

Programmeertechnieken/Programming Techniques

*Part 2*

*Data Structures*

Koen Pelsmaekers

Campus Groep T, 2022-2023

# Data Structures: introduction

# Data Structures

- Collection of data
- Example of ADT (“Abstract Data Type”)
  - Specification (= the interface) vs. Implementation
- Algorithms
  - Efficiency depends on kind of operations
  - Traversal, deletion, insertion, ...
  - ... and the implementation
    - array-based, tree-based, hashtable-based (is also array-based), ...
  - Big O-notation: indicates the order of magnitude of an algorithm

# Big-O notation



- $O(1)$ : runtime is constant, independent from the number of elements  $n$
- $O(\log n)$ : runtime grows logarithmically in proportion to the number of elements  $n$
- $O(n)$ : runtime grows directly in proportion to the number of elements  $n$  (linear)
- ...
- $O(n!)$ : runtime grows very fast and becomes big for even a small number of elements  $n$

# Data Structures in Java

# “Program to an interface”

- Excellent design principle:
  - Decouple declaration from implementation
    - “what” vs. “how”, “specification” vs. “implementation”
  - Information hiding or Encapsulation
    - Do not expose the internals of your implementation (so that you can change it if necessary, without any external visible change)
  - Defer choice of actual class
    - “I need something that implements interface x, that behaves like x”

More details: part 5 of this course

# Java Collections Framework (from javadoc)

*The collections framework is a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation.*

# Java Collections Framework: Advantages

- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

Only 25 classes and interfaces (1998): “Our main design goal was to produce an API that was reasonably small, both in size, and (more importantly) in ‘**conceptual weight**’.”



# Collections

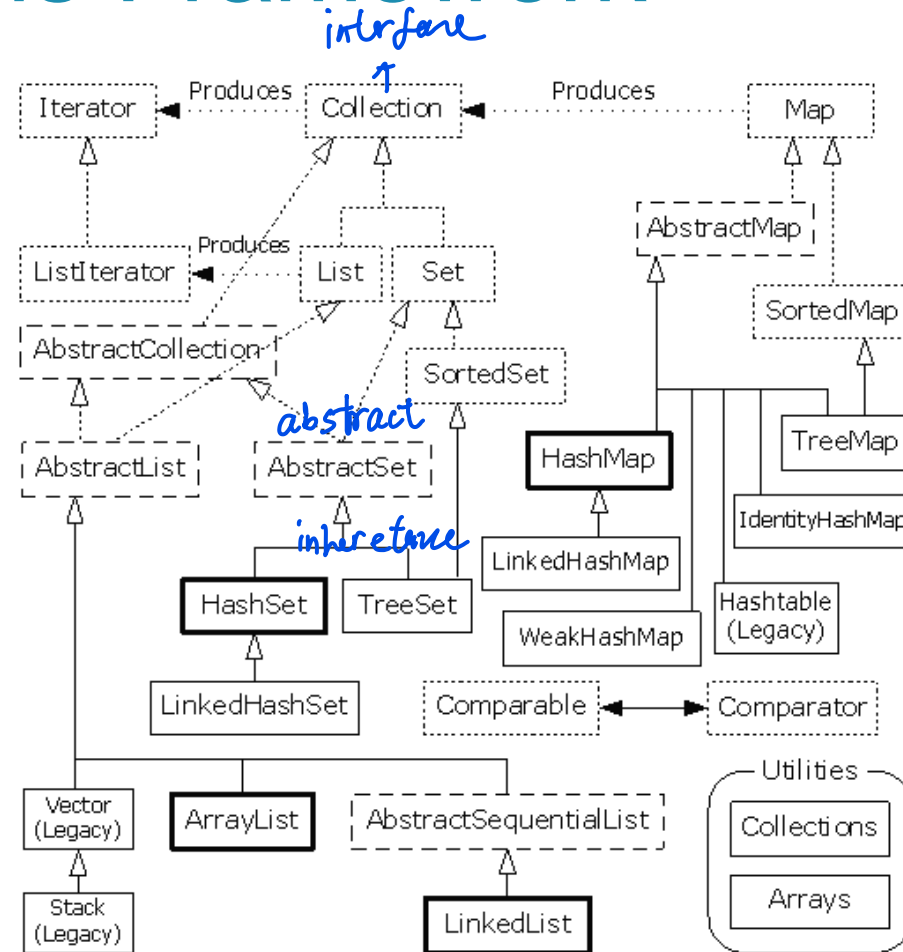
- Data structures
  - Interfaces
  - Implementations (general-purpose/specialized)
  - Algorithms
- ... see: “The Java Tutorials, Trail: Collections”, module: base, package: java.util

General purpose implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<a href="#">HashSet</a>		<a href="#">TreeSet</a>		<a href="#">LinkedHashSet</a>
List		<a href="#">ArrayList</a>		<a href="#">LinkedList</a>	
Deque		<a href="#">ArrayDeque</a>		<a href="#">LinkedList</a>	
Map	<a href="#">HashMap</a>		<a href="#">TreeMap</a>		<a href="#">LinkedHashMap</a>

Javadoc 17 (Java releases: <https://www.java.com/releases/>; Java 20: 21/03/2023; <https://dev.java/>); source code: <https://jdk.java.net/>

