DCSE, CEG, ANNA UNIVERSITY CHENNAI - 600025

MINI PROJECT

ME – CSE OPERATIONS RESEARCH

SEARCHING IN CONTACTS USING AVL TREE

2021184026
BALASUBRAMANIAN PT
9940661358
balap1311@gmail.com

INSTRUCTOR: **DR. SHILOAH ELIZABETH**

TABLE OF CONTENTS

ABSTRACT

Contact book searching specifically required for contacts applications which runs in every user's mobile. Since it is the basic feature in this application, we are using **AVL TREE** data structure for this feature which requires minimum time complexity of **O(logN).** Here we consider each contact as a node and each node has name of the user and contact number of them. Storage of each contacts will be based on the weight of user's name & weight of their numbers. Based on these two values, we construct the avl tree and here are some features added with them which are additional features we will be using later.

CHAPTER 1

INTRODUCTION

AVL Tree may be defined as height balanced binary search tree during which each node is related to a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree. A dictionary structure is created and the information of the each contacts is inserted into it. This mapping is done by a searching algorithm. AVL tree is employed as the searching algorithm. The searching algorithm matches the output member B for the given input member A. AVL Tree is said to be a self-balancing binary search tree. AVL trees take O(log n) time for insert, delete and search operations. The tree must be re balanced after insert and delete operations.

CHAPTER 2

RELATED WORKS

There are already multiple applications which use AVL for in-memory sorting and dictionary, Here we mainly focused on searching in contacts. Those contact applications are already proposed by many phone vendors like Samsung, Apple, google etc. where they provide all these features with their theme. Hereby I am proposing that an AVL tree approach is a binary tree which ensures performance of operations such as insertion and retrieval in logarithmic time. In this system , I have added some features like Adding new contact, deleting a contact and updating them.

CHAPTER 3

PROBLEM STATEMENT AND SOLUTION

Searching a contact needs requires more time than o(log n) i.e o(n) where n is the number of contacts. For this issue if we AVL tree we can do that searching in O(logN) time which is much efficiency than the any other data structure. So we build an Contact Book Tree which can retrieve a contact with higher efficiency.

CHAPTER 4

AVL TREE OPERATIONS

4.1 AVL-OPERATIONS

4.1.1 INSERT:

Insert the details of the new contact into the tree using basic Binary Search Tree insertion method. Check for the Balance Factor of every level, after insertion. If the Balance Factor of every level is 0 or 1 or -1 then go for next operation. If the Balance Factor of any level is other than 0 or 1 or -1 then that tree is imbalanced. In this case, perform required Rotation operation to make it balanced and proceed for next operation.

4.1.2 SEARCH:

The Search operation can be performed with the key value that needs to be searched is provided as an input in the given AVL tree. The search operation returns the information of the contact from a tree if the value matches with the key value. If the key value does not match with any contact, in the tree no information is returned. This means that the contact is not available in the system.

4.1.3 DELETE:

Deletion is more complex than insertion in avl trees. The deletion of a contact from the system might decrease the height of the tree which may cause unbalanced tree structure. So the required rotation operation is performed to rebalance the tree. The deletion removes all the details from the tree.

There are three possibilities:

1) **Contact** to be deleted is the leaf**:** Simply remove from the tree

2) Contact to be deleted has only one child***:*** Copy the child to the word and delete the child

3) Contact to be deleted has two children: Find in-order successor of the word. Copy contents of the in-order successor to the word and delete the in-order successor. Note that in-order predecessor can also be used.

4.1.4 TRAVERSAL:

One of the important operations on an AVL tree dictionary implementation is to find a way to traverse all the words in the tree. As we know traversing a linked list or array is quite easy. We just start from the first contact and traverse linearly until we come to the end of the list. But, it is not so trivial in an AVL tree. There are several types of traversing trees**.**

TYPES:

- **INORDER**
- **PREORDER**
- **POSTORDER**

CHAPTER 5

ROTATION

5.1 ROTATION

The AVL tree may become unbalanced after insert and delete operation. Rotation is the basic mechanism that rebalances the unbalanced tree.

The rotation is an adjustment to the tree, around an item, that maintains the required ordering of items. A rotation takes a constant time. Nodes that are not in the sub tree of the item rotated about are unaffected by the rotation.

There are four kinds of rotation.

5.1.1 Left Rotation

In Left Rotation operation the tree is rotated left side to rebalance the tree. Consider a tree of three nodes of the following structure.

- A - Root node.
- B - A's Right child node.
- C - B's Right child node.

After rotation the resulting tree structure will be as following.

- B becomes the root node.
- A becomes the B's Left child node.
- C will be B's Right Child node.

5.1.2 Right Rotation

In Right Rotation operation the tree is rotated right side to rebalance the tree. A right rotation is mirror to left rotation operation.

After rotation the resulting tree structure will be as following.

- B becomes the root node.
- A becomes the B's Right child node.
- C will be B's Left Child node.

5.1.3 Right-Left Rotation

In this case two rotation need to be performed in-order to balance the tree. First the tree is rotated right side and then to the left side.

The first rotation performed on B and C nodes in the tree.

• A - Root node

• C - A’s Right child node.

• B - will become C’s Right child node.

Then the tree is rotated left side in-order to obtain the balanced tree.

• C - Root node.

• A - will become C’s left child node.

• B - will become C’s right child node.

5.1.4 Left-Right Rotation

A Left-Right rotation is the mirror image of Right-Left Rotation operation. In this case two rotation need to be performed in-order to balance the tree. The tree is first rotated on left side and then on right side

After left rotation performed on B and C nodes, the tree structure looks like the following.

• A - Root node

• C - will become the left child node of A

• B - will become the left child node of C.

Now the tree is rotated right side in-order to obtain the balanced AVL tree structure.

• C - Root node

• B - will become the left child node of C

• A - will become the Right child node of C.

CHAPTER 6

IMPLEMENTATION

6.1 INITIAL SETUP:

- The project was implemented in C++ Programming Language.
- The OS used was Windows 11.
- C++ packages like bits/stdc++.h were used to reduce the time wasted in doing chores; especially when your rank is time sensitive.

6.2 TOOLS:

- Visual Studio Code
- C/C++ IntelliSense, debugging extension.

