Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

La struttura dati CODA

Andrea Piroddi - andrea.piroddi@unibo.it
Credit: Pietro Di Lena

Introduzione

- Una **coda** (o queue) è una **struttura dati astratta** le cui modalità di accesso sono di tipo **FIFO**.
  - FIFO (First In First Out): i dati sono estratti rispetto all'ordine di inserimento.
  - LIFO (Last In First Out): i dati sono estratti in ordine inverso rispetto al loro inserimento.
- Una struttura dati coda supporta essenzialmente due sole operazioni.
  1. **Enqueue**. Inserisce un oggetto in coda.
  2. **Dequeue**. Rimuove il primo oggetto in coda e ne ritorna il valore.
- La politica FIFO è comunemente adottata in numerosi contesti differenti.
  - Nella vita reale, è utilizzata in tutti (o quasi) i contesti in cui è necessario attendere per poter ottenere un servizio (coda al supermercato, allo sportello bancario, ecc).
  - In informatica, è utilizzata per la gestione dei processi su un sistema operativo, per la gestione del flusso di dati tra periferiche, ecc.
- Così come abbiamo visto per le pile, una struttura dati di tipo coda può essere vista come una struttura dati di tipo lista che supporta un numero limitato di operazioni.
  - Le operazioni `Enqueue` e `Dequeue` sono equivalenti ad inserimento in coda (o, equivalentemente, testa) e rimozione in testa (o in coda) della lista.

Implementazione del tipo di dato astratto coda in C

- Vediamo alcuni approcci per implementare una struttura dati di tipo coda in C.
  - Implementazione di code con capacità *illimitata*. Mostriamo come implementare una struttura coda utilizzando una libreria che implementa una struttura dati di tipo lista. Mostriamo una ulteriore implementazione ad-hoc tramite liste concatenate.
  - Implementazione efficiente di code con capacità *limitata*. Mostriamo una implementazione ad-hoc tramite array.
- Anche in questo caso ci concentriamo unicamente su implementazioni di code che ci permettano di memorizzare interi (tipo di dato `int`).
- Scelte progettuali generali (le stesse adottate per la struttura dati lista e pila).
  - Vogliamo che il nostro codice sia **efficiente**: dobbiamo fornire l'implementazione più efficiente in termini di **utilizzo di memoria** e **tempo di calcolo** (queste caratteristiche dipendono da come abbiamo definito il tipo di dato lista).
  - Vogliamo che il nostro codice sia **modulare**: le funzioni devono essere **semplici** (poche righe di codice), **specifiche** (un solo compito) e **riutilizzabili** (vogliamo poter riutilizzare una funzione in tutti i contesti in cui sia possibile).

Implementazioni delle operazioni sulla struttura dati coda di interi

- Consideriamo i seguenti prototipi, che definiscono le operazioni richieste su una coda.
- Aggiungiamo altre funzioni utili oltre alle due essenziali `enqueue()` e `dequeue()`.
- I prototipi prendono in input un puntatore al tipo di dato queue (da definire).
- Le operazioni su una coda sono essenzialmente le stesse viste per le pile: i prototipi sono *equivalenti* a quelli dichiarati per la struttura dati pila.
  - Unica differenza: in una coda l'inserimento è effettuato su un estremo della lista sottostante, mentre la rimozione sull'estremo opposto. In una pila le due operazioni sono effettuate sullo stesso estremo della lista sottostante.

```
1  queue init();
2
3  void clear(queue *Q);
4
5  int isempty(queue *Q);
6
7  int enqueue(queue *Q, int elem);
8
9  int dequeue(queue *Q, int *elem);
10
11 int front(queue *Q, int *elem);
```

Operazioni sulla struttura dati coda: creazione, distruzione e test

- Le seguenti funzioni ci permettono di:
  - *creare* un oggetto di tipo coda. Inizialmente la coda non contiene elementi,
  - *svuotare* l'intero contenuto di un oggetto di tipo coda.
  - *testare* se la coda è vuota.

```
1  /*
2   * Crea e ritorna una coda vuota.
3   */
4  queue init();
5
6  /*
7   * Elimina tutti gli elementi contenuti nella coda
8   * puntata da Q.
9   */
10 void  clear(queue *Q);
11
12 /*
13  * Ritorna 0 se la coda Ã¨ vuota,
14  * un intero diverso da zero altrimenti.
15  */
16 int isempty(queue *Q);
```

Operazioni sulla struttura dati coda: `enqueue()` e `dequeue()`

- Le seguenti funzioni permettono di inserire e rimuovere un elemento nella coda.
- La funzione `dequeue()` restituisce il valore dell'elemento in testa alla coda, oltre a rimuoverlo.
- La funzione `front()` ci permette solo di recuperare il valore dell'elemento in testa.

```
1  /*
2   * Inserisce elem in coda alla coda Q.
3   *
4   * Ritorna 0 se l'inserimento ha successo,
5   * un intero diverso da zero altrimenti.
6   */
7  int enqueue(queue *Q, int elem);
8  /*
9   * Rimuove l'elemento in testa alla coda Q e ne salva
10  * il valore in elem.
11  *
12  * Ritorna 0 se la rimozione ha successo,
13  * un intero diverso da zero altrimenti.
14  */
15 int dequeue(queue *Q, int *elem);
16 /*
17  * Salva in elem il valore in testa alla coda Q.
18  *
19  * Ritorna 0 se l'operazione ha successo,
20  * un intero diverso da zero altrimenti.
21  */
22 int front(queue *Q, int *elem);
```

