UAM
Universidad Autónoma
de Madrid

Escuela Politécnica Superior

Estructuras de Datos

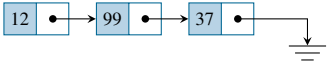
Listas enlazadas

Índice

Lista

Una **Lista** es una colección secuencial de elementos organizados de tal manera que:

- La inserción/extracción se puede hacer en cualquier posición
- Todos los elementos salvo el último tienen un elemento *siguiente*
- Todos los elementos salvo el primero tienen un elemento *anterior*
- Se puede usar para implementar otros TAD como Pila o Cola



Estructura de datos (I)

Primera opción: Usar un *array* estático de elementos

```
next(L[i])= L[i+1]
prev(L[i])= L[i-1]
```

Ventajas:

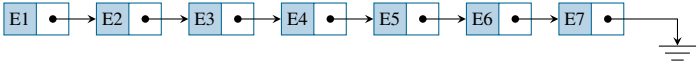
- Implementación sencilla
- Memoria estática y contigua

Inconvenientes:

- Desperdicio de memoria
- Ineficiente para insertar en/extraer de principio o medio

Estructura de datos (II)

Segunda opción: Usar una Lista Enlazada de nodos



Nodo: EdD con dos campos:

- **info:** Contiene los datos
- **next:** Puntero al siguiente nodo de la lista
- El campo *next* del último nodo de la lista apunta a `NULL`

Primitivas

- `List list_new()`: crea una nueva lista
- `list_free(List l)`: libera una lista
- `Boolean list_isEmpty(List l)`: verdadero si la lista está vacía
- `Status list_pushFront(List l, Element e)`: inserta un elemento al inicio de la lista
- `Status list_pushBack(List l, Element e)`: inserta un elemento al final de la lista
- `Element list_popFront(List l)`: extrae un elemento del principio de la lista
- `Element list_popBack(List l)`: extrae un elemento del final de la lista

Otras primitivas (comunes)

- `Integer list_size(List l)`: devuelve el número de elementos de la lista
- `Status list_insert(List l, Position i, Element e)`: inserta un elemento en la posición `i`
- `Element list_extract(List l, Position i)`: extrae un elemento de la posición `i`
- `list_getFront`, `list_getBack`, `list_getPosition`...

Primitivas en C++ (I)

Implementación en C++: `std::list`

Capacity:

`empty`	Test whether container is empty (public member function)
`size`	Return size (public member function)
`max_size`	Return maximum size (public member function)

Element access:

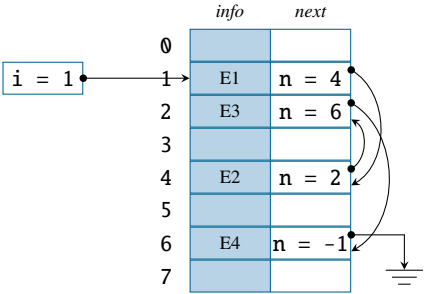
`front`	Access first element (public member function)
`back`	Access last element (public member function)

Primitivas en C++ (II)

Modifiers:

`assign`	Assign new content to container (public member function)
`emplace_front`	Construct and insert element at beginning (public member function)
`push_front`	Insert element at beginning (public member function)
`pop_front`	Delete first element (public member function)
`emplace_back`	Construct and insert element at the end (public member function)
`push_back`	Add element at the end (public member function)
`pop_back`	Delete last element (public member function)
`emplace`	Construct and insert element (public member function)
`insert`	Insert elements (public member function)
`erase`	Erase elements (public member function)
`swap`	Swap content (public member function)
`resize`	Change size (public member function)
`clear`	Clear content (public member function)

EdD para la lista enlazada: tabla estática



Ventajas:

- Memoria estática, implementación sencilla

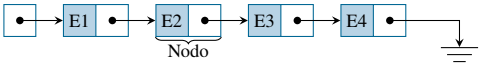
Inconvenientes:

- Desperdicio de memoria
- ¿Cuál es el siguiente nodo libre?

