

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

Ejemplo relacionado con el manejo de los horarios en un sistema de transporte, en este caso el metro

- 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.

NECESITAREMOS UN TAD DE HORAS QUE YA HEMOS VISTO EN EL PROBLEMA DE LA HORA DE LA PASTILLA



# Operaciones

- Crear un sistema de líneas de metro vacío. → Constructor
- Añadir una nueva línea de metro. operación añadir.
- Añadir una parada a una nueva línea. operación añadir parada a una lista o conjunto
  - Se indicará el tiempo de recorrido (en segundos) desde la parada anterior (cero si es la primera parada). Al añadirlo se indicará el tiempo entre paradas.
- 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. Línea 1 línea 2 línea 3...
- $P$  = 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); constructor  
    int horas() const; métodos para acceder a cada una de las 3 componentes.  
    int minutos() const;  
    int segundos() const;  
  
    Hora operator+(int secs) const;  
    Hora operator-(int secs) const;  
    int operator-(const Hora& otra) const; Devolverá la diferencia en segundos entre 2 horas.  
  
    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**

En vez de tener horas, minutos y segundos.

# Representación del TAD Hora

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

```
private:  
    int num_segundos;
```

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

**Constructor privado**

Inicializa el atributo con el número de segundos pasado por parámetros.



# 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:  
    ...  
};
```

lanza excepciones si los datos de entrada del constructor NO son correctos.

# Implementación del TAD Hora

```
class Hora {  
public:
```

```
...
```

```
Hora operator+(int segs) const {  
    return Hora(num_segundos + segs);  
}
```

Suma al atributo de la clase num\_segundos los segundos que se pasan por parámetro.

```
int operator-(const Hora& otra) const {  
    return num_segundos - otra.num_segundos;  
}
```

Resta el número de segundos que hay entre 2 horas

```
Hora operator-(int segs) const {  
    return Hora(num_segundos - segs);  
}
```

Resta a this.num\_segundos una cantidad de segundos que se reciben como parámetro.

```
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:  
    ...  
};
```

COMPARA HORAS.

# 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:  
    ...  
};
```

Es lo mismo que un nuevo horario de salida.

Número de salidas.

A una determinada parada

Pasarle la hora actual

# Colección de líneas

- Guardamos, para cada línea, la siguiente información:
  - Nombre (o número) de línea, que la identifica. Podría ser un String por ejemplo
  - 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.

$(1,1)$   $(1,1)$   
R

# Representación

```
using Linea = std::string;
```

```
class Metro {  
public:
```

```
...
```

```
private:
```

```
    struct InfoLinea { ... };
```

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

```
};
```

Diccionario que asocia líneas, en este caso strings, con información relativa a esas líneas.

# 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.

O vector o list.

Vector para accesos instantaneos en la lista  
List si queremos añadir o eliminar paradas facilmente.

Como no tenemos que hacer ninguna de las dos cosas utilizaremos list.



# 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. Puedes añadir un tren para las 10 y luego otro para las 9:45 por ejemplo
  - Necesitamos acceso eficiente al tren que sale después de una determinada hora.

Luego necesitaremos un conjunto y como necesitamos orden, usaremos set, es decir, árboles binarios de búsqueda.

# 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;
};
```



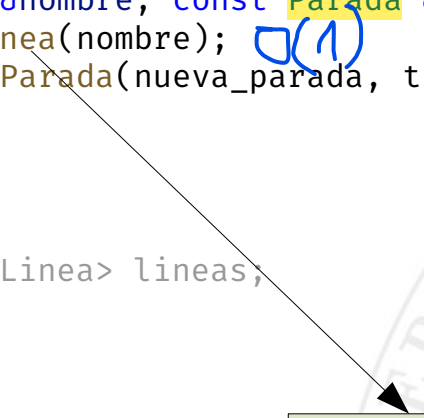
# Añadir una nueva línea

```
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;  
};
```

Ya que es una tabla Hash.

# 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;  
};
```



A diagram showing a call to the `buscar_linea` function from the `nueva_parada` function. A line starts from the `buscar_linea(nombre)` call in the `nueva_parada` function and points to the `buscar_linea` function definition in the box below.

```
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;  
}
```

coste constante.

# 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;  
};
```

Al conjunto de salidas de trenes le añado la hora que le pasamos por parámetro.

$\hookrightarrow \log(T)$

$O(1)$

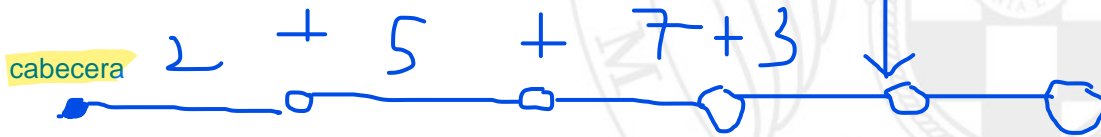
primero buscamos a ver si existe la línea correspondiente.

Obtengo el tamaño que tiene coste constante.

# 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);  $O(1)$ 
        int segs_desde_cabecera = buscar_parada(info_linea, parada);  $O(P)$ 
        Hora hora_salida = hora_actual - segs_desde_cabecera;

        auto it = info_linea.salida_trenes.lower_bound(hora_salida);  $O(\log |T|)$ 
        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;
};
```

Devuelve el primer elemento que es mayor/igual al pasado como parámetro.

$O(1)$



# Representación alternativa

Para poder llegar a mejorar los costes de la función anterior.



# 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;  
}
```

Lo único que cambia respecto a la implementación anterior es este método.

$O(\log T)$



$O(1)$