Java Collections Framework

Object-Oriented Programming

SOftEng
http://softeng.polito.it

Version 4.4.0

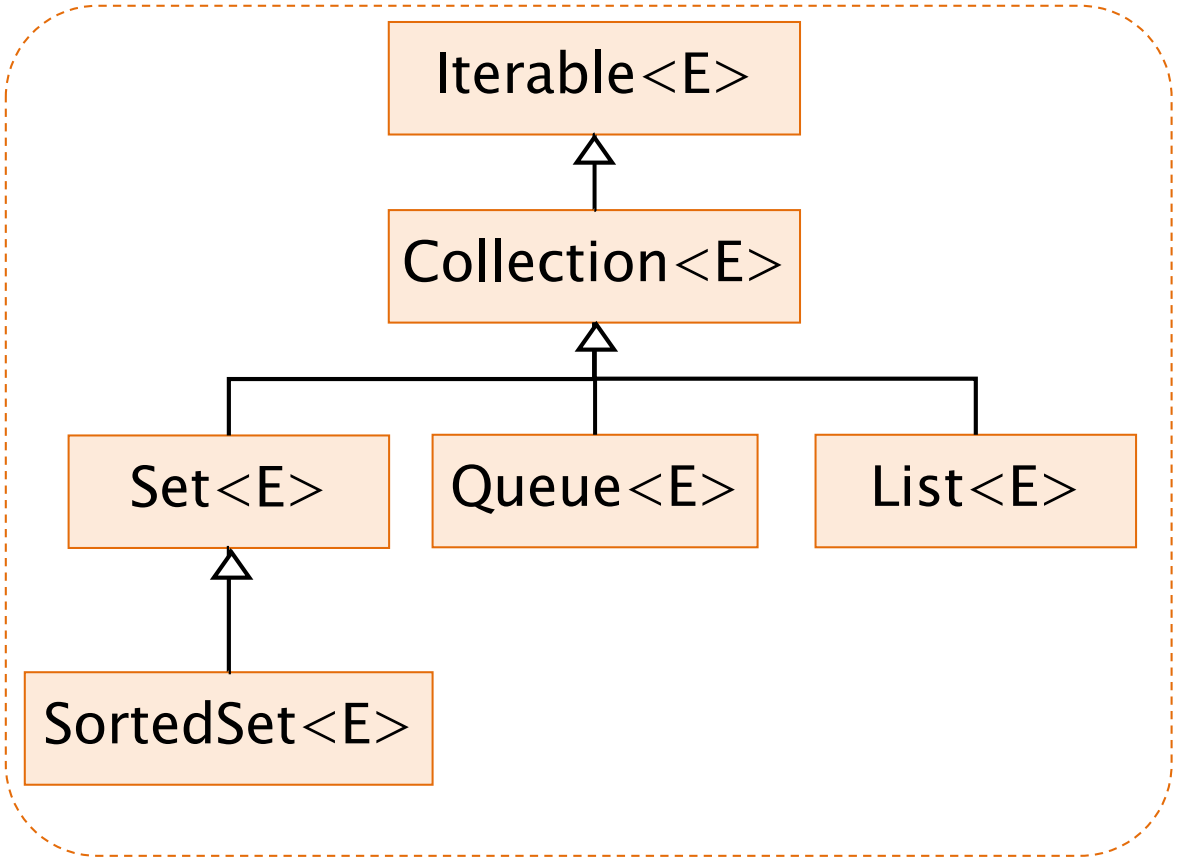
Licensing Note

Collections Framework

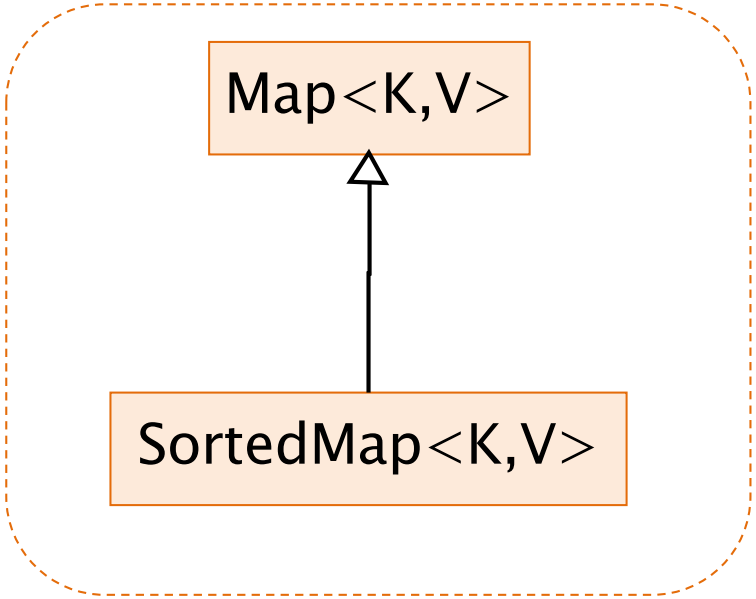
- Interfaces (ADT, Abstract Data Types)
- Implementations (of ADT)
- Algorithms (sort)
- Contained in the package **`java.util`**

