Contenitori: Pile e Code

DOTT. ING. LEONARDO RIGUTINI
DIPARTIMENTO INGEGNERIA DELL'INFORMAZIONE
UNIVERSITÀ DI SIENA
VIA ROMA 56 – 53100 – SIENA
UFF. 0577234850-7102
RIGUTINI@DII.UNISI.IT
HTTP://WWW.DII.UNISI.IT/~RIGUTINI/

Strutture contenitore

- Le strutture dati astratte possono essere utilizzate per realizzare alcuni tipi di **contenitori** di utilizzo frequente, che possono forzare una particolare modalità di accesso ai dati.

- Pila o Stack: struttura dati di tipo LIFO (Last In First Out). Viene tipicamente realizzata con array o liste.

- Coda: struttura dati di tipo FIFO (First In First Out). Viene tipicamente realizzata con array o liste.

La pila

La pila

- Pila o Stack viene usato in diversi contesti per riferirsi a strutture dati le cui modalità d'accesso seguono una politica LIFO (Last In First Out):
  - i dati vengono estratti (letti) in ordine rigorosamente inverso rispetto a quello in cui sono stati inseriti (scritti).

- Il nome di questa struttura dati è infatti la stessa parola inglese usata per indicare una "pila di piatti" o una "pila di giornali", e sottende per l'appunto l'idea che quando si pone un piatto nella pila lo si metta in cima, e che quando si preleva un piatto si prelevi, analogamente, quello in cima (da cui la dinamica LIFO).

- è possibile inserire o prelevare elementi anche dalla coda, infatti più in generale la pila è un particolare tipo di lista in cui le operazioni di inserimento ed estrazione si compiono dallo stesso estremo.

La pila

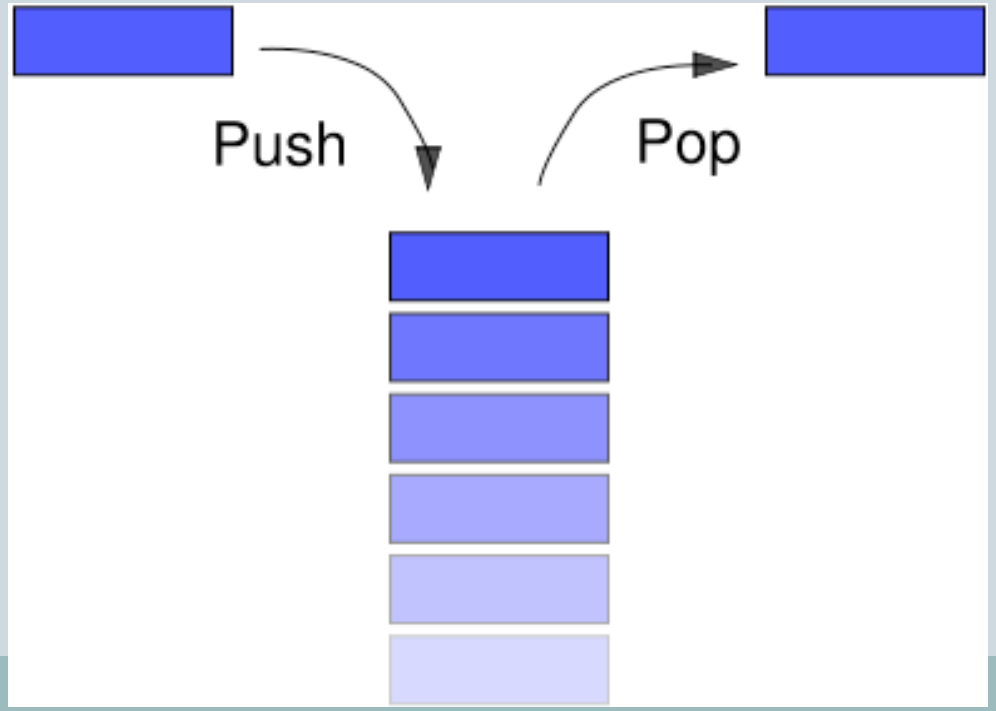
- Il termine viene usato in informatica in modo più specifico in diversi contesti:
  - la struttura dati a stack è un tipo di struttura dati che un programma può implementare e utilizzare per il proprio funzionamento;
  - lo stack è un elemento dell'architettura dei moderni processori, e fornisce il supporto fondamentale per l'implementazione del concetto di subroutine (vedi call stack);
  - le macchine virtuali di quasi tutti i linguaggi di programmazione ad alto livello usano uno stack dei record di attivazione per implementare il concetto di subroutine (generalmente, ma non necessariamente, basandosi sullo stack del processore);

La pila

- Le operazioni tipiche di una pila sono:
  - push, pop, top, isEmpty e isFull
- push – inserimento di un elemento in cima alla pila: push(elemento)
- Pop – rimuove l'oggetto in cima alla pila dalla pila stessa;
- top – restituisce l'elemento in cima alla pila senza però rimuoverlo dalla pila: una successiva operazione di top o di pop restituisce ancora l'elemento letto.

