МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

Кафедра інформаційних систем та технологій

Звіт

з комп'ютерного практикуму 5(7-8)

«Основні типи двійкові дерева пошуку»

з дисципліни

«Теорія алгоритмів»

Бригада – 10

Варіант № 1

Перевірила:

ст. вик. Солдатова М.О.

Виконали:

Бойко Катерина,

Гоголь Софія,

Павлова Софія,

Хіврич Володимир

Київ 2022

Комп’ютерний практикум 7-8

Тема: Основні типи двійкові дерева пошуку

Мета роботи: дослідження характеристик продуктивності основних типів двійкових дерев пошуку.

Завдання

Постановка задачі:

Варіант 1:

Допоможіть лепрікону якомога швидше знайти скарб зі злитків золота в одній з печер. Кожна печера містить різну кількість злитків, печери з’єднані тунелями. Визначте найзручнішу печеру для початку пошуку та покажіть шлях до заданого скарбу. Кількість шуканих злитків задається з клавіатури. Якщо такої печери не існує, то треба її викопати та з’єднати тунелем і там чарівним чином з’явиться золото. Знову покажіть лепрікону шлях. Якщо тунель обвалився, то печера стає недосяжною і її треба викреслити з можливих пошуків. Повідомити про це.

Мета задачі – допомогти лепрікону якомога швидше знайти скарб зі злитків золота, визначити найзручнішу печеру для початку пошуку та показати йому шлях до неї. Якщо печери не існує, її треба викопати та з'єднати тунелем і там чарівним чином з'явиться золото. Якщо тунель обвалився, то печера стає недосяжною, її треба викреслити з можливих пошуків.

Використані структури даних: масиви, покажчики, цілочисельні типи, створені власні класи (Node).

Модель:

Основні величини:

Назва змінної	Тип змінної	Значення змінної
left	Node*	корінь лівого піддерева
right	Node*	корінь правого піддерева
build(const vector<int>& arr, int lower, int upper)	Node*	повертає покажчик на кореневий вузол піддерева
buildBSTfromSortedArr(int* arr, int n)	Node*	створює скошене вправо дерево двійкового пошуку.
createArrayFromTree(Node* root, vector<int> arr = vector<int>())	vector<int>	створює масив із обходу дерева двійкового пошуку за порядком
sortArray(int* arr, int n)	int*	сортування масиву
getNodeByValue(Node* root, int value)	Node*	пошук елементу в дереві за ключем
IsAnElementOfAnArray(vector<int> arr, int element)	bool	перевірка, чи належить елемент масиву
insert(Node** head, int value)	void	Функція для створення нової печери
minValueNode (Node* node)	Node*	повертає вузол з мінімальним значенням ключа
deleteNode (Node* root, int key)	struct Node*	ця функція видаляє ключ і повертає новий корінь

Допоміжні величини:

Назва змінної	Тип змінної	Значення змінної
lower	int	нижній індекс масиву
upper	int	верхній індекс масиву
size	int	розмір масиву
middle	int	рекурсивний випадок (пов'язано з розміром масиву)
node	Node*	покажчик на корінь піддерева
arr	int*	покажчик на масив
n	int	розмір масиву
sortArray	int*	покажчик на відсортований масив
root	Node*	покажчик на дерево
newArr	vector<int>	масив утворений з дерева
newRoot	Node*	збалансоване дерево
temp	int	кількість злитків золота
varik	int	обвалилась печера чи ні

