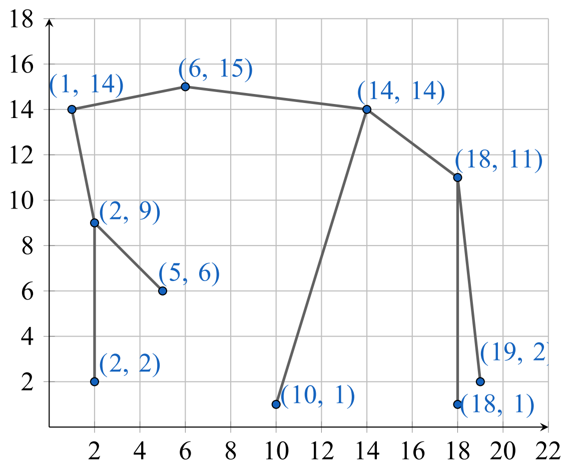
Last update: June 8, 2022 Translated From: e-maxx.ru

Treap (Cartesian tree)

A treap is a data structure which combines binary tree and binary heap (hence the name: tree + heap $\Rightarrow$ Treap).

More specifically, treap is a data structure that stores pairs (X, Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y. If some node of the tree contains values (X_0, Y_0), all nodes in the left subtree have $X \leq X_0$, all nodes in the right subtree have $X_0 \leq X$, and all nodes in both left and right subtrees have $Y \leq Y_0$.

A treap is also often referred to as a "cartesian tree", as it is easy to embed it in a Cartesian plane:



Treaps have been proposed by Raimund Siedel and Cecilia Aragon in 1989.

Advantages of such data organisation

In such implementation, X values are the keys (and at same time the values stored in the treap), and Y values are called **priorities**. Without priorities, the treap would be a regular binary search tree by X, and one set of X values could correspond to a lot of different trees, some of them degenerate (for example, in the form of a linked list), and therefore extremely slow (the main operations would have $O(N)$ complexity).

At the same time, **priorities** (when they're unique) allow to **uniquely** specify the tree that will be constructed (of course, it does not depend on the order in which values are added), which can be proven using corresponding theorem. Obviously, if you **choose the priorities randomly**, you will get non-degenerate trees on average, which will ensure $O(\log N)$ complexity for the main operations. Hence another name of this data structure - **randomized binary search tree**.

Operations

A treap provides the following operations:

- **Insert (X,Y)** in $O(\log N)$.
  Adds a new node to the tree. One possible variant is to pass only X and generate Y randomly inside the operation.

- **Search (X)** in $O(\log N)$.
  Looks for a node with the specified key value X. The implementation is the same as for an ordinary binary search tree.
- **Erase (X)** in $O(\log N)$.
  Looks for a node with the specified key value X and removes it from the tree.
- **Build** (X_1, ..., X_N) in $O(N)$.
  Builds a tree from a list of values. This can be done in linear time (assuming that $X_1, \dots, X_N$ are sorted).
- **Union** (T_1, T_2) in $O(M \log(N/M))$.
  Merges two trees, assuming that all the elements are different. It is possible to achieve the same complexity if duplicate elements should be removed during merge.
- **Intersect** (T_1, T_2) in $O(M \log(N/M))$.
  Finds the intersection of two trees (i.e. their common elements). We will not consider the implementation of this operation here.

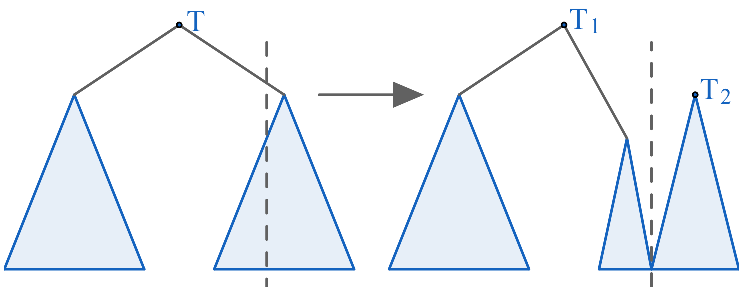
In addition, due to the fact that a treap is a binary search tree, it can implement other operations, such as finding the K-th largest element or finding the index of an element.

Implementation Description

In terms of implementation, each node contains X, Y and pointers to the left (L) and right (R) children.

We will implement all the required operations using just two auxiliary operations: Split and Merge.

