МІНІСТРЕСТВО ОСВІТИ Й НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

“Харківський Політехнічний Інститут”

Кафедра управління проєктами в інформаційних технологіях

Звіт з лабораторної роботи №3

“Однозв’язні та двозв’язні списки й робота з ними”

з дисципліни

“Алгоритми та структури даних”

Варіант №2

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

Виконав: ст. гр. КН-1223г Ставицький А.А.

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.

1. ﻿﻿﻿Знайти середнє значення одного з числових полів, зчитуючи дані з дерева.
2. ﻿﻿﻿Надрукувати значення бінарного дерева: а) при прямому обході дерева;б) при зворотному обході дерева; в) при симетричному обході дерева.
3. ﻿﻿﻿Дописати функцію видалення з памʼяті всього бінарного дерева.
4. ﻿﻿﻿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 памʼяті 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.

**Бінарне дерево** – це структура даних, у якій кожен вузол має не більше двох дочірніх вузлів, які називаються лівим і правим. Перший вузол у дереві називається коренем, а кожен дочірній вузол може містити власні дочірні вузли. Бінарні дерева широко використовуються для зберігання і впорядкування даних, що дозволяє швидко здійснювати пошук, додавання та видалення елементів. Існують різні типи бінарних дерев, зокрема бінарне дерево пошуку (BST), де для кожного вузла всі значення в лівому піддереві менші за значення в цьому вузлі, а всі значення в правому піддереві — більші; повне бінарне дерево, в якому кожен рівень, окрім останнього, повністю заповнений; збалансоване бінарне дерево, де висоти лівого і правого піддерева кожного вузла відрізняються не більше ніж на одиницю; та інші спеціалізовані дерева, наприклад, дерево Хаффмана, яке використовується в алгоритмах стиснення даних.

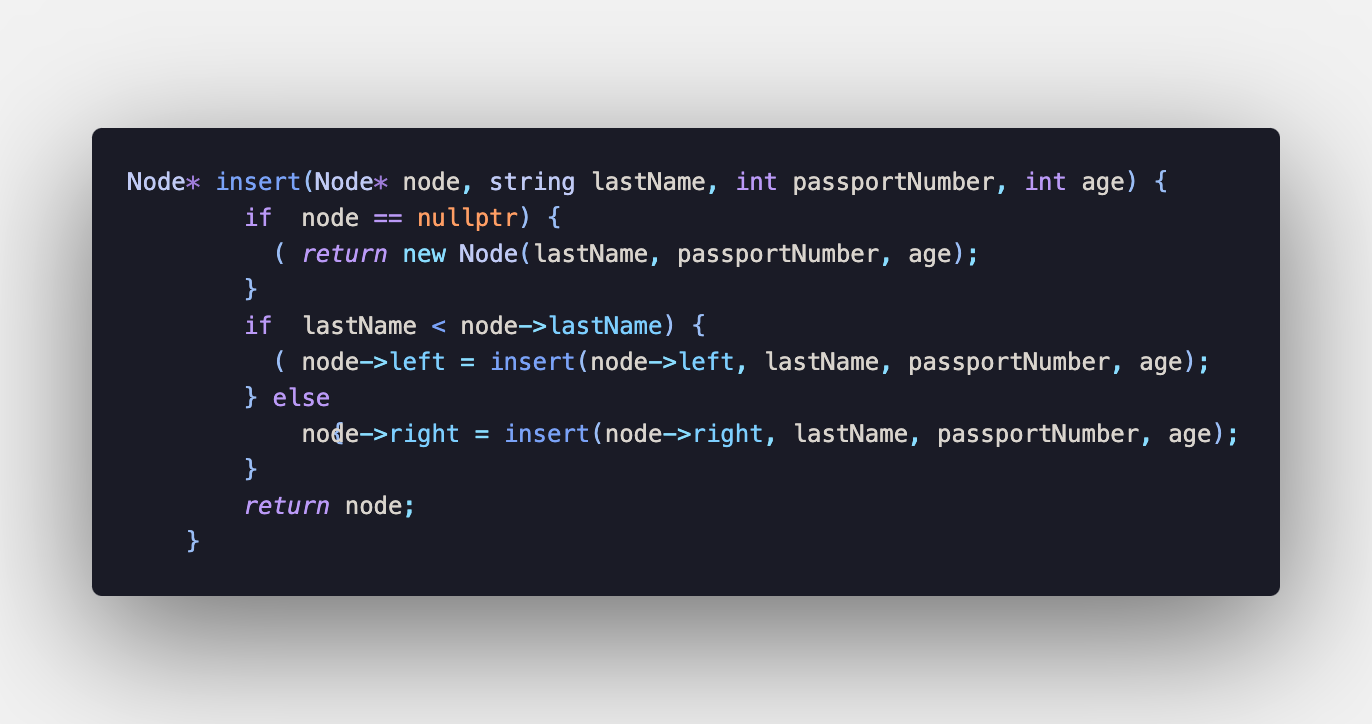
Функція insert вставляє новий елемент у бінарне дерево пошуку за прізвищем. (Див. Рис. 1) Ось як це відбувається:

1. **Перевірка на порожній вузол**: Якщо node дорівнює nullptr, це означає, що ми знайшли місце для нового елемента. У такому випадку створюється новий вузол з переданими параметрами lastName, passportNumber, та age, і повертається цей новий вузол.

2. **Рекурсивна вставка в піддерево**: Якщо lastName менше, ніж прізвище у поточному вузлі node->lastName, функція викликає саму себе рекурсивно для лівого піддерева (node->left). Це означає, що новий вузол буде розміщено ліворуч, якщо це можливо.

