Le pile (*o stack*)

Le pile sono delle strutture dati di tipo ***LIFO*** (ovvero "last-in-first-out"). Le operazioni di inserimento e rimozione, pertanto avvengono sempre **dalla cima**.

Esse possono essere viste come pile di piatti disposti dal basso verso l'alto e tolti dall'alto verso il basso. *Laverò per primo l'ultimo arrivato*.

> Una pila può essere o meno limitata, pertanto può essere implementata come un array o come una lista, pur restando **sempre una pila**.

Altro

Le operazioni che possono essere svolte con le pile

Indipendentemente dalla limitazione della pila, con esse possono sempre essere svolte determinate operazioni:

1. aggiunzione di un elemento alla pila (`push`)
2. rimozione di un elemento dalla pila (`pop`)

3. controllo se la pila è vuota (`isEmpty()`)

Da notare che

push non può sempre essere effettuato, infatti se la pila dovesse avere una dimensione massima occupata sarebbe sarebbe impossibile effettuare tale operazione perché si incorrerebbe in uno **Stack Overflow**.

Gli attributi di una pila

Indipendentemente dalla limitazione della pila, esse hanno le medesime proprietà:

1. la **dimensione** (`size`), ovvero il *numero di elementi **attualmente** presenti nella pila*;
2. la **cima**, ovvero il ***puntatore** all'elemento che si ottiene tramite l'istruzione* `top()`;
3. la **dimensione massima** (`maxSize`).

Eccezioni della pila

Le eccezioni sono oggetti che rappresentano *errori* nell'esecuzione del programma, alcuni di essi sono **predefiniti** nelle librerie standard. Nella programmazione ad oggetti estendiamo il concetto di *eccezioni* nel seguente modo:

1. se tento di estrarre un elemento dalla **pila vuota** si deve dire che *l'operazione non è permessa*;
2. se tento di inserire qualcosa in una **pila piena** si deve dire che *l'operazione non è permessa*.

Complessità delle operazioni dello STACK

La complessità dipende dalle implementazioni. Ma dato che solitamente nelle pile sono presenti tre operazioni si avranno queste complessità:

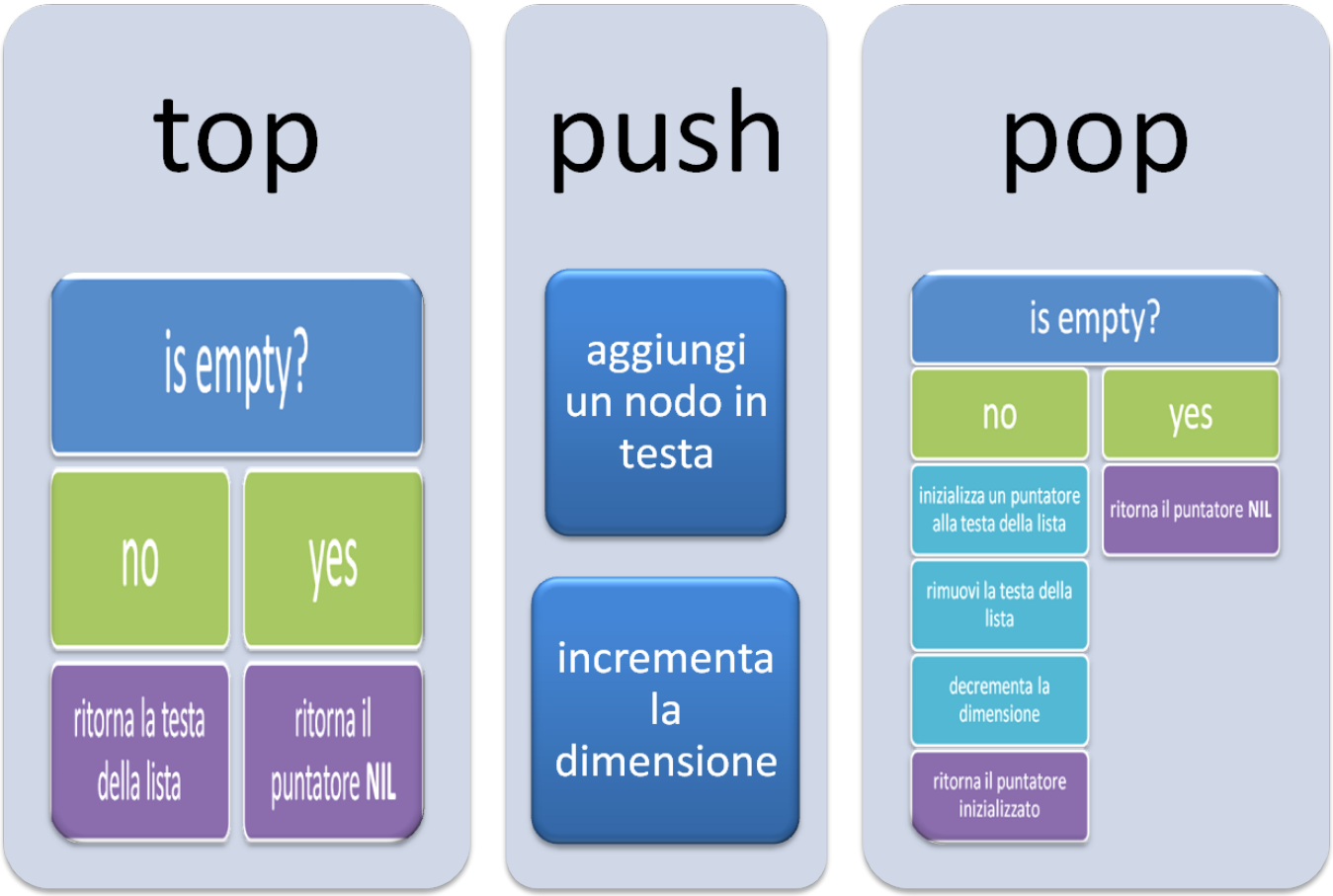
1. `push()` = O(1) *perché non serve scorrere la pila*;
2. `pop()` = O(1) *perché non serve scorrere la pila*;
3. `top` = O(1) *perché non serve scorrere la pila*.

