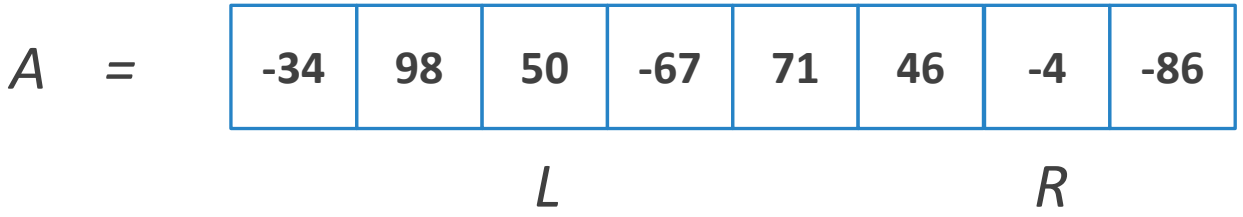
Segment Tree

EDD

Un problema (muy) sencillo

Imagina que tienes un arreglo de números *A* de largo *n*:



Y quieres saber la suma de los elementos entre las posiciones *L* y *R*.

¿Cómo harías esto? ¿Qué complejidad tiene?

Un problema (no tan) sencillo

Imagina que tienes un arreglo de números A de largo n:

A =

-34	98	50	-67	71	46	-4	-86

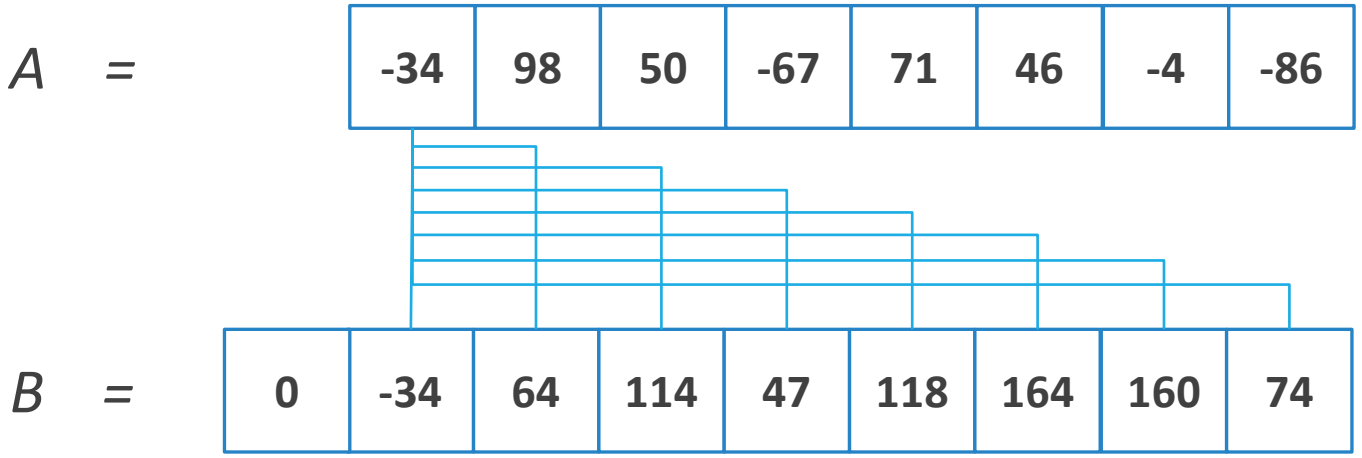
Y quieres saber lo mismo pero en m consultas:

$$(L_1, R_1), (L_2, R_2), \ldots, (L_m, R_m)$$

¿Cómo harías esto? ¿Qué complejidad tiene?

Un problema (no tan) sencillo

Hay una forma de resolver esto que toma tiempo $O(n+m)$.



Usamos un arreglo auxiliar *B* de **sumas acumuladas**.

¿Cuánto tiempo toma generar este arreglo?

Un problema (no tan) sencillo

En el arreglo B, el índice B[i] guarda el valor A[1] + … + A[i-1].

Para la consulta (L, R) podemos responder usando lo siguiente:

$$A[L] + A[L+1] + \ldots + A[R] = A[1] + \ldots + A[R] - (A[1] + \ldots + A[L-1])$$

$$= B[R+1] - B[L]$$

¡Cada consulta toma tiempo O(1)!

Un problema (harto menos) sencillo

Imagina que tienes un arreglo de números *A* de largo *n*:

A =

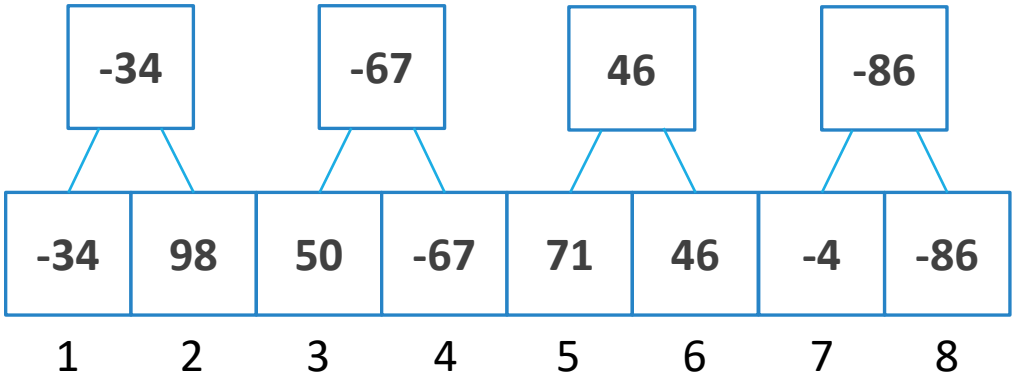
-34	98	50	-67	71	46	-4	-86

Tenemos nuestras *m* consultas (L_1, R_1), (L_2, R_2), … , (L_m, R_m), pero ahora en cada una queremos el **mínimo** de cada rango.

¿Nos sirve la idea anterior? ¿Existe algo mejor que O(n*m)?

Segment Tree – idea

Para empezar, usemos $n/2$ valores auxiliares

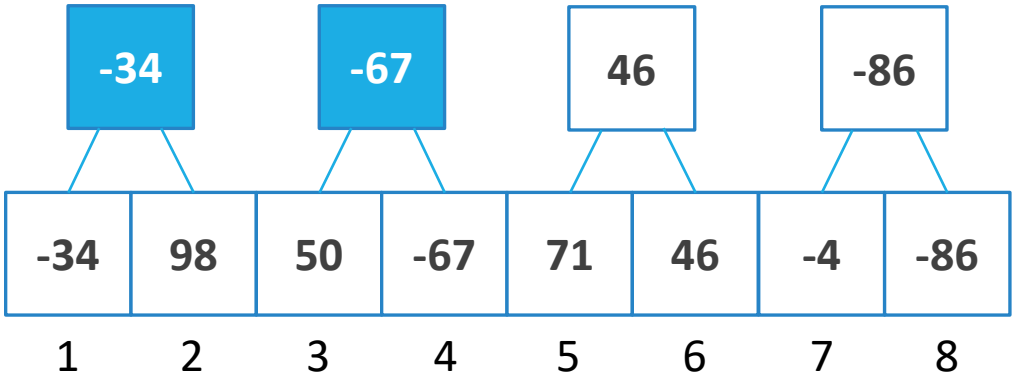


¿Qué información guardan estos valores?

¿Cómo buscarías el mínimo de algún rango aquí?

Segment Tree – idea

Por ejemplo, si el rango es (1,4)

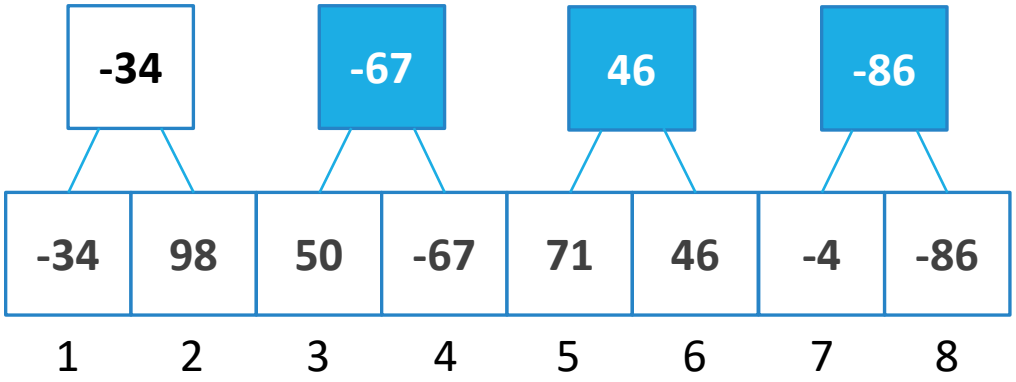


Revisamos estos valores, la respuesta es -67.

¿Por qué nos basta revisar esos dos valores?

Segment Tree – idea

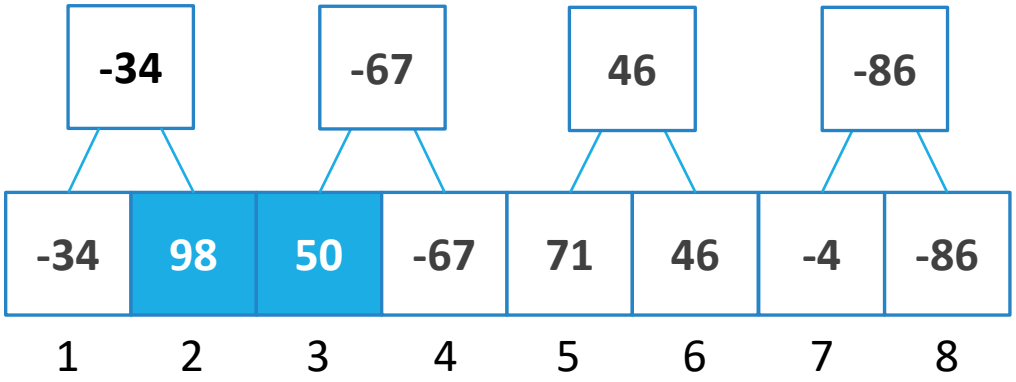
Por ejemplo, si el rango es (3,8)



Revisamos estos valores, la respuesta es -86.

Segment Tree – idea

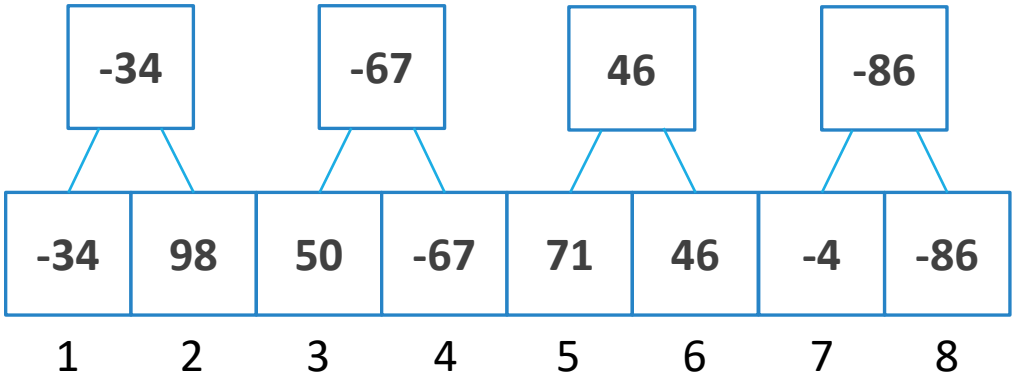
Por ejemplo, si el rango es (2,3)



Revisamos estos valores, la respuesta es 50.

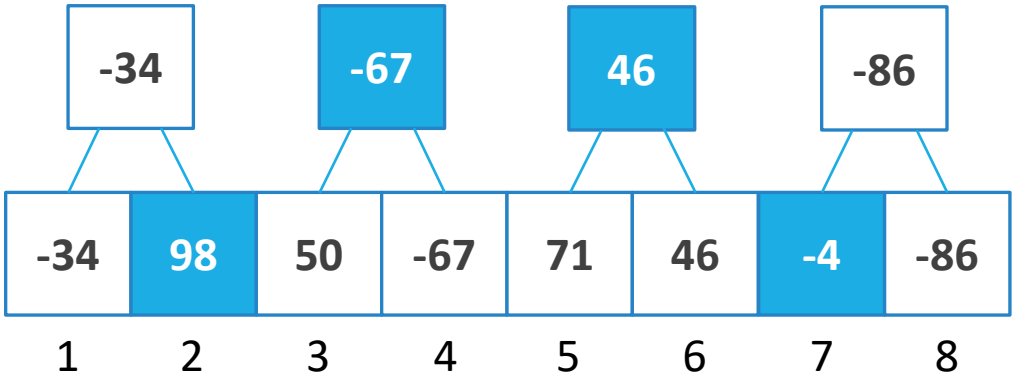
Segment Tree – idea

¿Y si el rango es (2,7)?



Segment Tree – idea

¿Y si el rango es (2,7)?



Revisamos estos.

Segment Tree – idea

Nuestro arreglo tiene tamaño 8, y antes en el peor caso teníamos que revisar 8 valores en una sola consulta.

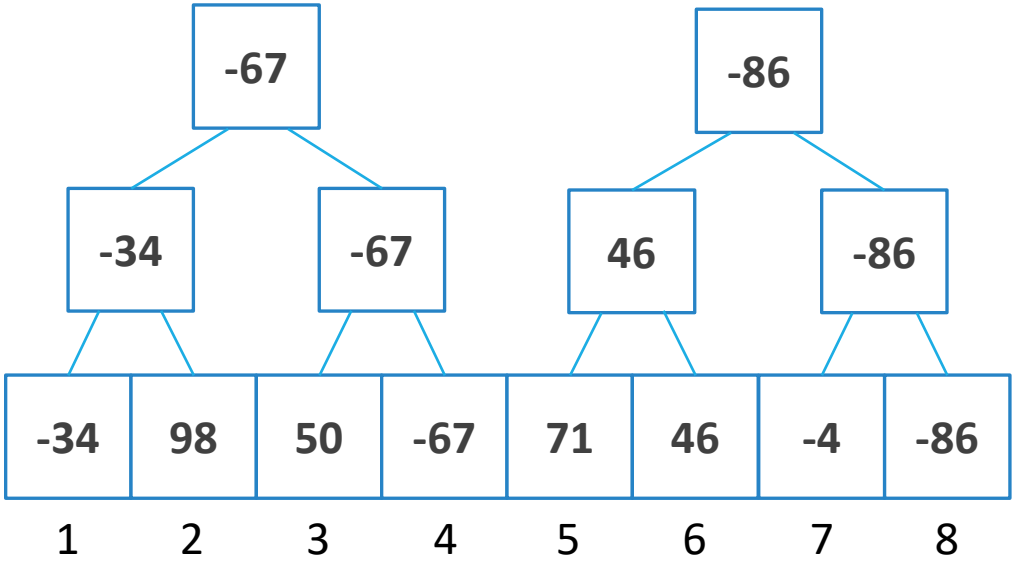
Ahora, en el peor caso vamos a revisar 4 valores.

¿Cómo baja el tiempo de las consultas **grandes**? ¿y las chicas?

¿Podemos mejorar esto?

Segment Tree – idea

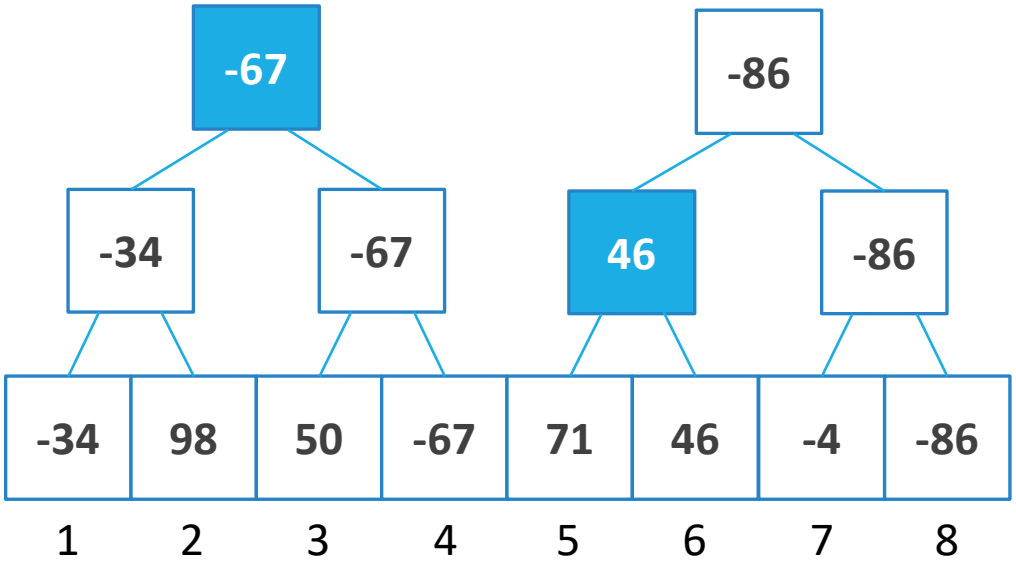
Vamos a armar un nivel más del Segment Tree:



¿Cómo buscarías el mínimo del rango (1,6)? ¿y el (2,8)?

