UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

BSc Thesis

Kasper Unn Weihe (PXH755)

Optimization of Hash Tables

Advisor: Mikkel Thorup

August 15, 2023

Abstract

A hash table is one of the most common data structures in computer software. In this thesis, a fast hash table has been implemented to gain an understanding of what optimizations can make a hash table more performant and how hash tables are currently implemented in standard- or third-party libraries in some of the most popular programming languages. The implementation uses a new variation of collision resolution based on Robin Hood hashing along with an extensive set of low-level optimizations. It has been benchmarked against several of the most popular hash table implementations in C++. While it is currently limited in functionality compared to the other implementations, it has shown good performance results. It is especially fast for unsuccessful search, insertion and has quite good memory usage.

Keywords: hash table, optimization, fast

Contents

Chapter 1

Introduction

A hash table is one of the most commonly used data structures in computer software, used for associative arrays, database indexing, sets, and much more. Some would even argue that it is the most used data structure aside from arrays. Google engineer Matt Kulukundis reported[12] that 1% of the CPU power in Google's server fleet is spent on hash tables, and hash tables own more than 4% of Google's RAM - and that is only for C++. Most programming languages have this data structure built into their standard libraries - C# and Python calls it `Dictionary`, JavaScript and Go: `map`, C++: `unordered_map`, and Java: `Hashtable`. A hash table most commonly stores objects in the form of key-value pairs. With a proper hash-function, each object can be retrieved in $O(1)$ expected time[14], whereas data structures such as a balanced binary search tree would have $O(\log(n))$ time complexity. The time complexity of the dictionary operations `INSERT`, `SEARCH`, and `DELETE` for hash tables is what often makes them an attractive choice for many problems.

1.1 Thesis objective

The goal of this project is to investigate and devise several optimization techniques for hash tables. This should be used to develop a fast hash table in a low-level language such as C, C++, or Rust. The hash table should be benchmarked up against established implementations to see how they compare. This project aims to build a hash table that is fast but not necessarily memory efficient. Speed should be the primary focus of the developed hash table. The goal is not to beat every hash table in existence because this is a field with years of research from major companies and universities. Implementing a fast hash table should help understand what makes a hash table fast and how they are implemented in some of the most popular libraries. This project should also examine how we can get a good picture of the performance characteristics of a hash table and how we can compare them fairly. This project does not have a focus on the implementation and math behind hash functions. The focus is on the organization of hash tables.

1.2 Thesis structure

Chapter 2 will present the foundation of hash tables and will introduce different variants/categories of hash tables and how they function. Chapter 3 will cover the design of a new type of hash table based on Robin Hood hashing. Furthermore, chapter 3 will explain the hash table implementation and some of the low-level optimizations. Chapter 4 will cover benchmarking. The performance of the table will be compared to the fastest hash tables created for C++, including implementations from Google and Facebook. Chapter 5 presents the future work of the implementation and provides a conclusion.

1.3 Source code repository

The source code for the implementation can be found at github.com/ka-weihe/hashmap. The README found in the repository explains how to build and run the benchmarks.

Chapter 2

Hash Table

A hash table is an effective data structure for implementing associative arrays (dictionaries). In an associative array, data is stored as collections in the form of key-value pairs. The index of the key-value pairs within the array is determined by using a hash function on the key. When we want to retrieve an element from a hash table, it will, with most implementations, have a worst-case complexity of $\Theta(n)$, just as if we were to search for a value with a matching key in a linked list or an unsorted array. Fortunately, under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$. When the number of keys stored in the hash table is relatively small compared to the total number of possible keys, the hash table becomes an excellent alternative to an array where the key maps directly to an index. For example, if the keys stored are a single byte each, you might as well keep the key-value pairs in an array of length 256, one index for each possible key. This is known as a direct-address table, and the worst-case lookup is $\Theta(1)$. Nevertheless, if the key type is 64-bit integers, you cannot use this trick because you would need an array of length 2^{64}, and that would be infeasible to store in computer memory. Instead of using the key to index directly, the index can be computed from the key with a hash function.

