CSE 5311 Notes 5: Hashing

(Last updated 1/12/21 11:23 AM)

CLRS, Chapter 11

Review: 11.2: Chaining - related to perfect hashing method

11.3: Hash functions, skim universal hashing
(aside: https://dl-acm-org.ezproxy.uta.edu/citation.cfm?doid=3116227.3068772)

11.4: Open addressing

COLLISION HANDLING BY OPEN ADDRESSING

Saves space when records are small and chaining would waste a large fraction of space for links.

Collisions are handled by using a *probe sequence* for each key – a permutation of the table's subscripts.

Hash function is $h(\text{key}, i)$ where i is the number of reprobe attempts tried.

Two special key values (or flags) are used: *never-used* (-1) and *recycled* (-2). Searches stop on *never-used*, but continue on *recycled*.

Linear Probing - $h(\text{key}, i) = (\text{key} + i) \% m$

Properties:

1. Probe sequences eventually hit all slots.
2. Probe sequences wrap back to beginning of table.
3. Exhibits lots of *primary clustering* (the end of a probe sequence coincides with another probe sequence):

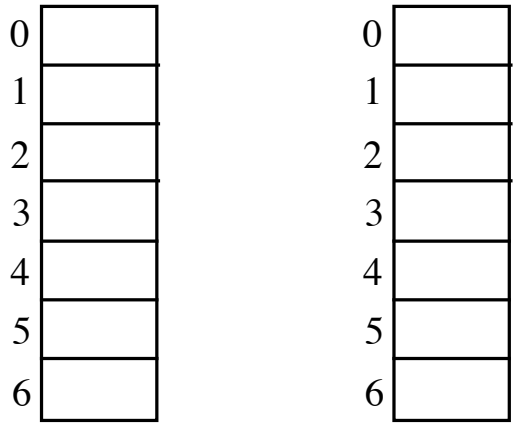
   $i_0\ i_1\ i_2\ i_3\ i_4\ \ldots\ i_j\ i_{j+1}\ \ldots$
   $i_j\ i_{j+1}\ i_{j+2}\ \ldots$

4. There are only m probe sequences.
5. Exhibits lots of *secondary clustering*: if two keys have the same initial probe, then their probe sequences are the same.

What about using $h(\text{key}, i) = (\text{key} + 2*i) \% 101$ or $h(\text{key}, i) = (\text{key} + 50*i) \% 1000$?

Suppose all keys are *equally likely* to be accessed. Is there a best order for inserting keys?

Insert keys: 101, 171, 102, 103, 104, 105, 106



Double Hashing – $h(\text{key}, i) = (h_1(\text{key}) + i*h_2(\text{key})) \% m$

Properties:

1. Probe sequences will hit all slots only if m is prime.
2. $m*(m-1)$ probe sequences.
3. Eliminates most clustering.

Hash Functions:

$h_1 = \text{key} \% m$

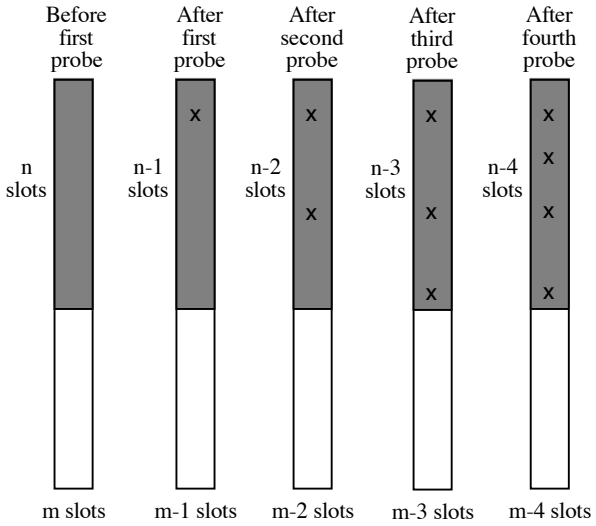
$h_2 = 1 + \text{key} \% (m-1)$

Load Factor $= \alpha = \dfrac{\#\text{ elements stored}}{\#\text{ slots in table}}$

UPPER BOUNDS ON EXPECTED PERFORMANCE FOR OPEN ADDRESSING

Double hashing comes very close to these results, but analysis assumes that hash function provides all $m!$ permutations of subscripts.

