西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object Oriented Programming

Chapter 8
Collections

Dr. Muhammad Umar Farooq Qaisar

22nd April 2025

Slides partially adapted from lecture notes by Cay Horstmann

Questions

- The data structures can make a BIG difference when you try to implement methods in a natural style or are concerned with performance.
  1. Do you need to **search** quickly through thousands (or even millions) of sorted items?
  2. Do you need to rapidly **insert** and **remove** elements in the middle of an ordered sequence?
  3. Do you need to **establish associations** between keys and values?
- *Different from the Data Structures course, we will skip the theory and just show you how to use the collection classes in the standard library.*

Contents

8.1 The Java Collections Framework

- The initial release of Java supplied only a small set of classes for the most useful data structures: `Vector`, `Stack`, `Hashtable`, `BitSet`, and the `Enumeration` interface that provides an abstract mechanism for visiting elements in an arbitrary container.
  - That was certainly a wise choice—it takes time and skill to come up with a comprehensive collection class library.

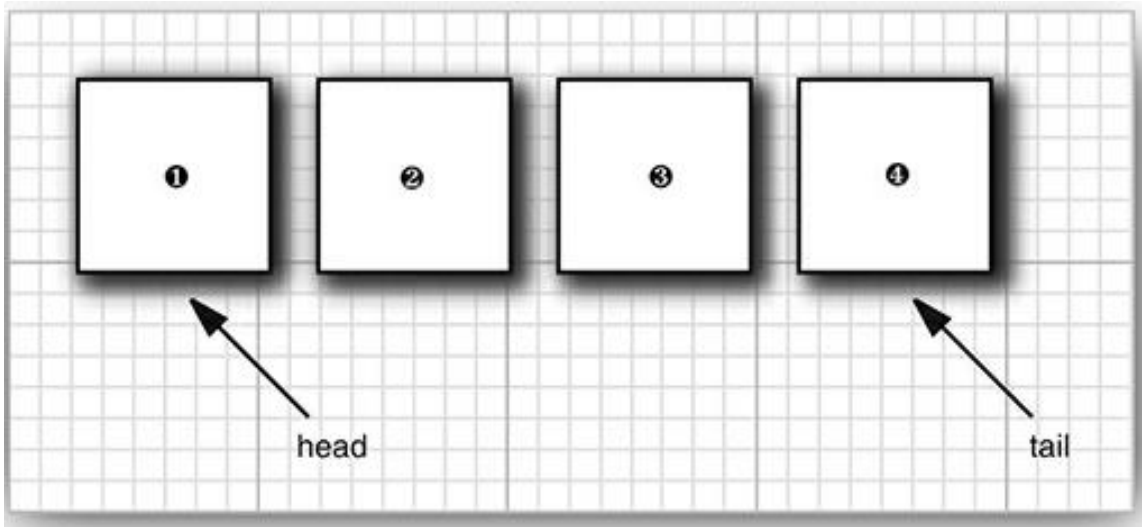
- As of Java 1.2, the designers felt that the time had come to roll out a full-fledged set of data structures.
  - The library should be ***small and easy to learn***.

8.1.1 Separating Collection Interfaces and Implementation

- The Java collection framework separates *interfaces* and *implementations*.
  - A *queue interface* provides abstract specification:

```
public interface Queue<E> { // simplified form
    void add(E element);
    E remove();
    int size();
}
```

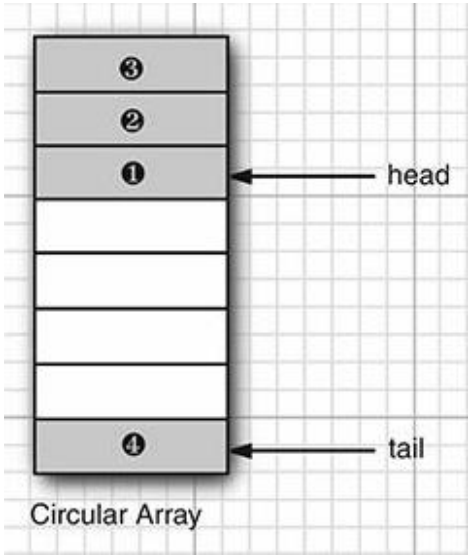
"first in, first out"

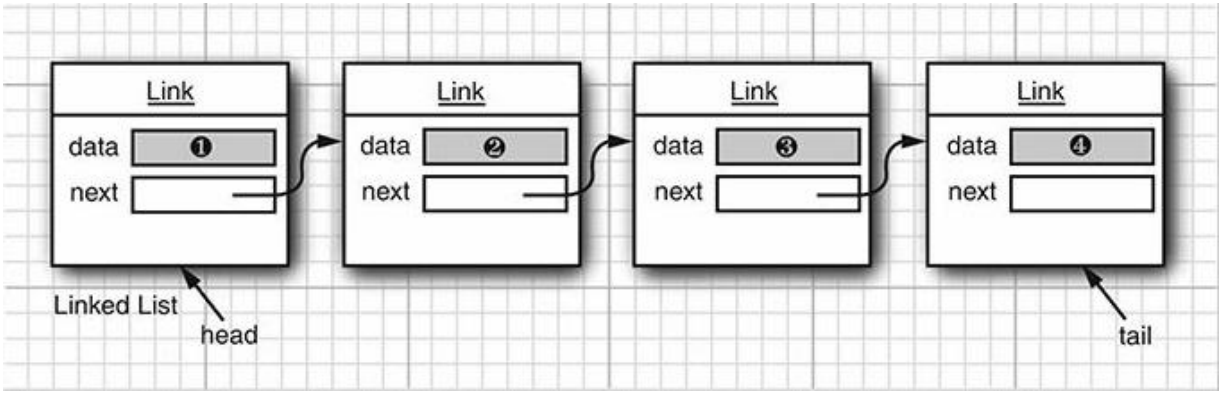


8.1.1 Separating Collection Interfaces and Implementation

- A collection interface can have multiple implementing classes that implement the Queue interface.

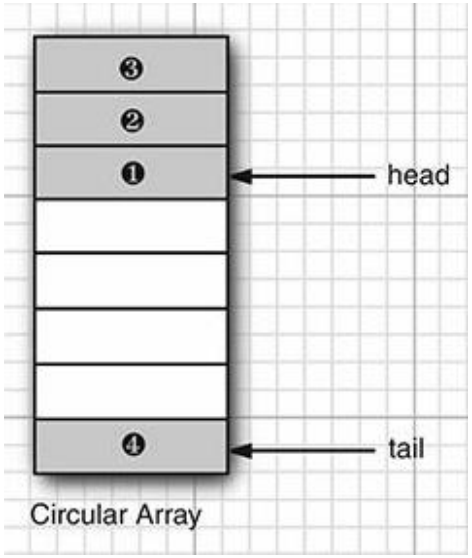
```
public class CircularArrayQueue<E> implements Queue<E>
public class LinkedListQueue<E> implements Queue<E>
    //not actual library classes
```

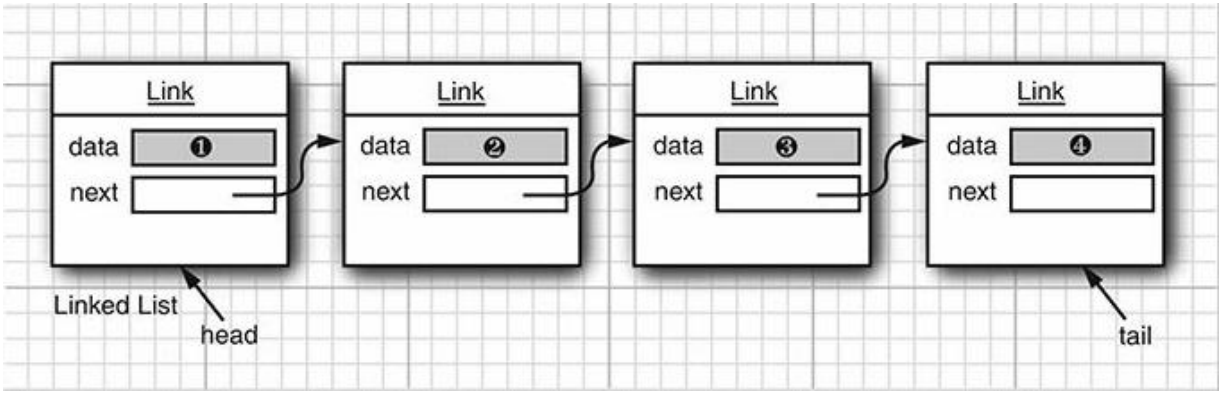




8.1.1 Separating Collection Interfaces and Implementation

- The interface tells you nothing about how the queue is implemented.
- Of the two common implementations of a queue, one uses a “circular array” and one uses a linked list.





8.1.1 Separating Collection Interfaces and Implementation

- Each implementation can be expressed by a class that implements the Queue interface.

```
public class CircularArrayQueue<E> implements Queue<E> // not an actual library
class
{
 private int head;
 private int tail;
 CircularArrayQueue(int capacity) { . . . }
 public void add(E element) { . . . }
 public E remove() { . . . }
 public int size() { . . . }
 private E[] elements;
}
```

```
public class LinkedListQueue<E> implements Queue<E> // not an actual library class
{
 private Link head;
 private Link tail;
 LinkedListQueue() { . . . }
 public void add(E element) { . . . }
 public E remove() { . . . }
 public int size() { . . . }
}
```

8.1.1 Separating Collection Interfaces and Implementation

- Always use the *interface type* to hold the collection reference after creation:

```
Queue<Customer> expressLane = new CircularArrayQueue<>(100);
expressLane.add(new Customer("Harry"));
```

- If you want to use a different implementation, change your program in the constructor call.

```
Queue<Customer> expressLane = new LinkedListQueue<>();
expressLane.add(new Customer("Harry"));
```

  - A circular array is somewhat more efficient than a linked list.
  - The circular array is a *bounded* collection—it has a finite capacity.
  - If you don't have an upper limit on the number of objects that your program will collect, you may be better off with a linked list implementation after all.

8.1.2 The Collection Interface

- `Collection<E>` has two fundamental methods:

```
public interface Collection<E> {
    boolean add(E element);
    Iterator<E> iterator();
    . . .
}
```

- The `add` method adds an element to the collection and returns `true` or `false` that indicates if the element added changes the collection.
- The `iterator` method returns an object that implements the `Iterator` interface. You can use the iterator object to visit the elements in the collection one by one.

8.1.3 Iterators

- The `Iterator` interface has four methods:

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
    default void forEachRemaining(Consumer<? super E> action);
}
```

- By repeatedly calling the next method, you can visit the elements from the collection one by one.
- However, if you reach the end of the collection, the next method throws a `NoSuchElementException`.
- Therefore, you need to call the hasNext method before calling next. That method returns true if the iterator object still has more elements to visit.
- If you want to inspect all elements in a collection, request an iterator and then keep calling the next method while hasNext returns true.

8.1.3 Iterators

- Get an iterator from a collection to visit all elements:

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
    String element = iter.next();
    // do something with element
}
```

- More concisely as the “for each” loop:

```
for (String element : c) {
    // do something with element
}
```

- The compiler simply translates the “for each” loop into a loop with an iterator.

8.1.3 Iterators

- The “for each” loop works with any object that implements the `Iterable` interface with a single abstract method:

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

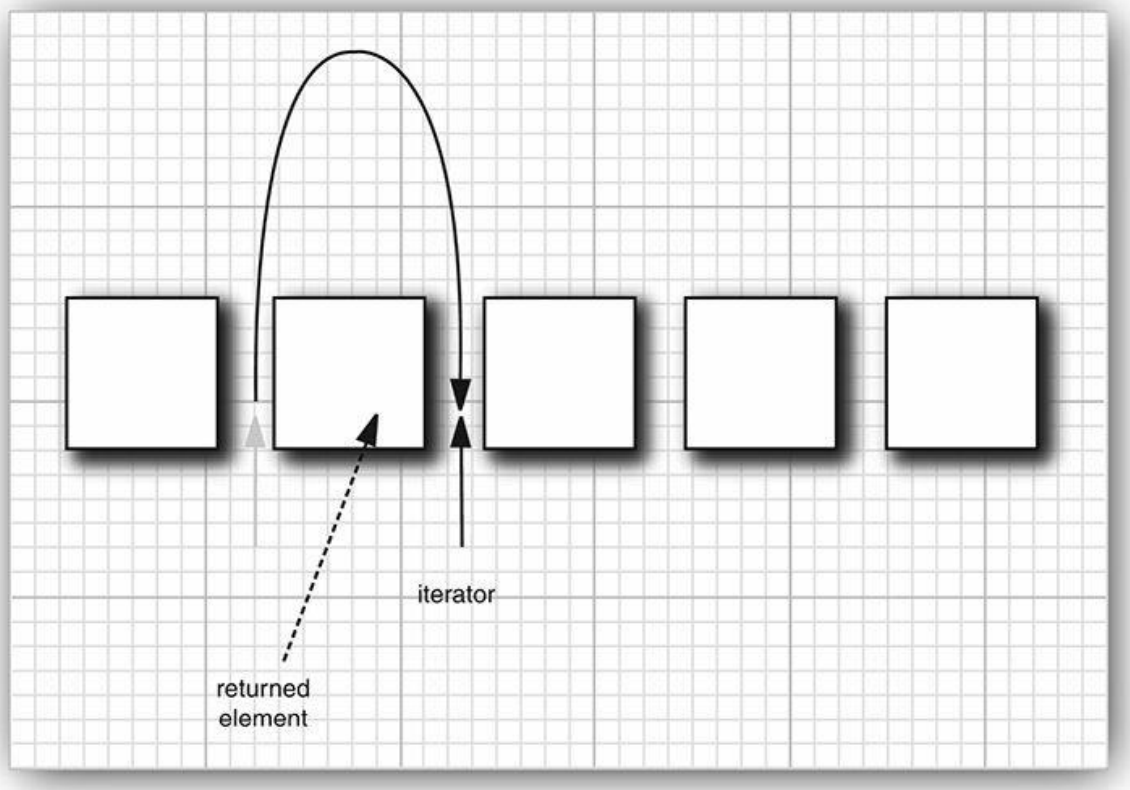
- The `Collection` interface extends the `Iterable` interface.
- Therefore, you can use the “`for each`” loop with any collection in the standard library
- Or without any loop:

```
iterator.forEachRemaining(element -> do something with element);
```

  - The order in which the elements are visited depends on the collection type.
  - The only way to look up an element is to call `next`, and that lookup advances the position.

8.1.3 Iterators

- **Think of Java iterators as being between elements.**
  - *When you call `next`, the iterator jumps over the next element, and returns a reference to the element that it just passed.*



8.1.3 Iterators

- The `remove` method removes the element that was just returned by next:

```
Iterator<String> it = c.iterator();
it.next();   // skip over the first element
it.remove(); // now remove it
```

- **Caution**: Calling remove twice in a row without calling next in between is an error.

```
it.remove();
it.remove(); // ERROR

it.remove();
it.next();
it.remove(); // OK
```

8.1.4 Generic Utility Methods

- The `Collection` and `Iterator` interfaces are generic.
  - You can write utility methods that operate on any kind of collection.
- The `Collection` interface declares quite a few useful methods that all implementing classes must supply.

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

8.1.4 Generic Utility Methods

java.util.Collection<E> 1.2

- `Iterator<E> iterator()`
  returns an iterator that can be used to visit the elements in the collection.
- `int size()`
  returns the number of elements currently stored in the collection.
- `boolean isEmpty()`
  returns `true` if this collection contains no elements.
- `boolean contains(Object obj)`
  returns `true` if this collection contains an object equal to `obj`.

java.util.Iterator<E> 1.2

- `boolean hasNext()`
  returns `true` if there is another element to visit.
- `E next()`
  returns the next object to visit. Throws a `NoSuchElementException` if the end of the collection has been reached.
- `void remove()`
  removes the last visited object. This method must immediately follow an element visit. If the collection has been modified since the last element visit, this method throws an `IllegalStateException`.
- `default void forEachRemaining(Consumer<? super E> action)` 8
  visits elements and passes them to the given action until no elements remain or the action throws an exception.

java.util.Collection<E> 1.2 *(Continued)*

- `boolean containsAll(Collection<?> other)`
  returns `true` if this collection contains all elements in the other collection.
- `boolean add(E element)`
  adds an element to the collection. Returns `true` if the collection changed as a result of this call.
- `boolean addAll(Collection<? extends E> other)`
  adds all elements from the other collection to this collection. Returns `true` if the collection changed as a result of this call.
- `boolean remove(Object obj)`
  removes an object equal to `obj` from this collection. Returns `true` if a matching object was removed.
- `boolean removeAll(Collection<?> other)`
  removes from this collection all elements from the other collection. Returns `true` if the collection changed as a result of this call.
- `default boolean removeIf(Predicate<? super E> filter)` 8
  removes all elements for which `filter` returns `true`. Returns `true` if the collection changed as a result of this call.
- `void clear()`
  removes all elements from this collection.
- `boolean retainAll(Collection<?> other)`
  removes all elements from this collection that do not equal one of the elements in the other collection. Returns `true` if the collection changed as a result of this call.
- `Object[] toArray()`
  returns an array of the objects in the collection.
- `<T> T[] toArray(T[] arrayToFill)`
  returns an array of the objects in the collection. If `arrayToFill` has sufficient length, it is filled with the elements of this collection. If there is space, a `null` element is appended. Otherwise, a new array with the same component type as `arrayToFill` and the same length as the size of this collection is allocated and filled.

8.1.4 Generic Utility Methods

- To make life easier for implementors, the library supplies a class `AbstractCollection`.

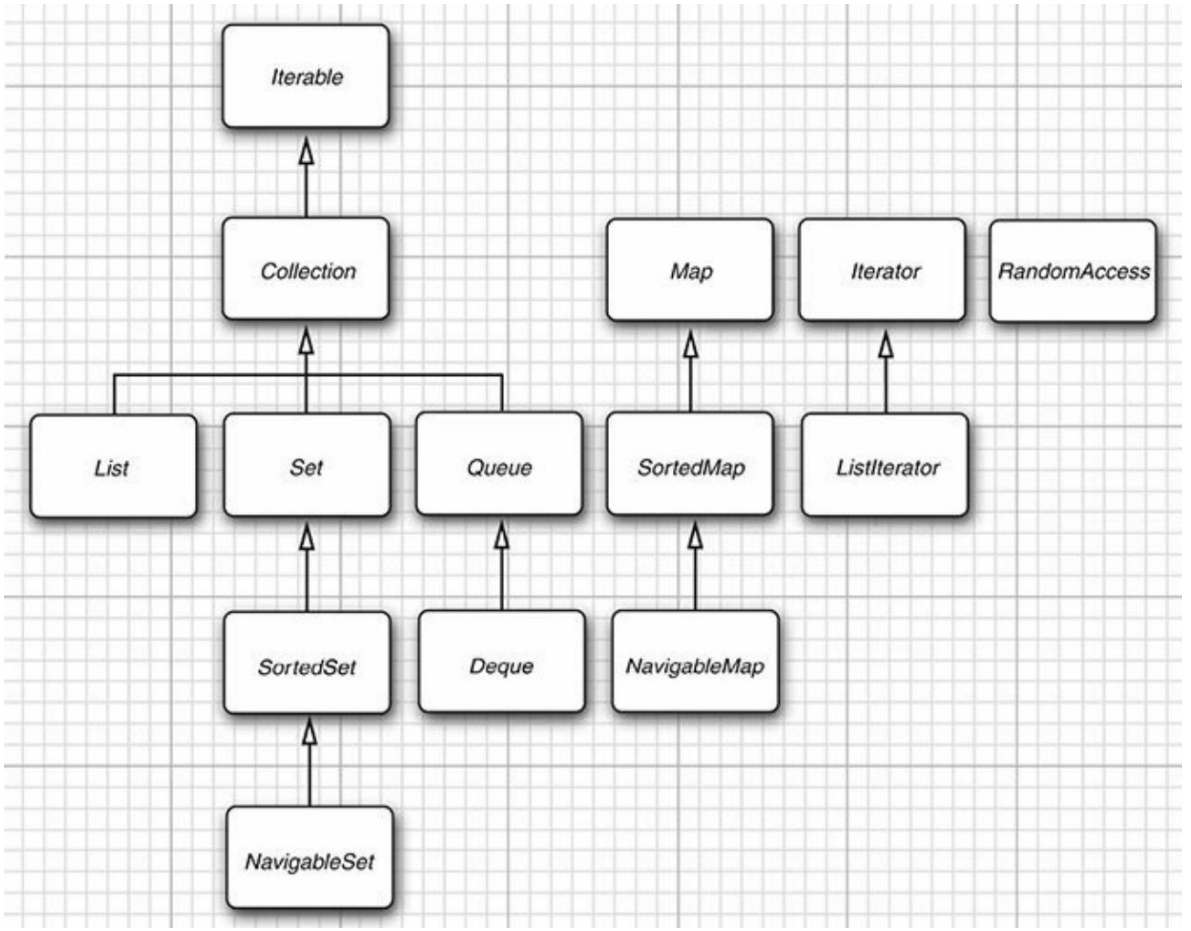
```
public abstract class AbstractCollection<E>
     implements Collection<E> {
     . . .
     public abstract Iterator<E> iterator();
     public boolean contains(Object obj) {
          for (E element : this)              // calls iterator()
               if (element.equals(obj))
                    return true;
          return false;
     }
}
```

- **A concrete collection class can extend the `AbstractCollection`.**
  - The concrete collection class can supply an `iterator` method, but the `contains` method has been taken care of by the `AbstractCollection` superclass.
  - However, if the subclass has a more efficient way of implementing `contains`, it is free to do so.

8.1.5 Interfaces in Collections

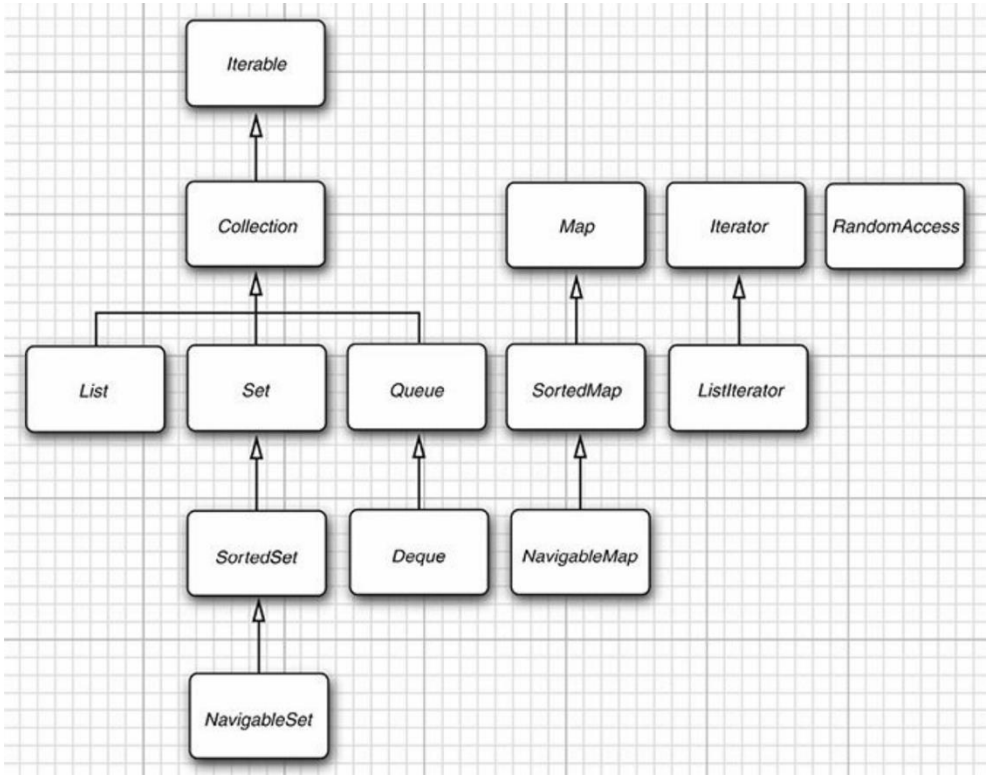
- The Java collections framework defines a number of interfaces for different types of collections.

Two fundamental interfaces for collections: `Collection` and `Map`



8.1.5 Interfaces in Collections

- `Collection` holds elements, `Map` holds key/value pairs.
- `List`: Ordered collection.
- `Set`: Unordered collection without duplicates.
- `SortedSet/SortedMap`: Traversed in sorted order.
- `NavigableSet/NavigableMap`: Additional methods for sorted sets/maps.



8.1.5 Interfaces in Collections

- There are two fundamental interfaces for collections: Collection and Map. As you already saw, you insert elements into a collection with a method.

```
boolean add(E element)
```

- However, maps hold key/value pairs, and you use the put method to insert them:

```
V put(K key, V value)
```

  - K key: This is the key in the map. The key is used to uniquely identify the corresponding value in the map. Each key in a map must be unique.
  - V value: This is the value associated with the given key. The value can be any object (or primitive) that you want to store in the map. The value is accessed using the key.
- To read elements from a collection, visit them with an iterator. However, you can read values from a map with the get method:

```
V get(K key)
```

8.1.5 Interfaces in Collections

- A List is an ordered collection. Elements are added into a particular position in the container. An element can be accessed in two ways: by an iterator or by an integer index.
- The *latter* is called random access because elements can be visited in any order. In contrast, when using an iterator, one must visit them sequentially.
- The List interface defines several methods for random access:

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

- The ListIterator interface is a sub-interface of Iterator. It defines a method for adding an element before the iterator position:

```
void add(E element)
```

8.1.5 Interfaces in Collections

- In practice, there are two kinds of ordered collections, with very different performance tradeoffs.
  - An ordered collection that is backed by an array has fast random access, and it makes sense to use the List methods with an integer index.
  - In contrast, a linked list, while also ordered, has slow random access, and it is best traversed with an iterator.
- The Set interface is identical to the Collection interface, but the behavior of the methods is more tightly defined.
  - The add method of a set should reject duplicates.
    - **Explanation**: The add() method rejects the duplicate "apple" because sets do not allow duplicate elements.
  - The equals method of a set should be defined so that two sets are identical if they have the same elements, but not necessarily in the same order.
    - **Explanation**: Despite the different order of elements in the two sets, set1.equals(set2) returns true because they contain the same elements.
  - The hashCode method should be defined so that two sets with the same elements yield the same hash code.
    - **Explanation**: Even though the sets may have different internal orderings, the hashCode() for set1 and set2 will be the same because they contain the same elements.

8.1.5 Interfaces in Collections

- Why make a separate interface if the method signatures are the same?
  - Conceptually, not all collections are sets. Making a Set interface enables programmers to write methods that accept only sets.
- The SortedSet and SortedMap interfaces expose the comparator object used for sorting, and they define methods to obtain views of subsets of the collections.
  - SortedSet and SortedMap allow sorting based on a comparator and provide views like subSet(), tailSet(), headSet() for SortedSet and subMap(), headMap(), tailMap() for SortedMap.
  - Comparator: It defines the sorting order, and it can be customized (e.g., sorting in descending order).
- Finally, Java 6 introduced interfaces NavigableSet and NavigableMap that contain additional methods for searching and traversal in sorted sets and maps.
  - The TreeSet and TreeMap classes implement these interfaces.

Top Hat Question

Q. What is the main purpose of the Java Collections Framework?

A. To define custom data structures

B. To provide a set of standard interfaces and classes for handling groups of objects

C. To define how objects are stored in memory

D. To improve the performance of algorithms

Answer: **B**. To provide a set of standard interfaces and classes for handling groups of objects

Explanation: The Collections Framework provides a well-defined set of classes and interfaces to store and manipulate collections of objects, such as lists, sets, and maps.

Top Hat Question

Q. Which collection interface allows duplicate elements?

A. Set

B. List

C. Map

D. Queue

Answer: B. List

Explanation:
A List allows duplicate elements, while a Set does not. A Map allows duplicate values but not duplicate keys.

Top Hat Question

Q. What is the default order of elements in a HashSet?

A. Sorted by their natural order

B. In the order they were added

C. Undefined

D. In descending order

Answer: C. Undefined

Explanation: The order of elements in a HashSet is undefined because it uses hashing for storage, so elements are not stored in any predictable order.

Top Hat Question

Q. Which of the following collections implements both Set and SortedSet interfaces?

A. HashSet

B. TreeSet

C. LinkedHashSet

D. PriorityQueue

Answer: B. TreeSet

Explanation: A TreeSet implements both the Set and SortedSet interfaces, maintaining elements in a sorted order.

Top Hat Question

Q. Which method of the List interface is used to insert an element at a specific index?

A. insert()

B. addAt()

C. add(index, element)

D. put(index, element)

Answer: C. add(index, element)

Explanation: The add(index, element) method is used to insert an element at a specified index in a List. This shifts elements at and after the index.

Top Hat Question

Q. Which collection interface allows storing key-value pairs?

A. List

B. Set

C. Map

D. Queue

Answer: C. Map

Explanation: The Map interface stores key-value pairs, where each key is associated with a value.

Top Hat Question

Q. What is the difference between LinkedList and ArrayList?

A. LinkedList stores elements in an array, while ArrayList stores elements in a linked list

B. LinkedList provides constant-time access by index, while ArrayList does not

C. LinkedList is faster for insertions and deletions, while ArrayList is faster for access by index

D. LinkedList is part of the Collections Framework, while ArrayList is not

Answer: C. LinkedList is faster for insertions and deletions, while ArrayList is faster for access by index

Explanation: LinkedList is better for frequent insertions and deletions as it uses a doubly linked list structure. ArrayList is faster for indexed access because it is backed by an array.

Code Breaker Puzzle

```
1) class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

This code will throw an error. What's wrong, and how can you fix it?

```
2) public class CodeBreakerPuzzle {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 25));
        people.add(new Person("Bob", 30));
        people.add(new Person("Charlie", 20));

 // Sorting
        Collections.sort(people);
        for (Person p : people) {
            System.out.println(p.name + " - " +
p.age);
        }
    }
}
```

Code Breaker Puzzle

Explanation: The Person class does not implement Comparable, which is required for sorting. We need to either implement Comparable or provide a Comparator to define how Person objects should be compared.

Fix by implementing Comparable:

```
class Person implements Comparable<Person> {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
// Sort by age
    }
}
```

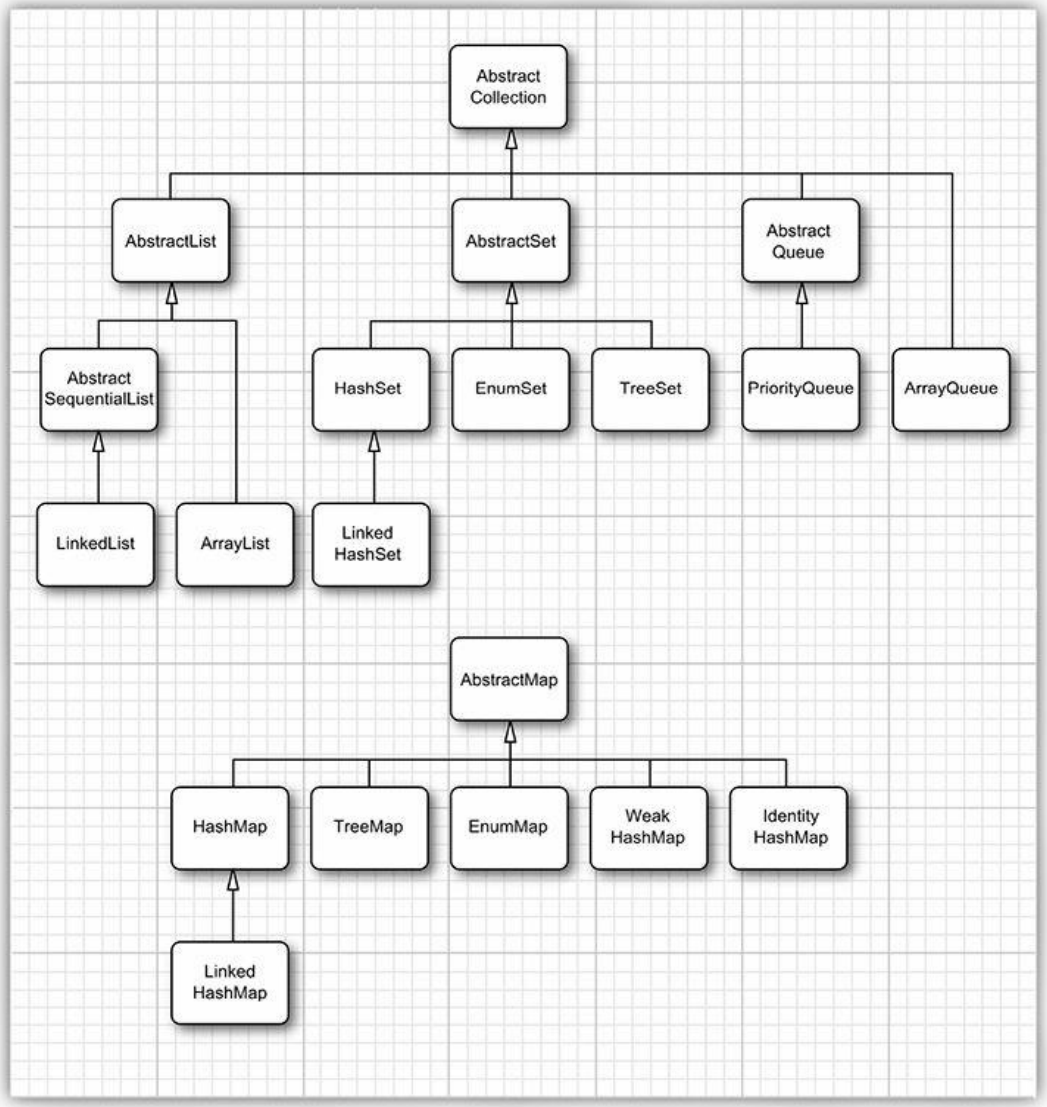
Alternatively, you can use a Comparator:

```
Collections.sort(people, (p1, p2) ->
Integer.compare(p1.age, p2.age));  // Sort by age
```

Contents

Collection Classes

Classes in the collections framework



Concrete Collections

ArrayList	An indexed sequence that grows and shrinks dynamically
LinkedList	An ordered sequence that allows efficient insertion and removal at any location
ArrayDeque	A double-ended queue that is implemented as a circular array
HashSet	An unordered collection that rejects duplicates
TreeSet	A sorted set
EnumSet	A set of enumerated type values
LinkedHashSet	A set that remembers the order in which elements were inserted
PriorityQueue	A collection that allows efficient removal of the smallest element
HashMap	A data structure that stores key/value associations
TreeMap	A map in which the keys are sorted
EnumMap	A map in which the keys belong to an enumerated type
LinkedHashMap	A map that remembers the order in which entries were added
WeakHashMap	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere
IdentityHashMap	A map with keys that are compared by ==, not equals

8.2.1 Linked Lists

- Two ordered collection implementations:
  - array lists and linked lists.
- Array lists manage an array that can grow or shrink.
- Inserting and removing in the middle is **slow**:
  - Because all array elements beyond the removed one must be moved toward the beginning of the array.
  - The same is true for inserting elements in the middle.

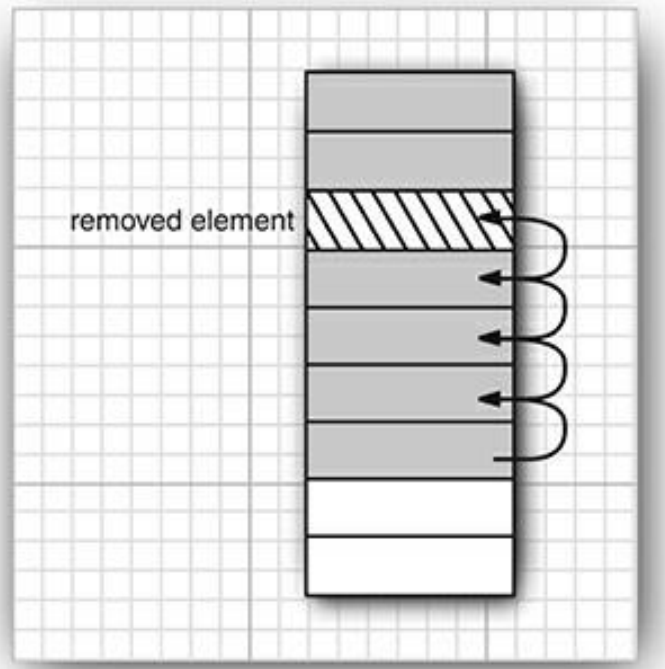


Figure 9.6 Removing an element from an array

8.2.1 Linked Lists

- Another well-known data structure, the linked list, solves this problem.
- Where an array stores object references in consecutive memory locations, a linked list stores each object in a separate link.
- Each link also stores a reference to the next link in the sequence.
- In the Java programming language, all linked lists are actually doubly linked; that is, each link also stores a reference to its predecessor

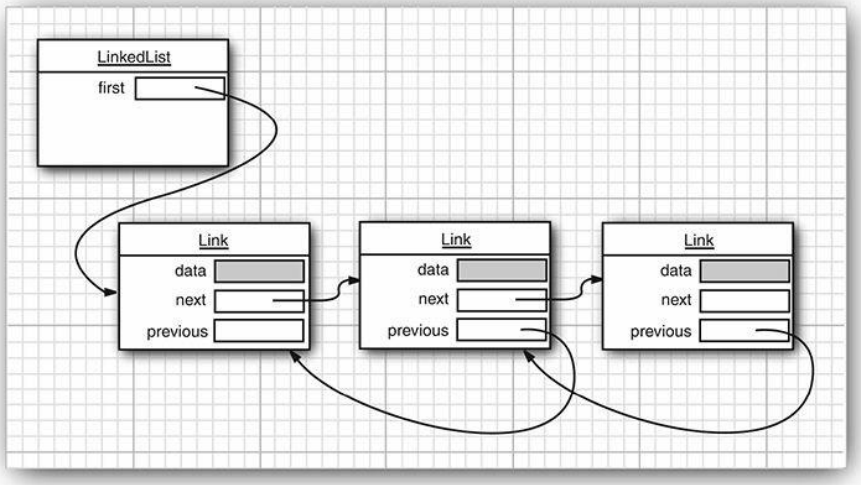


Figure 9.7 A doubly linked list

8.2.1 Linked Lists

- Linked list=chain of “links”:

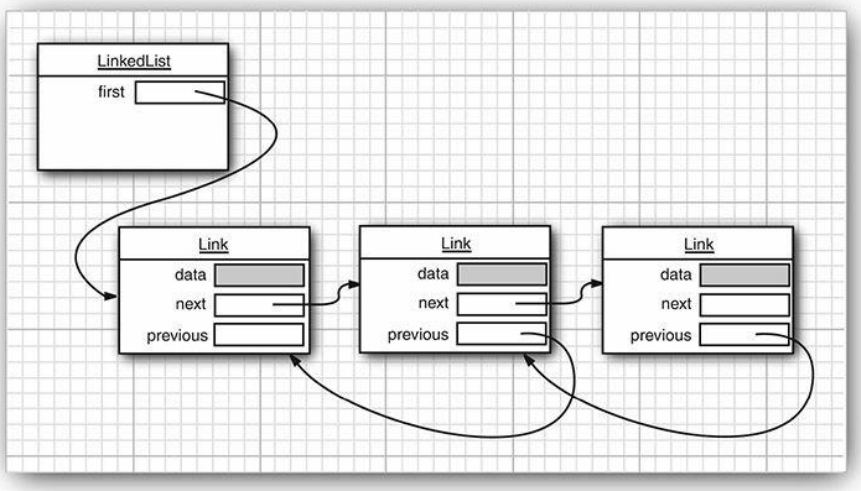


Figure 9.7 A doubly linked list

- Easy to remove in the middle:
  - Removing an element from the middle of a linked list is an inexpensive operation—only the links around the element to be removed need to be updated.

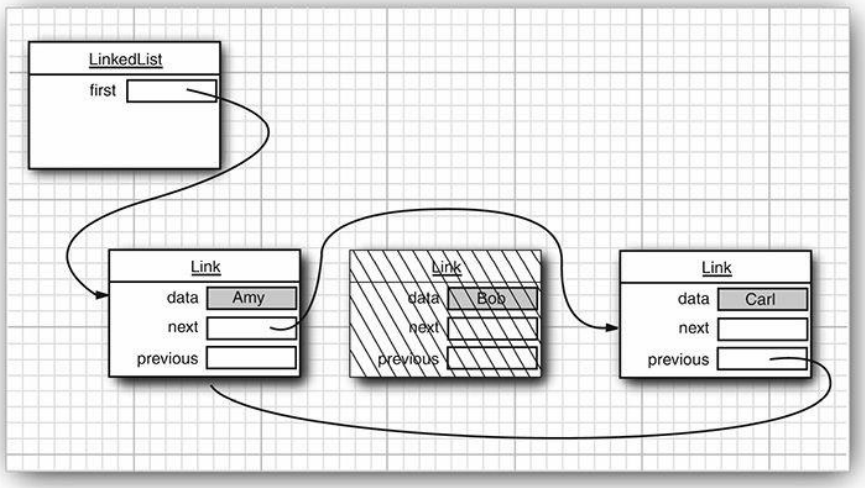


Figure 9.8 Removing an element from a linked list

8.2.1 Linked Lists

- Use the class `LinkedList` to *remove* and *add* elements in the linked list.

