

Gatto G.W. civica 61 -> io, veronica, yara, giulia
argomento ->

VERIFICA DI RIPETIZIONE -> Alleviamo un programma a livello macchina
che rappresenta un livello hardware.
Consistono più sezioni che hanno l'interazione
per essere collegate alla cella

size, constructor, isEmpty, push, pop, top

```
public Pila ArrayList() {  
    pila = new ArrayList();  
}  
  
public int size() {  
    return pila.size();  
}  
  
public boolean isEmpty() {  
    return pila.isEmpty();  
}  
  
public void push(T elem) {  
    pila.add(elem);  
}  
  
public T pop() {  
    pila.remove  
    return pila.remove(pila.size() - 1);  
}  
  
public T top() {  
    return pila.get(pila.size() - 1);  
}
```

```
public class Arraylist<Integer> {
```

```
    private Integer[] a;
    private int dl;
```

```
    public Arraylist() {
```

```
        a = new Integer[1]; // dl = a.length ->  
        dl = 0; // dim logica è uguale all'indice della prima cella  
    } // libera
```

```
    public boolean isEmpty() {
```

```
        return dl == 0;
```

```
    public int size() {
```

```
        return dl;
```

```
    public void add(Integer e) {
```

```
        if (dl < a.length) a[dl++] = e;  
        else {
```

```
            Integer[] a_new = new Integer[a.length * 2];
```

```
            System.arraycopy(a, 0, a_new, 0, dl);
```

```
            a_new[dl] = e;
```

```
            a = a_new;
```

```
}
```

```
} // possiamo scriverlo meglio?
```

```
if (dl == a.length) {  
    a = a_new;  
    // copia a in a_new  
    a = a_new;  
}  
a[dl] = e;
```

```
    public Integer get(int index) {
```

```
        if (index < 0 || index >= dl) throw new IndexOutOfBoundsException();  
        return a[index];
```

```
}
```

```
    public Integer remove(int index) { // ipotico di eliminare l'ultimo elemento
```

```
        if (index < 0 || index >= dl) throw new IndexOutOfBoundsException();
```

```
        if (index == dl - 1) {
```

```
            Integer z = a[index];
```

```
            dl--;
```

```
            return z;
```

```
} // codice per eliminare nel mezzo spostando la posizione di destra a  
// sinistra di 1.
```

```
}
```

$$T_{\text{add}} > ? \quad T_{\text{remove}} = m \quad T_{\text{get}} = 1 \quad T_{\text{size}} = m$$

Complessità computazionale

<code>isEmpty</code> : $O(1)$	<code>add(in fondo)</code> : in media $O(n)$	<code>remove(in fondo)</code> : $O(1)$
<code>size</code> : $O(1)$	<code>get</code> : $O(1)$	<code>set</code> : $O(1)$
<code>add(generico)</code> : $O(n)$	<code>remove(generico)</code> : $O(n)$	

Se volessi usare le stringhe invece che gli interi?

\Rightarrow Al posto di Integer mettiamo String

\Rightarrow è un'azione da informatici: NO

\Rightarrow Programmazione corretta: ArrayList è generico perché usa un array `E[]`;

Tutte le strutture dati sono dei contenitori dove si può verificare il numero di elementi presenti e se è vuota.

\Rightarrow definiamo un'interfaccia Contenitore!

`public interface Contenitore {`

`boolean isEmpty();`

`int size();`

`}`

Pila: LIFO (Last In First Out)



\Rightarrow Se volessi accedere all'elemento 3? accesso sequenziale
distruttivo

Come implementiamo una pila?

- uso ArrayList nei miei programmi e lo uso come una pila, ma le classi ArrayList fornisce funzionalità diverse della pila!
- creiamo una classe `Pila` che implementa la classe ArrayList ricordando all'utente le funzionalità diverse.
(L'utente potrà solo aggiungere e rimuovere in cima, non nel mezzo)

In anche, volendo, un array può magari un'altra particolare struttura dati.

