added: 11 Jun 2008 11:00
Edited: 25 Oct 2011 21:31

Segment tree

Segment tree - a data structure that allows efficient (ie the asymptotic behavior $O(\log n)$) to implement the following operations: finding the amount / minimum of array elements in a given interval ($a[l \dots r]$where $l$and $r$are input to the algorithm), while additionally possible to change elements of the array: how to change the values of a single element, and change elements on a whole array of subsegments (ie, all the elements are allowed to assign $a[l \dots r]$a value, or to add to all the elements of the array any number).

Generally, the segment tree - a very flexible structure and the number of problems solved her theoretically unlimited. In addition to the above types of transactions with trees segments, are also possible and much more complex operations (see "Complicated version tree segments"). In particular, the segment tree is easily generalized to higher dimensions: for example, to solve the problem of finding the amount / minimum in a given matrix podpryamougolnike (but only for the time already $O(\log^2 n)$).

An important feature of tree segments is that they consume a linear memory: standard segment tree requires about $4n$memory elements to work on the array size n.

Contents [hide]

Description of the tree segments in the base case

First, consider the simplest case of tree segments - segment tree for sums. If you pose the problem formally, then we have an array $a[0..n-1]$, and our segment tree should be able to find sum of elements from lth to rth (this request amounts) and also handle value change of a specified element of the array actually respond to the assignment $a[i] = x$(this modification request). Once again, the segment tree must handle both during query $O(\log n)$.

Tree Structure Segments

So, what is a segment tree?

We calculate and remember somewhere sum of all elements of the array, ie segment $a[0 \dots n-1]$. Also calculate the amount of two halves of the array: $a[0 \dots n/2]$and $a[n/2+1 \dots n-1]$. Each of these two halves, in turn, divide in half, calculate and store the sum at them, then divide in half again, and so on until it reaches the current segment length 1. In other words, we start with the segment $[0; n-1]$and each time we divide the current segment in half (if it has not yet become a segment of unit length), then calling the same procedure on both halves, and for each such segment we store the sum of numbers on it.

We can say that these segments in which we considered the sum, form a tree: the root of the tree - the segment $[0 \dots n-1]$, and each vertex has exactly two sons (except vertex leaf, which has a length of segment 1). Hence the name - "segment tree" (although the implementation is usually no tree is clearly not built, but on this below in the implementation section).

Now that we have a tree structure segments. Immediately, we note that it has a **linear dimension** , namely, contains less $2n$vertices. This can be understood as follows: the first level of the tree contains a vertex segments (segment $[0 \dots n-1]$), the second level - in the worst case, the two peaks on the third level in the worst case will be four vertices, and so on until it reaches the number of vertices n. Thus, the number of vertices in the worst case estimated amount $n + n/2 + n/4 + n/8 + \dots + 1 < 2n$.

It is worth noting that the $n$non-power of two, not all levels of the tree segments are completely filled. For example, if $n = 3$the left son of the root is a segment $[0 \dots 1]$having two children, while the right child of the root - cut $[2 \dots 2]$, is a leaf. No special difficulties in implementing it is not, but nevertheless it must be borne in mind.

The height of the tree is the value segments $O(\log n)$- for example, because the length of the segment is at the root of the tree n, and when you go to one level down the length of the segments is reduced by about half.

Construction

The process of constructing the tree segments for a given array $a$can be done efficiently as follows, from the bottom up: first, record the values of the elements $a[i]$in the corresponding leaves of the tree, and then calculate them on the basis of values for the vertices of the previous level as the sum of the values in two leaves, then similarly calculate values for more than one levels, etc. Is convenient to describe this operation recursively: we run the procedure of construction of the root segments, and the procedure of constructing, if not

caused by the sheet calls itself from each of two sons and summarizes the calculated values, and if it is caused by a sheet - simply writes a value of the array element.

Asymptotics tree construction segments will be so $O(n)$.

Request amount

We now consider the amount of the request. Are input two numbers $l$and r, and we have a time $O(\log n)$to calculate the sum of the segment $a[l \ldots r]$.

To do this, we will go down the tree segments constructed using to calculate the amount previously computed response at each node of the tree. Initially, we get to the root of the tree segments. Let's see in which of his two sons misses cut query $[l \ldots r]$(recall that the sons of the root segments - it stretches $[0 \ldots n/2]$and $[n/2+1 \ldots n-1]$). There are two possibilities: that the segment $[l \ldots r]$gets only one son in the root, and that, conversely, the segment intersects with two sons.

The first case is simple: just go to that son, in which lies our segment query and apply the algorithm described here to the current node.

In the second case, we no other options but to go first to the left child and count prompted it, and then - go to the right child, calculate the answer in it and add to our response. In other words, if the left son represented the segment $[l_1 \ldots r_1]$, and the right - segment $[l_2 \ldots r_2]$(note that $l_2 = r_1 + 1$), then we go to the left child with the request $[l \ldots r_1]$, and the right - with the request $[l_2 \ldots r]$.

Thus, the sum of query processing is a **recursive function** that calls itself whenever either the left child or from the right (without changing the query boundaries in both cases), or by both at once (at the same time sharing our inquiry into two corresponding subquery). However, recursive calls are not always going to do: if the current request coincided with the boundaries of the segment in the current top of the tree segments, the response will return the precomputed value of the sum in this segment, recorded in the tree segments.

In other words, the query evaluation is descending a tree segments, which extends all the necessary branches of the tree, and for quick work using already computed the amount in each segment in the tree segments.

Why **the asymptotic behavior** of the algorithm is $O(\log n)$? To do this, look at each level of the tree segments as maximum lengths could visit our recursive function when processing a request. It is argued that these segments could not be more than four, then, using the estimate $O(\log n)$for the height of the tree, we obtain the desired asymptotic behavior of the algorithm time.