• Інакше, якщо lastName не менше, ніж у поточного вузла, виконується рекурсивний виклик для правого піддерева (node->right), і новий вузол буде розміщено праворуч.

3. **Повернення вузла**: Після виконання вставки функція повертає корінь оновленого піддерева, забезпечуючи зв’язок у дереві.



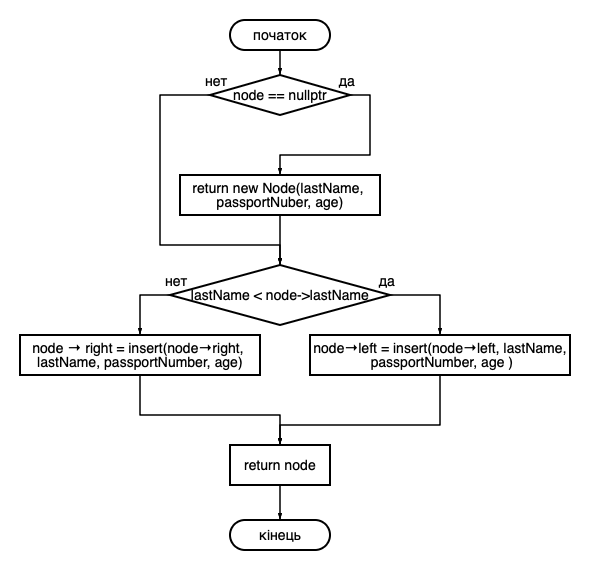


Рис. 1 Блок-схема функції insert

Функція displayPreOrder виконує обхід дерева в **прямому порядку** (pre-order traversal), виводячи інформацію про кожен вузол. (Див. рис. 2) Ось як вона працює:

1. **Перевірка на порожній вузол**: Якщо node дорівнює nullptr, функція повертається, завершуючи виконання для цього вузла. Це базовий випадок рекурсії, який зупиняє подальші виклики для порожніх вузлів.

2. **Виведення інформації про вузол**: Якщо вузол не порожній, функція виводить lastName, passportNumber і age поточного вузла у заданому форматі.

3. **Рекурсивні виклики для дочірніх вузлів**: Функція викликає саму себе для лівого піддерева (displayPreOrder(node->left)), продовжуючи обхід у лівому напрямку.

• Після завершення лівого піддерева, виконується рекурсивний виклик для правого піддерева (displayPreOrder(node->right)), продовжуючи обхід у правому напрямку.



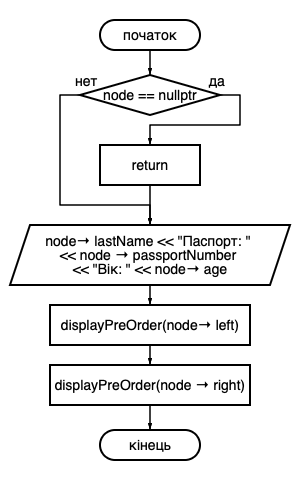


Рис. 2 Блок-схема до функції displayPreOrder

Функція displayInOrder виконує обхід бінарного дерева в **середньому порядку** (in-order traversal), виводячи інформацію про кожен вузол.(Див. рис. 3) Ось як вона працює:

1. **Перевірка на порожній вузол**: Якщо node дорівнює nullptr, функція повертається, завершуючи обхід для цього вузла. Це базовий випадок рекурсії, який зупиняє подальші виклики для порожніх вузлів.

2. **Рекурсивний обхід лівого піддерева**: Функція спочатку викликає саму себе для лівого піддерева (displayInOrder(node->left)). Це забезпечує обробку всіх вузлів лівого піддерева перед кореневим вузлом.

3. **Виведення інформації про вузол**: Після обходу лівого піддерева виводиться інформація про поточний вузол: lastName, passportNumber, та age.

4. **Рекурсивний обхід правого піддерева**: Після виведення інформації про поточний вузол функція викликає саму себе для правого піддерева (displayInOrder(node->right)), завершуючи обхід правих вузлів.



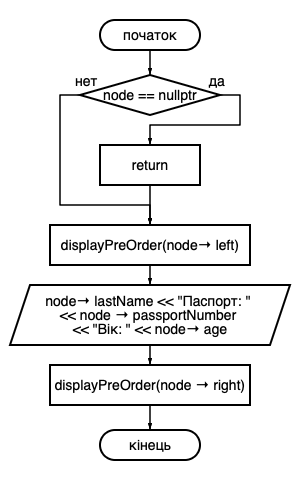


Рис. 3 Блок-схема до функції displayInOrder

Функція displayPostOrder виконує обхід дерева в **зворотному порядку** (post-order traversal), виводячи інформацію про кожен вузол. (Див. рис. 4) Ось як вона працює:

1. **Перевірка на порожній вузол**: Якщо node дорівнює nullptr, функція повертається, що зупиняє виконання для цього вузла. Це базовий випадок рекурсії, який не допускає подальших викликів для порожніх вузлів.

2. **Рекурсивний обхід правого піддерева**: Спочатку функція викликає саму себе для правого піддерева (displayPostOrder(node->right)), що дозволяє обробити всі вузли правого піддерева до обробки поточного вузла.

3. **Рекурсивний обхід лівого піддерева**: Після обходу правого піддерева функція переходить до лівого піддерева, викликаючи displayPostOrder(node->left). Це забезпечує обробку всіх вузлів лівого піддерева після правого.