Табл 1. Таблиця величин

Блок-схеми

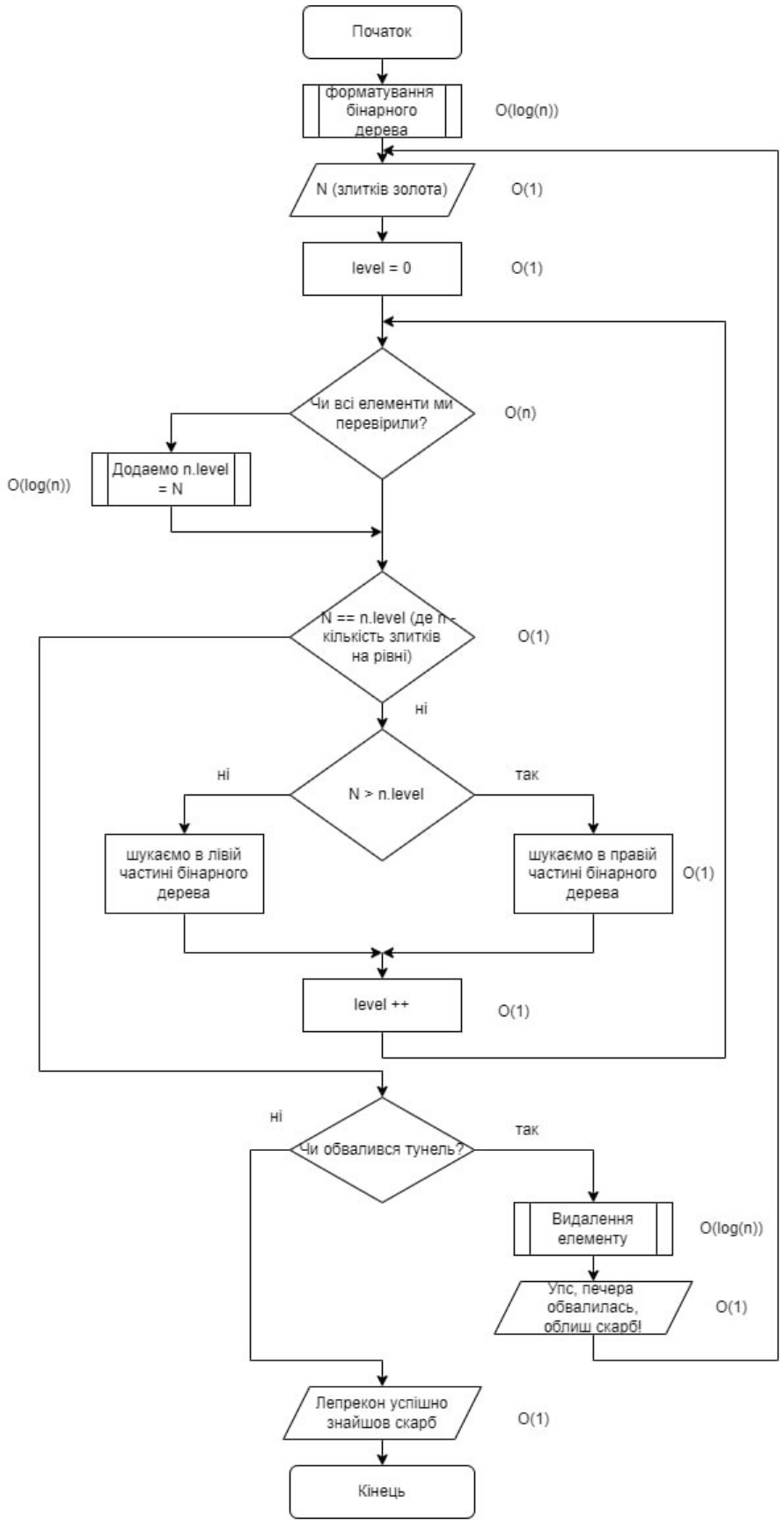


Рис. 1. Блок-схема пошуку елементу в бінарному дереві

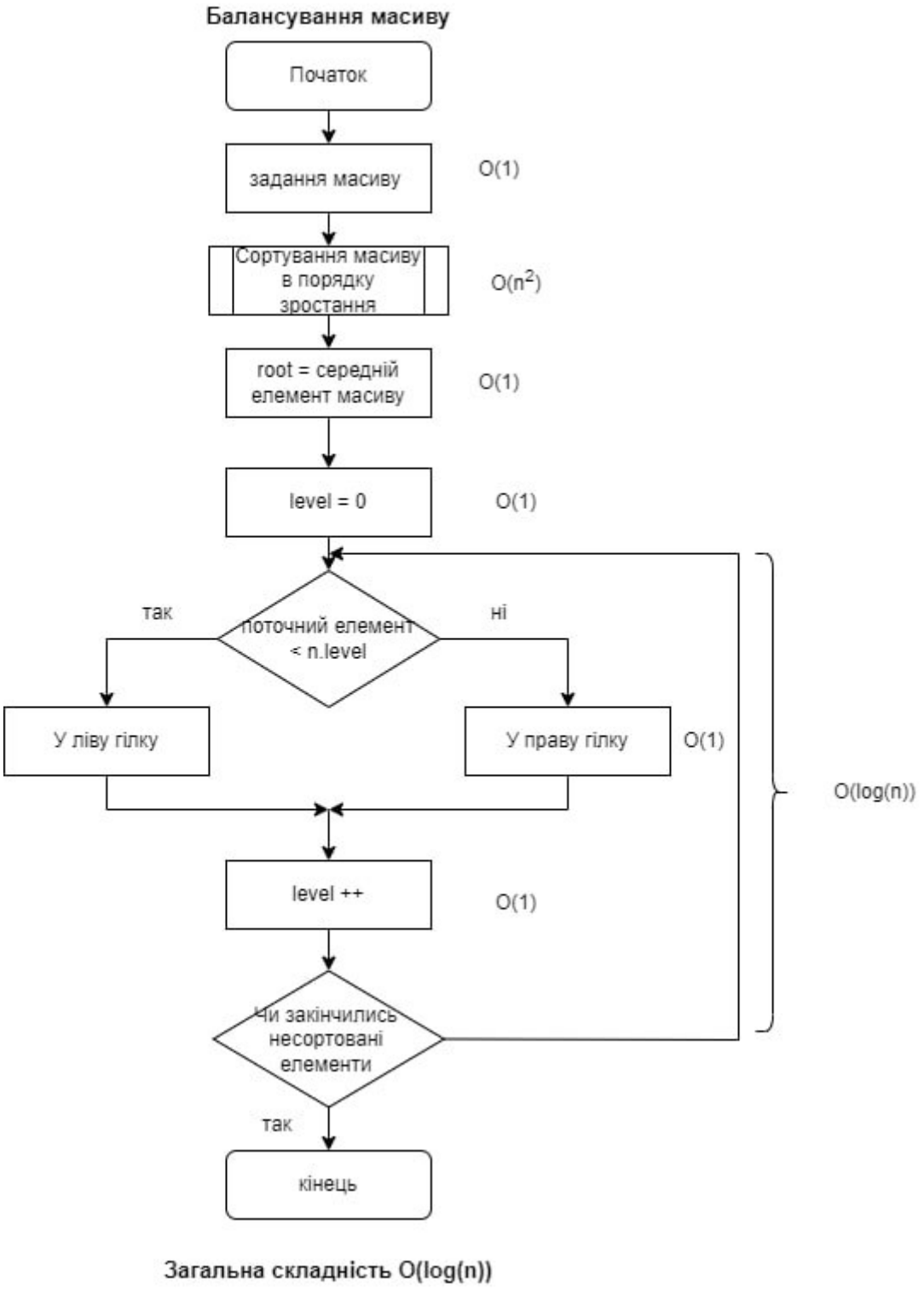


Рис. 2. Блок-схема балансування дерева

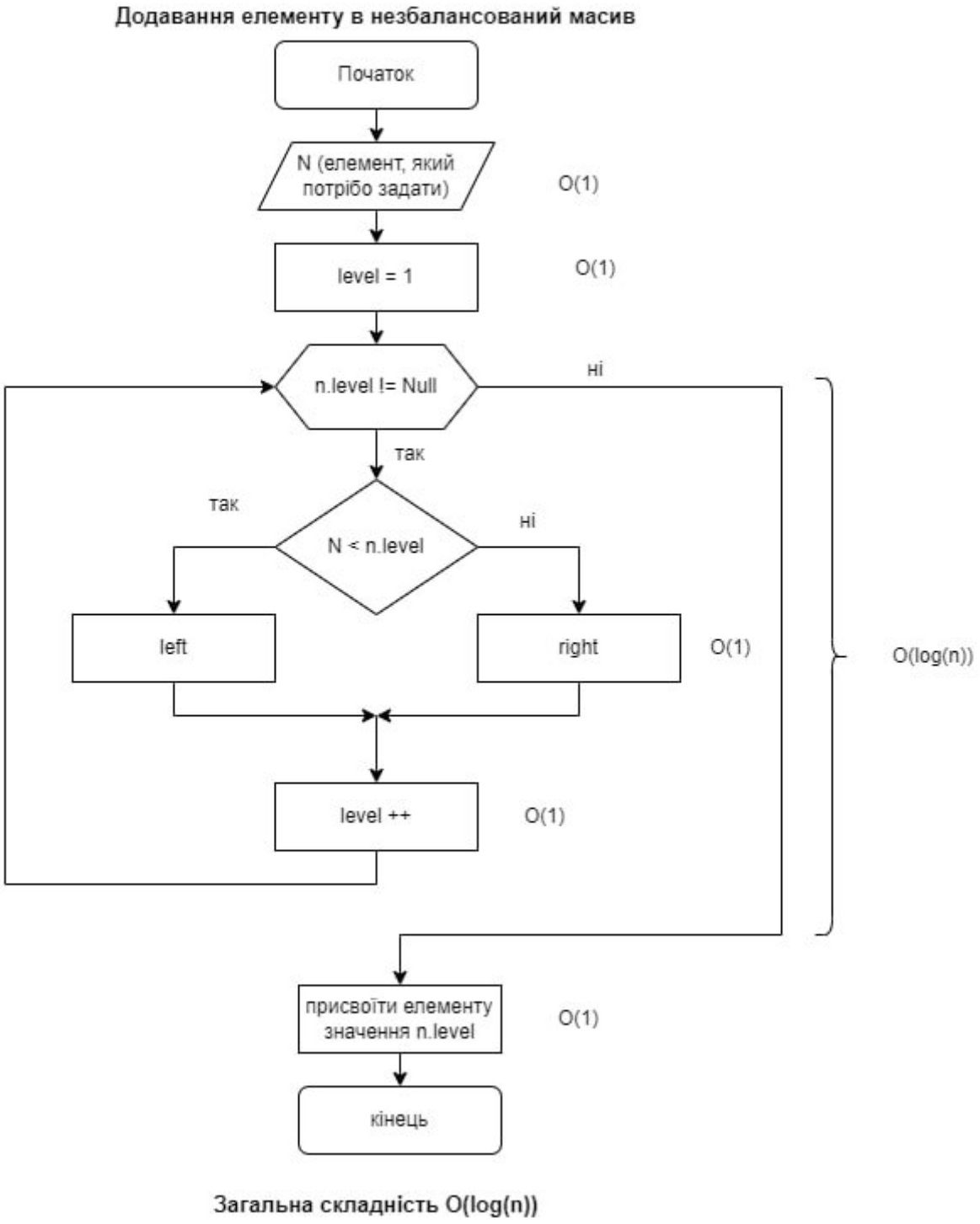


Рис. 3. Блок-схема додавання елементу в звичайне двійкове дерево

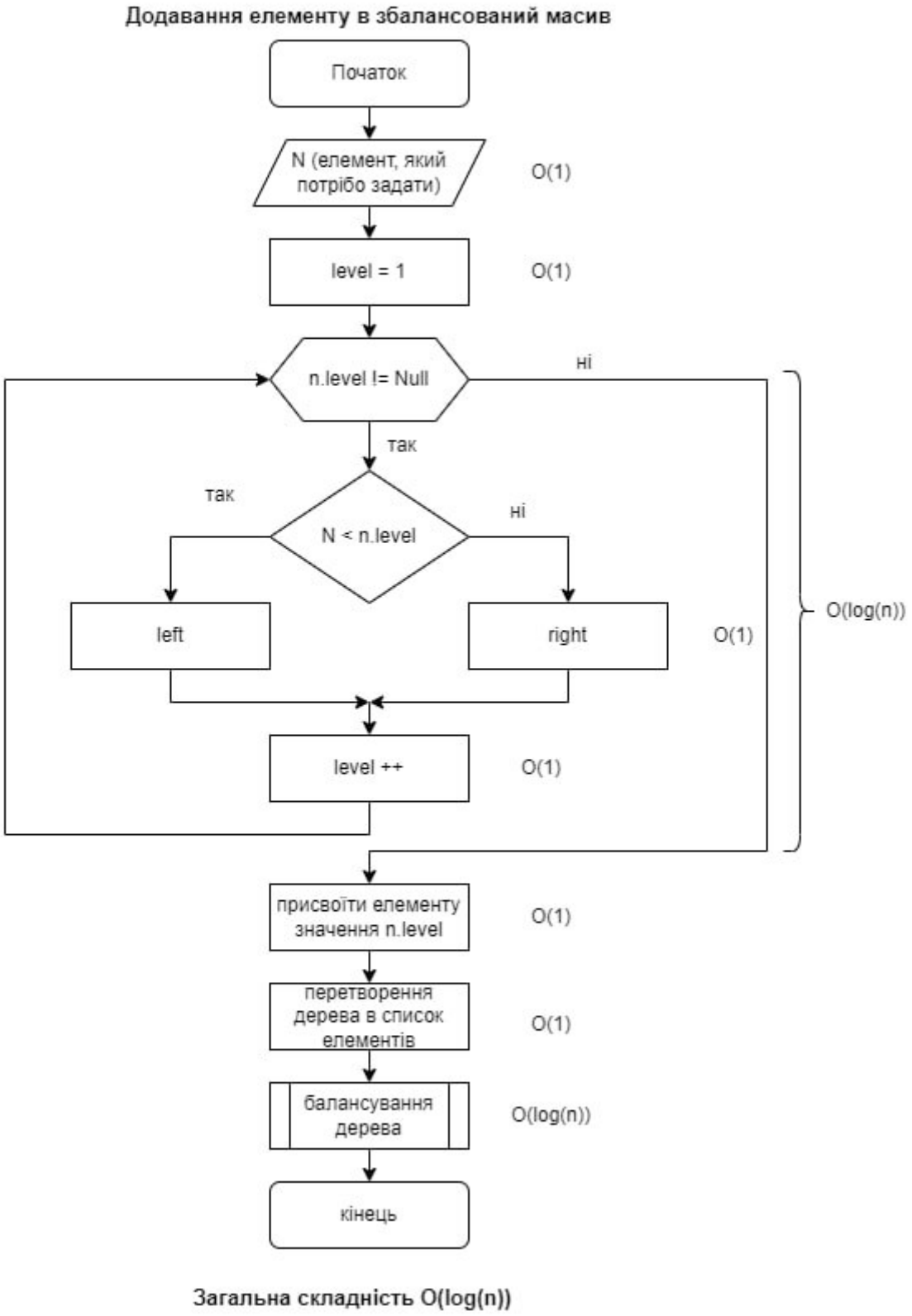


Рис. 4. Блок-схема додавання елементу в збалансоване двійкове дерево

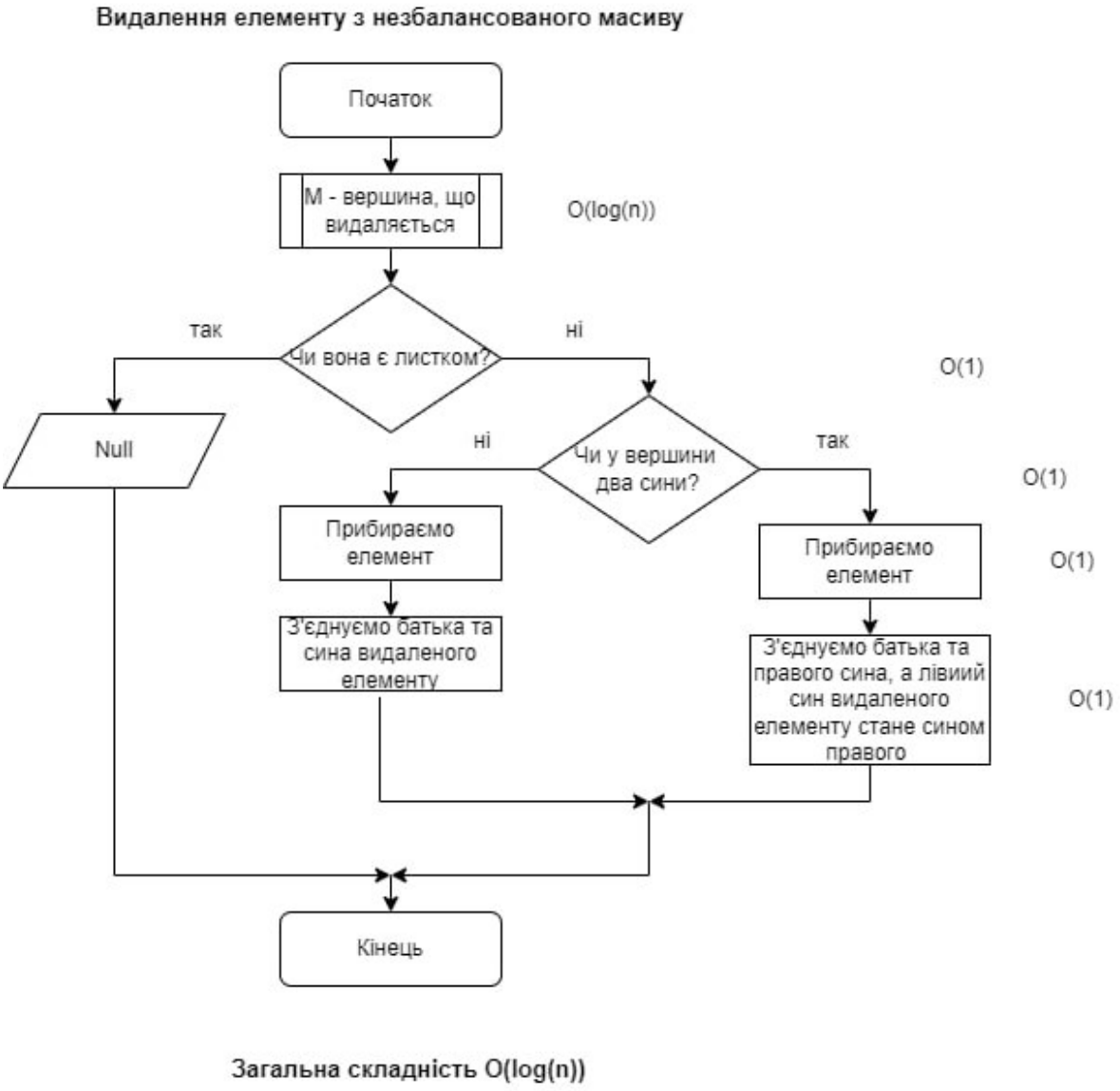


Рис. 5. Блок-схема видалення елементу зі звичайного двійкового дерева

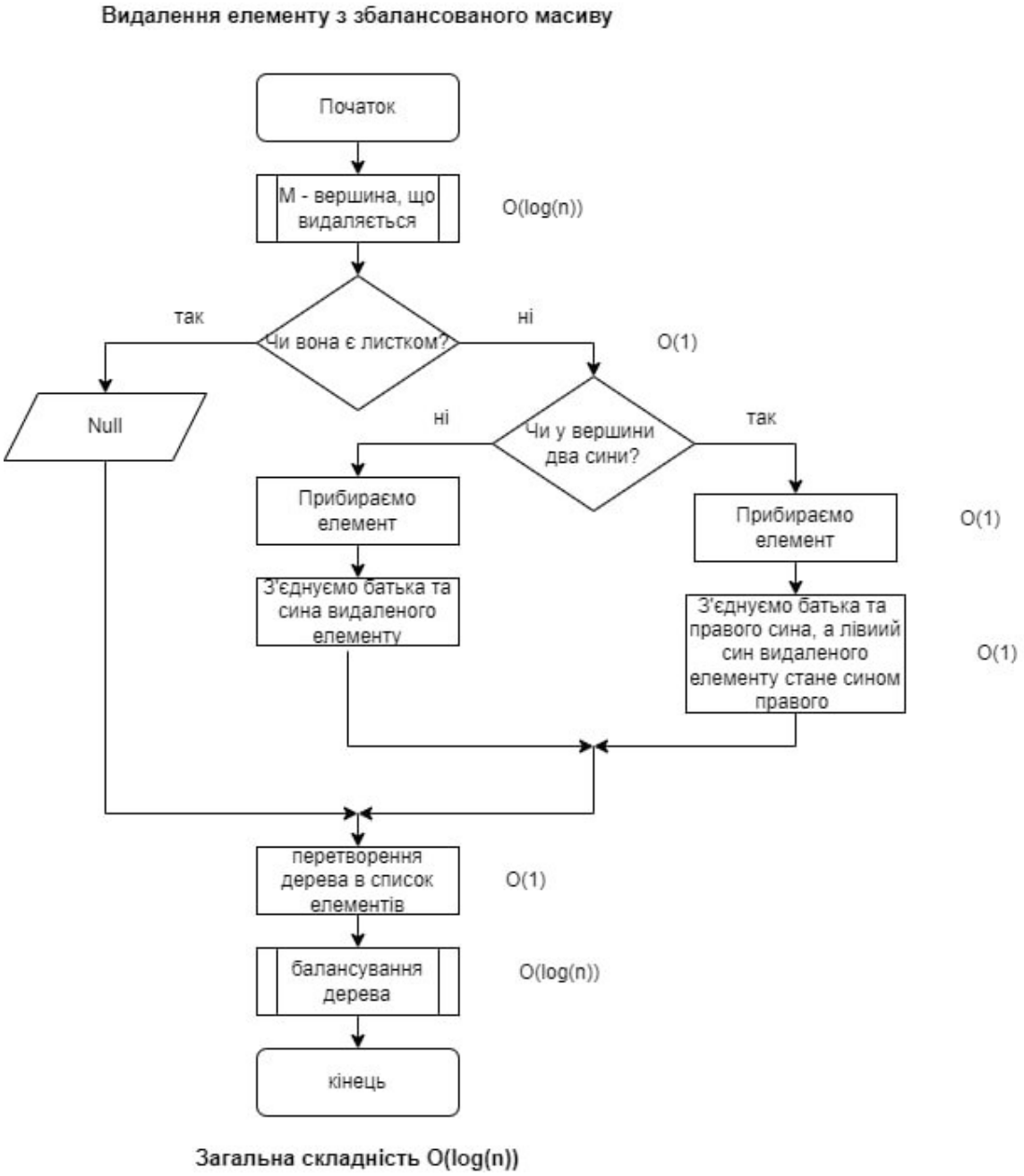


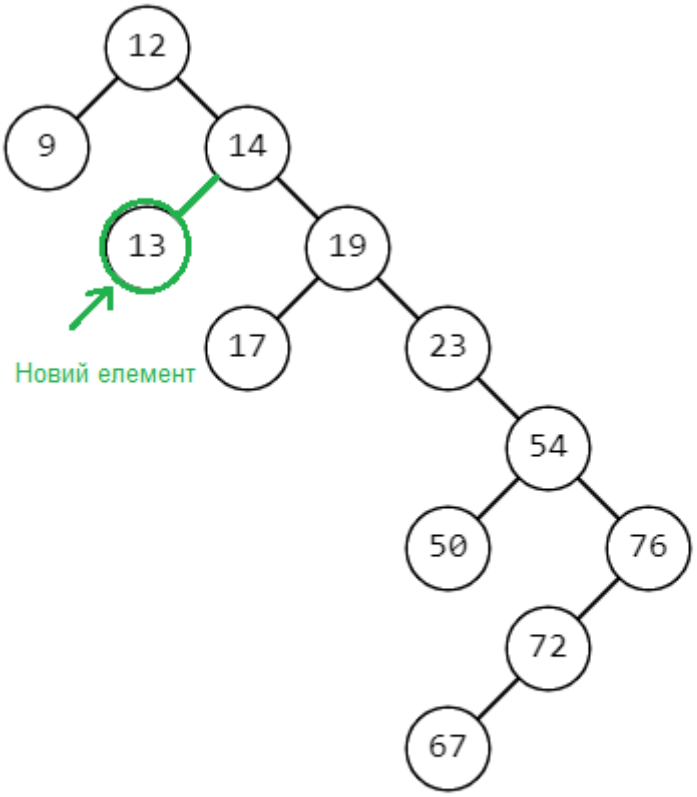
Рис. 6. Блок-схема видалення елементу зі збалансованого двійкового дерева

Зміни, що відбуваються з учасниками при балансуванні

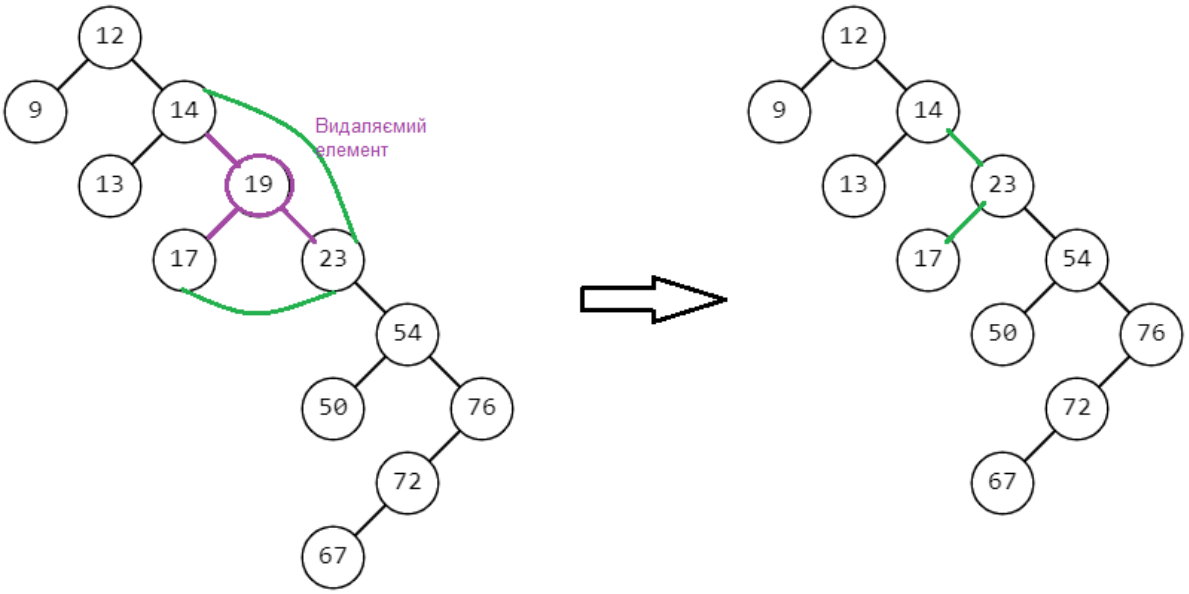
