Hash Tables

CSCI232, Data Structures and Algorithms

A Google Interview Question

- Question: You are asked to design some software to display the top 10 search terms on Google
- Assumptions:
- 1. Google provides you a log file which records the search terms entered by every user
- 2. The length of every search term is less than 255 Bytes
- 3. There are 10 million records in the log file
- 4. Because there are many duplicated search terms, there are only 3 million unique ones
- 5. You could not use more than 1G memory

Hash Tables: Overview

- Provide very fast insertion and searching
  - Both are O(1)
  - Is this too good to be true?
- Disadvantages
  - Based on arrays, so the size must be known in advance
  - Performance degrades when the table becomes full
  - No convenient way to sort data
- Summary
  - Best structure if you have no need to visit items in order and you can predict the size of your database in advance

Hash Table

- Idea
  - Provide easy searching and insertion by mapping keys to positions in an array
  - This mapping is provided by a *hash function*
    - Takes the key as input
    - Produces an index/value as output
    - Save the key into an array at the index/value position
- The array is called a *hash table*
- Application: Dictionary, caching, database indexing, routing table, keyword recognition/filtering, uniqueness checking
- Do you remember the freqTable in Huffman Tree?

Hash Function: Example

- The easiest hash function is the following:
  - H(key) = key % tablesize
  - H(key) in [0, tablesize-1]
- So if we inserted the following keys into a table of size10:
  - 13,11456, 2001, 157
  - You probably already see potential for collisions

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

What have we accomplished?

- We have stored keys of an unpredictable large range into a smaller data structure
- To find a key *k*, just retrieve table[*i*] where *i*=H(*k*)
- To insert a key *k*, just set table[*i*] = *k* where *i*=H(*k*)
- Both are O(1)!
- Different from Array?

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

What's our price?

- Multiple values in our range could map to the same hash table index
- For example, if we used hash function:
  - $H(k) = k \% 10$
- Then, tried to insert 207
  - $H(207) = 7$
- We have a *collision* at position 7

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

How to solve collisions?

- If we use hash tables, we need to handle collisions by a couple ways:
  - Open addressing: linear probing, quadratic probing, and double hashing
  - Separate chaining

- Also, the choice of the hash function is important
  - We can produce hash functions which are more or less likely to have high collision frequencies
  - We'll look at potential options

Linear Probing

- Presumably, you will have define your hash table size to be 'safe', i.e., larger than the maximum amount of items you expect to store
- As a result, there should be some available cells

- In *linear probing*, if an insertion results in a collision, search sequentially until a empty cell is found
  - Use wraparound if necessary

Linear Probing: Insertion

- Again, say we insert element 207
- H(207) = 207 % 10 = 7
- This results in a collision with element 157
- So we search linearly for the next available cell, which is at position 8
  - And put 207 there

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

Linear Probing: Searching

- And searching, is not simply a matter of applying H(*k*)
  - You apply H(*k*), and probe!
- Searching 207?
- Inserting 426?
  - We would have to check three cells before finding a empty position 9

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	426

Linear Probing: Clusters

- Linear probing tends to result in clusters
  - Large amounts of cells in a row are populated
  - Large amounts of cells are empty
- This becomes worse as the table fills up
  - Degrades performance

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	426

Linear Probing: Clusters

- A cluster is like a 'faint scene' at a mall
  - The first arrivals come
  - Later arrivals come because they wonder why everyone was in one place
  - As the crowd gets bigger, more are attracted
- Same thing with clusters!

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	426

Exercise

- Show what happens when the keys

S	N	I	Y	J	B	A	G	Q
18	13	8	24	9	1	0	6	16

are inserted into a hash table with 11 buckets/slots, in that order, using linear probing to resolve the collisions. The hash function is defined as H(X) = X % size of the hash table.

