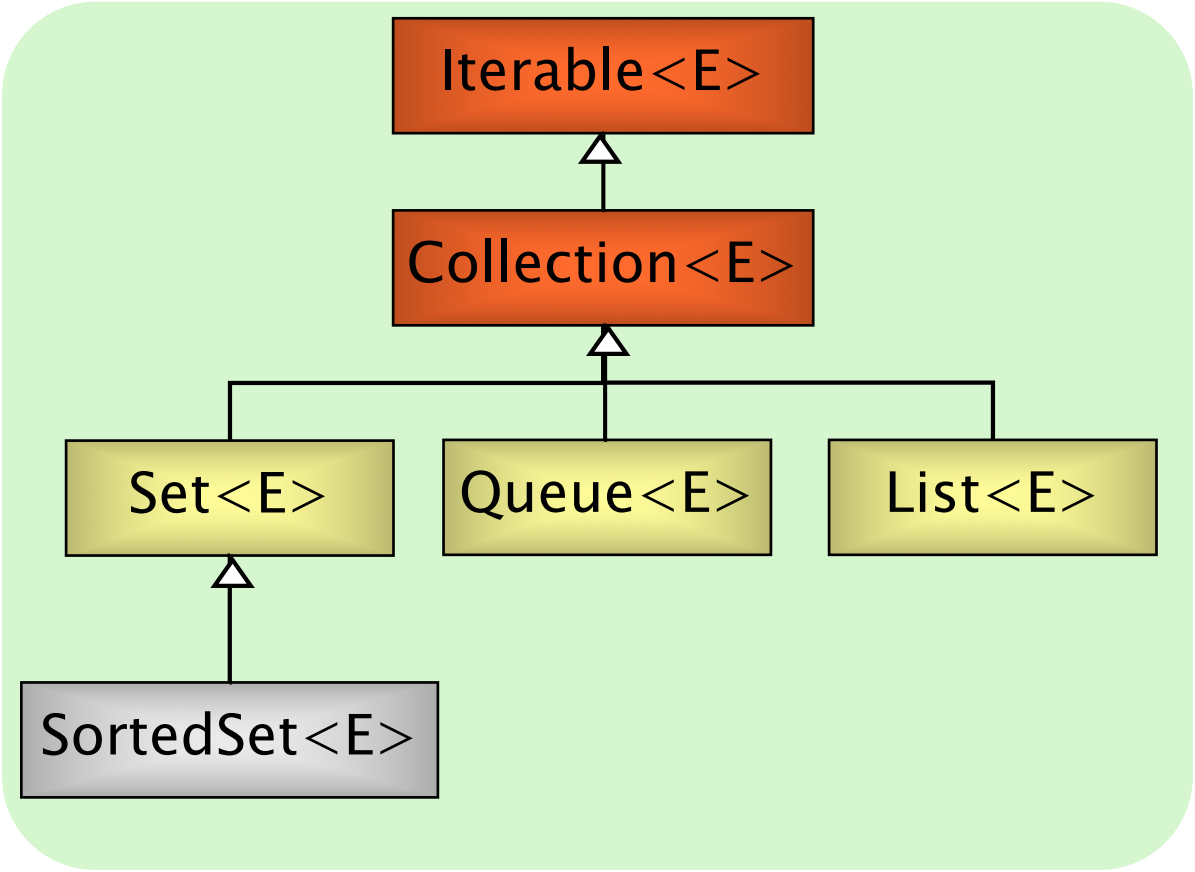
Java Collection Framework

SOftEng
http://softeng.polito.it

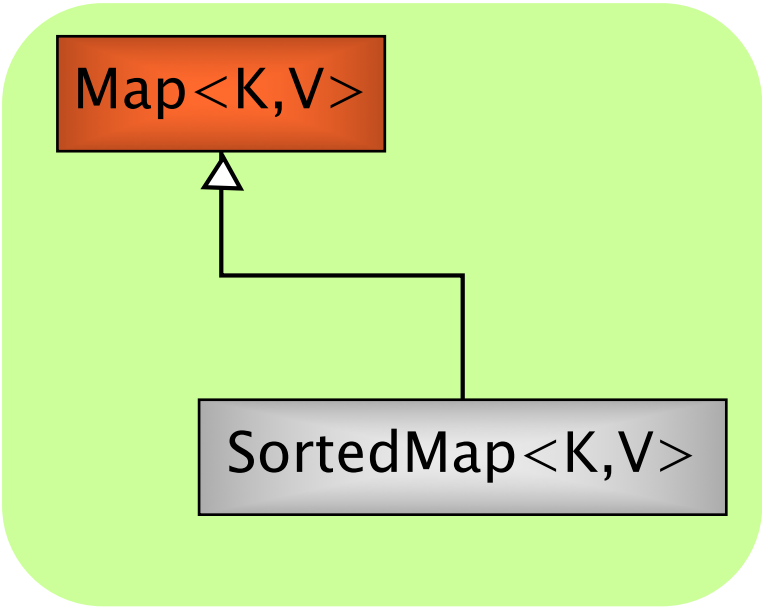
Framework

- Interfaces (ADT, Abstract Data Types)
- Implementations (of ADT)
- Algorithms (sort)
- Contained in package java.util.*
- Originally using Object, since Java 5 redefined as generic

