Tecnología de la programación
Sesión 21

Deberes para la evaluación continua

1. Implementar las operaciones añadir y existe del TAD Agenda.

Objetivos de la sesión

1. TAD Lista: implementación dinámica (transparencias 50-67).
2. Plantear el ejercicio del TAD Agenda (Hoja 7, ejercicio 10).

Guion

TAD Lista

Una vez entendidos el TAD Pila y el TAD Cola, llegamos al último, el TAD Lista. Es el más fácil de entender: es una estructura de datos lineal en la que se puede acceder, añadir y eliminar en cualquier posición. Es decir, no tenemos las restricciones de Pila y Cola. Podéis entenderlo como un vector, donde puedes modificar cualquier posición. Lo implementaremos dinámicamente como listas enlazadas de nodos, de esta forma podremos añadir tantos datos como queramos (con el límite de la memoria).

Existen variantes para el TAD Lista, y el comportamiento de las operaciones va a variar. Por ejemplo:

- Lista con posición: los elementos se almacenan en la posición que digamos explícitamente.
- Lista ordenada: los elementos se ordenan siguiendo un orden (por ejemplo, si la lista es de enteros, se puede añadir ordenado de menor a mayor).

Vamos a hacer la especificación, implementación y uso del primer tipo, es decir, listas con posición. El otro caso es muy similar, de hecho, hay muchos ejercicios donde ya habéis insertado en una lista ordenada.

Como operaciones del TAD, tendremos constructores (iniciar lista vacía, añadir un elemento en una posición), accesores (extraer un elemento de una posición), deconstructores (eliminar un elemento de una posición), operaciones básicas (mostrar, copiar, crear una lista a partir de una secuencia) y auxiliares (saber si una lista está vacía, saber la longitud de la lista).

Es decir, nuestra especificación sería la siguiente **(usando tElemento como tipo primitivo y asumiendo que el primer elemento está en la posición 1):**

ESPECIFICACIÓN TAD Lista con posición

```
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}

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}

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}

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}
```

Si tElemento fuese otro TAD, esto tendría que ser una acción

fin_especificación

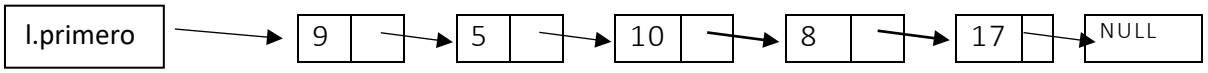
¿Cuál es la complejidad de las operaciones extraer, insertar y eliminar?

- Dependerá de la implementación que hagamos. Lo más seguro es que sean todas O(n), porque en el peor caso seguramente tendrás que recorrer toda la lista para extraer, insertar o eliminar al final.
- Nos damos cuenta entonces que hay una diferencia con el TAD Pila: allí las operaciones apilar, cima y desapilar eran O(1).
- También hay diferencia con el TAD Cola, donde las operaciones añadir, eliminar y primero eran también O(1).

IMPLEMENTACIÓN TAD LISTA

Vamos a implementar el TAD Lista directamente de forma dinámica usando listas enlazadas de nodos.

Implementaremos la lista como un registro con dos campos: uno será un puntero al primer nodo de la lista, y el otro campo será un entero que representará la longitud de la lista (el número de elementos que tiene).



La implementación de las operaciones es sencilla, porque son las que ya habéis hecho con listas enlazadas de nodos, o variantes de ellas. Si algo no se entiende, recomiendo dibujarlo paso a paso.

Implementación TAD Lista con posición
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.

Operaciones

```
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
```

```
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


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


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


acción eliminar (e/s lista l, ent entero pos)
```

Es muy similar al de insertar en una lista ordenada, mírate el vídeo del tema 4 si no lo entiendes.

```
{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
```

Atentos a la implementación del eliminar, os suele costar. ¡¡¡Dibújatelo!!!

```
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
```

Realmente, es como el copiar de pilas, es copiar una lista enlazada de nodos.

```
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


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
fin implementación
```

USO TAD LISTA

Por lo general, los ejercicios consistirán de un modo u otro en recorrer (o buscar) en una lista. Tenemos dos esquemas de recorrido

- Recorrido sin modificar la lista:

```
pos = 1
l = longitud(L)
mientras que pos <= l hacer
    d = extraer(L, pos)
    //tratar d
    pos = pos +1
fmq
```

Nota: la llamada a extraer(L,pos) es O(n) con la implementación vista, luego el esquema es $O(n^2)$.

- Recorrido destruyendo la lista:

```
mientras que NOT listaVacia(L) hacer
    d = extraer(L,1)
    //tratar el d
    eliminar(L,1)
fmq
```

Nota: la llamadas a extraer(L,1) y eliminar(L,1) son O(1) con la implementación vista, luego este esquema es O(n).

También tenemos la opción de hacer una copia de la lista y recorrer la copia, destruyéndola.

HOJA 7 EJERCICIO 10: TAD Agenda

El ejercicio está planteado a partir de la transparencia 68 (hasta el final del tema). Cuando nos encontremos con un ejercicio de este estilo, tenemos que pensar qué TAD (o variante de un TAD) nos interesa para modelarlo. En este caso, queremos gestionar una agenda telefónica, que almacenará nombre y número de teléfono de cada persona.

Además, nos imponen que se deben poder realizar búsquedas y mostrar un listado ordenado alfabéticamente de todas las personas almacenadas en la lista.

Si nos damos cuenta, nos interesa entonces usar el TAD Lista. Sin embargo, no nos interesa el TAD Lista con posición, sino crear un TAD Lista ordenada donde añadamos la lista ordenando por orden alfabético las personas. De esta forma, podremos mostrar el listado ordenado alfabéticamente con coste computacional O(n). Si hubiésemos insertado por posición en cualquier sitio, entonces la operación de listar alfabéticamente sería $O(n^2)$. Si nos imponen una operación que va a ser muy utilizada, es muy importante hacerla eficiente, y para ello es necesario usar la representación adecuada e incluirla en la especificación.

Seguro que en algún examen o ejercicio se os imponen condiciones así

Hay que tener mucho cuidado, porque ahora la lista será de personas (que es un TAD, no un tipo primitivo).

DEBERES: implementar las operaciones existe y añadir, suponiendo que el TAD Agenda se ha representado mediante una lista enlazada de nodos donde las personas se insertan ordenadas alfabéticamente por nombre. Hay que hacerlas lo más eficientes posible.