Hash Tables

Adnan Aziz

Based on CLRS, Ch 11.

1 Hashing

Many many applications—need dynamic set supporting insert, search, and deletes.

- symbol table in compilers, database severs, etc.

Hash tables —in worst case take $\Theta(n)$ time to perform these operations; in practice very fast.

- Assumption—accessing $A[i]$ takes $O(1)$ time

First attempt—*direct address tables*
If keys are integers in $U = \{0, 1, \ldots, m-1\}$, m small, and no two elements have the same key.

- just store elements with key i in slot$A[i]$ (initialize to NIL).
  - search, insert, delete—trivial

Very fast!
Catch—$|U|$ may be very large relative to the number of elements we will ever store.
Hash tables—working with set K, try to reduce storage requirements to $\Theta(K)$

- will keep $O(1)$ search, insert, delete times
  - here's the kicker: these are only in the average case

Hashing—first try:

- instead of inserting element with key k in slot k, use a *hash function* to store in array of size m, where $m \ll |U|$

Here h is **some** function mapping U to $\{0, 1, \ldots, m-1\}$.[1]
Will say element with key k *hashes* to slot $h(k)$; $h(k)$ is the *hash value* of k.

- Ex: U is integers, $h(x)$ is $x \bmod m$.

Problem: collisions—two elements may map to the same slot.
How to deal with this?

- make sure it doesn't happen (make h very "random")
  - unfortunately, since $m \ll |U|$, there is always the possibility of a collision

Chaining—one approach to dealing with collisions

- put all elements that hash to the same slot in a linked list
  - for simplicity, assume doubly linked, with pointers to head and tail
- `insert(T,x)`—put at end of `T[h(key[x])]`
  - will **always** perform a search before an insert, make sure never repeat entries
  - inserting at the end is confusing, why not the beginning? Ans: takes same time to insert at head or tail and makes analysis much simpler. (cf. CLRS problem 11.2-3)
- `search(T,x)`—search for element with key k in list `T[h(k)]`
- `delete(T,x)`—delete `x` from list `T[h(k)]`

Time complexity? Insertion is $O(1)$ plus time for search; deletion is $O(1)$ (assume pointer is given).
Complexity of search is difficult to analyze.
Model—T hash table, with m slots and n elements.

- define *load factor* $\alpha = n/m$

[1] Be careful—in this chapter, arrays are numbered starting at 0! (Contrast with chapter on heaps)

In worst case—search looks at $\Theta(n)$ elements.
How to analyze the average case behavior?

- depends critically on how h distributes the keys

Assumption on h —any element is equally likely to be hashed into any one of the m slots, regardless of where the other elements are hashed to. (This is called *simple uniform hashing*.)

- also assume that $h(k)$ takes $O(1)$ time to compute.

From assumptions, $\Rightarrow$ time taken to search for element with key k is proportional to length of `T[h(k)]`.
Now lets analyze the average time to search for key k.

- will consider two cases—search unsuccessful and search successful

Theorem 1 In a hash table in which collisions are resolved by chaining, an unsuccessful search takes $\Theta(1 + \alpha)$ time on average, assuming simple uniform hashing.

Proof: Any key k is equally likely to be in any of the m slots $\Rightarrow$ average time to search = average length of list = n/m = α.
Hence average time is $\Theta(1 + \alpha)$. (We added 1 for computing h.) ■

Theorem 2 In a hash table in which collisions are resolved by chaining, a successful search takes $\Theta(1 + \alpha)$ time on average, assuming simple uniform hashing.

Proof: Assume that the search is equally likely to be any of the n keys, and that inserts are done at the end of the list.
Expected # of elements examined = 1 + # elements examined when sought after element was inserted.
Take average over the n elements of 1 + expected length of list to which the i-th element was added.
The expected length of list to which i-th element is added is $(i - 1)/m$

$$
\begin{aligned}
(1/n) \cdot \Big(\sum_{i=1}^{n} (1 + (i-1)/m)\Big) &= 1 + \frac{1}{m \cdot n} \cdot \Big(\sum_{i=1}^{n} (i-1)\Big) \\
&= 1 + \frac{1}{m \cdot n} \cdot \big(\frac{n \cdot (n-1)}{2}\big) \\
&= 1 + \frac{\alpha}{2} - \frac{1}{2 \cdot m} \\
&= \Theta\Big(1 + \frac{\alpha}{2} - \frac{1}{2 \cdot m}\Big)
\end{aligned}
$$

Hence overall complexity is $\Theta(1 + \frac{\alpha}{2} - \frac{1}{2 \cdot m}) = \Theta(1 + \alpha)$. ■
Think about the case where $\alpha = 1$, when $\alpha \ll 1$, and when $\alpha \gg 1$.

2 Hash functions

What makes for a good hash function?

- comes close to "simple uniform hashing"—each key is equally likely to fall into any slot

Suppose entries are selected from universe with probability p, i.e., $p(k)$ is the probability of choosing key k. Ideally, we would like for each j

$$
\sum_{k:h(k)=j} p(k) = 1/m
$$

Problem: don't usually know p in advance
Idea—use heuristics

- From now on will assume keys are natural numbers
  - string—think of as sequence of bits $\Rightarrow$ an integer (possibly very large)

2.1 Division method

$h(k) = k \bmod m$

- need to avoid certain values of m, e.g., powers of 2, powers of 10

Fact—good values are primes which are not too close to powers of 2

- approx. 2000 strings, willing to examine 3 elements per unsuccessful search
  - use 701 slots

Experiment on real data.

2.2 Multiplication method

pick some $A \in (0, 1)$, hold constant

1. compute $k \cdot A$
2. extract fractional part of $k \cdot A$
3. multiply by m and take floor $\lfloor m \cdot (k \cdot A \bmod 1) \rfloor$

What value for A?

- Knuth suggests that $A = (\sqrt{5} - 1)/2$ which is $0.6180339887...$

2.3 Universal hashing

Pick a hash function from a family of hash functions.

- theoretical nice properties; see CLRS page 232 for details

2.4 Open addressing

Basic idea—store elements in hash table itself

- entry is either an element or NIL

First attempt

- insert k at $h(k)$; if entry at $h(k)$ is not NIL (i.e., something has already been stored there), go to next slot (and next, and next, etc., till you find an open slot.)

- lookup k at $h(k)$; if entry at $h(k)$ is NIL, not present; if key of entry at $h(k)$ is k, element is present, otherwise, examine next slot (as before)

Terrible performance!

- cookie monster performance

Generalize to "probing" $h : U \times \{0, 1, 2, \ldots, m-1\} \mapsto \{0, 1, 2, \ldots, m-1\}$

- examine entries in "probe sequence" $\Big\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \Big\rangle$
  - require $\Big\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \Big\rangle$ is a permutation of $\{0, 1, 2, \ldots, m-1\}$

```
Hash-Insert( T, k)
  i <- 0
  repeat j <- h(k,i)
    if T[j] = NIL
      then T[j] <- k
        return j
      else i <- i + 1
  until i=m
  error overflow
```

Tiny changes for `Hash-Lookup`

2.5 Linear probing

$h(k, i) = (h'(k) + i) \bmod m$, where h' is ordinary hash function
$\Rightarrow$ exactly what we described in our first attempt!

2.6 Quadratic probing

$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$, where c_1, c_2 are nonzero constants

- fairly complex requirements on c_1, c_2, m to get permutations
- if $h(k_1, 0) = h(k_2, 0)$ then $h(k_1, i) = h(k_2, i)$ for all i

2.7 Double hashing

$h(k, i) = \big(h_1(k) + i \cdot h_2(k)\big) \bmod m$

- need $h_2(k)$ to be relatively prime to m to generate permutations
  1. use m a power of 2, and h_2 to always be odd
  2. use m a prime, and h_2 to always be less than m
     - following works very well—$h_1(k) = k \bmod m$, and $h_2(k) = 1 + (k \bmod m')$, where m' is slightly smaller than m

Facts:

- For an open address hash table with load factor $\alpha = n/m < 1$, the average number of probes in an unsuccessful search is at most $1/(1 - \alpha)$
  - follows that on average, insertion requires $1/(1 - \alpha)$ probes
- For an open address hash table with load factor $\alpha = n/m < 1$, the average number of probes in a successful search is at most $1/\alpha + (1/\alpha) \cdot \ln\left(1/(1 - \alpha)\right)$

Real code for a hash function for strings:

```
st_strhash(string, modulus)
register char *string;
int modulus;
{
    register int val = 0;
    register int c;
```

```
    while ((c = *string++) != '\0') {
        val = val*997 + c;
    }

    return ((val < 0) ? -val : val)%modulus;
}
```