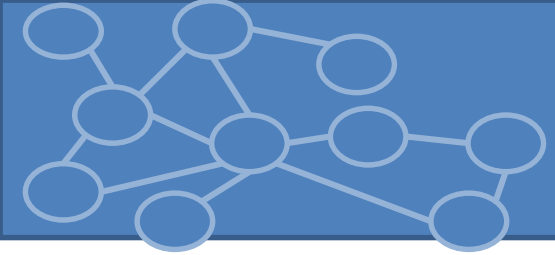
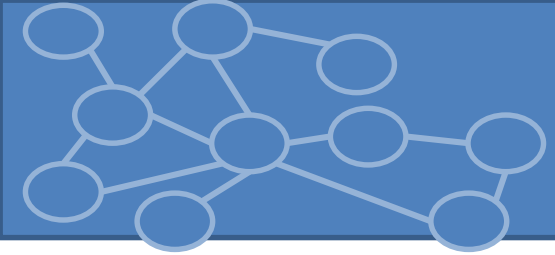


Laboratorio Reti di Calcolatori

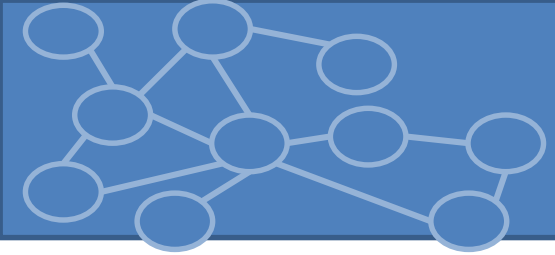
Laurea Triennale in Comunicazione Digitale

Anno Accademico 2013/2014



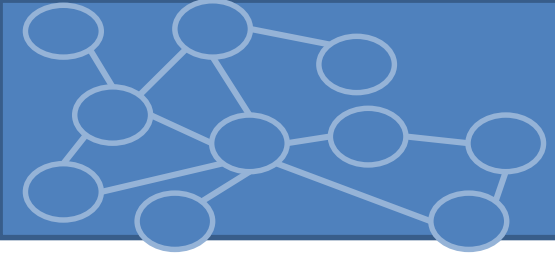
Collection Framework

- **Collection:** oggetto che raggruppa più elementi in una singola unità.
  - per memorizzare, per reperire e manipolare dati aggregati.
  - elementi che formano un gruppo naturale (mazzo di carte, contenitore di lettere, rubrica, libreria, playlist).
- **Struttura dati**: modo per memorizzare e organizzare i dati per facilitare l'accesso e la modifica. Nessuna struttura dati funziona efficientemente con tutte le operazioni.



Collections Framework

- **Collections framework**: strutture dati già pronte associate ad algoritmi e interfacce per operare su strutture dati => programmatore non si preoccupa dell'implementazione.
- Sono standardizzate, facilmente riusabili e condivisibili. Solitamente scritte per risultate computazionalmente (tempo e memoria) efficienti
- Classi e interfacce definite nel package *java.util*



Collection Framework

- Come in altri moderni linguaggi le librerie Java per le Collection **separano le interfacce dall'implementazione**.
- Es: Coda. Interfaccia specifica un metodo add, un remove e un metodo che restituisce il numero di elementi, il tutto rispetto la politica «first in, first out» (FIFO).
- interface Queue<E> {

