Introduction

Probabilistic data structures are specialized structures that use probabilistic algorithms to estimate certain properties of the stored data.

Unlike traditional data structures, which aim for precise and deterministic results, probabilistic counterparts trade accuracy for efficiency and scalability.

They are adept at handling large datasets, where traditional methods might be resource-intensive.

The appeal of probabilistic data structures is their ability to offer approximate answers with controlled error rates, consuming significantly less memory and computational resources than deterministic alternatives.

They thrive in situations where near-perfect results suffice.

Advantages of Probabilistic Data Structures

Efficiency in Memory Usage

Keyword: Memory Efficiency

Content: Probabilistic data structures use significantly less memory than traditional data structures for large datasets, making them ideal for applications with limited storage capacity or those that handle vast amounts of data.

Scalability

Keyword: Scalability

Content: They excel in environments where data grows continuously and unpredictably. Their design allows them to scale more gracefully, accommodating more data without a linear increase in resource consumption.

Speed

Keyword: Computational Speed

Content: Offers faster operations (like insertions and queries) on large datasets compared to deterministic structures, due to less stringent accuracy requirements.

Suitability for Big Data Applications

Keyword: Big Data Suitability

Content: Perfectly suited for big data applications where exactness is less critical than the overall trend or pattern detection, enabling quick decision-making based on large datasets.

Disadvantages of Probabilistic Data Structures

Accuracy and Error Rates

Keyword: Inaccuracy/Error Rates

Content: The trade-off for efficiency and scalability is a controlled but inherent inaccuracy in the form of false positives or approximations, making them unsuitable for applications requiring exact results.

Complexity in Understanding and Implementation

Keyword: Complexity

Content: The probabilistic nature and the algorithms used can be complex to understand and implement correctly, requiring specialized knowledge.

Difficulty in Tuning

Keyword: Tuning Difficulty

Content: Adjusting the parameters to optimize the balance between error rate, speed, and memory usage requires a deep understanding of the data structure and the specific application needs.

Limited Applicability

Keyword: Applicability Limits

Content: Their usefulness is confined to specific scenarios where approximations are acceptable. They may not be the right choice for all types of applications, especially those needing deterministic guarantees.

[Using Probabilistic Data Structures in Redis - Semaphore (semaphoreci.com)](https://semaphoreci.com/blog/probabilistic-data-structures-redis#:~:text=Probabilistic%20data%20structures%20are%20specialized,accuracy%20for%20efficiency%20and%20scalability.)

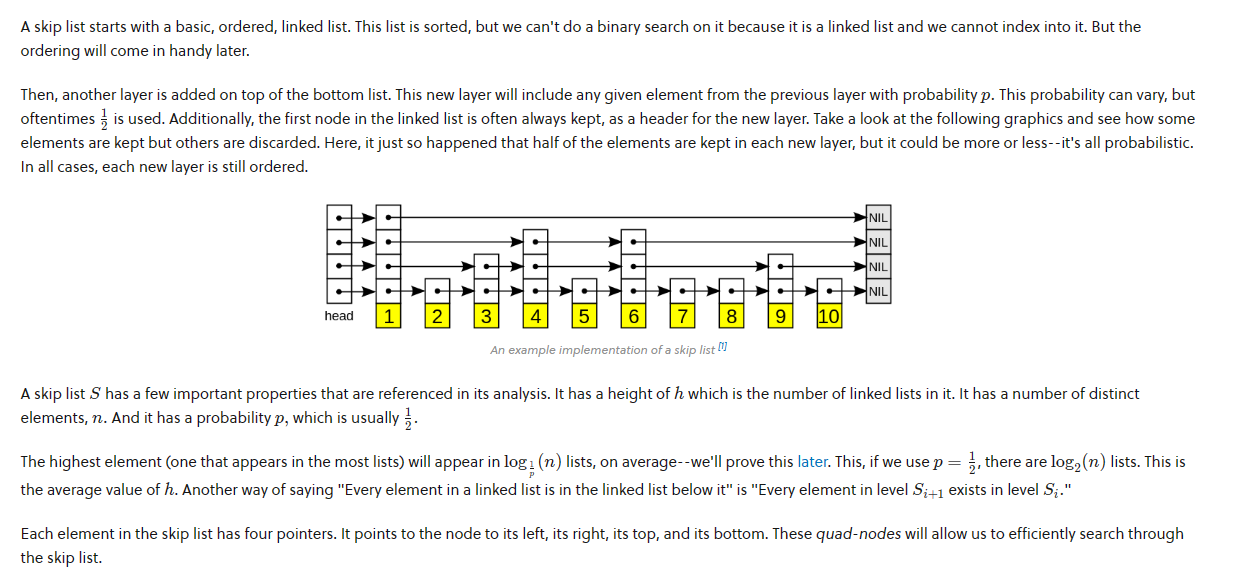
[probabilistic data structures - Google Search](https://www.google.com/search?sca_esv=d7ed808a3c5b3242&sca_upv=1&sxsrf=ACQVn09gQ62dMsXVb2Ub5lj1dtxwt9Y9cw:1712144763230&q=probabilistic+data+structures&uds=AMwkrPs3zfh7pDvEIUYe8xf2ljFSD186LhIgqZgxeBVOTBYWk0SgP6CLtGt5qDYqamNP2tMUHZGiQ_T51Rmo9XPZPQ2Eq6rCZSqJfCy27zCPMMYdqe-iKKiq30Z6g3ehO9AEFyYRl4goJ_sUa9Z5V0upGYGFydFxXsA4QXccgquacSps_PqGmMsjO2VhjNp6fDk2eZAQMdOSbHye7TeKxvJGpYGhwiLymC32VuNKlSWSxhc9F3xQhBtDCwp3QSDmp1Vk2z7LOdUE9vpkxLIk0dG6Oa-J0lhK_B2iE4FKgp_jW0MjlhH6mjUupf3g_4XjzYgXPLeV_KQs&udm=2&prmd=ivsnbmtz&sa=X&ved=2ahUKEwiQ-amj_KWFAxXkk4kEHfHYBi0QtKgLegQIERAB&biw=1872&bih=966&dpr=1#vhid=Jc9tdIDXXhiXdM&vssid=mosaic) – image

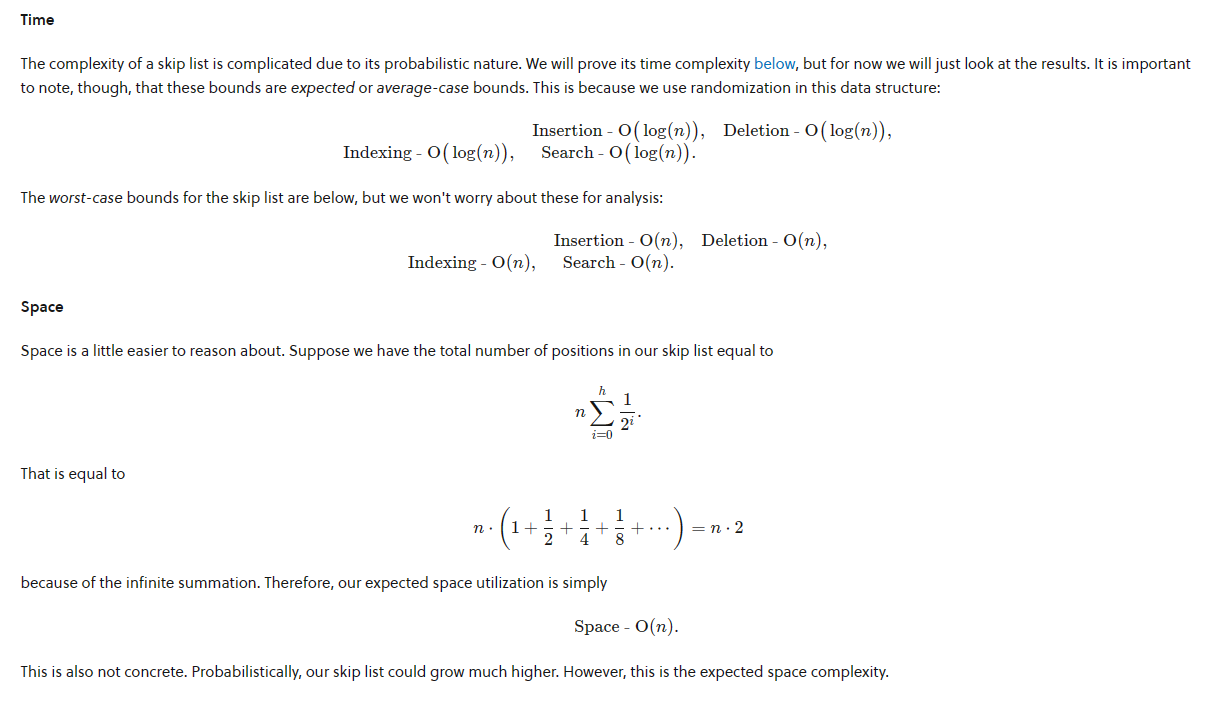
The skip list is a probabilisitc data structure that is built upon the general idea of a linked list. The skip list uses probability to build subsequent layers of linked lists upon an original linked list. Each additional layer of links contains fewer elements, but no new elements.

You can think about the skip list like a subway system. There's one train that stops at every single stop. However, there is also an express train. This train doesn't visit any unique stops, but it will stop at fewer stops. This makes the express train an attractive option if you know where it stops.

Skip lists are very useful when you need to be able to concurrently access your data structure. Imagine a red-black tree, an implementation of the binary search tree. If you insert a new node into the red-black tree, you might have to rebalance the entire thing, and you won't be able to access your data while this is going on. In a skip list, if you have to insert a new node, only the adjacent nodes will be affected, so you can still access large part of your data while this is happening.

[Skip List | Brilliant Math & Science Wiki](https://brilliant.org/wiki/skip-lists/#:~:text=The%20skip%20list%20is%20a,elements%2C%20but%20no%20new%20elements.)





lide 1: Introduction to Skip Lists

Title: Understanding Skip Lists

Text:

"Skip Lists: A probabilistic data structure enhancing traditional linked lists."

"Employs randomized algorithms to manage data efficiently."

Slide 2: Conceptual Overview of Skip Lists

Title: Skip Lists: A Subway System Analogy

Text:

"Imagine a subway system: Regular trains stop at every station, Skip List trains skip stops, arriving faster."

"Enables rapid transit through data, prioritizing speed and concurrency."

Slide 3: Structure and Design of Skip Lists

Title: Architecting Skip Lists

Text:

"Constructed with multiple layers, each a filtered 'express route' of the one below."

"Element inclusion by probabilistic selection—probability 'p', often 1/2."

"Ordered layers maintain a 'fast track' for data operations."

Slide 4: Example of a Skip List

Title: Visualizing Skip Lists

Text:

"Below is a representation of a skip list's layered structure."

"Note the 'head' element leading each layer and the diminishing number of elements as you ascend."

(Include the diagram of the skip list from your content)

Slide 5: Properties and Analysis of Skip Lists

Title: Analyzing Skip List Properties

Text:

"Height 'h' determines the list's maximum layer count."

"Probability 'p' influences layer population, with typical value 1/2."

"Quad-nodes point in four directions, providing efficient data navigation."

Slide 6: Time and Space Complexity

Title: Complexity in Skip Lists

Text:

"Time Complexity: Average-case insertion, deletion, and search operate in O(log(n)) time."

"Space Complexity: Optimized space usage with expected O(n) utilization."

Bloom Filters

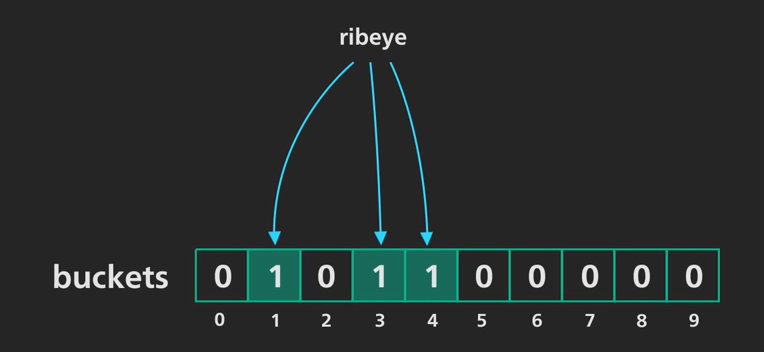
A bloom filter is a [probabilistic data structure](https://brilliant.org/wiki/probabilistic-data-structures/) that is based on [hashing](https://brilliant.org/wiki/hash-based-data-structure/). It is extremely space efficient and is typically used to add elements to a [set](https://brilliant.org/wiki/sets/) and test if an element is in a set. Though, the elements themselves are not added to a set. Instead a hash of the elements is added to the set.

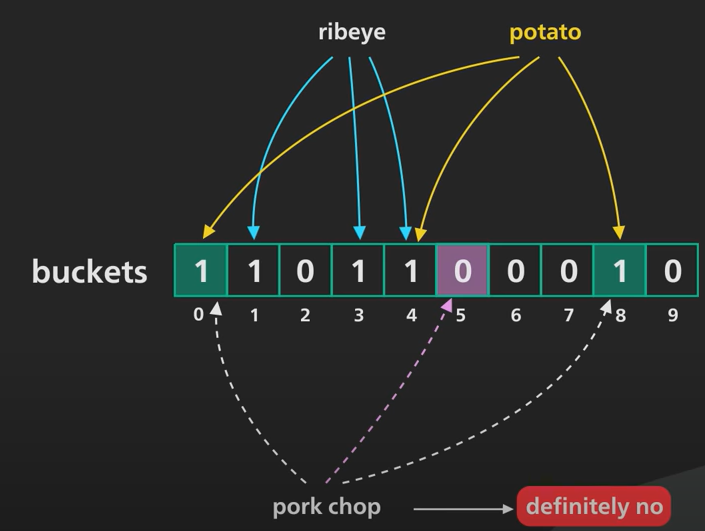
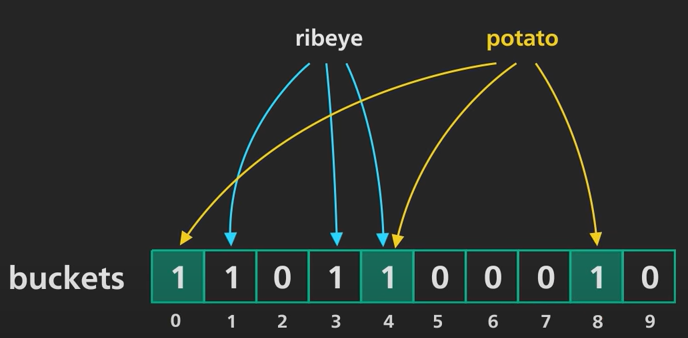
When testing if an element is in the bloom filter, false positives are possible. It will either say that an element is definitely not in the set or that it is possible the element is in the set.

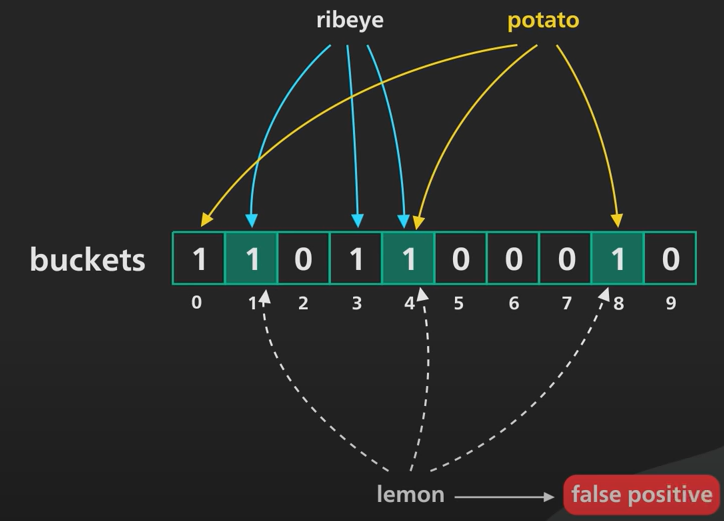
A bloom filter is very much like a [hash table](https://brilliant.org/wiki/hash-tables/) in that it will use a hash function to map a key to a bucket. However, it will not store that key in that bucket, it will simply mark it as filled. So, many keys might map to same filled bucket, creating false positives.

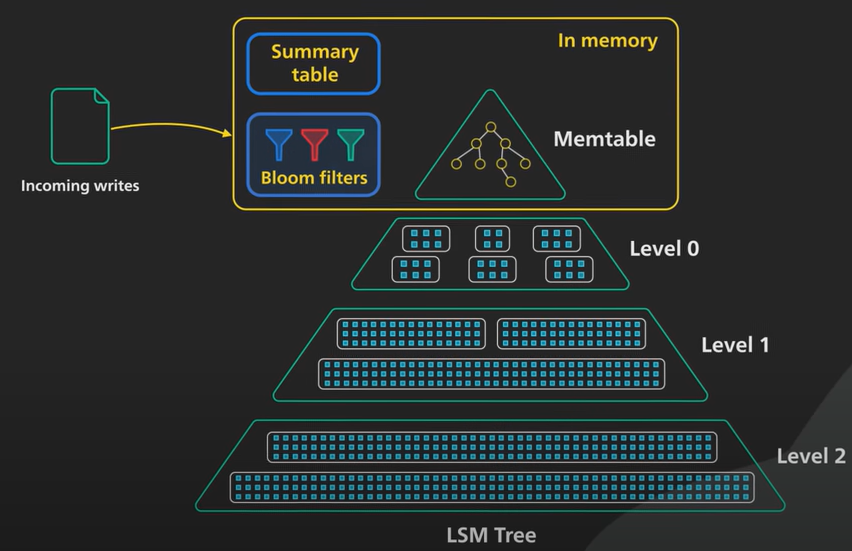
[Bloom Filter | Brilliant Math & Science Wiki](https://brilliant.org/wiki/bloom-filter/#:~:text=A%20bloom%20filter%20is%20a,is%20added%20to%20the%20set.)

[Bloom Filters | Algorithms You Should Know #2 | Real-world Examples (youtube.com)](https://www.youtube.com/watch?v=V3pzxngeLqw)









Cuckoo hashing

[How Cuckoo Hashing Work Part 1 (Introduction to Cuckoo Hashing) (youtube.com)](https://www.youtube.com/watch?v=GPiJUtdiUlo)

[Cuckoo Filter Basics. All that you need to know!! | Medium](https://medium.com/@humberto521336/cuckoo-filter-basics-44e03847788c)

Cuckoo filters and bloom filters are different in how they handle increased load. As the cuckoo filter increases load, insertions are more likely to fail, and so its time complexity increases exponentially. However, the false positive rate remains the same. Bloom filters can keep inserting items into the filter at the cost of an ever rising false positive rate.

A false positive in the context of Cuckoo filters occurs when a lookup is performed for an item not stored in the filter, and the filter incorrectly claims the item is present. This happens because Cuckoo filters rely on storing small, fixed-size fingerprints of items rather than the items themselves, and these fingerprints can collide—the same fingerprint can represent multiple items.

However, Cuckoo filters are designed to have a low rate of false positives and are generally more space-efficient than Bloom filters

Skip List

A skip list is a probabilistic alternative to balanced trees. It maintains multiple layers of linked lists, with each level skipping over fewer elements. The decision to include an element in a higher-level list is made randomly, following a fixed probability. This randomization ensures that, on average, the skip list remains balanced without the need for explicit rebalancing operations, unlike balanced trees.

Bloom Filter

The probabilistic nature comes from using multiple hash functions to map each element to several positions in a bit array. When inserting an element or checking for membership, all positions indicated by the hash functions are checked or set. The probability of false positives depends on the size of the bit array, the number of hash functions used, and the number of elements stored. The key aspect is that you can tune the parameters of a Bloom filter to achieve a desired balance between the space used and the false positive rate.

Cuckoo Filter

The probabilistic aspect of cuckoo filters arises from their use of cuckoo hashing, where each item can be in one of two possible locations (determined by two hash functions). If a new item needs to occupy a slot already taken, the existing item is "kicked out" and moved to its alternative location, potentially triggering a chain of relocations. This process can succeed or fail based on the chosen hash functions and the current state of the filter, with a well-designed filter ensuring high probability of success. However, there's always a probabilistic limit to how full the filter can get before insertions start failing due to cycles or too long kick-out chains, necessitating a resize or rehash.

Skip List:

I will discuss two pivotal papers on Skip Lists. The first presents novel methods for concurrent maintenance, enhancing access and updates efficiently. The second offers a scalable algorithm using optimistic synchronization, improving upon traditional lock mechanisms. Both works emphasize simplicity, provable correctness, and strong experimental performance.

Bloom Filters:

Turning to Bloom Filters, one study introduces variable-length signatures enabling the deletion of flows, and a bank of filters for packet action identification.

Another paper proposes a content-based approach to reduce memory usage and false positives, showing reductions in experimental evaluations.

Cuckoo Filter:

Lastly, for Cuckoo Filters, research has shown they offer dynamic capabilities over Bloom filters with the ability to add and remove items. They excel in space efficiency and are practically beneficial, as evidenced by their application in cybersecurity for password cracking, yielding considerable improvements in efficiency.

Future work:

As the framework is already setup we can tweak it to analyze the data storage

in different domains such as database, network, etc where ever there is tolerance for false positives.

Use this implementation as a core to build a complete data analysis product.

Make these implementation more robust make it available for general use case