Object Oriented Programming

Chapter 8
Collections

Dr. Helei Cui

25 Apr 2022

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 Java Collections Framework
- 8.2 Concrete Collections
- 8.3 Maps

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.

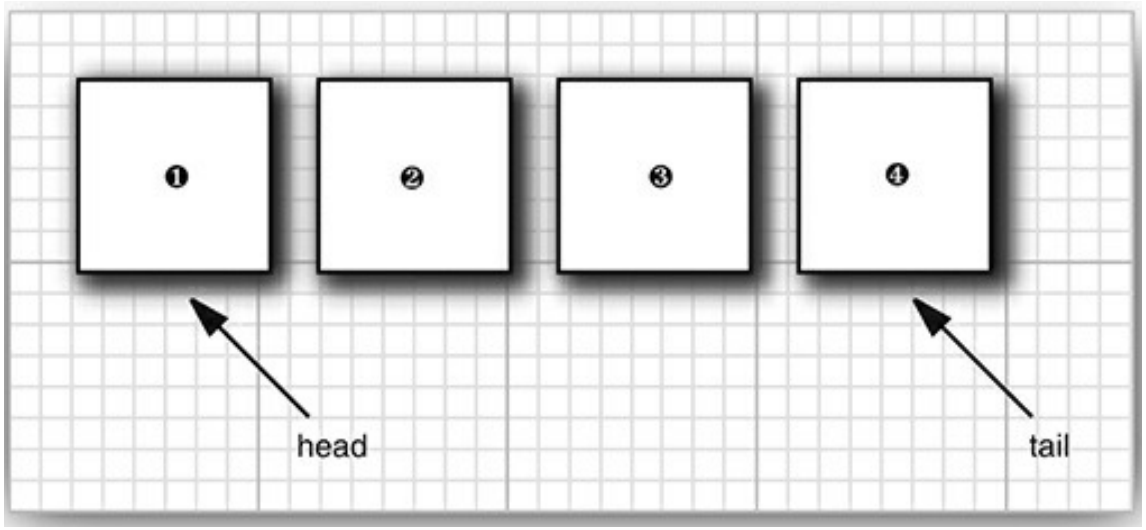
- As of Java 1.2, the designers felt that the time had come to roll out a full-fledged set of data structures.
  - The library should be ***small and easy to learn***.

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();
}
```

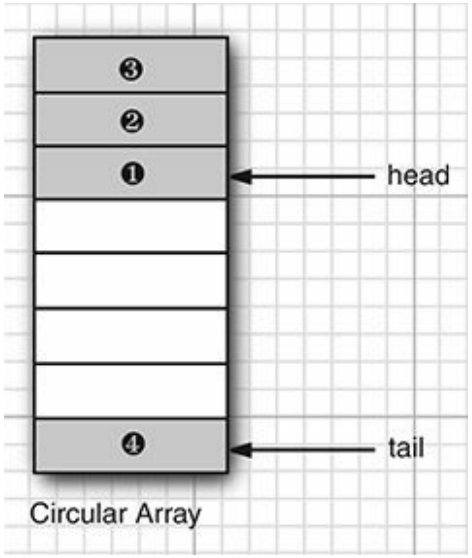
"first in, first out"

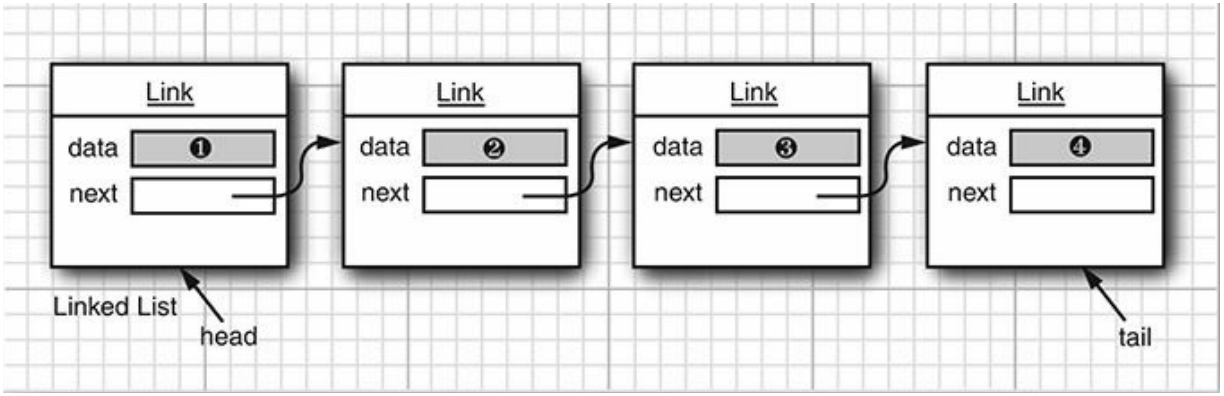


8.1.1 Separating Collection Interfaces and Implementation

- 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

- 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);
}
```

