Práctica Obligatoria Tipos Abstractos de Datos

Enunciados de los ejercicios a entregar

Esta entrega consiste en la realización de los ejercicios que se enuncian a continuación. Se abrirá una tarea en Studium para que, una vez finalizada su realización, el alumno suba un único fichero `.zip` o `.tgz` que contenga todos los ficheros con el código fuente que se solicitan. Esta tarea en Studium permanecerá abierta hasta el jueves, **11 de enero de 2024, a las 14:00**. Después de esa hora ya no se admitirá ninguna entrega.
A continuación se describe la tarea a implementar.

Enunciado 1

Utilización del TAD **Lista**, con el objetivo de comprobar que un TAD, una vez implementado, puede utilizarse de manera análoga a un tipo de datos primitivo. Como se ha explicado en clase, el diseño de algoritmos se debe hacer en función de la interfaz que proporciona el tipo abstracto que se vaya a utilizar. Se propone implementar una nueva versión del algoritmo de ordenación por *inserción directa*, de manera que utilice el TAD **Lista**, en sustitución de un vector para almacenar los valores a ordenar. La definición de la interfaz del TAD **Lista** se puede encontrar en el **Anexo** de este mismo enunciado. De forma más detallada, para completar este ejercicio de forma correcta se debe:

Diseñar e implementar una nueva versión del algoritmo de ordenación inserción directa, visto en el **Tema 4**, de forma que utilice el TAD **Lista**. Se utilizarán, única y exclusivamente, las funciones de la interfaz del TAD **Lista**. A la hora de implementar esta nueva versión del algoritmo se deben tener en cuenta algunos aspecto con respecto a la eficiencia:

- En la versión del algoritmo explicada en clase se utiliza un vector, una estructura de acceso directo, mientras que en esta nueva versión se propone utilizar una **Lista**, que solo permite acceso secuencial. Esto implica que ciertos recorridos penalizan la eficiencia de forma importante. Esto deberá ser tenido en cuenta a la hora de realizar de la búsqueda de la posición en la **sublista** ordenada para realizar la inserción.
- Los cambios en los recorridos pueden hacer que los comportamientos, en cuanto a eficiencia, de esta nueva versión cambien de forma significativa, pudiendo intercambiarse los casos peor y mejor con respecto a los que presenta la versión explicada en clase. De hecho, con alguna sencilla modificación, es posible conseguir que la nueva versión no tenga caso peor, solo caso mejor (lista ordenada ascendentemente o descendentemente) y caso medio.
- Se debe realizar un análisis teórico de la nueva versión del algoritmo y comprobar su validez con los resultados obtenidos en las pruebas.

Para realizar la tarea descrita se deben **tener implementadas las funciones del TAD Lista** siguiendo la definición e interfaz propuestas en la segunda sesión de prácticas correspondiente al **Tema 6**, `TAD Lista utilizando memoria dispersa (punteros)` y que de forma resumida se especifica en el **Anexo** de este mismo enunciado. Con el objetivo de facilitar y mejorar la eficiencia, se debe añadir tres funciones nuevas a las ya implementadas en la sesión de prácticas referida. Estas funciones son:

- `tipoElemento recuperaUltimo(Lista *l)`

  Devuelve el último valor de la lista que recibe como argumento. Si `l` es la colección de elementos $a_1, a_2, \ldots, a_{n-1}, a_n$, el resultado de ejecución de esta función es la devolución del valor a_n.

  La función devolverá un código negativo en caso de error.

- `int dividirLista(Lista *lOrigen, tipoPosicion p, Lista *lNueva)`

  Divide en dos listas la lista recibida como primer argumento, dejando como último elemento de dicha lista, el indicado por la posición `p`, segundo argumento de la función. Por lo tanto, el tercer argumento

recibido será una lista vacía y, una vez ejecutada la función, contendrá el resto de elementos. Si `lOrigen` es la colección de elementos $a_1, a_2, \ldots, a_{p-1}, a_p, a_{p+1}, \ldots, a_n$, el resultado de la ejecución de esta función es el siguiente:

- La lista `lOrigen` debe contener los elementos $a_1, a_2, \ldots, a_{p-1}, a_p$.
- La lista `lNueva` debe contener los elementos $a_{p+1}, a_{p+2}, \ldots, a_{n-1}, a_n$.

La función devolverá un cero en caso de éxito, y un entero negativo en caso de error.

