



# Структуры данных и Java коллекции



# Структуры данных

Структура данных – это контейнер, который хранит информацию в определенном виде (организованной форме).

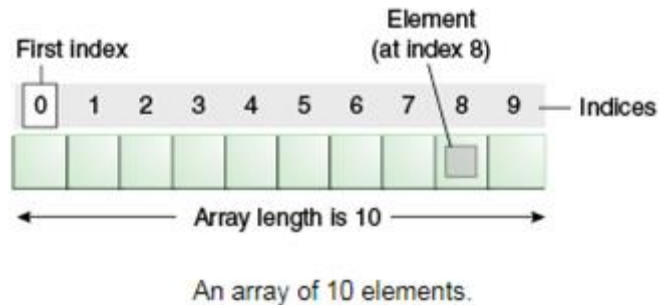
## **Наиболее часто используемые структуры данных:**

- Массив (Array)
- Стек (Stack)
- Очередь (Queue)
- Связный список (Linked List)
- Граф (Graph)
- Дерево (Tree)
- Префиксное дерево (Trie)
- Хэш-Таблица (Hash Table)



# Массив

Массивы занимают изначально определенный отрезок памяти который задается его размером. Поскольку все размеры ячеек одинаковые, массив обеспечивает доступ к ячейке за КОНСТАНТНОЕ время  $O(1)$ !



Ячейка = адрес в памяти + размер ячеек \* количество ячеек

Если предположить, что тип `int` занимает в памяти 2 байта, то адрес элемента, соответствующего индексу 3, вычисляется так: начальный адрес + 3 \* 2. Начальный адрес — это адрес ячейки памяти, начиная с которой располагается массив. Он формируется во время выделения памяти под массив.

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

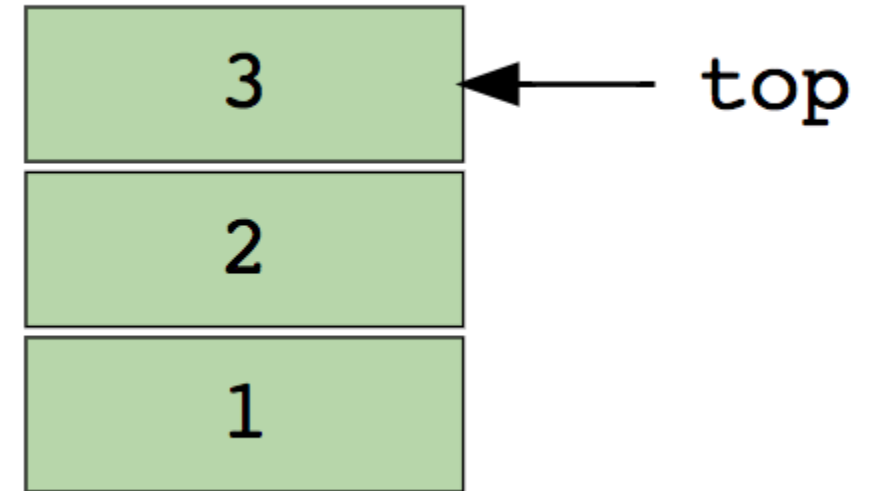


Стеки работают по принципу **LIFO** (Last In First Out, последним пришел – первым ушел).

Пример стека из реальной жизни – куча книг, лежащих друг на друге. Чтобы получить книгу, которая находится где-то в середине, вам нужно удалить все, что лежит сверху.

Основные операции со стеками:

- **Push** – вставка элемента наверх стека.
- **Pop** – получение верхнего элемента и его удаление.
- **Peek** – возвращает верхний элемент стека, но не удаляет его из стека



# Очередь

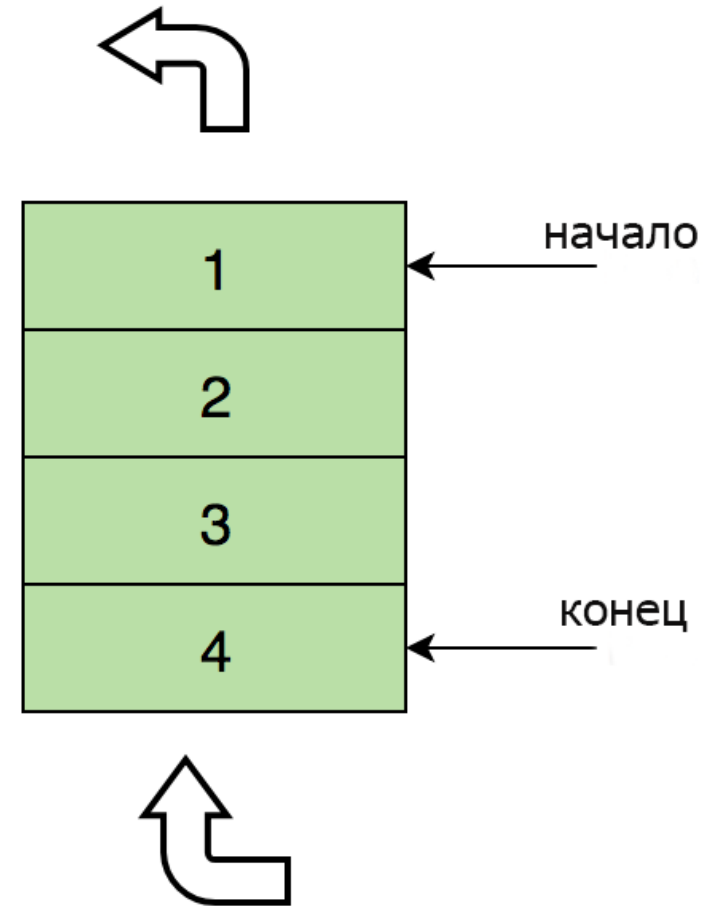


Очереди работают по принципу **FIFO** (FIRST IN—FIRST OUT, “первым вошел — первым вышел”).

Основаны на двух разных подходах – массива и очереди.

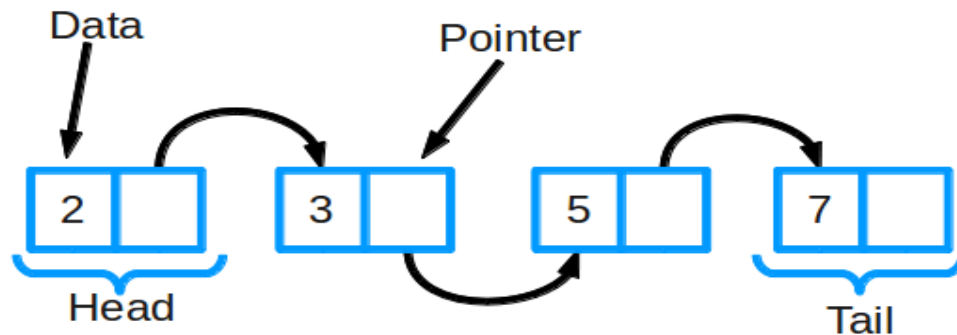
Указатель всегда смотрит на голову списка

Реализуется при реверса стека.



# 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 связанного 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.

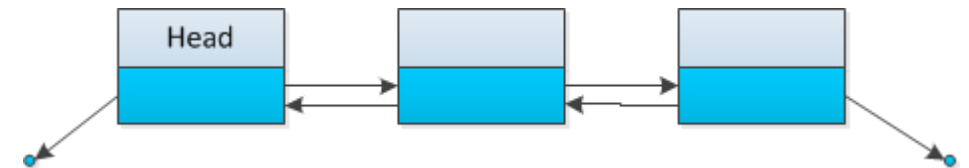
# Связанный список



**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





# Графы

**Граф** – это система, состоящая из точек и линий, которые их соединяют. Точки называются **вершинами** графа, а линии – **ребрами** графа.

