МІНІСТРЕСТВО ОСВІТИ Й НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

“Харківський Політехнічний Інститут”

Кафедра управління проєктами в інформаційних технологіях

Звіт з лабораторної роботи №3

“Бінарні дерева та червоно-чорні дерева”

з дисципліни

“Алгоритми та структури даних”

Варіант №5

Перевірив: ст. викл. каф. УПІТ Мошко Є.О.

Виконав: ст. гр. КН-1223г Шинкаренко О.В.

Харків – 2024

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

**Завдання:**

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 по 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.

1. Дописати функцію видалення з пам’яті всього бінарного дерева.
2. Створити рекурсивну функцію, яка: 

видаляє ліве піддерево, і ліву гілку занулює; 

видаляє праве піддерево, і праву гілку занулює; 

видаляє сам вузол.

Наприкінці програми видалити з пам’яті дерево.

1. "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).
2. Розфарбувати вершини заданого бінарного дерева в червоний і чорний кольори так, щоб воно стало червоно-чорним деревом.
3. Провести перевірку властивості червоно-чорного дерева, а саме:

кожен вузол промаркований червоним або чорним кольором; корінь дерева – чорний;

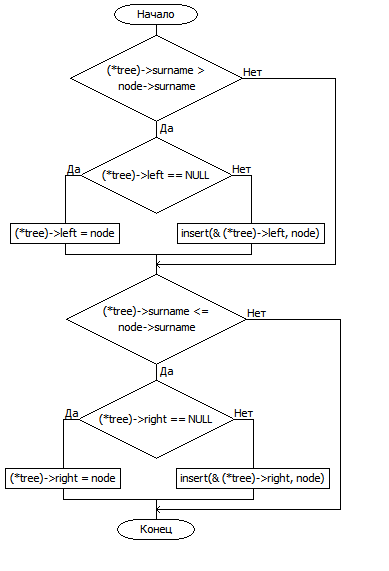
кінцеві вузли дерева – чорні;

біля червоного вузла батьківський вузол — чорний;

усі прості шляхи з будь-якого вузла до листя містять однакову кількість чорних вузлів.

1. У червоно-чорному дереві виконати додавання і видаленні вузлів та провести перевірку властивості червоно-чорного дерева. Наприкінці програми видалити з пам’яті дерево.

Вставка вузла в бінарне дерево



Блок-схема 1 Вставка вузла в бінарне дерево

Вхідні дані: нод, який потрібно вставити

Код функції:

void insert(tree\_node\*\* tree, tree\_node\* node) {

//node->height++;

if ((\*tree)->surname > node->surname) {

if ((\*tree)->left == NULL) (\*tree)->left = node;

else insert(&(\*tree)->left, node);

}

if ((\*tree)->surname <= node->surname) {

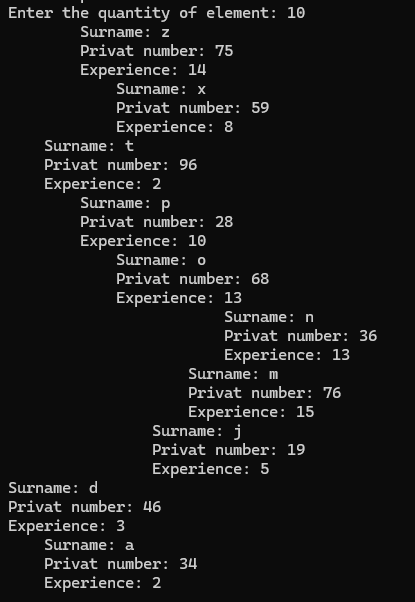
if ((\*tree)->right == NULL) (\*tree)->right = node;

else insert(&(\*tree)->right, node);

}

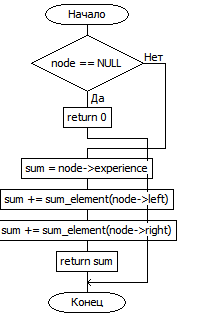
}

Результат у консолі:



Виведення у консоль 1 ставка вузла в бінарне дерево

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



Блок-схема 2 Знайти середнє значення одного з числових полів, зчитуючи дані з дерева

Вхідні дані: посилання на дерево

Код функції:

int count\_element(tree\_node\* node) {

int counter = 1;

if (node == NULL) return 0;

counter += count\_element(node->left);

counter += count\_element(node->right);

return counter;

}

int sum\_element(tree\_node\* node) {

if (node == NULL) return 0;

int sum = node->experience;

sum += sum\_element(node->left);

sum += sum\_element(node->right);

