**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 3 з дисципліни

«Проектування алгоритмів»

„ **Проектування структур даних**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-21 Корнійко С.С.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2023

Зміст

[1 Мета лабораторної роботи 3](#_Toc114359761)

[2 Завдання 4](#_Toc114359762)

[3 Виконання 7](#_Toc114359763)

[3.1 Псевдокод алгоритмів 7](#_Toc114359764)

[3.2 Часова складність пошуку 7](#_Toc114359765)

[3.3 Програмна реалізація 7](#_Toc114359766)

[3.3.1 Вихідний код 7](#_Toc114359767)

[3.3.2 Приклади роботи 7](#_Toc114359768)

[3.4 Тестування алгоритму 8](#_Toc114359769)

[3.4.1 Часові характеристики оцінювання 8](#_Toc114359770)

[Висновок 9](#_Toc114359771)

[Критерії оцінювання 10](#_Toc114359772)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

# Завдання

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД з графічним (не консольним) інтерфейсом користувача (дані БД мають зберігатися на ПЗП), з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Структура даних** |
| 1 | Файли з щільним індексом з перебудовою індексної області, бінарний пошук |
| 2 | Файли з щільним індексом з областю переповнення, бінарний пошук |
| 3 | Файли з не щільним індексом з перебудовою індексної області, бінарний пошук |
| 4 | Файли з не щільним індексом з областю переповнення, бінарний пошук |
| 5 | АВЛ-дерево |
| 6 | Червоно-чорне дерево |
| 7 | B-дерево t=10, бінарний пошук |
| 8 | B-дерево t=25, бінарний пошук |
| 9 | B-дерево t=50, бінарний пошук |
| 10 | B-дерево t=100, бінарний пошук |
| 11 | Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 12 | Файли з щільним індексом з областю переповнення, однорідний бінарний пошук |
| 13 | Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 14 | Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук |
| 15 | АВЛ-дерево |
| 16 | Червоно-чорне дерево |
| 17 | B-дерево t=10, однорідний бінарний пошук |
| 18 | B-дерево t=25, однорідний бінарний пошук |
| 19 | B-дерево t=50, однорідний бінарний пошук |
| 20 | B-дерево t=100, однорідний бінарний пошук |
| 21 | Файли з щільним індексом з перебудовою індексної області, метод Шарра |
| 22 | Файли з щільним індексом з областю переповнення, метод Шарра |
| 23 | Файли з не щільним індексом з перебудовою індексної області, метод Шарра |
| 24 | Файли з не щільним індексом з областю переповнення, метод Шарра |
| 25 | АВЛ-дерево |
| 26 | Червоно-чорне дерево |
| 27 | B-дерево t=10, метод Шарра |
| 28 | B-дерево t=25, метод Шарра |
| 29 | B-дерево t=50, метод Шарра |
| 30 | B-дерево t=100, метод Шарра |
| 31 | АВЛ-дерево |
| 32 | Червоно-чорне дерево |
| 33 | B-дерево t=250, бінарний пошук |
| 34 | B-дерево t=250, однорідний бінарний пошук |
| 35 | B-дерево t=250, метод Шарра |

# Виконання

## Псевдокод алгоритмів

**function** Add(root, value)

**if** root **is** **null**

**return** a new node with value

**else if** value < root.value

root.left = Add(root.left, value)

**else if** value > root.value

root.right = Add(root.right, value)

**else**

**return** root

root.height = 1 + max(height(root.left), height(root.right))

balance = getBalance(root)

**if** balance > 1 **and** value < root.left.value

**return** rightRotate(root)

**if** balance < -1 **and** value > root.right.value

**return** leftRotate(root)

**if** balance > 1 **and** value > root.left.value

root.left = leftRotate(root.left)

**return** rightRotate(root)

**if** balance < -1 **and** value < root.right.value

root.right = rightRotate(root.right)

**return** leftRotate(root)

**return** root

**end function**

**function** Remove(root, key)

**if** root **is NULL**

**return** root

**else if** key < root.key

root.left = Remove(root.left, key)

**else if** key > root.key

root.right = Remove(root.right, key)

**else**

**if** root.left **is NULL**

temp = root.right

**free**(root)

**return** temp

**else if** root.right **is NULL**

temp = root.left

**free**(root)

**return** temp

temp = minValueNode(root.right)

root.key = temp.key

root.right = Remove(root.right, temp.key)

**if** root **is NULL**

**return** root

root.height = 1 + max(height(root.left), height(root.right))

balance = getBalance(root)

**if** balance > 1

**if** getBalance(root.left) >= 0

return rightRotate(root)

