CMSC 341
Hashing (Continued)

Based on slides from previous iterations of this course

Today's Topics

- Review
  - 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

Review: Hash Tables

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 used to store (key, value) pairs
- There are two major components:
  - Bucket array
  - Hash function

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

- 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)

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

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 – 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

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

Horner's Rule

```
static int hash(String key, int tableSize)
{
  int hashVal = 0;

  for (int i = 0; i < key.length(); i++)
     hashVal = 37 * hashVal + key.charAt(i);

  hashVal %= tableSize;
  if(hashVal < 0)
     hashVal += tableSize;

  return hashVal;
}
```

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

- ***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

Handling Collisions

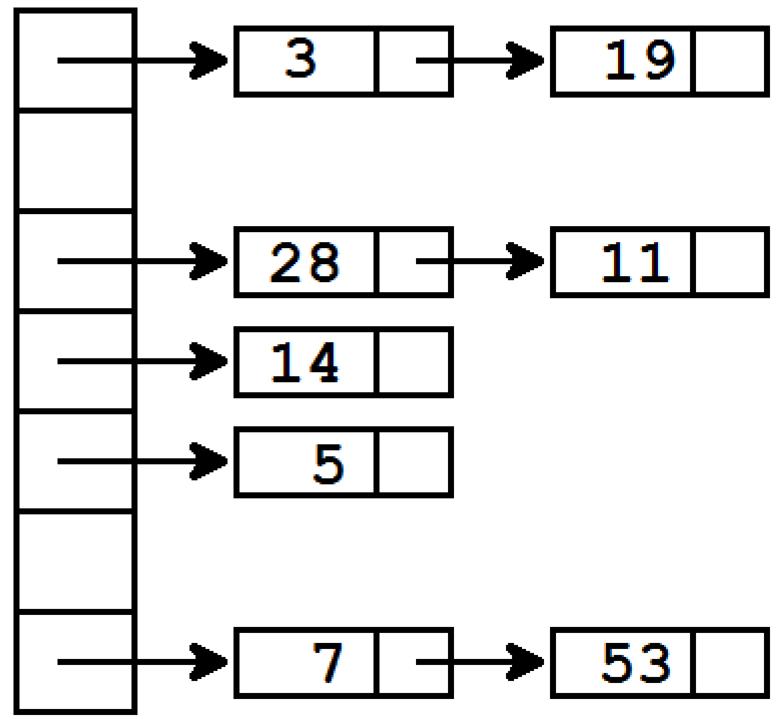
Methods of Handling Collisions

- 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 (list, BST, etc.)



Chaining Example - Division Method

- 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)
  - 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

- ***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” (c)

Linear Probing

- Use a linear function for f(i)

$$f(i) = c * i$$

- Example:

  h' (k) = k mod 10 in a table of size 10 , f(i) = i

So that

  h(k, i) = (k mod 10 + i) mod 10

  Insert the values U={89,18,49,58,69} into the hash table

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	13
		1	14	3	15	2	4	7		9	22			
pushed off by:			1		2	3	2				1			

Linear Probing (cont.)

- Problem: Clustering
  - When the table starts to fill up, performance → O(N)
- Asymptotic Performance
  - Insertion and unsuccessful find, average
    - λ is the "load factor" – what fraction of the table is used
    - Number of probes $\cong$ (½) (1+1/(1-λ)^2)
    - if λ $\cong$ 1, the denominator goes to zero and the number of probes goes to infinity

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)

Linear Probing Performance

- Insert and Find
  - Best case is O(1)
  - Worst case can become O(n)

- Delete is complicated
  - We can’t just delete the entry! (Why not?)
    - The empty space will confuse future probing
  - 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 + i * i), i >= 1`**: 1, 4, 9, 16, 25, etc.
  - **`(k + i + i`**^{2}**`), i >= 1`**: 2, 6, 12, 20, 30, etc.

Quadratic Probing

- Use a quadratic function for f(i)

  $$f(i) = c_2 i^2 + c_1 i + c_0$$

  The simplest quadratic function is $f(i) = i^2$
- Example:

  Let $f(i) = i^2$ and $m = 10$

  Let $h'(k) = k \bmod 10$

  So that

  $$h(k, i) = (k \bmod 10 + i^2) \bmod 10$$

  Insert the value U={89, 18, 49, 58, 69 } into an initially empty hash table

Quadratic Probing (cont.)

- Advantage:
  - Reduced clustering problem

- Disadvantages:
  - Reduced number of sequences
  - No guarantee that empty slot will be found if $\lambda \geq 0.5$, even if m is prime
  - If m is not prime, may not find an empty slot even if $\lambda < 0.5$

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)`**

index	0	1	2	3	4	5	6
		8	16	2		15	23

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!
  - f two keys have the same initial probe position, then their probe sequences are the same.
  - (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 <u>cannot</u> evaluate to 0! Why?
    - We will continually probe the same index

Unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k (h'(k) and h2(k)).

Double Hashing

- Let f(i) use another hash function

$$f(i) = i * h_2(k)$$

  Then $h(k, i) = (h'(k) + i * h_2(k)) \bmod m$

  and probes are performed at distances of

$$h_2(k),\ 2 * h_2(k),\ 3 * h_2(k),\ 4 * h_2(k), \text{ etc}$$

- Choosing $h_2(k)$
  - Don't allow $h_2(k) = 0$ for any k.
  - A good choice:
    $h_2(k) = R - (k \bmod R)$ with R a prime smaller than m
- Characteristics
  - No clustering problem
  - Requires a second hash function

Double Hashing Example

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

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

```
9 % 7 = 2
```

```
16 % 7 = 2
11-(16%11) = 6
(2 + 6) % 7 = 1
```

```
23 % 7 = 2
11-(23%11) = 10
(2 + 10) % 7 = 5
```

```
30 % 7 = 2
11-(30%11) = 3
(2 + 3) % 7 = 5
(5 + 3) % 7 = 1
(1 + 3) % 7 = 4
```

index	0	1	2	3	4	5	6
		16	9		30	23	

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
- 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

Multiple Copies of a Key

- How do we handle data that has duplicate keys, with unique associated values?
  - Depends heavily on the purpose of hash table

- Insert both, search/delete picks one arbitrarily
  - Pros? Cons?

- Replace the original entry with the new one
  - Pros? Cons?

Announcements

- Homework 6 will be out tomorrow (11/15)
  - Due Thursday, November 30th at 8:59:59 PM
- Project 5 will be out soon
  - Due Tuesday, December 12th at 8:59:59 PM

- Next Time:
  - Exam 2 Review