Dictionaries and Hash Tables

CS 240 Spring 2019

Dictionaries

Problem

You have been asked by a phone company to write some software that implements a caller id that will display a name associated with a phone number. Given an phone number, return the caller's name:

phone_num => name

Phone numbers are unique, however, not all phone numbers are in use. How can we store and look up Phone#/name pairs?

Dictionary

- A Dictionary is an ADT that is indexed by Key/Value pairs
  - every key must be unique (i.e., no duplicates).
    - *Keys must be unique and comparable*
      - arrays are dictionaries that use the element index as a key
- Keys are traditionally kept in sorted order
  - however, python dictionary keeps keys in insertion order
- Dictionary, like the stack and queue, is a secondary data structure
  - Uses another internal data structure, only adding behavior to it

Dictionary Implementation

- The underlying implementation Data Structure can range from a Linked List, Array, or a Tree.
  - Linked List:
    - *Insert is O(1) if not sorted*
    - *Search is linear, and far too slow*
      - But we don't have to always traverse to the end if the elements are sorted, why?
    - *List implementation remove is fast, but find is slow*
  - Array
    - *Insert is slow because everything needs to be moved down if sorted*
    - *Find is logn if sorted*
    - *remove requires moving data, which is slow*
    - *For array based, we have O(logn) we can use binary search*
  - Tree
    - *Tree would need to be rebalanced on every insertion and deletion to guarantee logn*

Dictionary in C++

- In C++ STL, the dictionary data structure is called a Map
    - std::map<char,int> first;
      first['a']=10;
        - *Items in the map are kept sorted by their key, which must be comparable*
- Inserting duplicate elements into a map is considered a noop, and nothing happens
    - but it does return an iterator to the existing element
    - Insert uses the STL utility class pair<s, t> to the dictionary
        - *void insert(pair<Comparable, Object>(Comparable k, Object e));*
            - Pair class example: std::pair <int,string> addr(192168001001, “home”);

Hash Tables

Dictionary

- All considered implementations of a dictionary have at best O(logn) access time
    - The access time grows as the data set grows
- What if we need nearly instant access?
    - For example, 911 requires nearly instant lookup for caller ID
- We want the ability to index into the dictionary like an array, so O(1) access time
    - So why not use an array?

Another Solution

- Element whose key, k, is obtained by indexing into the kth position of the array.
  - Perfect for when we can afford to allocate an array with one position for every possible key.
    - *i.e. when the possible keys is a small limited set.*
    - *Arrays are **Direct Access Tables***
- What size array would we need to store 7-digit phone numbers as indexes?
  - 10,000,000

Universe of Keys vs Set of Keys

- Universe of all possible keys (U) is the number of possible keys in the array
  - What is the U for our phone numbers?
    - *10 million*
- Set of keys (K) is the keys actually stored in the dictionary.
  - Even for the most populous city in the U.S., NYC, this is < 8 million
- So what do we do when U is very large, but K is not?
  - As seen with the caller ID problem, an array would result in a lot of wasted space

Problem with Direct Access Tables

- Direct addressing works well when U is relatively small, but what if the keys are 32-bit integers, such as an IP address?
  - Problem1: direct-address table could have only several thousand entries, but more than 4 billion 'spaces'
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be prohibitive
- Solution: map keys to smaller range than U
  - We need only store the number of actual keys rather than space for all possible keys.

Hash table

- Our goal is to store a set of Keys in tables a fraction of the size of U
- Store the values in a data structure called **hash table**.
  - Each element is assigned a unique key.
  - We use hashing to distribute entries (key/value pairs) uniformly across an array.
- By using that key you can still access the element in O(1) time.
  - which was part of our original requirement

Hash Function

- Let’s use an array of size proportional to |**K**| (the actual number of keys) instead of |**U**|
  - However, now we lose the direct-addressing ability.
- Solutions? Hash Function
  - Define a function that performs a transformation on the key that map keys from U to a slot within the hash table.
- Arrays vs Hash Tables
  - With arrays, key k maps to slot A[k].
  - With hash tables, key k maps or “hashes” to slot A[h(k)].
    - *h(k) is the hash value of key k.*

Classwork

Hash Function

```
int hashFunction(int phone_num){
        return phone_num % size_of_hash_table;
}
```

Common Hashing Function

- Using simple division is a common hashing technique that works on many kinds of data
- Mod: Map a key, k, into one of the m slots by taking the remainder of k divided by m. That is,
  - h(k) = k mod m
    - *where k is the key and m is the size of the table*
      - Example: m = 31 and k = 78 h(k) = 16.
- Square Median: Square the key, then take the middle two values
  - Example: 45^2 = 2025 = hashes to 02

Collisions

- Assume we have the following hash function
    - index = key % 31
        - *both 96 and 65 hash to 3*
- Multiple keys can hash to the same slot
    - When two keys hash to the same location this is called a **collision**
- Hash functions should be designed such that collisions are minimized
    - However, avoiding collisions is impossible.

Likelihood of Collisions

- What is the likelihood of collisions?
  - Everyone should go around the room and say their birthday
    - *If someone says your birthday before you do, raise your hand*
- With a k of size 23, and a U of 365, there is a 50% chance of a collision
- This problem illustrates the likelihood of collisions and the impossibility of creating a hash function that does not result in collisions
  - Given a distributed data set

Classwork

Solving Collisions

Solution # 1: Resizing

- We could, whenever there is a collision, resize the entire table to eliminate the collision
  - Resize such that it results in the fewest collisions:
    - *1.5-2 times the size of the current size of table*
    - *Table size should be a prime number*
  - For a table of size 10, we could resize to 17 or 19
- Resizing is the only solution that maintains O(1) access time
- What problem can arise here?
  - Resizing the table can result in a new collision requiring a further resize
  - Uses a lot of extra memory

Solution #2: Open Addressing

- Open Addressing allows for flexible addressing.
  - When collisions occur, use a systematic (consistent) strategy to store elements in free slots of the table.
    - *try alternate cells until empty cell is found.*
- Multiple strategies, for example:
  - Linear probing - Store on the next open slot
    - *When searching for a value, start at the hashed slot, then keep searching until the value is found or an empty slot is encountered*
  - Quadratic Probing - avoids the clustering problem of linear probing
    - *If slot hash(x) % S is full, then we try* $(hash(x) + 1^2) \% S$
    - *If* $(hash(x) + 1^2) \% S$ *is also full, then we try* $(hash(x) + 2^2) \% S$
    - *If* $(hash(x) + 2^2) \% S$ *is also full, then we try* $(hash(x) + 3^2) \% S$

Problems with Open Addressing

- Open addressing compounds the problem of collisions
    - As you place elements in ‘slots’ they don’t belong in, what happens when an object that does belong there gets inserted?
- As you hash table grows, more and more objects are in the ‘wrong’ locations
    - You begin losing the benefits of a hash table
- Open Addressing sacrifices lookup time and complexity for memory efficiency
    - If a table is full, lookup time could be O(n)
        - *Basically an unsorted array*

More Problems with Open Addressing

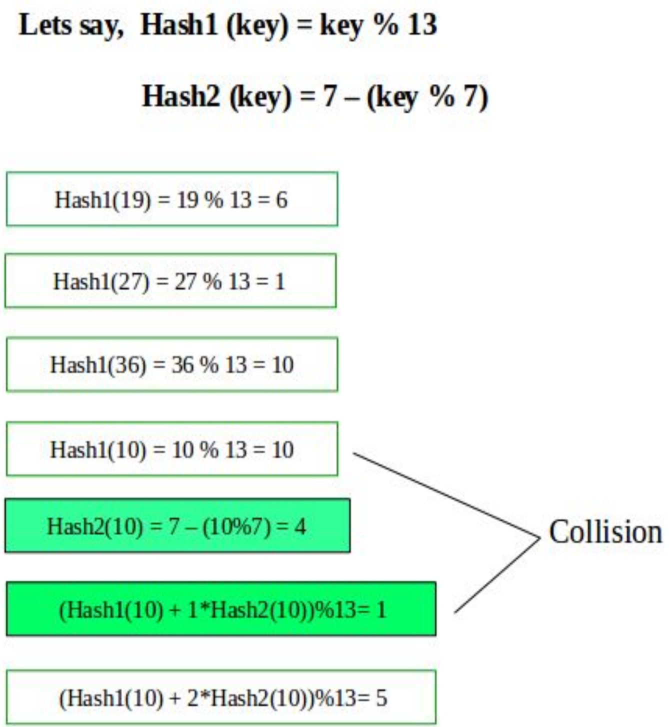
- What happens if you allow deletion
  - You may stop prematurely on deletion
  - The only solution is to have different markers for empty and deleted
- Eventually, as objects get added and deleted, we will have to search the entire table
- Open Addressing is useful when
  - Few or no deletions
  - Few collisions

Solution #3: Double Hashing

- Use a second hash function to re-index on a collision
  - Items that hash to the same initial location will have a different probe sequence
- A good double hashing function:
  - is a simple evaluation
  - produces a different value than the original hash function
- Primes are useful for this
  - R = prime < size
  - hash2(i) = R - (i % R)
- Double Hashing is a form of Open Addressing

