Corso di Introduzione agli algoritmi
Prof.ssa Tiziana Calamoneri

Strutture dati fondamentali: code e pile

Code (1)

La ***coda*** è una struttura dati che esibisce un comportamento ***FIFO*** (***First In First Out***). In altre parole, la coda ha la proprietà che gli elementi vengono da essa prelevati esattamente nello stesso ordine col quale vi sono stati inseriti.

La coda può essere visualizzata come una coda di persone in attesa ad uno sportello ed uno dei suoi più classici utilizzi è la gestione della coda di stampa, in cui documenti mandati in stampa prima vengono stampati prima.

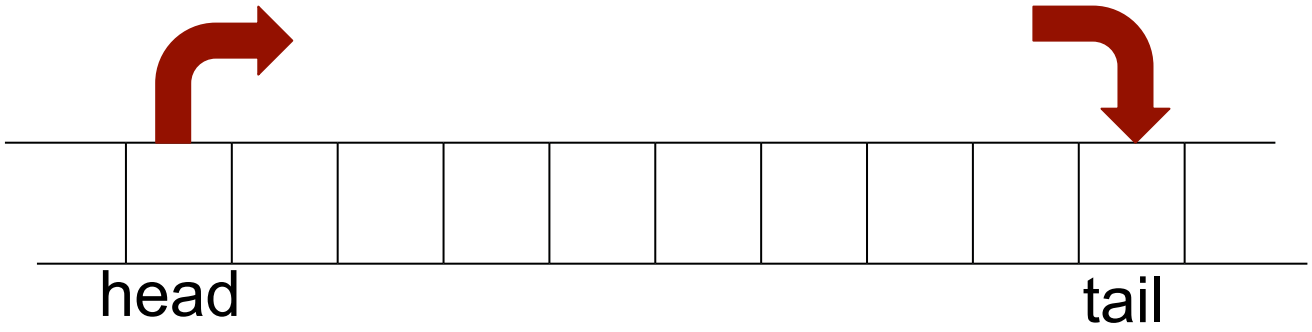
Code (2)

Su una coda sono definite solo due operazioni:

- l'inserimento (che di norma viene chiamata *Enqueue*)
- l'estrazione (che di norma viene chiamata *Dequeue*).

Di solito non si scandiscono gli elementi di una coda né si eliminano elementi con mezzi diversi dalla Dequeue.

La particolarità che garantisce la proprietà FIFO è che l'operazione Enqueue opera su una estremità della coda (***tail***) e la Dequeue opera sull'altra estremità (***head***)