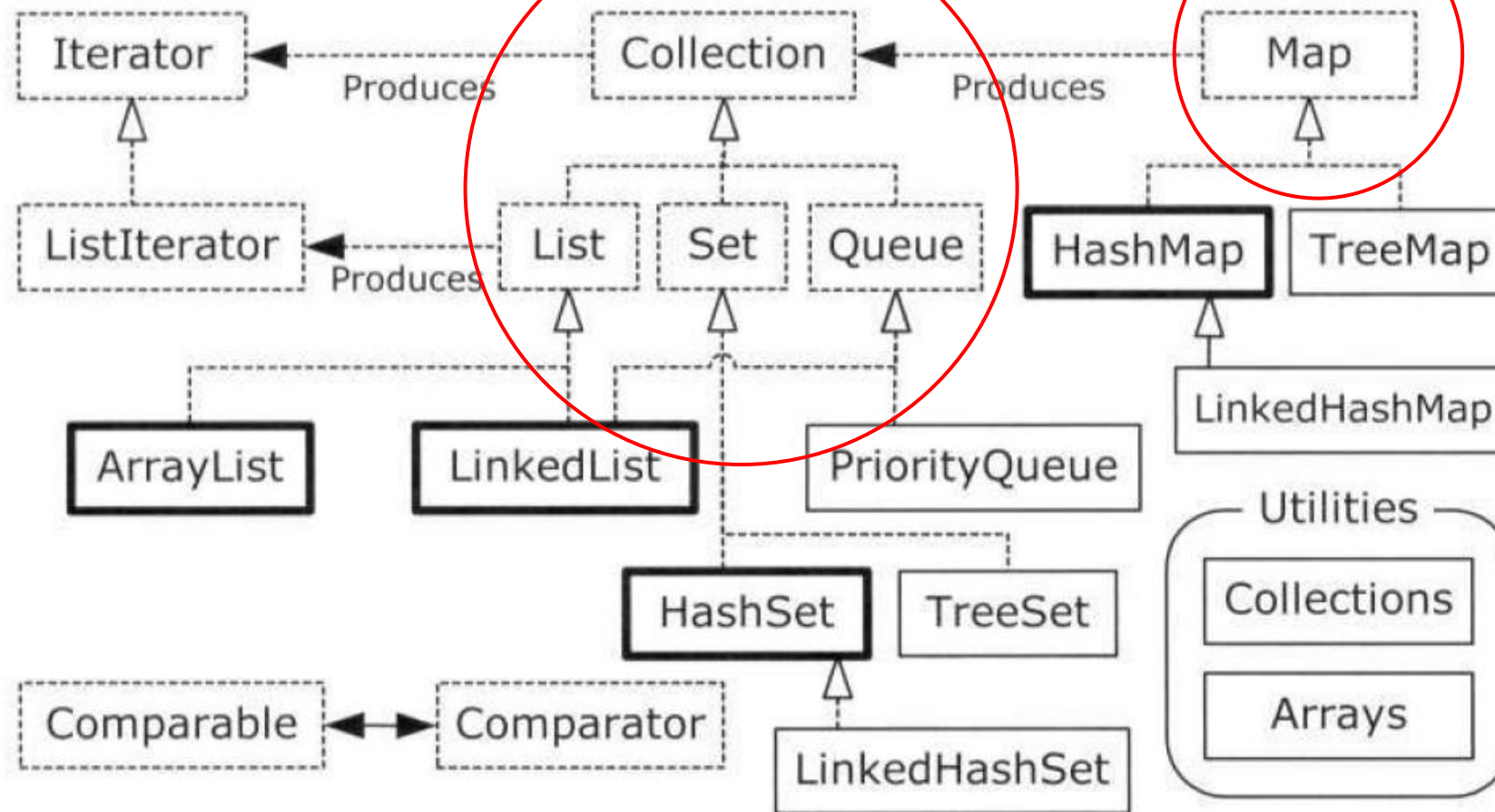
# Java Collections Framework



Source: Thinking in Java, Fourth Edition, Bruce Eckel  
No standard UML!

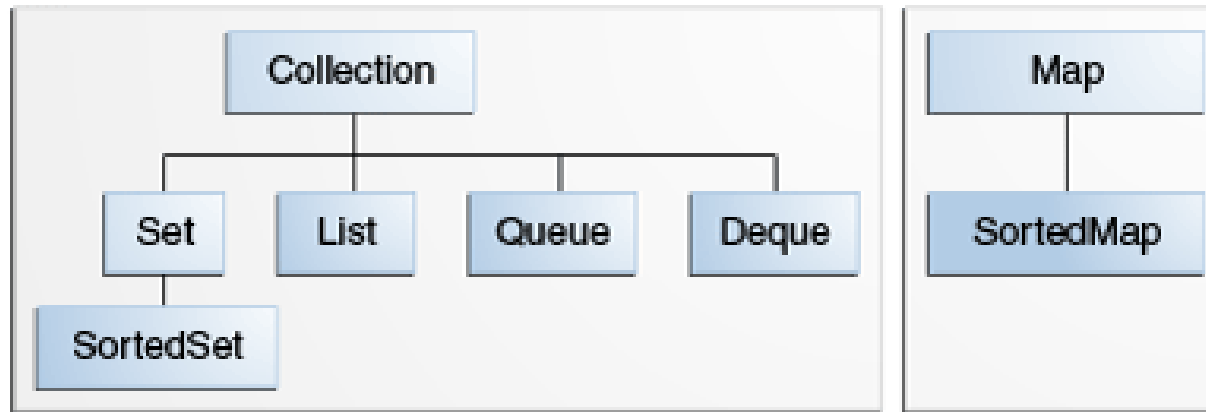
# Java Collections Framework

*abstract omitted*



Source: Thinking in Java, Fourth Edition, Bruce Eckel  
No standard UML, Simple Container Taxonomy

# Java Collections Framework



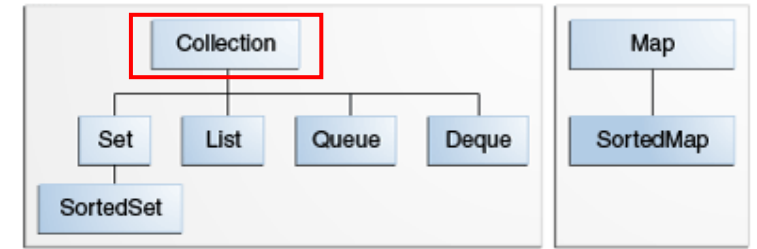
- Interfaces, implementations & algorithms
- No fixed size
- Objects → <generics>

No standard UML!

*in arraylist  
only pointers  
so need objects*

# Collection interface

“*bag*”



## From the API:

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

# Some Collection interface methods

<code>add(o)</code>	Add a new element o
<code>size()</code>	Returns the number of elements
<code>contains(o)</code>	Membership checking for o ( <code>equals()</code> )
<code>clear()</code>	Remove all elements
<code>remove(o)</code>	Remove the element o
<code>isEmpty()</code>	Whether the collection is empty
<code>iterator()</code>	Returns an iterator

No “exclusive” implementation(s).

# Collection interface implementations

No “exclusive” implementation

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#),  
[ArrayBlockingQueue](#), [ArrayDeque](#), [ArrayList](#), [AttributeList](#), [BeanContextServicesSupport](#),  
[BeanContextSupport](#), [ConcurrentHashMap.KeySetView](#), [ConcurrentLinkedDeque](#),  
[ConcurrentLinkedQueue](#), [ConcurrentSkipListSet](#), [CopyOnWriteArrayList](#),  
[CopyOnWriteArraySet](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#),  
[LinkedBlockingDeque](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#),  
[LinkedTransferQueue](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#),  
[Stack](#), [SynchronousQueue](#), [TreeSet](#), [Vector](#)

# Collection interface implementations

No “exclusive” implementation

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet,  
ArrayBlockingQueue, **ArrayDeque**, **ArrayList**, AttributeList, BeanContextServicesSupport,  
BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque,  
ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList,  
CopyOnWriteArraySet, DelayQueue, **EnumSet**, **HashSet**, JobStateReasons,  
LinkedBlockingDeque, LinkedBlockingQueue, **LinkedHashSet**, **LinkedList**,  
LinkedTransferQueue, PriorityBlockingQueue, **PriorityQueue**, RoleList, RoleUnresolvedList,  
Stack, SynchronousQueue, **TreeSet**, Vector



# Other Collection methods

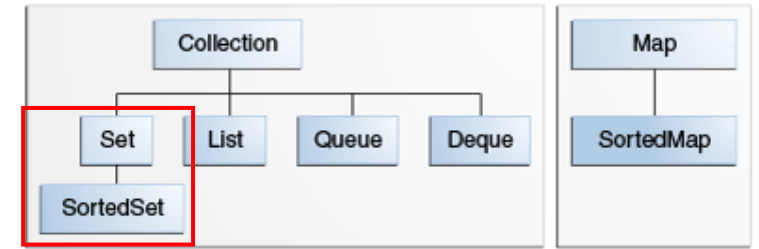
- Set theory operations
  - Intersection ( $a \cap b$ )
    - `a.retainAll(b)`
  - Union ( $a \cup b$ )
    - `a.addAll(b)`
  - Complement ( $a \setminus b$ )
    - `a.removeAll(b)`
- Other methods
  - `containsAll(Collection)`
  - `removeIf(Predicate)`
  - `stream()`
  - `parallelStream()`
  - `toArray()`

like  
shared  
document

Available for all implementations of sub-interfaces too.

# Set - SortedSet interface

*“bag with unique elements”*



**From the API:**

## Set

A collection that contains no duplicate elements. More formally, sets contain no pair of elements  $e_1$  and  $e_2$  such that  $e_1.equals(e_2)$ , and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

*no same thing*

## SortedSet

A Set that further provides a total ordering on its elements. The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of SortedMap).

# Some Set/SortedSet interface methods

<code>add(o)</code>	Add a new element o
<code>size()</code>	Returns the number of elements
<code>contains(o)</code>	Membership checking for o ( <code>equals()</code> )
<code>clear()</code>	Remove all elements
<code>remove(o)</code>	Remove the element o
<code>isEmpty()</code>	Whether the collection is empty
<code>iterator()</code>	Returns an iterator

# Set/SorteSet interface implementations

## Set

All Known Implementing Classes:

AbstractSet, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet,  
EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

## SortedSet

All Known Implementing Classes:

ConcurrentSkipListSet, TreeSet

# Set/SorteSet interface implementations

## Set

All Known Implementing Classes:

AbstractSet, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, **HashSet**, JobStateReasons, **LinkedHashSet**, **TreeSet**

## SortedSet

All Known Implementing Classes:

ConcurrentSkipListSet, **TreeSet**

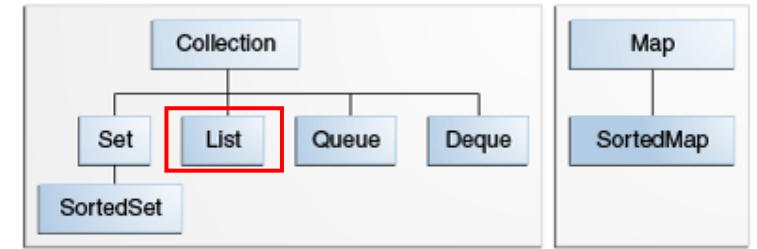
General purpose implementations					
Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<a href="#">HashSet</a>		<a href="#">TreeSet</a>		<a href="#">LinkedHashSet</a>
List		<a href="#">ArrayList</a>		<a href="#">LinkedList</a>	
Deque		<a href="#">ArrayDeque</a>		<a href="#">LinkedList</a>	
Map	<a href="#">HashMap</a>		<a href="#">TreeMap</a>		<a href="#">LinkedHashMap</a>

# Set interface examples

```
Set<Person> personSet = new HashSet<>();  
Person person = new Person("Jeff Nobody", 85.0);  
personSet.add(person);
```

```
int n = personSet.size();  
if (personSet.contains(person)){  
    // check membership based on equals()  
}  
Iterator<Person> it = personSet.iterator();  
while (it.hasNext()) {  
    Person p = it.next();  
    // without downcast ((Person)it.next())  
}
```

# List interface



## From the API:

An **ordered collection** (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer **index** (position in the list), and search for elements in the list.

Unlike sets, lists typically allow **duplicate elements**. More formally, lists typically allow pairs of elements  $e_1$  and  $e_2$  such that  $e_1.equals(e_2)$ , and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

# Some List interface methods

<code>add(i, o)</code>	Insert o at position i
<code>add(o)</code>	Append o to end
<code>get(i)</code>	Return the i-th element
<code>remove(i)</code>	Remove the i-th element
<code>remove(o)</code>	Remove the element o
<code>set(i,o)</code>	Replace the i-th element with o
<code>iterator()</code>	Returns an iterator

(Only the most important methods are shown [int i, Object o])



# List interface implementations

All Known Implementing Classes:

[AbstractList](#), [AbstractSequentialList](#), [ArrayList](#), [AttributeList](#),  
[CopyOnWriteArrayList](#), [LinkedList](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [Vector](#)

# List interface implementations

All Known Implementing Classes:

[AbstractList](#), [AbstractSequentialList](#), [ArrayList](#), [AttributeList](#),  
[CopyOnWriteArrayList](#), [LinkedList](#), [RoleList](#), [RoleUnresolvedList](#), [Stack](#), [Vector](#)

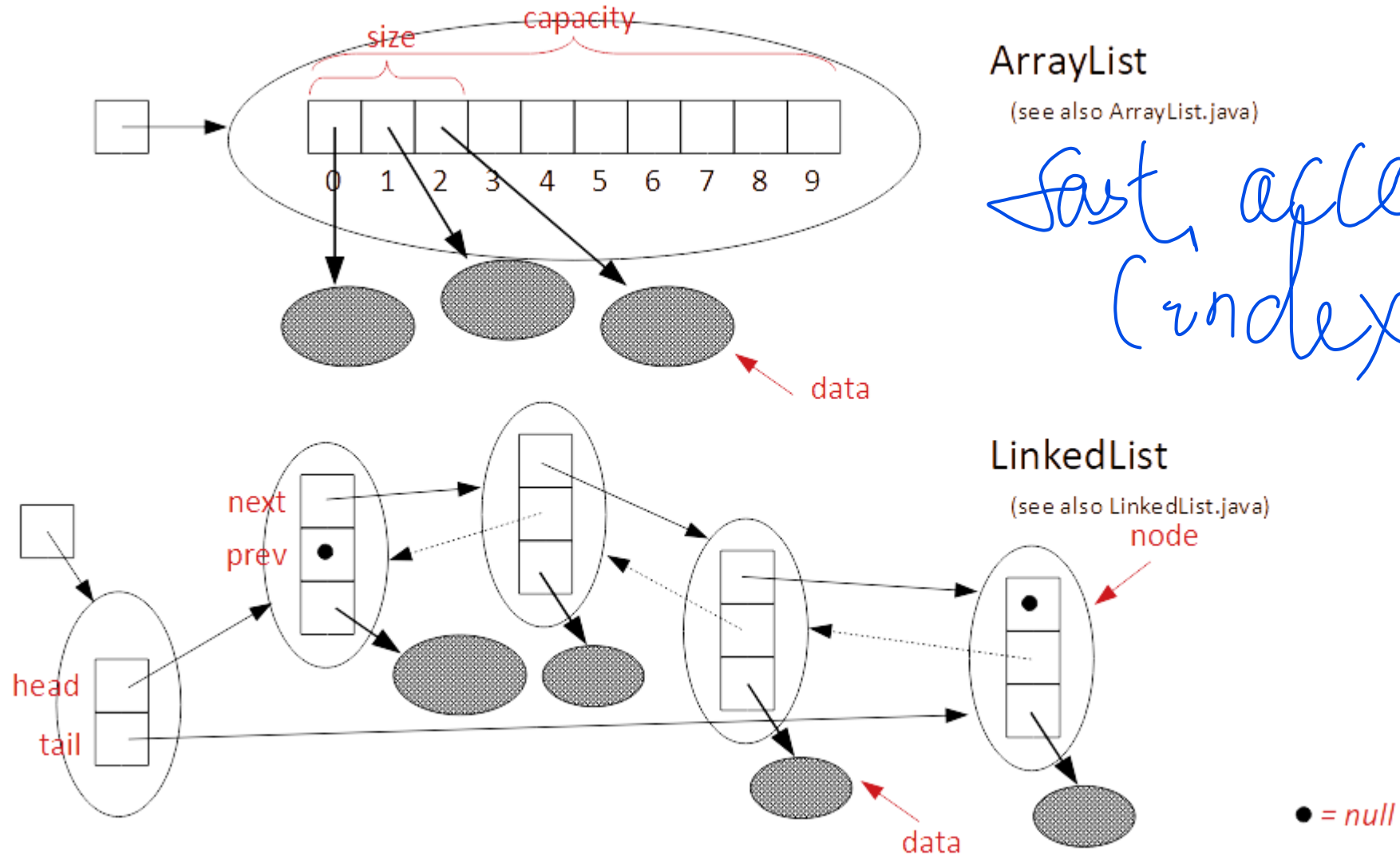
General purpose implementations					
Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<u><a href="#">HashSet</a></u>		<u><a href="#">TreeSet</a></u>		<u><a href="#">LinkedHashSet</a></u>
List		<u><a href="#">ArrayList</a></u>		<u><a href="#">LinkedList</a></u>	
Deque		<u><a href="#">ArrayDeque</a></u>		<u><a href="#">LinkedList</a></u>	
Map	<u><a href="#">HashMap</a></u>		<u><a href="#">TreeMap</a></u>		<u><a href="#">LinkedHashMap</a></u>

