Algoritmos y estructuras de datos

CENTRO UNIVERSITARIO DE TECNOLOGÍA Y ARTE DIGITAL

www.u-tad.com

Grado en Ingeniería en Desarrollo de Contenidos Digitales

1. Introducción a la construcción de software
2. Algoritmos no recursivos
3. Algoritmos recursivos
4. **Listas, pilas y colas**
5. Ordenación
6. Tablas hash
7. Árboles
8. Grafos

Introducción listas I

- Lista es una secuencia (finita) de cero o más elementos de un tipo determinado
  - En principio consideramos listas homogéneas (todos los elementos son del mismo tipo)…
  - … pero también es posible trabajar con listas heterogéneas…
  - … o que sus elementos sean a su vez listas
- El tamaño de la lista (“n”) es su número actual de elementos
- A veces también hablamos de “capacidad” de la lista, que es el tamaño máximo que puede llegar a tener

Introducción listas II

- La implementación de una lista puede ser:
  - Contigua: los elementos se almacenan uno detrás de otro en memoria
    - La capacidad de la lista puede ser estática (no cambia) o dinámica/redimensionable (va aumentando o disminuyendo según el tamaño de la lista va creciendo o disminuyendo)
    - También se llaman “arrays” o “vectores”
  - No contigua (basada en punteros): cada elemento apunta a su siguiente
    - También se llaman “listas enlazadas” o incluso a veces simplemente “listas”
    - No es relevante el concepto de “capacidad”, ya que siempre tiene la capacidad para los elementos actuales

Contiguas: capacidad I

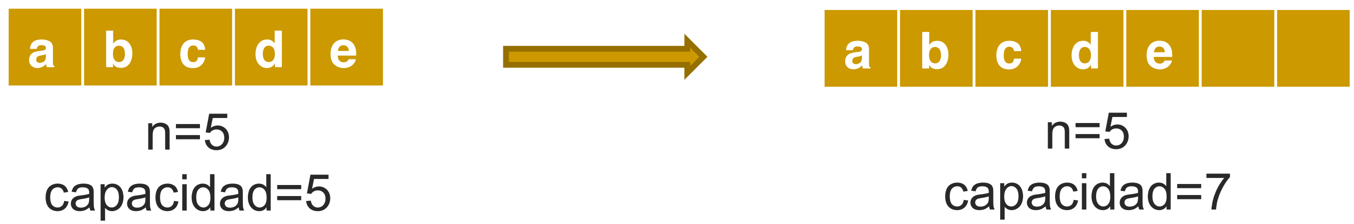
- Los elementos de las listas contiguas se almacenan en memoria consecutivamente, de acuerdo a su posición
- Utilizando punteros en vez de arrays estáticos, podemos cambiar la capacidad de la lista, ampliando o disminuyendo dinámicamente la memoria reservada para nuestra lista
  - Función realloc() en C

Contiguas: capacidad II

- Mecanismo de ampliación o disminución de memoria reservada (realloc). Tres casos:
  - Caso 1. Si hay memoria libre a continuación de la ya reservada, se limita a pedir al sistema operativo una ampliación (caso mejor)
  - Caso 2. Si no hay suficiente memoria libre a continuación de la ya reservada, tiene que pedir al sistema operativo una nueva zona lo suficientemente grande, copiar a la nueva todo lo que había en la vieja, y liberar la vieja (caso peor)
  - Caso 3. En el caso de una disminución de memoria, se limita a pedir al sistema operativo que la zona de memoria que ya teníamos reservada ahora acaba antes

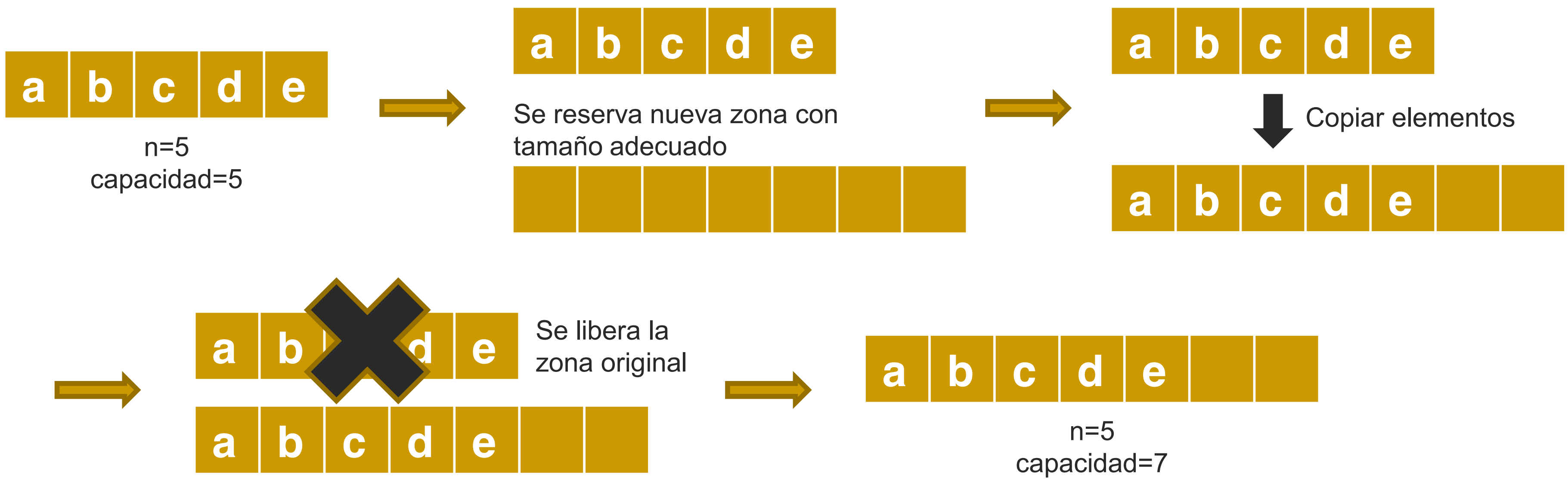
Contiguas: capacidad III

- Caso 1 (caso mejor): si hay memoria libre a continuación de la ya reservada:
  - Ejemplo: nuestra lista tiene n=5 y capacidad=5 (está llena). Queremos ampliar la capacidad en dos posiciones más



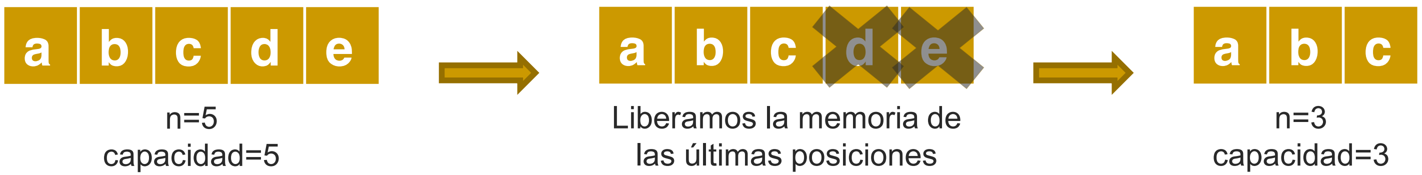
Contiguas: capacidad IV

- Caso 2 (caso peor): si NO hay memoria libre a continuación de la ya reservada:
  - Mismo ejemplo que antes



Contiguas: capacidad V

- Caso 3 (disminución de memoria reservada):
  - Mismo ejemplo que antes: queremos decrementar la memoria reservada en 2



Contiguas: capacidad VI

- Complejidad temporal de aumentar/disminuir capacidad:
  - Reservar/liberar memoria siempre es O(1), porque el sistema operativo se limita a anotar el principio y fin de la memoria reservada
  - Ampliar la capacidad es O(n) porque puede implicar copiar todos los elementos a una nueva zona de memoria más grande (caso peor)
  - Disminuir la capacidad es O(1) porque nos limitamos a decir al sistema operativo que la memoria reservada ahora termina antes

Contiguas: capacidad VII

- Complejidad espacial de aumentar/disminuir capacidad:
  - Ampliar la capacidad es O(nuevaCapacidad) en el caso peor porque en algún momento del algoritmo tenemos “nuevaCapacidad” posiciones de memoria extra reservadas
    - Hasta que liberamos la zona original, estamos gastando “nuevaCapacidad” de memoria adicional
  - Disminuir la capacidad es O(1) porque nunca gastamos memoria adicional
    - Nos limitamos a reducir la que ya tenemos

