Memory and Cache-friendly Algorithms

Ara Vardanyan, Tristan Barr

March 2021

Motivation

Reducing runtime by a small margin may have little implications on one program but on a world scale where we have billions of programs running at the same time, the ways we design our fundamental data structures and algorithms can have major implications on everyday life. Milliseconds of runtime efficiency may not mean much when you're doing a quick Google search but they can be the difference between life and death in many applications of computation. For example automated medical diagnosis, environment processing for self-driving vehicles, delivering time-sensitive security content, and more. Our work as software engineers has major hidden implications on the world and it is our responsibility to recognize this and use our skills and knowledge to improve welfare through computational efficiency.

Today, sharing information online is a backbone of society. Most people's lives are dependent on the timely delivery and processing of information by automated systems, no matter their profession or lifestyle. Streaming movies, searching online, or using digital maps all rely upon the efficiency of data transmission/retrieval. The efficiency of these information systems relies heavily on the ways in which information is stored, and accessed.

Caching is a technique by which information systems can effectively access data more rapidly by storing data closer in terms of lookup expense depending on it's significance. This significance is determined through algorithms which take into account frequency and recency of access, location of data and more. Caching is incredibly powerful, as this can be the difference between receiving information in a few minutes compared to a few seconds (when working with large enough data sets). Developing cache-friendly algorithms can tremendously improve performance of information retrieval, which is extremely important especially considering how much we all rely on the timely delivery of information.

Another key motivation for this blog is to examine memory-friendly algorithms. These algorithms can basically be characterized as algorithms that preserve as much memory as possible. Knowledge is power, not only to humans but also computers. With hardware being a limiting factor in terms of memory storage, designing our algorithms to be more memory-friendly can greatly optimize our devices and computational capabilities without us having to make breakthroughs in physics and hardware design research. Furthermore, saving on memory saves on money and resources. At big companies such as Google and Microsoft, saving memory can help save millions of dollars.

Throughout this course, we have learned how data structures and algorithms allow us to store, access, and manipulate data in different ways, each with a certain degree of cost in terms of runtime, affordance, security, etc. Our goal is to create the best possible way to store and retrieve data for any given task. Designing memory and cache-friendly algorithms can help us achieve this goal by

reducing latency between the solutions computation brings and tangible welfare in our society.

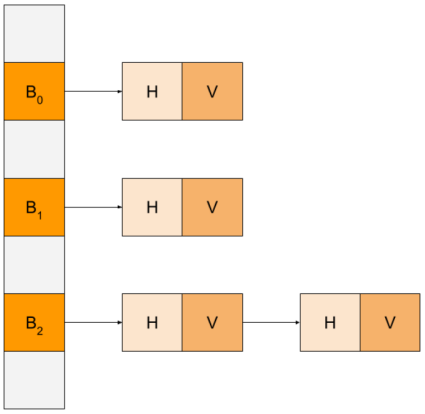
Our interest in these cache and memory friendly algorithms has been particularly intensified due to the current COVID-19 pandemic. The lives of our family and friends, not to mention our own, have been increasingly reliant on the successful updating of information from global and national institutions. Information about case counts, COVID hotspots, and safety guidelines have saved millions of lives. The sharing of medical research on vaccine development, and antibody testing has further improved our capacity to fight off this deadly virus that otherwise had the potential to take even more lives. If there was ever a time to get interested in the optimization of widespread information sharing, it would be now.

This blog will take a look at a few different data structures and the algorithms behind them, examining each from the perspective of efficiency of memory access and memory usage. We will discuss the practical applications of each, along with the benefits and drawbacks which pertain to each different structure.

Swiss Tables

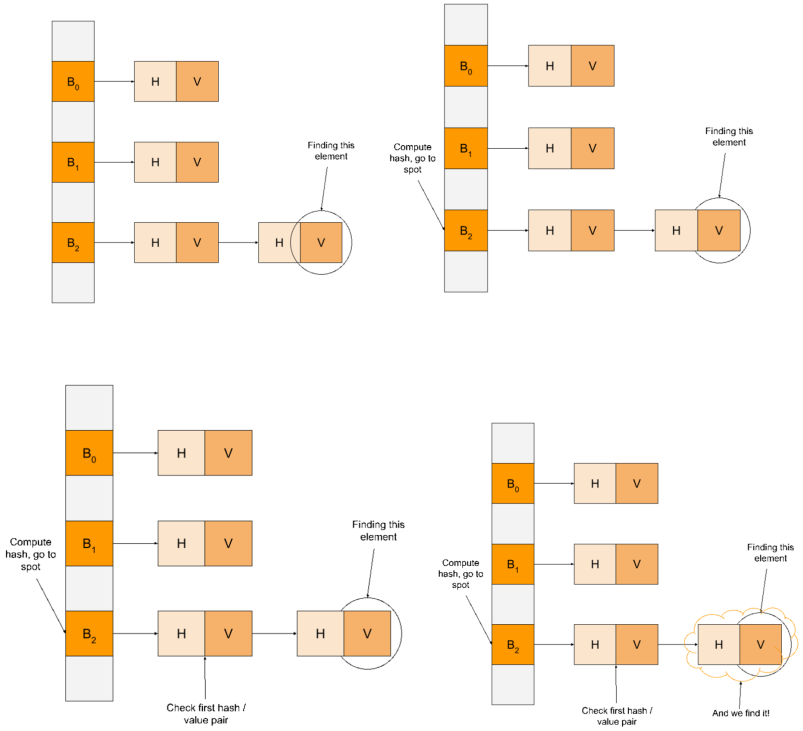
We really enjoyed our time researching Swiss Tables. Here we will explain exactly what Swiss tables are, and how they work.

