Estructuras de Datos y Algoritmos

Tema 3: Arrays y listas enlazadas

Departamento de Informática
Universidad de Valladolid

Curso 2021-22

Grado en Ingeniería Informática
Grado en Estadística

TADs versus Representación

- Un **TAD** define las **operaciones fundamentales**, la **relación entre los elementos** (ninguna, precedencia, jerarquía, vecindad) y una serie de **restricciones** (elementos repetidos si/no, existencia o no de operaciones de igualdad y comparación entre ellos).
- Una **representación** define el **cómo se almacenan** los elementos en memoria y puede establecer una **relación** de precedencia o jerarquía entre ellos (que puede **no coincidir** con la del TAD) y **añadir restricciones** a las impuestas por el TAD.
- La representación condiciona los **algoritmos** disponibles para realizar las operaciones del TAD y por tanto la **eficiencia** de las operaciones.

1. SECUENCIAS

Secuencias

- La categoría de representaciones basadas en **secuencias** definen una **relación de precedencia** entre los elementos.
- Pueden usarse para representar TADs que no impogan esa relación: Se puede representar un conjunto, por ejemplo, mediante una secuencia.
- Existen tres clasificaciones, ortogonales entre sí:
  - Secuencias **ordenadas** (orden interno) o **no ordenadas** (orden externo). Si no se dice nada se supone no ordenada.
  - Secuencias **contiguas** o **enlazadas**.
  - Secuencias **lineales** o **circulares**. Si no se dice nada se supone que son lineales.
- Los **arrays de bits** son un caso especial útil para los TADs conjuntos y mapas (si se dan ciertas condiciones).

Contiguo versus enlazado

- Las representaciones **contiguas** almacenan los elementos en posiciones **consecutivas** de memoria.
  - Se corresponden con el concepto de **array** de la mayoría de lenguajes de programación.
  - Aunque existen algunos lenguajes en los que no son la estructura de datos fundamental (Javascript → Mapas, Haskell → Lista enlazada)
  - Su ventaja es que el **acceso indexado es O(1).**
  - La desventaja es que las **modificaciones** (inserción, borrado, …) requieren **desplazar** elementos (peor caso **O(*n*)**)
- Las representaciones **enlazadas** almacenan los elementos en **cualquier posición** de memoria. A cada elemento se le añade información extra para indicar la secuencia.
  - Desventaja: El **acceso indexado es O(n)** en peor caso.
  - Ventaja: Las **modificaciones**, si se está situado en el punto adecuado, no requieren desplazar elementos: Su orden es **O(1)**

