CMSC 341
Hashing

Based on slides from previous iterations of this course

Hashing

Searching

- Consider the problem of searching an array for a given value
  - If the array is not sorted, the search requires O(n) time
    - If the value isn't there, we need to search all n elements
    - If the value is there, we search n/2 elements on average
  - If the array is sorted, we can do a binary search
    - A binary search requires O(log n) time
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - How about an O(1), that is, constant time search?
    - We can do it *if* the array is organized in a particular way

The Basic Problem

- We have lots of data to store.
- We desire efficient – O(1) – performance for insertion, deletion and searching.
- Too much (wasted) memory is required if we use an array indexed by the data’s key.
- The solution is a “hash table”.

Introduction

- If we wanted to find one person out of the possible 322,071,600 in the US, how would we do it?

- With no additional information, we may have to search through all 322M people!

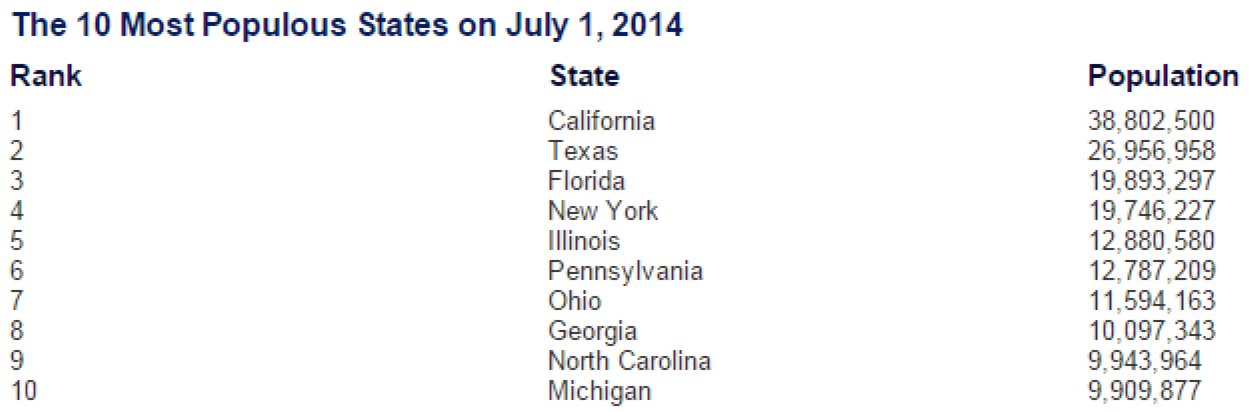
From: http://factfinder.census.gov/faces/tableservices/jsf/pages/productview.xhtml?src=bkmk

Introduction

- However, if we were to organize each of the people by the 50 states, we may greatly increase the speed to find them.

Introduction

- Now, we know that the populations of the states are not evenly distributed



The 10 Most Populous States on July 1, 2014

Rank	State	Population
1	California	38,802,500
2	Texas	26,956,958
3	Florida	19,893,297
4	New York	19,746,227
5	Illinois	12,880,580
6	Pennsylvania	12,787,209
7	Ohio	11,594,163
8	Georgia	10,097,343
9	North Carolina	9,943,964
10	Michigan	9,909,877

- When our n = 322,071,600 we would expect 322,071,600 / 50 = **6,441,432 in each state**

Introduction

- But, the important concept here is – as long as we know which state to look in, we can greatly reduce the data set to look in!

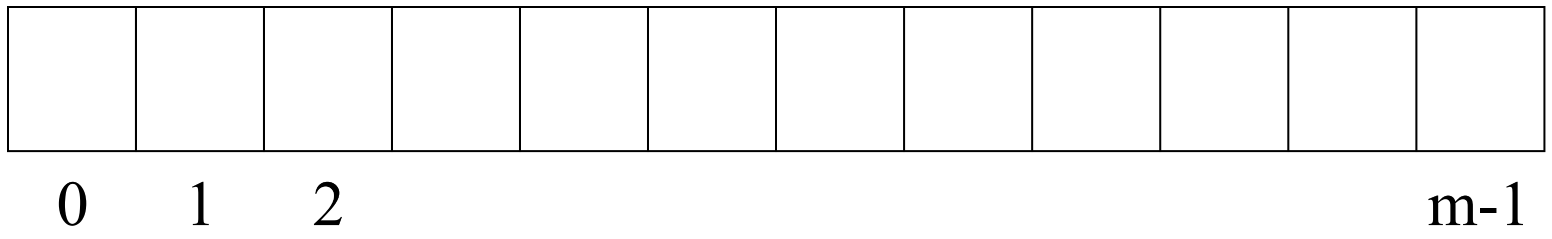
- Hashes take advantage of organizing the data into buckets (or slots) to help make the functions more efficient

