Utdrag ur JCF ämnat för EDAA30 ht15

java.lang
Interface Iterable<T>

Type Parameters: `T` - the type of elements returned by the iterator

Modifier and Type	Method and Description
`Iterator<T>`	`iterator()` Returns an iterator over a set of elements of type T.

java.util
Interface Iterator<E>

Type Parameters: `E` - the type of elements returned by this iterator

Modifier and Type	Method and Description
`boolean`	`hasNext()` Returns true if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

java.lang
Interface Comparable<T>

Type Parameters: `T` - the type of objects that this object may be compared to

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

java.lang
Interface Comparator<T>

Type Parameters: `T` - the type of objects that may be compared by this comparator

Modifier and Type	Method and Description
`int`	`compare(T o1, T o2)` Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Interface List<E>

Type Parameters:

`E` - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

Known Implementing Classes:

AbstractList, ArrayList LinkedList

```
public interface List<E>
extends Collection<E>
```

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.

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

java.util

Java™ Platform Standard Ed. 7

Interface Map<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

Known Subinterfaces:

SortedMap<K,V>

Known Implementing Classes:

HashMap, TreeMap

`public interface` **`Map<K,V>`**

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

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

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.

This interface is a member of the Java Collections Framework.

Nested Class Summary

Nested Classes

Modifier and Type	Interface and Description
`static interface`	**`Map.Entry<K,V>`** A map entry (key-value pair).

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`clear()`** Removes all of the mappings from this map (optional operation).
`boolean`	**`containsKey(Object key)`** Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue(Object value)`** Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a **`Set`** view of the mappings contained in this map.
`boolean`	**`equals(Object o)`** Compares the specified object with this map for equality.
`V`	**`get(Object key)`** Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`int`	**`hashCode()`** Returns the hash code value for this map.
`boolean`	**`isEmpty()`** Returns `true` if this map contains no key-value mappings.
`Set<K>`	**`keySet()`** Returns a **`Set`** view of the keys contained in this map.
`V`	**`put(K key, V value)`** Associates the specified value with the specified key in this map (optional operation).
`void`	**`putAll(Map<? extends K,? extends V> m)`** Copies all of the mappings from the specified map to this map (optional operation).
`V`	**`remove(Object key)`** Removes the mapping for a key from this map if it is present (optional operation).
`int`	**`size()`** Returns the number of key-value mappings in this map.
`Collection<V>`	**`values()`** Returns a **`Collection`** view of the values contained in this map.

java.util

Interface Map.Entry<K,V>

All Known Implementing Classes:
AbstractMap.SimpleEntry, AbstractMap.SimpleImmutableEntry

Enclosing interface:
Map<K,V>

`public static interface` **`Map.Entry<K,V>`**
A map entry (key-value pair). The `Map.entrySet` method returns a collection-view of the map, whose elements are of this class. The *only* way to obtain a reference to a map entry is from the iterator of this collection-view. These `Map.Entry` objects are valid *only* for the duration of the iteration; more formally, the behavior of a map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except through the `setValue` operation on the map entry.
Since:
1.2
See Also:
`Map.entrySet()`

Method Summary

Modifier and Type	Method and Description
`boolean`	**`equals(Object o)`** Compares the specified object with this entry for equality.
`K`	**`getKey()`** Returns the key corresponding to this entry.
`V`	**`getValue()`** Returns the value corresponding to this entry.
`int`	**`hashCode()`** Returns the hash code value for this map entry.
`V`	**`setValue(V value)`** Replaces the value corresponding to this entry with the specified value (optional operation).

Interface Queue<E>

All Superinterfaces:

Collection<E>, Iterable<E>

Known Implementing Classes:

LinkedList , PriorityQueue

`public interface` **`Queue<E>`**
`extends Collection<E>`

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.

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

Interface Set<E>

All Superinterfaces:

Collection<E>, Iterable<E>

Known Subinterfaces:

NavigableSet<E>, SortedSet<E>

All Known Implementing Classes:

HashSet, TreeSet

```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

This interface is a member of the Java Collections Framework.

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Interface SortedMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Superinterfaces:

Map<K,V>

Known Implementing Classes:

TreeMap

