Balanced Binary Search Trees

The be-oi team

August 25, 2018

Table of Contents

Table of Contents

Introduction

As we saw in a previous lesson, BSTs are powerful data structures that operate in $\mathcal{O}(\text{height of the tree})$, which can be kept to $\mathcal{O}(\log(n))$ for optimal performance.

There are **many** ways to do so, but most are complicated/long/cumbersome/error-prone. Treaps are however quite easy to understand and to implement.

Look that up if you are interested

B-trees (including Red-Black and AA trees), AVL trees, splay trees, many others.

Main idea

Treap = (binary search) tree + heap. Any treap node respects

- the BST property: key(left child) $<$ key(node) $<$ key(right child);
- the heap property: priority(node) $>$ priority(child).

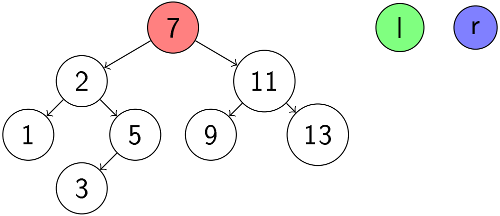
If we randomly assign the priorities, the complexity of the BST operations will be $\mathcal{O}(\log(n))$. We will use d in big oh notation to express the depth of the tree.

Implementation

Treaps can be implemented using rotations or using the split/merge helper functions. As they are equivalent, we will only discuss the split/merge strategy because it is shorter. Feel free to look up the other one on the Internet.

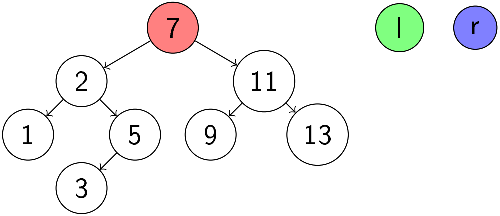
- $\mathrm{split}(t, X)$ separates treap t into treaps l and r containing elements $\leq X$ and $> X$ respectively.
- $\mathrm{merge}(l, r)$ combines treaps l and r into treap t *under the assumption that every value in* l *is* $<$ *than any value in* r.

Split



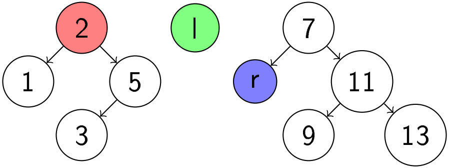
Let's start with an example to understand how split works. We will split t at 4, and we will store the result in l and r. Note that priorities don't matter when splitting.

Split



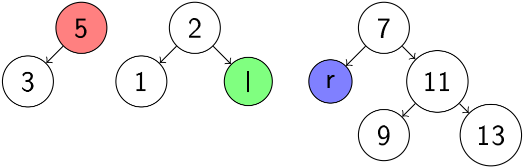
We see that t has a value of 7, which is > 4. Hence t and its right subtree clearly belong to r.

Split



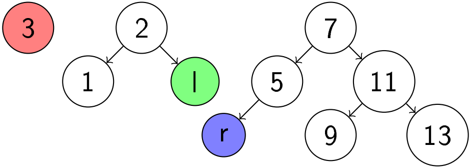
We see that t has a value of 2, which is ≤ 4. Hence t and its left subtree clearly belong to l.

Split



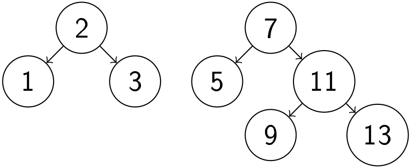
We see that t has a value of 5, which is > 4. Hence t and its right subtree clearly belong to r.

Split



We see that t has a value of 3, which is ≤ 4. Hence t and its left subtree clearly belong to l.

Split



And... we are done! Implementation time!

Treap node

```
struct node {
    node *l, *r;
    int key, val, pr;
};
using pnode = node*;
```

You can obviously change key and value depending on the problem you are trying to solve.

Split implementation

```
// TODO: test this code
void split(pnode t, int X, pnode* l, pnode* r) {
    if(!t) return;
    if(t->key > X) {
        pnode newt = t->l; t->l = 0;
        *r = t;
        split(newt, X, l, &t->l);
    } else {
        pnode newt = t->r; t->r = 0;
        *l = t;
        split(newt, X, &t->r, r);
    }
}
```

This is the straightforward implementation of the split function. It should be easy to understand. Note that we need a pointer to a pointer if we want to modify an edge (which is already a pointer by itself).

Split implementation

```
// TODO: test this code
void split(pnode t, int X, pnode* l, pnode* r) {
    if(!t)
        *l = *r = 0;
    if(t->key > X)
        split(t->l, X, l, &t->l), *r = t;
    else
        split(t->r, X, &t->r, r), *l = t;
}
```

This is a compressed version of the split function.

