

Range Queries Con Updates

Segment Tree y otras variantes

Ulises López Pacholczak

September 17, 2023

1 Introducción

2 Segment Tree

- Explicación de la Estructura
- Ejemplos de Queries y Updates
- Código
- Complejidad
- Resolución de un Problema

3 Generalización del Segment Tree

- Monoides
- Implementación Generalizada del Segment Tree

4 Resolviendo Más Problemas

Introducción

Ya sabemos como responder queries de suma en rango, en $O(n)$ tiempo de procesamiento ($O(n * m)$ si es una tabla) y $O(1)$ tiempo de consulta.

Introducción

Ya sabemos como responder queries de suma en rango, en $O(n)$ tiempo de procesamiento ($O(n * m)$ si es una tabla) y $O(1)$ tiempo de consulta.

También, se pueden resolver queries de máximo o mínimo en rangos con $O(n \log n)$ tiempo de procesamiento y $O(1)$ tiempo de consulta.

Introducción

Ya sabemos como responder queries de suma en rango, en $O(n)$ tiempo de procesamiento ($O(n * m)$ si es una tabla) y $O(1)$ tiempo de consulta.

También, se pueden resolver queries de máximo o mínimo en rangos con $O(n \log n)$ tiempo de procesamiento y $O(1)$ tiempo de consulta.

¿Pero esto es suficiente para todos los problemas?

Suma en Rangos con Updates

CSES 1648 - Dynamic Range Sum Queries

Dado un arreglo de n enteros, tu tarea es procesar q queries de los siguientes tipos:

- 1 Actualizar el valor en la posición k a un valor u
- 2 Cual es la suma de los valores en el rango $[a, b]$?

Tanto q y n pueden ser como máximo $2 * 10^5$

<https://cses.fi/problemset/task/1648>

Con lo que ya sabemos...

Por cada query de tipo 1, debemos realizar un reprocesamiento de $O(n)$.
Por ende, la complejidad del problema nos quedaría $O(n^2)$, lo mismo que hacerlo con fuerza bruta.

Con lo que ya sabemos...

Por cada query de tipo 1, debemos realizar un reprocesamiento de $O(n)$. Por ende, la complejidad del problema nos quedaría $O(n^2)$, lo mismo que hacerlo con fuerza bruta.

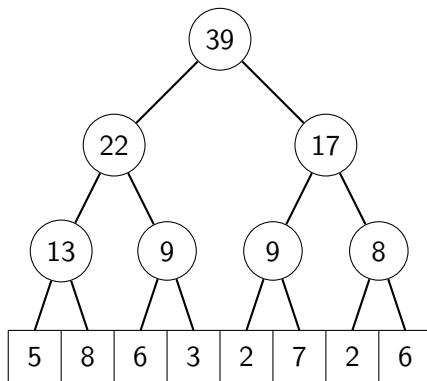
Es claro que necesitamos otra estructura de datos para poder manejar los updates de forma eficiente...

Segment Tree

El Segment Tree es una estructura de datos que nos permite realizar queries en rangos y updates en $O(\log n)$.

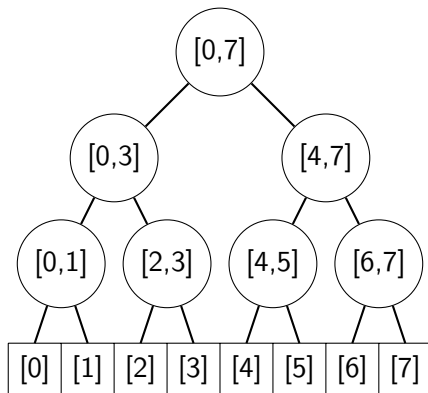
Segment Tree

El Segment Tree es una estructura de datos que nos permite realizar queries en rangos y updates en $O(\log n)$.



Funcionamiento

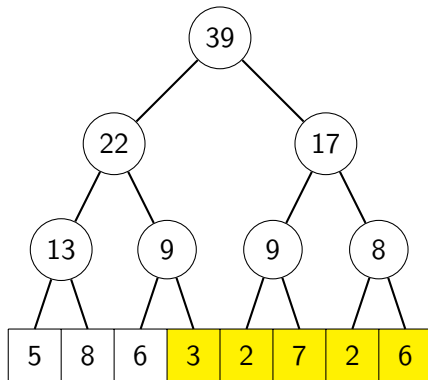
Cada nodo representa el valor de realizar una operación (en el caso del problema original, la suma) en un rango del arreglo.



Guardado de Segment Tree en Memoria

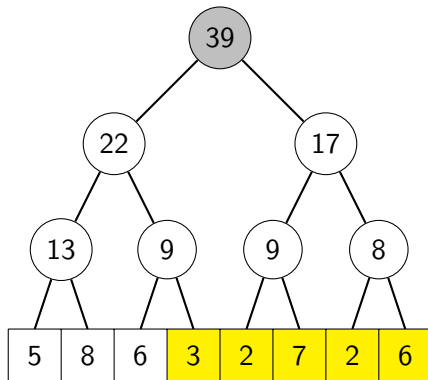
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Ejemplo de Query



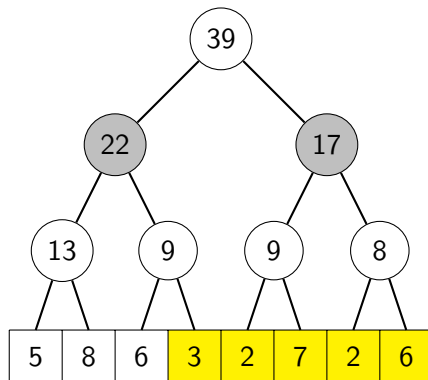
Suma total = 0

Ejemplo de Query



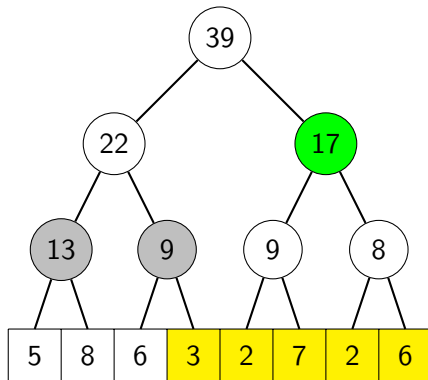
Suma total = 0

Ejemplo de Query



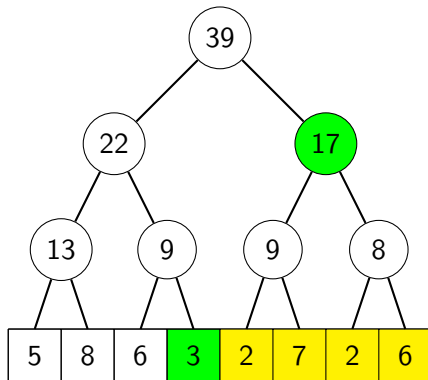
Suma total = 0

Ejemplo de Query



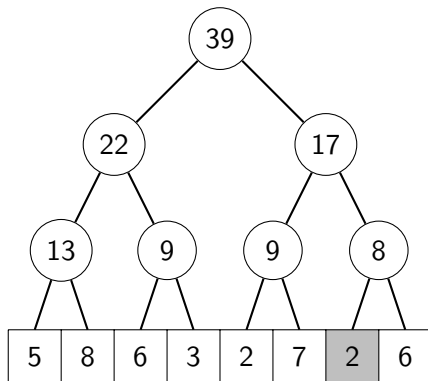
Suma total = 17

Ejemplo de Query



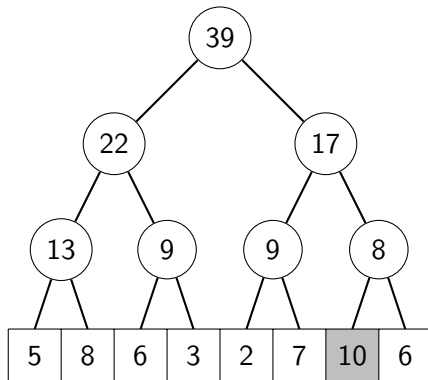
Suma total = 20

Ejemplo de Update



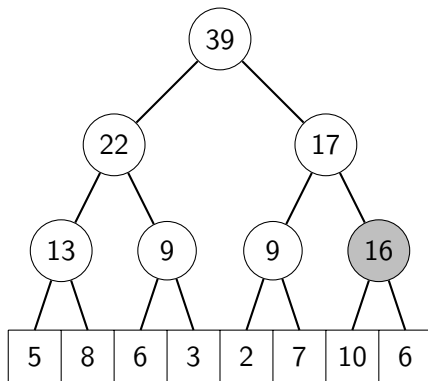
Actualizaremos la posición 6 del arreglo con el valor 10.

Ejemplo de Update



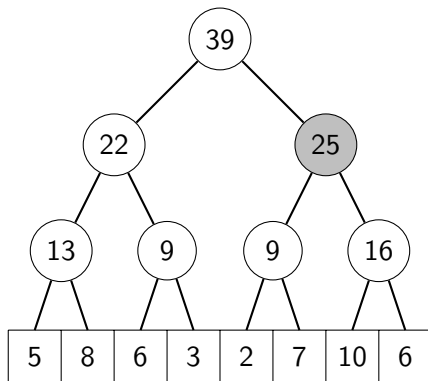
Actualizamos nodo 14

Ejemplo de Update



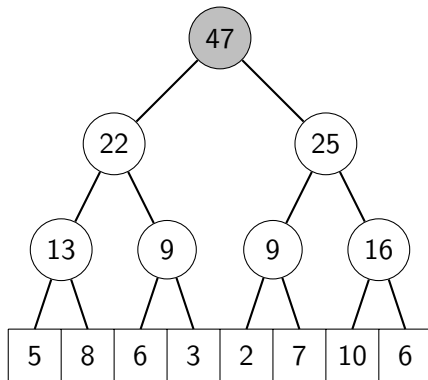
Actualizamos nodo 7 $\lfloor 14/2 \rfloor$

Ejemplo de Update



Actualizamos nodo 3 $\lfloor 7/2 \rfloor$

Ejemplo de Update



Actualizamos nodo 1 $\lfloor 3/2 \rfloor$

Version Iterativa

```
int retrieve(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;
}

void update(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];
    }
}
```

Version Recursiva

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}

void update(int pos, int k, int x, int y, int val) {
    if (x == y) { segTree[k] = val; return; }
    int d = (x + y) / 2;
    if (pos <= d) update(pos, k*2, x, d, val);
    else update(pos, k*2+1, d+1, y, val);
    segTree[k] = segTree[k*2] + segTree[k*2+1];
}
```


Complejidad

- Construcción: $O(n)$ (De todas formas, solemos construirlo usando la operación update)
- Query: $O(\log n)$
- Update: $O(\log n)$
- Memoria: $O(2n)$

Puede notarse que el árbol que construimos es un árbol binario completo, por lo que la cantidad de nodos es $2n - 1$, y tiene una profundidad de $\log_2(n)$.

Entendiendo la Complejidad de Query

Es fácil creer que la complejidad de query es mayor a $O(\log n)$, ya que en cada nivel del árbol se podría llamar a los dos nodos hijos. Pero, en total, notemos que en cada nivel solo se van a estar llamando como máximo 4 nodos en su versión recursiva, y 2 en la versión iterativa. Veamos el caso recursivo, que es el más confuso.

- El nodo está afuera del rango: no se llama a ningún hijo.
- El nodo está completamente dentro del rango: no se llama a ningún hijo.
- El nodo está parcialmente dentro del rango: se llama a los dos hijos. Esto puede ocurrir hasta dos veces por paso, ya que el intervalo es contiguo.

Resolución del Problema

Veamos el Código de "Dynamic Range Sum Queries".

<https://cses.fi/problemset/task/1648>

Pregunta Motivadora

La estructura de datos que trabajamos anteriormente, ¿sólo nos permite trabajar con la suma?

Pregunta Motivadora

La estructura de datos que trabajamos anteriormente, ¿sólo nos permite trabajar con la suma?

¡No! Podemos generalizarla para cualquier operación que sea asociativa y tenga un elemento neutro. A este tipo de operaciones las llamamos **monoides**.

Definición de Monoide

Un Monoide es una estructura algebraica con una operación binaria, que es asociativa y tiene elemento neutro. Un Monoide se lo puede denotar como (A, \oplus) y en lenguaje formal se define como:

- $\forall a, b, c \in A, (a \oplus b) \oplus c = a \oplus (b \oplus c)$ (es asociativa)
- $\exists ! e \in A, \forall a \in A, a \oplus e = e \oplus a = a$ (existe neutro)
- $\forall a, b \in A, a \oplus b \in A$ (operación cerrada)

Ejemplos de Monoide

Estos son algunos ejemplos de Monoides, que pueden ser interesantes para resolver problemas de Programación Competitiva:

- $(\mathbb{N}, +)$
- (\mathbb{N}, \times)
- (\mathbb{N}, \min)
- (\mathbb{N}, \max)
- (\mathbb{N}, \gcd)
- (\mathbb{N}, lcm)
- (\mathbb{N}, \wedge)
- (\mathbb{N}, \vee)
- (\mathbb{N}, \oplus) , donde \oplus es el XOR
- $(\text{string}, +)$, donde $+$ es la concatenación

Implementación Generalizada

Ahora, vamos a ver en C++ una implementación generalizada del Segment Tree, que nos permita trabajar con cualquier Monoide. Esta idea es gran cortesía de Tarche, que hizo que esté en nuestro Notebook para ICPC.

Resolviendo Más Problemas

Resolvamos los siguientes problemas de CSES con la implementación de Monoide:

- <https://cses.fi/problemset/task/1649>
- <https://cses.fi/problemset/task/1650>

Hotel Queries

CSES 1143 - Hotel Queries

Hay n hoteles en una calle. Para cada hotel conoces el número de habitaciones libres. Tu tarea es asignar habitaciones de hotel para grupos de turistas. Todos los miembros de un grupo quieren alojarse en el mismo hotel.

Los grupos llegarán a usted uno tras otro y usted sabrá para cada grupo la cantidad de habitaciones que necesita. Siempre asignas un grupo al primer hotel que tenga suficientes habitaciones. Después de esto, el número de habitaciones libres en el hotel disminuye.

Para cada grupo de turistas, imprima el número del hotel al que fueron asignados.

<https://cses.fi/problemset/task/1143>

Recorriendo Alternativamente el Segment Tree

Para resolver el problema, vamos a usar el Segment Tree de máximos como una estructura para hacer Binary Search. La idea es encontrar el primer hotel que tenga suficientes habitaciones para el grupo de turistas en $O(\log n)$.

Partimos del primer nodo (la raíz), y recorremos el de la izquierda si el nodo contiene un número mayor o igual a la cantidad de habitaciones que necesitamos, o el de la derecha si el nodo contiene un número menor.

Luego, actualizamos el Segment Tree para que tenga menos habitaciones libres. ¡Veamos el código!