Data Structures & Algorithms 2

Topic 6 – Hash Tables

Lecturer: Dr. Hadi Tabatabaee
Materials: Dr. Phil Maguire
Maynooth University
Online at http://moodle.maynoothuniversity.ie

Overview

Aims

- Learn different types of hashing
- Learn to use hash tables to store and retrieve data

Learning outcomes: You should be able to…

- Learn the advantages and disadvantages of Hashing
- Learning different hashing methods
- Learn how to deal with collisions

Motivation

Suppose we have 16,000,000 data items. We would like to be able to find a particular item.

- How long will it take to find the item?

It depends on the data structure.

Motivation

If the data structure is a linked list and items are not sorted, the search time is O(N).

If the data structure is a linked list and items are sorted, the search time of a binary search is $O(\log_2 N)$.

If the data structure is a binary search tree, the search time can be reduced to $O(\log_2 N)$.

log2 16,000,000 ≈ 24

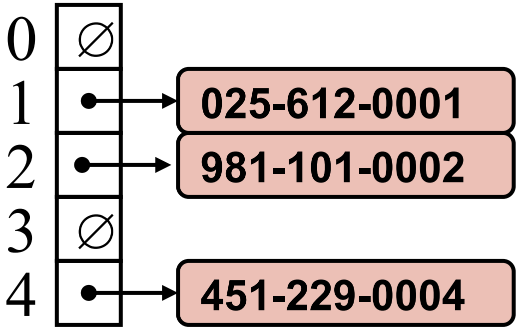
Motivation

Can we do even better than $O(\log_2 N)$?

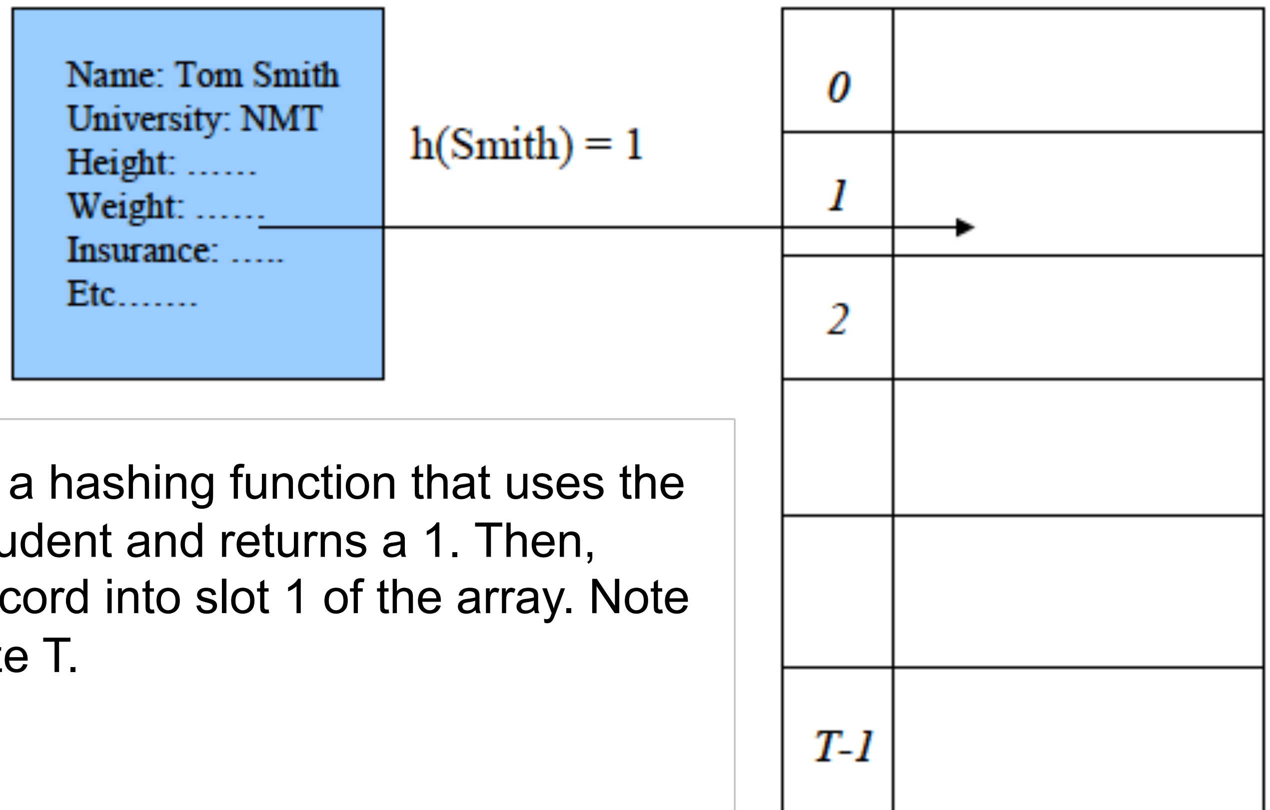
Yes we can. We can use “hash” functions.

Hashing

- The general idea behind hashing is to directly map each data item into a position in a table using some function *h*.
  - Complex data items must have a “key” in order to perform the hashing. Then: position = h(key).
- Generally, we use an array of some fixed size to hold the data.



Example



Data Item

Suppose we use a hashing function that uses the last name of a student and returns a 1. Then, place the data record into slot 1 of the array. Note the array is of size T.

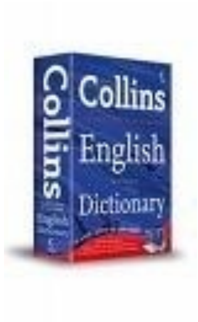
Hash Tables

- A hash table is a data structure that offers very fast insertion and searching.
- No matter how many data items there are, insertion and searching take close to constant time, nearly instantaneous or O(1)
- The disadvantage is that because they are based on arrays, they are difficult to expand
- Performance may degrade when a table becomes too full
- Items cannot be visited in any order
- However, if you know the size of your database and don't need to access items in order hash tables are excellent.

Hashing

- Hashing is used when the items to be stored do not have any obvious array index where you would look them up (what page is “cat” on in the dictionary?)
- The basic idea is that a range of key values is transformed into a range of array index values.
- The trick is that you can figure out what the appropriate array index is without having to do a binary search O(log N)

Example



- Let’s say we want to store a 50,000-word English language dictionary in memory.
- Ideally, every word should occupy its own slot in the 50,000 size array, making access very fast.
- But what’s the relationship of index numbers to words?
- For example, what slot do we access to find the definition for a word like *cats*?
- It’s not immediately obvious – we need some system

Conversion

- One possible system for converting words into an index number might be to add up the alphabetic number of each letter.
- For example, we could add up to convert cats into an index.
  - C = 3
  - A = 1
  - T = 20
  - S = 19

- This gives us a total of 43, indicating that cats will be stored in slot number 43 of the array according to this system.
- Now we can look up slot 43 instantly, without doing a linear search or even a binary search of the array.