6.3 CODE

```
#include<iostream>
#include<vector>
using namespace std;

class Node
{
public:
long int key;
Node *left;
Node *right;
string name;
int height;
};
class BNode{
public:
Node *contact;
BNode *left;
BNode *right;
};
```

```
class BST{
public:
BNode* head;
BNode* buildTree(BNode* root,Node* contact){
if (root==NULL) {
BNode* newNode = new BNode();
newNode->contact = contact;
newNode->left = NULL;
newNode->right = NULL;
return newNode;
}
if (contact->name > root->contact->name) {
root->right = buildTree(root->right, contact);
}
else {
root->left = buildTree(root->left, contact);
}
return root;
}
void printPhoneNumber(Node* contact){
string stuff(29, '-');
string space1(5,' ');
string space2(16,' ');
cout<<stuff<<endl;
for(int i=0;i<2;i++){
if(i == 0){
cout<<"|"<<space1<<"Name:"<<contact-
>name<<space2<<"|"<<endl;
cout<<"|"<<space1<<"Number:"<<contact-
>key<<space1<<"|"<<endl;
}
}
}
void printHelper(BNode* root){
if(root != NULL){
printHelper(root->left);
```

```
printPhoneNumber(root->contact);
printHelper(root->right);
}
}
void printBST(BNode* root){
string stuff(29, '-');
printHelper(root);
cout<<stuff<<endl;
}
};
class NumberArranger{
public:
int height(Node *N)
{
if (N == NULL)
return 0;
return N->height;
}

int max(int a, int b)
{
return (a > b)? a : b;
}

Node* newNode(long int key,string name)
{
Node* node = new Node();
node->key = key;
node->left = NULL;
node->right = NULL;
node->height = 1;
node->name = name;
return(node);
}

Node *rightRotate(Node *y)
```

```
{
Node *x = y->left;
Node *T2 = x->right;

x->right = y;
y->left = T2;

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

Node *leftRotate(Node *x)
{
Node *y = x->right;
Node *T2 = y->left;

y->left = x;
x->right = T2;

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 *N)
{
if (N == NULL)
return 0;
return height(N->left) -
height(N->right);
```

```
}

Node* insert(Node* node,string name,long int key)
{
if (node == NULL)
return(newNode(key,name));

if (key < node->key)
node->left = insert(node->left,name, key);
else if (key > node->key)
node->right = insert(node->right,name, key);
else
return node;

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

int balance = getBalance(node);

if (balance > 1 && key < node->left->key)
return rightRotate(node);

if (balance < -1 && key > node->right->key)
return leftRotate(node);

if (balance > 1 && key > node->left->key)
{
node->left = leftRotate(node->left);
return rightRotate(node);
}

if (balance < -1 && key < node->right->key)
{
node->right = rightRotate(node->right);
return leftRotate(node);
}
```

```
return node;
}

Node * minValueNode(Node* node)
{
Node* current = node;

while (current->left != NULL)
current = current->left;

return current;
}

Node* deleteNode(Node* root,long int key){
if (root == NULL)
return root;

if ( key < root->key )
root->left = deleteNode(root->left, key);

else if( key > root->key )
root->right = deleteNode(root->right, key);

else
{
if( (root->left == NULL) ||
(root->right == NULL) )
{
Node *temp = root->left ?
root->left :
root->right;

if (temp == NULL)
{
temp = root;
```

```
root = NULL;
}
else
*root = *temp;
free(temp);
}
else
{
Node* temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right,
temp->key);
}
}

if (root == NULL)
return root;

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

int balance = getBalance(root);

if (balance > 1 &&
getBalance(root->left) >= 0)
return rightRotate(root);

if (balance > 1 &&
getBalance(root->left) < 0)
{
root->left = leftRotate(root->left);
return rightRotate(root);
}
```

```
if (balance < -1 &&
getBalance(root->right) <= 0)
return leftRotate(root);

if (balance < -1 &&
getBalance(root->right) > 0)
{
root->right = rightRotate(root->right);
return leftRotate(root);
}

return root;
}
BNode* treeSort(Node* root)
{
BST bst;
BNode* head = NULL;
vector<Node*> contactArray;
contactArray = buildBst(root,contactArray);
cout<<"size:"<<contactArray.size()<<endl;
for(int i=0;i<contactArray.size();i++){
Node* current = contactArray[i];
head = bst.buildTree(head,current);
}
return head;
}
vector<Node*> buildBst(Node* root,vector<Node*> contact){
if(root != NULL)
{
contact.push_back(root);
contact = buildBst(root->left,contact);
contact = buildBst(root->right,contact);
return contact;
