Persistence Made Simple - Tutorial

tutorial

datastructure
persistence
tutorial

hikarico #1 June 13, 2017, 6:50pm

This assumes you already have a basic knowledge of segment trees.
For this discussion, I want to show everyone that ***persistence can actually be made simple*** *just by changing our way of thinking.*

Introduction

Usually, persistence is treated as a “really hard” kind of data structure. After all, it’s like a dream data structure with robust version control. It’s the crowd favorite for long challenges because of its fame to as the “hardest to implement”, especially when appended with truly hard concepts in other topics. **In reality, persistence can actually be made simple, as it is literally only one step away from the non-persistent version.**

Background

Why does the idea of a persistent data structure seem so hard for most programmers? I think this is because most of us started with imperative programming languages like C++, Java, and Python, **where changing values of variables is the norm**. We are familiar with having the power to mutate a value anytime we want. In contrast, most of us cringe whenever we deal with immutables. Hail C++ mutable `std::string`, screw Java’s immutable `String`. Nothing beats plain simple assignment operator (`s* = c`) compared to the read-only (`s.charAt(i)`). Most, if not all of us, grew with the mutable mindset. We live and breathe mutability in our code. It is our intuitive art.

Immutable Programming

But when you think about it, persistence is all about staying immutable. Being persistent means we discard our powers of mutability. Just like how Java's `String` class is immutable, we avoid changing individual values directly. Though at first this seems dumb and inefficient, when you think about it it's the straightforward way to do version control. Since original content stays the same, having a new version of the content won't damage the version of the original. For immutables, the way to "change" is to "create". The truth is `persistence == immutability`, hence we can summarize the act of staying persistent with one golden rule.

GOLDEN RULE OF PERSISTENCE

Create new nodes instead of changing them.

That's it? Yes, that's it! The way to be persistent is to *stop using the assignment operator*, and instead focus on recreation of nodes during a change. We *return new nodes* instead of mutating. That's the only additional step we need to transform from mutable to persistent. In REMOVING our capability to be mutable, we force ourselves to CREATE a new way to do version control. In effect, original content can stay the same.

The secret is that there is no secret

Persistence is actually straightforward. There's no need to fancy things around. Just replace all assignment operators with a `new node()` expression or similar. It is no exaggeration to say that it is a mere few lines of code away from the mutable version. The "god data structure" everyone thinks is so complicated is in truth something with *removed powers* than the original. But with this irony of a *removed power*, we instead gain a *new power* called efficient version control.

IMPLEMENTATION

Let's demonstrate an example of adding persistence to a segment tree. Suppose we want to solve the range sum query problem, but with a persistent data structure. We could implement it the usual segment tree way as follows:

Build (usual segment tree)

```
#define M ((L+R)/2)
int st[4*n];
void build(int arr[], int p=1, int L=0, int R=n-1) {
    if (L == R) st[p] = arr[L]; // construct as leaf
    else { // construct as parent
        build(arr, 2*p, L, M);
        build(arr, 2*p+1, M+1, R);
        st[p] = st[2*p] + st[2*p+1];
    }
}
```

Now how do we make this persistent? What we do is that we replace every assignment operator `st[p] = ...` with a `new()` operator. See my example below for the details.

Build (persistent segment tree)

```
#define M ((L+R)/2)
int build(int arr[], int L=0, int R=n-1) {
    if (L == R) return newleaf(arr[L]); // construct as leaf
    else return newparent(build(arr, L, M), build(arr, M+1, R)); // construc
}

// Usage: int root = build(arr, 0, n - 1);
```

We abstract the idea of "change" via "creation", as per the golden rule of persistence. Isn't it really simple? In fact, in some cases, we can achieve shorter code than the mutable version if we do it this way! Of course, this depends on the problem, but it is not wrong to say that it can be achievable.

