**EXPERIMENT NO 8**

**~ TRUSHANT RATHOD**

**COMPS 18**

**Exp No. 8: Binary Search Tree Operations**

**Aim :** **Implementation of Binary Search Tree ADT using Linked List.**

**Objective:**

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

**Theory:**

A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words node in a binary tree has at most two children and each child node is referred as left or right child.

Traversals in tree can be in one of the three ways : preorder, postorder, inorder.

Preorder Traversal

Here the following strategy is followed in sequence

1. Visit the root node R
2. Traverse the left subtree of R
3. Traverse the right sub tree of R



| **Description** | **Output** |
| --- | --- |
| Visit Root | A |
| Traverse left sub tree – step to B then D | ABD |
| Traverse right sub tree – step to G | ABDG |
| As left subtree is over. Visit root , which is already visited so go for right subtree | ABDGC |
| Traverse the left subtree | ABDGCEH |
| Traverse the right sub tree | ABDGCEHIF |

Inorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Visit the root node R
3. Traverse the right sub tree of R

| **Description** | **Output** |
| --- | --- |
| Start with root and  traverse left sub tree from A-B-D | D |
| As D doesn’t have left child visit D and go for right subtree of D which is G so visit this. | DG |
| Backtrack to D and then to B and visit it. | DGB |
| Backtract  to A and visit it | DGBA |
| Start with right sub tree from C-E-H and visit H | DGBAH |
| Now traverse through parent of H which is E and then I | DGBAHEI |
| Backtrack to C and visit it and then right subtree of E which is F | DGBAHEICF |

Postorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Traverse the right sub tree of R
3. Visit the root node R

| **Description** | **Output** |
| --- | --- |
| Start with left sub tree from A-B-D and then traverse right sub tree to get G | G |
| Now Backtrack to D and visit it  then to B and visit it. | GD |
| Now as the left sub tree is over go for right sub tree | GDB |
| In right sub tree start with leftmost child to visit H followed by I | GDBHI |
| Visit its root as E and then go for right sibling of C as F | GDBHIEF |
| Traverse its root as C | GDBHIEFC |
| Finally a root of tree as A | GDBHIEFCA |

**Algorithm**

Algorithm: PREORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in preorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then

visit(ptr)

PREORDER(LSON(ptr))\

PREORDER(RSON(ptr))

End if

1. Stop

Algorithm: INORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in inorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then

INORDER (LSON(ptr))

visit(ptr)

INORDER (RSON(ptr))

End if

1. Stop

Algorithm: POSTORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in postorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then

PREORDER(LSON(ptr))

PREORDER(RSON(ptr))

visit(ptr)

End if

1. Stop

**Code:**

#include<stdio.h>

#include<stdlib.h>

#define MAX\_SIZE 25

#define MAX(a, b) a>b ? a : b

struct TreeNode

{

int val;

struct TreeNode\* left;

struct TreeNode\* right;

};

struct TreeNode\* root = NULL;

void insert(int);

void delete1(int);

void exit(int);

int search(int);

struct TreeNode\* queue[MAX\_SIZE];

struct TreeNode\* new\_node = NULL;

struct TreeNode\* temp = NULL, \*last\_node = NULL, \*key\_node = NULL;

int main()

{

int n, node\_data;

char user\_active = 'Y';

while(user\_active == 'Y' || user\_active == 'y')

{

printf("\n1. Insert\n2. Delete\n3. Search\n4. Exit ");

printf("\n\nEnter Your Choice: ");

scanf("%d", &n);

printf("\n");

switch(n)

{

case 1:

printf("Enter data for new node: ");

scanf("%d", &node\_data);

insert(node\_data);

break;

case 2:

printf("Enter node data: ");

scanf("%d", &node\_data);

delete1(node\_data);

break;

case 3:

printf("Enter node data: ");

scanf("%d", &node\_data);

search(node\_data);

break;

case 4:

exit(n);

break;

default:

printf("Invalid choice");

}

}

return 0;

}

struct TreeNode\* create(int data)

{

struct TreeNode\* new\_node = (struct TreeNode\*) malloc (sizeof(struct TreeNode));

if(new\_node == NULL)

{

printf("\nMemory can't be allocated for new node\n");

return NULL;

}

new\_node->left = NULL;

new\_node->right = NULL;

new\_node->val = data;

return new\_node;

}

void insert(int data)

{

int front = -1;

int rear = -1;

if(root == NULL)

{

struct TreeNode\* new\_node = create(data);

if(new\_node)

{

root = new\_node;

printf("\n \* Node with data %d was inserted", data);

}

return;

}

queue[front+1] = root;

front = rear = 0;

while(front <= rear)

{

struct TreeNode\* temp = queue[front];

front++;

if(temp->left != NULL)

{

queue[++rear] = temp->left;

}

else

{

new\_node = create(data);

if(new\_node)

{

temp->left = new\_node;

printf("\n\* Node with data %d was inserted", data);

}

return;

}

if(temp->right != NULL)

{

queue[++rear] = temp->right;

}

else

{

new\_node = create(data);

if(new\_node)

{

temp->right = new\_node;

printf("\n\* Node with data %d was inserted", data);

}

return;

}

}

}

void delete1(int key)

{

int front = -1;

int rear = -1;

if(root == NULL)

{

return;

}

if(root->left == NULL && root->right == NULL)

{

if(root->val == key)

{

root = NULL;

printf("\n\* Node with data %d was deleted", key);

return;

}

else

{

return;

}

}

queue[front + 1] = root;

front = rear = 0;

while (front <= rear)

{

temp = queue[front];

front++;

if (temp->val == key)

{

key\_node = temp;

}

if (temp->left != NULL)

{

last\_node = temp;

queue[++rear] = temp->left;

}

if (temp->right != NULL)

{

last\_node = temp;

queue[++rear] = temp->right;

}

}

if (key\_node != NULL)

{

key\_node->val = temp->val;

if (last\_node->right == temp)

{

last\_node->right = NULL;

}

else

{

last\_node->left = NULL;

}

printf("\n\* Node with data %d was deleted", key);

free(temp);

return;

}

printf("\n\* Node with data %d was not found", key);

}

int search(int key)

{

int front = -1;

int rear = -1;

int has\_found;

if(has\_found == -1) {

printf("\nNode was not found!");

} else {

printf("\nNode was found");

}

if (root == NULL)

{

return -1;

}

queue[front + 1] = root;

front = rear = 0;

while (front <= rear)

{

struct TreeNode\* temp = queue[front];

front++;

if (temp->val == key)

{

return 1;

}

if (temp->left != NULL)

{

queue[++rear] = temp->left;

}

if (temp->right != NULL)

{

queue[++rear] = temp->right;

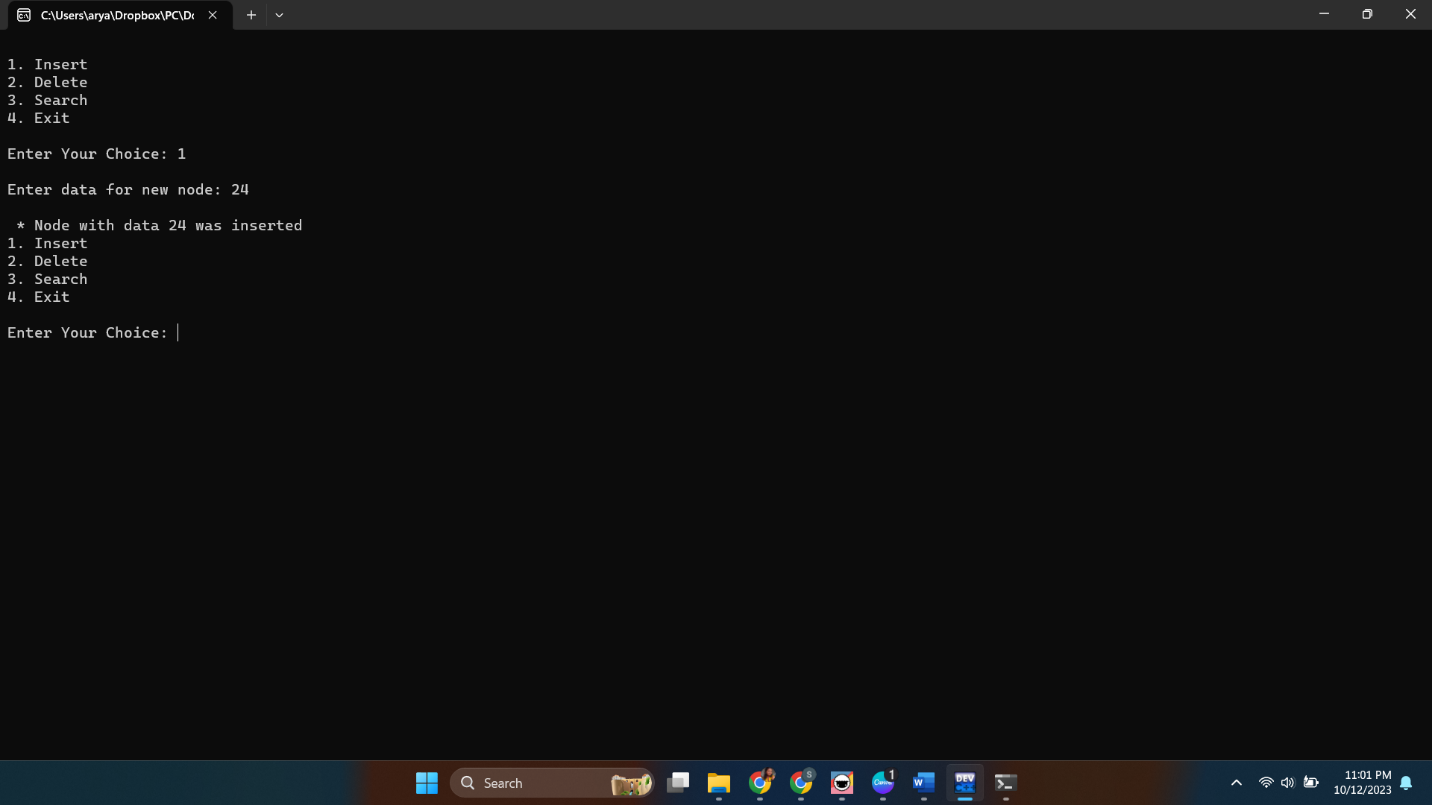
}

}

return -1;

}

Output:



**Conclusion:**

a Binary Search Tree (BST) ADT using a linked list provides efficient data sorting, dynamic resizing, and ordered data retrieval. However, it may be less memory-efficient and can become unbalanced, necessitating self-balancing techniques like AVL or Red-Black Trees. The recursive nature of BST operations should also be considered.