Estructuras de datos: Pilas, Colas, Listas

Algoritmos

Dep. de Computación - Fac. de Informática
Universidad de A Coruña

Santiago Jorge
santiago.jorge@udc.es

UNIVERSIDADE DA CORUÑA

Índice

Referencias bibliográficas

- M. A. Weiss. Listas, pilas y colas. En *Estructuras de datos y algoritmos*, capítulo 3, páginas 45–92. Addison-Wesley Iberoamericana, 1995.
- R. Peña Marí. Implementación de estructuras de datos. En *Diseño de Programas. Formalismo y abstracción*, capítulo 7, páginas 257–290. Prentice Hall, segunda edición, 1998.
- G. Brassard y T. Bratley. Estructura de datos. En *Fundamentos de algoritmia*, capítulo 5, páginas 167–210. Prentice Hall, 1997.

Índice

Pilas

- Acceso limitado al último elemento insertado
- Operaciones básicas: `apilar`, `desapilar` y `cima`.
- Cada operación debería tardar una cantidad constante de tiempo en ejecutarse.
  - Con independencia del número de elementos apiladas.

Pseudocódigo: Implementación a base de vectores (i)

```
tipo Pila = registro
  Cima_de_pila : 0..Tamaño_máximo_de_pila
  Vector_de_pila : vector [1..Tamaño_máximo_de_pila]
                          de Tipo_de_elemento
fin registro

procedimiento Crear Pila ( P ) O(1)
  P.Cima_de_pila := 0
fin procedimiento

función Pila Vacía ( P ) : test O(1)
  devolver P.Cima_de_pila = 0
fin función
```

Pseudocódigo: Implementación a base de vectores (ii)

```
procedimiento Apilar ( x, P ) O(1)
  si P.Cima_de_pila = Tamaño_máximo_de_pila entonces
    error Pila llena
  sino
    P.Cima_de_pila := P.Cima_de_pila + 1;
    P.Vector_de_pila[P.Cima_de_pila] := x
fin procedimiento
función Cima ( P ) : Tipo_de_elemento O(1)
  si Pila Vacía (P) entonces error Pila vacía
  sino devolver P.Vector_de_pila[P.Cima de Pila]
fin función
procedimiento Desapilar ( P ) O(1)
  si Pila Vacía (P) entonces error Pila vacía
  sino P.Cima_de_pila := P.Cima_de_pila - 1
fin procedimiento
```

Código C: pilas.h

```
#define TAMANO_MAXIMO_PILA 10
typedef int tipo_elemento;
typedef struct {
  int cima;
  tipo_elemento vector[TAMANO_MAXIMO_PILA];
} pila;
void crear_pila(pila *);
int pila_vacia(pila);
void apilar(tipo_elemento, pila *);
tipo_elemento cima(pila);
void desapilar(pila *);
/* ERRORES: cima o desapilar sobre la pila vacía
            apilar sobre la pila llena */
```

Código C: pilas.c (i)

```
#include <stdlib.h>
#include <stdio.h>
#include "pilas.h"
void crear_pila(pila *p) {
  p->cima = -1;
}
int pila_vacia(pila p) {
  return (p.cima == -1);
}
void apilar(tipo_elemento x, pila *p) {
  if (++p->cima == TAMANO_MAXIMO_PILA) {
    printf("error: pila llena\n"); exit(EXIT_FAILURE);
  }
  p->vector[p->cima] = x;
}
```

Código C: pilas.c (ii)

```
tipo_elemento cima(pila p) {
  if (pila_vacia(p)) {
    printf("error: pila vacia\n");
    exit(EXIT_FAILURE);
  }
  return p.vector[p.cima];
}
void desapilar(pila *p) {
  if (pila_vacia(*p)) {
    printf("error: pila vacia\n");
    exit(EXIT_FAILURE);
  }
  p->cima--;
}
```

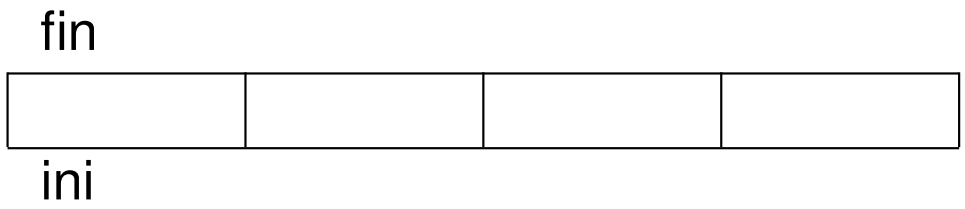
Índice

Colas

- Operaciones básicas: `insertar`, `quitarPrimero` y `primero`.
- Cada rutina debería ejecutarse en tiempo constante.
  - Con una implementación en base a punteros, resulta obvio; pero usando vectores, no es tan obvio.

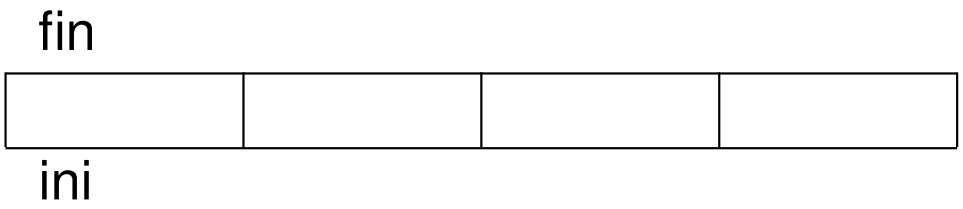
Problema de la implementación a base de vectores

1) Crear_Cola (C)

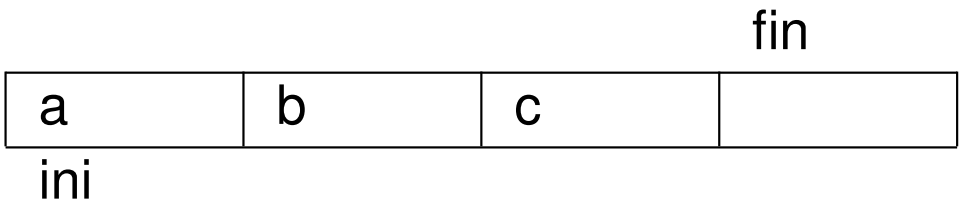


Problema de la implementación a base de vectores

1) Crear_Cola (C)

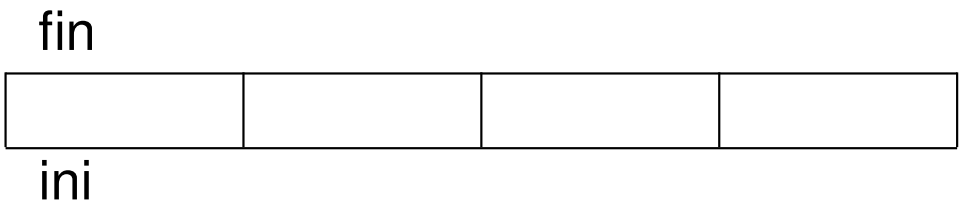


2) Insertar a, b, c

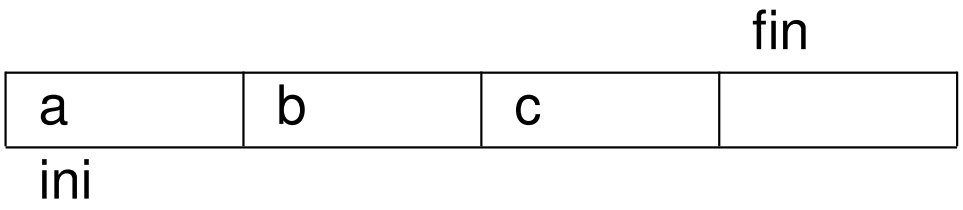


Problema de la implementación a base de vectores

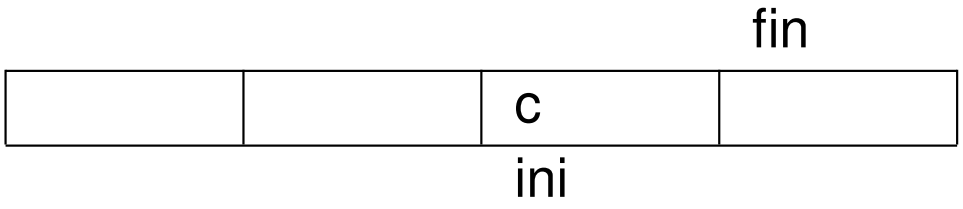
1) Crear_Cola (C)



2) Insertar a, b, c

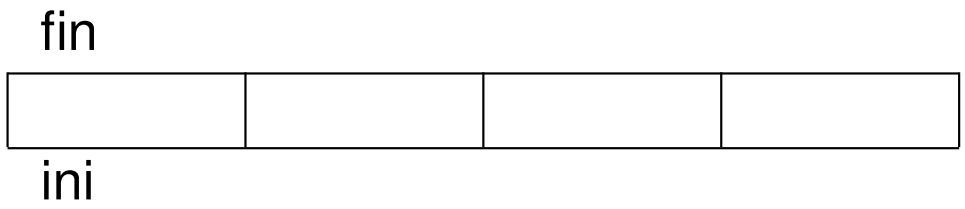


2) Quitar a, b

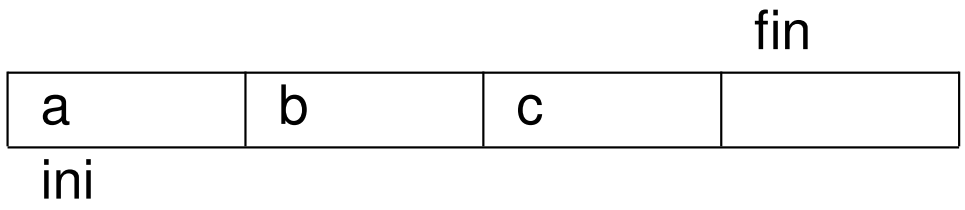


Problema de la implementación a base de vectores

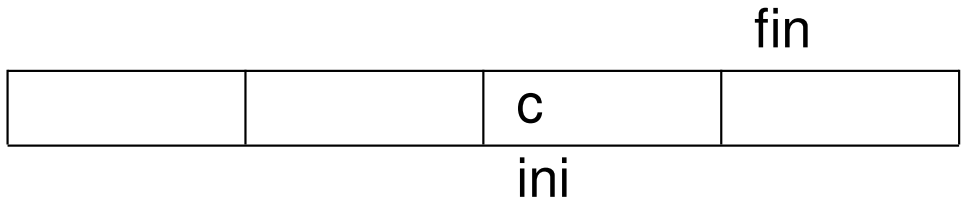
1) Crear_Cola (C)



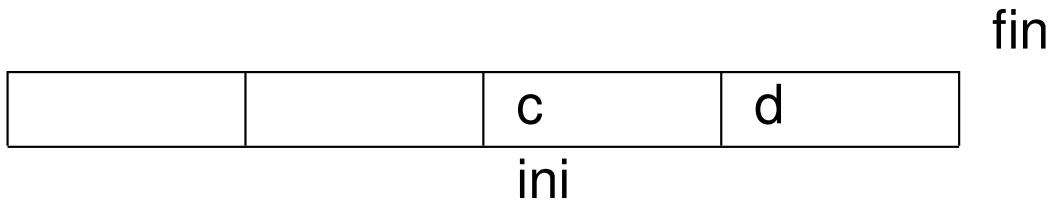
2) Insertar a, b, c



2) Quitar a, b

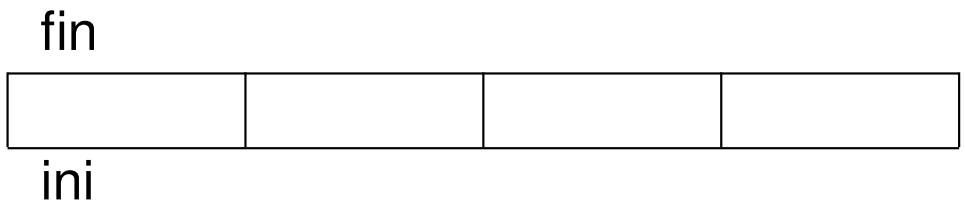


2) Insertar d

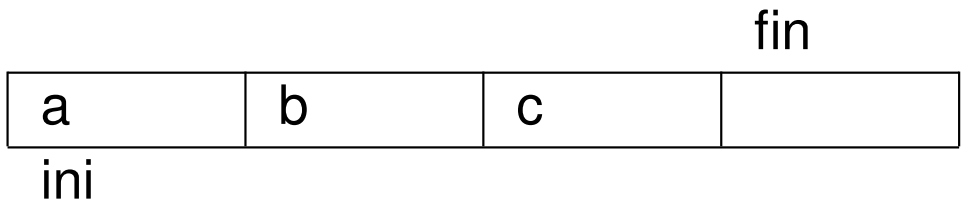


Problema de la implementación a base de vectores

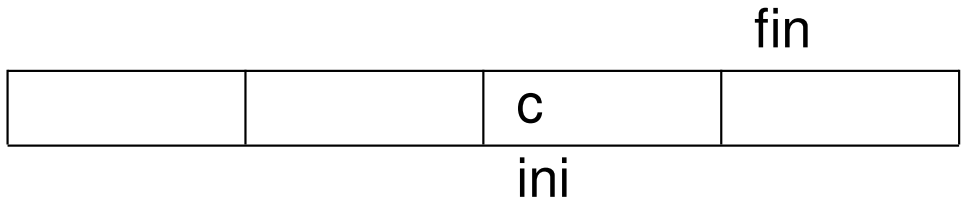
1) Crear_Cola (C)



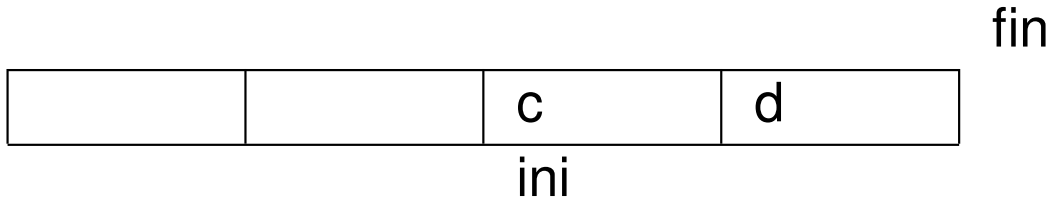
2) Insertar a, b, c



2) Quitar a, b



2) Insertar d

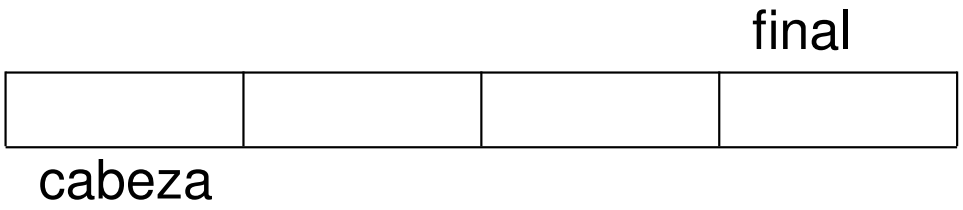


Ahora, `Insertar e` sería O(n) : tendríamos que desplazar `c` y `d`, tomando un número de pasos proporcional a la longitud del vector.

Implementación circular a base de vectores (i)

- La implementación circular devuelve cabeza y fin al principo del vector cuando rebasan la última posición.

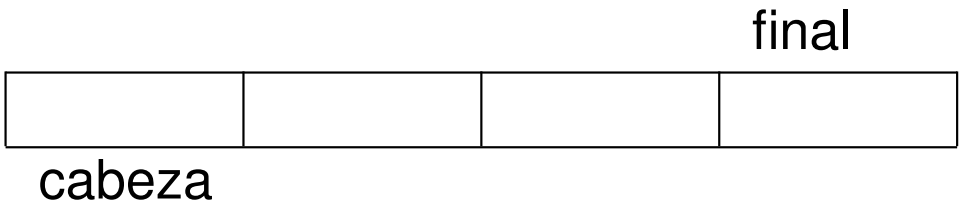
1) Crear_Cola (C)



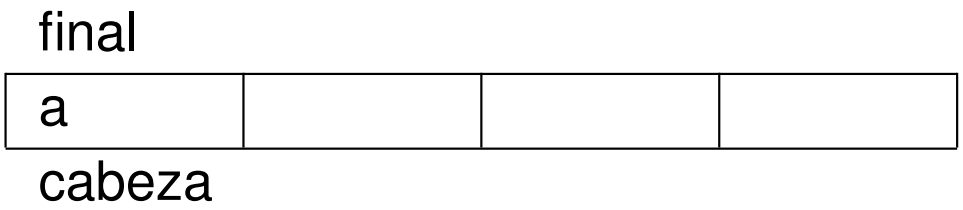
Implementación circular a base de vectores (i)

- La implementación circular devuelve cabeza y fin al principo del vector cuando rebasan la última posición.

1) Crear_Cola (C)



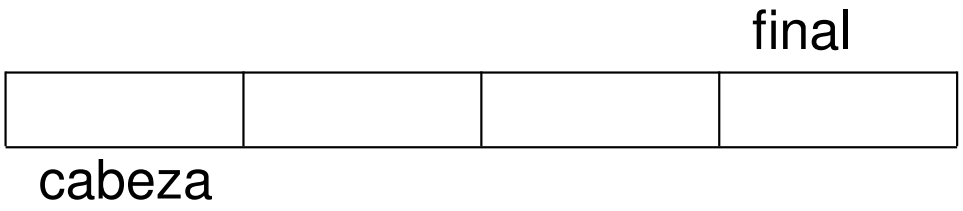
2) Insertar_en_Cola (a,C)



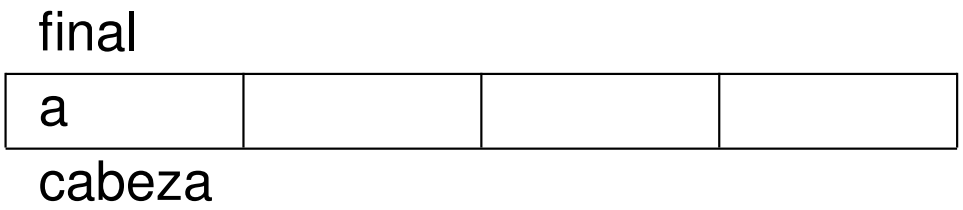
Implementación circular a base de vectores (i)

- La implementación circular devuelve cabeza y fin al principo del vector cuando rebasan la última posición.

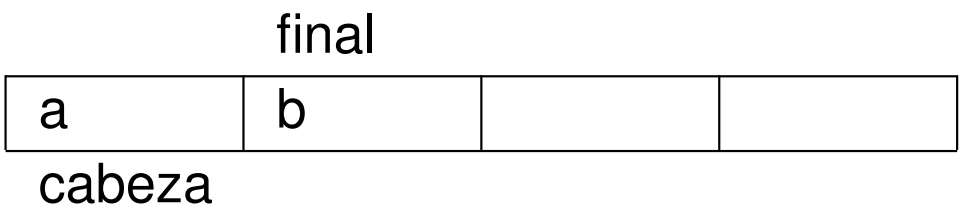
1) Crear_Cola (C)



2) Insertar_en_Cola (a,C)

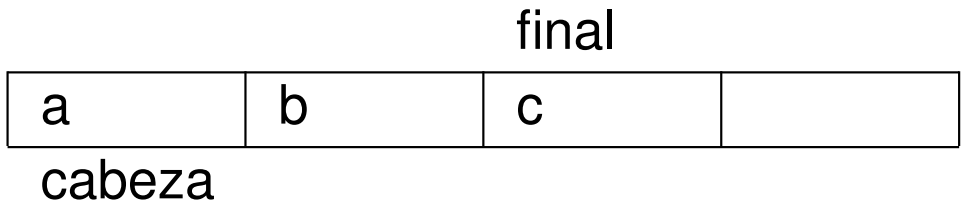


3) Insertar_en_Cola (b,C)



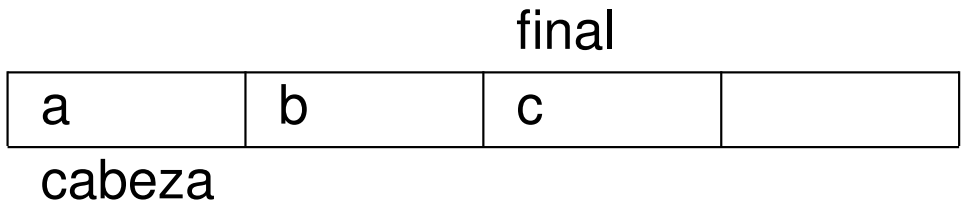
Implementación circular a base de vectores (i)

4) Insertar_en_Cola (c,C)

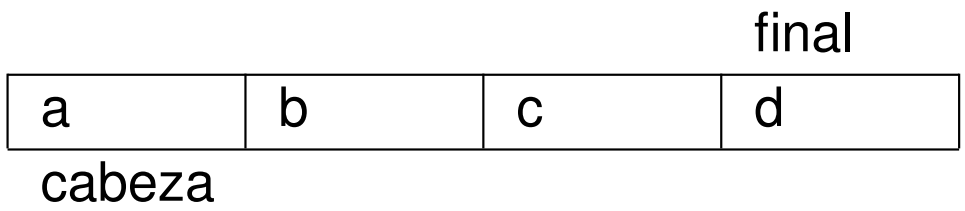


Implementación circular a base de vectores (i)

4) Insertar_en_Cola (c,C)

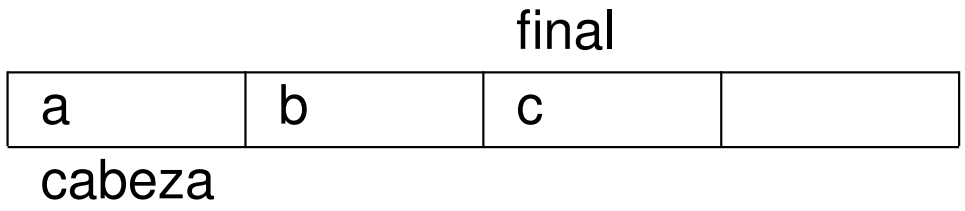


5) Insertar_en_Cola (d,C)

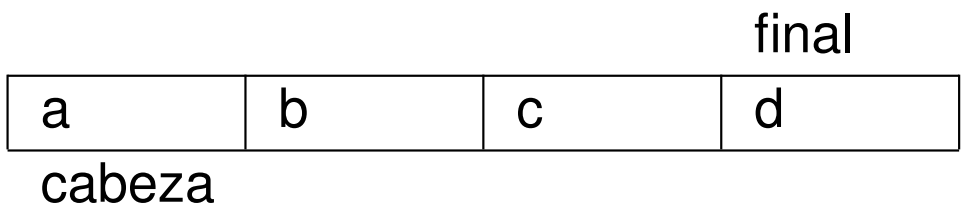


Implementación circular a base de vectores (i)

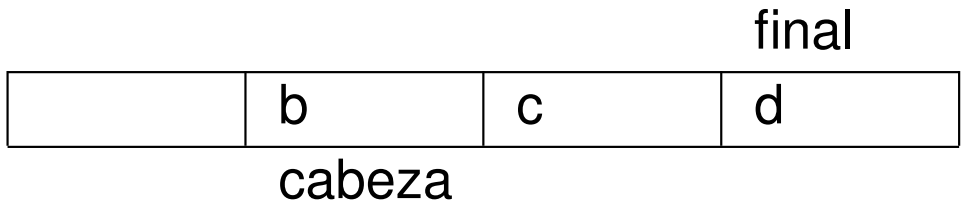
4) Insertar_en_Cola (c,C)



5) Insertar_en_Cola (d,C)

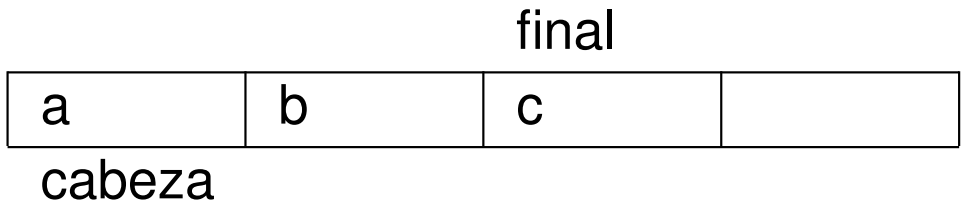


6) Quitar_Primero (C)

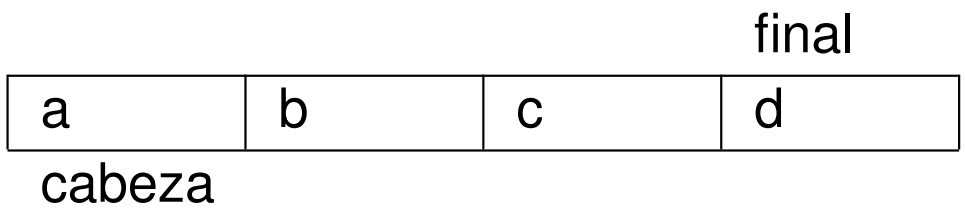


Implementación circular a base de vectores (i)

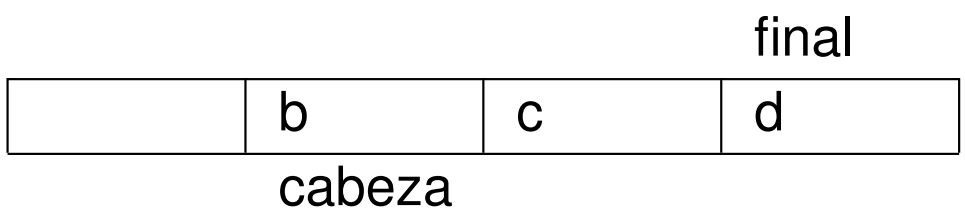
4) Insertar_en_Cola (c,C)



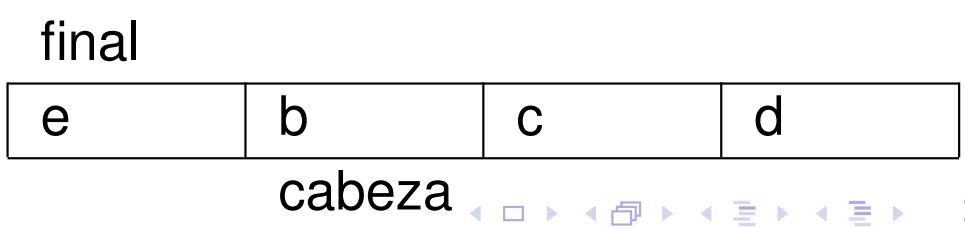
5) Insertar_en_Cola (d,C)



6) Quitar_Primero (C)



7) Insertar_en_Cola (e,C)



Pseudocódigo (i)

```
tipo Cola = registro
  Cabeza_de_cola, Final_de_cola: 1..Tamaño_máximo_de_cola
  Tamaño_de_cola : 0..Tamaño_máximo_de_cola
  Vector_de_cola : vector [1..Tamaño_máximo_de_cola]
                          de Tipo_de_elemento
fin registro
procedimiento Crear_Cola ( C ) O(1)
  C.Tamaño_de_cola := 0;
  C.Cabeza_de_cola := 1;
  C.Final_de_cola := Tamaño_máximo_de_cola
fin procedimiento
función Cola_Vacía ( C ) : test O(1)
  devolver C.Tamaño_de_cola = 0
fin función
```

Pseudocódigo (ii)

```
procedimiento incrementar ( x ) (* privado *) O(1)
  si x = Tamaño_máximo_de_cola entonces x := 1
  sino x := x + 1
fin procedimiento

procedimiento Insertar_en_Cola ( x, C ) O(1)
  si C.Tamaño_de_Cola = Tamaño_máximo_de_cola entonces
    error Cola llena
  sino
    C.Tamaño_de_cola := C.Tamaño_de_cola + 1;
    incrementar(C.Final_de_cola);
    C.Vector_de_cola[C.Final_de_cola] := x;
fin procedimiento
```

Pseudocódigo (iii)

```
función Quitar_Primero ( C ) : Tipo_de_elemento O(1)
  si Cola_Vacía ( C ) entonces
    error Cola vacía
  sino
    C.Tamaño_de_cola := C.Tamaño_de_cola - 1;
    x := C.Vector_de_cola[C.Cabeza_de_cola];
    incrementar(C.Cabeza_de_cola);
    devolver x
fin función
función Primero ( C ) : Tipo_de_elemento O(1)
  si Cola_Vacía ( C ) entonces
    error Cola vacía
  sino
    devolver C.Vector_de_cola[C.Cabeza_de_cola]
fin función
```

Código C: colas.h

```
#define TAMANO_MAXIMO_COLA 5
typedef int tipo_elemento;
typedef struct {
  int cabeza, final, tamano;
  tipo_elemento vector[TAMANO_MAXIMO_COLA];
} cola;
void crear_cola(cola *);
int cola_vacia(cola);
void insertar(tipo_elemento, cola *);
tipo_elemento quitar_primero(cola *);
tipo_elemento primero(cola);
/* ERRORES: quitar_primero o primero sobre una cola vacía
            insertar en una cola llena */
```

Código C: colas.c (i)

```
#include <stdlib.h>
#include <stdio.h>
#include "colas.h"
void crear_cola(cola *c) {
  c->tamano = 0;
  c->cabeza = 0;
  c->final = -1;
}
int cola_vacia(cola c) {
  return (c.tamano == 0);
}
void incrementar(int *x) {       /* privado */
  if (++(*x) == TAMANO_MAXIMO_COLA)
    *x = 0;
}
```

Código C: colas.c (ii)

```
void insertar(tipo_elemento x, cola *c) {
  if (c->tamano == TAMANO_MAXIMO_COLA) {
    printf("error: cola llena: %d\n", c->tamano);
    exit(EXIT_FAILURE);
  }
  c->tamano++;
  incrementar(&(c->final));
  c->vector[c->final] = x;
}
tipo_elemento primero(cola c) {
  if (cola_vacia(c)) {
    printf("error: cola vacia\n"); exit(EXIT_FAILURE);
  }
  return(c.vector[c.cabeza]);
}
```

Código C: colas.c (iii)

```
tipo_elemento quitar_primero(cola *c) {
  tipo_elemento x;
  if (cola_vacia(*c)) {
    printf("error: cola vacia\n");
    exit(EXIT_FAILURE);
  }
  c->tamano--;
  x = c->vector[c->cabeza];
  incrementar(&(c->cabeza));
  return x;
}
```

Índice

Listas

- Operaciones básicas:
  - `Visualizar` su contenido.
  - `Buscar` la posición de la primera ocurrencia de un elemento.
  - `Insertar` y `Eliminar` un elemento en alguna posición.
  - `Buscar_k_esimo`, que devuelve el elemento de la posición indicada

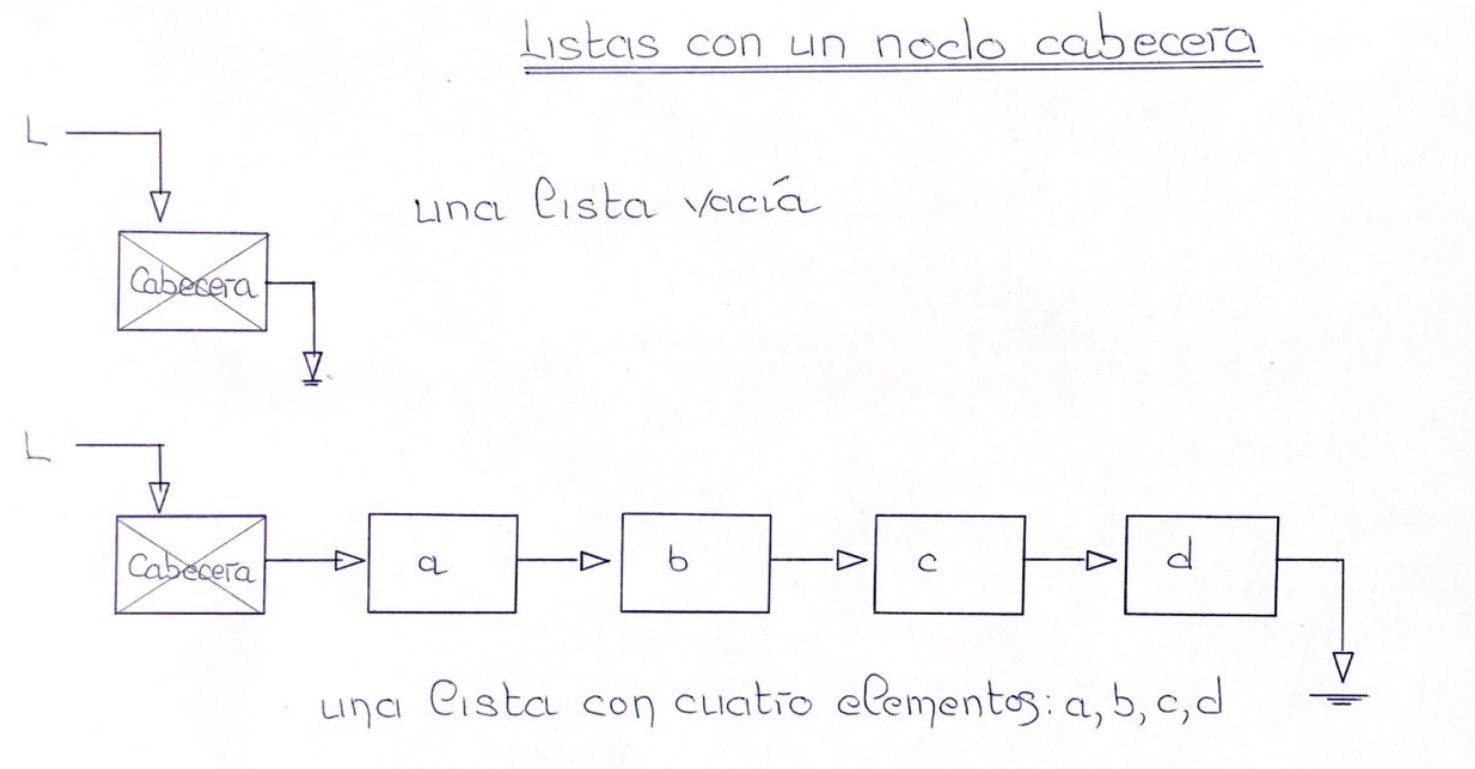
Implementación de listas a base de vectores

- Tiene que declararse el tamaño de la lista.
  - Exige sobrevaloración.
  - Consume mucho espacio.
- Complejidad computacional de las operaciones:
  - `Buscar_k_esimo`, tiempo constante
  - `Visualizar` y `Buscar`, tiempo lineal.
  - `Insertar` y `Eliminar` son costosas.
    - Insertar o eliminar un elemento exige, en promedio, desplazar la mitad de los valores, $O(n)$.
    - La construcción de una lista o la eliminación de todos sus elementos podría exigir un tiempo cuadrático.

Implementación de listas a base de apuntadores

- Cada nodo apunta al siguiente; el último no apunta a nada.
- La lista es un puntero al primer nodo (y al último).
- Complejidad computacional de las operaciones:
  - `Visualizar` y `Buscar`, tiempo lineal.
  - `Buscar_k_esimo`, tiempo lineal.
  - `Eliminar` realiza un cambio de apuntadores y una orden `dispose`, $O(1)$.
    - Usa `Buscar_anterior` cuyo tiempo de ejecución es lineal.
  - `Insertar` tras una posición `p` require una llamada a `new` y dos maniobras con apuntadores, $O(1)$.
    - Buscar la posición `p` podría llevar tiempo lineal.
  - Un nodo cabecera facilita la inserción y la eliminación al comienzo de la lista.

Lista simples con nodo cabecera



Implementación de listas doblemente enlazadas

- Cada nodo apunta al siguiente y al anterior.
- Duplica el uso de la memoria necesaria para los punteros.
- Duplica el coste de manejo de punteros al insertar y eliminar.
- La eliminación se simplifica.
  - No es necesario buscar el elemento anterior.

Pseudocódigo: Implementación con un nodo cabecera (i)

```
tipo PNodo = puntero a Nodo
     Lista = PNodo
     Posición = PNodo
     Nodo = registro
       Elemento : Tipo_de_elemento
       Siguiente : PNodo
     fin registro
procedimiento Crear Lista ( L ) O(1)
  nuevo ( tmp );
  si tmp = nil  entonces error Memoria agotada
  sino
     tmp^.Elemento := { nodo cabecera };
     tmp^.Siguiente := nil;
     L := tmp
fin procedimiento
```

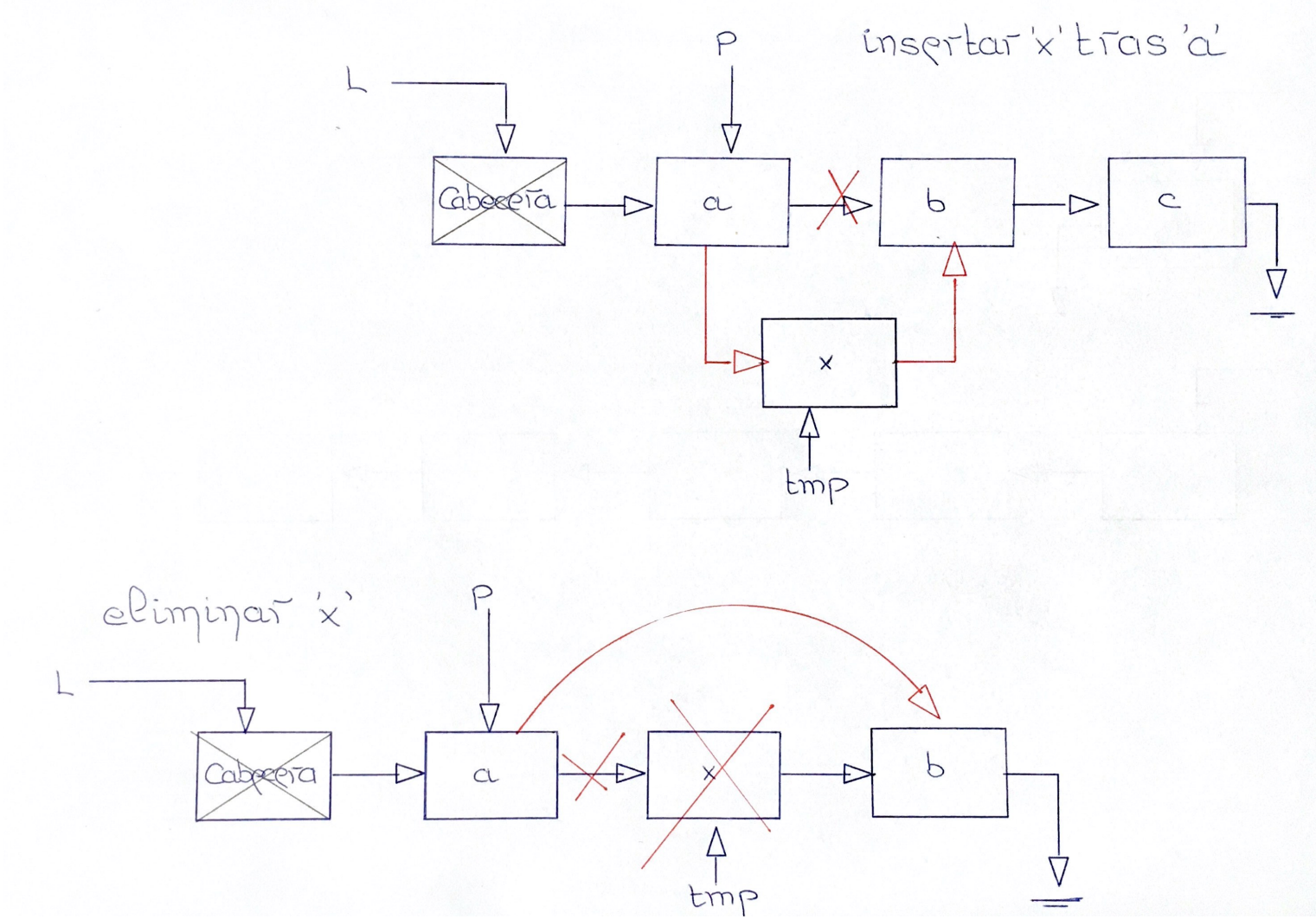
Pseudocódigo: Implementación con un nodo cabecera (ii)

```
función Lista Vacía ( L ) : test O(1)
  devolver L^.Siguiente = nil
fin función

función Buscar ( x, L ) : posición de la 1ª ocurrencia
                          o nil O(n)
  p := L^.Siguiente;
  mientras p <> nil y p^.Elemento <> x hacer
    p := p^.Siguiente;
  devolver p
fin función

función Último Elemento ( p ) : test { privada } O(1)
  devolver p^.Siguiente = nil
fin función
```

Pseudocódigo: Implementación con un nodo cabecera (iii)

```
función Buscar Anterior ( x, L ) : posición anterior a x
                                   o a nil { privada } O(n)
  p := L;
  mientras pˆ.Siguiente <> nil y
                    pˆ.Siguienteˆ.Elemento <> x hacer
    p := pˆ.Siguiente;
  devolver p
fin función
procedimiento Eliminar ( x, L ) O(n)
  p := Buscar Anterior ( x, L );
  si Último Elemento ( p ) entonces error No encontrado
  sino tmp := pˆ.Siguiente;
       pˆ.Siguiente := tmpˆ.Siguiente;
       liberar ( tmp )
fin procedimiento
```

Insertar y eliminar



Pseudocódigo: Implementación con un nodo cabecera (iv)

```
procedimiento Insertar ( x, L, p ) O(1)
  nuevo ( tmp );   { Inserta después de la posición p }
  si tmp = nil  entonces
    error Memoria agotada
  sino
    tmp^.Elemento := x;
    tmp^.Siguiente := p^.Siguiente:
    p^.Siguiente := tmp
fin procedimiento
```

Código C: `listas.h` (i)

```
struct nodo {
  void *elem; /* 'void *' es un apuntador 'generico' */
  struct nodo *sig;
};
typedef struct nodo *posicion;
typedef struct nodo *lista;
lista crearlista();
int eslistavacia(lista l);
void insertar(void *e, posicion p);
   /*inserta e tras el nodo apuntado por p*/
posicion buscar(lista l, void *e,
                int (*comp)(const void *x, const void *y));
  /*la función comp devuelve un número mayor, igual o menor
    que cero, según x sea mayor, igual, o menor que y*/
```

Código C: `listas.h` (ii)

```
/*
Como ejemplo, la función:
int strcmp(const char *str1, const char *str2)
de la biblioteca estándar C compara dos cadenas de
caracteres apuntadas por str1 y str2.

El valor devuelto por esta función es:
- menor que cero si str1 precede alfabéticamente a str2
- mayor que cero si str2 precede alfabéticamente a str1
- cero si ambas cadenas son iguales.
*/
```

Código C: `listas.h` (iii)

```
void borrar(lista l, void *e,
            int (*comp)(const void *x, const void *y));

posicion primero(lista l);
posicion siguiente(posicion p);
int esfindelista(posicion p);
void *elemento(posicion p);

  /* Para recorrer los elementos de la lista:

  for(p=primero(l); !esfindelista(p); p=siguiente(p)) {
      //hacer algo con elemento(p)
  }
  */
```

Código C: `listas.c` (i)

```
#include <stdlib.h>
#include <stdio.h>
#include "listas.h"
static struct nodo *crearnodo(){
  struct nodo *tmp = malloc(sizeof(struct nodo));
  if (tmp == NULL) {
    printf("memoria agotada\n"); exit(EXIT_FAILURE);
  }
  return tmp;
}
lista crearlista(){
  struct nodo *l = crearnodo();
  l->sig = NULL;
  return l;
}
```

Código C: `listas.c` (ii)

```
int eslistavacia(lista l){
  return (l->sig == NULL);
}
void insertar(void *x, posicion p) {
  struct nodo *tmp = crearnodo();
  tmp->elem = x; tmp->sig = p->sig;
  p->sig = tmp;
}
posicion buscar(lista l, void *e,
                int (*comp)(const void *x, const void *y)){
  struct nodo *p = l->sig;
  while (p != NULL && 0!=(*comp)(p->elem, e))
    p = p->sig;
  return p;
}
```

Código C: `listas.c` (iii)

```
static posicion buscarant(lista l, void *x,
                          int (*comp)(const void *, const void *)) {
  struct nodo *p = l;
  while (p->sig != NULL && 0!=(*comp)(p->sig->elem, x))
    p = p->sig;
  return p;
}

static int esultimoelemento(struct nodo *p) {
  return (p->sig == NULL);
}
```

Código C: `listas.c` (iv)

```
void borrar(lista l, void *x,
            int (*comp)(const void *, const void *)) {
  struct nodo *tmp, *p = buscarant(l, x, comp);
  if (!esultimoelemento(p)) {
    tmp = p->sig;
    p->sig = tmp->sig;
    free(tmp);
  }
}

posicion primero(lista l) { return l->sig; }
posicion siguiente(posicion p) { return p->sig; }
int esfindelista(posicion p) { return (p==NULL); }
void *elemento(posicion p) { return p->elem; }
```

Ejemplo con una función para comparar dos enteros

```
#include <stdio.h>
int intcmp(const void *a, const void *b) {
  int x = *(int *)a, y = *(int *)b;
  return (x - y);
}
int main(void){
  int v [] = {0, 5};
  printf("compara 0 5 : %d\n", intcmp(&(v[0]), &(v[1])));
  printf("compara 5 0 : %d\n", intcmp(&(v[1]), &(v[0])));
  printf("compara 5 5 : %d\n", intcmp(&(v[1]), &(v[1])));
/* salida: */
/* compara 0 5 : -5 */
/* compara 5 0 : 5 */
/* compara 5 5 : 0 */
}
```