Creating New Nodes

If you think about it, the difference between the non-persistent and the persistent version is that the former mutates an array called `st[]` while the latter returns a new node at every build. You can implement this a couple of ways. A famous way is to use a node class. But for me, the way I usually implement it is by allocating a pooled array:

```
int l[SIZE], r[SIZE], st[SIZE], NODES = 0;
int newleaf(int value) {
    int p = ++NODES;
    l[p] = r[p] = 0; // null
    st[p] = value;
    return p;
}
int newparent(int lef, int rig) {
    int p = ++NODES;
    l[p] = lef;
    r[p] = rig;
    st[p] = st[lef] + st[rig]; // immediately update value from children
    return p;
}
```

There are two operations, `newleaf()` and `newparent()`. We either construct nodes as a leaf or as a parent, where being a parent already pulls values from the children. In the same way as the mutable version, we have an array called `st[]` that stores the segment tree values depending on our queries. Each node also has pointer to the left and the right children stored at `l[]` and `r[]`

respectively. For the size, I usually allocate around SIZE \approx 8N \lceil log N \rceil. In practice I do this because it's faster than using classes. But of course, it all depends on your groove 🙂

What about updating nodes? When there's no persistence, we usually directly manipulate our segment tree array `st[]` like so:

Point Update (usual segment tree)

```
void update(int i, int x, int p=1, int L=0, int R=n-1) {
    if (L == R) {st[p] = x; return;}
    if (i <= M) update(i, x, 2*p, L, M);
    else update(i, x, 2*p+1, M+1, R);
    st[p] = st[2*p] + st[2*p+1];
}
```

If we want to add persistent update, we simply need to create new nodes via our `newleaf()` or `newparent()` functions instead of the usual assignment operators:

Point Update (persistent segment tree)

```
// return an int, a pointer to the new root of the tree
int update(int i, int x, int p, int L=0, int R=n-1) {
    if (L == R) return newleaf(st[p] + x);
    if (i <= M) return newparent(update(i, x, l[p], L, M), r[p]);
    else        return newparent(l[p], update(i, x, r[p], M + 1, R));
}

// Usage:
// int new_version_root = update(i, x, root);
// Both roots are valid, you can query from both of them!
```

The only change you need is wrapping the change in a new node at every update. If we merely consider persistence as a wrapper, we can make the implementation clean and concise.

Range Copy (persistent segment tree)

(Bonus) My favorite operation on a persistent tree is the range copy. This is the ultimate version control technique one can pull off with regards to reverting a range back to a certain version. Imagine doing a range reset back to initial values - what a breakthrough! Although I find it rare to see problems that need range copy (e.g. OAK), it's one of those operations that can come in handy. Fortunately, for a persistent tree, this can easily be achieved a few lines of code:

```
// revert range [a:b] of p
int rangecopy(int a, int b, int p, int revert, int L=0, int R=n-1) {
    if (b < L || R < a) return p; // keep version
    if (a <= L && R <= b) return revert; // reverted version
    return newparent(rangecopy(a, b, l[p], l[revert], L, M),
                     rangecopy(a, b, r[p], r[revert], M+1, R));
}

// Usage: (revert a range [a:b] back to an old version)
// int reverted_root = rangecopy(a, b, root, old_version_root);
```

We pass in two roots: the current tree root and the old version root. We traverse both side-by-side, and replace only the relevant nodes during our traversal. This meek five-lined function is the foundation of efficient version-control, which is but one of the many operations you can do with a persistent segment tree.

Complexity

In a usual tree, we change at most O(log N) nodes during an update/query. Therefore, for a persistent tree, we create at most O(log N) nodes as well.

Summary

Persistence is a powerful tool, but can also be made simple. After reading this, I hope that you have come to understand persistent data structures in new light. Trust me, the conversion process is really simple since we just convert all assignment operators to `new` leaves or parents.

Persistence is not only for segment trees - you can in fact use this technique for other data structures, such as BBSTs. By understanding persistence more generally, I purposefully omitted range query since I want you to try and code that on your own. By now, I hope you can find it easier to come up with your own code for various kinds of queries, and even for lazy propagation! **Feel free to comment on this thread to open discussion of such techniques.**

As for other resources, you can check out Anudeep's blog or the Wiki page for a more in-depth explanation of persistent trees, explained with problems to solve. With lots of practice, you can be ready hike your way to conquer next month's long challenge. Remember to stay persistent, and happy coding 🙂

UPD: added usage

UPD2: added list of problems in the comments below!

71 Likes

---

dp1priyesh3_3 #2 June 13, 2017, 11:55pm

Very helpful article! You made it simpler than normal segment tree 🙂
Although, I didn't properly understand the rangecopy function. What will be the initial parameter passed to the revert variable of this function?
Also, How is revert variable storing the version of the segment tree? Like suppose we build a RSQ

segment tree for 6 elements then we will have nodes from 1-11 in the segment tree(6 leaf node and 5 parent nodes). Now if I update an element at position 3 (1 indexed) then we will have 3 new nodes which will be part of 2nd version of the segment tree, right? If I again update element at position 5 then we will have 4 new nodes which will be part of 3rd version and so on… how are you managing these versions using revert variable?
Thanks in advance.

1 Like

---

natsudragneel #3 June 14, 2017, 11:07am

This is so well written !

1 Like

---

ksmukta #4 June 14, 2017, 4:45pm

Great tutorial but I need to befriend Recursion, to fully appreciate it. Could anyone please do a 'Recursion Made Simple'. Its often hard to analyse recursive functions, how actually the function produces the desired behaviour.

1 Like

---

hikarico #5 June 14, 2017, 5:02pm

List of Problems

Here's a list of problems to practice on. Enjoy!

Standard

- MKTHNUM - Kth Number
- KQUERYO - K-query Online
- SORTING - Sorting
- DQUERY - D-query
- XRQRS - Xor Queries
- SUBINVER - SubInversing
- FRBSUM - Forbidden Sum
- 786C - Till I Collapse

Path-to-root problems

- GOT - Gao on Tree
- COT - Count on Tree
- PSHTTR - Pishty and Tree

- GPD - Gotham PD
- 226E - Noble Knight's Path

Mixed

- PRMQ - Chef and Prime Queries
- CLONEME - Cloning
- FIBTREE - Fibonacci Numbers on Tree (Math, HLD, *hard*)
- QUERY - Observing the Tree (HLD, *hard*)
- OAK - Persistent Oak (HLD, *hard*)
- 464E - The Classic Problem (Persistent Convex Hull Trick, *hard*)

*This is community wiki, **you can edit**. Feel free to add problems you see fit!* 😃

25 Likes

---

vijju123 #6 June 15, 2017, 7:24pm

Okay, i finished reading this after some more brushing up of the topic, and it literally is one of the best tutorial i ever read. It explains the concept very simply and easily. Also, i like your writing style, its not that monotonous one we find in textbooks, so it was a refreshing read.

For suggestion, We dealt with update function and build function here. Is it possible to take up some sample/standard problem and explain the query function as well? I am in doubt that, since we are creating new node instead of updating the old ones, what change in query function is necessary?

