Java Collections Framework (JCF)

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)

Framework

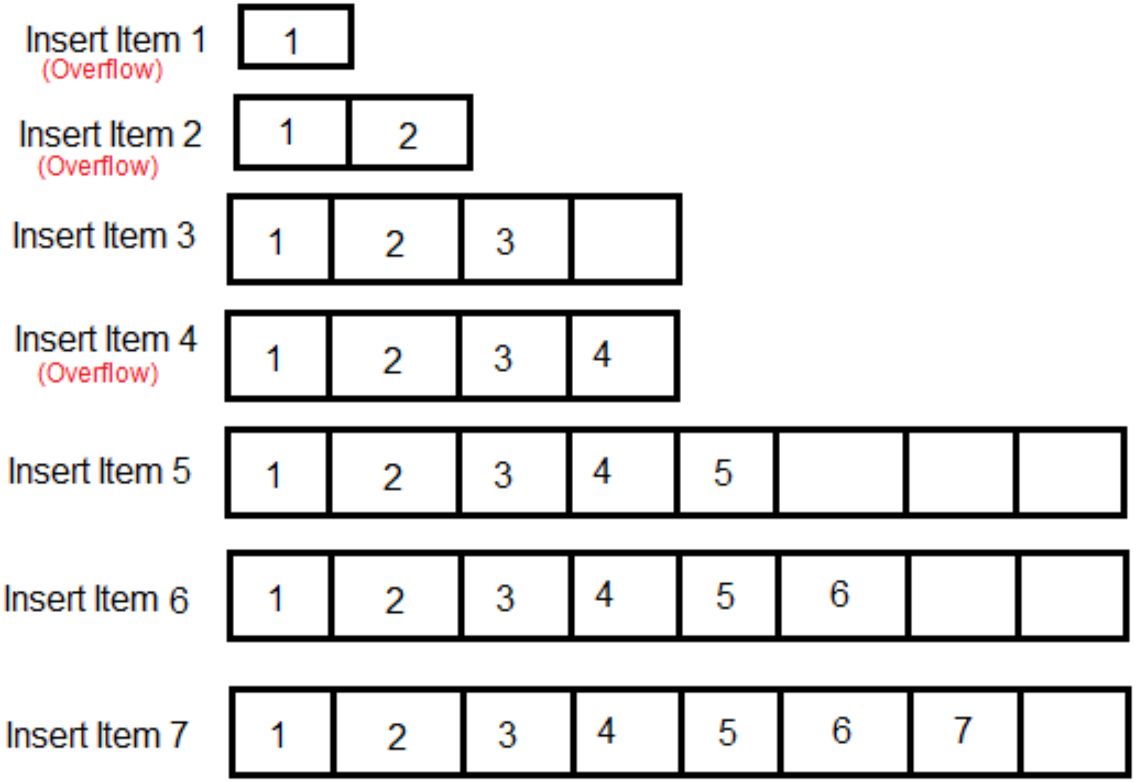
- The Java Collection Framework (JCF) is a set of classes and interfaces implementing commonly reusable data structures.
- The JCF provides both interfaces defining main functionalities; and classes implementing them.
  - Interfaces (Abstract Data Types)
  - Implementations (of ADT)
  - Algorithms (java.util.Collections)
  - java.util.*

Key Concepts

- Resizable Array
- Linked List
- Balanced Tree
- Hash Table

Resizable Array ~O(n)

