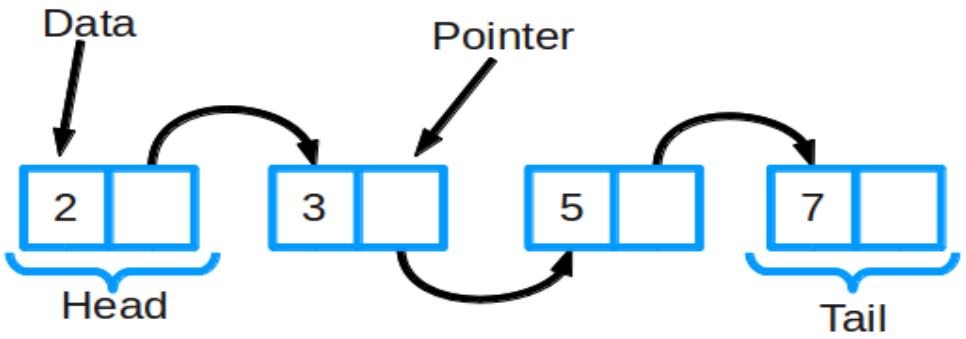
Java коллекции

Связанный список

Связный список — одна из базовых структур данных. Ее часто сравнивают с массивом, так как многие другие структуры можно реализовать с помощью либо массива, либо связного списка. У этих двух типов есть преимущества и недостатки.



Связный список состоит из группы узлов, которые вместе образуют последовательность. Каждый узел содержит две вещи: фактические данные, которые в нем хранятся (это могут быть данные любого типа) и указатель (или ссылку) на следующий узел в последовательности. Также существуют двусвязные списки: в них у каждого узла есть указатель и на следующий, и на предыдущий элемент в списке.

Связанный список

Связанный список – массив где каждый элемент является отдельным объектом и состоит из двух элементов – данных и ссылки на следующий узел.

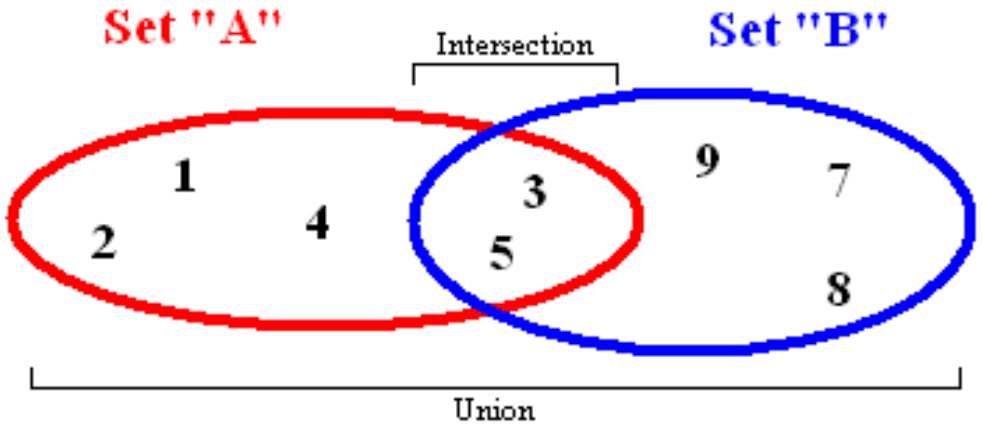
Однонаправленный, каждый узел хранит адрес или ссылку на следующий узел в списке и последний узел имеет следующий адрес или ссылку как NULL.
1->2->3->4->NULL

Двунаправленный, две ссылки, связанные с каждым узлом, одним из опорных пунктов на следующий узел и один к предыдущему узлу.
NULL<-1<->2<->3->NULL

Множества

Множество хранит значения данных без определенного порядка, не повторяя их. Оно позволяет не только добавлять и удалять элементы: есть ещё несколько важных функций, которые можно применять к двум множествам сразу.

- Объединение комбинирует все элементы из двух разных множеств, превращая их в одно (без дубликатов).
- Пересечение анализирует два множества и создает еще одно из тех элементов, которые присутствуют в обоих изначальных множествах.
- Разность выводит список элементов, которые есть в одном множестве, но отсутствуют в другом.
- Подмножество выдает булево значение, которое показывает, включает ли одно множество все элементы другого множества.



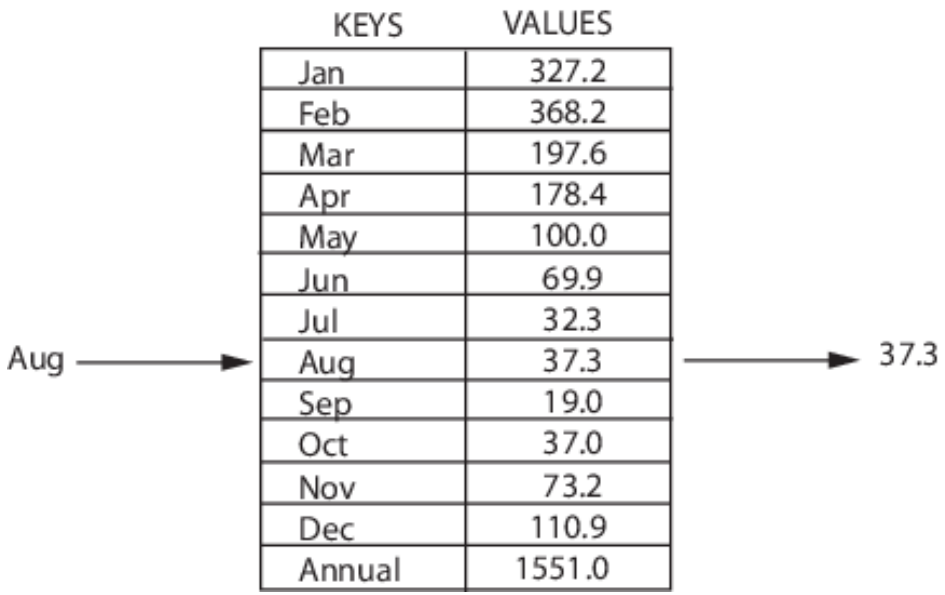
Map (ассоциативный массив)

Ассоциативный массив — абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

В паре *(k, v)* значение *v* называется значением, ассоциированным с ключом *k*. Где *k* — это key, а *v* — value. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов — например, строки.

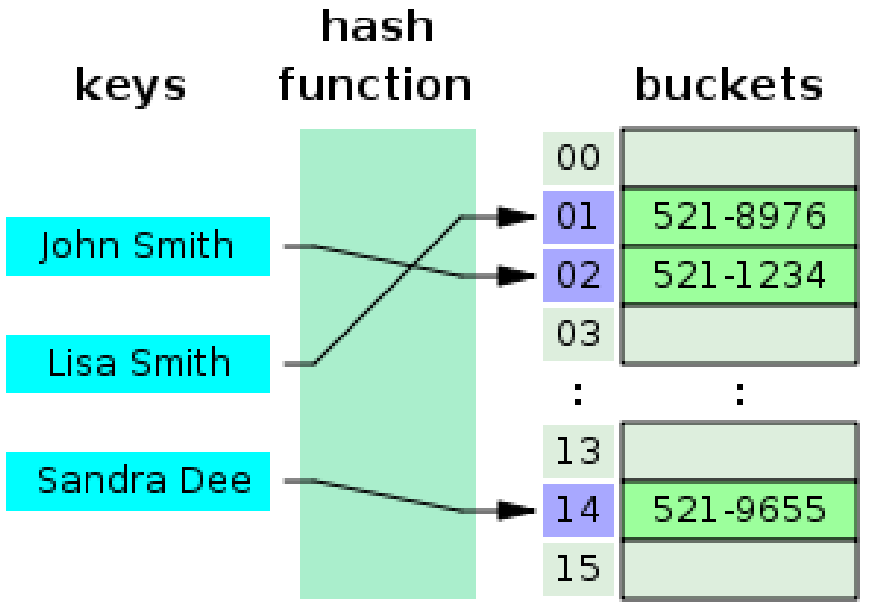


Хэш-таблицы

Хэш-таблица — это похожая на Map структура, которая содержит пары ключ/значение. Она использует хэш-функцию для вычисления индекса в массиве из блоков данных, чтобы найти желаемое значение.

Обычно хэш-функция принимает строку символов в качестве вводных данных и выводит числовое значение. Для одного и того же ввода хэш-функция должна возвращать одинаковое число. Если два разных ввода хэшируются с одним и тем же итогом, возникает коллизия. Цель в том, чтобы таких случаев было как можно меньше.

