Notes on Hashing

I am skipping the introductions and jumping to the more interesting parts.

Definitions and Notations

- $\mathcal{U}$: Universe of items to hash. $n = |\mathcal{U}|$.
- m : Size of hash table.
- h : Hash function. $h : \mathcal{U} \to [m]$.
- $\mathcal{H}$: A family of hash functions.
- T : Hash table.
- $[v] = \{0, 1, ..., v-1\}$, $[v]^+ = \{1, 2, ..., v-1\}$.
- $\alpha := n/m$ is the *load factor* of a hash table.
- $C_{x,y} = [h(x) = h(y)]$.

Important Types of Hash Function Families

- **Uniform**: $\Pr\limits_{h\in\mathcal{H}}[h(x) = i] = \frac{1}{m}, \ \ \forall x \in \mathcal{U}, i \in [m]$.
  - This concerns the distribution of the hashed values being uniform.
- **Universal**: $\Pr\limits_{h\in\mathcal{H}}[h(x) = h(y)] \le \frac{1}{m}, \ \ \forall x \in \mathcal{U}, y \in \mathcal{U}, x \neq y$.
  - This concerns the probability of a collision happening being ideal.
  - I will also use this term to define a single hash function, rather than a family.
- **Near-universal**: $\Pr\limits_{h\in\mathcal{H}}[h(x) = h(y)] \le \frac{2}{m}, \ \ \forall x \in \mathcal{U}, y \in \mathcal{U}, x \neq y$.
  - This concerns the probability of a collision happening being *close* to ideal.
  - The number 2 is not special and can be replaced with another constant.
  - This is a more "realistic" expectation of a family $\mathcal{H}$.
- **k-uniform**: $\Pr\limits_{h\in\mathcal{H}}\left[\bigwedge\limits_{j=1}^{k} h(x_j) = i_j\right] = \frac{1}{m^k}$ for all distinct $x_1, ..., x_k$ and all $i_1, ..., i_k$.
  - It means that for any k disjoint keys and any k hash values, the probability that a specific key hashes to a specific value is $1/m^k$.
  - Ideally, a random hash function should be k-uniform for every positive integer k.

Multiplicative Hashing Families

In here, hash functions will be specified by a *salt*, which is an integer parameter a. When a hash table is created, a salt is chosen uniformly at random. The salt remains fixed during the usage of the hash table.

Prime Multiplicative Hashing

We start by choosing a prime $p > n$. For any integer $a \in [p]^+$, we define a hash function $multp_a : \mathcal{U} \to [m]$ as:

$$multp_a(x) = (ax \mod p) \mod m$$

and the family $\mathcal{MP} := \{multp_a \mid a \in [p]^+\}$. The family $\mathcal{MP}$ of is **near-universal** [2].

Making it Universal

For any integers $a \in [p]^+$ and $b \in [p]$ – so, using two salts instead of one – we can define the hash function:

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

Let $\mathcal{MB}^+ := \{h_{a,b} \mid a \in [p]^+, b \in [p]\}$ be the family of all $p(p-1)$ such functions. $\mathcal{MB}^+$ is **universal** [2].

Binary Multiplicative Hashing

This is a more interesting variation because it's much easier to code. We make the assumption that $\mathcal{U} = [2^w]$ and $m = 2^\ell$, where w and ℓ are integers such that $\ell \leq w$. Also, we restrict the options of salt a to be only **odd** integers[1] within $[2^w]$. Thus, we define the hash function:

$$multb_a(x) = \left\lfloor \frac{a \cdot x \mod 2^w}{2^{w-\ell}} \right\rfloor$$

This is even possible to be implemented in a macro in C++: `#define multb(a,x) ((a) * (x)) >> (W - L))`. Again, we define the family $\mathcal{MB} := \{multb_a \mid a \in [2^w], a \equiv 1 \mod 2\}$. $\mathcal{MB}$ is **near-universal** [2].

Hashing Algorithms (and Collision Resolution)

Chaining

This is the most common method of handling collisions. It transforms each entry $T[i]$ of a hash table into a linked list. Collisions are handled by appending items to their corresponding linked list. If $\ell(x)$ is the length of $T[h(x)]$, then all of lookup, inserting, and deletion of x will take $O(1 + \ell(x))$ time.