La pila

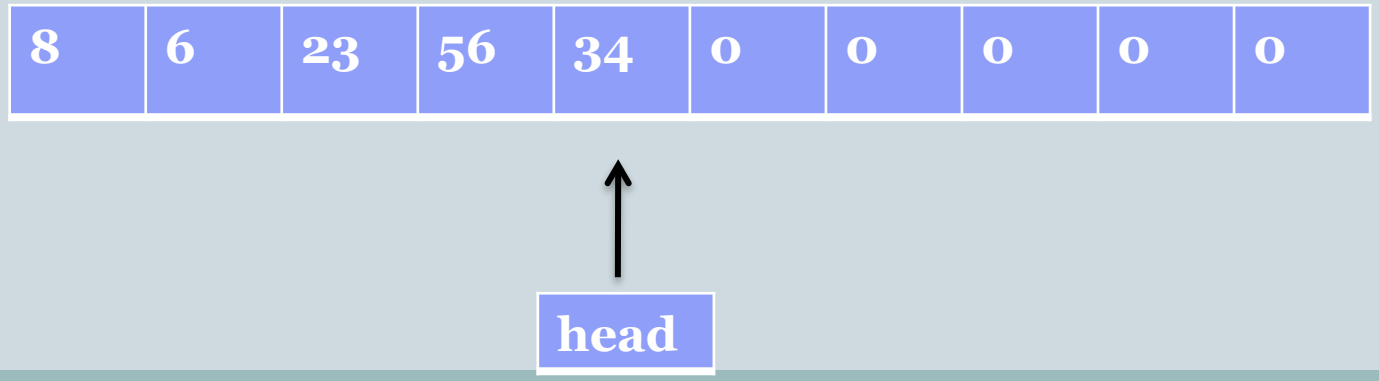
- isEmpty – restituisce **true** se la pila è vuota;
- isFull – restituisce **true** se la pila è piena (se l'implementazione specifica una dimensione massima)



Implementazione con array

- Per memorizzare gli elementi della pila viene utilizzato un array: questo implica una dimensione massima specificata dalla dimensione dell'array.

- Viene utilizzato un puntatore all'ultimo elemento (o alla prima cella libera): nel caso di push viene aumentato di 1, mentre nel caso di pop viene decrementato di 1.



Implementazione con array

```
typedef  struct  {
   int  capacity;
   int  head;
   int[]  buffer;
   }  pila;

pila  *  createPila(int  capacity) {
   pila * p=(pila*)malloc(sizeof(pila));
   p->buffer = int[capacity];
   p->capacity = capacity;
   p->head = -1;
   return  p;
}
```

isEmpty & isFull

```
bool  isEmpty(pila  *  p) {
    if (p==NULL) return true;
    return (p->head== -1);
}

bool  isFull(pila  *  p) {
    if  (p==NULL) return false;
    return (p->head==p->capacity-1);
}
```

push()

```
bool  push(int  valore, pila  *  p) {
    if (isFull(p)) {
            return false;
    }
    p->head++;
    p->buffer[p->head]=valore;
    return true;
}
```

top()

```
int  top(pila  *  p) {
    if (isEmpty(p)) {
            return ERROR;
    }
    return p->buffer[p->head];
}
```

- Dato che i valori memorizzati nella pila sono interi, è necessario scegliere un valore particolare per indicare che la pila è vuota e che il risultato di top() non è valido: ERROR.

- Ad esempio potrebbe essere ERROR=-1 o ERROR=0

pop()

```
int  pop(pila  *  p) {
    if (isEmpty(p)) {
            return ERROR;
    }
    p->head--;
    return p->buffer[p->head+1];
}
```

- Stesso ragionamento per quanto riguarda ERROR.

Implementazione con Lista

- L'implementazione con array pone un limite sulla capacità della pila. Per realizzare una pila con capacità"illimitata" si utilizza una lista.

- La pila è realizzata come una lista essa stessa, vincolando le operazioni di inserimento (push) e rimozione (pop).

- L'elemento della pila è definito come l'elemento di una lista:

```
typedef  struct  {
        int  value;
        struct  elemento *  next;
} elemento;
```

push

- Il push coincide con l'inserimento in testa di una lista: in questo modo l'ultimo elemento inserito è la testa della lista.