Il tipo di dato *coda illimitata*

- Esattamente come abbiamo visto per le pile, una struttura dati di tipo pila con capienza illimitata può essere implementata tramite una lista concatenata.
  - L'operazione di rimozione in testa è computazionalmente efficiente per tutte le rappresentazioni concatenate viste (lista concatenata semplice, doppiamente concatenata e circolare).
  - L'operazioni di inserimento in coda è computazionalmente efficiente unicamente nelle liste circolari.
  - Le liste circolari ci permettono quindi di implementare in modo efficiente una struttura dati di tipo coda.
- Come visto per le pile, è necessario includere il file di header in cui è definito il tipo di dato `list` per poter definire il tipo di dato queue.

```
1 #include "list.h"
2
3 typedef list queue;
```

- Il tipo di dato coda (queue) è equivalente al tipo di dato lista (`list`).
- Utilizziamo unicamente le funzioni della libreria `list.h` per poter implementare le funzioni sulla struttura dati pila: possiamo completamente tralasciare i dettagli implementativi della libreria `list.h`.

Creazione, distruzione e test su pila vuota: `init()`, `clear()` e `isempty()`

- Le seguenti funzioni permettono di creare una coda, di svuotare completamente la coda e di verificare se una coda è vuota.
- Da notare che abbiamo definito il tipo di dato `queue` come sinonimo del tipo di dato `list`: il tipo di ritorno della funzione `list_create()` è quindi perfettamente equivalente al tipo di ritorno della funzione `init()`.
- Le implementazioni di queste funzioni coincidono perfettamente con le implementazioni viste per il tipo di dato pila.

```
1 queue init(void) {
2   return list_create();
3 }
```

```
1 void clear(queue *Q) {
2   list_delete(Q);
3 }
```

```
1 int isempty(queue *Q) {
2   return is_empty(Q);
3 }
```

Inserimento, rimozione e selezione: `enqueue()`, `dequeue()`, `front()`

- La funzione `enqueue()` è implementata come funzione di inserimento in coda.
- La funzione `dequeue()` è implementata come funzione di rimozione in testa.
- Le implementazioni di `dequeue()` e `front()` coincidono perfettamente con le implementazioni di `pop()` e `top()`, rispettivamente, viste per le pile.

```
1 int enqueue(queue *Q, int elem) {
2   return tail_insert(Q,elem);
3 }
```

```
1 int dequeue(queue *Q, int *elem) {
2   if(is_empty(Q) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = head_select(Q);
6     return  head_delete(Q);
7   }
8 }
```

```
1 int front(queue *Q, int *elem) {
2   if(is_empty(Q) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = head_select(Q);
6     return 0;
7   }
8 }
```

Nuovo approccio per struttura dati di tipo *coda illimitata*

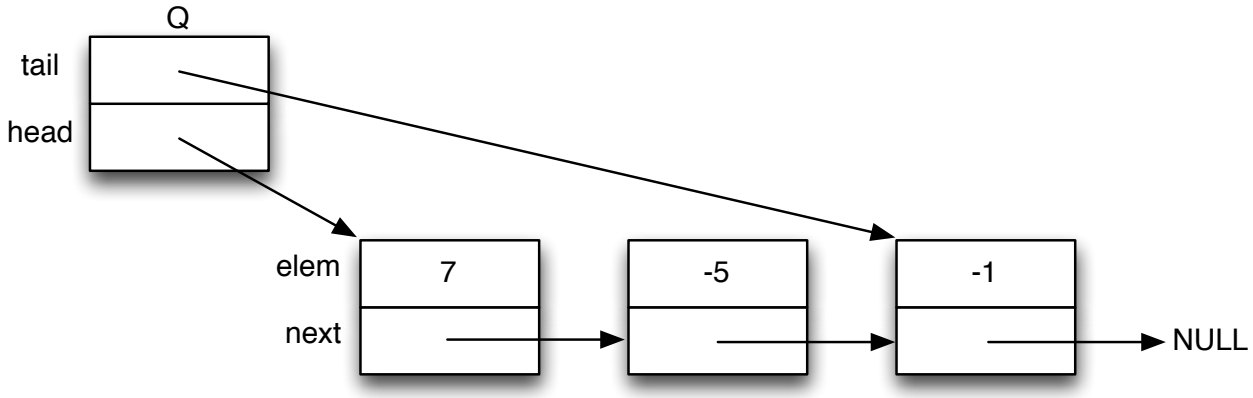
- Una struttura dati di tipo coda richiede che siano implementate in modo efficiente due sole operazioni.
  - l'operazione di inserimento in un estremo della lista sottostante (`enqueue()`).
  - l'operazione di rimozione nell'estremo opposto della lista (`dequeue()`).
- La rappresentazione tramite liste circolari doppiamente concatenate permette di effettuare efficientemente queste due operazioni ma ha come svantaggio l'overhead di memoria legato al puntatore al nodo precedente.
  - In una struttura dati coda basilare non abbiamo la necessità di attraversare la struttura dati dalla coda verso la testa: il doppio puntatore usato nelle liste circolari è superfluo, ci aiuta solo nel semplificare l'implementazione.
  - Le liste concatenate semplici *sprecano* meno memoria ma non ci permettono di accedere velocemente alla coda della lista.
- Riusciamo a trovare un approccio ad-hoc che sia efficiente sia in termini di tempo di calcolo che di utilizzo di memoria?

Nuovo approccio per struttura dati di tipo *coda illimitata*

- Possiamo migliorare la nostra implementazione di code con capienza illimitata in diversi possibili modi.
- Vediamo due possibilità.
  1. Implementiamo la lista circolare utilizzando una lista concatenata semplice.
     - Un nodo nella lista circolare punta al nodo successivo. E' necessario mantenere un puntatore all'ultimo nodo. L'implementazione potrebbe essere leggermente complessa.
  2. Cambiamo approccio e definiamo un nuovo tipo di dato: il nuovo tipo di dato coda è una struttura che contiene un puntatore alla testa ed uno (distinto) alla coda di una lista concatenata semplice.
- Mostriamo l'implementazione della struttura dati coda per questo secondo approccio.

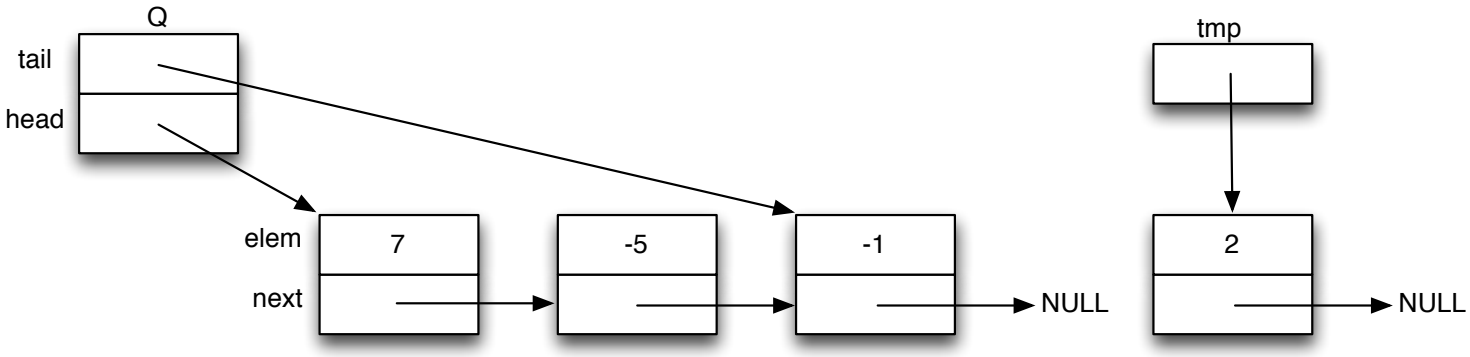
Il tipo di dato *coda illimitata*

- Il tipo di dato queue è una struttura che contiene due campi `head` e `tail` di tipo puntatore ad una struttura `node`.
- La struttura `node` ci permette di gestire una lista concatenata semplice.

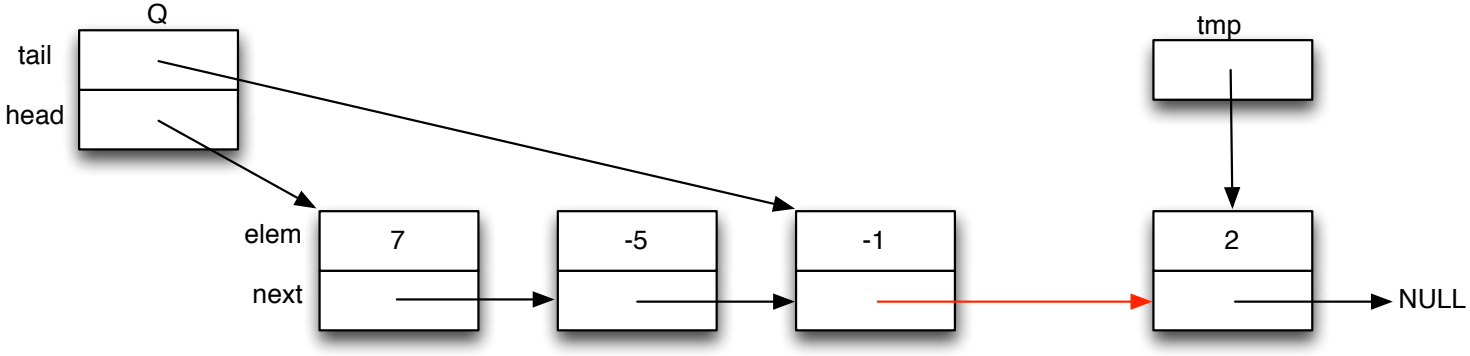



```
1 struct node {
2   int elem;
3   struct node *next;
4 };
5
6 typedef struct queue {
7   struct node *head;
8   struct node *tail;
9 } queue;
```

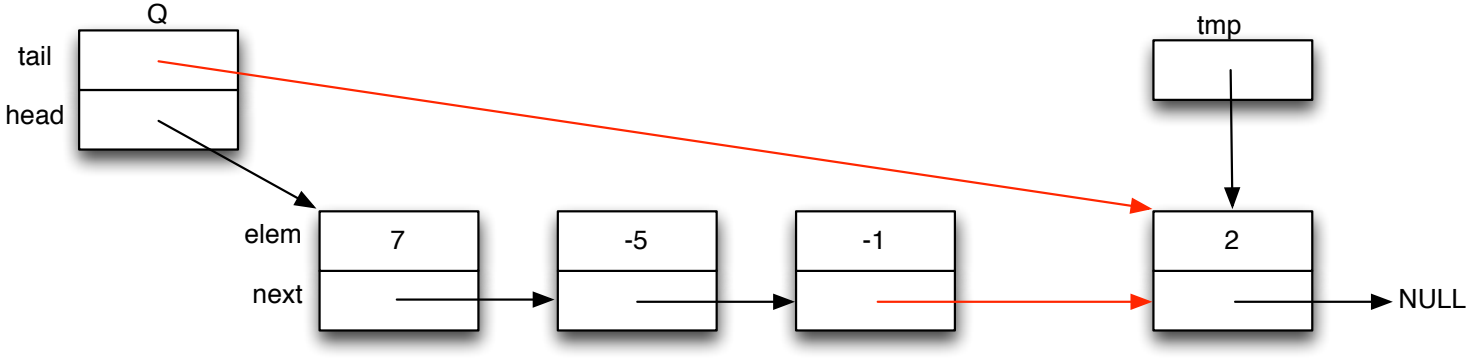
Inserimento di un elemento nella coda



A) Creazione del nuovo nodo da inserire (`tmp`)

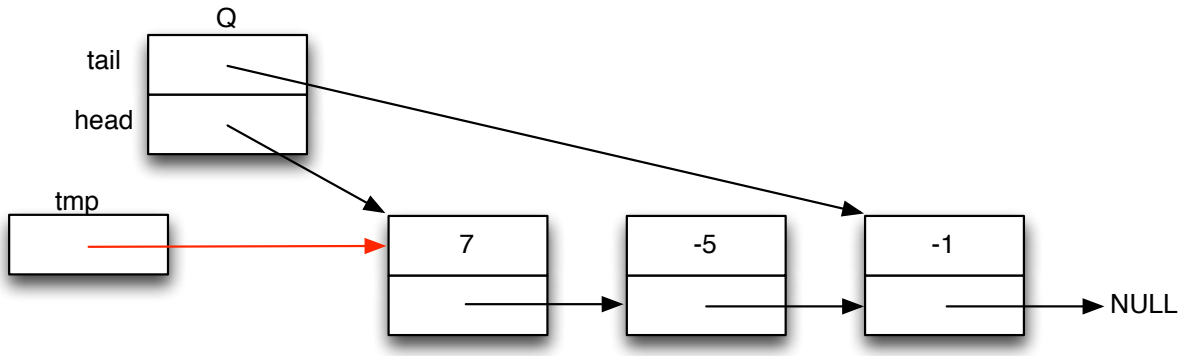


B) Aggiornamento del campo `next` del nodo in coda

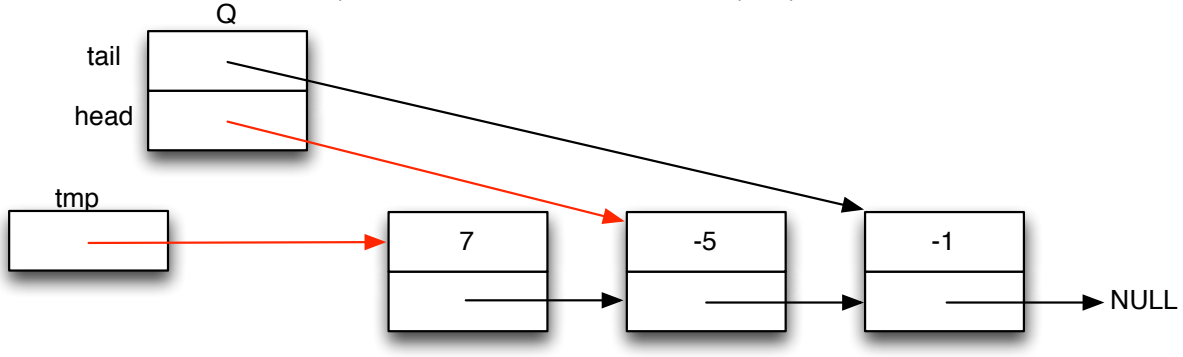


C) Aggiornamento del campo `tail` della struttura coda

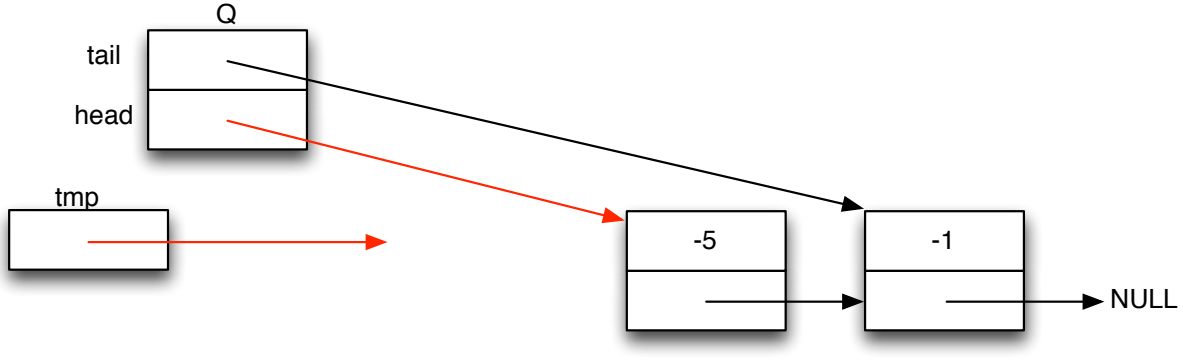
Rimozione di un elemento dalla coda



A) Puntatore al nodo da rimuovere (`tmp`)



B) Aggiornamento puntatore alla testa della coda (`head`)



C) Deallocazione del nodo da rimuovere

Creazione, distruzione e test: `init()`, `clear()`, `isempty()`

- Le seguenti funzioni permettono di creare una coda, di svuotare completamente una coda e di verificare se una coda è vuota.
- L'implementazione della funzione di distruzione di una coda (`clear()`), rimuove iterativamente l'elemento in testa.

```
1  queue init(void) {
2    queue  tmp = {NULL,NULL};
3    return tmp;
4  }
```

```
1  void clear(queue *Q) {
2    if(Q != NULL) {
3      struct node *tmp;
4
5      while((tmp = Q->head) != NULL) {
6        Q->head = Q->head->next;
7        free(tmp);
8      }
9      Q->head = Q->tail = NULL;
10   }
11 }
```

```
1  int isempty(queue *Q) {
2    return Q == NULL || Q->head == NULL;
3  }
```

Inserimento di un elemento nella coda: enqueue()

- L'operazione di inserimento di un elemento nella struttura dati coda viene gestita come un inserimento in coda ad una lista.
- La funzione utility `node_alloc()` è definita come abbiamo visto per liste concatenate.
- E' necessario gestire in modo specifico due casi distinti:
  1. La coda inizialmente è vuota (riga 4): sia il puntatore alla testa che alla coda puntano al nuovo nodo allocato (riga 5).
  2. La coda contiene almeno un elemento (blocco else a riga 6): l'operazione di inserimento è gestito come inserimento in coda in una lista concatenata. Non è necessario modificare il puntatore alla testa della coda.

```
1  int enqueue(queue *Q, int elem) {
2    if(Q == NULL) {
3      return 1;
4    } else if(isempty(Q)) {
5      return (Q->head = Q->tail = node_alloc(elem)) == NULL;
6    } else {
7      struct node *tmp = node_alloc(elem);
8
9      if(tmp != NULL) {
10       Q->tail->next = tmp;
11       Q->tail       = tmp;
12     }
13     return tmp == NULL;
14   }
15 }
```

Rimozione di un elemento nella coda: enqueue()

- L'operazione di rimozione di un elemento nella struttura dati coda viene gestito come rimozione della testa di una lista.
- E' necessario gestire in modo specifico tre casi distinti:
  1. La coda è vuota (riga 2): la funzione ritorna 1 (operazione non eseguita).
  2. La coda contiene un solo elemento (riga 4): è necessario aggiornare sia il puntatore alla testa che alla coda della lista (riga 7).
  3. La coda contiene almeno due elementi (blocco else a riga 9): l'operazione è gestita come rimozione della testa in una lista concatenata. Non è necessario modificare il puntatore alla coda della lista.

```
1  int dequeue(queue *Q, int *elem) {
2    if(isempty(Q) || elem == NULL) {
3      return 1;
4    } else if(Q->head == Q->tail) {
5      *elem = Q->head->elem;
6      free(Q->head);
7      Q->head = Q->tail = NULL;
8      return 0;
9    } else {
10     struct node *tmp = Q->head;
11     *elem = Q->head->elem;
12     Q->head = Q->head->next;
13     free(tmp);
14     return 0;
15   }
16 }
```

Accesso al primo elemento in coda: `front()`

- L'operazione di accesso al primo elemento in coda è gestita come l'operazione di selezione dell'elemento in testa in una lista.

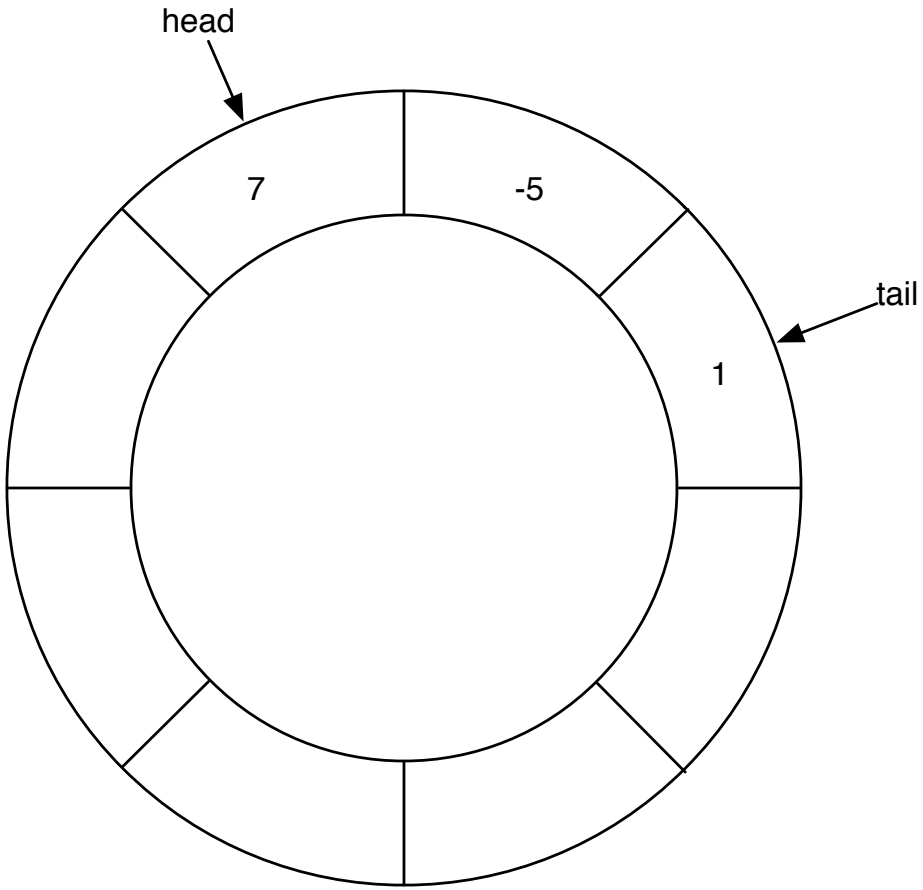
```
1 int front(queue *Q, int *elem) {
2   if(isempty(Q) || elem == NULL) {
3     return 1;
4   } else {
5     *elem = Q->head->elem;
6     return 0;
7   }
8 }
```

Caratteristiche della libreria per code limitate

- Come abbiamo visto per le pile, per alcuni problemi algoritmici che fanno uso di una struttura dati di tipo coda il numero massimo di oggetti da memorizzare può essere determinato a priori.
  - Anche in questo caso possiamo fornire una implementazione efficiente tramite rappresentazione della coda con array.
- A differenza delle pile limitate, in cui le operazioni di inserimento e rimozione sono implementate come operazioni di inserimento e rimozione in coda all'array, la rappresentazione di code limitate tramite array richiede una gestione *speciale* dello spazio di memoria a disposizione.
  - Una serie di inserimenti e rimozioni *consuma* spazio in coda e *libera* spazio in testa nell'array.
  - Come possiamo sfruttare interamente lo spazio a disposizione nell'array dopo una serie di inserimenti e rimozioni?
- L'approccio classico per l'implementazione di una struttura dati coda con capienza limitata è tramite **array circolare**.
  - L'array non è *fisicamente* circolare ma viene gestito come se lo fosse.

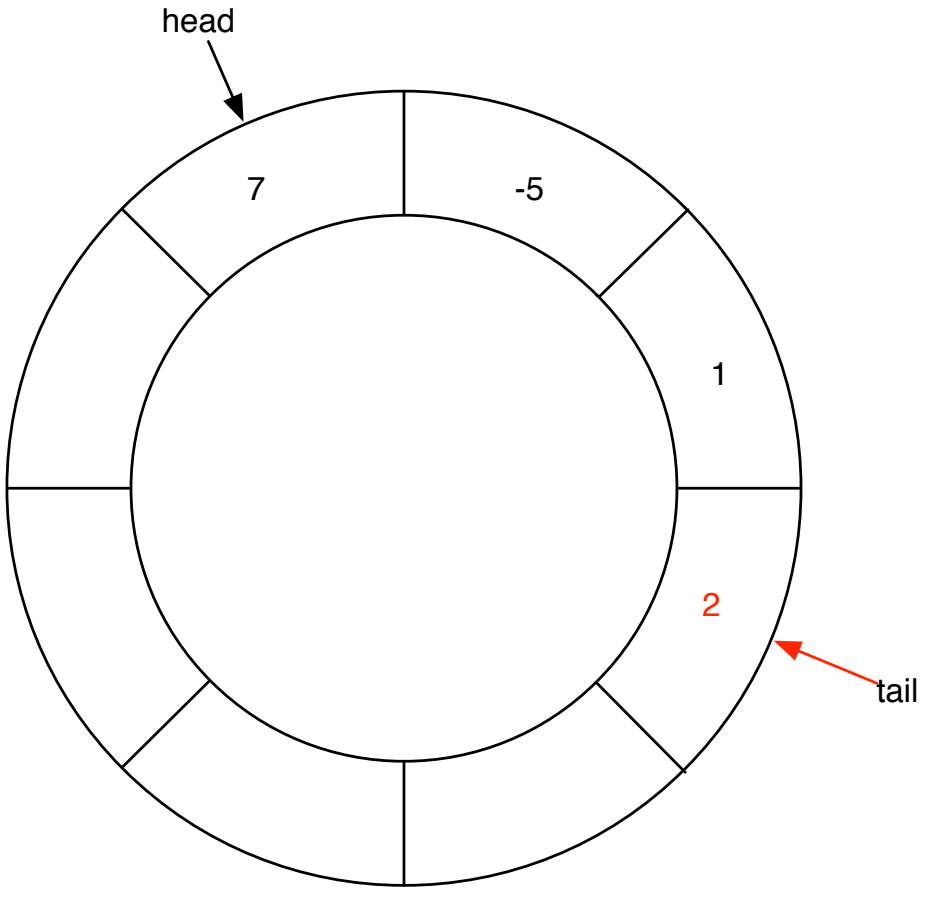
Code circolari

- A livello astratto una coda limitata viene rappresentata come un array circolare: l'ultima posizione nell'array è *contigua* alla prima posizione nell'array.
- Per mantenere traccia della posizione occupata dal primo (`head`) e ultimo `tail` elemento nella coda utilizziamo due indici distinti.
- Le operazioni di aggiornamento di tali indici sono operazioni di incremento modulo la lunghezza dell'array.



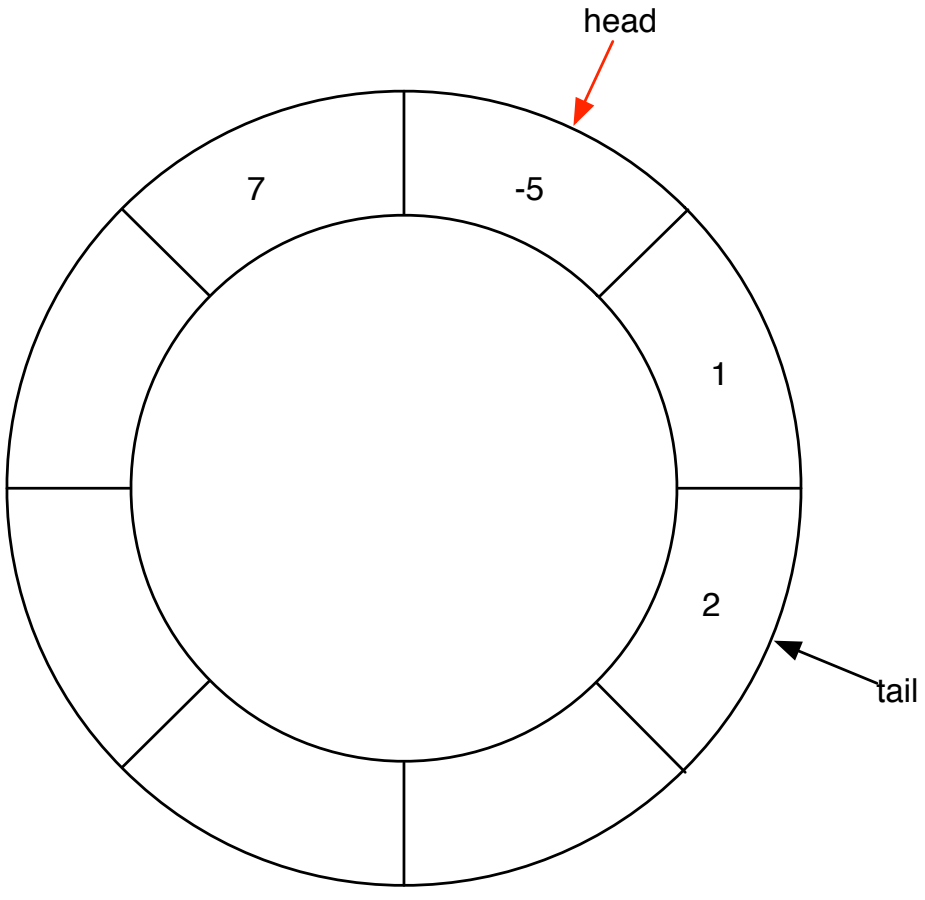
Code circolari: inserimento

- L'inserimento comporta l'incremento dell'indice che punta all'ultimo elemento nella coda.



Code circolari: rimozione

- La rimozione comporta l'incremento dell'indice che punta al primo elemento nella coda.



Prototipi delle funzioni di libreria per code limitate

- Caratteristiche specifiche della libreria per code con capienza limitata:
  - La funzione `init()` richiede come argomento la dimensione della coda.
  - Aggiungiamo una nuova funzione di libreria per determinare se la coda è piena.
  - La costante `NIL` serve per specificare un *indice nullo*.

```
1  #define NIL -1
2
3  typedef struct queue {
4    unsigned int size;
5    long int head;
6    long int tail;
7    int *data;
8  } queue;
9
10 queue init(unsigned int n);
11
12 void clear(queue *Q);
13
14 int isempty(queue *Q);
15
16 int isfull(queue *Q);
17
18 int enqueue(queue *Q, int elem);
19
20 int dequeue(queue *Q, int *elem);
21
22 int front(queue *Q, int *elem);
```

Creazione e distruzione di una coda: `init()` e `clear()`

- La funzione di creazione di una coda (`init()`) è equivalente alla funzione di creazione di una lista rappresentata con array. Come per la struttura dati pila con capienza limitata la dimensione della coda non verrà modificata a run-time.
- La funzione di distruzione (`clear()`) si occupa di deallocare l'array dinamico: la coda non può essere ulteriormente utilizzata, deve essere riallocata.

```
1 queue init(unsigned int n) {
2   queue tmp = {0,NIL,NIL,NULL};
3
4   tmp.data = (int *)malloc(n*sizeof(int));
5   if(tmp.data != NULL)
6     tmp.size = n;
7   return tmp;
8 }
```

```
1 void clear(queue *Q) {
2   if(Q != NULL) {
3     free(Q->data);
4     Q->data = NULL;
5     Q->head = NIL;
6     Q->tail = NIL;
7     Q->size = 0;
8   }
9 }
```

Funzioni di test sul numero di elementi nella lista: `isempty()` e `isfull()`

- Le funzioni di test verificano che la coda sia piena o vuota. Ritornano un valore booleano.
- Una coda è piena se la posizione successiva all'indice che punta all'ultimo elemento coincide con l'indice che punta al primo elemento della coda (riga 3 in `isfull()`).
- Da notare che se la coda viene deallocata utilizzando la funzione `clear()`, chiamate a `is_empty()` e `is_full()` su tale struttura ritornano *true*: una coda deallocata ha capacità nulla (è quindi sempre piena) e non contiene elementi.

```
1 int isempty(queue *Q) {
2   return Q == NULL || Q->head == NIL;
3 }
```

```
1 int isfull(queue *Q) {
2   return Q == NULL || Q->data == NULL ||
3          ((Q->tail + 1) % Q->size == Q->head);
4 }
```

Funzione di inserimento: `enqueue()`

- La funzione di inserimento esegue le seguenti operazioni:
  - Se la coda è piena (riga 2) non effettua l'inserimento e ritorna 1 (inserimento non riuscito).
  - Altrimenti (blocco else a riga 4)
    - Se la coda è vuota (riga 5), gli indici che puntano al primo e ultimo elemento nella coda sono entrambi inizializzati alla posizione 0 (riga 6).
    - Altrimenti (riga 7), l'indice che punta all'ultimo elemento è incrementato di 1, modulo la lunghezza dell'array (riga 8).
  - Il nuovo elemento è inserito nella nuova ultima posizione (riga 10).

```
1  int enqueue(queue *Q, int elem) {
2    if(isfull(Q)) {
3      return 1;
4    } else {
5      if (isempty(Q))
6        Q->tail = Q->head = 0;
7      else
8        Q->tail = (Q->tail + 1) % Q->size;
9
10     Q->data[Q->tail] = elem;
11     return 0;
12   }
13 }
```

Rimozione e selezione del primo elemento : `dequeue()` e `front()`

- La rimozione del primo elemento nella coda viene effettuata incrementando di 1, modulo la lunghezza dell'array, l'indice che punta al primo elemento (riga 9 in `dequeue()`).
- E' necessario gestire in modo speciale il caso in cui la coda contenga un solo elemento (riga 7 in `dequeue()`).

```
1  int dequeue(queue *Q, int *elem) {
2    if(isempty(Q) || elem == NULL) {
3      return 1;
4    } else {
5      *elem = Q->data[Q->head];
6      if(Q->head == Q->tail)
7        Q->head = Q->tail = NIL;
8      else
9        Q->head = (Q->head + 1) % Q->size;
10     return 0;
11   }
12 }
```

```
1  int front(queue *Q, int *elem) {
2    if(isempty(Q) || elem == NULL) {
3      return 1;
4    } else {
5      *elem = Q->data[Q->head];
6      return 0;
7    }
8  }
```