Таким образом, когда вы вводите пару ключ/значение в хэш-таблицу, ключ проходит через хэш-функцию и превращается в число. В дальнейшем это число используется как фактический ключ, который соответствует определенному значению. Когда вы снова введёте тот же ключ, хэш-функция обработает его и вернет такой же числовой результат. Затем этот результат будет использован для поиска связанного значения. Такой подход заметно сокращает среднее время поиска.



Дерево

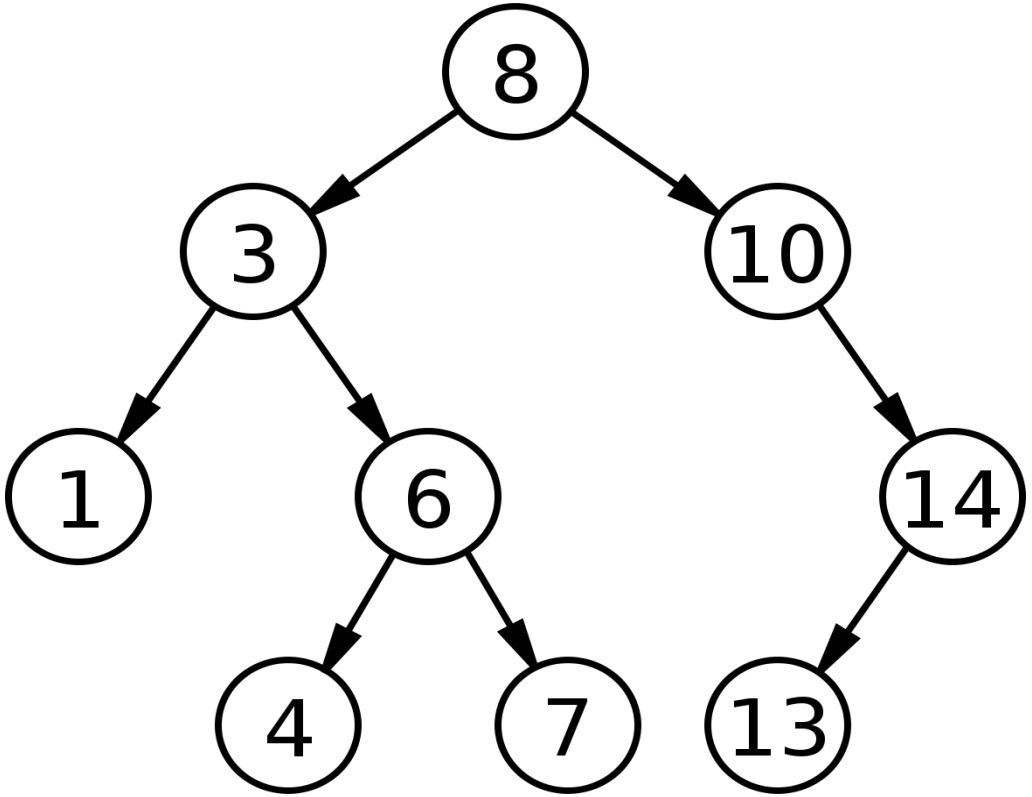
Дерево — это структура данных, состоящая из узлов. Ей присущи следующие свойства:

- Каждое дерево имеет корневой узел (вверху).
- Корневой узел имеет ноль или более дочерних узлов.
- Каждый дочерний узел имеет ноль или более дочерних узлов, и так далее.

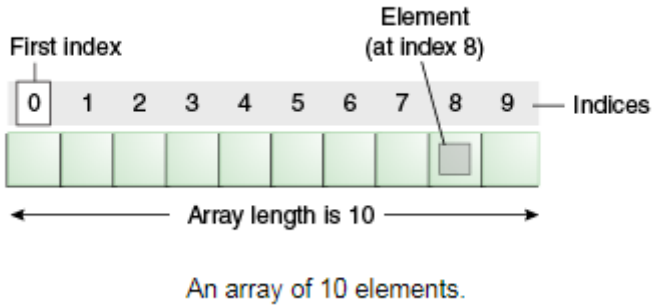
У двоичного дерева поиска есть два дополнительных свойства:

- Каждый узел имеет до двух дочерних узлов (потомков).
- Каждый узел меньше своих потомков справа, а его потомки слева меньше его самого.

Двоичные деревья поиска позволяют быстро находить, добавлять и удалять элементы. Они устроены так, что время каждой операции пропорционально логарифму общего числа элементов в дереве.



Массив



An array of 10 elements.

Массивы занимают изначально определенный отрезок памяти который задается его размером. Поскольку все размеры ячеек одинаковые, массив обеспечивает доступ к ячейке за КОНСТАНТНОЕ время!
Для этого берется

Ячейка = адрес в памяти + размер ячеек * количество ячеек

Объявление массивов:

```
byte[ ] anArrayOfBytes;
short[ ] anArrayOfShorts;
long[ ] anArrayOfLongs;
float[ ] anArrayOfFloats;
double[ ] anArrayOfDoubles;
boolean[ ] anArrayOfBooleans;
char[ ] anArrayOfChars;
String[ ] anArrayOfStrings;
```

Инициализация массивов:

```
int[ ] anArray = { 100, 200, 300, 400, 500,
600, 700, 800, 900, 1000 };

int[ ] anArray = new int [10]
```

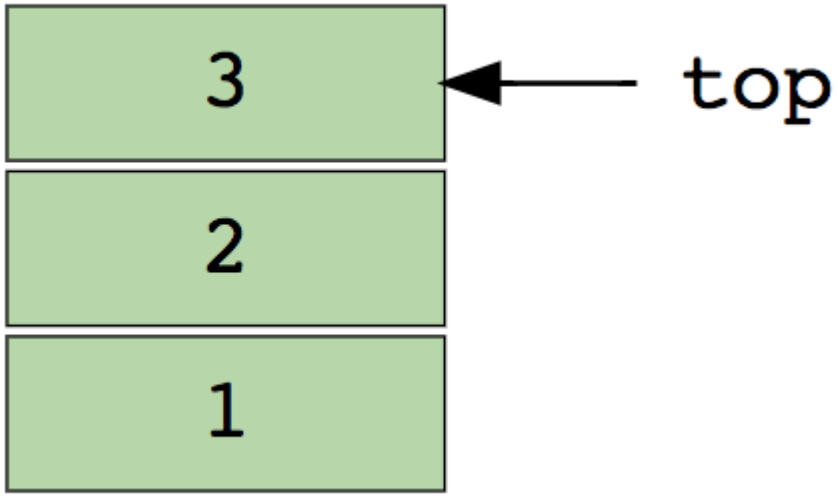
Стек

Стек – это структура данных, организованная по принципу "последним пришёл, первым вышел" (LIFO). Это значит, что последний элемент, добавленный в стек, будет первым извлечён из него.

Простыми словами: Стек можно представить как стопку тарелок. Чтобы взять тарелку со дна, нужно сначала снять все тарелки, которые лежат сверху.

Пример использования:

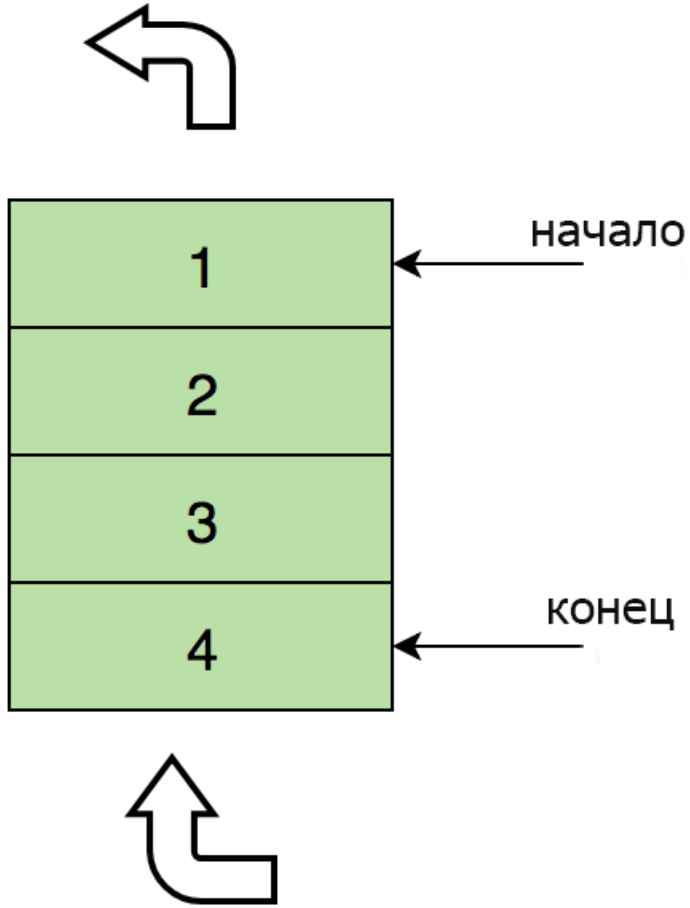
Стек вызовов функций: Когда одна функция вызывает другую, новая функция помещается на вершину стека вызовов. После выполнения функции, она удаляется из стека, и управление возвращается к предыдущей функции.



