Analysis of Hashing (12)

Michael L. Littman

October 21st, 1997

1 REVIEW

1.1 Announcements

Returning exams today.
Next time: stable marriage or string matching?

1.2 Hashing Idea

Direct-address table gives us fast insertions, deletions, and finds (store item with key k in entry $T[k]$).
Hashing emulates direct-address table when the table is much smaller than the size of the universe of keys. Key k is stored in entry $T[h(k)]$.
Need to cope with *collisions*: chaining (linked list), open addressing (extended hash function and probing).
Today: Analyze expected running time.

2 ANALYSIS

2.1 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?

2.2 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).

2.3 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.

2.4 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 $\sum_{i=1}^{n} \Pr(\text{key } i \text{ goes in slot } j) = \sum_{i=1}^{n} 1/m = n/m = \alpha$.

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

2.5 Open-Addressing Analysis

To discover that k is not in the table, we need to keep probing until we find an empty slot.
In math: (ML: THIS ISN'T QUITE RIGHT...)

$$
\begin{aligned}
&\text{expected running time} \\
&\quad = \text{expected probes until hit empty spot} \\
&\quad = \sum_{l=1}^{n} l \Pr(\text{miss } l-1 \text{ times}) \Pr(\text{hit on } l\text{th time}) \\
&\quad = \sum_{l=1}^{n} l\, n/m(n-1)/m(n-2)/m \ldots (n-l)/m(m-n)/m \\
&\quad \leq \sum_{l=1}^{n} l(n/m)^{l-1}(m-n)/m
\end{aligned}
$$

$$
\begin{aligned}
&= (m-n)/m\sum_{l=1}^{n} l\alpha^{l-1} \\
&= (m/n)(m-n)/m\sum_{l=0}^{n} l\alpha^{l} \\
&\leq (m/n)(m-n)/m\alpha/(1-\alpha)^2 \\
&= (m/n)(1-\alpha)\alpha/(1-\alpha)^2 \\
&= 1/(1-\alpha)
\end{aligned}
$$

2.6 In Words

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.

$$\Pr(\text{collide}) = \frac{\text{filled slots}}{\text{slots}} = \frac{n}{m} = \alpha.$$

$T = 1 + \alpha T$ (look familiar?)
$T = 1/(1-\alpha)$.

2.7 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: $\alpha = (m-1)/m$, $T = m$!

3 UNIVERSAL HASHING

3.1 Motivation

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 is a crock), but I don't think anyone uses it in practice.

3.2 A Universal Hash Function

Let's assume 20 bit keys, so $|U| = 2^{20}$, about a million.
Further, assume $m = 97$ (a prime), and chaining for collision resolution.
Break the key x into 5 blocks of 4 bits each (x_i is the number represented by the bits in the ith block of x).
Before beginning, pick (randomly uniformly) a multiplier for each block $0 \leq a_i < m$.

$$h(x) = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5) \bmod m.$$

3.3 Some Facts

Hash value always in the correct range. Why?
Consider $x = 1$, for all slots j, $\Pr(h(x) = j) = 1/m$. Why?
What about $x = 2$? For what value of a_5 does $h(x) = 1$?
Basically, any given $x \neq 0$ is equally likely to map to any slot.

3.4 Universal Hashing Summary

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.

4 DYNAMIC OPERATIONS

4.1 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.

4.2 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.

4.3 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.

5 EXAM 1

5.1 Comments

Lots of people forgot to circle the problems being done for credit. Annoying. Didn't take it out on you this time...

6 HOMEWORK

6.1 Homework 6

Background: Ch. 18.1, 18.2, 12.
Due October 23rd:

- 6.1: Last time, you created a binary search tree from the letters of the word "algorithm." Do it again, and annotate the tree with the number of credits the splay analysis would assign to each node. Now splay on "m." What is the depth of the resulting tree? Where are the credits now?
- 6.2: Use the amortized analysis template to argue that we can satisfy a sequence of n operations to a heap (initially empty) in $O(\log n)$ amortized time for inserts and $O(1)$ amortized time for delete-max.
- 6.3: Give worst-case bounds for executing min, max, min, max, min, max, ... k times on a balanced binary search tree and a splay tree. Which is asymptotically more efficient if $k = \Theta(\sqrt{n})$?
- 6.4: CLR Exercise 12.2-2 (pg 226).
- 6.5: CLR Exercise 12.2-6 (pg 226).
- 6.6: CLR Exercise 12.3-1 (pg 231).

- 6.7: When a hash table fills up, it is often necessary to expand it (especially with open addressing). Let's say that we start off with a table of size $m = 101$ and every time we reach $\alpha \geq 1/2$, we set $m \leftarrow 2m + 1$ and then reinsert all the items into the new table. Argue that the *amortized* time to insert a single item into the hash table over a sequence of n insertions is $O(\log n)$. Use the uniform-hashing assumption if you need it.

6.2 Homework 7 (Incomplete)

Background: Ch. 27.3.
Due October 30th:

- 7.1: Use a hash table to compute the mode of a list of n numbers in expected $O(n)$ time.