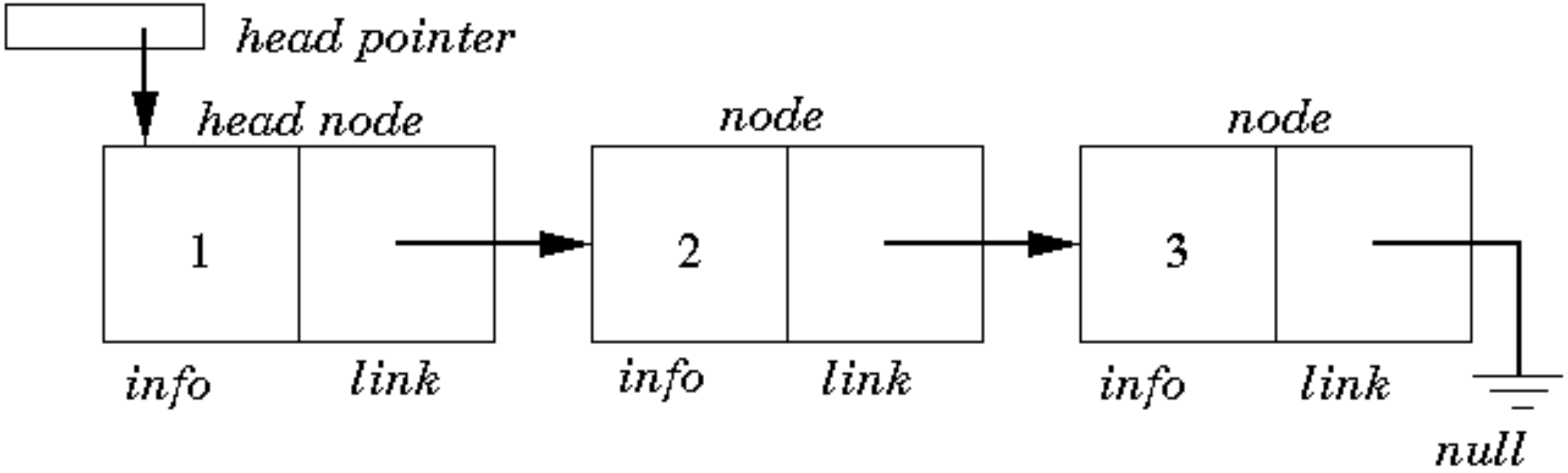
Initially table is empty and size is 0



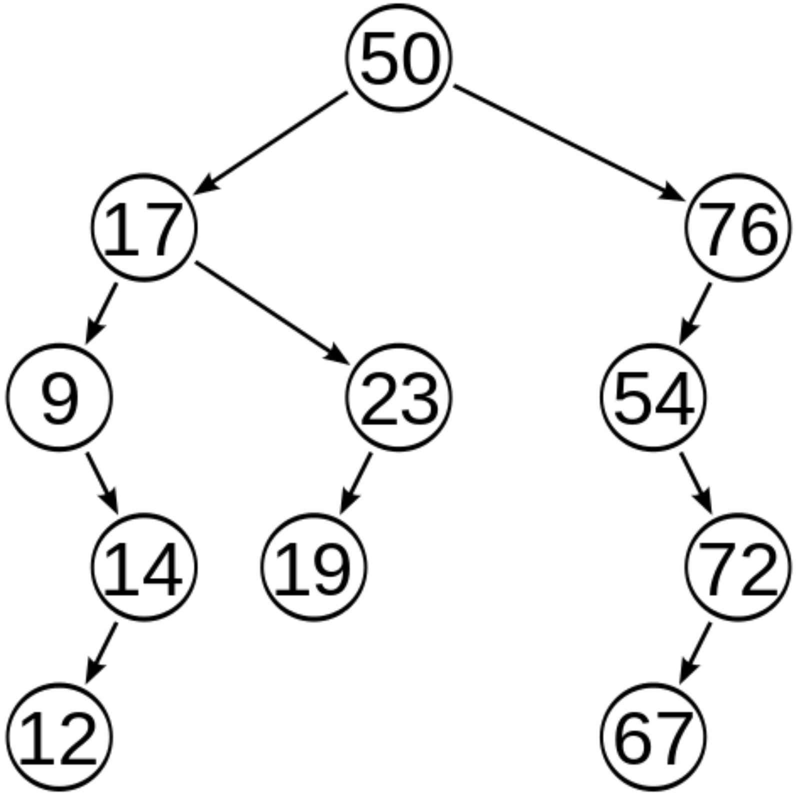
Insert Item 1 (Overflow)	1							
Insert Item 2 (Overflow)	1	2						
Insert Item 3	1	2	3					
Insert Item 4 (Overflow)	1	2	3	4				
Insert Item 5	1	2	3	4	5			
Insert Item 6	1	2	3	4	5	6		
Insert Item 7	1	2	3	4	5	6	7	

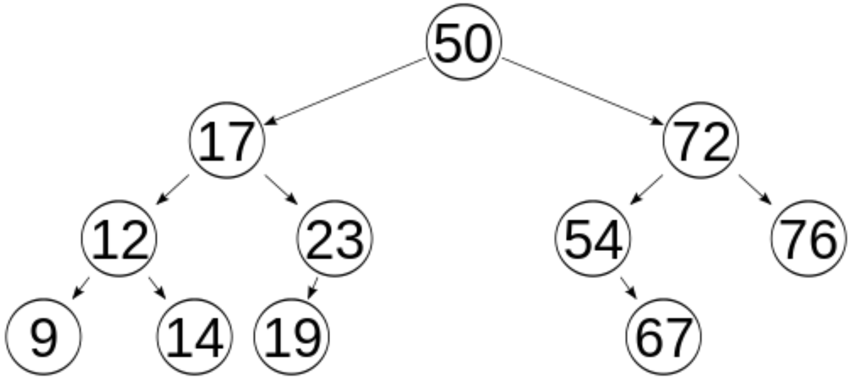
Next overflow would happen when we insert 9, table size would become 16

Linked List ~O(n)



