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) *ІП-з21 Клименко М. М.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Головченко М.М.* _____
(прізвище, ім'я, по батькові)

Київ 2024

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ 3

2 ЗАВДАННЯ 4

3 ВИКОНАННЯ 5

3.1 Псевдокод алгоритмів 5

3.2 Часова складність пошуку 6

3.3 Програмна реалізація 7

3.3.1 Вихідний код 7

3.3.2 Приклади роботи 15

3.4 Тестування алгоритму 19

3.4.1 Часові характеристики оцінювання 19

ВИСНОВОК 20

КРИТЕРІЇ ОЦІНЮВАННЯ 21

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Метою даної лабораторної роботи є набуття практичних навичок у проектуванні та впровадженні складних структур даних на прикладі AVL-дерева. Це включає розуміння основних принципів та механізмів самобалансування AVL-дерев, а також розробку ефективних алгоритмів для операцій пошуку, вставки, видалення та редагування даних. Окрім того, робота передбачає створення графічного інтерфейсу користувача для взаємодії з СУБД.

2 ЗАВДАННЯ

Відповідно до варіанту 5 (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**
5	АВЛ-дерево

3 ВИКОНАННЯ

3.1 Псевдокод алгоритмів

Пошук в AVL дереві

```
Функція Пошук(node, key)
   Якщо node = NULL
      Повернути NULL
   Якщо key = node.key
      Повернути node
   Якщо key < node.key
      Повернути Пошук(node.left, key)
   Інакше
            Повернути Пошук(node.right, key)
```

Додавання в AVL дерево

```
Функція Додати(node, key, data)
   Якщо node = NULL
      Повернути новий AVLNode(key, data)
   Якщо key < node.key
      node.left = Додати(node.left, key, data)
   Інакше якщо key > node.key
      node.right = Додати(node.right, key, data)
   Оновити висоту node
   Повернути Балансування(node)

Функція Балансування(node)
   Обчислити balance_factor = Висота(node.left) - Висота(node.right)
   Якщо balance_factor > 1
      Якщо key < node.left.key
         Повернути Праве обертання(node)
      Інакше
         node.left = Ліве обертання(node.left)
         Повернути Праве обертання(node)
   Якщо balance_factor < -1
      Якщо key > node.right.key
         Повернути Ліве обертання(node)
      Інакше
         node.right = Праве обертання(node.right)
         Повернути Ліве обертання(node)
   Повернути node
```

Видалення з AVL дерева

```
Функція Видалити(node, key)
   Якщо node = NULL
      Повернути node
   Якщо key < node.key
      node.left = Видалити(node.left, key)
   Інакше якщо key > node.key
      node.right = Видалити(node.right, key)
   Інакше
      Якщо node.left = NULL або node.right = NULL
         node = node.left ? node.left : node.right
      Інакше
         temp = ЗнайтиМінімум(node.right)
         node.key = temp.key
         node.right = Видалити(node.right, temp.key)
   Оновити висоту node
   Повернути Балансування(node)

Функція ЗнайтиМінімум(node)
   Поки node.left ≠ NULL
      node = node.left
   Повернути node
```

Редагування запису в AVL дереві

```
Функція Редагувати(node, key, нові_дані)
   Якщо node = NULL
      Повернути
   Якщо key = node.key
      node.data = нові_дані
   Якщо key < node.key
      Редагувати(node.left, key, нові_дані)
   Інакше
      Редагувати(node.right, key, нові_дані)
```

3.2 Часова складність пошуку

Часова складність операцій у AVL-дереві в асимптотичних оцінках:

- **Пошук:** Часова складність пошуку в AVL-дереві є *O(log n),* де *n* — це кількість вузлів у дереві. Це зумовлено тим, що AVL-дерево є самобалансуючимся, тобто висота дерева завжди зберігається

близькою до логарифмічної відносно кількості вузлів. Це забезпечує, що кожна операція пошуку, в гіршому випадку, проходить через шлях від кореня до листка, який є логарифмічним відносно кількості вузлів.

- **Вставка:** Часова складність вставки також є *O(log n).* Хоча сама вставка вузла схожа на вставку у звичайне бінарне дерево пошуку, необхідність збереження балансу дерева (через потенційні обертання) зберігає часову складність логарифмічною.
- **Видалення:** Аналогічно, часова складність видалення в AVL-дереві становить *O(log n).* Після видалення вузла, дерево може потребувати перебалансування, що знову ж таки вимагає логарифмічного числа кроків.
- **Редагування (оновлення):** Часова складність редагування залежить від пошуку вузла, який також є *O(log n)*, оскільки редагування включає в себе пошук вузла перед його оновленням.