We show that the evaluation of the four segments is correct. In fact, at the zero level of the tree is affected only request top - the root of the tree. Next on the first level recursive call in the worst case is split into two recursive calls, but it is important here is that the questions in these two calls will coexist, ie by l''request in the second recursive call is one more than the number of $r'$the request in the first recursive call. This implies that at the next level, each of these two calls could produce two more recursive calls, but in this case, half of the non-recursive queries will work, taking the desired value from the top of the tree segments. Thus, every time we will have no more than two branches of recursion really working (we can say that one branch close to the left border of the query, and the second branch - to the right), and total number of affected segments can not exceed the height of the tree segments, multiplied by four, i.e. it is the number $O(\log n)$.

Finally, you can lead an understanding of the query sum: the input section $[l \ldots r]$is divided into several subsegments, the answer to each of which is calculated and stored in the tree. If you do it the right way partition, thanks to the tree structure of segments required by subsegments will always be $O(\log n)$, which gives the efficiency of the tree segments.

Update request

Recall that the update request is received at the input index $i$value and x, rebuilds the tree and the segments in such a manner as to conform to the new value $a[i] = x$. This database should also be performed during $O(\log n)$.

It is more simple compared to the query request amount counting. The fact that the element $a[i]$involves only a relatively small number of vertices of the segments: namely, at $O(\log n)$the vertices - one on each level.

Then it is clear that the update request can be implemented as a recursive function: it is passed the current node of the tree lines, and this function performs a recursive call from one of his two sons (from that which contains the position $i$in its segment), and after that - recalculates the sum in the current node in the same way as we did in the construction of the tree segments (ie, the sum of the values for both sons of current node).

Implementation

Realizable main point - is that, as **stored** in the memory segment tree. For simplicity, we will not store a tree in an explicit form, and use this trick: we say that the root of the tree has a number of 1his sons - the rooms $2$and 3their sons - rooms 4on 7, and so on. Easy to understand the correctness of the following formula: if the node has a room i, then let her son left - is the pinnacle with a number $2i$, and the right - with the room $2i+1$.

This technique greatly simplifies the programming tree segments - now we do not need to be stored in the memory tree structure of segments, but only to have a array of amounts on each segment of the tree segments.

One has only to note that the size of the array in such a numbering is not necessary to put $2n$and $4n$. The fact that such a numbering is not working perfectly when $n$not a power of two - then there are missed calls, which do not correspond to any vertex of the tree (actually, numbering behaves just as if it $n$would be rounded up to the nearest power of two) . This does not pose any difficulties in the implementation, however leads to the fact that it is necessary to increase the size of the array to $4n$.

So segment tree we **keep** just as an array $t[]$, the size is more than four times the size of $n$the input data:

```
int n, t[4*MAXN];
```

The procedure for **constructing the tree of segments** for a given array $a[]$is as follows: a recursive function, passing it the array $a[]$, the number $v$of the current top of the tree, and the boundaries $tl$and $tr$the segment corresponding to the current top of the tree. From the main program should call this function with parameters $v = 1, tl = 0, tr = n - 1$.

```
void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = a[tl];
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = t[v*2] + t[v*2+1];
        }
}
```

Further, the function to **query the amount** also represents a recursive function in the same way that information is transmitted to the current top of the tree (ie, the number v, tl, trwhich in the main program should pass $1, 0, n - 1$respectively), and in addition - as border $l$and $r$the current request. In order to simplify this code fukntsii always makes two recursive calls, even if actually need one - just extra recursive call request be passed, which $l > r$it is easy to cut off an additional check in the beginning of the function.

```
int sum (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return sum (v*2, tl, tm, l, min(r,tm))
                + sum (v*2+1, tm+1, tr, max(l,tm+1), r);
}
```

Finally, **a modification request** . He just passed the information about the current top of the tree segments, and further, the index changing element, as well as its new value.

```
void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = new_val;
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = t[v*2] + t[v*2+1];
        }
}
```

It should be noted that the function $\mathrm{update}$is easy to make non-recursive because it tail recursion, ie branching never happens: one call can produce only one recursive call. When non-recursive implementation, speed can increase by several times.

Among other **optimizations** worth mentioning that multiplying and dividing by two is necessary to replace Boolean operations - it also slightly improves the performance of the tree segments.

Sophisticated version of the tree segments

Segment tree - a very flexible structure, and allows you to make generalizations in many different directions. Try to classify them below.

More complex functions and queries

Improve tree segments in this direction can be as pretty obvious (as in the case of the minimum / maximum instead of the sum), and a spectacular nontrivial.

Minimum / maximum

Little to change the conditions of the problem described above: instead of requesting a sum will produce now request the minimum / maximum on the interval.

Then the tree segments for this practically does not differ from the tree segments described above. Just need to change the method of

Then the tree segments for this practically does not differ from the tree segments described above. Just need to change the method of calculating $t[v]$ functions in build and update, as well as the calculation of the returned response function sum (to replace the summation by minimum / maximum).

Minimum / maximum value and the number of times it occurs

Problem is similar to the previous one, but now in addition to the maximum amount is also required to return it occurs. This problem arises in a natural way, for example, when using the decision tree segments the following problem: find the number of the longest increasing subsequence in a given array.

To solve this problem, each node of the tree segments will store a pair of numbers: in addition to the maximum number of its occurrences in the relevant segment. Then the construction of the tree we have just two such pairs obtained from the sons of the current node, get a pair for the current node.

Combining two such pairs in the same worth as a separate function because this operation will need to produce and modify the query, and the query search for the maximum.

```
pair<int,int> t[4*MAXN];

pair<int,int> combine (pair<int,int> a, pair<int,int> b) {
        if (a.first > b.first)
                return a;
        if (b.first > a.first)
                return b;
        return make_pair (a.first, a.second + b.second);
}

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = make_pair (a[tl], 1);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

pair<int,int> get_max (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return make_pair (-INF, 0);
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return combine (
                get_max (v*2, tl, tm, l, min(r,tm)),
                get_max (v*2+1, tm+1, tr, max(l,tm+1), r)
        );
}

void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = make_pair (new_val, 1);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

Find the greatest common divisor / least common multiple

le we want to learn to look for GCD / LCM of all the numbers in a given segment of the array.

It's pretty interesting generalization tree lengths obtained in exactly the same way as for the amount of trees segments / minimum / maximum: simply stored in each node of the tree GCD / LCM of all the numbers in the corresponding segment of the array.

Count the number of zeros, search kth zero

In this task, we want to learn how to respond to your request the number of zeros in a given segment of the array, as well as the request of finding $k$the first zero element.

Again slightly modify the data stored in the tree segments: we keep the array is now $t[]$the number of zeros occurring in the corresponding segments of the array. It is clear, how to maintain and use the data functions build, sum, update, - thus we solved the

problem of the number of zeros in a given segment of the array.

Now learn how to solve the problem of finding the position of $k$the first occurrence of zero in the array. To do this, we will go down the tree segments, starting from the root, and going every once in left or right child depending on which of the segments is required kth zero. In fact, to understand what we need to move on son, just look at the value stored in the left son if it is greater than or equal to k, the need to move in the left son (because it has at least a segment $k$of zeros), but otherwise - to move in the right child.

In the case of implementation can be cut off when the $k$zero-th exists even when the function returned as a response, for example -1.