# List interface examples

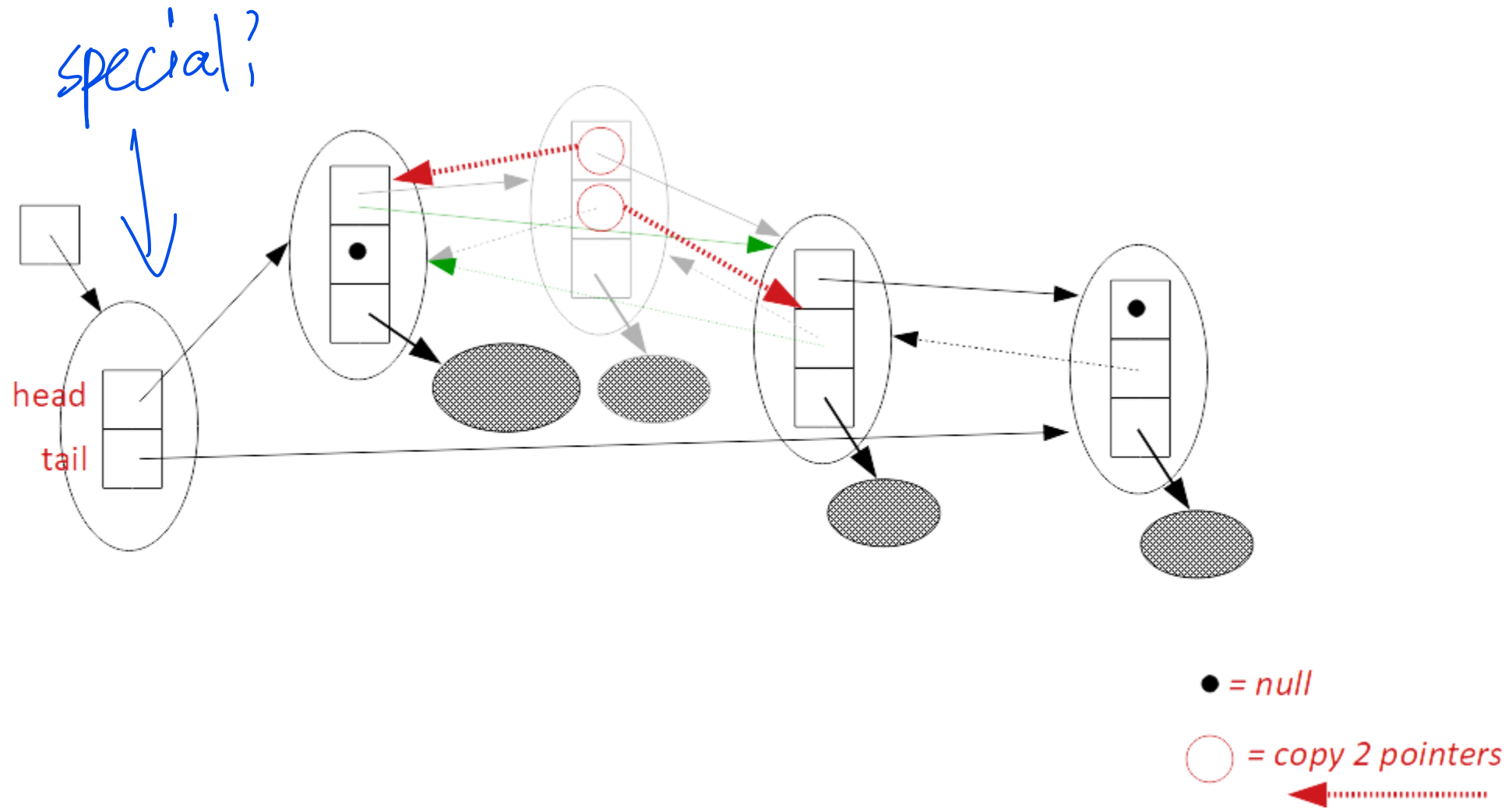
```
List<Person> personList = new ArrayList<>();
Person person = new Person("Jeff Nobody", 85.0);
personList.add(person);

int n = personList.size();
if (personList.contains(person)){
    // check membership based on equals()
}
Iterator<Person> it = personList.iterator();
while (it.hasNext()) {
    Person p = it.next();
    // without downcast ((Person)it.next())
}
```

# ArrayList vs. LinkedList



# LinkedList: remove an element



# ArrayList vs. LinkedList

- ArrayList *always use AL if you do not know what to use ← easily changed to others type*
  - + Fast get operation
  - + Fast add/remove at the back
  - Slow insert/remove operation at the front
  - Contiguous memory (re)allocation needed
- LinkedList
  - Slow get operation
  - + Fast add/remove operation at the front/back
  - + No memory reallocation needed

	<i>get</i>	<i>add</i>	<i>contains</i>	<i>next</i>	<i>remove(O)</i>	<i>Iterator.remove</i>
<i>ArrayList</i>	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
<i>LinkedList</i>	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
<i>CopyOnWriteArrayList</i>	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

# Some statistics: List implementations (+ demo)

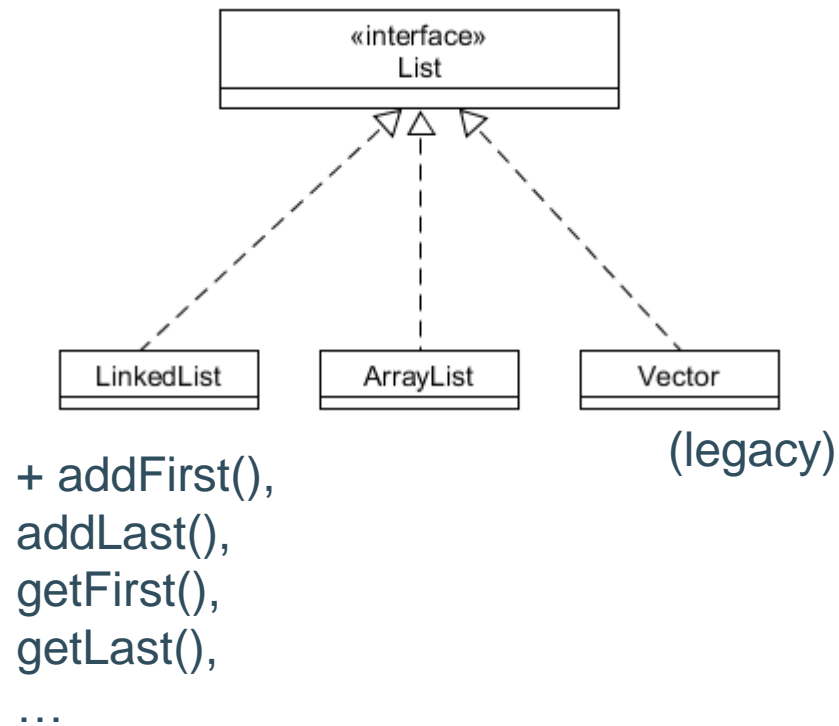
	add operation (back)	get operation	remove operation (front)	remove operation (2nd element)	remove operation (middle)	remove operation (last)	remove operation (one but last)	remove operation (100 but last)	insert operation (front)
test with ArrayList WITHOUT initial capacity	49	1	46	28	6	0	0	1	34
test with ArrayList WITH initial capacity	21	0	33	26	5	2	0	0	29
test with LinkedList	31	547	0	1	151	2	0	0	2

