CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 10 — February 16, 2023

Prof. Jelani Nelson *Scribe: Rohit Agarwal*

1 Recap and Overview

First, we will wrap up the final result from Fusion trees, namely state how to find the most-significant set bit (MSSB) in constant time. Then, we will discuss data structures which use hashing.

2 Most-Significant Set Bit in $O(1)$

Recall from last time that finding the most-significant bit in constant time is an important operation in implementing Fusion trees in the Word RAM model. In particular, when we compare the (imperfect) sketch of query with those of elements in a node. At a node this hinged on thinking about the indicator on branch bits in a node, B, and picking a magic $m = \sum_{i=0}^{r-1} 2^{m_i}$ with some properties, and defining

$$\left(\text{sk}(q) := ((q\&B) \times m)\&\left(\sum_{i=0}^{r-1} 2^{b_i+m_i}\right)\right) \gg (m_0 + b_0)$$

We were guaranteed by the properties of m that this sketch is only $O(r^4)$ bits long. We saw last time how to find in constant time where sk(*node*) and sk(q) agree. We will apply this principle to see this.
The main idea to take MSSB(x) is to break up x into $\sqrt{w}$ blocks, each of size $\sqrt{w}$. Call F to be the filter with a 1 every $\sqrt{w}$ bits, starting with the MSB. ANDing x and F, zeroes out every position except the MSB of each block. Furthermore, ANDing x and $\overline{F}$, where the bar is bitwise NOT, gives us a mask that keeps the rest of the things. Subtracting this from F yields a parallel comparison, wherein all nonzero blocks have leading coefficient 0 (since the subtraction had to carry) and all zero blocks have leading coefficient 1. One more bitwise NOT and mask with F makes it so the MSB of a block is 1 if and only if x had a non-MSB that was set. ORing with the original MSB check gives us z_is which indicate whether blocks of x where nonempty:

$$z = (x\&F)|(\overline{F-(x\&(\overline{F}))}\&F) = \underbrace{z_1 0\dots0}\underbrace{z_2 0\dots0}\dots\underbrace{z_{\sqrt{w}}0\dots0}$$

We use underbraces to denote blocks. Now we know which blocks are nonzero; we must extract the leading one of these. The following claim is proved in pset 3:

Claim 2.1. There exist $0 \le m' < u$, $0 \le t \le w$, and mask $0 \le s < u$ that does 'perfect squishing,' wherein

$$g = \big((z \cdot m') \gg t\big)\&s = \underbrace{0\dots0}_{w-\sqrt{w}}\underbrace{z_i}_{\sqrt{w}}$$

We next do a parallel comparison (e.g. subtraction) $P - Q$ where

$$P = \underbrace{110\ldots0}\underbrace{101\ldots0}\ldots\underbrace{100\ldots1} = \sum_{i=0}^{\sqrt{w}-1}(2^{(i+1)(\sqrt{w}+1)-1} + 2^{i+i(\sqrt{w}+1)})$$

and

$$Q = \underbrace{0g}\underbrace{0g}\ldots\underbrace{0g}$$

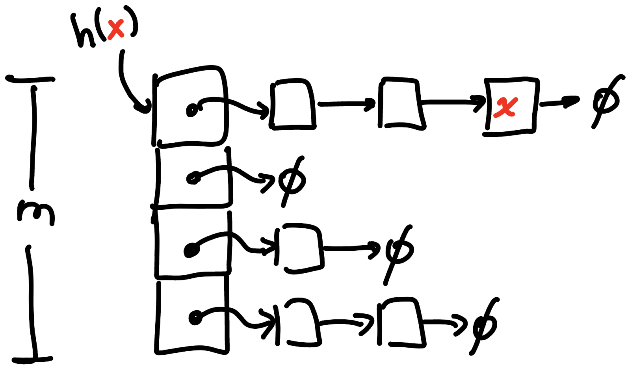
We know g lies between some powers of two, so the comparison will yield a 0 for the powers that are bigger, then yield 1 and keep yielding 1. We just need to count the number of ones, which we showed how to do last time with a multiplication by G, the filter containing blocks which all have LSB 1 and doing an appropriate shift and mask. Finally, this tells us what the MSB of g is, i.e. which z_i is 1. We have now found what block of x the MSSB is in. To finish off the operation, all we need to do is repeat the algorithm one more time on that block, which will give us our MSSB.

