CS 240 – Data Structures and Data Management

Module 7: Dictionaries via Hashing

A. Storjohann

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2018

References: Sedgewick 12.2, 14.1-4

version 2018-10-25 10:20

Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies

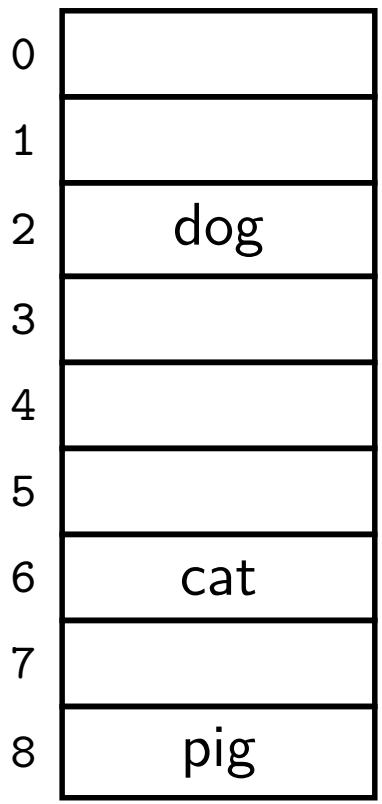
Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies

Direct Addressing

Consider special situation: For a given $M \in \mathbb{N}$, every key k is an integer with $0 \leq k < M$.

We can implement a dictionary easily: Use an array A of size M that stores (k, v) via $A[k] \leftarrow v$.



- *search*(k): Check whether $A[k]$ is empty
- *insert*(k, v): $A[k] \leftarrow v$
- *delete*(k): $A[k] \leftarrow$ *empty*

Each operation is $\Theta(1)$.
Total storage is $\Theta(M)$.

What sorting algorithm does this remind you of?

Hashing

Direct addressing isn't possible if keys are not integers.
And the storage is very wasteful if $n \ll M$.

Hashing idea: Map the keys to a small range of integers and then use direct addressing.

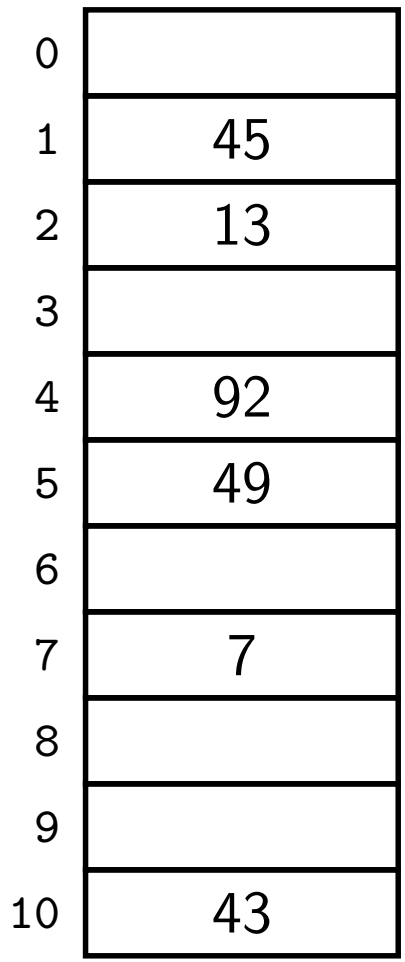
Details:

- **Assumption:** keys come from some *universe* U.
  (Typically $U = \mathbb{N}$.)
- We design a *hash function* $h : U \to \{0, 1, \ldots, M-1\}$.
  (Commonly used: $h(k) = k \bmod M$. We will see other choices later.)
- Store dictionary in *hash table*: Array T of size M.
  An item with key k should be stored in $T[h(k)]$.

Hashing example

$U = \mathbb{N}$, $M = 11$, $\quad h(k) = k \bmod 11$.

The hash table stores keys 7, 13, 43, 45, 49, 92. (Values are not shown).



Collisions

- Generally hash function h is not injective, so many keys can map to the same integer.
  - For example, $h(46) = 2 = h(13)$.
- We get *collisions*: we want to insert (k, v) into the table, but $T[h(k)]$ is already occupied.

- Two basic strategies to deal with collisions:
  - Allow multiple items at each table location (chaining)
  - Allow each item to go into multiple locations (open addressing)

    Could allow many other locations (probe sequence) or just one other (cuckoo hashing).