- They originally used **`Object`**, since Java 5 redefined as generic types

Interfaces

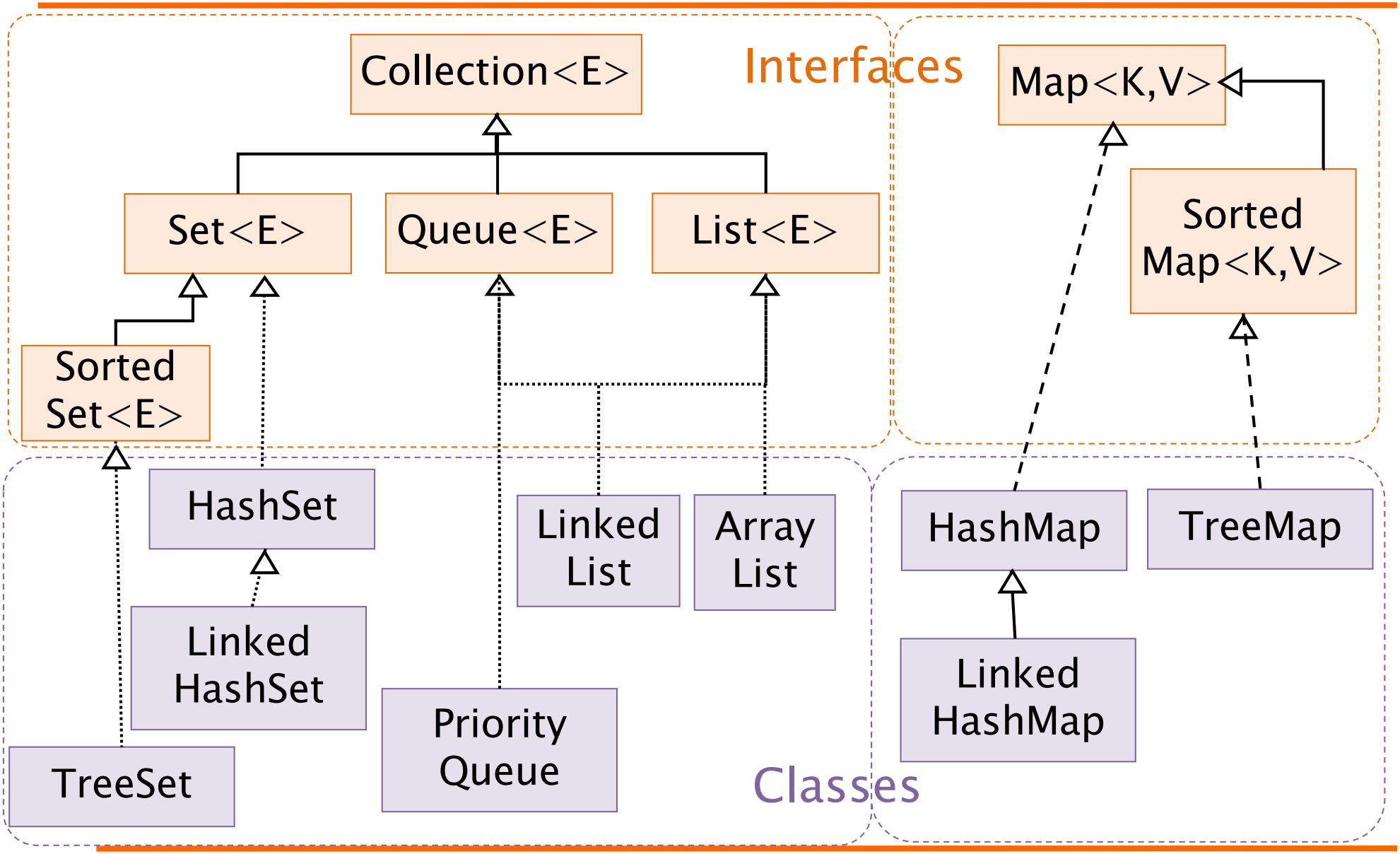


Group containers

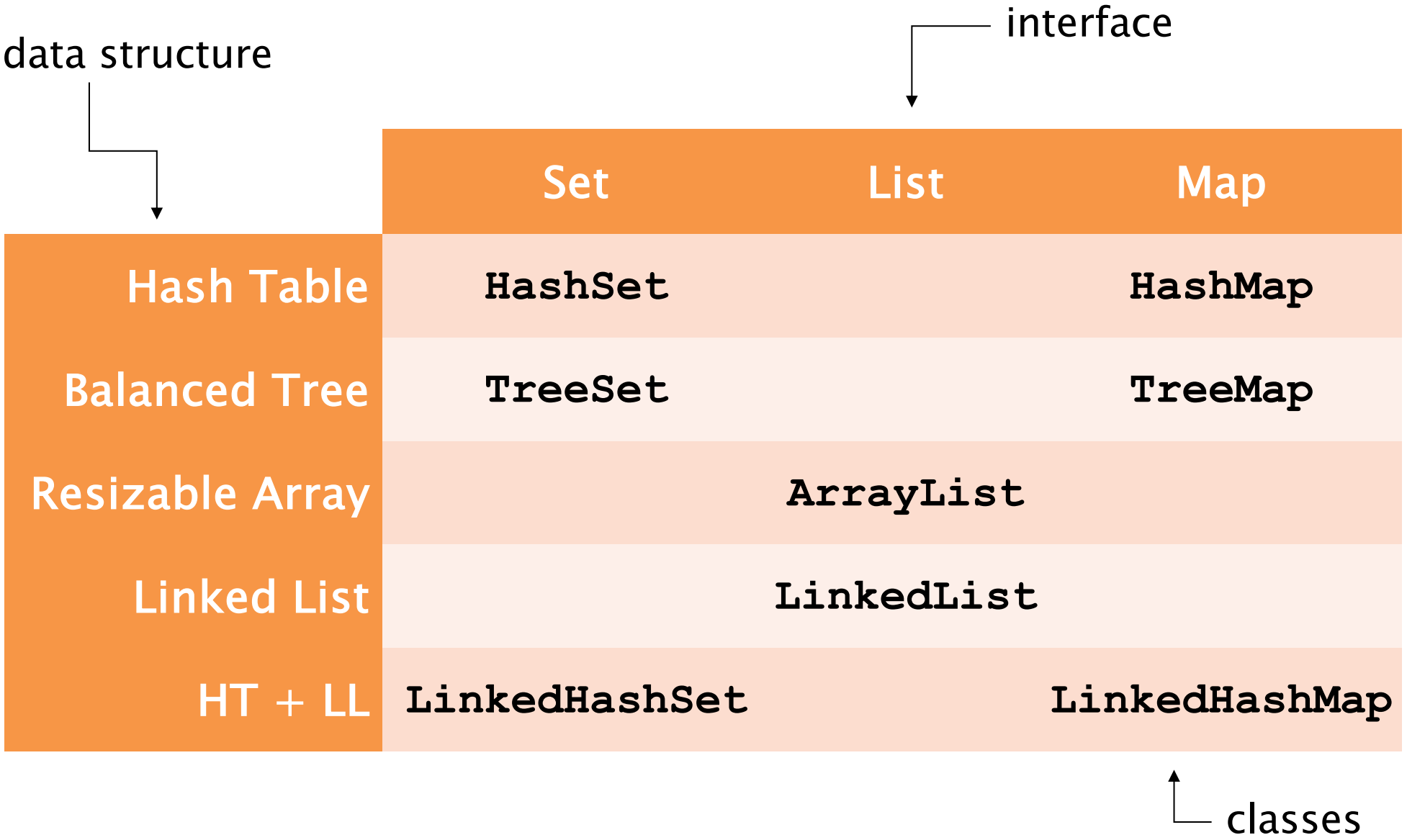


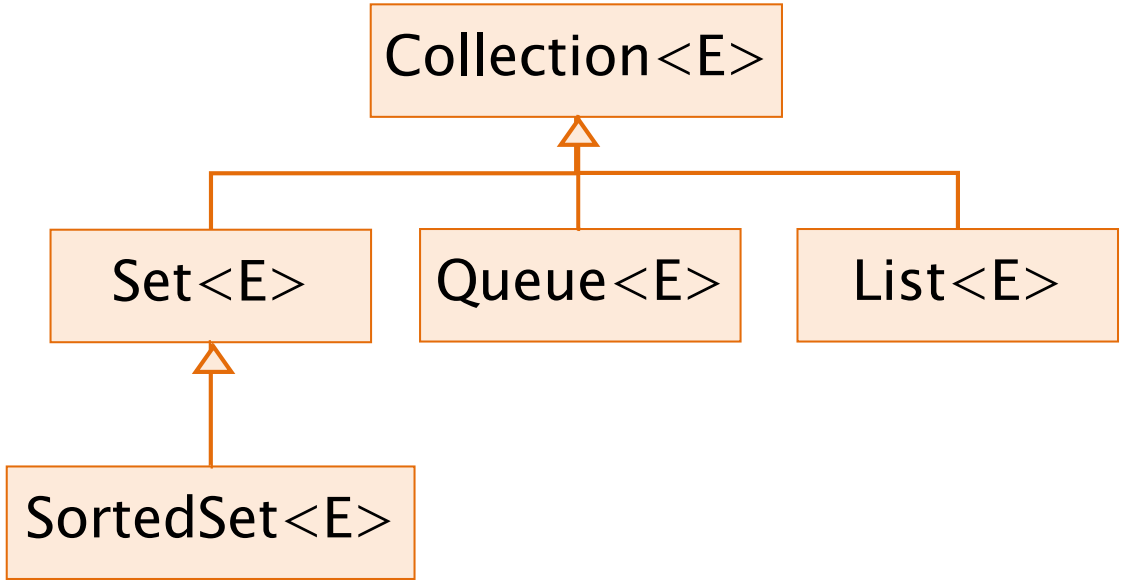
Associative containers

Implementations



Internals


interface
data structure
Set
List
Map
Hash Table
HashSet
HashMap
Balanced Tree
TreeSet
TreeMap
Resizable Array
ArrayList
Linked List
LinkedList
HT + LL
LinkedHashSet
LinkedHashMap
classes



GROUP CONTAINERS (COLLECTIONS)

`Collection`

- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Implements **`Iterable`**
- All classes implementing **`Collection`** shall provide two constructors
  - ***C*`()`**
  - ***C*`(Collection c)`**

`Collection` interface

```
int size()
boolean isEmpty()
boolean contains(E element)
boolean containsAll(Collection<?> c)
boolean add(E element)
boolean addAll(Collection<? extends E> c)
boolean remove(E element)
boolean removeAll(Collection<?> c)
void clear()
Object[] toArray()
Iterator<E> iterator()
```

Method `toArray()`

- `Object[] toArray()`
  - Due to type erasure the collections do not know the type of elements
  - For generality an `Object[]` is returned
- `<T> T[] toArray(T[])`
  - The correct type of array is passed
    - If not large enough a new array of the correct type and right length is created and returned

Collection example

```
Collection<Person> persons =
                        new LinkedList<Person>();
persons.add( new Person("Alice") );
System.out.println( persons.size() );
Collection<Person> copy =
                      new TreeSet<Person>();
copy.addAll(persons);//new TreeSet(persons)
Person[] array= copy.toArray(new Person[0]);
System.out.println( array[0] );
```

`List`

- Can contain duplicate elements
- Insertion order is preserved
- User can define insertion point
- Elements can be accessed by position
- Extends `Collection` interface

`List` interface

```
E get(int index)
E set(int index, E element)
void add(int index, E element)
E remove(int index)

boolean addAll(int index,Collection<E> c)
int indexOf(E o)
int lastIndexOf(E o)
List<E> subList(int from, int to)
```

Example

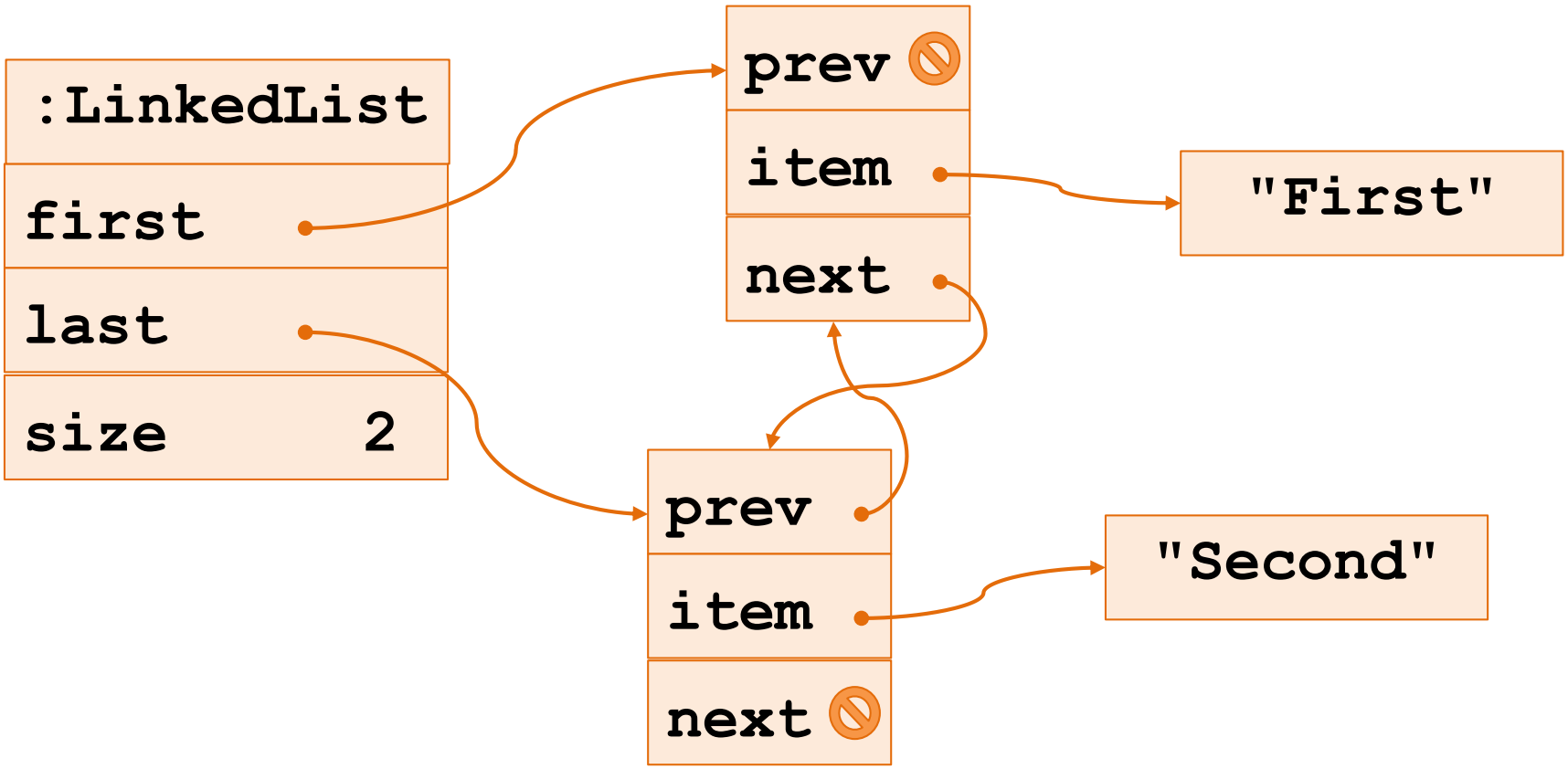
```
List<Integer> l = new ArrayList<>();


l.add(42);          // 42 in position 0
l.add(0, 13);       // 42 moved to position 1
l.set(0, 20);       // 13 replaced by 20
int a = l.get(1); // returns 42
l.add(9, 30);       // NO: out of bounds
```

