CMSC 351 - Introduction to Algorithms
Spring 2012
Lecture 12

Instructor: MohammadTaghi Hajiaghayi
Scribe: Rajesh Chitnis

1 Introduction

In this lecture we will look at Hashing.

2 Hashing

A hash table is a generalization of the simpler notion of an ordinary array. Many applications require a dynamic set that supports only the **dictionary operations**: Insert, Search and Delete. A hash table is an effective data structure for implementing dictionaries. If an application needs dictionary operations with keys drawn from the universe $\mathbb{U} = \{0, 1, \ldots, u-1\}$ where u is not too large, then we can have direct-address table method by using an array $T[0, 1, \ldots, u-1]$. What about when the universe $\mathbb{U}$ is so large that storing a table T of size $|\mathbb{U}|$ may be impractical and in addition the set K of keys actually stored may be so small relative to $\mathbb{U}$ (e.g. keys in a dictionary). With hashing, we can reduce space to $\theta(|K|)$ with search time only $O(1)$ (though in average). With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, a **hash function** is used to compute the slot from the key k. Here h maps the universe $\mathbb{U}$ of keys into the slots of a hash table $T[0, 1, \ldots, m-1]$, i.e., $h : \mathbb{U} \to \{0, 1, \ldots, m-1\}$. We say that an element with key k hashes to slot $h(k)$ or $h(k)$ is the hash value of key k.

The main issue is that two keys may hash to the same slot. This is called as a collision for which we see effective techniques in this lecture. One idea is to make h appear to be "random" to avoid collisions.

3 Hashing Functions

The average performance of hashing depends on how well it distributes the keys in the m slots, i.e., how "random" it is to avoid collisions. We consider

integers or characters, fixed length arrays (of integers) and variable length arrays (strings):

1. **Hashing integers:** The original proposal of Cater and Wegman was to pick a prime $p \geq u$ and define

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

where a, b are randomly chosen integers mod p and $a \neq 0$.

2. **Hashing fixed-length array:** The input is a vector $\overline{x} = \{x_0, x_1, \ldots, x_{h-1}\}$ of h integers (or bytes/characters) and we define

$$h(\overline{x} = \sum_{i=0}^{h-1})h^i_{a,b}(x_i)$$

where each $h^i_{a,b}$ is constructed by choosing a, b randomly mod p with $a \neq 0$. Here again we choose p to be a prime greater than u.

3. **Hashing strings:** If the length of a string can be bounded by a small number it is best to use a fixed-length array solution. Now we assume that we want to hash $\overline{x} = \{x_0, x_1, \ldots, x_L\}$ where a bound on L is not known a priori. Again if $x_i \in \{0, 1, \ldots, u-1\}$ then let $p \geq \max\{u, m\}$ be a prime and define

$$h_c(\overline{x}) = h_{a,b}\Big((\sum_{i=0}^{L} x_i c^i) \mod p\Big)$$

where $a, b\&c$ are randomly chosen integers mod p with $a, c, \neq 0$.

Definition 1 *A hash function is called* ***universal*** *if* $\forall\ x, y \in \mathbb{U}, x \neq y$ *we have* $\Pr[h(x) = h(y)] \leq \frac{1}{m}$.

Note that this is the ideal case in which any two keys of the universe collide with probability at most $\frac{1}{m}$. All functions above provide such a guarantee. Note that without randomness we cannot guarantee such functions. Universal hashing guarantees any given element is equally likely to hash into any slot.

However if we use any approach then there is a chance of collisions. So what are the solutions to prevent collisions?

1. **Collision Resolution by Chaining**
In chaining we put all elements that hash to the same slot in a linked list. The slot j contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements then slot j contains NIL. The dictionary operations are as follows:

   - Hash-Insert(T, X)
   Insert X at the head of the list $T[h(\text{key}[X])]$

- Hash-Search(T, k)
  Search for an element with key k in the list $T[h(k)]$
- Hash-Delete(T, X)
  Delete X from the list $T[h(k)]$

The worst case running time for insertion is $O(1)$ though for delete and search it is $O(|T[h(X)]|)$. Because of the uniformity property, the average length of each list is $\frac{n}{m}$ where n is the number of keys and m is the size of T. Thus running time for delete and search is in $O(1 + \frac{n}{m})$. Thus if m is at least proportional to n, then $n = O(m)$ and the running time is $O(1)$ for all operations.

2. **Open Addressing: Linear Probing**
   In open addressing, all elements are stored in the hash table itself and it avoids pointers (and waste of memory) altogether. Thus in open addressing, the hash table can fill up so that no further insertion can be made. Instead in the case of collision we need to examine (probe) a sequence of slots.

   In the simplest form of open addressing, we are using the method of linear probing as follows: given a key k, the first slot probed is $T[h(k)]$. We next probe $T[h(k) + 1]$ and so on up to slot $T[m - 1]$. Then we wrap around to slots $T[0], T[1], \ldots$ until we finally probe slot $T[h(k)]$. Note here that the initial probe position determines the entire probe. In insertion we probe according to this order until we find an empty space or give an error of "hash table overflow". For search, we probe accordingly until we find the element or an empty slot.

   Deletion from an open-address hash table is difficult. When we delete a key from slot i, we need to mark it as "deleted" instead of NIL such that search skips over it, but Insert can insert it. Again if load factor $\alpha = \frac{n}{m}$ is constant then Insert and Search can take $O(1)$ in average. However Delete can cause problems and make operations no longer dependent in α. For this reason Chaining is more commonly selected as a collision resolution technique when keys must be deleted often. Sometimes if the number of deletions is a lot we can do re-hashing, i.e., hash in a new array.

References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*