```
var staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator<String> iter = staff.iterator();
String first = iter.next();  // visit first element
String second = iter.next(); // visit second element
iter.remove();               // remove last visited element
```

  - The `LinkedList.add` method adds the object to the end of the list.
  - Use iterators to add elements in the middle of a list.
- The subinterface `ListIterator` contains an add method:

```
interface ListIterator<E> extends Iterator<E>{
    void add(E element);     //do not return a boolean
}
```

8.2.1 Linked Lists

- In addition, the `ListIterator` interface has two methods for traversing a list backwards.

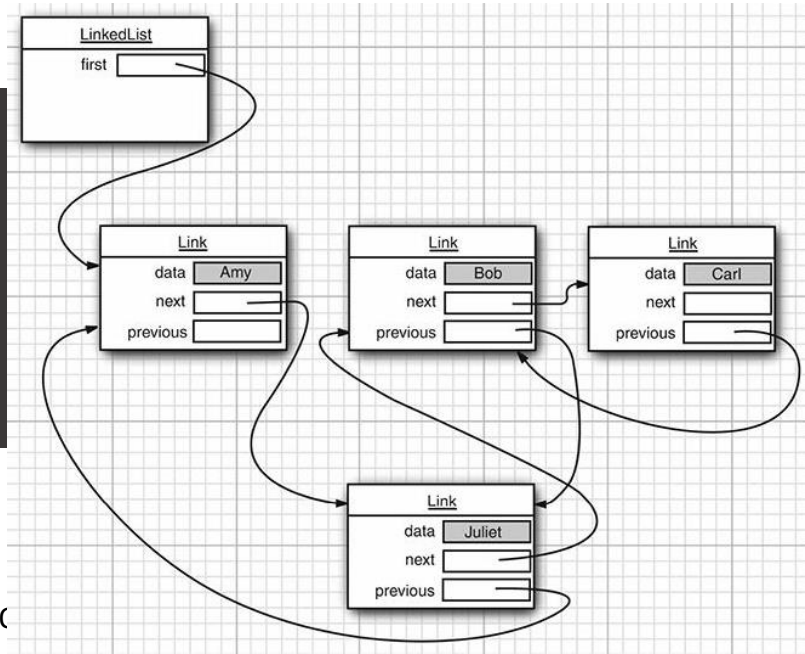
```
E previous()
boolean hasPrevious()
```

  - The `listIterator` method of the `LinkedList` class returns an iterator object that implements the `ListIterator` interface.

```
ListIterator<String> iter = staff.listIterator();
```

- The `add` method adds the new element before the iterator position.

```
var staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next();           // skip past first element
iter.add("Juliet");
```



8.2.1 Linked Lists

- A `set` method replaces the last element, returned by a call to next or previous, with a new element.

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // returns first element
iter.set(newValue);            // sets first element to newValue
```

- Linked list iterators detect concurrent modifications:

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next();          // throws ConcurrentModificationException
```

- The list and all iterators keep a **"modification count"**.
  - OK to have multiple readers and no writer.
  - OK to have one writer and no reader.

8.2.1 Linked Lists

- Remember to use a `ListIterator` to traverse the elements of the linked list in either direction and to add and remove elements.
- The `LinkedList` class supplies a `get` method that lets you access a particular element:

```
LinkedList<String> list = . . .;
String obj = list.get(n);
```

- The code is staggeringly inefficient.

```
for (int i = 0; i < list.size(); i++) {
    do something with list.get(i);}
```

The only reason to use `linkedList` is to minimize the cost of insertion and removal in the middle of the list. If you want random access into a collection, use an array or `ArrayList`, not a linked list.

8.2.2 Array Lists

- `ArrayList` is the other concrete implementation of the `List` interface which encapsulates a dynamically reallocated array of objects.
  - No need to use iterators since you have efficient random access with methods `get` and `set`.
- They are lists, so you may want to save references in `List` variables:

```
List<String> names = new ArrayList<>();
```

- Moment of truth: **You won't use linked lists much. Most of the time, an array list is fine.**
- Some methods give you a List value:

```
List<String> names = Arrays.asList("Peter", "Paul", "Mary");
```

  - It's a list, but you don't know which kind.

8.2.3 Hash Sets

- A well-known data structure for finding objects quickly is the ***hash table***.
  - A hash table computes an integer, called the *hash code*, for each object. A hash code is somehow derived from the instance fields of an object.
- Hash table uses hash codes to group elements into buckets:

String	Hash Code
"Lee"	76268
"lee"	107020
"eel"	100300

Table 9.2 Hash Codes Resulting from the hashCode Method

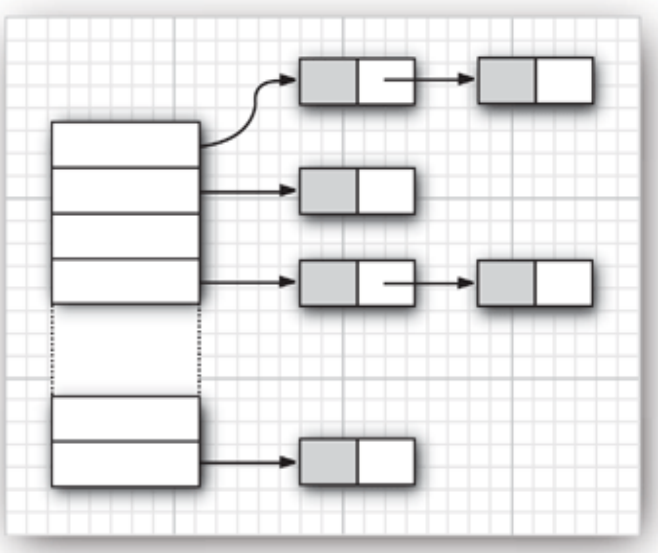


Figure 9.10 A hash table

8.2.3 Hash Sets

- Important notes:
  - If `a.equals(b)`, then `a` and `b` must have the same hash code.
  - Hit a bucket that is already filled - ***hash collision***.
  - Compare the new object with all objects in that bucket to see if it is already present.
  - If too many elements are inserted into a hash table, the number of collisions increases, and retrieval performance suffers.
  - Hash tables can be used to implement several important data structures: the *set* type.
    - The hash set iterator visits all buckets in turn.

8.2.4 Tree Sets

- Tree sets visit elements in sorted order.
  - You insert elements into the collection in any order.
  - When you iterate through the collection, the values are automatically presented in sorted order.
  - For example, suppose you insert three strings and then visit all elements that you added.

