

# CS 237: Probability in Computing

Wayne Snyder  
Computer Science Department  
Boston University

---

## Lecture 20: Bloom Filters

- Motivation: "Call before you go"
- Exception Testing
- Bloom Filters

# Probabalistic Algorithms and Data Structures

A **Probabalistic or Randomized Algorithm** is “an algorithm that employs a degree of randomness as part of its logic.” (Wikipedia)

There are two principle types of randomized algorithm, depending on whether the random behavior affects the running time, or the correctness of the results:

**Las Vegas Algorithm:** Guaranteed to give a correct answer, probably quickly, but running time is uncertain and it may even run forever.

**Monte Carlo Algorithm:** Guaranteed to terminate quickly, but the answer is only correct with some probability. Usually, you can run it longer to get better answers.

Punchline: “Monte Carlo algorithms are always fast, but only probably correct. On the other hand, Las Vegas algorithms are always correct, but only probably fast.” (Wikipedia)

# Las Vegas Algorithms

## Example 1: Hash Tables

Hashtables perform correct insert, delete, and member operations in  $O(1)$  expected time, but for some inputs, the running time is  $O(n)$ .

## Example 2: Las Vegas Algorithm for Member of a List

# Is x in the list L?

```
def member( x, L ) :  
    while True:  
        k = randint( len(L) )  
        if L[k] == x:  
            return True
```

# Is always correct, but may not terminate

# Monte Carlo Algorithms

## Example 1: Monte Carlo Algorithm for Storing at most N Integers in a List:

To test for membership, just do a linear search;

To delete, search for the element, and if it is there, remove it; and

To insert: If the element is not there, then **add it to the front of the list and remove the last element.**

## Example 2: Large File comparison

Input: File A and file B (both large)

Question: Are A and B identical?

Algorithm:

```
if (hash(A) == hash(B)):
    print("Files identical!")
else:
    print("Files not identical!")
```

Later in this this lecture, we will look at an algorithm/data structure called a **Bloom Filter**, which is a Monte Carlo algorithm for doing membership tests....

# Monte Carlo Algorithms for Exception Testing

The paradigm of “exception testing” is a perfect domain for Monte Carlo algorithm:

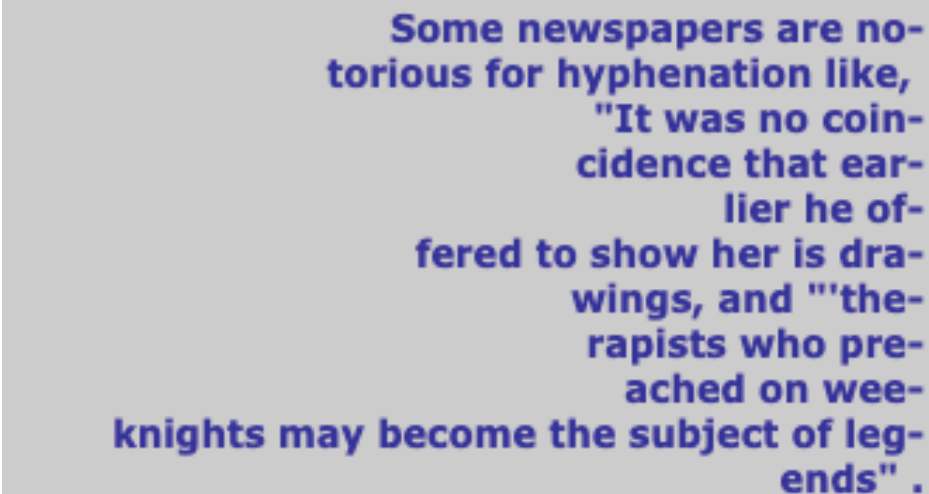
**Basic Idea:** “Call before you go!” Before doing an expensive operation, do a quick check to see if it is even necessary.



# Monte Carlo Algorithms for Exception Testing

**Example:** A **Hyphenation Algorithm** tells a word processor where to insert a hyphen to break words at syllable boundaries, for example, at the end of the line of right-justified text.

If you get this wrong, it is hard to read the result!



**Some newspapers are no-  
torious for hyphenation like,  
"It was no coin-  
cidence that ear-  
lier he of-  
fered to show her is dra-  
wings, and "the-  
rapists who pre-  
ached on wee-  
knights may become the subject of leg-  
ends" .**

# Exception Testing: Example

All typesetting programs (e.g., Word, Latex) have a hyphenation algorithm which determines how to break up words with hyphens. You can also check online:

**Hyphenate it!**

word	<input type="text" value="impeachment"/>
hyphenated	<input type="text" value="im•peach•ment"/>
language	<input type="text" value="English (US)"/>
	<input type="button" value="submit"/>

Copyright © ushuaia.pl

# Exception Testing: Example

The algorithm consists of checking various rules, and their exceptions:

**Rule 1.** Hyphenate prefixes when they come before proper nouns or proper adjectives.

**Examples:**

*trans-American*

*mid-July*

**Rule 2.** In describing family relations, *great* requires a hyphen, but *grand* becomes part of the word without a hyphen.

**Examples:**

*My grandson and my granduncle never met.*

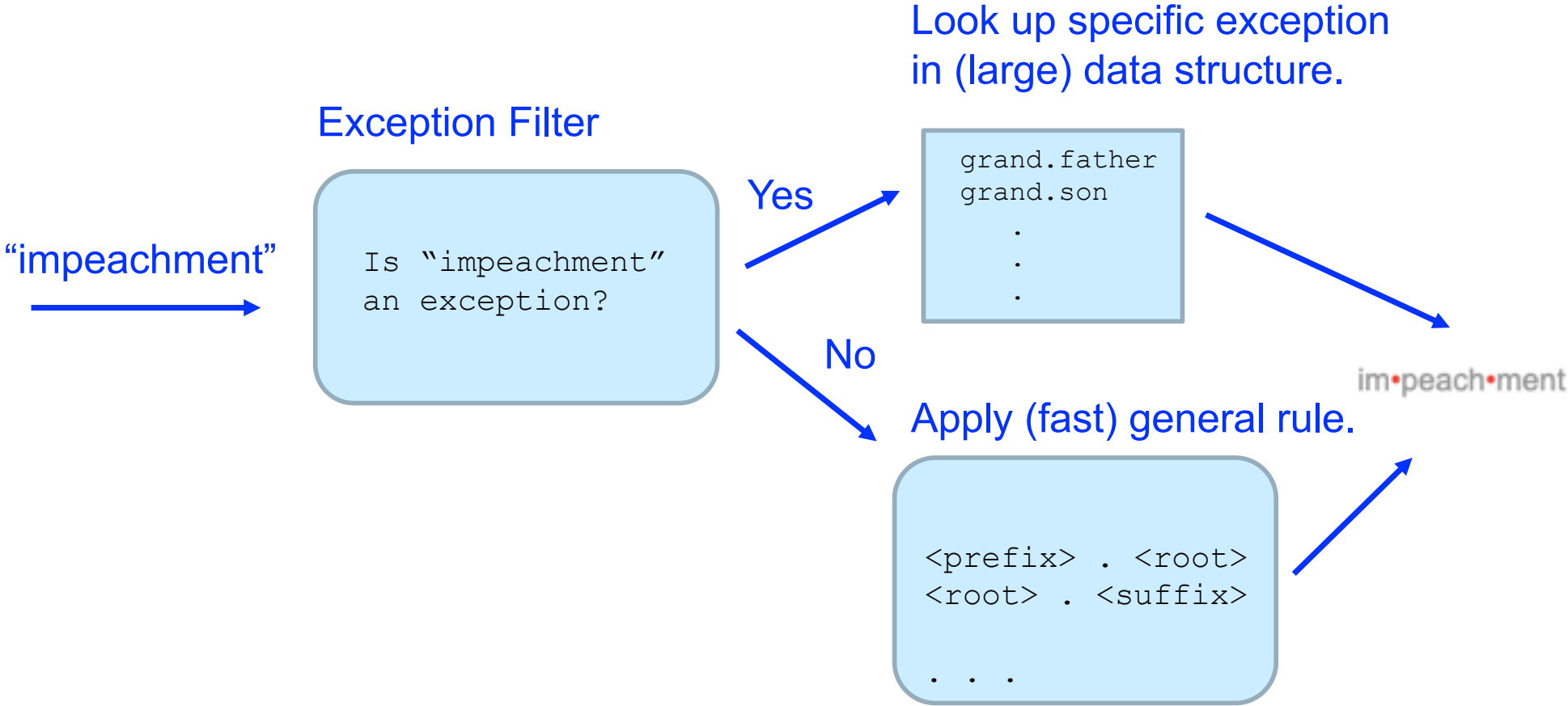
*My great-great-grandfather fought in the Civil War.*

Do not hyphenate *half brother* or *half sister*.



# Exception Testing: Example

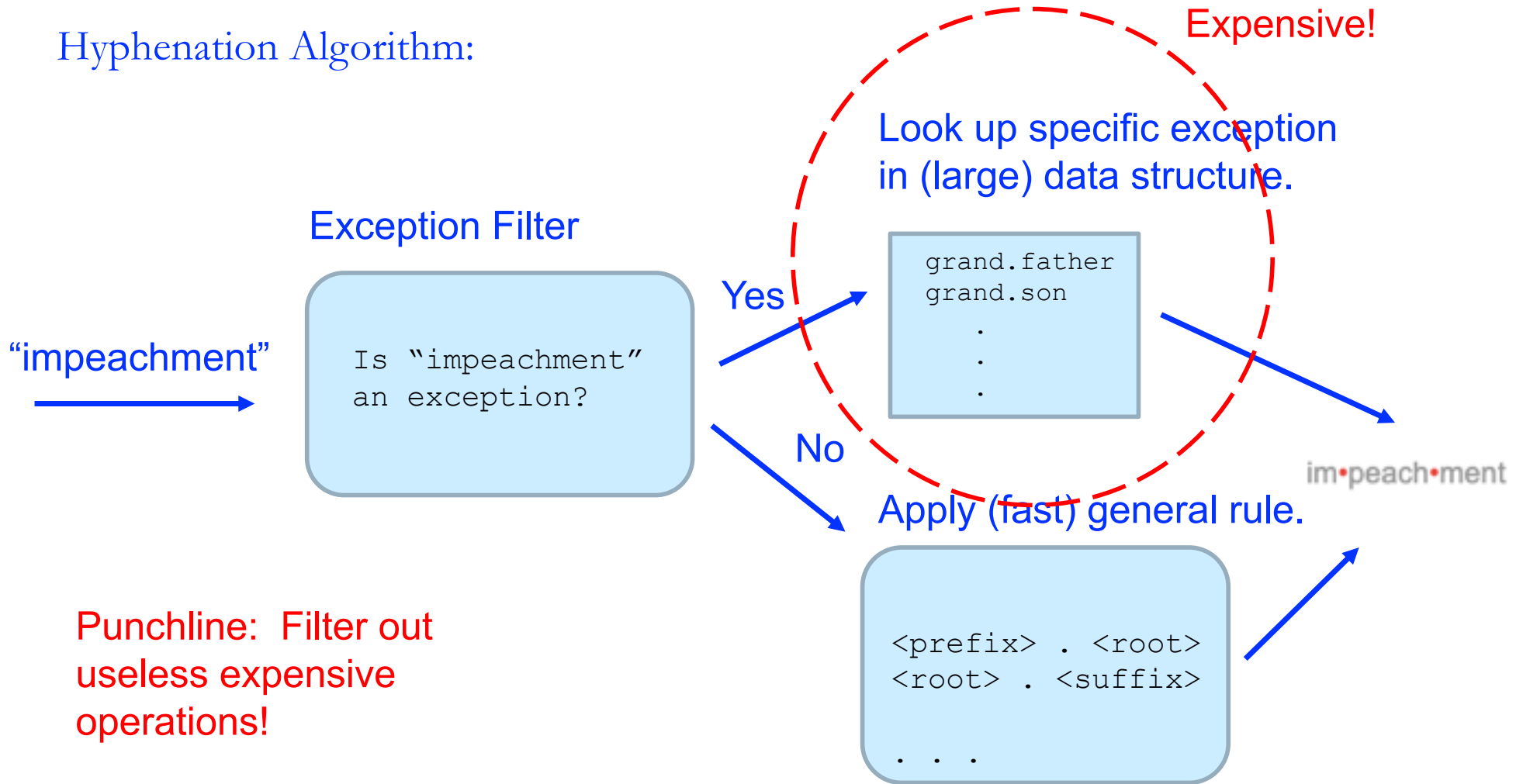
Hyphenation Algorithm:



# Exception Testing: Example

hy•phen•ation

Hyphenation Algorithm:

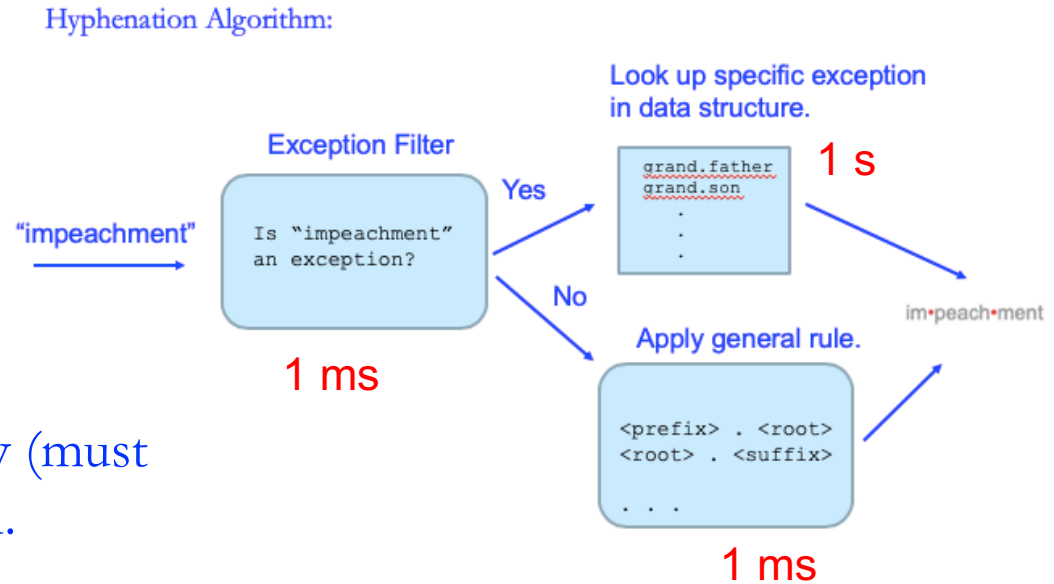


# Exception Testing: Example

## Quick Quiz 1:

Now, in general:

- **Applying general rule is fast** (no data lookup), say 1 ms ( 1/1000 sec );
- **Looking up specific word is slow** (must search the data structure), say 1 second.



Now, suppose the **Exception Filter** is also very fast, say 1 ms.

**Question:** Suppose you process 100 queries: how long would it take if

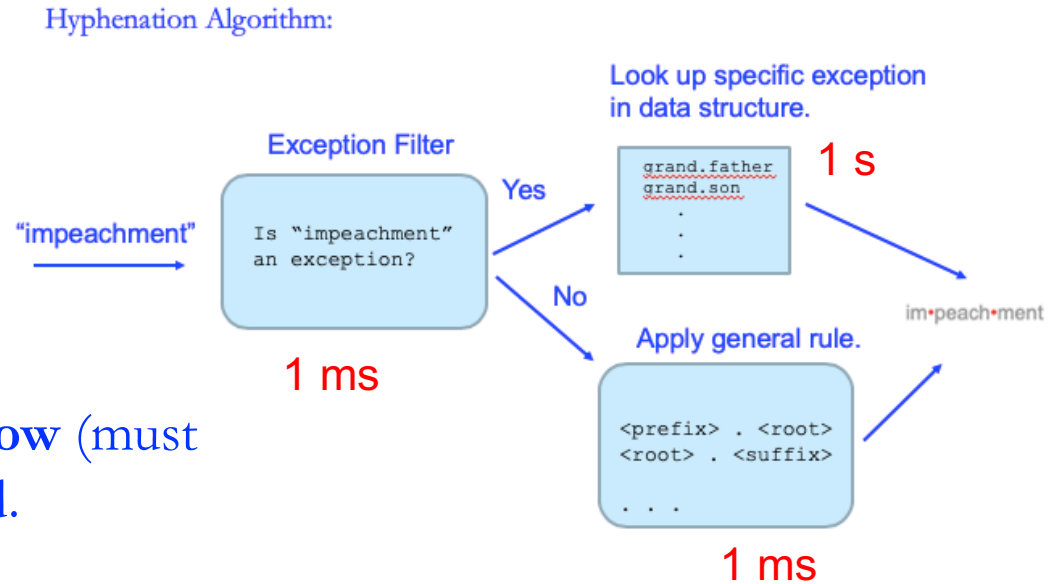
- (a) There are no exceptions?
- (b) All 100 are exceptions?

# Exception Testing: Example

## Quick Quiz 1:

Now, in general:

- Applying general rule is **fast** (no data lookup), say 1 ms ( 1/1000 sec );
- Looking up specific exception is **slow** (must search the data structure), say 1 second.



Now, suppose the **Exception Filter** is also very fast, say 1 ms.

**Question:** Suppose you process 100 queries: how long would it take if

(a) There are no exceptions?

$$100 * (1 \text{ ms} + 1 \text{ ms}) = 0.2 \text{ sec}$$

(b) All 100 are exceptions?

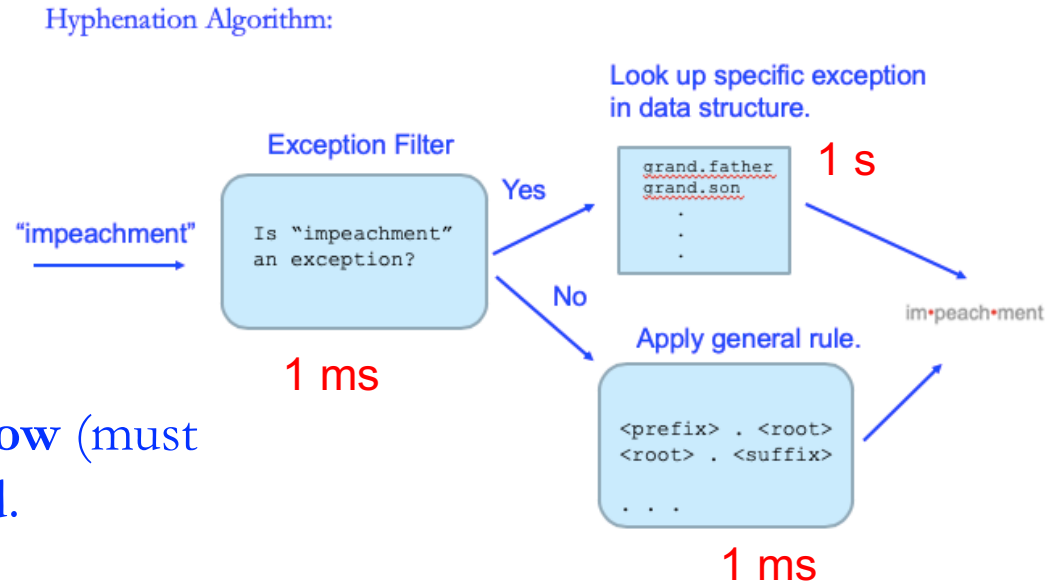
$$100 * (1 \text{ ms} + 1 \text{ s}) = 100.1 \text{ sec}$$

# Exception Testing: Example

## Quick Quiz 1:

Now, in general:

- Applying general rule is **fast** (no data lookup), say 1 ms ( 1/1000 sec );
- Looking up specific exception is **slow** (must search the data structure), say 1 second.



Now, suppose the **Exception Filter** is also very fast, say 1 ms.

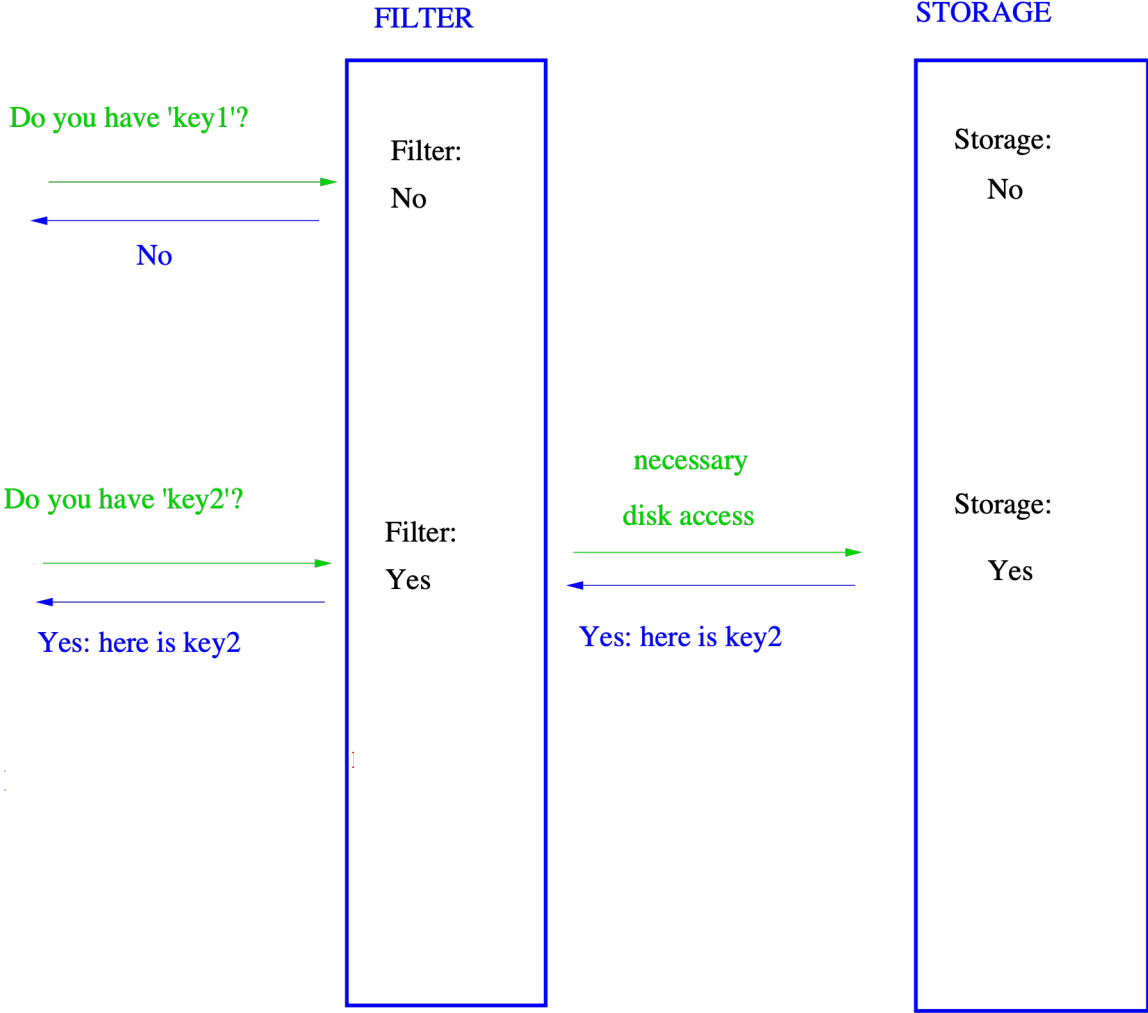
**Question:** Suppose you process 100 queries: how long would it take if

- (a) There are no exceptions?  $100 * (1 \text{ ms} + 1 \text{ ms}) = 0.2 \text{ sec}$
- (b) All 100 are exceptions?  $100 * (1 \text{ ms} + 1 \text{ s}) = 100.1 \text{ sec}$
- (c) Out of the 100, only 10 are exceptions?

$$90 * (1 \text{ ms} + 1 \text{ ms}) + 10 * (1 \text{ ms} + 1 \text{ s}) = 180 \text{ ms} + 10.01 \text{ s} = 10.19 \text{ sec}$$

# Exception Testing

So the general idea is: Do a fast membership test before a slow search of the disk, to avoid useless searches.



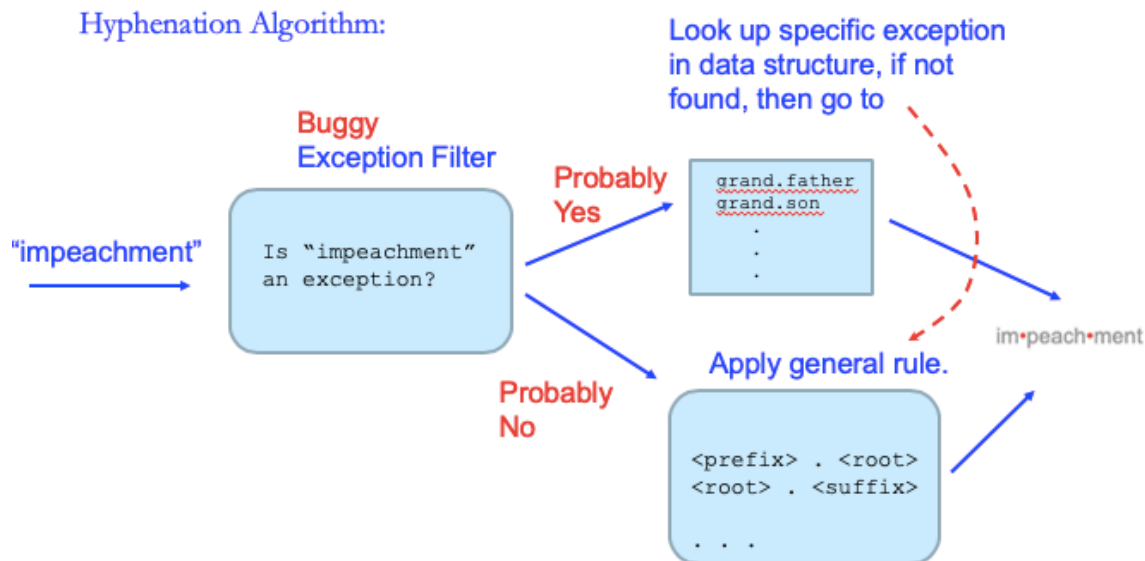
# Exception Testing: Example 1

## Quick Quiz 2:

Same as last time:

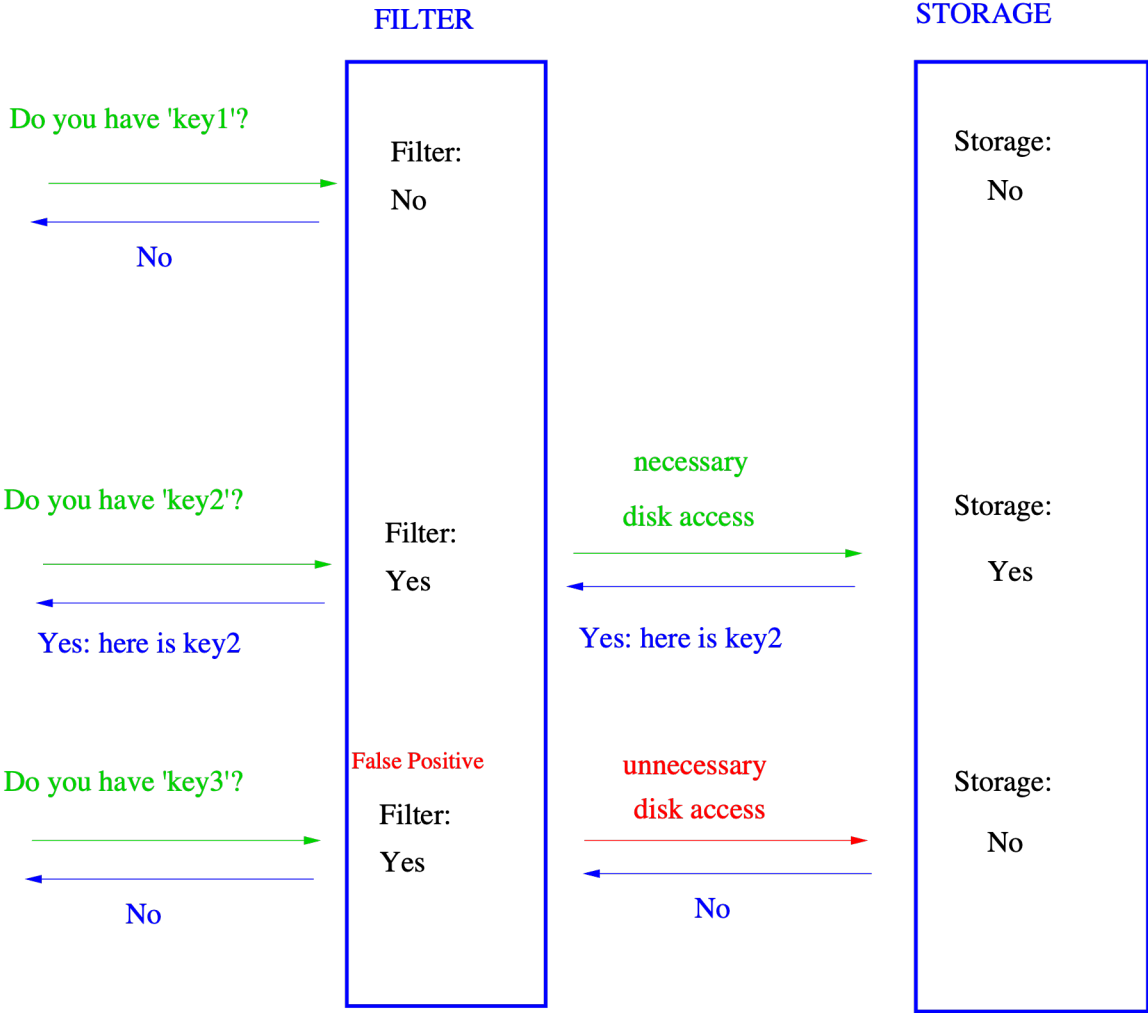
- Applying general rule is **fast** (no data lookup), say 1 ms ( 1/1000 sec );
- Looking up specific word is **slow** (must search the data structure), say 1 second.

BUT suppose the filter is buggy, and misclassifies 1% of the non-exceptions. In this case, an unnecessary search is made, and then the general rule is applied. Such mistakes are called **FALSE POSITIVES**.



# Exception Testing

BUT the Filter is not correct, and misclassifies 1% of the non-exceptions. In this case, an unnecessary search is made, and then the general rule is applied. Such mistakes are called FALSE POSITIVES.



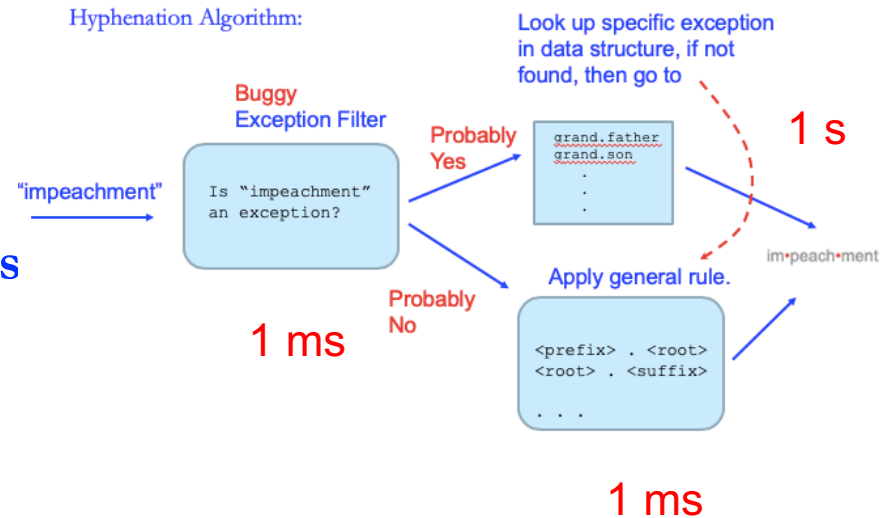


# Exception Testing

## Quick Quiz 2 continued:

Same as last time:

- Filtering and applying general rules are **fast** say 1 ms ( 1/1000 sec );
- Looking up specific word is **slow** (must search the data structure), say 1 second.



BUT the Filter is not correct, and misclassifies 1% of the non-exceptions. In this case, an unnecessary search is made, and then the general rule is applied. Such mistakes are called FALSE POSITIVES.

## Questions:

- Is this hypothesis algorithm correct, even if the filter is not? [Hint: what happens on a false positive?]
- Suppose you process 100 queries: how long would it take if 89 are not exceptions, 10 are true exceptions, and **1 is a false positive.**

# Exception Testing: Example 1

## Quick Quiz 2:

### Questions:

- a) Is this hypothesis algorithm correct, even if the filter is not? [Hint: what happens on a false positive?] **YES**
- b) Suppose you process 100 queries: how long would it take if 89 are not exceptions, 10 are true exceptions, and 1 is a false positive.

$$89 * (1 \text{ ms} + 1 \text{ ms}) + 10 * (1 \text{ ms} + 1 \text{ s}) + 1 * (1 \text{ ms} + 1 \text{ s} + 1 \text{ ms})$$
$$= 178 \text{ ms} + 10.01 \text{ s} + 1.002 \text{ s} = 11.19 \text{ sec}$$

### Compare:

**Question:** Suppose you process 100 queries: how long would it take if

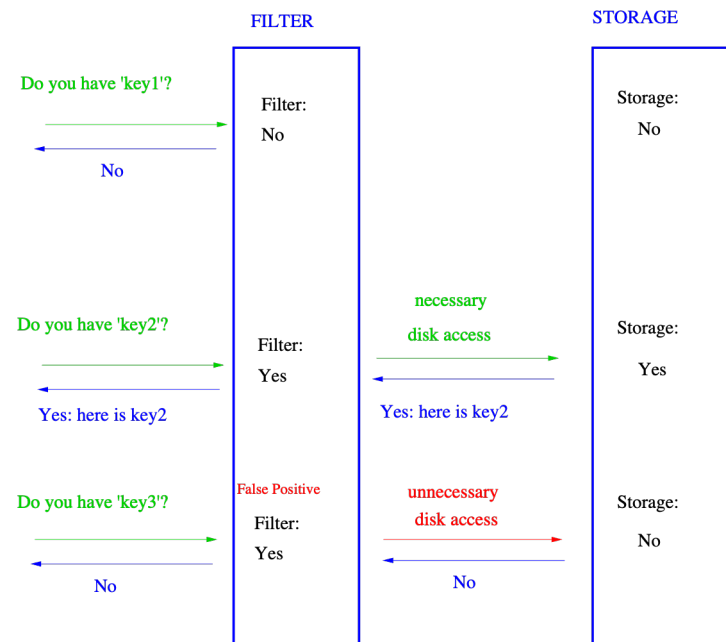
- (a) There are no exceptions?  $100 * (1 \text{ ms} + 1 \text{ ms}) = 0.2 \text{ sec}$
- (b) All 100 are exceptions?  $100 * (1 \text{ ms} + 1 \text{ s}) = 100.1 \text{ sec}$
- (c) Out of the 100, only 10 are exceptions?

$$90 * (1 \text{ ms} + 1 \text{ ms}) + 10 * (1 \text{ ms} + 1 \text{ s}) = 180 \text{ ms} + 10.01 \text{ s} = 10.19 \text{ sec}$$

# Exception Testing: Summary

What have we learned about the process of filtering out (expensive) exceptions?

- It makes sense to check for membership in a data structure before trying to do an access, IF the membership test is much faster than the access time.
- Occasionally failures of the membership test do not cause failures overall, and the result is still much faster than NOT filtering.



# Bloom Filters

So we need an algorithm/data structure which

- Only answers membership questions (yes/no);
- Is as efficient in time and space as possible; and
- Can give wrong answers with small probability.

A **Bloom Filter** is a variation of a hash table and a bit array which is designed for these exact conditions. A simple version is as follows:

Data Structure: A bit array A: 

0	1	0	1	1	1	0	0	0	0	0	1	0	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Algorithms:

```
def insert(key):  
    A[ hash(key) ] = 1  
  
def member(key):  
    return A[ hash(key) ] == 1
```

## Questions:

- What is the time and space complexity?
- When does it give wrong answers (false positives)?
- How likely are false positives?

# Bloom Filters

So we need an algorithm/data structure which

- Only answers membership questions (yes/no);
- Is as efficient in time and space as possible; and
- Can give wrong answers with small probability.

A **Bloom Filter** is a variation of a hash table and a bit array which is designed for these exact conditions. A simple version is as follows:

Data Structure: A bit array A: 

0	1	0	1	1	1	0	0	0	0	0	1	0	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Algorithms:

```
def insert(key):  
    A[ hash(key) ] = 1  
  
def member(key):  
    return A[ hash(key) ] == 1
```

## Questions

- What is the time and space complexity? **N bits, O(1) worst case**
- When does it give wrong answers (false positives)? **On hash collision.**
- How likely are false positives? **Depends....**

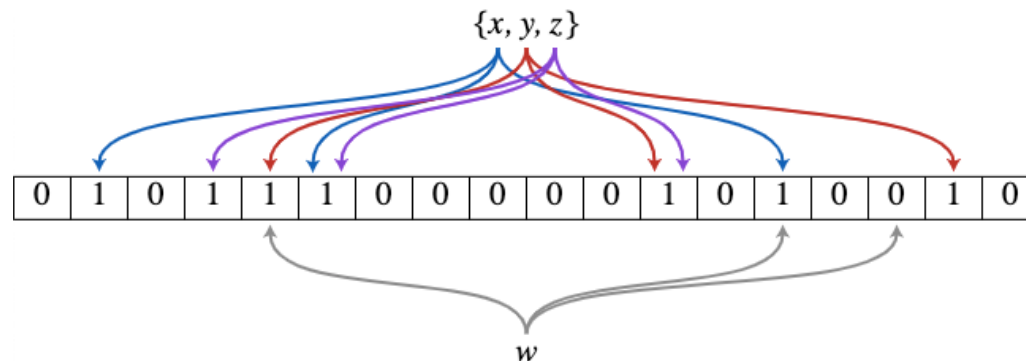
# Bloom Filters

A **Bloom Filter** is a variation of a hash table and a bit array which is designed for these exact conditions.

The actual Bloom Filter uses  $K$  hash functions  $h_1, h_2, \dots, h_k$  simultaneously:

```
def insert(key):  
    for each i:  
        A[ hi(key) ] = 1
```

```
def member(k):  
    return (A[h1(k)] == 1) and ... and (A[hk(k)] == 1)
```

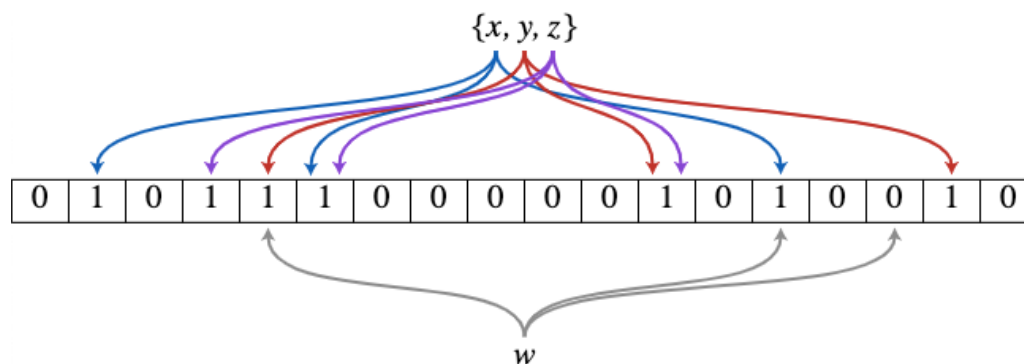


Let's do an example on the board....

# Bloom Filters

A **Bloom Filter** is a variation of a hash table and a bit array which is designed for these exact conditions.

The actual Bloom Filter uses  $K$  hash functions  $h_1, h_2, \dots, h_k$  simultaneously:



So: Constant time filter which gives false results some of the time!

For Discussion (other properties of the data structure):

Which of the following operations do you think would be easy, hard, or you don't know?

Deletion, Union (of two bloom filters), Intersection, Multiset