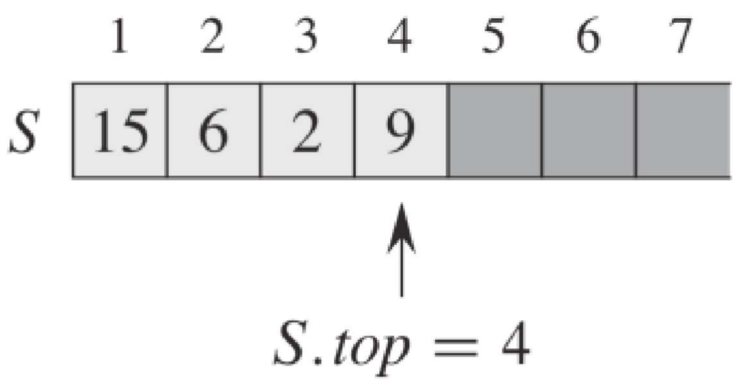
Intro

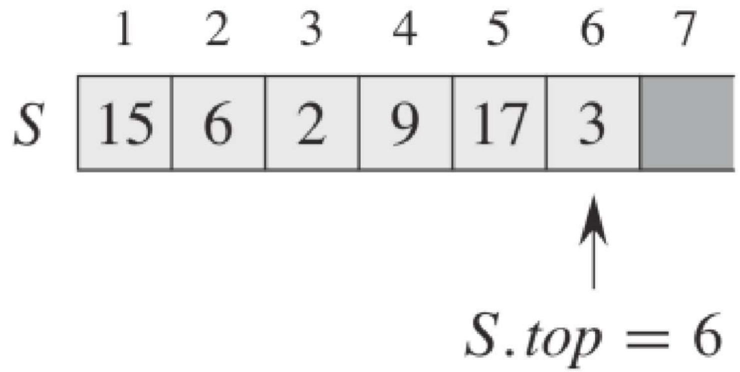
Pile e code sono strutture poco lontane dai semplici vettori, ma che permettono di l'inserimento e l'estrazione degli elementi solo secondo determinate regole, rendendole molto efficienti quando sufficienti a risolvere il problema

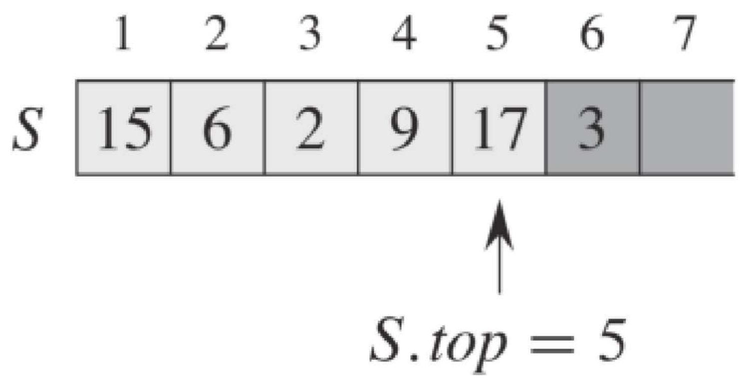
- **Pro**: Tutte le operazioni hanno costo costante
- **Contro**: poca flessibilità

Pile (Stacks)

> **Definizione**
>
> Vettore con politica **LIFO** (Last In First Out)
>
> - **Dati**: insieme di elementi S
> - **Operazioni**: inserimento (push), estrazione (pop)







