**Experiment No 3**

**AIM:** Study of Height Balance-AVL Tree as a non-linear Data Structure

**PROBLEM STATEMENT:** Write a C++ Program to create a height balance tree by taking input from user and perform following operation on it.

a. Insertion of a node

b. Searching of a node

c. Display by using any one traversal method

**REQUIREMENT:**Turbo C/ GCC Compiler

**OPERATING SYSTEM:** Windows/Linux/Unix.

**THEORY:**

**Height Balance Tree:**

A height-balanced binary tree is defined as a binary tree in which the height of the left and the right subtree of any node differ by not more than AVL tree, red-black tree are examples of height-balanced trees.

**Conditions for Height-Balanced Binary Tree:**

Following are the conditions for a height-balanced binary tree:

1. The difference between the heights of the left and the right subtree for any node is **not more than one**.
2. The left subtree is balanced.
3. The right subtree is balanced.

**Formula for Height-Balanced Binary Tree:**

***height balance of node  = height of right subtree – height of left subtree***

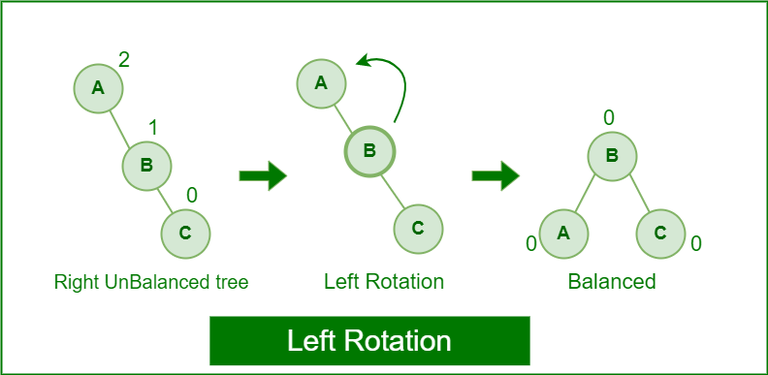
* If the right subtree is taller, the height balance of the node will be positive.
* If the left subtree is taller, the balance of the node will be negative.

**Rotating the subtrees in an AVL Tree:**

An AVL tree may rotate in one of the following four ways to keep itself balanced:

**Left Rotation (LL)**:

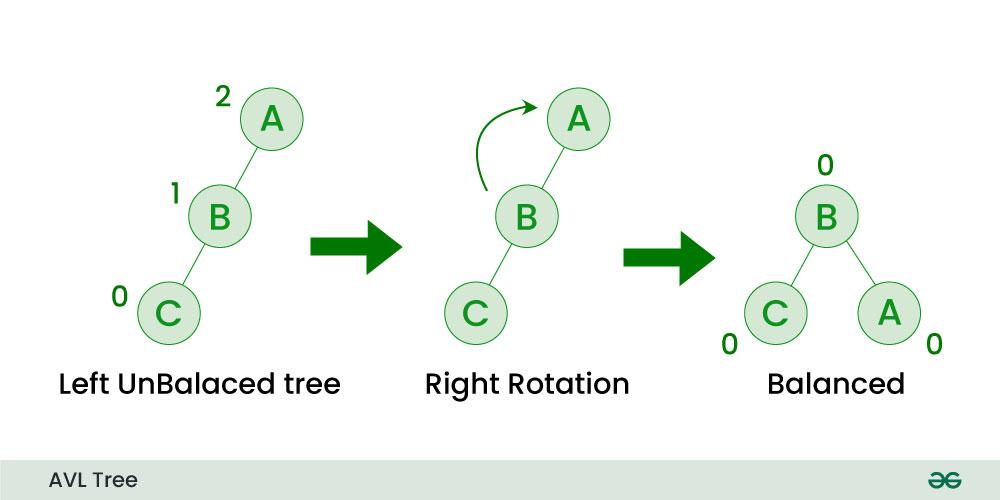
When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



*Left-Rotation in AVL tree*

**Right Rotation (RR)**:

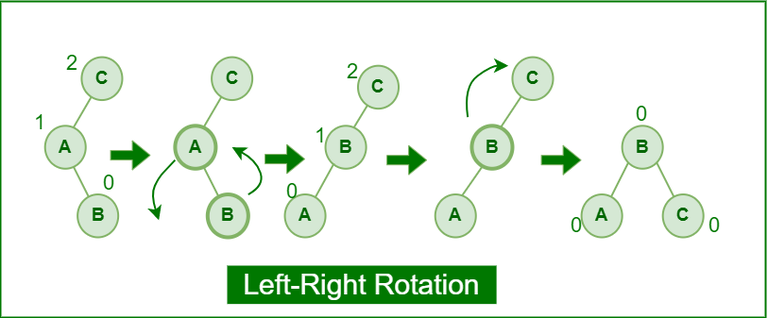
If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



*Right-Rotation in AVL Tree*

**Left-Right Rotation (LR)**:

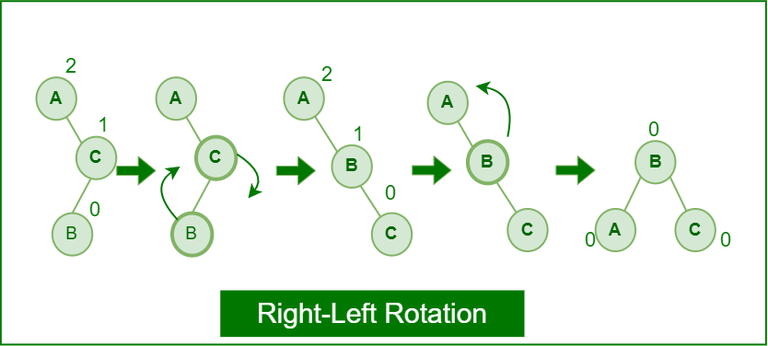
A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



*Left-Right Rotation in AVL tree*

**Right-Left Rotation (RL)**:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



*Right-Left Rotation in AVL tree*

**Advantages of Height-Balanced Binary Tree:**

* It will improve the worst-case lookup time at the expense of making a typical case roughly one lookup less.
* As a general rule, a height-balanced tree would work better when the request frequencies across the data set are more evenly spread,
* It gives better search time complexity.

**Disadvantages of Height-Balanced Binary Tree:**

* Longer running times for the insert and remove operations.
* Must keep balancing info in each node.
* To find nodes to balance, must go back up in the tree.

**CONCLUSION:** Program for AVL-Height balance tree is implemented successfully.

// Online C++ compiler to run C++ program online

#include <iostream>

#define FALSE 0

#define TRUE 1

struct AVLNode

{

int data ;

int balfact ;

struct AVLNode \*left ;

struct AVLNode \*right ;

} ;

struct AVLNode \* buildtree ( struct AVLNode \*, int, int \* ) ;

struct AVLNode \* deldata ( struct AVLNode \*, int, int \* ) ;

struct AVLNode \* del ( struct AVLNode \*, struct AVLNode \*, int \* ) ;

struct AVLNode \* balright ( struct AVLNode \*, int \* ) ;

struct AVLNode \* balleft ( struct AVLNode \*, int \* ) ;

void display ( struct AVLNode \* ) ;

void deltree ( struct AVLNode \* ) ;

int main( )

{

struct AVLNode \*avl = NULL ;

int h,val,ch ;

while(1)

{

std::cout<<"\n1:insert\n2:delete\n3:display\n4:Delete Tree\n5:Exit";

std::cin>>ch;

switch(ch)

{

case 1:

std::cout<<"\nEnter the data";

std::cin>>val;

avl = buildtree ( avl, val, &h ) ;

break;

case 2:

std::cout<<"\nEnter the data";

std::cin>>val;

avl = deldata ( avl,val, &h ) ;

break;

case 3:

std::cout<<"\nAVL tree:\n" ;

display ( avl ) ;

break;

case 4:

deltree(avl);

break;

case 5:

exit(0);

}

}

}

/\* inserts an element into tree \*/

struct AVLNode \* buildtree ( struct AVLNode \*root, int data, int \*h )

{

struct AVLNode \*node1, \*node2 ;

if ( !root )

{

root = ( struct AVLNode \* ) malloc ( sizeof ( struct AVLNode ) ) ;

root -> data = data ;

root -> left = NULL ;

root -> right = NULL ;

root -> balfact = 0 ;

\*h = TRUE ;

return ( root ) ;

}

if ( data < root -> data )

{

root -> left = buildtree ( root -> left, data, h ) ;

/\* If left subtree is higher \*/

if ( \*h )

{

switch ( root -> balfact )

{

case 1:

node1 = root -> left ;

if ( node1 -> balfact == 1 )

{

std::cout<<"\nRight rotation along "<< root -> data ;

root -> left = node1 -> right ;

node1 -> right = root ;

root -> balfact = 0 ;

root = node1 ;

}

else

{

std::cout<<"\nDouble rotation, left along "<< node1 -> data ;

node2 = node1 -> right ;

node1 -> right = node2 -> left ;

std::cout<< " then right along "<< root -> data<<"\n" ;

node2 -> left = node1 ;

root -> left = node2 -> right ;

node2 -> right = root ;

if ( node2 -> balfact == 1 )

root -> balfact = -1 ;

else

root -> balfact = 0 ;

if ( node2 -> balfact == -1 )

node1 -> balfact = 1 ;

else

node1 -> balfact = 0 ;

root = node2 ;

}

root -> balfact = 0 ;

\*h = FALSE ;

break ;

case 0:

root -> balfact = 1 ;

break ;

case -1:

root -> balfact = 0 ;

\*h = FALSE ;

}

}

}

if ( data > root -> data )

{

root -> right = buildtree ( root -> right, data, h ) ;

/\* If the right subtree is higher \*/

if ( \*h )

{

switch ( root -> balfact )

{

case 1:

root -> balfact = 0 ;

\*h = FALSE ;

break ;

case 0:

root -> balfact = -1 ;

break;

case -1:

node1 = root -> right ;

if ( node1 -> balfact == -1 )

{

std::cout<< "\nLeft rotation along "<<root -> data ;

root -> right = node1 -> left ;

node1 -> left = root ;

root -> balfact = 0 ;

root = node1 ;

}

else

{

std::cout<< "\nDouble rotation, right along "<< node1 -> data ;

node2 = node1 -> left ;

node1 -> left = node2 -> right ;

node2 -> right = node1 ;

std::cout<< " then left along "<<root -> data <<"\n" ;

root -> right = node2 -> left ;

node2 -> left = root ;

if ( node2 -> balfact == -1 )

root -> balfact = 1 ;

else

root -> balfact = 0 ;

if ( node2 -> balfact == 1 )

node1 -> balfact = -1 ;

else

node1 -> balfact = 0 ;

root = node2 ;

}

root -> balfact = 0 ;

\*h = FALSE ;

}

}

}

return ( root ) ;

}

/\* deletes an item from the tree \*/

struct AVLNode \* deldata ( struct AVLNode \*root, int data, int \*h )

{

struct AVLNode \*node ;

if ( !root )

{

std::cout<< "\nNo such data." ;

return ( root ) ;

}

else

{

if ( data < root -> data )

{

root -> left = deldata ( root -> left, data, h ) ;

if ( \*h )

root = balright ( root, h ) ;

}

else

{

if ( data > root -> data )

{

root -> right = deldata ( root -> right, data, h ) ;

if ( \*h )

root = balleft ( root, h ) ;

}

else

{

node = root ;

if ( node -> right == NULL )

{

root = node -> left ;

\*h = TRUE ;

free ( node ) ;

}

else

{

if ( node -> left == NULL )

{

root = node -> right ;

\*h = TRUE ;

free ( node ) ;

}

else

{

node -> right = del ( node -> right, node, h ) ;

if ( \*h )

root = balleft ( root, h ) ;

}

}

}

}

}

return ( root ) ;

}

struct AVLNode \* del ( struct AVLNode \*succ, struct AVLNode \*node, int \*h )

{

struct AVLNode \*temp = succ ;

if ( succ -> left != NULL )

{

succ -> left = del ( succ -> left, node, h ) ;

if ( \*h )

succ = balright ( succ, h ) ;

}

else

{

temp = succ ;

node -> data = succ -> data ;

succ = succ -> right ;

free ( temp ) ;

\*h = TRUE ;

}

return ( succ ) ;

}

/\* balances the tree, if right sub-tree is higher \*/

struct AVLNode \* balright ( struct AVLNode \*root, int \*h )

{

struct AVLNode \*node1, \*node2 ;

switch ( root -> balfact )

{

case 1:

root -> balfact = 0 ;

break;

case 0:

root -> balfact = -1 ;

\*h = FALSE ;

break;

case -1:

node1 = root -> right ;

if ( node1 -> balfact <= 0 )

{

std::cout<<"\nLeft rotation along "<< root -> data ;

root -> right = node1 -> left ;

node1 -> left = root ;

if ( node1 -> balfact == 0 )

{

root -> balfact = -1 ;

node1 -> balfact = 1 ;

\*h = FALSE ;

}

else

{

root -> balfact = node1 -> balfact = 0 ;

}

root = node1 ;

}

else

{

std::cout<< "\nDouble rotation, right along "<< node1 -> data ;

node2 = node1 -> left ;

node1 -> left = node2 -> right ;

node2 -> right = node1 ;

std::cout<<" then left along "<< root -> data <<"\n";

root -> right = node2 -> left ;

node2 -> left = root ;

if ( node2 -> balfact == -1 )

root -> balfact = 1 ;

else

root -> balfact = 0 ;

if ( node2 -> balfact == 1 )

node1 -> balfact = -1 ;

else

node1 -> balfact = 0 ;

root = node2 ;

node2 -> balfact = 0 ;

}

}

return ( root ) ;

}

/\* balances the tree, if left sub-tree is higher \*/

struct AVLNode \* balleft ( struct AVLNode \*root, int \*h )

{

struct AVLNode \*node1, \*node2 ;

switch ( root -> balfact )

{

case -1:

root -> balfact = 0 ;

break ;

case 0:

root -> balfact = 1 ;

\*h = FALSE ;

break ;

case 1:

node1 = root -> left ;

if ( node1 -> balfact >= 0 )

{

std::cout<<"\nRight rotation along"<<root -> data ;

root -> left = node1 -> right ;

node1 -> right = root ;

if ( node1 -> balfact == 0 )

{

root -> balfact = 1 ;

node1 -> balfact = -1 ;

\*h = FALSE ;

}

else

{

root -> balfact = node1 -> balfact = 0 ;

}

root = node1 ;

}

else

{

std::cout<<"\nDouble rotation, left along "<< node1 -> data ;

node2 = node1 -> right ;

node1 -> right = node2 -> left ;

node2 -> left = node1 ;

std::cout<< " then right along "<<root -> data<<"\n" ;

root -> left = node2 -> right ;

node2 -> right = root ;

if ( node2 -> balfact == 1 )

root -> balfact = -1 ;

else

root -> balfact = 0 ;

if ( node2-> balfact == -1 )

node1 -> balfact = 1 ;

else

node1 -> balfact = 0 ;

root = node2 ;

node2 -> balfact = 0 ;

}

}

return ( root ) ;

}

/\* displays the tree in-order \*/

void display ( struct AVLNode \*root )

{

if ( root != NULL )

{

display ( root -> left ) ;

std::cout<<"\t"<<root -> data ;

display ( root -> right ) ;

}

}

/\* deletes the tree \*/

void deltree ( struct AVLNode \*root )

{

if ( root != NULL )

{

deltree ( root -> left ) ;

deltree ( root -> right ) ;

}

free ( root ) ;

}