First, let's look at a typical unordered set:

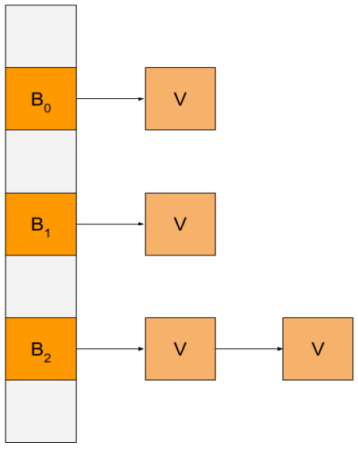


The typical structure for an unordered hash set above is what we are used to seeing. We observe that there are buckets which map to values (V). The buckets in which these values end up in are determined by their respective hash values (H), which is computed from a hash code. Alright, now that we have the basics, let's take a look into how much memory a single value entry requires, and how many memory accesses it takes to get to a given value.

We see that for each value we insert into the hash set, we need to also store a hash value. This requires extra memory. We also notice that there are pointers from one value to another. These pointers also require a little bit of extra memory. Let's look at what it takes to find a single value in this table. We have to compute the hash, follow that hash to the given bucket, and then search through that bucket for our given element. Accessing the given element in this case requires 3 memory accesses, which is shown by the figure below:

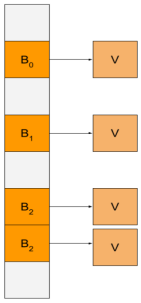


Let's see if we can't make this a little more memory efficient and quicker to access. How about we try to cut down on the memory first? To do this, we can get rid of the hash value. This, coincidentally, also reduces the amount of memory accesses required to get a certain value since we do not have to traverse through the hash value in memory. Here is our new structure:

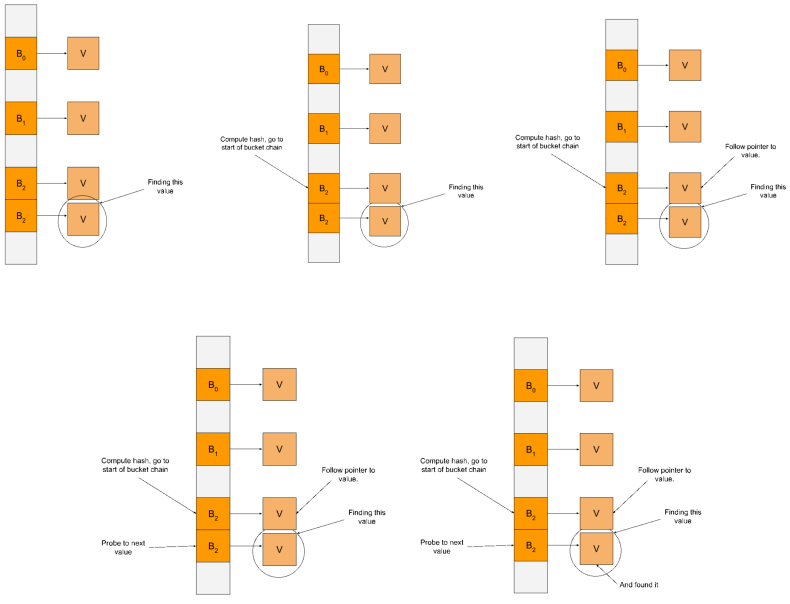


At a company like Google, this would save about 0.1% of fleetwide RAM. That's a very healthy amount of RAM. Who knew hash values held so much memory?

Let's take it a step further. We can get rid of the pointers from one value to another. Let's try it:



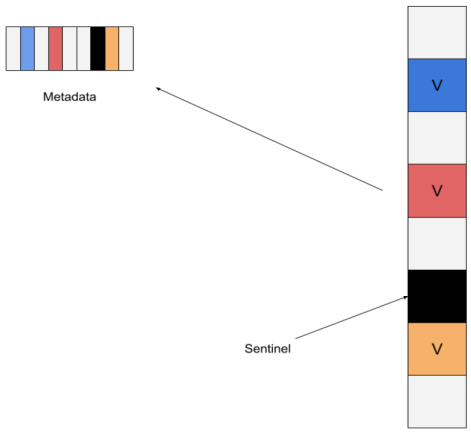
This is called a probing table. Let's see what it takes to find a given value in this structure. We compute the hash and go to the bucket. The first value isn't it, so we probe to the next spot, and we find it! This only took 2 memory accesses compared to the previous 3. Note also that the "go to the next spot" step was basically computed for free since these values are right next to each other in memory. Here's what it looks like visually:



Basically what this boils down to is reducing memory indirection, or accessing parts of memory that we don't need in finding our final value. But, notice that we are still exploring elements that we don't want to. These extra explorations cut down on performance.

