6. El TDA Lista

6.1. Definición

6.2. Operaciones del TDA

6.3. Implementación con vectores

6.4. Implementación con listas enlazadas

6.5. Comparativa

Bibliografía:

- Weiss

- Aho, Hopcroft y Ullman

6.1. Definición

Secuencia de cero o más elementos:

- $a_1, a_2, \ldots, a_n$ donde $n \geq 0$ y
- cada elemento a_i es de tipo genérico T.

- Pueden existir elementos repetidos (a diferencia de los conjuntos).

- Se suelen denominar:
  a_1: primer elemento de la lista,
  a_n: último elemento de la lista.

- Si n = 0 se dice que la lista está vacía.

- Los elementos pueden estar ordenados mediante una clave en función de sus posiciones: $a_i < a_{i+1}$.

- Es importante caracterizar la posición del último elemento de la lista.
- Puede ser una estructura flexible (tamaño variable).
- Puede accederse, insertarse y borrarse en cualquier posición.
- Pueden realizarse operaciones con listas, como concatenación o extraer una sublista.
- Son utilizadas en: recuperación de información, traducción de lenguajes de programación o simulación.

  Suponemos que nuestra Lista está ordenada, permitiendo elementos repetidos.

6.2. Operaciones del TDA Lista

Cada operación vendrá caracterizada por su cabecera:
Necesita: parámetros que se le pasan,
Modifica: parámetros dato-resultado,
Produce: parámetros resultado.

Ejemplo: resp/error: lógico;

Modifica/produce pueden ser intercambiables a la hora de su implementación.

Crear_lista (L: Lista, ok: lógico)
{Nec.: una lista, L, y un valor lógico, ok.
Mod.: ok, indicando si la operación tuvo o no éxito.
Prod.: una estructura de lista, L, vacía (si tuviese contenido anterior se perderá).**}**

Borrar_lista (L:Lista, ok: lógico)
{Nec.: una lista, L, y un valor lógico, ok.
Mod.: ok, indicando si la operación tuvo o no éxito.
Prod.: una estructura de lista, L, vacía.**}**

Tamaño (L: Lista, n: numérico)
{Nec.: una lista, L, y una variable numérica, n.
Prod.: el valor de n, indicando el tamaño de la lista.**}**

Vacía? (L: Lista, resp: lógico)
{Nec.: una lista, L, y un valor lógico, resp.
Mod.: resp, indicando si la lista está (cierto) o no (falso) vacía.**}**

Llena? (L: Lista, resp: lógico)
{Nec.: una lista, L, y un valor lógico, resp.
Mod.: el valor de resp, indicando si la lista está (cierto) o no (falso) llena.**}**

Sgte (L: Lista, p: posición, q: posición, error: lógico)
{Nec.: una lista, L, y una posición, p, dentro de L.
Mod.: la posición, q, indicando la siguiente posición a p en L, si es posible.
Prod.: error (cierto) si se intenta acceder a posiciones, p o q, inexistentes.**}**

Anterior (L: Lista, p: posición, q: posición, error: lógico)
{Nec: una lista, L, y una posición, p, dentro de ella.
Mod.: la posición, q, indicando la posición anterior a p en L, si es posible.
Prod.: error (cierto) si se intenta acceder a posiciones, p o q, inexistentes.**}**

Contenido (L: Lista, p: posición, E: elemento, error: lógico)
{Nec.: una lista, L, y una posición, p, dentro de la misma.
Mod.: el valor de E, que será una copia del elemento en la posición p de L.
Prod.: error (cierto) si intenta acceder a una posición no válida de L.**}**

Primero (L: Lista, p: posición, error: lógico)
{Nec.:una lista, L.
Mod.: el valor de p, para que apunte a la primera posición de L, si es posible.
Prod.: error (cierto) si la lista está vacía.**}**

Ultimo (L: Lista, p: posición, error: lógico)
{Nec.: una lista, L.
Mod.: el valor de p, para que apunte a la última posición de L, si es posible.
Prod.: error es cierto si la lista está vacía. **}**

