1 Introduction

Treaps can do a lot. Not only can they do everything a normal Segment Tree can do, but they can also mimic lazy propagation, reverse interval ranges, and delete contiguous subsequences. Best of all, they're quick to implement and extremely modifiable. In this expository paper, I will introduce what a Treap is. Since Binary Search Trees are a prerequisite to understand Treaps, I will also discuss BSTs shortly, although, ideally, the reader would already have some familiarity with BSTs.

2 Binary Search Trees

Binary Search Trees are arguably one of the simplest data structures.

2.1 Definition

Binary Search Trees (BSTs) are nothing more than special binary trees that satisfy what is called the **Binary Search Tree Property**:

- The keys in a node's left subtree are strictly smaller than the key of the node itself.
- The keys in a node's right subtree are strictly larger than the key of the node itself.

2.2 Searching

Binary Search Trees make it easy to **search** for a given value. To search for a value `val`, we can utilize the following recursive technique:

- Start at the root.
- If the value of the root is equal to `val`, then we have found the node in question.
- If the value of the root is less than `val`, then we know that the left subtree of our node is strictly less than `val`, so if `val` does indeed exist, it must exist in the right subtree. Recurse down to the right subtree.
- If the value of the root is greater than `val`, then we know that the right subtree of our node is strictly greater than `val`, so if `val` does indeed exist, it must exist in the left subtree. Recurse down to the left subtree.

In code, it looks as such:

```
TreeNode* searchBST(TreeNode* root, int val) {
    if (!root or root->val == val) {
        return root;
    }
    if (root->val > val) {
        return searchBST(root->left, val);
    } else {
        return searchBST(root->right, val);
    }
}
```

2.3 Adding a Node

Adding a node with a given key value `val` is quite simple as well. If `val` exists elsewhere in the tree, then we do not need to change anything. Otherwise, we can perform a very simple algorithm:

```
TreeNode* insertIntoBST(TreeNode* root, int val) {
    if (root == NULL) {
        return new TreeNode(val);
    }
    if (root->val > val) {
        root->left = insertIntoBST(root->left , val);
    } else {
        root->right = insertIntoBST(root->right , val);
    }
    return root;
}
```

2.4 Random Keys

Suppose we build a BST from a random permutation. Let $q(n)$ be the average number of nodes on the path from the root down to an arbitrary node in the BST.

Claim. $q(n) \leq 2 \cdot H_n$, where H_n is the nth Harmonic number.

Proof.

$$q(n) = 1 + \frac{1}{n^2} \sum_{x=1}^{n} \left(\sum_{r=1}^{x-1} q(x-1) + \sum_{r=x+1}^{n} q(n-x) \right) \tag{1}$$

$$= 1 + \frac{1}{n^2} \sum_{x=1}^{n} q(x-1) \cdot (x-1) + q(n-x) \cdot (n-x) \tag{2}$$

$$= 1 + \frac{2}{n^2} \sum_{x=1}^{n-1} q(x) \cdot x \tag{3}$$

With a recurrence relation for q under our belt, the next natural step is to bound q.

We will now prove using the method of mathematical induction that $q_n \leq 2 \cdot H_n$. We can verify the base case, when $n = 1$, quite easily because $q_1 = 1 \leq 2 \cdot H_1 = 2$. As for the inductive

step:

$$\begin{aligned}
q(n) &= 1 + \frac{2}{n^2}\sum_{x=1}^{n-1} q(x)\cdot x \\
&\leq 1 + \frac{2}{n^2}\sum_{x=1}^{n-1} 2xH_x \\
&= 1 + \frac{4}{n^2}\sum_{i=1}^{n-1}\frac{1}{i}\sum_{x=i}^{n-1} x \\
&= 1 + \frac{4}{n^2}\sum_{i=1}^{n-1}\frac{1}{i}\left(\binom{n}{2} - \binom{i}{2}\right) \\
&= 1 + \frac{4}{n^2}\binom{n}{2}H_{n-1} - \frac{4}{n^2}\sum_{i=1}^{n-1}\frac{1}{i}\binom{i}{2} \\
&= 1 + 2H_{n-1} - \frac{2}{n}H_{n-1} - \frac{n-1}{n} \\
&= \frac{1}{n} + 2H_{n-1} - \frac{2}{n}H_{n-1} \\
&\leq 2H_n,
\end{aligned}$$

from which it follows that indeed $q(n) \leq 2 \cdot H_n$ by induction. ■

This means that the average number of queries to search or insert a random key in a random BST of size n is around $\mathcal{O}(\log N)$. So BSTs tend to be fast. The problem is that it is very easy to construct a case in which BSTs insertions are slow: just add the keys in monotonically increasing or decreasing order.

