Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Strutture dati fondamentali: Pile e Code

Tiziana Calamoneri

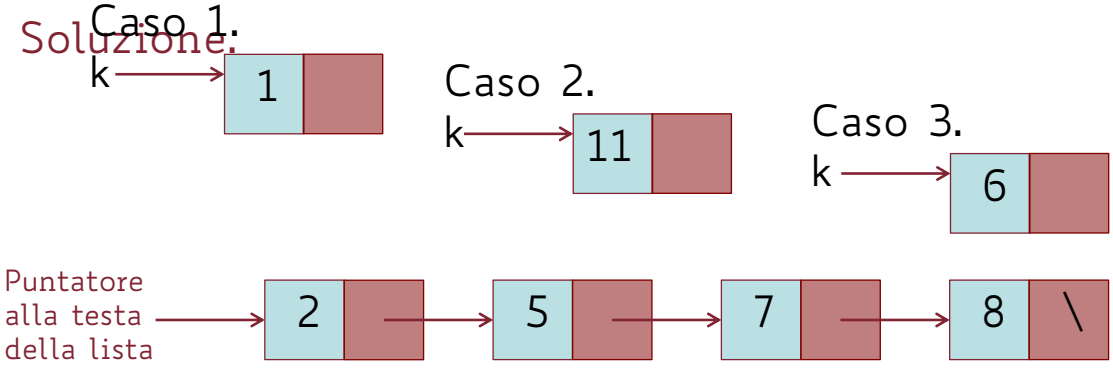
SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio svolto 1 (1)

Esercizio. Data in input una lista ordinata di interi tramite il puntatore al primo elemento, ed un elemento da inserire, aggiungere tale elemento alla lista in modo da rispettare l'ordinamento.

Soluzione:



Esercizio svolto 1 (2)

```
fun inserisci_inOrd(p: punt. a testa, k: punt. ad
elem. da inserire):
if p == None return k;
if k->key ≤ p->key /* 1° elem.
   k->next = p; p = k;
else p' = p; p'' = p'->next;
     while ((p''->key ≤ k->key)AND(p''≠None))
        p' = p''; p'' = p'->next;
         /*k va posizionato tra p' e p''
     if p''=None /*ultimo elem.
       p'->next = k;
     else p'->next = k; k->next = p''
return p
```


Esercizio svolto 1 (3)

Abbiamo già detto che le liste sono strutture dati inerentemente ricorsive. Pertanto, tutti gli algoritmi proposti possono essere implementati sia in versione iterativa che ricorsiva.

Spesso la versione ricorsiva è più compatta, consentendo di non dover distinguere i vari casi (inserimento in testa, in coda o in un punto intermedio della lista).

La funzione di inserimento in una lista ordinata ne è un esempio:

Esercizio svolto 1 (4)

```
fun inserisci_inOrd_Ric(p: punt. alla testa,
k: punt. ad elem. da inserire):
if p == None or k->key ≤ p->key
                /* k sarà la testa della lista
      k->next = p
      return k
p->next = inserisci_inOrd_Ric(p->next,k)
return p.
```


Esercizio svolto 2 (1)

Esercizio. Sia dato un array A di n interi. Si scriva un algoritmo ricorsivo che restituisca una lista puntata semplice contenente gli elementi di A nello stesso ordine.

Soluzione. Notiamo che, per mantenere lo stesso ordine che gli elementi hanno nell'array, è necessario inserire in testa alla lista gli elementi a partire dall'ultimo andando verso il primo.

Esercizio svolto 2 (2)

```
def esercizio2(A, i=0):
    if i== len(A):
        return None
    p = Nodo(A[i])  #allocazione di nuova memoria
    p.next= esercizio2(A, i+1) return p
```

Costo computazionale:

$$T(n) = T(n - 1) + \Theta(1)$$
$$T(0) = \Theta(1)$$

che ha soluzione $\Theta(n)$.

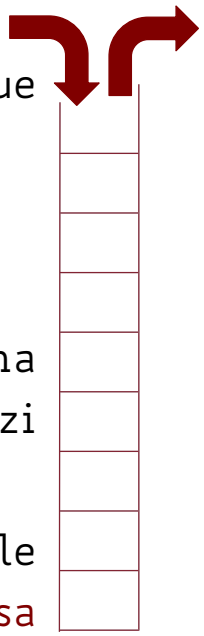
Pile (1)

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

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.

Esempio: la pila di sistema, con la quale vengono, tra l'altro, gestite le chiamate a funzione.

Pile (2)



Su una pila sono definite solo due operazioni:

- inserimento (*Push*)
- estrazione (*Pop*).

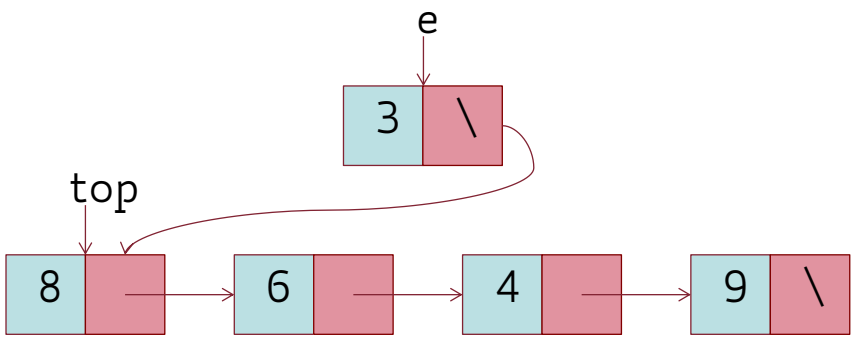
Non è previsto scandire gli elementi di una pila né eliminare elementi con mezzi diversi dalla Pop.

Particolarità della proprietà LIFO: le operazioni Push e Pop operano sulla stessa estremità della pila (attraverso il puntatore *top*).

Pile (3)

Pila implementata con le liste:

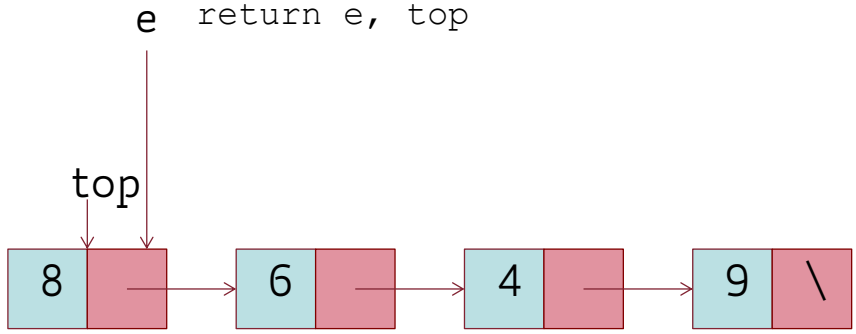
```
fun Push(top: punt.; e: punt. all'elem. da inserire)
            e->next= top
            top = e
            return top
```



Pile (4)

Pila implementata con le liste (segue):

```
fun Pop (top: puntatore)
      if (top==None) //la coda è vuota
            scrivi "Errore: coda vuota" e return None
      e = top
      top = e->next
      e->next=None
      return e, top
```



Pile (5)

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

Pile (6)

Pila implementata con gli arrays:

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

Problema: dobbiamo mantenere il costo delle funzioni Push e Pop costante, per cui non va bene inserire e cancellare in prima posizione.

Soluzione: il top della pila dovrà essere posizionato in corrispondenza dell'elemento più a destra, tramite un opportuno indice (non un puntatore!) all'ultima posizione dell'array occupata

Code (1)

La *coda* è una struttura dati con 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.

Esempio: la coda di stampa, in cui documenti mandati in stampa prima vengono stampati prima.

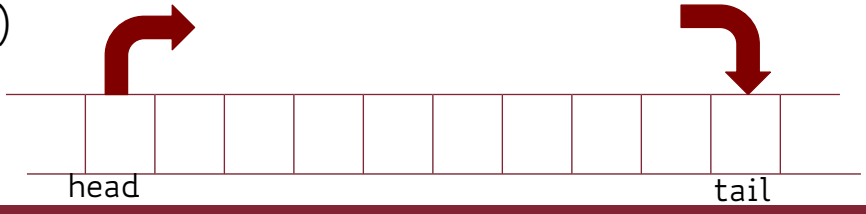
Code (2)

Su una coda sono definite solo due operazioni:

- inserimento (*Enqueue*)
- estrazione (*Dequeue*).

Non è prevista la scansione degli elementi né l'eliminazione con mezzi diversi dalla Dequeue.

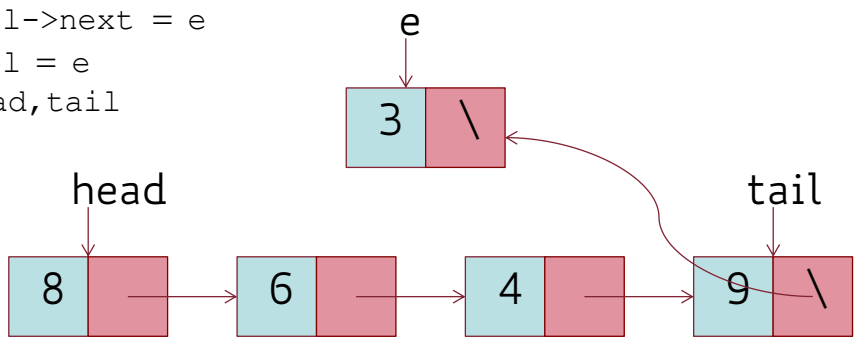
Per la proprietà FIFO, Enqueue opera su una estremità della coda (*tail*) e Dequeue sull'altra (*head*)



Code (3)

Coda implementata con le liste:

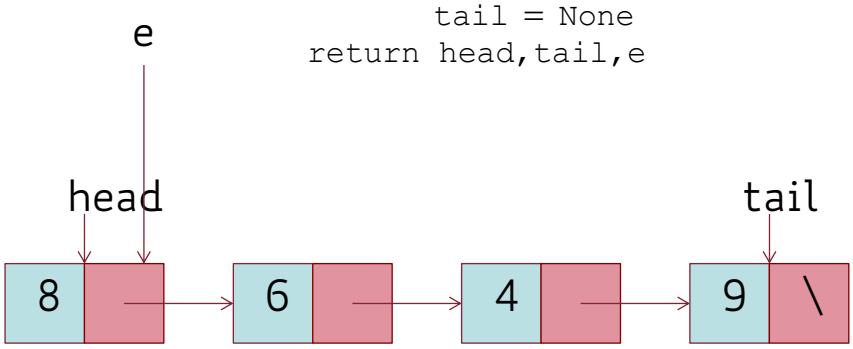
```
Funzione Enqueue (head: puntatore; tail: puntatore;
                          e: punt. all'elem. da inserire)
      if (tail == None) //la coda è vuota
            tail = e
            head = e
      else
            tail->next = e
            tail = e
      return head,tail
```



Code (4)

Coda implementata con le liste (segue):

```
Funzione Dequeue (head: puntatore; tail: puntatore)
        if (head == None) //la coda è vuota
                scrivi "Errore: coda vuota" e return None
        e = head
        head = e->next
        e->next = None
        if (head == None)
                tail = None
        return head,tail,e
```

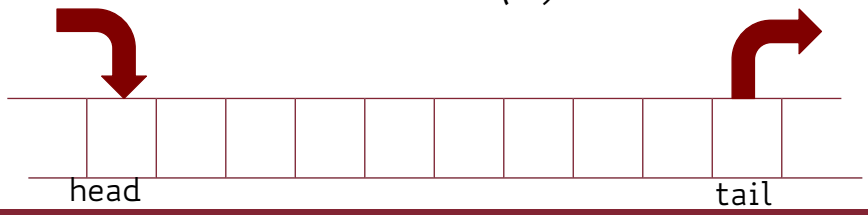


Code (5)

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 gli arrays:

Le code possono essere realizzate mediante arrays 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 dell'array e, quando la raggiungono, sembra non esserci più spazio per i successivi inserimenti.

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

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 sia 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 la coda sono implementate su un array. Infatti, come è noto, una lista non potrà mai essere piena.

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 dipende dal valore di una determinata grandezza (*priorità*).
- Gli elementi di una coda con priorità sono collocati in ordine rispetto alla grandezza considerata come priorità.

Code con priorità (2)

Esempi:

- La priorità è associata al valore numerico del campo *key*.
  In questo senso, un array 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.
- …

Code con priorità (3)

Esempi (segue):

- …
- Una coda può essere intesa come una coda con priorità, in cui la priorità è il maggior tempo di permanenza nella struttura dati.
- La pila è una coda con priorità, in cui la priorità è il minor tempo di permanenza nella struttura.

Code con priorità (4)

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.

Esercizio svolto (1)

Esercizio. Si vuole simulare una coda tramite due pile. In particolare, si possono usare – senza dettagliarle– le seguenti funzioni:

- push(x,i)
- pop(i)
- test_di_pila_vuota(i)

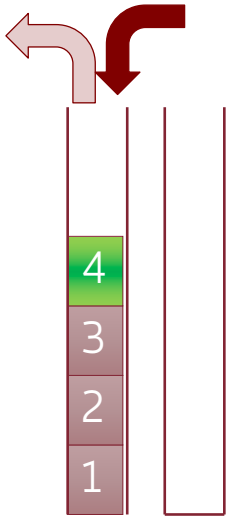
dove i è l'indice della pila sulla quale si eseguono le operazioni.

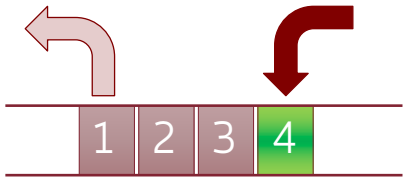
Descrivere le operazioni di Dequeue ed Enqueue con le 2 pile.

Esercizio svolto (2)

Soluzione. Utilizziamo una pila per memorizzare i dati ed una pila come appoggio.

Abbiamo molti modi di procedere; qui scegliamo quello in cui Enqueue coincide con Push:



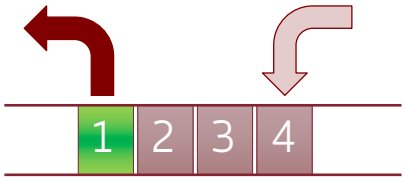



```
Enqueue(u)
        Push(u,1)
        return
```

Costo: $\Theta(1)$

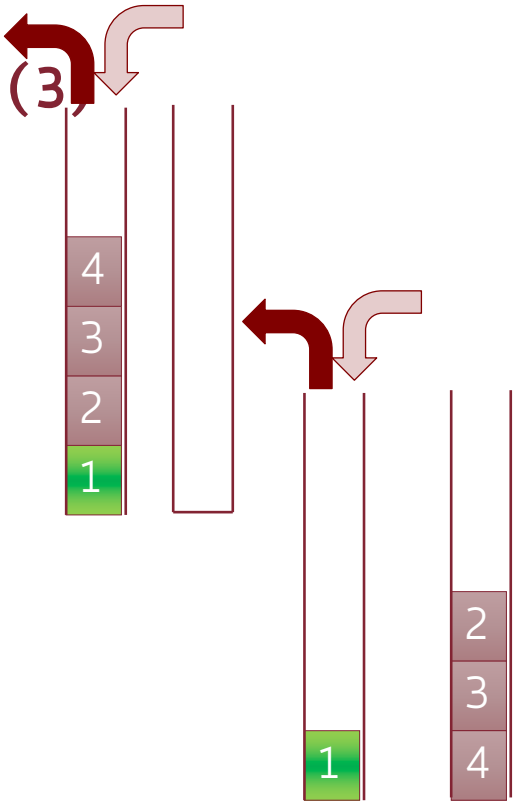
Esercizio svolto (3)

Per la Dequeue:



Dobbiamo rovesciare la pila 1 per estrarre l'ultimo elemento

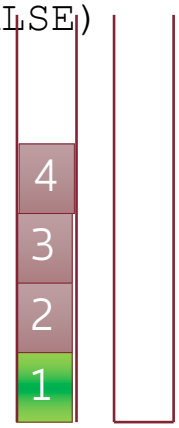
e poi riportare gli elementi indietro...



Esercizio svolto (4)

```
Dequeue()
      while (test_di_pila_vuota(1)=FALSE)
            Push(Pop(1),2)
      aux←pop(2)
      while (test_di_pila_vuota(2)=FALSE)
            Push(Pop(2),1)
      return aux
```

Costo: $\Theta(n)$



Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi

- Scrivere lo pseudocodice delle funzioni Enqueue e Dequeue quando la coda sia implementata su un array.
- Scrivere lo pseudocodice delle funzioni Push e Pop quando la pila sia implementata su un array.