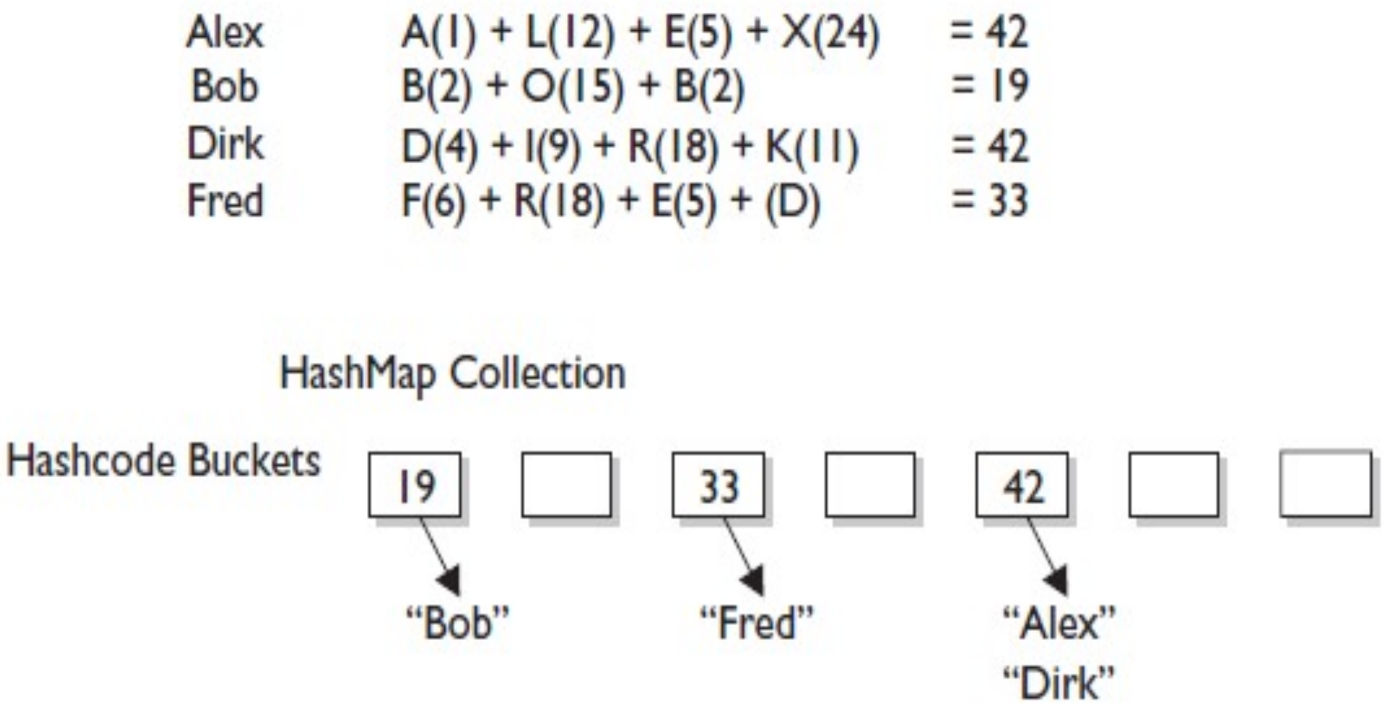
Balanced Tree ~O(log(n))





** A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1. A binary tree is balanced if for any two leaves the difference of the depth is at most 1.*

Hash Table ~O(1)



Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + (D)	= 33

HashMap Collection

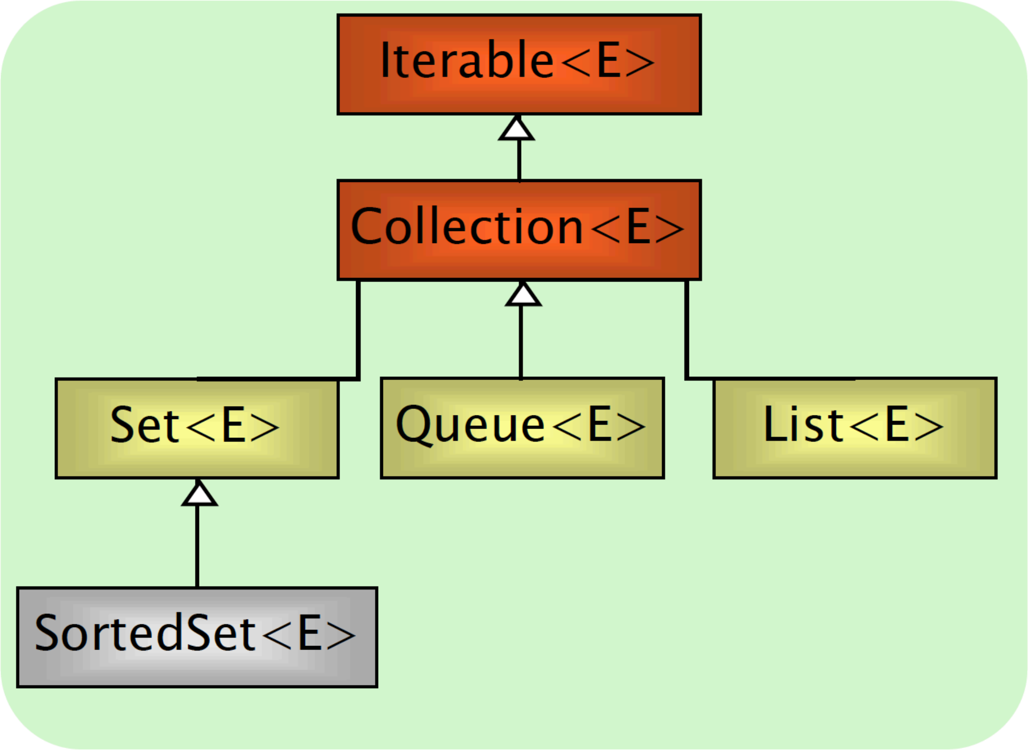
Hashcode Buckets

19 → "Bob"

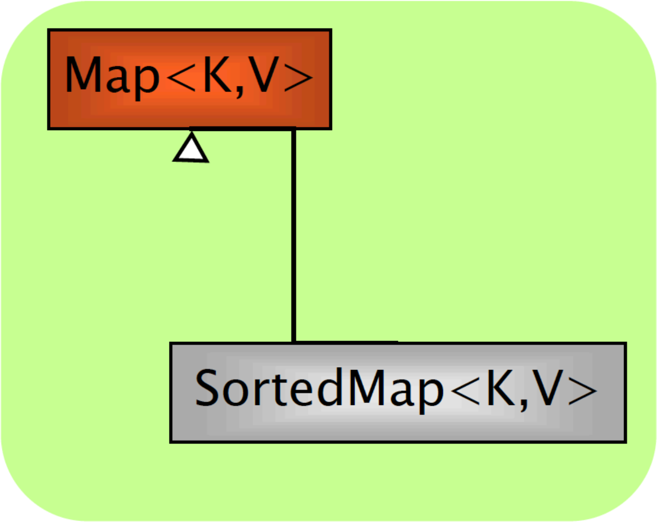
33 → "Fred"

42 → "Alex" "Dirk"

