Node.h:

#pragma once

#include<iostream>

#include<cstring>

#include "Contacts.h"

using namespace std;

class Node {

public:

Contacts d1;

Node\* left;

Node\* right;

Node(Contacts s);

Node() {};

};

Node.cpp:

#include "Node.h"

Node::Node(Contacts s) {

d1 = s;

left = NULL;

right = NULL;

}

Contacts.h:

#pragma once

#include<iostream>

#include<cstring>

using namespace std;

class Contacts {

public:

string Name;

string PhNumber;

Contacts();

Contacts(string name,string phoneNumber);

};

Contacts.cpp:

#include "Contacts.h"

Contacts::Contacts(string name, string phoneNumber)

: Name(name), PhNumber(phoneNumber) {

}

Contacts::Contacts() {}

BST.H:

#pragma once

#include"Node.h"

#include<iostream>

#include<conio.h>

#include<iomanip>

#include<string>

#include<windows.h>

#include<fstream>

#include<iostream>

#include<conio.h>

#include<iomanip>

#include<string>

#include<cmath>

#include<windows.h>

#include<fstream>

#include<dos.h>

#include<MMSystem.h>

#include "vfw.h"

#include <cstdio>

#pragma comment(lib, "Vfw32.lib")

#include"Node.h"

using namespace std;

class BinarySt

{

private:

Node\* root;

public:

BinarySt();

bool isempty();

void insert(string value, string name);

void inorder(Node\* ROOT);

void search(string d, Node\* ROOT);

Node\* getroot();

void deleteBST(Node\* ROOT);

int calculateallnodes(Node\* root);

bool dupliate(Node\* root, string value);

Node\* deleteNode(Node\* root, string value);

Node\* findMinNode(Node\* node1);

void modifydata(Node\* root, string value,string n);

void boldAccountdetails();

void choice();

void developersdetail();

void clearscreen();

void pic();

};

BST.CPP

#include"BST.h"

BinarySt::BinarySt() {

root = nullptr;

}

bool BinarySt::isempty() {

return (root == nullptr);

}

void BinarySt::insert(string value, std::string name) {

Node\* newNode = new Node;

newNode->d1.PhNumber = value;

newNode->d1.Name = name;

newNode->left = nullptr;

newNode->right = nullptr;

if (isempty()) {

root = newNode;

}

else {

Node\* temp = root;

while (true) {

if (value < (temp->d1.PhNumber)) {

if (temp->left == nullptr) {

temp->left = newNode;

break;

}

else {

temp = temp->left;

}

}

else if (value > temp->d1.PhNumber) {

if (temp->right == nullptr) {

temp->right = newNode;

break;

}

else {

temp = temp->right;

}

}

else {

std::cout << "Duplicate value. Ignoring insertion." << std::endl;

break;

}

}

}

}

void BinarySt::inorder(Node\* ROOT) {

if (ROOT != nullptr) {

inorder(ROOT->left);

std::cout << "Phone Number: " << ROOT->d1.PhNumber << ", Name: " << ROOT->d1.Name << std::endl;

inorder(ROOT->right);

}

}

void BinarySt::search(string d, Node\* ROOT) {

if (ROOT == nullptr) {

std::cout << "Contact not found." << std::endl;

return;

}

if (d == ROOT->d1.PhNumber) {

std::cout << "Phone Number: " << ROOT->d1.PhNumber << ", Name: " << ROOT->d1.Name << std::endl;

}

else if (d < ROOT->d1.PhNumber) {

search(d, ROOT->left);

}

else {

search(d, ROOT->right);

}

}

Node\* BinarySt::getroot() {

return root;

}

void BinarySt::deleteBST(Node\* ROOT) {

if (ROOT != nullptr) {

deleteBST(ROOT->left);

deleteBST(ROOT->right);

delete ROOT;

}

root = nullptr;

}

int BinarySt::calculateallnodes(Node\* root) {

if (root == nullptr) {

return 0;

}

return calculateallnodes(root->left) + calculateallnodes(root->right) + 1;

}

bool BinarySt::dupliate(Node\* root, string value) {

if (root == nullptr) {

return false;

}

if (root->d1.PhNumber == value) {

return true;

}

return (dupliate(root->left, value) || dupliate(root->right, value));

}

Node\* BinarySt::deleteNode(Node\* root, string value) {

if (root == nullptr) {

return root;