Contiguas: capacidad VIII

- ¿Cuándo aumentar la capacidad? (**actividad 4.1**)
  - Ampliar/disminuir la capacidad cada vez que la lista aumenta/disminuye en 1 su tamaño sería demasiado costoso
    - Es decir, mantener en todo momento n=capacidad (se llama "lista de capacidad ajustada")
    - Sería demasiado costoso porque aumentar la capacidad es O(n), y eso lo tendríamos que hacer cada vez que insertamos un nuevo elemento
  - Solución: cuando se llene la capacidad, aumentarla para unos cuantos ("incremento"). Igualmente, decrementarla en "incremento" posiciones sólo cuando nos sobre "2·incremento" posiciones
    - La idea es que, después de aumentar o disminuir el tamaño, no tengamos que volver a hacerlo inmediatamente al volver a insertar/eliminar un nuevo elemento

Contiguas: capacidad IX

- Ejemplo con lista de capacidad ajustada
  - Cada vez que insertamos, tenemos que ampliar la capacidad, y esto es muy costoso en tiempo –O(n)- y memoria adicional –O(n)-

a	b	c	d	e

n=5 | capacidad=5

↓ Insertamos “f” al final. Aumentamos capacidad en 1

a	b	c	d	e	f

n=6 | capacidad=6

↓ Insertamos “z” en medio. Aumentamos capacidad en 1

a	b	c	z	d	e	f

n=7 | capacidad=7

↓ Eliminamos “a” al principio. Disminuimos capacidad en 1

b	c	z	d	e	f

n=6 | capacidad=6

Contiguas: insertar y eliminar I

- Insertar/eliminar un elemento en la posición "i":
  - Insertar en la posición "i":
    - Ampliar la capacidad si es necesario
    - Mover una posición a la derecha ("desplazar" una posición a la derecha) todo lo que haya desde "i"
    - En el hueco que ha quedado en la posición "i", meter el nuevo elemento
  - Eliminar la posición "i":
    - Mover una posición a la izquierda ("desplazar" una posición a la izquierda) todo lo que haya desde "i+1"
    - Disminuir la capacidad si es necesario

Contiguas: insertar y eliminar II

- Para insertar/eliminar podemos utilizar "memmove()" en C para desplazar los elementos a la derecha/izquierda
  - La zona de memoria de origen y la zona de memoria destino se solapan ("overlap")
    - Copiar elemento a elemento mediante un bucle puede dar problemas porque podemos sobreescribir la zona de memoria de origen antes de haber terminado de leerla
  - Memcpy() no maneja los solapamientos
  - Memmove() utiliza un array temporal, en donde guarda la zona de memoria origen, para evitar los problemas de solapamientos
    - Por lo tanto memmove() tiene complejidad espacial O(n)
    - Podríamos programarnos un memmove() "eficiente" que no necesitase un array temporal, manejando con mucho cuidado el orden a seguir para copiar elemento a elemento
      - Habrá que distinguir dos casos: que la zona de memoria de origen esté antes o después de la zona de memoria de destino

Contiguas: insertar y eliminar III

- Complejidad temporal de insertar en la posición "i" (peor caso):
  - Ampliar la capacidad: O(n) en el peor caso, porque implica copiar de la vieja zona de memoria a la nueva
  - Desplazar a la derecha: O(n) en el peor caso (cuando i=0)
  - Poner el nuevo elemento en el hueco que queda en la posición "i": O(1)

Contiguas: insertar y eliminar IV

- Complejidad espacial de insertar en la posición "i" (peor caso):
  - Ampliar la capacidad: O(n) en el peor caso, porque hay un momento (antes de copiar a la nueva zona de memoria) en el cual tenemos reservadas ambas zonas a la vez (la nueva y la vieja)
    - Sería O(capacidad), pero como capacidad es n±incremento e incremento es constante, O(capacidad) = O(n)
  - Desplazar a la derecha:
    - O(n) si usamos memmove()
    - O(1) con nuestro memmove() "eficiente"
  - Poner el nuevo elemento: O(1)

Contiguas: insertar y eliminar V

- Complejidad temporal de eliminar en la posición "i" (peor caso):
  - Desplazar a la izquierda desde i+1: O(n) en el peor caso (cuando i=0)
  - Disminuir la capacidad: O(1)
- Complejidad espacial de eliminar en la posición "i" (peor caso):
  - Desplazar a la izquierda desde i+1:
    - O(n) si usamos memmove()
    - O(1) si usamos nuestro memmove() "eficiente"
  - Disminuir la capacidad: O(1)