Example

How good is this system?

- One problem is that the range of possible values is low
  - Assuming a limit of 10 letters, then the biggest possible index is 26 x 10 (zzzzzzzzzz)
- That means we have to squash ~50,000 words into just 260 slots in an array.
- With an average of 192 words per slot, we would have to perform a linear search through them, degrading the performance.
- Another problem is that the words would not be evenly distributed – some slots might have thousands of words.
  - Hundreds of words sum to the exact total as cats
  - Was, tin, give, tend, etc.

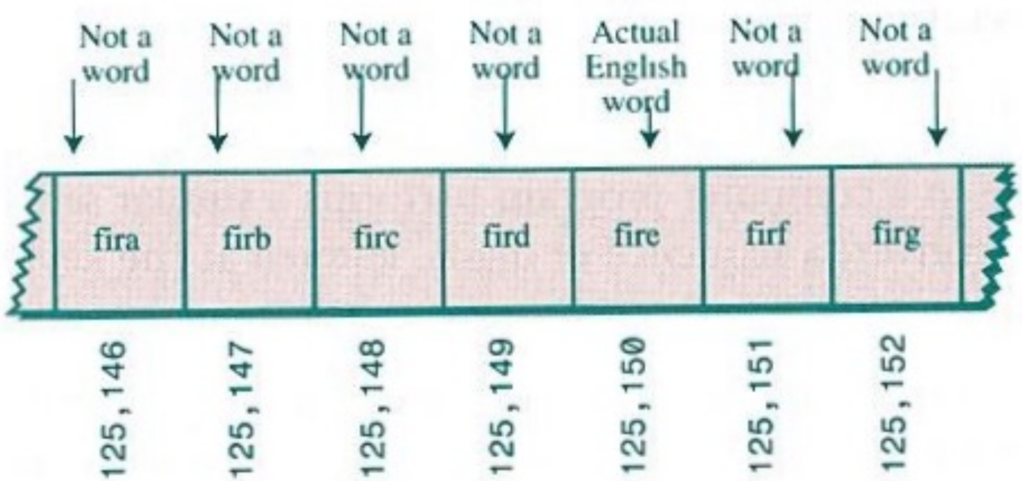
Better Idea

- To increase the range, we could try multiplying each letter instead of adding them
- If we multiply each letter by successive powers of 27, each letter is guaranteed to map to a separate index.
  - Cats would be $3*27^0 + 1*27^1 + 20*27^2 + 19*27^3 = 388{,}587$
  - Zzzzzzzzzz would be $26*27^0 + 26*27^1 + 26*27^2 + 26*27^3 + 26*27^4$

    $+ 26*27^5 + 26*27^6 + 26*27^7 + 26*27^8 + 26*27^9 = 7$ trillion
- Problem – the array is ridiculously big, and most slots will be empty!

Waste of Space

- Every single letter permutation is assigned a slot in the array.
- But most random permutations of letters do not form words!



Hashing

- What we need is a system to compress a huge range of numbers we obtain from the numbers-multiplied-by-powers system into a range that matches a reasonably sized array.
- If our dictionary has 50,000 words, then it's best to pick a storage array twice this size to maximize the probability that every word gets its own slot.
- A modulo operator (%) can be used to map a range of 0 to 7 trillion into a range of 0 to 99,999.
- If we apply a modulo of 100,000, then this means that every time the bigger number goes over this, it wraps back to 0, guaranteeing that it will be in the correct range

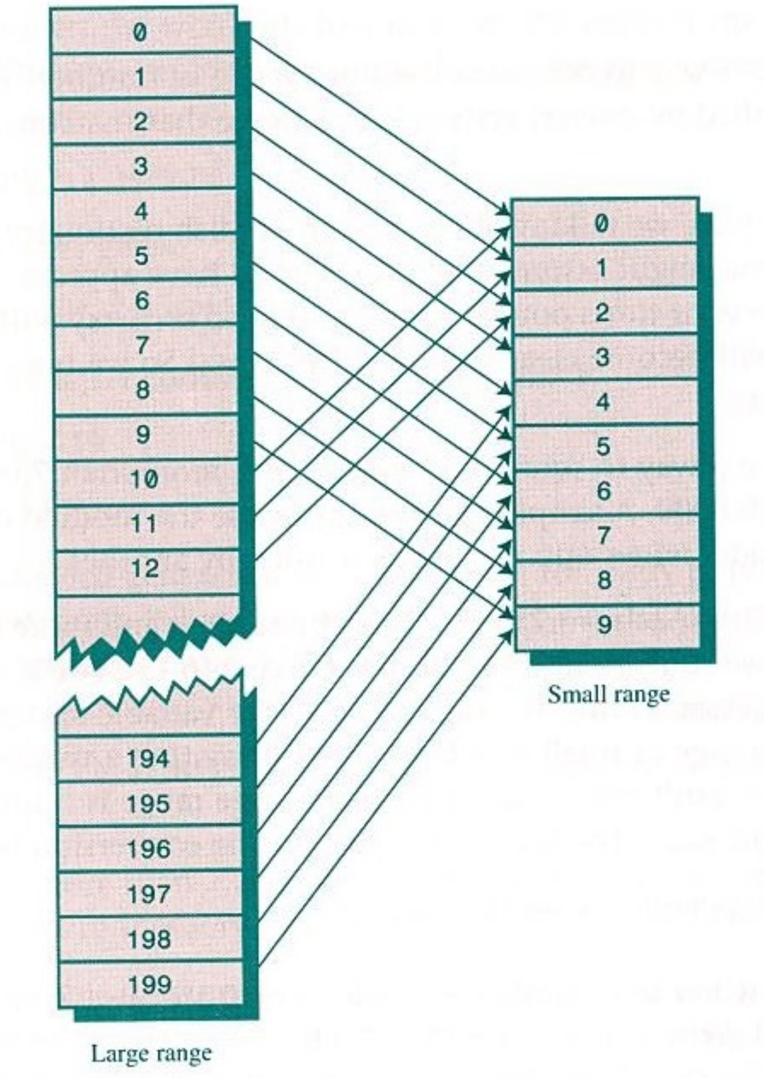
Modulo

- Modulo just divides the number by the modulus and gives us the remainder
  - 1,234,843,632 % 100,000 = 43,632
  - 255,243,764,325 % 100,000 = 64,325
  - 285 % 100,000 = 285
- No matter what number we use, the result will always be in the range of 0 to 99,999.
- This is an example of a hash function – we hash (convert) a value in a large range to a number in a smaller range.
- The smaller range corresponds to the index numbers in the array.
- An array into which data is inserted using a hash function is called a hash table.