Очередь

Очереди работают по принципу первый вошел –первый вышел (FIFO, англ. first in, first out).

Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди,
выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.



Java Collection

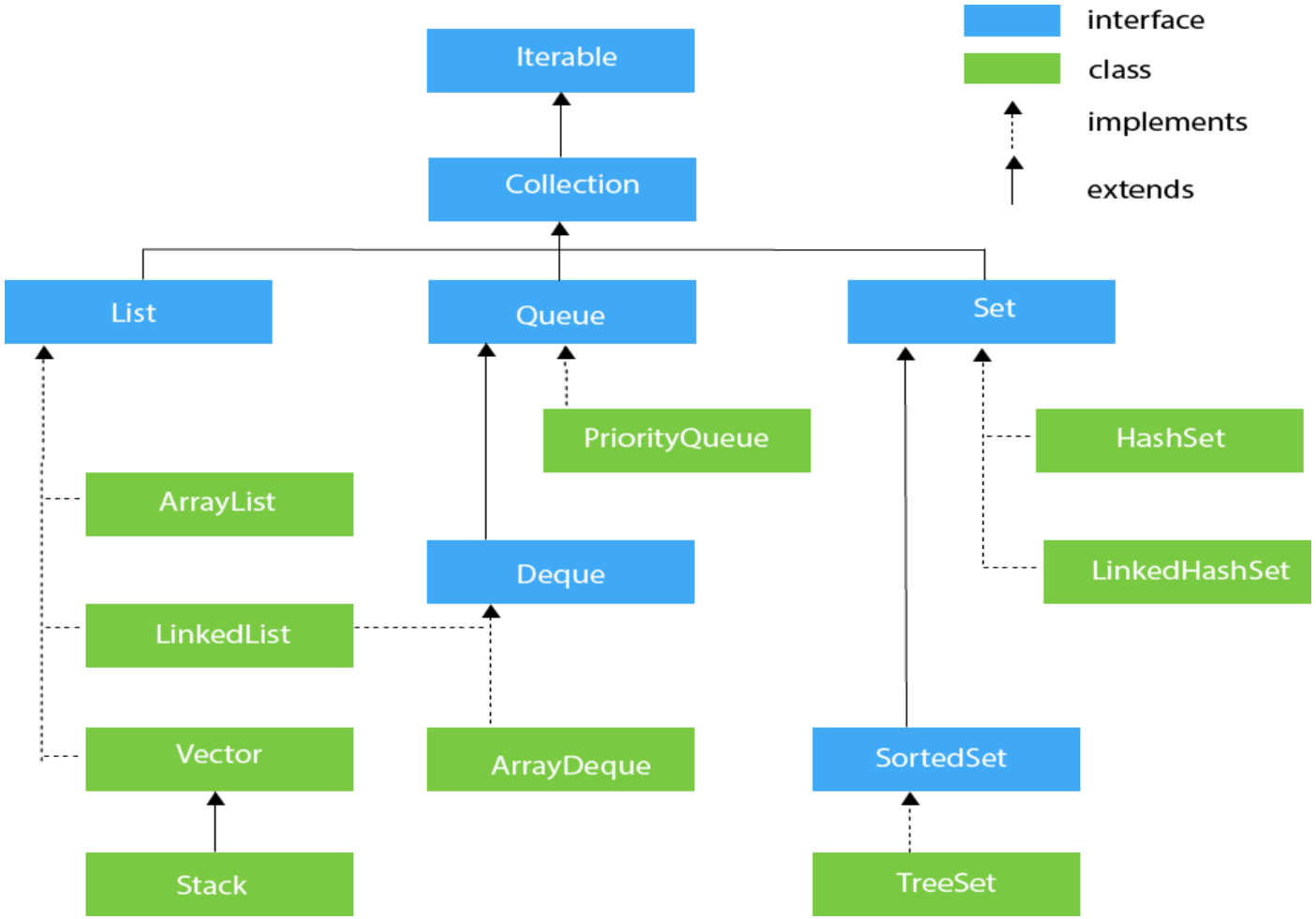
Java Collection - это фреймворк в языке программирования Java, предназначенный для работы с группами объектов. Он предоставляет классы и интерфейсы, которые позволяют создавать, хранить, обрабатывать и управлять коллекциями объектов.

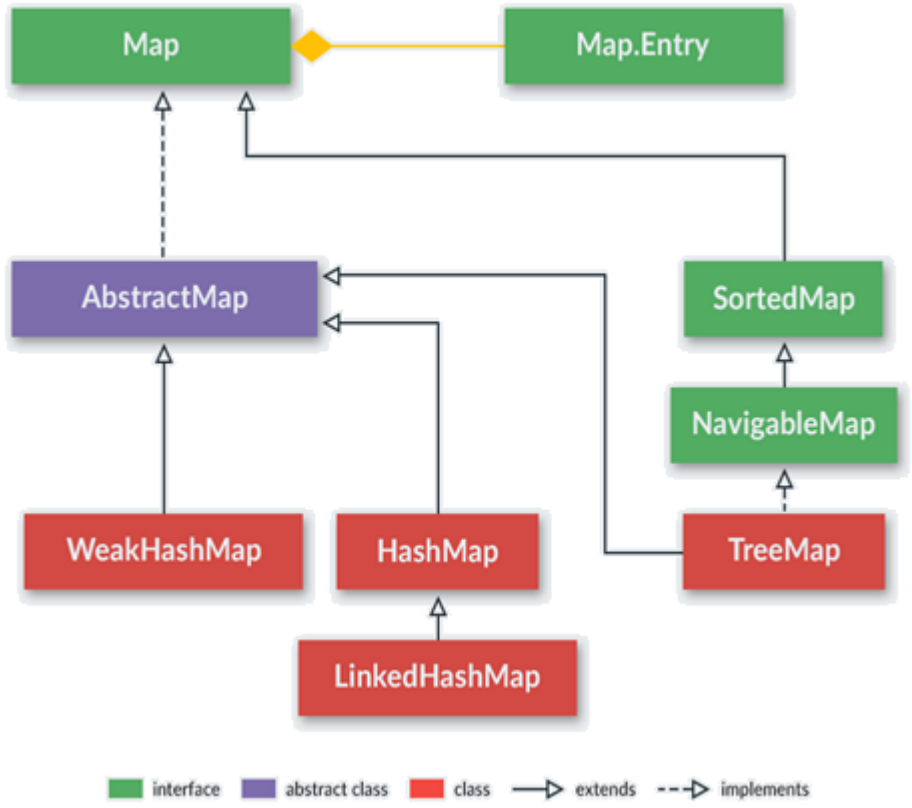
Коллекция в Java представляет собой контейнер, который содержит набор элементов одного типа или различных типов. Она может быть динамически расширяемой, что означает, что размер коллекции может изменяться в процессе выполнения программы.

Коллекции

- Примитивные типы нельзя хранить в коллекции.
- Хранимые в коллекции объекты называются элементами.
- Коллекции могут хранить только ссылки на объекты
- Классы коллекций хранятся в пакете java. util.
- Библиотека классов и интерфейсов для поддержки коллекций называется Java collections framework (JCF). Он появился начиная с версии Java 1.2. В версии 1.5 в JCF добавили поддержку обобщений.
- Помимо соответствующих классов и интерфейсов, в JCF реализовано множество общеупотребительных алгоритмов для поиска, сортировки и т.п.

Иерархия


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet


Map
Map.Entry
AbstractMap
SortedMap
NavigableMap
WeakHashMap
HashMap
TreeMap
LinkedHashMap
interface
abstract class
class
extends
implements

Iterable

Интерфейс **Iterable** является корневым интерфейсом для всех классов коллекций. Интерфейс Collection вместе со всеми его подклассы также реализуют интерфейс Iterable.

Имеет один метод **Iterator<T> iterator()**

Итератор

В Java Iterator - это интерфейс, предоставляющий способ последовательного перебора элементов в коллекции. Он является частью Java Collection Framework и определяет методы для доступа и обхода элементов коллекции.

Интерфейс Iterator определен в пакете java.util и содержит следующие методы:

1. boolean hasNext(): Возвращает значение true, если в коллекции есть следующий элемент, который можно извлечь методом next(). Если все элементы коллекции были пройдены, возвращает false.
2. E next(): Возвращает следующий элемент из коллекции. Итератор перемещается к следующему элементу в коллекции при каждом вызове этого метода.
3. void remove(): Удаляет текущий элемент из коллекции. Этот метод может быть вызван только после вызова метода next(). Если элемент был удален успешно, то состояние коллекции изменяется соответствующим образом.