Se volessi inserire `push()` in **coda** o estrarre `pop()` **in coda** la complessità diventa O(n).

Pila illimitata (*o dinamica*)

Per implementare una pila dinamica si istanzia una classe `template` che eredita in maniera *protected* da `List`.

Alla classe vanno poi posti l'attributo *private* dimensione (`size`) della pila e come metodi *public* `top()`, `push()` e `pop()`.



> Dato che nel `main` dovrò poter accedere **solo ai metodi della pila**, erediterò in maniera `protected` in modo tale da rendere gli attributi e i metodi della classe lista **inaccessibili** (ovviamente *all'esterno della classe pila*).
>
> > se infatti avessi utilizzato `public` anche nel main avrei potuto utilizzare - ad esempio - il metodo `insertHead()`.

Codice

Per prima cosa controlliamo se la pila (**lo Stack**) è già stata definita e la definiamo, includiamo le librerie e gli header necessari

```C++
#ifndef STACK_H
#define STACK_H
```

```
#include<iostream>
#include "list.h"

using namespace std;
```

Successivamente istanziamo una classe template che eredita da list in maniera protected e che abbia come parametro *private* la dimensione inizializzata a zero e il costruttore riportando il costruttore di `List`.

```
C++
template <typename T>
class Stack : protected List<T> {
    private:
    int size = 0;
    public:
    Stack() : List<T>{};
    ```

Implementiamo poi:
1. La funzione `isEmpty()` che torna il valore booleano se la dimensione dovesse essere pari a 0.

```cpp
    bool isEmpty(){
        return size == 0;
    }
        ```

2. La funzione `top` che restituisce il puntatore alla testa o a `nullptr` se la pila dovesse essere vuota

```cpp
```

```
Node<T>* top(){
    if(isEmpty()){
        retunr nullptr;
    }
    return List<T>::getHead();
}
```

3. La funzione `push` alla quale va passato un valore e che richiama la funzione `insertHead` dalla classe madre ed incrementa la dimensione

C++

```
void push(T val){
    List<T>::insertHead(val);
    size++;
}
```

4. la funzione `pop` che torna `nullptr` se la lista *è vuota* o il puntatore al top della lista altrimenti (dopo aver richiamato `removeHead()` dalla classe madre ed aver decrementato la dimensione)

C++

```
Node<T>* pop(){
    if(isEmpty()){
        return nullptr;
    }
    Node<T>* ptr = top();
    List<T>::removeHead();
    size--;
    return ptr;
}
```

5. la funzione overload di stampa

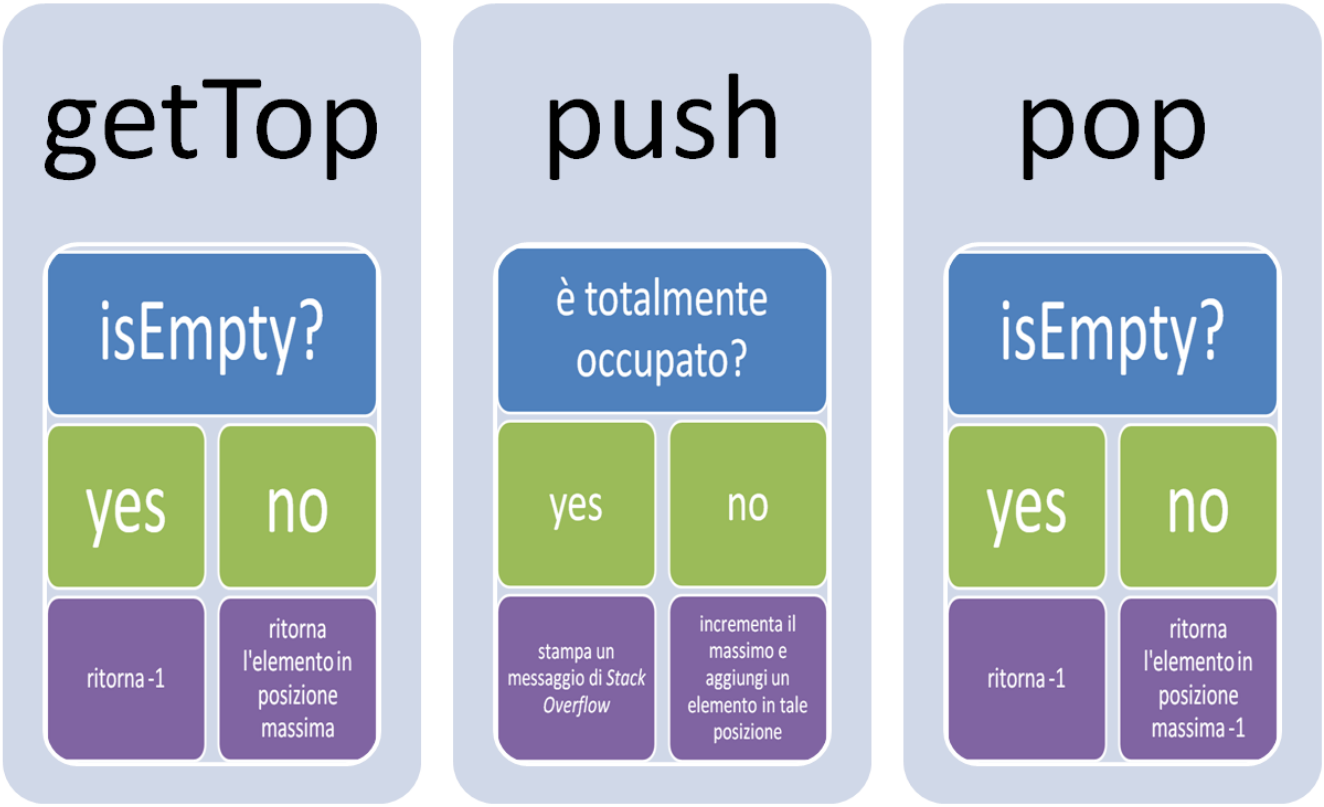
```C++
    friend ostream& operator<< (ostream& out, stack<T>& s)
{
        retur out << (List<T>)s; // cast da stack a list
    }
};

#endif
```

