Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Стратегічного управління "

|  |  |
| --- | --- |
|  |  |

**Звіт до лабораторної роботи №3**

з дисципліни "Алгоритми на структури даних"

Виконав: студент групи КН-321А

Карауланов А. О.

Перевірив: старший викладач

Мошко Є.О.

Харків 2022

**Тема лабораторної роботи.** Бінарні дерева та червоно-чорні дерева

**Мета:** Набуття практичних вмінь та навичок опрацювання динамічних структур даних, представлених у вигляді бінарних та червоно-чорних дерев.

**Порядок виконання роботи**:

1.Написати програму, в якій дані варіанту(таблиця 1) структури записуються в бінарне дерево(використати три поля для вузла –текстове дане та два числові. Наприклад, вузол дерева містить таку корисну інформацію: прізвище студента, рік народження, оцінка з іспиту). Ввести з клавіатури декілька "студентів" у двійкове дерево, організоване за порядком текстового поля. Роздрукувати отримане бінарне дерево.

2.Знайти середнє значення одного з числових полів, зчитуючи дані з дерева.

3. Надрукувати значення бінарного дерева:  
а) при прямому обході дерева;  
б) при зворотному обході дерева;  
в) при симетричному обході дерева.

4.Дописати функцію видалення з пам’яті всього бінарного дерева.

5.Створити рекурсивну функцію, яка:  
−видаляє ліве піддерево, і ліву гілку занулює;  
−видаляє праве піддерево, і праву гілку занулює;  
−видаляє сам вузол, потім зануливши вказівник на нього.

Наприкінці програми видалити з пам’яті дерево.

6."Пересипати" дані з першого дерева у друге дерево того ж типу, тільки організованого за першим числовим ключем (напр., рік народження) та роздрукувати його (а перше дерево стерти).

7.Розфарбувати вершини заданого бінарного дерева в червоний і чорний кольори так, щоб воно стало червоно-чорним деревом.

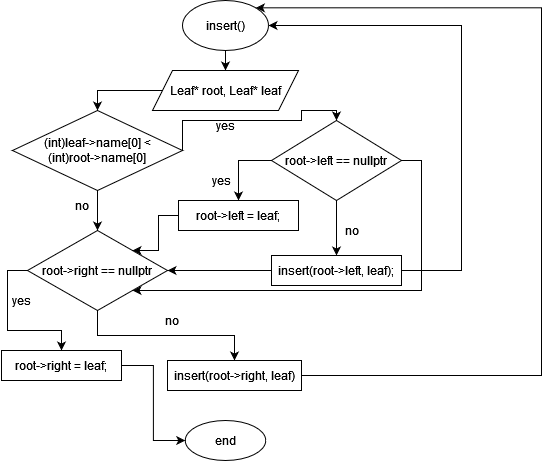
8. Провести перевірку властивості червоно-чорного дерева, а саме:  
−кожен вузол промаркований червоним або чорним кольором;  
−корінь дерева–чорний;  
−кінцеві вузли дерева –чорні;  
−біля червоного вузла батьківський вузол —чорний;  
−усі прості шляхи з будь-якого вузла до листя містять однакову кількість чорних вузлів.

9. У червоно-чорному дереві виконати додавання і видаленні вузлів та провести перевірку властивості червоно-чорного дерева.

Наприкінці програми видалити з пам’яті дерево.

# Бінарне дерево

Створення дерева (на основі додавання одного вузла):



Блок-схема 1.1 – Створення та/або додавання вузла до дерева

Код функції:

void CreateTree(BinaryTree\*& Root, string FullName, int IDpass, short int age) {

if (!Root) {

Root = new BinaryTree;

Root->FullName = FullName;

Root->IDpass = IDpass;

Root->age = age;

Root->LeftBranch = NULL;

Root->RightBranch = NULL;

return;

}

if (Root->age > age)

CreateTree(Root->LeftBranch, FullName, IDpass, age);

else

CreateTree(Root->RightBranch, FullName, IDpass, age);

}

Результат роботи алгоритму:

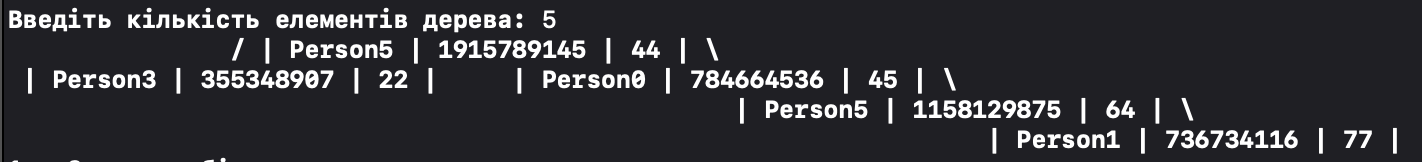
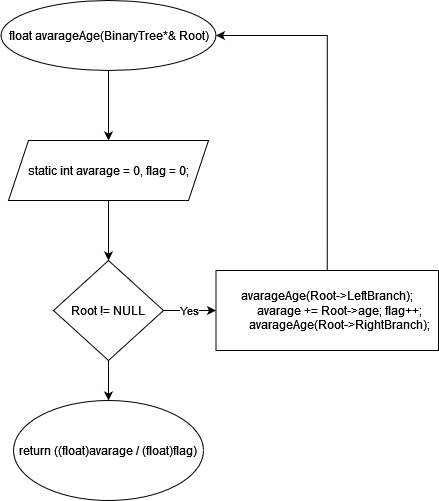


Рисунок 1.1 – Результат роботи алгоритму створення дерева

Пошук середнього арифметичного:



Блок-схема 1.2 – Алгоритм пошугу середнього арифметичного ( віку громадянина)

Код:

float avarageAge(BinaryTree\*& Root){

static int avarage = 0, flag = 0;

if (Root != NULL) {

avarageAge(Root->LeftBranch);

avarage += Root->age; flag++;

avarageAge(Root->RightBranch);

}

return ((float)avarage / (float)flag);

}

Результат роботи алгоритму:

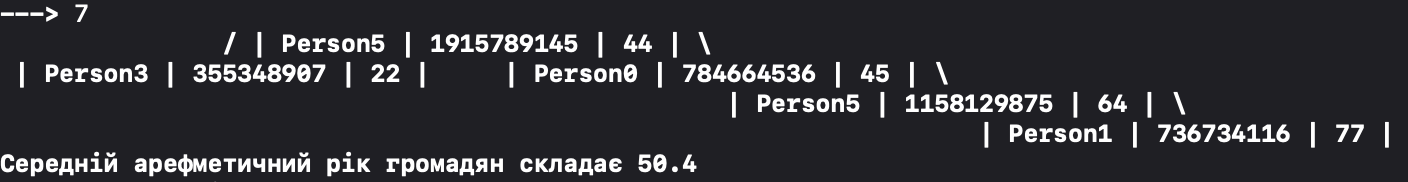
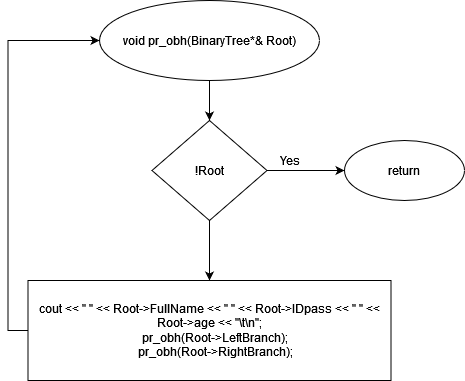


Рисунок 1.2 – Результат виконання алгоритму середнього арифметичного

Прямий обхід



Блок-схема 1.3 – Алгоритм прямого обходу дерева

Код:

void pr\_obh(BinaryTree\*& Root) {

if (!Root)return;

cout << " " << Root->FullName << " " << Root->IDpass << " " << Root->age << "\t\n";

pr\_obh(Root->LeftBranch);

pr\_obh(Root->RightBranch);

}

Результат роботи алгоритму:

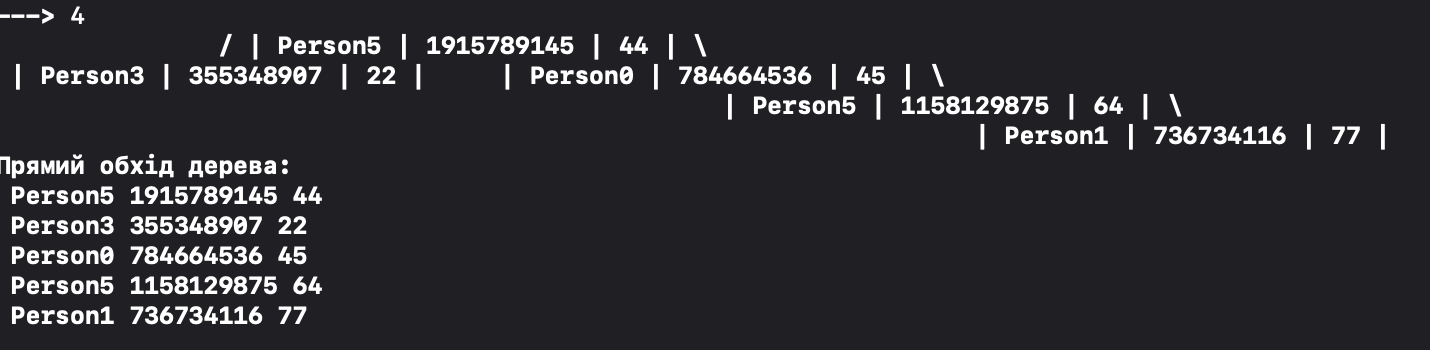
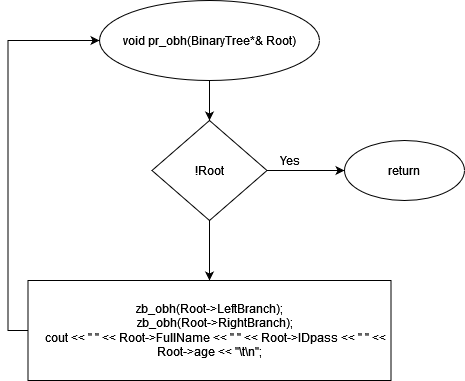


Рисунок 1.3 – Результат алгоритму прямого обходу дерева

Зворотній обхід



Блок-схема 1.4 – Алгоритм зворотного обходу дерева

Код:

void zb\_obh(BinaryTree\*& Root) {

if (!Root)return;

zb\_obh(Root->LeftBranch);

zb\_obh(Root->RightBranch);

cout << " " << Root->FullName << " " << Root->IDpass << " " << Root->age << "\t\n";

}

Результат роботи алгоритму:

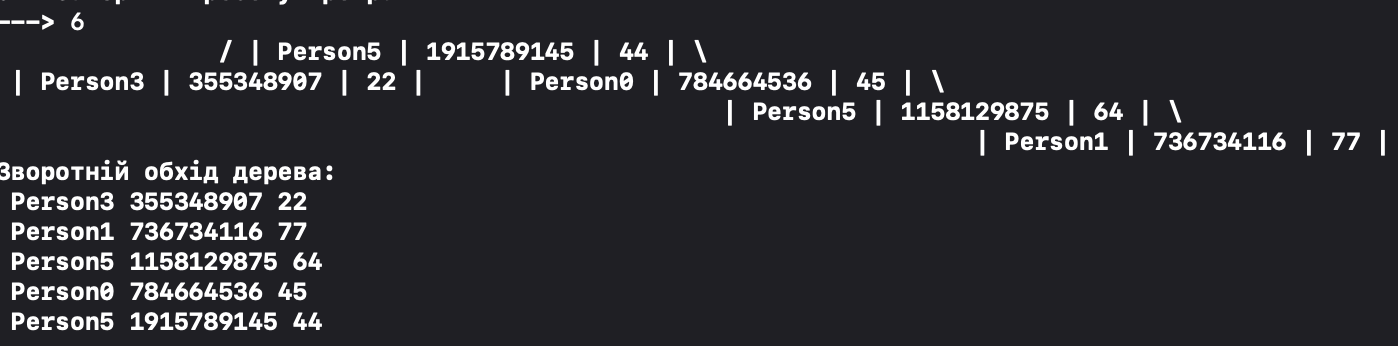
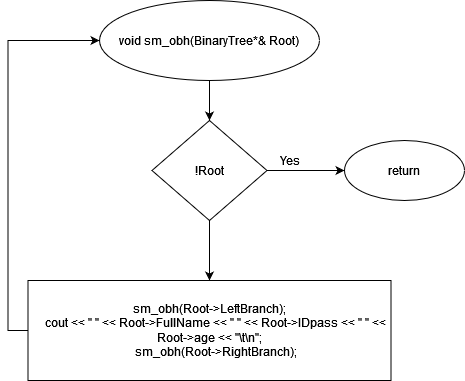


Рисунок 1.4 – Результат алгоритму зворотного обходу дерева

Симетричний обхід



Блок-схема 1.5 – Алгоритм семетричного обходу дерева

Код:

void sm\_obh(BinaryTree\*& Root) {

if (!Root)return;

sm\_obh(Root->LeftBranch);

cout << " " << Root->FullName << " " << Root->IDpass << " " << Root->age << "\t\n";

sm\_obh(Root->RightBranch);

}

Результат виконання роботи:

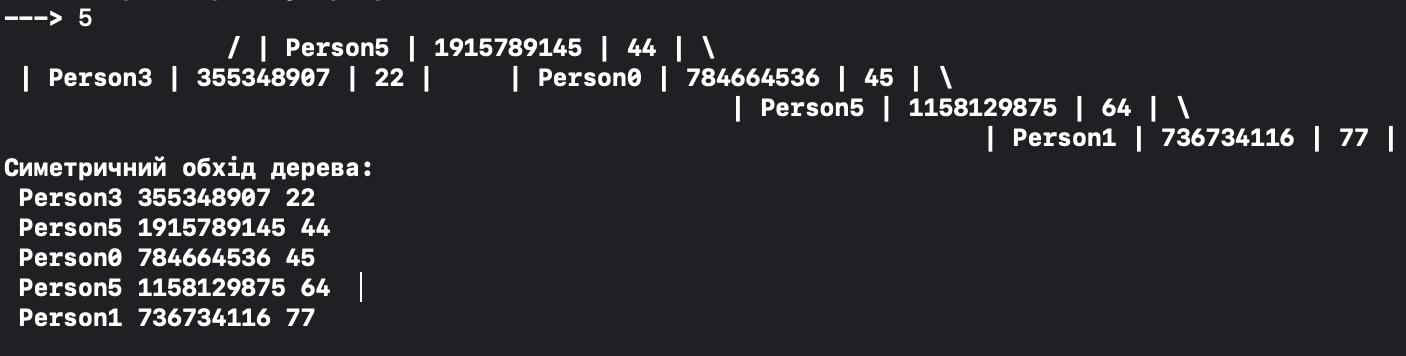
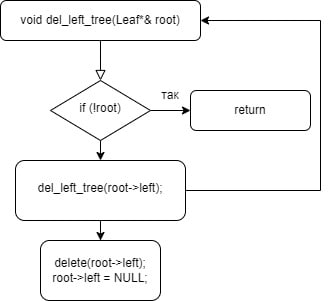


Рисунок 1.5 – Результат алгоритму семеричного обходу дерева

Видалення піддерева:



Блок-схема 1.6 – Алгоритм видалення піддерева

Код:

void del\_left\_tree(BinaryTree\*& Root) {

if (!Root)return;

del\_left\_tree(Root->LeftBranch);

delete(Root->LeftBranch);

Root->LeftBranch = NULL;

}

void del\_right\_tree(BinaryTree\*& Root) {

if (!Root)return;

del\_right\_tree(Root->RightBranch);

delete(Root->RightBranch);

Root->RightBranch = NULL;

}

Результат роботи алгоритму:

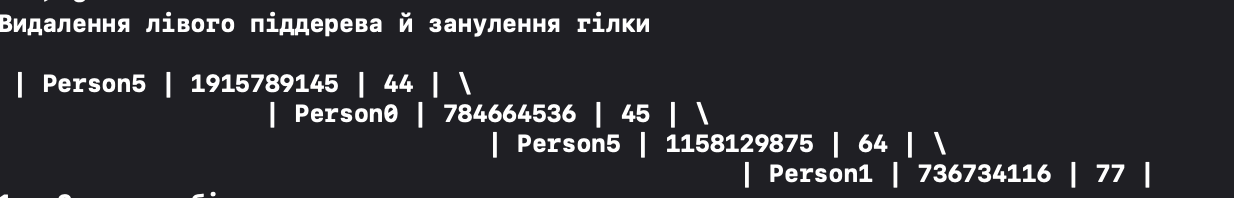
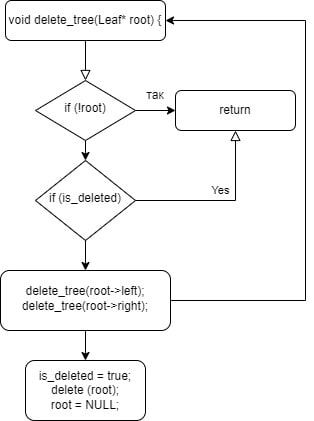


Рисунок 1.6 – Результат алгоритму видалення піддерева

Алгоритм очищення дерева з пам’яті



Блок-схема 1.7 – Алгоритм видалення дерева

Код:

void del(BinaryTree\*& Root) {

if (Root != NULL) {

del(Root->LeftBranch);

del(Root->RightBranch);

delete Root;

Root = NULL;

}

}

Результат роботи алгоритму:

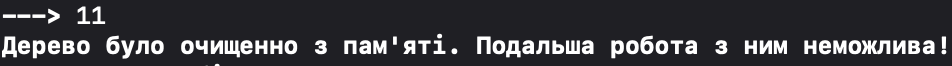
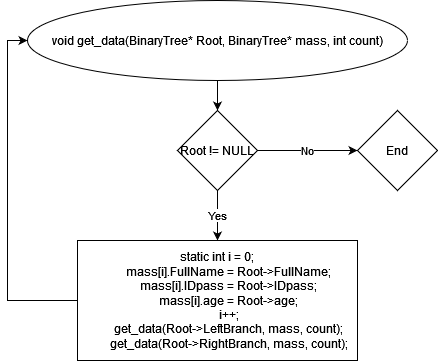


Рисунок 1.7 – Результат алгоритму видалення дерева

Алгоритм пересипання дерева за іншим ключем



Блок-схема 1.8 – Алгоритм запису в масив даних дерева

Після виконання цього алгоритму, данні масива передаються у функцію створення і додавання нових елементів дерева за ключем IDpass, за алгоритмом схожу, що продемонстровано на блок-схемі 1.1. Код алгоритму отримання та запису даних у масив приведено нижче, а також реалізація цього масива у бінарне дерево.

void get\_data(BinaryTree\* Root, BinaryTree\* mass, int count) {

if (Root != NULL) {

static int i = 0;

mass[i].FullName = Root->FullName;

mass[i].IDpass = Root->IDpass;

mass[i].age = Root->age;

i++;

get\_data(Root->LeftBranch, mass, count);

get\_data(Root->RightBranch, mass, count);

}

}

int TreeSize = size(Root);

BinaryTree\* NewTreeArray = new BinaryTree[TreeSize];

BinaryTree\* NewTree = NULL;

get\_data(Root, NewTreeArray, TreeSize);

for (size\_t i = 0; i < TreeSize; i++)

CreateNewTree(NewTree, NewTreeArray[i].FullName, NewTreeArray[i].IDpass, NewTreeArray[i].age);

Результат роботи алгоритму:



Рисунок 1.8 – Дерево до пересипання за новим ключем

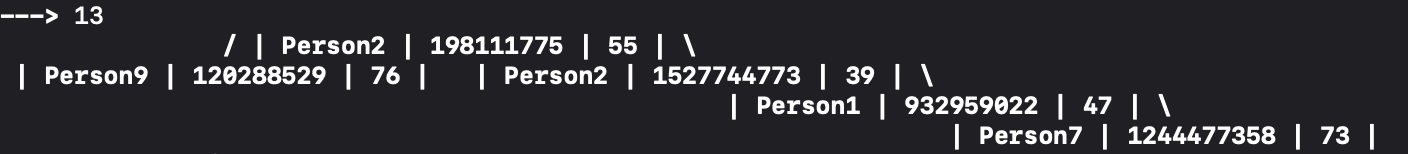
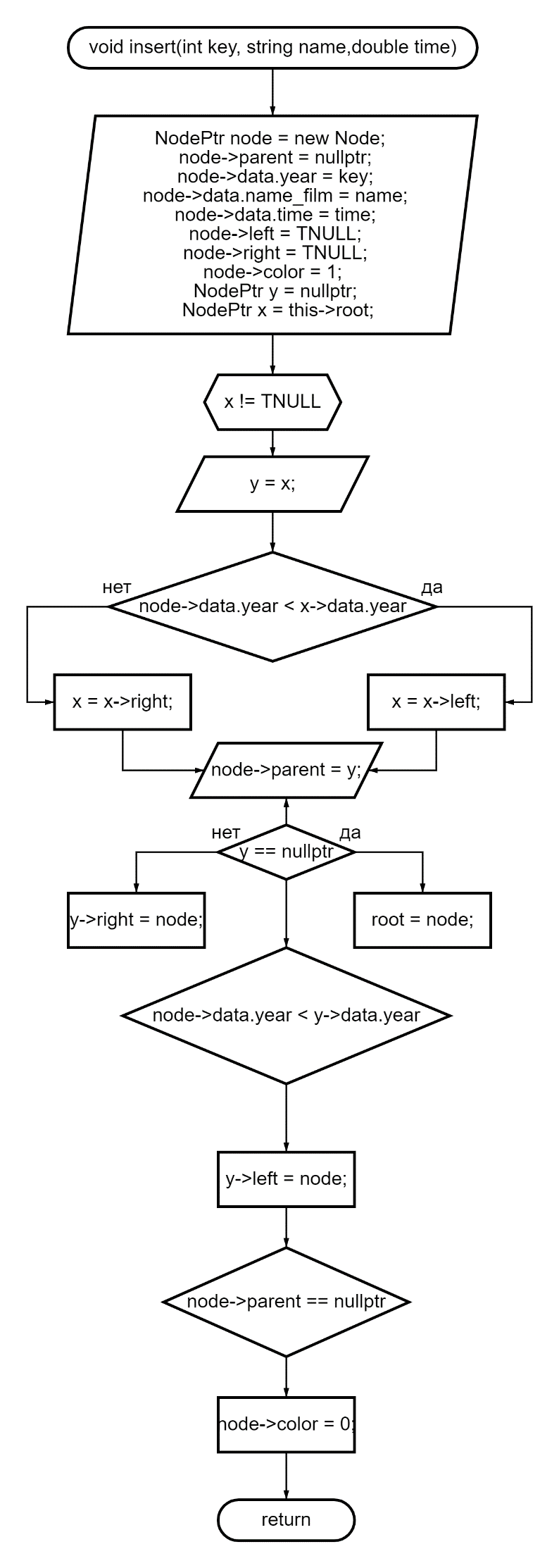


Рисунок 1.9 – Нове дерево за новим ключем IDpass

Повний код програми наведено у додатку Б та В.

# Червоно-чорне дерево

Додавання елемента (створення дерева-цикл з додавання)



Блок-схема 2.1 – Алгоритм додавання та/або створення червоно-чорного дерева

Код функції:

void fixInsert(NodePtr k) {

NodePtr u;

while (k->parent->color == 1) {

if (k->parent == k->parent->parent->right) {

u = k->parent->parent->left; // uncle

if (u->color == 1) {

// case 3.1

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->left) {

// case 3.2.2

k = k->parent;

rightRotate(k);

}

// case 3.2.1

k->parent->color = 0;

k->parent->parent->color = 1;

leftRotate(k->parent->parent);

}

}

else {

u = k->parent->parent->right; // uncle

if (u->color == 1) {

// mirror case 3.1

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->right) {

// mirror case 3.2.2

k = k->parent;

leftRotate(k);

}

// mirror case 3.2.1

k->parent->color = 0;

k->parent->parent->color = 1;

rightRotate(k->parent->parent);

}

}

if (k == root) {

break;

}

}

root->color = 0;

}

Результат роботи алгоритму:

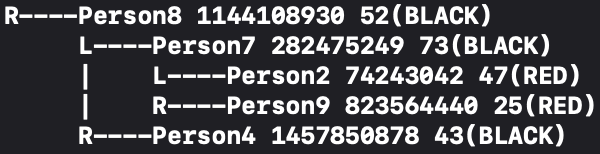


Рисунок 3.1 – Робота алгоритму створення дерева

Видалення елемента з дерева:

Алгоритм схожий до видалення в бінарному дереві, але потрібно пам’ятати про кольори.

Код функції:

void deleteNodeHelper(NodePtr node, int key) {

// find the node containing key

NodePtr z = TNULL;

NodePtr x, y;

while (node != TNULL) {

if (node->data.age == key) {

z = node;

}

if (node->data.age <= key) {

node = node->right;

}

else {

node = node->left;

}

}

if (z == TNULL) {

cout << "Couldn't find key in the tree" << endl;

return;

}

y = z;

int y\_original\_color = y->color;

if (z->left == TNULL) {

x = z->right;

rbTransplant(z, z->right);

}

else if (z->right == TNULL) {

x = z->left;

rbTransplant(z, z->left);

}

else {

y = minimum(z->right);

y\_original\_color = y->color;

x = y->right;

if (y->parent == z) {

x->parent = y;

}

else {

rbTransplant(y, y->right);

y->right = z->right;

y->right->parent = y;

}

rbTransplant(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

delete z;

if (y\_original\_color == 0) {

fixDelete(x);

}

}

Результат роботи алгоритму:

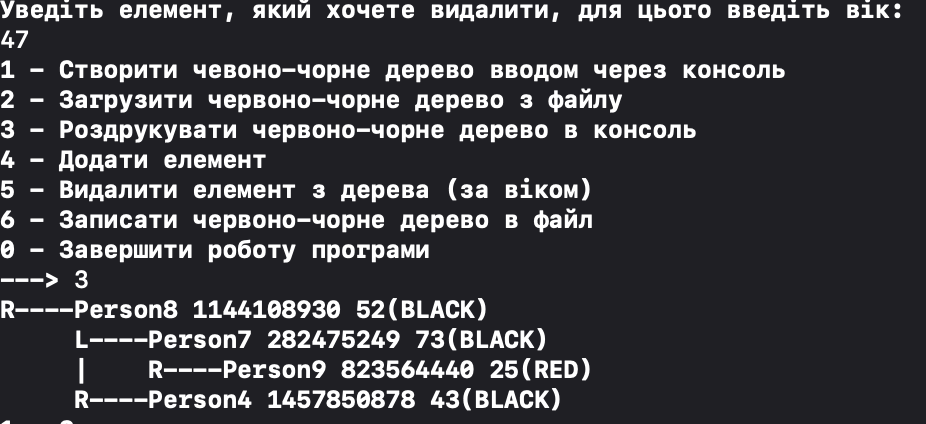


Рисунок 2.2 – Результат виконання алгоритму видалення елемента з дерева

# Висновок

Під час лабораторної роботи було засвоєно та використано на практиці знання х опрацювання динамічних структур даних такі як: бінарне дерево та червоно-чорне дерево.

Дерево – це структура даних, що є сукупністю елементів і відносин, що утворюють ієрархічну структуру цих елементів.

Піддерево – частина деревоподібної структури даних, яка може бути подана у вигляді окремого дерева.

Бінарне дерево – це динамічна структура даних, що являє собою дерево, в якому кожна вершина має не більше двох нащадків.

Упорядковане дерево – це дерево, у якого гілки, що виходять із кожної вершини, упорядковані за певним критерієм.

Прямий обхід(pre-order walk), у якому кожен вузол-предок проглядається раніше його нащадків

Зворотний обхід(post-order walk), коли спочатку проглядаються нащадки, потім предки.

Симетричний обхід, при якому обходиться спочатку ліве піддерево, потім вузол, потім праве піддерево.

Одним із головних властивостей бінарного дерева є те, що усі данні вже є відсортованими.

Червоно-чорне дерево – це один із видів самобалансуючих бінарних дерев пошуку, що гарантують логарифмічні зростання висоти дерева від числа вузлів і дозволяє швидко виконувати основні операції пошуку: додавання, видалення, та пошук вузла.

# Додаток А

#include <ctime>

#include <fstream>

#include <vector>

#include <functional>

#include <algorithm>

#include <tuple>

#include <stdexcept>

#include <memory>

using namespace std;

struct BinaryTree

{

string FullName;

int IDpass;

int age;

BinaryTree\* LeftBranch;

BinaryTree\* RightBranch;

};

int size(BinaryTree\*& Root);

void get\_data(BinaryTree\* Root, BinaryTree\* mass, int count);

void CreateNewTree(BinaryTree\*& Root, string FullName, int IDpass, short int age);

void Print(BinaryTree\*\* Root, int l = 0); // print test

bool Empty(BinaryTree\* Root);

void CreateTree(BinaryTree\*& Root, string FullName, int IDpass, short int age);

float avarageAge(BinaryTree\*& Root);

void del(BinaryTree\*& Root);

void pr\_obh(BinaryTree\*& Root);

void zb\_obh(BinaryTree\*& Root);

void sm\_obh(BinaryTree\*& Root);

void del\_left\_tree(BinaryTree\*& Root);

void del\_right\_tree(BinaryTree\*& Root);

void del\_branch(BinaryTree\*& Root, int age);

void write\_trea\_in\_file(BinaryTree\* Root);

void PrintTest(BinaryTree const\* node);

# Додаток Б

#include "Header.h"

static int count1 = 0;

int size(BinaryTree\*& Root) {

if (Root == NULL) return 0;

count1++;

size(Root->LeftBranch);

size(Root->RightBranch);

return count1;

}

void get\_data(BinaryTree\* Root, BinaryTree\* mass, int count) {

if (Root != NULL) {

static int i = 0;

mass[i].FullName = Root->FullName;

mass[i].IDpass = Root->IDpass;

mass[i].age = Root->age;

i++;

get\_data(Root->LeftBranch, mass, count);

get\_data(Root->RightBranch, mass, count);

}

}

bool Empty(BinaryTree\* Root) {

return (Root == NULL ? true : false);

}

void CreateTree(BinaryTree\*& Root, string FullName, int IDpass, short int age) {

if (!Root) {

Root = new BinaryTree;

Root->FullName = FullName;

Root->IDpass = IDpass;

Root->age = age;

Root->LeftBranch = NULL;

Root->RightBranch = NULL;

return;

}

if (Root->age > age)

CreateTree(Root->LeftBranch, FullName, IDpass, age);

else

CreateTree(Root->RightBranch, FullName, IDpass, age);

}

void CreateNewTree(BinaryTree\*& Root, string FullName, int IDpass, short int age) {

if (!Root) {

Root = new BinaryTree;

Root->FullName = FullName;

Root->IDpass = IDpass;

Root->age = age;

Root->LeftBranch = NULL;

Root->RightBranch = NULL;

return;

}

if (Root->IDpass > IDpass)

CreateTree(Root->LeftBranch, FullName, IDpass, age);

else

CreateTree(Root->RightBranch, FullName, IDpass, age);

}

float avarageAge(BinaryTree\*& Root){

static int avarage = 0, flag = 0;

if (Root != NULL) {

avarageAge(Root->LeftBranch);

avarage += Root->age; flag++;

avarageAge(Root->RightBranch);

}

return ((float)avarage / (float)flag);

}

void del(BinaryTree\*& Root) {

if (Root != NULL) {

del(Root->LeftBranch);

del(Root->RightBranch);

delete Root;

Root = NULL;

}

}

void pr\_obh(BinaryTree\*& Root) {

if (!Root)return;

cout << " " << Root->FullName << " " << Root->IDpass << " " << Root->age << "\t\n";

pr\_obh(Root->LeftBranch);

pr\_obh(Root->RightBranch);

}

void zb\_obh(BinaryTree\*& Root) {

if (!Root)return;

zb\_obh(Root->LeftBranch);

zb\_obh(Root->RightBranch);

cout << " " << Root->FullName << " " << Root->IDpass << " " << Root->age << "\t\n";

}

void sm\_obh(BinaryTree\*& Root) {

if (!Root)return;

sm\_obh(Root->LeftBranch);

cout << " " << Root->FullName << " " << Root->IDpass << " " << Root->age << "\t\n";

sm\_obh(Root->RightBranch);

}

void del\_left\_tree(BinaryTree\*& Root) {

if (!Root)return;

del\_left\_tree(Root->LeftBranch);

delete(Root->LeftBranch);

Root->LeftBranch = NULL;

}

void del\_right\_tree(BinaryTree\*& Root) {

if (!Root)return;

del\_right\_tree(Root->RightBranch);

delete(Root->RightBranch);

Root->RightBranch = NULL;

}

void del\_branch(BinaryTree\*& Root, int age) {

if (Root != NULL) {

if (Root->age == age) {

BinaryTree\* ptr = Root;

if (Root->LeftBranch == NULL && Root->RightBranch == NULL)

Root = NULL;

else if (Root->LeftBranch == NULL)

Root = ptr->RightBranch;

else if (Root->RightBranch == NULL)

Root = ptr->LeftBranch;

else {

Root = ptr->RightBranch;

BinaryTree\* ptr1;

ptr1 = Root;

while (ptr1 != NULL) {

ptr1 = ptr1->LeftBranch;

ptr1 = ptr->LeftBranch;

}

delete(ptr);

del\_branch(Root, age);

}

}

else {

del\_branch(Root->LeftBranch, age);

del\_branch(Root->RightBranch, age);

}

}

}

void Print(BinaryTree\*\* Root, int l) { // test print

int i;

if (\*Root != NULL) {

Print(&((\*\*Root).RightBranch), l + 1);

for (i = 1; i <= l; i++) cout << " "; {

cout << (\*\*Root).FullName << " " << (\*\*Root).IDpass << " " << (\*\*Root).age << endl;

}

Print(&((\*\*Root).LeftBranch), l + 1);

}

}

int tabs = 0;

void write\_trea\_in\_file(BinaryTree\* Root) {

ofstream fout;

fout.open("data.txt", fstream::app);

if (!Root) return;

tabs += 5;

write\_trea\_in\_file(Root->RightBranch);

for (int i = 0; i < tabs; i++) fout << " ";

fout << Root->FullName << endl;

for (int i = 0; i < tabs; i++) fout << " ";

fout << Root->IDpass << endl;

for (int i = 0; i < tabs; i++) fout << " ";

fout << Root->age << endl;

write\_trea\_in\_file(Root->LeftBranch);

tabs -= 5;

return;

fout.close();

}

static std::string ch\_hor = "-", ch\_ver = "|", ch\_ddia = "/", ch\_rddia = "\\", ch\_udia = "\\", ch\_ver\_hor = "|-", ch\_udia\_hor = "\\-", ch\_ddia\_hor = "/-", ch\_ver\_spa = "| ";

void PrintTest(BinaryTree const\* node) {

#define \_MAX(x, y) ((x) > (y) ? (x) : (y))

#define \_MIN(x, y) ((x) < (y) ? (x) : (y))

auto RepStr = [](std::string const& s, size\_t cnt) {

if (ptrdiff\_t(cnt) < 0)

throw std::runtime\_error("RepStr: Bad value " + std::to\_string(ptrdiff\_t(cnt)) + "!");

std::string r;

for (size\_t i = 0; i < cnt; ++i)

r += s;

return r;

};

std::function<std::tuple<std::vector<std::string>, size\_t, size\_t>(BinaryTree const\* node, bool)> Rec;

Rec = [&RepStr, &Rec](BinaryTree const\* node, bool left) {

std::vector<std::string> lines;

if (!node)

return std::make\_tuple(lines, size\_t(0), size\_t(0));

auto sval = " | " + node->FullName + " | " + std::to\_string(node->IDpass) + " | " + std::to\_string(node->age) + " | ";

//if (sval.size() % 2 != 1) sval += " ";

auto resl = Rec(node->LeftBranch, true), resr = Rec(node->RightBranch, false);

auto const& vl = std::get<0>(resl);

auto const& vr = std::get<0>(resr);

auto cl = std::get<1>(resl), cr = std::get<1>(resr), lss = std::get<2>(resl), rss = std::get<2>(resr);

size\_t lv = sval.size();

size\_t ls = vl.size() > 0 ? lss : size\_t(0);

size\_t rs = vr.size() > 0 ? rss : size\_t(0);

size\_t lis = ls == 0 ? lv / 2 : \_MAX(int(lv / 2 + 1 - (ls - cl)), 0);

size\_t ris = rs == 0 ? (lv + 1) / 2 : \_MAX(int((lv + 1) / 2 - cr), (lis > 0 ? 0 : 1));

size\_t dashls = (ls == 0 ? 0 : ls - cl - 1 + lis - lv / 2), dashrs = (rs == 0 ? 0 : cr + ris - (lv + 1) / 2);

//DEB(node->value); DEB(lv); DEB(ls); DEB(rs); DEB(cl); DEB(cr); DEB(lis); DEB(ris); DEB(dashls); DEB(dashrs); std::cout << std::endl;

lines.push\_back(

(ls == 0 ? "" : (RepStr(" ", cl) + ch\_ddia + RepStr(ch\_hor, dashls))) +

sval + (rs == 0 ? "" : (RepStr(ch\_hor, dashrs) + ch\_rddia + RepStr(" ", rs - cr - 1)))

);

//if (lines.back().size() != ls + lis + ris + rs)

// throw std::runtime\_error("Dump: First line wrong size, got " + std::to\_string(lines.back().size()) + ", expected " + std::to\_string(ls + lis + ris + rs));

for (size\_t i = 0; i < \_MAX(vl.size(), vr.size()); ++i) {

std::string sl = RepStr(" ", ls), sr = RepStr(" ", rs);

if (i < vl.size()) sl = vl[i];

if (i < vr.size()) sr = vr[i];

sl = sl + RepStr(" ", lis);

sr = RepStr(" ", ris) + sr;

lines.push\_back(sl + sr);

}

return std::make\_tuple(lines, (left || ls + lis == 0 || lv % 2 == 1) ? ls + lis : ls + lis - 1, ls + lis + ris + rs);

};

auto v = std::get<0>(Rec(node, true));

for (size\_t i = 0; i < v.size(); ++i)

std::cout << v[i] << std::endl;

#undef \_MAX

#undef \_MIN

}

# Додаток В

#include "Header.h"

int main()

{

system("chcp 1251");

srand(time(NULL));

BinaryTree\* Root = NULL;

string FullName = "Person"; int IDpass; short int age;

int button; string fileRead;

ifstream dataFile;

while (true){

cout << "1 - Створити бінарне дерево вводом через консоль\n"

<< "2 - Загрузити бінарне дерево з файлу\n"

<< "3 - Роздрукувати бінарне дерево в консоль\n"

<< "4 - Прямий обхід дерева\n"

<< "5 - Симетричний обхід дерева\n"

<< "6 - Зворотній обхід дерева\n"

<< "7 - Середнє арифметичне бінарного дерева (середній вік громадян)\n"

<< "8 - Видалення лівого піддерева й занулення гілки\n"

<< "9 - Видалення правого піддерева й занулення гілки\n"

<< "10 - Видалити вузол й занулити його вказівник\n"

<< "11 - Видалити все дерево\n"

<< "12 - Записати бінарне дерево в файл\n"

<< "13 - Пересипати у нове бінарне дерево (за ID паспорта)\n"

<< "0 - Завершити роботу програми\n"

<< "---> ";

cin >> button;

switch (button) {

case 1:

cout << "Введіть кількість елементів дерева: ";

cin >> button;

for (size\_t i = 0; i < button; i++) {

FullName += to\_string(rand() % 10);

IDpass = rand();

age = rand() % 60 + 20;

CreateTree(Root, FullName, IDpass, age);

FullName = "Person";

}

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

case 2: {

int count = 0;

dataFile.open("data.txt");

while (!dataFile.eof()) {

getline(dataFile, fileRead);

count++;

}

count = count / 3;

dataFile.close();

BinaryTree\* data = new BinaryTree[count];

dataFile.open("data.txt");

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

dataFile >> FullName;

dataFile >> IDpass;

dataFile >> age;

CreateTree(Root, FullName, IDpass, age);

FullName = "Person";

}

dataFile.close();

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

}

case 3:

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

case 4:

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

cout << "Прямий обхід дерева: " << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

pr\_obh(Root);

cout << endl;

break;

case 5:

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

cout << "Симетричний обхід дерева: " << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

sm\_obh(Root);

cout << endl;

break;

case 6:

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

cout << "Зворотній обхід дерева: " << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

zb\_obh(Root);

cout << endl;

break;

case 7:

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

if (!Empty)

cout << "Дерево не знайдено!\n";

else

cout << "Середній арефметичний рік громадян складає " << avarageAge(Root) << endl;

break;

case 8:

cout << "Видалення лівого піддерева й занулення гілки" << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

del\_left\_tree(Root);

cout << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

case 9:

cout << "Видалення правого піддерева й занулення гілки" << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

del\_right\_tree(Root);

cout << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

case 10:

cout << "Введіть елемент, який хочете видалить: ";

cin >> age;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

del\_branch(Root, age);

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

case 11:

cout << "Дерево було очищенно з пам'яті. Подальша робота з ним неможлива!" << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

del(Root);

if (!Empty)

cout << "Дерево не знайдено!\n";

else

PrintTest(Root);

break;

case 12:

cout << "Запис дерева у файл: " << endl;

if (!Empty)

cout << "Дерево не знайдено!\n";

else

write\_trea\_in\_file(Root);

break;

case 13:

if (!Empty)

cout << "Дерево не знайдено!\n";

else {

int TreeSize = size(Root);

BinaryTree\* NewTreeArray = new BinaryTree[TreeSize];

BinaryTree\* NewTree = NULL;

get\_data(Root, NewTreeArray, TreeSize);

for (size\_t i = 0; i < TreeSize; i++){

CreateNewTree(NewTree, NewTreeArray[i].FullName, NewTreeArray[i].IDpass, NewTreeArray[i].age);

}

PrintTest(NewTree);

}

break;

default: {

break;

}

}

}

}

# Додаток Д

#include <iostream>

#include<iomanip>

#include <string>

#include<Windows.h>

#include<fstream>

using namespace std;

struct Data {

string FullName;

int IDpass;

int age;

};

// data structure that represents a node in the tree

struct Node {

Data data; // holds the key

Node\* parent; // pointer to the parent

Node\* left; // pointer to left child

Node\* right; // pointer to right child

int color; // 1 -> Red, 0 -> Black

};

typedef Node\* NodePtr;

// class RBTree implements the operations in Red Black Tree

class RBTree {

private:

NodePtr root;//корінці

NodePtr TNULL;//листки

// initializes the nodes with appropirate values

// all the pointers are set to point to the null pointer

void initializeNULLNode(NodePtr node, NodePtr parent) {

node->data.IDpass = 0;

node->parent = parent;

node->left = nullptr;

node->right = nullptr;

node->color = 0;

}

void preOrderHelper(NodePtr node) {

if (node != TNULL) {

cout << node->data.FullName << " ";

cout << node->data.IDpass << " ";

cout << node->data.age << " ";

preOrderHelper(node->left);

preOrderHelper(node->right);

}

}

void inOrderHelper(NodePtr node) {

if (node != TNULL) {

inOrderHelper(node->left);

cout << node->data.FullName << " ";

cout << node->data.IDpass << " ";

cout << node->data.age << " ";

inOrderHelper(node->right);

}

}

void postOrderHelper(NodePtr node) {

if (node != TNULL) {

postOrderHelper(node->left);

postOrderHelper(node->right);

cout << node->data.FullName << " ";

cout << node->data.IDpass << " ";

cout << node->data.age << " ";

}

}

NodePtr searchTreeHelper(NodePtr node, int key) {

if (node == TNULL || key == node->data.IDpass) {

return node;

}

if (key < node->data.IDpass) {

return searchTreeHelper(node->left, key);

}

return searchTreeHelper(node->right, key);

}

// fix the rb tree modified by the delete operation

void fixDelete(NodePtr x) {

NodePtr s;

while (x != root && x->color == 0) {

if (x == x->parent->left) {

s = x->parent->right;

if (s->color == 1) {

// case 3.1

s->color = 0;

x->parent->color = 1;

leftRotate(x->parent);

s = x->parent->right;

}

if (s->left->color == 0 && s->right->color == 0) {

// case 3.2

s->color = 1;

x = x->parent;

}

else {

if (s->right->color == 0) {

// case 3.3

s->left->color = 0;

s->color = 1;

rightRotate(s);

s = x->parent->right;

}

// case 3.4

s->color = x->parent->color;

x->parent->color = 0;

s->right->color = 0;

leftRotate(x->parent);

x = root;

}

}

else {

s = x->parent->left;

if (s->color == 1) {

// case 3.1

s->color = 0;

x->parent->color = 1;

rightRotate(x->parent);

s = x->parent->left;

}

if (s->left->color == 0 && s->right->color == 0) {

// case 3.2

s->color = 1;

x = x->parent;

}

else {

if (s->left->color == 0) {

// case 3.3

s->right->color = 0;

s->color = 1;

leftRotate(s);

s = x->parent->left;

}

// case 3.4

s->color = x->parent->color;

x->parent->color = 0;

s->left->color = 0;

rightRotate(x->parent);

x = root;

}

}

}

x->color = 0;

}

void rbTransplant(NodePtr u, NodePtr v) {

if (u->parent == nullptr) {

root = v;

}

else if (u == u->parent->left) {

u->parent->left = v;

}

else {

u->parent->right = v;

}

v->parent = u->parent;

}

void deleteNodeHelper(NodePtr node, int key) {

// find the node containing key

NodePtr z = TNULL;

NodePtr x, y;

while (node != TNULL) {

if (node->data.age == key) {

z = node;

}

if (node->data.age <= key) {

node = node->right;

}

else {

node = node->left;

}

}

if (z == TNULL) {

cout << "Couldn't find key in the tree" << endl;

return;

}

y = z;

int y\_original\_color = y->color;

if (z->left == TNULL) {

x = z->right;

rbTransplant(z, z->right);

}

else if (z->right == TNULL) {

x = z->left;

rbTransplant(z, z->left);

}

else {

y = minimum(z->right);

y\_original\_color = y->color;

x = y->right;

if (y->parent == z) {

x->parent = y;

}

else {

rbTransplant(y, y->right);

y->right = z->right;

y->right->parent = y;

}

rbTransplant(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

delete z;

if (y\_original\_color == 0) {

fixDelete(x);

}

}

// fix the red-black tree

void fixInsert(NodePtr k) {

NodePtr u;

while (k->parent->color == 1) {

if (k->parent == k->parent->parent->right) {

u = k->parent->parent->left; // uncle

if (u->color == 1) {

// case 3.1

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->left) {

// case 3.2.2

k = k->parent;

rightRotate(k);

}

// case 3.2.1

k->parent->color = 0;

k->parent->parent->color = 1;

leftRotate(k->parent->parent);

}

}

else {

u = k->parent->parent->right; // uncle

if (u->color == 1) {

// mirror case 3.1

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->right) {

// mirror case 3.2.2

k = k->parent;

leftRotate(k);

}

// mirror case 3.2.1

k->parent->color = 0;

k->parent->parent->color = 1;

rightRotate(k->parent->parent);

}

}

if (k == root) {

break;

}

}

root->color = 0;

}

void printHelper(NodePtr root, string indent, bool last) {

// print the tree structure on the screen

if (root != TNULL) {

cout << indent;

if (last) {

cout << "R----";

indent += " ";

}

else {

cout << "L----";

indent += "| ";

}

string sColor = (root->color) == 1 ? "RED" : "BLACK";

cout << root->data.FullName << " " << root->data.IDpass << " " << root->data.age << "(" << sColor << ")" << endl;

printHelper(root->left, indent, false);

printHelper(root->right, indent, true);

}

// cout<<root->left->data<<endl;

}

void printHelper\_write\_file(NodePtr root, string indent, bool last) {

ofstream fout;

fout.open("data.txt", ios\_base::app);

if (root != TNULL) {

fout << indent;

if (last) {

//fout << "R----";

//indent += " ";

}

else {

//fout << "L----";

//indent += "| ";

}

//string sColor = (root->color) == 1 ? "RED" : "BLACK";

//fout << root->data.FullName << " " << root->data.IDpass << " " << root->data.age << "(" << sColor << ")" << endl;

fout << root->data.FullName << " " << root->data.IDpass << " " << root->data.age << endl;

printHelper\_write\_file(root->left, indent, false);

printHelper\_write\_file(root->right, indent, true);

}

fout.close();

}

public:

RBTree() {

TNULL = new Node;

TNULL->color = 0;

TNULL->left = nullptr;

TNULL->right = nullptr;

root = TNULL;

}

// Pre-Order traversal

// Node->Left Subtree->Right Subtree

void preorder() {

preOrderHelper(this->root);

}

// In-Order traversal

// Left Subtree -> Node -> Right Subtree

void inorder() {

inOrderHelper(this->root);

}

// Post-Order traversal

// Left Subtree -> Right Subtree -> Node

void postorder() {

postOrderHelper(this->root);

}

// search the tree for the key k

// and return the corresponding node

NodePtr searchTree(int k) {

return searchTreeHelper(this->root, k);

}

// find the node with the minimum key

NodePtr minimum(NodePtr node) {

while (node->left != TNULL) {

node = node->left;

}

return node;

}

// find the node with the maximum key

NodePtr maximum(NodePtr node) {

while (node->right != TNULL) {

node = node->right;

}

return node;

}

// find the successor of a given node

NodePtr successor(NodePtr x) {

// if the right subtree is not null,

// the successor is the leftmost node in the

// right subtree

if (x->right != TNULL) {

return minimum(x->right);

}

// else it is the lowest ancestor of x whose

// left child is also an ancestor of x.

NodePtr y = x->parent;

while (y != TNULL && x == y->right) {

x = y;

y = y->parent;

}

return y;

}

// find the predecessor of a given node

NodePtr predecessor(NodePtr x) {

// if the left subtree is not null,

// the predecessor is the rightmost node in the

// left subtree

if (x->left != TNULL) {

return maximum(x->left);

}

NodePtr y = x->parent;

while (y != TNULL && x == y->left) {

x = y;

y = y->parent;

}

return y;

}

// rotate left at node x

void leftRotate(NodePtr x) {

NodePtr y = x->right;

x->right = y->left;

if (y->left != TNULL) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

}

else if (x == x->parent->left) {

x->parent->left = y;

}

else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

// rotate right at node x

void rightRotate(NodePtr x) {

NodePtr y = x->left;

x->left = y->right;

if (y->right != TNULL) {

y->right->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

}

else if (x == x->parent->right) {

x->parent->right = y;

}

else {

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

// insert the key to the tree in its appropriate position

// and fix the tree

void insert(int key, string name, double age) {

// Ordinary Binary Search Insertion

NodePtr node = new Node;

node->parent = nullptr;

node->data.IDpass = key;

node->data.FullName = name;

node->data.age = age;

node->left = TNULL;

node->right = TNULL;

node->color = 1; // new node must be red

NodePtr y = nullptr;

NodePtr x = this->root;

while (x != TNULL) {

y = x;

if (node->data.IDpass < x->data.IDpass) {

x = x->left;

}

else {

x = x->right;

}

}

// y is parent of x

node->parent = y;

if (y == nullptr) {

root = node;

}

else if (node->data.IDpass < y->data.IDpass) {

y->left = node;

}

else {

y->right = node;

}

// if new node is a root node, simply return

if (node->parent == nullptr) {

node->color = 0;

return;

}

// if the grandparent is null, simply return

if (node->parent->parent == nullptr) {

return;

}

// Fix the tree

fixInsert(node);

}

NodePtr getRoot() {

return this->root;

}

// delete the node from the tree

void deleteNode(int data) {

deleteNodeHelper(this->root, data);

}

// print the tree structure on the screen

void prettyPrint() {

if (root) {

printHelper(this->root, "", true);

}

}

void wtite\_trea\_in\_file() {

if (root) {

printHelper\_write\_file(this->root, "", true);

}

}

void del(NodePtr node) {

if (node != NULL) {

del(node->left);

del(node->right);

delete node;

node = NULL;

}

}

};

int main() {

RBTree bst;

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

ifstream fin;

string FullName = "Person"; int IDpass, age;

int number;

do {

cout << "1 - Створити чевоно-чорне дерево вводом через консоль" << endl;

cout << "2 - Загрузити червоно-чорне дерево з файлу" << endl;

cout << "3 - Роздрукувати червоно-чорне дерево в консоль" << endl;

cout << "4 - Додати елемент" << endl;

cout << "5 - Видалити елемент з дерева (за віком)" << endl;

cout << "6 - Записати червоно-чорне дерево в файл" << endl;

cout << "0 - Завершити роботу програми" << endl;

cout << "---> ";

cin >> number;

switch (number)

{

case 1: {

int c;

cout << "Введіть кількість елементів дерева: ";

cin >> c;

/\*int c = rand()%(max-min)+min;\*/

Data l;

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

FullName += to\_string(rand() % 10);

IDpass = rand();

age = rand() % 60 + 20;

l.FullName = FullName; l.IDpass = IDpass; l.age = age;

bst.insert(l.IDpass, l.FullName, l.age);

FullName = "Person";

}

break;}

case 2: {

int count = 0;

string s;

fin.open("data.txt");

while (!fin.eof()) {

fin >> s;

count++;

}

//cout << count;

count = count / 3;

fin.close();

Data\* data = new Data[count];

fin.open("data.txt");

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

FullName = " ";

IDpass = 0;

age = 0;

fin >> FullName;

fin >> IDpass;

fin >> age;

data[i].FullName = FullName;

data[i].IDpass = IDpass;

data[i].age = age;

}

fin.close();

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

bst.insert(data[i].IDpass, data[i].FullName, data[i].age);

}

break; }

case 3: {

bst.prettyPrint();

break;}

case 4: {

Data l;

cout << "Введіть ім'я: ";

cin >> l.FullName;

cout << "Введіть ID паспорту: ";

cin >> l.IDpass;

cout << "Введіть вік: ";

cin >> l.age;

bst.insert(l.IDpass, l.FullName, l.age);

break; }

case 5: {

cout << "Уведіть елемент, який хочете видалити, для цього введіть вік: " << endl;

cin >> age;

bst.deleteNode(age);

break; }

case 6: {bst.wtite\_trea\_in\_file();

break; }

default:

cout << "Введіть коректне значення" << endl;

break;

}

} while (number != 0);

system("cls");

return 0;

}