}
else{
return contact;
```

```
}
}
void preOrder(Node *root)
{
if(root != NULL)
{
printHelper(root);
preOrder(root->left);
preOrder(root->right);
}
}
void printHelper(Node* contact){
string stuff(29, '-');
string space1(5,' ');
string space2(16,' ');
cout<<stuff<<endl;
for(int i=0;i<2;i++){
if(i == 0){
cout<<"|"<<space1<<"Name:"<<contact-
>name<<space2<<"|"<<endl;
cout<<"|"<<space1<<"Number:"<<contact-
>key<<space1<<"|"<<endl;
}
}
}
void printPhoneNumber(Node* contact){
string stuff(29, '-');
preOrder(contact);
cout<<stuff<<endl;
}
Node* searchNode(Node* root,long int key){
if(root != NULL)
{
if(root->key == key){
Node* temp = root;
temp->left = NULL;
```

```
temp->right = NULL;
printPhoneNumber(temp);
return temp;
}
else if(root->key > key){
return searchNode(root->left,key);
}
else{
return searchNode(root->right,key);
}
}
cout<<"No contacts Available"<<endl;
return root;
}
Node* updateNode(Node* root, long int key,string newName,long int
newKey){
if (root == NULL)
return root;
if ( key < root->key )
root->left = updateNode(root->left, key,newName,newKey);
else if( key > root->key )
root->right = updateNode(root->right, key,newName,newKey);
else
{
root->name = newName;
root->key = newKey;
}
if (root == NULL)
return root;
int balance = getBalance(root);
if (balance > 1 && key < root->left->key)
return rightRotate(root);
if (balance < -1 && key > root->right->key)
return leftRotate(root);
if (balance > 1 && key > root->left->key)
{
```

```
root->left = leftRotate(root->left);
return rightRotate(root);
}
if (balance < -1 && key < root->right->key)
{
root->right = rightRotate(root->right);
return leftRotate(root);
}
return root;
}
};
class NameArranger{
public:
int height(Node *N)
{
if (N == NULL)
return 0;
return N->height;
}
int max(int a, int b)
{
return (a > b)? a : b;
}
Node* newNode(string key,long int phone)
{
Node* node = new Node();
node->key = phone;
node->left = NULL;
node->right = NULL;
node->height = 1;
node->name = key;
return(node);
}
Node *rightRotate(Node *y)
{
Node *x = y->left;
```

```
Node *T2 = x->right;

x->right = y;
y->left = T2;
y->height = max(height(y->left),
height(y->right)) + 1;
x->height = max(height(x->left),
height(x->right)) + 1;
return x;
}
Node *leftRotate(Node *x)
{
Node *y = x->right;
Node *T2 = y->left;
y->left = x;
x->right = T2;
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 *N)
{
if (N == NULL)
return 0;
return height(N->left) -
height(N->right);
}
Node* insert(Node* node, string key,long int phone)
{
if (node == NULL)
return(newNode(key,phone));
if (key < node->name)
node->left = insert(node->left, key,phone);
else if (key > node->name)
```

```
node->right = insert(node->right, key,phone);
else
return node;
node->height = 1 + max(height(node->left),
height(node->right));
int balance = getBalance(node);
if (balance > 1 && key < node->left->name)
return rightRotate(node);
if (balance < -1 && key > node->right->name)
return leftRotate(node);

if (balance > 1 && key > node->left->name)
{
node->left = leftRotate(node->left);
return rightRotate(node);
}
if (balance < -1 && key < node->right->name)
{
node->right = rightRotate(node->right);
return leftRotate(node);
}
return node;
}
Node * minValueNode(Node* node)
{
Node* current = node;
while (current->left != NULL)
current = current->left;
return current;
}
Node* deleteNode(Node* root, string key){
if (root == NULL)
return root;
if ( key < root->name )
root->left = deleteNode(root->left, key);
else if( key > root->name )
```

```
root->right = deleteNode(root->right, key);
else
{
if( (root->left == NULL) ||
(root->right == NULL) )
{
Node *temp = root->left ?
root->left :
root->right;
if (temp == NULL)
{
temp = root;
root = NULL;
}
else
*root = *temp;
free(temp);
}
else
{
Node* temp = minValueNode(root->right);
root->name = temp->name;
root->right = deleteNode(root->right,
temp->name);
}
}
if (root == NULL)
return root;
root->height = 1 + max(height(root->left),
height(root->right));
int balance = getBalance(root);
if (balance > 1 &&
getBalance(root->left) >= 0)
return rightRotate(root);
if (balance > 1 &&
getBalance(root->left) < 0)
```

```
{
root->left = leftRotate(root->left);
return rightRotate(root);
}
if (balance < -1 &&
getBalance(root->right) <= 0)
return leftRotate(root);