Quindi, possiamo individuare la caratteristica "essere una Pila", che rappresentiamo con un'interfaccia

`public interface Pila<T> extends Contenitore {`

`void push(T elem); // add in cima`

`T pop(); // remove in cima`

`T top(); // insersione in cima`

STRUTTURE DATI

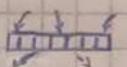
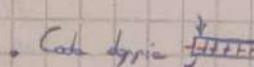
Una struttura dati è una struttura che serve a memorizzare e organizzare i dati a utilizzare in un programma.

Una struttura dati possiede sempre:

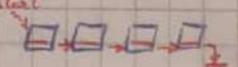
- un costruttore al coda
- operazioni di inserimento
- operazioni di rimozione per conoscere il numero di elementi (size)
- operazioni di istruzione
- operazioni di utilità → per sapere se la struttura è vuota (isEmpty)

Le strutture dati si differenziano per modalità di funzionamento interno
→ Bisogna utilizzare la corretta struttura dati in base al problema da risolvere (anche in base all'efficienza delle operazioni necessarie).

CATEGORIA DI ESEMPI DI STRUTTURE DATI:

• ArrayList  . Pilha  . Coda  . Coda doppia 

• Coda ordinata  . Coda prioritaria: è una coda dove si attribuisce una priorità agli elementi inseriti, si elimina l'elemento che ha una priorità più alta.

• Liste concatenate  oggett. puntato al successivo

• Insieme  oggetti non duplicati e non ordinati.

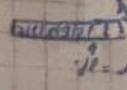
• Mappe: associazioni chiave - valore . Albero

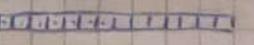


• Tabelle di Hash: gg → codice di hash (probabilmente univoco)

ArrayList:

ff -> lista


ff = dim lista

quando ff = df →  ridimensionamento in "grande"
(notare anche il ridimensionamento in "piccolo")

La rimozione in una coda è $O(n)$ mentre in una pila è $O(1)$.
È possibile portare il metodo remove dalla coda da $O(n)$ a $O(1)$.
Si, "liberando" l'estremo libro.

```
public class CodaArrayListEstremoFissoLibero {
```

```
    private ArrayList<T> coda;  
    private int il;
```

```
    public CodaArrayListEstremoFissoLibero() {  
        coda = new ArrayList<T>();  
        il = 0;  
    }
```

```
    public int size() {  
        return coda.size() - il;  
    }
```

```
    public boolean isEmpty() {  
        return size() == 0;  
    }
```

```
    public T deQueue() {  
        if (isEmpty()) throw new NSEE();  
        return coda.get(il++); // T ris = coda.get(il); }   
        // coda.set(il++, null); }   
        // return ris; }   
        memoiz
```

```
    public T getFront() {  
        if (isEmpty()) throw new NSEE();  
        return coda.get(il);  
    }
```

```
    public void enqueue(T e) {  
        coda.add(e);  
    }
```

oss:

$x = \boxed{\text{null}}$

$x = \boxed{\text{null}}$

ricorre gg non è
necessario da varuna variabile
oggetto alla gc
collezione si occuperà di
liberare posto di memoria

Dov chiamiamo una classe, che implementa l'interfaccia Pila, incapsulando un ArrayList (generico)

public class PilaArrayList <T> implements Pila {

private ArrayList <T> pila;

public PilaArrayList () {
pila = new ArrayList <T>();

public int size () {
return pila.size(); } $O(1)$

public boolean isEmpty () {
return pila.isEmpty(); } $O(1)$

public void push (T elem) {
pila.add (elem); } $O(1)$ in media

public T top () {
if (isEmpty ()) throw new NoSuchElementException ();
return pila.get (pila.size () - 1); } $O(1)$

public T pop () {
if (isEmpty ()) throw new NoSuchElementException ();
return pila.remove (size () - 1); } $O(1)$

Come usare questa classe generica?

