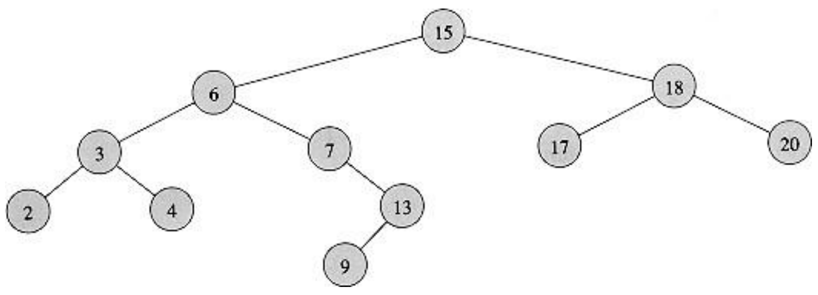


Figure 5.3: Example of Tree Traversals

PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Binary Search Tree(BST): It is a binary tree where elements are stored in a particular order of left child is less than the parent and parent is less than the right child. This small modification makes BST efficient for searching. It is also called as BST property.
Example of BST following fig is the example BST whose root is 8 and follows the BST property.



Methods for storing binary trees

Binary trees can be constructed from programming language primitives in several ways.

Nodes and references

In a language with records and references, binary trees are typically constructed by having a tree node structure which contains some data and references to its left child and its right child. Sometimes it also contains a reference to its unique parent. If a node has fewer than two children, some of the child pointers may be set to a special null value, or to a special sentinel node.

Binary Search Tree can be implemented as a linked data structure in which each node is an object with two pointer fields. The two pointer fields *left and right* points to the nodes corresponding to the left child and the right child NULL in any pointer field signifies that there exists no corresponding child.

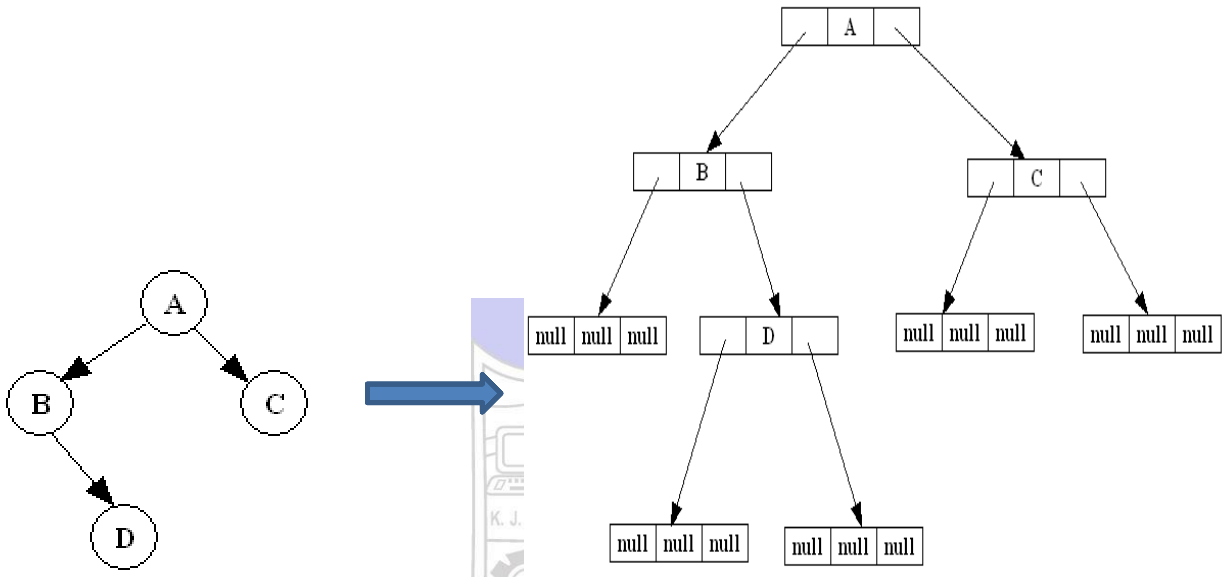


Figure : Binary Search Trees example

Algorithm :

