Parte II: Estructuras de datos y algoritmos

UNIVERSIDAD DE CANTABRIA

Tema 11. Tipos abstractos de datos.

- Conceptos básicos.
- Listas.
- Pilas.
- Colas.
- Vectores.
- Conjuntos.
- Mapas.
- Árboles.
- Árboles binarios.

Notas:

UNIVERSIDAD DE CANTABRIA

En este tema se revisan posibles especificaciones en Ada de los siguientes tipos de datos abstractos: Listas, Pilas, Colas, Vectores, Conjuntos, Mapas, Árboles y Árboles binarios. Se apuntan además algunas posibles implementaciones que cumplen con las especificaciones planteadas.

A modo de ejemplo revisaremos con mayor profundidad algunas posibles implementaciones sólo para el ADT Listas.

También a modo de ejemplo describiremos el ADT para las listas doblemente enlazadas definido en el estándar (anexo A.18.3 del Manual de Referencia del Ada).

1. Concepto de clase o ADT

UNIVERSIDAD DE CANTABRIA

Una clase o tipo de datos abstracto (ADT) es:

- **un tipo de datos con una determinada estructura, más**
- **un conjunto de operaciones para manejar esos datos**

El conjunto de operaciones permite el uso de la estructura de datos sin conocer los detalles de su implementación

- **los programas que usan la clase son *independientes* de la forma en la que éste se implementa**
- **no es necesario conocer los detalles internos del tipo de datos ni de su implementación**

Se dice que la clase *encapsula* el tipo de datos junto a sus operaciones, ocultando los detalles internos.

Notas:

UNIVERSIDAD DE CANTABRIA

El **tipo de dato** de una variable es el conjunto de valores que puede tomar esa variable. Un **tipo de datos abstracto (ADT) o clase** es un modelo matemático de una estructura de información con un conjunto de operaciones definidas sobre el modelo. Las operaciones se describen de forma independiente a como se implementan.

Las ventajas principales que introduce la utilización del concepto de clase son:

1. Las operaciones de la clase permiten que el programa que lo usa sea independiente de la forma en la que se implementa la clase.
2. Las clases **encapsulan** un tipo de datos junto a todas las operaciones a realizar con ese tipo de datos. Esto quiere decir que todo lo relativo a esa clase está localizado en un lugar concreto del programa, que tiene todos sus detalles internos ocultos

El encapsulado permite una gran facilidad en cualquier cambio en la estructura de una clase, ya que afecta sólo a un reducido número de rutinas. ¡El programa que utiliza el tipo de datos no necesita ser modificado!.

Por ejemplo, los conjuntos, junto con las operaciones de unión, intersección y diferencia de conjuntos, forman un ejemplo de tipo abstracto de datos. Independientemente de cómo se implemente el conjunto (mediante una lista, un array, etc.), el conjunto se puede utilizar en base a las operaciones que se han definido.

ADTs en Ada

UNIVERSIDAD DE CANTABRIA

El lenguaje Ada define un conjunto de paquetes dentro del estándar que implementan algunos tipos de datos abstractos.

Estos paquetes se agrupan dentro de `Ada.Containers` en varios paquetes hijo genéricos (anexo A.18), algunos de ellos son:

- Vectores: `Ada.Containers.Vectors`
- Listas enlazadas: `Ada.Containers.Doubly_Linked_Lists`
- Mapas: `Ada.Containers.Hashed_Maps`, `Ada.Containers.Ordered_Maps`
- Conjuntos: `Ada.Containers.Hashed_Sets`, `Ada.Containers.Ordered_Sets`

En las instancias de estos paquetes se proporciona el elemento.

2. Listas

UNIVERSIDAD DE CANTABRIA

Una lista es una secuencia de objetos ordenados, en la que se puede:

- insertar o eliminar elementos en cualquier posición
- recorrer los elementos de la lista hacia adelante y opcionalmente, hacia atrás
  - normalmente uno por uno mediante las operaciones `primer_elemento` y `siguiente` (`último` y `anterior`)
- buscar un elemento en la lista
- etc.

Los objetos son todos del mismo tipo, el `Elemento`

La posición de los objetos es del tipo abstracto `Posicion`

Notas:

UNIVERSIDAD DE CANTABRIA

Las **listas** son secuencias de objetos ordenados con una estructura particularmente flexible, que se puede aumentar o disminuir mediante la inserción o eliminación de elementos en cualquier posición.

Matemáticamente una lista es una secuencia de cero o más elementos de un tipo determinado, en la que el orden se define en función de la posición en la lista.

- $a_1, a_2, \ldots a_n$ con $n \geq 0$.

Todos los elementos de la lista son del mismo tipo de datos, que llamaremos tipo “Elemento”. Para independizar este tipo de la lista, lo declararemos como un parámetro genérico.

Las operaciones que se pueden realizar sobre listas son insertar o eliminar elementos en cualquier posición, recorrer los elementos de la lista, buscar un elemento, etc. Dependiendo de la forma en que está implementada la lista se permitirá recorrer la lista hacia adelante sólo, o indistintamente hacia adelante y hacia atrás. Generalmente se suele recorrer la lista elemento a elemento, para lo que existen operaciones para ver el primer elemento, y el siguiente elemento a uno dado. Ello permite recorrer toda la lista. Si se desea recorrer en orden inverso las operaciones son último elemento, y elemento anterior.

Cada objeto está localizado en una posición de la lista, que será del tipo abstracto “Posicion”. Este tipo no es necesariamente un número.

Operaciones básicas de las listas

operación	in	out	in out	errores
Haz_Nula			la_lista	
Inserta_Al-Principio	el _elemento		la_lista	no_cabe
Inserta_Al-Final	el _elemento		la_lista	no_cabe
Inserta_-Delante	el _elemento la_posicion		la_lista	no_cabe posicion_-incorrecta
Elimina	la_posicion	el_elemento	la_lista	posicion_-incorrecta
Esta_Vacia	la_lista	booleano		

Notas:

UNIVERSIDAD DE CANTABRIA

Para formar un tipo abstracto de datos basado en este modelo matemático debemos definir una serie de operaciones a realizar con los objetos del tipo LISTA. Las operaciones más habituales serán:

- *Haz_nula*. Inicializa la lista haciéndola nula. Imprescindible antes de poder usar una lista.
- *Inserta_Al_Principio*: Inserta un elemento al principio de la lista indicada (delante del primero); falla si la lista está llena y no caben más elementos.
- *Inserta_Al_Final*: Inserta un elemento al final de la lista indicada (detrás del último); falla si la lista está llena y no caben más elementos.
- *Inserta_Delante*: Inserta en la lista indicada el elemento indicado, delante del elemento cuya posición se indica; falla si la lista está llena y no caben más elementos, o si la posición indicada es incorrecta.
- *Elimina*: Elimina el elemento de la posición indicada de la lista indicada, y devuelve el elemento eliminado. Falla si la posición indicada es incorrecta.
- *Esta_Vacia*: Devuelve `True` si la lista indicada está vacia, y `False` en caso contrario.

Operaciones para recorrer la lista

UNIVERSIDAD DE CANTABRIA

operación	in	out	in out	errores
Localiza	el_elemento la_lista	la_posicion		
Elemento_de	la_posicion la_lista	el_elemento		posicion_- incorrecta
Primera_Pos	la_lista	la_posicion		
Siguiente	posicion_- actual la_lista	posicion_- siguiente		posicion_- incorrecta
Ultima_Pos	la_lista	la_posicion		
Anterior	posicion_- actual la_lista	posicion_- anterior		posicion_- incorrecta

Notas:

UNIVERSIDAD DE CANTABRIA

Operaciones para recorrer las listas:

- *Localiza*: Busca en la lista indicada el elemento indicado, devolviendo su posicion. Devuelve la constante **`Posicion_Nula`** si el elemento no se encuentra.
- *Elemento_De*: Es la operación contraria a Localiza; devuelve el elemento de la lista indicada que se encuentra en la posición indicada. Falla si la posición indicada es incorrecta.
- *Primera_Pos*: Devuelve la posición del primer elemento de la lista indicada. Si la lista no tiene elementos devuelve la constante **`Posicion_Nula`**.
- *Ultima_Pos*: Devuelve la posición del último elemento de la lista. Si la lista no tiene elementos devuelve la constante **`Posicion_Nula`**.
- *Siguiente*: Devuelve la posición siguiente a la indicada, en la lista indicada. Si la posición indicada es la última (en cuyo caso no hay más posiciones), devuelve **`Posicion_Nula`**. Falla si la posición indicada es incorrecta.
- *Anterior*: Devuelve la posición anterior a la indicada, en la lista indicada. Si la posición indicada es la primera (en cuyo caso no hay más posiciones), devuelve **`Posicion_Nula`**. Falla si la posición indicada es incorrecta.

Especificación de listas en Ada, con parámetros de error

UNIVERSIDAD DE CANTABRIA

```
with Elementos; use Elementos; -- definición del tipo Elemento
package Listas is

   type Posicion is private;
   Posicion_Nula : constant Posicion;
   type Lista is private;
   type Errores_Listas is (Correcto,Posicion_Incorrecta,No_Cabe);

   procedure Haz_Nula (La_Lista : in out Lista);
   procedure Inserta_Al_Principio
                (El_Elemento : in Elemento;
                 La_Lista : in out Lista;
                 Error : out Errores_Listas);
   procedure Inserta_Al_Final
                (El_Elemento : in Elemento;
                 La_Lista : in out Lista;
                 Error : out Errores_Listas);
```

Especificación de listas (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Inserta_Delante(
                    El_Elemento : in Elemento;
                    La_Posicion : in Posicion;
                    La_Lista : in out Lista;
                    Error : out Errores_Listas);

procedure Elimina  (La_Posicion : in Posicion;
                    La_Lista : in out Lista;
                    El_Elemento : out Elemento;
                    Error : out Errores_Listas);

function Esta_Vacia (La_Lista : in Lista) return Boolean;

function Primera_Pos (La_Lista : in Lista) return Posicion;
function Ultima_Pos (La_Lista : in Lista) return Posicion;

function Localiza     (El_Elemento : in Elemento;
                       La_Lista : in Lista)
                       return Posicion;
```

Especificación de listas (cont.)

UNIVERSIDAD DE CANTABRIA

```
      procedure Elemento_De(La_Posicion : in Posicion;
                            La_Lista : in Lista;
                            El_Elemento: out Elemento;
                            Error : out Errores_Listas);

      procedure Siguiente  (Posicion_Actual : in Posicion;
                            La_Lista : in Lista;
                            Posicion_Siguiente : out Posicion;
                            Error : out Errores_Listas);

      procedure Anterior   (Posicion_Actual : in Posicion;
                            La_Lista : in Lista;
                            Posicion_Anterior : out Posicion;
                            Error : out Errores_Listas);

private
         ...
end Listas;
```

Notas:

UNIVERSIDAD DE CANTABRIA

La especificación del ADT mostrado en las trasparencias anteriores está realizada sin utilizar excepciones:

- Para retornar el error cometido, todos los procedimientos en los que puede ocurrir un error retornan un código de error (del tipo **`Errores_Listas`**) en el parámetro de salida **`Error`**.
- En caso de que no haya error, el procedimiento devuelve el valor **`Correcto`** en el parámetro **`Error`**.

La especificación anterior es adecuada para lenguajes que no dispongan de excepciones. Pero en el caso de que el lenguaje soporte excepciones, como en Ada, la especificación (y su implementación también) es más elegante si se usan excepciones.

La especificación de las listas con excepciones, se muestra en las transparencias que aparecen a continuación. Observar que en este caso varios de los procedimientos se pueden convertir en funciones, cuyo uso resulta más cómodo.

Se utiliza además, un paquete genérico para independizar las listas del tipo Elemento.

Paquete Excepciones_Listas

UNIVERSIDAD DE CANTABRIA

Creamos un paquete con las excepciones de las listas

- así las excepciones de todas las instancias son iguales
- si no, cada instancia tendría sus propias excepciones

```
package Excepciones_Listas is

   Posicion_Incorrecta : exception;
   No_Cabe             : exception;

end Excepciones_Listas;
```

Especificación de listas en Ada, con excepciones (1/4)

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Listas;

generic
   type Elemento is private;
   with function "="(E1,E2 : Elemento) return Boolean;
package Listas is

   type Posicion is private;
   Posicion_Nula : constant Posicion;

   type Lista is private;

   Posicion_Incorrecta : exception
     renames Excepciones_Listas.Posicion_Incorrecta;

   No_Cabe : exception
     renames Excepciones_Listas.No_Cabe;

   procedure Haz_Nula
     (La_Lista : in out Lista);
```

Especificación de listas en Ada, con excepciones (2/4)

UNIVERSIDAD DE CANTABRIA

```
procedure Inserta_Al_Principio
   (El_Elemento : in Elemento;
    La_Lista : in out Lista);
-- puede elevar No_Cabe
procedure Inserta_Al_Final
   (El_Elemento : in Elemento;
    La_Lista : in out Lista);
-- puede elevar No_Cabe

procedure Inserta_Delante
   (El_Elemento : in Elemento;
    La_Posicion : in Posicion;
    La_Lista : in out Lista);
-- puede elevar No_Cabe o Posicion_Incorrecta

procedure Elimina
   (La_Posicion : in Posicion;
    La_Lista : in out Lista;
    El_Elemento : out Elemento);
-- puede elevar Posicion_Incorrecta
```

Especificación de listas en Ada, con excepciones (3/4)

UNIVERSIDAD DE CANTABRIA

```
function Esta_Vacia
   (La_Lista : in Lista)
   return Boolean;

function Primera_Pos
   (La_Lista : in Lista)
   return Posicion;

function Ultima_Pos
   (La_Lista : in Lista)
   return Posicion;

function Localiza
   (El_Elemento : in Elemento;
    La_Lista : in Lista)
   return Posicion;
```

Especificación de listas en Ada, con excepciones (4/4)

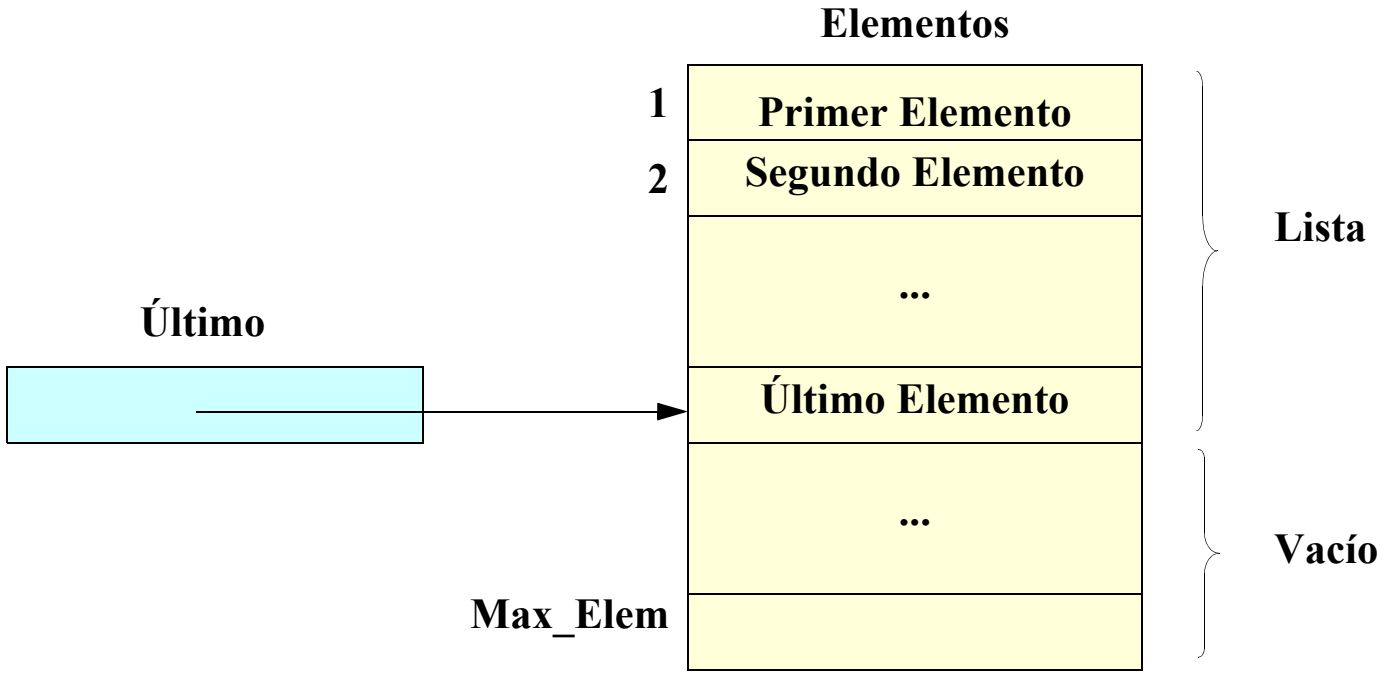
UNIVERSIDAD DE CANTABRIA

```
   function Elemento_De
     (La_Posicion : in Posicion;
      La_Lista : in Lista)
     return Elemento;
   -- puede elevar Posicion_Incorrecta

   function Siguiente
     (Posicion_Actual : in Posicion;
      La_Lista : in Lista)
     return Posicion;
   -- puede elevar Posicion_Incorrecta

   function Anterior
     (Posicion_Actual : in Posicion;
      La_Lista : in Lista)
     return Posicion;
   -- puede elevar Posicion_Incorrecta
private
   ...
end Listas;
```

2.1. Implementación de listas mediante arrays simples

UNIVERSIDAD DE CANTABRIA

La lista se representa mediante un array en el que cada casilla almacena un elemento, y los elementos se ordenan según el índice de la casilla



Estructura de datos privada

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Listas;
generic
   type Elemento is private;
   with function "="(E1,E2 : Elemento) return Boolean;
   Max_Elementos : Positive:= 100; -- Nuevo parámetro genérico
package Listas_Array_Simple is
   -- Todo igual que antes
private
   type Posicion is new Natural;
   Posicion_Nula : constant Posicion :=0;
   Max_Elem      : constant Posicion:=Posicion(Max_Elementos);

   type Los_Elementos is array
     (Posicion range 1..Max_Elem) of Elemento;

   type Lista is record
      Elementos : Los_Elementos;
      Ultimo : Posicion;
   end record;
end Listas_Array_Simple;
```

Implementación de algunas operaciones

UNIVERSIDAD DE CANTABRIA

```
procedure Inserta_Delante
   (El_Elemento   : in Elemento;
    La_Posicion   : in Posicion;
    La_Lista      : in out Lista)
is
begin
    if La_Lista.Ultimo >= Max_Elem then
       raise No_Cabe;
    elsif La_Posicion>La_Lista.Ultimo or La_Posicion=0 then
       raise Posicion_Incorrecta;
    else
       for Q in reverse La_Posicion .. La_Lista.Ultimo loop
          La_Lista.Elementos(Q+1):=La_Lista.Elementos(Q);
       end loop;
       La_Lista.Ultimo:=La_Lista.Ultimo+1;
       La_Lista.Elementos(La_Posicion):=El_Elemento;
    end if;
end Inserta_Delante;
```

Implementación de algunas operaciones (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Elimina
   (La_Posicion : in Posicion;
    La_Lista : in out Lista;
    El_Elemento : out Elemento)
is
begin
    if La_Posicion>La_Lista.Ultimo or La_Posicion=0 then
       raise Posicion_Incorrecta;
    else
       El_Elemento:=La_Lista.Elementos(La_Posicion);
       for I in La_Posicion..La_Lista.Ultimo-1 loop
          La_Lista.Elementos(I):=La_Lista.Elementos(I+1);
       end loop;
       La_Lista.Ultimo:=La_Lista.Ultimo-1;
    end if;
end Elimina;
```

Implementación de algunas operaciones (cont.)

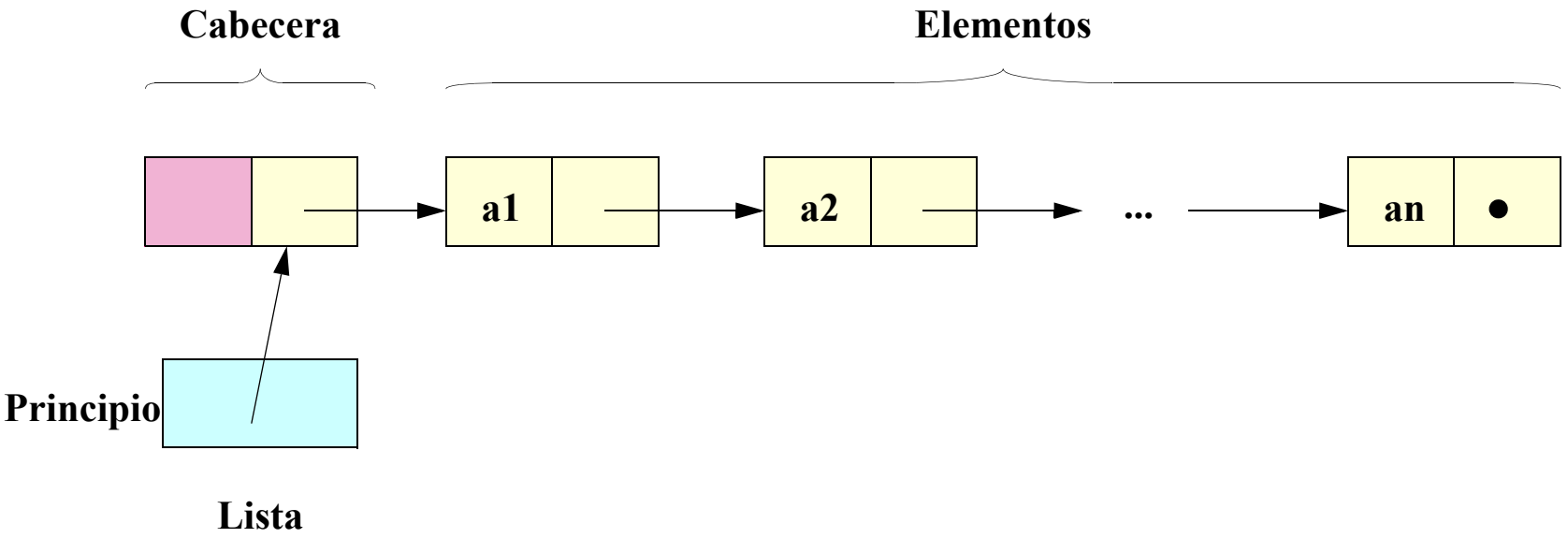
UNIVERSIDAD DE CANTABRIA

```
function Localiza
   (El_Elemento : in Elemento;
    La_Lista    : in Lista)
  return Posicion
is
begin
   for Pos in 1 .. La_Lista.Ultimo loop
      if La_Lista.Elementos(Pos) = El_Elemento then
         return Pos;
      end if;
   end loop;
   return Posicion_Nula;
end Localiza;
```

2.2.- Implementación de listas mediante punteros

UNIVERSIDAD DE CANTABRIA

La lista de celdas simplemente enlazadas con punteros



- el primer elemento está vacío,
- la posición de un elemento es un puntero al anterior

Notas:

UNIVERSIDAD DE CANTABRIA

En esta segunda implementación, denominada lista encadenada, utilizaremos punteros para conectar cada elemento de la lista con el siguiente, tal como se muestra en la figura superior. Se dice que es una lista simplemente enlazada, porque cada celda tiene un único puntero que apunta a la siguiente

La posición de un elemento se define como un puntero que apunta al elemento anterior al deseado. Esto se hace para conseguir una mayor eficiencia de la operación de eliminar. Para poder indicar el primer elemento de la lista se hace que la primera celda de la lista (denominada cabecera) esté vacía. Esta estructura se implementará en Ada de la forma siguiente:

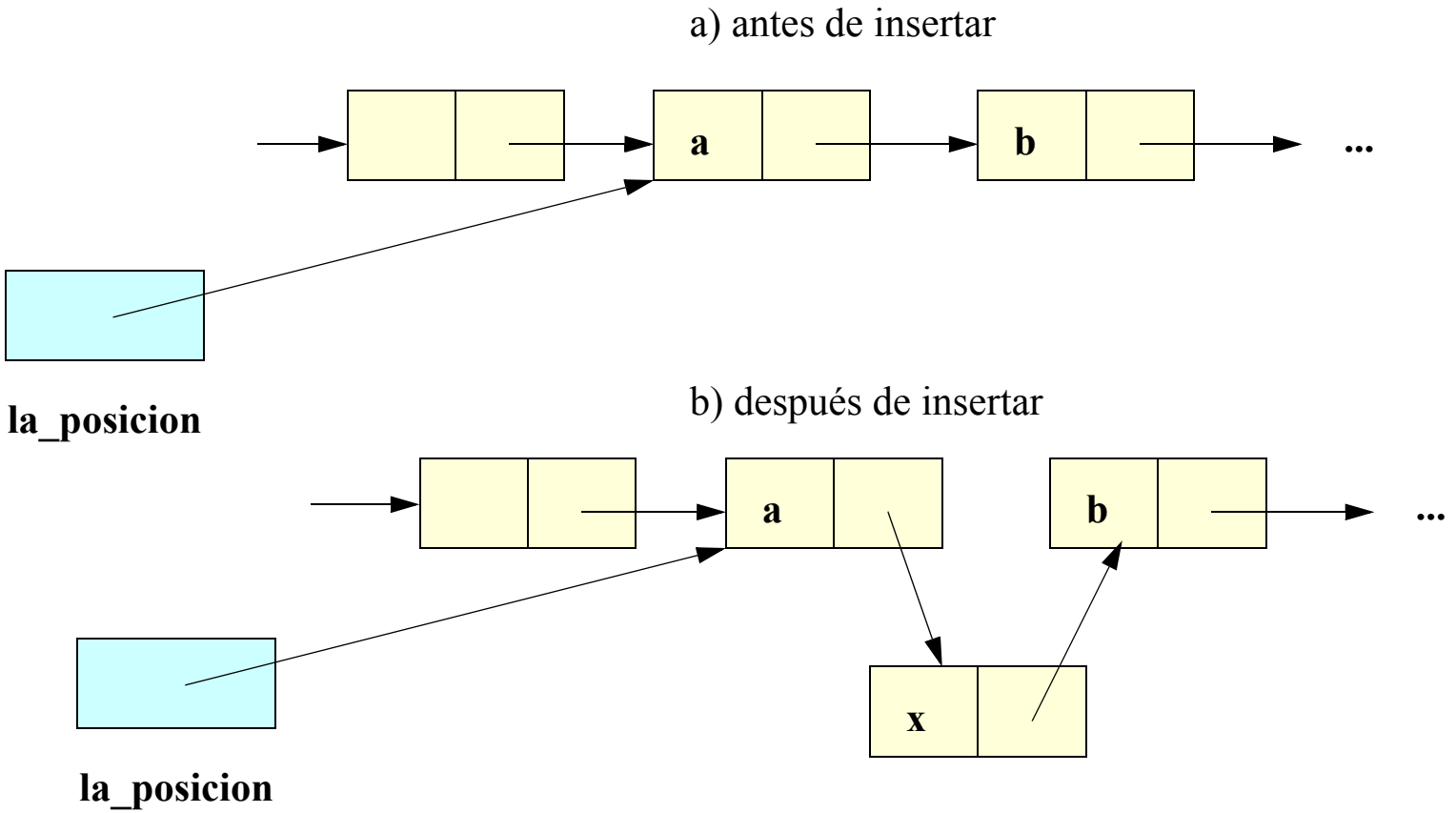
Estructura de datos privada

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Listas;
generic
   type Elemento is private;
   with function "="(E1,E2 : Elemento) return Boolean;
package Listas_Simplemente_Enlazadas is
   -- todo igual que antes
private
   type Tipo_Celda;
   type Posicion is access Tipo_Celda;
   Posicion_Nula : constant Posicion := null;

   type Tipo_Celda is record
      Contenido : Elemento;
      Proximo   : Posicion;
   end record;

   type Lista is record
      Principio : Posicion;
   end record;
end Listas_Simplemente_Enlazadas;
```

Implementación de las operaciones

Diagrama de inserta_delante:



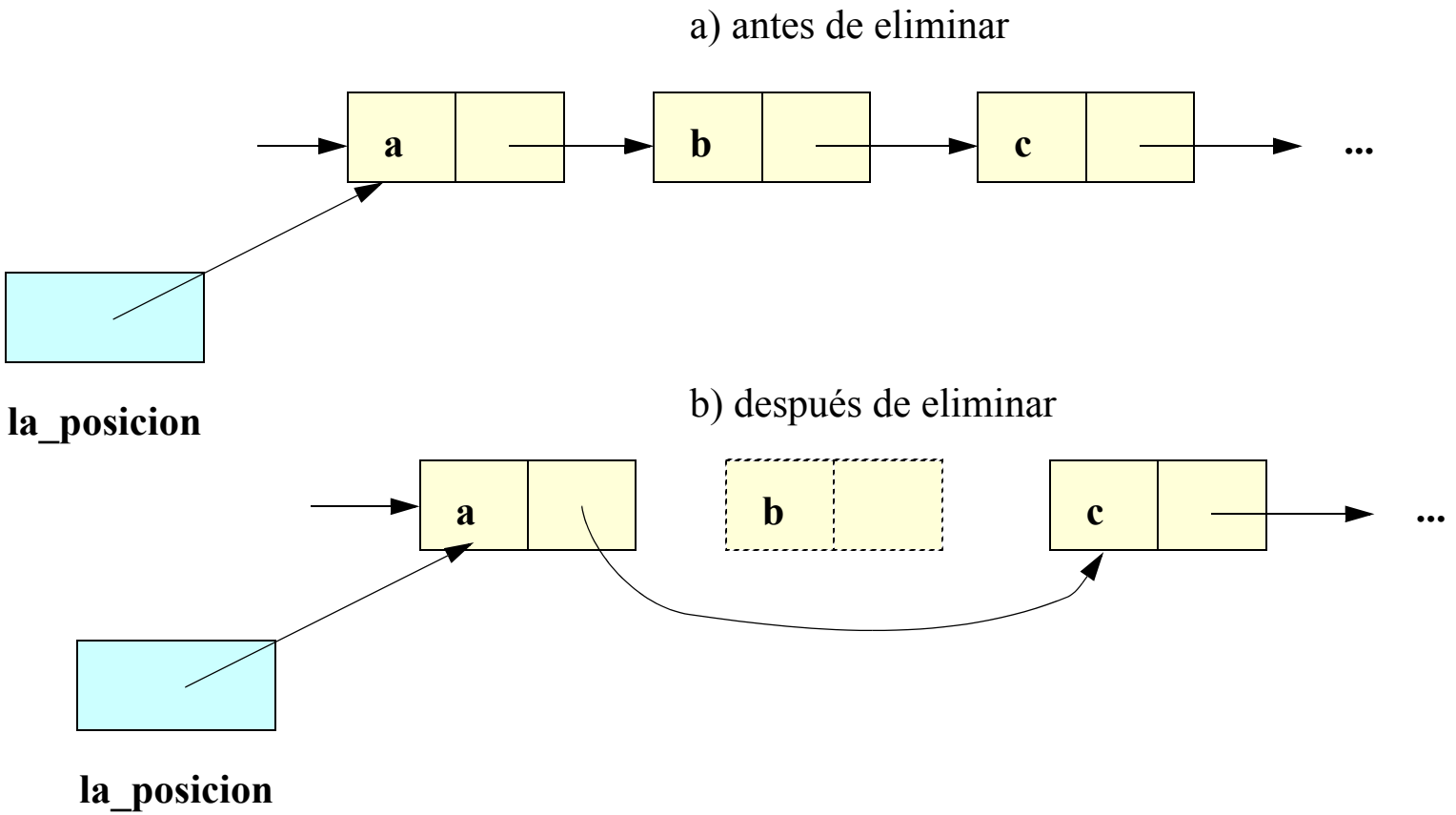
Inserta delante

UNIVERSIDAD DE CANTABRIA

```
procedure Inserta_Delante
   (El_Elemento    : in Elemento;
    La_Posicion    : in Posicion;
    La_Lista       : in out Lista)
is
    Temp:Posicion;
begin
    if La_Posicion=null or else La_Posicion.Proximo=null
    then
       raise Posicion_Incorrecta;
    else
       Temp:=La_Posicion.Proximo;
       La_Posicion.Proximo:=new Tipo_Celda;
       La_Posicion.Proximo.Contenido := El_Elemento;
       La_Posicion.Proximo.Proximo:=Temp;
    end if;
end Inserta_Delante;
```

Implementación (cont.)

Diagrama de elimina:



Elimina

UNIVERSIDAD DE CANTABRIA

```
procedure Elimina
   (La_Posicion: in Posicion;
    La_Lista   : in out Lista;
    El_Elemento: out Elemento)
is
begin
    if La_Posicion=null or else La_Posicion.Proximo=null then
       raise Posicion_Incorrecta;
    else
       El_Elemento:=La_Posicion.Proximo.Contenido;
       La_Posicion.Proximo:=La_Posicion.Proximo.Proximo;
    end if;
end Elimina;
```

Localiza

UNIVERSIDAD DE CANTABRIA

```
function Localiza
   (El_Elemento: in Elemento;
    La_Lista   : in Lista)
  return Posicion
is
    P : Posicion:= La_Lista.Principio;
begin
    if P/=null then
       while P.Proximo/=null loop
          if P.Proximo.Contenido = El_Elemento then
             return P;
          else
             P:=P.Proximo;
          end if;
       end loop;
    end if;
    return Posicion_Nula; -- No encontrado
end Localiza;
```

Primera_Pos

UNIVERSIDAD DE CANTABRIA

```
function Primera_Pos
   (La_Lista : in Lista)
   return Posicion
is
begin
    if Esta_Vacia(La_Lista) then
       return Posicion_Nula;
    else
       return La_Lista.Principio;
    end if;
end Primera_Pos;
```

Elemento_De

```
function Elemento_De
   (La_Posicion : in Posicion;
    La_Lista : in Lista)
  return Elemento
is
begin
   if La_Posicion=null or else La_Posicion.Proximo=null then
      raise Posicion_Incorrecta;
   else
      return La_Posicion.Proximo.Contenido;
   end if;
end Elemento_De;
```

2.3 Comparación de los métodos

	array simple	lista enlazada con punteros
Tamaño máximo lista	fijo	dinámico
Tiempo de Inserta_Al_Principio	O(n)	O(1)
Tiempo de Inserta_Al_Final	O(1)	O(n)
Tiempo de Inserta_Delante	O(n)	O(1)
Tiempo de Elimina	O(n)	O(1)
Tiempo de Localiza	O(n)	O(n)
Tiempo de Primera_pos	O(1)	O(1)
Tiempo de Ultima_pos	O(1)	O(n)
Tiempo de Siguiente	O(1)	O(1)
Tiempo de Anterior	O(1)	O(n)

Notas:

UNIVERSIDAD DE CANTABRIA

1. La implementación mediante arrays requiere especificar el tamaño máximo de la lista en tiempo de compilación.

2. La implementación mediante arrays puede malgastar espacio, ya que utiliza siempre el máximo tamaño declarado para la lista. Sin embargo, en las listas encadenadas es preciso reservar espacio para los punteros.

3. Algunas operaciones llevan más tiempo en unas implementaciones que en otras. Por ejemplo INSERTA_DELANTE y ELIMINA son mucho más rápidas en las listas enlazadas, para el caso medio. Sin embargo las operaciones ANTERIOR y ULTIMA_POS son más lentas. La localización de un elemento o recorrer la lista (con PRIMERA_POS y SIGUIENTE) son iguales en ambas implementaciones.

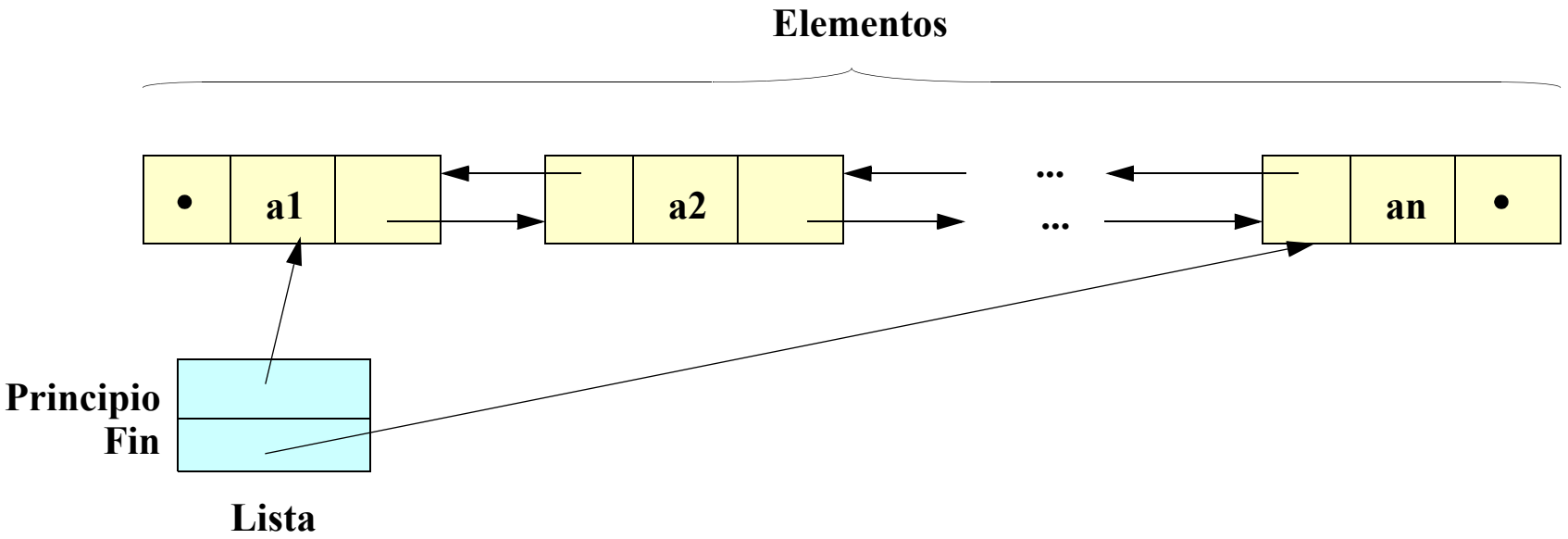
4. Para aplicaciones de tiempo real no se suele utilizar creación dinámica de variables, ya que habitualmente tiene un tiempo de respuesta poco predecible. Generalmente se prefieren implementaciones con arrays o cursores.

2.4. Listas doblemente enlazadas

UNIVERSIDAD DE CANTABRIA

En las listas enlazadas simples, las operaciones `Ultima_Pos` y `Anterior` son muy costosas (O(*n*))

Para evitar este problema se pueden hacer listas doblemente enlazadas:



Notas:

UNIVERSIDAD DE CANTABRIA

En las listas encadenadas mediante punteros las operaciones en las que se accedía al elemento de la lista previo a uno dado eran muy costosas. Para una aplicación en la que es preciso recorrerla hacia adelante y hacia atrás se pueden utilizar listas doblemente encadenadas:

En esta estructura de datos no son necesarias las cabeceras vacías. Además, la posición de un elemento es un puntero a ese mismo elemento (en las listas enlazadas se apuntaba al elemento anterior, por eficiencia).

Estructura de datos privada

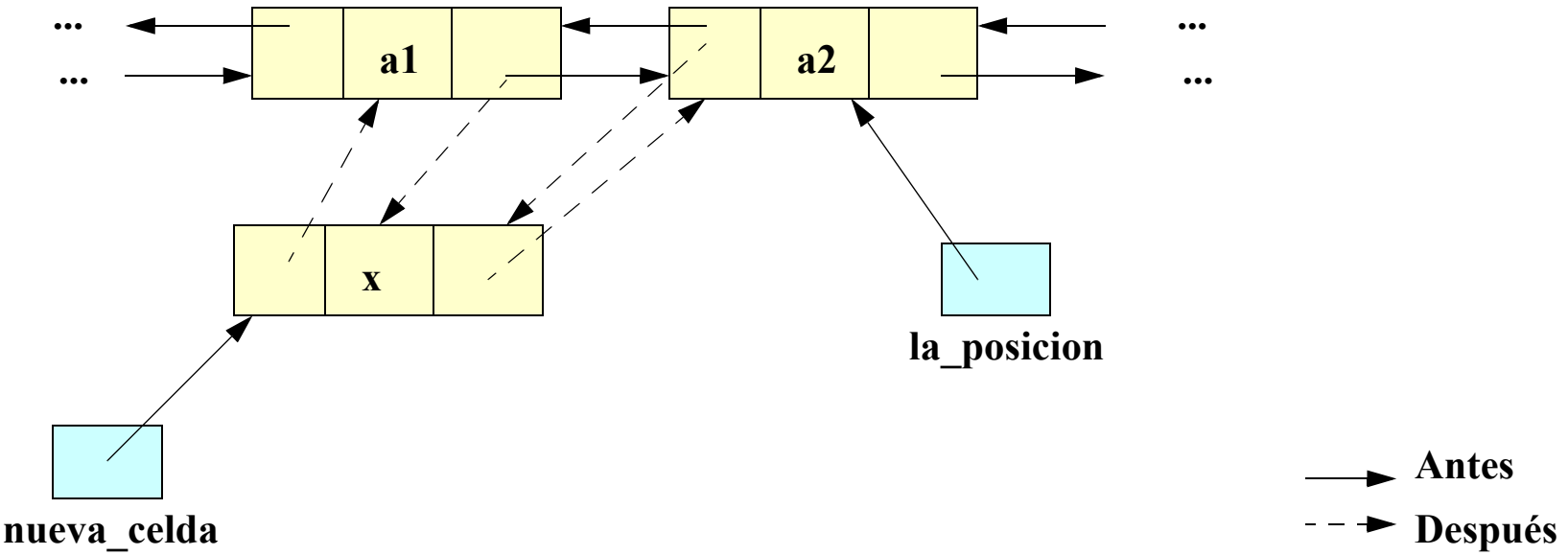
UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Listas;
generic
   type Elemento is private;
   with function "="(E1,E2 : Elemento) return Boolean;
package Listas_Doblemente_Enlazadas is
   -- Todo igual que antes
private
   type Tipo_Celda;
   type Posicion is access Tipo_Celda;
   Posicion_Nula : constant Posicion := null;

   type Tipo_Celda is record
      Contenido       : Elemento;
      Proximo,Previo : Posicion;
   end record;

   type Lista is record
      Principio,Fin : Posicion;
   end record;
end Listas_Doblemente_Enlazadas;
```

Operaciones con listas doblemente enlazadas

UNIVERSIDAD DE CANTABRIA

Insertar un elemento delante de otro:



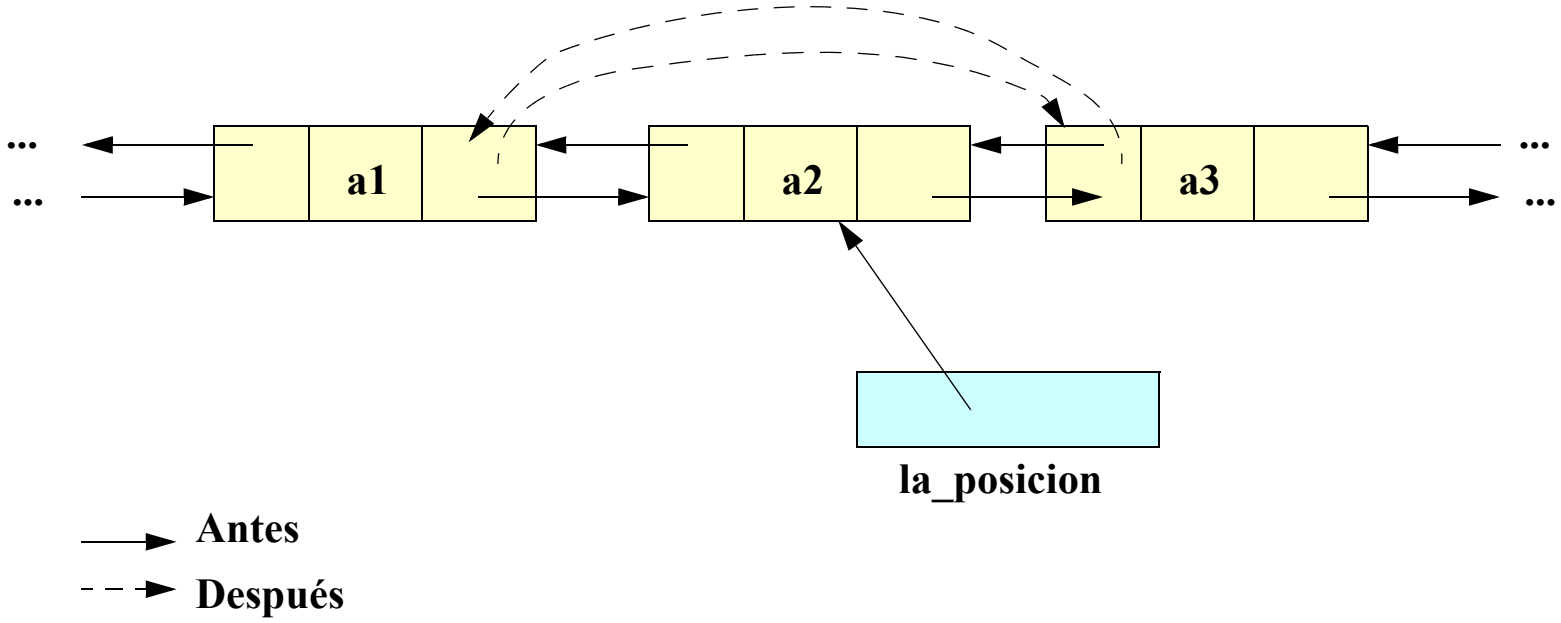
Inserta delante

UNIVERSIDAD DE CANTABRIA

```
procedure Inserta_Delante
   (El_Elemento : in Elemento;
    La_Posicion : in Posicion;
    La_Lista    : in out Lista)
is
  Nueva_Celda:Posicion:=new Tipo_Celda'(El_Elemento,null,null);
begin
    if La_Posicion=null or La_Lista.Principio=null then
       raise Posicion_Incorrecta;
    elsif La_Posicion=La_Lista.Principio then -- al principio
       La_Lista.Principio.Previo:=Nueva_Celda;
       Nueva_Celda.Proximo:=La_Lista.Principio;
       La_Lista.Principio:=Nueva_Celda;
    else
       Nueva_Celda.Proximo:=La_Posicion;
       Nueva_Celda.Previo:=La_Posicion.Previo;
       La_Posicion.Previo.Proximo:=Nueva_Celda;
       La_Posicion.Previo:=Nueva_Celda;
    end if;
end Inserta_Delante;
```

Operaciones (cont.)

Eliminación de un elemento:



Elimina

UNIVERSIDAD DE CANTABRIA

```
procedure Elimina
   (La_Posicion: in Posicion;
    La_Lista   : in out Lista;
    El_Elemento: out Elemento) is
begin
    if La_Posicion=null then raise Posicion_Incorrecta;
    else
       El_Elemento:=La_Posicion.Contenido;
       if La_Posicion.Previo /= null then --No es la primera
          La_Posicion.Previo.Proximo:=La_Posicion.Proximo;
       else
          La_Lista.Principio:=La_Posicion.Proximo;
       end if;
       if La_Posicion.Proximo /= null then --No es ultima
          La_Posicion.Proximo.Previo:=La_Posicion.Previo;
       else
          La_Lista.Fin:=La_Posicion.Previo;
       end if;
    end if;
end Elimina;
```

2.5. Uso de las listas

UNIVERSIDAD DE CANTABRIA

Para usar una de las implementaciones del paquete listas hay que instanciarlo

A continuación se muestra un ejemplo de:

- **instanciación para números enteros**
- **procedimiento para recorrer los elementos de la lista y mostrarlos en pantalla**

```
with Listas_Simplemente_Enlazadas,Ada.Text_IO;
use Ada.Text_IO;

procedure Prueba_Listas_Simplemente_Enlazadas is

   package Listas is new
      Listas_Simplemente_Enlazadas(Integer,"=");
   use Listas;
   ...
```

Recorre los elementos de una lista

UNIVERSIDAD DE CANTABRIA

```
procedure Recorre (L : in Lista) is
   Pos : Posicion;
   I : INTEGER:=1;
begin
   Put_Line("Recorre la lista");
   if Esta_Vacia(L) then
      Put_Line("La lista esta vacia");
   else
      Put_Line("Principio lista");
      Pos:=Primera_Pos(L);
      while Pos/=Posicion_Nula loop
         Put_Line("Elemento : " & Integer'Image(I) &
                              " = " & Integer'Image(Elemento_De(Pos,L)));
         Pos:=Siguiente(Pos,L);
         I:=I+1;
      end loop;
      Put_Line("Final lista");
   end if;
end Recorre;
```

Ejemplo de uso de operaciones de las listas

UNIVERSIDAD DE CANTABRIA

```
   -- variables del programa principal
   L1 : Lista;
   Pos : Posicion;
   Elem, I : Integer;
begin
   Haz_Nula(L1);
   Inserta_Al_Principio(I,L1);
   Pos:=Primera_Pos(L1);
   Elimina (Pos,L1,Elem);
   Inserta_Al_Final(200,L1);
   Inserta_Delante(300,Primera_Pos(L1),L1);
   Recorre(L1);
   Pos:=Localiza(102,L1);
   if Pos=Posicion_Nula then
      Put_Line("No encontrado");
   else
      Elem:=Elemento_De(Pos,L1);
      Put_Line("encontrado : "&Integer'Image(Elem));
   end if;
end Prueba_Listas_Simplemente_Enlazadas;
```

2.6. El paquete `Containers.Doubly_Linked_Lists`

UNIVERSIDAD DE CANTABRIA

Este paquete genérico proporciona los tipos privados `List` y `Cursor` para implementar listas doblemente enlazadas.:

- estos tipos son equivalentes a `Lista` y `Posicion` del ADT que hemos visto anteriormente

Un cursor designa a un nodo particular dentro de una lista y tiene sentido su uso siempre que el nodo referido siga perteneciendo a la lista.

Las siguientes páginas muestran la especificación de este contenedor, destacando los tipos y las operaciones que se corresponden con las vistas en el ADT Lista visto:

- para consultar el funcionamiento concreto de las operaciones ver el Manual de Referencia (A.18.3)

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
   pragma Preelaborate(Doubly_Linked_Lists);

   type List is tagged private; -- tipo Lista
   pragma Preelaborable_Initialization(List);

   type Cursor is private; -- tipo Posicion
   pragma Preelaborable_Initialization(Cursor);

   Empty_List : constant List;

   No_Element : constant Cursor; -- similar a Posicion_Nula

   function "=" (Left, Right : List) return Boolean;

   function Length (Container : List) return Count_Type;
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
function Is_Empty (Container : List) return Boolean;
-- Esta_Vacia

procedure Clear (Container : in out List);
-- Haz_Nula

function Element (Position : Cursor) return Element_Type;
-- Elemento_De

procedure Replace_Element (Container : in out List;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type);
procedure Query_Element
   (Position : in Cursor;
    Process  : not null access procedure (Element : in Element_Type));
procedure Update_Element
   (Container : in out List;
    Position  : in     Cursor;
    Process   : not null access procedure
                      (Element : in out Element_Type));
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Move (Target : in out List;
                Source : in out List);

procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  : out    Cursor;
                  Count     : in     Count_Type := 1);

procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  : out    Cursor;
                  Count     : in     Count_Type := 1);
-- Operaciones para Inserta_Delante con más opciones
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Prepend (Container : in out List;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1);
-- Inserta_Al_Principio

procedure Append (Container : in out List;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1);
-- Inserta_Al_Final

procedure Delete (Container : in out List;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1);

procedure Delete_First (Container : in out List;
                        Count     : in     Count_Type := 1);

procedure Delete_Last (Container : in out List;
                       Count     : in     Count_Type := 1);
-- Distintas versiones de Elimina
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Reverse_Elements (Container : in out List);

procedure Swap (Container : in out List;
                I, J      : in     Cursor);

procedure Swap_Links (Container : in out List;
                      I, J      : in     Cursor);

procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List);

procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor);

procedure Splice (Container: in out List;
                  Before   : in     Cursor;
                  Position : in     Cursor);
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
function First (Container : List) return Cursor;
-- Primera_Pos

function First_Element (Container : List)
   return Element_Type;

function Last (Container : List) return Cursor;
-- Ultima_Pos

function Last_Element (Container : List)
   return Element_Type;

function Next (Position : Cursor) return Cursor;
-- Siguiente

function Previous (Position : Cursor) return Cursor;
-- Anterior

procedure Next (Position : in out Cursor);
-- Otra versión de Siguiente como procedimiento
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Previous (Position : in out Cursor);
-- Otra versión de Anterior como procedimiento

function Find (Container : List;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor;

function Reverse_Find (Container : List;
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor;
-- Versiones de Localiza

function Contains (Container : List;
                   Item      : Element_Type) return Boolean;

function Has_Element (Position : Cursor) return Boolean;
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Iterate
   (Container : in List;
    Process   : not null access procedure (Position : in Cursor));

procedure Reverse_Iterate
   (Container : in List;
    Process   : not null access procedure (Position : in Cursor));
```

Containers.Doubly_Linked_Lists (cont.)

UNIVERSIDAD DE CANTABRIA

```
   generic
      with function "<" (Left, Right : Element_Type)
         return Boolean is <>;
   package Generic_Sorting is

      function Is_Sorted (Container : List) return Boolean;

      procedure Sort (Container : in out List);

      procedure Merge (Target  : in out List;
                       Source  : in out List);

   end Generic_Sorting;

private

   ... -- not specified by the language

end Ada.Containers.Doubly_Linked_Lists;
```

3. Pilas (Stacks)

UNIVERSIDAD DE CANTABRIA

Una pila es una lista especial en la que todos los elementos se insertan o extraen por un extremo de la lista (LIFO)

Operaciones:

operación	in	out	in out	errores
Haz_Nula			la_pila	
Inserta	el_elemento		la_pila	no_cabe
Extrae		el_elemento	la_pila	no_hay
Esta_Vacia	la_pila	booleano		

Notas:

UNIVERSIDAD DE CANTABRIA

Un stack o pila es una lista de un tipo especial, en la que todas las inserciones o eliminaciones de elementos tiene lugar en un extremo. En una pila el último elemento que entra es el primero que sale (LIFO).

Las operaciones básicas que se realizan sobre un stack suelen ser las siguientes:

- Haz_Nula(P): Inicializar una pila
- Inserta(x,P): Extraer un elemento de la pila
- Extrae(x,P): Introducir el elemento x en la pila
- Esta_Vacia(P) : BOOLEAN: Comprueba si la pila está vacía.

La especificación en Ada del ADT pila aparece en la transparencia siguiente.

3.1. Especificación en Ada de las pilas

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Pilas;
generic
   type Elemento is private;
   Max : Positive :=100;
package Pilas is

   type Pila is private;

   No_Hay : exception
     renames Excepciones_Pilas.No_Hay;
   No_Cabe : exception
     renames Excepciones_Pilas.No_Cabe;

   procedure Haz_Nula (La_Pila : in out Pila);

   procedure Inserta
     (El_Elemento : in  Elemento;
      La_Pila     : in out Pila);
   -- puede elevar No_Cabe
```

Especificación de las pilas (cont.)

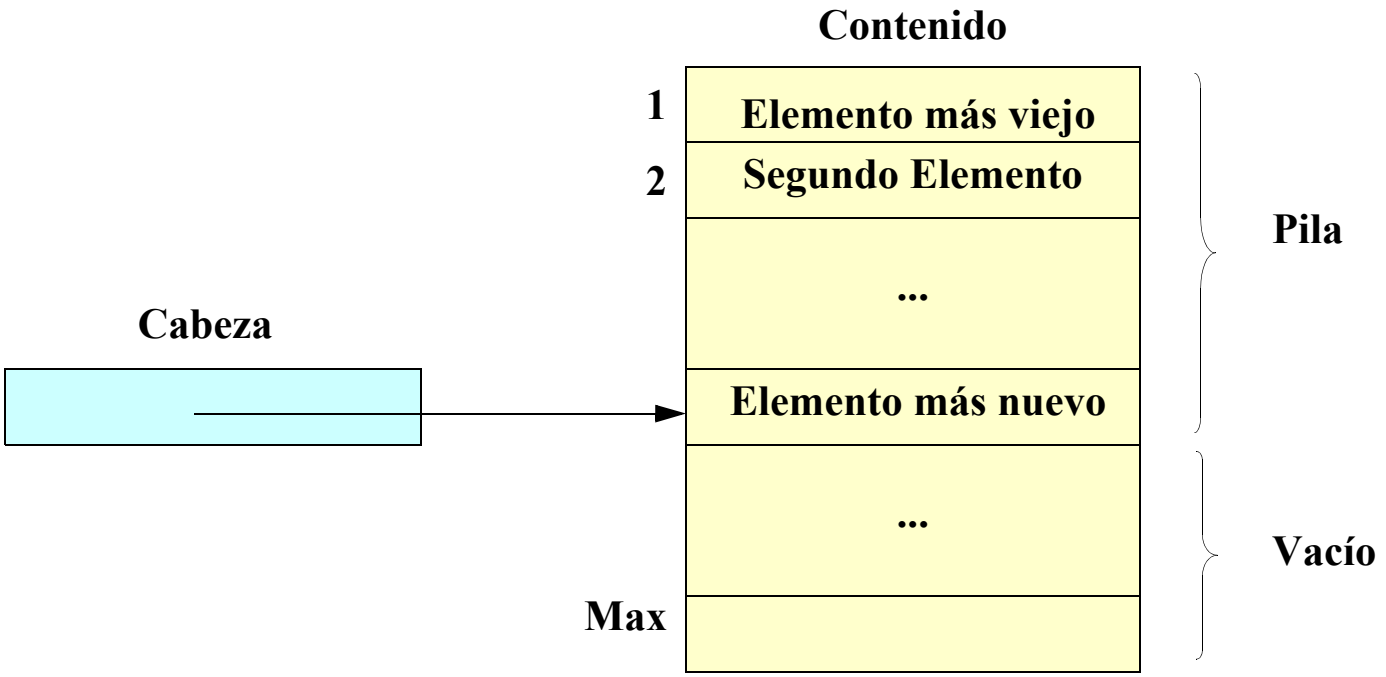
UNIVERSIDAD DE CANTABRIA

```
   procedure Extrae
      (El_Elemento : out Elemento;
       La_Pila     : in out Pila);
   -- puede elevar No_Hay

   function Esta_Vacia
      (La_Pila : in Pila)
      return Boolean;

private
   ...
end Pilas;
```

3.2 Implementación con array simple

Los elementos se guardan en un array

El extremo activo se guarda en la variable `Cabeza`



Notas:

UNIVERSIDAD DE CANTABRIA

Aunque una pila se puede implementar con una lista, es posible simplificar la implementación, ya que las operaciones con las pilas son más sencillas. Existen aún así diversas formas de implementarlas.

- Implementación mediante array simple: es la forma más simple. el espacio ocupado es fijo, pero la implementación es eficiente. todas las operaciones son O(1)
- Implementación mediante lista enlazada: los elementos se insertan y extraen por el principio de la lista; no hace falta elemento cabecera; el espacio ocupado es dinámico; todas las operaciones son O(1)

4. Colas

UNIVERSIDAD DE CANTABRIA

Una cola es otra lista especial en la que todos los elementos se insertan por un extremo y se extraen por el otro extremo de la lista (FIFO)

Operaciones:

operación	in	out	in out	errores
Haz_Nula			la_cola	
Inserta	el_elemento		la_cola	no_cabe
Extrae		el_elemento	la_cola	no_hay
Esta_Vacia	la_cola	booleano		

Notas:

UNIVERSIDAD DE CANTABRIA

Una cola es otro tipo especial de lista en la que los elementos se insertan por un extremo y se extraen por el otro. En esta estructura el primer elemento en llegar es el primero en salir (FIFO).

Las operaciones básicas con la cola son muy similares a las de los stacks:

- Haz_Nula(C) : Inicializar una cola
- Extrae(x,C): Extraer un elemento de la cola
- Inserta(x,C): Introducir el elemento x en la cola
- Esta_Vacia(C) : BOOLEAN: Comprueba si la cola vacía

La especificación en Ada del ADT cola aparece en la transparencia siguiente.

4.1. Especificación en Ada de las colas

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Colas;
generic
   type Elemento is private;
package Colas is
   type Cola is private;

   No_Hay : exception
     renames Excepciones_Colas.No_Hay;
   No_Cabe : exception
     renames Excepciones_Colas.No_Cabe;

   procedure Haz_Nula (La_Cola : in out Cola);

   procedure Inserta
     (El_Elemento : in  Elemento;
      La_Cola     : in out Cola);
   -- puede elevar No_Cabe
```

Especificación de las colas (cont.)

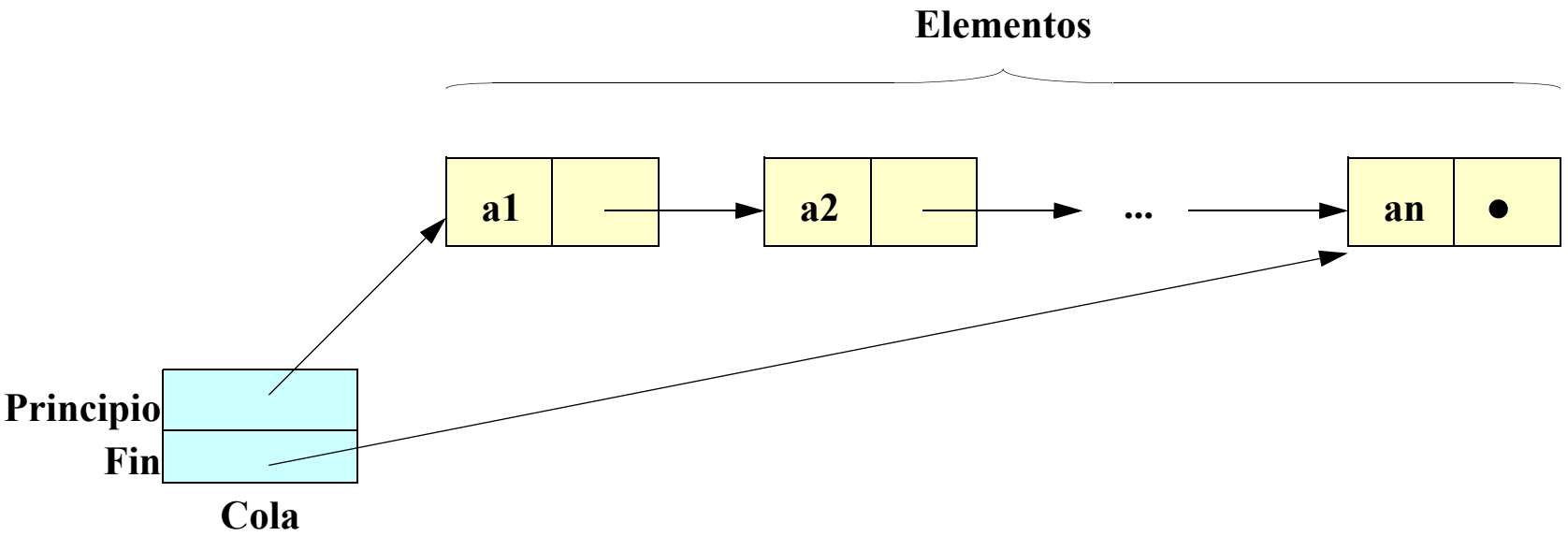
```
   procedure Extrae
      (El_Elemento : out Elemento;
       La_Cola     : in out Cola);
   -- puede elevar No_Hay

   function Esta_Vacia
      (La_Cola : in Cola)
      return Boolean;

private
   ...
end Colas;
```

4.2. Implementación de colas mediante punteros

UNIVERSIDAD DE CANTABRIA

Los elementos se guardan en una lista enlazada

La cola tiene un puntero al principio y otro al final; los elementos se insertan por el final y se extraen por el principio



Notas:

UNIVERSIDAD DE CANTABRIA

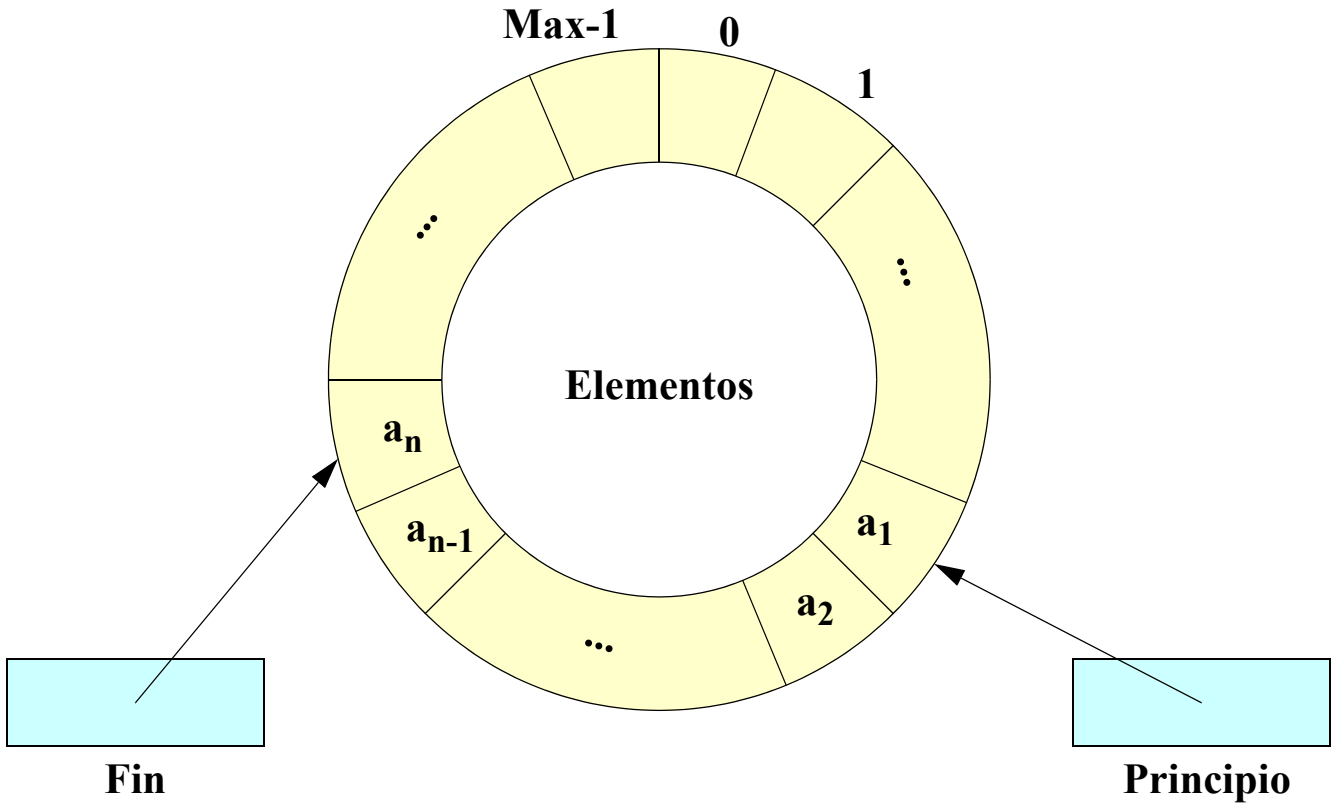
La implementación de las colas se puede hacer con cualquiera de las implementaciones existentes para las listas. Sin embargo existen implementaciones especiales que toman ventaja de la estructura de la cola. Las más usuales son:

- Implementación mediante punteros: en forma de lista encadenada, con punteros apuntando al principio y al final de la cola; no se necesita celda cabecera; las operaciones son O(1).
- Implementación mediante array circular: en forma de array circular (es decir suponiendo que a continuación de la última casilla viene la primera; se utilizan dos cursores, para apuntar al principio y al final de la cola; se deja una casilla vacía; las operaciones son todas O(1).

A continuación veremos ambas implementaciones, comenzando con la de punteros.

Puesto que las operaciones que se realizan con la cola afectan a los extremos, se puede disponer de dos punteros que apunten al principio y final de la cola, en la forma que se indica en la figura superior. La declaración en Ada de esta estructura es:

4.3. Implementación de colas mediante arrays circulares



Notas:

UNIVERSIDAD DE CANTABRIA

Se puede realizar una implementación de la cola en un array de una forma muy eficiente, si se considera el array como una estructura circular, que se puede recorrer de forma que si se llega al final se comienza de nuevo por el principio, tal como aparece en la figura de arriba.

En esta implementación la cola queda definida por un array y dos cursores que apuntan al principio y final de la lista.

5. Vectores

UNIVERSIDAD DE CANTABRIA

Un vector es una secuencia de objetos ordenados, en la que existe acceso aleatorio a los mismos

A diferencia de los arrays, permite cambiar su tamaño

Operaciones más importantes

- obtener y cambiar el tamaño
- obtener o modificar elementos en cualquier posición
- borrar elementos y saber si están borrados
- etc.

La posición de los objetos es del tipo entero (positivo) `Indice`

Notas:

UNIVERSIDAD DE CANTABRIA

Los Vectores son secuencias de objetos ordenados que permiten el acceso aleatorio a cualquiera de sus elementos. Son una generalización del array, con la diferencia fundamental de poder cambiar de tamaño. El tamaño (longitud) se puede consultar y cambiar.

El índice con el que nos referimos a una casilla es un subtipo de Positive.

Todos los elementos del vector son del mismo tipo de datos, que llamaremos tipo “Elemento”, y que definiremos como parámetro genérico.

Las "casillas" de un vector pueden contener o no un valor. Inicialmente están sin valor (lo que denominamos "borrada"). Cuando se modifican, se inserta un valor en ellas. Más adelante se puede borrar ese valor, volviendo a dejar la casilla en el estado "borrada".

Operaciones de los vectores

UNIVERSIDAD DE CANTABRIA

operación	in	out	in out	errores
Haz_Nulo			El_Vector	
Longitud	El_Vector	Longitud		
Cambia_ Longitud	Nueva_ Longitud		El_Vector	
Modifica	El_Elemento El_Indice		El_Vector	Indice_ Incorrecto
Elemento_De	El_Indice El_Vector	El_Elemento		Indice_ Incorrecto, Inexistente

Notas:

UNIVERSIDAD DE CANTABRIA

Operaciones de los vectores:

- *Haz_Nulo*: Inicializa el vector poniéndole un tamaño igual al parámetro genérico Longitud_Inicial, y dejando todas las casillas en estado "Borrado"
- *Longitud*: Devuelve la longitud actual del vector
- *Cambia_Longitud*: Cambia la longitud del vector al nuevo valor. Puede ser una operación lenta, que implique copiar los elementos a otro array más grande.
- *Modifica*: Modifica el elemento de la casilla indicada por El_Indice al nuevo valor indicado por El_Elemento. La casilla para a contener un valor y, por tanto, deja el estado "Borrada" si es que estaba en él. Si El_Indice es incorrecto, falla.
- *Elemento_De*: Devuelve el valor almacenado en la casilla El_Indice. Si no tiene valor o si El_Indice es incorrecto, falla.
- *Borra*: Deja la casilla indicada por El_Indice en estado "borrado", independientemente de cómo estuviese en ese momento. Si El_Indice es incorrecto, falla.
- *Esta_Borrado*: Devuelve true si la casilla El_Indice está borrada, y false en caso contrario. Si El_Indice es incorrecto, falla.

Operaciones de los vectores (cont.)

operación	in	out	in out	errores
Borra	El_Indice		El_Vector	Indice_ Incorrecto
Esta_Borrado	El_Indice El_Vector	Booleano		Indice_ Incorrecto

5.1. Especificación de los vectores en Ada (1/3)

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Vectores;
generic
   Longitud_Inicial : Positive;
   type Elemento is private;

package Vectores is

   subtype Indice is Positive;

   type Vector is private;

   Indice_Incorrecto : exception
     renames Excepciones_Vectores.Indice_Incorrecto;
   Inexistente  : exception
     renames Excepciones_Vectores.Inexistente;

   procedure Haz_Nulo (El_Vector : in out Vector);
```

Especificación de los vectores en Ada (2/3)

UNIVERSIDAD DE CANTABRIA

```
function Longitud (El_Vector : Vector) return Positive;

procedure Cambia_Longitud
   (El_Vector : in out Vector;
    Nueva_Longitud : in Positive);

procedure Modifica
   (El_Elemento : in Elemento;
    El_Indice   : in Indice;
    El_Vector   : in out Vector);
-- puede elevar Indice_Incorrecto

function Elemento_De
   (El_Indice : in Indice;
    El_Vector : in Vector)
  return Elemento;
-- puede elevar Indice_Incorrecto o Inexistente
```

Especificación de los vectores en Ada (3/3)

UNIVERSIDAD DE CANTABRIA

```
   procedure Borra
      (El_Indice   : in Indice;
       El_Vector   : in out Vector);
   -- puede elevar Indice_Incorrecto

   function Esta_Borrado
      (El_Indice : in Indice;
       El_Vector : in Vector)
     return Boolean;
   -- puede elevar Indice_Incorrecto

private
   ...
end Vectores;
```

5.2. Implementación de vectores con arrays dinámicos

UNIVERSIDAD DE CANTABRIA

El vector es un puntero a un array creado en memoria dinámica:

- de este modo, si es preciso cambiar el tamaño se crea un nuevo array y se mueven los elementos del viejo al nuevo

Cada casilla tiene un booleano que indica si está borrada o no

6. Conjuntos

UNIVERSIDAD DE CANTABRIA

Un conjunto es una secuencia de objetos no ordenados y no repetidos

Operaciones más importantes

- **añadir o borrar elemento**
- **recorrer los elementos**
- **pertenencia**
- **operaciones con dos conjuntos: unión, intersección, diferencia, inclusión**

Notas:

UNIVERSIDAD DE CANTABRIA

Los Conjuntos son secuencias de objetos no ordenados y no repetidos

Todos los elementos del conjunto son del mismo tipo de datos, que llamaremos tipo “Elemento”, y que definiremos como parámetro genérico.

Las dos operaciones más importantes para manipular un conjunto son la inserción y extracción de elementos. Si insertamos un elemento que ya existe, el conjunto queda igual. Lo mismo si extraemos un elemento que no existe.

Para usar el conjunto las operaciones más importantes son la pertenencia (saber si un elemento pertenece o no al conjunto) y recorrer los elementos. En este caso lo haremos con una operación de iteración, a la que le pasamos como parámetro genérico el procedimiento a ejecutar para cada elemento del conjunto.

También añadiremos operaciones para operar con dos conjuntos: unión, intersección, diferencia, e inclusión

Operaciones básicas de los conjuntos

UNIVERSIDAD DE CANTABRIA

operación	in	out	in out	errores
Vacio		Conjunto		
Inserta	Elemento		Conjunto	
Extrae	Elemento		Conjunto	
Pertenece	Elemento Conjunto	Booleano		
Itera	Procedimiento Conjunto			

Notas:

UNIVERSIDAD DE CANTABRIA

Operaciones básicas de los conjuntos:

- *Vacio*: Retorna un conjunto vacío, que no tiene elementos
- *Inserta*: Inserta el elemento indicado en el conjunto. Si el elemento ya estaba en el conjunto, no hace nada
- *Extrae*: Extrae el elemento indicado del conjunto. Si el elemento no pertenecía al conjunto, no hace nada
- *Pertenece*: Devuelve un booleano que indica si el elemento indicado pertenece o no al conjunto
- *Itera*: Recorre todos los elementos del conjunto y para cada uno de ellos ejecuta el procedimiento que se le pasa como parámetro
- *Intersección*: Devuelve un conjunto con los elementos comunes a ambos conjuntos A y B
- *Unión*: Devuelve un conjunto con todos los elementos de los conjuntos A y B
- *Diferencia*: Devuelve un conjunto con los elementos de A que no están en B
- *Inclusión*: Devuelve un booleano que indica si todos los elementos de A están también en B, o no.

Operaciones con dos conjuntos

UNIVERSIDAD DE CANTABRIA

operación	in	out	in out	errores
Intersección	A,B : Conjunto	Conjunto		
Unión	A,B : Conjunto	Conjunto		
Diferencia	A,B : Conjunto	Conjunto		
Inclusion	A,B : Conjunto	Booleano		

6.1. Especificación Ada de los conjuntos

UNIVERSIDAD DE CANTABRIA

```
generic
   type Elemento is private;
   with function "=" (E1,E2 : Elemento) return Boolean;
package Conjuntos is
   type Conjunto is private;

   function Vacio return Conjunto;          -- conjunto vacio

   procedure Inserta (E : Elemento;    -- anadir elemento
                      C : in out Conjunto);

   procedure Extrae  (E : Elemento;    -- extrae elemento
                      C : in out Conjunto);

   -- pertenencia
   function "<" (E: Elemento; C: Conjunto) return Boolean;

   generic
      with procedure Procesa (E : Elemento);
   procedure Itera (C : Conjunto);          -- Recorre los elementos
```

Especificación Ada de los conjuntos (cont.)

UNIVERSIDAD DE CANTABRIA

```
   function "+"      (A,B : Conjunto)          -- union
                      return Conjunto;

   function "-"      (A,B : Conjunto)          -- diferencia
                      return Conjunto;

   function "*"      (A,B : Conjunto)          -- interseccion
                      return Conjunto;

   function "<"      (A,B : Conjunto)          -- inclusion
                      return Boolean;

private
...
end Conjuntos;
```

6.2. Implementación de conjuntos mediante listas

UNIVERSIDAD DE CANTABRIA

En otro capítulo vimos la implementación de conjuntos mediante arrays de booleanos

- para conjuntos de elementos discretos
- cuando el número de posibles elementos es pequeño

Si la cantidad de valores es alta o el tipo no es discreto:

- implementación mediante listas de elementos
  - se puede usar el ADT *Listas*
- para más eficiencia se pueden usar las técnicas de los mapas con troceado (hash), que se verán más adelante

7. Mapas

UNIVERSIDAD DE CANTABRIA

Un "mapa" es una función de unos elementos de un tipo `origen`, en otros elementos de un tipo `destino`

Las operaciones asociadas al mapa son:

operación	in	out	in out	errores
Haz_Nulo			el_mapa	
Asigna	el_origen el_destino		el_mapa	no_cabe
Elimina	el_origen		el_mapa	
Calcula	el_origen el_mapa	el_destino existe		

Notas:

UNIVERSIDAD DE CANTABRIA

Un 'mapa' M es una función de unos elementos de un tipo origen, en otros elementos de otro (o el mismo) tipo denominado destino. Es una generalización del concepto de array para manejar índices (orígenes) no necesariamente discretos. Se representa como M(origen)=destino.

En ocasiones el mapa se puede indicar a través de una expresión matemática. Sin embargo en otras ocasiones es preciso almacenar para cada posible valor d el valor de M(origen).

En este caso se puede utilizar el tipo abstracto de datos denominado 'mapa', para el que se definen las siguientes operaciones:

- Haz_Nulo (M): Crear el mapa M haciéndolo nulo, es decir sin ninguna relación almacenada
- Asigna (M,o,d): Define M(o) igual a d; si M(o) ha sido asignado previamente, pierde su valor anterior
- Elimina(M,o): Elimina la actual relación de o (si existe) en el mapa M
- Calcula(M,o,d,existe): Devuelve el destino asociado (d) y existe=True si existe M(o) definido, y existe=False en caso contrario.

Implementaciones de mapas

UNIVERSIDAD DE CANTABRIA

Las implementaciones de mapas son muy diferentes según el tipo `origen`

- si es discreto con un número de valores pequeño:
  - se puede hacer la implementación con un array
  - para marcar las casillas vacías, se puede tener un booleano en cada una, o usar un valor especial del `destino` (*indefinido*)
- en caso contrario:
  - se puede implementar el mapa con una lista de relaciones (parejas `origen`-`destino`)
  - se puede usar una tabla de troceado (*hash*), más eficiente

Especificación de mapas discretos

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Mapas;

generic
   type Origen is (<>);
   type Destino is private;
   Indefinido : Destino;
package Mapas_Discretos is

   type Mapa is private;

   No_Cabe : exception renames Excepciones_Mapas.No_Cabe;

   procedure Haz_Nulo
     (El_Mapa : in out Mapa);

   procedure Asigna
     (El_Origen  : in Origen;
      El_Destino : in Destino;
      El_Mapa    : in out Mapa);
   -- puede elevar No_Cabe
```

Especificación de mapas discretos (cont.)

UNIVERSIDAD DE CANTABRIA

```
   procedure Elimina
      (El_Origen : in Origen;
       El_Mapa   : in out Mapa);

   procedure Calcula
      (El_Origen  : in Origen;
       El_Mapa    : in Mapa;
       El_Destino : out Destino;
       Existe     : out Boolean);

private
   type Mapa is array (Origen) of Destino;

end Mapas_Discretos;
```

Operaciones de los mapas discretos (1/2)

UNIVERSIDAD DE CANTABRIA

```
procedure Haz_Nulo(El_Mapa : in out Mapa) is
begin
   for I in Origen loop
      El_Mapa(I):=Indefinido;
   end loop;
end Haz_Nulo;


procedure Asigna
   (El_Origen  : in Origen;
    El_Destino : in Destino;
    El_Mapa    : in out Mapa) is
begin
   El_Mapa(El_Origen):=El_Destino;
end Asigna;
```

Operaciones de los mapas discretos (2/2)

UNIVERSIDAD DE CANTABRIA

```
procedure Elimina
   (El_Origen : in Origen;
    El_Mapa   : in out Mapa) is
begin
    El_Mapa(El_Origen):=Indefinido;
end Elimina;

procedure Calcula
   (El_Origen  : in Origen;
    El_Mapa    : in Mapa;
    El_Destino : out Destino;
    Existe     : out Boolean) is
begin
    if El_Mapa(El_Origen)=Indefinido then
       Existe:=False;
    else
       El_Destino:=El_Mapa(El_Origen);
       Existe:=True;
    end if;
end Calcula;
```

Notas:

UNIVERSIDAD DE CANTABRIA

De las implementaciones de los mapas vistas hasta ahora, la basada en un array es muy eficiente, con operaciones O(1), pero sólo sirve cuando el tipo origen es discreto y no tiene excesivos valores. La basada en listas elimina esta restricción, pero es muy poco eficiente, ya que todas las operaciones son O(n).

La implementación que presenta mejores características es la implementación mediante tablas “hash” o de “troceado”. Con esta implementación se consiguen tiempos breves, que pueden llegar a ser O(1), para asignar, eliminar y calcular.

La implementación se basa en encontrar una función “hash” que permite convertir un dato del tipo origen en otro discreto de un tipo que denominaremos llave. La función hash debe ser rápida.

Una vez encontrada la llave, los elementos se guardan en un array o tabla cuyo índice es del tipo llave.

El mayor problema que nos encontramos es la resolución de las colisiones que ocurren cuando dos datos del tipo origen tienen la misma llave. Existen dos soluciones a este problema, que veremos a continuación. En todo caso, para reducir las colisiones, la función hash debe distribuir los datos del tipo origen de una manera muy homogénea.

Implementación mediante tablas "hash" o de troceado

UNIVERSIDAD DE CANTABRIA

Permite asignar, eliminar y calcular en tiempo breve, incluso para datos no discretos

Se debe disponer de una función para convertir un dato del tipo `origen` en un dato discreto, de un tipo llamado `llave`

```
function Hash(D : origen) return Llave;
```

Los datos se guardan en un array cuyo índice es el tipo `llave`.

El principal problema es la resolución de colisiones

- cuando dos datos tienen la misma llave

La función `Hash` debe distribuir las llaves de modo muy homogéneo

Algunas funciones de troceado

UNIVERSIDAD DE CANTABRIA

Para enteros:

- `origen mod Max`, donde `Max` es el tamaño de la tabla.
- Suma `i=0` a `n` de: `(origen /(Max/n)^i) mod (Max/n)`

Para strings, si `Cod(i)` es el código ASCII del carácter `i`:

$$Key = \left(\sum_i (Cod(i) - Cod(\mathrm{A}))\right) \bmod \mathrm{Max}$$

$$Key = \left(\sum_i (Cod(i) \cdot 32^i)\right) \bmod \mathrm{Max}$$

$$Key = \sum_i ((Key \cdot 32 + Cod(i)) \bmod \mathrm{Max})$$

Ejemplo de función de troceado para strings

UNIVERSIDAD DE CANTABRIA

```
type Llave is mod 500; -- tipo modular

function Hash (Str : String) return Llave is
   Key : Llave:=0;
begin
   for I in Str'range loop
      Key:=Key*32+Llave(Character'Pos(Str(I)));
   end loop;
   return Key;
end Hash;
```

Notas:

UNIVERSIDAD DE CANTABRIA

Cuando el origen es un número entero con una distribución uniforme, una forma sencilla de obtener una buena llave es tomando el resto de la división entera.

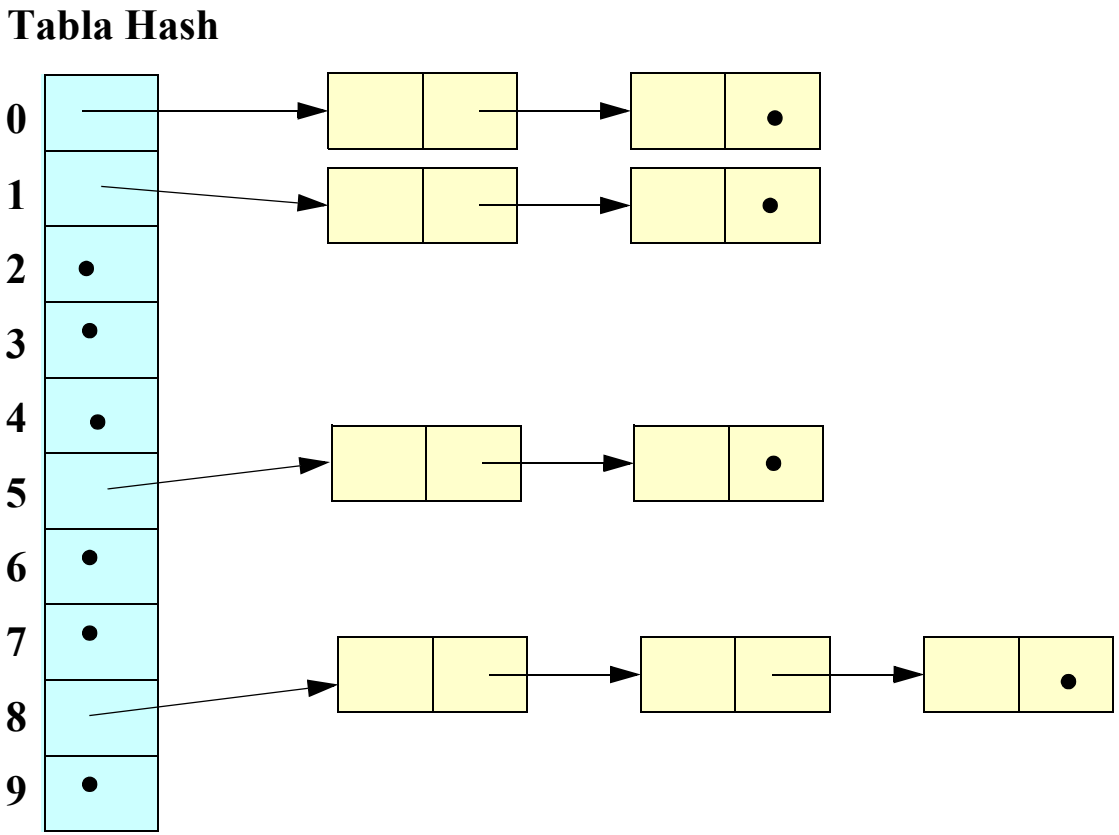
Si queremos que la parte más significativa del número también cuente, podemos trocear el número entero por sucesivas divisiones entre Max/n, calcular el módulo, y sumar todos los “trozos”. En este caso n es el número de trozos en que queremos partir el número.

Cuando el origen es un string, éste se trocea por sus caracteres. Podríamos sumar todos los códigos de los caracteres, pero este cálculo no da lugar a una cifra muy homogénea, debido a que los caracteres correspondientes a las letras son más frecuentes que otros. Esto se puede paliar restando a cada carácter el código ASCII de la letra ‘A’

Si todas las letras son mayúsculas o minúsculas, entonces una forma más homogénea consiste en multiplicar cada código ASCII por 32 elevado a i, ya que el número habitual de letras y símbolos es de 32. Se elige 32 porque la multiplicación por 32 se puede implementar con un desplazamiento, que suele ser más rápido que una multiplicación. Esta función requiere usar tipos modulares, para que si hay sobrepasamiento (overflow) en el cálculo no se eleve una excepción.

Alternativamente se puede ir calculando el módulo después de cada suma, con lo que desaparece la necesidad de usar tipos modulares, a costa de introducir más operaciones de división.

Resolución de conflictos: troceado abierto



Notas:

UNIVERSIDAD DE CANTABRIA

Un método para resolver conflictos es hacer que cada elemento de la tabla hash sea una lista enlazada simple. Cada elemento del mapa se guarda en la lista correspondiente a su llave

Este método sólo es eficiente cuando la cantidad de colisiones es pequeña. Para ello se necesita que la tabla tenga un tamaño mayor o igual que el número de elementos a almacenar y que, además, la función hash sea muy homogénea

Especificación del troceado abierto

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Mapas, Listas_Simplemente_Enlazadas;
generic
   type Llave is mod <>;
   type Origen is private;
   with function "="(O1,O2 : Origen) return Boolean;
   with function Hash(O : Origen) return Llave;
   type Destino is private;
package Mapas_Hash is

   type Mapa is private;
   No_Cabe : exception renames Excepciones_Mapas.No_Cabe;

   procedure Haz_Nulo
     (El_Mapa : in out Mapa);

   procedure Asigna
     (El_Origen  : in Origen;
      El_Destino : in Destino;
      El_Mapa    : in out Mapa);
   -- puede elevar No_Cabe
```

Especificación del troceado abierto (cont.)

UNIVERSIDAD DE CANTABRIA

```
   procedure Elimina
      (El_Origen : in Origen;
       El_Mapa   : in out Mapa);

   procedure Calcula
      (El_Origen  : in Origen;
       El_Mapa    : in Mapa;
       El_Destino : out Destino;
       Existe     : out Boolean);

private
   ...
end Mapas_Hash;
```

Resolución de conflictos: troceado cerrado

UNIVERSIDAD DE CANTABRIA

Si se detecta una colisión, se intenta calcular una nueva llave, hasta que se encuentre una vacía

$$Key_i(x) = (Hash(x) + f(i)) \bmod \text{Max}$$

La tabla debe ser bastante mayor que el número de relaciones

- al menos la mitad de las celdas vacías

La función $f(i)$ es la estrategia de resolución:

- lineal: $f(i)=i$
- cuadrática: $f(i)=i^2$ (el tamaño de la tabla debe ser primo)
- doble troceado: $f(i)=i*hash_2(x)$
  - $hash_2(x)$ nunca debe dar cero
  - ej.: R-(x mod R), siendo R primo

Notas:

UNIVERSIDAD DE CANTABRIA

Otra solución a las colisiones es el troceado cerrado. En este caso la tabla “hash” contiene las relaciones (origen-destino). Cuando se detecta una colisión, se intenta buscar otra celda vacía, buscando otra llave. Existen diversos métodos para encontrar otra llave:

- *lineal*: si la ocupación es menor que la mitad, en promedio se necesitan 2.5 búsquedas para asignar o calcular. Sin embargo, si la ocupación es mayor el número de búsquedas puede crecer mucho.
- *cuadrática*: es mejor que el lineal si los elementos tienden a agruparse en celdas contiguas. Es preciso que la ocupación sea menor que la mitad, y que el tamaño de la tabla sea un número primo.
- *doble troceado*: se utiliza una segunda función hash para calcular la nueva casilla después de una colisión. La segunda función no debe dar nunca cero (ya que en ese caso se intentaría guardar el dato en la misma casilla de antes). Una solución: R-(x mod R), siendo R primo.

Resolución de conflictos: troceado cerrado (cont)

UNIVERSIDAD DE CANTABRIA

Si borramos celdas, la resolución de conflictos falla

Debe usarse la técnica del borrado "*perezoso*"

- marcamos la celda como borrada, pero la mantenemos ocupando memoria

A veces será necesario "*recrear*" la tabla

- partir de una tabla vacía e insertar en ella todos los elementos útiles

Notas:

UNIVERSIDAD DE CANTABRIA

En esta solución no se deben borrar celdas, ya que en ese caso las búsquedas no funcionan. Para borrar una celda simplemente la marcamos como borrada, pero la mantenemos ocupando memoria. A esto se le llama borrado “perezoso”. Si la tabla cambia mucho, puede ser necesario recrear la tabla. Para ello se parte de una tabla vacía y se insertan en ella todos los elementos (exceptuando lógicamente los que habían sido borrados).

La recreación de tablas puede ser útil también si la tabla se llena mucho.

8. Árboles

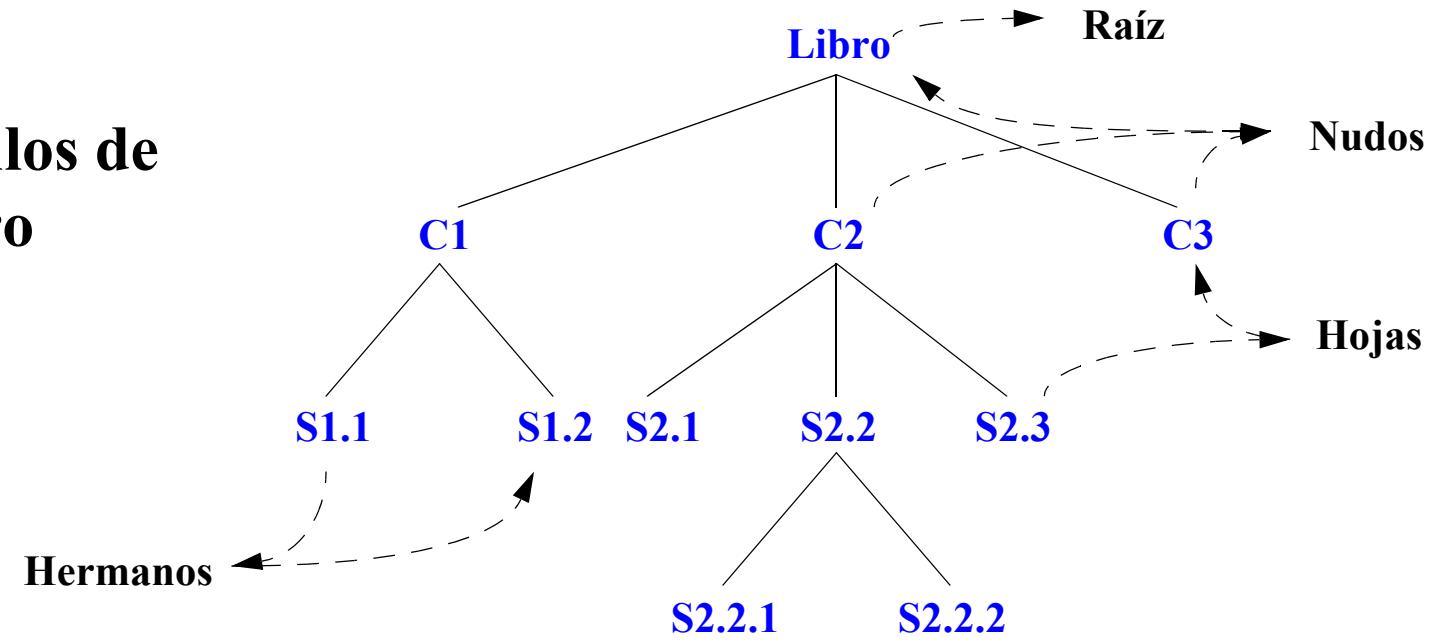
UNIVERSIDAD DE CANTABRIA

Un árbol es una estructura de datos jerarquizada

Cada dato reside en un nudo, y existen relaciones de parentesco entre nudos:

Ejemplo:

Capítulos de un libro



Notas:

UNIVERSIDAD DE CANTABRIA

Los **árboles constituyen estructuras de datos jerarquizados**, y tienen multitud de aplicaciones, como por ejemplo:

- Análisis de circuitos, Representación de estructuras de fórmulas matemáticas
- Organización de datos en bases de datos
- Representación de la estructura sintáctica en compiladores.
- En muchas otras áreas de las ciencias del computador.

Un **árbol** está constituido por una colección de elementos denominados **nudos**, uno de los cuales se distingue con el nombre **raíz**, junto con una relación de 'parentesco' que establece una estructura jerárquica sobre los nudos. Cada nudo tiene un padre (excepto el raíz) y puede tener cero o más hijos. Se denomina **hoja** a un nudo sin hijos. Como ejemplo se muestra en la figura superior la tabla de contenidos de un libro

Formalmente un **árbol** se puede definir recursivamente del siguiente modo:

- Un **nudo simple** constituye un **árbol**. Este nudo se denomina la **raíz** del árbol
- Supongamos que **n** es un **nudo** y $\mathbf{T_1}$, $\mathbf{T_2}$, ..., $\mathbf{T_k}$ son **árboles** cuyas **raíces** son $\mathbf{n_1}$, $\mathbf{n_2}$, ..., $\mathbf{n_k}$, respectivamente. Podemos construir un nuevo **árbol** haciendo que **n** sea el **padre** de los nudos $\mathbf{n_1}$, $\mathbf{n_2}$, ..., $\mathbf{n_k}$. En el nuevo árbol **n** es la **raíz** y $\mathbf{n_1}$, ..., $\mathbf{n_k}$ se denominan los **hijos** de n.

Definiciones

UNIVERSIDAD DE CANTABRIA

- *Camino*: secuencia de nudos tales que cada uno es hijo del anterior
- *Longitud del camino*: nº de nudos que tiene
- *Antecesor*: un nudo es antecesor de otro si hay un camino del primero al segundo
- *Descendiente*: un nudo es descendiente de otro si hay un camino del segundo al primero
- *Subárbol* o *Rama*: Un nudo y todos sus descendientes

Notas:

UNIVERSIDAD DE CANTABRIA

En muchas ocasiones es conveniente incluir en la definición el **árbol nulo**, que no contiene nudos.

Si **n_1, n_2, ..., n_k** es una secuencia de nudos de un árbol, tal que n_i es el padre de n_{i+1} para 1_i<k, esta secuencia se denomina **camino desde el nudo n_1 al n_k**. La **longitud del camino** es el número de nudos del camino menos uno.

Si existe un camino desde el nudo a al b, se dice que a es un **antecesor** de b, y que b es un **descendiente** de a.

Se denominan antecesores o descendientes **propios** de un nudo a aquellos distintos del propio nudo. Un nudo sin descendientes propios se denomina **hoja** del árbol. La **raíz** del árbol es el único nudo sin antecesores propios. Un **subárbol** de un árbol es un nudo junto con todos sus descendientes.

8.1. Ordenación de los nudos

UNIVERSIDAD DE CANTABRIA

Los hermanos se ordenan generalmente de izquierda a derecha

Dos árboles ordenados, distintos

La ordenación o recorrido de los nudos se suele hacer de 3 modos:

- preorden, postorden, e inorden

Notas:

UNIVERSIDAD DE CANTABRIA

La ordenación de los nudos de un árbol es en general de extremada importancia, ya que el árbol representa una estructura jerárquica de datos que generalmente precisan ser ordenados.

Los hijos de un mismo nudo (los **hermanos**), se ordenan normalmente de izquierda a derecha.

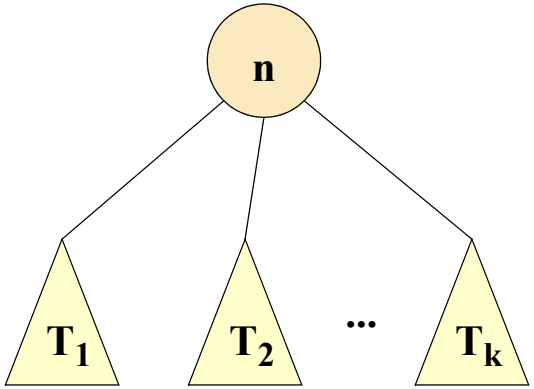
Esta definición de orden puede extenderse a nudos no hermanos, de forma que si a y b son hermanos, y a está a la izquierda de b, todos los descendientes de a están a la izquierda de los descendientes de b.

Existen diversas formas de realizar de forma sistemática la ordenación de todos los nudos de un árbol. Las formas más importantes son:

- Preorden u orden anterior
- Postorden u orden posterior
- Inorden (“Inorder”), u orden intermedio

Estas formas de ordenar se pueden usar también siempre que se necesite simplemente recorrer el árbol para hacer algo con todos sus nudos.

Ordenación de los nudos (cont.)



Preorden: $n,T_1,T_2,...,T_k$

Postorden: $T_1,T_2,...,T_k,n$

Inorden: $T_1,n,T_2,...,T_k$

Figura A

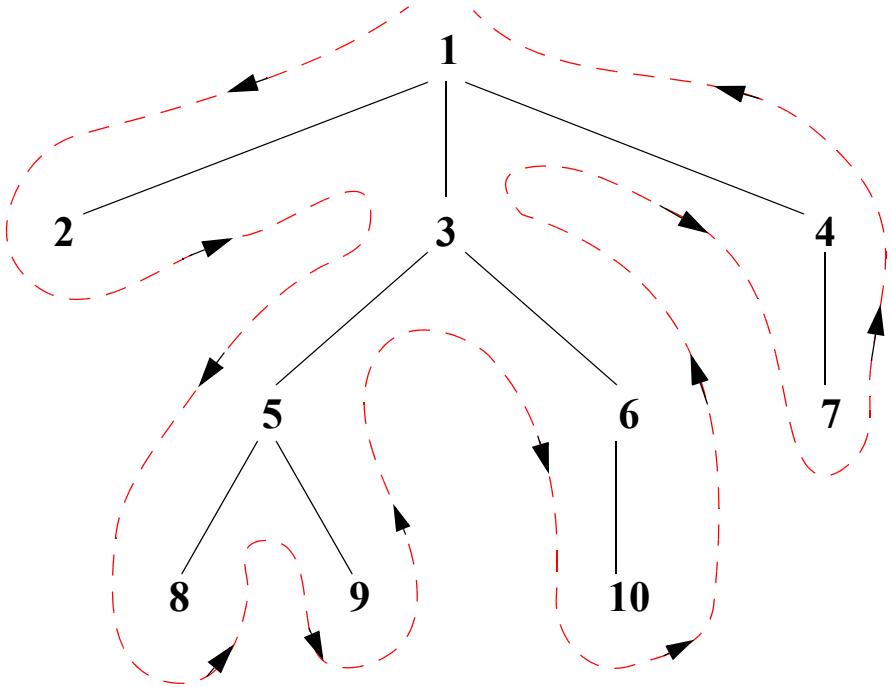


Figura B

Notas:

UNIVERSIDAD DE CANTABRIA

Estos tipos de ordenación se definen recursivamente de la forma siguiente:

1. Si el árbol T es nulo, entonces la lista vacía es la lista de T en preorden, postorden e inorden.
2. Si T tiene un solo nudo, entonces el nudo es la lista de T en preorden, postorden e inorden.
3. Si T consiste en un árbol con una raíz n y subárboles T_1, T_2, ..., T_k, como en la figura a de arriba:

   - a)La lista de T en preorden es la raíz n seguida de los nudos de T_1 en preorden, luego los nudos de T_2 en preorden, hasta finalizar con la lista de T_k en preorden.

   - b)La lista de T en inorden es la lista de los nudos de T_1 en inorden, seguida de la raíz n, luego los nudos de T_2, ..., T_k con cada grupo de nudos en inorden.

   - c)La lista de T en postorden es la lista de los nudos de T_1 en postorden, luego los nudos de T_2 en postorden, y así hasta la lista de T_k en postorden, finalizando con el nudo raíz n.

Un método para producir estas tres ordenaciones de nudos a mano consiste en recorrer los nudos en la forma que se indica en la figura b de arriba:

- Para ordenar en preorden se lista cada nudo la primera vez que se pasa por él. Para postorden la última vez. Para inorden se listan las hojas la primera vez que se pasa por ellas, pero los nudos interiores la segunda.

Procedimientos de ordenación

UNIVERSIDAD DE CANTABRIA

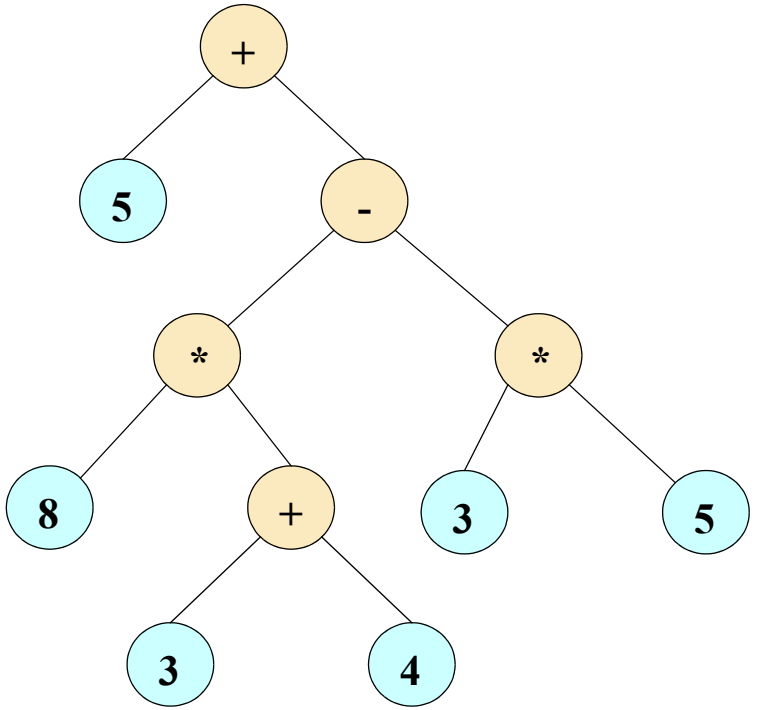
```
procedure Preorden (N : in Nudo; A : in Arbol) is
begin
   listar N;
   for cada hijo H de N, y empezando por la izquierda loop
         Preorden(H,A);
   end loop;
end Preorden;

procedure Inorden (N : in Nudo; A : in Arbol) is
begin
   if n es una hoja then listar n;
   else
      Inorden(hijo más a la izquierda de n,A);
      listar n;
      for cada hijo h de n, excepto el más a la
          izquierda, y empezando por la izquierda loop
           Inorden(H,A);
      end loop;
   end if;
end Inorden;
```

Ejemplo de ordenación de expresiones aritméticas

UNIVERSIDAD DE CANTABRIA

Expresión: 5+8*(3+4)-3*5:

- ***preorden***: **+5-*8+3,4*3,5**
- ***inorden***: **5+(8*(3+4)-(3*5)) es la expresión en notación matemática normal**
- ***postorden***: **5,8,3,4+*3,5*-+ es la expresión en Notación Polaca Inversa (RPN)**



8.2. El Tipo de datos abstracto árbol

UNIVERSIDAD DE CANTABRIA

Operaciones para crear y modificar el árbol:

operación	in	out	in out	errores
Haz_Nulo			El_Arbol	
Crea	El_Elemento	El_Arbol		No_Cabe
Anade_Rama-_Derecha	El_Nudo La_Rama		El_Arbol	Nudo_-Inco-rrecto No_Cabe
Anade_Hijo_-Mas_Izquierdo	El_Nudo El_Elemento		El_Arbol	Nudo_-Inco-rrecto No_Cabe
Anade_-Hermano_-Derecho	El_Nudo El_Elemento		El_Arbol	Nudo_-Inco-rrecto No_Cabe

Notas:

UNIVERSIDAD DE CANTABRIA

Las principales operaciones básicas que se asocian a los árboles para construir un tipo abstracto de datos son las siguientes:

Operaciones de creación y modificación de árboles

- HAZ_NULO(A): Hace nulo el árbol indicado
- CREA (v): ARBOL: Retorna un árbol con un único nudo (la raíz) cuyo valor es el elemento indicado
- ANADE_RAMA_DERECHA(n,A,R): Añade la rama R (que es un árbol) al árbol A, haciendo que n sea ascendiente de todos los nudos de R, y colocando la raíz de R como hijo más a la derecha de n.
- ANADE_HIJO_MAS_IZQUIERDO(v,n,A): Añade al nudo n del árbol A un hijo situado más a la izquierda de los actuales, y con el valor v
- ANADE_HERMANO_DERECHO(v,n,A): Añade al padre del nudo n del árbol A un hijo situado inmediatamente a la derecha del nudo n.

Las dos últimas operaciones no son necesarias, pues con CREA y ANADE-RAMA-DERECHA se puede construir cualquier árbol. Sin embargo, se ofrecen por comodidad, para poder ir construyendo un árbol elemento a elemento.

El ADT árbol (cont.)

UNIVERSIDAD DE CANTABRIA

Operaciones para recorrer el árbol:

operación	in	out	in out	errores
Raiz	El_Arbol	La_Raiz		
Padre	El_Nudo El_Arbol	El_Padre		Nudo_-Inco-rrecto
Hijo_Mas_-Izquierdo	El_Nudo El_Arbol	El_Hijo		Nudo_-Inco-rrecto
Hermano_-Derecho	El_Nudo El_Arbol	El_Hermano		Nudo_-Inco-rrecto
Elemento_De	El_Nudo El_Arbol	El_Elemento		Nudo_-Inco-rrecto

Notas:

UNIVERSIDAD DE CANTABRIA

Las operaciones para recorrer los nudos de un árbol en muchos casos retornan un nudo. Por este motivo es conveniente definir en el ADT la constante Nudo_Nulo, que retornarán estas operaciones si el nudo buscado no existe. Las operaciones son las siguientes:

- RAIZ(A): NUDO: Retorna el nudo raiz, o Nudo_Nulo si no existe
- PADRE(n,A) : NUDO: Retorna el nudo padre de n, o Nudo_Nulo si no tiene padre
- HIJO_MAS_IZQUIERDO(n,A) : NUDO: Retorna el hijo más izquierdo de n, o Nudo_Nulo si no tiene hijos
- HERMANO_DERECHO(n,A): NUDO: Retorna el hermano derecho de n, o Nudo_Nulo si no tiene
- ELEMENTO_DE(n,A): ELEMENTO: Retorna el valor del elemento almacenado en el nudo n

El error Nudo_Incorrecto que pueden retornar la mayoría de las operaciones corresponde a la situación en la que el nudo de entrada es un Nudo_Nulo.

Especificación de árboles

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Arboles;
generic
   type Elemento is private;
package Arboles is

   type Nudo is private;
   type Arbol is private;
   Nudo_Nulo : constant Nudo;

   Nudo_Incorrecto : exception
     renames Excepciones_Arboles.Nudo_Incorrecto;
   No_Cabe : exception
     renames Excepciones_Arboles.No_Cabe;

   procedure Haz_Nulo
     (El_Arbol : in out Arbol);

   procedure Crea
     (El_Elemento : in Elemento;
      El_Arbol    : out Arbol);
```

Especificación de árboles (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Anade_Rama_Derecha
   (El_Nudo   : in Nudo;
    La_Rama   : in Arbol;
    El_Arbol  : in out Arbol);
-- puede elevar Nudo_Incorrecto o No_Cabe

procedure  Anade_Hijo_Mas_Izquierdo
   (El_Nudo      : in Nudo;
    El_Elemento  : in Elemento;
    El_Arbol     : in out Arbol);
-- puede elevar Nudo_Incorrecto o No_Cabe

procedure  Anade_Hermano_Derecho
   (El_Nudo      : in Nudo;
    El_Elemento  : in Elemento;
    El_Arbol     : in out Arbol);
-- puede elevar Nudo_Incorrecto o No_Cabe
```

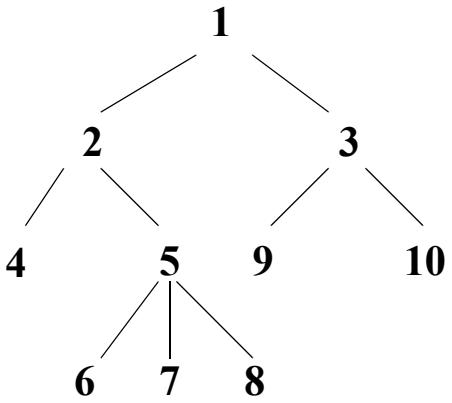
Especificación de árboles (cont.)

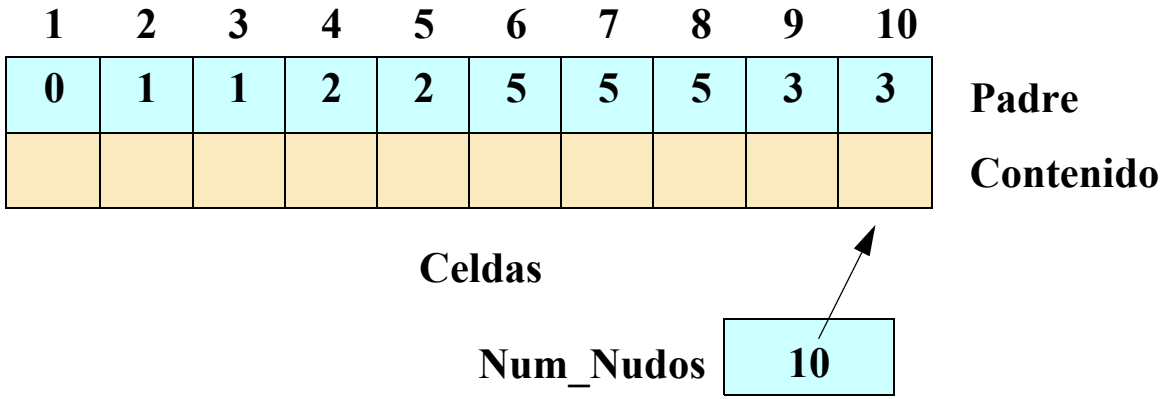
```
   function Raiz
     (El_Arbol : Arbol) return Nudo;

   function Padre
     (El_Nudo : Nudo; El_Arbol : Arbol) return Nudo;
   -- puede elevar Nudo_Incorrecto

   function Hijo_Mas_Izquierdo
     (El_Nudo : Nudo; El_Arbol : Arbol) return Nudo;
   -- puede elevar Nudo_Incorrecto
   function Hermano_Derecho
     (El_Nudo : Nudo; El_Arbol : Arbol) return Nudo;
   -- puede elevar Nudo_Incorrecto

   function Elemento_De
     (El_Nudo : Nudo; El_Arbol : Arbol) return Elemento;
   -- puede elevar Nudo_Incorrecto
private
   ...
end Arboles;
```

8.3. Implementación mediante arrays con cursor al padre

UNIVERSIDAD DE CANTABRIA

La implementación más sencilla es la de un array en el que cada elemento tiene un cursor al padre

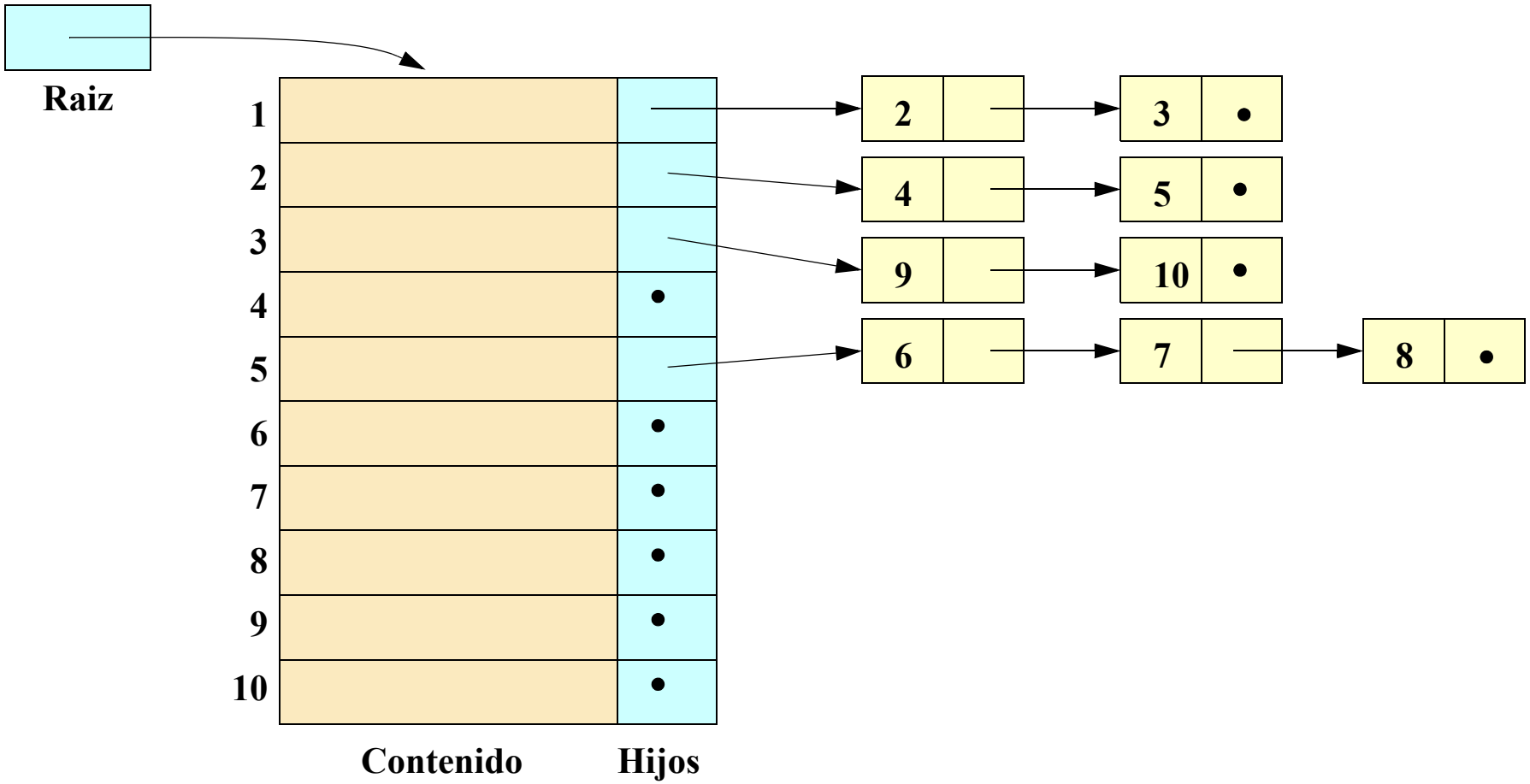




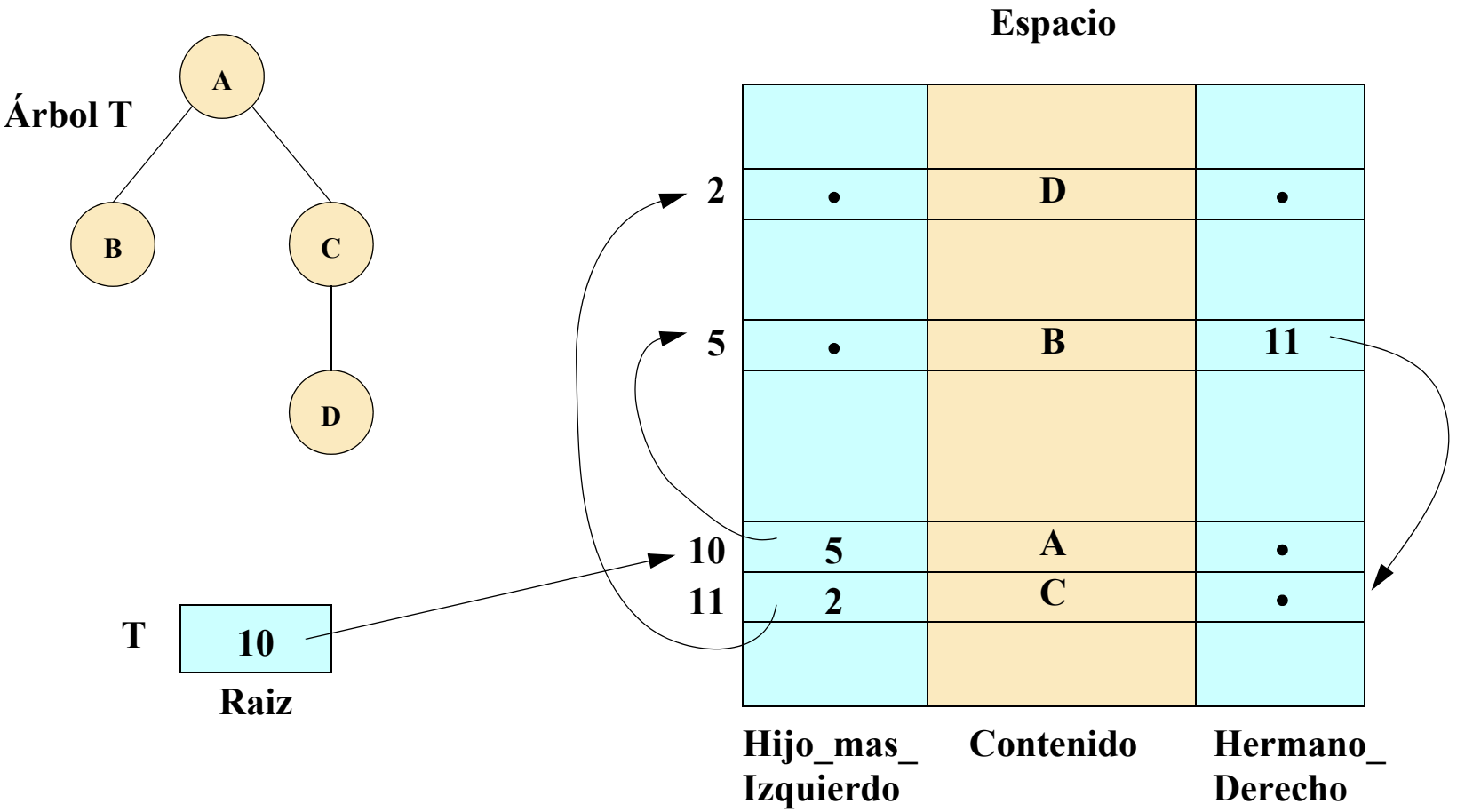
Para que el orden de los nudos esté bien definido

- se numeran los hijos con números mayores al del padre
- se numeran los hermanos en orden creciente de izquierda a derecha

8.4. Representación de árboles con listas de hijos



8.5. La representación hijo-mas-izqdo., hermano-derecho

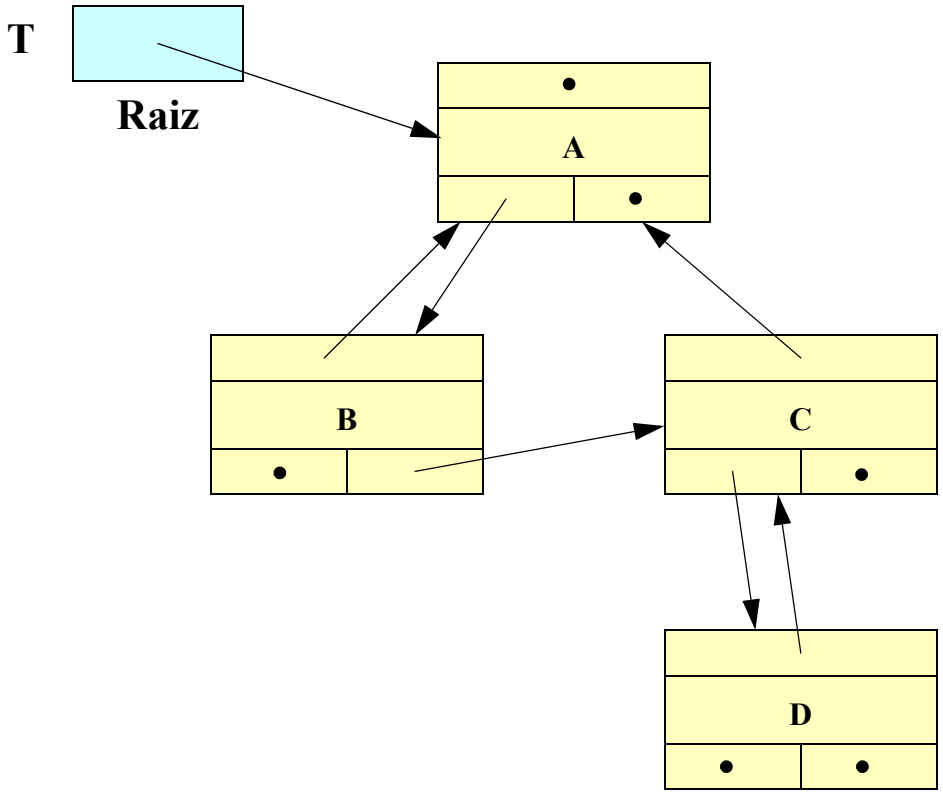


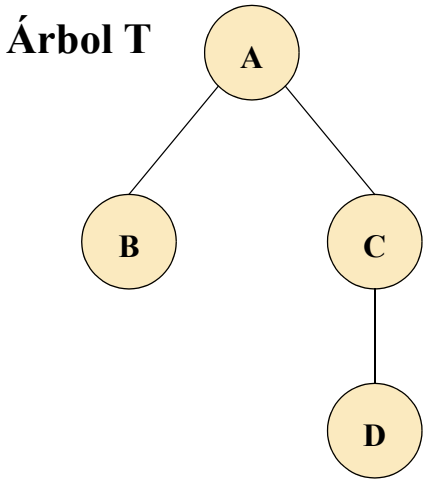
Notas:

UNIVERSIDAD DE CANTABRIA

La estructura de datos anterior presenta diversos problemas, como por ejemplo el hecho de que no se puede construir de forma sencilla un árbol grande a partir de otros pequeños. Por ello se pueden utilizar representaciones aún más complejas para los árboles, como la representación mediante hijo-mas-izquierdo, hermano-derecho. A continuación se representa la declaración de esta estructura de datos realizada mediante cursores, en la que pueden coexistir varios árboles en el mismo array. Para que la estructura funcione de forma eficiente es necesario disponer en ella de una lista de celdas libres

8.6 La representación padre, hijo-mas-izqdo., herm.-dcho.

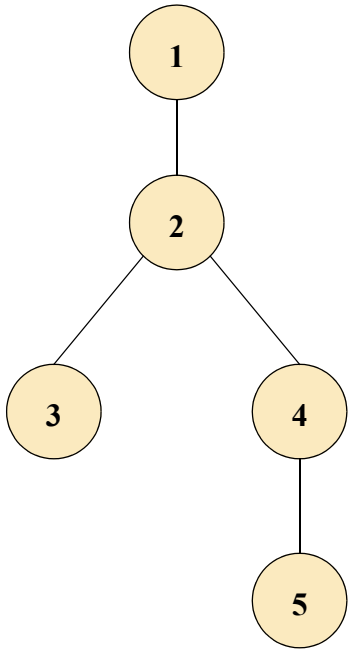




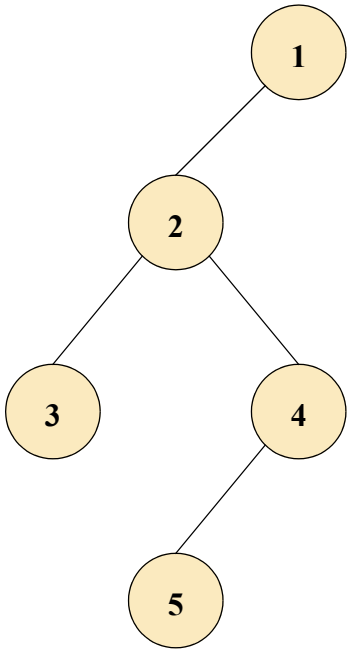
9. Árboles binarios

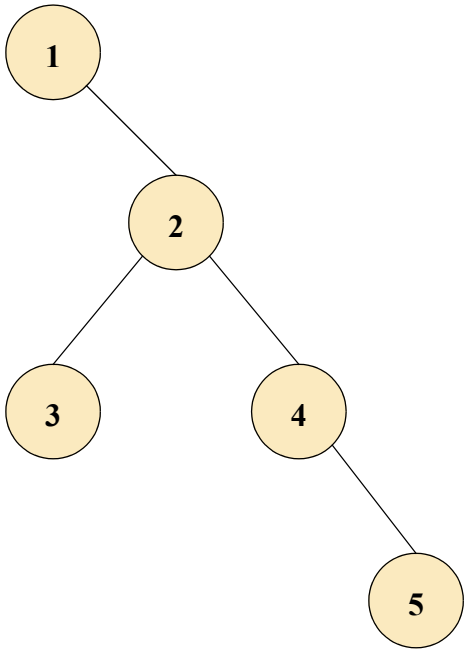
UNIVERSIDAD DE CANTABRIA

Un árbol binario es un árbol orientado y ordenado, en el que cada nudo puede tener un hijo izquierdo y un hijo derecho



Un árbol ordinario





Dos árboles binarios

Notas:

UNIVERSIDAD DE CANTABRIA

Un tipo de árbol diferente del árbol ordenado y orientado que hemos visto hasta ahora es el **árbol binario.** Un árbol binario se define como un árbol vacío, o un árbol en el que cada nudo puede no tener hijos, tener un hijo izquierdo, un hijo derecho, o ambos. La especificación de las operaciones de un árbol binario se muestra a continuación.

Especificación de árboles binarios

UNIVERSIDAD DE CANTABRIA

```
with Excepciones_Arboles;
generic
   type Elemento is private;
package Arboles_Binarios is

   type Nudo is private;
   type Arbol_Binario is private;

   Nudo_Nulo : constant Nudo;
   Arbol_Nulo : constant Arbol_Binario;

   Nudo_Incorrecto : exception
     renames Excepciones_Arboles.Nudo_Incorrecto;

   function Crea
     (El_Elemento_Raiz : in Elemento;
      Rama_Izquierda,
      Rama_Derecha : in Arbol_Binario:=Arbol_Nulo)
     return Arbol_Binario;
```

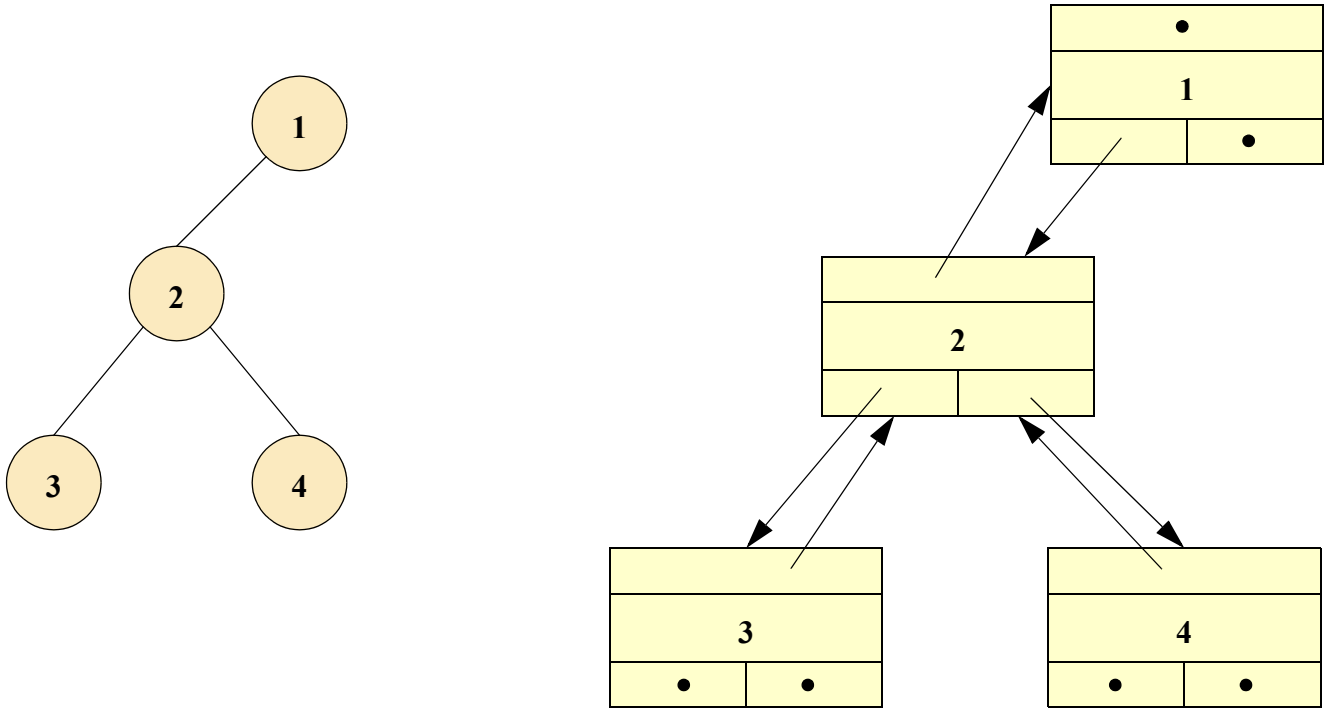
Especificación de árboles binarios (cont.)

UNIVERSIDAD DE CANTABRIA

```
   function Raiz
     (El_Arbol : Arbol_Binario) return Nudo;

   function Padre
     (El_Nudo : Nudo; El_Arbol : Arbol_Binario) return Nudo;
   -- puede elevar Nudo_Incorrecto

   function Hijo_Izquierdo
     (El_Nudo : Nudo; El_Arbol : Arbol_Binario) return Nudo;
   -- puede elevar Nudo_Incorrecto
   function Hijo_Derecho
     (El_Nudo : Nudo; El_Arbol : Arbol_Binario) return Nudo;
   -- puede elevar Nudo_Incorrecto

   function Elemento_De
     (El_Nudo : Nudo; El_Arbol : Arbol_Binario) return Elemento;
   -- puede elevar Nudo_Incorrecto
private
   ...
end Arboles_Binarios;
```

Implementación mediante punteros

Una posible implementación: cada nudo tiene un puntero al padre, al hijo izquierdo, y al hijo derecho:



Búsquedas en árboles binarios

UNIVERSIDAD DE CANTABRIA

Los árboles binarios se adaptan muy bien para buscar elementos de forma eficiente.

Para ello, todos los elementos se almacenan en el árbol ordenados:

- Todos los descendientes izquierdos de un nudo son menores que él
- Todos los descendientes derechos de un nudo son mayores que él

En este caso, la búsqueda es O(log n) en el caso promedio, si el árbol está equilibrado

Existen algoritmos sencillos de inserción equilibrada (ej:*AVL*)

Especificación de la función localiza

UNIVERSIDAD DE CANTABRIA

```
with Arboles_Binarios;
generic
   type Elemento is private;
   with function "<" (E1,E2 : Elemento) return Boolean;
   with package Arboles is new Arboles_Binarios(Elemento);
function Localiza
  (El_Elemento : Elemento;
   Desde       : Arboles.Nudo;
   El_Arbol    : Arboles.Arbol_Binario)
  return Arboles.Nudo;

   -- la funcion retorna Nudo_Nulo si no encuentra el elemento
   -- no funciona si el árbol no está ordenado
```

Cuerpo de la función localiza

UNIVERSIDAD DE CANTABRIA

```
function Localiza
   (El_Elemento : Elemento;
    Desde       : Arboles.Nudo;
    El_Arbol    : Arboles.Arbol_Binario)
  return Arboles.Nudo
is
   use Arboles;
begin
   if Desde = Nudo_Nulo then
      return Nudo_Nulo;
   elsif El_Elemento=Elemento_De(Desde,El_Arbol) then
      return Desde;
   elsif El_Elemento < Elemento_De(Desde,El_Arbol) then
      return Localiza(El_Elemento,
                      Hijo_Izquierdo(Desde,El_Arbol),El_Arbol);
   else
      return Localiza(El_Elemento,
                      Hijo_Derecho(Desde,El_Arbol),El_Arbol);
   end if;
end Localiza;
```