Hashing

Nishant Mehta

Lectures 15 and 16

Warm-up puzzle

We have a deck of n distinct cards (where n is large) and repeatedly sample a card uniformly at random, with replacement. On average, how many cards do we need to draw before we see *some* card twice (that is, before we have repeated a card)?

(a) $\Theta(n^2)$

(b) $\Theta(n)$

(c) $\Theta\left(\sqrt{n}\right)$

(d) $\Theta(\log n)$

(e) $\Theta(1)$

Warm-up puzzle

We have a deck of n distinct cards (where n is large) and repeatedly sample a card uniformly at random, with replacement. On average, how many cards do we need to draw before we see *some* card twice (that is, before we have repeated a card)?

(a) $\Theta(n^2)$

(b) $\Theta(n)$

(c) $\Theta\left(\sqrt{n}\right)$

(d) $\Theta(\log n)$

(e) $\Theta(1)$

For some amusing reading, check out the ***birthday paradox*** in Section 5.4.1 of CLRS

Dictionary

A dictionary is a data structure that contains key-value pairs

- Keys should be unique
- Values can be anything and need not be unique

Dictionary - Operations

SEARCH - Need to specify key k

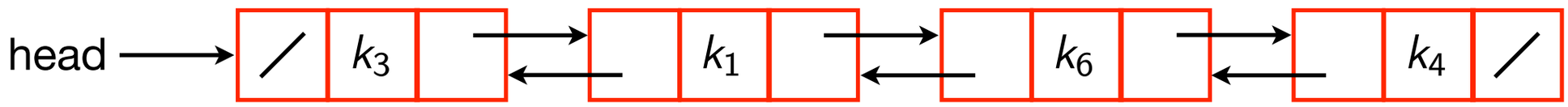
INSERT - Need to specify object x (obtain key via x.key)

DELETE - Need to specify object x

- We will see later why it is better to take as input x rather than x.key

Unordered list

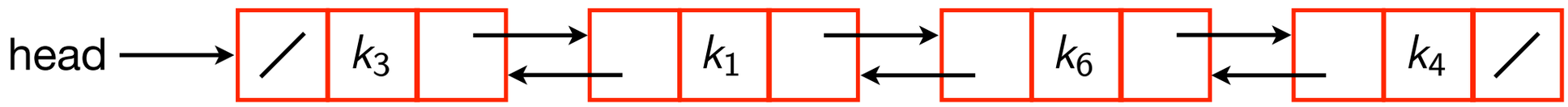
Suppose that we use an unordered list
(let's make it a doubly linked list)



Operation	Worst-case running time? (for n elements)
SEARCH(S, k)	
INSERT(S, x)	
DELETE(S, x)	

Unordered list

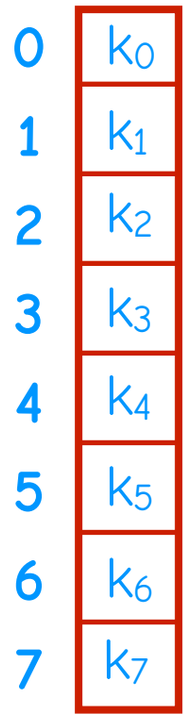
Suppose that we use an unordered list
(let's make it a doubly linked list)



Operation	Worst-case running time? (for n elements)
SEARCH(S, k)	$O(n)$
INSERT(S, x)	$O(1)$
DELETE(S, x)	$O(1)$

Ordered list

If we use an ordered list (in the form of an array), then searching is fast; other operations are slow



$k_0 < k_1 < k_2 < \ldots < k_7$

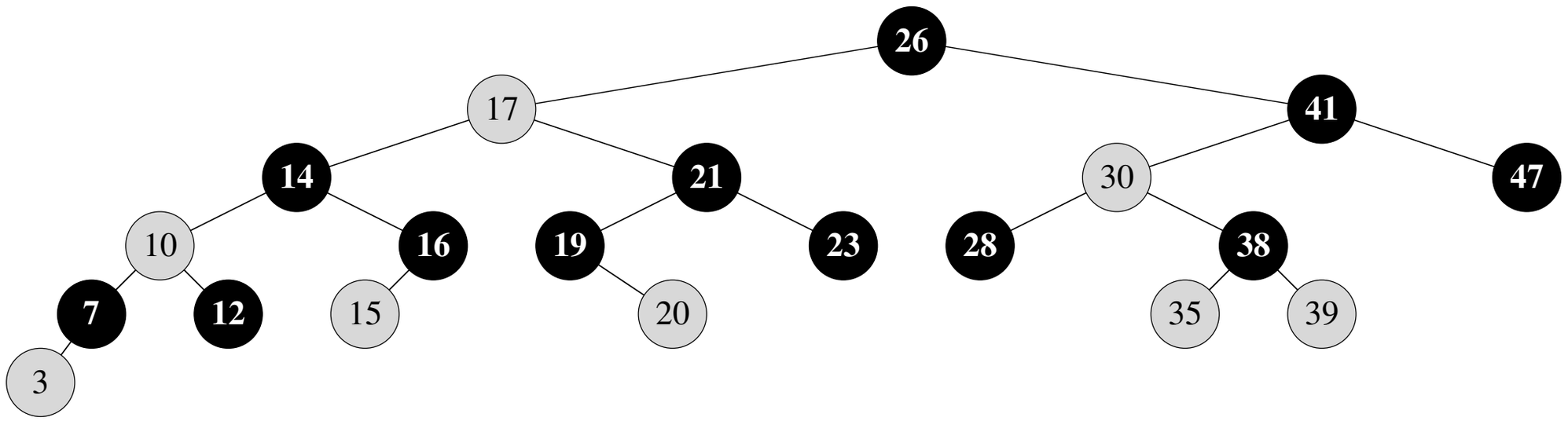
SEARCH(S, k) - binary search enables $O(\log n)$

INSERT(S, x) - $O(n)$

DELETE(S, x) - $O(n)$

Balanced binary search tree

Another strategy is to use a balanced binary search tree (e.g., red-black tree, AVL tree)



SEARCH(S, k) - $O(\log n)$

INSERT(S, x) - $O(\log n)$

DELETE(S, x) - $O(\log n)$

Direct-address table

Suppose the keys are in a universe $U = \{0, 1, \ldots, m-1\}$

In a direct-address table, we create an array T of size m (initialize all entries to NULL)

Element with key k is stored in $T[k]$

Worst-case running time?

SEARCH(S, k): return $T[k]$

INSERT(S, x): $T[x.\text{key}] = x$

DELETE(S, x): $T[x.\text{key}] = \text{NULL}$

Space complexity?

Direct-address table

Suppose the keys are in a universe $U = \{0, 1, \ldots, m-1\}$

In a direct-address table, we create an array T of size m (initialize all entries to NULL)

Element with key k is stored in $T[k]$

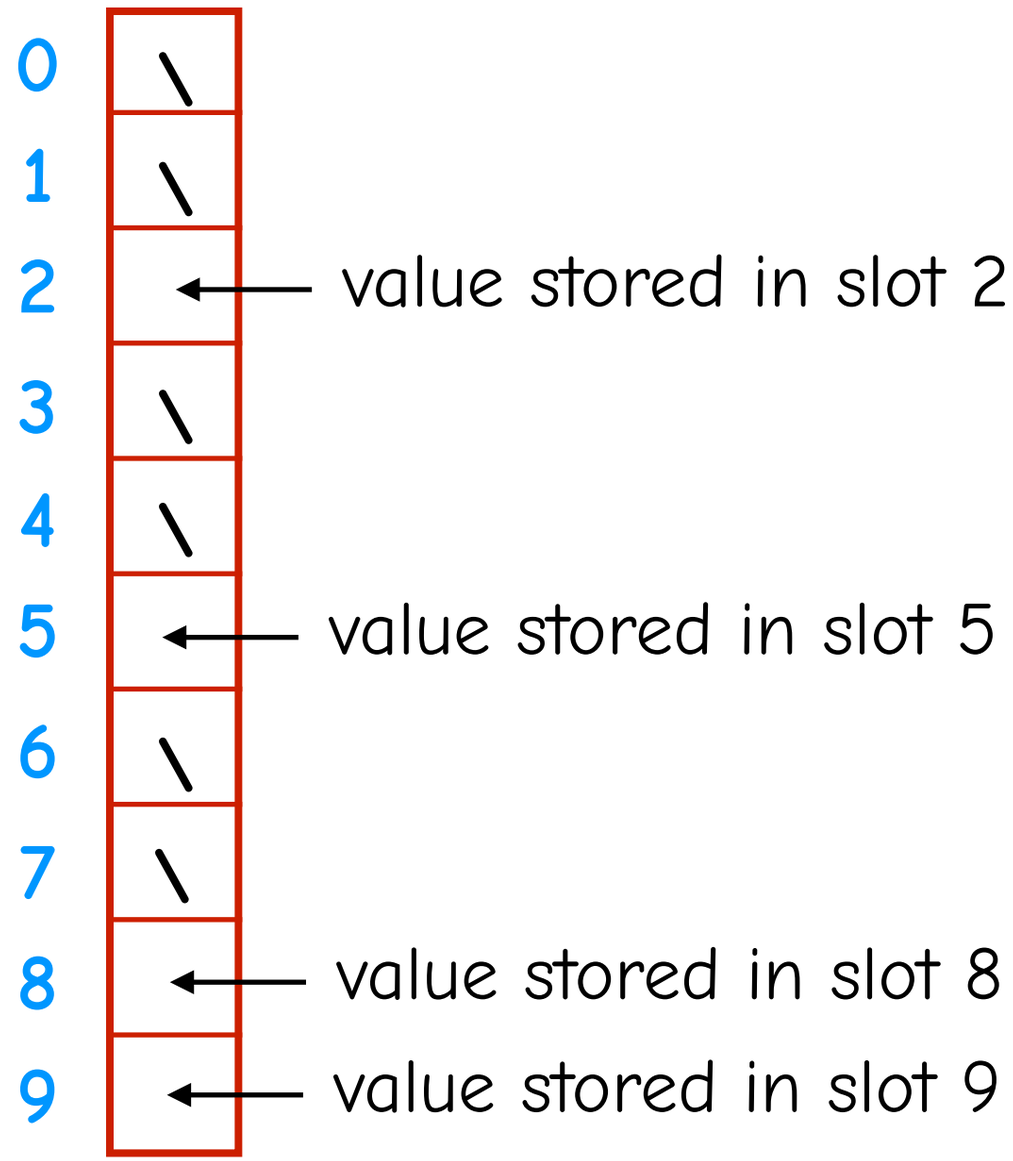
	Worst-case running time?
SEARCH(S, k): return $T[k]$	$O(1)$
INSERT(S, x): $T[x.\text{key}] = x$	$O(1)$
DELETE(S, x): $T[x.\text{key}] = \text{NULL}$	$O(1)$

Space complexity? $O(m)$

Direct-address table - Example 1

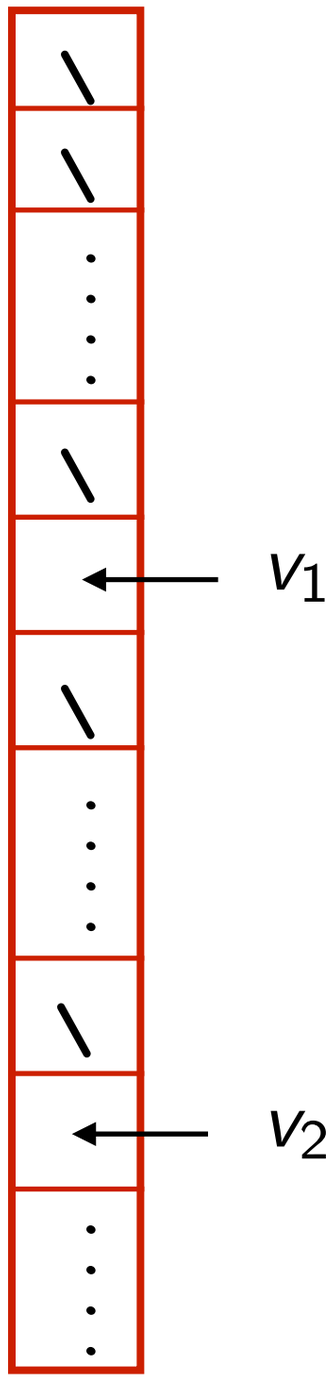
Universe $\{0, 1, \ldots, 9\}$

$n = 4$, with the 4 keys being: 2, 5, 8, 9



Direct-address table - Example 2

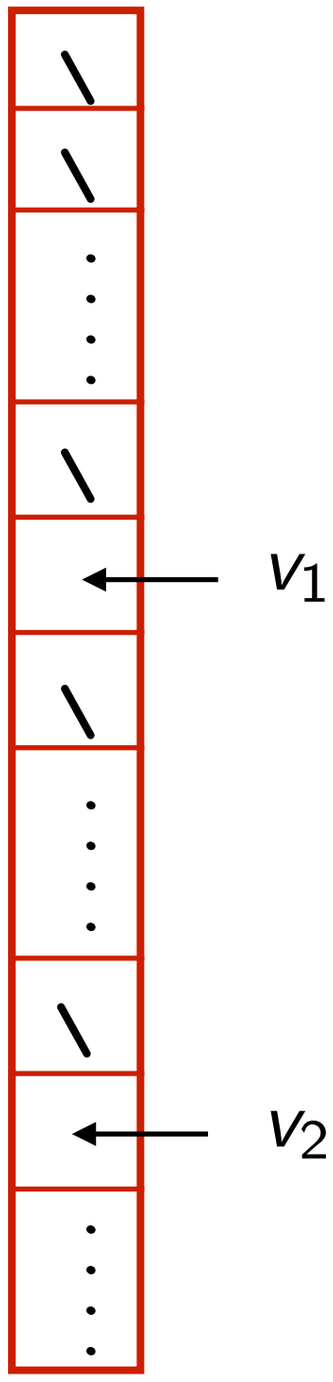
Universe $\{0, 1, \ldots, 2^{n-1}\}$ where we have only n keys



What fraction of space is being utilized?

Direct-address table - Example 2

Universe $\{0, 1, \ldots, 2^{n-1}\}$ where we have only n keys



What fraction of space is being utilized?

Storing n keys in 2^n space

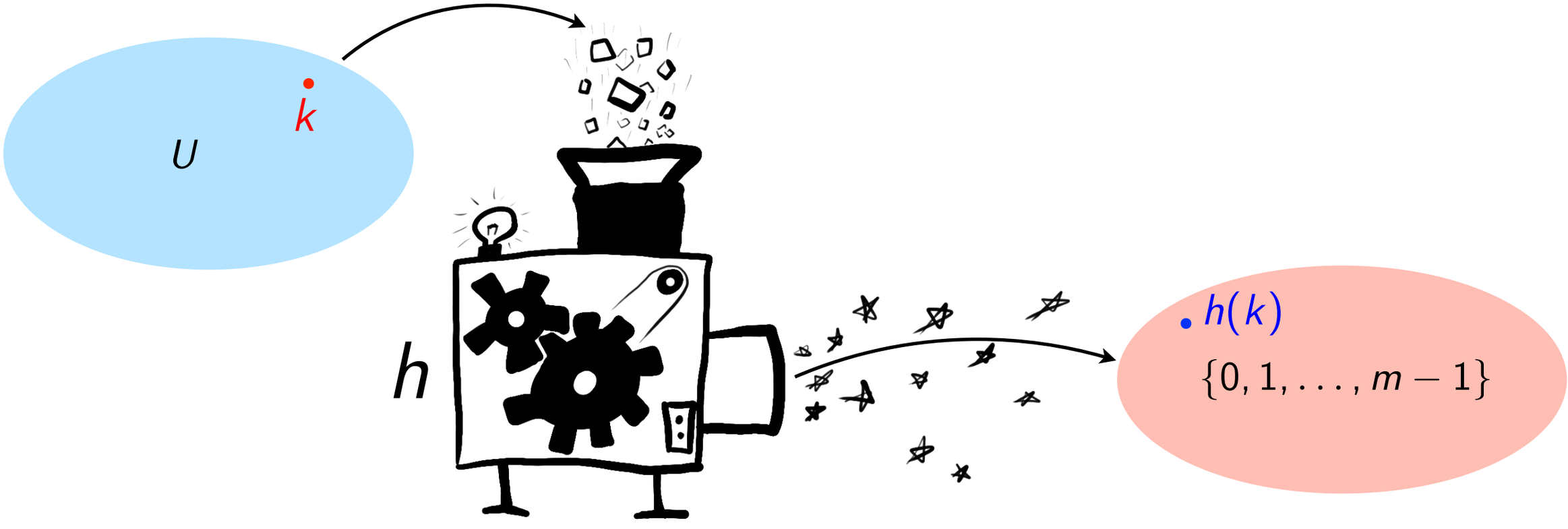
Utilization: $\frac{n}{2^n} \approx 0$

Hash tables

What if the universe is massive, but the set of keys that will actually be used is much smaller? ($n \ll |U|$)

Then a direct-address table is really wasteful.
Most entries of T will never be used!

Idea: Although $|U|$ is massive, we'll use a table of size m.
How? We use a ***hash function*** $h : U \to \{0, 1, \ldots, m-1\}$



Hash function and collisions

Let $h : U \to \{0, 1, \ldots, m-1\}$ be a hash function.

Given key k, we call $h(k)$ the ***hash value*** of key k

(not-so-smart) Example: $h(k) = k \bmod 10$

If two keys hash to the same slot, then we have a ***collision***

What if the key is not a natural number?

Suppose the key is a floating point number

- Easy to fix by rescaling

How to handle strings?

- Basic idea: interpret as number in given base and convert to decimal (more on this in a later lecture; stay tuned!)

Handling collisions

How can we handle collisions?

(1) Design a hash function which makes collisions as unlikely as possible (more on this soon)

Handling collisions

How can we handle collisions?

(1) Design a hash function which makes collisions as unlikely as possible (more on this soon)

OK... but what if a collision still happens? **How can we handle collisions?**

(2) Chaining - let each hash table slot store a linked list

(3) Open addressing - if the desired entry is already full, then try some other slots (using some fixed order; more on this next class)

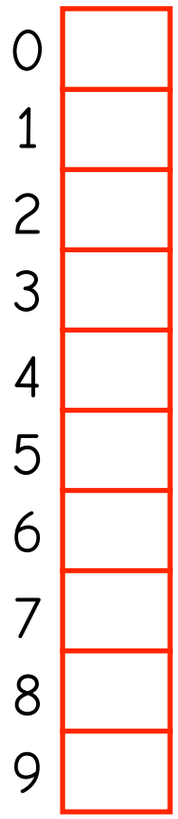
Chaining

In chaining, we store all elements that hash to the same slot j within a linked list $T[j]$

Example:

$h(k) = k \bmod 10$ (*again, not good hash function!*)

Insert these keys in this order: 17, 4, 7, 34, 1, 41, 21, 31



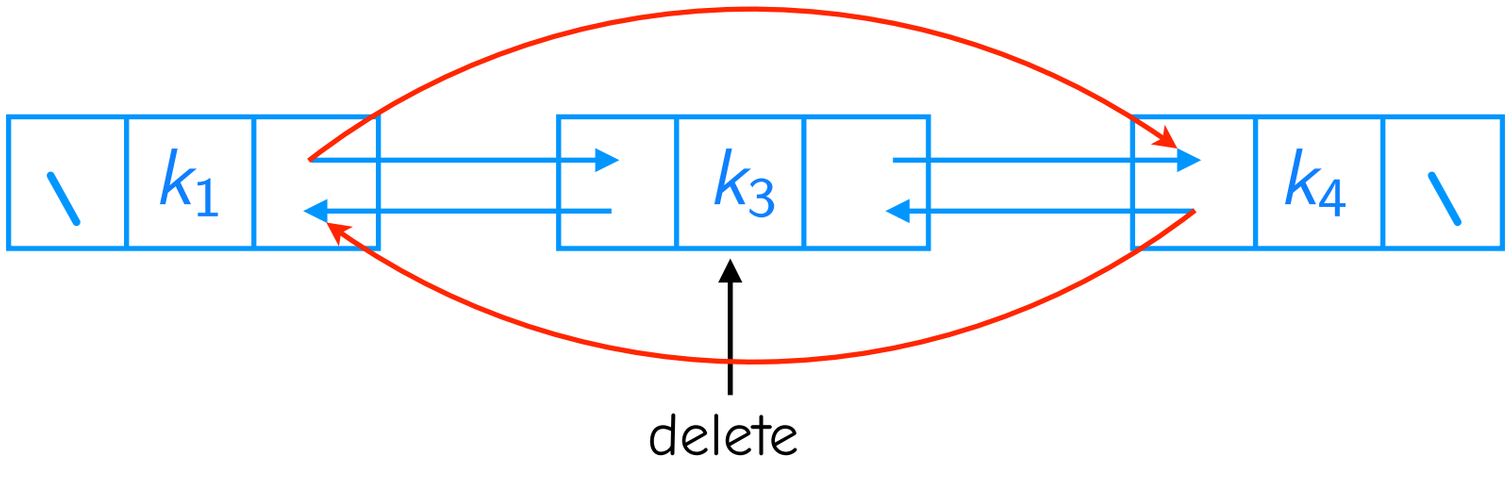
Operations

Worst-case running time?

SEARCH(S, k): Search list at $T[h(k)]$

INSERT(S, x): Insert x at the head of list $T[h(x.\text{key})]$

DELETE(S, x): Delete x from list $T[h(x.\text{key})]$



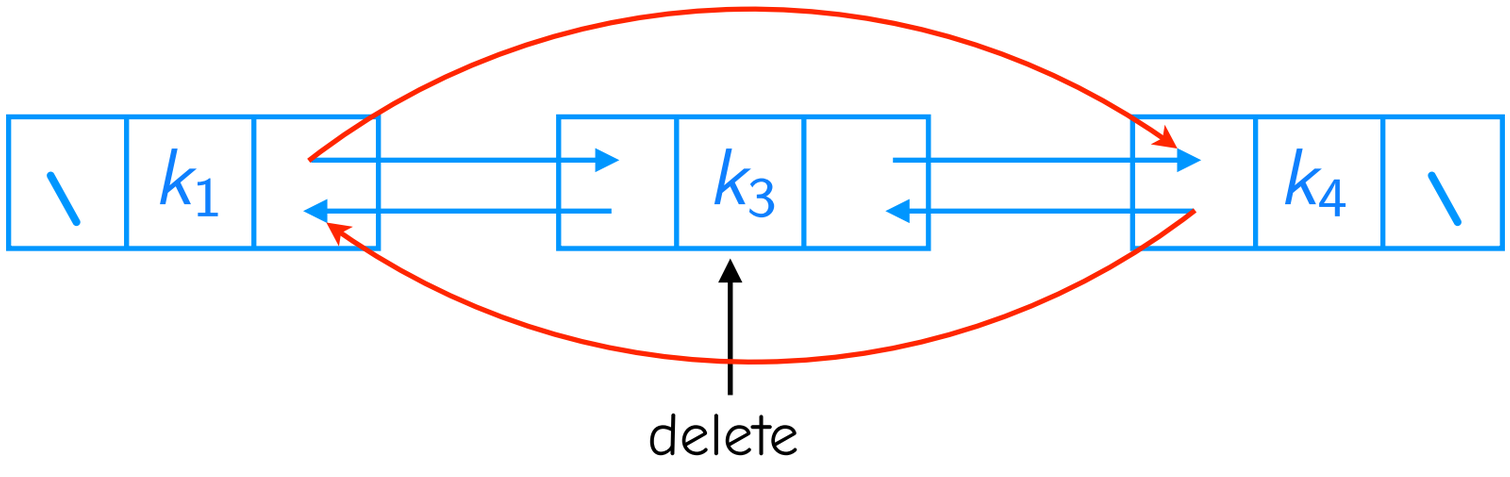
Operations

Worst-case running time?

SEARCH(S, k): Search list at $T[h(k)]$ $O(\text{length of } T[h(k)]) = O(n)$

INSERT(S, x): Insert x at the head of list $T[h(x.\text{key})]$ $O(1)$

DELETE(S, x): Delete x from list $T[h(x.\text{key})]$ $O(1)$



Load factor

Let T be a hash table of size m that stores n elements

The ***load factor*** α of T is the average length of a chain. This is simply the ratio of number of elements stored to number of slots. Therefore, $\alpha = n/m$

If we have a good hash function, the load is balanced (most chains have length α). In this case, the cost of each Search operation is close to α.

It can be challenging to find a good hash function which deterministically keeps most chains at length α. Instead, we will consider situations where a hash function is *randomly* selected such that, *on average*, any chain $T[j]$ has length α.

An ideal but useful assumption

Assumption: Simple uniform hashing

For any key k, its hash value $h(k)$ is drawn uniformly at random from $\{0, 1, \ldots, m-1\}$

Let n_j be the length of the chain $T[j]$

Suppose we insert n elements and the simple uniform hashing assumption holds. For any j in $\{0, 1, \ldots, m-1\}$, what is $\mathbb{E}[n_j]$?

Expected time for unsuccessful search

Proposition: The average-case cost of an unsuccessful search is $1 + \alpha$.

(Cost model: hash costs one, examining an element costs 1)

Expected time for successful search

Proposition: The average-case cost when searching for the i^{th} inserted key (after all n keys have been inserted) is:

$$2 + \frac{n - i}{m} \leq 2 + \alpha$$

Expected time for successful search

Proposition: The average-case cost when searching for the i^{th} inserted key (after all n keys have been inserted) is:

$$2 + \frac{n - i}{m} \leq 2 + \alpha$$

Corollary: The average-case cost when searching for an inserted key (also chosen uniformly at random from the set of n inserted keys) is:

$$2 + \frac{(n - 1)}{2m} \leq 2 + \frac{\alpha}{2}$$

How can we design a good hash function?

In some situations, simple uniform hashing assumption might be plausible

Otherwise, if we want to commit a single function, there are some heuristics which tend to work well in practice

- Division method ("modular hashing")
- Multiplication method

Division method

In the ***division method***, we simply divide by m and take the remainder:

$$h(k) = k \bmod m$$

Multiplication method

Multiplication Method:

(1) Select a constant A such that $0 < A < 1$

(2) Take fractional part of $k\,A$

(3) Multiply by m and truncate

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Multiplication method

Multiplication Method:

(1) Select a constant A such that $0 < A < 1$

(2) Take fractional part of $k\,A$

(3) Multiply by m and truncate

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

How to choose A?

$A = 1/\varphi = 0.61803398875\ldots$ tends to work well

(distributes nearby integers roughly uniformly in [0, 1])

Universal hashing

The previous methods *might* work well in practice, but we do not have rigorous guarantees for them

Instead, we consider an idea called ***universal hashing***

A ***universal hash family*** is a collection $\mathcal{H}$ of hash functions $h : U \to \{0, 1, \ldots, m-1\}$ such that, for any pair of keys j, k, at most $|\mathcal{H}|/m$ hash functions $h \in \mathcal{H}$ satisfy $h(j) = h(k)$

How can we use this?

> If we select h uniformly at random from $\mathcal{H}$, then for each pair of keys j, k, we have:
>
> $$\Pr\big(h(j) = h(k)\big) \leq 1/m$$

Average-case analysis for universal hashing

Proposition: Let h be drawn uniformly at random from a universal family of hash functions. Consider an arbitrary key k.

(i) If key k is not in the table, then the expected length of the list $T[h(k)]$ is at most α.

(ii) Otherwise, the expected length of the list is at most $1 + \alpha$.

Constructing a universal family of hash functions

Let p be prime number such that all keys k are in $\{0, 1, \ldots, p-1\}$.
For each a in $\{1, 2, \ldots, p-1\}$ and b in $\{0, 1, \ldots, p-1\}$,
define a hash function:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

Then $\mathcal{H} = \{h_{a,b} : 1 \leq a \leq p-1,\ 0 \leq b \leq p-1\}$ is a universal family of hash functions.

Proof sketch

Open addressing

Open addressing is another method for handling collisions.
Unlike chaining, each slot stores at most one key.
If we try to store a key in a slot but find that it is already occupied, we instead try some other slot, and if that slot is full, we try yet another slot, and so on.

This sequence of slots that we try when we are *probing* for an unoccupied slot is called a ***probe sequence***.

A first probe sequence:

$$\underbrace{h(k), h(k)+1, h(k)+2, \ldots, h(k)+(m-1)}_{\text{all mod } m}$$

Linear probing uses this probe sequence

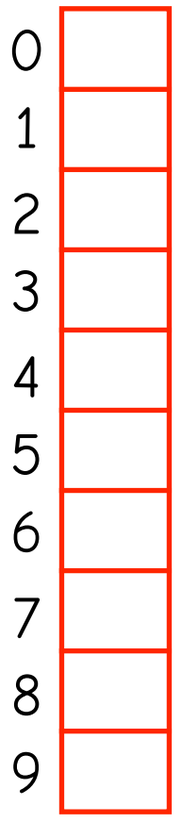
Linear probing

Using the hash function $h(k) = k \bmod 10$,
insert: 35, 21, 16, 45, 31, 8

How to do SEARCH? Use probe sequence and stop when we have either found the key or arrived at an unoccupied slot.

DELETE can cause trouble for Search. Why?

Solution: Upon deletion, mark slot with special value **DELETED**



Linear probing

- Doesn't work well in practice
  - Suffers from ***primary clustering*** problem
- Limited number of probe sequences (only m of them)

Quadratic probing

In quadratic probing, we use a somewhat more sophisticated probe sequence. For carefully selected positive constants c_1 and c_2, the probe sequence is

$$(h(k) + c_1 i + c_2 i^2) \bmod m \quad \text{for} \quad i = 0, 1, \ldots, m-1$$

Advantages: Avoids primary clustering problem

Disadvantages: Experiences ***secondary clustering*** problem

Still only m probe sequences

Double hashing

Let h_1 and h_2 be auxiliary hash functions

Double hashing uses the probe sequence:

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

General way of specifying elements in probe sequence

no common factors

$h_2(k)$ must be relatively prime to m in order for the whole table to be searched. How can this be achieved?

Average-case analysis of open addressing

Can we provide average-case guarantees for open addressing?

- **Yes!** Under a certain assumption

Assumption of ***uniform hashing*** - for each key, the probe sequence $h(k, i)$ is chosen uniformly at random from the set of all possible permutations of $(0, 1, \ldots, m-1)$.
(this is not realistic, but we might approximate it in practice using, e.g., double hashing)

Proposition: Given a hash table with load factor $\alpha = n/m < 1$, under uniform hashing the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Average-case analysis of open addressing

Proposition: Given a hash table with load factor $\alpha = n/m < 1$, under uniform hashing the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Proof: