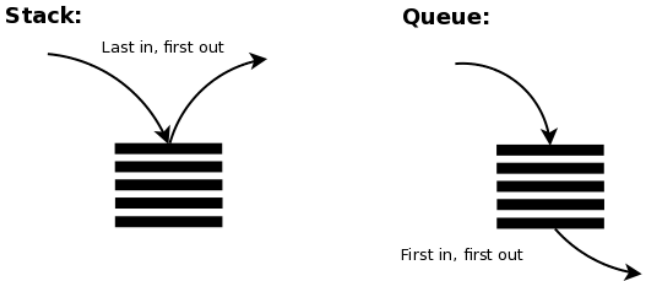
Strutture Dati Lineari

corso di laurea in **Matematica**

Informatica Generale, Lezione **16**

Ivano Salvo



SAPIENZA
UNIVERSITÀ DI ROMA

Code (Queue)

corso di laurea in **Matematica**

Informatica Generale, Lezione **16(a[1])**

Ivano Salvo

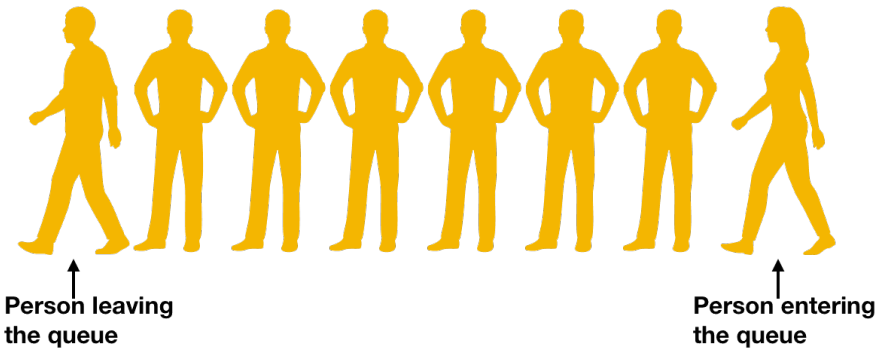
SAPIENZA
UNIVERSITÀ DI ROMA

Code (Queue)

Le code sono un tipo di dato utile a disciplinare l'accesso a risorse seguendo una disciplina **FIFO** (**F**irst **I**n, **F**irst **O**ut).

Le operazioni che ci aspettiamo dal **tipo di dato coda** sono:

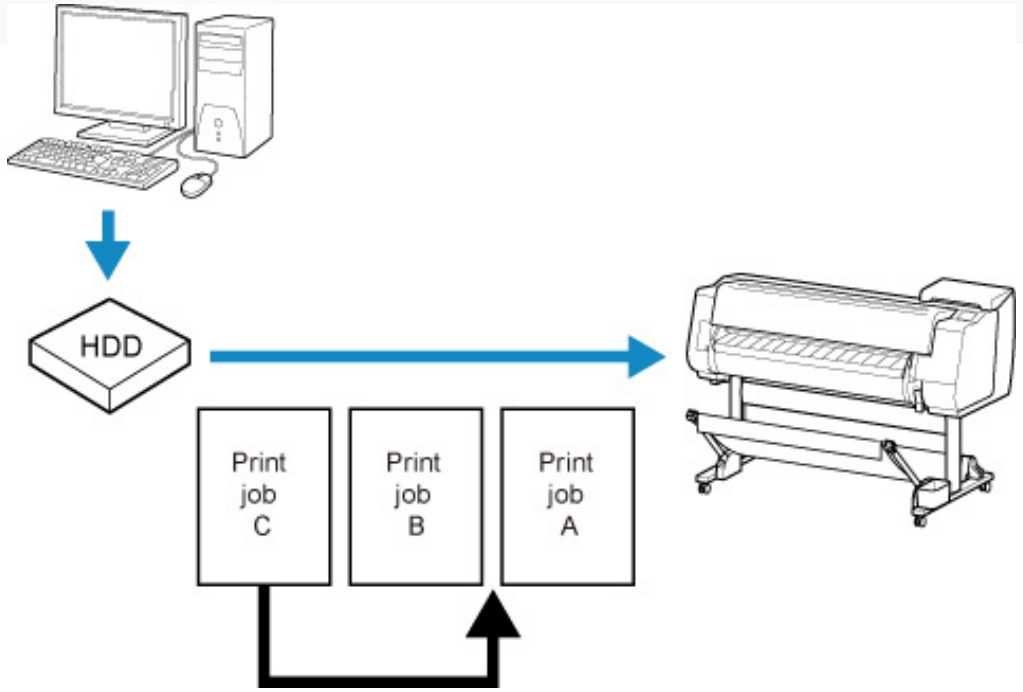
- `Q = newQueue([n])`: **crea** una nuova lista vuota [con *n* posti, **opzionale**].
- `enqueue(Q, x)`: **aggiunge** l'elemento *x* alla coda
- `y = dequeue(Q)`: **estrae** un elemento dalla coda (quello inserito per primo tra quelli memorizzati in Q) e ne **restituisce** il valore
- `b = isFull(Q)`: restituisce **TRUE se la coda è piena** (dà sempre FALSE se non è previsto un numero massimo di elementi nella coda)
- `b = isEmpty(Q)`: restituisce **TRUE se la coda è vuota (**non contiene nessun elemento).