Split



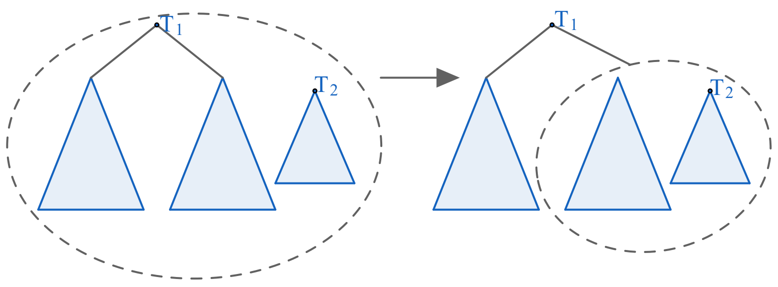
Split (T, X) separates tree T in 2 subtrees L and R trees (which are the return values of split) so that L contains all elements with key $X_L \le X$, and R contains all elements with key $X_R > X$. This operation has $O(\log N)$ complexity and is implemented using a clean recursion:

1. If the value of the root node (R) is $\le X$, then `L` would at least consist of `R->L` and `R`. We then call split on `R->R`, and note its split result as `L'` and `R'`. Finally, `L` would also contain `L'`, whereas `R = R'`.
2. If the value of the root node (R) is $> X$, then `R` would at least consist of `R` and `R->R`. We then call split on `R->L`, and note its split result as `L'` and `R'`. Finally, `L=L'`, whereas `R` would also contain `R'`.

Thus, the split algorithm is:

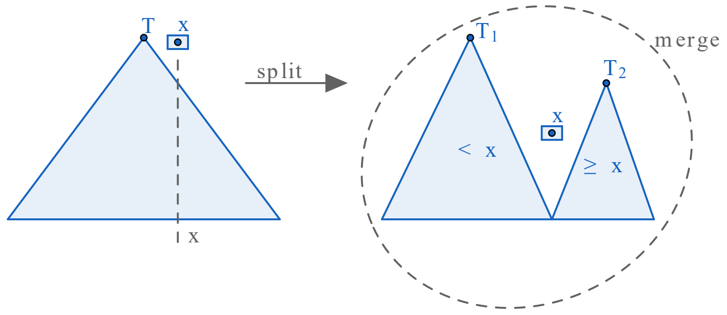
1. decide which subtree the root node would belong to (left or right)
2. recursively call split on one of its children
3. create the final result by reusing the recursive split call.

Merge



Merge (T_1, T_2) combines two subtrees T_1 and T_2 and returns the new tree. This operation also has $O(\log N)$ complexity. It works under the assumption that T_1 and T_2 are ordered (all keys X in T_1 are smaller than keys in T_2). Thus, we need to combine these trees without violating the order of priorities Y. To do this, we choose as the root the tree which has higher priority Y in the root node, and recursively call Merge for the other tree and the corresponding subtree of the selected root node.

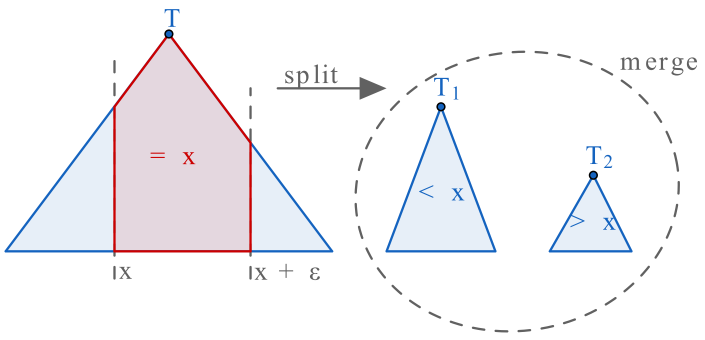
Insert



Now implementation of **Insert** (X, Y) becomes obvious. First we descend in the tree (as in a regular binary search tree by X), and stop at the first node in which the priority value is less than Y. We have found the place where we will insert the new element. Next, we call **Split (T, X)** on the subtree starting at the found node, and use returned subtrees L and R as left and right children of the new node.

Alternatively, insert can be done by splitting the initial treap on X and doing 2 merges with the new node (see the picture).

Erase



Implementation of **Erase** (X) is also clear. First we descend in the tree (as in a regular binary search tree by X), looking for the element we want to delete. Once the node is found, we call **Merge** on it children and put the return value of the operation in the place of the element we're deleting.

Alternatively, we can factor out the subtree holding X with 2 split operations and merge the remaining treaps (see the picture).

Build

We implement **Build** operation with $O(N \log N)$ complexity using N **Insert** calls.

Union

Union (T_1, T_2) has theoretical complexity $O(M \log(N/M))$, but in practice it works very well, probably with a very small hidden constant. Let's assume without loss of generality that $T_1 \rightarrow Y > T_2 \rightarrow Y$, i. e. root of T_1 will be the root of the result. To get the result, we need to merge trees $T_1 \rightarrow L$, $T_1 \rightarrow R$ and T_2 in two trees which could be children of T_1 root. To do this, we call Split (T_2, $T_1 \rightarrow X$), thus splitting T_2 in two parts L and R, which we then recursively combine with children of T_1: Union ($T_1 \rightarrow L$, L) and Union ($T_1 \rightarrow R$, R), thus getting left and right subtrees of the result.

Implementation

```
struct item {
    int key, prior;
    item *l, *r;
    item () { }
    item (int key) : key(key), prior(rand()), l(NULL), r(NULL) { }
    item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }
};
typedef item* pitem;
```

This is our item defintion. Note there are two child pointers, and an integer key (for the BST) and an integer priority (for the heap). The priority is assigned using a random number generator.

```
void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)
        l = r = NULL;
    else if (t->key <= key)
        split (t->r, key, t->r, r),  l = t;
    else
        split (t->l, key, l, t->l),  r = t;
}
```

`t` is the treap to split, and `key` is the BST value by which to split. Note that we do not `return` the result values anywhere, instead, we just use them like so:

```
pitem l = nullptr, r = nullptr;
split(t, 5, l, r);
if (l) cout << "Left subtree size: " << (l->size) << endl;
if (r) cout << "Right subtree size: " << (r->size) << endl;
```

This `split` function can be tricky to understand, as it has both pointers (`pitem`) as well as reference to those pointers (`pitem &l`). Let us understand in words what the function call `split(t, k, l, r)` intends: "split treap `t` by value `k` into two treaps, and store the left treaps in `l` and right treap in `r`". Great! Now, let us apply this definition to the two recursive calls, using the case work we analyzed in the previous section: (The first if condition is a trivial base case for an empty treap)

1. When the root node value is $\le$ key, we call `split (t->r, key, t->r, r)`, which means: "split treap `t->r` (right subtree of `t`) by value `key` and store the left subtree in `t->r` and right subtree in `r`". After that, we set `l = t`. Note now that the `l` result value contains `t->l`, `t` as well as `t->r` (which is the result of the recursive call we

made) all already merged in the correct order! You should pause to ensure that this result of `l` and `r` corresponds exactly with what we discussed earlier in Implementation Description.

2. When the root node value is greater than key, we call `split (t->l, key, l, t->l)`, which means: "split treap `t->l` (left subtree of `t`) by value `key` and store the left subtree in `l` and right subtree in `t->l`". After that, we set `r = t`. Note now that the `r` result value contains `t->l` (which is the result of the recursive call we made), `t` as well as `t->r`, all already merged in the correct order! You should pause to ensure that this result of `l` and `r` corresponds exactly with what we discussed earlier in Implementation Description.

If you're still having trouble understanding the implementation, you should look at it *inductively*, that is: do *not* try to break down the recursive calls over and over again. Assume the split implementation works correct on empty treap, then try to run it for a single node treap, then a two node treap, and so on, each time reusing your knowledge that split on smaller treaps works.

```
void insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r),  t = it;
    else
        insert (t->key <= it->key ? t->r : t->l, it);
}

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
}

void erase (pitem & t, int key) {
    if (t->key == key) {
        pitem th = t;
        merge (t, t->l, t->r);
        delete th;
    }
    else
        erase (key < t->key ? t->l : t->r, key);
}

pitem unite (pitem l, pitem r) {
    if (!l || !r)  return l ? l : r;
    if (l->prior < r->prior)  swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    return l;
}
```

Maintaining the sizes of subtrees

To extend the functionality of the treap, it is often necessary to store the number of nodes in subtree of each node - field `int cnt` in the `item` structure. For example, it can be used to find K-th largest element of tree in $O(\log N)$, or to find the index of the element in the sorted list with the same complexity. The implementation of these operations will be the same as for the regular binary search tree.

