Strutture dati di base

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

Definizioni di base

- **Struttura dati**: un'organizzazione sistematica dei dati e del loro accesso, che ne facilita la manipolazione
- **Algoritmo**: procedura suddivisa in passi elementari che, eseguiti in sequenza, consentono di svolgere un compito in tempo finito

Tipi di dato astratto

- Il tipo di dato astratto (Abstract Data Type — ADT) è un insieme di oggetti ed un insieme di operazioni definite su di esso
- L'ADT specifica **cosa** fa ogni operazione, non necessariamente **come**
- Tipicamente un ADT definisce delle operazioni che possono andare ad organizzare tipologie di dati differenti
- In Python, possiamo utilizzare il concetto di **Abstract Base Class** (ABC):
  - Definiamo delle classi che sono astratte per natura
  - Definiamo alcuni metodi all'interno di queste classi
  - Se una qualche nuova classe estende la classe ABC, diventa *obbligatorio* implementare questi metodi

Recap sulla tipizzazione in Python

- Python si basa sul concetto di “duck typing”:
  - *Se parla e si comporta come una papera, allora è una papera*
- Nel mondo dei linguaggi interpretati, l'interprete tenta di invocare un metodo su un oggetto appartenente ad una classe
- Se questo metodo esiste, allora va tutto bene
- Altrimenti, vengono generate eccezioni
  - i costrutti `try...except...finally` possono essere utili a gestire questi corner case, per evitare i crash delle applicazioni
  - si può anche utilizzare `hasattr` per verificare se un oggetto dispone dell'implementazione di un certo metodo

Recap sulla tipizzazione in Python

- Il duck typing ha degli effetti interessati:
  - Possiamo avere degli oggetti che si comportano come file: è sufficiente implementare il metodo `read` all'interno della classe
  - Possiamo avere degli oggetti su cui è possibile iterare (*iterable*): è sufficiente implementare il metodo `__iter__`
- Un oggetto, indipendentemente dalla sua classe o tipo, può essere conforme ad una certa *interfaccia* in funzione del *protocollo* che questo implementa
  - Alcuni esempi: `__len__`, `__contains__`, `__iter__`, …

Un esempio

```
class Team:
    def __init__(self, members):
        self.__members = members

    def __len__(self):
        return len(self.__members)

    def __contains__(self, member):
        return member in self.__members
```

Oggetti Contenitori

- Nella programmazione a oggetti, si tratta di una classe di oggetti il cui unico scopo è quello di contenere altri oggetti
- Formano strutture dati con alcuni metodi già implementati, per manipolare quindi *collezioni* di oggetti

- Tipicamente le librerie standard di molti linguaggi (es, Java, C++) forniscono una grande quantità di classi contenitori
  - Basate su template/generics

Un esempio in Java

```
public class Gen<X,Y> {
  private final X var1;
  private final Y var2;

  public Gen(X x,Y y) {
    var1 = x;
    var2 = y;
  }

  public X getVar1() {
    return var1;
  }

  public Y getVar2() {
    return var2;
  }

  public String toString() {
    return "(" + var1 +
           ", " + var2 +
           ")";
  }
}
```

Tipi di dato astratto

- Ci sono dei casi in cui i protocolli non sono sufficienti
- Per esempio, un `Uccello` ed un `Aereo` possono entrambi implementare il metodo `vola()`.
- Tuttavia, due oggetti di queste classi non sono la stessa cosa, anche se implementano lo stesso metodo!

- I tipi di dato astratto, permettono di rendere più formale la definizione di cosa un oggetto può/non può fare.

Abstract Base Classes

- Il concetto dietro le ABC è quello di definire classi che sono per loro natura astratta

```
import abc
```

modulo per definire ABC, definisce la metaclasse `ABCMeta`

```
class Uccello(abc.ABC):
    @abc.abstractmethod
    def vola(self):
        pass
```

`abc.ABC` è una classe che ha `ABCMeta` come metaclasse: ereditiamo da questa

Annotazione per indicare che si tratta di un metodo astratto

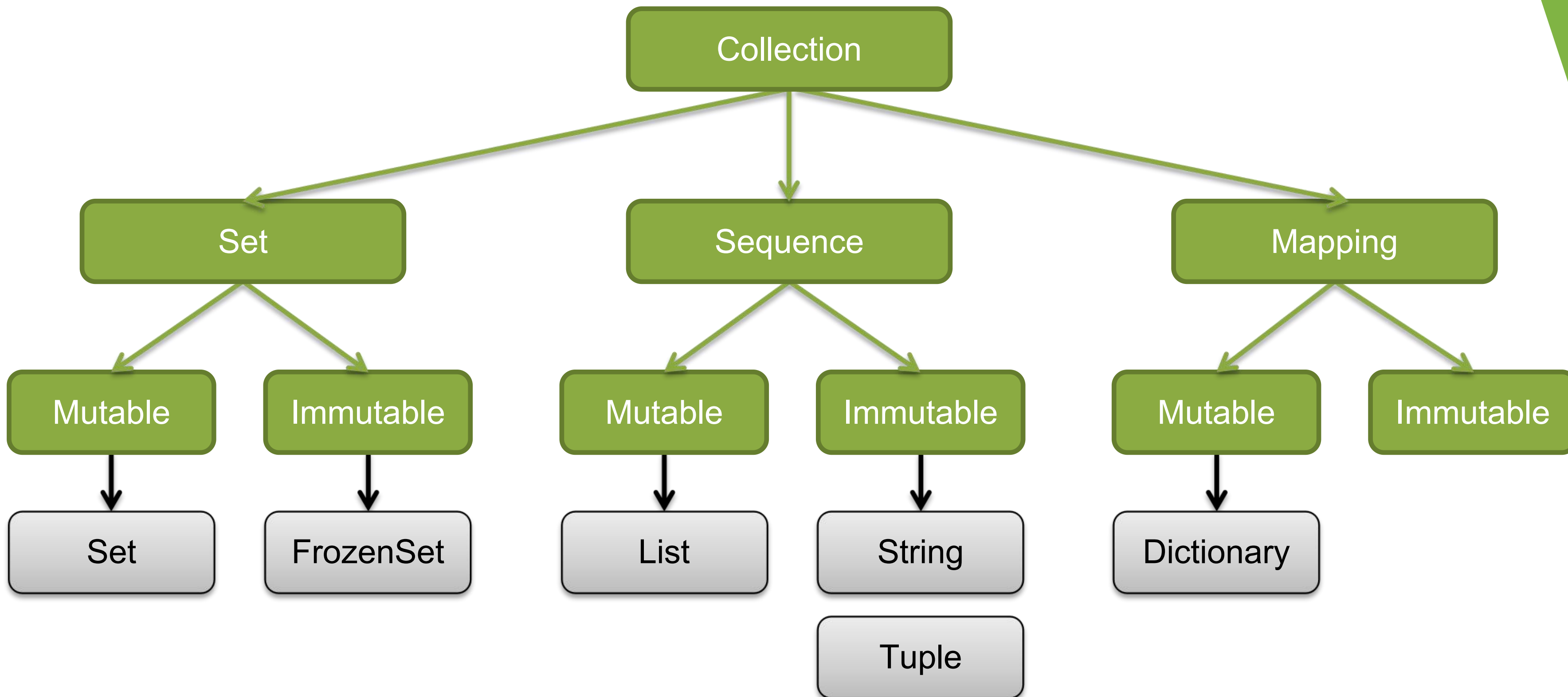
Mixins

- A volte, può essere interessante definire delle implementazioni di metodi che sono applicabili a più classi
- I mixin sono delle classi che implementano dei metodi che possono essere inclusi all'interno di altre classi, sfruttando l'ereditarietà multipla di Python