1. Unsuccessful search with load factor of $\alpha = \frac{n}{m}$. Each successive probe has the effect of decreasing table size and number of slots in use by one.



   a. Probability that all searches have a first probe 1
   b. Probability that search goes on to a second probe $\alpha = \frac{n}{m}$
   c. Probability that search goes on to a third probe $\alpha \frac{n-1}{m-1} < \alpha \frac{n}{m} < \alpha^2$
   d. Probability that search goes on to a fourth probe $\alpha \frac{n-1}{m-1} \frac{n-2}{m-2} < \alpha^2 \frac{n-2}{m-2} < \alpha^3$

   . . .

   Suppose the table is large. Sum the probabilities for probes to get upper bound on expected number of probes:

   $$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$ (much worse than chaining)

2. Inserting a key with load factor □

   a. Exactly like unsuccessful search

   b. $\frac{1}{1-\alpha}$ probes

3. Successful search

   a. Searching for a key takes as many probes as inserting *that particular key*.

   b. Each inserted key increases the load factor, so the inserted key number $i + 1$ is expected to take no more than

   $$\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i} \text{ probes}$$

   c. Find expected probes for n consecutively inserted keys (each key is equally likely to be requested):

   $$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \qquad \text{Sum is } \frac{1}{m}+\frac{1}{m-1}+\ldots+\frac{1}{m-n+1}$$

   $$= \frac{m}{n}\sum_{i=m-n+1}^{m}\frac{1}{i}$$

   $$\le \frac{m}{n}\int_{m-n}^{m}\frac{1}{x}dx \qquad \text{Upper bound on sum for decreasing function. CLRS, p. 1154 (A.12)}$$

   $$= \frac{m}{n}\left(\ln m - \ln(m-n)\right) = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} = -\frac{1}{\alpha}\ln(1-\alpha)$$

BRENT'S REHASH - On-the-fly reorganization of a double hash table (not in book)

`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=361952.361964`

During insertion, moves no more than *one other key* to avoid expected penalty on recently inserted keys.

Diagram shows how i+1, the *increase in the total number of probes* to search for each key once, is minimized.

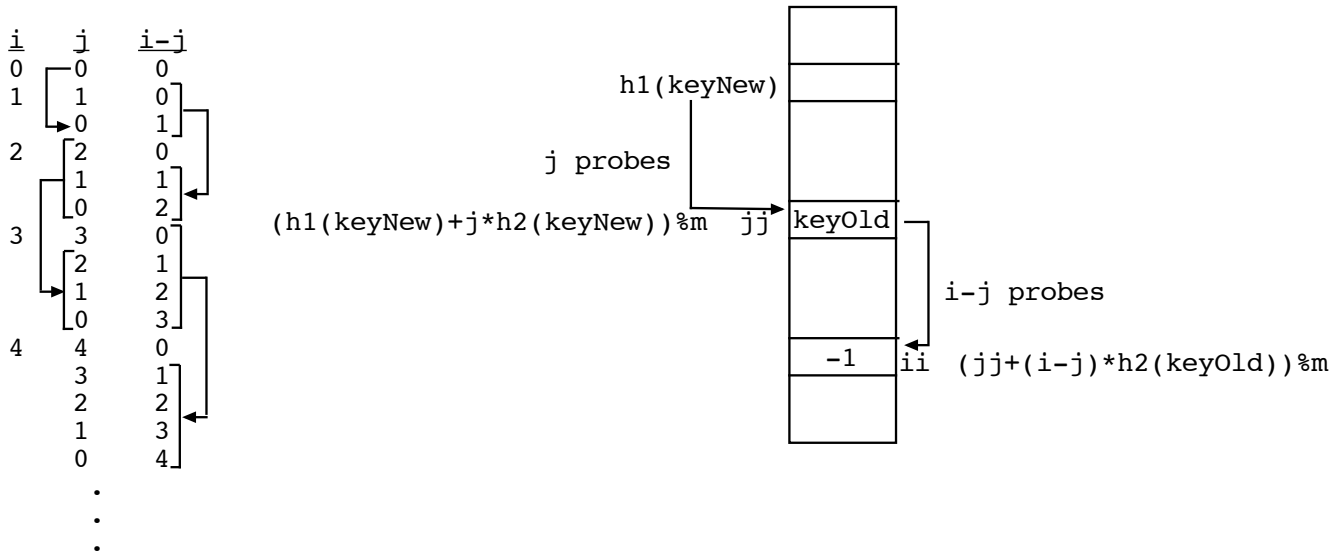
Expected probes for successful search ≤ 2.5. (Assuming uniform access probabilities.)