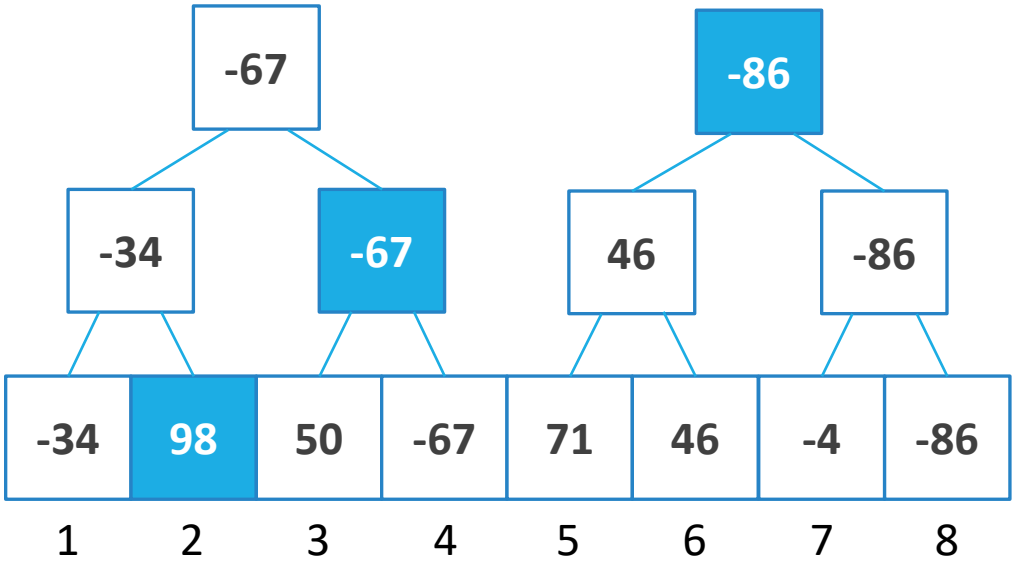
Segment Tree – idea

Para el rango (1,6)



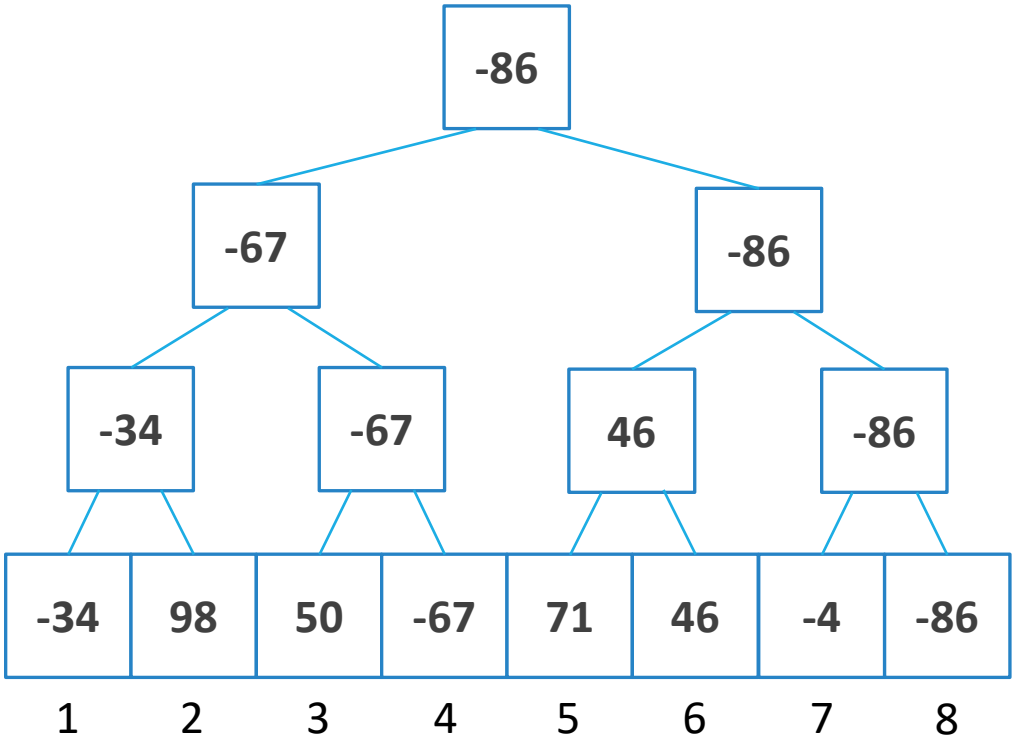
Segment Tree – idea

Y el rango (2,8)



Segment Tree

El Segment Tree completo se ve así:



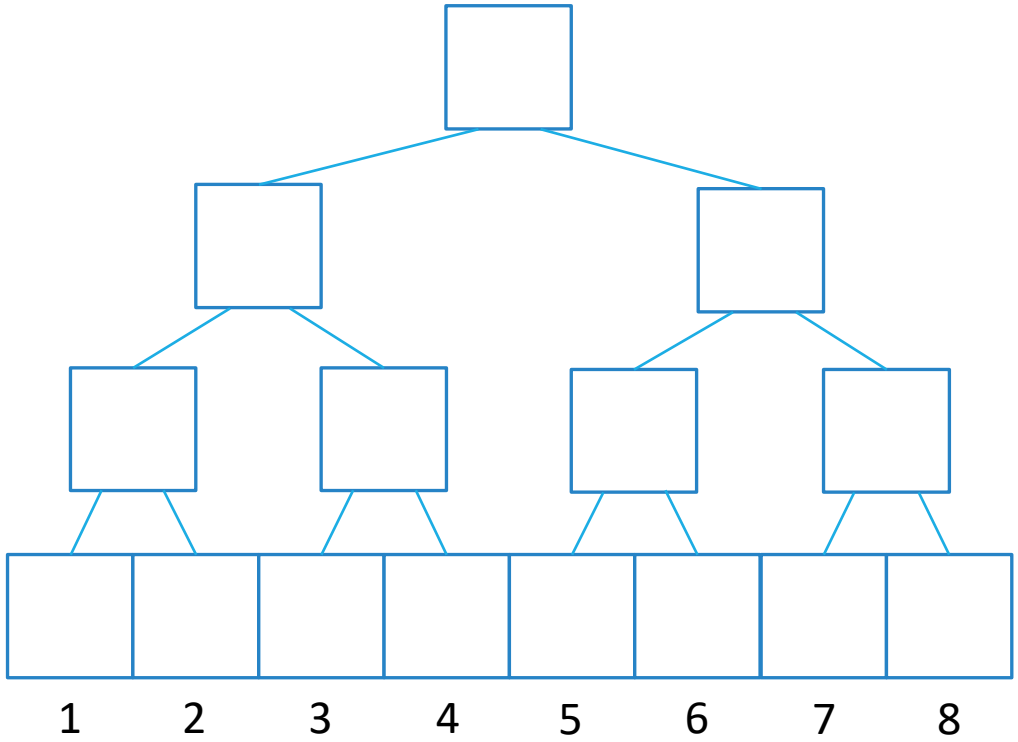
Se puede ver como un *torneo* entre todos los valores.

Segment Tree – Análisis

Asumamos por ahora que $n = 2^k$.

¿Cuántos niveles tiene el Segment Tree?

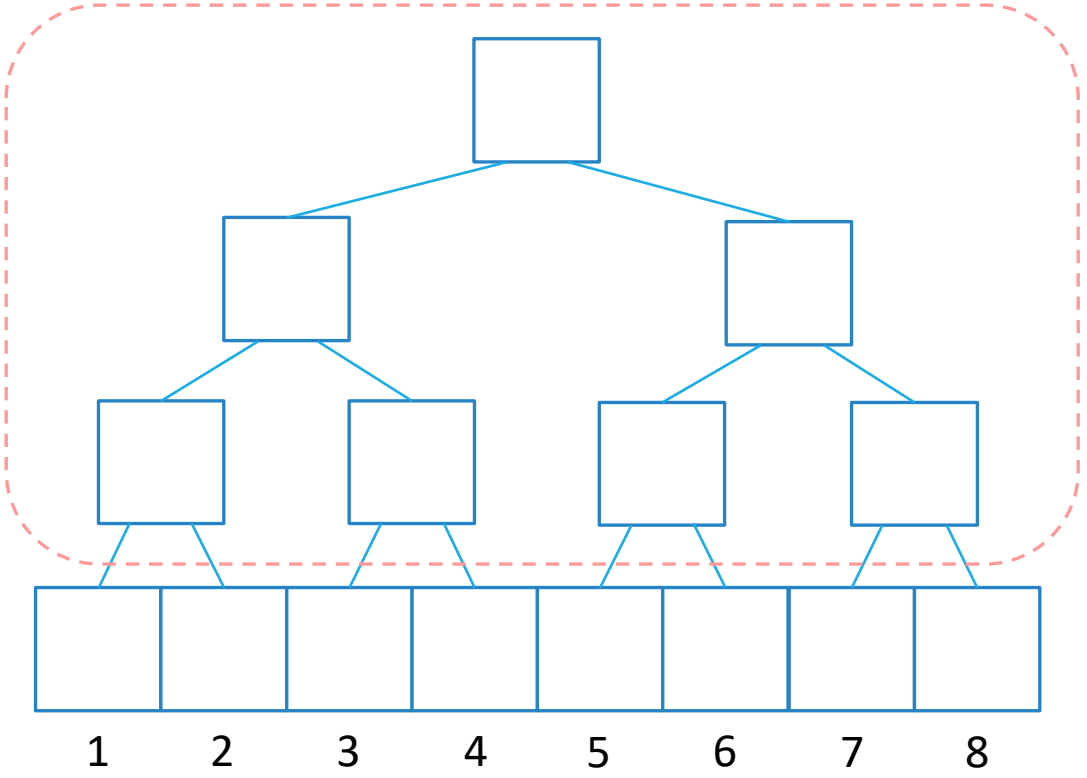
¿Por qué?



Segment Tree – Análisis

Asumiendo que $n = 2^k$.

¿Cuántos valores auxiliares necesito?



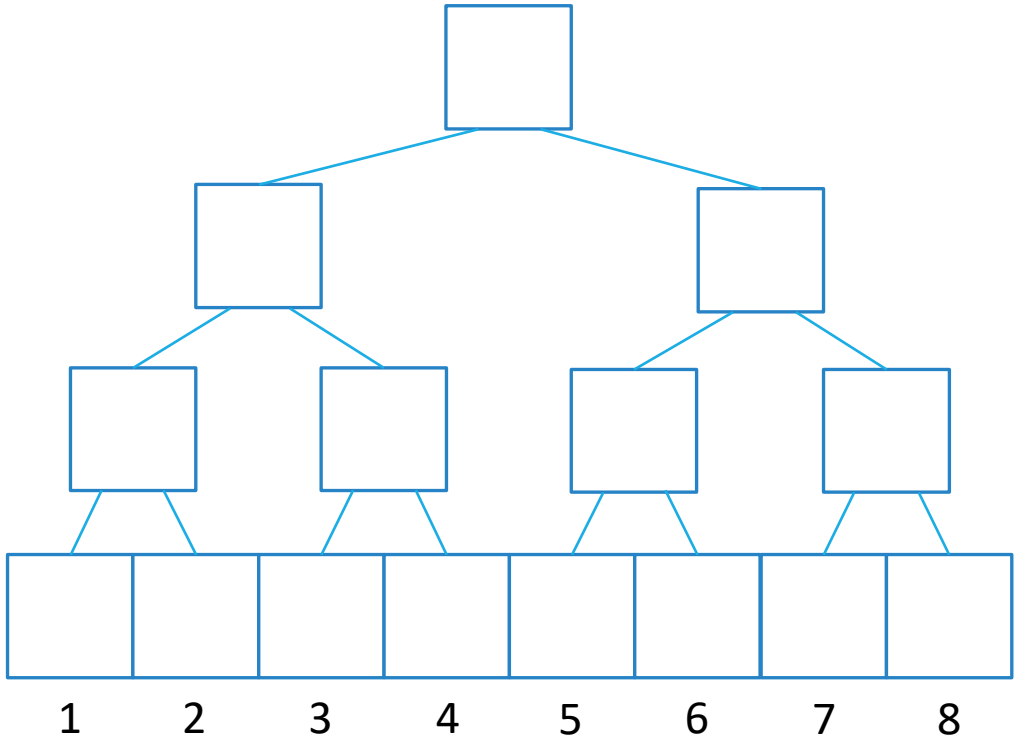
Segment Tree – Análisis

¿Cuántos valores necesito revisar en el peor caso?

Primero vamos a ver cómo es que se ve este peor caso.

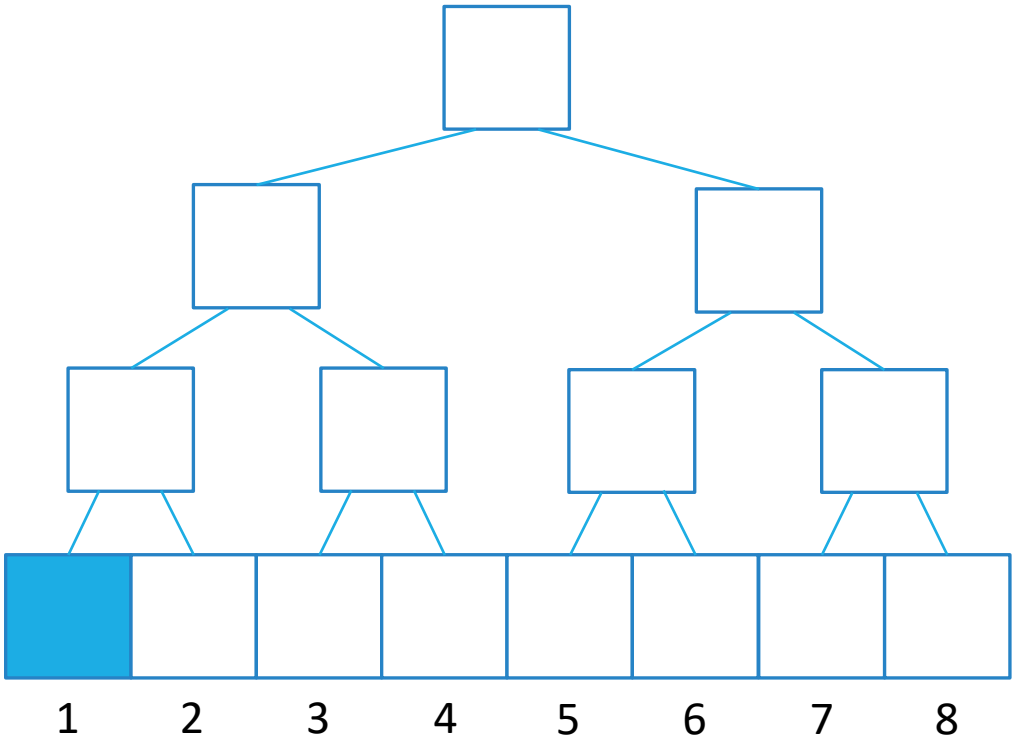
Segment Tree – Análisis

Por ahora consideremos sólo rangos que comienzan en 1, y busquemos el peor caso entre estos.



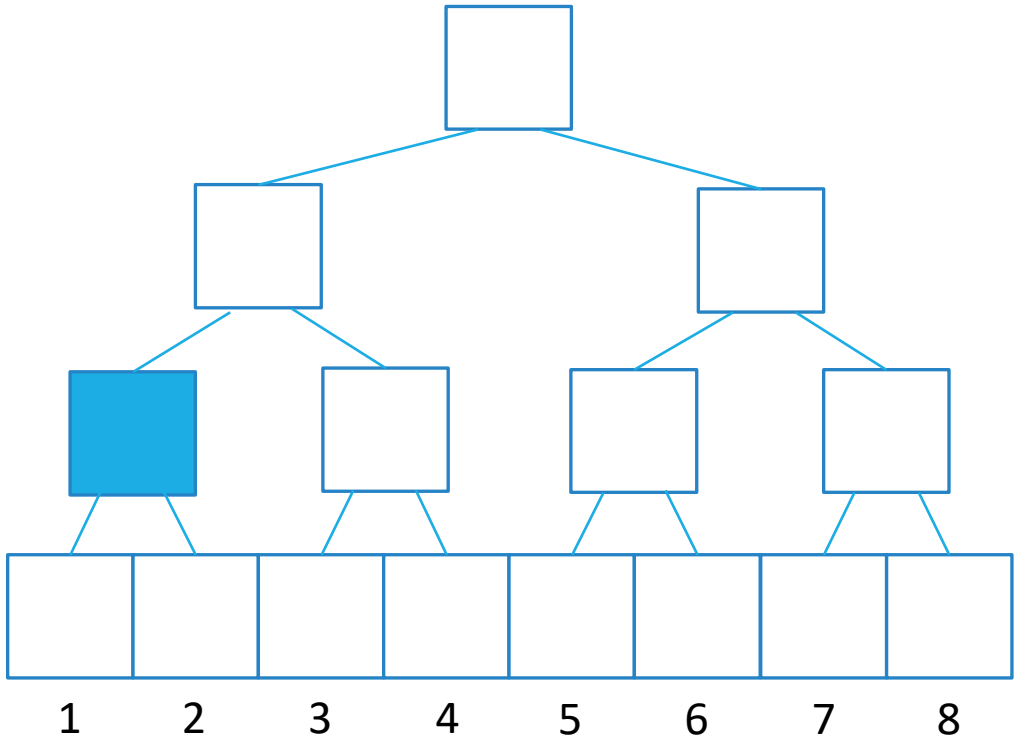
Segment Tree – Peor caso

Para el rango (1,1) revisamos 1 valor:



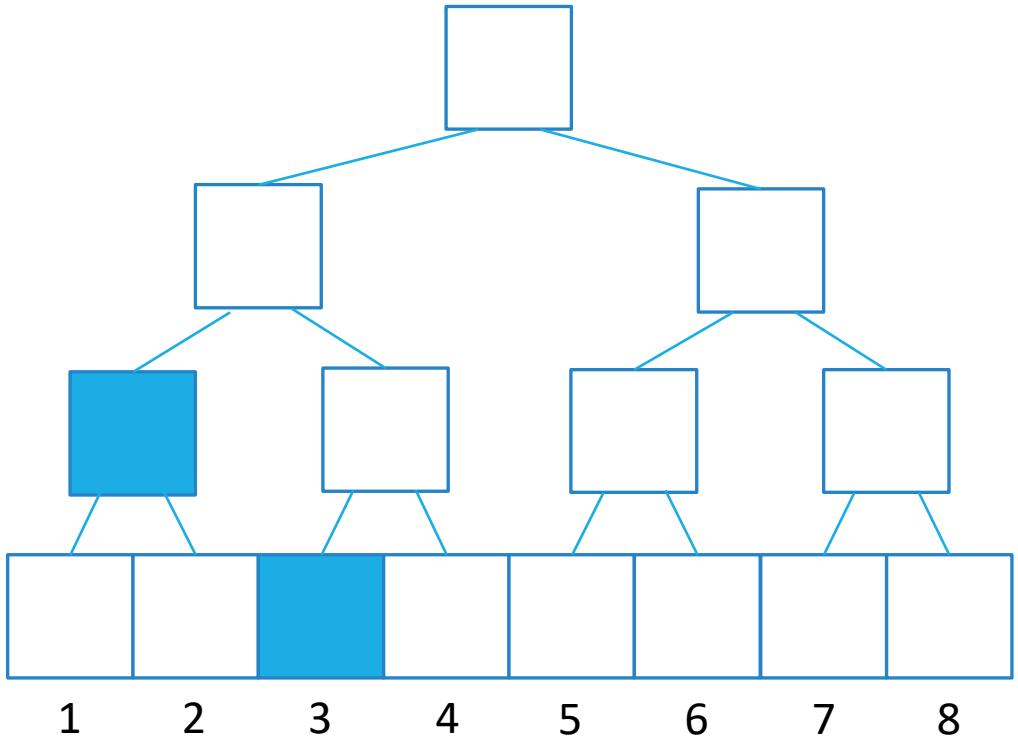
Segment Tree – Peor caso

Para el rango (1,2) revisamos 1 valor:



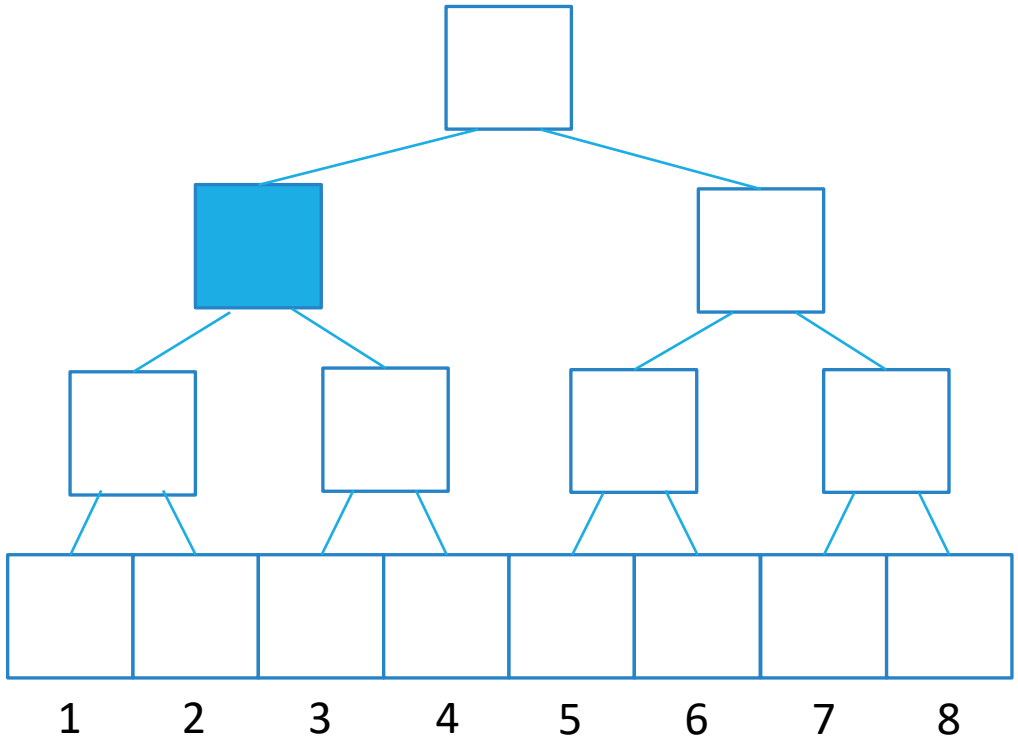
Segment Tree – Peor caso

Para el rango (1,3) revisamos 2 valores:



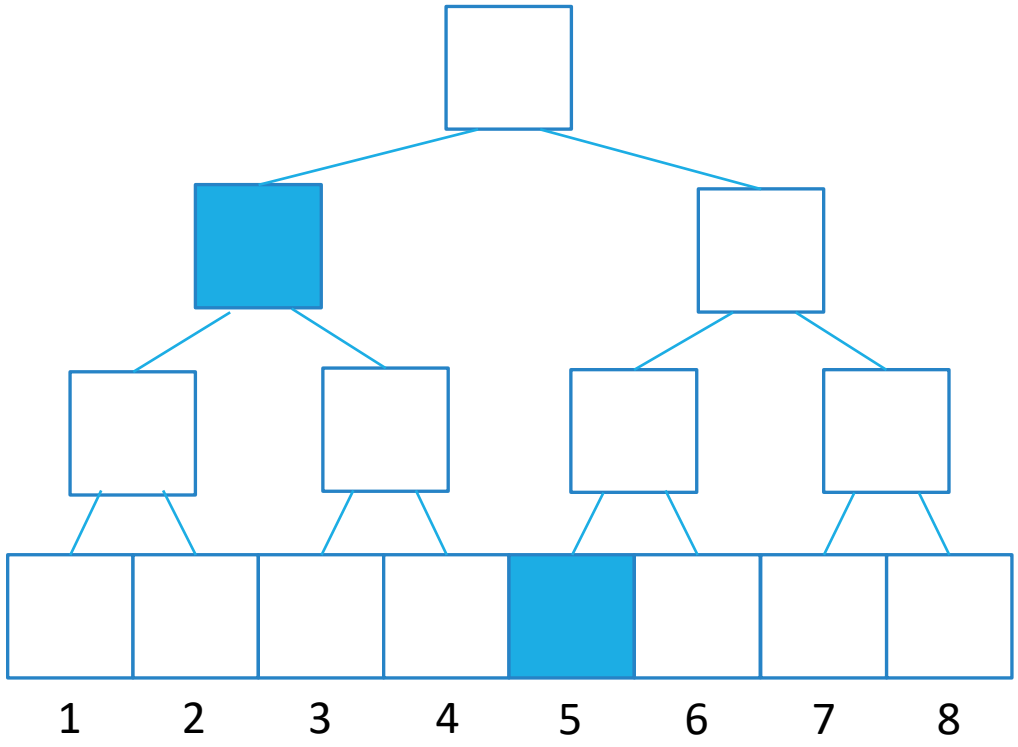
Segment Tree – Peor caso

Para el rango (1,4) revisamos 1 valor:



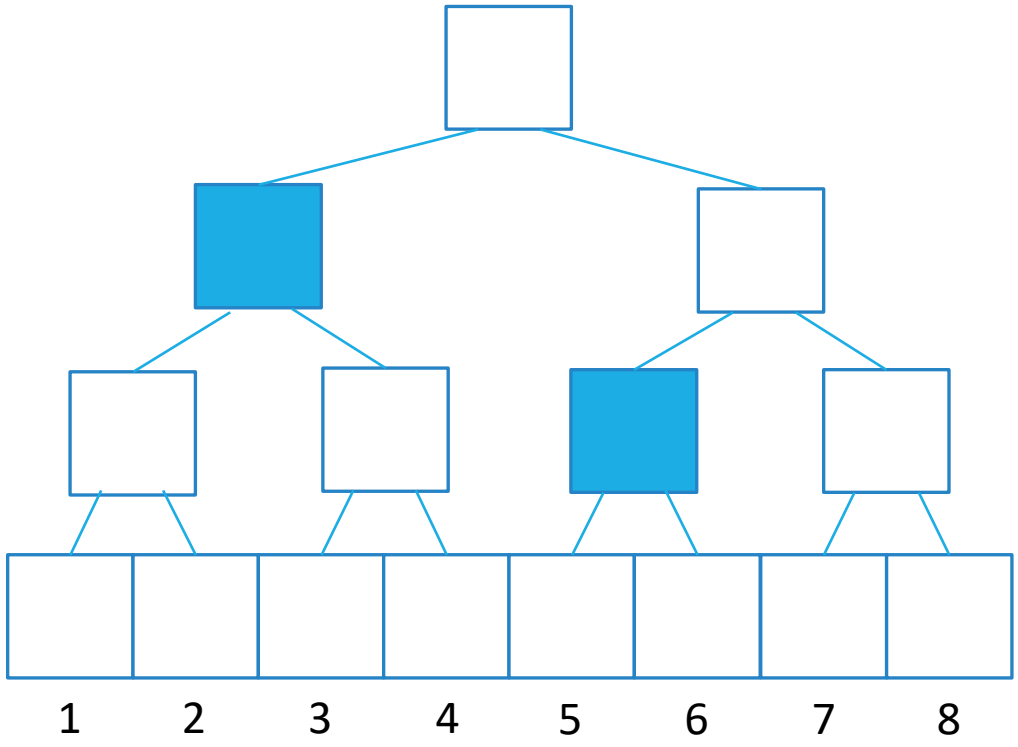
Segment Tree – Peor caso

Para el rango (1,5) revisamos 2 valores:



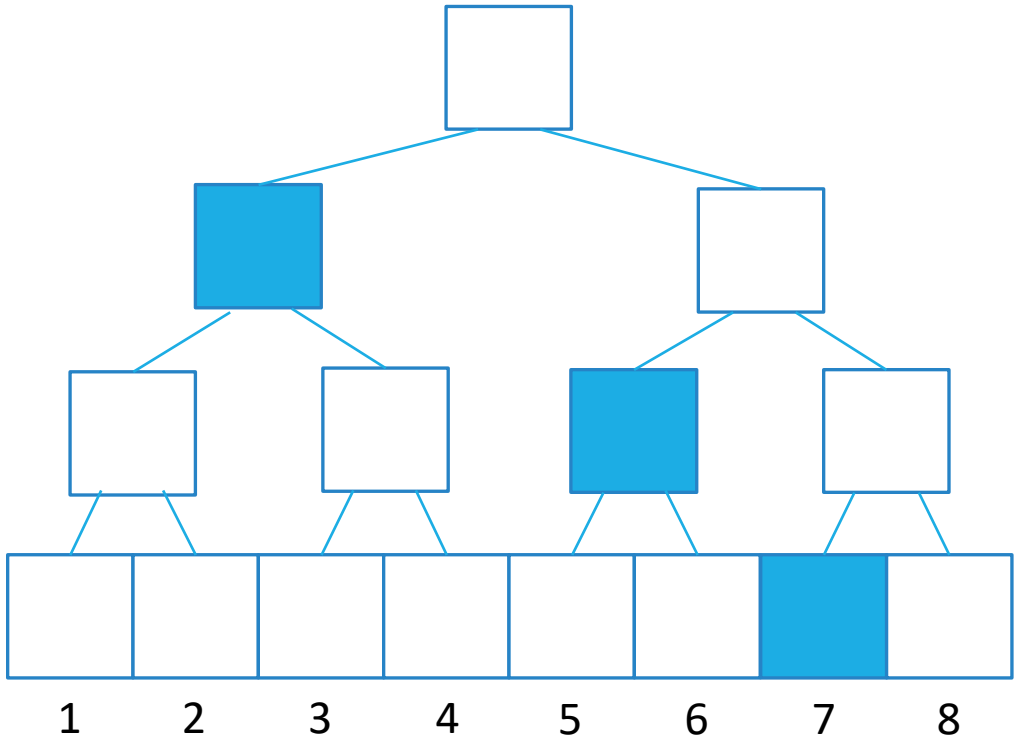
Segment Tree – Peor caso

Para el rango (1,6) revisamos 2 valores:



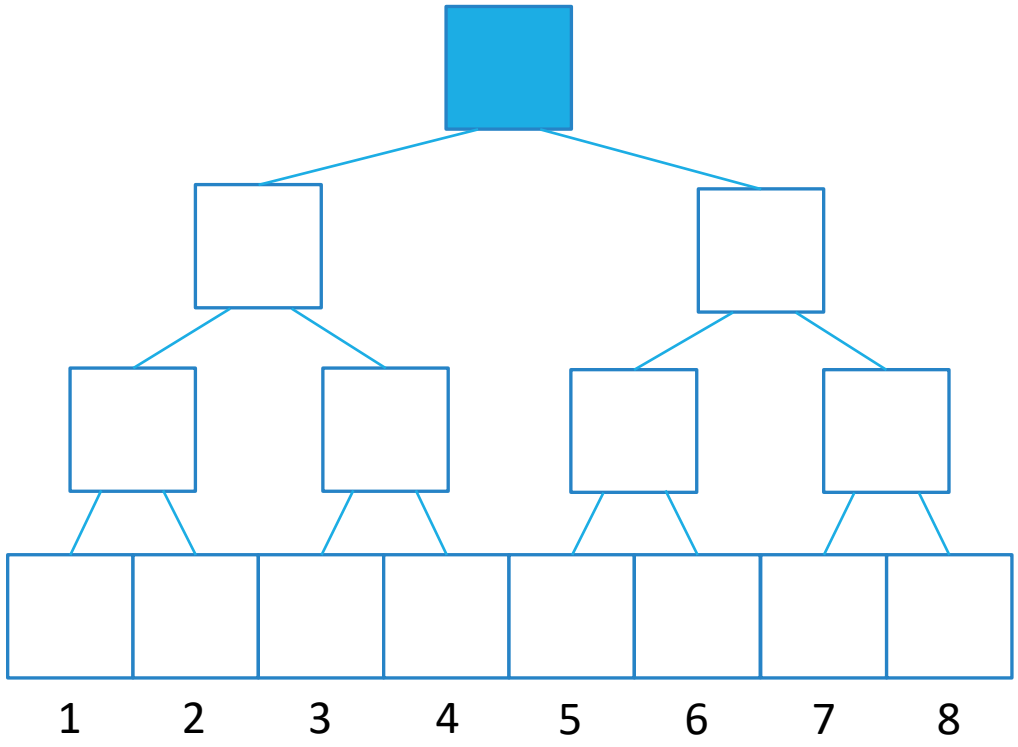
Segment Tree – Peor caso

Para el rango (1,7) revisamos 3 valores:



Segment Tree – Peor caso

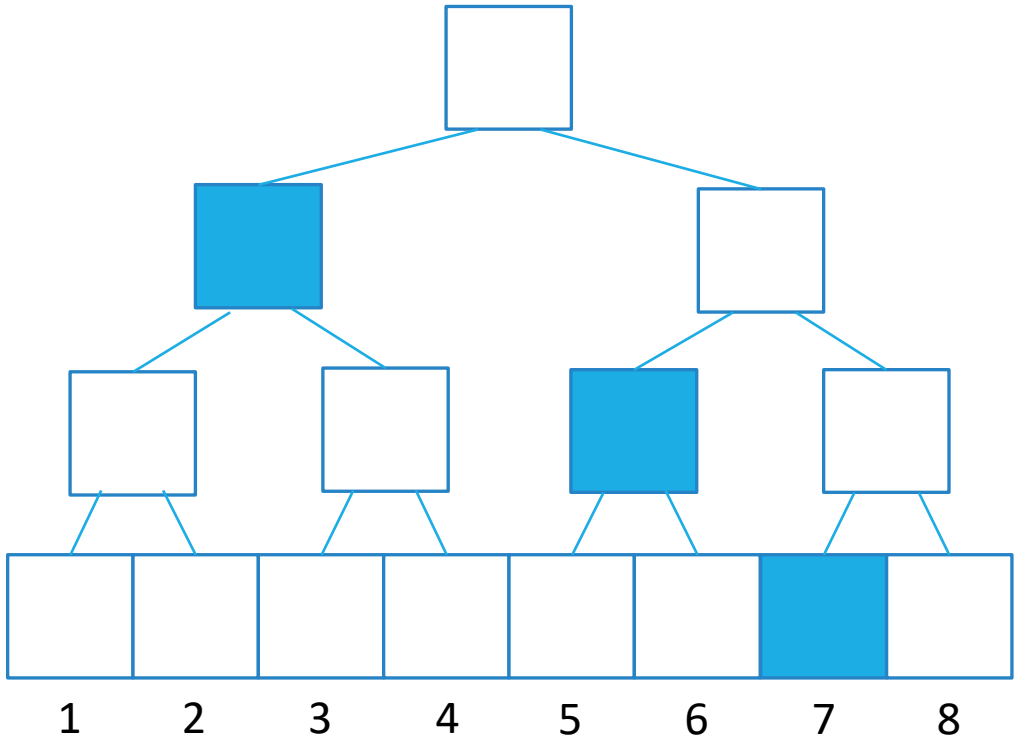
Para el rango (1,8) revisamos 1 valor:



Segment Tree – Peor caso

Y el ganador es… ¡el rango (1,7)!

En ser el peor rango de todos.



Segment Tree – Análisis

Este análisis esconde algo interesante.

¿Qué pueden decir de esta secuencia?

(1,*n*)	Revisiones
1	1
2	1
3	2
4	1
5	2
6	2
7	3
8	1

Segment Tree – Análisis

(1,*n*)	Revisiones	*n* en binario
1	1	1
2	1	10
3	2	11
4	1	100
5	2	101
6	2	110
7	3	111
8	1	1000

Segment Tree – Análisis

(1,*n*)	Revisiones	*n* en binario
1	1	1
2	1	10
3	2	11
4	1	100
5	2	101
6	2	110
7	3	111
8	1	1000

¡Es exactamente la cantidad de 1s!

¿Por qué pasa esto? ¿Cuál es la complejidad asint. de este valor?

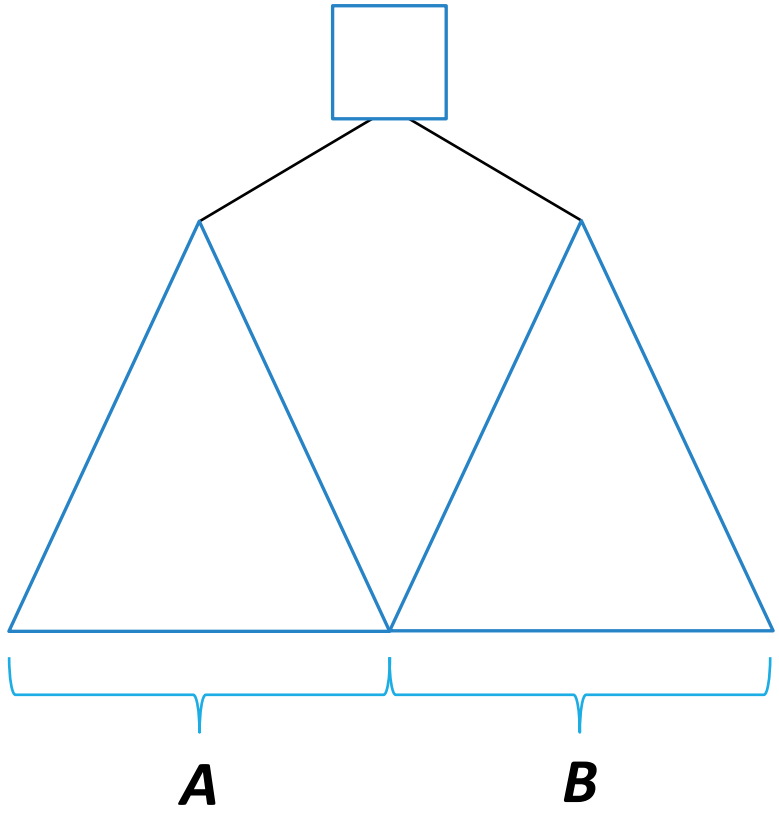
Segment Tree – Peor caso

Nos falta un paso para armar nuestro peor caso.

Llamemos *A* al rango (1,*n*/2)

Y *B* al rango (*n*/2+1, *n*).

¿Dónde van a estar el *L* y *R* del peor caso?

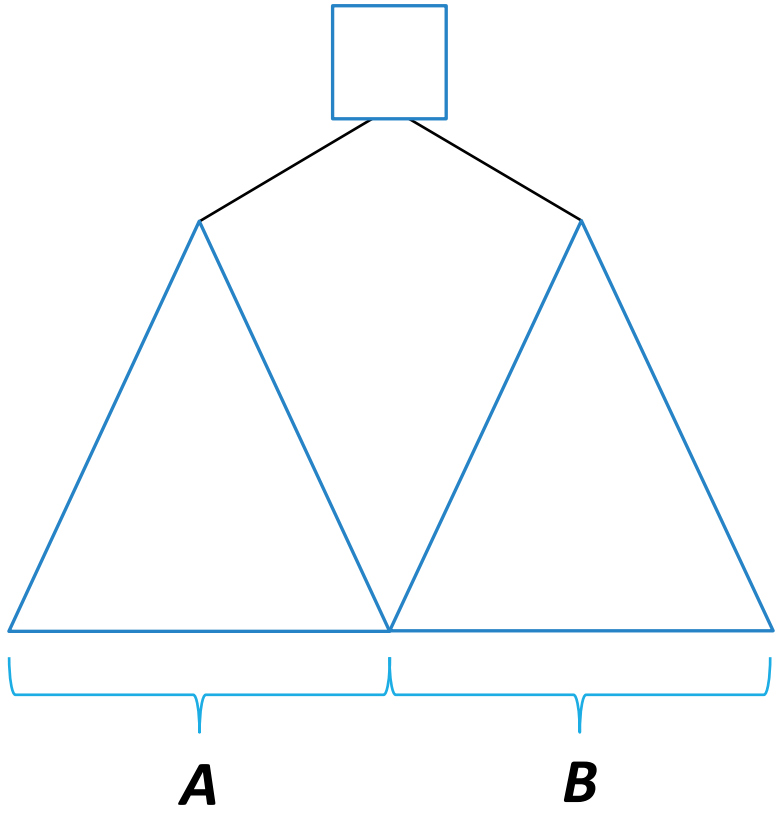


Segment Tree – Peor caso

Si *L* y *R* están en *A*, podemos encontrar un caso peor, ¿por qué?

Y lo mismo si están los dos en *B*.

¿Dónde están *L* y *R* entonces?

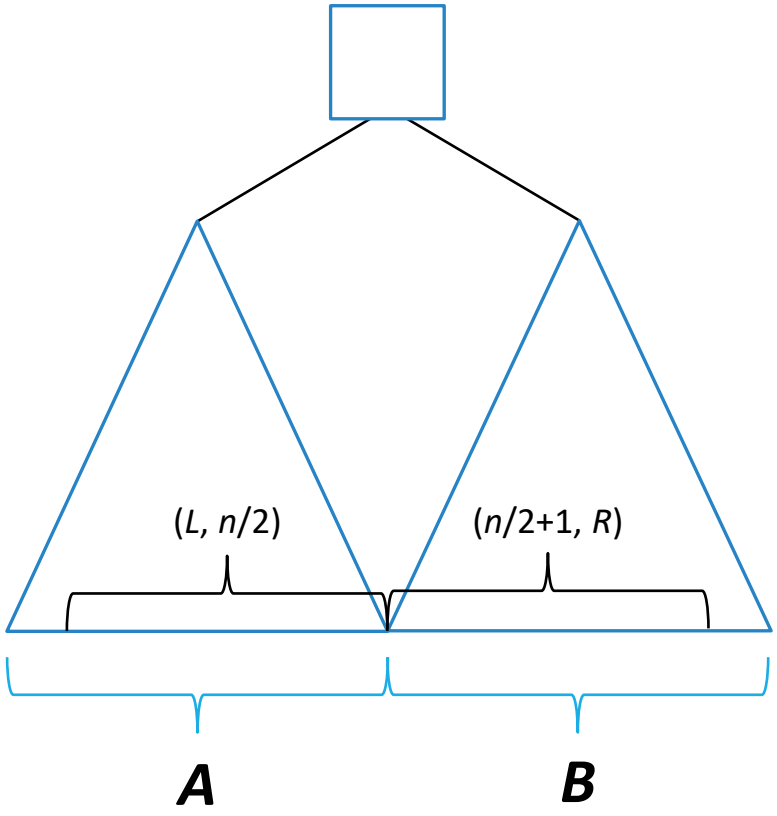


Segment Tree – Peor caso

Nuestro peor caso va a ser un *L* en *A* y un *R* en *B*.

El rango (*L*, *R*) es igual a la unión de (*L*, *n*/2) con (*n*/2+1, *R*).

¡Estos rangos son independientes!



Segment Tree – Peor caso

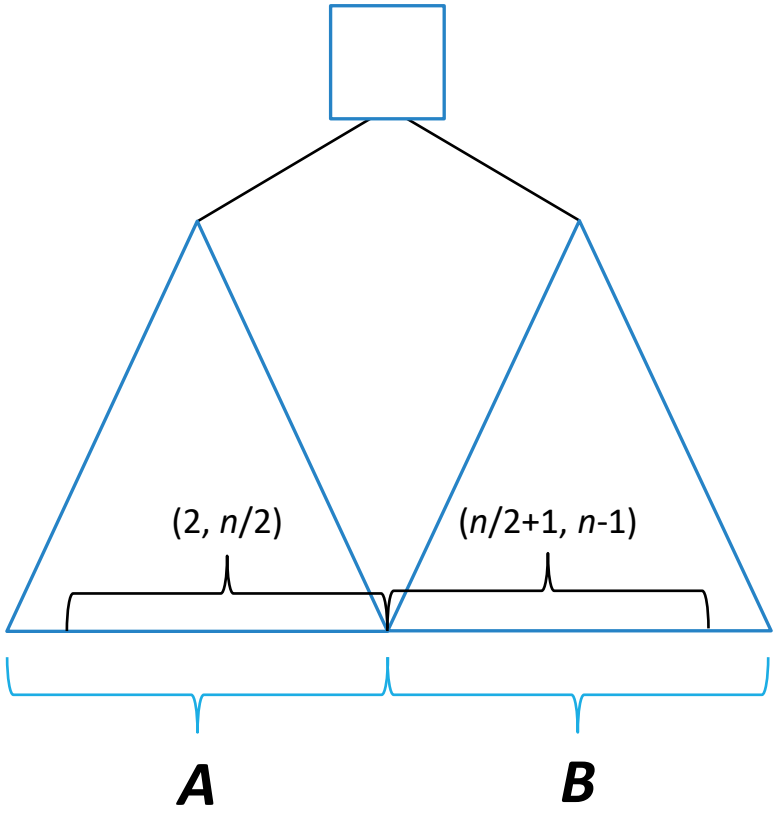
Juntamos el peor rango en B

$(n/2+1, R) = (n/2+1, n-1)$

Con el peor rango en A

$(L, n/2) = (2, n/2)$

¿Por qué estos son los peores rangos?



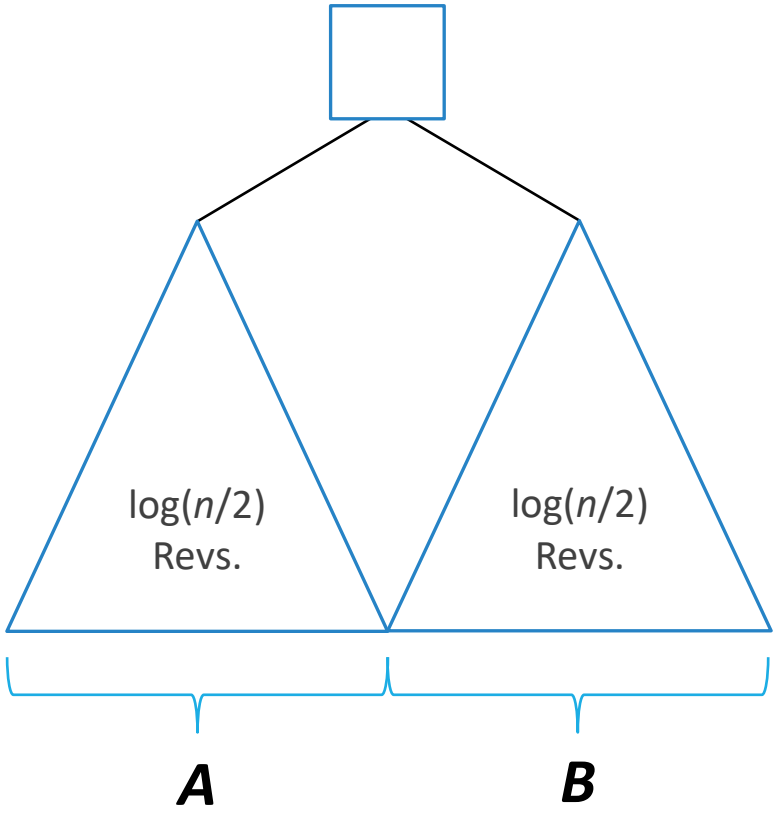
Segment Tree – Peor caso

Finalmente, nuestro peor rango es:

$$(L, R) = (2, n\text{-}2)$$

En cada mitad se hacen **log(*n*/2)** revisiones.

El peor caso son **2*log(*n*/2)** revisiones.

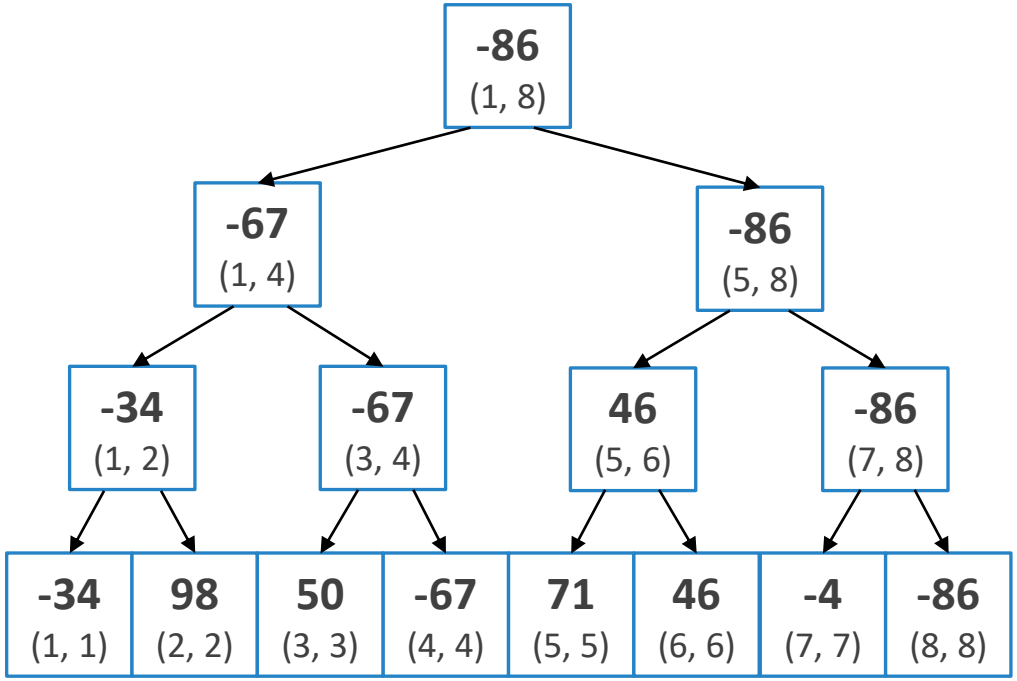


Segment Tree – Implementación

El Segment Tree lo implementamos como un **árbol binario**
(pero no de búsqueda)

Cada nodo tiene estos atributos:

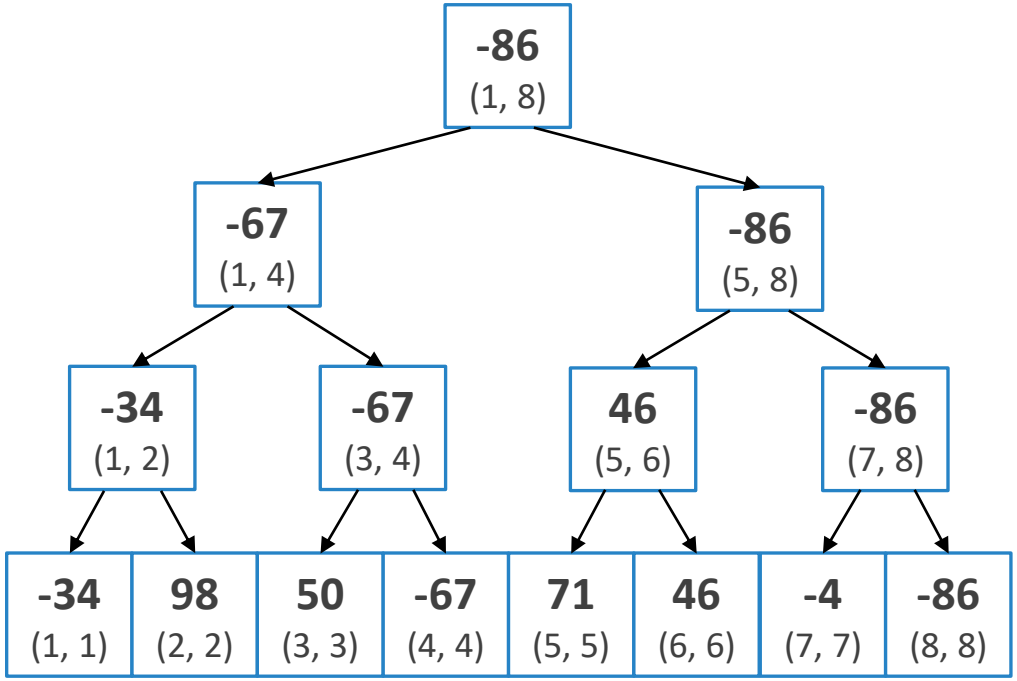
- `leftchild`
- `rightchild`
- `range (l, r)`
- `value`



Segment Tree – Búsqueda

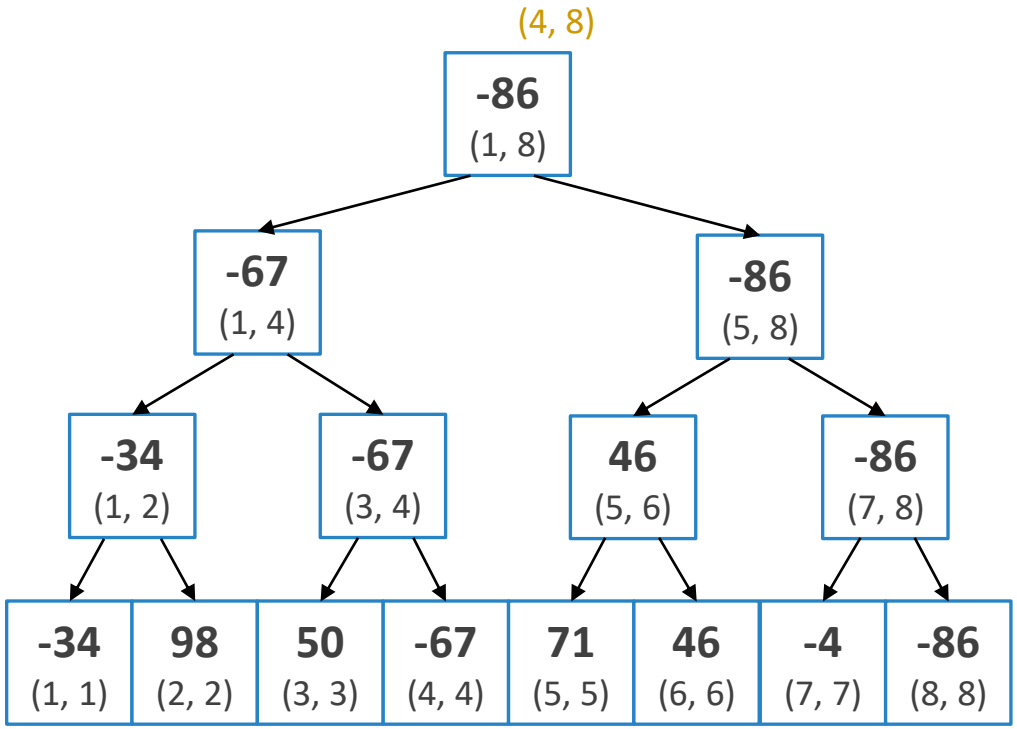
Veremos ahora cómo es que el Segment Tree responde a una consulta.

(O al menos una forma de implmentarlo)



Segment Tree – Búsqueda

Veamos qué hacemos con el rango (4,8).

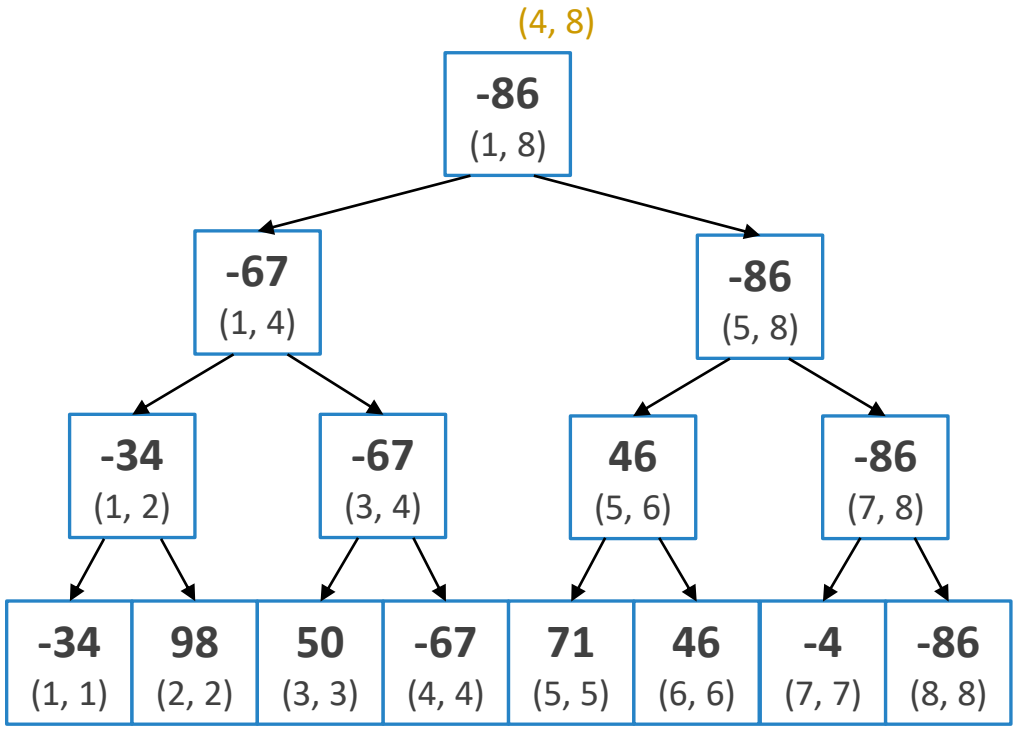


Segment Tree – Búsqueda

Lo primero que ve el primer nodo es que su valor guardado no le sirve para responder la consulta.

(¿por qué?)

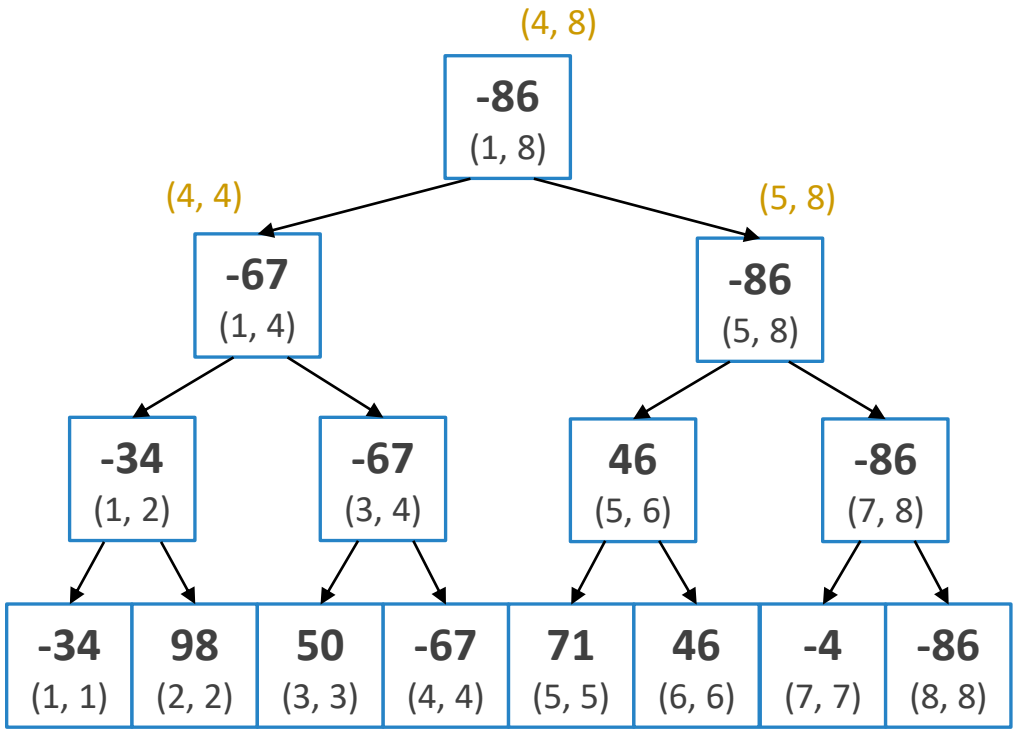
Así que le toca preguntarle a sus hijos.



Segment Tree – Búsqueda

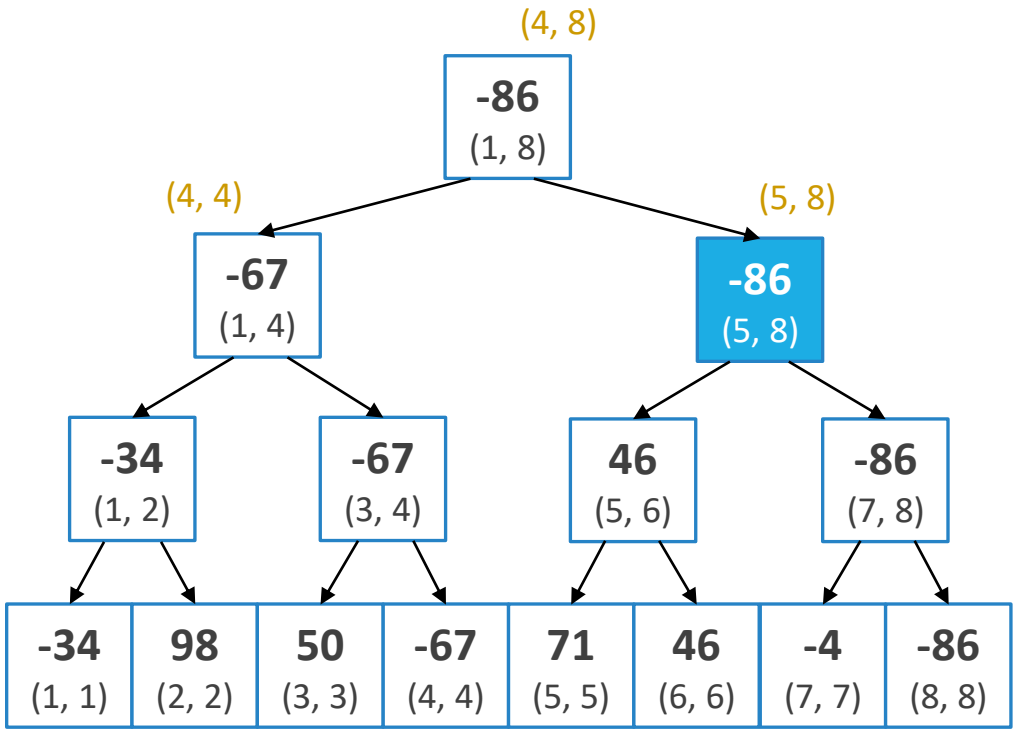
El hijo izquierdo no maneja ningún valor arriba de 4, así que le pregunta por el rango (4,4).

El hijo derecho no maneja ningún valor debajo de 5, así que le pregunta por el rango (5,8).



Segment Tree – Búsqueda

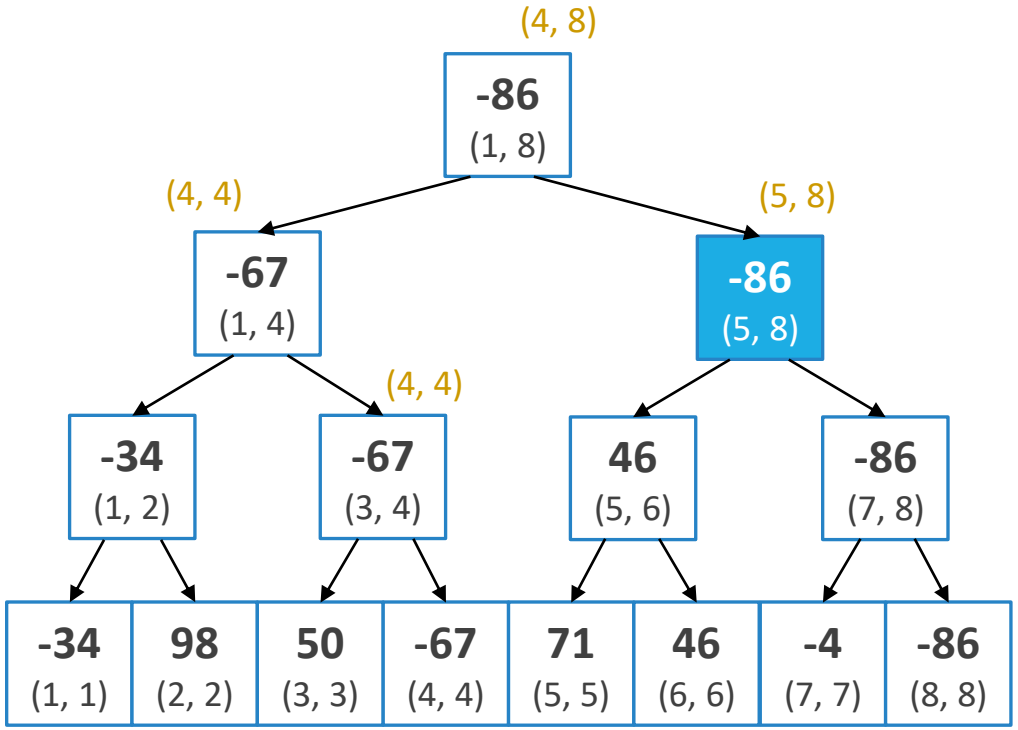
El hijo derecho puede responder a la consulta de inmediato.



Segment Tree – Búsqueda

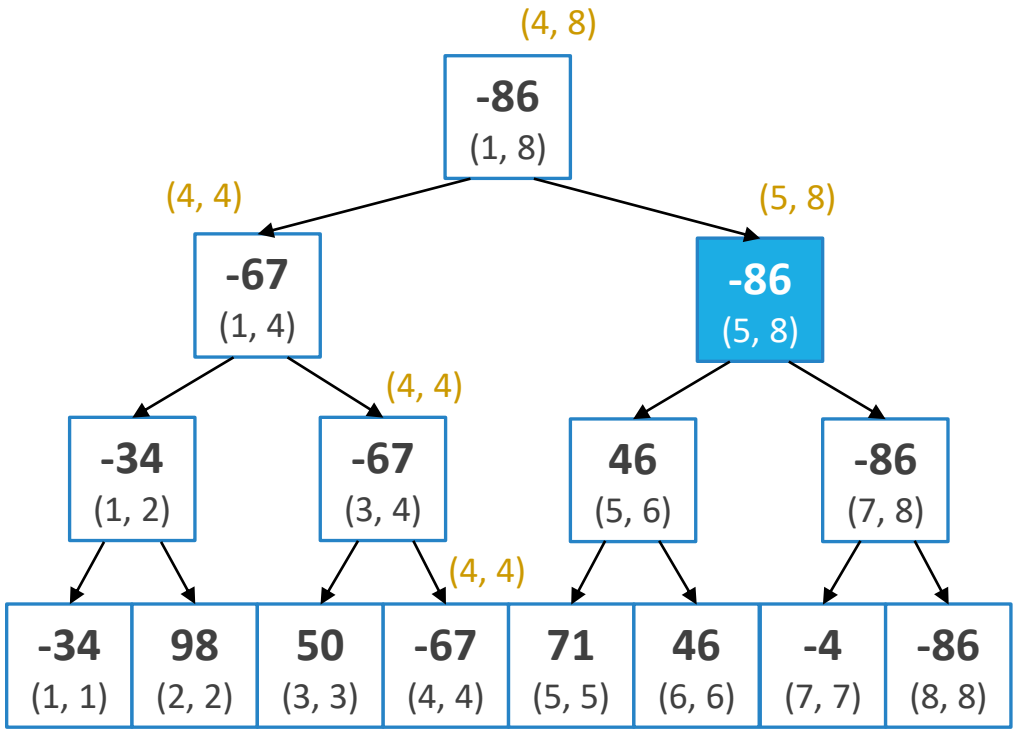
En cambio el hijo izquierdo no, así que le toca preguntarle a sus hijos.

Su hijo izquierdo no maneja ninguno de los valores que necesita, así que solo le pregunta al derecho.



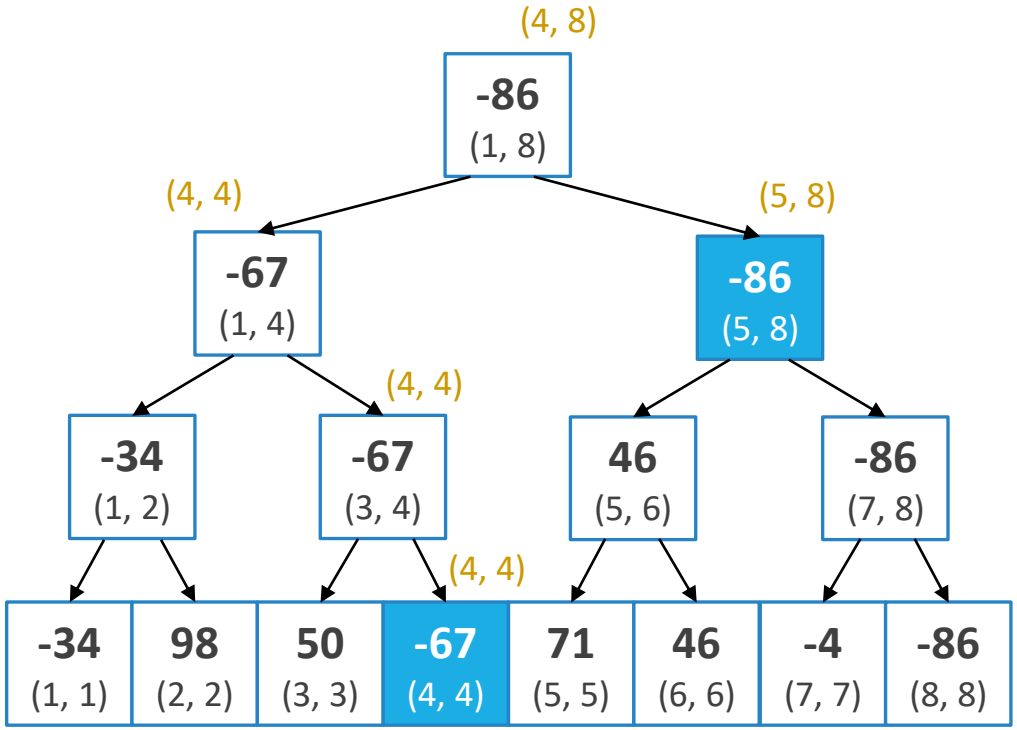
Segment Tree – Búsqueda

Este hijo, a su vez, hace lo mismo.



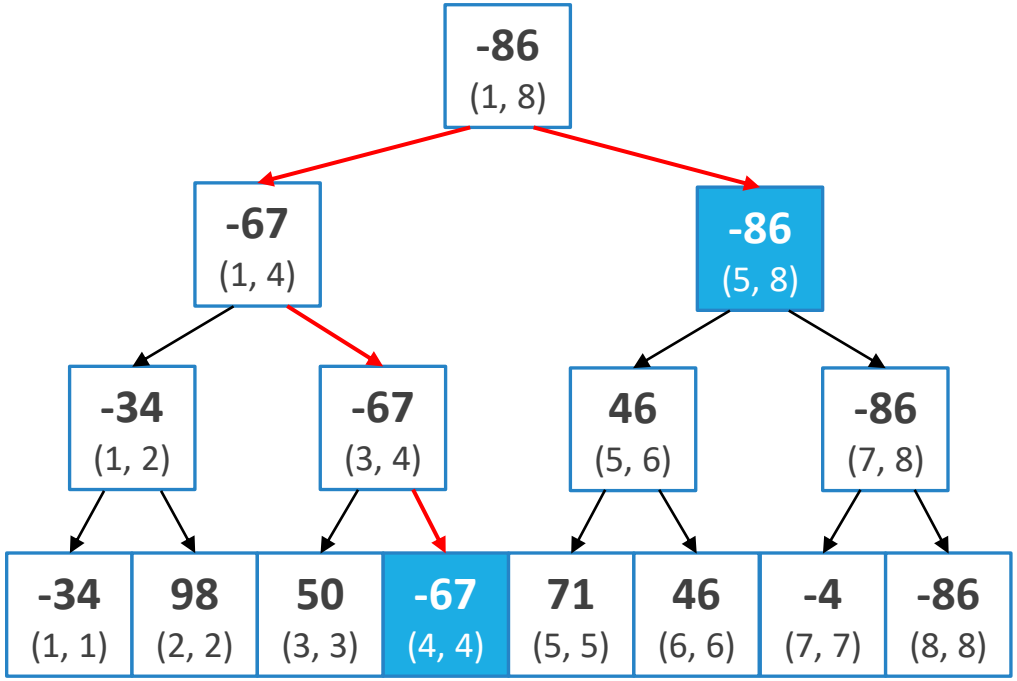
Segment Tree – Búsqueda

El nodo de la posición 4 puede responder la pregunta.



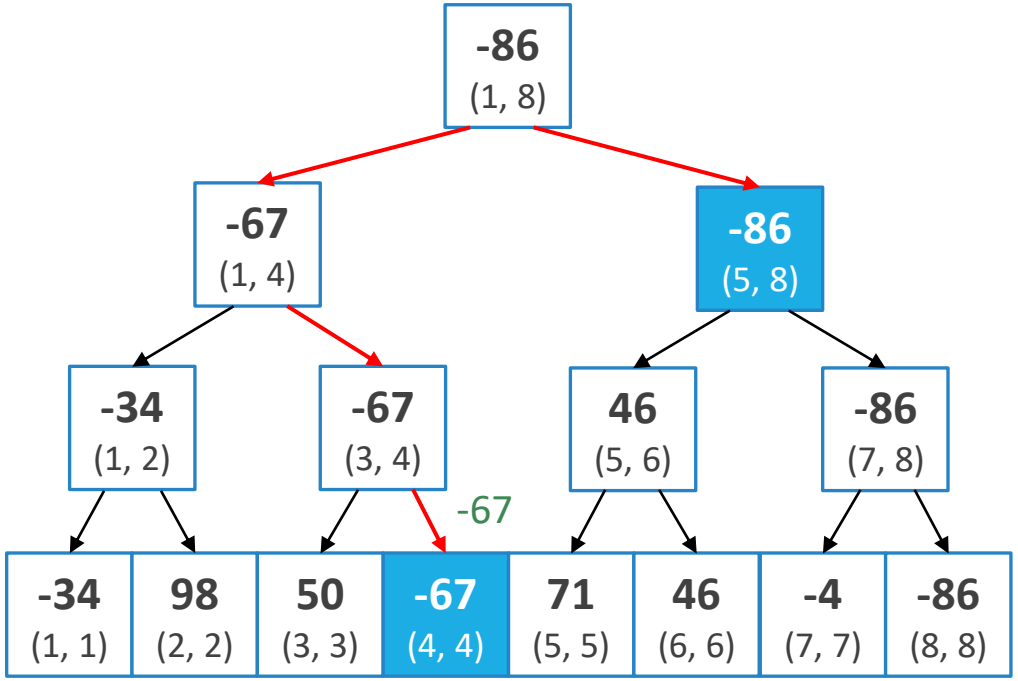
Segment Tree – Búsqueda

Ahora nos toca comparar todos estos valores.



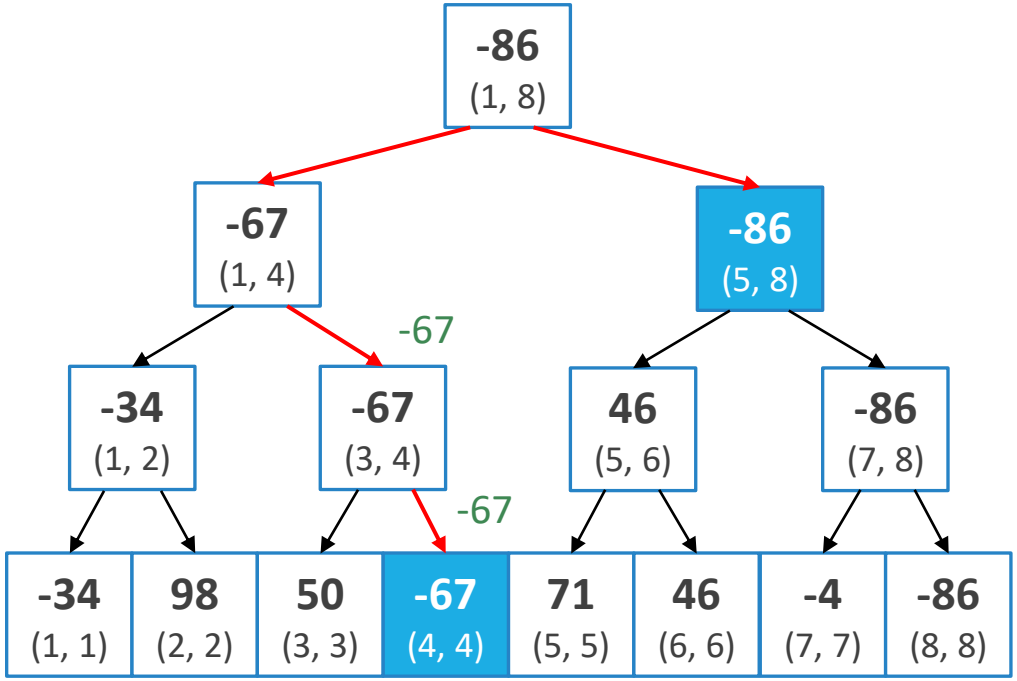
Segment Tree – Búsqueda

El nodo (4,4) retorna -67.



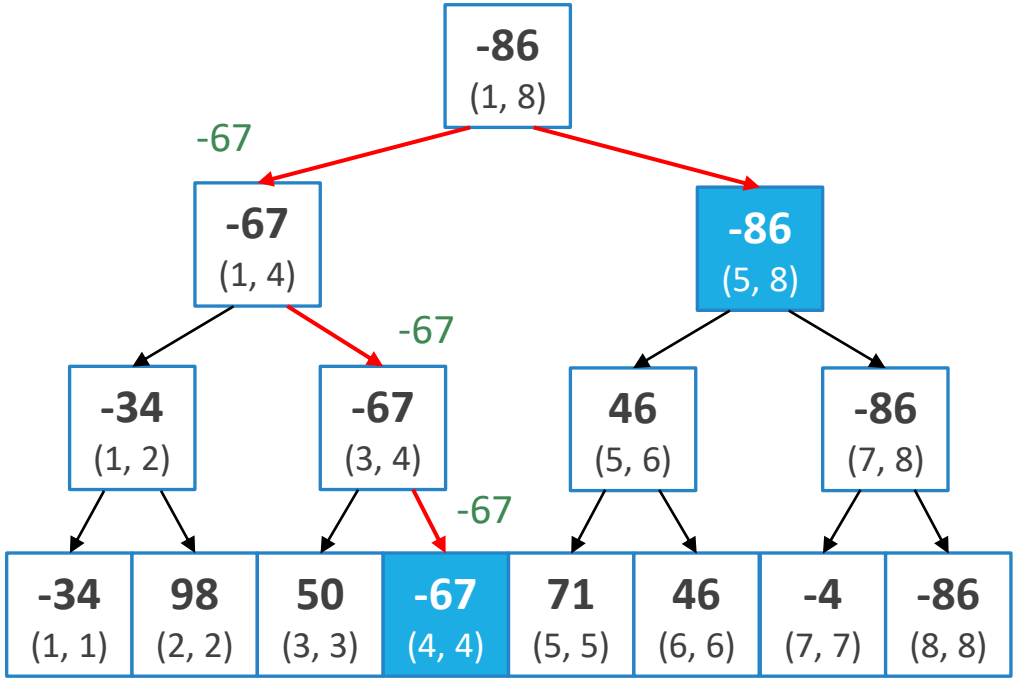
Segment Tree – Búsqueda

El nodo (3,4) recibe **nada** por la izquierda y -67 por la derecha, así que retorna -67.



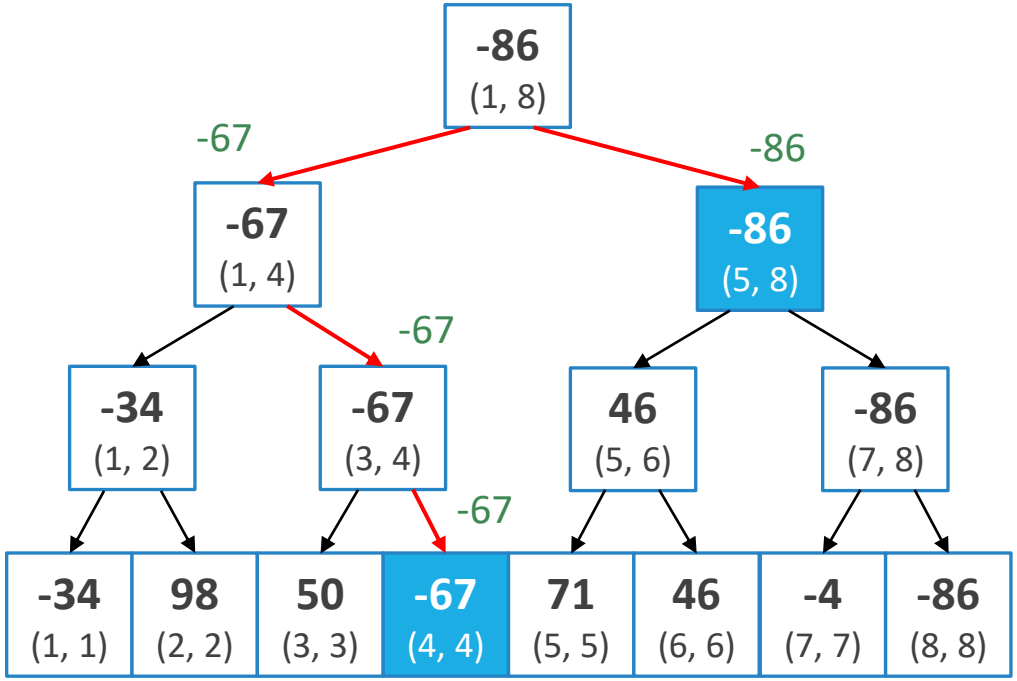
Segment Tree – Búsqueda

El nodo (1,4) recibe **nada** por la izquierda y -67 por la derecha, así que retorna -67.



