**ADVANCED DATA STRUCTURE**

**GROUP D**

**ASSIGNMENT 9**

**BATCH B1**

**YEAR: 2017-18**

**COLLEGE: VIIT**

**Date Of Completion :30/03/2018**

**Title:**

Height balanced Tree

**Problem Statement:**

A Dictionary stores keywords and amp; its meanings.Provide facility for adding new key-

words, deleting keywords, updating values of any entry. Provide facility to display whole

data sorted in ascending/ Descending order. Also find how many maximum comparisons

may require for finding any keyword. Use Height balance tree and find the complexity for

finding a keyword

**Objective:**

To understand and implement height balanced tree and perform certain operations on it.

**Software And Hardware Requirement:**

1. 64-bit Open source Linux or its derivative.

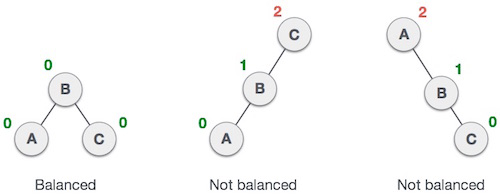
2. Open Source C++ Programming tool like G++/GCC.

**Theory:**

It is observed that BST’s worst-case performance is closest to linear search algorithms, that

is (n). In real-time data, we cannot predict data pattern and their frequencies. So, a need

arises to balance out the existing BST.



Named after their inventor Adelson, Velski and Landis, AVL trees are height balancing

binary search tree. AVL tree checks the height of the left and the right sub-trees and assures

that the difference is not more than 1. This difference is called the Balance Factor.

Here we see that the first tree is balanced and the next two trees are not balanced

1In the second tree, the left subtree of C has height 2 and the right subtree has height 0,

so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is

missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor)

to be only 1.

BalanceFactor = height(left-sutree) height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is bal-

anced using some rotation techniques.

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations

1. Left rotation

2. Right rotation

3. Left-Right rotation

4. Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations.

To have an unbalanced tree, we at least need a tree of height 2.

***Left Rotation***

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right

subtree, then we perform a single left rotation.

In our example, node A has become unbalanced as a node is inserted in the right subtree

of A’s right subtree. We perform the left rotation by making A the left-subtree of B.

***Right Rotation***

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree.

The tree then needs a right rotation.

As depicted, the unbalanced node becomes the right child of its left child by performing

a right rotation.

***Left-Right Rotation***

Double rotations are slightly complex version of already explained versions of rotations.

To understand them better, we should take note of each action performed while rotation.

Let’s first check how to perform Left-Right rotation. A left-right rotation is a combination

of left rotation followed by right rotation

***1. Right Rotation*** : A node has been inserted into the right subtree of the left subtree.

This makes C an unbalanced node. These scenarios cause AVL tree to perform left-

right rotation.

***2. Left Rotation*** : We first perform the left rotation on the left subtree of C. This makes

A, the left subtree of B.

***3. Left Rotation :*** Node C is still unbalanced, however now, it is because of the left-

subtree of the left-subtree.

***4. Right Rotation :*** We shall now right-rotate the tree, making B the new root node

of this subtree. C now becomes the right subtree of its own left subtree.

***5. Balanced Avl Tree :*** The tree is now balanced

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right

rotation followed by left rotation

***1. Left subtree of Right subtree :*** A node has been inserted into the left subtree of

the right subtree. This makes A, an unbalanced node with balance factor 2.

***2. Subtree Right Rotation :*** First, we perform the right rotation along C node, making

C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.

***3. Right Unbalanced Tree :*** Node A is still unbalanced because of the right subtree

of its right subtree and requires a left rotation.

***4. Left Rotation :*** A left rotation is performed by making B the new root node of the

subtree. A becomes the left subtree of its right subtree B.

***5. Balanced AVL Tree :*** The tree is now balanced.

**Algorithm:**

Insertion Operation in AVL Tree

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3:If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be unbalanced.

***Search Operation in AVL Tree***

Step 1:Read the search element from the user

Step 2:Compare, the search element with the value of root node in the tree.

Step 3:If both are matching, then display "Given node found!!!" and terminate the funct

Step 4: If both are not matching, then check whether search element is smaller or larger

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6:If search element is larger, then continue the search process in right subtree.

Step 7:Repeat the same until we found exact element or we completed with a leaf node

Step 8:If we reach to the node with search value, then display "Element is found" and t

Step 9: If we reach to a leaf node and it is also not matching, then display "Element no

***Deletion Operation in AVL Tree***

Case 1: Deleting a leaf node

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function

Case 2: Deleting a node with one child

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

Step 1:Find the node to be deleted using search operation

Step 2:If it has two children, then find the largest node in its left subtree (OR) the

Step 3:Swap both deleting node and node which found in above step.

Step 4:Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5:If it comes to case 1, then delete using case 1 logic.

Step 6:If it comes to case 2, then delete using case 2 logic.

Step 7:Repeat the same process until node is deleted from the tree.

If the tree is balanced after deletion then go for next operation otherwise perform.

**Code :**

#include<iostream>

4#include<string>

using namespace std;

typedef struct node

{

int word;

string mean;

//string word,mean;

struct node \*left,\*right;

}node;

class AVL

{

private:

int hl,hr,cnt;

public:

node \*temp,\*root,\*New;

AVL();

node \*CreateNode();

node \*Insert(node \*root);

node \*LL(node \*parent);

node \*RR(node \*parent);

node \*LR(node \*parent);

node \*RL(node \*parent);

int height(node \*parent);

int Balance\_factor(node \*parent);

void Ascending(node \*root);

void Descending(node \*root);

node \*Balance(node \*parent);

void Search(node \*root);

void Update(node \*root);

};

AVL::AVL()

{

temp=root=NULL;

cnt=0;

}

node \* AVL::CreateNode()

{

node \*New;

New=new node;

cout<<"\nEnter Word :";

cin>>New->word;

//cout<<"\nEnter Meaning :";

//cin>>New->mean;

5New->left=New->right=NULL;

return New;

}

node \*AVL::LL(node \*parent)

{

cout<<"\nLL Rotation";

node \*temp;

temp=parent->left;

parent->left=temp->right;

temp->right=parent;

return temp;

}

node \*AVL::RR(node \*parent)

{

cout<<"\nRR Rotation";

node \*temp;

temp=parent->right;

parent->right=temp->left;

temp->left=parent;

return temp;

}

node \*AVL::RL(node \*parent)

{

cout<<"\nRL Rotation";

node\* temp;

temp=parent->right;

parent->right=LL(temp);

return RR(parent);

}

node \*AVL::LR(node \*parent)

{

cout<<"\nLR Rotation";

node\* temp;

temp=parent->left;

parent->left=RR(temp);

return LL(parent);

}

int AVL::height(node \*parent)

{

if(parent==NULL)

return 0;

else

{

hl=height(parent->left);

hr=height(parent->right);

if(hl > hr)

6return (hl+1);

else

return (hr+1);

}

}

int AVL::Balance\_factor(node \*parent)

{

int hl,hr;

hl=height(parent->left);

hr=height(parent->right);

return (hl-hr);

}

void AVL::Ascending(node \*root)

{

if(root==NULL)

return;

else

{

Ascending(root->left);

cout<<root->word<<" => "<<root->mean<<endl;

Ascending(root->right);

}

}

void AVL::Descending(node \*root)

{

if(root==NULL)

return;

else

{

Descending(root->right);

cout<<root->word<<" => "<<root->mean<<endl;

Descending(root->left);

}

}

node \*AVL::Balance(node \*parent)

{

int bf=Balance\_factor(parent);

if(bf > 1)

{

if(Balance\_factor(parent->left) > 0)

parent=LL(parent);

else

parent=LR(parent);

}

else if(bf < -1)

{

7if(Balance\_factor(parent->right) < 0)

parent=RR(parent);

else

parent=RL(parent);

}

return parent;

}

node \*AVL::Insert(node \*root)

{

New=CreateNode();

if(root==NULL)

{

root=New;

//cout<<"\nRoot Node Created......!!!";

}

else

{

temp=root;

while(1)

{

if(New->word > temp->word)

{

if(temp->right==NULL)

{

temp->right=New;

//cout<<"\n\tBefore Balanced....!!!";

root=Balance(root);

//cout<<"\n\tAfter Balanced....!!!";

cout<<"\nAdded to right ";

break;

}

else

temp=temp->right;

}

else

{

if(temp->left==NULL)

{

temp->left=New;

//cout<<"\n\tBefore Balanced....!!!";

root=Balance(root);

//cout<<"\n\tAfter Balanced....!!!";

cout<<"\nAdded to left ";

break;

}

else

temp=temp->left;

8}

}

}

return root;

}

void AVL::Search(node \*root)

{

int flag=0;

int key;

//string key,NewMean;

cout<<"Enter the Word To Search : ";

cin>>key;

/\*if(root->word==key)

{

cout<<"\n\tWord Found...!!!";

flag=1;

return ;

}\*/

//else

{

temp=root;

while(temp!=NULL)

{

cnt++;

if(temp->word==key)

{

cout<<"\n\tWord Found...!!!";

cout<<"\n\tMeaning : "<<temp->mean;

flag=1;

break;

}

if(key > temp->word)

temp=temp->right;

else

temp=temp->left;

}

}

cout<<"\nNo. of Comparisons : "<<cnt;

if(flag==0)

cout<<"\n\tWord Not Found...!!!";

}

void AVL::Update(node \*root)

{

int flag=0;

int key;

//string key,NewMean;

cout<<"Enter the Word To Update Meaning : ";

9cin>>key;

/\*if(root->word==key)

{

cout<<"\n\tWord Found...!!!";

flag=1;

return ;

}

else\*/

{

temp=root;

while(temp!=NULL)

{

if(temp->word==key)

{

cout<<"\n\tWord Found...!!!";

flag=1;

//cout<<"\nEnter Its New Meaning : ";

//cin>>NewMean;

//temp->mean=NewMean;

break;

}

if(key > temp->word)

temp=temp->right;

else

temp=temp->left;

}

}

if(flag==0)

cout<<"\n\tWord Not Found...!!!";

}

int main()

{

node \*root=NULL;

AVL obj;

int choice;

do

{

cout<<"\n\n\n\*\*\*\*\* MENU \*\*\*\*\*\*\*"<<endl;

cout<<"1. Insert Word ."<<endl;

cout<<"2. Search Word ."<<endl;

cout<<"3. Ascending Order."<<endl;

cout<<"4. Descending Order."<<endl;

cout<<"5. Update Meaning."<<endl;

cout<<"\n\tEnter choice: ";

10cin>>choice;

switch(choice)

{

case 0:

return 0;

case 1:

root=obj.Insert(root);

break;

case 2:

obj.Search(root);

break;

case 3:

obj.Ascending(root);

break;

case 4:

obj.Descending(root);

break;

case 5:

obj.Update(root);

break;

default: cout<<"\nInvalid choice.";

}

}while(1);

return 0;

}

**Output :**

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 1

Enter Word :qwerty

Enter Meaning

:fgggfgf

11\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 1

Enter Word :bfbfb

Enter Meaning

:ererer

Added to left

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 1

Enter Word :ertrt

Enter Meaning

:trtgfbgf

LR Rotation

RR Rotation

LL Rotation

Added to right

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 3

bfbfb => ererer

ertrt => trtgfbgf

12qwerty => fgggfgf

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 4

qwerty => fgggfgf

ertrt => trtgfbgf

bfbfb => ererer

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 2

Enter the Word To Search : qwerty

Word Found...!!!

Meaning : fgggfgf

No. of Comparisons : 2

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 2

Enter the Word To Search : dgfg

No. of Comparisons : 4

Word Not Found...!!!

13\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 5

Enter the Word To Update Meaning : qwerty

Word Found...!!!

Enter Its New Meaning : kjlkorkgo

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice: 3

bfbfb => ererer

ertrt => trtgfbgf

qwerty => kjlkorkgo

\*\*\*\*\* MENU \*\*\*\*\*\*\*

1. Insert Word .

2. Search Word .

3. Ascending Order.

4. Descending Order.

5. Update Meaning.

Enter choice:

**Conclusion:**

Through this assignments we understood and implemented height balanced tree and per-

formed various operations on it.