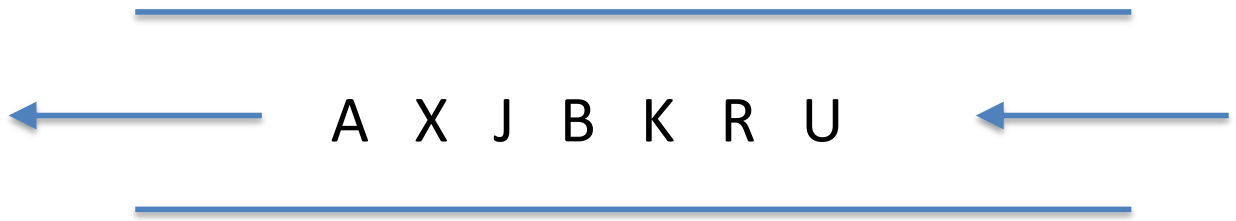
Colas

La clase queue

Tres operaciones básicas:

- Añadir un nuevo elemento (entrar, push)
- Eliminar el primer elemento que ha entrado (salir, pop)
- Ver quién es el primer elemento (primero, front)



Especificación de la clase queue

```
template <class T> class queue {
    public:
    // Constructoras

    // Pre: cert
    // Post: crea una cola vacía
    queue ();

    // Pre: cert
    // Post: crea una cola que es una copia de q
    queue (const queue& q);

    // Destructora
    ~queue();
```

```
// Modificadoras

/* Pre: La cola es [a1,...,an], n ≥ 0 */
/* Post: Se añade el elemento x como último de la cola,
   es decir, la cola será [a1,...,an,x] */
void push(const T& x);

/* Pre: La cola es [a1,...,an] y no está vacía (n>0) */
/* Post: Se ha eliminado el primer elemento de la cola
   original, es decir, la cola será [a2,...,an] */
void pop ();
```

```
// Consultoras

/* Pre: la cola es [a1,...,an] y no está vacía (n>0) */
/* Post: Retorna a1 */
T front() const;

/* Pre: cert */
/* Post: Retorna true si y solo si la cola está vacía */
bool empty() const;

private:
...

};
```

Laberinto

- Laberintos rectangulares: mxn posiciones
- Cada posición (i,j) puede estar libre o ser una pared
- Dadas dos posiciones ini y fin, queremos saber si hay un camino de ini a fin
- Una posición (i,j) es válida si $1 \leq i \leq m$ y $1 \leq j \leq n$
- Un camino de ini a fin es una secuencia de posiciones válidas, libres y adyacentes siendo ini la primera posición y fin la última.
- Todas las posiciones, salvo las del borde, tienen cuatro posiciones adyacentes: norte, sur, este y oeste.

La clase laberinto

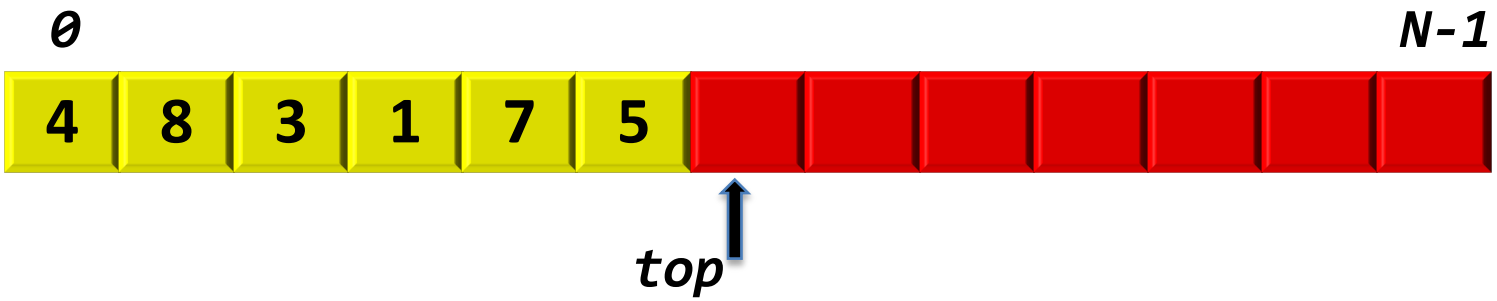
- Operaciones:
  - `marcar(p)`: deja una marca en la posición `p`.
  - `libre(p)`: retorna `true` si `p` es válida y está libre y `false` en caso contrario.
  - `marcada(p)`: retorna `true` si `p` está marcada y `false` en caso contrario.
  - …
- La clase `pos` representa un par `(fila, columna)` y tiene una constructora `pos(i,j)`, las consultoras `fila` y `col`, `es_igual` para comparar, etc.

```
// Pre: L.libre(ini), L.libre(fi) y todas las
// posiciones válidas de L están sin marcar
bool busca_cami(const Laberinto& L, pos ini, pos fin) {
    queue<pos> Q;
    Q.push(ini); L.marcar(ini);
    bool encontrado = false;
    while (not Q.empty() and not encontrado) {
        pos p = Q.front(); Q.pop();
        if (p.es_igual(fin)) encontrado = true;
        else {
        ...
        }
    }
// encontrado==true si y solo si hay un camino
// entre 'ini' y 'fin'
return encontrado;
}
```

```
// p = (i,j) != fi
pos norte(p.fila()-1, p.col());
if (L.libre(norte) and not L.marcada(norte)) {
   Q.push(nort); L.marcar(norte);
}
pos este(p.fila(), p.col()+1);
if (L.libre(este) and not L.marcada(este)) {
   Q.push(este); L.marcar(este);
}
pos sur(p.fila()+1, p.col());
if (L.libre(sur) and not L.marcada(sur)) {
   Q.push(sur); L.marcar(sur);
}
pos oeste(p.fila(), p.col()-1);
if (L.libre(oeste) and not L.marcada(oeste)) {
   Q.push(oeste); L.marcar(oeste);
}
```

Implementaciones con vectores

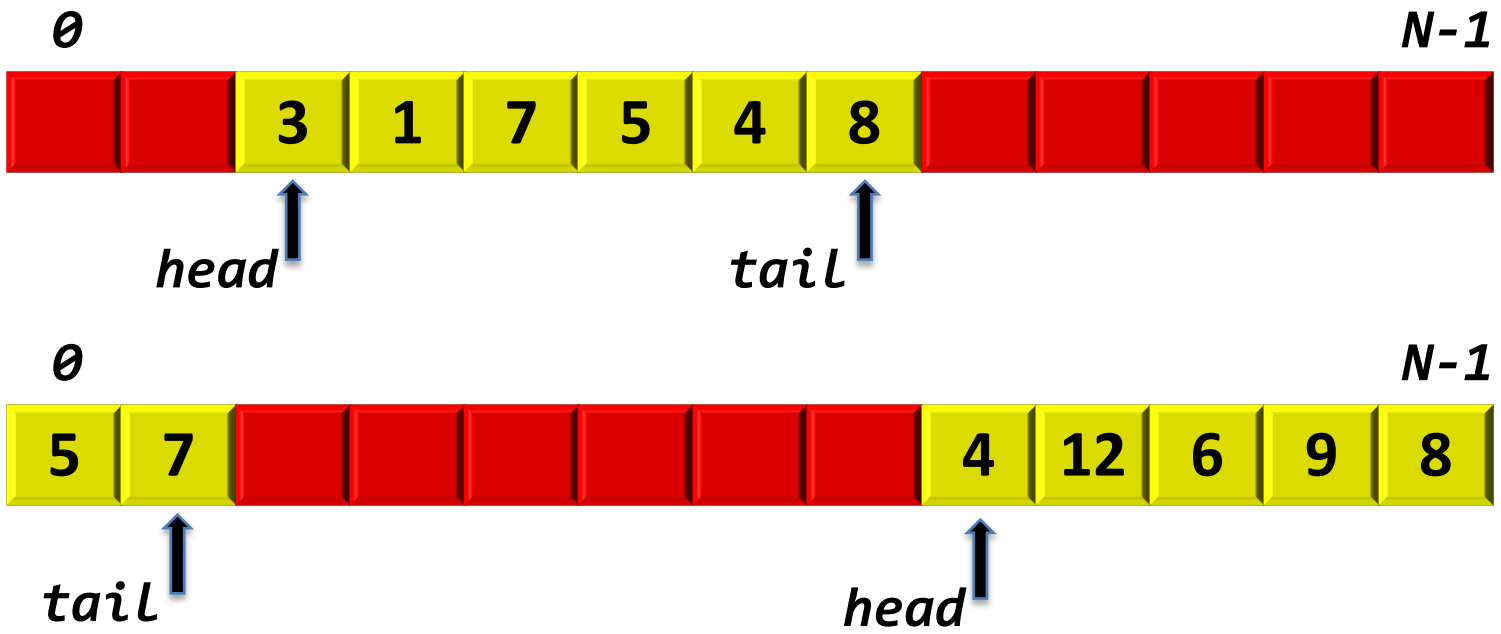
Implementación de pilas con vectores




```
template <class T> class stack {
public:
...

private:
vector <T> elems;
int top;
static const int MAX_SIZE = 100;
};
// Invariante: 0 ≤ cim ≤ elems.size() = MAX_SIZE
```

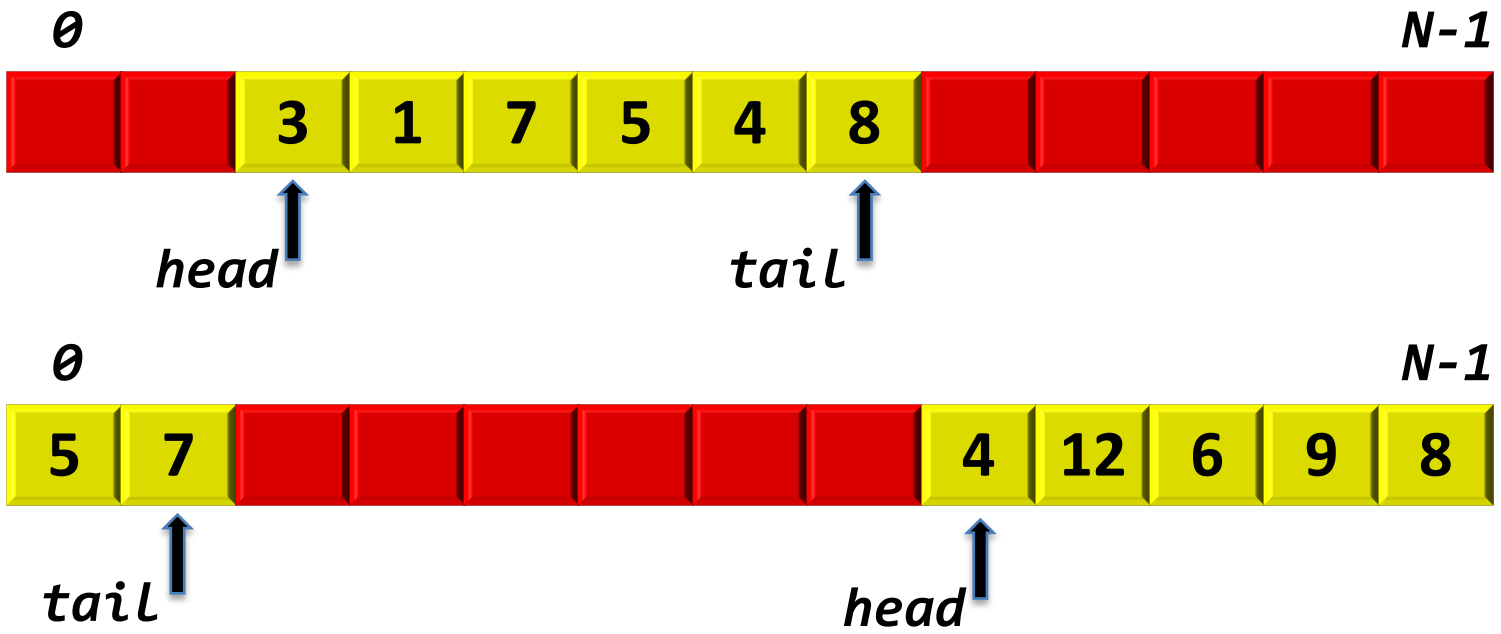
```
stack () {top = 0;
elems = vector<T>(MAX_SIZE);
};

void push(const T& x) {
      if (top == MAX_SIZE) ... // ERROR: pila llena
      elems[top] = x;
      ++top;
}

void pop() {
      if (top == 0) ... // ERROR: pila buida
      --top;
}

T top() const {
if (top == 0) ... // ERROR: pila buida
return elems[top - 1];
}

bool empty const {
      return top == 0;
}
```

Implementación de colas circulares con vectores



- `push: tail = (tail+1) % N`
- `pop: head = (head+1) % N`
- `size: (tail-head+1) % N`

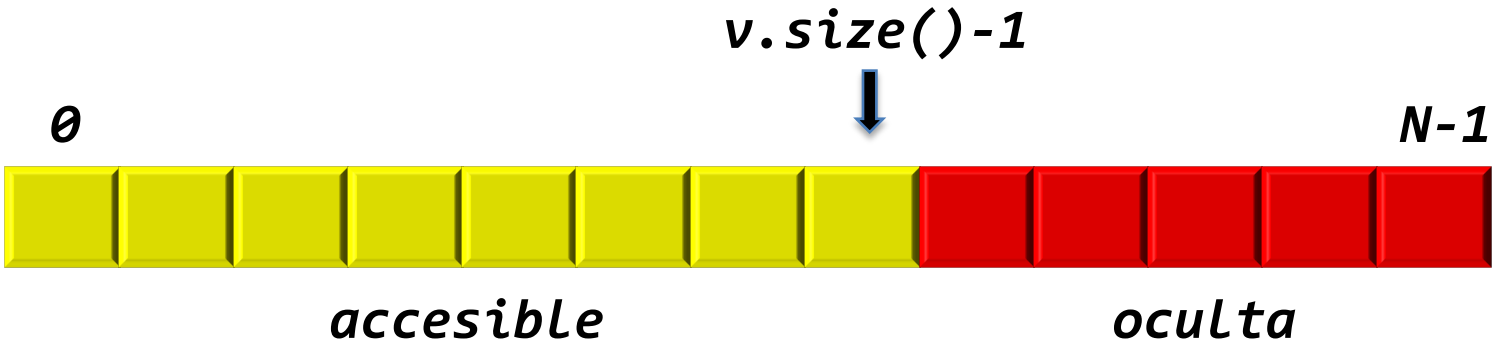
Implementación de colas *circulares* con vectores



- **`push: tail = (tail+1) % N`**
- **`pop: head = (head+1) % N`**
- **`size: (tail-head+1) % N`**
- Ambigüedad cuando **`tail = head-1`**. Se puede añadir un contador de elementos o no llenar nunca el vector (**`size < N`** siempre)

La operación push_back

- La operación **`v.push_back(e)`** añade el elemento **`e`** al final de **`v`** y **`v.size()`** se incrementa en 1.
- Implementación más obvia:
  - si la posición que ocuparía **`e`** está libre, se ocupa
  - en caso contrario, se mueve todo el vector a otra zona de la memoria
  - Muy ineficiente si hemos de mover el vector cada vez que hacemos un push_back
  - 1+2+...+(n-1) = N*(N-1)/2 movimientos para tener un vector de tamaño N



- Si queda algún sitio libre en la parte oculta, se ocupa, si no, se mueve a una zona libre de tamaño 2*v.size()
- Si el tamaño inicial es 1, para tener un vector de tamaño N, habriamos movido los siguientes elementos:

$$1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1}-1$$

siendo $2^k < N \leq 2^{k+1}$

- Como mucho movemos 2*N posiciones por cada push_back

Listas

Listas

Las listas son estructuras lineales que permiten:

- Recorridos secuenciales de sus elementos.
- Insertar elementos en cualquier punto.
- Eliminar cualquier elemento
- Concatenar una lista a otra.

Contenedores e iteradores

- Un *contenedor* es una estructura de datos (un template) para almacenar objetos.
- Las listas son casos particulares de contenedores.
- Un *iterador,* es un objeto que designa un elemento de un contenedor para desplazarnos por él.

Iteradores: declaración (instanciación)

- Método **`begin()`**
- Método **`end()`**
- **`list<Estudiant>::iterator it = l.begin();`**
- **`list<Estudiant>::iterator it2 = l.end();`**
- "::": La definición del iterador pertenece al contenedor
- Si una llista `l` está vacía, entonces **`l.begin() = l.end()`**

Operaciones con iteradores

- `it1 = it2;`
- `it1 == it2, it1 != it2`
- `*it (si it != l.end() )`
- `++it, --it (salvo si estamos en l.begin o en l.end() )`
- **NO**: `it + 3, it1 + it2, it1 < it2, ...`

Esquema frecuente

```
list<T> l;

...

list<T>::iterator it = l.begin();

while (it != l.end() and not
  cond(*it)) {

    ... acceder a *it ...

++it; }
```

Iteradores constantes

Los iteradores constantes prohiben modificar el objeto referenciado por el iterador. Por ejemplo:

```
list<Estudiant>::const_iterator it1;

list<Estudiant>::iterator it2 = l.end();
```

Estaría prohibido:

```
*it1 = ...;
```

pero no:

```
it1 = it2;

*it2 = ...;
```

```
void imprimir_llista(const list<Estudiant>& L) {
for(list<Estudiant >::const_iterator it = L.begin();
     it != L.end(); ++it)
  (*it).escriure();
}
```

La clase Lista

```
template <class T> class list {
public:
// Subclases de la clase lista
class iterator { ... };
class const_iterator { ... };
// Constructoras
list();
list(const list & original);
// Destructora:
~list();
```

```
// Modificadoras

void clear();

void insert(iterator it, const T& x);

iterator erase(iterator it);

void splice(iterator it, list& l);


// Consultoras

bool empty() const;

int size() const;
```

Suma de los elementos de una lista de enteros

```
/* Pre: true */
/* Post: El resultado es la suma de los elementos de l */
int suma(const list<int>& l) {
    int s = 0;
    for (list<int>::const_iterator it = l.begin();
                it != l.end(); ++it){
        s = s + *it;
    }
    return s;
}
```

Búsqueda en una lista de enteros

```
/* Pre: true */
/* Post: El resultado indica si x está o no en l */
bool pertenece(const list<int>& l, int x) {
    list<int>::const_iterator it = l.begin();
    while ((it != l.end()) and (*it != x))
        ++it;
    return it != l.end();
}
```

Suma k a todos los elementos de una lista

```
/* Pre: l=[x1,...,xn] */
/* Post: l=[x1+k,x2+k,...,xn+k] */
void suma_k(list<int>& l, int k) {
    list<int>::iterator it = l.begin();
    while (it != l.end()) {
        *it += k;
        ++it;
    }
}
```

Inserción en una lista ordenada

```
/* Pre: L=[x1,...,xn], está ordenada */
/* Post: L contiene a x, x1,...,xn, y está ordenada */
void inserc_ordenada(list<int>& L, int x) {
    list<int>::iterator it = L.begin();
    while (it != L.end() and (x > *it) ++it;
    L.insert(it,x);
}
```

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

```
/* Pre: L1=[x1,...,xn], L2=[y1,...,ym] y las dos listas
    están ordenadas */
/* Post: L1 contiene x1,...,xn,y1,...,ym y está ordenada */
void inserc_ordenada(list<int>& L1, const list<int>& L2) {
    list<int>::iterator it1 = L1.begin();
    list<int>::iterator it2 = L2.begin();
    while (it1 != L1.end() and it2 != L2.end() ) {
        if (*it1 < *it2) ++it1;
        else {L1.insert(it1,*it2); ++it2;
    }
    while (it2 != L2.end() ) {
        L1.insert(it1,*it2);
        ++it2;
    }
}
```

Vectores vs Listas

- Recorrido secuencial: tiempo lineal en los dos casos
- Acceso directo al i-ésimo elemento: constante en vectores, tiempo i en listas.
- Insertar un elemento es
  - constante en listas
  - Al final de un vector es constante (si no hay que reservar memoria adicional
  - En medio de un vector es costoso
- Borrar un elemento es constante en listas y costoso en vectores
- Splice: constante en listas y costoso en vectores