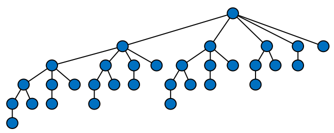
Lecture 16. Hashing and its Applications

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



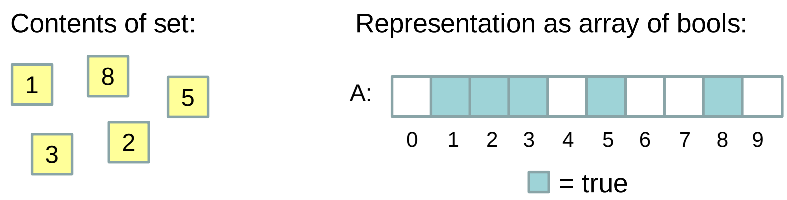
School of Computing
Clemson University
Spring, 2016

Dictionary/Set Data Structures

Insert	Remove	Find		
O(n)	O(n)	O(log n)	Sorted array	1 2 3 5 8
O(1)	O(1), post-find	O(n)	Unsorted array	5 8 1 3 2
O(1), post-find	O(1), post-find	O(n)	Sorted (doubly) linked list	2 → 3 → 5 → 8
O(1)	O(1), post-find	O(n)	Unsorted (doubly) linked list	5 → 3 → 8 → 2
O(log n)	O(log n)	O(log n)	Balanced BST, skip list, B-tree	
O(1) amortized	O(1) amortized	O(1) expected	Universal hash tables (today's topic)	

First Attempt: A Direct Access Table

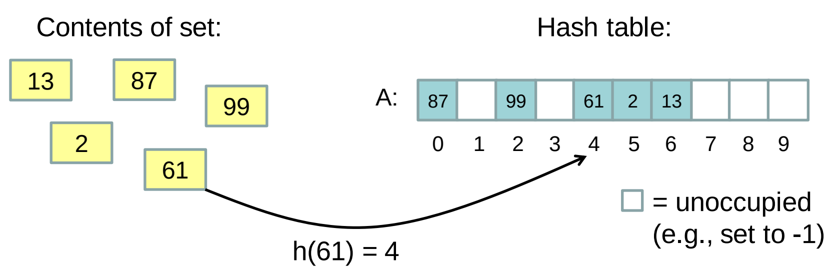
- Maintain a large array of bools
- Presence of key k in structure means A[k] is true.



- Insert, remove, and find all run in O(1) time!
- Serious drawback: **space usage** (and therefore also initialization time).

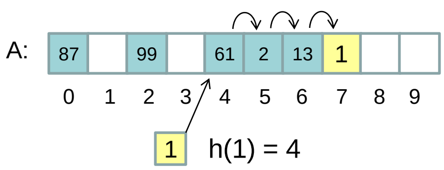
Hash Tables

- Store elements in an array.
- Key k stored in position h(k)
- h() is known as a "hash function" – it maps keys down to the range of indices in our array.
- Example: h(k) = (2971k + 101923) % 10.

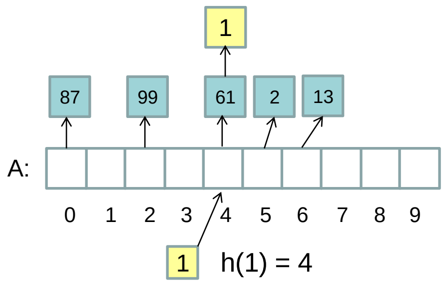


Collision Resolution

- Probing: store elements directly in table; careful when removing elements.
- Chaining: store linked list of colliding elements off each table cell.



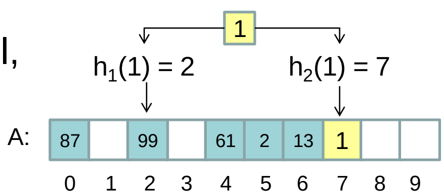
(Sequential) Probing



Chaining

Another Way to Resolve Collisions: Cuckoo Hashing

- Key k stored at either position $h_1(k)$ and $h_2(k)$.
  (make sure h_1 and h_2 map to different positions!)
- Find, remove now trivial, and O(1) time.
- Insert may lead to a chain of “evictions”.
  - If this goes on for too long (usually O(log n) steps), give up and rebuild the entire table with two new hash functions h_1 and h_2.
- When table gets too full, rebuild at 2x size, just as before.



