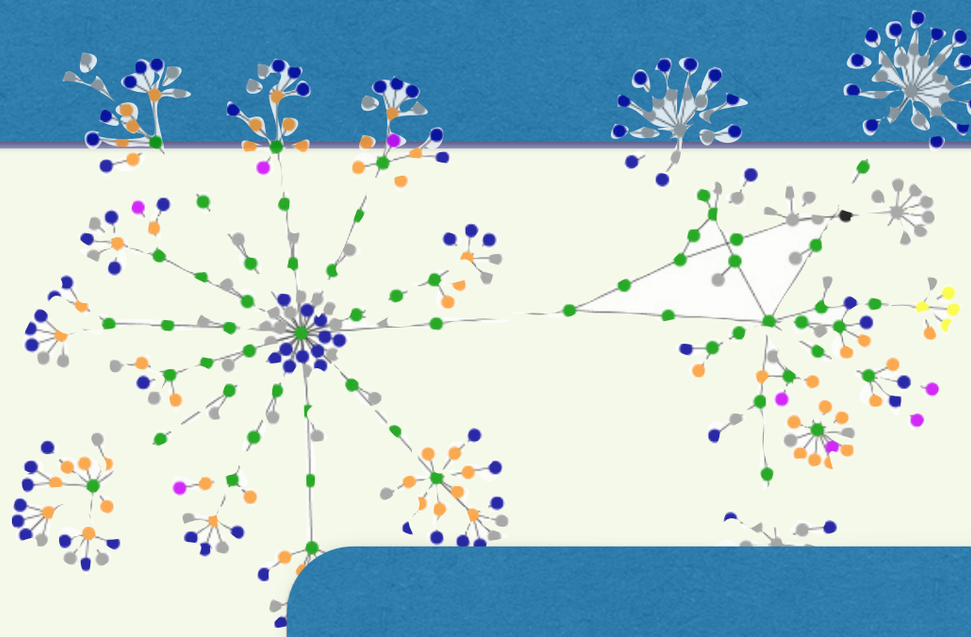
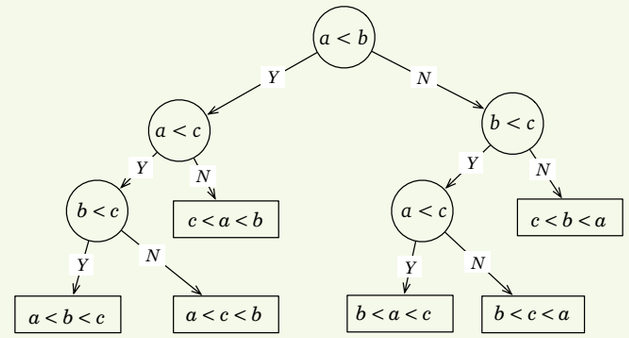




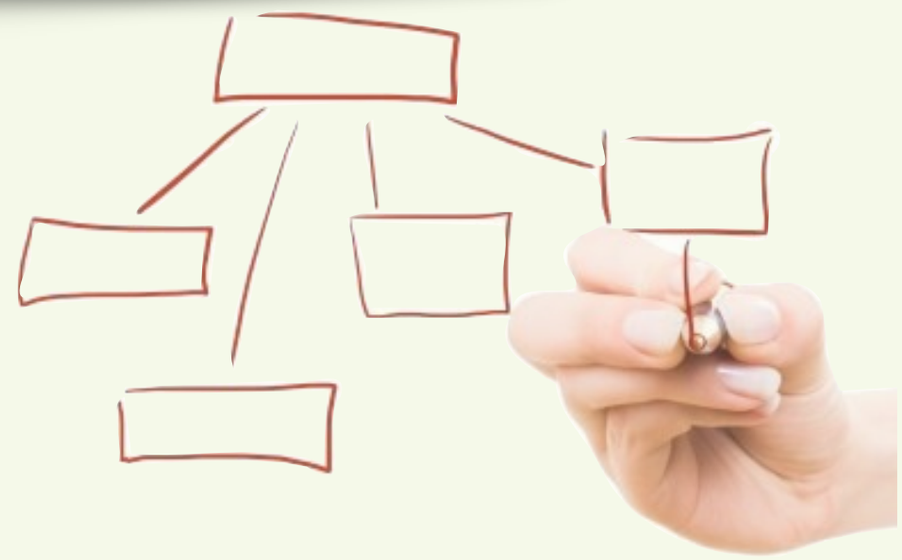
COURSE
"Técnicas Matemáticas para Big Data"
University of Aveiro

Algorithm 2.1: INSERTION-SORT(A)

```
1  for j ← 2 to A.size do
2      key ← A[j]
       // Insert A[j] into the sorted sequence A[1 . . j − 1]
3      i ← j − 1
4      while i > 0 and A[i] > key do
5          A[i + 1] ← A[i]
6          i−−
7      A[i + 1] ← key
```

Master in Data Science

Other Masters (optional)

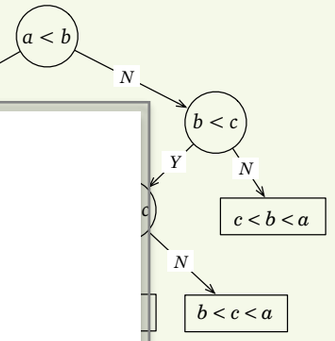


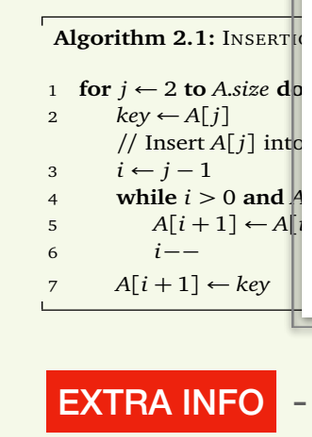
EXTRA INFO - this label will appear in slides with complementary information

SUMMARY: Frequent items in a data stream:

[T] Formal methods and concepts for streaming; Quality of an algorithm's answer; Averages and standard deviations over a stream; Quantile outliers and z-scores outliers; Finding frequent items deterministically; Frequency estimation by the Misra–Gries algorithm.

[P] Implementation of naive mean, std, and z-score outlier algorithms; Test and implementation of the MG-algorithm; Probabilistic top-10 elements





EXTRA INFO - this label will appear in slides with complementary information

1.1.2 The vanilla streaming model

In this V (i.e. 'velocity'), we are concerned with algorithms that compute some function of a massively long input stream $\sigma = \langle a_1, a_2, ..., a_m \rangle$, where the elements of the sequence (called tokens) are drawn from the universe $[n] := \{1, 2, ..., n\}$. Notice that to have a finite sequence is the same as to have an infinite stream and process the stream up to the m element.

QUESTION: How difficult is to have a streaming model where the universe is not $[n]$ but $\mathbb{R}$?

GOAL: The central goal will be to process the input stream σ using a small amount of space bits ("spaceBits"). We want the spaceBits to be sublinear in both m and n, i.e.

- $spaceBits = o(\min\{m, n\})$;
- $spaceBits = polylog(g(m, n))$ for some function g;
- $spaceBits = O(\log(m) + \log(n))$;

where $f(m, n) = polylog(g(m, n)$ if exists $c > 1$ such that $f(m, n) = O(\log(g(m, n))^c)$ and g is a given function.

Python Structure for Testing Streaming Algorithms (*template1.ipynb*)

```python
import .... 

## Your test stream
stream = [...]

## Implement the algorithm in here
class BaseAlg:
    stream = []
    results = {}
    x = 0
    ## your variables
    ...

    def alg(self):
        ...

    def verify(self):
        ...

## Do not change
class StreamAlg(BaseAlg):
    def __init__(self, stream):
        self.stream = stream
        self.exec()
        self.verify()

    def exec(self):
        for v in self.stream:
            self.x = v
            self.alg()
        print(self.results)

SA = StreamAlg(stream)
```

Add the packages you need.

Python Structure for Testing Streaming Algorithms (*template1.ipynb*)

```python
import .... 

## Your test stream
stream = [...]

## Implement the algorithm in here
class BaseAlg:
    stream = []
    results = {}
    x = 0
    ## your variables
    ...

    def alg(self):
        ...

    def verify(self):
        ...

## Do not change
class StreamAlg(BaseAlg):
    def __init__(self, stream):
        self.stream = stream
        self.exec()
        self.verify()

    def exec(self):
        for v in self.stream:
            self.x = v
            self.alg()
        print(self.results)

SA = StreamAlg(stream)
```

Add the packages you need.

Write the stream values as a list.

Python Structure for Testing Streaming Algorithms (*template1.ipynb*)

```python
import .... 

## Your test stream
stream = [...]

## Implement the algorithm in here
class BaseAlg:
    stream = []
    results = {}
    x = 0
    ## your variables
    ...

    def alg(self):
        ...

    def verify(self):
        ...

## Do not change
class StreamAlg(BaseAlg):
    def __init__(self, stream):
        self.stream = stream
        self.exec()
        self.verify()

    def exec(self):
        for v in self.stream:
            self.x = v
            self.alg()
        print(self.results)

SA = StreamAlg(stream)
```

Add the packages you need.

