CSE 5311 Notes 9: Hashing

(Last updated 7/5/15 1:07 PM)

CLRS, Chapter 11

Review: 11.2: Chaining - related to perfect hashing method

11.3: Hash functions, skim universal hashing

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):

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

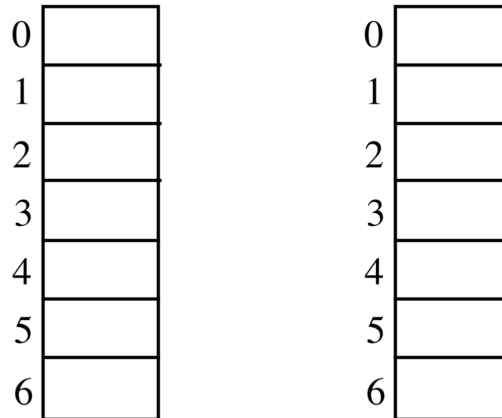
   $i_j\ i_{j+1}\ i_{j+2} \ldots$

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?



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:

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

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

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

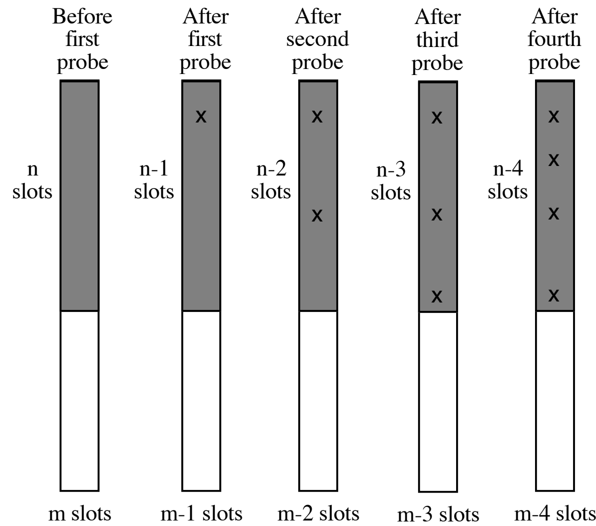
Load Factor $= \alpha = \dfrac{\#\text{ elements stored}}{\#\text{ slots in table}}$

When a very small α is desirable, constant-time initialization may be a useful trade-off - see p. 37 of `http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=2597757.2535933`

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 $\quad 1$

b. Probability that search goes on to a second probe $\quad \alpha = \frac{n}{m}$

c. Probability that search goes on to a third probe $\quad \alpha \frac{n-1}{m-1} < \alpha \frac{n}{m} < \alpha^2$

d. Probability that search goes on to a fourth probe $\quad \alpha \frac{n-1}{m-1} \frac{n-2}{m-2} < \alpha^2 \frac{n-2}{m-2} < \alpha^3$

. . .

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 α

   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):

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

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

$$\leq \frac{m}{n}\int_{m-n}^{m}\frac{1}{x}dx \qquad \text{Upper bound on sum for decreasing function. CLRS, p. 1067 (A.12)}$$

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

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 all keys, is minimized.

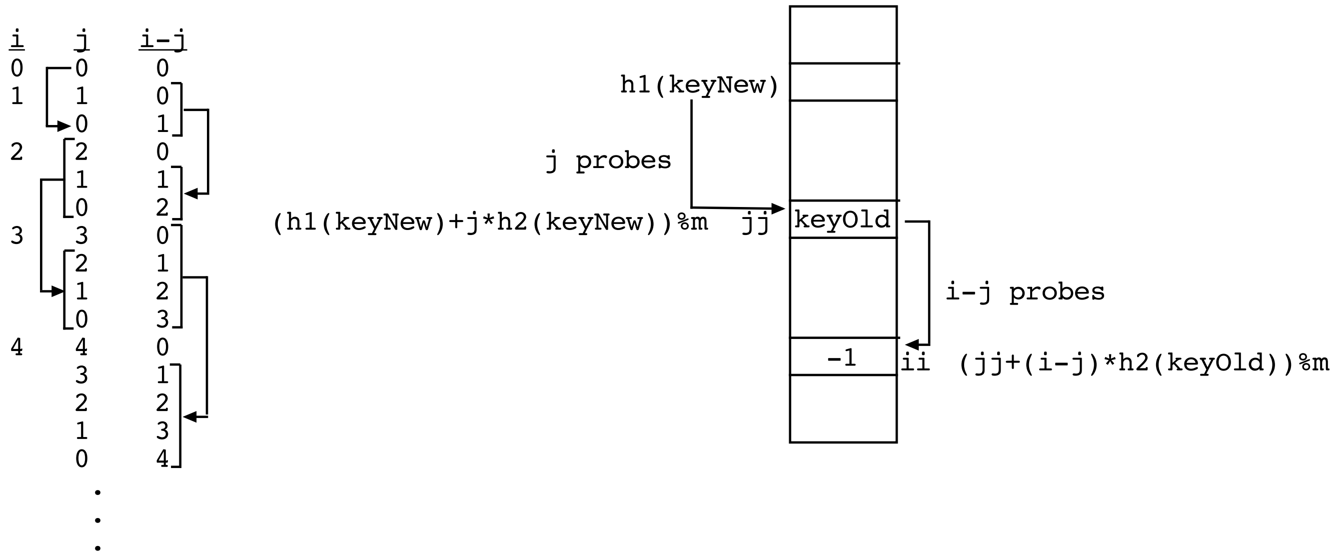
Expected probes for successful search ≤ 2.5. (Assumes uniform access probabilities.)

