

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ В.Н. КАРАЗІНА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК

КУРСОВА РОБОТА
з дисципліни «Теорія алгоритмів»
Тема «АВЛ-дерево»

Оцінка _____ балів / _____

Члени комісії:

_____ Щебенюк В. С.

_____ Олешко О. І.

_____ Лисицький К. Б.

Виконав студент 2 курсу

групи КС-21

Синюк Валентин Миколайович

Керівник:

доц. Щебенюк В. С.

ЗМІСТ

Вступ.....	3
1 Теоретичні основи	
1.1 Загальні теоретичні відомості.....	4
1.2 Область застосування прикладного рішення	4
1.3 Переваги та недоліки алгоритму	5
1.4 Основні операції для АВЛ-дерева	5
2 Методи та модель реалізації прикладної задачі	
2.1 Інструментальні засоби реалізації прикладної задачі	8
2.2 Модель програмного продукту	10
3 Реалізація прикладного рішення	
3.1 Побудова програмного продукту	11
3.2 Тестування програмного продукту.....	13
Висновок.....	16
Список джерел інформації.....	17
Додатки	18

ВСТУП

В роботі буде описано процес розробки програмної моделі (ПМ), яка представляє собою таку структури даних, як АВЛ-дерево. ПМ буде працювати, за певним сценарієм, тобто конфігурації вхідних даних (параметрів) та методів (дій), які будуть обробляти їх.

Буде вивчена відповідна література по проблематиці реалізації такої структури даних, зокрема будуть розглянуті ключові поняття та методики по роботі з нею. Розуміння методів, дадуть змогу практично застосувати набуті знання.

Структура курсової роботи складається зі вступу, трьох розділів, висновку, списку джерел інформації та додатків.

В першому розділі, будуть надані теоретичні відомості та область застосування прикладного рішення. Також, будуть розглянуті переваги та недоліки алгоритму. Окрім цього, будуть якісно описані основні методи, для роботи з АВЛ-деревом, які представляють основний функціонал. Даний розділ має теоретичну цінність.

У другому розділі, будуть детально розглянуті методи та модель прикладної задачі. Також, будуть описані різноманітні інструментальні засоби, для реалізації та візуалізації програмного продукту.

В третьому розділі, буде детально описуватися реалізація прикладного рішення, тобто, будуть розглянуті основні положення програмного продукту та відповідно буде зображено його побудова та тестування. Даний розділ має практичну цінність.

1 ТЕОРЕТИЧНІ ОСНОВИ

1.1 Загальні теоретичні відомості

АВЛ – аббревіатура, найпершого виду збалансованих довічних дерев пошуку, утворена першими літерами прізвищ авторів (радянських вчених) Адельсона-Бельського Георгія Максимовича і Ландіса Євгена Михайловича, які розробили його в 1962 році [1].

В загальному уявленні, АВЛ-дерево – це збалансоване по висоті бінарне дерево пошуку, ключі якого, задовольняють стандартній властивості: ключ будь-якого вузла дерева, не менше будь-якого ключа в лівому піддереві даного вузла і не більший від будь-якого ключа в правому піддереві цього вузла. Це означає, що для пошуку потрібного ключа, в АВЛ-дереві, може використовувати стандартний алгоритм.

Важливою особливістю АВЛ-дерева є те, що воно є збалансованим в наступному сенсі: для будь-якого вузла дерева, висота його правого піддерева, відрізняється від висоти лівого піддерева, не більше ніж як на одиницю. Доведено, що цієї властивості досить, для того щоб висота дерева логарифмічно залежала від числа його вузлів: висота h АВЛ-дерева з n ключами лежить в діапазоні від $\lceil \log_2 (n + 1) \rceil$ до $\lfloor 1.44 \log_2 (n + 2) - 0.328 \rfloor$. А так як основні операції над бінарними деревами пошуку лінійно залежать від його висоти, то отримуємо гарантовану логарифмічну залежність часу роботи цих алгоритмів від числа ключів, що зберігаються в дереві.

Нагадаємо, що рандомізовані дерева пошуку, забезпечують збалансованість тільки в ймовірнісному сенсі, тобто, ймовірність отримання сильно незбалансованого дерева, при великих n , хоча і є пренебрежимо малою, але залишається не рівною нулю [2].

1.2 Область застосування прикладного рішення

АВЛ-дерева широко використовуються в пам'яті комп'ютера, як динамічні пошукові структури. З ростом обсягів пам'яті зростає і актуальність

АВЛ-дерев, бо пошукові завдання, перш вимагали використання вінчестера, тепер задовольняються пошуком даних в пам'яті, тобто вирішуються набагато швидше [3].

1.3 Переваги та недоліки алгоритму

Двійкові дерева пошуку, в стандартній реалізації, дозволяють збільшити швидкість пошуку даних в інформаційних системах і надають точні результати, проте не здатні надати оцінку середнього часу виконання операції пошуку по базі даних фіксованого обсягу, тому було вигадано модифікований варіант двійкових дерев пошуку – АВЛ-дерев, який дозволяє давати точну оцінку швидкості виконання операції пошуку даних, а також зменшує час перебування шуканої інформації [4].

Перевага AVL-дерев полягає в їх збалансованості, яка підтримується відповідними алгоритмами вставки/видалення, проте це призводить до ускладнення алгоритму взаємодії з деревом з огляду на те, що після виконання кожної з таких операції доведеться виконувати операцію перевірки дерева на збалансованість, і при виявленні розбалансування – виконувати операції повороту вузлів дерева. Незважаючи на це, все одно отримуються переваги у вигляді підвищення швидкості виконання операцій, можливості більш точно передбачати час виконання операцій та зниження навантаження на обладнання, завдяки мінімізації висоти дерева [4].

