Programación II

TAD Lista

Especificación informal TAD Lista

TAD Lista
VALORES

- Una lista es una secuencia de cero o más elementos de un mismo tipo $(a_1, a_2, a_3, ..., a_n)$ donde n≥0:
  - Si n=0 se dice que la lista es vacía.
  - Los elementos de la lista están ordenados de forma lineal, no por su contenido, sino por la posición que ocupan. Es decir:
    - a_i precede a a_{i+1} (∀ i, i=1...n-1)
    - a_i sucede a a_{i-1} (∀ i, i=2...n)

OPERACIONES (SINTAXIS y SEMÁNTICA)

- Generadoras
  - `createEmptyList` → `List`
    { *Objetivo*: Crea una lista vacía y la inicializa
    Salida: Una lista vacía
    Poscondición: La lista sin datos }
  - `insertItem (Item, Position, List)` → `List, Boolean`
    {*Objetivo*: Si la posición es nula, añade un elemento al final de la lista. En caso contrario, el elemento quedará insertado justo antes del que actualmente ocupa la posición indicada.
    Entrada:

    Item: Contenido del elemento a insertar
    Position: Posición de referencia para la inserción
    List: Lista donde vamos a insertar

    Salida:

    List: Lista con el elemento Item insertado y
    verdadero si se ha podido insertar, falso en caso contrario

    Precondición:

    Position es una posición válida de la lista o es una posición nula

    Postcondición:

    Las posiciones de los elementos de la lista posteriores a la del elemento insertado pueden haber variado}

- Modificadoras

  - `copyList(List`$_1$`)` $\rightarrow$ `List`$_2$`, Boolean`
    {*Objetivo*: Copia una lista en otra
    Entrada:

    List$_1$: Lista que vamos a copiar

    Salida:

    List$_2$: Copia de la lista original y
    verdadero si se ha podido copiar, falso en caso contrario

    Precondición:

    La lista origen está inicializada }

  - `updateItem (Item, Position, List)` $\rightarrow$ `List`
    {*Objetivo*: Modifica el contenido de un elemento de la lista
    Entrada:

    Item: Nuevo contenido a asignar al elemento en Position
    Position: Posición del elemento que queremos modificar
    List: Lista a modificar

    Salida:

    List: Lista con el contenido del elemento modificado

    Precondición:

    Position es una posición válida de la lista }

- Destructoras

  - `deleteAtPosition (Position, List)` $\rightarrow$ `List`
    {*Objetivo*: Elimina de la lista un elemento con cierta posición
    Entrada:

    Position: Posición del elemento a eliminar
    List: Lista a modificar

    Salida:

    List: Lista sin el elemento correspondiente a Position

    Precondición:

    Position es una posición válida de la lista

    Postcondición:

    Las posiciones de los elementos de la lista posteriores a la de la posición eliminada pueden haber variado }

  - `deleteList (List)` $\rightarrow$ `List`
    {*Objetivo*: Elimina todos los elementos de la lista
    Entrada:

    List: Lista a borrar

    Salida: Lista vacía }

- Observadoras

  - `findItem (Item, List)` → `Position`
    {*Objetivo*: Busca el primer elemento con cierto contenido en la lista
    Entrada:

    Item: Contenido del elemento buscado
    List: Lista donde realizar la búsqueda

    Salida:

    Position: Posición del elemento encontrado o nulo si no se encuentra }

  - `isEmptyList (List)` → `Boolean`
    { *Objetivo*: Determina si la lista está vacía
    Entrada:

    List: Lista a comprobar

    Salida:

    Verdadero si la lista está vacía, falso en caso contrario }

  - `getItem (Position, List)` → `Item`
    {*Objetivo*: Recupera el contenido de un elemento de la lista
    Entrada:

    Position: Posición del elemento buscado
    List: Lista donde realizar la búsqueda

    Salida:

    Item: Contenido del elemento que está en Position

    Precondición:

    Position es una posición válida en la lista }

- `first (List)` → `Position`
  {*Objetivo*: Devuelve la posición del primer elemento de la lista
  Entrada:

  List: Lista a manipular

  Salida:

  Position: Posición del primer elemento

  Precondición: La lista no está vacía }

- `last(List)` → `Position`
  {*Objetivo*: Devuelve la posición del último elemento de la lista
  Entrada:

  List: Lista a manipular

  Salida:

  Position: Posición del último elemento

  Precondición: La lista no está vacía }

- `previous(Position, List)` → `Position`
  { *Objetivo*: Devuelve la posición del elemento anterior al actual
  Entrada:

  Position: Posición del elemento actual
  List: Lista a manipular

  Salida:

  Posición del elemento anterior o nulo si es el primero

  Precondiciónn: *Position* es una posición válida de la lista }

- `next (Position, List)` → `Position`
  {*Objetivo*: Devuelve la posición del elemento siguiente al actual
  Entrada:

  Position: Posición del elemento actual
  List: Lista a manipular

  Salida:

  Position: Posición del elemento siguiente o nulo si es el último

  Precondición: *Position* es una posición válida de la lista }

Programación II

TAD Lista Ordenada

Especificación informal TAD Lista Ordenada

TAD Lista Ordenada
VALORES

- Una lista ordenada es una secuencia de cero o más elementos de un mismo tipo $(a_1, a_2, a_3, ..., a_n)$ donde n≥0:
  - Si n=0 se dice que la lista es vacía.
  - Los elementos de la lista están ordenados de forma lineal por su contenido.

OPERACIONES (SINTAXIS y SEMÁNTICA)[1]

- Generadoras
  - `insertItem (Item, List)` → `List, Boolean`
    {*Objetivo*: Inserta un elemento en la lista según el criterio de ordenación sobre el campo Item
    Entrada:

    Item: Contenido del elemento a insertar
    List: Lista donde vamos a insertar

    Salida:

    List: Lista con el elemento Item insertado en la posición correspondiente según su contenido
    y verdadero si se ha podido insertar, falso en caso contrario

    Precondición:

    La lista está inicializada

    Postcondición:

    Las posiciones de los elementos de la lista posteriores a la del elemento insertado pueden haber variado}

[1] Sólo se incluyen las modificaciones respecto al TAD Lista.

Programación II

Comparación de las implementaciones de listas

Ventajas e inconvenientes de las implementaciones estática y dinámica del TAD Lista.

	Estática	Simple Enlace	Doble Enlace
Necesidad de memoria[1]	mucha	menos en promedio (más que simple enlace)	
Memoria contigua	si	no	
Acceso directo[2]	si	no	
Ampliable	no	si	
Operaciones más costosas[3]	insertItem deleteAtPosition (excepto al final)	insertItem (final) deleteAtPosition (final) previous, last deleteList, copyList	insertItem (final) last deleteList copyList
Seguridad[4]	Total	Relajada	

[1] En la realización con estructuras estáticas tenemos espacio de memoria fijo, por lo que, en general, utilizamos más memoria de la que necesitamos. Por otro lado, en las estructuras dinámicas cada nodo ocupará un espacio de memoria adicional correspondiente al puntero (a igualdad de elementos ocuparán más).

[2] Con la estática tenemos acceso directo a los elementos de la lista.

[3] En general, en las estructuras estáticas son más sencillas las operaciones que requieren acceder a los elementos de la lista por su posición. Por el contrario, hemos de realizar desplazamiento de los elementos cuando queramos insertar o eliminar uno de ellos.
Las operaciones no mencionadas en este apartado tienen igual complejidad en todas las implementaciones (ej. `findItem`).

[4] Cuando manejamos arrays, para alterar la lista hay que pasar la variable de tipo `tList` por referencia. Sin embargo esto no ocurre así con los punteros, ya que hemos visto que pasar un puntero como parámetro equivale a pasar por referencia la variable a la que apunta (véase, p.e., la implementación de `updateItem`).

La implementación concreta que elijamos dependerá de las características que deba tener la lista para nuestra aplicación y de las operaciones que vayamos a realizar. P.e., si el tamaño es más o menos conocido, si va a tener pocos cambios una vez establecida y se suelen hacer muchos accesos por posición, lo más apropiado, probablemente sea una implementación basada en arrays. Por otro lado, si la lista va a ser muy dinámica en cuanto a su tamaño o con frecuentes inserciones y eliminaciones de elementos, lo más apropiado será una implementación basada en variables dinámicas.

Programación II

Archivo de cabecera TAD Lista

Archivo de cabecera TAD Lista

```
#include <stdbool.h>

#define LNULL ...; //constante que representa posiciones nulas

typedef ... tItemL; //se define en funcion del problema
typedef ... tPosL;
typedef ... tList;

void createEmptyList(tList* L);
bool insertItem(tItemL d, tPosL p, tList* L);
bool copyList(tList L, tList* M);

void updateItem(tItemL d , tPosL p, tList* L);

void deleteAtPosition(tPosL p, tList* L);
void deleteList(tList* L);

tPosL findItem(tItemL d, tList L);
bool isEmptyList(tList L);
tItemL getItem(tPosL p, tList L);
tPosL first(tList L) ;
tPosL last(tList L);
tPosL previous(tPosL p, tList L);
tPosL next(tPosL p, tList L);
```