Chapter 11

Quick Sort with High Probability

By Sariel Har-Peled, April 26, 2022[1]

11.1. QuickSort runs in $O(n \log n)$ time with high probability

Consider a set T of the n items to be sorted, and consider a specific element $t \in T$. Let X_i be the size of the input in the ith level of recursion that contains t. We know that $X_0 = n$, and

$$\mathbb{E}\left[X_i \,\middle|\, X_{i-1}\right] \leq \frac{1}{2}\frac{3}{4}X_{i-1} + \frac{1}{2}X_{i-1} \leq \frac{7}{8}X_{i-1}.$$

Indeed, with probability $1/2$ the pivot is the middle of the subproblem; that is, its rank is between $X_{i-1}/4$ and $(3/4)X_{i-1}$ (and then the subproblem has size $\leq X_{i-1}(3/4)$), and with probability $1/2$ the subproblem might has not shrank significantly (i.e., we pretend it did not shrink at all).

Now, observe that for any two random variables we have that $\mathbb{E}[X] = \mathbb{E}_y[\mathbb{E}[X \,|\, Y = y]]$, see Lemma ??$_{\text{p??}}$.. As such, we have that

$$\mathbb{E}[X_i] = \mathop{\mathbb{E}}_{y}\left[\mathbb{E}\left[X_i \,\middle|\, X_{i-1} = y\right]\right] \leq \mathop{\mathbb{E}}_{X_{i-1}=y}\left[\frac{7}{8}y\right] = \frac{7}{8}\mathbb{E}[X_{i-1}] \leq \left(\frac{7}{8}\right)^i \mathbb{E}[X_0] = \left(\frac{7}{8}\right)^i n.$$

In particular, consider $M = 8\log_{8/7} n$. We have that

$$\mu = \mathbb{E}[X_M] \leq \left(\frac{7}{8}\right)^M n \leq \frac{1}{n^8}n = \frac{1}{n^7}.$$

Of course, t participates in more than M recursive calls, if and only if $X_M \geq 1$. However, by Markov's inequality (Theorem 11.5.1), we have that

$$\mathbb{P}\left[\begin{array}{c}\text{element } t \text{ participates}\\ \text{in more than } M \text{ recursive calls}\end{array}\right] \leq \mathbb{P}[X_M \geq 1] \leq \frac{\mathbb{E}[X_M]}{1} \leq \frac{1}{n^7},$$

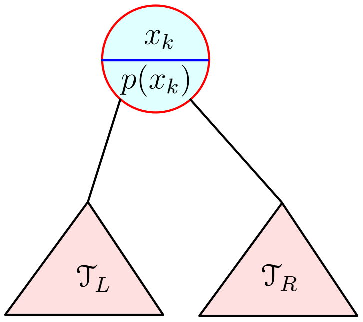
as desired. That is, we proved that the probability that any element of the input T participates in more than M recursive calls is at most $n(1/n^7) \leq 1/n^6$.

Theorem 11.1.1. *For n elements, QuickSort runs in $O(n \log n)$ time, with high probability.*

11.2. Treaps

Anybody that ever implemented a balanced binary tree, knows that it can be very painful. A natural question, is whether we can use randomization to get a simpler data-structure with good performance.

[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



11.2.1. Construction

The key observation is that many of data-structures that offer good performance for balanced binary search trees, do so by storing additional information to help in how to balance the tree. As such, the key Idea is that for every element x inserted into the data-structure, randomly choose a priority $p(x)$; that is, $p(x)$ is chosen uniformly and randomly in the range $[0, 1]$.

So, for the set of elements $X = \{x_1, \dots, x_n\}$, with (random) priorities $p(x_1), \dots, p(x_n)$, our purpose is to build a binary tree which is "balanced". So, let us pick the element x_k with the lowest priority in X, and make it the root of the tree. Now, we partition X in the natural way:

(A) L: set of all the numbers smaller than x_k in X, and
(B) R: set of all the numbers larger than x_k in X.

