

# Pile e code



*Fulvio CORNO - Matteo SONZA REORDA*  
*Dip. Automatica e Informatica*  
*Politecnico di Torino*

## Sommario

---

- ADT
- Pile
- Code.



## Sommario

---

- ADT
- Pile
- Code.



## ADT

---

Le regole che sovrintendono allo sviluppo del software in ambiente professionale fanno ampiamente uso del concetto di *Abstract Data Type* (ADT).

Gli ADT sono anche utilissimi nella fase di definizione e documentazione degli algoritmi, per dare agli stessi la massima generalità.

*Un ADT è un modello matematico sul quale è definito un insieme di operazioni.*



## Esempio

---

È possibile definire un ADT corrispondente ad un generico insieme di elementi, su cui sono definite le operazioni di *unione*, *intersezione* e *differenza*:

- `union (X, Y)`
- `intersect (X, Y)`
- `diff (X, Y)`

Le tre operazioni definite dall'ADT in questo caso hanno come operandi e producono come risultato degli elementi dell'ADT.



## ADT come estensione dei tipi primitivi

---

Il concetto di ADT combina e generalizza i concetti di tipi primitivi e di procedure.

I tipi primitivi sono quelli supportati da un linguaggio (ad esempio `int`, `float`, `char` per il C).

Attraverso gli ADT è possibile estendere l'insieme dei tipi di dato supportati.



## ADT come estensione delle procedure

---

Le procedure permettono di estendere la nozione di operatore:

- Gli **operatori** sono predefiniti, e permettono di eseguire le operazioni elementari sui tipi primitivi.
- Le **procedure** permettono di definire nuove operazioni sui tipi primitivi, o di definire operazioni su nuovi tipi.

Gli ADT permettono di associare ad un nuovo tipo un insieme di operazioni.



## Implementazione di un ADT

---

Un ADT è definito in maniera indipendente da:

- linguaggio di programmazione adottato
- scelte implementative, quali ad esempio quelle relative alla struttura dati.



## Esempio

---

Nel caso dell'ADT corrispondente agli insiemi di numeri interi sono possibili varie implementazioni in C, che differiscono innanzitutto per la struttura dati utilizzata:

- Vettore
- Lista.



## Implementazione in C

---

Quando si implementa in C un ADT, conviene:

- Definire in un file separato le procedure corrispondenti alle operazioni definite dall'ADT
- Definire un tipo di dato corrispondente all'ADT
- Lavorare sull'ADT esclusivamente utilizzando le procedure fornite.



## Esempio

---

Definizione del tipo:

```
typedef SET_ELEM int;
```

Esempi di procedure:

```
SET_ELEM *union (SET_ELEM *, SET_ELEM *);  
SET_ELEM *inters (SET_ELEM *, SET_ELEM *);  
SET_ELEM *diff (SET_ELEM *, SET_ELEM *);  
SET_ELEM *make_null( void);  
int size (SET_ELEM *);  
void dump (SET_ELEM *);
```



## Sommario

---

- ADT
- Pile
- Code.



## Pile

---

Una pila è un ADT su cui sono definite le seguenti operazioni:

- **Push**: inserisce un nuovo elemento nell'ADT
- **Pop**: estrae dall'ADT un elemento; viene ritornato l'elemento inserito da meno tempo
- **Empty**: ritorna vero se l'ADT contiene 0 elementi
- **Init**: crea un ADT vuoto.



## Pila = stack = struttura LIFO

---

Una pila (anche detta *stack*) è una struttura che implementa una strategia LIFO (*Last In First Out*).



## Esempio

---

Si supponga di aver eseguito le seguenti operazioni:

```
M = init()
```

```
Push (M, K1)
```

```
Push (M, K2)
```

```
Push (M, K3)
```

Se si esegue l'operazione `Pop (M)`, questa ritorna `K3`. Eseguendo nuovamente `Pop (M)`, questa ritorna `K2`. Attivando `Empty (M)` a questo punto, essa ritorna il valore `FALSE`.



