

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



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.

Métricas de coste

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



Interfaz

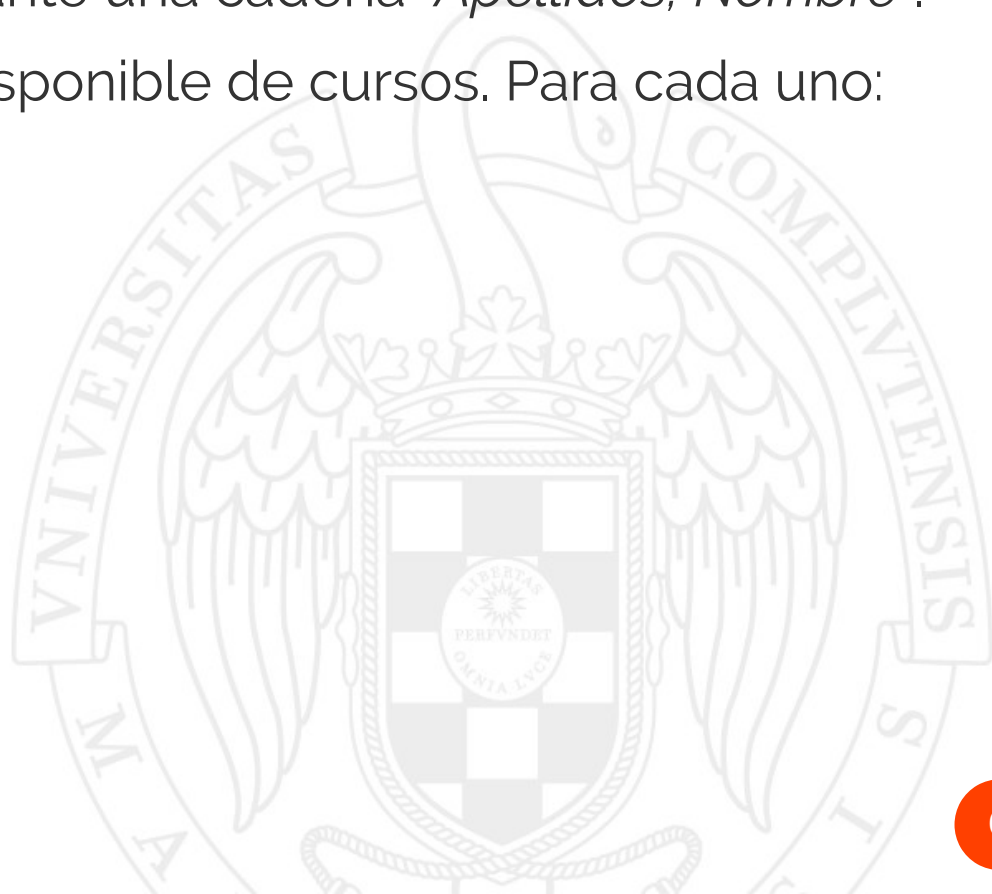
```
using Estudiante = std::string;
using Curso = std::string;

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:
    ...
}
```

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.



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.
 - Multiconjunto.
 - Multidiccionario.

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
 - unordered_map

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;
}
```



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;
}
```

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

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;  
}
```



Eliminar un curso

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



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;
}
```

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;
}
```


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_inserter<std::vector<std::string>>(result));  
        return result;  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```

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

Registro de estudiantes

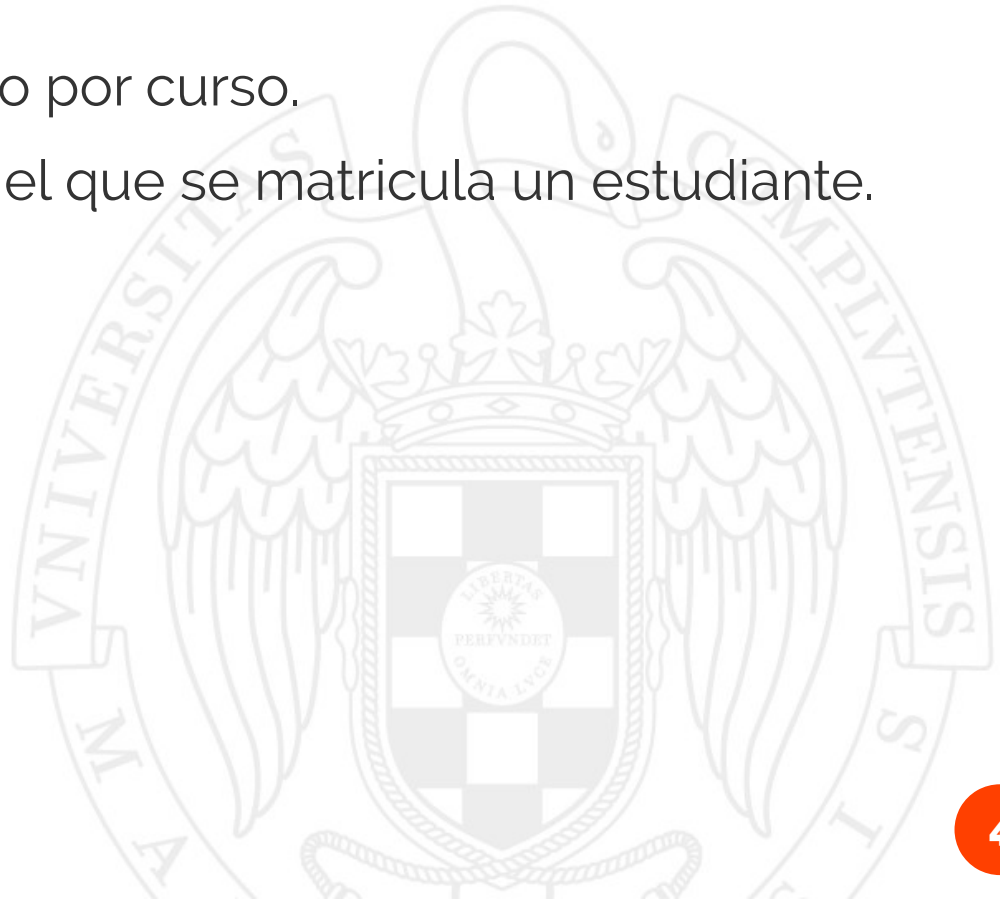


Requisitos

- 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.
- NC = número de estudiantes máximo por curso.
- MC = número de cursos máximo en el que se matricula un estudiante.



Registro de estudiantes

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

Ahora representa el NIF

```
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;  
}
```

Cambios

- `estudiantes_matriculados(curso)`

Debemos obtener los *nombres y apellidos*.

- Registro 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 $O(1)$.

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;  
}
```

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;
}
```

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);  
    };  
    ...  
}
```



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_inserter<std::vector<std::string>>(result));  
  
        return result;  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



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;
}
```

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;  
}
```



ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

Gestión de una academia (3)

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

Lista de espera



Requisitos

- Cada curso tiene una lista de espera.
- Si un estudiante se matricula en un curso y no hay plazas disponibles, se le pone en lista de espera.
- Cuando un estudiante se da de baja en un curso, se matricula automáticamente al primero de la lista de espera (si existe).
- Operaciones añadidas o modificadas:
 - Dar de baja a un estudiante.
 - La operación de matrícula de un curso devuelve un booleano indicando si el estudiante ha sido matriculado o está en lista de espera.

Lista de espera en cursos

```
class Academia {  
public:  
    ...  
  
private:  
    struct InfoCurso {  
        std::string nombre;  
        int numero_plazas;  
        std::unordered_set<Estudiante> estudiantes;  
        std::queue<Estudiante> lista_espera;  
  
        InfoCurso(const std::string &nombre,  
                   int numero_plazas);  
    };  
  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



Matrícula en un curso (cambios)

```
class Academia {
public:
    bool 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) {
            info_curso.estudiantes.insert(est);
            info_est.cursos.insert(curso);
            return true;
        } else {
            info_curso.lista_espera.push(est);
            return false;
        }
    }
...
private:
    ...
    std::unordered_map<Curso, InfoCurso> cursos;
}
```

Darse de baja en un curso

