Persistent Data Structures — Tutorial

Andrew Stankevich

September 28, 2019

Problem Tutorial: "Snowmen"

It is quite easy to see that the process described resembles operations with persistent stack. To answer queries, let us also keep track of the sum of elements in the stack.

Problem Tutorial: "Persistent Queue"

One way to implement persistent queue is to use array-based queue implementation and persistent array.

Note that implementing queue using two stacks would not work, because persistence doesn't work well with amortized analysis.

There is, however, a way to implement queue using stacks with all operations in real time, see Okasaki "Simple and Efficient Purely Functional Queues and Deques" for more details.

Another way that has a logarithmic slowdown is to use the minimum version of catable deques of Buchsbaum and Tarjan.

Just google for the keywords to get references to nice articles.

Problem Tutorial: "Persistent Array"

There are several ways to implement persistent array, the most notable is probably using segment tree, but with no operation. We just store values in leaves of the tree, and nothing in the internal nodes. To change the element of the array, we just change the corresponding value in the leaf, and user general persistence trick by creating instead of modifying, as usually.

Side note, any array based data structure can be made persistent using persistent array, with logarithmic slowdown (and lots of extra memory).

Problem Tutorial: "Intercity Express"

Let us again first solve the problem in offline setting.

Create three types of events:

- Seat c becomes free at station a, next time it is occupied at station b.
- Seat c becomes occupied at station a.
- Ticket is requested from station a to station b.

Sort request by station a. To process the request use Range Maximum Query segments tree storing the next station the seat becomes occupied, 0 if it is already occupied. To process free/occupy events just modify the values in the tree. Query request can be answered using tree descent similar to the previous problem. Go to the left subtree if there is a seat that is free until city b, go to the right subtree otherwise.

Now we can move to the online setting by using persistent segments tree.

Problem Tutorial: "K-th Element on a Segment"

Let us create points (i, a_i).

Let us first learn how to answer queries offline about prefixes. Run scanline by increasing of i, build segment tree on a_i-s as indices, store 1-s for already passed points. Operation in segment tree is sum. To find the k-th element on a prefix, descend in the segment tree when this prefix is processed. When descending, choose whether to go left or right, based on k and the sum in corresponding nodes.

We will now need two improvements.

Improvement 1: from offline setting to online setting. Just replace segments tree with its persistent version, and take the correct version when processing the query.

Improvement 2: move from prefixes to segments. Take two versions of segment tree, for the beginning of the query segment and for the end of the query segment. Now we virtually create their difference, but only use those elements that are needed for the descent, so only $O(\log n)$ elements will be evaluated.

Total complexity: $O(n \log n)$ preprocessing, $O(\log n)$ per query.

Problem Tutorial: "Rollback"

Let us answer to the query with $l_i = 1$. For each element of the array a_i set $b_i = 1$ if a_i is the first occurence of this value, or $b_i = 0$ otherwise. We need to find the longest prefix that has the sum of at least k_i.

Now let us move to $l_i = 2$. We must find the next value $a_j = a_1$, and set $b_j = 1$ because it is now the first occurence. Now the problem is the same: find the longest prefix that has the sum of at least k_i. Let us denote such j as $next[1]$, similarly, set $next[i]$ be the smallest $j > i$ such that $a_j = a_i$. These values can be found in one linear backwards pass.

To find the longest prefix that has the sum of at least k we can use Range Sum Query implementation of a segments tree. We want to find the minimal t such that the prefix sum is equal to k. Consider a node in the tree. If the sum in the left subtree $suml$ is smaller than k, the answer is in the right subtree, and we need to find the leftmost position with the sum $k - suml$ there. Otherwise we descend to the left subtree. The complexity is $O(\log n)$.

Now we can solve the problem in the offline setting. Sort all queries by l_i, and process them in this order. When we move from l_i to $l_i + 1$, set $b_{l_i} = 0$ and $b_{next[l_i]} = 1$, as described above.

To go to the online setting, just use persistent segments tree, as it was described at the lecture.

Problem Tutorial: "Urns and Balls"

The key idea in this problem is to process the movement commands in the reverse order.

Let us start from the end. For each urn let us store the value f_i — the number of the urn where the ball from this urn will be at the end of the process. The initial values (at the end of the process, we go in the reverse order) are $f_i = i$. The movement command from $l_i \dots r_i$ to $m_i \dots n_i$ tells us that the balls from l_i to r_i will finally be at the same urns as balls from m_i to n_i are. So we need to copy the values from the segment from $m_i \dots n_i$ to the segment $l_i \dots r_i$.

Let us use persistent treaps to store the array. To copy values, let us use two splits to cut away the segment $m_i \dots n_i$. Now similarly use two splits, to cut away the segment $l_i \dots r_i$ (we use persistence here by splitting the same original tree again). Now merge the two $1 \dots (l_i - 1)$ and $(r_i + 1) \dots n$ pieces together with the $m_i \dots n_i$ segment to form the new tree.