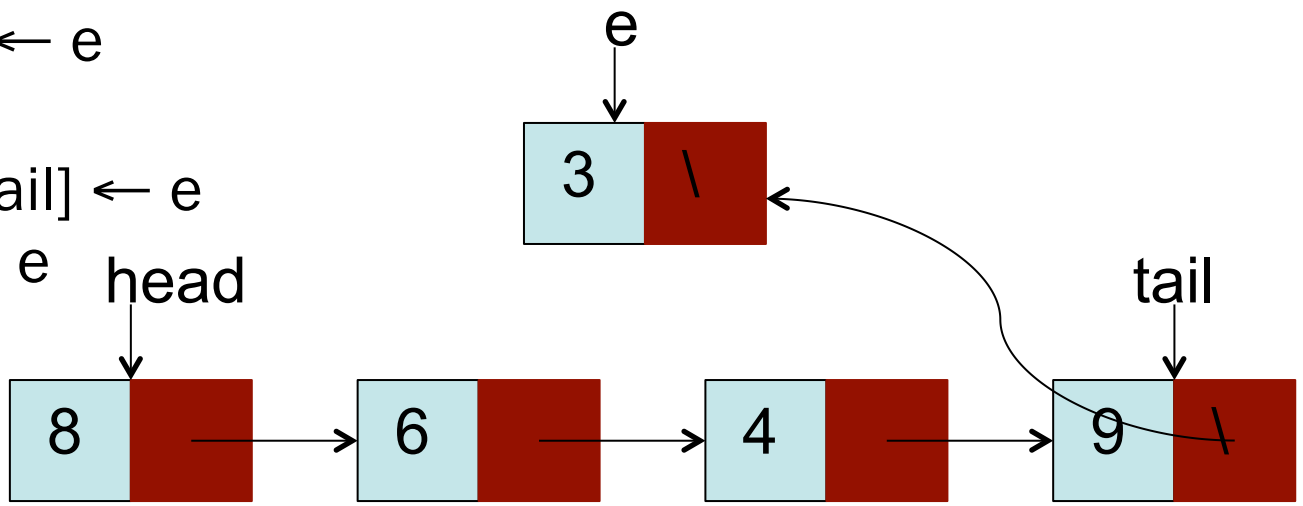


Code (3)

Coda implementata con le liste:

Funzione Enqueue (head: puntatore; tail: puntatore;
e: puntatore all'elemento da inserire)

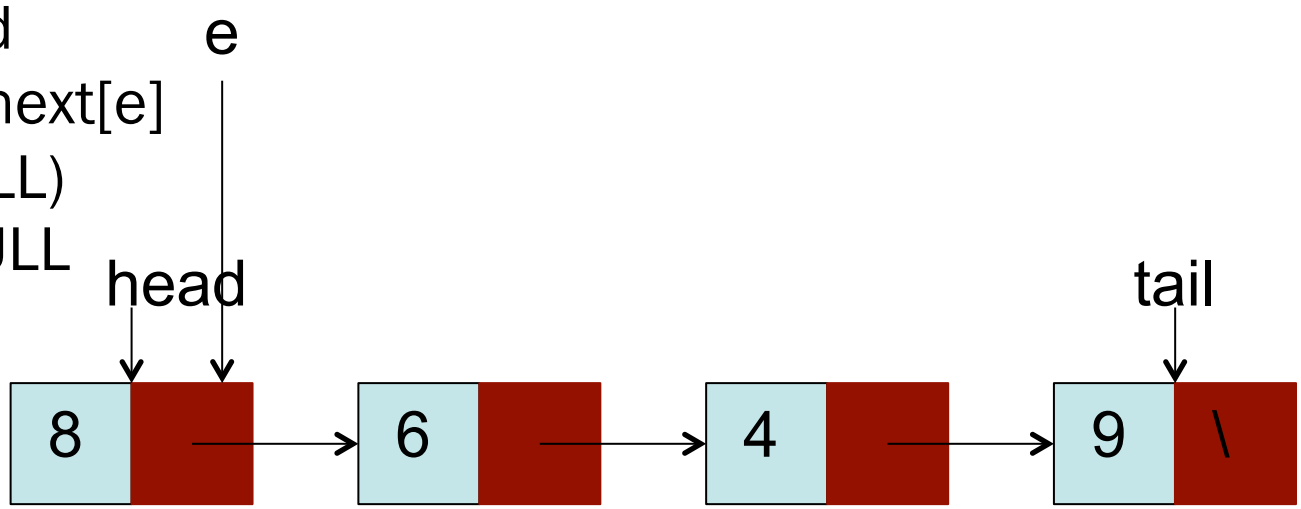
```
if (tail = NULL) //la coda è vuota
        tail ← e
        head ← e
else
        next[tail] ← e
        tail ← e
return
```



Code (4)

Coda implementata con le liste (segue):

```
Funzione Dequeue (head: puntatore; tail: puntatore)
        if (head = NULL) //la coda è vuota
        scrivi "Errore: coda vuota" e return NULL
    else
        e ← head
        head ← next[e]
    if (head = NULL)
        tail ← NULL
    return e
```