IndexOutOfBoundsException

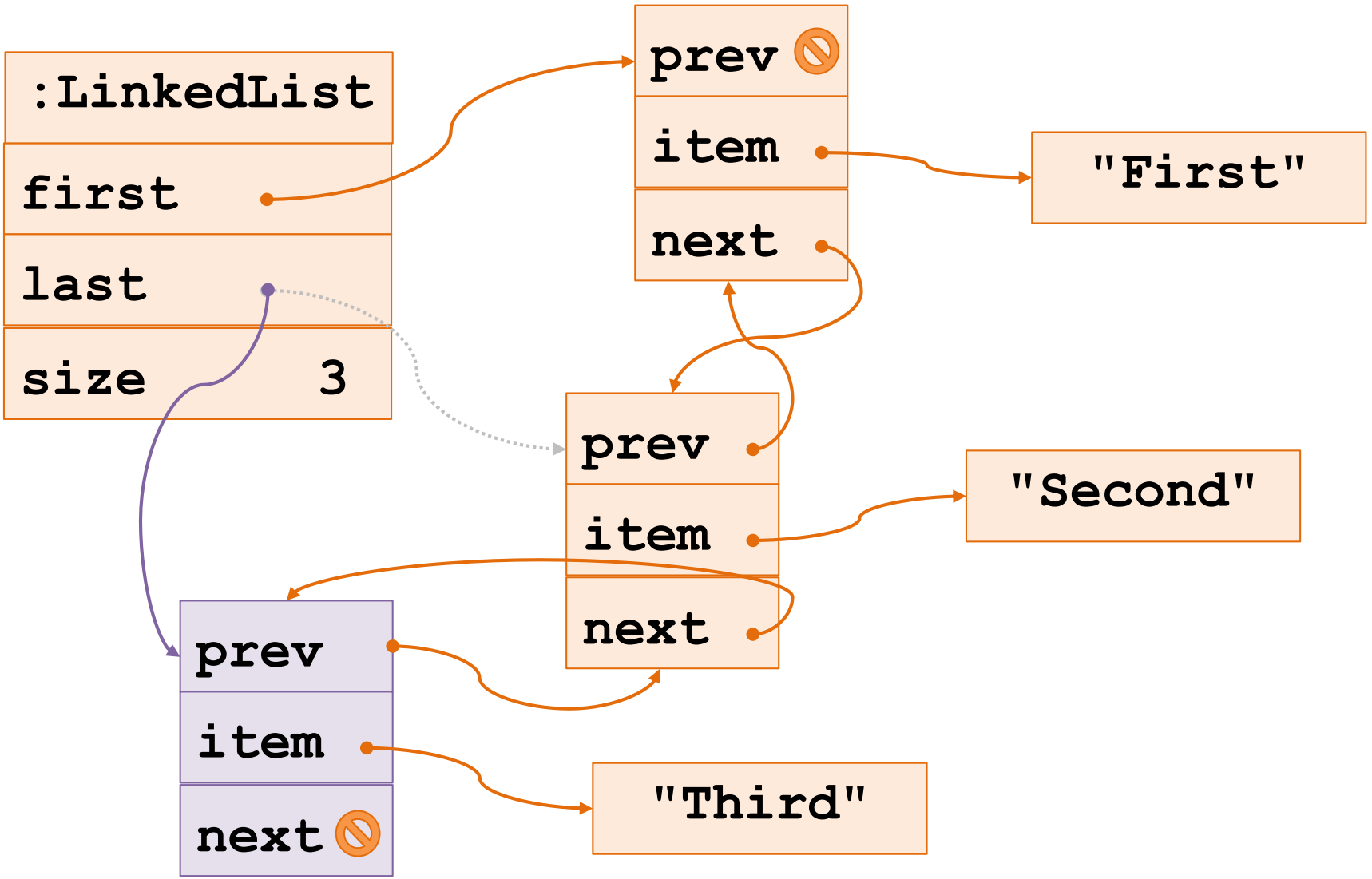
`List` implementations

- **`ArrayList<E>`**
  - `ArrayList()`
  - `ArrayList(int initialCapacity)`
  - `ArrayList(Collection<E> c)`
  - `void ensureCapacity(int minCapacity)`
- **`LinkedList<E>`**
  - `void addFirst(E o)`
  - `void addLast(E o)`
  - `E getFirst()`
  - `E getLast()`
  - `E removeFirst()`
  - `E removeLast()`

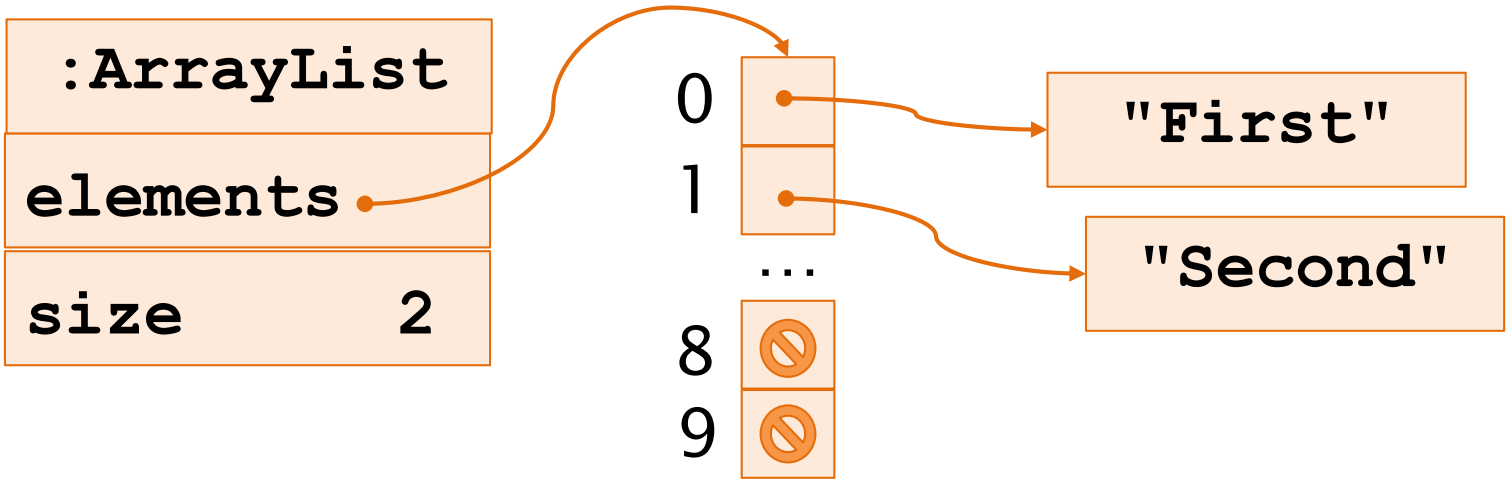
Linked list


:LinkedList
first
last
size 2
prev
item
next
"First"
prev
item
next
"Second"

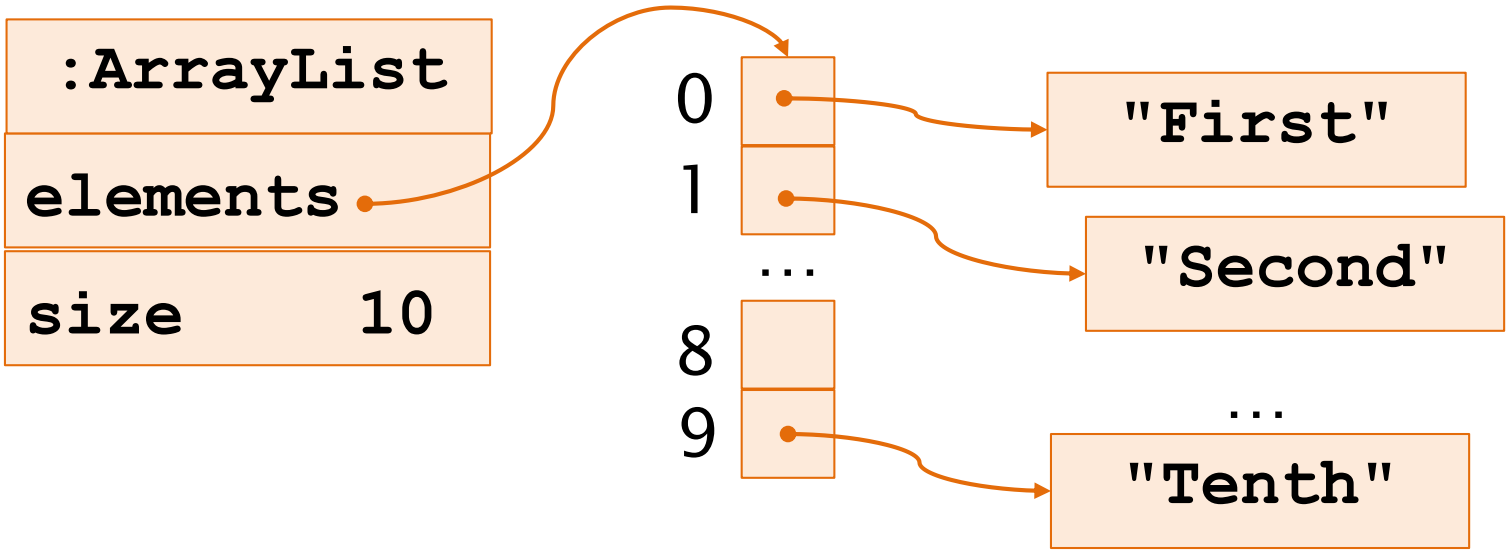
Linked list


:LinkedList
first
last
size 3
prev
item
next
"First"
prev
item
next
"Second"
prev
item
next
"Third"