3 Hashing

In hashing, we use a random function $h : [u] \to [m]$, called a **hash function**.

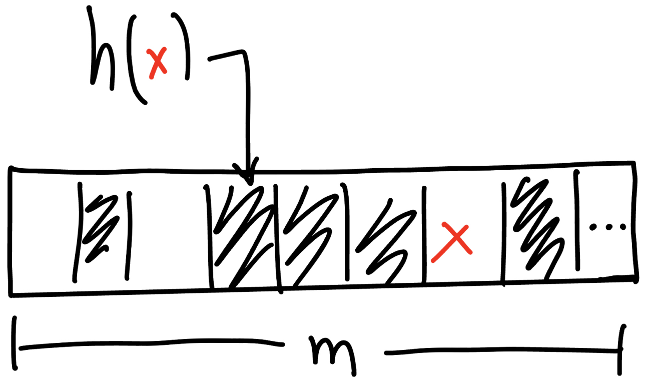
3.1 Dictionary

Suppose we are solving the dictionary problem on a universe of size u. Recall hashing with chaining; we initialize m "bins" and $h(x)$ tells you which bin the item should go in. If there is a hashing collision, where two items hash to the same thing, then we instead create a linked list with both the items. To query, you have to walk along the linked list to find your queried item.



Claim 3.1. For all $x \in [u]$, the expected time to query x is $O(1 + \frac{n}{m})$.

In static dictionary, there is a known data structure to take linear space and have constant time query. However, there is no known algorithm for this regime in the dynamic problem, nor is there a lower bound disallowing it.
However, this approach is not great for cache reasons, so instead we use linear probing. We still keep an array of size m, but when inserting x and finding a collision, we start at $h(x)$ and continue along in the array until we find an empty space. We do a similar walk for a query.



Note that if n is close to m, then one has to walk for a long time to find an empty space. In fact,

Theorem 3.2 ([1]). *In a hash table with linear probing with $m = (1+\epsilon)n$, then*

$$\mathbb{E}(\textit{query time}) = O(1/\epsilon^2)$$

3.2 Load Balancing

We discuss the load balancing problem, which is useful in the realm of cloud computing. Suppose there are m machines, n jobs and no machine is too overloaded (ideally each machine gets $\approx \frac{n}{m}$ jobs). There could be a centralized approach where we know all the jobs and just delegate them evenly. However, it's often in practice faster if there is no centralization, so we will study only assigning jobs at random. The simplest way to do this is uniformly randomly sending jobs to machines.
Let's take the case where $m = n$. Our criteria for success is that with low probability, the max load is too big. This is a tail bound, so the Chernoff bound is natural here.

Theorem 3.3 (Chernoff bound). *Let $X_1, \ldots, X_n$ be independent random variables supported on $\{0,1\}$ where $\mathbb{E}X_i = p_i$ and $X = \sum X_i$. Then, calling $\mu = \mathbb{E}X = \sum_{i=1}^n p_i$, for all $\epsilon > 0$:*

$$\mathbb{P}(X > (1+\epsilon)\mu) < \left[\frac{e^\epsilon}{(1+\epsilon)^{1+\epsilon}}\right]^\mu$$

Proof. We use the Markov bound and the idea of moment-generating functions. Introducing a parameter t and minimizing over it:

$$\begin{aligned}
\mathbb{P}(X > (1+\epsilon)\mu) &= \mathbb{P}(\exp(tX) > \exp((1+\epsilon)\mu t)) \\
&< \frac{\mathbb{E}\exp(tX)}{e^{(1+\epsilon)\mu t}} && \text{(Markov's Inequality)} \\
&= \frac{\mathbb{E}\left[\prod_i \exp(tX_i)\right]}{e^{(1+\epsilon)\mu t}} \\
&= \frac{\prod_{i=1}^n \mathbb{E}\exp(tX_i)}{e^{(1+\epsilon)\mu t}} && \text{(independence of the } X_i\text{)} \\
&= \frac{\prod_{i=1}^n \big(p_i e^t + (1-p_i)\big)}{e^{(1+\epsilon)\mu t}} \\
&= \frac{\prod_{i=1}^n \big(1 + p_i(e^t - 1)\big)}{e^{(1+\epsilon)\mu t}} \\
&\le \frac{\prod_{i=1}^n \exp\big(p_i(e^t - 1)\big)}{e^{(1+\epsilon)\mu t}} && (1 + z \le \exp(z)) \\
&= \frac{\exp\big(\mu(e^t - 1)\big)}{e^{(1+\epsilon)\mu t}} \\
&= \exp\big(\mu(e^t - 1 - t - \epsilon t)\big)
\end{aligned}$$

Taking the log and differentiating the right hand side for minimization reveals:

$$e^{t^*} - 1 - \epsilon = 0 \implies t^* = \ln(1+\epsilon)$$

Picking this value of t then yields:

$$\mathbb{P}(X > (1+\epsilon)\mu) < \exp(\mu(1+\epsilon-1-(1+\epsilon)\ln(1+\epsilon))) = \left[\frac{e^\epsilon}{(1+\epsilon)^{1+\epsilon}}\right]^\mu$$

□

There are two regimes to keep in mind when trying to understand the RHS of the Chernoff bound.

- If $\epsilon \gg 1$ (perhaps, say 2), then $(1+\epsilon) \approx \epsilon$ and one should consider e^ϵ as small

$$\mathbb{P}(X > (1+\epsilon)\mu) \lessapprox \epsilon^{-\epsilon\mu}$$

- If $\epsilon \ll 1$, we recall that by Taylor expansion $\ln(1+\epsilon) = \epsilon - \epsilon^2/2 + O(\epsilon^3)$. Then

$$\begin{aligned}
\mathbb{P}(X > (1+\epsilon)\mu) &< \frac{e^\epsilon}{(1+\epsilon)^{1+\epsilon}} \\
&= \exp(\mu(\epsilon - (1+\epsilon)\ln(1+\epsilon))) \\
&\approx \exp\left(\mu\left(\epsilon - (1+\epsilon)\left(\epsilon - \frac{\epsilon^2}{2}\right)\right)\right) \\
&\approx \exp\big(-\epsilon^2\mu/2\big)
\end{aligned}$$

We also will argue the following, which will be a useful result for load balancing.

Lemma 3.4. *If $k^k = n^p$ with $p \geq 1$, then $k = \frac{c \log n}{\log\log n}$ for some c depending on p.*

Proof. First let's find an input to $f(x) = x^x$ that undershoots the target n:

$$\begin{aligned}
\frac{k}{c} \log \frac{k}{c} &= \frac{\log n}{\log\log n} \cdot \log\left(\frac{\log n}{\log\log n}\right) \\
&= \frac{\log n}{\log\log n} \cdot (\log\log n - \log\log\log n) \\
&= \log n\left(1 - \frac{\log\log\log n}{\log\log n}\right) \\
&< p \log n
\end{aligned}$$

And furthermore, pick some K and follow the same logic, yielding:

$$\begin{aligned}
\frac{Kk}{c} \log \frac{Kk}{c} &= K \log n\left(1 - \frac{\log\log\log n - \log K}{\log\log n}\right) \\
&> K \log n\left(1 - \frac{\log\log\log n}{\log\log n}\right) \\
&> p \log n
\end{aligned}$$

for a sufficient constant K, due to the growth rates of the functions involved.
This means that there exists some constant in between where we have equality. Call c this constant. □

Now to use this for load balancing, pick a machine and define X_i as the indicator that job i maps to a machine. The load on that machine is thus $X = \sum_{i=1}^n X_i$. In expectation it is 1, since $m = n$. Pick $p = 10$ and thus $k := 1 + \epsilon = \frac{c \log n}{\log\log n}$ for some c we will fix later. Then this is much greater than 1, and the bound tells us

$$\mathbb{P}\left(X > \frac{c \log n}{\log\log n}\right) \lesssim \frac{1}{k^k} < \frac{1}{n^{10}}$$

Thus, we have vanishingly low probability of too much load for any fixed machine. Then by a union bound

$$\mathbb{P}(\exists \text{ machine with load } > \frac{c \log n}{\log\log n}) < n \cdot \frac{1}{n^{10}} = \frac{1}{n^9}$$

References

[1] Donald Knuth. Notes on "open" addressing, 1963. URL: http://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf.