Практическая работа №27
Тема: Изучение Java Collection Framework: нелинейные структуры данных

Задания (выполняются самостоятельно)

Программные примеры

Класс HashSet

HashSet реализует интерфейс Set на основе хешей.

HashSet не поддерживает порядок своих элементов. Следовательно, сортировка HashSet невозможна. тем не менее, элементы HashSet могут быть отсортированы косвенно, путем преобразования в List или TreeSet, но это сохранит элементы в новом целевом типе вместо типа HashSet.

Ниже приведена реализация вышеуказанного подхода:

```
 import java.util.*;
public class ExampleSortSet {
    public static void main(String args[])
    {
        // создаем HashSet
        HashSet<String> set = new HashSet<String>();

        // добавляем элементы в HashSet используя add()
        set.add("geeks");
        set.add("practice");
        set.add("contribute");
        set.add("ide");

        System.out.println("Original HashSet: "
                           + set);
  // сортировка HashSet с использованием List
        List<String> list = new ArrayList<String>(set);
        Collections.sort(list);

        // печатаем отсортированные элементы HashSet
        System.out.println("HashSet elements "
                           + "in sorted order "
                           + "using List: "
                           + list);
    }
}
```

Класс HashSet реализует интерфейс Set, поддерживаемый хеш-таблицей, которая на самом деле является экземпляром HashMap. Не гарантируется порядок итераций множества, что означает, что класс не гарантирует постоянный порядок элементов во времени. Этот класс разрешает иметь элемент с нулевыми ссылками. Класс также обеспечивает постоянную производительность по времени для основных операций, таких как добавление, удаление, удержание и размер, при условии, что хеш-функция правильно распределяет элементы между сегментами памяти.

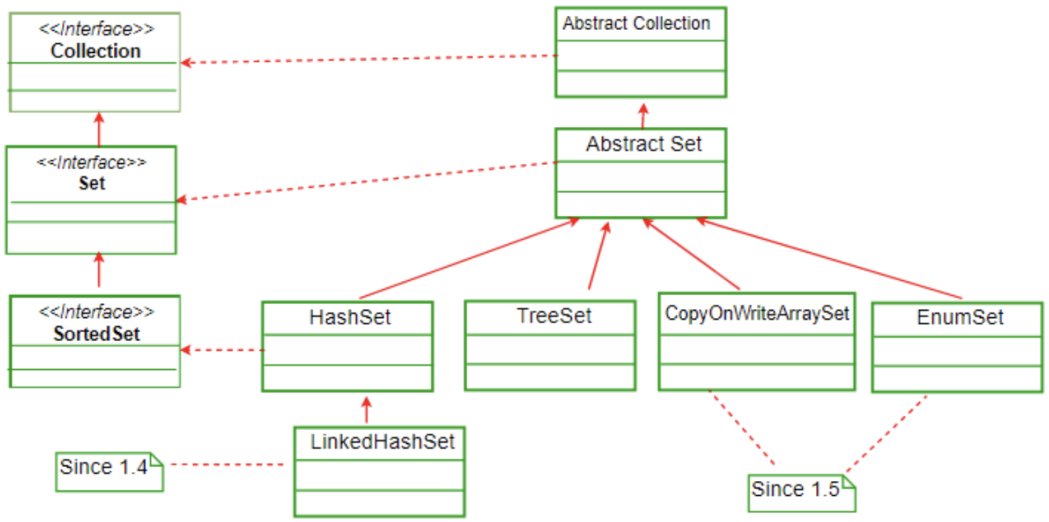
Класс реализует интерфейс Set.

Базовая структура данных для HashSet является хеш-таблицей.

Поскольку он реализует интерфейс Set, то повторяющиеся значения не допускаются. Объекты, которые вы вставляете в HashSet, не обязательно будут вставлены в том же порядке в каком вы их добавляете. Объекты всегда вставляются на основе их хэш-кода. Элементы NULL разрешены в HashSet.

Замечание: класс HashSet также реализует интерфейсы Searlizable и Cloneable.

Несколько важных омсобеннойстей HashSet:



Во-первых, для поддержания постоянной производительности по времени итерация по HashSet требует времени, пропорционального сумме размера экземпляра HashSet (количеству элементов) плюс «емкость» резервного экземпляра HashMap (количество сегментов). Таким образом, очень важно не устанавливать слишком высокую начальную емкость (или слишком низкий коэффициент загрузки), если важна производительность по итерациям.

Во-вторых: Реализация в HashSet не синхронизирована, в том смысле, что произойдет, если несколько потоков одновременно получают доступ к хэш-набору и хотя бы один из потоков изменяет набор, он должен быть синхронизирован извне. Обычно это достигается путем синхронизации с некоторым объектом, который естественным образом инкапсулирует набор. Если такого объекта не существует, набор следует «обернуть» с помощью метода Collections.synchronizedSet. Это лучше всего делать во время создания коллекции, чтобы предотвратить случайный несинхронизированный доступ к набору, как показано ниже:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Класс TreeSet

TreeSet реализует интерфейс Set на основе красно — черного дерева, что позволяет получить хорошую скорость операций.

В чем отличие между этими классами? Hashset не гарантирует время работы операций, но в среднем выполняет операцию вставки за константу, а TreeSet гарантирует выполнение любой операции пропорционально логарифму числа операций в нем. Используя интерфейс Set вы с легкостью сможете переключаться между этими классами, выбирая наиболее подходящий для вашей задачи.

К элементам массива можно обращаться к массиву не только через индекс, а по например по некторому ключу. Допустим у нас есть студент, мы знаем про него много информации, но все данные про студентов хранятся в массиве, тогда достать информацию из мы массива можем только по его номеру, поэтому нам приходиться просматривать весь массив, если мы знаем например только его имя или что — то еще о нем.

Иногда при разработке кода встают задачи стают задачи, при решении которых тривиальное решение это плохое решение и нужно искать более эффективные способы. В качестве примера можно привести ситуацию, когда, допустим мы хотим обращаться к элементу коллекции (организованный способ хранения) не только по целочисленному значению, а по любому ключу. Например, мы хотим просматривать всех пользователей по их логину, а не по некоторому номеру.

Очевидно, что логин — это строка, а мы знаем, что при работе со строками не может быть более-менее нормально обеспечен доступ к информации быстрее чем за O(n), если мы будем хранить данные в массиве. То есть тогда нужно искать другое решение. Вот тогда нам на помощь приходит ассоциативный массив или словарь.

Во-первых, ассоциативный массив удобен тем, что может работать с любым ключом. Не важно, что мы там будем хранить, например какое-то сложные объекты или строки.

Во-вторых, мы не будем осуществлять хранение никакой избыточной информации, у нас будут только реально существующие элементы, в нашем массиве.

Итак, например мы могли бы создать достаточно большой массив, брать хеш код у любого объекта и по нему класть этот объект в массив. Все это обеспечит бы нам работу с любыми ключами, так как известно, что хеш функцию с тем или иным успехом можно составить для любого объекта. Но как известно, при работе именно с массивом появляется другая проблема — это пустые места в массиве.

После прочтения выше написанного, у вас, возможно, сложилось мнение, что ассоциативные массивы это круто и видимо решают достаточно большой класс задач, но на самом деле не все так просто. В

зависимости от реализации ассоциативные массивы выполняют операции за время отличное от константы. Иногда их нужно применять, а иногда нет. Все зависит от решаемой задачи.

Рассмотрим интерфейс для работы с ассоциативными массивами.

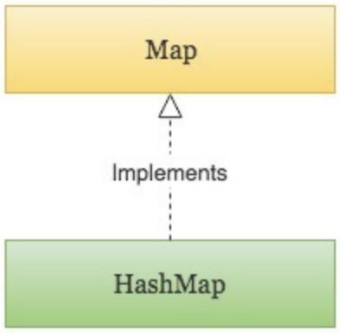
Интерфейс Map

При создании Интерфейса Map указывают тип ключа и тип хранимого значения:

```
Map<Integer, String> map;//Map<K,V>
```

Тип данных ключа будем обозначать K, а значения V.

- size() — возвращает количество элементов
- containsKey(Object key) — проверяет наличие ключа
- containsValue(Object value) — проверяет наличие значения в ассоциативном массиве
- get(Object key) — возвращает значение по ключу
- put(K key, V value) — кладет в ассоциативный массив значение value по ключу key. В случае наличия уже такого ключа происходит замена значения.
- values() — возвращает значения всех элементов в виде коллекции
- remove(Object key) — удаляет элемент с ключом key, возвращая значение этого элемента(вернет null в случае отсутствия)
- clear() — удаляет все элементы из массива
- isEmpty() — возвращает не пуст ли массив



Класс Hashtable

Класс Hashtable представляепт реализацию интерфейса Map, основанную на хеш-таблицах. Ниже приведена информация об этом классе:

```
public class Hashtable<K,V>
extends Dictionary<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Он не гарантирует какое-то определенное время выполнения, но в среднем работает за O(1). Тут следует сказать, что хеш-таблицы имеют два важных параметра: capacity (максимальное количество элементов) и loadFactor (загруженность таблицы). При достижении количества реально существующих элементов в размере 0,75 от максимального происходит автоматическое увеличение максимального количество элементов, т.е самой таблицы. 0,75 это показатель по умолчанию, конечно вы можете изменить загруженность таблицы:

```
Map<Integer, String> map = new Hashtable<Integer, String>(100, (float) 0.5);
//создаем таблицу с максимальным размером в 100 элементов
//и с содержимым 0.5
```

Кроме этого, существует класс HashMap, который имеет схожую функциональность, но при работе с потоками его лучше не использовать.

Класс TreeMap

Класс TreeMap представляет реализацию интерфейса Map, основанную на деревьях. Как и в любом дереве операции взятия элемента по ключу, вставка или удаления происходят за O(logn). Пример использования TreeMap:

```
Map<String, String> map = new TreeMap<String, String>();
```

С основными реализациями ассоциативного массива мы закончили, давайте попробуем реализовать пример про студентов.

Мы уже написали несколько программ, с использованием класса Student, например писали реализацию сортировки студентов по среднему баллу:

Допустим, что у нас есть некоторый класс Student:

```
package ru.mirea;
public class Student {
   private int number;//номер студенческого билета
   private String name;//имя студента
   private int age;//возраст

   @Override  //переопределяем метод equals
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;

      Student student = (Student) o;
```

```
        if (age != student.age) return false;
        if (number != student.number) return false;
        if (name != null ? !name.equals(student.name) : student.name != null) return false;
        return true;
    }

    @Override
    public int hashCode() {//переопределяем метод hashCode
        int result = number;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        result = 31 * result + age;
        return result;
    }
public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Student(int number, String name, int age) {
        this.number = number;
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return "Student{" +
                "number=" + number +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

Замечание: напоминаю вам, для того, чтобы пользоваться автоматической генерацией кода нажмите одновременно alt+insert(для ОС WIN, для MAC Command+N) а затем выберите объект, который хотите генерировать, кроме стандартных get() и set(), а также конструктора, добавьте equals, hashCode() и toString(). Эта процедура может понадобиться для того, чтобы использовать hashTable, дело в том, что для генерации хеша, данный класс использует вызов стандартной функции hashCode(), которая по умолчанию для пользовательских объектов (унаследованных от класса object) возвращает адрес, т.е возникает такая же ситуация, как и со строками, если их сравнивать через операцию ==.

Напишем класс Test, в котором просто создадим ассоциативный массив, и попробуем добавить туда некоторое количество элементов, а затем будем пытаться получать студентов по их имени:

```
package ru.mirea;

import java.util.Hashtable;
import java.util.Map;

public class Test {
   public void test()
   {
      Map<String, Student> map = new Hashtable<String, Student>();
      Student st = new Student(0,"Alex", 18);
      map.put("Alex", st);//добавляю студента Alex по ключу Alex
      System.out.println(map.get("Alex"));//работает
      System.out.println(map.get("Al" +"ex"));  //работает
      System.out.println(map.get(st.getName()));//работает
      String s = "a";//пытаюсь обмануть компилятор
      s = s.toUpperCase()+"lex";
      System.out.println(map.get(s)); //работает
   }
}
```

Задания для выполнения:

1. Преобразовать структуру данных HashSet в структуру TreeSet
2. Создайте класс Map<String, String> и добавьте десять записей, которые представляют пары (фамилия, имя). Проверьте, сколько людей имеют одинаковое имя или фамилию. Требования к программе:
   a) Программа не должна отображать текст на экране.
   b) Программа не должна считывать значения с клавиатуры

c) Метод createMap () должен создать и вернуть HashMap, который имеет элементы (String, String) и содержит 10 записей, представляющих пары (фамилия, имя).
d) Метод getSameFirstNameCount () должен возвращать количество людей с одинаковыми именами.
e) Метод getSameLastNameCount () должен возвращать количество людей с одинаковой фамилией

3. Реализовать хеш-таблицу, в которой ключи имеют вещественные значения. Пример хеш-функции приведен в Sedgewick2001, с. 575.

Материал для чтения:

1. https://ru.wikipedia.org/wiki/Хеш-таблица
2. https://www.codeflow.site/ru/article/java-hash-table
3. http://w ww.javable.com/tutorials/fesunov/lesson12/