

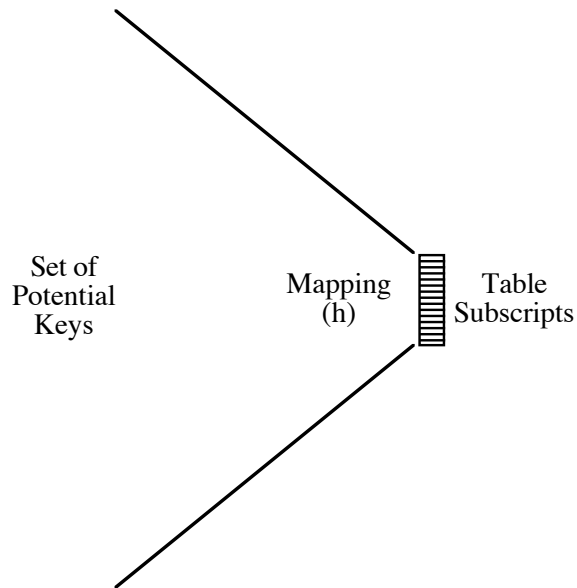
CSE 3318 Notes 12: Hashing

(Last updated 10/12/22 7:17 PM)

CLRS 11.1-11.4 (skip 11.3.4-11.3.5)

12.A. CONCEPTS

Goal: Achieve faster operations than balanced trees (nearly $O(1)$ expected time) by using “randomness” in key sets by sacrificing 1) generality and 2) ordered retrieval.



Regardless of the hash function, a dynamic set of keys will lead to *collisions*.

Birthday paradox (https://en.wikipedia.org/wiki/Birthday_problem)

366 different birthdays available

How many (random) persons are needed to have at least even odds of two persons with the *same* birthday? 23

Probability of k persons having k different birthdays is $\prod_{i=1}^{k-1} \frac{366-i}{366}$

probability of unique birthdays among 0 people is 1
probability of unique birthdays among 1 people is 1
probability of unique birthdays among 2 people is 0.997268
probability of unique birthdays among 3 people is 0.991818
probability of unique birthdays among 21 people is 0.557221
probability of unique birthdays among 22 people is 0.525249
probability of unique birthdays among 23 people is 0.493677
probability of unique birthdays among 24 people is 0.462654
probability of unique birthdays among 57 people is 0.0100102
probability of unique birthdays among 58 people is 0.00845124

12.B. HASH FUNCTIONS

Modular (AKA remaindering or division method)

$$h(\text{key}) = \text{key} \% m$$

m is the table size

Folklore: Make m prime, regardless of collision handling technique. Double hashing requires.

Multiplicative

$$\text{hash} = m * (0.710123587 * \text{key} - (\text{int})(0.710123587 * \text{key}));$$

Universal Hashing - aside

Use parameterized hash function to minimize chance of getting collisions beyond expectation.

Parameters are randomly generated when hash structure is initialized.

Text Strings as Key

```
scanf("%s",str);
hash=0;
for (i=0;
    str[i]!=0;
    i++)
    hash = (hash*10 + str[i]) % m;
printf("%s => %d\n",str,hash);
```

A string's *signature* may be stored in a non-hashing data structure. (Exercise 11.3-1)

12.C. COLLISION HANDLING BY CHAINING

Concept – Use table of pointers to unordered linked lists. Elements of a list have the same *signature*.

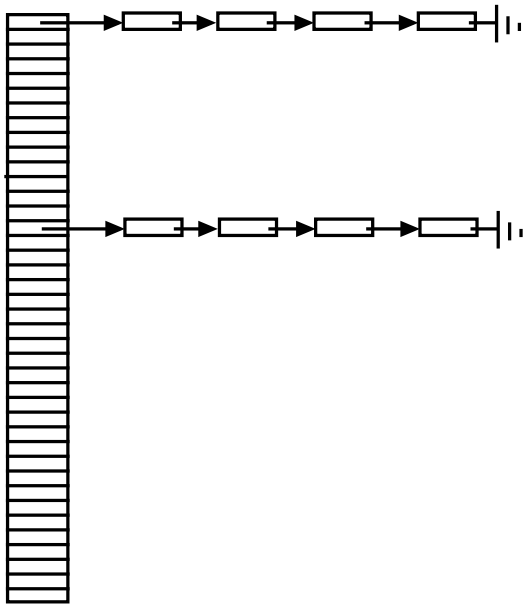
$$\text{Load Factor} = \alpha = \frac{\# \text{ elements stored}}{\# \text{ slots in table}}$$

(Sometimes stated as a per cent.)

For some methods, such as chaining, α can exceed 1.

Expected probes is $\frac{n}{2m} = \frac{\alpha}{2}$ for hits and $\frac{n}{m} = \alpha$ for misses.

Aside: Reorganization



12.D. COLLISION HANDLING BY OPEN ADDRESSING

Saves space when records are small, so 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 often used: *never-used* (-1) and *recycled* (-2). Searches stop on *never-used*, but continue on *recycled* (“deleted” in CLRS). (Aside: For linear probing, but not double hashing - can also reinsert records past the emptied slot for a deletion. See CLRS p. 302)

Linear Probing - $h(\text{key}, i) = (\text{key} + i) \% m$ (<https://ranger.uta.edu/~weems/NOTES3318/hashLP.c>)

Properties:

1. Probe sequences trivially hit all slots.
2. Probe sequences wrap back to beginning of table.
3. Long *clusters* of contiguous occupied slots are costly for misses.
(https://en.wikipedia.org/wiki/Primary_clustering)
4. There are only m probe sequences. Two keys hashing to same initial slot have *the same probe sequence*.