4. **Виведення інформації про вузол**: Після обробки обох піддерев функція виводить інформацію про поточний вузол: lastName, passportNumber, та age.



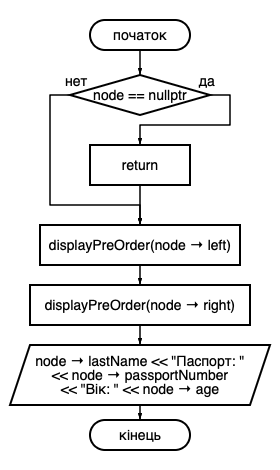


Рис. 4 Блок-схема до функції displayPostOrder

Функція displayTree призначена для візуалізації бінарного дерева у текстовому форматі, зокрема для наочності структури дерева. (Див. рис. 5) Ось як вона працює:

1. **Перевірка на порожній вузол**: Якщо node дорівнює nullptr, функція повертається, що зупиняє виконання для цього вузла. Це базовий випадок рекурсії, який запобігає подальшим викликам для порожніх вузлів.

2. **Обхід лівого піддерева**:

• Якщо у вузла є ліве піддерево (тобто node->left не дорівнює nullptr), функція викликає саму себе для лівого піддерева.

• При цьому prefix оновлюється, додаючи пробіли (залежно від того, чи є вузол лівим чи правим дочірнім), щоб підтримувати правильне форматування.

• Після рекурсивного виклику виводиться стрілка (”/”), що вказує на зв’язок із лівим піддеревом.

3. **Виведення інформації про вузол**: Функція виводить інформацію про поточний вузол, зокрема lastName, passportNumber, та age, з використанням наявного префіксу для візуалізації ієрархії.

4. **Обхід правого піддерева**:

• Якщо у вузла є праве піддерево (тобто node->right не дорівнює nullptr), перед рекурсивним викликом для правого піддерева виводиться стрілка (”\”), що вказує на зв’язок із правим піддеревом.

• Функція знову викликає саму себе для правого піддерева, використовуючи оновлений prefix.



Функція deleteLeftSubtree призначена для видалення лівого піддерева вказаного вузла у бінарному дереві. (Див. Рис. 6) Ось як вона працює:

1. **Перевірка наявності лівого піддерева**:

• Якщо node не є nullptr і у нього є ліве піддерево (node->left != nullptr), то функція продовжує виконання.

• У противному випадку, якщо лівого піддерева не існує, виводиться повідомлення про це.

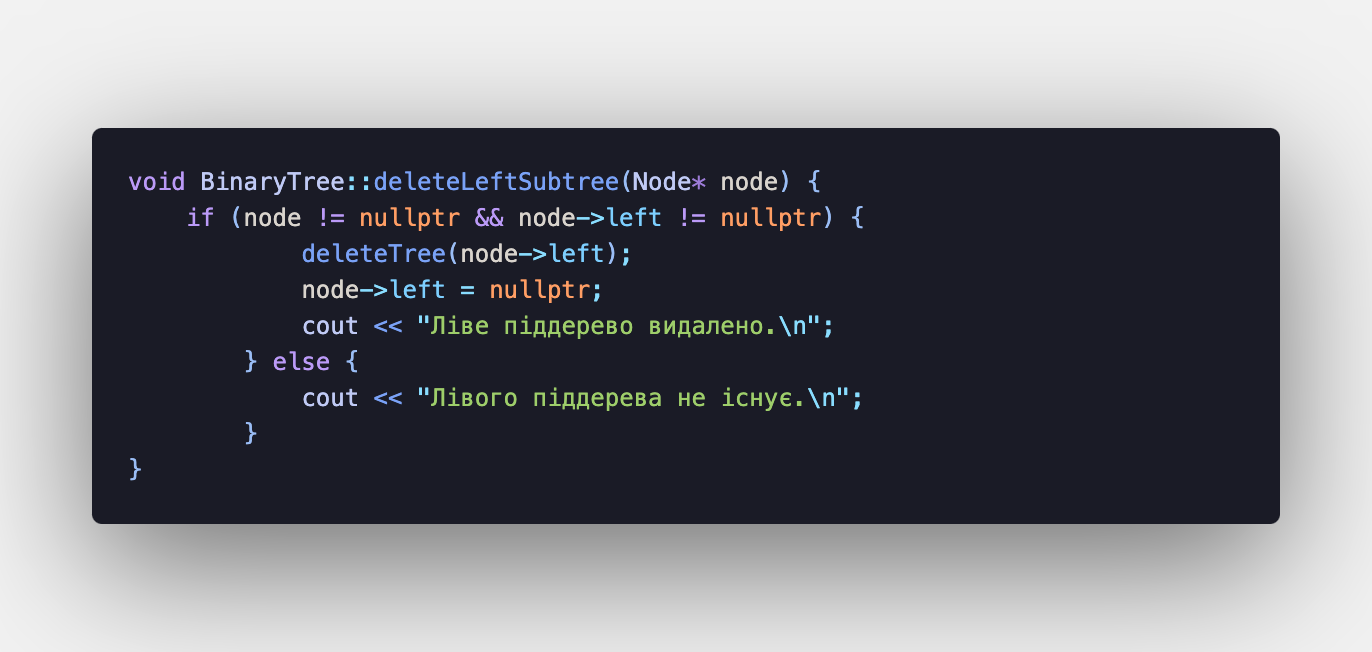
2. **Видалення лівого піддерева**:

• Якщо ліве піддерево існує, викликається функція deleteTree, яка, ймовірно, відповідає за рекурсивне видалення всіх вузлів лівого піддерева. (При цьому, для коректності реалізації, функція deleteTree повинна сама включати в себе логіку видалення вузлів, щоб уникнути витоків пам’яті.)