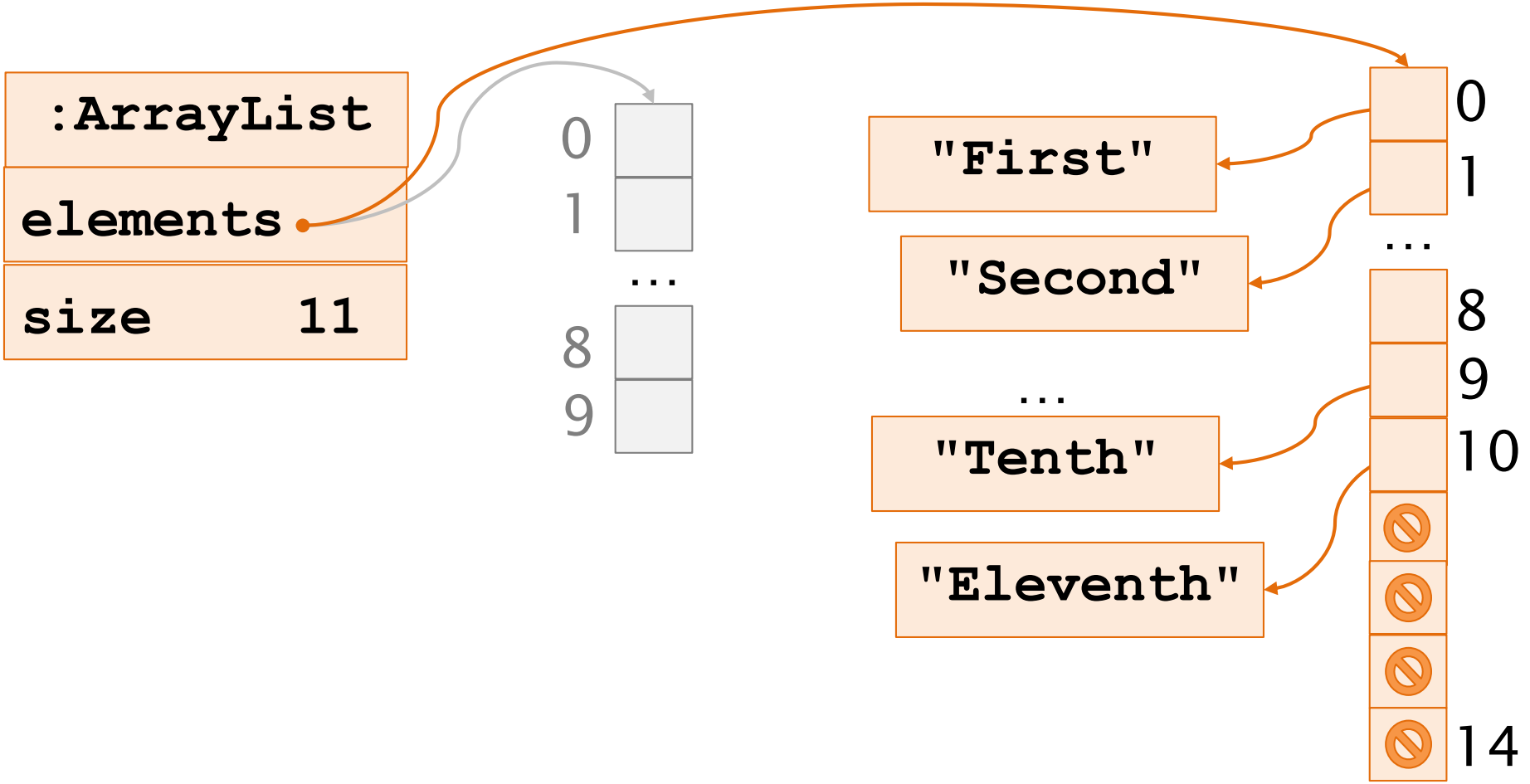
Array list



Array list



Array list


:ArrayList
elements
size 11
0
1
...
8
9
"First"
"Second"
...
"Tenth"
"Eleventh"
0
1
...
8
9
10
14

Example II

```
Car[] garage = new Car[20];

garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] =
garage[3] =

for(int i=0;
   garage[i]
}
```

```
List<Car> garage = new ArrayList<Car>(20);

garage.set( 0, new Car() );
garage.set( 1, new ElectricCar() );
garage.set( 2, new ElectricCar() );
garage.set( 3, new Car());

for(int i; i<garage.size(); i++){
   Car c = garage.get(i);
   c.turnOn();
}
```

Example LinkedList

```
LinkedList<Integer> ll =
                new LinkedList<Integer>();

ll.add(Integer.valueOf(10));
ll.add( 11 );

ll.addLast( 13 );
ll.addFirst( 20 );

//20, 10, 11, 13
```

`Queue` interface

- Collection whose elements are inserted using an
  - Insertion order (FIFO)
  - Element natural order (Priority queue)
- Defines a head position where is the first element that can be accessed
  - **`peek()`** just retrieves
  - **`poll()`** retrieves and removes

`Queue` implementations

- **`LinkedList`**
  - head is the first element of the list
  - FIFO: Fist-In-First-Out
- **`PriorityQueue`**
  - head is the smallest element

Queue example

```
Queue<Integer> fifo =
          new LinkedList<Integer>();
Queue<Integer> pq =
          new PriorityQueue<Integer>();
fifo.add(3); pq.add(3);
fifo.add(1); pq.add(1);
fifo.add(2); pq.add(2);
System.out.println(fifo.peek()); // 3
System.out.println(pq.peek());   // 1
```

`Set` interface

- Contains no methods
  - Only those inherited from **`Collection`**
- **`add()`** has the restriction that no duplicate elements are allowed
  - **`e1.equals(e2) == false`** $\forall$ `e1,e2` $\in \Sigma$
- Iterator
  - The elements are traversed in no particular order

`SortedSet` interface

- No duplicate elements
- Iterator
  - The elements are traversed according to the natural ordering (ascending)
- Augments Set interface
  - `E first()`
  - `E last()`
  - `SortedSet<E> headSet(E toElement)`
  - `SortedSet<E> tailSet(E fromElement)`
  - `SortedSet<E> subSet(E from, E to)`

`Set` implementations

- `HashSet` implements `Set`
  - Hash tables as internal data structure (faster)
- `LinkedHashSet` extends `HashSet`
  - Elements are traversed by iterator according to the insertion order
- `TreeSet` implements `SortedSet`
  - R-B trees as internal data structure (computationally expensive)

Note on sorted collections

- Depending on the constructor used they require different implementation of the custom ordering
- `TreeSet()`
  - Natural ordering (elements must be implementations of Comparable)
- `TreeSet(Comparator c)`
  - Ordering is according to the comparator rules, instead of natural ordering

ITERATORS

`Iterable` interface

- Container of elements that can be iterated upon
- Provides a single instance method:

```
Iterator<E> iterator()
```

  - It returns the iterator on the elements of the collection
- Collection extends Iterable

Iterators and iteration

- A common operation with collections is to iterate over their elements
- Interface Iterator provides a transparent means to cycle through all elements of a Collection
- Keeps track of last visited element of the related collection
- Each time the current element is queried, it moves on automatically

Iterator

- Allows the iteration on the elements of a collection
- Two main methods:
  - **`boolean hasNext()`**
    - Checks if there is a next element to iterate on
  - **`E next()`**
    - Returns the next element and advances by one position
  - **`void remove()`**
    - Optional method, removes the current element

`Iterator` examples

Print all objects in a list

```
Iterable<Person> persons =
                    new LinkedList<Person>();
…
for(Iterator<Person> i = persons.iterator();
                                 i.hasNext(); ) {
    Person p = i.next();
    …
   System.out.println(p);
}
```

`Iterator` examples

The for-each syntax avoids using iterator directly

```
Iterable<Person> persons =
                    new LinkedList<Person>();
…
for(Person p: persons) {
    …
   System.out.println(p);
}
```

Iterable `forEach`

- Iterable defines the default method

  `forEach(Consumer<? super T> action)`

- Can be used to perform operations of elements with a functional interface

```
Iterable<Person> persons;
…
persons.forEach( p -> {
   System.out.println(p);
});
```

Note well

- It is unsafe to iterate over a collection you are modifying (add/remove) at the same time

- Unless you are using the iterator's own methods
  - `Iterator.remove()`
  - `ListIterator.add()`