}

else if (value < root->d1.PhNumber) {

root->left = deleteNode(root->left, value);

}

else if (value > root->d1.PhNumber) {

root->right = deleteNode(root->right, value);

}

else {

if (root->left == nullptr && root->right == nullptr) {

delete root;

root = nullptr;

}

else if (root->left == nullptr) {

Node\* temp = root;

root = root->right;

delete temp;

}

else if (root->right == nullptr) {

Node\* temp = root;

root = root->left;

delete temp;

}

else {

Node\* temp = findMinNode(root->right);

root->d1 = temp->d1;

root->right = deleteNode(root->right, temp->d1.PhNumber);

}

}

return root;

}

Node\* BinarySt::findMinNode(Node\* node1) {

Node\* current = node1;

while (current && current->left != nullptr) {

current = current->left;

}

return current;

}

void BinarySt::modifydata(Node\* root, string value, string n) {

if (root == nullptr) {

std::cout << "Contact not found." << std::endl;

return;

}

if (value == root->d1.PhNumber) {

root->d1.Name = n;

std::cout << "Contact modified successfully." << std::endl;

}

else if (value < root->d1.PhNumber) {

modifydata(root->left, value, n);

}

else {

modifydata(root->right, value, n);

}

}

void BinarySt::boldAccountdetails() {

static CONSOLE\_FONT\_INFOEX fontex;

fontex.cbSize = sizeof(CONSOLE\_FONT\_INFOEX);

HANDLE hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

GetCurrentConsoleFontEx(hOut, 0, &fontex);

fontex.FontWeight = 220;

fontex.dwFontSize.X = 25;

fontex.dwFontSize.Y = 25;

SetCurrentConsoleFontEx(hOut, NULL, &fontex);

}

void BinarySt::choice() {

std::cout << " \n\t\tMAIN MENU" << std::endl;

std::cout << "\n\t\t ENTER 1 TO INSERT CONTACT " << std::endl;

std::cout << "\n\t\t ENTER 2 TO PRINT CONTACTS" << std::endl;

std::cout << "\n\t\t ENTER 3 TO SEARCH CONTACT " << std::endl;

std::cout << "\n\t\t ENTER 4 TO CALCULATE TOTAL CONTACTS " << std::endl;

std::cout << "\n\t\t ENTER 5 FOR SEEING DUPLICATE " << std::endl;

std::cout << "\n\t\t ENTER 6 TO RESET YOUR CONTACT BOOK " << std::endl;

std::cout << "\n\t\t ENTER 7 TO EXIT CONTACT BOOK " << std::endl;

std::cout << "\n\t\t ENETR 8 TO LET US TAKE YOUR PICTURE" << std::endl;

std::cout << "\n\t\t ENTER 9 TO DELETE ANY CONTACT " << std::endl;

std::cout << "\n\t\t ENTER 10 TO UPDATE ANY CONTACT " << std::endl << std::endl;

}

void BinarySt::developersdetail() {

std::cout << "\t\t\t DEVELOPERS" << std::endl;

std::cout << "\t\t 1) QURRAT UL AIN " << std::endl;

std::cout << "\t\t 2) ABDUL MOIZ KHALID" << std::endl;

}

void BinarySt::clearscreen() {

system("cls");

std::cout << "JUST WAIT A SECOND......" << std::endl;

Sleep(3000);

system("cls");

}

void BinarySt::pic() {

HWND hCam = capCreateCaptureWindow(

L"hoven",

WS\_CHILD,

0, 0, 0, 0,

::GetDesktopWindow(), 0);

if (capDriverConnect(hCam, 0)) {

capFileSaveDIB(hCam, L"shot.bmp");

std::cout << ("PIC HAS BEEN TAKEN ") << std::endl << std::endl;

}

else {

std::cout << ("Check camera?") << std::endl;

}

DestroyWindow(hCam);

}

HASH.CPP:

#include "HeashTable.h"

#include "Node.h"

Hash::Hash() {

capacity = 10; // Initial capacity of the hash table

hashtable = new Contacts \* [capacity];

HashTsize = 0;

for (int i = 0; i < capacity; i++) {

hashtable[i] = nullptr; // Initialize the hash table with nullptr to indicate empty slots

}

}

Hash::~Hash() {

clearAll();

delete[] hashtable;

}

int Hash::hashFunction1(string key) {

int sum = 0;

for (char c : key) {

sum += c;

}

return sum % capacity; // First hash function using sum of ASCII values

}

int Hash::hashFunction2(string key) {

int len = key.length();

return capacity - (len % capacity); // Second hash function using length of the key

}

int Hash::quadraticProbe(int index, int attempt) {

return (index + (attempt \* attempt)) % capacity; // Find the next available slot using quadratic probing

}

void Hash::resize() {

int newCapacity = primeGreaterThanOrEqual(capacity \* 2); // Double the capacity and find the nearest prime number

Contacts\*\* newHashtable = new Contacts \* [newCapacity];

for (int i = 0; i < newCapacity; i++) {

newHashtable[i] = nullptr; // Initialize the new hash table with nullptr to indicate empty slots

}

for (int i = 0; i < capacity; i++) {

if (hashtable[i] != nullptr) {

string& key = hashtable[i]->Name;

int index = hashFunction1(key);

int step = hashFunction2(key);

int j = 0;

while (newHashtable[(index + j \* step) % newCapacity] != nullptr) {

j++;

}

newHashtable[(index + j \* step) % newCapacity] = hashtable[i];

}

}

delete[] hashtable;

hashtable = newHashtable;

capacity = newCapacity;

}

bool Hash::checkDuplicate(string name) {

int index = hashFunction1(name);

int step = hashFunction2(name);

int i = 0;

while (hashtable[(index + i \* step) % capacity] != NULL) {

if (hashtable[(index + i \* step) % capacity]->Name == name) {

return true;

}

i++;

}

return false;

}

void Hash::insert(string name,string phoneNumber) {

if (HashTsize >= capacity / 2) {

resize();

}

if (checkDuplicate(name)) {

cout << "Contact with name " << name << " already exists. Duplicate entries are not allowed." << endl;

return;

}

Contacts\* Obj = new Contacts(name, phoneNumber);

string key = Obj->Name;

int index = hashFunction1(key);

int step = hashFunction2(key);

int i = 0;

while (hashtable[(index + i \* step) % capacity] != NULL) {

int newIndex = quadraticProbe(index, i);

i++;

}

hashtable[(index + i \* step) % capacity] = Obj;

HashTsize++;

}

void Hash::remove(string NameToRemove) {

int index = hashFunction1(NameToRemove);

int step = hashFunction2(NameToRemove);

int i = 0;

while (hashtable[(index + i \* step) % capacity] != NULL) {

if (hashtable[(index + i \* step) % capacity]->Name == NameToRemove) {

delete hashtable[(index + i \* step) % capacity]; // Free the memory occupied by the contact object

hashtable[(index + i \* step) % capacity] = NULL; // Mark the slot as empty

HashTsize--;

return;

}

i++;

}

cout << "Contact with name " << NameToRemove << " not found." << endl;

}

Contacts\* Hash::search(string name) {

int index = hashFunction1(name);

int step = hashFunction2(name);

int i = 0;

while (hashtable[(index + i \* step) % capacity] != NULL) {

if (hashtable[(index + i \* step) % capacity]->Name == name) {

return hashtable[(index + i \* step) % capacity];

}

i++;

}

return NULL;

}

void Hash::clearAll() {

for (int i = 0; i < capacity; i++) {

if (hashtable[i] != NULL) {

delete hashtable[i]; // Free the memory occupied by the contact objects

hashtable[i] = NULL; // Mark the slots as empty

}

}

HashTsize = 0;

}

void Hash::display() {

for (int i = 0; i < capacity; i++) {

if (hashtable[i] != NULL) {

cout << "Index " << i << ": " << hashtable[i]->Name << ", " << hashtable[i]->PhNumber << endl;

}

}

}

float Hash::loadFactor() {

return static\_cast<float>(HashTsize) / capacity;

}

void Hash::readFromFile(string filename) {

ifstream inputFile(filename);

if (!inputFile) {

cout << "Failed to open the file: " << filename <<endl;

return;

}

string name, phoneNumber;

while (getline(inputFile, name) && getline(inputFile, phoneNumber)) {

insert(name, phoneNumber);

}

cout << " " << " " << " " << "Contacts loaded from file." << endl;

inputFile.close();

}

int Hash::primeGreaterThanOrEqual(int n) {

while (!isPrime(n)) {

n++;

}

return n;

}

bool Hash::isPrime(int n) {

if (n <= 1) {

return false;

