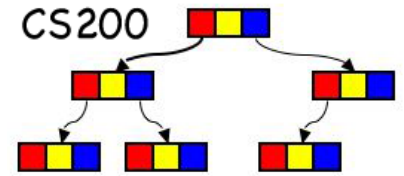
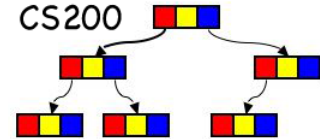


CS200: Hash Tables

Prichard Ch. 13.2

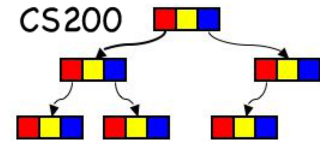
Table Implementations: average cases



	Search	Add	Remove
Sorted array-based	O(log n)	O(n)	O(n)
Unsorted array-based	O(n)	O(1)	O(n)
Balanced Search Trees	O(log n)	O(log n)	O(log n)

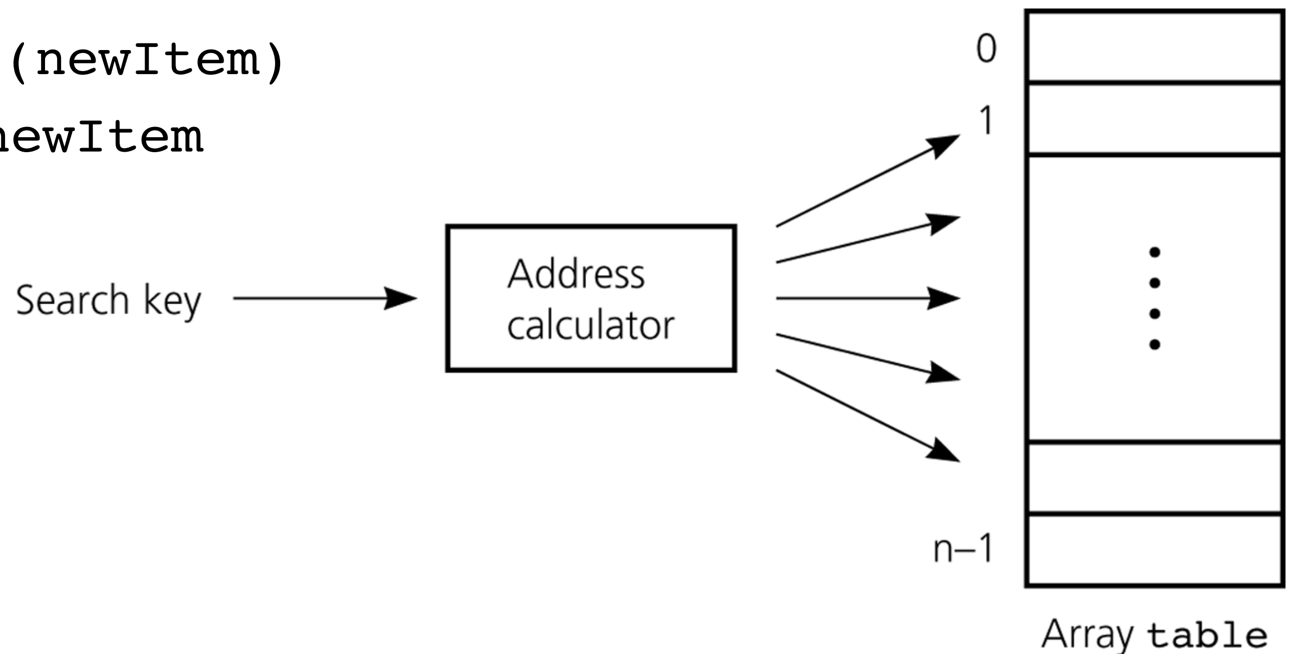
Can we build a faster data structure?

Fast Table Access

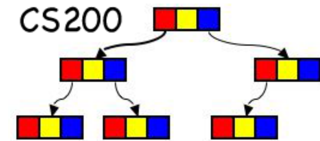


Suppose we have a magical address calculator…

```
tableInsert(in: newItem:TableItemType)
   // magiCalc uses newItem's search key to
   // compute an index
   i = magiCalc(newItem)
   table[i] = newItem
```



Hash Functions and Hash Tables



Magical address calculators exist:

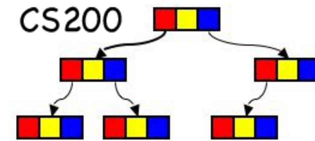
They are called hash functions

Search key → Address calculator → 0, 1, …, n–1

hash table

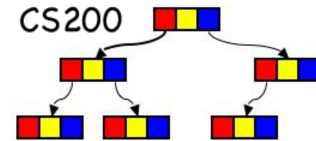
Array table

Hash Table: nearly-constant-time



- A hash table is an array in which the index of the data is determined directly from the key… which provides near constant time access!
- location of data determined from the key
  - table implemented using array(list)
  - index computed from key using a hash function or **hash code**
- close to constant time access if we have a nearly unique mapping from key to index
  - cost: extra space for unused slots