Alcune ABC di base

ABC	Eredita da	Metodi astratti	Mixin
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	Iterable	__next__	__iter__
Reversible	Iterable	__reversed__	
Sized		__len__	
Callable		__call__	
Collection	Sized, Iterable, Container	__contains__, __iter__, __len__	
Sequence	Reversible, Collection	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
Set	Collection	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __xor__, and isdisjoint
Mapping	Collection	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, and __ne__

Gerarchia di ABC e tipi di dato built-in



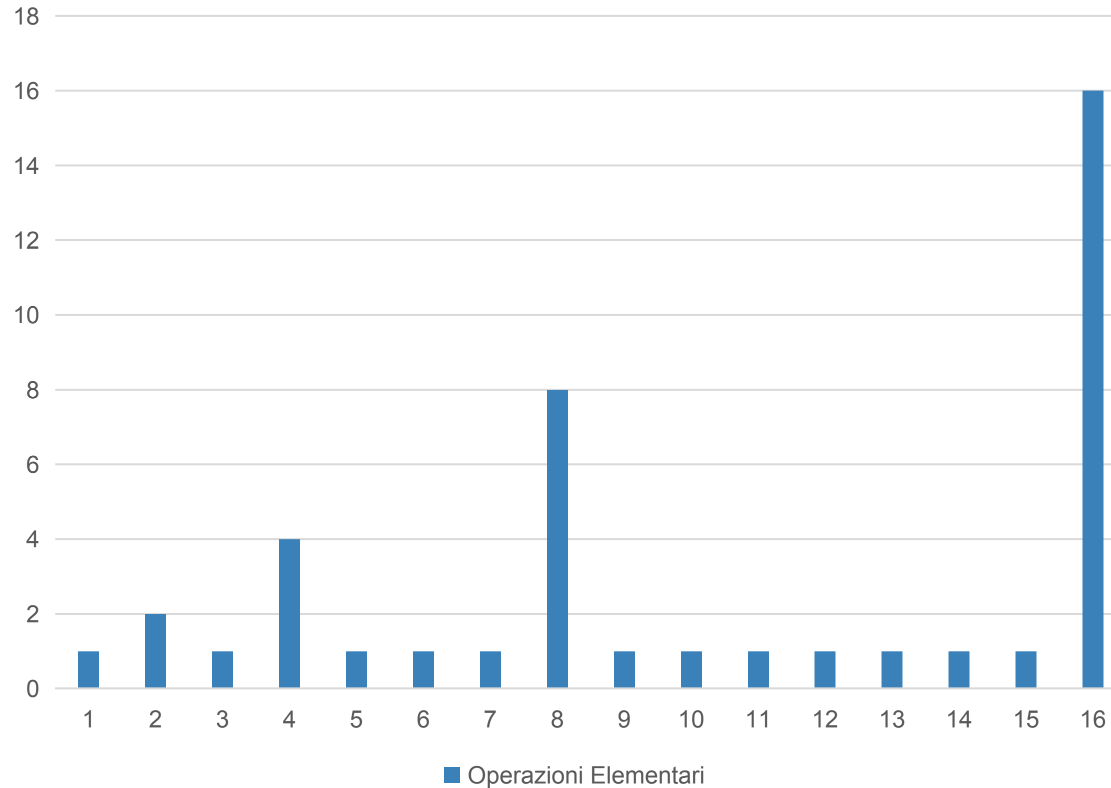
Esempio giocattolo: Vector

- Definiamo una classe Vector che rappresenta un vettore matematico
- Definiamo per questa classe un insieme di operazioni fondamentali:
  - Calcolo della norma
  - Calcolo del vettore normalizzato
  - Rotazione di un vettore
  - Moltiplicazione per matrice
  - Prodotto scalare
  - Prodotto vettoriale
  - Divisione, somma, sottrazione
  - Iterazione sulle componenti
  - …

Esempio giocattolo: Array Dinamico

- Si tratta di un tipo di dato astratto che può contenere al suo interno oggetti di qualsiasi tipo
- Basato sull'ABC Collection
- Fornisce un'unica operazione esplicita: `append()`
- Internamente conserva riferimenti agli oggetti passati all'interno di una lista

Analisi ammortizzata



Analisi ammortizzata

- Utilizziamo la tecnica degli accantonamenti
  - L'esecuzione del metodo `append()` richiede il "pagamento" di un certo importo, variabile a seconda del numero di elementi presenti nell'array dinamico
  - L'importo totale dipende dalla sequenza di operazioni
  - Possiamo "caricare" di più il costo di alcune operazioni a costo più basso per andare "a credito"

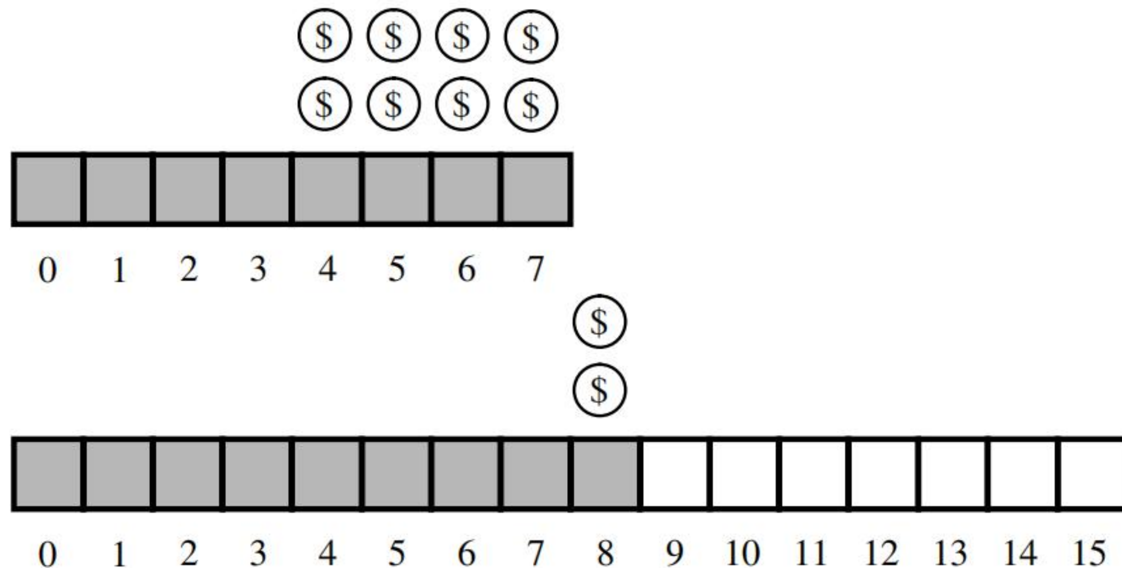
- Assunzioni:
  - Il costo di inserimento di un elemento è 1
  - Il costo per portare la dimensione di un vettore da k a $2k$ è pari a k (costo di inizializzazione del nuovo vettore)

Analisi ammortizzata

- Associamo ad ogni invocazione di `append()` un costo pari a 3
  - Carichiamo il costo di un credito pari a 2

- Quando l'array contiene $S = 2^i$ elementi, con $i \geq 0$, dobbiamo raddoppiare la dimensione dell'array
  - Questa operazione ha un costo 2^i
  - Il costo può essere recuperato dalle operazioni precedenti tra la 2^{i-1}-esima e la $(2^i - 1)$-esima

Analisi ammortizzata



- Pertanto, il costo di *n* invocazioni di append() può essere pagato con un credito di *3n*.
  - Il costo ammortizzato di un'operazione è $O(1)$