Hashing

Use the modulo operator to squeeze a huge range of numbers into a range about twice as big as the number we want to store

- arraySize = numberWords * 2
- arrayIndex = hugeNumber % arraySize



Hash Functions

- Hash functions involve a two-step process.
- The first step is to turn the item into a unique number.
- The second step is to generate a hash index by hashing the unique number into the range of the hash table using the modulo function.
- The item is then inserted into the slot in the hash table, which has the same number as the hash index.

First step

- If the item to be put into the hash table is already a unique number, then you can skip this step.
- If not, then the features of the object must be used to create a unique number.
- Number the features and combine them in such a way that each feature makes a unique contribution to the number.

First step

- A good system for hashing strings is to raise each letter to successive powers, so each letter makes a unique contribution.
  - Cats would be $3*27^0 + 1*27^1 + 20*27^2 + 19*27^3 = 373{,}977$

- Why?
  - There are 26 letters in the alphabet.
  - The contribution of the first letter can be between 1 → 26
  - The contribution of the second letter can be between 27 → 702
  - The contribution of the third letter can be between 729 → 18,954
  - Every letter contributes different multiples which do not overlap
  - Therefore, every word will generate a unique number
  - Given the number, we can always recover the word with certainty by working backward.

First step

- If you used *a smaller power* than the size of the alphabet, then the same words could generate the same number.

- Let’s use powers of 3 as an example
  - Cats would be $3*3^0 + 1*3^1 + 20*3^2 + 19*3^3 = 699$
  - Lass would be $12*3^0 + 1*3^1 + 19*3^2 + 19*3^3 = 699$

- If you use *a bigger power* than the size of the alphabet, then that’s fine, but you’re doing more calculations than you need to
- ASCII has 127 possible values, so to generate unique numbers for ASCII strings, it is sufficient to raise to powers of 128
- It doesn’t matter whether the powers count up or down or are in a random order so long as they are all different!

Second step

- The second step is to modulo the unique number (which could be huge) by the size of the hash table.
- This gives a number which corresponds to a slot in the hash table.
- It is important to use a prime number as the size of the hash table.
  - 9,672,566,767 % 100,000 = 66,767
  - 473,944,666,767 % 100,000 = 66,767
  - 66,767 % 100,000 = 66,767
  - 7,395,838,578,258,696,795,683,856,866,767 % 100,000 = 66,767

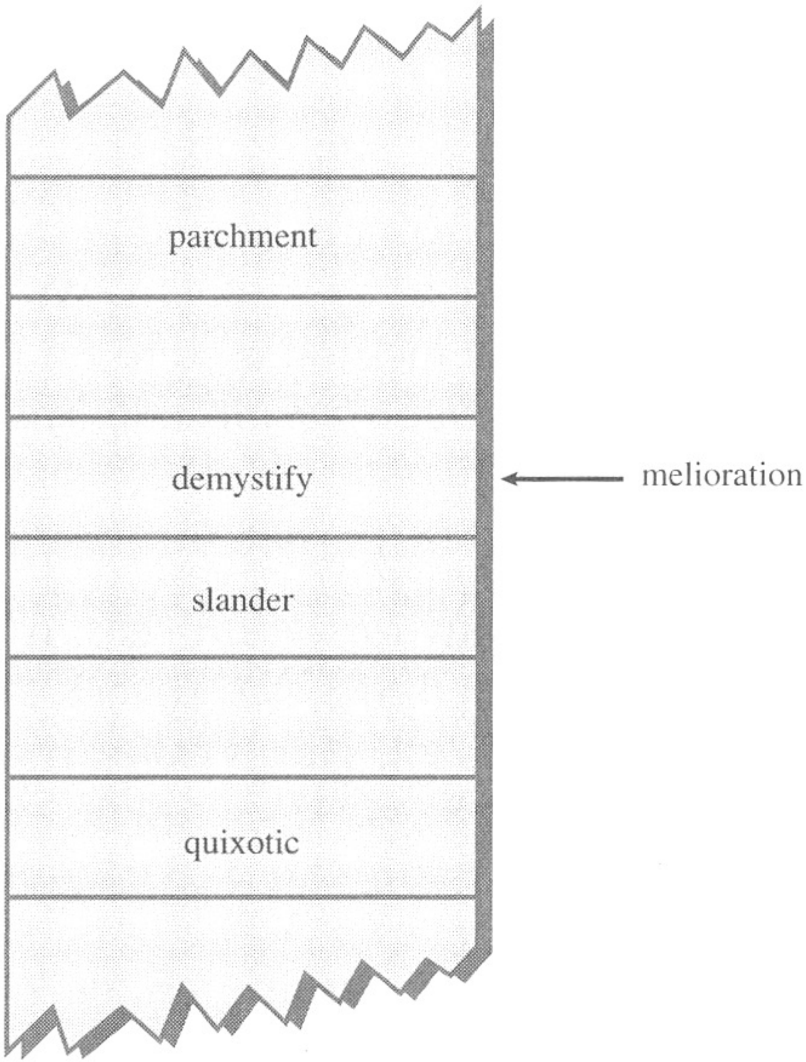
- If you don't use a prime number, then only part of the unique number will contribute to the hash index.
- Many different objects with a few features in common will end up generating the same hash index.

Prime Sized

- Prime-sized hash tables are better because a prime modulus will eliminate regular patterns in the data.
- Imagine we're hashing ten exam marks which have all been rounded up to the nearest 10%
- If we pick an array size of 20 then every mark will either be hashed into slot 10 (10%, 30%, 50%, 90%) or slot 0 (20%, 40%, 60%, 80%)
- If we use a prime number size, say 19, then all of these marks will hash into different slots.
- Any regular patterns in the data are eliminated by using a prime modulus.

Collisions

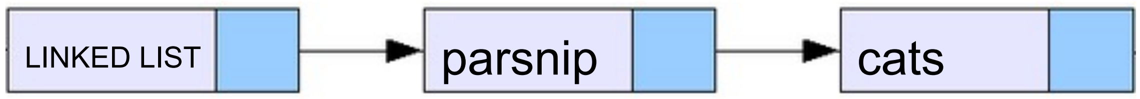
- On average, there should be one word in every second slot.
- However, there's no guarantee that two words won't hash to the same array index.
- Imagine you go to insert the word melioration, and you find that the appropriate slot already contains the word demystify.
- This is called a collision.