Write the stream values as a list.

Initialise the algorithm variables

Python Structure for Testing Streaming Algorithms (*template1.ipynb*)

```python
import .... 

## Your test stream
stream = [...]

## Implement the algorithm in here
class BaseAlg:
    stream = []
    results = {}
    x = 0
    ## your variables
    ...

    def alg(self):
        ...

    def verify(self):
        ...

## Do not change
class StreamAlg(BaseAlg):
    def __init__(self, stream):
        self.stream = stream
        self.exec()
        self.verify()

    def exec(self):
        for v in self.stream:
            self.x = v
            self.alg()
        print(self.results)

SA = StreamAlg(stream)
```

Add the packages you need.

Write the stream values as a list.

Initialise the algorithm variables

The stream algorithm for each x

Python Structure for Testing Streaming Algorithms (*template1.ipynb*)

```python
import .... 

## Your test stream
stream = [...]

## Implement the algorithm in here
class BaseAlg:
    stream = []
    results = {}
    x = 0
    ## your variables
    ...

    def alg(self):
        ...

    def verify(self):
        ...

## Do not change
class StreamAlg(BaseAlg):
    def __init__(self, stream):
        self.stream = stream
        self.exec()
        self.verify()

    def exec(self):
        for v in self.stream:
            self.x = v
            self.alg()
        print(self.results)

SA = StreamAlg(stream)
```

Add the packages you need.

Write the stream values as a list.

Initialise the algorithm variables

The stream algorithm for each x

An exact algorithm for comparison

Python Structure for Testing Streaming Algorithms (*template1.ipynb*)

```python
import .... 

## Your test stream
stream = [...]

## Implement the algorithm in here
class BaseAlg:
    stream = []
    results = {}
    x = 0
    ## your variables
    ...

    def alg(self):
        ...

    def verify(self):
        ...

## Do not change
class StreamAlg(BaseAlg):
    def __init__(self, stream):
        self.stream = stream
        self.exec()
        self.verify()

    def exec(self):
        for v in self.stream:
            self.x = v
            self.alg()
        print(self.results)

SA = StreamAlg(stream)
```

Add the packages you need.

Write the stream values as a list.

Initialise the algorithm variables

The stream algorithm for each x

An exact algorithm for comparison

Do not change!
Simulate the streaming input data.

Do exerc. S03.2

Simple Streaming Algorithm (z-score)

Mean and Std

- Stream: $\sigma = \langle a_1, a_2, ..., a_m \rangle$

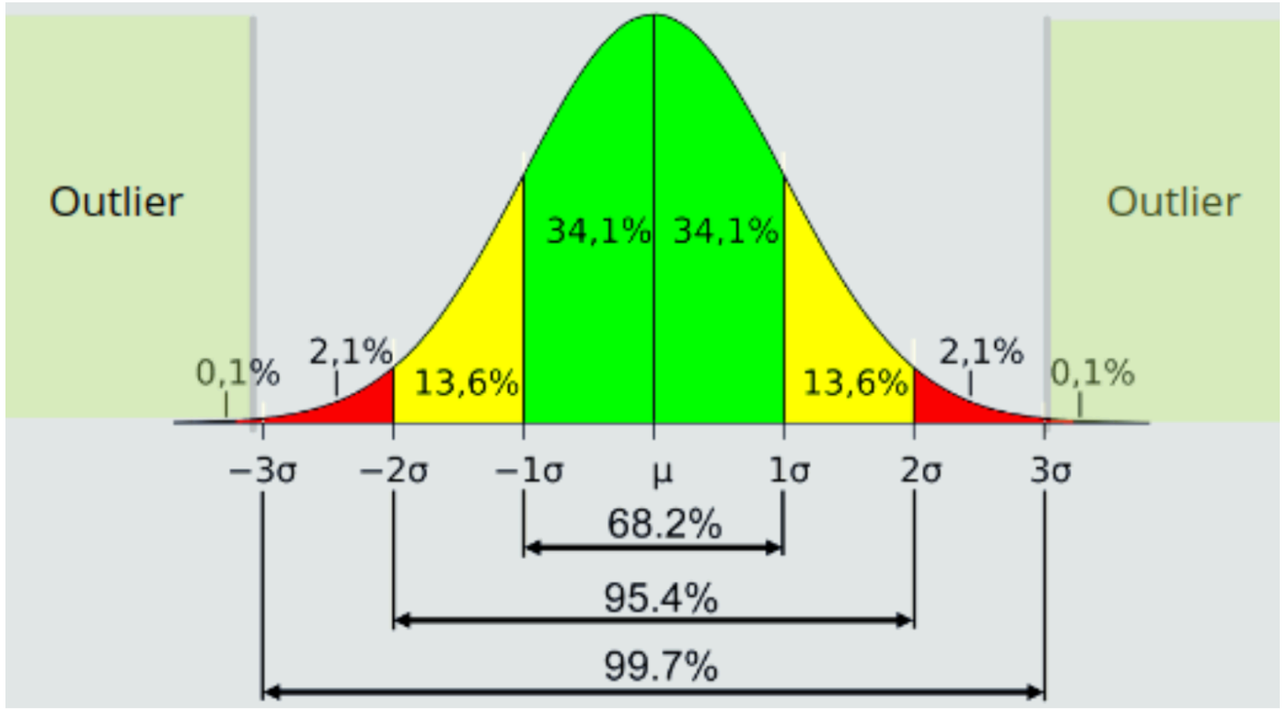
- Current Stream Value: x

- Algorithm Initialization:
  n = 0; ms = 0; ds = 0

- Algorithm (given x):

```
try:
    n += 1
    ms += x
    m = ms/n
    ds += (m-x)**2
except:
    n = 2
    ms = m + x
    m = ms / 2
    ds = (m - x)**2
```

z-score outliers



Mean and Std

- Stream: $\sigma = \langle a_1, a_2, ..., a_m \rangle$

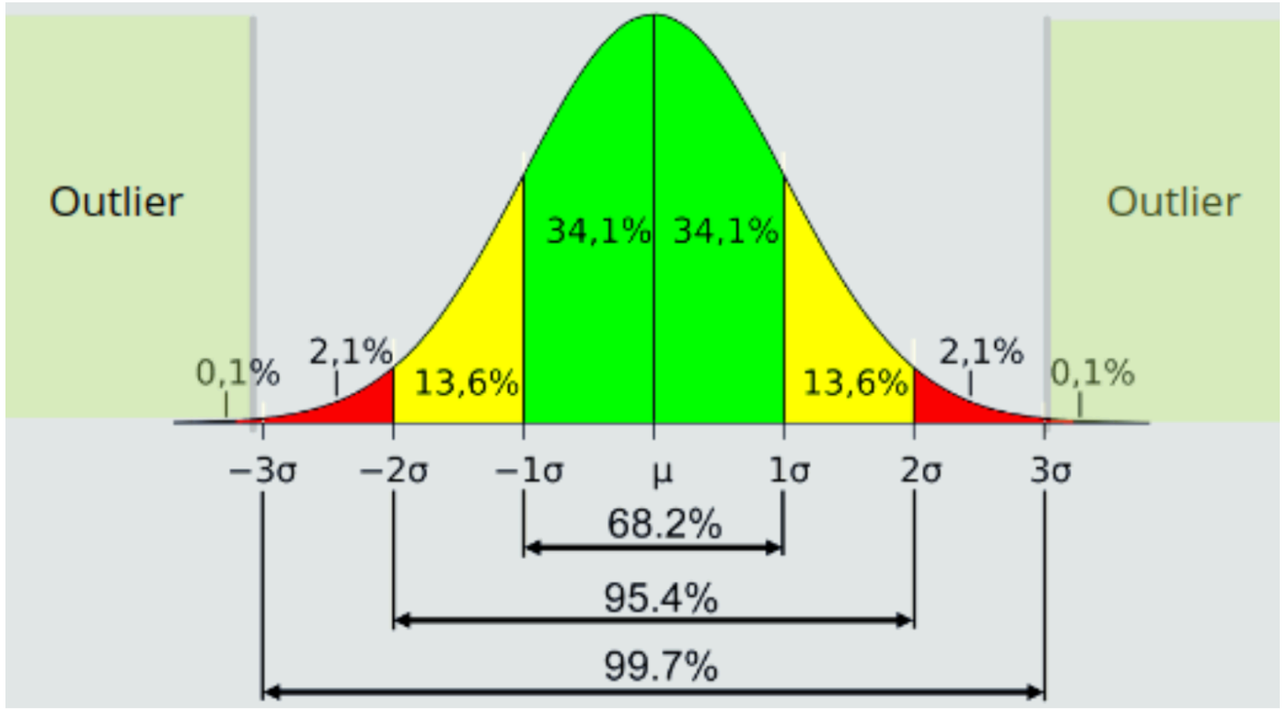
- Current Stream Value: x

- Algorithm Initialization:
  n = 0; ms = 0; ds = 0

- Algorithm (given x):

```
try:
    n += 1
    ms += x
    m = ms/n
    ds += (m-x)**2
except:
    n = 2
    ms = m + x
    m = ms / 2
    ds = (m - x)**2
```