Is there a way to get even better performance than what we have already accomplished? How about we place the values themselves directly into the array where the buckets should go. Let's take a look, and note that we changed the values to be different colors so we can tell which values belong in the same bucket:

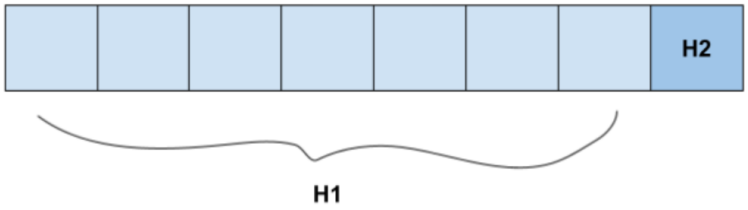
This is called a dense hash set. There are a few issues with this, however, one big issue being that if a value has been deleted it will still be there afterwards! These deleted values that still hold a place in memory are known as sentinels. To avoid accessing these (which would reduce performance), we can use metadata to store presence information about a given element, i.e. if it's full, empty, or deleted. Let's store this metadata in a parallel array so we know which elements we shouldn't access. Now, how about we delete the first element in the third bucket:



Notice how our metadata array now reflects that one of the elements has been deleted. Metadata is essentially just data that is used to describe other data. Note that the metadata in our parallel array maps to the indexes in our table. Very handy!

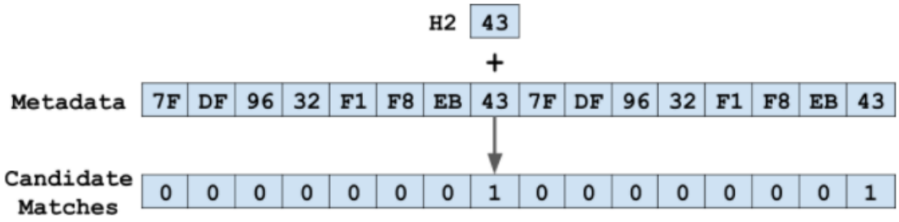
Now, notice that we basically want to keep track of 3 things: whether a given element is full, empty, or deleted. In order to fully represent this, we would need 1.5 bits of metadata (since we can keep track of 2 things with 1 bit, in order to keep track of three 3 we would need 1.5 bits). This is annoying, since we can't get away with 1 bit but we won't fully utilize 2 bits. Swiss tables employ a brilliant idea, which is to use a whole byte for metadata, use one bit as the control bit to say whether or not the element is empty or deleted, and the remaining 7 bits for the hash code. Why is this clever? Well, we can put the hash code in the lowest 7 bits of the metadata byte, and then use the first bit for comparisons of whether the given element is empty or deleted or neither. This will cut down on comparisons, because we won't check the value in memory if the value for this metadata does not match. Cutting down on comparisons is good, because it cuts down on memory accesses, improving performance.

We still need to know where to go in our overall big array. We can simply define our hash function to produce a 64-bit value. 57 bits is used to store the index in the table itself, we call this the H1 hash or just H1. H1 is what we normally take the modulus of when computing hash values. H2 is the second portion of the hash value and contains metadata for a given element.
In order to find an element in this table, we use the H1 hash to find the correct bucket chain to go to, and then we use the H2 hash for comparisons in the metadata. This is really nice, because now we won't have to compare the values in the memory directly since we can do comparisons in the metadata. This eliminates memory accesses, improving performance.

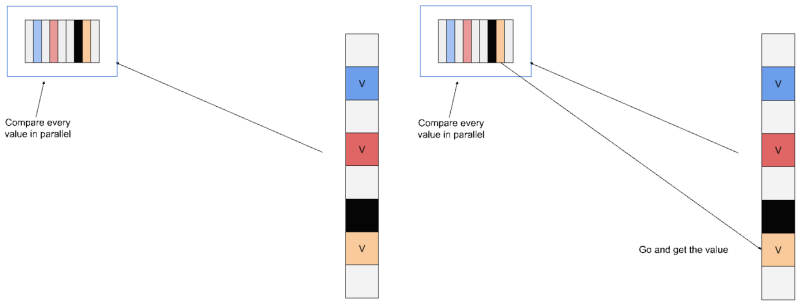


But this still is not good enough! How can we make this even faster to find an element? We can use SSE instructions. What are SSE instructions? We are not going to go into a ton of detail on this, but basically SSE instructions give us a way to examine all elements in the metadata in parallel (at the same time) by using a bit-mask. The SSE instructions will produce candidate matches from our given mask, since matches to the mask will produce 'FF' (hexadecimal for 11111111) if found or 0 (can be thought of as 00000000) if not found. The SSE instructions produce an array of values either 1 or 0 (1 for a match, 0 for not a match), which tells us whether or not the given element had the proper H2 hash! This can give us a 16 bit probe length (compare 16 values) in about 2 instructions, which is very efficient.