• Після видалення лівого піддерева, посилання node->left скидається на nullptr, що вказує на те, що тепер лівого піддерева немає.

3. **Вивід повідомлення**:

• Після успішного видалення лівого піддерева виводиться повідомлення “Ліве піддерево видалено.”.



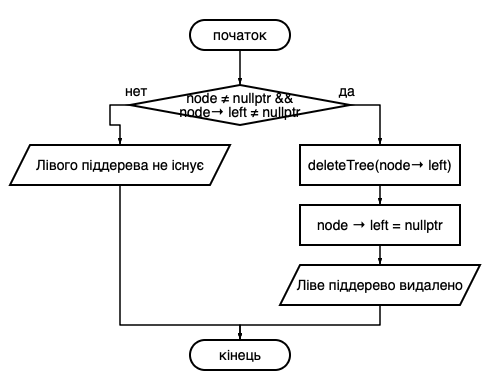


Рис. 6 Блок-схема до функції deleteLeftSubstree

Функція deleteRightSubtree відповідає за видалення правого піддерева вказаного вузла в бінарному дереві. (Див. Рис. 7) Ось як вона працює:

1. **Перевірка наявності правого піддерева**:

• Функція спочатку перевіряє, чи node не є nullptr та чи у нього є праве піддерево (node->right != nullptr). Якщо обидві умови виконуються, функція продовжує виконання.

• Якщо ж правого піддерева не існує, виводиться повідомлення “Правого піддерева не існує.”.

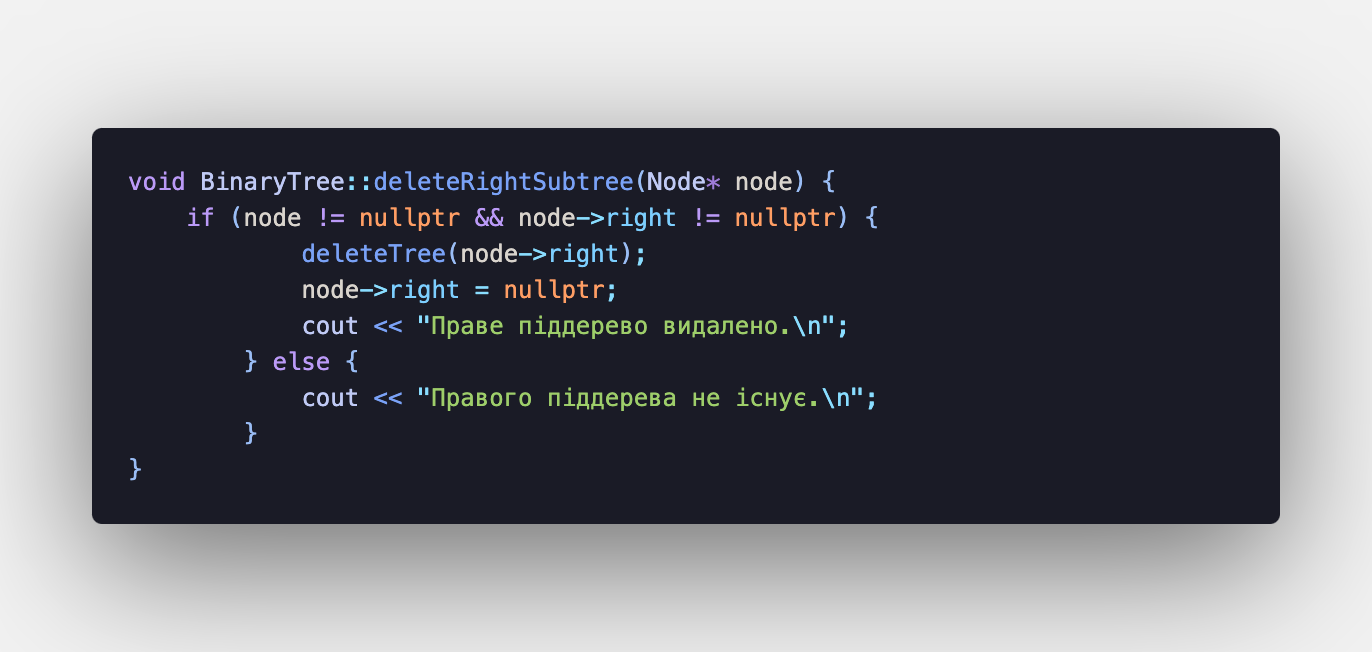
2. **Видалення правого піддерева**:

• Якщо праве піддерево існує, викликається функція deleteTree, яка, ймовірно, рекурсивно видаляє всі вузли правого піддерева. (При цьому, для забезпечення коректності реалізації, функція deleteTree повинна включати логіку видалення вузлів, щоб уникнути витоків пам’яті.)

• Після видалення правого піддерева, посилання node->right скидається на nullptr, що вказує на те, що правого піддерева більше немає.

3. **Вивід повідомлення**:

• Якщо видалення пройшло успішно, функція виводить повідомлення “Праве піддерево видалено.”.