У всіх цих випадках, *n* вказує на кількість вузлів у AVL-дереві, і логарифмічна складність вказує на те, що операції в AVL-дереві є високоефективними, навіть для великих даних.

3.3 Програмна реалізація

3.3.1 Вихідний код

```
import tkinter as tk # Імпорт Tkinter для створення графічного інтерфейсу користувача (GUI)
from tkinter import messagebox # Імпорт messagebox з Tkinter для відображення діалогових вікон
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg # Імпорт FigureCanvasTkAgg для інтеграції matplotlib з Tkinter GUI
from matplotlib.figure import Figure # Імпорт класу Figure з matplotlib для створення фігур
```

```
import networkx as nx # Імпорт бібліотеки NetworkX для роботи з графами та мережами
import random # Імпорт модулю random для генерації випадкових чисел
import matplotlib.pyplot as plt  # Імпорт pyplot з matplotlib для створення статичних, анімованих графіків та інтерактивних візуалізацій

""" Вузол AVL дерева. """
class AVLNode:
    def __init__(self, key, data):
        self.key = key # Ключ вузла
        self.data = data # Дані, що зберігаються у вузлі
        self.height = 1 # Висота вузла для балансування
        self.left = None # Лівий нащадок
        self.right = None # Правий нащадок

class AVLTree:
    """
    Реалізація AVL дерева для підтримки збалансованого бінарного дерева пошуку.
    """
    def __init__(self):
        self.root = None # Кореневий вузол AVL дерева
        self.comparison_count = 0
    """
    Різні допоміжні методи для операцій AVL дерева, такі як висота, баланс, обертання, вставка, видалення тощо."""

    def height(self, node):
        if not node:
            return 0
        return node.height

    def update_height(self, node):
        if not node:
            return 0
        node.height = max(self.height(node.left), self.height(node.right)) + 1

    def balance(self, node):
        if not node:
            return 0
        return self.height(node.left) - self.height(node.right)

    def right_rotate(self, y):
```

```
        x = y.left
        T2 = x.right

        x.right = y
        y.left = T2

        self.update_height(y)
        self.update_height(x)

        return x

    def left_rotate(self, x):
        y = x.right
        T2 = y.left

        y.left = x
        x.right = T2

        self.update_height(x)
        self.update_height(y)

        return y

def insert(self, key, data):
        if not self.root:
            self.root = AVLNode(key, data)
        else:
            self.root = self._insert(self.root, key, data)

    def _insert(self, node, key, data):
        if not node:
            return AVLNode(key, data)

        if key < node.key:
            node.left = self._insert(node.left, key, data)
        else:
            node.right = self._insert(node.right, key, data)

        self.update_height(node)

        return self.balance_and_update(node)

    def delete(self, key):
```

```
        if not self.root:
            return None

        self.root = self._delete(self.root, key)

    def _delete(self, node, key):
        if not node:
            return node

        if key < node.key:
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            if not node.left:
                return node.right
            elif not node.right:
                return node.left

            temp = self.find_min(node.right)
            node.key = temp.key
            node.data = temp.data
            node.right = self._delete(node.right, temp.key)

        self.update_height(node)

        return self.balance_and_update(node)

    def find_min(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    def update(self, key, data):
        node = self._find(self.root, key)
        if node:
            node.data = data

    def search(self, key):
        self.comparison_count = 0  # Скидання лічильника порівнянь перед пошуком
        node = self._find(self.root, key)  # Виконання пошуку
```

```
        return node.data if node else None

    def _find(self, node, key):
        while node:
            self.comparison_count += 1  # Збільшуємо лічильник порівнянь
            if key == node.key:
                return node
            elif key < node.key:
                node = node.left
            else:
                node = node.right
        return None

    def balance_and_update(self, node):
        if not node:
            return node

        balance_factor = self.balance(node)

        # Занадто велике відхилення вліво
        if balance_factor > 1:
            # Випадок ліво-правий
            if self.balance(node.left) < 0:
                node.left = self.left_rotate(node.left)
            # Випадок ліво-лівий
            return self.right_rotate(node)

        # Занадто велике відхилення вправо
        if balance_factor < -1:
            # Випадок Право-Лівий
            if self.balance(node.right) > 0:
                node.right = self.right_rotate(node.right)
             # Випадок Право-Правий