Interfaces

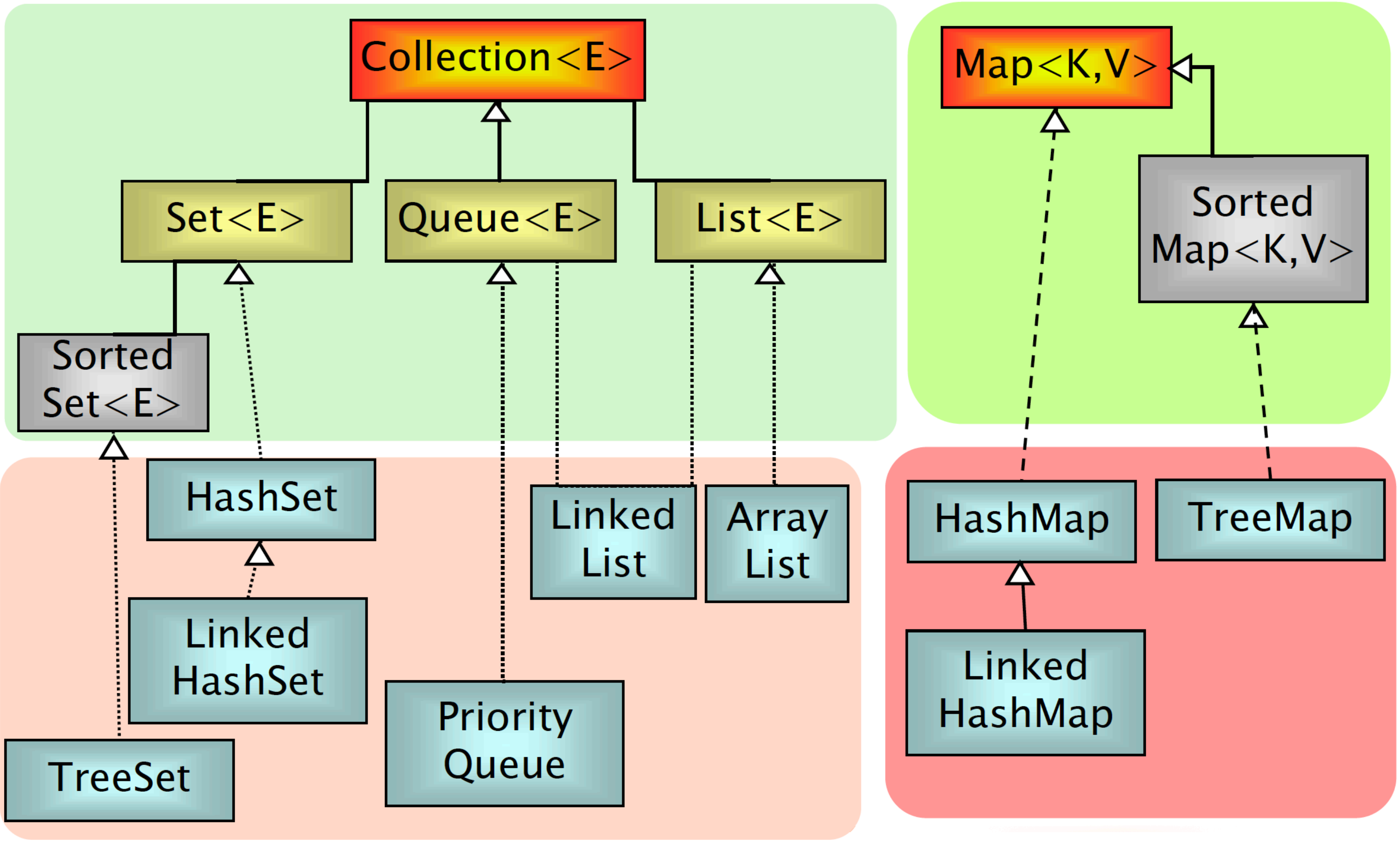


Group containers

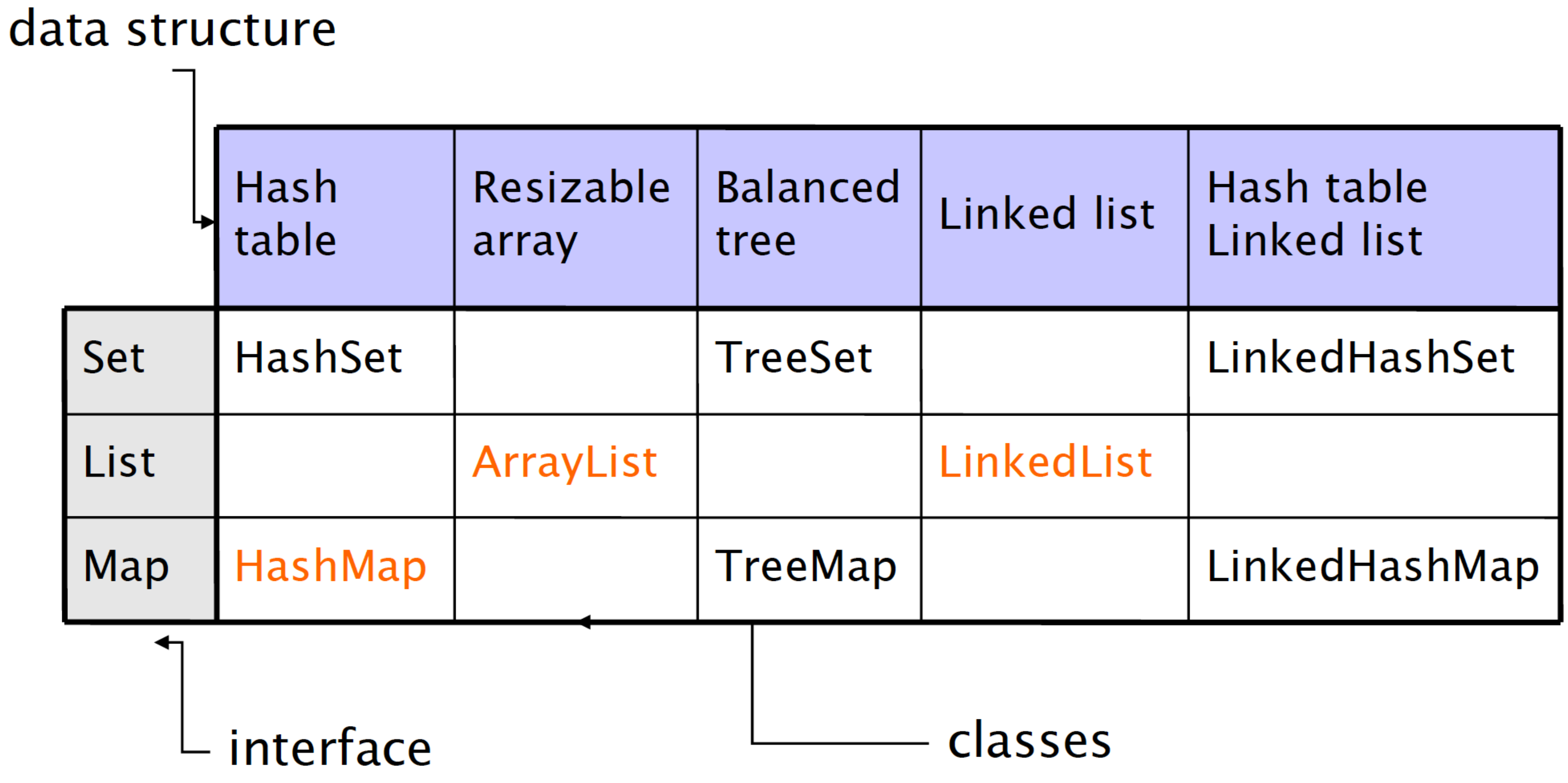


Associative containers

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

Iterable Interface

- The Iterable interface (java.lang.Iterable) is the root interface of the Java collection framework. The Collection interface extends Iterable, so all subtypes of Collection also implement the Iterable interface.
- Iterarable, literally, means that “can be iterated”. From a technical perspective, it means that an Iterator can be returned.
- Iterable objects can be used with the for-each loop:

```
List list = new ArrayList();
for(Object o : list){
    //do something;
}
```

- The Iterable interface has only one method:

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

Iterator Interface

- `boolean hasNext()`
- `object next()`
- `void remove()`

Collection Interface

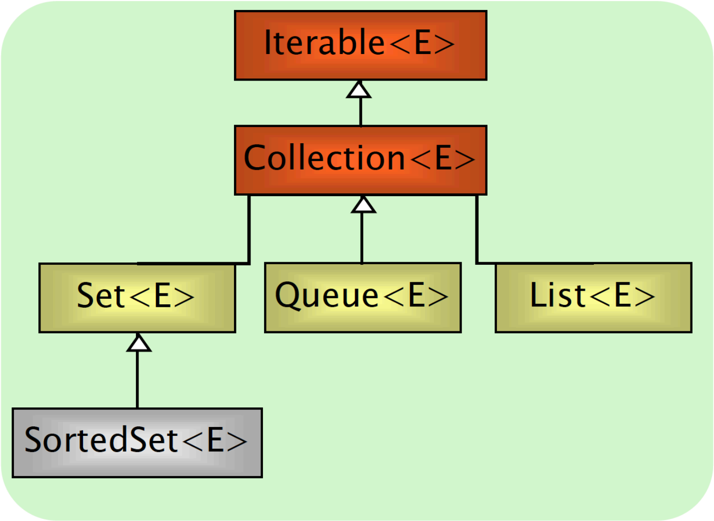
- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Following constructors are common to all classes implementing Collection
  - T()
  - T(Collection c)

Collection Interface

- int size()
- boolean isEmpty()
- boolean contains(Object element)
- boolean containsAll(Collection c)
- boolean add(Object element)
- boolean addAll(Collection c)
- boolean remove(Object element)
- boolean removeAll(Collection c)
- void clear()
- Object[] toArray()
- Iterator iterator()

List Interface

- Can contain duplicate elements
- Insertion order is preserved
- User can select arbitrary insertion points
- Elements can be accessed by position



List additional methods

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

- `boolean addAll(int index, Collection c)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `List subList(int fromIndex, int toIndex)`

List Implementations

- ArrayList
  - Get(n) -> Constant time
  - Insert (beginning) -> Linear time

- LinkedList
  - Get(n) -> Linear time
  - Insert (beginning) -> Constant time

ArrayList Example

```
List<Car> garage = new ArrayList<Car>();
garage.add(new Car());
garage.add(new ElectricCar());
garage.add(new ElectricCar());
garage.add(new Car());
for(int i; i < garage.size(); i++){
   Car c = garage.get(i);
   c.turnOn();
}
```

LinkedList Example

```
List<Car> garage = new LinkedList<Car>();
garage.add(new Car());
garage.add(new ElectricCar());
garage.add(new ElectricCar());
garage.add(new Car());
for(int i; i < garage.size(); i++){
   Car c = garage.get(i);
   c.turnOn();
}
```

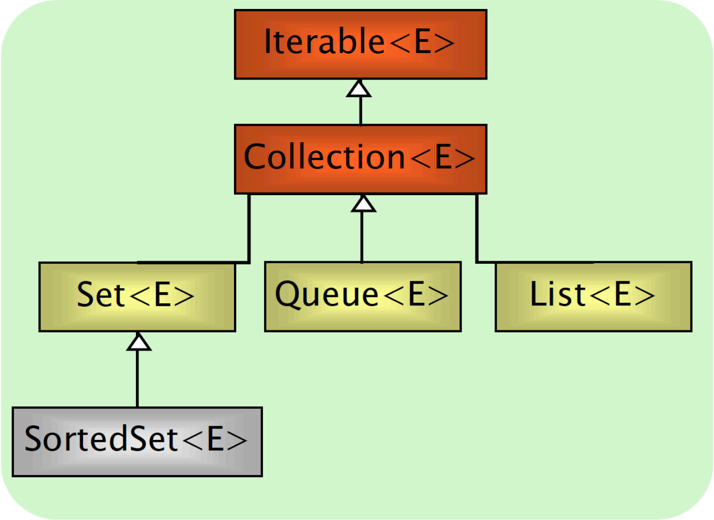
LinkedList Example

```
LinkedList<Car> garage = new LinkedList<Car>();
garage.add(new Car());
garage.add(new ElectricCar());

// LinkedList’s methods allowed!
garage.addFirst(new ElectricCar());
garage.addLast(new ElectricCar());
garage.getfirst();
garage.getLast();
garage.removeFirst();
garage.removeLast();
```

Queue Interface

- Collection whose elements have an order (*not and ordered collection!*)
- Defines a head (first element) and a tail (last element)
  - – peek(), retrieve but not removes!
  - – poll(), retrieves and removes!



Queue Implementations

- LinkedList
  - Insertion order
  - head is the first element of the list
  - FIFO internal policy: Fist-In-First-Out
- PriorityQueue
  - Natural ascending order

Queue Example

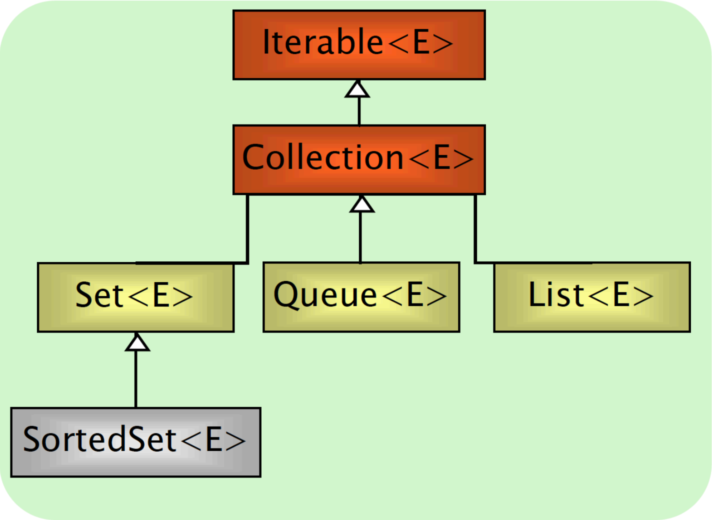
```
Queue<Integer> fifo = new LinkedList<Integer>();
Queue<Integer> pq = new PriorityQueue<Integer>();

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 other than those inherited from Collection
- **add()** has restriction that no duplicate elements are allowed
- **Iterator**: the elements are traversed in no particular order



Set Implementations

- HashSet implements Set
  - Hash tables as internal data structure (fast!)
  - No order
- LinkedHashSet extends HashSet
  - Insertion order
- TreeSet implements SortedSet
  - R-B trees as internal data structure (slow!)
  - Natural (ascending) order

TreeSet Internal Ordering

- Depending on the constructor used, SortedSet implementations can use different orderings
- TreeSet()
  - Natural ordering (elements must implement the Comparable Interface)
- TreeSet(Comparator c)
  - Ordering is according to the comparator rules, instead of natural ordering

HashSet Example

```
ArrayList<String> l = new ArrayList<String>();
l.add("Nicola"); l.add(”Agata");
l.add("Marzia"); l.add(”Agata");

HashSet<String> hs = new HashSet<String>(l);

System.out.println(l);
[Nicola, Agata, Marzia, Agata]
System.out.println(hs);
[Marzia, Nicola, Agata]
```

LinkedHashSet Example

```
ArrayList<String> l = new ArrayList<String>();
l.add("Nicola"); l.add(”Agata");
l.add("Marzia"); l.add(”Agata");

LinkedHashSet<String> lhs = new
LinkedHashSet<String>(l);

System.out.println(l);
[Nicola, Agata, Marzia, Agata]
System.out.println(lhs);
[Nicola, Agata, Marzia]
```

TreeSet Example

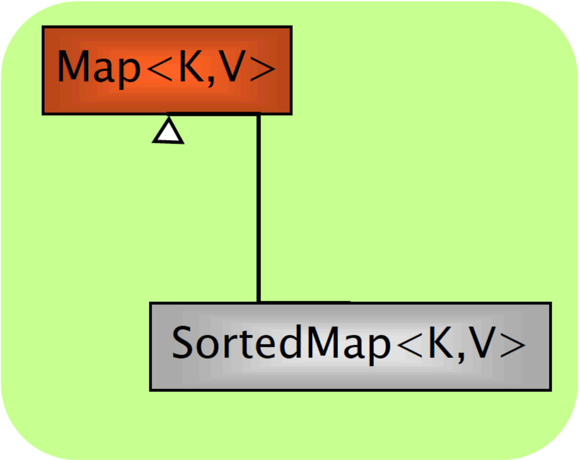
```
ArrayList<String> l = new ArrayList<String>();
l.add("Nicola"); l.add(“Agata");
l.add("Marzia"); l.add(“Agata");

TreeSet<String> ts = new TreeSet<String>(l);

System.out.println(l);
[Nicola, Agata, Marzia, Agata]
System.out.println(ts);
[Agata, Marzia, Nicola]
```

Map Interface

- An object storing pairs of (key, value) (e.g., Person => Phone number)
    - Keys and values must be objects
    - Keys must be unique
- Following constructors are common to all collection implementers
    - T()
    - T(Map m)



Map Interface

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

Map Implementations

- Similar to Set

- HashMap implements Map
  - No order
- LinkedHashMap extends HashMap
  - Insertion key order
- TreeMap implements SortedMap
  - Natural (ascending) key order

HashMap

- Get/set 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)

Map Example I

```
Map<String, Integer> m = new HashMap<String, Integer>();

m.put(“Agata”,  2);
m.put(“Marzia”, 3);
m.put(“Agata”,  4);
m.put(“Nicola”, 1);

System.out.println(m);
{Agata=4, Nicola=1, Marzia=3}
```

Map Example II

```
Map<String, Integer> m = new HashMap<String, Integer>();
…
// looping keys
List<String> keys = m.keySet();
for(String s : keys) {
    System.out.println(s + " -> " + m.get(s));
}

// contains key
If (m.containsKey(key)) {
    System.out.println(m.get(key));
}
```

Iterator

- 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 Interface

- `boolean hasNext()`
- `Object next()`
- `void remove()`

Iterator Example

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

/* C style */
for (int i = 0; i < pl.size; i++)
     System.out.println(pl.get(i))

/* Java style */
for (Person p : pl)
     System.out.println(p);

/* Iterator style */
for(Iterator<Person> i = pl.iterator(); i.hasNext();) {
     Person p = i.next();
     System.out.println(p);
}

/* While style */
Iterator i = pl.iterator();
while (i.hasNext())
     System.out.println((Person)i.next());
```

ListIterator

- An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
- A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

ListIterator Interface

- boolean hasNext()
- boolean hasPrevious()
- object next()
- object previous()
- void add()
- void set()
- void remove()
- int nextIndex()
- int previousIndex()

Caveat

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

- Unless you are using the iterator methods
  - **Iterator.remove()**
  - **ListIterator.add()**

Caveat

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

count = 0
for (Iterator<Integer> i = l.iterator(); itr.hasNext();){
    i.next();
    if (count++ == 1) l.remove(count);
    if (count++ == 2) l.add(new Integer(22));
    // Wrong! We modify the list while iterating
}
```

Caveat

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

count = 0
for (Iterator<Integer> i = l.iterator(); itr.hasNext();){
    i.next();
    if (count++ == 1) i.remove();
}
```

Caveat

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

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

The Comparable Interface

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

- Compares the receiving object with the specified object
- Return value must be:
  - < 0 if ***this*** precedes ***obj***
  - == 0 if ***this*** has the same order as ***obj***
  - > 0 if ***this*** follows ***obj***

The 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

The Comparable Interface

Given the following class:

```
class Person {
   protected String name;
   protected String surname;
   protected Integer name;
   …
}
```

The Comparable Interface

- How to define an ordering upon Person objects according to the “natural alphabetic order”

```
class Person implements Comparable<Person> {
   protected String firstName;
   protected String lastname;
   …
   public int compareTo(Student s) {…}
}
```

The Comparable Interface

```
public int compareTo(Student s) {
   // order by surname
   cmp = lastName.compareTo(s.lastName);
   if(cmp == 0)
      // if equal surnames, order by name
      cmp = firstName.compareTo(s.firstName);
   return cmp;
}
```

The Comparator Interface

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

- 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**

The Comparator Interface

```
public class StudentComparator implements
Comparator<Student> {
    public int compare(Student s1, Student s2) {
        cmp = s1.lastName.compareTo(s2.lastName);
        if(cmp == 0)
            cmp = s1.firstName.compareTo(s2.firstName);
        return cmp;
    }
}
```

Algorithms

- Static methods of java.util.Collections class
  - Work on lists

- sort() - merge sort implementation, n log(n)
- binarySearch() - requires ordered sequence
- shuffle() - unsort
- reverse() - requires ordered sequence
- rotate() - of a given distance
- min(), max() - in a Collection

Generic Collections

- From Java 5, all collection interfaces and classes have been redefined as Generics
- Use of generics lead to code that is
  - safer
  - more compact
  - easier to understand
  - equally performing

Generic Collections

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

public interface List<E>{
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E>{
    E next();
    booleanhasNext();
}
```