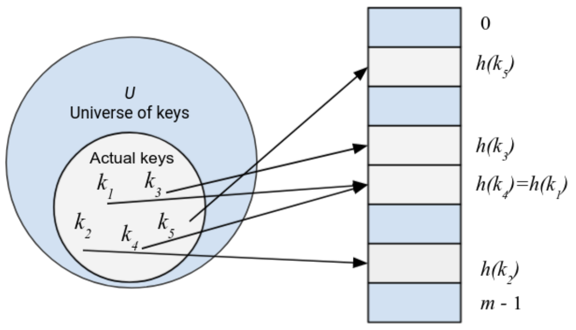

Figure 2.1: Using a hash function h to map keys from the universe U to hash-table slots. k_4 and k_1 hash to the same slot, so they collide.

2.1 Hash functions

A hash function is used to map data from an arbitrarily large universe U of keys to fixed-size values in range $[m] = \{0, \ldots, m-1\}$. For example, a hash function can turn strings of arbitrary length into 64-bit hash values. A truly independent hash function assigns an independent uniformly random variable $h(x)$ for every key in x. Unfortunately, this is not possible to implement as it would require too much memory[13]; instead, we can store only a small seed of randomness such that the hash function is sufficiently random for the desired application.

2.2 Universal hashing

The hashing algorithm will have to handle some data set $S \subseteq U$ where $|S| = n$ keys that are not known in advance. A deterministic hash function does not offer any guarantees, because an adversary may choose S such that all hash values collide, which makes the hashing useless. The solution is to generate a random hash function from a family of universal hash functions $H = \{h : U \to [m]\}$ where H is universal if the following holds:

$$\forall x,\ y \in U,\ x \neq y,\ \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$$

In other words, when h has been drawn randomly, the probability of two keys in U colliding is at most $\frac{1}{m}$. Unfortunately, hash functions such as `std::hash` which is an implementation of `MurmurHashUnaligned2` in `libstdc++-v3`[7] does not use universal hashing and thus have no guarantees for the probability of collisions. Variants of `MurmurHash` are used in many of the most popular hash table implementations such as `std::unordered_map` and Facebook's `folly::F14ValueMap`. Most of the hashing algorithms used by default in existing hash table implementations seem to value performance over mathematical guarantees. However, this comes at the price of being vulnerable to attacks such as Hash-flooding DoS [4].

2.3 From hash to index

Most string hash function implementations map to 32- or 64-bit hash values, but a hash table may only have m buckets where m is less than 2^{32} or 2^{64}, respectively. This begs the question of how we go from the hash value to an index that does not exceed $m - 1$. The naive solution would be modulo reduction: `hash % ` m. Unfortunately, modulo is an extremely slow operation in practice. The Intel manual lists it as taking anywhere between 80 and 95 cycles[11]. Luckily, if m is a power of two, we can do `hash & (`m` - 1)`. Bit-wise `AND` costs less than 1 cycle on modern hardware, so this ought to be a big optimization. Instead of taking the least significant bits, we can alternatively do `hash » (64 - `$\log_2$`(m))`. Some hash table implementations set m to numbers that are not powers of two, such as prime numbers, but this makes the optimizations explained above infeasible to use. One way of optimizing modulo reduction is by creating a function for every value that m can be. Modern compilers will optimize `hash % ` m where m is a compile-time constant, such that the resulting assembly does not use division, which is largely what makes it slow. While it has more instructions (as seen in listing 2.1), it is much faster. Pointers to these functions can be kept in an array in order of m. When the table resizes, we can thus pick the next element/function pointer from the array. One potential downside to this approach is that function pointers may prevent optimizations such as inlining and branch prediction. If we were optimizing for low memory usage, then we would want m to increase in small increments as opposed to in powers of two. In this case, compile-time modulo functions may be an excellent solution that would not hurt performance too much.

```
int mod(int a, int b) {              int mod11(int a) {
    return a % b;                        return a % 11;
}                                    }

mod:                                 mod11:
    mov     eax, edi                     movsxd  rax, edi
    cdq                                  imul    rcx, rax, 780903145
    idiv    esi                          mov     rdx, rcx
    mov     eax, edx                     shr     rdx, 63
    ret                                  sar     rcx, 33
                                         add     ecx, edx
                                         lea     edx, [rcx + 4*rcx]
                                         lea     ecx, [rcx + 2*rdx]
                                         sub     eax, ecx
                                         ret
```

Listing 2.1: Regular modulo `mod` (on the left) versus an example of compile time optimized modulo using a constant value `mod11` (on the right). Compiled from C to x86 assembly.

Here is a small benchmark of a simple hash table using each of approaches explained above. From the benchmarks it becomes clear that modulo is really slow, and the compile-time modulo approach also lags a bit behind. The approaches using `&` and » seem to have similar and superior performance in comparison.

