

# Strutture dati elementari

Luca Tagliavini

March 8-10, 2021

## Contents

# 1 Tipi di dati

- **Tipo di dato primitivo:** Tipi di dato forniti direttamente dal linguaggio e con una diretta rappresentazione a livello macchina con operazioni native offerte direttamente dal calcolatore. Esempi possono essere *int*, *float*, *double*.
- **Tipo di dato astratto:** Tipo di dato composto e astratto, che combina al suo interno vari dati astratti o primitivi e offre un insieme di operazioni da svolgere su questi valori. (*astratto rispetto alla sua rappresentazione*)
  - Specifica: il manuale d'uso, l'interfaccia visibile a chi utilizza il nostro codice dall'esterno. Nasconde i dettagli implementativi e consente al progettista di esporre solo le operazioni che ritiene giusto.
  - Implementazione: il codice vero e proprio con cui viene realizzata la struttura dati. Non e' visibile allo sviluppatore che utilizza la libreria e due diverse implementazioni possono avere una grande disparita' di performance.

## 1.1 Strutture dati in generale

I dati sono organizzati in memoria tramite *strutture dati* in modo da organizzare i dati e offrire una serie di operazioni per manipolare la struttura.

Esistono diversi metodi di definire strutture dati e le loro interfacce pubbliche, come ad esempio le interfacce in Java o i template in C++.

### 1.1.1 Esempio: Dizionario

```
public interface Dizionario {  
    public void insert(Comparable key, Object value);  
    public void delete(Comparable key);  
    public Object search(Comparable key);  
}
```

Decidiamo per esempio di inserire gli elementi come chiavi  $(k, v)$  in una array dinamico **ordinato**. I nostri metodi faranno dunque le seguenti operazioni:

- **search:** ricerca binaria sull'array ( $O(\log_2 n)$ )
- **insert:** inserimento ordinato, quindi scorrendo l'array e trovando il punto giusto dove inserire e spostare tutti gli elementi successivi per fare spazio per il nuovo elemento ( $O(n)$ )
- **delete:** ricerca binaria dell'elemento e rimozione spostando a sinistra tutti gli elementi successivi ( $O(n)$ )

Decidiamo per esempio di inserire gli elementi come chiavi  $(k, v)$  in una lista. I nostri metodi faranno dunque le seguenti operazioni:

- **search:** ricerca sulla lista ( $O(n)$ )
- **insert:** inserimento a testa ( $O(1)$ )
- **delete:** ricerca dell'elemento e cambiamento del puntatore ( $O(n)$ )

Come si puo' notare *varie implementazioni hanno vari tradeoff*.

## 1.2 Tipi di strutture dati

- *Lineari:* Elementi in una sequenza dove ogni elemento ha un index (array)
- *Non lineari:* Elementi inseriti in una sequenza senza una linearita' in memoria (liste)
- *Statiche:* Numero di elementi sempre costante nel tempo
- *Dinamiche:* Numero di elementi cambia nel tempo
- *Omogenee:* Dati tutti dello stesso tipo
- *Non Omogenee:* Dati possono essere di tipo diverso

## 2 Strutture dati elementari

### 2.1 Strutture concatenate (collegate)

Due tecniche per rappresentare collezioni di elementi:

**strutture collegate (liste)** oppure **vettori indicizzati**. Analizzeremo in particolare le strutture collegate che sono piu' interessanti come implementazione:

Tutte le strutture collegate (liste) hanno in comune di essere una collezione di **record**:

- tutti i record sono collegati tra di loro
- i record sono numerabili partendo dalla testa o dalla coda
- ogni record contiene un dato

#### 2.1.1 Liste bilinkate

Per alcuni algoritmi puo' essere svantaggioso avere un riferimento solo al nodo successivo e si usano dunque liste bilinkate dove ogni record ha un puntatore all'elemento precedente e a quello successivo. Tuttavia, l'aggiunta di un puntatore occupa una word in piu' in memoria per record. Si ha dunque un minimo overhead sulla memoria.

