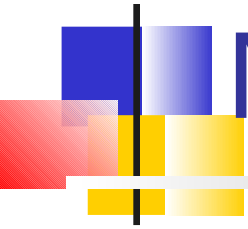
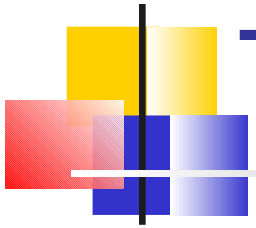


Multiestructuras

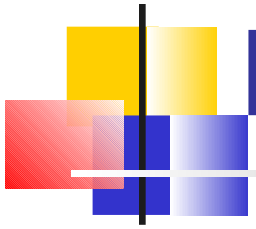
Teórico Programación 3

Curso 2008



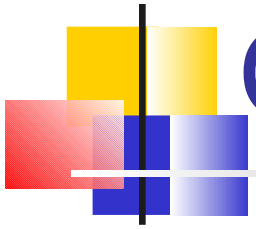
Temario

- Motivación
- Caso de Estudio I
  - Solución I
  - Solución II
  - Solución III
  - Solución IV
- Caso de Estudio II
  - Solución I
  - Solución II
  - Solución III



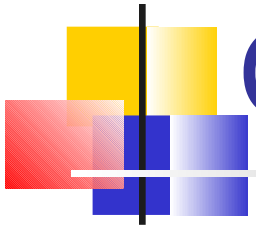
Motivación

- En general es posible utilizar distintas estructuras para modelar un mismo problema.
- En cada una de estas se resolverán eficientemente algunas operaciones y otras no.
- Nos va a interesar combinar varias estructuras de datos para resolver eficientemente la mayor cantidad de las operaciones que se deban realizar.



Caso de Estudio I

- Bedelía de Facultad, la cual debe llevar un registro de los estudiantes, cursos e inscripciones de estudiantes a cursos.
- Un estudiante puede estar inscripto a varios cursos y en un curso pueden haber muchos estudiantes inscriptos.
- Debemos poder realizar las siguientes consultas:
  - Dado un estudiante saber a que cursos está inscripto.
  - Dado un curso saber que estudiantes están inscriptos.



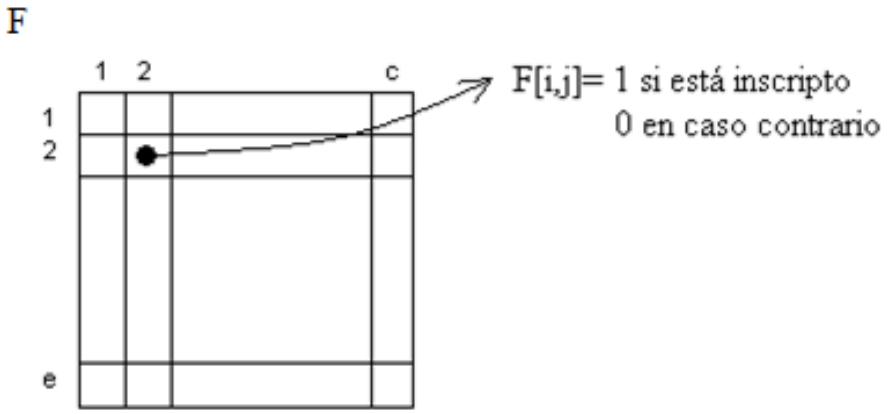
Caso de Estudio I

- En definitiva, nos interesa resolver eficientemente las siguientes operaciones:
  - *Inscribir*: Facultad x Estudiante x Curso → Facultad
  - *Desistir*: Facultad x Estudiante x Curso → Facultad
  - *Inscripto_A*: Facultad x Estudiante → Cursos
  - *Est_Inscriptos*: Facultad x Curso → Estudiantes
- Que estructura eligirían para resolver la realidad planteada?

Caso de Estudio I – Solución I

- Supongamos lo siguiente:
  - Cada estudiante se identifica con un valor en el rango 1..e.
  - Cada curso se identifica con un valor en el rango 1..c.

- Bajo las siguiente suposiciones podríamos elegir una matriz como estructura.



Caso de Estudio I – Solución I

- Con la estructura elegida las operaciones se resuelven de la siguiente manera:

  - *Inscribir*(F, e, c) => F[e,c]=1 O(1)
  - *Desistir*(F, e, c) => F[e,c]=0 O(1)
  - *Inscripto_A*(F, e) => recorrer la fila del estudiante O(c)
  - *Est_Inscriptos*(F, c) => recorrer la columna del curso O(e)

Caso de Estudio I – Sol. II

- La suposición anterior no refleja lo que es la realidad.
- Naturalmente los estudiantes se identifican a través de su CI, mientras que los cursos a través de un código alfanumérico.
- En este contexto, es posible seguir utilizando a la matriz como estructura de datos ?.
- La respuesta es SI, pueden utilizarse funciones de Hash tanto para estudiantes como para cursos.
  - E_h : CI → 1..e
  - C_h : Código → 1..c

Caso de Estudio I – Sol. II

- Sin considerar los eventuales problemas a los que nos enfrentamos al buscar funciones de hash, es posible mantener la matriz respetando los ordenes.

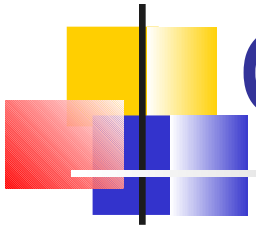
  - *Inscribir*(F, e, c) => $F[E_h(e), C_h(c)]=1$ O(1)
  - *Desistir*(F, e, c) => $F[E_h(e), C_h(c)]=0$ O(1)
  - *Inscripto_A*(F, e) => recorrer $F[E_h(e), j]$ $\forall j, 1 \leq j \leq c$ O(c)
  - *Est_Inscriptos*(F, c) => recorrer $F[i, C_h(c)]$ $\forall i, 1 \leq i \leq e$ O(e)

Caso de Estudio I – Sol. II

- Analizemos que sucede con la representación usando una matriz cuando se dispone de información sobre las cantidades de elementos.
  - a) 20.000 Estudiantes
  - b) 1.000 Cursos
  - c) cada estudiante se inscribe a 3 cursos en promedio

Caso de Estudio I – Sol. II

- Tamaño de Matríz = 20.000 x 1.000 = 20.000.000 de celdas
  - 20.000 x 3 = 60.000 tiene un 1 (0.3% info. relevante)
  - 99.7% tiene un 0 (info. NO relevante)
- La consulta *Est_Inscriptos* implica recorrer una columna de 20000 celdas para encontrar un promedio de 60 (20.000 x 0,3%) celdas con un 1. Situación análoga ocurre con la otra consulta.
- Los problemas anteriores se deben a que F es una matriz dispersa.
- Este tipo de matrices no son convenientes de usar debido al mal uso del recurso memoria que hacen.

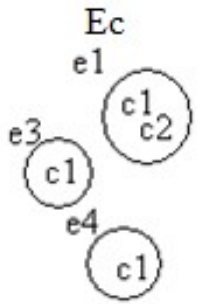


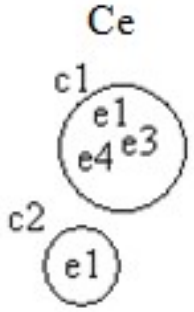
Caso de Estudio I – Sol. III

- Una mejor solución es disponer de una estructura que represente el conjunto I de celdas con un 1 en la matriz F.
- En esta nueva estructura las operaciones se resolverían:
  - *Inscribir* : implica agregar un elemento (pareja de estudiante y curso) al conjunto I
  - *Desistir* : implica eliminar un elemento del conjunto I.
- Seguimos teniendo problemas con las consultas ?
- SI, porque no queremos recorrer todo el conjunto I para resolverlas.
- Es necesario disponer de información adicional.

Caso de Estudio I – Sol. IV

- Para resolver *Inscripto_A* eficientemente sería necesario disponer de un conjunto E_c que contenga todos los cursos a los cuales un estudiante está inscripto.
- Lo mismo para *Est_Inscriptos* (conjunto C_e que contenga todos los estudiantes inscriptos a un curso).
- Si hiciéramos las siguientes inscripciones: *Inscribir*(F, e_1, c_1), *Inscribir*(F, e_1, c_2), *Inscribir*(F, e_3, c_1), *Inscribir*(F, e_4, c_1), tendríamos:





Caso de Estudio I – Sol. IV

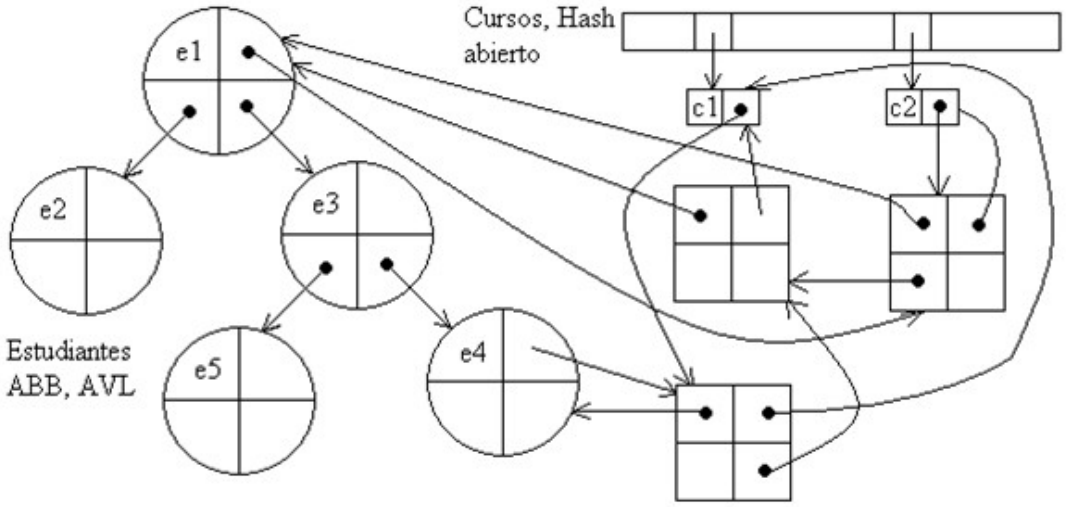
- Una forma de representar a los conjuntos E_c y C_e es a través de listas.
- Dado que no se conoce a priori la cantidad de cursos a los que un estudiante está inscripto ni viceversa, por cada estudiante se mantiene una lista con sus cursos, y por curso se mantiene una lista con sus estudiantes.

Estudiante	Curso
Siguiente curso al que está inscripto el estudiante	Siguiente estudiante inscripto al curso

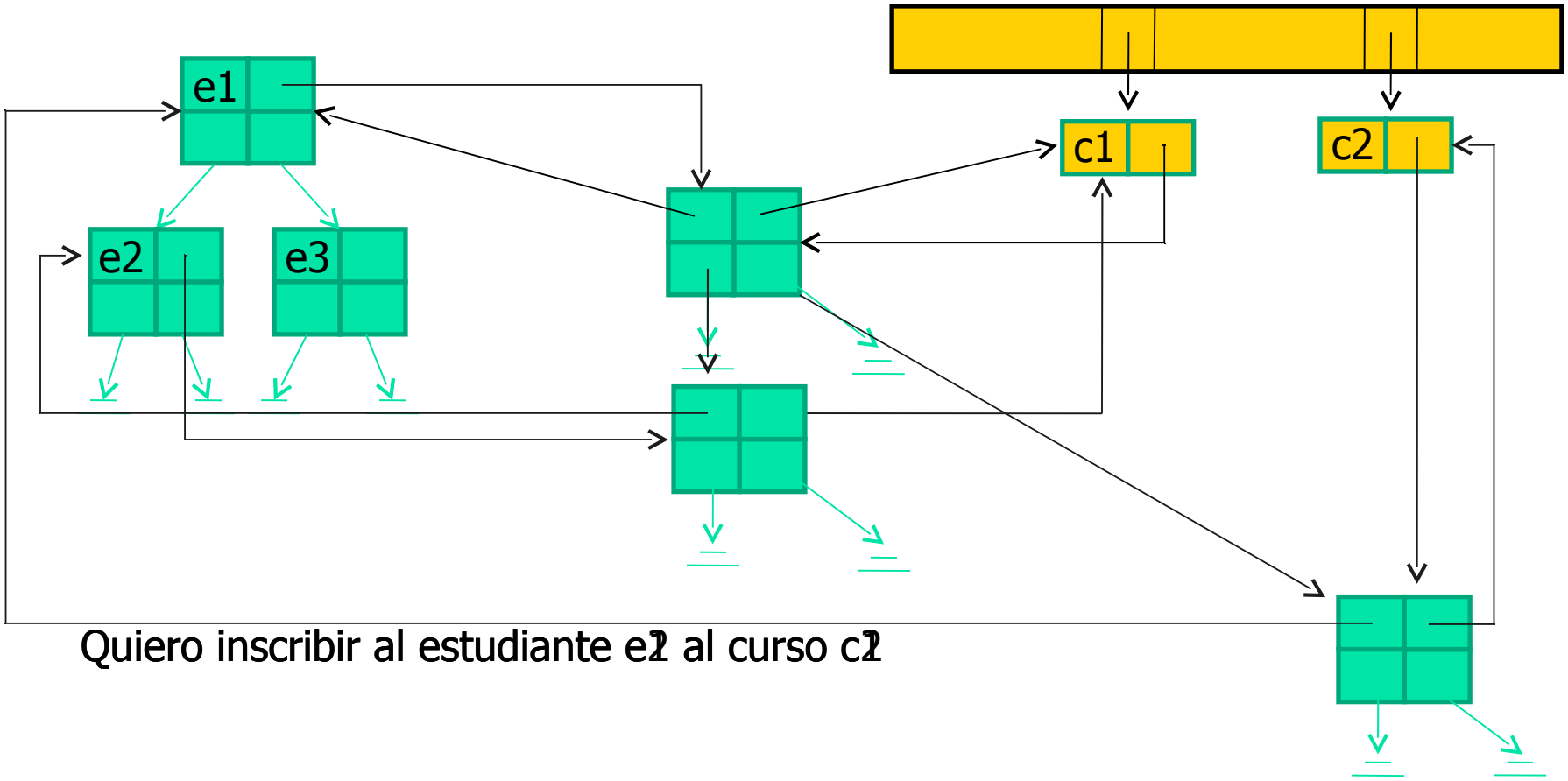
- Que problema tiene esta solución ?.
- Redundancia de Información.

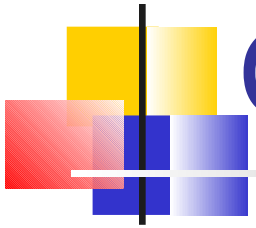
Caso de Estudio I – Sol. V

- Queremos lograr una solución eficiente y sin redundancia.
- Las estructuras posibles para estudiantes y cursos pueden ser varias, pero bajo las suposiciones hechas hasta el momento, una buena opción podría ser utilizar ABB (o AVL) para estudiantes y un hash abierto para los cursos.



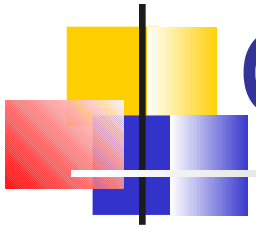
Caso de Estudio I – Sol. V





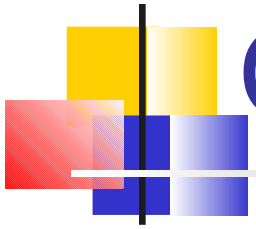
Caso de Estudio I – Sol. V

```
Inscribir(F,e,c)

//busca un estudiante en el ABB y retorna un puntero al estudiante
E = BuscarEstudiante(F,e)

//busca un curso en el hash y retorna un puntero al curso
C = BuscarCurso(F,c)

//Crea u nuevo registro con todos los punteros apuntando a NULL
N = CrearRegistro()
N->Estudiante = E
N->Curso = C

//la inserción se realiza adelante para no recorrer toda la lista
N-> Siguiente_estudiante_inscripto_al_curso = C->Estudiantes_del_curso
C->Estudiantes_del_curso = N

//la inserción se realiza adelante para no recorrer toda la lista
N -> Siguiente_curso_al_que_esta_inscripto_el_estudiante = E->Cursos_del_estudiante
E->Cursos_del_estudiante = N
```



Caso de Estudio II

- Se desea modelar un ranking de tenistas.
  - Cada tenista tiene asociado un valor que indica su posición en el ranking.
  - Los nuevos tenistas se agregan con el valor más alto de ranking.
  - Un tenista puede desafiar únicamente al tenista que se encuentra en el ranking inmediato superior al suyo, si le gana se intercambian las posiciones de ambos tenistas en el ranking.

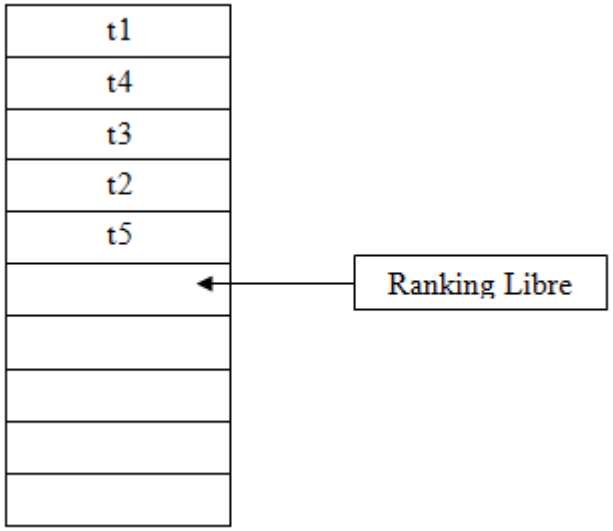


Caso de Estudio II

- Las operaciones que queremos modelar son:
  - *Agregar*(tenista t): agrega el tenista t con el ranking más alto (valor mas bajo representa al mejor tenista en el ranking)
  - tenista *Desafiar*(tenista t): retorna el tenista que ocupa la posición i-1 en el ranking, si el tenista t ocupa la posición i en el ranking (i>1)
  - *Cambiar*(int i): intercambia en el ranking los tenistas que ocupan las posiciones i e i-1 (i>1)
- Se supondrá que hay n tenistas.

Caso de Estudio II – Sol I

- Se utilizará un arreglo, en el cual en cada celda *i* se guarda el tenista t_i que ocupa la posición *i* en el ranking.

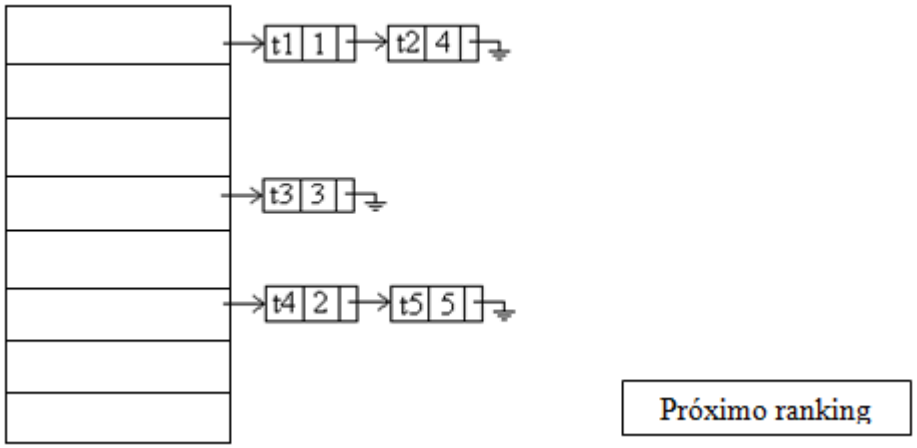


Caso de Estudio II – Sol I

- Con la estructura anterior las operaciones tienen los siguientes ordenes:
  - *Agregar*(tenista t): O(1), porque se agrega en Posición Libre
  - tenista *Desafiar*(tenista t): O(n), porque se debe recorrer todo el arreglo para encontrar al tenista t.
  - *Cambiar*(int i): O(1), porque se intercambian los elementos que ocupan las posiciones i e i-1

Caso de Estudio II – Sol II

- Se utilizará una tabla de hash para representar los tenistas.
  - Cada elemento del hash es una pareja formada por el tenista y su posición en el ranking.
  - Además, es necesario un indicador de cual es el próximo valor de ranking a utilizar.

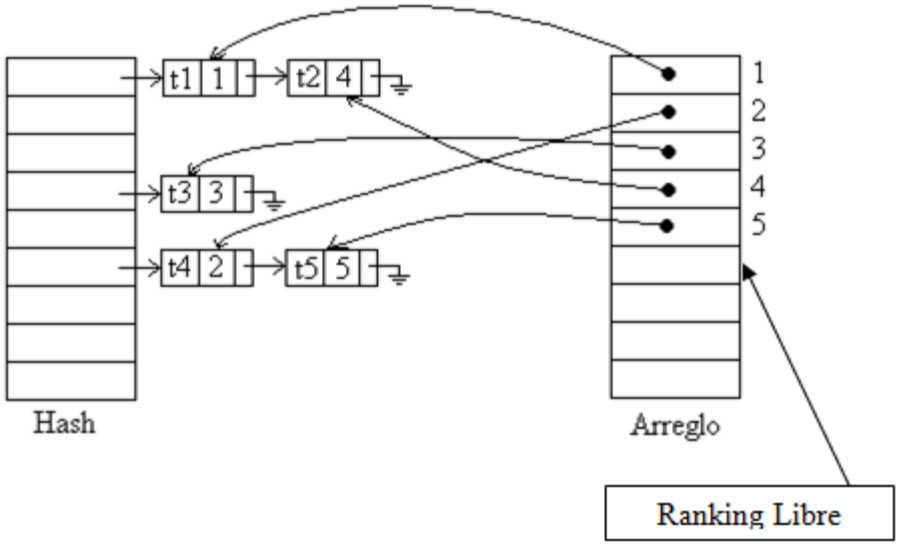


Caso de Estudio II – Sol II

- Con la estructura anterior las operaciones tienen los siguientes ordenes:
  - *Agregar*(tenista t): O(1)
  - tenista *Desafiar*(tenista t): O(n), porque es O(1) encontrar al tenista t y conocer su posición de ranking i, pero es O(n) recorrer el hash para encontrar al tenista a desafiar que ocupa la posición i-1 de ranking.
  - *Cambiar*(int i): O(n), porque se deben encontrar los tenistas que ocupan las posiciones i e i-1 para luego intercambiarlos.

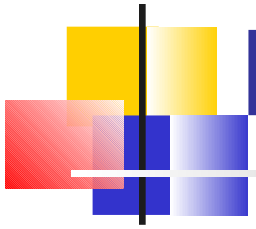
Caso de Estudio II – Sol III

- Con la estructura de hash la operación Cambiar es O(n), por lo cual el arreglo es mejor opción.
- Pero que sucede si se combinan ambas estructuras?



Caso de Estudio II – Sol III

- Los ordenes en el último caso serían:
  - ***Agregar*(tenista t):** O(1), agregar un elemento (tenista y Ranking Libre) al hash, apuntar desde la posición Ranking Libre del arreglo al elemento agregado en el hash, incrementar Ranking Libre.
  - **tenista *Desafiar*(tenista t):** O(1), buscar en el hash el tenista t y obtener su posición en el ranking, i, acceder desde el arreglo al elemento del hash apuntado por la celda i-1, retornar el tenista del elemento.
  - ***Cambiar*(int i):** O(1), acceder al elemento del hash apuntado por la celda i del arreglo y modificar su posición en el ranking a i-1, acceder al elemento del hash apuntado por la celda i -1 del arreglo y modificar su posición en el ranking a i, intercambiar los contenidos de las celdas i e i-1 en el arreglo.



Próxima Clase

- Seguiremos con Grafos.