Riflessione:
OOP e principi di design

Obiettivi OOP

- **Robustezza**
  - Programmi capaci di gestire situazioni **inaspettate** (es. Input inattesi)
  - Requisito importante in applicazioni "life-critical"
- **Adattabilità**
  - Programmi capaci di **evolvere** (es. funzionare su architetture diverse, oppure avere nuove funzionalità)
  - Concetto correlato alla portabilità
- **Riusabilità**
  - Codice utilizzabile come **componente** di diversi sistemi in varie applicazioni
  - Chiarezza su cosa fa e non fa il nostro codice

OOP – principi di design

- **Astrazione**
  - Distillare i concetti che meglio rappresentano un oggetto o un sistema
- **Information hiding**
  - Nascondere l'informazione a utenti esterni, lasciando vedere solo **l'interfaccia**
    - Parti del programma possono cambiare senza effetti sulle altre
- **Modularità**
  - Organizzare un sistema software in **componenti funzionali** separate

OOP - caratteristiche

- I tre principali strumenti concettuali messi a disposizione da un linguaggio OOP
  - Classi, oggetti, incapsulamento
    - L'informazione viene nascosta dentro "scatole nere" (le classi), e l'accesso ad essa è controllato
  - Ereditarietà
    - Una classe può essere estesa da sottoclassi, che ne ereditano le funzionalità e le specializzano
  - Polimorfismo
    - Il tipo di una variabile non determina completamente il tipo dell'oggetto a cui essa si riferisce

Strutture dati

Strutture Dati

- Contenitori che organizzano i dati in un formato specifico
- Progettate per rispondere a diverse esigenze in termini di
  - tempo di costruzione della struttura e accesso ai dati
  - occupazione di spazio in memoria
- Obiettivo: comprendere le caratteristiche per scegliere l'ottimo in base alle nostre esigenze

Panoramica

- Strutture dati
  - Lineari: array, liste concatenate
  - Non lineari: alberi, grafi

- Strutture dati astratte (o Abstract Data Type, ADT)
  - Pila, Coda
  - Mappa, Dizionario
  - Tabella, HashTable
  - Insieme

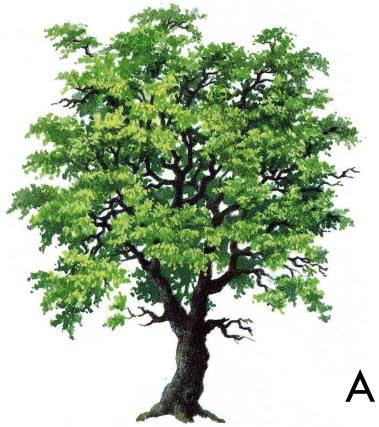
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Non queste pero'!

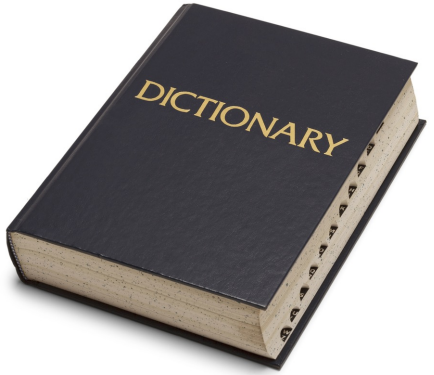
PILA

MAPPA

CODA



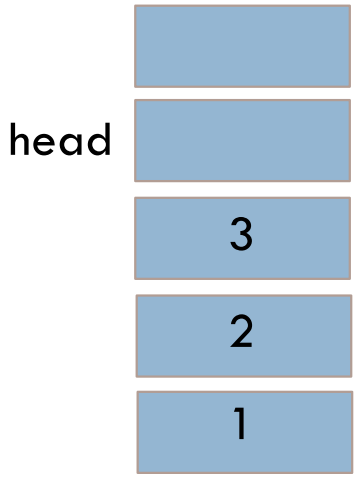
ALBERO



DIZIONARIO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Esempi di strutture dati e ADT



PILA

front

back

CODA

chiave unica

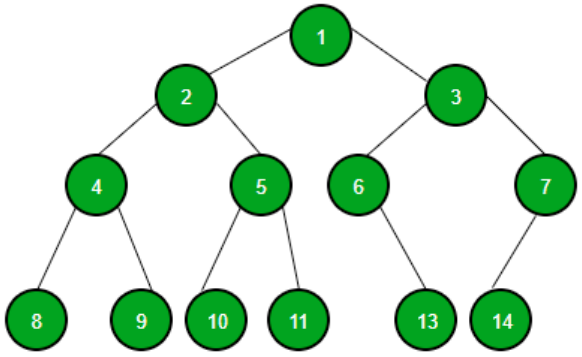
123456	Dato associato
127893	Dato associato
126543	Dato associato
128774	Dato associato

MAPPA

chiave non unica

mele	2
arance	12
mango	7
banane	1
mele	3
mango	10

DIZIONARIO

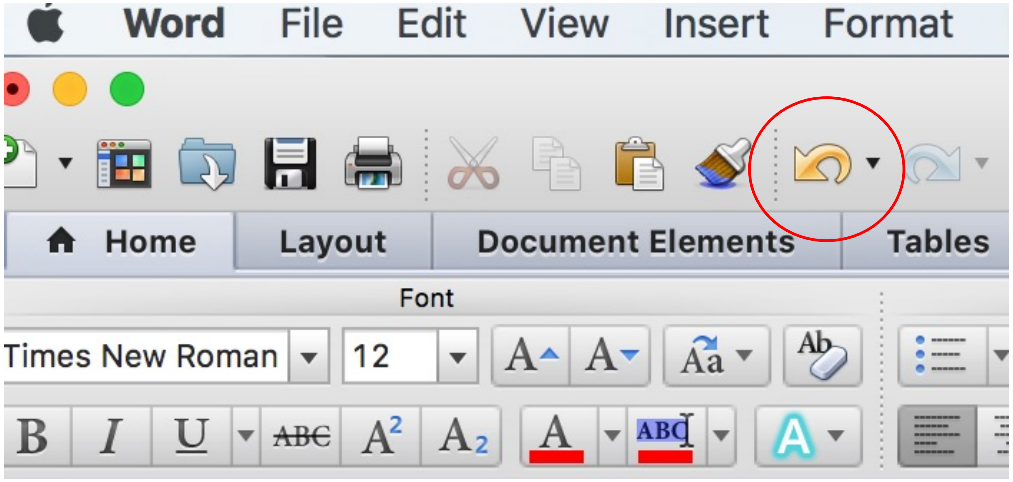


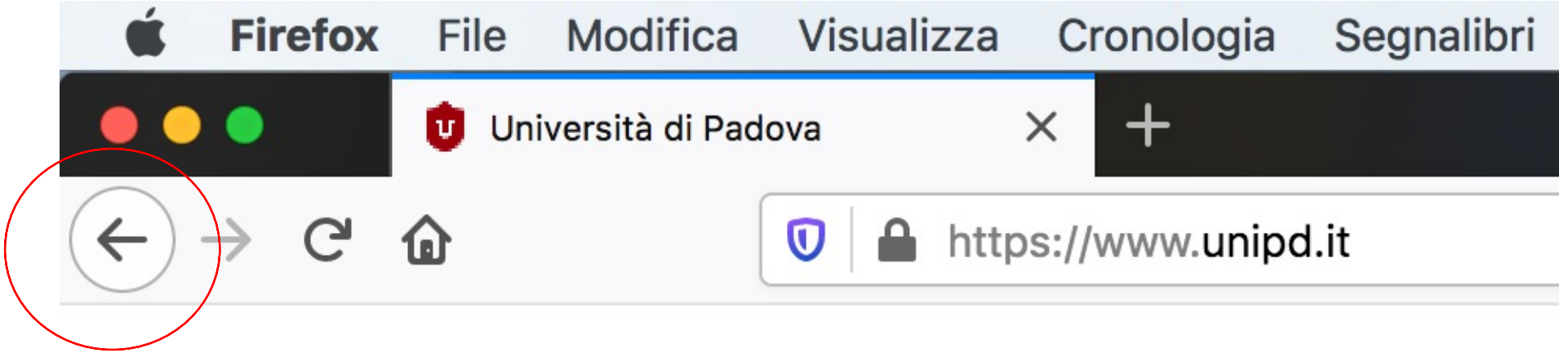
ALBERO

CODICE

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Le strutture dati "nascoste"





Uniweb | Area stampa | Dipartimenti | Scuole

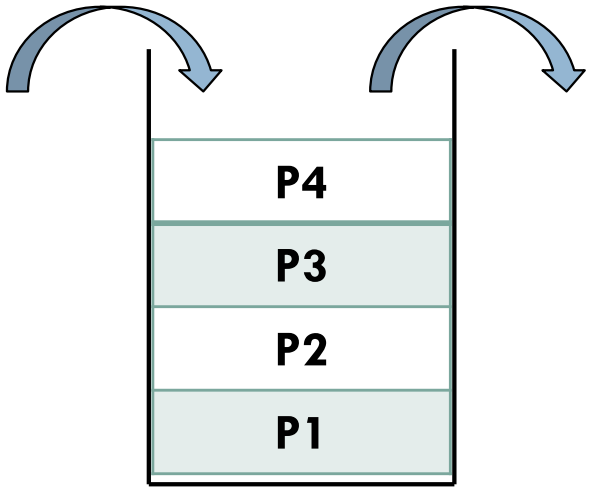
Le ultime informazioni inserite…

- Gli editor tengono traccia delle ultime (N) azioni compiute dall'utente
  - Con UNDO annulliamo l'ultima azione
  - E poi la penultima e poi la terz'ultima

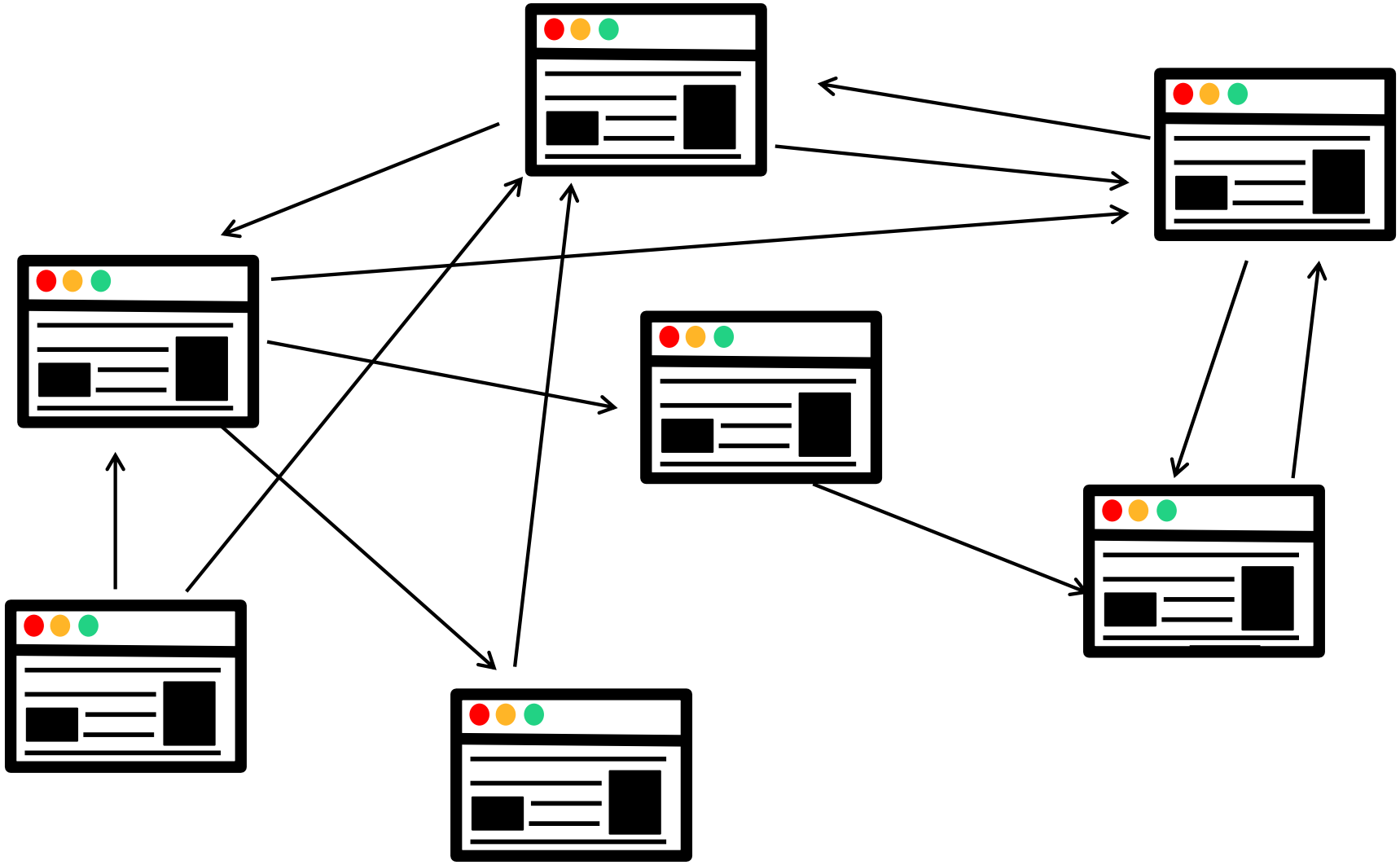
- I browser tengono traccia delle ultime (N) pagine visitate sul web
  - Con BACK tornate all'ultima pagina visitata prima di quella attuale
  - E poi la penultima e poi la terz'ultima