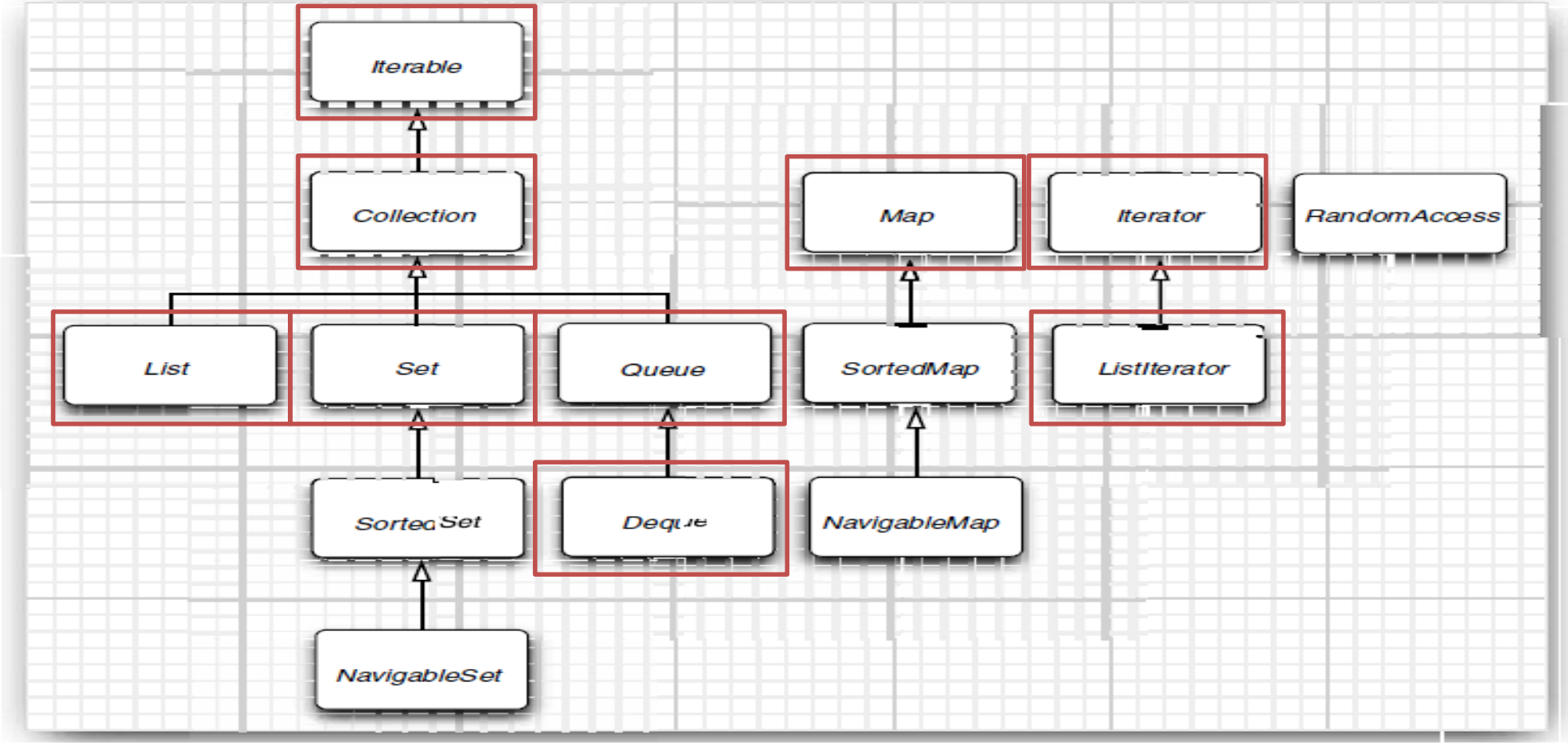
  void add(E element);

  E remove();

