

# **Algoritmi e Strutture Dati**

Schifano S. Fabio `schifano@fe.infn.it`

## Strutture Dati: Pile

La **pila** é una struttura dati che rappresenta una sequenza di elementi in cui:

- ▷ é possibile aggiungere elementi nuovi soltanto da un estremo chiamato **testa**
- ▷ é possibile togliere elementi soltanto da un estremo, chiamato **testa**

Esempio: Una catasta di libri o cartelle su una scrivania gestita in modo tale che l'aggiunta o il prelievo di un elemento avvenga sempre da un'estremo della catasta é un tipico esempio di pila.

La politica di gestione delle **pile**, chiamate anche **stack** o **pushdown list**, é chiamata **LIFO**, abbreviazione di **last-in first-out**.

## Specifica Sintattica delle Pile

La specifica sintattica delle pile, ovvero la definizione dei:

▷ nomi di tipo: **pila**, **tipoelem**

▷ nomi delle funzioni e relativi domini:

CREAPILA:	$() \longrightarrow \text{pila}$
PILAVUOTA:	$(\text{pila}) \longrightarrow \text{boolean}$
LEGGIPILA:	$(\text{pila}) \longrightarrow \text{tipoelem}$
FUORIPILA:	$(\text{pila}) \longrightarrow \text{pila}$
INPILA:	$(\text{tipoelem}, \text{pila}) \longrightarrow \text{pila}$

▷ nomi delle costanti: nessuna

Il nome comunemente più usato per l'operazione **FUORIPILA** é **POP**, mentre per l'operazione **INPILA** é **PUSH**.

## Specifica Semantica delle Pile

Indichiamo  $P$  una generica sequenza di elementi  $a_1, \dots, a_m$  di tipo **tipoelem**, la specifica semantica delle pile é così definita:

- ▷  $\text{CREAPILA}() = P'$   
Post:  $P' = \Lambda$ , la pila vuota
- ▷  $\text{PILAVUOTA}(P) = b$   
Post:  $b = \text{True}$  se  $P = \Lambda$ ,  $b = \text{False}$  altrimenti
- ▷  $\text{LEGGIPILA}(P) = a$   
Pre:  $P = a_1, a_2, \dots, a_n, \quad n \geq 1$   
Post:  $a = a_1$
- ▷  $\text{FUORIPILA}(P) = P'$   
Pre:  $P = a_1, a_2, \dots, a_n, \quad n \geq 1$   
Post:  $P' = a_2, \dots, a_n$  se  $n \geq 1$ ,  $P' = \Lambda$  se  $n = 1$
- ▷  $\text{INPILA}(a, P) = P'$   
Pre:  $P = a_1, a_2, \dots, a_n, \quad n \geq 0$   
Post:  $P = a, a_1, a_2, \dots, a_n$  se  $n \geq 1$ ,  $P' = a$  se  $n = 0$

## Realizzazione di una Pila con un Vettore

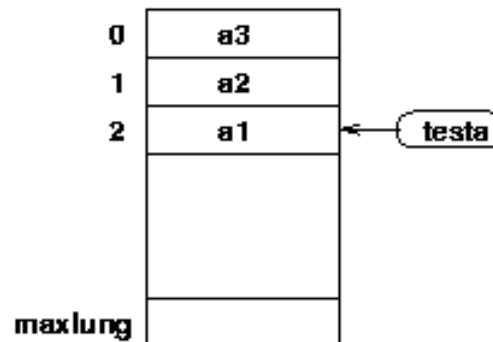
Una pila può essere implementata tramite un vettore di lunghezza predefinita **MAXLUNG**.

Ovviamente questa implementazione è di tipo **statica**.

Utile se si può dimensionare a priori il numero massimo degli elementi che la pila deve contenere.

Descrizione dell'implementazione:

- ▶ gli elementi della pila sono memorizzati come elementi di un vettore di lunghezza **MAXLUNG** i cui elementi sono di tipo **tipoelem**
- ▶ la **testa** (**head**), della pila è individuata dall'elemento di indice uguale al valore di una variabile detta **cursore**
- ▶ quando il **cursore** vale 0 (o  $-1$  dipende dall'implementazione), si ha la condizione di pila **vuota** (empty)
- ▶ quando il **cursore** vale **MAXLUNG** (o  $\text{MAXLUNG}-1$  dipende dall'implementazione) si ha la condizione di pila **satura** (piena o full)



## Realizzazione in C di una pila di Integer tramite vettore

```
#define MAXLUNG maxlung;

typedef int tipoelem;

struct structpila {
    int      testa;
    tipoelem elementi[MAXLUNG];
};

typedef struct structpila * pila;
```

Il tipo di dato **pila** é definito come un puntatore ad un record che definisce la struttura della pila.