... saranno le prime ad essere estratte

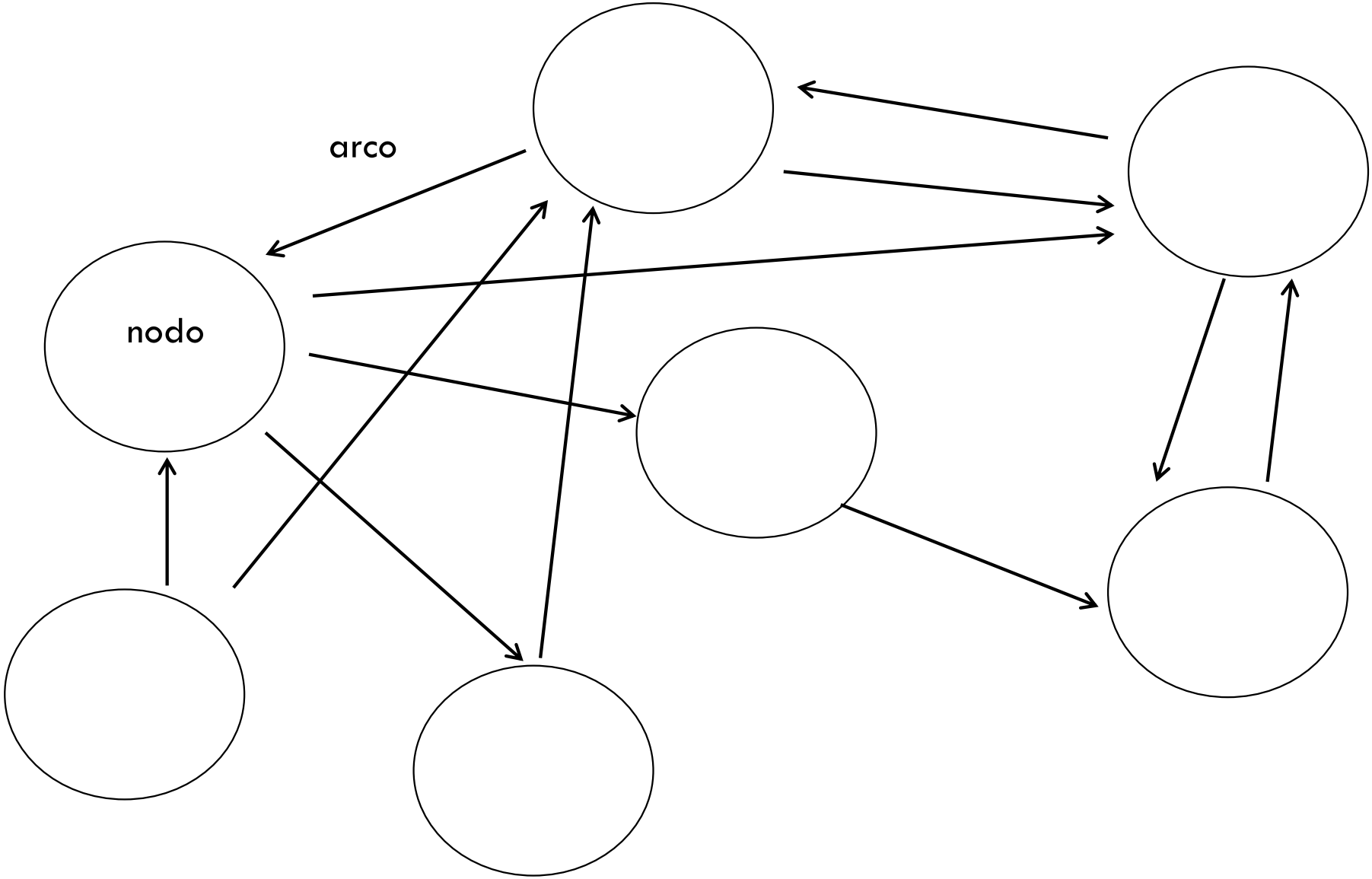
- La struttura dati astratta che le memorizza si chiama Pila o Stack
- In analogia con una pila di piatti:
  - I piatti si aggiungono uno alla volta, uno sopra l'altro
  - Se devo prendere un piatto prendo quello in cima alla pila, ovvero l'ultimo inserito
  - Per prendere un piatto in mezzo devo prima togliere quelli che ci sono sopra a partire dall'ultimo inserito
  - In una pila:
    - Inserisco, in ordine, p1, p2, p3, p4
    - Estraggo, in ordine, p4, p3, p2, p1



Il web...



… e' un grafo

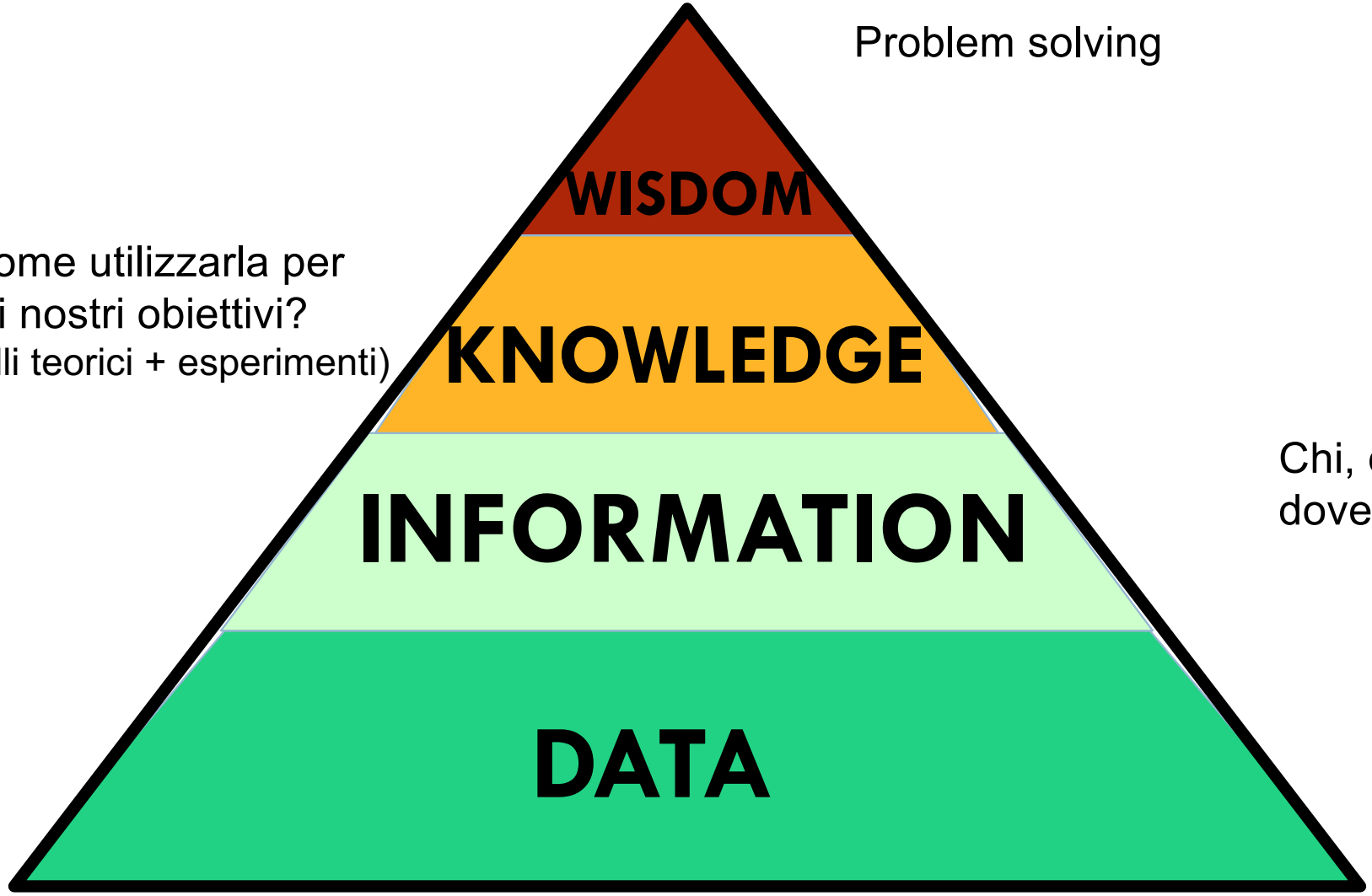


Altri esempi

- Grafi
  - Social networks
  - Biological networks
  - Citation networks
  - ...
- Alberi (di vario tipo)
  - Autocompletamento di parole
  - Compilatori
  - Classificazione gerarchica

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Perché elaborare i dati è importante? Il modello DIKW



Take home message

- I dati sono fondamentali
  - Senza dati non si arriva alla conoscenza
- L'elaborazione dei dati, anche se digitalizzati, costa
  - Tempo di elaborazione
  - Risorse di memoria
- Le strutture dati sono fondamentali
  - Ruolo cruciale nello sviluppo di soluzioni efficienti (assieme agli algoritmi)
  - Conoscere le caratteristiche delle strutture dati ci aiuta a fare scelte ottime per la soluzione dei problemi di analisi dei dati che dobbiamo risolvere

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

ADT e strutture dati in Java

Strutture dati

- Una ***struttura dati*** (***data structure***) è un modo sistematico per **organizzare i dati in un contenitore e** per **controllarne le modalità d'accesso**
  - In Java, si definisce una struttura dati con una classe

- L'informatica usa molte diverse strutture dati, specializzate e ottimizzate per la soluzione di problemi diversi
  - **Tutto si potrebbe fare con i soli array, ma vedremo che l'uso di strutture ad accesso controllato agevola il compito del programmatore e la manutenzione del codice**

Tipi di dati astratti

- Un tipo di dati astratto (ADT, Abstract Data Type) è una rappresentazione astratta di una struttura dati, un modello che specifica:
  - il tipo di dati memorizzati
  - le operazioni che si possono eseguire sui dati insieme al tipo di informazioni necessarie per eseguire le operazioni

Tipi di dati astratti

- In Java si definisce un tipo di dati astratto con una **interfaccia**
- Come sappiamo, un'interfaccia descrive un **comportamento** che sarà assunto da una classe che realizza l'interfaccia
  - è proprio quello che serve per definire un ADT
- Un ADT definisce **cosa** si può fare con una struttura dati che realizza l'interfaccia
  - la classe che rappresenta concretamente la struttura dati definisce invece **come** vengono eseguite le operazioni

Il pacchetto java.util

- Il pacchetto **java.util** della libreria standard contiene molte definizioni di ADT come interfacce e loro realizzazioni come classi
- La nomenclatura e le convenzioni usate in questo pacchetto sono, però, piuttosto diverse da quelle tradizionalmente utilizzate nella teoria dell'informazione (purtroppo e stranamente...)
- Quindi, proporremo un'esposizione teorica di ADT usando la terminologia tradizionale, senza usare il pacchetto **java.util** della libreria standard

Tipi di dati astratti

- Un ADT mette in generale a disposizione ***un costruttore che crea il contenitore vuoto*** e metodi per svolgere le seguenti azioni (a volte solo alcune)
  - ***inserimento*** di elementi nel contenitore
  - ***rimozione*** di elementi dal contenitore
  - ***ispezione*** degli elementi presenti nel contenitore
    - ***ricerca*** della presenza di un elemento nel contenitore
- Le diverse strutture dati differiscono per le modalità di funzionamento di queste tre azioni
  - Ad esempio, un array identifica gli elementi mediante un indice

Strutture dati

- In mancanza di vincoli specifici, una struttura dati deve poter ospitare tutti i dati che vi vengono inseriti, eventualmente ridimensionandosi in modo trasparente per l'utente
  - **Con l'ovvio vincolo della memoria disponibile:** se questa si esaurisce, solitamente la struttura dati non gestisce esplicitamente tale condizione, ma si affida alla gestione normale da parte della JVM, che prevede il lancio dell'eccezione **OutOfMemoryError** durante la creazione di un array

Un contenitore generico

```
public interface Container
{   boolean isEmpty();
    void makeEmpty();
}
```

- Anche le interfacce, come le classi, possono essere "estese"

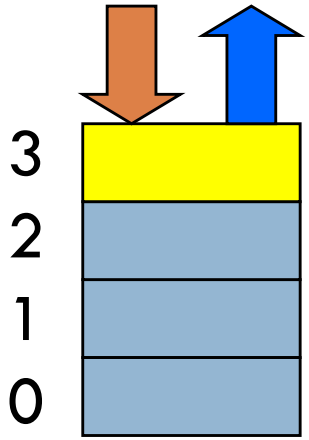
```
public interface Stack extends Container
{   ... // push, pop, top
}
```

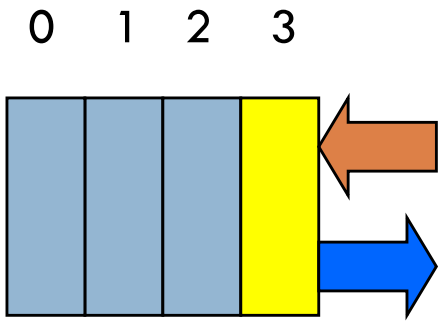
- Un'interfaccia eredita tutti i metodi della sua super-interfaccia
- Per realizzare un'interfaccia estesa occorre definire anche i metodi della sua super-interfaccia

Pila (*stack*)

Pila (*stack*)

- I dati sono inseriti e rimossi secondo la modalità **Last In – First Out: LIFO**
  - **l'ultimo dato che è stato inserito sarà il primo a essere rimosso**
- L'unico dato ispezionabile è quello che si trova "in cima" alla pila





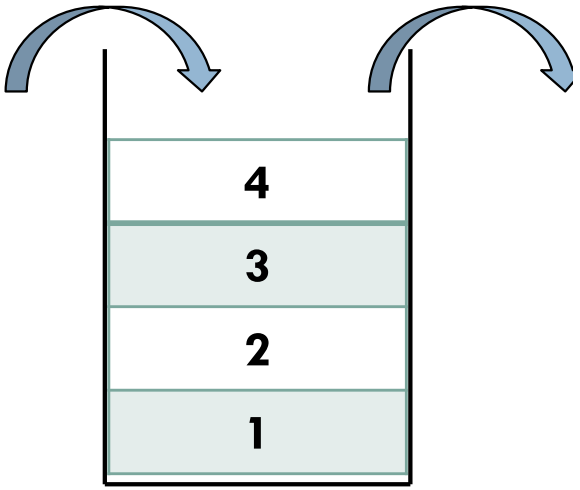
Utilizzo di pile (stack)

- I **browser** per internet memorizzano gli indirizzi dei siti visitati recentemente in una struttura di tipo pila.
  - Quando l'utente visita un sito, l'indirizzo è inserito (**push**) nella pila. Il browser permette all'utente di saltare indietro (**pop**) al sito precedente tramite il pulsante "indietro"
- Gli **editor di testo** forniscono generalmente un meccanismo di "undo" che cancella operazioni di modifica recente e ripristina precedenti stati del testo.
  - Questa funzione di "undo" è realizzata memorizzando le modifiche in una struttura di tipo pila.
- La **JVM** usa una pila per memorizzare l'elenco dei metodi in attesa durante l'esecuzione in un dato istante

Operazioni sullo stack

- Push
  - Inserimento di un elemento in cima alla pila
- Pop
  - Rimozione dell'elemento in cima alla pila (e sua restituzione)
- Top
  - Accesso all'elemento in cima alla pila senza rimozione

- Inserisco: push(1), push(2), push(3), push(4)
- Estraggo: pop(), pop(), pop(), pop()
  - In uscita avro', in ordine, 4, 3, 2, 1
- Alla fine ho la pila vuota



Pila (*stack*): metodi "classici"

```
public interface Stack extends Container
{   void push(Object obj);
    Object pop();
    Object top();
}
```

- Nota:
  - I metodi **pop** e **top** non possono essere invocati con una pila vuota
  - Lanciano, ad esempio, **java.util.EmptyStackException** oppure, più genericamente, **java.lang.IllegalStateException**

- Ogni ADT di tipo "Container" ha inoltre i metodi
  - **isEmpty** per sapere se il contenitore è vuoto
  - **makeEmpty** per svuotare il contenitore

Nota

Molto importante:

- Definiremo tutti gli **ADT** in modo che possano genericamente contenere oggetti di tipo **Object**
- Ciò consente di inserire nel contenitore oggetti di qualsiasi tipo (un riferimento di tipo **Object** può essere relativo a qualsiasi oggetto)