1. createBST( ):

```
Create a root node.
Set root = NULL.
```

2. insert( ):

```
Create newnode;
Set newnode -> data = input from user(d).
Set newnode -> left and right = NULL.
If root is equal to NULL then Set root = newnode.
Else if newnode -> data > temp -> data and temp ->right != NULL then use recursion and
call insert function by passing temp -> right (first time temp = root).
Here if temp -> right == NULL so we can directly insert node to the right.
Else repeat above 2 steps for  temp -> left.
```

3. search( ):

If root == NULL, then nothing to search as BST is empty.
If temp -> data == data entered by user(d), then data is found.
Else if d > temp -> data then use recursion and call search by passing temp -> right.
Else if d < temp -> data then use recursion and call search by passing temp -> left.
Else if after passing all nodes temp == NULL is satisfies it means no data found.

4. delete( ):

If root == NULL, then nothing to delete as BST is empty.
Else if root -> left and right == NULL, it means there is only 1 element(root) in BST.
If data entered by user(d) == root -> data then we delete it by free(root) and set root = NULL.
If data not matched it means invalid input by user.
Else Set t1 = t2 = root and call search function by passing in data entered by user.
The search function searches, sets t1 = parent of node to be deleted and calls delete1 function by passing in temp (node to be deleted).
Delete1 checks if the node to be deleted has 0, 1, 2 child nodes.
For 0 child (leaf node) check if temp -> left and right == NULL and then compare with Left and right of t1 and sets it to NULL and then free(temp).
For 1 child, similar process but only at the node which is not NULL.
Finally for 2 child nodes, we can find smallest from right sub tree or largest from left subtree by calling smallest or largest function and set that value to k.
now we repeat entire process by recursion for k delete k and so on.
Finally the node entered by user to be deleted is replaces by k and so on.

5. getParent( ):

If root == NULL, then nothing to search and get parent node as BST is empty.
Else if root -> data == input from user (d), then it is root node which has no parent.
Else calling search function by passing in d.
If match found then search sets t1 as parent node so parent is t1 -> data
If no match then data does not exist in the BST.

6. displayInorder( ):

If root == NULL, then nothing to display as BST is empty.
Else if first check temp -> left != NULL is true then use recursion and pass temp -> left.
If condition false then print temp -> data;
Then check if temp -> right != NULL is true then use recursion and pass temp -> right.
We follow this order as in inorder first left is printed, then root, then right and by Following this order the BST will be displayed in inorder.

---

1. createBST() – This function should create a ROOT pointer with NULL value as empty BST.
2. insert(typedef newelement) – This void function should take a newelement as an argument to be inserted on an existing BST following the BST property.
3. typedef search(typedef element) – This function should search for specified element on the non-empty BST and return it.
4. typedef delete(typedef element) – This function searches for an element and if found deletes it from the BST and returns the same.
5. typedef getParent(typedef element) - This function searches for an element and if found, returns its parent element.

6. DisplayInorder( ) – This is a void function which should go through non- empty BST, traverse it in inorder fashion and print the elements on the way.

NOTE : All functions should be able to handle boundary(exceptional) conditions.

Activity: Write pseudocode for each method and implement the same.

Results: A program depicting the correct behaviour of BST and capable of handling all possible exceptional conditions and the same is reflecting clearly in the output.

Program :

Code:-

```
#include<stdio.h>

struct node
{
    int data;
    struct node *left;
    struct node *right;
}*root, *newnode, *t1, *t2;

int k, a, flag;

void menu()
{
    int choice, d, s, x;
    printf("\n1. Insert");
    printf("\n2. Search");
    printf("\n3. Delete");
    printf("\n4. GetParent");
    printf("\n5. DisplayInorder");
    printf("\n6. Exit");
    printf("\n\nEnter choice (1, 2, 3, 4, 5, 6): ");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            printf("\nEnter data to be inserted in Binary Search Tree: ");
            scanf("%d",&d);
            insert(d, root);
            menu();
            break;
        case 2:
            printf("\nEnter data to be searched in Binary Search Tree: ");
            scanf("%d",&d);
            s = search(d, root);
            if (s == -2)
            {
                printf("\nThe Binary Search Tree is empty!\n");
            }
            else if (s == -1)
            {
                printf("\nData not found!\n");
            }
            else
            {
                printf("\nData searched and found is: %d\n", s);
            }
            menu();
            break;
        case 3:
            printf("\nEnter data to be deleted from the Binary Search Tree: ");
            scanf("%d",&d);
            x = delete(d);
            if (x == -3)
            {
                printf("\nData not found\n");
            }
            else
                if (x == -2)
            {
                //Already printed
            }
            else if (x == -1)
            {
                printf("\nDeletion not possible as the Binary Search Tree is empty!\n");
            }
            else
            {
                printf("\n%d has been deleted from the Binary Search Tree\n", x);
            }
            menu();
            break;
        case 4:
            printf("\nEnter data to find its Parent data: ");
            scanf("%d",&d);
            x = getParent(d);
            if (x == -1)
            {
                printf("\n%d is the root and hence has no Parent\n", d);
            }
            else if (x == -2)
            {
                printf("\n%d does not exist in the Binary Search Tree\n", d);
            }
```

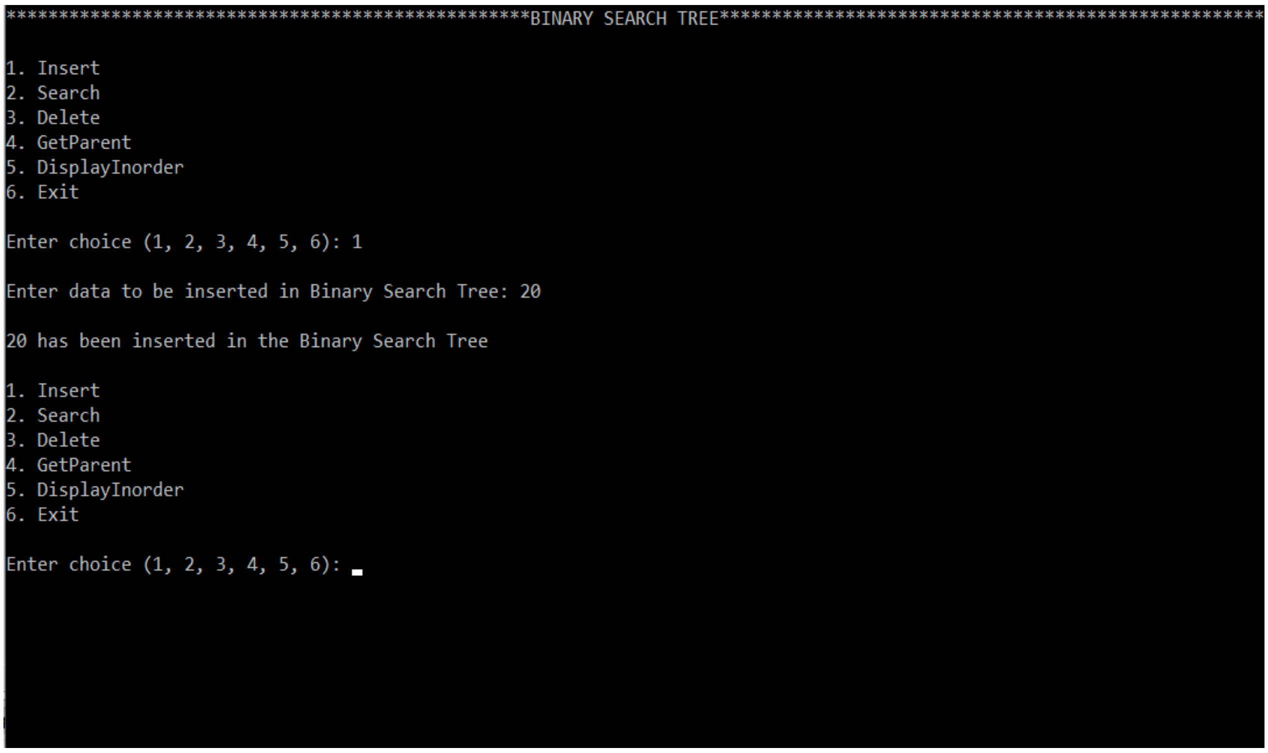
```
            else if (x == -3)
            {
                printf("\nCannot find parent as the Binary Search Tree is empty!\n");
            }
            else
            {
                printf("\nThe parent of %d is: %d\n", d, x);
            }
            menu();
        case 5:
            printf("\n");
            displayInorder(root);
            printf("\n");
            menu();
        case 6:
            exit(0);
        default:
            printf("\nInvalid Input!\n");
            menu();
    }
}

void createBST()
{
    root = (struct node*)malloc(sizeof(struct node));
    root = NULL;
}

void insert(int d, struct node *temp)
{
    newnode = (struct node*)malloc(sizeof(struct node));
    newnode -> data = d;
    newnode -> left = NULL;
    newnode -> right = NULL;

    if (root == NULL)
    {
        root = newnode;
        printf("\n%d has been inserted in the Binary Search Tree\n", d);
    }
    else
    {
        if ((newnode -> data > temp -> data) && (temp ->right != NULL))
        {
            insert(d, temp -> right);
        }
        else if ((newnode -> data > temp -> data) && (temp -> right == NULL))
        {
            temp -> right = newnode;
            printf("\n%d has been inserted in the Binary Search Tree\n", d);
        }
        else if ((newnode -> data < temp -> data) && (temp -> left != NULL))
        {
           insert(d, temp -> left);
        }
        else if ((newnode -> data < temp -> data) && (temp -> left == NULL))
        {
           temp -> left = newnode;
           printf("\n%d has been inserted in the Binary Search Tree\n", d);
        }
        else
        {
           printf("\n%d already exists in the Binary Search Tree and cannot be inserted\n", d);
        }
    }
}

int search(int d, struct node *temp)
{
    if (root == NULL)
    {
        return (-2);
    }
    else if (temp == NULL)
    {
        return (-1);
    }
    else if (temp -> data == d)
    {
        if (flag == 1)
        {
           delete1(temp);
           flag = 0;
        }
        return (temp -> data);
    }
    else if (d > temp -> data)
    {
        t1 = temp;
        search(d, temp -> right);
    }
    else if (d < temp -> data)
    {
        t1 = temp;
        search(d, temp -> left);
    }
}

int delete(int d)
{
    int t;
    if (root == NULL)
    {
        return(-1);
    }
    else if (root -> left == NULL && root -> right == NULL)
    {
        if (root ->  data == d)
        {
            printf("\n%d deleted from the Binary Search Tree, Tree is now empty!\n", root -> data);
            t = root -> data;
            free(root);
            root = NULL;
            return(-2);
        }
        else
        {
            return (-3);
        }
    }
    else
    {
        t1 = root;
        t2 = root;
        flag = 1;
        t = search(d, root);
        if (t == -1)
        {
            return(-3);
        }
        else
        {
            return(d);
        }
    }
}

int getParent(int d)
{
    if (root == NULL)
    {
        return (-3);
    }
    else if (root -> data == d)
    {
        return(-1);
    }
    else
    {
        a = search(d, root);
```

```
      if (a == -1)
      {
         return (-2);
      }
      else
      {
         return(t1 -> data);
      }
   }
}

void displayInorder(struct node *temp)
{
   if (root == NULL)
   {
      printf("There is no data to display as the Binary Search Tree is empty!");
      return;
   }
   if (temp -> left != NULL)
   {
      displayInorder(temp -> left);
   }
   printf("%d ", temp -> data);
   if (temp -> right != NULL)
   {
      displayInorder(temp -> right);
   }
}

/* To delete searched node */
void delete1(struct node *temp)
{
   int k;

   /* To delete leaf node */
   if ((temp -> left == NULL) && (temp -> right == NULL))
   {
      if (t1 -> left == temp)
      {
         t1 -> left = NULL;
      }
      else
      {
         t1 -> right = NULL;
      }
      free(temp);
      temp = NULL;
      return;
   }
```

