

# Data Structure IV

## Persistent Segment Trees

# Introduction

We review the notion of persistent data structure. Suppose we have a data structure that supports an update operation  $U$ , which takes an instance of the data structure and returns a new instance after the update. Consider  $D_1$  is an instance of the data structure, and we say it is of version 1.

We shall support the following operations efficiently.

- ▶ Query. Query the data structure at some version  $k$ .
- ▶ Update. Create  $D_{n+1} = U(D_k)$ , where  $n$  is the number of versions that we have ever created before this operation, and  $1 \leq k \leq n$ .

Usually, a linked data structure can be modified to be persistent data structure easily, without suffering the update time, and with additional space as the same with the update time.

So does Segment Trees.

# Application of Persistent Segment Trees


Main technique: direct use of the history version; difference between versions.

We start by an example showing the direct use of the history version.

# Distinct Sum<sup>1</sup>

Suppose we are given an array  $A[1, \dots, n]$ . We are to support the query  $Q(l, r)$  that asks the sum of the *distinct* elements in  $A_{[l,r]}$ .

---

<sup>1</sup><http://acm.hdu.edu.cn/showproblem.php?pid=3333> 

# Discretize $A$

We can discretize  $A$ , so that the value of  $A$  is in  $[1, n]$ . This can be achieved by building a one to one correspondence between the value of  $A$  to  $\{1, 2, \dots, n\}$ , and a possible way to do the mapping is to use STL map (or Java TreeMap).

## Observation

Suppose  $1 \leq r \leq n$  is fixed. We focus on  $A_{[1,r]}$ , and modify it to be  $A^{(r)}[1, 2, \dots, r]$ , such that  $A^{(r)}[i] = A[i]$  if and only if  $i$  is the largest index that the value  $A[i]$  appears, and other  $A^{(r)}[i] = 0$ . This way, to answer  $Q(l, r)$ , we can just return the sum of elements in  $A_{[l,r]}^{(r)}$ .

Moreover,  $A^{(r+1)}$  can be constructed from  $A^{(r)}$  easily, by modifying at most one element in  $A^{(r)}$ , and append at most one element to  $A^{(r)}$ .

# Proprocessing

We use a segment tree  $T_r$  to represent  $A_r$ , and then a query of the form  $Q(l, r)$  can be answered in  $O(\log n)$  time. In particular, we let  $T_r$  support 1) add an integer to some interval representing a single point, and 2) query the sum of some interval.

Then we can construct  $T_1$  to  $T_n$  starting from  $T_0$  that is set to all zeros, and build  $T_{r+1}$  from  $T_r$  in  $O(\log n)$  time and space.



# Query

For query  $Q(l, r)$ , we return the sum from  $l$  to  $r$  in  $T_r$ .

Note that we can use an array to record roots of  $T_i$ 's. Then, switching to  $T_r$  takes constant time.

## $k$ -th Smallest In An Interval<sup>2</sup>

We shall show an example use of the difference of versions.  
Suppose we are given an integer array  $A[1, 2, \dots, n]$ . We are to support the query  $Q(l, r, k)$  that asks the  $k$ -th smallest number in  $A_{[l,r]}$ .

---

<sup>2</sup><http://poj.org/problem?id=2104>

## Auxiliary Question

**Question.** Suppose we have an array  $V$  with  $n$  integers, and it is initially set to all 0. How can we use a segment tree to support the following operations?

- ▶  $U(i, x)$ . Set  $V[i] := x$ , where  $1 \leq x \leq n$ .
- ▶  $Q(k)$ . Query the  $k$ -th smallest integer in the array.

## Auxiliary Array

We are going to maintain an array  $B[1, 2, \dots, n]$  that is initially set to all 0. Whenever we are going to perform  $U(i, x)$ , we set  $B[V[i]] := B[V[i]] - 1$ , and then set  $B[x] = B[x] + 1$ . Then if it is  $Q(k)$ , then we return the smallest  $j$  such that  $\sum_{t=0}^j B[t] \geq k$ .

Intuitively,  $B[j]$  intends to represent the number of occurrences of elements  $V[i]$  such that  $V[i] = j$  for all  $i$ . You can verify the correctness of the way we answer the query.

## Using Segment Tree

We maintain a segment tree for  $B$ . For the update, we only need to support adding an integer to an interval representing a single point.

To support finding the largest  $j$ , we maintain the sum variable for each node in the segment tree. Then we define a recursive procedure  $F(r, k)$  that finds the smallest  $j$  such that  $\sum_{t=r.L}^j B[t] \geq k$ .

If  $r$  represents a leaf, return  $r.L$ .

If  $r$  is not a leaf, then we test if  $r.\text{leftChild.sum} \geq k$ . If this is the case, then we return  $F(r.\text{leftChild}, k)$ . Otherwise, we return  $F(r.\text{rightChild}, k - r.\text{leftChild.sum})$ .

Note that each invocation of the procedure increases the height of  $r$  by 1, and hence the procedure terminates in  $O(\log n)$  time.

## Original Problem: Preprocessing

We go back to the original problem.

So far, we have defined a segment tree that supports 1) add a value to a single point interval and 2) find the  $k$ -th smallest number in the whole interval.

We make this segment tree persistent. Then, define the original version  $T_0$  to be the segment tree with all zero values. Define  $T_{i+1}$  from  $T_i$  by increasing  $T_i[V[i]]$  by 1.

Observe there are  $O(n)$  versions in total, and creating each version takes  $O(\log n)$  time and space.

## Original Problem: Query

To answer a query  $Q(l, r, k)$ , the idea is to perform the finding  $k$ -th smallest operation on the segment tree defined by  $T_r - T_{l-1}$ . By  $T_r - T_{l-1}$ , we mean a new segment tree that has the “sum” value in each node to be the difference of those in  $T_r$  and  $T_{l-1}$ .

However, it is too costly to really build  $T_r - T_{l-1}$ . Instead, we observe that  $T_r$  and  $T_{l-1}$  has the same structure, and in the finding  $k$ -th smallest procedure, only  $O(\log n)$  “sum” value is actually needed.

Therefore, we can calculate each needed “sum” on the fly in the find  $k$ -th smallest procedure, and such a calculation can be done in  $O(1)$ .



## Some Details

We define the modified  $F(r, k)$  procedure  $F'$  with inputs  $F'(r_1, r_2, k)$  that doing the search on the difference tree induced by trees rooted at  $r_2$  and  $r_1$ , where  $r_2$  and  $r_1$  are the roots of two versions of segment trees.

During the procedure, we always let  $r_1$  and  $r_2$  represent the same corresponding nodes in its own version.

If  $r_1$  is a leaf, then return  $r_1.L$ .

Otherwise, test if  $r_1.\text{leftChild.sum} - r_2.\text{leftChild.sum} \geq k$ . If this is the case, then return  $F'(r_1.\text{leftChild}, r_2.\text{leftChild}, k)$ . Otherwise, return  $F'(r_1.\text{rightChild}, r_2.\text{rightChild}, k - (r_1.\text{leftChild.sum} - r_2.\text{leftChild.sum}))$ .