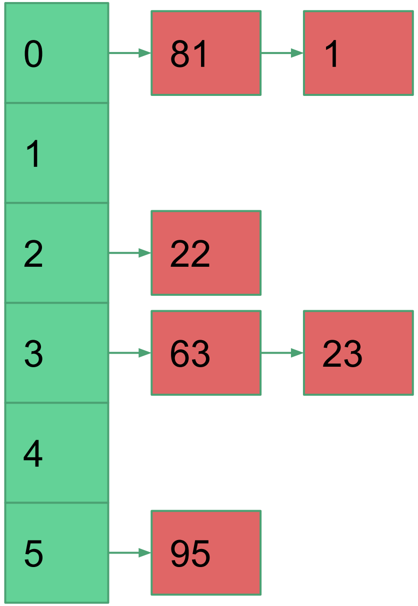
Double Hashing Example

```
if(collision(key)):
    key2 = h2(key)
    i = 1
    While collision(key2):
        key2 = key + i*key2%size
        i++
    key = key2
store key
```



Collision Resolution: Chaining

- The hash table is an array of linked lists
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.
- Notes:
  - As before, elements would be associated with the keys
  - We’re still using the hash function h(k) = k mod m

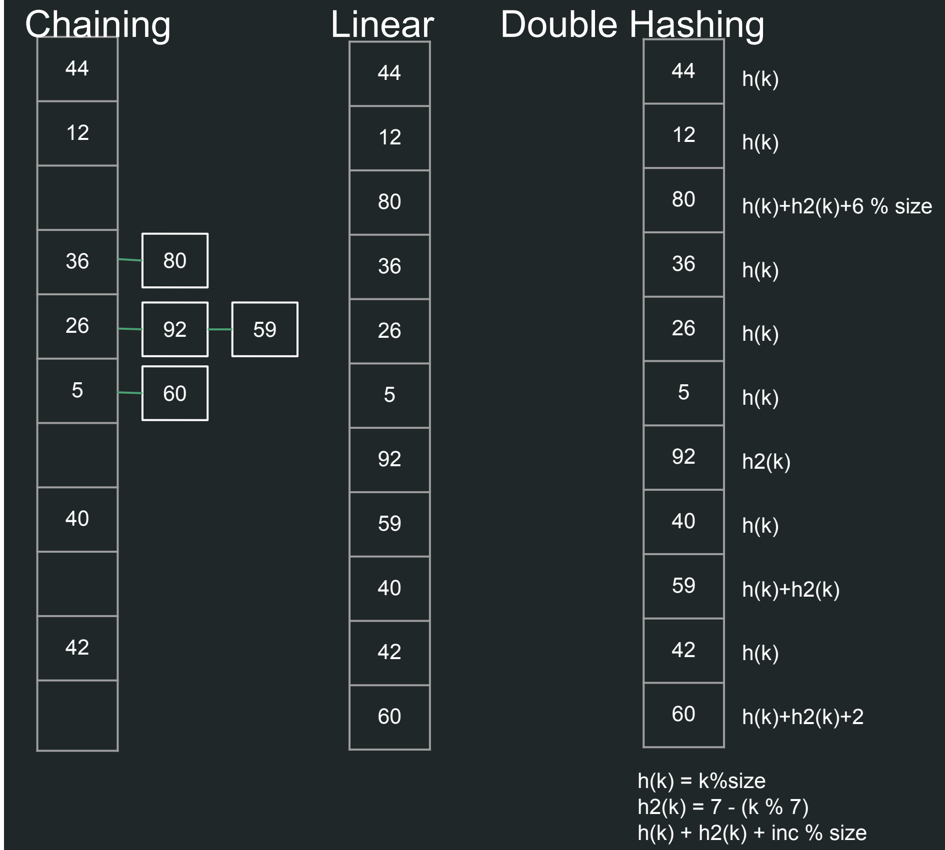


Problems with Chaining

- Similar to Open Addressing, if data clusters, then we lose the benefits of a hash table.
    - Our original requirement was constant time, or near constant time access
- We are also using much more memory
    - The reason we chose a hash table in the first place is because we wanted to use less memory
- Additional logic complexity
    - Overhead required for linked list

Classwork

Resizing The Hash Table



26, 42, 5, 44, 92, 59, 40, 36, 12, 60, 80

The Importance of a good Hash Function

- A Hash Function should satisfy the assumption of **simple uniform hashing.**
  - A hashing function should aim to distribute items in the hash table evenly.
  - An item to be hashed should have equal probability of going into any slot
    - *if your hash function doubles the value, then takes the last 2 numbers, you will never place a value in an odd slot*
- Not possible to satisfy the assumption in practice.
  - Often use heuristics, based on the key domain, to create a hash function
  - Hash value should not depend on patterns that might exist in the data
    - *data changes*