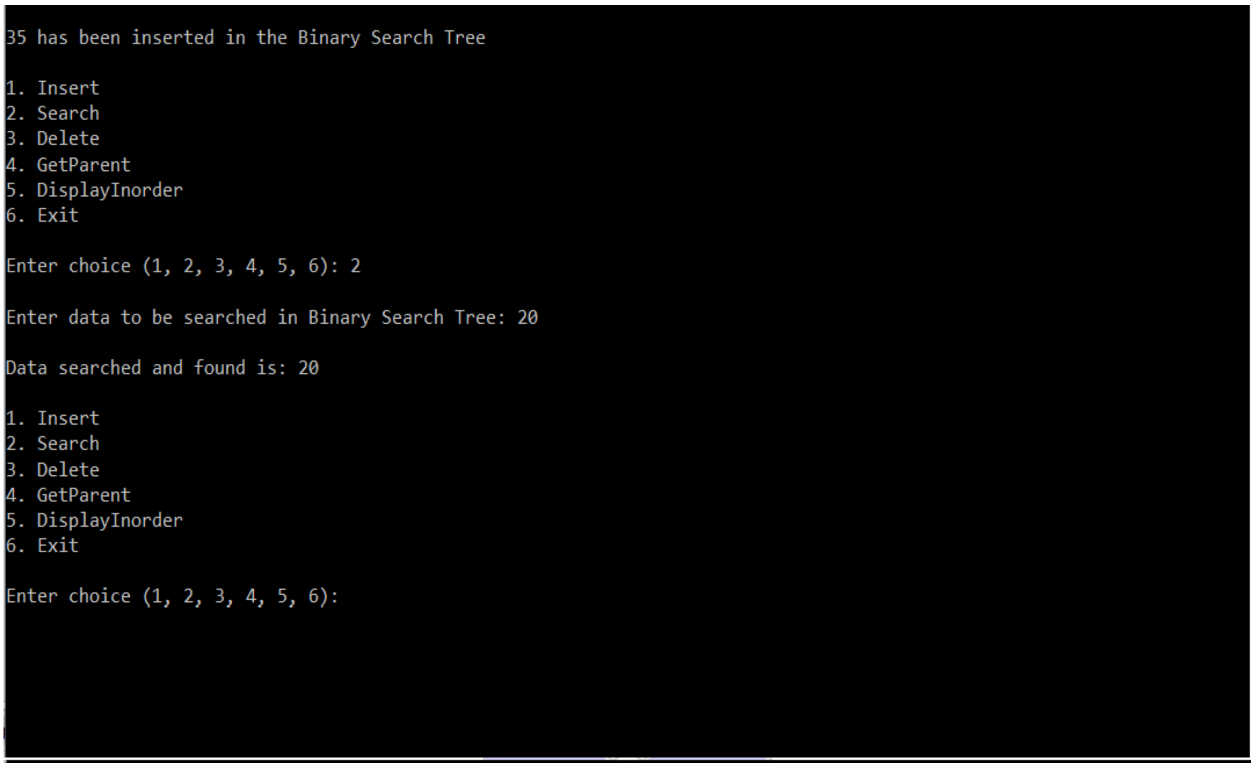
```
/* To delete node having one left hand child */
else if (temp -> right == NULL)
{
   if (t1 == temp)
   {
      root = temp -> left;
      t1 = root;
   }
   else if (t1 -> left == temp)
   {
      t1 -> left = temp -> left;

   }
   else
   {
      t1 -> right = temp -> left;
   }
   free(temp);
   temp = NULL;
   return;
}

/* To delete node having right hand child */
else if (temp -> left == NULL)
{
   if (t1 == temp)
   {
      root = temp -> right;
      t1 = root;
   }
   else if (t1 -> right == temp)
   {
      t1 -> right = temp -> right;
   }
   else
   {
      t1 -> left = temp -> right;
   }
   free(temp);
   temp == NULL;
   return;
}

/* To delete node having two child */
else if (temp -> left != NULL && temp -> right != NULL)
{
   t2 = root;
   if (temp -> right != NULL)
   {
      k = smallest(temp -> right);
```

```
        }
        else
        {
           k = largest(temp -> left);
        }
        flag = 1;
        search(k, root);
        temp -> data = k;
    }

}

/* To find the largest element in the left sub tree */
int largest(struct node *temp)
{
    if (temp -> right != NULL)
    {
        t2 = temp;
        return(largest(temp -> right));
    }
    else
    {
        return(temp -> data);
    }
}

/* To find the smallest element in the right sub tree */
int smallest(struct node *temp)
{
    t2 = temp;
    if (temp -> left != NULL)
    {
        t2 = temp;
        return(smallest(temp -> left));
    }
    else
    {
        return(temp -> data);
    }
}

void main()
{
    printf("**************************************************BINARY        SEARCH
TREE**************************************************");
    createBST();
    menu();
}
```