We can now build recursively the trees for L and R, and let denote them by T_L and T_R. We build the natural tree, by creating a node for x_k, having T_L its left child, and T_R as its right child.

We call the resulting tree a ***treap***. As it is a tree over the elements, and a heap over the priorities; that is, TREAP = TREE + HEAP.

Lemma 11.2.1. *Given n elements, the expected depth of a treap T defined over those elements is $O(\log(n))$. Furthermore, this holds with high probability; namely, the probability that the depth of the treap would exceed $c \log n$ is smaller than $\delta = n^{-d}$, where d is an arbitrary constant, and c is a constant that depends on d.*[2]

Furthermore, the probability that T has depth larger than $ct \log(n)$, for any $t \geq 1$, is smaller than n^{-dt}.

Proof: Observe, that every element has equal probability to be in the root of the treap. Thus, the structure of a treap, is identical to the recursive tree of **QuickSort**. Indeed, imagine that instead of picking the pivot uniformly at random, we instead pick the pivot to be the element with the lowest (random) priority. Clearly, these two ways of choosing pivots are equivalent. As such, the claim follows immediately from our analysis of the depth of the recursion tree of **QuickSort**, see Theorem 11.1.1. ■

11.2.2. Operations

The following innocent observation is going to be the key insight in implementing operations on treaps:

Observation 11.2.2. *Given n distinct elements, and their (distinct) priorities, the treap storing them is uniquely defined.*

[2] That is, if we want to decrease the probability of failure, that is δ, we need to increase c.

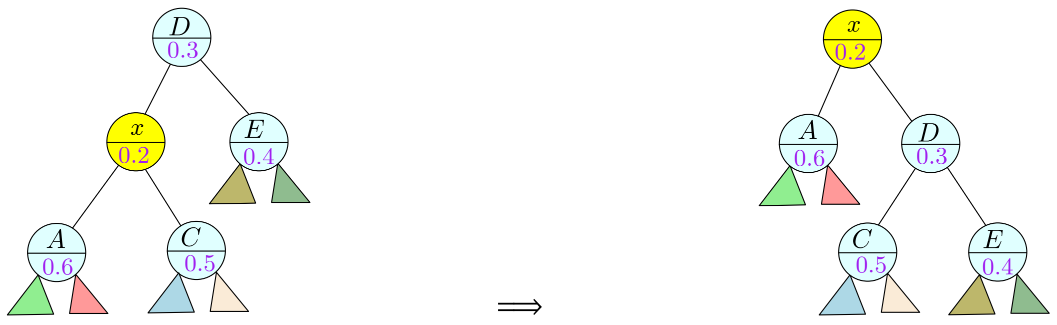


Figure 11.1: **RotateRight**: Rotate right in action. Importantly, after the rotation the priorities are ordered correctly (at least locally for this subtree.

11.2.2.1. Insertion

Given an element x to be inserted into an existing treap T, insert it in the usual way into T (i.e., treat it a regular search binary tree). This takes $O(\text{height}(T))$. Now, x is a leaf in the treap. Set x priority $p(x)$ to some random number $[0, 1]$. Now, while the new tree is a valid search tree, it is not necessarily still a valid treap, as x's priority might be smaller than its parent. So, we need to fix the tree around x, so that the priority property holds.

```
RotateUp($x$)
    $y \leftarrow$ parent($x$)
    while $p(y) > p(x)$ do
        if $y$.left_child $= x$ then
            RotateRight($y$)
        else
            RotateLeft($y$)
        $y \leftarrow$ parent($x$)
```

We call **RotateUp**(x) to do so. Specifically, if x parent is y, and $p(x) < p(y)$, we will rotate x up so that it becomes the parent of y. We repeatedly do it till x has a larger priority than its parent. The rotation operation takes constant time and plays around with priorities, and importantly, it preserves the binary search tree order. A rotate right operation **RotateRight**(D) is depicted in Figure 11.1. **RotateLeft** is the same tree rewriting operation done in the other direction.

