

7 - Pile e Code

Pila

Specifica sintattica

Una pila è una sequenza di elementi di un certo tipo in cui è possibile **aggiungere o togliere elementi solo da un estremo della sequenza (la "testa")**.

Può essere visto come un caso speciale di lista in cui l'ultimo elemento inserito è il primo a essere rimosso (LIFO) e non è possibile accedere ad alcun elemento che non sia quello in testa.

Tipi: `pila` `boolean` `tipoelem`

Operatori:

Operatore	Input (Dominio)	Output (Codominio)
<code>creapila</code>	<code>()</code>	<code>pila</code>
<code>pilavuota</code>	<code>(pila)</code>	<code>boolean</code>
<code>leggipila</code>	<code>(pila)</code>	<code>tipoelemento</code>
<code>fuoripila</code>	<code>(pila)</code>	<code>pila</code>
<code>inpila</code>	<code>(tipoelemento.pila)</code>	<code>pila</code>

Il comportamento di questi operatori è il seguente:

- `creapila=p` : crea semplicemente una pila vuota ($p = \langle \rangle$)
- `pilavuota(p)=b` : definito anche come **empty**, fa in modo che b restituisca **vero** se la pila p è una sequenza vuota, **falso** se è presente qualcosa, in modo tale da controllare se la pila è vuota o no.
- `leggipila(p)=a` : definito anche come **top**, per il suo utilizzo richiede che la pila sia piena poiché non si può prelevare dalla cima se non è presente nulla. Se questo è vero allora questo operatore restituirà il primo valore presente in cima alla lista senza rimuoverlo (ricordo quando con la monopoli la creammo noi da 0 sta funzione).
- `fuoripila(p)=p'` : definita anche come **pop**, essa restituisce una **nuova** pila p' che è uguale alla pila originale **senza** il primo elemento ($p' = \langle a_2, a_3, \dots, a_n \rangle$). Se la pila aveva un solo elemento ($n=1$), la nuova pila sarà vuota ($p' = \langle \rangle$). Attenzione alcune pile restituiscono anche l'elemento rimosso tramite questa funzione ma non è valido per tutte.
- `inpila(a,p)=p'` : detta anche **push**, intuibile come quest'ultima funzione infine metta l'elemento a in cima alla nuova pila p' .

Realizzazioni

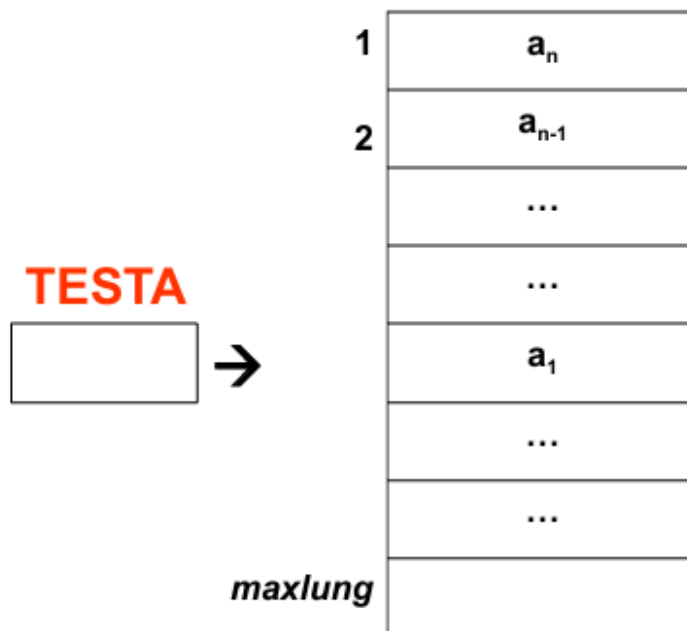
Una pila è semplicemente un **caso particolare di una lista**. È una lista in cui tutte le operazioni (inserimento, cancellazione, lettura) avvengono **solo su un estremo**: la testa della lista (cioè la `primolista`).