2 Likes

---

saatwik27 #7 June 16, 2017, 4:30am

This problem is based on Persistent Trie

Dexter's Random Generator

1 Like

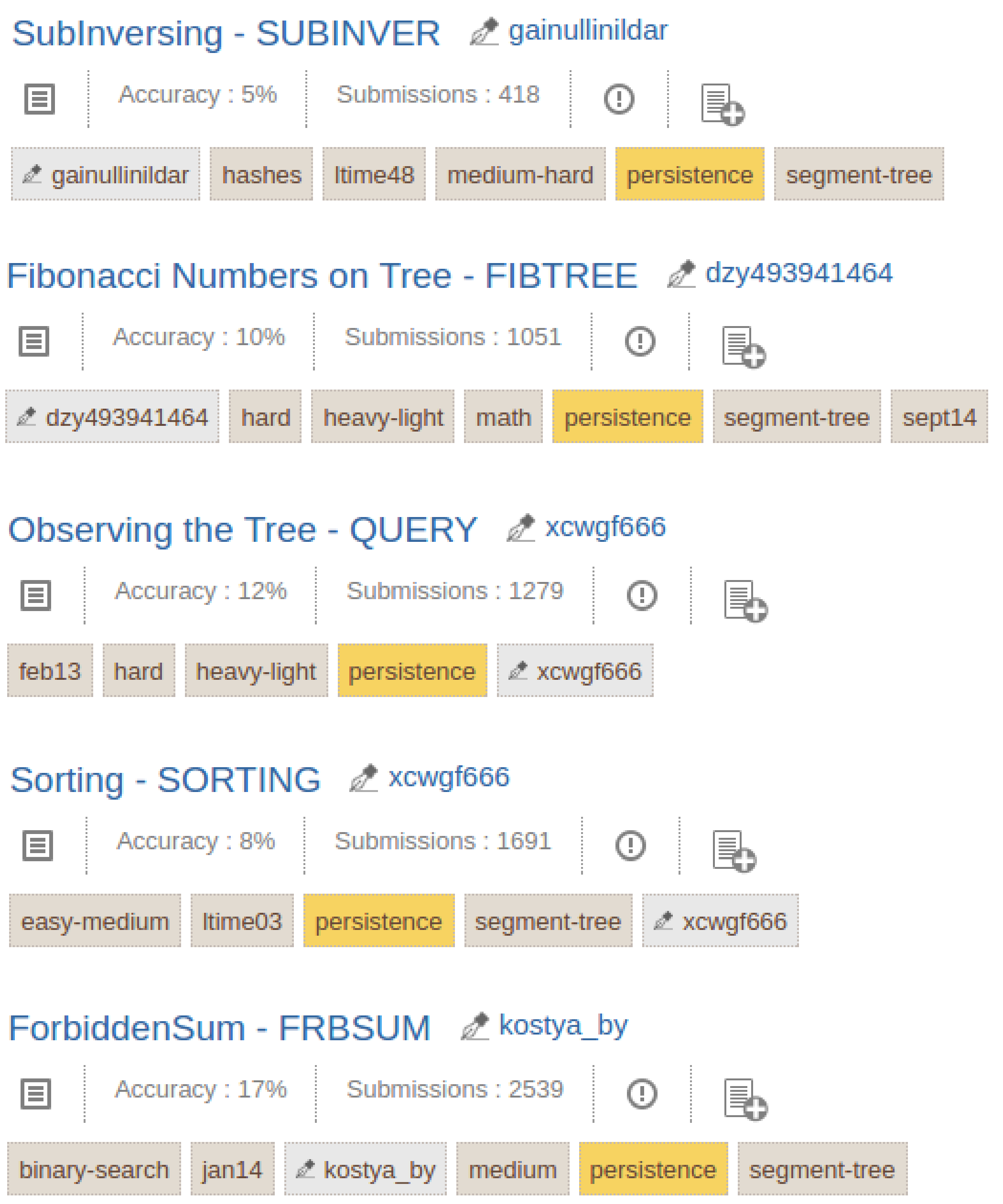
---

skbly7 #8 June 16, 2017, 4:43am

Awesome article upon this topic. Along with it, by seeing other answers, I wish to add.

I don't know if many of us are aware of less advertised tag feature on CodeChef present at https://www.codechef.com/tags.

Ex: These problems are related to persistence directly.
https://www.codechef.com/tags/problems/persistence



SubInversing - SUBINVER gainullinildar

Accuracy : 5% | Submissions : 418

gainullinildar | hashes | ltime48 | medium-hard | persistence | segment-tree

Fibonacci Numbers on Tree - FIBTREE dzy493941464

Accuracy : 10% | Submissions : 1051

dzy493941464 | hard | heavy-light | math | persistence | segment-tree | sept14

Observing the Tree - QUERY xcwgf666

Accuracy : 12% | Submissions : 1279

feb13 | hard | heavy-light | persistence | xcwgf666

Sorting - SORTING xcwgf666

Accuracy : 8% | Submissions : 1691

easy-medium | ltime03 | persistence | segment-tree | xcwgf666

ForbiddenSum - FRBSUM kostya_by

Accuracy : 17% | Submissions : 2539

binary-search | jan14 | kostya_by | medium | persistence | segment-tree

PS: Wrote especially as separate answer, to tell about tags, those weren't knowing about it already. 😃

