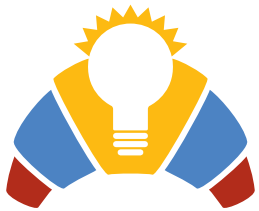
Estructuras de Datos

Inicial

Mariano Feresin

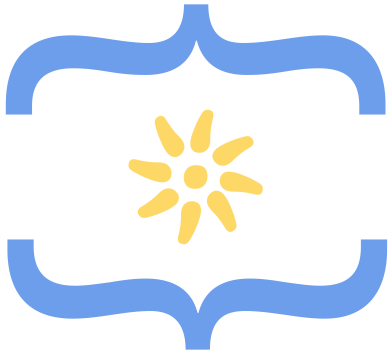
Universidad Nacional de La Plata – Facultad de Informática

Training Camp 2026



Gracias Sponsors!

Organiza



Facultad de INFORMÁTICA

UNIVERSIDAD NACIONAL DE LA PLATA

Diamond Sponsor

Outline

Outline

Para empezar, algunas preguntas

- ¿Para qué usamos estructuras de datos?
- ¿Por qué no metemos todo en un array?
- ¿Qué ventajas ganamos? ¿Velocidad de acceso? ¿De búsqueda?
- ¿Qué desventajas? ¿Más memoria?

¿Qué es una estructura de datos?

Con los datos de un problema siempre hacemos lo mismo: guardarlos, consultarlos y modificarlos.

Idea central

Una estructura de datos es una forma de **organizar** los datos para que ciertas operaciones sean **rápidas**.

¿Por qué no metemos todo en un array?

¡Podemos! El array soporta todas las operaciones... pero cada una tiene un costo.

Operación (array de N elementos)	**Costo**
Acceder por posición	$\mathcal{O}(1)$
Agregar al final	$\mathcal{O}(1)$
Buscar un elemento	$\mathcal{O}(N)$
Insertar / borrar al principio	$\mathcal{O}(N)$
Encontrar el mínimo	$\mathcal{O}(N)$

El array no es "malo": es rapidísimo en algunas operaciones y lento en otras.

Ventajas y desventajas: no hay magia

Toda estructura hace un intercambio (trade-off): gana velocidad en algunas operaciones a cambio de...

- usar **más memoria** (punteros, nodos, espacio extra),
- hacer **otras operaciones más lentas**, o directamente no soportarlas,
- a veces perder cosas, como el orden de inserción.

No existe la estructura "mejor en todo": si existiera, usaríamos siempre esa.

La pregunta clave

Cada vez que veamos una estructura, preguntarse:

¿Qué operaciones son **eficientes** en esta estructura de datos?

Receta para elegir estructura:

1. Listar las operaciones que necesita mi solución.
2. Contar cuántas veces se ejecuta cada una.
3. Elegir la estructura donde las operaciones frecuentes sean baratas.

En el resto de la clase vamos a recorrer las estructuras más usadas y, para cada una, responder siempre esta misma pregunta.

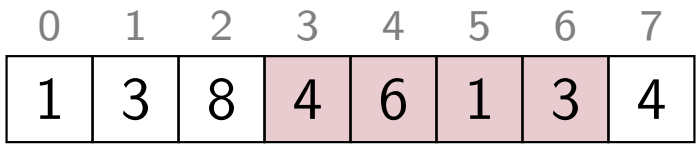
Outline

¿Qué es una range query?

En una **range query** nos piden calcular un valor sobre un **rango** $[a, b]$ de un array:

- $\texttt{sum}_q(a, b)$: la suma de los valores del rango $[a, b]$.
- $\texttt{min}_q(a, b)$: el mínimo del rango $[a, b]$.
- $\texttt{max}_q(a, b)$: el máximo del rango $[a, b]$.