}

for (int i = 2; i \* i <= n; i++) {

if (n % i == 0) {

return false;

}

}

return true;

}

HASH.H:

#pragma once

#include <iostream>

#include <conio.h>

#include <fstream>

#include<chrono>

#include<cmath>

#include <string>

#include "Contacts.h"

using namespace std;

class Hash {

public:

Contacts\*\* hashtable;

int capacity;

int HashTsize;

Hash();

~Hash();

int hashFunction1(string key);

int hashFunction2(string key);

int quadraticProbe(int index, int attempt);

void resize();

bool checkDuplicate(string name);

void insert(string name,string phoneNumber);

void remove(string name);

Contacts\* search(string name);

void clearAll();

void display();

float loadFactor();

int primeGreaterThanOrEqual(int n);

bool isPrime(int n);

void readFromFile(string filename);

void measureTimeComplexity();

void SortandSaveinFile();

};

#include"HeashTable.h"

#include<chrono>

#include<cmath>

#include<cstdlib>

#include"BST.h"

using namespace std;

void displayMenuHash()

{

system("cls"); // Clear the screen

cout <<" " << " " << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONTACT MANAGEMENT SYSTEM WIT HASH TABLES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << " " << " " << "1. Read Contacts from File" << endl;

cout << " " << " " << "2. Add Contact" << " " << endl;

cout << " " << " " << "3. Remove Contact" << endl;

cout << " " << " " << "4. Search Contact" << " " << endl;

cout << " " << " " << "5. Display All Contacts" << endl;

cout << " " << " " << "6. Exit" << " " << " " << endl;

cout << " " << " " << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << " " << " " << "Enter your choice: ";

}

void loading(int milliseconds)

{

cout << "Loading";

for (int i = 0; i < 2; i++)

{

Sleep(milliseconds);

cout << ".";

}

cout << endl;

Sleep(milliseconds);

}

int main()

{

system("color 70");

cout << " " << " " << " " << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* WELCOME TO CONTACT MANAGEMENT SYSTEM \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

int a;

cout << "Would you like to proceed with Hash tables or Binary trees?" << endl;

cout << "Press 1 for Binary Trees" << endl;

cout << "Press 2 for Hash Tbles" << endl;

while (true) {

cin >> a;

if (a == 1 || a == 2) {

break;

}

else {

cout << "Please enter a valid number (1 or 2): ";

}

}

if (a == 2) {

Hash hashTable;

string filename;

char choice;

system("color 70");

bool exitProgram = false;

while (!exitProgram)

{

displayMenuHash();

int option;

cin >> option;

switch (option)

{

case 1:

{

cout << "Enter the filename to read contacts from: ";

cin >> filename;

loading(200);

auto start = chrono::high\_resolution\_clock::now();

hashTable.readFromFile(filename);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

cout << "Contacts loaded from file succesfully." << endl;

break;

}

case 2:

{

string name, phoneNumber;

cout << "Enter the name: ";

cin.ignore();

getline(cin, name);

cout << "Enter the phone number: ";

getline(cin, phoneNumber);

auto start = chrono::high\_resolution\_clock::now();

hashTable.insert(name, phoneNumber);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

cout << "Contact added." << endl;

break;

}

case 3:

{

string name;

cout << "Enter the name to remove: ";

cin.ignore();

getline(cin, name);

auto start = chrono::high\_resolution\_clock::now();

hashTable.remove(name);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

break;

}

case 4:

{

string name;

cout << "Enter the name to search: ";

cin.ignore();

getline(cin, name);

auto start = chrono::high\_resolution\_clock::now();

Contacts\* contact = hashTable.search(name);

if (contact != nullptr)

{

cout << "Contact found: " << contact->Name << ", " << contact->PhNumber << endl;

}

else

{

cout << "Contact not found." << endl;

}

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

break;

}

case 5:

{

cout << "Contacts in the Hash Table:" << endl;

auto start = chrono::high\_resolution\_clock::now();

hashTable.display();

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

break;

}

case 6:

{

system("color 07"); // Change console color back to default

auto start = chrono::high\_resolution\_clock::now();

cout << "The load factor of the hash table is: " << hashTable.loadFactor() << endl;

cout << "Exiting the program...";

hashTable.clearAll();

exitProgram = true;

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

break;

}

default:

{

cout << "Invalid option. Please try again." << endl;

break;

}

}

if (!exitProgram)

{

char continueChoice;

cout << "Do you want to continue? (Y/N): ";

cin >> continueChoice;

if (continueChoice != 'Y' && continueChoice != 'y')

exitProgram = true;

}

}

}

else if (a == 1) {

system("color 70");

BinarySt Object;

int option, totalcontacts;

bool duplicatetest;

string name, phone, name2, n;

char a, b;

Object.boldAccountdetails();

cout << " \t\t\t\t\t WEILCOME TO OUR CONTACT BOOK LIST WITH BINARY SEARCH TREES" << endl << endl << endl << endl;

system("pause");

Object.clearscreen();

do {

Object.choice();

cin >> option;

system("cls");

switch (option)

{

case 1:

{

do

{

cout << "ENTER NAME OF CONTACT " << endl;

cin >> name;

cout << "ENTER PHONR NUMBER OF CONTACT" << endl;

cin >> phone;

auto start = chrono::high\_resolution\_clock::now();

Object.insert(phone, name);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

cout << endl << endl;

cout << "IF YOU WANT TO ENTER ANOTHER CONTACT PRESS Y" << endl;

cin >> a;

system("cls");

} while (a == 'y');

system("pause");

break;

}

case 2:

{

auto start = chrono::high\_resolution\_clock::now();

Object.inorder(Object.getroot());

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

cout << endl;

system("pause");

system("cls");

break;

}

case 3: {

cout << "ENTER PHONE NUMBER YOU WANT TO SEARCH" << endl;

cin >> phone;

auto start = chrono::high\_resolution\_clock::now();

Object.search(phone, Object.getroot());

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

system("pause");

system("cls");

break;

}

case 4:

{

auto start = chrono::high\_resolution\_clock::now();

totalcontacts = Object.calculateallnodes(Object.getroot());

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

cout << "TOTAL NUMBER OF CONTACTS ARE " << totalcontacts << endl;

system("pause");

system("cls");

break;

}

case 5:

{

cout << "ENTER NUMBER TO SEE IT HAS DUPLICATE OR NOT" << endl;

cin >> phone;

auto start = chrono::high\_resolution\_clock::now();

duplicatetest = Object.dupliate(Object.getroot(), phone);

Object.clearscreen();

if (duplicatetest == true)

{

cout << "YOUR DATA IS ALL READY PRESENT SO CANT ENTER DUPLICATE DATA" << endl;

}

else

{

cout << "DATA U ENTER IS NOT PRESENT IN FILE " << endl;

}

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

system("pause");

system("cls");

break;

}

case 6:

{

cout << "DELETING ALL DATA " << endl;

Object.clearscreen();

auto start = chrono::high\_resolution\_clock::now();

Object.deleteBST(Object.getroot());

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

Object.clearscreen();

cout << "ALL DATA IS DELETED" << endl;

exit(0);

}

case 7:

{

Object.developersdetail();

exit(0);

}

case 8:

{auto start = chrono::high\_resolution\_clock::now();

Object.pic();

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

Object.clearscreen();

cout << "\t\t PIC HAS BEES TAKEN" << endl;

system("pause");

system("cls");

break;

}

case 9:

{

cout << "ENTER NUM OF CONTACT YOU WANT TO DELETE" << endl;

cin >> n;

auto start = chrono::high\_resolution\_clock::now();

Object.deleteNode(Object.getroot(), n);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

system("pause");

Object.clearscreen();

break;

}

case 10:

{

cout << "ENTER PHONE NUM YOU WANT TO REPLACE" << endl;

cin >> phone;

Object.clearscreen();

cout << "ENTER NAME YOU WANT TO ENTER INSTEAD OF PREVIOUS NAME" << endl;

cin >> name;

auto start = chrono::high\_resolution\_clock::now();

Object.modifydata(Object.getroot(), phone, name);

auto end = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(end - start);

cout << "Time complexity of Function: " << duration.count() << " microseconds" << endl;

Object.clearscreen();

cout << "\n\n CONTACT IS UPDATED" << endl;

system("pause");

system("cls");

break;

}

default:

cout << "YOU ENTER WRONG NUM ENTER NUM BETWEEN 1-8" << endl;

}

cout << "ENTER Y TO GO BACK TO INTERFACE" << endl;

cin >> b;

system("cls");

} while (b == 'y');

}

return 0;

}