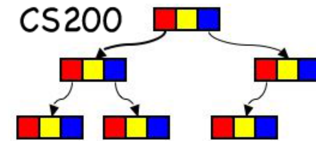
Hash Table: examples



- key is string of 3 letters
  - array of 17576 (26^3) entries, costly in space
  - hash code: letters are "radix 26" digits

    a/A -> 0, b/B -> 1, .. , z/Z -> 25,
  - Example: Joe -> 9*26*26+14*26+4
- key is student ID or social security #
  - how many likely entries?

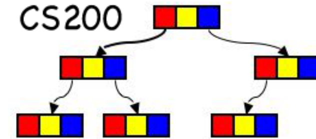
Hash Table Issues



bat
coat
dwarf
hoax
law

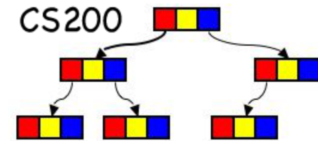
- Underlying data-structure
  - fixed length array, usually of prime length
  - each slot contains data
- Addressing
  - map key to slot index (hash code)
  - use a function of key
    - e.g., first letter of key
- What if we add 'cap'?
  - **collision** with 'coat'
  - collision occurs because hashcode does not give unique slots for each key.

Hash Function Maps Key to Index



- Desired Characteristics
  - uniform distribution, fast to compute
  - return an integer corresponding to slot index
    - within array size range
  - equivalent objects => equivalent hash codes
    - what is equivalent? Depends on the application, e.g. upper and lower case letters equivalent
      "Joe" == "joe"
- Perfect hash function: guarantees that every search key maps to unique address
  - takes enormous amount of space
  - cannot always be achieved (e.g., unbounded length strings)

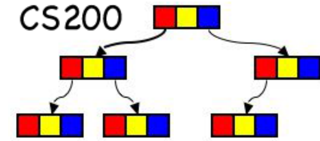
Hash Function Computation



- Functions on positive integers
  - Selecting digits (e.g., select a subset of digits)
  - Folding: add together digits or groups of digits, or pre-multiply with weights, then add
  - Often followed by modulo arithmetic:

    hashCode % table size

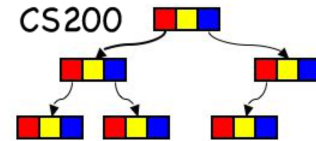
What could be the hash function if selecting digits?



- h(001364825) = 35
- h(9783667) = 37

- h(225671) = ?
  A. 39
  B. 31
  C. 61

Hash function: Folding



- Suppose the search key is a 9-digit ID.
- Sum-of-digits:

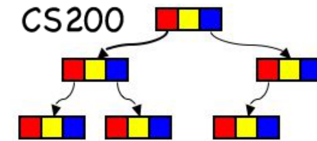
  h(001364825) = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5

  satisfies: 0 <= h(key) <= 81

- Grouping digits: 001 + 364 + 825 = 1190

  0 <= h(search key) <=3*999=2997

Hash function data distribution

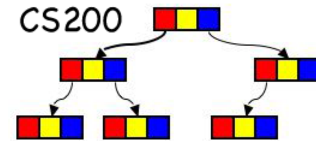


- Assume key is a String
- Pick a size; compute key to any integer using some hash code; index = hashCode(key)%size
- hashCode e.g.:

  Sum(i=0 to len-1)
  getNumericValue(string.charAt(i))*radix^i

  - similar to Java built-in hashCode() method
- This does not work well for very long strings with large common subsets (URL) or English words.

hashCode on words

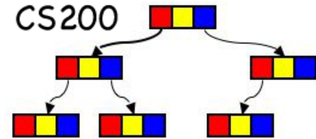


- Letter frequency is NOT UNIFORM in the English language (actually in no language)

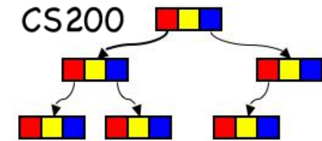
  Highest frequency for “e” : 12% followed by “t” : 9% followed by “a” : 8%

- The polynomial evaluation in hashCode followed by taking modulo hashSize gives rise **to non uniform hash** distribution.