Hash Tables

- A ***hash table*** is a data structure for storing key-value pairs
- Unlike a basic array, which uses index numbers for accessing elements, a hash table uses keys to look up table entries

- Two major components to a hash:
  - Bucket array (or slot)
  - Hash function

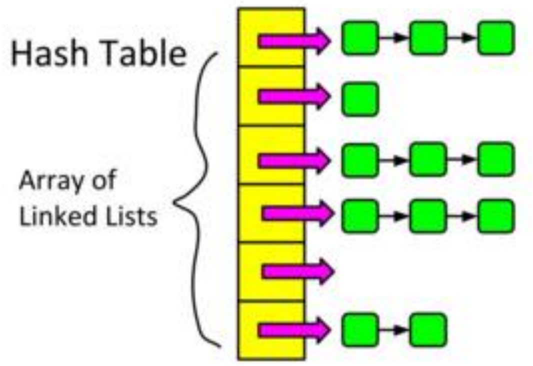
Hash Table



- Basic Idea
  - The hash table is a bucket array of size ‘m’
  - The storage index for an item determined by a *hash function* h(k): U → {0, 1, …, m-1}
- Desired Properties of h(k)
  - easy to compute
  - uniform distribution of keys over {0, 1, …, m-1}

Bucket Array

- A ***bucket array*** for a hash table is an array **A** of size **N**, where each cell of **A** is thought of as a "bucket"
- Obviously, we can also implement this using an array of linked lists as well



Hash Functions

- What if we had a "magic function" that, given a value to search for, would tell us exactly where in the array to look?
  - If it's in that location, it's in the array
  - If it's not in that location, it's not in the array
- This function would have no other purpose
  - If we look at the function's inputs and outputs, they probably won't "make sense"
  - This function is called a hash function because it "makes hash" of its inputs

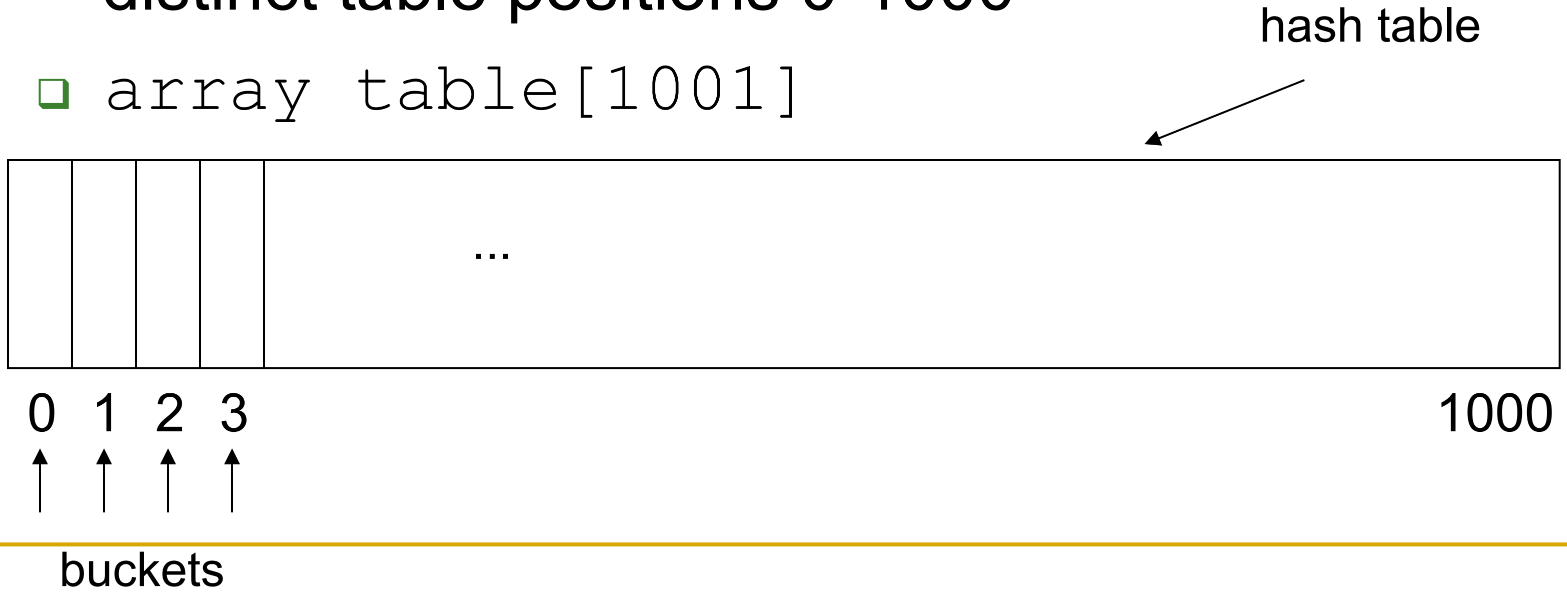
Hash Function

- The ***hash function*** is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought

- A hash function takes in an item key as its parameter and returns an index location for that particular item

Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 100 students
  - Hash function: h(k) = k-951000 maps ID into distinct table positions 0-1000
  - `array table[1001]`



Analysis (Ideal Case)

- O(b) time to initialize hash table (b number of positions or buckets in hash table)
- O(1) time to perform *insert, remove, search*

Ideal Case is Unrealistic

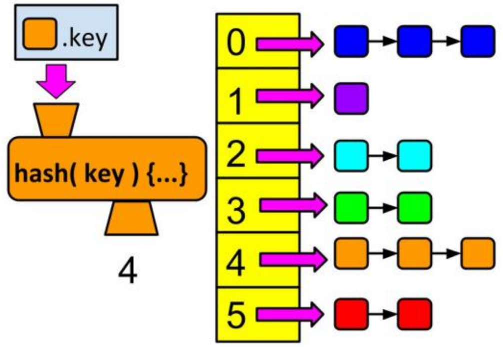
- Many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!

Example:

- Suppose key can take on values from 0 .. 65,535
- Expect ≈ 1,000 records at any given time
- Impractical to use hash table with 65,536 slots!

Hash Function

- Generally uses modulo arithmetic
- A key value is divided by the table length to generate an index number in the table
- This index number refers to a location, or bucket, in the hash table



From: http://pumpkinprogrammer.com/2014/06/21/c-tutorial-intro-to-hash-tables/

Hash Tables vs. Other Data Structures

- Implementations of the dictionary operations Insert(), Delete() and Search()/Find()
  - Arrays:
    - Can accomplish in O(1) time
    - But are not space efficient (assumes we leave empty space for keys not currently in dictionary)
  - Binary search trees
    - Can accomplish in O(log n) time
    - Are space efficient.
  - Hash Tables:
    - A generalization of an array that under reasonable assumptions is O(1) for Insert/Delete/Search of a key

Different Implementations

- As with almost all of the data structures that we have discussed so far, there are a variety of ways to implement them

- Let’s start by looking at some “real world” examples to help illustrate the data structure as well as some possible issues

Hash Function: Example 1
ASCII Values

Hash Function – Example 1

- Suppose our hash function
  - Takes in a string as its parameter
  - It adds up the ASCII values of all of the characters in that string to get an integer
  - The performs modulo math with the table size

```
int hash( string key )
{
    int value = 0;
    for ( int i = 0; i < key.length(); i++ )
        value += key[i];
    return value % tableLength;
}
```

From: http://pumpkinprogrammer.com/2014/06/21/c-tutorial-intro-to-hash-tables/

Hash Function – Example 1

- If the key is “pumpkin,” then the sum of the ASCII values would be 772
- For a table of size 13, the modulus of this number gives us an index of 5
- So the item with the key “pumpkin,” would go into bucket # 5 in the hash table

Char	Dec
p	112
u	117
m	109
p	112
k	107
i	105
n	110

From: http://pumpkinprogrammer.com/2014/06/21/c-tutorial-intro-to-hash-tables/

Bad Hash Function

- This isn’t a very good hash function – why?

- Words tend to use certain letters more often than other
- Words tend to be rather short
  - With a large table size, we wouldn’t necessarily use the full length

- But it illustrates one way that we could implement a hash function

Hash Function – Example 2

Hash Function – Example 2

- Suppose our hash function gave us the following values:

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Function – Example 2

- Suppose our hash function gave us the following values:

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Now, this is an IDEAL situation because the results put each in their own spot!

Hash Function – Example 2

- What happens if we add honeydew?