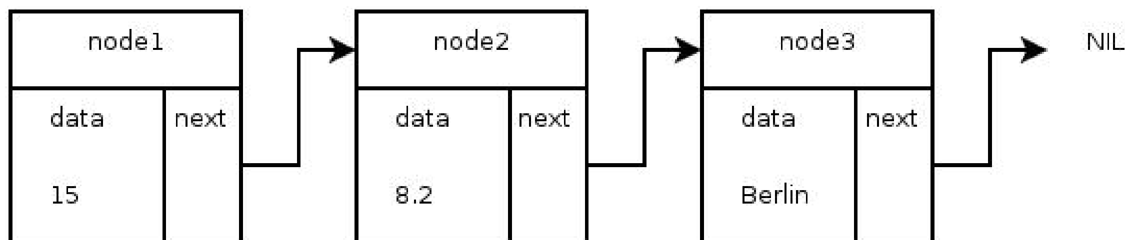
Liste Collegate

Liste collegate (o concatenate)

- Si tratta di una delle strutture dati fondamentali dell'informatica
- Permette di annidare all'interno di “nodi” dei dati (tipicamente dello stesso tipo)
  - Tipicamente implementate tramite:
    - vettori dinamici (le liste standard di Python)
    - oggetti collegati
- Fornisce un insieme di operazioni fondamentali:
  - Inserimento, scansione, eliminazione, ricerca
- Alcune varianti:
  - Liste singolarmente collegate
  - Liste doppiamente collegate
  - Liste con nodo testa e coda
  - Liste circolari

Liste singolarmente collegate

- Una *lista singolarmente collegata*, nella sua forma più semplice, è una collezione di nodi che forma una sequenza lineare. Ciascun nodo conserva un riferimento ad un oggetto che è un elemento della sequenza, ed un riferimento al nodo successivo della lista



Alcune definizioni

- **Nodo testa**: il primo nodo della lista collegata
- **Nodo coda**: l'ultimo nodo della lista collegata. Ha il suo successore impostato a NULL (`None` in python)
- **Attraversamento**: un'operazione che parte dal nodo testa e che, attraversando ciascun nodo navigando il riferimento all'elemento successivo, raggiunge il nodo coda

- La dimensione della lista non è nota a priori
  - utilizza una quantità di memoria proporzionale al numero di elementi attualmente contenuti

Alcune operazioni da supportare

- Inserimento in testa
- Inserimento in coda
- Inserimento in una certa posizione
- Rimozione di un elemento in testa
- Rimozione di un elemento in coda
- Rimozione dell'elemento i-esimo
- Ricerca di un elemento associato ad un certo valore

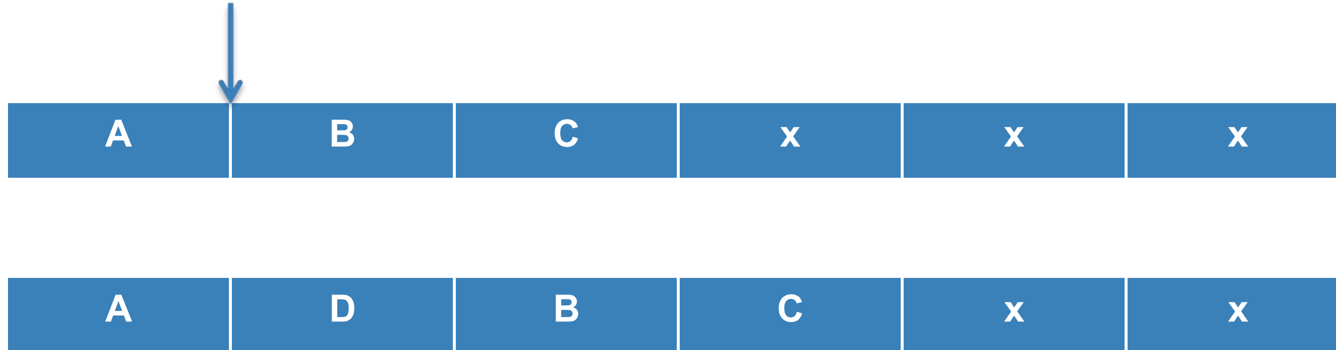
- Come possiamo realizzare queste operazioni?

Implementazione con array

- L'implementazione con array può fornire benefici dal punto di vista del consumo della memoria
- Le operazioni di inserimento/eliminazione in una data posizione richiedono però un costo maggiore

Implementazione con array

- L'implementazione con array può fornire benefici dal punto di vista del consumo della memoria
- Le operazioni di inserimento/eliminazione in una data posizione richiedono però un costo maggiore



Confronto dei costi

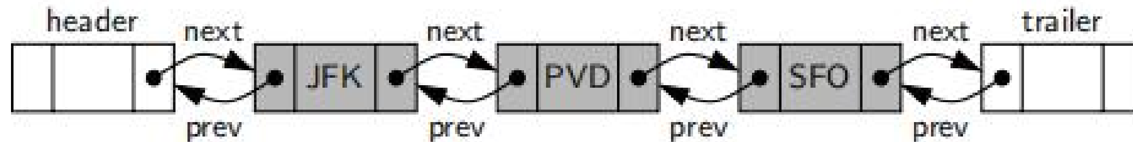
	Lista collegata	Array
Cerca elemento k-esimo		
Ricerca di un elemento		
Inserimento in posizione k-esima		
Inserimento testa/coda		
Eliminazione elemento k-esimo		

Confronto dei costi

	Lista collegata	Array
Cerca elemento k-esimo	$O(k)$	$O(1)$
Ricerca di un elemento	$O(n)$	$O(n)$
Inserimento in posizione k-esima	$O(k)$	$O(k)$
Inserimento testa/coda	$O(1)$	$O(1)$
Eliminazione elemento k-esimo	$O(k)$	$O(n)$

Liste doppiamente collegate

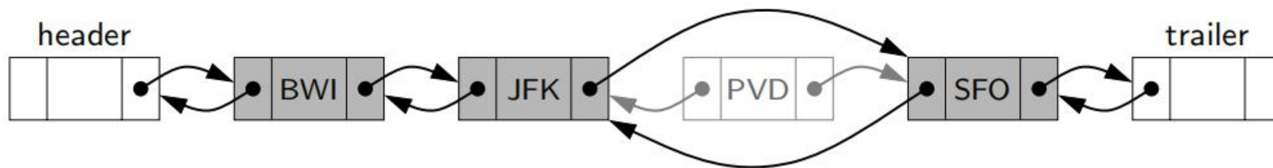
- Questa variante della lista collegata utilizza, per ciascun nodo, due riferimenti
- Permette di risolvere facilmente il problema dell'attraversamento della lista al contrario
- L'utilizzo di nodi sentinella consente di eliminare casi particolari nell'eliminazione/inserimento dei nodi all'inizio ed alla fine



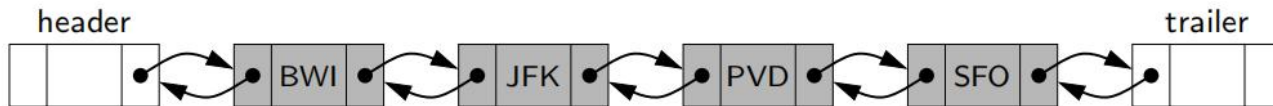
Inserimento in una lista doppiamente collegata

- L'inserimento di un nodo richiede l'aggiornamento dei puntatori next/prev nei nodi adiancenti

(a)



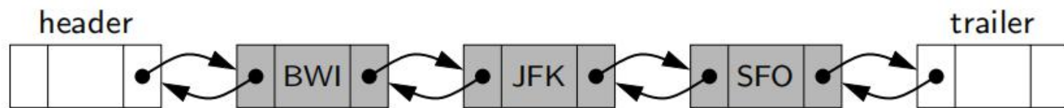
(b)



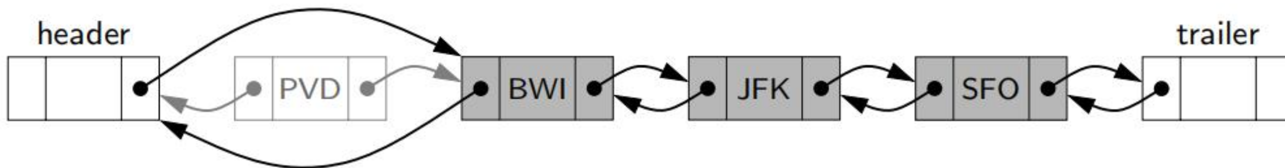
(c)

Inserimento in una lista doppiamente collegata

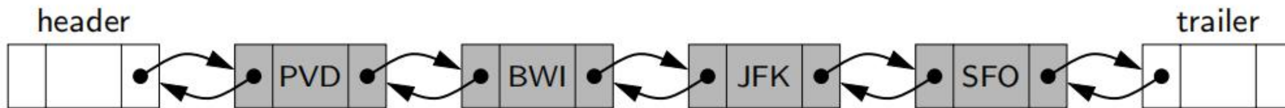
- L'inserimento di un nodo richiede l'aggiornamento dei puntatori next/prev nei nodi adiancenti



(a)



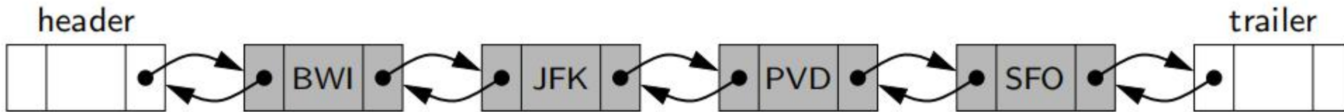
(b)



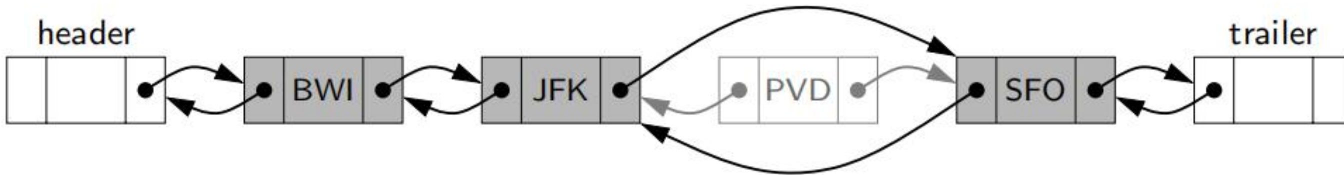
(c)

Eliminazione da una lista doppiamente collegata

- In maniera similare all'inserimento, occorre aggiornare i puntatori di entrambi i nodi adiacenti



(a)



(b)

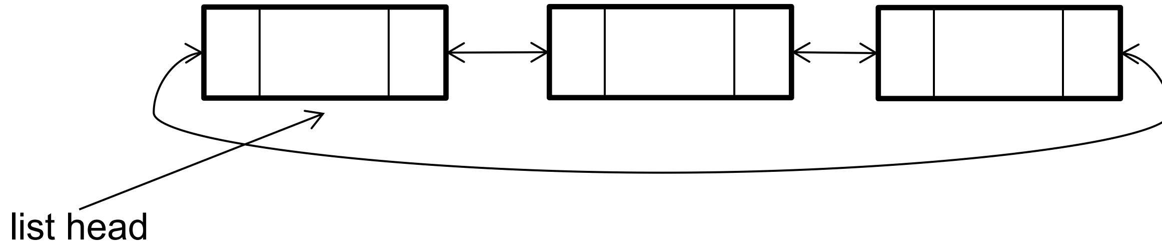
(c)

Vantaggi di una lista doppiamente collegata

- Con la costruzione della struttura dati mostrata, possiamo smettere di riferire le posizioni degli elementi parlando di indici ed accedere direttamente ai nodi
- In questo modo, è possibile utilizzare riferimenti ai nodi per velocizzare le implementazioni delle operazioni di inserimento ed eliminazione
  - È possibile arrivare ad implementazioni di costo $O(1)$
  - Diventa una struttura dati fortemente migliore rispetto alle liste basate su array dinamici

Liste circolari

- Una lista circolare fornisce un modello più generale, per dati che non hanno una nozione particolare di “inizio” e “fine”
- In una lista di questo tipo, è possibile effettuare una scansione (in avanti o indietro) a partire da un nodo qualsiasi
- È sufficiente mantenere un riferimento a un elemento qualsiasi della lista per poterla navigare



Deque

- La deque (pronunciata *deck*) è una “double-ended” queue, tipicamente tradotta con “lista testa-coda”
- È una variante della lista doppiamente concatenata che consente inserimenti/rimozioni unicamente dalla testa o dalla coda

- È possibile implementarla mediante array dinamico (costo delle operazioni $O(1)$ ammortizzato) o mediante lista doppiamente concatenata (costo delle operazioni $O(1)$)
  - Esercizio: modificare le implementazioni viste a lezione per realizzare una deque

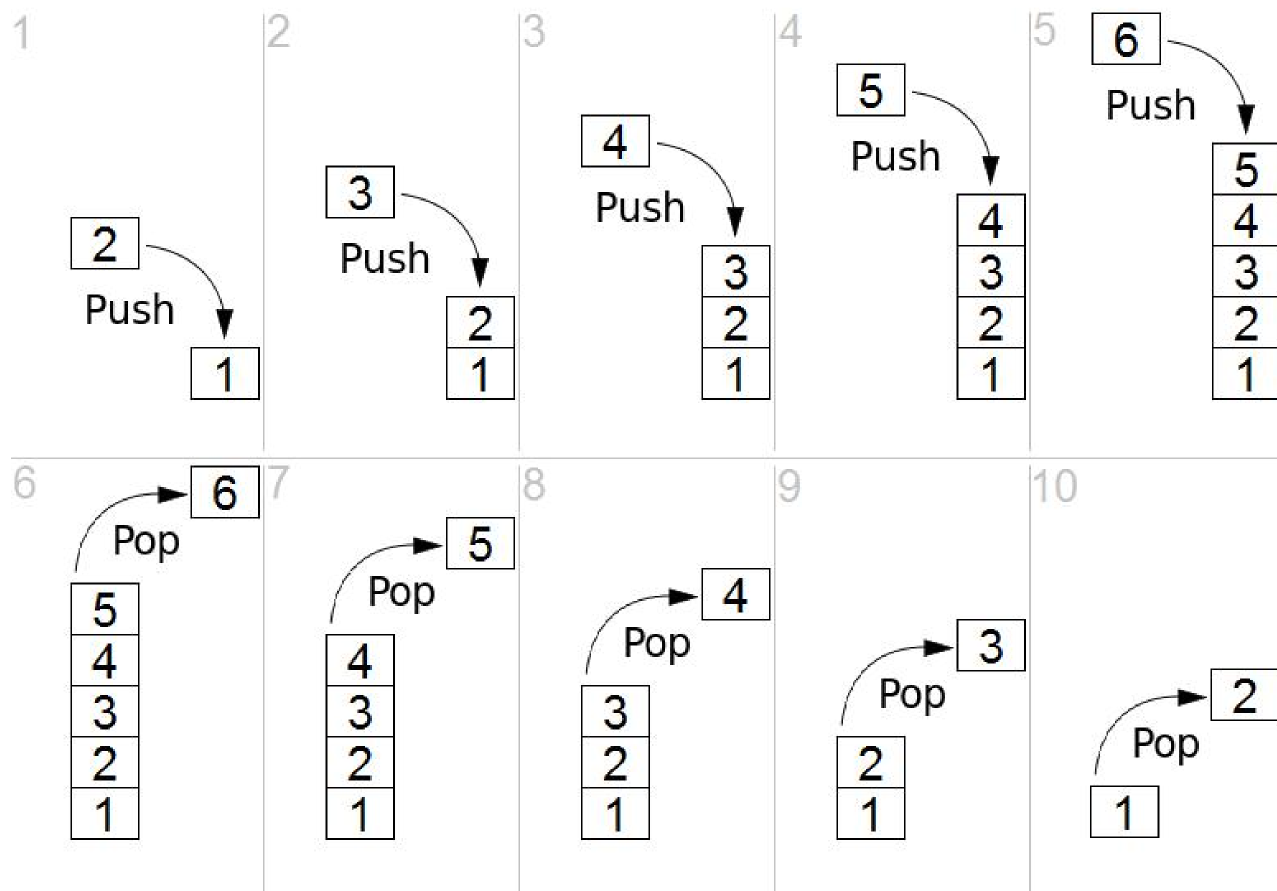
Pile

Specifica dell'Abstract Data Type

- Le pile (stack) sono strutture dati di tipo LIFO (last in first out)
- Implementano una collezione di elementi su cui è possibile invocare due tipologie di operazioni:
  - `push()`: viene aggiunto un elemento alla collezione
  - `pop()`: rimuove l'ultimo elemento inserito e non ancora rimosso

- Le operazioni supportate dalla pila operano unicamente sulla *testa* (o cima) dello stack
- È una struttura dati fondamentale nell'informatica
  - Permette l'implementazione semplice dell'esecuzione di subroutine/funzioni

Funzionamento delle pile



Implementazione tramite array

- Tramite array:
  - Se non si utilizza un array dinamico, la pila ha dimensione massima prefissata
  - L'implementazione deve tenere traccia di qual è l'ultimo elemento inserito nello stack per supportare una corretta esecuzione della procedura `pop()`

structure stack:

  maxsize : integer

  top : integer

  items : array of item

INITIALIZE(S, size):

  S.items ← new empty array of size items

  S.maxsize ← size

  S.top ← 0

Implementazione tramite array

```
PUSH(S, el):
    if S.top = S.maxsize then
        return false
    else
        S.items[S.top] ← el
        S.top ← S.top + 1
        return true
```

```
POP(S):
    if S.top = 0 then
        return ⊥
    else
        S.top ← S.top - 1
        return S.items[S.top]
```

Implementazione tramite lista

- L'implementazione tramite permette di realizzare una pila senza dimensione massima prefissata
- Le operazioni di inserimento/eliminazione lavorano unicamente sulla testa della lista
- Il costo di tutte le operazioni è $\Theta(1)$

```
structure stack:
    head: Node
    size: integer
```

```
INITIALIZE(S, size ):
    S.head ← ⊥
    S.size ← 0
```

Implementazione tramite lista

```
PUSH(S, el):
    newhead ← new Node
    newhead.data ← el
    newhead.next ← S.head
    S.head ← newhead
    S.size ← S.size + 1
```

```
POP(S):
    if S.head = ⊥ then
        return ⊥
    r ← S.head.data
    S.head ← S.head.next
    S.size ← S.size - 1
    return r
```

Un esempio di applicazione

- Gli stack sono strutture dati fondamentali per la realizzazione di parser e compilatori
- Permettono di implementari efficienti algoritmi di riconoscimento di strutture linguistiche basate su automi e grammatiche
- Un semplice esempio "manuale": riconoscimento di stringhe *parenteticamente corrette*
  - La stringa vuota è parenteticamente corretta
  - Se P_1, P_2, P_3 sono parenteticamente corrette, allora lo è anche $P_1(P_2)P_3$
  - Esempi:
    - `ab(ax)((b)du(mb))` è corretta
    - `a(ax)(c` e `a)b(e` non sono corrette

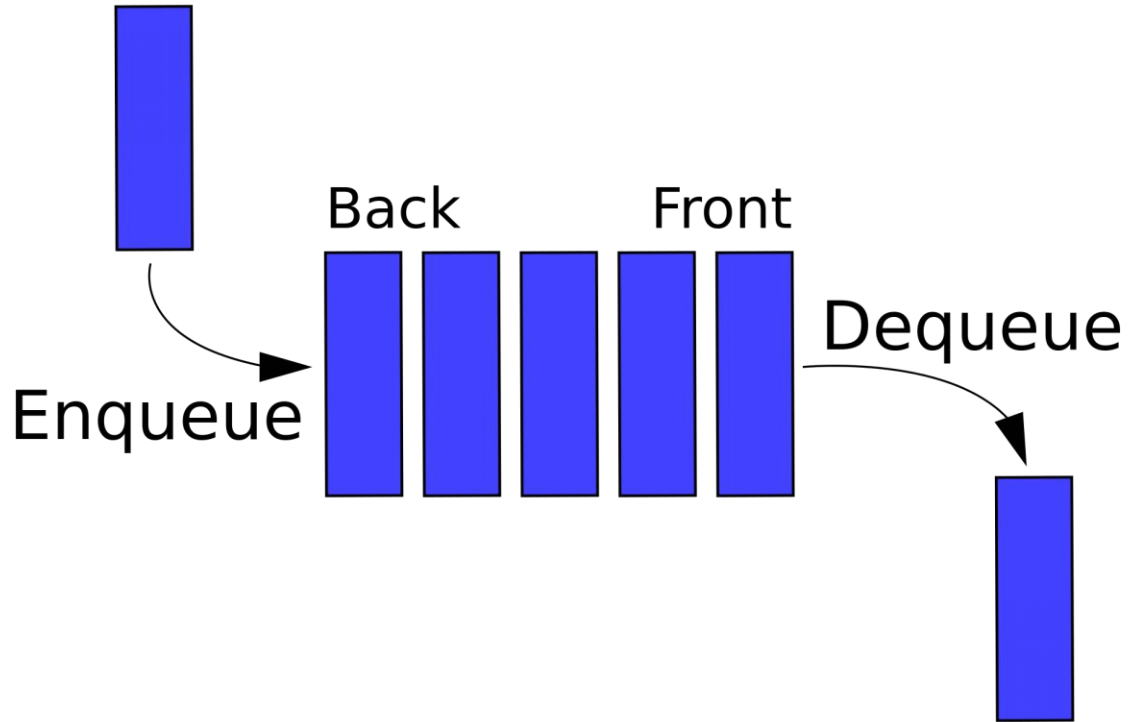
Code

Specifica dell'Abstract Data Type

- Le code (queue) sono strutture dati di tipo FIFO (first in first out)
- Implementano una collezione di elementi su cui è possibile invocare due tipologie di operazioni:
  - `enqueue()`: viene aggiunto un elemento alla collezione
  - `dequeue()`: viene rimosso l'elemento più vecchio inserito e non ancora rimosso

- È una struttura dati fondamentale nell'informatica
  - Tipicamente utilizzata per accodare operazioni in attesa di essere servite (scheduler dei sistemi operativi, instradamento di pacchetti nei router, richieste di client verso server web, …)

Funzionamento delle code



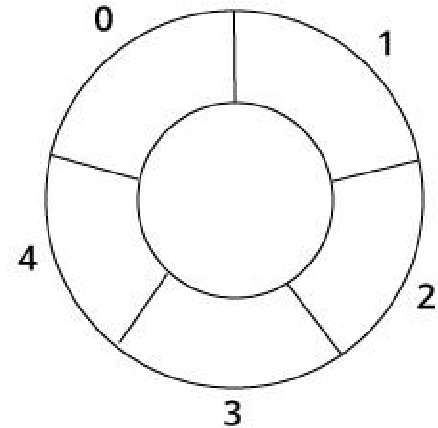
Implementazione tramite lista

- L'implementazione tramite permette di realizzare una coda senza dimensione massima prefissata
  - La Deque è una buona implementazione di base per la realizzazione di una coda
- Le operazioni di inserimento sulla testa della lista, mentre quelle di eliminazione lavorano sulla coda
- Il costo di tutte le operazioni è $\Theta(1)$

Implementazione tramite array circolare

- Si utilizza un vettore di dimensione prefissata
- L'ultimo elemento del vettore è precedente al primo!
- Il primo elemento del vettore è successivo all'ultimo!

- La coda può essere implementata utilizzando dei “cursori”:
  - posizione della prossima lettura
  - posizione della prossima scrittura



Utilizzo dei cursori

- I cursori definiti in questo modo crescono sempre
  - Dobbiamo riportare il valore “virtuale” del cursone al valore “fisico” della cella del vettore

- Per effettuare una lettura:

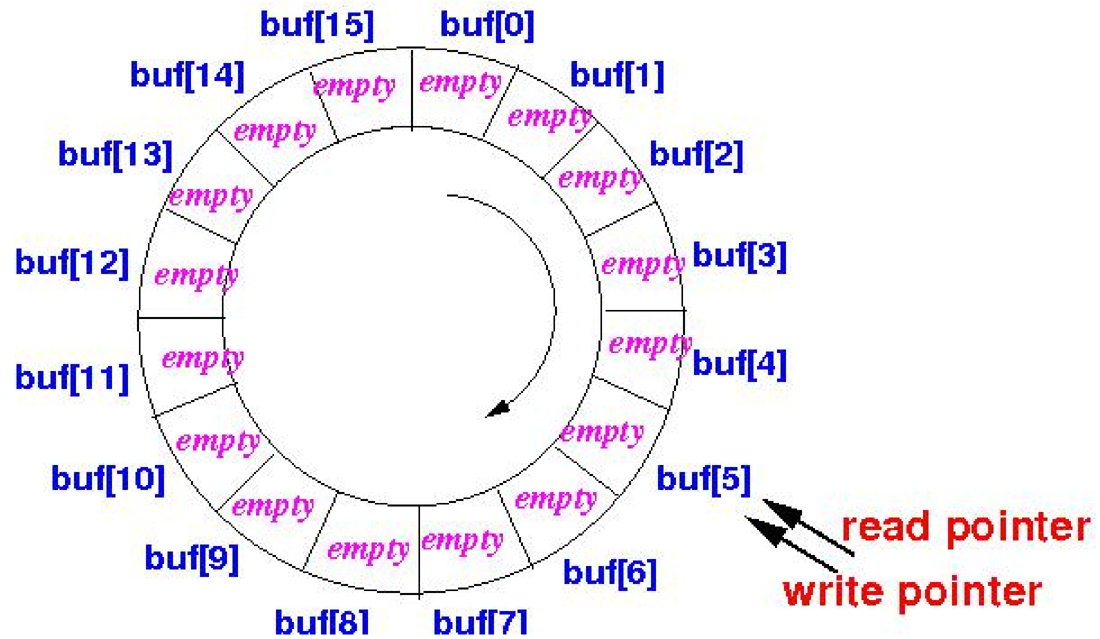
```
DEQUEUE(el):
    ret ← buffer[read]
    read ← (read + 1) % size
    return ret
```

- Per effettuare una scrittura:

```
ENQUEUE(el):
    buffer[write] ← el
    write ← (write + 1) % size
```

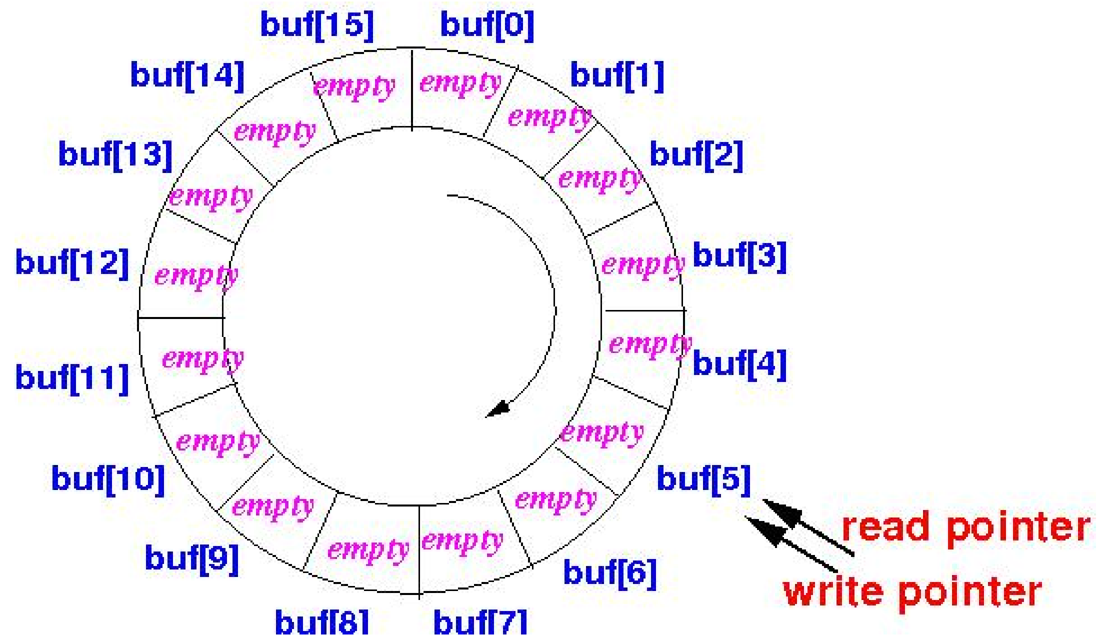
Utilizzo dei cursori

- Quand'è che la coda è vuota?



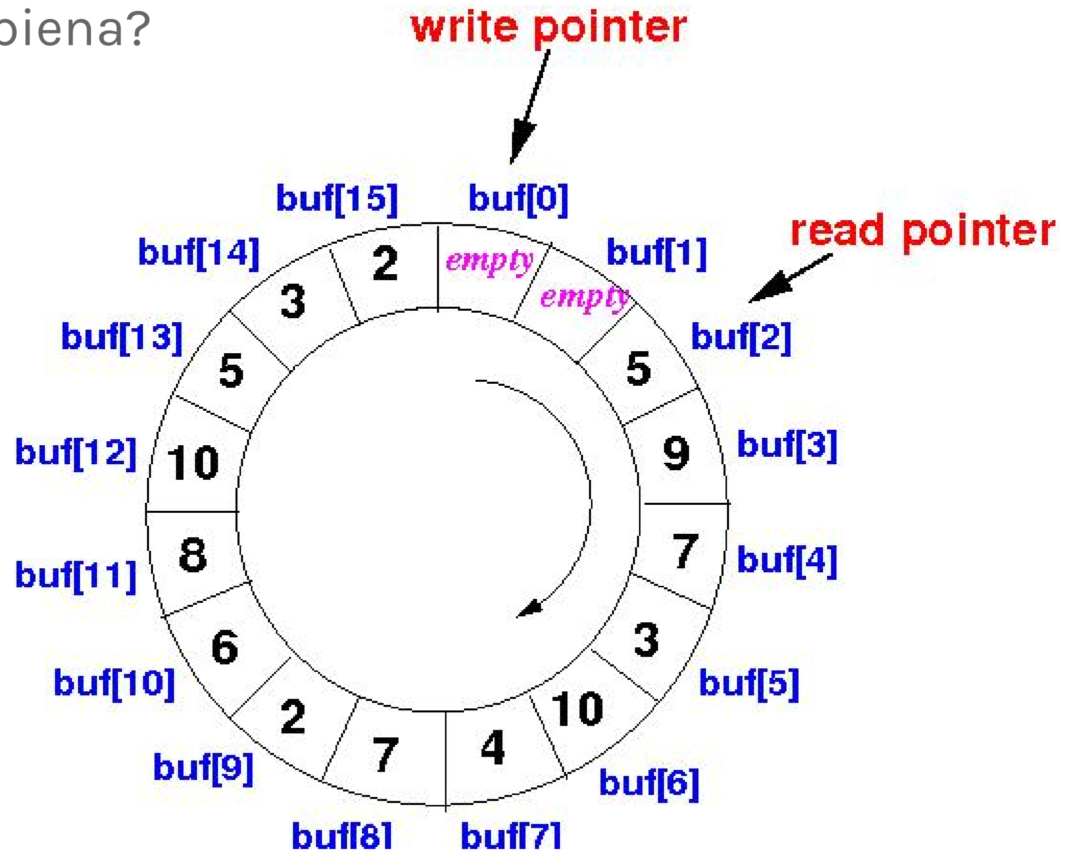
Utilizzo dei cursori

- Quand'è che la coda è vuota?
  - read == write



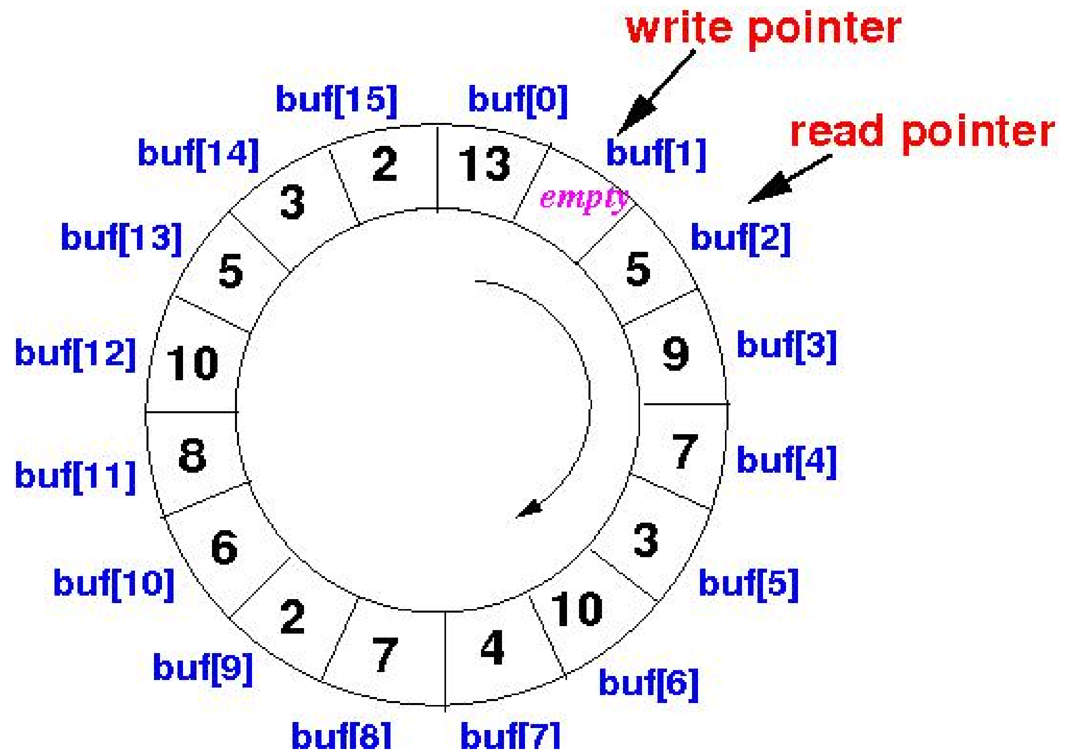
Utilizzo dei cursori

- Quand'è che la coda è piena?



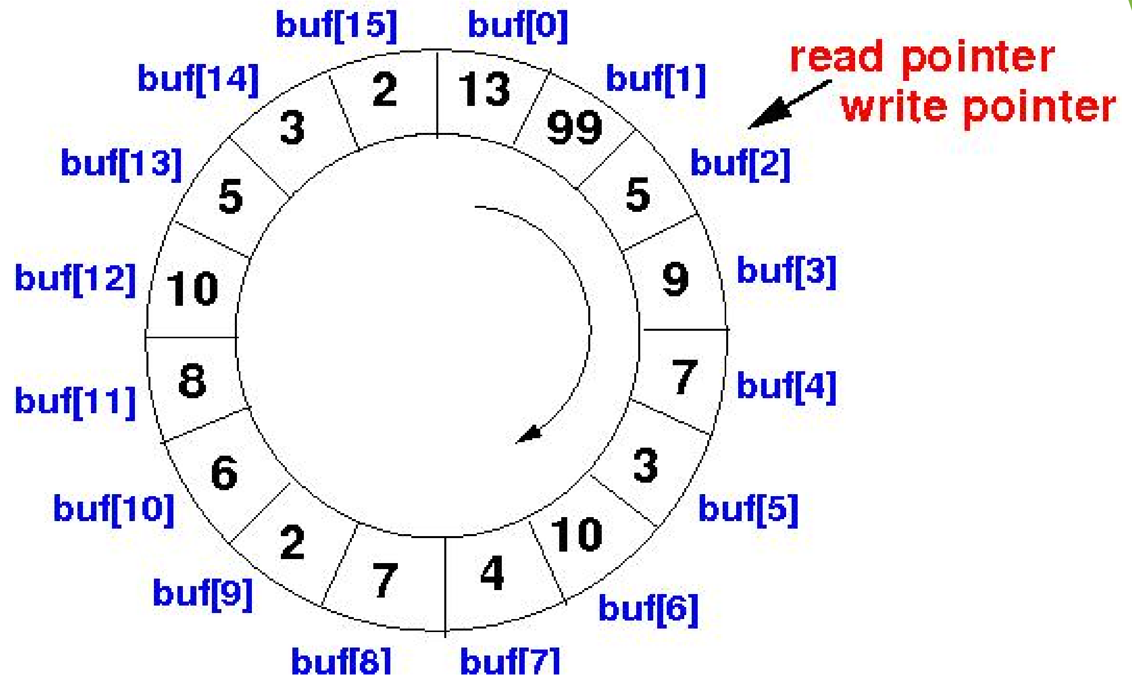
Utilizzo dei cursori

- Quand'è che la coda è piena?