При балансуванні початковий граф повністю змінює свою структуру, тобто початковий список ребер кардинально відрізняється від вихідного. Відповідно до умови задачі, при балансуванні або перебалансуванні у нас фактично змінюється структура печер: зникають старі печери, викопуються нові і поєднуються новими тунелями. Такі операції допустимі, так, як за умовою наші печери магічні, і можуть з’являтися або зникати безслідно. Отже, обрана структура даних підходить для розв’язку посталеної задачі.

Графи

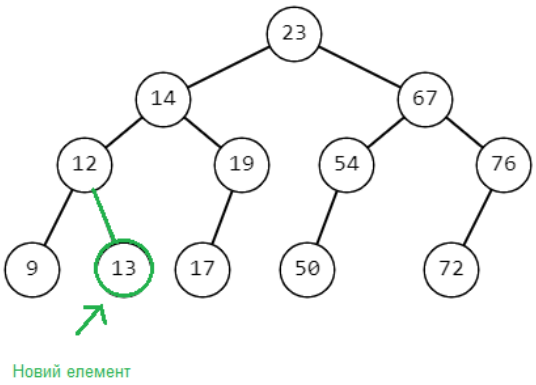
Додавання елементу (звичайне бінарне дерево)

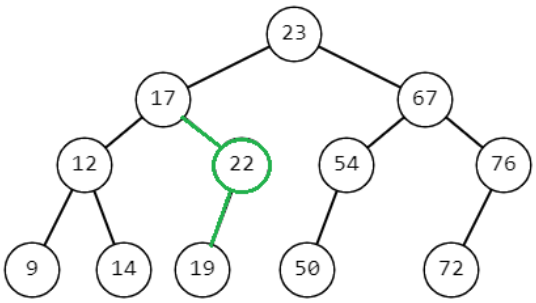


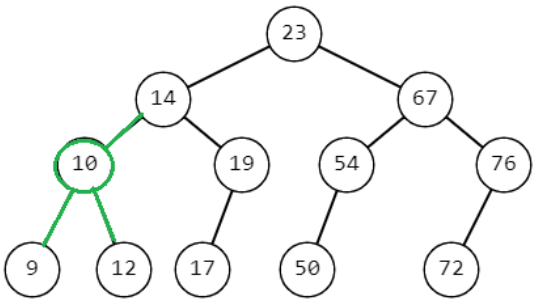
Видалення елементу (звичайне бінарне дерево)



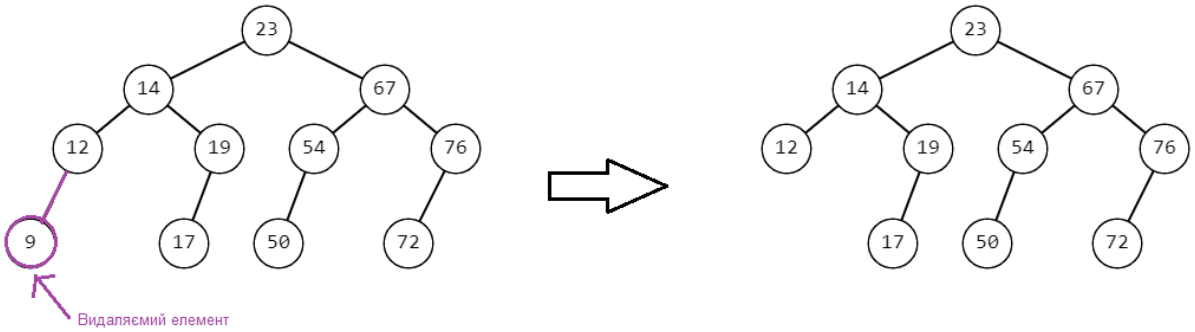
Додавання елементу (збалансоване)

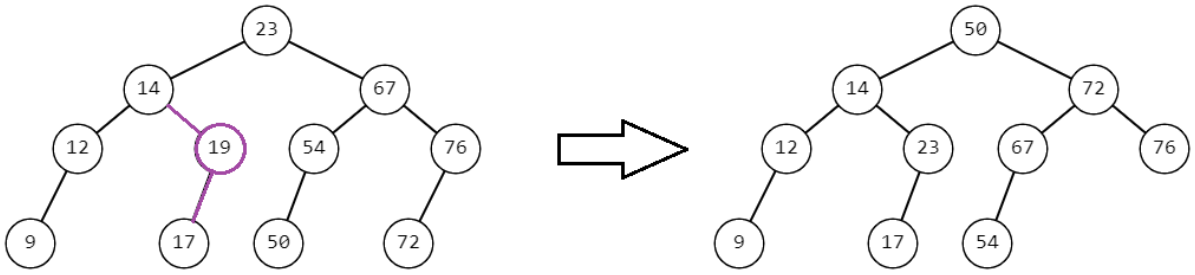


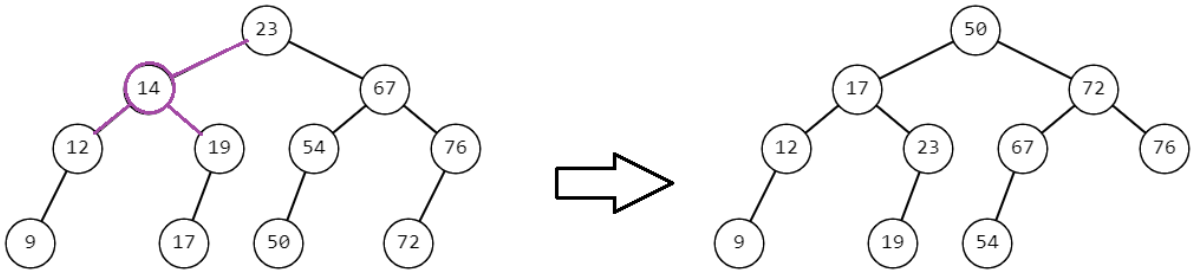




Видалення елементу (збалансоване)