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
hash("honeydew") = 6
```

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

What happens now?

Hash Example – Using SSN

- A social security application keeping track of people where the primary search key is a person's social security number (SSN)
- You can use an array to hold references to all the person objects
  - Use an array with range 0 - 999,999,999
  - Using the SSN as a key, you have O(1) access to any person object

Hash Example – Using SSN

- Unfortunately, the number of active keys (Social Security Numbers) is much less than the array size (1 billion entries)
  - Est. US population, November 2015: 322,071,600
  - Over 60% of the array would be unused

Example 3 – Hash Functions

Hash Function – Example 3

We have a small group of people who wish to join a club (say about 40 folks). Then, if each of these people have an ID# associated with them (from 1 to 40) we could store their information in an array and access it using the ID# as the array index.

Hash Function – Example 3

Now, we have 7 of these clubs, with consecutive ID#s going up to 280. Now what?

- We COULD create a 280 element array for each club and use 40 elements of the array. (wasteful?)
- We COULD create a 40 element array and calculate the index of each person using a mapping. (index = ID# % 40).

Hash Function – Example 3

Now, imagine that we are hosting a club on campus open to all students. We could use the PC ID# (8 digits long). How big should our array be?

THINGS TO CONSIDER:

- How many students do we expect to join?
- How can we create a key based on this number?

Hash Function – Example 3

- If we expect no more than 100 club members, we can use the last two digits of the PC ID# as our index (aka KEY). Do we see any problems with this?

- How do we get this number?
    - Take the remainder
        - (PC ID# % 100)

Hash Functions

- Taking the remainder is called the **Division-remainder technique (MOD)** and is an example of a **uniform hash function**

- A uniform hash function is designed to distribute the keys roughly evenly into the available positions within the array (or hash table)

Collisions

Collisions

- If no two values are able to map into the same position in the hash table, we have what is known as an "*ideal hashing*".
- Usually, ideal hashing is not possible (or at least not guaranteed). Some data is bound to hash to the same table element, in which case, we have a ***collision***
- How do we solve this problem?

Collisions

- Ideally the hash function should map each possible key to a different slot index; but this goal is rarely achievable in practice.
- Most hash table designs assume that ***hash collisions*** — pairs of different keys with the same hash values — are normal occurrences, and accommodate them in some way.

Collisions

- We can think of each table location as a “bucket” that contains several slots.

- Each slot is filled with one piece of data.
- This approach involves “chaining” the data.

- Thankfully, we can create an “array of arrays” or an “array of linked lists” as a way to help deal with collisions.

Collisions

- This is a common approach when the hash table is used as disk storage.
- For each element of the table, a linked list is maintained to hold data that map to the same location.
- Do we want to sort items upon entering them into the list?
  - Unsorted: easier to enter
  - Sorted: easier to retrieve

Collisions

- If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:

  $h(k_1) = \beta = h(k_2)$: k_1 and k_2 have collision at slot β

- Popular hash functions: hashing by division

  $h(k) = k\%D$, where D number of buckets in hash table

Collision

- Example: hash table with 11 buckets

$$h(k) = k\%11$$

80 →

40 →

65 →

58 →

Division Method

- The hash function:

  h(k) = k mod m where m is the table size.

- m must be chosen to spread keys evenly
  - Poor choice: m = a power of 10
  - Poor choice: m = 2^b, b> 1
- A good choice of m is a prime number.

Multiplication Method

- The hash function: $h(k) = \lfloor m(kA \mod 1) \rfloor$ where $0 < A < 1$
- A very good choice of A is the **inverse** of the "golden ratio. $A = (\text{sqrt}(5) - 1)/2$
- An advantage of the multiplication method is that the value of m is not critical.
- Given two positive numbers x and y, the ratio x/y is the "golden ratio" if $\phi = x/y = (x+y)/x$
  - The golden ratio:

    $x^2 - xy - y^2 = 0 \quad \Rightarrow \quad \phi^2 - \phi - 1 = 0$

    $\phi = (1 + \text{sqrt}(5))/2 \quad = \quad 1.618033989\ldots$

    $\sim= Fib_i/Fib_{i-1}$

Multiplication Method (cont.)

- Because of the relationship of the golden ratio to Fibonacci numbers, this particular value of A in the multiplication method is called “Fibonacci hashing.”
- Some values of

$$h(k) = \lfloor m(k\,\phi^{-1} - \lfloor k\,\phi^{-1} \rfloor) \rfloor$$

$= 0$ for $k = 0$

$= 0.618m$ for $k = 1$ ($\phi^{-1} = 1/ 1.618\ldots = 0.618\ldots$)

$= 0.236m$ for $k = 2$

$= 0.854m$ for $k = 3$

$= 0.472m$ for $k = 4$

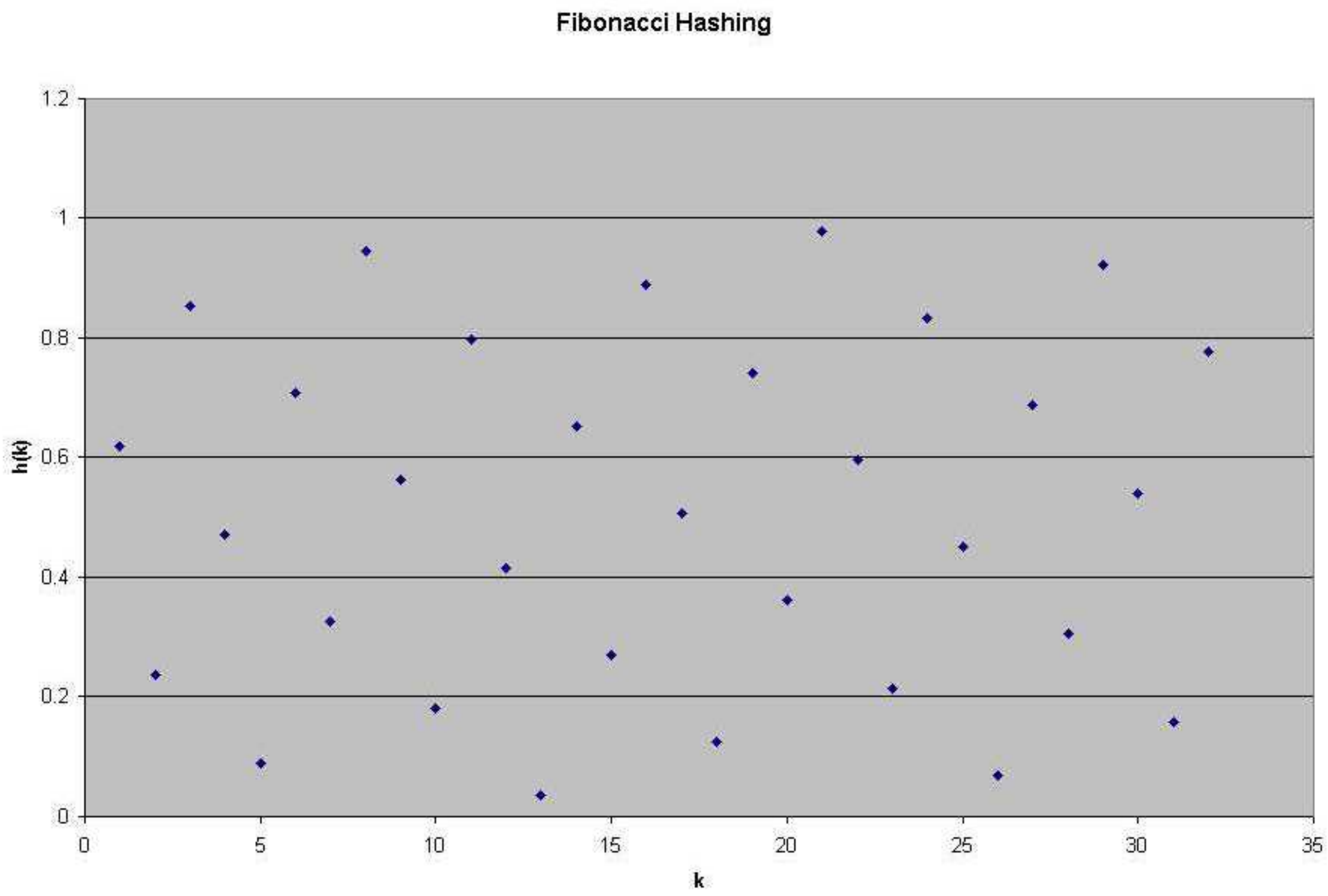
$= 0.090m$ for $k = 5$

$= 0.708m$ for $k = 6$

$= 0.326m$ for $k = 7$

$= \ldots$

$= 0.777m$ for $k = 32$


Fibonacci Hashing
h(k)
k
0
0.2
0.4
0.6
0.8
1
1.2
0
5
10
15
20
25
30
35

Finding the Optimal Hash Function

- When a collision occurs, there are a variety of ways that we can address the collision
- In a perfect world, we could come up with a magic function that helps us deal with collisions

Finding the Optimal Hash Function

- How can we come up with this magic function?
- In general, we cannot--there is no such magic function ☹
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function
- What is the next best thing?
  - A perfect hash would tell us exactly where to look
  - In general, the best we can do is a function that tells us where to *start* looking!

Collision Resolution Policies

- Two classes:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing

- Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)

Hash Function Methods

Hash Functions

- As we have mentioned, there are a variety of ways to implement hash functions. We will discuss:

1. Chaining – This is where we put additional information into a bucket
2. Linear Probing
3. Quadratic Probing

Linear Probing

- Have you ever been to a theatre or sports event where the tickets were numbered?
- Has someone ever sat in your seat?
- How did you resolve this problem?

Linear Probing

Linear Probing involves seeing an item in the hashed location and then moving by 1 through the array (circling to the beginning if necessary) until an open location is found.

Linear Probing

- Let’s say that we have 1000 numbered tickets to an event, but only sell 400. If we move the event to a smaller venue, we must also renumber the tickets. The hash function would work like this:
    - (ticket number) % 400.
    - How many folks can get the same hashed number? (3 - for example, tickets 42, 442, and 842)

Linear Probing