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

0		0	
1		1	
2		2	
3		3	
4		4	
5		5	
6		6	

Double Hashing – $h(\text{key}, i) = (h_1(\text{key}) + i * h_2(\text{key})) \% m$
 (<https://ranger.uta.edu/~weems/NOTES3318/hashDH.c>)

Properties:

1. Probe sequences will hit all slots only if m is prime.
2. $m(m - 1)$ probe sequences (based on interaction of h_1 and h_2 values). Unlikely that two keys hashing to the same initial slot will have the same probe sequence.
3. Minimizes effect of clustering.

Typical Hash Functions:

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

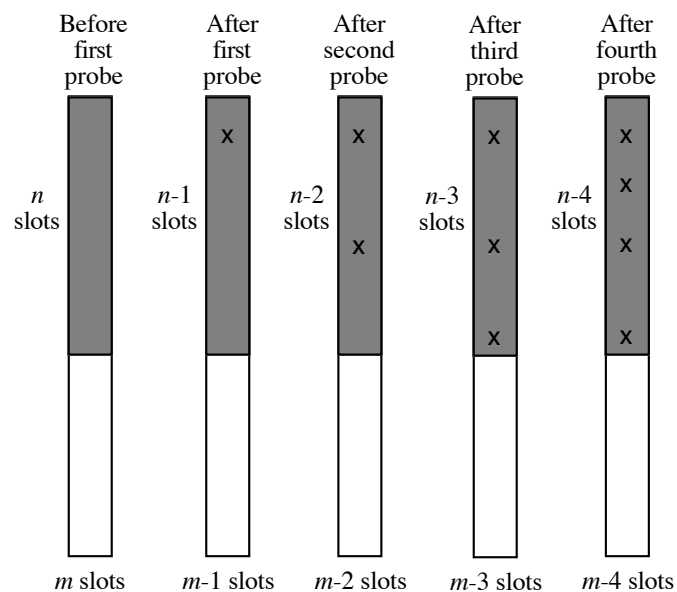
$$h_2 = 1 + \text{key} \% (m - 1)$$

		Key	h1	h2
0	1313			
1		1313	0	6
2	29	2626	0	11
3	136	24	11	1
4		19	6	8
5	2600	136	6	5
6	19	140	10	9
7	1305	23	10	12
8		29	3	6
9	23	2600	0	9
10	140	1305	5	10
11	2626			
12	24			

12.E. 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.

- Unsuccessful search when load factor is $\alpha = \frac{n}{m}$. Each successive probe has the effect of decreasing both the number of slots in the table and the number of occupied slots by one.



- a. Probability that a search has 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:

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

- 2. Inserting a key when load factor is α
 - a. Exactly like unsuccessful search
 - b. Upper bound of $\frac{1}{1-\alpha}$ 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 keys inserted into an empty table (each key is equally likely to be requested):

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \quad \text{Sum is } \frac{1}{m} + \frac{1}{m-1} + \dots + \frac{1}{m-n+1} \\ &= \frac{m}{n} \sum_{i=m-n+1}^m \frac{1}{i} \leq \frac{m}{n} \int_{m-n}^m \frac{1}{x} dx \quad \text{Upper bound on sum for decreasing function.} \\ &= \frac{m}{n} (\ln m - \ln(m-n)) = \frac{1}{\alpha} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

alpha 0.200	unsuccessful	(insert)	1.250	successful	1.116
alpha 0.250	unsuccessful	(insert)	1.333	successful	1.151
alpha 0.300	unsuccessful	(insert)	1.429	successful	1.189
alpha 0.350	unsuccessful	(insert)	1.538	successful	1.231
alpha 0.400	unsuccessful	(insert)	1.667	successful	1.277
alpha 0.450	unsuccessful	(insert)	1.818	successful	1.329
alpha 0.500	unsuccessful	(insert)	2.000	successful	1.386
alpha 0.550	unsuccessful	(insert)	2.222	successful	1.452
alpha 0.600	unsuccessful	(insert)	2.500	successful	1.527
alpha 0.650	unsuccessful	(insert)	2.857	successful	1.615
alpha 0.700	unsuccessful	(insert)	3.333	successful	1.720
alpha 0.750	unsuccessful	(insert)	4.000	successful	1.848
alpha 0.800	unsuccessful	(insert)	5.000	successful	2.012
alpha 0.850	unsuccessful	(insert)	6.667	successful	2.232
alpha 0.900	unsuccessful	(insert)	10.000	successful	2.558
alpha 0.910	unsuccessful	(insert)	11.111	successful	2.646
alpha 0.920	unsuccessful	(insert)	12.500	successful	2.745
alpha 0.930	unsuccessful	(insert)	14.286	successful	2.859
alpha 0.940	unsuccessful	(insert)	16.666	successful	2.993
alpha 0.950	unsuccessful	(insert)	20.000	successful	3.153
alpha 0.960	unsuccessful	(insert)	25.000	successful	3.353
alpha 0.970	unsuccessful	(insert)	33.333	successful	3.615
alpha 0.980	unsuccessful	(insert)	49.998	successful	3.992
alpha 0.990	unsuccessful	(insert)	99.993	successful	4.652

Aside (for those with IE 3301 background): “Fast and Powerful Hashing Using Tabulation”, CACM 60 (7), July 2017, <https://dl-acm-org.ezproxy.uta.edu/citation.cfm?id=3068772>