Lecture 15: Introduction to hashing (11.1-11.2)

CS303: Algorithms

Last update: February 11, 2014

1 Review

- Non-comparison based sorting (count and radix)
- Only applicable to integers with a range specified.

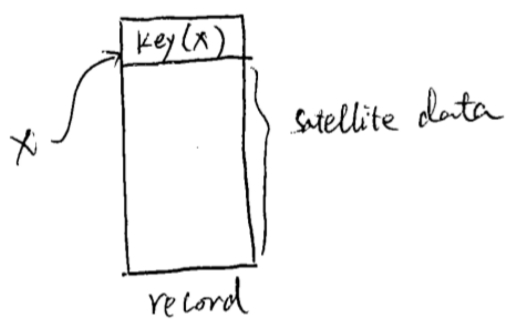
2 Introduction to hashing

Hashing is a very popular and useful technique that is used pretty everywhere where dynamic data are used. Let's use the following example

Q:/ How does google perform fast search?

A:/ Imagine we type in a keyword or a few key words into the search box. What's gonna happen is our keyword will be used as a key to be found in the huge databases maintained by google. The figure below shows the structure of a single entry in S.

table S holding n records



The satellite data are the webpages that google gives out based on the keyword.
Operations we want to do on the table

Insert $(S, x) : S \leftarrow S + \{x\}$
Delete $(S, x) : S \leftarrow S - \{x\}$

Search (S, k) :return x such that if $key[x] = k$ or return nil if no such x.

Note we delete the record using the record, not the key. Thus delete involves searching first.

Q:/ How can we solve this problem?

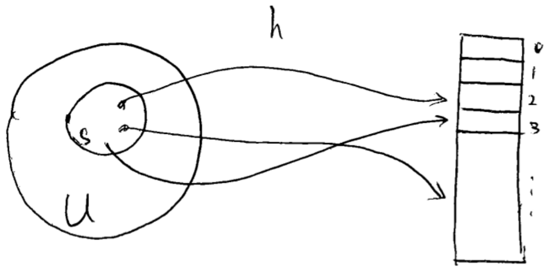
A:/ Idea 1: use direct access table (ie. Array)

It only works if keys are drawn from $U = 0, 1, 2, ..., m - 1$ and all keys are distinct.
Set up array $T[0, 1, ..., m - 1]$ to present the dynamic set where
$T[k] = \begin{cases} x & \text{if } x \in S, key[x] = k \\ nil & \text{otherwise} \end{cases}$
Thus all operations take $\Theta(1)$ time.
Q:/ What's the limitation?
A:/ m can be very big

e.g. if I have a few thousand records but I will draw my key as any integer. The reason of any integer is because the number of records may grow and shrink. So my array has to have billions of entries but only hold the current thousands of records.
So this is where hashing come in place

Idea: keep the table small but still "keeps" the constant time properties of the array idea.

use a hash function h that maps keys "randomly" into slots of table T. T is called the hash table.

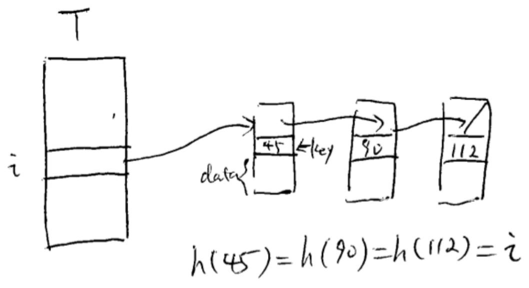


S is usually a small set of U and h basically maps the values in S to indexes in the table T.
Q:/ What kind of problem that may occur?

A:/ two keys may be mapped to the same slot in the table. $\rightarrow$ collision

When a record to be inserted maps to an already occupied slot, a collision occurs
Q:/ How to solve this problem?

A:/ make the entries in table T head of lists $\rightarrow$ open hashing.



2.1 Efficiency analysis

Worst case: it happens when every key in S were hashed to the same slot in T. This is basically a linked list. Search takes $\Theta(n)$ if $|S| = n$
Average case: Assuming that our hash function is ideal

Q:/ What should an ideal hash function do?
A:/ hash keys randomly and uniformly to slots. $\rightarrow$ called a simple uniform hashing

Load factor of hashing: $\alpha = n/m$ which tells us on average, how long the list is for open hashing. Then the average time for unsuccessful search or a successful search is both $\Theta(1 + \alpha)$. 1 is from computing the hashing value and access the slot and α is the average length of the list we have to go through.
Q:/ Why isn't it just $\Theta(1)$?
A:/ It is because α isn't a constant. We are talking about dynamic data here.

Q:/ When do we have $\Theta(1)$ time?
A:/ When α is a constant. ie. n isn't much larger than m. Thus the size of the hash table has to increase if you have more keys.

This analysis relies heavily that the hash function needs to be simple random hash function.

Generally, a hash function should:

- be easy to compute
- distribute keys about evenly throughout the hash table (ie. Minimize collisions)
- regularities in keys shouldn't affect uniformity (eg. If all keys are even numbers, the hash function shouldn't be affected).
- there is serious research going on for designing good hash functions. (google sparseHash)

2.2 Division method

$h(k) = k \ mod \ m$ where m is the number of entries in the table.
You don't want to pick m that is a power of 2.
For example, if $m = 2^p$, then $k \ mod \ m$ is just the last p bits of k. Unless we are sure the last k bits of p is random, we don't want to do that.
Usually we pick m as a prime number that isn't close to power of 2.