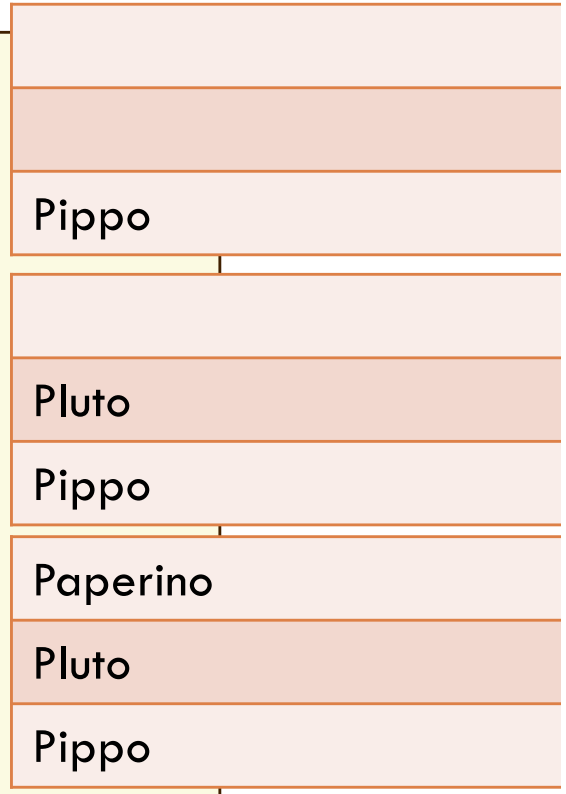
Utilizzo della pila

- Per evidenziare la potenza della definizione di tipi di dati astratti come interfacce, supponiamo che qualcun altro abbia progettato la seguente classe

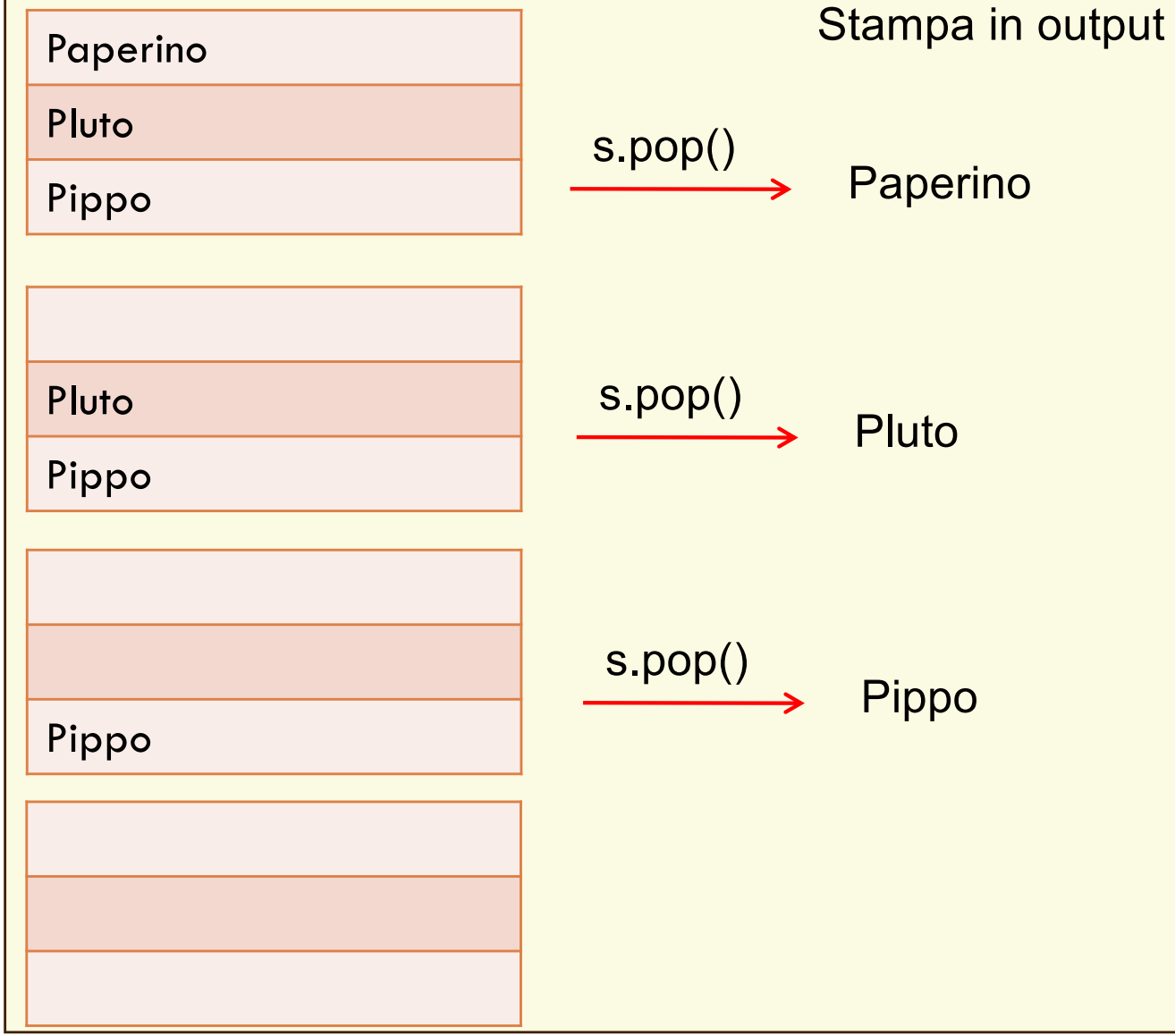
```
public class StackX implements Stack
{   ...
}
```

- Senza sapere come sia realizzata StackX, possiamo usarne un esemplare mediante il suo comportamento astratto definito in Stack
- Allo stesso modo, possiamo usare un esemplare di un'altra classe StackY che realizza Stack

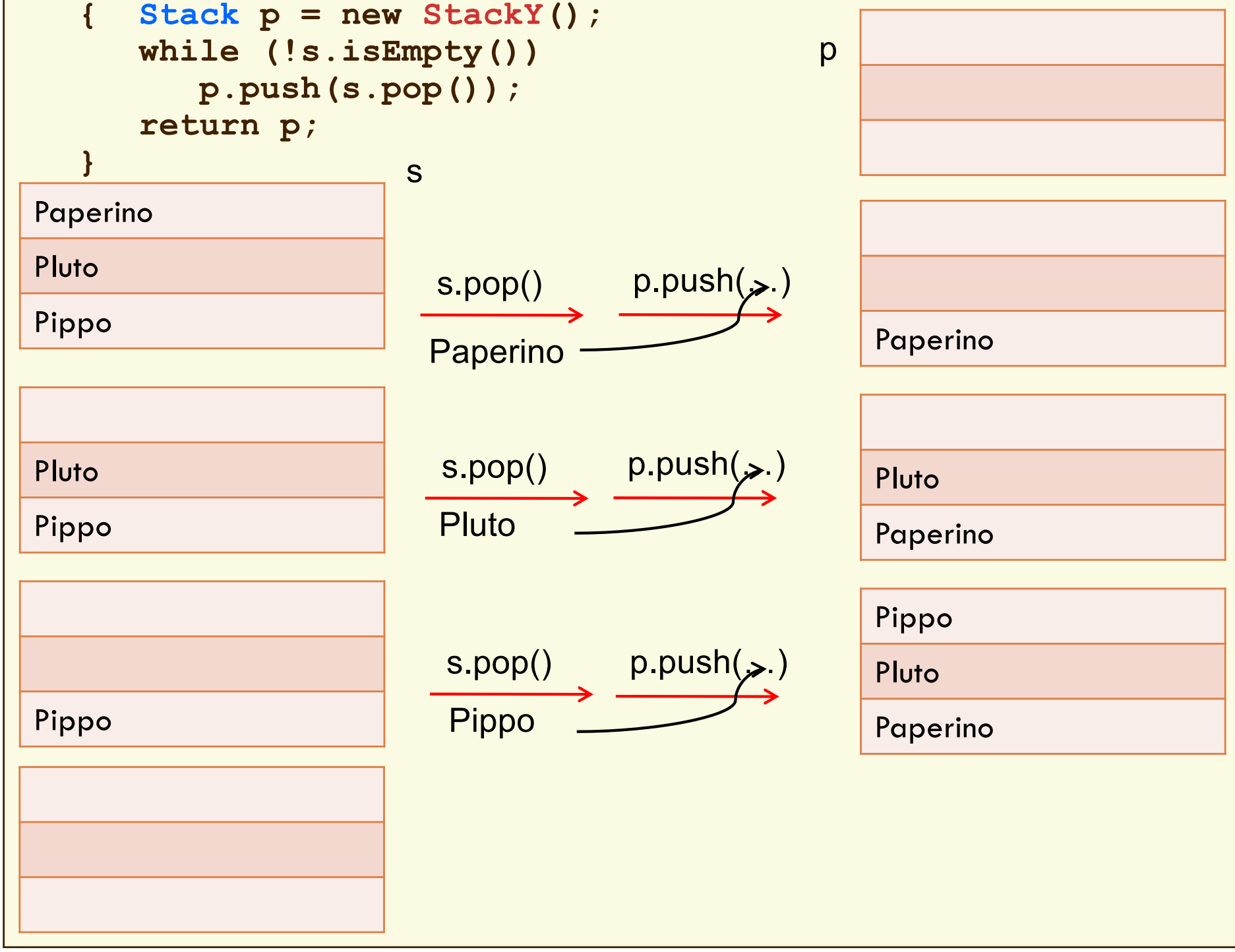
```
public class ReverseStack // UN ESEMPIO
{  public static void main(String[] args)
   {  Stack s = new StackX();
      s.push("Pippo");
      s.push("Pluto");
      s.push("Paperino");
      printAndClear(s);
      System.out.println();
      s.push("Pippo");
      s.push("Pluto");
      s.push("Paperino");
      printAndClear(reverseAndClear(s));
   }

   private static Stack reverseAndClear(Stack s)
   {  Stack p = new StackY();
      while (!s.isEmpty())
         p.push(s.pop());
      return p;
   }
   private static void printAndClear(Stack s)
   {  while (!s.isEmpty())
         System.out.println(s.pop());
   }
}
```



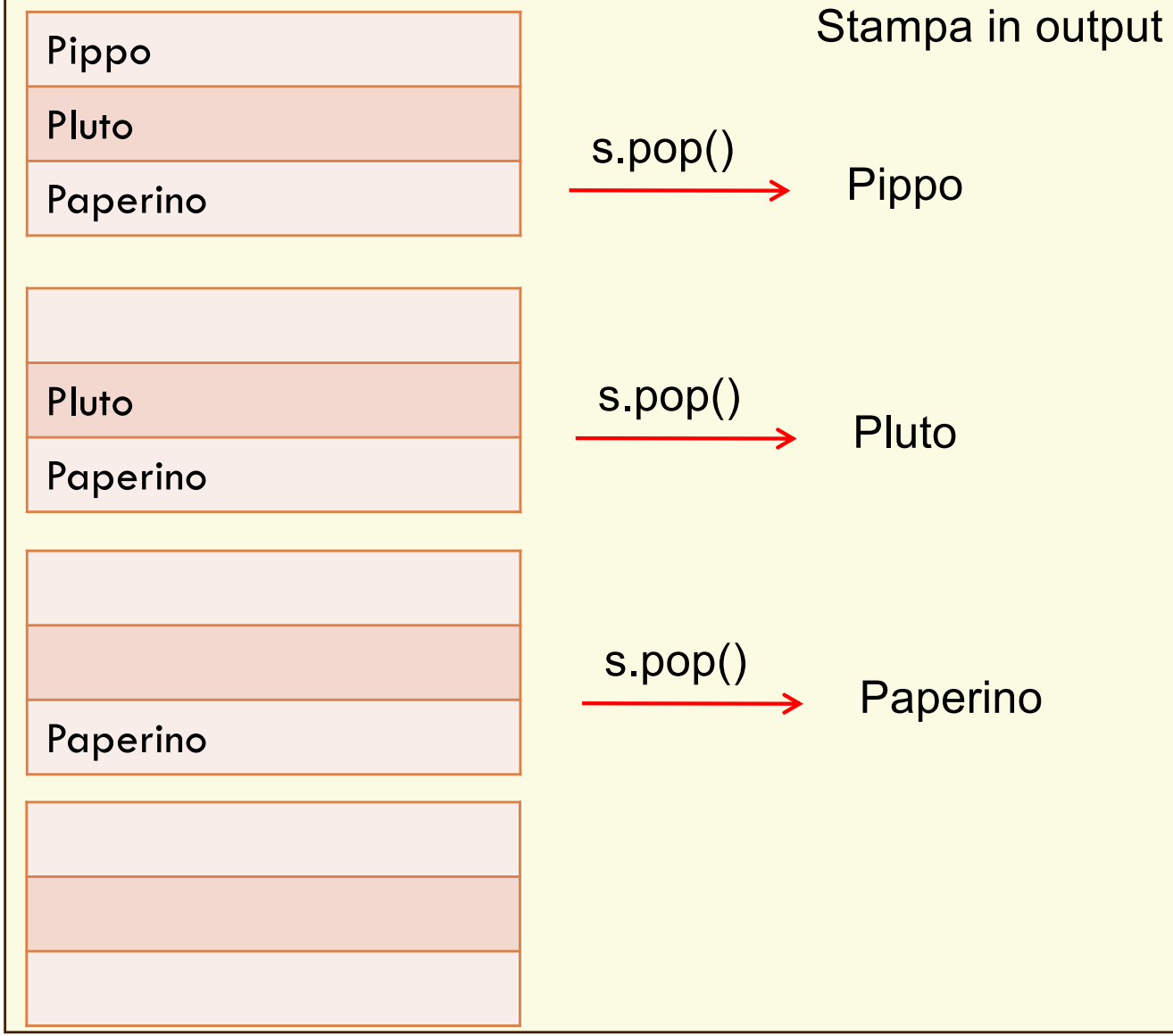
```
private static void printAndClear(Stack s)
   {  while (!s.isEmpty())
         System.out.println(s.pop());
   }
```



```
private static Stack reverseAndClear(Stack s)
   {  Stack p = new StackY();
      while (!s.isEmpty())
         p.push(s.pop());
      return p;
   }
```



```
private static void printAndClear(Stack s)
   {  while (!s.isEmpty())
         System.out.println(s.pop());
   }
```



Esercizio

- Data una stringa composta di lettere e asterischi, si consideri ogni lettera un push e ogni asterisco la **stampa** dell'elemento restituito da un pop. Scrivere l'output:

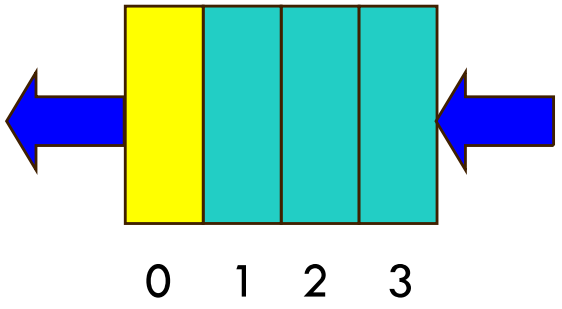
- Esempio: EAS*Y*QUE***ST***IO*N***
- Pila:
- Output:

Coda (*queue*)

Coda (queue)

- In una ***coda*** (***queue***) gli oggetti possono essere inseriti ed estratti secondo un comportamento definito **FIFO** (***First In, First Out***)
  - il primo oggetto inserito è il primo ad essere estratto
  - il nome è stato scelto in analogia con persone in ***coda***

