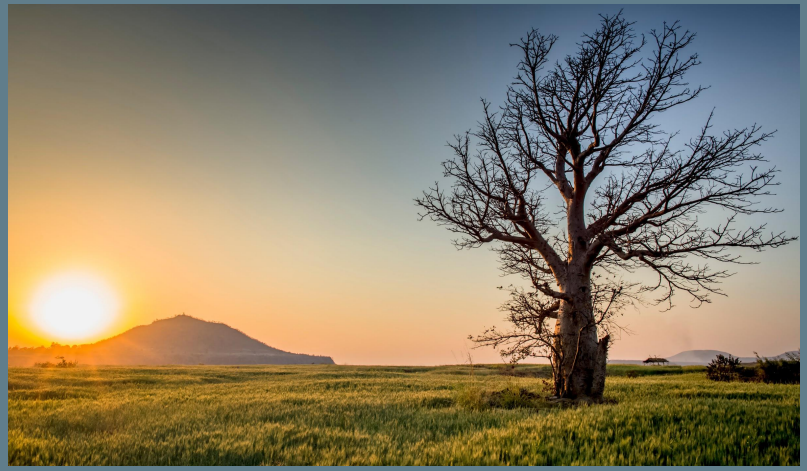
Estructuras II

Román Castellarin

Índice

Segment Trees



Problema

Se tiene un array A de largo N, y una operación asociativa ★.

Se quiere diseñar una estructura que soporte rápidamente las siguientes operaciones (abstractas):

- `query(a, b): A[a] ★ A[a+1] ★ ... ★ A[b-2] ★ A[b-1]`
- `set(i, x):  A[i] = x`

Veamos que pueden ser implementadas en O(log N) cada una.

Segment Tree

Un Segment Tree es una estructura de datos que sirve para realizar consultas sobre rangos.

Primero, necesitamos un elemento neutro $\underline{0}$ tal que $\underline{0} \bigstar x = x \bigstar \underline{0} = x$

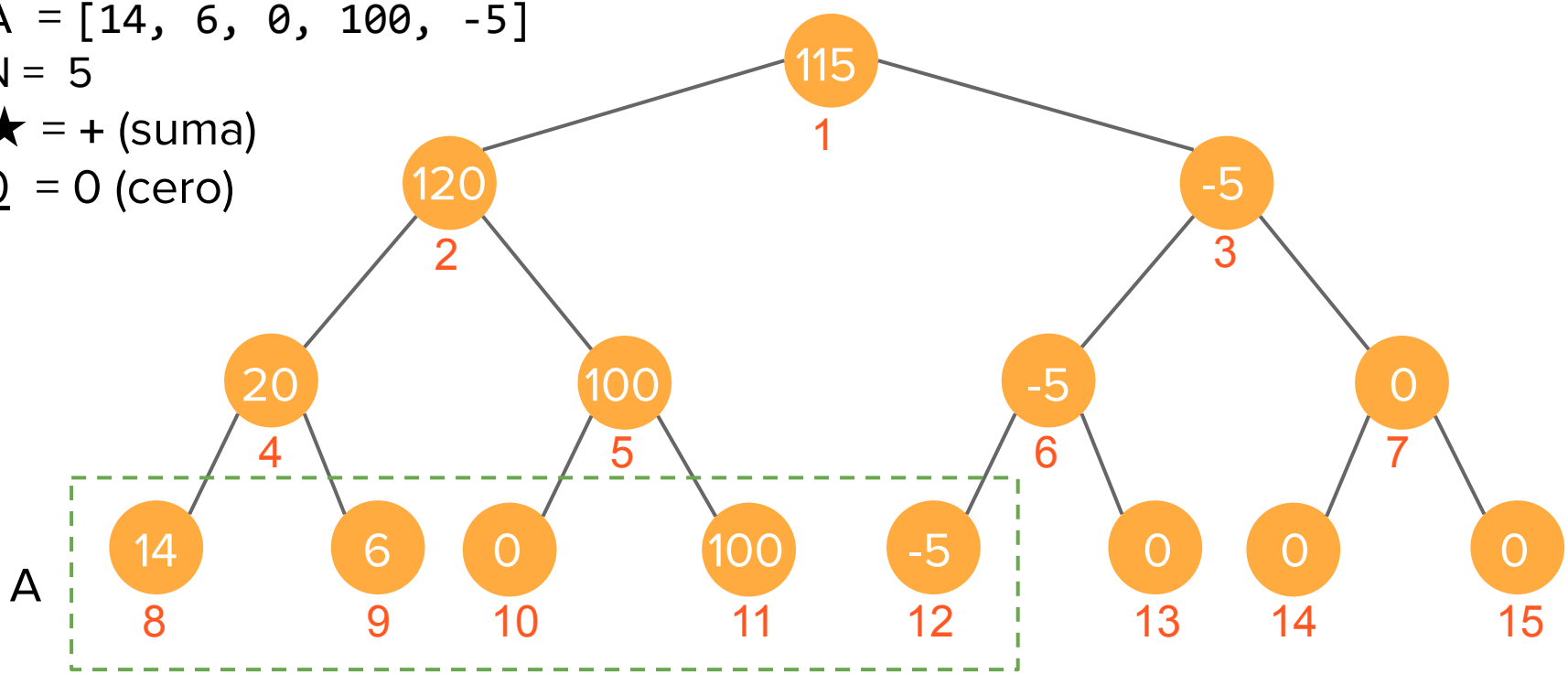
nota: si ★ no tiene uno, lo inventamos.

