МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Кафедра інформаційних технологій

**ЗВІТ**

**ЛАБОРАТОРНА РОБОТА № 10**

«[Алгоритми пошуку на бінарних деревах](http://cyber.onua.edu.ua/mod/page/view.php?id=2613)»

Виконала: студентка 2 курсу, 212 групи

Ярош Євгенія В’ячеславівна

Перевірив: доцент. к.т.н., Трофименко О. Г.

Одеса - 2022 р

**Завдання на лабораторну роботу:**

1. Створити шаблон класу бінарного дерева пошуку з такими операціями.
   1. долучення (вставлення) елемента в дерево;
   2. видалення елемента з дерева;
   3. пошук елемента в дереві. Створити два варіанти функції: перша функція повертає значення true, якщо елемент є в дереві, false – якщо його немає; друга функція повертає вказівник на елемент, якщо він є в дереві, і nullptr якщо його там немає.
   4. пошук мінімального і максимального елементів дерева;
   5. визначення кількості елементів у дереві;
   6. прямий, зворотний, центрований обхід дерева;
   7. обхід дерева в ширину;
   8. визначення висоти дерева;
   9. пошук батька елемента;
   10. формування лінійного дужкового запису дерева;
   11. видалення дерева.
2. На основі класу бінарного дерева реалізувати клас збалансованого дерева. Для балансування використовувати додатковий відсортований масив і бінарний пошук.
3. На основі класу бінарного дерева реалізувати клас збалансованого дерева. Для балансування використовувати алгоритм DSW.

**Програмний код:**

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#include <Windows.h>

using namespace std;

template <typename T>

class Tree {

public:

Tree() {

root = nullptr;

}

struct Node {

Node() {}

T data;

Node\* l = nullptr;

Node\* r = nullptr;

Node(T value) { data = value; }

};

void Add\_Node(T value) {

add\_node(value, root);

}

void PrintTree() {

int q = 30;

print\_tree(root, q);

}

void printTreeLNR() {

printTreeLNR\_(root);

}

void printTreeNLR() {

printTreeNLR\_(root);

}

void printTreeLRN() {

printTreeLRN\_(root);

}

void Insert(T value) {

root = doInsert(root, value);

size++;

vector\_.push\_back(value);

}

void Destroy(T value) {

Delete(root, value);

size--;

}

Node\* Find(T value) {

return search(root, value);

}

bool IsContain(T value) {

return SearchBool(root, value);

}

T Min() {

return FindMin(root);

}

T Max() {

return FindMax(root);

}

int Height() {

return getHeight(root);

}

void FindParent(T value) {

findParent\_(root, value, -1);

};

int TreeElemCounter() {

return TreeCounter(root);

}

void treeToString()

{

string str = "";

treeToString\_(root, str);

cout << str;

}

void BalancedTree() {

sort(vector\_.begin(), vector\_.end());

root = Convert(vector\_);

PrintTree();

}

void BalancedDSWTree() {

root = BalanceDSW(root);

PrintTree();

}

void FreeTheTree() {

FreeTree(root);

size = 0;

}

private:

Node\* root;

int size = 0;

vector<T> vector\_;

Node\* doInsert(Node\*& node\_, T value) {

if (node\_ == nullptr) {

return new Node(value);

}

if (value < node\_->data) {

node\_->l = doInsert(node\_->l, value);

}

else if (value > node\_->data) {

node\_->r = doInsert(node\_->r, value);

}

return node\_;

}

void Delete(Node\*& node\_, T value)

{

if (value < node\_->data)

Delete(node\_->l, value);

else

if (value > node\_->data)

Delete(node\_->r, value);

else

DeleteNode(node\_);

}

void getPredecessor(Node\* node\_, T& value)

{

while (node\_->r != nullptr)

node\_ = node\_->r;

value = node\_->data;

}

void DeleteNode(Node\*& node\_)

{

int key;

Node\* ptr = node\_;

if (node\_->l == NULL) {

node\_ = node\_->r;

delete ptr;

}

else if (node\_->r == NULL) {

node\_ = node\_->l;

delete ptr;

