Mejoras sobre un Segment Tree Persistencia y Lazy Propagation

Por Lautaro Lasorsa | Curso OIA - UNLaM 2020

Persistencia | Introducción

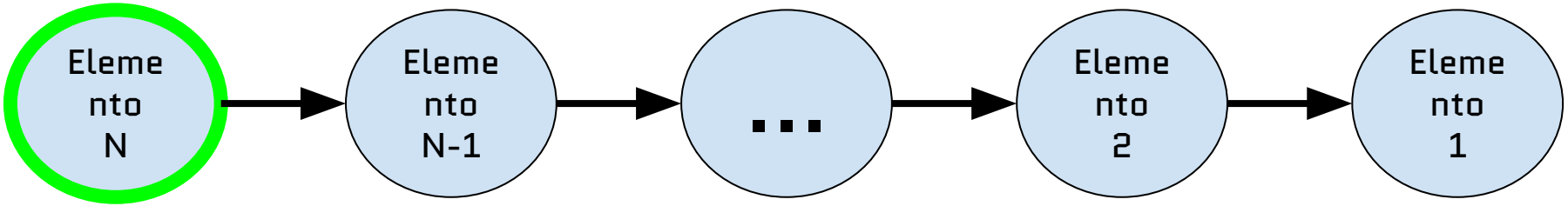
En una estructura de datos, la persistencia es poder guardarnos las diferentes versiones de la misma, sin hacer una copia de toda la estructura.

Es decir, que al realizar cambios no perdamos el estado que tenía la estructura de datos antes de realizar dicho cambio (pudiendo incluso tener varias ramas de versiones independientes)

Esto no es posible en todas las estructuras de datos, pero si lo es en el Segment Tree, y en un ejemplo más sencillo: una pila

Persistencia | Pila 1

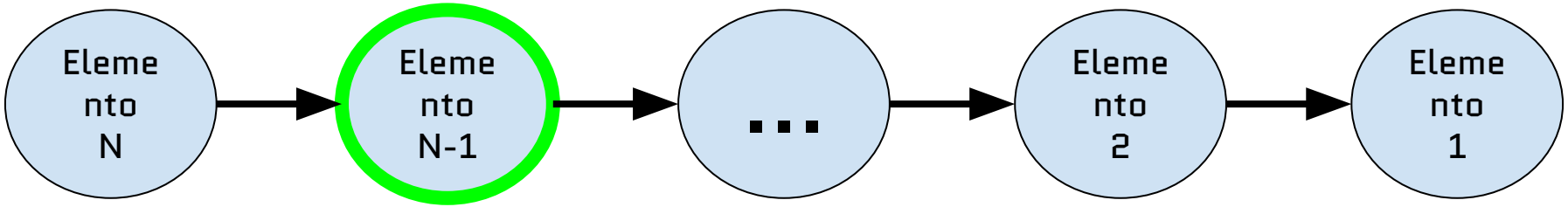
Si bien una forma de implementar una pila es utilizando un vector, existe otra, que consiste en hacerla utilizando un grafo. De la siguiente manera.



En este caso, el primer elemento es aquel que no apunta a ninguno. Y el último elemento es aquel al cual no apunta ninguno. A ese elemento accedemos al acceder a la pila.