EdD para la lista enlazada: lista dinámica de nodos

Los nodos se crean y destruyen dinámicamente:

- Memoria dinámica
- Creación del nodo con `malloc`
- Destrucción del nodo con `free`



Ventajas:

- Reserva solo la memoria que hace falta
- Capacidad ilimitada (si hay memoria disponible)
- No se desplaza la memoria cuando se insertan/extraen elementos

Inconvenientes:

- Solo acceso secuencial, el acceso aleatorio es complicado

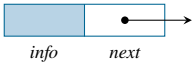
Implementación en C (I)

EdD para Nodo:

- Implementada en `list.c`
- Oculta para el usuario

Estructura en `list.c`

```
struct _Node {
  void *info;
  struct _Node *next;
};

typedef struct _Node Node;
```



Implementación en C (II)

Creación de Nodo:

Función `node_new` en `list.c`

```
Node *node_new() {
  Node *pn = NULL;

  pn = (Node *)malloc(sizeof(Node));
  if (!pn) {
    return NULL;
  }

  pn->info = NULL;
  pn->next = NULL;

  return pn;
}
```

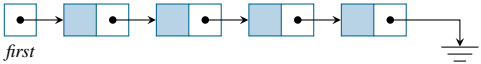
Destrucción de Nodo:

- Como la memoria de los elementos se reserva fuera de la lista, el nodo se destruye usando `free`

Implementación en C (III)

EdD para Lista:

- Colección de nodos enlazados
- La lista es un puntero al primer nodo (*first*)
- El campo *next* del último nodo es `NULL`



Tipo de dato en `list.h`

```
typedef struct _List List;
```

Estructura en `list.c`

```
struct _List {
  Node *first;
};
```

Implementación en C (IV)

Primitivas:

Cabeceras de las primitivas en `list.h`

```
List *list_new();
Boolean list_isEmpty(const List *pl);
Status list_pushFront(List *pl, const void *e);
Status list_pushBack(List *pl, const void *e);
void *list_popFront(List *pl);
void *list_popBack(List *pl);
void list_free(List *pl);
```

Implementación en C (V)

Crear e inicializar Lista:

Función `list_new` en `list.c`

```
List *list_new() {
  List *pl = NULL;

  pl = (List *)malloc(sizeof(List));
  if (pl == NULL) {
    return NULL;
  }

  pl->first = NULL;

  return pl;
}
```

Implementación en C (VI)

Comprobar si la lista está vacía:

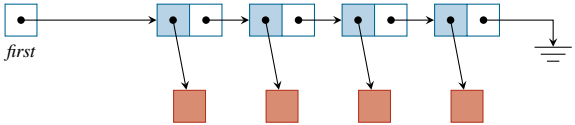
Función `list_isEmpty` en `list.c`

```
Boolean list_isEmpty(const List *pl) {
  if (pl == NULL) {
    return TRUE;
  }

  if (pl->first == NULL) {
    return TRUE;
  }

  return FALSE;
}
```

Nota: Se debe llamar a esta función antes de cualquier operación de extracción

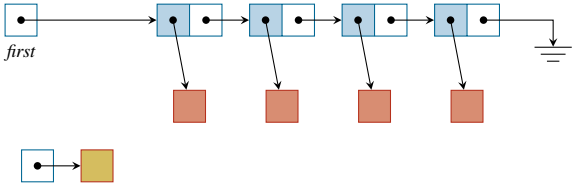
Implementación en C (VII)

Insertar elemento delante:



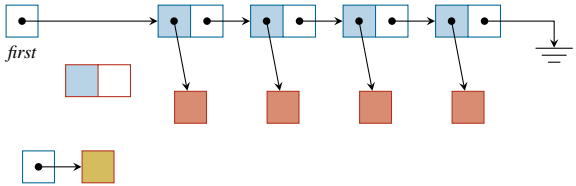
Implementación en C (VII)

Insertar elemento delante:



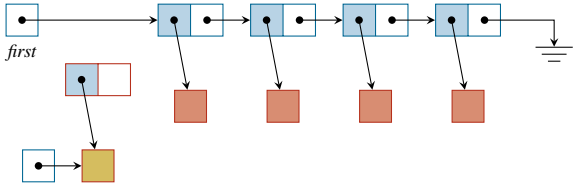
Implementación en C (VII)

Insertar elemento delante:



