**AVL Tree:**

Problem Statement:

Write a program to

1) Insert a node in AVL tree

2) Delete a node in AVL tree

3) Display the AVL tree by inorder traversal

4) Report the height of any node in AVL tree

5) Report the balance factor of any node in AVL tree

• Output example :

Type 1 to Insert a node in AVL tree

Type 2 to Delete a node in AVL tree

Type 3 to Display the AVL tree by inorder traversal

Type 4 to Report the height of any node in AVL tree

Type 5 to Report the balance factor of any node in AVL tree

Type 6 to terminate

Please give option between 1 to 6

1

Value of Node: 5

Node Inserted

1

Value of Node: 23

Node Inserted

1

Value of Node: 58

Node Inserted

3

5->23->58->

2

Value of Node: 58

Node Deleted

3

5->23->

1

Value of Node: 6

Node Inserted

1

Value of Node: 4

Node Inserted

3

4->5->6->23->

4

Value of Node: 6

Height of node 6 is 3

5

Value of Node: 6

Balance of node 6 is 1

6

Terminated

Proposed C Code:

/\* ------- main.c ------- \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct N

{

int val, height;

struct N \*right;

struct N \*left;

} Node;

// Initializing the tree

Node \*root = NULL;

int max(int a, int b)

{

if (a > b)

{

return a;

}

else

{

return b;

}

}

Node \*createNode(int val)

{

Node \*branch = (Node \*)malloc(sizeof(Node));

branch->val = val;

branch->height = 1;

branch->left = NULL;

branch->right = NULL;

return branch;

}

int height(Node \*branch)

{

if (branch == NULL)

{

return 0;

}

else

{

return branch->height;

}

}

Node \*rightRotation(Node \*y)

{

Node \*x = y->left;

Node \*t3 = x->right;

x->right = y;

y->left = t3;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

Node \*leftRotation(Node \*x)

{

Node \*y = x->right;

Node \*t3 = y->left;

y->left = x;

x->right = t3;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

int getBalance(Node \*branch)

{

if (branch == NULL)

{

return 0;

}

else

{

return height(branch->left) - height(branch->right);

}

}

Node \*insert(Node \*node, int val)

{

if (node == NULL)

{

return createNode(val);

}

if (val < node->val)

{

node->left = insert(node->left, val);

}

else if (val > node->val)

{

node->right = insert(node->right, val);

}

else

{

return node;

}

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

if (balance > 1 && val < node->left->val)

{

return rightRotation(node);

}

if (balance < -1 && val > node->right->val)

{

return leftRotation(node);

}

if (balance > 1 && val > node->left->val)

{

node->left = leftRotation(node->left);

return rightRotation(node);

}

if (balance < -1 && val < node->right->val)

{

node->right = rightRotation(node->right);

return leftRotation(node);

}

return node;

}

Node \*minValueNode(Node \*node)

{

Node \*current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL)

{

current = current->left;

}

return current;

}

Node \*deleteNode(Node \*node, int val)

{

if (node == NULL)

{

return node;

}

if (val < node->val)

{

node->left = deleteNode(node->left, val);

}

else if (val > node->val)

{

node->right = deleteNode(node->right, val);

}

else

{

if ((node->left == NULL) || (node->right == NULL))

{

Node \*temp = node->left ? node->left : node->right;

if (temp == NULL)

{

temp = node;

node = NULL;

}

else

{

node = temp;

}

free(temp);

}

else

{

Node \*temp = minValueNode(node->right);

node->val = temp->val;

node->right = deleteNode(node->right, temp->val);

}

}

if (node == NULL)

{

return node;

}

node->height = max(height(node->left), height(node->right)) + 1;

int balance = getBalance(node);

if (balance > 1 && getBalance(node->left) >= 0)

{

return rightRotation(node);

}

if (balance < -1 && getBalance(node->right) <= 0)

{

return leftRotation(node);

}

if (balance > 1 && getBalance(node->left) < 0)

{

node->left = leftRotation(node->left);

return rightRotation(node);

}

if (balance < -1 && getBalance(node->right) > 0)

{

node->right = rightRotation(node->right);

return leftRotation(node);

}

return node;

}

int heightNode(Node \*node, int val)

{

if (node == NULL)

{

printf("Value doesn't exist\n");

return -1;

}

if (val == node->val)

{

return node->height;

}

else if (val < node->val)

{

return heightNode(node->left, val);

}

else

{

return heightNode(node->right, val);

}

}

int balanceNode(Node \*node, int val)

{

if (node == NULL)

{

printf("Value doesn't exist\n");

return -99;

}

if (val == node->val)

{

return height(node->left) - height(node->right);

}

else if (val < node->val)

{

return balanceNode(node->left, val);

}

else

{

return balanceNode(node->right, val);

}

}

void display(Node \*node)

{

if (node != NULL)

{

display(node->left);

printf("%d->", node->val);

display(node->right);

}

}

int main()

{

int n, x;

printf("Type 1 to Insert a node in AVL tree\nType 2 to Delete a node in AVL tree\nType 3 to Display the AVL tree by inorder traversal \nType 4 to Report the height of any node in AVL tree \nType 5 to Report the balance factor of any node in AVL tree\nType 6 to terminate\n");

printf("Please give option between 1 to 6\n");

do

{

scanf("%d", &n);

switch (n)

{

case 1:

printf("Value of Node: ");

scanf("%d", &x);

root = insert(root, x);

printf("Node Inserted\n");

break;

case 2:

printf("Value of Node: ");

scanf("%d", &x);

root = deleteNode(root, x);

printf("Node Deleted\n");

break;

case 3:

display(root);

printf("\n");

break;

case 4:

printf("Value of Node: ");

scanf("%d", &x);

printf("Height of node %d is %d\n", x, heightNode(root, x));

break;

case 5:

printf("Value of Node: ");

scanf("%d", &x);

printf("Balance of node %d is %d\n", x, balanceNode(root, x));

break;

case 6:

printf("Terminated\n");

break;

}

} while (n != 6);

return 0;

}

/\* ---------------------- \*/

Conclusion:

The proposed algorithm has a runtime of O(logn), where n is the number of nodes in the AVL tree.

Limitations and assumptions for this algorithm include:

1.Here insertion & deletion happens in O(logn) time instead of O(1).

2.Do not try to delete, get height or balance of a node which isn’t exist in the AVL tree.