This is where Treaps come into play.

3 Treaps

3.1 Definition

In a **Treap** can be thought of as a randomized BST, in some ways. Each node has two values:

- A random `priority`. Usually, in practice, `priority` is a random value from 1 to 10^9.
- A `key`.

```
template <typename T>
struct Node {
    T x; //key
    T y; //order in which it is put into binary tree
    Node* left; //left child
    Node* right; //right child
    Node (int ind) { //randomly assign order index
        this->x = ind, this->y = rand() % (int)1e9;
        this->left = this->right = nullptr;
```

```
    }
};
```

For those familiar with heaps, the values in a Treap satisfy the Binary Search Property, and the priorities in a Treap satisfy the min-heap (or max-heap, if you prefer) property.

A Treap can be thought of as the resulting Binary Search Tree formed if you add the keys in increasing (or decreasing, it doesn't really matter) order of priority. Effectively, a Treap basically randomizes the insertion times of a Binary Search Tree.

3.2 Splitting & Merging

There are two simple, yet powerful, operations Treaps can do: splitting and merging.

- For any value x, you can split a Treap into two Treaps: one with keys $\leq x$ and another with keys $> x$.
- If all keys in one Treap are strictly smaller than all keys in another Treap, then we can merge those two Treaps.

The worst case time complexity for both of these operations is $\mathcal{O}(N)$, but on average, they run in $\mathcal{O}(\log N)$, making them extremely fast in practice.

3.2.1 Splitting

You can split a Treap recursively into two Treaps with keys $\leq x$ and $> x$ respectively:

- Start at the root.
- If the key of the current node is smaller than or equal to x:
  - If the right child doesn't exist, then we know that all nodes in that Treap are $\leq x$. We are done.
  - If the right child does indeed exist, then we recurse down to the right subtree and merge the results appropriately.
- If the key of our current node is greater than x:
  - If the left child doesn't exist, then we know that all nodes in that Treap are $> x$. We are done.
  - If the left child does indeed exist, then we recurse down to the left subtree and merge the results appropriately.

```
pair<Node*, Node*> split (Node* node, int x) { //<= x and > x
    if (node->x <= x) {
        if (!node->right)
            return make_pair(node, nullptr);
        auto p = split(node->right, x);
        node->right = p.first;
        return make_pair(node, p.second);
    } else {
```

```
        if (!node->left)
            return make_pair(nullptr, node);
        auto p = split(node->left, x);
        node->left = p.second;
        return make_pair(p.first, node);
}
```

The time complexity for splitting two randomized Treaps of size n is expected to be $\mathcal{O}(\log N)$, but it can be $\mathcal{O}(N)$ at worst.

3.3 Merging

We can easily recursively merge the Treaps `Treap1` and `Treap2` if all keys in `Treap1` are smaller than all keys in `Treap2`.

- Start at the root.
- If either `Treap1` or `Treap2` is empty, then we are done.
- If the priority of the root of `Treap1` is smaller than the priority of the root of `Treap2`, then we know that we just need to merge the results of `node1.right` and `node2`. We can do so recursively.
- If the priority of the root of `Treap1` is larger than the priority of the root of `Treap2`, then we know that we just need to merge the results of `node2.left` and `node1`. We can do so recursively.

```
Node* merge (Node* node1, Node* node2) {
    if (!node1)
        return node2;
    if (!node2)
        return node1;
    if (node1->y < node2-> y) {
        node1->right = merge(node1->right, node2);
        return node1;
    } else {
        node2->left = merge(node1, node2->left);
        return node2;
    }
}
```

The time complexity for merging two randomized Treaps of size n is expected to be $\mathcal{O}(\log N)$, but it can be $\mathcal{O}(N)$ at worst.

4 Interesting Problems

Below is a list of some competitive programming problems that can be solved with Treaps. They are taken from the CSES website, the Chinese National Olympiad in Informatics, and from Codeforces. [1]

[1] Most of the problems, if not all of them, require modifications to standard Treap techniques.

- https://cses.fi/problemset/task/2073
- https://cses.fi/problemset/task/2072
- https://cses.fi/problemset/task/1648
- https://cses.fi/problemset/task/1651
- https://cses.fi/problemset/task/1649
- https://dmoj.ca/problem/noi05p2
- https://dmoj.ca/problem/noi04p1
- https://codeforces.com/contest/455/problem/D
- https://codeforces.com/gym/100488/problem/L
- https://codeforces.com/contest/702/problem/F