Esempi di uso delle Code

Le code hanno numerose applicazioni in Informatica, ad esempio:

- processi in attesa dell'uso di una risorsa condivisa (immaginate le richieste dell'uso di una stampante o del processore);
- pacchetti che vengono instradati in una rete vengono messi in coda per ulteriori movimenti.
- le useremo più volte come struttura dati ausiliaria nella visita di alberi e grafi.



Implementazione delle Code

Le code possono essere implementate sia con **vettori** che con **liste** semplici. In entrambi i casi si possono implementare le funzioni `enqueue(Q, x)` e `dequeue(Q)` in **tempo $\theta(1)$**, con alcune differenze:

- **con Liste**: potendo le liste contenere un numero arbitrario di elementi, **non è nececessario prevedere un numero massimo** di elementi. Finché c'è memoria, c'è speranza.

Dovendo fare inserimenti e rimozioni da entrambe le estremità di una lista, risultano comode le **liste con pointer all'ultimo elemento**.

Tuttavia, **rimuovere l'ultimo elemento** è "**difficile**" perché non c'è modo di riaggiornare convenientemente il puntatore last, non potendo andare all'indietro…

Idea fondamentale: fare rimozioni in testa e inserimenti in coda.

- **con Vettori**: richiedono di definire una **lunghezza massima** all'atto della **creazione della coda**.

Idea fondamentale: vedere il vettore come "**circolare**" e mantenere gli indici del primo e ultimo elemento occupato.

Implementazione Code con liste

Da un punto di vista **concreto**, una coda è semplicemente una **lista con pointer all'ultimo elemento**.

Tuttavia, sono **ammesse solo alcune operazioni**, in modo da garantire la corretta implementazione della **disciplina FIFO** prevista dal **tipo di dato astratto coda**.

Qui omettiamo il numero di elementi e la verifica di coda piena.

```
def newQueue():
    L = creaListaFL()
    return L
```

```
def isEmpty(q):
    if first[q]=NULL
        return True
    return False
```

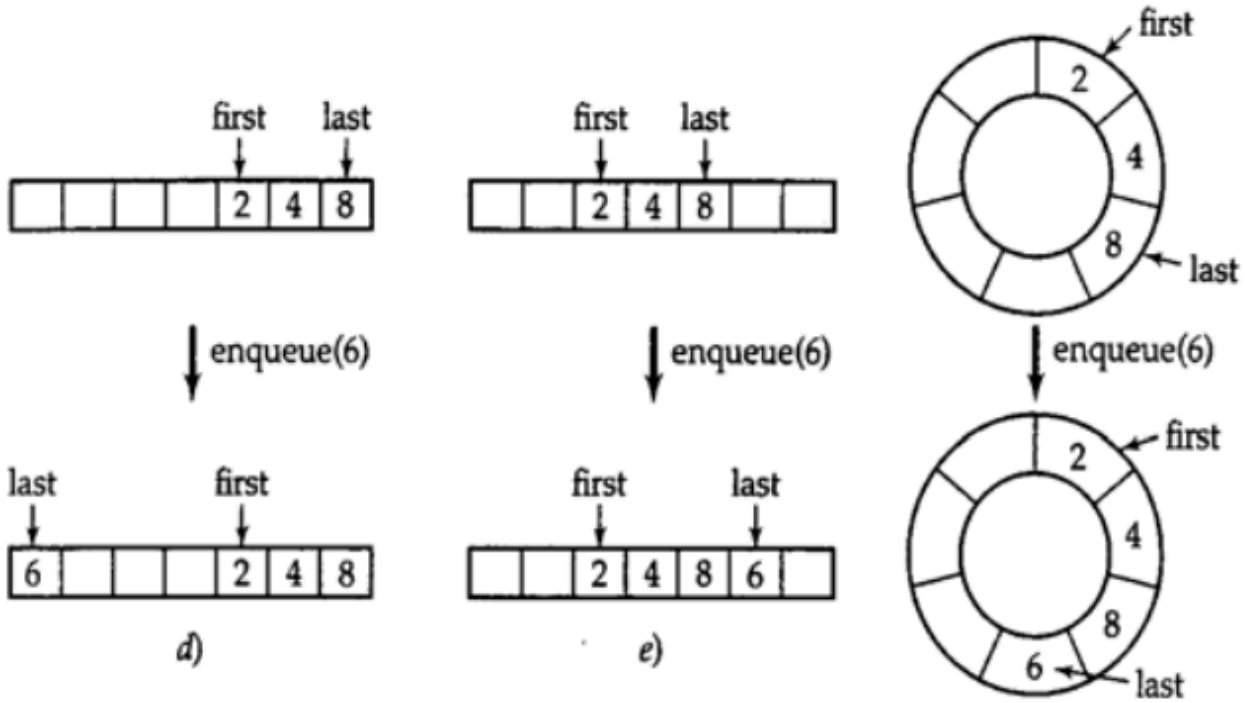
```
def dequeue(q):
# REQ: q non vuota
    tmp = key[last[q]]
    removeFirst[q]
    return tmp
```

```
def enqueue(Q, x):
    addLast(Q, x)
```