Лістінг програми:

Реалізація збалансованого двійкового дерева

```
#include <iostream>
#include <windows.h>
#include <vector>
using namespace std;
#define CMP_EQ(a, b) ((a) == (b))
#define CMP_LT(a, b) ((a) < (b))
#define CMP_GT(a, b) ((a) > (b))
class Node
{
public:
   Node* left; //a pointer to the root of the left subtree
   Node* right; //a pointer to the root of the right subtree
   int key;
   Node(int key) : key(key) { }
};
// build returns a pointer to the root Node of the sub-tree
// lower is the lower index of the array
// upper is the upper index of the array
Node* build(const vector<int>& arr, int lower, int upper)
{
   int size = upper - lower + 1;
   // base case: array of size zero
   if (size <= 0) return NULL;
```

```
    // recursive case
    int middle = size / 2 + lower;
    // make sure you add the offset of lower
    Node* subtreeRoot = new Node(arr[middle]);
    subtreeRoot->left = build(arr, lower, middle - 1);
    subtreeRoot->right = build(arr, middle + 1, upper);
    return subtreeRoot;
}
// For convenience, In-order printing a tree
void printTree(Node* root, const char* dir, int level)
{
    if (root)
    {
        cout << "рівень " << level << " " << dir << " =\t" << root->key << endl;
        printTree(root->left, "зліва", level + 1);
        printTree(root->right, "справа", level + 1);
    }
}
// For convenience, we create a right skewed BST.
// This BST will later be balanced.
Node* buildBSTfromSortedArr(int* arr, int n)
{
    if (n == 0) return nullptr;
    Node* root = new Node(arr[0]);
    Node* cur = root;
    for (int i = 1; i < n; i++){
```

```
        cur->right = new Node(arr[i]);
        cur = cur->right;
    }
    return root;
}
// Convenience function for creating an array from BST in-order traversal
vector<int> createArrayFromTree(Node* root, vector<int> arr = vector<int>())
{
    if (root == nullptr) return arr;
    arr = createArrayFromTree(root->left, arr);
    arr.push_back(root->key);
    arr = createArrayFromTree(root->right, arr);
    return arr;
}
// Function for sorting an array
int* sortArray(int* arr, int n)
{
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1]) swap(arr[j], arr[j + 1]);
    return arr;
}
// Function for searching an element in tree by it's key value
Node* getNodeByValue(Node* root, int value)
{
    while (root)
```

```
    {
        if (CMP_GT(root->key, value))
        {
            root = root->left;
            continue;
        }
        else if (CMP_LT(root->key, value))
        {
            root = root->right;
 continue;
        }
        else return root;
    }
    return NULL;
}

bool IsAnElementOfAnArray(vector<int> arr, int element)
{
    for (int i = 0; i < arr.size(); i++)
        if (arr[i] == element) return true;
    return false;
}
Node* getFreeNode(int key)
{
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->left = tmp->right = NULL;
```

```
    tmp->key = key;
    return tmp;
}
void insert(Node** head, int value)
{
    Node* tmp = NULL;
    Node* ins = NULL;
    if (*head == NULL)
    {
        *head = getFreeNode(value);
        return;
    }
    tmp = *head;
    while (tmp)
    {
        if (CMP_GT(value, tmp->key))
        {
            if (tmp->right)
            {
                tmp = tmp->right;
                continue;
            }
            else
            {
                tmp->right = getFreeNode(value);
                return;
```

```
            }
        }
        else if (CMP_LT(value, tmp->key))
        {
            if (tmp->left)
            {
                tmp = tmp->left;
                continue;
            }
            else
            {
                tmp->left = getFreeNode(value);
                return;
            }
        }
        else exit(2);
    }
}
/* Given a non-empty binary search tree, return the node
with minimum key value found in that tree. Note that the
entire tree does not need to be searched. */
Node* minValueNode(Node* node)
{
    Node* current = node;
    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
```

```
        current = current->left;
    return current;
}
/* Given a binary search tree and a key, this function
deletes the key and returns the new root */
struct Node* deleteNode(Node* root, int key)
{
    // If the key to be deleted is
    // smaller than the root's
    // key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    // If the key to be deleted is
    // greater than the root's
    // key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    // if key is same as root's key, then This is the node
    // to be deleted
    else {
        // node has no child
        if (root->left == NULL and root->right == NULL)
            return NULL;
        // node with only one child or no child
        else if (root->left == NULL) {
            Node* temp = root->right;
```

```
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        // node with two children: Get the inorder successor
        // (smallest in the right subtree)
        Node* temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
void callPrintTree(Node* newRoot)
{
    cout << "\n--------Карта скарбів-------" << endl << endl;
    printTree(newRoot, "початок", 0);
    cout << "\n----------------------------" << endl;
}
int main()
{
```

```
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
cout << "\t * -------- * Комп'ютерний практикум 5 * -------- *\n";
cout << "\t\t\t Бригада 10. Варіант 1  \n";
cout << "\t     Реалізація збалансованого бінарного дерева\n\n";
int arr[] = { 12, 9, 14, 19, 17, 23, 54, 50, 76, 72, 67 };
int n = 11, temp, tempRight, tempLeft, varik;
string temp0;
// Sort an array
int* sortArr = sortArray(arr, n);

// Build a right skewed tree
Node* root = buildBSTfromSortedArr(sortArr, n);

// Get the array from the tree
vector<int> newArr = createArrayFromTree(root);

// Make the skewed tree balanced
Node* newRoot = build(newArr, 0, n - 1);

// In-order printing to verify
callPrintTree(newRoot);

while (true)
{
    // Finding an element in tree
```

```
  cout << "\nЛеприкон хоче знайти N злитків золота\nN = ";
  cin >> temp0;
  if (atoi(temp0.c_str()) >= 1 && atoi(temp0.c_str()) <= 100) temp =
atoi(temp0.c_str());
  else
  {
    cout << "\n----------Помилка-----------" << endl;
    cout << "\nНемає такої кількості злитків" << endl;
    cout << "\n--------------------------" << endl;
    continue;
  }
  // If there is no such element in the tree
  if (!IsAnElementOfAnArray(newArr, temp))
  {
    insert(&newRoot, temp);
    newArr = createArrayFromTree(newRoot);
    newRoot = build(newArr, 0, newArr.size() - 1);
    cout << "\nТрішечки магії і з'являється золото!" << endl;
    callPrintTree(newRoot);
  }
  // Define wether the element is reached
  varik = rand() % 2;
  if (varik == 0)
  {
    cout << "\nУпс! Тунель до печери обвалився, швидше, облиш
золото, якщо хочеш жити!" << endl;
    deleteNode(newRoot, temp);
```

```
            newArr = createArrayFromTree(newRoot);
            newRoot = build(newArr, 0, newArr.size() - 1);
            callPrintTree(newRoot);
        }
        // If this element is reached
        else
        {
            Node* newTemp = getNodeByValue(newRoot, temp);
            // check on having the left brunch
            if (newTemp->left) tempLeft = newTemp->left->key;
            else tempLeft = 0;

            // check on having the right brunch
            if (newTemp->right) tempRight = newTemp->right->key;
            else tempRight = 0;
            cout << "\nУра! Леприкон знайшов " << newTemp->key << "
    злитків\n\tзліва від цієї печери " << tempLeft << " злитків\n\tсправа - "
    << tempRight << endl;
        }
    }
}
```

Реалізація не збалансованих кістякових дерев

```
#include <iostream>
#include <windows.h>
using namespace std;
#define CMP_EQ(a, b) ((a) == (b))
#define CMP_LT(a, b) ((a) < (b))
#define CMP_GT(a, b) ((a) > (b))
class Node
{
public:
    Node* left;
    Node* right;
    int key;
    Node(int key) : key(key) { }
};
Node* getFreeNode(int key)
{
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->left = tmp->right = NULL;
    tmp->key = key;
    return tmp;
}
void insert(Node** head, int value)
{
    Node* tmp = NULL;
    Node* ins = NULL;
```

```
if (*head == NULL)
{
    *head = getFreeNode(value);
    return;
}
tmp = *head;
while (tmp)
{
    if (CMP_GT(value, tmp->key))
    {
        if (tmp->right)
        {
            tmp = tmp->right;
            continue;
        }
        else
        {
            tmp->right = getFreeNode(value);
            return;
        }
    }
    else if (CMP_LT(value, tmp->key))
    {
        if (tmp->left)
        {
            tmp = tmp->left;
```

```
                continue;
            }
            else
            {
                tmp->left = getFreeNode(value);
                return;
        }
        }
        else exit(2);
    }
}

Node* getMaxNode(Node* root)
{
    while (root->right)
        root = root->right;
    return root;
}

Node* getNodeByValue(Node* root, int value)
{
    while (root)
    {
        if (CMP_GT(root->key, value))
        {
            root = root->left;
```

```
            continue;
        }
        else if (CMP_LT(root->key, value))
        {
            root = root->right;
            continue;
        }
        else return root;
    }
    return NULL;
}
Node* minValueNode(Node* node)
{
    Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}
struct Node* deleteNode(Node* root, int key)
{
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // node has no child
```

```
        if (root->left == NULL and root->right == NULL)
            return NULL;
        // node with only one child or no child
        else if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        // node with two children
        Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

void printTree(Node* root, const char* dir, int level)
{
    if (root)
    {
        cout << "рівень " << level << " " << dir << " =\t" << root->key << endl;
```

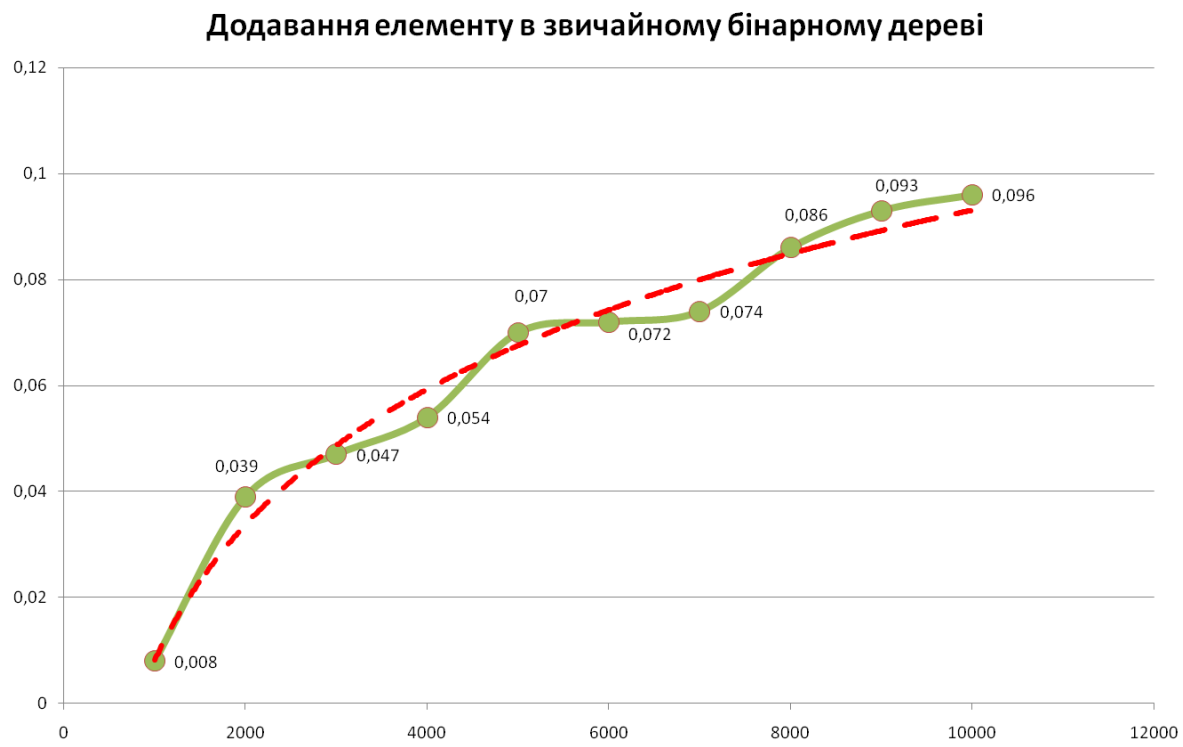
```
        printTree(root->left, "зліва", level + 1);
        printTree(root->right, "справа", level + 1);
    }
}

bool IsAnElementOfTree(Node* root, int n, int element, bool answer)
{
    if (root && !answer)
    {
        if (root->key == element)
        {
            answer = true;
            return answer;
        }
        answer = IsAnElementOfTree(root->left, n, element, answer);
        answer = IsAnElementOfTree(root->right, n, element, answer);
    }
    if (answer) return true;
    else return false;
}

void callPrintTree(Node* root)
{
    cout << "\n--------Карта скарбів-------" << endl << endl;
    printTree(root, "початок", 0);
    cout << "\n----------------------------" << endl;
```

```
}

void main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    cout << "\t * -------- * Комп'ютерний практикум 5 * -------- *\n";
    cout << "\t\t\t Бригада 10. Варіант 1  \n";
    cout << "\t\t      Реалізація бінарного дерева\n\n";

    int arr[] = { 12, 9, 14, 19, 17, 23, 54, 50, 76, 72, 67 };
    int n = 11, temp, tempRight, tempLeft, varik;
    bool answer = false;
    string temp0;
    Node* root = NULL;

    for (int i = 0; i < n; i++)
        insert(&root, arr[i]);
    callPrintTree(root);

    while (true)
    {
        // Finding an element in tree
        cout << "\nЛеприкон хоче знайти N злитків золота\nN = ";
        cin >> temp0;
```