## Implementazione tramite vettore

---

Richiede:

- La definizione di un vettore di `N` elementi
- L'introduzione di una variabile `top` che contiene l'indice dell'ultimo elemento inserito.

La memoria per il vettore è allocata interamente, indipendentemente dal numero di elementi in esso memorizzati.

Tutte le operazioni sulla pila hanno complessità  $O(1)$ .





## Pseudo-codice

---

```

STACK-EMPTY(S)
1  if  $top[S] = 0$ 
2    then return TRUE
3    else return FALSE

PUSH(S, x)
1   $top[S] \leftarrow top[S] + 1$ 
2   $S[top[S]] \leftarrow x$ 

POP(S)
1  if STACK-EMPTY(S)
2    then error "underflow"
3    else  $top[S] \leftarrow top[S] - 1$ 
4    return  $S[top[S] + 1]$ 
    
```

A.A. 2001/2002

APA - Pile e code

17



## C

---

```

#define MAX 100
int buff[MAX];
int index;
void push( int val);
int pop( void);
int empty( void);
void push( int val)
{
    buff[index++] = val;
    return;
}

int pop( void)
{
    return( buff[--index]);
}
int empty( void)
{
    if( index == 0)
        return (1);
    else
        return (0);
}
    
```

A.A. 2001/2002

APA - Pile e code

18



## Sommario

---

- ADT
- Pile
- Code.




## Code

---

Una coda è un ADT su cui sono definite le seguenti operazioni:

- **Insert** (o enqueue): inserisce un nuovo elemento nell'ADT
- **Extract** (o dequeue): estrae dall'ADT un elemento; viene ritornato l'elemento inserito da più tempo
- **Empty**: ritorna vero se l'ADT contiene 0 elementi
- **Init**: crea un ADT vuoto.



## Coda = struttura FIFO

---

Una coda è una struttura che implementa una strategia FIFO (*First In First Out*).

A.A. 2001/2002

APA - Pile e code

21



## Esempio

---

Si supponga di aver eseguito le seguenti operazioni:

```
M = init()
```

```
Enqueue (M, K1)
```

```
Enqueue (M, K2)
```

```
Enqueue(M, K3)
```

Se si esegue l'operazione `Dequeue (M)`, questa ritorna K1. Eseguendo nuovamente `Dequeue (M)`, questa ritorna K2. Attivando `Empty (M)` a questo punto, essa ritorna il valore FALSE.

A.A. 2001/2002

APA - Pile e code

22



## Implementazione tramite vettore

---

Richiede:

- La definizione di un vettore di  $N+1$  elementi
- L'introduzione di due variabili head e tail:
  - **Head** contiene l'indice dell'elemento inserito da più tempo
  - **Tail** contiene l'indice dell'elemento nel vettore in cui inserire un nuovo elemento.
- Un meccanismo che trasformi il vettore in un **buffer circolare**.

La memoria per il vettore è allocata interamente, indipendentemente dal numero di elementi in esso memorizzati.

Tutte le operazioni sulla pila hanno complessità  $O(1)$ .



## Buffer circolare

---

Head e tail vengono incrementate ad ogni chiamata di dequeue ed enqueue, rispettivamente.

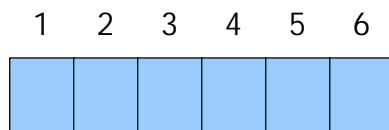
Quando head o tail giungono al valore  $n+1$ , devono venir forzate al valore 1.

All'inizio  $head = 1$ ,  $tail = 2$ .

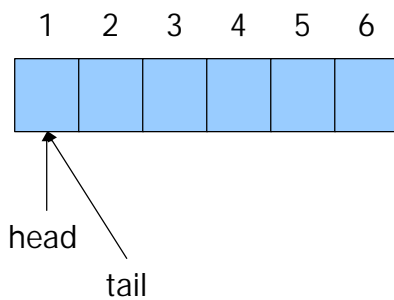
La condizione di coda piena (**overflow**) corrisponde a  $head = tail$ .

La condizione di coda vuota (**underflow**) corrisponde a  $head + 1 = tail$ .

## Esempio (n=5)

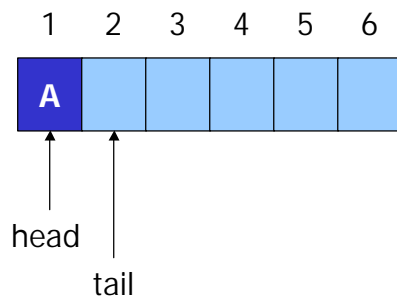


## Esempio (1)



## Esempio (2)

**Enqueue( Q, A)**



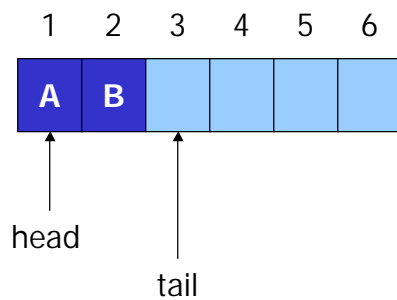
A.A. 2001/2002

APA - Pile e code

27

## Esempio (3)

**Enqueue( Q, B)**



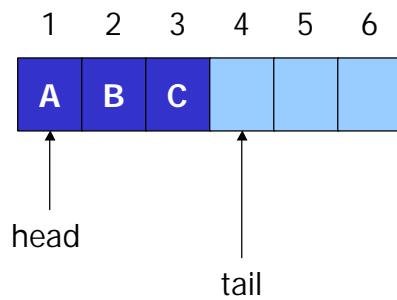
A.A. 2001/2002

APA - Pile e code

28

## Esempio (4)

**Enqueue( Q, C)**



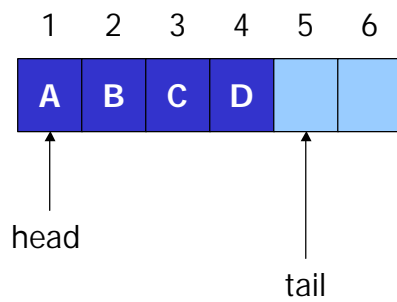
A.A. 2001/2002

APA - Pile e code

29

## Esempio (5)

**Enqueue( Q, D)**



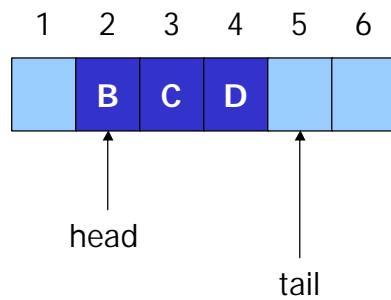
A.A. 2001/2002

APA - Pile e code

30

## Esempio (6)

**Dequeue( Q)**



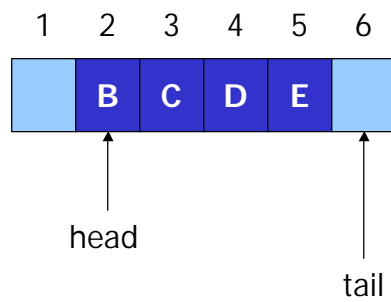