Если в графе используются стрелки, его называют **ориентированным** графом, если просто линии – **неориентированным** графом.

У каждой вершины может быть свое количество ребер. Также вершина может не иметь ребер вообще. Или наоборот, быть соединена ребрами со всеми остальными вершинами. Если в графе каждая вершина соединена ребром с каждой – такой граф называют **полным**.

Если в графе по ребрам можно добраться до любой вершины, такой граф называют **связным**. Граф состоящий из трех отдельных вершин, без ребер вообще, это все равно граф, но **несвязный**.



# Графы

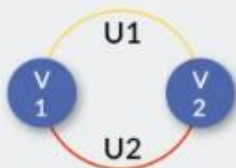


## Некоторые типы графов

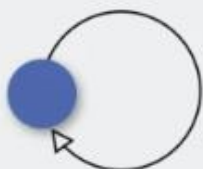
Нуль-граф (граф без рёбер)



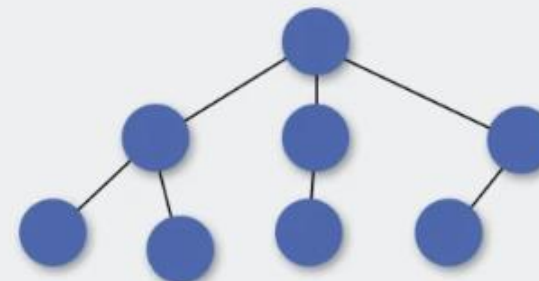
Мультиграф (граф, содержащий кратные рёбра).  
**U1, U2** — кратные рёбра или мультирёбра



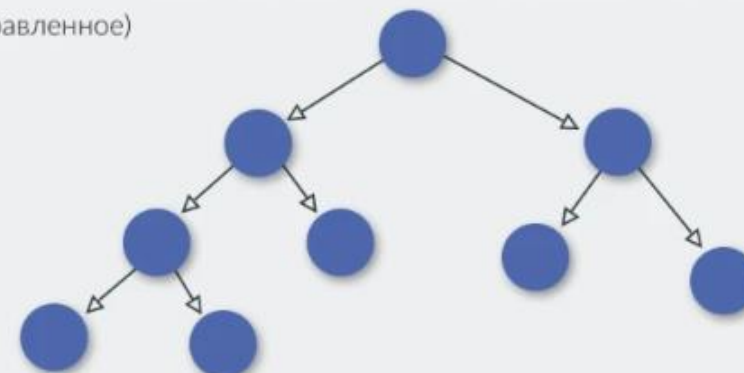
Петля



Дерево — связный граф без циклов, то есть без петель и кратных рёбер



Ориентированное (направленное) дерево



# Дерево

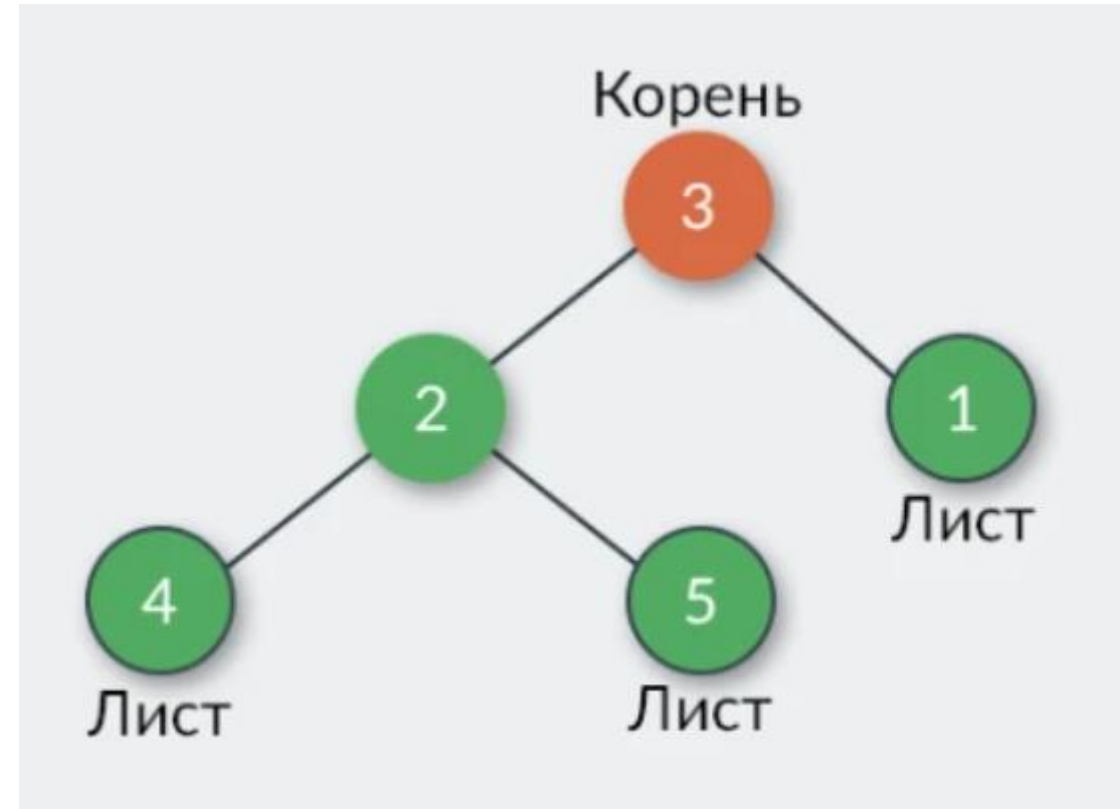
Чтобы соединить в связный граф  $N$  вершин, надо минимум  $N-1$  ребер. Такой граф называется деревом

- Каждое дерево имеет корневой узел (вершину).
- Остальные вершины называют ветвями
- Ветви дерева, которые не имеют своих ветвей, называют листьями

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.

