Tema 6: Estructuras de datos lineales: pilas, colas y listas

Tecnología de la Programación

Índice

Índice

Introducción

Sea $\mathcal{D}$ un conjunto de datos, se define el conjunto de secuencias (o cadenas de elementos) de $\mathcal{D}$, denotado por $\mathcal{D}^*$, del siguiente modo:

- $<>$ pertenece a $\mathcal{D}^*$
- Si d pertenece a $\mathcal{D}^*$ y s pertenece a $\mathcal{D}$; entonces sd, ds pertenecen a $\mathcal{D}^*$

Operaciones básicas sobre esta estructura:

- Crear la secuencia vacía
- Añadir un elemento a una secuencia
- Eliminar un elemento de una secuencia
- Obtener un elemento de una secuencia

Introducción

Dependiendo de por donde se puedan realizar las operaciones de añadir y eliminar tendremos unas estructuras u otras

Ejemplos:

- Pila de libros
- Cola del cine

Estructuras de datos lineales

Cada elemento tiene a lo más 1 anterior y 1 siguiente

Operaciones:

- Básicas:
  - Lectura
  - Mostrar
  - Copiar
- Constructores:
  - Crear la estructura vacía
  - Añadir un elemento
- Deconstructor:
  - Eliminar un elemento
- Accesor:
  - Acceso a un elemento
- Auxiliares:
  - Saber si hay un elemento
  - Saber cuántos elementos hay
  - ...

Estructuras de datos lineales

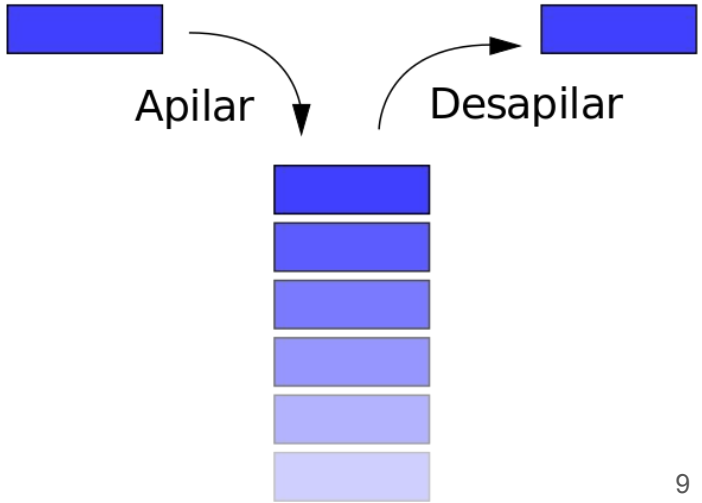
- Pilas:
  - Añadir por el final
  - Eliminar por el final
  - Acceso por el final
- Colas:
  - Añadir por el final
  - Eliminar por el principio
  - Acceso por el principio
- Listas:
  - Añaden, eliminan y dan acceso por cualquier posición
  - Tipos: ordenadas, con posición, con clave, …

Se diferencian en el comportamiento de añadir, eliminar y acceso

Índice

TAD Pila

- Pila es estructura lineal en la que la inserción y la eliminación de datos se produce por un extremo de la estructura al que denominaremos cima
- También se las conoce como estructuras *LIFO (Last-In, First-Out)*
- Único elemento al que se tiene acceso es a la cima
- Aplicaciones:
  - Editor de texto (ejemplo: operación deshacer)
  - Gestión de memoria
  - Transformación de funciones recursivas en iterativas
  - Evaluación de expresiones aritméticas



TAD Pila

- Especificación:
  - Constructores:
    - Crear pila vacía
    - Añadir elementos
  - Accesores:
    - Acceder a la cima
  - Deconstructores:
    - Eliminar un elemento
  - Operaciones básicas:
    - Crear una pila a partir de una secuencia de elementos
    - Mostrar una pila
    - Hacer una copia
  - Auxiliares:
    - Saber si quedan elementos en la pila

- Uso
- Implementación:
  - Estática
  - Dinámica

TAD Pila. Especificación

```
tad Pila(tElemento) // Pila formada por elementos de tipo tElemento
usa
  tElemento
genero
  pila
operaciones
  acción iniciarPila (sal pila p)
  {Pre: }
  {Post: inicia p como una pila vacía}

  acción apilar (e/s pila p, ent tElemento d)
  {Pre: p es una pila que ha sido iniciada previamente}
  {Post: apila en p el elemento d}
```

TAD Pila. Especificación (continuación)

operaciones (continuación)

```
función pilaVacia (pila p) dev booleano
{Pre: p es una pila que ha sido iniciada previamente}
{Post: dev verdad si p está vacía y falso en caso contrario}

función cima (pila p) dev tElemento
{Pre: p es una pila no vacía}
{Post: devuelve el último elemento apilado, no modifica la pila}

acción desapilar (e/s pila p)
{Pre: p es una pila no vacía}
{Post: modifica la pila p, eliminando el último elemento apilado}
```

TAD Pila. Especificación (continuación)

operaciones (continuación)

acción crearPila (s/ pila p)
{Pre: el usuario introducirá por pantalla una secuencia de elementos con una marca de finalización}
{Post: la pila p contiene la secuencia elementos pedida al usuario}

acción copiarPila(e/ pila p, s/ pila p2)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: p2 es una copia de p, p no se destruye}

acción mostrarPila(e/ pila p)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: muestra los elementos de p en orden inverso al que han sido apilados, p no se destruye}

Uso TAD Pila

Ejercicio. Definir un subalgoritmo que utilizando las operaciones del TAD pila, calcule la suma de todos los elementos de una pila de enteros.

Implementación de los TADs

- Cuando usamos un TAD en general, no sabemos cómo está implementado
- Si utilizamos una representación dinámica (punteros) para el TAD Pila:
  - Desapilar liberará memoria
  - Por lo tanto, independientemente de que trabajemos en los subprogramas con parámetros de entrada, la pila quedará destruida
- Para las pilas, y en general cualquier TAD, siempre usar parámetros de entrada/salida
- Habrá veces que interesa mantener la pila → reconstruirla o hacer una copia antes
- Si no es el caso → añadir postcondición diciendo que la pila puede quedar vacía

Implementación de los TADs

IMPORTANTE: Podemos crear pilas de cualquier cosa: pila de enteros, pila de booleanos, pila de caracteres o incluso pilas de cualquier otro TAD (por ejemplo, una pila de tEstudiantes, de libros o de números complejos).

1. Para considerar el caso general, diremos que **nuestra pila es de elementos de tipo tElemento**.
2. En la implementación de las siguientes transparencias, estamos suponiendo que **tElemento es un tipo primitivo** que se puede asignar, devolver, copiar, leer y mostrar con las operaciones elementales. **En caso de no ser** un tipo primitivo, tendríamos que tener **cuidado** y utilizar sus operaciones específicas para leer, mostrar, copiar, etc.

TAD Pila. Implementación estática

Representación

constante

```
    MAXPILA = … // representa tamaño máximo de la pila
```

tipo

```
    tipodef tElemento : tvector[MAXPILA]
    pila = registro
        tvector datos
        entero numElem
    freg
```

Interpretación

La pila tiene `numElem` elementos almacenados en las `numElem` primeras componentes del vector `datos` de forma que la cima está en la componente de `numElem-1`

TAD Pila. Implementación estática

```
acción iniciarPila (sal pila p)
{Pre: }
{Post: inicia p como una pila vacía}
principio
    p.numElem = 0
fin

acción apilar (e/s pila p, ent tElemento d)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: apila en p el elemento d}
principio
    si p.numElem < MAXPILA entonces
        p.datos[p.numElem] = d
        p.numElem = p.numElem+1
    fsi
fin
```

TAD Pila. Implementación estática

```
función pilaVacia (pila p) dev booleano
{Pre: p es una pila que ha sido iniciada previamente}
{Post: dev verdad si p está vacía y falso en caso contrario}
principio
     dev(p.numElem == 0)
fin

función cima (pila p) dev tElemento
{Pre: p es una pila no vacía}
{Post: devuelve el último elemento apilado, no modifica la pila}
principio
     dev(p.datos[p.numElem-1])
fin

acción desapilar (e/s pila p)
{Pre: p es una pila no vacía}
{Post: modifica la pila p, eliminando el último elemento apilado}
principio
     p.numElem = p.numElem-1
fin
```

TAD Pila. Implementación estática

```
acción crearPila (s/ pila p)
{Pre: el usuario introducirá por pantalla una secuencia de elementos con una marca de finalización}
{Post: la pila p contiene la secuencia elementos pedida al usuario}
variables
     tElemento n
principio
     p.numElem = 0
     escribir(“Introduce los elementos de la pila y finaliza con un 0”)
     leer(n)
     mientras que n != 0 AND p.numElem < MAXPILA hacer
         p.datos[p.numElem] = d
         p.numElem = p.numElem+1
         leer(n)
     fmq
fin
```

TAD Pila. Implementación estática

```
acción copiarPila(e/ pila p, s/ pila p2)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: p2 es una copia de p, p no se destruye}
variables
    entero i
principio
    p2.numElem = p.numElem
    para i = 0 hasta numElem - 1 hacer
        p2.datos[i] = p.datos[i]
    fpara
fin
```

TAD Pila. Implementación estática

```
acción mostrarPila(e/ pila p)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: muestra los elementos de p en orden inverso al que han sido apilados, p no se
destruye}
variables
    entero i
principio
    para i = p.numElem - 1 descendiendo hasta 0 hacer
        escribir (p.datos[i])
    fpara
fin
```

TAD Pila. Implementación dinámica

Representación

tipo

```
    Nodo = registro
        tElemento dato
        puntero a Nodo sig
    freg
    pila = puntero a Nodo
```

Interpretación

`Pila` es un puntero a `Nodo` que contiene la cima y un puntero que apunta a un nodo que contiene el dato que ha llegado anteriormente a la pila. El puntero del nodo correspondiente al primer dato apilado apunta a `NULL`. Si la pila está vacía, el puntero apunta a `NULL`.

TAD Pila. Implementación dinámica

```
acción iniciarPila (sal pila p)
{Pre: }
{Post: inicia p como una pila vacía}
principio
     p = NULL
fin

acción apilar (e/s pila p, ent tElemento d)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: apila en p el elemento d}
variables
     puntero a Nodo nuevo
principio
     nuevo = reservar(Nodo)
     dest(nuevo).dato = d
     dest(nuevo).sig = p
     p = nuevo
fin
```

TAD Pila. Implementación dinámica

```
función pilaVacia (pila p) dev booleano
{Pre: p es una pila que ha sido iniciada previamente}
{Post: dev verdad si p está vacía y falso en caso contrario}
principio
     dev(p == NULL)
fin

función cima (pila p) dev tElemento
{Pre: p es una pila no vacía}
{Post: devuelve el último elemento apilado, no modifica la pila}
principio
     dev(dest(p).dato)
fin
```

TAD Pila. Implementación dinámica

```
acción desapilar (e/s pila p)
{Pre: p es una pila no vacía}
{Post: modifica la pila p, eliminando el último elemento apilado}
variables
     puntero a Nodo aux
principio
     aux = p
     p = dest(p).sig
     liberar(aux)
fin
```

Nota. Observad que `cima` y `desapilar` pueden causar problemas si se aplican sobre una pila vacía, pero en la especificación ya imponemos que la pila no debe estar vacía, por lo tanto el usuario se tiene que preocupar de comprobar que la pila no esté vacía antes de llamar a cualquiera de estos dos subalgoritmos.

TAD Pila. Implementación dinámica

```
acción crearPila(sal/ pila p)
variables
     tElemento n
principio
     p=NULL
     escribir(“Introduce los elementos de la pila y finaliza con un 0”)
     leer(n)
     mientras que n != 0 hacer
         nuevo = reservar(Nodo)
         dest(nuevo).dato = d
         dest(nuevo).sig = p
          p = nuevo
         leer(n)
     fmq
fin
```

TAD Pila. Implementación dinámica

```
acción copiarPila(e/ pila p, sal/ pila p2)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: p2 es una copia de p, p no se destruye}
variables
    puntero a Nodo nuevo, aux
principio
    si p == NULL
        p2 = NULL
    si_no
        nuevo = reservar(Nodo)
        p2 = nuevo
        aux = p
        mientras que dest(aux).sig != NULL
            dest(nuevo).dato = dest(aux).dato
            aux = dest(aux).sig
            dest(nuevo).sig=reservar(Nodo)
            nuevo = dest(nuevo).sig
         fmq
         //Copiamos el último
         dest(nuevo).dato = dest(aux).dato
         dest(nuevo).sig = NULL
    fsi
fin
```

TAD Pila. Implementación dinámica

```
acción mostrarPila(e/ pila p)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: muestra los elementos de p en orden inverso al que han sido apilados, p no se
destruye}
variables
    puntero a Nodo aux
principio
    aux = p
    mientras que aux != NULL
        escribir(dest(aux).dato)
        aux = dest(aux).sig
    fmq
fin
```

Índice

TAD Cola

- Cola es estructura lineal en la que los datos se añaden por un extremo y se eliminan y se accede a los elementos por el otro
- También se las conoce como estructuras *FIFO (First-In, First-Out)*
- Ejemplos:
  - Cola para sacar entradas
  - Cola de impresión (aunque puede haber colas con prioridad)

TAD Cola

- Especificación:
  - Constructores:
    - Crear cola vacía
    - Añadir elementos
  - Accesores:
    - Acceder al primer elemento
  - Deconstructores:
    - Eliminar un elemento (que ya ha sido servido)
  - Operaciones básicas:
    - Crear una cola partir de una secuencia de elementos
    - Mostrar una cola
    - Copiar una cola
  - Auxiliares:
    - Saber si quedan elementos en la cola
- Uso
- Implementación:
  - Estática
  - Dinámica

TAD Cola. Especificación

```
tad Cola(tElemento) // Cola formada por elementos de tipo tElemento
usa
  tElemento
genero
  cola
operaciones
  acción iniciarCola (sal cola c)
  {Pre: }
  {Post: inicia c como una cola vacía}

  acción añadir (e/s cola c, ent tElemento d)
  {Pre: c es una cola que ha sido iniciada previamente}
  {Post: añade en c el elemento d}
```

TAD Cola. Especificación (continuación)

operaciones (continuación)

```
función colaVacia (cola c) dev booleano
{Pre: c es una cola que ha sido iniciada previamente}
{Post: dev verdad si c está vacía y falso en caso contrario}

función primero (cola c) dev tElemento
{Pre: c es una cola no vacía}
{Post: devuelve el elemento más antiguo de c y no la modifica}

acción eliminar (e/s cola c)
{Pre: c es una cola no vacía}
{Post: modifica la cola c, eliminando el elemento más antiguo}
```

TAD Cola. Especificación (continuación)

operaciones (continuación)

acción crearCola (s/ cola c)
{Pre: el usuario introducirá por pantalla una secuencia de elementos con una marca de finalización}
{Post: la cola c contiene la secuencia elementos pedida al usuario introducida en ese orden}

acción copiarCola(e/ cola c, s/ cola c2)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: c2 es una copia de c, c no se destruye}

acción mostrarCola(e/ cola c)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: muestra los elementos de c en el orden en el que se han introducido en la cola, c no se destruye}

Uso TAD Cola

Ejercicio. Construir un subalgoritmo que calcule la suma de todos los elementos almacenados en una cola de números reales y no destruya la cola.

TAD Cola. Implementación estática

Representación

constante

```
    MAXCOLA = … // representa tamaño máximo de la cola
```

tipo

```
    tipodef tElemento = tvector[MAXCOLA]
    cola = registro
        tvector datos
        entero primero, último, num
    freg
```

Interpretación

`Num` representa el número de elementos de la cola, los cuales se encuentran almacenados entre las componentes `primero` y `último` en el vector `datos`, estando en la componente `primero` el elemento más antiguo de la cola y en `último` el más reciente

TAD Cola. Implementación estática

```
acción iniciarCola (sal cola c)
{Pre: }
{Post: inicia c como una cola vacía}
principio
    c.num = 0
    c.primero = 0
    c.ultimo = -1
fin

función colaVacia (cola c) dev booleano
{Pre: c es una cola que ha sido iniciada previamente}
{Post: dev verdad si c está vacía y falso en caso contrario}
principio
    devuelve(c.num == 0)
fin
```

TAD Cola. Implementación estática

```
función sumaUno (entero n) dev entero
{Pre: n>=0}
{Post: suma uno a la posición n en el sentido circular de las agujas del reloj, con
tamaño máximo MAXCOLA}
principio
     dev((n+1) MOD MAXCOLA)
fin

acción añadir (e/s cola c, ent tElemento d)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: añade en c el elemento d}
principio
     si c.num < MAXCOLA entonces
          c.num = c.num+1
          c.ultimo = sumaUno(c.ultimo)
          c.datos[c.ultimo] = d
     fsi
fin
```

TAD Cola. Implementación estática

```
función primero (cola c) dev tElemento
{Pre: c es una cola no vacía}
{Post: devuelve el elemento más antiguo de c y no la modifica}
principio
     devuelve(c.datos[c.primero])
fin

acción eliminar (e/s cola c)
{Pre: c es una cola no vacía}
{Post: modifica la cola c, eliminando el elemento más antiguo}
principio
     c.num = c.num - 1
     c.primero = sumaUno(c.primero)
fin
```

TAD Cola. Implementación estática

```
acción crearCola (s/ cola c)
{Pre: el usuario introducirá por pantalla una secuencia de elementos con una marca de finalización}
{Post: la cola c contiene la secuencia elementos pedida al usuario introducida en ese orden}
variables
    tElemento n
principio
    c.num = 0
    c.primero = 0
    c.ultimo = -1
    escribir(“Introduce la secuencia”)
    leer(n)
    mientras que n != 0 AND c.num < MAXCOLA hacer
         c.num = c.num+1
         c.ultimo = sumaUno(c.ultimo)
         c.datos[c.ultimo] = n
         leer(n)
    fmq
fin
```

TAD Cola. Implementación estática

```
acción copiarCola(e/ cola c, s/ cola c2)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: c2 es una copia de c, c no se destruye}
variables
    entero i
principio
    c2.num = c.num
    c2.primero = c.primero
    c2.ultimo = c.ultimo
    i=c.primero
    mientras que i != sumaUno(c.ultimo) //Para que acabe tras copiar el último
        c2.datos[i] = c.datos[i]
        sumaUno(i)
    fmq
fin
```

TAD Cola. Implementación estática

```
acción mostrarCola(e/ cola c)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: muestra los elementos de c en el orden en el que se han introducido en la
cola, c no se destruye}
variables
     entero i
principio
    i=c.primero
    mientras que i != sumaUno(c.ultimo) //Para que acabe tras mostrar el último
        escribir(c.datos[i])
        sumaUno(i)
    fmq
fin
```

Nota. Con esta implementación sólo se pueden representar colas de tamaño máximo `MAXCOLA`.

TAD Cola. Implementación dinámica

Representación

tipo

```
    Nodo = registro
        tElemento dato
        puntero a Nodo sig
    freg
    Cola = registro
        puntero a Nodo primero, ultimo
    freg
```

Interpretación

`Primero` es un puntero a Nodo en el que está el primer elemento de la cola y un puntero a un nodo en el que está el siguiente elemento. El último nodo de la lista apunta a NULL.
`Ultimo` es un puntero que apunta al último elemento de la cola.

TAD Cola. Implementación dinámica

```
acción iniciarCola (sal cola c)
{Pre: }
{Post: inicia c como una cola vacía}
principio
    c.primero = NULL
    c.ultimo = NULL
fin

función colaVacia (cola c) dev booleano
{Pre: c es una cola que ha sido iniciada previamente}
{Post: dev verdad si c está vacía y falso en caso contrario}
principio
    devuelve(c.primero == NULL)
fin
```

TAD Cola. Implementación dinámica

```
acción añadir (e/s cola c, ent tElemento d)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: añade en c el elemento d}
variables
     puntero a Nodo nuevo
principio
     nuevo = reservar(Nodo)
     dest(nuevo).dato = d
     dest(nuevo).sig = NULL
     si c.primero == NULL entonces
          c.primero = nuevo
     sino
          dest(c.ultimo).sig = nuevo
     fsi
     c.ultimo = nuevo
fin
```

TAD Cola. Implementación dinámica

```
función primero (cola c) dev tElemento
{Pre: c es una cola no vacía}
{Post: devuelve el elemento más antiguo de c y no la modifica}
principio
     devuelve(dest(c.primero).dato)
fin

acción eliminar (e/s cola c)
{Pre: c es una cola no vacía}
{Post: modifica la cola c, eliminando el elemento más antiguo}
variables
     puntero a Nodo aux
principio
     aux = c.primero
     c.primero = dest(c.primero).sig
     liberar(aux)
     si c.primero == NULL entonces
          c.ultimo = NULL
     fsi
fin
```

TAD Cola. Implementación dinámica

```
acción crearCola (s/ cola c)
variables
    tElemento n,
    puntero a nodo nuevo
principio
    c.primero = NULL
    c.ultimo = NULL
    escribir(“Introduce la secuencia”)
    leer(n)
    mientras que n != 0 hacer
        nuevo = reservar(Nodo)
        dest(nuevo).dato = n
        dest(nuevo).sig = NULL
        si c.primero == NULL entonces
           c.primero = nuevo
        sino
           dest(c.ultimo).sig = nuevo
        fsi
        c.ultimo = nuevo
        leer(n)
    fmq
fin
```

TAD Cola. Implementación dinámica

```
acción copiarCola(e/ cola c, s/ cola c2)
variables
    puntero a Nodo nuevo, aux
principio //Se podría hacer similar al copiarPila, esto es otra alternativa.
    si c.primero == NULL //Si la cola está vacía, entonces solo tenemos que iniciar c2
        c2.primero = NULL
        c2.ultimo = NULL
    si_no //Si no está vacía, copiaremos el primero y luego iremos añadiendo al final.
        nuevo = reservar(Nodo)
        dest(nuevo).dato = dest(c.primero).dato //Guardo los datos del primer elemento
        dest(nuevo).sig = NULL
        c2.primero = nuevo
        c2.ultimo = nuevo
        aux = dest(c.primero).sig //El segundo elemento
        mientras que aux != NULL
            nuevo = reservar(Nodo)
            dest(nuevo).dato = dest(aux).dato
            dest(nuevo).sig = NULL
            dest(c2.ultimo).sig = nuevo
            c2.ultimo = nuevo
            aux = dest(aux).sig
        fmq
    fsi
fin
```

TAD Cola. Implementación dinámica

```
acción mostrarCola(e/ cola c)
{Pre: c es una cola que ha sido iniciada previamente}
{Post: muestra los elementos de c en el orden en el que se han introducido en la
cola, c no se destruye}
variables
    puntero a Nodo aux
principio
    aux = p
    mientras que aux != NULL
        escribir(dest(aux).dato)
        aux = dest(aux).sig
    fmq
fin
```

Índice

TAD Lista

- Lista es estructura de datos lineal en la que los elementos se acceden, añaden y eliminan en cualquier posición
- Existen distintas variantes y entre ellas varía el comportamiento de las operaciones:
  - Lista con posición
  - Lista ordenada

TAD lista con posición

Los elementos están indexados y se accede por posición

- Especificación:
  - Constructores:
    - Crear lista vacía
    - Añadir elemento en posición
  - Accesores:
    - Extraer elemento de posición
  - Deconstructores:
    - Eliminar elemento de posición
  - Operaciones básicas:
    - Crear una cola partir de una secuencia de elementos
    - Mostrar una cola
    - Copiar una cola
  - Auxiliares:
    - Saber si quedan elementos en la lista
    - Longitud de la lista
- Uso
- Implementación:
  - Dinámica

TAD Lista con posición. Especificación

```
tad Lista(tElemento) //Lista formada por elementos de tipo tElemento
usa
  tElemento
genero
  lista
operaciones
  acción iniciarLista (sal lista l)
  {Pre: }
  {Post: inicia l como una lista vacía}

  acción insertar (e/s lista l, ent tElemento d, ent entero pos)
  {Pre: l es una lista que ha sido iniciada previamente, pos>0}
  {Post: inserta el elemento d en la posición pos de l, si pos es
         mayor que longitud de l, inserta el elemento al final de l}
```

TAD Lista con posición. Especificación (cont.)

Operaciones (continuación)

```
función extraer (lista l, entero pos) dev tElemento
{Pre: l debe haber sido previamente iniciada y pos debe ser menor o igual
      que la longitud de l y mayor que 0}
{Post: devuelve el elemento que ocupa la posición pos en l y no modifica
       la lista}

acción eliminar (e/s lista l, ent entero pos)
{Pre: l es una lista que ha sido iniciada previamente y pos debe ser menor
      o igual que la longitud de l y mayor que 0}
{Post: modifica la lista l eliminando el elemento que ocupa la posición
       pos}
```

TAD Lista con posición. Especificación (cont.)

Operaciones (continuación)

```
función listaVacía (lista l) dev booleano
{Pre: l debe haber sido previamente iniciada}
{Post: devuelve Verdad si l está vacía y falso en caso contrario}

función longitud (lista l) dev entero
{Pre: l es una lista que ha sido iniciada previamente}
{Post: devuelve el número de elementos de l}
```

TAD Lista con posición. Especificación (cont.)

Operaciones (continuación)

```
acción crearLista (s/ lista l)
{Pre: el usuario introducirá por pantalla una secuencia de elementos con una marca de finalización}
{Post: la lista l contiene la secuencia elementos pedida al usuario}

acción copiarLista(e/ lista l, s/ lista l2)
{Pre: l es una lista que ha sido iniciada previamente}
{Post: l2 es una copia de l, l no se destruye}

acción mostrarLista(e/ lista l2)
{Pre: l es una lista que ha sido iniciada previamente}
{Post: muestra los elementos de l ordenados por posición}
```

TAD Lista con posición. Implementación dinámica

Representación

tipo

```
    Nodo = registro
        tElemento dato
        puntero a Nodo sig
    freg
    Lista = registro
        puntero a Nodo primero
        entero long
    freg
```

Interpretación

`Lista` es un registro donde `Primero` es un puntero que apunta a un nodo en cuyo campo `dato` se encuentra el primer elemento de la lista y cuyo campo `sig` apunta a un nodo en el que está él 2º y así sucesivamente. El puntero del nodo del último dato de la lista apunta a `NULL`. `long` es un entero que representa el número de elementos que tiene la lista.

TAD Lista con posición. Implementación dinámica

```
acción iniciarLista (sal lista l)
{Pre: }
{Post: inicia l como una lista vacía}
principio
    l.primero = NULL
    l.long = 0
fin

función listaVacía (lista l) dev booleano
{Pre: l debe haber sido previamente iniciada}
{Post: devuelve Verdad si l está vacía y falso en caso contrario}
principio
    devuelve(l.long == 0)
fin
```

TAD Lista con posición. Implementación dinámica

```
función longitud (lista l) dev entero
{Pre: l es una lista que ha sido iniciada previamente}
{Post: devuelve el número de elementos de l}
principio
    devuelve(l.long)
fin
```

TAD Lista con posición. Implementación dinámica

```
acción insertar (e/s lista l, ent tElemento d, ent entero pos)
{Pre: l es una lista que ha sido iniciada previamente, pos > 0}
{Post: inserta el elemento d en la posición pos de l, si pos es mayor que longitud de l, inserta el elemento
al final de l}
variables
      puntero a Nodo ant, aux, nuevo
      entero cont
principio
      nuevo = reservar(Nodo)
      aux = l.primero
      cont = 1
      mientras que cont < pos AND aux != NULL hacer
            cont++
            ant = aux
            aux = dest(aux).sig
      fmq
      dest(nuevo).dato = d
      dest(nuevo).sig = aux
      si aux == l.primero entonces
            l.primero = nuevo
      sino
            dest(ant).sig = nuevo
      fsi
      l.long++
fin
```

TAD Lista con posición. Implementación dinámica

```
función extraer (lista l, entero pos) dev tElemento
{Pre: l debe haber sido previamente iniciada y pos debe ser menor o igual que la
longitud de l y mayor que 0}
{Post: devuelve el elemento que ocupa la posición pos en l y no modifica la lista}
variables
     puntero a Nodo aux
     entero cont
principio
     aux = l.primero
     cont = 1
     mientras que cont < pos hacer
          cont++
          aux = dest(aux).sig
     fmq
     devuelve(dest(aux).dato)
fin
```

TAD Lista con posición. Implementación dinámica

```
acción eliminar (e/s lista l, ent entero pos)
{Pre: l es una lista que ha sido iniciada previamente y pos debe ser menor o igual que la longitud
de l y mayor que 0}
{Post: modifica la lista l eliminando el elemento que ocupa la posición pos}
variables
      puntero a Nodo ant, aux
      entero cont
principio
      aux = l.primero
      cont = 1
      mientras que cont < pos hacer
            cont++
            ant = aux
            aux = dest(aux).sig
      fmq
      si l.primero == aux entonces // para eliminar primer elemento
            l.primero = dest(l.primero).sig
      sino
            dest(ant).sig = dest(aux).sig
      fsi
      liberar(aux)
      l.long--
fin
```

TAD Lista con posición. Implementación dinámica

```
acción crearLista (s/ lista l)
variables tElemento n, puntero a Nodo aux
principio
    l.primero = NULL
    l.long = 0
    escribir(“Introduce la secuencia”)
    leer(n)
    aux = NULL
    mientras que n != 0 hacer
        nuevo = reservar(Nodo)
        dest(nuevo).dato = n
        dest(nuevo).sig = NULL
        si l.primero == NULL entonces
           l.primero = nuevo
        sino
           dest(aux).sig = nuevo
        fsi
        aux = nuevo
        l.long++
        leer(n)
    fmq
fin
```

TAD Lista con posición. Implementación dinámica

```
acción copiarLista(e/ lista l, s/ lista l2)
variables
    puntero a Nodo nuevo, aux
principio
    si l.long == 0
        l2.long = 0
        l2.primero = NULL
    si_no
        l2.long = l.long
        nuevo = reservar(Nodo)
        l2.primero = nuevo
        aux = l.primero
        mientras que dest(aux).sig != NULL
            dest(nuevo).dato = dest(aux).dato
            aux = dest(aux).sig
            dest(nuevo).sig=reservar(Nodo)
            nuevo = dest(nuevo).sig
        fmq
        //Copiamos el último
        dest(nuevo).dato = dest(aux).dato
        dest(nuevo).sig = NULL
    fsi
fin
```

TAD Lista con posición. Implementación dinámica

```
acción mostrarLista(e/ lista l)
{Pre: l es una lista que ha sido iniciada previamente}
{Post: muestra los elementos de l ordenados por posición}
variables
    puntero a Nodo aux
principio
    aux = l.primero
    mientras que aux != NULL
        escribir(dest(aux).dato)
        aux = dest(aux).sig
    fmq
fin
```

Uso TAD lista con posición

- ¿Complejidad de las operaciones extraer, insertar y eliminar?
- Siempre debemos elegir representaciones que permitan implementar las operaciones de la forma más eficiente posible
- Uso del TAD lista:
  - Esquema de recorrido (sin modificación)
  - Esquema de recorrido (destruyendo la lista)

Ejercicio 10 Hoja 7: TAD Agenda

Queremos gestionar una agenda telefónica. De cada persona queremos almacenar su nombre y su número de teléfono.

- Definir un TAD Agenda que simule el comportamiento de una agenda telefónica. Entre otras operaciones, se deben poder realizar búsquedas.
- Diseñar un subalgoritmo que muestre un listado ordenado alfabéticamente con los datos de las personas incluidas en la agenda.
- Implementar dinámicamente el TAD Agenda (tipos y operaciones) y dar el coste computacional de cada una de las operaciones.

Ejercicio 10 Hoja 7: TAD Agenda

- Cada elemento tiene un anterior y un siguiente ⇒ Estructura lineal
- Acceso a cualquier elemento ⇒ Lista:
  - No usamos lista con posición
  - Lista con acceso por clave
- Operaciones. Como mínimo harán falta las siguientes operaciones, dependiendo de lo que se pida pueden hacer falta más cosas
  - Constructures: iniciar y añadir
  - Accesores: consultar ← acceso por clave
  - Deconstructores: eliminar ← por clave
  - Función auxiliar: existe la clave

TAD Persona. Especificación.

```
tad Persona
genero
  persona
operaciones
  acción CrearPersona (ent cadena nombre, ent cadena tfno, sal persona p)
  {Pre: }
  {Post: Inicia p como una persona con nombre nomb y teléfono tfno}

  acción nombre(ent persona p, sal cadena nomb)
  {Pre: La persona p está creada}
  {Post: nomb contiene el nombre de la persona p}

  acción telefono(ent persona p, sal cadena tfno)
  {Pre: La persona p está creada}
  {Post: tfno contiene el teléfono de la persona p}
```

TAD Persona. Especificación.

Operaciones (continuación)

```
función personasIguales(ent persona p1, ent persona p2) devuelve booleano
{Pre: Las personas p1 y p2 están creadas}
{Post: devuelve verdad si p1 y p2 tienen el mismo nombre, falso en caso
contrario}

acción copiarPersona(ent persona p1, sal persona p2)
{Pre: Las personas p1 y p2 están creadas}
{Post: Se realiza una copia de los datos de p1 en p2}

acción mostrarPersona(ent persona p)
{Pre: La persona p está creada}
{Post: Muestra por pantalla los datos de la persona p}
```

TAD Persona. Implementación.

Representación

```
    tipo
    persona = registro
        cadena nomb
        cadena tfno
    freg
```

Interpretación: persona es un registro que contiene dos campos: nomb que representa el nombre de la persona y tfno representa el teléfono.

Implementación de las operaciones:

.....

TAD Agenda. Especificación.

```
tad Agenda (de elementos de tipo Persona)
usa
  persona
genero
  agenda
operaciones
  acción iniciarAgenda (sal agenda A)
  {Pre: }
  {Post: Inicia A como una agenda vacía}

  acción añadir(e/s agenda A, ent persona p)
  {Pre: La persona p está creada y la agenda A está iniciada}
  {Post: Si en la agenda A no existe una persona con el mismo nombre que
         p, añade a la agenda A los datos de la persona p}
```

TAD Agenda. Especificación.

Operaciones (continuación)

```
función existe (agenda A, persona p) devuelve booleano
{Pre: la agenda A y la persona p están iniciadas}
{Post: devuelve verdad si en la agenda A existe una persona con el mismo
        nombre que p y falso en caso contrario}

acción consultar(ent agenda A, ent cadena nomb, sal cadena tfno)
{Pre: La agenda A está iniciada y existe en ella una persona cuyo nombre
      coincide con el contenido la variable nomb}
{Post: tfno contiene el teléfono de la persona de la agenca A cuyo
        nombre es nomb}

acción borrar(ent/sal agenda A, ent cadena nomb)
{Pre: La agenda A está iniciada}
{Post: Si en la agenda A existe una persona cuyo nombre sea nomb lo
        elimina}
```

TAD Agenda. Especificación.

El ejercicio pide diseñar un subalgoritmo que muestre un listado ordenado alfabéticamente con los datos de las personas incluidas en la agenda.

acción listar (ent agenda A)
{Pre: la agenda A está iniciada}
{Post: muestra por pantalla un listado ordenado alfabéticamente conteniendo los datos de todas las personas almacenadas en la agenda. La agenda NO puede modificarse.}

- Si no se incluye: su implementación será $O(n^2)$.
- Si se incluye: se puede implementar el TAD Agenda mediante una lista enlazada de nodos de personas que estén ordenadas alfabéticamente por sus nombres. De esta forma, la implementación de la acción listar podrá aprovecharse de la implementación del TAD Agenda y ser O(n).

TAD Agenda. Implementación dinámica.

Representación

```
    tipo
    Nodo = registro
        persona per
        puntero a Nodo sig
    freg
    agenda = puntero a Nodo
```

Interpretación: agenda es un puntero al primer nodo, que es un registro que contiene una persona y un puntero al siguiente nodo, de forma que las personas estén ordenadas alfabéticamente por su campo nombre. El puntero del último Nodo apunta a NULL.

Implementación de las operaciones:

.....