Dictionaries and Hash Tables

CS 580U Fall 2017

Dictionaries

Problem

You have been asked by a VOIP company to write some software that implements a caller id by IP address. Given an IP address, return the caller’s name:

ip_address => name

IP addresses are unique, however, not all IP addresses are in use. How can we store and look up IP/name pairs?

Dictionary

- A Dictionary is an ADT that is indexed by Key/Value pairs
- The Dictionary ADT insists that every comparable index must be unique (i.e., no duplicates).
    - This is achieved through index keys
        - *Keys must be unique*
        - *Keys must be comparable*
    - Arrays are dictionaries that use the element index as a key
- Dictionary's values are "just along for the ride"

CRUD

- Insert: Keys are traditionally kept in sorted order, but not necessarily
  - Python dictionary keeps keys in random order
- Find: allows quick searching of elements
  - If elements are sorted, improves search time
    - *Iterator*
  - Iterators are important for dictionaries because a user could not get a record of the dictionary that she didn't already know the key for.

Tradeoffs

- No traversal
- Dictionary, like the stack and queue, is a secondary data structure
  - Uses another internal data structure, only adding behavior to it
- Underlying data structures have pros and cons
  - List implementation is slow for search
  - Array Implementation is slow for insert and delete
    - *Can use an extra array to speed up deletion at the cost of memory and complexity*
  - Tree is okay for both, but without constant rebalancing you lose the benefits of a tree

Hash Tables

Dictionary

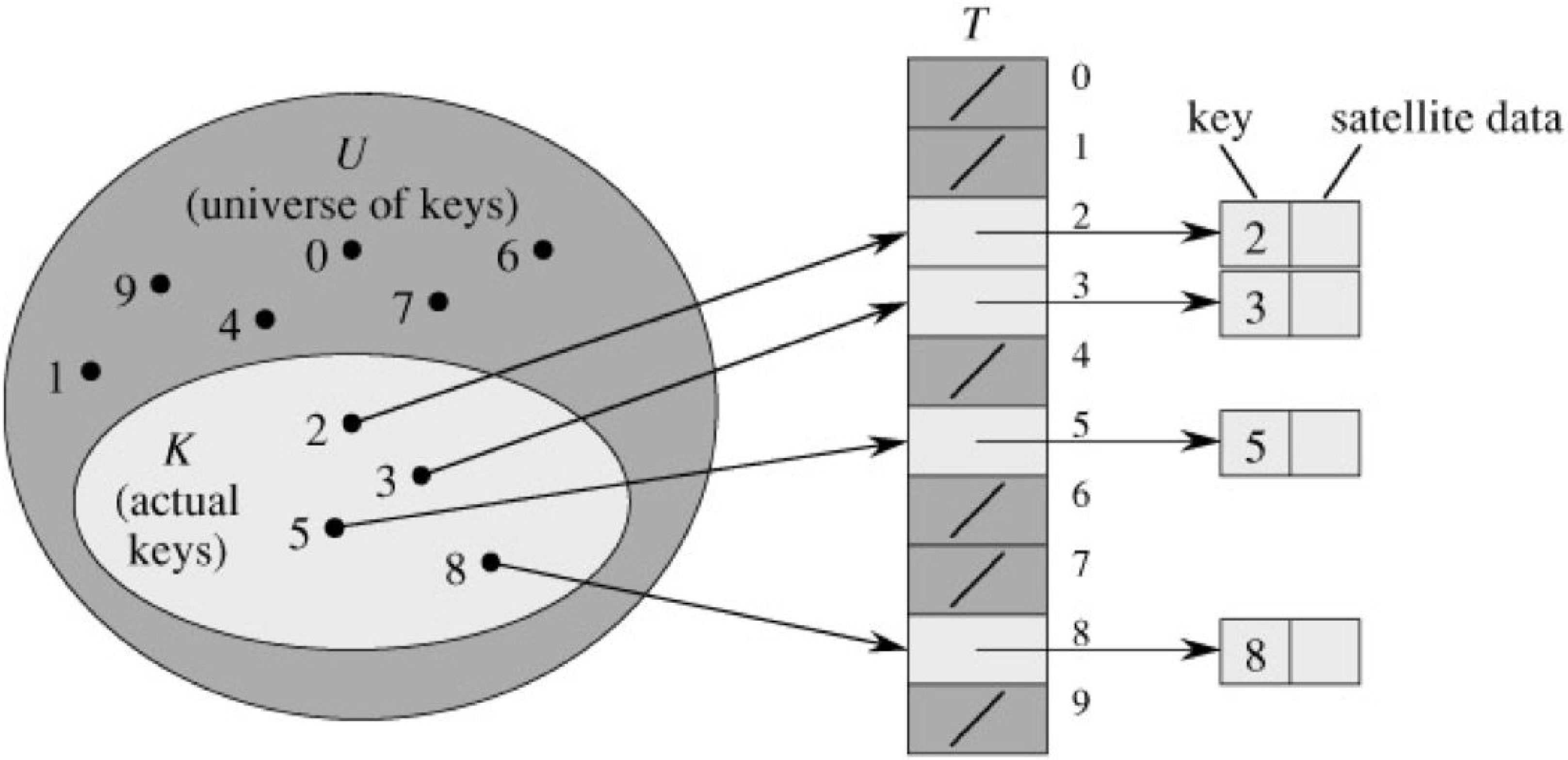
- Dynamic data structure for storing items indexed using keys.
  - Dictionaries define access, not implementation
- How can we implement a Dictionary?
  - Simplest way
    - *As 2 collections*
      - one for keys
      - one for values
    - *A collection of pair objects*
      - Iterate through to find the key
- These solutions grow in time as the data set grows