Por ejemplo, para el rango $[3, 6]$:



$$\texttt{sum}_q(3, 6) = 14, \quad \texttt{min}_q(3, 6) = 1, \quad \texttt{max}_q(3, 6) = 6.$$

La solución directa

Recorrer el rango

```
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += array[i];
    }
    return s;
}
```

Cada consulta cuesta $\mathcal{O}(n)$. Con q consultas: $\mathcal{O}(n \cdot q)$.

¿Y si $n, q \leq 10^5$**?** $\quad 10^5 \cdot 10^5 = 10^{10} \Rightarrow$ no entra en tiempo.

¿Podemos responder una consulta sin recorrer todo el rango?

Arrays estáticos: prefix sums

Si el array **no cambia** entre consultas, podemos **precalcular**.

Prefix sum array: en la posición k guardamos $\texttt{sum}_q(0, k)$, la suma de todo el array hasta k:

	0	1	2	3	4	5	6	7
array	1	3	4	8	6	1	4	2
prefix	1	4	8	16	22	23	27	29

Se construye en $\mathcal{O}(n)$ con una sola pasada:
$\texttt{prefix}[i] = \texttt{prefix}[i-1] + \texttt{array}[i]$.

Prefix sums: consultas en $\mathcal{O}(1)$

La suma de un rango es una **resta de dos prefijos**:

$$\texttt{sum}_q(a, b) = \texttt{prefix}[b] - \texttt{prefix}[a-1]$$

Por ejemplo: $\texttt{sum}_q(3, 6) = \texttt{prefix}[6] - \texttt{prefix}[2] = 27 - 8 = 19$.

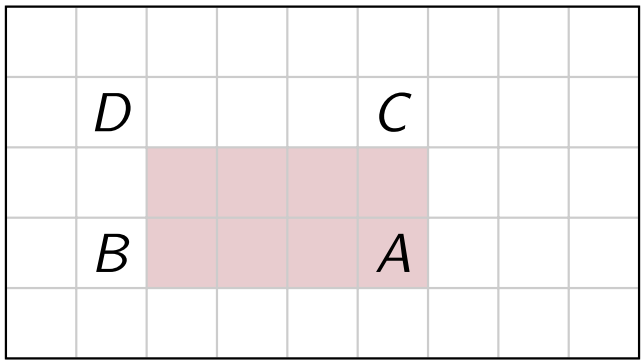
Código C++

```
int sum(int a, int b) {
    if (a == 0) return prefix[b];
    return prefix[b] - prefix[a-1];
}
```

Construimos **una sola vez** en $\mathcal{O}(n)$, y después cada consulta sale en $\mathcal{O}(1)$.

Prefix sums en 2D

La misma idea funciona en una matriz: precalculamos $S(X)$, la suma del rectángulo desde la esquina superior izquierda hasta la posición X.



$$\text{suma del rectángulo gris} = S(A) - S(B) - S(C) + S(D)$$

Cada consulta rectangular también sale en $\mathcal{O}(1)$.

¿Y si el array cambia?

Hasta acá el array era **estático**. **¿Qué pasa si entre consultas nos piden actualizar un valor?**

- Array solo: update $\mathcal{O}(1)$, pero consulta $\mathcal{O}(n)$.
- Prefix sums: consulta $\mathcal{O}(1)$, pero cada update obliga a reconstruir todo: $\mathcal{O}(n)$.

Con n y q grandes, cualquiera de las dos opciones explota.

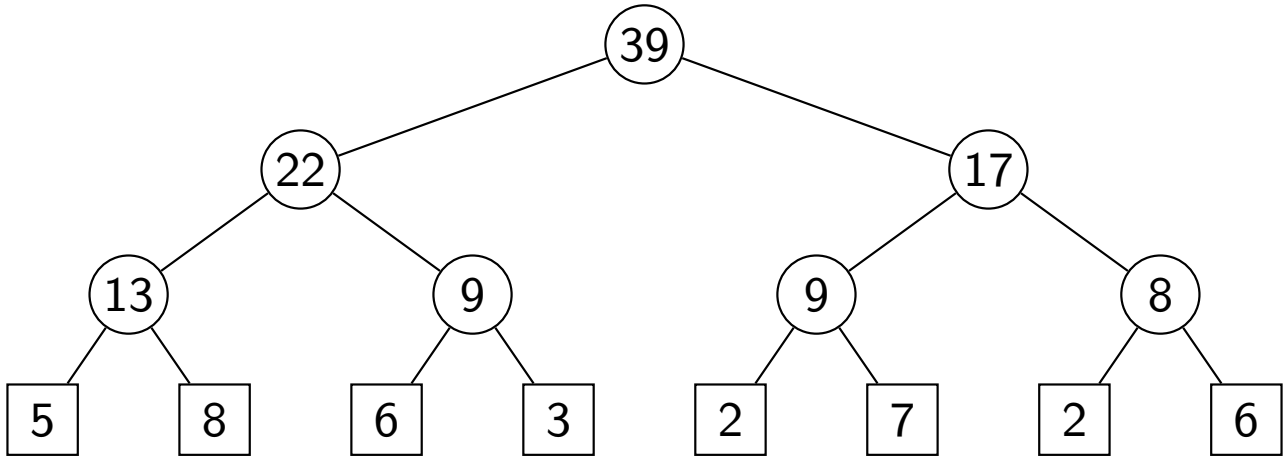
La pregunta clave, otra vez

Necesitamos una estructura donde **consulta y update** sean eficientes **a la vez**.

Segment tree

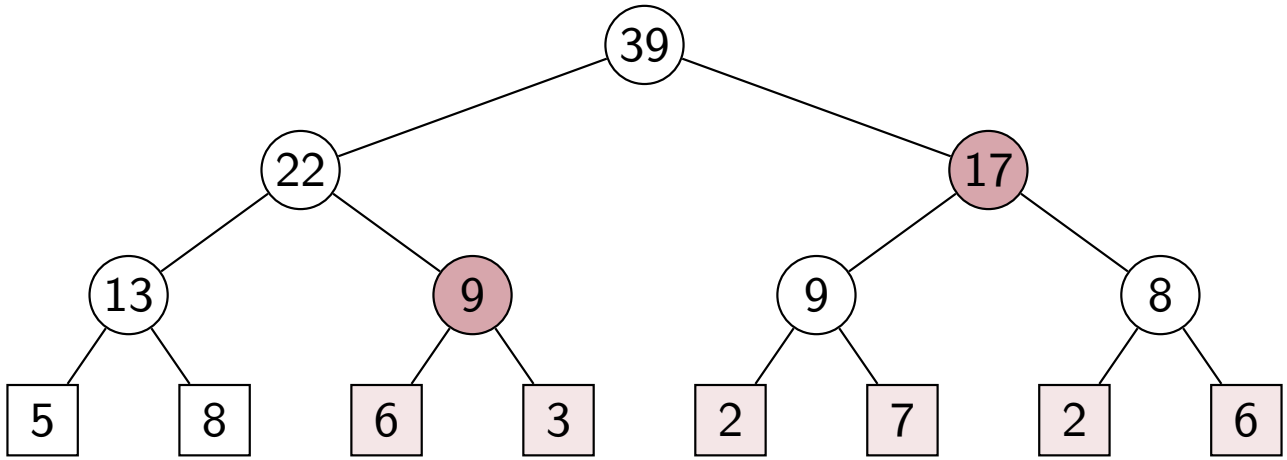
La estructura clásica para tener **consulta y update eficientes a la vez** es el **segment tree**:

- Árbol binario: las hojas son los elementos del array.
- Cada nodo interno resume su rango (suma, mín, máx, gcd, . . .).
- Consulta y update en $\mathcal{O}(\log n)$.



Segment tree: consulta y update en $\mathcal{O}(\log n)$

Consulta: todo rango se cubre con $\mathcal{O}(\log n)$ nodos (a lo sumo 2 por nivel). Por ejemplo, $\texttt{sum}_q(2, 7) = 9 + 17 = 26$:



Update: al cambiar una hoja, solo hay que recalcular el **camino hasta la raíz**: $\mathcal{O}(\log n)$ nodos.

Segment tree: código

Se guarda en un array de $2n$ (n potencia de 2): la raíz es `tree[1]`, los hijos de `tree[`k`]` son `tree[`$2k$`]` y `tree[`$2k+1$`]`, y las hojas van de `tree[`n`]` a `tree[`$2n-1$`]`.

Consulta: $\texttt{sum}_q(a, b)$

```
int sum(int a, int b) {
  a += n; b += n;
  int s = 0;
  while (a <= b) {
    if (a%2 == 1) s += tree[a++];
    if (b%2 == 0) s += tree[b--];
    a /= 2; b /= 2;
  }
  return s;
}
```

Update: sumar x en k

```
void add(int k, int x) {
  k += n;
  tree[k] += x;
  for (k /= 2; k >= 1; k /= 2)
    tree[k] = tree[2*k]
            + tree[2*k+1];
}
```

Para mínimo: cambiar las sumas por min.

Extra 1: compresión de índices

¿Y si los índices son enormes? Por ejemplo, posiciones hasta 10^9: no podemos crear un array de 10^9 elementos.

Si conocemos **de antemano** todos los índices que se usan, los reemplazamos por $1, 2, 3, \ldots$ **conservando el orden** (si $a < b$, entonces $c(a) < c(b)$):

$$c(8) = 1, \qquad c(555) = 2, \qquad c(10^9) = 3$$

Con los índices comprimidos ya podemos usar cualquier estructura de este capítulo.

Extra 2: updates de rango

Situación inversa: **sumar x a todo un rango** $[a, b]$ y consultar valores sueltos. **Difference array:** las diferencias entre valores consecutivos (el array original es su prefix sum):

	0	1	2	3	4	5	6	7
array	3	3	1	1	1	5	2	2
diff	3	0	-2	0	0	4	-3	0

Sumar x en $[a, b]$ = tocar **solo 2 posiciones**: `diff`$[a] \mathrel{+}= x$ y `diff`$[b+1] \mathrel{-}= x$. Con un segment tree sobre `diff`, todo en $\mathcal{O}(\log n)$.

¿Qué estructura uso?

Estructura	**Consulta**	**Update**	**Sirve para**
Prefix sums	$\mathcal{O}(1)$	—	sumas, estático
Segment tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	suma, mín, máx, ...

La pregunta clave en acción

¿Qué operaciones necesita mi solución? ¿Hay updates o el array es estático? La respuesta elige la estructura sola.

Otras cosas que existen (para ir a buscar)

Nombres para googlear cuando esta clase les quede corta:

- **Fenwick tree (BIT):** sumas con updates en $\mathcal{O}(\log n)$, en $\sim$10 líneas y con constante chica (libro, pág. 86).
- **Sparse table:** mín/máx en un array estático con consultas $\mathcal{O}(1)$ (libro, pág. 85).
- **Segment tree con lazy propagation:** updates a un rango entero **y** consultas de rango, todo en $\mathcal{O}(\log n)$.
- **Segment tree implícito:** índices gigantes (10^{18}) sin comprimir: los nodos se crean a medida que se usan.
- **Sqrt decomposition:** partir el array en bloques de $\sqrt{n}$; menos eficiente pero muy flexible.
- **Mo's algorithm:** responder muchas consultas de rango **offline**, reordenándolas con astucia.

Problemas para practicar (CSES)

- Static Range Sum Queries
- Static Range Minimum Queries
- Dynamic Range Sum Queries
- Dynamic Range Minimum Queries
- Range Xor Queries
- Range Update Queries
- Forest Queries

(Hay pistas escritas en blanco al lado de cada problema: píntenlas con el mouse si se traban.)

Outline

Problema: los grupos de la fiesta

En una fiesta hay N personas, cada una en su propio grupo. Llegan Q eventos, de dos tipos:

- `u a b`: a y b se hacen amigos, y **los dos grupos se juntan en uno solo**.
- `? a b`: ¿están a y b en el **mismo grupo**?

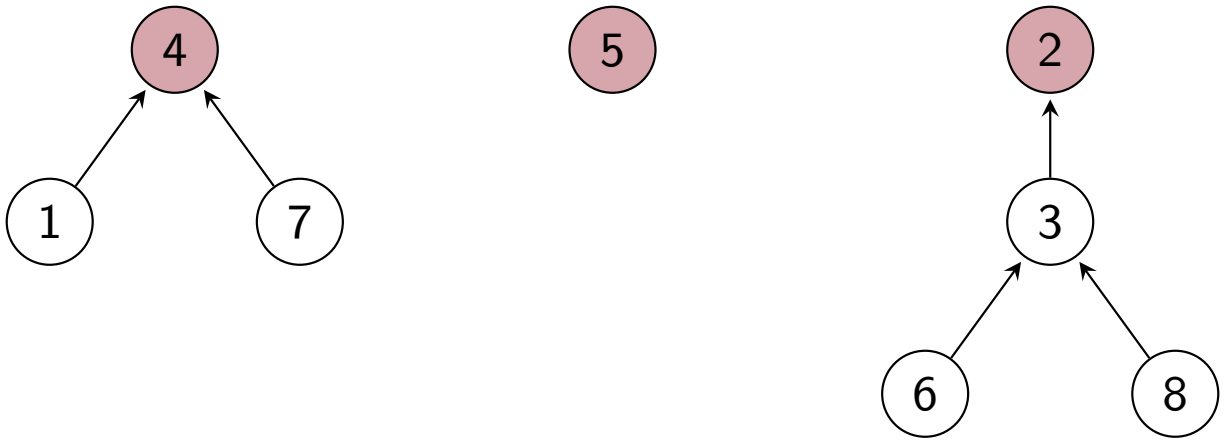
con $N, Q \leq 10^5$.

Para pensar

Si guardo el grupo de cada persona en un array, ¿cuánto cuesta **juntar** dos grupos?

La idea: un representante por grupo

Guardamos una **colección de conjuntos disjuntos** (nadie está en dos grupos a la vez). En cada conjunto elegimos un elemento **representante**, y todos los demás forman una **cadena** que lleva hasta él.

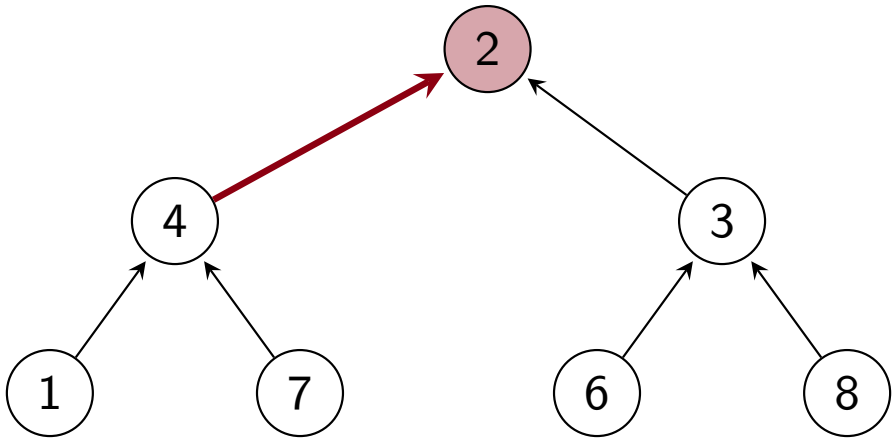


Acá los conjuntos son $\{1, 4, 7\}$, $\{5\}$ y $\{2, 3, 6, 8\}$, con representantes 4, 5 y 2. Para saber el representante de 6 seguimos la cadena $6 \to 3 \to 2$.

Dos elementos están en el mismo conjunto exactamente cuando tienen el mismo representante.

Unir dos conjuntos

Para unir dos conjuntos alcanza con **colgar el representante de uno del representante del otro**. Uniendo $\{1, 4, 7\}$ con $\{2, 3, 6, 8\}$:



El 2 pasa a ser el representante de todo $\{1, 2, 3, 4, 6, 7, 8\}$, y el viejo representante 4 ahora apunta a 2.

¿De cuál colgamos cuál? Unión por tamaño

Siempre colgamos el representante del conjunto **más chico** del representante del **más grande**. Así ninguna cadena se hace más larga que $\mathcal{O}(\log n)$.

DSU: código

`link[x]`: el siguiente en la cadena (o `x` mismo si es representante).
`sz[x]`: el tamaño del conjunto, solo válido en los representantes.

Inicialización

```
for (int i = 1; i <= n; i++) {
  link[i] = i;
  sz[i] = 1;
}
```

find y same

```
int find(int x) {
  while (x != link[x]) x = link[x];
  return x;
}
bool same(int a, int b) {
  return find(a) == find(b);
}
```

unite (unión por tamaño)

```
void unite(int a, int b) {
  a = find(a);
  b = find(b);
  if (a == b) return;
  if (sz[a] < sz[b]) swap(a,b);
  sz[a] += sz[b];
  link[b] = a;
}
```

Las tres operaciones cuestan $\mathcal{O}(\log n)$: lo único que hacen es recorrer una cadena.

Path compression: casi $\mathcal{O}(1)$

Cada vez que recorremos una cadena, la podemos **aplanar**: hacer que todos los nodos del camino apunten directo al representante.

find con compresión de caminos

```
int find(int x) {
  if (x == link[x]) return x;
  return link[x] = find(link[x]);  // guarda el resultado
}
```

Es **una línea más** y no cambia el resultado: solo deja el árbol más chato para las próximas consultas.

Costo

Con unión por tamaño **y** compresión de caminos, cada operación cuesta $\mathcal{O}(\alpha(n))$, donde α crece tan despacio que para cualquier n que entre en la memoria vale menos que 5: **en la práctica, constante**.

¿Para qué sirve el DSU?

Cada vez que un problema tenga **grupos que solo se unen** (nunca se separan), el DSU es la respuesta:

- **Componentes conexas** de un grafo, manteniéndolas mientras se agregan aristas.
- Contar **cuántos grupos** quedan, o el **tamaño del grupo más grande** (se lleva con un contador al unir).
- Detectar si una arista **cierra un ciclo**: pasa exactamente cuando sus dos puntas ya están en el mismo conjunto.
- **Algoritmo de Kruskal** para el árbol generador mínimo (libro, pág. 143).

La pregunta clave, de nuevo

Unir y consultar son baratísimos... pero **separar un conjunto no se puede**. Ojo con eso al elegirlo.

Otras cosas que existen (para ir a buscar)

Si esta estructura les gusta, los nombres que siguen:

- **DSU con rollback:** permite **deshacer** las últimas uniones (se resigna la compresión de caminos).
- **DSU sobre grafos que se borran:** muchos problemas de "se cae un puente" se resuelven **procesando las consultas al revés**, y así los borrados pasan a ser uniones.
- **Small to large:** la misma idea de "el chico se cuelga del grande" aplicada a fusionar sets, mapas o listas.
- **DSU con pesos / bipartito:** guardar además la **relación** de cada elemento con su representante.

Problemas para practicar (CSES)

- Road Construction
- Building Roads
- Road Reparation

(Las pistas están escritas en blanco al lado de cada problema.)

Outline

Bitset: un array de bits

Un **bitset** es un array donde cada valor es 0 o 1, con el tamaño fijado al compilar:

Código C++

```
bitset<10> s;
s[1] = 1; s[3] = 1;
s[4] = 1; s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

¿Por qué no un array común? Memoria: cada elemento usa **1 bit**. Un array de `int` con n valores usa $32n$ bits; el bitset usa n.

Bitset: operar de a 64 bits

La verdadera gracia: los **operadores de bits** procesan el bitset entero en bloques de 64 bits $\Rightarrow$ unas **64 veces menos operaciones** que el mismo loop sobre un array.

Código C++

```
bitset<10> a(string("0010110110")); // se lee de derecha a izq.
bitset<10> b(string("1011011000"));
cout << a.count() << "\n"; // 5 (cantidad de unos)
cout << (a&b) << "\n";     // 0010010000
cout << (a|b) << "\n";     // 1011111110
cout << (a^b) << "\n";     // 1001101110
```

Útil para acelerar soluciones que manejan conjuntos de $\{0, \ldots, n-1\}$ o tablas grandes de verdadero/falso.

Outline

Problema: la tabla del torneo

Problema de ejemplo: en un torneo hay N equipos. Llegan Q eventos, de dos tipos:

- `p e x`: el equipo e pasa a tener x puntos.
- `? k`: ¿qué equipo está en la **posición** k de la tabla?

con $N, Q \leq 10^5$.

Reordenar la tabla en cada consulta: $\mathcal{O}(N \log N)$ por consulta $\Rightarrow$ no entra en tiempo. Un `set` mantiene el orden... pero **no sabe responder "¿quién va k-ésimo?"**.

Para pensar

¿Qué le tendríamos que poder pedir a un `set` para que este problema salga?

Un set con superpoderes

El `set` responde rápido "¿está x?". Pero:

- **¿Cuál es el k-ésimo elemento más chico?**
- **¿Cuántos elementos son menores que x?**

Con un `set` común no hay forma eficiente de responderlas.

El compilador g++ trae estructuras extra que **no son parte del estándar** de C++: las **policy-based data structures**.

Para usarlas

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

indexed_set

Definimos un `indexed_set`: igual que un `set`, pero se puede **indexar como si fuera un array ordenado**:

Definición y uso

```
typedef tree<int,null_type,less<int>,rb_tree_tag,
    tree_order_statistics_node_update> indexed_set;

indexed_set s;
s.insert(2); s.insert(3);
s.insert(7); s.insert(9);
```

La definición se copia tal cual. Sigue siendo un set: **no admite repetidos**.

Truco: convertirlo en multiset

Guardar **pares** (*valor*, *id*) con un *id* único por inserción (cambiando `int` por `pair<int,int>`): los repetidos pasan a ser pares distintos.

order_of_key y find_by_order

Con $s = \{2, 3, 7, 9\}$:

Las dos operaciones nuevas, ambas en $\mathcal{O}(\log n)$

```
auto x = s.find_by_order(2); // elemento en la posicion 2
cout << *x << "\n";          // 7

cout << s.order_of_key(7) << "\n"; // 2 (posicion del 7)
cout << s.order_of_key(6) << "\n"; // 2 (posicion que tendria)
cout << s.order_of_key(8) << "\n"; // 3
```

Si el elemento no está, `order_of_key` devuelve la posición que **tendría**: nos regala "¿cuántos elementos son menores que x?".

Y el torneo sale solo: guardar {`-puntos, equipo`} (negativos: el puntero primero); update = `erase` + `insert`, posición k = `find_by_order`. Para practicar: Josephus Problem II (CSES).

Consultas

Pueden consultarme durante esta semana, o me pueden enviar un mail a:

- marianoferesin@gmail.com