1 Choosing a Hash Function when Using Chaining

In order to choose a good has function, you should have an idea of the distribution of keys if at all possible. There are two general methods to generate hash function that are both simple and for reasonable distributions of keys appearing in the real world will satisfy the simple uniform hashing property well.

Criteria for choosing a good hash function:

- it should distribute keys roughly uniformly into slots,
- regularity in key distribution should not affect the uniformity.

1.0.1 Division Method

Pick a slot size m. The hash function is simply as follows.

$$h(k) = k \mod m.$$

This division method works fairly well if m is picked very carefully.

It is good to pick m so that it does not have any small divisors. Suppose, for instance, that m is even and all the keys happen to be even. Then $h(k)$ will always be even and only half the slots will be used! Another case is, if one picks $m = 2^r$ for some r, then only the r least-significant bits are used to distinguish values. If these bits are regularly-patterned, then there will be many collisions. In practice, a good rule of thumb is to pick m to be a large prime number not too close to a power of 2 or 10. This tends to work well, as one of the more common sources of regularity in the world comes from working in binary or decimal.

1.0.2 Multiplication Method

A more commonly used method is the multiplication method: Pick m to be some power of 2, $m = 2^r$. As we discussed previously, a computer has w bit words. Let A be an odd integer such that $2^{w-1} < A < 2^w$. Now define the hash function as

$$h(k) = ((A \cdot k) \mod 2^w) >> (w - r).$$

This method is extremely efficient, more efficient than the division method since both $\mod 2^w$ and right shifting ($>>$) is often implemented in hardware.

Knuth says to use $A \approx \frac{\sqrt{5}-1}{2} \cdot 2^{32}$ for $w = 32$ bit words. Again, this function ends up working well for most sets of keys from real world distributions.

2 Open Addressing

In the previous lecture, hash collisions were resolved by chaining colliding entries into a linked list. An alternative approach is to store all the entries in the hash table itself (with no nested lists), and to supply a hash function that sequentially tries different positions in the table until it finds a free slot.

Terminology: hash table entries consist of key, value pairs. Keys are drawn from a **universe** U. The hash table is an array T of length m. The hash function is defined as

$$h(k, p) : U \times \{0, \cdots, m-1\} \to \{0, \cdots, m-1\},$$

that is, h accepts a key k and a **probe number** p and outputs a position in the table T.

To insert an element with key k and value v, the algorithm increments p until $h(k, p)$ probes a free spot:

Algorithm 1: Hash-Insert(T, k, v)

```
for p = 0, ··· , m − 1 do
    x ← h(k, p)
    if T[x] = NIL then
        T[x] ← entry(k, v)
        return
error "hash table overflow"
```

Similarly, searches involve running through the probe numbers until a matching element is found:

Algorithm 2: Hash-Search(T, k)

```
for p = 0, ··· , m − 1 do
    x ← h(k, p)
    if T[x] = NIL then
        return NIL
    if T[x].key = k then
        return T[x]
return NIL
```

The sequence $\langle h(x, 0), \cdots, h(x, m-1) \rangle$ is the **probe sequence** of x. Observe that a good selection for h generates a probe sequence that is a permutation of $\langle 0, \cdots, m-1 \rangle$ for each $x \in U$. If this were not the case, then the probe sequence would contain repeats (which means wasted work for Hash-Insert and Hash-Search) and would not contain every integer in the range $[0, m-1]$ (which means Hash-Insert would not take advantage of T's full size). This criteriion motivates two basic implementations of h.

2.1 Linear probing

A simple definition of h is

$$h(k, p) = (h_1(k) + p) \mod m.$$

This simply uses a hash function $h_1(k) : U \to \{0, \cdots, m-1\}$ to pick a starting point, and consecutively probes the elements behind it. In practice, linear probing performs poorly because of **primary clustering**, a phenomenon where long runs of filled slots build up. The loose explanation is that an empty slot preceded by i full slots is filled next with probability $(i+1)/m$, a positive feedback loop which causes long runs to get even longer.

2.2 Double hashing

A better idea for h is to space out the probes:

$$h(k, p) = (h_1(k) + p \cdot h_2(k)) \mod m.$$

This time, instead of probing forward one element at a time, h probes forward $h_2(k)$ elements at a time (looping around to the start of T if necessary). To ensure that h actually generates a probe sequence that's a

permutation of $\langle 0, \cdots, m-1 \rangle$, a necessary and sufficient condition is that $h_2(k)$ is coprime to m (The proof of this is beyond the scope of the class, and you don't need to remember this).

Double hashing is extremely effective in practice. Implementations typically choose m as a power of 2, because this means the modulo operation can be performed efficiently using a bit mask.

2.3 Analysis

Next, we analyze the performance of open addressing. The analysis uses an assumption of **uniform hashing**, which assumes that h has the property that each key's access sequence is equally likely to be any one of the $m!$ possible table index permutations. Neither linear hashing nor double hashing necessarily satisfy this criterion, however, in practice, they often almost satisfy it (for the input distributions in the real-world). There also exist hash functions that provably satisfy this criterion (see CLRS for details on this).

As usual, we define the load factor as $\alpha = \frac{n}{m} \leq 1$ (question: why isn't α ever > 1?). The expected runtime of search and insert is bounded above by the expected number of searches in the probe sequence. This in turn is bounded by the following handy result.

Theorem 2.1. *Let X be the number of consecutive non-nil entries in a probe sequence generated with uniform hashing. Then*

$$E[X] \leq \frac{\alpha}{1-\alpha}.$$

Proof. Let X_i be the number of consecutive non-nil entries after the i^{th} entry of the probe sequence, given that the first i entries are non-nil. A recursive formulation of X_i can be given by

$$E[X_i] = \frac{n-i}{m-i}(1 + E[X_{i+1}]).$$

To see why this is, observe that if the first i entries are non-nil, then there are $m-i$ remaining spots in the probe sequence, of which $n-i$ are non-nil. By the assumption of uniform hashing, there's a probability $\frac{n-i}{m-i}$ that the next entry is non-nil. If the next entry is non-nil, the sequence length is 1 plus X_{i+1}. Otherwise it's zero. Now, using the property $\alpha = \frac{n}{m} \leq 1$,

$$E[X_i] = \frac{n-i}{m-i}(1 + E[X_{i+1}]) \leq \alpha(1 + E[X_{i+1}]).$$

Therefore

$$\begin{aligned} E[X] = E[X_0] &\leq \alpha(1 + E[X_1]) \\ &\leq \alpha(1 + \alpha(1 + E[X_2])) \\ &\leq \alpha(1 + \alpha(1 + \alpha(1 + ... \\ E[X] &\leq \alpha + \alpha^2 + \alpha^3 + \cdots = \frac{\alpha}{1-\alpha}. \end{aligned}$$

□

Any search conducted by Hash-Insert or Hash-Search will consider at most X non-nil elements plus the nil element that indicates when to stop probing. Thus, an upper bound on the expected runtime of either operation is $O(1 + E[X]) = O\left(\frac{1}{1-\alpha}\right)$.