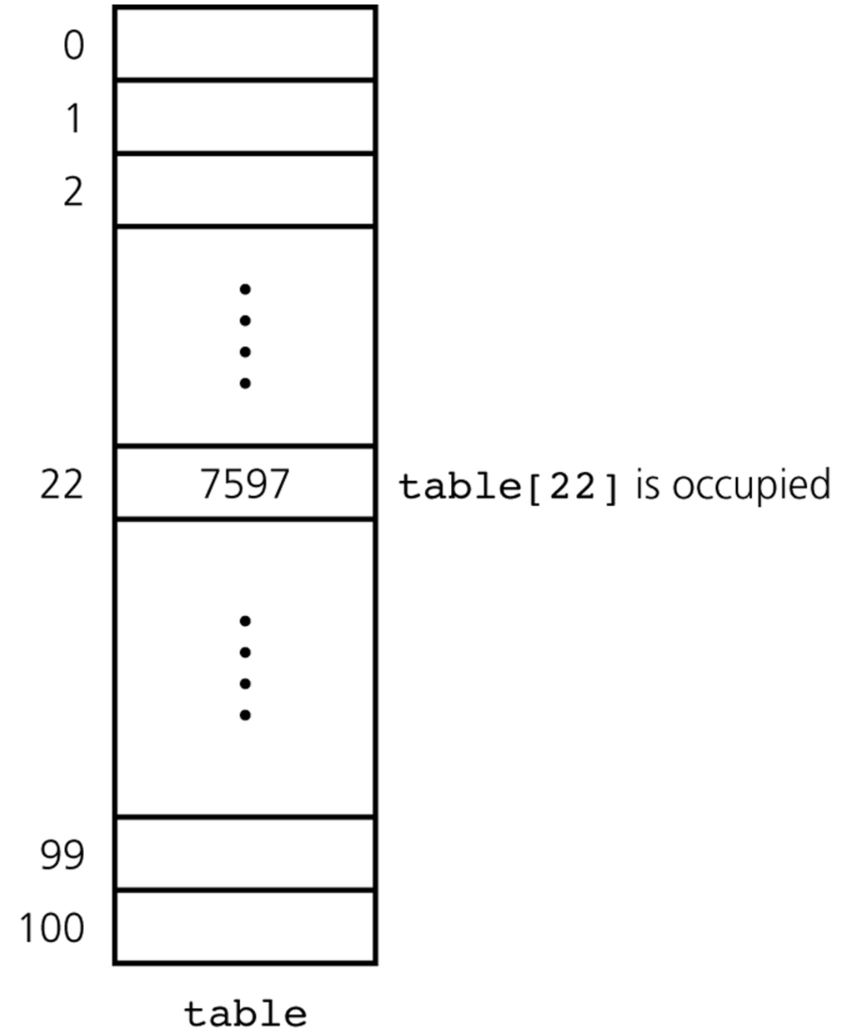
hashSize = 1000 vs 1009



Collisions



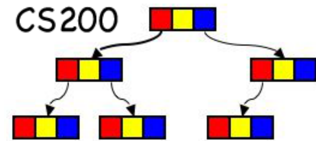
Collision: two keys map to the same index



Hash function: key%101

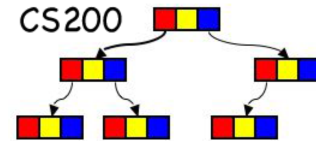
both 4567 and 7597 map to 22

The Birthday Problem



- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than ½?
- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely

The Birthday Problem



- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than ½?
- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely
- p_n – the probability that all people have different birthdays

$$p_n = 1 \frac{365}{366} \frac{364}{366} \cdots \frac{366-(n-1)}{366}$$

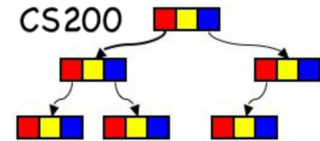
- at least two have same birthday:

$$n = 23 \rightarrow 1 - p_n \approx 0.506$$

The Birthday Problem: Probabilities

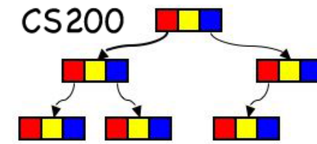
N: # of people	P(N): probability that at least two of the N people have the same birthday.
10	11.7 %
20	41.1 %
23	50.7 %
30	70.6 %
50	97. 0 %
57	99.0%
100	99.99997%
200	99.99999999999999999999999999998%
366	100%