PilaArrayList <String> p = new PilaArrayList <String>();

OSS! Attenzione a non invocare metodi su oggetti di tipo generico
T dove: - elem, size(), NO perché non so se size() ci sarà!
= new T() NO!

Coda: ~~queue~~ + FIFO (First In First Out)

load tail
ultimo ultimo
fin libero

add = enqueue
remove = dequeue

get = getFront

public interface Coda < T > extends Contenitore { }

void enqueue (T e);
T dequeue ();
T getFront ();

}

public class CodaArrayList < T > implements Coda { }

private ArrayList < T > coda;
public CodaArrayList () {
coda = new ArrayList < T > ();
}

public int size () {
return coda.size (); O(1)
}

public boolean isEmpty () {
return coda.isEmpty (); O(1)
}

public void enqueue (T e) {
coda.add (e); O(n) - in media
}

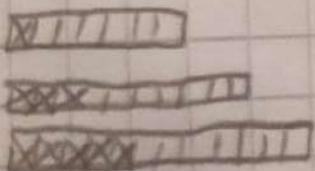
public T getFront () {
if (isEmpty ()) throw new NSEE (); O(1)
return coda.get (0);

public T dequeue () {
if (isEmpty ()) throw new NSEE (); O(n)
return coda.remove (0);

}

055: X ~~100~~

Double Coda circolare:



Gli oggetti vengono liberati, ma le celle di memoria sono sempre riservate all'arraylist quindi diventerà sempre più grande con molte celle inutilizzabili!

↔  coda circolare con array \rightarrow - punto di voler avere una dimensione fissa

* maxima prima

Coda circolare:

~~public class CodaCircolareArray<T> implements Coda {~~

~~private Integer[] coda;~~

~~private int head;~~

~~private int tail;~~

~~private int n;~~

~~public CodaCircolareArray<T> (int capacity) {~~

~~n = 0;~~

~~head = 0;~~

~~tail = -1;~~

~~coda = new Integer[capacity];~~

~~}~~

~~public boolean isEmpty() {~~

~~return n == 0;~~

~~}~~

~~public int size() {~~

~~return n;~~

~~}~~

~~public void enqueue (Integer e) {~~

~~if (coda.length == n) throw new E ("No space");~~

~~int index = (tail + 1) % coda.length;~~

~~coda[index] = e;~~

~~tail = index;~~

~~n++;~~

~~}~~

~~public Integer dequeue () {~~

~~if (n == 0) throw new NSEE();~~

~~Integer r = new Integer(coda[head]); coda[head] = null;~~

~~head = (head + 1) % coda.length;~~

~~n--;~~

~~return r;~~

~~}~~

~~public Integer getFront () {~~

~~if (n == 0) throw new NSEE();~~

~~return coda[head];~~

~~}~~

oss: Tutti i metodi sono $O(1)$ e coda di avere una
grandezza fissa

Coda doppia: (doppio) ArrayBegue in libreria

~~class~~ ~~Empty~~: O(n) ~~size~~: O(n)
getFirst: O(n) getlast: O(n)
removeFirst: O(n) removeLast: O(n)

addFirst(T a): O(n) in media addLast(T a): O(n) in media

o: verifica se una sequenza di numeri è palindroma
ArrayBegue < Integers
ArrayBegue < strings

public boolean isPalindrome (ArrayBegue<Integers> a) {
Boolean r = true; // devo clonarlo, poi agire sul clone
while (a.size() > 1) {
if (a.removeFirst() . equals(a.removeLast())) r = false;
}
return r;

public boolean isPalindrome (ArrayBegue<String> a) {
ArrayBegue<String> copy = (ArrayBegue<String>) a.clone();