If the hash function h is universal, then $E[\ell(x)] = \alpha$.[2] Similarly, if h is near-universal, then $E[\ell(x)] = 2\alpha$. However, the expected *worst-case* operations are not as good. If $\alpha = 1$, the value of $E[\max_x \ell(x)] = \Theta(\log n / \log\log n)$ [2]. There are a few ways to improve worst-case expectations, and some will be discussed here. The following sub-sections describe alternative algorithms for handling this.

Open Addressing

The idea of open addressing is to look at various entries in the table until we either find the key K or find an empty position (concluding that it is not in the table.) It removes links entirely. For each value x, we define a sequence $\langle h_0(x), h_1(x), ..., h_m(x)\rangle$ that is a permutation of $[m]$. The search and insertion[3] implementation would look something like the following:

```
# T -> Hash table
# h -> hash function sequence (length m)
# x -> value we are searching
def OpenAddressSearch(T, h, x):
    for i in range(len(h)):
        if T[h[i](x)] == x:
            return 'Found'
        elif T[h[i](x)] is None:
            return 'Not found'
    return 'Full'
```

```
# T -> Hash table
# h -> hash function sequence (length m)
# x -> value we are inserting
def OpenAddressInsertion(T, h, x):
    for i in range(len(h)):
        if T[h[i](x)] == x:
            return 'Already in table'
        elif T[h[i](x)] is None:
            T[h[i](x)] = x
            return 'Inserted'
    return 'Full'
```

Linear Probing

Linear probing is a special case of open addressing. A single hash function h is chosen, and, for a certain x, we use the sequence $\langle h(x), h(x)-1, ..., 0, m-1, m-2, ..., h(x)+1\rangle$. In this case, we say that this sequence has a *step size* 1. Linear probing has good cache performance [2] and is extremely simple to code. It also works fine until the table begins to get full [1].

Depending on the hashing method used, consecutive keys $\{K, K+1, ...\}$ can have consecutive hashed values. This would considerably slow down linear probing by forming "clusters" of occupied values in the hash table. However, multiplicative hashing can break up these clusters. Another way to protect against consecutive hashes is to instead change the step size from 1 to c in the created sequence (any positive c will work well, as long as it is relatively prime with m) [1].

[1]Note that if m is a power of two, any odd integer a is relatively prime with m.

[2]Mini-proof: $E[\ell(x)] = \sum_{y \in T} E[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m} = \alpha$.

[3]Deletion is considerably more complicated. Deleting an element by just finding where it is stored and removing it may lead to future false negatives when searching for other items. One turnaround is to just simply change the value stored from x to a special code value meaning that x was deleted. I'm personally not a huge fan of this idea because it seems wasteful. Knuth [1] suggested an algorithm for deletion in linear probing that was also a bit slow. I'm yet to read on how to very efficiently do this.

Double Hashing

This is very similar to linear probing. The difference is that it uses two hash functions $h_1(x)$ and $h_2(x)$. The first, $h_1(x)$ will produce values in $[m]$, while $h_2(x)$ produces values in $[m]^+$ that are relatively prime with m.[4] If we are trying to search or insert a specific key k, the algorithm suggested by Knuth in [1] consists of doing a linear probing with initial value $h_1(k)$ and step size $h_2(k)$.

My implementation follows:

```
def DoubleHashingSearch(T, m, h1, h2, k):
    i = h1(k)
    c = h2(k)
    for _ in range(m):
        if T[i] == k:
            return 'Found'
        elif T[i] is None:
            return 'Not Found'
        i = (i - c + m) % m
    return 'Full'
```

```
def DoubleHashingInsert(T, m, h1, h2, k):
    i = h1(k)
    c = h2(k)
    for _ in range(m):
        if T[i] is None:
            T[i] = k
            return 'Inserted'
        i = (i - c + m) % m
    return 'Full'
```

There are several techniques that I am not going to go in detail on how to pick h_1 and h_2. They can be found in [1]. In those techniques, h_1 and h_2 are essentially independent. Hence, the probability that $h_1(x) = h_2(x)$ is approximately proportional to $1/m^2$.

