EXPERIMENT NO. 07

Author: Harsheet Chordiya

Roll No.: 37

Sem and Section: 3 CSEB

Source file: expt07.c

Date: 25/01/2021

==

Aim: To study binary search tree (BST) and implement various operations (including traversals) on a BST.

Problem Statement: Consider a binary search tree node – treeNode, representing a self-referential structure.

Write a menu driven program to — (1) insert a node into BST, (2) delete a node from BST, (3) print traversals (inorder, preorder, postorder) for a BST, (4) find height of BST, (5) count and print all leaf nodes, parent nodes.

Your program shall also allow for creating a pre-allocated list before the start of BST ADT.You must not use global variables.

==

THEORY

==

A Binary Search Tree is a rooted ordered binary tree with following Properties:

At each node, its left subtree contains nodes having keys lesser than node's key
At each node, its right subtree contains nodes having keys greater than node's key

Both the left and right- subtree is a BST

Duplicate keys are not allowed.

BST has been amongst the first indexing structures used for external memory.

In a BST implementation the inorder traversal always results in keys sorted in ascending order. BST supports an efficient mechanism for searching a key, taking a maximum of log2(N) probes.

BST is a well-known and frequently implemented library routine.

==

ALGORITHMS

==

<u>Procedure INORDER (ROOT)</u>

Given a rooted binary tree denoted by ROOT, this procedure prints the inorder traversal of the tree.

```
1. Recurse, on an existing node

If ROOT = NULL

       Call INORDER (LCHILD(ROOT)) /* Traverse left subtree */

       Write(DATA(ROOT)) /* Print Key at Node */

       Call INORDER (RCHILD (ROOT)) /* Traverse right subtree */

2. Otherwise, return routine call

      Return
```

<u>Procedure PREORDER (ROOT)</u>

Given a rooted binary tree denoted by ROOT, this procedure prints the preorder traversal of the tree.

```
1. Recurse, on an existing node

      If ROOT NULL

             Write(DATA(ROOT)) /* Print Key at Node */

             Call PREORDER (LCHILD (ROOT)) * Traverse left subtree */

             Call PREORDER (RCHILD(ROOT)) /* Traverse right subtree */

 2. Otherwise, return routine call

      Return
```

<u>Procedure POSTORDER (ROOT)</u>

Given a rooted binary tree denoted by ROOT, this procedure prints the postorder traversal of the tree.

```
1. Recurse, on an existing node

      If ROOT NULL

            Call POSTORDER (LCHILD (ROOT)) /* Traverse left subtree */

          Call POSTORDER (RCHILD (ROOT)) /* Traverse right subtree */
          Write(DATA(ROOT)) /* Print Key at Node
```

```
2. Otherwise, return routine call
       Return
```

Function HEIGHT BIN TREE (ROOT)

Given a rooted binary tree pointed by ROOT, this function recursively computes the height of the tree. A leaf node is assumed at height = 0.

```
1. Empty Tree??Return -1 [Terminating Case-1]
       If ROOT = NULL
       Return -1
2. Leaf Node? Return 0 [Terminating Case-2]
       If LCHILD(ROOT)= NULL AND RCHILD(ROOT)= NULL
              Return 0
3. Recursion?? Compute height
       Return MAX (Call HEIGHT_BIN_TREE (LCHILD (ROOT), Call HEIGHT_BIN_TREE
       (RCHILD(ROOT)) + 1
```

Function CREATE_BST()

```
1. Initialize ROOT
       ROOT = NULL
2. Set up the iteration for insertion of a node
       Repeat Step 3 thru 4 until KEY = STOP
3. Acquire the node value read(KEY)
4. Insert the Node
       If KEY • STOP
              Call INSERT_BST (ROOT, KEY)
5. Return the tree
       Return ROOT
```

Function DELETE NODE BST(ROOT, KEY)

Given a rooted Binary Search Tree pointed by ROOT, this function deletes a node with data value equal to KEY from the tree. TEMP is the local tree pointer.

```
1. Is empty tree??
       If ROOT = NULL
```

```
            Write('Delete Failed, Empty Tree')

            Return NULL

2. Traverse the subtrees to locate the node to be deleted

      If KEY DATA(ROOT)

            LCHILD(ROOT) = Call DELETE_NODE_BST (LCHILD(ROOT), KEY) / Traverse
            Left Subtree */

      Else If KEY > DATA(ROOT)

            RCHILD(ROOT) = Call DELETE_NODE_BST (RCHILD(ROOT), KEY) /* Traverse
            Right Subtree */

      Else /* The intended node */

            If LCHILD (ROOT) NULL AND RCHILD(ROOT) NULL /* Case-1: Node has
      both   children*/

                  TEMP = Call FIND_MIN_NODE (RCHILD(ROOT))

                  DATA(ROOT) = DATA(TEMP)

                  RCHILD(ROOT) = Call DELETE_NODE_BST (RCHILD(ROOT),
                  DATA(ROOT))

            Else

            TEMP = ROOT

            If LCHILD(ROOT) = NULL /* Case-2: Node has only right child*/

                  ROOT = RCHILD(ROOT) /* Case-3: Node has only left child */

            Else If RCHILD (ROOT) = NULL

                  ROOT = LCHILD(ROOT) * Case-4: Its a Leaf node, Restore */

            Restore TEMP

3.Return the tree

      Return (ROOT)
```