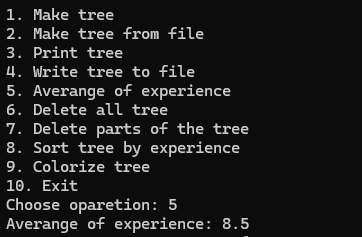
return sum;

}

double averange\_ex(tree\_node\* node) {

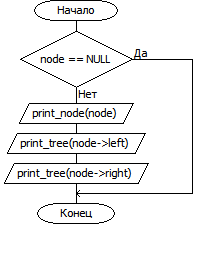
return (double)((double)sum\_element(node) / (double)count\_element(node));

}



Виведення у консоль 2 Знайти середнє значення одного з числових полів, зчитуючи дані з дерева

Прямий обхід дерева



Блок-схема 3 Прямий обхід дерева

Вхідні дані: посилання на корінь дерева

Код функції:

void print\_preorder\_tree(tree\_node\* node) {

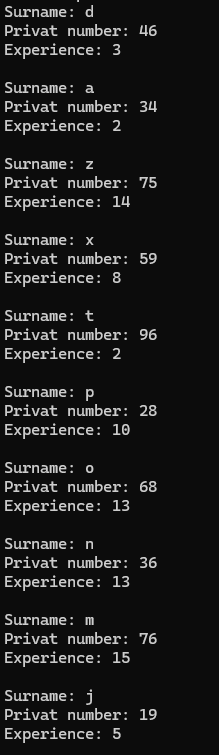
if (node == NULL) return;

print\_node(node);

print\_tree(node->left);

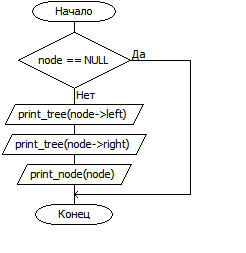
print\_tree(node->right);

}



Виведення у консоль 3 Прямий обхід дерева

Зворотній обхід дерева



Блок-схема 4 Зворотній обхід дерева

Вхідні дані: посилання на корінь дерева

Код функції:

void print\_postorder\_tree(tree\_node\* node) {

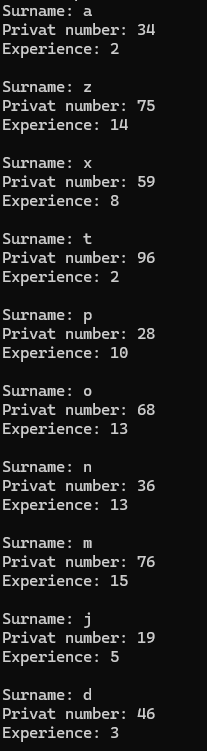
if (node == NULL) return;

print\_tree(node->left);

print\_tree(node->right);

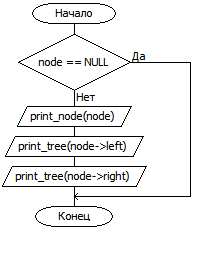
print\_node(node);

}



Виведення у консоль 4 Зворотній обхід дерева

Семитричний обхід дерева



Блок-схема 5 Семитричний обхід дерева

Вхідні дані: посилання на корінь дерева

Код функції:

void print\_tree(tree\_node\* node) {

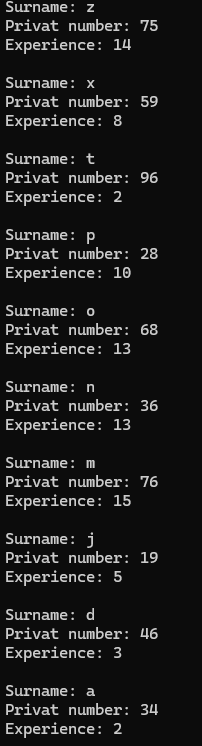
if (node == NULL) return;

print\_tree(node->right);

print\_node(node);

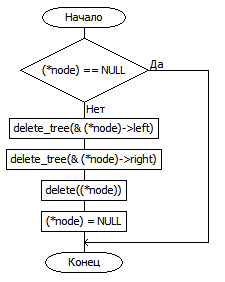
print\_tree(node->left);

}



Виведення у консоль 5 Семитричний обхід дерева

Видалення дерева з пам’яті



Блок-схема 6 Видалення дерева з пам’яті

Вхідні дані: посилання на корінь дерева

Код функції:

void delete\_tree(tree\_node\*\* node) {

if ((\*node) == NULL) return;

delete\_tree(&(\*node)->left);

delete\_tree(&(\*node)->right);

delete((\*node));

