COMP90038 Algorithms and Complexity

Hashing

Lars Kulik

Lecture 17

Semester 1, 2016

Hashing

Hashing is a standard way of implementing the abstract data type "dictionary".

Implemented well, it makes data retrieval very fast.

A key can be anything, as long as we can map it efficiently to a positive integer. In particular, the set K of keys need not be bounded.

Assume we have a table of size m (the hash table).

The idea is to have a function $h : K \rightarrow \{1..m\}$ (the hash function) determine where records are stored: A record with key k should be stored in location $h(k)$.

The address $h(k)$ is the hash address.

The Hash Table

We can think of the hash table as an abstract data structure supporting operations:

- find
- insert
- lookup (search and insert if not there)
- initialize
- delete
- rehash

The challenges:

- Design of hash functions
- Collision handling

The Hash Function

If we have a hash table of size m and keys are integers, we may define $h(n) = n \bmod m$.

But keys may be other things, such as strings of characters, and the hash function should apply to these and still be easy (cheap) to compute.

We need to choose m so that it is large enough to allow efficient operations, without taking up excessive memory.

The hash function should distribute keys evenly along the cells of the hash table.

Hashing of Strings

For simplicity assume A $\mapsto 0$, B $\mapsto 1$, etc.
Also assume we have, say, 26 characters, and $m = 101$.

Then we can think of a string as a long binary string:

M Y K E Y $\mapsto$ 0110011000010100010011000 $(= 13379736)$

$$13379736 \bmod 101 = 64$$

So 64 is the position of string M Y K E Y in the hash table.

We deliberately chose m to be prime.

$$13379736 = 12 \times 32^4 + 24 \times 32^3 + 10 \times 32^2 + 4 \times 32 + 24$$

With $m = 32$, the hash value of any key is the last character's value!

Handling Long Strings as Keys

More precisely, let *chr* be the function that gives a character's number (between 0 and 25 under our simple assumptions), so for example, $chr(\mathrm{c}) = 2$.

Then we have $hash(s) = \sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}$.

For example,

$$hash(\mathrm{V\ E\ R\ Y\ L\ O\ N\ G\ K\ E\ Y}) = (21 \times 32^{10} + 4 \times 32^{9} + \cdots)\ mod\ 101$$

The stuff between parentheses quickly becomes an impossibly large number!

Horner's Rule

Fortunately there is a trick that allows us to avoid large numbers in the hash calculations. Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} \cdots$$

factor out repeatedly:

$$(\cdots ((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24$$

Now utilize these properties of modular arithmetic:

$$\begin{array}{rcl} (x + y) \bmod m & = & ((x \bmod m) + (y \bmod m)) \bmod m \\ (x \times y) \bmod m & = & ((x \bmod m) \times (y \bmod m)) \bmod m \end{array}$$

So for each sub-expression it suffices to take values modulo m.

Hashing of Strings

Horner’s rule is easily implemented. For example, here it is in C:

```
unsigned hash(char *v) {
  int h;
  for (h = 0; *v != ’\0’; v++)
    h = (h<<5 + *v) % m;
  return h;
}
```

Here v is a pointer traversing the string.
C uses a special symbol, \0, to mark the end of a string.

The h<<5 means “shift the bits in h 5 places to the left and fill with zeroes”—a quick way of calculating 32*h.

The ‘%’ is the modulus operation in C.

The Hash Function and Collisions

The hash function should be as random as possible.

Here we assume $m = 23$ and $h(k) = k \text{ mod } m$.

In some cases different keys will be mapped to the same hash table address.

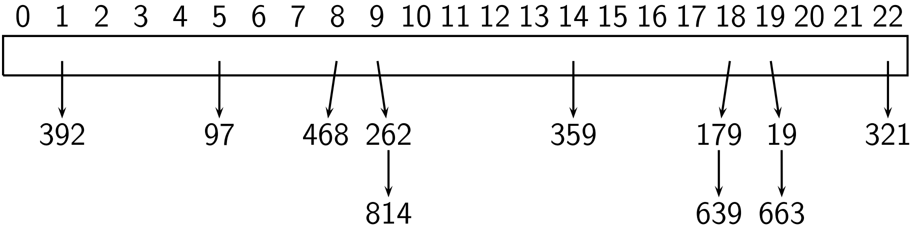
When this happens we have a collision.

Different hashing methods resolve collisions differently.

Key	Address
19	19
392	1
179	18
359	14
663	19
262	9
639	18
321	22
97	5
468	8
814	9

Separate Chaining

Element *k* of the hash table is a list of keys with the hash value *k*.



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

392 97 468 262 359 179 19 321

814 639 663

This gives easy collision handling.

The load factor $\alpha = n/m$, where *n* is the number of items stored.

Number of probes in successful search $\approx 1 + \alpha/2$.