keyNew uses (j+1)-prefix of its probe sequence to reach slot with keyOld

keyOld is moved i-j *additional* positions down its probe sequence (a key may be moved by several insertions)

Insertion is more expensive, but typically needs only three searches per key to break even.

Unsuccessful search performance is the same as conventional double hashing.



(Conceptual) Test Problem: The hash table below was created using double hashing with Brent's rehash. The initial slot ($h_1(key)$) and rehashing increment ($h_2(key)$) are given for each *key*. Show the result from inserting 9000 using Brent's rehash when $h_1(9000) = 5$ and $h_2(9000) = 4$. (10 points)

	key	$h_1(key)$	$h_2(key)$		Probe Sequences (stopping at empty slot) (part of finding solution)
					1 2 3 4 5 (probes for `keyNew`)
0					2 5 1 1 0 (probes for `keyOld`)
					3 7 4 5 5 (total additional probes)
1				9000	5 2 6 3 0
2	5000	2	1		2 3 4 5 6 0
3	4000	3	4		3 0
4	3000	4	1		4 5 6 0
5	2000	5	5		5 3 1
6	1000	6	2		6 1

An implementation for significant *n* will apply the concept incrementally by increasing i and expanding the set of old keys under consideration.

```
void insert (int keyNew, int r[])
{
int i, ii, inc, init, j, jj, keyOld;

init = hashfunction(keyNew);  // h1(keyNew)
inc = increment(keyNew);      // h2(keyNew)
for (i=0;i<=TABSIZE;i++)
{
  printf("trying to add just %d to total of probe lengths\n",i+1);
  for (j=i;j>=0;j--)
  {
    jj = (init + inc * j)%TABSIZE; // jj is the subscript for probe j+1 for keyNew
    keyOld = r[jj];
    ii = (jj+increment(keyOld)*(i-j))%TABSIZE; // Next reprobe position for keyOld
    printf("i=%d j=%d jj=%d ii=%d\n",i,j,jj,ii);
    if (r[ii] == (-1))  // keyOld may be moved
    {
      r[ii] = keyOld;
      r[jj] = keyNew;
      n++;
      return;
    }
  }
}
}
```

GONNET'S BINARY TREE HASHING - Generalizes Brent's Rehash (aside)

http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=800105.803401

Has same goal as Brent's Rehash (minimize increase in the total number of probes to search for all keys), but allows moving an *arbitrary* number of other keys.

"Binary Tree" refers to the search strategy for minimizing the increase in probes:

a. Root of tree corresponds to placing (new) key at its h_1 slot (assuming slot is empty).
b. Left child corresponds to placing key of parent using same strategy as parent, but then moving the key from the taken slot using one additional h_2 offset. (The old key is associated with this left child.)
c. Right child corresponds to placing key of parent using one additional h_2 offset (assuming slot is empty). (The key of parent is also associated with this right child.)

Search can either be implemented using a queue for generated tree nodes or by a recursive "iterative deepening".

Expected probes for successful search ≤ 2.14. (Assuming uniform access probabilities.)

CUCKOO HASHING (not in book, http://www.itu.dk/people/pagh/papers/cuckoo-undergrad.pdf)

A more recent open addressing scheme with very simple reorganization.

Presumes that a low load factor is maintained by using dynamic tables (CLRS 17.4).