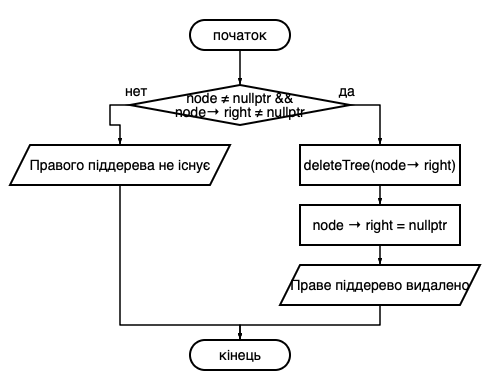


Рис. 7 Блок-схема до функції deleteRightSubstree

Функція calculateAverage розраховує середнє значення віку (або іншого показника) для всіх вузлів у бінарному дереві. (Див. Рис. 8) Ось як вона працює:

1. **Перевірка на порожній вузол**:

• Якщо node дорівнює nullptr, функція повертає 0. Це базовий випадок рекурсії, який зупиняє виконання для порожніх вузлів.

2. **Рекурсивний обхід лівого піддерева**:

• Функція рекурсивно викликає саму себе для лівого піддерева (calculateAverage(node->left, count)) та зберігає результат у змінній leftSum. Цей виклик обчислює суму віку всіх вузлів у лівому піддереві.

3. **Рекурсивний обхід правого піддерева**:

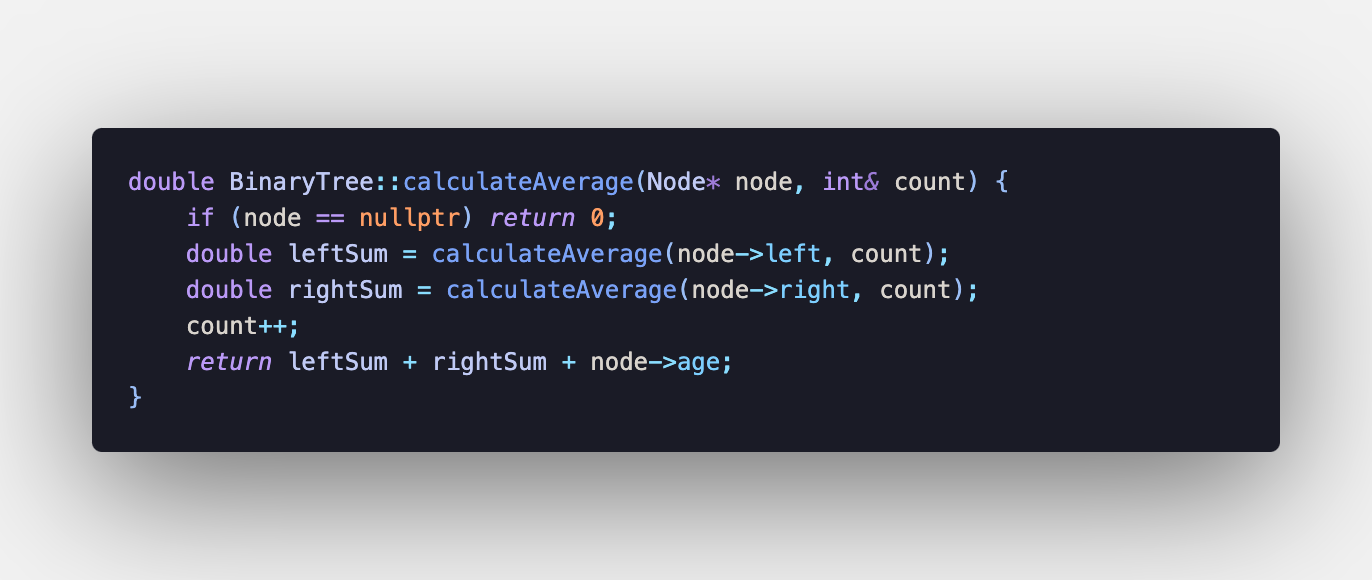
• Аналогічно, функція викликає себе для правого піддерева (calculateAverage(node->right, count)) та зберігає результат у змінній rightSum. Це обчислює суму віку всіх вузлів у правому піддереві.

4. **Підрахунок кількості вузлів**:

• Після обчислення сум функція збільшує лічильник count на 1, що вказує на те, що один вузол (поточний) було оброблено.

5. **Повернення загальної суми**:

• Функція повертає загальну суму, що складається з leftSum, rightSum та віку поточного вузла (node->age).



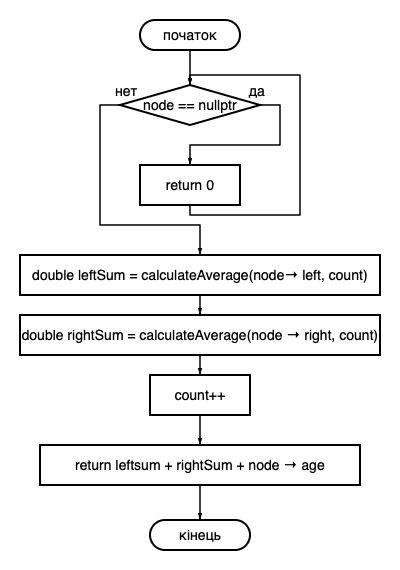


Рис. 8 Блок-схема до функції calculateAverage

Функція printAverage відповідає за виведення середнього значення віку всіх вузлів у бінарному дереві.(Див. Рис. 9) Ось як вона працює:

1. **Ініціалізація змінних**:

• Оголошується змінна count, яка буде використовуватися для підрахунку кількості вузлів у дереві, і ініціалізується значенням 0.

• Оголошується змінна sum, яка буде використовуватися для зберігання загальної суми віку вузлів. Її значення обчислюється за допомогою виклику функції calculateAverage з коренем дерева (root) як аргументом.