Interfaces …

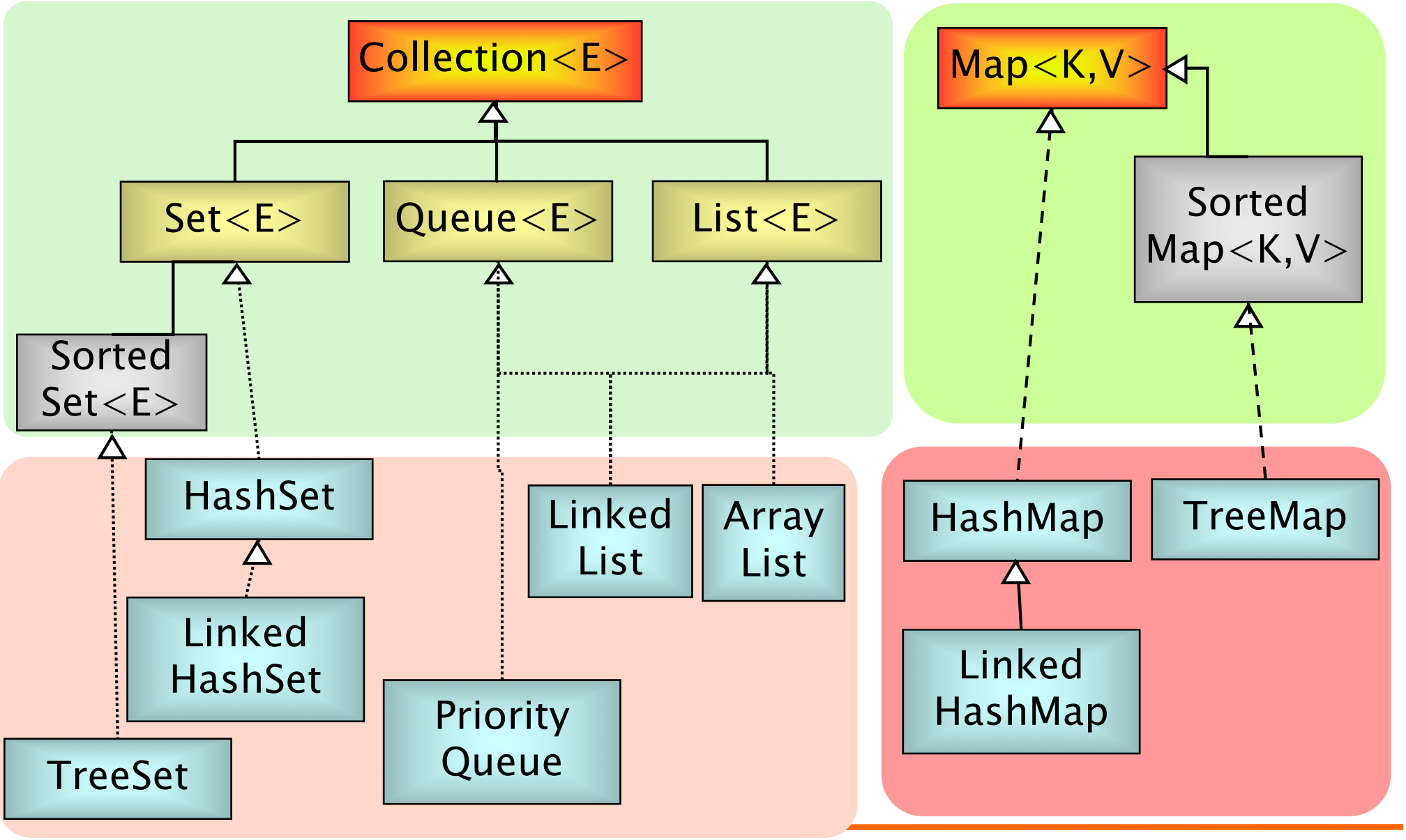


Group containers

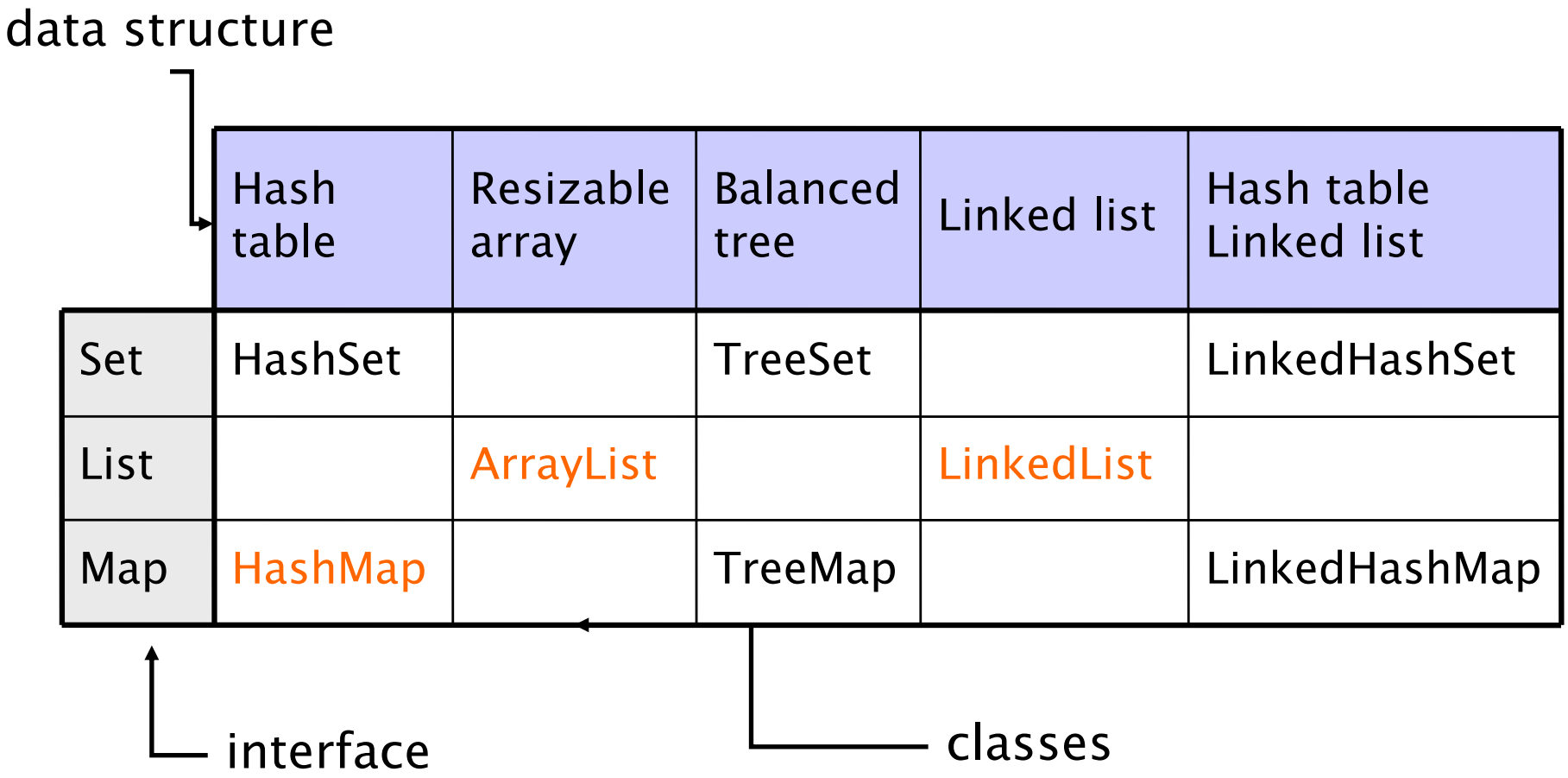


Associative containers

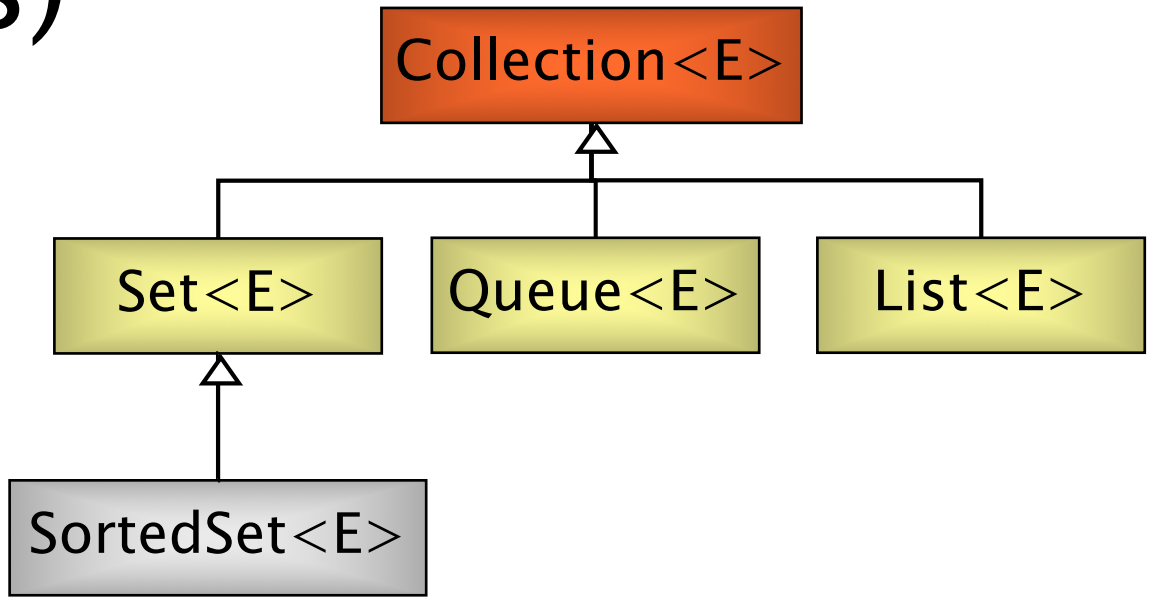
… and implementations



Internals


data structure
Hash table
Resizable array
Balanced tree
Linked list
Hash table Linked list
Set
HashSet
TreeSet
LinkedHashSet
List
ArrayList
LinkedList
Map
HashMap
TreeMap
LinkedHashMap
interface
classes

Group containers (Collections)



Collection

- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Two constructors common to all classes implementing the interface Collection
  - T()
  - T(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()`