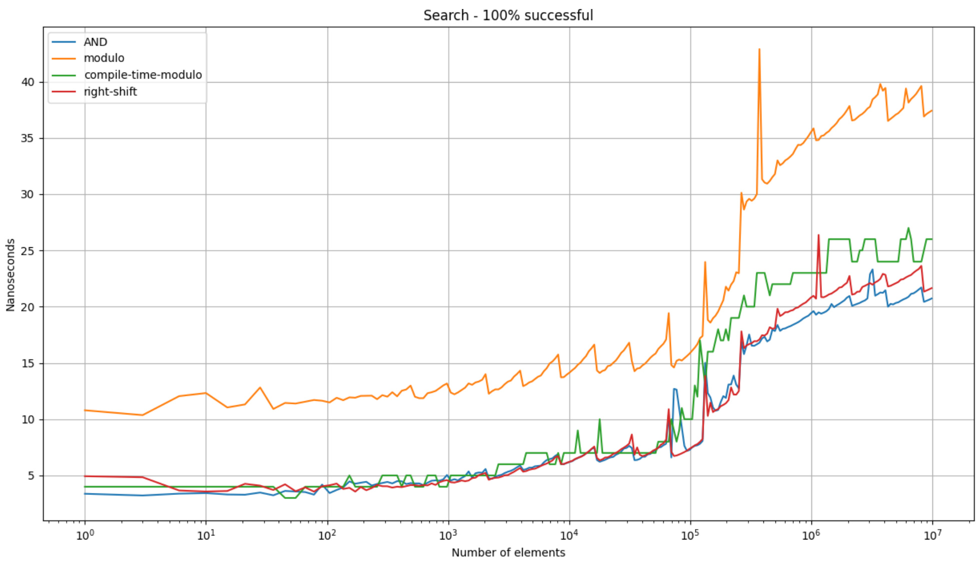

Figure 2.2: Average hash table search time with different ranging functions.

2.4 Separate chaining tables

Hash collisions are practically unavoidable when hashing from a large universe of possible keys to small fixed-size values. If we are hashing 64-bit integers to 32-bit hash values, collisions can naturally occur because 2^{64} of unique keys can not all map to a unique hash value. There must be a minimum of $2^{64} - 2^{32}$ collisions. Also, most hash tables typically start with a relatively small size of buckets, say 8. If we hash the key to get an index between 0 and 7, we can do `index = hash % 8`. With this in mind, it becomes even clearer that collisions are unavoidable, so what is the resolution to collisions? One simple but common way of handling collisions is to chain entries with the same index in a linked list or any other data structure that supports the required operations, such as an array. A self-balancing binary search tree can also be used to bring the worst-case time for the dictionary operations to $O(\log n)$ instead of $O(n)$. `std::unordered_map` in C++ is implemented using linked lists. Some chaining implementations may also store the first entry of each chain in the slot array itself to improve cache efficiency - this is known as "separate chaining with list head cells". If the distribution of keys is sufficiently uniform, the average cost to find an element in a chaining hash-table depends only on the average number of keys per bucket.

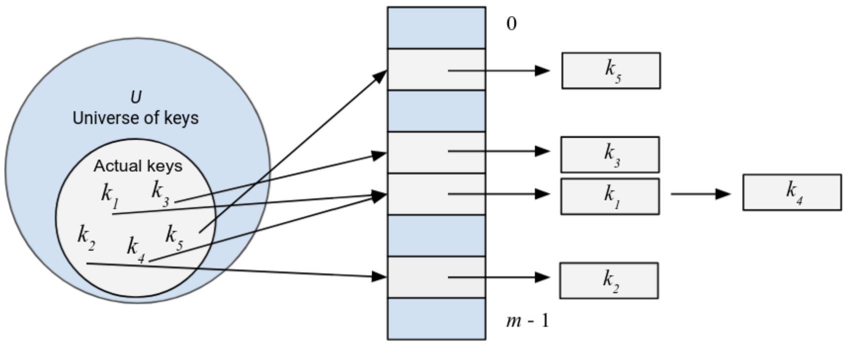

Figure 2.3: Collision resolution by chaining. Each slot in the table holds a linked list. k_4 and k_1 hash to the same slot, so they collide and form a linked list. The list can be either singly- or doubly-linked.