Observe that as x is being rotated upwards, the priority properties are being fixed – in particular, as demonstrated in Figure 11.1, nodes are being hanged on nodes that were previously their ancestors, so priorities are still monotonically decreasing along a path.

In the end of this process, both the ordering property and the priority property holds. That is, we have a valid treap that includes all the old elements, and the new element. By Observation 11.2.2, since the treap is uniquely defined, we have updated the treap correctly. Since every time we do a rotation the distance of x from the root decrease by one, it follows that insertions takes $O(\text{height}(T))$.

11.2.2.2. Deletion

Deletion is just an insertion done in reverse. Specifically, to delete an element x from a treap T, set its priority to $+\infty$, and rotate it down it becomes a leaf. The only tricky observation is that you should rotate always so that the child with the lower priority becomes the new parent. Once x becomes a leaf deleting it is trivial - just set the pointer pointing to it in the tree to null.

11.2.2.3. Split

Given an element x stored in a treap T, we would like to split T into two treaps – one treap $T_\leq$ for all the elements smaller or equal to x, and the other treap $T_>$ for all the elements larger than x. To this end, we set x priority to $-\infty$, fix the priorities by rotating x up so it becomes the root of the treap. The right child of x is the treap $T_>$, and we disconnect it from T by setting x right child pointer to null. Next, we restore x to its real priority, and rotate it down to its natural location. The resulting treap is $T_\leq$. This again takes time that is proportional to the depth of the treap.

11.2.2.4. Meld

Given two treaps T_L and T_R such that all the elements in T_L are smaller than all the elements in T_R, we would like to merge them into a single treap. Find the largest element x stored in T_L (this is just the element stored in the path going only right from the root of the tree). Set x priority to $-\infty$, and rotate it up the treap so that it becomes the root. Now, x being the largest element in T_L has no right child. Attach T_R as the right child of x. Now, restore x priority to its original priority, and rotate it back so the priorities properties hold.

11.2.3. Summery

Theorem 11.2.3. *Let T be a treap, initialized to an empty treap, and undergoing a sequence of $m = n^c$ insertions, where c is some constant. The probability that the depth of the treap in any point in time would exceed $d \log n$ is $\leq 1/n^f$, where d is an arbitrary constant, and f is a constant that depends only c and d.*

In particular, a treap can handle insertion/deletion in $O(\log n)$ time with high probability.

Proof: Since the first part of the theorem implies that with high probability all these treaps have logarithmic depth, then this implies that all operations takes logarithmic time, as an operation on a treap takes at most the depth of the treap.

As for the first part, let $T_1, \ldots, T_m$ be the sequence of treaps, where T_i is the treap after the ith operation. Similarly, let X_i be the set of elements stored in T_i. By Lemma 11.2.1, the probability that T_i has large depth is tiny. Specifically, we have that

$$\alpha_i = \mathbb{P}[\mathrm{depth}(T_i) > tc' \log n^c] = \mathbb{P}\left[\mathrm{depth}(T_i) > c't\left(\frac{\log n^c}{\log |T_i|}\right) \cdot \log |T_i|\right] \leq \frac{1}{n^{t \cdot c}},$$

as a tedious and boring but straightforward calculation shows. Picking t to be sufficiently large, we have that the probability that the ith treap is too deep is smaller than $1/n^{f+c}$. By the union bound, since there are n^c treaps in this sequence of operations, it follows that the probability of any of these treaps to be too deep is at most $1/n^f$, as desired. ■

11.3. Extra: Sorting Nuts and Bolts

Problem 11.3.1 (Sorting Nuts and Bolts). You are given a set of n nuts and n bolts. Every nut have a matching bolt, and all the n pairs of nuts and bolts have different sizes. Unfortunately, you get the nuts and bolts separated from each other and you have to match the nuts to the bolts. Furthermore, given a nut and a bolt, all you can do is to try and match one bolt against a nut (i.e., you can not compare two nuts to each other, or two bolts to each other).

When comparing a nut to a bolt, either they match, or one is smaller than other (and you known the relationship after the comparison).

How to match the n nuts to the n bolts quickly? Namely, while performing a small number of comparisons.