Open Addressing

There are two main approaches for resolving collisions.

- One approach called open addressing is to search the array in some systematic way, looking for the next empty cell
  - if cats hashes to 5,421, but this location is already occupied by parsnip, then we could try to insert cats in slot 5,422
- A second approach is called separate chaining, where linked lists are stored in each slot of the array.
  - Add cats as a new node of the linked list stored in slot 5,421



Open Addressing

There are several different methods for open addressing.

- Linear probing – move along linearly, looking for the next available space.
- Quadratic probing – makes systematic jumps to look for free spaces.
- Double hashing – hash the key again to find how big the jumps should be.

Linear Probing

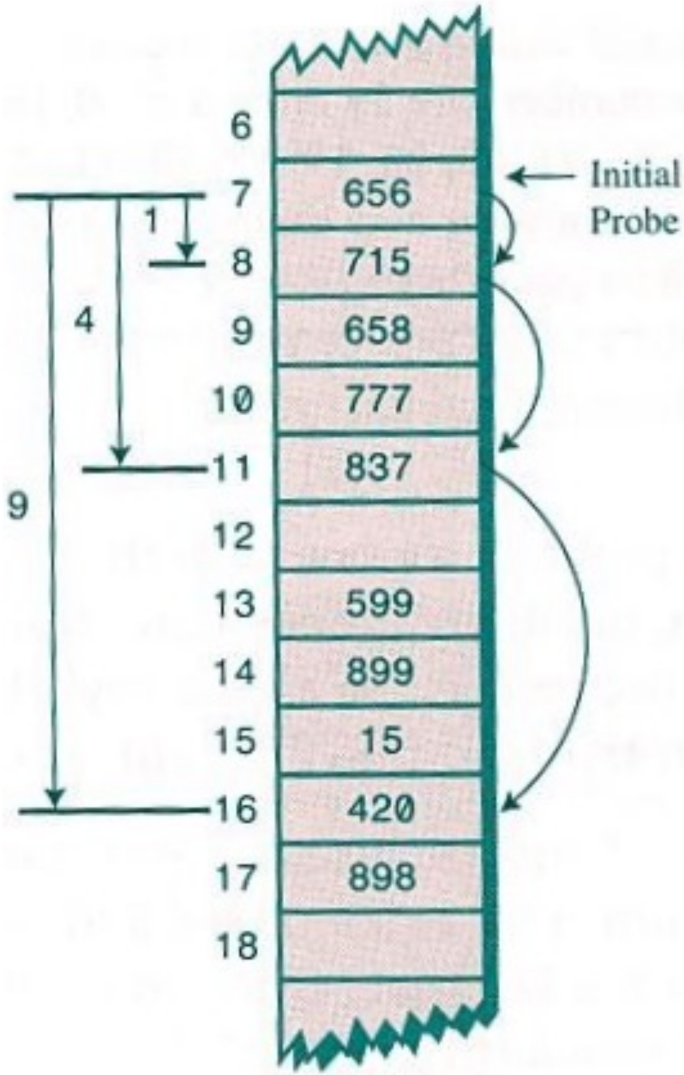
- Here we search sequentially for vacant slots.
- If slot 5,421 is occupied, then we next try 5,422, then 5,423, etc., until we find an empty cell.
- The problem is that in areas with few empty slots, even more items tend to build up in the vicinity.
- This effect is called clustering.
- If the data is not randomly distributed, then you can get major bottlenecks where there are big groupings of items with no free slots.
- Performance degrades because nearly every item is displaced, and you have to go through long linear searches to find them.

Quadratic Probing

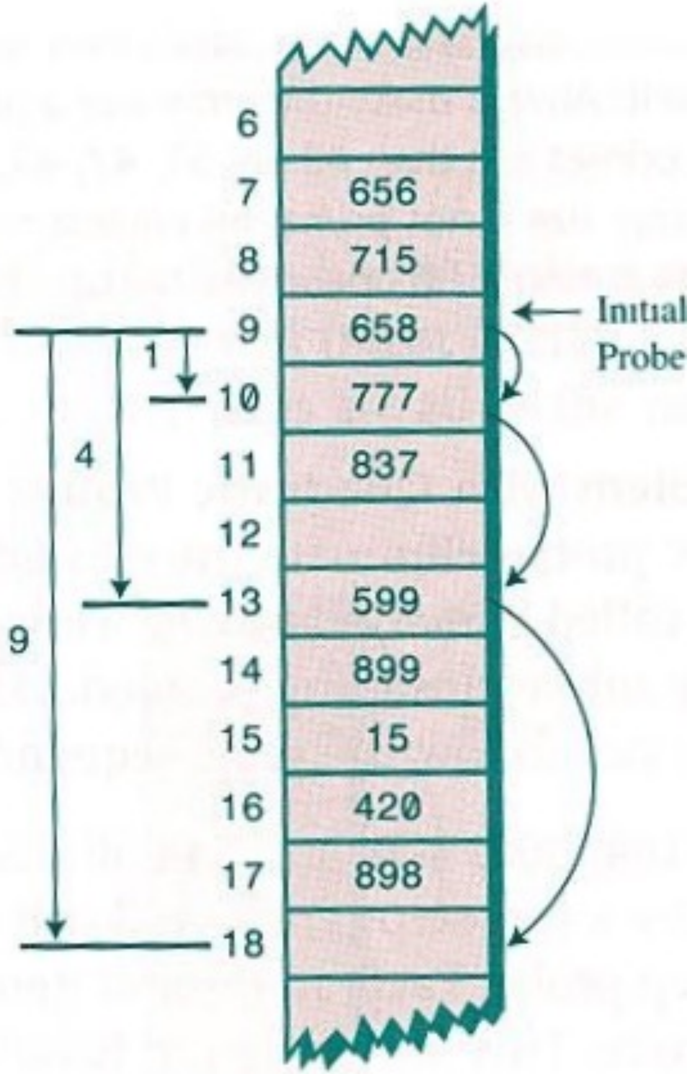
- Once a cluster forms, it gets bigger and bigger because new items have nowhere to go, so they are pushed out to the edges of the cluster.
- Quadratic probing is an attempt to keep clusters from forming.
- The idea is to probe widely separated slots rather than those adjacent to the primary hash site.
- Every time there is a collision, the algorithm jumps further and further away from the original slot.