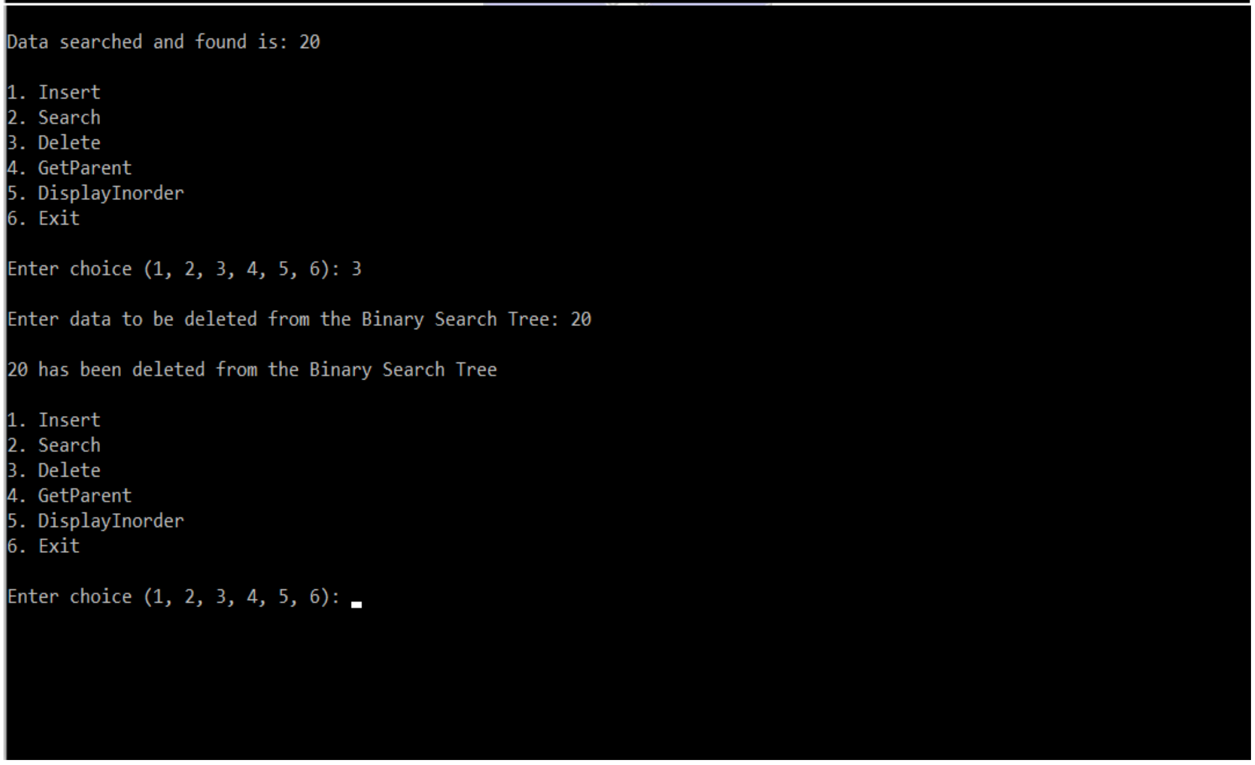
Output:-



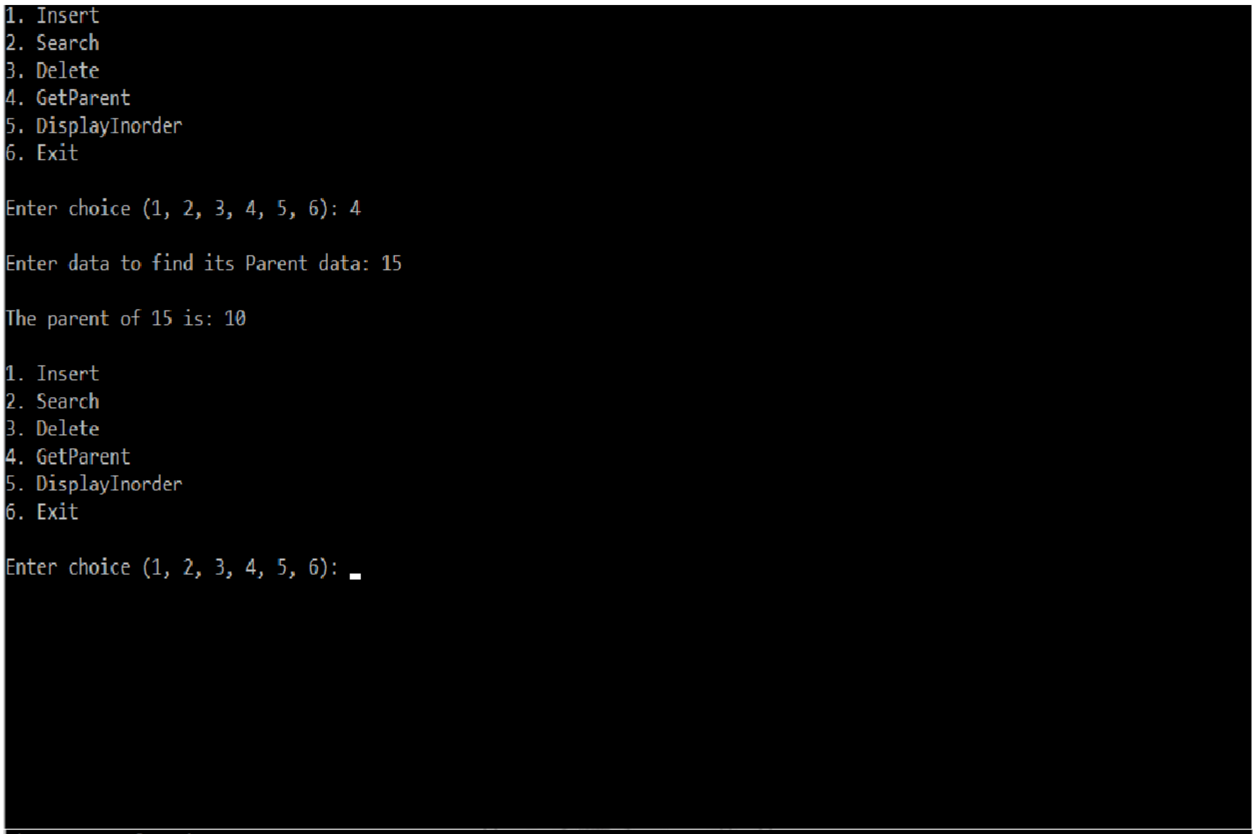
```
************************************************BINARY SEARCH TREE**************************************************

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): 1

Enter data to be inserted in Binary Search Tree: 20

20 has been inserted in the Binary Search Tree

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6):
```



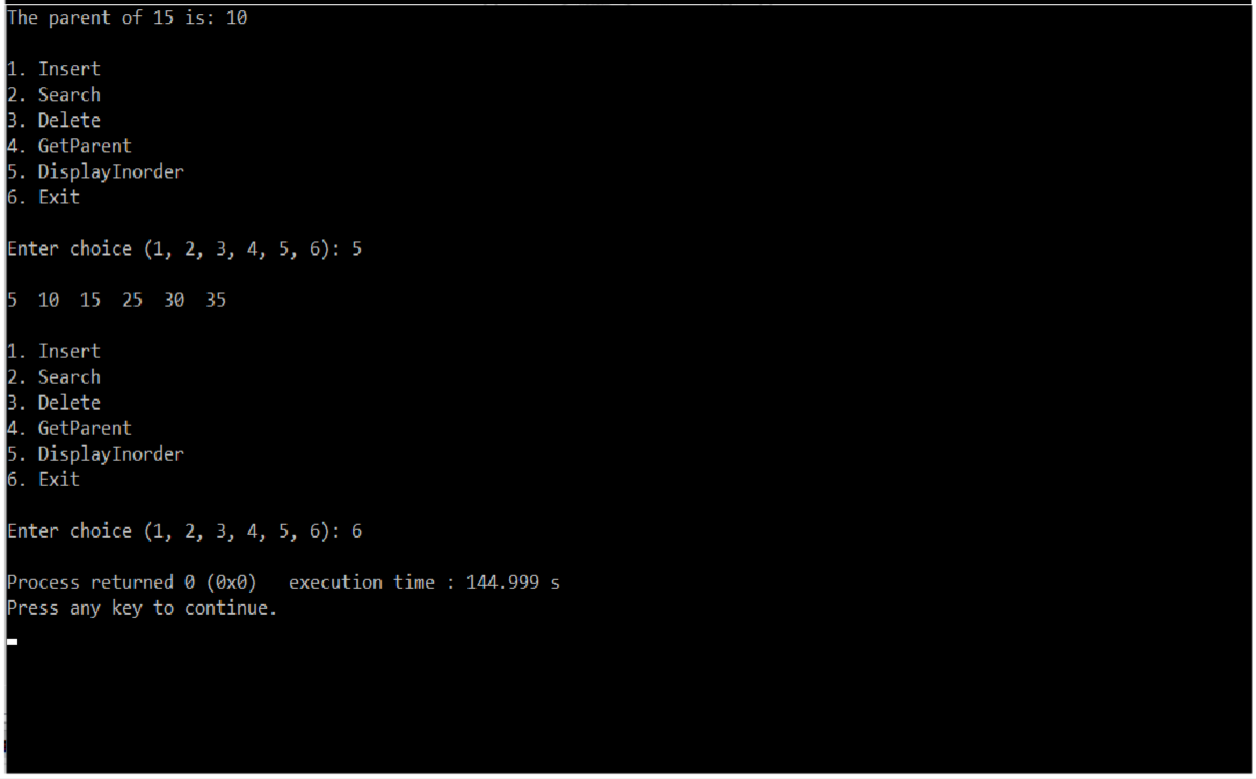
```
35 has been inserted in the Binary Search Tree

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): 2

Enter data to be searched in Binary Search Tree: 20

Data searched and found is: 20

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6):
```



```
Data searched and found is: 20

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): 3

Enter data to be deleted from the Binary Search Tree: 20

20 has been deleted from the Binary Search Tree

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6):
```



```
1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): 4

Enter data to find its Parent data: 15

The parent of 15 is: 10

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): _
```



```
The parent of 15 is: 10

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): 5

5  10  15  25  30  35

1. Insert
2. Search
3. Delete
4. GetParent
5. DisplayInorder
6. Exit

Enter choice (1, 2, 3, 4, 5, 6): 6

Process returned 0 (0x0)   execution time : 144.999 s
Press any key to continue.
```

Outcomes:

CO2: Apply linear and non-linear data structure in application development.

Conclusion:

Binary Search Tree has been successfully implemented using a C program. The program follows the problem statement and can insert, search, delete, get parent and display in inorder form in the binary search tree. This experiment makes concepts clearer and helps in a better understanding of the Binay search Tree.

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

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites:

- Y. Langsam, M. Augenstin and A. Tannenbaum, “Data Structures using C”, Pearson Education Asia, 1st Edition, 2002.