- 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
}
```

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();
    . . .
}
```

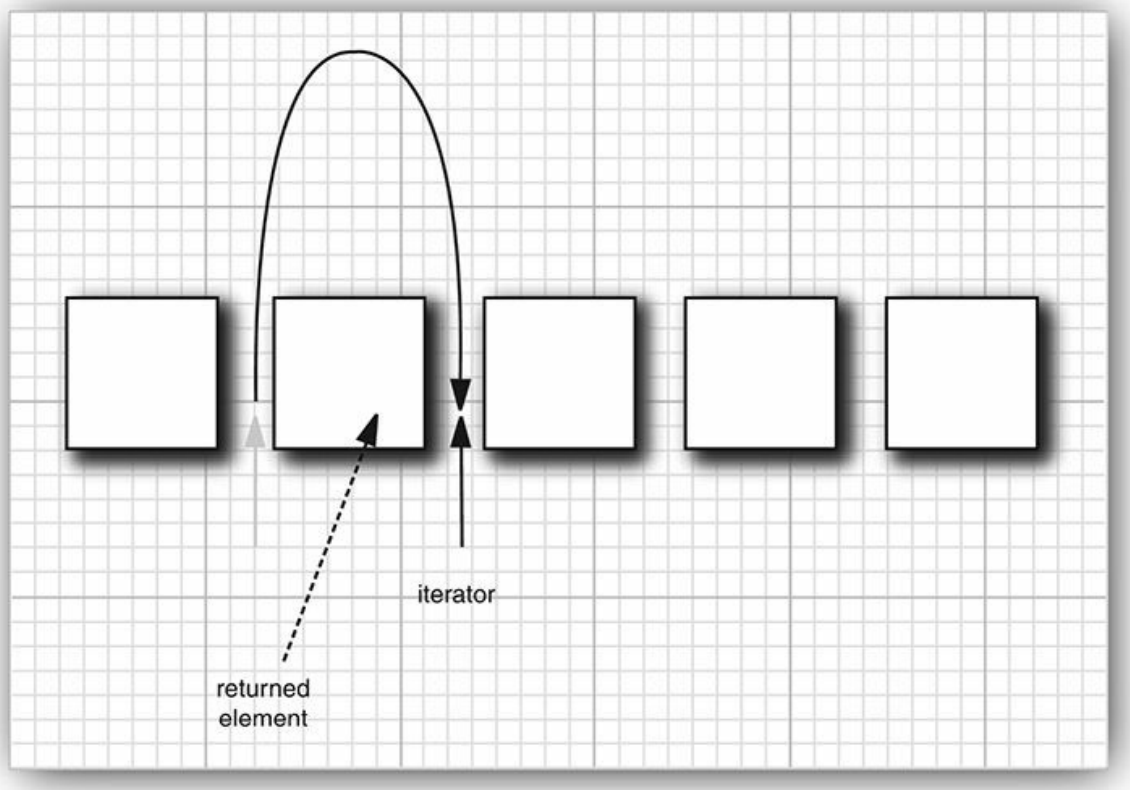
- The `Collection` interface extends the `Iterable` interface.
- Or without any loop:

```
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

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

