**Assignment 10**

**Aim**:

A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, 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 learn to implement Height Balance Tree.

**Theory**:

AVL tree is binary search tree with additional property that difference between height of left sub-tree and right sub-tree of any node can’t be more than 1. Here are some key points about [AVL trees](https://www.geeksforgeeks.org/tag/avl-tree/):

* If there are n nodes in AVL tree, minimum height of AVL tree is floor(log2n).
* If there are n nodes in AVL tree, maximum height can’t exceed 1.44\*log2n.
* If height of AVL tree is h, maximum number of nodes can be 2h+1 – 1.
* Minimum number of nodes in a tree with height h can be represented as:  
  N(h) = N(h-1) + N(h-2) + 1 for n>2 where N(0) = 1 and N(1) = 2.
* The complexity of searching, inserting and deletion in AVL tree is O(log n).

**Algorithm:**

Insert and rebalance:

Avlnode \*y=new avlnode

If(k<pp->key) pp->left=y

Else pp->right=y

//adjusting balancing factor from a to pp

int d;

Avlnode\*b,\*c;

If(k>a->key){b=p=a->right;d=-1}

Else{b=p=a->left;d=1}

While(p!=y)

{if(k>p->key){p->bf=-1;p=p->right}

Else

{p->bf=1;p=p->left}

Rotations:

* LL Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

a->left = b->right ;

b->right = a ;

a->bf = b->bf = 0;

* RR Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

// c will point to node which is going to replace unbalanced node

a->right = b -> left ; b -> left = a ;

a->bf = b -> bf = 0

* LR Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

// c will point to node which is going to replace unbalanced node

c = b -> right ; //LR rotation

b-> right = c ->left ;

a->left = c ->right ;

c -> left = b ;

c-> right = a;

switch ( c -> bf )

{ case 1 : a->bf = -1 ; b-> bf = 0 ; break ;

case -1 : b->bf = 1 ; a ->bf = 0 ; break ;

case 0 : a->bf = b->bf = 0 ; break;

}

c->bf = 0 ;

b = c ;

* RL Rotation

// a will point to unbalanced node

// b will point to node next to 'a' where insertion done

// c will point to node which is going to replace unbalanced node

c = b -> left ; //RL rotation

b ->left = c ->right ;

a ->right = c->left ;

c->right = b ;

c->left = a ;

switch (c->bf )

{

case 1: a->bf =0 ; b->bf = -1 ; break ;

case -1 : a->bf = +1 ; b->bf = 0 ; break ;

case 0 : a->bf =b->bf = 0 ;break ;

}

c->bf = 0

b= c ;

**Program:**

#include<iostream>

#include<string.h>

using namespace std;

typedef struct node

{

string word;

string meaning;

int fact;

node \*left;

node \*right;

}node;

class bst

{

public:

int diff(node\*);

void display(node\*);

node\* LL(node \*);

node\* RR(node\*);

node\* RL(node\*);

node\* LR(node\*);

int search(node\*,string);

node\* insert(node\*,string,string);

node\* balance(node\*);

int height(node\*);

void update(node\*,string);

void disprev(node\*);

};

void bst::update(node \*root,string key)

{

int temp;

temp = search(root,key);

if(temp==1)

{

//cout<<"\nCurrent meaning of "<<root->word<<" is "<<temp->meaning;

}

else

{

cout<"\nDoesn't Exist\n";

}

}

int bst::search(node \*root,string key)

{

if(root==NULL)

{

return 0;

}

if(key.compare(root->word)==0)

{

cout<<"\n key is : "<<key ;

cout<<"\n root->word : "<<root->word<<" meaning : "<<root->meaning;

cout<<"\nEnter the new meaning\n";

cin>>root->meaning;

return 1;

}

else if(key.compare(root->word)<0)

{

search(root->left,key);

}

else if(key.compare(root->word)>0)

{

search(root->right,key);

}

//return NULL;

}

int bst::height(node \*temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = height (temp->left);

int r\_height = height (temp->right);

int max\_height = max (l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int bst::diff(node \*temp)

{

int l = height(temp->left);

//cout<<"\nL : "<<l;

int r = height(temp->right);

//cout<<"\nR : "<<r<<endl;

int f = l-r;

return f;

}

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

{

node \*temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

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

{

node \*temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

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

{

node \*temp;

temp = parent->left;

parent->left = RR(temp);

return LL(parent);

}

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

{

node \*temp;

temp = parent->right;

parent->right = LL(temp);

return RR(parent);

}

node\* bst::balance(node \*temp)

{

int bal\_factor = diff (temp);

if (bal\_factor > 1)

{

if (diff (temp->left) > 0)

temp = LL(temp);

else

temp = LR(temp);

}

else if (bal\_factor < -1)

{

if (diff (temp->right) > 0)

temp = RL(temp);

else

temp = RR(temp);

}

return temp;

}

node\* bst::insert(node \*root, string value,string mean)

{

if (root == NULL)