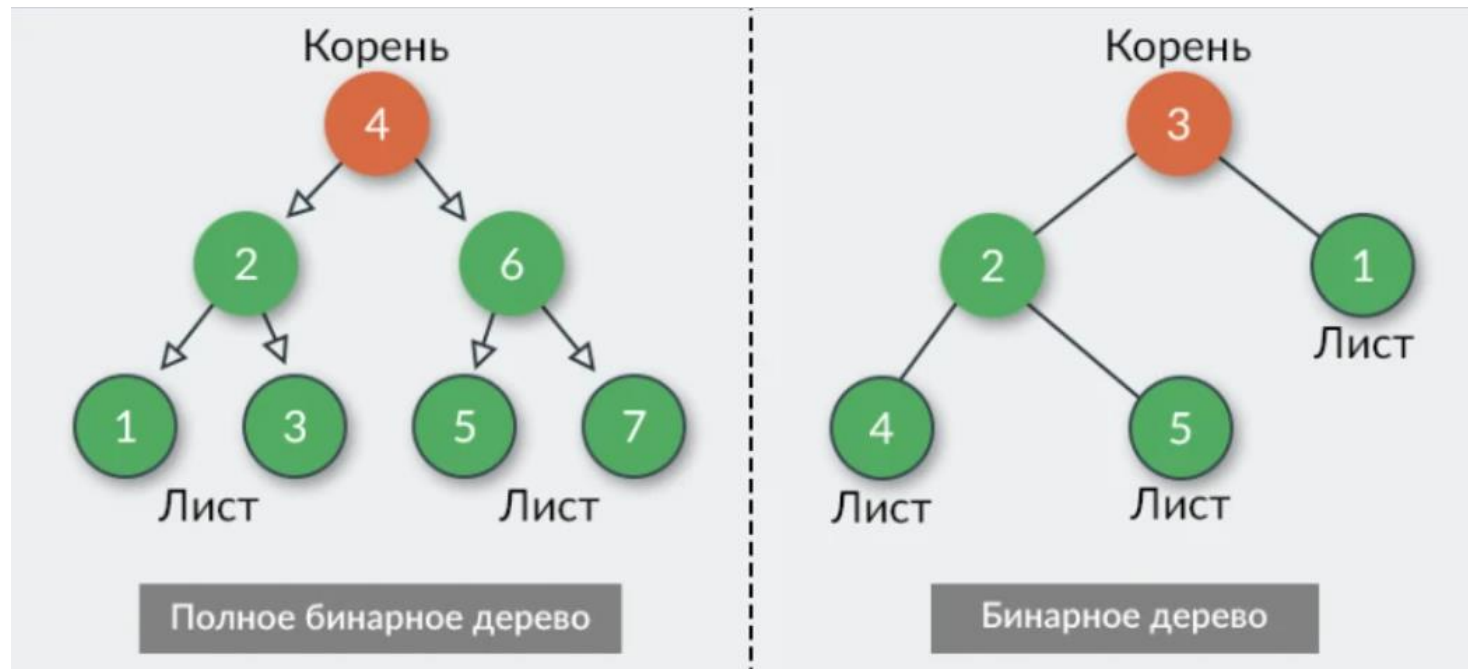
Двоичные деревья поиска позволяют быстро находить, добавлять и удалять элементы. Они устроены так, что время каждой операции пропорционально логарифму общего числа элементов в дереве.



# Дерево

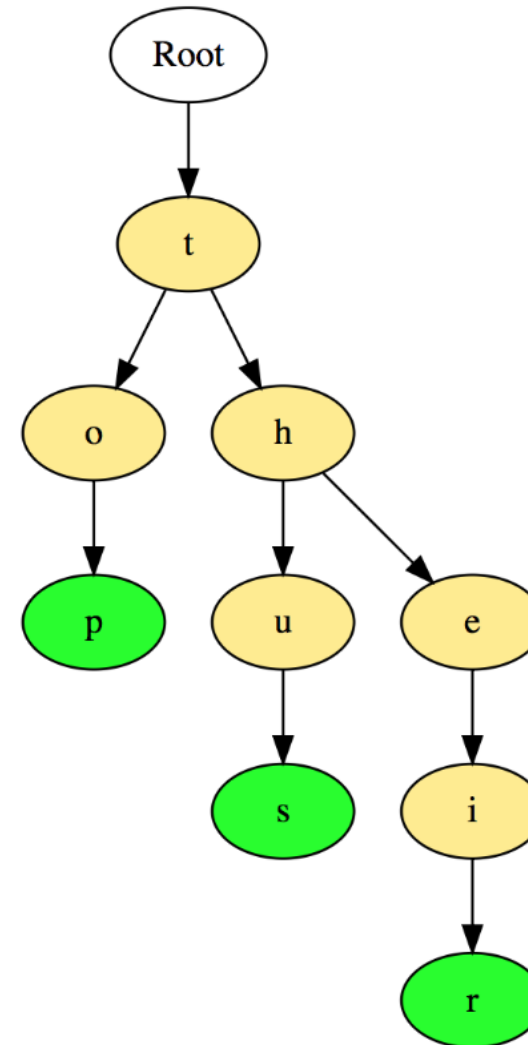
Дерево называют **бинарным**, если у каждого элемента дерева не более двух потомков. Т.е. их может быть 0, 1 или 2. Выше справа как раз изображено бинарное дерево.

Дерево называют **полным бинарным деревом**, когда у каждой ветви 2 потомка, а все листья (без потомков) находятся в одном ряду.

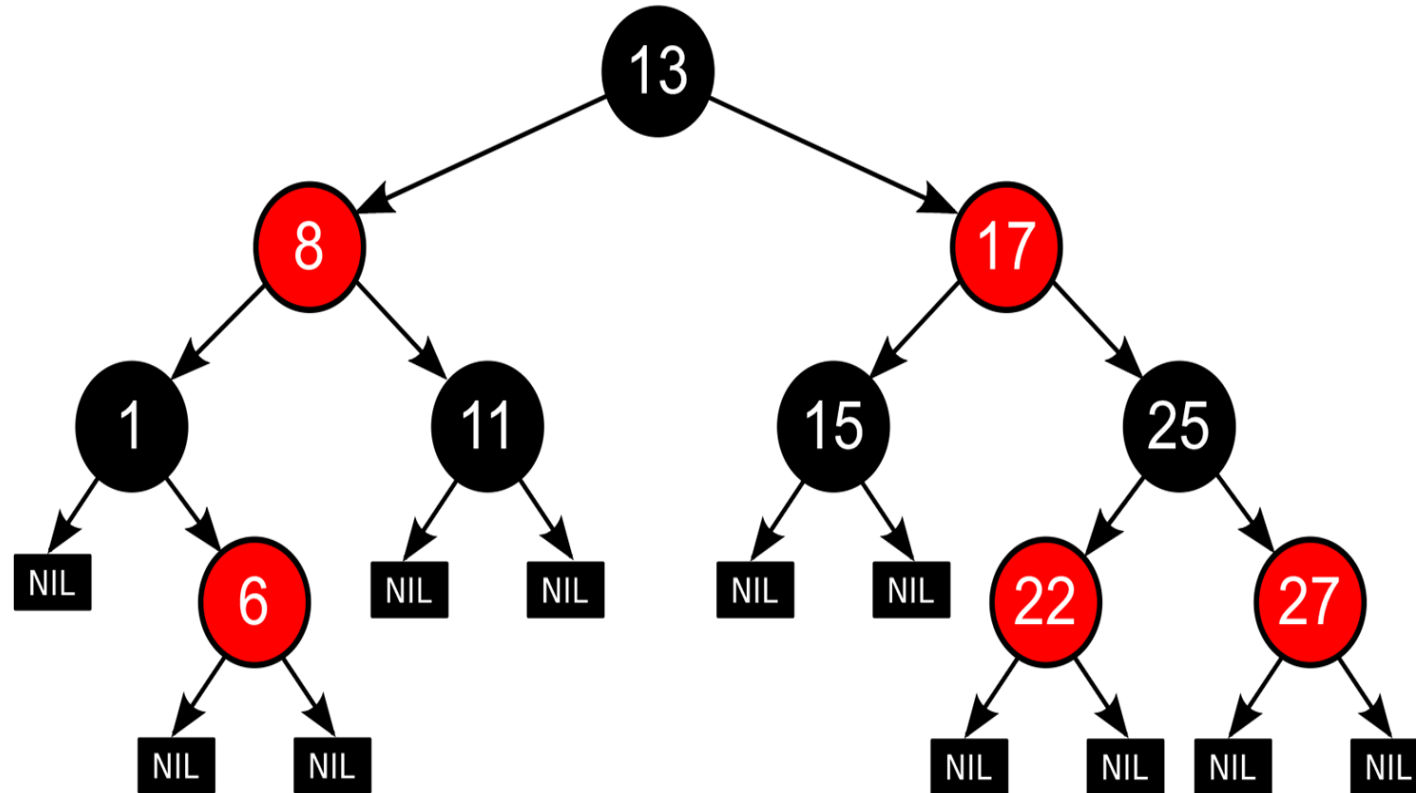


# Префиксное дерево

**Префиксные деревья (tries)** – древовидные структуры данных, эффективные для решения задач со строками. Они обеспечивают быстрый поиск и используются преимущественно для поиска слов в словаре, автодополнения в поисковых системах и даже для IP-маршрутизации. Слова размещаются сверху вниз. Выделенные зеленым элементы показывают конец каждого слова.



