Estructures de dades i algorismes

Tema 1 Introducció: disseny i ús de la Jerarquia Java d'una EDA

Curs 2018-2019

Introducció

Motivació

- El ordinadors s'usen per a manipular col·leccions de dades dinàmiques de talla elevada. El codi ha de ser:
  - **Eficient**: amb el menor cost temporal i espacial possible.
  - **Reutilitzable** en la mesura del possible.
- És necessari conèixer els **tipus abstractes de dades** i les diferents alternatives per a la seva implementació, és a dir, les **estructures de dades**.

Objectius d'aquest tema

- Definir les estructures de dades (EDAs) més usades i els algorismes que s'apliquen a elles.
- Revisar les eines que ofereix Java per al disseny i ús eficient d'una EDA.
  - Definició i ús de la jerarquia Java d'una EDA de tipus genèric.
  - Jerarquia `java.util.Collection`, la definició en l'estàndard de Java de les EDAs més freqüents.

Tipus abstractes i estructures de dades

Tipus abstracte de dades i EDAs

- Un tipus abstracte de dades (TAD o model) defineix:
  - Un conjunt de valors.
  - Les operacions suportades.
  - Les restriccions en el comportament.
  - **Tot això definit de manera independent de qualsevol representació.** Exemples:
    - Una *pila* o *stack* es defineix mitjançant les operacions de `apilar`, `desapilar`, `top`, `esBuida`.
    - Una *cua* es defineix per les operacions `encuar`, `desencuar`, `primer`, `esBuida`.
- Una estructura de dades (EDA) és una forma particular d'organitzar dades per implementar un tipus abstracte de dades.

Tipus abstracte de dades i EDAs

- Un TAD pot ser implementat usant diferents EDA's. Per exemple, la cua pot ser implementada usant una llista enllaçada, un vector, etc.
  - Podem implementar-los a partir de l'especificació.
  - Podem utilitzar-los sense saber com estan implementats.
- No sempre una EDA és millor que una altra en termes *absoluts*.
- La més adequada dependrà de si volem prioritzar espai o temps, de quines operacions es realitzen més freqüentment, etc.
- Una EDA pot ser usada per a diferents TAD's. Exemple: Els arbres binaris de cerca implementen operacions per al tipus de dada diccionari (o `Map`) i per a una cua de prioritat.

Disseny d'una EDA en Java

Disseny d'una EDA en Java

Les usarem per a especificar una interfície amb els mètodes que ofereix, acompanyats de comentaris explicatius (comportament i restriccions).

Exemple:

```
public interface Cua<E> {
  // insereix l'Element e en una Cua, o el situa a la fi
  void encuar(E e);
  // si !esBuida(), obte i elimina de la cua
  // l'Element que ocupa el principi
  E desencuar();
  // si !esBuida(), obte l'Element que ocupa el
  // principi, el primer en ordre D'Insercio
  E primer();
  // comprova si la cua esta buida
  boolean esBuida();
}
```

Interfícies en Java

- En Java no existeix herència múltiple general (una classe solament pot fer `extends` d'una altra) però una classe pot implementar (`implements`) tantes interfícies com siga necessari.
- Les interfícies no permeten definir atributs, únicament capçaleres de mètodes. Que una classe implemente una interfície és garantia que ofereix aquests mètodes.
- Diverses classes que no tenen relació entre si a nivell de jerarquia poden, no obstant això, implementar una mateixa interfície.
- Igual que una classe, una interfície pot ser genèrica (com en l'exemple anterior), recorda que la genericitat permet restriccions com en aquest exemple:

```
public interface CuaPrioritat<E extends Comparable<E>> {
...
}
```

Interfícies en Java

Exemple: ens donen una classe `Figura` i una interfície `Volum`:

```
public abstract class Figura {
   public abstract double area();
}
public interface Volum {
   double volum();
}
```

no totes les `Figura` implementen `Volum`:

```
public class Triangle extends Figura {
  ...
  public double area() { return base*altura/2.0; }
}
public class Esfera extends Figura implements Volum {
  ...
  public double area() { ... }
  public double volum() { ... }
}
```

Interfícies en Java

En aquest exemple, si ens passen una referència a `Figura` podem fer alguna cosa així:

```
Figura f = ... ; /* f referencia a una Figura */
if (f instanceof Volum) {
   double v = ((Volum)f).volum();
   ...
}
```

- En una variable de tipus `Figura` podria haver alguna cosa amb o sense volum, podem usar `instanceof` per esbrinar-ho. És necessari un càsting a `Volum` per usar `volum()` doncs aquest no està en la classe `Figura`.
- En una variable de tipus interfície `Volum` podem rebre una referència a un objecte que no sabem de quina classe és però sí que és d'una classe que implementa el mètode `volum()`

Extensió d'interfícies

Si necessitem una extensió d'un tipus abstracte de dades (necessitem operacions que no figuren en ell), podem ampliar la interfície via herència dissenyant una subinterfície que afegisca la nova funcionalitat.

Exemple: necessitem el mètode `talla` en la classe `Cua`:

```
interface CuaExt<E>extends Cua<E> {
// retorna el nombre d'elements continguts
   int talla();
}
```

- `CuaExt` hereta la llista de mètodes de `Cua` i afegeix un de més (`talla`).
- Si una classe `implements CuaExt` també implementa la interfície `Cua`.

Organització de jerarquies de EDAs en packages

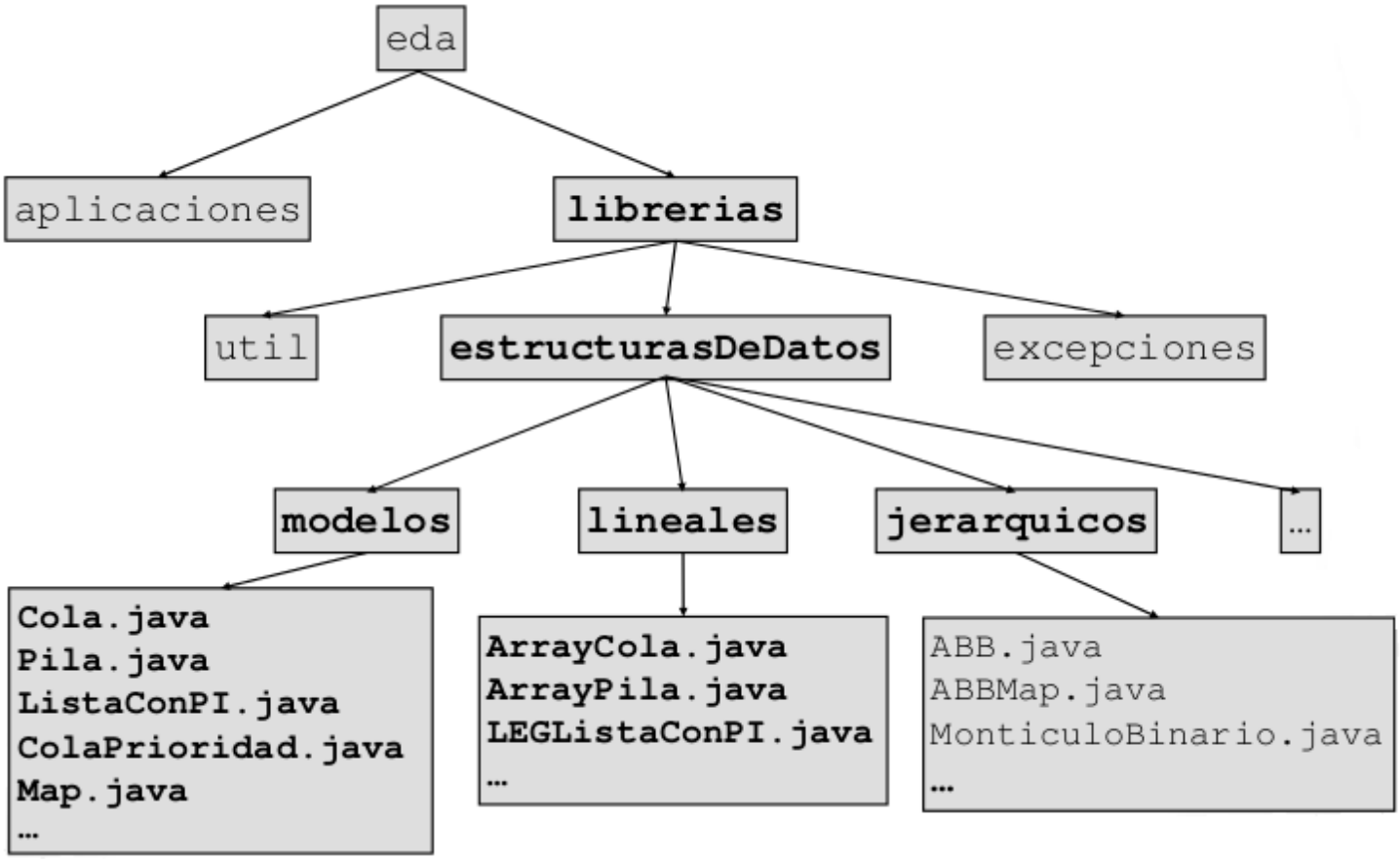

Figura: Paquets d'algunes classes utilitzades en l'assignatura

Paquets i visibilitat (repàs)

- És recomanable dividir el codi en **paquets** per controlar l'accés a classes i interfícies.
  - Permet estalviar temps de compilació davant modificacions,
  - afavoreix l'eficiència del treball en equip i
  - evita conflictes amb identificadors.
- Els *paquets* són directoris que contenen fitxers amb classes precompilades (`.class`) seguint una jerarquia diferent a la seguida en l'herència. Normalment se segueix un criteri de cohesió entre funcionalitats de les classes.
