Министерство науки и высшего образования РФ

ФГАОУ ВО «Уральский федеральный университет

имени первого Президента России Б. Н. Ельцина»

ИРИТ-РТФ

Центр ускоренного обучения

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1**

по дисциплине «Технологии программирования»

**Тема:** Исследование возможностей коллекций в языке программирования Java

Студент группы РИЗ-200028у: В.А. Кудрявцев

Преподаватель: Н.А. Архипов,

ст. преподаватель

Екатеринбург 2021

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc88330545)

[2 Описание работы 4](#_Toc88330546)

[2.1. Общее описание коллекций в Java 4](#_Toc88330547)

[2.2 Описание коллекций в соответствии с вариантом 7](#_Toc88330548)

[2.3 Сравнение коллекций в соответствии с вариантом 18](#_Toc88330549)

[3 Заключение 19](#_Toc88330550)

1. Постановка задачи

Цель: Исследование возможностей коллекций в языке программирования

Java, сравнение коллекций по времени выполнения операций, знакомство с

реализацией коллеккций.

Учебные вопросы:

1. Общее описание коллекций в Java;

2. Описание коллекций в соответствии с вариантом;

3. Сравнение коллекций в соответствии с вариантом;

4. Ответы на вопросы.

Вариант №2

Обзор коллекций HashSet , ArrayList, SortedSet

1. Описание работы

2.1. Общее описание коллекций в Java

****В пакете java.util**** содержится библиотека коллекций(****collection framework****), которая предоставляет большие возможности для работы с множествами, хэш-таблицами, векторами, разными видами списков и т.д.

****Коллекция**** - это объект, способный хранить группу одинаковых элементов. Она содержит методы для операций с однородными данными. Изначально Java поддерживала работу с коллекциями в рамках классов ****Vector**** и ****Hashtable****, но с появлением JDK 1.2 возможности работы с коллекциями были расширены, возникло много открытых интерфейсов и различных видов классов, которые были включены в библиотеку коллекций.

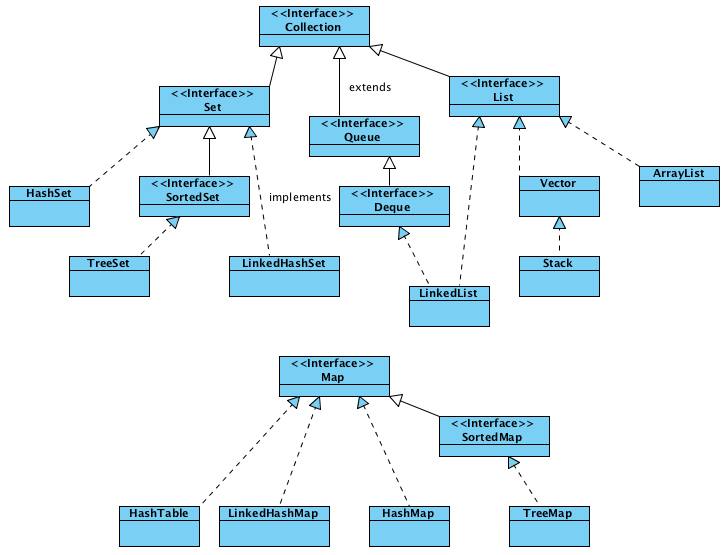


Рисунок 1 – Иерархия коллекций Java

Основные преимущества классов **collection framework**(перед классами, разрабатываемыми самостоятельно) заключаются в следующем:

1. ускоряется процесс разработки и улучшается качество кода;
2. обеспечивается поддержка повторного использования кода;
3. производится стандартизация интерфейса ваших классов;
4. реализуется поддержка многопоточного доступа.

Проверке готовых классов-коллекций уделялось много внимания, поэтому правомерно говорить об улучшении качества кода. Данные классы можно считать хорошо отлаженными с минимальным количеством ошибок, что зачастую невозможно обеспечить в самостоятельных проектах.

Основу библиотеки составляют открытые интерфейсы, которые можно использовать для создания собственных коллекций. Каждый интерфейс объявляет набор методов, которые вы обязаны реализовать в своей программе:

Таблица 1. Интерфейсы в Java

|  |  |
| --- | --- |
| Интерфейс | Описание |
| **Collection** | общий интерфейс, объединяющий интерфейсы Set и List. Содержит методы для добавления и удаления элементов коллекции, проверки их правильности, наличия и другие |
| **Set** | множество элементов(без дублирования), неупорядоченный набор неповторяющихся элементов, расширяет интерфейс Collection. Если производится попытка добавить в набор элемент, который уже в нем содержится, она будет проигнорирована |
| **SortedSet** | то же самое, что Set, только элементы упорядочены |
| **List** | упорядоченный список, служит для работы с упорядоченными коллекциями. К каждому элементы такой коллекции можно обратиться по индексу. Расширяет интерфейс Collection. |
| **Map** | словарь, то есть коллекция, в которой каждый элемент имеет уникальный ключ, предназначен для работы с коллекциями-словарями, в которых содержатся ключи и соответствующие им значения(каждому ключу соответствует только одно значением). Словарь может содержать произвольное число элементов. |
| **SortedMap** | то же самое, что и Map, однако элементы упорядочены |
| **Queue** | интерфейс для работы с очередью, содержит методы для работы с очередями: в них элементы добавляются с одного конца, а извлекаются с другого |