Implementazione Code con vettori

Con vettori si usa un array visto come **circolare**: cioè $v[0]$ è il successore di $v[n\text{-}1]$.

Quest'idea permette di fare inserimenti/rimozioni senza dover far scivolare avanti e indietro gli elementi nel vettore.



Code con vettori: codice

L'implementazione con vettori è **più complicata**: occorre prevedere una dimensione massima (**dim**) e tenere due indici per sapere dov'è il primo (**fst**) e l'ultimo (**lst**) elemento della coda.

Infine bisogna sempre sapere quanti elementi sono presenti (**num**).

```
def newQueue(n):
    Q = allocaQ()
     # alloca un record
    Q[v] = alloca(n)
    dim[Q], num[Q] = n, 0
    fst[Q], lst[Q] = 0, 0
```

```
def isEmpty(q):
    return num[Q]==0
```

```
def isFull(q):
    return num[Q]==dim[Q]
```

```
def dequeue(Q):
# REQ: num[Q] > 0
    tmp = v[Q][fst]
    Q[fst] = (Q[fst] + 1) % n
    num[Q] = num[Q] - 1
    return tmp
```

```
def enqueue(Q, x):
# REQ: num[Q] < dim[Q]
    Q[v][last] = x
    Q[last] = Q[last] + 1 % n
    num[Q] = num[Q] + 1
```

Trattamento circolare del vettore

Pile (Stack)

corso di laurea in **Matematica**

Informatica Generale, Lezione **16(a^2)**

Ivano Salvo

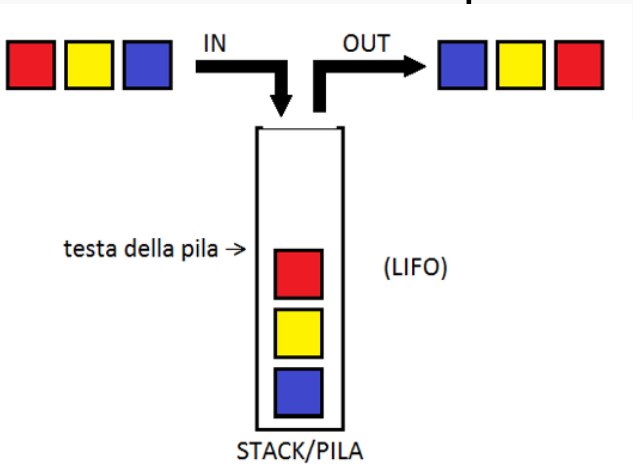
SAPIENZA
UNIVERSITÀ DI ROMA

Pile (Stack)

Le pile sono un tipo di dato utile a disciplinare l'accesso a risorse seguendo una disciplina **LIFO** (**L**irst **I**n, **F**irst **O**ut).

Le operazioni che ci aspettiamo dal **tipo di dato pila** sono:

- `S = newStack([n])`: **crea** una nuova pila **vuota** [con n posti].
- `push(S, x)`: **aggiunge** l'elemento x in **cima** alla pila S.
- `y = top(S)`: restituisce il **valore** dell'elemento **in cima** alla pila S, ma **senza rimuoverlo**.
- `y = pop(S)`: restituisce il **valore** dell'elemento **in cima** alla pila S, e lo **rimuove** dalla pila.
- `b = isFull(S)`: restituisce TRUE se la pila è **piena** (dà sempre FALSE se non è previsto un numero massimo di elementi nella pila)
- `b = isEmpty(S)`: restituisce TRUE se la pila è **vuota** e non contiene nessun elemento;



Esempi di uso delle Pile

Le pile hanno numerose applicazioni in Informatica, ad esempio:

- riconoscimento di **linguaggi parentesizzati** (esempio: linguaggi di programmazione oppure espressioni aritmetiche): ogni parentesi chiusa chiude l'ultima parentesi aperta.
- **pila di sistema** per gestire le chiamate di funzione. È necessaria per gestire adeguatamente la **ricorsione**.
- le useremo più volte come **struttura dati ausiliaria** nella visita di alberi e grafi (usualmente una **pila è alternativa** all'uso di **algoritmi ricorsivi**).

