COMPSCI 330: Design and Analysis of Algorithms **Apr 9, 2019**

Lecture 20: Hashing

Lecturer: Rong Ge *Scribe: Haoming Li*

Overview

In this lecture, we study *hash table*, a data structure that uses a *hash function* to map keys to values, providing quick and efficient access to them.

20.1 Motivation and Goal

Hash table is motivated by the set- and map-like data structure: an array whose index can be any object. For example, if we pass in key "hash", we would expect the data structure to return value "a dish of diced or chopped meat and often vegetables".

In this lecture, we want to maintain a set of n numbers from the *universe* $\{0, 1, 2, \ldots, N-1\}$, where N is a very large number (say 2^{64}). If we pass in a key that is in the set, we'd expect the data structure to return value 1, and otherwise 0. We'd like our data structure to have the following nice properties:

- Efficient lookup: checking whether a number is in the set takes $O(1)$.
- Efficient space: the data structure takes $O(n)$ space

20.2 Naive Implementation

There are two ways we can immediately think of to maintain such a set:

Linked List: It is possible to store all the numbers in the set as a linked list. Since there are n numbers, the space required is $O(n)$, which is optimal. However, lookup takes $O(n)$.

Large Array: We can also just allocate a very large array $a[\,]$, and let $a[i] = 1$ if i is in the set. This way, lookup time is $O(1)$. However, this array will take up huge amount of memory (2^{64}).

20.3 Hash Table

Inspired by the two naive methods, we can achieve our goal using a *hash table*, which is defined by:

- Choosing a size m of the hash table. m is small, say a constant times n.
- Choosing a *hash function* $f : \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, m-1\}$.
- Allocating an array of size m. At each location, have a pointer to a linked list.
- Operations (say a lookup call) on number x will be performed on the linked list at location $f(x)$ of the array.

Knowing that our array is by construction space-efficient, the best case scenario of our data structure would be that every number in the set is in its own location (all linked lists have size ≤ 1), thereby achieving $O(1)$ lookup time. The worst case scenario, however, would be that every number in the set is mapped to the same location by the hash function, thereby resulting in $O(n)$ lookup time (essentially a linked list). In fact, as long as we are using some fixed hash function, it is always possible to run in to such *collisions* for some bad examples.

To get around such worst cases, one may suggest that we use a new hash function every time we make a query (e.g. lookup call). However, this does not work: we may store 123 at position 4 because $f(123) = 4$, but after we choose a new hash function, $f'(123)$ may not be equal to 4. The hash table should perform lookup consistently, telling us whether a number is indeed in the set.

20.4 Random Hash Functions

Indeed, we have to choose a random hash function when creating the hash table. After that it is fixed for all the operations. That is, after the function f is chosen, $f(123)$ *always* returns, say, location 4. This makes sure we can access the numbers consistently.

We then quickly run into another problem if we choose our hash function randomly out of all possible hash functions. For each number in the universe there are m choices. Hence there are m^n possible hash function. Hence just storing a totally random hash function would take $\log_2 m^N = N \log_2 m$ bits, and, as we have established earlier, N is a huge number.

20.4.1 Pairwise Independent Hash Family

Our goal is then to construct a hash function that is random enough to avoid collision but does not take that much of space. In practice, we use a function from a *pairwise independent hash family*. A family F of hash functions is *pairwise independent* if for any $x \neq y$, we have

$$Pr_{f \sim F}[f(s) = f(y)] = \frac{1}{m}$$

where m is the size of the hash table. There are pairwise independent hash families of size N^2, hence storing a random function in this family takes $\log_2 N^2 = 2 \log_2 N$ bits.

20.4.2 $O(1)$ Lookup

To show that such hash function indeed achieves $O(1)$ lookup, let's consider the following question: Given a hash table with n elements $\{x_1, x_2, \ldots, x_n\}$ and a query y (want to find out whether $y \in \{x_1, x_2, \ldots, x_n\}$), what is the expected running time of the query? Observe that running time is in fact the length of the linked list at location $f(y)$. Let X be the length of the linked list. We want to compute $E[X]$.

Let X_i be a random variable such that $X_i = 1$ if $f(x_i) = f(y)$ and 0 if otherwise. Since f is in a pairwise independent hash family, we have $E[X_i] = 1$ if $x_i = y$ and $1/m$ if otherwise. Hence $X = X_1 + \cdots + X_n$. Hence $E[X] = E[X_1] + \cdots + E[X_n] \leq 1 + \frac{n-1}{m} \leq O(1 + \frac{n}{m})$. If we choose $m = \Theta(n)$, we would have $E[X] = O(1)$.