

COLAS DE PRIORIDAD

- En estas colas los elementos NO salen por orden de llegada, salen atendiendo a su prioridad. EL primero que sale es el más prioritario.



UNIVERSIDAD
COMPLUTENSE
M A D R I D

ALBERTO VERDEJO

Colas de prioridad

First In- First Out de ED

- En las colas “ordinarias” se atiende por riguroso orden de llegada (FIFO).
- También hay colas en las cuales se atiende según la prioridad de los elementos y no según el orden de llegada: son las **colas de prioridad**.
- Cada elemento tiene una prioridad que determina quién va a ser el primero en ser atendido; hace falta tener un **orden total** sobre ellos.
- El primero en ser atendido puede ser el elemento con menor valor o el elemento con mayor valor según se trate de **colas de prioridad de mínimos o de máximos**, respectivamente.

$$a < b \quad | \quad b > a$$

En la cola de mínimos, el elemento menor será el más prioritario y en la de máximos el elemento mayor será el más prioritario..

Ejemplo: cola de prioridad de máximos

- Empezamos con una cola vacía

Cola:

Ejemplo: cola de prioridad de máximos

- Llega el 15

Cola:

Ejemplo: cola de prioridad de máximos

Cola: 15



más prioritario

Ejemplo: cola de prioridad de máximos

- Llega el 7

Cola: 15



más prioritario

Ejemplo: cola de prioridad de máximos

Cola: 15 7



más prioritario

Ejemplo: cola de prioridad de máximos

- Llega el 10

Cola: 15 7



más prioritario

Como el 7 es menor que el 15 es MENOS PRIORITARIO EL 7 y se coloca detrás

Ejemplo: cola de prioridad de máximos

Cola: 15 10 7



más prioritario

El 10 es menor que el 15 pero más grande que el 7 por tanto se coloca entre medias. Aquí lo ponemos como una secuencia ordenada pero NO QUIERE DECIR que esto se represente así. No lo vamos a tener así.

Ejemplo: cola de prioridad de máximos

- Sale el más prioritario

Cola: 15 10 7



más prioritario

Ejemplo: cola de prioridad de máximos

Cola: 10 7



más prioritario

Ejemplo: cola de prioridad de máximos

- Llega el 30

Cola: 10 7



más prioritario

RECORDAR: En las colas FIFO el primero que entra es el primero que sale. Aquí el primero que entra NO ES el primero que sale. el primero que sale es el más prioritario.

Ejemplo: cola de prioridad de máximos

Cola: 30 10 7



más prioritario

Ejemplo: cola de prioridad de máximos

- Llega el 23

Cola: 30 10 7



más prioritario

Ejemplo: cola de prioridad de máximos

Cola: 30 23 10 7



más prioritario

Ejemplo: cola de prioridad de máximos

- Sale el más prioritario

Cola: 30 23 10 7



más prioritario

Ejemplo: cola de prioridad de máximos

Cola: 23 10 7

↑
más prioritario

Al salir, el elemento más prioritario va cambiando.

TAD de las colas de prioridad

El TAD de las colas de prioridad cuenta con las siguientes operaciones:

- ▶ crear una cola de prioridad vacía
- ▶ insertar un elemento, `void push(T const& e)`
- ▶ consultar el elemento más prioritario, `T const& top() const`
- ▶ eliminar el primer elemento, `void pop()`
- ▶ determinar si la cola de prioridad es vacía, `bool empty() const`
- ▶ consultar el número de elementos de la cola, `int size() const`

TAD Genérico que sirve para cualquier tipo de dato.

Colas de prioridad en la STL de C++

- ▶ La librería **queue** de la STL contiene la clase **priority_queue** que implementa colas de prioridad de máximos.
dado un orden como el menor.
- ▶ Dado un orden, como $<$, la operación **top()** devuelve el elemento mayor, el que se encuentra más a la derecha en el orden

$$a_1 < a_2 < \dots < a_n$$

El mayor es el que está a la derecha del todo.

Colas de prioridad en la STL de C++

- Podemos utilizar esas colas como **colas de mínimos** si cambiamos el objeto comparador, utilizando el operador **>**:

$$b_1 > b_2 > \dots > b_n$$

El menor también es el que está a la derecha del todo.

- El comparador es el tercer argumento de la plantilla:

```
priority_queue<int, vector<int>, greater<int>> cola_min;
```

tipo de elemento que vamos a almacenar.

tipo de soporte que vamos a utilizar para almacenar los elementos

Comparador, para utilizar el OPERADOR MAYOR y que el elemento MENOR quede a la derecha.

Unidad Curiosa de Monitorización

EJEMPLO DE PROBLEMA EN EL QUE INTERESA UTILIZAR LAS COLAS DE PRIORIDAD.

La Unidad Curiosa de Monitorización (UCM) se encarga de leer los datos proporcionados por una serie de sensores y enviar con cierta periodicidad los datos obtenidos y procesados a los usuarios que se han registrado previamente.



La UCM admite que los usuarios se registren proporcionando un Identificador único y un Periodo, el intervalo de tiempo que transcurrirá entre dos envíos consecutivos de información a ese usuario.