Quadratic Probing

- The main problem with linear probing was clustering
- Quadratic probing attempts to address this
  - Instead of linearly searching for these next available cell
    - i.e., search cell x+1, x+2, x+3,….
  - Search quadratically
    - i.e., search cell x+1, x+4, x+9, …, x+i^2, …

- Idea (applied in network domain as well)
  - On a collision, initially assume a small cluster and go to x+1
  - If that's occupied, assume a larger cluster and go to x+4
  - If that's occupied assume an even larger cluster, and go to x+9

Quadratic Probing: Example

- Inserting 207
- H(207) = 207 % 10 = 7
- This results in a collision with element 157
- Slot 7 is occupied but slot 7+1=8 is open, so we put it there

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

Quadratic Probing

- Inserting 426
- H(426) = 426 % 10 = 6
- Slot 6+1=7 is also occupied
- So we check slot 6+4=10
  - This passes the end, so we wraparound to slot 0 and insert there

Index	Value
0	426
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

Quadratic Probing

- Clusters will tend to be smaller
  - Instead of having large clusters
  - And largely sparse areas

- Thus quadratic probing got rid of what we call *primary clustering*

Index	Value
0	426
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

Problem of Quadratic Probing

- *Secondary clustering* would happen if we inserted
  - 827, 10857, 707 1117
  - Because they all hash to 7
- Not as serious a problem as primary clustering
- But there is a better solution that avoids both

Index	Value
0	426
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

Double Hashing

- The problem is: probe sequences are always the same, e.g.,
  - Linear probing always generates x+1, x+2, x+3…
  - Quadratic probing always generates x+1, x+4, x+9…

- Solution – Double Hashing:
  - Hash the key once to get the location
  - Hash the key a second time to get the probe
  - Make both the hash location and the probe dependent upon the key

Second Hash Function

- Second hash function for the probe
  - It cannot be the same as the first hash function
  - It can NEVER hash to zero

- Experts have discovered the following type of hash function works for the probe:
  - $\boldsymbol{probe} = \boldsymbol{c} - (\boldsymbol{key} \% \boldsymbol{c})$
  - Where ***c*** is a prime number that is smaller than the array size

Double Hashing: Example

- Inserting 207
- H(207) = 207 % 10 = 7

- This results in a collision with element 157
- We hash again to get probe
  - Suppose we choose *c*=5
    - P(207) = 5 – (207 % 5)
    - P(207) = 5 – 2 = 3

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

Double Hashing: Example

- We insert 207 at position:
  - H(207) + P(207)
  - 7+3 = 10
- Wrapping around, this will put 207 at position 0

Index	Value
0	207
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

Double Hashing: Example

- Inserting value 426
- We run the initial hash:
  - H(426) = 426 % 10 = 6
- Get a collision, we probe:
  - P(426) = 5 – (426 % 5)
  - 5 – 1 = 4
- Insert at location:
  - H(426) + P(426) = 10
  - Wrapping around, we get 0
  - Another collision!

Index	Value
0	207
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

Double Hashing: Example

- We probe again with the same probe 4
- We insert 426 at location 0+4 = 4, and this time there is no collision

- Double hashing will produce the fewest clusters
  - Because both the hash and probe are key-dependent

Index	Value
0	207
1	2001
2	
3	13
4	426
5	
6	11456
7	157
8	
9	

Note...

- What is a potential problem with choosing a hash table of size 10 and a number *c* of 5 for the probe?

- Suppose we had a key k where $H(k) = 0$ and $P(k) = 5$
  - i.e., $k = 0$

- What would the probe sequence be?
- What's the problem?

Probe Sequence of Double Hashing

- The probe sequence may never find an open cell!
- Because H(0) = 0, we'll start at hash location 0
  - If we have a collision, P(0) = 5 so we'll next check 0+5=5
  - If we have a collision there, we'll next check 5+5=10, with wraparound we get 0
  - We'll infinitely check 0 and 5, and never find an open cell!

What is the problem?

- The table size is not prime!
  - If the size were 11
  - Locations after re-hashing: 0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6
  - If there is one open cell, the probing is guaranteed to find it

Index	Value
0	207
1	2001
2	
3	13
4	426
5	
6	11456
7	157
8	
9	
10	

Double Hashing Requirements

- It is important to have the following requirement for double hashing: the table size is *prime*
  - So our previous table size of 10 is not a good idea
  - We would want 11, or 13, etc.

- Generally, for open addressing, double hashing is the best
  - Open addressing means there are some open spaces in the array
  - What if you increase the size of the table?
  - What if you use a linked list/tree at each position of the array?

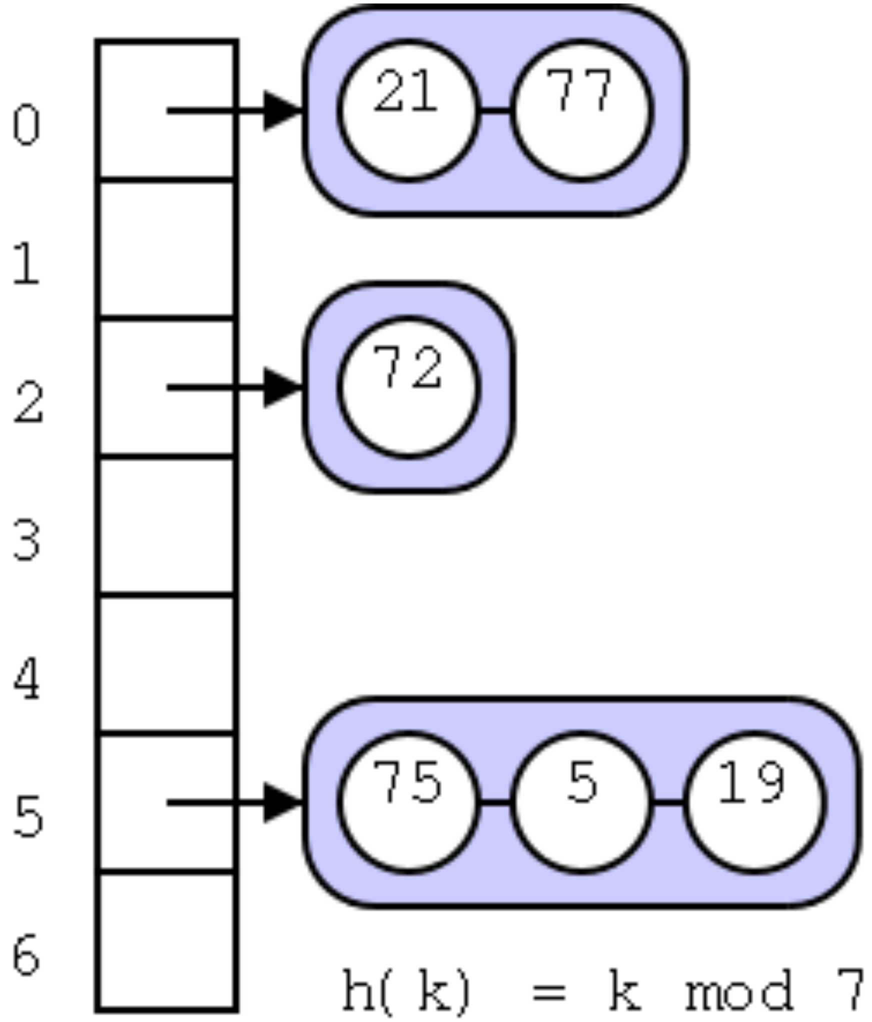
Re-Hash

- If the table becomes full enough, double its size
- Note this is not quite as simple as it seems, why?

Index	Value
0	
1	2001
2	
3	
4	
5	
6	426
7	207
8	
9	
10	
11	
12	
13	13
14	
15	
16	11456
17	157
18	
19	
20	

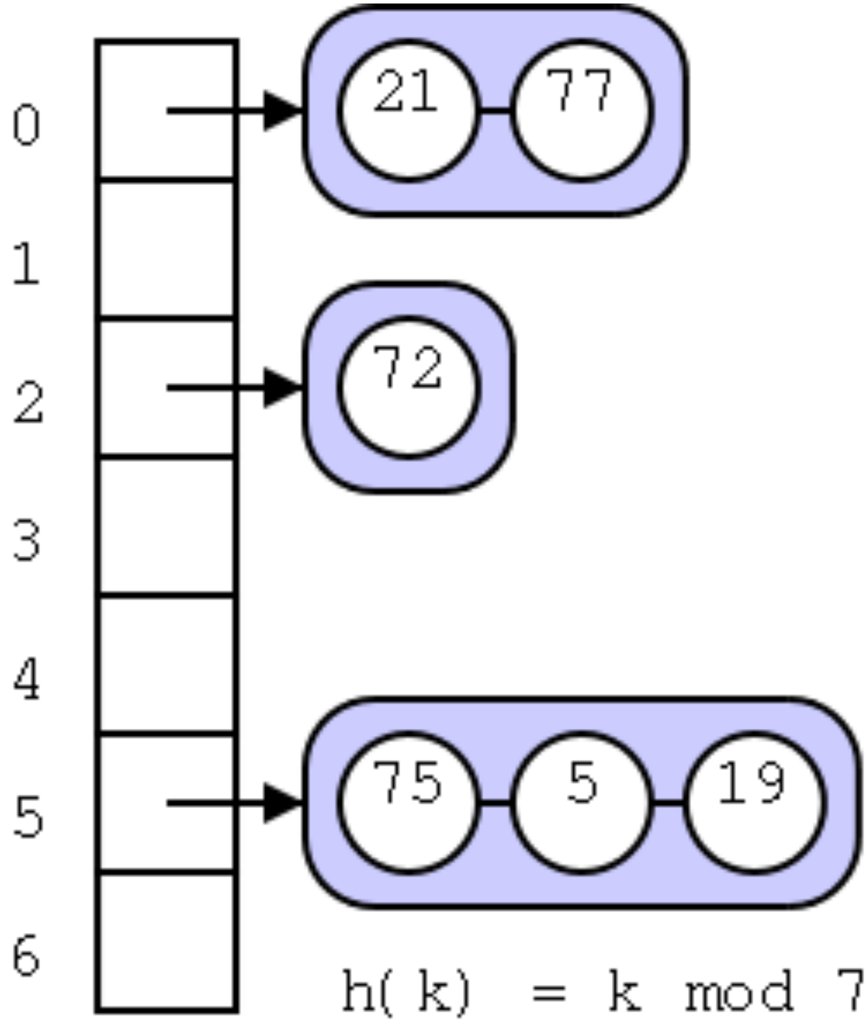
Separate Chaining

- The alternative to open addressing
- Does not involve probing to different locations in the hash table
- Rather, every location in the hash table contains a linked list of keys