When a tree changes (nodes are added or removed etc.), `cnt` of some nodes should be updated accordingly. We'll create two functions: `cnt()` will return the current value of `cnt` or 0 if the node does not exist, and `upd_cnt()` will update the value of `cnt` for this node assuming that for its children L and R the values of `cnt` have already been updated. Evidently it's sufficient to add calls of `upd_cnt()` to the end of `insert`, `erase`, `split` and `merge` to keep `cnt` values up-to-date.

```
int cnt (pitem t) {
    return t ? t->cnt : 0;
}

void upd_cnt (pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt (t->r);
}
```

Building a Treap in $O(N)$ in offline mode

Given a sorted list of keys, it is possible to construct a treap faster than by inserting the keys one at a time which takes $O(N \log N)$. Since the keys are sorted, a balanced binary search tree can be easily constructed in linear time. The heap values Y are initialized randomly and then can be heapified independent of the keys X to build the heap in $O(N)$.

```
void heapify (pitem t) {
    if (!t) return;
    pitem max = t;
    if (t->l != NULL && t->l->prior > max->prior)
        max = t->l;
    if (t->r != NULL && t->r->prior > max->prior)
        max = t->r;
    if (max != t) {
        swap (t->prior, max->prior);
        heapify (max);
    }
}

pitem build (int * a, int n) {
    // Construct a treap on values {a[0], a[1], ..., a[n - 1]}
    if (n == 0) return NULL;
    int mid = n / 2;
    pitem t = new item (a[mid], rand ());
    t->l = build (a, mid);
    t->r = build (a + mid + 1, n - mid - 1);
    heapify (t);
    upd_cnt(t)
    return t;
}
```

Note: calling `upd_cnt(t)` is only necessary if you need the subtree sizes.

The approach above always provides a perfectly balanced tree, which is generally good for practical purposes, but at the cost of not preserving the priorities that were initially assigned to each node. Thus, this approach is not feasible to solve the following problem:

acmsguru - Cartesian Tree

Given a sequence of pairs (x_i, y_i), construct a cartesian tree on them. All x_i and all y_i are unique.

Note that in this problem priorities are not random, hence just inserting vertices one by one could provide a quadratic solution.

One of possible solutions here is to find for each element the closest elements to the left and to the right which have a smaller priority than this element. Among these two elements, the one with the larger priority must be the parent of the current element.

This problem is solvable with a minimum stack modification in linear time:

```
void connect(auto from, auto to) {
    vector<pitem> st;
    for(auto it: ranges::subrange(from, to)) {
        while(!st.empty() && st.back()->prior > it->prior) {
            st.pop_back();
        }
        if(!st.empty()) {
            if(!it->p || it->p->prior < st.back()->prior) {
                it->p = st.back();
            }
        }
        st.push_back(it);
    }
}

pitem build(int *x, int *y, int n) {
    vector<pitem> nodes(n);
    for(int i = 0; i < n; i++) {
        nodes[i] = new item(x[i], y[i]);
    }
    connect(nodes.begin(), nodes.end());
    connect(nodes.rbegin(), nodes.rend());
    for(int i = 0; i < n; i++) {
        if(nodes[i]->p) {
            if(nodes[i]->p->key < nodes[i]->key) {
                nodes[i]->p->r = nodes[i];
            } else {
                nodes[i]->p->l = nodes[i];
            }
        }
    }
    return nodes[min_element(y, y + n) - y];
}
```

Implicit Treaps

Implicit treap is a simple modification of the regular treap which is a very powerful data structure. In fact, implicit treap can be considered as an array with the following procedures implemented (all in $O(\log N)$ in the online mode):

- Inserting an element in the array in any location
- Removal of an arbitrary element
- Finding sum, minimum / maximum element etc. on an arbitrary interval
- Addition, painting on an arbitrary interval
- Reversing elements on an arbitrary interval

The idea is that the keys should be null-based **indices** of the elements in the array. But we will not store these values explicitly (otherwise, for example, inserting an element would cause changes of the key in $O(N)$ nodes of the tree).

Note that the key of a node is the number of nodes less than it (such nodes can be present not only in its left subtree but also in left subtrees of its ancestors). More specifically, the **implicit key** for some node T is the number of vertices $cnt(T \rightarrow L)$ in the left subtree of this node plus similar values $cnt(P \rightarrow L) + 1$ for each ancestor P of the node T, if T is in the right subtree of P.

Now it's clear how to calculate the implicit key of current node quickly. Since in all operations we arrive to any node by descending in the tree, we can just accumulate this sum and pass it to the function. If we go to the left subtree, the accumulated sum does not change, if we go to the right subtree it increases by $cnt(T \rightarrow L) + 1$.

Here are the new implementations of **Split** and **Merge**:

```
void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    int cur_key = add + cnt(t->l); //implicit key
    if (key <= cur_key)
        split (t->l, l, t->l, key, add),  r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}
```

In the implementation above, after the call of $split(T, T_1, T_2, k)$, the tree T_1 will consist of first k elements of T (that is, of elements having their implicit key less than k) and T_2 will consist of all the rest.

Now let's consider the implementation of various operations on implicit treaps:

- **Insert element**.
  Suppose we need to insert an element at position pos. We divide the treap into two parts, which correspond to arrays $[0..pos-1]$ and $[pos..sz]$; to do this we call $split(T, T_1, T_2, pos)$. Then we can combine tree T_1 with the new vertex by calling $merge(T_1, T_1, \text{new item})$ (it is easy to see that all preconditions are met). Finally, we combine trees T_1 and T_2 back into T by calling $merge(T, T_1, T_2)$.
- **Delete element**.
  This operation is even easier: find the element to be deleted T, perform merge of its children L and R, and replace the element T with the result of merge. In fact, element deletion in the implicit treap is exactly the same as in the regular treap.
- Find **sum / minimum**, etc. on the interval.
  First, create an additional field F in the `item` structure to store the value of the target function for this node's subtree. This field is easy to maintain similarly to maintaining sizes of subtrees: create a function which calculates this value for a node based on values for its children and add calls of this function in the end of all functions which modify the tree.
  Second, we need to know how to process a query for an arbitrary interval $[A; B]$.
  To get a part of tree which corresponds to the interval $[A; B]$, we need to call $split(T, T_2, T_3, B+1)$, and then $split(T_2, T_1, T_2, A)$: after this T_2 will consist of all the elements in the interval $[A; B]$, and only of them. Therefore, the response to the query will be stored in the field F of the root of T_2. After the query is answered, the tree has to be restored by calling $merge(T, T_1, T_2)$ and $merge(T, T, T_3)$.
- **Addition / painting** on the interval.
  We act similarly to the previous paragraph, but instead of the field F we will store a field `add` which will contain the added value for the subtree (or the value to which the subtree is painted). Before performing any operation we have to "push" this value correctly - i.e. change $T \to L \to add$ and $T \to R \to add$, and to clean up `add` in the parent node. This way after any changes to the tree the information will not be lost.
- **Reverse** on the interval.
  This is again similar to the previous operation: we have to add boolean flag `rev` and set it to true when the subtree of the current node has to be reversed. "Pushing" this value is a bit complicated - we swap children of this node and set this flag to true for them.

Here is an example implementation of the implicit treap with reverse on the interval. For each node we store field called `value` which is the actual value of the array element at current position. We also provide implementation of the function `output()`, which outputs an array that corresponds to the current state of the implicit treap.

```
typedef struct item * pitem;
struct item {
    int prior, value, cnt;
    bool rev;
    pitem l, r;
};

int cnt (pitem it) {
    return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void push (pitem it) {
    if (it && it->rev) {
        it->rev = false;
        swap (it->l, it->r);
        if (it->l)  it->l->rev ^= true;
        if (it->r)  it->r->rev ^= true;
    }
}

void merge (pitem & t, pitem l, pitem r) {
    push (l);
    push (r);
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)
        split (t->l, l, t->l, key, add),  r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}

void reverse (pitem t, int l, int r) {
    pitem t1, t2, t3;
    split (t, t1, t2, l);
    split (t2, t2, t3, r-l+1);
    t2->rev ^= true;
    merge (t, t1, t2);
    merge (t, t, t3);
}

void output (pitem t) {
    if (!t)  return;
    push (t);
    output (t->l);
    printf ("%d ", t->value);
    output (t->r);
}
```

Literature

- Blelloch, Reid-Miller "Fast Set Operations Using Treaps"

Practice Problems

- SPOJ - Ada and Aphids
- SPOJ - Ada and Harvest
- Codeforces - Radio Stations
- SPOJ - Ghost Town
- SPOJ - Arrangement Validity
- SPOJ - All in One
- Codeforces - Dog Show
- Codeforces - Yet Another Array Queries Problem
- SPOJ - Mean of Array
- SPOJ - TWIST
- SPOJ - KOILINE
- CodeChef - The Prestige
- Codeforces - T-Shirts
- Codeforces - Wizards and Roads
- Codeforces - Yaroslav and Points

Contributors:
Leefrost (41.84%) adamant-pwn (28.28%) tcNickolas (8.97%) GaurangTandon (7.36%) meooow25 (6.44%) Morass (3.22%) Dalgerok (1.38%) jakobkogler (0.92%) algmyr (0.92%) princezna (0.46%) kroist (0.23%)