Load factor and re-hashing

- We will evaluate strategies by the cost of *search*, *insert*, *delete*
- This evaluation is done in terms of the *load factor* $\alpha = n/M$.
  - The example has load factor $\frac{6}{11}$.
- We keep the load factor small by *rehashing* when needed:
  - Keep track of n and M throughout operations
  - If α gets too large, create new (twice as big) hash-table, new hash-functions and re-insert all items in the new table.
  - Rehashing costs $\Theta(M + n)$ but happens rarely enough that we can ignore this term when amortizing over all operations.
  - We should also re-hash when α gets too small, so that the space is always $\Theta(n)$.

Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies

Separate Chaining

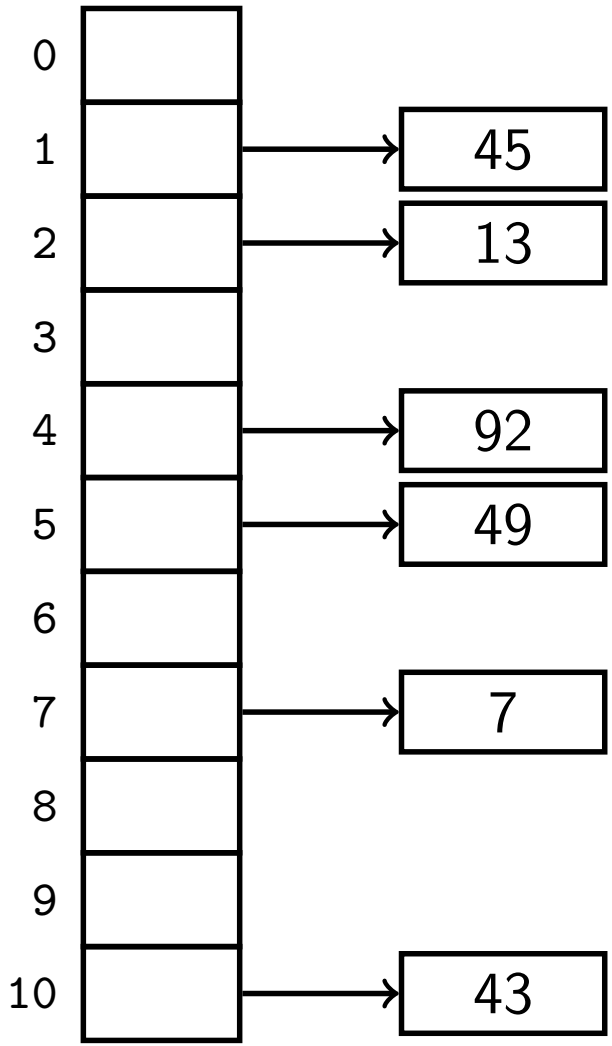
Each table entry is a *bucket* containing 0 or more KVPs.
This could be implemented by any dictionary (even another hash table!).

The simplest approach is to use an unsorted linked list in each bucket.
This is called collision resolution by *separate chaining*.

- *search*(k): Look for key k in the list at $T[h(k)]$.
- *insert*(k, v): Add (k, v) to the front of the list at $T[h(k)]$.
- *delete*(k): Perform a search, then delete from the linked list.

Chaining example

$M = 11, \qquad h(k) = k \bmod 11$



Complexity of chaining

Recall the **load factor** $\alpha = n/M$.

Uniform Hashing Assumption: Each hash function value is equally likely. (This depends on the input and how we choose the function $\rightsquigarrow$ later.)

Assuming uniform hashing, average bucket size is exactly α.

Analysis of operations:

search : $\Theta(1+\alpha)$ average-case, $\Theta(n)$ worst-case

insert : $O(1)$ worst-case, since we always insert in front.

delete : Same cost as *search*

space : $\Theta(M+n) = \Theta(n/\alpha + n)$.

If we maintain $\alpha \in \Theta(1)$, then average costs are $O(1)$ and space is $\Theta(n)$. This is typically accomplished by rehashing whenever $n < c_1 M$ or $n > c_2 M$, for some constants c_1, c_2 with $0 < c_1 < c_2$.

Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies

Open addressing

Main idea: Each hash table entry holds only one item,
but any key k can go in multiple locations.

search and *insert* follow a *probe sequence* of possible locations for key k:
$\langle h(k,0), h(k,1), h(k,2), \ldots \rangle$ until an empty spot is found.

