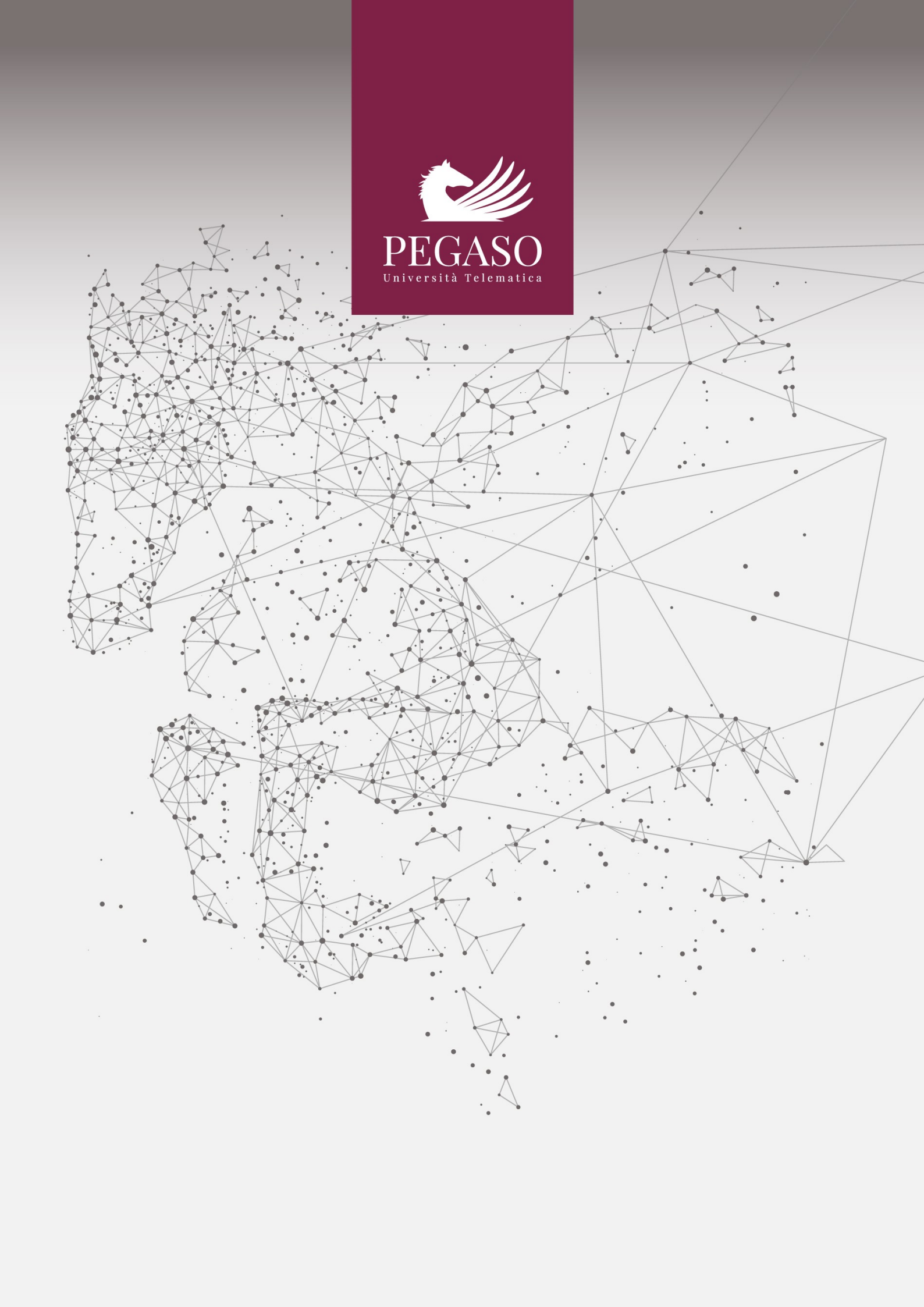
PEGASO
Università Telematica

Indice

1. Coda

Una coda è una struttura che realizza un criterio di inserimento e cancellazione dei dati chiamato **FIFO** (**First in First Out**): in questo caso il primo elemento che può essere cancellato coincide con il primo elemento introdotto, cioè il più vecchio tra quelli presenti nella struttura.