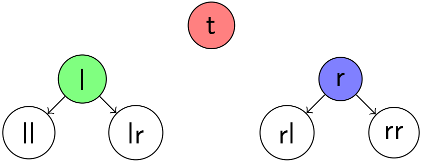
Split implementation

```
// TODO: test this code
void split(pnode t, int X, pnode& l, pnode& r) {
    if(!t)
        l = r = 0;
    else if(t->key > X)
        split(t->l, X, l, t->l), r = t;
    else
        split(t->r, X, t->r, r), l = t;
}
```

This is the same compressed version with references instead of pointers. You should use this one in contests.

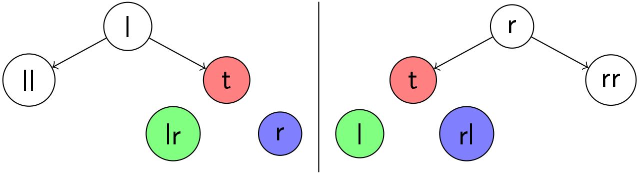
Merge

Merge is simpler than split because less things are going on. Let's see how we merge l and r into t.



Merge

We can either assign l or r to t and recursively merge whatever's left.



How do we choose? Priorities, of course!

Note that merge doesn't care about the keys because of the assumption that every key in l is smaller than every key in r.

Merge implementation

```
// TODO: test this code
void merge(pnode& t, pnode l, pnode r) {
    if(!l || !r)
        t = l ? l : r;
    else if(l.pr > r.pr)
        merge(l->r, l->r, r), t = l;
    else
        merge(r->l, l, r->l), t = r;
}
```

Insert implementation

To insert a node with key X, we go down the treap until we find a node with priority $< X$. Then we split that node at X.

```
// TODO: test this code
void insert(pnode& t, pnode it) {
    if(!t)
        t = it;
    else if(t.pr < it.pr)
        split(t, it->l, it->r), t = it;
    else
        insert(it->key > t->key ? t->r : t->l, it);
}
```

Erase implementation

To erase the node with key X, we find it and then merge its children.

```
// TODO: test this code
void erase(pnode& t, int X) {
    if(!t)
        return; // Not found
    if(t->key == X)
        merge(t, t->l, t->r);
    else
        erase(X > t->key ? t->r : r->l, X);
}
```

Keep track of size

You can lazily keep track of the size of a treap, with no additional complexity. Add a |size| field to the |node| struct and update it at the end of split, merge, insert and erase.

```
int size(pnode t) {
    return t ? t->size : 1;
}
void update_size(pnode t) {
    if(!t) return;
    t->size = size(t->l) + 1 + size(t->r);
}
```

Additional queries

We can use this information to quickly the index of the element with key X.

```
// TODO: test this code
int find_index(pnode& t, int X, int before=0) {
    if(!t)
        return -1; // Not found.
    else if(t->key == X)
        return before + size(t->r);
    else
        return find_index(t->key < X ? t->r : t->l, X,
        before + (t->key < X ? size(t->r) + 1 : 0));
}
```

Additional queries

We can also perform range … queries on a key interval. Here is an rmq example:

```
// TODO: test this code
int get_min(pnode t) {
    return t ? t->min : INF;
}
void update_min(pnode t) {
    // call after split, merge, insert, erase
    if(!t) return;
    t->min = min(min(get_min(t->l), get_min(t->r)), t->val);
}
int rmq(pnode& t, int l, int r) {
    pnode a, b, c;
    split(t, l-1, a, b);
    split(b, r, b, c);
    int result = get_min(b);
    merge(b, b, c);
    merge(t, a, b);
    return result;
}
```

Conclusion

Treaps are easy to implement and support a wide range of operations, all in $\mathcal{O}(d)$. Haven't had enough? Time to introduce you to implicit treaps.

Table of Contents

Implicit treaps

An implicit treap is a treap that uses the index of an element as its (implicit) key, like an array. Additionally, it supports insertion/deletion/modification and all kinds of range updates and queries, just like a standard treap.

For this to work, we need to keep track of the position of the current element everywhere we would have used the key in a standard treap.

Modified split implementation

Here is a modified split implementation for reference. Insert and erase are left as an exercise.

```
// TODO: test this code
void split(pnode t, int X, pnode& l, pnode& r, int before = 0) {
    if(!t)
        l = r = 0; return;
    int key = before + size(t->l);
    if(key > X)
        split(t->l, X, l, t->l, before), r = t;
    else
        split(t->r, X, t->r, r, before+size(t->l)+1), l = t;
    update_size(t);
}
```

Range updates

Treaps support range updates with lazy propagation. Just propagate at the beginning of every function.

Example problem: reverse interval [l, r] up to 10^5 times. Ideas?

Range reverse

```
// TODO: test this code
void reverse(pnode t) {
    if(!t) return;
    t->rev = !t->rev;
}
void propagate(pnode t) {
    if(!t) return;
    if(t->rev) {
        reverse(t->l); reverse(t->r);
        swap(t->l, t->r); t->rev = false;
    }
}
void range_reverse(pnode& t, int l, int r) {
    pnode a, b, c;
    split(t, l-1, a, b);
    split(b, r, b, c);
    reverse(b);
    merge(b, b, c);
    merge(t, a, b);
}
```