The naive algorithm is of course to compare each nut to each bolt, and match them together. This would require a quadratic number of comparisons. Another option is to sort the nuts by size, and the bolts by size and then "merge" the two ordered sets, matching them by size. The only problem is that we can not sorts only the nuts, or only the bolts, since we can not compare them to each other. Indeed, we sort the two sets simultaneously, by simulating QuickSort. The resulting algorithm is depicted on the right.

```
MatchNuts&Bolts ($N$: nuts, $B$: bolts)
  Pick a random nut $n_{pivot}$ from $N$
  Find its matching bolt $b_{pivot}$ in $B$
  $B_L \leftarrow$ All bolts in $B$ smaller than $n_{pivot}$
  $N_L \leftarrow$ All nuts in $N$ smaller than $b_{pivot}$
  $B_R \leftarrow$ All bolts in $B$ larger than $n_{pivot}$
  $N_R \leftarrow$ All nuts in $N$ larger than $b_{pivot}$
  MatchNuts&Bolts($N_R$,$B_R$)
  MatchNuts&Bolts($N_L$,$B_L$)
```

11.3.1. Running time analysis

Definition 11.3.2. Let $\mathcal{RT}$ denote the random variable which is the running time of the algorithm. Note, that the running time is a random variable as it might be different between different executions on the *same input.*

Definition 11.3.3. For a randomized algorithm, we can speak about the expected running time. Namely, we are interested in bounding the quantity $\mathbb{E}[\mathcal{RT}]$ for the worst input.

Definition 11.3.4. The ***expected running-time*** of a randomized algorithm for input of size n is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbb{E}[\mathcal{RT}(U)],$$

where $\mathcal{RT}(U)$ is the running time of the algorithm for the input U.

Definition 11.3.5. The ***rank*** of an element x in a set S, denoted by $\text{rank}(x)$, is the number of elements in S of size smaller or equal to x. Namely, it is the location of x in the sorted list of the elements of S.

Theorem 11.3.6. *The expected running time of* MatchNuts&Bolts *(and thus also of* QuickSort*) is $T(n) = O(n \log n)$, where n is the number of nuts and bolts. The worst case running time of this algorithm is $O(n^2)$.*

Proof: Clearly, we have that $\mathbb{P}\big[\text{rank}(n_{pivot}) = k\big] = \frac{1}{n}$. Furthermore, if the rank of the pivot is k then

$$\begin{aligned} T(n) &= \mathop{\mathbb{E}}_{k=\text{rank}(n_{pivot})}[O(n) + T(k-1) + T(n-k)] = O(n) + \mathop{\mathbb{E}}_{k}[T(k-1) + T(n-k)] \\ &= O(n) + \sum_{k=1}^{n} \mathbb{P}[Rank(Pivot) = k] * (T(k-1) + T(n-k)) \\ &= O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k)), \end{aligned}$$

by the definition of expectation. It is not easy to verify that the solution to the recurrence $T(n) = O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k))$ is $O(n \log n)$. ■

11.4. Bibliographical Notes

Treaps were invented by Siedel and Aragon [SA96]. Experimental evidence suggests that Treaps performs reasonably well in practice, despite their simplicity, see for example the comparison carried out by Cho and Sahni [CS00]. Implementations of treaps are readily available. An old implementation I wrote in `C` is available here: `http://valis.cs.uiuc.edu/blog/?p=6060`.

11.5. From previous lectures

Theorem 11.5.1 (Markov's Inequality). *Let Y be a random variable assuming only non-negative values. Then for all $t > 0$, we have*

$$\mathbb{P}\bigl[Y \geq t\bigr] \leq \frac{\mathbb{E}\bigl[Y\bigr]}{t}.$$

References

[CS00] S. Cho and S. Sahni. *A new weight balanced binary search tree. Int. J. Found. Comput. Sci.*, 11(3): 485–513, 2000.

[SA96] R. Seidel and C. R. Aragon. *Randomized search trees. Algorithmica*, 16: 464–497, 1996.