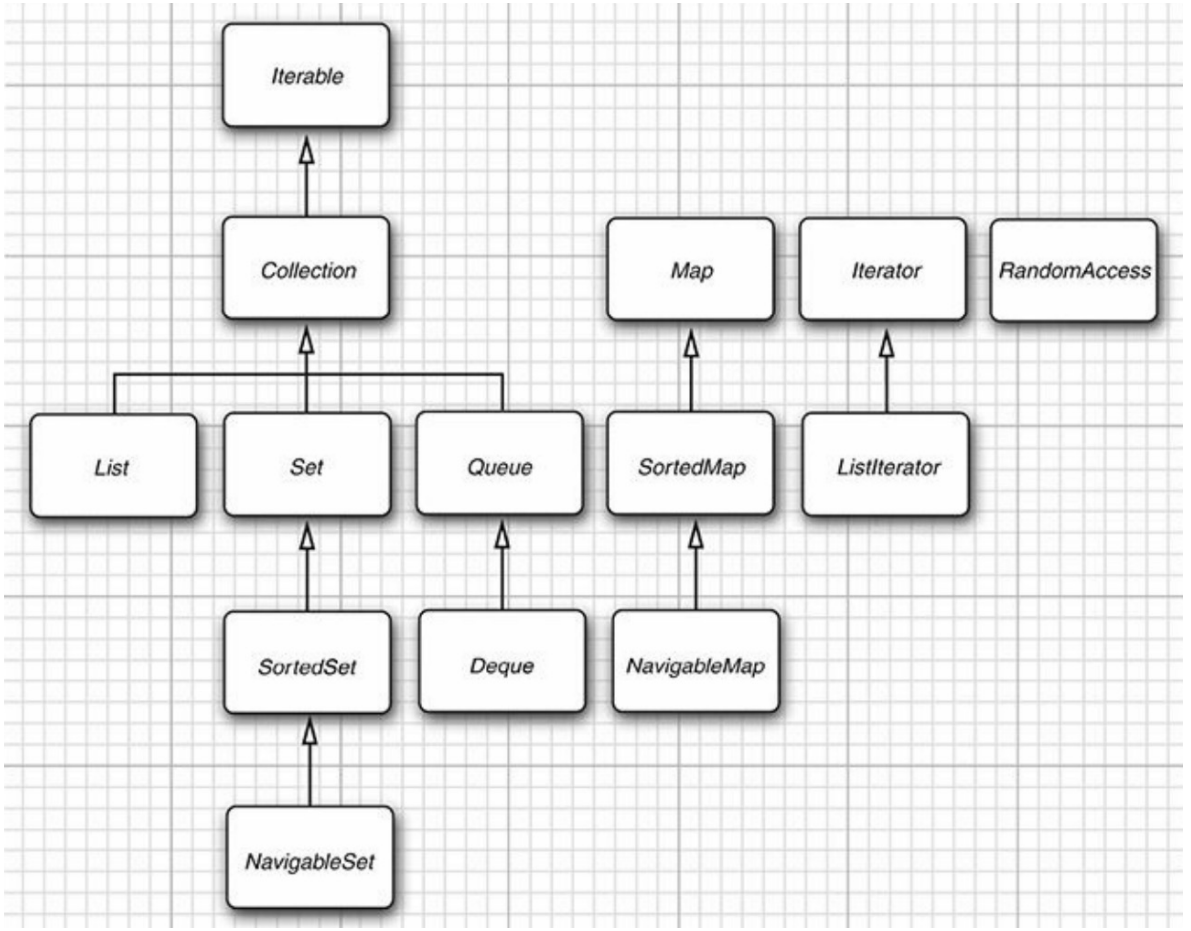
```
public abstract class AbstractCollection<E>
     implements Collection<E> {
     . . .
     public abstract Iterator<E> iterator();
     public boolean contains(Object obj) {
          for (E element : this)              // calls iterator()
               if (element.equals(obj))
               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

