Lazy Propagation 2: The Invariant Approach

Veteran Track

Gabee De Vera

Recall

- We learned how to implement Lazy Propagation last week!
- Lazy Propagation allows us to support *both* range queries *and* range updates in $O(\log n)$ time per query.
- Admittedly, Lazy Propagation is difficult not only to implement, but also to conceptualize. There are *many moving parts* 😢
- Today, we will learn about an *alternative way to conceptualize Lazy Propagation*.

Recall

- Back when we were doing binary search, we encountered the **invariant-based approach**.
- When doing invariant-based binary search, you always think: "what remains true as I proceed with the binary search?"
- For example, when binary searching for the largest value in a sorted list a that is less than or equal to v, we maintain two sets of indices L and R, representing everything *we know* is less than or equal to v and greater than v, respectively.
- Then, you can think of the invariant as "what remains true". In this case, the invariant is that everything in L is less than or equal to v, and everything in R is greater than v.

Recall

Invariant-based binary search simply "updates" what we know to be true at each step of the algorithm

Invariant-Based Lazy Propagation

Invariant-Based Lazy Propagation

- Applying the invariant-based approach to lazy propagation means considering *what remains true* after applying a range update.
- Again, recall that in a lazy propagating segment tree, we only ever need to worry about lazily updating a node if the query interval completely contains the node interval. Therefore, we only need to consider the case where we apply a range update over the whole interval.
- Thus, we can rephrase our goal as follows: Consider *what remains true* after applying a range upate *over the entire interval*.

Example: Counting 1s in a Range

- Consider the following problem. You have an array consisting of only 0s and 1s. You need to efficiently support the following two operations:
    1. Range sum
    2. Range NOT (i.e., all 0s become 1s and vice versa)

Example: Counting 1s in a Range

- Consider the following problem. You have an array consisting of only 0s and 1s. You need to efficiently support the following two operations:
    i. Range sum
    ii. Range NOT (i.e., all 0s become 1s and vice versa)
- To solve this problem, we will consider *what remains true* after applying a range NOT over the entire interval.

Example: Counting 1s in a Range

- Notice that, to compute the range sum, you simply need to *count the number of* 1*s in an interval*.
- What happens to the number of 1s after negating all the bits in the array?

Example: Counting 1s in a Range

- Notice that, to compute the range sum, you simply need to *count the number of* 1*s in an interval*.
- What happens to the number of 1s after negating all the bits in the array? **The number of 1s now becomes the original number of 0s**.
- Therefore, if we *know* the original number of 0s, we will also know the new number of 1s.
- We can also store the number of 0s for each node. Alternatively, you can notice that the number of 0s for each node is simply the size of the interval minus the number of 1s, since all elements are either 0 or 1.

Example: Counting 1s in a Range

- Therefore, to ensure that the number of 1s remains correct after a range update, we do $(\#1) \leftarrow r - l + 1 - (\#1)$, where $(\#1)$ is the number of 1s.
- For the lazy variable, you simply need to store a boolean of whether you need to negate the child nodes or not.

Implementation: Counting 1s in a Range

- The Implementation is quite long. Check the GitHub for the implementation: https://github.com/RedBlazerFlame/reboot-materials/tree/main/compprog-materials/veteran/18-lazy-propagation-2/solutions/1-count.cpp

Example 2: Range Permutation Assignment

- Now, consider this new problem: You are given an array a of nonnegative integers in the range $[0, k)$, where k is a given constant. You need to support the following operations:
  i. Range Sum
  ii. Range Permutation Assignment: You are given a list b of k elements. b is a permutation of the list $[0, 1, 2, \ldots, k-1]$. For each number $a[i]$ in the range, you must assign it to $b[a[i]]$. In other words, all 0s become $b[0]$, all 1s become $b[1]$, all 2s become $b[2]$, and so on.
- Your goal is to find an algorithm that runs in $O(k \log n)$ per query.

Example 2: Range Permutation Assignment

- Let us consider an example. Suppose you have an array $[3, 1, 4, 1, 5]$, with $k = 6$
.
- Suppose that we want to perform a range repermutation over the whole array, with $b = [2, 3, 0, 5, 1, 4]$.
- Since $b[3] = 5$, all 3s become 5.
- Since $b[1] = 3$, all 1s become 3s.
- Since $b[4] = 1$, all 4s become 1s.
- Finally, since $b[5] = 4$, all 5s become 4s.
- Thus, the array will become $[5, 3, 1, 3, 4]$.

Example 2: Range Permutation Assignment

- Now, let us solve this problem. Again, we consider what remains true *after updating the entire interval*.

Example 2: Range Permutation Assignment

- Now, let us solve this problem. Again, we consider what remains true *after updating the entire interval*.
- Let us consider how many times each number appears in the interval. Denote $\mathrm{count}[i]$ as the number of times the number i appears in the interval.

Example 2: Range Permutation Assignment

- Now, let us solve this problem. Again, we consider what remains true *after updating the entire interval*.
- Let us consider how many times each number appears in the interval. Denote $\mathrm{count}[i]$ as the number of times the number i appears in the interval.
- Consider a number i. After an update, it becomes $b[i]$.
- Therefore, we have the identity $\mathrm{count}_{\mathrm{new}}[b[i]] = \mathrm{count}[i]$, since all numbers that are i become $b[i]$.

Example 2: Range Permutation Assignment

- Now, let us solve this problem. Again, we consider what remains true *after updating the entire interval*.
- Let us consider how many times each number appears in the interval. Denote $\mathrm{count}[i]$ as the number of times the number i appears in the interval.
- Consider a number i. After an update, it becomes $b[i]$.
- Therefore, we have the identity $\mathrm{count}_{\mathrm{new}}[b[i]] = \mathrm{count}[i]$, since all numbers that are i become $b[i]$.
- Since i could be any number from 0 to $k-1$, we must store the counts of all numbers from 0 to $k-1$.

Example 2: Range Permutation Assignment

- Then, to perform a range update, simply perform the reassignment that we described previously if the update interval contains the current node's interval. Otherwise, recurse both children.
- Then, to combine the states of the two children, simply sum up the counts of each number i. in other words, $\text{count}[i] := \text{count}_l[i] + \text{count}_r[i]$.
- This takes $O(k \log n)$ time.

Example 2: Range Permutation Assignment

- To obtain the range sum, notice that each number i contributes $i \cdot \mathrm{count}[i]$ to the sum.
- You can simply sum up the contributions of all numbers from 0 to $k-1$ in $O(k)$ time.
- Therefore, the total complexity of a range sum is $O(k \log n)$.

Example 2: Range Permutation Assignment

- It should be noted that this problem is *not* a trivial application of lazy propagation.
- In particular, it may be hard to combine two lazy updates together! I leave it to you to figure this out. You may consult the implementation in the next slide for help or ask one of the trainers for further clarifications.

Implementation: Range Permutation Assignment

- The Implementation is quite long. Check the GitHub for the implementation: https://github.com/RedBlazerFlame/reboot-materials/tree/main/compprog-materials/veteran/18-lazy-propagation-2/solutions/repermute.cpp

Homework

- Check the Reboot Website for the homework this week. This week, you wil only be assigned one problem, but it will be *very difficult*. In fact, you'll be tackling a previous IOI problem! Feel free to **collaborate and discuss with your fellow trainees**. You may also **ask for help from the trainers** and even **read the editorial (but only when you're really stuck)** 😄
- For this week's problem, my biggest tip for you is to consider what remains true after applying an operation. Try experimenting with the state of each node! ^^