Luego, un Segment Tree sobre un array A consiste de un árbol binario lleno cuyas hojas tienen los valores de A (completamos con $\underline{0}$)

Cada nodo interno, tiene el resultado de aplicar ★ a sus dos nodos hijos.

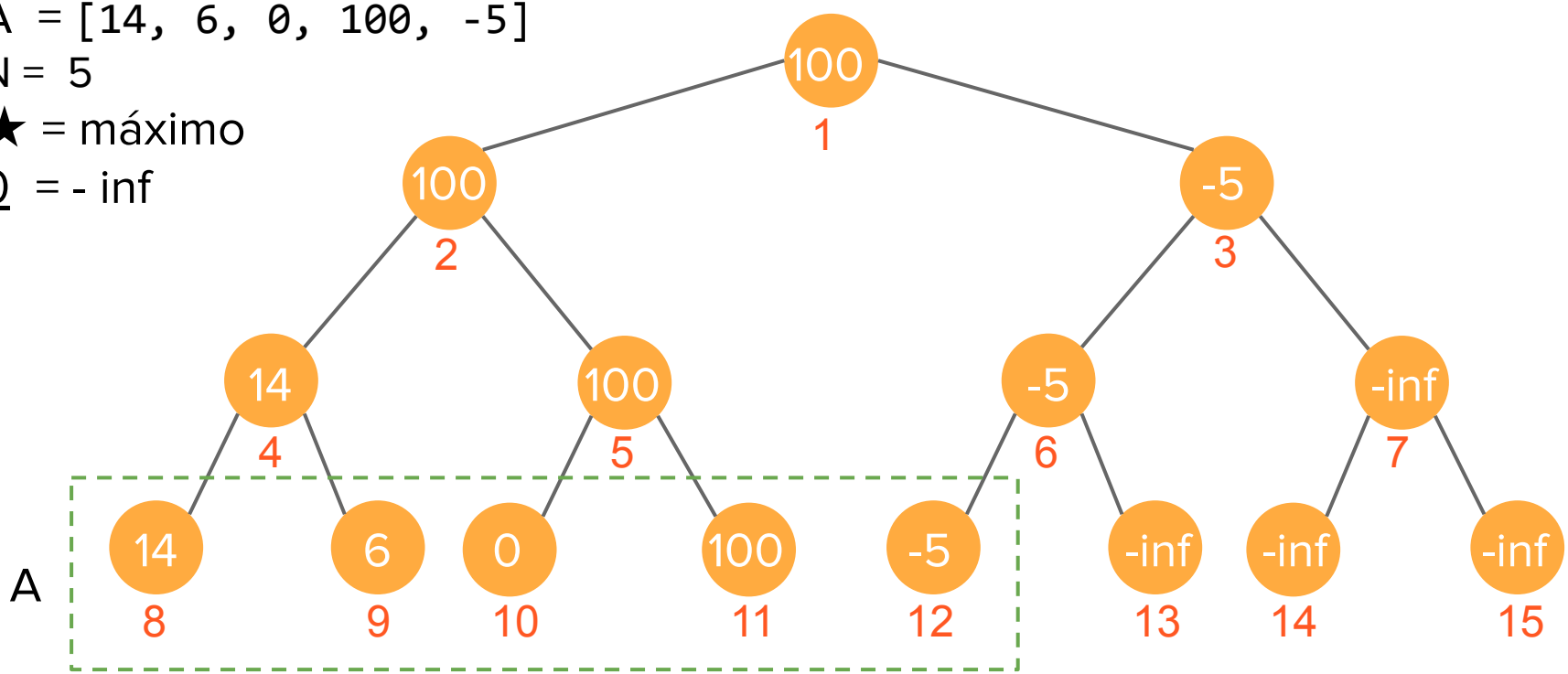
Ejemplo

A = `[14, 6, 0, 100, -5]`
N = 5
★ = + (suma)
0 = 0 (cero)