A.A. 2001/2002

APA - Pile e code

31

## Esempio (7)

**Enqueue( Q, E)**

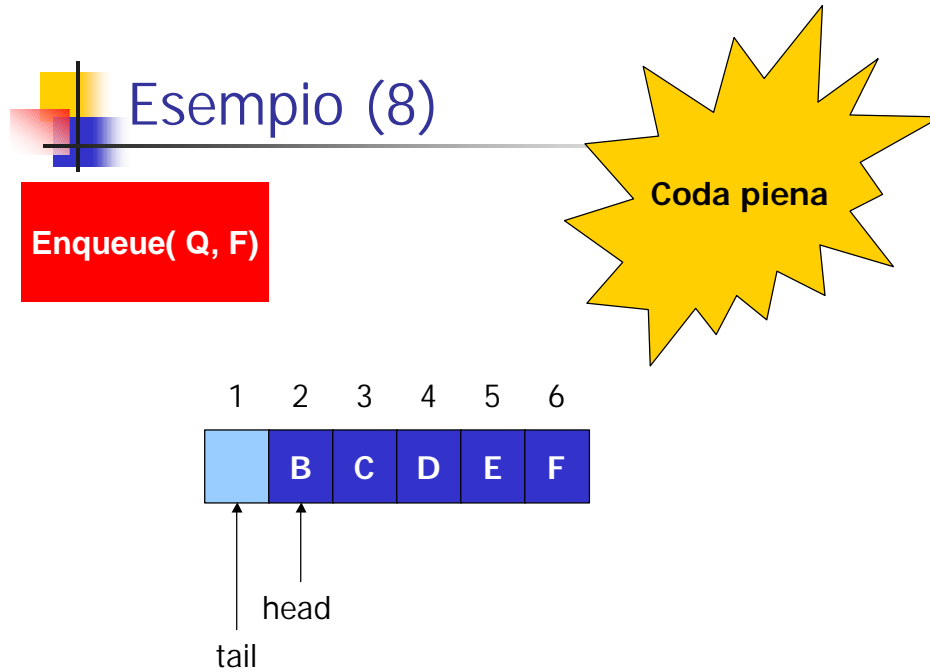


A.A. 2001/2002

APA - Pile e code

32

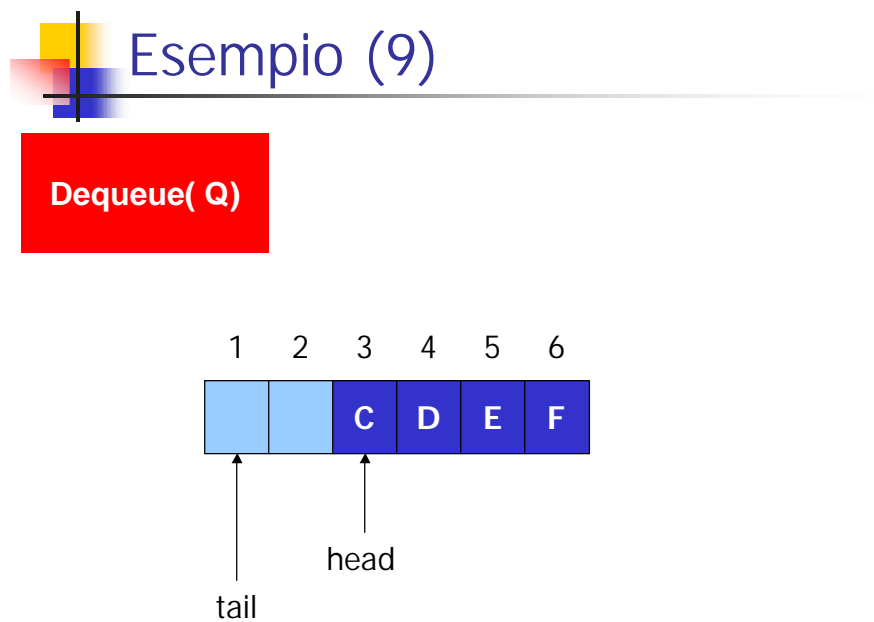




A.A. 2001/2002

APA - Pile e code

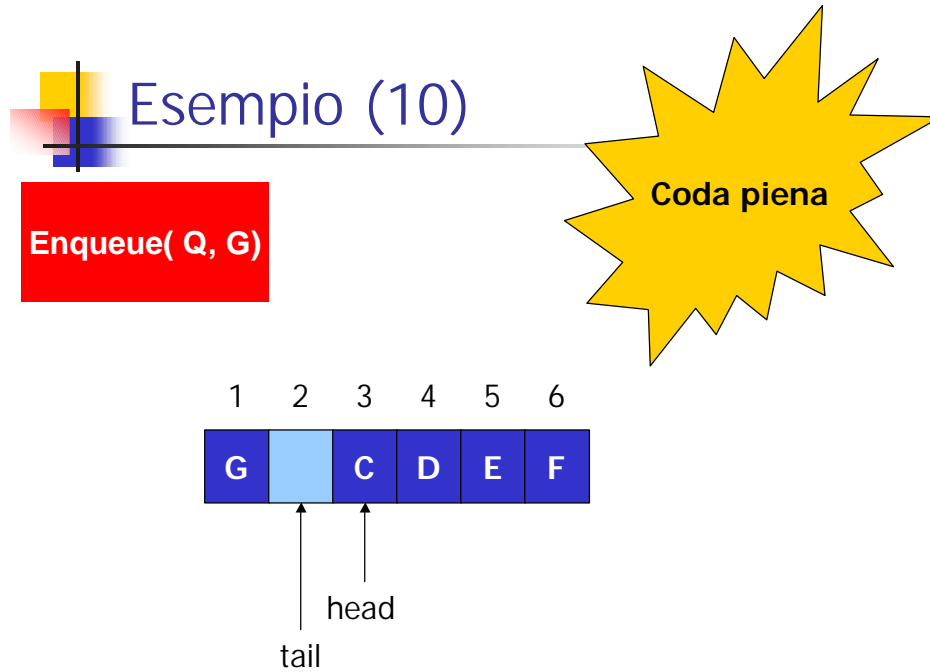
33



A.A. 2001/2002

APA - Pile e code

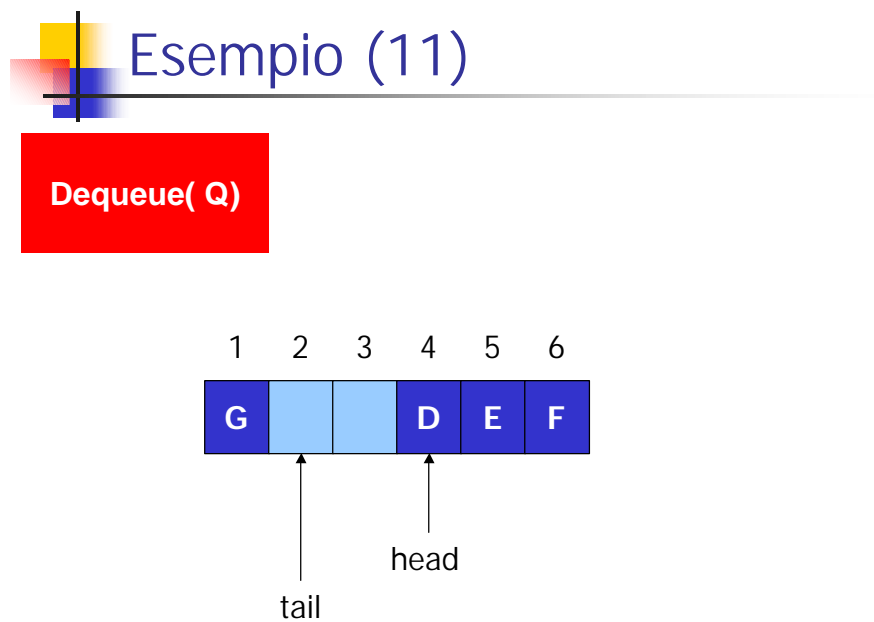
34



A.A. 2001/2002

APA - Pile e code

35



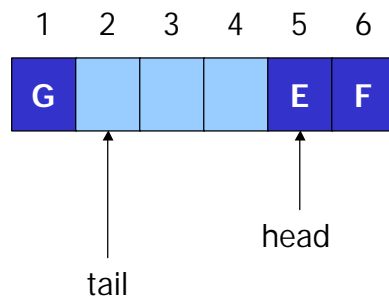
A.A. 2001/2002

APA - Pile e code

36

## Esempio (12)

**Dequeue( Q)**



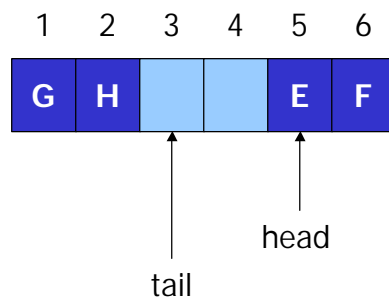
A.A. 2001/2002

APA - Pile e code

37