- La funzione in questo caso restituisce un valore, è cioè la nuova pila con in testa l'elemento appena inserito:

```
elemento * push(int  valore, elemento  * pila){
        elemento new_el=(elemento*)malloc(sizeof(elemento));
        new_el->value=valore;
        new_el->next=pila;
        return new_el;
}
```

top

- La funzione top semplicemente restituisce il valore memorizzato in head senza rimuoverlo:

```
int  top(elemento  *  pila) {
        if (pila==NULL) {
                return  ERRORE;
        }
        return pila->value;
}
```

pop

- La funzione pop è implementata dalla funzione rimozione in testa.
- In questo caso il risultato è memorizzato nella variabile **int** risultato passata per riferimento ed è restituito il puntatore alla pila modificata.

```
elemento * pop(elemento  *  p) {
    if (p==NULL) {
            return NULL;
    }
    elemento  *  n=p->next;
    free(p);
    return  n;
}
```

La coda

La coda (Queue)

- In Informatica per coda si intende una struttura dati di tipo FIFO, First In First Out (il primo in ingresso è il primo ad uscire).

- Un esempio pratico sono le code che in un paese civile si fanno per ottenere un servizio, come pagare al supermercato o farsi tagliare i capelli dal parrucchiere:

  idealmente si viene serviti nello stesso ordine con cui ci si è presentati.

- Questo tipo di struttura dati è molto utilizzata in Informatica, ad esempio nella gestione delle operazioni da eseguire da parte di un sistema operativo, ed è fondamentale nelle telecomunicazioni, in particolare nelle reti a commutazione di pacchetto, dove descrive la gestione dei pacchetti in attesa di essere trasmessi su un collegamento.

Le code

- Le operazioni su una coda sono: enqueue, front, dequeue, isEmpty, isFull

  - enqueue – accodamento di un elemento, serve a mettere un elemento in coda.
  - dequeue – rimozione dell'elemento in testa alla coda.
  - front – legge l'elemento in testa alla coda senza rimuoverlo.
  - isEmpty – restituisce **true** se la coda è vuota.
  - IsFull – restituisce **true** se la pila è piena (nel caso l'implementazione preveda una capacità massima)

Realizzazione con array

- La realizzazione con array prevede:
  - l'uso di un array per memorizzare gli elementi nella coda;
  - Un **int** che memorizza la dimensione massima dell'array
  - Un indice che individua l'elemento attualmente in testa all'array
  - Un indice che individua l'elemento attualmente in coda alla coda

Implementazione con array

```
typedef struct {
    int [] buffer;
    int capacity;
    int head;
    int tail;
} coda;

coda * creaCoda(int capacity) {
    coda * c=(coda*)malloc(sizeof(coda));
    c->buffer=int[capacity];
    c->capacity=capacity;
    c->head=-1;
    c->tail=-1;
    return c;
}
```

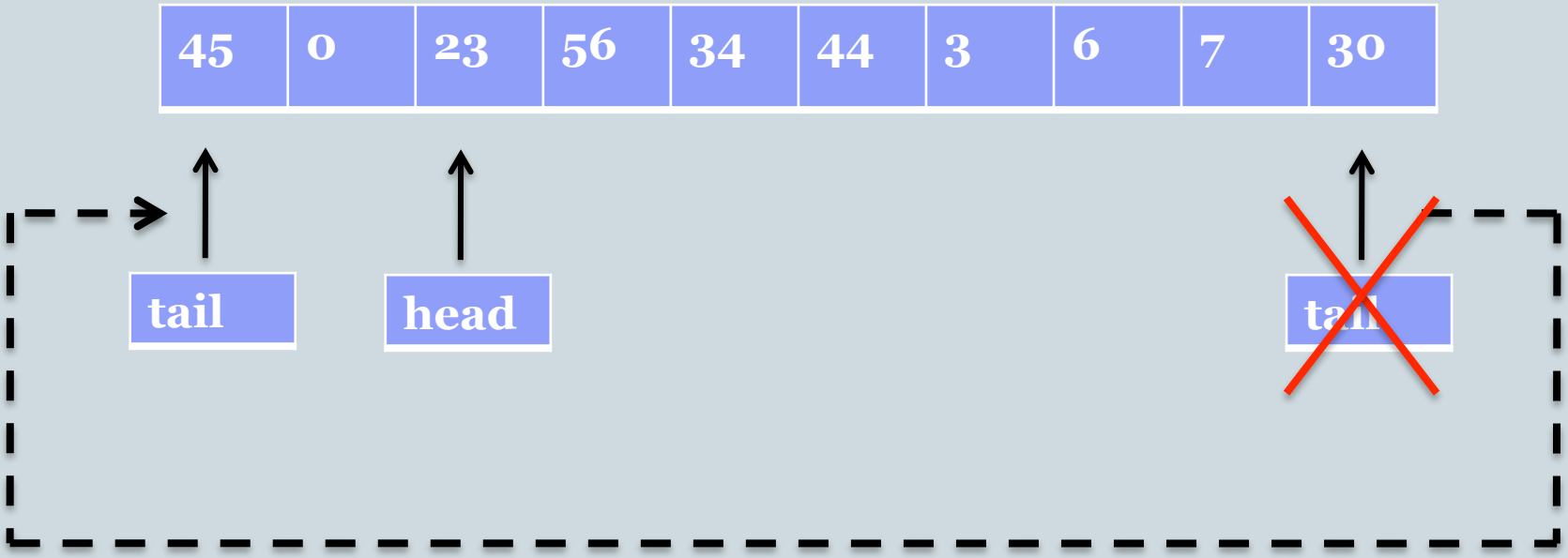
enqueue

- L'inserimento avviene spostando avanti di 1 l'indicatore della cella di coda (tail).

```
bool  enqueue(int  valore, coda *  c) {
    if (isFull(c)) return false;
    c->tail=(c->tail+1)%c->capacity;
    c->buffer[c->tail]=valore;
    if (c->head== -1)  {
            c->head=c->tail;
    }
    return true;
}
```

Realizzazione con array circolare

- Una volta che tail raggiunge l'ultima cella, la coda diventa inutilizzabile. E' necessario far ripartire tail dalla posizione 0 (solo se non è occupata da head): tail=(tail+1)%N dove % indica la divisione modulo N e N è la capacità dell'array



isFull

- La condizione di coda piena è quindi che aumentando di 1 tail (modulo N), esso non si vada a sovrapporre a head: in tal caso ho coda piena.

```
bool  isFull(coda *  c) {
   if (c==NULL) return false;
   if (isEmpty(c)) return false;
   return (c->head==((c->tail+1)%c->capacity));
}
```

Front

- front restituisce l'elemento in testa alla coda, ovvero quello che viene rimosso se è fatta una operazione di dequeue.

```
int  front(coda  *  c) {
   if (isEmpty(c)) {
         return  ERROR;
   }
   return c->buffer[c->head];
}
```

dequeue

- Estrarre un elemento dalla coda vuol dire spostare avanti di 1 head.
- Se dopo questa operazione head > tail , la coda è diventata vuota e sia head che tail sono riportati a -1.

dequeue

```
bool  dequeue(coda  *  c) {
   if  (isEmpty(c)) {
         return false;
   }
   c->head=(c->head+1)%c->capacity;
   if (c->head==(c->tail+1)%c->capacity) {
         c->head = -1;
         c->tail = -1;
   }
   return true;
}
```

isEmpty()

- Da come è stato definita l'operazione di dequeue, la verifica se la coda è vuota è fatta semplicemente verificando se head=-1

```
bool  isEmpty(coda  * c) {
   return  (c->head== -1);
}
```

Realizzazione mediante lista

- La realizzazione mediante lista implementa una coda attraverso una lista puntata che memorizza sia l'elemento head che l'elemento tail.

```
typedef  struct  {
   int  value;
   elemento  *  head;
   elemento *   tail;
}  coda;
```

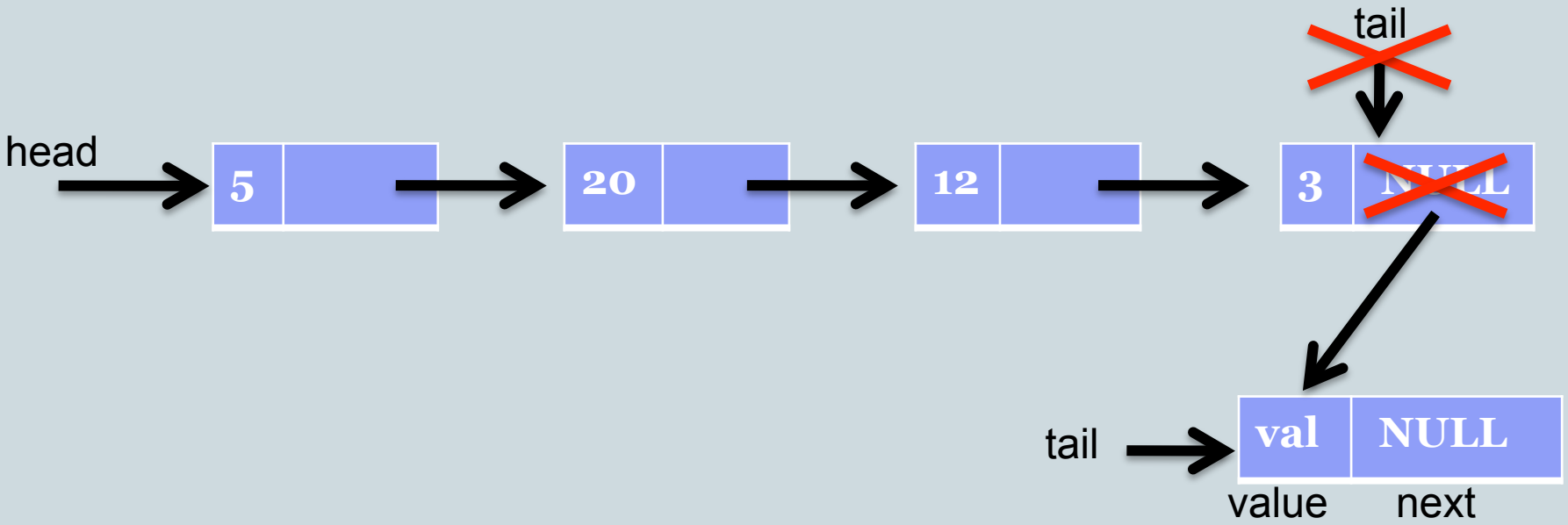
isEmpty

- Per verificare se la coda è vuota basta verificare se head==NULL

```
bool  isEmpty(coda  *  c) {
   if  (c==NULL) {
         return true;
   }
   return  (c->head==NULL);
}
```

enqueue

- Tenendo memorizzato il puntatore all'ultimo elemento, l'inserimento in coda ha costo O(1):



enqueue

```
void   enqueue(int  valore, coda  *  c) {
    elemento  *  new_el=(elemento*)malloc(sizeof(elemento));
    new_el->value=valore;
    new_el->next=NULL;
    if (isEmpty(c)) {
            c->head=new_el;
            c->tail=new_el;
    } else {
            c->tail->next=new_el;
            c->tail=new_el;
    }
}
```

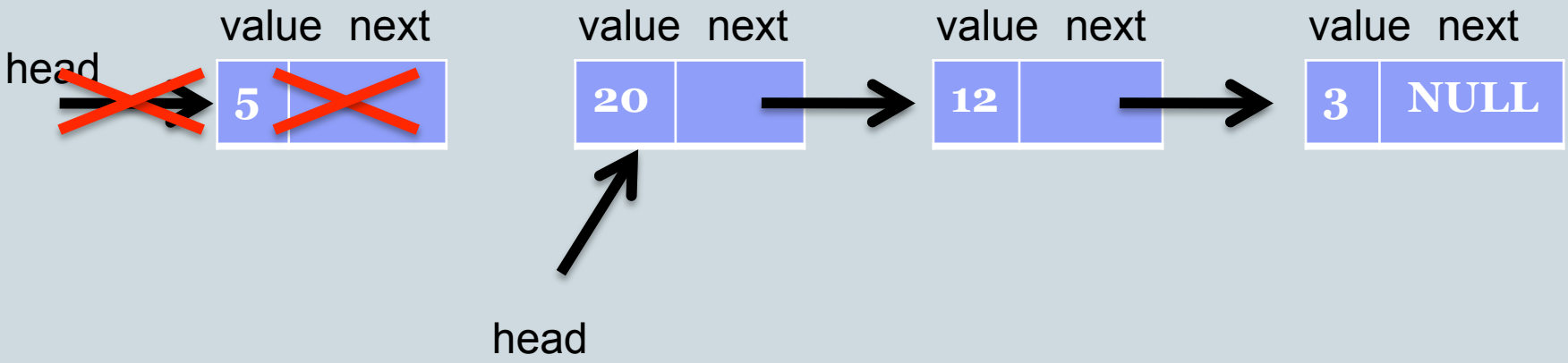
front

- La lettura dell'elemento in cima alla coda è semplicemente la lettura dell'elemento puntato da head

```
int  front(coda  *  c) {
   if  (isEmpty(c)) {
         return ERRORE;
   }
   return c->head->value;
}
```

dequeue

- La funzione dequeue avviene tramite la funzione di rimozione in testa.



dequeue

```
void  dequeue(coda  *  c) {
    if (isEmpty(c)) return;
    elemento * t=c->head;
    c->head=c->head->next;
    free(t);
    if (c->head==NULL) {
            c->tail==NULL;
    }
}
```