Занятие 5

Anastasiya Solodkaya, Denis Stepulenok

LevelUP

21 октября 2016 г.

Дженерики

- Позволяют задать с помощью параметра типа тип используемых данных
- Являются исключительно compile-time фичей
- На самом деле "стираются"в коде

Дженерики - как задать

```
class MyArray<T> {
    ...
    public T get(int index) {
        ...
    }

    public T[] getAsArray() {
        ...
    }
    ...
}
```

Дженерики - как использовать

```
List<Integer> list = new ArrayList<Integer>();
List<Integer> list = new ArrayList<>();
Map<Integer, String> map
    = new HashMap<Integer, String>();
Map<Integer, String> map = new HashMap<>();
```

Зачем?

- Один и тот же тип может хранить разные данные. Например, один экземпляр хранит яблоки, а другой - гайки.
- Мы можем ограничить (и гарантировать себе), что в списке лежит определенный тип данных.

Дженерики - нюансы

Стирание типов

Стирание типов: для виртуальной машины нет разницы между **List<String>**, **List<Integer>** и **List**. Поэтому следующий код не работает

```
List<?> l;

...

if (l instanceof List<String>) {
    ...
}
```

Дженерики - нюансы

Стирание типов

Точно так же для нее нет разницы между **new Object[]** и **new T[]**, то есть следующий код не работает:

```
class MyClass<T> {

    public T[] array = new T[10];

}
```

Дженерики - больше о них

Дженерик методы

Здесь метод внутри класса объявляется с дженериком класса:

```
class MyClass<T> {
    public T get(int index) { ... }
}
```

А здесь у метода свои параметры типа:

```
class MyClass<T> {
    public<P> P update(P p, int index) { ... }
}
```

Дженерики - wildcards

- Unbounded wildcard - "что угодно"

```
List<?> list = new ArrayList<Integer>();
```

- Upper-bounded wildcards - "тип или его наследник"

```
List<? extends Number> list = new
    ArrayList<Integer>();
```

- Lower-bounded wildcards - "тип или его родитель"

```
List<? super String> list = new
    ArrayList<Object>();
```

Дженерики - ограничения в ваших классах

- Ограничение (класс или интерфейс)

```
class MyClass<T extends Number> { ... }
```

- Ограничение (несколько классов или интерфейсов)

```
class MyClass<T extends Number & Cloneable> {
    ... }
```