The dictionary operations for a hash table `T` can be implemented as such:

```
INSERT(T, x)
    insert x at the head of list T[h(x.key)] if not present
SEARCH(T, k)
    search for an element with key k in list T[h(x.key)]
DELETE(T, x)
    delete x from the list T[h(x.key)]
```

2.5 Open addressing tables and linear probing

With open addressing, all elements are stored in the hash table (or in the array representing the hash table) itself. Unlike chaining, no list and no elements have to be stored outside the table. Collisions are resolved with probing, meaning: when a new element is inserted, we check if the cell is empty; if not, we proceed to the next cell and probe the element until an empty index is found. Therefore, it will eventually fill up as we insert new elements, but as the array gets closer and closer to full, the number of extra steps, or probes, required to find any piece of data can grow a lot. The benefit of this approach is that we can eliminate pointers altogether. This is much more cache friendly (i.e. in terms of locality of reference) because all elements are next to each other in memory. With chaining, we have to follow pointers to different places on the heap, which is notoriously bad for performance, hence why most fast hash tables, if not all, use some form of open addressing. We can also save memory by not having to store the large number of pointers that chaining requires.

It is called linear probing when we increase the index by one when a cell is occupied. Other variants such as quadratic probing and double hashing exist, but these are generally not very performant. With linear probing, entries can very easily be probed far away from where they originally hash to. This naturally has a bad reflection on performance because many comparisons have to be made to determine that an entry is present in the hash table. Linear probing is also more sensitive to the quality of its hash function than some other collision resolution schemes. However, it takes constant expected time per search, insertion, and deletion when implemented using a 5-independent hash function[14].

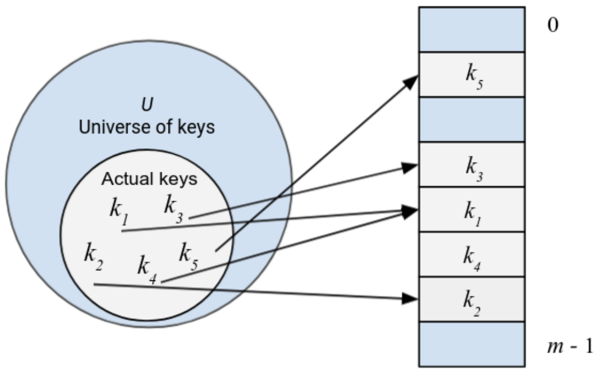

Figure 2.4: Collision resolution by linear probing. k_4 and k_1 hash to the same slot, so the last inserted key k_4 moves to the next free slot.

The dictionary operations for a hash table `T` can be implemented as such:

```
INSERT(T, x)
    Insert x at the first empty bucket at T[h(x.key)] or after
SEARCH(T, k)
    Search for an element with key k at T[h(x.key)] or after;
    Stop on empty bucket
DELETE(T, k)
    Search for x with key k and delete it
    Move succeeding entries back if they have been probed past x
```

Robin Hood hashing

Robin Hood hashing was first introduced in [5] and hoped to alleviate the problem of elements being far away from where they originally hash to. The idea is to steal from the rich and give back to the poor. In this case, rich meaning high probe count, and poor meaning low probe count. Probe count is the number of times an entry has been probed from its original index. An entry may displace an entry already inserted if its probe count is larger than that of the key of the current position. This ensures that all entries are placed as close as possible to where they originally hash/index to. Observe figure 2.5. We have the following keys inserted in order: a, b, c, d, e, f, g, h. We have that every key is hashed to an empty cell except h. h is hashed to index 0 like a and is thus probed all the way to the end. With Robin Hood hashing, h takes the place of b because it has a higher probe count, which causes a ripple effect of all the remaining elements being probed once.

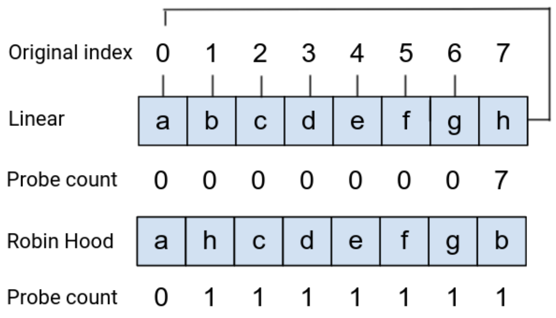

Figure 2.5: Linear probing compared to Robin Hood hashing.

While this does not reduce the total sum of probe counts, it does reduce the variance of the probe counts. The original paper shows that the expected variance of an infinite Robin Hood table is generally much smaller than a standard linear probing table as seen in table 2.1.

α	Robin Hood Hashing	Standard Method
0.7	0.527	2.906
0.8	0.700	6.428
0.9	0.983	16.207

Table 2.1: Expected variance at different load factors from [5]. a is the load factor - the load factor is a measure of how full a hash table is. A load factor of 1 means that the entire capacity of the hash table is filled. A load factor of 0 means that the hash table is empty.

Why is a low variance good? A low variance is good because the cost of looking up elements is largely fetching the cache line. Ideally, we want all probe sequences to fit in single cache lines. However, cache lines are typically 64 bytes, so large probe sequences would have to fetch multiple cache lines. Since standard linear probing has more significant variance, it will therefore have more cache misses.

We know that the average number of probes required for successful search in a Robin Hood table and a standard linear probing table is the same[5]. However, there is a big difference in terms of unsuccessful searches. With standard linear probing, we search until we reach an empty cell. However, with Robin Hood Hashing, we stop searching when we reach an element with a larger probe count or reach an empty cell. This is under the assumption that we save the probe counts of all elements in the table. One way of implementing Robin Hood hashing is by storing the probe count of every entry in an info-byte where the most significant bit is used for marking the bucket as taken or empty. This is how the bit layout is:

7	6	5	4	3	2	1	0
full?	probe count						
0/1	0-127						

When storing four elements a, b, c, d where a, b map to hash position 1 and c, d map to hash position 2, this is how the entries would be stored:

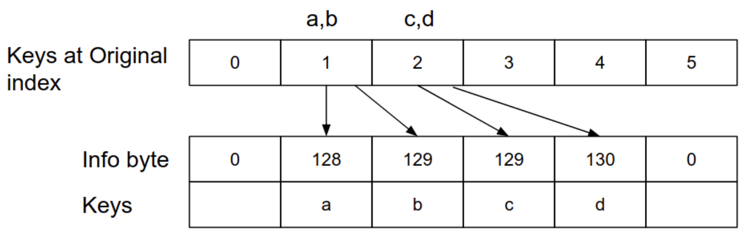

Figure 2.6: Robin Hood hashing with info-byte.

Using 7 bits for the probe count is quite a lot. The probe count should ideally never even come close to 127. Instead, we can lower the maximum probe count to a much lower number and store some of the bits from the hash. There are multiple benefits to this: we save many key comparisons because we only need to compare the keys if the hashes match. Comparing the keys can be expensive, especially if they are strings. The hash bits can also be used when resizing the hash table, such that we don't have to rehash every key to find new indices for all entries. This is how the info-byte with hash-bits could look:

7	6	5	4	3	2	1	0
full?	probe count				hash bits		
0/1	0-15				0-3		

With the use of Robin Hood hashing and info-bytes, we have that lookup would be:

```
search(key) {
    idx = hash(key) & (size_of_hashtable - 1)
    info = 128
    while (info < info_bytes[idx]) {
        idx++
        info++
    }
    while (info == info_bytes[idx]) {
        if (key == keys[idx]) {
            return vals[idx]
        }
        idx++
        info++
    }
    // nothing found
    return 0;
}
```

Listing 2.2: Robin Hood search.

Swiss Tables

Swiss tables are another type of open addressing tables first presented by Google at CppCon in 2017[3]. Many of the fastest hash-tables, such as Facebooks F14 tables[6] and the excellent "parallel hashmap" by Gregory Popovitch[1], are variants of Swiss tables. Swiss tables hold a densely packed array of metadata containing a byte for every entry in the table. The hash-function produces a 64-bit value that is split into two:

- H1 - 57 bits of the hash used to identify the index of an entry.
- H2 - 7 bits of the hash value that is stored as a part of the metadata in the table.

The metadata bytes stored in the hash-table have the most significant bit as a control bit. The remaining 7-bits are the H2. The control-bit in combination with the H2 determines whether an entry is full, empty, or deleted. It is full if the control bit is 0, it has been deleted if all of the bits are 1, and it is empty if the control bit is 1, but the remaining bits are 0. The metadata is laid out consecutively for every 16 entries:

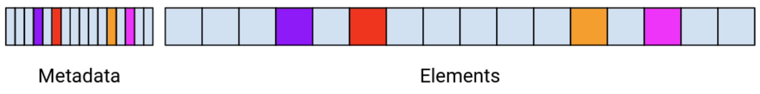

Figure 2.7: 16 elements with their respective metadata bytes. Elements with the same H1 hash have the same color, and blue elements are empty.