Quadratic Probing

- In a linear probe, if the primary hash index is x, subsequent probes are x+1, x+2, x+3, etc.
- In quadratic probing, probes go to x+1, x+4, x+9, x+16, x+25 etc.
- The distance from the initial problem is the square of the step number: e.g., $x+5^2$ is step 5
- The probe becomes increasingly desperate and keeps searching further and further away with each step.
- Clusters won't form as easily because if items are already bunched up, the new item will be placed far away



a) Successful search for 420



b) Unsuccessful search for 481

Quadratic Probes

- Quadratic probes can eliminate primary clustering, but they lead to a more subtle clustering problem.
- The problem is that all the keys that hash to a particular slot will follow the same jumps in trying to find a vacant slot.
- Secondary clustering is caused when many items hash to the same slot.
- To avoid primary and secondary clustering, double hashing is often used

Double Hashing

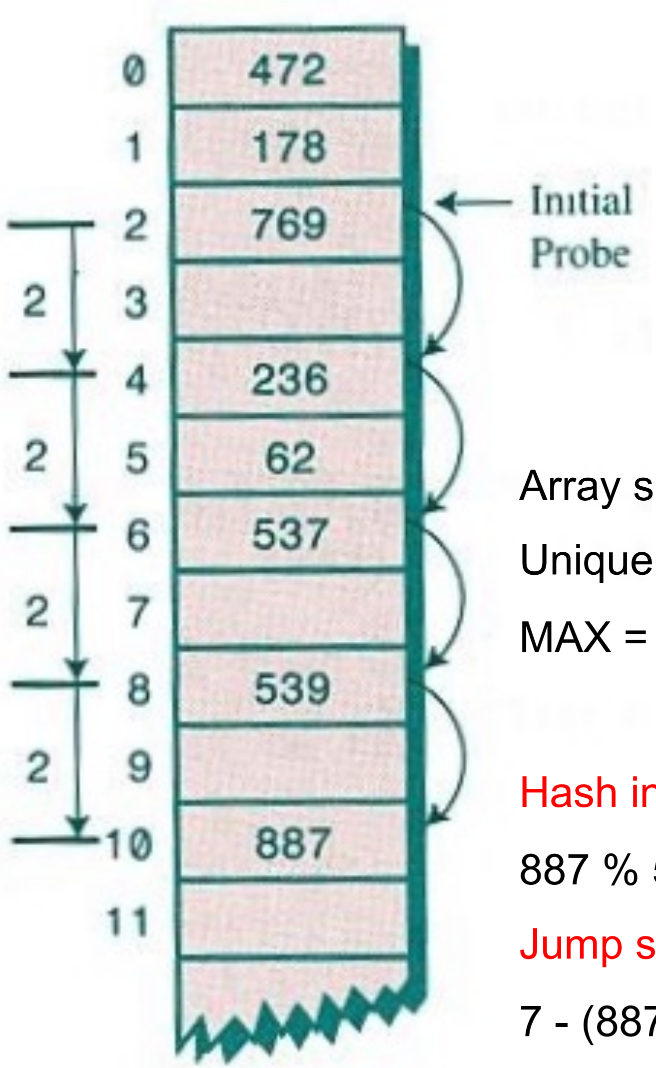
- Secondary clustering results because two items that hash to the same slot will always follow the same sequence of jumps.
  - 1, 4, 9, 16, 25...
- If a particular slot is very popular, then secondary clustering develops around those jump sizes.
- We want items that generate the same hash index to follow different jump sizes.
- The trick is to use a different hash function to generate a unique jump size for items that generated the same hash index.
  - The primary hash function generates the hash index for the item.
  - The secondary hash function generates the jump size for the item.

Double Hashing

- The secondary hash function takes in the unique number for the item and outputs a jump size in a given range.
  - The jump size is obviously smaller than the size of the hash table.

- The secondary hash function must have certain characteristics.
  - It must not be the same as the primary hash function.
  - It must never output a 0 (or else there would be no step, the algorithm would go into an infinite loop, and the program would seize up).

- The following formula outputs a value between 1 and MAX
  - jumpSize = MAX – (unique number % MAX)
  - Let MAX be 7 and unique number be 258
  - jumpSize = 7 – (258 % 7) = 1



a) Successful search for 887

Array size of 59

Unique number = 887

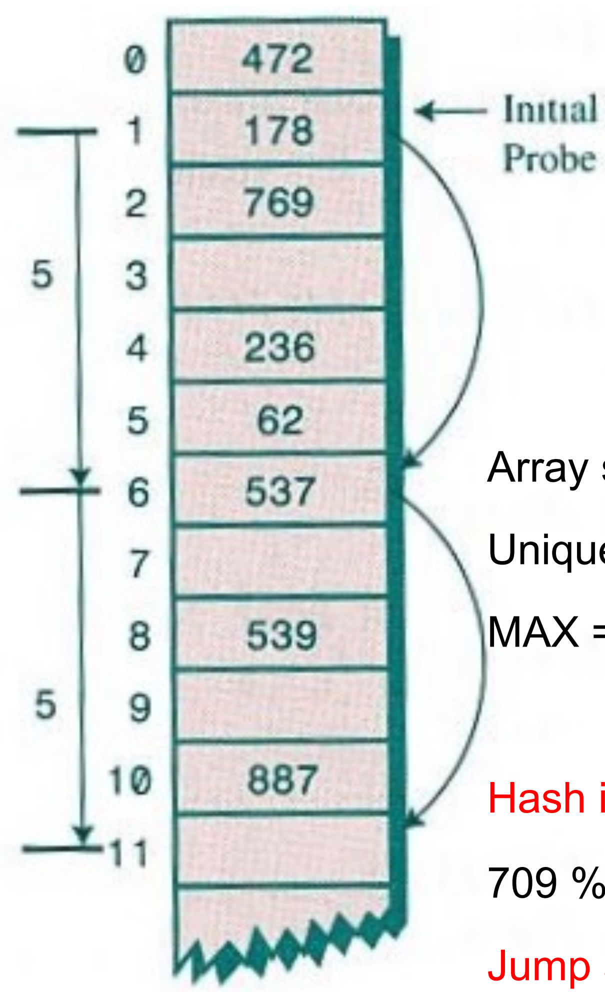
MAX = 7

Hash index =

887 % 59 = 2

Jump size =

7 - (887 % 7)= 2



b) Unsuccessful search for 709

Array size of 59

Unique number = 709

MAX = 7

Hash index =

709 % 59 = 1

Jump size =

7 - (709 % 7)= 5

Prime Sized

- Double hashing also requires the size of the hash table to be prime.
- If the array size was 15 and a particular key hashes to 0 with a jump size of 5, then the probe sequence will be.
  - 0, 5, 10, 0, 5, 10, 0 etc.