Итераторы используются для безопасного обхода элементов коллекции независимо от ее конкретной реализации. Они предоставляют универсальный способ доступа к элементам и позволяют выполнять операции, такие как чтение, удаление или модификация элементов коллекции.

Итератор пример

```
List<String> names = new ArrayList<>();
names.add("John");
names.add("Jane");
names.add("Alice");

Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}
```

ListIterator

В Java ListIterator - это интерфейс, расширяющий интерфейс Iterator, предоставляющий дополнительные операции для манипуляции с элементами в списке. Он также является частью Java Collection Framework и определен в пакете java.util.

ListIterator предоставляет все методы интерфейса Iterator и добавляет следующие дополнительные методы:

1. boolean hasPrevious(): Возвращает true, если существует предыдущий элемент в списке, который можно извлечь методом previous(). Если предыдущий элемент отсутствует, возвращает false.
2. E previous(): Возвращает предыдущий элемент в списке. Итератор перемещается к предыдущему элементу при каждом вызове этого метода.
3. int nextIndex(): Возвращает индекс следующего элемента в списке.
4. int previousIndex(): Возвращает индекс предыдущего элемента в списке.
5. void set(E element): Заменяет последний элемент, который был возвращен методом next() или previous(), указанным элементом.
6. void add(E element): Вставляет указанный элемент в список между элементом, который будет возвращен следующим вызовом next(), и элементом, который будет возвращен следующим вызовом previous(). Если ни next(), ни previous() еще не были вызваны, элемент будет добавлен в начало списка.

ListIterator предоставляет возможность проходить по списку в обоих направлениях (вперед и назад), а также позволяет добавлять и удалять элементы во время обхода списка.

ListIterator пример

```
List<String> names = new ArrayList<>();
names.add("John");
names.add("Jane");
names.add("Alice");

ListIterator<String> iterator = names.listIterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}

// В обратном направлении
while (iterator.hasPrevious()) {
    String name = iterator.previous();
    System.out.println(name);
}
```

Методы Collection

- **boolean add(E element):** Добавляет элемент в коллекцию и возвращает true, если операция выполнена успешно. Если добавление элемента невозможно (например, если коллекция имеет ограничение на размер), будет выброшено исключение.
- **boolean remove(Object element):** Удаляет указанный элемент из коллекции и возвращает true, если элемент найден и успешно удален. Если элемент не найден, возвращается false.
- **boolean contains(Object element):** Проверяет, содержится ли указанный элемент в коллекции. Возвращает true, если элемент найден, и false в противном случае.
- **int size():** Возвращает текущее количество элементов в коллекции.
- **boolean isEmpty():** Проверяет, является ли коллекция пустой. Возвращает true, если коллекция не содержит элементов, и false в противном случае.
- **void clear():** Удаляет все элементы из коллекции, оставляя ее пустой.

Методы Collection

- **boolean containsAll(Collection<?> collection):** Проверяет, содержатся ли все элементы из указанной коллекции в текущей коллекции. Возвращает true, если все элементы найдены, и false в противном случае.
- **boolean addAll(Collection<? extends E> collection):** Добавляет все элементы из указанной коллекции в текущую коллекцию. Возвращает true, если коллекция изменилась после выполнения операции.
- **boolean removeAll(Collection<?> collection):** Удаляет из текущей коллекции все элементы, которые содержатся в указанной коллекции. Возвращает true, если коллекция изменилась после выполнения операции.
- **boolean retainAll(Collection<?> collection):** Удаляет из текущей коллекции все элементы, которые не содержатся в указанной коллекции. Возвращает true, если коллекция изменилась после выполнения операции.
- **Object[] toArray():** Возвращает массив, содержащий все элементы из коллекции.
- **Iterator<E> iterator():** Возвращает итератор для обхода элементов в коллекции.

List

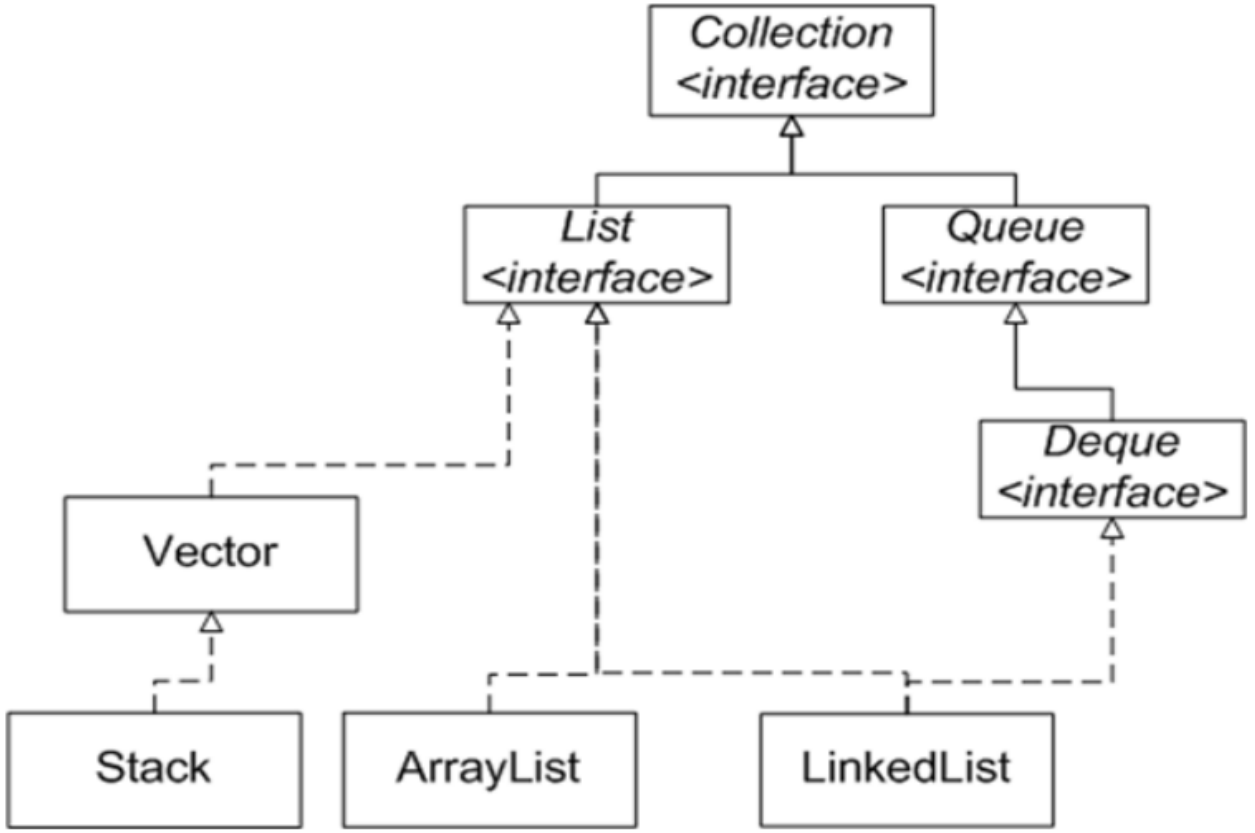
Интерфейс List в Java расширяет интерфейс Collection и представляет собой упорядоченную коллекцию объектов с возможностью дублирования элементов. Он определяет дополнительные методы, специфичные для работы со списками. Вот некоторые из методов, определенных в интерфейсе List:

Методы интерфейса List

1. **void add(int index, E element):** Вставляет указанный элемент в список по указанному индексу. Существующие элементы сдвигаются вправо.
2. **boolean remove(Object element):** Удаляет первое вхождение указанного элемента из списка, если он присутствует.
3. **E remove(int index):** Удаляет элемент из списка по указанному индексу и возвращает удаленный элемент.
4. **E get(int index):** Возвращает элемент из списка по указанному индексу.
5. **E set(int index, E element):** Заменяет элемент в списке по указанному индексу новым элементом и возвращает старый элемент.
6. **int indexOf(Object element):** Возвращает индекс первого вхождения указанного элемента в списке. Если элемент не найден, возвращает -1.
7. **int lastIndexOf(Object element):** Возвращает индекс последнего вхождения указанного элемента в списке. Если элемент не найден, возвращает -1.
8. **List<E> subList(int fromIndex, int toIndex):** Возвращает представление списка, ограниченное указанными индексами fromIndex (включительно) и toIndex (исключительно).
9. **boolean addAll(int index, Collection<? extends E> collection):** Вставляет все элементы из указанной коллекции в список, начиная с указанного индекса.
10. **ListIterator<E> listIterator():** Возвращает ListIterator для обхода элементов в списке.
11. **ListIterator<E> listIterator(int index):** Возвращает ListIterator для обхода элементов в списке, начиная с указанного индекса.