La struttura della pila **structpila** contiene due informazioni

- ▷ **testa**, la variabile cursore che punta all'elemento in cima (**head**) alla pila (**stack**)
- ▷ **elementi** il vettore che contiene i valori degli elementi della pila

In questa specifica implementazione l'operazione di CREAPILA avrà il compito di allocare fisicamente lo spazio di memoria per contenere gli elementi della pila, cioè effettuare una chiamata di sistema alla funzione **malloc**.

Le operazioni sulla pila sono implementate nel seguente modo:

```
pila CREAPILA ( ) {  
    pila P;  
    P = (pila) malloc(sizeof(struct structpila));  
    P->testa = -1;  
    return P; /* return pointer to a "structpila" */  
}
```

La funzione CREAPILA effettua le seguenti operazioni:

1. dichiara una variabile  $P$  di tipo **pila**
2. la inizializza con un indirizzo ad uno spazio di memoria che può contenere un record di tipo **structpila**.  
Quanti byte deve allocare?  
Un intero occupa 4 bytes o una cella di memoria a 32-bit, quindi il numero totale di bytes necessari per memorizzare un record di tipo **structpila** é:  
$$4 + 4 * \text{MAXLUNG}$$
**testa** sarà memorizzata all'offset 0, il primo elemento all'offset 4 (in bytes), il secondo all'offset 8, . . .
3. inizializza il valore del cursore **testa** ad un indice non valido, cioè in modo da non farlo puntare a nessun elemento del vettore (e.g. -1 in C).
4. ritorna il **valore** di  $P$  come valore di ritorno, cioè l'indirizzo di memoria a partire dal quale si trova memorizzata la struttura **structpila**.

```
int PILAVUOTA ( pila P ) {  
    return ( P->testa == -1 ) ? 1 : 0;  
}
```

Il predicato (funzione booleana) PILAVUOTA verifica se il valore del cursore non punta ad alcun elemento del vettore, cioè, ad esempio in C se il valore del cursore é uguale a  $-1$ .

```
tipoelem LEGGIPILA ( pila P ) {  
    tipoelem elemento;  
    if ( ! PILAVUOTA ( P ) ) {  
        elemento = P->elementi[P->testa];  
    } else {  
        fprintf(stderr, "ERROR pila vuota !\n");  
        elemento = -1;  
    }  
    return elemento;  
}
```

La funzione LEGGIPILA, se la condizione PILAVUOTA é **falsa**, resituisce il valore dell'elemento contenuto nel vettore all'indice specificato dal valore del cursore.

Se la pila é **vuota** stampa un messaggio di errore ed il valore dell'elemento restituito non può essere considerato valido.



```
void FUORIPILA ( pila P ) { // pop
    if ( ! PILAVUOTA ( P ) ) {
        P->testa = P->testa - 1;
    }
}
```

La procedura FUORIPILA **elimina** un elemento dalla pila se la pila non é **vuota**.

Tale effetto é ottenuto semplicemente decrementando il valore della variabile cursore.

```
void DISTRUGGIPILA ( pila P ) {
    if ( ! PILAVUOTA(P) ) {
        fprintf(stderr, "WARNING: destroying a non void pila\n");
    }
    free(P);
}
```

La procedura **DISTRUGGIPILA** dealloca la memoria allocata per una struttura **structpila**. Utile nel caso in cui si ha bisogno di creare delle pile temporanee come strutture di appoggio.

```
void INPILA ( tipoelem a, pila P ) { // push
    int ret = 0;
    if ( P->testa == MAXLUNG ) {
        fprintf(stderr, "ERROR pila satura !\n");
        ret = -1;
    } else {
        P->testa = P->testa + 1;
        fprintf(stderr, "testa: %d\n", P->testa);
        P->elementi[P->testa] = a;
    }
    return ret;
}
```

La procedura INPILA inserisce un nuovo elemento in cima alla pila, ovvero dopo l'elemento puntato dalla variabile cursore.