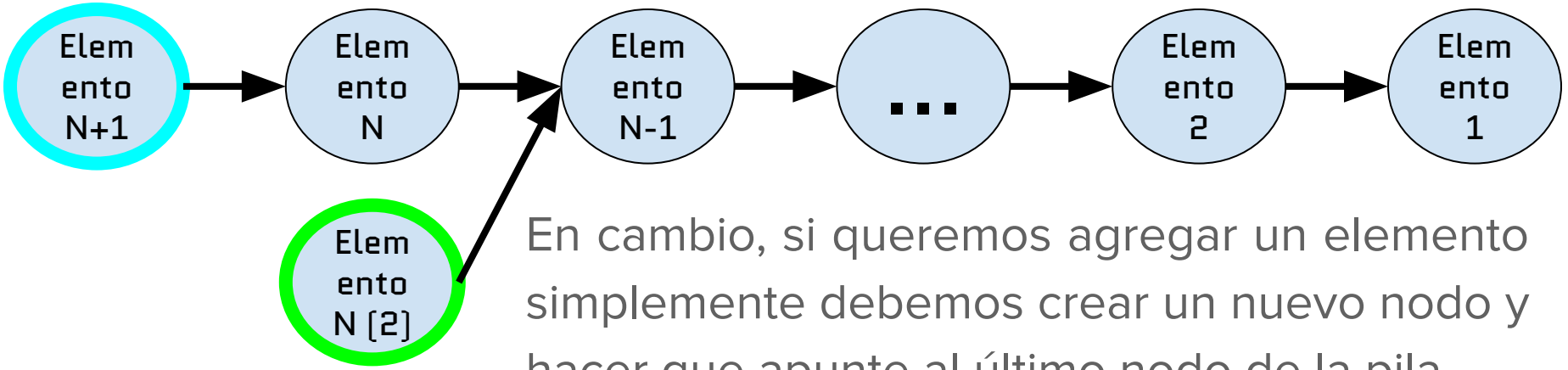
Persistencia | Pila 2

Si yo quiero borrar el último elemento, no necesito borrarlo explícitamente (liberar esa memoria), ya que una referencia al nodo (N-1) representa en sí a una pila sin el elemento N.



Por tanto, podemos simultáneamente guardar una referencia al nodo N y al nodo (N-1), y nos guardaremos los 2 estados de la pila sin hacer una copia de la pila entera.

Persistencia | Pila 3



En cambio, si queremos agregar un elemento simplemente debemos crear un nuevo nodo y hacer que apunte al último nodo de la pila.

Y si nos guardamos tanto la referencia a la pila anterior como al nodo que estamos creando (que representa la nueva pila), podemos guardar ambos estados simultáneamente. En este ejemplo el se ve con los nodos N(2) y (N+1), que son ramas diferentes de updates.

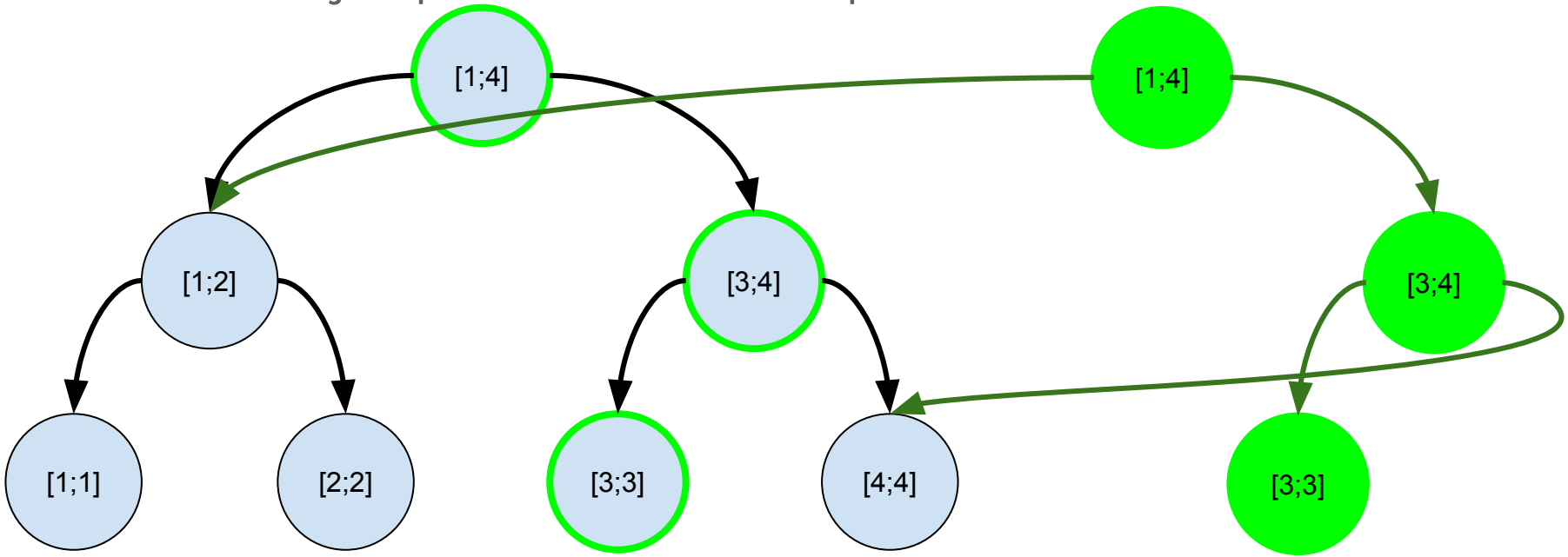
Persistencia | Segment Tree 1

En el caso del Segment Tree, lo interesante para hacer persistencia es que cuando hacemos una actualización podemos solo crear nuevas copias de los nodos que se modifican en el update, conservando todos los demás, haciendo que los nuevos referencian a la estructura ya existente.

A nivel implementativo deberemos usar un struct/clase, ya que cada nodo debe guardar cuales son sus hijos (es decir, ya no podemos hacer el juego de índices como en la versión normal)

Persistencia | Segment Tree 2

La versión del Segment Tree es representada por la referencia a su raíz. En este ejemplo modificamos la posición 3.



Persistencia | Segment Tree 3

Notar que en este caso, tanto la actualización como la consulta deben hacerse forma recursiva. (o al menos quedan extremadamente más cómodas así)

Por otro lado, a la hora de implementar un ST Persistente lo recomendable es la técnica de “bolsa de nodos”, es decir crear inicialmente un gran array de la estructura nodo, y que los nodos que usemos sean posiciones en este array.

Esto es más eficiente que usar la función new() de C++ y utilizar punteros a nodo (que los reemplazamos con índices en el array)

Persistencia | Segment Tree 4

Notar que en cada actualización solo se modifican log(N) nodos, por lo cual si tenemos un vector de N posiciones y realizamos Q updates, la complejidad tanto en tiempo como en espacio será del orden de Q*log(N).

Persistencia | Segment Tree 5

Una variante de la persistencia es la llamada Lazy Creation, que consiste en que originalmente todos nuestras posiciones tienen un valor por defecto (en general, y lo más cómodo, el neutro de nuestra operación).

En este caso, podemos partir de que solo tenemos la raíz, y solo creamos los nodos que se modifiquen durante una actualización, lo que nos permite representar vectores de tamaños mucho más grandes que la forma usual, manteniendo una complejidad que solo depende de las actualizaciones que hagamos.

Lazy Propagation | Introducción

Lazy Propagation es una técnica que puede utilizarse solamente en un Segment Tree de una dimensión, y permite realizar actualizaciones en rango.

La idea consiste en que cada nodo va a guardarse no sólo el resultado de la operación en el rango que representa, sino también las actualizaciones que lo afectan. El nodo debe tener la lógica interna para utilizar esta información para responder la query.

Propagar es cuando un nodo se desprende de una actualización y le pasa esa información a sus hijos para que se actualicen.

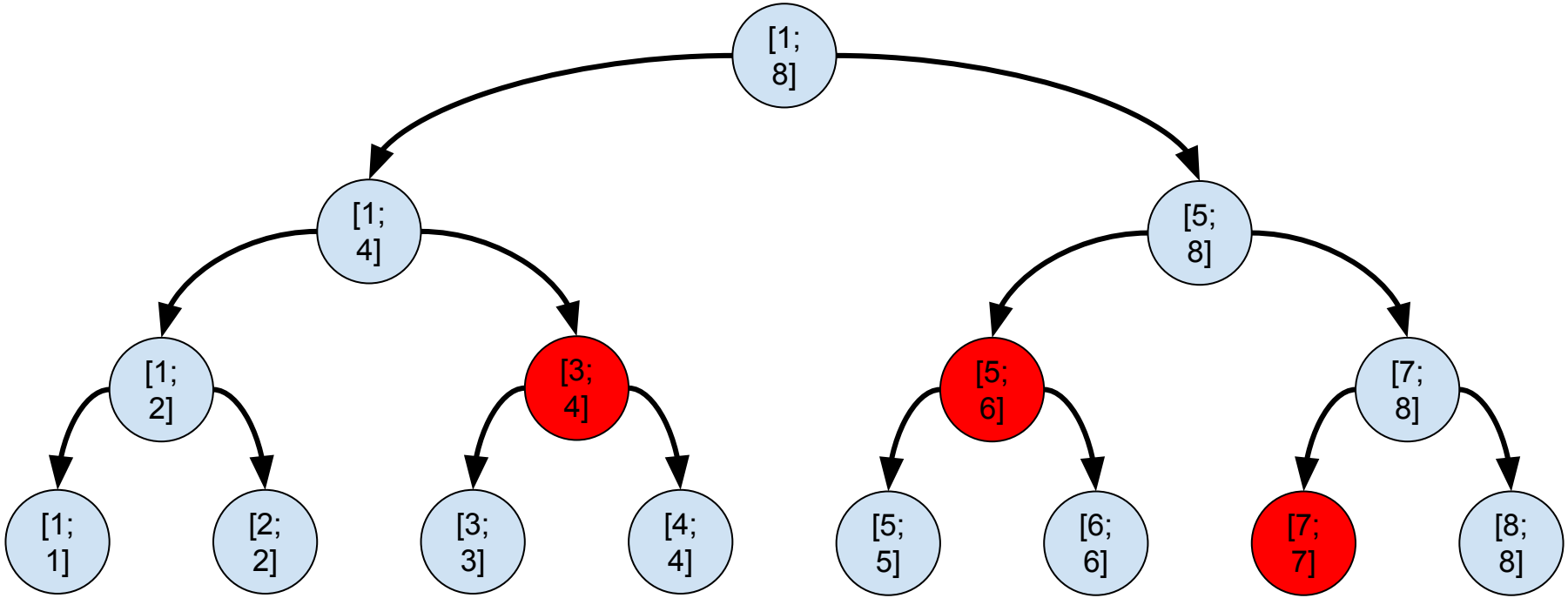
Lazy Propagation | Propagación

Cuando una query o update alcanza un nodo están las siguientes posibilidades:

- Que no abarque en nada al rango del nodo, por lo cual no hay nada que hacer.
- Que abarque completamente al rango nodo, por lo que podrá actualizar ese nodo o responder la query solo con ese nodo.
- Que lo abarque parcialmente, caso en el cuál debe primero propagar los updates que tiene guardados a sus hijos, y luego pasar la recursión actual a sus hijos.

Lazy Propagation | Ejemplo 1

Ejemplo de actualización en el rango [3;7]



Lazy Propagation | Ejemplo 2

Un ejemplo en el que puedo aplicar Lazy Propagation es este:

- Tengo un vector de enteros.
- Realizó queries de suma en rango.
- Las actualizaciones son aumentar todos los valores de un rango en un entero dado (que puede ser negativo).

En este caso los nodos se guardaran la suma de las actualizaciones que no propagaron (al propagar esta suma pasa a 0). La respuesta de un nodo es la suma de las de sus hijos más el valor lazy multiplicado por el tamaño del rango que representa.

Implementaciones

Segment Tree Persistente :

https://pastebin.com/qbja3Wyq

Segment Tree con Lazy Propagation:

https://pastebin.com/4aju5pYS

Ejemplos de problemas

Persistencia:

- https://www.spoj.com/problems/MKTHNUM/
- https://www.spoj.com/problems/DQUERY/
- https://www.spoj.com/problems/PSEGTREE/
- https://cses.fi/problemset/task/1737

Lazy Propagation:

- https://cses.fi/problemset/task/1735
- https://cses.fi/problemset/task/1736

Gracias por ver!