z-score outliers



Welford's online algorithm

$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)$$

$$\sigma_n^2 = \frac{M_{2,n}}{n}$$

Mean and Std

- Stream: $\sigma = \langle a_1, a_2, ..., a_m \rangle$

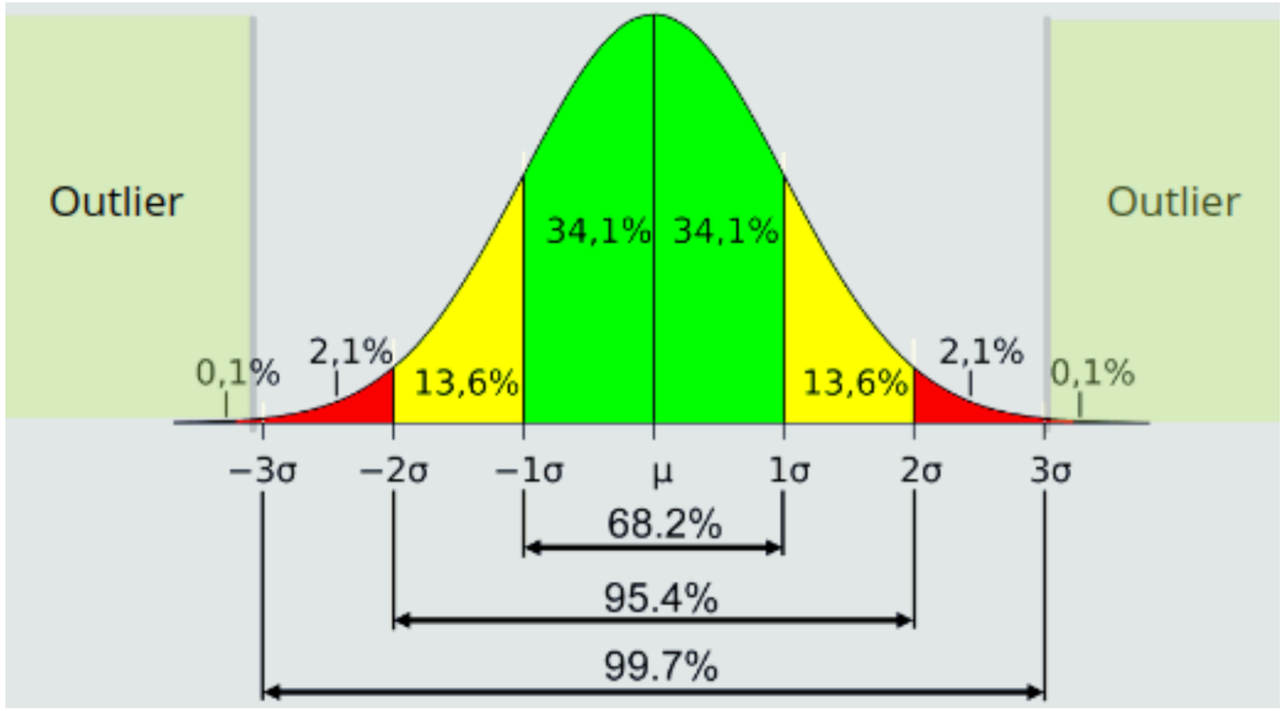
- Current Stream Value: x

- Algorithm Initialization:
  n = 0; ms = 0; ds = 0

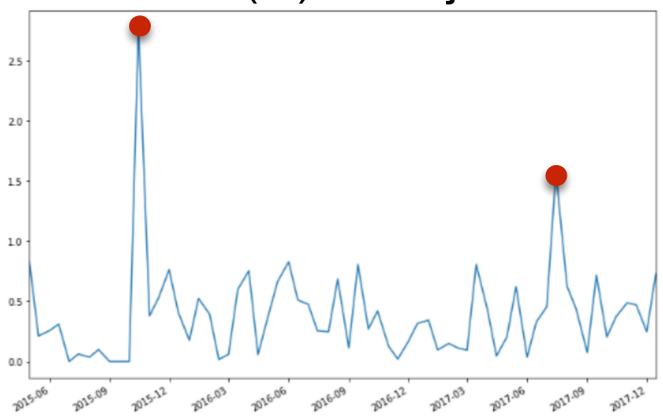
- Algorithm (given x):

```
try:
    n += 1
    ms += x
    m = ms/n
    ds += (m-x)**2
except:
    n = 2
    ms = m + x
    m = ms / 2
    ds = (m - x)**2
```

z-score outliers



Problem (a) - adjust



Mean and Std

- Stream: $\sigma = \langle a_1, a_2, ..., a_m \rangle$

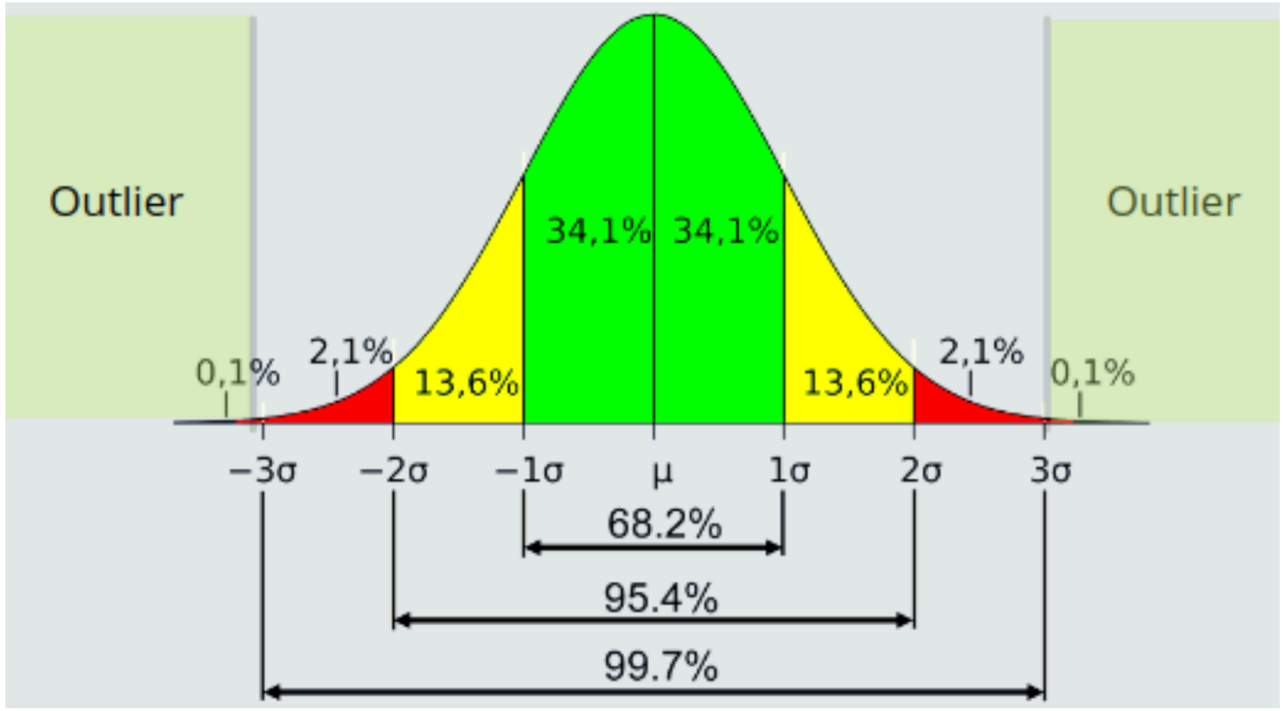
- Current Stream Value: x

- Algorithm Initialization:
  n = 0; ms = 0; ds = 0

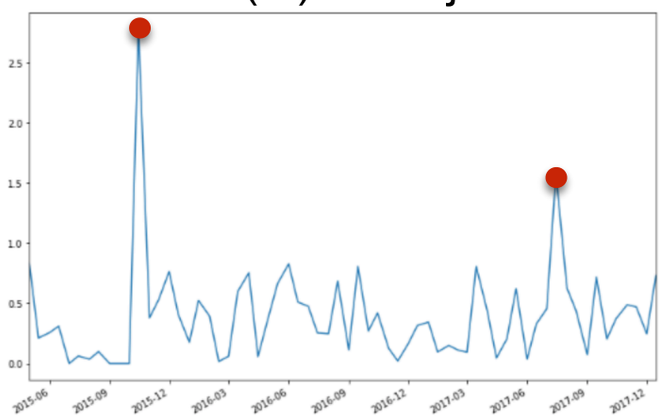
- Algorithm (given x):

```
try:
    n += 1
    ms += x
    m = ms/n
    ds += (m-x)**2
except:
    n = 2
    ms = m + x
    m = ms / 2
    ds = (m - x)**2
```