```
int find_kth (int v, int tl, int tr, int k) {
        if (k > t[v])
                return -1;
        if (tl == tr)
                return tl;
        int tm = (tl + tr) / 2;
        if (t[v*2] >= k)
                return find_kth (v*2, tl, tm, k);
        else
                return find_kth (v*2+1, tm+1, tr, k - t[v*2]);
}
```

Search prefix array with a given sum

Problem is this: for a given value requires $x$quickly find such $i$that the sum of the first $i$element of the array $a[]$is greater than or equal to x(assuming that the array $a[]$contains only non-negative integers).

This problem can be solved using binary search, calculating each time inside the sum on a particular prefix array, but it will lead to a solution for the time $O(\log^2 n)$.

Instead, you can use the same idea as in the previous paragraph, and look for the desired position of a descent on a tree, turning every once in a left or right child depending on the value of the sum in the left son. Then the answer to the search request will be one such descending a tree, and, hence, will run for $O(\log n)$.

Search subsegments with a maximum amount

Still given to the input array $a[0 \dots n-1]$, and there are questions (l, r), which mean: find a subsegment $a[l' \dots r']$that $l \le l'$, $r' \le r$and the amount of this segment $a[l' \dots r']$is maximized. Requests modification of individual elements of the array are allowed. Array elements can be negative (and, for example, if all the numbers are negative, the optimal subsegments will be empty - it sum is zero).

This is a very nontrivial generalization tree segments obtained as follows. Will be stored in each node of the tree segments four values: the amount in this segment, the maximum amount of all the prefixes of this segment, the maximum amount of all suffixes, as well as the maximum amount of subsegments on it. In other words, for each segment of the tree segments the answer to it already predposchitan and additionally answer counted among all the segments that are limited in the left margin of the segment, as well as among all the segments that are limited in the right border.

How do you build tree to such data? Again we come to this point of view with recursive let for the current top four values in the left son and son in law already counted, count them now for the summit. Note that the answer is in the very top:

- or response in the left son, which means that the best subsegment in the current vertex is placed entirely in the segment of the left son
- or answer in the right son, that means that the best subsegment in the current vertex is placed entirely in the segment right child,
- or maximum amount of the suffix in the left son and the maximum prefix in the right son, that means that the best subsegment is its origin in the left son, and the end - in the right.

Hence, in response to the current vertex is the maximum of these three values. Recalculate the maximum amount on the same prefix and suffix even easier. We present the implementation of functions $\mathrm{combine}$, which will be passed two structures data, inclusive of the left and right of the sons, and which returns the data in the current vertex.

```
struct data {
        int sum, pref, suff, ans;
};

data combine (data l, data r) {
        data res;
        res.sum = l.sum + r.sum;
        res.pref = max (l.pref, l.sum + r.pref);
        res.suff = max (r.suff, r.sum + l.suff);
        res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
        return res;
}
```

So we learned how to build a tree segments. Is easy to obtain and implement modification request: as in the simple tree segments, we recalculate the values in the tree tops all changed segments, which all use the same function $\mathrm{combine}$. To calculate the values of tree leaves as a helper function $\mathrm{make_data}$that returns a structure data, the calculated one number val.

```
data make_data (int val) {
        data res;
        res.sum = val;
        res.pref = res.suff = res.ans = max (0, val);
        return res;
}

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = make_data (a[tl]);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = make_data (new_val);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

It remains to deal with the response to the request. To do this, we have also, as before, down the tree, breaking thus the segment query $[l \dots r]$into several subsegments, coincides with the segment tree segments, and combine the answers to them in a single answer to the whole problem. Then it is clear that the work is no different from ordinary wood work segments, instead of just need a simple summation / minimum / maximum values to use the function $\mathrm{combine}$. The following implementation is slightly different from the implementation of the query sum: it does not allow for cases when the left boundary of $l$the query exceeds the right boundary r (otherwise you will experience unpleasant events - what structure datacomes back when the query interval is empty? ..).

```
data query (int v, int tl, int tr, int l, int r) {
        if (l == tl && tr == r)
                return t[v];
        int tm = (tl + tr) / 2;
        if (r <= tm)
                return query (v*2, tl, tm, l, r);
        if (l > tm)
                return query (v*2+1, tm+1, tr, l, r);
        return combine (
                query (v*2, tl, tm, l, tm),
                query (v*2+1, tm+1, tr, tm+1, r)
        );
}
```

Preserve all the subarray in each node of the tree segments

This is a separate subsection standing apart from the rest because each node of the tree segments we will not store any information about this compressed subsegments (sum, minimum, maximum, etc.), and **all** elements of the array lying in the subsegments. Thus, the root of the tree segments will store all elements of the array, the left son of the root - the first half of the array, right child of the root - the second half, and so on.

The easiest option of this technique - where each node of the tree segments stored sorted list of all numbers appearing in the corresponding interval. In more complex embodiments, the lists are not stored and any data structures constructed on these lists (set, mapetc.). But all these methods have in common is that each node of the tree segments stored some data structure in memory having a size of about the length of the corresponding segment.

The first natural question is posed when considering trees segments of this class - this is **the amount of memory consumed** . It is argued that if each node of the tree stores a list of all the segments appearing on this segment numbers, or any other data structure size of the same order, the sum of all tree segments will occupy $O(n \log n)$memory. Why is this so? Because each number $a[i]$falls into the $O(\log n)$wood segments segments (at least because there are segments of tree height $O(\log n)$).

Thus, despite the apparent extravagance of the tree lines, it consumes memory is not much longer than normal wood segments.

Below described are some typical uses such a data structure. It is worth noting immediately clear analogy trees segments of this type with **two-dimensional data structures** (actually, in a sense, this is a two-dimensional data structure, but rather disabilities).

Find the smallest number greater than or equal to the specified in this interval. No modification requests

Required to respond to requests from the following: (l, r, x)that is to find the minimum number in the interval $a[l \dots r]$that is greater than or equal to x.

Construct a segment tree, in which each node will store the sorted list of all the numbers appearing on the corresponding interval. For example, the root will contain an array $a[]$in sorted order. How to build a segment tree as efficiently? For this approach the task, as usual, in terms of recursion: let the left and right children of the current node, these lists have already been constructed, and we need to build this list for the current vertex. In this formulation, the question becomes almost obvious that this can be done in linear time: we just need to merge two sorted lists into one that is done in one pass on them with two pointers. C + + users even easier, because the merging algorithm is already included in the standard library STL:

```
vector<int> t[4*MAXN];

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = vector<int> (1, a[tl]);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                merge (t[v*2].begin(), t[v*2].end(), t[v*2+1].begin(), t[v*2+1].end(),
                        back_inserter (t[v]));
        }
}
```

We already know that so constructed segment tree will occupy $O(n \log n)$memory. And with such an implementation time of its construction also has value $O(n \log n)$- because each list is constructed in linear time with respect to its size. (Incidentally, there is an obvious analogy here with the algorithm **merge sort** : only here we store the information from all stages of the algorithm, not just the outcome.)

Now consider the **response to the inquiry** . Will descend on the tree, as it makes a standard response to a request in the tree segments, breaking our segment $a[l \dots r]$into several subsegments (some $O(\log n)$units). It is clear that the answer to the whole problem is the minimum of the responses to each of these subsegments. Understand now how to respond to a request for one such subsegments, coinciding with a vertex of the tree.

So, we came to the top of some tree segments and want to compute the answer to it, ie find the smallest number greater than or equal to this x. For this we just need to perform a **binary search** on the list, considered in the top of the tree, and return the first number on the list, more than or equal to x.

Thus, the answer to a query on one subsegments occurs $O(\log n)$, and the whole query is processed in time $O(\log^2 n)$.

```
int query (int v, int tl, int tr, int l, int r, int x) {
        if (l > r)
                return INF;
        if (l == tl && tr == r) {
                vector<int>::iterator pos = lower_bound (t[v].begin(), t[v].end(), x);
                if (pos != t[v].end())
                        return *pos;

                return INF;
        }
        int tm = (tl + tr) / 2;
        return min (
                query (v*2, tl, tm, l, min(r,tm), x),
                query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
        );
}
```

Constant INFequal to some large number, certainly more than any number in the array. It carries the meaning of "response in a given interval does not exist."

Find the smallest number greater than or equal to the specified in this interval. Allowed modification requests

Problem is similar to the previous one, only now allowed modification requests: assignment process $a[i] = y$.

The solution is also similar to the preceding problem, but instead of a simple list in each node of the tree segments we store a balanced list, which allows you to quickly search for the desired number, remove it and insert a new number. Given that the general number of the input array can be repeated, the best choice is the STL data structure $\mathrm{multiset}$.

Constructing such a tree segments occurs approximately the same as in the previous problem, but now we need to unite not sorted lists, and $\mathrm{multiset}$that will lead to the fact that the asymptotic behavior of building deteriorate to $n \log^2 n$(although, apparently, red-black trees allow to merge two trees in linear time, but the STL does not guarantee it).

Response to **a search request** has practically equivalent to the code above, but now $\mathrm{lower_bound}$need to call from $t[v]$.

Finally, **a modification request** . To handle it, we have to go down the tree, making changes to all $O(\log n)$lists containing affect items. We simply remove the old value of this element (not forgetting that we do not need to remove them all together with the number of repetitions) and insert the new value.

```
void update (int v, int tl, int tr, int pos, int new_val) {
        t[v].erase (t[v].find (a[pos]));
        t[v].insert (new_val);
        if (tl != tr) {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
        }
        else
                a[pos] = new_val;
}
```

Treatment of this query is also a time $O(\log^2 n)$.

Find the smallest number greater than or equal to the specified in this interval. Acceleration using the technique of "partial cascading"

Improve response time to the search time $O(\log n)$by applying the technique **of "partial cascading"** ("fractional Cascading").

Partial cascading - is a simple technique that helps to improve the work of several binary searches being conducted by the same value. In fact, the response to the search request is that we divide our task into several subtasks, each of which is then solved for the number of binary search x. Partial cascading allows to replace all of these binary searches on one.

The simplest and most obvious example of a partial cascading is **the following problem** : There are several sorted lists of numbers, and we have in each list to find the first number is greater than or equal to the specified value.

If we solved the problem of "head" that would have to run a binary search on each of these lists, if a lot of these lists, it becomes a very important factor: if the entire list k, then the asymptotic behavior happens $O(k \log(n/k))$where n- the total size of all lists (asymptotic behavior is because the worst case - when all the lists are approximately equal in length to each other, ie equal n/k).

Instead, we could combine all these lists into one sorted list, where each number $n_i$will keep a list of positions: first position in the list of the first number is greater than or equal to n_ia similar position in the second list, and so on. In other words, for each number we keep occurring at the same time the number of results a binary search on it in each of the lists. In this case, the asymptotic behavior of query response is obtained $O(\log n + k)$, which is much better, but we are forced to pay large memory consumption: namely, we need $O(nk)$memory.

Tech partial cascading going on in this task and achieves memory consumption $O(n)$at the same time respond to the request $O(\log n + k)$. (To do this, we do not store the length of one big list n, and again return to the $k$list, but with each list is stored every second element from the list, we will again have the number with each record its position in both lists (current and next), but it will continue to effectively respond to a request: we do a binary search on the first list, and then go on these lists in order, moving each time the next list using predposchitannyh pointers, and taking one step to the left, thereby taking into account that half Numbers following list was not taken into account).

But we in our application to the tree segments **do not need** the full power of this technology. The fact that the list contains the current node to all the numbers which can occur in the left and right sons. Therefore, to avoid a binary search through the list of his son, it is sufficient for each list in the tree segments for each count of the number of its position in the list of the left and right sons (more precisely, the position of the first number is less than or equal to the current).

Thus, instead of a simple list of all the numbers we store a list of triples: the number itself, the position in the list left child, the right position in the list of his son. This will allow us in $O(1)$a position to determine the list left or right son, instead of doing a binary list on it.

The easiest way to apply this technique to the problem when no modification requests - then these positions are just numbers, and count them in the construction of the tree very easily inside the algorithm merge two sorted sequences.

In the case of modification requests are allowed, all a bit more complicated: these positions now be stored in the form of iterators inside $\mathrm{multiset}$, and when you request an update - the right to reduce / increase for those elements for which it is required.

Anyway, the problem is reduced to net realizable subtleties, but the basic idea - replacing $O(\log n)$a binary search binary search on the list in the root of the tree - fully described.

Other possible

Note that this technique implies a whole class of possible applications - everything is determined by the structure of the data selected for storage in each node of the tree. We have examined the application using $\mathrm{vector}$and $\mathrm{multiset}$, while in general you can use any other compact data structure: other tree segments (about this little discussed below in the section on multi-dimensional index tree), tree Fenwick , a Cartesian tree , etc.

Update on the interval

We have considered only the problem when a modification request affects only element of the array. In fact, the segment tree allows

queries that apply to entire segments of contiguous elements, and on these requests in the same time $O(\log n)$.

Addition to the interval

We begin our consideration of trees such segments with the simplest case: a modification request is the addition of all the numbers in some subsegments $a[l \dots r]$of a number x. Read request - still reads the value of a number $a[i]$.

To make a request for adding effectively will be stored in each node of the tree segments as necessary to add all the numbers of this segment as a whole. For example, if a request comes in, "add to the entire array $a[0 \dots n-1]$by 2 ", we will deliver in the root of the tree number 2. Thus we will be able to process a request for the addition of any subsegments efficiently, rather than changing all $O(n)$ values.

Now, if a request comes in to read the value of a number, it is sufficient to go down the tree, summing all encountered on the way the values written in the tree tops.

```
void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = a[tl];
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
        }
}

void update (int v, int tl, int tr, int l, int r, int add) {
        if (l > r)
                return;
        if (l == tl && tr == r)
                t[v] += add;
        else {
                int tm = (tl + tr) / 2;
                update (v*2, tl, tm, l, min(r,tm), add);
                update (v*2+1, tm+1, tr, max(l,tm+1), r, add);
        }
}

int get (int v, int tl, int tr, int pos) {
        if (tl == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return t[v] + get (v*2, tl, tm, pos);
        else
                return t[v] + get (v*2+1, tm+1, tr, pos);
}
```

Assigning the interval

Suppose now that a modification request is assigned to all the elements of a certain length of $a[l \dots r]$a certain value p. As a second request will be considered read array values $a[i]$.

To make a modification on the whole segment have in each node of the tree segments store, painted this segment entirely in any number or not (and if painted, the store itself is a number). This will allow us to make **"retarded" Update** tree segments: the modification request, we, instead of the values in the tree tops in a variety of segments, will change only some of them, leaving the flags "painted" for other segments, which means that this whole segment together with their subsegments should be painted in this color.

So, after making a request modifications segment tree becomes, generally speaking, irrelevant - it remained Shortfall some modifications.

For example, if the request came modification "to assign the entire array $a[0 \dots n-1]$a number ", the tree sections we will only change - label the root of the tree that he painted entirely in that number. The rest of the top of the tree will remain unchanged, but in fact the entire tree must be painted in the same number.

Suppose now that in the same tree segments came second modification request - to paint the first half of the array $a[0 \dots n/2]$at any other number. To handle such a request, we need to paint the entire left child of the root in this new color, but before we do that, we must deal with the root of the tree. The subtlety here is that the tree should be preserved, that the right half is painted in the old color, but at the moment no information in the tree for the right half was not saved.

Output is: make **pushing** information from the root, ie if the root of the tree was painted in any number, then paint in the number of its right and left his son, and from the root to remove this mark. After that, we can safely paint left child of the root, without losing any necessary information.

Summarizing, we obtain for any queries with tree (modification request or reading) while descending the tree, we should always do push information from the current node in both of her sons. You can understand it so that when descending the tree, we use the retarded modification, but only as much as needed (not to worsen with the asymptotic behavior $O(\log n)$).

When implemented, this means that we need to make a function pushwhich will be transferred to tree top segments, and it will produce pushing information from this vertex in both her sons. Should call this function at the beginning of request processing functions (but not call it from the leaves, because of the push plate information is not necessary, and nowhere).

```
void push (int v) {
        if (t[v] != -1) {
                t[v*2] = t[v*2+1] = t[v];
                t[v] = -1;
        }
}

void update (int v, int tl, int tr, int l, int r, int color) {
        if (l > r)
                return;
        if (l == tl && tr == r)
                t[v] = color;
        else {
                push (v);
                int tm = (tl + tr) / 2;
                update (v*2, tl, tm, l, min(r,tm), color);
                update (v*2+1, tm+1, tr, max(l,tm+1), r, color);
        }
}


int get (int v, int tl, int tr, int pos) {
        if (tl == tr)
                return t[v];
        push (v);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return get (v*2, tl, tm, pos);
        else
                return get (v*2+1, tm+1, tr, pos);
}
```

Function getcould be implemented in another way: do it delayed updates, and immediately return a response as soon as it hits the top of the tree segments, entirely painted in a particular color.

The addition of the interval, the maximum request

Suppose now that the modification request will again request the addition of all subsegments including some of the same number and a request is to find the maximum reading in some subsegments.