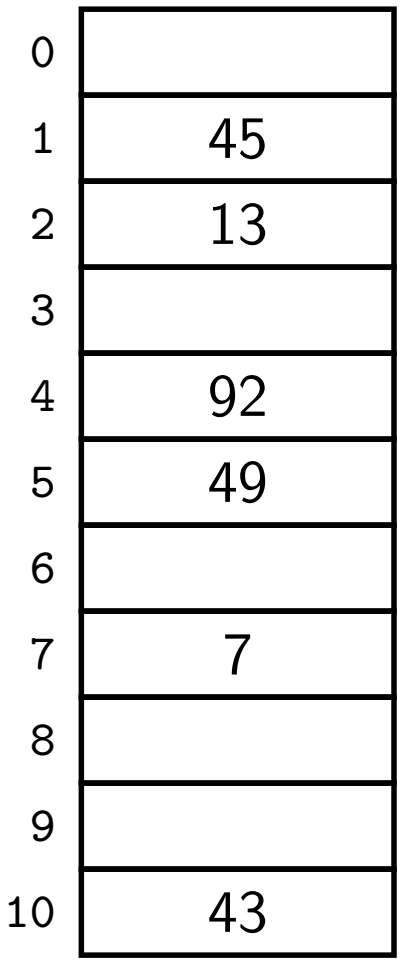
delete becomes problematic:

- Cannot leave an *empty* spot behind; the next search might otherwise not go far enough.
- Idea 1: Move later items in the probe sequence forward.
- Idea 2: **lazy deletion**. Mark spot as *deleted* (rather than *empty*) and continue searching past deleted spots.

Simplest method for open addressing: *linear probing*
$h(k,i) = (h(k) + i) \bmod M$, for some hash function h.

Linear probing example

$M = 11, \qquad h(k, i) = (h(k) + i) \bmod 11.$



Probe sequence operations

```
probe-sequence-insert(T, (k, v))
1.    for (j = 0; j < M; j++)
2.        if T[h(k, j)] is "empty" or "deleted"
3.            T[h(k, j)] = (k, v)
4.            return "success"
5.    return "failure to insert"
```

```
probe-sequence-search(T, k)
1.    for (j = 0; j < M; j++)
2.        if T[h(k, j)] is "empty"
3.            return "item not found"
4.        else if T[h(k, j)] has key k
5.            return T[h(k, j)]
6.         // ignore "deleted" and keep searching
7.    return "item not found"
```

Independent hash functions

- Some hashing methods require **two** hash functions h_1, h_2.
- These hash functions should be **independent** in the sense that the random variables $P(h_1(k) = i)$ and $P(h_2(k) = j)$ are independent.
- Using two modular hash-functions may often lead to dependencies.
- Better idea: Use *multiplicative method* for second hash function: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$,
  - A is some floating-point number with $0 < A < 1$
  - $h(k)$ computes fractional part of kA (which is in $[0, 1)$
  - Then multiply with M and round down.

  Knuth suggests $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618$.

Double Hashing

- Assume we have two hash independent functions h_1, h_2.
- Assume further that $h_2(k) \neq 0$ and that $h_2(k)$ is relative prime with the table-size M for all keys k. ($\rightsquigarrow$ Choose M prime.)

- *Double hashing*: open addressing with probe sequence

$$h(k, i) = h_1(k) + i \cdot h_2(k) \bmod M$$

- *search*, *insert*, *delete* work just like for linear probing, but with this different probe sequence.

Double hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies

Cuckoo hashing

This is a relatively new idea from Pagh and Rodler in 2001.

Again, we use two independent hash functions h_1, h_2.

Main idea: An item with key k can **only** be in $T[h_1(k)]$ or $T[h_2(k)]$.

- *Search* and *Delete* then take constant time.
- *Insert* *always* puts a new item into $T[h_1(k)]$.

  If $T[h_1(k)]$ was occupied: "kick out" the other item, which we then attempt to re-insert into its alternate position.

  This may lead to a loop of "kicking out". We detect this by aborting after too many attempts.

  In case of failure: rehash with a larger M and new hash functions.

  Insert may be slow, but is expected to be constant time if the load factor is small enough.