- `int traspasarNodo(tipoPosicion p, Lista *la, tipoPosicion q, Lista *lb)`

  Operación que traspasa el nodo de la posición `p`, indicada en el primer parámetro, de la lista referenciada por el parámetro `la`, a la posición `q`, indicada en el tercer parámetro, de la lista referenciada por el parámetro `lb`. Si la es la colección de elementos $a_1, a_2, a_{p-1}, a_p, a_{p+1}, \ldots, a_n$ y `lb` es la colección $b_1, b_2, b_{q-1}, b_q, b_{q+1}, \ldots., b_n$, el resultado de la ejecución de esta función será:

  - La lista `la` debe quedar con los elementos $a_1, a_2, a_{p-1}, a_{p+1}, \ldots, a_n$
  - La lista `lb` debe quedar con los elementos $b_1, b_2, b_{q-1}, a_p, b_q, b_{q+1}, \ldots., b_n$. Si q es `fin(lb)` entonces a_p se añade al final.

  La función devolverá un cero en caso de éxito, y un entero negativo en caso de error.

La primera función será útil en las modificaciones que se proponen para evitar que el caso mejor se convierta en un caso peor. La segunda función facilita la tarea de dividir la lista inicial en dos, una con un único elemento, sublista ordenada, y otra con el resto de elementos, sublista desordenada. La tercera función se utilizará para traspasar nodos de la sublista desordenada a la ordenada en construcción, una vez se ha localizado la posición que le corresponde.

Se proporciona un programa de prueba, `pruebaInsercion.c`, que el estudiante utilizará para comprobar el correcto funcionamiento del algoritmo a implementar.

Finalmente, una vez implementado y probado el algoritmo se debe hacer un **estudio experimental** del comportamiento del mismo y aportar un fichero de resultados, semejante a los obtenidos en las sesiones de prácticas, que muestre resultados para diferentes tamaños de la lista que debe ordenar.

Enunciado 2

TAD PILA

Tipo especial de lista con la restricción de que las inserciones y eliminaciones solo pueden realizarse en una posición, denominada tope o cima de la pila. Se trata de una lista de tipo LIFO (*Last In, First Out*), en la que existe un elemento (siempre que no este vacía) en la cima de la pila que es el único visible o accesible. A continuación se definen las operaciones básicas que típicamente incluye un TAD pila:

creaVacía(p): inicia o crea la pila `p` como una pila vacía, sin ningún elemento

vacía(p): devuelve verdadero si la pila `p` está vacía, y falso en caso contrario

inserta(x,p): añade el elemento `x` a la pila `p` convirtiéndolo en el nuevo tope o cima de la pila

suprime(p): devuelve y suprime el elemento del tope o cima de la pila

Realizar la implementación del TAD Pila descrito previamente utilizando listas simplemente enlazadas **sin** nodo ficticio, según muestra la siguiente Figura 1

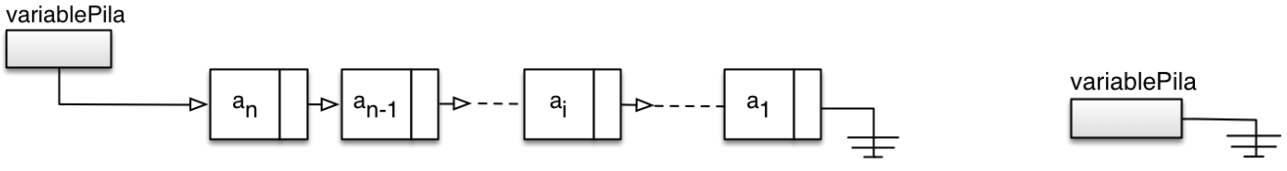

Figura 1: Pila implementada mediante lista símplemente enlazada sin nodo ficticio.

Como se puede apreciar en Figura 1, la variable `variablePila` es un puntero de tipo `tipoCelda` que apunta al nodo que contiene el último elemento (tope o cima) insertado en la pila. Una pila vacía es dicha variable a `NULL` (ver Figura 1).
La implementación debe hacerse ajustándose a los tipos y prototipos que se adjuntan en el fichero cabecera `pila.h` que se encuentra en la carpeta `pilas` que se entrega junto con este enunciado.

Enunciado 3

TAD COLA

Tipo especial de lista con la restricción de que las inserciones se realizan por un extremo de la lista denominado fondo y las eliminaciones por el otro extremo, denominado frente. Se trata de una lista de tipo FIFO (*First In, First Out*), en las que el primer elemento que entra es el primero en salir. A continuación se definen las operaciones básicas que típicamente incluye un TAD cola:

creaVacía(c): inicia o crea la cola `c` como una cola vacía, sin ningún elemento.

vacía(c): devuelve verdadero si la cola `c` está vacía, y falso en caso contrario

inserta(x, c): añade el elemento x a la cola c convirtiéndolo en el último elemento de la cola. Se añade por el fondo o final de la cola.

suprime(c): devuelve y suprime el primer elemento de la cola, de forma que el siguiente elemento pasa a ser el nuevo frente

Realizar la implementación del TAD Cola descrito previamente utilizando listas simplemente enlazadas **sin** nodo ficticio, según muestra la siguiente Figura 2

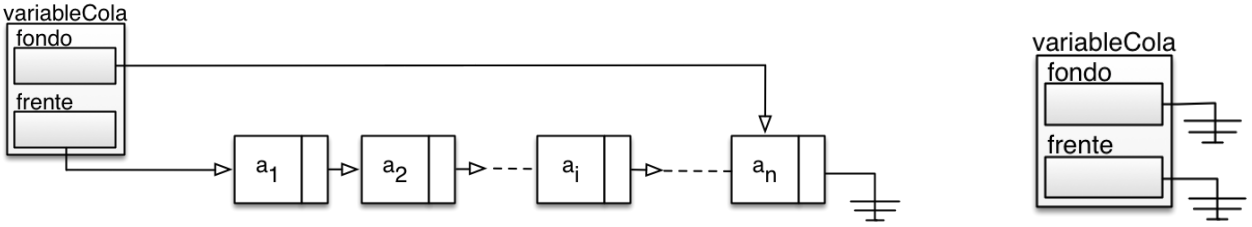

Figura 2: Cola implementada mediante lista símplemente enlazada sin nodo ficticio.

Como se puede apreciar en Figura 2, la variable `variableCola` es una estructura con dos punteros de tipo `tipoCelda`:

- el puntero `frente` que apunta siempre al primer nodo de la lista enlazada, por donde se extraen los elementos.
- el puntero `fondo` que apunta siempre al último nodo de la lista enlazada, por donde se añaden los nuevos elementos.

En el caso de estar vacía la cola, ambos punteros estarán a `NULL` (ver Figura 2). La implementación debe hacerse ajustándose a los tipos y prototipos que se adjuntan en el fichero cabecera `cola.h` que se encuentra en la carpeta `colas` que se entrega junto con este enunciado.

Consideraciones generales sobre la implementación

El alumno es responsable de crear programas con los que poder probar la correcta implementación de las funciones que se piden. Para ello se puede apoyar en los ejercicios propuestos durante las sesiones de prácticas de este tema.

La gestión de errores es responsabilidad del estudiante. **Importante**: Las funciones a implementar no mostrarán ningún mensaje por pantalla, en ningún caso.

El fichero `entregaTAD2023.zip` que se entrega contiene la siguiente estructura de carpetas:

- `entregaTAD2023.pdf`, el presente documento de enunciado en formato PDF
- `listas`, carpeta que contiene el ficheros de cabecera `lista.h`, con las definiciones de tipos y prototipos de funciones. El estudiante añadirá el fichero `lista.c` con la implementación de las funciones realizada en la sesión de prácticas ya indicada.
- `pilas`, carpeta que contiene el ficheros de cabecera `pila.h`, con las definiciones de tipos y prototipos de funciones, y el fichero `pila.c` con el código que debe completar el estudiante.
- `colas`, carpeta que contiene el ficheros de cabecera `cola.h`, con las definiciones de tipos y prototipos de funciones, y el fichero `cola.c` con el código que debe completar el estudiante.

Una vez implementadas todas las funciones y probadas, se debe crear un nuevo fichero `entregaTAD2023.zip` que contendrá:

- las implementaciones de las funciones de `lista.c`, `pila.c` y `cola.c` en sus respectivas carpetas, manteniendo la misma estructura descrita
- los ficheros `insercion.c` y `insercion.h`, conteniendo la implementación del algoritmo,
- el fichero `pruebaInsercion.c` modificado para la obtención los resultados de ejecución solicitados,
- los ficheros con los resultados citados y su interpretación con respecto al análisis teórico del algoritmo de inserción directa implementado. Es importante tener en cuenta que la modificación de la estructura del algoritmo explicado en teoría al utlizar el TAD propuesto puede implicar variaciones en el análisis teórico.

Este es el fichero que se debe subir a Studium dentro del plazo establecido: 14:00 horas del jueves, 11 de enero de 2024.

Anexo: Definición TAD LISTA

Se define el TAD lista como una secuencia ordenada de cero o más elementos de un determinado tipo de dato $a_1, a_2, a_3, \ldots, a_n$. El orden de los elementos está determinado por su posición en la secuencia, el elemento a_i está en la posición i.

Se debe implementar en C el TAD **lista**, utilizando para ello listas simplemente enlazadas y con nodo ficticio. Para mejorar la eficiencia de algunas funciones se define el tipo de dato, que llamaremos `Lista`, utilizando un registro (`struct`) con dos campos, siendo el primer campo un puntero al nodo ficticio (campo `raiz` del registro) y el segundo campo un puntero que apuntará al último nodo de la lista enlazada (campo `ultimo` del registro). En la Figura 3 se puede observar el registro y la lista enlazada que representa el tipo `Lista` conteniendo valores y una lista vacía, lo que implica que la lista enlazada sólo contiene el nodo ficticio, con los dos punteros apuntando a dicho nodo ficticio, y el campo `sig` de dicho nodo ficticio puesto a `NULL`.

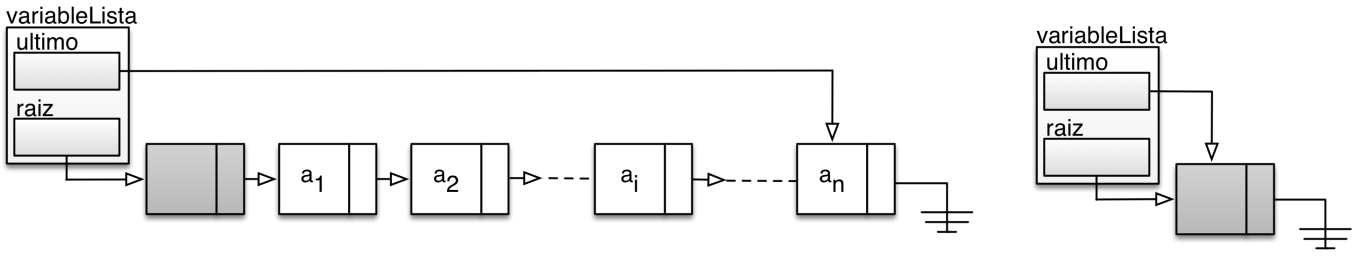

Figura 3: Lista implementada mediante una lista simplmente enlazada con nodo ficticio.

Además se utilizará la definición de **posición** que se describe en clase de teoría: la posición del elemento a_i en la lista es el puntero al nodo que contiene, en su campo `sig`, la dirección o puntero al nodo que almacena el elemento a_i, es decir,

- la posición del elemento a_i será el puntero al nodo que contiene a_{i-1} para $i = 2, 3, 4, \ldots, n$
- la posición de a_1 será el puntero al nodo cabecera o nodo ficticio
- la posición `fin(l)` es el puntero al último nodo de la lista enlazada, es decir, el nodo que contiene el elemento a_n. Importante, la posición del elemento a_n es un puntero al nodo que contiene a_{n-1}, por lo tanto, la posición `fin(l)` no corresponde a elemento alguno de la lista, de forma análoga a como sucede con la implementación mediante matrices.
- Si la lista está vacía `fin(l)` devolverá la dirección del nodo ficticio.

Listado de funciones que ya debería tener implementado el alumno tras la realización de la práctica `TAD Lista utilizando memoria dispersa (punteros)` del **Tema 6** siguiendo las indicaciones del enunciado de dicha práctica y las definiciones, dadas en clase de teoría:

- `int creaVacia(Lista *l)`
- `int vacia(Lista *l)`
- `int anula(Lista *l)`
- `int destruye(Lista *l)`
- `void imprime(Lista *l)`
- `tipoPosicion siguiente(tipoPosicion p,Lista *l)`
- `tipoPosicion anterior(tipoPosicion p,Lista *l)`
- `tipoPosicion primero(Lista *l)`
- `tipoPosicion fin(Lista *l)`
- `int inserta(tipoElemento x, tipoPosicion p, Lista *l)`
- `int suprime (tipoPosicion p, Lista *l)`
- `tipoElemento recupera(tipoPosicion p, Lista *l)`