Departamento
de Ingeniería de la Información y las Comunicaciones
UMU

Tecnología de la Programación

Título de Grado en Ingeniería Informática

Curso 2024/25

Grupos 2, 3 y 4

Tema 3. TDAs Fundamentales

Dept. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

Tema 3. TDAs Fundamentales

3.1 Motivación y definiciones previas (1/3)

Determinados TDAs adquieren **relevancia** ya que son usados en **multitud de aplicaciones**. En estos casos, resulta conveniente establecer **modelos generales** para ellos de forma que puedan ser utilizados en cualquier aplicación.

Por otra parte, la implementación de **nuevos TDAs** puede basarse en **TDAs ya construidos**, estableciéndose así un doble nivel de abstracción.

En este tema trataremos los siguientes TDAs fundamentales, por ser de los más usados en la mayoría de las aplicaciones: ***Listas***, ***Colas***, ***Pilas*** y ***Conjuntos*.**

3.1 Motivación y definiciones previas (2/3)

Las Listas, Colas y Pilas son tipos de datos abstractos que se corresponden con secuencias de cero o más elementos, es decir, existe una relación sucesor-predecesor. A diferencia de los Conjuntos, en estos TDAs los elementos pueden repetirse.

Las Listas son el caso más general. En ellas, los accesos, inserciones y eliminaciones pueden realizarse en cualquier lugar de la Lista.

Las Colas son casos particulares de Listas en los que las inserciones se realizan por un extremo mientras que los accesos y eliminaciones se realizan por el extremo opuesto. Por ello, las Colas son denominadas también Listas *FIFO* (*First In First Out*).

Las Pilas son otros casos particulares de Listas en los que las inserciones, accesos y eliminaciones se realizan siempre por el mismo extremo denominado tope de la Pila. Por ello, las Pilas son denominadas también Listas *LIFO* (*Last In First Out*).

3.1 Motivación y definiciones previas (3/3)

En esta sección se verán **especificaciones informales** para Listas, Colas, Pilas y Conjuntos, las cuales se introducirán en orden creciente de complejidad para un mejor aprendizaje, es decir, primero Pilas, después Colas, después Listas, y finalmente Conjuntos.

Se verán implementaciones con **representaciones contiguas y enlazadas**, con **simple y doble enlace** haciendo referencia a la **complejidad de las operaciones** y **requerimientos de memoria** para cada una de las implementaciones.

Se citarán algunas **variantes** de Listas, Colas y Pilas así como algunas **aplicaciones** típicas de estos TDAs.

Finalmente se mostrarán algunos **ejemplos** que usan estos TDAs para resolver determinados problemas.

3.2 Especificación del TDA Pila (1/2)

TDA Pila. Definición

El TDA Pila es un TDA contenedor que define una secuencia de cero o más elementos de un tipo homogéneo (tipo `Elemento`) en el cual las inserciones, accesos y supresiones de elementos se realizan siempre por el mismo extremo, denominado ***tope***. Se denomina también lista ***LIFO*** (*Last In First Out*).

El TDA Pila es mutable en esta especificación.

3.2 Especificación del TDA Pila (2/2)

TDA Pila. Operaciones

`Pila crea()`

efecto: devuelve una nueva pila vacía.

`void inserta(Pila p, Elemento e)`

efecto: inserta el elemento `e` en el tope de la pila `p`.

`void suprime(Pila p)`

precondición: la pila `p` no es vacía.

efecto: suprime el elemento situado en el tope de la pila `p`.

`Elemento recupera(Pila p)`

precondición: la pila `p` no es vacía.

efecto: devuelve el elemento situado en el tope de la pila `p`.

`int vacia(Pila p)`

efecto: devuelve 0 (falso) si la pila `p` no es vacía, y 1 (cierto) en caso contrario.

3.2 Especificación del TDA Cola (1/2)

TDA Cola. Definición

El TDA Cola es un TDA contenedor que define una secuencia de cero o más elementos de un tipo homogéneo (tipo `Elemento`) en la cual las inserciones de elementos se realizan en un extremo, denominado ***posterior o final*** (quedando el elemento como último de la cola), y los accesos y supresiones de elementos se realizan por el otro extremo, denominado ***anterior o frente*** (el primero de la cola). Se denomina también lista ***FIFO*** (*First In First Out*).

El TDA Cola es mutable en esta especificación.

3.2 Especificación del TDA Cola (2/2)

TDA Cola Operaciones

`Cola crea()`

efecto: devuelve una nueva cola vacía.

`void inserta(Cola c, Elemento e)`

efecto: inserta el elemento `e` en el último lugar de la cola `c`.

`void suprime(Cola c)`

precondición: la cola `c` no es vacía.
efecto: suprime el elemento situado en el primer lugar de la cola `c`.

`Elemento recupera(Cola c)`

precondición: la cola `c` no es vacía.
efecto: devuelve el elemento situado en el primer lugar de la cola `c`.

`int vacia(Cola c)`

efecto: devuelve 0 (falso) si la cola `c` no es vacía, y 1 (cierto) en caso contrario.

3.2 Especificación del TDA Lista (1/5)

TDA Lista. Definición

El TDA Lista es un TDA contenedor que define una secuencia de cero o más elementos de un tipo homogéneo (tipo `Elemento`) ordenada de acuerdo a las posiciones de estos (relación predecesor-sucesor):

$$L = (a_1, a_2, \ldots, a_n)$$

donde $n \geq 0$, y $a_i \in$ `Elemento`, para $i = 1,\ldots,n$.

- n: *longitud* de la lista.
- Si $n = 0$, L es la *lista vacía*.
- a_1 es el *primer* elemento.
- a_n es el *último* elemento.
- a_i ocupa la *i-ésima posición*, $i = 1,\ldots,n$.
- a_i es *predecesor* de a_{i+1}, $i = 1,\ldots,n-1$.
- a_i es *sucesor* de a_{i-1}, $i = 2,\ldots,n$.

El TDA Lista es mutable en esta especificación.

3.2 Especificación del TDA Lista (2/5)

TDA Lista. Definición

La **longitud** de una lista aumenta al insertar elementos y disminuye al suprimirlos.

Los **accesos, inserciones y supresiones** pueden realizarse en **cualquier lugar de la lista**.

El tipo `Posicion` representa **posiciones válidas** de una lista, y será utilizado para establecer las posiciones de acceso, inserción o supresión.

Para cada elemento de la lista existirá un valor del tipo `Posicion` asociado. Adicionalmente existe un valor del tipo `Posicion` que indicará el **fin de la lista,** y se utilizará para la inserción de elementos al final de la lista y como marca de fin de lista:

$$L = (\underset{p(a_1)}{a_1}, \underset{p(a_2)}{a_2}, \ldots, \underset{p(a_n)}{a_n}\ \underset{p(fin)}{})$$

donde $p(a_i) = posición\ de\ a_i$ y $p(fin) = posición\ fin$.

3.2 Especificación del TDA Lista (3/5)

TDA Lista. Operaciones

Operaciones de Construcción y Acceso:

```
Lista crea()
void inserta(Lista l, Posicion p, Elemento e)
void suprime(Lista l, Posicion p)
Elemento recupera(Lista l, Posicion p)
void asigna(Lista l, Posicion p, Elemento e)
int longitud(Lista l)
```

Operaciones de Posicionamiento:

```
Posicion inicio(Lista l)
Posicion fin(Lista l)
Posicion iesima(Lista l, int indice)
Posicion siguiente(Lista l, Posicion p)
Posicion anterior(Lista l, Posicion p)
```

3.2 Especificación del TDA Lista (4/5)

TDA Lista. Operaciones

`Lista crea()`

efecto: devuelve una nueva lista vacía.

`void inserta(Lista l, Posicion p, Elemento e)`

precondición: `p` es una posición de la lista `l`.

efecto: inserta `e` en la posición `p` de la lista `l`. Los casos del tipo `Posicion` quedan indefinidos.

`void suprime(Lista l, Posicion p)`

precondición: `p` es una posición de la lista `l` distinta de la posición fin de la lista `l`.

efecto: suprime el elemento situado en la posición `p` de la lista `l`. Los casos del tipo `Posicion` quedan indefinidos.

`Elemento recupera(Lista l, Posicion p)`

precondición: `p` es una posición de la lista `l` distinta de la posición fin de la lista `l`.

efecto: devuelve el elemento situado en la posición `p` de la lista `l`.

`void asigna(Lista l, Posicion p, Elemento e)`

precondición: `p` es una posición de la lista `l` distinta de la posición fin de la lista `l`.

efecto: asigna el elemento `e` en la posición `p` de la lista `l`.

`int longitud(Lista l)`

efecto: devuelve el número de elementos de la lista `l`.

3.2 Especificación del TDA Lista (5/5)

TDA Lista. Operaciones

`Posicion inicio(Lista l)`

efecto: devuelve la posición del primer elemento de la lista `l` si la lista `l` no es vacía, en otro caso devuelve la posición fin de la lista `l`.

`Posicion fin(Lista l)`

efecto: devuelve la posición fin de la lista `l`.

`Posicion iesimo(Lista l, int i)`

precondición: `1<=indice<=longitud(l)`

efecto: devuelve la posición del elemento `i`-ésimo de la lista `l`.

`Posicion siguiente(Lista l, Posicion p)`

precondición: la lista `l` no es vacía y `p` es una posición de la lista `l` distinta de la posición fin de la lista `l`.

efecto: devuelve la posición del elemento sucesor al elemento que ocupa la posición `p` de la lista `l`.

`Posicion anterior(Lista l, Posicion p)`

precondición: la lista `l` no es vacía y `p` es una posición de la lista `l` distinta de la posición del primer elemento de la lista `l`.

efecto: devuelve la posición del elemento predecesor al elemento que ocupa la posición `p` de la lista `l`.

3.2 Especificación del TDA Conjunto (1/2)

TDA Conjunto. Definición

El TDA Conjunto es un TDA contenedor que representa una agrupación de elementos no repetidos de un tipo homogéneo (tipo `Elemento`).

El TDA Conjunto es mutable en esta especificación.

3.2 Especificación del TDA Conjunto (1/2)

TDA Conjunto. Operaciones

`Conjunto crea()`

efecto: Devuelve un nuevo conjunto vacío.

`void inserta(Conjunto c, Elemento e)`

efecto: inserta el elemento `e` en el conjunto `c` si `e` no pertenece a `c`.

`void suprime(Conjunto c, Elemento e)`

efecto: suprime el elemento `e` del conjunto `c` si `e` pertenece a `c`.

`int pertenece(Conjunto c, Elemento e)`

efecto: Devuelve 0 (falso) si el elemento `e` no pertenece al conjunto `c`, y 1 (cierto) en caso contrario.

`int cardinalidad(Conjunto c)`

efecto: Devuelve el número de elementos del conjunto `c`.

`Lista enumera(Conjunto c)`

efecto: Devuelve una lista con los elementos del conjunto `c`.

3.6 Implementaciones del TDA Pila

3.6.1 Implementación del TDA Pila: Fichero de cabecera

3.6.2 Implementación del TDA Pila con representación contigua

3.6.3 Implementación del TDA Pila con representación enlazada

3.6.1 Implementación del TDA Pila: Fichero de cabecera

Pila.h

```
#ifndef __PILA_H
#define __PILA_H

typedef int Elemento;
typedef struct PilaRep * Pila;

Pila crea();
void libera(Pila p);
void inserta(Pila p, Elemento e);
void suprime(Pila p);
Elemento recupera(Pila p);
int vacia(Pila p);
int llena(Pila p); // Sólo para Pilas acotadas

#endif
```

3.6.2 Implementación del TDA Pila con representación contigua (1/2)

Pila.c

```
#include "Pila.h"
#include <stdlib.h>
#define MAX 100
struct PilaRep
{
     Elemento elem[MAX];
     int tope;
};
Pila crea()
{
     Pila p = malloc(sizeof(struct PilaRep));
     p->tope = -1;
     return p;
}
void libera(Pila p) { free(p); }
```

3.6.2 Implementación del TDA Pila con representación contigua (2/2)

Pila.c

```
void inserta(Pila p, Elemento e)
{
     p->tope++;
     p->elem[p->tope] = e;
}

void suprime(Pila p) { p->tope--; }

Elemento recupera(Pila p) { return p->elem[p->tope]; }

int vacia(Pila p) { return (p->tope==-1); }

int llena(Pila p) { return (p->tope==MAX-1); }
```

3.6.3 Implementación del TDA Pila con representación enlazada (1/2)

Pila.c

```
#include "Pila.h"
#include <stdlib.h>
struct PilaRep {
      Elemento elem;
      Pila sig;
};
Pila crea() {
      Pila p = malloc(sizeof(struct PilaRep));
      p->sig = NULL;
      return p;
}
void libera(Pila p) {
      while(p!=NULL)
      {
            Pila aux = p;
            p = aux->sig;
            free(aux);
      }
}
```

3.6.3 Implementación del TDA Pila con representación enlazada (2/2)

Pila.c

```
void inserta(Pila p, Elemento e)
{
      Pila nuevo = malloc(sizeof(struct PilaRep));
      nuevo->elem = e;
      nuevo->sig = p->sig;
      p->sig = nuevo;
}
void suprime(Pila p)
{
      Pila eliminado = p->sig;
      p->sig = eliminado->sig;
      free(eliminado);
}
Elemento recupera(Pila p)
{
      return p->sig->elem;
}
int vacia(Pila p)
{
      return (p->sig==NULL);
}
```

3.7 Implementaciones del TDA Cola

3.7.1 Implementación del TDA Cola: Fichero de cabecera

Cola.h

```
#ifndef __COLA_H
#define __COLA_H

typedef int Elemento;
typedef struct ColaRep * Cola;

Cola crea();
void libera(Cola c);
void inserta(Cola c, Elemento e);
void suprime(Cola c);
Elemento recupera(Cola c);
int vacia(Cola c);
int llena(Cola c); // Sólo para Colas acotadas

#endif
```

3.7.2 Implementación del TDA Cola con representación contigua circular (1/2)

Cola.c

```
#include "Cola.h"
#include <stdlib.h>
#define MAX 100
struct ColaRep
{
     Elemento elem[MAX];
     int frente, posterior, n;
};
Cola crea()
{
     Cola c = malloc(sizeof(struct ColaRep));
     c->frente = 0;
     c->posterior = 0;
     c->n = 0;
     return c;
}
void libera(Cola c) { free(c); }
```

3.7.2 Implementación del TDA Cola con representación contigua circular (2/2)

Cola.c

```
void inserta(Cola c, Elemento e)
{
     c->elem[c->posterior] = e;
     c->posterior = (c->posterior+1)%MAX;
     c->n++;
}

void suprime(Cola c)
{
     c->frente = (c->frente+1)%MAX;
     c->n--;
}

Elemento recupera(Cola c) { return c->elem[c->frente]; }

int vacia(Cola c) { return (c->n==0); }

int llena(Cola c) { return (c->n==MAX); }
```

3.7.3 Implementación del TDA Cola con representación enlazada (1/3)

Cola.c

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

struct Celda
{
     Elemento elem;
     struct Celda * sig;
};

typedef struct Celda * Celda;

struct ColaRep
{
     Celda frente, posterior;
};
```

3.7.3 Implementación del TDA Cola con representación enlazada (2/3)

Cola.c

```
Cola crea()
{
     Cola c = malloc(sizeof(struct ColaRep));
     c->frente = malloc(sizeof(struct Celda));
     c->frente->sig = NULL;
     c->posterior = c->frente;
     return c;
}

void libera(Cola c)
{
     while(c->frente!=NULL)
     {
          Celda aux = c->frente;
          c->frente = aux->sig;
          free(aux);
     }
     free(c);
}
```

3.7.3 Implementación del TDA Cola con representación enlazada (3/3)

Cola.c

```
void inserta(Cola c, Elemento e)
{
     c->posterior->sig = malloc(sizeof(struct Celda));
     c->posterior = c->posterior->sig;
     c->posterior->elem = e;
     c->posterior->sig = NULL;
}

void suprime(Cola c)
{
     Celda eliminado = c->frente;
     c->frente = eliminado->sig;
     free(eliminado);
}

Elemento recupera(Cola c) { return c->frente->sig->elem; }

int vacia(Cola c) { return (c->frente->sig==NULL); }
```

3.8 Implementaciones del TDA Lista

3.8.1 Listas vs. Pilas y Colas

El TDA Lista difiere de los TDAs Pila y Cola, dado que en el caso de las listas se deberá permitir el **acceso a cualquier posición de la lista**.

En este curso, **por simplicidad y eficiencia**, se utilizarán **dos interfaces públicas diferentes**, una para listas con representación contigua, en donde las posiciones son de tipo `int`, y otra para listas con representaciones enlazadas, en donde las posiciones son apuntadores.

3.8.2 Implementación del TDA Lista con representación contigua (1/4)

Lista.h

```
#ifndef __LISTA_H
#define __LISTA_H
typedef struct ListaRep * Lista;
typedef int Posicion;
typedef int Elemento;
Lista crea();
void libera(Lista l);
void inserta(Lista l, Posicion p, Elemento e);
void suprime(Lista l, Posicion p);
Elemento recupera(Lista l, Posicion p);
void asigna(Lista l, Posicion p, Elemento e);
int longitud(Lista l);
Posicion inicio(Lista l);
Posicion fin(Lista l);
Posicion iesimo(Lista l, int i);
Posicion siguiente(Lista l, Posicion p);
Posicion anterior(Lista l, Posicion p);
int llena(Lista l); // Sólo para Listas acotadas
#endif
```

3.8.2 Implementación del TDA Lista con representación contigua (2/4)

Lista.c

```
#include "Lista.h"
#include <stdlib.h>
#define MAX 100
struct ListaRep
{
    Elemento elem[MAX];
    int n;
};
Lista crea()
{
    Lista l = malloc(sizeof(struct ListaRep));
    l->n = 0;
    return l;
}
void libera(Lista l) { free(l); }
```

3.8.2 Implementación del TDA Lista con representación contigua (3/4)

Lista.c

```
void inserta(Lista l, Posicion p, Elemento e)
{
     for (int i=l->n;i>p;i--) l->elem[i]=l->elem[i-1];
     l->elem[p]=e;
     l->n++;
}
void suprime(Lista l, Posicion p)
{
     l->n--;
     for(int i=p;i<l->n;i++) l->elem[i]=l->elem[i+1];
}
Elemento recupera(Lista l, Posicion p) { return l->elem[p]; }
void asigna(Lista l, Posicion p, Elemento e) { l->elem[p] = e; }
```

3.8.2 Implementación del TDA Lista con representación contigua (4/4)

Lista.c

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

Posicion inicio(Lista l) { return 0; }

Posicion fin(Lista l) { return l->n; }

Posicion iesimo(Lista l, int i) { return i-1; }

Posicion siguiente(Lista l, Posicion p) { return p+1; }

Posicion anterior(Lista l, Posicion p) { return p-1; }

int llena(Lista l) { return (l->n==MAX); }
```

3.8.3 Implementación del TDA Lista con representación enlazada – simple enlace (1/5)

Lista.h

```
#ifndef __LISTA_H
#define __LISTA_H
typedef struct ListaRep * Lista;
typedef struct PosicionRep * Posicion;
typedef int Elemento;
Lista crea();
void libera(Lista l);
void inserta(Lista l, Posicion p, Elemento e);
void suprime(Lista l, Posicion p);
Elemento recupera(Lista l, Posicion p);
void asigna(Lista l, Posicion p, Elemento e);
int longitud(Lista l);
Posicion inicio(Lista l);
Posicion fin(Lista l);
Posicion iesimo(Lista l, int i);
Posicion siguiente(Lista l, Posicion p);
Posicion anterior(Lista l, Posicion p);
#endif
```

3.8.3 Implementación del TDA Lista con representación enlazada – simple enlace (2/5)

Lista.c

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

struct ListaRep
{
     Posicion primera, ultima;
     int n;
};

struct PosicionRep
{
     Elemento elem;
     Posicion sig;
};
```

3.8.3 Implementación del TDA Lista con representación enlazada – simple enlace (3/5)

Lista.c

```
Lista crea()
{
     Lista l = malloc(sizeof(struct ListaRep));
     l->primera = malloc(sizeof(struct PosicionRep));
     l->primera->sig = NULL;
     l->ultima = l->primera;
     l->n = 0;
     return l;
}
void libera(Lista l)
{
     while(l->primera!=NULL)
     {
          Posicion aux = l->primera;
          l->primera = aux->sig;
          free(aux);
     }
     free(l);
}
```

3.8.3 Implementación del TDA Lista con representación enlazada – simple enlace (4/5)

Lista.c

```
void inserta(Lista l, Posicion p, Elemento e)
{
     Posicion nuevo = malloc(sizeof(struct PosicionRep));
     nuevo->elem = e;
     nuevo->sig = p->sig;
     p->sig = nuevo;
     if (l->ultima==p) l->ultima = p->sig;
     l->n++;
}
void suprime(Lista l, Posicion p)
{
     if (p->sig==l->ultima) l->ultima=p;
     Posicion eliminado = p->sig;
     p->sig = eliminado->sig;
     free(eliminado);
     l->n--;
}
```

3.8.3 Implementación del TDA Lista con representación enlazada – simple enlace (5/5)

Lista.c

```
Elemento recupera(Lista l, Posicion p) { return p->sig->elem; }
void asigna(Lista l, Posicion p, Elemento e) { p->sig->elem = e; }
int longitud(Lista l) { return l->n; }
Posicion inicio(Lista l) { return l->primera; }
Posicion fin(Lista l) { return l->ultima; }
Posicion iesimo(Lista l, int i) {
     Posicion aux = l->primera;
     for(int j = 1; j < i; j++) aux = aux->sig;
     return aux;
}
Posicion siguiente(Lista l, Posicion p) { return p->sig; }
Posicion anterior(Lista l, Posicion p)
{
     Posicion aux = l->primera;
     while(aux->sig!=p) aux = aux->sig;
     return aux;
}
```

3.8.4 Implementación del TDA Lista con representación enlazada – doble enlace (1/5)

Lista.h (nótese que el fichero de cabecera es el mismo que el de simple enlace)

```
#ifndef __LISTA_H
#define __LISTA_H
typedef struct ListaRep * Lista;
typedef struct PosicionRep * Posicion;
typedef int Elemento;
Lista crea();
void libera(Lista l);
void inserta(Lista l, Posicion p, Elemento e);
void suprime(Lista l, Posicion p);
Elemento recupera(Lista l, Posicion p);
void asigna(Lista l, Posicion p, Elemento e);
int longitud(Lista l);
Posicion inicio(Lista l);
Posicion fin(Lista l);
Posicion iesimo(Lista l, int i);
Posicion siguiente(Lista l, Posicion p);
Posicion anterior(Lista l, Posicion p);
#endif
```

3.8.4 Implementación del TDA Lista con representación enlazada – doble enlace (2/5)

Lista.c

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

struct ListaRep
{
     Posicion primera;
     int n;
};

struct PosicionRep
{
     Elemento elem;
     Posicion sig;
     Posicion ant;
};
```

3.8.4 Implementación del TDA Lista con representación enlazada – doble enlace (3/5)

Lista.c

```
Lista crea()
{
      Lista l = malloc(sizeof(struct ListaRep));
      l->primera = malloc(sizeof(struct PosicionRep));
      l->n = 0;
      l->primera->sig = l->primera;
      l->primera->ant = l->primera;
      return l;
}
void libera(Lista l)
{
      while(l->primera->sig!=l->primera)
      {
            Posicion aux = l->primera->sig;
            l->primera->sig = aux->sig;
            free(aux);
      }
      free(l->primera);
      free(l);
}
```

3.8.4 Implementación del TDA Lista con representación enlazada – doble enlace (4/5)

Lista.c

```
void inserta(Lista l, Posicion p, Elemento e)
{
     Posicion nuevo = malloc(sizeof(struct PosicionRep));
     nuevo->elem = e;
     nuevo->sig = p->sig;
     nuevo->ant = p;
     p->sig->ant = nuevo;
     p->sig = nuevo;
     l->n++;
}
void suprime(Lista l, Posicion p)
{
     Posicion eliminado = p->sig;
     p->sig = eliminado->sig;
     eliminado->sig->ant = p;
     free(eliminado);
     l->n--;
}
```

3.8.4 Implementación del TDA Lista con representación enlazada – doble enlace (5/5)

Lista.c

```
Elemento recupera(Lista l, Posicion p) { return p->sig->elem; }
void asigna(Lista l, Posicion p, Elemento e) { p->sig->elem = e; }
int longitud(Lista l) { return l->n; }
Posicion inicio(Lista l) { return l->primera; }
Posicion fin(Lista l) { return l->primera->ant; }
Posicion iesimo(Lista l, int i)
{
     Posicion aux = l->primera;
     for( int j = 1; j < i; j++) aux = aux->sig;
     return aux;
}
Posicion siguiente(Lista l, Posicion p) { return p->sig; }
Posicion anterior(Lista l, Posicion p){ return p->ant; }
```

3.9 Implementaciones del TDA Conjunto

3.9.1 Implementación del TDA Conjunto: Fichero de cabecera

Conjunto.h

```
#ifndef __CONJUNTO_H
#define __CONJUNTO_H
#include "Lista.h"

typedef struct ConjuntoRep * Conjunto;
typedef int Elemento;

Conjunto crea();
void libera(Conjunto c);
void inserta(Conjunto c, Elemento e);
void suprime(Conjunto c, Elemento e);
int pertenece(Conjunto c, Elemento e);
int cardinalidad(Conjunto c);
Lista enumera(Conjunto c);

#endif
```

3.9.2 Implementación del TDA Conjunto con representación enlazada – simple enlace (1/5)

Conjunto.c

```
#include "Conjunto.h"
#include <stdlib.h>
#include <stdio.h>
struct Nodo
{
     Elemento elem;
     struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;
struct ConjuntoRep
{
     NodoPtr primera;
     int n;
};
```

3.9.2 Implementación del TDA Conjunto con representación enlazada – simple enlace (2/5)

Conjunto.c

```
Conjunto crea()
{
     Conjunto c = malloc(sizeof(struct ConjuntoRep));
     c->primera = malloc(sizeof(struct Nodo));
     c->primera->sig = NULL;
     c->n = 0;
     return c;
}
void libera(Conjunto c) {
     while(c->primera!=NULL)
     {
          NodoPtr aux = c->primera;
          c->primera = aux->sig;
          free(aux);
     }
     free(c);
}
```

3.9.2 Implementación del TDA Conjunto con representación enlazada – simple enlace (3/5)

Conjunto.c (se mantienen los elementos ordenados por eficiencia)

```
void inserta(Conjunto c, Elemento e)
{
     NodoPtr aux = c->primera;
     while((aux->sig!=NULL)&&(aux->sig->elem<e))
          aux = aux->sig;
     if ( aux->sig == NULL || aux->sig->elem>e)
     {
          NodoPtr nuevo = malloc(sizeof(struct Nodo));
          nuevo->elem = e;
          nuevo->sig = aux->sig;
          aux->sig = nuevo;
          c->n++;
     }
}
```

3.9.2 Implementación del TDA Conjunto con representación enlazada – simple enlace (4/5)

Conjunto.c

```
void suprime(Conjunto c, Elemento e)
{
     NodoPtr aux = c->primera;
     while((aux->sig!=NULL)&&(aux->sig->elem < e))
          aux = aux->sig;
     if ( aux->sig!=NULL && aux->sig->elem == e)
     {
          NodoPtr eliminado = aux->sig;
          aux->sig = eliminado->sig;
          free(eliminado);
          c->n--;
     }
}
```

3.9.2 Implementación del TDA Conjunto con representación enlazada – simple enlace (5/5)

Conjunto.c

```
int pertenece(Conjunto c, Elemento e)
{
     NodoPtr aux = c->primera->sig;
     while((aux != NULL)&&(aux->elem < e))
          aux = aux->sig;
     return (aux != NULL && aux->elem == e);
}
Lista enumera(Conjunto c)
{
     Lista lista=ListaCrea();
     for(NodoPtr aux = c->primera->sig; aux != NULL; aux = aux->sig)
          ListaInserta(lista, ListaFin(lista), aux->elem);
     return lista;
}
int cardinalidad(Conjunto c)
{
     return c->n;
}
```

3.10 Comparación de las implementaciones (1/2)

Pila y Colas		
Tipo de representación	**Contigua**	**Enlazada**
Complejidad de las operaciones	*O(1)*	*O(1)*
Memoria (Pilas y Colas de `int` de 4 bytes)	*4*MAX*	*8*N*

Listas			
Tipo de representación	**Contigua**	**Simple enlace**	**Doble enlace**
Complejidad de inserta	*O(N)*	*O(1)*	*O(1)*
Complejidad de suprime	*O(N)*	*O(1)*	*O(1)*
Complejidad de anterior	*O(1)*	*O(N)*	*O(1)*
Complejidad de iesima	*O(1)*	*O(N)*	*O(N)*
Complejidad resto de operaciones	*O(1)*	*O(1)*	*O(1)*
Memoria (Listas de `int` de 4 bytes)	*4*MAX*	*8*N*	*12*N*

3.10 Comparación de las implementaciones (2/2)

Conjuntos		
Tipo de representación	**Simple enlace**	**Árbol Binario de Búsqueda (balanceado, no balanceado)**
Complejidad de las operaciones	*O(N)*	*O(log N), O(N)*
Memoria (Conjuntos de `int` de 4 bytes)	*8*N*	*12*N*

3.11 Algunas variantes de listas (1/2)

Listas, Colas y Pilas acotadas

Se dice que una Lista, Cola o Pila es acotada cuando su tamaño máximo queda limitado en su especificación. En estos casos, el conjunto de operaciones se amplía con una operación **llena** la cual devuelve "cierto" cuando el número de elementos almacenados es igual al máximo especificado.

En Listas, Colas y Pilas no acotadas con representaciones contiguas, el número máximo de elementos debe ser aumentado en el caso de que se requiera un número mayor de elementos. La función `realloc` puede utilizarse para este propósito.

Colas de Prioridad

Los elementos se atienden en orden a una prioridad asociada a cada uno. Si varios elementos tienen la misma prioridad se atenderán según el criterio FIFO.

3.11 Algunas variantes de listas (2/2)

Bicolas

Tipos especiales de Cola que permiten la inserción, acceso y supresión de elementos en ambos extremos de la Cola.

Listas circulares o anillos

Son tipos especiales de listas en donde el primer elemento es el sucesor del último y el último elemento es el predecesor del primero. No hay que confundir las listas circulares con las representaciones circulares de listas.

Listas ordenadas

Son tipos especiales de listas en donde los elementos de la lista siguen un orden determinado creciente o decreciente. Estas listas asumen, por tanto, que existe un orden definido en el tipo de sus elementos.

3.12 Aplicaciones y ejemplos de utilización (1/11)

Son múltiples las aplicaciones que utilizan Listas, Colas o Pilas en su diseño. A continuación se citan algunas de las más extendidas:

- Los compiladores utilizan una Pila para la gestión de las llamadas a las funciones y activación de bloques. La recursividad puede implementarse en los lenguajes de programación gracias al uso de Pilas.
- Los lenguajes de programación usan Pilas para la evaluación de las expresiones.
- Se utilizan Pilas en el reconocimiento sintáctico en los lenguajes de programación.
- Algunos algoritmos de búsqueda de valores óptimos (fuerza bruta, ramificación y poda, etc.) usan Colas.
- La planificación del uso de los recursos de la computadora en los sistemas operativos utilizan Colas.
- Un consultorio médico un sistema de Colas para asignar turno en las distintas especialidades del centro. Este sistema de turnos por Colas es utilizado en muchas otras instituciones públicas tales como la seguridad social.
- La agenda de contactos y el registro de llamadas realizadas de un dispositivo móvil, los buzones de entrada y salida de correos electrónicos, una ráfaga de balas en un videojuego, el texto editado en un procesador de texto, la lista de sitios favoritos de internet en el navegador, etc.

A continuación se muestran algunos ejemplos de problemas concretos que usan Pilas, Colas o Listas.

3.12 Aplicaciones y ejemplos de utilización (2/11)

Función que imprime el contenido de una Pila

```
#include "Pila.h"
#include <stdio.h>
void imprime(Pila p)
{
     Pila p1 = crea();
     while(!vacia(p))
     {
          Elemento e = recupera(p);
          printf("%d ",e);
          inserta(p1,e);
          suprime(p);
     }
     while(!vacia(p1))
     {
          inserta(p,recupera(p1));
          suprime(p1);
     }
     libera(p1);
}
```

3.12 Aplicaciones y ejemplos de utilización (3/11)

Función que comprueba si las llaves, corchetes y paréntesis están correctamente balanceados en una cadena de caracteres.

```
#include "Pila.h"
int balanceo(char * s)
{
      int bal = 1;
      Pila p = crea();
      for ( int i=0 ; (s[i]!=0) && bal ; i++ )
      {
            switch(s[i])
            {
                  case '(': inserta(p,1); break;
                  case '[': inserta(p,2); break;
                  case '{': inserta(p,3); break;
                  case ')': if (vacia(p)||(recupera(p)!=1)) bal=0; else suprime(p);break;
                  case ']': if (vacia(p)||(recupera(p)!=2)) bal=0; else suprime(p);break;
                  case '}': if (vacia(p)||(recupera(p)!=3)) bal=0; else suprime(p);break;
            }
      }
      int completo = vacia(p);
      libera(p);
      return (bal && completo);
}
```

3.12 Aplicaciones y ejemplos de utilización (4/11)

Función que imprime el contenido de una Cola

```
#include "Cola.h"
#include <stdio.h>
void imprime(Cola c)
{
     Cola c1 = crea();
     while(!vacia(c))
     {
          Elemento e = recupera(c);
          printf("%d ",e);
          inserta(c1,e);
          suprime(c);
     }
     while(!vacia(c1))
     {
          inserta(c,recupera(c1));
          suprime(c1);
     }
     libera(c1);
}
```

3.12 Aplicaciones y ejemplos de utilización (5/11)

Funciones que imprimen una Lista de principio a fin y de fin a principio

```
#include "Lista.h"
#include <stdio.h>
void imprime(Lista l)
{
     for(Posicion p=inicio(l); p!=fin(l); p=siguiente(l,p))
          printf("%d ",recupera(l,p));
}

void imprimeInverso(Lista l)
{
     Posicion p=fin(l);
     while(p!=inicio(l))
     {
          p=anterior(l,p);
          printf("%d ",recupera(l,p));
     }
}
```

3.12 Aplicaciones y ejemplos de utilización (6/11)

Implementación del TDA Conjunto mediante una Lista – Conjunto.c

```
#include "Conjunto.h"
#include "Lista.h"
#include <stdlib.h>
#include <stdio.h>
#define STRING_LONG 100

struct ConjuntoRep {
    Lista l;
};
Conjunto ConjuntoCrea() {
    Conjunto c = malloc(sizeof(struct ConjuntoRep));
    c->l = ListaCrea();
    return c;
}
void ConjuntoLibera(Conjunto c) {
    ListaLibera(c->l);
    free(c);
}
```

3.12 Aplicaciones y ejemplos de utilización (7/11)

Implementación del TDA Conjunto mediante una Lista – Conjunto.c

```
void ConjuntoInserta(Conjunto c, Elemento e)
{
    if (!ConjuntoPertenece(c,e))
       ListaInserta(c->l,ListaInicio(c->l),e);
}

int ConjuntoPertenece(Conjunto c, Elemento e)
{
    Posicion p = ListaInicio(c->l);
    while((p!=ListaFin(c->l))&&(ListaRecupera(c->l,p)!=e))
       p=ListaSiguiente(c->l,p);
    return (p!=ListaFin(c->l));
}

int ConjuntoCardinalidad(Conjunto c)
{
     return ListaLongitud(c->l);
}
```

3.12 Aplicaciones y ejemplos de utilización (8/11)

```
void ConjuntoSuprime(Conjunto c, Elemento e)
{
      Posicion p = ListaInicio(c->l);
      while((p!=ListaFin(c->l))&&(ListaRecupera(c->l,p)!=e))
            p=ListaSiguiente(c->l,p);
      if (p!=ListaFin(c->l))
            ListaSuprime(c->l,p);
}

char * ConjuntoToString(char * s, Conjunto c)
{
    char aux[STRING_LONG];
    for(Posicion p=ListaInicio(c->l);p!=ListaFin(c->l);p=ListaSiguiente(c->l,p))
    {
        sprintf(aux,"%d ",ListaRecupera(c->l,p));
        strcat(s,aux);
    }
    return s;
}
```

3.12 Aplicaciones y ejemplos de utilización (9/11)

Operaciones de uso del TDA Conjunto

```
void Conjunto_imprime(Conjunto c) {
    Lista datos = Conjunto_enumera(c);
    Posicion p = Lista_inicio(datos);
    while (p != Lista_fin(datos))
    {
        int dato = Lista_recupera(datos,p);
        printf("%d ",dato);
        p = Lista_siguiente(datos,p);
    }
    printf("\n");
    Lista_libera(datos);
}
```

3.12 Aplicaciones y ejemplos de utilización (10/11)

Operaciones de uso del TDA Conjunto

```
Conjunto Conjunto_union(Conjunto a, Conjunto b) {
      Conjunto u = Conjunto_crea();
      Lista datos = Conjunto_enumera(a);
      Posicion p = Lista_inicio(datos);
      while (p != Lista_fin(datos)) {
            int dato = Lista_recupera(datos,p);
            Conjunto_inserta(u,dato);
            p = Lista_siguiente(datos,p);
      }
      Lista_libera(datos);
      datos = Conjunto_enumera(b);
      p = Lista_inicio(datos);
      while (p != Lista_fin(datos)) {
            int dato = Lista_recupera(datos,p);
            Conjunto_inserta(u,dato);
            p = Lista_siguiente(datos,p);
      }
      Lista_libera(datos);
      return u;
}
```

3.12 Aplicaciones y ejemplos de utilización (11/11)

Operaciones de uso del TDA Conjunto

```
Conjunto Conjunto_interseccion(Conjunto a, Conjunto b) {
    Conjunto i = Conjunto_crea();
    Lista datos = Conjunto_enumera(a);
    Posicion p = Lista_inicio(datos);
    while (p != Lista_fin(datos)) {
        int dato = Lista_recupera(datos, p);
        if (Conjunto_pertenece(b, dato))
            Conjunto_inserta(i, dato);
        p = Lista_siguiente(datos, p);
    }
    Lista_libera(datos);
    return i;
}
```

3.13 Ejercicios resueltos (1/7)

Objetivos:

- Implementar variantes de TDAs fundamentales.
- Usar TDAs fundamentales en la implementación de aplicaciones.

1) Implementar en *C* el TDA Cola según el fichero de cabecera `Cola.h` incluido a continuación. Debe utilizarse una estructura enlazada para su implementación. Todas las operaciones deben implementarse con un tiempo *O(1)* excepto `ColaLibera` que será *O(n)*.

   Escribir una aplicación en la que se representen las cajas de un supermercado mediante un array de colas. Para ello, realizar un módulo `main.c` que use el TDA Cola e implemente las funciones `main` e `inserta` que realicen los siguientes pasos:

3.13 Ejercicios resueltos (2/7)

```
#include "Cola.h"

// Inserta un cliente con identificador id en la cola que tenga menos clientes del
// array colas. El número de colas del array es n.
// Si hay varias colas con el mismo número de clientes, inserta en la cola que se
// encuentre en el menor índice del array.
void inserta(Cola colas[], int n, int id)
{
...
}

int main(int argc, char * argv[])
{
// Declara un array con capacidad para 10 colas y asigna a cada elemento del
// array una cola vacía.
...
// Usa la función inserta para insertar 100 clientes con identificadores del
// 1 al 100 en el array de colas anteriormente declarado.
...
// Libera las colas del array.
...
return 0;
}
```

3.13 Ejercicios resueltos (3/7)

```
#ifndef __COLA_H
#define __COLA_H
typedef struct ColaRep * Cola;
// Devuelve una cola vacía.
Cola ColaCrea();
// Libera la cola c.
void ColaLibera(Cola c);
// Inserta en la cola c un cliente con el identificador asignado.
void ColaInserta(Cola c, int id);
// Suprime de la cola c el cliente que llegó antes.
// Devuelve el identificador del cliente eliminado.
// El número de clientes de la cola c debe ser mayor que 0.
int ColaSuprime(Cola c);
// Devuelve el número de clientes en la cola c.
int ColaN(Cola c);
#endif // __COLA_H
```

3.13 Ejercicios resueltos (4/7)

```
#include <stdlib.h>
#include "Cola.h"
typedef struct NodoRep * Nodo;
struct NodoRep
{
     int id;
     Nodo sig;
};
struct ColaRep
{
     Nodo frente, posterior;
     int n;
};
Cola ColaCrea()
{
     Cola c = malloc(sizeof(struct ColaRep));
     c->frente = malloc(sizeof(struct NodoRep));
     c->frente->sig = NULL;
     c->posterior = c->frente;
     c->n = 0;
     return c;
}
```

3.13 Ejercicios resueltos (5/7)

```
void ColaLibera(Cola c)
{
     while (c->frente!=NULL)
     {
          Nodo aux = c->frente;
          c->frente = aux->sig;
          free(aux);
     }
     free(c);
}
void ColaInserta(Cola c, int id)
{
     c->posterior->sig = malloc(sizeof(struct NodoRep));
     c->posterior = c->posterior->sig;
     c->posterior->id = id;
     c->posterior->sig = NULL;
     c->n++;
}
```

3.13 Ejercicios resueltos (6/7)

```
int ColaSuprime(Cola c)
{
     int resultado = c->frente->sig->id;
     Nodo eliminado = c->frente;
     c->frente = eliminado->sig;
     free(eliminado);
     c->n--;
     return resultado;
}
int ColaN(Cola c)
{
     return c->n;
}
```

3.13 Ejercicios resueltos (7/7)

```
#include "Cola.h"
void inserta(Cola colas[], int n, int id)
{
     int imin = 0, nmin = ColaN(colas[0]);
     for(int i=1; i<n; i++)
     {
          int ni = ColaN(colas[i]);
          if (ni<nmin) { nmin = ni; imin = i; }
     }
     ColaInserta(colas[imin],id);
}
int main()
{
     int n = 10;
     Cola colas[n];
     for(int i=0; i<n; i++) colas[i] = ColaCrea();
     for(int id=1; id<=100; id++) inserta(colas,n,id);
     for(int i=0; i<n; i++) ColaLibera(colas[i]);
     return 0;
}
```

3.14 Ejercicios propuestos (1/2)

Objetivos:

- Implementar variantes de TDAs fundamentales.
- Usar TDAs fundamentales en la implementación de aplicaciones.

1) Considérense el TDA Lista con representación enlazada – simple enlace con el fichero de cabecera `Lista.h` definido en la sección 3.8.3, y el TDA Pila con el fichero de cabecera `Pila.h` definido en en la sección 3.6.1. Implementar, en un nuevo módulo `misc.c`, la función `imprime_inverso`, la cual imprime en pantalla una lista `l` en un tiempo de ejecución *O(n)*, usando para ello una pila.

```
#include "Lista.h"
#include "Pila.h"
void imprime_inverso(Lista l)
{
     Pila p;
     ...
}
```

3.14 Ejercicios propuestos (2/2)

2) Considérense el TDA Lista con representación enlazada – simple enlace con el fichero de cabecera `Lista.h` definido en la sección 3.8.3, y el fichero de cabecera `Pila.h` definido en la sección 3.6.1. Implementar, en un nuevo módulo `Pila.c`, el TDA Pila (no acotada) sujeto al fichero de cabecera `Pila.h`, usando el TDA Lista para representa una pila.

3) Considérense el TDA Lista con representación enlazada – simple enlace con el fichero de cabecera `Lista.h` definido en la sección 3.8.3, y el fichero de cabecera `Cola.h` definido en la sección 3.7.1. Implementar, en un nuevo módulo `Cola.c`, el TDA Cola (no acotada) sujeto al fichero de cabecera `Cola.h`, usando el TDA Lista para representa una cola.