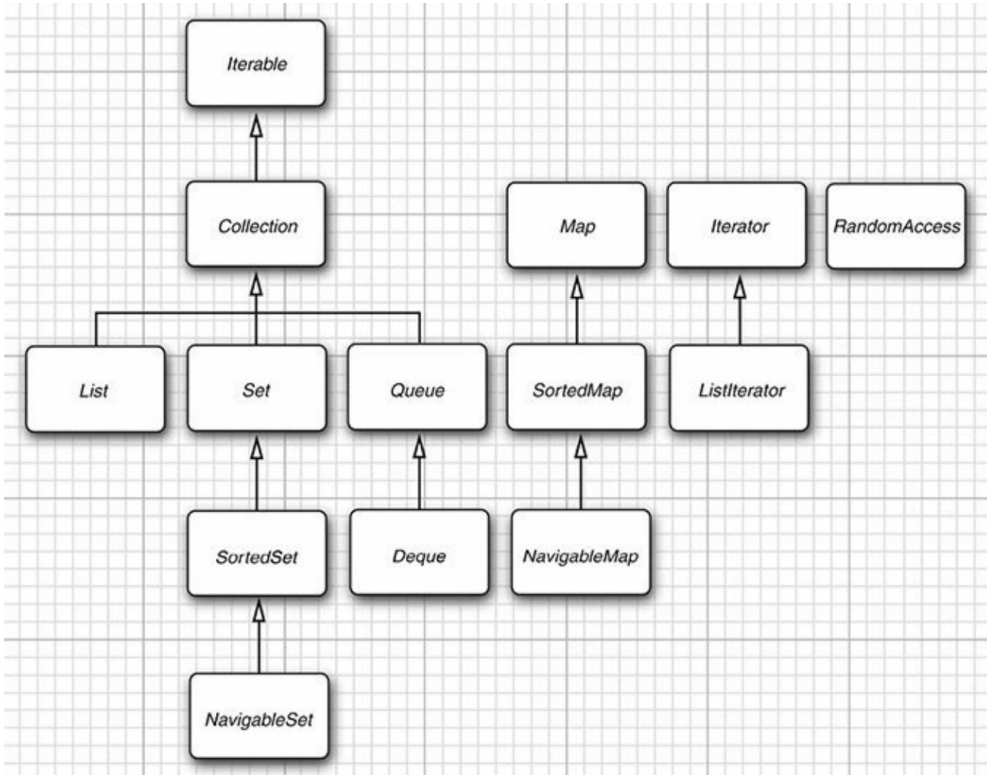
- The Java collections framework defines a number of interfaces for different types of collections.

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.

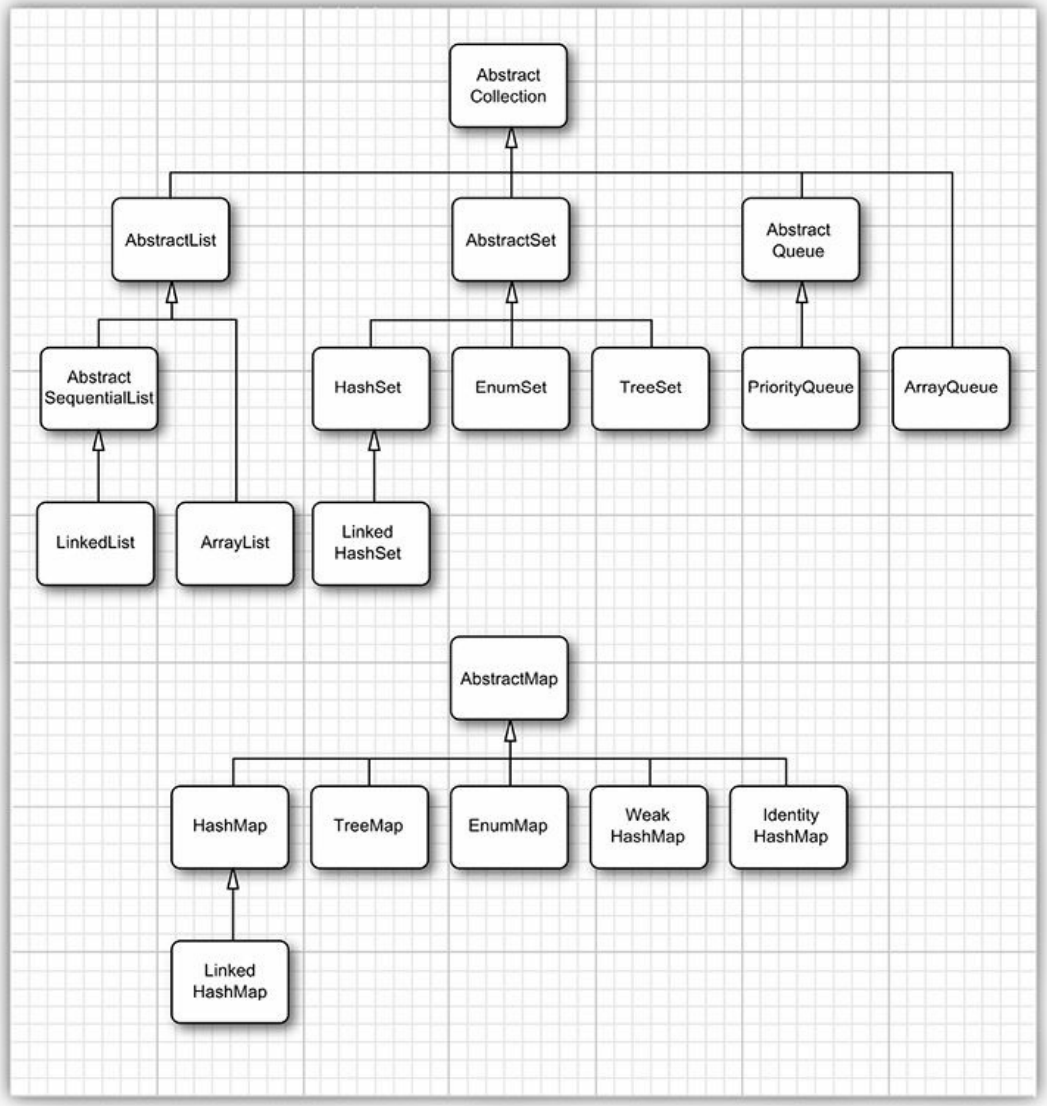


Contents

- 8.1 Java Collections Framework
- 8.2 Concrete Collections
- 8.3 Maps

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.

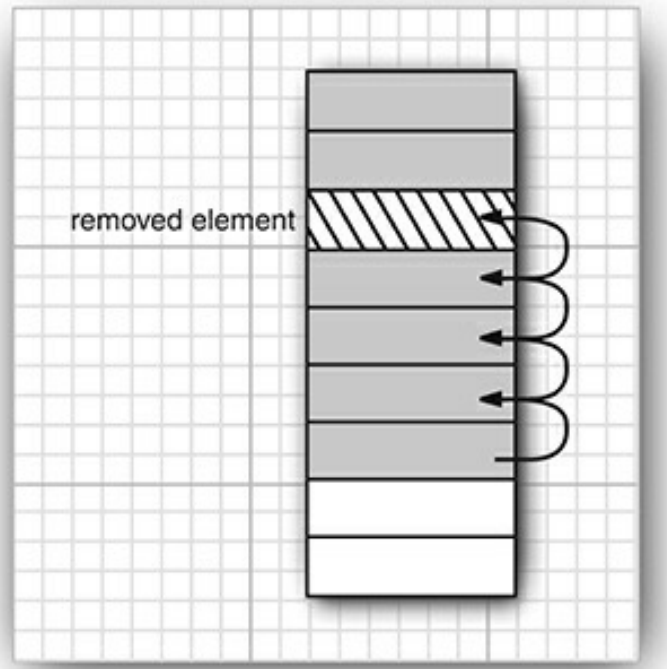


Figure 9.6 Removing an element from an array

8.2.1 Linked Lists

- Linked list=chain of “links”:

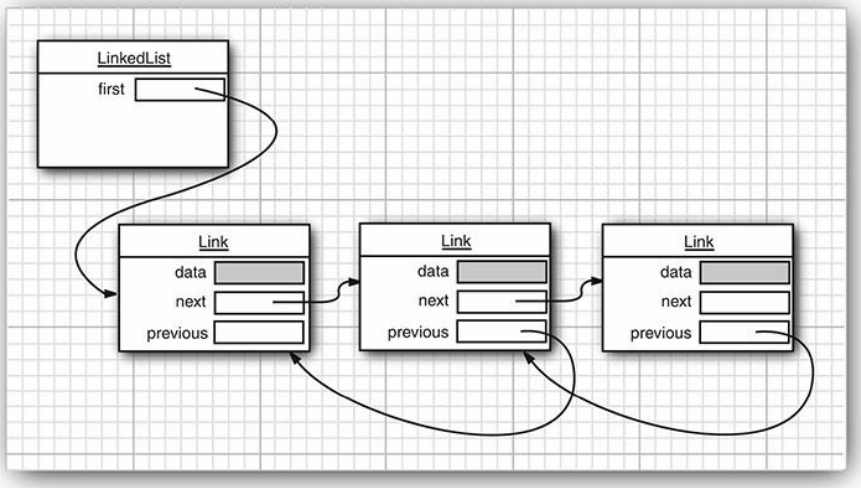


Figure 9.7 A doubly linked list

- Easy to remove in the middle:

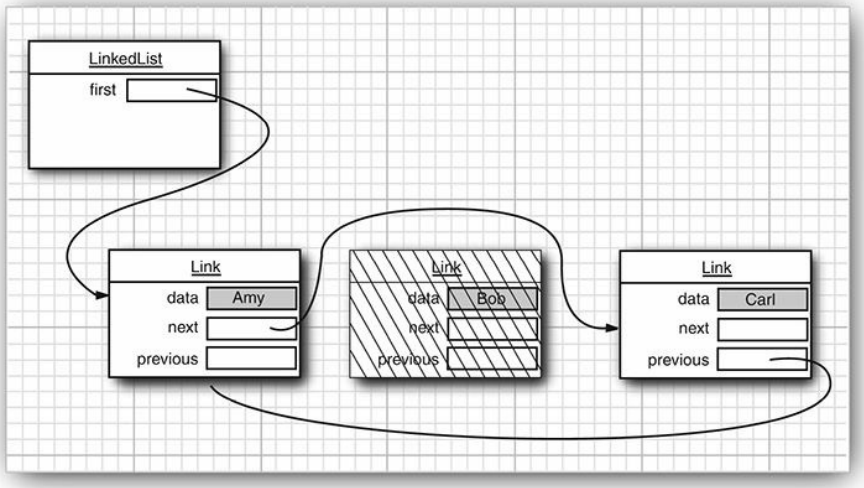


Figure 9.8 Removing an element from a linked list

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.

