Chapter C5: Hash Tables

C5.1 Buckets and Hash Functions

The ***hash table*** is designed to do the unsorted dictionary ADT. It consists of:

1. an array of fixed size (normally prime) of ***buckets***
2. a ***hash function*** that assigns an element to a particular bucket

There will be ***collisions***: multiple elements in the same bucket. There are several choices for the hash function, and several choices for handling collisions.

Ideally, a hash function should appear "random"! A hash function has two steps:

1. convert the object to `int`.
2. convert the `int` to the required range by taking it ***mod*** the table-size

A natural method of obtaining a hash code for a string is to convert each char to an int (e.g. ASCII) and then combine these. While concatenation is possibly the most obvious, a simpler combination is to use the sum of the individual char's integer values. But it is much better to use a function that causes strings differing in a single bit to have wildly different hash codes. For example, ome might compute the sum

$$\sum_i a_i \, 37^i$$

where a_i are the codes for the individual letters.

C5.2 Collision Resolution

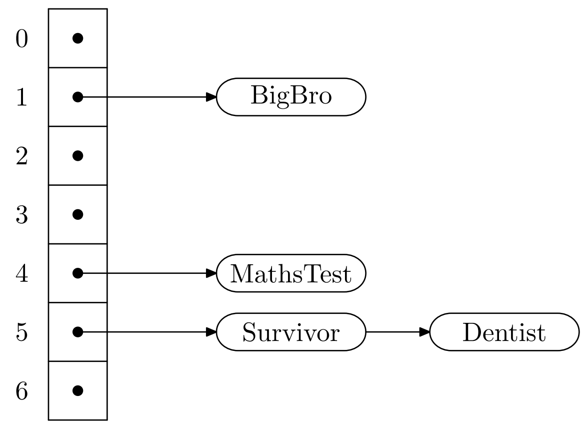
◊ ***Chaining***

The simplest method of dealing with collisions is to put all the items with the same hash-function value into a common bucket implemented as an unsorted linked list: this is called ***chaining***. It can be shown that if one inserts n elements at random into n buckets, then the expected maximum occupancy of a bucket is $O(\log n)$.

The ***load factor*** of a table is the ratio of the number of elements to the table size. Chaining can handle load factor near 1.

Example. Suppose hashcode for a string is the string of 2-digit numbers giving letters (A=01, B=02 etc.) Hash table is size 7. Suppose we store:

BigBro = 020907021815 $\to$ 1
Survivor = 1921182209221518 $\to$ 5
MathsTest = 130120081920051920 $\to$ 4
Dentist = 04051420091920 $\to$ 5



◊ ***Open Addressing***

An alternative to chaining is called ***open addressing***. In this collision-resolution method: if the intended bucket h is occupied, then try another one nearby. And if that is occupied, try another one.

There are two simple strategies for searching for a nearby vacant bucket:

- ***linear probing***: move down the array until find vacant bucket (and wrap around if needed): look at $h, h+1, h+2, h+3, \dots$
- ***quadratic probing***: move down the array in increasing increments: $h, h+1, h+4, h+9, h+16, \dots$ (again, wrap around if needed)

Linear probing causes ***chunking*** in the table, and open addressing likes load factor below 0.5.

The operations of search and delete become a bit more complex. For example, how do we determine if string is already in table? And deletion must be done by ***lazy deletion***: when the entry in a bucket is deleted, the bucket must be marked as "previously used" rather than "empty". Why?

C5.3 Rehashing

If the table becomes too full, the obvious idea is to replace the array with one double the size. However, we cannot just copy the contents over, because the hash value is different. Rather, we have to go through the array and re-insert each entry.

One can show (amortized analysis again) that this does not significantly affect the average running time. The point is that one can spread the cost of the dynamic resizing. In particular, if the hash table was previously rehashed at size d and we now rehash at size $2d$, then there have been d insert's since the previous rehash. Thus the $O(2d)$ cost of the rehash is constant per insert.

C5.4 Other Applications of Hashing

Hash functions can be used for quick testing for duplicates: if the hash function is "random" then two different items will almost surely hash to a different value. This idea can be used to test in expected linear time whether a collection of integers from some bounded interval has duplicates.

$\boxed{\text{Add Me}}$

Exercises

1. $\boxed{\text{Add Me}}$