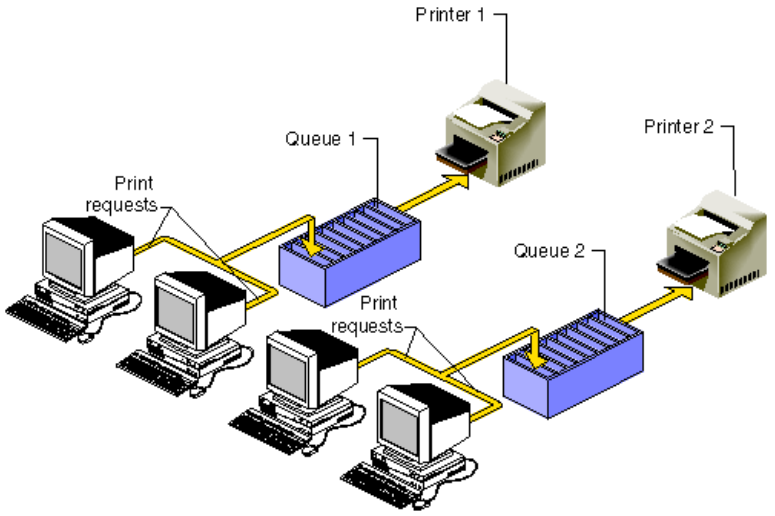
- L'unico oggetto che può essere ispezionato è quello che verrebbe estratto



Coda (queue)

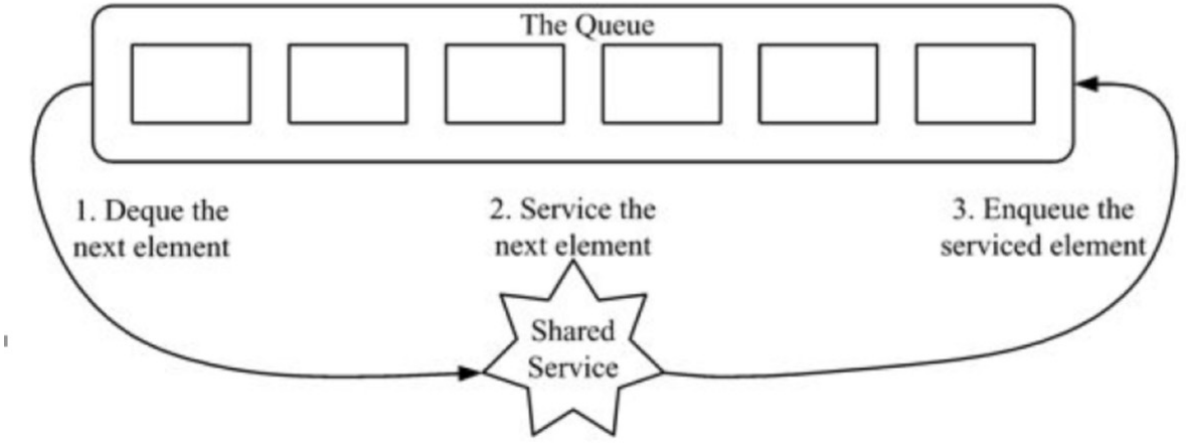
- Esistono molti possibili utilizzi di una struttura dati con questo comportamento
  - Simulazione del funzionamento di uno sportello bancario
    - Clienti inseriti in coda per rispettare priorità di servizio

  - File da stampare vengono inseriti in una **coda di stampa.**
    - La stampante estrae dalla coda e stampa un file alla volta



Coda circolare

- Spesso si utilizza una coda secondo una modalità **circolare**: gli elementi vengono **estratti** dalla prima posizione, "**serviti**", e **reinseriti** in ultima posizione
  - Esempio: lo **scheduler** di un sistema operativo assegna le risorse della **CPU** a molti processi attivi in parallelo



Coda (queue)

- I metodi che caratterizzano una coda sono
  - **enqueue** per inserire un oggetto nella coda
  - **dequeue** per esaminare ed eliminare dalla coda l'oggetto che vi si trova da più tempo
  - **getFront** per esaminare l'oggetto che verrebbe eliminato da **dequeue**, senza estrarlo
- Infine, ogni ADT di tipo "contenitore" ha i metodi
  - **isEmpty** per sapere se il contenitore è vuoto
  - **makeEmpty** per vuotare il contenitore

Coda (queue)

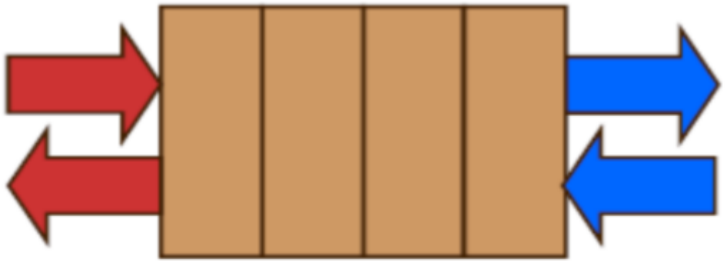
```
public interface Queue extends Container
{   void enqueue(Object obj);
    Object dequeue();
    Object getFront();
}
```

- Si notino le similitudini con la pila
  - **enqueue** corrisponde a **push**
  - **dequeue** corrisponde a **pop**
  - **getFront** corrisponde a **top**

ADT Coda doppia (Deque)

Coda doppia

- In una **coda doppia** gli oggetti possono essere **inseriti** ed **estratti** ai due estremi di una disposizione lineare, cioè all'**inizio** e alla **fine** della coda doppia
  - Consente l'ispezione degli oggetti che verrebbero estratti, alle due estremità



Coda doppia

- Double-ended queue, ovvero coda con due estremità terminali
  - Si usa la parola **deque** (le prime due lettere sono le iniziali di double-ended) che si pronuncia **deck**, per non confondersi con il metodo deque

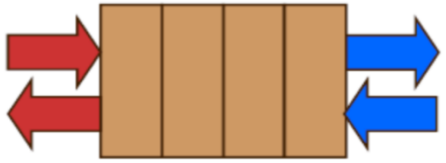
Coda doppia (deque)

```
public interface Deque extends Container
{
   void addFirst(Object obj);
   void addLast(Object obj);

   Object removeFirst() throws EmptyDequeException;
   Object removeLast() throws EmptyDequeException;

   Object getFirst() throws EmptyDequeException;
   Object getLast() throws EmptyDequeException;
}
```

Coda doppia (deque)

```
public interface Deque extends Container
{   . . . }
```

- Si noti che, pur essendo la coda doppia una "coda con ulteriori funzionalità" tradizionalmente **non** viene definita come estensione di queue
- Quindi, dal punto di vista dell'ereditarietà tra interfacce, una coda doppia NON è una coda particolare
- D'altra parte una coda doppia può essere vista anche come pila doppia!



Strutture dati associative

Associazioni tra i dati

- Uno dei metodi più semplici e versatili per organizzare le informazioni è archiviarle in **tabelle.**
- Il modello relazionale (utilizzato nella progettazione di database) è centrato su questa idea:
  - I dati sono organizzati in raccolte di tabelle bidimensionali chiamate "relazioni".

Tabella ORDINI

ordine	prodotti
1	tv
2	radio
3	radio
4	computer
5	tv

Tabella CLIENTI

cliente	ordine
Rossi	1
Verdi	2
Bianchi	3
Rossi	4
Verdi	5

ordine	prodotti	cliente
1	tv	Rossi
5	tv	Verdi

Relazioni binarie

Nella maggior parte dei casi siamo interessati ad una tabella costruita di **relazioni binarie** ovvero costituite dalla coppia (**CHIAVE**, **VALORE**), ad esempio

1. "Mario Rossi"
2. "Alda Bianchi"
3. "Silvia Verdi"

Obiettivo

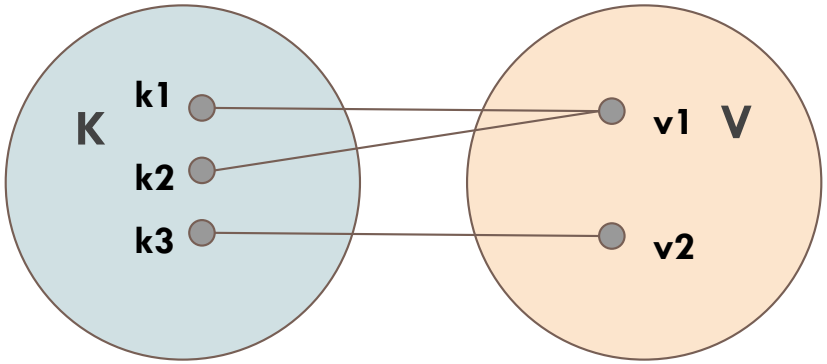
- Studiare le caratteristiche di ADT associative
  - Mappa (Map)
    - Chiave unica
  - Dizionario – Multimappa (Dictionary – Multimap)
    - Chiave non unica
  - Tabella (Table)
    - Chiave intera in un range noto a priori
  - Tabella Hash (Hash-table)
    - Chiave generica in un range non noto a priori

ADT Mappa
Map

Mappa: definizione matematica

- In una **Mappa (map)** ogni elemento dell'insieme delle chiavi e' collegato ad al piu' un solo elemento dell'insieme dei valori

$$K \times V = \{ (k, v) \mid k \in K \text{ and } v \in V \}$$

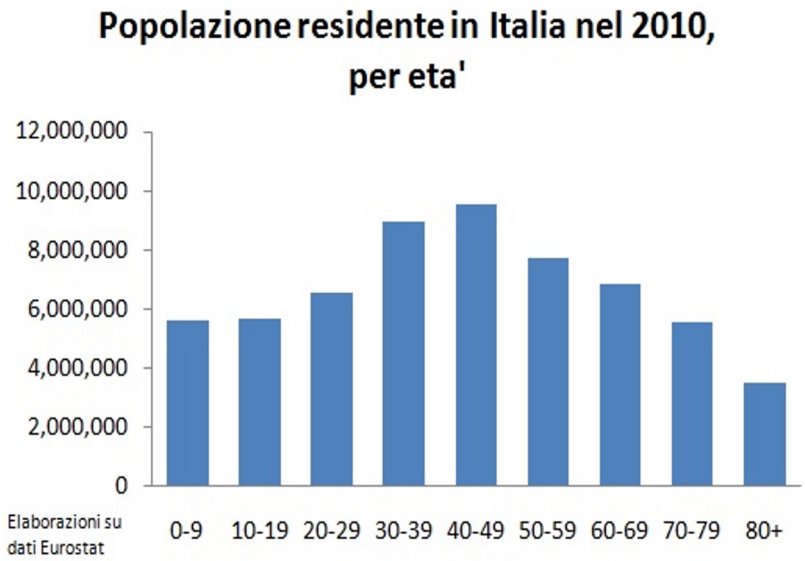


A ciascuna chiave corrisponde un solo valore

Esempio

Ad esempio potremmo considerare la funzione che descrive l'istogramma della popolazione italiana per classe di età come rilevata dal censimento del 2010.

Le **chiavi** in questo caso sono tutte **le classi di età individuate**. Mentre il **valore** e' il **numero di residenti in una data classe di età.**



ADT Mappa: definizione

- Un contenitore con le seguenti proprietà
  - Contiene dati che sono coppie (associazioni) di tipo **chiave/valore**
  - Non può contenere coppie con identica chiave: ogni chiave deve essere **unica** nell'insieme dei dati memorizzati
  - Consente di inserire nuove coppie chiave/valore
  - Consente di effettuare ricerca e rimozione di valori **usando la chiave come identificatore**
  - Una mappa realizza, quindi, una **funzione** (mapping) tra l'insieme delle chiavi e l'insieme dei valori

Mappa - interfaccia

```
public interface Map extends Container
{

   /**
      restituisce il valore al quale la chiave
      specificata e' associata, o null se questa mappa
      non contiene associazione per la chiave.
    */
   Object get(Object key);

   /**
      rimuove l'associazione di chiave specificata se
      questa e' presente nella mappa. Restituisce il
      valore associato (o null se non c'era
associazione)
   */
   Object remove(Object key);


   ...
}
```

Mappa - interfaccia

```
public interface Map extends Container
{
    . . .
    /**
    Inserisce l'associazione di chiave e valore specificati
    in questa mappa. Se la mappa gia' conteneva un'asso-
    ciazione con questa chiave (altrimenti null), il
    vecchio valore e' sostituito con il valore specificato
    e restituito in uscita.
    */
    Object put(Object key, Object value);

    /**
    restituisce un array contenente le chiavi di tutte le
    associazioni presenti nella mappa
    */
    Object[] keys();
}
```

Mappa con chiavi ordinabili

- Se si pone il vincolo aggiuntivo che le chiavi siano Comparable, è possibile definire un ADT "mappa ordinata", **SortedMap**
  - Aggiunge il solo metodo **sortedKeys**, che restituisce un array contenente le chiavi ordinate

```
public interface SortedMap extends Map
{
    Comparable[] sortedKeys();
}
```

- Le implementazioni di **sortedMap** devono però lanciare eccezioni ogni volta che viene fornita a **put** una chiave che non sia **Comparable**

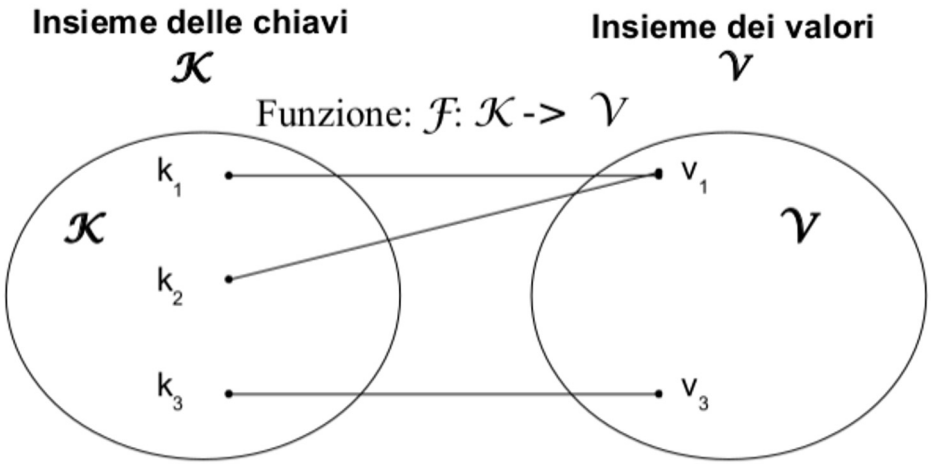
ADT Dizionario (Multimappa)

Dizionario

- L'**ADT dizionario** ha molte similitudini con l'**ADT mappa**
  - E' un contenitore di associazioni chiave/valore
  - Valgono tutte le proprietà dell'ADT mappa, tranne una
    - **Non** si richiede che le chiavi siano uniche nel dizionario

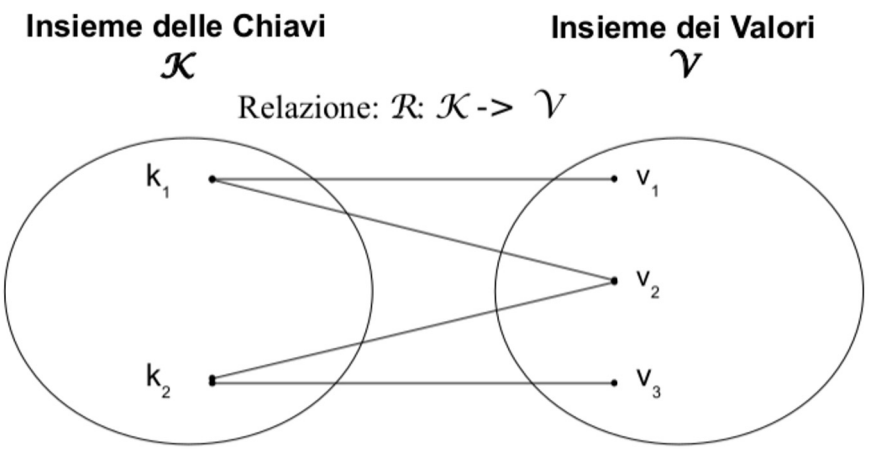
Mappa vs Dizionario

MAPPA



- A ciascun elemento del dominio (insieme delle chiavi) corrisponde uno e uno solo elemento del codominio (insieme dei valori)