if (balance < -1 &&
getBalance(root->right) > 0)
{
root->right = rightRotate(root->right);
return leftRotate(root);
}
return root;
}
Node* updateNode(Node* root, string key,string newName,long int
newKey){
if (root == NULL)
return root;

if ( key < root->name )
root->left = updateNode(root->left, key,newName,newKey);

else if( key > root->name )
root->right = updateNode(root->right, key,newName,newKey);

else
{
root->name = newName;
root->key = newKey;
}

if (root == NULL)
return root;
```

```
int balance = getBalance(root);

if (balance > 1 && key < root->left->name)
return rightRotate(root);

if (balance < -1 && key > root->right->name)
return leftRotate(root);

if (balance > 1 && key > root->left->name)
{
root->left = leftRotate(root->left);
return rightRotate(root);
}

if (balance < -1 && key < root->right->name)
{
root->right = rightRotate(root->right);
return leftRotate(root);
}

return root;
}
Node* search(Node* root,string key){
if (root == NULL)
cout<<"Zero contact"<<endl;

if ( key < root->name ){
return search(root->left, key);
}
else if( key > root->name ){
return search(root->right, key);
}
else if(key == root->name)
{
cout<<"number found"<<endl;
Node* current = root;
```

```
current->left = NULL;
current->right = NULL;
printPhoneNumber(root);
return root;
}
return root;
}
Node* searchNode(Node* root,string key){
if(root != NULL)
{
if(root->name == key){
Node* temp = root;
temp->left = NULL;
temp->right = NULL;
printPhoneNumber(temp);
return temp;
}
else if(root->name > key){
return searchNode(root->left,key);
}
else{
return searchNode(root->right,key);
}
}
cout<<"No contacts Available"<<endl;
return root;
}
void printHelper(Node* contact){
string stuff(29, '-');
string space1(5,' ');
string space2(16,' ');
cout<<stuff<<endl;
for(int i=0;i<2;i++){
if(i == 0){
cout<<"|"<<space1<<"Name:"<<contact-
>name<<space2<<"|"<<endl;
```

```
cout<<"|"<<space1<<"Number:"<<contact-
>key<<space1<<"|"<<endl;
}
}
}
void printPhoneNumber(Node* contact){
string stuff(29, '-');
preOrder(contact);
cout<<stuff<<endl;
}
BNode* treeSort(Node* root)
{
BST bst;
BNode* head = NULL;
vector<Node*> contactArray;
contactArray = buildBst(root,contactArray);
cout<<"size:"<<contactArray.size()<<endl;
for(int i=0;i<contactArray.size();i++){
Node* current = contactArray[i];
head = bst.buildTree(head,current);
}
return head;
}
vector<Node*> buildBst(Node* root,vector<Node*> contact){
if(root != NULL)
{
contact.push_back(root);
contact = buildBst(root->left,contact);
contact = buildBst(root->right,contact);
return contact;
}
else{
return contact;
}
}
void preOrder(Node *root)
```

```
{
if(root != NULL)
{
printHelper(root);
preOrder(root->left);
preOrder(root->right);
}
}
};
int main(){
long int newNumber;
string searchName;
string newName;
string delName;
long int searchNo;
long int delNo;
Node* searchDel = NULL;
Node* x;
Node* y;
long int updateNo;
long int newNoupdate;
string newNameUpdate;
string updateName;
string stuff(29, '-');
NumberArranger contactBook1;
NameArranger contactBook2;
Node* root1 = NULL;
Node* root2 = NULL;
BST binaryTree;
BNode* bst;
root1 = contactBook1.insert(root1,"b",9940661358);
root1 = contactBook1.insert(root1,"s" ,9442585822);
root1 = contactBook1.insert(root1,"k",9940135866);
root1 = contactBook1.insert(root1,"m", 4146259699);
root1 = contactBook1.insert(root1,"n", 7397063193);
root1 = contactBook1.insert(root1,"p", 9442234397);
```

```
root1 = contactBook1.insert(root1,"t", 9080891358);
root1 = contactBook1.insert(root1,"a", 9442159699);
root1 = contactBook1.insert(root1,"i", 9487419569);
root2 = contactBook2.insert(root2,"b",9940661358);
root2 = contactBook2.insert(root2,"s" ,9442585822);
root2 = contactBook2.insert(root2,"k",9940135866);
root2 = contactBook2.insert(root2,"m", 4146259699);
root2 = contactBook2.insert(root2,"n", 7397063193);
root2 = contactBook2.insert(root2,"p", 9442234397);
root2 = contactBook2.insert(root2,"t", 9080891358);
root2 = contactBook2.insert(root2,"a", 9442159699);
root2 = contactBook2.insert(root2,"i", 9487419569);
int ch;
int updateCheckParams;
cout<<"1.insert new contact"<<endl;
cout<<"2.delete contact by Name"<<endl;
cout<<"3.delete contact by Number"<<endl;
cout<<"4.search by Number"<<endl;
cout<<"5.search by Name"<<endl;
