CMSC 341
Lecture 16/17 Hashing, Parts 1 & 2

Prof. John Park

Based on slides from previous iterations of this course

Today's Topics

- Overview
  - Uses and motivations of hash tables
  - Major concerns with hash tables
- Properties
  - Hash function
  - Hash table size
  - Load factor
- Operations
  - Collision handling
  - Resizing/Expanding
  - Deletion

Introduction

If we wanted to find one person out of the possible 326,071,600 in the US, how would we do it?

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

As of November 2015

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

Motivation

- We want a data structure that supports fast:
  - Insertion
  - Deletion
  - Searching
- (We don't care about sorting)

- We could use direct indexing in an array, but that is not space efficient
- The solution is a ***hash table***

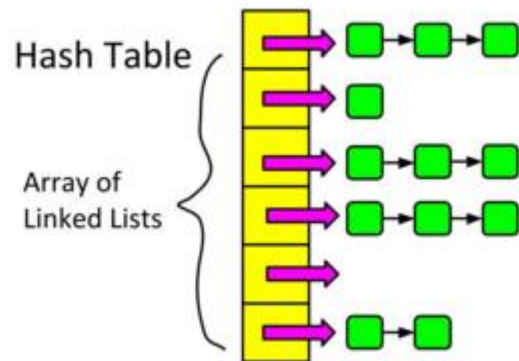
Hash Tables

- 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:
  - Bucket array (or slot)
  - Hash function

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

Bucket Array

- A ***bucket array*** is an array of size N where each cell can be thought of as a “bucket” that holds a collection of key/value pairs
- Obviously, we can also implement this using an array of linked lists as well



Hash Function

- If the ***keys*** are unique integers that fit in the range [0, N-1] then the bucket array is all we need – no hash function at all!
  - However, this is rarely (*i.e.*, never) the case

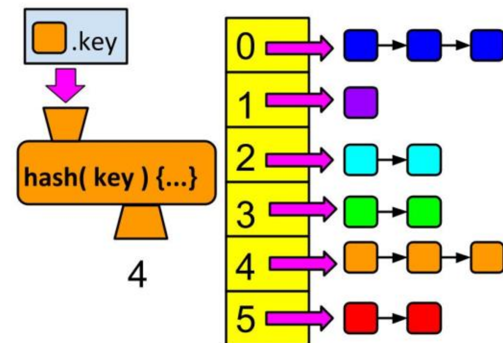
Hash Function

- A ***hash function*** is needed to take our initial keys and map them into the range [0, N-1]

- Two parts to a hash function:
  - Hash code
    - Converts key into an integer
  - Compression function
    - Converts integer to index in the correct range
  - (Often combined into one function)

Hash Function

- In particular, 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/

Uses of Hash Functions

- Convert non-integer keys (like strings) into an integer index for easy storage
- Compress sparsely-populated indexes into a more space-efficient format
- For fast access
  - Possibly as fast as O(1)
  - As long as sorting is not a concern

Hash Functions

- Suppose we were to come up with 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 tables vs. Other Data Structures

- We want to implement the dictionary operations Insert(), Delete() and Search()/Find() efficiently.
- 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 some reasonable assumptions is O(1) for Insert/Delete/Search of a key

Major Concerns

- How big to make the bucket array?
  - Want to minimize space needed
  - Want to minimize number of collisions
- How to choose hash function?
  - Want it to be efficient
  - Want it to produce evenly distributed indexes
- How to handle collisions?
  - Want to minimize time spent searching

Hash Table Properties

Hash Function

- The hash function maps the given keys to integer values in the range of the table size
  - These integer values are then used to index into specific locations in the table

- A good hash function should:
  - Be relatively easy/fast to compute
  - Create a uniform distribution
    - (Very important!)

Hash Functions – Trivial

- Some “obvious” hash functions:
  - With SSN as a key, use the last 4 as the hash
  - Convert a string key to ASCII and sum values
  - Use first three letters of a string key as the hash
- These functions perform very poorly at creating a uniform distribution
  - Leads to lots of collisions
  - Which is something we want to avoid

Hash Function–Trivial ASCII Example

- Suppose the hash function takes in a string as its parameter, and then it adds up the ASCII values of all of the characters in that string to get an integer.

```
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.
- After that, the hash function takes the modulus of this number by the table length to get an index number.

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

Hash Function – Example 1

- If the table length is 13, then 772 modulo 13 is 5.
- So the item with the key "pumpkin," would go into bucket # 5 in the hash table.
- This isn't a great algorithm because words tend to use certain letters more often than others and tend to be rather short. The algorithm will also map anagrams (same letters, different order) to the same index; it just illustrates one way that we *could* implement the hash function

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

Hash Function – Example 2

- 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
- Unfortunately, the number of active keys (Social Security Numbers) is much less than the array size (1 billion entries)
  - Over 60% of the array would be unused
- Maybe use last 4 digits?

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).

Hash Function – Integers

- Here is a decent hash for integer keys

```
((a * key + b) % P) % N)
a, b: positive integers
N    : number of buckets
P    : large prime, P >> N
```

- Having a prime number somewhere in the hash function is important
  - So values aren't easily divisible by some number

Polynomial Hash Codes

- $X_0 a^{k-1} + x_1 a^{k-2} + \ldots + x_{k-1}$
- Actually, easy to compute: just multiply partial sum by *a*, then add next *x*

Hash Functions – Strings

- Here is a decent hash for string keys