Классы которые реализуют List

- ArrayList
- LinkedList
- Vector(deprecated)



ArrayList

Одной из реализаций интерфейса List является класс **ArrayList**. Он поддерживает динамические массивы, которые могут расти по мере необходимости.

Объект класса ArrayList, содержит свойства elementData и size. Хранилище значений elementData есть не что иное, как массив определенного типа (указанного в generic).

Если пользователь добавит в ArrayList больше элементов чем его размерность, ничего плохого не произойдет (в отличие от массивов, где будет выброшено ArrayIndexOutOfBoundsException исключение). В этом случае просто произойдет пересоздание внутреннего массива elementData, и это произойдет неявно для пользователя.

Конструкторы класса ArrayList

1. **ArrayList()**: Создает пустой список ArrayList с начальной емкостью 10.
2. **ArrayList(Collection<? extends E> collection)**: Создает список ArrayList, содержащий элементы из указанной коллекции, в том же порядке, в котором они возвращаются итератором коллекции.
3. **ArrayList(int initialCapacity)**: Создает пустой список ArrayList с указанной начальной емкостью. Начальная емкость представляет собой количество элементов, которое список может содержать без изменения его размера.
4. **ArrayList(List<? extends E> list)**: Создает список ArrayList, содержащий элементы из указанного списка, в том же порядке, в котором они расположены в исходном списке.

Достоинства и недостатки ArrayList

Достоинства

- Быстрый доступ по индексу. Скорость такой операции - O(1).
- Быстрая вставка и удаление элементов с конца. Скорость операций опять же - O(1).

Недостатки

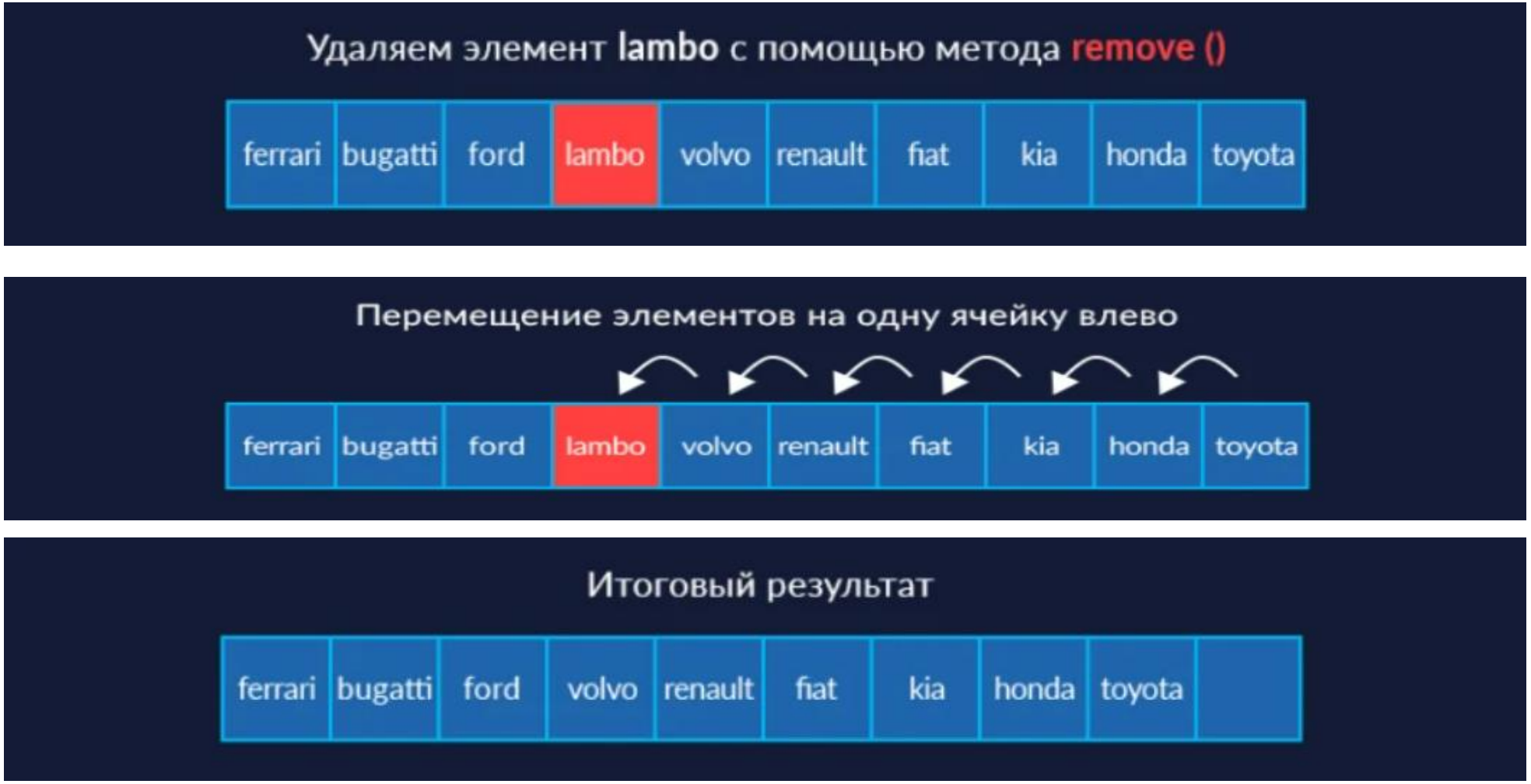
- Медленная вставка и удаление элементов из середины. Такие операции имеют сложность близкую к O(n). Поэтому, если вы понимаете, что вам придется выполнять достаточно много операций такого типа, может быть лучше выбрать другой класс.

Вставка когда массив полон

- Создается новый массив размером, в 1.5 раза больше исходного.
- Все элементы из старого массива копируются в новый массив
- Новый массив сохраняется во внутренней переменной объекта ArrayList, а старый массив объявляется мусором.

Вставка когда массив полон

Удаление элемента



trimToSize()


элемент элемент элемент элемент элемент элемент элемент элемент элемент элемент элемент
trimToSize()
результат
элемент элемент элемент элемент элемент элемент элемент элемент элемент элемент элемент

LinkedList

LinkedList - является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы. Поэтому итератор поддерживает обход в обе стороны

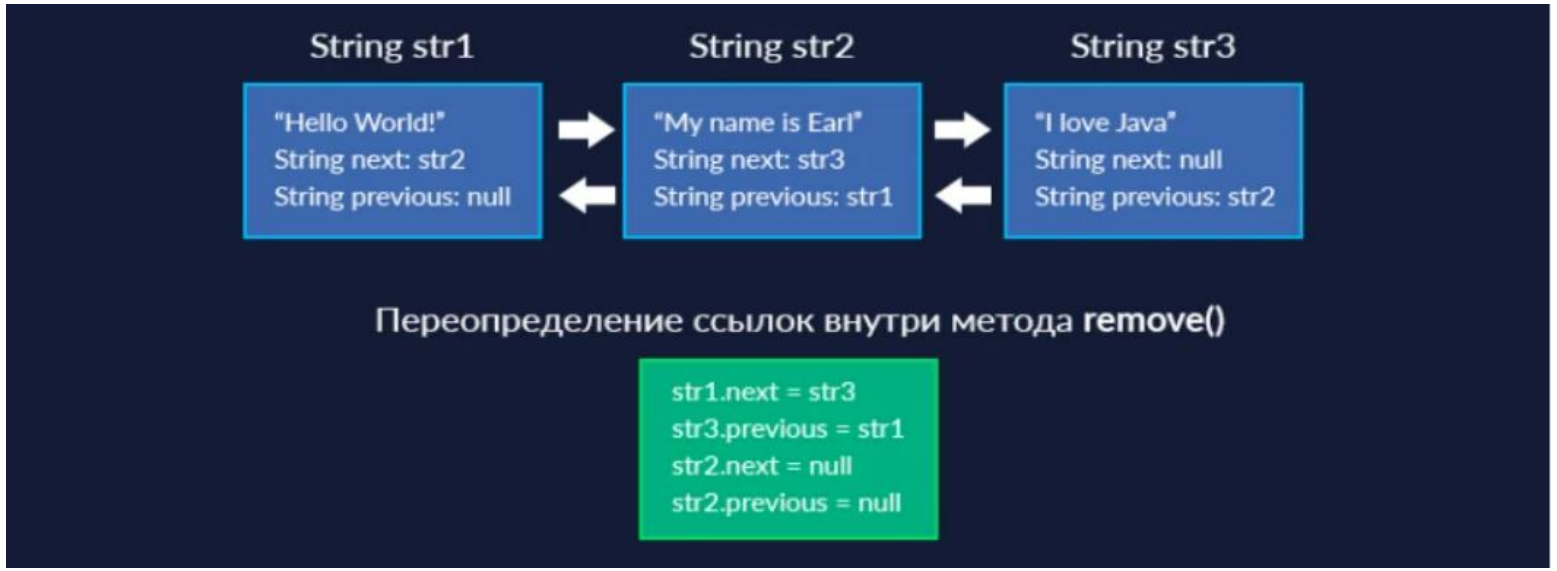
Реализует методы получения, удаления и вставки в начало, середину и конецсписка.