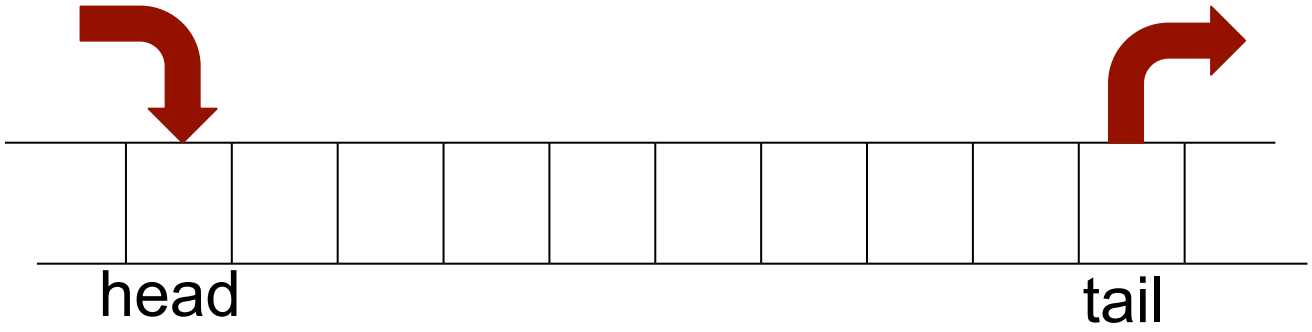


Code (5)

Coda implementata con le liste (segue):

Il costo computazionale di entrambe le operazioni *Enqueue* e *Dequeue* è $\Theta(1)$.

Si noti che se decidessimo di estrarre dalla coda ed inserire in testa, sulla struttura dati così come mostrata (senza campo *prec)* l'operazione di *Dequeue* richiederebbe costo $\Theta(n)$.



Code (6)

Coda implementata con i vettori:

Le code possono essere realizzate mediante vettori se il numero massimo di elementi è noto a priori.

Problema: a seguito di ripetute operazioni di Enqueue e Dequeue, gli elementi della coda si spostano via via verso una delle due estremità del vettore, e quando la raggiungono non vi è apparentemente più spazio per i successivi inserimenti.

Soluzione: gestire il vettore in modo circolare, considerando cioè il primo elemento come successore dell'ultimo.

Code con priorità (1)

- La *coda con priorità* è una variante della coda.
- Come nella coda, l'inserimento avviene ad un'estremità e l'estrazione avviene all'estremità opposta.
- A differenza della coda, la posizione di ciascun elemento non dipende dal momento in cui è stato inserito, ma dal valore di una determinata grandezza, detta *priorità*, la quale in generale è associata ad uno dei campi presenti nell'elemento stesso.
- Quindi, gli elementi di una coda con priorità sono collocati in ordine crescente (o decrescente, a seconda dei casi) rispetto alla grandezza considerata come priorità.

Code con priorità (2)

Esempi:

- La priorità è associata al valore numerico del campo *key*, quando un nuovo elemento avente *key* = *x* viene inserito in una coda con priorità (crescente) esso viene collocato come predecessore del primo elemento presente in coda che abbia *key* $\geq$ *x*.
- In questo senso, un vettore ordinato è una coda con priorità, e la priorità coincide con la chiave.
- Anche la struttura dati heap è una coda con priorità rispetto alla stessa priorità.
- Anche una coda può essere intesa come una coda con priorità, ma qui la priorità è il maggior tempo di permanenza nella struttura dati.
- La pila (che vedremo tra poco) è una coda con priorità dettata dal minor tempo di permanenza nella struttura.

Code con priorità (3)

La coda con priorità presenta un potenziale pericolo di ***starvation*** (***attesa illimitata***): un elemento potrebbe non venire mai estratto, se viene continuamente scavalcato da altri elementi di priorità maggiore che vengono via via immessi nella struttura dati.

Pile (1)

La ***pila*** è una struttura dati che esibisce un comportamento ***LIFO*** (***Last In First Out***). In altre parole, la pila ha la proprietà che gli elementi vengono prelevati dalla pila nell'ordine inverso rispetto a quello col quale vi sono stati inseriti.

La pila può essere visualizzata come una pila di piatti: ne aggiungiamo uno appoggiandolo sopra quello in cima alla pila, e quando dobbiamo prenderne uno preleviamo quello più in alto.

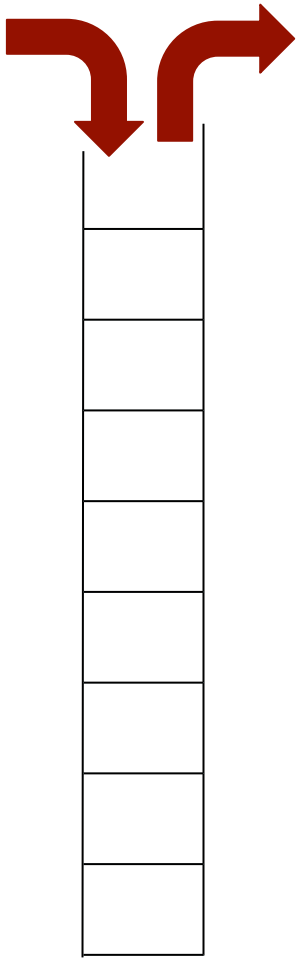
Un esempio di utilizzo di questa struttura dati è la pila di sistema, con la quale vengono tra l'altro gestite le chiamate a funzione (ricorsive e non).

Pile (2)

Su una pila sono definite solo due operazioni: l'inserimento (che di norma viene chiamata ***Push***) e l'estrazione (che di norma viene chiamata ***Pop***).

Non è previsto né scandire gli elementi di una pila né eliminare elementi con mezzi diversi dalla Pop.

La particolarità che garantisce la proprietà LIFO è che le operazioni Push e Pop operano sulla stessa estremità della pila (attraverso il puntatore ***top***).



Pile (3)

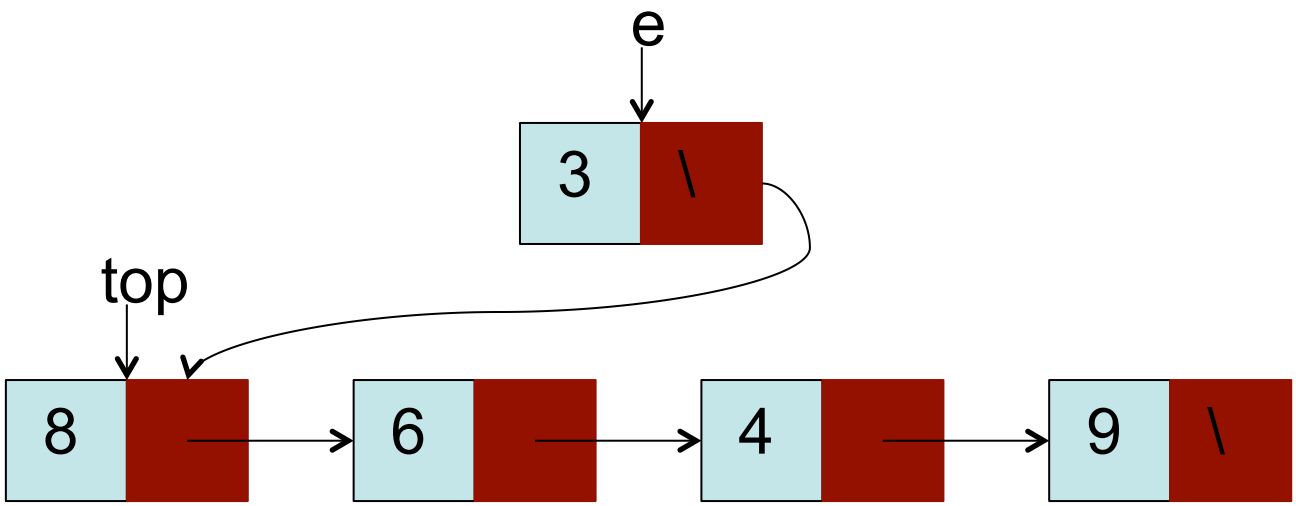
Pila implementata con le liste:

Funzione Push(top: puntatore; e: puntatore all'elem. da inserire)

next[e]← top

top← e

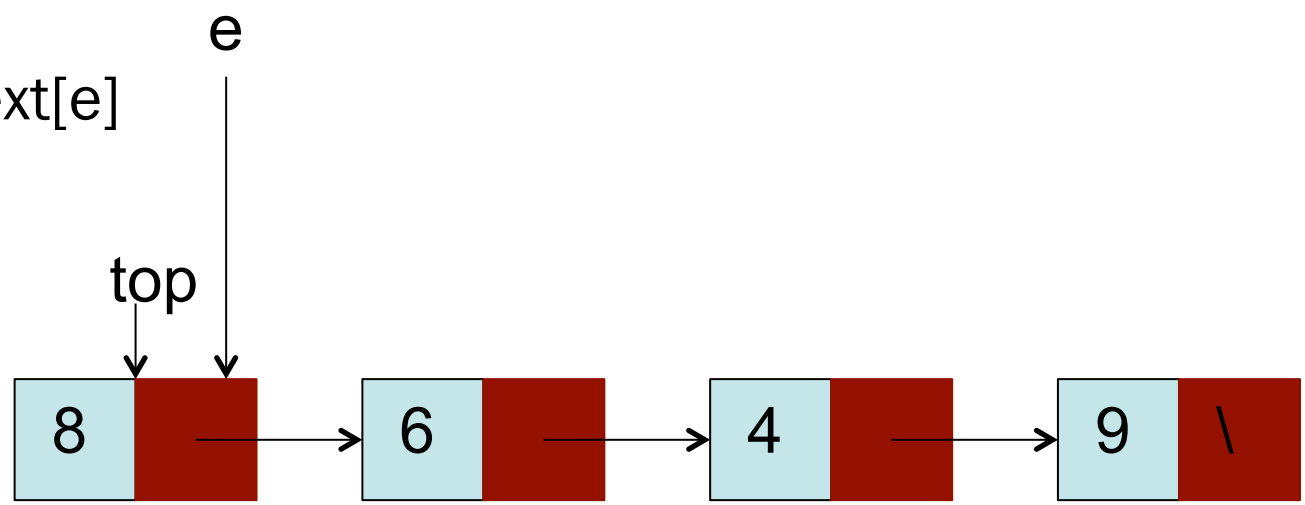
return top



Pile (4)

Pila implementata con le liste (segue):

```
Funzione Pop (top: puntatore)
        if (top= NULL) //la coda è vuota
        scrivi “Errore: coda vuota” e return NULL
    else
        e ← top
        top ← next[e]
    return e
```



Pile (5)

Il costo computazionale di entrambe le operazioni, *Push* e *Pop*, è $\Theta(1)$.

Pile e Code (1)

In entrambe le strutture, è possibile verificare se esse sono vuote, implementando le funzioni di PilaVuota e CodaVuota.

Esse prendono come parametro la struttura dati e restituiscono True se essa è vuota e False altrimenti.

Chiaramente, è possibile effettuare un'estrazione (Dequeue o Pop) solo se la struttura relativa non è vuota.

Pile e Code (2)

Si può anche verificare se una delle due strutture dati è piena, tramite le funzioni di PilaPiena e CodaPiena.

Analogamente, esse prendono come parametro la struttura dati e restituiscono True se essa è piena e False altrimenti.

Ha senso effettuare questo controllo SOLO se la pila o coda sono implementate su un vettore. Infatti, come è noto, una lista non potrà mai essere piena.

Esercizi

- Scrivere lo pseudocodice delle funzioni Enqueue e Dequeue quando la coda sia implementata su un vettore.
- Scrivere lo pseudocodice delle funzioni Push e Pop quando la pila sia implementata su un vettore.
- Si vuole simulare il comportamento di una coda utilizzando due pile; in particolare, si hanno a disposizione le funzioni:
  - push(x, i)
  - pop(i)
  - test_di_pila_vuota(i)

  dove i indica su quale pila si sta operando. Si descrivano le operazioni di Dequeue ed Enqueue e si calcoli il loro costo computazionale.