1.4 Основні операції для АВЛ-дерева

Дамо характеристику основним операціям для роботи з АВЛ-деревими.

1 Операція вставки. Вставка нового ключа в АВЛ-дерево виконується, за великим рахунком, так само, як це робиться в простих деревах пошуку: спускаємося вниз по дереву, вибираючи правий чи лівий напрямок руху, в залежності від результату порівняння ключа, в поточному вузлі, і вставляємо свій ключ. Єдина відмінність полягає в тому, що при поверненні з рекурсії (тобто після того, як ключ вставлений або в праве, або в ліве піддерево, і це дерево

збалансовано) виконується балансування поточного вузла. Строго доводиться, що дисбаланс, при такій вставці, буде в будь-якому вузлі по шляху руху та не буде перевищує двох, а значить, застосування функції балансування є коректним [5].

2 Операція видалення. Якщо вершина – лист, то просто видалимо її, інакше знайдемо найближчу за значенням вершину a , поміняємо її місцями з видаляємою вершиною та видалимо. Від віддаленої вершини будемо підніматися вгору до кореня та перераховувати фактор балансу вершин. Якщо ми піднялися в вершину i з лівого піддерева, то фактор балансу зменшується на одиницю, якщо з правого, то збільшується на одиницю. Якщо ми прийшли в вершину та її баланс, став рівним 1 або -1, то це означає, що висота піддерева не змінилася і підйом можна зупинити. Якщо баланс вершини став рівним нулю, то висота піддерева зменшилася і підйом потрібно продовжити. Якщо баланс вершини a , в яку ми збираємося йти з її лівого піддерева, дорівнює -1, то робиться поворот для цієї вершини a . Аналогічно робимо поворот, якщо баланс вершини a , в яку ми йдемо з її правого піддерева, дорівнює 1. Якщо в результаті зміни вузла, фактор балансу став дорівнює нулю, то підйом триває, інакше зупиняється [5].

3 Балансування. Опишемо операції балансування, а саме малий лівий поворот, великий лівий поворот і випадки їх виникнення. Балансування нам потрібне для операцій додавання і видалення вузла. Для виправлення факторів балансу, досить знати фактори балансу двох (в разі великого повороту - трьох) вершин перед поворотом, і виправити значення цих же вершин після повороту. Позначимо фактор балансу вершини i , як $balance[i]$. Операції повороту робляться на тому етапі, коли ми знаходимося в правому сині вершини a , якщо ми робимо операцію додавання, і в лівому сині, якщо ми виробляємо операцію видалення. Обчислення проводимо заздалегідь, щоб не допустити значення 2 або -2 в вершині a . На кожній ілюстрації зображений один випадок висот піддерев. Неважко переконатися, що в інших випадках все теж буде коректно [5].

4 Пошук значення. Для пошуку елемента в дереві, можна скористатися наступною функцією, яка приймає в якості параметрів корінь дерева і шуканий

ключ. Для кожного вузла функція порівнює значення його ключа з шуканим ключем. Якщо ключі однакові, то функція повертає поточний вузол, в іншому випадку функція викликається рекурсивно для лівого або правого піддерева [5].

5 Інші, другорядні операції. Ці операції не так важливі як перші чотири, але без них, не можливо уявити адекватну роботу АВЛ-дерева. До таких операцій належать: очищення вмісту дерева, реалізація поворотів та супутні з цим методи, віднаходження мінімального та максимального значень, підрахунок висоти дерева, підрахунок загальної кількості вузлів дерева, відображення вмісту дерева, за допомогою трьох різних обходів.

2 МЕТОДИ ТА МОДЕЛЬ РЕАЛІЗАЦІЇ ПРИКЛАДНОЇ ЗАДАЧІ

2.1 Інструментальні засоби реалізації прикладної задачі

Проектування – застосовується на початку розробки програмного продукту. Найчастіше, розробники програмного забезпечення (ПЗ), схиляються до проектування візуальної моделі за принципами каскадної (водоспадної) моделі. Вона представляє собою послідовність розробки ПЗ. Ієрархія такої моделі складається з декількох основних стадій реалізації та впровадження: складання вимог до програмного продукту (ПП), проектування та реалізація програмної моделі (ПМ), верифікація, тестування та технічне обслуговування ПП. Розглянемо етап проектування ПМ.

Інструментальні засоби, які використовуються для проектування, тобто моделювання та візуалізації ПМ, ґрунтуються на UML – уніфікованій мові моделювання [9]. На теперішній день, вже існує досить багато редакторів та засобів для створення UML діаграм. Найпоширенішими програмами є: Microsoft Visio, Visual Paradigm, Eclipse IDE, IntelliJ IDEA.

Для проектування візуальної моделі, яка зображена на рисунку 2.1, використовувався інструмент з середовища розробки програмного забезпечення Eclipse, а саме додатково встановлений модуль (плагін) – «The ObjectAid UML Explorer for Eclipse».

Це легкий та гнучкий інструмент для графічного представлення існуючого коду. За допомогою нього, можна створювати діаграми класів та послідовностей, які автоматично оновлюються не виходячи з синхронізації. Автоматична генерація діаграм з вихідного коду та вбудований функціонал, дозволяють якісно та швидко відтворювати модель прикладної задачі.

Плагін дуже простий у використанні. Управління відбувається за допомогою контекстного меню, при натисканні правої кнопки миші.

Модель ПП буде розроблятися на такій мові програмування, як Java. Дана мова програмування дає змогу реалізувати основні парадигми об'єктно-орієнтованого програмування (ООП): абстракція, інкапсуляція, успадкування та

поліморфізм.

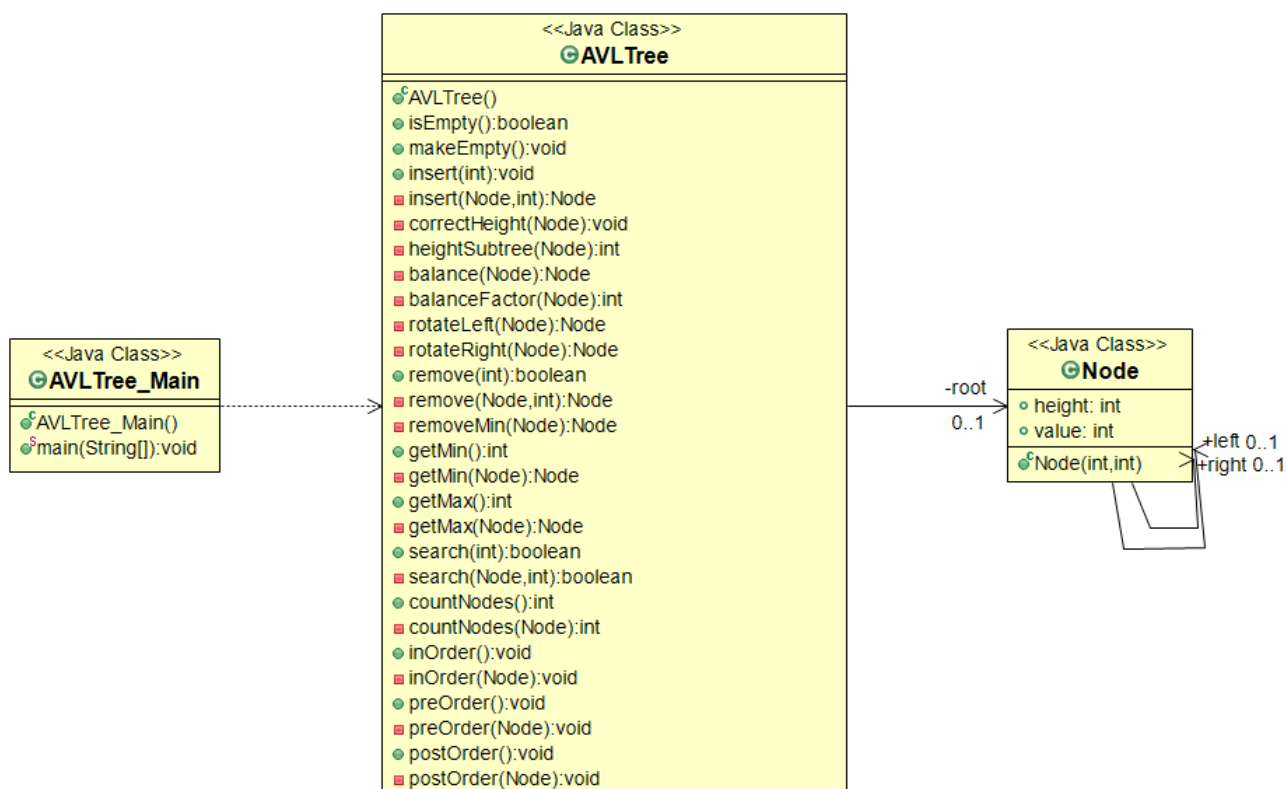


Рисунок 2.1 – UML діаграма рішення ПМ

Виконання програм на Java базується на використанні важливого елементу мови – віртуальної машини (JVM), яка є основою виконуючої системи Java, так званої Java Runtime Environment (JRE). JVM виконує байт-код, попередньо створений з вихідного тексту Java-програми компілятором Java (Javac). Компілятор приймає вихідний код, відповідний специфікації Java Language Specification і повертає байт-код, вже відповідний специфікації Java Virtual Machine Specification.

Другорядним, але не менш важливим елементом програмування на Java є – середовище розробки ПЗ (IDE), яке необхідне для реалізації прикладних задач та потребує ретельного підбору, вивчення та ознайомлення. Зважаючи на всі переваги та недоліки багатьох широкодоступних IDE, був визначений гідний кандидат, як інструмент, що використовуватиметься – IntelliJ IDEA версії

Ultimate. Він представляє собою інтегроване середовище розробки ПЗ для багатьох мов програмування, розроблене компанією JetBrains. Існують дві версії цієї IDE:

- а) Community Edition – безкоштовна версія з урізаним функціоналом;
- б) Ultimate Edition – комерційна повнофункціональна версія.

Порівняємо та надамо стислу характеристику цих версій. Більшість користувачів даної IDE використовують саме безкоштовну версію. Вона підтримує велику кількість бажаних інструментів для розробки ПЗ, наприклад, такі мови програмування як: Java, Groovy, Kotlin, Scala; такі засоби складання як: Gradle, Maven; та такі системи контролю версій як: Git, SVN, Mercurial.

Комерційна версія відрізняється наявністю підтримки багатьох інших важливих структур та технологій JVM, включаючи: Hibernate, Guice, FreeMarker, Velocity, Thymeleaf, Struts тощо. Також, придбавши цю версію, з'являться інструменти для роботи з базами даних, SQL.

2.2 Модель програмного продукту

Коротко розглянемо структуру ПП, зображену на діаграмі класів.

1 Клас AVLTree_Main – головний клас, оскільки містить в собі точку входу в програму. Також, в ньому створюється об'єкт типу класу AVLTree та містить меню для керування виконання програми.

2 Клас AVLTree – клас, який відповідає, за всю функціональність ПП, оскільки в ньому реалізовані всі затребувані методи з ПЗ. Також, клас, тримає зв'язок з класом Node.

3 Клас Node – клас, який представляє собою зразок вузла дерева, з такими даними: лівий та правий наслідники, висота піддерева в даному вузлі та зберігає значення у цьому самому вузлі.