`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

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

Unsuccessful search performance is the same.




```
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;
    }
  }
}
}
```

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 (part of solution)
0					
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

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".

CUCKOO HASHING (aside, not in book)

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 tables, 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
```

http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf

PERFECT HASHING (CLRS 11.5)

Static key set

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

1. Preprocessing (constructing hash functions)

   and/or

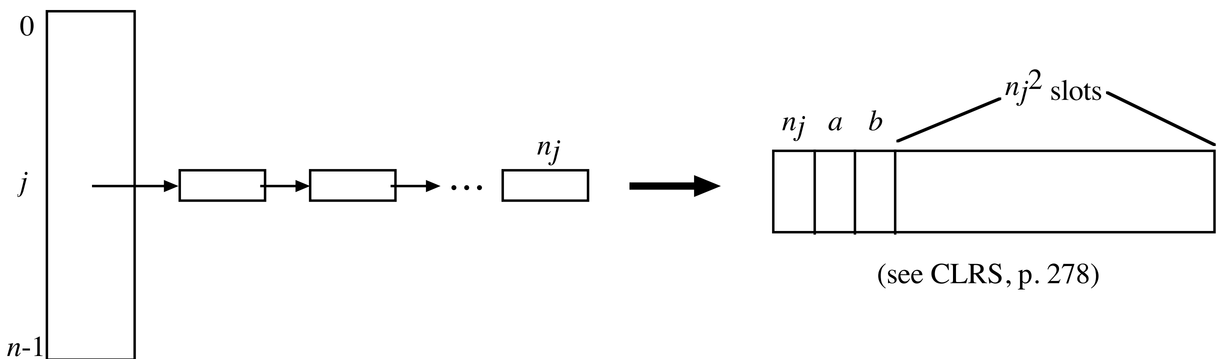
2. Extra space (makes success more likely) - want cn, where c is small

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 19.6 million keys

```
0.910 l.f. expected=1.861 CPU 5.681
0.920 l.f. expected=1.895 CPU 5.877
0.930 l.f. expected=1.934 CPU 6.101
0.940 l.f. expected=1.979 CPU 6.365
0.950 l.f. expected=2.031 CPU 6.693
0.960 l.f. expected=2.097 CPU 7.149
0.970 l.f. expected=2.188 CPU 7.934
0.980 l.f. expected=2.336 CPU 9.922
Retrievals took 3.525 secs
Worst case probes is 372
Probe counts:
Number of keys using 1 probes is 9610647
Number of keys using 2 probes is 4751688
Number of keys using 3 probes is 2247617
Number of keys using 4 probes is 1153021
Number of keys using 5 probes is 635040
Number of keys using 6 probes is 376082
Number of keys using 7 probes is 231966
Number of keys using 8 probes is 151831
Number of keys using 9 probes is 102159
Number of keys using 10 probes is 71484
Number of keys using 11 probes is 51254
Number of keys using 12 probes is 38126
Number of keys using 13 probes is 28381
Number of keys using 14 probes is 22448
Number of keys using 15 probes is 17626
Number of keys using 16 probes is 14066
Number of keys using 17 probes is 11247
Number of keys using 18 probes is 9468
Number of keys using 19 probes is 7914
Number of keys using 20 probes is 6766
Number of keys using 21 probes is 5731
Number of keys using 22 probes is 4904
Number of keys using 23 probes is 4295
Number of keys using 24 probes is 3714
Number of keys using 25 probes is 3339
Number of keys using 26 probes is 2858
Number of keys using 27 probes is 2677
Number of keys using 28 probes is 2381
Number of keys using 29 probes is 2021
Number of keys using >=30 probes is 29251
```

CLRS Perfect Hashing - 20 million keys