```
var sorter = new TreeSet<String>();
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.out.println(s);
```

  - Then, the values are printed in sorted order: Amy Bob Carl. As the name of the class suggests, the sorting is accomplished by a tree data structure.
  - Every time an element is added to a tree, it is placed into its proper sorting position.

8.2.4 Tree Sets

- In practice, a bit slower than hash sets.
  - Adding an element to a tree is slower than adding it to a hash table—see Table 9.3 for a comparison. But it is still much faster than checking for duplicates in an array or linked list.
  - Therefore, performance is guaranteed, whereas hash sets can perform poorly when the hash function does not scramble values well.

Table 9.3 Adding Elements into Hash and Tree Sets

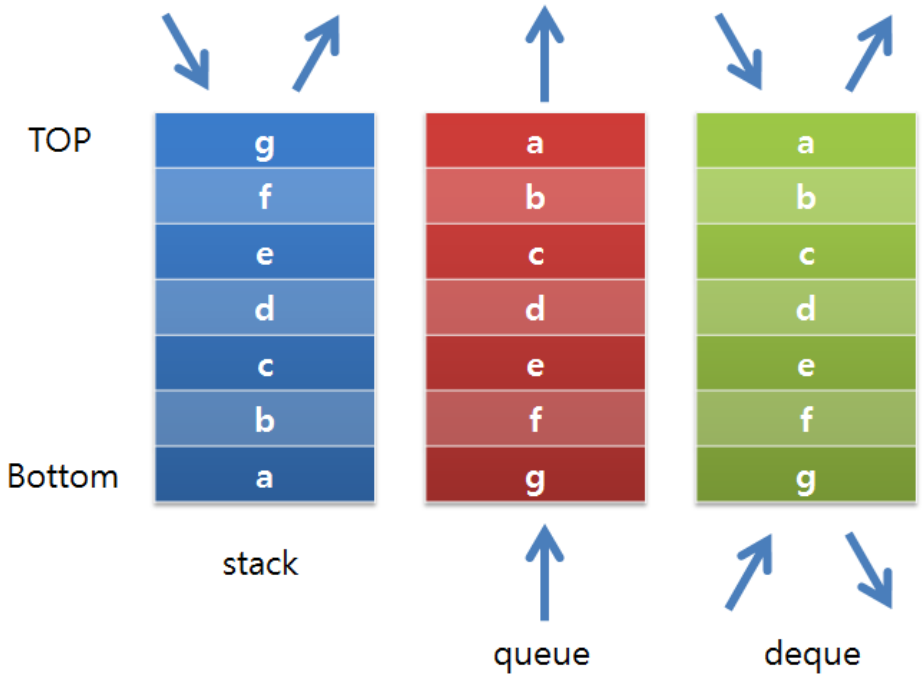
Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

- Tree set needs total ordering - not always easy to find.
  - In a total ordering, two elements compare identically only when they are equal.

Use tree sets when your elements are comparable, and you need traversal in sorted order.

8.2.5 Queues and Deques

- A `queue` can add elements at the tail and remove elements from the head.
- A double-ended queue, or `deque`, can add or remove elements at the head and tail.
  - `Deque` interface are implemented by the `ArrayDeque` and `LinkedList` classes.
  - Both of which provide deques whose size grows as needed.



8.2.6 Priority Queues

- **A priority queue retrieves elements in sorted order after they were inserted in arbitrary order.**
  - Makes use of an elegant and efficient data structure **heap**.
  - A heap is a self-organizing binary tree in which the `add` and `remove` operations cause the smallest element to gravitate to the root, without wasting time on sorting all elements.
- It can either hold elements of a class that implements the `Comparable` interface or a `Comparator` object you supply in the constructor.
- A typical use is **job scheduling**.
  - Each job has a priority. When removing, the "highest priority" job is removed.

Contents

8.3 Maps

- Now, we know that the set is a collection that lets you quickly find an existing element. However, to search for an element, you need to have an exact copy of the element to find.
- That isn't a very common lookup—usually, you have some key information, and you want to look up the associated element.
- The map data structure serves that purpose.
- A map stores key/value pairs. You can find a value if you provide the key.
- For example, you may store a table of letters, where the keys are the ID numbers, and the values are letters.
- Next, we are about to learn how to work with maps.

key	value
1	A
2	B
3	C

8.3.1 Basic Map Operations

- A map stores key/value pairs.
  - `HashMap` hashes the keys, `TreeMap` organizes them in sorted order.
- Add an association to a map:

```
var staff = new HashMap<String, Employee>();
var harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
```

- Retrieve a value with a given key:

```
var id = "987-98-9996";
Employee e = staff.get(id); // gets harry
```

- The get method returns `null` if the key is absent. Better approach:

```
Map<String, Integer> scores = . . .;
int score = scores.getOrDefault(id, 0);
// gets 0 if the id is not present
```

8.3.1 Basic Map Operations

- **Keys must be unique.**
- The `put` returns the previous value associated with its key parameter.
- The `remove` method removes an element with a given key from the map.
- The `size` method returns the number of entries in the map.
- Easiest way to iterate over a map:

```
scores.forEach((k, v) ->
    System.out.println("key=" + k + ", value=" + v));
```

8.3.2 Updating Map Entries

- **Updating a map entry is tricky because the first time is special.**
- Consider updating a word count:

```
counts.put(word, counts.get(word) + 1);
```

- What if word wasn't present?

```
counts.put(word, counts.getOrDefault(word, 0) + 1);
```

- Another approach is to first call the `putIfAbsent` method.

```
counts.putIfAbsent(word, 0);
counts.put(word, counts.get(word) + 1);
    // now we know that get will succeed
```

- The `merge` method simplifies this common operation.

```
counts.merge(word, 1, Integer::sum);
```

  - If word wasn't present, put 1. Otherwise, put the sum of 1 and the previous value (combines the previous value and 1, using the Integer::sum function).

8.3.3 Map Views

- In the Java collections framework, a map isn't a collection.
  - But can obtain views of the map - objects that implement the `Collection` interface or one of its subinterfaces.

- Collections in Java (like List, Set, and Queue) represent a group of elements that can be stored and accessed. These elements are all typically of the same type.

- A Map, on the other hand, stores key-value pairs. It associates a key with a value and does not treat its elements as a simple collection of objects. A Map contains a set of keys and a set of values, but keys and values are treated differently in the map.

8.3.3 Map Views

- Three views:
  - the set of keys,
  - the collection of values (which is not a set), and
  - the set of key/value pairs.

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

8.3.3 Map Views

- To visit all keys, can use:

```
Set<String> keys = map.keySet();
for (String key : keys) {
    // do something with key
}
```

- If you want to look at both keys and values, you can avoid value lookups by enumerating the entries.

```
for (Map.Entry<String, Employee> entry : staff.entrySet()) {
    String k = entry.getKey();
    Employee v = entry.getValue();
    // do something with k, v
}
```

- When iterating over a Map, if you were to iterate over just the keys and then use map.get(key) to get the values, this would involve additional lookups for each value, which can be inefficient.
- However, by using entrySet(), you can access both the key and the value directly without needing to perform a separate lookup for the value. This is often more efficient, especially in large maps.

8.3.3 Map Views

- You can avoid the cumbersome `Map.Entry` by using a `var` declaration.

```
for (var entry : map.entrySet()){
    // do something with entry.getKey(), entry.getValue()
}
```

- Or simply use the `forEach` method:

```
map.forEach((k, v) -> {
    // do something with k, v
});
```

- **Calling remove on the key set removes the key and associated value from the map.**

8.3.4 Weak Hash Maps

- The garbage collector traces **live** objects.
  - As long as the map object is live, all buckets in it are live and won't be reclaimed.
  - Thus, your program should take care to remove unused values from long-lived maps.
- Or you can use a `WeakHashMap` instead which cooperates with the garbage collector to remove key/value pairs when the only reference to the key is the one from the hash table entry.
  - The `WeakHashMap` uses weak references to hold keys.
  - A `WeakReference` object holds a reference to another object - in our case, a hash table key.
  - The operations of the `WeakHashMap` periodically check that queue for newly arrived weak references.

8.3.5 Linked Hash Sets and Maps

- The `LinkedHashSet` and `LinkedHashMap` classes remember in which they were added.
- As entries are inserted into the table, they are joined in a doubly linked list.

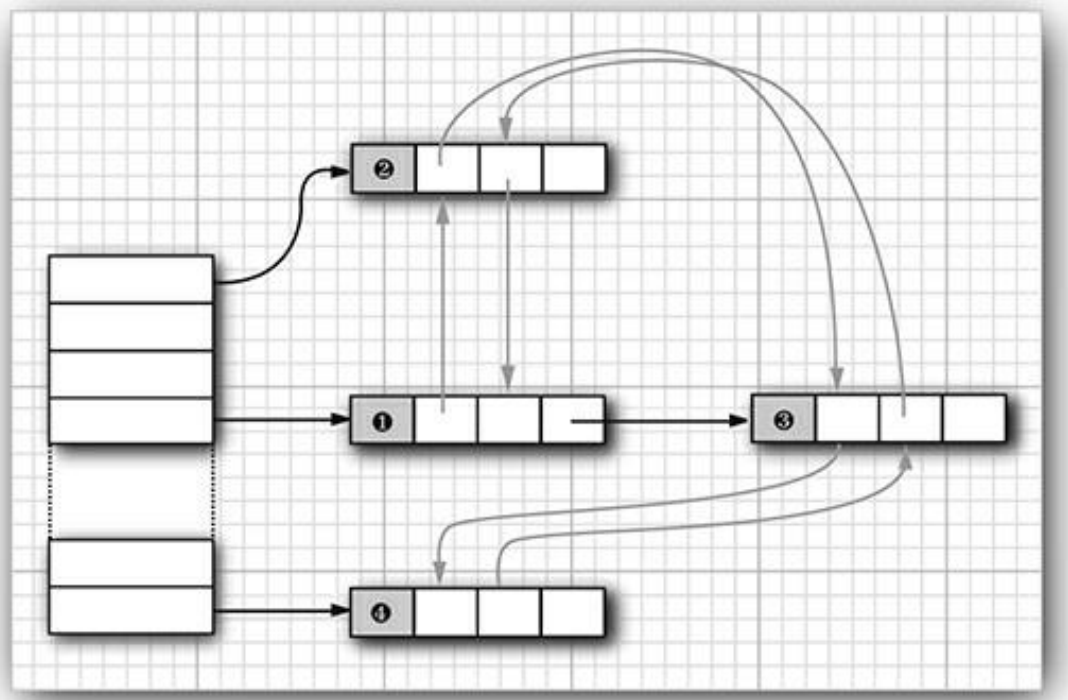


Figure 9.11
A linked hash table

8.3.5 Linked Hash Sets and Maps

- A linked hash map can alternatively use ***access order***, not insertion order, to iterate through the map entries.
- To construct such a hash map, call

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

- Access order is useful for implementing a "least recently used" discipline for a cache. Automate the process:

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

- Adding a new entry then causes the `eldest` entry to be removed whenever your method returns `true`.

```
var cache = new LinkedHashMap<K, V>(128, 0.75F, true) {
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > 100;
    }
};
```

8.3.6 Enumeration Sets and Maps

- The `EnumSet` is an efficient set implementation with elements that belong to an enumerated type.
- The `EnumSet` is internally implemented as a sequence of bits.
- The `EnumSet` class has no public constructors and use a static factory method to construct the set:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY}
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY,
    Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY,
    Weekday.WEDNESDAY, Weekday.FRIDAY);
```

- An `EnumMap` is a map with keys that belong to an enumerated type. Specify the key type in the constructor:

```
var personInCharge = new EnumMap<Weekday, Employee>
(Weekday.class);
```

8.3.7 Identity Hash Maps

- In `IdentityHashMap`, the hash values for the keys should not be computed by the `hashCode` method but by the `System.identityHashCode` method.

- For comparison of objects, the `IdentityHashMap` uses ==, not `equals`.
  - In other words, different key objects are considered distinct even if they have equal contents.

- This class is useful for implementing object traversal algorithms, such as object serialization, in which you want to keep track of which objects have already been traversed.

Recap

Main collection classes	Duplicate elements is allowed?	Elements are ordered?	Elements are sorted?	The collection is thread-safe?
ArrayList	Yes	Yes	No	No
LinkedList	Yes	Yes	No	No
Vector	Yes	Yes	No	Yes
HashSet	No	No	No	No
LinkedHashSet	No	Yes	No	No
TreeSet	No	Yes	Yes	No
HashMap	No	No	No	No
LinkedHashMap	No	Yes	No	No
Hashtable	No	No	No	Yes
TreeMap	No	Yes	Yes	No

https://www.codejava.net/java-core/collections/java-collections-framework-summary-table

Top Hat Question

Q. What does the put() method of HashMap return?

A. The old value associated with the key

B. The new value associated with the key

C. True or false based on insertion success

D. Null if the key does not exist

Answer: A. The old value associated with the key

Explanation: If the key already exists, the put() method returns the previous value associated with the key, or null if the key was not previously mapped.

Top Hat Question

Q. Which of the following classes does NOT implement the List interface?

A. ArrayList

B. LinkedList

C. Vector

D. HashSet

Answer: D. HashSet

Explanation: HashSet is a Set implementation, which does not allow duplicates and does not implement the List interface. The others (ArrayList, LinkedList, Vector) are List implementations.

Top Hat Question

Q. Which of the following is the correct way to iterate over a Map?

A. for (Map.Entry<K, V> entry : map) {}

B. for (Map<K, V> entry : map.entrySet()) {}

C. for (Map.Entry<K, V> entry : map.entrySet()) {}

D. for (Map.Entry<K, V> entry : map.values()) {}

Answer: C. for (Map.Entry<K, V> entry : map.entrySet()) {}

Explanation: To iterate over a Map, you can use map.entrySet(), which returns a set of Map.Entry objects (key-value pairs).

Top Hat Question

Q. Which method in LinkedHashMap can be overridden to implement cache eviction (like LRU)?

A. removeEldestEntry()

B. clear()

C. containsKey()

D. get()

Answer: A. removeEldestEntry()

Explanation: The removeEldestEntry() method can be overridden to implement eviction logic when the map exceeds a certain size.

Top Hat Question

Q. Which collection class should be used when you need a queue that processes elements in priority order?

A. LinkedList

B. PriorityQueue

C. ArrayList

D. TreeMap

Answer: **B**. PriorityQueue

Explanation: PriorityQueue processes elements based on their priority, not the order they were inserted, which is useful for scheduling or processing tasks in order of importance.

Top Hat Question

Q. A LinkedHashMap maintains the order of key-value pairs as they are inserted.

A. True

B. False

Answer: **B**. True

Explanation: A LinkedHashMap maintains insertion order of key-value pairs, meaning it preserves the order in which elements were added to the map.