Based on having two hash functions, but no reprobing is used:

```
procedure insert(x)
  if T[h1(x)] = x or T[h2(x)] = x then return;
  pos ← h1(x);
  loop n times {
    if T[pos] = NULL then { T[pos] ← x; return};
    x ↔ T[pos];
    if pos= h1(x) then pos← h2(x) else pos← h1(x);}
  rehash(); insert(x)
end
```

PERFECT HASHING (CLRS 11.5)

Static key set

Obtain O(1) hashing ("no collisions") using:

1. Preprocessing (constructing hash functions)

   and/or

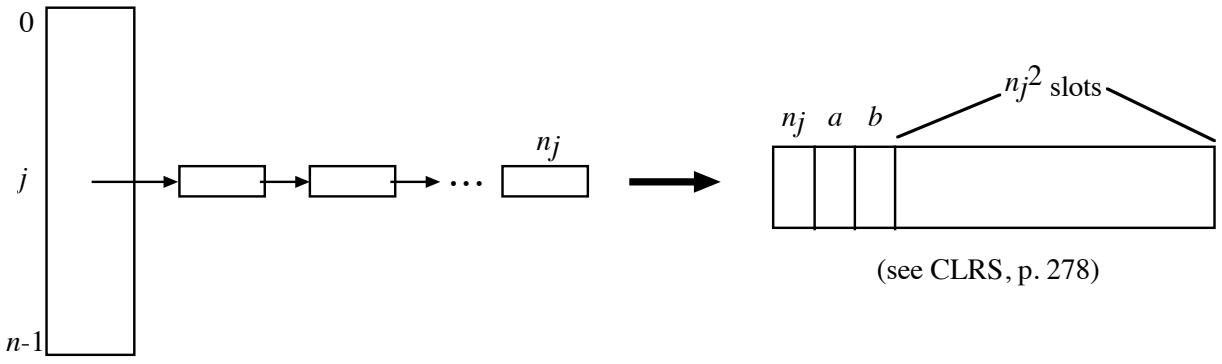
2. Extra space (makes success more likely) - want *cn*, where *c* is small

Many informal approaches - typical application is table of reserved words in a compiler.

CLRS approach:

1. Suppose *n* keys and $m = n^2$ slots. Randomly assigning keys to slots gives prob. < 0.5 of any collisions.

2. Use two-level structure (in one array):



(see CLRS, p. 278)

$\sum E\left[n_j^2\right] < 2n$, but there are three other values when $n_j > 1$ and one other value otherwise.

Brent's method - about 19.8 million keys

```
0.910 l.f. expected=1.837 CPU 3.709
0.920 l.f. expected=1.867 CPU 3.810
0.930 l.f. expected=1.901 CPU 3.920
0.940 l.f. expected=1.938 CPU 4.045
0.950 l.f. expected=1.980 CPU 4.187
0.960 l.f. expected=2.028 CPU 4.354
0.970 l.f. expected=2.085 CPU 4.560
0.980 l.f. expected=2.155 CPU 4.838
0.990 l.f. expected=2.252 CPU 5.290
Retrievals took 2.410 secs
Worst case probes is 47
Total probes 44319520
Expected probes is 2.238
Probe counts:
Number of keys using 1 probes is 9597267
Number of keys using 2 probes is 4793331
Number of keys using 3 probes is 2315618
Number of keys using 4 probes is 1201658
Number of keys using 5 probes is 682835
Number of keys using 6 probes is 403494
Number of keys using 7 probes is 257422
Number of keys using 8 probes is 166305
Number of keys using 9 probes is 112059
Number of keys using 10 probes is 76750
Number of keys using 11 probes is 53787
Number of keys using 12 probes is 37887
Number of keys using 13 probes is 27777
Number of keys using 14 probes is 19969
Number of keys using 15 probes is 14443
Number of keys using 16 probes is 10590
Number of keys using 17 probes is 7893
Number of keys using 18 probes is 5808
Number of keys using 19 probes is 4082
Number of keys using 20 probes is 3046
Number of keys using 21 probes is 2147
Number of keys using 22 probes is 1560
Number of keys using 23 probes is 1223
Number of keys using 24 probes is 846
Number of keys using 25 probes is 628
Number of keys using 26 probes is 465
Number of keys using 27 probes is 320
Number of keys using 28 probes is 222
Number of keys using 29 probes is 175
Number of keys using >=30 probes is 395
```