```
malloc'ed 240000000 bytes
realloc for 252870512 more bytes
final structure will use 16.644 bytes per key
Subarray statistics:
Number with 0 keys is 7359244
Number with 1 keys is 7355361
Number with 2 keys is 3678240
Number with 3 keys is 1227685
Number with 4 keys is 306079
Number with 5 keys is 61508
Number with 6 keys is 10160
Number with 7 keys is 1514
Number with 8 keys is 193
Number with 9 keys is 14
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 10.178
Time to retrieve each key once 6.280
```

RIVEST'S OPTIMAL HASHING (not in book)

Application of the *assignment problem* (weighted bipartite matching)

m rows, n columns ($m \le n$), positive weights (some may simulate ∞)

Choose an entry in each row of M such that:

No column has two entries chosen.

The *sum* of the m chosen entries is minimized (similar to binary tree hashing).

Aside: Solved using the *Hungarian method* in $O\left(m^2 n\right)$ time

(https://en.wikipedia.org/wiki/Hungarian_algorithm,
http://www.amazon.com/Stanford-GraphBase-Platform-Combinatorial-Computing/dp/0201542757)

or iterative use of Floyd-Warshall to find sub-optimal negative cycles
(http://ranger.uta.edu/~weems/NOTES5311/assignFW.c, Notes 11 reviews, CSE 2320 Notes 16 details)
in $O\left(n^4\right)$ time

Minimizing *expected* number of probes by *placement* of keys to be retrieved by a specific open addressing technique (i.e. finishes what other methods started):

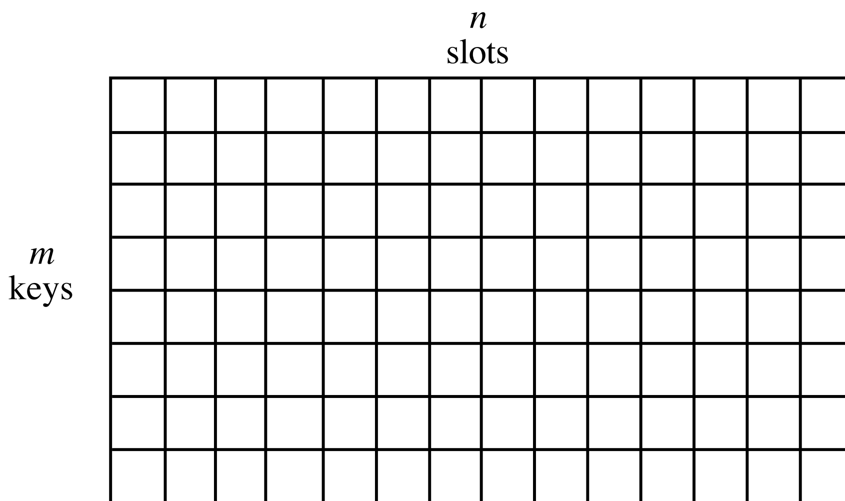
Static set of m keys for table with n slots. Key i has request probability P_i.

Follow m-prefix of probe sequence for key i: $S_{i1}, S_{i2}, \cdots, S_{im}$.

Assign weight jP_i to $M_{iS_{ij}}$. (Unassigned entries get ∞.)

Solve resulting assignment problem.

If M_{ij} is chosen, key i is stored at slot j.



Minimizing *worst-case* number of probes (i.e. alternative to perfect hashing):

No probabilities are needed.

Similar approach to expected case, but now assign m^j to $M_{iS_{ij}}$.

Any matching that uses a smaller maximum j must have a smaller total weight:

$$m^j + \text{other costs} > m \bullet m^{j-1}$$

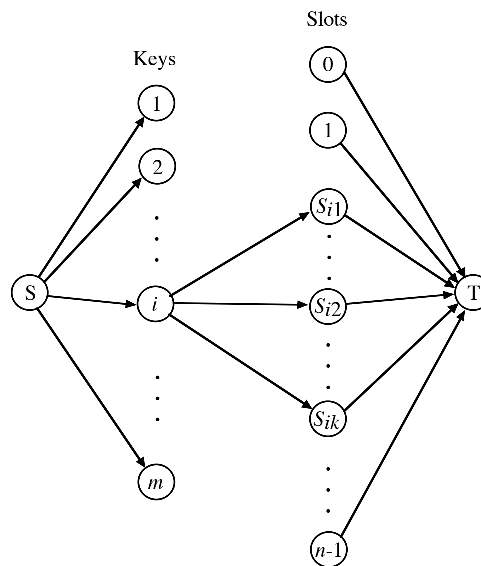
Wide range of magnitudes will be inconvenient in using Hungarian method, but result will not leave empty elements in the prefix $S_{i1}, S_{i2}, \cdots, S_{ij-1}$ for a selected $M_{iS_{ij}}$.

Also possible to use binary search on instances of (unweighted) bipartite matching:

low = 1
high = m // Can also hash all m keys to get a better upper bound
while low ≤ high
  k = (low + high)/2
  Generate instance of bipartite matching that connects each key to k-prefix of that key's probe sequence.
  Attempt to find matching with m edges.
  if successful
    high = k - 1
    Save this matching since it may be the final solution.
  else
    low = k + 1
< low is the worst-case number of probes needed>

Iteratively patch solution so that prefix elements of probe sequences are not left empty.

Generating instance of bipartite matching as maximum flow problem (unit capacity edges):



BLOOM FILTERS (not in book)

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

k independent hash functions

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 $p = \left(\frac{1}{2}\right)^k$.

Instead, *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 . . .)

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

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

B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors", *C.ACM 13 (7)*, July 1970, 422-426. (http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=362686.362692)