{

root = new node;

root->word = value;

root->meaning = mean;

root->left = NULL;

root->right = NULL;

return root;

}

else if ((value.compare(root->word))<0)

{

root->left = insert(root->left, value,mean);

root = balance (root);

}

else if ((value.compare(root->word))>0)

{

root->right = insert(root->right, value,mean);

root = balance (root);

}

return root;

}

void bst::display(node\* root)

{

if(root)

{

display(root->left);

cout<<root->word<<" means "<<root->meaning<<endl;

display(root->right);

}

}

void bst::disprev(node \*root)

{

if(root)

{

disprev(root->right);

cout<<root->word<<" means "<<root->meaning<<endl;

disprev(root->left);

}

}

int main()

{

bst obj;

node \*rt;

int ch;

int choice;

char x;

string key,mean;

string keys;

rt = NULL;

do

{

cout<<"\nEnter Choice\n";

cout<<"\n1.Insert the data into dictionary. \n2.Update the dictionary ";

cout<<"\n3.Display the data in ascending order.\n4.Display the data in descending order";

cin>>choice;

switch(choice)

{

case 1:

do

{

cout<<"\nEnter the word\n";

cin>>key;

cout<<"\nEnter the meaning of the word\n";

cin>>mean;

rt = obj.insert(rt,key,mean);

//cout<<"\nROOT is : "<<rt->word;

cout<<"\nDo you want to insert more words? Y or N\n";

cin>>x;

}while(x=='y'||x=='Y');

break;

case 2:

cout<<"\nEnter the word whose meaning is supposed to be updated\n";

cin>>keys;

obj.update(rt,keys);

break;

case 3:

cout<<"\nDisplay :\n";

obj.display(rt);

break;

case 4:

cout<<"Reverse display:\n";

obj.disprev(rt);

break;

}

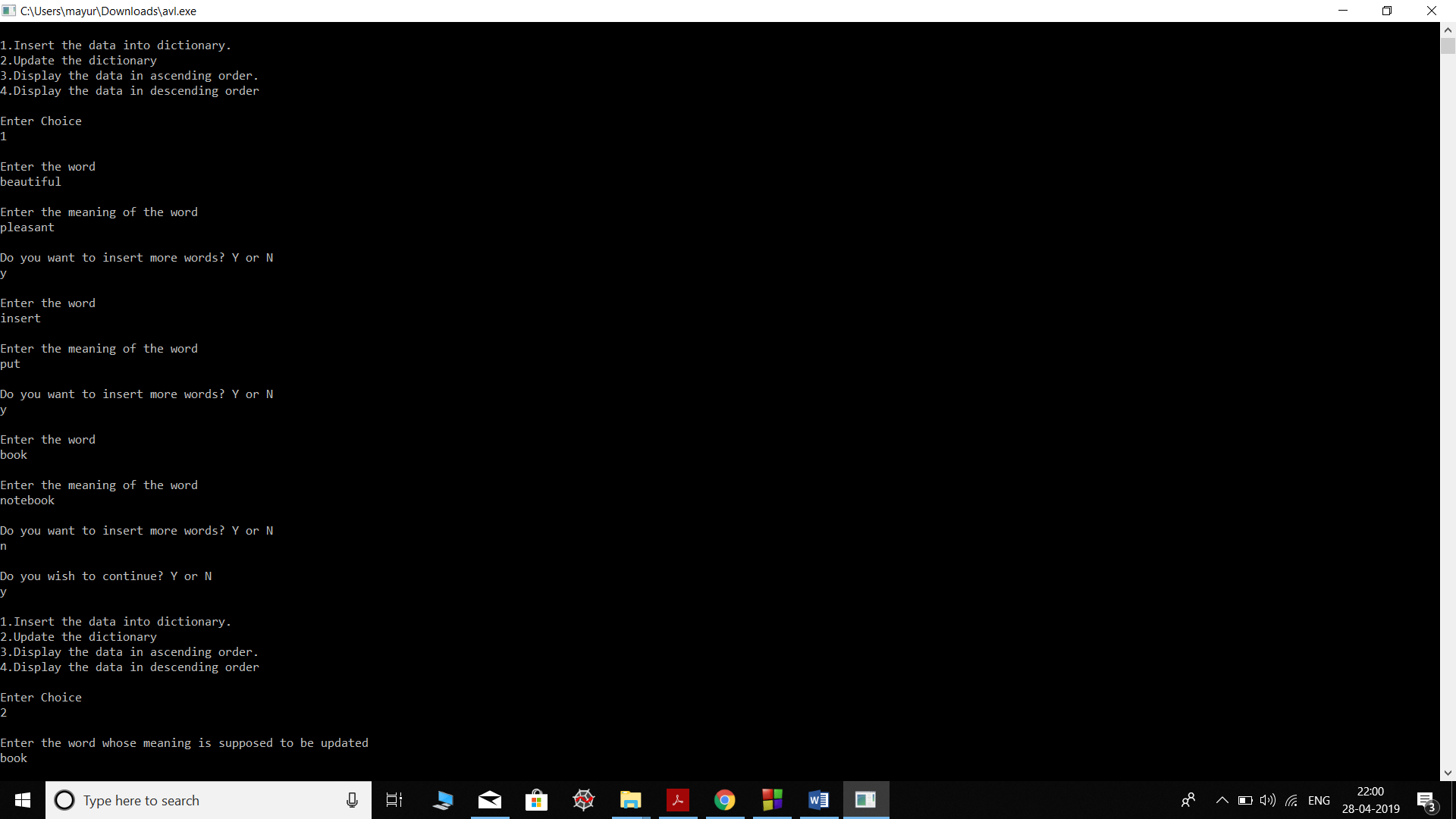
cout<<"\nDo you wish to continue? Y or N \n";

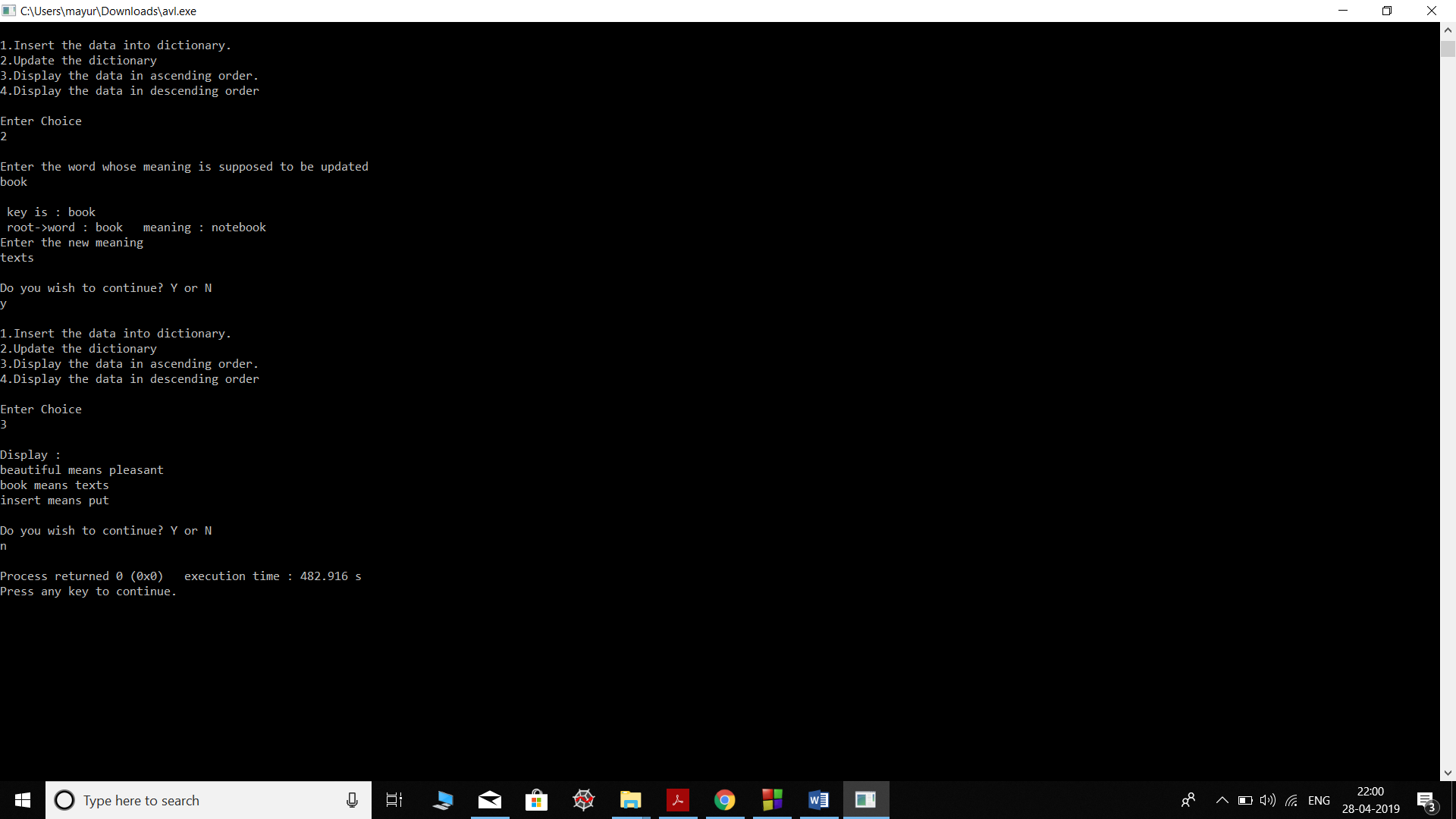
cin>>x;

}while(x=='y' || x=='Y');

return 0;

}

**Output:**



**Conclusion:**

Thus we have implemented a dictionaries and its functions using a height balanced trees.