- Les classes poden importar-se individualment o per paquets afegint el comodí `*` al final de la instrucció per indicar que es desitja tenir accés a totes les classes del paquet. Les importacions s'expliciten al principi del fitxer. Per exemple:

```
import librerias.estructurasDeDatos.modelos.*;
...
```

Paquets i visibilitat (repàs)

- Podem definir més d'una classe en un mateix fitxer però només una pot ser pública, en aquest cas el nom del fitxer ha de coincidir amb el de la classe pública (amb l'extensió `.java`). La resta de les classes del fitxer només seran visibles dins del paquet.
- Quan es vol que una classe pertanya a una llibreria, el nom del paquet al que pertany s'ha d'especificar-se en la primera línia del fitxer.
- El paquet consisteix en el camí relatiu des del directori on es guarda tot el projecte fins al directori on es guardarà la classe.

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;
...
```

- L'accessibilitat a les classes i interfícies depèn (entre altres coses) del paquet en el qual es troben.

Paquets i visibilitat (repàs)

Sintaxi declaració d'una classe:

```
modifAcces modifClase class NomClase [extends NomClase]
                                     [implements llistaInterficies]
```

- **modifAcces** des d'on es pot accedir a l'ús de la classe:
  - `public`: la classe és visible des de qualsevol altra classe sense importar el paquet en el qual estiga.
  - *sense especificar*: accessible només a les classes del mateix paquet.
  - `private`: només és visible en la classe en la qual es defineix (és possible definir *classes internes* dins d'altres classes).
- **modifClase** afecta a les classes derivades:
  - `abstract`: per a les classes abstractes.
  - `final`: evita que la classe puga ser derivada.

Paquets i visibilitat (repàs)

Sintaxi declaració d'una classe:

```
modifAcces modifClase class NomClase [extends NomClase]
                                     [implements llistaInterficies]
```

- **Herència** tipus de classes de les quals es deriva:
  - `extends`: s'utilitza per indicar que la classe hereta de NomClase, en java només es permet heretar d'una única classe pare. En cas de no incloure la clàusula extends, s'assumirà que s'està heretant directament de la classe `java.lang.Object`.
  - `implements`: indica que esta classe és dels tipus d'interfície indicats per llistaInterficies, podent existir tants com vulguem separats per comes.

Paquets i visibilitat (repàs)

Sintaxi declaració de variables:

```
[modifVisibilidad] [modifAtribut] tipus nomVariable;
```

- **modifVisibilidad** delimita l'accés des de l'exterior de la classe.
  - `public`: accessible des de qualsevol classe.
  - `private`: només és accessible des de la classe on es defineix.
  - `protected`: accessible des de la classe on es defineix i les seues derivades.
  - sense especificar: accessible des de qualsevol classe del mateix paquet.
- **modifAtribut**:
  - `static`: la variable és comuna a tots els objectes de la classe.
  - `final`: el primer valor que rep la variable és inamovible.
  - `transient` i `volatile` no els anem a utilitzar.

Implementant una EDA: ArrayCua

Exemple: Implementació de la `Cua` en forma de vector (array) circular, la classe `ArrayCua`:

```
public class ArrayCua<E> implements Cua<E> {
  protected E elArray[];
  protected static final int CAPACITAT_PER_DEFECTE = 30000;
  protected int finalC, principiC, talla;
  public ArrayCua(){...}
  public String toString(){...}
  protected void duplicarArrayCircular(){...}
  protected int incrementar(int index){...}
  public void encuar(E e){...}
  public E desencuar(){...}
}
```

Implementant una EDA: ArrayCua

Exemple: Detall d'alguns mètodes:

```
public ArrayCua(){
    elArray = (I[]) new Object[CAPACITAT_PER_DEFECTE];
    talla = 0; principiC = 0; finalC = 0;
}
protected int incrementar(int index){
    if (++index==elArray.length) index = 0;
    return index;
}
public void encuar(I i){
    if ( talla==elArray.length ) duplicarArrayCircular();
         elArray[finalC] = i;
    finalC = incrementar(finalC); talla++;
}
```

Implementant una EDA: LegCua

Implementació de la `Cua` basant-nos en una llista enllaçada.

```
class NodeLEG<E> {
    E dada;
    NodeLEG<E> seg;
    // constructor
    NodeLEG(E dada, NodeLEG<E> seg) {
        this.dada = dada;
        this.seg = seg;
    }
    // un altre constructor
    NodeLEG(E dada) {
        this(dada, null);
    }
}
```

Implementant una EDA: LegCua

```
public class LEGCua<E> implements Cua<E> {
  protected NodeLEG<E> primer, fi;  // Atributs
  public LEGCua() { // Constructor
    primer = fi = null;
  }
  public void encuar(E x) { // insereix x a la fi
    if (primer == null) primer = fi = new NodeLEG<E>(x);
     else {
       fi.seg = new NodeLEG<E>(x);
       fi = fi.seg;
     }
  }
```

Implementant una EDA: LegCua

```
  // SII !esBuida(): elimina el primer
  public E desencuar() {
    E elPrimer = primer.dada;
    primer = primer.seg;
    if (primer == null) fi = null;
      return elPrimer;
  }
  // SII !esBuida(): torna la primera dada de la cua
  public E primer() {
      return primer.dada;
  }
  // comprova si la Cua esta o no buida
  public boolean esBuida() {
      return primer == null;
  }
}
```

Implementant EDA: llista amb punt d'interès

Descripció

- L'accés a la dada que ocupa el punt d'interès (PI) es realitza en temps constant.
- A l'inici de la gestió seqüencial el PI està situat sobre la primera dada de la llista.
- Per a accedir a la següent dada d'una llista durant la seua gestió seqüencial cal avançar el PI sobre ella.
- En inserir, recuperar o eliminar una dada d'una llista el PI roman inamovible.
- Si una llista està buida o bé l'accés seqüencial ha arribat a la seua fi no hi ha dada alguna que ocupe el PI.

Implementant EDA: llista amb punt d'interès