Ejemplo

A = [14, 6, 0, 100, -5]
N = 5
★ = máximo
0 = - inf



Segment Tree

Indexamos los nodos desde el 1, por nivel, así obtenemos las siguientes relaciones:

```
pad(i) = i / 2
izq(i) = 2 * i
der(i) = 2 * i + 1
```

Notemos que la altura del Segment Tree es O(log N).

Al modificar una hoja, sólo es necesario actualizar los valores de los ancestros ⇒ set en O(log N).

Segment Tree

El rango (a cargo) de un nodo es la porción de A sobre la cual acumula ★.

En un mismo nivel, puede haber a lo sumo 2 nodos cuyo rango interseque parcialmente el rango de la query, y es el único caso en que se recursa.

Por tanto, en una consulta se visitan a lo sumo 4 nodos por nivel.

Así, cada query puede ser respondida en O(log N).

Implementación

sea MAXN la primer potencia de 2 mayor o igual a N

```
node ST[2*MAXN]; // inicializados en 0

void set(int i, node v){
    i += MAXN;
    ST[i] = v;
    for(i = pad(i); i; i = pad(i))
        ST[i] = ST[izq(i)] ★ ST[der(i)]
}
```

Implementación

sea [A..B) el rango de la query

```
node query(int n, int a, int b){
    if( b <= A or B <= a ) return Q;
    if( A <= a and b <= B ) return ST[n];
    int m = (a + b) / 2;
    return query(izq(n), a, m) ★ query(der(n), m, b);
}
```

Consultas complejas

Los nodos pueden almacenar más información.

Por ejemplo, dado un array A de 0s y 1s, querría poder consultar

"en el rango [a, b), ¿cuál es el subintervalo de 1s consecutivos más largo?"

Consultas complejas

Para cada nodo, me refiero a su rango a cargo:

Si sé - el prefijo más largo de 1 consecutivos,

- el sufijo
- la respuesta calculada hasta el momento, y
- si el rango está completo

de los dos nodos hijos, entonces puedo calcular el nodo padre.

```
struct node{
    int prefix;
    int middle;
    int suffix;
    bool full;
};
```

Consultas complejas

```
const node UNO  = {1, 1, 1, true};
const node CERO = {0, 0, 0, false};

node operator+(const node &a, const node &b){
    node ans;
    ans.prefix = a.prefix + (a.full ? b.prefix : 0);
    ans.middle = max(a.suffix + b.prefix, max(a.middle, b.middle)),
    ans.suffix = b.suffix + (b.full ? a.suffix : 0);
    ans.full   = a.full and b.full;
    return ans;
}
```

Merge Tree (otro ejemplo de nodos complejos)

"en el rango [a, b), ¿cuántos elementos son menores a k?"

```
using node = vector<int>;
const node NEUTRO = node();

node operator+(const node &a, const node &b){
    node ans(a.size() + b.size());
    merge(a.begin(), a.end(), b.begin(), b.end(), ans.begin());
    return ans;
}
```

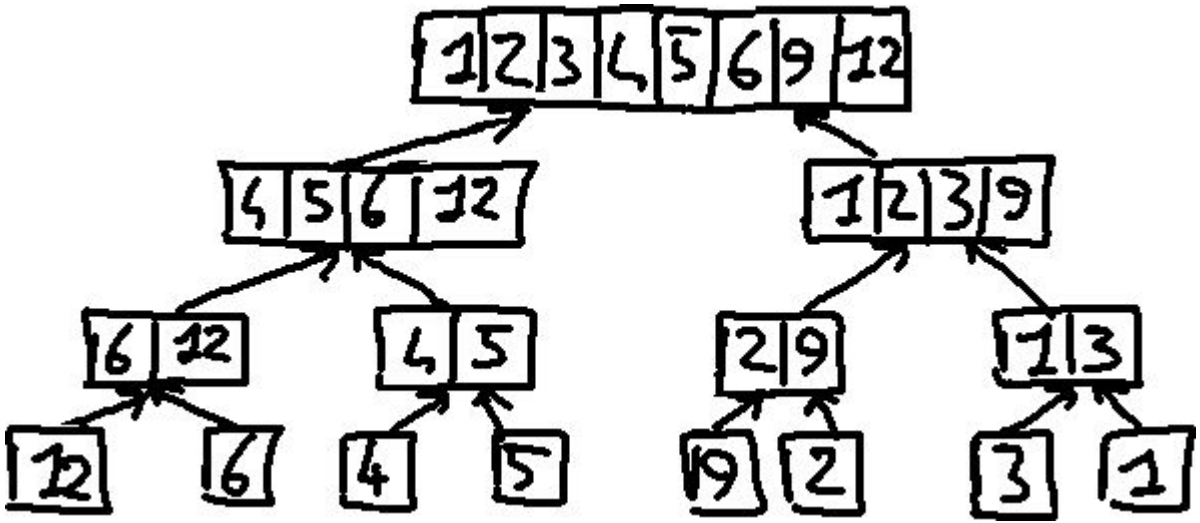
La consulta es $O(\log^2 N)$: un factor log extra por búsqueda binaria.

La memoria total es $O(N \log N)$.

Merge Tree

Es un caso particular de Segment tree con ★ = merge y $\underline{0}$ = []

¡Los sets son caros! O(N) cada uno.



Lazy Creation

Alternativamente, podríamos no tener un array inicial sino que iremos poblando el Segment Tree a través de repetidas llamadas a set.

Si los índices no están acotados (por ejemplo, sólo sabemos que son ints) No podremos representar un Segment Tree explícitamente.

Una alternativa que nos saca rápido del apuro es utilizar un
`unordered_map<int, node> ST;` ← *`node() debe ser el elem. neutro`*
en lugar de
`node ST[2 * MAXN];`

Lazy Creation

Sin embargo, más eficiente es tener una estructura enlazada:

```
struct ST{
    int left = -1, right = -1;
    node value;
};
```

Usamos índices de un array global de nodos en lugar de punteros simplemente por cuestiones de performance.

Lazy Creation

Podriamos obtener los hijos de un nodo así:

```
int NextNode;
ST Pool[MAXNODES]; ← máxima cantidad de nodos ≈ Q * log RANGOINDICES

inline int left(int n){
    return Pool[n].left == -1 ? Pool[n].left = NextNode++ : Pool[n].left;
}

inline int right(int n){
    return Pool[n].right == -1 ? Pool[n].right = NextNode++ : Pool[n].right;
}
```

Coordinates Compression

Muchas veces podemos ahorrarnos tener que hacer un SegTree con lazy creation si las queries son conocidas de antemano, ya que podemos hacer compresión de coordenadas. Ejemplo:

1 8 7 6 4 7 16 2 4 2 8 → 0 5 4 3 2 4 6 1 2 1 5

```
map<int, int> comp;
...
int i = 0;
for(auto &s: comp)
    s.second = i++;
```

Lazy Propagation

Queremos extender el segment tree agregando una operación de **actualización** en rangos. Tres ejemplos:

```
update(a, b, x): A[i]  = x   ∀ a ≤ i < b      (set in range)
update(a, b, x): A[i] += x   ∀ a ≤ i < b      (add in range)
update(a, b):    A[i] ^= 1   ∀ a ≤ i < b      (flip range)
```

Esta operación puede simularse mediante b-a llamados a set.
Sin embargo, eso resulta en una complejidad de O(N log N).
Veremos que es posible implementarlo en O(log N)

Lazy Propagation

La idea es tener nodos lazy, que almacenen toda la información necesaria para actualizar el nodo, pero que no lo hagan hasta que se vean obligados a hacerlo.

Un nodo lazy sólo debe ser expandido cuando sea necesario acceder a él, (i.e., durante una query o update).

En una expansión, se recalcula el valor del nodo, y se propaga la información de la actualización a sus hijos, haciéndolos lazy.

Lazy Propagation

La operación de actualización debe ser componible consigo misma.

Por ejemplo, para "sumar k":
"sumar 15" ∘ "sumar 5" = "sumar 20"

Para "reemplazar por k":
"reemplazar por 15" ∘ "reemplazar por 5" = "reemplazar por 15"

Para "negar el bit":
"negar" ∘ "negar" = id (no hacer nada)

Lazy Propagation

La operación de consulta debe ser recalculable desde la información lazy:
Supongamos que ★ = min, luego:

Para "sumar k":
valor nuevo = valor viejo + k (el mínimo en el rango aumentó en k)

Para "reemplazar por k":
valor nuevo = k (el mínimo en el rango es k)

Para "negar": ? ← *¿qué necesitaríamos?*

Implementación

(llamar sólo en nodos lazy)

```
void propagate(int n){
    valor[n] = recalcular( valor[n], datos_lazy[n] )
    if( n no es hoja ){
        lazy[izq(n)] = true
        datos_lazy[izq(n)] = componer(datos_lazy[izq(n)], datos_lazy[n])
        lazy[der(n)] = true
        datos_lazy[der(n)] = componer(datos_lazy[der(n)], datos_lazy[n])
    }
    lazy[n] = false
    datos_lazy[n] = vacio
}
```

Ejemplo complicado

★ = suma

`update(a, b, p, q): A[a+i] += p + q*i  ∀  0 ≤ i < b-a`

Si un nodo n es lazy, con rango [a, b) y `datos_lazy[n] = (p, q)`,
al propagarlo resulta:

```
valor[n]           += p * r + q * r * (r-1) / 2
datos_lazy[izq(n)] += (p, q)
datos_lazy[der(n)] += (p + q * r / 2, q)
```

donde `r = b-a`

Persistencia

Con cada actualización, estamos mutando el estado del Segment Tree.

Una estructura de datos **persistente** permite generar una versión nueva con cada actualización sin modificar la anterior, de modo que sea posible consultar cualquier versión en cualquier momento.

Un Segment Tree puede fácilmente ser tornado persistente ya que en cada actualización sólo se modifican O(log N) nodos, y estos están enlazados mediante punteros.

Implementación

Sea `[A..B)` el rango a consultar

```
node query(ST *n, int a, int b){
    if( not n or b <= A or B <= a ) return 0;
    if( A <= a and b <= B ) return n->value;
    int m = (a + b) / 2;
    return query(n->izq, a, m) ★ query(n->der, m, b);
}
```

Implementación

Sean `I X` los parámetros del set, definimos ahora recursivamente:

```
ST* set(ST *n, int a, int b){
    if( a + 1 == b ) return new ST(X);
    int m = (a + b) / 2;
    if( I < m ) return new ST(set(n->izq, a, m), n->der); ← calcula ★
    else        return new ST(n->izq, set(n->der, m, b)); ← calcula ★
}
```

Problema ejemplo

Dado un array `A[1..N]` de enteros responder consultas del estilo

`query(i, j, k)`:
¿qué valor estaría en `A[i+k]` *si* `A[i..j]` *estuviese ordenado?*

La solución trivial es O(M N log N).
Se puede resolver en O(N + M log N).

Problema ejemplo

Solución:

- Comprimir coordenadas (mapear comp: `A[i] ⇔ [0..N-1]`)

- Por cada i, armar un segment tree `ST(i)` sobre `A[1..i]` que me permita calcular la cantidad de elementos menores a k (almacenando las frecuencias). Notemos que `ST(i-1)` y `ST(i)` difieren sólo en un set.

- Cada `query(i, j)` puede ser respondida en O(log N) haciendo búsqueda binaria en `ST(j) - ST(i-1)`.

Problema

Dado un array `A[1..N]`, se desea implementar la siguiente operación:

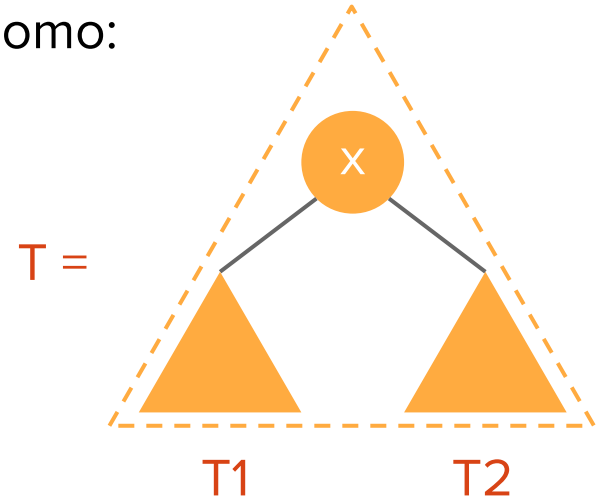
`update(i, j)`: revierte el subarray `A[i..j]`

La implementación trivial por operación es O(N)
Veamos que puede ser llevado a O(log N)

Árboles Binarios

Un **árbol binario (AB)** es una estructura definida como:

- vacío
- un dato, y dos árboles binarios



Un AB se dice (min) **heap** sii:

T1 y T2 son heaps y $\forall\ u \in T1, v \in T2 : x < u, v$

Un AB se dice **de búsqueda** sii:

T1 y T2 son de búsqueda y $\forall\ u \in T1, v \in T2 : u < x < v$

Treap

Una **Treap** es un AB donde cada nodo almacena un par (k, p)

Al considerar sólo las claves (k) de cada nodo, resulta de búsqueda.
Al considerar sólo las prioridades (p) de cada nodo, resulta una heap.

Para una treap de N nodos, si las claves fueron escogidas al azar de 1..N, entonces la altura esperada de la treap es O(log N)

Las treaps son estrictamente más poderosas que los segment trees.

Operaciones fundamentales

(explicit) `split(t, x)`:

devuelve `(t1, t2)` donde `t1` tiene todos los valores ≤ x.

(implicit) `split(t, n)`:

devuelve `(t1, t2)` donde `t1` tiene los primeros n elementos de `t`.

`merge(t1, t2)`:

devuelve la unión de las treaps `t1` y `t2` considerando que `t1` va a la izquierda de `t2`.

Explicit split funciona sólo cuando las claves forman un ABB

Operaciones derivadas

A partir de estas dos, pueden definirse más operaciones...

`insert(t, x)`: inserta el nodo x en t

`search(t, k)`: devuelve (si existe) el nodo de clave k

`erase(t, k)`: elimina (si existe) el nodo de clave k

etc...

Implementación

```
using treap = node *;

struct node{
    T value;     ← el valor de acumular ★ en el subárbol
    int size;    ← es útil guardar el tamaño de cada subárbol
    int p, k;    ← prioridad - clave
    treap l, r;
}

inline int size(treap t)     { return t ? t->size : 0; }
inline void upd_size(treap t){ if(t) t->size = size(t->l) + size(t->r); }
```

Implementación

```
void split (treap t, int key, treap &l, treap &r) {
    if (!t)                 l = r = nullptr;
    else if (key < t->k)  split (t->l, key, l, t->l),  r = t;
    else                  split (t->r, key, t->r, r),  l = t;
    upd_size(t);
}

void merge (treap &t, treap l, treap r) {
    if (!l || !r)             t = l ? l : r;
    else if (l->p > r->p)    merge (l->r, l->r, r),  t = l;
    else                     merge (r->l, l, r->l),  t = r;
    upd_size(t);
}
```

Implementación

Podemos suponer que las claves son implícitas, y están dadas por los índices de los nodos en un recorrido en-orden.

```
void split (treap t, treap &l, treap &r, int key) {
    if (!t)                  return void( l = r = 0 );
    if (key <= size(t->l))   split (t->l, l, t->l, key),              r = t;
    else                     split (t->r, t->r, r, key-1-size(t->l)), l = t;
    upd_size(t);
}
```

Lazy Propagation

Podemos utilizar la misma técnica que usamos con Segment Tree.

```
struct node{
    int p, value;
    bool lazy;
    T lazy_data;
    treap l, r;
};
```

Para nuestro problema alcanza:

```
struct node{
    int p, value;
    bool rev;
    treap l, r;
};
```

porque rev ∘ rev = id

Lazy Propagation

```
void push(treap t){
    swap(t->l, t->r);
    if( t->l ) t->l->rev ^= true;
    if( t->r ) t->r->rev ^= true;
    t->rev = false;
}
```

Problema

https://www.spoj.com/problems/ALLIN1

Inicialmente se cuenta con una lista vacía, y luego se realizan operaciones de uno de los siguientes tipos:

1) insertar un valor x a la lista (en orden)
2) eliminar un valor x de la lista
3) imprimir el índice de un valor x
4) imprimir el valor de un índice i

En todo momento no hay repeticiones en la lista.

Problema

https://www.spoj.com/problems/COUNT1IT

Inicialmente se cuenta con una bolsa con n números, y luego se realizan operaciones de uno de los siguientes tipos:

1) sea y la cantidad de elem. menores o iguales a un valor x, insertar x+y a la bolsa
2) imprimir la cantidad de elementos menores o iguales a un valor x
3) imprimir el k-ésimo elemento más chico de la bolsa (considerar repeticiones)

En todo momento se deben considerar repeticiones.

Gracias!