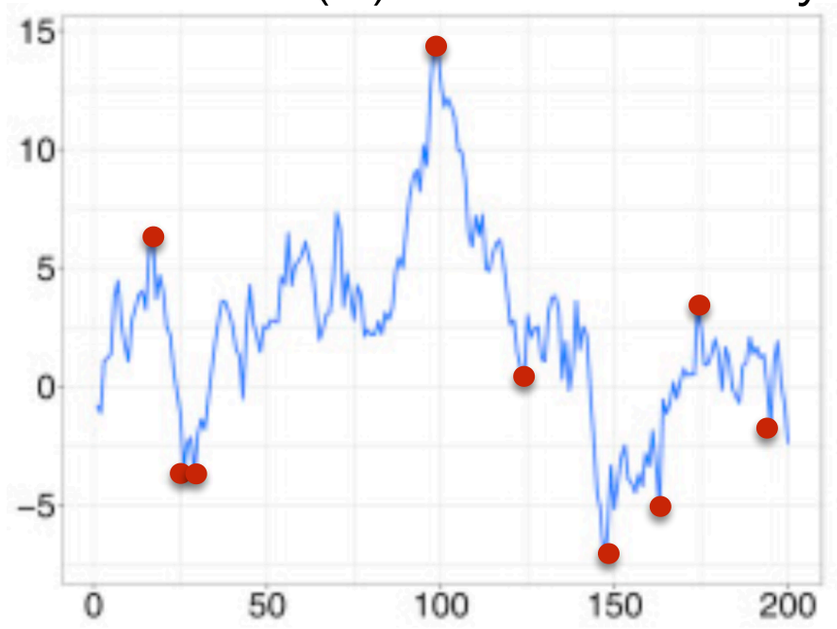
z-score outliers



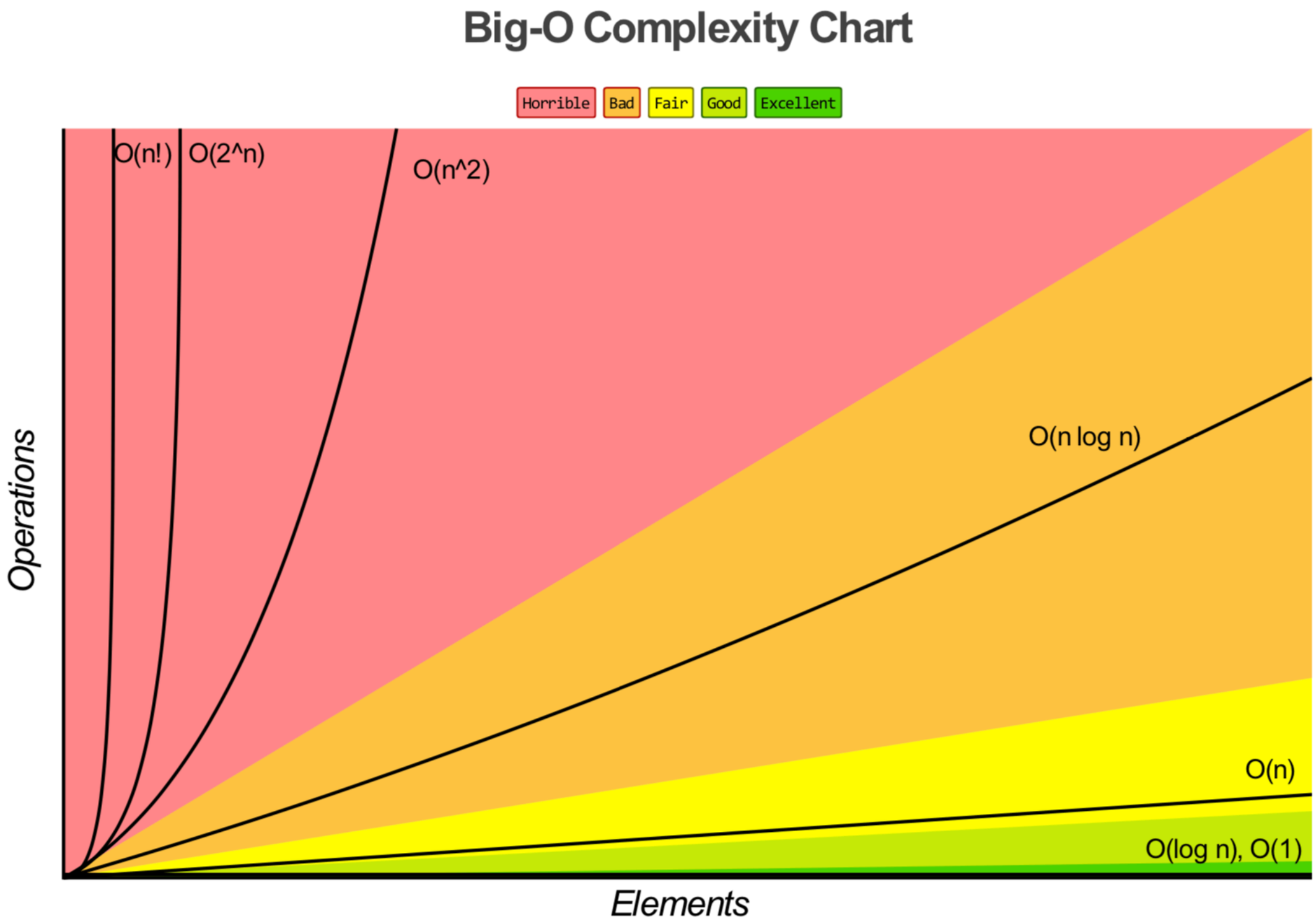
Problem (a) - adjust



Problem (b) - nonstationary



Streaming Algorithm based on Hash Functions


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Open Data Structures

(basic/standard)

An open content textbook

About «

Open Data Structures covers the implementation and analysis of data structures for sequences (lists), queues, priority queues, unordered dictionaries, ordered dictionaries, and graphs.

Data structures presented in the book include stacks, queues, deques, and lists implemented as arrays and linked-lists; space-efficient implementations of lists; skip lists; hash tables and hash codes; binary search trees including treaps, scapegoat trees, and red-black trees; integer searching structures including binary tries, x-fast tries, and y-fast tries; heaps, including implicit binary heaps and randomized meldable heaps; graphs, including adjacency matrix and ajacency list representations; and B-trees.

The data structures in this book are all fast, practical, and have provably good running times. All data structures are rigorously analyzed and implemented in Java and C++. The Java implementations implement the corresponding interfaces in the Java Collections Framework.

The book and accompanying source code are free (*libre* and *gratis*) and are released under a Creative Commons Attribution License. Users are free to copy, distribute, use, and adapt the text and source code, even commercially. The book's LaTeX sources, Java/C++/Python sources, and build scripts are available through github.

pseudocode edition (free)

- html pdf python sources screen pdf

java edition (free) get the book and sources

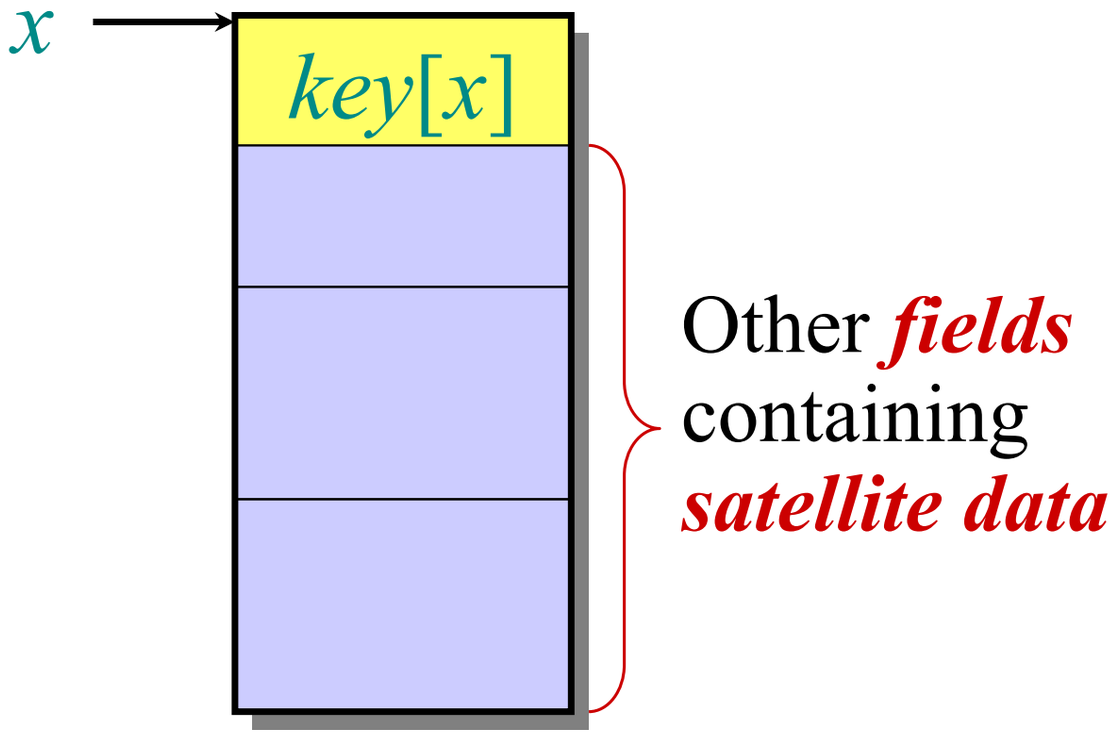
- html pdf java sources screen pdf

c++ edition (free) get the beta version

- html pdf c++ sources screen pdf

Symbol-table problem

Symbol table S holding n ***records***:



Hashing I

- Direct-access tables
- Resolving collisions by chaining
- Choosing hash functions
- Open addressing

Operations on S:

- INSERT(S, x)
- DELETE(S, x)
- SEARCH(S, k)

How should the data structure S be organized?

Direct-access table

IDEA: Suppose that the keys are drawn from the set $U \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct. Set up an array $T[0 \,..\, m-1]$:

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

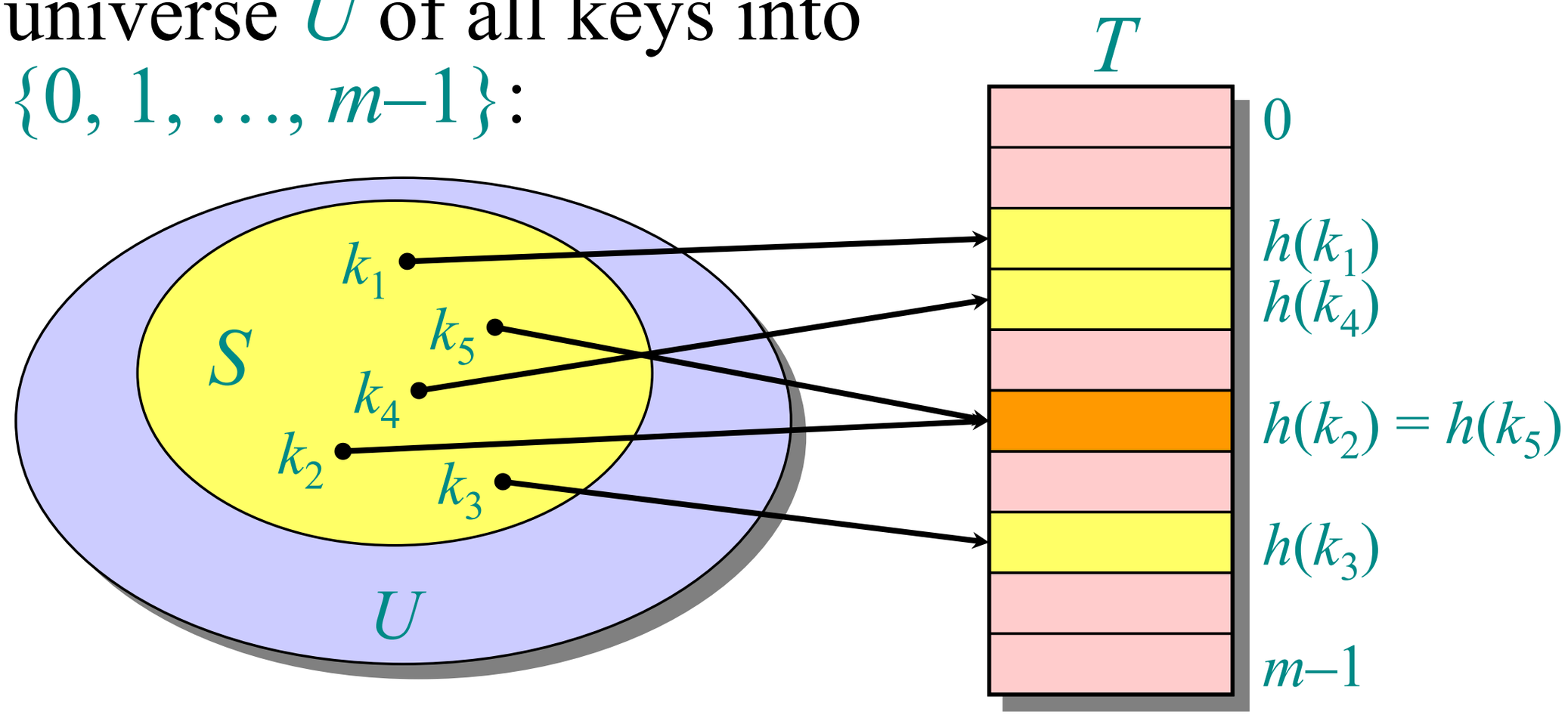
Then, operations take $\Theta(1)$ time.

Problem: The range of keys can be large:

- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- character strings (even larger!).

Hash functions

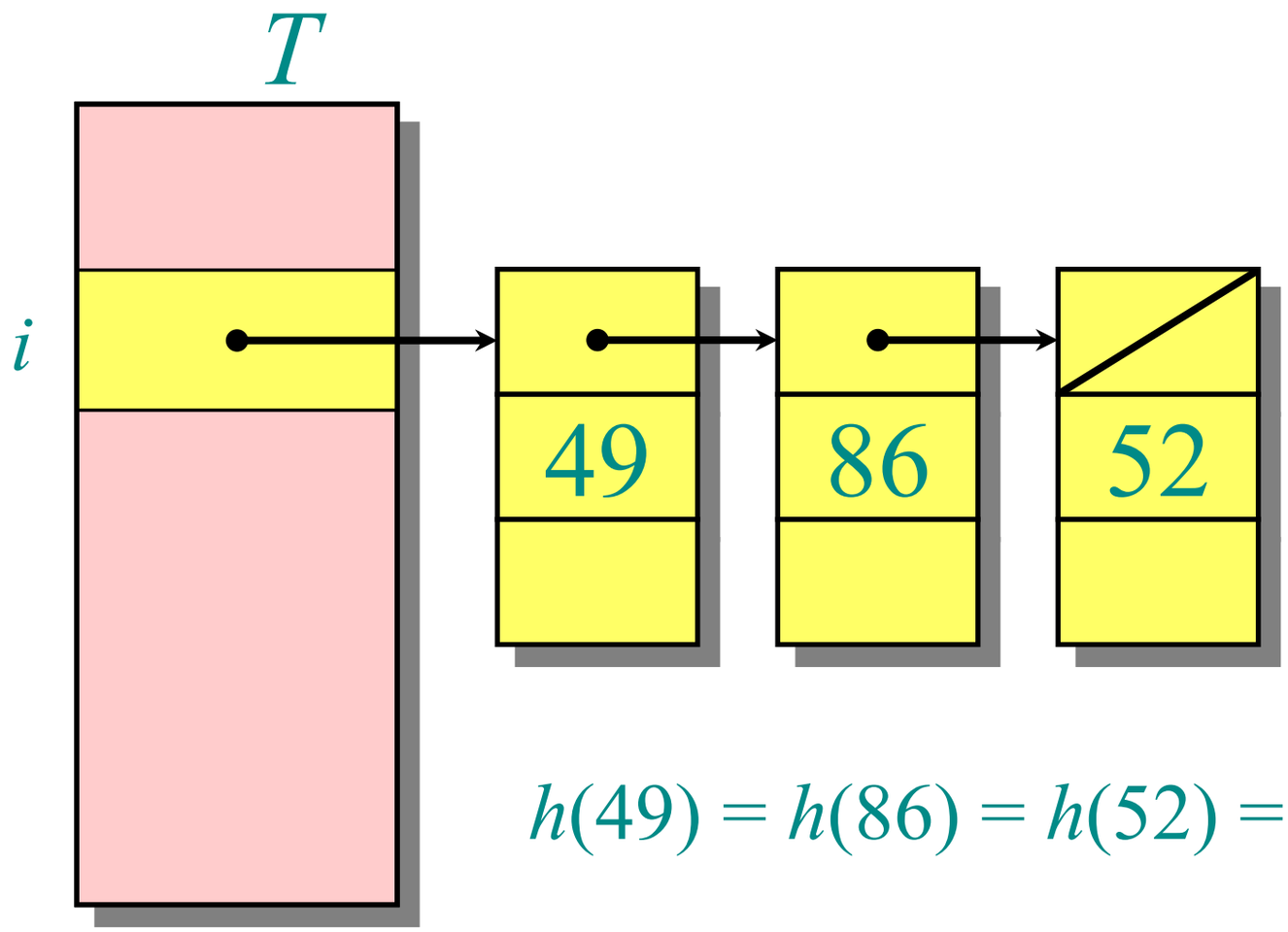
Solution: Use a ***hash function*** h to map the universe U of all keys into $\{0, 1, \ldots, m–1\}$:



When a record to be inserted maps to an already occupied slot in T, a ***collision*** occurs.