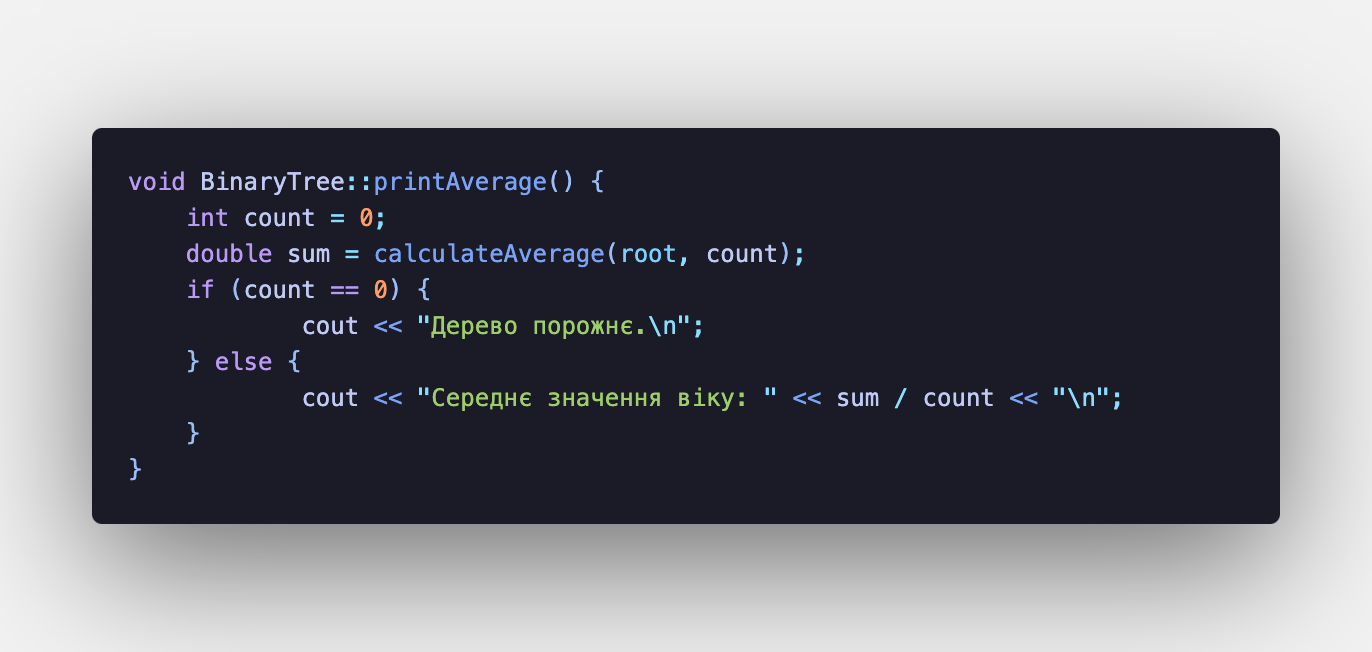
2. **Перевірка наявності вузлів**:

• Якщо count дорівнює 0, це означає, що дерево порожнє, і функція виводить повідомлення “Дерево порожнє.”

3. **Виведення середнього значення**:

• Якщо кількість вузлів більша за 0, функція обчислює середнє значення, поділивши загальну суму sum на кількість вузлів count.

• Результат виводиться на екран у форматі “Середнє значення віку: [значення]”.



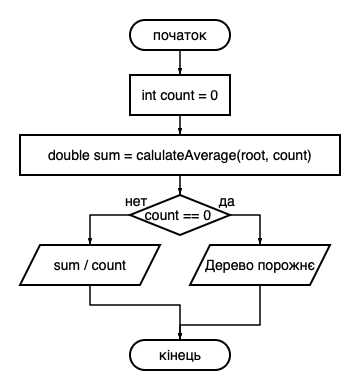


Рис. 9 Блок-схема до функції printAverage

Функція deleteTree відповідає за рекурсивне видалення всіх вузлів у бінарному дереві, починаючи з заданого вузла. (Див. Рис. 10) Ось як вона працює:

1. **Перевірка наявності вузла**:

• Функція спочатку перевіряє, чи node не є nullptr. Якщо це так, то виклик функції завершується (це базовий випадок рекурсії, який запобігає подальшим викликам для порожніх вузлів).

2. **Рекурсивне видалення лівого піддерева**:

• Якщо вузол існує, функція викликає саму себе для лівого піддерева (deleteTree(node->left)). Це призводить до рекурсивного видалення всіх вузлів у лівій частині дерева.

3. **Рекурсивне видалення правого піддерева**:

• Аналогічно, функція викликає себе для правого піддерева (deleteTree(node->right)), що викликає видалення всіх вузлів у правій частині дерева.

4. **Видалення поточного вузла**:

• Після видалення всіх дочірніх вузлів, функція виконує delete node;, що звільняє пам’ять, зайняту поточним вузлом.



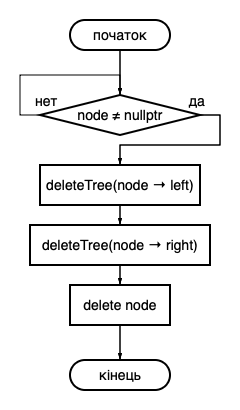


Рис. 10 Блок-схема до функції deleteTree

**Червоне-чорне дерево** — це самобалансуюче бінарне дерево, яке забезпечує ефективні операції вставки, видалення та пошуку з логарифмічною складністю в середньому та в найгіршому випадках. Кожен вузол у червоно-чорному дереві має два атрибути: ключ і колір, який може бути червоним або чорним.