Another Solution

- What if we need nearly instant access?
  - For example, 911 requires nearly instant lookup for caller ID
- Hash Table
  - A hash table is a generalization of ordinary arrays.
    - *Arrays are Direct Access Tables*
- 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 limited set.*

Hash Table

- Universe of all possible keys (U) is the number of possible keys in the array
  - What is U for a standard array? - limited by memory
    - *What is the U for an alphabet based index (26)?*
- Set of keys (K) actually stored in the dictionary.
  - |K| = n.
    - *The number of keys is the current size of the array*
- What about when U is very large? Arrays are not practical.
  - As seen with the caller ID problem, array would result in a lot of wasted space



U
(universe of keys)
0
6
9
7
4
1
K
(actual
keys)
2
3
5
8
T
0
1
2
3
4
5
6
7
8
9
key
satellite data
2
3
5
8

Problem with Direct Access Tables

- Direct addressing works well when the range m of keys 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
  - We need only store the number of actual keys rather than space for all possible keys.

Hash Function

- Let's use a collection of size proportional to |K| (the actual number of keys)
  - However, now we lose the direct-addressing ability.
- Solutions? Hash Function
  - Define 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;
}
```

- Heap
- Linked List
- Hash
- Balanced Binary Search Tree

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 91 and 65 hash to 2*
- 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 an distributed data set

Classwork

Solving Collisions

Resizing on Collisions

- We could, whenever there is a collision, resize the entire table to eliminate the collision