Acaban de registrarse varios usuarios. ¿Podrías decir a quiénes irán dirigidos los K primeros envíos de información? Si dos o más usuarios tienen que recibir la información al mismo tiempo, los envíos se realizan en orden creciente de sus identificadores de usuario.

Tendremos una cola de prioridad con los usuarios registrados con prioridad el momento en el que tienen que recibir el envío.

El que tenga el menor momento recibirá el siguiente envío.

Unidad Curiosa de Monitorización

```
struct registro {  
    int momento; // cuándo le toca  
    int id;      // identificador (se utiliza en caso de empate)  
    int periodo; // tiempo entre consultas  
};
```

```
bool operator<(registro const& a, registro const& b) {  
    return b.momento < a.momento ||  
        (a.momento == b.momento && b.id < a.id);  
}
```

Instante de tiempo en el que le toca ser atendido.

Trata de comparar cuando b es el MÁS PRIORITARIO.

A igualdad de momentos comparamos los ids. Necesario añadir este operador para poder comparar los registros.

Unidad Curiosa de Monitorización

```
bool resuelveCaso() {  
    int N; // número de usuarios registrados  
    cin >> N;  
  
    if (N == 0) return false; // no hay más casos
```

```
priority_queue<registro> cola;
```

cola vacía de registros. NO HACE FALTA CAMBIAR EL COMPARADOR EN LA INSTANCIA. NO SE QUÉ ESTRUCTURA DE DATOS ES LA QUE SE UTILIZA PARA ALMACENAR ESOS REGISTROS.

```
// leemos los registros  
for (int i = 0; i < N; ++i) { coste O(Nlog(N))  
    int id_usu, periodo;  
    cin >> id_usu >> periodo;  
    cola.push({periodo, id_usu, periodo});  
}
```

$O(\log(N))$

El momento coincide con el periodo porque la primera vez que el usuario será notificado cuando haya transcurrido periodo unidades de tiempo

Unidad Curiosa de Monitorización

```
int envios; // número de envíos a mostrar  
cin >> envios;  
  
O(envios log(N))  
while (envios--) {  
    auto e = cola.top(); cola.pop();  
    cout << e.id << '\n';  
    e.momento += e.periodo;  
    cola.push(e);  
}  
cout << "---\n";  
return true;  
}
```

O(1) O(log(N))

Consultamos y eliminamos de la cola de prioridad el más prioritario, el del momento menor.

Le reiniciamos el periodo y lo volvemos a meter en la cola con el momento siendo igual que su periodo.

Para saber el coste de este algoritmo debemos saber el coste de cada una de las operaciones de las colas de prioridad-

COSTE TOTAL DEL ALGORITMO = $O(N \log N + \text{envios} \log N)$

Unidad Curiosa de Monitorización

¿Y si el < ordena de menor a mayor?

Si ordenara de menor a mayor, el mayor sería el de la derecha del todo

```
struct registro {  
    int momento; // cuándo le toca  
    int id;      // identificador (se utiliza en caso de empate)  
    int periodo; // tiempo entre consultas  
};
```

```
bool operator<(registro const& a, registro const& b) {  
    return a.momento < b.momento || a es menor cuando tiene un momento menor  
        (a.momento == b.momento && a.id < b.id);  
}
```

Estaríamos considerando como más prioritario el que tiene un momento mayor.

```
bool operator>(registro const& a, registro const& b) {  
    return b < a;  
}
```

Al ser una cola de prioridad de máximos necesitamos añadir esto. haciendo esto, consideraría como más prioritario el que tiene un momento MENOR QUE ES LO QUE QUEREMOS.

Unidad Curiosa de Monitorización

```
bool resuelveCaso() {  
    int N; // número de usuarios registrados  
    cin >> N;  
  
    if (N == 0) return false; // no hay más casos  
  
    priority_queue<registro, vector<registro>, greater<registro>> cola;  
    // leemos los registros  
    for (int i = 0; i < N; ++i) {  
        int id_usu, periodo;  
        cin >> id_usu >> periodo;  
        cola.push({periodo, id_usu, periodo});  
    }  
}
```

UTILIZAMOS EL OPERADOR MAYOR.

Unidad Curiosa de Monitorización

¿Y si el < ordena de una forma diferente?

```
struct registro {  
    int momento; // cuándo le toca  
    int id;      // identificador (se utiliza en caso de empate)  
    int periodo; // tiempo entre consultas  
};
```

STRUCT = CLASE DONDE TODOS LOS MÉTODOS SON PÚBLICOS

```
struct comp_registro {  
    bool operator()(registro const& a, registro const& b) {  
        return b.momento < a.momento ||  
               (a.momento == b.momento && b.id < a.id);  
    }  
};
```

Nos dice cuándo el elemento segundo es más prioritario.

b será más prioritario cuando sea menor que a.

Unidad Curiosa de Monitorización

```
bool resuelveCaso() {  
    int N; // número de usuarios registrados  
    cin >> N;  
  
    if (N == 0) return false; // no hay más casos  
  
    priority_queue<registro, vector<registro>, comp_registro> cola;  
  
    // leemos los registros  
    for (int i = 0; i < N; ++i) {  
        int id_usu, periodo;  
        cin >> id_usu >> periodo;  
        cola.push({periodo, id_usu, periodo});  
    }  
}
```