# Красно-чёрное дерево



**Красно-черное дерево** - это вид бинарного дерева, основной сутью которого является способность к самобалансировке.

Принципы организации (свойства) КЧД:

1. Корень дерева черный.
2. Все листья, не содержащие данных, черные
3. Оба потомка каждого красного узла - черные
4. Черная глубина любого листа одинакова (черной глубиной называют количество черных вершин на пути из корня).

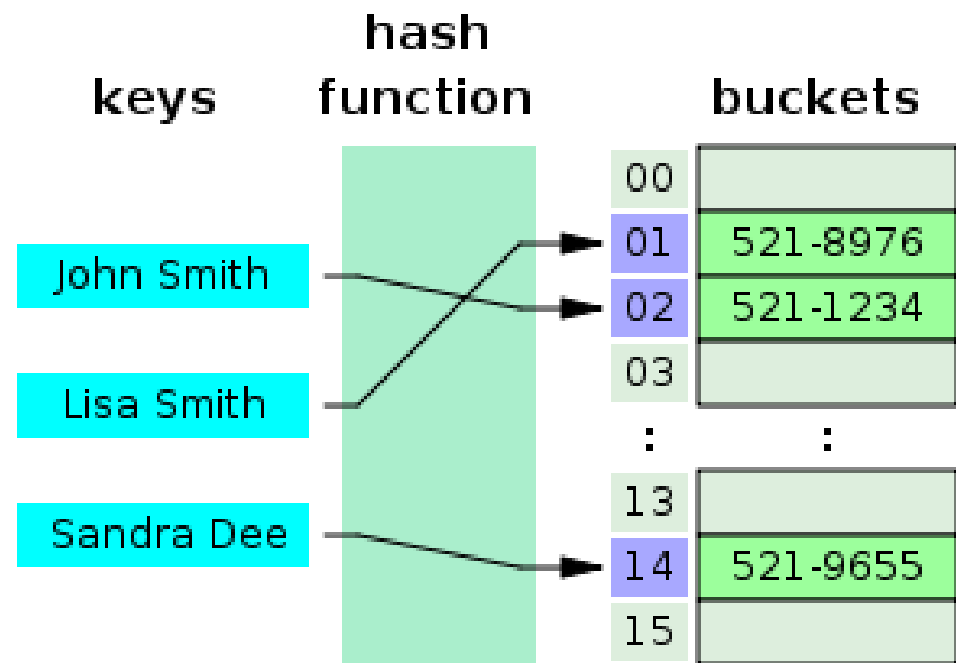


# Хэш-таблицы

**Хеш-таблица** — это структура данных для хранения пар ключей и их значений. По сути она представляет собой массив, где местоположение элемента зависит от значения самого элемента. Связь между значением элемента и его позицией в хеш-таблице задает хеш-функция.

Обычно хэш-функция принимает строку символов в качестве входных 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.



# 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.

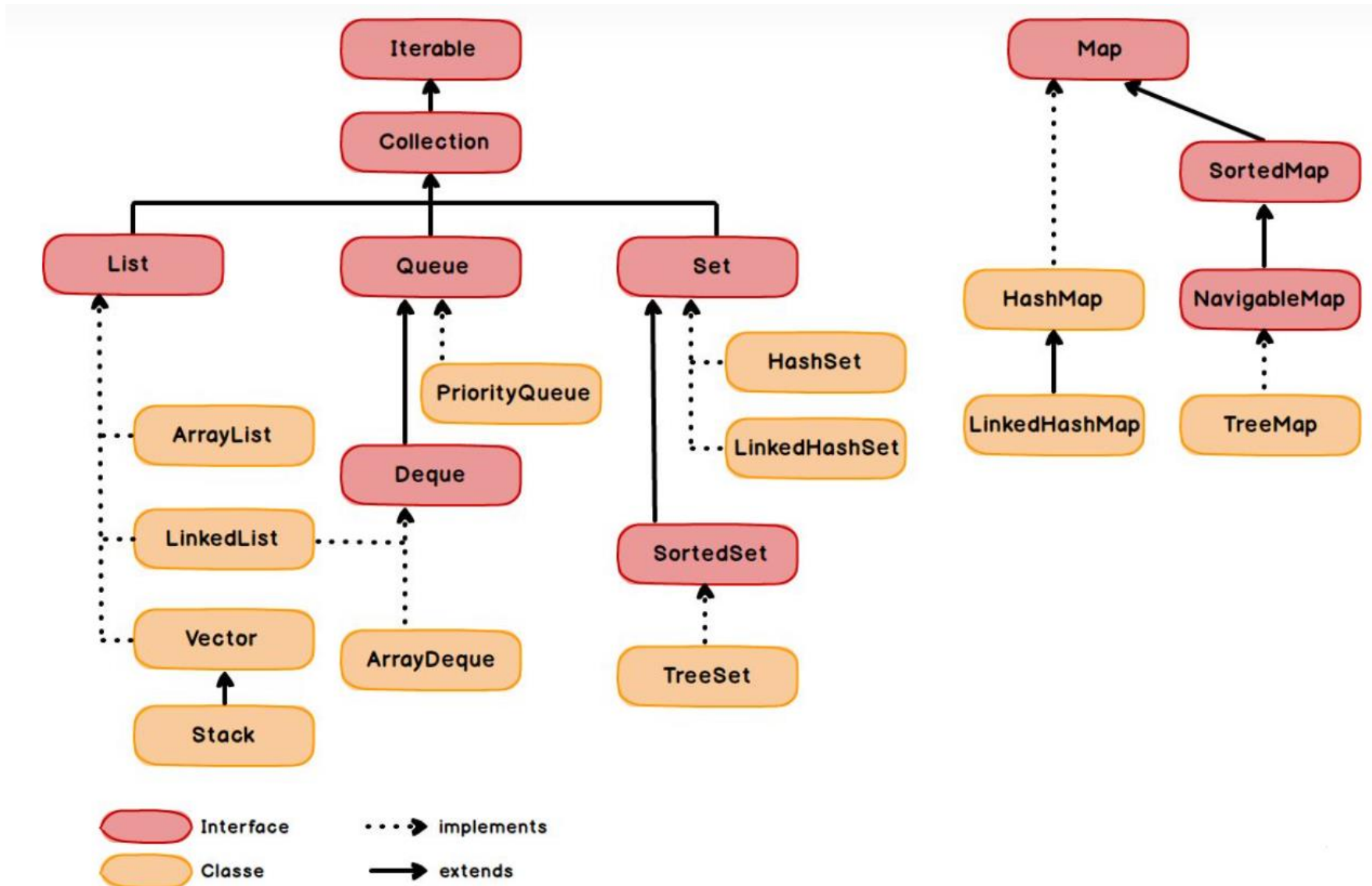


# Коллекции

- 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 коллекций





# 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()**

# Iterator



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.



# Iterator

Использование итератора не очень удобно – слишком много лишнего и очевидного кода. Ситуация упростилась, когда в Java появился цикл по итератору – `foreach`.

Было	Стало
<pre>1  TreeSet&lt;String&gt; set = new TreeSet&lt;String&gt;(); 2  Iterator&lt;String&gt; iterator = set.iterator(); 3 4  while (iterator.hasNext()) 5  { 6      String item = iterator.next(); 7      System.out.println(item); 8  }</pre>	<pre>1  TreeSet&lt;String&gt; set = new TreeSet&lt;String&gt;() 2 3  for(String item : set) 4  { 5      System.out.println(item); 6  }</pre>

Это один и тот же код! Итератор используется и там, и там.

# 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 можно получить вызывая метод listIterator() для коллекций, реализующих List.

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



```
import java.util.Arrays;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorDemo {
    public static void main(String[] args) {
        List<String> arrayList = Arrays.asList("A", "B", "C", "D");

        ListIterator<String> listIterator = arrayList.listIterator();
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            listIterator.set(element + "+");
        }

        System.out.print("Измененный arrayList в обратном порядке: ");
        while (listIterator.hasPrevious()) {
            String element = listIterator.previous();
            System.out.print(element + " ");
        }
    }
}
```



# 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.



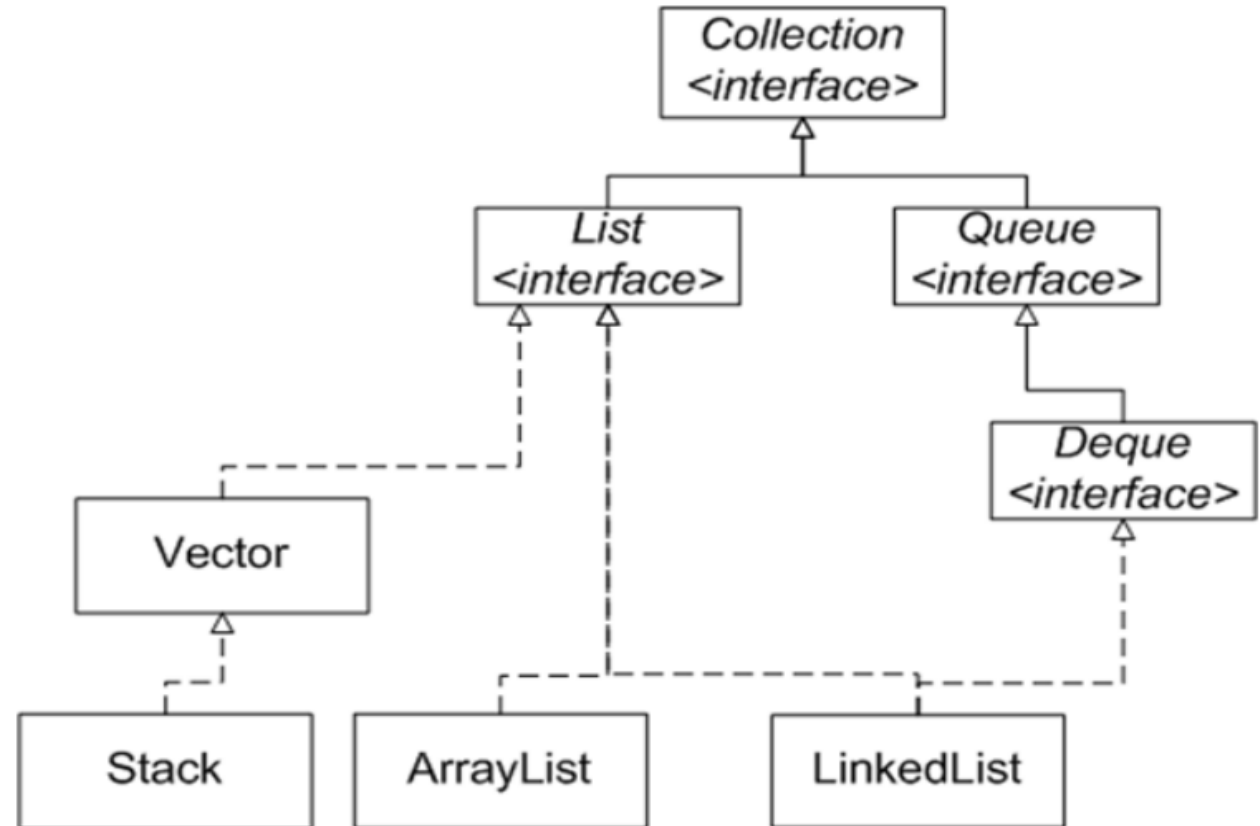
# Методы интерфейса List

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)
- Stack (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 (при наличии свободных ячеек) и удаление элементов с конца. Скорость операций -  $O(1)$ .

## 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.

# Добавление элемента



• Car maserati  
↓

ferrari	bugatti	ford	lambo	volvo	renault	fiat	kia	honda	toyota
---------	---------	------	-------	-------	---------	------	-----	-------	--------



Места нет!  
Создаем новый массив и копируем туда элементы



ferrari	bugatti	ford	lambo	volvo	renault	fiat	kia	honda	toyota

Новый массив на 16 ячеек.  
Все элементы из предыдущего массива скопированы



# Вставка когда массив полон

## Итоговый результат

ferrari	bugatti	ford	maserati	lambo	volvo	renault	fiat	kia	honda
toyota									

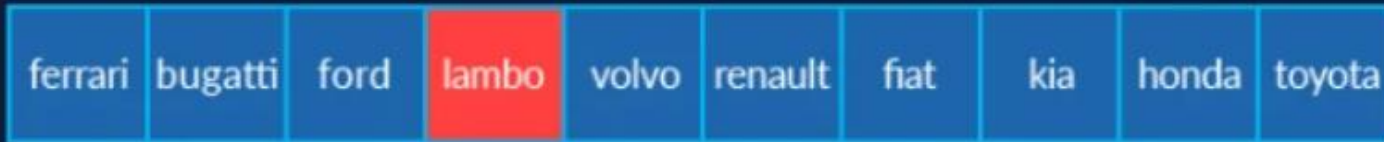
- Создается новый массив размером, в 1.5 раза больше исходного, плюс один элемент.
- Все элементы из старого массива копируются в новый массив
- Новый массив сохраняется во внутренней переменной объекта ArrayList, а старый массив объявляется мусором.



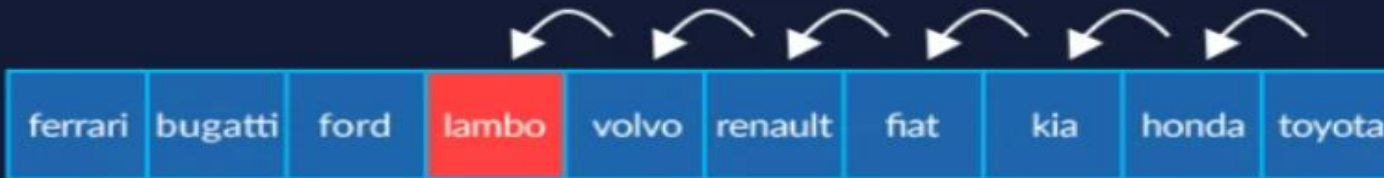
# Удаление элемента



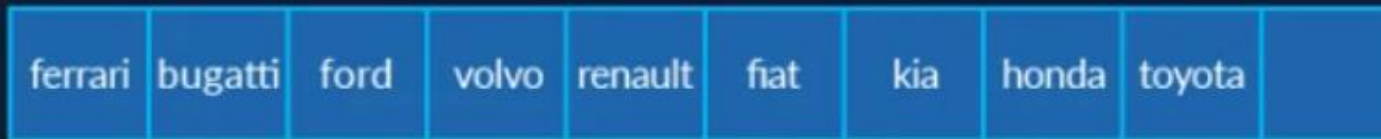
Удаляем элемент **lambo** с помощью метода **remove ()**



Перемещение элементов на одну ячейку влево



Итоговый результат





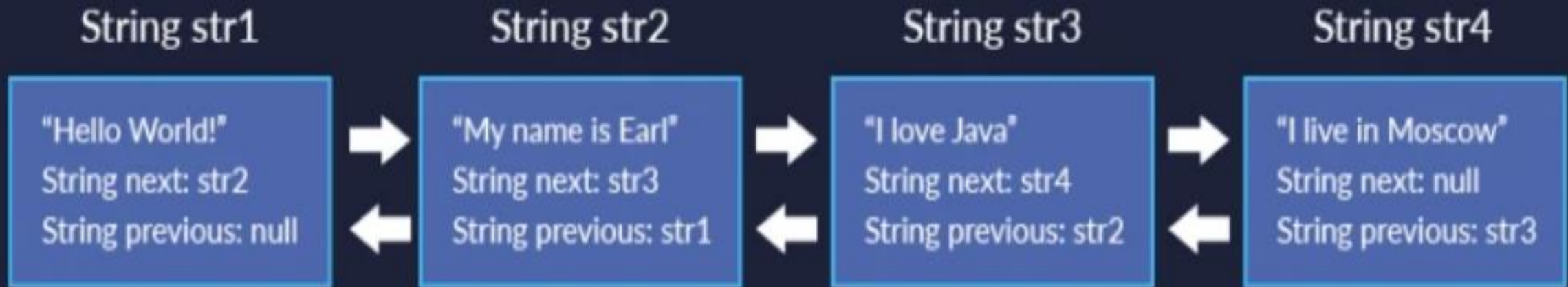


# LinkedList

**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 конец списка.

# Структура LinkedList



## Строение LinkedList

В данном случае он состоит из 4 элементов-строк (String). Каждый элемент помимо содержимого (строки с текстом) хранит ссылку на следующий и предыдущий элемент.

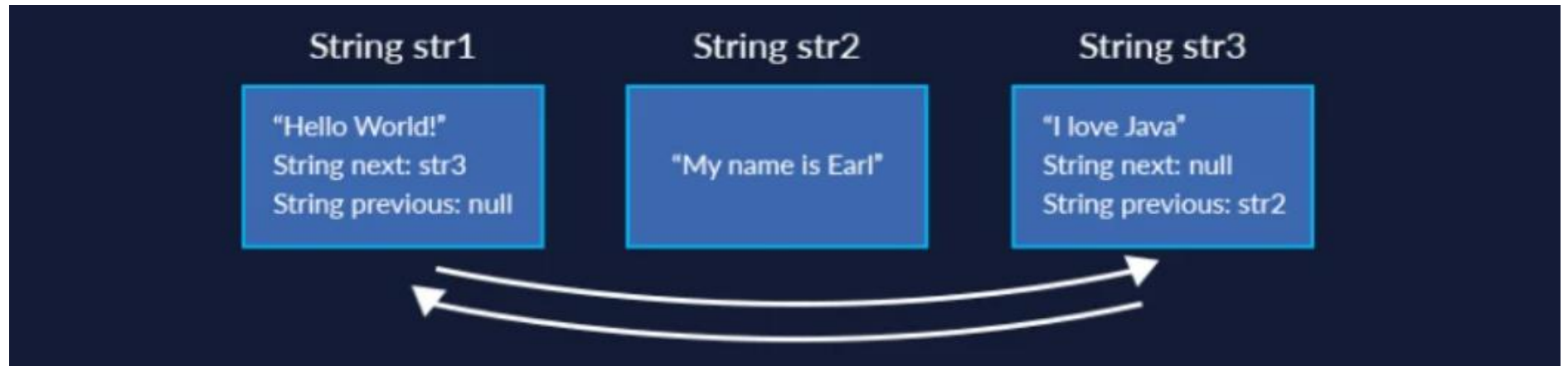
# Добавление в LinkedList



# Вставка в середину LinkedList



# Удаление из середины LinkedList







# Достоинства и недостатки LinkedList


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

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.

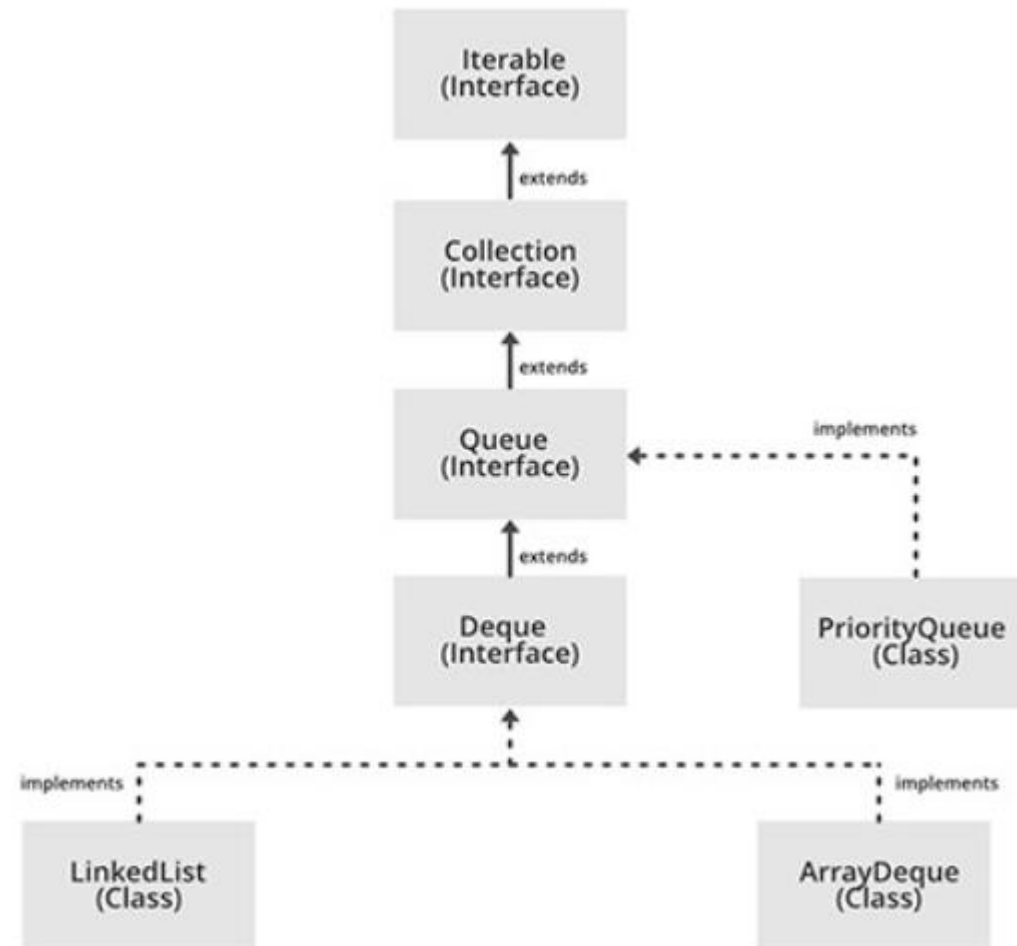
**LinkedList** обычно используется, если необходимо часто добавить или удалить элементы в списке, особенно в начале списка. Либо если нам нужна вставка элемента в конец за гарантированное время