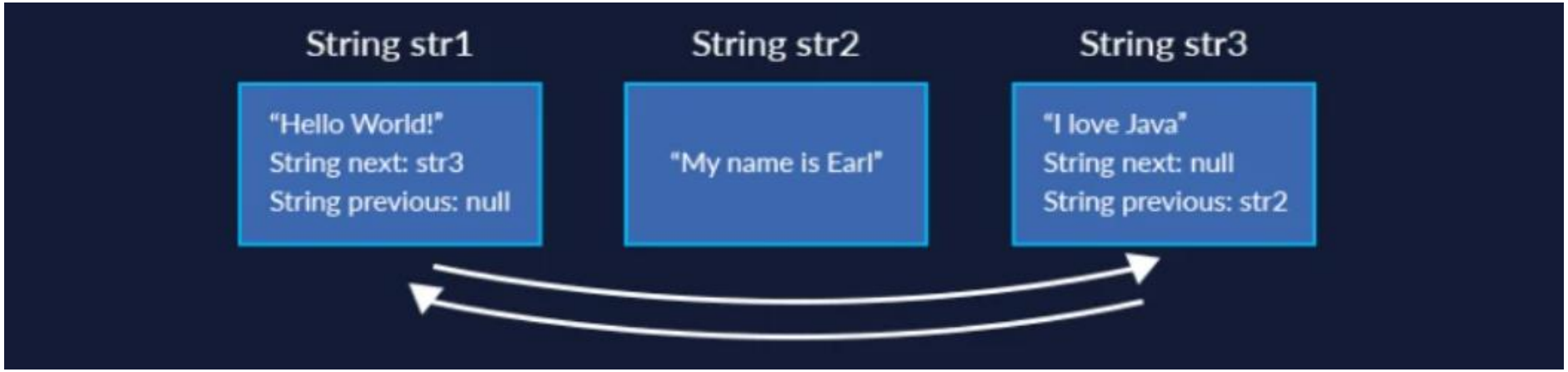
Структура LinkedList

Добавление в LinkedList

Вставка в середину LinkedList

Удаление из середины LinkedList





Достоинства и недостатки LinkedList

Добавление элемента в конец списка с помощью методом **add(value)**, **addLast(value)** и добавление в начало списка с помощью **addFirst(value)** выполняется за время O(1). 0(1).

Вставки и удаления тоже выполняются очень быстро в **LinkedList**. Однако, доступ к элементу влечет за собой обход узлов один за одним, так что это достаточно медленный процесс.

LinkedList обычно используется, если необхолимо часто добавить или удалить элементы в списке, особенно в начале списка. Либо если нам нужна вставка элемента в конец за гарантированное время

Сравнение LinkedList и ArrayList

Описание	Операция	ArrayList	LinkedList
Взятие элемента	get	Быстро	Медленно
Присваивание элемента	set	Быстро	Медленно
Добавление элемента	add	Быстро	Быстро
Вставка элемента	add(I, value)	Медленно	Быстро
Удаление элемента	remove	Медленно	Быстро

Интерфейс Queue

В Java интерфейс Queue представляет собой коллекцию, реализующую структуру данных "очередь" (queue) - это упорядоченная коллекция элементов, в которой элементы добавляются в конец очереди и удаляются из начала очереди. Он расширяет интерфейс Collection и определяет методы для работы с очередью. Некоторые из основных методов, определенных в интерфейсе Queue, включают:

1. **boolean add(E** element**)**: Добавляет элемент в конец очереди. Если очередь заполнена и не может принять новый элемент (например, если она имеет ограничение на размер), будет выброшено исключение.
2. **boolean offer(E element)**: Добавляет элемент в конец очереди. Если очередь заполнена и не может принять новый элемент, возвращает false.
3. **E remove()**: Удаляет и возвращает элемент из начала очереди. Если очередь пуста, будет выброшено исключение.
4. **E poll()**: Удаляет и возвращает элемент из начала очереди. Если очередь пуста, возвращает null.
5. **E element()**: Возвращает элемент из начала очереди, не удаляя его. Если очередь пуста, будет выброшено исключение.
6. **E peek()**: Возвращает элемент из начала очереди, не удаляя его. Если очередь пуста, возвращает null.

Интерфейс Queue также наследует методы из интерфейса Collection, такие как **size()**, **isEmpty()**, **contains()**, **remove()**, **addAll()** и другие, которые можно использовать для работы с очередью.

Пример использования Queue

```
import java.util.Queue;
import java.util.LinkedList;

public class QueueExample {
    public static void main(String[] args) {
        // Создание очереди
        Queue<String> taskQueue = new LinkedList<>();

        // Добавление элементов в очередь
        taskQueue.offer("Task 1");
        taskQueue.offer("Task 2");
        taskQueue.offer("Task 3");
        taskQueue.offer("Task 4");

        // Обработка элементов в очереди
        while (!taskQueue.isEmpty()) {
            String task = taskQueue.poll(); // Получение и удаление элемента
из начала очереди
            System.out.println("Processing task: " + task);
        }
    }
}
```

Интерфейс Dequeue

В Java интерфейс Deque (Double Ended Queue) представляет собой коллекцию, реализующую структуру данных "двусторонняя очередь". Он расширяет интерфейс Queue и добавляет методы для работы с элементами как с начала, так и с конца очереди. Интерфейс Deque поддерживает добавление, удаление и доступ к элементам с обоих концов очереди. Некоторые из основных методов, определенных в интерфейсе Deque, включают:

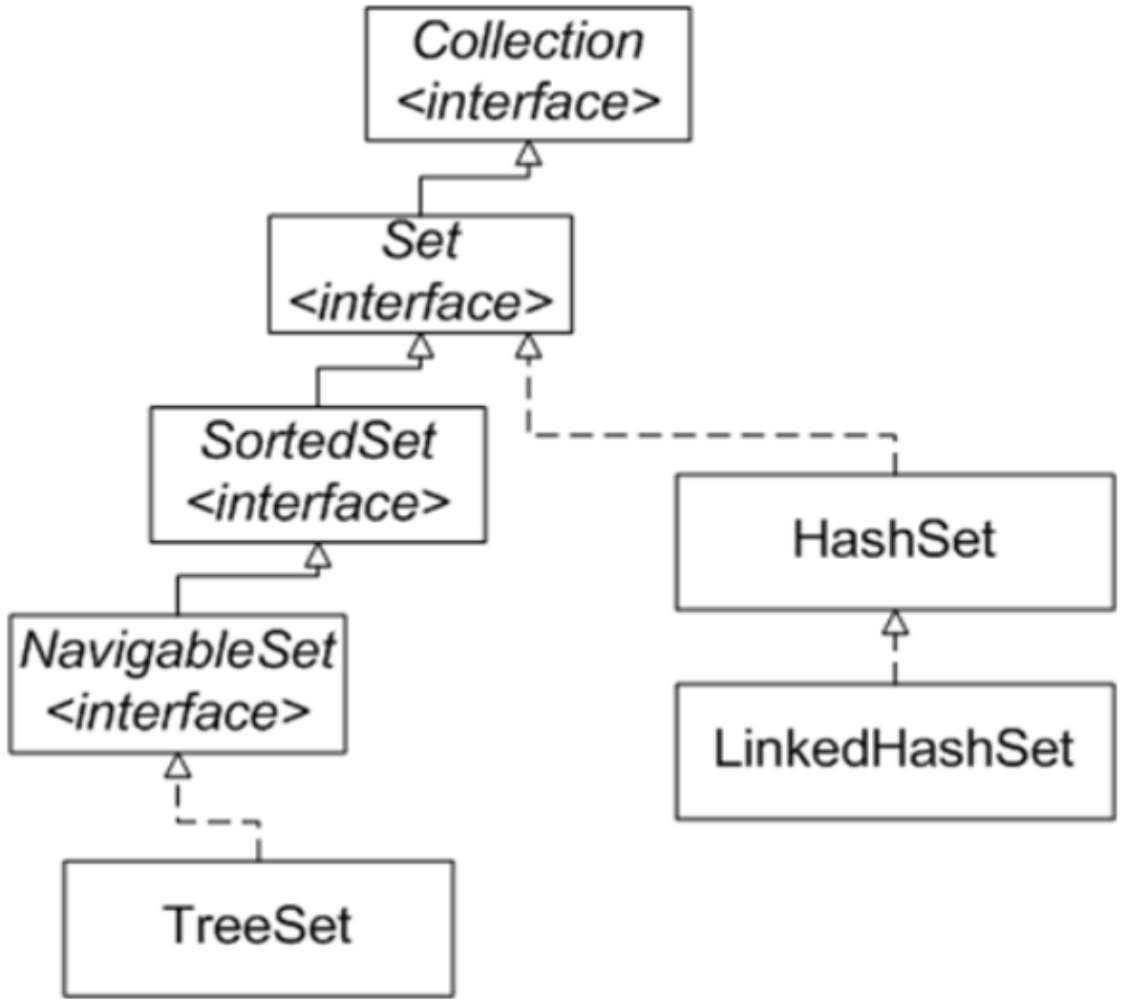
1. **void addFirst(E element):** Добавляет элемент в начало очереди.
2. **void addLast(E element):** Добавляет элемент в конец очереди.
3. **boolean offerFirst(E element):** Добавляет элемент в начало очереди. Если очередь заполнена и не может принять новый элемент, возвращает false.
4. **boolean offerLast(E element):** Добавляет элемент в конец очереди. Если очередь заполнена и не может принять новый элемент, возвращает false.
5. **E removeFirst():** Удаляет и возвращает элемент из начала очереди. Если очередь пуста, будет выброшено исключение.
6. **E removeLast():** Удаляет и возвращает элемент из конца очереди. Если очередь пуста, будет выброшено исключение.
7. **E pollFirst():** Удаляет и возвращает элемент из начала очереди. Если очередь пуста, возвращает null.
8. **E pollLast():** Удаляет и возвращает элемент из конца очереди. Если очередь пуста, возвращает null.
9. **E getFirst():** Возвращает элемент из начала очереди, не удаляя его. Если очередь пуста, будет выброшено исключение.
10. **E getLast():** Возвращает элемент из конца очереди, не удаляя его. Если очередь пуста, будет выброшено исключение.
11. **E peekFirst():** Возвращает элемент из начала очереди, не удаляя его. Если очередь пуста, возвращает null.
12. **E peekLast():** Возвращает элемент из конца очереди, не удаляя его. Если очередь пуста, возвращает null.

Интерфейс Set

Интерфейс **Set** определяет множество (набор).

Set расширяет **Collection** и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод add() возвращает false, если делается попытка добавить дублированный элемент в набор.

Интерфейс **Set** заботится об уникальности хранимых объектов, уникальность определяется реализацией метода equals().



Методы Set

1. **boolean add(E element):** Добавляет элемент в множество. Если элемент уже присутствует в множестве, метод возвращает false.
2. **boolean remove(Object element):** Удаляет указанный элемент из множества, если он присутствует. Возвращает true, если элемент был удален, и false в противном случае.
3. **boolean contains(Object element):** Проверяет, содержит ли множество указанный элемент. Возвращает true, если элемент присутствует, и false в противном случае.
4. **int size():** Возвращает количество элементов в множестве.
5. **boolean isEmpty():** Проверяет, является ли множество пустым. Возвращает true, если множество не содержит элементов, и false в противном случае.
6. **void clear():** Удаляет все элементы из множества.
7. **Iterator<E> iterator():** Возвращает итератор для обхода элементов в множестве.
8. **boolean addAll(Collection<? extends E> collection):** Добавляет все элементы из указанной коллекции в множество. Если какой-либо элемент уже присутствует в множестве, он будет проигнорирован. Если хотя бы один элемент был добавлен, метод возвращает true.
9. **boolean removeAll(Collection<?> collection):** Удаляет из множества все элементы, которые также присутствуют в указанной коллекции. Возвращает true, если множество изменилось в результате вызова метода.
10. **boolean retainAll(Collection<?> collection):** Удаляет из множества все элементы, кроме тех, которые также присутствуют в указанной коллекции. Возвращает true, если множество изменилось в результате вызова метода.
11. **boolean containsAll(Collection<?> collection):** Проверяет, содержит ли множество все элементы из указанной коллекции. Возвращает true, если все элементы присутствуют в множестве, и false в противном случае.

Реализации Set

1. HashSet: Реализация Setна основе хэш-таблицы. Элементы в HashSet не упорядочены* и могут быть доступны в произвольном порядке. HashSet позволяет хранить null элементы.
2. TreeSet: Реализация Set на основе сбалансированного дерева (обычно красно-черного дерева). Элементы в TreeSet хранятся в отсортированном порядке по их естественному порядку или с использованием заданного компаратора.
3. LinkedHashSet: Реализация Set, которая объединяет хэш-таблицу с двусвязным списком. Элементы в LinkedHashSet упорядочены в порядке их вставки.

Особенности HashSet

- Выгода от хеширования состоит в том, что оно обеспечивает постоянство время выполнения операций add(), contains(), remove() и size(), даже для больших наборов.
- Класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств.
- Фактически под копотом HasSet - находится HashMap а сама структура HashSet - это набор ключей HashMap. Подробнее работу с хэштаблицами можно будет увидеть далее при разборе Map

Контракт методов equals() и hashCode()

- Для одного и того же объекта, хеш-код всегда будет одинаковым.
- Если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот).
- Если хеш-коды равны, то входные объекты не всегда равны.
- Если хеш-коды разные, то и объекты гарантированно будут разные.

Свойства методов equals() и hashCode()

- **Рефлексивность**: для любой ссылки на значение x, x.equals(x) вернет true;
- **Симметричность**: для любых ссылок на значения x и y, x.equals(y) должно вернуть true, тогда и только тогда, когда y.equals(x) возвращает true.
- **Транзитивность**: для любых ссылок на значения x, y и z, если x.equals(y) и y.equals(z) возвращают true, тогда и x.equals(z) вернёт true;
- **Непротиворечивость**: для любых ссылок на значения x и y, если несколько раз вызвать x.equals(y), постоянно будет возвращаться значение true либо постоянно будет возвращаться значение false при условии, что никакая информация, используемая при сравнении объектов, не поменялась.
- Для любой ненулевой ссылки на значение x выражение x.equals(null) должно возвращать false.

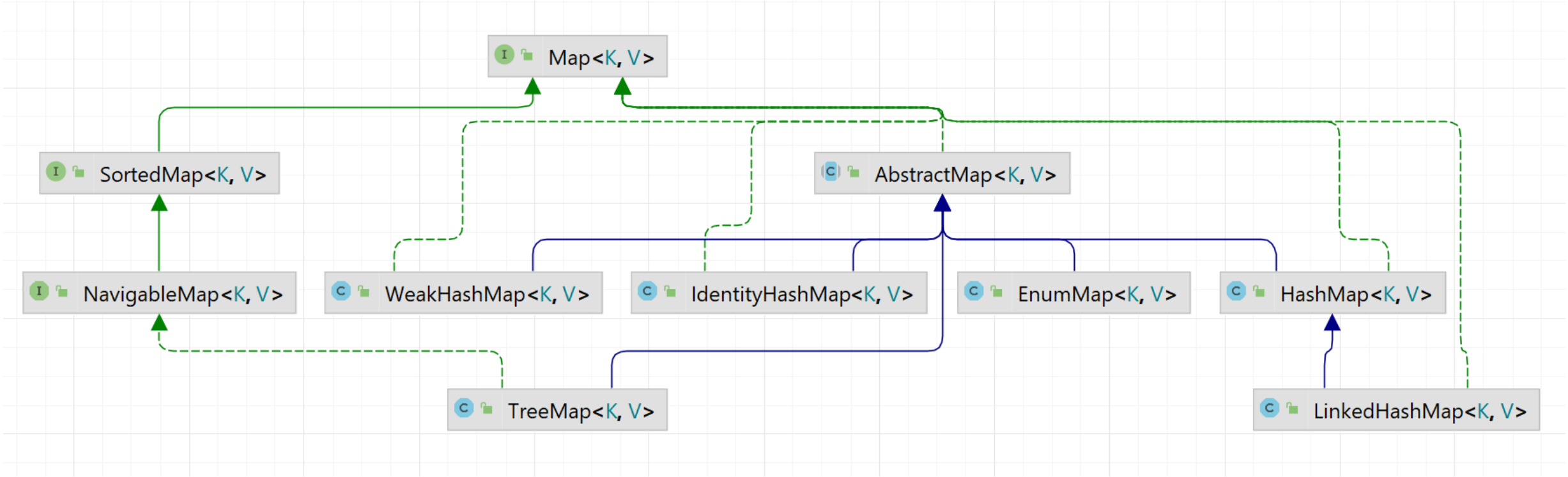
Интерфейс Map

Отображение (или карта) представляет собой объект, сохраняющий связи между ключами и значениями в виде пар "ключ-значение". По заданному ключу можно найти его значение.

