Estructuras de datos

1. Tipos abstractos de datos y algoritmia
2. **Listas, pilas, colas y conjuntos** ←
3. Árboles
4. Grafos

Universidad Rey Juan Carlos

www.urjc.es

Grado en Ingeniería de Computadores

Curso 2020/2021

Prof. Dr. Carlos Grima Izquierdo

www.carlosgrima.com

v3

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
- En algunos tipos de listas también tenemos la “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”
    - Aquí ya no existe el concepto de “capacidad”, pues ya no es relevante (la capacidad es ilimitada)

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 acabe antes

Contiguas: capacidad III

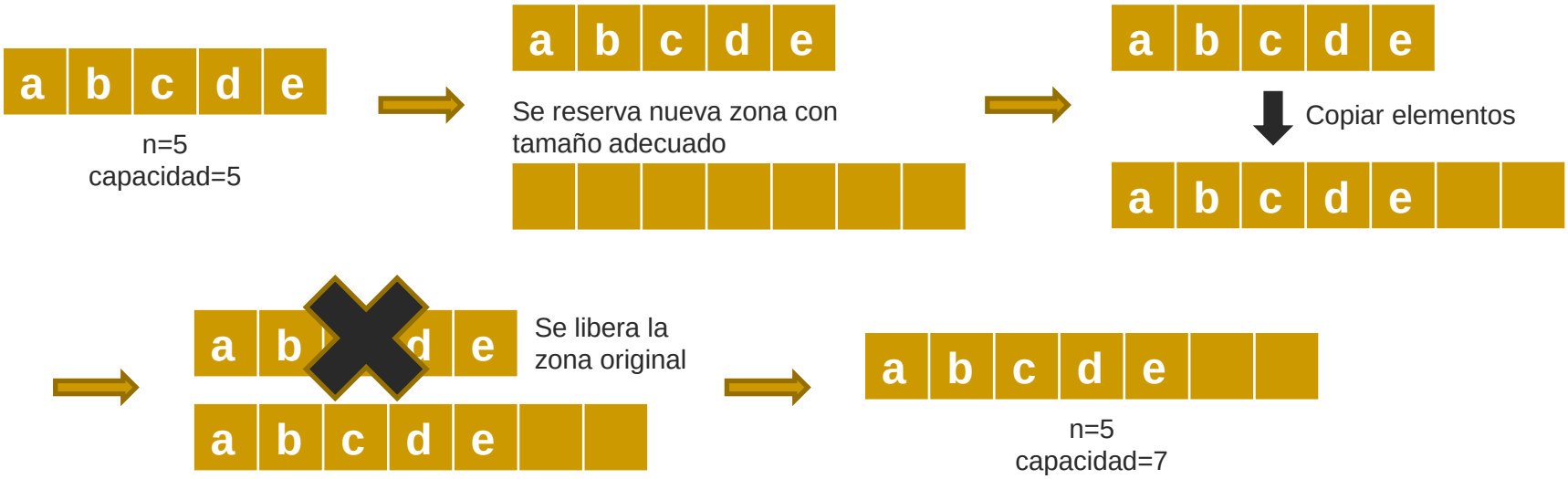
- Caso 1 (caso mejor): si hay memoria libre a continuación de la ya reservada:
  - Ejemplo: nuestra lista tiene n=5 y capacidad=5 (está llena). Queremos ampliar la capacidad en dos posiciones más