==

PROGRAM

==

```
//Header File Inclusion
#include <stdio.h>
#include <stdlib.h>
// Structure Declaration
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
typedef struct node tree;
// User-defined Functions
     //Function to create tree
tree* create(tree *root) {
    int n, i;
    int a[20];
    printf("Enter no of elements:");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter element %d:", i + 1);
        scanf("%d", &a[i]);
    }
    tree *ptr, *temp, *prev;
    root = NULL;
    for (i = 0; i < n; i++) {
        temp = (tree *) malloc(sizeof(tree));
        temp->data = a[i];
        temp->left = NULL;
        temp->right = NULL;
        if (root == NULL)
            root = temp;
        else {
            ptr = root;
            while (ptr != NULL) {
                prev = ptr;
                if (ptr->data < temp->data)
                    ptr = ptr->right;
                else
                    ptr = ptr->left;
            }
            if (prev->data < temp->data)
                prev->right = temp;
            else
                prev->left = temp;
        }
    }
    return root;
}
//Function to calculate Min
tree*findMin(tree* node){
    if(node==NULL)
```

```
        return NULL;
    else
        if(node->left==NULL)
            return node;
        else
            return (findMin(node->left));
}
//Function to delete node
tree* delete(tree* node,int x) {
    if(node==NULL)
    {
        return 0;
    }
    tree *temp;
    if (node == NULL)
        printf("Element %d is not found in the tree...\n", x);
    else
        if (x < node->data)
            node->left = delete(node->left, x);
        else
             if (x > node->data)
                node->right = delete(node->right, x);
             else
                 if (node->left && node->right)
                 {
                    temp = findMin(node->right);
                    node->data = temp->data;
                    node->right = delete(node->right, node->data);
                 }
                 else
                     {
                        temp = node;
                        if (node->left == NULL)
                            node=node->right;
                        else if (node->right == NULL)
                            node=node->left;
                        free(temp);
                     }

    return node;
}
//Function to display preorder
void preorder(tree *node)
{
    if(node==NULL)
    {
        return;
    }
    else
    {
        printf("%d",node->data);
        preorder(node->left);
        preorder(node->right);
    }
```

```
}
//Function to display inorder
void inorder(tree *node)
{
   if(node==NULL)
    {
        return;
    }
    else
    {
        inorder(node->left);
        printf("%d",node->data);
        inorder(node->right);
    }
}
//Function to display postorder
void postorder(tree *node)
{
    if(node==NULL)
    {
        return;
    }
    else
    {
        postorder(node->left);
        postorder(node->right);
        printf("%d",node->data);
    }
}
//Function to calculate nodes
int nodes(tree *node)
{
    int count = 0;
    if(node != NULL)
    {
        nodes(node->left);
        count++;
        nodes(node->right);
    }
    return count;
}
//Function to calculate leaf nodes
int leaf(tree *node)
{
    int count = 0;
    if(node != NULL)
    {
        leaf(node->left);
        if((node->left == NULL) && (node->right == NULL))
        {
            count++;
        }
        leaf(node->right);
    }
    return count;
}
//Function to calculate non-leaf nodes
int nonleaf(tree *node)
{
    int count = 0;
    if(node != NULL)
    {
        nonleaf(node->left);
        if((node->left != NULL) || (node->right != NULL))
        {
            count++;
        }
        nonleaf(node->right);
    }
    return count;
}
//Function to calculate half tree
int half(tree *node)
{
   if (node == NULL)
   {
        return 0;
   }
   int result = 0;
   if ((node->left == NULL && node->right != NULL) || (node->left
!= NULL && node->right ==NULL))
```

```
    {
       result++;
    }
    result += (half(node->left) + half(node->right));
    return result;
}
//Function For Height of tree
int heightBST(tree* node){
     int leftH,rightH;
     if(node==NULL || (node->left==NULL && node->right==NULL))
         return 0;
     leftH=heightBST(node->left)+1;
     rightH=heightBST(node->right)+1;
     return(leftH>=rightH ? leftH:rightH);
}
// Main Function
int main()
{
     int choice,x;
     tree *root1;
     struct node *root, *newnode;
     do
     {
         printf("\nEnter choice:\n1.Create\n2.Preorder
traversal\n3.Inorder traversal\n4.Postorder traversal\n5.Count
nodes\n6.Count leaf nodes\n7.Count non-leaf nodes\n8.Count half
nodes\n9.Delete\n10.Height of BST\n");
         scanf("%d",&choice);
```

```
        switch(choice)
        {

            case 1:
                    root1 = create(root1);
                    break;
            case 2:
                    printf("Preorder: ");
                    preorder(root1);
                    break;
            case 3:
                    printf("Inorder: ");
                    inorder(root1);
                    break;
            case 4:
                    printf("Postorder: ");
                    postorder(root1);
                    break;
            case 5:
                    printf("No. of nodes: %d",nodes(root1));
                    break;
            case 6:
                    printf("No. of leaf nodes: %d",leaf(root1));
                    break;
            case 7:
                    printf("No. of non leaf nodes:
%d",nonleaf(root1));
```

```
                    break;
            case 8:
                    printf("No. of nodes with only 1 child: 
%d",half(root1));
                    break;
            case 9:
                    printf("Enter the element to be deleted:\n");
                    scanf("%d",&x);
                    delete(root1, x);
                    break;
            case 10:
                    printf("Height: =%3d\n", heightBST(root1));
                    break;




        }
    } while (choice!=0);
}
```

==

EXECUTION-TRAIL

==

```
C:\Users\Harsheet\CLionProjects\cmake-build-debug\second.c.exe


Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

1

Enter no of elements:3

Enter element 1:1

Enter element 2:2

Enter element 3:3


Enter choice:

1.Create

2.Preorder traversal
```

```
3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

2

Preorder: 123

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

3

Inorder: 123

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes
```

```
7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

4

Postorder: 321

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

5

No. of nodes: 1

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST
```

```
6

No. of leaf nodes: 0

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

7

No. of non leaf nodes: 1

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

8

No. of nodes with only 1 child: 2

Enter choice:

1.Create
```

```
2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

9

Enter the element to be deleted:

2


Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

2

Preorder: 13

Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal
```

```
4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

10

Height: =  1


Enter choice:

1.Create

2.Preorder traversal

3.Inorder traversal

4.Postorder traversal

5.Count nodes

6.Count leaf nodes

7.Count non-leaf nodes

8.Count half nodes

9.Delete

10.Height of BST

0


Process finished with exit code 0
```

==

VIVA-VOICE

==

1)What are expression trees?For an arbitrary arithmetic expression[involving all operators] construct its expression tree.

→The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand

For expression (3*2)+((9-5)/2) would be:

2)what are a winner tree and a looser tree?Explain with examples.

→

Winner Trees:

Complete binary tree with n external nodes and n - 1 internal nodes. External nodes represent tournament players. Each internal node represents a match played between its two children; the winner of the match is stored at the internal node. Root has the overall winner.It is easy to see that the winner tree can be computed in O(logn) time.Consider some keys 3, 5, 6, 7, 20, 8, 2, 9

loser tree:

The complete binary tree for n players in which there are n external nodes and n-1 internal nodes then the tree is called loser tree. The loser of the match is stored in internal nodes of the tree. Consider some keys 10, 2, 7, 6, 5, 9, 12,

3)Can you create a BST from its preorder and inorder walk? Show with example traversal for minimum 8 node trees.

→We have already discussed the construction of trees from its preorder and inorder traversal . The idea is similar.

Let us see the process of constructing tree from in[] = {4, 8, 2, 5, 1, 6, 3, 7} and post[] = {8, 4, 5, 2, 6, 7, 3, 1}

1) We first find the last node in post[]. The last node is “1”, we know this value is root as the root always appears at the end of postorder traversal.

2) We search “1” in in[] to find the left and right subtrees of the root. Everything on the left of “1” in in[] is in the left subtree and everything on right is in the right subtree.

```
            1

       /         \

[4, 8, 2, 5]   [6, 3, 7]
```

3) We recur the above process for the following two.

    b) Recur for in[] = {6, 3, 7} and post[] = {6, 7, 3}

        Make the created tree as the right child of the root.

    a) Recur for in[] = {4, 8, 2, 5} and post[] = {8, 4, 5, 2}.

        Make the created tree as the left child of the root.

4)define a forest .Give the algorithm to convert a forest into a binary tree and show a trace on suitable forest.

→Forest is a collection of disjoint trees. In other words, we can also say that forest is a collection of an acyclic graph which is not connected.

The steps to convert a forest into a binary tree are:

(1) First convert each tree to a binary tree;

(2) The first binary tree does not move, starting from the second binary tree, and sequentially taking the root node of the latter binary tree as the front. The right child node of the root node of a binary tree is connected with a line. The binary tree obtained when all the binary trees are connected is the binary tree converted from the forest