### 2.1.2 Liste circolari

Possiamo oltretutto estendere le liste bilinkate in modo che il nodo precedente della lista in testa punti all'ultimo e il nodo successivo dell'ultimo record punti al primo. In questo modo si ha una lista bilinkata circolare che rende molto più veloci operazioni come rimozione in coda o aggiunta in coda.

Chiaramente con questo tipo di liste si può avere un problema di guardie per capire quando si è in testa o in coda nella lista. (basterebbe controllare quando si analizza il nodo testa per la seconda volta, al quale punto abbiamo dunque iterato su tutta la lista)

### 2.1.3 Liste circolari con nodo sentinella

Si possono implementare liste circolari (non pulitisse) come sopra ma inserendo in testa un nodo vuoto che svolge l'unico compito di segnalare quando si è giunti alla testa della lista.

## 2.2 Stack (o pile)

Le pile sono una struttura di dati molto diffusa e può essere pensata come una lista dove si aggiunge sempre in testa e si possono rimuovere elementi solo in testa.

In questo modo abbiamo un data flow LIFO (Last In, First Out). Alcuni metodi predefiniti per una pila potrebbero essere:

- **push**: aggiunge un elemento in cima alla lista
- **pop**: rimuove e restituisce l'elemento in cima alla lista
- **top**: restituisce (e non rimuove) il primo elemento sulla lista
- **isEmpty**: restituisce un booleano che indica se la lista è vuota

Alcuni linguaggi come Java Bytecode o Hack VM funzionano con una interfaccia a pile.

Ci sono due principali tecniche di implementazione:

- **liste concatenate**: una lista come descritta sopra dove si accede solo all'elemento in testa per i push/pop.  
Pro: grandezza infinita  
Con: più dispendiosa come memoria
- **array**: utilizziamo un array statico di dimensione fissa tenendo traccia dell'index dell'ultimo elemento aggiunto (cima della pila).  
Pro: meno dispendioso a livello di memoria  
Con: grandezza fissa limitata

## 2.3 Coda (o queue)

La struttura dati coda, a differenza della pila, segue una politica FIFO (First In, First Out).

Le operazioni standard sono:

- **enqueue**: aggiunge un elemento in coda
- **dequeue**: rimuove il primo elemento inserito nella coda
- **first**: restituisce il primo elemento della coda
- **isEmpty**: restituisce un booleano che indica se la lista e' vuota

Possibili implementazioni sono:

- **liste**: liste normali o bilinkate, che hanno una performance migliore, ma sono decisamente piu' complicate dal punto di vista implementativo. Serve tenere traccia della *head* e *tail* rispettivamente per l'estrazione e l'inserimento.
- **array circolari**: dimensione limitata, overhead minore.

## 3 Analisi ammortizzata

L'idea e' che se abbiamo una operazione che nel caso peggiore richiede  $O(n)$  ma nel casi normali richiede un  $O$  minore, possiamo fare un calcolo ammortizzato, che tiene conto del tempo risparmiato nelle chiamate meno costose e ci da un risultato piu' realistico e piu' basso rispetto a quello del caso pessimo.

La tecnica piu' facile e che vederemo e' detta *tecnica dei crediti*.

Partiamo dalla definizione di costo artificiale

$$T_{am}^i(n) = T_{ef}^i(n) + \text{deposito}^i(n) - \text{prelievo}^i(n)$$

Dove  $T_{am}^i$  e' il costo ammortizzato e  $T_{ef}^i$  e' il costo effettivo. Dimostriamo che:

$$\sum_{i=1}^k T_{am}^i(n) \geq T(n, k)$$

Sviluppiamo il primo membro per notare che

$$\begin{aligned} & \sum_{i=1}^k (T_{ef}^i(n) + \text{deposito}^i(n) - \text{prelievo}^i(n)) \\ & \sum_{i=1}^k T_{ef}^i(n) + \underbrace{\sum_{i=1}^k \text{deposito}^i(n) - \sum_{i=1}^k \text{prelievo}^i(n)} \end{aligned}$$

la differenza sottolineata e' sempre  $\geq 0$  dunque si ha che

$$\sum_{i=1}^k T_{am}^i(n) \geq \sum_{i=1}^k T_{ef}^i(n) = T(n, k)$$