Resolving collisions by chaining

- Link records in the same slot into a list.



$h(49) = h(86) = h(52) = i$

Worst case:

- Every key hashes to the same slot.
- Access time = $\Theta(n)$ if $|S| = n$

Average-case analysis of chaining

We make the assumption of ***simple uniform hashing:***

- Each key $k \in S$ is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the ***load factor*** of T to be

$$\alpha = n/m$$

$$= \text{average number of keys per slot.}$$

Search cost

The expected time for an ***unsuccessful*** search for a record with a given key is $= \Theta(1 + \alpha)$.

search the list

apply hash function and access slot

Expected search time $= \Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.

Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.

Division method

Assume all keys are integers, and define

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

Deficiency: Don't pick an m that has a small divisor d. A preponderance of keys that are congruent modulo d can adversely affect uniformity.

Extreme deficiency: If $m = 2^r$, then the hash doesn't even depend on all the bits of k:

- If $k = 10110001110110101_2$ and $r = 6$, then $h(k) = 011010_2$.

$h(k)$

Division method (continued)

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

Pick m to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

Annoyance:

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.

Multiplication method

Assume that all keys are integers, $m = 2^r$, and our computer has w-bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r),$$

where rsh is the "bitwise right-shift" operator and A is an odd integer in the range $2^{w-1} < A < 2^w$.

- Don't pick A too close to 2^{w-1} or 2^w.
- Multiplication modulo 2^w is fast compared to division.
- The rsh operator is fast.

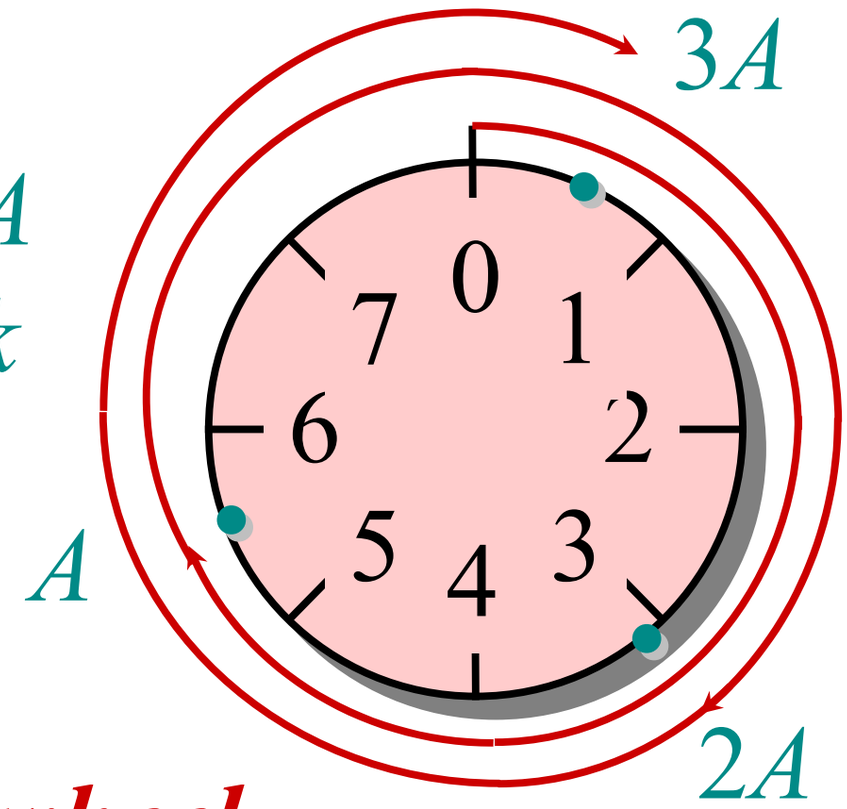
Multiplication method

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words:

$$\begin{array}{rr} & 1\,0\,1\,1\,0\,0\,1 = A \\ \times & 1\,1\,0\,1\,0\,1\,1 = k \\ \hline 1\,0\,0\,1\,0\,1\,0\,0\,1\,1\,0\,0\,1\,1 & \end{array}$$

$h(k)$



Modular wheel

Resolving collisions by open addressing

No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:

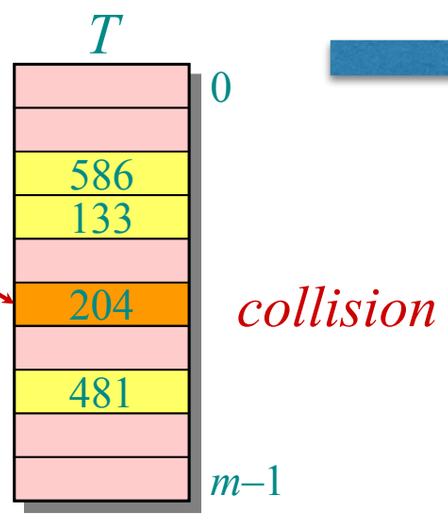
  $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$.

- The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$.
- The table may fill up, and deletion is difficult (but not impossible).

Example of open addressing

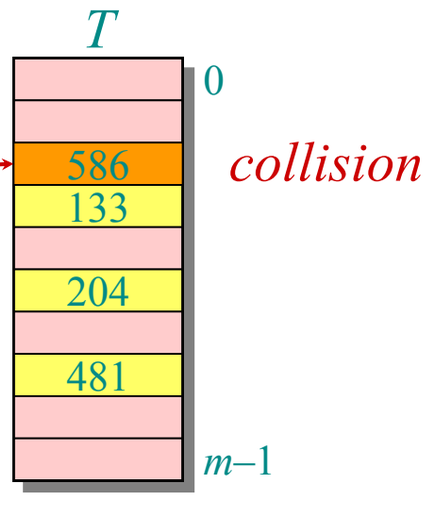
Insert key $k = 496$:

0. Probe $h(496,0)$



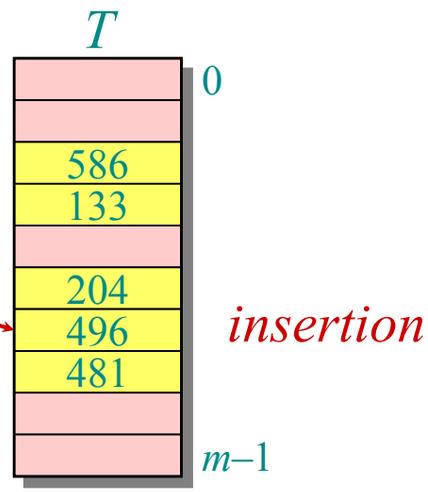
Insert key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$



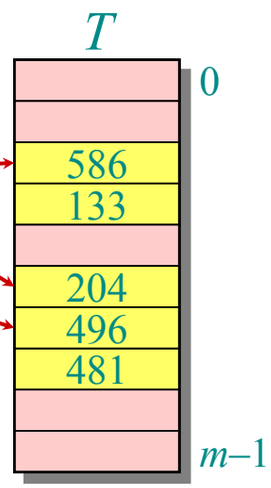
Insert key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

Probing strategies

Linear probing:

Given an ordinary hash function $h'(k)$, linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \bmod m.$$

This method, though simple, suffers from ***primary clustering***, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

Probing strategies

Double hashing

Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

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

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.

Analysis of open addressing

We make the assumption of ***uniform hashing:***

- Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Proof.

- At least one probe is always necessary.
- With probability n/m, the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n–1)/(m–1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n–2)/(m–2)$, the third probe hits an occupied slot, etc.

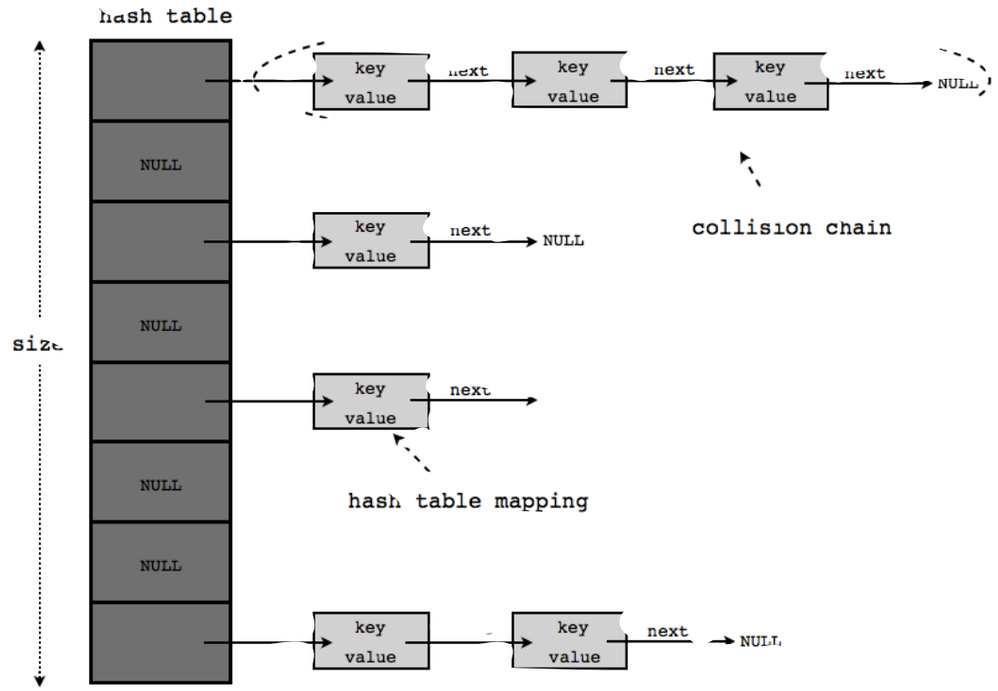
Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$.