- The idea is that even though these number hash to the same location, they need to be given a slot based on their hash number index. Using linear probing, the entries are placed into the *next available* position.

Linear Probing

- Associated with closed hashing is a *rehash strategy*:

  "If we try to place *x* in bucket *h(x)* and find it occupied, find alternative location $h_1(x)$, $h_2(x)$, etc. Try each in order, if none empty table is full,"
- *h(x)* is called *home bucket*
- Simplest rehash strategy is called *linear hashing*

$$h_i(x) = (h(x) + i) \% D$$

- In general, our collision resolution strategy is to generate a sequence of hash table slots (probe sequence) that can hold the record; test each slot until find empty one (probing)

Linear Probing

- Consider the data with keys: 24, 42, 34,62,73 into a table of size 10. These entries can be placed into the table at the following locations:

Linear Probing

- 24 % 10 = 4. Position is free. 24 placed into element 4
- 42 % 10 = 2. Position is free. 42 placed into element 2
- 34 % 10 = 4. Position is occupied. Try next place in the table (5). 34 placed into position 5.
- 62 % 10 = 2. Position is occupied. Try next place in the table (3). 62 placed into position 3.
- 73 % 10 = 3. Position is occupied. Try next place in the table (4). Same problem. Try (5). Then (6). 73 is placed into position 6.

Linear Probing

- How would it look if the numbers were:
  - 28, 19, 59, 68, 89??

Finding and Deleting

- Finding?
  - Given a key, look it up with the hashing function and then if it's not there, keep looking one bucket down until it is or is not found. Search until you find an empty slot or search the whole array.
- Deleting?
  - We must be more careful. Having found the element, we can't just remove it. Why?
    - Removing an element from the hash table would cause the table to need "reordering" so that we wouldn't lose other data (What if we needed element 73 but element 62 had been removed? Would we ever find it?)
  - Use **lazy deletion** (Mark an element as deleted instead of erasing it)

Primary Clustering

- Sometimes, data will cluster – this is caused when many elements hash to the same (or similar) location and linear probing has been used often.

- We can help with this problem by choosing our divisor carefully in our hash function and by carefully choosing our table size.

Designing a Good Hash Function

- If the divisor is even and there are more even than odd key values, the hash function will produce an excess of even values. This is also true if there are an excessive amount of odd values.
- However, if the divisor is odd, then either kind of excess of key values would still give a balanced distribution of odd/even results.
- Thus, the divisor should be **odd**. But, this is not enough.

Designing a Good Hash Function

- Thus, the divisor should be **odd**. But, this is not enough.
- If the divisor itself is divisible by a small odd number (like 3, 5, or 7) the results are unbalanced again
- Ideally, it should be a **prime number** (close to m).
- If no such prime number works for our table size (the divisor, remember?), we should use an odd number with no small factors.

Problems of Linear Probing

- The majority of the problems are caused by clustering. These problems can be helped by using Quadratic probing or better double hashing instead.
- We’ll cover this next time!

Announcements

- Homework 5
  - Due Thursday, November 9th at 8:59:59 PM
- Exam 2 is Tuesday, November 21st

- Next Time:
  - More on Hashing