ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

Gestión de una academia (2)

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

Vamos a extender nuestro TAD para poder manejar la academia con el fin de añadir un registro de estudiante.

Registro de estudiantes

Requisitos

- Queremos mantener un registro de estudiantes.
- Antes de matricularse en un curso, los estudiantes han de estar registrados en la academia.
- Los estudiantes no se identificarán por nombre y apellidos. El identificador de un estudiante es su número de documento de identidad (NIF, NIE, etc.)
- Operaciones soportadas:
  - Añadir un estudiante a la academia.
  - Obtener un listado (ordenado alfabéticamente) de los cursos en los que está matriculado un estudiante.

Métricas de coste

- M = número de cursos total.
- N = número de estudiantes total. NUEVO RESPECTO DEL ANTERIOR
- NC = número de estudiantes máximo por curso.
- MC = número de cursos máximo en el que se matricula un estudiante.

NUEVO RESPECTO DEL ANTERIOR

Registro de estudiantes

Pero sigue siendo una cadena. Solo que ahora, la cadena representa el NIF en vez del nombre y apellido del estudiante.

Ahora representa el NIF

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

class Academia {
public:
  …

private:
  struct InfoCurso { … };

  struct InfoEstudiante {
    Estudiante id_est;
    std::string nombre;
    std::string apellidos;

    InfoEstudiante(const Estudiante &id_est,
                   const std::string &nombre, const std::string &apellidos);
  };

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

diccionario para el registro de los estudiantes.

Cambios

- `estudiantes_matriculados(curso)`

  Debemos obtener los *nombres y apellidos*.

- Registro `InfoCurso`. teníamos un set<Estudiante> contenía nombres y apellidos de estudiantes matriculados en Infocurso

  Ahora se almacenan los NIFs de los/as estudiantes matriculados/as en `InfoCurso`.

  El conjunto de estudiantes matriculados puede ser `unordered_set`.

- `matricular_en_curso(id_est, curso)`

  Ahora es necesario comprobar si el estudiante está registrado.

  Pasa a tener coste $O(1)$. → Hemos cambiado el set por el unordered_set-

Obtener 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 &id_est: info_curso.estudiantes) {
      const InfoEstudiante &info_est = estudiantes.at(id_est);
      result.push_back(info_est.apellidos + ", " + info_est.nombre);
    }

    std::sort(result.begin(), result.end());
    return result;
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Coste global-

O(1)

O(1) O(NL)

O(NC log NC)

Ordenar la lista resultado y devolver.

Inicialmente vacía. Aquí vamos a ir añadiendo el listado de los estudiantes del curso y metemos el estudiante en un vector por nombre y apellido.

Obtener cursos de un estudiante

```
class Academia {
public:
  std::vector<std::string> cursos_estudiante(const Estudiante &id_est) const {
    std::vector<std::string> result;

    for (auto entrada: cursos) {
      const InfoCurso &info_curso = entrada.second;
      if (info_curso.estudiantes.contains(id_est)) {
        result.push_back(info_curso.nombre);
      }
    }
    sort(result.begin(), result.end());
    return result;
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

Devuelve un vector con los nombres de los cursos en los que está matriculado.

Todos los cursos

Cojo el conjunto de estudiantes.

si el estudiante esta en el conjunto de estudiantes....

pushearlo al vector

ordenar los estudiantes por nombre.

O(1)

O(1)

O(M)

O(nc key Mc)

Cursos matriculados por cada estudiante

```
class Academia {
public:
  …

private:
  struct InfoCurso { … };

  struct InfoEstudiante {
    Estudiante id_est;
    std::string nombre;
    std::string apellidos;

    std::set<std::string> cursos;

    InfoEstudiante(const Estudiante &id_est,
                   const std::string &nombre, const std::string &apellidos);
  };

  …
}
```

Para mantener los cursos en orden alfabético.

Obtener estudiantes matriculados (cambios)

```
class Academia {
public:
  std::vector<std::string> cursos_estudiante(const Estudiante &id_est) const {
    const InfoEstudiante &info_est = buscar_estudiante(id_est);
    std::vector<std::string> result;
    copy(info_est.cursos.begin(), info_est.cursos.end(),
         std::back_insert_iterator<std::vector<std::string>>(result));

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

O(1)

O(Mg)

Más eficiente que lo anterior.

Matrícula en un curso (cambios)

```
class Academia {
public:
  void matricular_en_curso(const Estudiante &est, const Curso &curso) {
    InfoCurso &info_curso = buscar_curso(curso);
    InfoEstudiante &info_est = buscar_estudiante(est);
    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);
    info_est.cursos.insert(curso);
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

O(log M C)

Eliminar un curso (cambios)

```
class Academia {
public:
  void eliminar_curso(const Curso &curso) {
    auto it = cursos.find(curso);
    if (it ≠ cursos.end()) {
      InfoCurso &info_curso = it→second;
      for (Estudiante id_est : info_curso.estudiantes) {
        estudiantes.at(id_est).cursos.erase(curso);
      }
      cursos.erase(it);
    }
  }
  …
private:
  …
  std::unordered_map<Curso, InfoCurso> cursos;
}
```

$O(1)$

$O(NC \cdot \log MC)$

Es para cada curso.

Borro el curso que eliminamos de la academia de la lista de cursos del estudiante-

$O(\log(M \cdot C))$