}

else if (node\_->r == NULL && node\_->l == NULL) {

}

else {

getPredecessor(node\_->l, key);

node\_->data = key;

Delete(node\_->l, key);

}

}

Node\* search(Node\* node\_, T value)

{

while (node\_ != nullptr)

{

if (value == node\_->data) return node\_;

else

if (value < node\_->data) node\_ = node\_->l;

else node\_ = node\_->r;

}

return nullptr;

}

bool SearchBool(Node\* node\_, T value)

{

while (node\_ != nullptr)

{

if (value == node\_->data) return true;

else

if (value < node\_->data) node\_ = node\_->l;

else node\_ = node\_->r;

}

return false;

}

T FindMax(Node\* node\_)

{

T max = node\_->data;

while (node\_ != nullptr)

{

max = node\_->data;

node\_ = node\_->r;

}

return max;

}

T FindMin(Node\* node\_){

T min = node\_->data;

while (node\_ != nullptr)

{

min = node\_->data;

node\_ = node\_->l;

}

return min;

}

int getHeight(Node\* node\_) {

if (node\_ == nullptr) {

return 0;

}

else {

int lheight = getHeight(node\_->l);

int rheight = getHeight(node\_->r);

if (lheight > rheight) {

return (lheight + 1);

}

else {

return (rheight + 1);

}

}

}

T TreeCounter(Node\* node\_) {

int c = 1;

if (node\_ == NULL)

return 0;

else

{

c += TreeCounter(node\_->l);

c += TreeCounter(node\_->r);

return c;

}

};

void print\_tree(Node\* node\_, int& q)

{

if (node\_ != NULL)

{

q++;

print\_tree(node\_->l, q);

for (int i = 1; i <= q; i++) cout << " ";

cout << node\_->data << endl;

print\_tree(node\_->r, q);

q--;

};

};

void findParent\_(Node\* node\_, T value, T parent)

{

if (node\_ == NULL) return;

if (node\_->data == value) {

if (parent == -1) { cout << "Немає батька"; }

else { cout << "Батько: " << parent; };

}

else {

findParent\_(node\_->l, value, node\_->data);

findParent\_(node\_->r, value, node\_->data);

}

}

void printTreeLNR\_(Node\* node\_) {

if (node\_ != NULL) {

printTreeLNR\_(node\_->l);

cout << node\_->data << " ";

printTreeLNR\_(node\_->r);

}

}

void printTreeNLR\_(Node\* node\_) {

if (node\_ != NULL) {

cout << node\_->data << " ";

printTreeNLR\_(node\_->l);

printTreeNLR\_(node\_->r);

}

}

void printTreeLRN\_(Node\* node\_) {

if (node\_ != NULL) {

printTreeLRN\_(node\_->l);

printTreeLRN\_(node\_->r);

cout << node\_->data << " ";

}

}

void treeToString\_(Node\* node\_, string& str)

{

if (node\_ == NULL)

return;

str.append(to\_string(node\_->data));

if (!node\_->l && !node\_->r)

return;

str.push\_back('(');

treeToString\_(node\_->l, str);

str.push\_back(')');

if (node\_->r) {

str.push\_back('(');

treeToString\_(node\_->r, str);

str.push\_back(')');

}

}

void Convert2(vector<T> vector\_, int low, int high, Node\*& node\_)

{

if (low > high) {

return;

}

int mid = (low + high) / 2;

node\_ = new Node(vector\_[mid]);

Convert2(vector\_, low, mid - 1, node\_->l);

Convert2(vector\_, mid + 1, high, node\_->r);

}

Node\* Convert(vector<T> vector\_)

{

Node\* node\_ = nullptr;

Convert2(vector\_, 0, vector\_.size() - 1, node\_);

return node\_;

}

int BST\_To\_Vine(Node\* grand){

int count = 0;

Node\* tmp = grand->r;

while (tmp) {

if (tmp->l) {

Node\* oldTmp = tmp;

tmp = tmp->l;

oldTmp->l = tmp->r;

tmp->r = oldTmp;

grand->r = tmp;

}

else {

count++;

grand = tmp;

tmp = tmp->r;

}

}

return count;