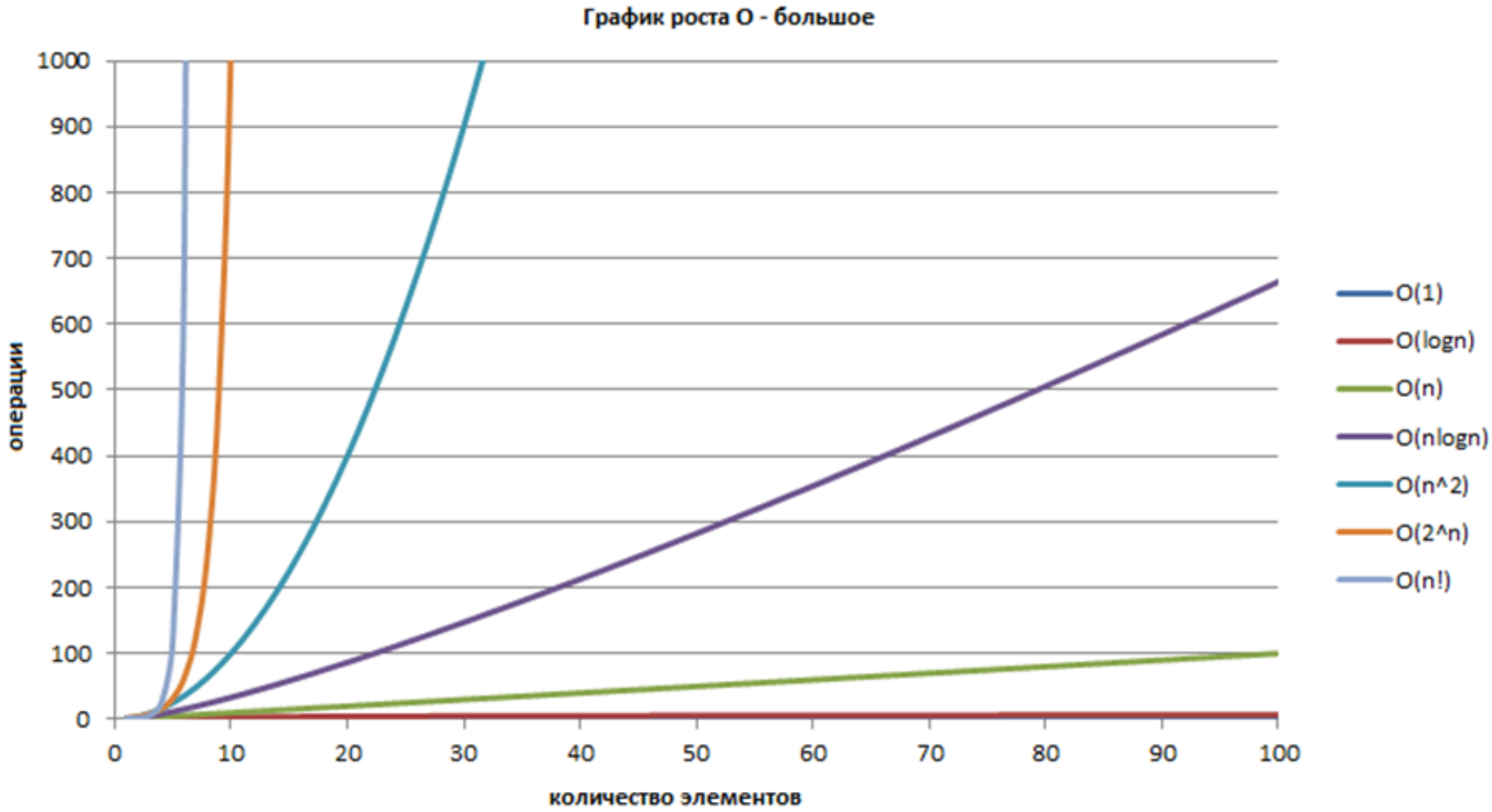
- Параметр передается в суперкласс

```
class MyClass<T> extends List<T> { ... }
```

- Параметр передается в суперкласс + ограничения

```
class MyClass<T extends Number > extends
    List<T> { ... }
```

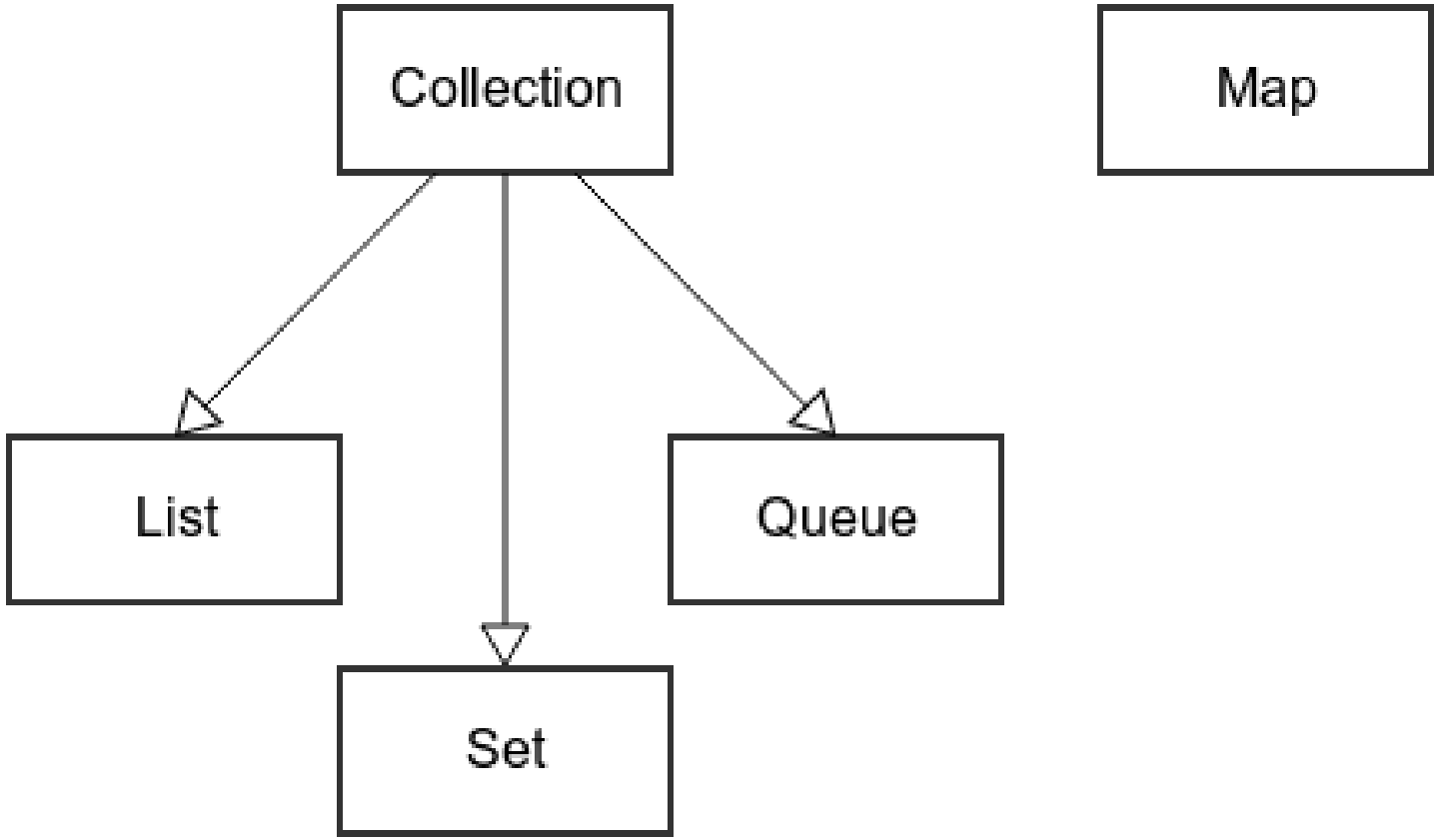
Асимптотическое поведение



Collection framework

- Коллекция – набор объектов, заключенных в одном объекте.
- java.util и java.util.concurrent

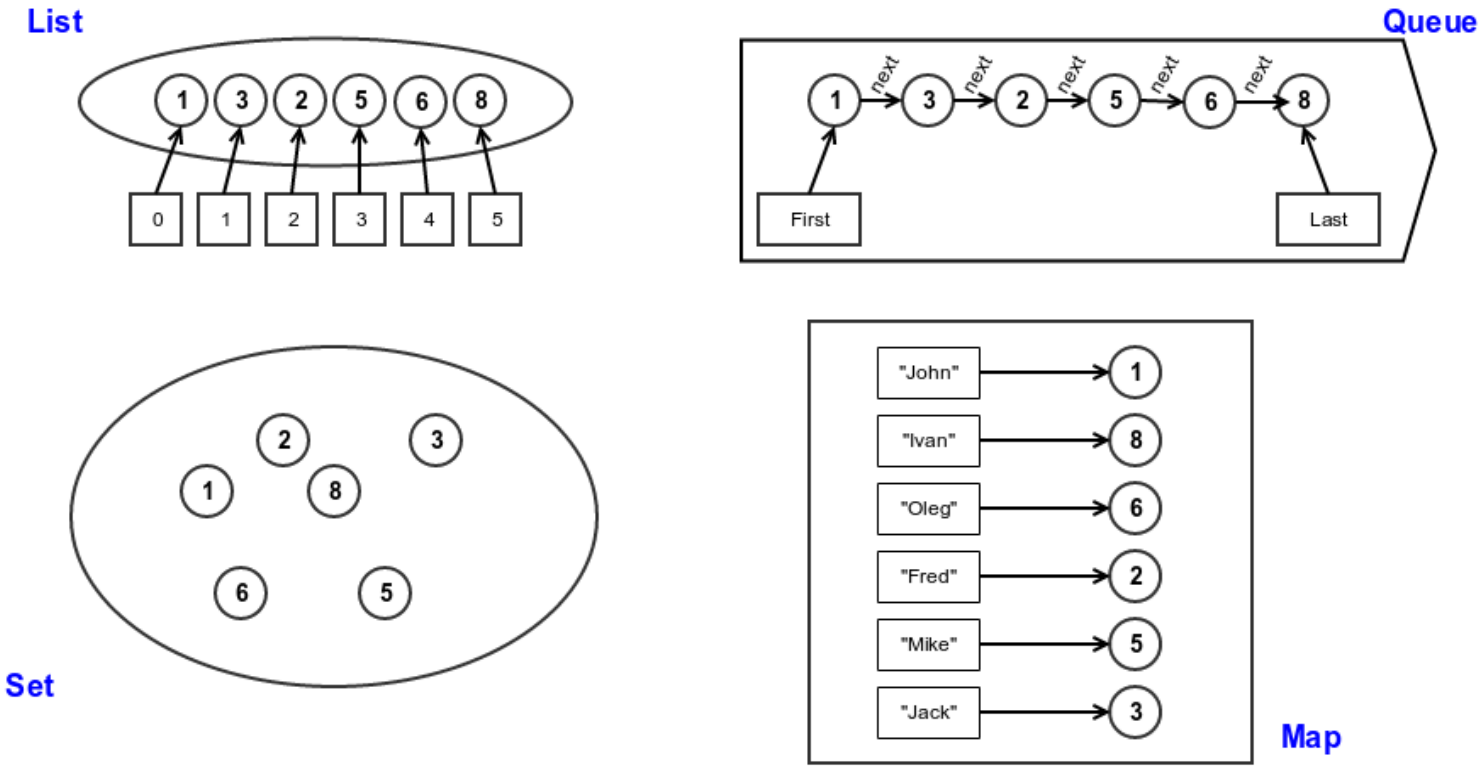
Основные интерфейсы



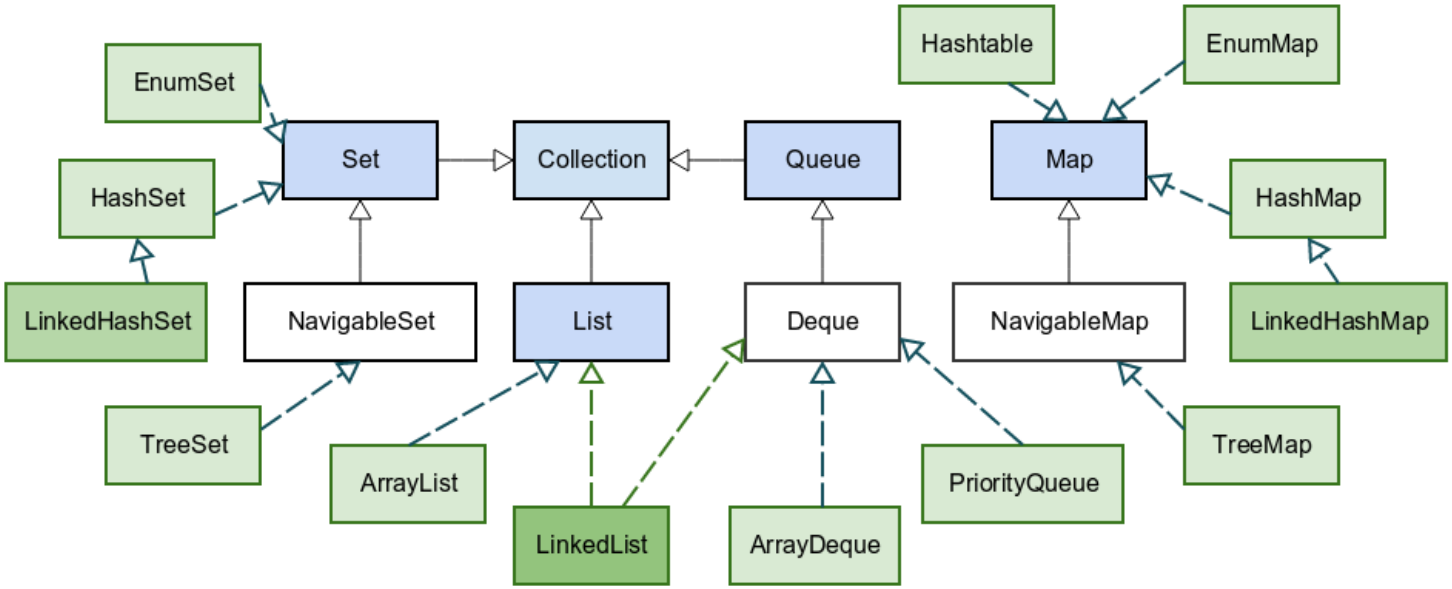
Основные интерфейсы

- List
  - упорядочен
  - доступ по индексу
  - разрешены дубликаты
- Set
  - просто набор элементов
  - дубликаты не разрешены
- Queue- упорядоченные элементы, как правило порядок либо FIFO, либо LIFO
- Map - набор пар ключ-значение

Основные структуры



Реализации


EnumSet