- If these cells are all full, the algorithm will loop forever.
- Prime numbers make sure every cell is considered.
- Imagine an array size of 13 with a jump size of 5.
  - 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3 etc.

- If there is any empty cell, the algorithm will find it with a prime jump size.

Open Addressing

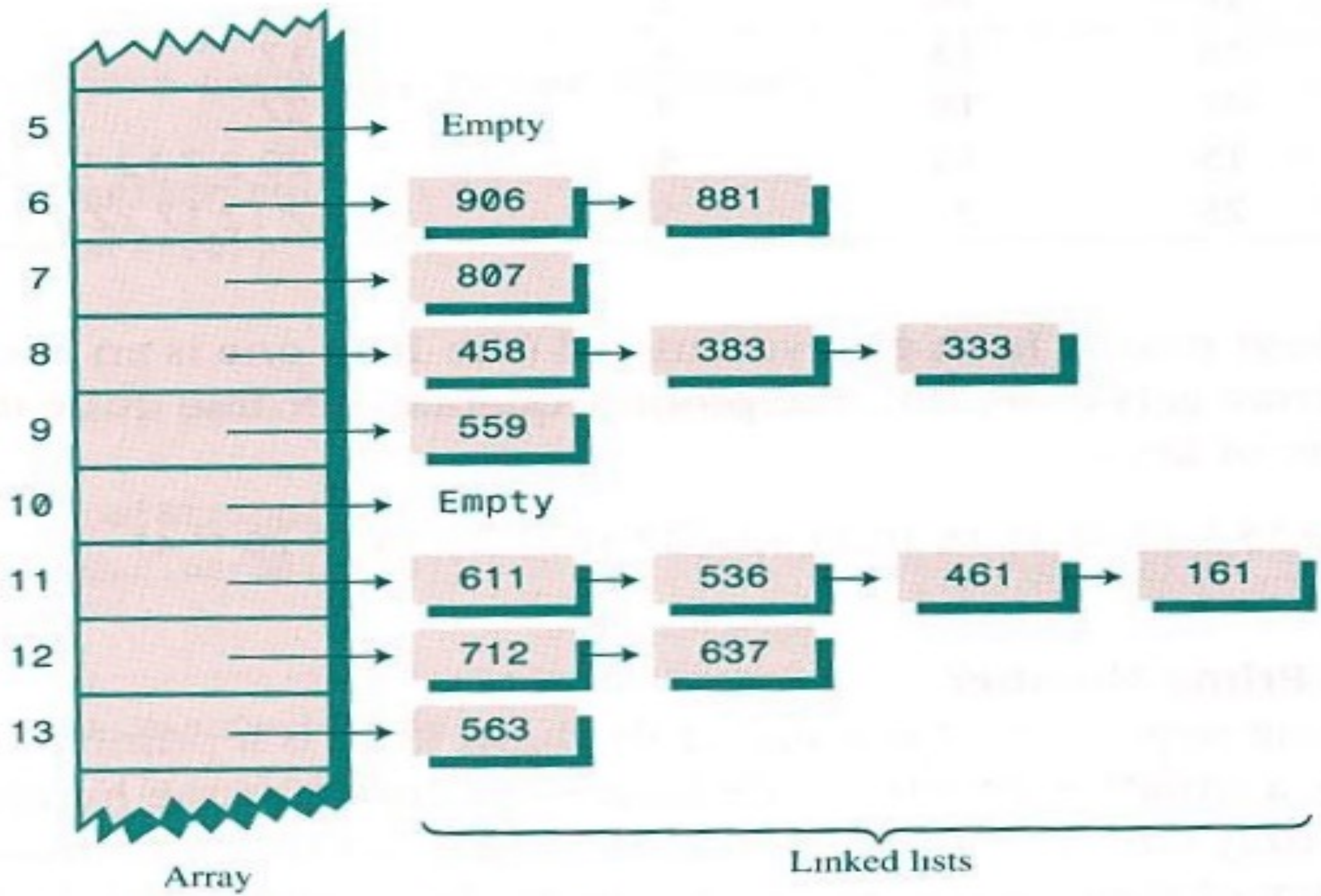
- Open addressing leads to a problem with deletion.
- A probe gives up its search once it comes across an empty slot
- But what if the item you're searching for is there, but one of the items it collided with before it has been since deleted?
- We solve this problem by filling a slot with a deleted value (e.g., 1) when an item has been deleted rather than making it empty.
- This way, the probe will know to keep going instead of stopping.
- This is called a tombstone
- When tombstones build-up you should periodically rehash your table to get rid of them.

Index	Contents
5016	parsnip
5017	-1
5018	house
5019	cats

Separate Chaining

- Rather than trying to put each item in a separate slot, another approach is to install a linked list in each slot.
- If collisions occur and several items end up in the same slot, then they are simply added to the linked list.
- When looking for an item at a particular slot, simply search through the linked list containing all items hashing to that slot.
- So long as the linked lists don't become too long, then the performance shouldn't be too badly affected.

Separate Chaining



Load Factor

- The load factor is the ratio between the size of the hash table and the number of items in it.
  - Load factor of 1 = as many items as there are slots
  - Load factor of 0 = hash table is empty
- With open addressing, the maximum load factor is obviously 1 because each slot can only hold one item.
- With separate chaining, it is normal for the load factor to be higher than 1 because each slot holds a linked list.
  - Finding the initial slot is O(1) time but searching through the linked list is O(M) where M is the average number of items in each linked list.
  - Therefore, we'd like to keep M around 1
  - With a load factor of 1, about a third of slots will be empty, about a third will hold one item, and about a third will hold two or more items.