**else**

root.left = leftRotate(root.left)

return rightRotate(root)

**if** balance < -1

**if** getBalance(root.right) <= 0

return leftRotate(root)

**else**

root.right = rightRotate(root.right)

return leftRotate(root)

**return** root

**end function**

**function** Find(root, key)

**if** root **is NULL** or root.key = key

**return** root

**if** root.key < key

**return** Find(root.right, key)

**return** Find(root.left, key)

**end function**

**function** Edit(root, key, newValue)

**if** root **is NULL**

**return** root

**else if** key < root.key

root.left = Edit(root.left, key, newValue)

**else if** key > root.key

root.right = Edit(root.right, key, newValue)

**else**

root.value = newValue

**return root**

**end function**

## Часова складність пошуку

Оскільки АВЛ дерево – це вид бінарного дерева пошуку, яке є збалансованим, то висота дерева становить O(log(N)). Оскільки для пошуку елементу в найгіршому випадку необхідно пройти шлях від вершини до листка довжиною у висоту дерева, то складність пошуку також O(log(N)).

## Програмна реалізація

### Вихідний код

#include "stdafx.h"

#include <iostream>

#include <ctime>

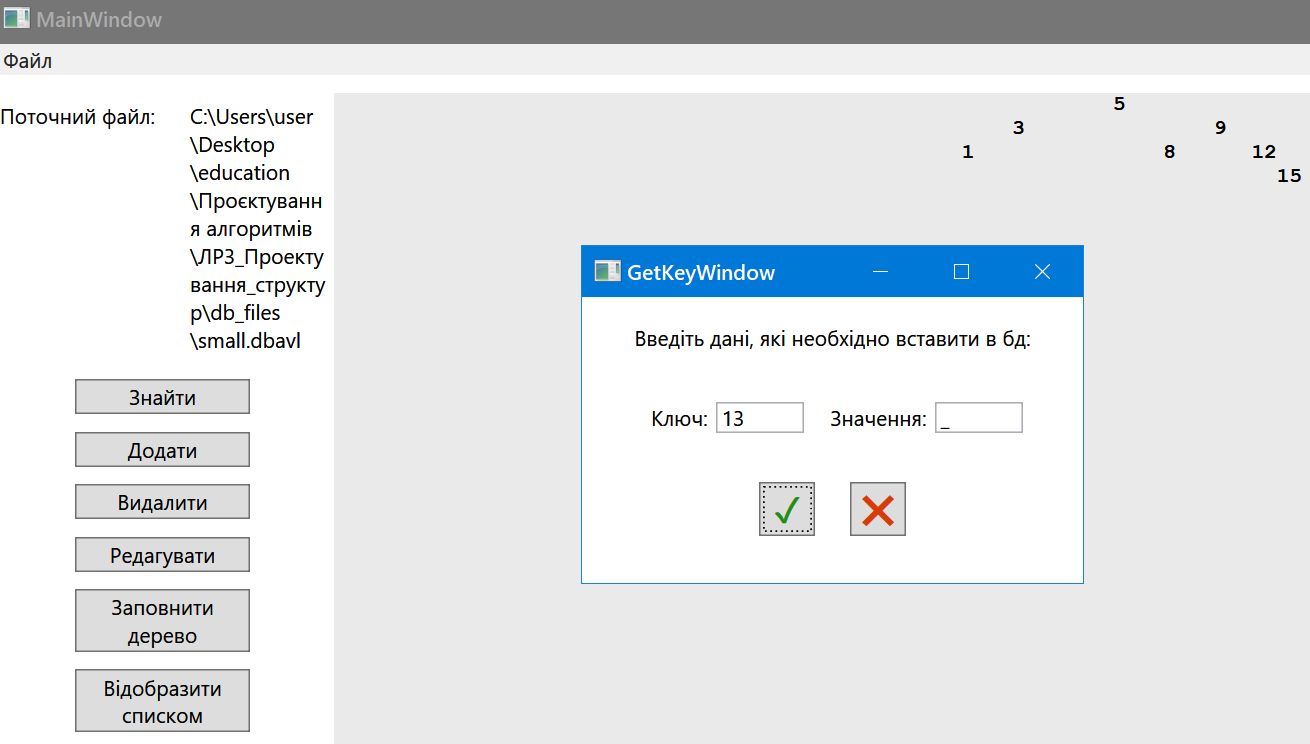
#include <iomanip>

using namespace std;

…

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.



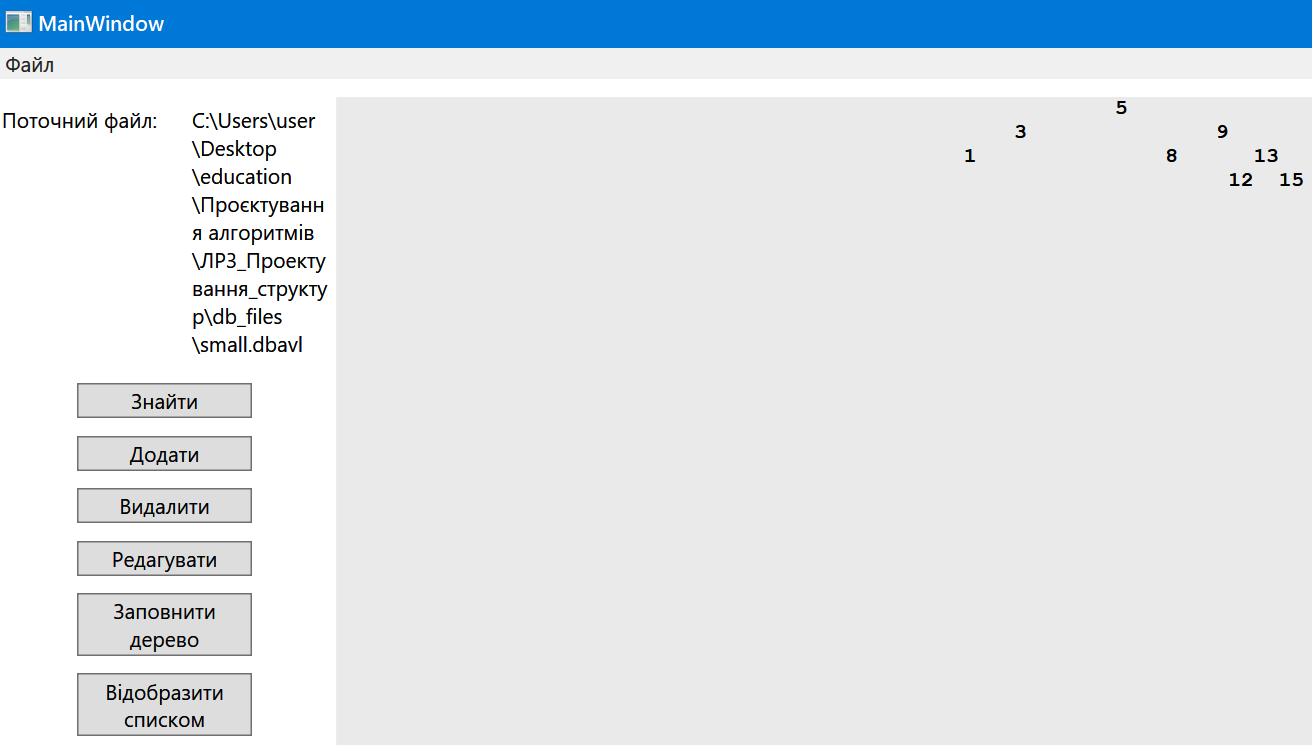
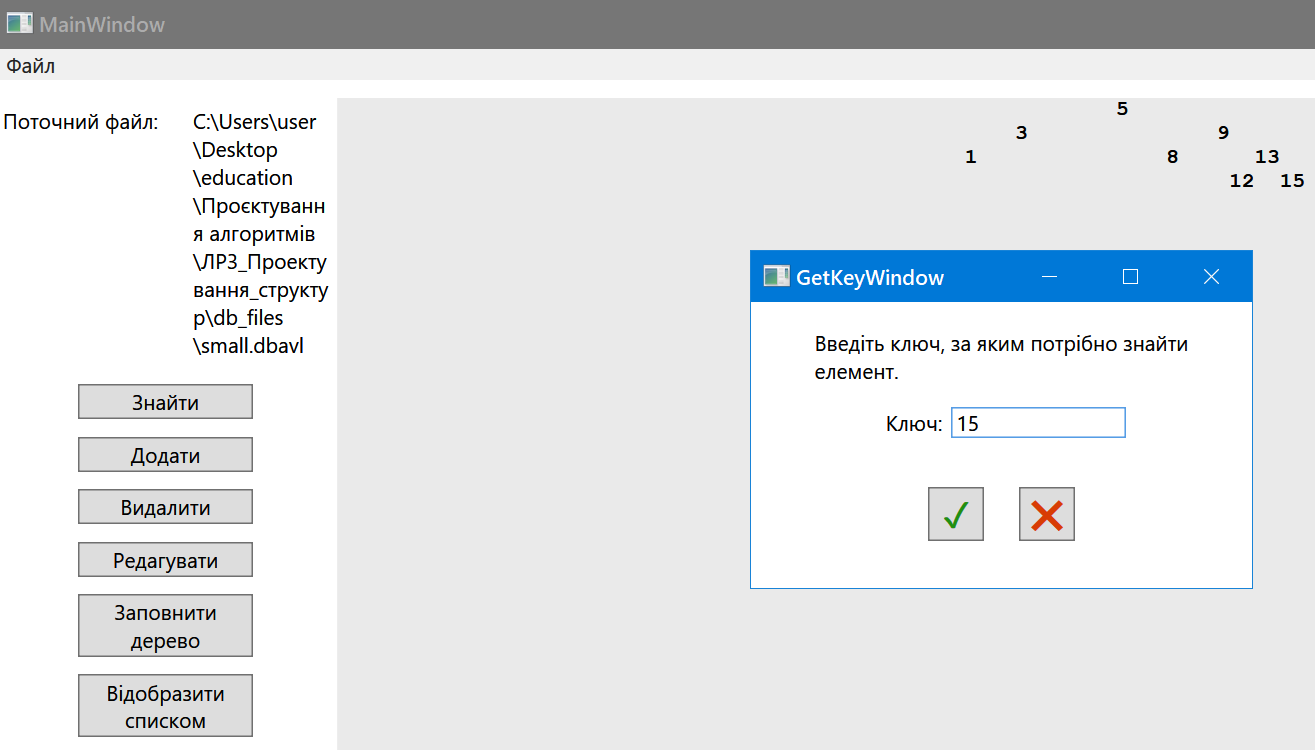


Рисунок 3.1 –Додавання запису



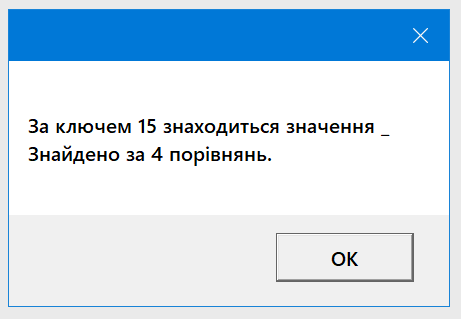


Рисунок 3.2 – Пошук запису

## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для 15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

|  |  |
| --- | --- |
| Номер спроби пошуку | Число порівнянь |
| 1 | 10 |
| 2 | 13 |
| 3 | 12 |
| 4 | 12 |
| 5 | 13 |
| 6 | 11 |
| 7 | 7 |
| 8 | 10 |
| 9 | 11 |
| 10 | 8 |
| 11 | 5 |
| 12 | 9 |
| 13 | 3 |
| 14 | 13 |
| 15 | 10 |

Висновок

В рамках лабораторної роботи було вивчено структуру даних АВЛ дерево. Для виконання роботи була створена програма, яка реалізує вставку, пошук, видалення та редагування елементів, що зберігаються у структурі. Результати цих операцій переглядаються у графічному інтерфейсі користувача, в якому відображено дерево з вершинами, підписаними відповідними ключами. Тестуючи вставку та видалення елементів за ключем було перевірено роботу розробленої програми на різних випадках поворотів дерева. При оформленні звіту було оцінено складність пошуку елементу за ключем у дереві, що складає O(log(N)). Цю оцінку було підтверджено на практиці у 15 вимірюваннях кількості порівнянь для знаходження вершини дерева за ключем. В середньому приблизно 10 порівнянь необхідно алгоритму для пошуку елементу в масиві розміром 10^4 елементів, що підтверджує його логарифмічну складність.

Отже, АВЛ дерево – це вид бінарного дерева пошуку, яке є збалансованим, за рахунок чого досягається висока швидкість операцій над вмістом структури даних, проте за рахунок здійснення поворотів сповільнюється вставка та видалення елементів, а також збільшується розмір необхідної пам’яті, оскільки доводиться зберігати інформацію про висоту кожного елемента в дереві, тому цю структуру даних буде доцільно використовувати у випадку, коли необхідно отримати високу швидкість пошуку і редагування, за рахунок більш повільної вставки та видалення елементів, а також використання додаткового об’єму пам’яті.

Критерії оцінювання

За умови здачі лабораторної роботи до 26.11.2023 включно максимальний бал дорівнює – 5. Після 26.11.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 10%;
* аналіз часової складності – 5%;
* програмна реалізація алгоритму – 50%;
* робота з гіт – 20%
* тестування алгоритму – 10%;
* висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного відображення структури ключів.

+1 додатковий бал можна отримати за виконання та захист роботи до 19.11.2023.