Se la pila é **piena**, cioè, in questa specifica implementazione, se la variabile cursore vale **MAXLUNG**, emette un messaggio di errore.

Altrimenti, se la pila non é **piena**, incrementa il valore del cursore e memorizza il nuovo elemento.

## Osservazioni

Il tipo di dato pila é stato definito come un tipo di dato **astratto** cioè chi scrive il programma non ha bisogno di conoscere l'implementazione interna della struttura, ma é sufficiente conoscere la semantica di ciascuna operazione, funzione o procedura, che manipola il tipo di dato.

Fra i tipi di esercizi che facciamo distinguiamo:

1. esercizi che chiedono di **definire** un tipo di dato, ovvero di dare la specifica, **sintattica** e **semantica**
2. esercizi che chiedono di **realizzare** o **implementare** un certo tipo di dato in C. In questo caso bisogna dare l'implementazione del tipo di dato in base alle sue specifiche
3. esercizi che chiedono di **utilizzare** un tipo di dato già definito a lezione. In questo caso bisogna utilizzare il tipo di dato così come é stato specificato ed implementato a lezione
4. esercizi che chiedono di **estendere** un certo tipo di dato già definito. In questo caso bisogna dare la specifica e l'implementazione delle estensioni.

## Funzione APPARTIENEPILA

Data una pila  $P$  ed un elemento  $a$ , l'unico modo che abbiamo per verificare se l'elemento  $a$  appartiene alla pila  $P$  é di ripetere la seguente sequenza di operazioni:

1. leggere un elemento della pila tramite la funzione LEGGIPILA
2. verificare se l'elemento letto sia uguale a quello cercato
3. in caso di esito negativo, **eliminare** l'elemento di testa tramite la procedura FUORIPILA

finché, o si trova l'elemento cercato o la pila si é svuotata.

L'algoritmo APPARTIENEPILA appena definito ha il grosso svantaggio di distruggere la pila in cui si vuole effettuare la ricerca.

Per ovviare a questo problema abbiamo due possibilità:

1. la funzione predicato APPARTIENEPILA scandisce la pila  $P$ , secondo l'algoritmo descritto precedentemente, ed inserisce gli elementi letti in una seconda pila  $Q$ . Alla fine, indipendentemente dall'esito della ricerca, bisogna re-inserisce gli elementi di  $Q$  in  $P$ .
2. **estendere** il tipo di dato pila per definire un'operatore ad-hoc che scandisce il vettore (in questo caso abbiamo visibilità dell'implementazione) e verifica se l'elemento cercato é nella pila o meno.

## Funzione APPARTIENEPILA versione distruttiva

```
int APPARTIENEPILA(tipoelem a, pila P) {  
    int      found = 0;  
    tipoelem tmp;  
  
    while ( ! found && ! PILAVUOTA(P) ) {  
  
        if ( a == LEGGIPILA(P) ) {  
            found = 1;  
        } else {  
            FUORIPILA(P); // l'unico modo che abbiamo per muovere il cursore  
        }  
  
    }  
  
    return found;  
}
```

## Esercizio: APPARTIENEPILA versione non distruttiva

**Esercizio:** definire una implementazione della funzione APPARTIENEPILA che **non** distrugga la pila  $P$ :

```
int APPARTIENEPILA(tipoelem a, pila P) {  
    int found = 0;  
    pila Q;           // pila di appoggio  
    Q = CREAPILA();  
    while ( ! found && ! PILAVUOTA(P) ) {  
        if ( a == LEGGIPILA(P) ) {  
            found = 1;  
        } else {  
            INPILA(LEGGIPILA(P), Q);  
            FUORIPILA(P);  
        }  
    }  
    while ( ! PILAVUOTA(Q) ) {  
        a = LEGGIPILA(Q);  
        INPILA(a, P);  
        FUORIPILA(Q);  
    }  
    DISTRUGGIPILA(Q);  
    return found;  
}
```

## Esercizio

**Estendere** il tipo di dato pila **implementato** tramite vettore con l'operatore: **APPARTIENEPILA** avente la seguente specifica:

APPARTIENEPILA: (tipoelem, pila)  $\longrightarrow$  boolean

APPARTIENEPILA: (a, p ) = b

Pre:  $P = ( a_1, a_2, \dots, a_n )$

Post:  $b = \text{True}$  se  $\exists 1 \leq i \leq n \mid a_i = a$ ,  $b = \text{False}$  altrimenti

```
int APPARTIENEPILA ( int a, pila P ) {  
    int found, testa;  
  
    found = 0;  
    testa = P->testa;  
  
    while ( ! found && ( testa != -1 ) ) {  
        if ( P->elementi[testa] == a ) {  
            found = 1;  
        } else {  
            testa--;  
        }  
    }  
    return found;  
}
```

## Esercizio

Scrivere una funzione C che calcola il fattoriale di un numero naturale **utilizzando** una pila.

```
int fattoriale ( int n ) {  
    pila P;  
    int f;  
  
    for ( i = 1; i <= n; i++ ) {  
        INPILA(i, P);  
    }  
  
    f = 1;  
  
    while ( ! PILAVUOTA(P) ) {  
        f = f * LEGGIPILA(P);  
        FUORIPILA(P);  
    }  
  
    return f;  
}
```

La soluzione dell'esercizio non presuppone alcuna specifica implementazione del tipo di dato pila.

Nel caso di implementazione con vettore la chiamata alla funzione **INPILA** fallisce per valori di  $n > \text{MAXLUNG}$ .



## Realizzazione di una pila tramite liste

Il tipo di dato pila può essere realizzato utilizzando la definizione di tipo di dato lista in cui le operazioni sulle code sono realizzate tramite operazioni sulle liste:

- ▷  $\text{CREAPILA}() = \text{CREALISTA}()$
- ▷  $\text{PILAVUOTA}(P) = \text{LISTAVUOTA}(P)$
- ▷  $\text{LEGGIPILA}(P) = \text{LEGGILISTA}(\text{PRIMOLISTA}(P))$
- ▷  $\text{FUORIPILA}(P) = \text{CANCLISTA}(\text{PRIMOLISTA}(P))$
- ▷  $\text{INPILA}(a, P) = \text{INSLISTA}(a, \text{PRIMOLISTA}(P))$

Alternativamente posso definire le operazioni sulla pila nel seguente modo:

- ▷  $\text{CREAPILA}() = \text{CREALISTA}()$
- ▷  $\text{PILAVUOTA}(P) = \text{LISTAVUOTA}(P)$
- ▷  $\text{LEGGIPILA}(P) = \text{LEGGILISTA}(\text{ULTIMOLISTA}(P))$
- ▷  $\text{FUORIPILA}(P) = \text{CANCLISTA}(\text{ULTIMOLISTA}(P))$
- ▷  $\text{INPILA}(a, P) = \text{INSLISTA}(a, \text{SUCCLISTA}(\text{ULTIMOLISTA}(P)))$

## Realizzazione di una pila in C tramite liste

```
typedef lista pila;  
  
lista CREAPILA () {  
    lista L;  
    L = CREALISTA();  
    return L;  
}
```

In questa specifica implementazione le pile sono implementate tramite una lista, cioè un'altro tipo di dato che abbiamo già definito.

Una pila é dichiara essere una lista, quindi i suoi operatori saranno definiti tramite operatori sulle liste.

In virtù del fatto che vogliamo realizzare un tipo di dato astratto, l'**interfaccia** del tipo di dato pila deve restare invariata, ovvero la definizione sintattica e semantica degli operatori della pila deve rimanere invariata.

## Realizzazione di una pila in C tramite liste

```
int PILAVUOTA (pila P) {
    return LISTAVUOTA(P);
}

tipoelem LEGGIPILA ( pila P ) {
    return (LEGGILISTA(ULTIMOLISTA(P)));
}

void FUORIPILA ( pila P ) {
    posizione p = ULTIMOLISTA(P);
    CANCLISTA(&p);
}

void INPILA ( tipoelem a, pila P) {
    posizione p = SUCCLISTA(ULTIMOLISTA(P)); // NB: INSLISTA e' una operazione PREINS
    INSLISTA(a, &p);
}

void DISTRUGGIPILA (pila P) {
    if ( PILAVUOTA(P) ) {
        free(P);
    } else {
        fprintf(stderr, "ERROR: trying to destroy a non void pila\n");
    }
}
```