Contiguas: insertar y eliminar VI

- Acceso a un elemento (para ver o editar) en la posición “i”:
  - Complejidad temporal: O(1)
    - Conocemos la dirección de comienzo del array…
    - Conocemos la posición del elemento dentro del array (posición “i”)…
    - … Por lo tanto la dirección del elemento es “comienzo+i”
      - Calcular esto es elemental, porque sólo es una suma
      - Acceder a una dirección de memoria ya conocida es también elemental, pues la memoria RAM es de acceso “aleatorio” (se puede acceder a cualquier posición sin pasar por las posiciones anteriores)
  - Complejidad espacial: O(1)
    - No necesitamos memoria adicional

Análisis de contiguas

- Ventajas de las listas contiguas (**actividad 4.2**):
  - Complejidad temporal de acceder a un elemento es O(1)
  - Podemos ordenar con mucha eficiencia usando métodos avanzados como quicksort, con complejidad temporal O(nlogn)
  - Podemos hacer búsqueda binaria, con complejidad temporal de O(logn), si la lista está ordenada.
  - Ocupa la memoria casi justa: únicamente lo que ocupan sus elementos más el "incremento"
- Inconvenientes de las listas contiguas:
  - Necesitamos que existan grandes zonas de memoria libres y contiguas
    - Casi siempre hay memoria libre de sobra, pero no nos serviría si está fragmentada (es decir, en pequeños trozos)
  - Insertar y eliminar elementos son operaciones muy costosas:
    - Complejidad temporal es O(n)
    - Complejidad espacial es:
      - O(n) si utilizamos memmove(). Además la memoria adicional requerida tiene que ser contigua
      - O(1) si utilizamos nuestro memmove() "eficiente"

No contigua I

- Una implementación no contigua de una lista es:
  - Una lista de nodos
  - Cada nodo se puede almacenar en una posición libre cualquiera de la memoria
    - Por lo tanto el atributo “capacidad” ya no existe
  - Cada elemento se almacena en un nodo
  - Los nodos se identifican por su posición dentro de la lista, empezando por 0 y terminando en n-1

No contigua II

- Un “nodo” es una estructura de datos que:
  - Contiene un elemento
  - Contiene un puntero al siguiente nodo
  - Puede también contener un puntero al nodo anterior (listas doblemente enlazadas)
- La estructura de datos “listaNoContigua” contendrá al menos un atributo que será un puntero al primer nodo
  - Será NULL si la lista está vacía
  - En ocasiones nos puede interesar tener también un puntero al último nodo

No contigua III

- Tipos de listas no contiguas:
  - Lista enlazada simple
    - Cada nodo tiene un puntero que apunta a la dirección del siguiente nodo
    - El puntero del nodo en la posición n-1 (el último) contiene NULL, pues ya no hay ningún nodo después de él
  - Lista doblemente enlazada
    - Cada nodo tiene dos punteros: uno que apunta al siguiente nodo (como antes) y otro que apunta al anterior
    - En el nodo de posición n-1 (el último de la lista), su puntero al nodo siguiente contiene un NULL (al igual que en la lista enlazada simple)
    - En el nodo de posición 0 (el primero de la lista), su puntero al nodo anterior contiene un NULL.
    - Su utilidad es para cuando queremos saber el nodo anterior a uno dado
  - Lista enlazada simple circular
    - Como la lista enlazada simple, pero el último nodo apunta al primero (el siguiente del último es el primero)
    - Por lo tanto, no hay ningún puntero que tenga NULL en ningún momento
  - Lista doblemente enlazada circular
    - Como la lista doblemente enlazada, pero el siguiente nodo del último es el primero, y el anterior al primero es el último
    - Por lo tanto, no hay ningún puntero que tenga NULL en ningún momento

No contigua IV

- Acceso a un elemento es O(n):
  - Tenemos que localizar el nodo al que queremos acceder, pasando por todos sus anteriores
    - Esto es complejidad temporal O(n) en el peor caso (acceder al último en enlazada simple)
  - Una vez localizado obtenemos su dirección y ya podemos hacer operaciones sobre él
    - Ver
    - Modificar

No contigua V

- Cómo insertar un elemento en la posición “i”:
  - Se crea un nodo con ese elemento
  - Se localiza el elemento de la posición “i”
  - Hacemos que el campo “siguiente” del nodo i-1 apunte al nuevo nodo
  - Hacemos que el campo “siguiente” del nuevo nodo apunte al antiguo nodo i
  - Si la lista es doblemente enlazada, también hacemos que:
    - El campo “anterior” del nuevo nodo apunte al nodo i-1
    - El campo “anterior” del antiguo nodo i apunte al nuevo nodo
  - Posiblemente tengamos que distinguir los casos especiales de insertar en la primera posición (posición “0”) e insertar al final de la lista (insertar en posición “n”)