}

void Compress(Node\* node\_, int m)

{

Node\* tmp = node\_->r;

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

Node\* oldTmp = tmp;

tmp = tmp->r;

node\_->r = tmp;

oldTmp->r = tmp->l;

tmp->l = oldTmp;

node\_ = tmp;

tmp = tmp->r;

}

}

Node\* BalanceDSW(Node\* node\_)

{

Node\* grand = new Node(0);

grand->r = node\_;

int count = BST\_To\_Vine(grand);

int h = log2(count + 1);

int m = pow(2, h) - 1;

Compress(grand, count - m);

for (m = m / 2; m > 0; m /= 2) {

Compress(grand, m);

}

return grand->r;

}

void FreeTree(Node\*& node\_) {

if (node\_ != nullptr)

{

FreeTree(node\_->l);

FreeTree(node\_->r);

delete node\_;

node\_ = nullptr;

}

}

};

int main()

{

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

Tree <int> tree;

int a, b, c, d, e;

cout << "Введіть розмір дерева: ";

cin >> a;

cout << "Введіть елементи:" << endl;

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

{

cin >> e;

tree.Insert(e);

}

cout << "Дерево:" << endl;

tree.PrintTree();

cout << "\n\n\n\n\n\n" << "Збалансоване дерево за масивом:" << endl;

tree.BalancedTree();

cout << "\n\n" << "Вхідне значення для батьківського елемента: ";

cin >> c;

tree.FindParent(c);

cout << "\n\n\n\n\n\n" << "Збалансоване дерево алгоритму DSW:" << endl;

tree.BalancedDSWTree();

cout << "\n\n" << "Вхідне значення для батьківського елемента: ";

cin >> c;

tree.FindParent(c);

cout << "\n\n" << "Дерево з використанням LNR:" << endl;

tree.printTreeLNR();

cout << "\n\n" << "Дерево з використаннням LRN:" << endl;

tree.printTreeLRN();

cout << "\n\n" << "Дерево з використанням NLR:" << endl;

tree.printTreeNLR();

cout << "\n\n" << "Дерево в рядок:" << endl;

tree.treeToString();

cout << "\n\n" << "Висота: " << tree.Height();

cout << "\n\n" << "Введіть значення для пошуку: ";

cin >> b;

cout << "\n" << "Чи містить воно число? (nullptr, якщо ні, покажчик елемента, якщо так): " << tree.Find(b);

cout << "\n" << "Чи містить воно число? (0 якщо ні, 1 якщо так): " << tree.IsContain(b);

cout << "\n\n" << "Мінімальне значення: " << tree.Min();

cout << "\n" << "Максимальне значення: " << tree.Max();

cout << "\n\n" << "Кількість елементів: " << tree.TreeElemCounter();

cout << "\n\n\n" << "Вхідне значення для стирання: ";

cin >> d;

if (tree.IsContain(d)) {

tree.Destroy(d);

cout << "\n" << "Дерево:" << endl;

tree.PrintTree();

}

else { cout << "Такого елемента в дереві немає!"; }

cout << "\n\n" << "Кількість елементів: " << tree.TreeElemCounter();

cout << "\n\n" << "Звільнення дерева:";

tree.FreeTheTree();

tree.PrintTree();

cout << "\n\n\n";