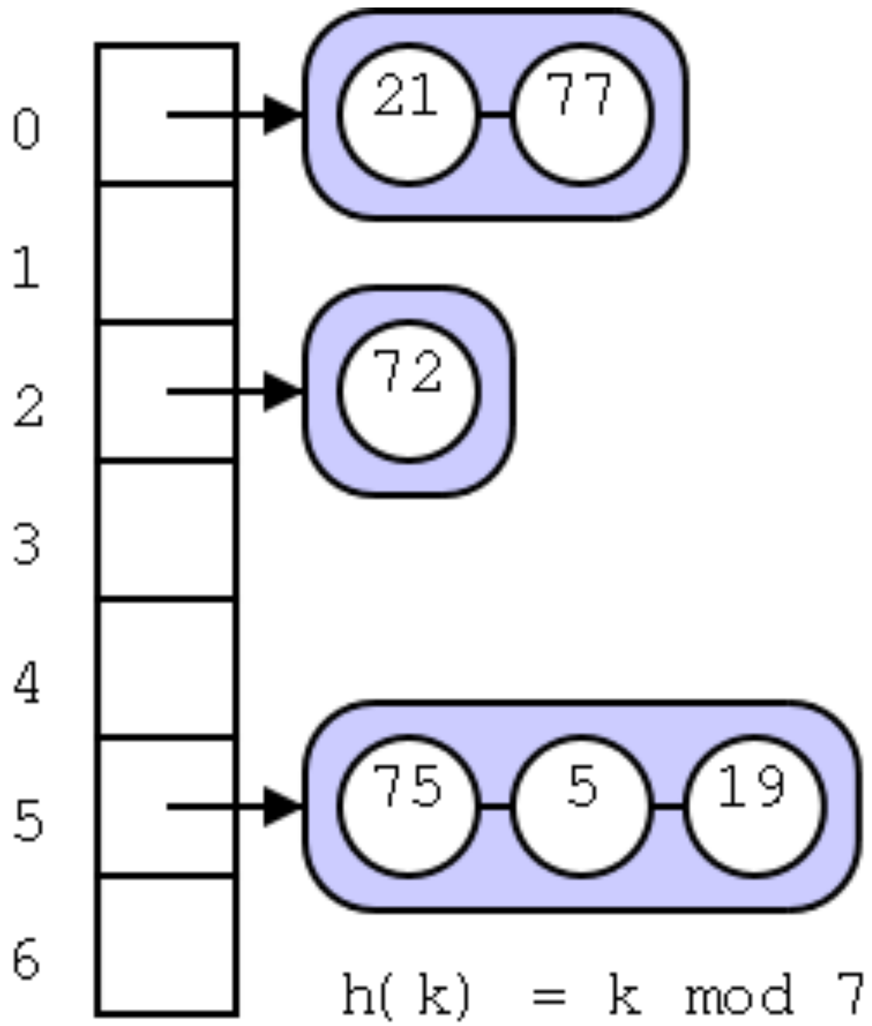
Separate Chaining

- Simple case, 7 element hash table
- H(k) = k % 7
- So:
  - 21, 77 each hash to location 0
  - 72 hashes to location 2
  - 75, 5, 19 hash to location 5
- Each is simply appended to the correct linked list



Separate Chaining

- In separate chaining, trouble happens when a list gets too full
- Generally, we want to keep the size of the biggest list, call it M, much smaller than N
  - Searching and insertion will then take O(M) time in the worst case



A Good Hash Function

- Has two properties:
  - Is computable quickly; so as not to degrade performance of insertion and searching
  - Can take a range of key values and transform them into indices such that the key values are distributed randomly across the hash table

- For random keys, the modulo (%) operator is good
- It is not always an easy task!

For example…

- Data can be highly non-random
- For example, a car-part ID:
  - 033-250-03-94-05-0-535

- For each set of digits, there can be a unique range or set of values! Thus it is not random.
  - i.e. Digits 3-5 could be a category code, where the only acceptable values are 100, 150, 200, 250, up to 850.
  - Digits 6-7 could be a month of introduction (0-12)
  - Digit 12 could be "yes" or "no" (0 or 1)
  - Digits 13-15 could be a checksum, a function of all the other digits in the code

Rule #1: Don’t Use Non-Data

- Compress the key fields down enough until every bit counts
- For example:
  - The category (bits 3-5, with restricted values 100, 150, 200, … , 850) counting by 50s needs to be compressed down to run from 0 to 15
  - The checksum is not necessary, and should be removed. It is a function of the rest of the code and thus redundant with respect to the hash table

Rule #2: Use All of the Data

- Every part of the key should contribute to the hash function
- More data portions that contribute to the key, more likely it will be that the keys hash evenly
  - Saving collisions, which cause trouble no matter what the algorithm you use

Rule #3: Use a Prime Number for Modulo Base

- This is a requirement for double hashing
- Important for quadratic probing
- Especially important if the keys may not be randomly distributed
    - The more keys that share a divisor with the array size, the more collisions
    - Example, non-random data which are multiples of 50
        - If the table size is 50, they all hash to the same spot
        - If the table size is 10, they all hash to the same pot
        - If the table size is 53? Better!