3 РЕАЛІЗАЦІЯ ПРИКЛАДНОГО РІШЕННЯ

3.1 Побудова програмного продукту

В результаті розробки програмного продукту (ПП), виконано все, що було заплановано та затребувалося прикладною задачею (ПЗ), критерії по реалізації ПП дотримані, характеристика усім компонентам надана, всі можливі результати і ситуації – враховані. Тому отримана структура ПП є коректною та складається з наступних класів, з переліками основних методів.

1 Клас `AVLTree_Main` – головний клас, оскільки містить в собі точку входу в програму, у вигляді метода `main(String[] args)`. Також, в ньому створюється об'єкт типу класу `AVLTree`. Окрім цього, він містить меню, з можливістю вибору всіх доступних операцій з АВЛ-деревом, для більшого уявлення про клас, у додатку А представлений відповідний код реалізації.

2 Клас `AVLTree` – клас, який відповідає за всю функціональність ПП, оскільки в ньому реалізовані всі затребувані методи з ПЗ, вихідний код представлено у додатку Б. Даний клас містить такі методи як:

а) `insert(int value)` – метод, за допомогою якого, можлива вставка значення до дерева. Разом з цим, відбувається виклик методу для збалансування вузлів (`balance(Node node)`);

б) `balance(Node node)` – метод, який необхідний для правильного збалансування вузлів дерева. Для цього реалізовані методи лівого та правого поворотів, які зображені на рисунку 3.1, та декілька допоміжних;

в) `boolean remove(int value)` – метод, який необхідний для видалення вузла дерева, по передаваному значенню. Після процесу видалення відбувається збалансування вузлів методом `balance(Node node)`. Повертає логічне значення `true`, якщо видалення вдалося;

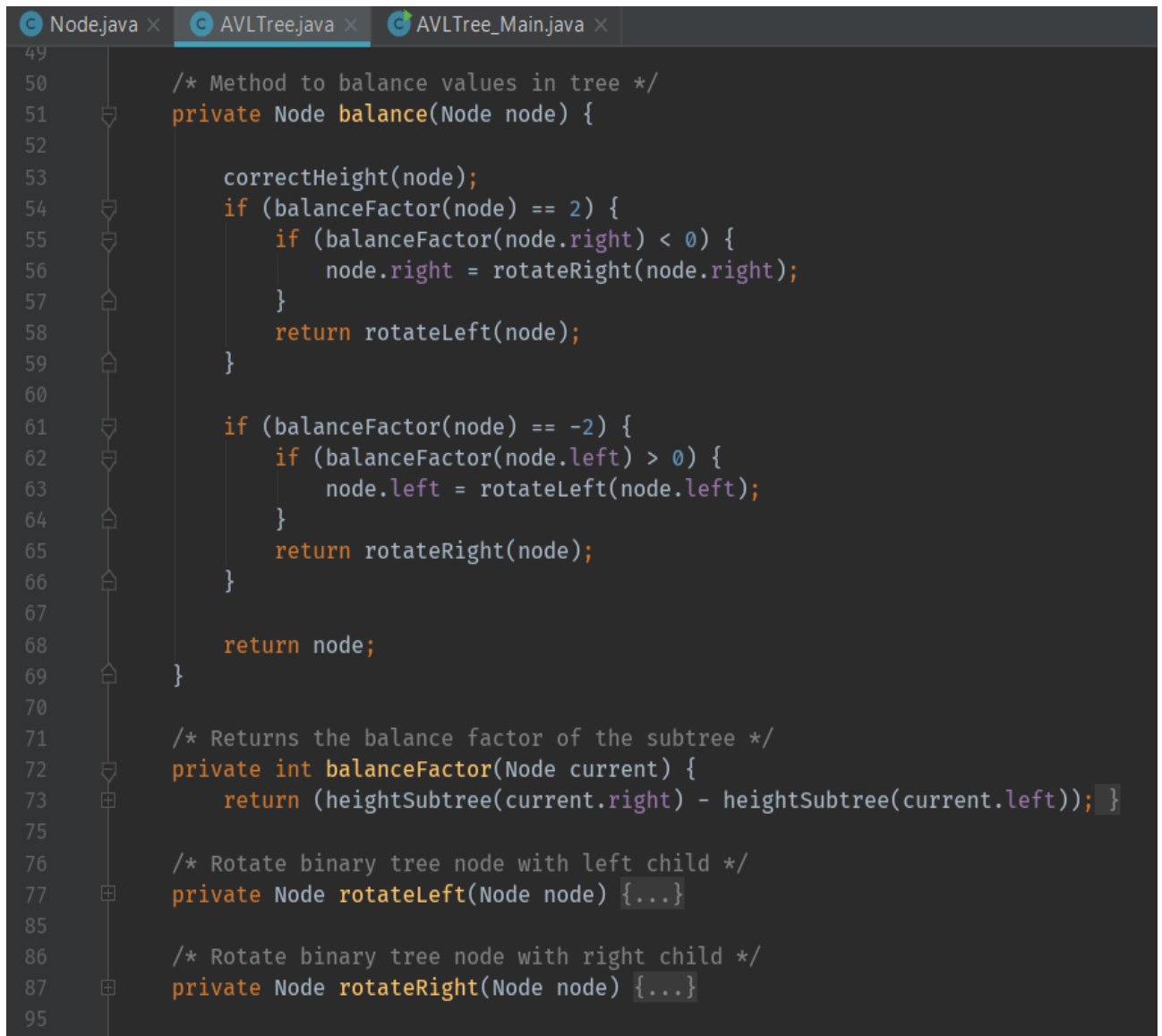
г) `int getMin()` – метод, для отримання мінімального значення в дереві;

д) `int getMax()` – метод, для отримання максимального значення в дереві;

е) `boolean search(int value)` – метод, за допомогою якого, можливий пошук відповідного значення, яке було передано з головного класу. Повертає логічне значення `true`, якщо пошук вдалий;

ж) `countNodes()` – метод, який підраховує кількість вузлів дерева;

и) `inOrder()`, `preOrder()`, `postOrder()` – методи, для друку значень дерева різними способами (обходами).



```

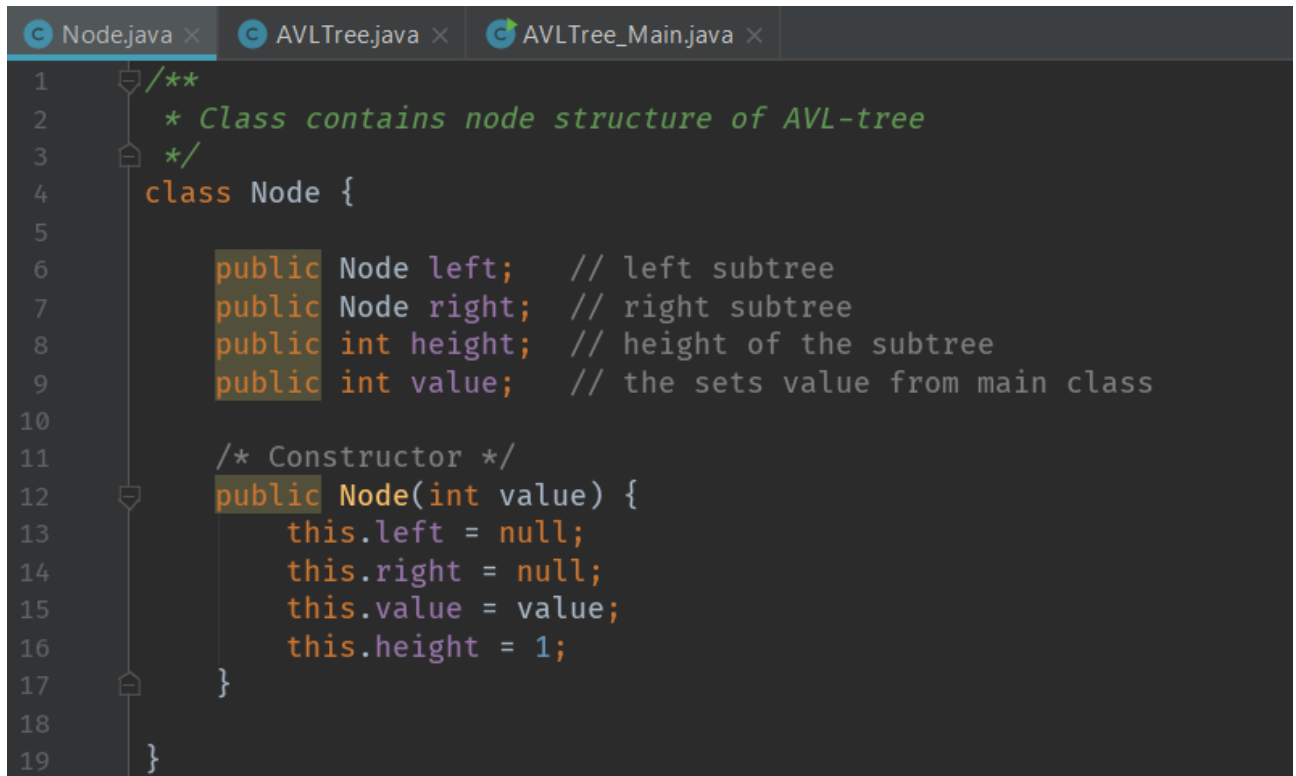
49
50  /* Method to balance values in tree */
51  private Node balance(Node node) {
52
53      correctHeight(node);
54      if (balanceFactor(node) == 2) {
55          if (balanceFactor(node.right) < 0) {
56              node.right = rotateRight(node.right);
57          }
58          return rotateLeft(node);
59      }
60
61      if (balanceFactor(node) == -2) {
62          if (balanceFactor(node.left) > 0) {
63              node.left = rotateLeft(node.left);
64          }
65          return rotateRight(node);
66      }
67
68      return node;
69  }
70
71  /* Returns the balance factor of the subtree */
72  private int balanceFactor(Node current) {
73      return (heightSubtree(current.right) - heightSubtree(current.left));
74  }
75
76  /* Rotate binary tree node with left child */
77  private Node rotateLeft(Node node) {...}
78
79
80
81
82
83
84
85
86  /* Rotate binary tree node with right child */
87  private Node rotateRight(Node node) {...}
88
89
90
91
92
93
94
95

```

Рисунок 3.1 – Балансування вузлів

3 Клас `Node` – клас, який представляє собою зразок вузла дерева, з такими даними: лівий та правий насліники, висота піддерева в даному вузлі та зберігаєме

значення у цьому самому вузлі. Дані отримуються з класу AVLTree та обробляються (ініціалізуються значення об'єкту) конструктором. Вихідний код представлено у додатку В та зображено на рисунку 3.2.



```

1  /**
2   * Class contains node structure of AVL-tree
3   */
4  class Node {
5
6      public Node left;    // left subtree
7      public Node right;   // right subtree
8      public int height;   // height of the subtree
9      public int value;    // the sets value from main class
10
11     /* Constructor */
12     public Node(int value) {
13         this.left = null;
14         this.right = null;
15         this.value = value;
16         this.height = 1;
17     }
18
19 }

```

Рисунок 3.2 – Вміст класу Node

3.2 Тестування програмного продукту

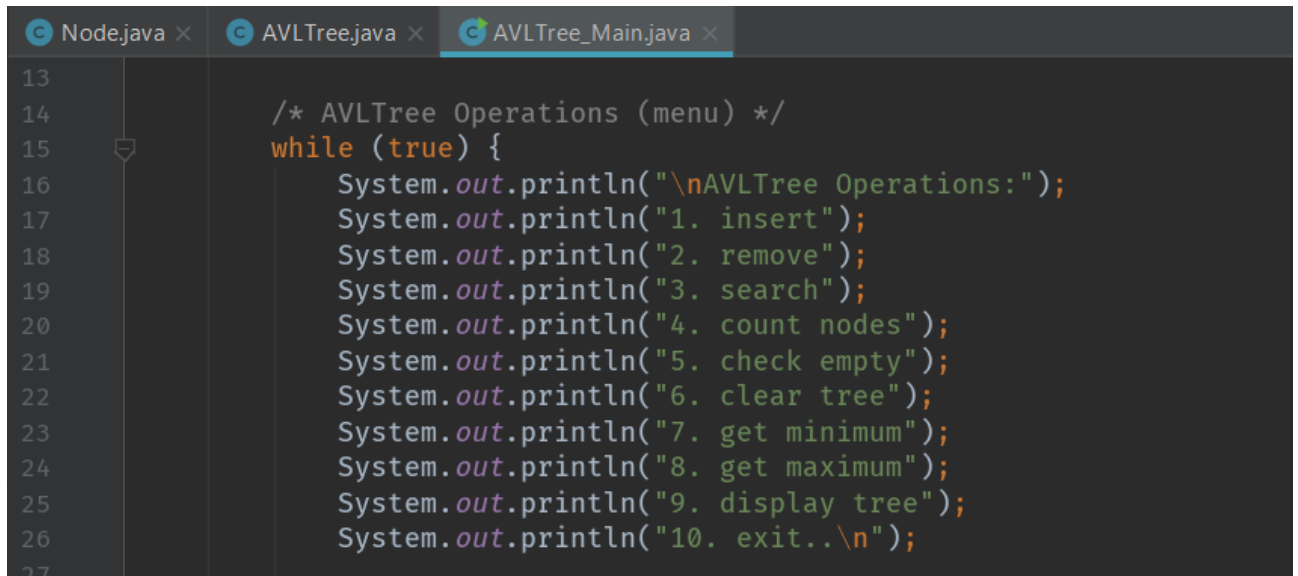
Процес функціонування ПП, абстрактно, можна поділити на декілька стадій:

- 1) створення та ініціалізація відповідних полів;
- 2) отримання від користувача затребуваних пунктів меню, які представляють собою деякі завдання, та подальше їх оброблення в відповідних методах, з візуалізацією результатів на консоль IDE.

Протестуємо ПП на виявлення неточностей, помилок та багів, в кожній з виділених стадій окремо, за допомогою відлагоджувача програмного середовища IntelliJ IDEA.

Перша стадія представляє собою створення об'єкта класу AVLTree.

Одразу ж після цього, формується запит, у вигляді меню, на вибір необхідної процедури для модифікації або відображення АВЛ-дерева, реалізація вище сказаного, зображена на рисунку 3.3.



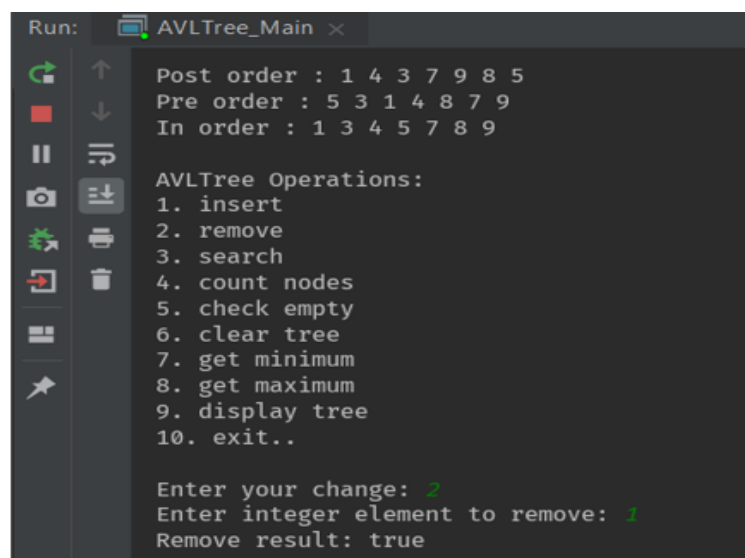
```

13
14      /* AVLTree Operations (menu) */
15      while (true) {
16          System.out.println("\nAVLTree Operations:");
17          System.out.println("1. insert");
18          System.out.println("2. remove");
19          System.out.println("3. search");
20          System.out.println("4. count nodes");
21          System.out.println("5. check empty");
22          System.out.println("6. clear tree");
23          System.out.println("7. get minimum");
24          System.out.println("8. get maximum");
25          System.out.println("9. display tree");
26          System.out.println("10. exit..\n");
27

```

Рисунок 3.3 – Меню програми

Розроблена модель працює в автономному режимі, тому на другій стадії функціонування ПП, розпочнуть свою роботу основні процеси. До них належать всі методи з класу AVLTree. Передбачена візуалізація результатів виконання роботи, як зображено на рисунку 3.4.



```

Run: AVLTree_Main x
Post order : 1 4 3 7 9 8 5
Pre order : 5 3 1 4 8 7 9
In order : 1 3 4 5 7 8 9

AVLTree Operations:
1. insert
2. remove
3. search
4. count nodes
5. check empty
6. clear tree
7. get minimum
8. get maximum
9. display tree
10. exit..

Enter your change: 2
Enter integer element to remove: 1
Remove result: true

```

```
AVLTree Operations:
1. insert
2. remove
3. search
4. count nodes
5. check empty
6. clear tree
7. get minimum
8. get maximum
9. display tree
10. exit..

Enter your change: 9

Post order : 4 3 7 9 8 5
Pre order : 5 3 4 8 7 9
In order : 3 4 5 7 8 9
```

Рисунок 3.4 – Візуалізація роботи методів видалення та друку

ВИСНОВОК

В роботі був описаний процес розробки програмної моделі (ПМ), яка представляє собою таку структури даних, як АВЛ-дерево. ПМ працює, за певним сценарієм, тобто конфігурації вхідних даних (параметрів) та методів (дій), які будуть обробляти їх.

Була вивчена відповідна література по проблематиці реалізації такої структури даних, зокрема розглянуті ключові поняття та методики по роботі з нею. Сформоване розуміння методів, що дало змогу практично застосувати набуті знання. Тому, була розроблена відповідна демонстраційна програма.

Структура курсової роботи складається зі вступу, трьох розділів, висновку, списку джерел інформації та додатків.

В першому розділі, надані теоретичні відомості та область застосування прикладного рішення. Також, розглянуті переваги та недоліки алгоритму. Окрім цього, якісно описані основні методи, для роботи з АВЛ-деревом, які представляють основний функціонал. Даний розділ має теоретичну цінність.

У другому розділі, детально розглянуті методи та модель прикладної задачі. Також, описані різноманітні інструментальні засоби, для реалізації та візуалізації програмного продукту.

В третьому розділі, були застосовані набуті знання, отримані з двох попередніх розділів. Тому, було детально описано реалізацію прикладного рішення, тобто, розглянуті основні положення програмного продукту та відповідно зображено його побудова та тестування. Даний розділ має практичну цінність.

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1 АВЛ-деревья [Электронный ресурс] – Режим доступа до ресурсу: <https://habr.com/post/150732/.2>.

2 АВЛ-дерево [Электронный ресурс] – Режим доступа до ресурсу: <https://ru.wikipedia.org/wiki/АВЛ-дерево>.

3 Поисковая система длительного пользования на базе авл-дерева [Электронный ресурс] – Режим доступа до ресурсу: <http://network-journal.mpei.ac.ru/cgi-bin/main.pl?l=ru&n=7&pa=4&ar=1>.

4 Сергеев М. И. АВЛ-деревья, выполнение операций над ними [Электронный ресурс] / М. И. Сергеев, А. Г. Янишевская – Режим доступа до ресурсу: <https://docplayer.ru/34310396-Avl-derevyu-vypolnenie-operaciy-nad-nimi-m-i-sergeev-a-g-yanishevskaya.html>.

5 АВЛ-дерево [Электронный ресурс] – Режим доступа до ресурсу: <https://neerc.ifmo.ru/wiki/index.php?title=АВЛ-дерево>.

6 Kishori S. Beginning Java 8 Fundamentals. Language Syntax, Arrays, Data Types, Objects, and Regular Expressions. / Sharan Kishori., 2014. – 789 с. – (Apress).

7 Oracle. The Java Tutorials [Электронный ресурс] / Oracle – Режим доступа до ресурсу: <https://docs.oracle.com/javase/tutorial/java/index.html>.

8 Object Oriented Programming – Java OOPs Concepts With Examples [Электронный ресурс] – Режим доступа до ресурсу: <https://edureka.co/blog/object-oriented-programming>.

9 Fowler M. UML Distilled. A Brief Guide to the Standard Object Modeling Language / Martin Fowler. // Addison-Wesely. – 2004. – №3. – С. 175.

ДОДАТКИ

Додаток А. Лістинг класу AVLTree_Main

```

import java.util.Scanner;

/**
 * The main class
 */
public class AVLTree_Main {

    public static void main(String[] args) {

        /* create an object of the class AVLTree */
        AVLTree avlTree = new AVLTree();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\nAVLTree Operations:");
            System.out.println("1. insert");
            System.out.println("2. remove");
            System.out.println("3. search");
            System.out.println("4. count nodes");
            System.out.println("5. check empty");
            System.out.println("6. clear tree");
            System.out.println("7. get minimum");
            System.out.println("8. get maximum");
            System.out.println("9. display tree");
            System.out.println("10. exit..\n");

            System.out.print("Enter your change: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1: {
                    System.out.print("Enter element to insert: ");
                    avlTree.insert(scanner.nextInt());
                    break;
                }
                case 2: {
                    System.out.print("Enter element to remove: ");
                    System.out.println("Remove result: " +
avlTree.remove(scanner.nextInt()));
                    break;
                }
                case 3: {
                    System.out.print("Enter element to search: ");
                    System.out.println("Search result: " +
avlTree.search(scanner.nextInt()));
                    break;
                }
            }
        }
    }
}

```

```

        case 4: {
            System.out.println("Count nodes: " +
avlTree.countNodes());
            break;
        }
        case 5: {
            System.out.println("Empty status: " +
avlTree.isEmpty());
            break;
        }
        case 6: {
            System.out.println("\nTree Cleared!");
            avlTree.makeEmpty();
            break;
        }
        case 7: {
            if (avlTree.getMin() == 0) {
                System.out.println("Tree is empty!");
            } else System.out.println("Minimum element: "
+ avlTree.getMin());
            break;
        }
        case 8: {
            if (avlTree.getMax() == 0) {
                System.out.println("Tree is empty!");
            } else System.out.println("Maximum element: "
+ avlTree.getMax());
            break;
        }
        case 9: {
            System.out.print("\nPost order : ");
            avlTree.postOrder();
            System.out.print("\nPre order : ");
            avlTree.preOrder();
            System.out.print("\nIn order : ");
            avlTree.inOrder();
            break;
        }
        case 10: { // exit
            return;
        }
        default: {
            System.out.println("\nWrong Entry!");
            break;
        }
    }
}
}
}
}

```

Додаток Б. Лістинг класу AVLTree

```

/**
 * The class contains the basic methods for working with the AVL-
 * tree
 */
class AVLTree {

    /* The root node */
    private Node root;

    public AVLTree() {
        root = null;
    }

    /* Method to check if tree is empty */
    public boolean isEmpty() {
        return root == null;
    }

    /* Make the tree empty */
    public void makeEmpty() {
        root = null;
    }

    /* Method to insert value in tree */
    public void insert(int value) {
        root = insert(root, value);
    }

    private Node insert(Node current, int value) {
        if (current == null) return new Node(value);
        if (value < current.value) {
            current.left = insert(current.left, value);
        } else {
            current.right = insert(current.right, value);
        }
        return balance(current);
    }

    /* Subtree height correction */
    private void correctHeight(Node node) {
        node.height = 1 + Math.max((heightSubtree(node.left)),
        (heightSubtree(node.right)));
    }

    /* Returns the height of the subtree */
    private int heightSubtree(Node current) {
        if (current == null) return -1;
        return current.height;
    }
}

```

```

/* Method to balance values in tree */
private Node balance(Node node) {

    correctHeight(node);
    if (balanceFactor(node) == 2) {
        if (balanceFactor(node.right) < 0) {
            node.right = rotateRight(node.right);
        }
        return rotateLeft(node);
    }

    if (balanceFactor(node) == -2) {
        if (balanceFactor(node.left) > 0) {
            node.left = rotateLeft(node.left);
        }
        return rotateRight(node);
    }

    return node;
}

/* Returns the balance factor of the subtree */
private int balanceFactor(Node current) {
    return (heightSubtree(current.right) -
heightSubtree(current.left));
}

/* Rotate binary tree node with left child */
private Node rotateLeft(Node node) {
    Node tempNode = node.right;
    node.right = tempNode.left;
    tempNode.left = node;
    correctHeight(node);
    correctHeight(tempNode);
    return tempNode;
}

/* Rotate binary tree node with right child */
private Node rotateRight(Node node) {
    Node tempNode = node.left;
    node.left = tempNode.right;
    tempNode.right = node;
    correctHeight(node);
    correctHeight(tempNode);
    return tempNode;
}

/* Method to remove value from tree */
public boolean remove(int value) {
    return remove(root, value) != null;
}

```

```

private Node remove(Node node, int value) {
    if (node == null) return null;
    if (value < node.value) {
        node.left = remove(node.left, value);
    } else if (value > node.value) {
        node.right = remove(node.right, value);
    } else {
        Node tempLeft = node.left;
        Node tempRight = node.right;
        if (tempRight == null) {
            return tempLeft;
        } else {
            Node min = getMin(tempRight);
            min.right = removeMin(tempRight);
            min.left = tempLeft;
            return balance(min);
        }
    }

    return balance(node);
}

/* Method remove minimum value */
private Node removeMin(Node node) {
    if (node.left == null) {
        return node.right;
    } else {
        node.left = removeMin(node.left);
    }
    return balance(node);
}

/* Method get minimum value */
public int getMin() {
    if (root == null) {
        return 0;
    } else {
        return getMin(root).value;
    }
}

private Node getMin(Node current) {
    if (current.left == null) {
        return current;
    } else {
        return getMin(current.left);
    }
}

/* Method get maximum value */
public int getMax() {
    if (root == null) {
        return 0;
    }
}

```

```

    } else {
        return getMax(root).value;
    }
}

private Node getMax(Node current) {
    if (current.right == null) return current;
    return getMax(current.right);
}

/* Method to search for an element */
public boolean search(int value) {
    return search(root, value);
}

private boolean search(Node current, int value) {
    boolean ifFound = false;
    while ((current != null) && !ifFound) {
        int tempValue = current.value;
        if (value < tempValue)
            current = current.left;
        else if (value > tempValue)
            current = current.right;
        else {
            ifFound = true;
            break;
        }
        ifFound = search(current, value);
    }
    return ifFound;
}

/* Method to count number of nodes */
public int countNodes() {
    return countNodes(root);
}

private int countNodes(Node node) {
    if (node == null)
        return 0;
    else {
        int count = 1;
        count += countNodes(node.left);
        count += countNodes(node.right);
        return count;
    }
}

/* Function for inOrder traversal */
public void inOrder() {
    inOrder(root);
}

```

```

private void inOrder(Node node) {
    if (node != null) {
        inOrder(node.left);
        System.out.print(node.value + " ");
        inOrder(node.right);
    }
}

/* Function for preOrder traversal */
public void preOrder() {
    preOrder(root);
}

private void preOrder(Node node) {
    if (node != null) {
        System.out.print(node.value + " ");
        preOrder(node.left);
        preOrder(node.right);
    }
}

/* Function for postOrder traversal */
public void postOrder() {
    postOrder(root);
}

private void postOrder(Node node) {
    if (node != null) {
        postOrder(node.left);
        postOrder(node.right);
        System.out.print(node.value + " ");
    }
}
}

```


Додаток В. Лістинг класу Node

```
/**
 * Class contains node structure of AVL-tree
 */
class Node {

    public Node left;    // left subtree
    public Node right;   // right subtree
    public int height;   // height of the subtree
    public int value;    // the sets value from main class

    public Node(int value) {
        this.left = null;
        this.right = null;
        this.value = value;
        this.height = 1;
    }
}
```