3 Likes

---

blake_786 #9 June 16, 2017, 9:00pm

Nice tutorial!!! Can you provide the pseudo code in case we are doing a range update using lazy popagation??

2 Likes

syamphanindra #10 June 17, 2017, 12:07am

Nice Tutorial man!!!

2 Likes

hikarico #11 June 17, 2017, 11:16am

This is an answer to @blake_786's question regarding updates with lazy propagation, since my answer would not fit into a comment.

Persistent Lazy Propagation

Lazy propagation is the act of updating by need. To do lazy propagation with a persistent tree, like for regular segment tree, we flag nodes whenever they need to update and subsequently pass them down to the children after visiting. But the difference is, since we are doing things persistently, we create new nodes during update.

The thing is, propagation varies from problem to problem. Range set-value update is different from range increase, range flip, etc. But the template is similar. Let's make two arrays, `hasflag[]` and `flag[]` to mark if a node has a lazy flag or not. Then the code of persistent propagation will look like the following:

```
bool hasflag[]; // if node has a flag (sometimes, you can omit this)
int flag[];     // the actual value of the flag

// returns a new node with a lazy flag
int newlazykid(int node, int delta, int L, int R);

void propagate(int p, int L, int R) {
    if (hasflag[p]) {
        if (L != R) { // spread the laziness
            l[p] = newlazykid(l[p], flag[p], L, M);
            r[p] = newlazykid(r[p], flag[p], M + 1, R);
        }
        // reset flag
        hasflag[p] = false;
    }
}
```

The `propagate()` method spreads the laziness from parent to children. In doing so, we update the values of the children depending on our flag. But if you notice, we want persistence so the way to update this is to create "new lazy kids" during propagation. (Note, we don't need to create a "new non-lazy parent” node after propagation, since it’s the same parent that was just too lazy to update).

Lazy Children

To propagate properly, we need a way to create lazy child nodes. That’s what the `newlazykid()` method is for. I left it blank since the propagation style largely depends on the problem. But by default, the lazy kid should make a new cloned node, update it, then setup its flags. Here’s an example template code:

```
int newlazykid(int node, int delta, int L, int R) {
    int p = ++NODES;
    l[p] = l[node];
    r[p] = r[node];
    flag[p] = flag[node]; // need this since lazy kid might be lazy before
    hasflag[p] = true;

    /* range increase */
    flag[p] += delta;
    st[p] = st[node] + (R - L + 1) * delta;
    /* edit depending on the problem */

    return p;
}
```

After setting up flags, the child needs to know what needs to change based on the flag. It depends on the problem. For example, for range increase the kid can have value `st[node] + (R - L + 1)*delta`. If it’s range set-value we simply make the kid have value `delta`, if it’s range minify the value is `min(st[p], delta)` and so on.

Lazy Range Update/Query

Now that we have propagation, how does the final range update code look like? It’s amazingly succint if you ask me 🙂

```
// range update on [a:b] with value x, on the tree rooted at p

int update(int a, int b, int x, int p, int L=0, int R=n-1) {
    if (b < L || R < a)   return p;
    if (a <= L && R <= b) return newlazykid(p, x, L, R);
    propagate(p, L, R); // always do this before going down
    return newparent(update(a, b, l[p], x, L, M),
                     update(a, b, r[p], x, M + 1, R));
}
```

Now, every time we query down we should also propagate. We’ll achieve very similar code after.

```
// range query on [a:b], on the tree rooted at p

int query(int a, int b, int p, int L=0, int R=n-1) {
    if (b < L || R < a)   return 0;
    if (a <= L && R <= b) return st[p];
    propagate(p, L, R); // always do this before going down
    return query(a, b, l[p], x, L, M) + query(a, b, r[p], x, M + 1, R);
}
```

Hope that was helpful. I'd like to have your thoughts on this.

9 Likes

shuklacoder55 #12 June 17, 2017, 10:08pm

Awesome! I mean how can anyone spend lots of time writing these blogs just for the sake of others … great tutorial …

Well I was trying http://www.spoj.com/problems/GOT/ question on spoj mentioned above …

I am using same approach as explained above but not able to implement it … Can someone help me or if possible send me the link of solution to this problem so that it might help me … Thanks in advance!

1 Like

rishus23 #13 June 18, 2017, 2:16pm

hey @hikarico I'm trying to understand your approach(https://www.codechef.com/viewsolution/14097242) in June17 CLONEME question. You have used persistent hash tree. but I'm not getting your approach from line 133. If the left or right any of them are not equal then we are searching for leftmost or rightmost, but it could be in between also…? did I misunderstand something here?

I dont have enough karma points to reply to your comment on original post.

1 Like

bens59 #14 June 19, 2017, 8:30pm

So well explained.

ganesh5 #15 June 23, 2017, 10:04am

nice tutorial

1 Like

prakhariitd #16 June 27, 2017, 6:34pm

Has anyone solved [DQUERY][1] or [KQUERY][2] using persistent segment tree?

I have done these using segment tree and offline approach.

Edit: done both using persistent segment tree also.
[1]: http://www.spoj.com/problems/DQUERY/
[2]: http://www.spoj.com/problems/KQUERY/

python_rider #17 February 19, 2018, 9:36pm

best tutorial for leaning all about persistent segment tree.

vikram_91 #18 June 1, 2018, 10:44am

One of the best tutorial I ever saw.
Great work.

vic2002 #19 December 27, 2018, 10:21pm

Could somebody give me the implimentation of this problem with this code of persistent seg tree please.I don't know why it gives me wrong answer. This problem

The code

vijju123 #20 June 13, 2017, 6:55pm

Just a basic knowledge of segment trees is enough? I have just started wth this thing (solved Q on minimum in range [l,r] and other basic ones). Will that be enough to understand it? Or do i need to see/practice more before starting this?

---

hikarico #21 June 13, 2017, 7:00pm

Yes, basic knowledge is enough. If you have done RSQ or RMQ, even without lazy propagation, I believe you can understand how persistence works

3 Likes

---

divyansh_gaba7 #22 June 13, 2017, 10:29pm

Amazing article!

1 Like

---

hikarico #23 March 28, 2019, 1:02am

Good question. To revert, you pass the root of an old version. Let's say range set update, sample usage below:

```
int arr[] = {7, 1, 5, 2, 3}; n=5;
int tree0 = build(arr);
int tree1 = update(1, -4, tree0);  // {7, -4, 5, 2, 3}
int tree2 = update(2, -10, tree1); // {7, -4, -10, 2, 3};
int tree3 = update(4, -25, tree2); // {7, -4, -10, 2, -25};

// revert range [0..2] back to initial [tree0]
int tree4 = rangecopy(0, 2, tree3, tree0); // {7, 1, 5, 2, -25}
```

During recursion, the nodes of tree4 inside the range [0:2] will be the nodes from tree0. Hope that answers it ^

2 Likes

---

likecs #24 June 14, 2017, 11:49am

Linking some starting problems will make the amazing tutorial more complete, i guess. examples : MKTHNUM, SORTING etc.

5 Likes

dp1priyesh3_3 #25 March 28, 2019, 1:02am

Yes, that answers my question. Thanks a lot.

