

CS 240 – Data Structures and Data Management

Module 7: Dictionaries via Hashing

Mark Petrick

Based on lecture notes by many previous cs240 instructors

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

Fall 2020

References: Sedgewick 12.2, 14.1-4
Goodrich & Tamassia 6.4

version 2020-10-13 11:58

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

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

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

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

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

Direct Addressing

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

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

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

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

Each operation is $\Theta(1)$.

Total space is $\Theta(M)$.

What sorting algorithm does this remind you of?

Direct Addressing

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

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

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

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

Each operation is $\Theta(1)$.

Total space is $\Theta(M)$.

What sorting algorithm does this remind you of?

Count Sort

Hashing

Two disadvantages of direct addressing:

- It cannot be used if the keys are not integers.
- It wastes space if M is unknown or $n \ll M$.

Hashing idea: Map (arbitrary) keys to integers in range $\{0, \dots, M-1\}$ and then use direct addressing.

Details:

- **Assumption:** We know that all keys come from some **universe** U . (Typically $U = \mathbb{N}$.)
- We design a **hash function** $h : U \rightarrow \{0, 1, \dots, M-1\}$. (Commonly used: $h(k) = k \bmod M$. We will see other choices later.)
- Store dictionary in **hash table**, i.e., an array T of size M .
- An item with key k should ideally be stored in **slot** $h(k)$, i.e., at $T[h(k)]$.

Hashing example

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

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

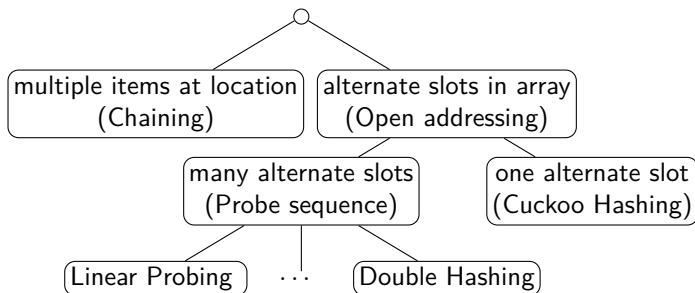
0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Collisions

- Generally hash function h is not injective, so many keys can map to the same integer.
 - ▶ For example, $h(46) = 2 = h(13)$ if $h(k) = k \bmod 11$.
- We get **collisions**: we want to insert (k, v) into the table, but $T[h(k)]$ is already occupied.

Collisions

- Generally hash function h is not injective, so many keys can map to the same integer.
 - ▶ For example, $h(46) = 2 = h(13)$ if $h(k) = k \bmod 11$.
- We get **collisions**: we want to insert (k, v) into the table, but $T[h(k)]$ is already occupied.
- There are many strategies to resolve collisions:



Outline

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

Separate Chaining

Simplest collision-resolution strategy: Each slot stores a **bucket** containing 0 or more KVPs.

- A bucket could be implemented by any dictionary realization (even another hash table!).
- The simplest approach is to use unsorted linked lists for buckets. This is called collision resolution by **separate chaining**.

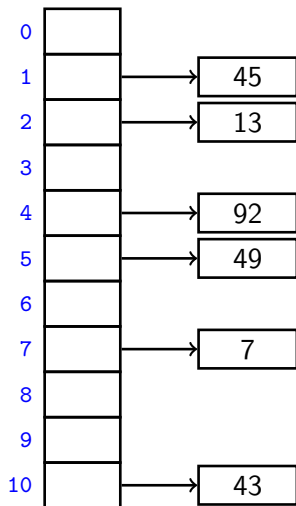
Separate Chaining

Simplest collision-resolution strategy: Each slot stores a **bucket** containing 0 or more KVPs.

