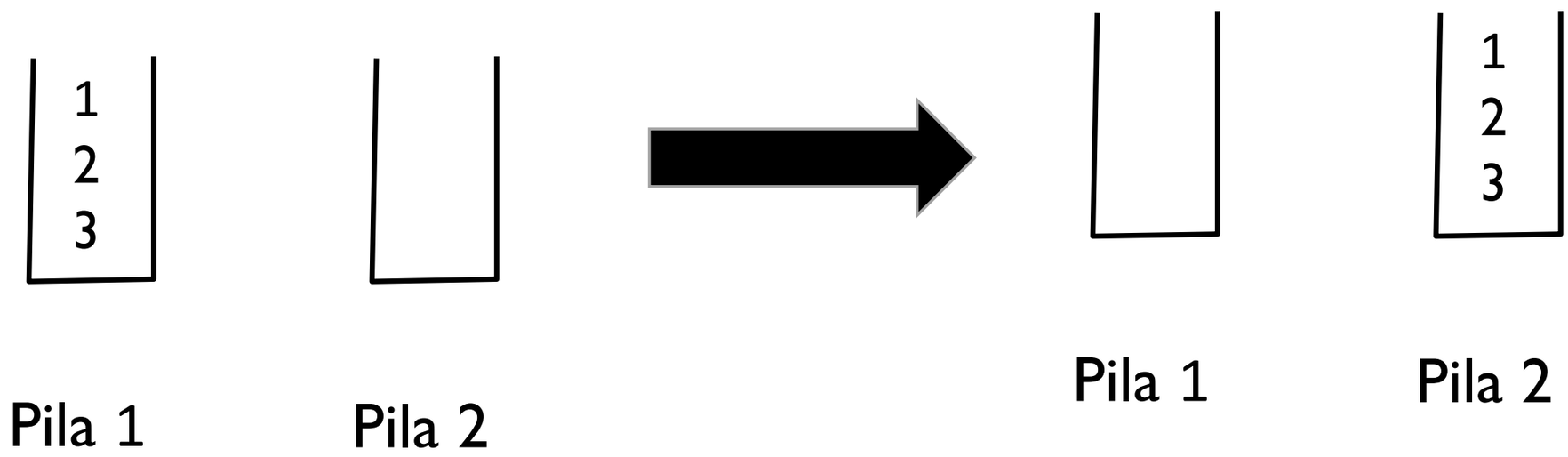
UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

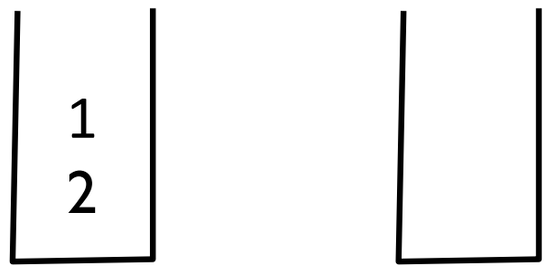
Pensemos como mover una pila de enteros a otra que está vacía y que queden sus elementos en el mismo orden, Solo contamos con dos pilas de enteros y algunas variables enteras (pensar en recursividad...):



UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

```
// Mueve pila p1 a p2 y los elementos quedan en el mismo orden
// que en p1
void mover(Pila &p1, Pila &p2){
    int x;
    if (esVacia(p1)) return;
    x = tope(p1);
    desempilar(p1);
    mover(p1,p2);
    empilar(p2,x);
}
```

corramos el algoritmo con pila p1 con dos elementos



```
void mover(Pila &p1, Pila &p2){
    int x;
    if (estaVacia(p1)) return;
    x = tope(p1);
    desempilar(p1);
    mover(p1,p2);
    empilar(p2,x);
}
```

¿ Cómo hacemos para Invertir una pila de enteros usando otra pila y algunas variables enteras, y recursión?

Utilizamos el algoritmo anterior ¿Cómo?

Ejercicio para la casa:

Haga un algoritmo recursivo para agregar un elemento x al final de una pila y utilícelo en otro algoritmo recursivo que invierta una pila.

Podemos **implementar una pila PILA** mediante un arreglo dinámico o mediante un vector utilizando las funciones de un vector, o mediante una lista doblemente enlazada.

Hagamos una implementación con un vector donde el tope es el último elemento del vector

Archivo Pila.h:

```
// Implementando una Pila de
// enteros con un vector
#include <iostream>
#include <vector>
using vector;

// El tipo Pila
typedef  vector<int>  Pila;
```

```
// Empila data en la pila v
void empilar(Pila & v, int data)

// Desempila el tope de la pila v
// La pila no puede estar vacía
void desempilar(Pila & v)

// Devuelve el tope de la Pila v
// la pila no debe estar vacía
int tope(const Pila & v)

//Verificar si la pila v está vacía
bool esVacia(const Pila & v)

void imprimir(const Pila & v)
```

Archivo Pila.cpp:

```
// Implementando una Pila de
// enteros con un vector
#include "Pila.h"

// Empila data en la pila v
void empilar(Pila &v , int data)
{
    v.push_back(data);
}
```

```
// Desempila el tope de la pila v
// La pila no puede estar vacía
void desempilar(Pila &v)
{  // la pila no debe estar vacía
    v.pop_back();  // función de vector
}

// Devuelve el tope de la Pila v
// la pila no debe estar vacía
int tope(const Pila &v)
{
    return v.back();
}
```

```
// Verificar si la pila v está vacía
bool esVacia(const Pila &v) {
   return v.empty();
}

void imprimir(Pila &v){
    cout << "El tope es el ultimo “
            << " elemento: ";
    for(auto elem:v)
      cout << elem << " ";
    cout << endl;
}
```

```
int main()
{  // uso de Pila
   Pila miPila;
   // empilar elementos
   empilar(miPila,10);
   empilar(miPila,20);
   // desempilar elemento
   if (!esVacia(miPila)) desempilar(miPila);
   // devolver el tope de la pila si no es vacia
   if (!esVacia(miPila)) {
      int top = tope(miPila);
      cout << "\nEl tope es: " << top << endl;
   }
   imprimir(miPila);
   return 0;
}
```

Si implementamos **Pila** con **un vector** las operaciones **empilar** y **desempilar** las hacemos al final del vector y así es un número constante de operaciones.

Si implementamos **Pila** con una **lista doblemente enlazada con centinelas** podemos tomar el primero de la lista (o el último) como el tope, Y también el número de operaciones sería constante.

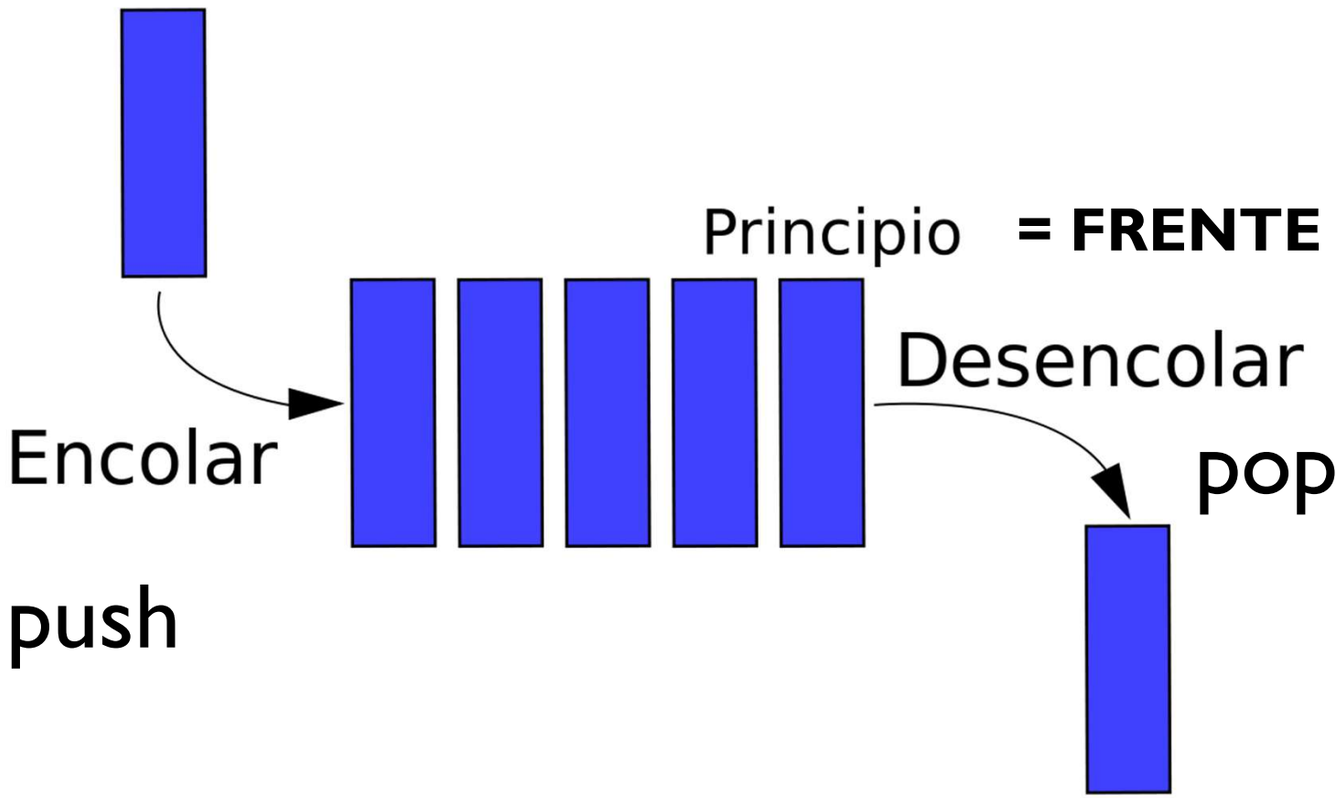
Ejercicios para la casa:

1) Implementar Pila con Lista doblemente enlazada con centinelas. El tipo Lista doblemente enlazada con centinelas deberán implementarlo primero con .h y .cpp y luego el tipo Pila con su .h y su .cpp y hacer pruebas

2) Hacer un programa C++ para evaluar una expresión aritmética postfija leída por teclado, los operandos (que serán números) y operadores van separados por un espacio. Suponga que la guarda en un vector de strings. Debe utilizar el tipo Pila en su programa.

El Tipo de dato abstracto COLA

Una Cola es una secuencia con el comportamiento: primero en entrar primero en salir. First in First out (FIFO)



Ejemplo:

Si la cola es la secuencia <7,3,7,5,4,2> supongamos que el frente es el primero

Desencolar resultaría en <3,7,5,4,2>

Desencolar resultaría en <7,5,4,2>

Encolar 8 resultaría en <7,5,4,2,8> coloca a 8 de último en la cola

El tipo de dato abstracto **COLA** (**Queue** en inglés) es una secuencia de objetos del mismo tipo **T** con las operaciones siguientes:

- **boolean esVacia** ( ) : devuelve verdad si la cola está vacía
- **void encolar**(T x): coloca de último en la cola al elemento x
- **void desencolar** ( ): elimina el frente de la cola
- **T frente** ( ): devuelve el frente de la cola
- **Cola crearVacia** ( ): que convierte una cola existente en vacía
- **int numElem**( ): devuelve el número de elementos de la cola

Si utilizamos **un arreglo ó vector** para implementar una **cola** y el frente de la cola es el primero del arreglo:

¿Qué orden del número de operaciones elementales tendrían desencolar y encolar en una cola con N elementos?

Desencolar: del orden del número de elementos de la cola

Encolar: seria de orden constante pues lo colocamos de último en el arreglo

Si utilizamos **un arreglo** para implementar una **cola** y el frente de la cola es el último del arreglo:

¿Qué orden tendrían desencolar y encolar en una cola con N elementos?

Si utilizamos **una lista doblemente enlazada con centinelas cabeza y cola** para implementar una cola entonces el número de operaciones sería una constante.

Mucho mas eficiente que arreglos o vectores!!

Una implementación de Cola con Lista doblemente enlazada con centinelas:
Recordemos el listaDoble.h de Lista_dob_enl con las funciones que vimos:

```
struct Nodo {
  int data;
  Nodo *anterior;
  Nodo *proximo;
};
struct Lista_dob_enl{
  Nodo* cabeza = nullptr;
  Nodo* cola = nullptr;
  int num_elem=0;
} ;
```

```
Lista_dob_enl      crear_lista_vacia();

void insertar(Lista_dob_enl &, int , int pos);

void imprimir(const Lista_dob_enl &);

void eliminar(Lista_dob_enl &, int pos);

bool esVacia(const Lista_dob_enl &);
```

Una implementación de Cola con Lista doblemente enlazada con centinelas:

Primero el archivo header **cola.h**

```
#include "listaDoble.h"
typedef Lista_dob_enl Cola;

Cola crearVacia();
int frente(const Cola& cola);
void desencolar(Cola& cola);
void encolar(Cola& cola, int elem);
int num_elem(const Cola& cola);
void imprimir_cola(const  Cola &);
// al ser Cola una Lista_dob_enl no hace falta definir esVacia()
// porque esta función ya está en la definición del tipo Lista_dob_enl
```

Ahora el archivo cola.cpp

```
#include "cola.h"
Cola crearVacia(){
    return crear_lista_vacia();
}
int frente(const Cola& cola){
    // el frente es el primero de la lista
    // se supone que la cola no esta vacia
    return cola.cabeza->proximo->data;
}
void desencolar(Cola& cola){
    // elimina el frente de la cola
    // se supone que la cola no esta vacia
    eliminar(cola, 0);
```

```
void encolar(Cola& cola, int elem){
    // lo inserta de ultimo en la lista
    insertar(cola, elem, cola.num_elem);
}
int num_elem(const Cola& cola){
    return cola.num_elem;
}
void imprimir_cola(const Cola& cola){
    imprimir(cola);
} // utilizamos directamente
  // imprimir de Lista_dob_enl
```

¿De qué orden es el número de operaciones de cada función? Hagámoslo juntos

Encolar podria ser de orden constante...

```
void encolar(Cola& cola, int elem){
    // lo inserta de ultimo en la lista
    insertar(cola,   elem,   cola.num_elem);
}
```

¿Cómo haríamos?...
Conocemos la referencia a la cola

Hagámoslo juntos....

Recordemos la función insertar de lista doblemente enlazada (busca primero la posición):

```
void insertar( Lista_dob_enl lista,    int x,    int i )

{  if ((i<0)&&(i> lista.num_elem) return;
   Nodo* p  = obtenerRef (lista, int i) ;
   Nodo* nuevo_nodo = new Nodo{ x, p.anterior, p };
   (nuevo_nodo.anterior)  ->  siguiente = nuevo_nodo;
   p.anterior = nuevo_nodo;
   num_elem ++;  // se suma 1 al número de elementos de la
lista
}
```

```
void encolar(Cola& cola, int elem){
        Nodo* p = cola.cola;
        Nodo* nuevo_nodo = new Nodo{ elem, p->anterior, p };
        (nuevo_nodo->anterior)  ->  proximo = nuevo_nodo;
        p->anterior = nuevo_nodo;
        cola.num_elem ++;
 }
```

Lo conceptualmente correcto es tener una función insertar_cola en el tipo List_dob_enl, pues el tipo Cola no se debería enterar cómo se implementa List_dob_enl

Implementación de una cola de enteros con un vector, el último elemento del vector es el frente:

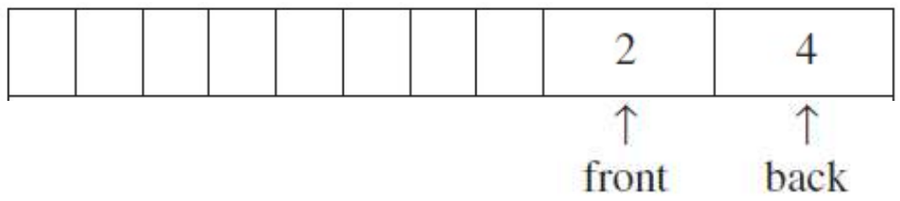
```
#include <vector>
using std::vector;
typedef vector<int> Cola;

void encolar(Cola& cola, int val){
    cola.insert(cola.begin(), val);
}
void desencolar(Cola& cola){
    cola.pop_back();
}
```

```
void imprimir(Cola& cola){
    for (auto elem:cola){
        std::cout << elem << " " ;
    }
    std::cout << std::endl;
}
int frente(const Cola& cola){
   return cola[cola.size()-1];
}
bool esVacia(const Cola& cola){
    return cola.empty();
}
```

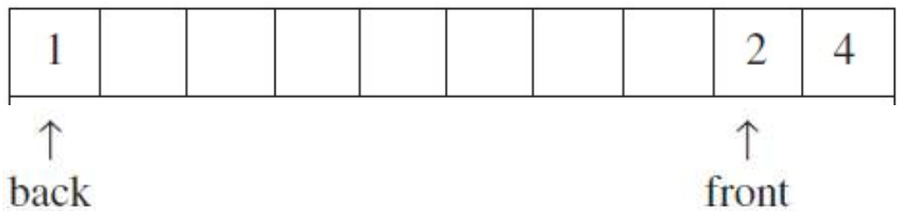
¿De qué orden es el número de operaciones de cada función? Hagámoslo juntos

Existe una implementación del tipo cola utilizando **un arreglo y donde todas las operaciones son de orden constante**. Es **un arreglo circular**, manteniendo un índice **front** que apunta al frente de la cola y un índice **back** que apunta al último de la cola. Los elementos de la cola van desde frente hacia la derecha hasta llegar circularmente al último

Estado de la cola en algún momento



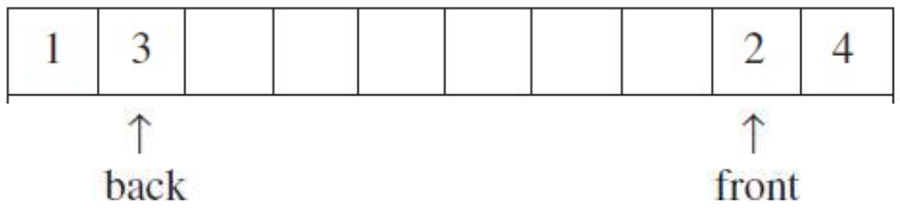
Encolamos 1



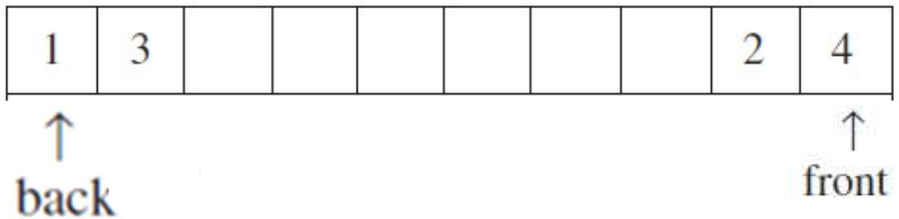
¿Qué significa circularmente?

Si el arreglo tiene, por ejemplo 10 elementos (sus posiciones son de 0 a 9), el siguiente circularmente de 9 será (9 +1)%10 = 0, el siguiente del siguiente de 9 será (9+2)%10 =1

Encolar 3

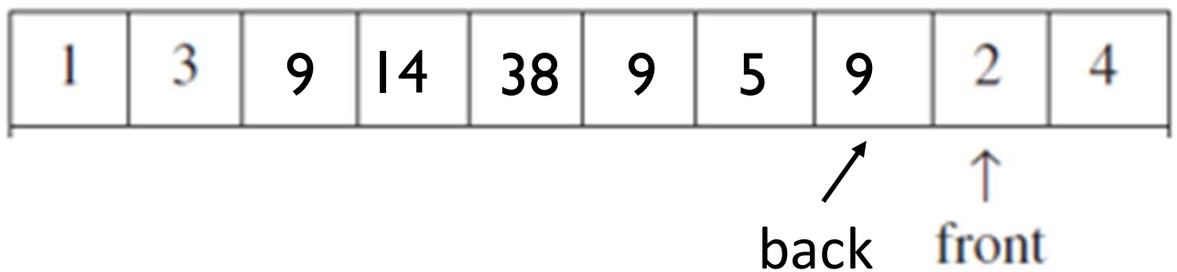


Desencolar



Hay que mantener una variable **numElem** con el tamaño de la cola porque **no podemos representar a la cola vacía, así cola vacía será tam = 0 y back y front en la posición cero (hay que distinguir cola vacía)**

Y cola llena es cuando back + 1 circularmente sea igual a front



La capacidad de la cola está limitada al número de elementos del arreglo

¿Cómo sería la estructura de datos para representar una cola con un arreglo (vector) circular de 100 elementos ?

```
Struct Cola {
    int numElem =0;
    int front =0;
    int back = 0;
    vector<int>  v(100);
}
```

Corramos paso a paso y con dibujos las siguientes operaciones de Cola en un arreglo circular de 4 elementos:

Comenzamos con una cola vacía: numElem=front=back=0

encolar 1, encolar 2, encolar 3, encolar 4 desencolar, desencolar, encolar 5, encolar 6

un arreglo circular de 4 elementos
Comenzamos con una cola vacía: numElem=front=back=0
encolar 1, encolar 2, encolar 3, encolar 4 desencolar, desencolar, encolar 5, encolar 6

Y la función de **encolar**. **Se supone que la cola no está llena**. Habría que agregar una función que verifique la cola no está llena.

Protocolo:

void encolar (Cola& cola, int x)

Veámoslo juntos...

```
bool encolar  (Cola & cola, int x){
    if (cola.numElem==0) { cola.v[0]=x; cola.numElem++;
                           } // front y back no se modifican
    else {
        int b = (cola.back+1)%100;
        cola.v[b] = x;
        cola.back = b;
    }
    return true;
}
```

Ejercicio para la casa:

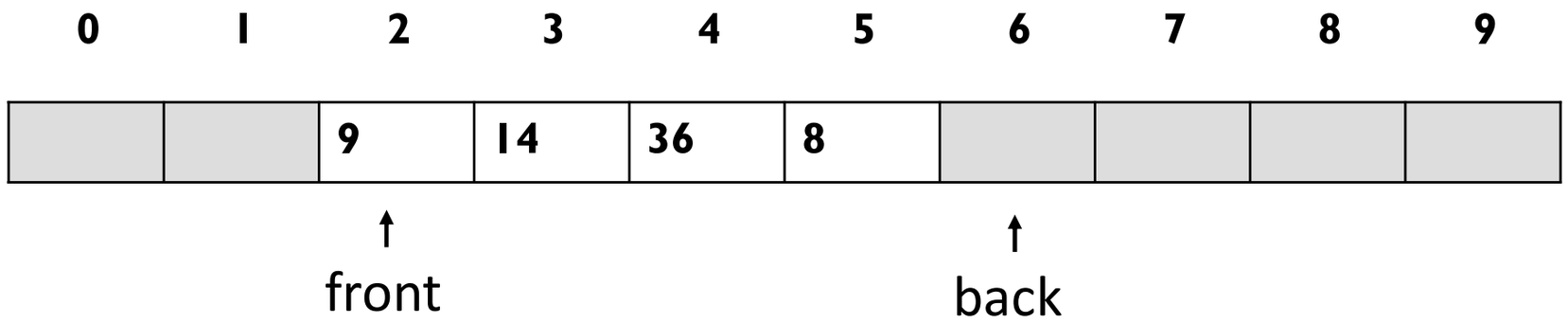
Y la función de **desencolar** ¿cómo sería?. Se supone que la cola no está vacía. Protocolo:

void desencolar (Cola cola&)

UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

```
void desencolar  (Cola cola&){
    if (cola.front==cola.back) { // tiene un solo elemento
                                 //se convierte en vacía
            cola.front = cola.back = cola.numElem=0;       }
    else  cola.front = (cola.front + 1)%100;
}
```

Si no queremos mantener el tamaño de la cola, hay otra representación circular que distingue entre cola llena y cola vacía, pero desperdicia un elemento del arreglo:

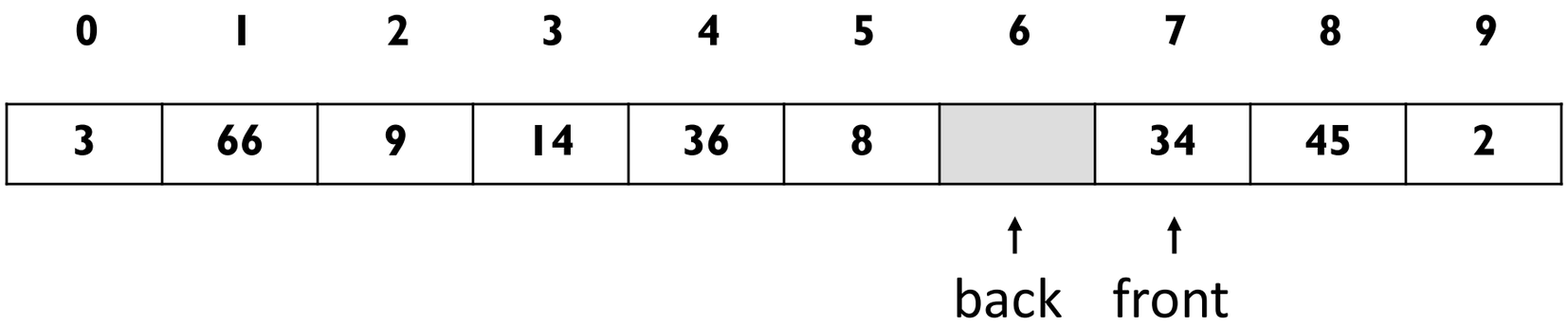


Note que back apunta a un elemento con basura (un centinela).

Note que back apunta a un elemento con basura.

Cola vacía sería front = back

Y **cola llena sería front = back+1** (circularmente (back+1)%10, pues 10 es el tamaño del arreglo)



Implementemos juntos el tipo Pila de enteros usando solo el tipo Cola de enteros. Suponga que el tipo Cola tiene una función que devuelve el número de elementos de la cola.

Typedef Cola Pila;

El frente de la cola será el tope de la pila (**que es una cola**)

Discutamos juntos la función **empilar**....

```
void empilar(Pila &pila, int x) {
    encolar(pila, x);  // Agrega el elemento a la cola.
    int numelem = numElem(pila);
    // Mueve los elementos de la cola para simular la pila
    for (int i = 1; i < numelem(pila) ; i++) {
       encolar(pila, frente(pila));  // Encola el frente de la cola.
       desencolar(pila);  // Desencola el frente de la cola.
    }
```

EJERCICIO

- IMPLEMENTAR EN C++ EL TIPO COLA UTILIZANDO UN ARREGLO CIRCULAR (EL QUE DISTINGUE COLA LLENA DE VACíA DESPERDICIANDO UN ELEMENTO) DE 100 ELEMENTOS. LLAME AL TIPO COLA_CIRCULAR. INCLUIR LA FUNCION QUE INVIERTE UNA COLA.

- IMPLEMENTAR PILA CON LISTA DOBLEMENTE ENLAZADA Y AGREGAR LA FUNCION QUE INVIERTA RECURSIVAMENTE UNA PILA (COMO HICIMOS)

PREGUNTAS???