Основні властивості червоно-чорного дерева включають:

1. **Колір вузлів**: Кожен вузол може бути червоним або чорним.

2. **Корінь чорний**: Корінь дерева завжди чорний.

3. **Червоні вузли**: Якщо вузол червоний, то його діти (нащадки) повинні бути чорними. Це означає, що не може бути двох червоних вузлів підряд.

4. **Чорна висота**: Для будь-якого вузла, всі простежувані шляхи від вузла до його нащадків (листків) повинні мати однакову кількість чорних вузлів. Цю кількість називають “чорною висотою” вузла.

5. **Листки**: Листки дерева (які є «пустими» вузлами) завжди чорні.

Завдяки цим властивостям, червоно-чорні дерева зберігають приблизно збалансовану структуру, що дозволяє підтримувати висоту дерева пропорційною логарифму кількості вузлів. Це робить їх особливо корисними для реалізації асоціативних масивів, словників, множин та інших структур даних, де необхідні швидкі операції вставки, видалення та пошуку.

Функція balanceTree виконує балансування червоно-чорного дерева після вставки нового вузла. Червоно-чорне дерево є спеціальним видом бінарного дерева, яке забезпечує балансування, що дозволяє підтримувати логарифмічну висоту. (Див. Рис. 11)Ось як працює ця функція:

**Основні етапи роботи функції**

1. **Цикл для балансування**:

• Цикл while продовжується до тих пір, поки node не є коренем, і поки батьківський вузол є червоним (node->parent->color == true).

2. **Перевірка батька**:

• Якщо node є лівим нащадком батька (node->parent == node->parent->parent->left), виконується балансування лівого піддерева:

• **Знаходження дядька**: Дядько (uncle) визначається як правий нащадок дідуся.

• **Випадок 1 (дядько червоний)**: Якщо дядько не є nullptr і також червоний (uncle->color == true):

• Батько node і дядько стають чорними (color = false).

• Дідусь стає червоним (color = true).

• node перенаправляється на дідуся, що дозволяє продовжити балансування вищими рівнями.

• **Випадок 2 (дядько чорний)**: Якщо дядько чорний:

• Якщо node є правим нащадком батька, виконується обертання вліво (функція rotateLeft).

• Батько node стає чорним, а дідусь стає червоним.

• Виконується обертання вправо (функція rotateRight).

3. **Балансування правого піддерева**:

• Якщо node є правим нащадком батька, то логіка аналогічна, але відбувається з перевернутими ролями:

• Визначається лівий дядько, і перевіряються його кольори.

• Знову, якщо дядько червоний, то відбувається зміна кольорів, і node переміщається вгору по дереву.

• Якщо дядько чорний, виконується обертання вправо або вліво, залежно від положення node.

4. **Завершення**:

• Після виходу з циклу (коли дерево стало збалансованим), корінь дерева (root) завжди стає чорним, щоб дотримуватися властивостей червоно-чорного дерева.



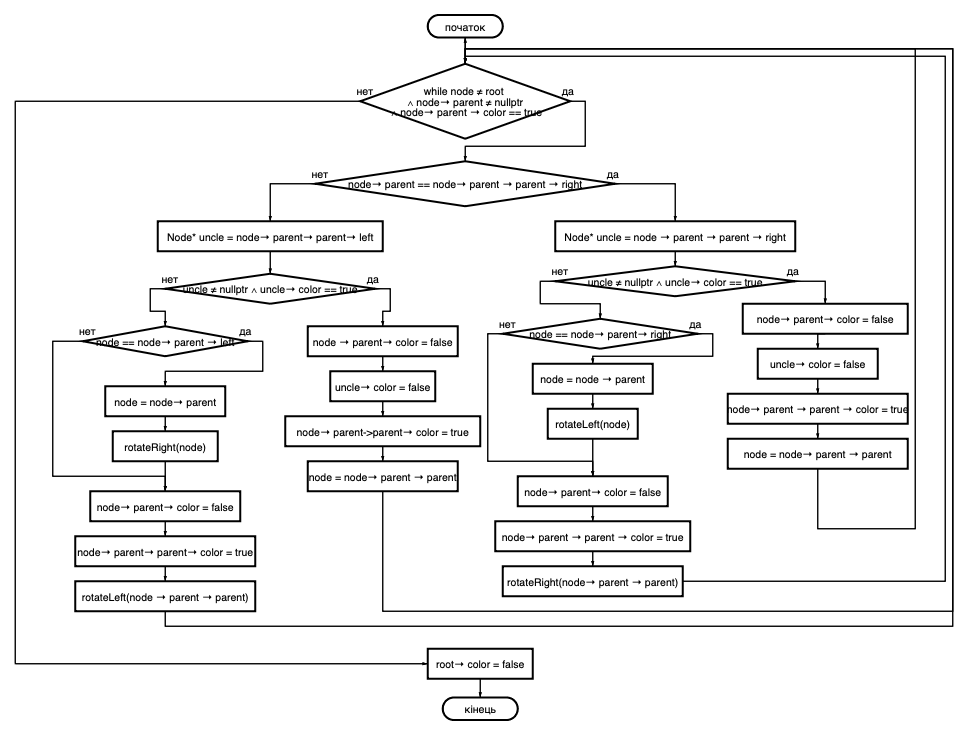


Рис. 11 Блок-схема до функції balanceTree

Функція rotateRight реалізує обертання вправо в червоно-чорному дереві. Це важлива операція для балансування дерева після вставки або видалення вузлів. (Див. Рис. 12) Ось детальний опис того, як працює ця функція:

**Опис роботи функції**

1. **Визначення вузла для обертання**:

• Node\* y = node->left;: Вузол y є лівим нащадком вузла node, який ми будемо обертати вправо.

2. **Збереження правого піддерева**:

• node->left = y->right;: Праве піддерево вузла y (якщо воно існує) переміщується на місце лівого піддерева вузла node.

3. **Оновлення батьківського вузла**:

• if (y->right != nullptr) { y->right->parent = node; }: Якщо у y є правий нащадок, то його батьківський вузол тепер стає node.

4. **Зміна батьківського зв’язку для** y:

• y->parent = node->parent;: Батьківський вузол y стає батьківським вузлом node.

5. **Оновлення зв’язків для кореня**:

• if (node->parent == nullptr) { root = y; }: Якщо node є коренем дерева, то y стає новим коренем.

• else if (node == node->parent->right) { node->parent->right = y; }: Якщо node є правим нащадком свого батька, то y стає правим нащадком батька.

• else { node->parent->left = y; }: Інакше y стає лівим нащадком батька.

6. **Переміщення** node **під** y:

• y->right = node;: Тепер node стає правим нащадком y.

• node->parent = y;: Батьківський вузол для node змінюється на y.



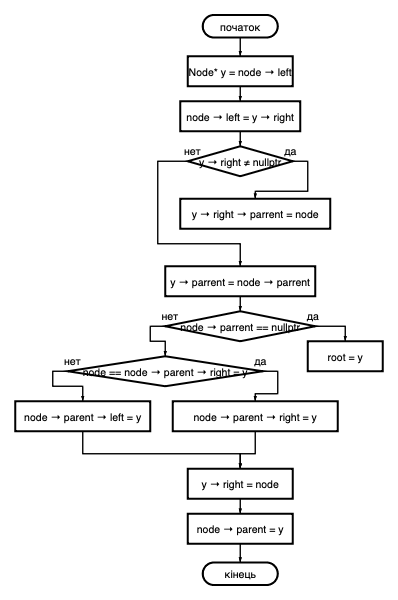


Рис. 12 Блок-схема функції rotateRight

Функція rotateLeft реалізує обертання вліво в червоно-чорному дереві. Це важлива операція, яка використовується для балансування дерева після вставки або видалення вузлів. (Див. Рис. 13) Ось детальний опис того, як працює ця функція:

**Опис роботи функції**

1. **Визначення вузла для обертання**:

• Node\* y = node->right;: Вузол y є правим нащадком вузла node, який ми будемо обертати вліво.

2. **Збереження лівого піддерева**:

• node->right = y->left;: Ліве піддерево вузла y (якщо воно існує) переміщується на місце правого піддерева вузла node.

3. **Оновлення батьківського зв’язку**:

• if (y->left != nullptr) { y->left->parent = node; }: Якщо у y є лівий нащадок, то його батьківський вузол тепер стає node.

4. **Зміна батьківського зв’язку для** y:

• y->parent = node->parent;: Батьківський вузол y стає батьківським вузлом node.

5. **Оновлення зв’язків для кореня**:

• if (node->parent == nullptr) { root = y; }: Якщо node є коренем дерева, то y стає новим коренем.

• else if (node == node->parent->left) { node->parent->left = y; }: Якщо node є лівим нащадком свого батька, то y стає лівим нащадком батька.

• else { node->parent->right = y; }: Інакше y стає правим нащадком батька.

6. **Переміщення** node **під** y:

• y->left = node;: Тепер node стає лівим нащадком y.

• node->parent = y;: Батьківський вузол для node змінюється на y.



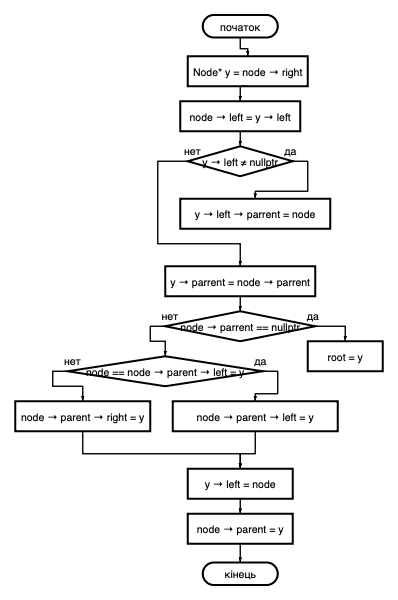


Рис. 13 Блок-схема функції rotateLeft

**Висновки:** Під час виконання лабораторної роботи я мав можливість детально вивчити структури даних, зокрема звичайні бінарні дерева та червоно-чорні дерева.

Працюючи з звичайними бінарними деревами, я зрозумів основні принципи їх роботи, включаючи вставку, видалення та обходи (прямий, симетричний і зворотний). Ці операції дозволяють ефективно організувати дані, проте виявилися недостатньо оптимальними в плані часу виконання, особливо у випадках, коли дерево стає незбалансованим.

Вивчаючи червоно-чорні дерева, я зрозумів, як їх властивості допомагають підтримувати баланс. Я навчився виконувати обертання вузлів, що є критично важливим для забезпечення швидкості операцій. Завдяки своїм властивостям червоно-чорні дерева гарантують, що висота дерева залишається логарифмічною від кількості вузлів, що робить їх набагато ефективнішими для великих наборів даних.

Загалом, виконання лабораторної роботи дало мені цінні знання про різні типи дерев і їх переваги, що в майбутньому дозволить обирати оптимальні рішення для конкретних задач у програмуванні.