public boolean isPalindrome (ArrayBegue<Integers> a) O(n)
if (a == null) throw new NPE();
if (a.isEmpty()) throw new NSEE();
ArrayBegue<Integers> copy = (ArrayBegue<Integers>) a.copy();
boolean r = true;
while (copy.size() > 1) {
if (copy.removeFirst() != copy.removeLast()) r = false;
}
return r; // return true;

public class PilaArrayBegueT-in
private ArrayBegue<pile>
public PilaArrayBegue ()
pile = new ArrayBegue<pile>()

public int size() {
return pile.size();

public Boolean isEmpty() {
return pile.isEmpty();

public T top() {
return pile.getlast();

public void push(T e) {
pile.addLast(e);

public T pop() {
return pile.removeLast();

```

if (n.equals(next)) {
    azioni.enqueue(new Azione(q, p));
} else {
    azioni.dequeue(new Azione(q, p));
}

```

Coda prioritaria:

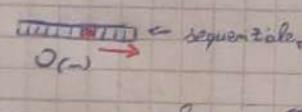
FIFO ma c'è una priorità: argomento Priority Queue

(argomento $a[::]$)

Priority Queue $\langle T \rangle$ in libreria

$\rightarrow T$ deve implementare Comparable $\langle T \rangle$

- size: $O(1)$	- add: ?	- peek: $O(n)$
- isEmpty: $O(1)$	- remove: ?	dipende da come viene implementata la coda prioritaria

- ArrayList:
 sequenziale
 $O(n) \rightarrow$
 corso il punto dove inserire l'oggetto
 $\Rightarrow O(mg(n)+m) = O(m)$

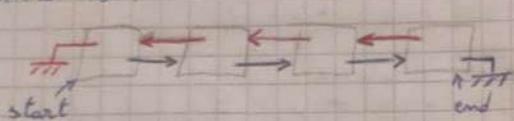
- oggetti: vengono mantenuti in senso crescente d'priorità
- add: $O(n)$
- remove: $O(1)$

- oggetti: ordinati in senso decrescente
- add: $O(n)$
- remove: $O(n)$

- LinkedList:
 Il miglior modo per implementare una coda prioritaria è usando un Heap

Migliore
 - Heap (ArrayList):
 $\Delta h = \log(n)$
 add $O(\log(n))$ min-heap $O(n)$ $\Delta h = \log(n)$
 remove $O(1)$ funzioni = i remove $O(\log(n))$

5
Linked list:



~~not~~
~~remove~~

lista dinamicamente concatenata

In libreria: è generica

perché per individuare la posizone
dove inserire l'oggetto → dobbiamo

.size: O(n) . addFirst: O(1) . addLast: O(n) recorrere la

.isEmpty: O(1) . addFirst: O(1) lista in modo sequenziale

.getFirst: O(1) . getLast: O(1) ⇒ non vi è necessario direttamente

ma solo sequenziale

.get: O(n) . removeFirst: O(1) . removeLast: O(1)

.remove: O(n)

Per iterare sulla lista Y si usa un ITERATORE

metto con es: 

hasNext → next

hasNext → next

add(w) => A B w | C D E

hasPrevious => previous => A B | w C D E

Oss: il funzionamento
è un valore

Sceglieremo gli oggetti

while (it.hasNext())

 T class

 // elabro

 3

È equivalente a:

La classe ListIterator

- add -

4

- remove -

5

- Ultra -

il metodo

next =

`while (impost, risult) > 0 {`

~~int
int~~

list di elementi creato
per individuare la posiz.
de l'oggetto stessa
Un verso: One recorre a
ste in modo sequenziale
non si è nessun doppio
ma solo sequenziale

nextList = 0;

next

C D E

`List<List<T> > L; L.addList(i, T);`

Però la lista concatenata è `List<List<T>> L = L.addAll(i, j)`

Ora: il funzionamento è sempre sequenziale (anche se specifiche
una valica "in modo" per i)

Succiamo gli oggetti della lista (da i:0)