# Сравнение LinkedList и ArrayList

Algorithm	ArrayList 	LinkedList
Get the element at the front	$O(1)$	$O(1)$
Get the element at the end	$O(1)$	$O(1)$
Get element at a specific index	$O(1)$	$O(N)$
Insert an element at the front	$O(N)$	$O(1)$
Insert an element at the back	$O(1)$	$O(1)$
insert at a specific index	$O(N)$	$O(N)$

# Интерфейс Queue





# 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



```
1 import java.util.Queue;
2 import java.util.LinkedList;
3
4 public class QueueExample {
5     public static void main(String[] args) {
6         // Создание очереди
7         Queue<String> taskQueue = new LinkedList<>();
8
9         // Добавление элементов в очередь
10        taskQueue.offer("Task 1");
11        taskQueue.offer("Task 2");
12        taskQueue.offer("Task 3");
13        taskQueue.offer("Task 4");
14
15        // Обработка элементов в очереди
16        while (!taskQueue.isEmpty()) {
17            String task = taskQueue.poll(); // Получение и удаление элемента из начала очереди
18            System.out.println("Processing task: " + task);
19        }
20    }
21 }
```



# 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:

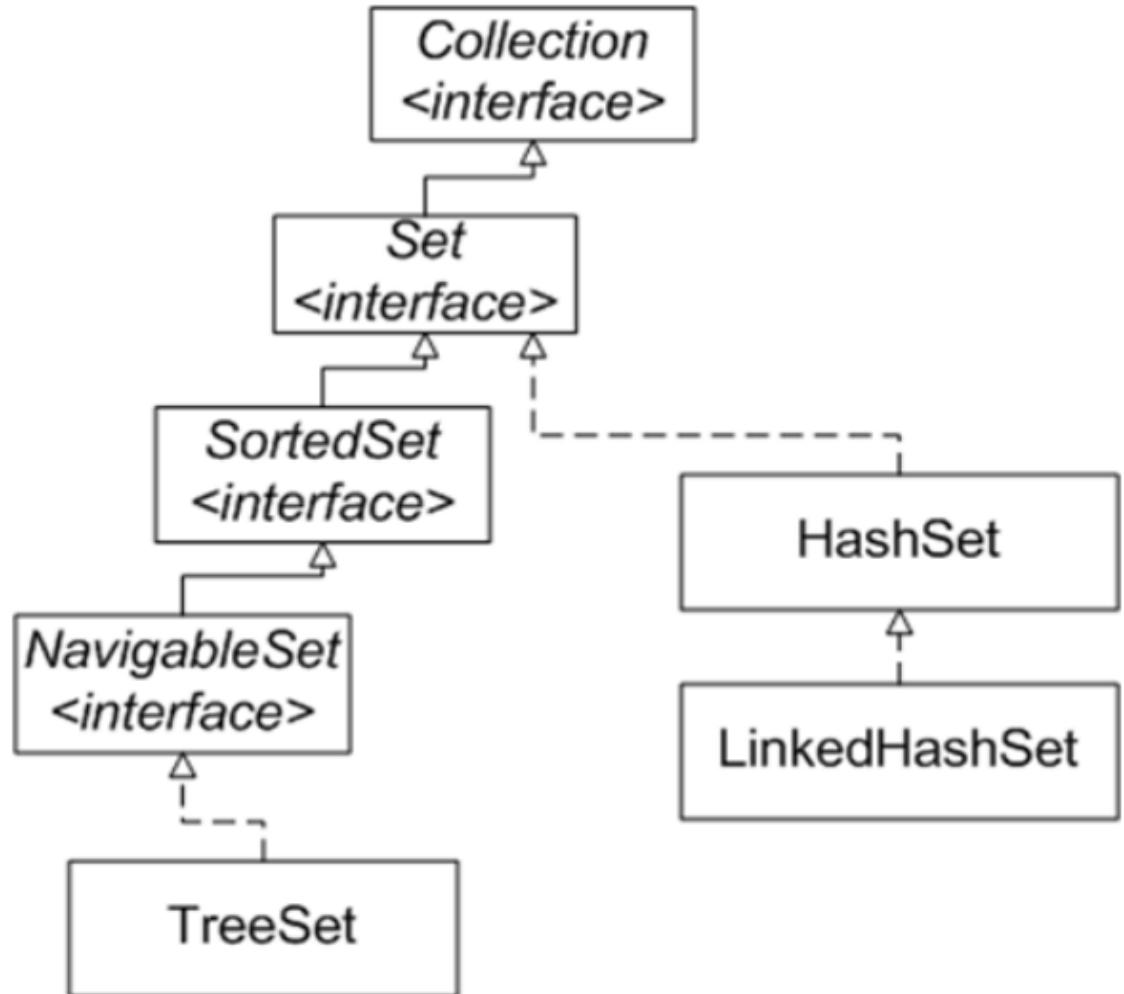
1. **void addFirst(E element):** Добавляет элемент в начало очереди.
2. **void addLast(E element):** Добавляет элемент в конец очереди.
3. **boolean offerFirst(E element):** Добавляет элемент в начало очереди. Если очередь заполнена и не может принять новый элемент, возвращает false.
4. **boolean offerLast(E 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 `Set` на 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.





# Контракт методов equals() и hashCode()

- Для одного и того же объекта, хеш-код всегда будет одинаковым.
- Если объекты одинаковые (по equals), то и хеш-коды одинаковые (но не наоборот).
- Если хеш-коды равны, то объекты по equals 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, если x!=null;
- **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;
- **Согласованность.** 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.
- **Сравнение с null.** Для любой ненулевой ссылки на значение x выражение x.equals(null) должно возвращать false.



# 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 капотом” `HashSet` - находится `HashMap` а сама структура `HashSet` - это набор ключей `HashMap`. Подробнее работу с хэш-таблицами можно будет увидеть далее при разборе `Map`.