```
public interface LlistaAmbPI<E> {
  // insereix e abans del PI, que roman inalterat
  void inserir(E e);
  // SII !esFi(): elimina la dada que ocupa el seu PI
  void eliminar();
  // SII !esFi(): torna la dada que ocupa el PI
  E recuperar();
  // situa el PI de una llista en el seu inici
  void inici();
  // SII !esFi(): avanc,a el PI d'una llista
  void seguent();
  // comprova si el PI esta darrere de l'ultim element
  boolean esFi();
  boolean esBuida(); // comprova si la llista esta buida
  void fi(); // situa el PI darrere de l'ultim element
  int talla(); // torna la talla de la llista
}
```

Implementant EDA: llista amb punt d'interès

Suposem que tenim una llista `li` que emmagatzema cadenes i que conté (el punt d'interès està marcat en negreta):

creïlles, *cireres*, *llet*, *formatge*

Executem `li.seguent()`:

creïlles, ***cireres***, *llet*, *formatge*

S'insereix abans del PI, si executem `li.inserir("julivert")`:

creïlles, *julivert*, ***cireres***, *llet*, *formatge*

Si executem `li.eliminar()`:

creïlles, *julivert*, ***llet***, *formatge*

Implementant EDA: llista amb punt d'interès

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;

public class LEGLlistaAmbPI implements LlistaAmbPI {
    protected NodeLEG pri, ant, ult;
    public LEGLlistaAmbPI(){
        pri = ult = ant = new NodeLEG(null);
    }
    ...
}
```

Exercici: completa aquesta classe.

Ús de la jerarquia Java d'una EDA

- Mitjançant **composició**: declarar com a atribut de la classe una variable del tipus de dades que ens interesse.

  Quan necessitem l'ús concret del tipus abstracte de dades. Exemple: una classe gestor de pacients pot tenir una cua de pacients.

- Mitjançant **herència**: per a ampliar la jerarquia de la EDA.

  Quan el disseny exigeix operacions addicionals podem crear una subinterfície com l'exemple de la `CuaExt`.

- Ambdues aproximacions solen fan ús de paquets i de `import`.

Classes genèriques restringides per Comparable

Mètode equals

```
public boolean equals(Object obj)
```

- Definit en la classe `Object`, per la qual cosa està sempre disponible en objectes (java també té tipus primitius).
- En redefinir-ho reps un Object, normalment comproves que és del mateix tipus que la pròpia classe...

```
public class Rectangle extends Figura {
  protected double base, altura;
  ...
  public boolean equals(Object obj){
    return obj instanceof Rectangle &&
           this.base   == ((Rectangle) obj).base &&
           this.altura == ((Rectangle) obj).altura;
  }
}
```

Interfície Comparable

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

- Té un sol mètode `compareTo` que retorna un sencer negatiu, zero o positiu segons l'argument rebut siga menor, igual o major que el propi objecte.
- El que implemente aquesta interfície ha de ser responsable que es complisquen algunes propietats d'ordre (antisimètrica, transitiva).
- Recomanat que siga consistent amb `equals`, és a dir, que `i1.compareTo(i2)==0` done el mateix resultat que `i1.equals(i2)`.
- Permet que els objectes de classes que implementen aquesta interfície puguen ordenar-se en `Collections.sort`, `Arrays.sort`, que puguen ser usats com a claus en `sorted map` i en `sorted set`.

Interfície Comparator

```
public interface Comparator {
  // rep 2 objectes a ser comparats i retorna negatiu,
  // zero o positiu (igual que amb Comparable)
  int compare(T o1, T o2);
}
```

- Alguns algorismes d'ordenació permeten que li passem una instància adequada de `Comparator` en lloc d'usar *l'ordre natural* o inherent de la pròpia classe donat per `compareTo` (si la classe implementa la interfície `Comparable`).
- El motiu: poder ordenar una mateixa col·lecció per criteris diferents (ordenar persones alfabèticament en uns casos, per edat en uns altres, etc.).
- No ho anem a utilitzar en aquesta assignatura, ho hem afegit ací per completitut.

==, equals(), compareTo() y compare()

- L'ús de '==' i de '!=' compara tipus de dades **primitives**, per a objectes compara **referències**.
- `a.equals(b)` mira la igualtat dels valors. En ser un mètode definit en la classe `Object` està sempre disponible, encara que no sempre fa el que hauria, llevat que es redefinisca apropiadament.
- `a.compareTo(b)` per a les classes que implementen la interfície `Comparable` que defineix un ordre (permet saber menor que, igual, major que) inherent a cada classe.
- `compare(a,b)` de la interfície `Comparator` permet poder usar diferents ordres sobre objectes d'una mateixa classe (exemple: ordenar segons criteris diferents usarien instàncies diferents de `Comparator`.

Classes de tipus restringit per Comparable

Exemple: La classe Figura Comparable per la seua àrea.

```
public abstract class Figura implements Comparable<Figura>{
  ...
  public abstract double area();
  final public int compareTo(Figura f) {
    double areaDelAltre = f.area(),
           areaDAquest  = this.area();
    if (areaDAquest < areaDelAltre) return -1;
    if (areaDAquest > areaDelAltre) return  1;
    /* les arees son iguals */   return 0;
  }
}
```

Exemple: Cua de prioritat

Volem manipular la cua d'una impressora en la qual primer cal servir els treballs de major prioritat. En cas d'empat (en prioritat) es processa el més antic.

```
public interface CuaPrioritat<E extends Comparable<E>> {
   // Insereix x a la cua
   void inserir(E x);
   // SII !esBuida(): retorna la dada de major prioritat
   E recuperarMin();
   // SII !esBuida(): retorna i elimina la dada de major prioritat
   E eliminarMin();
   // Retorna true si la cua aquesta buida
   boolean esBuida();
}
```

Cua de prioritat

Al llarg de l'assignatura veurem estructures de dades que poden servir per a implementar aquesta interfície:

- Quin seria el cost de cada operació en una llista no ordenada?
- I en una llista ordenada?
- En el tema 4 veurem que els arbres binaris poden suportar aquestes operacions de manera relativament eficient.
- **PERÒ** l'estructura de dades més habitual per a implementar una cua de prioritat són els monticles o *heaps* que veurem en el tema 5.
- La classe `PriorityQueue` de la biblioteca estàndard de Java implementa una cua de prioritat basada en un heap.

Interfícies estàndard en la jerarquia de Java

Interfícies estàndard en la jerarquia de Java

Java conté en la seua biblioteca/llibreria `java.util` les jerarquies `Collection` i `Map` per a la representació i manipulació de col·leccions de dades. Les interfícies són els següents:

- `List`: col·lecció ordenada / seqüència.
- `Deque`: fusió dels models pila i cua.
- `Map`: diccionari (emmagatzemar parells *(clau,valor)* i poder cercar per la clau).
- `Set`: manipulació de conjunts.

Recorda que les classes i interfícies genèriques en Java no es poden instanciar amb tipus bàsics, si bé pots utilitzar les corresponents classes embolcall.