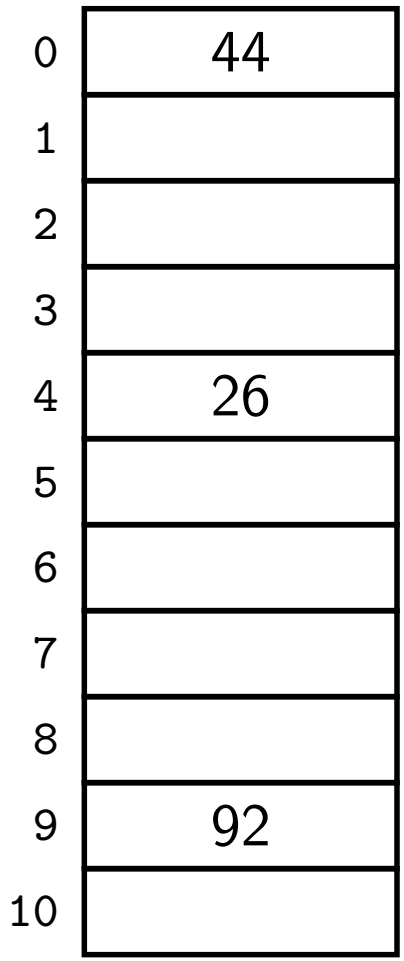
Cuckoo hashing insertion

cuckoo-insert(T, x)
T: hash table, x: new item to insert

```
1.    y ← x,   i ← h1(x.key)
2.    do at most n times:
3.        swap(y, T[i])
4.        if y is "empty" then return "success"
5.        // swap i to be the other hash-location
6.        if i = h1(y.key) then i ← h2(y.key)
7.        else i ← h1(y.key)
8.    return "failure"
```

Cuckoo hashing example

$M = 11, \qquad h_1(k) = k \bmod 11, \qquad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$



Complexity of open addressing strategies

For any open addressing scheme, we *must* have $\alpha < 1$ (why?).
Cuckoo hashing requires $\alpha < 1/2$.

The following gives asymptotic average-cast costs:

	search	*insert*	*delete*
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$
Double Hashing	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$
Cuckoo Hashing	1	$\frac{\alpha}{(1-2\alpha)^2}$	1

Summary: All operations have $O(1)$ average-case run-time if the hash-function is uniform and α is kept sufficiently small.

Outline

1. Dictionaries via Hashing
   - Hashing Introduction
   - Separate Chaining
   - Probe Sequences
   - Cuckoo hashing
   - Hash Function Strategies

Choosing a good hash function

- **Goal:** Satisfy uniform hashing assumption
  (each hash-index is equally likely)
- Proving this is usually impossible, as it requires knowledge of the input distribution and the hash function distribution.
- We can get good performance by choosing hash-function tht is
  - unrelated to any possible patterns in the data, and
  - depends on all parts of the key.
- We saw two basic methods for integer keys:
  - **Modular method**: $h(k) = k \bmod M$.
    We should choose M to be a prime.
  - **Multiplicative method**: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$,
    for some constant floating-point number A with $0 < A < 1$.

Universal Hashing

Every hash functions *must* fail for some sequences of inputs.
Everything hashes to same value. $\rightsquigarrow$ terrible worst case!

Rescue: Randomization!

- When initializing or re-hashing, choose a prime number $p > M$ and *random* numbers $a, b \in \{0, \ldots p-1\}$, $a \neq 0$.
- Use as hash function

$$h(k) = ((ak + b) \bmod p) \bmod M$$

- Can prove: For any (fixed) numbers $x \neq y$, the probability of a collision using this random function h is at most $\frac{1}{M}$.
- Therefore the expected run-time is $O(1)$ if we keep the load-factor α small enough.

We have again shifted the average-case performance to the expected performance via randomization.

Multi-dimensional Data

What if the keys are multi-dimensional, such as strings in Σ^*?

Standard approach is to *flatten* string w to integer $f(w) \in \mathbb{N}$, e.g.

$$\begin{aligned} A \cdot P \cdot P \cdot L \cdot E &\rightarrow (65, 80, 80, 76, 69) \quad \text{(ASCII)} \\ &\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 68R^0 \\ &\qquad \text{(for some radix } R\text{, e.g. } R = 255) \end{aligned}$$

We combine this with a modular hash function: $h(w) = f(w) \bmod M$

To compute this in $O(|w|)$ time without overflow, use Horner's rule and apply mod early. For exampe, $h(APPLE)$ is

$$\Big(\Big(\Big(\Big(\Big(\big((65R{+}80) \bmod M\big)R{+}80\Big) \bmod M\Big)R{+}76\Big) \bmod M\Big)R{+}69\Big) \bmod M$$

Hashing vs. Balanced Search Trees

Advantages of Balanced Search Trees

- $O(\log n)$ worst-case operation cost
- Does not require any assumptions, special functions, or known properties of input distribution
- Predictable space usage (exactly n nodes)
- Never need to rebuild the entire structure
- supports ordered dictionary operations (rank, select etc.)

Advantages of Hash Tables

- $O(1)$ operations (if hashes well-spread and load factor small)
- We can choose space-time tradeoff via load factor
- Cuckoo hashing achieves $O(1)$ worst-case for search & delete