  - Heuristic studies show that a table with the following properties results in the fewest collisions:
    - *1.5-2 times the size of the set of K*
    - *Table size is a prime number*
  - When we resize, we should find a prime number in the range 1.5-2 times the current size of the table
- What problem can arise here?
  - Resizing the table can result in a new collision requiring a further resize

Collision Resolution: Open Addressing

- Open Addressing stores only in the hash table itself.
    - 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*
    - Double Hashing - use a second hash function to determine the index on a collision
        - *Items that hash to the same initial location will have a different probe sequence*

Problems with Open Addressing

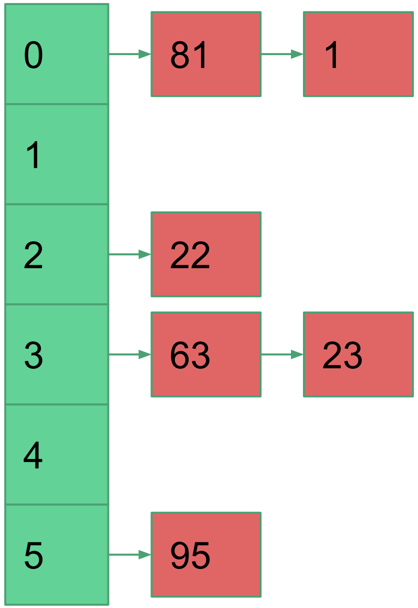
- Open Addressing sacrifices lookup time and complexity for memory efficiency
  - If a table is full, lookup time could be O(n)
    - *Basically an unsorted array*
- What happens if you allow deletion
  - You may stop prematurely on deletion
  - The only solution is to have different markers for empty and deleted
- Eventually, as objects get added and deleted, we will have to search the entire table

More 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 your hash table grows, more and more objects are in the ‘wrong’ locations
  - You begin losing the benefits of a hash table
- Open Addressing is useful when
  - Few or no deletions
  - Few collisions
  - Memory is expensive

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

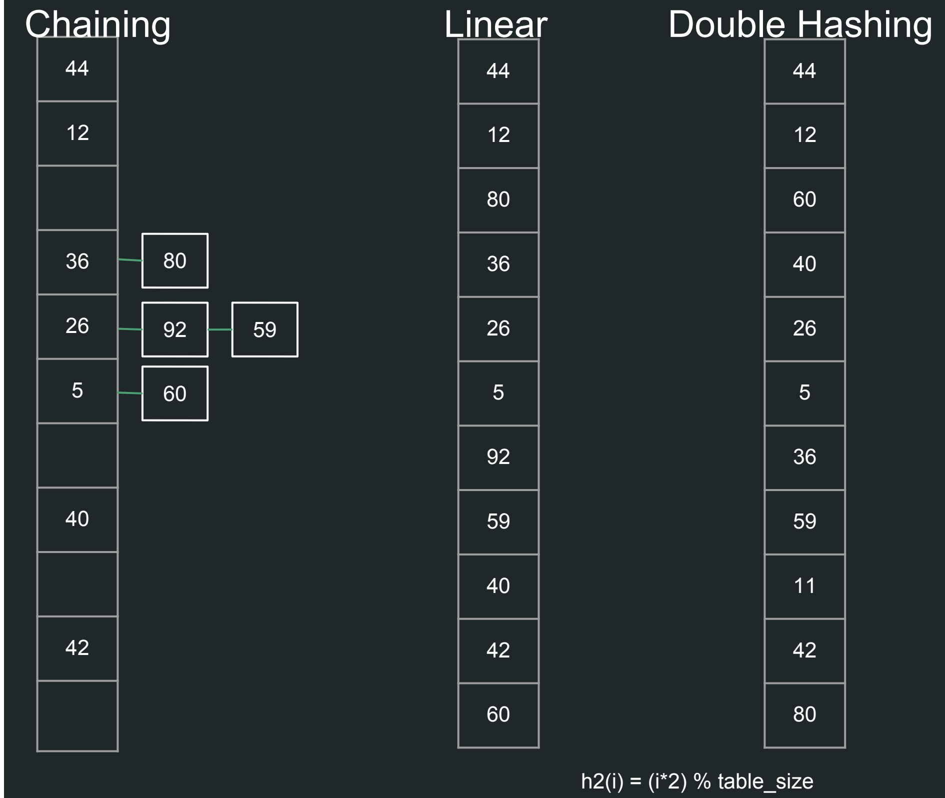


Problems with Chaining

- 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



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*