Lecture 27 — Hashing II

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Margaret Reid-Miller — 26 April 2012

Today:

- Parallel Hashing
- Hashing Applications
- Hash Functions
- Recent Hash Table Concepts

1 Parallel Hashing

How might we parallelize open addressing? In the parallel context, instead of inserting, finding or deleting one key at a time, each operation takes set of keys. Since a hash function distributes keys across slots in the table, we can expect many keys will be hashed to different locations. The idea is to use open addressing in multiple rounds. For `insert`, each round attempts to write the keys into the table at their appropriate hash position. For any key that cannot be written because another key is already there, the key continues for another round using its next probe location. Rounds repeat until it writes all the keys to the table.

In order to prevent writing to a position already occupied in the table, we introduce a variant of the `inject` function. The function

$$\texttt{injectCond}(IV,S) : (\texttt{int} \times \alpha)\ \texttt{seq} \times (\alpha\ \texttt{option})\ \texttt{seq} \to (\alpha\ \texttt{option})\ \texttt{seq}$$

takes a sequence of index-value pairs $\langle (i_1,v_1),\ldots,(i_n,v_n)\rangle$ and a target sequence S and conditionally writes each value v_j into location i_j of S. In particular it writes the value only if the location is set to NONE and there is no previous equal index in IV. That is, it conditionally writes the value for the *first* occurrence of an index; recall `inject` uses the *last* occurrence of an index.

```
1  fun insert(T,K) =
2  let
3    fun insert'(T,K,i) =
4      if |K| = 0 then T
5      else let
6        val T' = injectCond({(h(k,i),k) : k ∈ K}, T)
7        val K' = {k : k ∈ K | T[h(k,i)] ≠ k}
8      in
9        insert'(T',K',i+1)      end
10 in
11   insert'(T,k,1)
12 end
```

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

For round i, `insert` uses each key's i^{th} probe in its probe sequence and attempts to write the key to the table. To see whether it successfully wrote a key to the table, it reads the value written to the table and checks if it is the same as the key. In this way it can filter out all keys that it successfully wrote to the table. It repeats the process on the keys it didn't manage to write, using the keys' $(i+1)$ probes.

For example, let's say that the table has the following entries before round i:

$T =$

0	1	2	3	4	5	6	7
	A		B			D	F

If $K = \langle E, F \rangle$ and $h(E, i)$ is 1 and $h(F, i)$ is 2, then $IV = \langle (1, E), (2, F) \rangle$ and `insert'` would fail to write E to index 1 but would succeed in writing F to index 2 resulting in the following table:

$T' =$

0	1	2	3	4	5	6	7
	A	F	B			D	F

It then repeats the process with $K' = \langle E \rangle$ and $i + 1$.

Note that if T is implemented using a single threaded array, then parallel `insert` basically does the same work as the sequential version which adds the keys one by one. The difference is that the parallel version may add keys to the table in a different order than the sequential. For example, with linear probing, the parallel version adds F first using 1 probe and then adds E at index 4 using 4 probes:

$T_P =$

0	1	2	3	4	5	6	7
	A	F	B	E		D	F

Whereas, the sequential version might add E first using 2 probes, and then F using 3 probes:

$T_S =$

0	1	2	3	4	5	6	7
	A	E	B	F		D	F

Both make 5 probes in the table. Since we showed that, with suitable hash functions and load factors, the expected cost of insert is $O(1)$, the expected work for the parallel version is $O(|K|)$. In addition, in each round, the expected size of K decreases by a constant fraction, so span is $O(\log |K|)$.

How does a hash table implementation compare with a treap implementation of Set and Table?

- search is faster (in expectation)
- insert is faster (in expectation)
- map, reduce, remain linear
- union/merge is slower or faster depending on particular settings

2 Hash Functions

Hash tables are not the only application of hash functions. There is a surprising wide range of applications where it useful to have random or pseudo random functions that map from some large

set to a smaller set. Such functions might be used to hide data, to reorder data in a random order, or to spread data smoothly over a range. Here we consider some applications of each.

