ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

Gestión de una academia (1)

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Versión inicial

El grueso de esta asignatura consiste en hacer y hacer ejercicios. Son ejercicios que ponen en prácticas los TAD de este curso. Empezamos con un ejercicio que más tarde iremos ampliando.

Requisitos

- Academia que ofrece una serie de cursos.
- Cada curso tiene un límite de plazas.
- Operaciones soportadas:
  - Crear una academia vacía (sin cursos ni estudiantes).
  - Añadir un curso a la academia.
  - Eliminar un curso de la academia.
  - Matricular a un estudiante en un curso.
  - Saber el número de plazas libres de un curso.
  - Obtener un listado de personas matriculadas en un curso, ordenado alfabéticamente por apellido.

Cuando el número de estudiantes sobrepasa ese número ya no se permiten más en teoría.

Métricas de coste

- M = número de cursos total.
- NC = número de estudiantes máximo por curso.

Nos interesan estás métricas porque queremos saber de una Academia cuantos cursos se pueden hacer y de cada curso cuántos estudiantes están matriculados

Interfaz

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  Academia();
  void anyadir_curso(const std::string &nombre, int numero_plazas);
  void eliminar_curso(const Curso &curso);
  void matricular_en_curso(const Estudiante &est, const Curso &curso);
  int plazas_libres(const Curso &curso) const;
  std::vector<std::string> estudiantes_matriculados(const Curso &curso) const;

private:

  …
}
```

Identifican cada una de las entidades.

Representación

- Cada curso se identifica mediante su nombre.
- Cada estudiante se identifica mediante una cadena *"Apellidos, Nombre"*.
- Debemos almacenar el catálogo disponible de cursos. Para cada uno:
  - Número de plazas total.
  - Estudiantes matriculados. la colección de estudiantes.

NO es lo mejor ya que puede haber dos personas que se llamen igual. Mejor id o NIE o NIF.

Un ID de estudiante o DNI sería mejor

Colección de cursos

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  Academia();
  void anyadir_curso(nombre, numero_plazas);
  void eliminar_curso(curso);
  void matricular_en_curso(est, curso);
  int plazas_libres(curso);
  vector<...> estudiantes_matriculados(curso);

private:

  …
}
```

- ¿Qué TAD necesitamos para almacenar los cursos?
  - Lista.
  - Pila / cola / doble cola.
  - Conjunto.
  - Diccionario.
    - Por árboles binarios
    - Por tablas hash..
  - Multiconjunto.
  - ~~Multidiccionario.~~

Pero no va a haber claves duplicadas. Es decir, no van a haber cursos repetidos.

Necesitaremos una estructura clave-valor. La clave sería el nombre del curso. Y el valor toda la info necesaria relativa al curso.

Colección de cursos

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  Academia();
  void anyadir_curso(nombre, numero_plazas);
  void eliminar_curso(curso);
  void matricular_en_curso(est, curso);
  int plazas_libres(curso);
  vector<...> estudiantes_matriculados(curso);

private:

  …
}
```

- ¿Necesitamos recorrer los cursos en un determinado orden?
  - map — No es necesario el orden-
  - unordered_map

Utiliza tablas hash para el diccionario de cursos.

Nos da igual el orden de los cursos. El que si que nos importa es el de los estudiantes.

Colección de cursos: representación

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  …

private:
  struct InfoCurso {
    std::string nombre;
    int numero_plazas;
    ??? estudiantes;

    InfoCurso(const std::string &nombre,
              int numero_plazas);
  };

  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Nombre del curso

Numero plazas

estudiantes matriculados.

Nombres de curso a infocurso.

Colección de estudiantes

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  …

private:
  struct InfoCurso {
    std::string nombre;
    int numero_plazas;
    ??? estudiantes;

    InfoCurso(const std::string &nombre,
              int numero_plazas);
  };

  std::unordered_map<Curso, InfoCurso> cursos;
}
```

NO necesitamos una estructura clave-valor.
Tampoco vamos a necesitar el orden en el que los insertamos.

- ¿Qué TAD necesitamos para almacenar la colección de estudiantes?
  - ~~Lista~~.
  - ~~Pila / cola / doble cola~~.
  - Conjunto.
  - ~~Diccionario~~.
  - ~~Multiconjunto~~.
  - ~~Multidiccionario~~.

NO HAY ESTUDIANTES DUPLICADOS.

Colección de estudiantes

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  …

private:
  struct InfoCurso {
    std::string nombre;
    int numero_plazas;
    ??? estudiantes;

    InfoCurso(const std::string &nombre,
              int numero_plazas);
  };

  std::unordered_map<Curso, InfoCurso> cursos;
}
```

- ¿Necesitamos mantener los estudiantes matriculados en un determinado orden?
  - set
  - ~~unordered_set~~

Colección de estudiantes: representación

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
  …

private:
  struct InfoCurso {
    std::string nombre;
    int numero_plazas;
    std::set<Estudiante> estudiantes;

    InfoCurso(const std::string &nombre,
              int numero_plazas);
  };

  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Añadir un curso

```
class Academia {
public:

  void anyadir_curso(const std::string &nombre, int numero_plazas) {
    if (cursos.contains(nombre)) {
      throw std::domain_error("curso ya existente");
    }
    cursos.insert({nombre, InfoCurso(nombre, numero_plazas)});
  }

  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Lanza error

O(1)

Inserta el nombre

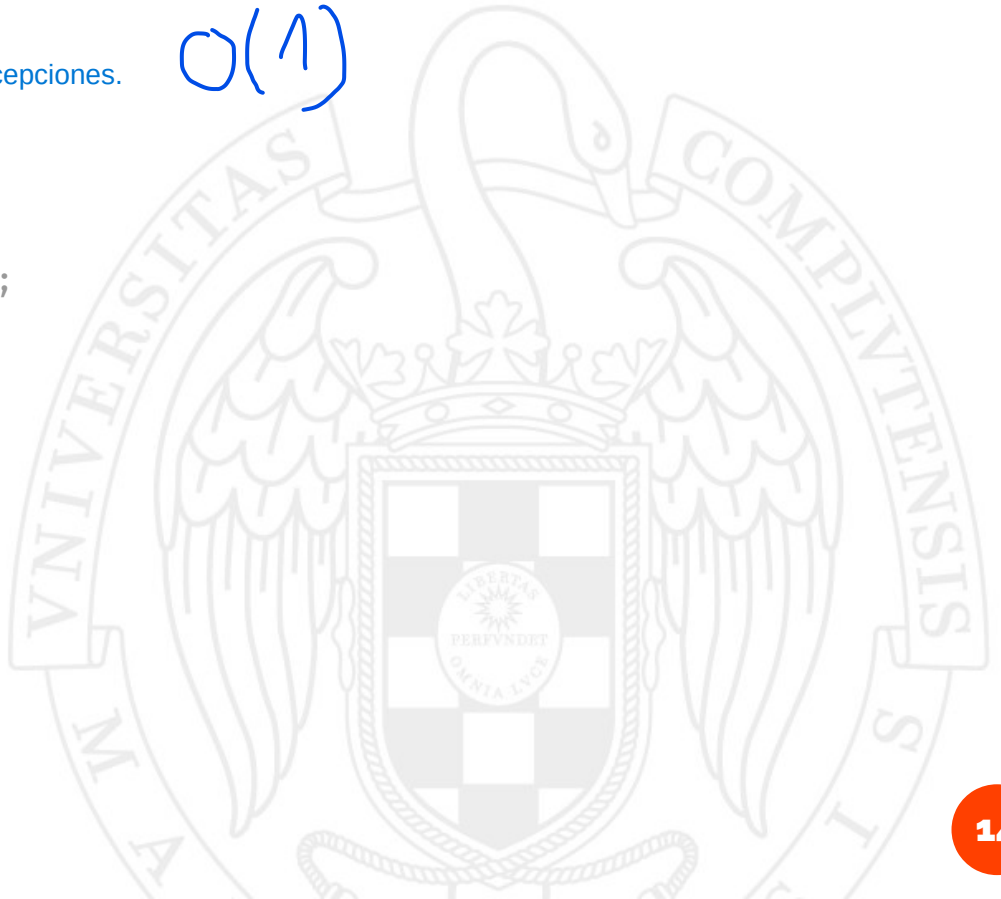
O(1)

Eliminar un curso

```
class Academia {
public:

  void eliminar_curso(const Curso &curso) {
    cursos.erase(curso);
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

No es necesario manejar excepciones.

O(1)



Matrícula en un curso

```
class Academia {
public:
  void matricular_en_curso(const Estudiante &est, const Curso &curso) {
    if (!cursos.contains(curso)) {
      throw std::domain_error("curso no existente");
    }
    InfoCurso &info_curso = cursos.at(curso);
    if (info_curso.estudiantes.contains(est)) {
      throw std::domain_error("estudiante ya matriculado");
    }

    if (info_curso.estudiantes.size() ≥ info_curso.numero_plazas) {
      throw std::domain_error("no hay plazas disponibles");
    }

    info_curso.estudiantes.insert(est);
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Si no existe el curso

si existe el curso, ver la informacion de ese curso para comprobar el num_plazas y si el estudiante esta o no matriculado.

COSTE TOTAL ES LOGARITMICO RESPECTO AL NÚMERO DE ESTUDIANTES.

Matrícula en un curso

```
class Academia {
public:
  void matricular_en_curso(const Estudiante &est, const Curso &curso) {
    auto it = cursos.find(curso);
    if (it == cursos.end()) {
      throw std::domain_error("curso no existente");
    }
    InfoCurso &info_curso = it->second;
    if (info_curso.estudiantes.contains(est)) {
      throw std::domain_error("estudiante ya matriculado");
    }

    if (info_curso.estudiantes.size() ≥ info_curso.numero_plazas) {
      throw std::domain_error("no hay plazas disponibles");
    }

    info_curso.estudiantes.insert(est);
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

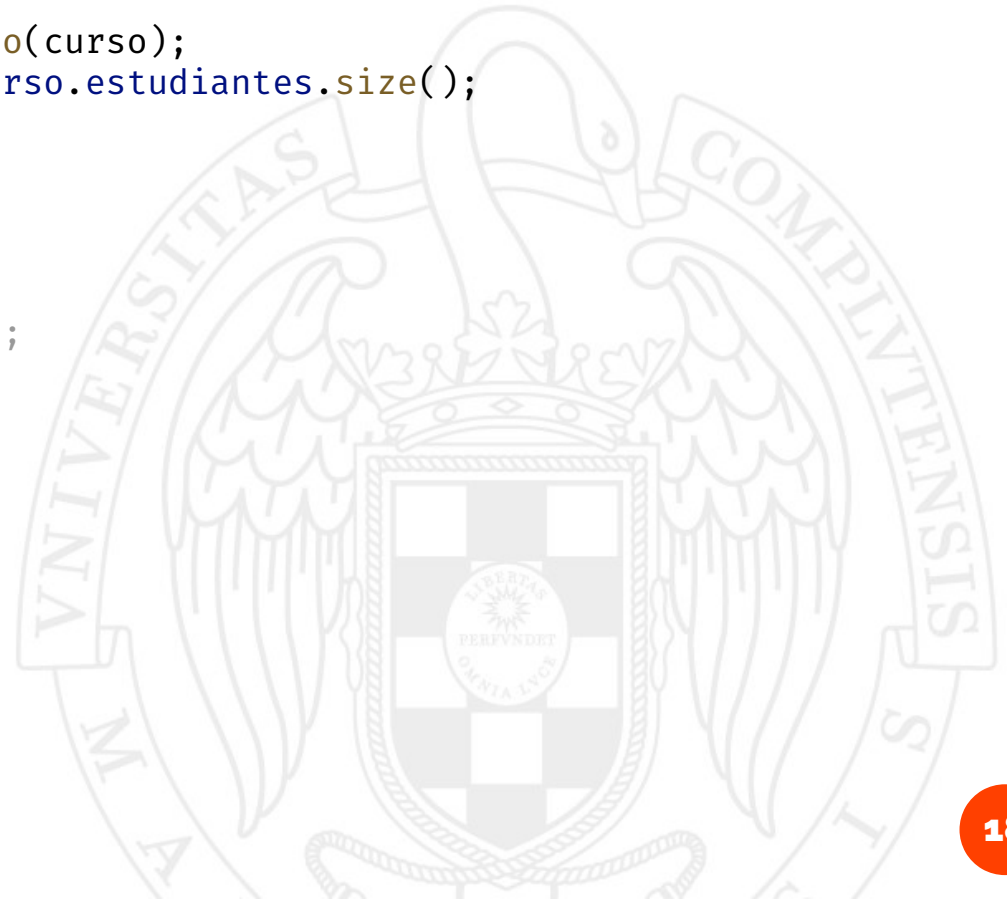
buscar_curso

Matrícula en un curso

```
class Academia {
public:

  void matricular_en_curso(const Estudiante &est, const Curso &curso) {
    InfoCurso &info_curso = buscar_curso(curso);
    if (info_curso.estudiantes.contains(est)) {
      throw std::domain_error("estudiante ya matriculado");
    }

    if (info_curso.estudiantes.size() ⩾ info_curso.numero_plazas) {
      throw std::domain_error("no hay plazas disponibles");
    }

    info_curso.estudiantes.insert(est);
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Número de plazas disponibles

```
class Academia {
public:

  int plazas_libres(const Curso &curso) {
    const InfoCurso &info_curso = buscar_curso(curso);
    return info_curso.numero_plazas - info_curso.estudiantes.size();
  }

  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```



Estudiantes matriculados

```
class Academia {
public:

  std::vector<std::string> estudiantes_matriculados(const Curso &curso) const {
    const InfoCurso &info_curso = buscar_curso(curso);
    std::vector<std::string> result;
    for (const Estudiante &est: info_curso.estudiantes) {
      result.push_back(est);
    }

    return result;
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Estudiantes matriculados

```
class Academia {
public:

  std::vector<std::string> estudiantes_matriculados(const Curso &curso) const {
    const InfoCurso &info_curso = buscar_curso(curso);
    std::vector<std::string> result;
    std::copy(info_curso.estudiantes.begin(), info_curso.estudiantes.end(),
                  std::back_insert_iterator<std::vector<std::string>>(result));
    return result;
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```