DIZIONARIO



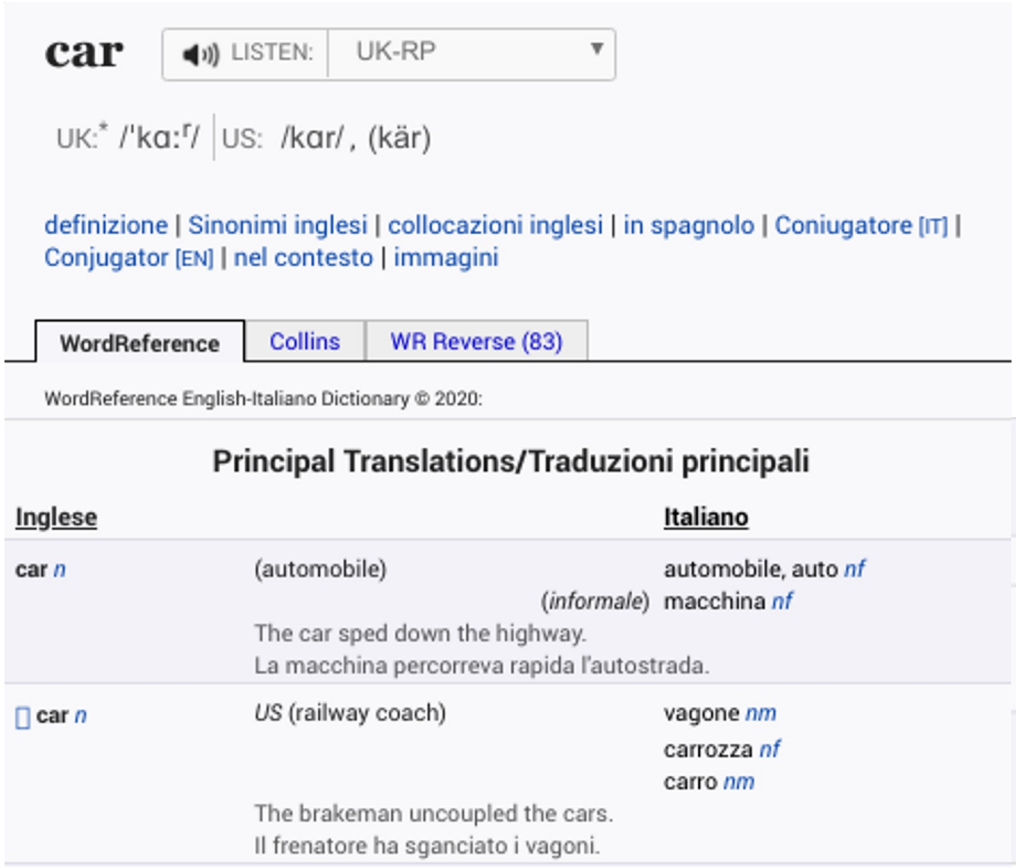
- A ciascun elemento del dominio (insieme delle chiavi) corrispondono uno o più elementi del codominio (insieme dei valori)

Dizionario

- C'è analogia con il dizionario di uso comune, in cui
  - Le **chiavi** sono le singole **parole**
  - I valori sono le **definizioni** delle parole nel dizionario
  - Le chiavi (parole) possono **essere associate a più valori** (definizioni) e quindi apparire più volte nel dizionario
  - La ricerca di un valore avviene tramite la sua chiave

Esempio

Dizionario che associa a una parola inglese tutte le possibili traduzioni.



car LISTEN: UK-RP

UK:* /ˈkɑːʳ/ | US: /kar/ , (kär)

definizione | Sinonimi inglesi | collocazioni inglesi | in spagnolo | Coniugatore [IT] | Conjugator [EN] | nel contesto | immagini

WordReference | Collins | WR Reverse (83)

WordReference English-Italiano Dictionary © 2020:

Principal Translations/Traduzioni principali

Inglese		Italiano
car *n*	(automobile)	automobile, auto *nf*
	(*informale*)	macchina *nf*
	The car sped down the highway.	
	La macchina percorreva rapida l'autostrada.	
car *n*	*US* (railway coach)	vagone *nm*
		carrozza *nf*
		carro *nm*
	The brakeman uncoupled the cars.	
	Il frenatore ha sganciato i vagoni.	
car *n*	*US* (streetcar)	tram *nm*
	I paid my fare as I stepped into the car.	
	Ho pagato il biglietto mentre salivo sul tram.	

Traduzioni aggiuntive

Inglese		Italiano
car *n*	(elevator)	ascensore *nm*
	(*di ascensore*)	cabina *nf*
	The car stopped on the second floor.	
	L'ascensore si è fermato al secondo piano.	

La traduzione della parola inglese "**car**" e' collegabile ai termini:
{ "automobile", "auto", "macchina", "vagone", "carrozza", "carro", "tram", "ascensore", "cabina" }

Dizionario

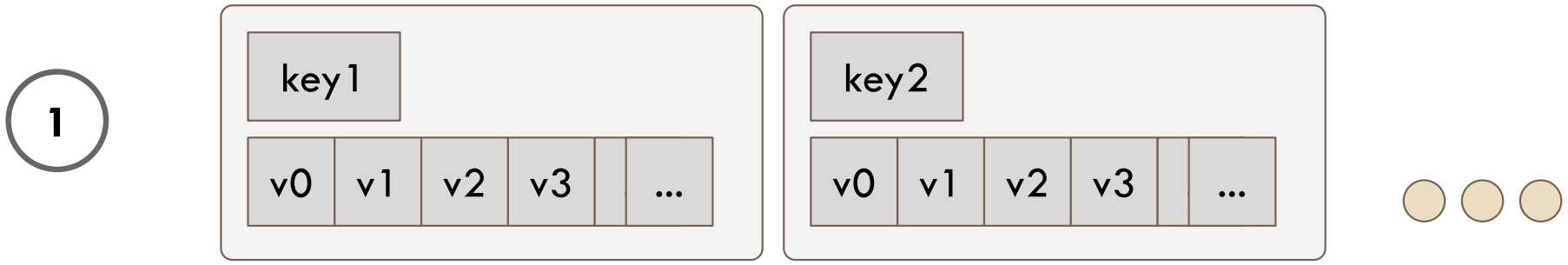
- Si distinguono **dizionari ordinati** e **dizionari non-ordinati**
  - A seconda che sull'insieme delle chiavi sia o no definita una relazione totale di ordinamento, cioè (in Java) che le chiavi appartengano ad una classe che implementa **Comparable**
  - Nella realizzazione del dizionario non ordinato, per ricercare e confrontare si usa il metodo equals()
  - Nella realizzazione del dizionario ordinato, per ricercare e confrontare si usa il metodo compareTo()

Dizionario: scelte implementative

- Dizionari con chiavi non ordinate
  - Sempre possibile
  - Inserimento veloce, ricerca lenta
- Dizionari con chiavi ordinate
  - Se le chiavi sono ordinabili
  - Inserimento lento, ricerca veloce

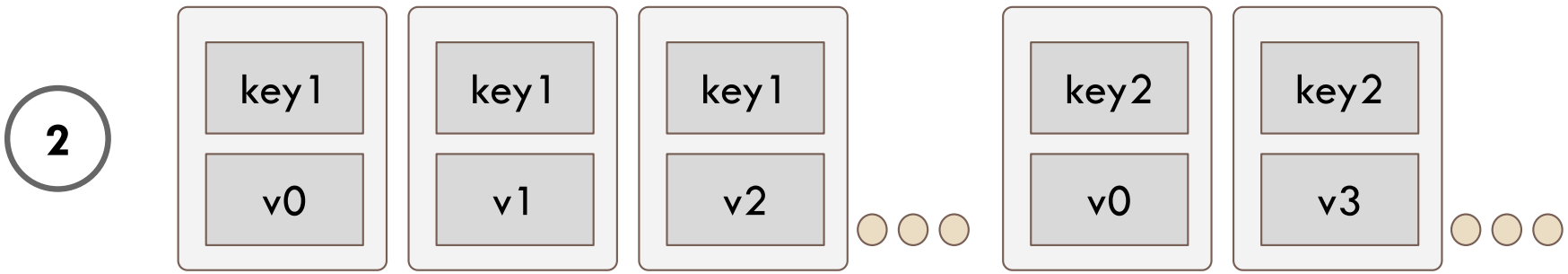
Dizionario: scelte implementative

- Gestione delle occorrenze multiple
  - Usiamo una struttura dati che abbia una chiave e un array di valori (simulo una mappa: array di coppie con chiave univoca associata a sua volta a un contenitore di valori).



Dizionario: scelte implementative

- Gestione delle occorrenze multiple
  - Manteniamo una struttura di coppia standard (chiave, valore)
  - Gestiamo la presenza di piu' chiavi identiche che associano valori diversi



Dizionario

```
public interface Dictionary extends Container
{
 /**
    Inserisce nel dizionario un'associazione avente
    chiave uguale a key e valore uguale a value.
    @param key la chiave specificata
    @param value il valore specificato
    @throws IllegalArgumentException se key o value sono null
  */
    void insert(Object key, Object value);

 /**
    Elimina dal dizionario un'associazione di chiave specificata
    restituendone il valore associato oppure null se non e'
    presente nel dizionario.
    @param key la chiave specificata
    @return un valore associato alla chiave specificata,
    se presente (di solito il primo trovato), o null altrimenti
 */
    Object remove(Object key);

}
```

Dizionario

```
public interface Dictionary extends Container
{
    . . .
   /**
    Se ci sono associazioni di chiave uguale a key, ne
    restituisce i valori, altrimenti restituisce un
    array vuoto. Le associazioni trovate sono eliminate dalla
    struttura dati.
    @param key la chiave specificata
    @return un array con i  valori associati alla chiave
    specificata, se presente, o un array vuoto altrimenti
    */
    Object[] removeAll(Object key);

   /**
    @return un array contenente le chiavi del
    dizionario, eventualmente ripetute. Restituisce un
    array vuoto (0 elementi) se il dizionario e' vuoto
    */
    Object[] keys();


}
```

Dizionario

```
public interface Dictionary extends Container
{
    . . .
  /**
    Restituisce uno dei valori associati alla chiave specificata
    @param key la chiave specificata
    @return uno dei valori associati se la chiave specificata e'
        presente, altrimenti null
    */
    Object find(Object key);

    /**
    Se il dizionario contiene una o più associazioni aventi chiave
    uguale a key, restituisce i valori, altrimenti restituisce un
    array vuoto
    @param key la chiave specificata
    @return un array con i valori associati alla chiave specificata,
    se presente, o un array vuoto se non presente
    */
    Object[] findAll(Object key);

    . . .
}
```

ADT Tabella
(Table)

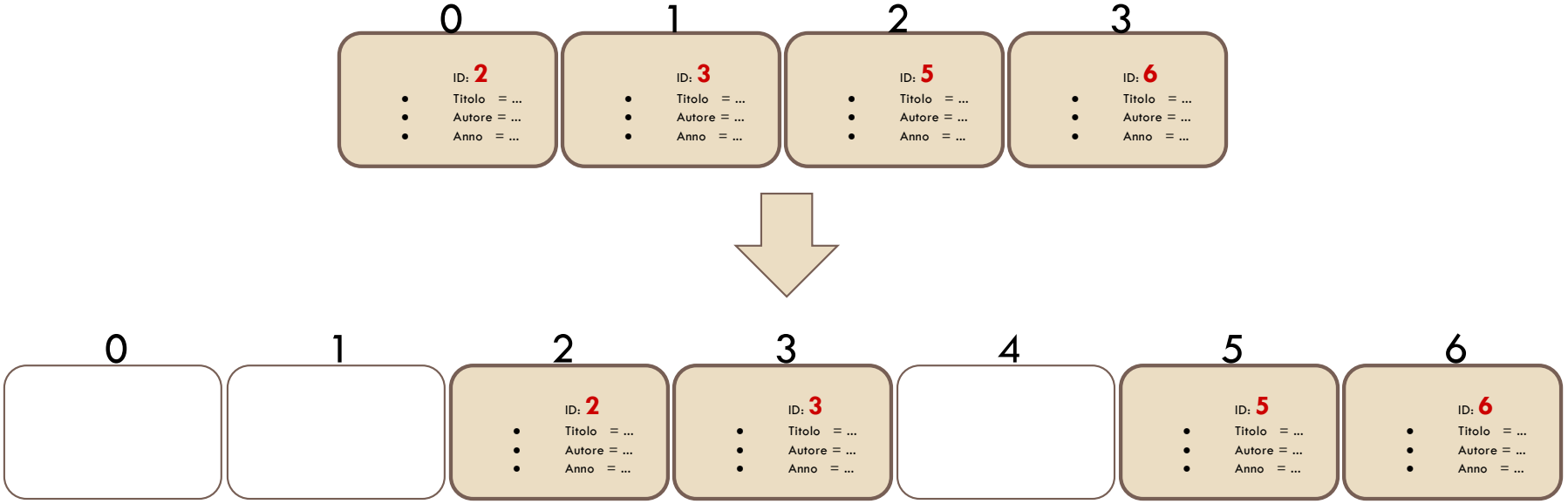
Chiavi numeriche

- Imponiamo una restrizione al campo di applicazione di una mappa
  - Supponiamo che le chiavi siano **numeri interi** in un intervallo noto a priori

  allora si può realizzare una mappa con

  prestazioni **O(1)** per tutte le operazioni
- Si usa un array che contiene soltanto i riferimenti ai valori, usando **le chiavi come indici nell'array**
  - Celle dell'array che hanno come indice una chiave che non appartiene alla mappa hanno valore **null**

Esempio biblioteca

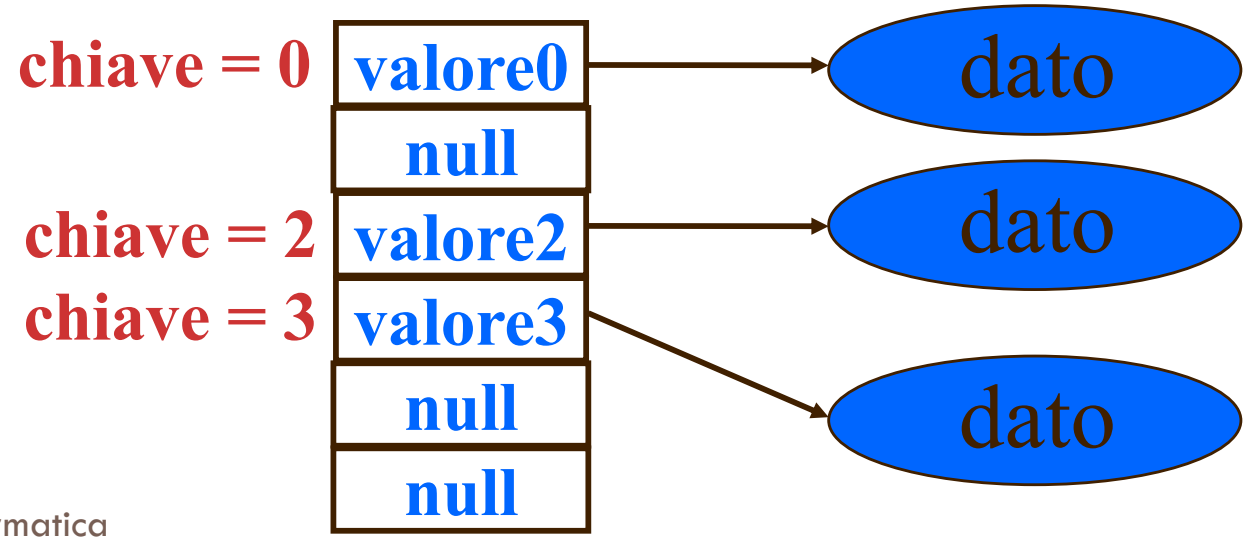


Accedere all'elemento ID = 2 corrisponde ad accedere al elemento di indice 2 dell'array.