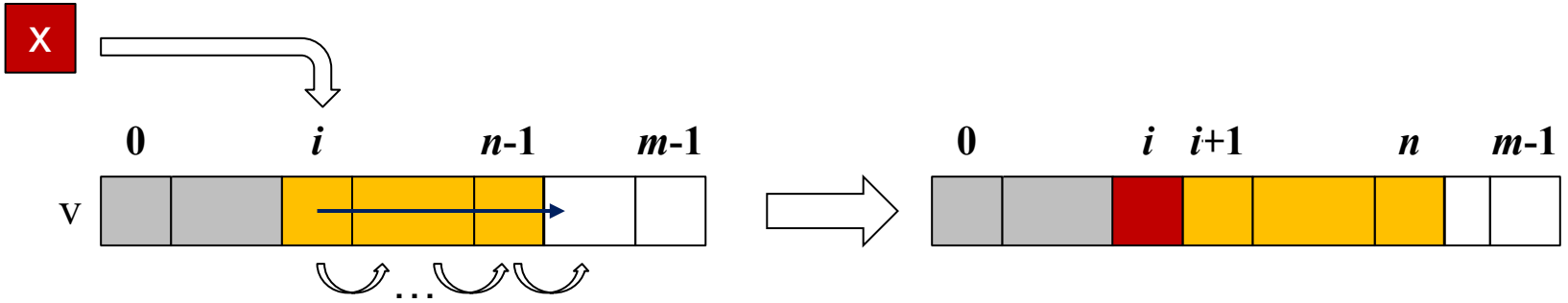
2. REPRESENTACIONES CONTIGUAS

Contigua lineal

- Consiste en un array de capacidad *m* que almacena *n* elementos ($n \leq m$) en sus primeras posiciones.
- El acceso basado en índice es $O(1)$
- El acceso basado en valor (búsqueda) es $O(n)$ en el peor caso: Algoritmo de búsqueda secuencial.
- Se supone que el array se puede ampliar: Si se inserta un elemento en un array lleno ($n = m$) se duplica su capacidad:
  - En general eso implica crear un nuevo array de capacidad $2 \cdot n$ y copiar los datos del array antiguo en el nuevo.
  - Supone un coste $O(n)$
  - Pero garantiza que las siguientes n-1 inserciones tienen espacio.

Contigua lineal (II)

- La inserción de un elemento para que se situe en el índice *i* supone desplazar hacia delante los siguientes elementos:

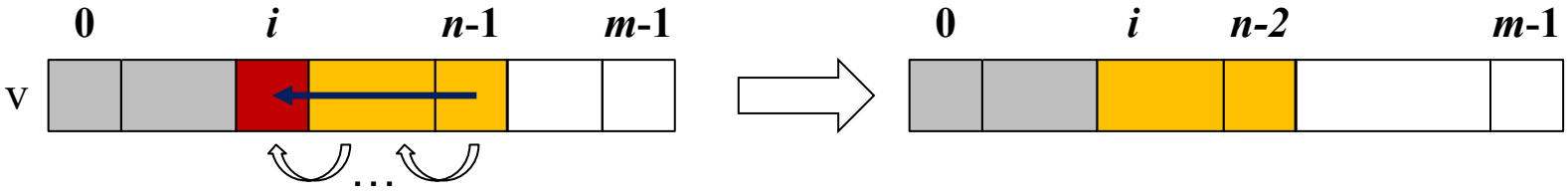


- La inserción al final es *O*(1) en tiempo amortizado (mejor caso).
- La inserción al principio es *O*(*n*) (peor caso).
- El caso promedio es *O*(*n*)

```
∀ k ∈ [n-1..i] : v[k+1] ← v[k]
v[i] ← x
n ← n+1
```

Contigua lineal (III)

- El borrado del elemento situado en el índice *i* supone desplazar hacia atrás los siguientes elementos:



- El borrado del último es *O*(1) (mejor caso).
- La borrado del primero es *O*(*n*) (peor caso).
- El caso promedio es *O*(*n*)

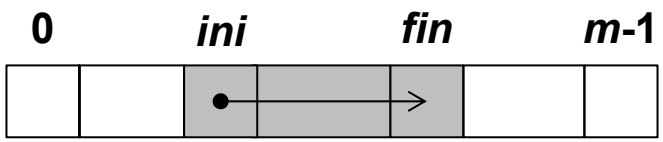
```
∀ k ∈ [i+1..n-1] : v[k-1] ← v[k]
n ← n-1
```

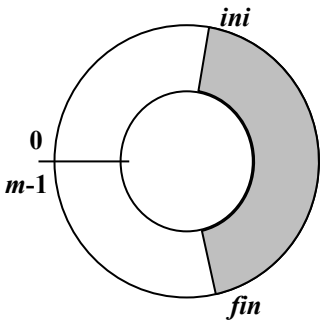
Contigua circular (I)

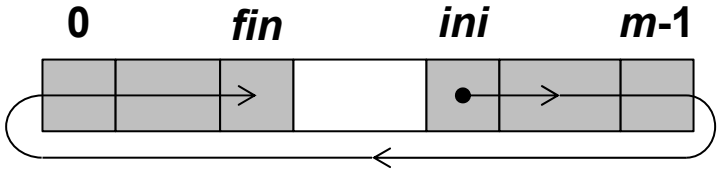
- Una representación contigua lineal es adecuada para representar una **pila**, eligiendo como cabeza (extremo por el que se realizan las operaciones de inserción y borrado) el último elemento.
- Todas las operaciones serían $O(1)$ en tiempo amortizado.
- El problema consiste en que al representar una **cola** o bien la inserción o bien el borrado deben hacerse sobre el primer elemento (y tendrían un coste $O(n)$).
- La representación circular consigue que las operaciones de inserción o borrado sobre ambos extremos (primer y último elemento) tengan coste $O(1)$.
- Para ello se elimina la restricción de que los elementos se encuentren situados al principio del array (aunque siguen estando contiguos, sin “huecos” entre ellos)

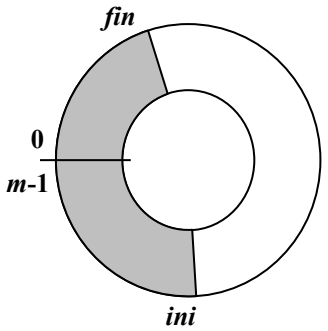
Contigua circular (II)

- Se añaden dos variables, *ini* y *fin,* que marcan la zona del vector ocupada por los elementos.
- Se supone que los extremos del array estan unidos: Tras la posición *m*-1 se encuentra la posición 0, y la posición anterior a la 0 es la *m*-1. Puede darse que *ini* > *fin*
- Todos los incrementos o decrementos de índices se realizan módulo *m*.









Contigua circular (III)

- Las operaciones de índice deben usar **aritmética modular**:
  - Incremento del índice k: **(k+1) % m**
  - Decremento del índice k: **(k+m-1) % m** (se suma *m* para trabajar siempre con valores positivos)
  - Número de elementos: **(fin-ini+m+1) % m**
  - Se debe evitar que *ini* = *fin* (existe ambiguedad en este caso)
- El **acceso por índice** a los elementos se realiza contando a partir de la posición dada por *ini*: **v[(ini+i) % m]**
- **Inserción al principio**: **ini ← (ini+m-1) % m; v[ini] ← x**
- **Inserción al final**: **fin ← (fin+1) % m; v[fin] ← x**
- **Borrado del primero**: **ini ← (ini+1) % m**
- **Borrado del último**: **fin ← (fin+m-1) % m**

Contigua circular - Inserción


```
{ Insertar x en posición i-ésima (0-based)
  Se supone que no es necesario ampliar el vector }
```

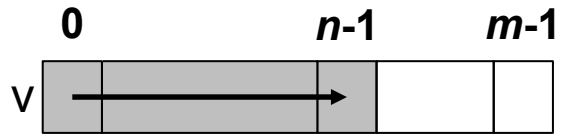
```
p ← (ini+i-1+m) % m { Posición (i-1)-ésima en el vector }
k ← fin
{ Se mueven hacia delante los elementos [p..fin] }
mientras k ≠ p
  v[(k+1) % m] ← v[k]; { v[k+1] ← v[k] }
  k ← (k-1+m) % m;     { k ← k-1 }
{ Insertar el elemento en posición i-ésima }
v[(p+1) % m] ← x
{ Incrementar el índice al último }
fin ← (fin+1) % m
```

Contigua circular - Borrado


```
{ Borrar elemento en posición i-ésima (0-based) }

p ← (ini+i) % m { Posición i-ésima en el vector }
k ← p
{ Se mueven hacia atras los elementos [p+1..fin] }
mientras k ≠ fin
   v[k] ← v[(k+1) % m]; { v[k] ← v[k+1] }
   k ← (k+1) % m;       { k ← k+1 }
{ Decrementar el índice al último }
fin ← (fin-1+m) mod m
```

Contigua ordenada

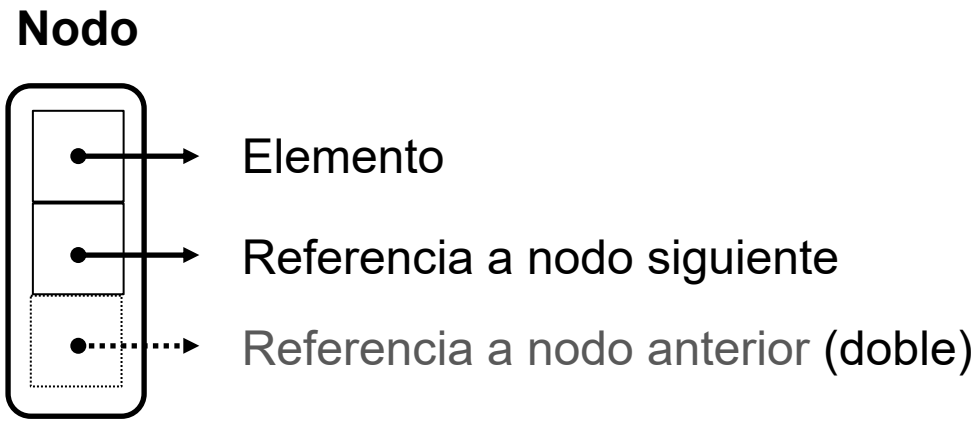


- En una representación contigua ordenada los elementos se disponen en **orden interno** (de menor a mayor o de mayor a menor, según convenga).
- La operación de **acceso al *i*-ésimo menor** es O(1).
- La operación de **búsqueda** puede usar el algoritmo de búsqueda binaria, y es de orden O(log *n*)
- Las operaciones de **inserción** y **borrado** son O(*n*) en el peor caso porque tienen que desplazar elementos.
  - El **borrado del máximo** es O(1) si el orden es de menor a mayor.
  - El **borrado del mínimo** es O(1) si el orden es de mayor a menor.
  - La operación de inserción debe **buscar** el punto de inserción, y **desplazar** los elementos: O(*n* log *n*) + O(*n*)
- No existen ventajas en representarla de forma circular

3. REPRESENTACIONES ENLAZADAS

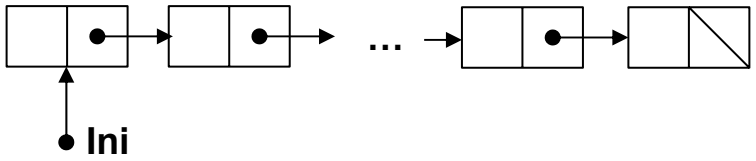
Representación Enlazada

- En las representaciones enlazadas los elementos se almacenan en **nodos**, que añaden al menos un campo para indicar donde se encuentra el **elemento siguiente**.
- La posición de los nodos en memoria es **irrelevante**: No tienen porqué ser contiguos y pueden existir “huecos” entre ellos.
- La secuencia se establece por la **información extra** que incorpora el nodo.

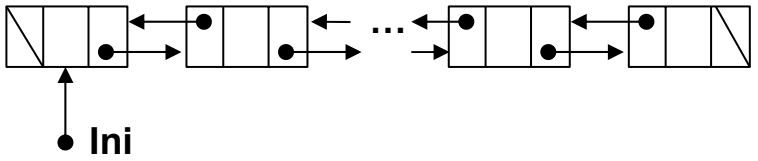


Variantes

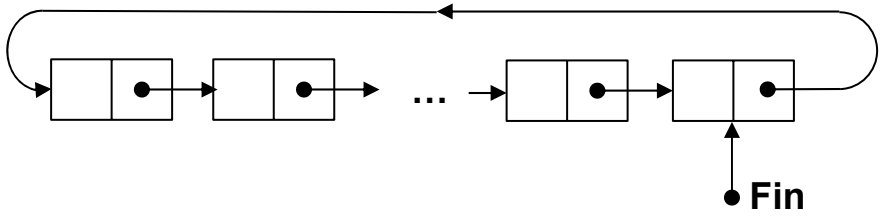
- **Simple, lineal**



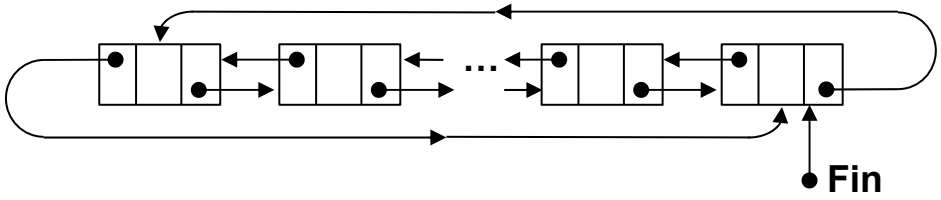
- **Doble, lineal**



- **Simple, circular**



- **Doble, circular**

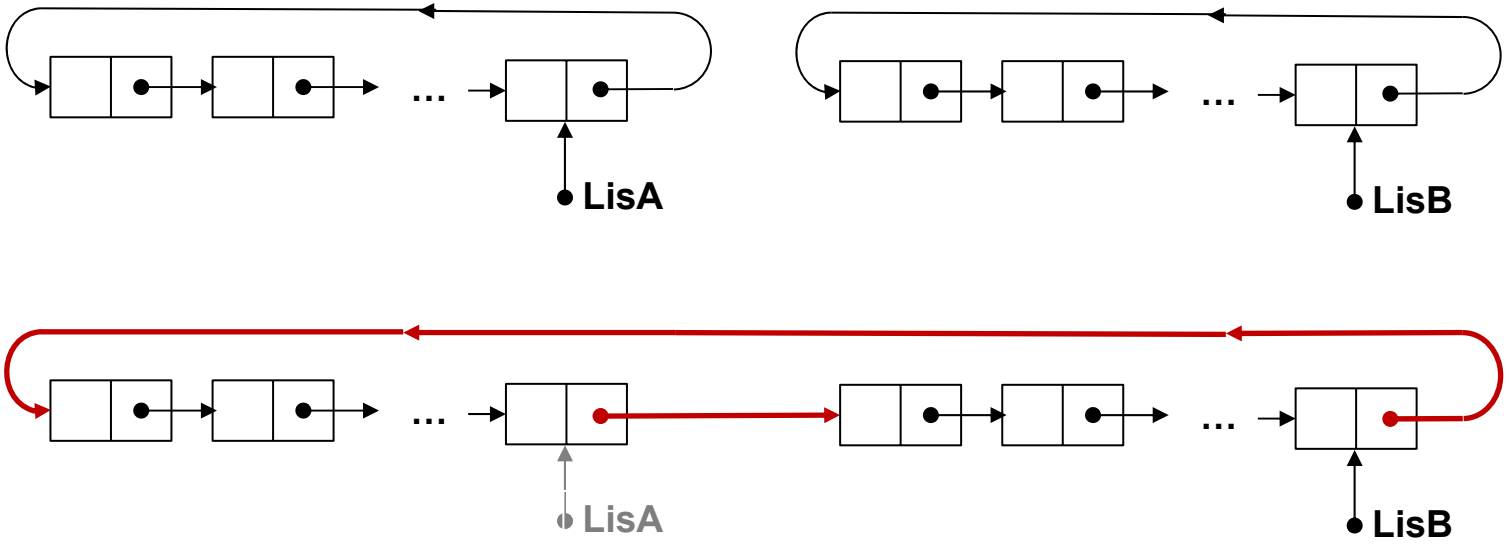


Eficiencia

- **Simple, lineal**: Acceso, borrado e inserción del primer elemento son operaciones O(1). Sobre el último elemento son O(n)
- **Doble, lineal**: Las inserciones y borrados basados en iterador son más sencillas de codificar.
- **Simple, circular**: Acceso e inserción son O(1) sobre ambos extremos. El borrado del primero es O(1), y el del último O(n).
- **Doble, circular**: Acceso, borrado e inserción por ambos extremos son operaciones O(1)
- Las **operaciones basadas en índice** requieren pasar por todos los elementos anteriores, y son **O(n)** en el peor caso.
- El algoritmo de **búsqueda binaria** es **O(n)**, debido al desplazamiento para encontrar el punto medio.
- La **ordenación por fusión** es **O(n log n)** y **O(log n)** en espacio.

Concatenación

- **Permitiendo destrucción de 1ª lista: O(1)**



4. ARRAYS DE BITS

Arrays de bits

- Si los elementos que se van a almacenar son **enumerables** (se pueden poner en correspondencia con un rango de los números naturales) es entonces posible usar los elementos como **índices** de un array.
- Se obtiene una representación dada por un array, con una celda por cada posible valor del elemento, que almacena:
  - Un valor booleano (bit) indicando si el elemento pertenece o no a la estructura (caso de **conjuntos**)
  - Un enlace al valor asociado (o nulo si no existe) en el caso de un **mapa** o **diccionario**.
- El **espacio** ocupado (y las operaciones de recorrido) es proporcional al número de posibles valores de los elementos (***m***) y no al del número de elementos almacenados *(n).*

5. ANÁLIS DE EFICIENCIA

Eficiencia TADs Conjunto/Mapa

	Contigua	Contigua ordenada	Enlazada (no repetidos)	Enlazada ordenada	Array de bits
Pertenencia (conjunto) Acceso por clave (mapa)	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Borrado (por valor/clave)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Borrado (por iterador)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Inserción (por valor)	$O(1)$*	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Unión (ambos tamaño n)	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$	$O(m)$
Espacio	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(m)$

* *Tiempo amortizado (incremento tamaño array)*

Eficiencia TAD Lista

	Contigua	Enlazada lineal	Enlazada circular
Acceso por índice	$O(1)$	$O(n)$	=
Acceso por iterador	$O(1)$	$O(1)$	=
Borrado por índice	$O(n)$	$O(n)$	=
Borrado por iterador	$O(n)$	$O(1)$	=
Inserción por índice	$O(n)$	$O(n)$	=
Inserción por iterador de lista	$O(n)$	$O(1)$	=
Búsqueda	$O(n)$	$O(n)$	=
Concatenación	$O(n)$	$O(n)$	$O(1)$

Eficiencia TAD Pila/Cola/Bicola

	Contigua lineal	Contigua circular	Enlazada simple lineal	Enlazada simple circular	Enlazada doble circular
Acceso primero	O(1)	O(1)	O(1)	O(1)	O(1)
Acceso último	O(1)	O(1)	O(*n*)	O(1)	O(1)
Borrado primero	O(*n*)	O(1)	O(1)	O(1)	O(1)
Borrado último	O(1)	O(1)	O(*n*)	O(*n*)	O(1)
Inserción al principio	O(*n*)	O(1)*	O(1)	O(1)	O(1)
Inserción al final	O(1)*	O(1)*	O(*n*)	O(1)	O(1)

Eficiencia TAD Lista Ordenada y Cola de Prioridad

	Contigua ordenada	Enlazada ordenada	Contigua
Acceso mínimo	O(1)	O(1)	O(1)[b]
Acceso *i*-ésimo menor	O(1)	O(*n*)	O(*n*)[c]
Borrado mínimo	O(1)[a]	O(1)	O(*n*)[b]
Borrado *i*-ésimo menor	O(*n*)	O(*n*)	O(*n*)[c]
Borrado por iterador	O(*n*)	O(1)	---[d]
Inserción por valor	O(*n*)*	O(*n*)	O(1)[b]
Búsqueda	O(log *n*)	O(*n*)	O(*n*)
Fusión	O(*n*)	O(*n*)	O(*n*)

Notas:

a) Se supone ordenada de mayor a menor

b) Se almacena referencia al mínimo, adaptada en la inserción y el borrado.

c) Tiempo promedio, algoritmo basado en partición

d) El coste de mantener un iterador ordenado lo hace inviable

Eficiencia TAD Diccionario

	Contigua ordenada	Enlazada ordenada
Acceso por clave	$O(\log n)$	$O(n)$
Acceso clave *i*-ésima menor	$O(1)$	$O(n)$
Acceso por iterador	$O(1)$	$O(1)$
Borrado por clave	$O(n)$	$O(n)$
Borrado clave *i*-ésima menor	$O(n)$	$O(n)$
Borrado por iterador	$O(n)$	$O(1)$
Inserción por valor	$O(n)$	$O(n)$