Final? (L: Lista, p: posición, resp: lógico)
{Nec.: una lista, L, y una posición dentro de la misma, p.
Prod.: resp es cierto si p es la posición siguiente al último dentro de L.
resp es falso en caso contrario.**}**

Recorrer_lista (L: Lista)
{Nec.: una lista L.
Prod.: muestra por pantalla los elementos de L, si hay alguno.**}**

Buscar (L: Lista, E: elemento, p: posición, error: lógico)
{Nec.: una lista, L, y un elemento de la misma, E.
Prod.: la posición p de L donde se encuentra E.
Si el elemento no está, error será cierto y p indicará la posición de la lista donde debería encontrarse.**}**

Insertar (L: Lista, E: elemento, error: lógico) /
Insertar (L: Lista, E: elemento, p: posición, error: lógico)
{Nec.: una lista, L, un elemento del mismo tipo que los elementos de L, E.
Mod.: la lista, insertando ordenadamente E, si no hay restricciones de espacio.
Prod.: error será cierto si no ha podido insertarlo.**}**

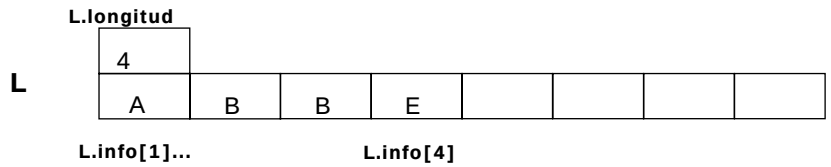
Eliminar (L: Lista, E: elemento, error: lógico) /
Eliminar (L: Lista, E: elemento, p: posición, error: lógico)
{Nec.: una lista, L, un elemento de L, E.
Mod.: L, borrando E de ella, si es que pertenece.
Prod.: error si el elemento no está o no puede eliminarlo.**}**

Además se van a utilizar operaciones del TDA Elemento:

Mostrar (E: elemento, error: lógico)
{Nec.: un elemento, E.
Prod.: muestra el contenido del elemento, si es posible.
Mod.: error a cierto/falso si no puede mostrarlo o no ha podido acceder al valor**}**

Elemento_nulo (E:elemento, error: lógico)
{Nec.: un elemento, E.
Prod.: un elemento vacío asociado a E, borrándolo si tenía algún contenido.
Mod.: error será cierto, si no ha podido crearlo o borrarlo.**}**

Iguales? (E1, E2:elemento, resp: lógico) /
Menor? (E1, E2:elemento, resp: lógico) /
Mayor? (E1, E2:elemento, resp: lógico) /...
{Nec.: 2 elementos, E1 y E2.
Prod.: respuesta es cierto/falso en función de la comparación de los valores.**}**



6.3. Implementación con vectores

```
{Declaración de tipos}
ELEMENTO =  T; {Genérico}
POSICIÓN =  numérico;
LISTA = registro de
    longitud: numérico;
    INFO: vector [1..TAM] de ELEMENTO;
fin registro;

/* IMPLEMENTACIÓN EN C */

#define TAM 1000
typedef T elemento;
typedef int posicion;
typedef struct {
    int longitud;
    elemento info[TAM];
} LISTA;

Lista L;
```

6.3.1. Operaciones elementales con vectores
6.3.1.1. Operaciones independientes de la implementación

```
Algoritmo Vacía? (L: Lista, resp: lógico) es
tam_real: numérico;
Inicio
    tamaño (L, tam_real);
    si tam_real = 0 entonces
        resp := cierto;
    sino
        resp := falso;
    finsi
Fin

Algoritmo Llena? (L: Lista, resp: lógico) es
tam_real : numérico;
Inicio
    tamaño (L, tam_real);
    si tam_real = TAM entonces
        resp := cierto;
    sino
        resp := falso;
    finsi
Fin
```