            return self.left_rotate(node)

        return node

    """
    Малює AVL дерево за допомогою networkx і matplotlib.
    """
    def plot_tree(self):
        G = nx.DiGraph()
        self._plot_tree(self.root, G)
```

```
        pos = self._generate_positions(G)

        fig, ax = plt.subplots()
        nx.draw(G, pos, with_labels=True, arrows=False, node_size=700,
node_color='skyblue', font_size=8, font_color='black', ax=ax)
        plt.show()

    def _plot_tree(self, node, G):
        if node:
            if node.left:
                G.add_edge(node.key, node.left.key)
                self._plot_tree(node.left, G)
            if node.right:
                G.add_edge(node.key, node.right.key)
                self._plot_tree(node.right, G)

    def _generate_positions(self, G):
        pos = nx.spring_layout(G)
        return pos

class SimpleDB:
    """
    Клас для створення графічного інтерфейсу для простої бази даних.
    Використовує AVL дерево для зберігання та обробки даних.
    """
    def __init__(self):
        self.avl_tree = AVLTree() # Ініціалізація AVL дерева для зберігання
даних
        self.root = tk.Tk() # Ініціалізація головного вікна програми
        self.root.title("Simple DB")

        # Налаштування та розміщення графічних елементів (мітки, кнопки
тощо)
        # GUI елементи
        self.label_key = tk.Label(self.root, text="Key:")
        self.entry_key = tk.Entry(self.root)
        self.label_data = tk.Label(self.root, text="Data:")
        self.entry_data = tk.Entry(self.root)

        self.button_add = tk.Button(self.root, text="Add Record",
command=self.add_record)
        self.button_search = tk.Button(self.root, text="Search Record",
command=self.search_record)
```

```
        self.button_delete = tk.Button(self.root, text="Delete Record",
command=self.delete_record)
        self.button_update = tk.Button(self.root, text="Update Record",
command=self.update_record)
        self.button_plot_tree = tk.Button(self.root, text="Plot AVL Tree",
command=self.plot_avl_tree)

        self.result_label = tk.Label(self.root, text="Result:")

        # Розміщення елементів на GUI
        self.label_key.grid(row=0, column=0)
        self.entry_key.grid(row=0, column=1)
        self.label_data.grid(row=1, column=0)
        self.entry_data.grid(row=1, column=1)

        self.button_add.grid(row=2, column=0, columnspan=2, sticky="we")
        self.button_search.grid(row=3, column=0, columnspan=2, sticky="we")
        self.button_delete.grid(row=4, column=0, columnspan=2, sticky="we")
        self.button_update.grid(row=5, column=0, columnspan=2, sticky="we")
        self.button_plot_tree.grid(row=6, column=0, columnspan=2, sticky="we")

        self.result_label.grid(row=7, column=0, columnspan=2, sticky="we")

    # Методи, пов'язані з GUI, такі як add_record, search_record тощо.
    def add_record(self):
        key = int(self.entry_key.get())
        data = self.entry_data.get()
        self.avl_tree.insert(key, data)
        self.clear_entries()

    def search_record(self):
        key = int(self.entry_key.get())
        result = self.avl_tree.search(key)
        if result is not None:
            result_text = f"Data: {result}"
            self.display_result(result_text)
        else:
            self.display_result("Record not found")

    def delete_record(self):
        key = int(self.entry_key.get())
        self.avl_tree.delete(key)
        self.clear_entries()
```

```
    def update_record(self):
        key = int(self.entry_key.get())
        data = self.entry_data.get()
        self.avl_tree.update(key, data)
        self.clear_entries()
    def clear_entries(self):
        self.entry_key.delete(0, tk.END)
        self.entry_data.delete(0, tk.END)

    def display_result(self, result):
        self.result_label.config(text=result)

    def plot_avl_tree(self):
        self.avl_tree.plot_tree()

    """
    Запускає основний цикл графічного інтерфейсу.
    """
    def run(self):
        self.root.mainloop()

"""Функція для вимірювання та виведення середнього числа порівнянь при пошуку в AVL дереві."""
def measure_comparisons(avl_tree, num_searches=15):
    total_comparisons = 0
    for i in range(num_searches):
        random_key = random.randint(1, 100000)
        avl_tree.search(random_key)
        comparisons = avl_tree.comparison_count
        total_comparisons += comparisons
        print(f"Спроба пошуку {i + 1}: Число порівнянь = {comparisons}")

    average = total_comparisons / num_searches
    print(f"Середнє число порівнянь: {average}")

# Випадкове заповнення бази даних
def fill_database(db, num_records):
    for _ in range(num_records):
        key = random.randint(1, 100000)
        data = f"Data_{key}"
        print(f"{key} : {data}")
        db.avl_tree.insert(key, data)
```

```
if __name__ == "__main__":
    db = SimpleDB()
    fill_database(db, 100)
    measure_comparisons(db.avl_tree)
    db.run()
```