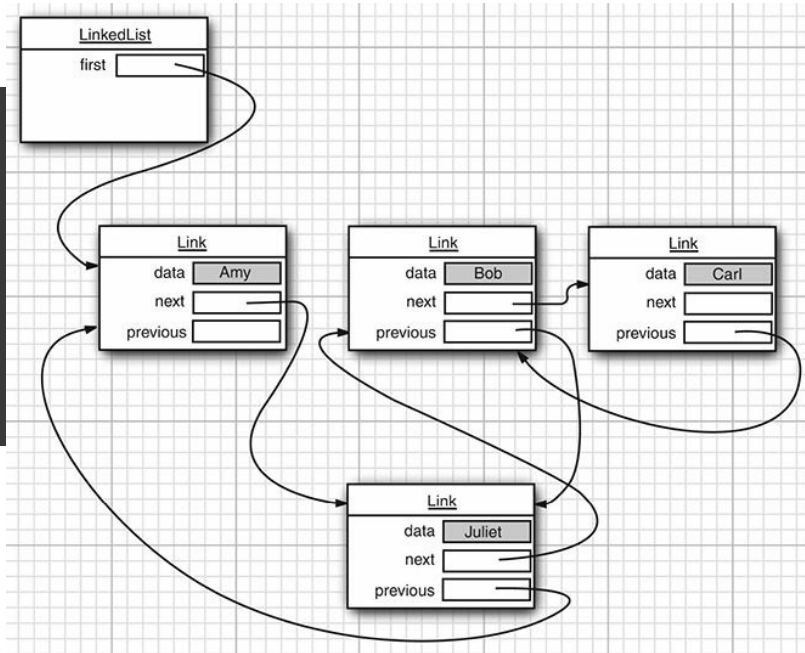
```
E previous()
boolean hasPrevious()
```

  - 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

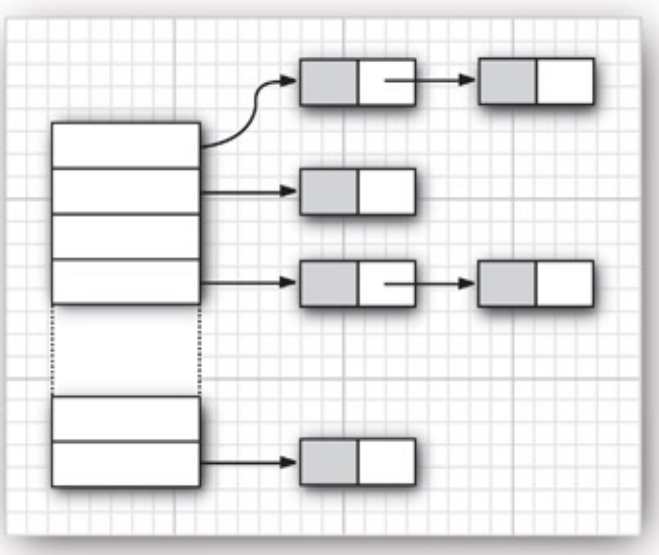


Figure 9.10 A hash table

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.