Why “Hashing”…?

Hash (n) A dish of cooked meat cut into small pieces and recooked, usually with potatoes.

7

Why “Hashing”…?

Hash (n) A dish of cooked meat cut into small pieces and recooked, usually with potatoes.

Hash (v) To jumble or mix up.

8

Choosing a Good Hash Function In Practice

- We want h() to behave somewhat "randomly".
- Good Examples:
  - h(k) = k mod m.
  - h(k) = $\lfloor mk / C \rfloor$.
  - h(k) = (ak) mod m.
  - h(k) = (ak + b) mod m.

m = table size
C = max possible key value

a and b should be chosen in a somewhat "arbitrary" fashion (primes often used).

- Be sure to fully utilize "all the bits" in a key. For example, h(k) = k mod 256 is a somewhat poor hash function if k is a 32-bit IP address.
- Be sure to use the entire hash table; e.g., make sure h(k) isn't always even.

9

Amortized Rebuilding

- How do we pick the initial size of our table?
- We ideally want to keep m = Θ(n), where m = table size and n = # of elements stored in table.
- To do this, double table size and re-hash when table becomes too full, and halve table size and re-hash when table becomes too empty.
- Since it takes only linear time to re-build the table, this makes insert and delete take + O(1) extra amortized time.

Hashing Arrays

- Any complicated object can be "serialized" and represented by an array of small integers.
  E.g., "Cpsc" → 'C', 'p', 's', 'c' → 67, 112, 115, 99.
- So how do we hash an array A[0 … N – 1] to get an output value in the range 0 … M – 1?

11

Hashing Arrays

- Any complicated object can be "serialized" and represented by an array of small integers.
  E.g., "Cpsc" → 'C', 'p', 's', 'c' → 67, 112, 115, 99.
- So how do we hash an array A[0 … N – 1] to get an output value in the range 0 … M – 1?

- How about something like this:

$$h(A[]) = (A[0] + A[1] + \ldots + A[N-1]) \bmod M$$

12

Polynomial Hash Functions

- Think of A[] as the coefficients of a polynomial:

  $p_A(x) = A[0] + A[1]\ x + A[2]\ x^2 + \ldots + A[N\text{-}1]\ x^{N-1}$

- To hash A[], evaluate p(x) mod M at a randomly-chosen point x. How do we do this quickly?
- If M is prime, then the probability two different arrays A and B collide at most (N-1) / M:
  – If A and B collide, then x is a root of $p_A(x) - p_B(x)$ (mod M).
  – A polynomial of degree N-1 can have at most N-1 roots (true in any algebraic field, such as arithmetic over complex numbers or over integers modulo a prime)

13

The Case Against Deterministic Hash Functions

- Any deterministic hash function has a "bad" set of input keys…
- If we are hashing n keys in the range 0…C-1 down to a table A[0…m-1] and C ≥ m(n – 1) + 1, then some set of ≥ n different keys will be mapped to the same table cell.
- This reduces our hash table to nothing more than a fancy linked list, so *find* takes Θ(n) time!
- So if we want a good worst-case guarantee for *find,* we cannot use a deterministic hash function. We **must** use randomness in some way.

14

Why Not Use a Completely Random Hash Function?

- Suppose we choose a hash function h(k) that maps every key $k \in \{0, \ldots, C-1\}$ to a completely random location in $\{0, \ldots, m-1\}$.
- This gives us **E**[L] = n / m, so *find* does indeed run in O(1 + n / m) = O(1) time.
  - Please always assume (from here on) that we use dynamic resizing to maintain m = Θ(n), so n / m = O(1).
  - How do we show that **E**[L] = n / m? Linearity of expectation!
- Fatal flaw: …?

15

Why Not Use a Completely Random Hash Function?

