Черкаський національний університет імені Богдана Хмельницького

Кафедра програмного забезпечення автоматизованих систем

КУРСОВА РОБОТА

з дисципліни «Об’єктно-орієнтоване програмування»

НА ТЕМУ «Поліморфні реалізації словникової структури даних»

Студента 2 курсу, групи КС-211

Галузі знань 12 «Інформаційні технології»

Спеціальності 121 «Інженерія програмного забезпечення»

Ягудін К.О.

Керівник: викл. кафедри ПЗАС

Порубльов І.М.

Національна шкала:_____

Кількість балів:_____ Оцінка: ECTS_____

Члени комісії _____ _____

(підпис) (прізвище та ініціали)

_____ _____

(підпис) (прізвище та ініціали)

_____ _____

(підпис) (прізвище та ініціали

м. Черкаси – 2023 рік

Зміст

Вступ

Поліморфні реалізації словникової структури даних є однією з ключових тем в області програмування та комп'ютерних наук. Словникова структура даних, також відома як асоціативний масив або словник, є важливим інструментом для зберігання та організації даних, де кожен елемент має унікальний ключ, пов'язаний зі значенням. Застосування словникових структур даних можна знайти в різних сферах, таких як бази даних, інформаційні системи, комп'ютерні програми та інші.

Однак, існують ситуації, коли потрібно мати здатність використовувати різні реалізації словників залежно від конкретного контексту або вимог. Наприклад, може знадобитися використовувати різні алгоритми пошуку або зберігання даних в залежності від розміру словника, типу ключів або інших факторів. У таких випадках поліморфні реалізації стають корисними, оскільки вони дозволяють гнучко пристосовувати структуру даних до конкретних потреб.

Метою даної роботи є вивчення та аналіз поліморфних реалізацій словникової структури даних з метою виявлення їх переваг та недоліків. Зокрема, планується дослідити різні підходи до поліморфних реалізацій, такі як використання інтерфейсів, наслідування та шаблонів програмування.

Для досягнення поставленої мети, у роботі будуть розглянуті наступні завдання:

1. Вивчення теоретичних аспектів словникових структур даних, їх властивостей та особливостей.
2. Аналіз різних підходів до поліморфних реалізацій словникових структур даних.
3. Порівняння ефективності та використання різних поліморфних реалізацій у реальних сценаріях застосування.
4. Виявлення переваг та недоліків кожного підходу та надання рекомендацій щодо їх використання.

Базові поняття

Для належного розуміння роботи необхідно ознайомитися з деякими базовими поняттями, пов'язаними зі словниковими структурами даних та поліморфізмом:

1. Словникова структура даних: це колекція елементів, кожен з яких має унікальний ключ, пов'язаний зі значенням. Вона дозволяє ефективно здійснювати операції вставки, пошуку та видалення елементів.
2. Поліморфізм: це принцип об'єктно-орієнтованого програмування, який дозволяє об'єктам різних типів виконувати однакові дії. У контексті поліморфних реалізацій словникових структур даних, це означає можливість використовувати різні реалізації словників залежно від потреб програми чи ситуації.
3. Інтерфейс: це контракт, який визначає набір методів та їх сигнатур, доступних для взаємодії з об'єктом. Використання інтерфейсів дозволяє реалізувати поліморфізм шляхом використання спільного набору методів.
4. Наслідування: це механізм, що дозволяє класу успадковувати властивості та методи від іншого класу. Використання наслідування може бути корисним для реалізації поліморфних реалізацій словникових структур, де базовий клас визначає загальні методи, а похідні класи надають конкретну реалізацію.

Завершуючи вступ, необхідно відзначити, що розуміння поліморфних реалізацій словникових структур даних є важливим для програмістів та дослідників, оскільки воно дозволяє знайти оптимальні рішення для зберігання та організації даних в різних ситуаціях. Далі в роботі будуть розглянуті різні підходи до поліморфних реалізацій словникових структур даних та їх порівняння для досягнення більш глибокого розуміння та вироблення рекомендацій.

Розділ 1. Дослідження словникових структури даних.

1.1 Словник: властивості та принцип роботи.

Словник (англ. Dictionary) є структурою даних, що дозволяє зберігати дані у вигляді пар ключ-значення. Давайте розглянемо основні властивості та принципи роботи цієї структури. У структурі даних dictionary вставка нових пар ключ-значення відбувається за допомогою ключа, що використовується для ідентифікації і доступу до відповідного значення.

Наприклад, при вставці пари (ключ, значення) у хеш-таблицю, спочатку обчислюється хеш-код ключа, який визначає індекс, де буде зберігатися значення. У випадку колізій, коли два або більше ключі мають однаковий хеш-код, застосовуються методи вирішення колізій, такі як ланцюжки або відкрите адресування. При видаленні пари ключ-значення з dictionary, виконується пошук за ключем, а потім видаляється відповідна пара. Оновлення значень також може відбуватися шляхом пошуку ключа та заміни його значення.

Наведений приклад описує принцип роботи dictionary на основі хеш-таблиці, але існують інші реалізації наприклад реалізація на основі дерева, вона використовує збалансоване дерево пошуку для збереження пар. Також існують реалізації, які працюють швидко у певних виключних випадка, наприклад enum dictionary, яка є оптимізацією для реалізації структури даних словника, у якій ключем пари є значення enum(-у).

У наступних пунктах будуть розглянуті найпопулярніші приклади реалізації цієї структури даних: hash table та red-black tree.

1.2 Словникова структура даних на основі hash table

Хеш-таблиця є однією з найпоширеніших структур даних для реалізації словників. Вона працює на основі механізму хешування, який дозволяє швидко знаходити значення за ключем [1]. Принцип роботи хеш-таблиці полягає в тому, що ключі перетворюються в хеш-коди за допомогою хеш-функції, і ці хеш-коди використовуються для індексування значень в масиві.

Хеш-таблиця складається з масиву бакетів, а кожен бакет може містити декілька пар ключ-значення. Хеш-функція приймає ключ і повертає індекс бакета, в якому має зберігатися відповідне значення. Якщо два ключі мають однаковий хеш-код, тоді виникає колізія. Існує кілька методів управління колізіями, наприклад:

1) Відкрите хешування: коли відбувається колізія, значення зберігаються в наступному вільному бакеті або використовується послідовний пошук вільного бакета.
2) Ланцюжкове хешування: кожен бакет містить посилання на список або іншу структуру даних, де зберігаються всі значення, які мають однаковий хеш-код.

Для вставки значення в хеш-таблицю спочатку обчислюється хеш-код ключа, потім використовуючи цей хеш-код, знаходиться відповідний бакет. Якщо в бакеті вже є значення, то застосовується відповідний метод управління колізіями для збереження нового значення. Алгоритм вставки може бути реалізований таким чином:

1. Обчислити хеш-код ключа за допомогою хеш-функції.

2. Знайти відповідний бакет за отриманим хеш-кодом.

3. Перевірити, чи в бакеті вже є значення з таким же ключем.

- Якщо таке значення знайдено, оновити його.

- Якщо такого значення немає, додати пару ключ-значення в бакет.

Операція пошуку в хеш-таблиці також працює на основі хеш-коду ключа. При пошуку хеш-таблиця використовує цей хеш-код для швидкого локалізування відповідного бакета. Алгоритм пошуку може мати наступний вигляд:

1. Обчислити хеш-код ключа за допомогою хеш-функції.

2. Знайти відповідний бакет за отриманим хеш-кодом.

3. Пройтися по значеннях в бакеті і знайти значення з відповідним ключем.

Хеш-таблиці притаманні висока швидкодія і ефективність, особливо при великому обсязі даних. Операції вставки, пошуку та видалення можуть займати сталий час, який зазвичай залежить від кількості елементів у хеш-таблиці.