Cuckoo Hashing

Cuckoo hashing uses two independent hash functions h_1 and h_2 from a near-universal family $\mathcal{H}$. An item x can only be stored in one of two positions in the hash table: $h_1(x)$ or $h_2(x)$. Hence, search is guaranteed to be constant time. The idea of the insertion algorithm is to attempt to insert a new item x into $h_1(x)$. If $T[h_1(x)]$ is empty, the item is inserted and the algorithm terminates. Otherwise, we remove the item y stored in $T[h_1(x)]$ and attempt to insert y into the position off its alternate hashing function.[5] The algorithm recurses until it finds an empty position or a cycle. If the latter happens, we choose two new independent hashing functions from $\mathcal{H}$, rehash, and increase the size of the hash table. This last step is suggested in [3].

Implementations inspired by [3] follow. Insertion was done iteratively, rather than recursively. Notice that search and deletion are very simple.

```
def CuckooHashingSearch(T, h1, h2, x):
    if T[h1(x)] == x or T[h2(x)] == x:
        return True
    return False

def CuckooHashingDelete(T, h1, h2, x):
    if T[h1(x)] == x:
        T[h1(x)] = None
    elif T[h2(x)] == x:
        T[h2(x)] = None
    return 'Not present'

def rehash(T, h1, h2):
    pass
```

```
def CuckooHashingInsert(T, n, h1, h2, x):
    if CuckooHashingSearch(T, h1, h2, x):
        return 'Already inserted'

    pos = h1(x)
    for _ in range(n):
        if T[pos] is None:
            T[pos] = x
            return 'Inserted'
        x, T[pos] = T[pos], x  # Swap
        pos = h2(x) if pos == h1(x) else h1(x)

    rehash(T, h1, h2)
    CuckooHashingInsert(T, n, h1, h2, x)
```

Search and deletion are clearly fast and simple. Intuitively, it looks like insertion could be slow if there are too many collisions – or, even worse, if there is a cycle to be detected and rehashing is necessary. However, if $\mathcal{H}$ is near-universal and m is large enough, this turns out to not be a problem. As shown in [3], rehashing is clearly $O(n)$, and, if m is at least $c > 1$ times larger than n, cycles and rehashing will be rare enough that the amortized insertion cost is $O(1/n)$.[6]

[4]A nice thing about this is that, if m is prime, any value in $[m]^+$ can be produced. Alternatively, if m is a power of two, then $h_2(x)$ can produce any odd value.

[5]That is, if y is currently in position $h_1(y)$, we try to insert it in position $h_2(y)$. Otherwise, we try to insert it in $h_1(y)$.

[6]I purposefully made this a little vague to cut down on the math. More details are in [3].

Perfect Hashing

Perfect hashing consists of two levels of hashing [2]. The top level contains a hash table with size $m = n$ and a near-universal hash function. Collisions within the top level are resolved with secondary hash tables. Notice that there are several secondary hash tables (in fact, there are m of them). The k-th[7] secondary hash table should have size $2n_k^2$, where n_k is the number of items whose primary hash value is k. With probability 1/2, there will be no collisions at all in a secondary hash table [2], making the worst-case search time $O(1)$. If a collision happens in a secondary hash table, it should be rebuilt with a new near-universal hash function. Assuming near-universal hashing, we have $E\left[\sum_i n_i^2\right] < 3n$. Thus, in expectation, this data structure needs $O(n)$ amount of space.

Erickson did not explain how to build this in order to support insertion. Since we do not know the values n_k beforehand (and they are subject to change), we are doomed to start with a sub-optimal scenario. One way I thought would be easy to implement this is to initially have every $n_k = 1$. If a collision ever happens in the k-th secondary table, we double n_k, rehash, and rebuilt the new table of size $2n_k^2$. Note that, with this change, the sizes n_k no longer strictly reflect the number of items whose primary hash is k, and it now simply use to set the size of the k-th secondary table.

Other Topics

There are other topics that I read a little about, but not nearly enough yet. Knuth mentioned *Linear Hashing* was incredibly important because it makes the address space grow and shrink dynamically. I was curious about *Bloom Filters* because they are efficient, and also Chazelle's *Bloomier Filters* because they seemed more powerful. Lastly, I glanced over Erickson's section on *Binary Probing.*

If I ever read more on those, I'll write about them.

References

[1] Donald Knuth. *The Art of Computer Programming, Vol 3.* Addison-Wesley, 1968.

[2] Jeff Erickson. *Algorithms (Director's Cut)*,
`http://jeffe.cs.illinois.edu/teaching/algorithms/`

[3] Rasmus Pagh. *Cuckoo Hashing for Undergraduates.* 2006.
`https://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf`

[7] I should mention that this is 0-indexed.