1. We saw how in treaps, we used a hash function to hash the keys to generate the "random" priorities.
2. In cryptography hash functions can be used to hide information. One such applications is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.
3. A one-way hash function is used to hide a string, for example for password protection. Instead of storing passwords in plain text, only the hash of the password is stored. To verify whether a password entered is correct, the hash of the password is compared to the stored value. These signatures can be use to *authenticate* the source of the document, ensures the *integrity* of the document as any change to the document invalidates the signature, and prevents *repudiation* where the sender denies signing the document.
4. String commitment protocols use hash functions to hide to what string a sender has committed so that the receiver gets no information. Later, the sender sends a key that allows the receiver to reveal the string. In this way, the sender cannot change the string once it is committed, and the receiver can verify the revealed string is the committed string. Such protocols might be used to flip a coin across the internet: The sender flips a coin and commits the result. In the mean time the receiver calls heads or tails, and the sender then sends the key so the receiver can reveal the coin flip.
5. Hashing can be used to approximately match documents, or even parts of documents. *Fuzzy matching* hashes overlapping parts of a document and if enough of the hashes match, then it conclude that two documents are approximately the same. Big search engines look for similar documents so that on search result pages they don't show the many slight variations of the same document (e.g., in different formats). It is also used in spam detection, as spammers make slight changes to email to bypass spam filters or to push up a document's content rank on a search results page. When looking for malware, fuzzy hashing can quickly check if code is similar to known malware. Geneticists use it to compare sequences of genes fragments with a known sequence of a related organism as a way to assemble the fragments into a reasonably accurate genome.

There is a deep and interesting theory of hash functions. Depending on the application, the needs of the hash function are very different. We will not cover the theory here but you will likely see it in more advanced algorithms classes.

For hash table applications a hash function should have the following properties:

- It should distribute the keys evenly so there are not many collisions.
- It should be fast to compute.

So what are some reasonable hash functions? Here we consider some simple ones.

For hashing integers we can use

$$h(x) = (ax + b) \bmod p$$

where $a \in [1, \ldots, p-1]$, $b \in [0, \ldots, p-1]$, and p is a prime. This is called a linear congruential hash function that has some nice properties that you will likely learn about in 15-451.

For strings we can simply use a polynomial

$$h(S) = \left(\sum_{i=0}^{|S|-1} = s_i a^i \right) \bmod p$$

where $a \in [2, ..., p-1]$ and p is a prime number.

Sequentially, Horner's method avoids computing a^i explicitly. In parallel, simply use scan with multiplication. This hash function tends to mix bits from earlier characters with bits in later characters.

2.1 Universal Hash Functions

In the same way as having a deterministic strategy for picking pivots in quicksort, using a fixed hash function gives good average time behavior, but can lead to terrible performance on some inputs. There is no hash function that will behave well on all inputs (even within an application domain).

A better strategy is to pick a hash function randomly from a suitable family of hash functions at the time you create the hash table. Then for *any* set $K \subseteq U$, the hash function will behave well in expectation. Of course, if you are very unlucky, the hash function could hash K with lots of collisions. But you will not consistently be unlucky every time you create a table for K, as most hash functions in the family will lead to good hash table performance.

Definition 2.1. A finite set $\mathcal{H}$ over hash functions from U to $\{1, \ldots, m\}$ is **universal** or a**universal hash family** if for all $x \neq y$ in U, we have

$$\mathbf{Pr}\left[h(x) = h(y)\right] \leq 1/m$$

for all $h \in \mathcal{H}$

That is, the probability that x and y collide for any of the hash functions $h \in \mathcal{H}$ is the same as though x and y are randomly and independently assigned values in $[0 \ldots\ m-1]$. The number of hash functions $h \in H$ that would cause x and y to collide is exactly $|\mathcal{H}|/m$.

Theorem 2.2. *If $\mathcal{H}$ is universal, then for any set $K \subseteq U$, any $x \in K$, and h selected at random from $\mathcal{H}$, the **expected** number of collisions between x and the other elements in K is at most n/m.*

Proof. Each $y \in K$ $(y \neq x)$ has at most a $1/m$ chance of colliding with x by the definition of universal. So,

- Let random indicator variable $C_{xy} = 1$ if x and y collide and 0 otherwise.
- Let C_x denote the total number of collisions for x. So, $C_x = \sum_{y \in K, y \neq x} C_{xy}$

- We know $\mathbf{E}\left[C_{xy}\right] = \mathbf{Pr}\left[x \text{ and } y \text{ collide}\right] \leq 1/m$.
- So, by linearity of expectation, $\mathbf{E}\left[C_x\right] = \sum_y \mathbf{E}\left[C_{xy}\right] < n/m$

□

Can we construct a universal hash family $\mathcal{H}$? Yes, consider the following example.

Example of a universal hash family Suppose we can decompose a key $x = \langle x_1, x_2, \ldots, x_r \rangle$ (e.g., decompose the key into a sequence of bytes). Now pick a sequence of coefficients $a = \langle a_1, a_2, \ldots, a_r \rangle$ chosen at random from set $[0, \ldots, p-1]$. We define $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_1^r a_i x_i \mod p,$$

where p is prime. Then $\mathcal{H} = \cup_a h_a$ is a universal hash family with p^r members.

Proof. To prove that $\mathcal{H}$ is a universal hash family, we need to show that for any $h \in \mathcal{H}$, the probability that two unequal keys x and y collide is $1/p$. Consider $x \neq y$. There must be at least one component $x_i \neq y_i$. Without loss of generality, suppose $i = r$. We need to compute the probability $\mathbf{Pr}\left[h_a(x) = h_a(y)\right]$. For any fixed $\langle a_1, \ldots, a_{r-1} \rangle$ there is one a_r that satisfies equation $h_a(x) = h_a(y)$. It is a solution to

$$a_r(x_r - y_r) = (-\sum_{i=1}^{r-1} a_i(x_i - y_i)) \mod p.$$

That is, $x_r - y_r \neq 0$ has multiplicative inverse ($\mod p$), namely a_r. Now suppose we choose the random hash function h_a by first drawing $a_1, a_2, \ldots, a_{r-1}$. What is the probability that the next number a_r will be the multiplicative inverse of $x_r - y_r$? Since it is only one out of p possible values, the probability is $1/p$. □

If we happen to draw all zeros except for a few of the a values we would likely get poor hash performance as only a small part of the keys would be used to determine the hash slot. If that part is say the first 3 digits of a telephone area code, we can expect to see a lot of collisions. But the probability of picking a hash function that is so bad is very low.

2.2 Perfect hashing

Amazingly, when you know the set of keys K that want to be put in a hash table in advance, it is possible to find a hash function for which none of the keys collide. The catch is that you need a hash table with $|K|^2$ locations in order to find such a hash function in a reasonable number of attempts. Once you find it, then every find operations is guaranteed to take $O(1)$ work just like an array.

The idea is to pick a random hash function h from universal hash family $\mathcal{H}$ and simply to see if any collisions occur.

Claim 2.3. *Probability there are no collisions in a set of K keys is $\geq 1/2$.*

How many pairs of keys in K could collide? There $\binom{|K|}{2}$ pairs, and each can collide with probability $\leq n^2$ since h is from a universal hash family. The probability that at least one collision exist is $\leq \binom{|k|}{2}/|K|^2 < 1/2$. Therefore no collision occurs with probability $\geq 1/2$.

If there are collisions, try another h from $\mathcal{H}$.

Exercise 1. *: What is the expected number of hash functions you need to try before you find a perfect one?*

But using $|K|^2$ space is wasteful. The solution is to use a two level scheme to bring the table size to $O(|K|)$. To find the hash functions needed, first pick a random hash function h to hash to a table of size $|K|$, using separate chaining. Now, for each chain i of size n_i, find a perfect hash function h_i using n_i^2 space as above. As you may see in 15-451, the probability the first level hash function h has the property $\sum_i n_i^2 > 4|K|$ is $< 1/2$. If $\sum_i n_i^2$ is too big then keep trying with another h.

Therefore, we can find a perfect hashing scheme that takes $O(|K|)$ space: First look up $h(x)$ in a table that stores a second hash function h_i and an offset O_i for that subtable, and then use $h_i(x) + O_i$ to find the key in the second table, where $O_i = \sum_{k<i} n_i$. In this way, exactly two lookups finds the key.

The problem with perfect hashing is that it is static. If you add a new key, it will likely collide in its subtable. You would then need to find a new perfect hash function for that subtable and reorganize the tables. This scheme then is slow to construct, uses $O(n)$ space, and has guaranteed constant time lookup but no insertion.

Some recent advances in hashing (in theory) make it possible to get dynamic hash tables with lookup performance like perfect hashing. One scheme is an extension to separate chaining, the other to open addressing. As these are advanced topics, we discuss the overall approach without proofs.

2.3 Multiple-choice hashing

Multiple-choice hashing is a generalized separate chaining. It applies d hash functions to a key k and picks the shortest chain to which to add k.

If $d = 1$ multiple-choice hashing is simply standard separate chaining. If $n = m$ the length of longest chain is $(1 + o(1)))\log n/\log\log n$ with high probability.

If $d \geq 2$ the length of longest chain is $\log\log n/\log d + O(1)$ with high probability. For practical values of n and with $d = 2$ the length of the longest chain is only 6.

But to find a key requires searching d chains. Of course, these searches can be done in parallel. Thus, we have a hashing scheme that is fast to build and uses $O(n)$ space, and the cost for find is not only constant in expectation but, more importantly, close to constant for all lookups with high probability.

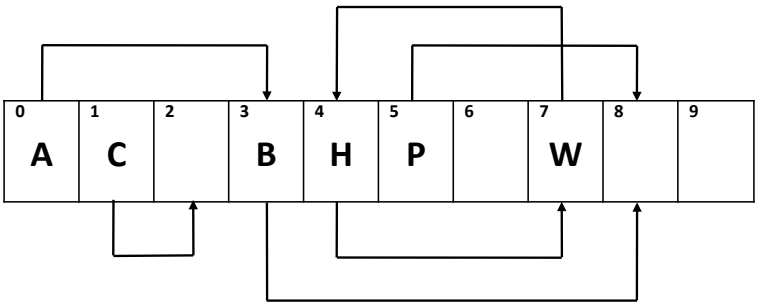
2.4 Cuckoo hashing

Cuckoo hashing was introduced by Pagh and Rodler in 2001. Like multiple-choice hashing, it uses 2 functions, but it is based open addressing. Amazingly, looking up or deleting a key requires at most 2 probes just like perfect hashing, guaranteed. But cuckoo hashing handles dynamic insertions in expected $O(1)$ work, although it can take as much as $O(\log n)$ work. The down side is that the insertion cost degrades quickly once the table is more than 50% full, and can fail completely requiring rehashing the table.

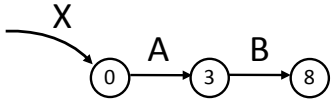
Instead of using a probe sequence that probes every position in the table (or at least half of the table in quadratic probing), Cuckoo hashing uses only two probes, $h_1(k)$ and $h_2(k)$. Again as in open addressing, it stores the key directly in the table, which means there can be only one key in each location.

What does it do if both locations are filled? As its name implies, it behaves like a nesting cuckoo chick: It kicks out the key x at $h_1(k)$ to make room for k. It then repeats the process to reinsert x at its alternative position. If its alternative position is filled then x evicts the key there. This process continues until either it finally finds a location for the latest evicted key, or it is in a cycle in which case it rehashes all the keys using new hash functions. Interestingly, you don't need to create a new table. Just rehash in place, deleting and inserting keys that are not in their intended place.

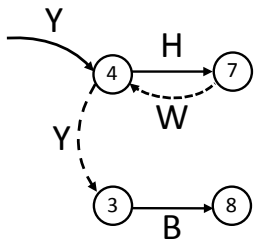
Because keys can be at only two locations, finding a key requires at most two probes, guaranteed. Deletion is easy also, just locate the key and delete it, perhaps moving to its h_1 location–no dummy hold values are needed.



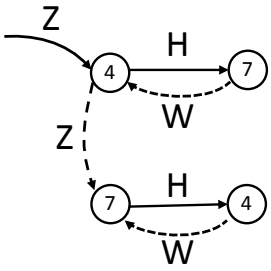
Cuckoo hashing induces a graph $G = (V, E)$ where the vertices are the positions in the table, $V = \langle 0, 1, 2, \ldots, m-1 \rangle$ and edges link two vertices if a key hashes to the two vertices, $(h_1(k), h_2(k) \in E$. For example, the figure above shows a cuckoo graph, where the arrows show the alternate position of the keys. Suppose we want to insert a new key X and $h_1(X) = 0$ and $h_2(X) = 5$. Since both positions are occupied, X is inserted at position 0, kicking out A. Then A is inserted at its alternate position 3, kicking out B. Finally, B is inserted at its alternate position 8. That is, X moves A, which moves B. The walk is shown below.



Notice that the graph can have cycles: keys H and W hash to the same two positions. A few cycles may not be a problem, as each key has an alternate path to follow. For example, suppose instead we want to insert key Y and it has $h_1(Y) = 4$ and $h_2(Y) = 3$. Then Y goes to 4, H goes to 7, W goes to back to 4 and evicts Y, this time Y goes to its alternative hash position 3, and evicts B which goes to 8.



One the other hand, if both hash positions lead to a cycle, the key cannot be inserted and the table needs to be rehashed with two new hash functions. For example, let's say we insert key Z with $h_1(Z) = 4$ and $h_2(Z) = 7$. Then Z goes to 4, H goes to 7, W goes to 4, as shown below. Then Z tries its alternate 7, which pushes H back to 4, and W back to 7, and the cycle repeats as keys have returned to their original positions. In this case the table needs to be rehashed with two new hash functions.



It is possible to show that when the load factor is less than $1/2$ the expected number of evictions needed to insert a new key is constant and the longest path is $O(\log n)$. Insertion will likely succeed. Unfortunately, cycles are likely when $\alpha \geq 1/2$. The analysis is base on random graphs. When the number of edges is small the likelihood of a cycle is small. But once there more than $m/2$ edges, there is likely at least one cycle. As more edges are added the number of cycles increases rapidly and the longest cycle is large.

That is, there is a sharp threshold at $\alpha = 1/2$ where below $1/2$ insertion performance is reasonable, and when above $1/2$ insertion has high failure rate. By using 3 hash functions, though, this threshold can be move to 91% full, based on experiments.

Parallel cuckoo hashing is similar to parallel open addressing. Finding keys is easy, since cuckoo hashing knows exactly where to look for the keys. To insert it first swaps the keys with the keys in the table. That is, before writing a key to the table, it first gets the key in the table (or NONE), and then uses `inject` to write the new key. Those keys that don't successfully write to the table (as they hash to the same location) and any keys swapped out of the table go another round. For each key, if it used hash function i in last round, it uses hash function $i + 1$ to reinsert itself in the next

round. Note that the values of i may be different for different keys. To keep collisions low, three or four hash functions might be used instead of just two. After $O(\log n)$ rounds, if some keys have been unsuccessful, the table needs to be rehashed (rare).

A recent experiment on GPUs compared parallel cuckoo hashing using four hash functions, separate chaining, and quadratic probing. Since the separate chaining algorithm used arrays, the table could not grow without rebuilding it. Searching with cuckoo hashing was notably faster than separate chaining when the load factor was high (95%) or the tables were small. Whereas building a table using separate chaining was faster than cuckoo hashing. For load factors below 50% quadratic probing performed best for both building and querying, unless the percent of queries that are unsuccessful is high ($\geq$ 80%), in which case separate chaining was better.

In summary cuckoo hashing, like perfect hashing, has that advantage that it needs to hash to only a constant number positions to find or delete a key in the worst case. Building the hash table can be somewhat slow, as each insertion may need to move $O(1)$ keys on average and with a non-trivial probability it needs $O(logn)$ number of moves. Insertion can fail (with low probability) because it can get into a cycle, which may take a long time to detect and requires rebuilding the table. Thus cuckoo behaves like perfect hashing, but can handle insertions gracefully.

In addition cuckoo hashing is adaptable in a number of ways. If the load becomes large, it can dynamically increase the number of hash functions to increase the capacity. It can stop temporarily in the middle of cascading moves, as the table is always in a consistent state, as long as it keeps a copy of the last item evicted. It can move frequently accessed items to their first hash function. Even better, it can grow the table without fully rehashing the table all at once: It marks old locations for lookup only and adds new hash functions that map to the larger table. To find the key, use both the new and old hash functions. Every time it finds a key using the old hash function, it moves it to the new hash function positions. Once all keys are migrated (keep a counter) retire the lookup-only hash functions.