1.3 Словникова структура даних на основі red-black tree

Словникова структура даних на основі Red-Black Tree є ефективним способом зберігання та операцій зі словником. Red-Black Tree є видом бінарного дерева пошуку, в якому кожен вузол має додаткове поле - кольору (червоний або чорний) [2]. Ця структура даних використовується для забезпечення балансу та швидкодії операцій.

Принцип роботи Red-Black Tree полягає в тому, що всі властивості бінарного дерева пошуку зберігаються, а також додаткові правила, які гарантують балансування дерева. Одним з найважливіших правил є те, що шлях від кореня до будь-якого листка має містити однакову кількість чорних вузлів.

Операції вставки, видалення та пошуку в Red-Black Tree можуть бути реалізовані за допомогою рекурсивних алгоритмів. Основні кроки алгоритмів виглядають наступним чином:

Приклад псевдокоду вставки в Red-Black Tree:

1. *Перевірити, чи існує корінний вузол.*

   - Якщо немає, створити новий вузол і призначити його коренем.

   - Якщо є, викликати рекурсивну функцію вставки з коренем як початковим вузлом.

2. *Вставка (вузол, ключ, значення):*

- Якщо вузол є листком, створити новий вузол з ключем і значенням, додати його як лівого або правого нащадку в залежності від порівняння з ключем поточного вузла.

- Якщо ключ вузла менший, викликати рекурсивну функцію вставки з лівим нащадком.

- Якщо ключ вузла більший, викликати рекурсивну функцію вставки з правим нащадком.

- Викликати функцію балансування для підтримки правил Red-Black Tree.

Приклад псевдокоду пошуку в Red-Black Tree:

1. *Пошук (вузол, ключ):*

   - Якщо вузол є листком, значення не знайдено.

   - Якщо ключ співпадає з ключем поточного вузла, повернути його значення.

   - Якщо ключ менший, викликати рекурсивну функцію пошуку з лівим нащадком.

   - Якщо ключ більший, викликати рекурсивну функцію пошуку з правим нащадком.

Операції видалення в Red-Black Tree є трохи складнішими і включають розгалуженість залежно від стану дерева. При видаленні необхідно забезпечити виконання правил балансу.

Загальний приклад псевдокоду алгоритму видалення:

Алгоритм видалення (tree, node):

Якщо tree порожнє або node порожній, повернути tree

Якщо node має лівого та правого нащадка:

Знайти наступний за значенням вузол, який назвемо "successor"

Замінити значення node значенням successor

Викликати рекурсивний виклик видалення для successor

Якщо node - червоний вузол:

Видалити node

Якщо node - чорний вузол та має червоного нащадка:

Замінити значення node значенням його червоного нащадка

Видалити червоного нащадка node

Якщо node - чорний вузол та не має нащадків:

Замінити node листком (пустим вузлом)

Викликати функцію виправлення для підтримки правил Red-Black Tree

Повернути tree.

Розділ 2. Проектування програмного продукту

2.1 Інтерфейс словника та навігаційного словника.

У цьому розділі ми детальніше розглянемо інтерфейси, пов'язані зі словниками та навігаційними словниками. Ці інтерфейси визначають основні методи, які повинні бути реалізовані в різних класах словників.

Інтерфейс *IDictionary* (словник) визначає основні методи для роботи зі словниками, а також має внутрішній інтерфейс *IEntry*, який представляє запис словника і використовується для збереження ключа і значення, цей внутрішній інтерфейс використовується у методах інтерфейсів нащадків, та абстрактних класів, для реалізації логіки методів не прив’язуючись до

конкретних об’єктів. Майбутні нащадки *IDictionary* будуть реалізувати клас запис з додатковою логікою, якщо це буде потрібно, який буде реалізувати інтерфейс *IEntry*.

Методи інтерфейсу *IEntry*:

- getKey(): K: Повертає ключ запису.
- getValue(): V: Повертає значення запису.

Методи інтерфейсу *IDictionary*:

- toString(): String: Повертає рядкове представлення словника.
- getOrDefault(key: K, defaultValue: V): V: Повертає значення, пов'язане з вказаним ключем. Якщо ключа немає в словнику, повертається значення за замовчуванням (defaultValue).
- remove(key: K): V: Видаляє запис зі словника, пов'язаний з вказаним ключем, і повертає його значення.
- putIfAbsent(key: K, value: V): V: Додає запис з вказаним ключем і значенням у словник, якщо відповідного ключа ще немає. Повертає попереднє значення, якщо воно було встановлене.
- put(key: K, value: V): V: Додає запис з вказаним ключем і значенням у словник. Якщо запис з таким ключем вже існує, значення оновлюється, а попереднє значення повертається.
- get(key: K): V: Повертає значення, пов'язане з вказаним ключем. Якщо ключа немає, повертається null.
- containsValue(value: V): boolean: Перевіряє, чи містить словник запис з вказаним значенням.
- computeIfPresent(key: K, remappingFunction: BiFunction<K, V, V>): V: Застосовує вказану функцію до запису, пов'язаного з вказаним ключем, якщо такий запис існує. Повертає оновлене значення.
- values(): Collection<V>: Повертає колекцію всіх значень у словнику.
- computeIfAbsent(key: K, mappingFunction: Function<K, V>): V: Застосовує вказану функцію до вказаного ключа, якщо такого

ключа немає у словнику. Повертає значення, яке було додано або вже існувало.

- replace(key: K, value: V): V: Замінює значення запису, пов'язаного з вказаним ключем, на нове значення. Повертає попереднє значення запису.
- containsKey(key: K): boolean: Перевіряє, чи містить словник запис з вказаним ключем.
- remove(key: K, value: V): boolean: Видаляє запис зі словника, якщо ключ пов'язаний з вказаним значенням. Повертає true, якщо запис був видалений.
- keys(): Collection<K>: Повертає колекцію всіх ключів у словнику.
- isEmpty(): boolean: Перевіряє, чи є словник порожнім.
- size(): int: Повертає кількість записів у словнику.

Інтерфейс *INavigableDictionary* (навігаційний словник) розширює функціональність базового словника *IDictionary* та надає можливість навігації по словнику та виконання додаткових операцій.

Додаткові методи NavigableDictionary:

- lowerEntry(key: K): IEntry<K, V>: Повертає запис з найбільшим ключем, що менший за вказаний ключ.
- lowerKey(key: K): K: Повертає найбільший ключ, що менший за вказаний ключ.
- floorEntry(key: K): IEntry<K, V>: Повертає запис з найбільшим ключем, що менший або рівний вказаному ключу.
- floorKey(key: K): K: Повертає найбільший ключ, що менший або рівний вказаному ключу.
- ceilingEntry(key: K): IEntry<K, V>: Повертає запис з найменшим ключем, що більший або рівний вказаному ключу.
- ceilingKey(key: K): K: Повертає найменший ключ, що більший або рівний вказаному ключу.
- higherEntry(key: K): IEntry<K, V>: Повертає запис з найменшим ключем, що більший за вказаний ключ.

- higherKey(key: K): K: Повертає найменший ключ, що більший за вказаний ключ.

2.1 Абстрактний шар з загальною реалізацією методів

AbstractDictionary, надає загальну реалізацію деяких методів інтерфейсу *IDictionary*, що дозволяє визначити поведінку методів і зберегти логіку використовуючи абстракції, завдяки цьому майбутні наслідники *AbstractDictionary* зможуть перевизначити логіку методів, якщо це буде потрібно, або залишити її загальною. Цей клас реалізує функціональність для словника та може бути використаний, як базовий шар для реалізації конкретних видів словників.

Реалізовані методи в *AbstractDictionary*. У багатьох методах для отримання записів словника використовується метод entrySet(), тому саме його реалізація у класах нащадках є особливо важливою.

- values(): Повертає колекцію, яка містить всі значення зі словника. Метод використовує entrySet() для отримання всіх записів словника, і потім за допомогою стріму витягує значення кожного запису.
- keys(): Повертає колекцію, яка містить всі ключі зі словника. Аналогічно до методу values(), використовує entrySet() для отримання всіх записів словника, і потім за допомогою стріму витягує ключі кожного запису.
- isEmpty(): Перевіряє, чи є словник порожнім. Перевіряє розмір словника і повертає true, якщо розмір дорівнює 0.
- containsKey(key: K): Перевіряє, чи містить словник заданий ключ. Проходиться по всім записам словника за допомогою ітератора і порівнює ключі. Якщо знайдено відповідний ключ, повертає true, в іншому випадку - false.
- remove(key: K, value: V): Видаляє запис зі словника, якщо він містить вказаний ключ і значення. Порівнює поточне значення запису зі значенням, яке передано в метод. Якщо значення не

співпадають або ключ не знайдено в словнику, повертає false. У іншому випадку, видаляє запис за ключем і повертає true.

- containsValue(value: V): Перевіряє, чи містить словник задане значення. Проходиться по всім записам словника за допомогою ітератора і порівнює значення. Якщо знайдено відповідне значення, повертає true, в іншому випадку - false.

```
@Override
public boolean containsValue(V value) {
    Iterator<IEntry<K, V>> i = entrySet().iterator();
    if (value == null) {
        while (i.hasNext()) {
            IEntry<K, V> e = i.next();
            if (e.getValue() == null)
                return true;
        }
    } else {
        while (i.hasNext()) {
            IEntry<K, V> e = i.next();
            if (value.equals(e.getValue()))
                return true;
        }
    }
    return false;
}
```

Рис 2.1 Приклад реалізації метода containsValue в абстрактному класі.

- put(key: K, value: V): Кидає виключення UnsupportedOperationException. Цей метод не має дефолтної реалізації і потребує реалізації в конкретному підкласі.
- get(key: K): Повертає значення, що відповідає заданому ключу. Проходиться по всім записам словника за допомогою ітератора і знаходить запис, у якого ключ співпадає зі шуканим ключем. Якщо знайдено відповідний запис, повертає його значення, в іншому випадку - null.
- getOrDefault(key: K, defaultValue: V): Повертає значення, що відповідає заданому ключу. Якщо ключ знайдено в словнику, повертає відповідне значення. Якщо ключ не знайдено, повертає defaultValue.
- size(): Повертає кількість записів у словнику. Використовує entrySet().size() для отримання розміру множини записів.

- remove(key: K): Видаляє запис зі словника за вказаним ключем. Проходиться по всім записам словника за допомогою ітератора і знаходить запис, у якого ключ співпадає зі шуканим ключем. Якщо знайдено відповідний запис, видаляє його за допомогою ітератора і повертає старе значення.
- replace(key: K, value: V): Замінює значення запису в словнику за вказаним ключем на нове значення. Якщо запис з таким ключем вже існує, встановлює нове значення і повертає попереднє значення. Якщо запису з таким ключем немає, повертає null.
- putIfAbsent(key: K, value: V): Додає новий запис до словника за вказаним ключем і значенням, якщо запису з таким ключем немає. Якщо запис з таким ключем вже існує, повертає його значення.

```
@Override
public V replace(K key, V value) {
    V curValue;
    if (((curValue = get(key)) ≠ null) || containsKey(key)) {
        curValue = put(key, value);
    }
    return curValue;
}

2 usages  5 overrides
@Override
public V putIfAbsent(K key, V value) {
    V v = get(key);
    if (v == null) {
        v = put(key, value);
    }

    return v;
}
```

Рис. 2.2 Приклад методів putIfAbsent і reaplace.

- computeIfAbsent(key: K, mappingFunction: Function<? super K, ? extends V>): Виконує обчислення значення за вказаним ключем, якщо запису з таким ключем немає в словнику. Використовує функцію mappingFunction для обчислення значення на основі ключа. Якщо обчислене значення не є null, додає новий запис з ключем і обчисленим значенням до словника і повертає обчислене значення. Якщо запис з таким ключем вже існує, повертає його значення.

- computeIfPresent(key: K, remappingFunction: BiFunction<? super K, ? super V, ? extends V>): Виконує обчислення нового значення за вказаним ключем і поточним значенням, якщо запис з таким ключем існує в словнику. Використовує функцію remappingFunction для обчислення нового значення на основі ключа і поточного значення. Якщо обчислене значення не є null, оновлює запис з новим значенням і повертає обчислене значення. Якщо запис з таким ключем не існує, повертає null.
- toString(): Повертає рядок, що представляє словник у вигляді {key1=value1, key2=value2, ...}. Проходиться по всім записам словника за допомогою ітератора і формує рядок, додаючи пару "ключ=значення" для кожного запису.

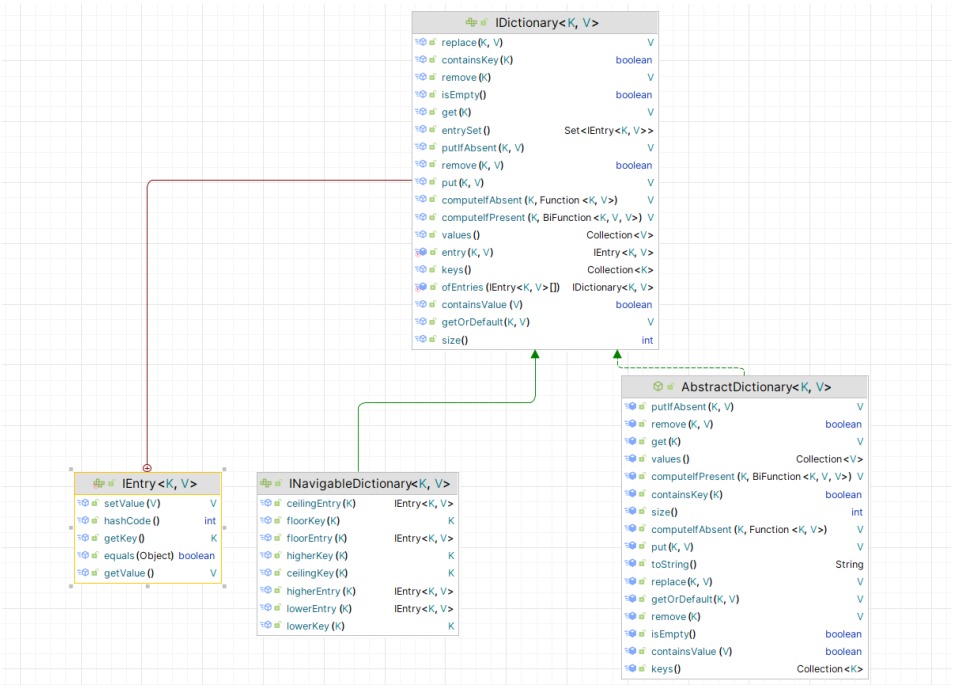

Рис. 2.3 Основні інтерфейси та абстрактний клас.

2.3 Види реалізацій словникових структур даних

ImmutableDictionary<K, V> (Імутабельний словник): Цей клас представляє реалізацію словника з довільною фіксованою кількістю елементів. Він є незмінним, це означає, що будь-які методи, які змінюють стан словника кидають помилку. Для збереження даних він використовує особливу методику хешування, яка гарантує унікальність ключів.

EnumDictionary<K, V> (Словник з використанням перелічувального типу): Цей клас реалізує словник, в якому ключі є перелічувальним типом.

Він надає методи для роботи зі словником, такі як отримання, додавання та видалення елементів, перевірка наявності ключа та значення, отримання розміру словника тощо. Його особливістю є швидкість роботи, із-за того, що максимальна кількість ключів відома, і кожен ключ має свій гарантовано унікальний індекс, операції зі словником відбуваються за константне O(1).

HashDictionary<K, V> (Хеш-словник): Цей клас реалізує словник, в якому внутрішні дані зберігаються за допомогою хеш-таблиці [3]. Він надає методи для отримання, додавання та видалення елементів, перевірки наявності ключа та значення, обчислення хеш-коду, отримання розміру словника тощо. Така загальна реалізація словника є найкращою, швидкість операцій O(1), але додається час потрібний для розширення словника. Для розв’язання проблеми колізій використовується техніка ланцюжкового хешування. Реалізація інтерфейсу IEntry<K, V> представлена внутрішнім класом Node<K, V>, який окрім збереження ключа і значення зберігає посилання на наступний елемент, саме завдяки цьому розв’язується проблема колізій і реалізується техніка ланцюжкового хешування.

IdentityHashDictionary<K, V> (Ідентичнісний хеш-словник): Цей клас схожий за реалізацією на HashDictionary, але використовує метод System.identityHashCode() для обчислення хеш-коду ключів. Він використовує ідентичність об'єктів для вирішення колізій при хешуванні. Іншими словами, в IdentityHashDictionary два ключі k1 і k2 вважаються рівними тоді і тільки тоді, коли (k1==k2). (У звичайних реалізаціях IDictionary (наприклад, HashDictionary) два ключі k1 і k2 вважаються рівними тоді і тільки тоді, коли (k1==null ? k2==null : k1.equals(k2)).) Цей клас не є реалізацією словника загального призначення. Хоча цей клас реалізує інтерфейс IDictionary, він навмисно порушує загальний договір словників, який зобов’язує використання методу equals() під час порівняння об’єктів. Цей клас призначений для використання лише в рідкісних випадках, коли потрібна семантика посилальної рівності.

MultiDictionary<K, V>(Мультисловник): Підклас AbstractDictionary, який представляє словник, де одному ключу може відповідати кілька значень [4]. Для реалізації буде використовуватися внутрішній словник

IDictionary<K, Set<V>>, який буде відповідати за збереження записів. Перевагами цієї реалізації є те, що вона бере на себе на себе відповідальність за управління сетами, які зберігають значення і користувачу не потрібно постійно створювати, перевіряти на існування, видаляти сети зі значеннями. Із-за таких властивостей методи у MultiDictionary працюють не так, як у звичайних реалізаціях словників, наприклад метод put(key: K, value: V) не затирає ключ, якщо він вже був присутнім у словнику, а додає значення до сету.

SortedArrayNavigableDictionary<K extends Comparable<K>, V>(Словник на основі відсортованого масиву): Цей клас реалізує інтерфейс INavigableDictionary. Він працює на основі відсортованого масиву, операції для отримання елементу працюють за O(log(n)), що є найкращим результатом для словника що реалізовує інтерфейс INavigableDictionary, але із-за того що масив має постійно знаходитися у відсортованому стані, операції ставки працюють дуже повільно в порівняння з наприклад TreeDictionary, який теж реалізує INavigableDictionary.

TreeDictionary<K extends Comparable<K>, V>(Словник на основі дерева): Цей клас реалізує інтерфейс INavigableDictionary і представляє відсортований словник [5]. Він реалізований за допомогою червоно-чорного дерева, це балансуюче себе бінарне дерево у якого кожна нода має колір, червоний або чорний. Операції вставок, видалення і пошуку працюють за O(log(n)), але операція видалення потребує виконання додаткових перевірок, які забезпечують збалансованість дерева. Реалізація інтерфейсу IEntry<K, V> представлена внутрішнім класом Node<K, V>, який окрім збереження ключа і значення зберігає посилання на правого і лівого нащадку та колір ноди, з яких і будується червоно-чорне дерево.

TrieDictionary<K, V>(Префіксний словник): Цей клас реалізує словник (dictionary) на основі структури даних Trie [6]. Він розширює абстрактний клас AbstractDictionary і реалізує інтерфейс IDictionary. Внутрішній клас TrieNode представляє вузол Trie. Він реалізує інтерфейс IEntry<String, V>, що означає, що кожен вузол Trie має ключ типу String та значення типу V. Збереження даних виконується за допомогою префіксів ключів. Загалом,

TrieDictionary забезпечує швидкий доступ до даних, особливо в ситуаціях, коли ключі мають спільний префікс. В порівнянні з іншими структурами даних, такими як хеш-таблиці, Trie часто виявляється ефективним для операцій пошуку, вставки та видалення зі словника.

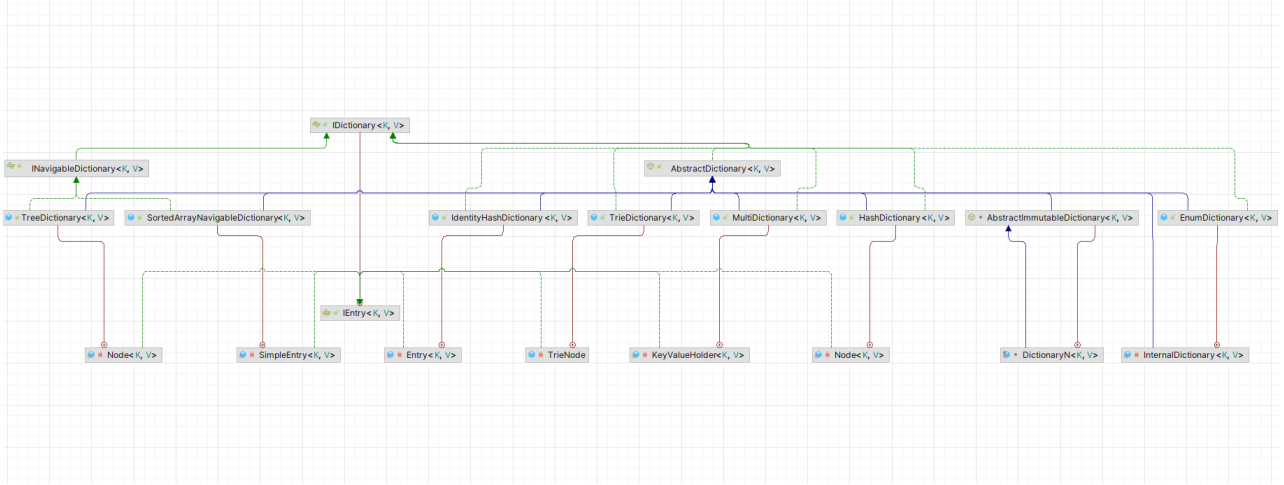

Рис 2.4 Спроектована ієрархія всіх класів.

Розділ 3. Реалізація словникових структур даних

3.1 Імутабельний словник

Клас AbstractImmutableDictionary<K, V> є абстрактним класом, який реалізує інтерфейс IDictionary<K, V> та розширює клас AbstractDictionary<K, V>. Цей клас надає базову реалізацію незмінного словника (dictionary) з методами доступу та маніпуляції даними.

Основні особливості цієї реалізації включають:

Клас AbstractImmutableDictionary<K, V> є абстрактним, тобто його не можна створити безпосередньо. Його нащадки повинні реалізувати незавершені методи та, за необхідності, можуть додати свою функціональність.

Клас AbstractImmutableDictionary<K, V> успадковує від AbstractDictionary<K, V>, що надає базову реалізацію інтерфейсу IDictionary<K, V>. Це забезпечує спільні методи та функціональність, яка може бути спільною для імутабельних словників.

Внутрішній клас DictionaryN<K, V> є конкретною реалізацією незмінного словника. Він розширює AbstractImmutableDictionary<K, V> і реалізує недостаючі методи.

Реалізація використовує масив table, в якому пари ключ-значення зберігаються по черзі. Кожний елемент масиву table відповідає ключу або значенню.

Для забезпечення швидкого доступу до елементів, використовується пробування (probing). Пробування визначає індекс у масиві, де потенційно може знаходитися елемент. Якщо елемент вже присутній в словнику, він може бути знайдений за допомогою цього індексу.

Метод probe(Object pk) відповідає за визначення індексу в масиві table для заданого ключа pk в реалізації незмінного словника. Основна мета методу probe полягає в тому, щоб забезпечити швидкий доступ до елементів, враховуючи особливості внутрішнього представлення словника.

Основні кроки алгоритму probe наступні:

- Визначення початкового індексу idx шляхом обчислення хеш-коду ключа pk. Це виконується за допомогою методу hashCode(). Значення хеш-коду зводиться до відповідного діапазону індексів за допомогою операції floorMod для забезпечення позитивного числа.
- Початковий індекс idx використовується для перевірки наявності ключа в словнику. Якщо елемент з таким індексом є нульовим, це означає, що ключ не знайдено, і від'ємне значення -(idx + 1) повертається для вказання позиції, в якій ключ міг би бути вставлений (при збереженні незмінного стану).
- Якщо елемент з індексом idx не є нульовим, виконується перевірка на рівність ключів. Якщо ключі рівні, повертається позитивне значення idx, що вказує на знайдений ключ.
- Якщо індекс idx не є останнім елементом масиву table, індекс збільшується на 2, щоб перейти до наступної пари ключ-значення. Якщо idx стає рівним довжині масиву table, він переходить на початок масиву (циклічний зсув).
- Повторюється процес пошуку, перевіряючи наявність наступних елементів. Цей процес продовжується до знаходження вільного місця для вставки ключа або знайдення збігу ключів.

```
private int probe(Object pk) {
    // guarantee that index would be positive number
    int idx = Math.floorMod(pk.hashCode(), table.length >> 1) << 1;
    while (true) {
        /unchecked/
        K ek = (K) table[idx];
        if (ek == null) {
            return -idx - 1;
        } else if (pk.equals(ek)) {
            return idx;
        } else if ((idx += 2) == table.length) {
            idx = 0;
        }
    }
}
```

Рис. 3.1 Метод probe для пошуку індексу.

Метод probe гарантує, що елементи словника зберігаються в масиві table з оптимальним розташуванням, що дозволяє швидкий доступ до них шляхом використання хеш-кодів та лінійного зондування для розв'язання конфліктів. Цей підхід забезпечує ефективну реалізацію операцій пошуку, вставки та вилучення в незмінному словнику.

Ключові методи, які змінюють словник (наприклад, put, remove, replace), кидають виключення UnsupportedOperationException. Це означає, що незмінний словник не підтримує модифікацію.

```
@Override
public V put(K key, V value) { throw uoe(); }

@Override
public V remove(K key) { throw uoe(); }

@Override
public V replace(K key, V value) { throw uoe(); }

2 usages
@Override
public V putIfAbsent(K key, V value) { throw uoe(); }

5 usages
@Override
public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) { throw uoe(); }
```

Рис. 3.2 Кидання помилки при спробі зміни стану словника.

Деякі методи, такі як containsKey, containsValue, get, size, isEmpty, надають стандартну реалізацію для перевірки присутності, отримання значень інших характеристик словника.

Метод entrySet повертає множину, яка містить всі пари ключ-значення в словнику.

Внутрішній клас KeyValueHolder<K, V> використовується для зберігання пари ключ-значення як окремого об'єкту.

Константа EMPTY_DICTIONARY представляє пустий словник із загальним типом ключа та значення Object. Це сінглтонний об'єкт, який можна використовувати для представлення порожнього словника.

Загальною метою цієї реалізації є забезпечення незмінності словника та ефективного доступу до даних. Методи, які зазвичай змінюють словник, викликають виключення, тим самим гарантуючи, що словник залишається незмінним.

3.2 Словник з використанням перелічувального типу

Клас EnumDictionary використовує внутрішній клас InternalDictionary, який є приватним імплементатором словника. Цей внутрішній клас використовує масив values для зберігання значень, пов'язаних з кожним ключем типу Enum. Індексація значень у масиві виконується за допомогою ordinal ключів, що дозволяє ефективно знаходити і отримувати значення за ключем.

Проведемо аналіз методу знаходження індексів у цього класу:

```
/unchecked/
@Override
public V get(K key) {
    if (keyType.isInstance(key)) {
        K enumKey = key;
        return (V) values[enumKey.ordinal()];
    }
    return null;
}
```

Рис. 3.3 Приклад методу get.

Метод get приймає ключ key і повертає відповідне значення V для цього ключа. В першу чергу, метод перевіряє, чи тип ключа key є підтипом keyType, за допомогою keyType.isInstance(key). Якщо це так, значить ключ належить до типу K, і виконується наступне:

Ключ key приводиться до типу K (представленого як enumKey).

Значення enumKey.ordinal() повертає позицію ключа enumKey в масиві keyUniverse. Значення values[enumKey.ordinal()] повертає відповідне значення для ключа enumKey. Якщо тип ключа key не відповідає типу K, метод повертає null, оскільки неможливо знайти відповідне значення.

Метод getOrDefault має подібну логіку:

```
/unchecked/
@Override
public V getOrDefault(K key, V defaultValue) {
    if (keyType.isInstance(key)) {
        K enumKey = key;
        V value = (V) values[enumKey.ordinal()];
        return (value ≠ null) ? value : defaultValue;
    }
    return defaultValue;
}
```

Рис. 3.4 Метод getOrDefault.

Метод containsKey перевіряє, чи ключ key належить до типу K і чи існує відповідне значення:

```
@Override
public boolean containsKey(K key) {
    if (keyType.isInstance(key)) {
        K enumKey = key;
        return (values[enumKey.ordinal()] ≠ null);
    }
    return false;
}
```

Рис. 3.5 Метод containsKey.

Метод containsValue перевіряє, чи існує в словнику значення value:

```
@Override
public boolean containsValue(Object value) {
    for (Object val : values) {
        if (Objects.equals(value, val)) {
            return true;
        }
    }
    return false;
}
```

Рис. 3.6 Метод containsValue.

Ці методи використовують масив values, де кожному ключу enumKey відповідає певне значення values[enumKey.ordinal()]. Це забезпечує швидкий доступ до значень за допомогою індексів ключів у масиві.

Таким чином, методи класу EnumDictionary використовують механізм ordinal для отримання позицій ключів у масиві values, що дозволяє ефективно отримувати та перевіряти значення в словнику.

3.3 Хеш-словник

Клас HashDictionary є реалізацією словника з використанням хеш-таблиці. Він реалізує інтерфейс IDictionary<K, V>, що визначає основні операції для роботи зі словником. Давайте розглянемо його основні властивості та методи:

Конструктори HashDictionary(), HashDictionary(int capacity): Конструктори створюють новий екземпляр HashDictionary. Перший конструктор встановлює початкову ємність словника за замовчуванням, а другий конструктор дозволяє вказати власну початкову ємність capacity. Якщо capacity менше або дорівнює 0, викидається виключення IllegalArgumentException.

Метод getIndex(K key): Цей приватний метод обчислює хеш-код ключа key та повертає індекс в масиві buckets, куди буде зберігатись відповідний вузол. Використовується функція Objects.hashCode() для отримання хеш-коду ключа, а потім використовується залишок від ділення на розмір масиву buckets.length, щоб отримати діапазон припустимих індексів.

```
private int getIndex(K key) {
    int hashCode = Objects.hashCode(key);
    return Math.abs(hashCode) % buckets.length;
}
```

Рис. 3.7 Метод getIndex.

Перевизначені методи isEmpty(), containsKey(K key), containsValue(V value), put(K key, V value), get(K key), getOrDefault(K key, V defaultValue), size(), remove(K key): Ці методи виконують стандартні операції, які повинен підтримувати словник. Вони перевіряють, чи словник порожній, чи містить певний ключ або значення, додають пару ключ-значення до словника, отримують значення за ключем, видаляють значення за ключем тощо.

Приватний метод resizeBuckets(): Цей метод збільшує розмір масиву buckets удвічі, коли відношення кількості елементів до загальної ємності перевищує задане значення LOAD_FACTOR. При збільшенні розміру масиву перехешовуються всі елементи, розподіляючи їх по новим індексам.

```java
private void resizeBuckets() {
    int newCapacity = buckets.length * 2;
    Node<K, V>[] newBuckets = new Node[newCapacity];
    for (Node<K, V> node : buckets) {
        while (node != null) {
            int newIndex = getIndex(node.key);
            Node<K, V> nextNode = node.next;
            node.next = newBuckets[newIndex];
            newBuckets[newIndex] = node;
            node = nextNode;
        }
    }
    buckets = newBuckets;
}
```

Рис. 3.8 Метод resizeBuckets.

Метод entrySet(): Цей метод повертає набір усіх записів (IEntry<K, V>) словника. Він перебирає всі вузли (Node<K, V>) в масиві buckets та додає їх до множини entrySet.

Перевизначені методи equals(Object o) і hashCode(): Ці методи перевіряють рівність двох словників на основі елементів, які вони містять. Вони порівнюють розмір словників та порівнюють значення для кожного ключа. Метод equals(Object o) також перевіряє, чи об'єкт o є екземпляром класу HashDictionary.

Клас HashDictionary також містить вкладений (приватний) клас Node<K, V>, який представляє вузол з ключем та значенням. Вузли утворюють зв'язаний список для кожного індексу в масиві buckets, що дозволяє розв'язувати колізії. Кожен вузол містить посилання на наступний вузол в ланцюжку. Клас Node<K, V> також реалізує інтерфейс IEntry<K, V>, який визначає методи для отримання ключа, значення та встановлення значення вузла.

```
private static class Node<K, V> implements IEntry<K, V> {
    11 usages
    private final K key;
    13 usages
    private V value;
    13 usages
    private Node<K, V> next;

    1 usage
    public Node(K key, V value) {
        this.key = key;
        this.value = value;
        this.next = null;
    }

    @Override
    public K getKey() { return key; }

    @Override
    public V getValue() { return value; }
```

Рис. 3.10 Внутрішній клас Node

Клас HashDictionary дозволяє ефективно зберігати та доступатись до елементів словника за допомогою хеш-таблиці. Це дозволяє швидко виконувати операції додавання, видалення та отримання елементів у середньому в константний час, оскільки часова складність цих операцій залежить від завантаженості хеш-таблиці.

3.4 Хеш-словник ідентичності

Клас IdentityHashDictionary представляє словник, який використовує хеш-таблицю з ідентичністю ключів. Основним принципом роботи цього словника є визначення індексу для кожного ключа і вирішення колізій.

Індекс для ключа визначається за допомогою методу getIndex(K key), який використовує System.identityHashCode(key), щоб отримати хеш-код ключа, і обчислює індекс шляхом використання побітового І-оператора (&) з маскою table.length - 1. Це забезпечує рівномірний розподіл ключів по різних індексах хеш-таблиці.

```
private int getIndex(K key) {
    return System.identityHashCode(key) & (table.length - 1);
}
```

Рис. 3.11 Метод getIndex з використанням ідентичності.

Колізії вирішуються за допомогою методу лінійного пробування. При вставці нового запису метод findEmptyIndex(K key) шукає наступний

порожній індекс, розпочинаючи з обчисленого початкового індексу. Він перевіряє кожний індекс у таблиці послідовно, здійснюючи кроки вправо ((index + 1) % table.length), поки не знайде порожній індекс або не повернеться до початкового індексу. Якщо таблиця повністю заповнена і немає порожніх індексів, викидається виняток IllegalStateException.

```
private int findEmptyIndex(K key) {
    int index = getIndex(key);
    int startIndex = index;
    do {
        if (table[index] == null) {
            return index;
        }
        index = (index + 1) % table.length;
    } while (index != startIndex);
    throw new IllegalStateException("Hash table is full.");
}
```

Рис. 3.12 Метод findEmptyIndex.

При пошуку запису за ключем метод findEntryIndex(K key) також використовує лінійне пробування. Він перевіряє кожний індекс у таблиці послідовно, переходячи вправо ((index + 1) % table.length), поки не знайде запис з таким самим ключем або не повернеться до початкового індексу.

```
private int findEntryIndex(K key) {
    int index = getIndex(key);
    int startIndex = index;
    do {
        Entry<K, V> entry = table[index];
        if (entry != null && entry.getKey() == key) {
            return index;
        }
        index = (index + 1) % table.length;
    } while (index != startIndex);
    return -1;
}
```

Рис. 3.13 Метод findEntryIndex

Цей підхід добре працює з об'єктами, у яких ідентичність визначається посиланням на пам'ять. Тому, якщо два ключі мають одне і те ж посилання, вони співпадатимуть при пошуку і вставці у хеш-таблицю. Отже, колізії вирішуються ефективно, забезпечуючи швидкий доступ до елементів за допомогою ідентичності ключів.

Також в класі є внутрішній приватний клас Entry, який представляє запис у словнику. Він має поля key (ключ) і value (значення) та методи доступу до них. Клас також перевизначає методи equals, hashCode і toString для коректної роботи з об'єктами типу Entry.

Цей клас реалізує основні принципи роботи хеш-таблиці з ідентичністю ключів, забезпечуючи швидкий доступ до елементів за допомогою хеш-функцій і уникнення колізій шляхом використання лінійного пробування. Ключові методи, такі як put, get, remove, ефективно працюють, забезпечуючи швидкий доступ до елементів у середньому за константний час (O(1)).

3.5 Словник на основі відсортованого масиву

Клас SortedArrayNavigableDictionary є реалізацією навігаційного словника, який зберігає дані у відсортованому масиві entries. Основна ідея полягає в тому, що словник підтримує швидкий пошук за ключем за допомогою бінарного пошуку.

```
private int binarySearch(K key) {
    return Collections.binarySearch(entries, new SimpleEntry<>(key, null),
            Comparator.comparing(IEntry::getKey));
}
```

Рис. 3.14 Бінарний пошук.

Основні методи та їх принцип роботи:

containsKey(K key): Використовує бінарний пошук для пошуку ключа в масиві entries. Повертає true, якщо ключ знайдений, і false в іншому випадку.

containsValue(V value): Перевіряє, чи міститься задане значення в словнику шляхом перебору всіх записів. Повертає true, якщо значення знайдено, і false в іншому випадку.

put(K key, V value): Використовує бінарний пошук для знаходження місця вставки нового запису або заміни існуючого запису з таким самим ключем. Записи зберігаються у відсортованому порядку за ключем.

get(K key): Використовує бінарний пошук для знаходження запису з заданим ключем. Повертає значення, пов'язане з ключем, або null, якщо ключ не знайдено.

remove(K key): Використовує бінарний пошук для видалення запису з заданим ключем. Повертає значення, пов'язане з ключем, або null, якщо ключ не знайдено.

entrySet(): Повертає множину усіх записів у словнику.

Асимптотична оцінка:

Пошук (containsKey, containsValue, get, floorEntry, ceilingEntry, lowerEntry, higherEntry, lowerKey, floorKey, ceilingKey, higherKey) має складність O(log n), де n - кількість записів у словнику. Це завдяки використанню бінарного пошуку.

```java
@Override
public IEntry<K, V> ceilingEntry(K key) {
    int index = binarySearch(key);
    if (index >= 0) {
        return entries.get(index);
    } else {
        index = -(index + 1);
        if (index < entries.size()) {
            return entries.get(index);
        }
    }
    return null;
}
```

Рис. 3.15 Приклад методу cielingEntry, який використовує бінарний пошук.

Вставка (put) та видалення (remove) мають складність O(n), де n - кількість записів у словнику. При вставці або видаленні потрібно здійснити зсув елементів масиву. Оскільки масив відсортований, це вимагає лінійного часу.

```java
@Override
public V put(K key, V value) {
    int index = binarySearch(key);
    if (index >= 0) {
        IEntry<K, V> entry = entries.get(index);
        V oldValue = entry.getValue();
        entry.setValue(value);
        return oldValue;
    } else {
        int insertIndex = -(index + 1);
        entries.add(insertIndex, new SimpleEntry<>(key, value));
        return null;
    }
}
```

Рис. 3.16 Приклад методу put.

entrySet() має складність O(1), оскільки просто повертає посилання на існуючий масив entries.

Загалом, SortedArrayNavigableDictionary надає швидкий доступ до даних за допомогою бінарного пошуку, але вставка та видалення можуть бути повільними для великих словників через потребу в зсуві елементів у масиві.

3.6 Словник на основі дерева

Клас TreeDictionary розширює абстрактний клас AbstractDictionary і реалізує інтерфейс INavigableDictionary. Це означає, що він успадковує функціональність з AbstractDictionary і надає додаткові методи та функції для навігації по словнику.

Клас TreeDictionary реалізований як червоно-чорне дерево, що забезпечує балансування під час вставки, видалення та інших операцій над словником. Червоно-чорне дерево - це самобалансуючеся бінарне дерево пошуку, де кожен вузол має додаткове поле кольору, що допомагає підтримувати балансування.

Основні поля класу TreeDictionary:

root: кореневий вузол дерева.

comparator: об'єкт типу Comparator, використовується для порівняння ключів. Якщо comparator має значення null, використовується порівняння ключів за замовчуванням (key1.compareTo(key2)).

size: кількість вузлів в дереві.

Клас TreeDictionary містить вкладений приватний клас Node, який представляє вузол дерева. Кожен вузол містить ключ, значення, посилання на лівого та правого дочірніх вузлів, кольору вузла (червоний або чорний) та кількість вузлів у піддереві з коренем в даному вузлі.

```
private static class Node<K, V> implements IEntry<K, V> {
    16 usages
    private K key;
    14 usages
    private V value;
    47 usages
    private Node<K, V> left;
    35 usages
    private Node<K, V> right;
    17 usages
    private boolean color;
    11 usages
    private int count;
```

Рис. 3.17 Клас Node.

Принципи балансування в дереві пошуку, такому як червоно-чорне дерево, гарантують, що висота дерева залишається приблизно збалансованою, що забезпечує ефективність операцій пошуку, вставки та видалення. Балансування відбувається за допомогою різних операцій, таких як лівий та правий повороти, зміна кольору та рекурсивні зміни в структурі дерева.

Основні принципи балансування в червоно-чорному дереві:

- Кожен вузол має призначений колір: червоний або чорний.
- Корінний вузол завжди є чорним.
- Всі листя (завершальні вузли) є чорними.
- Якщо вузол червоний, то обидва його дочірні вузли є чорними.
- На всіх шляхах від будь-якого вузла до його листя має бути однакова кількість чорних вузлів. Ця властивість називається "чорна висота" (black height).

Процес балансування відбувається під час вставки або видалення вузлів з дерева. Основні операції балансування в червоно-чорному дереві включають:

Лівий поворот (rotateLeft): Ця операція виконується, коли дерево незбалансоване вправо. Вона здійснюється шляхом зміни позицій вузлів, щоб підняти правого дочірнього вузла на рівень батьківського вузла, а лівий дочірній вузол стає новим коренем піддерева.

Правий поворот (rotateRight): Ця операція виконується, коли дерево незбалансоване вліво. Вона здійснюється аналогічно лівому повороту, але у протилежному напрямку.

Зміна кольору (flipColors): Ця операція виконується, коли обидва дочірні вузли червоні. Вона змінює кольори вузлів, щоб батьківський вузол став червоним, а дочірні вузли - чорними.

Рекурсивне оновлення лічильника (updateCount): Під час вставки або видалення вузлів, лічильник count вузлів оновлюється для відображення змін у розмірі піддерева.

```
private Node<K, V> put(Node<K, V> node, K key, V value) {
    if (node == null) {
        return new Node<>(key, value, RED);
    }

    int cmp = compare(key, node.key);
    if (cmp < 0) {
        node.left = put(node.left, key, value);
    } else if (cmp > 0) {
        node.right = put(node.right, key, value);
    } else {
        // Key already exists, update the value and return the previous value
        prevPut = node.value; //Set previous value
        node.value = value;
        return node;
    }

    //balancing magic
    if (isRed(node.right) && !isRed(node.left)) {
        node = rotateLeft(node);
    }
    if (isRed(node.left) && isRed(node.left.left)) {
        node = rotateRight(node);
    }
    if (isRed(node.left) && isRed(node.right)) {
        flipColors(node);
    }

    node.count = size(node.left) + size(node.right) + 1;
    return node;
}
```

Рис. 3.18 Метод put, який виконує балансування дерева після вставки

```
private Node<K, V> remove(Node<K, V> node, K key) {
    int cmp = compare(key, node.key);
    if (cmp < 0) {
        if (!isRed(node.left) && !isRed(node.left.left))
            node = moveRedLeft(node);
        node.left = remove(node.left, key);
    } else {
        if (isRed(node.left))
            node = rotateRight(node);
        if (compare(key, node.key) == 0 && (node.right == null))
            return null;
        if (!isRed(node.right) && !isRed(node.right.left))
            node = moveRedRight(node);
        if (compare(key, node.key) == 0) {
            Node<K, V> min = minimum(node.right);
            node.key = min.key;
            node.value = min.value;
            node.right = deleteMin(node.right);
        } else {
            node.right = remove(node.right, key);
        }
    }
    return balance(node);
}
```

Рис. 3.19 Метод remove який виконує балансування дерева після виконання.

Ці операції застосовуються відповідно до правил та умов червоно-чорного дерева для збереження балансу та властивостей дерева. Комбінація цих операцій гарантує, що дерево залишається збалансованим після вставки або видалення вузлів, зберігаючи ефективність пошуку та інших операцій.

3.7 Префіксний словник

Клас TrieDictionary розширює абстрактний клас AbstractDictionary і реалізує інтерфейс IDictionary. Основними принципами роботи цього класу є наступне:

Структура TrieDictionary використовує TrieNode як основний елемент для зберігання даних. Клас TrieNode містить словник children, який зберігає посилання на дочірні TrieNode, а також значення value, яке асоціюється з кінцевим вузлом (термінальним символом).

Метод put(K key, V value) додає елемент в TrieDictionary. Ключ key перетворюється в рядок і передається методу put TrieNode. Метод рекурсивно пройшовши через вузли, створює нові TrieNode для символу, якщо він не існує, і продовжує обробку решти рядка ключа. Значення value зберігається в останньому вузлі.

```
public V put(String key, V value) {
    if (key.isEmpty()) {
        V oldValue = this.value;
        this.value = value;
        return oldValue;
    }

    char firstChar = key.charAt(0);
    TrieNode child = children.get(firstChar);
    if (child == null) {
        child = new TrieNode();
        children.put(firstChar, child);
    }

    return child.put(key.substring(1), value);
}
```

Рис. 3.20 Метод put з використанням префіксу.

Метод get(K key) повертає значення, пов'язане з ключем key. Він також рекурсивно пройшовши через вузли, перевіряє наявність символу у дітей вузла і продовжує пошук по решті рядка ключа. Повертає значення, збережене в останньому вузлі.

```
public V get(String key) {
    if (key.isEmpty()) {
        return this.value;
    }

    char firstChar = key.charAt(0);
    TrieNode child = children.get(firstChar);
    if (child == null) {
        return null;
    }

    return child.get(key.substring(1));
}
```

Рис. 3.21 Метод get з рекурсивним викликом і використанням префіксу.

Метод remove(K key) видаляє елемент, пов'язаний з ключем key. Він також рекурсивно пройшовши через вузли, перевіряє наявність символу у дітей вузла і продовжує пошук по решті рядка ключа. Після видалення значення вузла, перевіряється, чи він порожній (не має дітей та значення), і у такому випадку видаляється зі словника.

Метод entrySet() повертає набір елементів, що складаються з ключа та значення, усіх елементів TrieDictionary. Він рекурсивно проходиться через TrieNodes і збирає пари ключ-значення в набір entrySet.

Цей клас має також реалізовані методи toString(), equals(Object obj) та hashCode(), які дозволяють використовувати TrieDictionary у контексті порівняння та виводу.

Особливістю TrieDictionary є те, що він забезпечує швидкий доступ до даних, зокрема пошук і видалення, за рахунок використання префіксного дерева. Це особливо корисно, коли потрібно здійснювати операції над словником зі швидкістю, що залежить від довжини ключа. TrieDictionary також ефективно використовує пам'ять, оскільки спільні префікси у ключах зберігаються в одних вузлах, зменшуючи кількість необхідних вузлів.

3.8 Приклади поліморфізму

Для того, щоб показати поліморфізм у код спочатку ініціалізуємо словники, використовуючи саме їх клас, а не інтерфейс або абстрактний клас.

```
//Dictionary initialization
HashDictionary<Number, String> hashDictionary = new HashDictionary<>();
IdentityHashDictionary<Number, String> identityHashDictionary = new IdentityHashDictionary<>();
MultiDictionary<Number, String> multiDictionary = new MultiDictionary<>();
TrieDictionary<Number, String> trieDictionary = new TrieDictionary<>();
TreeDictionary<Double, String> treeDictionary = new TreeDictionary<>();
SortedArrayNavigableDictionary<Double, String> sortedArrayNavigableDictionary = new SortedArrayNavigableDictionary<>();
```

Рис. 3.22 Ініціалізація словників

Тепер створюємо метод, який прийматиме параметром інтерфейс IDictionary і буде заповнювати масив значеннями. У цьому прикладі використовується метод put інтерфейсу, але під час виконання коду замість нього підставляється реалізація цього методу з нащадку.

```
/unchecked/
public static <K extends Number, V extends CharSequence> IDictionary<K, V> fillDictionary(
        IDictionary<K, V> dictionary,
        int quantity
) {
    for (int i = 0; i < quantity; i++) {
        K key = (K) Double.valueOf(random.nextDouble());
        V value = (V) randomAlphabeticString(4);
        dictionary.put(key, value);
    }

    return dictionary;
}
```

Рис. 3.23 Поліморфний метод fillDictionary

У результаті ми можемо передати всі словники у метод та заповнити їх значеннями, хоча ми явно не вказуємо, конкретну реалізацію у методі.

```
//IDictionary polymorphism
fillDictionary(hashDictionary, 10);
fillDictionary(identityHashDictionary, 10);
fillDictionary(multiDictionary, 10);
fillDictionary(trieDictionary, 10);
fillDictionary(treeDictionary, 10);
fillDictionary(sortedArrayNavigableDictionary, 10);
```

Рис. 3.24 Поліморфізм IDictionary

Тепер розглянемо поліморфізм для INavigatbleDictionary, створимо методи, які буде приймати вхідним значенням цей інтерфейс і ключ, і повертатимуть вищий і нижчий ключ відповідно.

```
public static <K, V> K lowerKey(
        INavigableDictionary<K, V> dictionary,
        K key
) {
    return dictionary.lowerKey(key);
}

2 usages
public static <K, V> K higherKey(
        INavigableDictionary<K, V> dictionary,
        K key
) {
    return dictionary.higherKey(key);
}
```

Рис. 3.25 Поліморфні методи для INavigableDictionary

У метод ми передаємо інстанси конкретних класів під виглядом інтерфейсу, у яких потім і викликаються методи.

```
//INavigableDictionary polymorphism
double treeLowerKey = lowerKey(treeDictionary, 0.5);
double arrayLowerKey = lowerKey(sortedArrayNavigableDictionary, 0.5);

double treeHigherKey = higherKey(treeDictionary, 0.5);
double arrayHigherKey = higherKey(sortedArrayNavigableDictionary, 0.5);
```

Рис. 3.26 INavigableDictionary поліморфізм

Останній приклад поліморфізму з абстрактним класом, ми можемо звертатися до його реалізацій методів, навіть якщо у нащадку вони не реалізовані.

```
// Abstract methods example,
// note that the methods are not implemented in the inheritor
IDictionary<Integer, String> dictionary = new TreeDictionary<>();
final int ten = 10;
final int five = 5;
//put 10="ten" | abstract method call
dictionary.putIfAbsent(ten, String.valueOf(ten));
//change 10="ten" to 10="11" | abstract method call
dictionary.computeIfPresent(ten, (key, value) -> String.valueOf(++key));
//put 5="5" | abstract method call
dictionary.computeIfAbsent(five, Object::toString);
//set 5="10" if value for 5 is absent | abstract method call | fail
dictionary.computeIfAbsent(five, key -> String.valueOf(ten));
//remove 5="10"
dictionary.remove(five);
//set 5="10" if value for 5 is absent | abstract method call | complete
dictionary.computeIfAbsent(five, key -> String.valueOf(ten));

dictionary.put(1, "one");
//replace 1="one" to 1="1" | abstract method call
dictionary.replace(1, "1");
```

Рис. 3.27 AbstractDictionary поліморфізм

Висновок

В ході даної роботи були вивчені та проаналізовані поліморфні реалізації словникових структур даних з використанням інтерфейсів, наслідування та шаблонів програмування. Різні підходи до поліморфних реалізацій були досліджені з метою виявлення їх переваг та недоліків.

У результаті були реалізовані різноманітні класи словникових структур даних. Інтерфейси IDictionary і INavigableDictionary були використані для визначення основних методів, що повинні бути реалізовані в класах словників. Абстрактний клас AbstractDictionary надав загальну реалізацію деяких методів та можливість перевизначення логіки методів у нащадках. Крім того, були реалізовані конкретні класи словників, такі як ImmutableDictionary, EnumDictionary, HashDictionary, IdentityHashDictionary, MultiDictionary, SortedArrayNavigableDictionary і TreeDictionary, кожен з яких має свої особливості та переваги.

Загальний висновок полягає в тому, що вивчення та аналіз поліморфних реалізацій словникових структур даних було успішним і дозволило виявити різноманітні підходи та їх особливості. Кожен з реалізованих класів має свої переваги та обмеження, і вибір конкретного класу залежить від потреб і вимог конкретної задачі.

Список літератури та використаних джерел

1. Wikipedia Hash table [електронний документ] https://en.wikipedia.org/wiki/Hash_table.
2. Red-black tree [електронний документ] https://cs.kangwon.ac.kr/~leeck/file_processing/red_black_tree.pdf.
3. Hash dictionary [електронний документ] https://www.cs.cmu.edu/~15122/handouts/13-hdict.pdf.
4. Multi map [електронний документ] https://learn.microsoft.com/en-us/cpp/standard-library/multimap-class?view=msvc-170.
5. Sorted map ст. 792(778) §23.4.4 [електронний документ] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
6. Wikipedia Trie data structure [електронний документ] https://en.wikipedia.org/wiki/Trie.