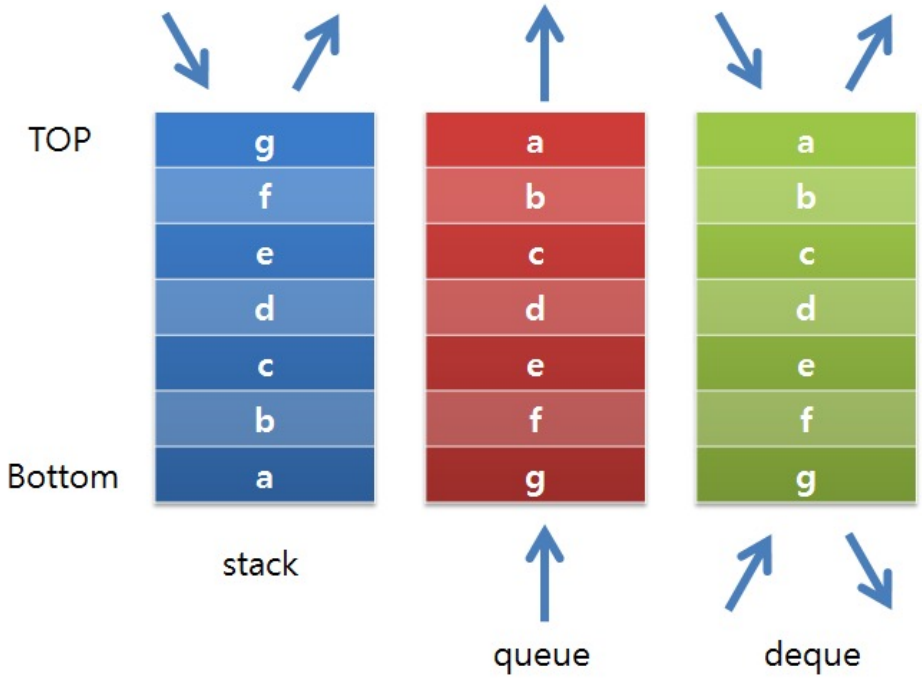
8.2.4 Tree Sets

- Tree sets visit elements in sorted order.
  - Every time an element is added to a tree, it is placed into its proper sorting position.
- In practice, a bit slower than hash sets.
  - But performance is guaranteed, whereas hash sets can perform poorly when the hash function does not scramble values well.
- 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

- 8.1 Java Collections Framework
- 8.2 Concrete Collections
- 8.3 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.

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.
- 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
}
```

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.

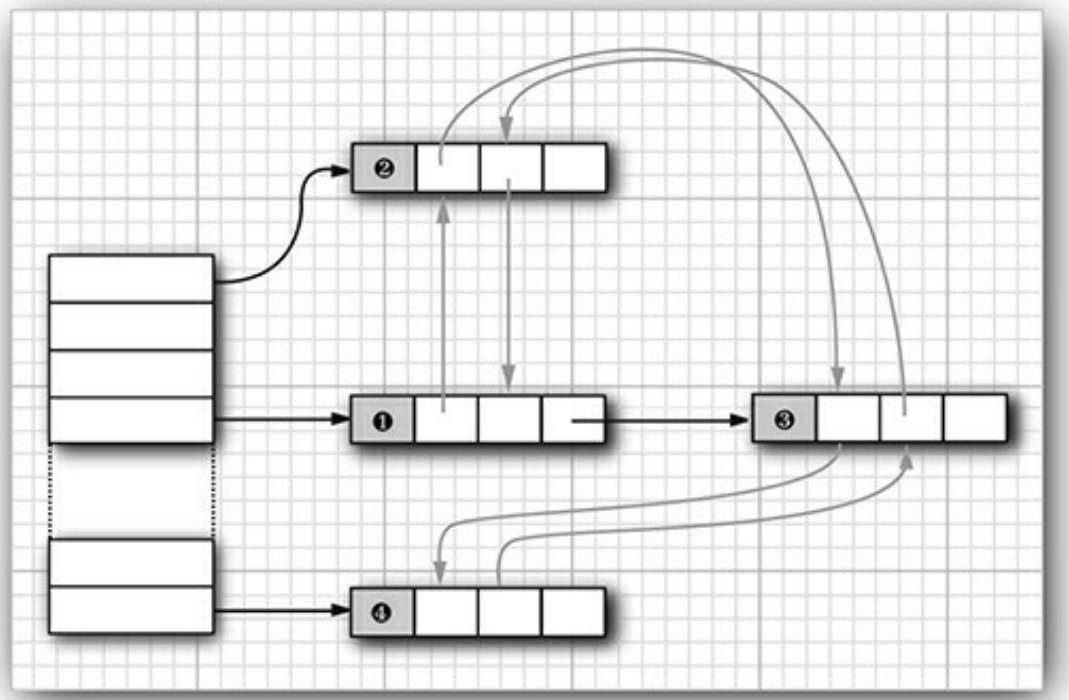


Figure 9.11
A linked hash table

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