Разумеется, интерфейсы были бы "пустыми", если бы в них не существовало встроенных классов, реализующих необходимые функции:

Таблица 2. Встроенные классы

|  |  |
| --- | --- |
| **Встроенный класс** | **Функции** |
| **ArrayList** | список List как массив элементов |
| **LinkedList** | список List, выполняющий функции связанного списка |
| **HashSet** | множество Set как хэш-таблица |
| **TreeSet** | множество SortedSet, используемое как дерево |
| **HashMap** | индексированный словарь хэш |
| **TreeMap** | коллекция SortedMap древовидной структуры |

Задача каждого из интерфейсов - обеспечить простоту и удобство работы с большим количеством однотипных данных.

2.2 Описание коллекций в соответствии с вариантом

1. HashSet

[HashSet](http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html) — реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (new Object()). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.

Интерфейс Set расширяет интерфейс Collection и представляет набор уникальных элементов. Set не добавляет новых методов, только вносит изменения унаследованные. В частности, метод add() добавляет элемент в коллекцию и возвращает true, если в коллекции еще нет такого элемента.

Обобщенный класс HashSet представляет хеш-таблицу. Он наследует свой функционал от класса AbstractSet, а также реализует интерфейс Set.

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

Для создания объекта HashSet можно воспользоваться одним из следующих конструкторов:

Таблица 3. Конструкторы в HashSet

|  |  |
| --- | --- |
| Конструктор | Действия |
| HashSet(): | создает пустой список |
| HashSet(Collection<? extends E> col): | создает хеш-таблицу, в которую добавляет все элементы коллекции col |
| HashSet(int capacity) | параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна 16 |
| HashSet(int capacity, float koef) | параметр koef или коэффициент заполнения, значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение. Например, коэффициент 0.75 указывает, что при заполнении емкости на 3/4 произойдет ее расширение. |

Класс HashSet не добавляет новых методов, реализуя лишь те, что объявлены в родительских классах и применяемых интерфейсах.

Таблица 4. Методы в HashSet

|  |  |
| --- | --- |
| Метод | Описание |
| **boolean add(E e)** | добавляет элемент в HashSet, если таковой отсутствует, если же такой элемент уже присутствует, метод возвращает false |
| **void clear()** | удаляет все элементы из множества |
| **boolean contains(Object o)** | Возвращает true, если данный элемент присутствует в множестве |
| **boolean remove(Object o)** | удаляет данный элемент из множества, если таковой присутствует. |
| **Iterator iterator()** | возвращает итератор для элементов множества |
| **boolean isEmpty()** | возвращает true, если в множестве нет элементов |
| **Object clone()** | выполняет поверхностное клонирование HashSet |

HashSet поддерживается с помощью экземпляра HashMap. В HashSet элементы не упорядочены, нет никаких гарантий, что элементы будут в том же порядке спустя какое-то время. Операции добавления, удаления и поиска будут выполняться за константное время при условии, что хэш-функция правильно распределяет элементы по «корзинам».

Несколько важных пунктов о HashSet:

Так как класс реализует интерфейс Set, он может хранить только уникальные значения;

1. Может хранить NULL – значения;
2. Порядок добавления элементов вычисляется с помощью хэш-кода;
3. HashSet также реализует интерфейсы Serializable и Cloneable.

Для поддержания постоянного времени выполнения операций время, затрачиваемое на действия с HashSet, должно быть прямо пропорционально количеству элементов в HashSet + «емкость» встроенного экземпляра HashMap (количество «корзин»). Поэтому для поддержания производительности очень важно не устанавливать слишком высокую начальную ёмкость (или слишком низкий коэффициент загрузки).

Начальная емкость – изначальное количество ячеек («корзин») в хэш-таблице. Если все ячейки будут заполнены, их количество увеличится автоматически.

Коэффициент загрузки – показатель того, насколько заполненным может быть HashSet до того момента, когда его емкость автоматически увеличится. Когда количество элементов в HashSet становится больше, чем произведение начальной емкости и коэффициента загрузки, хэш-таблица ре-хэшируется (заново вычисляются хэшкоды элементов, и таблица перестраивается согласно полученным значениям) и количество ячеек в ней увеличивается в 2 раза. Коэффициент загрузки равняется частному от количества хранимых элементов в таблице и размера хэш-таблицы.

Например, если изначальное количество ячеек в таблице равно 16, и коэффициент загрузки равен 0,75, то из этого следует, что когда количество заполненных ячеек достигнет 12, их количество автоматически увеличится.

Коэффициент загрузки и начальная емкость – два главных фактора, от которых зависит производительность операций с HashSet. Коэффициент загрузки, равный 0,75, в среднем обеспечивает хорошую производительность. Если этот параметр увеличить, тогда уменьшится нагрузка на память (так как это уменьшит количество операций ре-хэширования и перестраивания), но это повлияет на операции добавления и поиска. Чтобы минимизировать время, затрачиваемое на ре-хэширование, нужно правильно подобрать параметр начальной емкости. Если начальная емкость больше, чем максимальное количество элементов, поделенное на коэффициент загрузки, то никакой операции ре-хэширования не произойдет в принципе.

Важно учитывать, что HashSet не является структурой данных с встроенной синхронизацией, поэтому если с ним работают одновременно несколько потоков, и как минимум один из них пытается внести изменения, необходимо обеспечить синхронизированный доступ извне. Часто это делается за счет другого синхронизируемого объекта, инкапсулирующего HashSet. Если такого объекта нет, то лучше всего подойдет метод Collections.synchronizedSet(). На данный момент это лучшее средство для предотвращения несинхронизированных операций с HashSet.

Листинг 1. Пример кода работы коллекции HashSet

import java.util.\*;

class Test

{

public static void main(String[]args)

{

HashSet<String> h = new HashSet<String>();

// Добавляем элементы в HashSet с помощью метода add()

h.add("Индия");

h.add("Австралия");

h.add("Южная Африка");

h.add("Индия");// пытаемся добавить еще один такой же элемент

// Выводим элементы HashSet в консоль

System.out.println(h);

System.out.println("Список содержит Индия или нет:" +

h.contains("Индия"));

// Удаляем элементы из множества с помощью метода remove()

h.remove("Австралия");

System.out.println("Список после удаления Австралия:"+h);

// Проходимся по элементам HashSet с помощью итератора:

System.out.println("Повторение списка:");

for (String s : h) System.out.println(s);

}

}

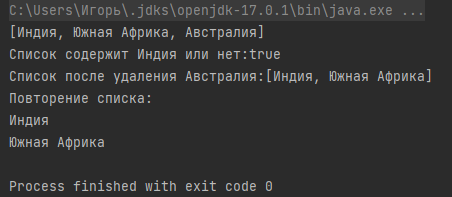


Рисунок 2 – Результат выполнения программы

***2. ArrayList***

Для создания простых списков применяется интерфейс List, который расширяет функциональность интерфейса Collection.

Таблица 5. Конструкторы в ArrayList

|  |  |
| --- | --- |
| Конструктор | Описание |
| ArrayList() | создает пустой список |
| ArrayList(Collection <? extends E> col) | создает список, в который добавляются все элементы коллекции col |
| ArrayList (int capacity): | создает список, который имеет начальную емкость capacity |

Некоторые наиболее часто используемые методы интерфейса ArrayList:

Таблица 6. Методы класса ArrayList

|  |  |
| --- | --- |
| Метод | Описание |
| **add(E e)** | Добавляет новый элемент в конец списка. Возвращает boolean-значение (true — успех, false — не добавлено) |
| **add(int index, E element)** | Добавляет элемент element в позицию index. При добавлении происходит сдвиг всех элементов справа от указанного индекса на 1 позицию вправо. Очень полезен, когда нужно вставить элемент в произвольное место списка, однако для частых операций вставки в начало и середину ArrayList может оказаться не очень удачным выбором |
| **addAll(Collection <? extends E> collection)** | Добавление всех элементов коллекции collection в список в порядке их расположения в collection |
| **addAll(int index, Collection <? extends E> collection)** | Добавление всех элементов collection в список начиная с индекса index. При этом все элементы сдвинутся вправо на количество элементов в списке collection |
| **clear()** | Удаление всех элементов из списка |
| **clone()** | Возвращает объект-копию массива. Следует обратить внимание, что метод clone() возвращает Object, так что после его вызова потребуется сделать приведение к необходимому классу. При клонировании создается новый независимый объект. |
| **contains(Object o)** | Проверка наличие объекта в списке, возвращает boolean-значение |
| **ensureCapacity(int minCapacity)** | Увеличивает размер внутреннего массива, чтобы в него поместилось количество элементов, переданных в minCapacity. Если массив достаточно вместителен, никакие преобразования не производятся. Этот метод полезен, когда возникает потребность вместить большое количество элементов в несколько итераций. Например, при создании списка емкость его внутреннего массива — 10. При загрузке данных по сети они обрабатываются асинхронно порциями и результаты помещаются в массив. Если ожидается доставка 10 000 элементов, может быть неэффективно просто добавлять эти данные каждый раз: достаточно будет в начале обработки вызвать метод ensureCapaciry(10000) и записывать туда данные по мере необходимости |
| **forEach(Consumer<? super E> action)** | В качестве аргумента передается реализация интерфейса Consumer, в котором нужно переопределить метод accept() |
| **get(int index)** | Возвращает элемент, который расположен в указанной позиции списка. |
| **indexOf(Object o)** | Метод возвращает индекс первого вхождения элемента в списке. Если элемента не существует в списке, метод вернет -1 |
| **isEmpty()** | Метод возвращает *true*, если список пустой, *false* в обратном случае. Если в списке содержатся только элементы null, метод вернет *false*. Иными словами, null элементы также учитываются этим методом |
| **iterator()** | Возвращает итератор для списка для последующего использования в цикле или при любой другой обработке. Итератор для ArrayList — fail-fast. Это значит, что если коллекция изменится во время итерации, будет выброшено исключение ConcurrentModificationException |
| **lastIndexOf(Object o)** | Функционал метода похож на indexOf(), отличие в том, что возвращается индекс последнего элемента в списке. Если элемент не найден, также возвращает -1 |
| **remove(int index)** | Удаление элемента в указанной позиции индекса. После удаления сдвигает все элементы влево для заполнения освободившегося пространства. Если index<0 или >= количество элементов списка, будет выброшено исключение IndexOutOfBoundsException. В результате метод возвращает элемент, который был удален |
| **remove(Object o)** | Метод удаляет из списка переданный элемент o. Если элемент присутствует в списке, он удаляется, а все элементы смещаются влево. Если элемент существует в списке и успешно удален, метод возвращает true, в обратном случае — false. |
| **removeAll(Collection<?> c)** | Если необходимо удалить несколько элементов, не стоит делать это в цикле по условию: гораздо удобнее и безопаснее воспользоваться методом removeAll(). Он принимает коллекцию элементов, которая будет удалена из списка. Коллекция должна содержать элементы того же типа, которые хранит целевой список. В обратном случае будет выброшен ClassCastException. Метод вернет *true*, если список был изменен в результате вызова метода. |
| **set(int index, E element)** | Замена элемента в указанной позиции index на переданный element. Индекс также должен быть больше нуля и меньше индекса последнего элемента, иначе будет выброшено исключение IndexOutOfBoundsException |
| **size()** | Лучший способ (практически единственный) для того, чтобы узнать размер массива |
| **sort(Comparator<? super E> c)** | Сортировка списка по заданному правилу. Правило сортировки представляет собой реализованный интерфейс Comparator с переопределенным методом compareTo(). Переопределение нужно, если коллекция содержит объекты собственного класса. При работе со стандартными классами (Integer, String и так далее) переопределение compareTo() требуется только для нестандартной сортировки |
| **toArray()** | Превращает список в фиксированный массив, метод возвращает массив объектов (Object[]). Если необходимо привести список в массив объектов определенного типа, в качестве параметра в метод можно передать массив, куда будут перемещены элементы списков |

По умолчанию в Java есть встроенная реализация этого интерфейса - класс ArrayList. Класс ArrayList представляет обобщенную коллекцию, которая наследует свою функциональность от класса AbstractList и применяет интерфейс List. Проще говоря, ArrayList представляет простой список, аналогичный массиву, за тем исключением, что количество элементов в нем не фиксировано.

Емкость в ArrayList представляет размер массива, который будет использоваться для хранения объектов. При добавлении элементов фактически происходит перераспределение памяти - создание нового массива и копирование в него элементов из старого массива. Изначальное задание емкости ArrayList позволяет снизить подобные перераспределения памяти, тем самым повышая производительность.

Используем класс ArrayList и некоторые его методы в программе:

Листинг 2

import java.util.ArrayList;

public class Program{

public static void main(String[] args) {

ArrayList<String> people = new ArrayList<String>();

// добавим в список ряд элементов

people.add("Tom");

people.add("Alice");

people.add("Kate");

people.add("Sam");

people.add(1, "Bob"); // добавляем элемент по индексу 1

System.out.println(people.get(1));// получаем 2-й объект

people.set(1, "Robert"); // установка нового значения для 2-го объекта

System.out.printf("ArrayList has %d elements \n", people.size());

for(String person : people){

System.out.println(person);

}

// проверяем наличие элемента

if(people.contains("Tom")){

System.out.println("ArrayList contains Tom");

}

// удалим несколько объектов

// удаление конкретного элемента

people.remove("Robert");

// удаление по индексу

people.remove(0);

Object[] peopleArray = people.toArray();

for(Object person : peopleArray){

System.out.println(person);

}

}

}

По факту выполнения программы терминал выдаст на экране:

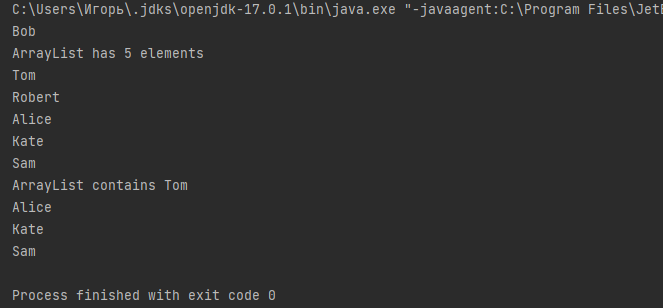


Рисунок 3 – Результат выполнения программы с классом ArrayList

Здесь объект ArrayList типизируется классом String, поэтому список будет хранить только строки. Поскольку класс ArrayList применяет интерфейс Collection<E>, то мы можем использовать методы данного интерфейса для управления объектами в списке.

Для добавления вызывается метод add. С его помощью мы можем добавлять объект в конец списка: people.add("Tom"). Также мы можем добавить объект на определенное место в списке, например, добавим объект на второе место (то есть по индексу 1, так как нумерация начинается с нуля): people.add(1, "Bob")

Метод size() позволяет узнать количество объектов в коллекции.

Проверку на наличие элемента в коллекции производится с помощью метода contains. А удаление с помощью метода remove. И так же, как и с добавлением, мы можем удалить либо конкретный элемент people.remove("Tom");, либо элемент по индексу people.remove(0); - удаление первого элемента.

Получить определенный элемент по индексу мы можем с помощью метода get():

String person = people.get(1);

а установить элемент по индексу с помощью метода set:

people.set(1, "Robert");

С помощью метода toArray() мы можем преобразовать список в массив объектов.

И поскольку класс ArrayList реализует интерфейс Iterable, то мы можем пробежаться по списку в цикле для for-each: for(String person : people).

Хотя мы можем свободно добавлять в объект ArrayList дополнительные объекты, в отличие от массива, однако в реальности ArrayList использует для хранения объектов опять же массив. По умолчанию данный массив предназначен для 10 объектов. Если в процессе программы добавляется гораздо больше, то создается новый массив, который может вместить в себя все количество. Подобные перераспределения памяти уменьшают производительность. Поэтому если мы точно знаем, что у нас список не будет содержать больше определенного количества элементов, например, 25, то мы можем сразу же явным образом установить это количество, либо в конструкторе:

ArrayList<String> people = new ArrayList<String>(25);

либо с помощью метода ensureCapacity:

people.ensureCapacity(25);

***3. SortedSet***

Интерфейс SortedSet языка Java, расширяющий интерфейс Set, описывает упорядоченное множество, отсортированное в возрастающем порядке или по порядку, заданному реализацией интерфейса Comparator.

Таблица 7. Методы интерфейса SortedSet

|  |  |
| --- | --- |
| Метод | Описание |
| Comparator<? super E> comparator() | возвращает компаратор сортированного множества. Если для множества применяется естественный порядок сортировки, возвращается null |
| E first() | возвращает первый элемент вызывающего сортированного множества |
| E last() | возвращает последний элемент вызывающего сортированного множества |
| SortedSet headSet(E toElement) | возвращает SortedSet, содержащий элементы из вызывающего множества, которые предшествуют end |
| SortedSet subSet(E fromElement, E toElement) | возвращает SortedSet, содержащий элементы из вызывающего множества, находящиеся между start и end-1 |
| SortedSet tailSet(E fromElement) | возвращает SortedSet, содержащий элементы из вызывающего множества, которые следуют за end |

Несколько методов вызывают исключение NoSuchElementException, если в вызывающем наборе не содержится элементов. ClassCastException вызывается, когда объект несовместим с элементами в наборе.

Исключение NullPointerException выдается, если совершается попытка использовать нулевой объект и null не допускается в наборе.

Рассмотрим пример использования методов subSet(), headSet(), tailSet(), first(), last():

Листинг 3

import java.util.SortedSet;

import java.util.TreeSet;

public class TreeSetDemo2 {

public static void main(String[] args) {

SortedSet<String> treeSet = new TreeSet<>();

treeSet.add("Москва");

treeSet.add("Екатеринбург");

treeSet.add("Нижний Тагил");

treeSet.add("Пермь");

treeSet.add("Москва");

System.out.println(treeSet);

SortedSet<String> subSet = treeSet.subSet("Екатеринбург", "Нижний Тагил");

System.out.println("SubSet: " + subSet);

System.out.println("HeadSet: " + treeSet.headSet("Нижний Тагил"));

System.out.println("TailSet: " + treeSet.tailSet("Нижний Тагил"));

System.out.println("Первый элемент: " + treeSet.first());

System.out.println("Последний элемент: " + treeSet.last());

}

}

По факту выполнения программы терминал выдаст на экране:

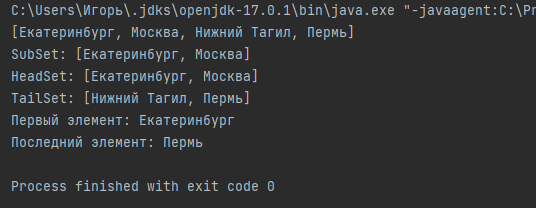


Рисунок 4 – Результат выполнения программы с классом SortedSet

* 1. Сравнение коллекций в соответствии с вариантом

Необходимо произвести вычисление времени работы нижеперечисленных операций, количество элементов коллекциях и сами коллекции указаны в индивидуальном варианте. После выполнения вычислений необходимо заполнить таблицы и привести скриншоты кода программы и данных вывода в консоль.

***Вариант №2 HashSet , ArrayList, SortedSet***

Прежде всего, привожу листинги написанных программ для каждой коллекции.

Листинг 4. Код для работы HashSet

import java.util.HashSet;

public class HashSetTest {

public static void main(String[] args) {

HashSet HashSetTest = new HashSet();

System.out.println("Время выполнения операции добавления в HashSet = " + getAddIndexRunningTime(HashSetTest));

System.out.println("Время выполнения операции удаления в HashSet = " + getRemoveIndexRunningTime(HashSetTest));

System.out.println("Время выполнения операции поиска в HashSet = " + getSearchIndexRunningTime(HashSetTest));

}

//добавление элементов

private static long getAddIndexRunningTime (HashSet time){

long start = System.currentTimeMillis();

for (int i = 0; i < 2000000; i++){

time.add(i + Math.random() \* 50);

}

long end = System.currentTimeMillis();

return end - start;

}

//удаление элементов

private static long getRemoveIndexRunningTime (HashSet HashSetTest){

for (int i = 0; i < 2000000; i++){

HashSetTest.add(i \* Math.random() \* 50);

}

long start = System.currentTimeMillis();

HashSetTest.remove(1);

long end = System.currentTimeMillis();

return end - start;

}

//поиск элемента

private static long getSearchIndexRunningTime (HashSet HashSetTest){

for (int i = 0; i < 2000000; i++){

HashSetTest.add(i + Math.random() \* 50);

}

HashSetTest.remove(1999999);

long start = System.currentTimeMillis();

System.out.println(HashSetTest.contains(1999999));

long end = System.currentTimeMillis();

return end - start;

}

}

Листинг 5. Код для работы ArrayList

import java.util.ArrayList;

public class ArrayListTest {

public static void main(String[] args) {

ArrayList ArrayListTest = new ArrayList();

System.out.println("Время выполнения операции добавления в ArrayList = " + getAddIndexRunningTime(ArrayListTest));

System.out.println("Время выполнения операции удаления в ArrayList = " + getRemoveIndexRunningTime(ArrayListTest));

System.out.println("Время выполнения операции поиска в ArrayList = " + getSearchIndexRunningTime(ArrayListTest));

}

//добавление элементов

private static long getAddIndexRunningTime (ArrayList ArrayListTest){

long start = System.currentTimeMillis();

for (int i = 0; i < 2000000; i++){

ArrayListTest.add(i + Math.random() \* 50);

}

long end = System.currentTimeMillis();

return end - start;

}

//удаление элементов

private static long getRemoveIndexRunningTime (ArrayList ArrayListTest){

for (int i = 0; i < 1\_000\_000; i++){

ArrayListTest.add(i \* Math.random() \* 50);

}

long start = System.currentTimeMillis();

ArrayListTest.remove(1);

long end = System.currentTimeMillis();

return end - start;

}

//поиск элемента

private static long getSearchIndexRunningTime (ArrayList ArrayListTest){

for (int i = 0; i < 2000000; i++){

ArrayListTest.add(i + Math.random() \* 50);

}

ArrayListTest.remove(1999999);

long start = System.currentTimeMillis();

System.out.println(ArrayListTest.get(1999999));

long end = System.currentTimeMillis();

return end - start;

}

}

Листинг 6. Код для работы SortedSet

import java.util.\*;

public class SortedSetTest {

public static void main(String[] args) {

SortedSet <Integer> sortedSet = new TreeSet<Integer>();

System.out.println("Время выполнения операции добавления в SortedSet = " + getAddIndexRunningTime(sortedSet));

long start = System.currentTimeMillis();

System.out.println("Время выполнения операции удаления в SortedSet = " + getRemoveRunningTime(sortedSet));

long end = System.currentTimeMillis();

System.out.println(end - start);

System.out.println("Время выполнения операции поиска в SortedSet = " + getAddIndexRunningTime(sortedSet));

}

//добавление

private static long getAddIndexRunningTime (SortedSet SortedSetTest){

long start = System.currentTimeMillis();

for (int i = 0; i < 2000000; i++){

SortedSetTest.add(i + "asd" + i);

}

long end = System.currentTimeMillis();

return end - start;

}

//удаление

private static long getRemoveRunningTime (SortedSet SortedSetTest){

int i = 0;

for (i = 0; i < 2000000; i++){

SortedSetTest.add(i + "asd" + i);

}

long start = System.currentTimeMillis();

SortedSetTest.remove(i + "asd" + i);

long end = System.currentTimeMillis();

return end - start;

}

//поиск

private static long getSearchIndexRunningTIme (SortedSet SortedSetTest){

for (int i = 0; i < 2000000; i++){

SortedSetTest.add(i + "asd" + i);

}

long start = System.currentTimeMillis();

SortedSetTest.subSet(1999999, 1999998);

long end = System.currentTimeMillis();

return end - start;

}

}

Работу данных программ могут продемонстрировать изображения с терминала ПО IntellijIDEA.

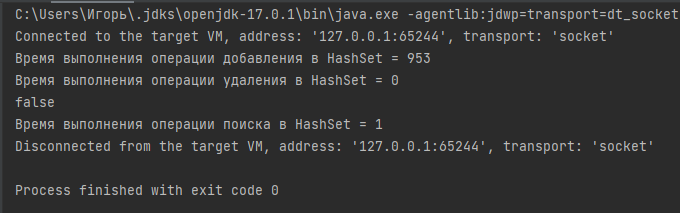


Рисунок 5 – Результат работы HashSet

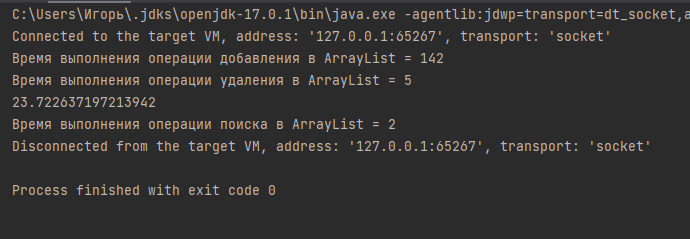
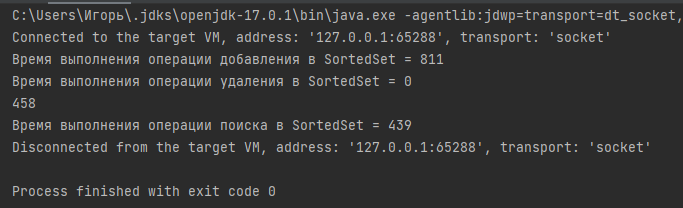


Рисунок 6 – Результат работы ArrayList

Рисунок 7 – Результат работы SortedSet

Добавление, удаление и поиск элемента в случаях

1. Начало коллекции
2. Середина коллекции
3. Конец коллекции

Таблица 8. Сравнение времени операции добавления элемента в коллекцию

|  |  |  |  |
| --- | --- | --- | --- |
| Коллекция | в начало коллекции | в середину коллекции | в конец коллекции |
| HashSet | 28 | 50 | 54 |
| ArrayList | 0 | 113 | 124 |
| SortedSet | 18 | 20 | 18 |

Таблица 9. Сравнение времени операции удаления элемента из коллекции

|  |  |  |  |
| --- | --- | --- | --- |
| Коллекция | в начало коллекции | в середину коллекции | в конец коллекции |
| HashSet | 0 | 0 | 0 |
| ArrayList | 5 | 5 | 2 |
| SortedSet | 0 | 0 | 0 |

Таблица 10. Сравнение времени операции поиска элемента в коллекции

|  |  |  |  |
| --- | --- | --- | --- |
| Коллекция | в начало коллекции | в середину коллекции | в конец коллекции |
| HashSet | 0 | 0 | 1 |
| ArrayList | 2 | 0 | 0 |
| SortedSet | 19 | 25 | 27 |

Ниже приложены скриншоты с результатами выполнения программ.

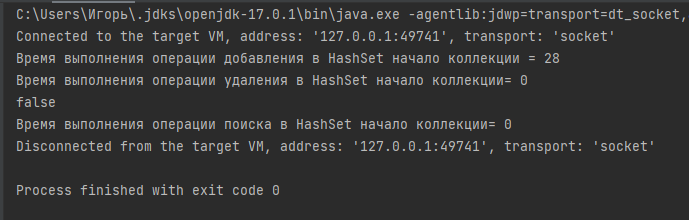


Рисунок 8 – HashSet начало коллекции

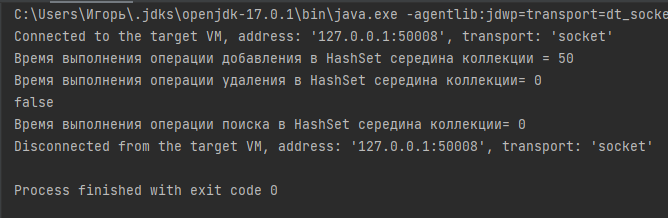


Рисунок 8 – HashSet середина коллекции

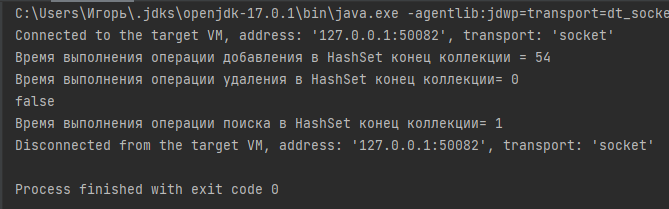


Рисунок 8 – HashSet конец коллекции

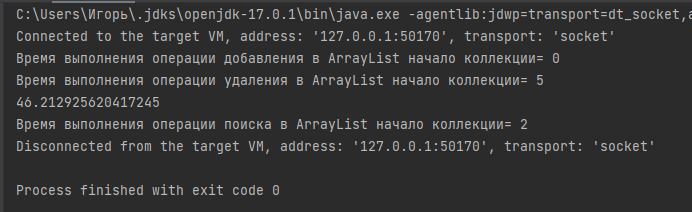


Рисунок 9 – ArrayList начало коллекции

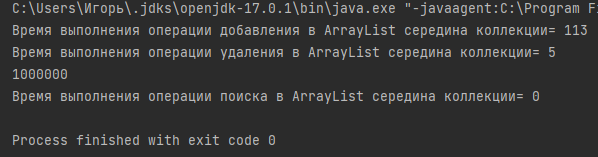


Рисунок 10 – ArrayList середина коллекции

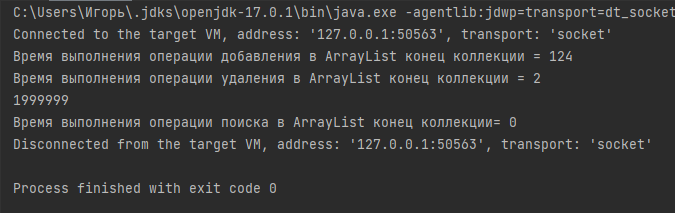


Рисунок 11 – ArrayList конец коллекции

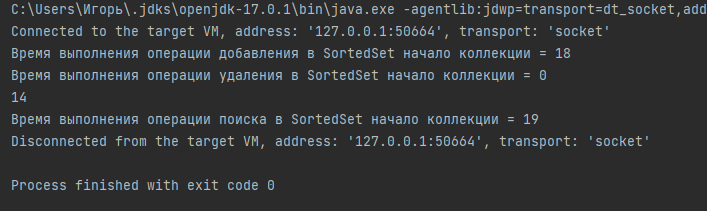


Рисунок 12 – SortedSet начало коллекции

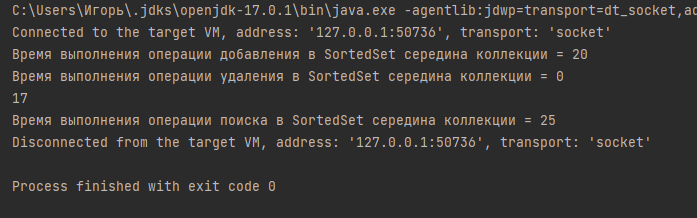


Рисунок 13 – SortedSet середина коллекции

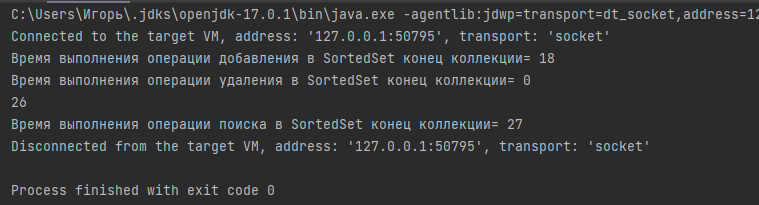


Рисунок 14 – SortedSet конец коллекции

Ответ на вопрос в соответствии с вариантом

В чем разница между Iterator и Enumeration?

Основное различие между Iterator и Enumeration - это удаление элемента при обходе коллекции. Итератор может удалить элемент во время обхода коллекции, поскольку у него есть метод remove (). В перечислении нет метода remove ()