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

Randomized Heap

A randomized heap is a heap that, through using randomization, allows to perform all operations in expected logarithmic time.

A **min heap** is a binary tree in which the value of each vertex is less than or equal to the values of its children. Thus the minimum of the tree is always in the root vertex.

A max heap can be defined in a similar way: by replacing less with greater.

The default operations of a heap are:

- Adding a value
- Extracting the minimum
- Removing the minimum
- Merging two heaps (without deleting duplicates)
- Removing an arbitrary element (if its position in the tree is known)

A randomized heap can perform all these operations in expected $O(\log n)$ time with a very simple implementation.

Data structure

We can immediately describe the structure of the binary heap:

```
struct Tree {
    int value;
    Tree * l = nullptr;
    Tree * r = nullptr;
};
```

In the vertex we store a value. In addition we have pointers to the left and right children, which are point to null if the corresponding child doesn't exist.

Operations

It is not difficult to see, that all operations can be reduced to a single one: **merging** two heaps into one. Indeed, adding a new value to the heap is equivalent to merging the heap with a heap consisting of a single vertex with that value. Finding a minimum doesn't require any operation at all - the minimum is simply the value at the root. Removing the minimum is equivalent to the result of merging the left and right children of the root vertex. And removing an arbitrary element is similar. We merge the children of the vertex and replace the vertex with the result of the merge.

So we actually only need to implement the operation of merging two heaps. All other operations are trivially reduced to this operation.

Let two heaps T_1 and T_2 be given. It is clear that the root of each of these heaps contains its minimum. So the root of the resulting heap will be the minimum of these two values. So we compare both values, and use the smaller one as the new root. Now we have to combine the children of the selected vertex with the remaining heap. For this we select one of the children, and merge it with the remaining heap. Thus we again have the operation of merging two heaps. Sooner of later this process will end (the number of such steps is limited by the sum of the heights of the two heaps)

To achieve logarithmic complexity on average, we need to specify a method for choosing one of the two children so that the average path length is logarithmic. It is not difficult to guess, that we will make this decision **randomly**. Thus the implementation of the merging operation is as follows:

```
Tree* merge(Tree* t1, Tree* t2) {
    if (!t1 || !t2)
        return t1 ? t1 : t2;
    if (t2->value < t1->value)
        swap(t1, t2);
    if (rand() & 1)
        swap(t1->l, t1->r);
    t1->l = merge(t1->l, t2);
    return t1;
}
```

Here first we check if one of the heaps is empty, then we don't need to perform any merge action at all. Otherwise we make the heap `t1` the one with the smaller value (by swapping `t1` and `t2` if necessary). We want to merge the left child of `t1` with `t2`, therefore we randomly swap the children of `t1`, and then perform the merge.

Complexity

We introduce the random variable $h(T)$ which will denote the **length of the random path** from the root to the leaf (the length in the number of edges). It is clear that the algorithm `merge` performs $O(h(T_1) + h(T_2))$ steps. Therefore to understand the complexity of the operations, we must look into the random variable $h(T)$.

Expected value

We assume that the expectation $h(T)$ can be estimated from above by the logarithm of the number of vertices in the heap:

$$\mathbf{E} h(T) \le \log(n+1)$$

This can be easily proven by induction. Let L and R be the left and the right subtrees of the root T, and n_L and n_R the number of vertices in them ($n = n_L + n_R + 1$).

The following shows the induction step:

$$\mathbf{E} h(T) = 1 + \frac{\mathbf{E} h(L) + \mathbf{E} h(R)}{2} \le 1 + \frac{\log(n_L+1)\log(n_R+1)}{2}$$

$$= 1 + \log\sqrt{(n_L+1)(n_R+1)} = \log 2\sqrt{(n_L+1)(n_R+1)}$$

$$\le \log\frac{2\left((n_L+1)+(n_R+1)\right)}{2} = \log(n_L+n_R+2) = \log(n+1)$$

Exceeding the expected value

Of course we are still not happy. The expected value of $h(T)$ doesn't say anything about the worst case. It is still possible that the paths from the root to the vertices is on average much greater than $\log(n+1)$ for a specific tree.

Let us prove that exceeding the expected value is indeed very small:

$$P\{h(T) > (c+1)\log n\} < \frac{1}{n^c}$$

for any positive constant c.

Here we denote by P the set of paths from the root of the heap to the leaves where the length exceeds $(c+1)\log n$. Note that for any path p of length $|p|$ the probability that it will be chosen as random path is $2^{-|p|}$. Therefore we get:

$$P\{h(T) > (c+1)\log n\} = \sum_{p \in P} 2^{-|p|} < \sum_{p \in P} 2^{-(c+1)\log n} = |P| n^{-(c+1)} \le n^{-c}$$

Complexity of the algorithm

Thus the algorithm `merge`, and hence all other operations expressed with it, can be performed in $O(\log n)$ on average.

Moreover for any positive constant ϵ there is a positive constant c, such that the probability that the operation will require more than $c \log n$ steps is less than $n^{-\epsilon}$ (in some sense this describes the worst case behavior of the algorithm).

Contributors:
jakobkogler (93.8%) adamant-pwn (4.65%) Kakalinn (1.55%)