Di conseguenza, non abbiamo bisogno di reinventare le implementazioni. Possiamo mappare direttamente gli operatori della Pila (Stack) su quelli della Lista (List):

- `creapila()` corrisponde a `crealista()`.
- `pilavuota(p)` corrisponde a `listavuota(p)`.
- `leggipila(p)` (Legge la cima) corrisponde a `leggilista(primolista(p), p)` (Legge il primo elemento della lista).
- `fuoripila(p)` (Rimuove la cima) corrisponde a `canclista(primolista(p), p)` (Cancella il primo elemento della lista).
- `inpila(a, p)` (Aggiunge in cima) corrisponde a `inslista(a, primolista(p), p)` (Inserisce come primo elemento della lista).

Realizzazione con Vettore

Si utilizza un array di **dimensione fissa** (`maxlung`) e un "**cursore**" (un indice intero) che chiamiamo `TESTA`.



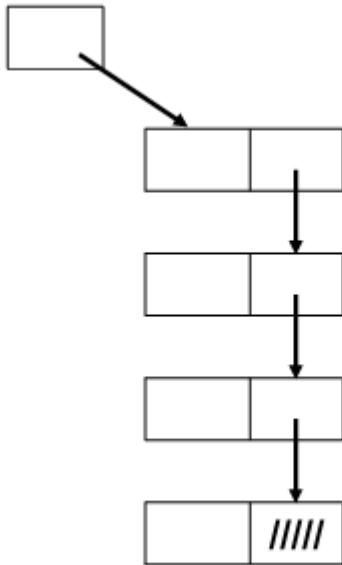
Il funzionamento di questa implementazione ha una complessità computazionale pari a $O(1)$. A differenza dell'implementazione *generale* della lista (dove `insLista` e `canclista` in mezzo richiedevano uno scorrimento), qui le operazioni avvengono solo a un'**estremità**. Il cursore `TESTA` tiene traccia della posizione del **primo elemento libero** (o dell'ultimo inserito).

- **`inpila (push)`**: Inserisce l'elemento nella posizione `TESTA` e incrementa `TESTA`. Questa è un'operazione a **tempo costante con C.C pari a: $O(1)$** .
- **`fuoripila (pop)`**: Decrementa `TESTA`, rendendo anch'essa a **tempo costante $O(1)$** .

- **leggipila (top):** Legge l'elemento alla posizione `TESTA-1`, con C.C. sempre a **tempo costante**, come i precedenti.

Realizzazione con Puntatori

L'implementazione è una semplice **lista concatenata**



A differenza della rappresentazione vettoriale essa ha il vantaggio di essere **dinamica** (non ha un `maxlung` fisso) e mantiene l'efficienza di $O(1)$ per tutte le operazioni.

Si utilizza un **singolo puntatore** (`testa`) che punta **solo ed esclusivamente** alla cella in cima alla pila.

- **inpila (push):** Crea una nuova cella, fa in modo che il suo puntatore `next` punti alla vecchia `testa`, e infine aggiorna la `testa` facendola puntare alla nuova cella (equivale a `inslista` in `testa`).
- **fuoripila (pop):** Salva un puntatore temporaneo alla `testa`, aggiorna la `testa` facendola puntare a `testa->next`, e dealloca la cella temporanea (equivale a `canclista` in `testa`).
- **leggipila (top):** Restituisce il valore contenuto nella cella puntata da `testa`.

Pile e Procedure Ricorsive

Una delle funzioni più particolari ed interessanti della pila è la possibilità di eseguire programmi **ricorsivi**.

L'esecuzione di una procedura ricorsiva prevede il **salvataggio dei dati** su cui lavora la procedura al momento della **chiamata** ricorsiva. Questi dati vengono letteralmente **impilati** in una pila e viene chiamata poi il programma interno che procede con la sua esecuzione. Quando la chiamata interna termina, il suo stato viene rimosso dalla pila (`pop`), e i dati della chiamata precedente vengono "ripristinati" (`top/pop`), permettendole di continuare da dove si era interrotta l'esecuzione esterna.

Se si studia il meccanismo appena descritto si nota come le chiamate ricorsive vengano *annidate* una dentro l'altra; la più **recente** è sempre la **prima a terminare**, questo non è altro

che il funzionamento del **LIFO**, ovvero una pila in tutto e per tutto.