- Suppose we choose a hash function h(k) that maps every key $k \in \{0, \ldots, C-1\}$ to a completely random location in $\{0, \ldots, m-1\}$.
- This gives us **E**[L] = n / m, so *find* does indeed run in O(1 + n / m) = O(1) time.
  - Please always assume (from here on) that we use dynamic resizing to maintain m = Θ(n), so n / m = O(1).
  - How do we show that **E**[L] = n / m? Linearity of expectation!

- Fatal flaw: requires Θ(C) space to store the hash function, same as the direct access table.

16

Universal Hashing

- A (randomized) hash function is **universal** if the probability (over random parameters in the function) of two different keys colliding is O(1 / m).
- Example of a universal hash function (several others are discussed in the book):

$$h(k) = [(ak + b) \bmod p] \bmod m, \text{ with } \begin{cases} p\text{: any prime number} \geq C. \\ a\text{: random integer in } \{1, \ldots, p - 1\} \\ b\text{: random integer in } \{0, \ldots, p\}. \end{cases}$$

- With a universal hash function, *find* runs in O(1) expected time with chaining!
  – Easy proof using linearity of expectation…

17

O(1) Expected Running Time for Find With Universal Hashing

- Let T denote the running time of an <u>unsuccessful</u> call to find(k).
  – Clearly, **E**[T] ≥ the expected time required for a successful find operation, so our analysis also provides a bound on the expected time of a successful find.
- Compute **E**[T] using linearity of expectation:

  Let $k_1 \ldots k_n$ denote the keys stored in our hash table.

  Write $T = X_1 + X_2 + \ldots + X_n$.

  X_j: indicator random variable taking value 1 if $h(k_j) = h(k)$.

  Since $k \neq k_j$, $\mathbf{E}[X_j] \leq 1 / m$ (using universal hashing).

  So $\mathbf{E}[T] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_n] = n\,\mathbf{E}[X_j] \leq n / m = O(1)$.

18

Example Applications

- Consider the following problems:
  - **Element Uniqueness**: Given n numbers $A_1 \ldots A_n$, are all of these distinct, or are two of them equal?
  - **Set Intersection / Union / Difference**: Given two sets A and B (specified by unsorted arrays) containing n elements in total, output an array containing A ∩ B, A U B, or A \ B.
- All of these problems have Ω(n log n) worst-case lower bounds in the comparison model.
- However, we can solve them in O(n) expected time with universal hashing.

Levels of Independence Needed for O(1) Expected Performance

- "Strongly" universal hashing functions are not fully random, but provide a weaker guarantee of <u>pairwise independence</u>.
  - If we only look at 2 keys at a time, these are hashed in a completely random fashion, just as with a fully random function (but not for 3 or more keys…)
- Higher levels of independence are possible using higher-degree polynomial hash functions. For example, this hash function is 4-wise independent:

  $$h(k) = [(ak^3 + bk^2 + ck + d) \bmod p] \bmod m$$
- Recent result [Pagh et al. '11, Patrascu-Thorup '10]: 5-wise independence necessary and sufficient for O(1) expected performance with linear probing.

Hashing: Much More than Just a Good Set Data Structure…

- The general idea of mapping a large, complicated object down to a simpler object appears in many areas of computing…
- Example: by hashing a file down to a small integer “fingerprint”, we can:
  - Compare (approximately) two files extremely quickly.
  - Test if two files at different locations (e.g., original, backup) are identical with a minimum amount of communication.
  - Detect tampering in critical system files.
  - Ensure integrity of the file when transferred over a noisy channel, by appending the fingerprint as a checksum.

Hashing in Security

- We can detect tampering in data (a file, or a message) if we also store a hash of the data.
  - Particularly if we use a cryptographic hash function (e.g., MD5, SHA1), which has specifically been designed to be hard to invert.
  - Difficult to change the data while keeping the same hash: by the birthday paradox, it takes ~2^{64} guesses to find a single colliding pair of keys if our hash output is a 128-bit integer.
- Store passwords on a computer system as hashes instead of “in the clear” – still allows you to log in!