6.3.1.2. Operaciones dependientes de la implementación

```
Algoritmo Crear_lista (L: Lista, ok: lógico) es
Inicio
    ¿Borrar_lista?/¿Crear estructura?
Fin

Algoritmo Tamaño (L: Lista, tam_real: numérico) es
Inicio
    tam_real := L.longitud;
Fin

Algoritmo Borrar_lista (L:Lista, OK: lógico) es
i: entero;
error: lógico;
Inicio
    {No debe tener errores, al ser variables estáticas}
    ok := cierto;
    L.longitud : = 0;
    para i desde 1 hasta TAM hacer
        Elemento_nulo (L.INFO(i), error);
        si error = cierto entonces
          ok := falso;
        finsi
    finpara;
Fin

Algoritmo Final? (L: Lista, p: posición, resp: lógico) es
tam_real: numérico;
Inicio
    tamaño (L, tam_real);
    si p > tam_real entonces
        resp := cierto;
    sino
        resp := falso;
    finsi
Fin
```

```
Algoritmo Contenido (L: Lista, p: posición, E: elemento, error: lógico) es
tam_real: numérico;
Inicio
    Tamaño (L, tam_real);
    si p > 0 AND p <= tam_real entonces
        E := L.INFO(p);
        error := falso;
    sino {p apuntaba al siguiente al último}
      Escribir "Error: acceso a posición inexistente";
      error := cierto;
    finsi;
Fin

Algoritmo Primero (L: Lista, p: posición, error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces
        p := 1;
        error := falso;
    sino
        error := cierto;
        Escribir "Error: Lista Vacía";
    finsi
Fin
```

```
Algoritmo Ultimo (L: Lista, p: posición, error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces
            tamaño (L, p);
            error := falso;
    sino
            error := cierto;
            Escribir "Error: Lista Vacía";
    finsi
Fin
```

```
Algoritmo Sgte (L: Lista, p: posición, q: posición, error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces
            Final? (L, p, resp);
            si not resp entonces
               q := p + 1;    {Puede ser el sgte al último}
            sino
              {p apuntaba al siguiente al último}
               Escribir "Error: acceso a posición inexistente";
            finsi
    sino
            {la lista está vacía}
            Escribir "Error: Lista Vacía";
    finsi
    error := resp;
Fin
```

```
Algoritmo anterior (L: Lista, p: posición, q: posición, error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces {la lista no está vacía}
            si p = 1 entonces  {se pregunta por el anterior al primero}
               Escribir "Error: no hay anterior al primero";
               error := cierto;
            sino
              q := p – 1;
              error := falso;
            finsi;
    sino   {la lista está vacía}
            error := cierto;
            Escribir "Error: Lista Vacía";
    finsi
Fin

{opción 2}
Inicio
    Vacía? (L, resp);
    si not resp entonces {la lista no está vacía}
            primero(L, r);
            si p = r entonces
               Escribir "Error: no hay anterior al primero";
               error := cierto;
            sino
              q := p – 1;
              error := falso;
            finsi;
    sino
            error := cierto;
            Escribir "Error: Lista Vacía";
    finsi
Fin
```

6.3.2. Operaciones no elementales con vectores

6.3.2.1. Operaciones independientes de la implementación

Algoritmo Recorrer_lista (L: Lista) es

```
p: posicion;
resp: lógico;
E: elemento;
Inicio
    primero (L, p, resp);
    mientras not resp hacer
      contenido (L, p, E, resp);
      mostrar (E, resp);
      sgte (L, p,  p, resp);
     {p era una posición válida, no fallará, pero puede ser el sgte al
     último}
      final? (L, p, resp);
    fin mientras;
Fin
```

{opción 2}

```
Inicio
    Vacía? (L, resp);
    si  resp entonces
          Escribir "Lista vacía";
    sino  {Al menos hay un elemento}
          {Sólo  final?() puede hacer que resp sea cierto}
          {Podemos quitar resp sin afectar al resultado}
          primero (L, p, resp);
          repetir
            contenido (L, p, E, resp);
            mostrar (E, resp);
            sgte (L, p,  p, resp);
            final? (L, p, resp);
          hasta que resp = cierto;
    fin si;
Fin
```

```
{Búsqueda en una lista ordenada. Si se usase la búsqueda dicotómica, sería dependiente de la
    implementación}
Algoritmo Buscar (L: Lista, E: elemento, p: posición, error: lógico) es
encontrado, es_menor, es_final: lógico;
E2: elemento;
Inicio
    Vacía? (L, error);  {si está vacía, error es cierto, no está}
    si  not error  entonces  {hay al menos un elemento}
          Primero (L, p, es_final); {es_final es falso}
          es_menor : = cierto;
          mientras not es_final and es_menor hacer
            Contenido (L, p,  E2, resp);
            menor? (E2, E, es_menor);
            si es_menor entonces {E2 < E}
               Sgte (L, p, p, es_final);
               Final? (L, p, es_final);
            finsi;
          fin mientras:
          si es_final entonces {E no está}
            error := es_final;
          sino   {es_menor = falso; E2 >= E}
            iguales? (E, E2, encontrado);
            error := not encontrado;
          finsi
    finsi
Fin
```