}

**Результати виконання програми:**

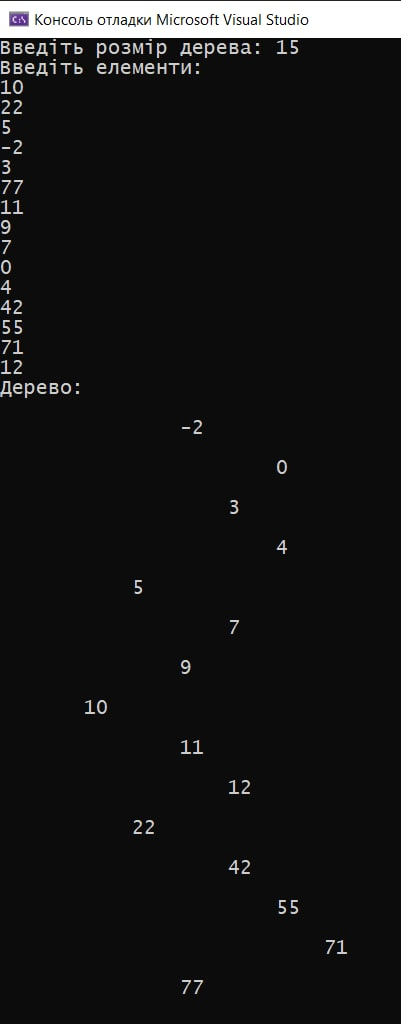


Рис. 1. Вигляд результату в консолі

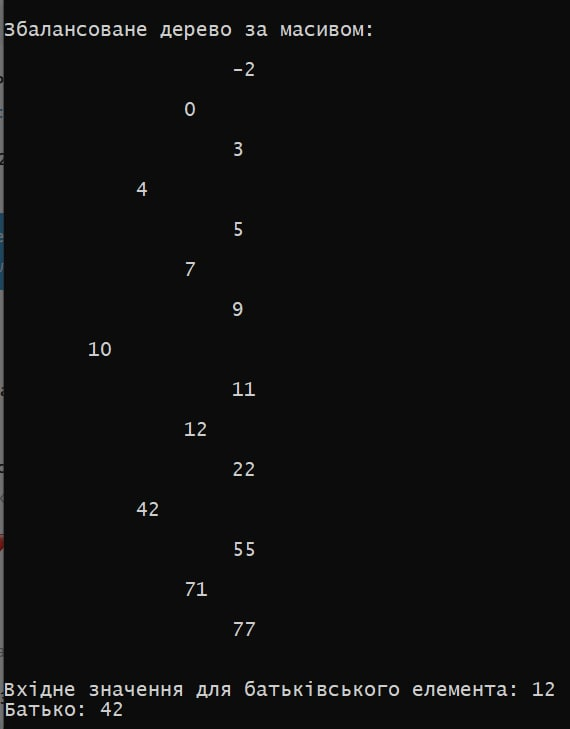


Рис. 2. Вигляд результату в консолі

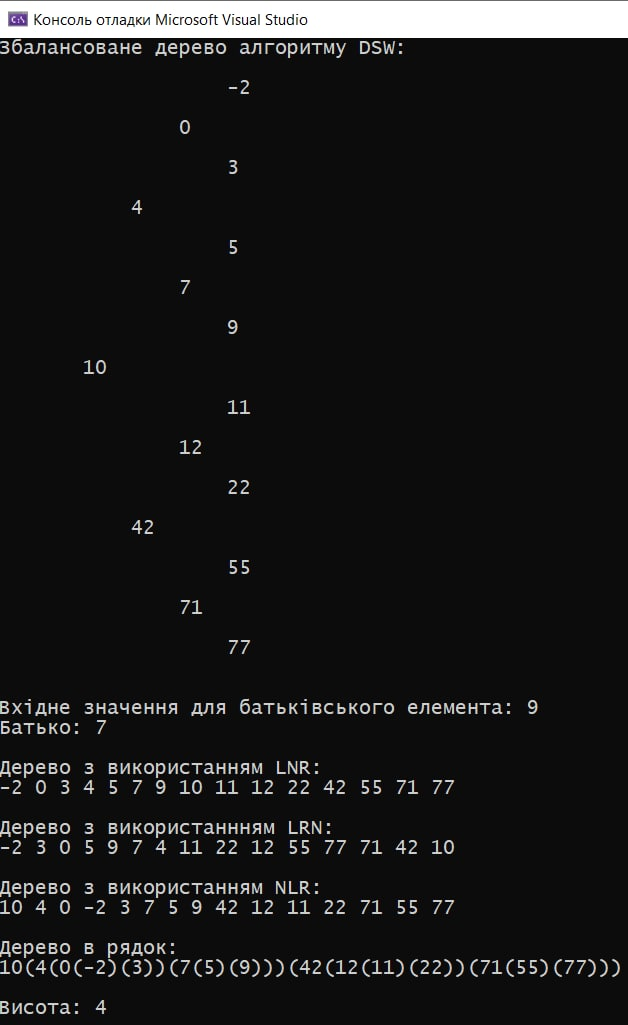