Interfícies estàndard en la jerarquia de Java

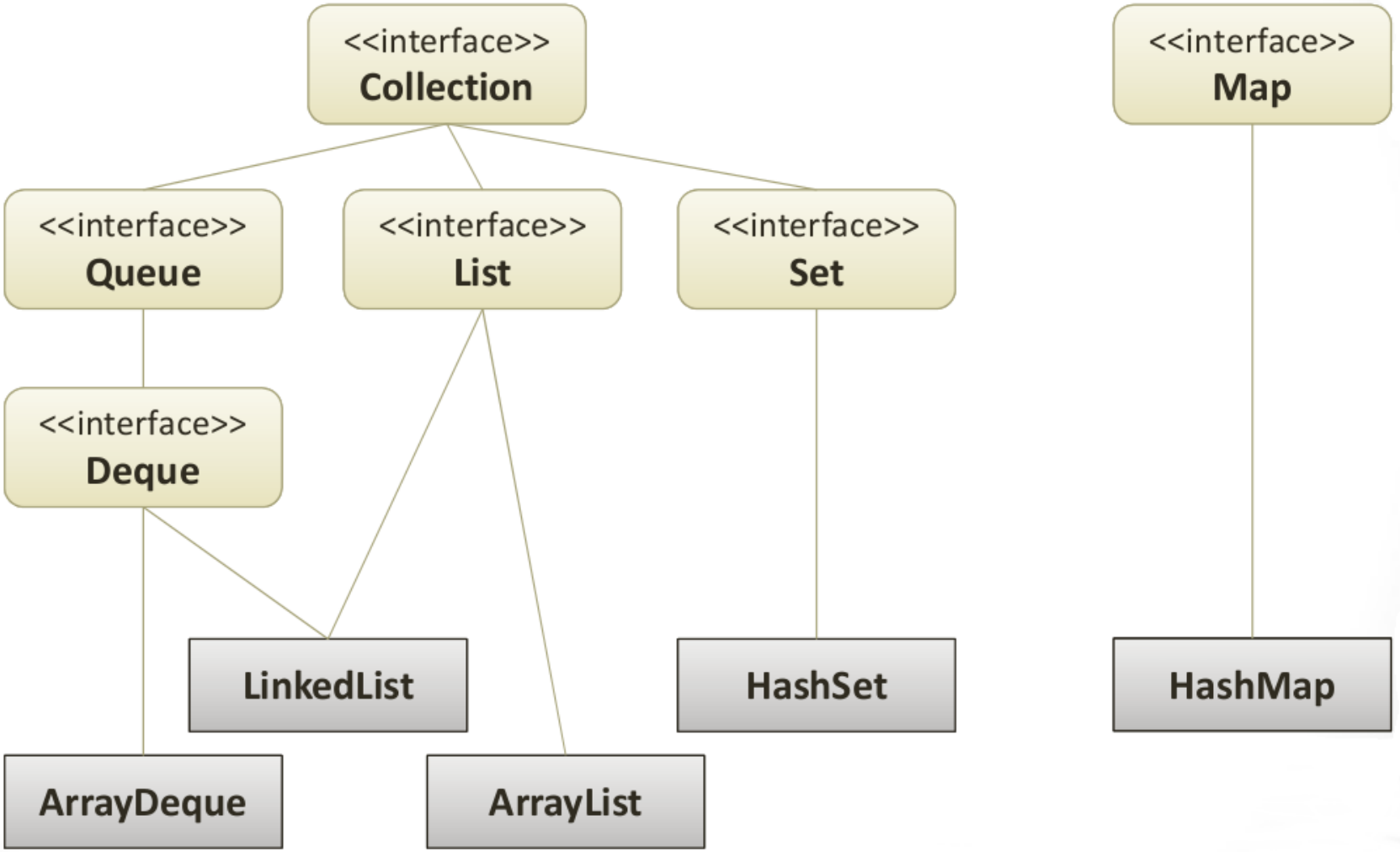

Figura: Les jerarquies Collection i Map

Collection

```
public interface Collection extends Iterable
```

- `Collection` representa un conjunt d'elements.
- Algunes col·leccions permeten elements duplicats, unes altres no (exemple: `Set`).
- En aquesta assignatura no veurem com s'implementen els iteradors, encara que sí hem vist alguna cosa bastant semblant: llistes enllaçades amb punt d'interès.
- Existeix una classe abstracta denominada `AbstractCollection` que ofereix una implementació mínima de la interfície `Collection` per a reduir el cost d'implementar aquesta interfície. D'aquesta manera, l'usuari solament ha de redefinir alguns mètodes.