# Other List methods

- Sort all elements (since Java 8, “default implementation”)
  - “In an ordered collection elements can be sorted.”
    - `myList.sort(Comparator)`
    - `myList.sort(null)` (Comparable interface, natural or internal ordering)



# Program towards the interface



```
Person p = new Person("Jeff", 85.6);
List<Person> myList1 = new ArrayList<>();
List<Person> myList2 = new LinkedList<>();
myList1.addFirst(p); // valid?
myList2.addFirst(p); // valid?
```

*both not*

# Compile-time type vs. run-time type

```
Person p = new Person("Jeff", 85.6);
```

```
List<Person> myList1 = new ArrayList<>();
```

```
List<Person> myList2 = new ArrayList<>();
```

```
myList1.addFirst(p); // valid?
```

```
// compiler error: Cannot resolve method 'addFirst' in 'List'
```

```
Person p2 = myList2.getLast(p); // valid?
```

```
// compiler error: Cannot resolve method 'getLast' in 'List'
```

```
myList1 = new LinkedList<>(); // valid?
```

```
myList1.addFirst(p); // valid?
```

① type cast

```
((LinkedList<Person>) myList1).addFirst(p); // valid?
```

```
Person p3 = ((LinkedList<Person>) myList2).getLast(); // valid?
```

the static type

dynamic binding

change the static type

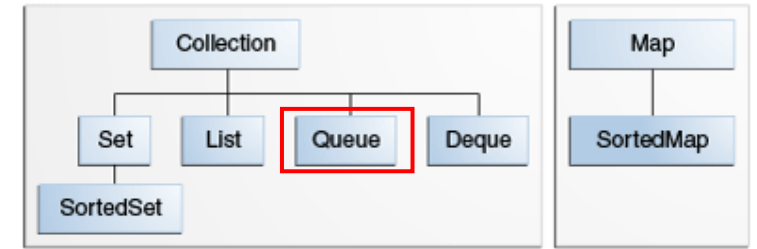
Do the test!

# Queue interface

## From the API:

A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.



# Some Queue interface methods

add(o)	Insert o (throws exception)
offer(o)	Insert o (returns special value: boolean)
remove()	Remove and return the head of the queue (throws exception)
poll()	Remove and return the head of the queue (returns special value: null)
element()	Return the head of the queue (throws exception)
peek()	Return the head of the queue (returns special value: null)

(in case of empty queue or queue with no capacity)

(Only the most important methods are shown.)

# Queue interface implementations

All Known Implementing Classes:

[AbstractQueue](#), [ArrayBlockingQueue](#), [ArrayDeque](#), [ConcurrentLinkedDeque](#), [ConcurrentLinkedQueue](#),  
[DelayQueue](#), [LinkedBlockingDeque](#), [LinkedBlockingQueue](#), [LinkedList](#), [LinkedTransferQueue](#),  
[PriorityBlockingQueue](#), [PriorityQueue](#), [SynchronousQueue](#)

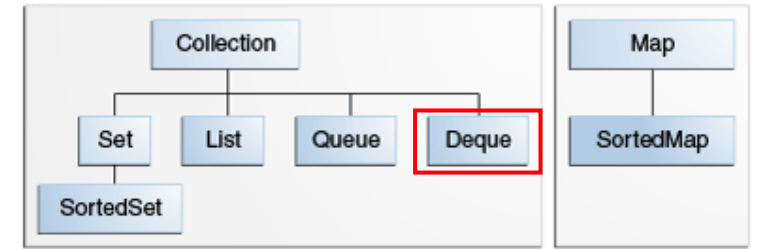
# Queue interface examples

```
Queue<Person> personQueue = new ArrayDeque<>();
Person person1 = new Person("Jeff Nobody", 85.0);
Person person2 = new Person("Julia Who", 60.5);
personQueue.add(person1);
personQueue.offer(person2);
System.out.println(personQueue.peek()); //Person{name='Jeff Nobody', weight=85.0}

Person person3 = personQueue.remove();
System.out.println(personQueue.element()); //Person{name='Julia Who', weight=60.5}

person3 = personQueue.remove();
person3 = personQueue.poll(); // returns null
person3 = personQueue.remove(); // NoSuchElementException
```

# Deque interface



## From the API:

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Deque implementations; in most implementations, insert operations cannot fail.

# Some Deque interface methods

The twelve methods described above are summarized in the following table:

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	<i>Throws exception</i>	<i>Special value</i>	<i>Throws exception</i>	<i>Special value</i>
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Comparison of Queue and Deque methods

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

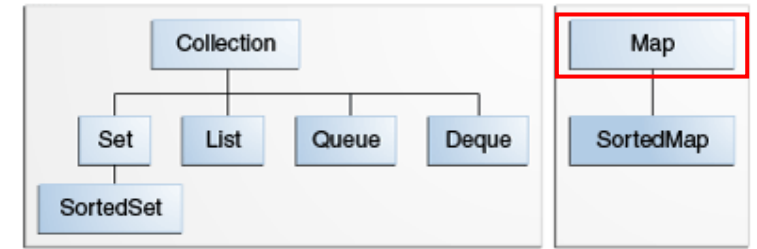
(Only the most important methods are shown.)



# Map interface

From the API:

*unique*




An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

The Map interface provides three *collection views*, which allow a map's contents to be viewed as a **set of keys**, **collection of values**, or **set of key-value mappings**. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

# Map

- Key-Value pair data structure
- “Associative array”
- Python: “Dictionary”

# “Maps” are everywhere

- Dictionary
  - word -> explanation
- Contact list on mobile device
  - (nick)name -> extra information about a contact
- HTTP-headers/Request parameters
  - Host -> [www.kuleuven.be](http://www.kuleuven.be)
  - Accept -> text/html
  - Accept-language -> en-US
  - ...
  - `<url>?source=xy&q=Berlin&output=json...`

# Some Map interface methods

<code>put(k,v)</code>	Put mapping for k to v
<code>get(k)</code>	Get the value associated with k
<code>keySet()</code>	The set of keys
<code>values()</code>	The collection of values
<code>entrySet()</code>	The set of key-value pairs
<code>containsKey(k)</code>	Whether contains a mapping for k
<code>containsValue(v)</code>	Whether contains a mapping to v
<code>putIfAbsent(k,v)</code>	Put mapping for k to v if none exists
<code>getOrDefault(k, v)</code>	Get the value associated with k, v if no mapping exists

(Only the most important methods are shown [k=key, v=value])

# Map interface implementations

All Known Implementing Classes:

[AbstractMap](#), [Attributes](#), [AuthProvider](#), [ConcurrentHashMap](#), [ConcurrentSkipListMap](#), [EnumMap](#),  
[HashMap](#), [Hashtable](#), [Headers](#), [IdentityHashMap](#), [LinkedHashMap](#), [PrinterStateReasons](#),  
[Properties](#), [Provider](#), [RenderingHints](#), [ScriptObjectMirror](#), [SimpleBindings](#), [TabularDataSupport](#),  
[TreeMap](#), [UIDefaults](#), [WeakHashMap](#)