Tempo di ricerca/accesso O(1) a scapito di possibile consumo di memoria per posizioni mancanti.

Tabella

- Una mappa con chiavi numeriche intere viene detta **tabella** (**table**)
- L'analogia è la seguente
  - un valore è una riga nella tabella
  - le righe sono numerate usando le chiavi
  - alcune righe possono essere vuote (senza valore)



Tabella

- Definiamo il tipo di dati astratto **Table** con un comportamento identico alla mappa
  - l'unica sostanziale differenza è che le chiavi non sono riferimenti ad oggetti di tipo **Comparable**, ma sono **numeri interi** (che evidentemente sono confrontabili)

```
public interface Table extends Container
{   void insert(int key, Object value);
    Object remove(int key);
    Object find(int key);
}
```

Tabella – punti deboli

- La tabella non utilizza la memoria in modo efficiente
  - l'occupazione di memoria richiesta per contenere n dati non dipende da n in modo lineare (come invece avviene per tutti gli altri ADT) ma dipende dal contenuto informativo presente nei dati
    - in particolare, dal valore della chiave massima
- Può essere necessario un array di milioni di elementi per contenere poche decine di dati
  - si definisce **fattore di riempimento** (**load factor**) della tabella il numero di dati contenuti nella tabella diviso per la dimensione della tabella stessa

Tabella – punti deboli

- La tabella è un mappa con prestazioni ottime
  - tutte le operazioni sono **O(1)**
- ma con le seguenti limitazioni
  - le **chiavi** devono essere **numeri interi** (non negativi)
    - in realtà si possono usare anche chiavi negative, sottraendo ad ogni chiave il valore dell'estremo inferiore dell'intervallo di variabilità
  - **l'intervallo di variabilità delle chiavi deve essere noto a priori**
    - per dimensionare la tabella (ed avere inserimento O(1))
  - se il fattore di riempimento è molto basso, si ha un grande **spreco di memoria**
    - ciò avviene se le chiavi sono molto "disperse" nel loro insieme di variabilità

Hash Table

Hash Table

- Cerchiamo di eliminare una delle limitazioni
  - **l'intervallo di variabilità delle chiavi deve essere noto a priori** per dimensionare la tabella
- Risolviamo il problema in modo diverso
  - fissiamo la dimensione della tabella in modo arbitrario
    - in questo modo si definisce di conseguenza l'intervallo di variabilità delle chiavi utilizzabili
  - per usare chiavi esterne all'intervallo, si **usa una funzione di trasformazione delle chiavi**
    - **funzione di hash**

Funzione di hash

- Una funzione di hash ha
  - come **dominio** l'insieme delle chiavi che identificano univocamente i dati da inserire nella mappa
  - come **codominio** l'insieme degli indici validi per accedere ad elementi della tabella
    - il risultato dell'applicazione della funzione di hash ad una chiave si chiama **chiave ridotta**
- Se manteniamo la limitazione di usare numeri interi come chiavi
  - una semplice funzione di hash è il calcolo del **resto della divisione intera tra la chiave e la dimensione della tabella**

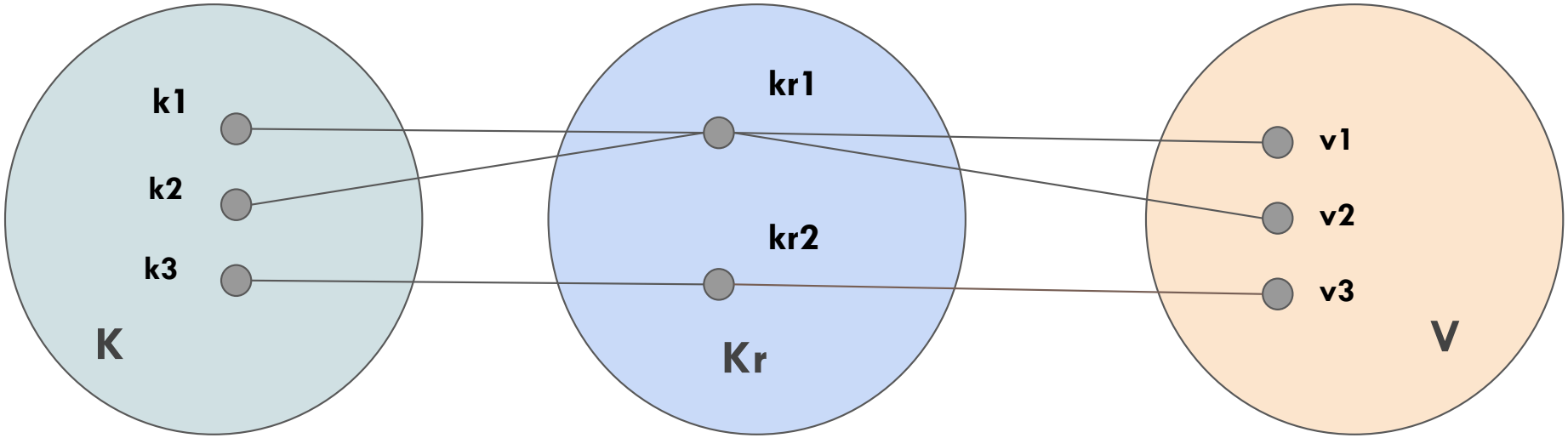
$$Kr = k \% dim$$

Funzione di hash

Insieme delle chiavi

Insieme delle chiavi ridotte

Insieme dei valori

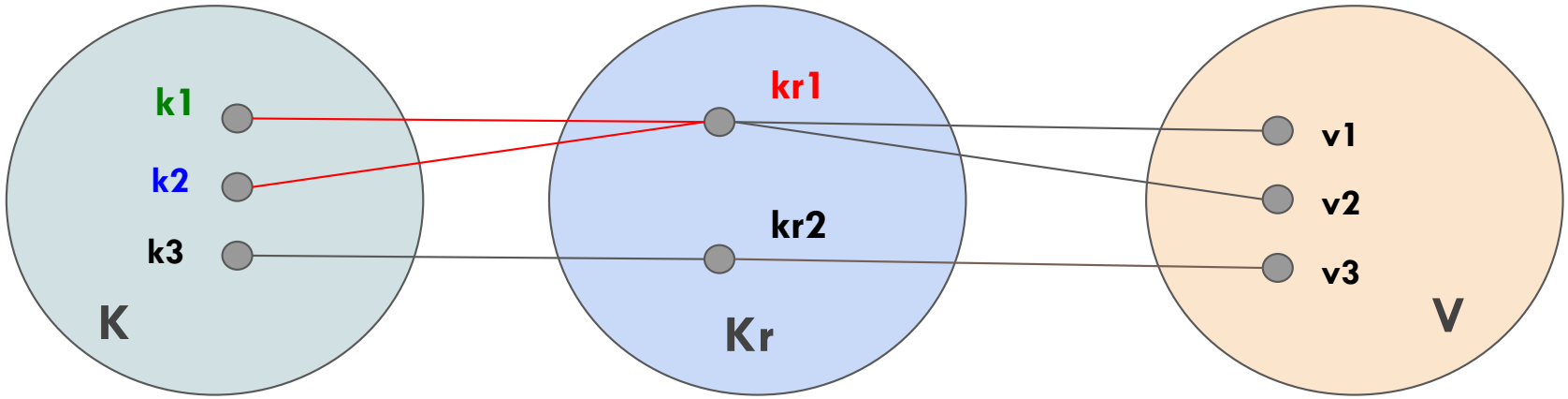


Funzione di hash

- Per come è definita, la **funzione di hash** è **generalmente non biunivoca**, cioè non è invertibile
  - **chiavi diverse possono avere lo stesso valore per la funzione di hash**
- Per questo si chiama funzione di **hash**
  - è una funzione che "**fa confusione**", nel senso che "mescola" dati diversi…

Funzione di hash

- Il fatto che la funzione di hash non sia univoca genera un problema nuovo
  - inserendo un valore nella tabella, può darsi che la sua chiave ridotta sia uguale a quella di un valore già presente nella tabella e avente una **diversa** chiave, **ma la stessa chiave ridotta**



Funzione di hash

- Usando i criteri già visto per la tabella
  - Il nuovo valore andrebbe a sostituire il vecchio
  - Il nuovo valore viene ignorato
- In entrambi i casi perdo un valore
  - questo non è corretto perché i due valori hanno, in realtà, chiavi diverse
- Questo fenomeno si chiama **collisione** nella tabella e si può risolvere in molti modi diversi

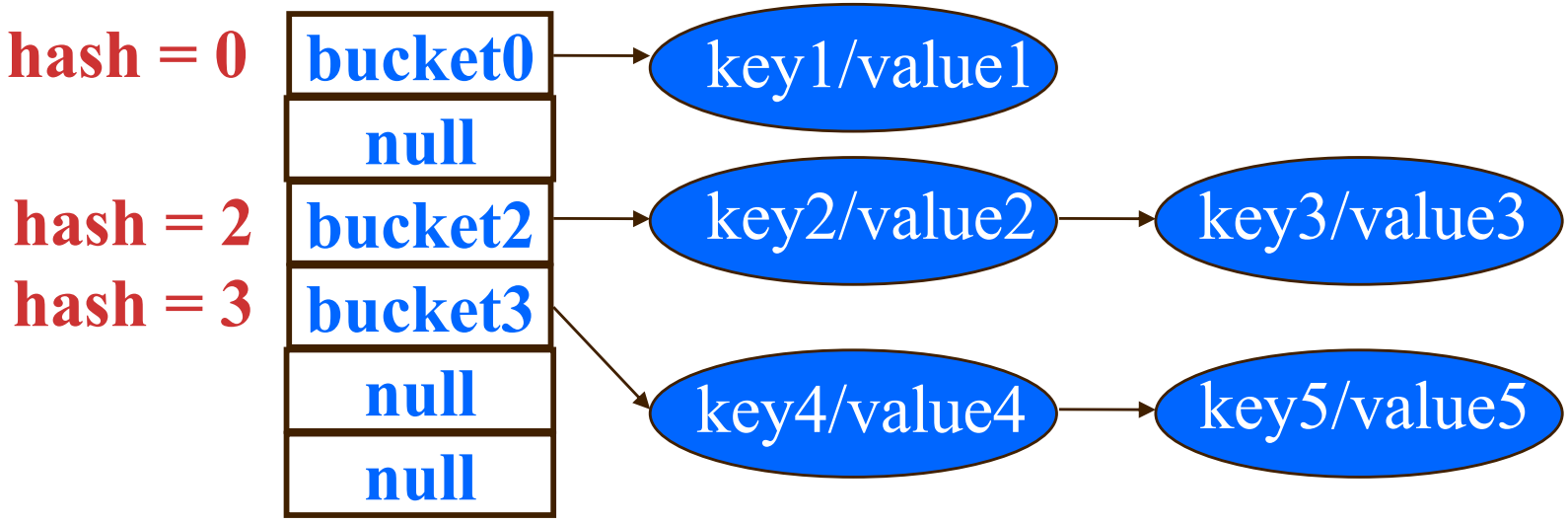
Risoluzione delle collisioni

- **ciascun valore deve essere memorizzato insieme alla sua vera chiave, per poter fare ricerche correttamente**

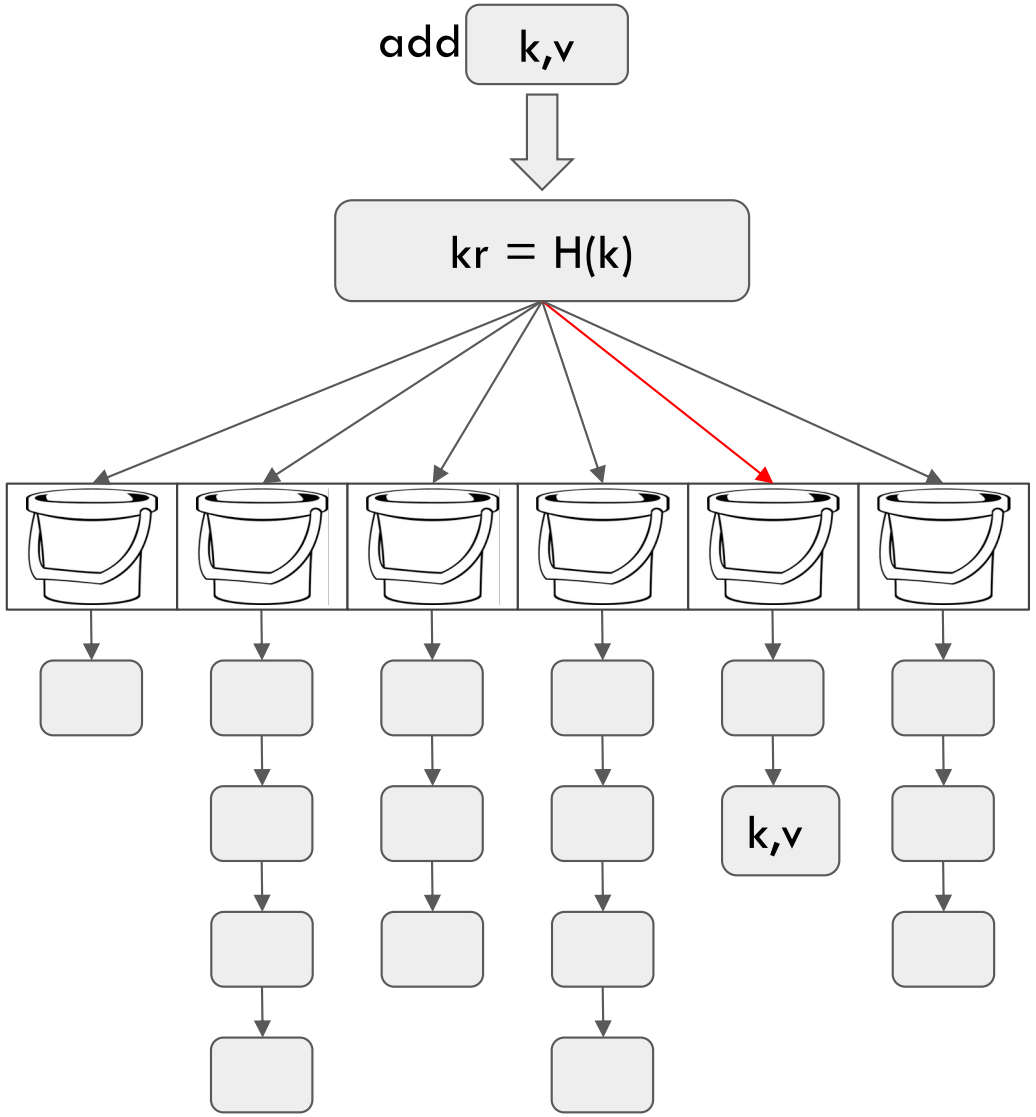
- Possiamo risolvere il problema **usando una lista** per ogni cella dell'array
  - Usiamo un array che è un array di riferimenti a liste
  - ciascuna lista contiene le coppie **chiave/valore** che hanno la stessa chiave ridotta