This image is taken from the abseil design notes on Swiss tables, and is a good visual representation of how this works:



We get 1's wherever a candidate matched our query, and we did it all at once. Let's see what finding our element looks like now, using SSE instructions:



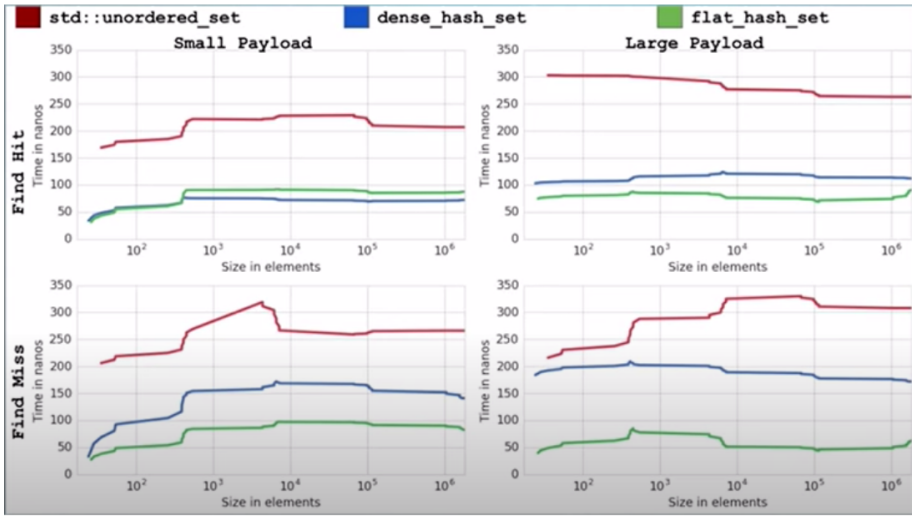
Let's put the picture above into words: we compare all elements in the metadata in parallel, and then we jump directly to the matching element. Notice that the only memory access we needed was to retrieve the matching element. So this took a single memory access, which is awesome.

And voila, this is a Swiss Table. It actually goes by the name of "Flat Hash Set" at Google but we like the sound of Swiss Table better. This hash table is incredibly efficient because it can examine a ton of bits at once using SSE instructions. Furthermore, these comparisons are made in the metadata so most of the time we do not even need to directly access memory. We also saved on the overall RAM our table takes up, since we got rid of hash values being attached to each value in a given bucket chain, and we removed a memory indirection by placing the values themselves directly into the table (by putting it in vector form). So not only did we improve performance, we also decreased overall memory usage!

Let's recap the steps here:

1. Compute the hash (H1) and go to the start of the bucket chain in the table.
2. Create the mask from the H2 hash.
3. Use SSE to produce the 0's and 1's.
4. Check all of the values that produced 1's.
5. If the element desired is not found, probe to the next group of values in the table.

And this is pretty dang efficient. Let's look at some data presented at 2017 CPPcon in Bellevue, Washington to see how efficient these tables are:



We can see that Flat Hash Set is more efficient than unordered set in all scenarios pictured, and more efficient than dense hash set in most scenarios pictured. What can we observe? We see that the extra control bits used by Swiss Tables on very small data sets actually just slows down performance since one or two bits carries a lot more weight when looking at 4-bit elements (for example). We also see the power of keeping this metadata when we search for a value that is not in the table (find miss). We see that Swiss Tables are much faster in this category, since if the value is not in the table we can figure that out immediately from the metadata and do not even need to access values in memory.

So, what are these not so great at? We see that these tables are very good for improving performance and decreasing memory usage but this, of course, has limitations. Since we removed the pointers and the chaining for this table, we no longer have pointer stability. Pointer stability states that no matter what

happens to your table, the values will not move. You will not lose values by modifying your table. This is useful for when you would like to mutate a table a lot. Swiss Tables, then, are very much NOT good for applications where mutations on the table occur frequently. This could mean reusing your tables, or repeatedly deleting and iterating. Another limitation is one that we observed in the graphs, in that when working with very small data sets, Swiss tables can be slower since the bits used to store metadata can take up a significant amount of space relative to the size of the data.

Another thing to consider is what kind of hash function you are using to compute the hash values. If the hash function has most of its entropy in the first 7-bits, we will get a lot of H1 collisions, and if it has most of its entropy in the last 7-bits, we will get a lot of H2 collisions. Remember that collisions occur when we find two values to have the same hash function, and this is a bad thing. We need to ensure that we have a hash function that distributes things evenly when using Swiss Tables.

Regardless of the limitations, Google has created an incredibly efficient alternative to the conventional unordered hash set, one that both improves performance and decreases memory usage in the most elegant of ways.

Bloom Filters

Bloom filters are a relatively computationally inexpensive way to add elements and check whether elements are in a set. Bloom filters allow us to reduce expensive lookups for non-existent keys through the use of hash functions and a bit vector set to a desired size.

Adding an element:

1. Feed the element to different hash functions which output positions to set bits at.
2. Set bits accordingly at the resulting positions in the bit vector.

This allows us to test if an element is in the filter very efficiently.

Testing for an element:

1. Feed the element desired to be tested to the same hash functions.
2. Check if any of the bits at these positions are not set.

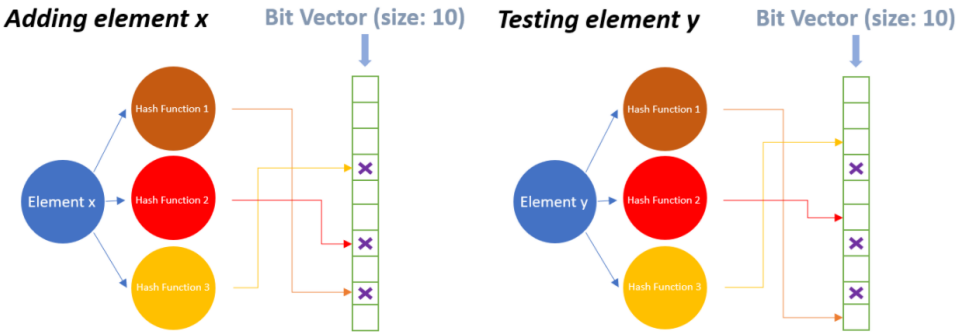


Figure 1: Adding and Testing elements in Bloom Filters

In this case we see that when we tested for element y, the bits at the positions given to us by our hash functions were not set, so we know that the element y is definitely not in the set. Since we know that the element is definitely not in our set, we effectively avoid having to perform an expensive disk or network lookup.

Bloom filters are not perfect however. Sometimes hash functions produce overlapping positions causing collisions. This is why we receive an output of "probably in the set" instead of "definitely in the set" when we test an element and see that all necessary locations on the bit vector are marked. The false positive rate can be decreased by adding more hash functions at the expense of lookup

time or increasing the size of our bit vector at the expense of memory usage. Thus, our number of hash functions and the size of our bit vector will depend on the significance of false positives to our application and significance of runtime and memory usage.

If the performance metrics of our application does not strongly depend on accuracy, we can decrease the number of hash functions to maximize lookup efficiency. Furthermore, if we have plenty of memory but weak computational power, we can increase the size of our bit vector.

Looking at runtime complexity of inserting and "finding" for this structure, we notice that performing hash functions is typically free so we can say that computing hash functions is simply constant O(1). Since inserting and finding are only dependent on the time it takes to compute our hash functions, we can say both have a runtime complexity of O(1). This is very efficient!

One drawback of bloom filters is that they do not support deletion, since we are not guaranteed to find the value we want and we won't know if one index holds information for more than a single value. So, if we want to delete an element, we will have to delete the entire table and restart, which can be incredibly inefficient.

How about memory usage? Well, we are only storing 1's and 0's in our bit array, so this means that we are guaranteed to store less things in memory than storing the element itself. This makes bloom filters the most memory efficient data structure discussed in this blog.

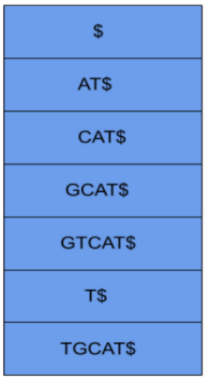
To summarize, bloom filters give an incredibly efficient way to insert and "possibly find" an element from a given data set. This means that if we have an application where we can potentially sacrifice accuracy for runtime complexity, it is a very valid choice. Furthermore, since these tables only store 1's and 0's we will use less space than storing elements themselves in memory. However, we cannot delete a given element, nor can we rely on a query match as an absolute confirmation a given item is in the data set.

Suffix Arrays

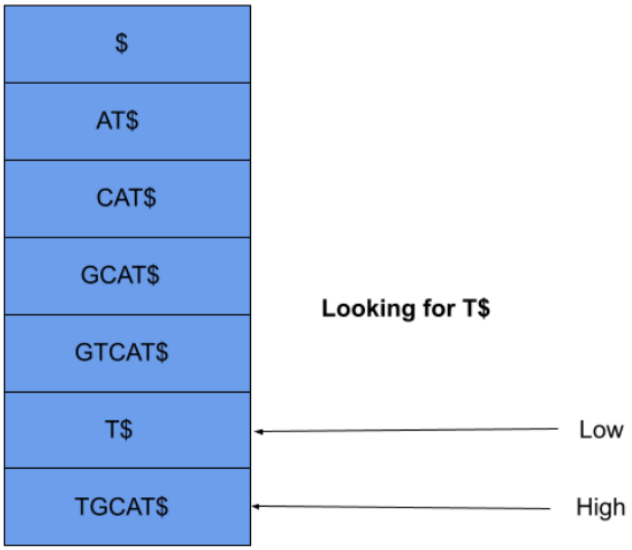
Suppose we have a very large data set composed of sequences of letters. What if we want to get the values of that data set that begin with a certain substring? For instance, the aforementioned data set is composed of DNA sequences, and we are looking for a particular DNA sequence. How can we accomplish this in a reasonable amount of time?

What if we try to find all of the matches using multiple linear searches? Well, for data sets that incredibly long, this would be very inefficient. Moreover, the more elements we want to search for, the slower it will be, since each new query will require searching through the entire data set. How about we find a quicker way.

This is the motivation behind suffix arrays. If we can create an array that stores the starting sequences to each suffix in lexicographical order, we can directly access the desired sequence. This is a lot more efficient than looping through the entire array, comparing letter by letter, since we only need to search the indexes that show promise of matching our given query. But this still is rather annoying, since the worst-case runtime of this would be the exact same as the original data set linear search to find a match. This worst-case runtime would be a case where our query matches every single suffix in our array, and we would have to search every single letter in every single index. Do we know of a way to guarantee we find our value from an array in a better runtime? Binary search, of course. How could we make binary search a valid method for computation here? Well, in order to perform binary search, the array has to be sorted. So, we can alphabetize an array by the first letter for each grouping. Here is what this would look like for the String "GTGCAT":

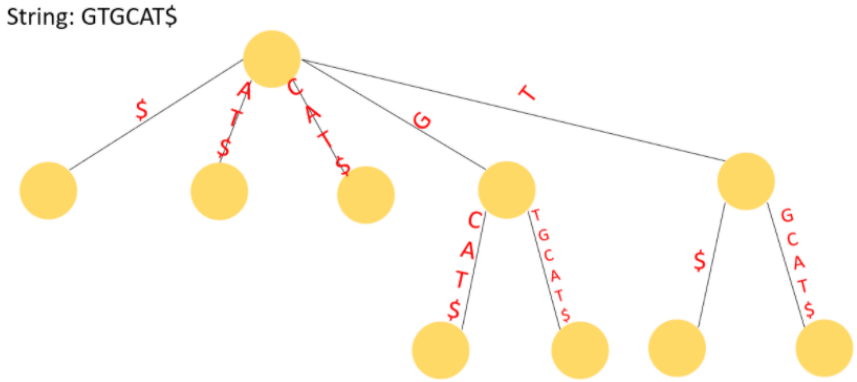


Now, we can do binary search as we normally would. That is a whole lot more efficient as we we are now guaranteed to find our value from this array in O(log(n)). But what if we have multiple matches for our given query? Binary search is only going to give us a single match, and we don't even know if that match was the first one that appeared in our data sequence. How do we solve this? We need to get a range. Since our array is sorted, we know that all values that match our query will be next to each other. So we just need to find the first time our query appears, and the last time our query appears. Here's what we are talking about visually, this example is looking for "T":



The way we find the low and the high points is by running two binary searches. One finds the low point that is equal to our index being equal to the query and also our index minus 1 being not equal to the query, and the second finds the high point where our index is equal to the query and our index plus 1 is not equal to the query. This effectively gives us the ranges of indexes that gives us our desired query, and we can extract our matches straight from the initial data set using the indexes.

Suffix arrays also have tree correspondences. Here is the previous example in Suffix Tree form:



Searching this suffix tree would be quicker than a linear search through our suffix array, which we will prove later when discussing runtimes. The problem with a tree structure however, is that we have used up a lot of space in constructing the tree. This is one of the main things we are trying to limit with our memory-friendly algorithms.

Instead of using a tree representation for quick searching purposes, we can instead run a binary search on our suffix array. This helps us avoid the memory issue created by using a tree and allows us to accomplish the same goal. In our suffix array, we do not actually store the suffix for each given index, we only store the corresponding indices at which the suffix can be found in our original data set. This allows us to avoid constructing additional fields, in turn saving memory.

Let's take a look at the runtime for this. So we perform binary search in $O(log(n))$ time. Remember, though, that we had to sort the entire thing first. So the preprocessing took $O(nlog(n))$. If we want to find matches for a given query that is k letters long, we will have to compare all k letters to guarantee that we have found a match. So, we factor that in and get a runtime of $O(knlog(n))$. Let the number of letters that we have to compare be n, and we get $O(n^2log(n))$. Finding an element in a suffix tree would take less time, since we would only actually need to search for the value we would be able to find any value in $O(k)$ where k is the number of letters we have. Note that this is if we ignore the preprocessing step. Let's ignore our preprocessing time for simplicity and, again, take k to be roughly n, and we get a finding time of $O(nlog(n))$ for our suffix array, and $O(n)$ for our suffix tree.

One downside to suffix arrays is that we still have to compare every letter found in our query to the suffix at the given index, which factors into runtime (if only

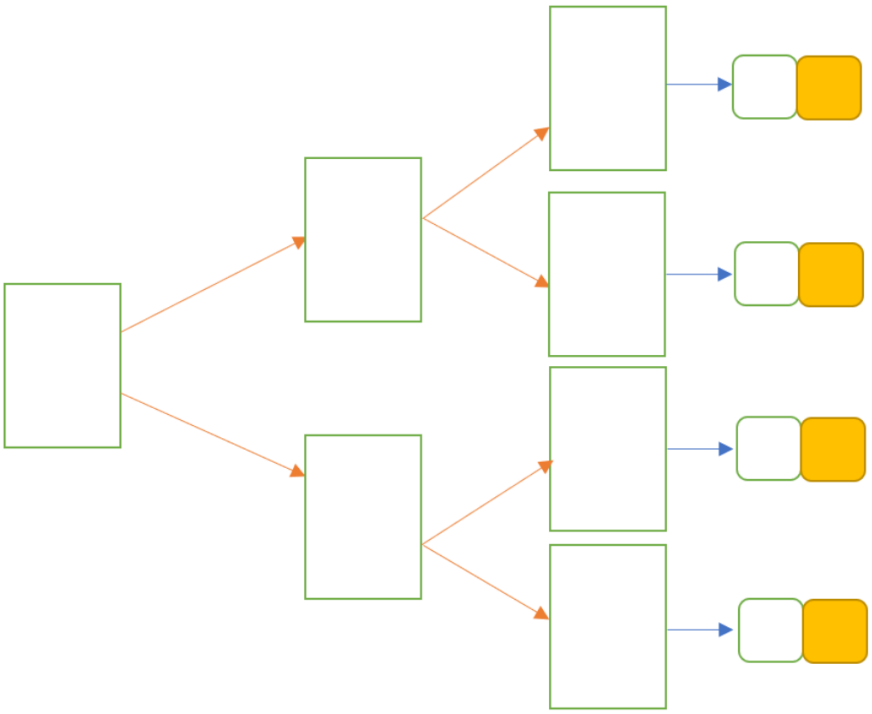
we could use SSE instructions to compare a wide range of suffix values at once to our query like in Swiss Tables, sigh). Another disadvantage is that suffix arrays will require $O(nlog(n))$ to be constructed, which can be more computationally expensive than some other data structures.

In summary, a suffix array can drastically speed up the amount of time it takes to search a very large data set for a given string sequence since the values for where to look in the data set are precomputed. We simply need to perform two separate binary searches on our suffix array (comparing each suffix in the data set to our given query as we go) producing the range of indices we need and use the indices stored in the suffix array to retrieve the values. Furthermore, suffix arrays improve on suffix trees by reducing the amount of space required in memory.

Learning Target

In this section we are going to combine a few of the ideas we have gone over. We are going to try and create a hash table with multiple hash functions (like in bloom filters) and create a tree-like structure with each bucket in the hash table mapping to another bucket and so on and so forth, until we have run out of hash codes to compute. Would this really be faster? For our purposes, we will say that every hash function evenly distributes all N elements, but has a fixed number of M buckets. Let's try to unpack all of this and get a worst case runtime analysis.

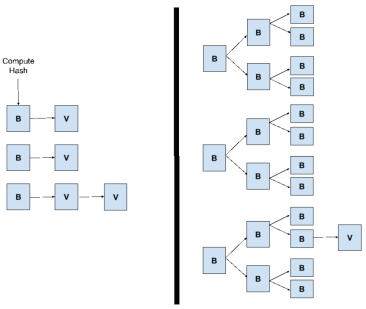
Let's start with an image of what we have:



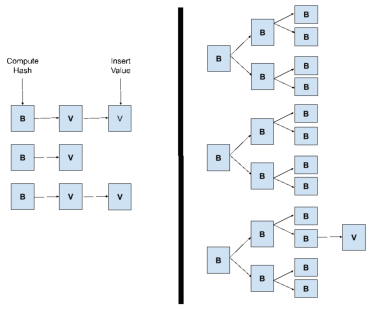
Notice that the only things each bucket contains is a link to another bucket, with exception to the last bucket which actually stores the elements. What does it take to insert an item into this structure? Well, we have to compute all of the hash functions that we have. If our number of hash functions is M, then it would take M computations to find the bucket the element belongs to. If we think back to the runtime of insertion for a typical hash table, we recall that typical insertion time is just constant ($O(1)$).

In the images below, it's easy to see that adding an element into this table will be slower than your typical hash table, the convention here is that B is used to indicate buckets and V indicates the values themselves:

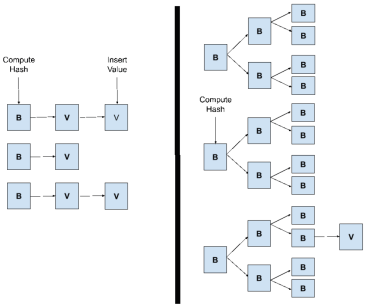
[1]



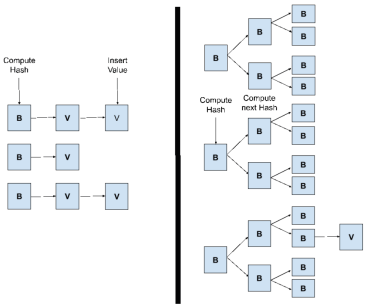
[2]



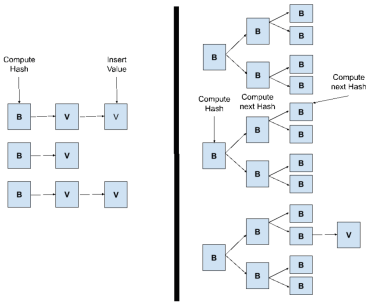
[3]



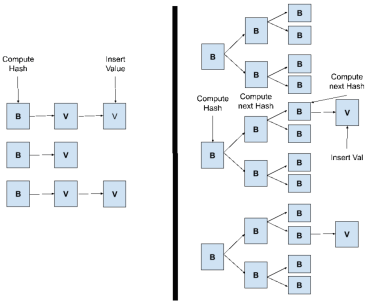
[4]



[5]



[6]



How much slower would this be? Well, it depends on how computationally expensive your hash code is. If it is something simple like an int or a double, the difference isn't noticeable, but if it is a long, complicated hash code then it would be noticeably slower. We will touch on this idea later, but, for now, let's get back to the analysis.

So, not a great start, but what about finding an element? In a hash table, the typical time complexity for finding an element is constant $O(1)$. Can we match it? Well, we have to compute the hash in order to find our value, but the problem is that we have M hash functions to compute. So we, again, need to do $O(M)$ work. Then we iterate through our bucket chain until we find our element or run out of space ($O(N)$). Add them together and you get $O(M+N)$, which is still a little slower than your typical hash table, which will only have to compute a single hash value.

How about we try deleting? We perform the same steps as finding the element, and then we remove it. Removing will take $O(M + N)$ in the worst case. In a typical hash table, deleting is only $O(N)$. So again, this structure would take longer.

Inserting, finding, and deleting elements in this new structure would be a little slower than in a normal hash table. Does it help us save on memory? Well, remember that each bucket contains a mapping to another bucket. These buckets take up space in memory. So, if we have M hash functions, we will actually be taking up M minus 1 extra spaces in memory when compared to a typical hash table.

Notice that above we use M to describe the work needed to compute the extra hash functions for our new structure. Remember, though, that computing hash functions is basically free in most applications. So, this means that the extra work required to find, insert, and delete in our new structure is basically constant if we use an efficient hash function. Does this mean we can ignore this extra work then? No. We still need to take into account these extra computations, as they become increasingly more important the more expensive it is to compute our hash functions. This is one of those things that becomes an implementation detail, but when designing our systems we have to consider the worst case.

So is this totally useless? Well, not necessarily. We have a structure here that is sort of like a bloom filter, but the difference is that we can access values directly rather than using a bit array. So, the advantage here is that we will never get any false positives. This structure is sort of analogous to a bloom filter that sacrifices memory and runtime efficiency for accuracy of lookups. Also, the structure of having extra hash functions sort of gives us an improvement on traditional hash tables, since there is a lower chance that any two elements will have all the same hash values from every hash function resulting in less collisions. So, this would be a decent choice if you wanted to employ a hash table that limits the number of collisions and you do not necessarily care about the runtime or memory usage. But, then again, collisions can just be reduced by picking a good hash function, and, as is the whole message of this blog, we like reducing memory usage and runtime complexity.

Throughout this blog, we have learned the fundamentals of Swiss Tables, Bloom filters, and Suffix arrays, examining each from the perspective of memory usage and efficiency. Hopefully, we have noticed that there is no single universal data structure to rule them all. Every structure has trade offs, and something one structure is good at another may be equally bad at. It is our job as software engineers to consider how the trade-offs affect our application. The topics introduced in this blog were just a few of very many memory-friendly data structures and algorithms that have clever use-cases. We just have to be clever enough to pick the right ones for our job.

The ways we store and retrieve information can have major implications on everyday life. A small delay in the delivery of information can be the difference between life and death, much like decreasing memory utilization can save millions of dollars worldwide. But these optimizations come with certain algorithms and structures, and with these structures comes certain drawbacks. We must ensure that we employ these algorithms and data structures correctly such that the drawbacks do not stray us further from our original goal; increasing welfare through optimizing the interface between people and the solutions computing can provide.