15:853: Algorithms in the Real World
Fall 2019

Lecture 19

November 8, 2019

Lecturer: Rashmi Vinayak *Scribe: Christopher Canel*

19.1 Recap from last class

See the instructor notes for a recap of the basics of hashing, universal hash functions, handling collisions in hash tables, perfect hashing, k-wise independent hash functions, open addressing, and linear probing.

19.2 Overview

This lecture continues the hashing module, describing cuckoo hashing and the basics of Bloom filters.

19.3 Cuckoo hashing

Cuckoo hashing is a form of open addressing, meaning that all values are stored in the main hash table data structure (i.e., there are no secondary arrays to store the values of keys that collide).

Consider a universe $\mathcal{U}$ and a dictionary $\mathcal{S}$, where $\mathcal{S} \in \mathcal{U}$ and $N = |\mathcal{S}|$. Typically, $N \ll |\mathcal{U}|$. Consider a cuckoo hash table $\mathcal{T}$, where $M = |\mathcal{T}|$, and assume that $M = \mathcal{O}(N)$.

Let $\mathcal{H}$ be a family of ideal hash functions (the "ideal" distinction is not necessary, but makes the analysis easier). Assume $\mathcal{H}$ to be fully random. Cuckoo hashing uses two hash functions $h_1, h_2 \in \mathcal{H}$.

19.3.1 Insertion

There are several variants of the insertion algorithm, but we will analyze just one. To insert a key $x \in \mathcal{S}$ into $\mathcal{T}$:

1. Consider table locations $t_1 = \mathcal{T}[h_1(x)]$ and $t_2 = \mathcal{T}[h_2(x)]$. If either t_1 or t_2 is free, then insert x into that location.

2. If both t_1 and t_2 are occupied, then we have a collision. Pick either t_1 or t_2 and evict its current occupant, which we will call y. Insert x into the newly-vacant location.

3. Since we use two hash functions for every key, there are always two locations where a value can be stored. Consider both of the possible locations for y, $t'_1 = \mathcal{T}[h_1(y)]$ and $t'_2 = \mathcal{T}[h_2(y)]$ (recall that y will have just been evicted from either t'_1 or t'_2 and replaced by x). If either t'_1 or t'_2 is vacant, then re-insert y into that vacant location. I.e., if y's other location is vacant, then move it to that location. If both t'_1 and t'_2 are occupied, then evict one of their occupants, which we will call z, and insert y into the newly-vacant location. Recursively apply Step 3 until a free slot is eventually found.

   If any key is relocated more than once, then h_1 and h_2 are not "good enough". Rehash all keys in the table.

19.3.2 Lookup/deletion

To lookup or delete a key x, simply check both locations $\mathcal{T}[h_1(x)]$ and $\mathcal{T}[h_2(x)]$. Depending on which (if any) location x resides in and whether the two location checks are conducted serially or in parallel, the lookup/deletion process may result in a total of two hash function computations and two hash table accesses. Still, lookups and deletions are $\mathcal{O}(1)$, which is much better than for closed addressing schemes that require searching sidecar data structures in the event of a collision.

19.3.3 Analysis of insertion time

Theorem 19.1. *The expected time to performs an insert operation into a cuckoo hash table is $\mathcal{O}(1)$ if $M \geq 4N$.*

Proof (sketch, Theorem@19.1): We will use the notion of a cuckoo graph $\mathcal{G} = (V, E)$, where the vertices, V, are the locations in the cuckoo hash table ($|V| = M$) and $E = \{\forall x \in \mathcal{S}, e_x = (h_1(x), h_2(x))\}$ ($|E| = N$). I.e., for each possible key, an edge connects that key's two possible locations. Since h_1 and h_2 are random, $\mathcal{G}$ is random as well. Figure 19.1 gives a small example.

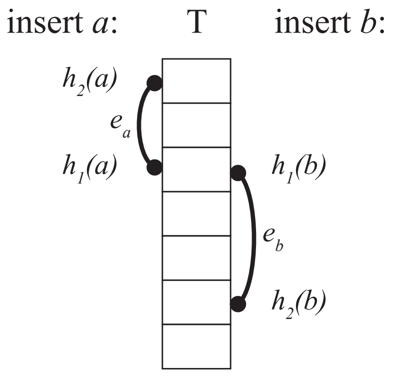

Figure 19.1. A cuckoo graph created by inserting two keys, a and b.

Furthermore, $\forall x \in \mathcal{S}$, let the *bucket* of x, $B(x)$, be the set of vertices in $\mathcal{G}$ that are reachable from either vertex $h_1(x)$ or vertex $h_2(x)$. I.e., $B(x)$ is the set of vertices that the recursive eviction process may traverse. Equivalently, $B(x)$ is the *connected component* of $\mathcal{G}$ that contains e_x. To determine the insertion time, we need to understand $|B(x)|$.

In this connected component, the number of vertices is greater than the number of edges because by definition there are no collisions, and the number of edges corresponds to the keys, the number of vertices corresponds to the hash table's capacity, and all keys must fit within the capacity.

Suppose we want to insert a key x. Then:

$$E[|\text{insertion time of } x|] = E[|B(x)|] \tag{19.1}$$

Goal of proof sketch: Show that $E[|B(x)|] \leq \mathcal{O}(1)$.

$$E[|B(x)|] = \sum_{y \in \mathcal{S}, y \neq x} P[e_y \in B(x)] \tag{19.2}$$

Apply the union bound:

$$E[|B(x)| \leq N \cdot P[e_y \in B(x)] \tag{19.3}$$

Therefore, since Section 19.3 assumes that $M = \mathcal{O}(N)$, it is sufficient to show:

$$P[e_y \in B(x)] \leq \mathcal{O}(\frac{1}{M}) \tag{19.4}$$

Lemma 19.2. *For any two locations i and j in the space of hash table locations (i.e., $\forall i, j \in [M]$):*

$$P[\exists \textit{ a path of length } l \textit{ between } i \textit{ and } j \textit{ in } \mathcal{G}] \leq \frac{1}{2^l M} \tag{19.5}$$

Proof (Lemma 19.2): Use induction on l.

Base case: $l = 1$.

$$P[\exists \text{ an edge directly between } i \text{ and } j] = P[\exists y \in \mathcal{S} \text{ s.t. } e_y \in \mathcal{G}] \tag{19.6}$$

I.e., The probability that there exists an edge directly between i and j is equal to the probability that there exists a key y such that its edge, e_y, is in $\mathcal{G}$. At an even higher level, this is the probability that the two locations that a key y can be stored at are i and j.

Explicitly, we know that $e_y = (h_1(y), h_2(y))$. For e_y to span i and j, either $h_1(y) = i$ and $h_2(y) = j$ or $h_1(y) = j$ and $h_2(y) = i$. We know that $P[h_i(y) = \alpha] = \frac{1}{M}$, so the probability of each of these two cases is $\frac{1}{M} \cdot \frac{1}{M} = \frac{1}{M^2}$, for a total probability of $\frac{2}{M^2}$ per possible value of y. There are N such vertices y because $N = |\mathcal{S}|$, therefore:

$$P[\exists \text{ an edge directly between } i \text{ and } j] = N \cdot \frac{2}{M^2} \tag{19.7}$$

Using our initial assumption that $M \geq 4N$, we know that $N \leq \frac{M}{4}$, so:

$$P[\exists \text{ an edge directly between } i \text{ and } j] \leq \frac{M}{4} \cdot \frac{2}{M^2} \tag{19.8}$$

$$\leq \frac{1}{2M} \tag{19.9}$$

Done with the base case.

Inductive step: Left as an exercise.

This proves the above lemma, showing that the probability of having a large l is small. □

Returning to the proof of $P[e_y \in B(x)] \leq \mathcal{O}(\frac{1}{M})$:

$$P[e_y \in B(x)] \leq \sum_{l \geq 1} \text{“Lemma 19.2”} \tag{19.10}$$

$$= \sum_{l \geq 1} \frac{1}{2^l M} \tag{19.11}$$

$$= \mathcal{O}(\frac{1}{M}) \tag{19.12}$$

$$P[e_y \in B(x)] \leq \mathcal{O}(\frac{1}{M}) \tag{19.13}$$

Plugging this in to (19.3):

$$E[|B(x)| \leq N \cdot \mathcal{O}(\frac{1}{M}) \tag{19.14}$$

Applying the assumption in Section 19.3 that $M = \mathcal{O}(N)$:

$$E[|B(x)| \leq \mathcal{O}(1) \tag{19.15}$$

The bucket size is $\mathcal{O}(1)$, so (19.1) states that the insertion time is $\mathcal{O}(1)$ as well. □

Another key metric is *occupancy*, which determines how large of a cuckoo hash table we will need. Our condition that $M \geq 4N$ means that all of the keys that we want to store in the hash table (S, where $N = |S|$) will fill at most 25% of our table data structure ($M = |\mathcal{T}|$). That is a lot of wasted space, so can we do better? People have experimented with using more hash functions, which provides higher occupancy (but there is no rigorous proof to that effect).

Instead of using two hash functions that map to the same table, people sometimes describe cuckoo hashing in terms of two separate hash tables, each with one hash function. A key can then be found in at most one of the two tables.

For our proofs, we need the hash functions to be $\mathcal{O}(\log(N))$-wise independent. With cuckoo hashing, simple tabulation hash functions work well, unlike in other hash tables.

19.4 Bloom filters

A **Bloom filter** is a type of hash table adapted for approximate membership queries. Bloom filters are used widely in high-performance systems to guard access to data structures with longer access times. E.g., a fast Bloom filter may be used to guard a slow database: All keys in the database are also inserted into the Bloom filter. At query time, if the Bloom filter does not contain a certain key, then the program does not need to query the slow database.

Since a Bloom filter is used solely for membership queries, it stores only keys (no values), thereby using far fewer bits than a full hash table. A Bloom filter may have false positives (it may report that it contains a key that it does not actually contain), but no false negatives (if it reports that it does not contain a key, then it definitely does not contain that key). This is a useful property for membership queries when most keys will not be in the underlying data structure (i.e., $N \ll |U|$), e.g., a web browser maintaining a list of malicious or blocked websites. However, they do not support deletion. Bloom filters may be implemented using a cuckoo hash table.

A Bloom filter consists of two parts:

1. An array $\mathcal{T}$ of M bits, initially all set to 0.
2. k hash functions, $\{h_1, h_2, \ldots, h_k\}$, where $h_i = \mathcal{U} \to [M]$. For analysis purposes, assume that the hash functions are completely random.

19.4.1 Insertion

Consider a key $x \in \mathcal{S}$. $\forall i$ s.t. $1 \leq i \leq k$, set $\mathcal{T}[h_i(x)] = 1$. I.e., set x's hash locations to 1.

19.4.2 Lookup

Consider a key $x \in \mathcal{S}$. If, $\forall i$ s.t. $1 \leq i \leq k$, $\mathcal{T}[h_i(x)] = 1$, then report x as being in the Bloom filter. I.e., report that x is in the Bloom filter if all of its hash locations are 1.

19.4.3 Analysis

Q: Why are there no false negatives? If an item is present, then the corresponding bits in $\mathcal{T}$ will be set.

Q: Why are there false positives? The bits corresponding to a key could have been set by the other keys' insertion processes. I.e., the hash functions may map multiple keys to each location.

Understanding the problem of false positives. A false positive occurs for a key x when all of the k bits corresponding to x have been set by other keys. The probability that a hash function mapped a key to a particular location is $\frac{1}{M}$, so $(1 - \frac{1}{M})$ is the probability that a hash function did not map a key to a particular location. There are k hash functions and N keys, so the location selection process occurs kN times, each instance of which is independent. Therefore, the probability that no hash function mapped a key to a particular location is:

$$P[\text{a location in } \mathcal{T} = 0] = \left(1 - \frac{1}{M}\right)^{kN} \tag{19.16}$$

To be continued in the next lecture...