- A bucket could be implemented by any dictionary realization (even another hash table!).
- The simplest approach is to use unsorted linked lists for buckets. This is called collision resolution by **separate chaining**.
- *search*(k): Look for key k in the list at $T[h(k)]$.
Apply MTF-heuristic!
- *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

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

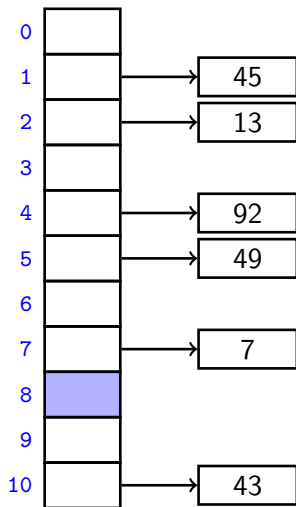


Chaining example

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

insert(41)

$$h(41) = 8$$

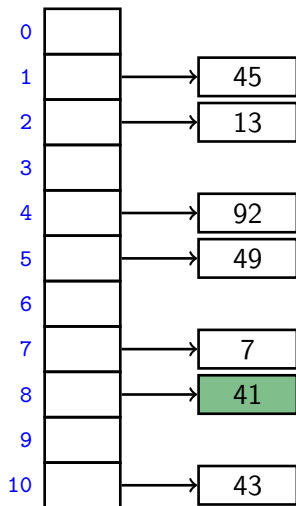


Chaining example

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

insert(41)

$$h(41) = 8$$

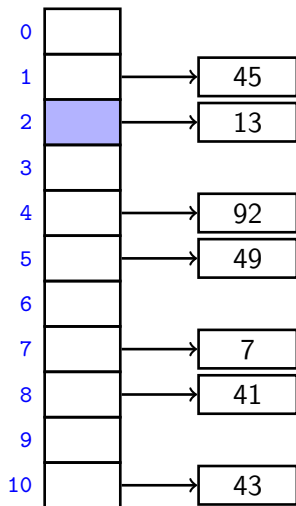


Chaining example

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

insert(46)

$$h(46) = 2$$

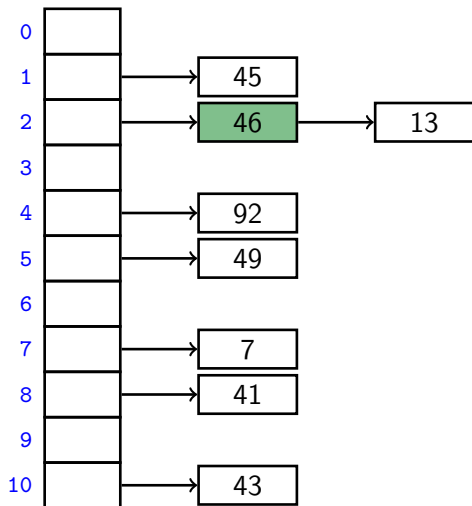


Chaining example

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

insert(46)

$$h(46) = 2$$

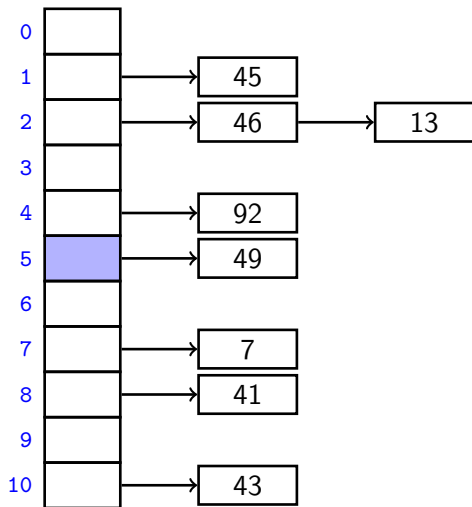


Chaining example

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

insert(16)

$$h(16) = 5$$

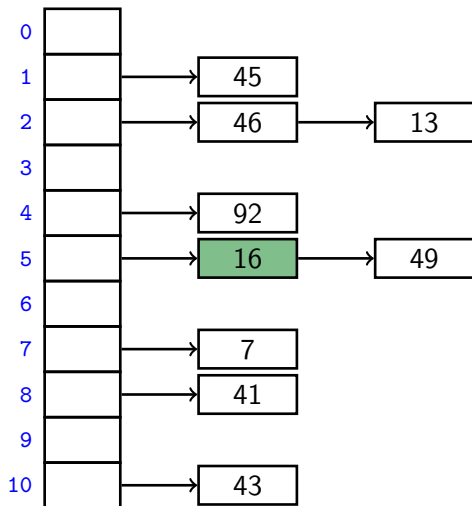


Chaining example

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

insert(16)

$$h(16) = 5$$

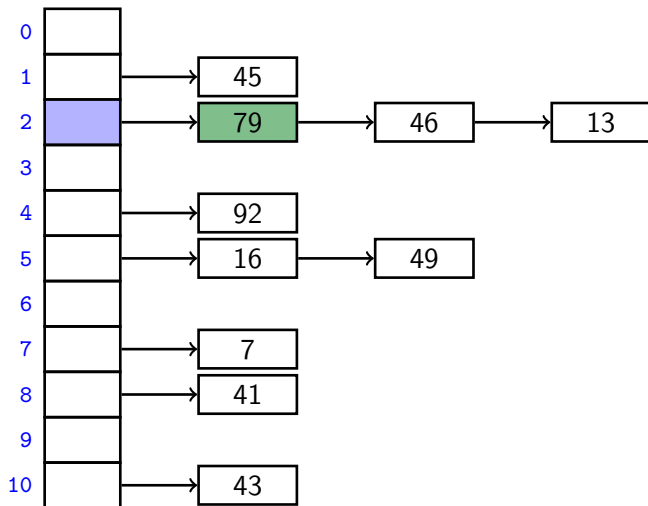


Chaining example

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

insert(79)

$$h(79) = 2$$



Complexity of chaining

Run-times: *insert* takes time $\Theta(1)$.

search and *delete* have run-time $\Theta(1 + \text{size of bucket } T[h(k)])$.

- The *average* bucket-size is $\frac{n}{M} =: \alpha$.
(α is also called the **load factor**.)
- However, this does not imply that the *average-case* cost of *search* and *delete* is $\Theta(1 + \alpha)$.
(If all keys hash to the same slot, then the average bucket-size is still α , but the operations take time $\Theta(n)$ on average.)
- **Uniform Hashing Assumption:** Each hash value is equally likely.
(This depends on the input and how we choose the function \rightsquigarrow later.)
- Under this assumption, each key collides is expected to collide with $\frac{n-1}{M}$ other keys and the average-case cost of *search* and *delete* is hence $\Theta(1 + \alpha)$.

Load factor and re-hashing

- For all collision resolution strategies, the run-time evaluation is done in terms of the *load factor* $\alpha = n/M$.
- 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-function(s) 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 $M \in \Theta(n)$ throughout, and the space is always $\Theta(n)$.

Summary: If we maintain $\alpha \in \Theta(1)$, then (under the uniform hashing assumption) the average cost for hashing with chaining is $O(1)$ and the space is $\Theta(n)$.

Outline

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

Open addressing

Main idea: Avoid the links needed for chaining by permitting only one item per slot, but allowing a key k to be in multiple slots.

search and *insert* follow a **probe sequence** of possible locations for key k : $\langle h(k, 0), h(k, 1), h(k, 2), \dots \rangle$ 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 NIL) and continue searching past deleted spots.

Open addressing

Main idea: Avoid the links needed for chaining by permitting only one item per slot, but allowing a key k to be in multiple slots.

search and *insert* follow a **probe sequence** of possible locations for key k : $\langle h(k, 0), h(k, 1), h(k, 2), \dots \rangle$ 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 NIL) 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

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

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

Linear probing example

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

insert(41)

$$h(41, 0) = 8$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear probing example

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

insert(84)

$$h(84, 0) = 7$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear probing example

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

insert(84)

$$h(84, 1) = 8$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear probing example

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

insert(84)

$$h(84, 2) = 9$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing example

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

insert(20)

$$h(20, 0) = 9$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing example

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

insert(20)

$$h(20, 1) = 10$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing example

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

insert(20)

$$h(20, 2) = 0$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing example

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

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

delete(43)

$$h(43, 0) = 10$$

Linear probing example

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

search(63)

$$h(63, 0) = 8$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

Linear probing example

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

search(63)

$$h(63, 1) = 9$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

Linear probing example

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

search(63)

$$h(63, 2) = 10$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

Linear probing example

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

search(63)

$$h(63, 3) = 0$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

Linear probing example

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

search(63)

$$h(63, 4) = 1$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

Linear probing example

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

search(63)

$$h(63, 5) = 2$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

Linear probing example

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

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	<i>deleted</i>

search(63)

$$h(63, 6) = 3$$

not found

Probe sequence operations

probe-sequence::insert($T, (k, v)$)

1. **for** ($j = 0; j < M; j++$)
2. **if** $T[h(k, j)]$ is NIL or “deleted”
3. $T[h(k, j)] = (k, v)$
4. **return** “success”
5. **return** “failure to insert” // need to re-hash

probe-sequence-search(T, k)

1. **for** ($j = 0; j < M; j++$)
2. **if** $T[h(k, j)]$ is NIL
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_0, h_1 .
- These hash functions should be *independent* in the sense that the random variables $P(h_0(k) = i)$ and $P(h_1(k) = j)$ 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
 - ▶ $kA - \lfloor kA \rfloor$ computes fractional part of kA , which is in $[0, 1)$
 - ▶ Multiply with M to get floating-point number in $[0, M)$
 - ▶ Round down to get integer in $\{0, \dots, M - 1\}$

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

Double Hashing

- Assume we have two hash independent functions h_0, h_1 .
- Assume further that $h_1(k) \neq 0$ and that $h_1(k)$ is relative prime with the table-size M for all keys k .
 - ▶ Choose M prime.
 - ▶ Modify standard hash-functions to ensure $h_1(k) \neq 0$
E.g. modified multiplication method: $h(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$
- **Double hashing**: open addressing with probe sequence

$$h(k, i) = h_0(k) + i \cdot h_1(k) \bmod M$$

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

Double hashing example

$$M = 11, \quad h_0(k) = k \bmod 11, \quad h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

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

Double hashing example

$$M = 11, \quad h_0(k) = k \bmod 11, \quad h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

insert(41)

$$h_0(41) = 8$$

$$h(41, 0) = 8$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double hashing example

$$M = 11, \quad h_0(k) = k \bmod 11, \quad h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

insert(194)

$$h_0(194) = 7$$

$$h(194, 0) = 7$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double hashing example

$$M = 11, \quad h_0(k) = k \bmod 11, \quad h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

insert(194)

$$h_0(194) = 7$$

$$h(194, 0) = 7$$

$$h_1(194) = 9$$

$$h(194, 1) = 5$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double hashing example

$$M = 11, \quad h_0(k) = k \bmod 11, \quad h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

insert(194)

$$h_0(194) = 7$$

$$h(194, 0) = 7$$

$$h_1(194) = 9$$

$$h(194, 1) = 5$$

$$h(194, 2) = 3$$

0	
1	45
2	13
3	194
4	92
5	49
6	
7	7
8	41
9	
10	43

Outline

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

Cuckoo hashing

We use two independent hash functions h_0, h_1 and two tables T_0, T_1 .

Main idea: An item with key k can *only* be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$.

- *search* and *delete* then take constant time.
- *insert always* initially puts a new item into $T_0[h_0(k)]$

If $T_0[h_0(k)]$ is occupied: “kick out” the other item, which we then attempt to re-insert into its alternate position $T_1[h_1(k)]$

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( $k, v$ )
1.    $i \leftarrow 0$ 
2.   do at most  $2n$  times:
3.       if  $T_i[h_i(k)]$  is NIL
4.            $T_i[h_i(k)] \leftarrow (k, v)$ 
5.           return "success"
6.       swap(( $k, v$ ),  $T_i[h_i(k)]$ )
7.        $i \leftarrow 1 - i$ 
8.   return "failure to insert"    // need to re-hash
```

After $2n$ iterations, there definitely was a loop in the “kicking out” sequence (why?)

In practice, one would stop the iterations much earlier already.

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

0	44
1	
2	
3	
4	59
5	
6	
7	
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(51)

$$i = 0$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$

0	44
1	
2	
3	
4	59
5	
6	
7	
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(51)

$$i = 0$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$

0	44
1	
2	
3	
4	59
5	
6	
7	51
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(95)

$$i = 0$$

$$k = 95$$

$$h_0(k) = 7$$

$$h_1(k) = 7$$

0	44
1	
2	
3	
4	59
5	
6	
7	51
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

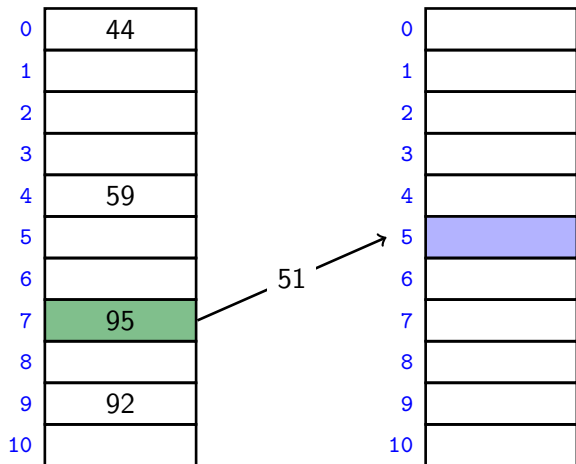
insert(95)

$$i = 1$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$



Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(95)

$$i = 1$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$

0	44
1	
2	
3	
4	59
5	
6	
7	95
8	
9	92
10	

0	
1	
2	
3	
4	
5	51
6	
7	
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(26)

$$i = 0$$

$$k = 26$$

$$h_0(k) = 4$$

$$h_1(k) = 0$$

0	44
1	
2	
3	
4	59
5	
6	
7	95
8	
9	92
10	

0	
1	
2	
3	
4	
5	51
6	
7	
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

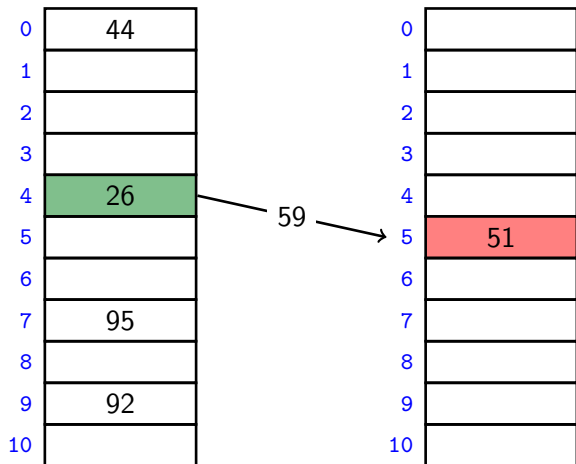
insert(26)

$$i = 1$$

$$k = 59$$

$$h_0(k) = 4$$

$$h_1(k) = 5$$



Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

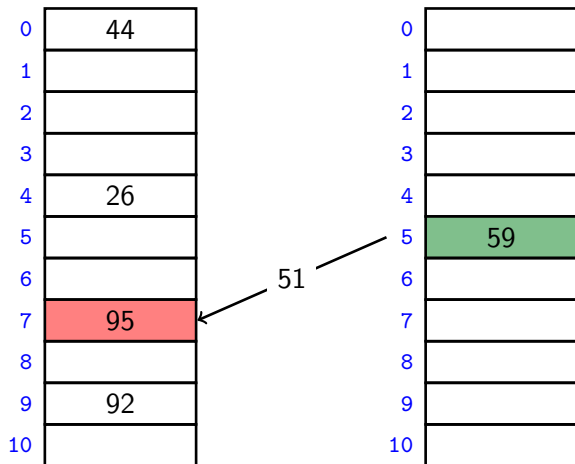
insert(26)

$$i = 0$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$



Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

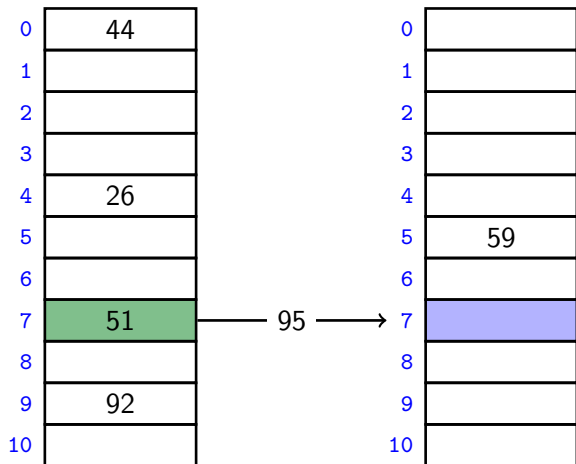
insert(26)

$$i = 1$$

$$k = 95$$

$$h_0(k) = 7$$

$$h_1(k) = 7$$



Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(26)

$$i = 1$$

$$k = 95$$

$$h_0(k) = 7$$

$$h_1(k) = 7$$

0	44
1	
2	
3	
4	26
5	
6	
7	51
8	
9	92
10	

0	
1	
2	
3	
4	
5	59
6	
7	95
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

search(59)

$$h_0(59) = 4$$

$$h_1(59) = 5$$

0	44
1	
2	
3	
4	26
5	
6	
7	51
8	
9	92
10	

0	
1	
2	
3	
4	
5	59
6	
7	95
8	
9	
10	

Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

delete(59)

$$h_0(59) = 4$$

$$h_1(59) = 7$$

0	44
1	
2	
3	
4	26
5	
6	
7	51
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	95
8	
9	
10	

Cuckoo hashing discussions

- The two hash-tables need not be of the same size.
- *Load factor* $\alpha = n / (\text{size of } T_0 + \text{size of } T_1)$
- One can argue: If the load factor α is small enough then insertion has $O(1)$ expected run-time.
- This crucially requires $\alpha < \frac{1}{2}$.

Cuckoo hashing discussions

- The two hash-tables need not be of the same size.
- *Load factor* $\alpha = n / (\text{size of } T_0 + \text{size of } T_1)$
- One can argue: If the load factor α is small enough then insertion has $O(1)$ expected run-time.
- This crucially requires $\alpha < \frac{1}{2}$.

There are many possible variations:

- The two hash-tables could be combined into one.
- Be more flexible when inserting: Always consider both possible positions.
- Use $k > 2$ allowed locations (i.e., k hash-functions).

Complexity of open addressing strategies

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

Cuckoo hashing requires $\alpha < 1/2$.

Avg.-case costs:	<i>search</i> (<i>unsuccessful</i>)	<i>insert</i>	<i>search</i> (<i>successful</i>)
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 (worst-case)	$\frac{\alpha}{(1 - 2\alpha)^2}$	1 (worst-case)

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

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 a hash-function that 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 function *must* do badly for some sequences of inputs:

- If the universe contains at least $M \cdot n$ keys, then there are n keys that all hash to the same value.
- For this set of keys, we have the worst case.

Idea: Randomization!

- When initializing or re-hashing, use as hash function

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

where $p > M$ is a prime number, and a, b are *random* numbers in $\{0, \dots, p - 1\}$, $a \neq 0$.

- 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 α is kept small enough.

We have again shifted the performance from “bad input” to “bad luck”.

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.

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

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 example, $h(\text{APPLE})$ is

$$\left(\left(\left(\left(\left(\left((65R+80) \bmod M \right) R+80 \right) \bmod M \right) R+76 \right) \bmod M \right) R+69 \right) \bmod M$$

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