PUSH(S, x)

```
S.top = S.top + 1
S[S.top] = x
```

Parametri: *S=stack, x=elemento da aggiungere*

POP(S)

```
value = S[S.top]
S.top = S.top - 1
return value
```

Parametri: *S=stack*
Return: *elemento in fondo alla pila (ultimo inserito)*

Costo Operazioni

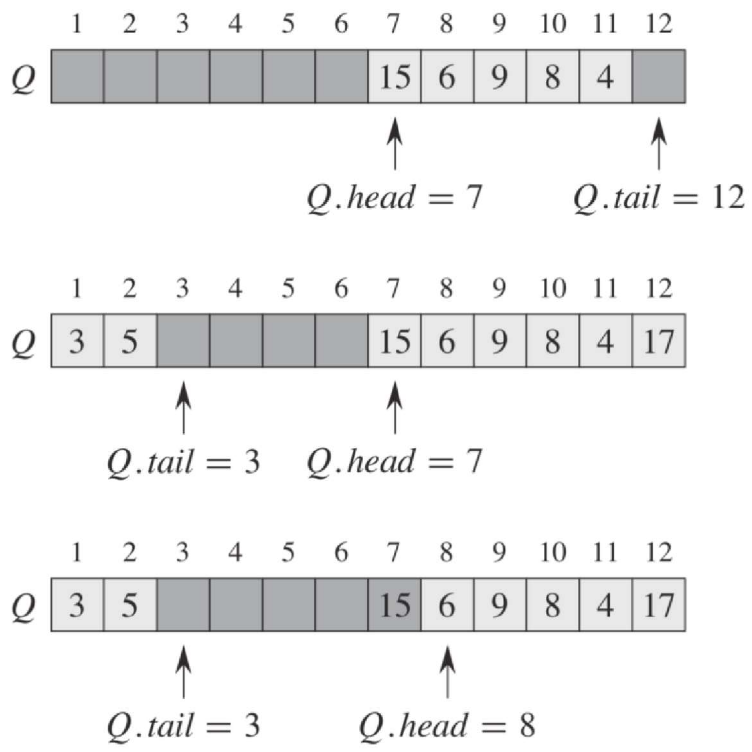
$\Theta(1)$

Code (Queues)

Definizione

Vettore con politica **FIFO** (First In First Out)

- **Dati**: insieme di elementi Q
- **Operazioni**: inserimento (enqueue), estrazione (dequeue)



ENQUEUE(Q, x)

```
Q[Q.tail] = x
if Q.tail == Q.length
        Q.tail = 1
else
        Q.tail = Q.tail + 1
```

Parametri: *Q=queue, x=elemento da aggiungere*

DEQUEUE(Q)

```
x = Q[Q.head]
if Q.head == Q.length
        Q.head = 1
else
        Q.head = Q.head + 1
return x
```

Parametri: *Q=queue*
Return: *elemento in cima alla coda (primo inserito)*

Costo Operazioni

$\Theta(1)$

Puntatori

Definizione

I puntatori sono **indirizzi di memoria** salvati in variabili (come se fossero interi, booleani o qualsiasi altro tipo di dato)

Vengono in genere utilizzati per la creazione di strutture dati **dinamiche e flessibili**

NIL

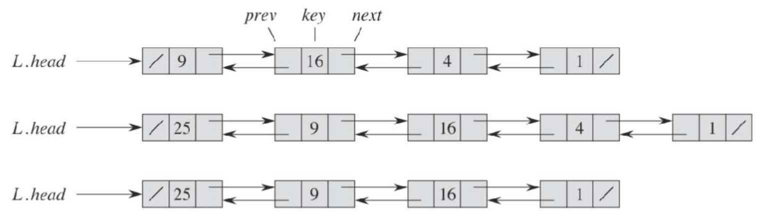
Il valore `NIL` viene assegnato ad una variabile puntatore che non presenta alcun indirizzo di memoria salvato

Liste

Definizione

Una lista è costituita da una **serie di nodi** i quali contengono una chiave (con eventuali dati satellite annessi) e dei **puntatori agli altri nodi**, così da creare una sequenza

A seconda del numero di collegamenti in ogni nodo, una lista può essere **unidirezionale**, **doppiamente collegata** o rappresentare strutture più complesse di una sequenza



Esempio di liste doppiamente collegate

Indicizzazione

Per quanto permetta una maggiore flessibilità rispetto ai vettori, si ha una perdita di efficienza rispetto a quest'ultimi in quanto non è possibile **indicizzare in tempo costante**

Iterazione

Quando viene iterata una lista, è importante non modificare direttamente il valore del puntatore alla testa, o sarà complicato riportarla al valore iniziale originale

LIST-INSERT(L, x)

```
x.next = L.head
if L.head != NIL
        L.head.prev = x
L.head = x
x.prev = NIL
```

***Parametri**: L=lista, x=puntatore al nodo da inserire (in testa)*

Costo Inserimento in Testa

$$\Theta(1)$$

LIST-SEARCH(L, k)

```
x = L.head
while x != NIL and x.key != k
        x = x.next
return x
```

***Parametri**: L=lista, k=chiave da cercare*
***Return**: puntatore all'elemento con la chiave richiesta o `NIL` se non presente*

Costo Ricerca

$$\Theta(n)$$

LIST-DELETE(L, x)

```
if x.prev != NIL
        x.prev.next = x.next
else
        L.head = x.next

if x.next != NIL
```

```
        x.next.prev = x.prev
```

Parametri: *L=lista, x=puntatore al nodo da eliminare*

ⓘ Costo Eliminazione

$$\Theta(1)$$

Osservazione: Eliminazione Tramite Chiave

Nell'operazione `LIST-DELETE` è richiesto di essere passato direttamente il puntatore `x` al nodo da eliminare.

Spesso però sarà necessario eliminare un nodo sapendone solo la chiave: in quel caso, basterà cercarlo in lista tramite la funzione `LIST-SEARCH`, rendendo il costo dell'eliminazione $\Theta(n)$

Utilizzo

Le liste permettono di creare sequenze dinamiche molto utili per diverse strutture dati

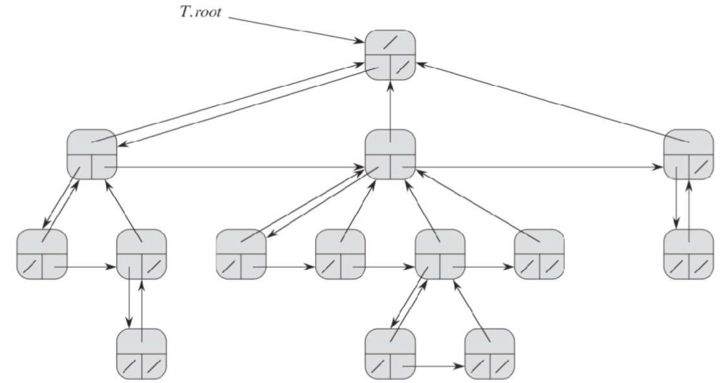
Esempio: Pile e Code

Creando delle pile e code tramite liste, avremo una sequenza di dati con grandezza dinamica mantenendo un costo costante nelle operazioni necessarie

Esempio: Alberi

Possiamo rappresentare **alberi irregolari di qualunque dimensione** assegnando ad ogni nodo 3 puntatori

- Un puntatore al **nodo padre**
- Un puntatore al **primo figlio**
- Una lista di **nodi fratelli**



È possibile ovviamente rappresentare anche alberi regolari (anche in maniera spesso più semplice), ottenendo una struttura piú flessibile rispetto ad un vettore

Operazioni aggiuntive

INVERTI-LISTA(l)

```
if l == NIL or l.next == NIL
        return l
else
        y = INVERTI-LISTA(l.next)
        l.next.next = l
        l.next = NIL
        return y
```

Parametri: *l=nodo di partenza di lista unidirezionale*

ⓘ Costo Inversione

$$\Theta(n)$$

DISPARI(l)

```
if l == NIL or l.next == NIL
        return l
```

```
else
        l.next = DISPARI(l.next.next)
        return l
```

Parametri: *l=nodo di partenza di lista unidirezionale*

Costo Dispari

$$\Theta(n)$$

PARI(l)

```
if l == NIL or l.next == NIL
        return NIL
else
        l = l.next
        l.next = PARI(l.next.next)
        return l
```

Parametri: *l=nodo di partenza di lista unidirezionale*

Costo Pari

$$\Theta(n)$$

Osservazione: Riutilizzo del Codice

Per la natura stessa del problema, possiamo utilizzare le funzioni pari e dispari una per risolvere l'altra

PARI(l)

```
if l == NIL or l.next == NIL
        return NIL
else
        return DISPARI(l.next)
```