Number of probes in unsuccessful search $\approx \alpha$.

Separate Chaining Pros and Cons

Compared with sequential search, reduces the number of comparisons by a factor of m.

Good in a dynamic environment, when (number of) keys are hard to predict.

The chains can be ordered, or records may be “pulled up front” when accessed.

Deletion is easy.

However, separate chaining uses extra storage for links.

Open-Addressing Methods

With open-addressing methods (also called closed hashing) all records are stored in the hash table itself (not in linked lists hanging off the table).

There are many methods of this type. We only discuss two:

- linear probing
- double hashing

For these methods, the load factor $\alpha \leq 1$.

Linear Probing

In case of collision, try the next cell, then the next, and so on.

After the arrival of 19, 392 (1), 179 (18), 663 (19), 639 (18), 321 (22):

0	1	2	3	16	17	18	19	20	21	22
	392					179	19	663	639	321

Search proceeds in a similar fashion.

If we get to the end of the hash table, we wrap around.

For example, if key 20 now arrives, it will be placed in cell 0.

Linear Probing

Again let m be the table size,
and n be the number of records stored.

As before, $\alpha = n/m$ is the load factor.

Average number of probes:

- Successful search: $\frac{1}{2} + \frac{1}{2(1-\alpha)}$
- Unsuccessful: $\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$

For successful search:

α	#probes
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Linear Probing Pros and Cons

Space-efficient.

Worst-case performance miserable; must be careful not to let the load factor grow beyond 0.9.

Comparative behaviour, $m = 11113$, $n = 10000$, $\alpha = 0.9$:

- Linear probing: 5.5 probes on average (success)
- Binary search: 12.3 probes on average (success)
- Linear search: 5000 probes on average (success)

Clustering is a major problem: The collision handling strategy leads to clusters of contiguous cells being occupied.

Deletion is almost impossible.

Double Hashing

To alleviate the clustering problem in linear probing, there are better ways of resolving collisions.

One is double hashing which uses a second hash function s to determine an offset to be used in probing for a free cell.

For example, we may choose $s(k) = 1 + k \text{ mod } 97$.

By this we mean, if $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

This is another reason why it is good to have m being a prime number. That way, using $h(k)$ as the offset, we will eventually find a free cell if there is one.

Rehashing

The standard approach to avoiding performance deterioration in hashing is to keep track of the load factor and to rehash when it reaches, say, 0.9.

Rehashing means allocating a larger hash table (typically about twice the current size), revisiting each item, calculating its hash address in the new table, and inserting it.

This “stop-the-world” operation will introduce long delays at unpredictable times, but it will happen relatively infrequently.

Rabin-Karp String Search

The Rabin-Karp string search algorithm is based on string hashing.

To search for a string p (of length m) in a larger string s, we can calculate $hash(p)$ and then check every substring $s_i \cdots s_{i+m-1}$ to see if it has the same hash value. Of course, if it has, the strings may still be different; so we need to compare them in the usual way.

If $p = s_i \cdots s_{i+m-1}$ then the hash values are the same; otherwise the values are almost certainly going to be different.

Since false positives will be so rare, the $O(m)$ time it takes to actually compare the strings can be ignored.

Rabin-Karp String Search

Repeatedly hashing strings of length m seems like a bad idea. However, the hash values can be calculated incrementally. The hash value of the length-m substring of s that starts at position j is:

$$hash(s, j) = \sum_{i=0}^{m-1} chr(s_{j+i}) \times a^{m-i-1}$$

where a is the alphabet size. From that we can get the next hash value, for the substring that starts at position $j + 1$, quite cheaply:

$$hash(s, j + 1) = (hash(s, j) - a^{m-1} chr(s_j)) \times a + chr(s_{j+m})$$

modulo m. Effectively we just subtract the contribution of s_j and add the contribution of s_{j+m}, for the cost of two multiplications, one addition and one subtraction.

Why Not Always Use Hashing?

Some drawbacks:

- If an application calls for traversal of all items in sorted order, a hash table is no good.
- Also, unless we use separate chaining, deletion is virtually impossible.
- It may be hard to predict the volume of data, and rehashing is an expensive “stop-the-world” operation.

When to Use Hashing?

All sorts of information retrieval applications involving thousands to millions of keys.

Typical example: Symbol tables used by compilers. The compiler hashes all (variable, function, etc.) names and stores information related to each—no deletion in this case.

When hashing is applicable, it is usually superior; a well-tuned hash table will outperform its competitors.

Unless you let the load factor get too high, or you botch up the hash function. It is a good idea to print statistics to check that the function really does spread keys uniformly across the hash table.

Next Up

Dynamic programming and optimization.

An outline of the remaining weeks' topics is now on the LMS's schedule.

A reminder that Assignment 2 is due on 20 May 2016 at 8:59 am.