Hashtable
EnumMap
Set
Collection
Queue
Map
HashSet
HashMap
LinkedHashSet
NavigableSet
List
Deque
NavigableMap
LinkedHashMap
TreeSet
ArrayList
LinkedList
ArrayDeque
PriorityQueue
TreeMap

Использование коллекций

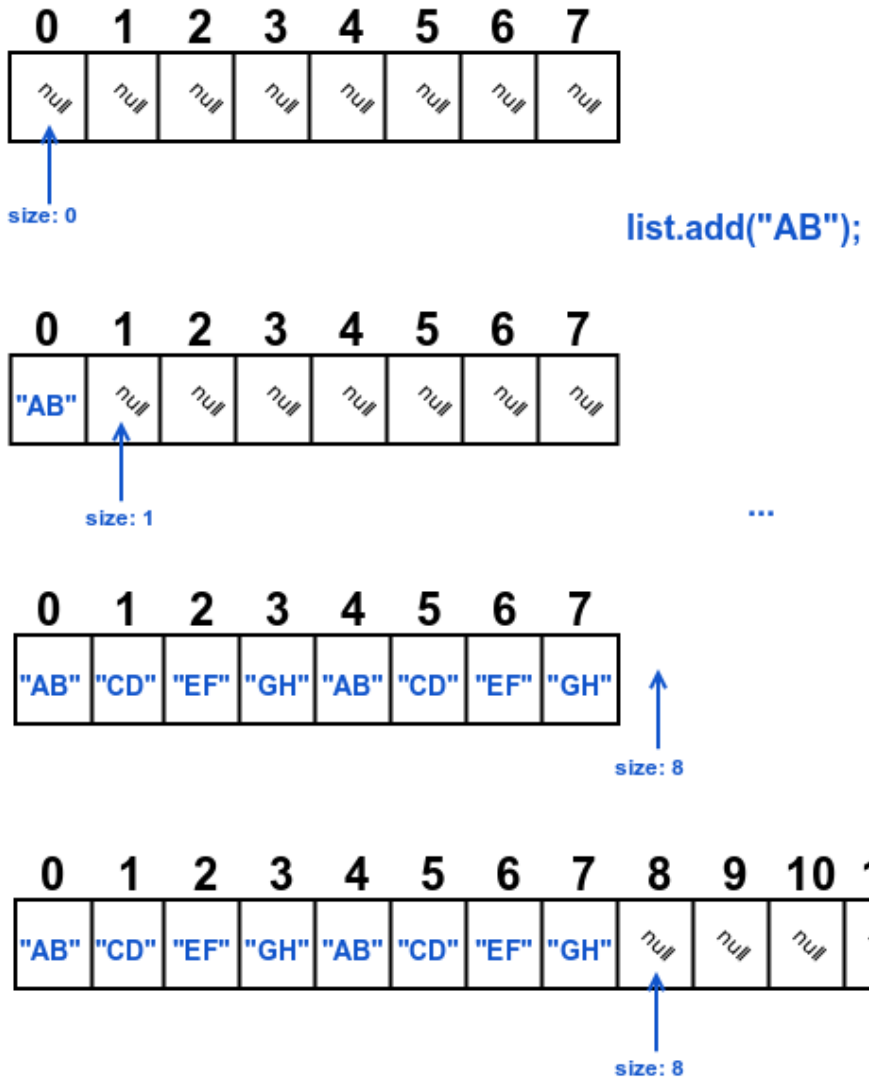
Очень часто (если не требуются какие-то частные методы) переменная объявляется как какой-то базовый тип:

```
List<String> list = new ArrayList<String>();
```