```
int hashVal = 0;
for(char in string):
    hashVal = (37 * hashVal + ASCII_of_char
                % 16908799 );
hashVal % = tableSize;
```

- Prime number (16908799) is very large so **`hashVal`** doesn't go over size for integers

Designing Hash Functions

- Hash functions can perform differently on different types of input
  - Should always test a hash function on sample input to evaluate performance

- Probably not a good idea to design your own hash function when you need one
  - There are good hash functions available, that were created by more experienced programmers and have been extensively tested

Hash Table Size

- Important to keep in mind two things when choosing a hash table size

- Interaction with hash function
  - Either table size needs to be prime
  - Or hash function needs to contain a prime
  - (Preferably both)
- Load factor
  - How full the table will be, and the rate of collisions

Load Factor

- ***Load factor*** refers to the percentage of buckets in the array containing entries
  - General rule is below 75% - 80%
  - Balance between minimizing the space needed for storage and the number of collisions

- For implementations with multiple entries per bucket, want to consider list size as well
- If actual load factor is much higher/lower than ideal, we *might* consider resizing hash table

Collisions

- If no two values map into the same position in the hash table, we have what is known as an "*perfect hashing*".

Collisions

- Usually, perfect 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***.
- 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.
- How do we solve this problem?

Collisions

- ***Collisions*** are when two keys map to the same index in the hash table
  - Affected by function, table size, and load factor
- Collisions are unavoidable in practice
- Collision-resolution strategy greatly affects effectiveness and performance of hash table
  - Many different strategies are available

Perfect Hashing

- If all of the keys that will be used are known ahead of time, and there are no more keys than can fit the hash table, a perfect hash function can be used to create a perfect hash table, in which there will be no collisions. If minimal perfect hashing is used, every location in the hash table can be used as well.
- Perfect hashing allows for constant time lookups in the worst case. This is in contrast to most chaining and open addressing methods, where the time for lookup is low on average, but may be arbitrarily large.

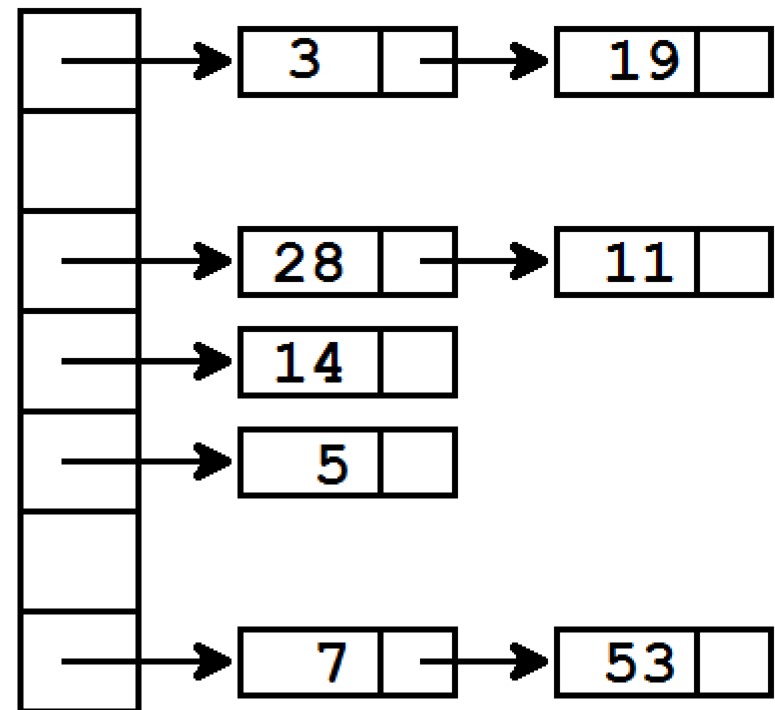
Handling Collisions

Methods of Handling Collisions

- Chaining
  - Lists (linked list, array, etc.)
  - Data structures (BST)
    - Only worth it if minimizing delay is super important
- Open addressing (probing)
  - (Entries stored directly in the bucket array)
  - Linear probing
  - Quadratic probing
  - Double hashing

Chaining

- ***Chaining*** "accepts" the collisions, and allows storage of multiple entries in one index
