CP-Algorithms

Page Authors

Search

Treap (Cartesian tree)

Table of Contents

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. Assuming that all X and all Y are different, we can see that if some node of the tree contains values (X_0, Y_0), all nodes in the left subtree have $X < X_0$, all nodes in the right subtree have $X > X_0$, and all nodes in both left and right subtrees have $Y < Y_0$.

Treaps have been proposed by Siedel and 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** 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 (while ensuring that it's different from all other priorities in the tree).
- **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), but we will not discuss this implementation here. We will just use N serial calls of **Insert** operation, which has $O(N \log N)$ complexity.
- **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 (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 $X_L < X$, and R contains all elements with key $X_R > X$. This operation has $O(\log N)$ complexity and is implemented using an obvious recursion.

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.

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.

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.

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

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 $T_1 \to L$, $T_1 \to R$ and T_2 in two trees which could be children of T_1 root. To do this, we call Split (T_2, $T_1 \to X$), thus splitting T_2 in two parts L and R, which we then recursively combine with children of T_1: Union ($T_1 \to L$, L) and Union ($T_1 \to R$, R), thus getting left and right subtrees of the result.

Implementation

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

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

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 =
    else
        insert (it->key < t->key ? t->l : t->r
}

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)
        merge (t, t->l, t->r);
    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->pri
        max = t->l;
    if (t->r != NULL && t->r->prior > max->pri
        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]
    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);
    return t;
}
```

It's also possible to compute the sizes of the subtrees with this approach. It is enough to call `upd_cnt(t)` before returning from the `build` function.

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

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, 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_1, T_2, A), and then

`split` (T_2, T_2, T_3, B - A + 1): 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
    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 =
    else
        split (t->r, t->r, r, key, add + 1 + c
    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