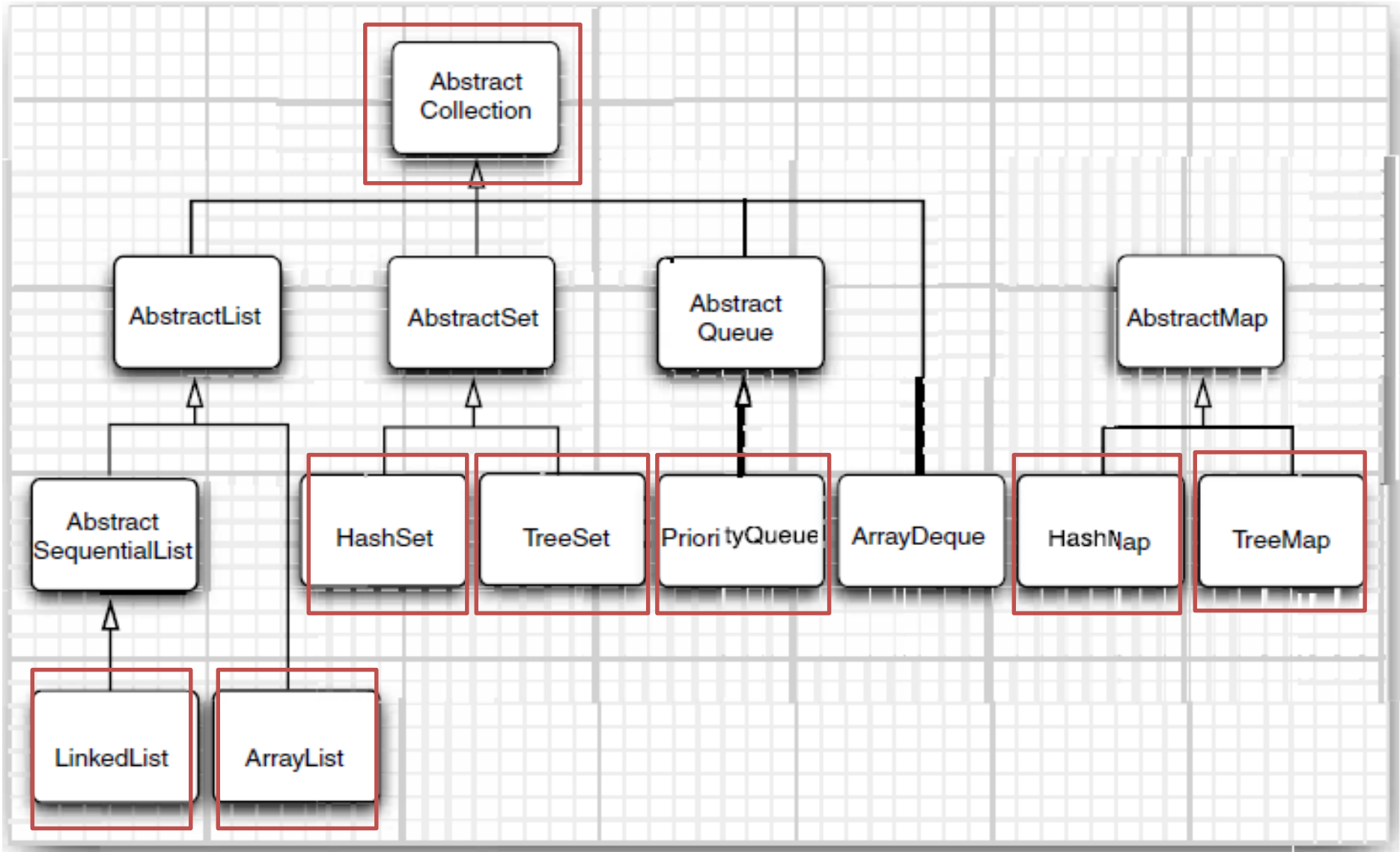
  Int size(); }
- Non so come implementato (circular array o linked list). L'interfaccia usata per mantenere il riferimento all'oggetto. Cambio solo il costruttore se trovo un'implementazione più efficiente.

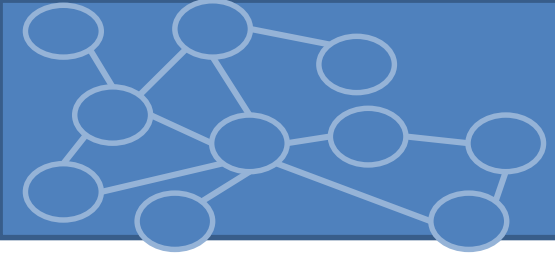
Collections Framework

Framework: insieme di classi come base per funzionalità avanzate



Classi nel Collections Framework





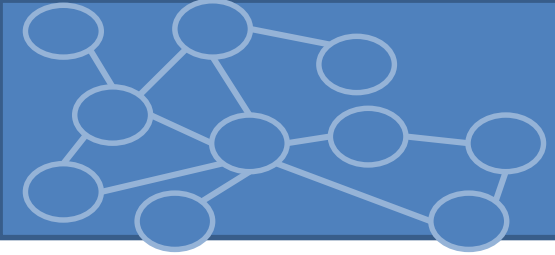
Interfaccia Collection

- Collection: interfaccia fondamentale per collezioni in Java

http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

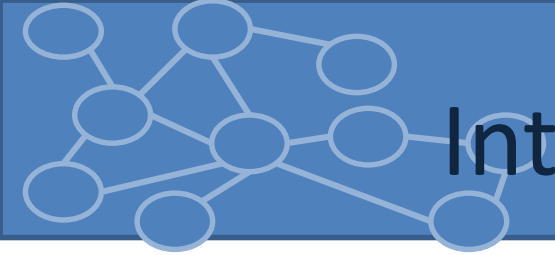
```
public interface Collection<E>{
        boolean add(E element);
        Iterator<E> iterator(); …}
```