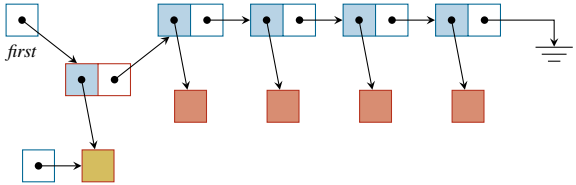
Implementación en C (VII)

Insertar elemento delante:



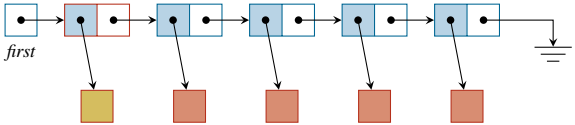
Implementación en C (VII)

Insertar elemento delante:



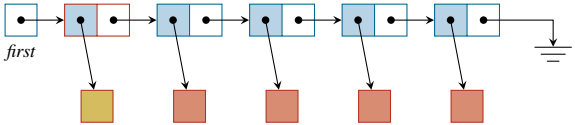
Implementación en C (VII)

Insertar elemento delante:



Implementación en C (VII)

Insertar elemento delante:



- Las modificaciones se hacen desde *first*

Implementación en C (VIII)

Insertar elemento delante:

Función `list_pushFront` en `list.c`

```
Status list_pushFront(List *pl, const void *e) {
  Node *pn = NULL;

  if (pl == NULL || e == NULL) {
    return ERROR;
  }

  pn = node_new();
  if (pn == NULL) {
    return ERROR;
  }

  pn->info = (void *)e;
```

...

Implementación en C (IX)

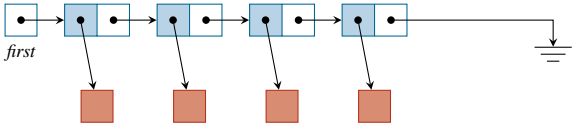
Insertar elemento delante:

Función `list_pushFront` **en** `list.c`

```
                                                    ...
  pn->next = pl->first;
  pl->first = pn;

  return OK;
}
```

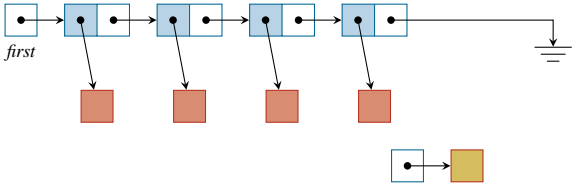
Implementación en C (X)

Insertar elemento detrás:



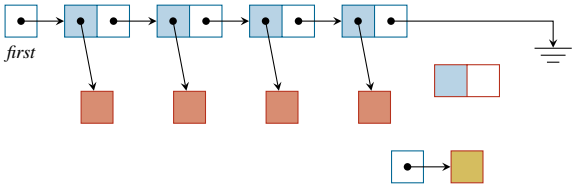
Implementación en C (X)

Insertar elemento detrás:



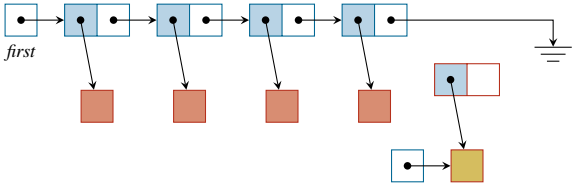
Implementación en C (X)

Insertar elemento detrás:



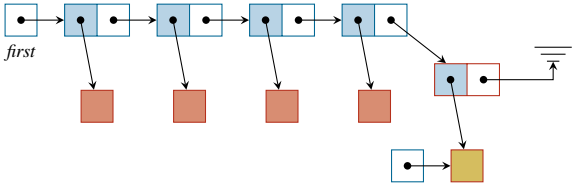
Implementación en C (X)

Insertar elemento detrás:



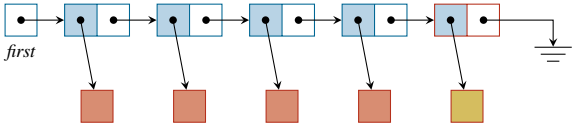
Implementación en C (X)

Insertar elemento detrás:



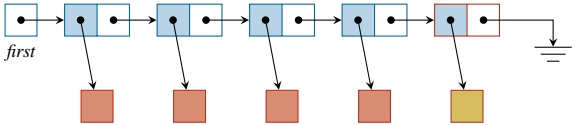
Implementación en C (X)

Insertar elemento detrás:



Implementación en C (X)

Insertar elemento detrás:



- Las modificaciones se hacen desde el último nodo

Implementación en C (XI)

Insertar elemento detrás:

Función `list_pushBack` en `list.c`

```
Status list_pushBack(List *pl, const void *e) {
  Node *pn = NULL, *qn = NULL;

  if (pl == NULL || e == NULL) {
    return ERROR;
  }

  pn = node_new();
  if (pn == NULL) {
    return ERROR;
  }

  pn->info = (void *)e;
```

...

Implementación en C (XII)

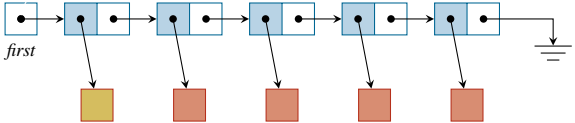
Insertar elemento detrás:

Función `list_pushBack` **en** `list.c`

```
                                                    ...
  /* Case 1: empty List - insert at initial position: */
  if (list_isEmpty(pl) == TRUE) {
    pl->first = pn;
    return OK;
  }

  /* Case 2: non empty List - traverse the List and insert: */
  qn = pl->first;
  while (qn->next != NULL) {
    qn = qn->next;
  }
  qn->next = pn;

  return OK;
}
```

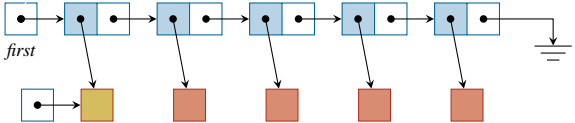
Implementación en C (XIII)

Extraer elemento de delante:



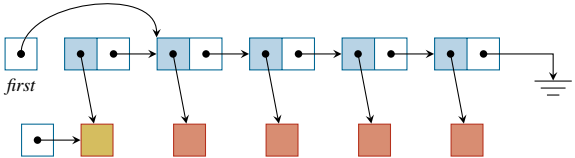
Implementación en C (XIII)

Extraer elemento de delante:



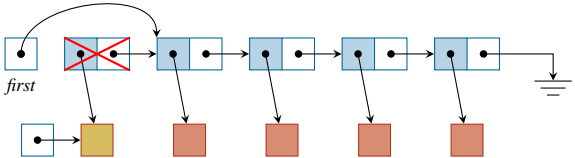
Implementación en C (XIII)

Extraer elemento de delante:



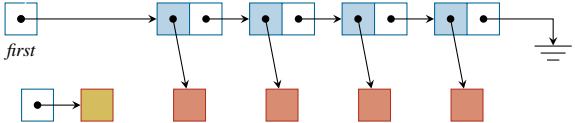
Implementación en C (XIII)

Extraer elemento de delante:



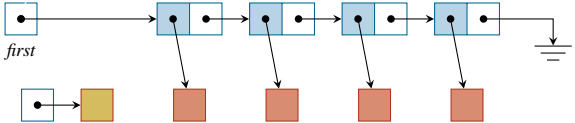
Implementación en C (XIII)

Extraer elemento de delante:



Implementación en C (XIII)

Extraer elemento de delante:



- Las modificaciones se hacen desde *first*

Implementación en C (XIV)

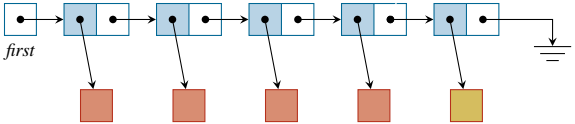
Extraer elemento de delante:

Función `list_popFront` en `list.c`

```
void *list_popFront(List *pl) {
  Node *pn = NULL;
  void *pe = NULL;

  if (pl == NULL || list_isEmpty(pl) == TRUE) {
    return NULL;
  }

  pn = pl->first;
  pe = pn->info; /* Equivalently: pe = pl->first->info */
  pl->first = pn->next;

  free(pn);

  return pe;
}
```

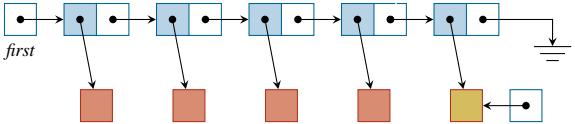
Implementación en C (XV)

Extraer elemento de detrás:



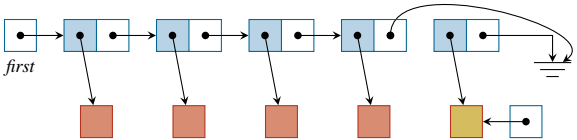
Implementación en C (XV)

Extraer elemento de detrás:



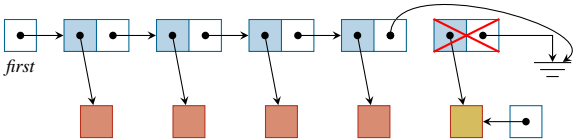
Implementación en C (XV)

Extraer elemento de detrás:



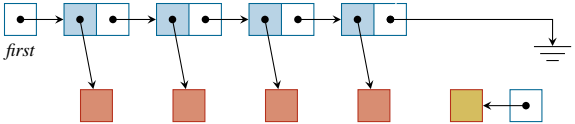
Implementación en C (XV)

Extraer elemento de detrás:



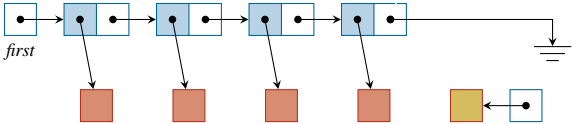
Implementación en C (XV)

Extraer elemento de detrás:



Implementación en C (XV)

Extraer elemento de detrás:



- Las modificaciones se hacen desde el penúltimo nodo

Implementación en C (XVI)

Extraer elemento de detrás:

Función `list_popBack` en `list.c`

```
void *list_popBack(List *pl) {
  Node *pn = NULL;
  void *pe = NULL;

  if (pl == NULL || list_isEmpty(pl) == TRUE) {
    return NULL;
  }

  /* Case 1: List with one single element - extract the first element: */
  if (pl->first->next == NULL) {
    pe = pl->first->info;
    free(pl->first); pl->first = NULL;

    return pe;
  }
```

...

Implementación en C (XVII)

Extraer elemento de detrás:

Función `list_popBack` en `list.c`

```
                                                    ...
  /* Case 2: List with more than one element - traverse the List and extract: */
  pn = pl->first;
  while (pn->next->next != NULL) {
    pn = pn->next;
  }

  pe = pn->next->info;
  free(pn->next);
  pn->next = NULL;

  return pe;
}
```

Implementación en C (XVIII)

Destrucción de Lista (usando las primitivas):

Función `list_free` en `list.c`

```
void list_free(List *pl) {
  if (pl == NULL) {
    return;
  }

  while (list_isEmpty(pl) == FALSE) {
    list_popFront(pl);
  }

  free(pl);
}
```

Implementación en C (XIX)

Destrucción de Lista (accediendo a la EdD):

Función `_list_freeDS` en `list.c`

```
void _list_freeDS(List *pl) {
  Node *pn = NULL;

  if (pl == NULL) {
    return;
  }

  while (pl->first != NULL) {
    pn = pl->first;
    pl->first = pn->next;
    free(pn);
  }

  free(pl);
}
```

Implementación en C (XX)

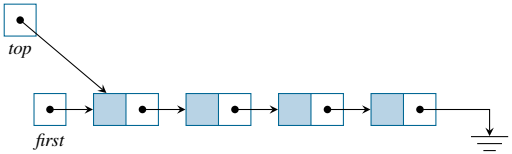
Destrucción de Lista (usando recursión):

Función `_list_freeRec` en `list.c`

```c
void _list_freeRec(List *pl) {
  if (pl == NULL) {
    return;
  }

  _list_freeRecInner(pl->first);
  free(pl);
}
void _list_freeRecInner(Node *pn) {
  if (pn == NULL) {
    return;
  }

  _list_freeRecInner(pn->next);
  free(pn);
}
```

Implementación de Pila usando una lista enlazada (I)

EdD para Pila:



Tipo de dato en `stack.h`

```
typedef struct _Stack Stack;
```

Estructura en `stack.c`

```
struct _Stack {
  List *pl;
};
```

Implementación de Pila usando una lista enlazada (II)

Primitivas:

Cabeceras de las primitivas en `stack.h`

```
Stack *stack_new();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const void *e);
void *stack_pop(Stack *ps);
```

Implementación de Pila usando una lista enlazada (III)

Crear e inicializar Pila:

Función `stack_new` en `stack.c`

```
Stack *stack_new() {
  Stack *ps = NULL;

  ps = (Stack *)malloc(sizeof(Stack));
  if (ps == NULL) {
    return NULL;
  }

  ps->pl = list_new();
  if (ps->pl == NULL) {
    free(ps);
    return NULL;
  }

  return ps;
}
```

Implementación de Pila usando una lista enlazada (IV)

Destrucción de Pila:

Función `stack_free` en `stack.c`

```
void stack_free(Stack *ps) {
  if (ps != NULL) {
    list_free(ps->pl);
    free(ps);
  }
}
```

Implementación de Pila usando una lista enlazada (V)

Comprobar si la pila está vacía:

Función `stack_isEmpty` en `stack.c`

```
Boolean stack_isEmpty(const Stack *ps) {
  if (ps == NULL) {
    return TRUE;
  }

  return list_isEmpty(ps->pl);
}
```

Nota: Se debe llamar a esta función antes de cualquier operación de extracción

Implementación de Pila usando una lista enlazada (VI)

Comprobar si la pila está llena:

Función `stack_isFull` en `stack.c`

```
Boolean stack_isFull(const Stack *ps) {
  if (ps == NULL) {
    return TRUE;
  }

  return FALSE;
}
```

Devuelve siempre `FALSE`, la pila nunca está llena

Implementación de Pila usando una lista enlazada (VII)

Insertar elemento:

Función `stack_push` en `stack.c`

```
Status stack_push(Stack *ps, const void *e) {
  if (ps == NULL || e == NULL) {
    return ERROR;
  }

  return list_pushFront(ps->pl, e);
}
```

Implementación de Pila usando una lista enlazada (VIII)

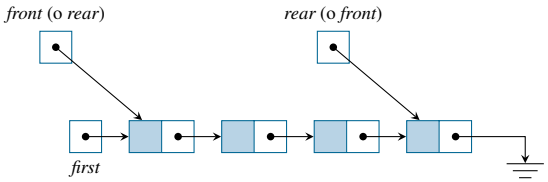
Extraer elemento:

Función `stack_pop` en `stack.c`

```
void *stack_pop(Stack *ps) {
  if (ps == NULL) {
    return NULL;
  }

  return list_popFront(ps->pl);
}
```

Implementación de Cola usando una lista enlazada (I)

EdD para Cola:



Tipo de dato en `queue.h`

```
typedef struct _Queue Queue;
```

Estructura en `queue.c`

```
struct _Queue {
  List *pl;
};
```

Implementación de Cola usando una lista enlazada (II)

Primitivas:

Cabeceras de las primitivas en `queue.h`

```
Queue *queue_new();
void queue_free(Queue *pq);
Boolean queue_isEmpty(const Queue *pq);
Boolean queue_isFull(const Queue *pq);
Status queue_push(Queue *pq, const void *e);
void *queue_pop(Queue *pq);
```

Implementación de Cola usando una lista enlazada (III)

Crear e inicializar Cola:

Función queue_new **en** queue.c

```
Queue *queue_new() {
  Queue *pq = NULL;

  pq = (Queue *)malloc(sizeof(Queue));
  if (pq == NULL) {
    return NULL;
  }

  pq->pl = list_new();
  if (pq->pl == NULL) {
    free(pq);
    return NULL;
  }

  return pq;
}
```

Implementación de Cola usando una lista enlazada (IV)

Destrucción de Cola:

Función `queue_free` en `queue.c`

```
void queue_free(Queue *pq) {
  if (pq != NULL) {
    list_free(pq->pl);
    free(pq);
  }
}
```

Implementación de Cola usando una lista enlazada (V)

Comprobar si la cola está vacía:

Función `queue_isEmpty` **en** `queue.c`

```
Boolean queue_isEmpty(const Queue *pq) {
  if (pq == NULL) {
    return TRUE;
  }

  return list_isEmpty(pq->pl);
}
```

Nota: Se debe llamar a esta función antes de cualquier operación de extracción

Implementación de Cola usando una lista enlazada (VI)

Comprobar si la cola está llena:

Función `queue_isFull` en `queue.c`

```
Boolean queue_isFull(const Queue *pq) {
  if (pq == NULL) {
    return TRUE;
  }

  return FALSE;
}
```

Nota: Devuelve siempre `FALSE`, la cola nunca está llena

Implementación de Cola usando una lista enlazada (VII)

Insertar elemento:

Función queue_push **en** queue.c

```
Status queue_push(Queue *pq, const void *e) {
  if (pq == NULL || e == NULL) {
    return ERROR;
  }

  return list_pushBack(pq->pl, e);
}
```

Nota: El coste de esta función es $\mathcal{O}(N)$, pero si se intercambian *front* y *rear* y se usa `list_pushFront` pasa a ser $\mathcal{O}(1)$

Implementación de Cola usando una lista enlazada (VIII)

Extraer elemento:

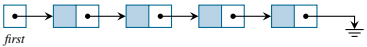
Función queue_pop **en** queue.c

```
void *queue_pop(Queue *pq) {
  if (pq == NULL) {
    return NULL;
  }

  return list_popFront(pq->pl);
}
```

Nota: El coste de esta función es $\mathcal{O}(1)$, pero si se intercambian *front* y *rear* y se usa `list_popBack` pasa a ser $\mathcal{O}(N)$

Tipos de listas

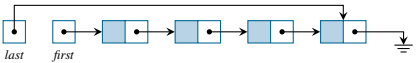
- Lista enlazada (simple)




```
struct _Node {
  void *info;
  struct _Node *next;
};
```

```
struct _List {
  Node *first;
};
```

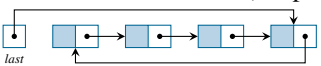
- Lista enlazada (simple) con acceso posterior




```
struct _Node {
  void *info;
  struct _Node *next;
};
```

```
struct _List {
  Node *first;
  Node *last;
};
```

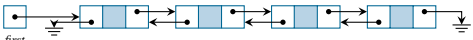
- Lista enlazada circular (simple)




```
struct _Node {
  void *info;
  struct _Node *next;
};
```

```
struct _List {
  Node *last;
};
```

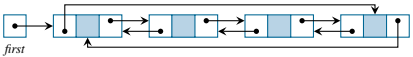
- Lista doblemente enlazada




```
struct _Node {
  void *info;
  struct _Node *next;
  struct _Node *prev;
};
```

```
struct _List {
  Node *first;
};
```

- Lista doblemente circular




```
struct _Node {
  void *info;
  struct _Node *next;
  struct _Node *prev;
};
```

```
struct _List {
  Node *first;
};
```

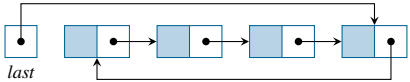
Lista enlazada circular (I)

Lista enlazada:

- Inserciones y extracciones de la última posición son costosas e ineficientes, con un coste computacional de $\mathcal{O}(N)$
- Es necesario iterar sobre todos los nodos de la lista

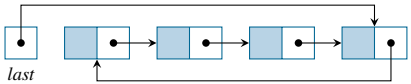
Lista enlazada circular:

- La lista apunta al último nodo (*last*)
- El campo *next* del último nodo apunta al primer nodo



Implementación de Lista Circular (I)

EdD para Lista Circular:



Tipo de dato en `clist.h`

```
typedef struct _CList CList;
```

Estructura en `clist.c`

```
struct _CList {
  Node *last;
};
```

Implementación de Lista Circular (II)

Primitivas:

Cabeceras de las primitivas en `clist.h`

```
CList *clist_new();
Boolean clist_isEmpty(const CList *pl);
Status clist_pushFront(CList *pl, const void *e);
Status clist_pushBack(CList *pl, const void *e);
void *clist_popFront(CList *pl);
void *clist_popBack(CList *pl);
void clist_free(CList *pl);
```

Implementación de Lista Circular (III)

Insertar elemento delante:

Función `clist_pushFront` en `clist.c`

```
Status clist_pushFront(CList *pl, const void *e) {
  Node *pn = NULL;

  if (pl == NULL || e == NULL) {
    return ERROR;
  }

  pn = node_new();
  if (pn == NULL) {
    return ERROR;
  }

  pn->info = (void *)e;
```

...

Implementación de Lista Circular (IV)

Insertar elemento delante:

Función `clist_pushFront` en `clist.c`

```
                                                    ...
  if (clist_isEmpty(pl)) {
    pn->next = pn;
    pl->last = pn;
  } else {
    pn->next = pl->last->next;
    pl->last->next = pn;
  }

  return OK;
}
```

Implementación de Lista Circular (V)

Insertar elemento detrás:

Función `clist_pushBack` en `clist.c`

```
Status clist_pushBack(CList *pl, const void *e) {
  Node *pn = NULL;

  if (pl == NULL || e == NULL) {
    return ERROR;
  }

  pn = node_new();
  if (pn == NULL) {
    return ERROR;
  }

  pn->info = (void *)e;
```

...

Implementación de Lista Circular (VI)

Insertar elemento detrás:

Función `clist_pushBack` en `clist.c`

```
                                                    ...
  if (clist_isEmpty(pl) == TRUE) {
    pn->next = pn;
    pl->last = pn;
  } else {
    pn->next = pl->last->next;
    pl->last->next = pn;
    pl->last = pn;
  }

  return OK;
}
```

Implementación de Lista Circular (VII)

Extraer elemento de delante:

Función `clist_popFront` en `clist.c`

```
void *clist_popFront(CList *pl) {
  Node *pn = NULL;
  void *pe = NULL;

  if (pl == NULL || clist_isEmpty(pl) == TRUE) {
    return NULL;
  }

  pn = pl->last->next;
  pe = pn->info;
```

...

Implementación de Lista Circular (VIII)

Extraer elemento de delante:

Función `clist_popFront` en `clist.c`

```
                                                    ...
  if (pl->last->next == pl->last) {
    pl->last = NULL;
  } else {
    pl->last->next = pn->next;
  }

  free(pn);

  return pe;
}
```

Implementación de Lista Circular (IX)

Extraer elemento de detrás:

Función `clist_popBack` en `clist.c`

```
void *clist_popBack(CList *pl) {
  Node *pn = NULL;
  void *pe = NULL;

  if (pl == NULL || clist_isEmpty(pl) == TRUE) {
    return NULL;
  }

  if (pl->last->next == pl->last) {
    pe = pl->last->info;
    free(pl->last);
    pl->last = NULL;

    return pe;
  }
```

...

Implementación de Lista Circular (X)

Extraer elemento de detrás:

Función `clist_popBack` en `clist.c`

```
                                                    ...
  pn = pl->last;
  while (pn->next != pl->last) {
    pn = pn->next;
  }

  pe = pl->last->info;
  pn->next = pl->last->next;
  free(pl->last);
  pl->last = pn;

  return pe;
}
```

Lista Circular

Ventajas:

- Como ahora la primitiva `clist_pushBack` es eficiente, no hace falta un bucle
- No requiere memoria adicional respecto a la implementación `List`
- Se puede implementar una cola con costes de operación de $\mathcal{O}(1)$

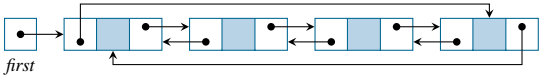
Inconvenientes:

- La primitiva `clist_popBack` aún requiere iterar sobre toda la lista
- Un puntero al penúltimo nodo no resuelve este problema

Lista Doblemente Circular (I)

Solución:

- Lista doblemente circular
- Acceso tanto al elemento *next* como al elemento *previous*
- Añade complejidad a todas las primitivas

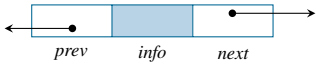


Lista Doblemente Circular (II)

EdD para Nodo:

Estructura en `dclist.c`

```
struct _Node {
  void *info;
  struct _Node *next;
  struct _Node *prev;
};

typedef struct _Node Node;
```



EdD para Lista Doblemente Circular:

Tipo de dato en `dclist.h`

```
typedef struct _DCList DCList;
```

Estructura en `dclist.c`

```
struct _DCList {
  Node *first;
};
```