```
public interface SortedMap<K,V>
extends Map<K,V>
```

A `Map` that further provides a *total ordering* on its keys. The map is ordered according to the natural ordering of its keys, or by a `Comparator` typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the `entrySet`, `keySet` and `values` methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of `SortedSet`.)

All keys inserted into a sorted map must implement the `Comparable` interface (or be accepted by the specified comparator). Furthermore, all such keys must be *mutually comparable*: `k1.compareTo(k2)` (or `comparator.compare(k1, k2)`) must not throw a `ClassCastException` for any keys `k1` and `k2` in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a `ClassCastException`.

This interface is a member of the Java Collections Framework.

Nested Class Summary

Nested classes/interfaces inherited from interface java.util.Map

`Map.Entry<K,V>`

Method Summary

Methods

Modifier and Type	Method and Description
`Comparator<? super K>`	**`comparator()`** Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a `Set` view of the mappings contained in this map.
`K`	**`firstKey()`** Returns the first (lowest) key currently in this map.
`SortedMap<K,V>`	**`headMap(K toKey)`** Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`Set<K>`	**`keySet()`** Returns a `Set` view of the keys contained in this map.
`K`	**`lastKey()`** Returns the last (highest) key currently in this map.
`SortedMap<K,V>`	**`subMap(K fromKey, K toKey)`** Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
`SortedMap<K,V>`	**`tailMap(K fromKey)`** Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
`Collection<V>`	**`values()`** Returns a `Collection` view of the values contained in this map.

Interface SortedSet<E>

All Superinterfaces:

Collection<E>, Iterable<E>, Set<E>

Known Implementing Classes:

TreeSet

```
public interface SortedSet<E>
extends Set<E>
```

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`.)

All elements inserted into a sorted set must implement the `Comparable` interface (or be accepted by the specified comparator). Furthermore, all such elements must be *mutually comparable*: `e1.compareTo(e2)` (or `comparator.compare(e1, e2)`) must not throw a `ClassCastException` for any elements `e1` and `e2` in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a `ClassCastException`.

This interface is a member of the Java Collections Framework.

Method Summary

Methods

Modifier and Type	Method and Description
`Comparator<? super E>`	**`comparator()`** Returns the comparator used to order the elements in this set, or `null` if this set uses the **natural ordering** of its elements.
`E`	**`first()`** Returns the first (lowest) element currently in this set.
`SortedSet<E>`	**`headSet(E toElement)`** Returns a view of the portion of this set whose elements are strictly less than `toElement`.
`E`	**`last()`** Returns the last (highest) element currently in this set.
`SortedSet<E>`	**`subSet(E fromElement, E toElement)`** Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
`SortedSet<E>`	**`tailSet(E fromElement)`** Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.

Class HashMap<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

Implemented Interfaces:

Map<K,V>

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

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.

This class is a member of the Java Collections Framework.

Nested Class Summary

Nested classes/interfaces inherited from class java.util.AbstractMap

AbstractMap.SimpleEntry<K,V>, AbstractMap.SimpleImmutableEntry<K,V>

Constructor Summary

Constructors

Constructor and Description
HashMap() Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).
HashMap(int initialCapacity) Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).
HashMap(int initialCapacity, float loadFactor) Constructs an empty `HashMap` with the specified initial capacity and load factor.
HashMap(Map<? extends K,? extends V> m) Constructs a new `HashMap` with the same mappings as the specified `Map`.

Method Summary

Methods

Modifier and Type	Method and Description
void	**clear**() Removes all of the mappings from this map.
Object	**clone**() Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
boolean	**containsKey**(Object key) Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns `true` if this map maps one or more keys to the specified value.
Set<**Map.Entry**<K,V>>	**entrySet**() Returns a **Set** view of the mappings contained in this map.
V	**get**(Object key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map.
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key) Removes the mapping for the specified key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a **Collection** view of the values contained in this map.

Class HashSet<E>

Implemented Interfaces:

Iterable<E>, Collection<E>, Set<E>

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the `null` element.

This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

This class is a member of the Java Collections Framework.

Constructor Summary

Constructors

Constructor and Description
`HashSet()` Constructs a new, empty set; the backing `HashMap` instance has default initial capacity (16) and load factor (0.75).
`HashSet(Collection<? extends E> c)` Constructs a new set containing the elements in the specified collection.
`HashSet(int initialCapacity)` Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and default load factor (0.75).
`HashSet(int initialCapacity, float loadFactor)` Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and the specified load factor.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	**`add(E e)`** Adds the specified element to this set if it is not already present.
`void`	**`clear()`** Removes all of the elements from this set.
`Object`	**`clone()`** Returns a shallow copy of this `HashSet` instance: the elements themselves are not cloned.
`boolean`	**`contains(Object o)`** Returns `true` if this set contains the specified element.
`boolean`	**`isEmpty()`** Returns `true` if this set contains no elements.
`Iterator<E>`	**`iterator()`** Returns an iterator over the elements in this set.
`boolean`	**`remove(Object o)`** Removes the specified element from this set if it is present.
`int`	**`size()`** Returns the number of elements in this set (its cardinality).

java.util

Class LinkedList<E>

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>

implements List <E>
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Constructor Summary

Constructors

Constructor and Description
LinkedList() Constructs an empty list.
LinkedList(Collection<? extends E> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Appends the specified element to the end of this list.
void	**add(int index, E element)** Inserts the specified element at the specified position in this list.
boolean	**addAll(Collection<? extends E> c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll(int index, Collection<? extends E> c)** Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**addFirst(E e)** Inserts the specified element at the beginning of this list.
void	**addLast(E e)** Appends the specified element to the end of this list.
void	**clear()** Removes all of the elements from this list.
Object	**clone()** Returns a shallow copy of this `LinkedList`.
boolean	**contains(Object o)** Returns `true` if this list contains the specified element.
Iterator<E>	**descendingIterator()** Returns an iterator over the elements in this deque in reverse sequential order.
E	**element()** Retrieves, but does not remove, the head (first element) of this list.
E	**get(int index)** Returns the element at the specified position in this list.
E	**getFirst()** Returns the first element in this list.
E	**getLast()** Returns the last element in this list.
int	**indexOf(Object o)** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	**lastIndexOf(Object o)** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator(int index)** Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
boolean	**offer(E e)** Adds the specified element as the tail (last element) of this list.
boolean	**offerFirst(E e)** Inserts the specified element at the front of this list.
boolean	**offerLast(E e)** Inserts the specified element at the end of this list.
E	**peek()** Retrieves, but does not remove, the head (first element) of this list.
E	**peekFirst()** Retrieves, but does not remove, the first element of this list, or returns `null` if this list is empty.
E	**peekLast()** Retrieves, but does not remove, the last element of this list, or returns `null` if this list is empty.
E	**poll()** Retrieves and removes the head (first element) of this list.
E	**pollFirst()** Retrieves and removes the first element of this list, or returns `null` if this list is empty.
E	**pollLast()** Retrieves and removes the last element of this list, or returns `null` if this list is empty.
E	**pop()** Pops an element from the stack represented by this list.
void	**push(E e)** Pushes an element onto the stack represented by this list.
E	**remove()** Retrieves and removes the head (first element) of this list.

E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(**Object** o) Removes the first occurrence of the specified element from this list, if it is present.
E	**removeFirst**() Removes and returns the first element from this list.
boolean	**removeFirstOccurrence**(**Object** o) Removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
E	**removeLast**() Removes and returns the last element from this list.
boolean	**removeLastOccurrence**(**Object** o) Removes the last occurrence of the specified element in this list (when traversing the list from head to tail).
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Methods inherited from interface java.util.List

containsAll, equals, hashCode, isEmpty, iterator, listIterator, removeAll, retainAll, subList

Class PriorityQueue<E>

Implemented Interfaces:

Iterable<E>, Collection<E>, Queue<E>

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

Constructor Summary

Constructors

Constructor and Description
PriorityQueue() Creates a `PriorityQueue` with the default initial capacity (11) that orders its elements according to their **natural ordering**.
PriorityQueue(**Collection**<? extends **E**> c) Creates a `PriorityQueue` containing the elements in the specified collection.
PriorityQueue(int initialCapacity) Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to their **natural ordering**.
PriorityQueue(int initialCapacity, **Comparator**<? super **E**> comparator) Creates a `PriorityQueue` with the specified initial capacity that orders its elements according to the specified comparator.
PriorityQueue(**PriorityQueue**<? extends **E**> c) Creates a `PriorityQueue` containing the elements in the specified priority queue.
PriorityQueue(**SortedSet**<? extends **E**> c) Creates a `PriorityQueue` containing the elements in the specified sorted set.

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Inserts the specified element into this priority queue.
void	**clear**() Removes all of the elements from this priority queue.
Comparator<? super E>	**comparator**() Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the **natural ordering** of its elements.
boolean	**contains**(Object o) Returns true if this queue contains the specified element.
Iterator<E>	**iterator**() Returns an iterator over the elements in this queue.
boolean	**offer**(E e) Inserts the specified element into this priority queue.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns null if this queue is empty.
boolean	**remove**(Object o) Removes a single instance of the specified element from this queue, if it is present.
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this queue.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Class TreeMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

Implemented Interfaces:

Map<K,V>, SortedMap<K,V>

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

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*.

This class is a member of the Java Collections Framework.

Nested Class Summary

Nested classes/interfaces inherited from class java.util.AbstractMap

AbstractMap.SimpleEntry<K,V>, AbstractMap.SimpleImmutableEntry<K,V>

Constructor Summary

Constructors

Constructor and Description
TreeMap() Constructs a new, empty tree map, using the natural ordering of its keys.
TreeMap(Comparator<? super K> comparator) Constructs a new, empty tree map, ordered according to the given comparator.
TreeMap(Map<? extends K,? extends V> m) Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
TreeMap(SortedMap<K,? extends V> m) Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

Method Summary

Methods

Modifier and Type	Method and Description
`Map.Entry<K,V>`	`ceilingEntry(K key)` Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
`K`	`ceilingKey(K key)` Returns the least key greater than or equal to the given key, or `null` if there is no such key.
`void`	`clear()` Removes all of the mappings from this map.
`Object`	`clone()` Returns a shallow copy of this `TreeMap` instance.
`Comparator<? super K>`	`comparator()` Returns the comparator used to order the keys in this map, or `null` if this map uses the **natural ordering** of its keys.
`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`NavigableSet<K>`	`descendingKeySet()` Returns a reverse order `NavigableSet` view of the keys contained in this map.
`NavigableMap<K,V>`	`descendingMap()` Returns a reverse order view of the mappings contained in this map.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
`Map.Entry<K,V>`	`firstEntry()` Returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`K`	`firstKey()` Returns the first (lowest) key currently in this map.
`Map.Entry<K,V>`	`floorEntry(K key)` Returns a key-value mapping associated with the greatest key less than or equal to the given key, or `null` if there is no such key.
`K`	`floorKey(K key)` Returns the greatest key less than or equal to the given key, or `null` if there is no such key.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`SortedMap<K,V>`	`headMap(K toKey)` Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`NavigableMap<K,V>`	`headMap(K toKey, boolean inclusive)` Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
`Map.Entry<K,V>`	`higherEntry(K key)` Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.
`K`	`higherKey(K key)` Returns the least key strictly greater than the given key, or `null` if there is no such key.
`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.
`Map.Entry<K,V>`	`lastEntry()` Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`K`	`lastKey()` Returns the last (highest) key currently in this map.
`Map.Entry<K,V>`	`lowerEntry(K key)` Returns a key-value mapping associated with the greatest key strictly less than the given key, or `null` if there is no such key.
`K`	`lowerKey(K key)` Returns the greatest key strictly less than the given key, or `null` if there is no such key.
`NavigableSet<K>`	`navigableKeySet()` Returns a `NavigableSet` view of the keys contained in this map.
`Map.Entry<K,V>`	`pollFirstEntry()` Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`Map.Entry<K,V>`	`pollLastEntry()` Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map.
`void`	`putAll(Map<? extends K,? extends V> map)` Copies all of the mappings from the specified map to this map.
`V`	`remove(Object key)` Removes the mapping for this key from this TreeMap if present.
`int`	`size()` Returns the number of key-value mappings in this map.
`NavigableMap<K,V>`	`subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)` Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`.
`SortedMap<K,V>`	`subMap(K fromKey, K toKey)` Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
`SortedMap<K,V>`	`tailMap(K fromKey)` Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
`NavigableMap<K,V>`	`tailMap(K fromKey, boolean inclusive)` Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.
`Collection<V>`	`values()` Returns a `Collection` view of the values contained in this map.

java.util

Java™ Platform Standard Ed. 7

Class TreeSet<E>

Implemented Interfaces:

Iterable<E>, Collection<E>, Set<E>, SortedSet<E>

```
public class TreeSet<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

A `NavigableSet` implementation based on a `TreeMap`. The elements are ordered using their natural ordering, or by a `Comparator` provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (`add`, `remove` and `contains`).

This class is a member of the Java Collections Framework.

Constructor Summary

Constructors

Constructor and Description
`TreeSet()` Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
`TreeSet(Collection<? extends E> c)` Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.
`TreeSet(Comparator<? super E> comparator)` Constructs a new, empty tree set, sorted according to the specified comparator.
`TreeSet(SortedSet<E> s)` Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	**`add(E e)`** Adds the specified element to this set if it is not already present.
`boolean`	**`addAll(Collection<? extends E> c)`** Adds all of the elements in the specified collection to this set.
`E`	**`ceiling(E e)`** Returns the least element in this set greater than or equal to the given element, or `null` if there is no such element.
`void`	**`clear()`** Removes all of the elements from this set.
`Object`	**`clone()`** Returns a shallow copy of this `TreeSet` instance.
`Comparator<? super E>`	**`comparator()`** Returns the comparator used to order the elements in this set, or `null` if this set uses the **natural ordering** of its elements.
`boolean`	**`contains(Object o)`** Returns `true` if this set contains the specified element.
`Iterator<E>`	**`descendingIterator()`** Returns an iterator over the elements in this set in descending order.
`NavigableSet<E>`	**`descendingSet()`** Returns a reverse order view of the elements contained in this set.
`E`	**`first()`** Returns the first (lowest) element currently in this set.
`E`	**`floor(E e)`** Returns the greatest element in this set less than or equal to the given element, or `null` if there is no such element.
`SortedSet<E>`	**`headSet(E toElement)`** Returns a view of the portion of this set whose elements are strictly less than `toElement`.
`NavigableSet<E>`	**`headSet(E toElement, boolean inclusive)`** Returns a view of the portion of this set whose elements are less than (or equal to, if `inclusive` is true) `toElement`.
`E`	**`higher(E e)`** Returns the least element in this set strictly greater than the given element, or `null`