- **add**: aggiunge un elemento, true se aggiunta cambia la collezione, false collezione non cambia
- **iterator**: restituisce un oggetto che implementa interfaccia *Iterator*. Uso Iterator per visitare gli elementi di una collezione
- Convenzione sui costruttori: nessun argomento + Collection



Interfaccia Collection

Tipo	Metodo
Iterator<E>	iterator()
Int	size()
boolean	isEmpty()
boolean	contains(Object o)
boolean	containsAll(Collection<?>)
boolean	Add(Object)
boolean	addAll(Collection<? extend E>)
boolean	remove(Object)
Boolean	removeAll(Collection<?>)
void	clear()
boolean	retainAll(Collection<?>)
Object[]	toArray()
<T> T[]	toArray(T[])

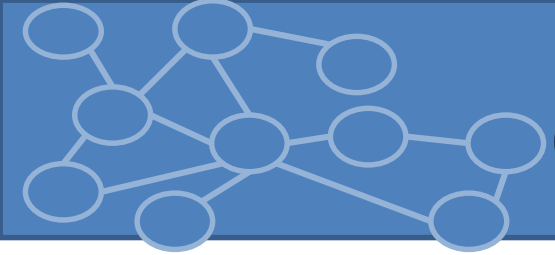


Interfacce Iterator e Iterable

- Interfaccia *Iterator*
  http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

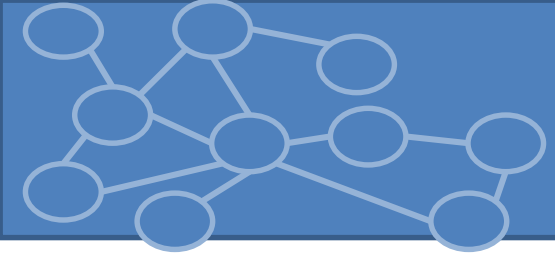
```
public interface Iterator{
        E next();
        boolean hasNext();
        void remove();}
```

- Invocando ripetutamente *next* visito ogni elemento. Se raggiungo la fine e invoco *next*, viene sollevata *NoSuchElementException* => invocare il metodo *hasNext* prima di una *next*.
- Iterare sulla collezione con **while** o **for each** loop (>=5.0)
- For each funziona con oggetti che implementano interfaccia Iterable
- **Collection estende Iterable**
  http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html



Come funziona Iterator

- Ordine di visita dipendente dal tipo di collezione:
  - Es*: ArrayList* è ordinata, *HashMap* è casuale
  - Viene assicurato che entro la fine dell'iterazione vengono visitati tutti gli oggetti.
- Lookup e cambio di posizione fortemente dipendenti: quando invoco *next* il puntatore passa al prossimo elemento e restituisce quello già passato.
- *remove*: rimuove l'elemento che è stato appena restituito dall'ultima *next* => prima di eliminare un elemento controllo se deve essere effettivamente rimosso. Non è permesso invocare *remove* senza prima invocare *next* (IllegalStateException)



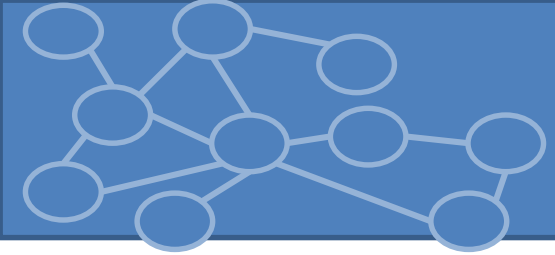
Interfaccia List

- *List*: collezione ordinata -> accesso al dato attraverso un indice *index*

http://docs.oracle.com/javase/7/docs/api/java/util/List.html

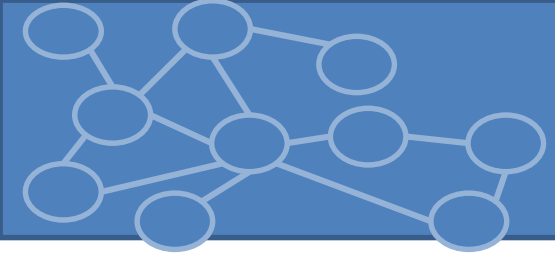
  - Permette elementi duplicati
  - Metodi per accesso indicizzato
    - boolean add(int index, E element)
    - boolean addAll(int index, Collection<? extends E> c)
    - E get(int index)
    - E remove(int index)
    - E set(int index, E element)

Implementazione principali: ArrayList, LinkedList, Vector, Stack.