- The bucket array contains pointers to a data structure that can hold multiple entries (array, list, BST, etc.)



Chaining Example

- Exercise:
  - For a table of size 7, insert the following keys (where the hash function is just **`key % 7`**)
  - 1, 4, 7, 8, 9, 10, 14, 15, 17, 20, 21, 24, 27, 29

index	0	1	2	3	4	5	6
	7	1	9	10	4		20
	14	8		17			27
	21	15		24			
		29					

Chaining Performance

- Insert
  - For linked lists is O(1)
    (also amortized O(1) for vectors!)
  - For BSTs is O(log n)
- Delete and Find
  - **Worst** case for linked lists: O(n)
    - All of the entries are in one index's list
    - (This means the hash function is pretty terrible)
  - Average case for linked lists:
    O(1) when load factor is less than 100%

Probing

- Other option is ***open addressing***, or "***probing***"
  - Each index holds only one entry
- If an index already holds an entry, the question becomes – what index do we try next?
  - Random would be great – but isn't repeatable
- Three common choices
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

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*** handles collisions by finding the next available index in the bucket array
  - If it reaches the end of the bucket array, it wraps back around to the first index
- Each table cell inspected is one "probe"
- Linear probing is normally sequential, but can be implemented to probe with larger "jumps"

Linear Probing Example

- Exercise:
  - For a table of size 13, insert the following keys (where the hash function is just **`key % 13`**)
  - 1, 14, 3, 15, 2, 9, 22, 4, 7
  - What do you notice?

index	0	1	2	3	4	5	6	7	8	9	10	11	12
		1	14	3	15	2	4	7		9	22		
pushed off by:			1		2	3	2				1		

Clustering

- ***Clustering*** is when indexes in the hash table become filled in long unbroken stretches
- Most commonly occurs with linear probing
  - Especially sequential probing
- Severely degrades performance of all the operations of the hash table
  - Drops from ideal O(1) to close to O(n)
- 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

- 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**
  - If no such prime number works as our table size, we should at least use an odd number with no small factors.

Advantages

- Fast – average constant time (O(1)) for finding information – esp apparent when the table is large.
- If the key/value pairs are known before programming (disallowing insertions/deletions of new data into the table), the programmer can reduce average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures. (Sometimes this allows for "perfect hashing")

Linear Probing Performance

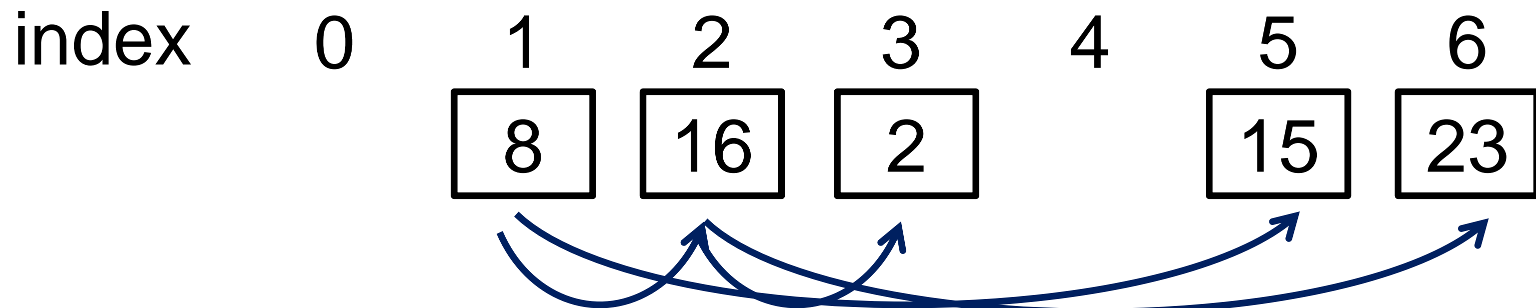
- Insert and Find
  - Best case is O(1)
  - Worst case can become O(n)—starting at hash index, must examine every sequential slot until we find, or hit our first empty bucket
- Delete is complicated
  - We can't just delete the entry! (Why not?)
    - The empty space will confuse future probing
  - Use **lazy deletion**
    - We'll discuss the details of deleting later

Quadratic Probing

- ***Quadratic probing*** is similar to linear probing
- Rather than checking in sequence, “jump” further away with each consecutive probe
  - Helps to prevent clustering problems
- Quadratic function implementation can vary
  - `(k + i²),` `i >= 1`: 1, 4, 9, 16, 25, etc.
  - `(k + 2ⁱ),` `i >= 0`: 1, 2, 4, 8, 16, etc.
  - `(k + i + i²),` `i >= 1`: 2, 6, 12, 20, 30, etc.

Quadratic Probing Example

- Exercise:
  - For a table of size 7, insert the following keys: 8, 16, 15, 2, 23
  - Using quadratic formula `(k + i * i)`



Quadratic Probing Concerns

- With many common quadratic functions, it is best to keep the table less than half full
  - No guarantee of finding an empty cell!
  - (Depends on interaction between size and probe)

- Trade off
  - Faster probing and clustering is less common
  - Table cannot have a load factor greater than 50%

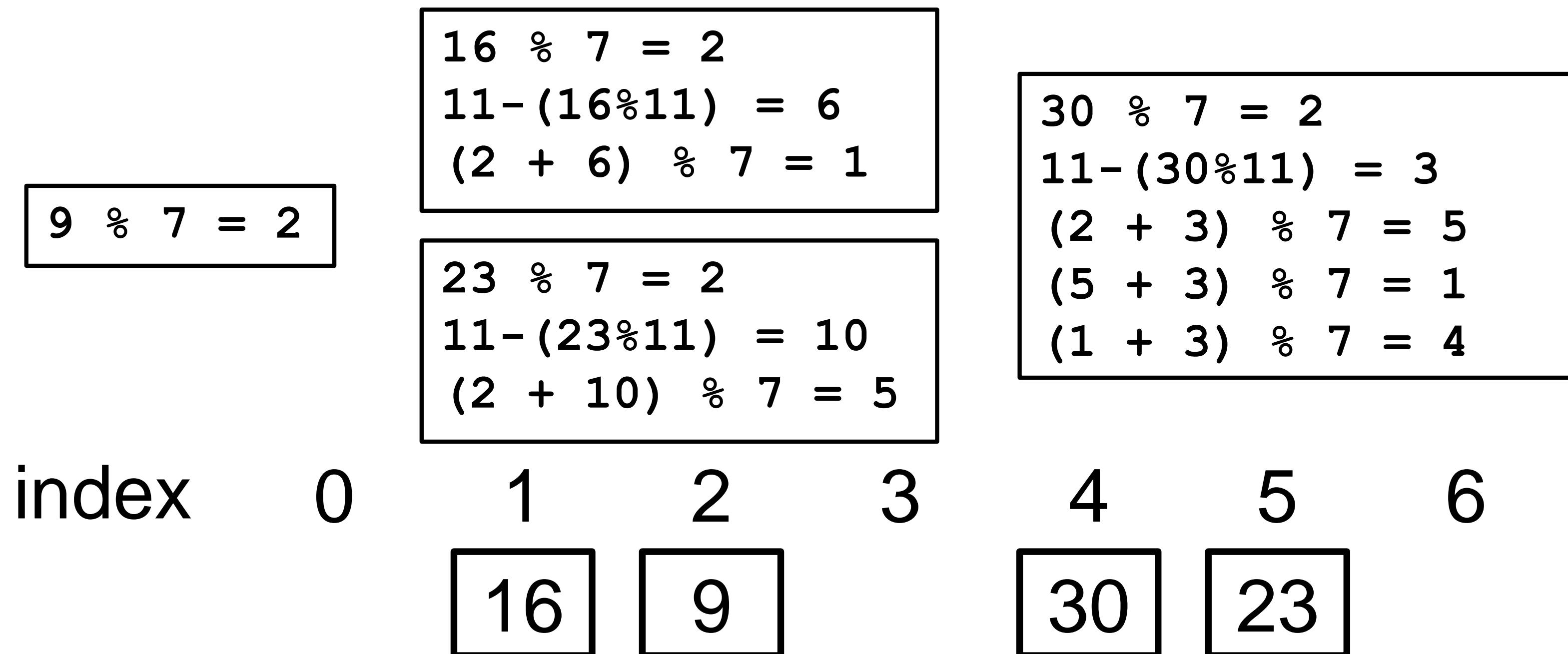
Double Hashing

- ***Double hashing*** is a form of collision-handling where a second hash function determines how much the probe "jumps" by for each probe
- Both hash functions should give uniform distributions, and should be independent
  - Second hash function cannot evaluate to 0! Why?
    - We will continually probe the same index
  - Even values that map to the same initial index are likely to have a unique probe interval

Double Hashing Example

- Exercise:
  - For a table of size 7, insert numbers 9, 16, 23, 30

`h1 = key % 7` `h2 = 11 - (key % 11)`



Hash Tables: Other Details

When to Use a Hash Table?

- Good for when you need fast access
  - Average find/insert/delete is O(1)
- Very poor choice if sorting is a concern
  - Indexing is essentially random based on value
  - Used in dictionaries, maps in various languages
- Hash functions are also used in cryptography
  - The primary goal with crypto is to have hash functions that can't be reverse-engineered

Deleting from a Hash Table

- With open addressing, deletion is a concern
  - “Empty” indexes affect search pattern
- Lazy deletion
  - Mark an element as deleted
    - Treat element as empty when inserting
    - Treat element as occupied when searching
- Rehash the entire table
  - Time consuming, but makes sense in some cases

Resizing a Hash Table

- Ideally, hash tables should be resized when the load factor becomes too high
  - May also be resized if load factor is very low

- Performance of resizing a hash table?
  - O(n)
  - All (key, value) pairs are rehashed to new indexes
- If run-time is critical (such as in real-time systems) we may use another option

Incremental Resizing

- ***Incremental resizing*** is a method of resizing a hash table that is done incrementally
  - Often used for real-time and disk-based tables
- Allocate a new hash table, but keep old one
  - Find and Delete look for value in both tables
  - Insert new values only into new table
    - At each insertion, also move some number of elements from the old table to the new table
      - “Incrementally” rehashing the values