Рис. 3. Вигляд результату в консолі

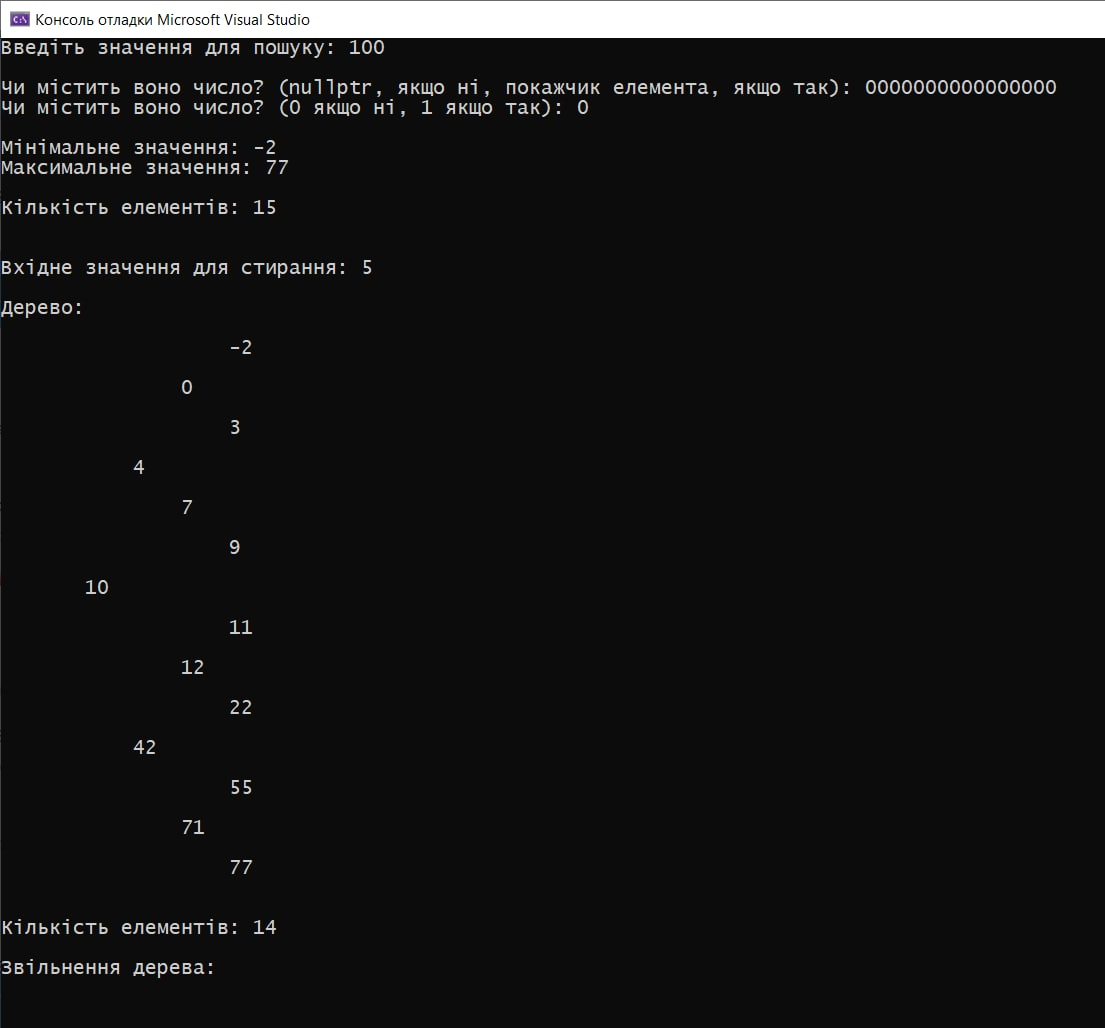


Рис. 4. Вигляд результату в консолі

***Висновки:*** я ознайомилася з принципами побудування бінарних дерев пошуку і найпростішими методами їх балансування.