Probability of Collision

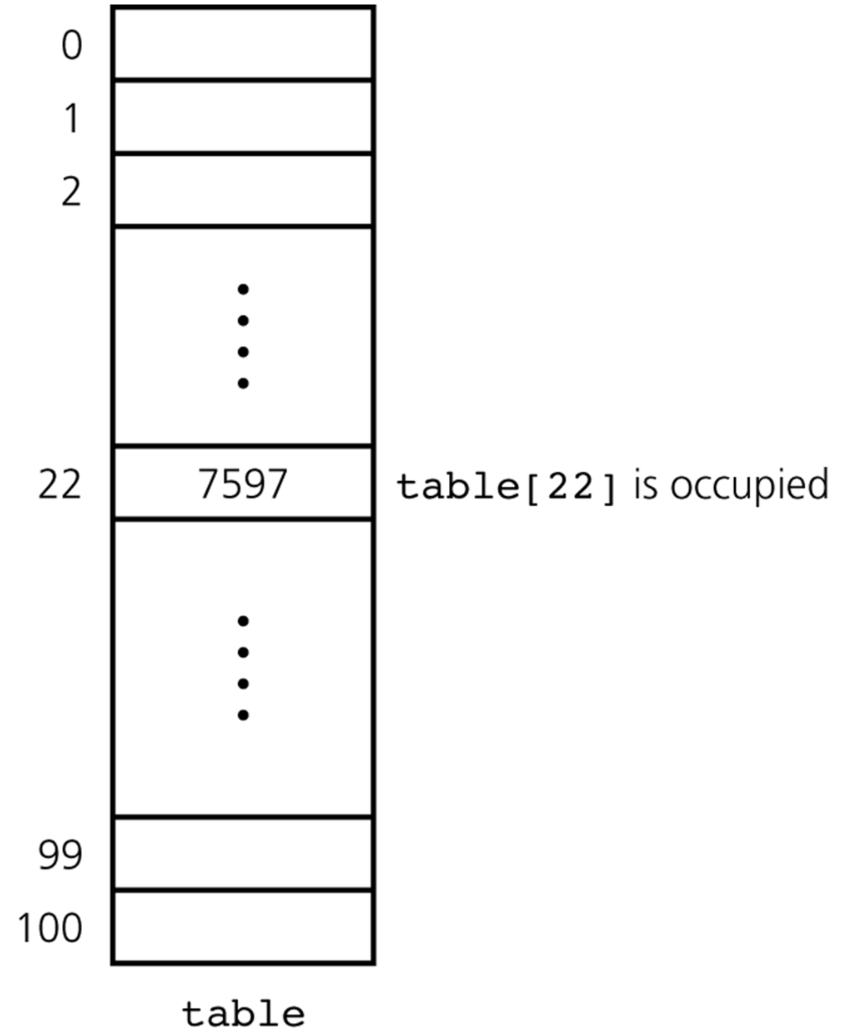


- How many items do you need to have in a hash table, so that the probability of collision is greater than ½?
- For a table of size 1,000,000 you only need 1178 items for this to happen!

Collisions



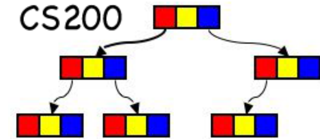
Collision: two keys map to the same index



Hash function: key%101

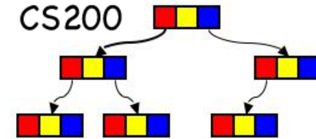
both 4567 and 7597 map to 22

Methods for Handling Collisions



- Approach 1: Open addressing
  - Probe for an empty slot in the hash table
- Approach 2: Restructuring the hash table
  - Change the structure of the array table: make each hash table slot **a collection** (e.g. ArrayList, or linked list)

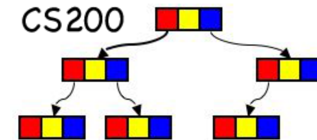
Open addressing



- When colliding with a location in the hash table that is already occupied
  - Probe for some other empty, open, location in which to place the item.
  - Probe sequence
    - The sequence of locations that you examine
    - **Linear probing** uses a constant step, and thus probes loc, (loc+step)%size, (loc+2*step)%size, etc.

In the sequel we use **step=1** for linear probing examples

Linear Probing, step = 1

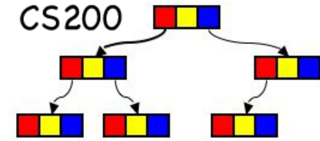


- Use first char. as hash function
  - Init: ale, bay, egg, home
- Where to search for
  - *egg* hash code 4
  - *ink* hash code 8
- Where to add
  - *gift* 6 empty
  - *age* 0 full, 1 full, 2 empty

ale
bay
age
egg
gift
home

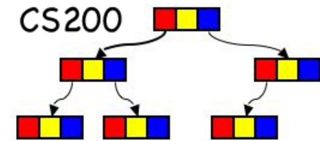
Question: During the process of linear probing, if there is an empty spot,
A. Item not found ?
or
B. There is still a chance to find the item ?

Open addressing: Linear Probing



- **Deletion:** The empty positions created along a probe sequence could cause the retrieve method to stop, incorrectly indicating failure.
- **Resolution:** Each position can be in one of three states **occupied, empty, or deleted**. Retrieve then continues probing when encountering a deleted position. Insert into empty or deleted positions.

Linear Probing (cont.)