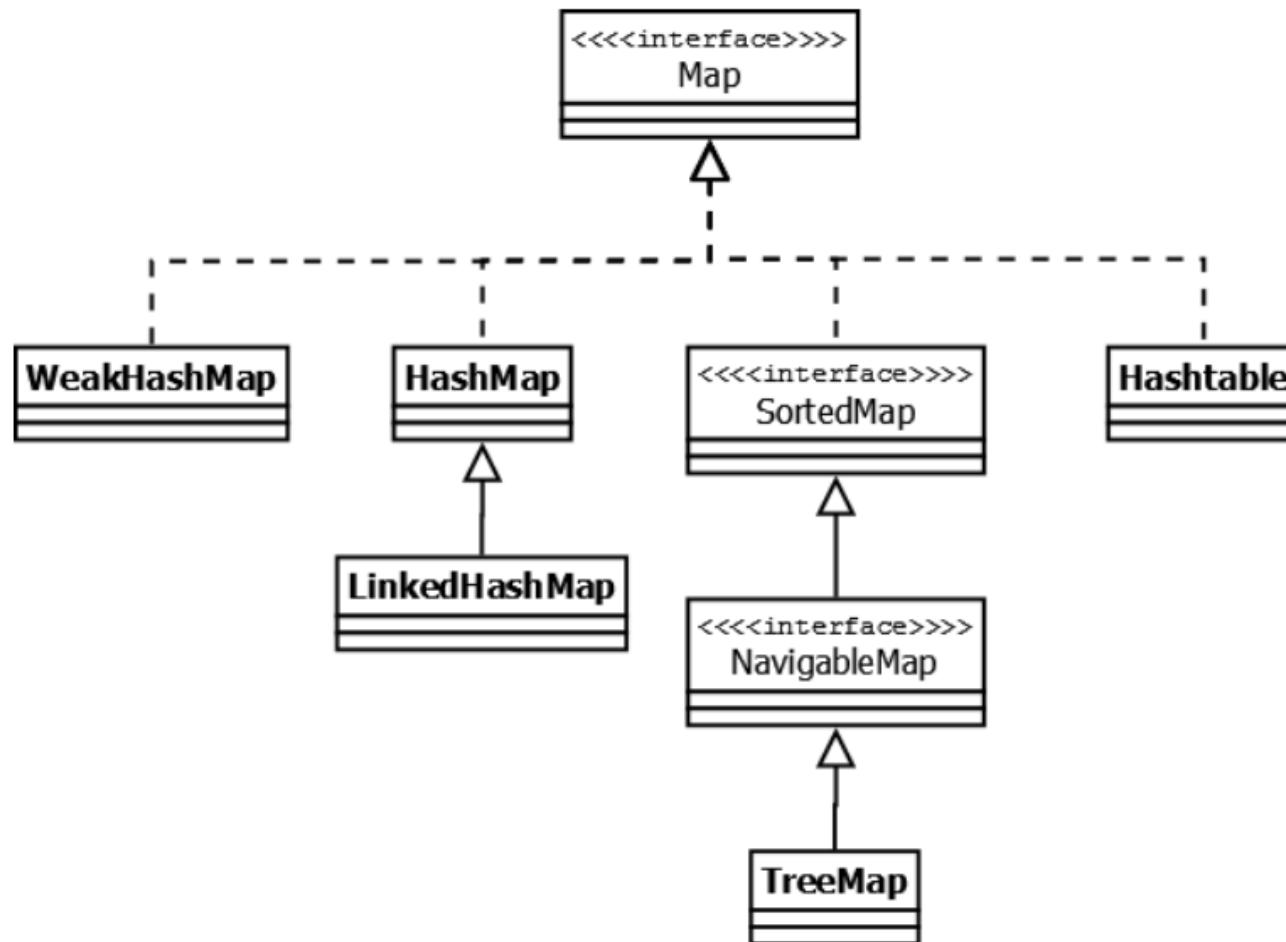
# Интерфейс Map

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.

Для корректной работы с картами необходимо переопределить методы `equals()` и `hashCode()`.

# 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:

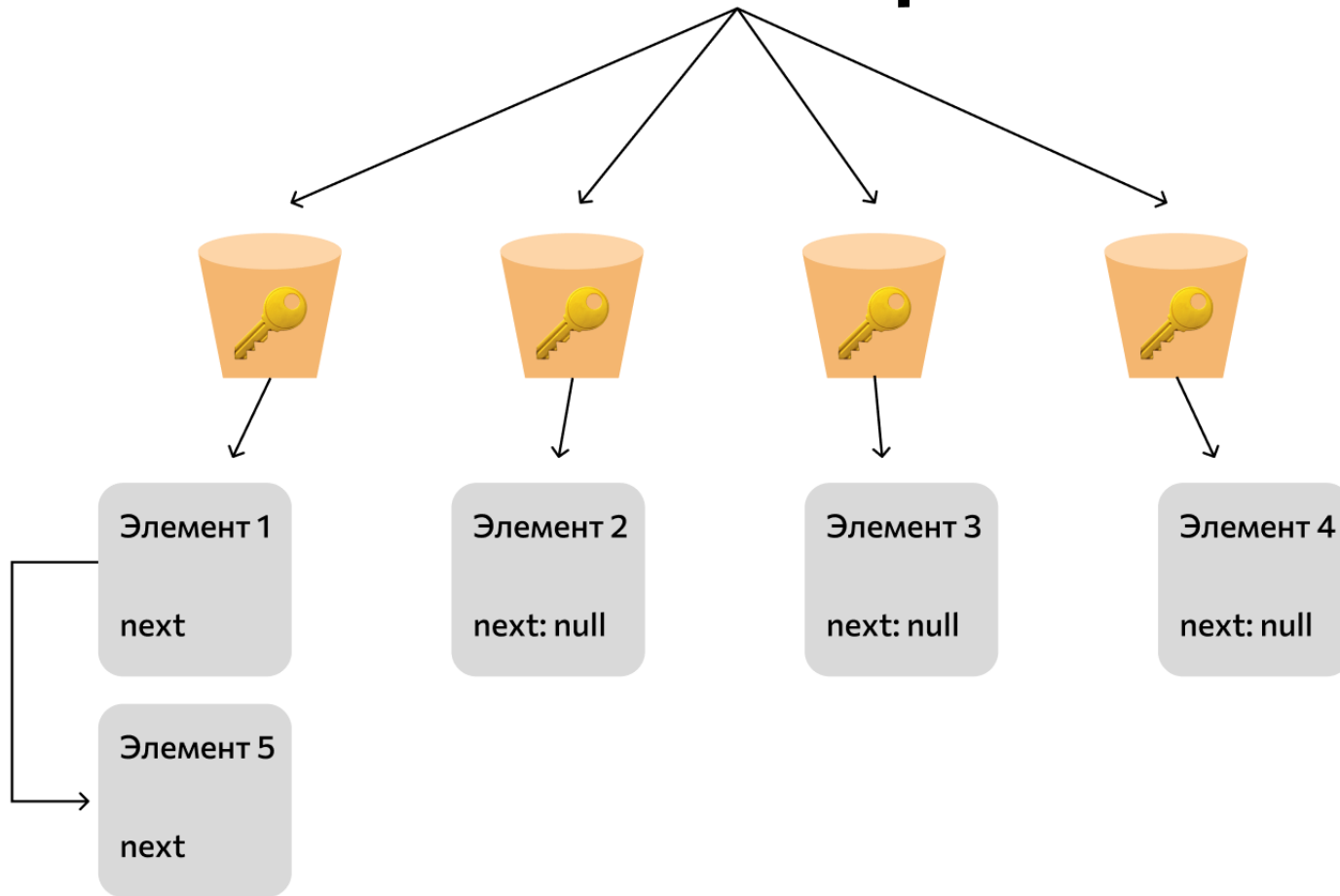
- Вычисляется значения `hashCode` у ключа с помощью одноименной функции
- Определяется бакет (ячейка массива) в которую будет добавлен новый элемент. Номер определяется по остатку от деления хэш-кода на кол-во ячеек. В более новых версиях Java с помощью бинарного сдвига.
- Далее, если бакет пустой - 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 хэшкам.
- Если хэшcodes неравны, то идем к следующему элементу
- Если хэшcodes 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



## HashMap



Элемент в бакете

Node	
hash	2306996
key	"KING"
value	100
next	null

# Big O

Big-O Notation — это математическая функция, используемая в информатике для описания сложности алгоритма. Она призвана показать, как сильно увеличится количество операций при увеличении размера данных

Big O	Name
$O(1)$	Константная
$O(\log n)$	Логарифмическая
$O(n)$	Линейная
$O(n \log n)$	Квазилинейная
$O(n^2)$	Квадратичная
$O(2^n)$	Экспоненциальная
$O(n!)$	Факториальная

# 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	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

# Основные алгоритмы

1. Поиск
  - a. Линейный поиск
  - b. Бинарный поиск
2. Сортировка
  - a. Пузырьковая сортировка
  - b. Сортировка вставками
  - c. Сортировка выбором
  - d. Сортировка слиянием
  - e. Быстрая сортировка
  - f. Пирамидальная сортировка

# Литература



- [https://ru.hexlet.io/courses/algorithms-trees/lessons/prefix/theory\\_unit](https://ru.hexlet.io/courses/algorithms-trees/lessons/prefix/theory_unit)
- <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>
- <https://javarush.com/groups/posts/2496-podrobnihy-razbor-klassa-hashmap>