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();
System.out.println( array[0] );
```

List

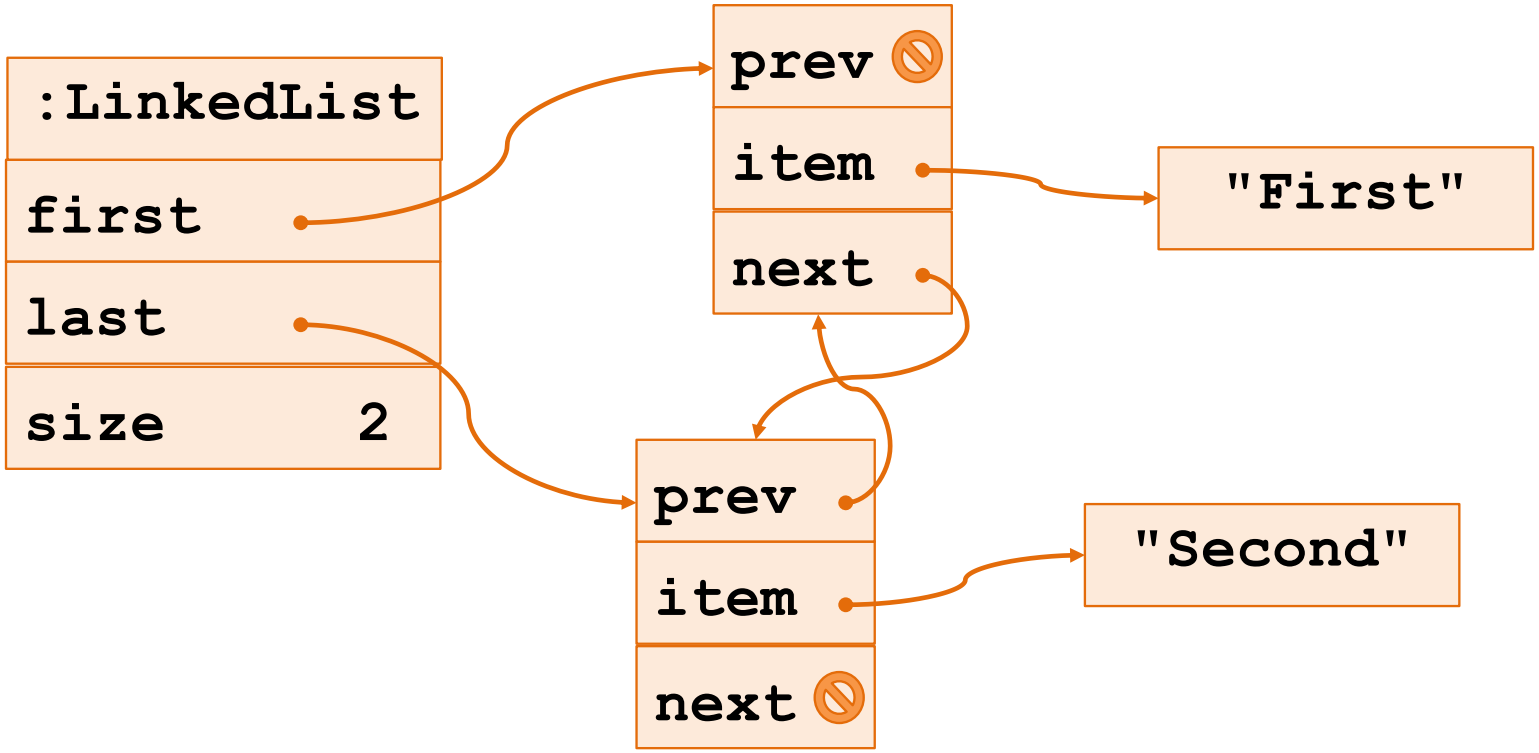
- Can contain duplicate elements
- Insertion order is preserved
- User can define insertion point
- Elements can be accessed by position
- Augments the Collection interface

List interface: further methods

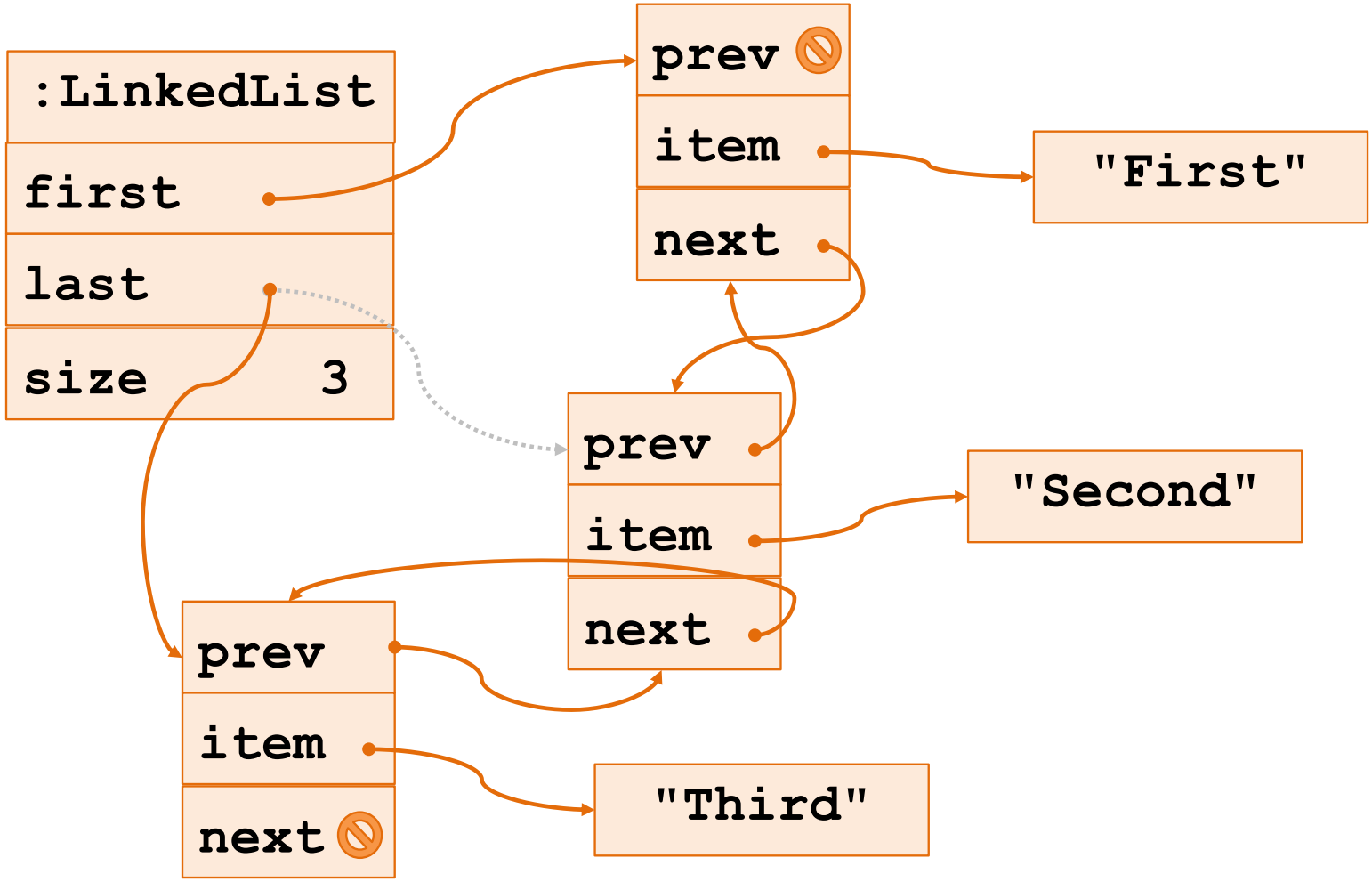
- `E get(int index)`
- `E set(int index, E element)`
- `void add(int index, E element)`
- `E remove(int index)`

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

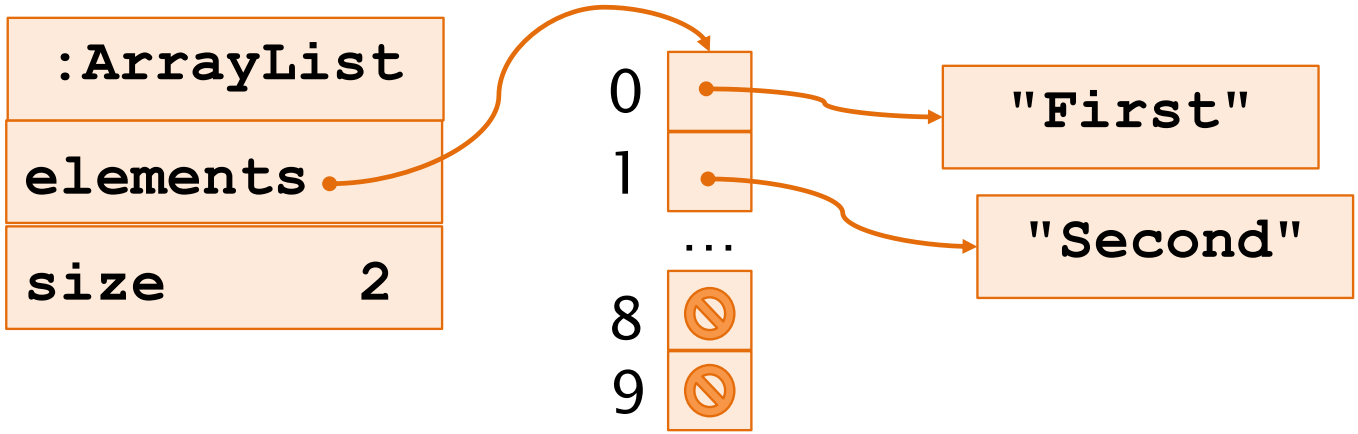
LinkedList


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

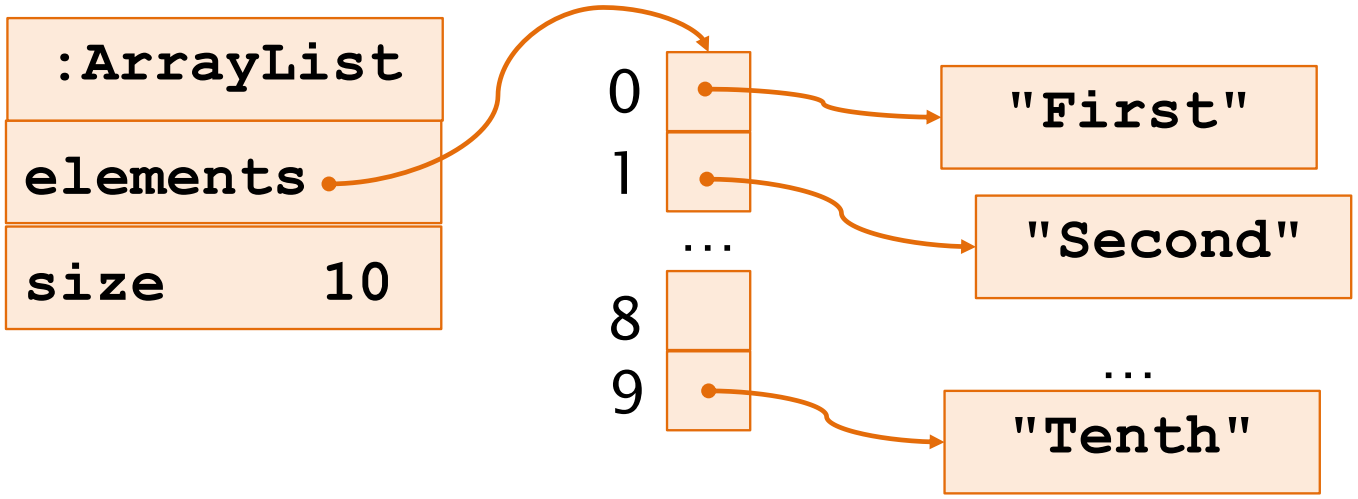
LinkedList


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

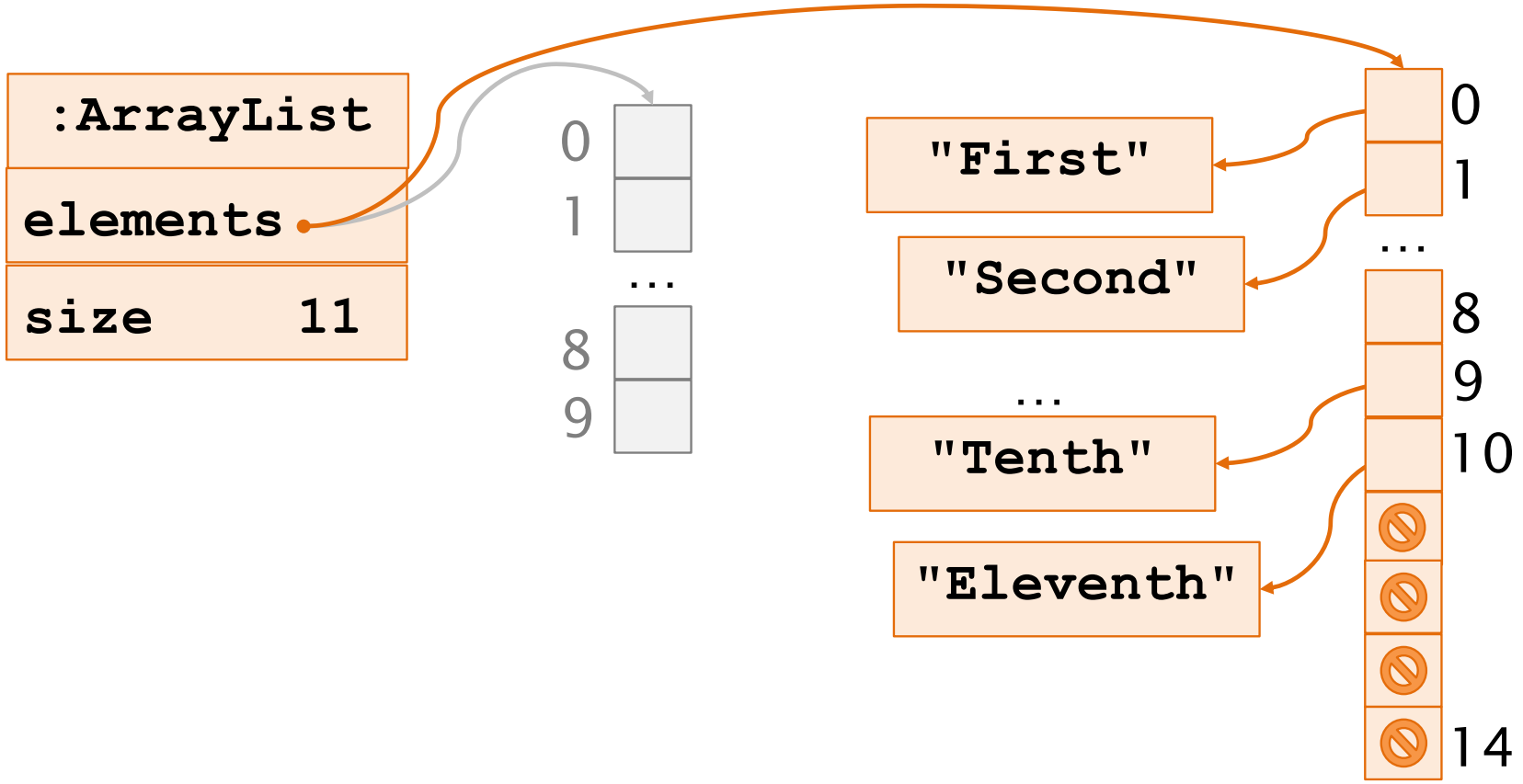