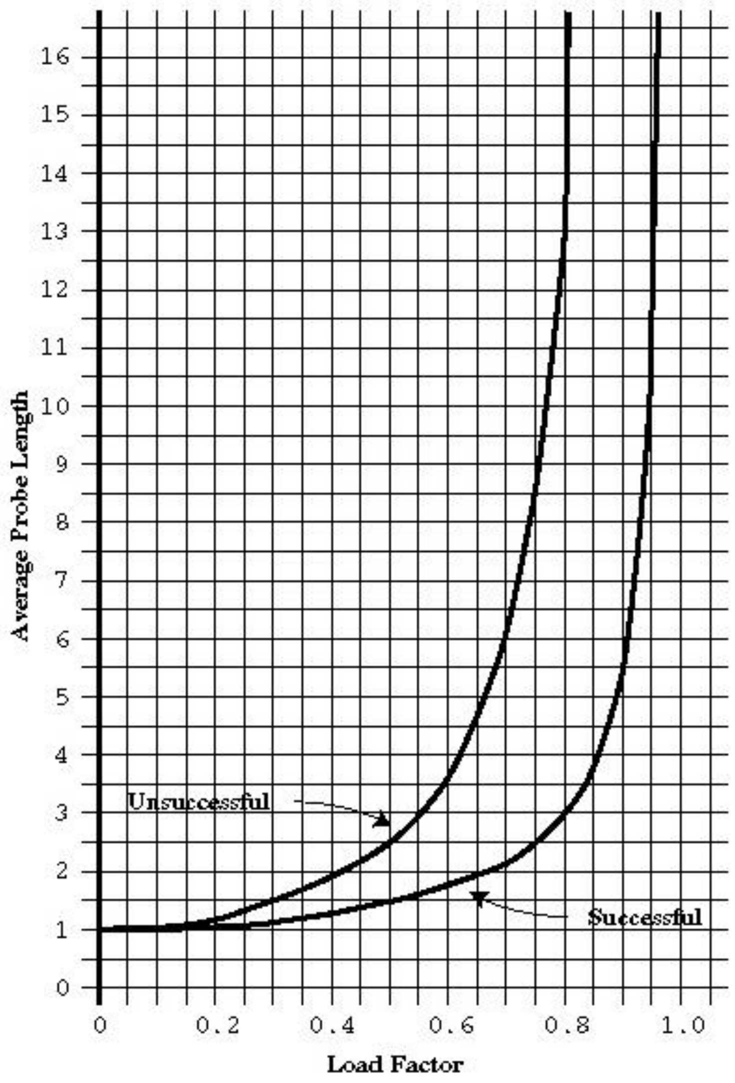
Hashing Efficiency

- Insertion and Searching are O(1) in the best case
  - This implies no collisions
  - If you minimize collisions, you can approach this runtime

- If collisions occur:
  - Access times depend on resulting probe lengths
  - Every probe equals one more access
  - So every worst case insertion or search time is proportional to:
    - The number of required probes if you use open addressing
    - The number of links in the longest list it you use separate chaining