# Map interface implementations

All Known Implementing Classes:

[AbstractMap](#), [Attributes](#), [AuthProvider](#), [ConcurrentHashMap](#), [ConcurrentSkipListMap](#), [EnumMap](#), [HashMap](#), [Hashtable](#), [Headers](#), [IdentityHashMap](#), [LinkedHashMap](#), [PrinterStateReasons](#), [Properties](#), [Provider](#), [RenderingHints](#), [ScriptObjectMirror](#), [SimpleBindings](#), [TabularDataSupport](#), [TreeMap](#), [UIDefaults](#), [WeakHashMap](#)

General purpose implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<a href="#"><u>HashSet</u></a>		<a href="#"><u>TreeSet</u></a>		<a href="#"><u>LinkedHashSet</u></a>
List		<a href="#"><u>ArrayList</u></a>		<a href="#"><u>LinkedList</u></a>	
Deque		<a href="#"><u>ArrayDeque</u></a>		<a href="#"><u>LinkedList</u></a>	
Map	<a href="#"><u>HashMap</u></a>		<a href="#"><u>TreeMap</u></a>		<a href="#"><u>LinkedHashMap</u></a>

# Map interface examples

```
Map<String,Person> map = new HashMap<>();
map.put("Jeff", new Person("Jeff Nobody", 85.0));
Person person1 = map.get("Jeff");
System.out.println(person1); //Person{name='Jeff Nobody', weight=85.0}
Person person2 = map.remove("Jeff");
System.out.println(person2); //Person{name='Jeff Nobody', weight=85.0}
person2 = map.get("Jeff");
System.out.println(person2); // null
person2 = map.getDefault("Jeff", new Person("default", 0.0));
System.out.println(person2); //Person{name='default', weight=0.0}
if (map.containsKey("Jeff")) { }
if (map.containsValue(person1)) { }
```

# Map implementations

- It is a **challenge** to find a performant Map implementation:
  - “endless” possible keys vs. limited storage capacity
- Examples
  - Linear list (not provided in Java Collections API, but you can/should try it)
  - Hashtable (HashMap in Java)
  - Balanced Binary Tree (TreeMap in Java)



# Hashtable implementation

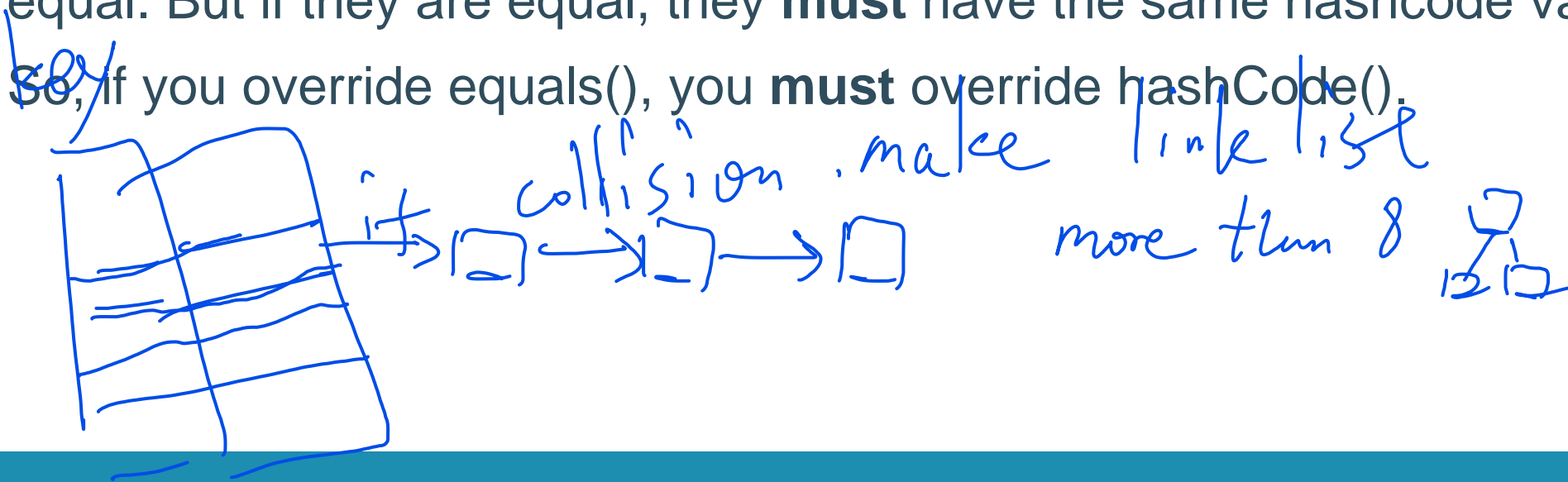
*if 2 keys' hash  
is same*

- Table with limited number of buckets (rows)
- put
  - $\text{hashCode()} \% \text{tablelength} \Rightarrow \text{index in table}$
  - collisions possible
    - each bucket is a (smaller) Collection
- get
  - calculate index ( $\text{hashCode()}$ )
  - look for the key value in bucket found
- goal: maximize **hit ratio**

*if it is filled  
with other key  
already  
just go to next  
empty*

# hashCode() and equals()

- If two objects are equal, the **must** have matching hashcodes.
- If two objects are equal, calling equals() on either object **must** return true. In other words, if (a.equals(b)) then (b.equals(a)).
- If two objects have the same hashCode value, they are **not** required to be equal. But if they are equal, they **must** have the same hashCode value.
- *key* So, if you override equals(), you **must** override hashCode().



# Hashtable: conclusion

- Array based
- Maximize hit ratio
  - Avoid key collisions
  - hashCode(): as unique as possible
  - equals() should follow hashCode()
- Tuneable by using
  - Size – Capacity – <sup>16</sup>Load factor<sup>0.75</sup>
  - Reallocation => re-hashing
- Special case: LinkedHashMap
- big-O...

if >  $16 \times 0.75$   
new table  
rehash

# HashMap: Java implementation

## From the API:

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important. An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.

# Balanced Binary Tree: implementation

- Binary tree: each node has maximum two subtrees
- Balanced: nr of elements for each subtree differs max. 1
- Ordering:
  - all values in left subtree are smaller than the node value
  - all values in right subtree are larger than the node value
  - recursively!
- How to add the following numbers?
  - 71, 92, 1, 13, 64, 27, 54, 89, 44, 37

# Balanced Binary Tree: example

## CONSTRUCTING a PERFECT BINARY TREE

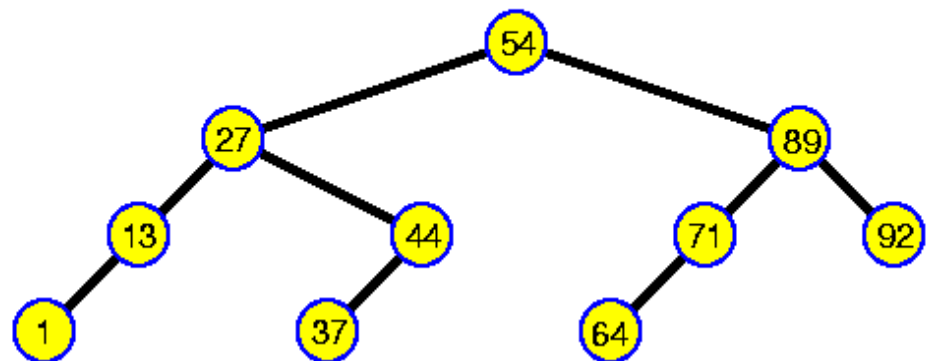
Unsorted set of nodes:



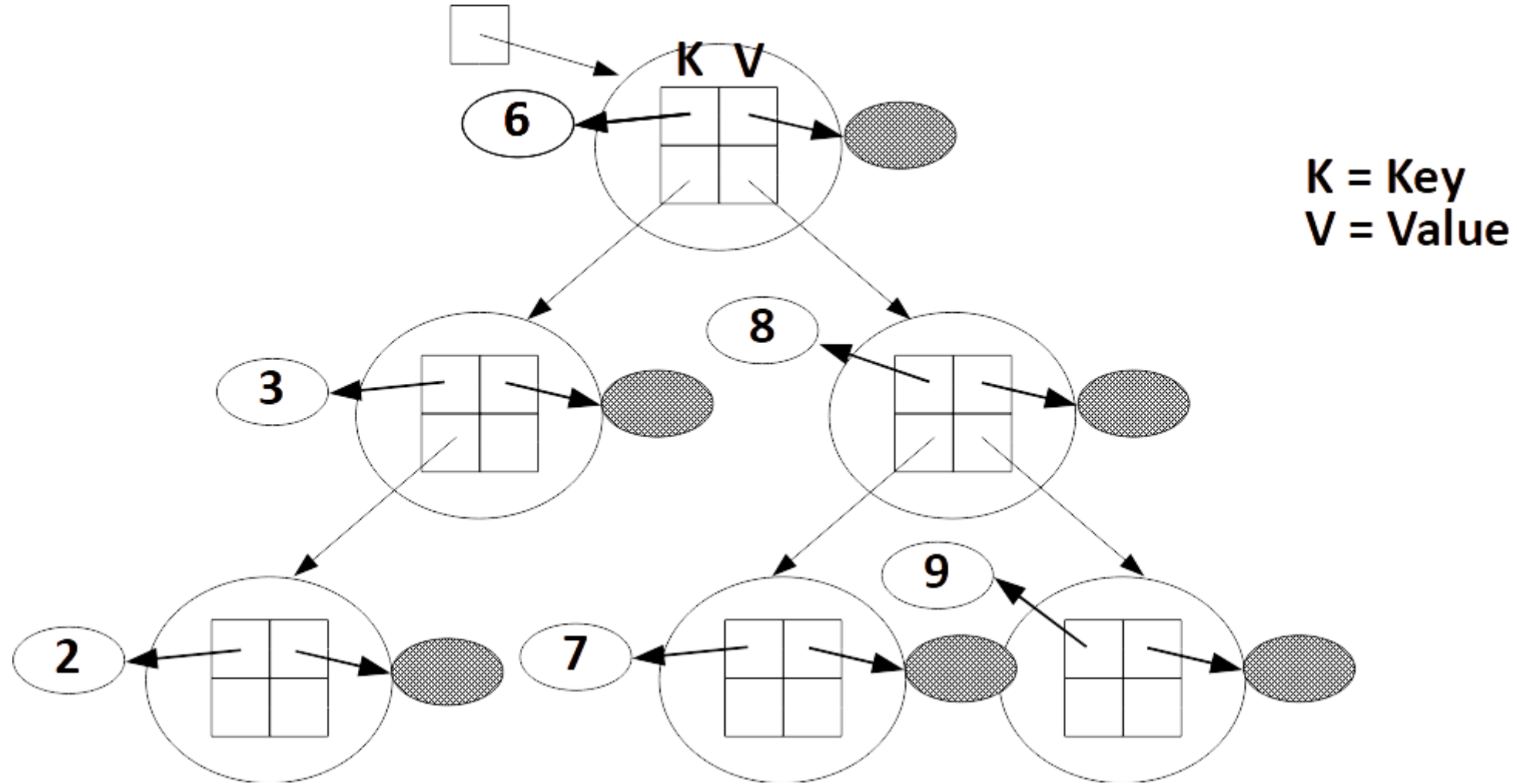
After sorting on key values:



After recursively taking middle node as root of subtree:



# TreeMap<Integer,V>



# TreeMap: conclusion

- Keys have to implement the Comparable interface or a Comparator has to be provided (constructor parameter of the TreeMap)
  - Keys are sorted
- Balanced binary tree
  - Internal ordering (Comparable)
  - External ordering (Comparator)
- Re-balance if necessary
- Binary search

	<i>get</i>	<i>containsKey</i>	<i>next</i>	<i>Note</i>
<i>HashMap</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>LinkedHashMap</i>	O(1)	O(1)	O(1)	
<i>IdentityHashMap</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>EnumMap</i>	O(1)	O(1)	O(1)	
<i>TreeMap</i>	O(log n)	O(log n)	O(log n)	
<i>ConcurrentHashMap</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>ConcurrentSkipListMap</i>	O(log n)	O(log n)	O(1)	



# TreeMap: Java implementation

## From the API:

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed  $\log(n)$  time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface

# LinkedHashMap: Java implementation

## From the API

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map. (A key *k* is reinserted into a map *m* if *m.put(k, v)* is invoked when *m.containsKey(k)* would return true immediately prior to the invocation.)

This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashMap (and Hashtable), without incurring the increased cost associated with TreeMap. It can be used to produce a copy of a map that has the same order as the original, regardless of the original map's implementation.

...

A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (access-order). This kind of map is well-suited to building LRU caches. Invoking the *put*, *putIfAbsent*, *get*, *getOrDefault*, *compute*, *computeIfAbsent*, *computeIfPresent*, or *merge* methods results in an access to the corresponding entry (assuming it exists after the invocation completes). The *replace* methods only result in an access of the entry if the value is replaced. The *putAll* method generates one entry access for each mapping in the specified map, in the order that key-value mappings are provided by the specified map's entry set iterator. No other methods generate entry accesses. In particular, operations on collection-views do not affect the order of iteration of the backing map.

# Map: some statistics/trends (only 1 run)

HashMap WITHOUT capacity estimation (50000 elements)

part2.PersonWithUniqueHashCode  
put: 54 ms; nr of hashCode() calls: 50000  
get: 9 ms; nr of hashCode() calls: 20000

\*\*\*\*\*

HashMap WITH capacity estimation (100000 buckets)(50000 elements)

part2.PersonWithUniqueHashCode  
put: 20 ms; nr of hashCode() calls: 50000  
get: 0 ms; nr of hashCode() calls: 20000

\*\*\*\*\*

HashMap with collisions (every Person with same hashCode)("50000" elements)

part2.PersonWithoutUniqueHashCode  
put: 75095 ms; nr of hashCode() calls: 50000  
get: 0 ms; nr of hashCode() calls: 20000

Treemap (50000 elements)

part2.PersonWithUniqueHashCode  
put: 84 ms; nr of hashCode() calls: 0  
get: 10 ms; nr of hashCode() calls: 0

\*\*\*\*\*

Treemap (50000 elements)

part2.PersonWithoutUniqueHashCode  
put: 95 ms; nr of hashCode() calls: 0  
get: 5 ms; nr of hashCode() calls: 0