## Esercizi

1. Scrivere una procedura  $C$  che copia il contenuto di una pila in un'altra pila indipendentemente dall'implementazione del tipo di dato pila.
2. **Estendere** il tipo di dato pila implementato tramite liste, che distrugge il contenuto di una pila . Occorre scorrere la lista e deallocare tutti i suoi elementi. Scrivere una versione iterativa ed una ricorsiva.
3. **Estendere** il tipo di dato pila **implementato** tramite un vettore con il seguente operatore: **STAMPAPILA**, che presa in input una pila ne stampa il contenuto.
4. **Estendere** il tipo di dato pila **implementato** tramite un vettore con il seguente operatore: **POPPILA**, la cui specifica semantica é la seguente:  
POPPILA: pila  $\longrightarrow$  tipoelem  
POPPILA(P) = a

Pre:  $P = (a_1, a_2, \dots, a_n)$  Post:

- ▷  $P = (a_2, \dots, a_n)$  e  $a = a_1$  se  $n > 1$
- ▷  $P = \Lambda$  e  $a = a_1$  se  $n = 1$
- ▷ ERROR se  $n = 0$

## Strutture Dati: Le code

La struttura dati **coda** o **queue**:

- ▷ é una struttura dati sequenziale
- ▷ ha un punto di accesso per immettere i dati chiamato **testa**
- ▷ ha un secondo punto di accesso per estrarre i dati chiamato **fondo**

La politica di accesso alle code é chiamata **FIFO** acronimo di **First-In First-Out**.

Una coda puó essere intesa come una particolare lista in cui:

- ▷ il primo elemento ad essere inserito é anche il primo ad essere estratto
- ▷ non é possibile accedere ad alcun altro elemento della struttura se non a quello di testa o a quello di fondo

**Esempio:** una fila di persone in attesa per un qualche servizio é un tipico esempio di coda.

In un sistema operativo multi-processo, un insieme di processi pronti ad essere eseguito puó essere memorizzato in una coda che determina la politica di scheduling del sistema operativo.

## Le code: Specifica Sintattica e Semantica

Definiremo il tipo di dato **coda** come un tipo di dato astratto, quindi:

1. diamo la specifica sintattica:

- ▷ definiamo i nomi di tipo
- ▷ definiamo i nomi dei domini e codomini delle operazioni sulle code
- ▷ definiamo i nomi delle costanti

2. diamo la specifica semantica:

- ▷ associamo un insieme ad ogni nome di tipo
- ▷ associamo una funzione o procedura ad ogni operazione
- ▷ associamo un valore ad ogni costante

3. infine diamo una **specifica** implementazione del tipo di dato in un **particolare** linguaggio di programmazione

N.B.: i primi due punti sono **sufficienti** a definire algoritmi che fanno uso del tipo di dato **coda**.

## Le Operazione sulle Code:

Le operazioni tipiche della struttura dati coda sono:

- ▷ CREACODA: crea una coda vuota
- ▷ CODAVUOTA: predicato che verifica se una coda é vuota
- ▷ LEGGICODA: funzione che legge l'elemento di testa della coda
- ▷ FUORICODA: procedura che estrae un elemento dalla coda
- ▷ INCODA: procedura che inserisce un elemento in coda

Tali operazioni ci permettono di manipolare il contenuto di coda.

## Le code: Specifica Sintattica e Semantica

▷ CREACODA:  $() \longrightarrow \text{coda}$

$\text{CREACODA}() = Q'$

Pre:

Post:  $Q' = \Lambda$

▷ CODAVUOTA:  $\text{coda} \longrightarrow \text{boolean}$

$\text{CODAVUOTA}(Q) = b$

Pre:

Post:  $b = \text{True}$  se  $Q = \Lambda$ ,  $b = \text{False}$  altrimenti

▷ LEGGICODA:  $\text{coda} \longrightarrow \text{tipoelem}$

$\text{LEGGICODA}(Q) = a$

Pre:  $Q = (a_1, a_2, \dots, a_n)$

Post:  $a = a_1$



## Le code: Specifica Sintattica e Semantica

▷ FUORICODA: coda  $\longrightarrow$  coda

**FUORICODA(Q) = Q'**

Pre:  $Q = (a_1, a_2, \dots, a_n), n \geq 1$

Post:  $Q' = (a_2, \dots, a_n)$ , se  $n > 1$

Post:  $Q' = (\Lambda)$ , se  $n = 1$

Q' deve essere inteso come Q modificato e non come return value

▷ INCODA: (tipoelem, coda)  $\longrightarrow$  coda

**INCODA(a, Q) = Q'**

Pre:  $Q = (a_1, a_2, \dots, a_n), n \geq 0$

Post:  $Q' = (a_1, a_2, \dots, a_n, a)$ , se  $n \geq 1$

Post:  $Q' = (a)$ , se  $n = 0$

Q' deve essere inteso come Q modificato e non come return value

## Esempio: le code

**Esercizio:** Supposto di aver a disposizione il tipo di dato **coda** scrivere un predicato che ritorna True se un dato elemento appartiene alla coda e False altrimenti.

```
int ricercacoda ( tipoelem a, coda q ) {  
    int found = 0;  
  
    // scandisco gli elementi della coda  
    while ( ! CODAVUOTA(q) && ! found ) {  
  
        if ( a == LEGGICODA(q) ) {  
            found = 1;  
        } else {  
            FUORICODA(q); // WARNING: sto svuotando la coda !!!  
        }  
    }  
  
    return found;  
}
```

L'unico modo a disposizione per scandire gli elementi di una coda consiste nell'estrarre ad uno ad uno gli elementi stessi.

## Le code: Produttore Consumatore

**Esercizio:** Scrivere due processi, uno produttore che produce elementi e li inserisce in una coda ed un'altro consumatore che attinge dalla medesima coda e consuma gli elementi:

```
void produttore ( coda q ) { // inserisce elementi random nella coda
    int a;

    a = rand();
    INCODA(a, q);
}

void consumatore ( coda q ) { // legge gli elementi dalla coda se non vuota
    int a;

    if ( ! CODAVUOTA ( q ) ) {
        a = LEGGICODA(q);
        FUORICODA(q);
        printf("a: %d \n", a);
    }
}
```

**Esercizio:** scrivere un **main** che, dichiara una coda, inizializza il generatore di numeri random e chiama le funzioni **produttore** e **consumatore** in sequenza.

## Implementazione tramite liste bidirezionale

Sfruttando il tipo di dato lista bidirezionale circolare possiamo implementare o realizzare tramite di esso il tipo di dato coda.

**Esercizio** Definire il tipo di dato coda tramite il tipo di dato lista bidirezionale circolare. La corrispondenza tra le operazioni della coda e le operazioni sulla lista sono di seguito elencate:

- ▷  $\text{CREACODA}() = \text{CREALISTA}()$
- ▷  $\text{CODAVUOTA}(Q) = \text{LISTAVUOTA}(Q)$
- ▷  $\text{LEGGICODA}(Q) = \text{LEGGILISTA}(\text{ULTIMOLISTA}(Q))$
- ▷  $\text{FUORICODA}(Q) = \text{CANCLISTA}(\text{ULTIMOLISTA}(Q))$
- ▷  $\text{INCODA}(a, Q) = \text{INSLISTA}(a, \text{PRIMOLISTA}(Q))$

**Esercizio** Definire una realizzazione in C degli operatori della coda secondo le specifiche date.

## Implementazione tramite lista monodirezionale

**Esercizio** Definire il tipo di dato coda tramite il tipo di dato lista **monodirezionale non circolare** (aka record concatenati) in modo tale che la procedura INCODA(a, Q) abbia complessità  $\mathcal{O}(1)$  .

```
typedef struct structelem * posizione;  
  
struct structelem {  
    tipoelem elemento;  
    posizione successivo;  
};  
  
struct structcoda {  
    posizione testa;  
    posizione fondo;  
};  
  
typedef struct structcoda * coda;
```

- ▷ **structelem** é un record che rappresenta un singolo elemento della lista monodirezionale
- ▷ **posizione** é un puntatore ad record di tipo **structelem**
- ▷ il tipo di dato **coda** é definito come un puntatore ad una struttura che contiene due informazioni:
  - ▷ la posizione dell'elemento di **testa**
  - ▷ la posizione dell'elementi di **fondo**

procedura **INCODA**:

```
void INCODA ( tipoelem a, coda Q ) {  
    posizione p = malloc(sizeof(struct structelem));  
    p->elemento = a;  
    p->successivo = NULL;  
  
    if ( Q->fondo != NULL ) {  
        Q->fondo->successivo = p;  
    }  
  
    if ( Q->testa == NULL ) {  
        Q->testa = p;  
    }  
  
    Q->fondo = p;  
}
```

L'implementazione di **INCODA** ha complessità  $\mathcal{O}(1)$  poiché l'accesso alla posizione dell'elemento di **fondo** è memorizzata nella struttura **structcoda** e quindi avviene in modo diretto.