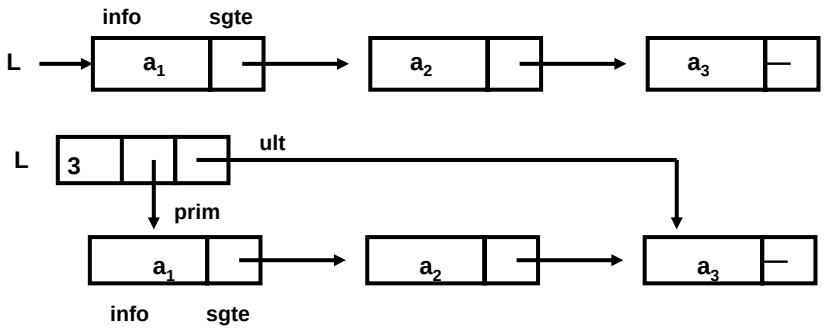
6.3.2.2. Operaciones dependientes de la implementación

{**Insertar**(): Si la lista no está llena, realizar una búsqueda, insertándolo en la posición p, manteniendo el orden; para ello desplazamos los elementos entre L.longitud y p}

```
Algoritmo Insertar (L: Lista, E: elemento, error: lógico) es
resp, no_está: lógico;
p, q: posición;
Inicio
 error := falso;
 Llena? (L, resp);
 si not resp entonces
    Vacía? (L, resp);
    si not resp entonces
      Buscar (L, E, p, no_está);
      {Si el elemento no está y p fuese la siguiente al último: (L.longitud + 1) y no se entrará
           en el siguiente bucle}
       para q desde L.longitud hasta p paso -1 hacer
          L.info(q+1) := L.info(q);
       fin para;
    sino
      {Lista vacía, debe insertarse en la posición 1}
      p := 1;
    fin si;
    {acciones comunes}
    L.info(p) := E;
   L.longitud := L.longitud + 1;
 sino
    error := cierto;
finsi
Fin
```

```
Algoritmo Eliminar (L: Lista, E: elemento, error: lógico) es
resp, no_está: lógico;
p, q: posición;
Inicio
 error := falso;
 Vacía? (L, resp);
  si resp = falso entonces
     Buscar (L, E, p, no_está);
     {si el elemento no está, no_está será cierto}
     si no_está entonces
           error := cierto; {elemento no está en L}
     sino
       {el elemento está en la posición p}
       para q desde p hasta L.longitud - 1 hacer
           L.info(q) := L.info(q+1);
       fin para;
       L.longitud := L.longitud - 1;
      finsi
  sino
     error := cierto; {por lista vacía}
  finsi
Fin
```

6.4. Implementación con listas enlazadas



Ejemplo de tipo de dato cuya definición es recursiva.

Opciones:
- Lista sin cabecera
(toda la estructura está formada por variables dinámicas),
- Lista con cabecera
(la cabecera es una variable estática; el contenido son variables dinámicas)

```
{Declaración de tipos}
    ELEMENTO = T;
    NODO = registro de
            info: ELEMENTO; {Genérico}
            sgte: puntero a NODO;
    fin registro;
    POSICION = puntero a NODO;
    LISTA = registro de
            longitud: numérico;
            prim, ult: POSICION;
    fin registro;

/* IMPLEMENTACIÓN EN C */
typedef  T elemento;
struct NODE {
    elemento info;
    struct NODE * sgte;
} ;
typedef struct NODE nodo;
typedef struct NODE * posición;
typedef struct {
    int longitud;
    posición prim, ult;
} LISTA;
```

