Treap

Nicolas Camenisch

Januar 2019

1 What is a treap?

A treap (conjunction of the words **tre**e and he**ap**) is a binary search tree that uses randomization and the binary heap property to maintain balance with a high probability. It maintains a dynamic set of ordered keys and allows binary searches among these keys.[1]

Each node of a treap contains two values: A key (ordered using binary search tree properties) and a priority (ordered using heap properties). The heap order can either be maintained using a max heap or a min heap. In all following examples a max heap will be used.

The structure of a treap follows these rules:[2]

1. The nodes in the left subtree of the root will have key values that are less than the key of the root. $\forall \texttt{x} \in \texttt{root.left}, \quad \texttt{x.key} < \texttt{root.key}$
2. The nodes in the right subtree of the root will have key values that are greater than the key of the root. $\forall \texttt{x} \in \texttt{root.right}, \quad \texttt{x.key} > \texttt{root.key}$
3. All priorities are unique.
4. The priority value of the root is greater than the priority values of its children. $\forall \texttt{x} \in \texttt{root.left} \cup \texttt{root.right}, \quad \texttt{x.priority} < \texttt{root.priority}$

These rules ensure, that there is only one unique and valid structure for each possible set of keys and priorities.

2 Operations

Treaps support a variety of operations. These operations can either be implemented using tree rotations (Figure 1), or using merge and split. Because the latter supports more operations, we're going to focus on merge and spit.

2.1 Merge & Split

We start out by defining the struct for our treap and the corresponding nodes (Listing 1). The `Treap` struct is very basic and only contains a pointer to the root node and some functions. A `Node` consists of two pointers pointing to its left and right child, the `key` and `priority` values, and a `size` variable that keeps track of the size of both subtrees. During initialization the priority gets assigned a random integer.

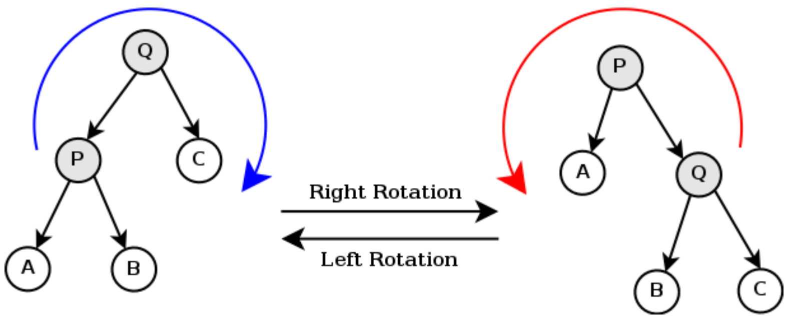

Figure 1: A tree rotation is an operation on a binary tree that changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down.[3] [4]

```cpp
template <typename T>
struct Treap {
    struct Node {
        int priority;
        T key;
        size_t size = 0;
        std::unique_ptr<Node> l, r;

        Node(T key): priority(rand()), key(key) {}
    };

    std::unique_ptr<Node> root;

    void insert(const T& key);
    void erase(const T& key);
    T at(size_t index);
    size_t size() const {
        return (root == NULL ? 0 : root->size + 1);
    }

private:
    static std::pair<std::unique_ptr<Node>, std::unique_ptr<Node>> split(std::
    ↪ unique_ptr<Node> current, const T& key);
    static std::unique_ptr<Node> merge(std::unique_ptr<Node> l, std::unique_ptr<Node> r
    ↪ );
};
```

Listing 1: The Treap struct used in the following implementation.

2.1.1 Split

The split function (Listing 2) splits a treap into two separate treaps, those smaller or equal the key x, and those larger than the key x. The following implementation is in-place destructive, but there also exist nondestructive algorithms.

```cpp
static std::pair<std::unique_ptr<Node>, std::unique_ptr<Node>> split(std::unique_ptr<
    ↪ Node> current, const T& key) {
    std::unique_ptr<Node> l, r;

    if (current == NULL)
```

```
        return {std::move(l), std::move(r)};
    if (key < current->key) {
        r = std::move(current);
        tie(l, r->l) = split(std::move(r->l), key);
        r->size -= (l == NULL ? 0 : l->size + 1);
    } else {
        l = std::move(current);
        tie(l->r, r) = split(std::move(l->r), key);
        l->size -= (r == NULL ? 0 : r->size + 1);
    }

    return {std::move(l), std::move(r)};
}
```

Listing 2: The `split` function.[5]

2.1.2 Merge

The merge function (Listing 3) merges two treaps into one treap. For this to work the greatest key in the first treap has to be less than or equal the smallest value in the second treap. This is always true if the first and second tree both are the result of a split operation. Like the split function, this implementation is in-place destructive, but there also exist nondestructive versions.

```
static std::unique_ptr<Node> merge(std::unique_ptr<Node> l, std::unique_ptr<Node> r) {
    if (l == NULL || r == NULL)
        return (l != NULL) ? std::move(l) : std::move(r);

    if (l->priority > r->priority) {
        l->size += r->size + 1;
        l->r = merge(std::move(l->r), std::move(r));
        return l;
    } else {
        r->size += l->size + 1;
        r->l = merge(std::move(l), std::move(r->l));
        return r;
    }
}
```

Listing 3: The `merge` function.[5]

2.2 Basic operation

Based on the split and merge functions we can implement three different basic operations: search, insert and delete. Because the search operation is a standard binary search in a binary tree (ignoring the priorities), we'll skip over it. Since the following operations only modify the treap by calling the `merge` and `split` functions, the heap invariant never gets violated.

2.2.1 Insert

To insert a node `N`, you first search the correct position to insert it into. This means a position that doesn't violate either the BST or max heap properties. This can be done by splitting the treap into two treaps `l` and `r`, one where all values are less than or equal the key of `N` (`l`), and one where the key is greater (`r`). Finally you merge `l`, `N` and `r` to form a treap that contains the new node `N` (Listing 4).

```
void insert(const T& key) {
    auto node = std::make_unique<Node>(key);

    if (root == NULL) {
        root = std::move(node);
    } else {
        auto [l, r] = split(std::move(root), key);
        root = merge(merge(std::move(l), std::move(node)), std::move(r));
    }
}
```

Listing 4: The `insert` function.[5]

2.2.2 Delete

To remove a key k, you have to split the treap into three separate treaps: one containing only keys that are smaller than k (`l`), equal to k (`eq`) and greater than k (`r`). You then merge the left and the right subtree of `eq` to remove the node of highest priority with key k. Finally you merge all three treaps back together to form a treap where the node with key k is missing (Listing 5).

```
void erase(const T& key) {
    if (root == NULL)
        return;

    std::unique_ptr<Node> l, r, eq;

    tie(l, r)  = split(std::move(root), key);
    tie(l, eq) = split(std::move(l), key - 1);

    eq = merge(std::move(eq->l), std::move(eq->r)); // Remove node with max prioirity
    root = merge(merge(std::move(l), std::move(eq)), std::move(r));
}
```

Listing 5: The `erase` function.[5]

2.3 Indexing

A treap provides random access (Listing 6) in $O(\log n)$ time. This is possible by storing the size of the subtrees of each node and then walking down the tree until the specified index is reached.

```
T at(size_t index) {
    Node* current = root.get();

    while (current != NULL) {
        size_t leftSize = (current->l != NULL ? current->l->size+1 : 0);

        if (index < leftSize) {
            current = current->l.get();
        } else if (index > leftSize) {
            current = current->r.get();
            index -= leftSize + 1;
        } else {
            return current->key;
        }
    }

    throw std::out_of_range("");
}
```

Listing 6: The `at` function.

2.4 Bulk Operations

Treaps also support several efficient bulk operations: union, intersection and set difference. The time complexity of these bulk operations is $O(m \log \frac{n}{m})$, where m and n are the treap sizes and $m \leq n$.

3 Analysis

Due to the random nature of a treap the expected depth of a node in a tree is less than $2 \ln n = O(\log n)$, but the worst case depth is $O(n)$.[6] Both the split and merge operation have a running time proportional to the height of the treap because they only traverse the path from the root down to the node being split / the node where it was split. Because the expected depth of any node is $O(\log n)$, the expected run time of split and merge is $O(\log n)$.

Operation	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Indexing	$O(\log n)$	$O(n)$

Table 1: Time complexity of a treap in big O notation.

4 Motivation

During a normal programming contest you have 5 hours of time to solve three to four tasks. Due this tightly limited amount of time it is important to be able to implement algorithms fast and without bugs. When it comes to implementing a balanced BST, there are many options to choose from: 2-3 tree, red-black tree, scapegoat tree and many others. But by far the simplest to implement is the treap. The implementation of all other BSTs is way to complex and most of the times, the performance improvement due the guaranteed maximal height of $\log n$ isn't event necessary.[7]

If, for example, you wanted to maintain an ordered set of elements, you could use the `std::set` provided by the C++ standard library. But since a `std::set` only provides "random access" in $O(n)$ time, you sometimes have to implement your own set. In that case a treap is the simplest way to implement an ordered set with random access in $O(\log n)$ time.

References

[1] Wikipedia contributors. Treap, . URL `https://en.wikipedia.org/wiki/Treap`.

[2] Apple Juice Teaching. Data structures: Treaps explained. URL https://www.youtube.com/watch?v=6podLUYinH8.

[3] Wikipedia contributors. Tree rotation, . URL https://en.wikipedia.org/wiki/Tree_rotation.

[4] Ramasamy. File:tree rotation.png. URL https://commons.wikimedia.org/wiki/File:Tree_rotation.png.

[5] ABDELKARIM0. Treap simple implementation. URL https://acmcairoscience.wordpress.com/2016/03/31/treap-simple-implementation/.

[6] Margaret Reid-Miller. Lecture 16 — treaps; augmented bsts. URL http://www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/lectures/lecture16.pdf.

[7] Paul Shved. Treap: simple balanced search tree. URL http://coldattic.info/post/4/.