LinkedList

- array e *ArrayList* hanno un svantaggio: rimuovere un elemento nel centro richiede spostare tutti gli elementi dopo quello eliminato di una posizione: non efficiente.
- La lista concatenata risolve il problema
  - Memorizza un riferimento al dato.
  - Due riferimenti all' elemento precedente e successivo nella lista.
- Rimozione nel mezzo è efficiente => aggiorno solo i riferimenti.
- In Java c'è la classe *LinkedList<E>* http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
- È una sequenza ordinata: l'**ordine degli elementi conta**
  - Metodo add() che dipende da posizione: indice è responsabilità di un iterator.



ListIterator

Interface *ListIterator<E>* **extends** *Iterator<E>*{

void *add*(E elemento);

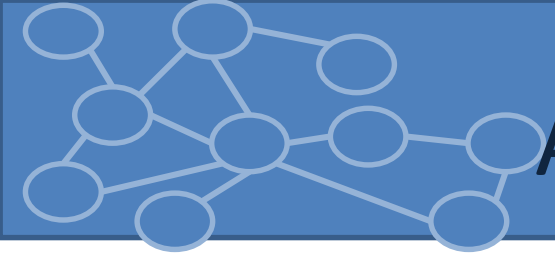
E *previous*();

boolean *hasPrevious*(); }

http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html

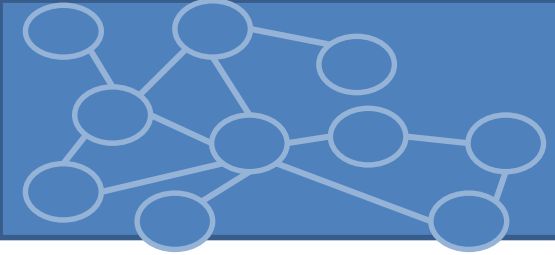
Assume che add modifichi sempre la lista => non restituisce un boolean

- **add** inserisce l'elemento prima della posizione attuale dell'iteratore.
- **set** sostituisce elemento restituito dalla next o previous con un nuovo elemento



Attenzione agli iterators

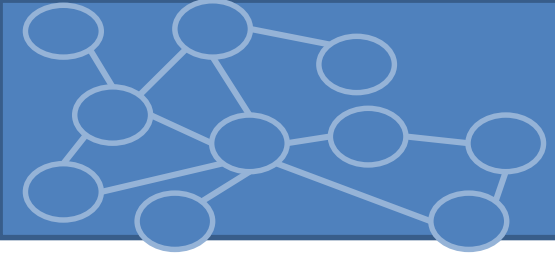
- Se iterator scorre una collection mentre un altro iteratore la modifica si possono creare strane situazioni.
  - Se un iterator rileva che la sua collezione è stata modificata da un altro iterator o da un metodo della collezione solleva una ConcurrentModificationException
- Per evitare: posso attaccare quanti iterator voglio basta che tutti leggano oppure un solo iterator che legge e scrive.



LinkedList e indici

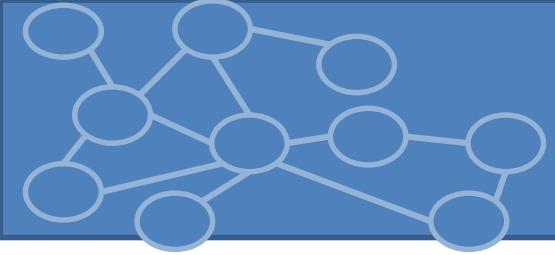
- Lista concatenata non supporta un efficiente metodo per accesso indicizzato. Se voglio accedere a n-esimo elemento della lista devo scorrere n-1 elementi
- A disposizione un metodo inefficiente *get(int index)*
- Cosa ne pensate di questo

```
for (int i = 0; i < list.size(); i++)
        fai qualcosa con list.get(i);
```



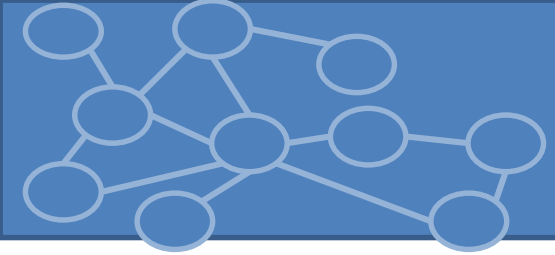
Interfaccia Set

- *Set*: collezione con non contiene elementi duplicati (secondo metodo *equals*) e al massimo un elemento **null.**
- Non aggiunge ulteriori metodi rispetto all'interfaccia *Collection.*
- Implementazioni più usate: HashSet, TreeSet, LinkedHashSet.