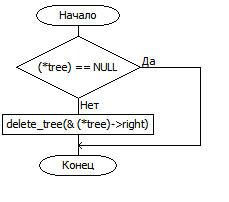
(\*node) = NULL;

}



Виведення у консоль 6 Видалення дерева з пам’яті

Видалення правої гілки



Блок-схема 7 Видалення правої гілки

Вхідні дані: посилання на корінь дерева

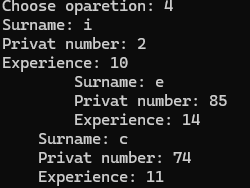
Код функції:

void delete\_right\_tree(tree\_node\*\* tree) {

if ((\*tree) == NULL) return;

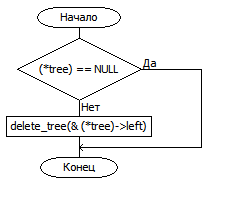
delete\_tree(&(\*tree)->right);

}



Виведення у консоль 7 Видалення правої гілки

Видалення лівої гілки



Блок-схема 8 Видалення лівої гілки

Вхідні дані: посилання на корінь дерева

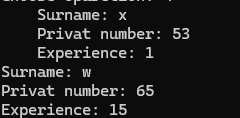
Код функції:

void delete\_left\_tree(tree\_node\*\* tree) {

if ((\*tree) == NULL) return;

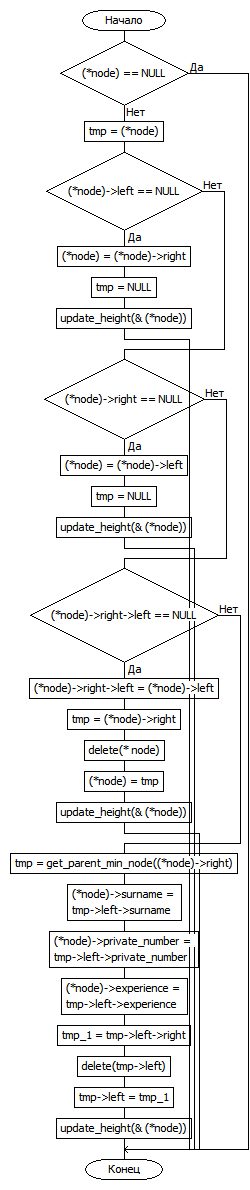
delete\_tree(&(\*tree)->left);

}



Виведення у консоль 8 Видалення лівої гілки

Видалення елемента



Блок-схема 9 Видалення елемента

Вхідні дані: посилання на корінь дерева, посилання на елемент

Код функції:

tree\_node\* get\_parent\_min\_node(tree\_node\* tree) {

//if (tree == NULL) return NULL;

//if (tree->right == NULL) return tree;

if (tree->left->left == NULL) return tree;

get\_parent\_min\_node(tree->left);

}

void delete\_node(tree\_node\*\* node) {

if ((\*node) == NULL) return;

tree\_node\* tmp = (\*node);

if ((\*node)->left == NULL) {

(\*node) = (\*node)->right;

delete tmp;

tmp = NULL;

update\_height(&(\*node));

return;

}

if ((\*node)->right == NULL) {

(\*node) = (\*node)->left;

delete tmp;

tmp = NULL;

update\_height(&(\*node));

return;

}

if ((\*node)->right->left == NULL) {

(\*node)->right->left = (\*node)->left;

tmp = (\*node)->right;

delete (\*node);

(\*node) = tmp;

update\_height(&(\*node));

return;

}

tmp = get\_parent\_min\_node((\*node)->right);

(\*node)->surname = tmp->left->surname;

(\*node)->private\_number = tmp->left->private\_number;

(\*node)->experience = tmp->left->experience;

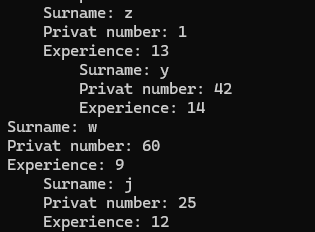
tree\_node\* tmp\_1 = tmp->left->right;

delete(tmp->left);

tmp->left = tmp\_1;