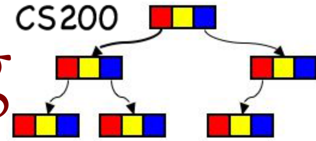


- insert
  - *bay*
  - *age*
  - *acre*
- remove
  - *bay*
  - *age*
- retrieve
  - *acre*

Question: Where does almond go now?

ale
egg
gift
home

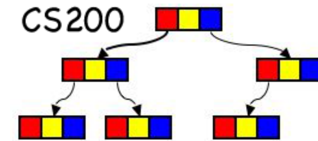
Open Addressing 1: Linear Probing



- Primary Clustering Problem
  - keys starting with 'a', 'b', 'c', 'd' **all** compete for same open slot (3)

ale
bay
age
egg
gift
home

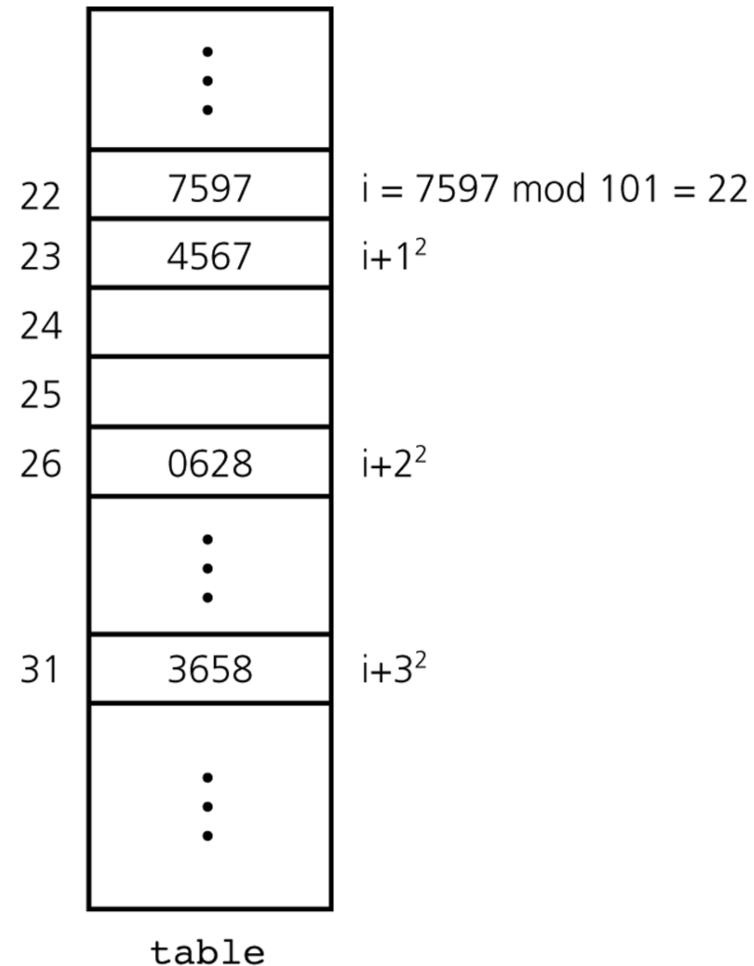
Open Addressing: Quadratic Probing



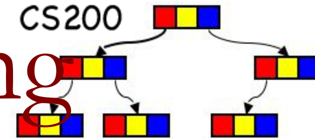
- check

  `h(key) + 1`^{2}`, h(key) + 2`^{2}`, h(key) + 3`^{2}`,…`
- Eliminates the primary clustering phenomenon
- But secondary clustering: two items that hash to the same location have the same probe sequence

  is not solved



Open Addressing: Double Hashing



Use two hash functions:

- h_1(key) – determines the position
- h_2(key) – determines the step size for probing
  - the secondary hash h_2 needs to satisfy:

    $h_2(key) \neq 0$

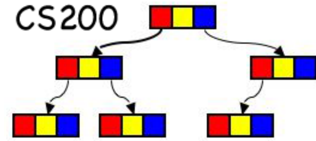
    $h_2 \neq h_1$ (bad distribution characteristics)

  So which locations are now probed?

  $h_1, h_1+h_2, h_1+2*h_2, \ldots, h_1+i*n_2, \ldots$

- Now two different keys that hash with h_1 to the same location most likely (but not for sure, see next slide) have different secondary hash h_2