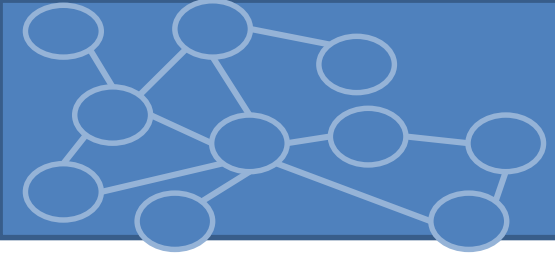
HashSet(1)

- Se non mi interessa l'ordine degli elementi, posso usare strutture dati che mi permettono di trovare gli elementi più velocemente.
- Una di quelle più conosciute è la hash table
  - Per ogni oggetto calcolo (**hash function**) un intero = hash code tale che oggetti con valore dei campi diversi abbiano codici diversi
  - Devo implementare il metodo *hashCode()* che deve essere compatibile con equals. Due oggetti sono uguali se hanno stesso hash code.
- Calcolo dell'hash code dipende solo dallo stato dell'oggetto.



HashSet(2)

- Implementati come array di liste concatenate. Ogni lista è detta **bucket**. Per trovare un oggetto:
  - Calcolo hash code
  - Individuo il bucket modulo(hash code,# buckets)
  - Se bucket già occupato (**hash collision**) controllo se hash code è uguale ad altri oggetti nella lista.
- hash function, numero di bucket
- Rehashing e load factor
- Classe *HashSet* http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html

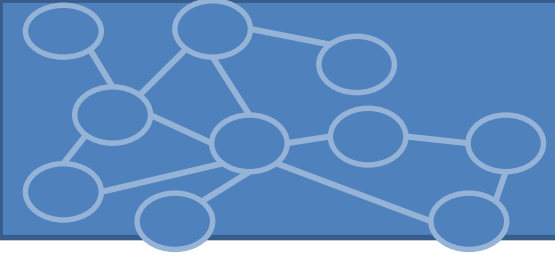


TreeSet

http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html

- Simile ad un hash set ma è una collezione ordinata.
  - Inserimento in qualsiasi ordine
  - Quando itero i valori sono automaticamente presentati in modo ordinato
- Ordinamento dato da struttura ad albero (red-black tree)
- Aggiunta di un elemento più lenta rispetto ad hash set ma oggetti sono ordinati

Costruttori
TreeSet()
TreeSet(Collection<? Extend E>)

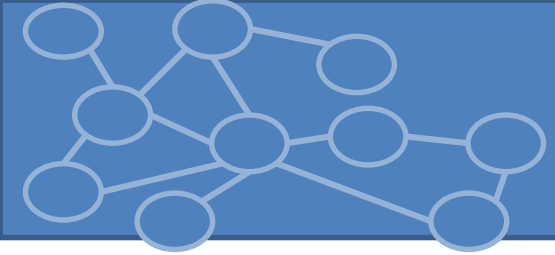


Confronto tra oggetti

- *TreeSet* assume che gli elementi inseriti implementino l'interfaccia *Comparable*

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

- *compareTo*: restituisce 0 se a e b sono uguali, un valore negativo se a < b, un valore positivo se a > b.
  - Molte classi standard implementano l'interfaccia (es. String usa ordine lessicografico)
- E se volessi ordinare i miei elementi secondo un diverso criterio ?
- Indico un altro criterio di confronto passando un oggetto *Comparator* che implementa la relativa interfaccia data dal metodo
  - **int compare(T a, T b)**



Queue e Deque

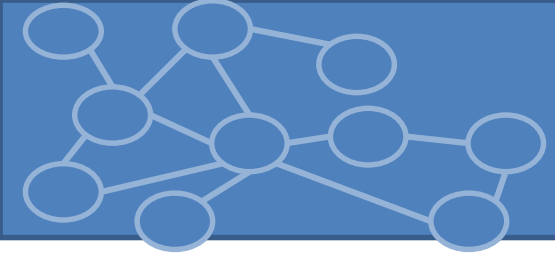
Queue<E>
Boolean add(E)
Boolean offer(E)
E remove()
E poll()
E element()
E peek()