1 Like

hikarico #26 March 28, 2019, 1:02am

This is great! Will add these problems to the list

hikarico #27 March 28, 2019, 1:02am

It’s the same as segment tree. We’re not really creating new nodes during query (except when lazy), so it’s the same code. Except that instead of passing 2*p and 2*p + 1, we pass l[p] and r[p] during recursion:

```
// Example RSQ, call sum_query(a, b, root)

int sum_query(int a, int b, int p, int L=0, int R=n-1) {
    if (b < L || R < a) return 0; // outside range
    if (a <= L && R <= b) return st[p]; // inside range
    return sum_query(a, b, l[p], L, M)
         + sum_query(a, b, r[p], M + 1, R);
}
```

1 Like

vijju123 #28 March 28, 2019, 1:02am

Thanks!! I will re-attempt the questions with this in mind (got stuck at thinking part) ^^

vijju123 #29 March 28, 2019, 1:02am

Very nice ^^

bhishma #30 March 28, 2019, 1:02am

Another problem

hikarico #31 March 28, 2019, 1:02am

Thanks! See my other answer for lazy propagation above link

hikarico #32 March 28, 2019, 1:02am

Let's say I want to compare if array `[1, 2, 2, 3, 4]` is similar to `[1, 2, 3, 3, 4]`. My trees represent hashing of frequency tables `{1:1, 2:2, 3:1, 4:1}` and `{1:1, 2:1, 3:2, 4:1}`

```
  Tree 1              Tree 2
   [1, 2, 1, 1]         [1, 1, 2, 1]
     /      \             /      \
  [1, 2]  [1, 1]      [1, 1]  [2, 1]
   /  \    /  \        /  \    /  \
 ...                  ...
```

Look at second layer, there's a special case where both left and right have mismatch, but the result is still "similar". This can be solved by comparing rightmost and leftmost in both trees.

hikarico #33 March 28, 2019, 1:02am

You'll need to do LCA with jumping pointers. You have `tree[a]` contain all nodes from root to a (you can use range sum). Then to find out if c is in the path from a to b, you check if `query(c, tree[a]) + query(c, tree**) - query(c, tree[parent(LCA(a, b))]) == 1` with principle of inclusion-exclusion 🙂

rishus23 #34 March 28, 2019, 1:02am

Thanks mate, It is clear now. I'm gonna write a small post on persistent hash tree.

1 Like

---

shuklacoder55 #35 March 28, 2019, 1:02am

Great ! thanks 🙂

---

adecemberguy #36 June 27, 2017, 10:15am

So when we are building PST(initially) we always build a copy of original SegT ? (As we are doing in `build()` function by calling `newleaf()` and `newparent()`)

---

hikarico #37 June 27, 2017, 10:50am

By convention yes, if you have an initial array. But if tree is empty at the start, you can make also do without build by assigning tree as null (`0`) at the start, and do point updates one by one (e.g. path-to-root problems)

---

adecemberguy #38 June 27, 2017, 11:42am

and `st[]` here stores the sum of elements in pre-order fashion, not conventional parent to child `[2*i,2*i+1]` way, does it make things easy ?

---

prakhariitd #39 March 28, 2019, 1:02am

KQUERY can't be solved using persistent segment tree? I am getting TLE.

2 Likes

---

hruday968 #40 March 28, 2019, 1:02am

hey i have done DQUERY using MO's algorithm,but i am unable to solve that using persistent segment tree.can you please share your code?

---

hikarico #41 March 28, 2019, 1:02am

SPOJ set the time limit really strict for KQUERY, they specifically wanted offline solution for that with a faster data structure (e.g. fenwick). That's why they created KQUERYO as a separate online problem.

2 Likes

---

hikarico #42 March 28, 2019, 1:02am

I assume you did DQUERY by sorting queries by right endpoint? Then you can do the same technique using persistent tree. Same technique, but during build you store the tree at index R as tree[R]. You can now solve queries online, just call tree[R].range_query(L, R)

For KQUERY, the TLE is strict there so only offline works (cuz SPOJ)… but there's an online counterpart KQUERYO, you can solve that with persistent

---

karangreat234 #43 April 29, 2019, 7:16pm

Bookmarked!! :-)))))

1 Like