Ключи и значения являются объектами. Ключи должны быть уникальными, а значения могут быть дублированными. Уникальность ключей определяет реализация метода equals().

Для корректной работы с картами необходимо переопределить методы equals() и hashCode(). Допускается добавление объектов без переопределения этих методов, но найти эти объекты в Map вы не сможете.

Иерархия Map



Методы Map

1. **V put(K key, V value):** Добавляет элемент с указанным ключом и значением в Map. Если элемент с таким ключом уже существует, его значение будет обновлено, и предыдущее значение будет возвращено.
2. **V get(Object key):** Возвращает значение, связанное с указанным ключом в Map. Если ключ не найден, возвращается null.
3. **boolean containsKey(Object key):** Проверяет, содержит ли Map элемент с указанным ключом. Возвращает true, если ключ присутствует, и false в противном случае.
4. **boolean containsValue(Object value):** Проверяет, содержит ли Map элемент с указанным значением. Возвращает true, если значение присутствует, и false в противном случае.
5. **V remove(Object key):** Удаляет элемент с указанным ключом из Map и возвращает его значение. Если ключ не найден, возвращается null.
6. **int size():** Возвращает количество элементов в Map.
7. **boolean isEmpty():** Проверяет, является ли Map пустым. Возвращает true, если Map не содержит элементов, и false в противном случае.
8. **void clear():** Удаляет все элементы из Map.
9. **Set<K> keySet():** Возвращает набор всех ключей в Map в виде Set.
10. **Collection<V> values():** Возвращает коллекцию всех значений в Map.
11. **Set<Map.Entry<K, V>> entrySet():** Возвращает набор всех элементов (пар "ключ-значение") в Map в виде Set<Map.Entry>.

Основные классы Map

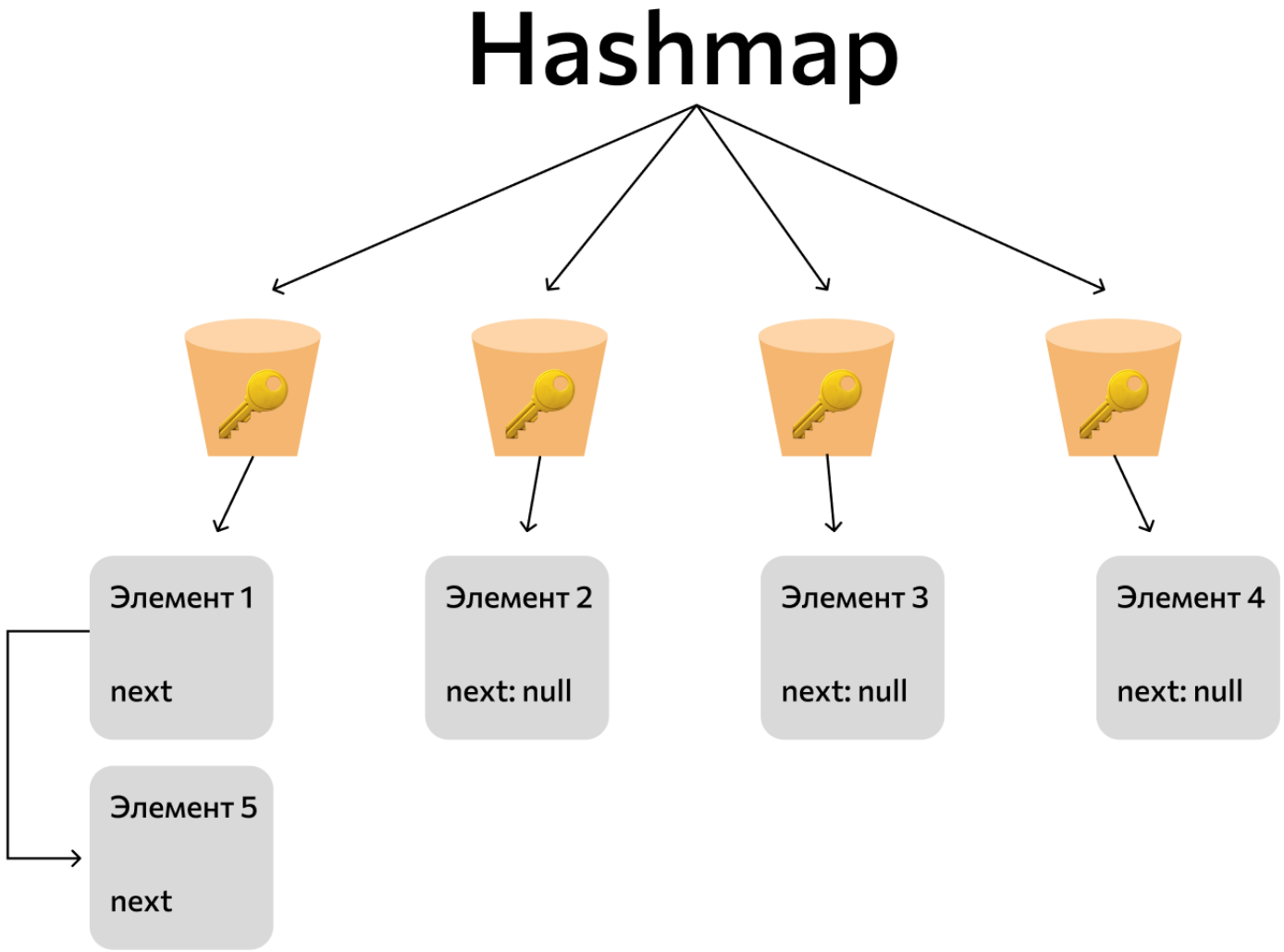
- **AbstractMap<K, V>** - реализует интерфейс Map<K, V>;
- **HashMap<K, V>** - расширяет AbstractMap<K, V>, используя хэш - таблицу, в которой ключи отсортированы относительно значений их хэш-кодов;
- **TreeMap<K, V>** - расширяет AbstractMap<K, V>, используя дерево, где ключи расположены в виде дерева поиска в строгом порядке.
- **WeakHashMap<K, V>** - позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения.
- **LinkedHashMap<K, V>** - запоминает порядок добавления объектов в карту и образует при этом дважды связанный список ключей.

Добавление в HashMap

При добавлении нового элемента в HashMap с помощью метода put(key, value) выполняются следующие действия:

- Высчитывается значения hashCode у ключа с помощью одноименной функции
- Определяется бакет(ячейка массива) в которую будет добавлен новый элемент. Номер определяется по остатку от деления хэшкода на кол-во ячеек. В более новых версиях Java с помощью бинарного сдвига.
- Далее, если бакет пустой - то элемент просто добавляется. Если не пустой, то, там LinkedList.
- Если бакет не пустой - мы идем по этому списку и сравниваем ключ добавляемого элемента и ключ элемента в списке по хэшкодам.
- Если хэшкоды неравны, то идем к следующему элементу
- Если хэшкоды равны, то далее сравниваем по Equals.
- Если ключи равны по Equals, то перезаписываем value по этому ключу
- Если ключи не равны по Equals, то переходим к следующему элементу
- Если мы не нашли ключ в списке, то мы добавляем этот элемент в конец списка

Добавление в HashMap



Big O

Big-O Notation — это математическая функция, используемая в информатике для описания сложности алгоритма. Обычно это мера времени выполнения, необходимого для выполнения алгоритма.

Но вместо того, чтобы сообщать вам, насколько быстро или медленно выполняется алгоритм, он сообщает вам, как производительность алгоритма изменяется в зависимости от размера ввода (размера n).

Big O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
	Quadratic
	Cubic
	Exonential
O(n!)	Factorial

Временная сложность

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

Литература

- https://habr.com/ru/post/310794/
- https://habr.com/ru/post/422259/
- https://habr.com/ru/post/156361/
- https://habr.com/ru/company/netologyru/blog/334914/
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

Вопросы

Как сравниваются элементы коллекций?

Почему Map не относиться к Collection?

Что такое итератор и как он работает в коллекциях?

Как можно перебрать все значения HashMap?

Что будет если обект который выступал в качестве ключа в структуре MAP поменяется?

Как перебрать все значения HashMap?

Разница между Iterable и Iterator.

Что будет если вызвать Iterator.next()

Что будет если добавит еще один элемент в коллекцию во время работы итератора