Deque<E>
Void addFirst(E)
Void addLast(E)
boolean offerFirst()
Boolean offerLast()
E removeFirst()
E removeLast()
E pollFirst()
E pollLast()
E getFirst()
E getLast()
E peekFirst()
E peekLast()

PriorityQueue

- http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html
- Restituisce gli elementi in modo ordinato dopo che sono stati inseriti in modo arbitrario.
  - Se invoco *remove()* mi viene restituito l'elemento minore
  - La *PriorityQueue* non ordina i suoi elementi ma utilizza un heap => *add* e *remove* pongono l'elemento minore nella radice dell'albero
  - Può contenere elementi che implementano *Comparable* o ricevere un oggetto *Comparator* nel costruttore

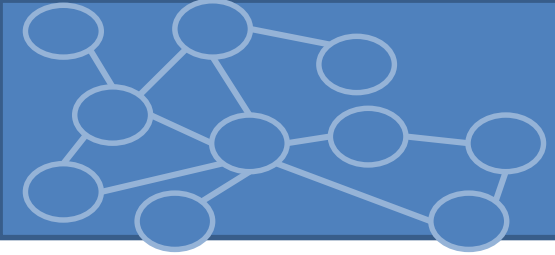
Costruttori
PriorityQueue()
PriorityQueue(int capacitàIniziale)
PriorityQueue(int capacitàIniziale, Comparator<? Super E>)



Interfaccia Map

- Per trovare un elemento in un insieme devo avere una copia dell'oggetto che voglio trovare => Limiti
- La Map supera questo limite memorizzando una coppia chiave/valore. Posso trovare un valore se fornisco la chiave
- Classi *TreeMap* e *HashMap* che implementano interfaccia *Map*
  - albero e hash function applicata solo alla chiave
  - **Chiave deve essere unica**.
- Map **non** considerata come una **collezione** ma posso ottenere 3 viste diverse

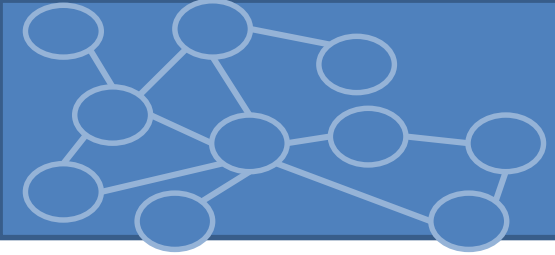
Metodi
Set<K> keySet()
Collection<K> values()
Set<Map.Entry<K,V>> entrySet()



Interfaccia Map

	Metodi e costruttori
V	getKey(K chiave)
V	put(K chiave, V valore)
void	putAll(Map<? extend k,? extend V> elementi)
boolean	containsKey(Object chiave)
boolean	containsValue(Object valore)
Set<K>	keySet()
Collection<V>	values()
Set<Map.Entry<K,V>>	entrySet()

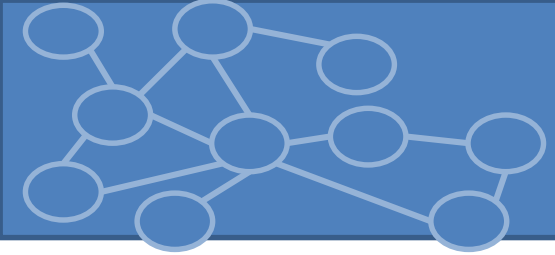
http://docs.oracle.com/javase/7/docs/api/java/util/Map.html



Vista sincronizzata

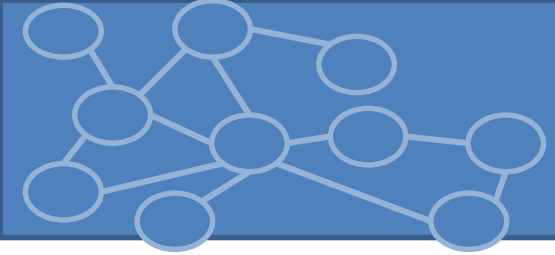
- Se si accede ad una collezione con più thread, devo preservare la collezione da danni derivati dall'uso concorrente (es: uno add e altro rehashing)
- Meccanismo della vista per rendere la collezione thread safe
- Serie di metodi per creare viste sincronizzate fornite dalla classe Collections