6.4.1. Operaciones elementales dependientes de la implementación con listas

- La operación tamaño queda igual que en el caso de los vectores.

```
Algoritmo Final? (L, p, resp) es
L: Lista; p: posición; resp: lógico;
{Posición siguiente al último será un puntero a nil}
Inicio
    si p = nil entonces
            resp := cierto;
    sino
            resp := falso;
    finsi
Fin
```

```
Algoritmo Primero (L: Lista, p: posición, error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces
            p := L.prim;
            error := falso;
    sino
            error := cierto;
            Escribir "Error: Lista Vacía";
    finsi
Fin
```

```
Algoritmo Ultimo (L: Lista, p: posición, error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces
            p:= L.ult;
            error := falso;
    sino
            error := cierto;
            Escribir "Error: Lista Vacía";
    finsi
Fin
```

```
Algoritmo Sgte (L: Lista, p: posición, q: posición,  error: lógico) es
resp: lógico;
Inicio
    Vacía? (L, resp);
    si not resp entonces
            {Puede apuntar al sgte al último}
            Final? (L, p, resp);
            si not resp entonces
               q := p→.sgte;
            sino
               {p apuntaba al siguiente al último}
                Escribir "Error: acceso a posición inexistente";
            finsi
    sino {la lista está vacía}
            Escribir "Error: Lista Vacía";
    finsi
    error := resp;
Fin
```

```
Algoritmo Contenido (L: Lista, p: posición, E: elemento, error: lógico) es
Inicio
    Final? (L, p, error);
    si not error entonces
             E := p→.info;
    sino {p apuntaba al siguiente al último}
       Escribir "Error: acceso a posición inexistente";
    finsi;
Fin
```

6.4.2. Operaciones no elementales dependientes de la implementación con listas

- Todas las operaciones no elementales tendrán que realizar un recorrido parcial o total de la lista.
- La operación Crear_lista llamaba a Borrar_lista que, además de inicializar los valores de la cabecera, ahora tiene que liberar la memoria que ocupan todos los nodos de la lista.
- Usaremos las primitivas obtener() y liberar() que vimos en el tema de gestión de memoria dinámica; será necesario particularizar ambas ya que tenemos que obtener o liberar un elemento de tipo posición, o lo que es lo mismo, puntero a nodo.

Empezaremos por la operación anterior, que recorrerá una lista enlazada; el procedimiento está basado en la operación buscar(), que es independiente de la implementación

- si la lista está vacía dará error;
- si la posición p es la primera de la lista, devolverá nil;
- en otro caso un apuntador a la posición anterior a p y, en ambos casos, error = falso;
- si p no está en la lista, devolverá q = L.ult

```
Algoritmo anterior (L: Lista, p: posición, q: posición, error: lógico) es
resp, encontrado: lógico;
s : posición;
Inicio
error := falso;
Vacía? (L, resp);
si not resp entonces {la lista no está vacía}
        primero (L, s, resp); {la primera vez es falso}
        q := nil;
        encontrado := falso;
        repetir
          si p = s entonces encontrado := cierto;
          sino
            q := s;
            sgte (L, s, s, resp);
            final? (L, s, resp);
          finsi
        hasta que resp = cierto o encontrado = cierto;
sino {la lista está vacía}
        error := cierto;
finsi
Fin
```

```
{Usando las operaciones anterior() y sgte()}
Algoritmo Insertar(L: Lista, E: elemento, error: lógico) es
p, q, tmp: posición;
resp: lógico;
Inicio
Llena? (L, error);
si not error entonces  {Localiza las posiciones p y q}
  q := nil; {sería la anterior al primero}
  vacía? (L, resp);
  si not resp entonces {no vacía → hay algún elmento}
    buscar (L, E, p, resp); {p será una posición de L}
    anterior (L, p, q, resp); {resp será falso}
  sino {si la lista está vacía}
     p := nil;  {cuando L está vacía}
  finsi;
  {p apunta a donde debería estar; insertaremos tras q}  {Acciones comunes a insertar}
  L.longitud := L.longitud + 1;
  Obtener (tmp);
  tmp→.info := E;
  tmp→.sgte := p; {p puede ser nil}
 {Análisis de los casos}
  si p = nil  entonces {inserta en una lista vacía o p es el siguiente al último}
     L.ult := tmp;
  finsi;
  si q = nil entonces {Inserta al principio de L}
    L.prim := tmp;
   sino {Inserta en una posición intermedia, ya que q <> nil}
     q→.sgte := tmp;
   finsi
finsi {lista llena}
Fin
```

```
Algoritmo Eliminar (L: Lista, E: elemento, error: lógico) es
resp, no_está: lógico;
p, q: posición; {p, posición de E si está; q, posición anterior}
Inicio
 error := falso;
 Vacía? (L, resp);
  si resp = falso entonces
     Buscar (L, E, p, no_está); {si el elemento no está, no_está será cierto}
     si no_está entonces
           error := cierto; {elemento no está en L}
     sino {el elemento está en la posición p}
       anterior (L, p, q, resp);   {Casos especiales: eliminar único elemento de la  lista,
                                    o eliminar sólo el primero o eliminar sólo el último}
        si p = L.prim y p = L.ult entonces  {es el único elemento de L}
              L.prim := nil;
              L.ult := nil;
        sino  {Hay más de uno}
              si p = L.prim entonces {eliminar el primero}
                 L.prim := p→.sgte;
              sino
                si p = L.ult entonces {eliminar el último}
                   q→.sgte := nil;  {≡ q→.sgte := p→.sgte;}
                   L.ult := q;
                sino q→.sgte := p→.sgte;
                finsi;
              finsi;
        finsi;
        L.longitud := L.longitud - 1;
        Liberar (p); {Común para casos especiales y normal}
 sino error := cierto; {por lista vacía}
 finsi
Fin
```

```
{Borrar_lista:
```

- Pone a 0 la longitud de la lista y la elimina de memoria, si fuese necesario.
- Si no hubo problemas, OK = cierto, si no, OK = falso.
- Como en esta implementación la cabecera de la lista no es un objeto dinámico, no tiene sentido ni crear ni destruir la estructura. Por lo tanto, será lo mismo borrar una lista que crearla, salvo que hay que tener en cuenta que hay que eliminar los elementos para los que se hizo reserva dinámica de memoria. **}**

```
Algoritmo Borrar_lista (L:Lista, OK: lógico) es
p, q: posición;
resp: lógico;
Inicio
Vacía? (L, ok);
si not ok entonces
     L.longitud := 0;
     primero (L, p, resp); {final? (L, p, resp); Si la lista no está vacía, puede eliminarse}
     repetir
           q := p;
           sgte (L, p, p);
           liberar (q);
           final? (L, p, resp);
     hasta que resp = cierto;
     ok := resp;
     L.prim := nil;
     L.ult := nil;
finsi
Fin
```

6.5. Comparativa

Ventajas/Inconvenientes de la implementación con vectores:

- Las operaciones elementales tienen coste O(1): crear_lista(), borrar_lista(), tamaño(), vacía? (), llena? (), final? (), sgte(), anterior(), primero(), ultimo(), contenido()
- Operaciones no elementales tienen costes:
  buscar() -O(n/2) o O(log n)-,
  recorrer_lista() -O(n)-,
  insertar() o eliminar() -O(n+ log n).
- El tamaño está limitado (lo cual contradice la noción de flexibilidad), tanto cuando hay pocos (se desperdicia) como cuando hay demasiados (overflow).

Ventajas/Inconvenientes de la implementación con listas enlazadas:

- Algunas de las operaciones tienen coste O(1):
  crear_lista(), borrar_lista(), tamaño(), vacía? (), llena? (),
  final? (), sgte(), primero(), ultimo(), contenido()
- Operaciones no elementales
  buscar() -O(n/2) -,
  recorrer_lista(), insertar(), o eliminar() -O(n)-,
  anterior() -O(n/2).
- El tamaño es variable, en función de las necesidades.