```
class Academia {
public:
    void dar_de_baja_en_curso(const Estudiante &id_est, const Curso &nombre_curso) {
        InfoCurso &curso = buscar_curso(nombre_curso);

        auto it_estudiante = curso.estudiantes.find(id_est);
        if (it_estudiante != curso.estudiantes.end()) {
            curso.estudiantes.erase(it_estudiante);
            it_estudiante->cursos.erase(curso.nombre);

            while (!curso.lista_espera.empty() && curso.estudiantes.size() < curso.numero_plazas) {
                const Estudiante &nif_primerero = curso.lista_espera.front();
                curso.lista_espera.pop();
                if (!curso.estudiantes.contains(nif_primerero)) {
                    curso.estudiantes.insert(nif_primerero);
                    estudiantes.at(nif_primerero).cursos.insert(curso.nombre);
                }
            }
        }
    }
    ...
private:
    ...
    std::unordered_map<Curso, InfoCurso> cursos;
}
```

ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

Líneas de metro

Manuel Montenegro Montes

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

Requisitos

- Queremos implementar un sistema de gestión de líneas de metro.
- Contendrá información sobre líneas del suburbano, paradas disponibles en cada línea, y horarios de salida de trenes en cada línea.

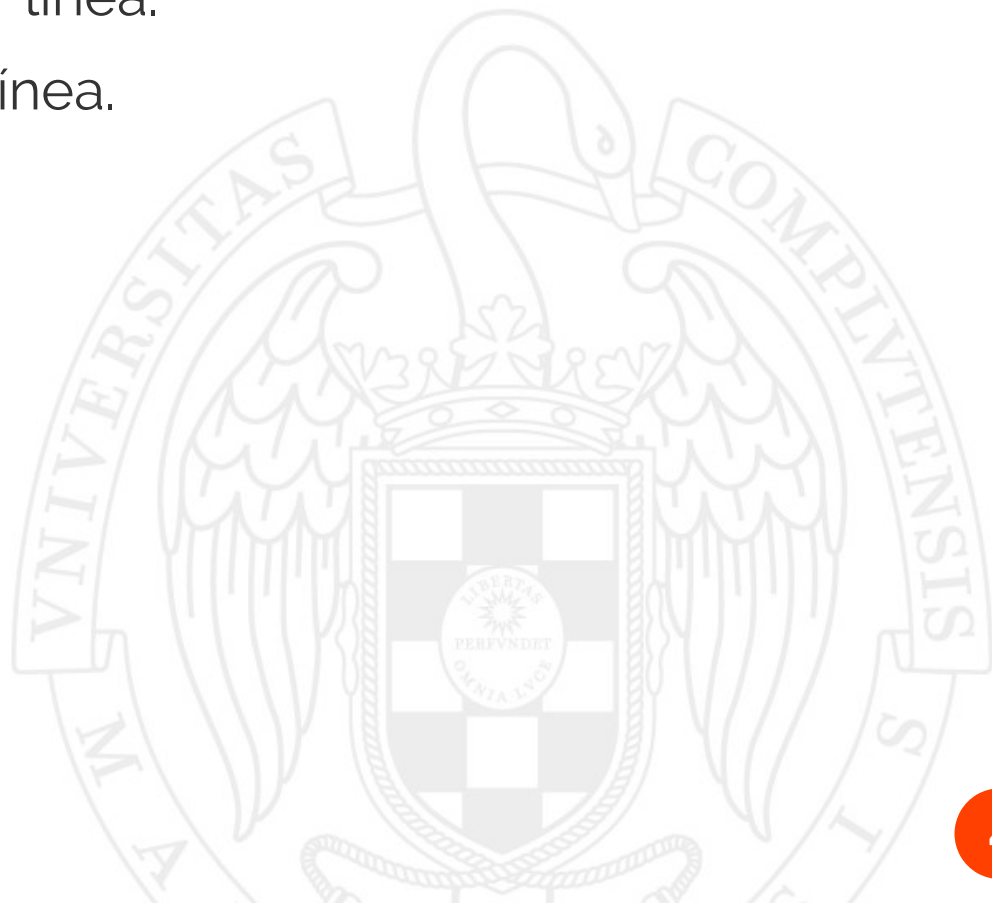


Operaciones

- Crear un sistema de líneas de metro vacío.
- Añadir una nueva línea de metro.
- Añadir una parada a una nueva línea.
 - Se indicará el tiempo de recorrido (en segundos) desde la parada anterior (cero si es la primera parada).
- Añadir una nueva hora de salida en una línea (hora de salida desde cabecera).
- Obtener el número de trenes que salen diariamente en una línea.
- Obtener el tiempo de espera hasta el próximo tren en una parada determinada.

Métricas de coste

- L = Número de líneas.
- P = Número de paradas máximo por línea.
- T = Número máximo de trenes por línea.



TAD de gestión de horarios



Interfaz del TAD Hora

```
class Hora {  
public:  
    Hora(int horas, int minutos, int segundos);  
    int horas() const;  
    int minutos() const;  
    int segundos() const;  
  
    Hora operator+(int segs) const;  
    Hora operator-(int segs) const;  
    int operator-(const Hora& otra) const;  
  
    bool operator==(const Hora &otra) const;  
    bool operator<(const Hora &otra) const;  
  
private:  
    ...  
};
```



Representación del TAD Hora

```
class Hora {  
public:  
...
```

```
private:  
    int num_segundos;  
  
    Hora(int num_segundos);  
};
```

**Segundos transcurridos desde
la hora 00:00:00**



Representación del TAD Hora

```
class Hora {  
public:  
    ...
```

```
private:  
    int num_segundos;
```

```
    Hora(int num_segundos);  
};
```



Constructor privado

Implementación del TAD Hora

```
class Hora {  
public:  
  
    Hora(int horas, int minutos, int segundos): num_segundos(horas * 3600 + minutos * 60 + segundos) {  
  
        if (horas < 0 || minutos < 0 || minutos ≥ 60 || segundos < 0 || segundos ≥ 60) {  
            throw std::domain_error("hora no válida");  
        }  
  
    }  
  
    int horas() const { return num_segundos / 3600; }  
    int minutos() const { return (num_segundos / 60) % 60; }  
    int segundos() const { return num_segundos % 60; }  
  
private:  
    ...  
};
```

Implementación del TAD Hora

```
class Hora {  
public:  
  
    ...  
  
    Hora operator+(int segs) const {  
        return Hora(num_segundos + segs);  
    }  
  
    int operator-(const Hora& otra) const {  
        return num_segundos - otra.num_segundos;  
    }  
  
    Hora operator-(int segs) const {  
        return Hora(num_segundos - segs);  
    }  
  
private:  
    ...  
};
```



Implementación del TAD Hora

```
class Hora {  
public:  
  
    ...  
  
    bool operator==(const Hora &otra) const {  
        return num_segundos == otra.num_segundos;  
    }  
  
    bool operator<(const Hora &otra) const {  
        return num_segundos < otra.num_segundos;  
    }  
  
private:  
    ...  
};
```



TAD de gestión de líneas de metro



Interfaz

```
class Metro {  
public:  
    Metro();  
  
    void nueva_linea(const Linea &nombre);  
  
    void nueva_parada(const Linea &nombre, const Parada &nueva_parada, int tiempo_desde_anterior);  
  
    void nuevo_tren(const Linea &nombre, const Hora &hora_salida);  
  
    int numero_trenes(const Linea &nombre) const;  
  
    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual);  
  
private:  
    ...  
};
```

Colección de líneas

- Guardamos, para cada línea, la siguiente información:
 - Nombre (o número) de línea, que la identifica.
 - Paradas de esa línea.
 - Horarios de salida de esa línea.
- Cada operación del TAD Metro necesita acceder a la información de una línea. Necesitamos acceso rápido a esa información.
- Solución: diccionario que asocia nombres de líneas con información de cada línea.

Representación

```
using Linea = std::string;

class Metro {
public:
    ...

private:
    struct InfoLinea { ... };
    std::unordered_map<Linea, InfoLinea> lineas;
};
```



Colección de líneas

- InfoLinea debe contener:
 - Nombre (o número) de línea.
 - Paradas de esa línea.
 - Horarios de salida de esa línea.
- ¿Cómo almacenamos la colección de paradas?
 - Existe un orden entre las paradas; viene dado por el orden en el que las inserte.
 - Tenemos que recorrer las paradas hasta una determinada posición.

Colección de líneas

- InfoLinea debe contener:
 - Nombre (o número) de línea.
 - Paradas de esa línea.
 - Horarios de salida de esa línea.
- ¿Cómo almacenamos la colección de horarios de salida?
 - Existe un orden entre los horarios, pero no viene dado por el orden en el que se inserten.
 - Necesitamos acceso eficiente al tren que sale después de una determinada hora.

Representación

```
using Linea = std::string;

class Metro {
public:
    ...

private:

    struct InfoLinea {
        Linea nombre;
        std::set<Hora> salida_trenes;
        std::list<InfoParada> paradas;

        InfoLinea(const Linea &nombre): nombre(nombre) {}
    };

    std::unordered_map<Linea, InfoLinea> lineas;
};
```



Representación

```
using Parada = std::string;

class Metro {
public:
    ...

private:

    struct InfoParada {
        Parada nombre;
        int tiempo_desde_anterior;

        InfoParada(const Parada &nombre, int tiempo_desde_anterior);
    };

    struct InfoLinea { ... };
    std::unordered_map<Linea, InfoLinea> lineas;
};
```



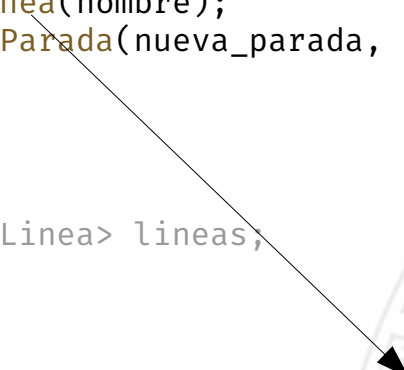
Añadir una nueva línea

```
class Metro {  
public:  
  
    void nueva_linea(const Linea &nombre) {  
        if (lineas.contains(nombre)) {  
            throw std::domain_error("línea ya existente");  
        }  
        lineas.insert({nombre, InfoLinea(nombre)});  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



Añadir una nueva parada

```
class Metro {  
public:  
  
    void nueva_parada(const Linea &nombre, const Parada &nueva_parada, int tiempo_desde_anterior) {  
        InfoLinea &linea = buscar_linea(nombre);  
        linea.paradas.push_back(InfoParada(nueva_parada, tiempo_desde_anterior));  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



```
InfoLinea & buscar_linea(const Linea &linea) {  
    auto it = lineas.find(linea);  
    if (it == lineas.end()) {  
        throw std::domain_error("línea no encontrada");  
    }  
    return it->second;  
}
```

Añadir un nuevo horario de salida

```
class Metro {
public:

    void nuevo_tren(const Linea &nombre, const Hora &hora_salida) {
        InfoLinea &linea = buscar_linea(nombre);
        linea.salida_trenes.insert(hora_salida);
    }

    int numero_trenes(const Linea &nombre) const {
        const InfoLinea &linea = buscar_linea(nombre);
        return linea.salida_trenes.size();
    }

private:
    ...
    std::unordered_map<Linea, InfoLinea> lineas;
};
```



Tiempo hasta el próximo tren

- Necesitamos un método auxiliar que calcule el tiempo de trayecto desde la cabecera de línea hasta una parada dada.

```
int buscar_parada(const InfoLinea &info_linea, const Parada &parada) {  
    int segs_desde_cabecera = 0;  
  
    auto it = info_linea.paradas.begin();  
    while (it != info_linea.paradas.end() && it->nombre != parada) {  
        segs_desde_cabecera += it->tiempo_desde_anterior;  
        ++it;  
    }  
  
    if (it == info_linea.paradas.end()) {  
        throw std::domain_error("parada no encontrada");  
    }  
  
    segs_desde_cabecera += it->tiempo_desde_anterior;  
  
    return segs_desde_cabecera;  
}
```

Tiempo hasta el próximo tren

```
class Metro {
public:

    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual) {
        const InfoLinea &info_linea = buscar_linea(linea);
        int segs_desde_cabecera = buscar_parada(info_linea, parada);
        Hora hora_salida = hora_actual - segs_desde_cabecera;

        auto it = info_linea.salida_trenes.lower_bound(hora_salida);
        if (it == info_linea.salida_trenes.end()) {
            return -1;
        }

        const Hora &hora_salida_siguiente = *it;
        const Hora &hora_parada_siguiente = hora_salida_siguiente + segs_desde_cabecera;
        return hora_parada_siguiente - hora_actual;
    }

private:
    ...
    std::unordered_map<Linea, InfoLinea> lineas;
};
```

Representación alternativa



Representación alternativa

- En lugar de almacenar el tiempo de recorrido desde la parada anterior, podemos almacenar el tiempo desde la cabecera de línea.
- Podemos cambiar la lista de paradas por un diccionario que asocia cada parada con el tiempo de recorrido desde la cabecera.
- Necesitamos almacenar, para cada línea, el tiempo total de recorrido desde la cabecera hasta la última parada.

```
struct InfoLinea {  
    Linea nombre;  
    std::set<Hora> salida_trenes;  
    std::list<InfoParada> paradas;  
    ...  
};
```

```
struct InfoLinea {  
    Linea nombre;  
    std::set<Hora> salida_trenes;  
    std::list<InfoParada> paradas;  
    int tiempo_total;  
    std::unordered_map<Parada, int> tiempos_desde_cabecera;  
    ...  
};
```


Añadir una nueva parada (modificado)

```
class Metro {  
public:  
  
    void nueva_parada(const Linea &nombre, const Parada &nueva_parada, int tiempo_desde_anterior) {  
        InfoLinea &linea = buscar_linea(nombre);  
        linea.tiempo_total += tiempo_desde_anterior;  
        linea.tiempos_desde_cabecera.insert({nueva_parada, linea.tiempo_total});  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



Tiempo hasta el próximo tren

```
class Metro {
public:

    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual) {
        const InfoLinea &info_linea = buscar_linea(linea);
        int segs_desde_cabecera = buscar_parada(info_linea, parada);
        Hora hora_salida = hora_actual - segs_desde_cabecera;

        auto it = info_linea.salida_trenes.lower_bound(hora_salida);
        if (it == info_linea.salida_trenes.end()) {
            return -1;
        }

        const Hora &hora_salida_siguiente = *it;
        const Hora &hora_parada_siguiente = hora_salida_siguiente + segs_desde_cabecera;
        return hora_parada_siguiente - hora_actual;
    }

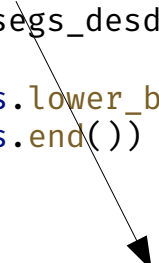
private:
    ...
    std::unordered_map<Linea, InfoLinea> lineas;
};
```

Tiempo hasta el próximo tren

```
class Metro {  
public:
```

```
    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual) {  
        const InfoLinea &info_linea = buscar_linea(linea);  
        int segs_desde_cabecera = buscar_parada(info_linea, parada);  
        Hora hora_salida = hora_actual - segs_desde_cabecera;  
  
        auto it = info_linea.salida_trenes.lower_bound(hora_salida);  
        if (it == info_linea.salida_trenes.end()) {  
            return -1;  
        }  
    }
```

```
    const Hora &hora_salida_s  
    const Hora &hora_parada_s  
    return hora_parada_siguie  
}  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



```
int buscar_parada(const InfoLinea &info_linea, const Parada &parada) {  
    auto it = info_linea.tiempos_desde_cabecera.find(parada);  
    if (it == info_linea.tiempos_desde_cabecera.end()) {  
        throw std::domain_error("parada no encontrada");  
    }  
    return it->second;  
}
```