## Implementazione tramite Vettore Circolare

Una implementazione **statica** delle code può essere effettuata tramite vettore circolare:

- ▷ i dati di una coda sono memorizzati in un vettore di lunghezza prefissata **MAXLUNG**
- ▷ l'indice dell'elemento di **testa** è memorizzato in una variabile cursore **testa**
- ▷ la lunghezza corrente della coda è memorizzata in un variabile **lung**
- ▷ l'indice dell'elemento di **fondo** è calcolato come  $(\text{testa} + \text{lung} - 1) \bmod \text{MAXLUNG}$

In C una coda può essere realizzata tramite vettore circolare nel seguente modo:

```
#define MAXLUNG N;

struct structcoda {
    tipoelem elementi[MAXLUNG];
    int testa;
    int lung;
}

typedef structcoda * coda
```

## Realizzazione in C tramite Vettore Circolare

```
coda CREACODA ( ) {  
    coda q;  
    q = (coda) malloc (sizeof ( struct structcoda ) );  
    q->testa = 0;  
    q->lung  = 0;  
    return q;  
}
```

```
int CODAVUOTA ( coda q ) {  
    return ( q->lung == 0 ) ? 1 : 0;  
}
```

```
tipoelem LEGGICODA ( coda q ) {  
    if ( ! CODAVUOTA(q) ) {  
        return q->elementi[q->testa];  
    } else {  
        fprintf(stderr, "ERROR coda vuota !\n");  
    }  
}
```

```
tipoelem FUORICODA ( coda q ) {  
    if ( ! CODAVUOTA(q) ) {  
        q->testa = (q->testa + 1) % MAXLUNG;  
        q->lung  = q->lung - 1;  
    }  
}
```



```
void INCODA ( tipoelem a, coda q ) {  
    if ( q->lung == MAXLUNG ) {  
        fprintf(stderr, "ERROR coda piena !\n");  
    } else {  
        q->elementi[(q->testa + q->lung) % MAXLUNG] = a;  
        q->lung = q->lung + 1;  
    }  
}
```

## Esercizio

Definire una implementazione delle code tramite vettore circolare che utilizzi un cursore **testa** ed uno **fondo**, rispettivamente al primo ed ultimo elemento, **senza mantenere in alcuna variabile la lunghezza della coda**

## Considerazioni

Abbiamo visto tre implementazioni diverse del tipo di dato coda.

Domanda:

Quale usare ?

Risposta:

dipende da caso a caso.

- ▷ vettore circolare
  - ▷ **pro**: occupa poco spazio in memoria e gli elementi sono memorizzati in modo contiguo
  - ▷ **contro**: bisogna conoscere a priori il numero massimo di elementi che la coda deve contenere
- ▷ implementazione tramite liste:
  - ▷ **pro**: é una implementazione espandibile
  - ▷ **contro**: gli elementi non sono memorizzati in celle di memoria contigue e l'occupazione di spazio é maggiore

## Esercizio: Valutazioni di Espressioni in Forma Polacca Inversa

La notazione polacca inversa é un modo **comodo** per rappresentare espressioni aritmetiche senza fare uso di parentesi che permette di rispettare la precedenza degli operatori.

Esempio: l'espressione

$$((10 \times 2) + 22) / (2 + 7)$$

viene rappresentata in notazione polacca inversa in una pila  $P$  nel seguente modo:

10	2	×	22	+	2	7	+	/
----	---	---	----	---	---	---	---	---

Per valutare una espressione in forma polacca inversa, si procede secondo il seguente algoritmo di valutazione finché la pila non contiene un solo elemento, il risultato dell'espressione:

1. si estrae dalla un elemento  $e$
2. se  $e$  é un numero lo si inserisce in un'altra pila  $Q$
3. se  $e$  é in operatore si estrae dalla pila  $Q$  due operandi  $a$  e  $b$
4. si effettua l'operazione  $e(a, b)$  e si inserisce il risultato nella pila  $P$

**Esercizio:** scrivere una procedura  $C$  che valuta una espressione in forma polacca inversa.