An Implementation of Dictionaries using Randomized Hashing

Avi Weinstock
Jim Olyha
Neil McGlohon

Problem

- Given a large universe U of possible elements, we want to keep track of a set $S \subseteq U$
  - Insert(x) - Add element x to S
  - Delete(x) - Remove element x from S
  - Lookup(x) - Determine if element x is in S
  - Ideally, we want to perform all of these operations in expected constant time.

Example: Find name from SSN

- Suppose we want to store a bunch of names and be able to look them up based on the person's SSN
- Naive approach:
  - Make an array of size m=1,000,000,000 and store each SSN at the proper index
  - If John Doe's SSN is 123456789, then we set array[123456789] = "John Doe"
  - If we are storing much less than 1,000,000,000 people in the array, this is spacially inefficient

A Slightly Better Solution

- Store a linked list of all of the SSNs that we have encountered so far
- Insert each new SSN and Name as a tuple at the beginning of the list - **O(1)** time
- Search through list when delete() or lookup() are called.
  - If n elements in list, then these would take O(n) time but only O(n) space (down from O(m))

A Better Solution

- HashTable
  - Efficient Adds - O(1) in expectation
  - Efficient Deletes - O(1) in expectation
  - Efficient Lookups - O(1) in expectation
- HashTable is a Key:Value store
  - Maps a given key to a value

Hash Functions

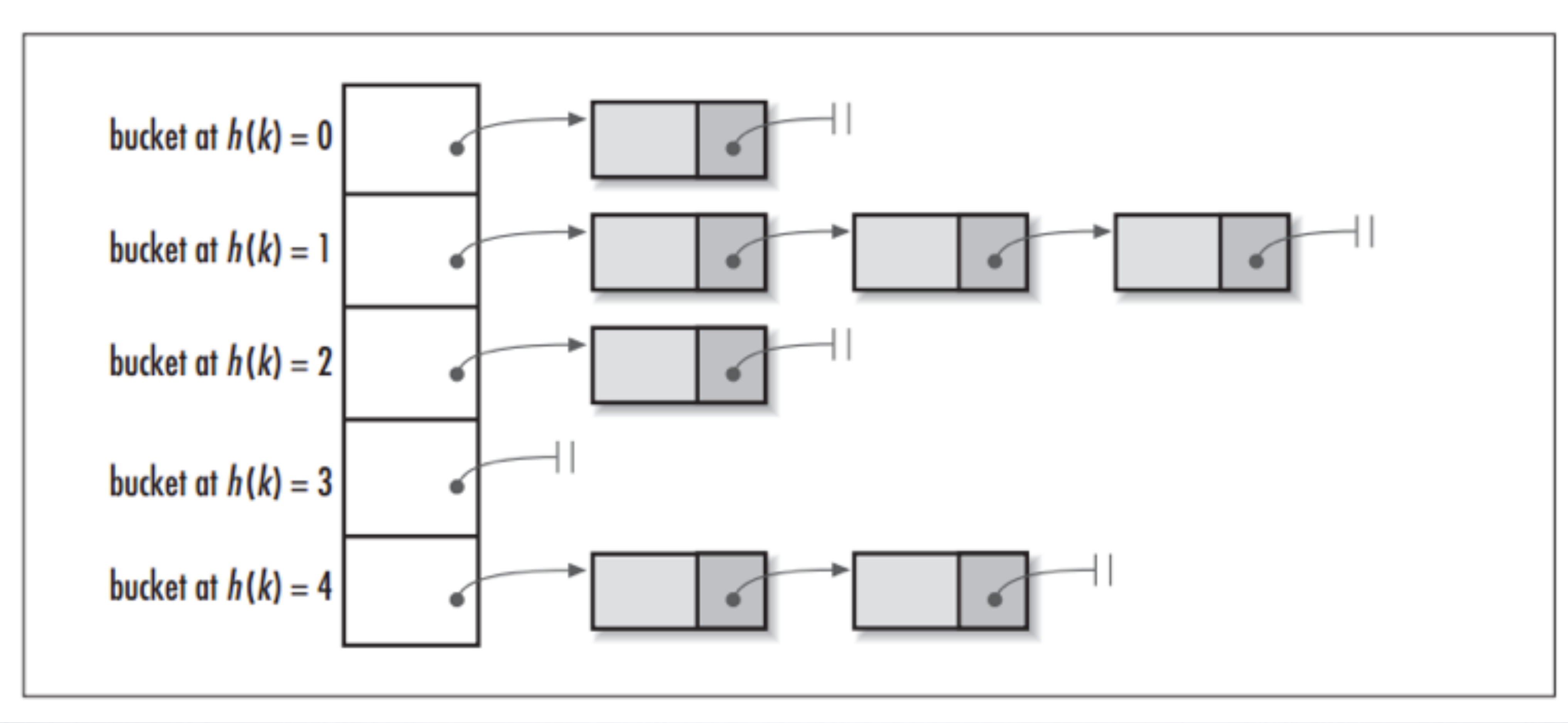
- Choose where to put the value in the table of elements based on some computation on the key
- Example Hash Function: Use first digit
  - h(123456789) = 1
  - h(213456789) = 2
  - This obviously would result in an issue…

Hash Functions

- What if we were to add another element:
  - H(112345678) = 1
  - This is called a collision
  - When there is a collision, we can create a linked list at the hash index

Hash Functions

- Too many collisions?
  - Slower lookup() and delete() operations
    - Searching over long lists



Writing Good Hash Functions

- Want to minimize the chance of collisions
- We don't want to store much information about location of stored elements
- If we have a table of n elements then the probability of collision between any two elements should be 1/n.

Random Hashing Schemes: Outline

- Foolish randomized hashing schemes
- Defining a better randomized hashing scheme
  - Universal class of hashing functions
- Bounding probability of collisions to 1/n
- Given the shown bound, the expected time complexity for each operation (Insert(), Delete(), Lookup()) is O(1)

Foolishly Randomized Hashing Scheme

- Pick an index uniformly at random to place an element in the table

- Probability of two elements colliding is $1/n$

    - Of the n^2 possible choices for the pair of values $(h(u), h(v))$, all are equally likely, and exactly n of these choices results in a collision.[1]

- **What's the problem with this?**

1. Kleinberg and Tardos, Algorithm Design, pg. 738

Foolishly Randomized Hashing Scheme

- "Where did I put it?"
- A lookup() or delete() operation would have to search through at most the entire table.
- Well then why not create a hash table that stores the indices of the values that we stored in our original hash table...?
  - Back at square one

Universal Class of Hash Functions

- Set of hash functions such that[2]:
  - For any pair of elements $u, v \in U$, the probability that a randomly chosen $h \in H$ satisfies that $h(u) = h(v)$ (a collision) is at most $1/n$.
  - Each $h \in H$ can be compactly represented and, for a given $h \in H$ and $u \in U$, we can compute the value $h(u)$ efficiently.

2. Kleinberg and Tardos, Algorithm Design, pg. 738

Universal Class of Hash Functions

- If there is a 1/n chance of collisions, then we can expect the size of each entry in the table (a linked list) to be 1.

- Therefore the expected time complexity of each operation will be:

  - Insert() - O(1)

  - Delete() - O(1)

  - Lookup() - O(1)

Challenge

- To achieve these expected time complexities we must design good hash functions that fall into the Universal Class of Hash Functions