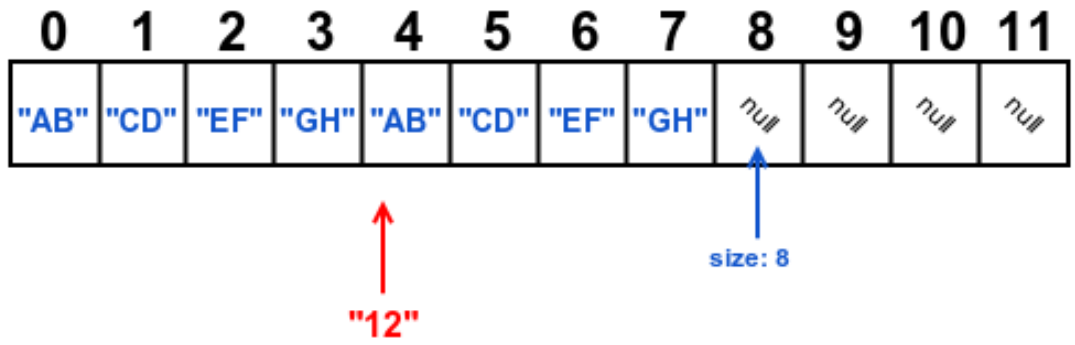
ArrayList

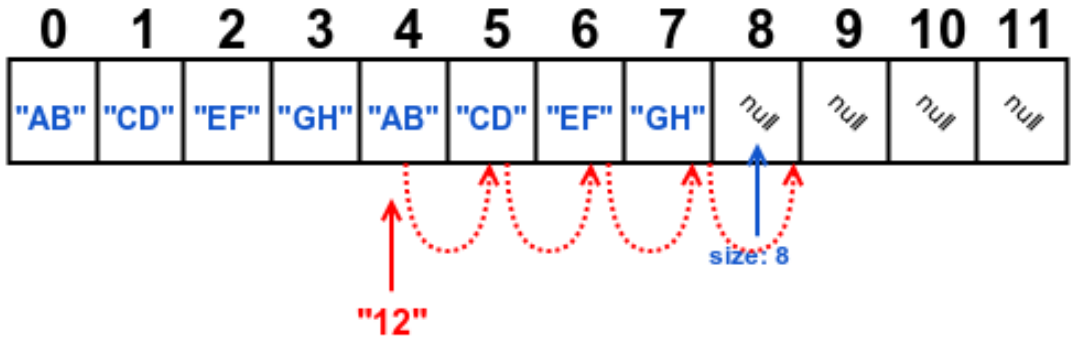
- Реализует интерфейс **List**
- Позволяет хранить *null*
- Быстрый доступ по индексу
- Медленные операции - добавление в середину, поиск, удаление из середины
- Добавление в конец иногда очень медленное

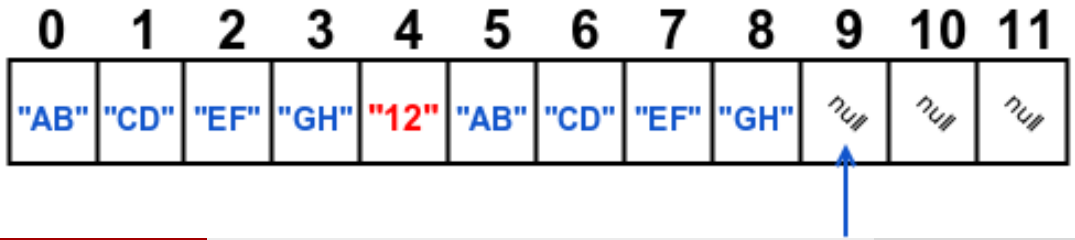
ArrayList


0 1 2 3 4 5 6 7
null null null null null null null null
size: 0
list.add("AB");
0 1 2 3 4 5 6 7
"AB" null null null null null null null
size: 1
...
0 1 2 3 4 5 6 7
"AB" "CD" "EF" "GH" "AB" "CD" "EF" "GH"
size: 8
0 1 2 3 4 5 6 7 8 9 10 11
"AB" "CD" "EF" "GH" "AB" "CD" "EF" "GH" null null null null
size: 8