Risoluzione delle collisioni

- Questo sistema di risoluzione delle collisioni si chiama **tabella hash con bucket**
  - un **bucket** è una delle liste associate ad una chiave ridotta



Add in una hash table



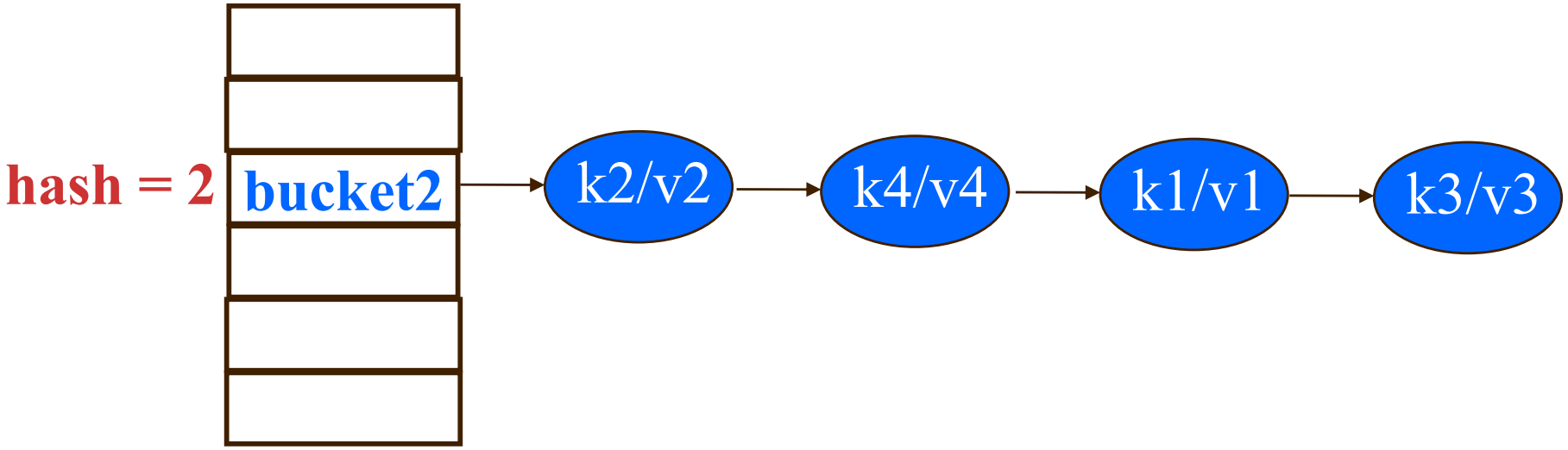
Risoluzione delle collisioni

- Le prestazioni della tabella hash con bucket non sono più, ovviamente, **O(1)** per tutte le operazioni

- **Le prestazioni dipendono fortemente da**
  - **Numero di bucket**
  - **Dalle caratteristiche della funzione di hash**

Impatto funzione di hash

- **caso peggiore**: la funzione di hash restituisce sempre la stessa chiave ridotta, per ogni chiave possibile
  - tutti i dati vengono inseriti in un'unica lista
    - le prestazioni della tabella hash degenerano in quelle di una lista
    - tutte le operazioni sono **O(n)**



Hash table con bucket

- **caso migliore**: la funzione di hash restituisce chiavi ridotte che si **distribuiscono uniformemente** nella tabella
  - Dato un insieme S di n elementi, la distribuzione uniforme attribuisce a ciascun elemento la stessa probabilità di occorrere
    - Es: in un dato non truccato, ogni faccia ha probabilità 1/6 di occorrere
    - Es: in una moneta non truccata, ogni faccia ha probabilità ½ di occorrere
    - Es: Se la nostra tabella ha 100 posizioni, quindi 100 chiavi ridotte, ciascuna chiave ridotta ha probabilità 1/100 di essere il risultato della funzione di hash

Hash table con bucket

- In questo caso tutte le liste associate ad una chiave ridotta hanno la stessa lunghezza media
  - se **M** è la dimensione della tabella, la lunghezza media di ciascuna lista è **n/M**
    - es: se ho n=100 chiavi e una tabella di dimensione M=20, ciascuna posizione(=chiave ridotta) avrà (circa) 5 elementi (cui corrispondono chiavi diverse trasformate nella stessa chiave ridotta)
  - tutte le operazioni sono **O(n/M)**
- per avere prestazioni **O(1)** occorre dimensionare la tabella in modo che **M** sia dello stesso ordine di grandezza di **n**

Riassumendo

- In una ***hash table con bucket*** si ottengono prestazioni ottimali (tempo-costanti) se:
  - la dimensione della tabella è circa uguale al numero di dati che saranno memorizzati nella tabella
    - fattore di riempimento circa unitario
      - così si riduce al minimo anche lo spreco di memoria
  - la funzione di hash genera chiavi ridotte uniformemente distribuite
    - liste di lunghezza quasi uguale alla lunghezza media

Insieme (Set)

Insieme (set)

- Il tipo di dati astratto "**insieme**" (**set**) è un contenitore (eventualmente vuoto) di oggetti **distinti** (cioè non contiene duplicati)
  - senza alcun particolare ordinamento
  - senza memoria dell'ordine temporale in cui gli oggetti vengono inseriti od estratti
  - si comporta come un insieme matematico

Set

```
public interface Set extends Container
{   void add(Object obj);
    boolean contains(Object obj);
    Object[] toArray();
}
```

- Le operazioni consentite sull'insieme sono
  - **inserimento** di un oggetto
    - fallisce silenziosamente se l'oggetto è già presente
  - **verifica della presenza** di un oggetto
  - **ispezione di tutti** gli oggetti
    - metodo che restituisce un array di riferimenti agli oggetti contenuti nell'insieme, senza alcun requisito di ordinamento di tale array (anche perché i dati non sono ordinabili…)

Set

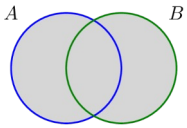
```
public interface Set extends Container
{   void add(Object obj);
    boolean contains(Object obj);
    Object[] toArray();
}
```

- **Non** esiste un'operazione di **rimozione**
  - si usa la sottrazione tra insiemi

Operazioni definite su insieme

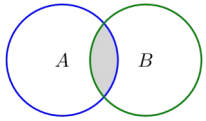
Per due insiemi A e B, si definiscono le operazioni caratteristiche di:

- **unione** A ∪ B



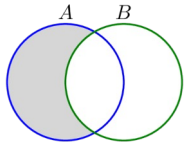
appartengono all'unione di due insiemi tutti e soli gli elementi che appartengono ad almeno uno dei due insiemi

- **intersezione** A ∩ B



appartengono all'intersezione di due insiemi tutti e soli gli elementi che appartengono ad entrambi gli insiemi

- **sottrazione** A - B (oppure anche A \ B)



appartengono all'insieme sottrazione A-B tutti e soli gli elementi che appartengono all'insieme A e non appartengono all'insieme B

non è necessario che B sia un sottoinsieme di A

Link utili

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Realizzazione della pila

Realizzazione della pila

- **Per realizzare una pila** è facile ed efficiente usare una struttura di tipo **array** "riempito solo in parte"
- In fase di realizzazione vanno affrontati due problemi
  - Cosa fare quando **l'array è pieno** e viene invocato il metodo **push**
    - la prima soluzione proposta prevede il **lancio di un'eccezione**
    - la seconda soluzione proposta usa il **ridimensionamento dell'array**
  - Cosa fare quando vengono invocati i metodi **pop** o **top** e la pila **è vuota**
    - Una possibile soluzione è lanciare un'eccezione

Eccezioni nella pila

- Definiamo due eccezioni (che sono classi), **EmptyStackException** e **FullStackException**, come estensione di **RuntimeException**, in modo che chi usa la pila non sia obbligato a gestirle

```
public class EmptyStackException extends RuntimeException
{   }

public class FullStackException extends RuntimeException
{   }
```

Pila senza ridimensionamento

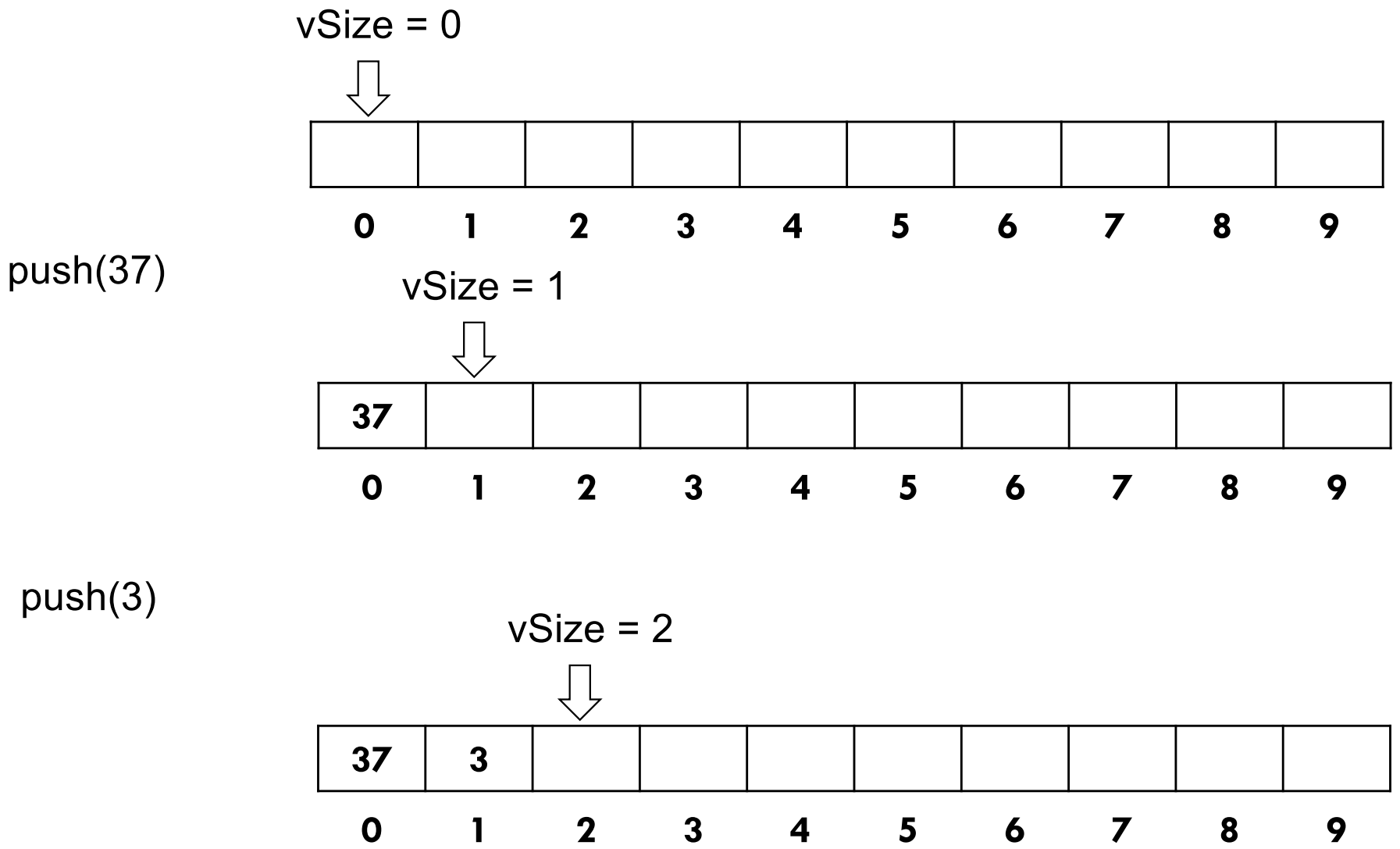
DIPARTIMENTO

```
public class FixedArrayStack implements Stack
{  // v è un array riempito solo in parte
   protected Object[] v;
   protected int vSize;
   public FixedArrayStack()
   {  v = new Object[100]; // 100 è un numero scelto
                           // a caso
      // per rendere vuota la struttura, invoco
      // il metodo makeEmpty, è sempre meglio evitare
      // di scrivere codice ripetuto
      makeEmpty();
   }
   // dato che Stack estende Container,
   // occorre realizzare anche i suoi metodi
   public void makeEmpty()
   {  vSize = 0;
   }
   public boolean isEmpty()
   {  return (vSize == 0);
   }
   ...
}
```

Pila senza ridimensionamento

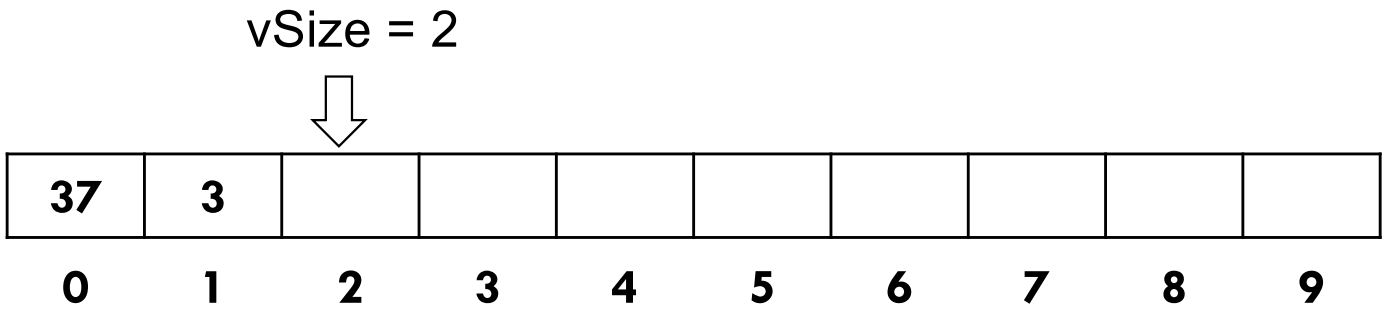
```
public class FixedArrayStack implements Stack
{   ...
    public void push(Object obj)
    {   if (vSize == v.length)
            throw new FullStackException();
        v[vSize++] = obj;
    }
    public Object top()
    {   if (isEmpty())
            throw new EmptyStackException();
        return v[vSize - 1];
    }
    public Object pop()
    {   Object obj = top();
        vSize--;
        return obj;
    }
}
```