Double Hashing, example



POSITION: $h_1(key) = key \% 11$

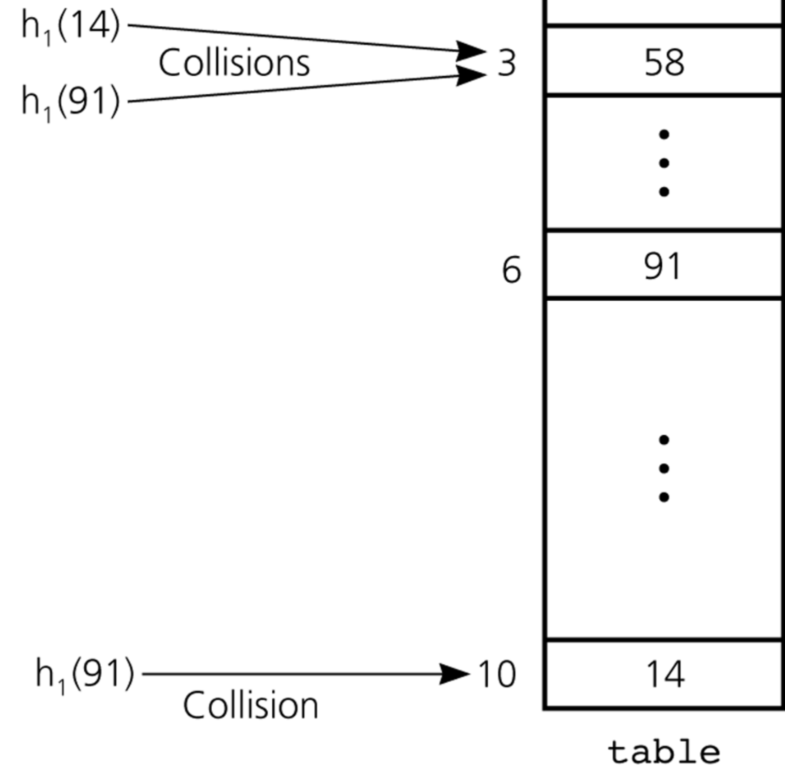
STEP: $h_2(key) = 7 - (key \% 7)$

Insert 58, 14, 91

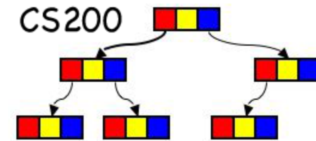
h1(58) = 3, put it there

h1(14) = 3 collision
h2(14) = 7-(14%7) = 7
put it in (3+7)%11 = 10

h1(91) = 3 collision
h2(91) = 7-(91%7) = 7
3+7 = 10   collision
put it in (10+7)%11 = 6

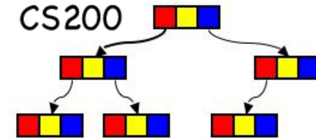


Open Addressing: Increasing the table size



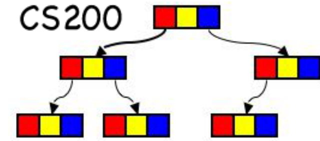
- Increasing the size of the table: as the table fills the likelihood of a collision increases.
  - Cannot simply increase the size of the table – need to run the hash function again

Restructuring the Hash Table: Hybrid Data Structures

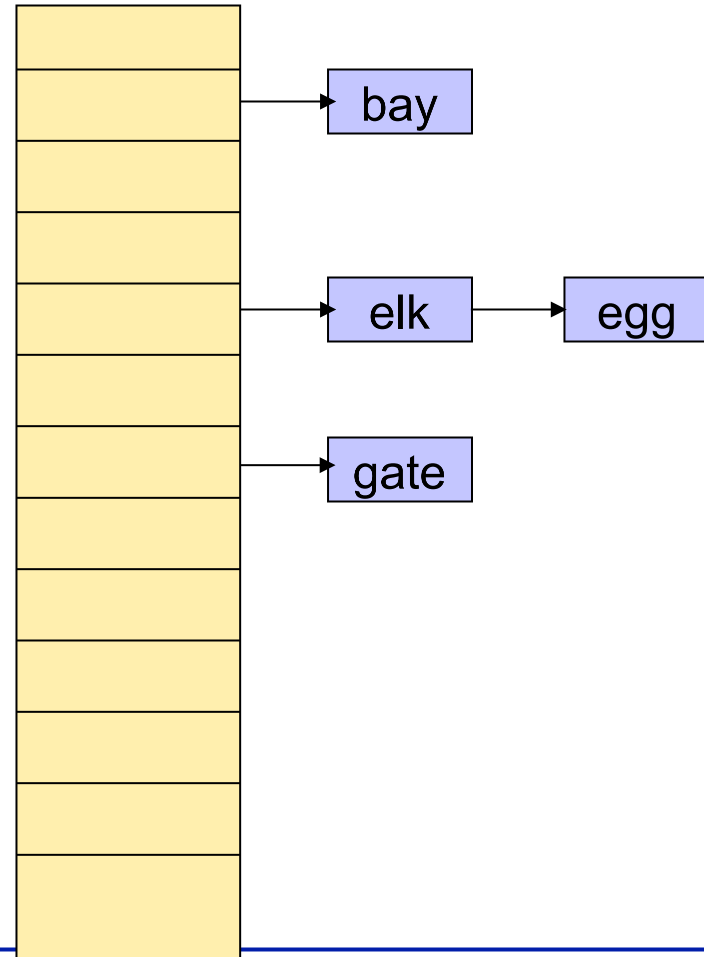


- elements in hash table become collections
  - elements hashing to same slot grouped together in a collection (or **"chain"**)
  - the chain is a separate structure
    - e.g., ArrayList or linked-list, or BST
- a good hash function keeps a near uniform distribution, and hence the collections small
- chaining does not need special case for removal as open addressing does

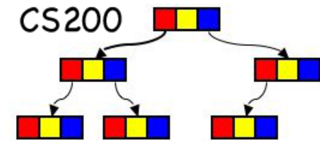
Separate Chaining Example



- Hash function
  - first char
- Locate
  - egg
  - gift
- Add
  - bee?
- Remove
  - bay?

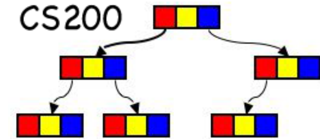


The Efficiency of Hashing



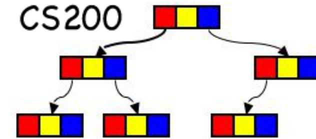
- Consider a hash table with n items
  - Load factor $\alpha = n \,/\, tableSize$
  - n: current number of items in the table
  - tableSize: maximum size of array
  - α : a measure of how full the hash table is.
    - measures difficulty of finding empty slots
- Efficiency decreases as n increases

Size of Table



- Determining the size of Hash table
  - Estimate the largest possible n
  - Select the size of the table to get the load factor small.
  - Rule of thumb: load factor should not exceed 2/3.

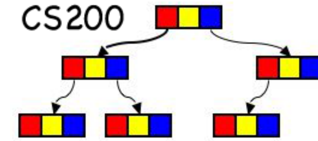
Hashing: Length of Probe Sequence



- **Average** number of comparisons that a search requires,
  - Linear Probing
    - successful $\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$
    - unsuccessful $\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$
  - Quadratic Probing and Double Hashing
    - successful $\frac{-\log_e(1-\alpha)}{\alpha}$
    - unsuccessful $\frac{1}{1-\alpha}$

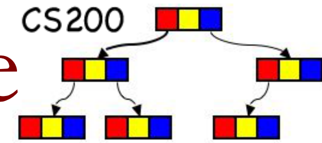
From D.E. Knuth, Searching and Sorting, Vol. 3 of The Art of Computer Programming

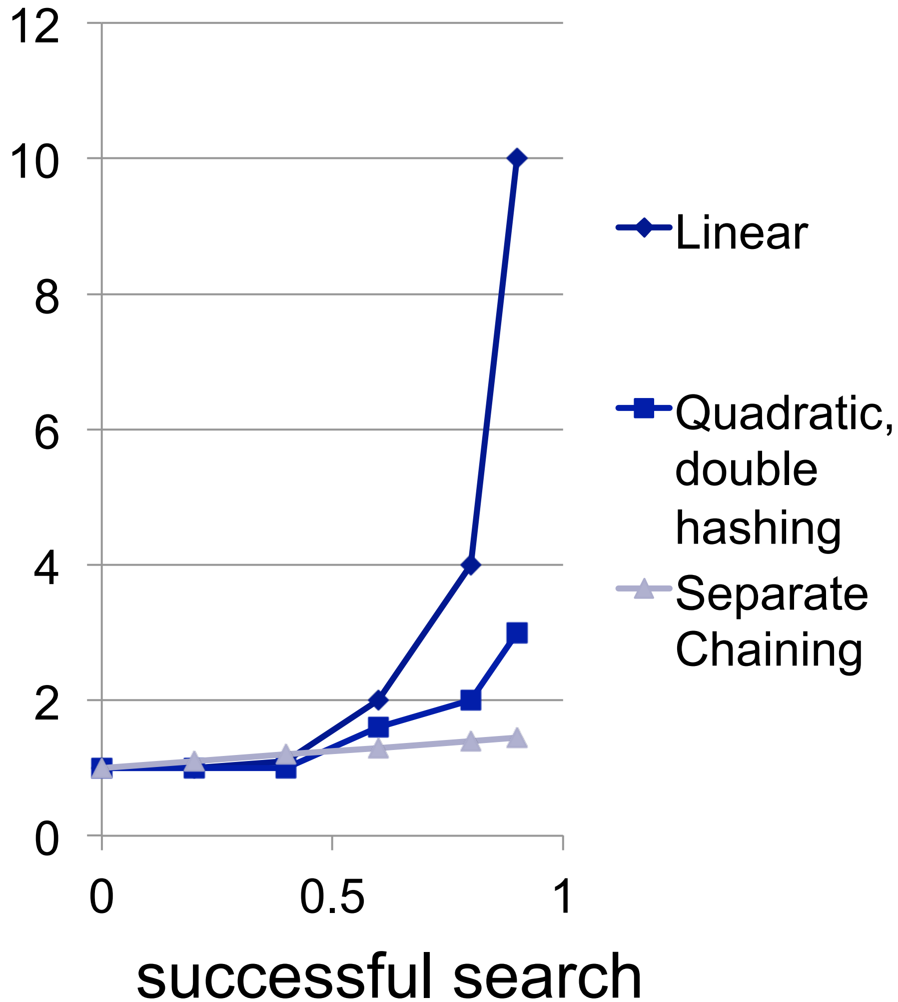
Hashing: Length of Probe Sequence

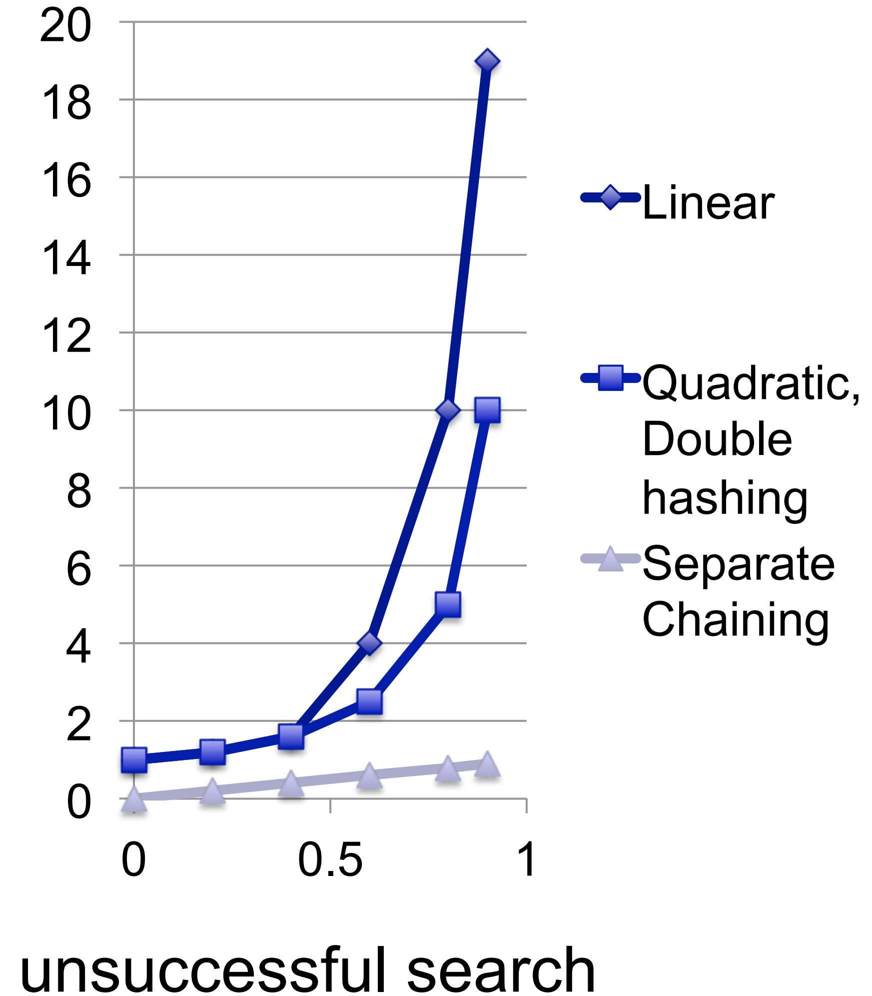


- **Average** number of comparisons that a search requires,
  - Separate chaining
    - successful: $1 + \alpha/2$
    - unsuccessful: α
  - Note that α can be > 1 for chaining
- From this we can conclude (see Prichard):
  - Linear probing is worst
  - Quadratic probing and double hashing are better
  - Separate chaining is best
  - BUT it is all average case!

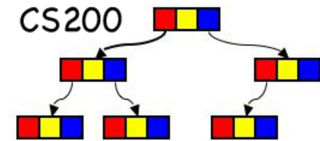
Average length of probe sequence





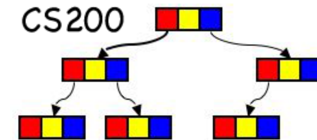


Traversal of Hash Tables



- Hash tables good for random access
- If you need to traverse your tables by the sorted order of keys – hash tables may not be the appropriate data structure.

Hash Tables in Java



From the JAVA API: “A map is an object that maps keys to values… The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.” Both provide methods to create and maintain a hash table data structure with key lookup.
Load factor (default 75%) specifies when the hash table capacity is automatically increased.

```
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>

public Hashtable(int initialCapacity, float loadFactor)
   public Hashtable(int initialCapacity)  //default loadFactor: 0.75
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>
   public HashMap(int initialCapacity, float loadFactor)
   public HashMap(int initialCapacity)  //default loadFactor: 0.75
```