Grazie a questa struttura possiamo sempre trasformare un programma ricorsivo in uno **iterativo**. I programmi ricorsivi, come sappiamo, sono ottimi per problemi di rango n poiché permettono in termini di avere soluzioni al problema di rango inferiore a n , è definibile una soluzione per assioma sul problema di rango minimo (che va da 0 a 1).

Un esempio pratico è l'**aspirale di Fibonacci**.

Questa successione è definita come una sequenza dove il $k - \text{esimo}$ elemento è uguale alla somma dei due che lo precedono, per $k > 2$.

$$fib(1) = 1 \quad fib(2) = 1 \quad fib(k) = fib(k-1) + fib(k-2)$$

Questa sequenza è facilmente ricreabile tramite funzione ricorsiva che calcola il $k - \text{esimo}$ numero della successione:

```
int fib(int k){
    if(k=1) or (k=2) then
        f=1
    else
        f=fib(k-1)+fib(k-2)
    return f
}

//Supponendo siano per k=4, nella pila vedremo
fib(4)
fib(3),fib(4)
fib(2),fib(3),fib(4)
fib(1),fib(3),fib(4)
```

La torre di Hanoi

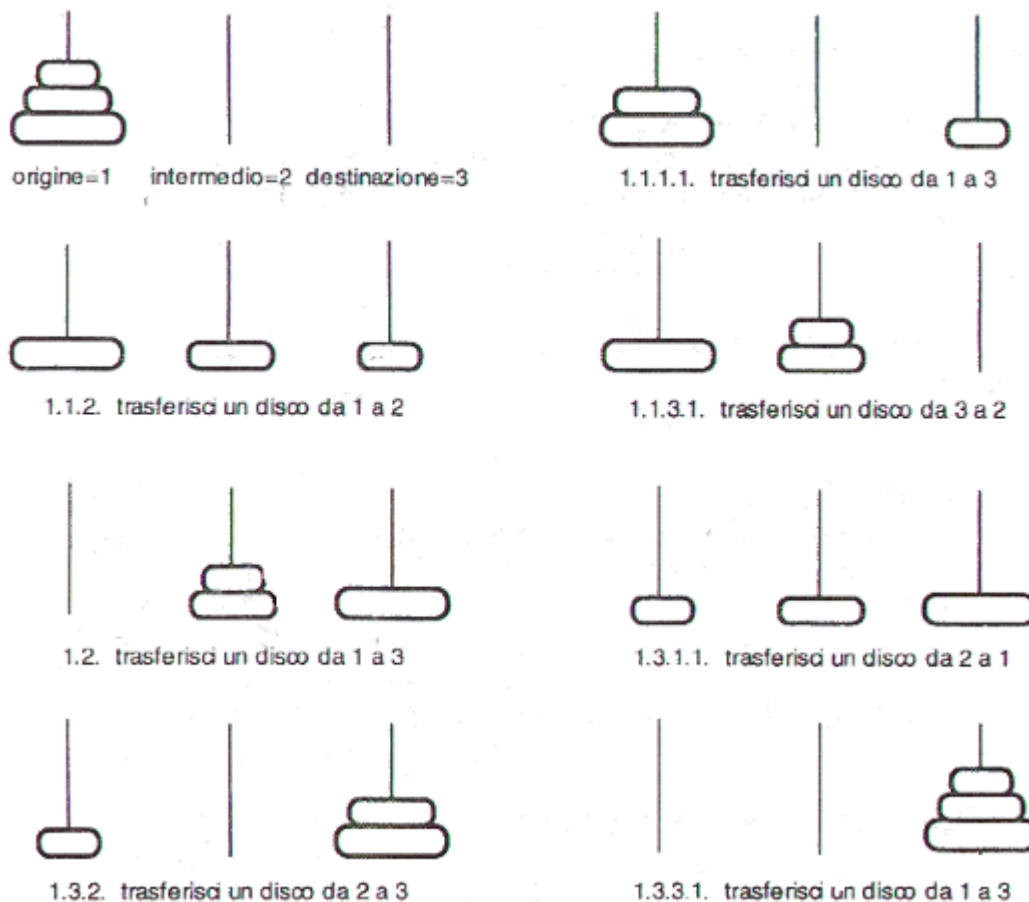
La logica dietro il problema del monastero tibetano dice:

Ci sono n dischi d'oro, tutti di diametro diverso, infilati in un piolo (origine).

Sono impilati in ordine di diametro decrescente, dal più grande in basso al più piccolo in cima.

- **Obiettivo:** Spostare l'intera pila di dischi su un altro piolo (destinazione), creando una catasta identica a quella di partenza.
- Le **Regole:**
 1. È possibile usare un terzo piolo ("intermedio") per gli spostamenti.
 2. Si può spostare **un solo disco per volta**.
 3. Non è **mai** possibile sovrapporre un disco più grande su uno più piccolo.

La soluzione ricorsiva



Come si può vedere nella figura, la strategia per spostare n dischi da un'origine a una destinazione è definita scomponendo il problema:

1. Muovi i primi $n - 1$ dischi (i più piccoli) dal piolo di *origine* al piolo *intermedio* (usando la destinazione come appoggio).
2. Muovi il disco rimanente (l'unico rimasto, il più grande) dal piolo di *origine* al piolo di *destinazione*.
3. Muovi gli $n - 1$ dischi che avevi "parcheggiato" sull'intermedio, dal piolo *intermedio* al piolo di *destinazione* (usando l'origine come appoggio).

In pseudo-code ricorsivo sarebbe:

```
move(int n, pole origine, pole destinazione, pole intermedio){
    if n=1 then
        muovi_un_disco_da_origine_a_destinazione
    else
        move(n-1, origine, intermedio, destinazione)
        muovi_un_disco_da_origine_a_destinazione
        move(n-1, intermedio, destinazione, origine)
}
```

Definizione dei tipi usati:

- `pole` : enumerativo con valore origine, destinazione e intermedio
- `interpos` : intero maggiore di 0.

Il codice diviene:

```

torre(){
    cin>>numerodischi
    cout<<"per "<<numerodischi<<"dischi i movimenti richiesti sono: "
    move(numerodischi,origine,intermedio,destinazione)
}

printpole(pole p){
    case p of
        origine: cout<<"origine"
        intermedio: cout<<"intermedio"
        destinazione: cout<<"destinazione"
    }
}

muovisorg_a_destin(){
    cout<<"muovi un disco da"
    printpole(origine)
    cout<<"a "
    printpole(destinazione)
    cout<<"usando come intermedio"
    printpole(intermedio)
}

move(interpos n, pole ori, pole intermedio, pole destin){
    if n=1 then
        muovisorg_a_destin()
    else
        move(n-1,ori,destin,intermedio)
        muovisorg_a_destin()
        move(n-1,intermedio,ori,destin)
}

```

La Trasformazione Iterativa

È sempre possibile trasformare una procedura ricorsiva in una iterativa usando una pila .

Il metodo consiste nel simulare manualmente ciò che il computer fa automaticamente con la pila delle chiamate di sistema:

- Si crea una Pila all'inizio.
- Ogni chiamata ricorsiva viene sostituita da una sequenza di istruzioni che:
 1. **Salva (in pila)** sulla Pila lo stato corrente: i valori dei parametri, le variabili locali e, soprattutto, un'etichetta di "ritorno" (per sapere dove riprendere dopo).
 2. Assegna ai parametri i nuovi valori per la "finta" chiamata successiva.
 3. Effettua un **goto** (salto) all'inizio della procedura.
- Alla fine della procedura, si aggiunge un blocco di codice che:
 1. Controlla se la Pila non è vuota.
 2. **Estrae (fuoripila / leggipila)** lo stato salvato.
 3. Esegue un **goto** all'etichetta di ritorno che era stata salvata.

L'Implementazione Iterativa

```
/* * Definizione di tipi (pseudocodice):
* tipoelem: elemento strutturato con componenti
* - numerodischi: interopos
* - piolorig, piolodest, pioloaus: pole
* - ritorno: intero
*/

// Procedura iterativa per la Torre di Hanoi
move(int n, pole sorgente, pole destinazione, pole ausiliario)
{
    // s è la pila per gestire la ricorsione
    creapila(s);

1: // Etichetta per l'inizio della "chiamata"
    if (n == 1) {
        muovisorg_a_destin;
        goto 3; // "Ritorna" dalla chiamata
    }

    // --- Simula la chiamata: move(n-1, sorgente, ausiliario,
    destinazione) ---
    // Salva lo stato corrente prima della "ricorsione"
    stato.numerodischi = n;
```

```

    stato.piolorig = sorgente;
    stato.pioloaus = ausiliario;
    stato.piolodest = destinazione;
    stato.ritorno = 2; // Indirizzo di ritorno
    inpila(stato, s);

    // Imposta i parametri per la nuova "chiamata"
    n = n - 1;
    // Scambia destinazione e ausiliario
    temp = destinazione;
    destinazione = ausiliario;
    ausiliario = temp;

    goto 1; // "Chiama"

2: // Etichetta per il ritorno dalla prima chiamata
    muovisorg_a_destin; // Mossa centrale

    // --- Simula la chiamata: move(n-1, ausiliario, destinazione,
    sorgente) ---
    // Salva lo stato corrente
    stato.numerodischi = n;
    stato.piolorig = sorgente;
    stato.pioloaus = ausiliario;
    stato.piolodest = destinazione;
    stato.ritorno = 3; // Indirizzo di ritorno
    inpila(stato, s);

    // Imposta i parametri per la nuova "chiamata"
    n = n - 1;
    // Scambia sorgente e ausiliario
    temp = sorgente;
    sorgente = ausiliario;
    ausiliario = temp;

    goto 1; // "Chiama"

3: // Etichetta per il "ritorno" (pop dalla pila)
    if (not pilavuota(s)) {
        // Ripristina lo stato precedente

```



```

    stato = leggipila(s);
    fuoripila(s);
    n = stato.numerodischi;
    sorgente = stato.piolorig;
    destinazione = stato.piolodest;
    ausiliario = stato.pioloaus;

    // Salta all'indirizzo di ritorno corretto
    case stato.ritorno of
    {
        2: goto 2;
        3: goto 3;
    }
}
// Se la pila è vuota, l'algoritmo termina
}

```

Questo codice è complesso, ma simula la ricorsione:

1. **Definizione:** Definisce un `tipoelem` (elemento della Pila) che è una struttura (`stato`) per salvare tutti i parametri (`n`, `pioli`) e l'etichetta di ritorno .
2. **Inizio:** Crea una Pila (`creapila(s)`) e definisce l'etichetta `1:` .
3. **Prima Chiamata Ricorsiva (Sostituzione):** La prima `move(n-1, ...)` della versione ricorsiva viene sostituita :
 - Salva lo stato attuale e `stato.ritorno = 2` (l'etichetta `2:` da cui ripartire) .
 - Fa `inpila(stato, s)` .
 - Aggiorna le variabili per la chiamata `n-1` .
 - Salta a `goto 1` .
4. **Seconda Chiamata Ricorsiva (Sostituzione):** Alla label `2:` , dopo aver mosso il disco , la seconda `move(n-1, ...)` viene sostituita :
 - Salva lo stato attuale e `stato.ritorno = 3` (l'etichetta `3:` che è la fine) .
 - Fa `inpila(stato, s)` .
 - Aggiorna le variabili .
 - Salta a `goto 1` .
5. **Il Ritorno (Gestione Pila):** All'etichetta `3:` , gestisce la Pila:
 - Se la pila non è vuota, estrae l'ultimo stato (`stato = leggipila(s)`, `fuoripila(s)`) .
 - Ripristina tutte le variabili (`n = stato.numerodischi` , ecc.).
 - Usa un `case` per saltare all'etichetta di ritorno salvata (`goto 2` o `goto 3`)

Valutazione di un'espressione postfissa

Calcolare un'espressione aritmetica complessa, come

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

non è banale perché richiede di **memorizzare i risultati intermedi** (ad esempio, $9+8$ deve essere calcolato e "messo da parte" prima di essere moltiplicato).