n=5
capacidad=5

n=5
capacidad=7

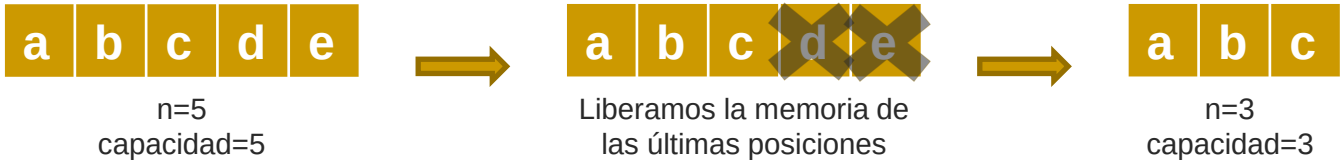
Contiguas: capacidad IV

- 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

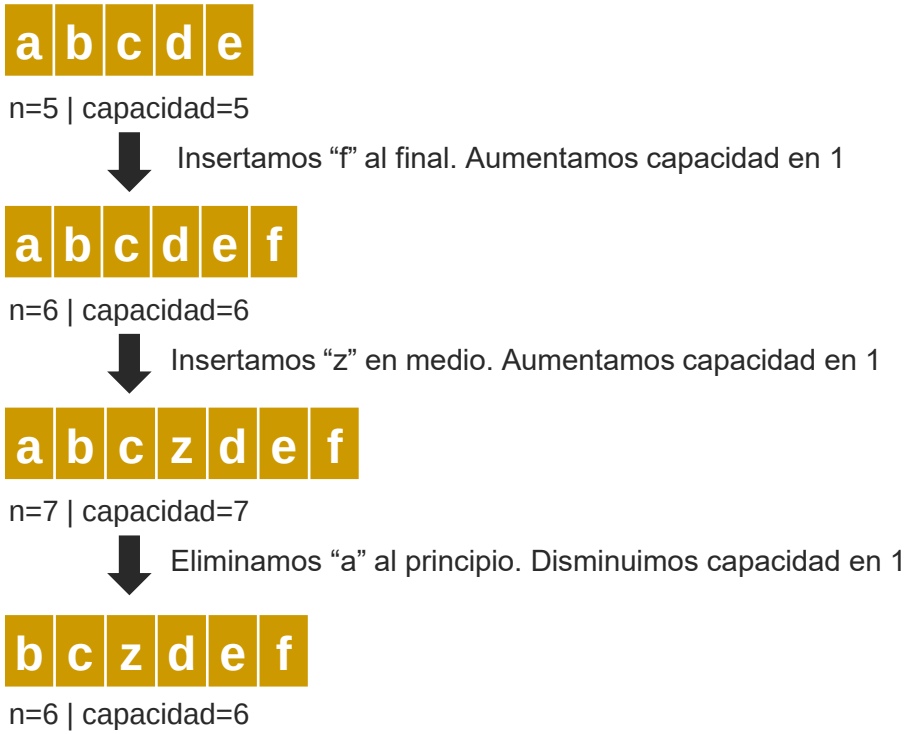
- Complejidad temporal de aumentar/disminuir capacidad:
  - Reservar/liberar memoria siempre es O(1), porque el sistema operativo se limita a anotar el principio y fin de la memoria reservada
  - Ampliar la capacidad es O(n) porque puede implicar copiar todos los n elementos a una nueva zona de memoria 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

- ¿Cuándo aumentar la capacidad?
  - 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…
    - 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, nos siga sobrando “incremento” posiciones, y así no tengamos que volver a aumentar/disminuir inmediatamente al volver a insertar/eliminar un nuevo elemento

Contiguas: capacidad VIII

- Ejemplo con lista de capacidad ajustada (EjercicioListaContigua1)
  - Cada vez que insertamos, tenemos que ampliar la capacidad, y esto es muy costoso en tiempo –O(n)-



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
    - 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) es O(n):
  - 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 temporal de eliminar en la posición “i” (peor caso) es O(n):
  - Desplazar a la izquierda desde i+1: O(n) en el peor caso (cuando i=0)
  - Disminuir la capacidad: O(1)

Contiguas: insertar y eliminar V

- Acceso a un elemento (para ver o editar) en la posición “i” es 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)

Análisis de contiguas

- Ventajas de las listas contiguas (EjercicioListaContigua2):
  - 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.
    - La búsqueda binaria de un elemento en un vector ordenado es buscar el elemento en medio. Si no lo encontramos, buscarlo en la primera mitad del vector si lo buscado es inferior al medio, o en la segunda mitad si lo buscado es superior al medio
      - El método podría devolver un booleano (true si encuentra lo buscado, false si no), o bien la posición en donde se ha encontrado el elemento (o -1 si no se encuentra).
  - 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... es lo habitual)
  - Insertar y eliminar elementos son operaciones muy costosas:
    - Complejidad temporal es O(n)

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 fácilmente 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
    - Si la lista tuviera sólo un nodo, el siguiente a ese nodo sería él mismo
  - 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

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)
- Practica ahora programando una lista enlazada simple (EjercicioEnlazadaSimple)

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 temporal extra durante el algoritmo (como sí lo hacían las listas contiguas al usar realloc)
  - 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
  - No podemos ordenar con algoritmos eficientes del orden de O(nlogn), como QuickSort, 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
  - Cuando el tamaño de la lista no sea muy grande
    - Será probable encontrar zonas de memoria contiguas
  - 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
  - Cuando haya muchas más inserciones/eliminaciones que accesos
    - Las inserciones/eliminaciones ahora son más eficientes que en las listas contiguas
  - Cuando no nos importe desperdiciar memoria
    - Pues la lista ocupa más, al tener que guardar los punteros para apuntar al nodo anterior y siguiente
  - Cuando no tengamos grandes porciones de memoria contiguas
    - Cada nodo se puede guardar en cualquier lugar de la memoria, no necesariamente contiguo al anterior
  - 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
    - NUNCA podremos usar listas contiguas si el tamaño de la lista es enorme, independientemente del resto de criterios
  - 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.
- Practica ahora con el EjercicioDobleCircular

Listas ordenadas I

- Podemos decidir mantener siempre ordenada una lista
  - Buscar:
    - Como la lista ya está ordenada, la búsqueda es O(logn) si es contigua (con búsqueda binaria) o O(n) si es enlazada (con búsqueda secuencial)
      - Si la búsqueda es secuencial, en cuanto encontremos el elemento ya paramos, y así reducimos las constantes ocultas
  - Insertar:
    - Como la lista ya está ordenada, para insertar sólo hace falta buscar su lugar apropiado (con el método buscar visto) y meterlo ahí para que la lista permanezca ordenada
      - La búsqueda será binaria o secuencial según el tipo de lista
  - Eliminar:
    - Como la lista ya está ordenada, buscamos el elemento a eliminar y, al quitarlo, la lista permanecerá ordenada
      - La búsqueda también será binaria o secuencial según el tipo de lista

Listas ordenadas II

- ¿Para qué nos sirve mantener siempre ordenada una lista?
  - Es útil para cuando recorremos muchas veces la lista (para imprimirla, por ejemplo) y queremos que el recorrido sea en orden
  - Es útil para cuando necesitamos obtener el menor o el mayor muy rápidamente (están al principio y/o al final).
    - Ej: colas de prioridad (las veremos más adelante)
  - Es útil cuando necesitamos hacer muchas búsquedas y pocas inserciones/eliminaciones, pues la búsqueda sobre una lista contigua ordenada es más eficiente que en una desordenada
    - Si la lista es contigua, la búsqueda es O(logn) en una lista ordenada, pero O(n) si está desordenada
    - Si la lista es enlazada, la búsqueda es O(n) tanto si está ordenada como si no

Listas ordenadas III

- A cambio de tener la lista siempre ordenada, hacemos más trabajo para insertar y para eliminar
  - Insertar en una lista contigua desordenada:
    - Insertamos el elemento en la posición especificada como parámetro: O(n), pues recordemos que es posible que haya que desplazar a la derecha los elementos siguientes
    - Total: O(n)
  - Insertar en una lista contigua ordenada:
    - Buscamos (con búsqueda binaria) la posición en donde insertar el elemento: O(logn)
    - Insertamos en la posición encontrada: O(n)
    - Total: O(n), pero las constantes ocultas son mayores que si la lista está desordenada, pues aquí trabajamos más (tenemos que buscar la posición primero)
  - Insertar en una lista enlazada desordenada:
    - Buscamos el nodo de la posición especificada como parámetro: O(n), pues recordemos que hay que recorrer la lista
    - Insertamos el elemento entre ese nodo y el anterior: O(1), pues sólo hay que cambiar punteros
    - Total: O(n)
  - Insertar en una lista enlazada ordenada:
    - Buscamos (con búsqueda secuencial) la posición en donde insertar el elemento: O(n)
    - Metemos el elemento ahí: O(1), pues sólo hay que cambiar punteros
    - Total: O(n), con constantes ocultas iguales que si la lista enlazara estuviera desordenada
  - Asintóticamente, por lo tanto, tenemos O(n) tanto si la lista se mantiene ordenada como si no, pero observamos que hacemos más trabajo en las listas contiguas en el caso de querer mantenerla ordenada (las constantes ocultas son mayores)
  - Casos similares para eliminar, con similares soluciones
- Practica ahora, programando una lista contigua ordenada (EjercicioOrdenada)

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)
    - Pop implica devolver el elemento y a continuación eliminarlo
    - 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

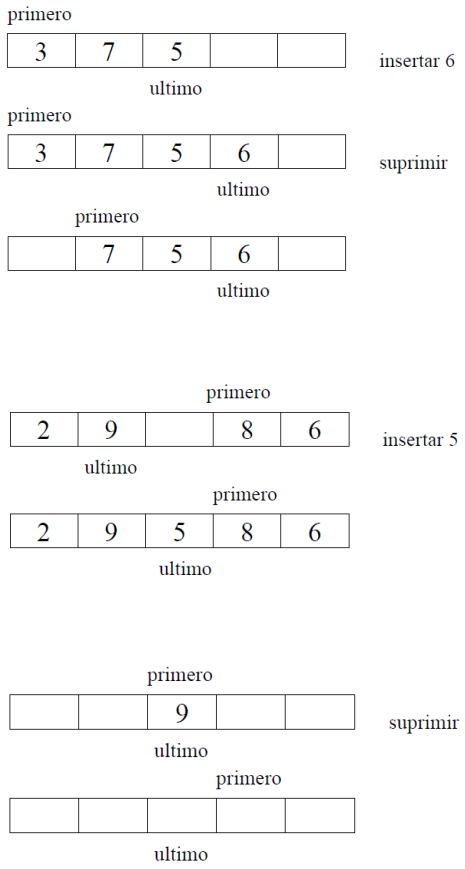
- Implementación de pilas. El objetivo es que push() y pop() sean O(1)
  - A partir de una lista contigua (suponiendo que no hay que incrementar la capacidad)
    - La cima es el último elemento, para no tener que mover todos los elementos cada vez que hacemos push() o pop()
  - A partir de una lista 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 atributo “n”, no las posiciones de “primero” y “último”
  - Ejercicio sugerido: implementa una cola con un vector circular



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(), metemos el elemento de tal modo que la lista siga estando ordenada. Como vimos, insertar en una lista ordenada es O(n) tanto en listas enlazadas como contiguas.
    - Hacemos pop() por donde nos sea más fácil, para conseguir O(1). Ej: si la lista está ordenada de menor a mayor prioridad, haremos pop() por el final.
  - Mediante una lista no ordenada
    - Cuando llega un elemento se hace push() por donde nos sea más fácil, para conseguir O(1)
    - Al hacer pop(), se busca el elemento deseado de forma secuencial, y por lo tanto en O(n)
  - ¿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 una lista contigua en el cual cada uno de sus elementos es a su vez una cola
    - Cada componente de la lista contigua 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
      - Para eliminar, habrá que ir recorriendo la lista contigua 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: 2+3 en notación postfija es 23+
      - 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 (haciendo “push”) 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 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
  - Problemas de la recursividad que se solucionan al convertirlo en iterativo:
    - Hay lenguajes que no soportan la recursividad
    - Una versión iterativa puede ser más eficiente
    - Un programa recursivo requiere memoria 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
- Practica ahora con EjercicioPila