Esempio

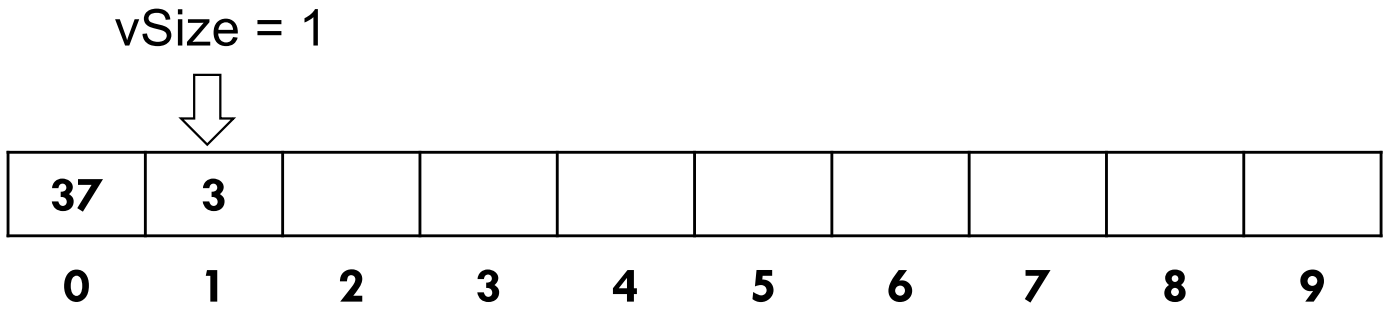

vSize = 0
0 1 2 3 4 5 6 7 8 9
push(37)
vSize = 1
37
0 1 2 3 4 5 6 7 8 9
push(3)
vSize = 2
37 3
0 1 2 3 4 5 6 7 8 9

Esempio

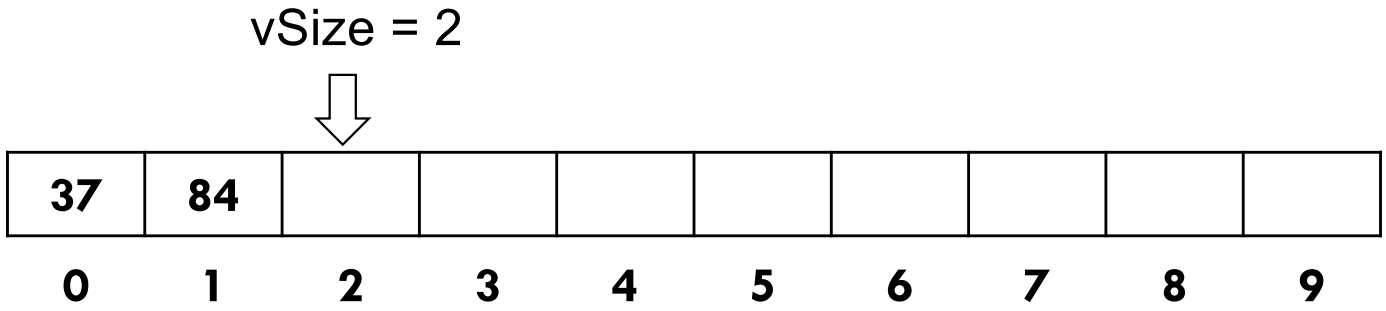
3 ⟸ top()



3 ⟸ pop()



push(84)



Pile di oggetti e pile di numeri

- Abbiamo visto che la pila così definita è in grado di gestire dati di qualsiasi tipo, cioè riferimenti ad oggetti di qualsiasi tipo (stringhe, conti bancari…)
- I tipi primitivi definiti nel linguaggio Java (**int, double, char**…) pero' non sono oggetti
- Possiamo gestire una pila di numeri come quella dell'esempio precedente? V. codice StackTester.java

Classi involucro e conversioni automatiche ("auto-boxing")

Auto-boxing

```
Stack s = new FixedArrayStack();
s.push(2);
```

- In Java 5.0 è stato introdotto l'auto-boxing
  - Se un valore di un tipo di dato primitivo viene assegnato ad una variabile della corrispondente classe involucro, viene implicitamente creato un esemplare di tale classe

```
Integer intObj = 2;
// equivale a Integer intObj = new Integer(2);
```

  - La stessa "conversione" implicita avviene se il valore di tipo primitivo viene assegnato ad una variabile di tipo Object

```
Object obj = 2;
// equivale a Object obj = new Integer(2);
```

Auto-boxing

- Attenzione:
  - Alcune conversioni che potrebbero apparire "ovvie" non funzionano con l'auto-boxing

```
Double doubleObj = 2.0;
// questo funziona...
Double doubleObj = 2;
// questo non viene compilato perché equivale a
// Double doubleObj = new Integer(2);
```

Pila con ridimensionamento

Pila con ridimensionamento

- Definiamo una pila che non generi mai l'eccezione **FullStackException**

```
public class GrowingArrayStack implements Stack
{
   public void push(Object obj)
   {  if (vSize == v.length)
         v = resize(v, 2*vSize);
      v[vSize++] = obj;
   }
   ... // tutto il resto è identico!
}
```

- Possiamo evitare di riscrivere tutto il codice di **FixedArrayStack** in **GrowingArrayStack**?

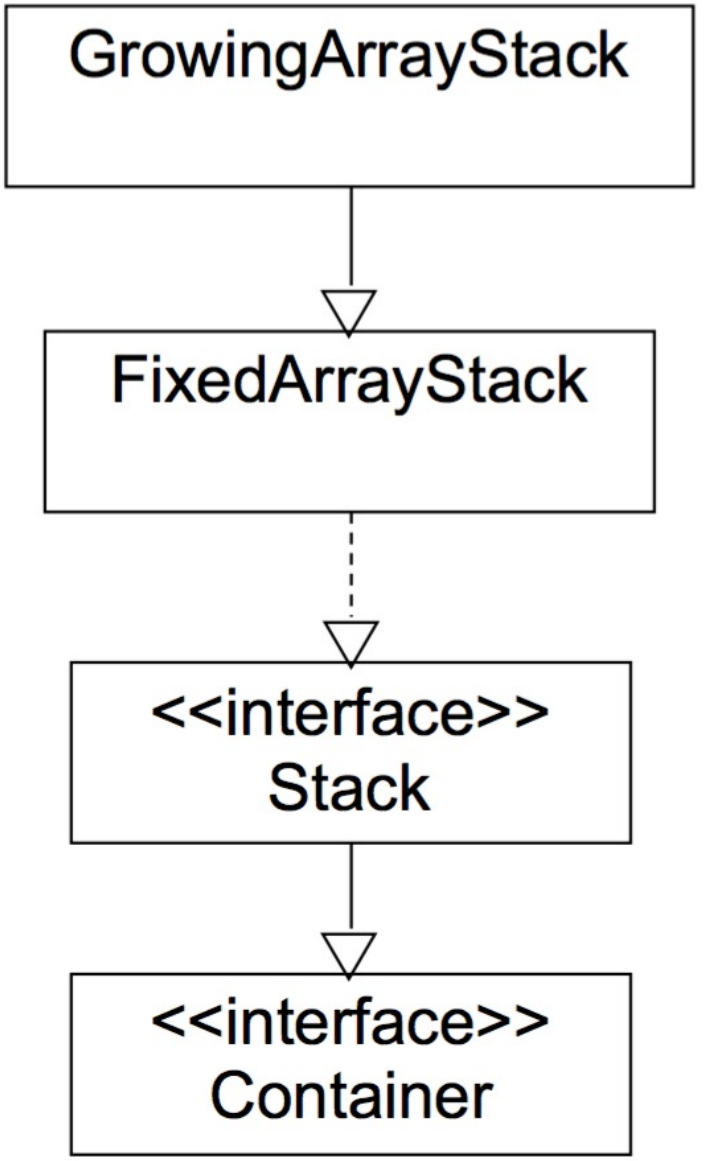
Pila con ridimensionamento

```
public class GrowingArrayStack extends FixedArrayStack{

   public void push(Object obj){
      if (vSize == v.length){
         v = resize(v, 2*vSize);
      }
      super.push(obj);
   }
}
```

- Il metodo **push** sovrascritto deve poter accedere alle variabili di esemplare della superclasse
- Questo è consentito dalla definizione **protected**
  - le variabili **protected** sono come private, ma vi si può accedere anche dalle classi derivate

Gerarchie di classi e interfacce

- Abbiamo realizzato la seguente **gerarchia** di classi e interfacce
  - L'interfaccia **Stack** estende l'interfaccia **Container**
  - La classe **FixedArrayStack** implementa l'interfaccia **Stack**
  - La classe **GrowingArrayStack** estende la classe **FixedArrayStack**



Prestazioni della pila e analisi ammortizzata

Prestazioni della pila

- Il tempo di esecuzione di ogni operazione su una **pila realizzata con array di dimensioni fisse** è **costante**, cioè non dipende dalla dimensione **n** della struttura dati stessa (non ci sono cicli…)
  - si noti che **le prestazioni dipendono dalla definizione della struttura dati** e non dalla sua interfaccia…
  - per valutare le prestazioni è necessario conoscere il codice che realizza le operazioni!

Prestazioni della Pila

- Nella realizzazione con array ridimensionabile, l'unica cosa che cambia è l'operazione push
  - "**alcune volte**" richiede un tempo **O(n)**
    - tale tempo è necessario per copiare tutti gli elementi nel nuovo array, all'interno del metodo **resize**
    - il ridimensionamento viene fatto ogni **n** operazioni
- cerchiamo di valutare **il costo medio** di ciascuna operazione
  - tale metodo di stima si chiama **analisi ammortizzata** delle prestazioni asintotiche

Analisi ammortizzata

- Dobbiamo calcolare il valore medio di n operazioni, delle quali
  - n-1 richiedono un tempo O(1)
  - una richiede un tempo O(n)

$$T(n) = [(n-1)*O(1) + O(n)]/n = O(n)/n = O(1)$$

- Distribuendo il tempo speso per il ridimensionamento in parti uguali a tutte le operazioni **push**, si ottiene quindi ancora **O(1)**

Analisi ammortizzata

- Le prestazioni di **push** con ridimensionamento rimangono **O(1)** per qualsiasi costante **moltiplicativa** usata per calcolare la nuova dimensione, anche diversa da 2
- Se, invece, si usa una costante **additiva**, cioè la dimensione passa da **n** a **n+k**, si osserva che su **n** operazioni di inserimento quelle "lente" sono **n**/**k**
  - Ad esempio se k=20, su 100 inserimenti ho 5 volte ridimensionamento
- **T(n)= [(n-n/k)*O(1)+(n/k)*O(n)]/n**
  = [O(n) + n*O(n)]/n
  = O(n)/n + O(n^2)/n
  = O(1) + O(n) = O(n)

Analisi ammortizzata

- Tecnica di **analisi delle prestazioni** di un algoritmo
- Si usa per mostrare che, in una sequenza di operazioni, il **costo medio** di una operazione è piccolo, anche se una singola operazione della sequenza è "costosa"
- E' diversa dall' **analisi di caso medio** vista in precedenza
  - Analisi di caso medio: si basa su stime statistiche dell'input
  - Analisi ammortizzata: fornisce il **costo medio** di una singola operazione **nel caso peggiore**

Schema riassuntivo delle prestazioni

operazione	FixedArray	GrowingArray
push	O(1)	O(1) *
top	O(1)	O(1)
pop	O(1)	O(1)

* Costo medio

Esercizio
Realizzazione di una pila "sicura"

Realizzazione di una "pila sicura"

- Tutte le strutture dati che abbiamo definito possono contenere oggetti di qualsiasi tipo
- Solitamente vengono utilizzate per contenere oggetti omogenei
  - cioè oggetti che siano tutti dello stesso tipo ma in generale questo vincolo non c'è

Realizzazione di una "pila sicura"

- Anche quando vengono usate come contenitori di oggetti omogenei, è possibile introdurre oggetti disomogenei, per errore
- Vogliamo realizzare una struttura dati, ad esempio una pila, che possa contenere soltanto oggetti di un certo tipo

```
// rivediamo la definizione della pila
public class ArrayStack implements Stack
{  private Object[] v = new Object[100];
   private int vSize = 0;

   public boolean isEmpty() { return vSize == 0; }
   public void makeEmpty() { vSize = 0; }

   public void push(Object obj)
   {  if (vSize == v.length) v = resize(v, 2*v.length);
      v[vSize++] = obj; }
   public Object top()
   {  if (isEmpty()) throw new EmptyStackException();
      return v[vSize - 1]; }
   public Object pop()
   {   Object obj = top();
       vSize--;
       return obj;
    }
private static Object[] resize(Object[] old, int newLength)
   {  Object[] newArray = new Object[newLength];
      for (int i = 0; i < old.length; i++)
         newArray[i] = old[i];
      return newArray;
    }
}
```

L'operatore instance of

L'operatore *instanceof*

- Sintassi: `variabileOggetto instanceof NomeClasse`
- Scopo: è un operatore booleano che restituisce **true** se e solo se la ***variabileOggetto*** contiene un riferimento ad un oggetto che è un esemplare della classe ***NomeClasse*** (o di una sua sottoclasse)
  - in tal caso l'assegnamento di ***variabileOggetto*** ad una variabile di tipo ***NomeClasse*** **NON** lancia l'eccezione **ClassCastException**
- Nota: il risultato non dipende dal tipo dichiarato per la ***variabileOggetto***, ma dal tipo dell'oggetto a cui la variabile si riferisce effettivamente al momento dell'esecuzione

Realizzazione di una "pila sicura"

- Per realizzare una pila che possa contenere soltanto stringhe, ad esempio
  - si può riscrivere tutto il codice modificando il solo metodo **push**

```
public class StringArrayStack implements Stack{
   ...
    public void push(Object obj){
       if (!(obj instanceof String)){
          throw new InvalidTypeException();
       }
       if (vSize == v.length){
          v = resize(v, 2*v.length);
       }
       v[vSize++] = obj;
    }
}
```

Realizzazione di una "pila sicura"

- Si potrebbe pensare di evitare il controllo di tipo nel metodo modificando il tipo del parametro

```
public class StringArrayStack implements Stack
{   ... // NON FUNZIONA
    public void push(String obj)
    {   if (vSize == v.length)
            v = resize(v, 2*v.length);
        v[vSize++] = obj; }
}
```

- **Non funziona:** il compilatore segnala che la classe non definisce tutti i metodi richiesti dall'interfaccia **Stack**
  - il metodo **push** ha una firma diversa

Realizzazione di una "pila sicura"

- Un modo più "elegante" sfrutta l'ereditarietà
  - che, come sappiamo, è spesso utile per evitare di ricopiare codice…

```
public class StringArrayStack extends FixedArrayStack
{  public void push(Object obj)
   {  if (!(obj instanceof String))
         throw new InvalidTypeException();
      super.push(obj);
   }
}
```

- Il metodo **push** viene sovrascritto (si noti la firma identica…), per cui negli esemplari di **StringArrayStack** verrà invocato il metodo qui definito e non quello di **ArrayStack**

Realizzazione di una "pila sicura"

- Vediamo come utilizzare **StringArrayStack**

```
Stack s = new StringArrayStack();
s.push("pippo");
String st = (String) s.pop();
s.push(new Integer(2)); // lancia un'eccezione
```

- Usando **StringArrayStack** è possibile fare il cast esplicito dopo l'estrazione di un elemento con la certezza che andrà a buon fine

Realizzazione di una "pila sicura"

- Viene la tentazione di ridefinire il metodo **pop** come segue, per non dover fare il cast

```
{   ...
    public String pop() { ... }
}
```

- Questo di nuovo non funziona, per lo stesso motivo citato in precedenza
  - il compilatore segnala che la classe non definisce tutti i metodi richiesti dall'interfaccia **Stack**
    - il metodo **pop** ha una firma diversa