## Esempio (13)

**Enqueue( Q, H)**



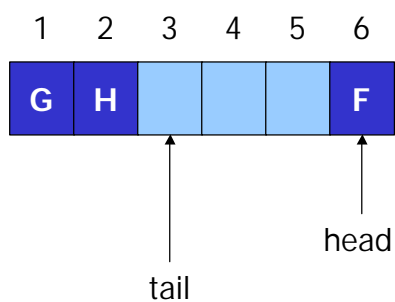
A.A. 2001/2002

APA - Pile e code

38

## Esempio (14)

**Dequeue( Q)**



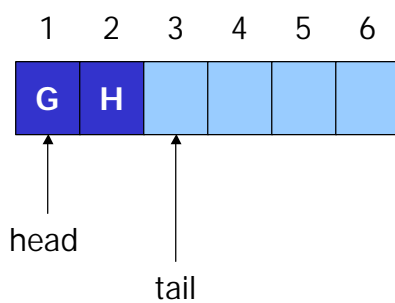
A.A. 2001/2002

APA - Pile e code

39

## Esempio (15)

**Dequeue( Q)**



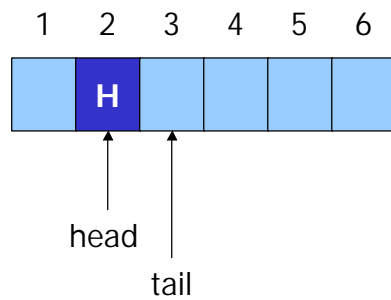
A.A. 2001/2002

APA - Pile e code

40

## Esempio (16)

**Dequeue( Q)**



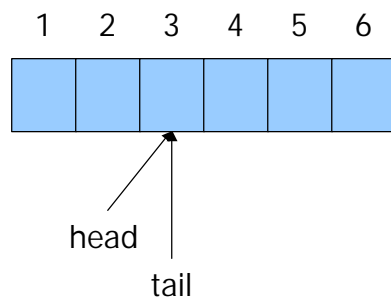
A.A. 2001/2002

APA - Pile e code

41

## Esempio (17)

**Dequeue( Q)**



A.A. 2001/2002

APA - Pile e code

42



## Pseudo-codice

```

ENQUEUE (Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3    then tail[Q] ← 1
4    else tail[Q] ← tail[Q] + 1

DEQUEUE (Q)
1  x ← Q[head [Q]]
2  if head [Q] = head [Q]
3    then head [Q] ← 1
4    else head [Q] ← head [Q] + 1
5  return x
    
```



## C

```

#define DIM 10
int  buffer[DIM+1];
int  tail=0,
     head=0;
int  insert( int elem);
int  extract( void);
int  empty( void);

int insert( int elem)
{
    if( (tail+1) < head)
        return( -1);

    buffer[tail++] = elem;
    if( tail == (DIM+1))
        tail = 0;
    return( 0);
}
    
```

 C (2)

---

```
int extract( void)
{ int ret;

  if( head == tail)
    return( -1);
  ret = buffer[out++];
  if( head == (DIM+1))
    head = 0;
  return( ret);
}

int empty( void)
{
  if( head == tail)
    return( 1);
  else
    return( 0);
}
```