http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html



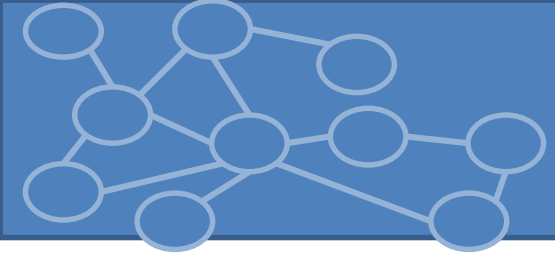
Riferimenti

- Riferimenti bibliografici
  - Horstmann, Cornell, ’’Core Java, Volume I – Fundamentals’’, Sun Microsystem Press (Capitolo 13)
  - Arnold Ken, Gosling James, Holmes David,’’Il linguaggio Java. Manuale ufficiale’’ Pearson (Capitolo 23)
  - Collection Tutorial : http://docs.oracle.com/javase/tutorial/collections/index.html



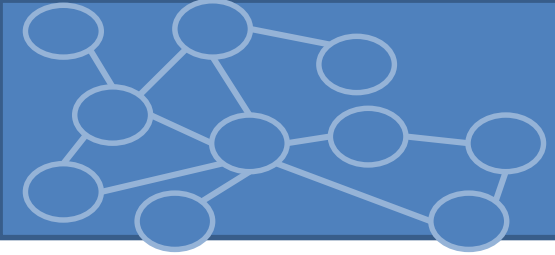
Esercizi

1. Implementare le classi *LinkedListaCoda<E>* e *CircularArrayQueue<E>* che implementano l'interfaccia *Coda<E>*. Sul sito del laboratorio trovate il codice dell'interfaccia *Coda<E>*
2. Si crei un metodo *static void rimuoviNomi*(List<String> lista, String lettera) che elimina da 'lista' tutti i nomi che iniziano con la lettera specificata. Nella soluzione si utilizzi un iteratore. (La classe in cui definire il metodo non ha importanza)
3. Nella classe Libro (scorsa lezione) si definisca un metodo *void raddrizzAutori()*. Per ogni autore di un libro, se il secondo nome non è definito, si crea un nuovo oggetto autore con campo secondo definito dalla stringa «NonDefinito». Si verifichi il corretto funzionamento del metodo.



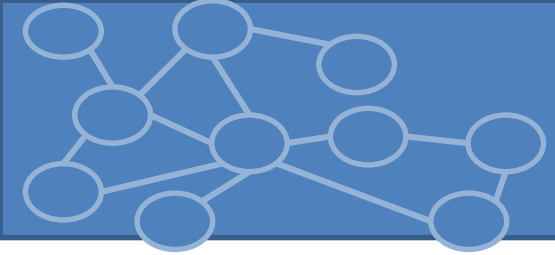
Esercizi

4. Si crei un metodo main (nella classe Alice) che utilizzando l'interfaccia Set conti il numero di parole distinte contenute nel file alice.txt (file scaricabile dal sito del laboratorio).
5. Si completi il codice del main nella classe Esercizo5. Da file vengono letti nome, cognome e identificativo (intero) da cui viene creato un oggetto Impiegato (da implementare). Si creino un HashSet e un TreeSet per l'inserimento di impiegati. L'hashcode dipende da nome e cognome, mentre compareTo di Impiegato dipende dall'id dell'impiegato.
6. Si confrontino le prestazioni dell'inserimento in un HashSet e un TreeSet.



Esercizi

7. Usando un numero opportuno di TreeSet ordinare gli impiegati in base al nome, al cognome e all'id.
8. Si crei una PriorityQueue di GregorianCalendar, si aggiungano varie date e si stampi in ordine crescente le date aggiunte.
9. Si crei una classe PlayList di oggetti Canzone dotati di un campo int che indica quanto quella canzone mi piace. Un campo della classe PlayList deve essere una PriorityQueue<Canzone>. Si devono implementare i metodi
   - void *addCanzone*(Canzone c)
   - void *play*()
     - Stampa titolo e cantante delle canzoni in base alla preferenza. Tra una canzone e l'altra si deve aspettare un secondo (*Thread.sleep(1000)*)



Esercizi

10. Utilizzando una mappa si determinino le frequenze delle parole nel testo alice.txt