Delete

```
List<Integer> lst=new LinkedList<>();
lst.add( 10 );
lst.add( 11 );
lst.add( 13 );
lst.add( 20 );

int count = 0;
for (Iterator<?> itr = lst.iterator();
                          itr.hasNext(); ) {
   itr.next();
   if (count==1)
      lst.remove(count); // wrong
   count++;
}
```

ConcurrentModificationException

Delete (cont'd)

```
List<Integer> lst = new LinkedList<>();
lst.add( 10 );
lst.add( 11 );
lst.add( 13 );
lst.add( 20 );

int count = 0;
for (Iterator<?> itr = lst.iterator();
                    itr.hasNext(); ) {
   itr.next();
   if (count==1)
      itr.remove(); // ok
   count++;
}
```

Correct

Add

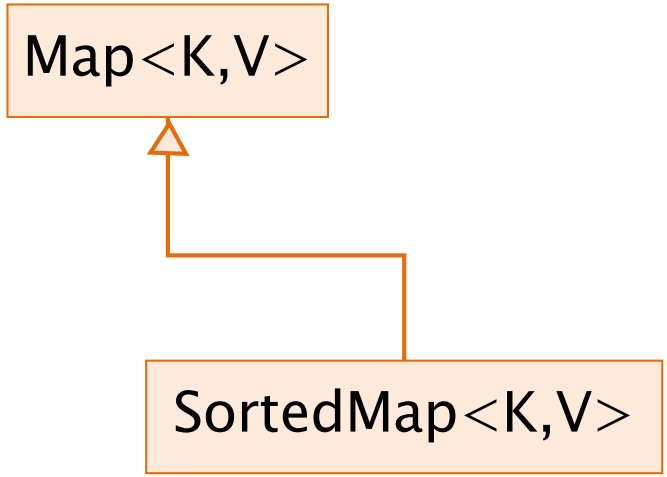
```
List<Integer> lst = new LinkedList<>();
lst.add( 10 );
lst.add( 11 );
lst.add( 13 );
lst.add( 20 );

int count = 0;
for (Iterator itr = lst.iterator();
                    itr.hasNext(); ) {
   itr.next();
   if (count==2)
     lst.add(count, 22 );//wrong
   count++;
}
```

ConcurrentModificationException

Add (cont'd)

```
List<Integer> lst = new LinkedList<>();
lst.add( 10 );
lst.add( 11 );
lst.add( 13 );
lst.add( 20 );

int count = 0;
for (ListIterator<Integer> itr =
      lst.listIterator(); itr.hasNext();){
   itr.next();
   if (count==2)
      itr.add(new Integer(22)); // ok
   count++;
}
```

Correct



Associative containers (Maps)

Map

- A container that associates keys to values (e.g., SSN ⇒ Person)
- Keys and values must be objects
- Keys must be unique
  - Only one value per key

- Following constructors are common to all collection implementers
  - `M()`
  - `M(Map m)`

`Map` interface

- `V put(K key, V value)`
- `V get(K key)`
- `Object remove(K key)`
- `boolean containsKey(K key)`
- `boolean containsValue(V value)`
- `public Set<K> keySet()`
- `public Collection<V> values()`
- `int size()`
- `boolean isEmpty()`
- `void clear()`

Map example: put and get

```
Map<String,Person> people =new HashMap<>();
people.put( "ALCSMT", //ssn
  new Person("Alice Smith") );
people.put( "RBTGRN", //ssn
  new Person("Robert Green") );

if( ! people.containsKey("RBTGRN"))
  System.out.println( "Not found" );

Person bob = people.get("RBTGRN");

int populationSize = people.size();
```

Map ex.: values and keySet

```
// Print all people
for(Person p : people.values()){
  System.out.println(p);
}
// Print all ssn
for(String ssn : people.keySet()){
  System.out.println(ssn);
}
```

`SortedMap` interface

- The elements are traversed according to the keys' natural ordering
  - Or using comparator passed to ctor
- Augments **`Map`** interface
  - **`SortedMap subMap(K fromKey, K toKey)`**
  - **`SortedMap headMap(K toKey)`**
  - **`SortedMap tailMap(K fromKey)`**
  - **`K firstKey()`**
  - **`K lastKey()`**

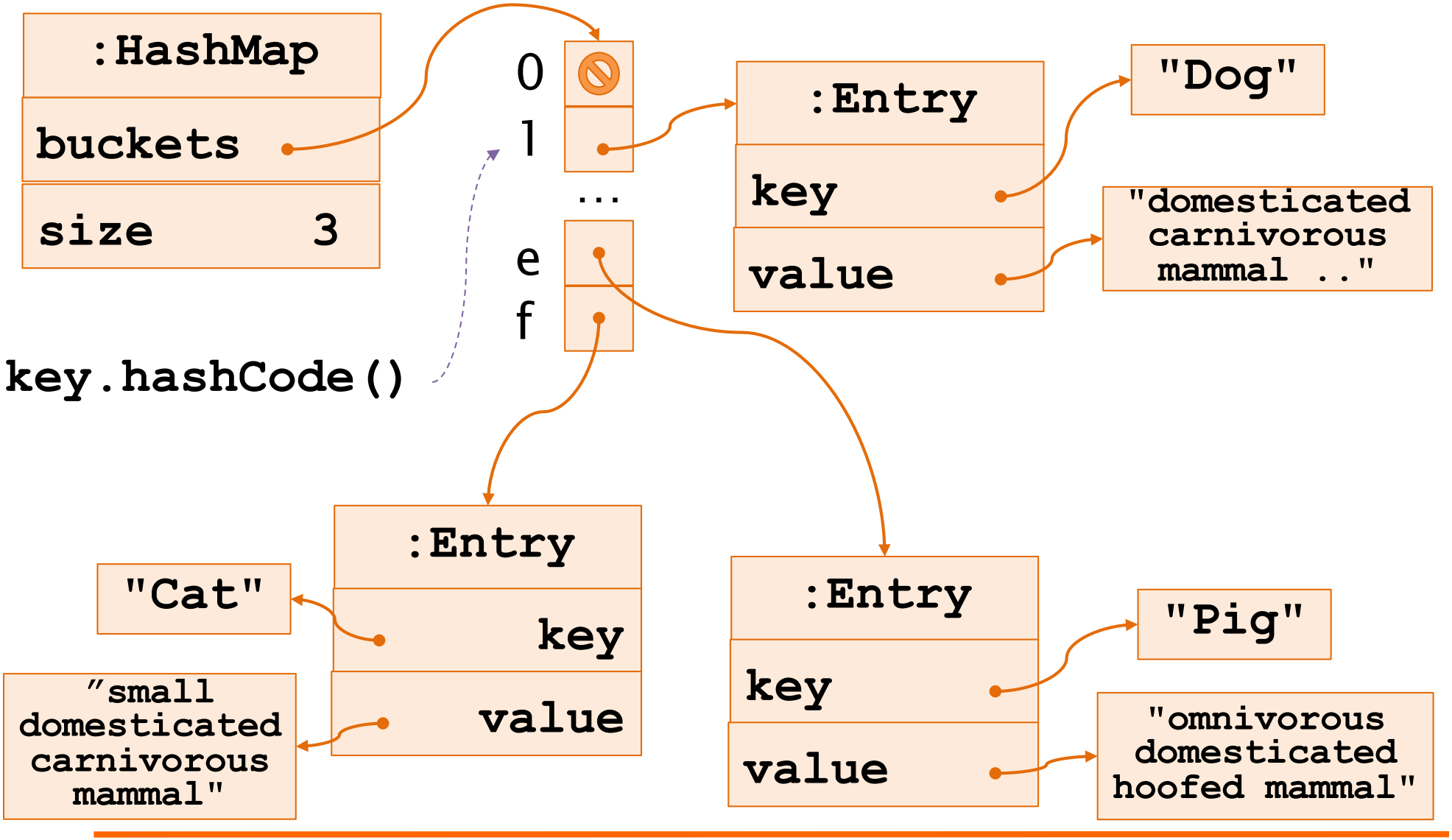
`Map` implementations

- Similar to `Set`
- `HashMap` implements `Map`
  - No order
- `LinkedHashMap` extends `HashMap`
  - Insertion order
- `TreeMap` implements `SortedMap`
  - Ascending key order

HashMap

- Get/put takes constant time (in case of no collisions)
- Automatic re-allocation when load factor reached
- Constructor optional arguments
  - load factor (default = .75)
  - initial capacity (default = 16)

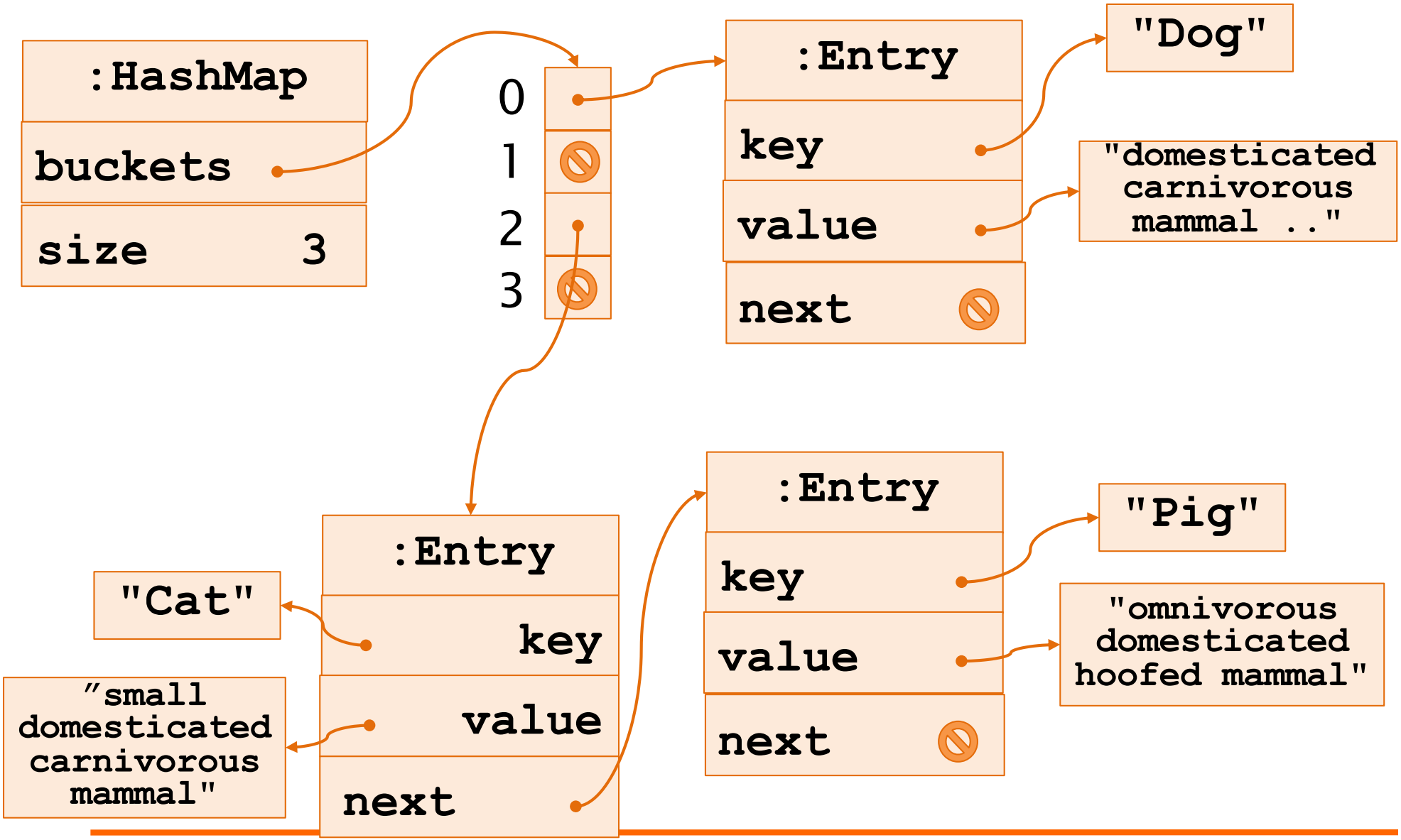
Hashmap


:HashMap
buckets
size 3
0
1
...
e
f
key.hashCode()
:Entry
key
value
"Dog"
"domesticated carnivorous mammal .."
:Entry
key
value
"Cat"
"small domesticated carnivorous mammal"
:Entry
key
value
"Pig"
"omnivorous domesticated hoofed mammal"

Hash limitations

- Hash based containers **`HashMap`** and **`HashSet`** work better if entries define a suitable **`hashCode()`** method
  - Values must be as spread as possible
  - Otherwise, collisions occur
    - When two entries fall in the same bucket
    - In such a case elements are put in a chained in a list
    - Chaining reduces time efficiency

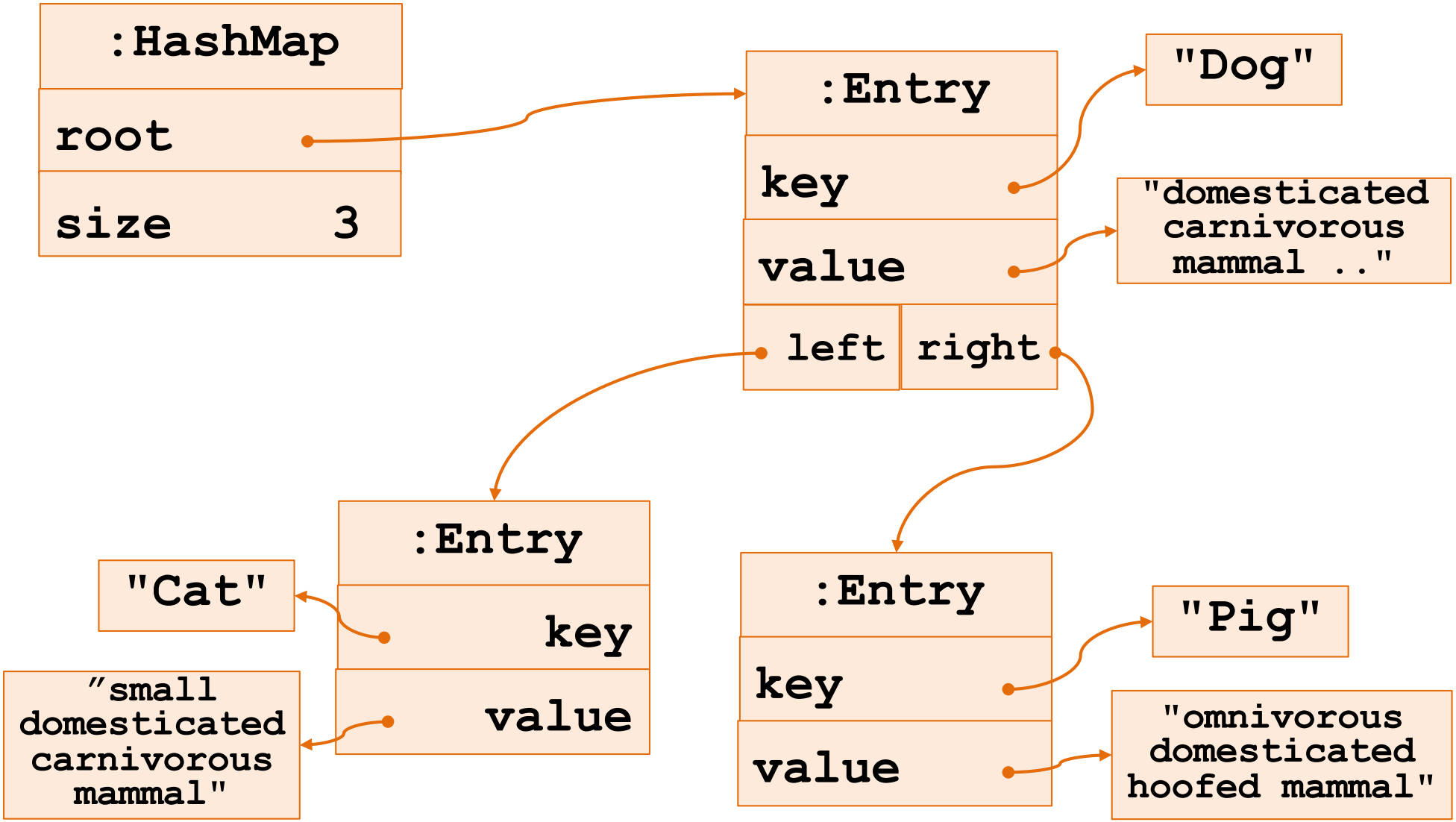
Hashmap (chaining)



`TreeMap`

- Based on a Red-Black tree
- Get/put takes log time
- Keys are maintained and will be traversed in order
  - Key class must be **`Comparable`**
  - Or a **`Comparator`** must be provided to the constructor

TreeMap


:HashMap
root
size 3
:Entry
key
value
left
right
"Dog"
"domesticated carnivorous mammal .."
:Entry
"Cat"
key
value
"small domesticated carnivorous mammal"
:Entry
key
value
"Pig"
"omnivorous domesticated hoofed mammal"

Tree limitations

- Tree based containers (**`TreeMap`** and **`TreeSet`**) require either
  - Entries with a natural order (**`Comparable`**)
  - A **`Comparator`** to sort entries
- **`TreeMap`** maintains keys sorted, and return values sorted by key

Optional

Nullability problem

- The typical convention in Java APIs is to let a method return a **`null`** reference to represent the absence of a result.
- The caller must check the return value of the method to detect that case
- In absence of checks *NPE*s may occur
  - *NPE* is **`NullPointerException`**

`Optional`

- `Optional` is a class used to represent a potential value
- Methods returning `Optional<T>` make explicit that the return value may be missing
  - Forces the clients to deal with potentially empty optional

Optional<T>

- Access to embedded value through
  - **`boolean isPresent()`**
    - checks if Optional contains a value
  - **`ifPresent(Consumer<T> block)`**
    - executes the given code if a value is present.
  - **`T get()`**
    - returns the value if present; otherwise it throws a **`NoSuchElementException`**.
  - **`T orElse(T default)`**
    - returns the value if present; otherwise it returns a **`default`** value.
  - **`T orElse(Supplier<T> s)`**
    - when empty return the value supplied by **`s`**

`Optional<T>`

- Creation uses static factory methods:
  - **`of(T v)`**:
    - throw exception if **`v`** is **`null`**
  - **`ofNullable(T v)`**:
    - returns an empty Optional when **`v`** is **`null`**
  - **`empty()`**
    - returns an empty Optional
  - Such methods force the programmer to think about what he's about to return

Using Collections

Use general interfaces

- E.g. `List<>` is better than `LinkedList<>`

- General interfaces are more flexible for future changes
- Makes you think
  - First about the type of container
  - Then about the implementation

