Hash Tables

18.1 Dictionary

The dictionary ADT supports:

- insertItem(e): Insert new item e
- lookup(e): Look up item based on key; return access/boolean

Applications include counting how many times each word appears in a book, or the symbol table of a compiler. There are several implementations: for example, red-black trees do both operations in $O(\log n)$ time. But we can do better...

18.2 Components

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

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

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

18.3 Hash Functions

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

- convert the object to int.
- 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, compute the expression

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

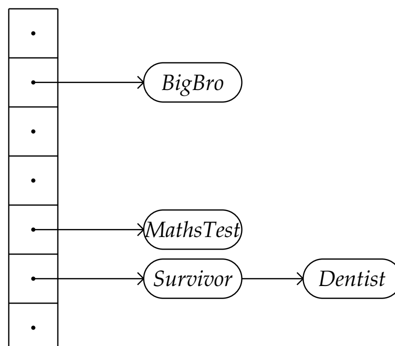
where a_i are the codes for the individual letters.

18.4 Collision-Resolution

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***.

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 store:
> BigBro = 020907021815 $\rightarrow$ 1
> Survivor = 1921182209221518 $\rightarrow$ 5
> MathsTest = 130120081920051920 $\rightarrow$ 4
> Dentist = 04051420091920 $\rightarrow$ 5



An alternative approach to chaining is called ***open addressing***. In this collision-resolution method: if intended bucket h is occupied, then try another 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 array until find vacant (and wrap-around if needed): look at $h, h+1, h+2, h+3, \ldots$
- ***quadratic probing***: move down array in increasing increments: $h, h+1, h+4, h+9, h+16, \ldots$

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

Operations of search and delete become more complex. For example, how do we determine if string is already in table? And deletion must be done by ***lazy deletion***: why?

18.5 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.