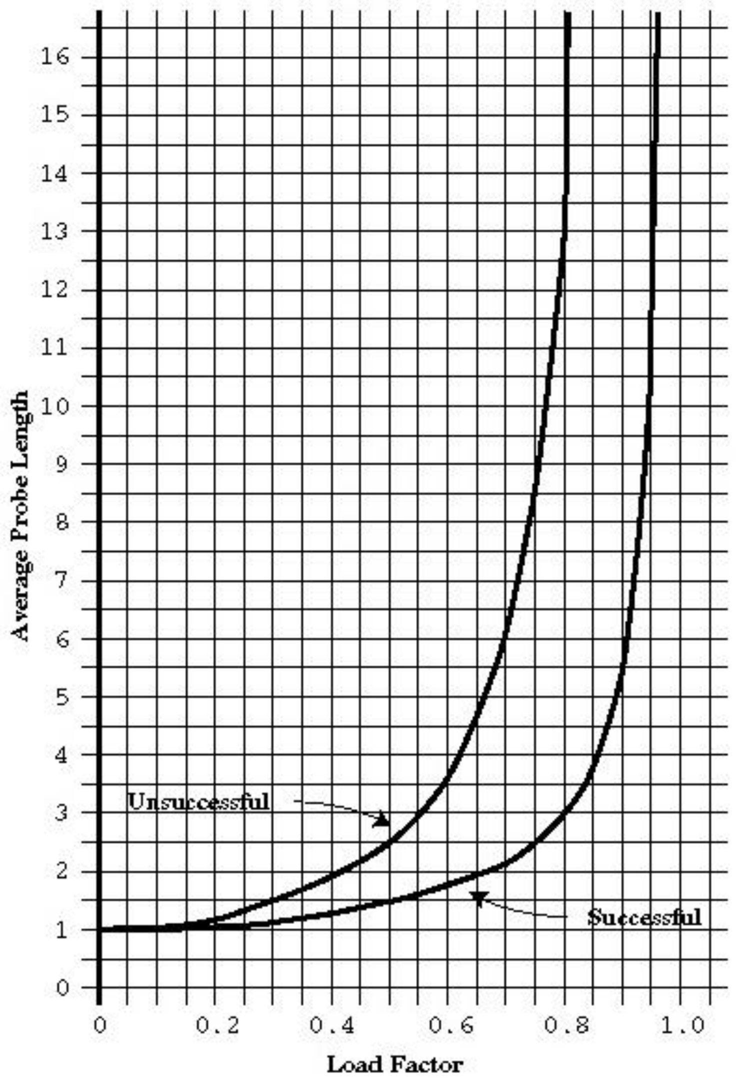
Efficiency: Linear Probing

- Let's assume a load factor L, where L is the percentage of hash table slots which are occupied.
- Knuth showed that, for a successful search:
  - $P = (1 + 1 / (1 - L)^2) / 2$
- For an unsuccessful search:
  - $P = (1 + 1 / (1 - L)) / 2$