Efficiency

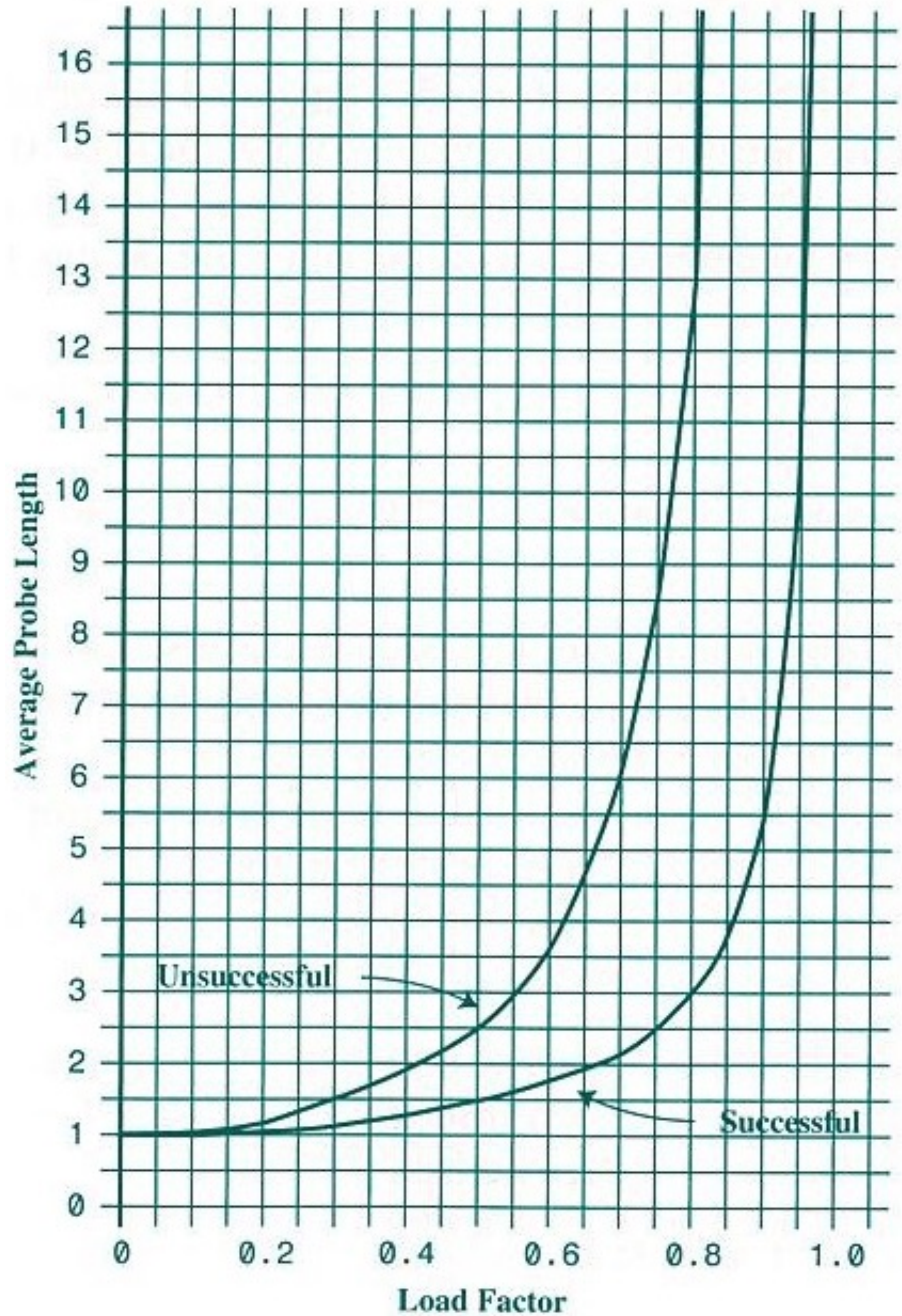
- Insertion and searching in hash tables should approach O(1) time
  - The number of items in the hash table should have no impact on the amount of time taken to find an item.
- If no collision occurs all we need to do is call the hash function and access one slot in the array.
- If collisions occur, access times become dependent on the resulting probe lengths.
  - A slot must be checked to see if it is empty or if it contains the item we're looking for.
- Search or insertion time is therefore proportional to the length of the probe.
- Average probe length is in turn dependent on the load factor.

Open Addressing

- As load factor increases, probe lengths grow longer, and efficiency drops.
- Loss of efficiency with high load factors is more serious for open addressing than for separate chaining.
- Unsuccessful searches take the longest because the probe must go all the way through the probe sequence before it is sure it can't find the item.
- When nearly every slot is full you might end up looking through half the hash table before finding an empty slot!

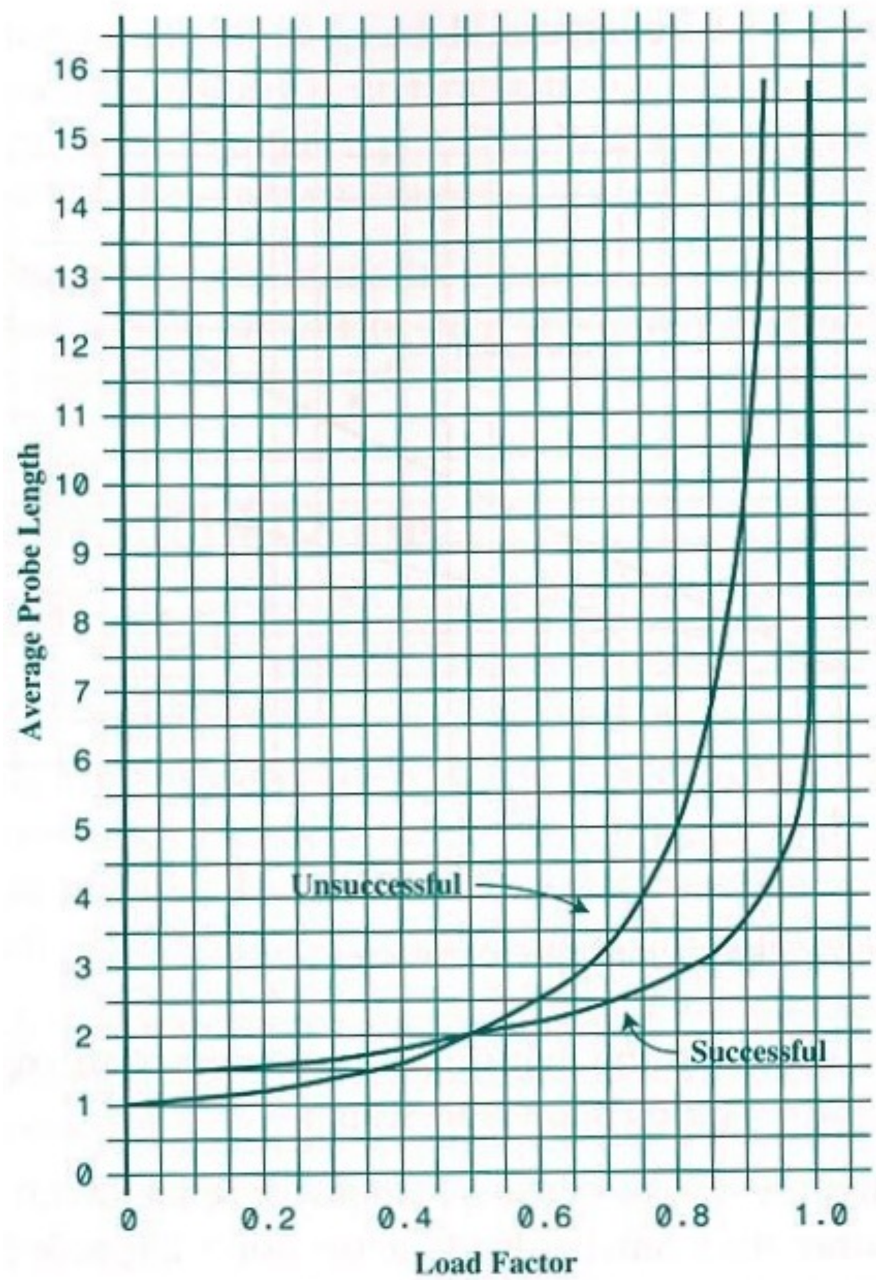
Linear Probing

Load factor needs to be kept below 2/3 for decent performance and preferably below 1/2



Quadratic probing / Double Hashing

Slightly higher load factor can be tolerated



Separate Chaining

- The most time-consuming part is comparing the search key of the item with the keys of other items in the linked list
- If N data items have been inserted into a hash table then
  - AverageLinkedListLength = N / HashTableSize
- In other words, average list length = load factor

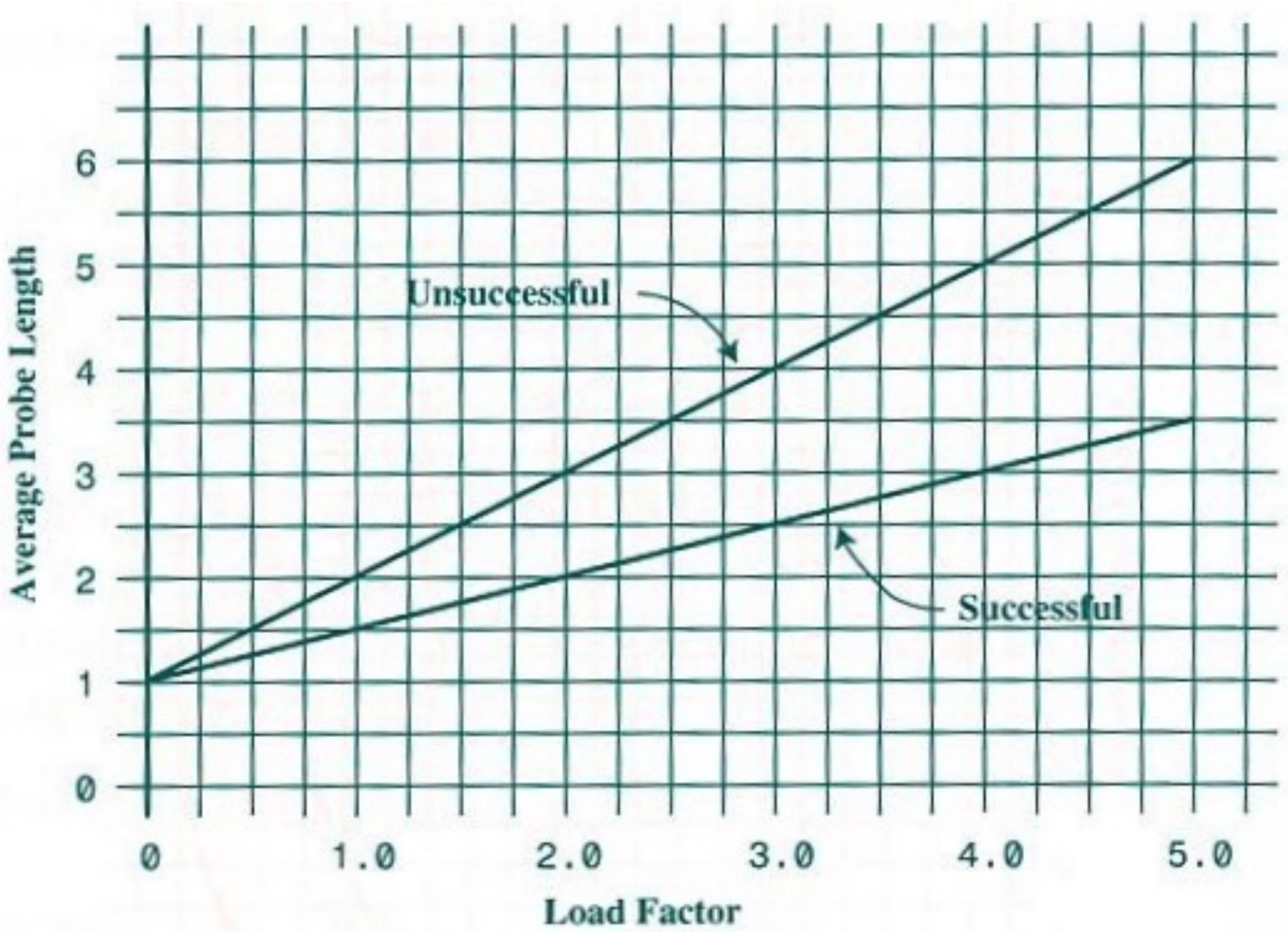
Separate Chaining

- In a successful search, on average half of the items in a linked list will have to be checked
  - 1 step needed for the hashing operation
  - Search time = 1 + loadFactor / 2
- In an unsuccessful search, all of the items must be checked
  - Search time = 1 + loadFactor
- If the linked list is ordered then on average only half of the items must be examined in an unsuccessful search
- In this case the search times for successful and unsuccessful searches converge

Insertion

- If the linked lists are not ordered then insertion is always is immediate so the insertion time is 1
- If the lists are ordered then on average half of the items in each list must be examined so the insertion time is 1 + loadFactor / 2

Separate Chaining



Summary

- In open addressing performance degrades badly as the load factor increase above ½
- In separate chaining it can rise above 1 without hurting performance too much
- As load factor increases, performance only degrades linearly using separate chaining
- Deletion is also easier with separate chaining because we don’t need to worry about probes being misled by encountering empty slots where an item has been deleted
- Using open addressing every deletion leaves behind a tombstone which adds to the load factor

Summary

- If using open addressing, double hashing is slightly better than quadratic probing
- However, if there's plenty of memory and the data won't expand after the table is created then linear probing can be handier to implement and will cause little performance penalty with a load factor under ½
- If the number of items to be inserted into the hash table is unknown then separate chaining is a safer bet
- If there are going to be many deletions then separate chaining is the way to go because an open addressing hash table will fill up with tombstones

Questions