

Índice

1. Teoría

- a. TADs
- b. Pilas
- c. Colas
- d. Listas

2. Ejemplos de código

- a. Bolsa
- b. Pila
- c. Cola
- d. Lista

Tipos abstractos de datos

Tipo de datos

- Colección de valores y operaciones.
 - Básicos: int, float char...
 - Definidos por el programador: enumerados, subrango
 - Opacos: Representación invisible al programador, operaciones predefinidas
- Tipos estructurados: Genericidad, pero riesgo de crear valores sin semántica (ej: fecha)
- Tipo abstracto de datos: Colección de valores y operaciones definidas por el programador

Tipo abstracto de datos

- Son opacos, por lo que su implementación se realiza en un ámbito inaccesible al resto de los programas
- El conjunto de operaciones debe permitir generar cualquier valor del tipo
- Partes diferenciadas:
 - **Especificación (interfaz)**: Conocida por el usuario. ([archivo.h](#))
 - Contiene nombre del tipo y especificación de operaciones
 - Parte sintáctica (nombre operación) y semántica (funcionalidad)
 - Debe ser estable y lo más sencillo posible
 - **Implementación**: Conocida por el programador. ([archivo.c](#))
 - Representación del tipo mediante otros tipos y realización de operaciones en un lenguaje de programación
 - Suele ser compleja y propensa a cambiar

Clasificación de TADs

- **Simples:** Cambian su valor pero no estructura. Ocupan siempre el mismo espacio (enteros, reales...)
- **Contenedores:** Cambian valor y estructura. Varía nº de elementos y espacio de almacenamiento (listas¹, colas, pilas, árboles, grafos)
- **Inmutables:** No existen operaciones de modificación, los casos no pueden modificarse. Se definen al ser asignados.
 - Representación inmutable o mutable (ej mutable: $\frac{1}{2}$ se puede representar como $\frac{2}{4}$)
- **Mutables:** Sus casos se pueden modificar mediante operaciones de modificación

Especificación de TADs

- **Informal:** Lenguaje natural, ambiguas pero sencillas de escribir y entender
 - Cabecera: Nombre de las operaciones
 - Descripción: se explica en qué consiste la abstracción sin entrar en detalles. Se puede hacer en términos de otros tipos más familiares y utilizar gráficos o abstracciones matemáticas
 - Se puede incluir si es inmutable o mutable
 - Especificación de las operaciones:
 - Cabecera: Nombre de la operación y tipos y nombre de las variables a utilizar. ej: **suma(a,b:racional) devuelve (racional)**
 - Cuerpo: Información semántica
 - Requerimientos: Dominio de procedimiento. Si se pueden omitir se considera abstracción total. Es responsabilidad del usuario que se cumplan. [opcional]
 - Modifica: Entradas que cambian de valor [opcional]
 - Efecto: Salidas producidas y las modificaciones producidas en el previo apartado, si se cumplen los requerimientos.

¹ usando malloc e tal, non listas nas que predefines o tamaño antes de enchela

- **Formal:** Lenguaje algebraico, breves y precisas pero complejas de entender. Permite verificación formal respecto al código.
 - **Tipo:** Nombre del TAD
 - **Sintaxis:** Forma de las operaciones
[nombre función (tipo argumentos) → tipo resultado]
 - **Semántica:** Comportamiento de las operaciones en algunos casos concretos para demostrar su funcionamiento
 - Algunas funciones, denominadas axiomas o constructoras no requieren reglas semánticas y se usan para crear cualquier elemento del tipo. Se marcan con * en la sintaxis
 - El resultado puede ser recursivo o contener referencias a algunos otros tipos ('cierto', 'falso', 'error')
 - Las condiciones se escriben como
si **condicion** → **valor_si_cierto**|**valor_si_falso**

Ejemplo especificación formal [implementación]

- **Tipo:** bolsa(elemento)
- **Sintaxis:**
 - *bolsavacia² → bolsa
 - *poner(bolsa,elemento) → bolsa
 - esvacia(bolsa) → booleano
 - cuantos(bolsa,elemento) → natural
- **Semántica:** $\forall b \in \text{bolsa}, \forall e, f \in \text{elemento}$
 - esvacia(bolsavacia) => cierto
 - esvacia(poner(b,e)) => falso
 - cuantos(bolsavacia, e) => cero
 - cuantos(poner(**b**,**f**), **e**) => si $f=e$ => sucesor(cuantos(b,e))³
|cuantos(b,e)

² non é un puntero, indica que esto é unha funcion constructora e polo tanto non ten semántica

³ sucesor(a): equivalente a a+1

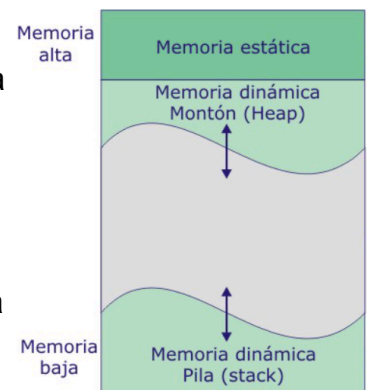
- Ejemplo: `cuantos(poner(poner(poner(bolsavacia,34),34),27),34)`
devolvería `sucesor(sucesor(0)) = 2`

Manejo de errores

- Lenguaje debe ser independiente de la implementación
- Se puede añadir a las listas de argumentos un parámetro de error
- También se pueden incluir operaciones de destrucción

Modelo lógico de memoria

- **Memoria alta:** Variables globales e instrucciones. Se asigna al comenzar la ejecución
- **Heap:** Variables con memoria asignada en tiempo de ejecución (`malloc`), crece y se reduce
- **Stack:** Variables locales y datos relacionados con llamada a funciones, crece y se reduce



Asignación dinámica de memoria

- **Variables dinámicas:** Variables cuyo espacio de almacenamiento se asigna en tiempo de ejecución. Se almacenan en heap
 - No tienen nombre, se accede a ellas con punteros
 - Se crean y liberan con `malloc` y `free`
 - Ej: `(int *) malloc(3*sizeof(int))`. La primera parte es el type cast que indica el uso que se le dará a la memoria, la segunda parte es la cantidad de memoria a reservar [en este caso, 3 enteros]
- **Variables puntero:** Apuntan a variables dinámicas
 - Si es global puede ser estático, si es local es dinámico

typedef

- Permite crear un nuevo nombre para otro tipo
- ej: `typedef int tipo;` permite usar `tipo` para crear un `int`