No contigua VI

- Complejidad temporal de insertar es O(n):
  - Crear un nuevo nodo es O(1):
    - Reservar memoria: O(1)
    - Meter el elemento en el nodo: O(1) porque no depende de n, sino de la naturaleza del elemento
  - Localizar un nodo en la posición "i" es O(n), porque hay que pasar por todos los anteriores para llegar a él
  - Actualizar todos los punteros es O(1), porque son operaciones elementales
- Complejidad espacial de insertar es O(1)
  - La memoria adicional que necesitamos es sólo la del nuevo nodo que estamos creando, por lo tanto es constante independientemente de n

No contigua VII

- Cómo eliminar un elemento de la posición “i”:
  - Se localiza el elemento de la posición “i”
  - Hacemos que el campo “siguiente” del nodo i-1 apunte al nodo i+1
  - Eliminamos el nodo i, liberando su memoria
  - Si la lista es doblemente enlazada, también hacemos que el campo “anterior” del nodo i+1 apunte al nodo i-1
  - Posiblemente tengamos que distinguir los casos especiales de eliminar el primer (posición “0”) o último (posición “n-1”) nodo de la lista

No contigua VIII

- Complejidad temporal de eliminar es O(n)
  - Localizar el nodo es O(n)
  - Actualizar los punteros es O(1)
  - Eliminar el nodo es O(1)
- Complejidad espacial de eliminar es O(1)
- Practica ahora programando una lista enlazada simple (**actividad 4.3**)

No contigua IX

- A la hora de implementar insertar o eliminar, es posible que haya que distinguir casos especiales:
  - Cuando la lista está vacía
    - El atributo de la lista que apunta al primer nodo (y al último nodo, si lo hay) es NULL
  - Cuando la lista tiene sólo un nodo
    - Lista enlazada simple circular: el puntero “siguiente” del nodo se apunta a sí mismo
    - Lista doblemente enlazada y circular: los punteros “anterior” y “siguiente” del nodo apuntan al propio nodo
  - Al insertar o eliminar en la primera o última posición, habrá que tener cuidado en actualizar el atributo de la lista que apunta al primer/último nodo
  - Para asegurarse que todo funciona, siempre hacer pruebas de caja negra también con casos límite (además de los casos “normales”):
    - Listas con 0, 1 y 2 elementos
    - Insertar/eliminar en las posiciones 0, 1, n-2, n-1, n
    - Insertar en listas vacías o con 1 elemento y eliminar en listas con 1 ó 2 elementos.

No contigua X

- Ventajas de las listas enlazadas:
  - No necesitamos grandes bloques de memoria contigua
    - Esto es una gran ventaja cuando n es muy grande, pues es difícil encontrar bloques grandes de memoria contigua
    - De hecho, esta es la principal razón de usar listas enlazadas en vez de contiguas
  - Insertar y eliminar no gastan memoria adicional
  - No necesitamos preocuparnos por la capacidad
  - Eliminar e insertar son O(n) en tiempo como en listas contiguas (debido a que es O(n) localizar la posición en donde insertar o eliminar), pero las constantes ocultas son mucho menores en las listas enlazadas porque:
    - Insertar en contiguas requería dos copiados: al aumentar la capacidad, y al desplazar los elementos
    - Además para localizar un nodo en una lista doblemente enlazada y circular necesitaremos pasar, como máximo, por n/2 nodos

No contigua XI

- Inconvenientes de las listas enlazadas:
  - Acceder a un elemento es O(n)
  - Para almacenar la lista necesitamos más memoria que con las contiguas, por los punteros que contiene cada nodo
    - ¿Cuánta memoria más? Del orden de O(n)
  - No podemos ordenar con algoritmos eficientes del orden de O(nlogn), porque sólo en localizar un elemento ya tardamos O(n)
    - Ejercicio sugerido: ¿Cuál sería la complejidad temporal de la ordenación por Selección en una lista enlazada?
  - Tampoco podemos aplicar la búsqueda binaria en O(logn)
    - Por la misma razón: localizar un elemento ahora es O(n)
    - Ejercicio sugerido: ¿Cuál sería ahora la complejidad temporal de la búsqueda binaria?

Comparación entre listas I

- ¿Cuándo usar una lista contigua?
  - Cuando tengamos muchos más accesos que inserciones o eliminaciones
    - Pues los accesos son O(1), pero las inserciones/eliminaciones son O(n) y además con grandes constantes ocultas
  - Cuando queramos ahorrar memoria
    - Pues la lista ocupa justo lo necesario para sus elementos
  - Cuando tengamos grandes porciones de memoria contiguas
    - La necesitamos para guardar la lista
    - La necesitamos para la memoria adicional de insertar y eliminar
  - Cuando el tamaño de la lista no sea muy grande
    - Será probable encontrar zonas de memoria contiguas
    - No importará mucho la memoria adicional de insertar/eliminar
  - Cuando necesitemos ordenar o buscar con eficiencia
    - Podremos usar algoritmos avanzados de ordenación de O(nlogn)
    - Podremos usar la búsqueda binaria de O(logn)

Comparación entre listas II

- ¿Cuándo usar una lista enlazada? En general, cuando no podamos usar una lista contigua (**actividad 4.4**)
  - Cuando haya muchas más inserciones/eliminaciones que accesos
    - Las inserciones/eliminaciones ahora son más eficientes que en las listas contiguas
      - Sus constantes ocultas son menores
      - No requieren memoria adicional
  - Cuando no nos importe desperdiciar memoria
    - Pues la lista ocupa más, al tener que guardar los punteros para apuntar al nodo anterior y siguiente
    - Desde otro punto de vista, insertar/eliminar no requieren memoria adicional, por lo cual funcionan mejor en entornos con poca memoria
  - Cuando no tengamos grandes porciones de memoria contiguas
    - Cada nodo se puede guardar en cualquier lugar de la memoria, no necesariamente contiguo al anterior
    - Insertar/eliminar no requieren memoria adicional contigua (ni no contigua)
  - Cuando el tamaño de la lista sea muy grande
    - Al ser muy grande, es imposible encontrar grandes trozos de memoria contigua, y por lo tanto no nos queda más remedio que usar una enlazada
  - Cuando no necesitemos ordenar o buscar con mucha eficiencia

Aplicaciones de listas I

- Polinomios
  - Un polinomio se puede ver como una lista de sumandos, cada uno con un exponente
  - Ejemplo: $P(x) = -x^{10} + 8x – 3$
    - Implementación con una lista de coeficientes, en donde cada coeficiente se guarda en una posición u otra según su exponente:
      - [-1, 0, 0, 0, 0, 0, 0, 0, 0, 8, -3]
      - Observemos que desperdiciamos mucha memoria, pues muchos elementos son iguales (ceros)
    - Con dos listas, una guarda el exponente y otra el coeficiente (no desperdiciamos memoria como antes):
      - Exponentes: [10, 1, 0]
      - Coeficientes: [-1, 8, -3]
    - Con una lista, en la que cada elemento contiene tanto el exponente como el coeficiente:
      - [(10,-1), (1,8), (0,-3)]

Aplicaciones de listas II

- Números grandes
  - En ocasiones los tipos numéricos soportados por la máquina no son suficientes
    - Ej: un entero de 1024 bits para guardar el número de electrones del universo
  - Cualquier número se puede ver como un polinomio, debido al Teorema Fundamental de la Numeración
    - Ej: 1492,36 en base 10

$$1492,36 = 1 \cdot 10^3 + 4 \cdot 10^2 + 9 \cdot 10^1 + 2 \cdot 10^0, +3 \cdot 10^{-1} + 6 \cdot 10^{-2}$$

  - Una vez transformado en polinomio, lo podemos guardar como una lista de sumandos, tal como hemos visto
  - Operar con ellos requerirá algoritmos que no serán O(1)

Aplicaciones de listas III

- Matrices dispersas
  - Una matriz dispersa es aquella que tiene “muchos” ceros.
  - Si utilizamos las matrices proporcionadas por el lenguaje (ej: arrays de punteros a arrays), estamos desperdiciendo mucha memoria pues casi todos los elementos son iguales (ceros)
  - Al igual que en el caso de los polinomios, cada fila podría ser una lista de los elementos distintos de cero.
    - La matriz entera será una lista de filas.

Introducción a pilas y colas I

- Una pila es una lista en la cual:
  - Cada vez que metemos un elemento (hacer “push”), éste se introduce por un lado de la lista (ej: por el final)
  - Cada vez que sacamos un elemento (hacer “pop”), éste se saca por el mismo lado de la lista por el cual los elementos se meten (ej: por el final)
    - No podemos hacer pop si la pila está vacía
  - El lado por el que se meten o sacan elementos se llama “cima” de la pila
- También se llaman listas LIFO: last input is the first output (el último en entrar es el primero en salir). Sale el elemento más nuevo.
- Ejemplos: pila de platos, pila de libros

Introducción a pilas y colas II

- Una cola es una lista en la cual:
  - Cada vez que metemos un elemento (hacer “push”), éste se introduce por un lado de la lista (ej: por el final)
  - Cada vez que sacamos un elemento (hacer “pop”), éste se saca por el lado de la lista opuesto al lado por el cual los elementos se meten (ej: por el principio)
    - No podemos hacer pop si la cola está vacía
- También se llaman listas FIFO: first input is the first output (el primero en entrar es el primero en salir). Sale el elemento más antiguo.
- Ejemplos: cola del supermercado, cola de tareas pendientes

Implementación pilas y colas I

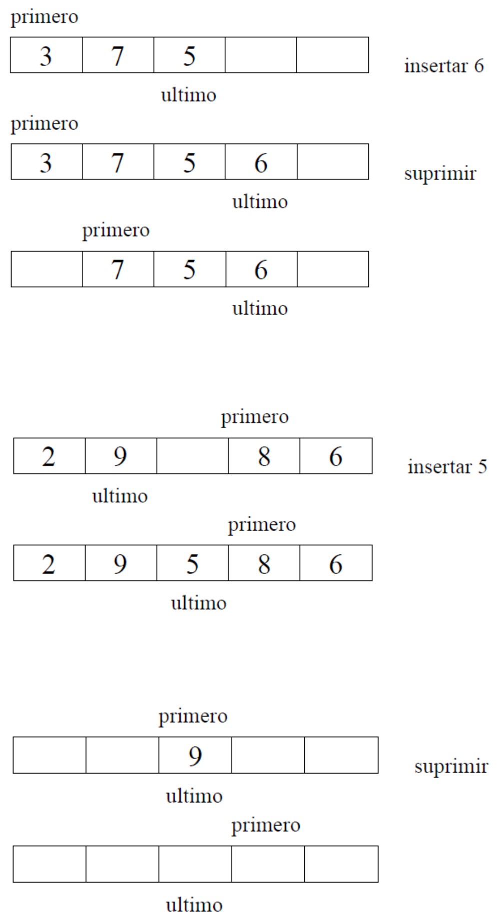
- Implementación de pilas (**actividad 4.5**). El objetivo es que push() y pop() sean O(1)
  - A partir de una lista contigua
    - La cima es el último elemento, para no tener que mover todos los elementos cada vez que hacemos push() o pop()
  - A partir de una lista no contigua
    - La cima debe ser tal que hacer pop() y push() sea O(1)
      - Generalmente esto se consigue si la cima es el primer elemento de la lista
      - Si la lista es doblemente enlazada y circular, la cima también podría ser el último elemento de la lista, pues se accede de forma inmediata desde el primero
    - Como solo vamos a eliminar e insertar por un lado, no merece la pena usar listas doblemente enlazadas ni circulares
      - Basta con una lista enlazada sencilla en donde la cima sea el primer elemento

Implementación pilas y colas II

- Implementación de colas. Objetivo: que push y pop sean O(1)
  - Con listas enlazadas
    - Perfecto si es doblemente enlazada y circular. Podemos elegir:
      - Insertar por el principio y eliminar por el último
      - Insertar por el último y eliminar por el principio
    - Si es enlazada simple, tendríamos que tener también un puntero al último elemento, no sólo al primero
      - Es una solución más sencilla que la doblemente enlazada y circular
  - No es adecuada una lista contigua tal cual, porque no conseguimos que las dos operaciones (insertar y eliminar) sean O(1)
    - Si elegimos insertar por el principio y eliminar por el final, insertar implica mover todos los elementos siguientes a la derecha
    - Si elegimos eliminar por el principio e insertar por el final, eliminar implica mover todos los elementos siguientes hacia la izquierda

Implementación pilas y colas III

- Implementación de colas (cont.):
  - Podemos utilizar una lista contigua si la convertimos en un "vector circular"
    - El "vector circular" en principio tiene un tamaño limitado, pero también podríamos ir aumentando su capacidad dinámicamente
      - Tendríamos que insertar posiciones libres entre el último y el primero si la lista no está vacía
    - Para saber si la lista está vacía o llena, utilizamos "n", no las posiciones de "primero" y "último"



Colas de prioridad I

- Una cola de prioridad es una lista en la cual:
  - Cada elemento tiene una prioridad asociada
  - Se hace pop() sólo del elemento de mayor prioridad
    - Si hay varios con prioridad máxima, el primero que llegó
- No es una cola exactamente
  - Comparten características con las colas y con las listas ordenadas

Colas de prioridad II

- Implementación de una cola de prioridad
  - Mediante una lista que siempre permanece ordenada por prioridades
    - Cada vez que hacemos push(), ordenamos la lista
    - Hacemos pop() por donde nos sea más fácil, para conseguir O(1)
  - Mediante una lista no ordenada
    - Cuando llega un elemento se hace push() por donde nos sea más fácil
    - Al hacer pop(), se busca el elemento deseado de forma secuencial
  - ¿Cuál es mejor? Depende de si vamos a hacer más veces push() o pop()
  - ¿Por dónde nos es más fácil hacer push() o pop()?
    - Si la lista es contigua, por el final
    - Si la lista es enlazada simple, por el principio
    - Si la lista es enlazada doble y circular, nos da igual por el principio o por el final

Colas de prioridad III

- Implementación de una cola de prioridad (cont.):
  - Vector de colas
    - Si el número de prioridades se conoce de antemano y es pequeño, podemos hacer un vector en el cual cada componente es a su vez una cola
    - Cada componente del vector se asocia de forma fija con una prioridad
      - El componente 0 es prioridad 0, el 1 es prioridad 1… el m es prioridad m
    - Cada componente es una cola en donde se guardan todos los elementos que tienen la misma prioridad
    - Insertamos en O(1) y eliminamos en O(m), donde m es el número de prioridades distintas
      - Habrá que ir recorriendo el vector hasta que encontremos una cola no vacía (peor caso: recorremos m posiciones). Entonces sacaremos un elemento de dicha cola en O(1)

Aplicaciones pilas y colas I

- Interpretar expresiones matemáticas:
  - Una aplicación de las pilas es interpretar expresiones matemáticas en tiempo de ejecución
  - Las expresiones matemáticas se pueden escribir:
    - Con notación infija
      - Primero escribimos el primer operador, luego el operando, y luego el segundo operador
      - Es lo habitual entre los humanos
      - Ej: 1+2*3+(4*5+6)*7
    - Con notación postfija:
      - Primero escribimos los dos operandos, y luego el operador
      - Son más fáciles de interpretar (ejecutar) por los ordenadores
      - Ej: 1+2*3+(4*5+6)*7 en notación postfija es 123*+45*6+7*+
  - Las pilas se usan para transformar infijo en postfijo, y para interpretar la expresión en postfijo, dando el resultado

Aplicaciones pilas y colas II

- Mecanismo de llamadas entre funciones:
  - El mecanismo de llamadas entre métodos y funciones usa una pila:
    - Cuando un método o función “A” llama a otro “B”, debemos de almacenar en una pila el estado de “A” (variables locales, por ejemplo) para recuperarlo cuando “B” termine
    - Por lo tanto, cuando “B” termina, se hace pop() de esa pila para recuperar el estado de “A” en el momento en que fue interrumpido para llamar a B
    - Debemos de utilizar una pila porque “B” a su vez puede llamar a “C” (se hace push() del estado de “B”), “C” a “D” (se hace push() del estado de “C”), etc. Luego todos ellos van retornando en orden inverso: primero “D” (se hace pop() para recuperar el estado de “C”), luego “C” (se hace pop() para recuperar el estado de “B”), etc.
  - Por esta razón, un programa recursivo usa memoria adicional para la pila que se necesita para todas las llamadas recursivas

Aplicaciones pilas y colas III

- Eliminación de la recursividad (transformar un programa recursivo en iterativo):
  - Utilizando colas existe un método general para transformar un programa recursivo en otro iterativo que realiza lo mismo y sin perder eficiencia
  - Problemas de la recursividad que se solucionan al convertirlo en iterativo:
    - Hay lenguajes que no la soportan
    - Una versión iterativa puede ser más eficiente (ej: Fibonacci)
    - Un programa recursivo requiere memoria adicional para la pila que almacena el estado de las llamadas
  - Los compiladores suelen eliminar la recursividad y transformarla en iteratividad…
    - … pero alguien tiene que hacer los compiladores