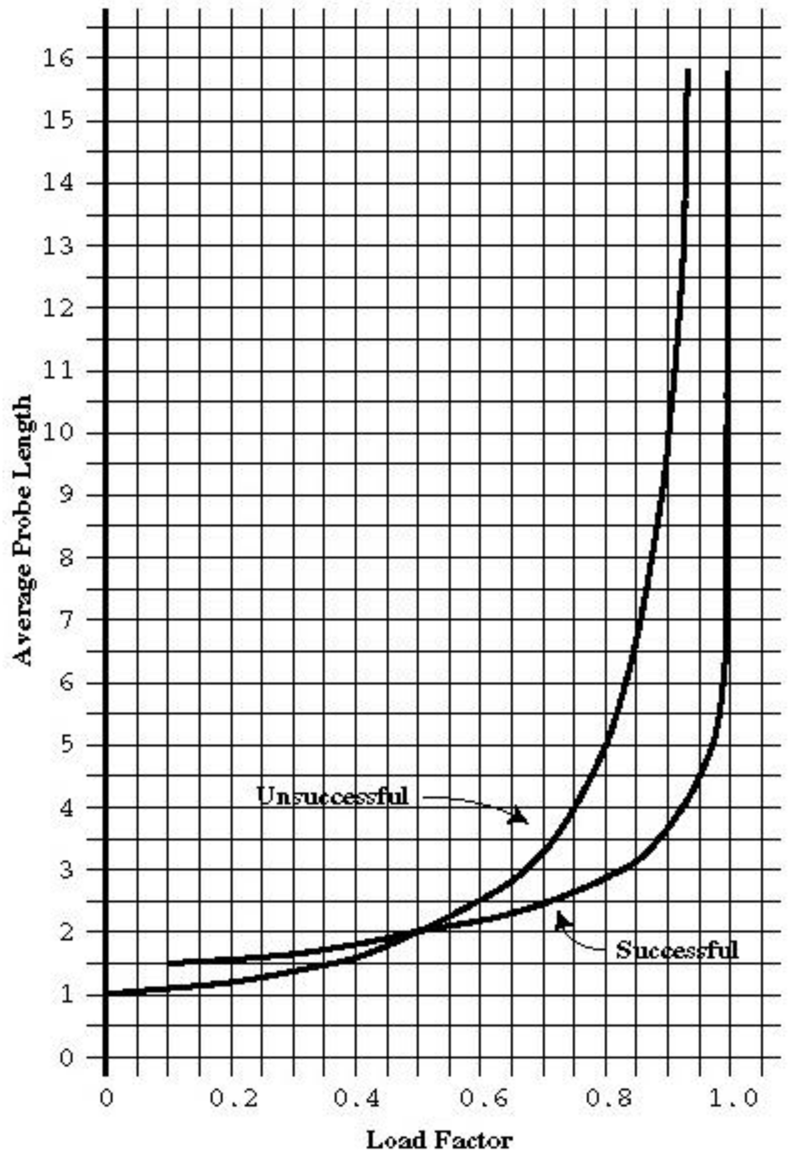
Efficiency: Linear Probing

- What's the ideal load factor?
- At L=0.5:
  - Successful search takes 1.5 probes
  - Unsuccessful takes 2.5
- At L = 2/3:
  - Successful: 2.0
  - Unsuccessful: 5.0
- Good to keep load factor under 0.5!



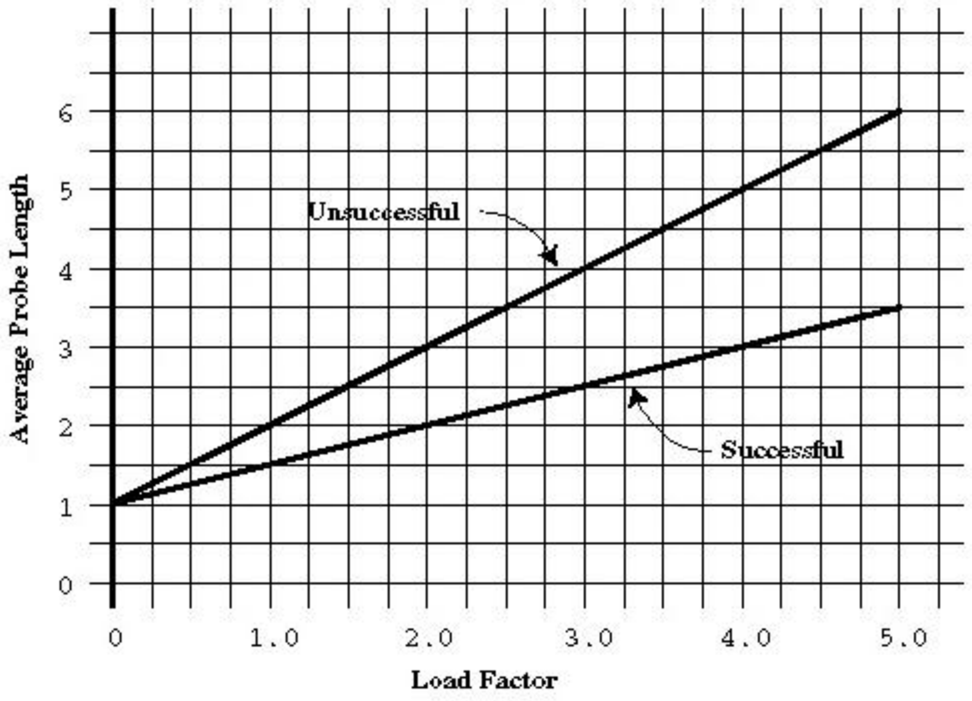
Efficiency: Quadratic Probing and Double Hashing

- Again, assume a load factor L, where L is the percentage of hash table slots which are occupied.
- Knuth showed that, for an unsuccessful search:
  - P = 1 / (1 – L)
- For an successful search:
  - P = -log(1-L) / L
- Can tolerate somewhat higher L



Efficiency: Separate Chaining

- Here L is a bit more complicated:
  - For N elements
  - And an array of size S
  - L = N / S
- Successful (average):
  - 1 + (L/2)
- Unsuccessful:
  - 1 + L



Summary: When to use What

- If the number of items that will be inserted is uncertain, use separate chaining
  - Must create a LinkedList class
  - But performance degrades only linearly with increased L
  - With open addressing, major penalties for high L

- Otherwise, use double hashing, unless…
  - Plenty of memory available
  - Low load factors
  - Then linear or quadratic probing should be done for ease of implementation