ArrayList


0 1 2 3 4 5 6 7 8 9 10 11
"AB" "CD" "EF" "GH" "AB" "CD" "EF" "GH" null null null null
size: 8
"12"

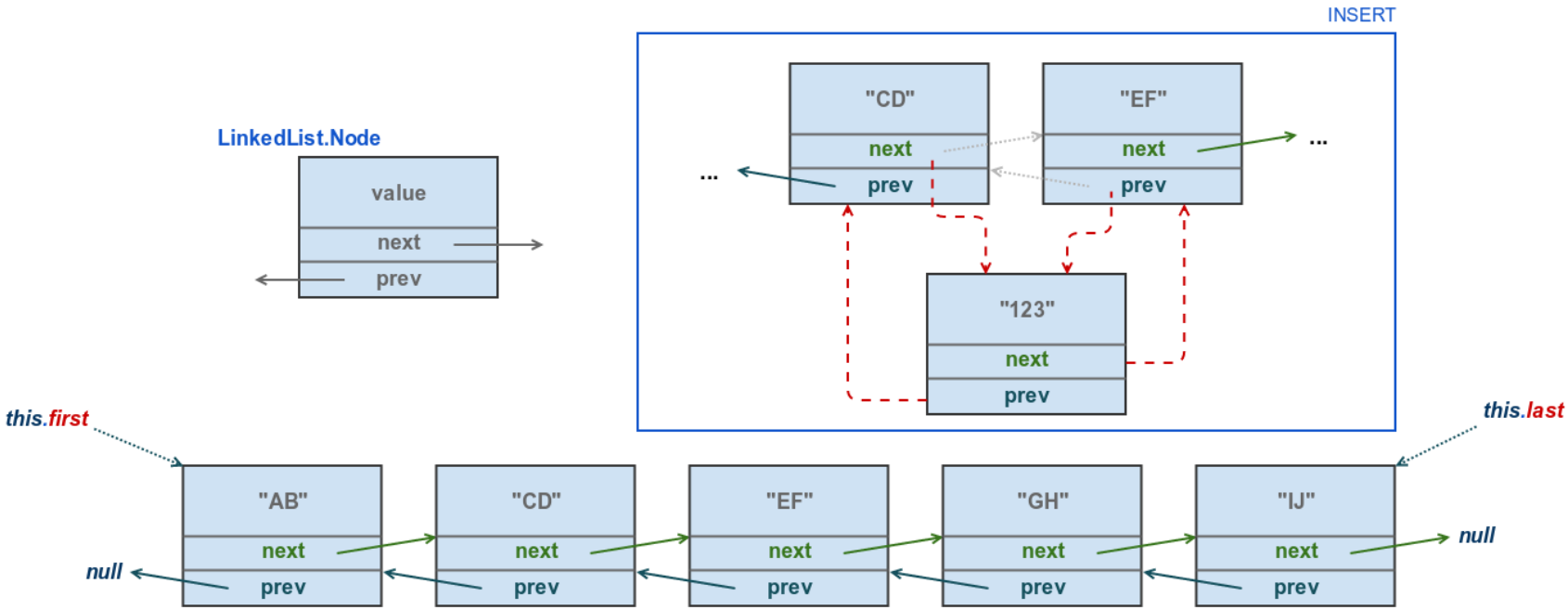

0 1 2 3 4 5 6 7 8 9 10 11
"AB" "CD" "EF" "GH" "AB" "CD" "EF" "GH" null null null null
size: 8
"12"


0 1 2 3 4 5 6 7 8 9 10 11
"AB" "CD" "EF" "GH" "12" "AB" "CD" "EF" "GH" null null null