Then at each node of the tree segments will have to additionally store a maximum of all these subsegments. But subtlety here is how to recalculate these values.

For example, suppose there was a request "added to the entire first half, i.e. $a[0 \dots n/2]$, the number 2. " Then it will be reflected in the tree record the number 2in the left child of the root. How now calculate the new value of the maximum in the left son and the root? Here it becomes important not to get confused - what the maximum is stored in the tree top: maximum without adding on top of all this, or considering it. You can choose any of these approaches, but most importantly - consistently use it everywhere. For example, when you first approach a maximum at the root will be obtained as the maximum of two numbers, a maximum in the left son, plus the addition of the left son, and the son of a maximum in the right plus an addition to it. In the second approach, the same maximum in the root will be obtained as an addition to the root plus a maximum of maxima in the left and right sons.

Other Destinations

Here were considered only basic application segments trees in problems with the modifications on the segment. Other objects are obtained based on the same ideas as described herein.

It is only important to be very careful when dealing with pending modifications: it should be remembered that even if the current vertex we have "pushed" the pending revision, the left and right sons are likely to have not done so. Therefore it is often necessary to call is push also on the left and right children of the current node, or as carefully consider pending modifications in them.

Generalization to higher dimensions

Segment tree generalized quite naturally on the two-dimensional and multi-dimensional case at all. If the one-dimensional case we broke array indexes on segments, the two-dimensional case will now first break all the indices in the first, and for each segment for the first index - build a common tree segments for the second index. Thus, the basic idea of the solution - it is inserting trees segments on the second index into the tree segments for the first index.

Let us explain this idea for an example of the problem.

Two-dimensional segment tree in the simplest case

Dana is a rectangular matrix $a[0 \dots n-1, 0 \dots m-1]$, and enter search queries amount (or minimum / maximum) at some podpryamougolnikah $a[x_1 \dots x_2, y_1 \dots y_2]$and requests modification of individual elements of the matrix (ie, queries of the form $a[x][y] = p$).

So, we will build a two-dimensional tree segments: first segment tree in the first coordinate (x), and then - on the second (y).

To **build process** more understandable, it is possible to forget that the original two-dimensional array has been, and leave only the first coordinate. We will construct the usual one-dimensional segment tree, working only with the first coordinate. But as the value of each segment, we will not record a number, as in the one-dimensional case, and the whole tree segments: ie at this point we are reminded that we still have and the second coordinate, but since at this point is recorded that the first coordinate is an interval $[l \dots r]$, we actually work with the band $a[l \dots r, 0 \dots m-1]$, and it builds a tree segments.

We present the implementation of operations for constructing a two-dimensional tree. It actually consists of two separate units: the construction of the tree segments in the coordinate x($\mathrm{build\_x}$) and the coordinate y($\mathrm{build\_y}$). If the first function is almost indistinguishable from the conventional one-dimensional tree, the latter is forced to deal separately with the two cases: when the current segment in the first coordinate ($[tlx \dots trx]$) has unit length, and when - length greater than one. In the first case, we simply take the required value from the matrix $a[][]$, and the second - combine the values of two tree lengths of the left and right son son coordinate x.

```
void build_y (int vx, int lx, int rx, int vy, int ly, int ry) {
        if (ly == ry)
                if (lx == rx)
                        t[vx][vy] = a[lx][ly];
                else
                        t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
        else {
                int my = (ly + ry) / 2;
                build_y (vx, lx, rx, vy*2, ly, my);
                build_y (vx, lx, rx, vy*2+1, my+1, ry);
                t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
        }
}

void build_x (int vx, int lx, int rx) {
        if (lx != rx) {
                int mx = (lx + rx) / 2;
                build_x (vx*2, lx, mx);
                build_x (vx*2+1, mx+1, rx);
        }
        build_y (vx, lx, rx, 1, 0, m-1);
}
```

This segment tree takes still linear memory, but more constant: $16nm$memory cells. It is clear that it is built above procedure $\mathrm{build\_x}$ also in linear time.

We now proceed to **process requests** . Respond to the two-dimensional inquiry will on the same principle: first break request to the first coordinate, and then when we got to the top of the tree some segments in the first coordinate - initiate a request from the relevant sections of the tree to the second coordinate.

```
int sum_y (int vx, int vy, int tly, int try_, int ly, int ry) {
        if (ly > ry)
                return 0;
        if (ly == tly && try_ == ry)
                return t[vx][vy];
        int tmy = (tly + try_) / 2;
        return sum_y (vx, vy*2, tly, tmy, ly, min(ry,tmy))
                + sum_y (vx, vy*2+1, tmy+1, try_, max(ly,tmy+1), ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
        if (lx > rx)
                return 0;
        if (lx == tlx && trx == rx)
                return sum_y (vx, 1, 0, m-1, ly, ry);
        int tmx = (tlx + trx) / 2;
        return sum_x (vx*2, tlx, tmx, lx, min(rx,tmx), ly, ry)
                + sum_x (vx*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry);
}
```

This function works in time $O(\log n \log m)$as she walks down the first tree in the first coordinate, and for each vertex passed this tree - makes a request for a normal tree segments to the second coordinate.

Finally, we consider the **modification request** . We want to learn how to modify the segment tree in accordance with a change in the value of any item $a[x][y] = p$. It is clear that changes will occur only in those segments of the first tree tops that cover the coordinate x (and they will $O(\log n)$), and trees for the segments corresponding to them - will change only in those tops that cover the coordinate y

(and there is $O(\log m)$). Therefore, the implementation of the modification request will not differ greatly from the one-dimensional case, only now we first down in the first coordinate, and then - on the second.

```
void update_y (int vx, int lx, int rx, int vy, int ly, int ry, int x, int y, int new_val) {
        if (ly == ry) {
                if (lx == rx)
                        t[vx][vy] = new_val;
                else
                        t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
        }
        else {
                int my = (ly + ry) / 2;
                if (y <= my)
                        update_y (vx, lx, rx, vy*2, ly, my, x, y, new_val);
                else
                        update_y (vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
                t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
        }
}

void update_x (int vx, int lx, int rx, int x, int y, int new_val) {
        if (lx != rx) {
                int mx = (lx + rx) / 2;
                if (x <= mx)
                        update_x (vx*2, lx, mx, x, y, new_val);
                else
                        update_x (vx*2+1, mx+1, rx, x, y, new_val);
        }
        update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
```

Compression of two-dimensional tree segments

Suppose that the problem is as follows: there are $n$points in the plane defined by its coordinates (x_i, y_i), and receives requests like "count the number of points lying in a rectangle $((x_1, y_1), (x_2, y_2))$. " It is clear that in the case of such a problem becomes unnecessarily wasteful to build a two-dimensional index tree $O(n^2)$elements. Much of this memory will be wasted because each separate point can be reached only in the $O(\log n)$segments of the tree segments in the first coordinate, and therefore, the total "useful" size of all segments of the trees to the second coordinate is the value $O(n \log n)$.

Then proceed as follows: each node of the tree segments in the first coordinate will store segment tree built only on those second coordinates, which are found in the current segment of the first coordinates. In other words, the construction of the tree segments within some vertex with the number $vx$and boundaries tlx, trx, we will consider only those points that fall in this segment $x \in [tlx; trx]$, and build a segment tree just above them.

Thus we ensure that every segment tree to the second coordinate will take exactly as much memory as it should. As a result, the total **amount of memory** is reduced to $O(n \log n)$. **Responding to a request** , we will continue for $O(\log^2 n)$just now when you call request from the tree segments the second coordinate, we'll have to do a binary search on the second coordinate, but it will not worsen the asymptotic behavior.

But the payback would be the impossibility of making an arbitrary **modification request** : in fact, if a new point, it will lead to what we would have in any tree segments to the second coordinate add a new element in the middle that can not be done effectively.

In conclusion, we note that a concise manner described two-dimensional segment tree is practically **equivalent to** the above-described modification of the one-dimensional tree segments (see "Saving the entire subarray in each node of the tree segments"). In particular, it turns out that what is described here two-dimensional segment tree - just a special case of a subarray of conservation at each vertex of the tree where the subarray itself is stored in a tree segments. It follows that if you have to abandon the two-dimensional tree segments due to inability to perform one or another query, it makes sense to try to replace the embedded tree segments to any more powerful data structure, such as the Cartesian tree .

Tree to preserving the history of its values (improvement to persistent-data structure)

Persistent-data structure called a data structure such that at each modification remembers its previous state. This allows to apply to any version that we are interested in the data structure and execute the request on it.

Segment tree is one of those data structures that can be converted into a persistent-data structure (of course, we consider the persistent-efficient structure, and not one that copies the entire himself entirely before each update).

In fact, any database changes to the tree causes a change of segments of data in $O(\log n)$the vertices and along a path starting from the root. So, if we keep on the segment tree pointers (ie pointers to the left and right sons do pointers stored in the top), then the update query instead we should just change the existing vertices to create new vertices of which are direct links to the old top. Thus, when requesting an update will be created $O(\log n)$new heights, including creates a new tree root segments, and all previous version of the tree, hanging over the old root, will remain unchanged.

Here is an example implementation for the simplest tree segments: when there is only a request for calculating the amount of subsegments and modification request singular.

```
struct vertex {
        vertex * l, * r;
        int sum;

        vertex (int val)
                : l(NULL), r(NULL), sum(val)
        { }

        vertex (vertex * l, vertex * r)
                : l(l), r(r), sum(0)
        {
                if (l)  sum += l->sum;
                if (r)  sum += r->sum;
        }
};

vertex * build (int a[], int tl, int tr) {
        if (tl == tr)
                return new vertex (a[tl]);
        int tm = (tl + tr) / 2;
        return new vertex (
                build (a, tl, tm),
                build (a, tm+1, tr)
        );
}

int get_sum (vertex * t, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && tr == r)
                return t->sum;
        int tm = (tl + tr) / 2;
        return get_sum (t->l, tl, tm, l, min(r,tm))
                + get_sum (t->r, tm+1, tr, max(l,tm+1), r);
}

vertex * update (vertex * t, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                return new vertex (new_val);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return new vertex (
                                update (t->l, tl, tm, pos, new_val),
                                t->r
                        );
        else
                return new vertex (
                                t->l,
                                update (t->r, tm+1, tr, pos, new_val)
                        );
}
```

By using this approach can be turned into persistent-data structure virtually any segment tree.

4 • Ответить • Поделиться ›

orga • год назад
could you add an implementation for range minimum query ? to update some interval(x, y) and retrieve min and max from some interval (x, y) ? I know it's already here, but in 2 different implementations
2 | 1 • Ответить • Поделиться ›

Andrey Naumenko • 2 года назад
Можно ли в 2-мерном дереве реализовать операцию присвоения или добавления на прямоугольнике за log^2(n), обобщив одномерный случай? Есть подозрение, что нельзя.
1 • Ответить • Поделиться ›

Tranvick ➔ Andrey Naumenko • 2 года назад
http://habrahabr.ru/post/13107...
2 • Ответить • Поделиться ›

Jeferson • год назад
How would I change the get_max function to a get_min function ? Thanks
1 • Ответить • Поделиться ›

e_maxx Модератор ➔ Jeferson • год назад
At first, invert the sign of "-INF". Then modify combine() function: invert both comparisons.
• Ответить • Поделиться ›

Sparik • 5 месяцев назад
"стандартному дереву отрезков требуется порядка элементов памяти для работы над массивом размера ."

достаточно 2n
• Ответить • Поделиться ›

Vlad • 7 месяцев назад
Помогите, пожалуйста. Вот я допустим на отрезке присвоил или прибавил процедурой update. Как мне теперь подсчитать новые значения в дереве? Т.е после выполнения update я сразу буду готов отвечать на запрос суммы/минимума/чего-то еще, или нужно как-то вызвать get? Потому что я так и не понял, что делает get и как ее вызвать и что она сделает. Помогите, пожалуйтса. Заранее спасибо!
• Ответить • Поделиться ›

Ivan • 11 месяцев назад
А зачем в "присвоение на отрезке" в функции "int get(int v, int tl, int tr, int v)" в функции "push(v)" t[v] = -1;
• Ответить • Поделиться ›

e_maxx Модератор ➔ Ivan • 11 месяцев назад
Чтобы снять отметку о том, что всё поддерево с корнем в вершине v надо перекрашивать: эту отметку мы уже "отдали" детям: v*2 и v*2+1. В противном случае, если не сделать t[v]=-1, то в дальнейшем, к примеру, если вершина v*2 целиком перекрасится в какой-то другой цвет, вершина v всё равно перезатрёт эти изменения своим цветом.
• Ответить • Поделиться ›

Steve_jobs • год назад
можно вычислять количество разных элементов в интервале с помощью дерево отрезков? Заранее спасибо.
• Ответить • Поделиться ›

e_maxx Модератор ➔ Steve_jobs • год назад
Давайте начнём с оффлайнового решения, когда все запросы известны заранее. Т.е. мы начинаем с пустого массива, дальше на каждом шаге мы добавляем в рассмотрение очередной элемент массива, и отвечаем на все запросы, правая граница которых оканчивается на текущем элементе. Как это делать: давайте для каждого числа X будем поддерживать позицию LAST[X] его последнего встреченного вхождения (изначально все они равны минус бесконечности). Тогда ответ на запрос "количество различных чисел на отрезке, начинающемся в L и заканчивающемся в текущей позиции" будет равен количеству чисел X, для которых LAST[X] >= L. Таким образом, построив и поддерживая дерево отрезков над этим массивом LAST[], мы можем отвечать на запрос количества различных чисел за O(log N) в оффлайне.

Чтобы перейти к онлайновому решению, (кажется), достаточно перейти к персистентному дереву отрезков: т.е. заранее промоделировать и сохранить деревья отрезков после добавления каждого элемента массива, а потом отвечать на запрос, обращаясь к дереву нужной версии.
• Ответить • Поделиться ›

Steve_jobs → e_maxx • год назад
Спасибо большое!!!
• Ответить • Поделиться ›

quest • год назад
а есть ли какая-нибудь структура данных, которая может поддерживать операции: прибавить на отрезке число x, найти нод на отрезке? и есть ли какие-нибудь задачи на эту штуку?
• Ответить • Поделиться ›

e_maxx Модератор → quest • год назад
Почти наверняка нет, поскольку после применения прибавления - факторизация чисел и, следовательно, НОД, могли сильно и труднопредсказуемо измениться (например, к массиву [2,7] с НОД=1 прибавили 3 - получился [5,10] с НОД=5).
• Ответить • Поделиться ›

guest → e_maxx • 2 месяца назад
Это можно сделать.

Заметим следующий факт: gcd(a + x, b + x) = gcd(b - a, b + x) = gcd(a - b, a + x) = gcd(b - a, a + x) = gcd(a - b, b + x).

Теперь, допустим, у нас есть числа a, b, c, d, к которым надо прибавить x.

gcd{a + x, b + x, c + x, d + x} = gcd{gcd(a + x, b + x), gcd(b + x, c + x), gcd(c + x, d + x)}.
Исходя из факта выше: gcd{a + x, b + x, c + x, d + x} = gcd{gcd(b - a, b + x), gcd(c - b, c + x), gcd(d - c, d + x)} =

gcd{gcd{b - a, c - b, d - c}, gcd{b + x, c + x, d + x}} = ... = gcd{b - a, c - b, d - c, d + x}.

Это значит, что чтобы прибавить к отрезку X и посчитать gcd на отрезке надо взять

gcd(gcd{разностей соседних элементов}, самый правый элемент на отрезке + x).
Т.к. при прибавлении на отрезке разности соседних элементов не меняются, то gcd{разности соседних элементов} можно поддерживать в

дереве отрезков.
Итак, что будем хранить в вершинах дерева:

показать больше
• Ответить • Поделиться ›

Reidor96 • 2 года назад
Объясните, пожалуйста, как реализовать следующее дерево: запрос модификации - прибавление на отрезке; запрос чтения - сумма на отрезке.
Заранее спасибо!
• Ответить • Поделиться ›

Reidor96 → Reidor96 • 2 года назад
Вопрос снят..
• Ответить • Поделиться ›

Lena → Reidor96 • 5 месяцев назад
Ответьте пожалуйста на этот вопрос!
• Ответить • Поделиться ›

Miras Mirzakerey • 2 года назад
Можно ли с помощью дерева отрезков перевернуть какой-то отрезок от l до r, сохраняя при этом дерево (не перестраивать) и с сравнительно небольшой асимптотикой?
• Ответить • Поделиться ›

e_maxx Модератор → Miras Mirzakerey • 2 года назад
На всякий случай напишу, что известная структура данных, позволяющая делать перевороты на отрезке, - это декартово дерево (http://e-maxx.ru/algo/treap).
1 • Ответить • Поделиться ›

e_maxx Модератор → Miras Mirzakerey • 2 года назад
Лично мне способа переворачивать с помощью "обычного" дерева отрезков (такого рода, как описаны в статье)

неизвестно.

1 • Ответить • Поделиться ›

Nurzhan Dyussenaliyev → e_maxx • 2 года назад

Можно же также как и в декартовом дереве. Номер левого потомка уже не будет v * 2, а будет L[v], и аналогично для правого. Это если нет запросов вставлений и удалений элементов из массива.

1 • Ответить • Поделиться ›

e_maxx Модератор → Nurzhan Dyussenaliyev • год назад

Что-то непонятно, как так можно. Допустим, длина массива равна 4 (т.е. индексы от 0 до 3), и нас попросили перевернуть с 1 по 3. Получается, одна половинка переворачиваемого (с 1 по 1) лежит в левом поддереве, а другая (с 2 по 3) - в правом поддереве, и как их обменять местами (особенно учитывая, что они имеют разную длину)?

1 • Ответить • Поделиться ›

Ruvin • 2 года назад

Что за функция int get (int v, int tl, int tr, int pos) в
Прибавление на отрезке и что такое pos