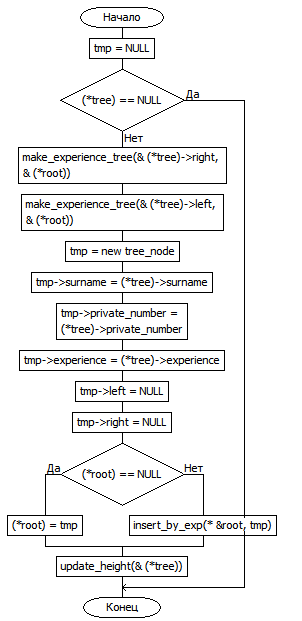
update\_height(&(\*node));

}



Виведення у консоль 9 Видалення елемента

Пересипання за досвідом



Блок-схема 10 Пересипання за досвідом

Вхідні дані: посилання на корінь дерева

Код функції:

void make\_experience\_tree(tree\_node\*\* tree, tree\_node\*\* root) {

tree\_node\* tmp = NULL;

if ((\*tree) == NULL) return;

make\_experience\_tree(&(\*tree)->right, &(\*root));

make\_experience\_tree(&(\*tree)->left, &(\*root));

tmp = new tree\_node;

tmp->surname = (\*tree)->surname;

tmp->private\_number = (\*tree)->private\_number;

tmp->experience = (\*tree)->experience;

tmp->left = NULL;

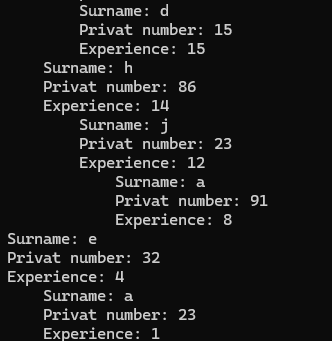
tmp->right = NULL;

if ((\*root) == NULL) (\*root) = tmp;

else insert\_by\_exp(\*&root, tmp);

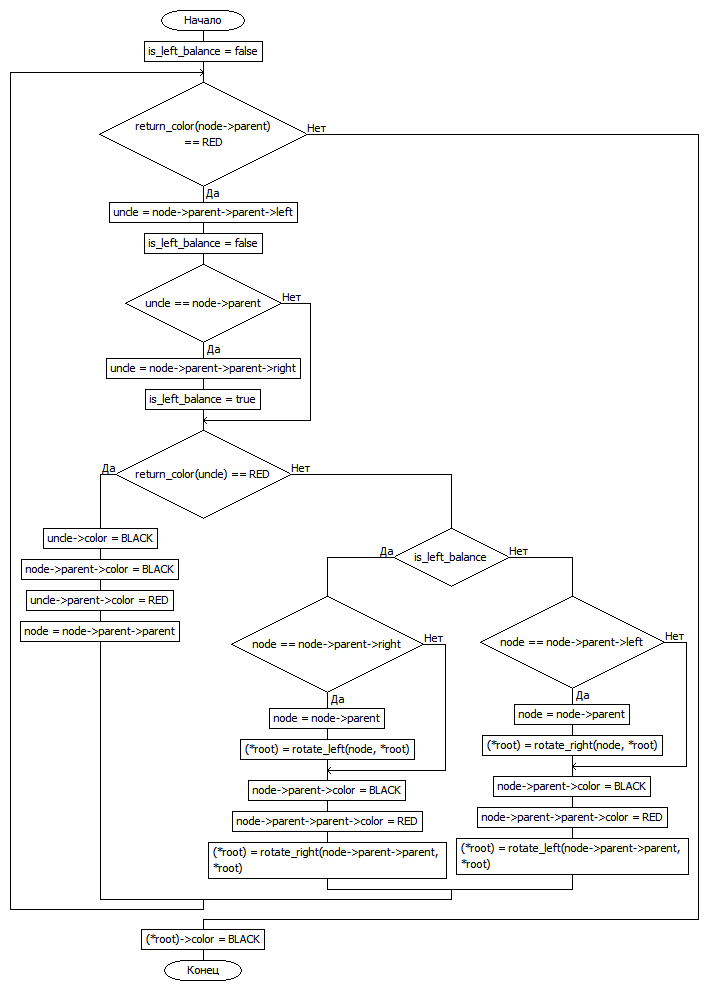
update\_height(&(\*tree));

}



Виведення у консоль 10 Пересипання за досвідом

Розфарбовування дерева



Блок-схема 11 Розфарбовування дерева

Вхідні дані: посилання на корінь бінарного дерева

Код функції:

void balance\_tree(rb\_tree\* node, rb\_tree\*\* root) {

rb\_tree\* uncle;

bool is\_left\_balance = false;

//while (return\_color(node->parent) == RED) {

while (return\_color(node->parent) == RED) {

uncle = node->parent->parent->left;

is\_left\_balance = false;

if (uncle == node->parent) {

uncle = node->parent->parent->right;

is\_left\_balance = true;

