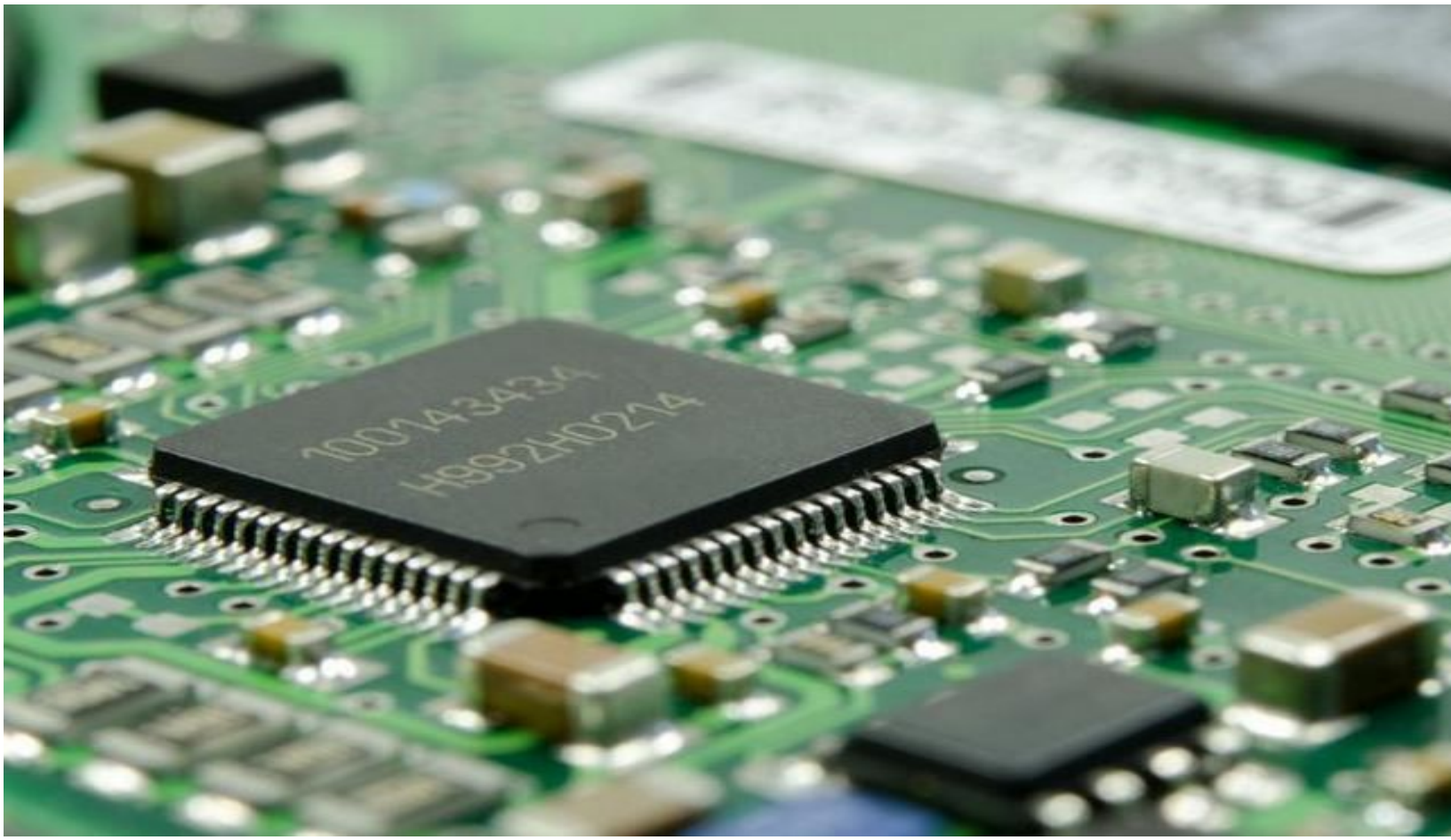
CSE373 Research Project

Winter 2021

Sean Sexton
Donavan Erickson
Melissa Truong

Memory and Cache-Friendly Algorithms



Looking at Swiss Tables, Bloom Filters, & Suffix Arrays

Motivation:

We know that runtime is important, but what has been neglected in recent years is the amount of memory that our programs are using. We have been in the heyday of Moore's law, which states that every two years the amount of transistors that can fit in a certain area will double. As many experts on the subject have pointed out, this can only happen for so long before we hit different limits.

We are currently getting into sub 10nm technology in the microchip where we must be able to control our chip manufacturing process at Angstrom levels. Processes are being deployed to even deposit and etch metals at an atomic level of precision to continue to keep us in the golden age of Moore's law. The fact is, we are going to hit the point within our lifetimes where Moore's law is no longer applicable.

What does all this mean for the computer industry?

As Moore's law begins to slow, access to cheaper, smaller, and more efficient storage may begin to slow it's progress. The ability to replace existing hardware with more efficient and higher capacity storage devices may stagnate. This means that as the businesses and usage of a data center may continue to grow, the storage and cost to keep up with demand may sky rocket. For this reason, programmers need to start to take into consideration how to make their code more efficiently use the memory and storage that is available to them, instead of focusing solely on runtime.

To answer the question: how can we design more memory-friendly and cache-efficient data structures, we will review the design and implementation of three different algorithms:

1) Swiss tables
2) Bloom filters
3) Suffix arrays

These three different algorithms will build upon the concepts learned in CSE373, specifically sets, maps, hash tables and arrays. We dive deeper into modifications of these data structures and how they can reduce memory storage and increase/improve runtime efficiency.

Algorithm #1: Swiss Tables:

What are Swiss Tables?

Swiss tables are hash tables that were designed by a team at Google. It has:

- N buckets
- Each bucket are of size 16 (grouping)

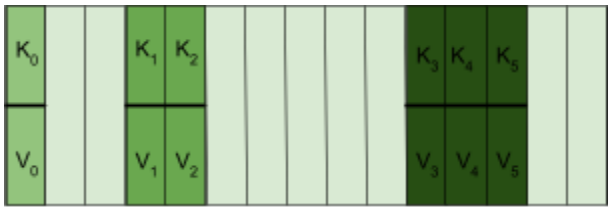
It deploys a 2-level hash table (*see details below*). These hash tables are designed to be replacements for `std::unordered_map` and `std::unordered_set`.

Looking at an overview, these hash tables, which live within the **Abseil** `container` library, include 4 different classes:

- **`absl::flat_hash_map and absl::flat_hash_set`**
- **`absl::node_hash_map and absl::node_hash_set`**

`What is the difference between flat vs node hash_map/hash_set?`

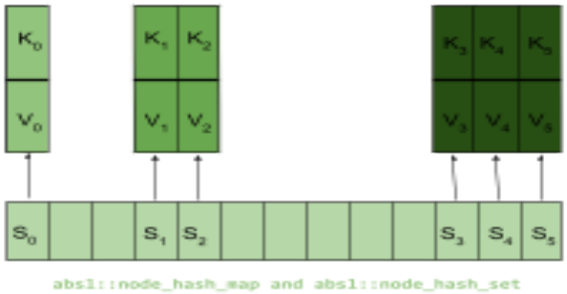
Flat: This is the "default" type. The flat type stores keys and values inline within the main array to avoid memory indirections. Avoiding memory indirections means that this type does not have the ability to reference something using a name, reference, or container instead of the value itself. In addition to avoiding memory indirection, the flat type does not have pointer stability because the table is rehashed when data is moved. Having no pointer stability means that values in the main array will be moved (rehashed) and that a pointer to a value does not remain valid due to the moving.



absl::flat_hash_map and absl::flat_hash_set

Node: Unlike the flat type, the node type does have pointer stability where the address of the object does not change. It has pointer stability because there is separate allocation of the value's type in nodes outside the main array. Despite this separate

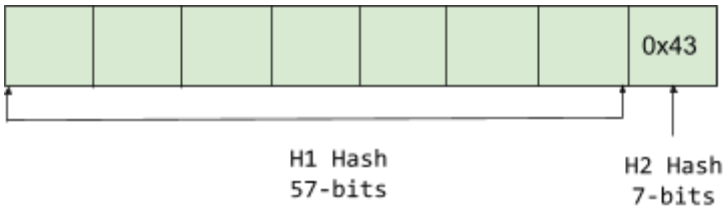
allocation, it only takes up 8 bytes to store the data value. Note on the diagram below that there is a separate node to hold the value type whereas the diagram above does not have a separate entity to hold the value type.



For more information, refer to this documentation.
https://abseil.io/blog/20180927-swisstables

Overall, these hash tables group elements into buckets of size 16, where each bucket can be packed as another small hash table with additional metadata probing that is stored separately (*more details on metadata below*). With this type of design, the operation of these buckets gets 2-3x better performance with significant memory reductions, compared to unordered maps.

How does it create a Two-Level Hash Table?



Each element has an associated hash function output. This hash function output is a 64-bit value. The 64-bit value is split into two levels.

For reference: One byte = 8-bits

Two components, that encompasses 64-bit hash value:

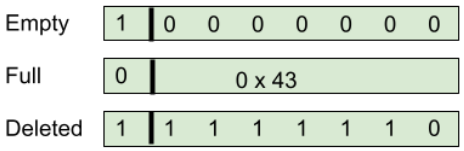
- H1 Hash
    - 57-bits size
    - Holds the position/index in array information

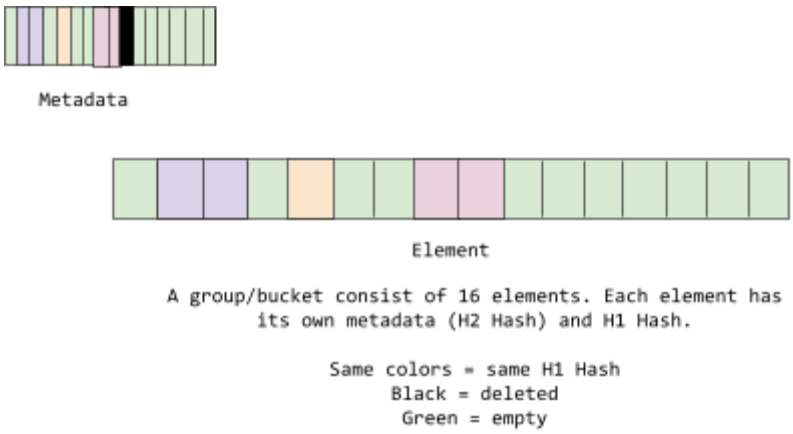
- H2 Hash
    - 7-bits size
    - Holds the metadata information
    - The element's key holds this H2 value
    - The metadata entry is 1 byte (8-bits) sized
        - single control bit
        - the 7 bit H2 hash

What is the metadata?

Like mentioned above, the metadata is stored separately, but in a parallel array. Within this metadata, it gives presence information about elements on whether it is empty, whether it is full, or whether it is deleted. To make the presence information work, the metadata uses a whole byte for each element known as the single control bit and 7-bits of hash code known together as the H2 hash.

A control bit can have 3 types values of:





For more information, refer to this documentation:

https://abseil.io/about/design/swisstables

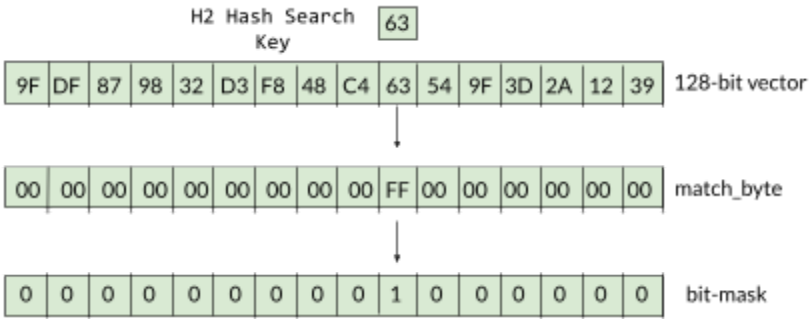
How Does Lookup Optimization Work and What is it's Runtime?

Short Explanation:

Searching the hash table, we can compare the search key's H2 Hash to each control byte. If there is no match, a value is 0. If there is a match aka a value of 1, the key of that H2 Hash is looked at.

Long Explanation:

- Each group/bucket is probed with its H1 Hash and the slots of each element are matched to its H2 Hash in parallel.
    - A 128-bit vector is made with 16 * 8-bit control bytes
- Make a mask from the H2 Hash called match_byte
    - Produces a 16-bit vector with values of 0 or FF if the bytes line up
- Make a bit mask to tell us which of the 16 control bit has the proper H2 hash
    - Values of 0 if they do not match and 1 if they do match



For the lookup, insertion, and deletion implementation, the runtime is O(1) with a faster experimental runtime compared to `std::unordered_map` and `std::unordered_set,` because it can narrow down the 16 control bit-mask to those with matching H2 Hash and "probing is done by incrementing the current bucket by a triangularly increasing multiple of Groups: jump by 1 more group every time. So first we jump by 1 group (meaning we just continue our linear scan), then 2 groups (skipping over 1 group), then 3 groups (skipping over 2 groups), and so on" (Beingessner). Another reason why the SwissTables are faster is because the 128-bit vector created can fit into a L1 cache line. This means that we can probe an *entire* group really quickly, before having to fetch another group.

For better details, refer to *CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"* and this document for insertion and deletion implementation.

https://www.youtube.com/watch?v=ncHmEUmJZf4

https://gankra.github.io/blah/hashbrown-tldr/

What are Some Tradeoffs?

- Gain:
  - Avoid memory indirections
  - Less RAM used compared to `std::unordered_set`
  - If an element is not in the set/maps, the Swiss tables do not go into the values (the slots). It will mostly stay in the metadata (control bytes). Due to this, runtime is more efficient in both small payloads (4 bytes) and large payloads (64 bytes), compared to other data structures like `dense_hash_sets` and `std::unordered_sets`. (Refer to figure 7)
- Loss:
  - Loss in pointer stability in the flat hashmaps/sets
  - For small payloads, such as finding an element in 4 byte, the efficiency of Swiss tables is not optimal due to the spending of 1 byte for the metabase probing. Due to this, other data structures like dense_hash_sets slightly outperform, specifically if the element is a "hit". (Refer to figure 7)

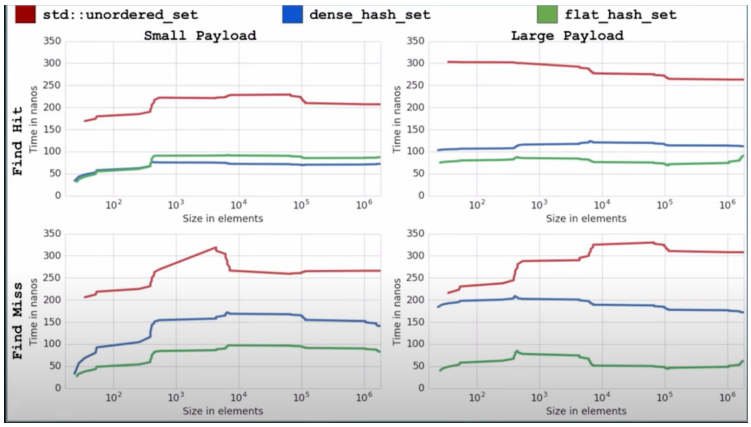


Figure 7: Runtime Efficiency

Courtesy of CppCon 2017: Matt Kulukundis "Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step"

How is it Memory and/or Cache Friendly?

Swiss tables are memory-friendly because they use less memory overhead, which means that the tables have a higher max_load_factor (how full can the table get before it grows to a bigger array). Additionally, the Swiss Tables has only 1 byte overhead per entry, as seen with the H2 Hash (1 bit for control bit and 7 bit for metadata), therefore for big payloads, the experimental runtime outperforms other types of hash tables as seen in Figure 7. Furthermore, from the discussion in `How Does Lookup Optimization Work and What is it's Runtime?,` we can find results from 16 slots in little as 3 CPU instructions. Lastly, the optimization is done via SSE instructions. Since modern CPUs support SSE instructions, it means that some operations of multiple values can be done at the same time in a processor.

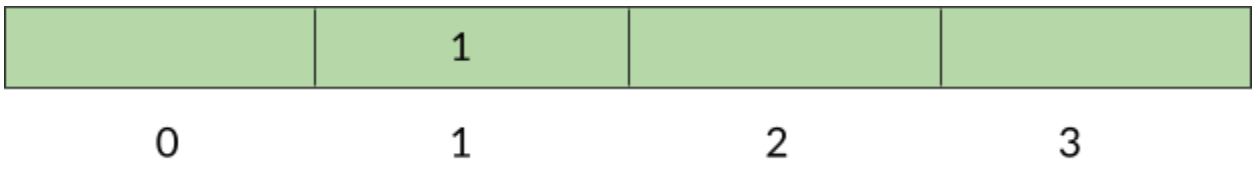
In terms of cache-friendly, the Swiss Tables fit the criteria due to the expense of not maintaining reference stability as seen in `What is the difference between flat vs node hash_map/hash_set?`

For more information about SSE instructions, please refer to this documentation: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

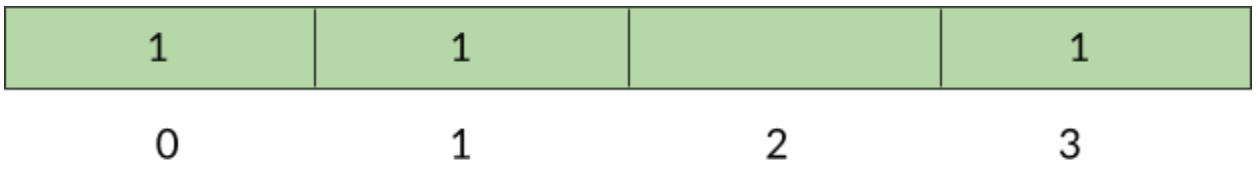
Algorithm #2: Bloom filters:

What are Bloom Filters?

A Bloom Filter is a way to hash an object and create an array to see if we have seen that object before. A quick example of this would be using strings with a hash function that is based on the length of the string. Let's say we have an array that is 4 wide. Our hash code is now the length of string % 4. So let's add a few words into our data structure. We will add "Hello", "Fine", and "And". First we start with hello, which will give us a hash code of 1. We, then, set the 1 bucket to 1 or true.



For the words, "Fine" and "And", we get the hash codes of 0 and 3 respectively.



Now that we have inserted those three words into our bloom filter, buckets 0, 1, and 3 all have been marked as being seen. If we make a request for the word "As", the Bloom Filter will tell us that it doesn't exist in memory because bucket 2 is not true. This is the advantage of a Bloom Filter. We are able to easily and quickly tell if something is not in memory.

Now if we make a request for the string "Hello", we would go to bucket 1 and see that it was marked 1. This doesn't mean that the word "Hello" is stored in memory, but there is a possibility that it is. This may sound weird, but the reason that it is a possibility is because the bucket for 1 can contain many different words. Say we want to know if the word "Apple'' is stored in memory. Well according to our Bloom Filter, it could because

the bucket 1 contains a true. We call this a false positive. This is the main downside of using a Bloom Filter. When we use simple hash codes or small arrays, we begin to see a lot of overlap between words. For our table above, our false positive rate is 75% because 3 out of 4 of our buckets could return us that something may exist in memory, when it really doesn't.

How do we mitigate the effects of these false positives? One leading method is to use multiple hash codes. By using multiple hash codes, we can cut down on the false positives that fall into one bucket by having to have multiple buckets be true. Another method is to make sure that our hash codes will evenly distribute whatever we are storing evenly among our buckets. Having little overlap will cut down on the rate of false positives.

The runtime of a bloom filter will depend on the hash function or functions used. For the runtime complexity of a single array bloom filter, the runtime will be O(k), where k is the runtime of the hash functions. The space complexity will be equal to the amount of indexes in the array. So if m is the amount of indexes in the bloom filter array, then the space complexity will be O(m).

You might be asking what are some of the uses for a filter like this? Well one easy example would be checking usernames. As users sign up for your website, a bloom filter begins to make a log of what names exist and which don't. Then when a user wants to login, when they enter an incorrect user name, there is no wasted time to look up the non-existent username. There may be some false positives, but if the hashing is done efficiently, the false positive rate should be small enough that you save time, space, and computing power by throwing out the bad requests for usernames.

What are Some Tradeoffs?

- Gain:

    - Small and efficient storage element
    - Quickly tells you if you need an element may exist in memory
    - Cuts down on wasted cycles looking for information that doesn't exist
- Loss:
    - We lose the ability to know if something exist in memory for sure
    - We get a false positive rate that may end up slowing down our system
    - We have to use advanced hashing techniques to control the false positives

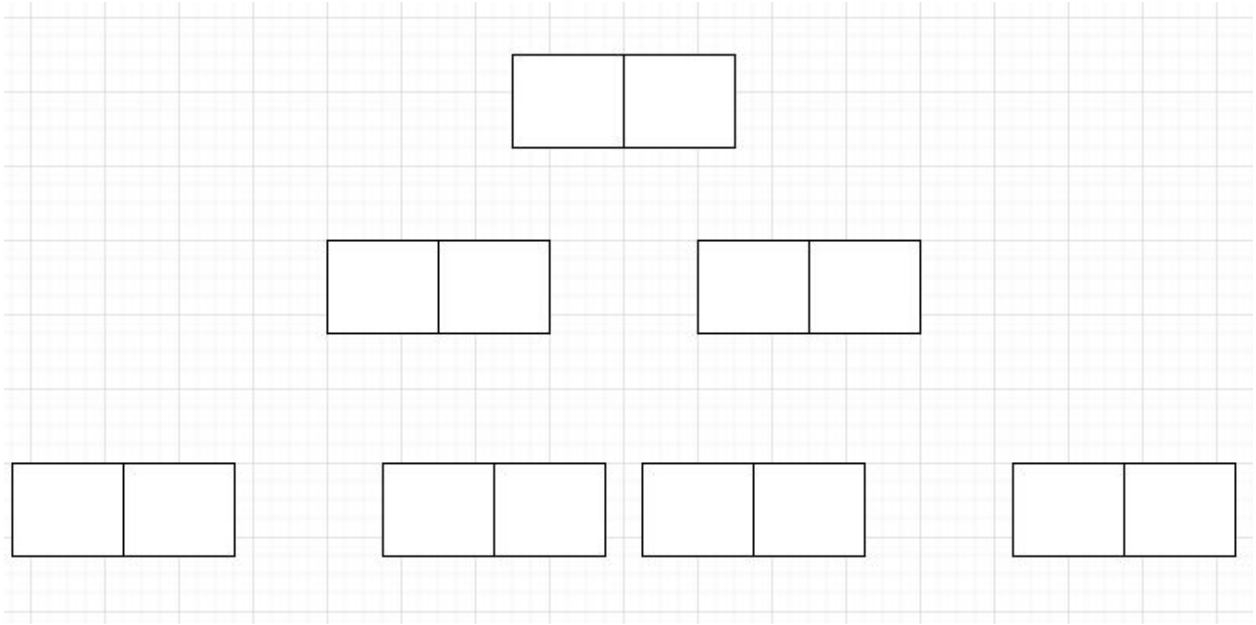
How is it Memory and/or Cache Friendly?

A Bloom Filter is cache efficient because it is a way to find out if an element exists with a 1 or 0 implementation. The overall memory used would be the size of the bloom filter array and the storage of a 1 or 0. This comes with it's own set of problems like having a false positive as a possibility. The idea is that a Bloom Filter would help with minimizing the amount of work needed to know when we need to access different pieces of data. We solve the problem of needing huge pieces of cache memory to store dictionary type data structures, and instead use a small table to know if the information requested even exists. In this way we cut down on memory accesses and the need to store a huge data structure in cache.

Learning target

We are under the assumption that this recursive hash table data structure has good hash functions and is balanced, the runtime is **O(log(N)/M)**.

The idea of a recursive bloom filter for collision resolution follows the pattern of a binary tree. If the elements of the recursive bloom filters are distributed evenly, then the recursive bloom filters will grow in the same way that a balanced search tree grows. For example, we would have something like the following:



In the diagram above, we have a fixed bucket size of 2. If we are evenly distributed, then the first two elements would only be in row 1. Then up to 6 elements, we would have 2 levels. When we reach 14 elements, we would have 3 levels. Each level would include a new hash function, but we are assuming that hash functions run in constant time. This would continue to grow in the fashion of a balanced binary search tree. Since we are growing in the same way, we can equate the asymptotic runtime to that of a balanced search tree which is O(log(N)). The difference with the bloom filter is that we have a different amount of buckets at each level. As the number of buckets increases, the runtime will decrease for searching because each level can contain more elements. This would give us a runtime similar to O(log(N)/M), where M is the fixed amount of buckets. Since we are looking at asymptotic runtime though, then the M will become insignificant as N goes to infinity, so our worst case asymptotic runtime simplifies to O(log(N)).

Algorithm #3: Suffix Arrays:

What are Suffix Arrays?

A Suffix Array is a way to quickly see and find the location of a suffix inside a string. This is very useful for situations such as DNA indexing where searching for specific suffixes in a large string of characters is done repeatedly.

For an example, let's say we have the DNA sequence "GCATCGC". For setting up our suffix array, we first need to make an array of size N (N in this case is the length of our original string). Inside this array, we store the indexes of all possible suffixes of our string in alphabetical order.

- List of suffixes are:

  Index 0: GCATCGC
  Index 1: CATCGC
  Index 2: ATCGC
  Index 3: TCGC
  Index 4: CGC
  Index 5: GC
  Index 6: C

The suffix array after sorting starting indexes alphabetically:

2	6	1	4	5	0	3

Now we are all ready to start searching for suffixes inside our string using our array. To search for a suffix we binary search through our array, comparing the suffix we are looking for to the to the index inside our original word "GCATCGC" and decide whether to go left or right by comparing alphabetically. We then continue binary searching through our array until no more matches can be found. When a match is found, to ensure we return all possible matches, we shrink the high and low pointers of our binary search until they also match our query. Due to our array being sorted

alphabetically, if our high and low pointer both match the query then all suffixes in between them should as well.

Walkthrough of Searching for suffixes with a Suffix Array:

If we want to search for the suffix “GC”, we point at the high, middle and low index of our array. We then compare the suffix starting associated with the value inside our middlepointer alphabetically to our queried suffix. In this case the value our middlepointer is looking at is 4, so our high suffix to look at is “CGC”. We compare “GC” to “CGC” alphabetically and it tells us to head right. We then move our high pointer to the right of our middle index and set our middle pointer equal to the mid point between our high and low pointers indexes. So the midpoint of 5, 0, and 3, which in this case is 0. We compare “GC” to “GCATCGC” and find there is a match! Despite finding a match we continue as there can be more than one suffix match. We compare the high and low pointers to our queried suffix, moving them closer to our middle pointer if they don’t match and leaving the pointers there if they do. In the end we return all the values in between our high and low pointers which would give us 5 and 0 as the starting indexes inside “GCATCGC” where the suffix “GC” is located.

Runtimes:

For a suffix array the construction of the array takes $O(N^2log(N))$ time with N being the length of our string. The reason it is $O(N^2log(N))$ is because comparison-based sorting takes O(Nlog(N)) comparisons and a suffix comparison in the worst case can be O(N) if the suffix is length N. This gives us a total runtime for construction of $O(N^2log(N))$. This runtime seems very slow but the benefit is that it only needs to be constructed from the string once, but allows us to search for suffixes in the future much faster. This runtime is also only based on comparison-based sorting algorithms but can situationally be sped up by advanced sorting algorithms made for the specific problem.

The main benefit of suffix arrays comes with their search time complexity. Suffix arrays have a worst case search time complexity of O(kN) because if every suffix is considered a match that gives us N run time (one comparison for each element) times k comparisons (one extra comparison per step for each letter inside our query k). This only happens when every possible suffix and a match such as the case would be with the “G” suffix and string “GGGGGGG”. The in-practice search runtime generally comes out to O(klog(N)) instead because we are using binary search which has O(log(N)) time complexity and our query is length k so it takes k comparisons in the worst case at each step to check for a match.

How is it Memory and/or Cache Friendly?

A Suffix Array is cache-efficient because it only stores the indexes of the suffixes and not the suffixes themselves. It ends up taking O(N) space due to storing a single index for every character our original String contains. This helps save on space even compared to similar structures like Suffix Trees. Suffix Trees are a very similar concept to our ternary search tree that we used for TernarySearchTreeAutocomplete. Suffix trees take up more space due to storing every single suffix as well as their indexes. A suffix array by comparison only stores the indexes the suffixes are located at, cutting down on space used and ultimately making it have linear space storage with respect to N (O(N) space complexity).