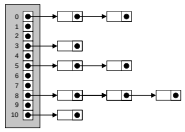


CSIS 3103

Hashing

Hash Tables

Goal: Access an entry based on its key
(no searching to determine its location)

Using a hash table enables us to retrieve an item in
– constant time (on average)
– linear (worst case)

A Lookup Problem

A phone company wants to provide caller ID capability
– given a phone number, return the caller's name

Phone numbers look like:

999-999-9999

Could be represented as integers in the range $0..R$ where $R = 10^{10} - 1$
… lots of these are not actually used

Solutions we've seen so far

- Linear search: $O(N)$
- Binary search: $O(\log N)$

I have a data structure where the key is found on the first attempt ***every time***!

The Ultimate Solution

An array indexed by the phone number

(null)	(null)	...	Olan	...	(null)
000-000-0000	000-000-0001	...	609-652-4587	...	999-999-9999

– The search time is optimal, $O(1)$
– But the space required is ***huge***, $O(R)$

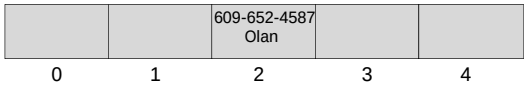
Another Solution

- We need to reduce the number of cells in the array
- Many of the 10-digit numbers cannot be used as phone numbers anyway
- A *hash table* is an alternative which has $O(1)$ *expected* search time (*not* worst-case) but with a reduced space requirement

Example

- Suppose we pick a table of size $N = 5$
- We can map the phone number (key) to a slot in the table by calculating

`index` = `key` *mod* 5



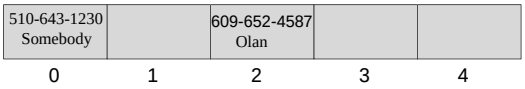
6096524587 *mod* 5 = 2

Example

Lookup uses the same process:

Map the key to an index and check the array cell at that index

- Next, insert (609-404-1230, Somebody)
- Then insert (609-643-8362, Whozit)
  Oops, a *collision occurs* at index = 2

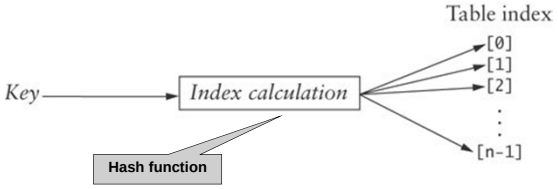


Hashing

- Recall binary search's success is based on divide-and-conquer
  – divide the data set size in half at every step
- Hashing divides the set of keys into m roughly equal sized subsets (*buckets*)
- Search time is reduced to a value proportional to n/m on average, where n is the total number of data items
- What we need is a way to quickly pick the subsets where the keys will be put

Hash Codes and Index Calculation

The basis of hashing is to transform an item's key into an integer value which will then be transformed into a table index



Hash Codes and Index Calculation

h(*key*) → index in array

- A "good" hash function minimizes the probability of collisions
- The Java `Object` class defines a `hashCode()` method that returns an `int` for any object Object.hashCode()
- To use a hashcode as an index, it must be in the range $0 \le h < m$ (table size)

```
int h = key.hashCode() % table.length;
```

Generating Hash Codes

- The number of possible key values is much larger than the table size
- Generating good hash codes is somewhat of an experimental process
- The hash function should generate a uniform random distribution of its values, and be relatively efficient to compute

Java `HashCode` Method

- For strings, summing the `int` values of all characters doesn't work well
  - Consider the hash codes for *sign* and *sing*
- `String.hashCode()` uses the formula:

  $s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$

  where s_i is the *i*th character of the string, and *n* is the length of the string

  "Cat" has a hash code of:

  'C' x 31^2 + 'a' x 31 + 't' = 67510

Java `HashCode` Method

The `String.hashCode` method distributes the hash code values fairly evenly

- The probability of two strings having the same hash code is low

The probability of a collision with

```
s.hashCode() % table.length
```

is proportional to how full the table is