- También permite abreviar el nombre de una estructura, por ejemplo `typedef struct datos {...} sdatos` define una estructura 'struct datos' y pasa a llamarle 'sdatos'.
- Tras crear un TAD (ej: una estructura) se suele usar **typedef** para darle nombre a un puntero a esta estructura, que será el usado por las funciones

Punteros dinámicos

- `p = (int **) malloc (sizeof (int*));` reserva espacio de memoria para un puntero `int*`.
- Luego, se asigna este puntero: `*p = (int*) malloc (sizeof(int))`. De esta forma se crea el entero: `**p=14;`
- Se puede usar por ejemplo para matrices de datos
- Tras ejecución se libera primero `free(*p)` y después `free(p)`

Tipo de datos opaco

- En el `.h` debe ser declarado como puntero a void (ej: `void * bolsa;`)
- En el `.c` debe ser declarado como puntero al tipo apropiado

Técnicas de implementación

- **Tipos de implementación** según asignación de memoria:
 - Estática: asignación en tiempo de compilación. Ej: array
 - Dinámica: asignación en tiempo de ejecución
- **Tipos de representaciones**:
 - Contiguas: Direcciones de memoria contiguas
 - Enlazadas: Direcciones dispersas, enlazadas mediante punteros

Representaciones contiguas y enlazadas

- Si la cantidad de casos del TAD es fija y conocida, se define con `typedef tipoelem4 estructura[n]`
- Si es fija pero se conoce durante ejecución, se crea con un malloc de tamaño `(n*sizeof(tipoelem))`
- Si la cantidad de casos varía en tiempo de ejecución:
 - Alternativa 1: contigua. utiliza un vector estático, con un maximo que se conoce durante compilación
 - Alternativa 2: contigua. utiliza un vector dinamico, con un max de n elementos. Se usa si se conoce el max durante ejecución
 - Alternativa 3: enlazada. no requiere max por lo que es más eficiente en memoria, pero la destrucción se hace uno por uno

⁴ 'tipoelem' es un nombre dado mediante typedef al tipo de elemento que almacene el vector. en este caso, puede ser por ejemplo int.

Pilas

Descripción

- Tipo de secuencia enlazada. Se inserta y se suprime por la cima
- El último elemento en ser añadido es el primero en ser eliminado
- Ejemplos:
 - Llamadas a subprogramas (almacenan dirección del programa de origen y estado de las variables del programa al realizar la llamada)
 - Recursividad, operaciones aritméticas, quicksort

Operaciones

- PILAVACIA: crea pila vacía
- ESVACÍA: devuelve 'cierto' si está vacía
- CIMA: devuelve el elemento en el tope de la pila
- PUSH: añade un elemento al tope
- POP: suprime un elemento en el tope

Especificación **formal** del TAD Pila

- Tipo: `pila(tipoelem)`
- Sintaxis:
 - `*PilaVacía → pila`
 - `esVacía(pila) → booleano`
 - `cima(pila) → tipoelem`
 - `*push(pila, tipoelem) → pila`
 - `pop(pila) → pila`
- Semántica: $\forall P \in \text{pila}, \forall E \in \text{tipoelem}$:
 - $\text{esVacía}(\text{PilaVacía}) \Rightarrow \text{cierto}$
 - $\text{esVacía}(\text{push}(P, E)) \Rightarrow \text{falso}$
 - $\text{cima}(\text{PilaVacía}) \Rightarrow \text{error}$
 - $\text{cima}(\text{push}(P, E)) \Rightarrow E$
 - $\text{pop}(\text{PilaVacía}) \Rightarrow \text{error}$
 - $\text{pop}(\text{push}(P, E)) \Rightarrow P$

Implementación contigua (vectores)

- La pila se almacena como un **registro** de dos campos:
 - Vector de elementos
 - Entero 'cima' que indica el índice del elemento más alto
- Push incrementa el tope y añade el nuevo elemento, pop reduce el tope

Implementación con punteros

- Cada elemento de la pila será un registro (nodo) con dos elementos:
 - Valor del elemento de la pila
 - Puntero al siguiente nodo
- Identificaremos 'pila' con un puntero: el que apunta al elemento tope



-
- Push(p,e) crea una nueva pila que tiene el elemento e como cima, y esta cima apunta a p como siguiente elemento
- Pop se realiza con $*p = (*p) \rightarrow \text{sig}$

Comparación de implementaciones

- La implementación por arrays es mejor si:
 - se conoce la longitud estimada de la pila (permite max)
 - Los elementos de la pila son pequeños
 - Es común crear y destruir pilas, o recorrer sus elementos
- La implementación por punteros:
 - No requiere especificar max
 - Es más eficiente para insertar o eliminar elementos de una pila

Colas

Descripción

- Tipo de secuencia enlazada.
- Conjunto ordenado de elementos homogéneos en los cuales los elementos se eliminan por un extremo, el **principio** o cabeza, y se añaden por el **final**.
- El primer elemento en entrar será el primero en salir (**FIFO**)

Especificación formal

- **Valores:** Colección de elementos homogéneos que opera según un modelo FIFO.
- **Sintaxis:**
 - * ColaVacia \rightarrow TCOLA
 - * AñadirCola (TCOLA, TELEMENTO) \rightarrow TCOLA
 - PrimeroCola (TCOLA) \rightarrow TELEMENTO
 - EliminarCola (TCOLA) \rightarrow TCOLA
 - EsColaVacia (TCOLA) \rightarrow BOOLEAN
- **Semántica:** $\forall c \in \text{TCOLA}, \forall e \in \text{TELEMENTO}$
 - EsColaVacia (ColaVacia) \Rightarrow TRUE
 - EsColaVacia (AñadirCola(c,e)) \Rightarrow FALSE
 - PrimeroCola (ColaVacia) \Rightarrow ERROR
 - PrimeroCola (AñadirCola(c,e)) \Rightarrow **si** EsColaVacia(c) **entonces** e
si no PrimeroCola(c)
 - EliminarCola (ColaVacia) \Rightarrow ERROR
 - EliminarCola⁵ (AñadirCola(c,e)) \Rightarrow **si** EsColaVacia(c) **entonces** ColaVacia **si no** AñadirCola((EliminarCola(c),e)

