1 Preliminaries

1.1 Indicator Random Variables

An indicator random variable is a special kind of random variable associated with the occurrence of an event. The indicator random variable I_A associated with event A has value 1 if event A occurs and has value 0 otherwise. In other words, I_A maps all outcomes in the set A to 1 and all outcomes outside A to 0.

In our case, event A is some key k mapping to some slot x. So an indicator random variable $I_{h,k,x}$ has the following definition:

$$I_{h,k,x} = \begin{cases} 1 & \text{if } h(k) = x \\ 0 & \text{otherwise} \end{cases}$$

With using above definition, we can define an indicator random variable for collisions in the following way:

$$I_{h,k,x} = \begin{cases} 1 & \text{if } h(k) = h(x) \\ 0 & \text{otherwise} \end{cases}$$

If the hash function is clear from the context and we are referring to a list of keys $k_1, k_2, \ldots, k_n$, we can simplify the notation further to obtain

$$I_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$$

One useful property of indicator random variables is their expected value is the probability of that event happening. More formally

Lemma: $E[I_{i,j}] = Pr\{h(k_i) = h(k_j)\}$

Proof:

$$E[I_{i,j}] = 0 \cdot Pr\{h(k_i) \neq h(k_j)\} + 1 \cdot Pr\{h(k_i) = h(k_j)\}$$

$$= 0 + Pr\{h(k_i) = h(k_j)\}$$

$$= Pr\{h(k_i) = h(k_j)\}$$

1.2 Linearity of Expectation

One useful property of expected value is that it distributes over summation. In other words, expected value of sum of random variables is equal to sum of expected values of that variables. Formally

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

1.3 k-wise Independent Hash Functions

A k-wise independent hash function h is a function that provides the guarantee that, for any k distinct elements $e_1, \ldots, e_k$ and any k possible values $x_1, \ldots, x_k$, the probability that $h(e_i) = x_i$, for all i, is exactly $\frac{1}{m^k}$.

2 Simple Universal Hashing Assumption

Remember that SUHA stated that hashed values are uniformly distributed among buckets. Where do this randomness comes from? There are two possibilities.

1. **Keys are uniformly distributed:** Most of the time this is not a reasonable assumption. Real life data tend to have patterns and connections inside them.

2. **Hash function chosen randomly:** This is better than the above because this way, you don't need to assume anything about the input keys. However, you need to choose your hash function uniformly among *all* possible hash functions. Because the set of all possible hash functions is large, this approach is not practical either.

How can we do better? Here is how!

3 Advanced Hashing Techniques

In the first lecture, constant time insert and search is achieved based on simple universal hashing assumption. However this assumption is pretty strong. Therefore some more advanced hashing schemes are invented to relax this assumption. We will investigate some of them.

3.1 Universal Hashing

The idea behind universal hashing is selecting your hash function such a way that it is guaranteed that collision probability is small. If we express it formally, a set of hash functions $\mathbb{H}$ is universal if for all distinct pair of keys, probability of collision is smaller than or equal to the $\frac{1}{m}$ if the hash function h is randomly chosen from $\mathbb{H}$.

Using this type of function guarantees that expected value of the number of items in a slot is small because probability of k keys colliding in a single slot will be less than or equal to $\frac{1}{m^k}$.

Some universal hash families

- Polynomials with random coefficients
- Dot product hash functions
- Random 0/1 matrices

Exercise: Show that expected number of items in a slot is upper bounded by $1 + \alpha$. *(Hint: Use indicator random variables for collisions)*

What is the advantage of using a universal hash function? The chosen universal set will be much smaller than the set of all hash functions (which is also universal, but a bad one). Therefore it will be easy to generate a hash function randomly.

3.2 Open Addressing

One other approach can be changing how we handle the collisions. Open addressing is a collision handling mechanism that inserts the element to another location if the original slot is occupied by another element.

Where the element should be inserted in case of collision is determined with a probing sequence. It just is a sequence of indexes where a key can be inserted. Key will be inserted to the first empty slot following the index sequence.

Another way to think of it is hash function also taking a natural number with key as an input and returning the index corresponding to the position of that natural number. More precisely, if $p_0, p_1, \ldots, p_m$ is the probing sequence of key k, then $h(k, i) = p_i$.

If you will insert key k, you first "probe" $h(k, 0)$. If it is empty, place it there. If not, "probe" $h(k, 1)$ and continue this until you insert key k.

3.2.1 Linear Probing

One simple technique for probing is just jumping c slots forward when the current slot is full where hash function is the division method. Precisely, hash function is the following

$$h(k, i) = (k + c \cdot i) \mod m$$

Pugh et al. showed that using a 5-wise independent hash function with linear probing gives expected $O(1)$ time operations.

Exercise: Insert $[35, 14, 27, 22, 9]$ into a hash table using linear probing with $m = 13$ and $c = 2$.

Question: In which collision resolution method (chaining vs linear probing) individual search operations take more time if same keys inserted to the same size hash tables with same hash functions? Why?

Question: When would it be better to use linear probing instead of chaining? Why?

3.3 Cuckoo Hashing

Idea behind cuckoo hashing is maintaining two hash tables with different hash functions and "kicking out" a value to the other table if the slot is occupied. This continues until no value is kicked out, entered in an infinite loop, or $O(logn)$ iterations passed.

Advantage of cuckoo hashing is that search only takes two lookups because each key is guaranteed to be in one of their corresponding positions in the two tables.

Disadvantage is that insertion is complicated and potentially time consuming. Because of this, the hash functions should be chosen more carefully.

Exercise: Insert $[35, 14, 27, 22, 9]$ into two hash tables using division method as a hash function with $m_1 = 13$ and $m_2 = 7$.

3.4 Perfect Hashing

Up until now, we were aiming for *expected* $O(1)$ time for operations. What if we have a fixed set of data (meaning that there won't be any insertions after the data hashed) and we want to run many search queries on them? One real life example would be building a catalog for items a factory produces.

Can you use hashing to guarantee *all* search queries will take $O(1)$ time? Perfect hashing is a way to do that. It is essentially a two step hashing scheme where in the first step, you are using universal hashing with table size $m_1 = n$. Lets say that each slot i has c_i collisions after that. then you create another hash table with universal hashing for each slot i with table size c_i^2.

By birthday paradox, you are "guaranteed" to have no collisions on second level. However same "guarantee" can be obtained by universal hashing with table size $m = n^2$. Advantage of this two-level complex scheme is that its space usage is $O(n)$, compared to the simple scheme's $O(n^2)$.