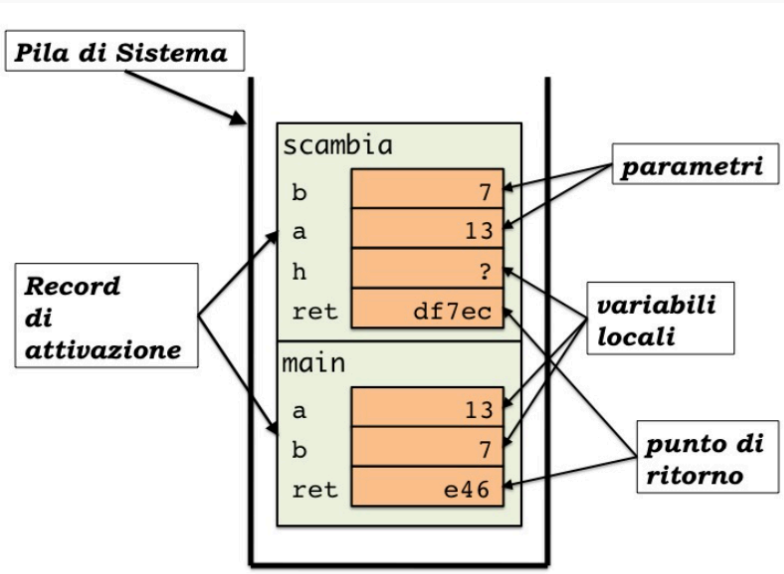


Figura 4: Stack di attivazione delle funzioni

Implementazione delle Pile

Le pile possono essere implementate sia con **vettori** che con **liste** semplici. In entrambi i casi si possono implementare le funzioni `push(S, x)`, `top(S)` e `pop(S)` in **tempo $\theta(1)$**, con alcune differenze:

- **con Liste**: potendo le liste contenere un numero arbitrario di elementi, non è nececessario prevedere un numero massimo di elementi.

L'implementazione è ancora più semplice delle code, perché **inserimenti** e **rimozioni** avvengono **nella stessa estremità**.

- **con Vettori**: richiedono di definire una **lunghezza massima** all'atto della **creazione della pila**.

È sufficiente mantenere un indice `top` alla **prima posizione libera** per l'inserimento.

Implementazione Pile con liste

Da un punto di vista **concreto**, una pila è semplicemente un "**guscio**" per una **lista semplice**.

Tuttavia, sono **ammesse solo alcune operazioni**, in modo da garantire la corretta implementazione della **disciplina LIFO** prevista dal **tipo di dato astratto coda**.

Qui omettiamo il numero di elementi e la verifica di lista piena.

```
def newStack():
    S = allocaS()
    L[S] = NULL
    return L[S]
```

```
def isEmpty(S):
    if L[S] == NULL
        return True
    return False
```

```
def top(S):
# REQ: q non vuota
    return key[L[S]]
```

```
def pop(S):
# REQ: q non vuota
    tmp = key[fst[L[S]]]
    L[S] = removeFirst[L[S]]
    return tmp
```

```
def push(S, x):
    L[S] = cons(L[S], x)
```

Implementazione Liste con vettori

Anche l'implementazione con vettori **è più semplice di quelle delle code**: è sufficiente prevedere una dimensione massima (**dim**) e tenere un solo indice (**top**) per sapere quale sia la prima posizione libera.

Il numero di elementi presenti è sempre **top**.

```
def newStack(n):
    S = allocaS()
     # alloca un record
    v[S] = alloca(n)
    dim[S], top[S] = n, 0
```

```
def isEmpty(S):
    return top[S]==0
```

```
def isFull(S):
    return top[S]==dim[S]
```

```
def pop(S):
# REQ: top[S] > 0
    top[S] = top[S] - 1
    return v[S][top[S]]
```

```
def top(S):
# REQ: num[Q] < dim[Q]
    return v[S][top[S]-1]
```

```
def push(S, x):
# REQ: num[Q] < dim[Q]
    v[s][top[S]] = x
    top[S] = top[S] + 1
```