⁵ non elimina a cola, elimina o primeiro elemento da cola

Implementación mediante memoria estática

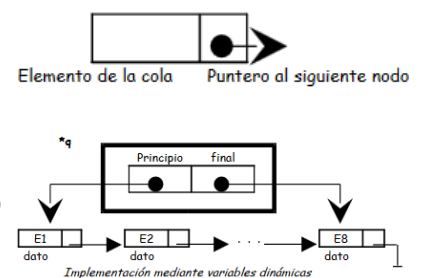
- Podemos representar una cola mediante una estructura de dos campos:
 - Un **array** donde se almacenen los elementos de la cola, con un máximo predefinido
 - Dos **índices** que indiquen el principio y final de la cola
 - EsColaLlena si el índice de final es el máximo del vector
 - EsColaVacía si el índice de final y inicio son el mismo
 - AñadirCola aumenta el final en 1, EliminarCola aumenta el principio en 1
- A medida que se eliminan elementos, sus posiciones al principio del vector van quedando vacías e inutilizables, por lo que se alcanza el máximo cuando queda espacio libre.
 - Para solucionarlo, se puede utilizar una estructura circular

Implementación mediante vector circular

- Definimos que la posición siguiente al último elemento es el primer elemento.
 - El caso especial se realiza con una función **siguiente**, que devuelve $[i+1]$ si la posición no es la final y $[0]$ si lo es.
- Ahora, la condición de EsColaVacía se activa también con una cola llena. Para evitar esto, se podría dejar una posición libre en la cola que nunca será ocupada.
 - La posición liberada es la indicada por el campo principio de la cola
 - La condición de ColaLlena será cuando el final de la cola se encuentre justo antes de la posición liberada

Implementación mediante punteros

- Definimos una estructura TNODO con dos campos: el elemento del nodo y un puntero al nodo siguiente
- Definimos una estructura STCOLA con dos campos que son ambos punteros: un puntero al TNODO principio y otro al TNODO final.



Dicolas

- Colas bidireccionales que permiten eliminar/añadir elementos por ambos extremos.
- Incluyen nuevas operaciones:
 - `void ultimo (TCOLA C, TIPOELEM *E)` devuelve el último elemento
 - `void insertar_frente (TCOLA *C, TIPOELEM E)` añade un elemento al inicio, y el principio de la cola retrocede 1 posición
 - `void suprimir_final (TCOLA *C)` elimina el último elemento, y el final de la cola retrocede 1 posición
- También se pueden incluir restricciones a en que extremos se puede añadir o eliminar elementos

Colas de prioridad

- Colas a cuyos elementos se les asigna una **prioridad**.
 - El elemento con mayor prioridad es procesado primero
 - Dos elementos de igual prioridad se procesan en el orden en que fueron incluidos.
- Se pueden guardar los elementos en orden de prioridad o guardarlos en orden de inclusión y luego tener en cuenta la prioridad al sacarlos.
 - Es más sencillo insertarlos en orden de prioridad
- **Implementación mediante secuencia enlazada:** Lista enlazada donde cada nodo tiene su dato, un número con su prioridad y un puntero al siguiente
 - Se pueden insertar datos en cualquier posición
- **Implementación mediante secuencia de colas:** Una cola por cada nivel de prioridad
 - Se utiliza un array de registros (el nº de niveles de prioridad es conocido)
 - Para añadir un elemento con prioridad m, se busca la cola que se corresponde con la prioridad m. Si no existe se crea, si existe se añade el elemento en el final
 - Para suprimir un elemento de máxima prioridad, se busca la cola de máxima prioridad no vacía y se suprime su elemento frente.

Listas

Descripción

- Tipo de secuencia enlazada.
- Conjunto ordenado de elementos homogéneos ordenados de acuerdo a las posiciones de estos (relación predecesor-sucesor).
 - El primer elemento será el a_1 (no a_0).
- Su longitud puede variar y se pueden insertar elementos en cualquier posición
- Operaciones: Construcción, posicionamiento, consulta, modificación
- 'principio' será la posición del primer elemento, 'final' es una posición vacía tras el último elemento

Especificación informal

- **lista** = TAD con operaciones crea, fin, primero, siguiente, anterior, esVacia, recupera, longitud, inserta, suprime, modifica
- **Descripción:**
 - Sus valores son del tipo **tipoelem**. Las posiciones son del tipo **posicion**
 - Las listas son mutables: existen funciones **inserta**, **suprime** y **modifica** para añadir, eliminar y modificar elementos.
- **Operaciones:**
 - **crea()** devuelve (**lista**) devuelve la lista vacía
 - **fin(L:lista)** devuelve (**posicion**) devuelve la posición fin (NO el elemento final)
 - **primero(L:lista)** devuelve (**posicion**) devuelve la posición del primer elemento
 - Requerimientos: L no es vacía
 - **siguiente (L:lista; P:posicion)** devuelve (**posicion**) devuelve la posicion siguiente al elemento P
 - Requerimientos: L no es vacía y $P \neq \text{fin}(L)$

- **anterior** (L:lista; P:posicion) devuelve (posicion) devuelve la posicion anterior al elemento P
 - Requerimientos: L no es vacía y P!=**primero**(L)
- **esVacia** (L:lista) devuelve (booleano) devuelve CIERTO si L es vacía
- **recupera** (L:lista; P:posicion) devuelve el elemento de L que ocupa la posicion P
 - Requerimientos: L no es vacía y P!=**fin**(L)
- **inserta** (L:lista; P:posicion; E:tipoelem) modifica L e inserta E como predecesor del elemento que ocupa la posicion P
 - Si P es la posición fin, E pasa a ser el elemento final
- **suprime**(L:lista; P:posicion) elimina de la lista L el elemento en la posición P
 - Requerimientos: L no es vacía y P!=**fin**(L)
- **modifica** (L:lista; P:posicion; E:tipoelem) reemplaza el elemento en la posición P por el nuevo elemento E.
 - Requerimientos: L no es vacía y P!=**fin**(L)

Representaciones contiguas (arrays)

- Los elementos se almacenan en celdas contiguas de un array
- Las posiciones se representan con los índices del array
- En C, se definirá posición como un **puntero a tipoelem**, que apuntará a su respectiva posición en la lista.
 - $I \rightarrow \text{elementos}$ es un puntero al primer elemento de la lista. Para acceder al elemento i , se utiliza $I \rightarrow \text{elementos} + i$.
- **Inconvenientes:** requiere definir un 'max' y insertar/eliminar un elemento requiere reubicar los demás **[O(n)]**
 - Al insertar un elemento en una posición p, se van desplazando 1 hacia la derecha todos los elementos desde el final hasta p

Representaciones enlazadas (punteros)

- Para cada elemento se crea una variable dinámica celda con dos partes: el **elemento** y un **puntero** a la siguiente celda
- Se puede realizar con **simple** o **doble** enlace

Representación con simple enlace

- Una celda está formada por un tipoelem y un puntero a la siguiente celda
- Una lista está formada por dos posiciones (punteros a celda): la de inicio y la del final, y un int de longitud. Posibilidades:
 - 1 - Puntero posición a_i apunta a la celda que contiene a_i , y la posición **fin** apunta a la posición posterior al último elemento
 - Problemas: Insertar en a_i requiere acceder a a_{i-1} para insertar el elemento entre a_{i-1} y a_i lo cual es imposible directamente⁶, requiere recorrer lista
 - 2 - Puntero posición a_i apunta a la celda antes de a_i . La posición **fin** apunta a la posición que contiene el último elemento⁷.
 - Permite acceder directamente a a_{i-1} .
 - La posición de a_1 es un puntero a la celda de encabezamiento.
- **Operaciones** y complejidad algorítmica:
 - Creación de lista vacía es $O(1)$. Se inicializan inicio y fin a la posición inicial, y su campo siguiente a NULL, con longitud=0.
 - Destrucción de lista libera todos los elementos uno por uno desde el principio, luego libera la lista en si. $O(n)$
 - Operaciones de consulta son $O(1)$ [esVacia, recupera, longitud]
 - Acceder a la posición anterior de un elemento es $O(n)$
 - Para insertar un elemento en una posición p [$O(1)$]:
 - La celda actualmente siguiente a p se guarda como q

⁶ Cada celda solo tiene como información su contenido y un puntero a la siguiente, pero no a la previa. Es imposible pasar directamente de una celda a su anterior.

⁷ En una lista vacía, será un puntero a la lista de encabezamiento, igual que a_1

- Se reserva espacio para una nueva celda, que será la siguiente a p. Su contenido será el que se quiere insertar y su siguiente será q.
 - Si q es nula, se modifica el final de la lista.
- La longitud de la lista aumenta en 1
- Para suprimir un elemento en la posición p [$O(1)$]:
 - Se declara q como el siguiente a p, que será el elemento suprimido
 - El siguiente a p pasa a ser el siguiente a q. Si es nulo, p pasa a ser el final de la lista.
 - Se libera q. La longitud de la lista disminuye en 1

Representación con doble enlace

- Una **celda** incluye su contenido, un puntero a la celda siguiente y otro puntero a la celda anterior.
- Una **lista** incluye un puntero al fin de la lista y la longitud
- El puntero posición a_i apunta a la celda a_i
 - Para representar la posición fin de lista, se puede dejar una celda final vacía que se crea al declarar la lista vacía
- Ventajas: punteros posición más intuitivos, no es necesario actualizar fin, permite acceder a ambos extremos en $O(1)$, mayor eficiencia al recorrer lista en ambos sentidos
- Inconvenientes: Dos elementos de información adicional por elemento en la lista
- **Operaciones** y complejidad algorítmica:
 - Creación [$O(1)$]: reserva espacio para lista y celda final, cuyos campos ant y sig apuntan a si misma
 - Destrucción [$O(n)$]: libera cada celda y después la lista
 - Posicionamiento [$O(1)$]: primero, fin, siguiente(p), anterior(p)
 - Consulta [$O(1)$]: esVacia, recupera, longitud

- Inserta [$O(1)$]: Reserva espacio, guarda elemento y modifica enlaces previos y posteriores
- Suprime [$O(1)$]: Modifica enlaces, libera memoria de elemento suprimido

Comparación de implementaciones

- **Resumen:** Contigua es $O(n)$ para insertar/suprimir, simple enlace es $O(n)$ para destruir/anterior, doble enlace solo para destruir.
- Si no se conoce la longitud estimada o los elementos son grandes es mejor enlazada

Tipos de listas

- **Pila:** lista donde solo se puede insertar/suprimir por el primer elemento. Es óptimo utilizar listas de enlace simple.
- **Colas:** Lista donde solo se puede insertar por el final y suprimir por el principio. Es óptimo utilizar listas de enlace simple.
- **Listas ordenadas:** Al insertar un elemento no se escoge la posición, sino que viene calculada por una función POSICIÓN posinser (TLISTA I, ELEMENTO e)
 - No se permiten duplicados → función para eliminar duplicados
- **Lista circular:** El siguiente del último elemento es el primero
 - No confundir con implementación interna (ej doble enlace)

Ejemplos de implementación TADs

TAD Bolsa

- TAD contenedor
- Definición mutable
- Representación enlazada
- **NOTA:** las funciones que sólo acceden a una bolsa reciben una bolsa b, las funciones que modifican una bolsa reciben un puntero a bolsa *b.
 - Esto se aplica a todos los TAD y se pregunta mucho en los tests de prácticas

bolsa.h

```
#include "errores.h"
```

```
typedef void * bolsa; //un tipo opaco  
typedef int tipoelem;
```

```
void bolsavacia(bolsa *b);
```

```
void poner (bolsa *b, tipoelem e, codigo_error * cod);
```

```
int esvacia (bolsa b);
```

```
short cuantos (bolsa b, tipoelem e);
```

```
void dest (bolsa *b);
```

bolsa.c

```
#include <stdlib.h>
#include "errores.h"

typedef int tipoelem;
typedef struct celda {
    tipoelem elemento;
    struct celda * siguiente;
} tipocelda; //Crea un tipo de datos 'struct celda' e pasa a chamarlle 'tipocelda'.
              tipocelda ten dous componentes: un elemento (de tipo int) e un
              punteiro a outro struct celda, que será o seguinte.

typedef tipocelda * puntero;
typedef puntero bolsa; //(non sei por que fai o paso intermedio pero) define 'bolsa'
                        como un punteiro a tipocelda. Desta forma, 'bolsa' será un
                        punteiro ao primeiro elemento da bolsa.

void bolsavacia(bolsa *b) {
    *b=NULL;           //Non libera memoria, simplemente marca unha bolsa como
}                      vacía. Para liberar está a función dest

void poner (bolsa *b, tipoelem e, codigo_error * cod) { // nótese que colle como
// parámetro un punteiro a bolsa (sería un punteiro a un punteiro a unha struct)
    puntero aux;
    aux=(puntero) malloc(sizeof(tipocelda)); // Reserva memoria para unha nova
                                                celda, onde irá o elemento.

    if (aux == NULL)
        *cod = meminsu;
    else {
        aux->elemento = e; // Garda o novo elemento na celda creada
        aux->siguiente = *b; // O elemento 'seguinte' ao novo elemento será o
                              previo primeiro elemento da bolsa. Desta forma,
                              este novo elemento continúa enlazado co resto
                              da bolsa.

        *b = aux;           // A bolsa toma agora 'aux' como o primeiro
                              elemento. Collese *b porque b aquí é un
                              punteiro a bolsa, non unha bolsa.

        *cod = exito;
    }
}
```

}

}

}

TAD Pila (implementación con punteros)

- TAD contenedor
- Implementación enlazada
- Un elemento pila es un puntero al tope, la primera celda.
- Un elemento celda contiene su elemento y un puntero a la celda siguiente

pila.h

```
typedef void * TPILA; //un tipo opaco
typedef int tipoelem;

void PilaVacía(TPILA *p);

int EsVacía (TPILA b);

void Push(TPILA *b, TELEMENTO e);

void Cima(TPILA *b, TELEMENTO *pe);

void Pop(TPILA *b);

void leer(TPILA *P, short N)
```

pila.c

```
typedef int tipoelem;

typedef struct nodo {
    TELEMENTO dato;
    struct nodo * sig; } TNodo;

typedef TNodo* TPILA;

void PilaVacía (TPILA *b) {
    *b=NULL;           //Non libera memoria, simplemente marca unha pila como vacía
}
```

```

int EsVacia(TPILA p) {
    if (p==NULL) return 1;
    else return 0;
}

void Push(TPILA * p , TELEMENTO e) {
    TPILA q;
    q= (TPILA) malloc (sizeof(TNodo)); // se define nueva pila
    q->dato = e ; // se pone en la cima de la pila el nuevo elemento
    q->sig = *p ; // debajo de la cima está la pila previa
    *p = q;
}

void Cima (TPILA p, TELEMENTO * pe) {
    int respuesta = EsVacia(p);
    if (respuesta == 1)
        printf ("ERROR, la pila no tiene elementos\n");
    else *pe = p->dato;
}

void Pop(TPILA * p) {
    TPILA q;
    int respuesta;
    respuesta = EsVacia(*p);
    if (respuesta==1)
        printf("ERROR, ....\n");
    else {
        q = *p;
        *p = (*p)->sig; // la nueva cima es el siguiente de la cima actual
        free(q); // se libera la previa cima
    }
}

void leer(TPILA *P, short N) { //TPILA es un puntero a estructura pila
    TELEMENTO e;
    short i;
    PilaVacia(P);
    for (i=1 ; i<=N ; i++) {
        printf("Dime un entero:");
        scanf("%d", &e);
        push(P,e);
    }
}

```

```

void imprimir (TPILA *pila) {
    TPILA aux;
    TELEMENTO E;

    PilaVacía(&aux); // fija aux a null
    while (esVacía(*pila) == 0 ){
        cima(*pila,&E); // guarda la cima actual en E
        pop(pila); // elimina la cima actual de la pila
        push(&aux, E); // añade la cima actual a aux
        printf("%d\n",E); // imprime la cima actual
    } // de esta forma, se vacían todos los elementos de P en aux, cambiadas de
    orden

    while (esVacía(aux) == 0 )
    {
        cima(aux,&E);
        pop(&aux);
        push(pila, E);
    } // se hace lo mismo para volver a guardar los elementos en pila, vuelven a
    quedar en el orden correcto
}

```

TAD Cola (implementación con vector circular)

- TAD contenedor
- Implementación contigua
- Formado por un vector de elementos y dos enteros: uno indica la posición de inicio y otro la posición inicial
- Se sacrifica un elemento del array para evitar confusión entre cola llena y cola vacía.
 - Cola vacía: principio == final
 - Cola llena: Principio == Siguiente(final)
 - Permite vaciar la cola por completo, pero no rellenarla por completo. Si está rellena excepto por la casilla sacrificada, contará como cola llena.

cola.h

```
typedef void * TCOLA;      //un tipo opaco
typedef int tipoelem;      //aqui pon int pero podria ser outra coisa

void ColaVacia(TCOLA *q);

void AnadirCola(TCOLA *q, tipoelem e);

int EsColaVacia (TCOLA q);

void EliminarCola(TCOLA *q);

void PrimeroCola(TCOLA q, tipoelem *pe);

int Siguiente(int pos);
```


cola.c

```
#define MAX 100 //permite almacenar 99 elementos por lo descrito antes
```

```
typedef int TELEMENTO;
```

```
typedef struct {  
    TELEMENTO arrayelementos[MAX];  
    int principio, final;  
} STCOLA;
```

```
typedef STCOLA* TCOLA;
```

```
int Siguiente(int pos) { //devuelve un entero, la posición siguiente a pos  
    if (pos < MAX -1) return (pos+1);  
    else return 0;  
}
```

```
void ColaVacia(TCOLA *q) { //reserva espacio para una nueva cola.  
    *q = (TCOLA) malloc (sizeof(STCOLA));  
    (*q)->final = MAX -1 ;    //el primer objeto añadido ocupará el índice siguiente  
    (*q)->principio = MAX -1; que es 0  
}
```

```
int EsColaVacia(TCOLA p) { //vacía si es recién creada o si se ha eliminado el último  
    elemento. en estas circunstancias, el índice del final será igual que el del principio.  
    if (q->final==q->principio) return 1;  
    else return 0;  
}
```

```
int EsColaLlena(TCOLA p) { // llena si la casilla siguiente a la última ocupada es  
    q->principio, que es la casilla que intencionadamente se deja vacía.  
    if (Siguiente(q->final)==q->principio) return 1;  
    else return 0;  
}
```

```
void AnadirCola(TCOLA * q , TELEMENTO e) { // añade un elemento en la posición  
                                         siguiente a la final
```

```
    TCOLA q;  
    int respuesta;  
    respuesta = EsColaLlena(*q);  
    if (respuesta==1)  
        printf("ERROR, ....\n");  
    else {  
        (*q)->final = Siguiente((*q)->final) // la cola crece en 1  
        (*q)->arrayelementos[(*q)->final] = e; }  
}
```

```
void PrimeroCola (TCOLA p, TELEMENTO * pe) { // guarda en *pe el primer  
elemento de la cola.
```

```
    int respuesta = EsVacia(p);  
    if (respuesta == 1) printf ("ERROR, la pila no tiene elementos\n");  
    else *pe = q -> arrayelementos[Siguiente(q->principio)]; // q->principio es la  
casilla vacía, se debe tomar la siguiente  
}
```

```
void EliminarCola (TCOLA * p) { // elimina el primer elemento de la cola
```

```
    TCOLA q;  
    int respuesta;  
    respuesta = EsVacia(*p);  
    if (respuesta==1) printf("ERROR, ....\n");  
    else (*q)->principio = Siguiente(q->principio); //no se borra el elemento, se  
avanza el principio en 1. la cola disminuye 1 elemento
```

//al ser una representación con array no se hace free() al eliminar cada
elemento, permanentemente está reservada la memoria para la lista
completa aunque no se use. se liberaría al final de usar la lista con una
función distinta de destruirlista.

```
}
```

TAD Cola (implementación con punteros)

- TAD contenedor
- Implementación enlazada
- Estructura formada por dos punteros a estructuras, denominadas nodos. Una será el nodo de principio y otra el final.
- Los nodos en si mismos están formados por su elemento y un puntero al siguiente.
- El archivo.h sería idéntico al previo.

```
typedef int TELEMENTO;
typedef struct nodo {
    TELEMENTO dato;
    struct nodo * sig; } TNode;
typedef struct {TNode * principio, * final; } STCOLA;
typedef STCOLA * TCOLA;
```

```
void ColaVacia ( TCOLA * q ) { // Reserva memoria y define nueva cola
    *q = (TCOLA) malloc (sizeof (STCOLA) ) ;
    (*q)->final = NULL; (*q)->principio = NULL;
}
```

```
int EsColaVacia ( TCOLA q ) {
    if ( (q->final == NULL) && (q->principio == NULL) ) return 1;
    else return 0;
}
```

```
void PrimeroCola ( TCOLA q, TELEMENTO * e) { // Guarda en *e el primer
elemento de la cola
    int respuesta;
    respuesta = EsColaVacia(q);
    if ( respuesta == 1) printf("ERROR, la cola no tiene elementos");
    else *e = (q->principio)->dato;
}
```

```

void EliminarCola (TCOLA * q) { // Elimina el primer elemento de la cola
    int respuesta;
    TNode * aux;
    respuesta = EsColaVacia(*q);
    if ( respuesta == 1 ) printf("ERROR, cola vacía");
    else {
        aux = (*q)->principio;           // se guarda en aux para poder
                                          // liberarlo luego
        (*q)->principio = aux->sig;       // el principio avanza 1
        posición
        if ((*q)->principio == NULL)      // si la lista queda vacía
            (*q)->final = NULL;          // se declara como vacía anulando
                                          // también el nodo final

        free (aux);
    }
}

```

```

void AnadirCola (TCOLA * q , TELEMENTO e) { // añade un elemento al final de la
                                          // lista

    int respuesta;
    TNode * aux;
    aux = (TNode *) malloc (sizeof (TNode) ); // se crea la nueva casilla, que
    aux->dato = e;                             // toma como dato e y como
    aux->sig = NULL;                             // siguiente NULL
    respuesta = EsColaVacia (*q);
    if (respuesta == 1) (*q)->principio = aux; // si era vacía, la nueva casilla es la
                                          // primera.
    else (*q)->final->sig = aux;              // si no, el previo elemento final
                                          // toma como siguiente el añadido
    (*q)->final = aux;                        // el elemento añadido es ahora el
                                          // final
}

```

TAD Lista(implementación con vector circular)

- TAD contenedor
- Implementación contigua
- Formado por un vector dinámico de elementos 'POSICION' y un entero que define la longitud actual
 - 'POSICION' es un puntero a tipoelem
- Al crear el vector vacío se reserva con malloc la memoria maxima que puede ocupar (requiere constante MAX)

lista.h

```
typedef void * TLISTA;      //un tipo opaco
typedef void * POSICION;
typedef int tipoelem;       //aqui pon int pero podria ser outra cousa
```

```
void crea(TLISTA *l);
void destruye(TLISTA *l);

POSICION primero(TLISTA *l);
POSICION fin(TLISTA *l);
POSICION siguiente(TLISTA *l, POSICION p);
POSICION anterior(TLISTA *l, POSICION p);

int esVacia(TLISTA *l);
int longitud(TLISTA *l);
void recupera(TLISTA *l, POSICION p; tipoelem *e);

void inserta(TLISTA *l, POSICION p; tipoelem e);
void suprime(TLISTA *l, POSICION p);
void modifica(TLISTA *l, POSICION p; tipoelem e);
```

lista.c

```
#define MAX 100
```

```
typedef int TIPOELEM;
```

```
typedef TIPOELEM * POSICION;
```

```
typedef struct {  
    POSICION elementos;  
    int longitud;  
} TCELDA;
```

```
typedef TCELDA * TLISTA;
```

```
void crea(TLISTA *l) { //reserva espacio para una nueva lista  
    *l = (TCELDA *) malloc (sizeof(TCELDA));  
    (*l)->elementos = (POSICION) malloc (MAX * sizeof(TIPOELEM));  
    (*l)->longitud = 0;  
}
```

```
void destruye(TLISTA *l) {  
  
    free((*l)->elementos); //l porque l es un puntero a puntero a celda  
    free(*l);  
}
```

```
POSICION primero(TLISTA l) {  
  
    return l->elementos; //elementos es un vector dinámico, por lo que es un  
                           puntero a su primer elemento, que es de tipo  
                           POSICION  
}
```

```
POSICION fin(TLISTA l) {  
  
    POSICION final = l->elementos + l->longitud ;  
  
    return final;
```

```
}  
POSICION siguiente(TLISTA l, POSICION p) {  
    return p+1;  
}
```

```
POSICION anterior(TLISTA l, POSICION p) {  
    return p-1;  
}
```

```
int esVacia(TLISTA l) {  
    if (l->longitud == 0) return 1;  
    else return 0;  
}
```

```
void recupera(TLISTA l, POSICION p, TIPOELEM *e) {  
    *e = *p;  
}
```

```
int longitud(TLISTA l) {  
    return l->longitud;  
}
```

```

void inserta(TLISTA *l, POSICION p, TIPOELEM e) {
    POSICION q, r;

    q=fin(*l);    //empieza desde el final [fin es la posición que va después del
                  //último elemento], y va retrocediendo hasta

    while (q!=p) { //llegar a la posición donde se quiere insertar e

        r=q;    // r es la posición actual

        q--;    // q retrocede un paso

        *r = *q; // se guarda en r su elemento predecesor, avanzando la lista 1
                // posición hacia la derecha

    } // al final del bucle,

    *p=e; //se guarda en p el nuevo valor (el previo valor de p ya se guardó en
          //p+1)

    (*l)->longitud++;
}

```

```

void supprime(TLISTA *l, POSICION p) {
    POSICION q,r;

    (*l)->longitud -- ;

    q=p; //empieza desde p, donde se guardará el valor de p+1

    while (q!=fin(*l)) {

        r=q;

        q++;

        *r=*q; //en cada casilla se guarda el valor de la casilla posterior

    }

}

void modifica(TLISTA *l, POSICION p, TIPOELEM e) {

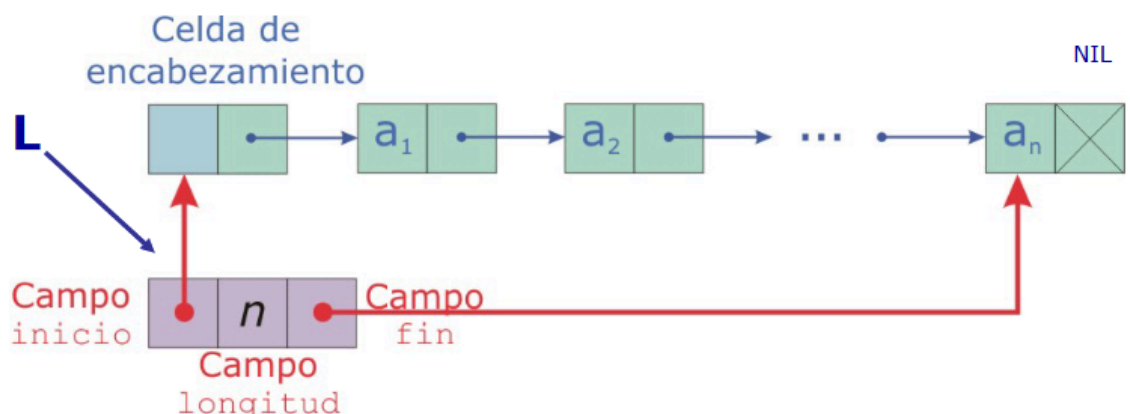
    *p=e;

}

```


TAD lista (implementación con punteros con simple enlace)

- TAD contenedor
- Implementación enlazada
- Una **celda** está formada por un elemento y un puntero a la siguiente celda
- Una **POSICION** es un puntero a una celda
 - La posición de a_i es un puntero a la celda que precede a a_i
 - La posición fin es un puntero a la celda que contiene el último elemento
 - La posición inicio es un puntero a la celda de encabezamiento
- **TLISTA** es una estructura formada por dos posiciones, inicio y fin, y un entero de longitud
- La celda de encabezamiento se mantiene vacía.
- El archivo.h sería idéntico al previo.



lista.c

```
typedef int TIPOELEM;
typedef struct celda{
    TIPOELEM elemento;
    struct celda* sig; } TCELDA;
typedef TCELDA * POSICION;

typedef struct lista {
    POSICION inicio;
    POSICION fin;
    int longitud; } TLISTA;
```

```

void crea( TLISTA *l ) { //Reserva memoria y define nueva lista
    *l=(TLISTA *) malloc(sizeof(TLISTA));
    (*l)->inicio = (POSICION) malloc(sizeof(TCELDA)); //Celda de
    encabezamiento
    (*l)->inicio->sig = NULL; /
    (*l)->fin = (*l)->inicio ; //Inicio y fin apuntan a celda encabezamiento
    (*l)->longitud=0;
}

```

```

void destruye(TLISTA *l) { //Elimina lista completa

    (*l)->fin = (*l)->inicio; //Puntero fin apunta a celda de encabezamiento
    while ((*l)->fin != NULL) {
        (*l)->fin = (*l)->fin->sig; //Recorre la lista de principio a fin, liberando
        free((*l)->fin); //cada elemento
        (*l)->inicio = (*l)->fin;
    }
    free(*l);
}

```

```

POSICION primero(TLISTA l) {

    return l->inicio;

}

```

```

POSICION fin(TLISTA l) {

    return l->fin;

}

```

```

POSICION siguiente(TLISTA l, POSICION p) {

    return p->sig;

}

```

```

int esVacia(TLISTA l) {
    if (l->longitud == 0) return 1;
    else return 0;
}

void recupera(TLISTA l, POSICION p, TIPOELEM *e) {
    *e = p->sig->elemento;
}

int longitud(TLISTA l) {
    return l->longitud;
}

void inserta(TLISTA *l, POSICION p, TIPOELEM e) {
    POSICION q ;
    q=p->sig;
    p->sig = (TCELDA *) malloc(sizeof(TCELDA)); //se reserva espacio después
                                                de p
    p->sig->elemento = e;    //se guarda en esta celda el elemento
    p->sig->sig = q;        //se enlaza con el resto de la lista
    if (q==NULL) (*l)->fin=p->sig; //si el elemento se añade al final de la lista,
                                    pasa a ser el nuevo final
    (*l)->longitud ++ ;
}

```

```

void supprime(TLISTA *l, POSICION p) {
    POSICION q;

    q=p->sig;          //q será el elemento a eliminar

    p->sig=q->sig;      //se enlaza p con el siguiente a q

    if (p->sig==NULL)   //si no existe, p pasa a ser el último de la lista
        (*l)->fin = p;

    free(q);           //dado que p apunta al elemento que precede a ap, su siguiente,
                        //q, apunta a ap, por lo que será el puntero a liberar para suprimir ap

    (*l)->longitud -- ;
}

void modifica(TLISTA *l, POSICION p, TIPOELEM e) {

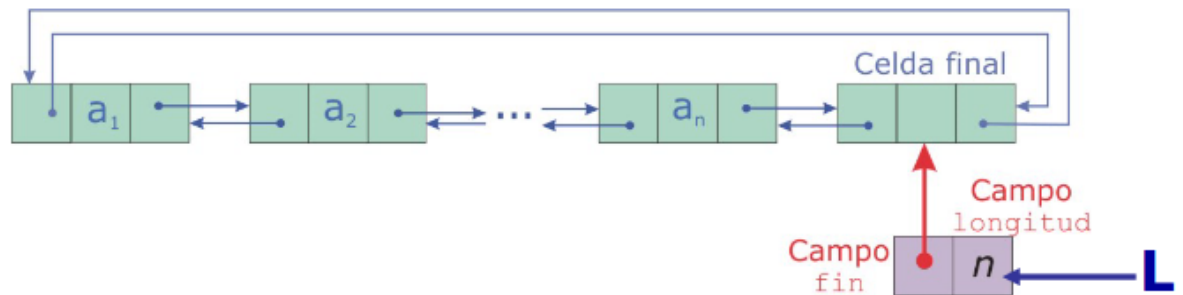
    p->sig->elemento=e; //se toma el siguiente porque p es un puntero al
                        //elemento que precede a ap

}

```

TAD lista (implementación con punteros con doble enlace)

- TAD contenedor
- Implementación enlazada
- Una **celda** está formada por un elemento, un puntero a la siguiente celd y otro puntero a la anterior
- Una **POSICION** es un puntero a una celda
 - La posición de a_i es un puntero a la celda que contiene a_i
 - La posición fin es un puntero a la celda final, que está vacía.
- **TLISTA** es una estructura formada por la posición fin y la longitud
- El archivo.h sería idéntico al previo.



lista.c

```
typedef int TIPOELEM;
typedef struct celda{
    TIPOELEM elemento;
    struct celda* ant,sig; } TCELDA;
typedef TCELDA * POSICION;

typedef struct lista {
    POSICION fin;
    int longitud; } TLISTA;
```

```

void crea( TLISTA *l ) { //Reserva memoria y define nueva lista
    *l=(TLISTA *) malloc(sizeof(TLISTA));
    (*l)->fin = (POSICION) malloc(sizeof(TCELDA)); //Celda final
    (*l)->fin->sig = *l->fin; //los campos ant y sig de la celda final apuntan a si
    (*l)->fin->ant = *l->fin; //misma
    (*l)->longitud=0;
}

```

```

void destruye(TLISTA *l) { //Elimina lista completa

```

```

    puntero aux=*l->fin, aux2;
    while (aux!=NULL) {
        aux2 = aux;
        aux = aux->sig;
        free(aux2);
        *l->fin = aux ;
    }
    free(*l);
}

```

```

POSICION primero(TLISTA l) {
    return l->fin->sig;
}

```

```

POSICION fin(TLISTA l) {
    return l->fin;
}

```

```

POSICION siguiente(TLISTA l, POSICION p) {
    return p->sig;
}
POSICION anterior(TLISTA l, POSICION p) {
    return p->ant;
}

```

```

int esVacia(TLISTA l) {
    if (l->longitud == 0) return 1;
    else return 0;
}

void recupera(TLISTA l, POSICION p, TIPOELEM *e) {
    *e = p->elemento;
}

int longitud(TLISTA l) {
    return l->longitud;
}

void inserta(TLISTA *l, POSICION p, TIPOELEM e) {
    POSICION q ;
    q=p->ant;
    p->ant = (TCELDA *) malloc(sizeof(TCELDA)); //se reserva espacio antes
                                                    de p
    p->ant->elemento = e;    //se guarda en esta celda el elemento
    p->ant->ant = q;        //se enlaza con el resto de la lista
    p->ant->sig = p;
    q->sig = p->ant;

    //(creo que) non hai que facer ningun caso especial para comprobar se se
    está añadiendo ao principio ou final da lista. por exemplo se añades na
    posición inicial, quedará ese elemento no principio da lista, co previo
    elemento inicial como seguinte e coa celda fin como anterior, o cal é correcto

    (*l)->longitud ++ ;
}

```

```
void supprime(TLISTA *l, POSICION p) {  
    POSICION q;  
    q=p->ant;  
    q->sig=p->sig;    //se completan los enlaces sin incluir p  
    free(p);  
    (*l)->longitud -- ;  
}  
  
void modifica(TLISTA *l, POSICION p, TIPOELEM e) {  
    p->elemento=e;  
}
```