List

- `List` serveix per a representar col·leccions ordenades (seqüències).
- Permet introduir i esborrar elements en qualsevol posició arbitrària, cercar un element pel seu valor o per la seua posició.
- Existeixen diverses implementacions:
  - `Vector` / `ArrayList`: Arrays redimensionables. Manté una capacitat que sempre ha de ser major o igual que la grandària actual de la llista. La diferència entre `Vector` i `ArrayList` té a veure amb la sincronització si s'accedeix des de diversos fils.
  - `LinkedList` és una llista enllaçada que també implementa la interfície `Deque`.

Deque

- `Deque` és l'abreviatura de *double ended queue*.
- Representa una col·lecció lineal que suporta insercions i esborrats eficients en tots dos extrems.
- Existeix una classe denominada `AbstractQueue` que deriva de `AbstractCollection`, implementa la interfície `Deque` i proporciona un esquelet per a implementar classes que implementen la interfície `Queue`.
- Existeixen diverses implementacions:
  - `ArrayDeque` s'implementa mitjançant un array redimensionable. La majoria de les operacions tenen un cost temporal **amortitzat** constant.
  - `LinkedList`, que ja hem vist en la transparència anterior.

Map (o diccionari)

- `Map` representa un objecte que associa claus a valors. No pot haver-hi claus repetides i cada clau té un sol valor.
- Permet veure els seus continguts com a conjunt de claus, col·lecció de valors o conjunt de parells *(clau,valor)*.
- Existeixen diverses implementacions:
  - `HashMap` / `HashTable`: basats en tècniques com les quals veurem en el tema 3, permeten costos constants *amortitzats* per a cercar, inserir i esborrar en condicions generals.
  - `TreeMap` requereix que els elements es puguen comparar, utilitza un arbre balancejat (ho veurem en tema 4). Garanteix un cost logarítmic per a cercar, inserir i esborrar.
  - `LinkedHashMap`: és una taula hash i llista enllaçada alhora (permet actuar com a diccionari per a cercar claus i recórrer els elements per l'ordre relatiu a les insercions).

Set (o conjunt)

- `Set` no permet elements duplicats.
- Existeixen diverses implementacions:
  - `HashSet`: utilitza una taula hash (un `HashMap`) com els que estudiarem en el tema 3.
  - `TreeSet`: es basa en un arbre `TreeMap` on els elements s'ordenen (ordre proporcionat per `Comparable` o mitjançant `Comparator`, aquest ordre ha de ser compatible amb `Equals`).
  - `LinkedHashSet` és com `LinkedHashMap` en la mesura en què implementa la interfície `Set` i al mateix temps permet iterar sobre els elements de manera predictible (ordre d'inserció).

Bibliografia

Bibliografia

- *Data structures, algorithms, and applications in Java* (Sahni, Sartaj) Universities Press.
- *Algorithms in Java. Third edition.* R. Sedgewick. Addison-Wesley, 2003. ISBN: 0-201-36120-5.
- *Introduction to algorithms* (Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald; Stein, Clifford). MIT Press
- *Fundamentos de algoritmia* (Brassard, Gilles). Pearson Prentice Hall.
- *Data structures and algorithms in Java* (Goodrich, Michael T.). John Wiley & Sons, Inc.
- The JavaTM Tutorials: Collections
- La subjerarquía Queue de Collection en el API de Java