cout<<"6.update contact by Name"<<endl;
cout<<"7.update contact by Number"<<endl;
cout<<"8.print contact"<<endl;
cout<<"9.preorderTraversal of Trees"<<endl;
cout<<"0.exit"<<endl;
while(1){
cout<<"Enter the choice:";
cin>>ch;
switch (ch)
{
case 1:
cout<<"enter Name:";
cin>>newName;
cout<<endl;
cout<<"enter Number:";
cin>>newNumber;
root1 = contactBook1.insert(root1,newName,newNumber);
```

```
root2 = contactBook2.insert(root2,newName,newNumber);
break;
case 2:
cout<<"enter name to be deleted:";
cin>>delName;
searchDel = contactBook2.searchNode(root2,delName);
root1 = contactBook1.deleteNode(root1,searchDel->key);
root2 = contactBook2.deleteNode(root2,delName);
break;
case 3:
cout<<"enter number to be deleted:";
cin>>delNo;
x = contactBook1.searchNode(root1,delNo);
cout<<"Name:"<<x->name<<" no:"<<x->key<<endl;
root1 = contactBook1.deleteNode(root1,delNo);
root2 = contactBook2.deleteNode(root2,x->name);
break;
case 4:
cout<<"enter number to be search:";
cin>>searchNo;
y = contactBook1.searchNode(root1,searchNo);
break;
case 5:
cout<<"enter number to be search:";
cin>>searchName;
x = contactBook2.searchNode(root2,searchName);
break;
case 6:
cout<<"enter name to update"<<endl;
cin>>updateName;
cout<<"Enter 1 to update only Number"<<endl;
cout<<"Enter 2 to update only Name"<<endl;
cout<<"Enter 0 to update both Number and Name"<<endl;
cin>>updateCheckParams;
if(updateCheckParams == 1){
cout<<"enter New No to update:";
```

```
cin>>newNoupdate;
x = contactBook2.searchNode(root2,updateName);
if(x !=NULL){
newNameUpdate = x->name;
root1 = contactBook1.deleteNode(root1,x->key);
root2 = contactBook2.deleteNode(root2,x->name);
root1 = contactBook1.insert(root1,newNameUpdate,newNoupdate);
root2 = contactBook2.insert(root2,newNameUpdate,newNoupdate);
}
}
if(updateCheckParams == 2){
cout<<"enter New Name to update:";
cin>>newNameUpdate;
x = contactBook2.searchNode(root2,updateName);
if(x != NULL){
newNoupdate = x->key;
root1 = contactBook1.deleteNode(root1,x->key);
root2 = contactBook2.deleteNode(root2,x->name);
root1 = contactBook1.insert(root1,newNameUpdate,newNoupdate);
root2 = contactBook2.insert(root2,newNameUpdate,newNoupdate);
}
}
if(updateCheckParams == 0){
cout<<"enter New Name to update:";
cin>>newNameUpdate;
cout<<"enter number to update:";
cin>>newNoupdate;
x = contactBook2.searchNode(root2,updateName);
if(x != NULL){
root1 = contactBook1.deleteNode(root1,x->key);
root2 = contactBook2.deleteNode(root2,x->name);
root1 = contactBook1.insert(root1,newNameUpdate,newNoupdate);
root2 = contactBook2.insert(root2,newNameUpdate,newNoupdate);
}
}
break;
```

```
case 7:
cout<<"enter number to update"<<endl;
cin>>updateNo;
cout<<"Enter 1 to update only Number"<<endl;
cout<<"Enter 2 to update only Name"<<endl;
cout<<"Enter 0 to update both Number and Name"<<endl;
cin>>updateCheckParams;
if(updateCheckParams == 1){
cout<<"enter New No to update:";
cin>>newNoupdate;
y = contactBook1.searchNode(root1,updateNo);
if(y!=NULL){
newNameUpdate = y->name;
root1 = contactBook1.deleteNode(root1,y->key);
root2 = contactBook2.deleteNode(root2,y->name);
root1 = contactBook1.insert(root1,newNameUpdate,newNoupdate);
root2 = contactBook2.insert(root2,newNameUpdate,newNoupdate);
}
}
if(updateCheckParams == 2){
cout<<"enter New Name to update:";
cin>>newNameUpdate;
y = contactBook1.searchNode(root1,updateNo);
newNoupdate = updateNo;
if(y!=NULL){
root1 = contactBook1.deleteNode(root1,y->key);
root2 = contactBook2.deleteNode(root2,y->name);
root1 = contactBook1.insert(root1,newNameUpdate,newNoupdate);
root2 = contactBook2.insert(root2,newNameUpdate,newNoupdate);
}
}
if(updateCheckParams == 0){
cout<<"enter New Name to update:";
cin>>newNameUpdate;
cout<<"enter number to update:";
cin>>newNoupdate;
```

```
y = contactBook1.searchNode(root1,updateNo);
if(y!=NULL){
root1 = contactBook1.deleteNode(root1,y->key);
root2 = contactBook2.deleteNode(root2,y->name);
root1 = contactBook1.insert(root1,newNameUpdate,newNoupdate);
root2 = contactBook2.insert(root2,newNameUpdate,newNoupdate);
}
}
break;
case 8:
bst = contactBook1.treeSort(root1);
binaryTree.printBST(bst);
break;
case 9:
contactBook1.preOrder(root1);
cout<<stuff<<endl;
cout<<endl;
contactBook2.preOrder(root2);
cout<<stuff<<endl;
break;
case 0:
exit(1);
}
}
return 0;
}
```

