Hashing (11)

Michael L. Littman

October 15th, 1998

1 SPELL CHECKING

1.1 Dynamic Dictionary

The built-in English word list on my computer has 45k words in it (probably 15k roots, plus various sufficies and such).

The `ispell` program preprocesses the word list and then can quickly take new words and determine if they are on the list. Words not on the list are flagged as misspellings.

Supposedly misspelled words (proper names, technical terms) can be added to the word list so they won't be flagged later.

1.2 Dictionary

A "dictionary" is a dynamic data structure that supports:

- create n
- insert
- delete
- find

Different from sorting: Order doesn't matter.

1.3 Implementing a Dictionary

We've already seen several data structures that can support these operations (create n, insert, delete, find).

- unsorted list
- sorted list
- binary search tree

- balanced binary search tree
- splay tree (worst case per op)
- splay tree (amortized)

	create n	insert	delete	find
unsorted list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
sorted list	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(n)\Theta(\log n)$	
binary search tree	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
balanced binary search tree	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
splay tree (worst case per op)	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
splay tree (amortized)	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
hash	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

1.4 Similar Applications

Many other standard programs use dictionary-type data structures.

- Caching of previously encountered and optimized regular expressions (regexperl).
- Mapping from hostnames to IP addresses.
- Keeping track of variable names and reserved words (compilers, gperf).
- Mapping from symbols to internal objects (Lisp).
- Mapping from command names to pathnames of executables (shell).
- Associative arrays (perl, awk).

2 HASHING BASICS

2.1 Motivation

How could we do these things in constant time?
Consider a simpler example.
In Rhode Island, license plates consist of two letters (A-Z) followed by 3 digits. How many distinct plates are there? 676,000.
How could we store the valid plate numbers and be able to quickly check if a given number is on the list?
No problem to allocate an array of that size (this is called a *direct-address table*).
How do we insert, delete, and find plate numbers?
Running time?

2.2 Universal Concepts

Here's a vocabulary:

- The *universe* of keys U is the set of possible key values we might want to include in our data structure (around 700k unique plate numbers).
- The *number of slots* m in the table T (same as $|U|$ in this case).
- The *objects* in the universe $K \subseteq U$ are the things we actually insert into the table (the actual license-plate numbers in use).
- The *number of objects* is $n = |K|$ (probably around 500k actual cars on the road, judging from a 1990 population of 1M people and an estimate of 0.49 vehicles per person).
- The *load factor* $\alpha = n/m$ is the amount full the table is (0.73 in this case).

2.3 Extending Tables

If $|U|$ fits on a computer (as it does for Rhode Island license plates, or Duke student ID numbers), we're golden.
But what if n is small enough to fit, but $|U|$ is large? For example, let's consider Duke student names:

- U is the set of all possible names, so $|U| = 10^{22}$ (this is a conservative estimate based on 16-letter names).
- n is around 10,000 total students.
- We can store a table of around $m = 500,000$ with no difficulty.
- This would make $\alpha = n/m = 0.02 < 1$.

2.4 Hash Function

When we have $m < |U|$, we need some way of shoe-horning the universe into the table.
A hash function $h : U \to [0, m-1]$ does this for us.
The *hash value* of an object $k \in U$ is $h(k)$.
A *perfect* hash function has the property that all the inserted objects K have a unique hash value.
For a perfect hash function h:

- Insert(k): $T[h(k)] \leftarrow k$
- Delete(k): $T[h(k)] \leftarrow$ empty
- Find(k): $h(k)$
- Member(k): $T[\text{Find}(k)] = 1$?

Acts just like the direct-address table: everything $O(1)$ worst case.

2.5 Collisions

While there are cases in which perfect hash functions can be constructed, this is unusual. Usually, we will have two distinct objects k_1 and k_2 such that $h(k_1) = h(k_2)$ (their hash values are the same).
We call this a *collision*, and all the action in hashing is in dealing with them.

3 COLLISION RESOLUTION

3.1 Chaining: Basic Idea

Keep a linked list in each "slot" of the table.
If two keys hash to the same slot, stick them both on the list for that slot.

- Insert(k): $T[h(k)] \leftarrow T[h(k)] : k$
- Delete(k): $T[h(k)] \leftarrow \text{remove}(k, T[h(k)])$
- Find(k): $\text{assoc}(k, T[h(k)])$.

Note: `ispell` and `perl` use this scheme.

3.2 Chaining: Evaluation

Some good things:

- Ok to have $n > m$. (What does this say about α?)
- Very simple concept.

Some bad things:

- Requires memory management.
- Once chains get long, performance suffers.

3.3 Open Addressing

Basic idea is to fold the chains list back into the table.
Advantage: no messing around with pointers.
Disadvantage: table can fill up (uh oh, need a new table).
Also: What do we do when there's no place for k?

3.4 Probing

When we get a collision in open addressing, we need to find another slot in which to store the new key.
Probing is the idea of picking the next place to look.
Extend hash function... $h'(k, i)$ is the slot to look in for key k on the ith probe. Ought to be a permutation for all k (over the possible values of $i = 0, \ldots, m-1$).

- Initalize $T[i] \leftarrow$ empty.
- Insert(k): Let $j = \min_i(T[h'(k,i)] = \text{empty})$. $T[h'(k,j)] \leftarrow k$.
- Find(k): Let $j = \min_i(T[h'(k,i)] = \text{empty or } k)$. $h'(k,j)$.
- Member(k): $T[\text{Find}(k)] = k$?

Why does insert actually work? How about find?

3.5 Problem with Delete

What's wrong with this?

- Delete(k): $T[\text{Find}(k)] \leftarrow$ empty.

Alternative:

- Delete(k): $T[\text{Find}(k)] \leftarrow$ deleted.
- Insert(k): Let $j = \min_i(T[h'(k,i)] = \text{empty or deleted})$. $T[h'(k,j)] \leftarrow k$.

Why is Find still ok?

3.6 Linear Probing

- $h'(k,i) = (h(k) + i) \bmod m$.
- That is... take an ordinary hash function, than start scanning forward (with wrap around) for an empty spot.
- Simple!
- Prone to clumping (primary clustering).

3.7 Desirable Property of Hash Functions

We will see from the analysis that an ideal hash function is essentially random... it takes a set of keys and sprays them out uniformly along the length of the table.
So, the hash function should *look* random but be deterministically computable. In crytography, this is called a *one-way function* (easy to compute $h(k)$ from k, but no real pattern linking k back to $h(k)$).
Like anti-sorting!

4 ANALYSIS

4.1 Worst Case

Let's say we insert n distinct keys into an initially empty hash table of size m and then we want to do a find. What's the running time for the find?
Load factor is $\alpha = n/m$.
Worst case for chaining?
Worst case for open addressing?

4.2 Uniform-Hashing Assumption

So, clearly hashing with a bad hash function is not terribly useful.
So, let's assume we have a good one.

- For chaining: $\Pr(h(k) = j) = 1/m$ (the probability that a key ends up in slot j is $1/m$ for all slots).
- For open addressing: $\Pr(h'(k, i) = j) = 1/m$ (the probability that a key ends up in slot j on the ith probe is $1/m$ for all slots).

4.3 Comments on the Assumption

Very odd assumption (even somewhat ill-defined).
Stronger (less general, worse) than the assumption in randomized quicksort. There, we showed that the randomization in the algorithm made it so no particular input elicits worst-case behavior.
Weaker (more general, better) than the assumption needed to show non-randomized quicksort efficient. There, we'd need to assume that the probability of a bad input (reversed list) was small.
Here, we are putting a restriction on the *combination* of our hash function and our inputs. The better our hash function is, the less we care about the key sequence.

4.4 Chaining Analysis

Expected number of steps to discover an item k is not in the hash table:

- We need to look at each item in the list $T[h(k)]$.
- Under our uniform-hashing assumption, the $h(k)$ slot looks like all the others.
- What's the expected number of keys in a slot?
- By *linearity of expectation*, the expected number of keys in a slot j is equal to $\sum_{i=1}^{n} \Pr(\text{key } i \text{ goes in slo}$
$\sum_{i=1}^{n} 1/m = n/m = \alpha$.

So, running time is proportional to the load factor: $\Theta(1 + \alpha)$.

4.5 Open-Addressing Analysis

To discover that k is not in the table, we need to keep probing until we find an empty slot. Here's a simple (and familiar) upper bound.

$$\begin{aligned} P &= 1 + \alpha P + (1-\alpha) \cdot 0 \\ (1-\alpha)P &= 1 \\ P &= 1/(1-\alpha) \end{aligned}$$

As an upper bound, we assume that each probe hits a completely random slot. This is an upper bound because the probability that we hit an empty slot goes up as we use up the filled slots.

4.6 Punchline

As long as we keep the table only a constant-fraction full, we are ok. For $\alpha = 9/10$, $T = 10$. But, if the table is more full than that: $\alpha = (m-1)/m$, $T = m$!

4.7 Universal Hashing

Even a clever hash function is vulnerable to bad expected-case performance.
Any fixed hash function has some set of keys that will result in a pile up.
We can use randomness to get around this, mostly.
Resulting algorithm is the only truly provably good hashing scheme (uniform hashing assumption is a crock), but I don't think anyone uses it in practice.
Like using random pivots in quicksort, the idea is that we'll pick a random hash function from a family of hash functions.
If the family has a particular "universal" property (not difficult to have), we can prove a true expected running time for worst-case inputs.
The critical property is that any pair of keys collides in just $1/m$ of the possible hash functions.
Not always useful in practice, because often we use our hash table only once (and the randomization, in this case, only occurs once per use).
Still, gives us a warm, fuzzy feeling.

5 DYNAMIC OPERATIONS

5.1 Data Structures

Now we've got another data structure to add to our collection (hashing joins heaps, lists, binary search trees, splay trees, etc.)
Each has different running time properties.
We can also combine them in various ways.

5.2 Operations

Here is a collection of dynamic operations:

- insert: put item into data structure
- find min: return pointer to item with minimum key
- find max: return pointer to item with maximum key
- delete: delete item given pointer
- find: return pointer to item with given key
- successor: return pointer to next largest item given pointer to item
- first: return pointer to the item inserted earliest
- next: given a pointer to an item, return a pointer to the next most recent inserted item.

5.3 Combining Data Structures

How would you:

- insert, delete, find max, find min: all amortized $O(1)$, except insert $O(\log n)$.
- insert, delete, first, next: all $O(1)$.
- insert, delete, find, first, next: all expected $O(1)$, except first, next $O(1)$ worst case.
- insert, delete, find, find min: all $O(\log n)$, except find is $O(1)$ expected.