Therefore, the expected number of probes is

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$
$$\le 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots)))$$
$$\le 1+\alpha+\alpha^2+\alpha^3+\cdots$$
$$= \sum_{i=0}^{\infty}\alpha^i$$
$$= \frac{1}{1-\alpha}. \square$$

Implications of the theorem

- If α is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is $1/(1–0.5) = 2$.
- If the table is 90% full, then the expected number of probes is $1/(1–0.9) = 10$.



Hashing II

- Universal hashing
- Universality theorem
- Constructing a set of universal hash functions
- Perfect hashing

A weakness of hashing

Problem: For any hash function h, a set of keys exists that can cause the average access time of a hash table to skyrocket.

- An adversary can pick all keys from $\{k \in U : h(k) = i\}$ for some slot i.

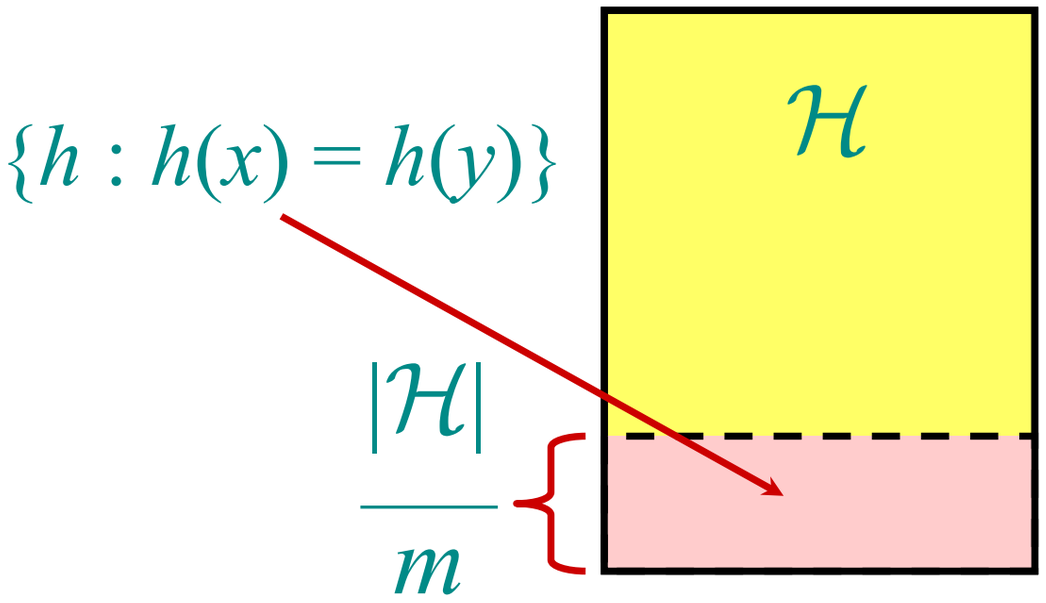
IDEA: Choose the hash function at random, independently of the keys.

- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn't know exactly which hash function will be chosen.

Universal hashing

Definition. Let U be a universe of keys, and let $\mathcal{H}$ be a finite collection of hash functions, each mapping U to $\{0, 1, \ldots, m-1\}$. We say $\mathcal{H}$ is ***universal*** if for all $x, y \in U$, where $x \neq y$, we have $|\{h \in \mathcal{H} : h(x) = h(y)\}| = |\mathcal{H}|/m$.

That is, the chance of a collision between x and y is $1/m$ if we choose h randomly from $\mathcal{H}$.



Universality is good

Theorem. Let h be a hash function chosen (uniformly) at random from a universal set $\mathcal{H}$ of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T. Then, for a given key x, we have

$$E[\#\text{collisions with } x] < n/m.$$

Proof. Let C_x be the random variable denoting the total number of collisions of keys in T with x, and let

$$c_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

Note: $E[c_{xy}] = 1/m$ and $C_x = \sum_{y \in T - \{x\}} c_{xy}$.

$$\begin{aligned} E[C_x] &= E\left[\sum_{y \in T - \{x\}} c_{xy}\right] && \bullet \text{ Take expectation of both sides.} \\ &= \sum_{y \in T - \{x\}} E[c_{xy}] && \bullet \text{ Linearity of expectation.} \\ &= \sum_{y \in T - \{x\}} 1/m && \bullet\ E[c_{xy}] = 1/m. \\ &= \frac{n-1}{m} . \ \square && \bullet \text{ Algebra.} \end{aligned}$$

Constructing a set of universal hash functions

Let m be prime. Decompose key k into $r + 1$ digits, each with value in the set $\{0, 1, \ldots, m-1\}$. That is, let $k = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

Randomized strategy:

Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from $\{0, 1, \ldots, m-1\}$.

Define $h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m$.

Dot product, modulo m

How big is $\mathcal{H} = \{h_a\}$? $|\mathcal{H}| = m^{r+1}$. ← **REMEMBER THIS!**

Universality of dot-product hash functions

Theorem. The set $\mathcal{H} = \{h_a\}$ is universal.

Proof. Suppose that $x = \langle x_0, x_1, \ldots, x_r \rangle$ and $y = \langle y_0, y_1, \ldots, y_r \rangle$ be distinct keys. Thus, they differ in at least one digit position, wlog position 0. For how many $h_a \in \mathcal{H}$ do x and y collide?

We must have $h_a(x) = h_a(y)$, which implies that

$$\sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod{m} .$$

Proof (continued)

Equivalently, we have

$$\sum_{i=0}^{r} a_i (x_i - y_i) \equiv 0 \pmod{m}$$

or

$$a_0 (x_0 - y_0) + \sum_{i=1}^{r} a_i (x_i - y_i) \equiv 0 \pmod{m},$$

which implies that

$$a_0 (x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i (x_i - y_i) \pmod{m}.$$

Fact from number theory

Theorem. Let m be prime. For any $z \in \mathbb{Z}_m$ such that $z \neq 0$, there exists a unique $z^{-1} \in \mathbb{Z}_m$ such that

$$z \cdot z^{-1} \equiv 1 \pmod{m}.$$

Example: $m = 7$.

z	1	2	3	4	5	6
z^{-1}	1	4	5	2	3	6

Back to the proof

We have

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m},$$

and since $x_0 \neq y_0$, an inverse $(x_0 - y_0)^{-1}$ must exist, which implies that

$$a_0 \equiv \left(-\sum_{i=1}^{r} a_i(x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \pmod{m}.$$

Thus, for any choices of $a_1, a_2, \ldots, a_r$, exactly one choice of a_0 causes x and y to collide.

Back to the proof

Q. How many h_a's cause x and y to collide?

A. There are m choices for each of $a_1, a_2, \ldots, a_r$, but once these are chosen, exactly one choice for a_0 causes x and y to collide, namely

$$a_0 = \left(\left(-\sum_{i=1}^{r} a_i (x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \right) \bmod m .$$

Thus, the number of h's that cause x and y to collide is $m^r \cdot 1 = m^a = |\mathcal{H}|/m$. □

Theorem. Let $\mathcal{H}$ be a class of universal hash functions for a table of size $m = n^2$. Then, if we use a random $h \in \mathcal{H}$ to hash n keys into the table, the expected number of collisions is at most $1/2$.

Proof. By the definition of universality, the probability that 2 given keys in the table collide under h is $1/m = 1/n^2$. Since there are $\binom{n}{2}$ pairs of keys that can possibly collide, the expected number of collisions is

$$\binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2} . \quad \square$$

Corollary. The probability of no collisions is at least $1/2$.

Proof. ***Markov's inequality*** says that for any nonnegative random variable X, we have

$$\Pr\{X \geq t\} \leq E[X]/t.$$

Applying this inequality with $t = 1$, we find that the probability of 1 or more collisions is at most $1/2$. □

Thus, just by testing random hash functions in $\mathcal{H}$, we'll quickly find one that works.