Segment Tree – Búsqueda

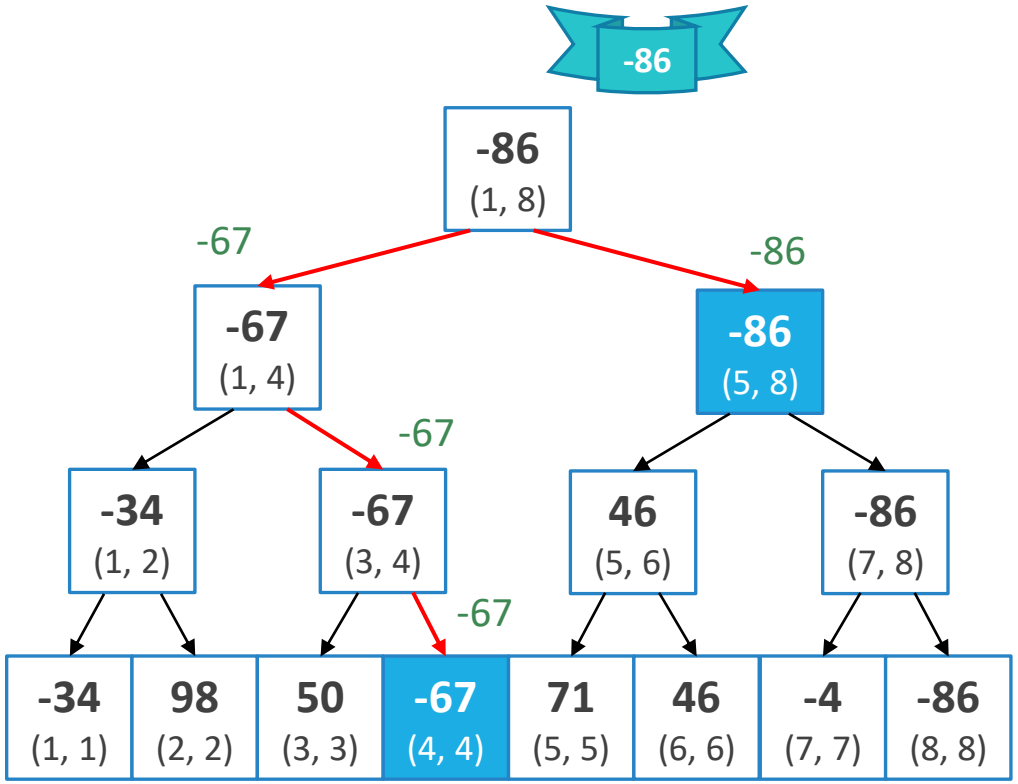
El nodo (5,8) retorna -86.



Segment Tree – Búsqueda

Finalmente, el nodo (1,8) recibe -67 por la izquierda, y -86 por la derecha.

El resultado es -86.

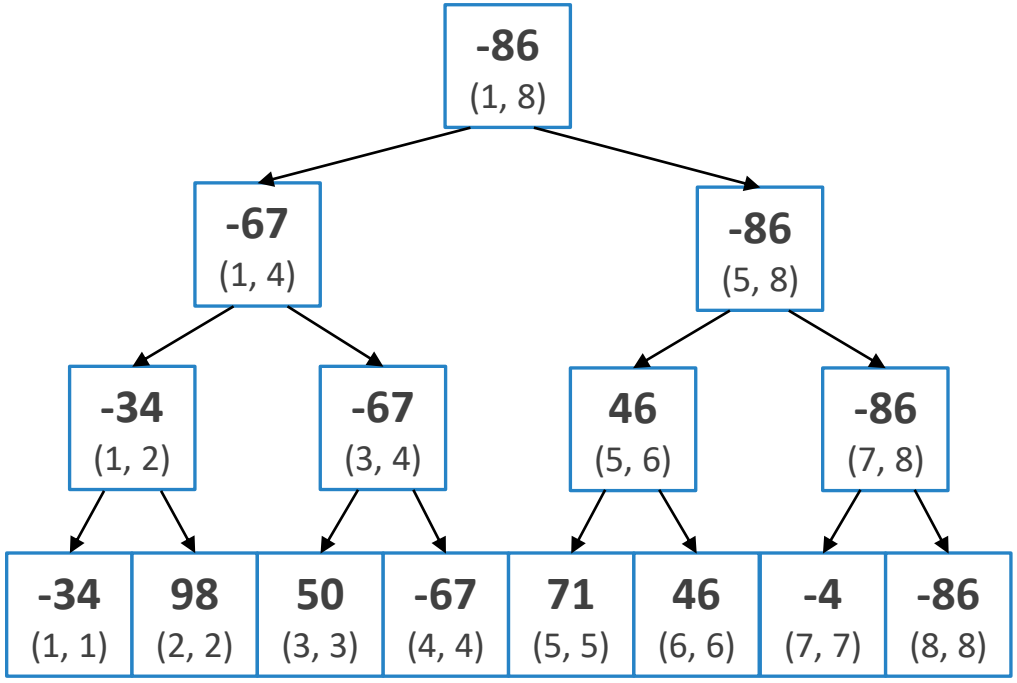


Segment Tree – Búsqueda

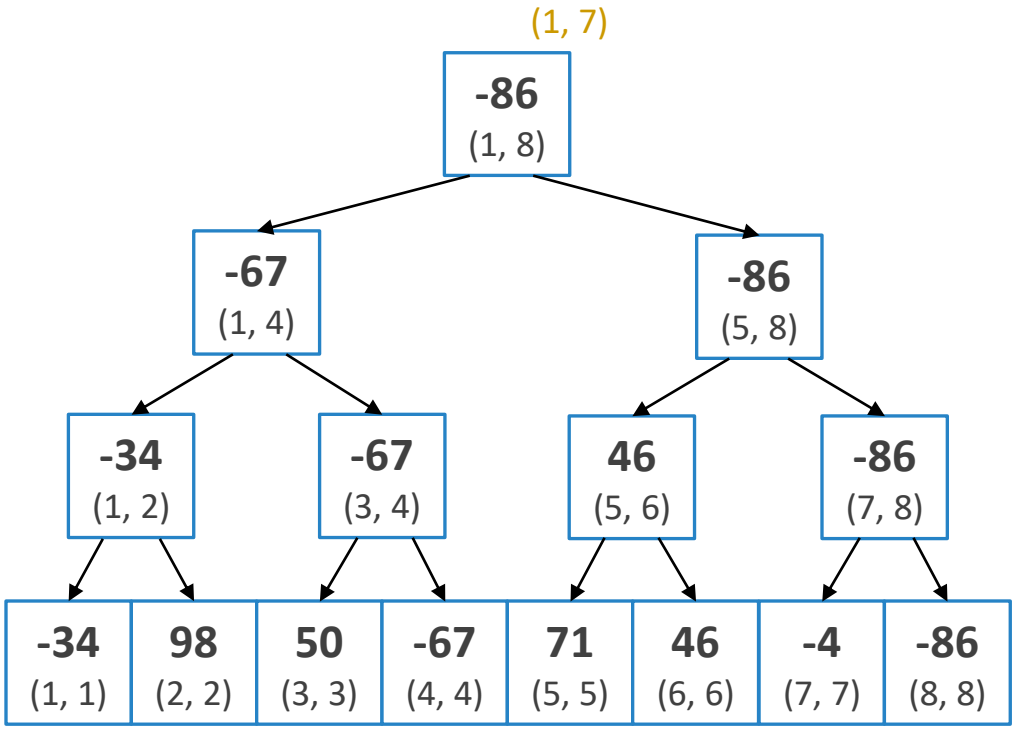
Veamos un segundo ejemplo.

Vamos a ver que pasa con el rango (1,7).

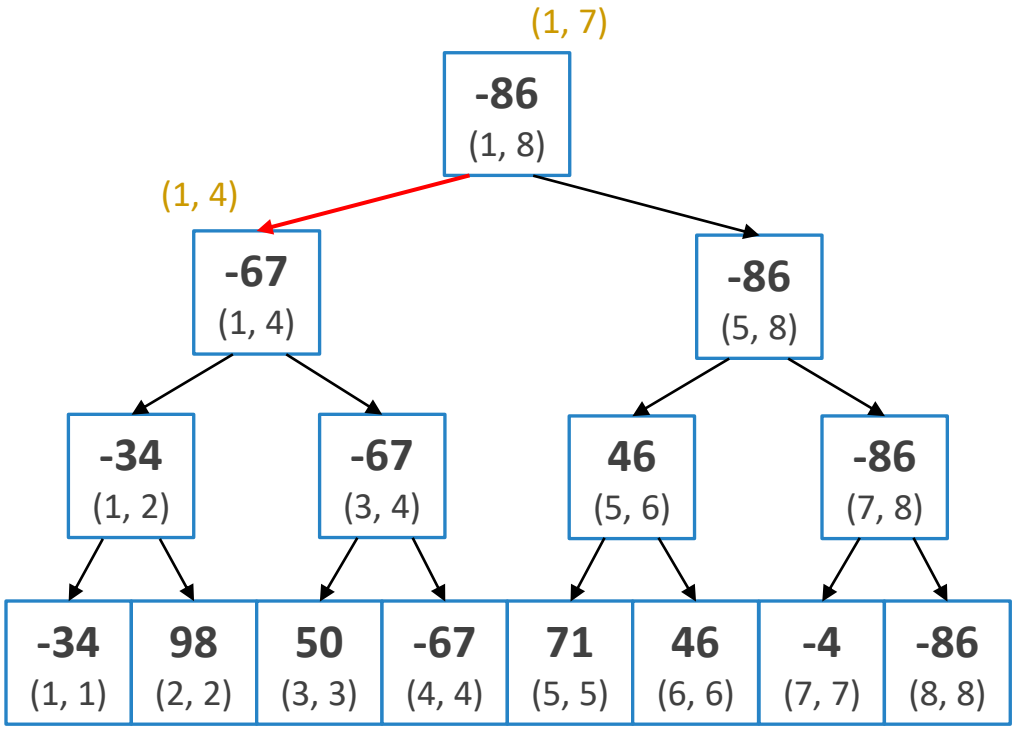
Ahora vamos a seguir el orden **real** que seguiría el programa.



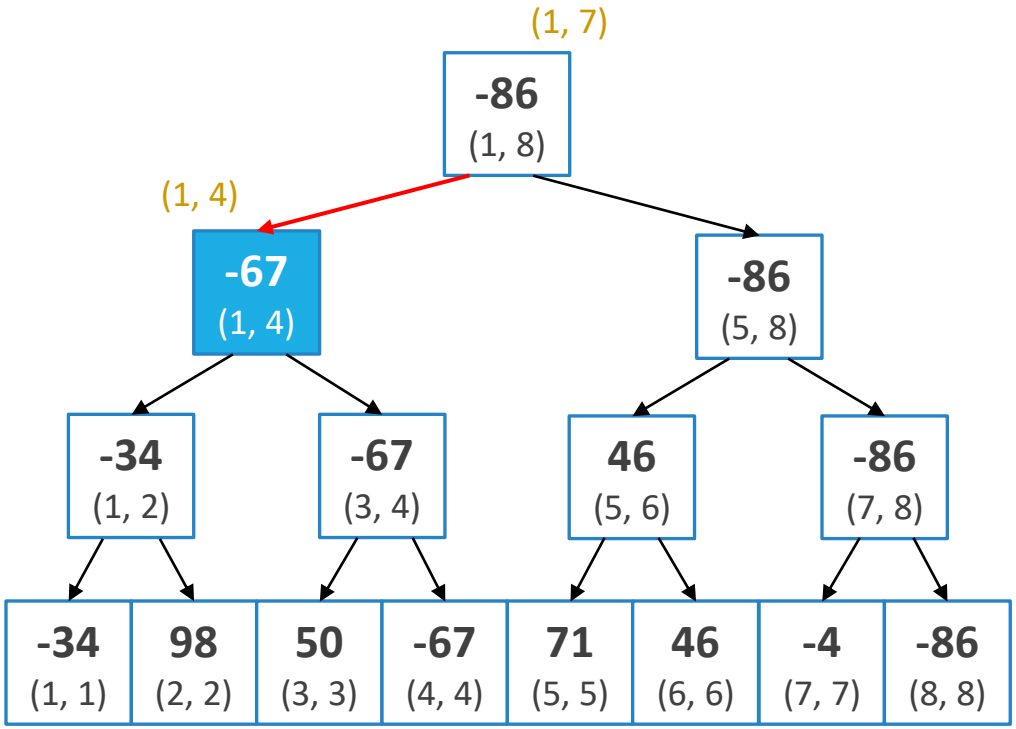
Segment Tree – Búsqueda



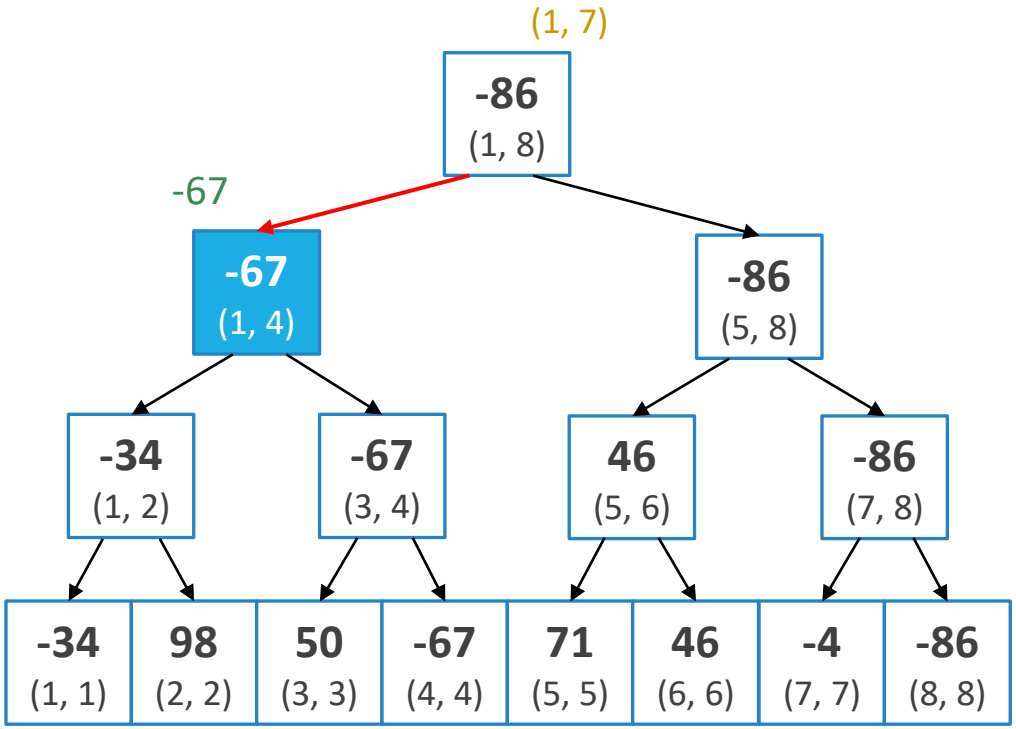
Segment Tree – Búsqueda



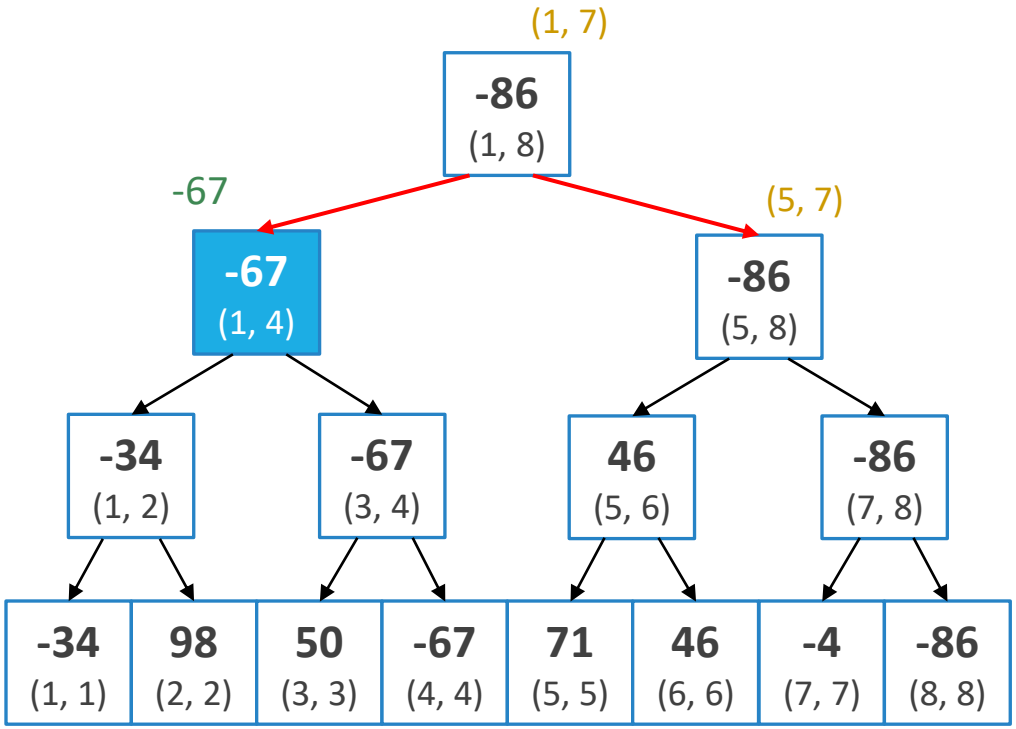
Segment Tree – Búsqueda



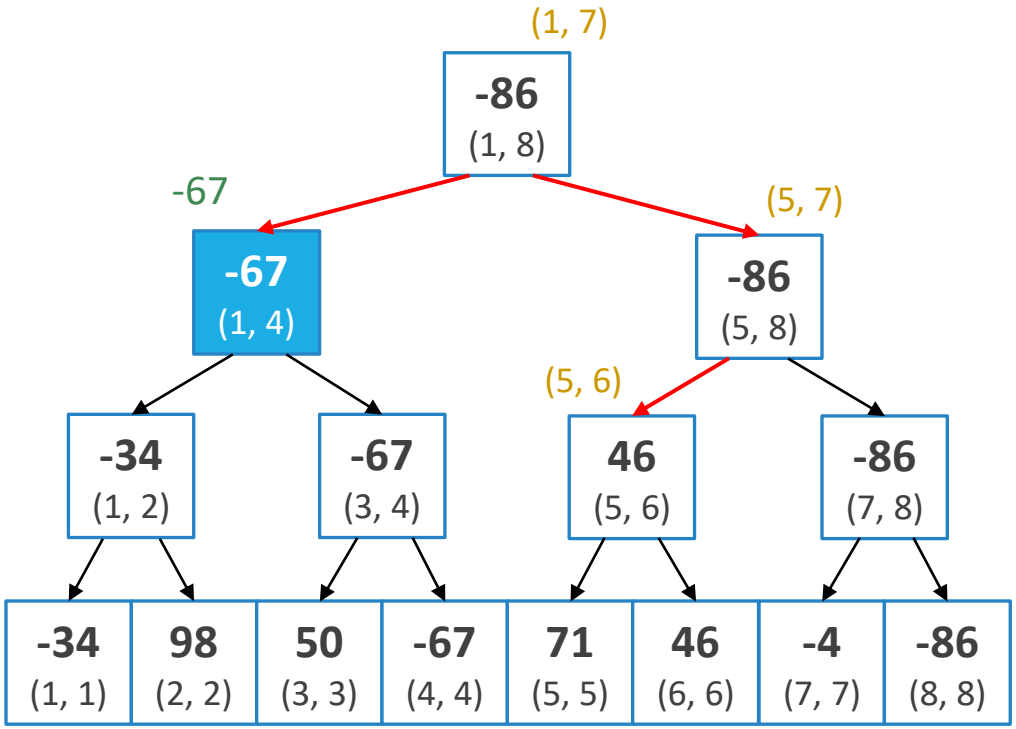
Segment Tree – Búsqueda



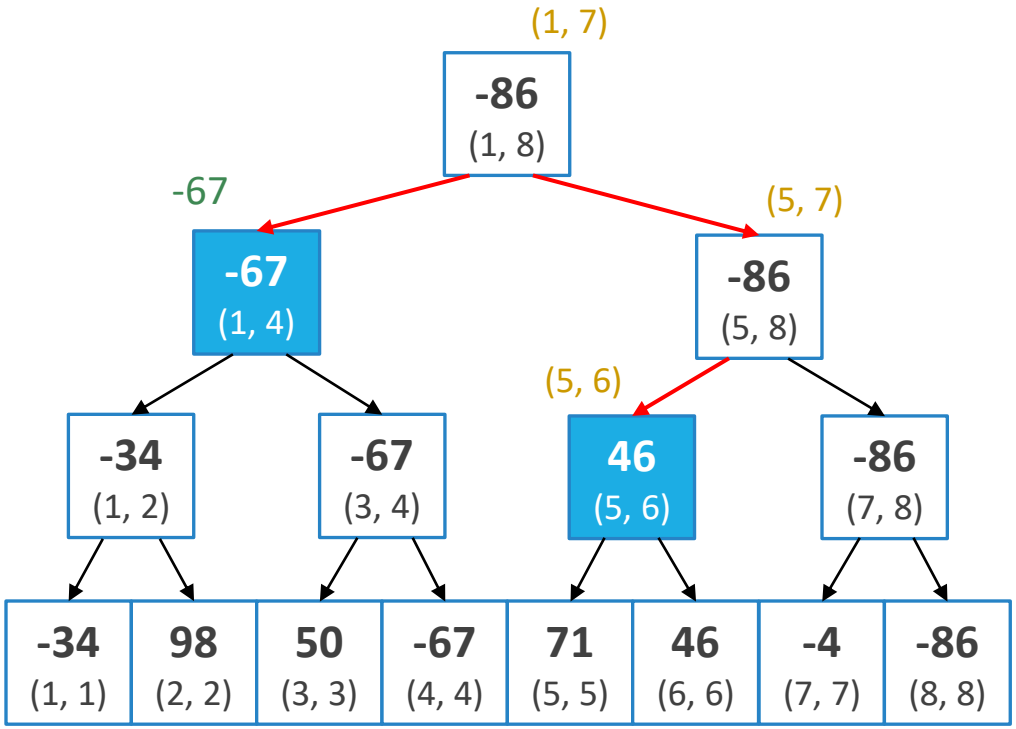
Segment Tree – Búsqueda



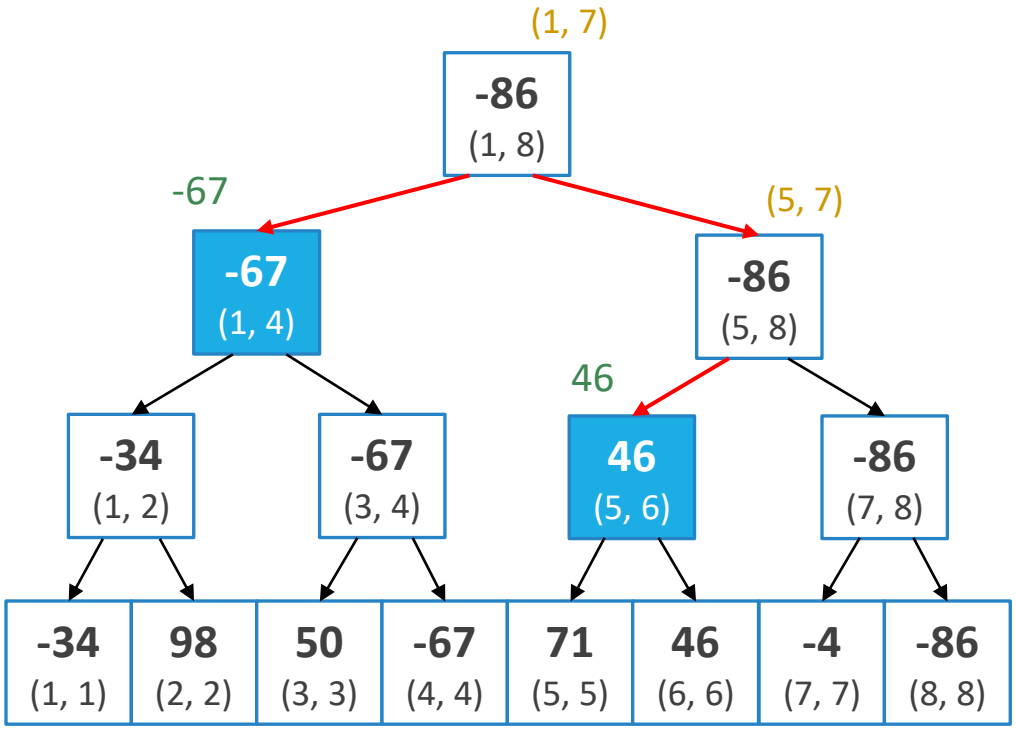
Segment Tree – Búsqueda



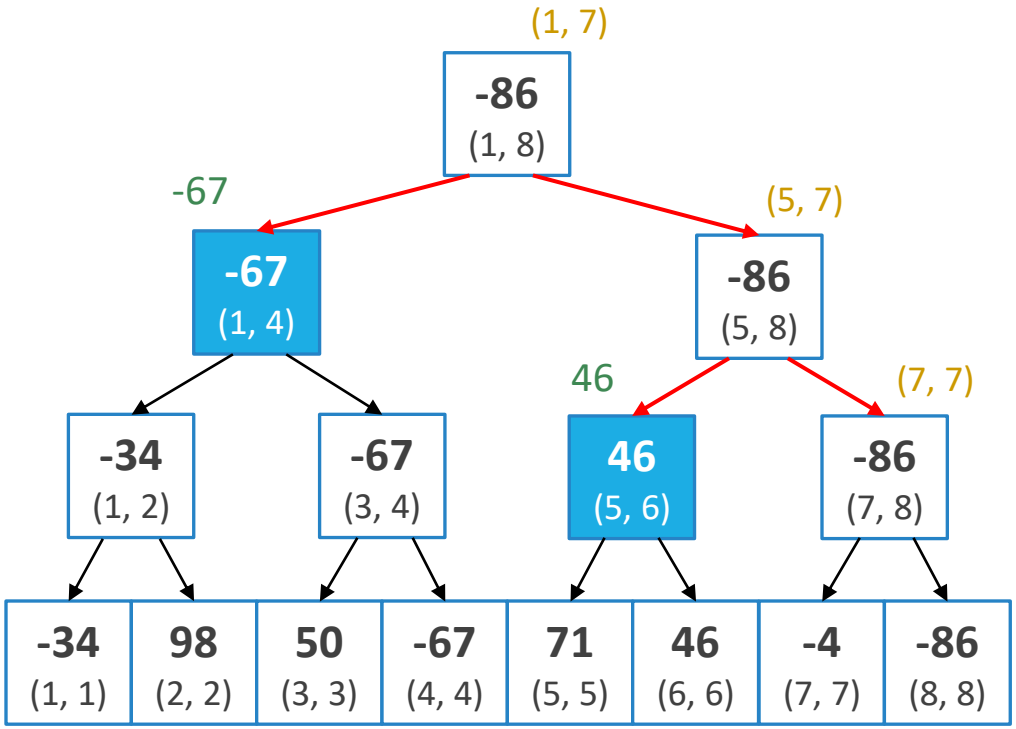
Segment Tree – Búsqueda



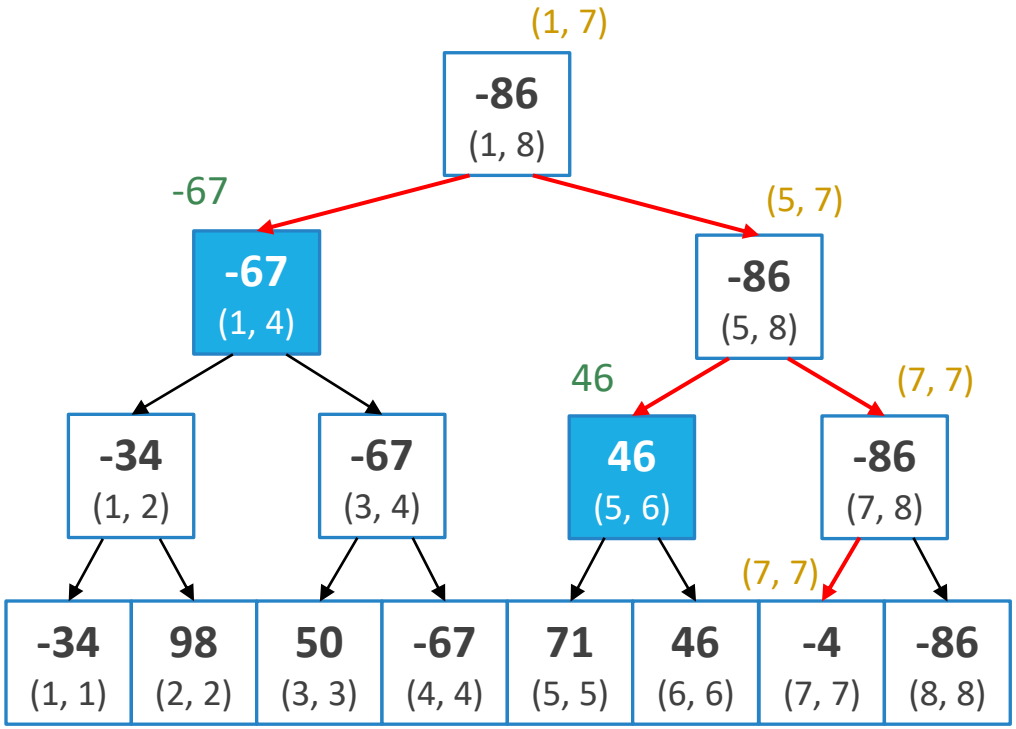
Segment Tree – Búsqueda



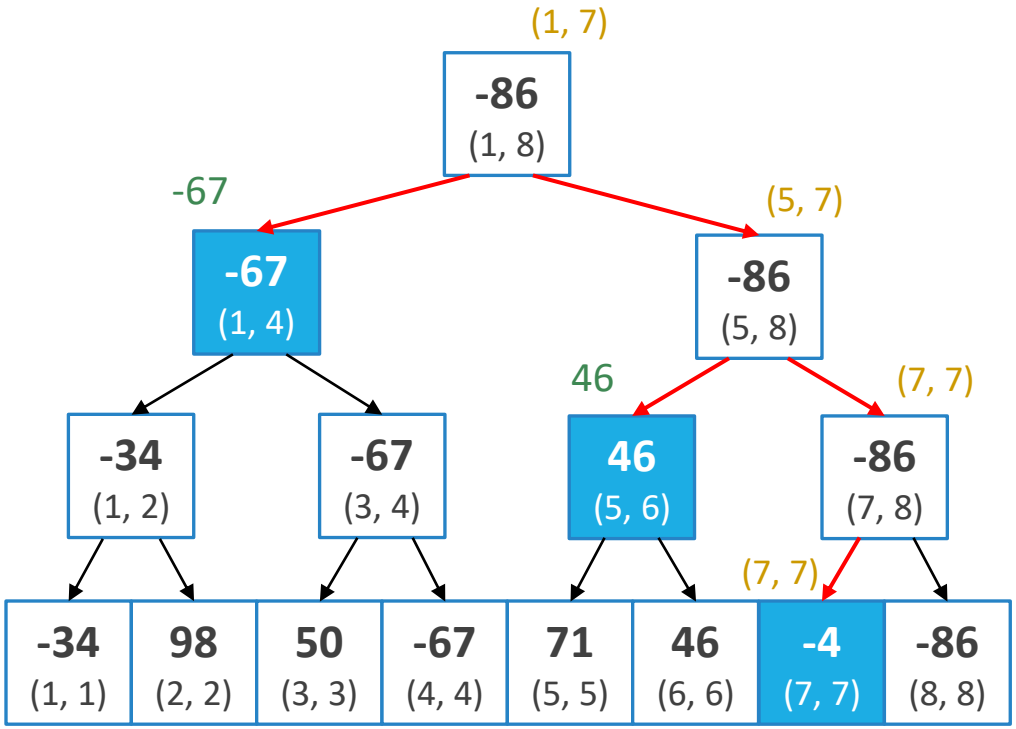
Segment Tree – Búsqueda



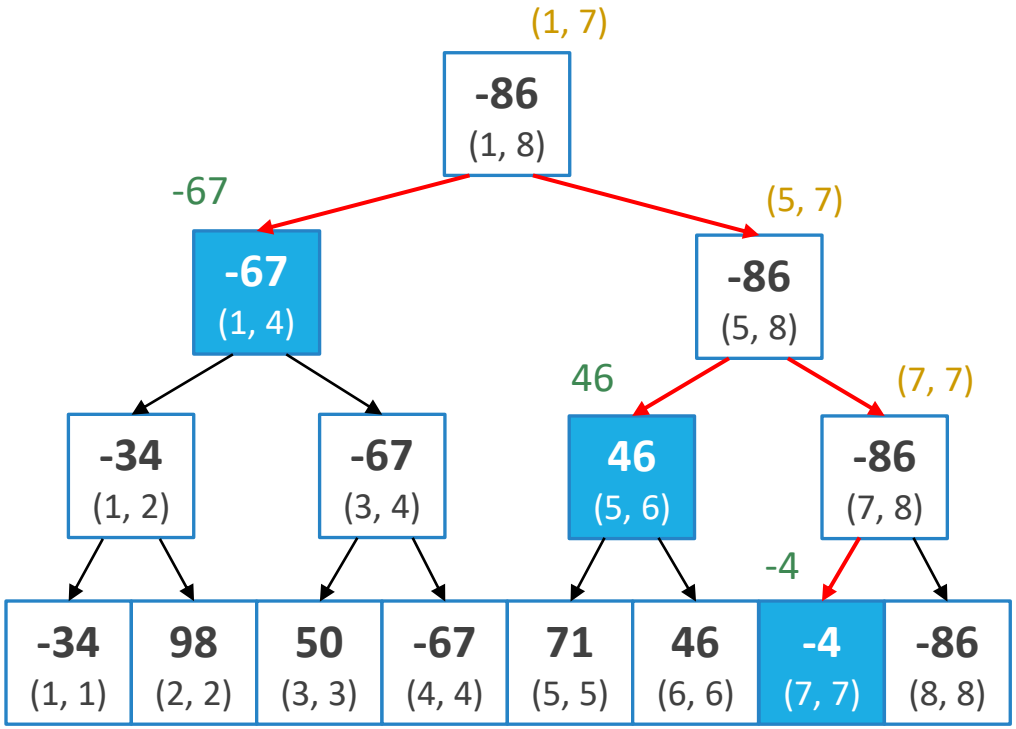
Segment Tree – Búsqueda



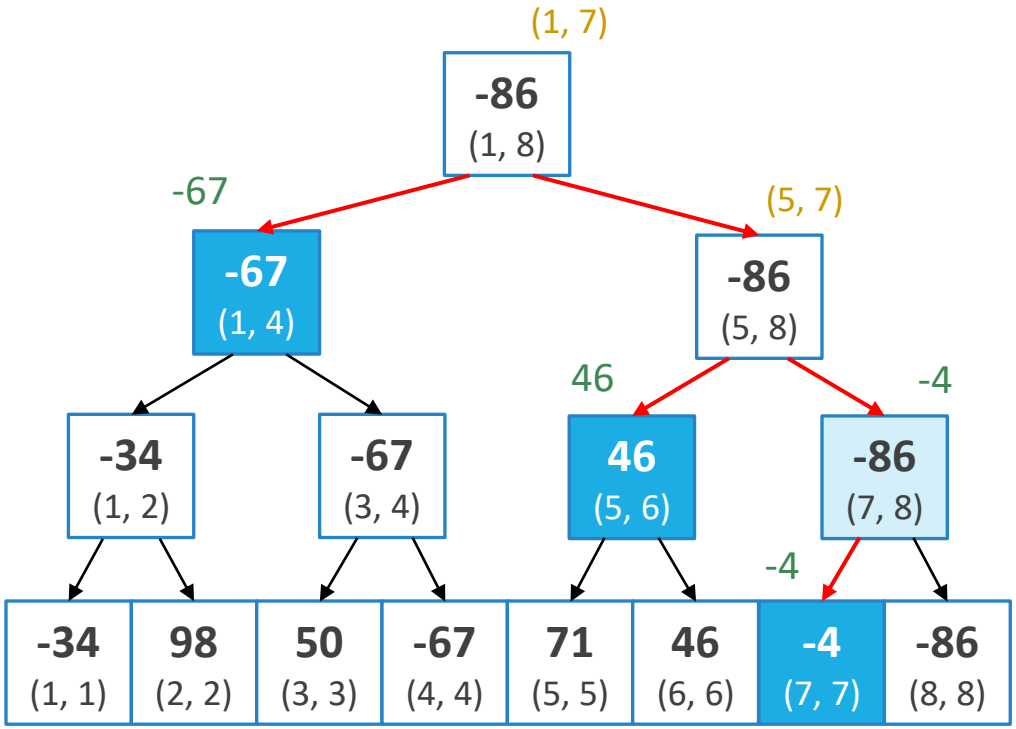
Segment Tree – Búsqueda



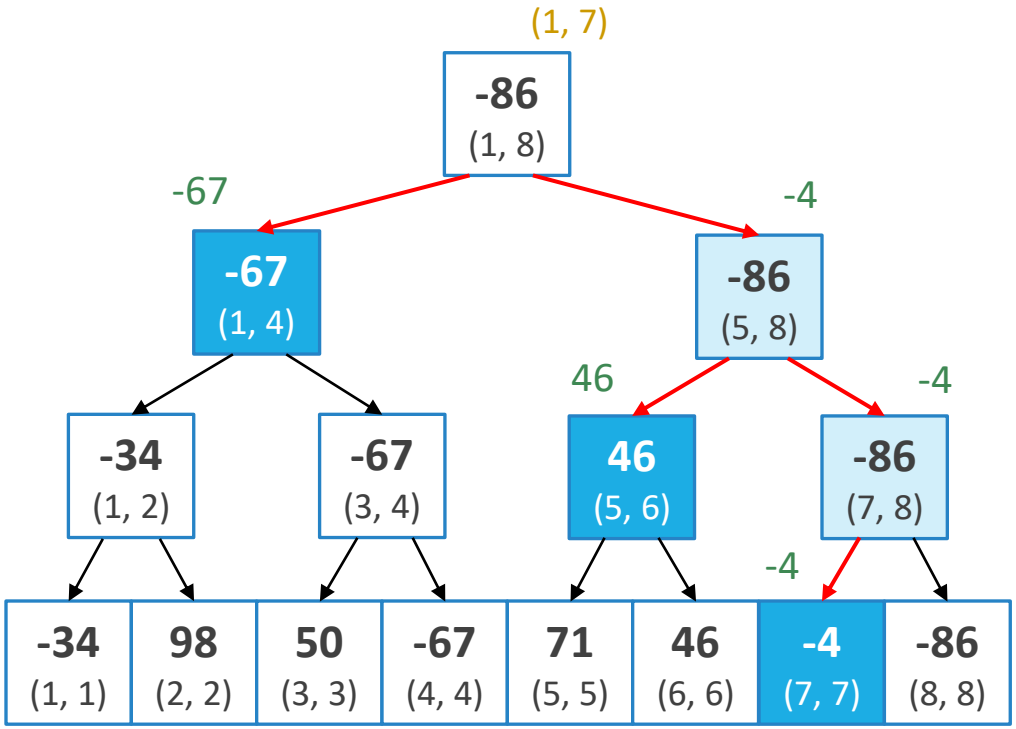
Segment Tree – Búsqueda



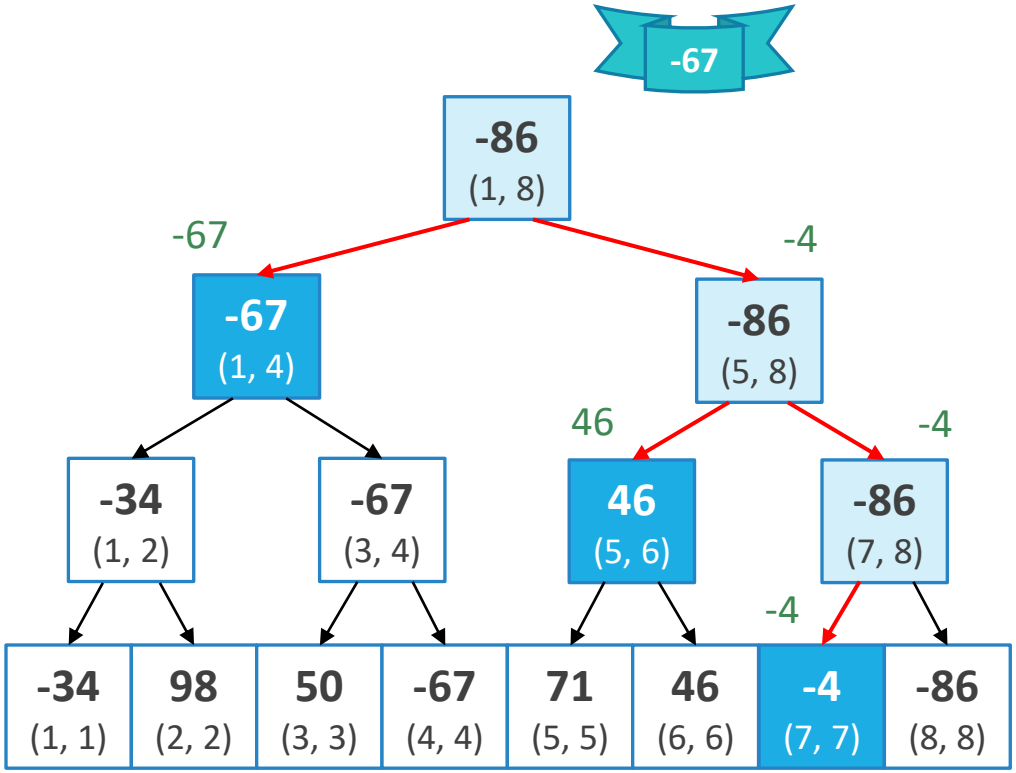
Segment Tree – Búsqueda



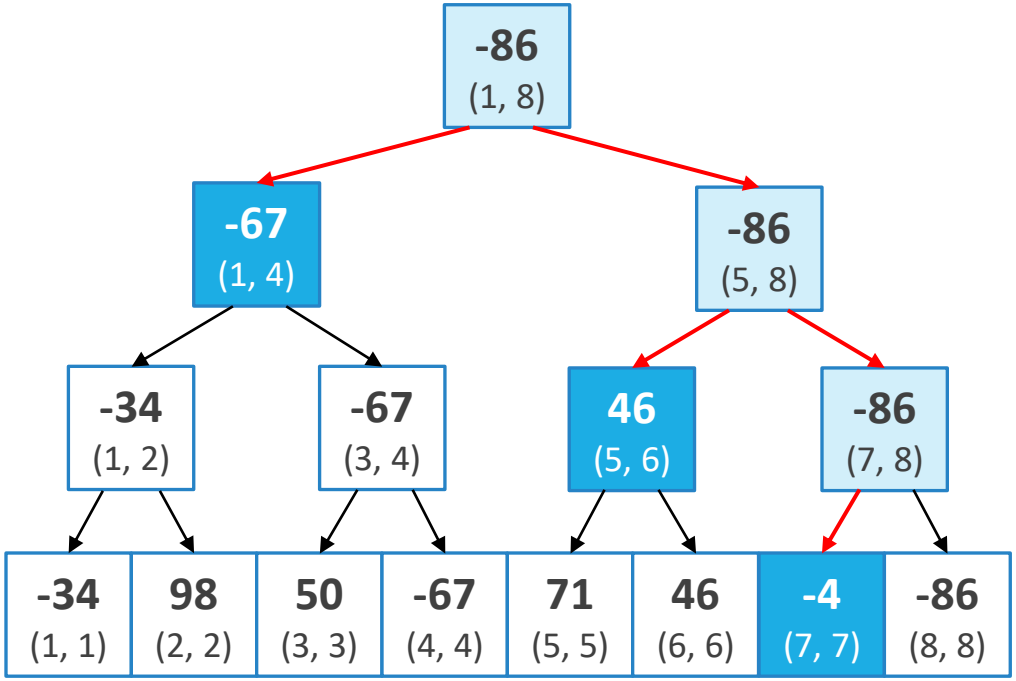
Segment Tree – Búsqueda



Segment Tree – Búsqueda



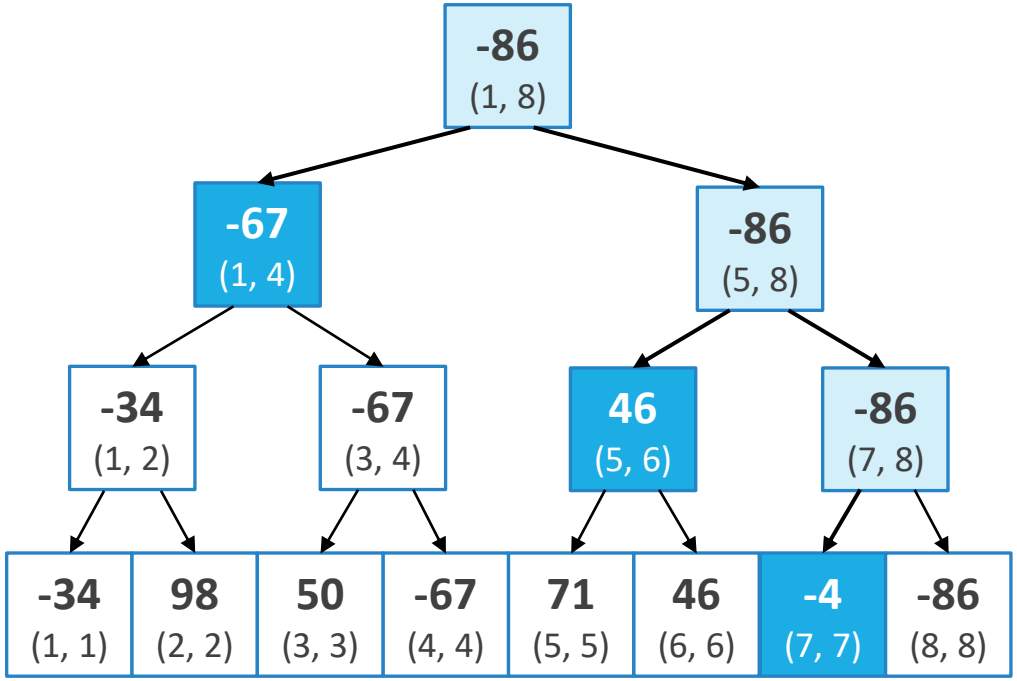
Segment Tree – Más análisis



Segment Tree – Más análisis

Sabemos que en cada búsqueda solo revisamos los valores de O(log *n*) nodos.

Pero no podemos decir que la búsqueda completa toma siempre O(log *n*).

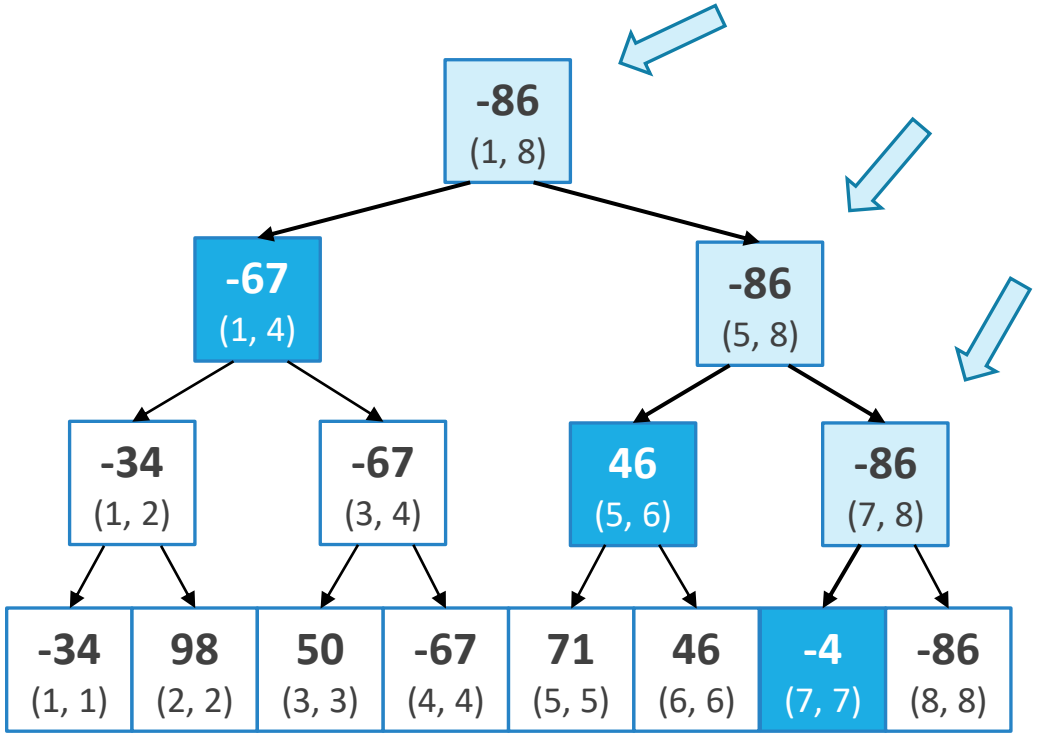


Segment Tree – Más análisis

Sabemos que en cada búsqueda solo revisamos los valores de O(log *n*) nodos.

Pero no podemos decir que la búsqueda completa toma siempre O(log *n*).

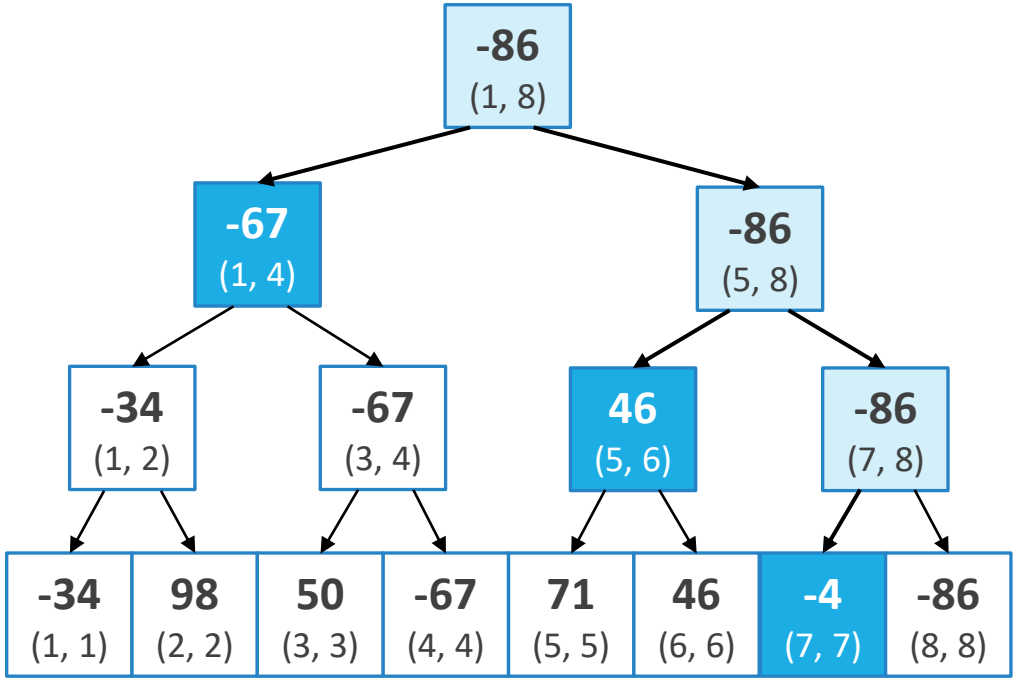
¿Qué hay de estos nodos?



Segment Tree – Más análisis

Vamos a separar la búsqueda en nodos de valor y nodos de búsqueda.

Los nodos de valor son aquellos en que revisamos su **valor**, y los nodos de búsqueda son los que no.



Segment Tree – Más análisis

La gran pregunta es:

¿Cuántos nodos de búsqueda vemos en el peor caso?

Vamos a argumentar que en cada nivel no podemos ver muchos nodos de búsqueda.

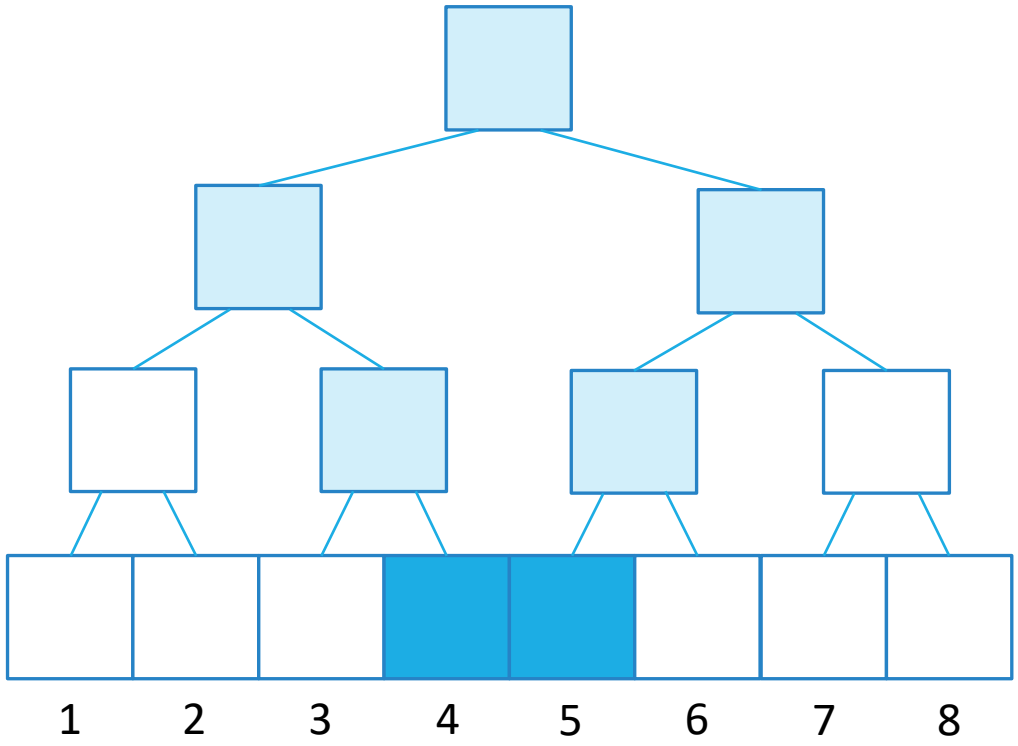
Segment Tree – Análisis

¿Podemos tener dos nodos de búsqueda en el mismo nivel?

Segment Tree – Análisis

¿Podemos tener dos nodos de búsqueda en el mismo nivel?

Perfectamente.



Segment Tree – Análisis

¿Podemos tener tres?

Segment Tree – Análisis

¿Podemos tener tres?

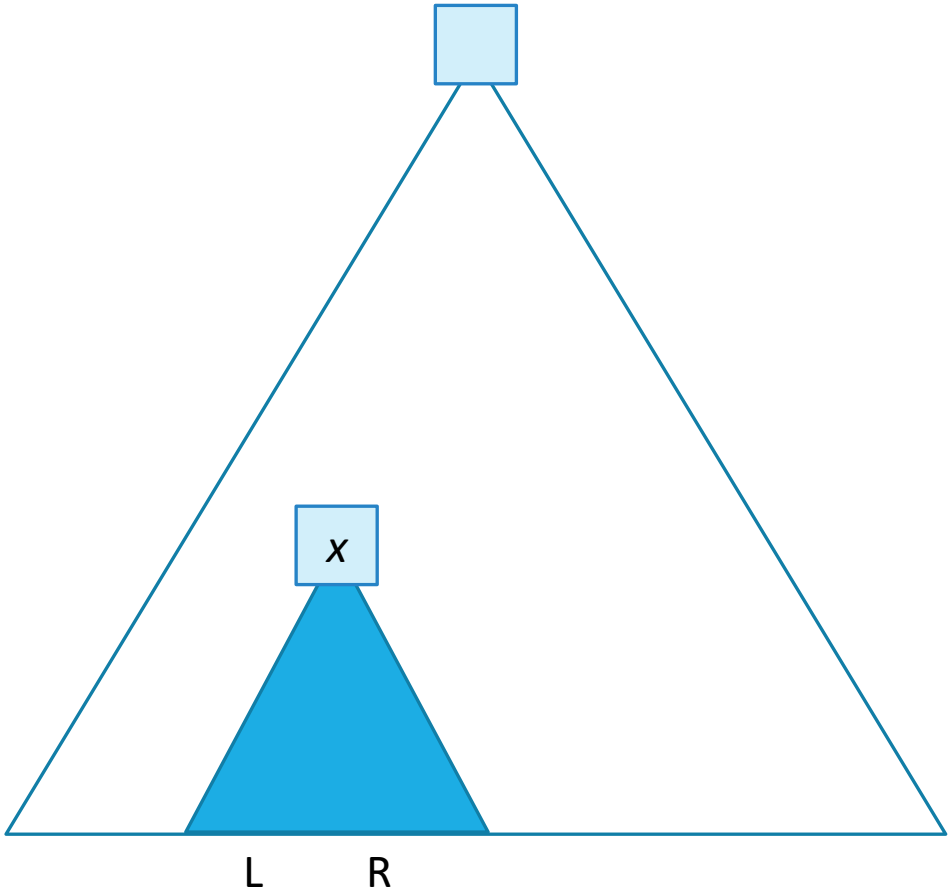
Acá la respuesta es no.

Primero veamos cómo es que se ve una búsqueda arbitraria.

Segment Tree – Análisis

Pensemos en el nodo más alto de la búsqueda cuyo rango contiene a (L, R).

¿Cómo se ve el camino que va de la raíz hacia él?



Segment Tree – Análisis

Pensemos en el nodo más alto de la búsqueda cuyo rango contiene a (L, R).

¿Cómo se ve el camino que va de la raíz hacia él?

¡Son sólo nodos de búsqueda!