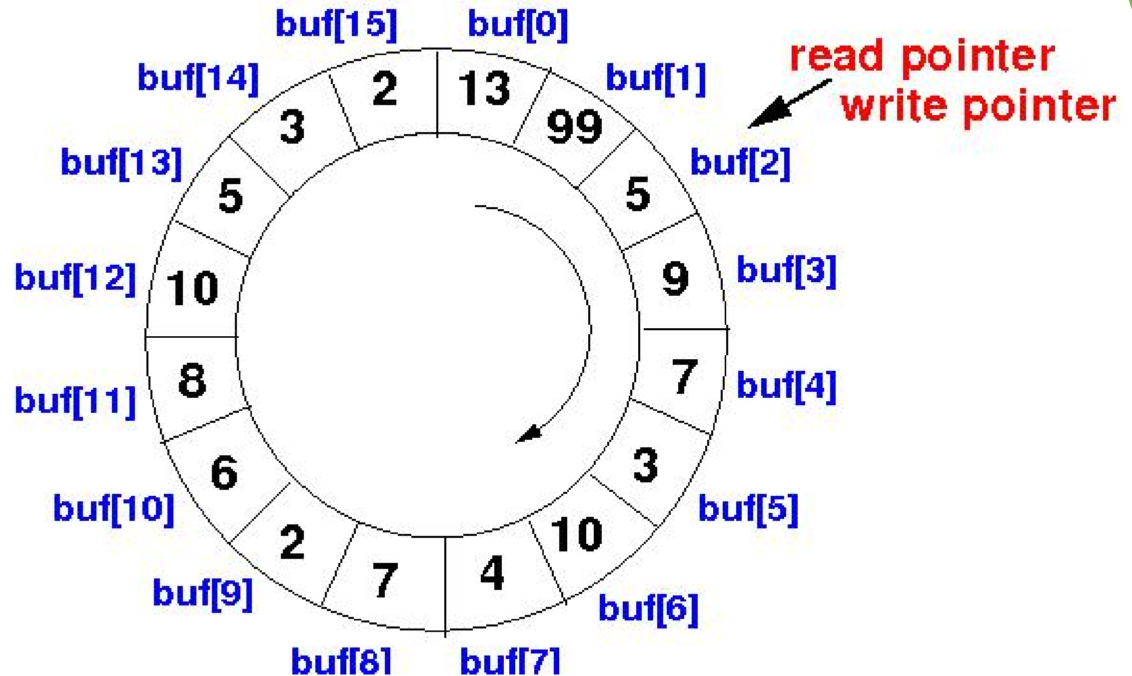
Utilizzo dei cursori

- Quand'è che la coda è piena?



Utilizzo dei cursori

- Quand'è che la coda è piena?
  - write == read



Utilizzo dei cursori

- PROBLEMA!
  - Le condizioni per verificare se la coda è piena o è vuota sono la stessa!

- Tipicamente, questo problema si aggira “sprecando” uno slot e considerando la coda come piena quando resta un solo slot libero.
  - Coda vuota: read == write
  - Coda piena: read == (write + 1) % size

Code di priorità

Specifica dell'Abstract Data Type

- Ciascun elemento è associato ad una *priorità*
- Elementi ad alta priorità vengono estratti dalla coda prima di quelli a bassa priorità
- La priorità massima può essere il valore più grande o più piccolo
- Tipicamente, non è possibile inserire un elemento a priorità minore dell'ultimo elemento estratto
- Operazioni da supportare:
  - `enqueue(prio, el)`: inserisce all'interno della coda l'elemento `el` associato alla priorità `prio`
  - `getMin()`: restituisce l'elemento a priorità più grande attualmente presente nella coda

Calendar Queue (Brown 1988)

- Si basa sul concetto di un calendario da scrivania
  - Si possono inserire gli appuntamenti per ciascun giorno
  - Gli appuntamenti vengono messi in ordine di tempo
- È però un calendario da scrivania "al risparmio"
  - Si utilizza un solo foglio per tutti i mesi
  - In un giorno possono esserci gli appuntamenti di mesi differenti

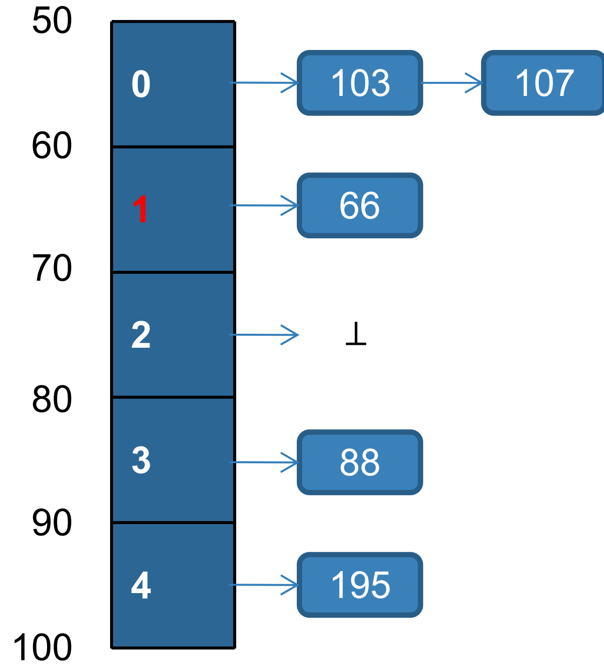
Calendar Queue (Brown 1988)

- L'asse temporale viene suddiviso in *bucket*, ciascuno dei quali ha una certa *grandezza* (o *copertura temporale*) w.
  - Solamente n bucket vengono effettivamente allocati
- Ha la nozione di "ultima priorità estratta" (o *tempo corrente*)
- Se viene richiesto l'inserimento di un nuovo elemento con priorità $p > tempo\ corrente$, questo verrà inserito all'interno del bucket:

$$\left\lfloor \frac{p}{w} \right\rfloor \bmod n$$

- n e w dovrebbero essere scelti in maniera tale da minimizzare il numero di elementi presenti all'interno di ciascun bucket
  - Operazione di *ridimensionamento*: si raddoppia/dimezza n se il numero di elementi per bucket cresce/decresce troppo

Calendar Queue (Brown 1988)



- 5 buckets
- width = 10
- current time = 63

Ridimensionamento del calendario

- Basato su un approccio statistico
  - La grandezza w viene ricalcolata considerando la *separazione media tra gli eventi*
  - Questo approccio funziona bene se, nel futuro prossimo del calendario, gli eventi abbiano già una distribuzione uniforme
  - Per evitare problemi: si escludono dal calcolo gli eventi con separazione troppo grande

- La nuova copertura temporale viene calcolata come $3 \cdot \overline{separazione}$

Analisi Ammortizzata

- Il costo della singola operazione è $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- Il costo totale è $A = \sum_{i=1}^{n} a_i = C + \Phi(D_n) - \Phi(D_0)$
- Se $\Phi(D_n) - \Phi(D_0) \geq 0$, allora il costo della struttura è $O(A)$

- $\Phi(D_0) = 0$
- $\Delta\Phi(D_i) =+ 2$ per le enqueue/dequeue, $\Delta\Phi(D_i) =+ 2 - n$ nel caso di esecuzione di una resize

- Il costo effettivo è 1 per le operazioni di enqueue e dequeue
- Prendiamo una sequenza di operazioni sfavorevole in cui viene effettuata a tempo molto ravvicinato una coppia di resize

Analisi ammortizzata

Operazione	$\Delta\Phi(D_i)$	$\Phi(D_i)$	Stato della struttura
enqueue()	+2	2	nBucks = 2, top = 4, bot = 0, size = 1
enqueue()	+2	4	nBucks = 2, top = 4, bot = 0, size = 2
enqueue()	+2	6	nBucks = 2, top = 4, bot = 0, size = 3
enqueue()	+2	8	nBucks = 2, top = 4, bot = 0, size = 4
enqueue() + resize()	+2 - 5	5	nBucks = 4, top = 8, bot = 0, size = 5
enqueue()	+2	7	nBucks = 4, top = 8, bot = 0, size = 6
enqueue()	+2	9	nBucks = 4, top = 8, bot = 0, size = 7
enqueue()	+2	11	nBucks = 4, top = 8, bot = 0, size = 8
enqueue() + resize()	+2 - 9	4	nBucks = 8, top = 16, bot = 2, size = 9

Prova empirica