`while (L.hasNext()) {`

`T elem = L.next();`

l'elenco l'elemento... → non è un accorgere di list, ma
è un'istruzione

È equivalente a: `for (T elem : L) {`

 // elabora dati con elem...
}

La classe ListIterator ha anche i seguenti metodi:

- add → inserisce un nuovo elemento dopo l'attuale e
sposta quest'ultimo dopo l'elemento aggiunto
it.add(oggetto)

- remove → rimuove l'elemento appena "spostato" da l'iteratore
. next → A/B/C → A/B/C → A/C

- previous → A/B/C → /A/B/C → /B/C
it.remove()

- Oltra a next e hasNext esiste previous e hasPrevious e
il metodo set per aggiornare il nodo appena reperito da
next e previous

a) I think that this role might be really ~~interesting~~
with

b) I hope to grow within the company

c) I left my last job because it had become too stressful
and

Funzionamento di una coda prioritaria che usa una lista
circolare e rappresentata da numeri. Per semplicità, molti sono numeri e la priorità massima è
associata a valori bassi.

4 6 6 3 7 7 9 manteniamo la lista ordinata

4 6 6 3 7 7 9

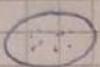
✓ inserire nella posizione corretta quando la lista
4 6 6 2 7 7 8 9 ordinata

inserire un duplicato 7

se i nodi di stessa priorità utilizzano un approccio FIFO

4 6 6 7 7 7 9

Immeni:



Collezione di elementi non duplicati, dove non
è definito l'ordine di ordinamento

Inserimento, rimozione e ricerca sono efficienti:
interfaccia Set<E>

↳ HashSet<E>

↳ classi concrete
TreeSet<E>

↳

Immeni implementati
tramite tabella di:
Hash

Immeni implementati.
tramite alberi.

HashSet: HashSet<E>

↳ E deve implementare il metodo hashCode

hashCode: oggetto → codice univoco associato all'oggetto
(con alta probabilità)

Quando 2 o più oggetti ~~sono~~ diversi hanno stesso hashCode si dice che
esiste una collisione

hashcode
00
01
02
03
04

P(collisione) deve essere molto
bassa

Object: equals, hashCode
utilizzano gli indirizzi di memoria per ottenere l'output

Nelle nostre classi dobbiamo entrambe le seguenti situazioni:

- ridefiniamo equals ma non hashCode → 2 ogg. con diversi indirizzi, ma con uguale contenuto;
equals = true
hashCode = codici diversi
con già

- ridefiniamo hashCode ma non equals → 2 ogg. con diversi indirizzi, ma con uguale contenuto
equals = falso
hashCode = codici uguali

⇒ ridefinire sempre entrambi i metodi, per renderli concordi.

15 → 11111

non che ordine inseriscono?

- ↳ TreeSet: ordine insieme del compagno
- ↳ HashSet: "casuale" (nel senso che è fatto dall'ordine di inserimento)

OSS! I metodi add e remove sono delle classi HashSet e TreeSet mentre nella lista concatenata sono invocati sull'iteratore perché? Nei set non ha senso aggiungere o rimuovere per indice

OSS! I set non hanno posizioni

OSS! interfaccia Set<E> viene implementata da HashSet<E> e TreeSet<E>

Set<E> set = new HashSet<E>();
 (TreeSet)

E 14.11

- Coppia (String descrizione, int priorità)

- Comparatore Coppia (che implementa Comparable)

PriorityQueue<Coppia> pg = new PriorityQueue<Coppia>(comparatore);

boolean end = false;

while (!end) {

Scanner s = new Scanner(System.in);

sop("1 per aggiungere, 2 rimuovere, 0 uscire"); // saliamo

int u = s.nextInt(); // controlli

if (u == 1) {

sop("inserire descrizione");

String d = s.next();

sop("inserire priorità");

int p = s.nextInt();

pg.add(new Coppia(d, p));

}

else if (u == 2) { // saliamo controlli e try/catch

sop("l'ultima più prioritaria da fare è " + pg.remove());

}

else end = true;

}

hashCode, Object: a partire dagli indirizzi di memoria degli oggetti
E' a partire dalla variabile `this` l'istanza degli oggetti

es: (hashCode in String)

```
public int hashCode() {
    final int HASH_MULTIPLIER = 31;
    int h = 0;
    for (int i = 0; i < this.length(); i++) {
        h = h * HASH_MULTIPLIER + this.charAt(i);
    }
    return h;
}
```

(hashCode in Country => String-name, double-area)

```
public int hashCode() {
    int h1 = name.hashCode(); // in String
    int h2 = double.hashCode(); // in Double
    final int HASH_MULTIPLIER = 31;
    int h = h1 * HASH_MULTIPLIER + h2;
    return h;
}
```

↓
public int hashCode() {
 return Objects.hash(name, area);
}

Metod. di HashSet: isEmpty() O(1), size() O(1) add remove contains

(dipende dalla bontà dell'algoritmo
che calcola gli hash)

TreeSet è > 
tieni ordinati gli oggetti del set
(l'ordinamento non è quello d'inserimento)
→ E deve implementare Comparable

add remove contains

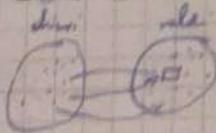
O(log n)

OSS!: quando sceglievi HashSet e TreeSet → se interessava ad Tree
se interessava eff. Hash

OSS!: scegliere un set

```
Iterator<String> it = nomeSet.iterator();
while (it.hasNext()) {
    String s = it.next...;
```

Mappe: mappiamo coppie / associazioni chiave - valore



Map < K, V =

HashMap<V> TreeMap<K, V>
(che non ha chiavi) (che ha chiavi)

Esempio: Map<String, Integer> m = new HashMap<>();

OSS! Le mappe vengono molto utilizzate per tenere traccia dei contagi associati alle istanze (continua la memoria delle chiavi).

- metodi → • put: m.put(chiave, valore)
↳ se la chiave è già presente il
valore associato viene sovrascritto
nella mappa
- get: m.get(chiave) → restituisce il valore associato, null
se la chiave non è presente
- remove: m.remove(chiave) → funziona come get ma
l'elemento chiave-valore
viene rimosso

Scorrirete? Come per i Set!
Bisogna usare una mappa

```
Set<String> chiavi = m.keySet();
for (String k : chiavi) {
    Integer v = m.get(k);
    System.out.println("Chiave: " + k + " - Value: " + v);
}
```

Esempio: Mettete l'oggetto in un file

ipotesi: file.txt è un file che contiene parole separate
solo da caratteri di spaziatura

Stampare per ogni parola nel file il numero di volte in

```

public static void mRicorrente(String f) {
    try {
        Scanner in = new Scanner(new File(f));
        Map<String, Integer> m = new HashMap<String>();
        while (in.hasNext()) {
            String key = in.next();
            if (m.containsKey(key)) m.put(key, 1);
            Set<String> chiavi = m.keySet();
            for (String k : chiavi) {
                if (k.equals(key)) {
                    m.put(key, m.get(k) + 1);
                } else {
                    m.put(key, 1);
                }
            }
            Set<String> chiavi = m.keySet();
            for (String k : chiavi) {
                Integer v = m.get(k);
                System.out.println("Parola: " + k + " / Numero apparizioni: " + v);
            }
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}

```

CORRETTO

```

Scanner in = new Scanner(new File("file.txt"));
Map<String, Integer> m = new HashMap<String>();
while (in.hasNext()) {
    String k = in.next();
    Integer count = m.get(k);
    if (count == null) m.put(k, 1); ] count = 0
    else m.put(k, count + 1);
    → m.put(k, count + 1);
}

Soltanto se zero la mappa e stampo con il codice prec

```