Pila limitata (*o statica*)

Per implementare una pila statica si istanzia una classe `template` con un **array**.

Pertanto gli attributi *private* sono un array template, il `top` (*indice dell'ultimo elemento aggiunto*), la `size` e la `maxSize` (posta a -1). I metodi *public* sono `getTop`, `push` e `pop`.



Codice

In primo luogo controlliamo se la pila (**stack**) statica è stata già definita, altrimenti la definiamo. includiamo le librerie necessarie

```C++
#ifndef STATIC_STACK_H
#define STATIC_STACK_H
#include<iostream>
using namespace std;
```

Successivamente istanziamo una classe *template* che abbia come parametri *private* un array `T*` e tre interi: `top` (*indice dell'elemento in cima*), `size` (la dimensione attuale), `maxSize` (dimensione massima)

```C++
template<typename T>
class StaticStack(){
    T* array;
    int top = -1;
    int size = 0;
    int maxSize=-1;
}
```

Successivamente implemento in public:

1. il costruttore (in cui inizializzo l'array e maxSize)

```C++
StaticStack(int _maxSize) : maxSize(_maxSize){
    array = new T[maxSize];
}
```

2. `getTop()` che restituisce -1 se la pila è vuota, il valore in posizione top altrimenti

```C++
T getTop(){
    if(isEmpty()){
        return -1;
    }
    return array[top];
}
```

3. `push` a cui passo come parametro un valore. Se il top è pari a `maxSize-1` (quindi è **pieno**) stampo un messaggio di errore, altrimenti incremento il top e aggiungo il valore

```C++
void push(T val){
    if(top == maxSize-1){
        cout << "Stack overflow" << endl;
        return;
    }
    array[++top] = val;
}
```

4. `pop` che restituisce l'elemento eliminato e decrementa il top se la lista non è vuota, -1 altrimenti

```C++
T pop(){
    if (isEmpty()){
        return -1;
    }
```

```
        return array[top--];
    }
```

5. `isEmpty()` che restituisce l'elemento booleano *true* se il top è pari a -1

```C++
    bool isEmpty(){
        return top = -1;
    }
```

6. overload di ostream.

```C++
    friend ostream& operator<<(ostream& out,
StaticStack<T>&s){
        out << "Static Stack: maxSize " << s.maxSize <<
endl;
        out << "--------------" << endl;
        for(int i=s.top; i=0; i--){
            out << s.array[i] << " - " ;
        }
    };

#endif
```

Le code (*o queue*)

Le code sono delle strutture dati di tipo ***FIFO*** (ovvero first-in-first-out). Le operazioni di inserimento e rimozione, pertanto non avvengono **unicamente** dalla cima.

Esse possono essere viste come code di persone (alla posta, ad un concerto, al supermercato...) dove le persone arriveranno mano mano e si metteranno "uno dietro l'altro". *Il primo ad essere servito sarà il primo ad essere arrivato.*

> Una coda può essere o meno limitata, pertanto può essere implementata come un array o come una lista, pur restando *sempre una coda*

Altro

Le operazioni che possono essere svolte con le code

Indipendentemente dalla limitazione della coda, con esse possono sempre essere svolte determinate operazioni:

1. controllo se la coda è vuota (`isEmpty()`);
2. **enqueue**, inserimento in coda (`insertTail()`);
3. **dequeue**, rimozione dalla coda (`removeTail()`).

Gli attributi di una coda

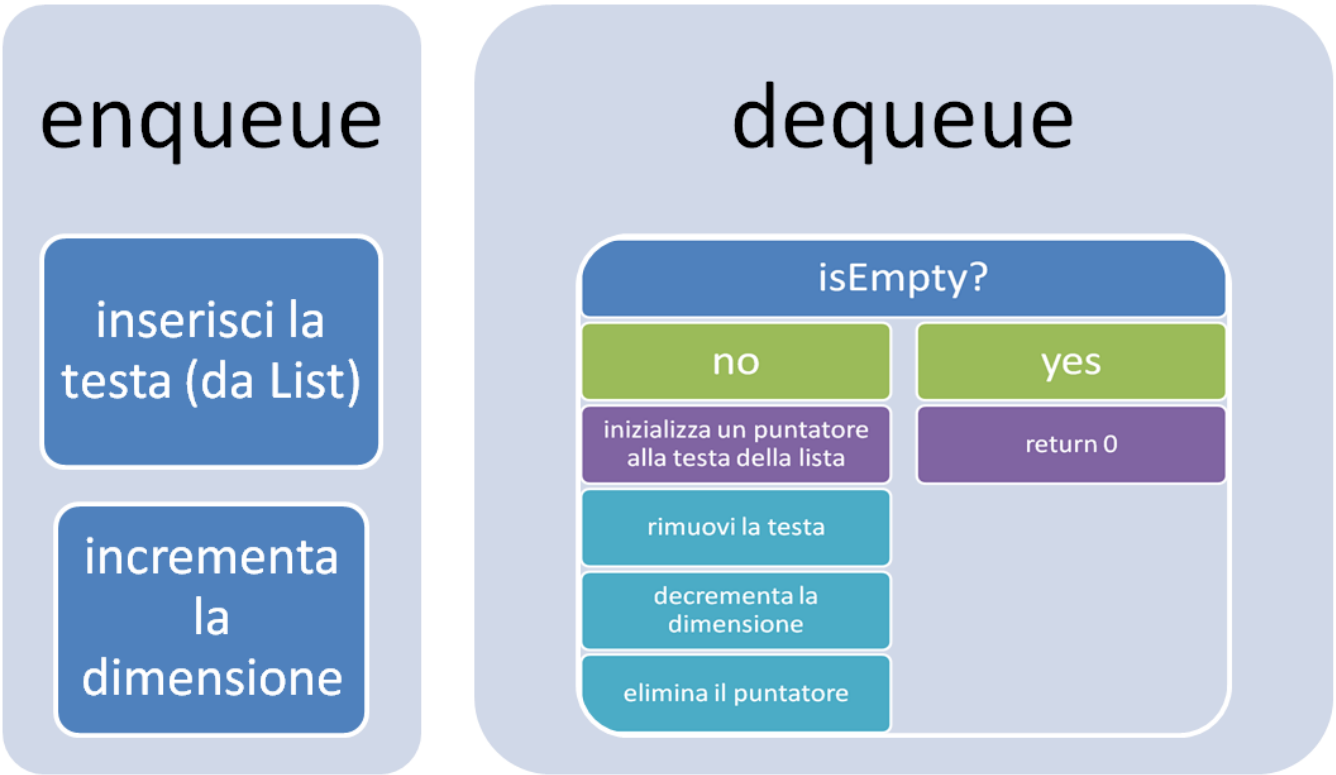
Indipendentemente dalla limitazione della coda, esse hanno le medesime proprietà:

1. la **dimensione** (`size`), ovvero il *numero di elementi* ***attualmente*** *presenti nella pila*;
2. la **testa** (`head`);
3. la **coda** (`tail`).

Coda illimitata (*dinamica*)

Per implementare una coda dinamica si istanzia una classe `template` che eredita in maniera *protected* da `DLList`.

Alla classe vanno poi posti l'attributo *protected* dimensione e i metodi *public* `enqueue`, `dequeue`, `isEmpty`.



> nella classe DLList il modificatore d'accesso *private* deve essere cambiato in **protected** per poter accedere agli attriuti `head` e `tail` da una sottoclasse.

> Dato che nel `main` dovrò poter accedere **solo ai metodi della coda**, erediterò in maniera `protected` in modo tale da rendere gli attributi e i metodi della classe DLlist **inaccessibili** (ovviamente *all'esterno della classe coda*).
>
> > se infatti avessi utilizzato `public` anche nel main avrei potuto utilizzare - ad esempio - il metodo `insertTail()`.

Codice

In primo luogo controlliamo se QUEUE è già stata definita e includiamo l'**header** `DLList` e le librerie necessarie

```C++
#infndef QUEUE_H
#define  QUEUE_H
#include "DLlist.h"
#include<iostream>
using namespace std;
```

Successivamente istanzio una classe `template` che erediti da `DLList` in maniera ***protected***, come attributo *protected* implementerò un intero riferito alla dimensione della coda (inizializzato a 0)

```C++
template <typename T>
class Queue : protected DLList<T>{
    protected:
    int size = 0;
```

E come metodi public istanzio:

1. il costruttore, richiamando il costruttore della classe madre

```C++
    public:
    Queue() : DLList<T>(){};
```

2. `enqueue` al quale passo un parametro val che servirà per inserire un nuovo elemento richiamando dalla classe madre la funzione `insertTail()` e incrementando la dimensione della coda

```C++
    void enqueue(T val){
        DLList<T> :: insertTail(val);
        size++;
    }
```

3. `dequeue` che ritorna 0 se la lista è vuota, altrimenti ccrea un riferimento alla testa, rimuove la testa, decrementa la dimensione e restituisce il riferimento

```C++
    DLNode<T> dequeue(){
        if (isEmpty()){
            return 0;
        }

        DLNode<T> ptr = *(DLList<T>::head); // riferimento
alla testa
        DLList<T> :: removeHead();
        size--;
        return ptr;
    }
```

4. `isEmpty` che ritorna il valore booleano true se la dimensione è pari a zero

```C++
    bool isEmpty(){
        return (size == 0);
    }
```

5. l'overload dell'operatore ostream&

```C++
    friend ostream& operator<<(ostream& out, Queue<T>&
queue){
        out << "Queue starting at " << &(queue, head);
        DLNode<T> *ptr = queue.head;
        while (ptr){
            out << *ptr << endl;
            ptr = ptr->getNext();
        }
        return out;
    }
};
#endif
```

Attenzione a:

1. scrivere dllist o dlnode
2. Con la rimozione eliminiamo il valore, ma il puntatore viene conservato. Per mantenere il valore potrei fare `delete pointer` o spostare la testa al suo successore
3. Per fare `delete` conservando il nodo potrei utilizzare il costruttore di copia o creare un nuovo nodo uguale nel quale mantengo il

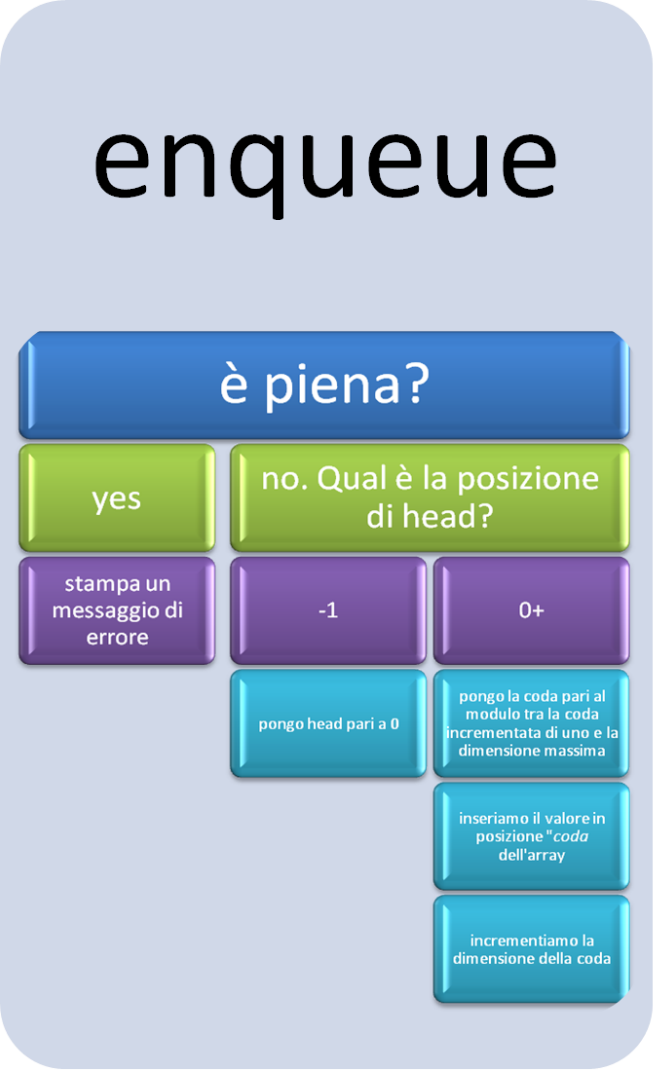
valore del nodo eliminato

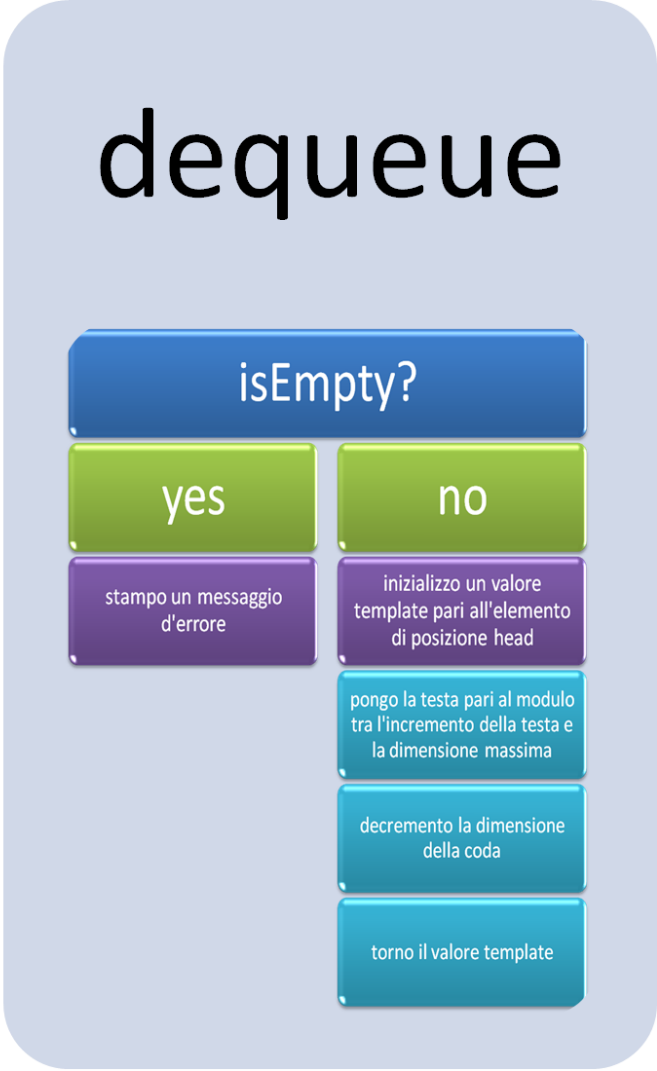
4. Per stampare correttamente devo considerare anche il caso in cui *gli elementi siano maggiori*

Coda limitata (*statica*)

Per implementare una coda statica si istanzia una classe `template` con un **array**.

Pertanto gli attributi *private* sono un array template, il `top` (*indice dell'ultimo elemento aggiunto*), la `size`,la `maxSize` (posta a -1), la `head` e la `tail` (poste entrambe a -1). I metodi *public* sono `dequeue` ed `enqueue`.





> per evitare di avere *errori con gli indici* a causa di una serie di `enqueue()` seguiti da corrispettivi `dequeue`, usiamo la funzione **modulo** evitando errori di ***segmantation fault*** (cioè errori causati dall'utilizzo di una *posizione "non conosciuta"*)

Codice

In primo luogo controllo se `STATIC_QUEUE_H` è già stata definita, altrimenti la definisco; includo le librerie necessarie e definisco la dimensione massima dell'array

```C++
#ifndef STATIC_QUEUE_H
#define STATIC_QUEUE_H
#include<iostream>
#define MAX_SIZE 1000
```

Successivamente istanzio una classe `template` che abbia come attributi *private* un array template e quattro interi (head, tail, size e maxSize).

```C++
template<typename T>
class StaticQueue{
    T* array;
    int size = 0;
    int maxSize = MAX_SIZE;
    int head = -1;
    int tail = -1;
```

E come metodi public istanzio:

1. il costruttore, inizializzando la dimensione massima e l'array

```C++
public:
StaticQueu(int maxsize = MAX_SIZE) : maxSize(maxSize){
    this->array = new T[maxSize];
}
```

2. `enqueue()`, passando come *parametro un valore template*. Controllo se la dimensione è pari alla dimensione massima e in quel caso stampo un **messaggio d'errore**, altrimenti controllo se head è pari a -1 per impostarla a 0 oppure pongo tail pari al ***modulo*** tra tail (incrementata di 1) e la dimensione massima, inserisco in quel posto il valore e incremento la dimensione

```C++
void enqueue(T val){
    if (size == maxSize){
        cout << "queue is full" << endl;
        return;
    }

    if (head = -1)
        head = 0;
    tail = (++tail%maxSize);
    array[tail] = val;
    size++;
}
```

3. `dequeue()` che restituisce *-1 se la coda è vuota* (e stampa un messaggio di errore), altrimenti inizializza un valore al valore in posizione head, pone *head pari al modulo tra head (incrementata di 1) e la dimensione massima* e **decrementa la dimensione**; infine

ritorna il valore. Se head corrisponde a tail non è possibile effettuare la `dequeue`, per evitare errori va utilizzata la **funzione modulo**.
Il valore viene restituito perché è importante sia rimuovere il nodo *che* conoscere il contenuto del nodo rimosso.

```C++
T dequeue(){
    if (size == 0){
        cout << "queue is empty" << endl;
        return -1;
    }

    T val = array[head];
    head = (++head%maxSize);
    size--;
    return val;
```

4. overload dell'operatore ostream&, se la dimensione della lista è pari a zero lo comunica. Bisogna fare attenzione che head e tail rispettino la condizione (**head<tail**), altrimenti occorrerà utilizzare la ***funzione modulo***. (*questo potrebbe accadere se facendo dei dequeue la tail diventa 0 e la head resta pari a 1*). Quindi il modulo va messo **nell'incremento**, utilizzare il for non è funzionale a meno che non si utilizzi un'altra variabile perché non conviene fare il for "su" la testa, motivo per cui si utilizza `count`.

```C++
friend ostream& operator<< (ostream& out, StaticQueue<T>& queue){
    if (queue.size == 0){
        return out << "queue is empty" << endl;
    }
```

```
    out << "Static Queue - Size " << queue.size << ";
    out << ", maxSize= " << queue.maxSize << endl;
    for (int i=queue.head, count = 0; count<queue.size;
cout++){
        cout << "Queue[" << i << "]=" << queue.array[i] <<
endl;
        i = (i+1) % queue.maxSize;
    }
    return out;
}

#endif
```

Le code circolari

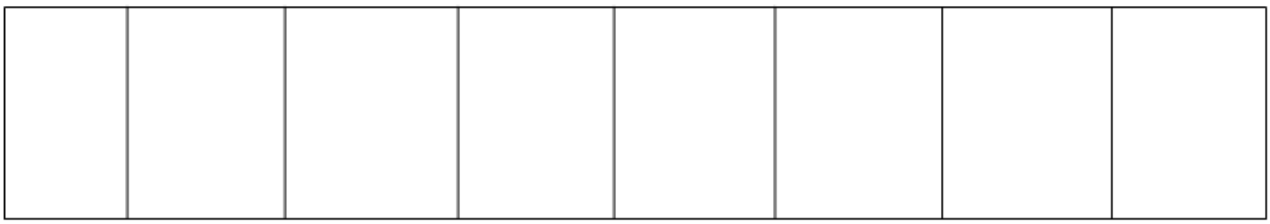
Quando si parla di code statiche ci si riferisce a delle code *circolari*. Questo nome è dato dal fatto che a seguito di una serie di dequeue() la testa sarà di volta in volta "*spostata*" all'elemento successivo, ma i valori **non saranno eliminati**. La loro eliminazione sarà data da una ***sovrascrittura*** che avviene grazie - appunto alla funzione metodo -.

Nell'implementazione delle liste doppiamente linkate è possibile formalizzare una coda circolare collegando `tail` e `head` _(sarebbe: `tail->next=head` e `head->prev=tail`).

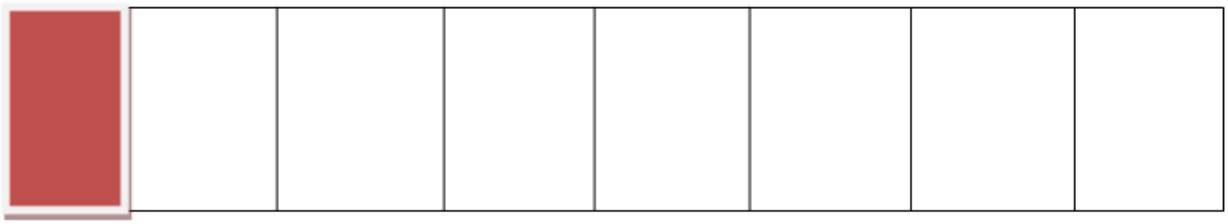
Un caso in cui torna utile il loro uso è quando si hanno condivisioni di risorse.

Esempio grafico

Il seguente è un array vuoto



che man mano va così riempito:



La stella bianca rappresenta il valore in testa

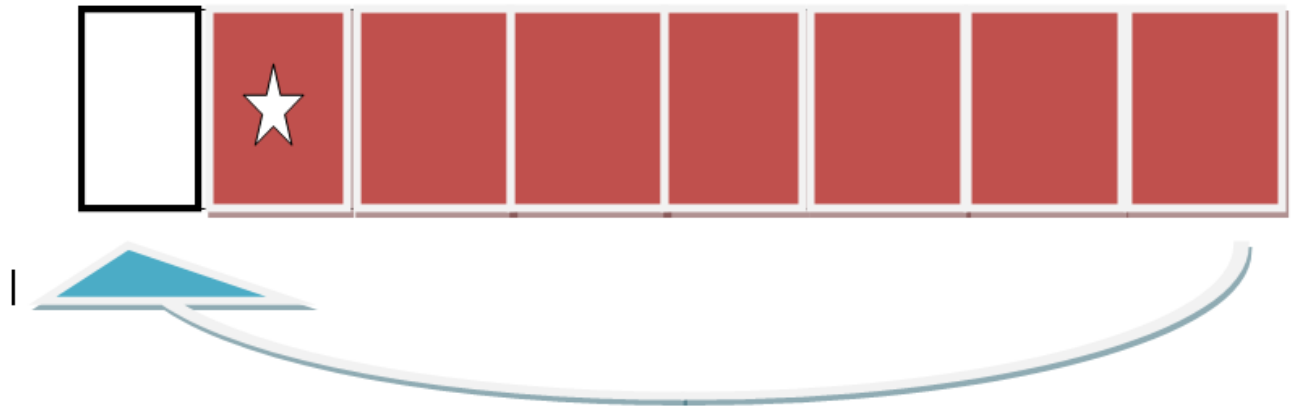
Una volta eliminato il valore in testa, la testa sarà il nodo successivo, **ma** il valore all'interno del nodo eliminato *non andrà perso*. Il solo elemento perso del nodo è il **riferimento** (in testa).

Una volta che l'array sarà *apparentemente pieno*, ovvero sarà occupato anche il valore di posizione *maxSize-1*

Il procedimento non si arresterà perché - in realtà - è presente un **nodo vuoto**.

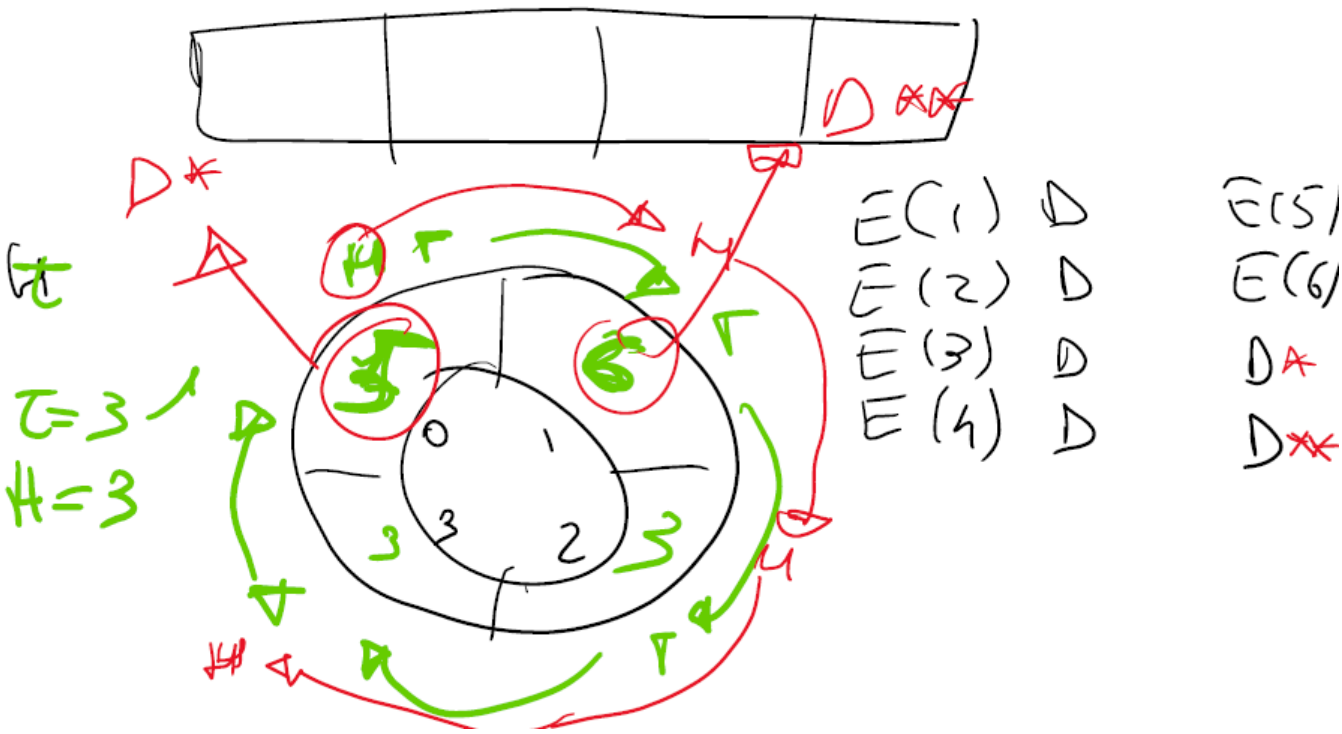
Pertanto anche quello dovrà essere occupato per considerare l'array pieno.

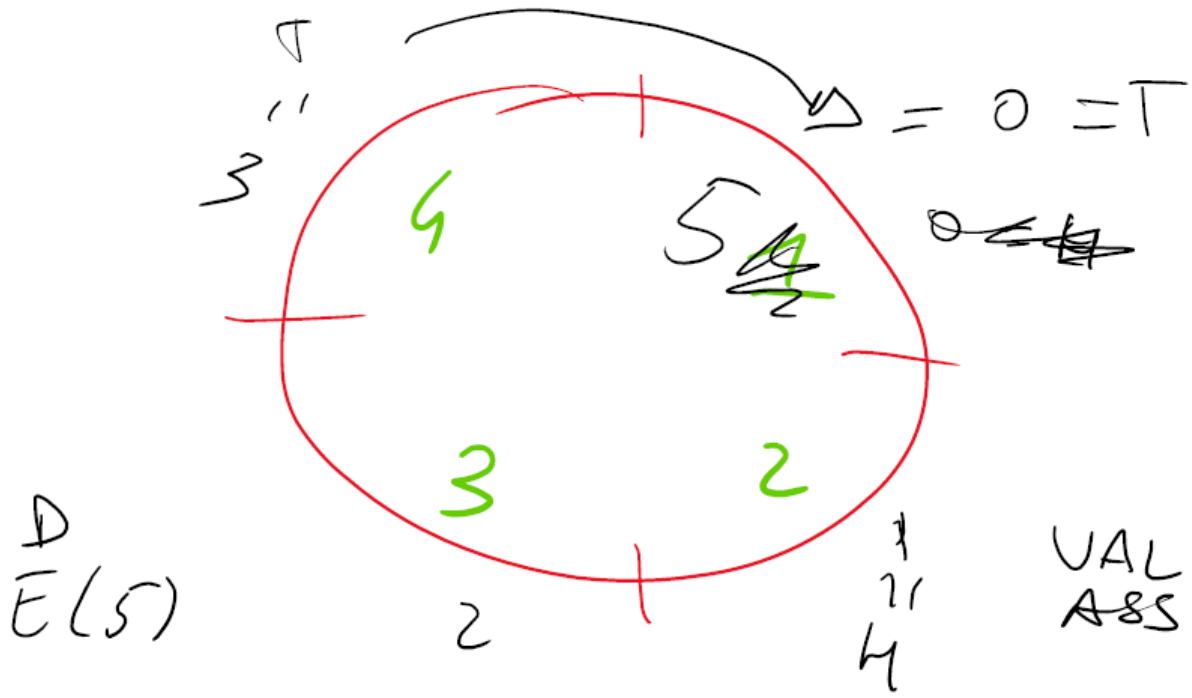
Per far ciò "torniamo indietro" con l'aiuto del `%`.



E aggiungiamo - quindi - un nuovo valore.

Esempio grafico del prof




= 0 = T
4
5
3
2
D
E(5)
2
VAL
ASS