LinkedList

- Реализует интерфейсы **List**, **Queue** и **Deque**
- Позволяет хранить *null*
- Медленный доступ по индексу и поиск
- Быстрое добавление/удаление из середины при условии, что элемент уже найден
- Добавление и удаление из конца и начала - быстрое
- Стек, очередь, двойная очередь
- Требует немало дополнительной памяти

LinkedList


INSERT
LinkedList.Node
value
next
prev
"CD"
"EF"
"123"
...
this.first
this.last
"AB"
"CD"
"EF"
"GH"
"IJ"
null
null

HashMap

- Реализует интерфейс **Map**
- Позволяет хранить *null*
- Быстрое взятие элемента по ключу (при равномерном распределении hash-функции), но в худшем случае до $O(n)$
- Быстрое добавление элемента
- Требует наличие *hashCode* и *equals* у ключа
- Наподобии *ArrayList* расширяется и перераспределяет данные при максисмальном заполнении массива (см. $threshold = (capacity * loadFactor)$)

HashSet

- Реализует интерфейс **Set**
- Позволяет хранить *null*
- Отсутствует сортировка
- Самый быстрый - все базовые операции $O(1)$
- Сделан на основе *HashMap*

TreeSet

- Реализует интерфейсы **Set**, **SortedSet** и **NavigableSet**
- Запрещены *null*
- Есть сортировка
- Медленнее, чем HashSet - все базовые операции $O(logn)$
- Сделан на основе *TreeMap* (красно-черное дерево)

Коллекции, которые не входят в Collection Framework

Достаточно медленные

- Vector
- Stack
- Hashtable

Все зависит от вашей задачи

- Какие требования к скорости работы?
- Какие требования к расходу памяти?
- Какую задачу вы решаете?

Список, карта или множество?

- Необходим ли вам доступ к элементу по индексу?
- Уникальны ли элементы в коллекции?
- Хотите ли вы быстро получать элемент по какому-то свойству (или ключу)?
- Нужен ли какой-то порядок элементов (по добавлению, по алфавиту, числовой, и т.д.)
- Хотите ли вы хранить соответствие между какими то объектами?

Список, карта или множество - примеры

Примеры:

- Список покупок?
- Хранилище свойств файловой системы по названию?
- Список учеников класса?

LinkedList или ArrayList?

- В 90% случаев используют **ArrayList**.
- Если вам нужно часто добавлять или удалять элементы из начала или конца, то **LinkedList**
- Если нужен быстрый доступ по индексу, то нужен **ArrayList**
- Оба они обеспечивают поиск элемента за время, пропорциональное их длине

LinkedList или ArrayList - примеры

- Очередь из документов на печать (добавляем в конец, удаляем из начала)
- Список учеников класса, который необходимо демонстрировать на экране и редактировать
- Список товаров в корзине пользователя интернет-магазина
- Реализация карточной игры "дурак"(колода)

HashSet или TreeSet?

Необходимо в первую очередь помнить, что:

- **HashSet** быстрее, чем **TreeSet**
- **HashSet** не поддерживает сортировку вообще, а **TreeSet** поддерживает натуральный порядок.

HashSet или TreeSet - примеры

- Вы пишете приложение, в котором вы случайным образом выбираете из списка элементов один элемент и переносите его в множество. В конце программы вам надо проверить, что определенный элемент попал в множество.
- Вы выбираете последовательность из уникальных случайных чисел, а после передаете ее в другую часть программы, чтобы там использовать их в порядке добавления.

HashMap

- Хранение свойств файлов
- Хранение информации о том, какому сотруднику какой компьютер принадлежит
- Хранение диалогов пользователей в социальной сети
- Хранение информации о загруженных из www страниц (для того, чтобы не загружать информацию заново)

Список, аналогичный ArrayList

Необходимо создать список, аналогичный **ArrayList**. Добавьте базовые методы.
Надо добавить возможность задавать изначальную величину массива и, при нехватке места, увеличивать размер массива в 1.5 раза.
Этот список вы будете использовать в своем проекте для хранения данных. При необходимости, можно (и нужно) добавлять новые методы.