Perfect hashing

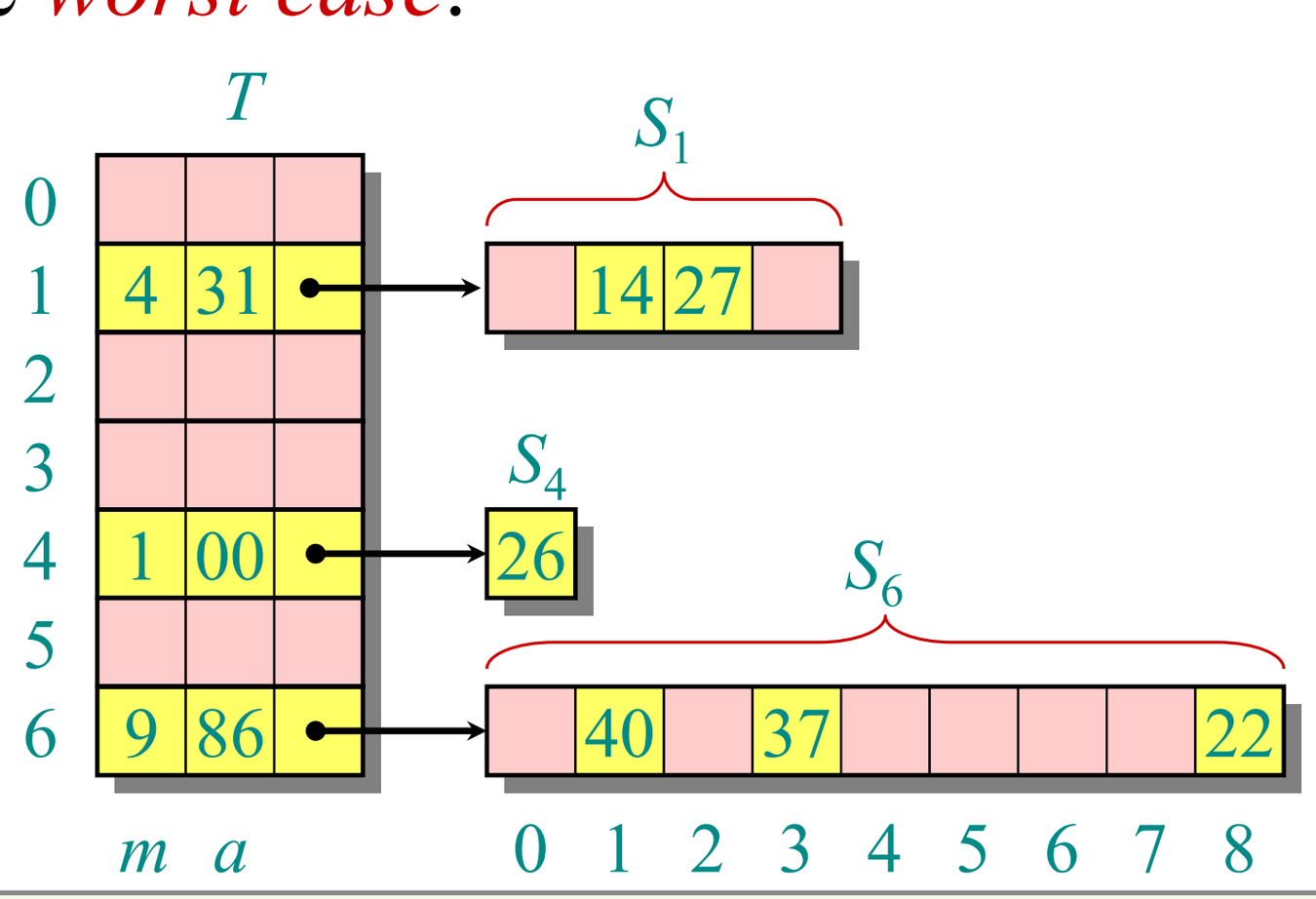
Although hashing is most often used for its excellent expected performance, hashing can be used to obtain excellent *worst-case* performance when the set of keys is ***static:*** once the keys are stored in the table, the set of keys never changes.

Some applications naturally have static sets of keys: consider the set of reserved words in a programming language, or the set of file names on a CD-ROM. We call a hashing technique ***perfect hashing*** if the worst-case number of memory accesses required to perform a search is $O(1)$.

Given a set of n keys, construct a static hash table of size $m = O(n)$ such that SEARCH takes $\Theta(1)$ time in the *worst case*.

IDEA: Two-level scheme with universal hashing at both levels.

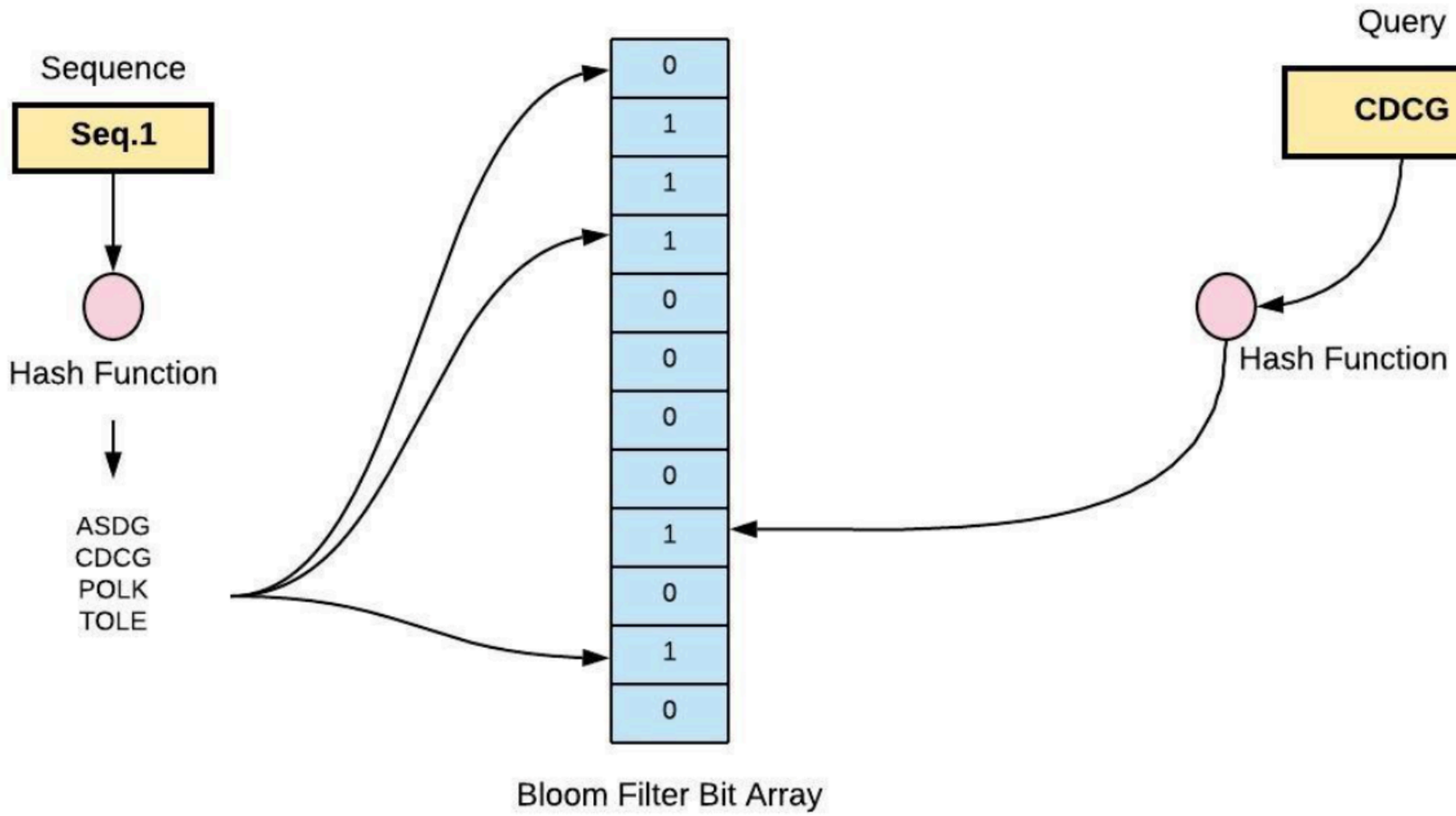
No collisions at level 2!



New problem: How can we know if x is a member of S

New problem: How can we know if x is a member of S

- Bloom Filters



Q: Which Hash functions?
R: https://en.wikipedia.org/wiki/MurmurHash

https://pypi.org/project/bloom-filter/

bloom-filter 1.3

```
pip install bloom-filter
```

```
from bloom_filter import BloomFilter

# instantiate BloomFilter with custom settings,
# max_elements is how many elements you expect the filter to hold.
# error_rate defines accuracy; You can use defaults with
# `BloomFilter()` without any arguments. Following example
# is same as defaults:
bloom = BloomFilter(max_elements=10000, error_rate=0.1)

# Test whether the bloom-filter has seen a key:
assert "test-key" in bloom is False

# Mark the key as seen
bloom.add("test-key")

# Now check again
assert "test-key" in bloom is True
```

Test on Google Colab