ArrayList



ArrayList



ArrayList


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

List implementations

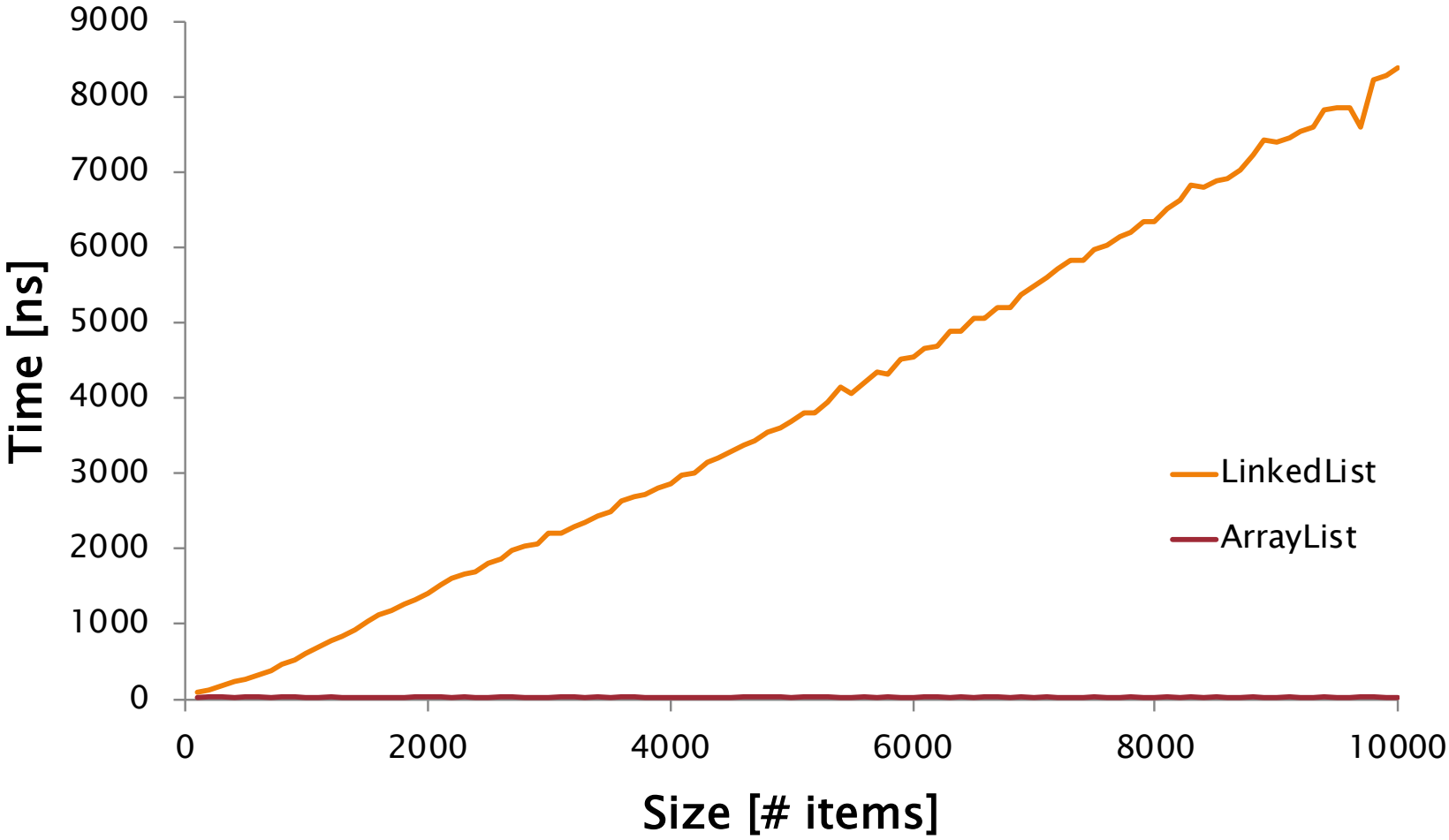
ArrayList

- `get(n)`
  - Constant
- Insert/add (beginning) and delete while iterating
  - Linear

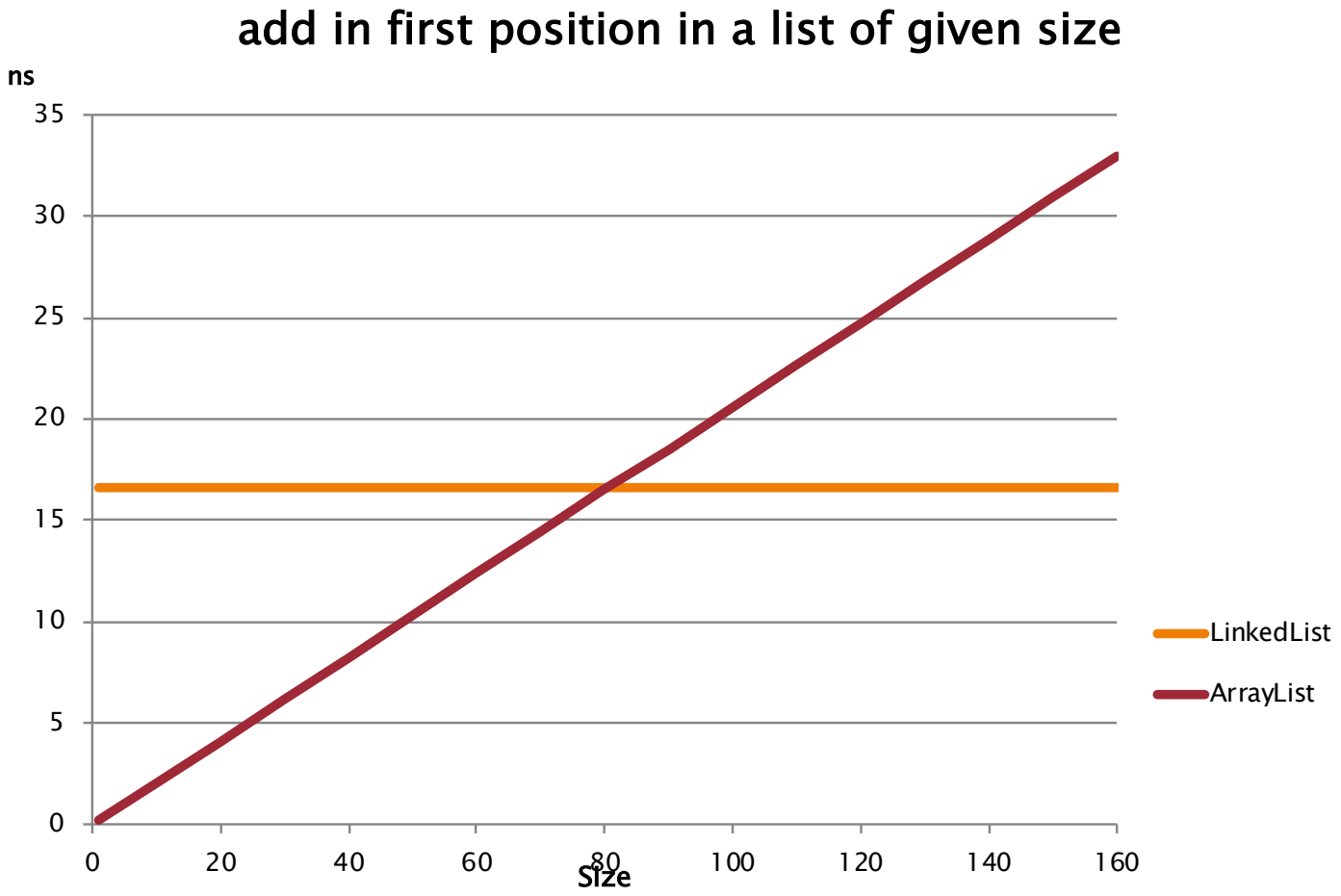
LinkedList

- `get(n)`
  - Linear
- Insert/add (beginning) and delete while iterating
  - Constant

List implementations – Get



List implementations



List implementations

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

Example I

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

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

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

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

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 III

```
List l = new ArrayList(2); // 2 refs to null

l.add(new Integer(11));    // 11 in position 0
l.add(0, new Integer(13)); // 11 in position 1
l.set(0, new Integer(20)); // 13 replaced by 20

l.add(9, new Integer(30)); // NO: out of bounds
l.add(new Integer(30));    // OK, size extended
```

Queue

- Collection whose elements have an order
  - Not and ordered collection though
- Defines a head position where is the first element that can be accessed
  - **`peek()`**
    - Return the element, without removing it
  - **`poll()`**
    - Returns the element and removes it

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

- Contains no methods other than those inherited from Collection
- **`add()`** has restriction that no duplicate elements are allowed
- The elements are traversed in no particular order

SortedSet

- No duplicate elements allowed
- The elements are traversed according to the natural ordering (ascending)
- Augments the 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 according to the insertion order
- TreeSet implements SortedSet
  - R-B trees as internal data structure (computationally expensive)

Iterators

SOftEng
http://softeng.polito.it

Iterating on a Collection

- A common operation with collections is to iterate over their elements
- Collection extends Iterable, an interface describing a container of elements that can be iterated upon
- A container implementing Iterable provides a single method that returns an Iterator
  - `Iterator<E> iterator()`

Iterators and iteration

- 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 interface

- 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

```
Collection<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

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

Iterator examples (until Java 1.4)

Print all objects in a list

```
Collection persons = new LinkedList();
…
for(Iterator i= persons.iterator(); i.hasNext(); ) {
    Person p = (Person)i.next();
    …
}
```

Note well

- It is unsafe to iterate over a collection while modifying (add/remove) at the same time
- Except if using the iterator's own methods
  - `Iterator.remove()`
  - `ListIterator.add()`

Delete

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

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

Add

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

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

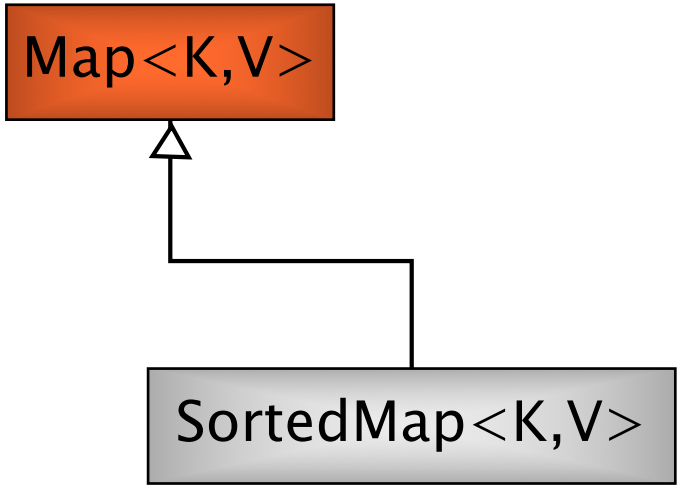
ConcurrentModificationException

Add (cont'd)

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

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

Associative containers (Maps)



SOftEng
http://softeng.polito.it

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
- Two constructors common to all classes implementing the interface Collection
  - `T()`
  - `T(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

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


Person bob = people.get("RBTGRN");
if( bob == null )
  System.out.println( "Not found" );


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

SortedMap interface

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

Map implementations

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

HashMap

- A.k.a. hash table, a data structure that maps keys to values by using a *hash function* to compute an *index*, also called a *hash code*, into an array of buckets or slots the given element is recorded into
- Collisions when same index assigned to different keys (solved, e.g., using linked lists)

HashMap

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

Using HashMap

```
Map<String,Student> students =
        new HashMap<String,Student>();

students.put("123",
         new Student("123","Joe Smith"));

Student s = students.get("123");

for(Student si: students.values()){

}
```

TreeMap

- Get/put takes log time
- Based on a Red-Black tree
- Keys are maintained and will be traversed in order
- Constructor optional arguments
  - Comparator to replace the natural order of keys

Iteration examples

Print all values in a map (variant using while)

```
Map<String,Person> people =
                new HashMap<String,Person>();
…
Collection<Person> values = people.values();

for(Person p: values) {
   System.out.println(p);
}
```

Iteration examples

Print all keys AND values in a map

```
Map<String,Person> people =
                new HashMap<String,Person>();
…
Collection<String> keys = people.keySet();
for(String ssn: keys) {
    Person p = people.get(ssn);
    System.out.println(ssn + " - " + p);
}
```

Iteration examples (until Java 1.4)

Print all values in a map (variant using while)

```
Map people = new HashMap();
…
Collection values = people.values();
Iterator i = values.iterator();
while( i.hasNext() ) {
    Person p = (Person)i.next();
    …
}
```

Iteration examples (until Java 1.4)

Print all keys AND values in a map

```
Map people = new HashMap();
…
Collection keys = people.keySet();
for(Iterator i= keys.iterator(); i.hasNext();) {
    String ssn = (String)i.next();
    Person p = (Person)people.get(ssn);
    …
}
```

Ordering

SOftEng
http://softeng.polito.it

Comparable interface

```
public interface Comparable<T> {
    public int compareTo(T obj);
}
```

- Defined in java.util
- Compares the receiving object (this) with the specified (passed) object
- Return value must be:
  - <0 if ***this*** precedes ***obj***
  - $==0$ if ***this*** has the same order as ***obj***
  - >0 if ***this*** follows ***obj***

Comparable interface

- The interface is implemented by language common types in packages java.lang and java.util
  - String objects are lexicographically ordered
  - Date objects are chronologically ordered
  - Number and sub-classes are ordered numerically

Ordering “the old way”

- Before the introduction of Generics
  - `public int compareTo(Object obj)`
- No control on types
- A cast had to be performed within the method
  - Possible ClassCastException when comparing objects of unrelated types

Ordering “the old way”

```
public int compareTo(Object obj){

Student s = (Student)obj;

  int cmp = lastName.compareTo(s.lastName);

  if(cmp!=0)
    return cmp;
  else
   return firstName.compareTo(s.firstName);
}
```

possible run-time error

Custom ordering with Generics

- Example: how to define an ordering upon Student objects according to the "natural alphabetic order"

```
public class Student
   implements Comparable<Student>{
    private String first;
    private String last;
    public int compareTo(Student o){
        ...
    }
}
```

Custom ordering with Generics

```
public int compareTo(Student o){

   int cmp = lastName.compareTo(s.lastName);

   if(cmp!=0)
     return cmp;
   else
    return firstName.compareTo(s.firstName);
}
```

Custom ordering (alternative)

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

- Defined in java.util
- Compares its two arguments
- Return value must be
  - <0 if o1 precedes o2
  - ==0 if o1 has the same ordering as o2
  - >0 if o1 follows o2

Custom ordering (alternative)

```
class StudentIDComparator
        implements Comparator<Student> {

  public int compare(Student s1, Student s2){
     return s1.getID() - s2.getID();
  }
}
```

- Used to define alternative orderings to Comparable (e.g., order same objects in two different ways)
- The “old way” version compares Object refs.

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

Algorithms

SOftEng
http://softeng.polito.it

Algorithms

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

sort()

- Two generic overloads:
  - on Comparable objects

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

  - using a Comparator object

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

Custom ordering w/ Comparator

```
List students = new LinkedList();

students.add(new Student("Mary","Smith",34621));
students.add(new Student("Alice","Knight",13985));
students.add(new Student("Joe","Smith",95635));

Collections.sort(students); // sort by name

Collections.sort(students,
new StudentIDComparator()); // sort by ID
```

Search

- `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
- `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

Algorithms – Arrays

- Static methods of java.util.Arrays class
- Work on object arrays
  - `sort()`
  - `binarySearch()`

Search – Arrays

- `int binarySearch(Object[] a, Object key)`
  - Searches the specified object
  - Array must be sorted into ascending order according to natural ordering
- `int binarySearch(Object[] a, Object key, Comparator c)`
  - Searches the specified object
  - Array 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