Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.: 10	Implementation of AVL Tree	Date:02/05/2024

Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.

Algorithm:

1) Start
2) Define the AVL Node Structure.
3) Implement Rotation Operations (left and right rotations).
4) Insert new nodes into the AVL tree, updating heights and balancing as needed.
5) Delete nodes from the AVL tree, updating heights and balancing as needed.
6) Implement traversal functions (in-order, pre-order, post-order) to navigate through the tree.
7) Implement a search function to find specific elements within the AVL tree.
8) Test the AVL tree implementation with various scenarios.
9) Optionally, optimize the implementation for better performance. 10) Stop

Data Structures Odyssey: Exploring the Foundations of Computing

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>

struct node
{  int data;    struct
node* left;    struct
node* right;
 int ht;
};

struct node* root = NULL;

struct node* create(int);   struct node*
insert(struct node*, int);   struct node*
delete(struct node*, int);   struct node*
search(struct node*, int);   struct node*
rotate_left(struct node*);   struct node*
rotate_right(struct node*);   int
balance_factor(struct node*);   int
height(struct node*);   void
inorder(struct node*);   void
preorder(struct node*);   void
postorder(struct node*);

int main()
{
   int user_choice, data;

   char user_continue = 'y';
struct node* result = NULL;
```

```
    while (user_continue == 'y' || user_continue == 'Y')
    {
        printf("\n\n------- AVL TREE --------\n");
printf("\n1. Insert");          printf("\n2.
Delete");          printf("\n3. Search");
printf("\n4. Inorder");          printf("\n5.
Preorder");          printf("\n6. Postorder");
printf("\n7. EXIT");

        printf("\n\nEnter Your Choice: ");
scanf("%d", &user_choice);

        switch(user_choice)
        {
case 1:
            printf("\nEnter data: ");
scanf("%d", &data);                root
= insert(root, data);
break;

            case 2:
            printf("\nEnter data: ");
scanf("%d", &data);                root
= delete(root, data);
break;


            case 3:
            printf("\nEnter data: ");
scanf("%d", &data);                result
= search(root, data);                if
(result == NULL)
            {
```

```
                    printf("\nNode not found!");
            }
else
            {
               printf("\n Node found");
            }
break;              case 4:
inorder(root);
break;

         case 5:
preorder(root);
break;

         case 6:
            postorder(root);
break;

         case 7:
            printf("\n\tProgram Terminated\n");
            return 1;

         default:

            printf("\n\tInvalid Choice\n");
      }

      printf("\n\nDo you want to continue? ");
scanf(" %c", &user_continue);
   }

   return 0;
}
```

```
struct node* create(int data)
{
   struct node* new_node = (struct node*) malloc (sizeof(struct node));
if (new_node == NULL)
   {
      printf("\nMemory can&'t be allocated\n");
return NULL;
   }
   new_node->data = data;
new_node->left = NULL;
new_node->right = NULL;      return
new_node;
}
struct node* rotate_left(struct node* root)
{
   struct node* right_child = root->right;
root->right = right_child->left;
right_child->left = root;      root->ht =
height(root);      right_child->ht =
height(right_child);    return right_child;

}
struct node* rotate_right(struct node* root)
{
   struct node* left_child = root->left;      root-
>left = left_child->right;      left_child->right =
root;


   root->ht = height(root);      left_child-
>ht = height(left_child);            return
left_child;
}


int balance_factor(struct node* root)
```

```
{     int lh, rh;     if (root
== NULL)          return 0;
if (root->left == NULL)
      lh = 0;     else         lh
= 1 + root->left->ht;     if
(root->right == NULL)
      rh = 0;     else         rh =
1 + root->right->ht;
return lh - rh;
}
int height(struct node* root)
 {     int lh,
rh;     if
(root ==
NULL)

   {
      return 0;
   }
   if (root->left == NULL)
      lh = 0;     else         lh
= 1 + root->left->ht;     if
(root->right == NULL)
      rh = 0;     else         rh =
1 + root->right->ht;


   if (lh > rh)
return (lh);
return (rh);
}


struct node* insert(struct node* root, int data)
{
   if (root == NULL)
```

```
  {
    struct node* new_node = create(data);
if (new_node == NULL)
    {
      return NULL;
    }
    root = new_node;
  }
  else if (data > root->data)
  {
        root->right = insert(root->right, data);
if (balance_factor(root) == -2)
    {
      if (data > root->right->data)
      {
        root = rotate_left(root);
      }
else
      {
        root->right = rotate_right(root->right);
root = rotate_left(root);
      }
    }
}
else
  {
    root->left = insert(root->left, data);
if (balance_factor(root) == 2)
    {
      if (data < root->left->data)
      {
        root = rotate_right(root);
```

```
        }
else
        {
            root->left = rotate_left(root->left);
root = rotate_right(root);
        }
      }
  }
  root->ht = height(root);
return root;

}


struct node * delete(struct node *root, int x)
{
  struct node * temp = NULL;

  if (root == NULL)
  {
    return NULL;
  }

  if (x > root->data)
  {
    root->right = delete(root->right, x);
if (balance_factor(root) == 2)
      {
        if (balance_factor(root->left) >= 0)
        {
            root = rotate_right(root);
        }
else
        {
```

```
            root->left = rotate_left(root->left);
root = rotate_right(root);
         }
      }
   }
   else if (x < root->data)
   {
      root->left = delete(root->left, x);



            if (balance_factor(root) == -2)
      {
         if (balance_factor(root->right) <= 0)
         {
            root = rotate_left(root);
         }
else
{
root->right = rotate_right(root->right);   root
= rotate_left(root);
}
}
} else
{
if (root->right != NULL)
{
temp = root->right;   while
(temp->left != NULL)
temp = temp->left;

root->data = temp->data;   root->right =
delete(root->right, temp->data);   if
(balance_factor(root) == 2)
```

```
{
if (balance_factor(root->left) >= 0)
{
root = rotate_right(root);
}


else
{
root->left = rotate_left(root->left);   root
= rotate_right(root);
}
}
}
else
{
return (root->left);
}
}
root->ht = height(root);   return
(root);
}
struct node* search(struct node* root, int key)
{
 if(root == NULL)
{
return NULL;
}

if(root->data == key)
{
return root;
}
```

```
if(key > root->data)
{
search(root->right, key);
}

else
{       search(root->left,
key);
}
}
void inorder(struct node* root)
{
if (root == NULL)
{
return;
}
inorder(root->left);   printf("%d
", root->data);   inorder(root-
>right);
}
void preorder(struct node* root)
{
   if(root == NULL)
{
return;
}
printf("%d ", root->data); preorder(root-
>left); preorder(root->right);
}
void postorder(struct node* root)
{
if(root == NULL)
```

{

return;

}

postorder(root->left);

postorder(root->right);   printf("%d

", root->data);

}

OUTPUT:

```
------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 15


Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 20


Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 25


Do you want to continue? y


------- AVL TREE --------

1. Insert
```

```
Enter data: 30

 Node found

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 4
15 20 30 30

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 5
20 15 30 30

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 6
15 30 30 20

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
```

RESULT: Thus, the program was successfully executed.