}

if (return\_color(uncle) == RED) {

uncle->color = BLACK;

node->parent->color = BLACK;

uncle->parent->color = RED;

node = node->parent->parent;

}

else {

if (is\_left\_balance) {

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

node = node->parent;

(\*root) = rotate\_left(node, \*root);

}

node->parent->color = BLACK;

node->parent->parent->color = RED;

(\*root) = rotate\_right(node->parent->parent, \*root);

}

else {

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

node = node->parent;

(\*root) = rotate\_right(node, \*root);

}

node->parent->color = BLACK;

node->parent->parent->color = RED;

(\*root) = rotate\_left(node->parent->parent, \*root);

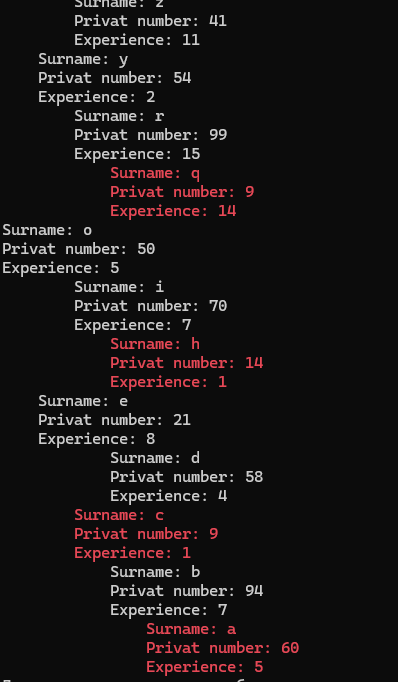
}

}

}

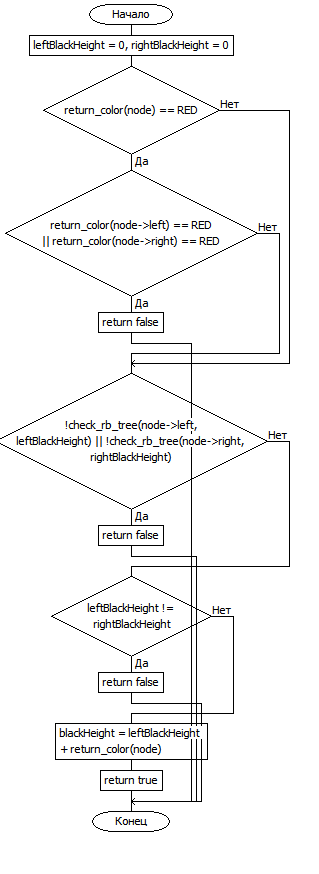
(\*root)->color = BLACK;

}



Виведення у консоль 11 Розфарбовування дерева

Перевірити властивості



Блок-схема 12 Перевірити властивості

Вхідні дані: посилання на корінь бінарного дерева

Код функції:

bool check\_rb\_tree(rb\_tree\* node, int& blackHeight) {

if (node == nullptr) {

blackHeight = 1;

return true;

}

int leftBlackHeight = 0, rightBlackHeight = 0;

if (return\_color(node) == RED) {

if (return\_color(node->left) == RED || return\_color(node->right) == RED) {

return false;

}

}

if (!check\_rb\_tree(node->left, leftBlackHeight) || !check\_rb\_tree(node->right, rightBlackHeight))

return false;

if (leftBlackHeight != rightBlackHeight) {

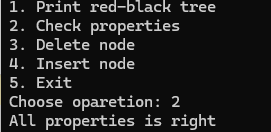
return false;

}

blackHeight = leftBlackHeight + return\_color(node);

return true;

}



Виведення у консоль 12 Перевірити властивості

**Висновки**

В процессі виконання роботи я навчилась реалізовувати та використовувати стеки, черги та деки. В ході виконання я помітила переваги та недоліки, які хочу зазначити.

Стек простий у реалізації та забезпечує швидкі операції додавання та видалення, але обмежує доступ до елементів лише з верхівки. Черга гарантує порядок обробки запитів і легко управляється, проте має обмежений доступ до елементів, що може спричинити затримки.

Дек, на відміну від стека і черги, забезпечує гнучкість, дозволяючи додавати та видаляти елементи з обох кінців, але його реалізація може бути складнішою і вимагати більше пам’яті через необхідність зберігати вказівники. Вибір між цими структурами даних залежить від конкретних вимог задачі та характеристик, що необхідні для ефективної обробки даних.