# Map: some statistics (100 iterations)

HashMap with unique hashcode without capacity, "put": 46 ms

HashMap with unique hashcode with 3 \* capacity, "put": 31 ms

HashMap without unique hashcode without capacity, "put": 51545 ms

HashMap without unique hashcode with 3 \* capacity, "put": 47156 ms

# Set implementations

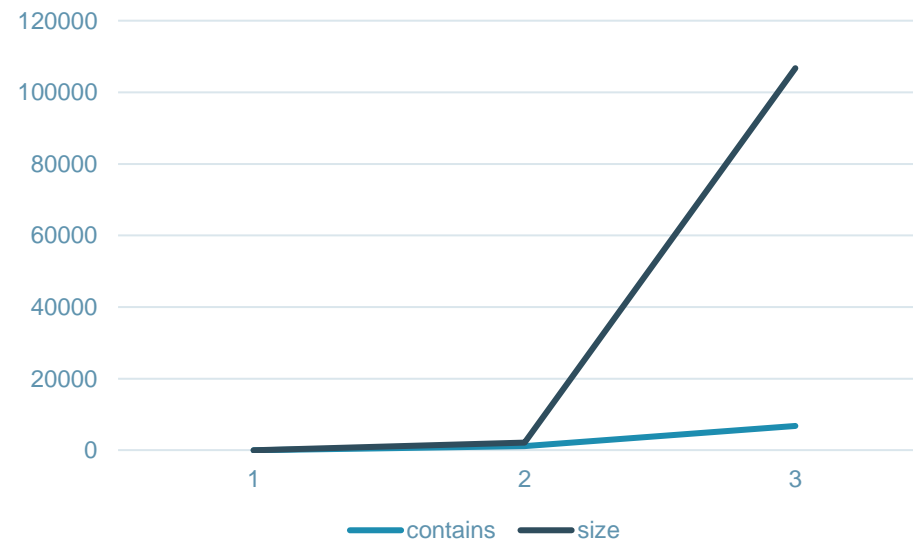
- Similar to Map implementations
  - keys only (unique)
  - no values
- Most common concrete implementations in Java
  - HashSet
  - TreeSet
  - LinkedHashSet

# Set: some statistics related to size (+ demo)

Contains operation on TreeSet with size 10: 15 ms

Contains operation on TreeSet with size 1000: 1136 ms

Contains operation on TreeSet with size 100000: 6783 ms



# Collections

## Conclusion

# Iterate through Collections

- Low-level loops: for, while, do while
- Iterator interface
- Java 5: enhanced for-loop (“foreach loop”)
- Java 8: forEach + lambda expression



# Collection implementations overview

General purpose implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<a href="#">HashSet</a>		<a href="#">TreeSet</a>		<a href="#">LinkedHashSet</a>
List		<a href="#">ArrayList</a>		<a href="#">LinkedList</a>	
Deque		<a href="#">ArrayDeque</a>		<a href="#">LinkedList</a>	
Map	<a href="#">HashMap</a>		<a href="#">TreeMap</a>		<a href="#">LinkedHashMap</a>

# Big-O and Java Collections

	<i>get</i>	<i>add</i>	<i>contains</i>	<i>next</i>	<i>remove(O)</i>	<i>Iterator.remove</i>
<i>ArrayList</i>	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
<i>LinkedList</i>	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
<i>CopyOnWriteArrayList</i>	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

	<i>get</i>	<i>containsKey</i>	<i>next</i>	<i>Note</i>
<i>HashMap</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>LinkedHashMap</i>	O(1)	O(1)	O(1)	
<i>IdentityHashMap</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>EnumMap</i>	O(1)	O(1)	O(1)	
<i>TreeMap</i>	O(log n)	O(log n)	O(log n)	
<i>ConcurrentHashMap</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>ConcurrentSkipListMap</i>	O(log n)	O(log n)	O(1)	

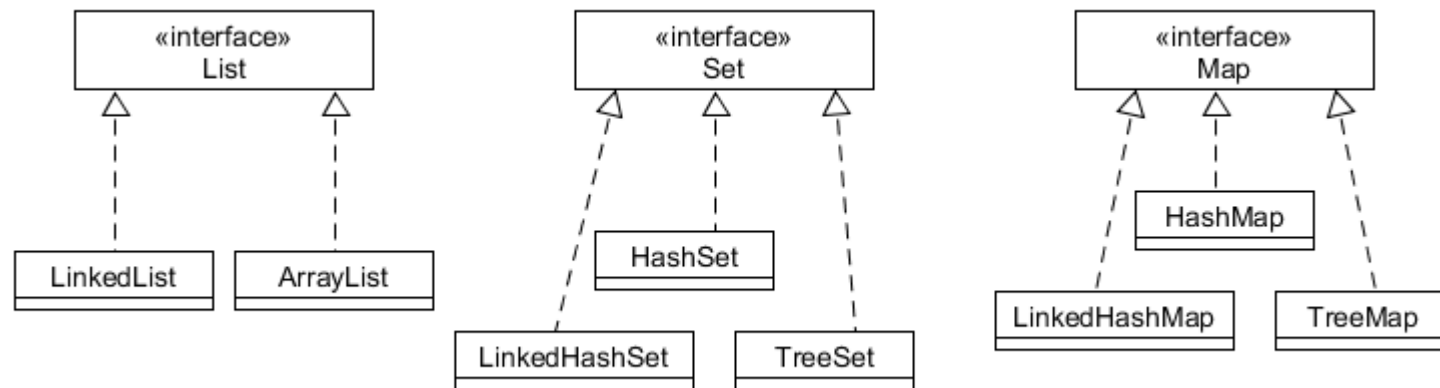
	<i>add</i>	<i>contains</i>	<i>next</i>	<i>Note</i>
<i>HashSet</i>	O(1)	O(1)	O(h/n)	h is the table capacity
<i>LinkedHashSet</i>	O(1)	O(1)	O(1)	
<i>CopyOnWriteArraySet</i>	O(n)	O(n)	O(1)	
<i>EnumSet</i>	O(1)	O(1)	O(1)	
<i>TreeSet</i>	O(log n)	O(log n)	O(log n)	
<i>ConcurrentSkipListSet</i>	O(log n)	O(log n)	O(1)	

	<i>offer</i>	<i>peek</i>	<i>poll</i>	<i>size</i>
<i>PriorityQueue</i>	O(log n)	O(1)	O(log n)	O(1)
<i>ConcurrentLinkedQueue</i>	O(1)	O(1)	O(1)	O(n)
<i>ArrayBlockingQueue</i>	O(1)	O(1)	O(1)	O(1)
<i>LinkedBlockingQueue</i>	O(1)	O(1)	O(1)	O(1)
<i>PriorityBlockingQueue</i>	O(log n)	O(1)	O(log n)	O(1)
<i>DelayQueue</i>	O(log n)	O(1)	O(log n)	O(1)
<i>LinkedList</i>	O(1)	O(1)	O(1)	O(1)
<i>ArrayDeque</i>	O(1)	O(1)	O(1)	O(1)
<i>LinkedBlockingDeque</i>	O(1)	O(1)	O(1)	O(1)

Source:Wikipedia

# Collections: conclusion

- If possible: program towards the interface, unless you want to use class-specific methods (see: List)
- Defer the choice for a concrete collection implementation until you know about the actions to be performed on the collection
- Performance should be the last concern



# Collections: exercises

- 1) Count the number of **different** words in a text and
  - a) show this number
  - b) show the words
  - c) show the words in alphabetic order
  - d) show the words in reverse alphabetic order
- 2) Count the word **frequency** in a text and
  - a) show the result (word + frequency)
  - b) show the result with the words in the order of their first occurrence in the text
  - c) show the result with the words in an alphabetic order
  - d) show the result with the words in a reverse alphabetic order
  - e) show the result from highest to lowest frequency