Se le pile possono essere interpretate come liste nelle quali le operazioni di proiezione, inserimento e cancellazione si possono applicare solo all'ultimo elemento, anche per le cose si tratta di una sorta di restrizione della nozione di lista e la differenza tra le due strutture è limitata alle loro operazioni.

Le operazioni principali che si possono effettuare su una coda sono: inserimento di un elemento (**enqueue**), rimozione di un elemento (**dequeue**), e recupero dell'elemento in cima alla coda (**front**).

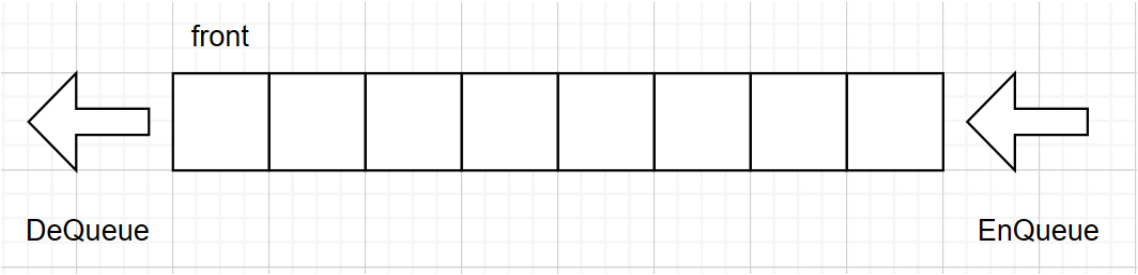

Figura 1 *- Rappresentazione di una coda*

La scelta della struttura dati da utilizzare, potrebbe essere in prima battuta a favore di un array ma questo presenta alcuni problemi di gestione.

Immaginiamo infatti che tale coda possa avere una dimensione massima pari a 10 (al più 10 elementi in coda). Quando viene eseguita una operazione di inserimento in coda, se questa non è piena, l'elemento verrà inserito nella prima posizione disponibile; quando viene eseguita però una operazione di rimozione, quello che succede è che si dovrebbe spostare "tutta" la coda in avanti:

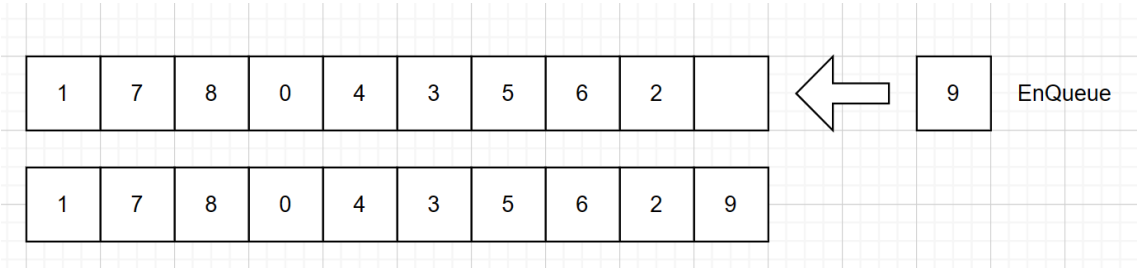

Figura 2 *- EnQueue*

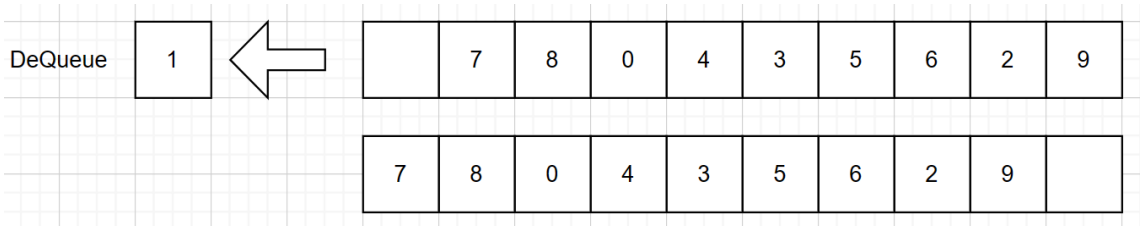

Figura 3 *- DeQueue*

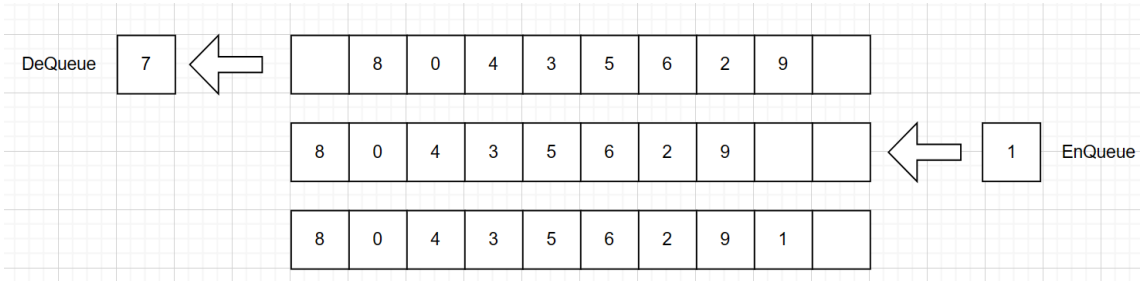

Figura 4 *- DeQueue & EnQueue*

Si intuisce chiaramente che l'operazione di "shift" degli elementi è cmq un'operazione "costosa" da eseguire ogni volta che si esegue un DeQueue; pertanto, un'alternativa a questa può essere offerta dall'introduzione di una coda "circolare".

Immaginiamo di avere una coda così fatta: [3, 2, 4, 1, 8, 3, 5, 9] essendo 3 l'elemento front (il primo entrato in coda ed il primo ad uscire). La coda circolare potrebbe essere rappresentata nel seguente modo (essendo 3 l'elemento in testa – head e 9 l'elemento in coda – tail):

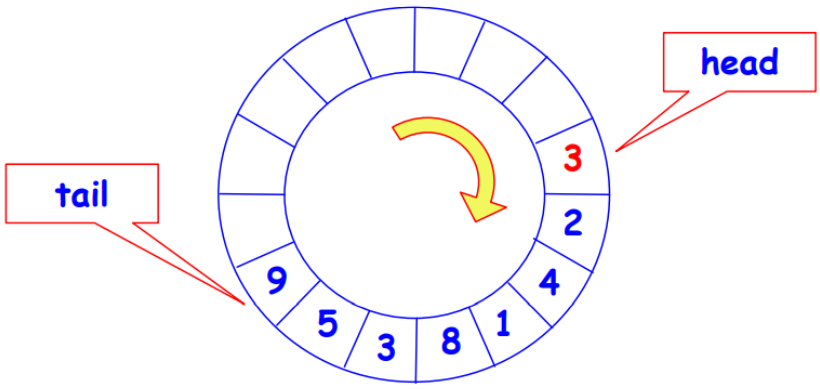

Figura 5 *- Coda circolare*

Se eseguiamo un DeQueue, l'elemento 3 verrà eliminato dalla coda e la nuova rappresentazione sarà la seguente:

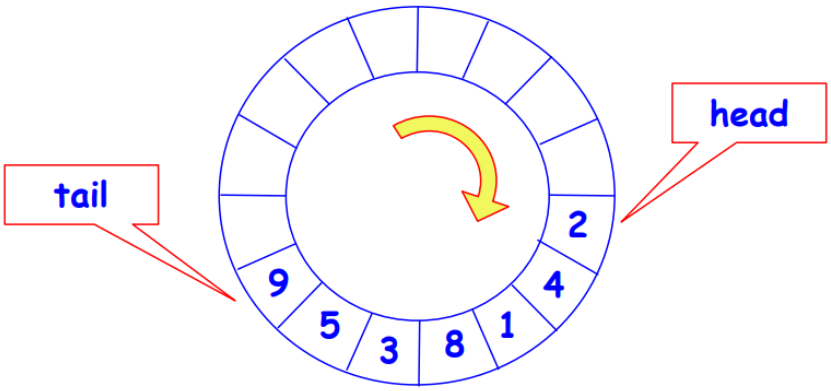

Figura 6 *- DeQueue*

Eseguiamo ora un EnQueue dell'elemento 7, la nuova rappresentazione sarà:

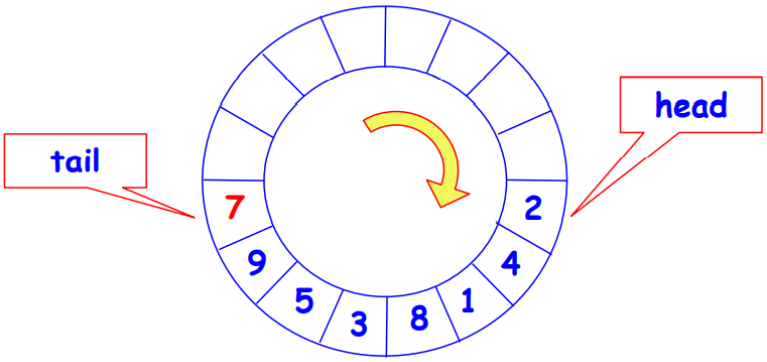

Figura 7 *- EnQueue*

Mediante questa strategia, non è più necessario eseguire uno shift ad ogni DeQueue, ma sarà sufficiente aggiornare la variabile head.

Ovviamente questa struttura può anche essere implementata tramite un array:

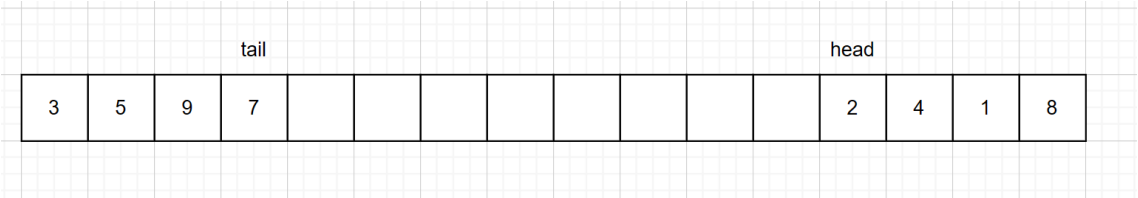

Figura 8 *- Array associato a coda circolare*

Come si evince dalla figura si ha dunque un array di 16 celle (esattamente come le 16 celle della coda circolare) con head e tail che contengono il valore dell'indice dell'elemento in testa alla coda e dell'ultimo elemento in coda.

Se dunque indichiamo con **len** la dimensione della coda (nel caso precedente, dunque, len=16), nel momento in cui si esegue una operazione di DeQueue, l'aggiornamento di head sarà: $head = (head + 1)\ \%\ len$. Una operazione di EnQueue invece prevede l'aggiornamento: $tail = (tail + 1)\ \%\ len$:

- **DeQueue**: $head = (head + 1)\ \%\ len$
- **EnQueue**: $tail = (tail + 1)\ \%\ len$

Ovviamente tail potrà raggiungere ma non potrà superare head: in tal caso la coda è piena e dunque non si può aggiungere alcun elemento; allo stesso modo head potrà raggiungere ma non potrà superare tail: in tal caso la cosa è vuota.

Tuttavia, c'è uno scenario in cui head e tail possono coincidere:

- La coda è vuota: in tal caso head e tail identificano la prima posizione vuota della coda.
- La coda ha un solo elemento: in tal caso head e tail identificano lo stesso elemento presente.

Per gestire tale scenario vi sono 2 possibilità:

1. Lasciare sempre una casella vuota e far indicare a tail la prima posizione vuota: in tal caso head e tail possono sovrapporsi solo se la coda è vuota

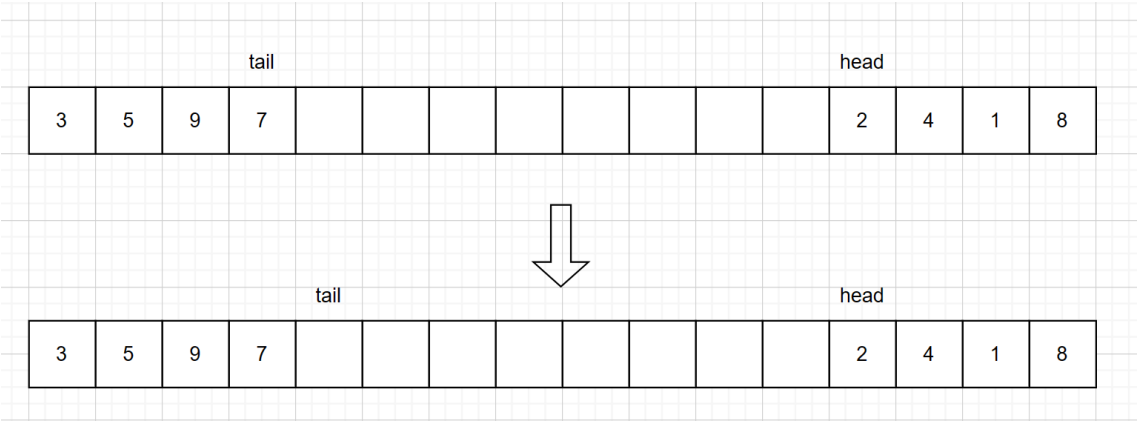

Figura 9 *- tail punta alla locazione libera e non all'ultimo elemento*

2. Usare una variabile booleana per sapere se la cosa contiene almeno un elemento

In generale le operazioni che è possibile effettuare sulla coda sono:

- **EnQueue(x)**: inserire un elemento x in coda.
- **DeQueue()**: togliere un elemento dalla coda.

- **IsEmpty()**: verificare se la coda è vuota.
- **ClearQueue()**: svuotare la coda.
- **Front()**: leggere il primo elemento della coda.
- **IsFull()**: verificare se la coda è piena (nel caso in cui la coda ha una capacità massima).

2. Implementazione

Riportiamo di seguito le varie implementazioni iniziando dallo scenario basato su una struttura dati array:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100

int queue[MAX_SIZE];
int front = 0;
int rear = -1;
int item_count = 0;

void insert(int data) {
    if (item_count == MAX_SIZE) {
        printf("Error: Queue overflow\n");
    } else {
        rear = (rear + 1) % MAX_SIZE;
        queue[rear] = data;
        item_count++;
    }
}

int remove_data() {
    int data;
    if (item_count == 0) {
        printf("Error: Queue underflow\n");
        return -1;
    } else {
        data = queue[front];
        front = (front + 1) % MAX_SIZE;
        item_count--;
        return data;
    }
```

```
}

void print_queue() {
    int i;
    if (item_count == 0) {
        printf("Error: Queue is empty\n");
    } else {
        for (i = 0; i < item_count; i++) {
            int index = (front + i) % MAX_SIZE;
            printf("%d\n", queue[index]);
        }
    }
}
```

Passiamo alla CircularQueue implementata in C++:

```
#include <iostream>

using namespace std;
const int MAX_SIZE = 10;

class CircularQueue {
  public:
    CircularQueue() {
      head = tail = -1;
      buffer = new int[MAX_SIZE];
    }

    ~CircularQueue() {
      delete [] buffer;
    }

    void enqueue(int value) {
      if ((tail + 1) % MAX_SIZE == head) {
        cout << "Coda piena, non è possibile inserire elementi." << endl;
        return;
      }
```

```
            tail = (tail + 1) % MAX_SIZE;
            buffer[tail] = value;

            if (head == -1) {
              head = tail;
            }
          }

          int dequeue() {
            if (head == -1) {
              cout << "Coda vuota, non è possibile rimuovere elementi." << endl;
              return -1;
            }

            int value = buffer[head];
            if (head == tail) {
              head = tail = -1;
            } else {
              head = (head + 1) % MAX_SIZE;
            }

            return value;
          }

        private:
          int head, tail;
          int *buffer;
      };
```

In c++ esiste anche una classe queue che consente di gestire direttamente in maniera "nativa" una coda:

```
      #include <iostream>
      #include <queue>

      using namespace std;

      int main() {
```

```
    queue<int> queue;

    queue.push(10);
    queue.push(20);
    queue.push(30);

    while (!queue.empty()) {
      int front = queue.front();
      cout << front << endl;
      queue.pop();
    }

    return 0;
  }
```

I metodi offerti dalla classe Queue sono i seguenti:

- void push(const T& x): Inserisce un elemento x in coda alla coda.
- void pop(): Rimuove l'elemento in testa alla coda.
- T& front(): Restituisce un riferimento all'elemento in testa alla coda.
- const T& front() const: Restituisce un riferimento costante all'elemento in testa alla coda.
- T& back(): Restituisce un riferimento all'elemento in coda alla coda.
- const T& back() const: Restituisce un riferimento costante all'elemento in coda alla coda.
- bool empty() const: Restituisce vero se la coda è vuota, falso altrimenti.
- size_t size() const: Restituisce la dimensione della coda.

Riportiamo di seguito anche le implementazioni in Python; si seguito l'implementazione di una coda non circolare:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if len(self.queue) == 0:
```

```
                return None
            else:
                return self.queue.pop(0)

        def is_empty(self):
            return len(self.queue) == 0

queue = Queue()
queue.enqueue(1)
print(queue.dequeue())
```

Di seguito l'implementazione di una coda circolare:

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.head = self.tail = -1

    def enqueue(self, item):
        if ((self.tail + 1) % self.size == self.head):
            print("Queue is Full")
        elif (self.head == -1):
            self.head = 0
            self.tail = 0
            self.queue[self.tail] = item
        else:
            self.tail = (self.tail + 1) % self.size
            self.queue[self.tail] = item

    def dequeue(self):
        if (self.head == -1):
            print("Queue is Empty")
        elif (self.head == self.tail):
            temp = self.queue[self.head]
            self.head = -1
            self.tail = -1
            return temp
```

```
            else:
                temp = self.queue[self.head]
                self.head = (self.head + 1) % self.size
                return temp

        def display(self):
            if(self.head == -1):
                print("No element in the Queue")
            elif (self.tail >= self.head):
                for i in range(self.head, self.tail + 1):
                    print(self.queue[i], end=" ")
            else:
                for i in range(self.head, self.size):
                    print(self.queue[i], end=" ")
                for i in range(0, self.tail + 1):
                    print(self.queue[i], end=" ")

    queue = CircularQueue(5)
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    queue.display()
    print("\n")
    print(queue.dequeue())
    queue.display()
```

3. Complessità

In generale, le operazioni di base eseguite su una coda circolare sono di complessità costante $O(1)$. Ciò significa che il tempo richiesto per eseguire un'operazione non dipende dalla dimensione della coda, ma è sempre lo stesso.

Se invece non utilizziamo una coda circolare, la complessità asintotica di questa implementazione per le operazioni di inserimento (enqueue) e rimozione (dequeue) di elementi è di $O(n)$, dove n è la lunghezza corrente della coda. Ciò significa che il tempo richiesto per eseguire queste operazioni dipende dalla lunghezza della coda e diventa sempre più lungo man mano che la coda diventa più lunga.

La memoria utilizzata dalla coda è proporzionale alla dimensione massima della coda. Questo significa che se la dimensione massima della coda è N, allora l'utilizzo di memoria sarà $O(N)$.

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione.
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione.
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley.
- Sedgewick: Algoritmi in C, Pearson, 2015.
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.