Gonnet's method - about 19.8 million keys

```
maxkeysmoved is now 1
0.010 l.f. expected=1.005 CPU 0.024
0.130 l.f. expected=1.066 CPU 0.341
maxkeysmoved is now 2
0.420 l.f. expected=1.231 CPU 1.248
maxkeysmoved is now 3
0.580 l.f. expected=1.348 CPU 1.880
maxkeysmoved is now 4
0.770 l.f. expected=1.542 CPU 2.823
maxkeysmoved is now 5
0.880 l.f. expected=1.715 CPU 3.786
maxkeysmoved is now 6
0.940 l.f. expected=1.860 CPU 4.850
maxkeysmoved is now 7
0.950 l.f. expected=1.891 CPU 5.235
0.960 l.f. expected=1.925 CPU 5.694
0.970 l.f. expected=1.965 CPU 6.326
maxkeysmoved is now 9
0.980 l.f. expected=2.010 CPU 7.397
0.990 l.f. expected=2.067 CPU 9.891
0.990 l.f. expected=2.067 CPU 9.891
Retrievals took 2.426 secs
Worst case probes is 20
Total probes 40709494
Expected probes is 2.056
Probe counts:
Number of keys using 1 probes is 9039472
Number of keys using 2 probes is 5332877
Number of keys using 3 probes is 2825972
Number of keys using 4 probes is 1399169
Number of keys using 5 probes is 671102
Number of keys using 6 probes is 301893
Number of keys using 7 probes is 135027
Number of keys using 8 probes is 56484
Number of keys using 9 probes is 23258
Number of keys using 10 probes is 9199
Number of keys using 11 probes is 3450
Number of keys using 12 probes is 1291
Number of keys using 13 probes is 502
Number of keys using 14 probes is 192
Number of keys using 15 probes is 71
Number of keys using 16 probes is 29
Number of keys using 17 probes is 8
Number of keys using 18 probes is 2
Number of keys using 19 probes is 2
Number of keys using 20 probes is 2
Number of keys using 21 probes is 0
Number of keys using 22 probes is 0
Number of keys using 23 probes is 0
Number of keys using 24 probes is 0
Number of keys using 25 probes is 0
Number of keys using 26 probes is 0
Number of keys using 27 probes is 0
Number of keys using 28 probes is 0
Number of keys using 29 probes is 0
Number of keys using >=30 probes is 0
```

CLRS Perfect Hashing - 19.8 million keys

```
malloc'ed 158400000 temporary bytes
malloc'ed 79200000 permanent bytes
realloc for 250328152 more permanent bytes
final structure will use 16.643 bytes per key
Subarray statistics:
Number with 0 keys is 7284696
Number with 1 keys is 7283515
Number with 2 keys is 3641233
Number with 3 keys is 1214078
Number with 4 keys is 304196
Number with 5 keys is 60575
Number with 6 keys is 10072
Number with 7 keys is 1411
Number with 8 keys is 201
Number with 9 keys is 21
Number with 10 keys is 2
Number with 11 keys is 0
Number with 12 keys is 0
Number with 13 keys is 0
Number with >=14 keys is 0
Time to build perfect hash structure 4.912
Time to retrieve each key once 4.478
```

BLOOM FILTERS (not in CLRS)

http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=362686.362692

http://en.wikipedia.org/wiki/Bloom_filter

M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge Univ. Press, 2005.

m bit array used as filter to avoid accessing slow external data structure for misses

k independent hash functions for the *m* bit array

Static set of *n* elements to be represented in filter, but not explicitly stored:

```
for (i=0; i<m; i++)
  bloom[i]=0;
for (i=0; i<n; i++)
  for (j=0; j<k; j++)
    bloom[(*hash[j])(element[i])]=1;
```

Testing if candidate element is possibly in the set of *n*:

```
for (j=0; j<k; j++)
{
  if (!bloom[(*hash[j])(candidate)])
    <Can't be in the set>
}
<Possibly in set>
```

The relationship among *m*, *n*, and *k* determines the *false positive* probability *p*.

Given *m* and *n*, the *optimal* number of functions is $k = \frac{m}{n}\ln 2$ to minimize $p = \left(\frac{1}{2}\right)^k$.

More realistically, *m* may be determined for a desired *n* and *p*: $m = -\frac{n \ln p}{(\ln 2)^2}$ (and $k = \lg \frac{1}{p}$).

What interesting property can be expected for an optimally "configured" Bloom filter?

(*m* coupon types, *nk* cereal boxes . . . how many 0's and 1's in bit array?)

Aside: A high-level survey of the broad range of hashing techniques and applications

https://dl-acm-org.ezproxy.uta.edu/citation.cfm?id=3047307