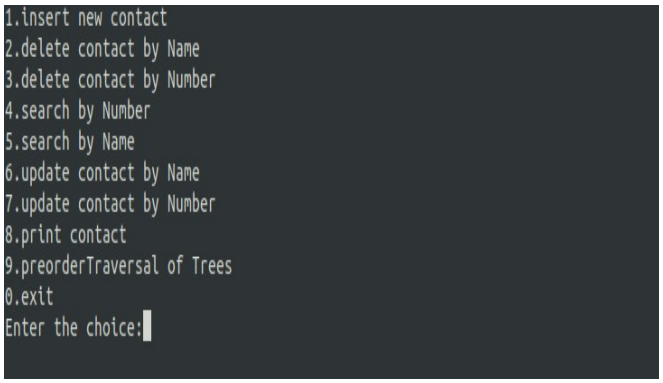
CHAPTER 7

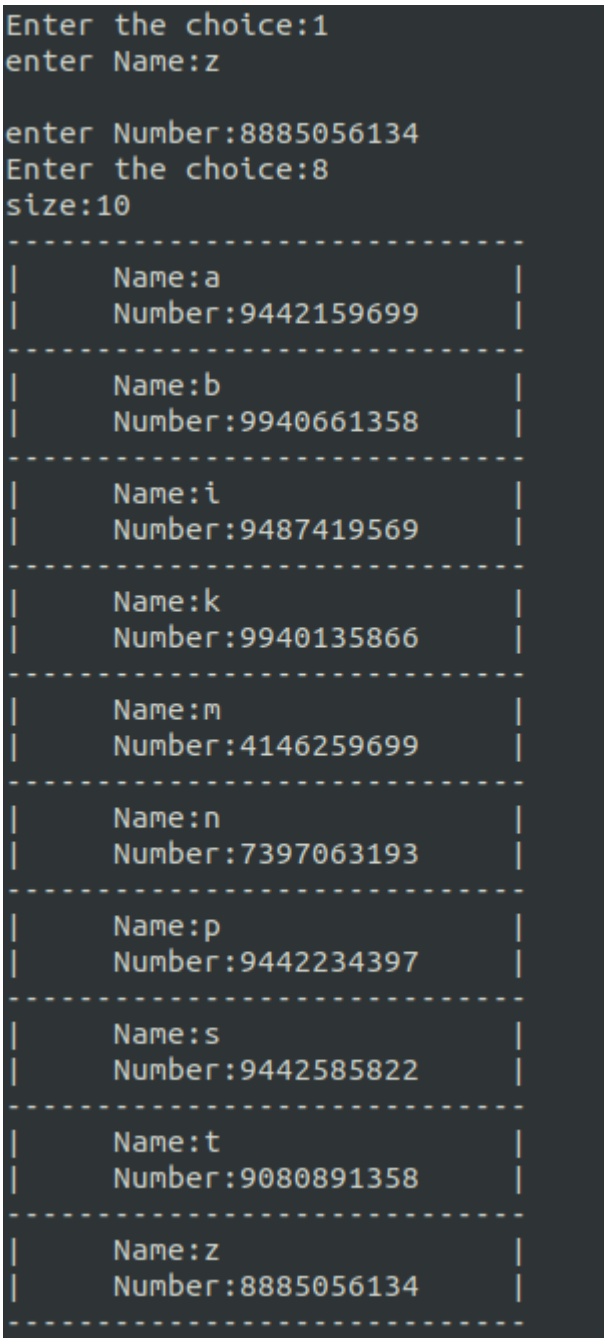
RESULTS

Searching in contact Book using AVL tree data structure gives much efficiency than normal searching. Some features like update,delete and insert also takes minimum time as we can see here.

SCREENSHOTS:

Insert:





Search:

```
Enter the choice:5
enter number to be search:a
-----------------------------
|     Name:a                |
|     Number:9442159699     |
-----------------------------
Enter the choice:5
enter number to be search:z
No contacts Available
```

```
Enter the choice:4
enter number to be search:8904561358
No contacts Available
Enter the choice:4
enter number to be search:9442585822
-----------------------------
|     Name:s                |
|     Number:9442585822     |
-----------------------------
```

Delete:

```
Enter the choice:2
enter name to be deleted:z
-----------------------------
|     Name:z                |
|     Number:8885056134     |
-----------------------------
```

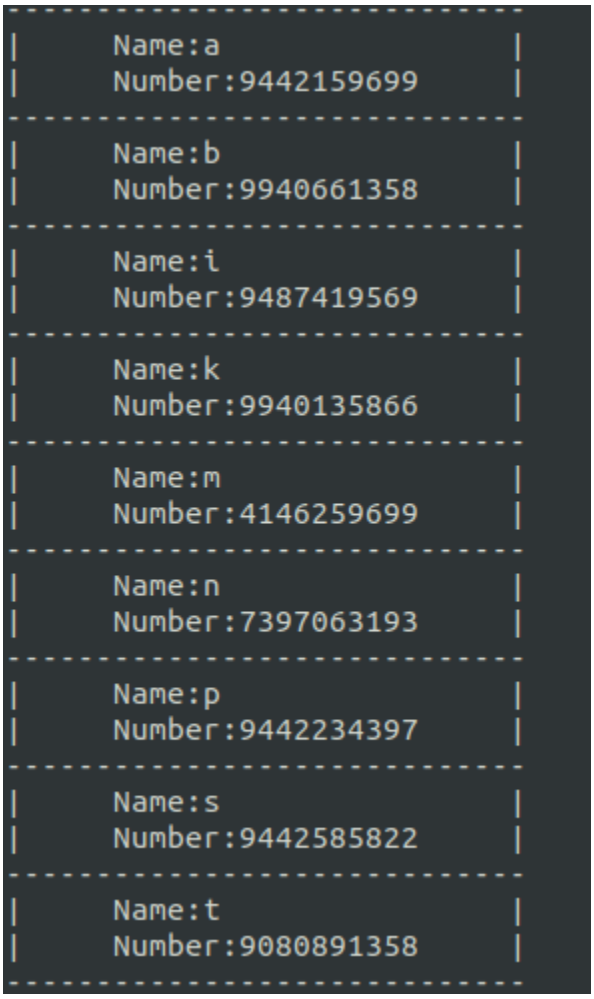
```
Enter the choice:3
enter number to be deleted:9940661358
-----------------------------
|     Name:b                |
|     Number:9940661358     |
-----------------------------
```

Update:

```
Enter the choice:6
enter name to update
t
Enter 1 to update only Number
Enter 2 to update only Name
Enter 0 to update both Number and Name
0
enter New Name to update:z
enter number to update:9940661358
```

```
-----------------------------
|     Name:z                |
|     Number:9940661358     |
-----------------------------
```

Display:



GIT HUB LINK:

https://github.com/balasubramanianperumal/contactsBook.git

CHAPTER 8

FUTURE SCOPE

As for as we mainly worked in searching inserting and deletion of contact,need to attach DB to it(Suggested No SQL DB such as MongoDB,Firebase etc),and some small features like availability of number in whats app or any other social message applications. Main extension of this project will be converting the db structure to B-Tree where we can retrieve multiple values for single input(example: if search value is b, then I will be getting contacts whose name starts with b same as name, number starts with specific digit like 9).This process can be done with less complication by using adding database to it.

CHAPTER 9

REFERENCES

[1] Google contacts,Apple contacts applications etc.

[2] https://www.geeksforgeeks.org/avl-tree/

Appendix A: Project Approval

The undersigned acknowledge they have completed the project SEARCHING IN CONTACTS USING AVL TREE and agree with the approach it presents.

Signature: Date: 16-05-2022

Name : BALASUBRAMANIAN PT

Appendix B: Key Terms

The following table provides definitions for terms relevant to this document.

Term	Definition
AVL tree	It is a type of binary search tree that is height balanced so that the BST is not highly unbalanced towards one side.
Balance factor	Balance factor is the difference between the heights of the left and right children of a node. It is used to balance the node of an AVL tree.
Binary search tree	It is a type of binary tree which is constructed in such a way that the least element is the leftmost and the highest element is the rightmost and all other node between them are arranged in ascending order from left to right