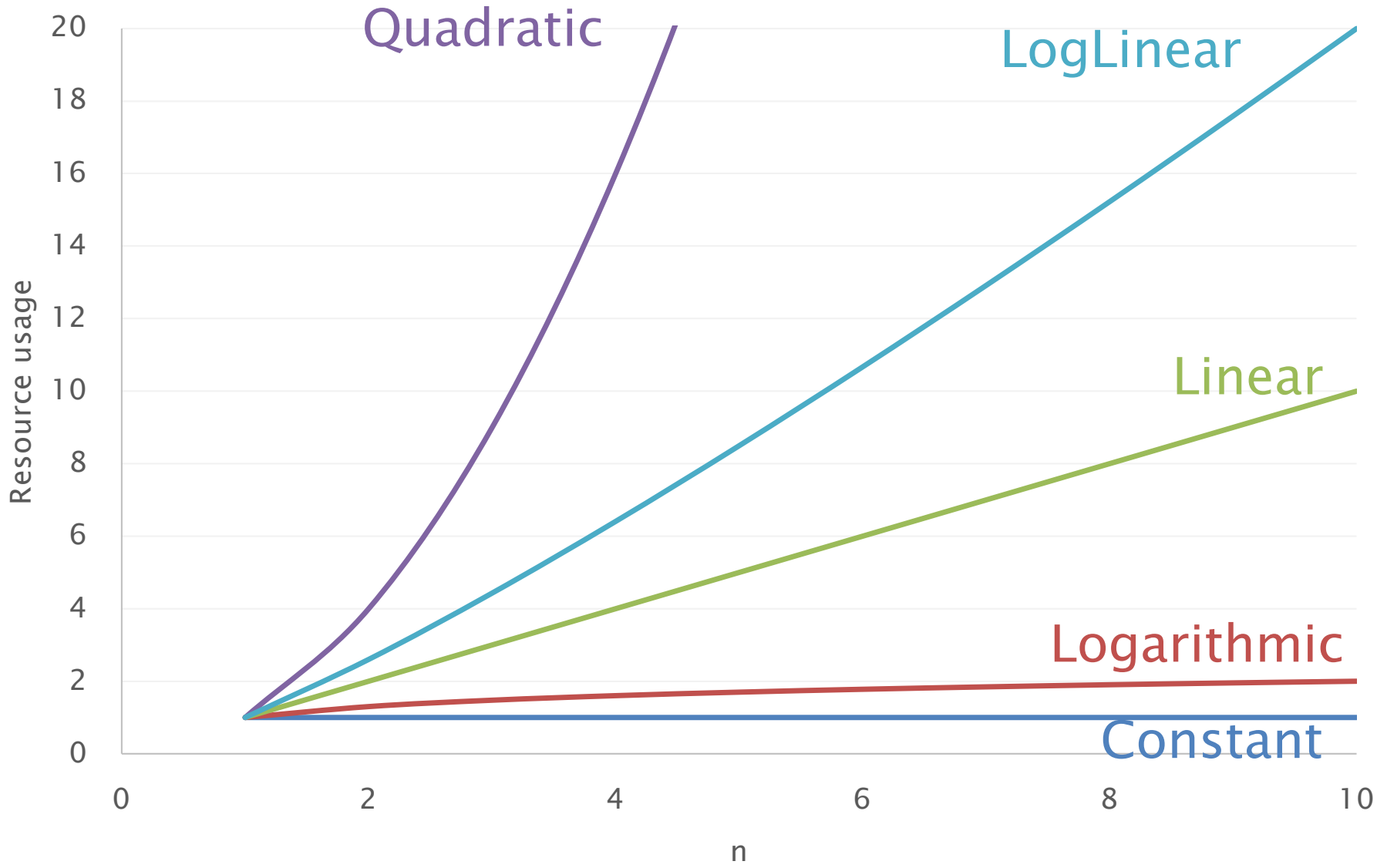
Selecting the container type

- If access by key is needed use a **`Map`**
  - If values sorted by key use a **`SortedMap`**
- Otherwise use a **`Collection`**
  - If indexed access, use a **`List`**
    - Class depends on expected typical operation
  - If access in order, use a **`Queue`**
  - If no duplicates, use a **`Set`**
    - If elements sorted, use a **`SortedSet`**

Efficiency

- Time and Space
- Computed as a function of the number (n) of elements contained
  - Constant: independent of n
  - Logarithmic: grows as $log(n)$
  - Linear: grows proportionally to n
  - Loglinear: grows as $n\ log(n)$
  - Quadratic: grows proportionally to n^2

Efficiency


Quadratic
LogLinear
Linear
Logarithmic
Constant
Resource usage
n
0
2
4
6
8
10
12
14
16
18
20

`List` implementations

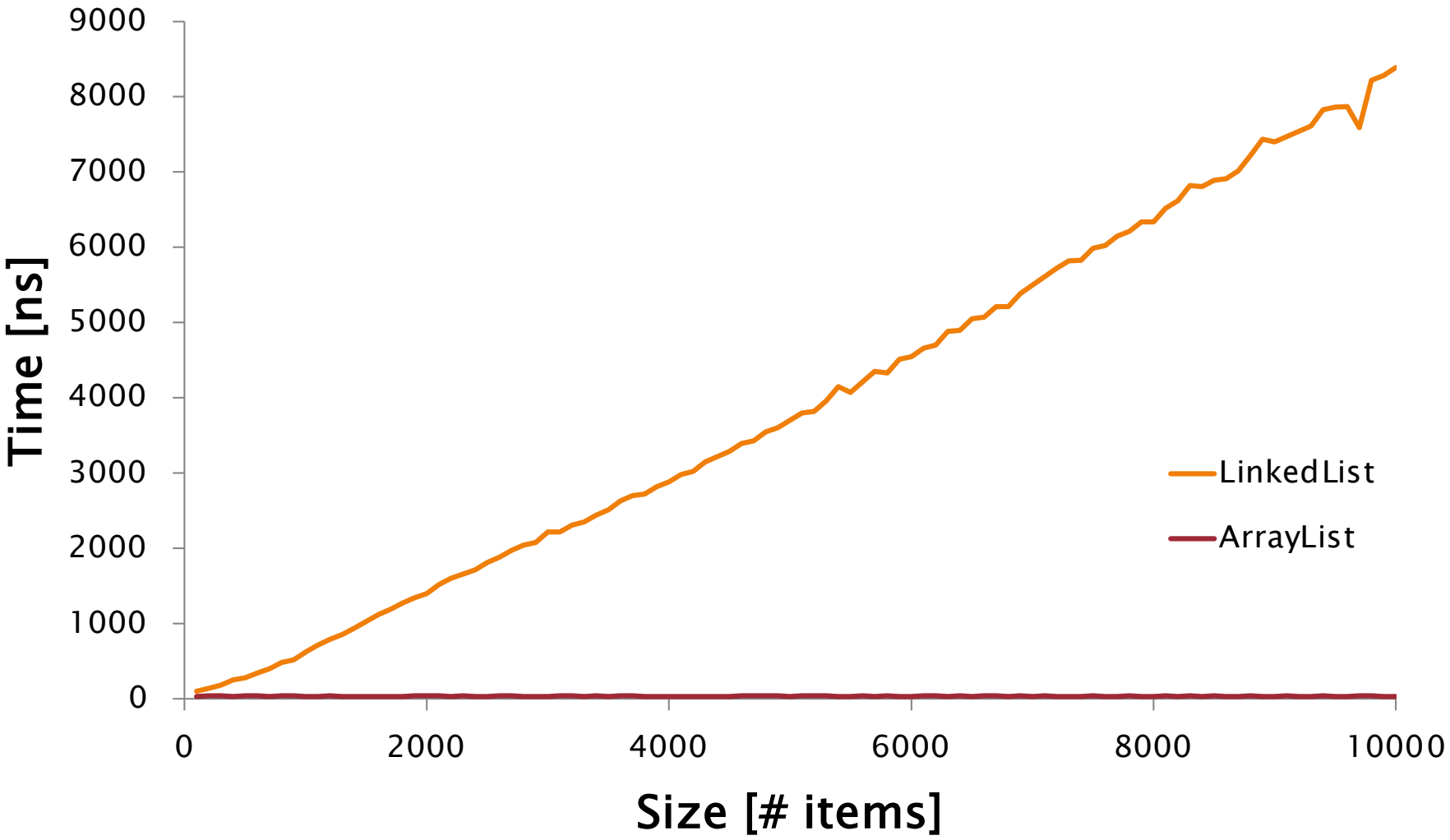
`ArrayList`

- **`get(n)`**
  - Constant
- **`add(0,…)`**
  - Linear
- **`add()`**
  - Constant