- **Lo Strumento:** Una **Pila** (Stack) è il meccanismo ideale per gestire questa memoria temporanea, grazie alla sua natura LIFO.
- **La Semplificazione (Notazione Postfissa):** Invece di valutare la complessa notazione "infissa" (dove l'operatore sta *tra* gli operandi), la slide introduce un problema più semplice: valutare un'espressione in **forma postfixa** (o "notazione polacca inversa").
 - **Definizione di Postfissa:** In questa notazione, ogni operatore appare *dopo* i suoi due argomenti.
 - Esempio: $(9 + 8)$ diventa $9\ 8\ +$.

Conversione di un'espressione infissa in postfixa

Ma come otteniamo l'espressione postfixa?

È possibile **usare una Pila anche per convertire** un'espressione da infissa (con parentesi) a postfixa.

L'Algoritmo di Conversione: è un'algoritmo preciso che scorre l'espressione infissa da sinistra a destra:

1. **Se incontri un numero (operando):** Lo scrivi direttamente nell'**output**.
2. **Se incontri una parentesi aperta (:** La ignori.
3. **Se incontri un operatore (+ , * , ...):** Lo inserisci (push) nella **Pila**.
4. **Se incontri una parentesi chiusa) :** Estrai (pop) l'operatore che sta in cima alla Pila e lo scrivi nell'**output**.

Tracciamento (Trace) dell'Esempio: L'esecuzione di questo algoritmo sull'espressione $5 * (((9 + 8) * (4 * 6)) + 7)$ (le parentesi esterne sono implicite per la regola 4).

Input Letto	Azione	Output	Pila
5	Scrivi numero	5	< >
*	Push operatore	5	< * >
9	Scrivi numero	5 9	< * >
+	Push operatore	5 9	< + , * >
8	Scrivi numero	5 9 8	< + , * >
)	Pop operatore	5 9 8 +	< * >
*	Push operatore	5 9 8 +	< * , * >
4	Scrivi numero	5 9 8 + 4	< * , * >

Input Letto	Azione	Output	Pila
*	Push operatore	5 9 8 + 4	<*, *, *>
6	Scrivi numero	5 9 8 + 4 6	<*, *, *>
)	Pop operatore	5 9 8 + 4 6 *	<*, *>
)	Pop operatore	5 9 8 + 4 6 * *	<*>
+	Push operatore	5 9 8 + 4 6 * *	<+, *>
7	Scrivi numero	5 9 8 + 4 6 * * 7	<+, *>
)	Pop operatore	5 9 8 + 4 6 * * 7 +	<*>
)	Pop operatore	5 9 8 + 4 6 * * 7 + *	< >

Alla fine, la pila è vuota e l'output è l'espressione postfissa corretta.

Coda

Una Coda è un tipo astratto per una sequenza di elementi, ma con regole di accesso specifiche.

Funzionamento:

- È possibile **aggiungere** elementi solo a un estremo, chiamato "**fondo**" (o retro).
- È possibile **togliere** elementi solo dall'altro estremo, chiamato "**testa**".

Questa strategia è nota come **FIFO (First In First Out)**. Il primo elemento che è entrato sarà il primo ad uscire.

La coda è perfetta per rappresentare situazioni di **attesa** (come una fila alla cassa), dove chi arriva prima viene servito per primo.

Sintassi

- **Tipo:** coda, boolean, tipoelem
- **Operatori:**

Operatore	Input (Dominio)	Output (Codominio)	Descrizione
creacoda	()	coda	Crea una coda nuova.
codavuota	(coda)	boolean	Controlla se la coda è vuota.
leggicoda	(coda)	tipoelem	Legge l'elemento in testa.
fuoricoda	(coda)	coda	Rimuove l'elemento in testa.
incoda	(tipoelem, coda)	coda	Aggiunge un elemento in fondo.

Specifica Semantica

Una coda q è vista come una sequenza $q = \langle a_1, a_2, \dots, a_n \rangle$, dove a_1 è la **testa** (il "First In") e a_n è il **fondo**.

- **creacoda** = q' : Crea una coda vuota, $q' = \langle \rangle$.
- **codavuota**(q) = b : Restituisce vero se q è vuota, falso altrimenti.
- **leggicoda**(q) = a (**Leggere dalla Testa**): La coda non deve essere vuota ($n \geq 1$) e se non è vuota, restituisce l'elemento in testa, $a = a_1$.
- **fuoricoda**(q) = q' (**Rimuovere dalla Testa**): Prima di essere eseguito bisogna controllare che la coda non sia vuota ($n \geq 1$). Dopo di che restituisce una nuova coda q' che contiene tutti gli elementi, *cancellando* il primo: $q' = \langle a_2, a_3, \dots, a_n \rangle$.
- **incoda**(a, q) = q' (**Aggiungere in Fondo**): Premettendo che non ci siano elementi nella coda ($n \geq 0$). Esso restituisce una nuova coda q' con il nuovo elemento a aggiunto **in fondo** alla sequenza: $q' = \langle a_1, a_2, \dots, a_n, a \rangle$.

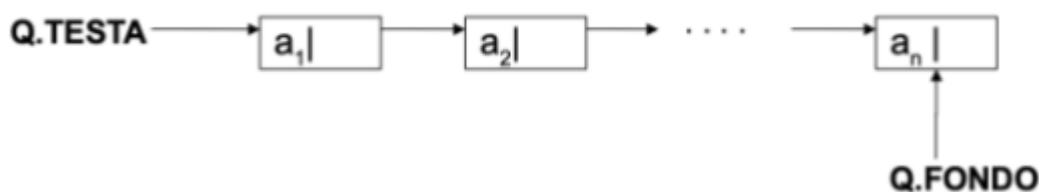
Rappresentazione

A differenza della Pila (dove si agisce solo su un estremo), la coda richiede accesso a **entrambi gli estremi** poiché eseguo la prima operazione aggiungere in fondo, successivamente deve togliere dalla testa.

La rappresentazione sequenziale (un semplice array) che era accettabile per la pila, qui diventa **inappropriata**.

Questo perché se usassimo un array semplice, l'operazione **fuoricoda** (togliere dalla testa, indice 0) richiederebbe di "**spostare**" tutti gli altri elementi di una posizione a sinistra. Questa è un'operazione molto costosa: $O(n)$, cioè il suo tempo di esecuzione dipende dalla **grandezza** della coda.

Rappresentazione con Puntatori



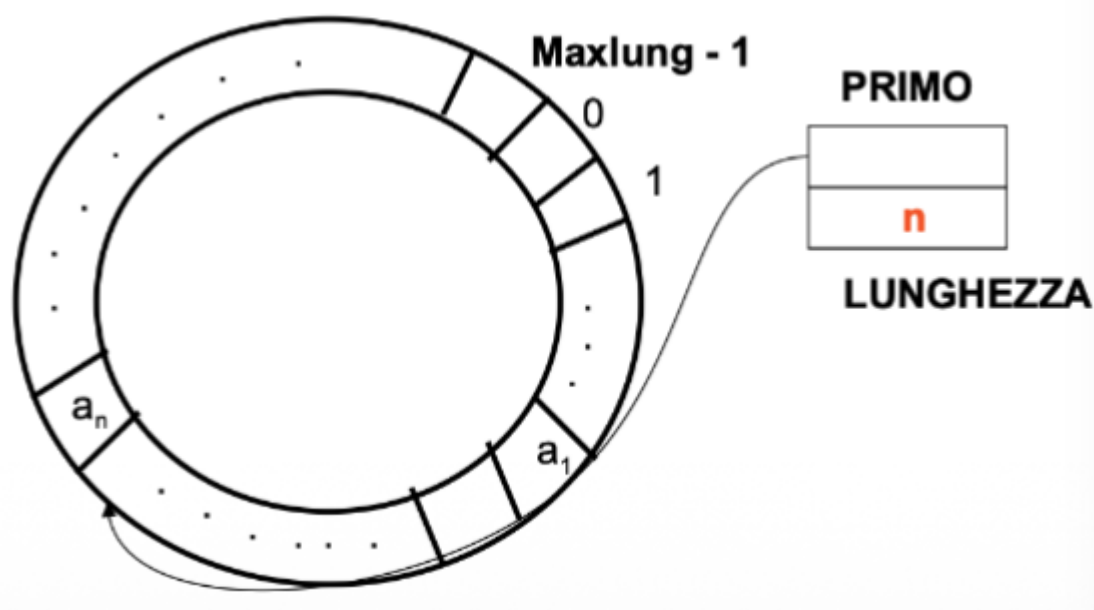
Questa implementazione è la più **flessibile**, è basata su una **lista concatenata** ove vengono usati **due puntatori**:

1. **Q.TESTA** : che punta alla prima cella della lista
2. **Q.FONDO** che punta all'ultima cella della lista

In modo tale che **Q.TESTA** venga usata per le funzione di **leggicoda** e **fuoricoda**, mentre **Q.FONDO** svolge le operazioni di **incoda**.

In questo modo è facilmente riconoscibile la **coda vuota**, poiché il puntatore alla testa risulterà **NULL**.

Realizzazione con Vettore Circolare



Questa è la variante "intelligente" che risolve il problema dello spostamento verso sinistra nell'array.

Invece di spostare gli elementi, in questa versione si **spostano i puntatori**.

Si usano due indici:

1. **PRIMO** : indica la posizione dell'elemento in testa (a_1).
2. **LUNGHEZZA** : tiene traccia di quanti elementi ci sono o dove si trova l'ultimo.

• Operazioni (Esempio):

- **incoda(e) (Aggiungere)**: Si inserisce e nella posizione $(\text{PRIMO} + \text{LUNGHEZZA}) \% \text{maxlung}$ e si incrementa **LUNGHEZZA**.
- **fuoricoda() (Togliere)**: Non si tocca l'array, si incrementa solo l'indice $\text{PRIMO} = (\text{PRIMO} + 1) \% \text{maxlung}$ e si decrementa $\text{LUNGHEZZA} - 1$.

Anche in questo caso, entrambe le operazioni principali **incoda** e **fuoricoda** diventano a **tempo costante** $O(1)$. Il suo unico limite è la dimensione fissa **maxlung** dell'array ma sicuramente aumenta l'efficienza dei precedenti.

esercizi su code e pile:

Data una coda q di interi, creare una nuova coda q_1 che contenga solo i numeri positivi di q .

Algoritmo (estrai):

1. Crea una q_1 vuota.
2. Finché q non è vuota:
3. Leggi l'elemento e dalla testa di q .
4. Se $e > 0$, mettilo in coda su q_1 .
5. Togli l'elemento da q (**fuoricoda**(q)).

```

estrai (coda q per riferimento, coda q1 per riferimento)
    creacoda(q1)
    while not codavuota(q)
        e = leggicoda(q)
        if e > 0 then
            incoda(e, q1)
        fuoricoda(q)

```

Attenzione! l'algoritmo **distrugge la coda originale** q . Alla fine del processo, q sarà vuota.

Per risolvere questo problema bisogna utilizzare una **coda ausiliaria** `qaux` :

```

estrai1 (coda q per riferimento, coda q1 per riferimento)
    creacoda(q1)
    creacoda(qaux)
    while not codavuota(q) do
        e = leggicoda(q)
        if e > 0 then
            incoda (e, q1)
        fuoricoda(q)
        incoda(e, qaux)
    creacoda(q)

// ripristino della coda originaria

while not codavuota(qaux) do
    e = leggicoda(qaux)
    incoda (e, q)
    fuoricoda (qaux)

```

Algoritmo (`estrai1`):

1. Crea `q1` e `qaux` vuote.
2. **Primo ciclo (Svuotamento di `q`):**
 - Finché `q` non è vuota:
 - Leggi `e` da `q` .
 - Se `e > 0` , metti `e` in `q1` .
 - Togli `e` da `q` .
 - **Metti `e` in `qaux`** (questo salva ogni elemento, positivi o no, nell'ordine originale).
3. **Secondo ciclo (Ripristino di `q`):**

- Ora `q` è vuota e `qaux` contiene tutti gli elementi originali.
- Finché `qaux` non è vuota:
- Leggi `e` da `qaux`, togliilo da `qaux` e rimettilo in `q`.