Treaps, Verification, and String Matching

October 14, 2003

1 Treaps

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

A natural example for this is skip-lists (which you probably already saw before).

I find SkipList not elegant.

**Observation 1.1** Red-black tree uses additional information to make a tree balanced. Can we somehow generate such information randomly, and get a simple balanced tree?

Idea: For every element $x$ inserted into the DS, randomly choose a priority $p(x)$ which is random real number between 0 and 1.

So, let assume that we have elements:

$X = \{x_1, \ldots, x_n\}$

with (random) priorities $p(x_1), \ldots, p(x_n)$.

We want to construct a binary tree which is “balanced” for those numbers.

So, let us pick the number $x_k$ of lowest priority, and make it the root.

$L$ - numbers smaller than $x_k$ in $X$

$R$ - be all the numbers larger than $x_k$ in $X$

We can now build recursively the tree:

We call the resulting data structure a **treap**

. Why? Well it is a tree over the elements, and a heap over the priorities. So:

**treap=tree+heap**.

Q: What is the expected depth of a treap of $n$ elements?
A: logarithmic with high probability, since we can apply the proof of the quicksort recursion depth to the treap.

**Lemma 1.2** Given \( n \) elements, the expected depth of a treap 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 \( n^{-d} \), where \( d \) is an arbitrary constant, and \( c \) is a constant that depends on \( c \).

**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. As such, the claim follows immediately from our analysis of the depth of the recursion tree of QuickSort.

Q: How to do insertions?

Idea: When inserting an element \( y \) pick randomly its priority \( p(y) \).

**Observation 1.3** Once the priorities of the elements are determined, the structure of the treap is uniquely defined.

So... We “just” need to move \( y \) to its rightful place.

Well, we need a way manipulate trees: rotations.

![Diagram of a treap](image)

Q: What is problematic with this rotation?

A: The priorities of the treap are violated

So, how to perform an insertion?

Set \( p(y) = \infty \). Insert it into the treap \( T \), as you would insert \( y \) into a regular tree. At this point in time, the new tree \( T' \) is a valid treap, because the \( y \) must be a leaf as its priority is high. Next, start decreasing the priority of \( y \) till the heap condition is violated. Next, use rotations to move it up. Continue, till \( p(y) \) reach the initial (random) priority assigned to it. It is easy to verify that at this point we have a valid treap.

Q: How to handle deletions?

A: Set priority to \( \infty \) and then perform rotations till it becomes a leaf. Once it is a leaf, just delete it.

**Theorem 1.4** Let \( T \) be a treap, initialized to an empty treap, and undergoing a sequence of \( m = n^c \) insertions. The probability that the depth of the treap in any point in time would exceed \( d \log n \) is \( \leq 1/n^f \), where \( f \) is a constant that depends on \( c \) and \( d \).

In particular, a treap can handle insertion/deletion in \( O(\log n) \) time with high probability.
Proof: By the above observation, in fact represents $m$ treaps: $T_1, \ldots, T_m$. Each one of them has logarithmic depth with probability $\geq (1 - n^{-f}/m)$. As such, all of them have logarithmic depth with probability:

$$\Pr[\text{one of } T_1, \ldots, T_m \text{ is bad}] \leq \sum_{i=1}^{m} \Pr[T_i] \leq m \cdot \frac{n^{-f}}{m} \leq n^{-f}.$$ 

\[ \square \]

2 Verifying Product of Matrices

Problem 2.1 $A, B, C$ : given matrices of size $n \times n$. The claim is that $C = AB$.

Q: How to verify this claim?

A: We can of course, compute $AB$ but that would be expensive, as the fastest algorithms takes more than quadratic time ($O(n^{2.376})$). Can we do better?

Consider, $D = AB - C$. If the claim is correct, then $D$ is the zero matrix.

Note, that multiplying $D$ by a vector $v$ takes only $O(n^2)$ time.

$$Dv = (AB - C)v = ABv - Cv = A(Bv) - Cv$$

All the operations used in the last expression, can be computed in quadratic time.

So, let us choose $v$ randomly from $\{0, 1\}^n$.

Lemma 2.2 For a random $v$, the probability that $Dv = 0$ if $D$ is not zero, is smaller than $1/2$.

Proof: Consider a row $r$ in the matrix $D$ that is not zero, and let $r_i$ be an entry in $r$ which is not zero. Now, by definition, $r \cdot v$ is one of the numbers in the vector $Dv$.

We claim,

$$\Pr[r \cdot v \neq 0 | D \neq 0] \geq 1/2.$$ 

Indeed, imagine that we had determined all the values of $v$ except $v_i$. Then, $\alpha = \sum_{k \neq i} r_kv_k$.

Now, if $v_i$ is zero, then $r \cdot v = \alpha$. Otherwise, $v_i = 1$, and then $\beta = r \cdot v = \alpha + r_i$. Since $r_i$ is non-zero, either $\alpha$ or $\beta$ is not zero. Each one of those values has probability half. Thus, with probability at least half, we return a non-zero value.

Lemma 2.3 Given $A, B$ and $C$ as above, such that $C \neq AB$ then one can discover this in $O(n^2)$ with probability at least half. If the algorithm decides they are not equal it is always correct. If it claims that they are equal, then it might be wrong.

Theorem 2.4 Given $A, B$ and $C$ as above, one can determine if $AB = C$ with probability $1/2^m$ of the decision being correct, where the running time is $O(mn^2)$. 


3 Comparing Numbers

Question 3.1 Given \( m \) large numbers \( x_1, \ldots, x_m \) decide if they are all unique.

If all the numbers are smaller than \( 2^n \) then comparing two of those numbers takes \( O(n) \) time.

3.1 Comparing two numbers

Let assume that we are given two large integer numbers, and we want to decide whether they are equal or not.

Let \( a, b \) be the two numbers, such that \( a, b \leq 2^n \).

Idea: Pick a random prime number \( p \) between 1 and \( n \) and compute \( a \mod p \) and \( b \mod p \). If they are equal, we would claim that they are equal. Otherwise, we know that they are not equal.

Theorem 3.2 The number of prime numbers between 1 and \( t \) is asymptotically \( t/\log t \).

We assume that we can decide quickly if a random number is prime (this would be shown in the lecture two weeks from now). Thus, finding primes is easy (randomly pick a number, and check if it is prime). After \( O(\log n) \) trials we hope to succeed.

Lemma 3.3 A number \( x \leq 2^n \) has at most \( n \) prime divisors.

Proof: \( x = p_1 \cdots p_t \) where \( p_1, \ldots, p_t \) are the prime divisors of \( x \). Clearly \( x \geq 2^t \), which implies the lemma.

Let \( \tau = \mu n \log \mu n \) for \( \mu \) a large enough.

Pick a random prime \( p \) in the range \( 1 \ldots \tau \).

Compute \( \alpha = a \mod p \) and \( \beta = b \mod p \).

Return that \( a \) is equal to \( b \) if \( \alpha = \beta \), and no otherwise.

Lemma 3.4 Given two numbers \( a, b \leq 2^n \), the probability that if \( a \neq b \) this algorithm fails, is smaller than \( O(1/\mu) \).

Proof: We want to bound the probability

\[
\Pr[\alpha = \beta | a \neq b]
\]

Namely, \( a \mod p = b \mod p \). Namely, \( (a - b) \mod p = 0 \). Namely, \( p \) divides the number \( \delta = |a - b| \).

But \( \delta \leq 2^n \), and as such, \( \delta \) has at most \( n \) divisors. So, what is the probability that \( p \) is one of those divisors?

Well, we know that the number of primes in the range \( [1, \tau] \) is

\[
\geq c \frac{\tau}{\log \tau} = c \cdot \frac{\mu n \log \mu n}{\log(\mu n \log \mu n)} \geq c \cdot \frac{\mu n \log \mu n}{2 \log(\mu n)} = c \frac{\mu n}{2}
\]

where \( c \) is a constant.

Thus, the probability that \( \alpha = \beta \) (given that \( a \neq b \)) is at most \( n/(c \mu n/2) = 2/(c \mu) \).
So, why is this interesting? Well, note that the number $p$ we are using is (roughly) the number of digits of the numbers $a$ and $b$. Namely, it is much smaller. Also, computing $a \mod p$ is easy given the binary representation of $a = a_n a_{n-1} \ldots a_0 = \sum_{i=0}^{n} a_i \cdot 2^i$. Indeed,

$$a \mod p = \sum_{i=0}^{n} a_i 2^i \mod p = \left(\sum_{i=0}^{n} a_i (2^i \mod p)\right) \mod p,$$

which can be easily computed using numbers of size $\log \tau = O(\log n \mu)$.

Given $m$ large numbers $a_1, \ldots, a_m \leq 2^n$ deciding if a pair of them is equal can be done by picking a random $p$, computing $\alpha_i = a_i \mod p$, and deciding if there is a pair of them which is equal. Note that since the $\alpha$s are small, comparing them can be done in constant time. Picking $\mu = O(\nu m^2)$ guarantees that the probability of success is at least $1 - 1/\nu$.

At this point, except for the mathematical fancy, it is unclear why this is interesting. Hopefully, the next example would be a useful application.

## 4 Pattern Matching

$X = x_1 x_2 \ldots x_n$ - string
$Y = y_1 y_2 \ldots y_m$ - pattern

Q: Is there a $j$ such that $x_{j+k} = y_k$ for $k = 1, \ldots, m$.

Namely, does $Y$ appear as a substring of $X$?

We assume that each character is either 0 or 1. Extending this to general alphabet is easy, and as such omitted.

$X(j) = x_jx_{j+1} \ldots x_{j+m-1}$

A match occurs if there is a $j$ s.t. $X(j) = Y$.

We want to find the first such match.

We can interpret $X(j)$ as a binary number in the range $0 \ldots 2^m$. So, $X(j)$ is from this point on a number. Now, all we want to know is whether one of those numbers is equal to the number $Y$.

Using the same idea as before, let

$\alpha_j = X(j) \mod p$

where $p$ is a random prime number.

Now, we want to decide if any of the $\alpha$s is equal to $\beta = y \mod p$.

Of course, once we computed the $\alpha$s this can be done in linear time. By the analysis in the previous section, by setting $\tau = O(n^2 m \log(n^2 m))$ we get that the probability of a false match is $O(1/n)$.

More interesting, is the running time in this case. We have:

$X(j + 1) = 2(X(j) - 2^{m-1}x_j) + x_{j+m}$

Thus,

$\alpha_{j+1} = X(j + 1) \mod p = (2(\alpha_j - 2^{m-1}x_j) + x_{j+m}) \mod p$

Let $L = 2^{m-1} \mod p$. We have

$\alpha_{j+1} = (2(\alpha_j - Lx_j) + x_{j+m}) \mod p.$
Which can be computed in constant time given $\alpha_j$ because all the numbers involved are small.

We conclude:

**Theorem 4.1** The above (Monte Carlo algorithm - it might return an incorrect result) for pattern matching requires $O(n + m)$ time and has a probability of failing $O(1/n)$.

Failure, mean that the algorithm finds an incorrect match. Of course, if does find a match, we can go and verify whether this match is correct or not. If the algorithm find a false match, we abandon it, and use the brute force algorithm.

Thus, we have a new algorithm that does not fail. Its expected running time is

$$O(n + m + nm \cdot \Pr[\text{Failure}]) = O(n + m)$$

because the probability of failure is $O(1/n)$, and the running time of the brute force algorithm is $O(mn)$.

**Theorem 4.2** Given a string $X$ of length $n$ and a pattern $Y$ of length $m$ one can solve the pattern matching problem for $X$ and $Y$ in expected $O(n + m)$ time.