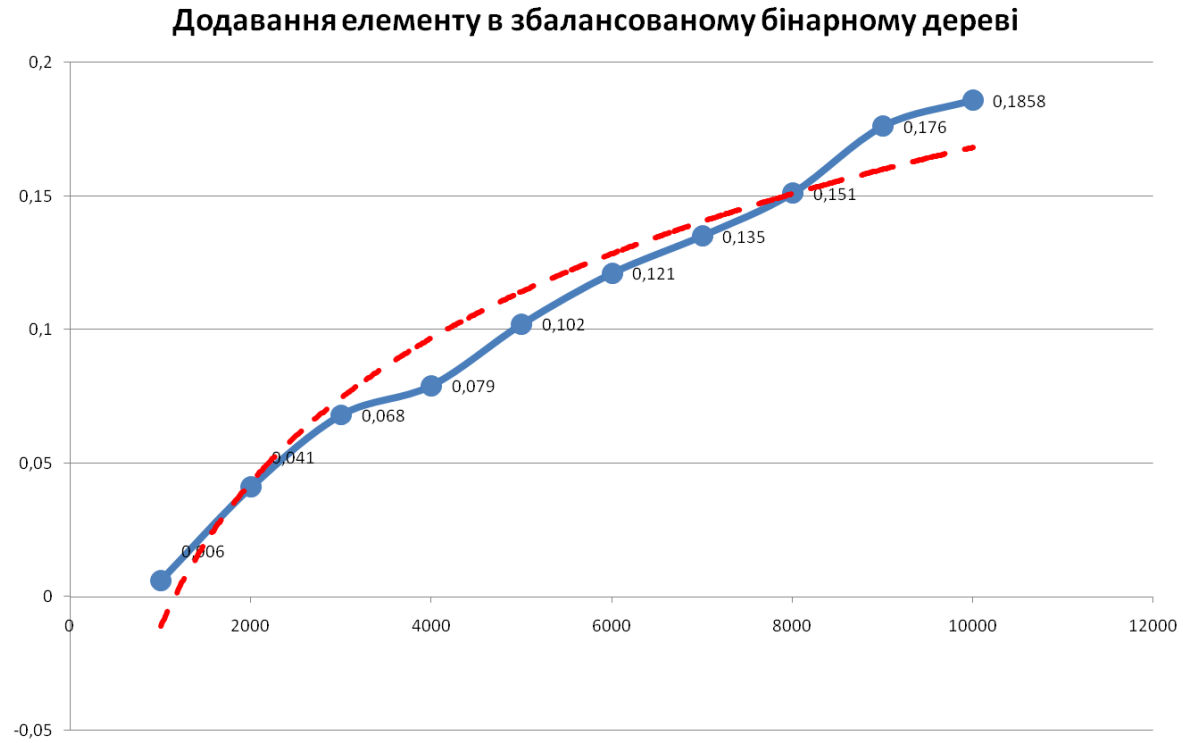
```
  if (atoi(temp0.c_str()) >= 1 && atoi(temp0.c_str()) <= 100) temp =
atoi(temp0.c_str());

  else

  {

     cout << "\n----------Помилка-----------" << endl;

     cout << "\nНемає такої кількості злитків" << endl;

     cout << "\n---------------------------" << endl;

     continue;

  }


  // If there is no such element in the tree

  if (!IsAnElementOfTree(root, n, temp, answer))

  {

     insert(&root, temp);

     n++;

     cout << "\nТрішечки магії і з'являється золото!" << endl;

     callPrintTree(root);

  }

  // Define wether the element is reached

  varik = rand() % 2;

  if (varik == 0)

  {

     cout << "\nУпс! Тунель до печери обвалився, швидше, облиш
золото, якщо хочеш жити!" << endl;

     deleteNode(root, temp);

     n--;

     callPrintTree(root);
```

```
        }
        // If this element is reached
        else
        {
            Node* newTemp = getNodeByValue(root, temp);

            // check on having the left brunch
            if (newTemp->left) tempLeft = newTemp->left->key;
            else tempLeft = 0;

            // check on having the right brunch
            if (newTemp->right) tempRight = newTemp->right->key;
            else tempRight = 0;
            cout << "\nУра! Леприкон знайшов " << newTemp->key << "
    злитків\n\tзліва від цієї печери " << tempLeft << " злитків\n\tсправа - "
    << tempRight << endl;
        }
    }
}
```

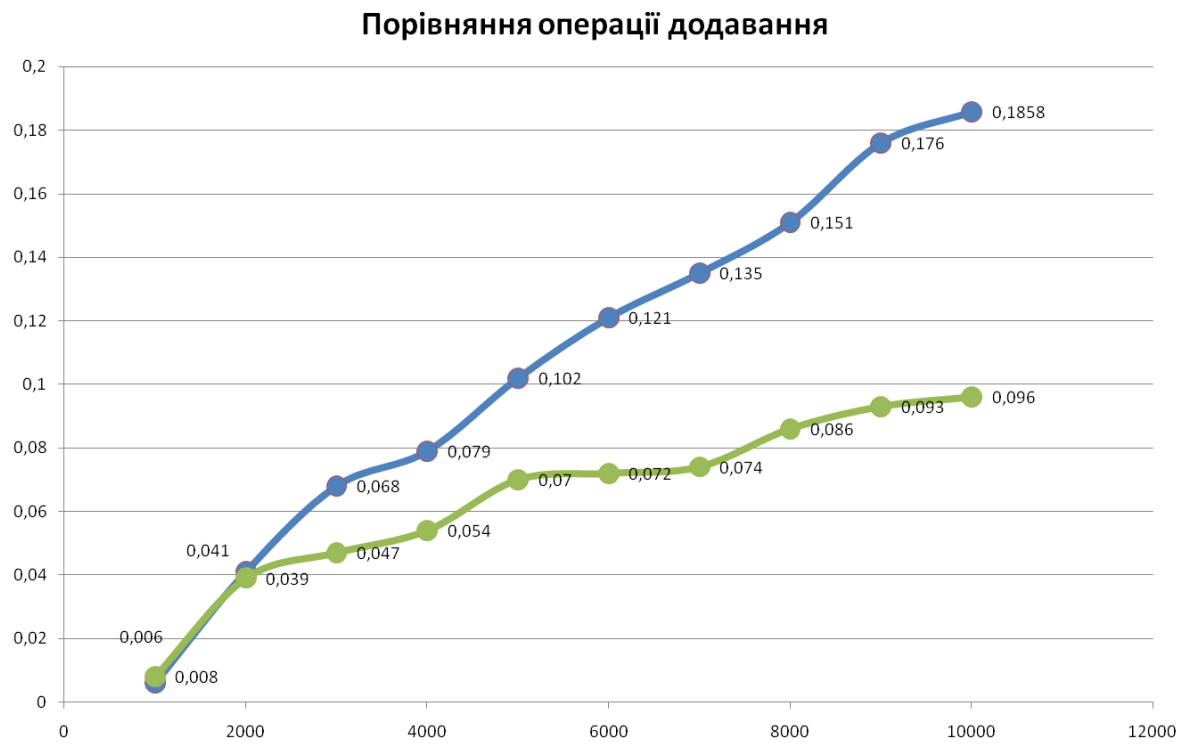
Перевірка обчислювальної складності програми:



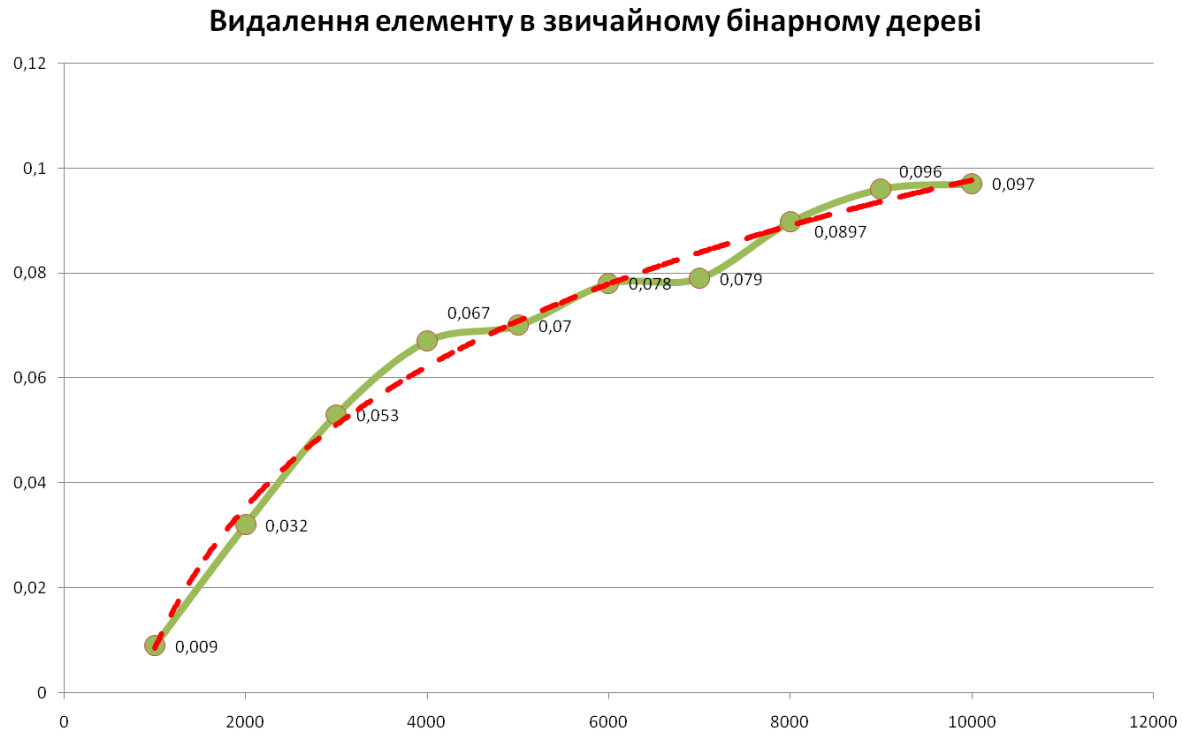
Діагр. 1. Додавання елементу у звичайному бінарному дереві



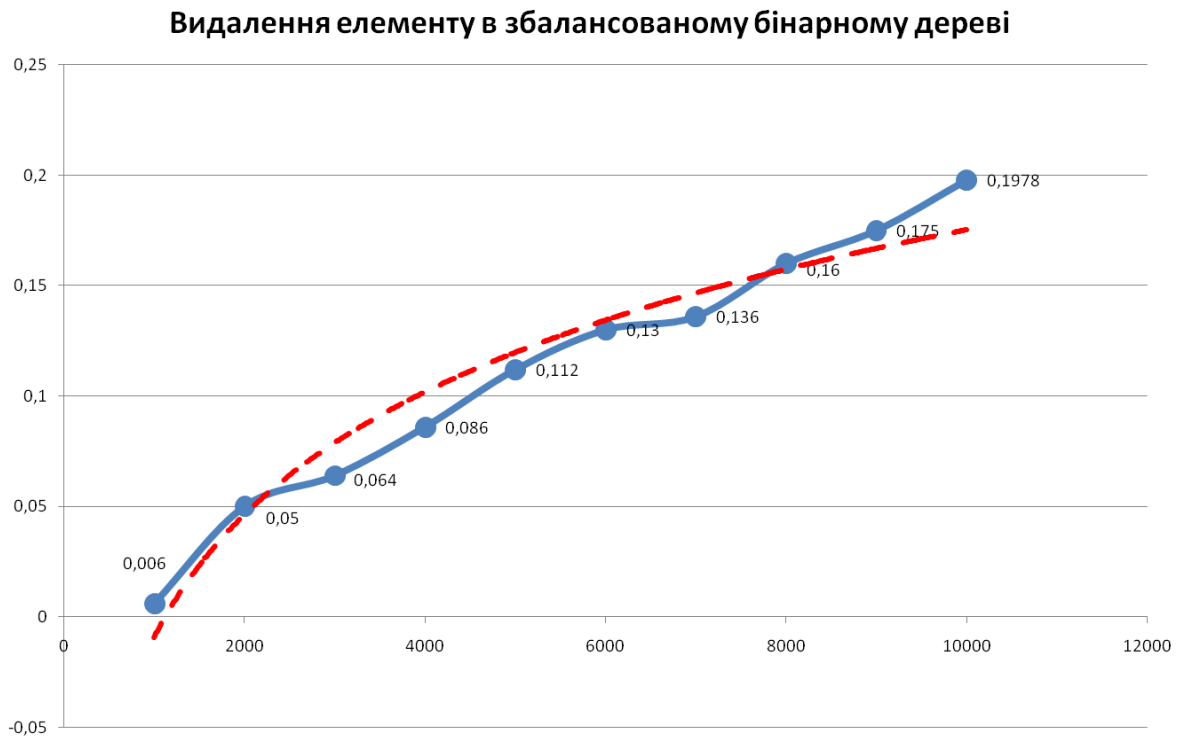
Діагр. 2. Додавання елементу у збалансованому бінарному дереві



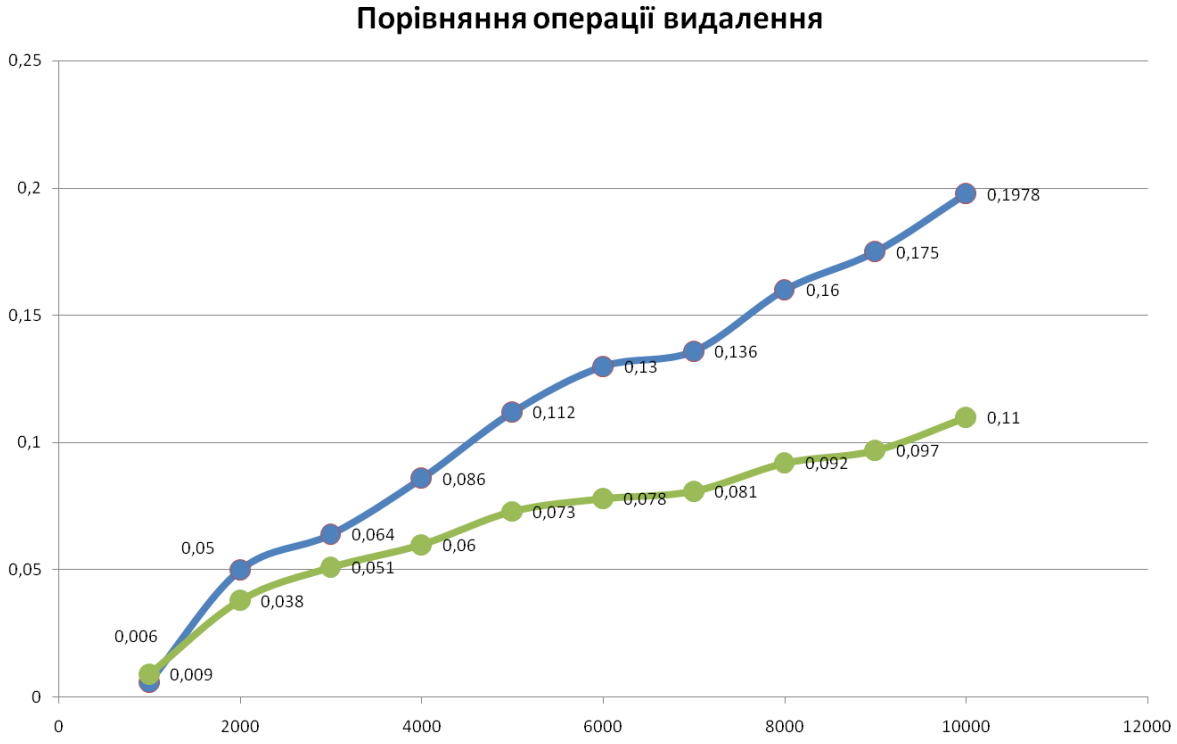
Діагр. 3. Аналіз обчислюваної складності функції додавання елементу в бінарне дерево



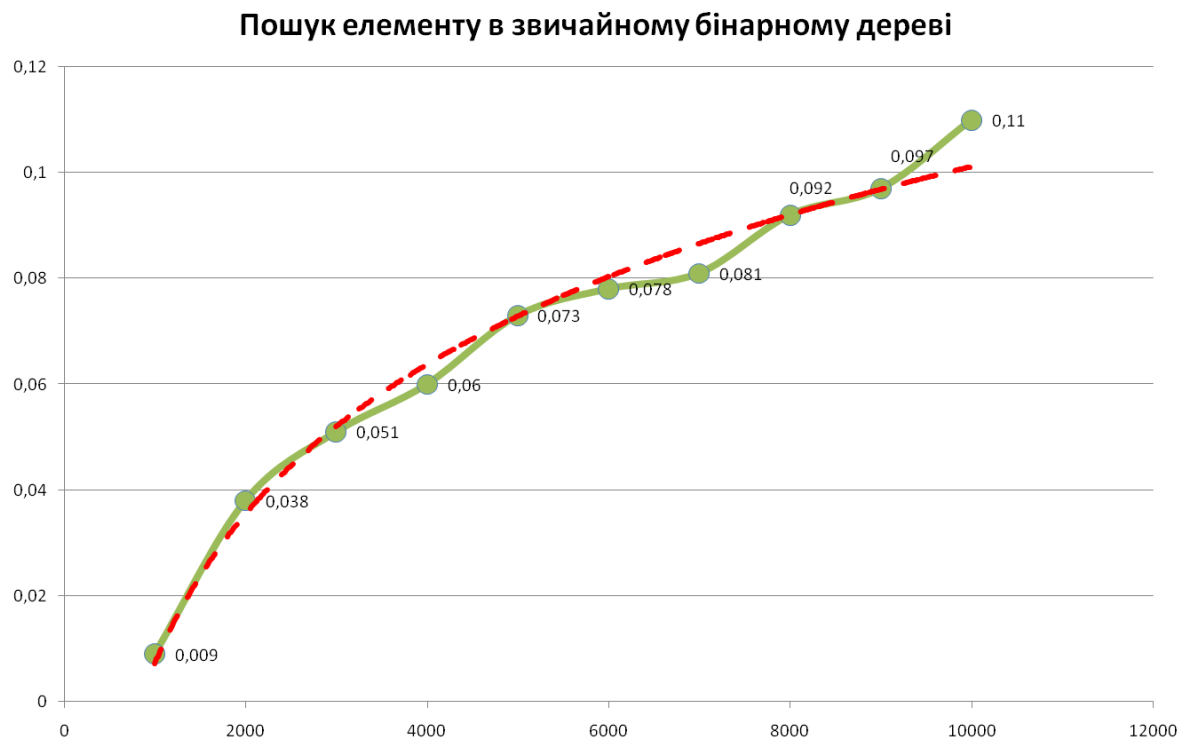
Діагр. 4. Видалення елементу у звичайному бінарному дереві



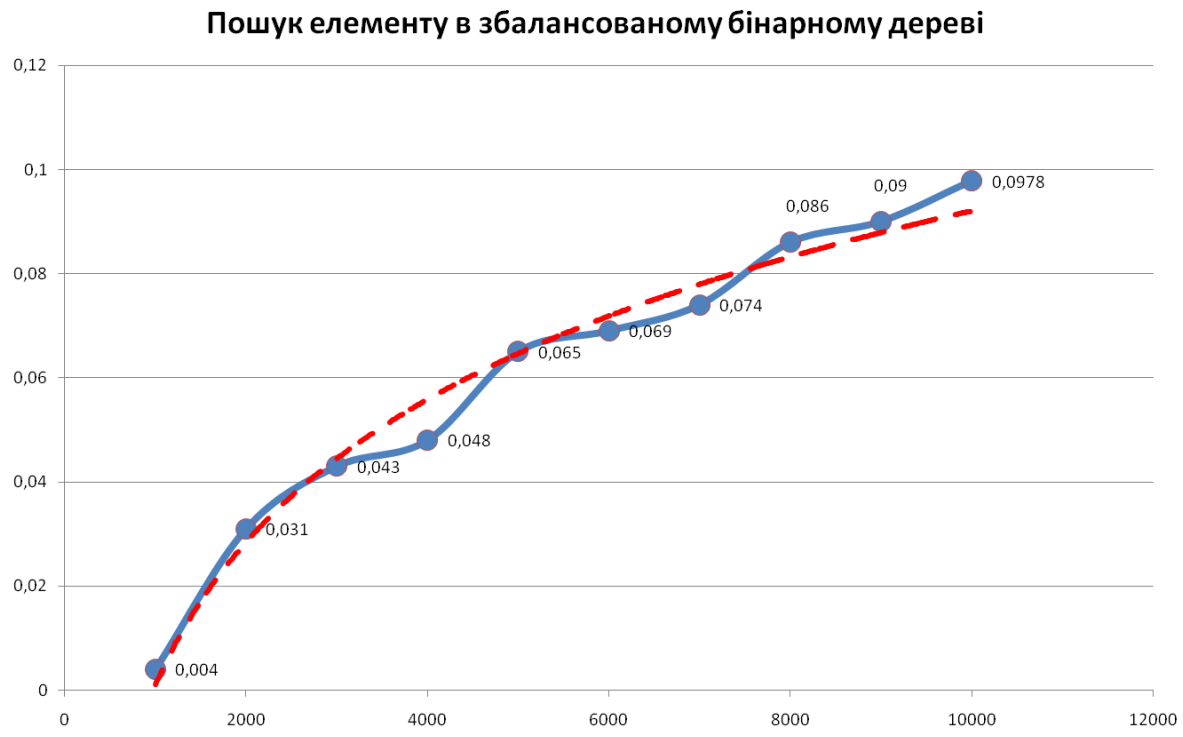
Діагр. 5. Видалення елементу у збалансованому бінарному дереві



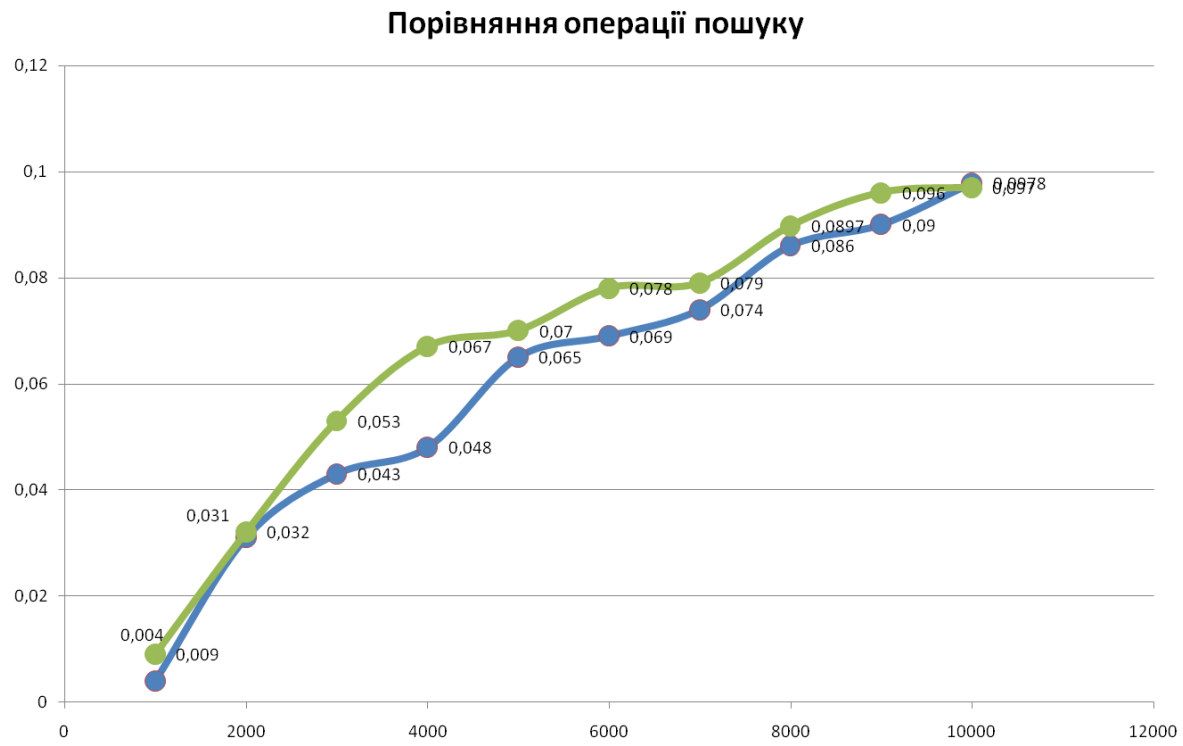
Діагр. 6. Аналіз обчислюваної складності функції видалення елементу з бінарного дерева



Діагр. 7. Пошук елементу у звичайному бінарному дереві



Діагр. 8. Пошук елементу у збалансованому бінарному дереві



Діагр. 9. Аналіз обчислюваної складності функції пошуку елемента в бінарному дереві

Червоний пунктирний графік – теоретична складність.
Синій/Зелений графік – реальна складність.

Перевірка обчислювальної складності операцій для бінарного дерева підтвердила аналітичну оцінку складності алгоритмів O(logn).

Фактори, що найбільше впливають на продуктивність

На швидкість виконання програми найбільше впливає висота дерева. Так, як у цій роботі розглянуто реалізацію збалансованого бінарного дерева, то операція пошуку для нього буде виконуватись найшвидше. Проте, операції додавання та видалення елементів зі збалансованого дерева можуть потребувати його перебалансування, що займає додатковий час. Через це, додавання та видалення елементів ефективніше реалізовує алгоритм з використанням звичайного (не збалансованого) бінарного дерева.

Скріншоти роботи програми:

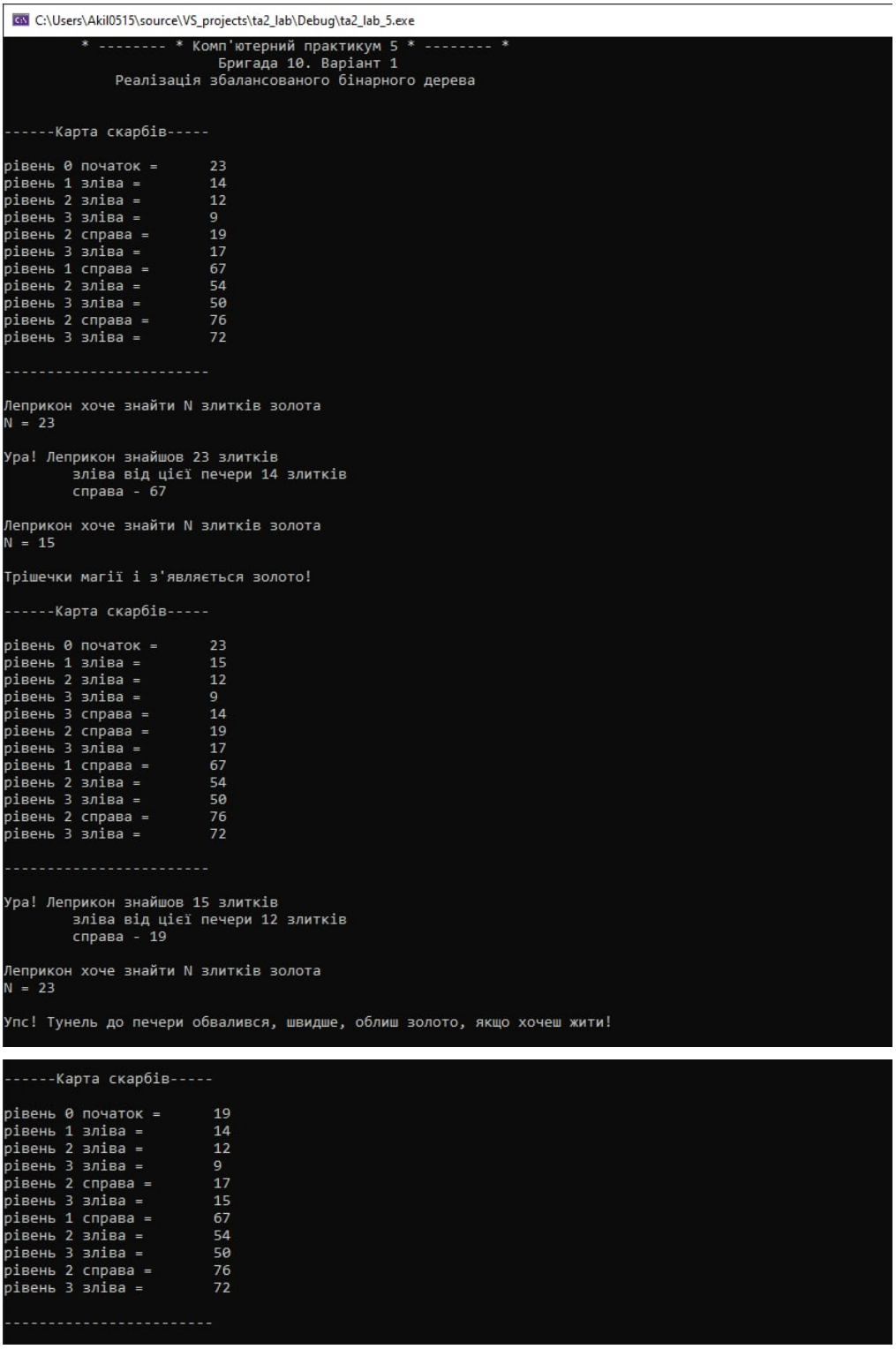


Рис. 7. Реалізація збалансованого двійкового дерева

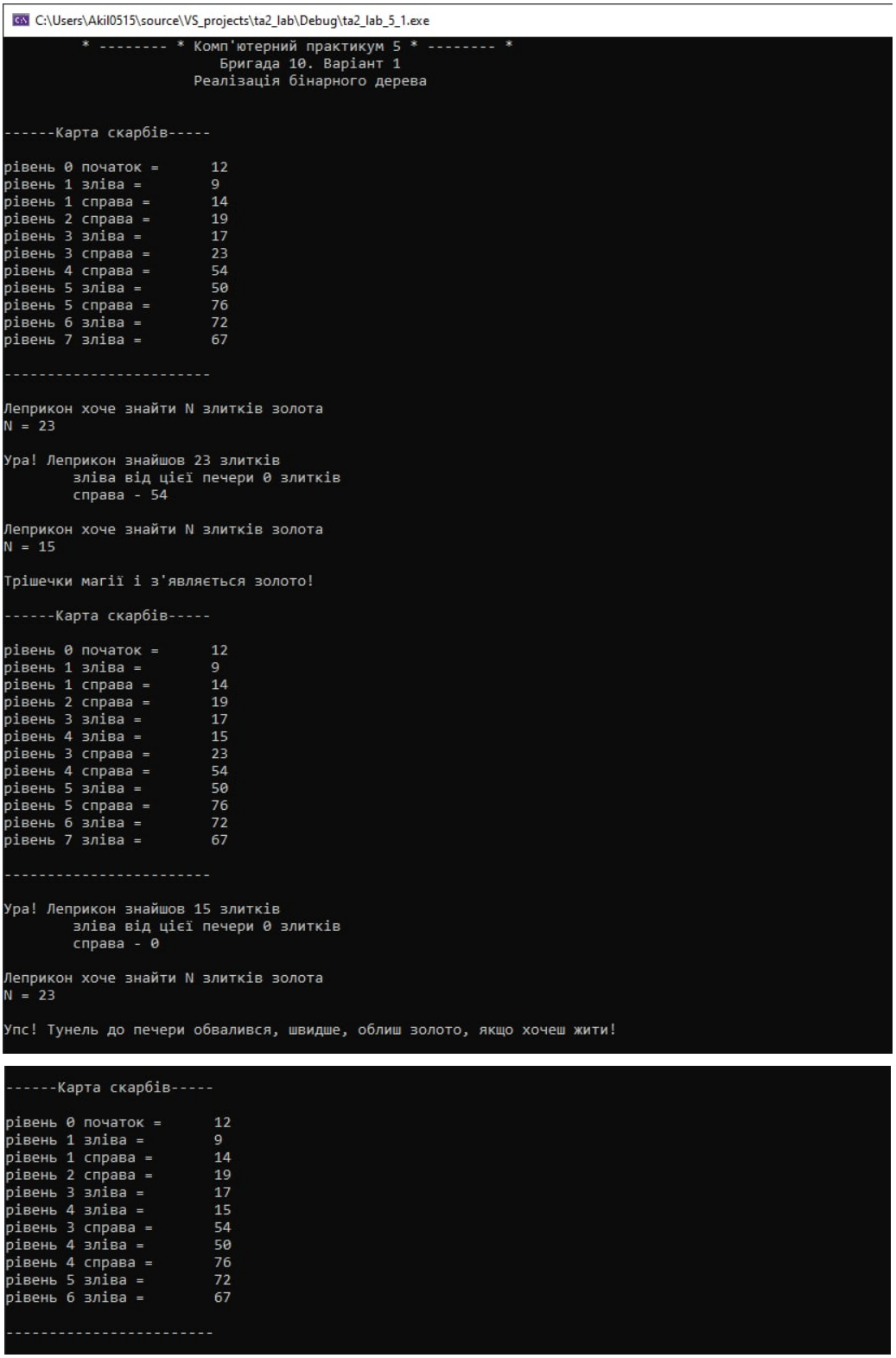


Рис. 8. Реалізація не збалансованого двійкового дерева

Перевірка правильності програми:

Вхідні дані	Результат	Призначення тесту
temp		
-1	Відомий	Перевірка, завершення некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
2	Відомий	Дані коректні, програма продовжує виконувати свою роботу
b	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
*	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
101	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки)
1.7 => результат буде 1	Відомий	Перевірка реакції на числа з плаваючою точкою(додавання повідомлень про помилки)

Табл. 2. Таблиця тестування програми

Висновок

Під час виконання лабораторної роботи ми дослідили характеристики продуктивності основних типів двійкових дерев пошуку, розглянули різні реалізації до поставленої задачі, зробили аналіз перевірки практичної складності різних операцій з бінарними деревами.

Відповіді на контрольні запитання:

1. *Чим відрізняються дерева бінарного пошуку та для чого вони використовуються?*

Бінарне дерево пошуку — двійкове дерево, в якому кожній вершині x зіставлене певне значення val[x]. При цьому такі значення повинні задовольняти умові впорядкованості:

– нехай x — довільна вершина двійкового дерева пошуку.

– Якщо вершина y знаходиться в лівому піддереві вершини x, то $val[y] \leq val[x]$.

– Якщо y знаходиться у правому піддереві x, то $val[y] \geq val[x]$.

Таке структурування дозволяє надрукувати усі значення у зростаючому порядку за допомогою простого алгоритму центрованого обходу дерева.

Властивості, які відрізняють:

– Всі вузли лівого піддерева менше кореневого вузла.

– Всі вузли правого піддерева більше кореневого вузла.

– Обидва піддерева кожного вузла також є BST, тобто вони мають дві вищевказані властивості.

2. *Перерахуйте та проілюструйте послідовність дій при додаванні вузла в бінарне дерево пошуку.*

Дерева змінної структури, тобто такі, кількість елементів яких під час роботи певної програми може збільшуватися або зменшуватися, можна використовувати, зокрема, при створенні частотного словника. Задача створення частотного словника формулюється так: потрібно визначити, скільки разів зустрічається кожне слово в заданій послідовності слів. Послідовність слів можна зобразити у вигляді дерева бінарного пошуку. В інформаційних полях вузла такого дерева зберігатиметься слово і кількість його повторень. У разі виявлення в тексті чергового слова дерево переглядають, починаючи з кореневого вузла. Якщо слово в дереві

знайдене, то лічильник його повторень збільшується. Але якщо слово в дереві не знайдене, воно включається в дерево із значенням лічильника, що дорівнює 1.

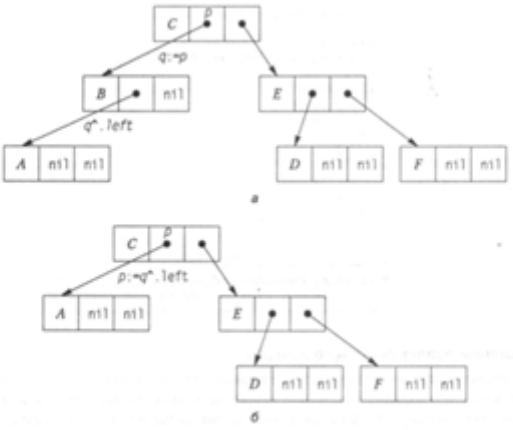
3. Перерахуйте та проілюструйте послідовність дій при видаленні вузла з бінарного дерева пошуку.

Можливі три випадки видалення вузла бінарного дерева пошуку :

- вузол, що видаляється, є листком дерева;
- вузол, що видаляється, має одного нащадка;
- вузол, що видаляється, має двох нащадків.

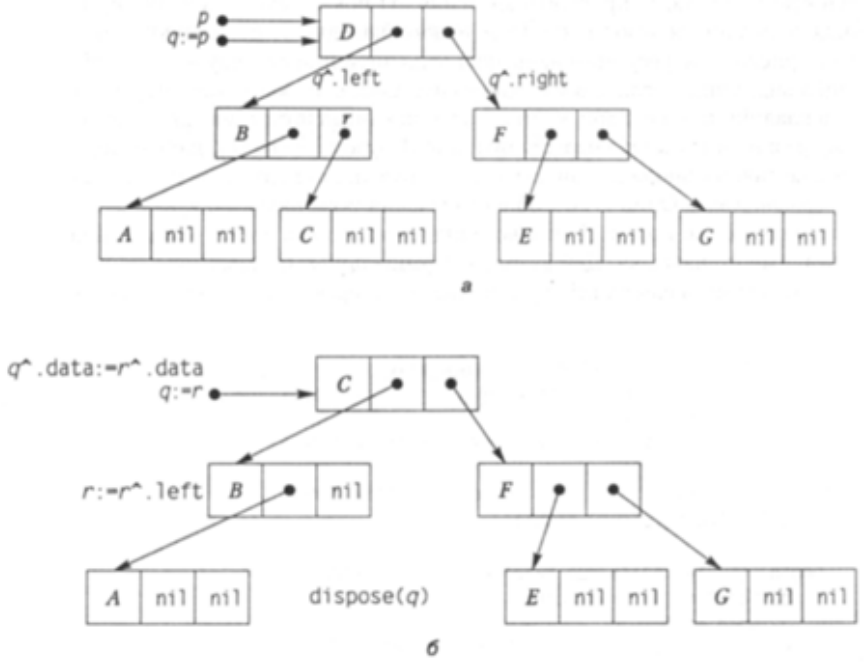
Задача розв'язується найпростішим способом тоді, коли вузол, що видаляється, є листком дерева. У такому разі слід звільнити ділянку динамічної пам'яті, яку цей вузол займав, та присвоїти значення nul покажчикові на даний вузол.

Якщо видаляється вузол x, який має одного нащадка, то покажчику на цей вузол слід присвоїти адресу його нащадка і звільнити пам'ять, яку вузол x займав.



Якщо видаляється вузол x, який має двох нащадків, на місце x слід переставити інший вузол дерева так, щоб не порушувалася властивість впорядкованості ключів. Вузол, що переставляється, називається *термінальним*. Один зі способів визначення термінального вузла полягає у виконанні спуску по правій гілці лівого піддерева x доти, доки не буде знайдено вузла без правого нащадка. Цей вузол і є термінальним, його значення може бути записане до вузла x без порушення впорядкованості

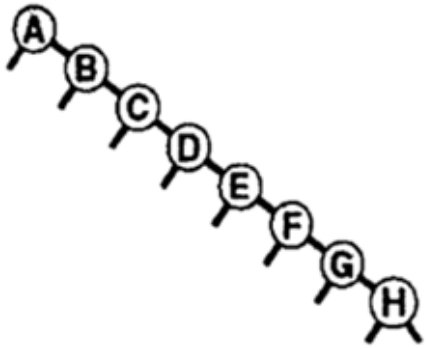
ключів. Сам термінальний вузол має бути видалений після копіювання його значення до вузла x.

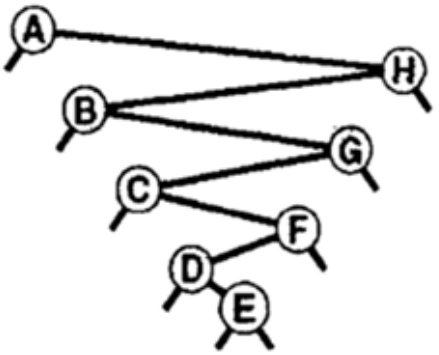


4. *Прокоментуйте характеристики продуктивності дерев бінарного пошуку на конкретних прикладах.*

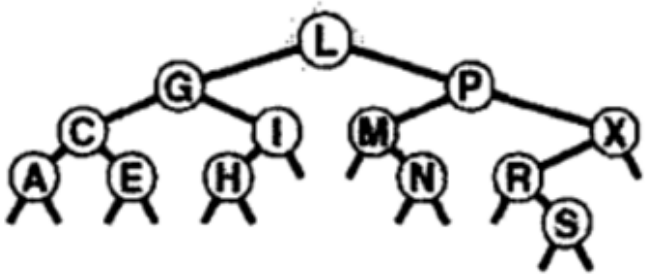
Час виконання алгоритмів обробки BST-дерев залежить від форм дерев. В кращому разі дерево може бути повністю збалансованим і містити приблизно logN вузлів між коренем і кожним із зовнішніх вузлів, але у гіршому разі в кожний з шляхів пошуку може містити N вузлів. Зокрема, довжина шляху і висота бінарних дерев безпосередньо пов'язані з витратами на пошук в BST-деревах. Висота визначає вартість пошуку у гіршому разі, довжина внутрішнього шляху безпосередньо пов'язана з вартістю влучень при пошуку, а довжина зовнішнього шляху безпосередньо пов'язана з вартістю промахів при пошуку.

Найгірші випадки дерев пошуку

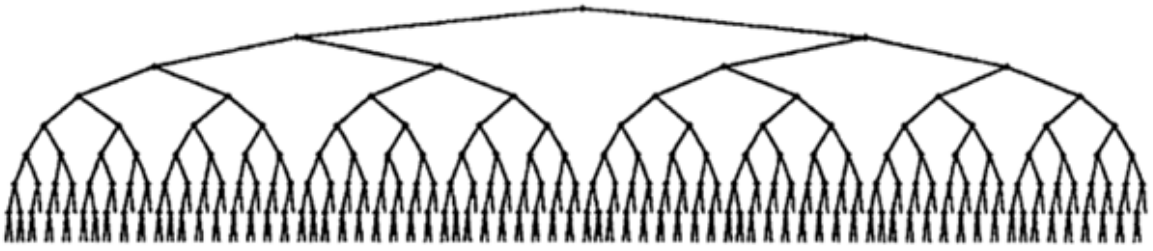




Хороший випадок дерева пошуку



Найкращий випадок дерева пошуку



5. *Перерахуйте та прокоментуйте переваги та недоліки 2-3-4-дерев. Поясніть основні принципи їх побудови.*

Головна відмінність 2-3-4 дерев від 2-3 полягає в тому, що вони можуть містити більше трьох дочірніх вузлів, що дає можливість створювати

чотиримісні вузли (вузли, що мають чотири дочірні вузли та три елементи даних). Можна побачити відмінності візуально на гіфці під цим текстом. На першому слайді показано 2-3 дерева, на другому - 2-3-4.

До даних, що розміщуються у вузлах 2-3-4 дерева висуваються деякі вимоги (як і до даних, що розміщуються у 2-3 деревах)

1. Якщо вузол містить 2 елементи і має 2 дочірні вузли, то вузол повинен містити один елемент, значення якого має бути більше, ніж значення лівого дочірнього вузла, і менше, ніж значення правого дочірнього вузла
2. Якщо вузол містить 2 елементи і має 3 дочірні вузли, то вузол повинен задовольняти наступним співвідношенням: значення X більше значень лівого дочірнього вузла і менше значень середнього дочірнього вузла; значення Z більше значень середнього дочірнього вузла та менше значень правого дочірнього вузла.
3. Якщо вузол містить 3 елементи і має 4 дочірні вузли, то вузол повинен задовольняти наступним співвідношенням: значення X більше значень лівого дочірнього вузла і менше значень лівого середнього дочірнього вузла; значення Y більше значень лівого середнього дочірнього вузла та менше значень правого середнього дочірнього вузла; значення Z більше значень правого середнього дочірнього вузла та менше значень правого дочірнього вузла.
4. Аркуш може містити один, два або три елементи.

Головні плюси 2-3-4 дерев у порівнянні з 2-3 деревами полягають у тому, що стандартні операції вставки та видалення елементів здійснюються за меншу кількість кроків.

Головним мінусом є кількість необхідної пам'яті, адже по 2-3-4 дерева можуть містити більшу кількість елементів, які потрібно десь зберігати, треба буде споживати більше пам'яті. Для того, щоб усунути цю проблему можна використовувати червоно-чорне бінарне дерево (red-black tree) спеціального виду.

6. *Перерахуйте та прокоментуйте переваги та недоліки червоно-чорних дерев. Поясніть основні принципи їх побудови.*

Червоно-чорне дерево являє собою бінарне дерево пошуку з одним додатковим бітом кольору в кожнім вузлі. Колір вузла може бути або

червоним, або чорним. Відповідно до обмежень, що накладаються на вузли дерева, шлях у червоно-чорному дереві не відрізняється від іншого по довжині більш раза у два рази, червоно-чорні дерева є приблизно збалансованими. Кожен вузол дерева містить поля color, key, left, right і p. Якщо не існує дочірнього чи батьківського вузла стосовно даного, відповідний вказівник приймає значення NULL. Ми будемо розглядати ці значення NULL як вказівники на зовнішні вузли (листи) бінарного дерева пошуку. При цьому всі “нормальні” вузли, що містять поле ключа, стають внутрішніми вузлами дерева.

Бінарне дерево пошуку є червоно-чорним деревом, якщо воно задовольняє наступним червоно-чорним властивостям.

1. Кожен вузол є червоним чи чорним.

2. Корінь дерева є чорним.

3. Кожен лист дерева (NULL) є чорним.

4. Якщо вузол — червоний, то обоє його дочірніх вузла — чорні

Чорно-червоні дерева являють собою дерева пошуку в двійковій системі координат. У цих системах у будь-якого сайту є певне значення кольору. Воно може брати на себе одне з вищезазначених позначень

7. *Перерахуйте та прокоментуйте переваги та недоліки навантажених дерев. В яких випадках їх використання найефективніше?*

Навантажені дерева призначені для представлення множин, що складаються з символьних рядків (деякі з методів можуть працювати і з рядками об'єктів іншого типу, наприклад з рядками цілих чисел). Структура навантажених дерев підтримує оператори множин: вставка, видалення, очищення, друк. У навантаженому дереві кожен шлях від кореня до листа відповідає одному слову з множини. При такому підході вузли дерева відповідають префіксам слів множини. Щоб уникнути колізій слів, подібних THE і THEN, вводиться спеціальний символ $, маркер кінця, що вказує закінчення будь-якого слова. Тоді тільки слова будуть словами, але не префікси.