TAD Conjunto I

- Un conjunto es una colección de cero o más elementos:
  - Que no guardan ninguna organización entre sí (no hay ningún orden entre ellos)
    - Las listas tenían una organización secuencial
    - Los árboles y grafos tienen una organización no secuencial, pero la tienen
  - Y que no se pueden repetir
    - Un conjunto de personas no puede contener dos veces la misma persona

TAD Conjunto II

- Posibles operaciones del TAD Conjunto:
  - Poner un elemento nuevo en el conjunto
    - Precondición: el elemento no estaba previamente en el conjunto (no se admite la repetición)
    - El orden de inserción no es importante, pues no hay orden interno entre los elementos
      - Ej: si el conjunto se implementa con una lista, dará igual meter el nuevo elemento por el principio o el final
  - Eliminar un elemento
    - Precondición: el elemento existe en el conjunto
    - Internamente, se buscará el elemento y a continuación se eliminará
  - Buscar un elemento
    - Saber si está o no el elemento
    - No nos puede devolver la posición del elemento dentro del conjunto, pues el concepto de “posición” no existe en un Conjunto
  - Obtener el tamaño actual, interseccionar con otro conjunto, unir con otro conjunto, saber si es un subconjunto de otro, etc.

TAD Conjunto III

- Posibles implementaciones del TAD Conjunto:
  - Mediante un array booleano tan grande como posibles elementos puede llegar a tener el conjunto
    - Si el conjunto tiene un elemento, se marca con verdadero la posición del array asociada a ese elemento. Si no, se marca como falso
  - Mediante una lista
    - Contendrá, en cualquier orden, todos los elementos que tiene actualmente el conjunto
  - Mediante una tabla hash
    - Es la implementación más eficiente en tiempo
    - Lo veremos en el resto del tema
    - Las tablas hash también se utilizan para implementar bases de datos

Tablas hash I

- La operación más importante de un conjunto o una lista (o base de datos) es la búsqueda
  - En general, hacemos muchísimas más búsquedas que inserciones/eliminaciones
    - Pensemos en Internet (una inmensa base de datos no organizada) y sus “buscadores”
    - Los conjuntos generalmente sirven para saber si un elemento pertenece o no pertenece a él, lo cual es realmente una búsqueda
  - La razón de ser de las bases de datos o conjuntos es para que mucha gente pueda buscar en ellas
  - Las inserciones/eliminaciones masivas se suelen hacer al principio de la vida útil de la base de datos o el conjunto
    - Después la base de datos se vuelve “estable” y las búsquedas superan a las inserciones/eliminaciones en un factor de 100 o incluso mayor
- Por lo tanto es muy importante optimizar todo lo que podamos el tiempo de las búsquedas

Tablas hash II

- Hasta ahora, las mejores búsquedas tienen una complejidad de O(logn) si se realizan sobre listas contiguas ordenadas
  - Gracias a la búsqueda binaria
- ¿Podríamos mejorar aún más este rendimiento?
  - A simple vista parece difícil, ya que tendríamos que conseguir búsquedas en O(1)
- Las tablas hash son un tipo especial de listas contiguas que nos permiten hacer búsquedas en O(1) o muy próximo a este valor
  - Por lo tanto, todas las bases de datos modernas se implementan con tablas hash
  - A cambio de conseguir O(1), las tablas hash pueden desperdiciar gran cantidad de memoria

Tablas hash III

- Una tabla hash es una lista con las siguientes características:
  - Generalmente se almacena de forma contigua
  - Los elementos de la lista están desordenados
  - Cada elemento es un par: (clave, valor)
    - La clave es una información que identifica al elemento de forma unívoca, ningún otro elemento puede tener la misma clave
      - Ej: DNI de una persona
    - El valor puede ser todo lo complicado que queramos y tener a su vez muchos campos
      - Ej: el resto de información de una persona: nombre, apellidos, fecha de nacimiento, etc.

Tablas hash IV

- Otros nombres de una tabla hash:
  - Tabla (a secas)
  - Diccionarios
    - La clave es la palabra y el valor es su definición
  - Aplicaciones
  - Mapas (maps)
  - Memorias asociativas

Tablas hash V

- Vectores vs tablas hash
  - En los vectores (listas contiguas), se accede a los elementos (“direccionamiento”) mediante su posición en el vector
  - En las tablas hash, el direccionamiento de sus elementos es mediante su clave
    - No sabemos ni nos interesa en qué posición interna de la tabla hash está almacenada una persona…
    - En su lugar, accedemos a la persona mediante su clave, mediante su DNI
      - Por lo tanto, decimos: “Dame la persona cuyo DNI es XXX” en vez de “Dame la persona que está en la posición YYY”
  - Por lo tanto, en los vectores el direccionamiento (la clave) es un entero (la posición que ocupa) y el valor es lo que se guarda en esa posición, mientras que en las tablas hash la clave es cualquier tipo de dato (ej: DNI, un string, etc.) y no tiene nada que ver con la posición que se ocupa
    - Hay algunos lenguajes de programación en los cuales existen tablas hash de forma nativa (ej: PERL). No es el caso de C/C++.

Operaciones de tablas hash I

- Las operaciones de una tabla hash (desde el punto de vista del usuario de la tabla, no del programador) son las siguientes:
  - Ver si un elemento existe
    - El usuario de la tabla proporciona la clave del elemento
    - La tabla busca internamente un elemento con esa clave.
    - Conseguiremos que esta búsqueda sea O(1) o muy próximo a este valor
    - Si lo encuentra, devuelve verdadero. Falso en caso contrario
  - Obtener un elemento
    - Es la operación más importante
    - Precondición: el elemento existe en la tabla
    - Se da la clave del elemento, la tabla lo busca internamente y, una vez encontrado, nos devuelve su valor asociado
    - No existe la operación de obtener un elemento por posición
      - Ni, en general, operaciones para las que sea necesario el número de posición. Por lo tanto no existe recuperar el primero, o el último, o el siguiente, o el anterior…

Operaciones de tablas hash II

- Insertar un nuevo elemento
  - Necesitamos dar la clave y el valor del nuevo elemento
  - Precondición: ningún elemento con esa misma clave existe ya en la tabla
  - La tabla guarda ambos valores en una posición que no conocemos ni nos interesa
- Eliminar un elemento existente
  - Se da la clave del elemento
  - Precondición: el elemento existe en la tabla
  - La tabla busca internamente el elemento a partir de su clave
  - Una vez encontrado, la tabla elimina el elemento entero (clave y valor)

Operaciones de tablas hash III

- Modificar un elemento existente
  - Se da la clave del elemento y su nuevo valor
  - Precondición: el elemento existe en la tabla
  - La tabla busca internamente el elemento a partir de su clave
  - Una vez encontrado, la tabla modifica el valor del elemento (pero no su clave) a su nuevo valor
- Otras operaciones secundarias:
  - Comprobar si está llena o vacía
  - Obtener el número de elementos (n)
  - Etc…

Conjuntos en tablas hash

- ¿Cómo usaríamos una tabla hash para implementar un conjunto?
  - Los elementos del conjunto están unívocamente determinados mediante su clave
  - Si cada elemento del conjunto tiene más datos (ej: una Persona), esos datos son el valor de cada elemento
  - Si nuestro elemento sólo es su clave, sólo almacenamos las claves y el valor de cada elemento estará vacío
    - Ej: un conjunto de números enteros. Cada elemento es un número entero, cuya clave es el propio número entero

Implementación ideal de tablas hash I

- Si el rango de posibles claves fuera de 1 hasta n podríamos utilizar un vector de capacidad n
  - La posición en donde se guardaría cada elemento sería simplemente “clave-1”
  - La búsqueda sería O(1) siempre
  - En cada posición del vector hay que añadir un campo (un bit) que nos permita saber si la posición está ocupada o no (si no está ocupada, tiene basura)
  - El problema es que se desperdicia mucha memoria pues no siempre vamos a guardar en la tabla hash todos los elementos con todas sus posibles claves

Implementación ideal de tablas hash II

- Ejemplo:
  - Una empresa necesita una tabla hash para guardar sus 500 empleados
  - La clave de cada empleado es su DNI, ya que es una persona y cada persona se identifica unívocamente por su DNI
  - El valor de cada empleado es su nombre, apellidos, sueldo, fecha de nacimiento, número de seguridad social, etc.
  - El rango de posibles DNIs que existen es 100 millones
  - Por lo tanto necesitaríamos una tabla hash de capacidad para 100 millones de objetos de la clase Empleado más 100 millones de bits de ocupado.
  - Si un empleado tiene como DNI 13032823, lo guardaríamos en esa misma posición de la tabla y activamos el bit de ocupado
  - Cuando quisiéramos obtener al empleado con DNI 13032823, iríamos a esa posición de la tabla (O(1) porque es una lista contigua) y recuperaríamos el objeto entero guardado allí, siempre que el bit de esa posición indique que esa posición está ocupada
  - Para borrar al empleado con DNI 13032823, iríamos a esa posición y pondríamos el bit de dicha posición a falso (para indicar que esa posición está vacía y sólo tiene basura)
  - Desperdiciamos 100.000.000 – 500 = 99.999.500 posiciones de memoria

Hashing I

- Para intentar desperdiciar menos memoria, hacemos “hashing”
  - Consiste en crear lo que se llama una “función hash”
- En vez de tener un vector con una capacidad que cubra todo el rango posible de claves, disminuímos esta capacidad
  - Ya no podemos mapear directamente la clave con la posición en la tabla, porque no hay posiciones suficientes para todas las posibles claves
  - Por lo tanto, necesitamos crear una función (la “función hash”) que asigne a cada clave su posición en la tabla
  - Una función hash muy típica es utilizar el resto de la división entera de la clave entre la capacidad de la tabla, para una tabla cuyas posiciones van desde 0 hasta capacidad-1
    - Si la clave no es numérica (ej: un texto), se pueden utilizar otro tipo de funciones hash que transformen dicha clave no numérica en una posición dentro de la tabla

Hashing II

- Sigamos con el ejemplo de los empleados, pero teniendo en cuenta ahora que puede haber colisiones:
  - Recordemos que la empresa tiene aproximadamente 500 empleados
  - En vez de tener un vector con 100 millones de posiciones, tenemos un vector con 10.000 posiciones de capacidad
  - La función hash nos devolvería el resto de dividir el DNI entre 10.000
    - Ése número sería la posición en donde guardar el empleado con ese DNI
  - Si queremos insertar un empleado con DNI 13.032.823, lo guardamos en la posición 2.823 de la tabla (porque 13.032.823%10.000 = 2.823), poniendo a verdadero el bit “ocupado” de esa posición
  - Para eliminar o buscar, accederíamos también a la posición 2.823
  - Problema: contratamos a un nuevo empleado cuyo DNI es 78.502.823… ¡le correspondería la misma posición de la tabla, pero ésta ya está ocupada! Se ha producido una “colisión”
- Practiquemos ahora haciendo una tabla hash que guarde estudiantes de una universidad, y en donde se exige que no haya colisiones (EjercicioSinColision)

Hashing III

- Denominaremos “hashing” al proceso mediante el cual transformamos una clave en una posición
- A la función hash le exigiremos que:
  - Sea una operación rápidamente calculable, es decir con complejidad temporal O(1)
  - Distribuya de manera bastante uniforme los elementos sobre la tabla, para disminuir la probabilidad de colisiones
    - Si la función hash llevara a la misma posición muchas claves diferentes, se incrementa la posibilidad de que dos elementos distintos se intenten guardar en la misma posición…
      - … es decir, se incrementa la probabilidad de colisión
- Obviamente, si el tamaño de la tabla es menor que el número posible de claves, varias claves distintas tendrán asociada la misma posición (puede producirse una “colisión”)

Colisiones I

- Colisión:
  - Cuando, al intentar meter un nuevo elemento con una determinada clave, la función hash de esa nueva clave nos da una posición ya ocupada por otro elemento que existía previamente en la tabla
  - Es decir: a dos elementos distintos le corresponde la misma posición en la tabla

Colisiones II

- Probabilidad de colisión:
  - Si la tabla hash tiene tanta capacidad como posibles claves, nunca se producen colisiones (“implementación ideal”)
  - A medida que vamos disminuyendo la capacidad de la tabla, aumenta la probabilidad de colisiones, pues a más cantidad de claves distintas les toca la misma posición
  - Una mala función hash (que no distribuya uniformemente) también puede aumentar la probabilidad de colisiones
    - Ej: coger los primeros dígitos del DNI en vez de los últimos, sabiendo que los DNI de las personas de una misma provincia (ej: la provincia en la que reside la empresa) empiezan por los mismos dígitos.

Colisiones III

- ¿Cómo resolvemos las colisiones? El manejador de colisiones será el mecanismo encargado de hacerlo
- Dos tipos de manejador de colisiones:
  - Dispersión cerrada. También llamado direccionamiento abierto
  - Dispersión abierta. También llamado encadenamiento separado

Dispersión cerrada I

- Dispersión cerrada:
  - Si hay colisión, se busca un lugar de la tabla no ocupado y ahí se inserta el nuevo elemento
  - Se denomina:
    - Dispersión cerrada porque todos los elementos se almacenan dentro de la tabla
    - Direccionamiento abierto porque, a pesar de que asociamos una posición a cada clave, el elemento puede acabar en otra posición
  - Tendremos diversas variantes en función de cómo busquemos otra posición libre

Dispersión cerrada II

- El método más simple de la dispersión cerrada se denomina “exploración lineal”
  - Existen otros métodos más avanzados como “exploración cuadrática” o “desplazamiento cociente” (no los vemos)
- La exploración lineal consiste en mirar en la siguiente posición hasta que encontremos una libre
  - La siguiente a la última es la primera
  - Así, para buscar un elemento iremos a la posición que le toca, y hasta que le encontremos o la celda esté vacía (o hasta que hayamos recorrido toda la tabla si está llena), iremos a la siguiente

Dispersión cerrada III

- Exploración lineal (cont.):
  - Insertar: basta con encontrar un hueco a partir de la posición que le toca
  - Eliminar:
    - Ya no basta con marcar la casilla como vacía, como cuando no teníamos colisiones
    - Necesitamos también un bit que nos indique si la casilla, aunque esté vacía, ha sido previamente borrada
      - Ahora vemos ejemplos que ilustren esto
  - Buscar:
    - Iremos a la posición que, según la función hash, nos toca. Si la clave que buscamos no está ahí, iremos avanzando al igual que para insertar… pero en vez de parar en la primera casilla vacía, pararemos en la primera casilla vacía no borrada. Es decir, las casillas borradas y vacías nos las saltaremos y seguiremos buscando

Dispersión cerrada IV

- Ejemplo de exploración lineal:
  - Tenemos una función hash que calcula el módulo de la clave entre la capacidad de la tabla
  - Tenemos una tabla de capacidad 4 inicialmente vacía
  - Insertamos un elemento con clave 40
    - Su posición “ideal” (la que nos da la función hash) es 0, porque 40%4=0
    - Como la posición 0 no está ocupada (miramos el flag “ocupado”), metemos ahí el elemento. Ponemos el bit de ocupado a verdadero
  - Insertamos un elemento con clave 104
    - Su posición ideal es 0, porque 104%4=0
    - Como la posición 0 está ocupada (lo vemos en el bit “ocupado”), miramos la siguiente (la posición 1)
    - Como la posición 1 no está ocupada, lo metemos ahí. Ponemos el bit de ocupado a verdadero.

B?	Oc?	Clave	Valor
No	No	Basura	Basura
No	No	Basura	Basura
No	No	Basura	Basura
No	No	Basura	Basura

Tabla hash de capacidad 4 inicialmente vacía

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	No	Basura	Basura
No	No	Basura	Basura
No	No	Basura	Basura

Insertamos elemento con clave 40

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
No	No	Basura	Basura
No	No	Basura	Basura

Insertamos elemento con clave 104

Dispersión cerrada V

- Ejemplo (cont.):
  - ¿Qué ocurre si ahora quiero buscar el 80, ya sea para ver si está en la tabla, o para devolver su valor, o para borrarlo?
    - Su posición ideal es la 0, porque 80%4=0
    - La posición 0 está ocupada, y la clave guardada ahí no es 80, por lo tanto el elemento buscado no está en la posición 0 y tenemos que mirar en la siguiente (la posición 1)
    - En la posición 1 el elemento buscado tampoco está. Por lo tanto miramos en la posición 2.
    - La posición 2 está vacía y además nunca ha sido usada (si no, su bit de borrado estaría a verdadero). Podemos parar aquí y concluir que el elemento cuya clave es 80 no está en la lista.
      - Así nos evitamos tener que mirar todas las posiciones de la tabla

Dispersión cerrada VI

- Ejemplo (cont.): seguimos haciendo operaciones sobre la tabla
  - Insertamos un elemento con clave 121
    - Su posición ideal es 1, porque 121%4=1
    - Como la posición 1 está ocupada, miramos la siguiente (la posición 2)
    - Como la posición 2 no está ocupada, lo metemos ahí. Ponemos el bit de ocupado a verdadero
  - Insertamos un elemento con clave 20
    - Su posición ideal es 0, porque 20%4=0
    - Como la posición 0 está ocupada, miramos la siguiente (la posición 1)
    - Como la posición 1 está ocupada, miramos la siguiente (la posición 2)
    - Como la posición 2 está ocupada, miramos la siguiente (la posición 3)
    - Como la posición 3 no está ocupada, lo metemos ahí. Ponemos el bit de ocupado a verdadero
  - Borramos elemento con clave 121
    - Su posición ideal es 1, porque 121%4=1
    - Como la posición 1 está ocupada (para ello miramos el flag “ocupado”), entonces leemos la clave que hay en la posición 1
    - Como la clave guardada en la posición 1 no es 121 (es 104), entonces el elemento cuya clave es 121 no está en la posición 1. Por lo tanto, miramos la siguiente (la posición 2)
    - Como la posición 2 está ocupada, entonces leemos la clave que hay en la posición 2
    - La clave guardada en la posición 2 es 121, por lo tanto el elemento que queremos borrar está guardado en la posición 2
    - Borramos el elemento en la posición 2, poniendo el bit de ocupado a falso, y el bit de borrado a verdadero
    - Los campos de clave y valor de la posición 2 se quedan con basura

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
No	Sí	121	zzz
No	No	Basura	Basura

Insertamos elemento con clave 121

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
No	Sí	121	zzz
No	Sí	20	aaa

Insertamos elemento con clave 20

B?	Oc?	Clave	Valor
No	Sí	40	xxx
No	Sí	104	yyy
Sí	No	121 (basura)	zzz (basura)
No	Sí	20	aaa

Borramos elemento con clave 121

Dispersión cerrada VII

- Ejemplo (cont.):
  - ¿Qué ocurre si ahora quiero buscar el 20, ya sea para ver si está en la tabla, o para devolver su valor, o para borrarlo?
    - Su posición ideal es la 0, porque 20%4=0
    - Como en la 0 no está (porque está ocupada y además la clave guardada ahí no es 20), entonces miramos en la siguiente posición (la 1)
    - En la posición 1 tampoco está, por lo tanto miramos en la 2
    - La posición 2 está vacía… ¿nos detenemos aquí y creemos que el 20 no está? No nos deberíamos detener ante una casilla vacía que ha sido borrada, deberíamos seguir buscando y por lo tanto deberíamos pasar a la posición 3
      - Para eso sirve el bit de borrado
    - En la posición 3 encontramos el elemento buscado

Dispersión cerrada VIII

- Ejemplo (cont.):
  - ¿Qué ocurre si ahora quiero insertar el elemento cuya clave es 83?
    - Le toca idealmente la posición 3 porque 83%4=3. Pero la posición 3 está ocupada, por lo tanto miramos la siguiente. La siguiente a la posición 3 es la posición 0.
    - La posición 0 también está ocupada. Por lo tanto miramos la siguiente.
    - La posición 1 también está ocupada. Por lo tanto miramos la siguiente.
    - La posición 2 está libre. Por lo tanto lo metemos ahí y ponemos el bit de ocupado a verdadero
      - El bit de borrado no hace falta que lo toquemos, pues cuando la casilla está ocupada, el bit de borrado no se usa para nada
- Practica ahora la exploración lineal con el EjercicioDispersionCerrada

Dispersión cerrada IX

- Reestructuración de una tabla hash:
  - A medida que pasa el tiempo, la tabla hash va llenándose de celdas cuyo bit de borrado está activo
    - Cada vez la tabla hash es más “caótica”, más desordenada
  - Cuando hay muchas celdas borradas, la complejidad temporal de las búsquedas se va alejando de O(1) y acercándose a O(capacidad), siendo “capacidad” la capacidad de la tabla
    - … ya que recordemos que la búsqueda no para ante las celdas borradas
  - Por lo tanto, es necesario “reestructurar” la tabla hash cada cierto tiempo (ej: cada día por la noche, o los fines de semana…)
    - Consiste en “reagrupar” los datos de modo que queden lo más cerca posible de su posición “ideal”, según el método de exploración que tenga la tabla
      - Es como una “defragmentación” de un disco duro
    - Después de una reestructuración, no existirán celdas borradas

Dispersión cerrada X

- Algoritmos de reestructuración:
  - Sin gastar memoria extra:
    - Algoritmo complicado: “subir” cada elemento que no esté en su sitio ideal a la posición más cercana a dicho sitio ideal. Similar al algoritmo de “defragmentación” de un disco duro
  - Gastando memoria extra de forma temporal (prográmalo como ejercicio recomendado opcional):
    - Se crea una nueva tabla hash vacía de igual capacidad que la tabla antigua
    - Se va recorriendo secuencialmente la tabla hash antigua. Por cada celda en la que haya un elemento en la tabla antigua, se inserta dicho elemento en la tabla nueva en la posición en que le toque en la tabla nueva, llamando al método de inserción de la nueva tabla.
      - Obviamente, es muy posible que al elemento no le toque la misma posición que tenía en la tabla antigua
    - Una vez insertados todos los elementos de la tabla antigua en la tabla nueva, se libera la vieja y nos quedamos con la nueva
    - Complejidad temporal:
      - O(capacidad) en el mejor caso (es “capacidad” y no “n” porque tenemos que recorrer toda la tabla para examinar si cada posición está ocupada o no). Es cuando todas las inserciones en la tabla nueva son O(1)
      - O(capacidad^2) en el peor caso, cuando todas las inserciones en la tabla nueva son O(capacidad).

Complejidad de la dispersión cerrada I

- Lo importante en una tabla hash son las búsquedas, así pues cuando hablamos de complejidad temporal de una tabla hash nos referimos a la complejidad temporal de la búsqueda en ella
- Concluimos (no lo demostramos) que la complejidad temporal dependerá únicamente de lo “cargada” que esté la tabla hash
  - Es decir, de L = n/capacidad
  - Por lo tanto, mejor complejidad cuanto más memoria sobre en la tabla

Complejidad de la dispersión cerrada II

- Mejor caso: O(1) cuando el elemento está en su posición ideal
  - Ocurre cuando no se producen colisiones
  - Cuanto más cargada esté la tabla, más probabilidad de colisión
- Peor caso: O(capacidad) si llegáramos a tener que recorrer toda la tabla para llegar a encontrar (o no encontrar) el elemento
  - Ocurre cuando se producen muchas colisiones…
  - … es decir, cuando la tabla está muy cargada
- Caso medio: más próximo a O(1) cuando menos cargada está la tabla
  - No lo demostramos

Complejidad de la dispersión cerrada III

- Ejemplo de la influencia de la carga en el caso medio en dispersión cerrada (no lo demostramos):
  - T_E es el número medio de intentos para una búsqueda exitosa (el elemento está en la tabla)
  - T_S es el número medio de intentos para una búsqueda sin éxito (el elemento no está en la tabla)
  - L es la carga de la tabla: n/capacidad

	T_S	T_E
L =0.5	2	1.39
L =0.9	10	2.56

Dispersión abierta I

- En cada posición tenemos un conjunto de elementos
  - Por lo tanto, todos los elementos van a su posición “ideal”… se van metiendo en el conjunto que hay en dicha posición
- Denominación:
  - Se denomina dispersión abierta porque los elementos pueden estar fuera de la tabla
  - También se llama “Encadenamiento separado” porque el conjunto suele implementarse con una lista enlazada
    - Aunque lo suyo sería que lo implementásemos a su vez con otra tabla hash

Dispersión abierta II

- Características:
  - El número de elementos puede ser mayor que el tamaño de la tabla
  - Necesitamos memoria extra para el conjunto que hay en cada posición de la tabla
    - El conjunto además va creciendo a medida que vamos insertando elementos en esa posición de la tabla
  - La eliminación no da lugar a pérdida de rendimiento en la búsqueda, como sí pasaba en dispersión cerrada
    - Ahora ya no necesitamos bit de borrado
    - Por tanto tampoco necesitamos un algoritmo de reestructurar
  - Para minimizar el impacto de las colisiones, se utilizan dos funciones de dispersión diferentes (o una que devuelve dos posiciones) e insertamos en la posición que tenga menos elementos guardados en ese momento
    - Implicará también buscar en esas dos posiciones cada vez que busquemos un elemento

Complejidad de la dispersión abierta

- Con dispersión abierta, todos los elementos van siempre al conjunto que hay en su posición ideal (supongamos que el conjunto está implementado con una lista)
- Mejor caso: O(1)
  - Cuando en la posición del elemento hay una lista de un solo elemento…
  - … por lo tanto no hay que buscar en dicha lista
- Peor caso: O(n)
  - Cuando todos los elementos de la tabla (n) están en la misma posición
  - Tendremos una lista de n elementos
  - Tendremos que buscar dentro de dicha lista. Si la lista es enlazada, tendremos que implementar la búsqueda secuencial, que es O(n)
- Caso medio cuando n>capacidad: O(n/capacidad) = O(L)
  - Si los elementos están uniformemente distribuidos por la tabla hash tendremos en cada lista “n/capacidad” elementos, por lo tanto la búsqueda secuencial en dicha lista será O(n/capacidad) = O(L)
- Practiquemos ahora la programación de la dispersión abierta con el EjercicioDispersionAbierta