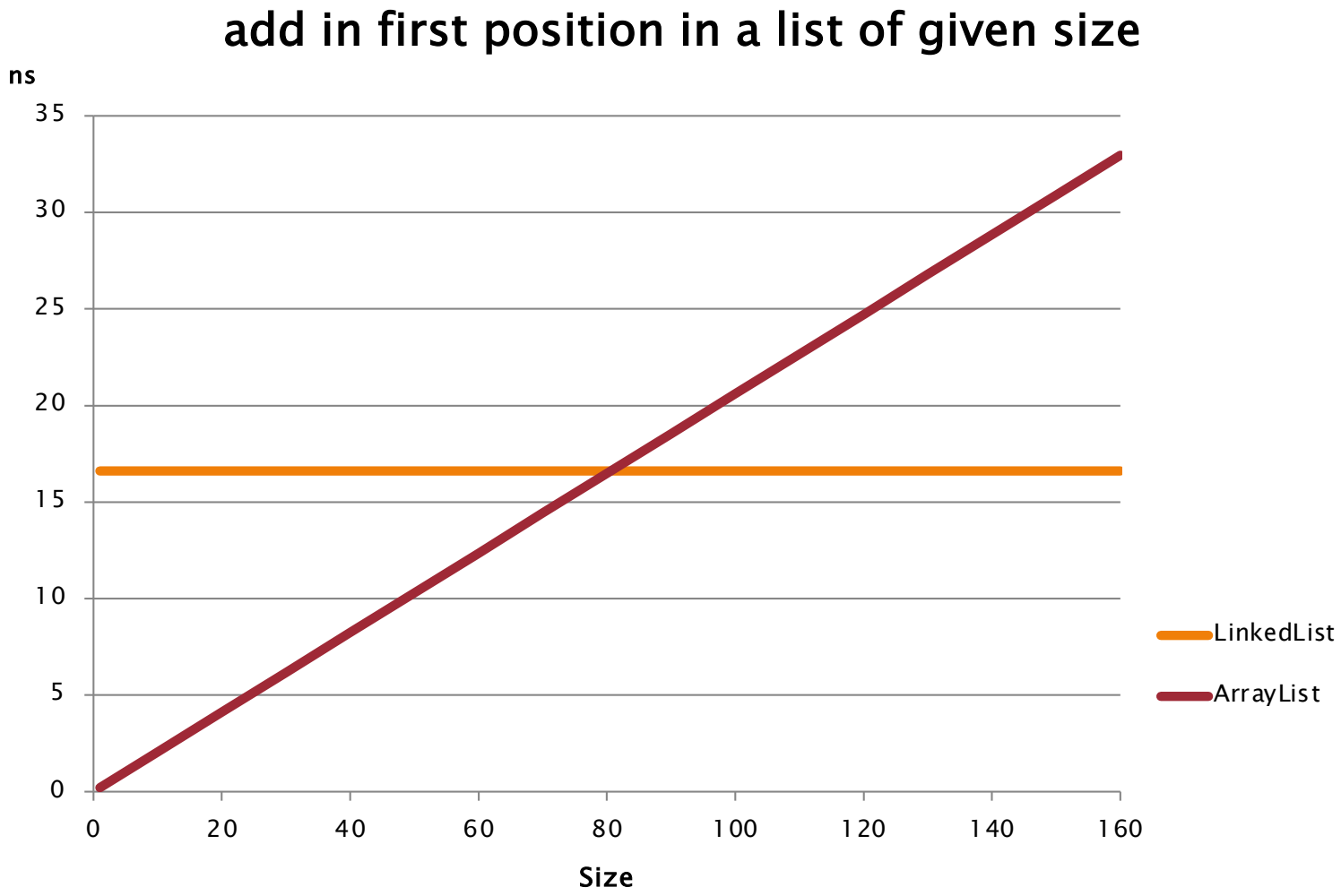
`LinkedList`

- **`get(n)`**
  - Linear
- **`add(0, … )`**
  - Constant
- **`add()`**
  - Constant

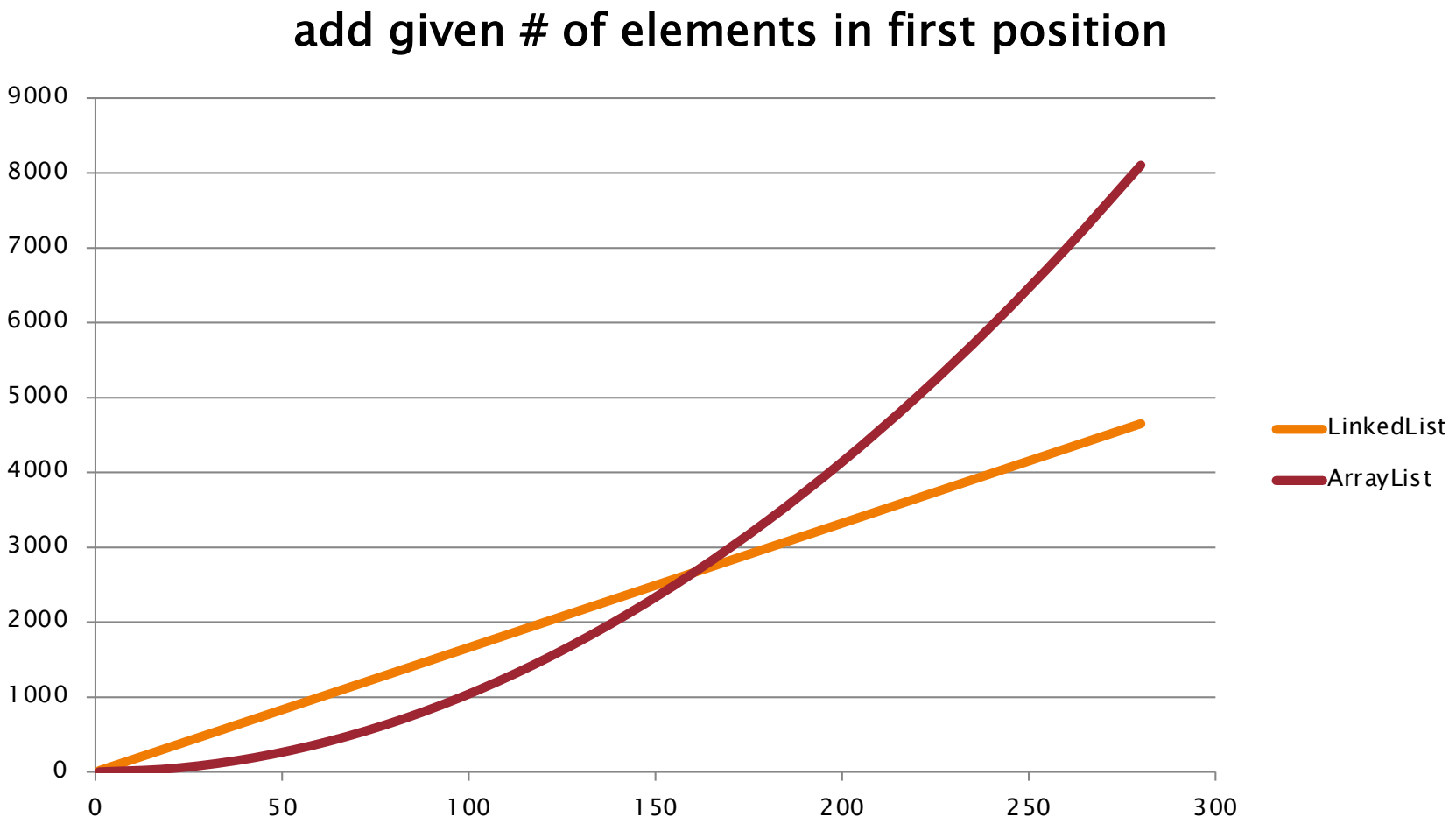
`List` implementations – Get



`List` Implementations – Add



`List` Implementations – Add



`List` implementation – Models

	`LinkedList`	`ArrayList`
Add in first pos. in list of size *n*	$t(n) = C_L$	$t(n) = n \cdot C_A$
Add *n* elements	$t(n) = n \cdot C_L$	$t(n) = \sum_{i=1}^{n} C_A \cdot i$ $= \frac{C_A}{2} n \cdot (n-1)$

C_L = 16.0 ns

C_A = 0.2 ns

Using maps – getting an item

```
String val = map.get(key);
if( val == null ) val = "Undefined";
```

- Or

```
if( ! map.containsKey(key))
    val = "Undefined";
else
    val = map.get(key);
```

- Or

```
String val = map.getOrDefault(key,
                              "Undefined");
```

Using maps

- Updating entries
  - E.g. counting frequencies

```
Map<String,Integer> wc=new XMap<>();
for(String w : words) {
   Integer i= wc.get(w);
   wc.put(w, i==null?1:i+1);
}
```

Using maps

- Updating entries
  - E.g. counting frequencies

```
Map<String,Integer> wc=new XMap<>();
for(String w : words) {
  wc.compute(w,(k,v)->v==null?1:v+1);
}
```

Autoboxing hides memory fee of 16 bytes per increment due to object creation: `Integer.valueOf(v.intValue()+1)`

Using maps

```
class Counter {
  int i=0;
  public String
  toString(){
   return ":"+i; }
}
```

- Updating entries
  - E.g. counting frequencies

```
Map<String,Counter> wc=new XMap<>();
for(String w : words) {
  wc.computeIfAbsent(w,
                     k->new Counter()).i++;
}
```

~40% faster than with `Integer`
– 16 bytes per each increment

Using maps

- Keeping items sorted
  - Using sorted maps

```
SortedMap<…> wc=new TreeMap<>();
```

  - “A”=1, “All”=3, “And"=2, “Barefoot”=1,…

```
Map<…> wc=new HashMap<>();
```

  - “reason”=1, “been”=1, “spoke”=1, “let”=1

Search efficiency

- Example:
  - 100k searches in a container require

size	HashMap	TreeMap	ArrayList	LinkedList
100k	3ms	60ms	40s	>1h
200k	3ms	65ms	110s	

ALGORITHMS

Algorithms

- Static methods of `java.util.Collections`
  - Work on List since it has the concept of position
- `sort`() – merge sort of List, $n \log(n)$
- `binarySearch`() – requires ordered sequence
- `shuffle`() – unsort
- `reverse`() – requires ordered sequence
- `rotate`() – of given a distance
- `min`(), `max`() – in a Collection

`sort()` method

- Operates on `List<T>`
  - Needs access by index to sort
- Two variants:

```
<T extends Comparable<? super T>>
void sort(List<T> list)

<T> void sort(List<T> list,
              Comparator<? super T> cmp)
```

Sort generic

```
T extends Comparable<? super T>
```

MasterStudent Student MasterStudent

- Why `<? super T>` instead of just `<T>` ?
  - Suppose you define
    - `MasterStudent extends Student { }`
  - Intending to inherit the Student ordering
    - It does not implement `Comparable<MasterStudent>`
    - But `MasterStudent` extends (indirectly) `Comparable<Student>`

Search

- `<T> int binarySearch(List<? extends Comparable<? super T>> l, T key)`
  - Searches the specified object
  - List must be sorted into ascending order according to natural ordering
- `<T> int binarySearch(List<? extends T> l, T key, Comparator<? super T> c)`
  - Searches the specified object
  - List must be sorted into ascending order according to the specified comparator

Wrap-up

- The collections framework includes interfaces and classes for containers
- There are two main families
  - Group containers
  - Associative containers
- All the components of the framework are defined as generic types