CSc 3102: Hash Tables

Hasning & Common Hashing Functions

- The Dictionary Problem
- Hash Functions
- Collision Resolution

1 The Dictionary Problem

Definition 1. Let D be a totally ordered set of elements called keys. The dictionary problem is one of maintaining a collection of items drawn from D. The basic operations are inserting a key, deleting a key, and searching for a key.

The motivation at the heart of a hash table is to assign keys to data items and then use the keys to search for or remove data items. The underlying data structure is called a *dictionary*. It supports such operations as $put(k, v)$ (insert), $remove(k)$ and $get(k)$ (search/find). A *dictionary* is also called a *map*. A map is a collection of $\langle key, value \rangle$ pairs. Generally, each key maps to a unique value. Some authors allow for a key to map to multiple values. In this course if the keys in a container are not unique, we will refer to the data structure as a *multimap*, not a map. We may think of a map as an associative array that associates each key to a value. Implementing a map is trivial if each key k is an integer in the range $[0, N-1]$ where N is some small number. Without this assumption, we need a special function $h(k)$, called a hash *function*, that maps each key to an integer in $[0, N-1]$.

Other operations on a map include searching for the maximum or minimum keys, accessing the keys in sorted order, etc. One approach to use when implementing a map is a *direct-addressing* scheme in which each element is

mapped to a unique position in an array. Although the three primitive hashing operations (put, remove and get) will all have constant time complexity, O(1), this approach is not feasible: the size of the array may be too large or only very few positions in the array may be utilized. The goal of hashing is to define an appropriate mapping h from the set D to a smaller set D' consisting of consecutive integers. Although this approach saves memory, its drawback is that h may map multiple keys to the same integer in D'. When determining a hash function, we must approximate a random function $h : D \to D'$, from a source (or universe) set U of type D to a destination set of type D'. The source set is usually significantly larger than the destination set so we end up with a many-to-one mapping. We end up with *collsions*, when multiple keys map to the same value. Hence a good hash function must have these two characteristics: it minimizes the number of collisions and it is easily computable.

2 Uniform Hash Function

The number of collisions is minimized if each key k from D is equally likely to be mapped by the hash function h to any slot in the hash table. That is,

$$P\{h(k) = i\} = \frac{1}{m}, \qquad i = 0, 1, ..., m-1$$

Very often the probability distribution is not known and even when it is known, $h(k)$ satisfying the condition above is difficult to compute.

3 Open Addressing or Closed Hashing

The keys in open addressing are stored directly in the hash table. So one key may result in multiple collisions. Collisions can be reduced by using a good hash function that uniformly distributes the keys. It is virtually impossible to eliminate collisions since a perfect hash function is difficult to find. We now discuss three commonly used collision resolution strategies:

Definition 2. Linear probing: Search the table sequentially starting from the original hash position.

$$(h(k) + i)\ mod\ m, \quad i = 0, 1, ..., m-1.$$

This may create clustering. Why? In practice, we want to keep the load factor below 0.7 to reduce clustering and collisions.

Definition 3. Quadratic Probing: This can eliminate primary clustering but causes secondary clustering.

$$(h(k) + i^2)\ mod\ m, \quad i = 0, 1, ..., m-1.$$

For quadratic probing and linear probing a good choice of m is a prime number close to a power of 2.

Definition 4. Double hashing: This drastically reduces clustering, by far more than quadratic probing. Double hashing defines key-dependent probe sequences. In this scheme the probe sequence still searches the table in a linear order, starting at the location $h_1(k)$, but a second hash function $h_2(k) \neq 0$ determines the size of the steps taken.

$$(h_1(k) + h_2(k)i)\ mod\ m, \quad i = 0, 1, ..., m-1.$$

In general, we want to ensure that every cell can be checked when there is a collision; that is we want to reduce the potential for failure on an insertion when there are still many open slots in the hash table. To achieve this we want $h_2(k)$ and m to be relatively prime. Two possible ways to do this is to either make $m = 2^k$ and design $h_2(k)$ so it is always odd, or make m prime and $h_2(k) < m$, where $h_2(k) \neq 0$. In class we will discuss how to compute the probability that there is no collision and the probability that there is at least one collision in an open hash table after n keys are inserted.

4 Closed Addressing or Open Hashing

Definition 5. Separate chaining: in this approach each entry table[i] is a linked list. So we have an array of linked lists in which the keys are stored. Under the assumption of uniform distribution, the average complexity of unsuccessful search is given by:

$$U_n = \alpha = \frac{n}{m}$$

since $\frac{n}{m}$ is the average length of a chain. $\frac{n}{m}$ is the so-called **load factor**. In closed addressing, collision is resolved by adding the key at the end (or beginning) of a linked list that belongs to the slot. Three common questions that arise in close and open hashing are:

1. What is the expected number of items, also referred to as *keys*, that map to the same slot? This is $\frac{n}{m}$, where m is the table size, number of slots, and n is the number of items inserted in the table.

2. What is the expected number of empty slots? The probability that an item is not inserted into a given slot is $1 - \frac{1}{m}$, where m is the number of slots. Since we are assuming that each item is equally likely to map to any slot, the probability that a slot remains empty after n items are inserted is $\left(1 - \frac{1}{m}\right)^n$. So the expected number of empty slots for $n = m$ is $m\left(1 - \frac{1}{m}\right)^n \approx \frac{m}{e}$ as $n \to \infty$. Since $\frac{1}{e} = 0.36787944117$, for $n = m$ we expect about a third of the slots to be empty.

3. What is the expected number of collisions? Suppose ϵ is the number of empty slots after the insertion of n items. Then $m - \epsilon$ is the number of non-empty slots. Therefore, $n - m + \epsilon$ will lead to collisions. So for $n = m$, we expect $\frac{n}{e}$ or approximately a third of the items to cause collisions.

We now discuss two classical problems in statistics that can be applied to closed addressing: the *Birthday Paradox* and the *Coupon Collector's Problem*. What is the likelihood that there is at least one collision? This question is related to the *birthday paradox*: When there are n or more people in a room, what is the chance that at least two people have the same birthday? It turns out that for a table of size 365 you need only 23 keys for a 50% chance of a collision, and as little as 60 keys for a 99% chance. These values can be obtained using elementary probability theory. When hashing to m slots, you can expect (50% chance) a collision after only after approximately $\sqrt{\frac{1}{2}\pi m}$ insertions. What is the expected number of items that when inserted into a table of size m that uses closed addressing fills all slots? The question is related the popular statistical problem called the *coupons collector's problem*. This number, with proof omitted, is mH_m, where H_m is the m^{th} harmonic number: $H_m = \sum_{i=1}^{m} \frac{1}{i}$. Also, $H_m \approx \ln m + \gamma + \frac{1}{2m}$, where $\gamma = 0.5772156649$ is the Euler's constant (not the Euler's number e).