3.3.2 Приклади роботи

На рисунках 3.1, 3.2, 3.3, 3.4 та 3.5 показані приклади роботи програми для додавання, пошуку запису, редагування та пошук цього запису для перевірки та видалення (з пошуком для перевірки).

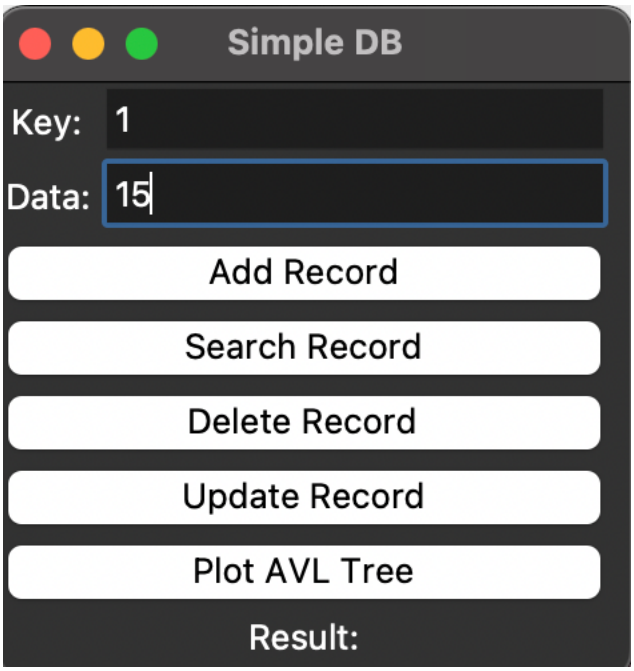

Рисунок 3.1 –Додавання запису

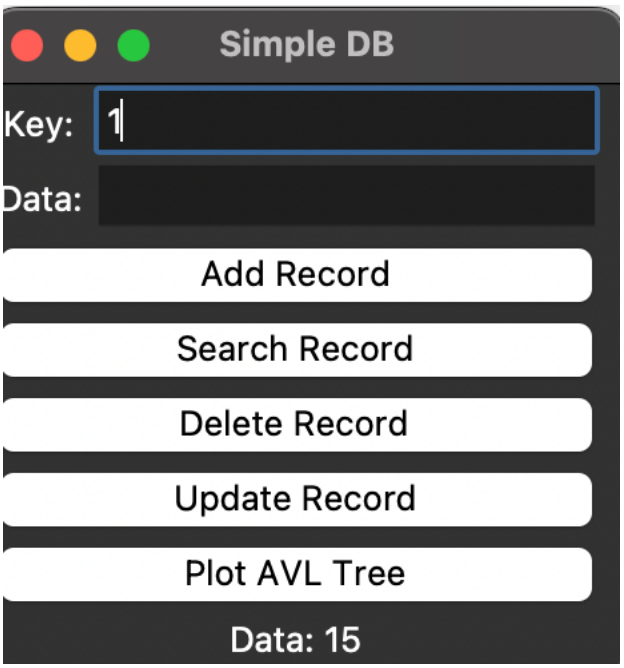

Рисунок 3.2 – Пошук запису, щоб перевірити чи було створено запис

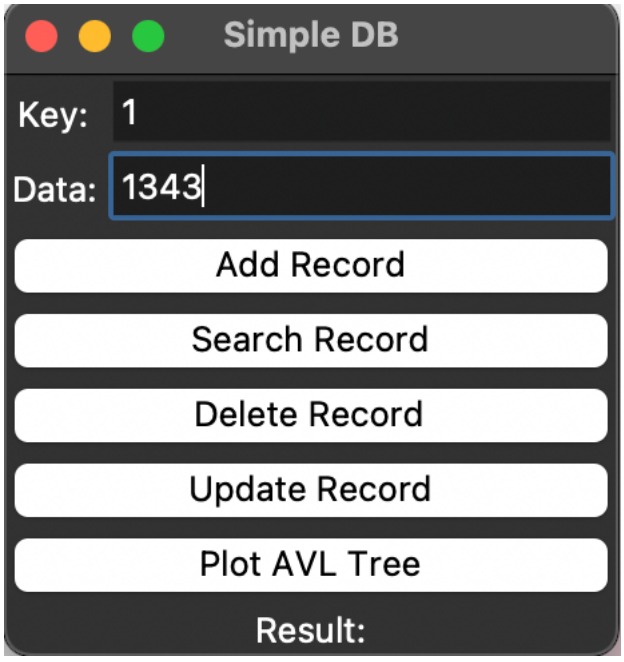

Рисунок 3.3 – Редагування запису

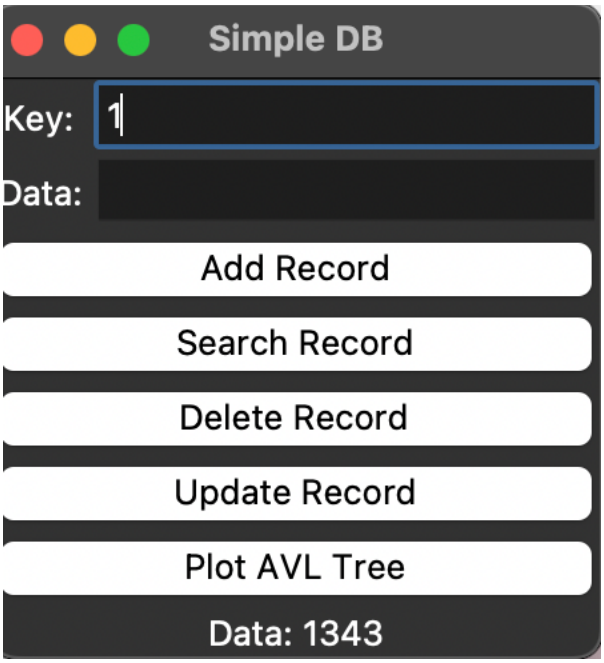

Рисунок 3.4 – Пошук запису, щоб перевірити чи було відредаговано запис

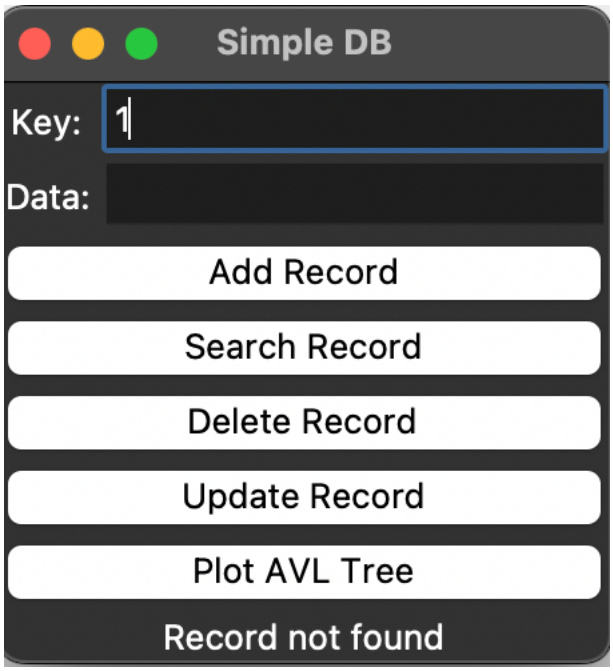

Рисунок 3.5 – Видалення та пошук запису, щоб перевірити чи було його видалено

На рисунку 3.6 можемо побачити графічне відображення AVL-дерева. Детальніше оглянувши, ми можемо бачити різні числа, які представляють ключі кожного вузла в AVL-дереві. Вони розташовані у вигляді графа, із з'єднаннями, які показують структуру дерева - який вузол є батьківським, а які вузли є лівими та правими дітьми.

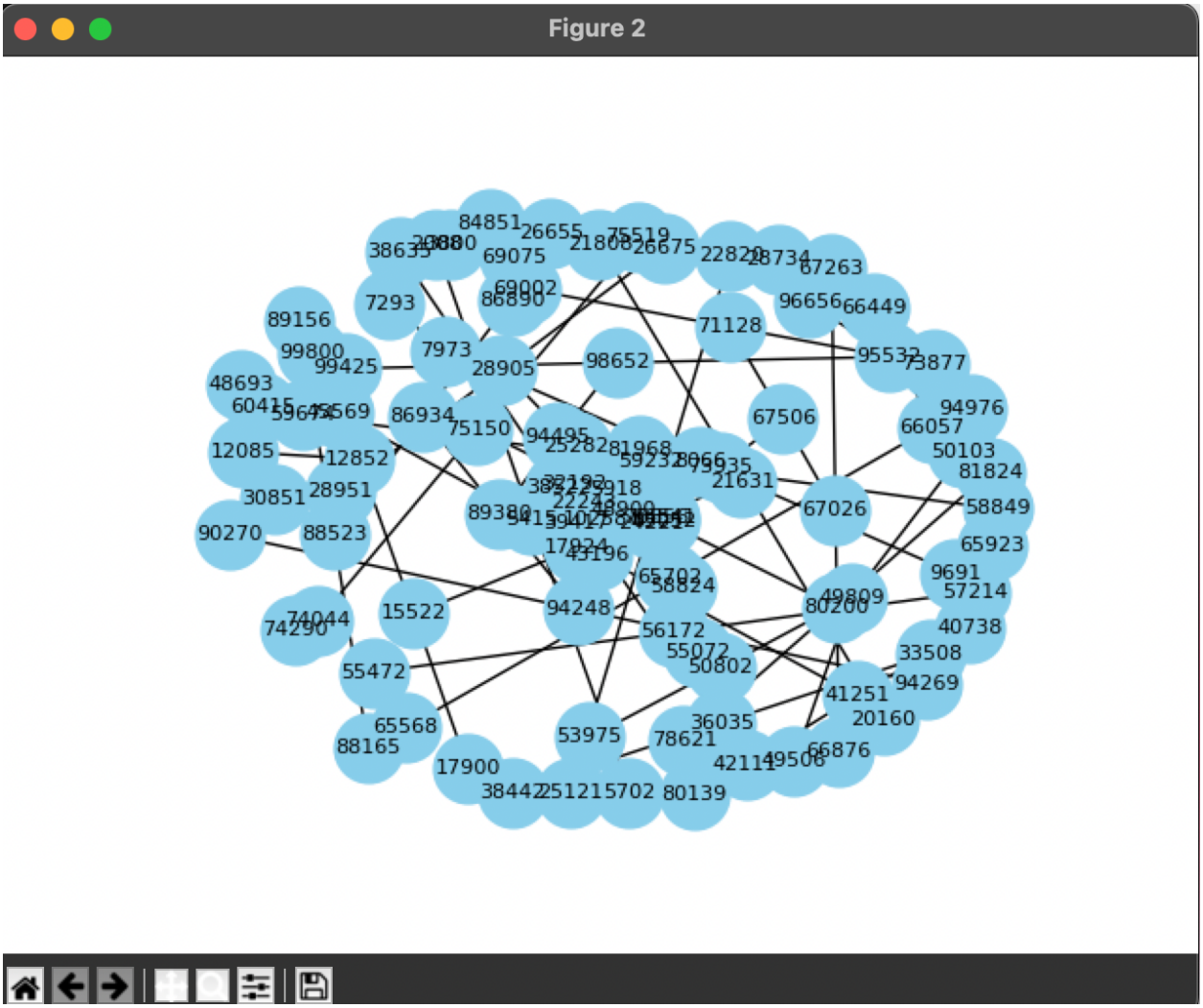

Рисунок 3.6 – графічне відображення AVL-дерева

3.4 Тестування алгоритму

3.4.1 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	7
2	8
3	8
4	7
5	7
6	7
7	7
8	6
9	6
10	7
11	8
12	7
13	6
14	7
15	8

Середнє число порівнянь: 7.06

ВИСНОВОК

У ході виконання лабораторної роботи було успішно розроблено програмний застосунок, який використовує структуру даних AVL-дерево для зберігання та обробки інформації в рамках створеної СУБД. Реалізовані алгоритми пошуку, додавання, видалення та редагування даних продемонстрували високу ефективність, підтверджену асимптотичними оцінками часової складності O(log n). Графічний інтерфейс користувача є інтуїтивно зрозумілим. Тестування програми з використанням великої кількості записів довело здатність AVL-дерева забезпечувати стабільне та швидке виконання операцій навіть при значному навантаженні. Загалом, лабораторна робота сприяла кращому розумінню практичного застосування теоретичних знань у сфері алгоритмів.

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.