This layout enables the use of SSE instructions[10] to lookup elements with instruction-level parallelism. The H1 hash is used to find the bucket chain of 16 elements, and the H2 hash gets broadcasted to a 128-bit mask. Every byte of the mask (which are now all equal to H2) gets compared for equality to the metadata bytes. If they match, we have a candidate, and we check for key equality. If no candidates are found, we probe.

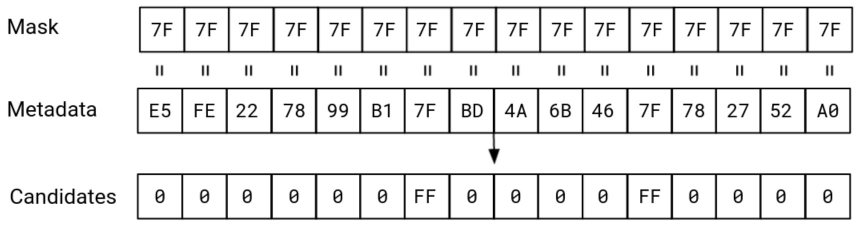
Mask
7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F 7F
Metadata
E5 FE 22 78 99 B1 7F BD 4A 6B 46 7F 78 27 52 A0
Candidates
0 0 0 0 0 0 FF 0 0 0 0 FF 0 0 0 0

Figure 2.8: Testing the mask for equality with the metadata using SSE-instructions. The H2 hash of the key we're searching for is 7F.

The SSE enabled C++ code[3] to find candidates is shown below:

```
Mask Match(h2_t hash) const {
  auto match = _mm_set1_epi8(hash);
  return Mask(_mm_movemask_epi8(_mm_cmpeq_epi8(match, metadata)));
}
```

Note that there is no loop. This code compares the H2 hash to 16 elements in parallel to find the candidates. Therefore, long probe chains don't necessarily take much longer than short ones. It can search through very deep probe chains relatively inexpensively. While the SSE instructions above are x86 specific, most other architectures have similar intrinsics.

Chapter 3

Design and Implementation

For this project, I had to think of new ways to beat or improve upon current designs in terms of performance. Making the fastest hash table for every situation and every type of key would be next to impossible. I had to zero in on a specific area of hash tables. My focus is on string keys because strings are the most common key type (see figure 3.1) and are more expensive relative to other primitives - hashing and comparing strings is much more costly than integers, for example.

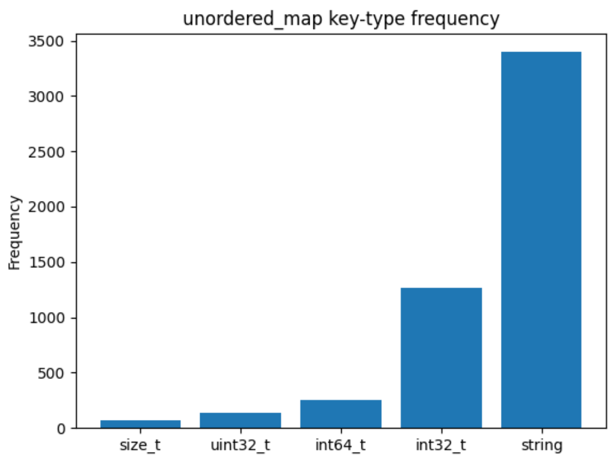

Figure 3.1: Frequency of primitive key type in 100GB of the most popular C++ repositories on GitHub.

Another focus for the implementation is how the dictionary operations are prioritized. The table is designed with the following order of priority (from highest to lowest): search, insert, and delete. This priority reflects how I assess that hash tables usually are being used in programs. In my experience, search has by far the most use, whereas delete has very little use. With a focus on only a subset of hash tables, we should have a much higher chance of creating something useful and perhaps innovative.

3.1 Design

What approach should this hash table take? Most of the fast hash tables for C++ are either Robin Hood hash tables or Swiss tables, according to Martin Ankerl's very detailed benchmarks[2]. Swiss tables are open addressing hash tables that use SSE instructions to perform search operations on multiple entries with instruction-parallelism[3]. These are very complex and also machine-specific because the instructions used are extensions to the x86 architecture. Robin Hood hashing seemed a better place to draw inspiration from because of its simplicity, and this will also allow for an implementation that runs on architectures such as ARM and RISC-V without having to specify the proper intrinsics for every architecture. The memory layout for the hash table of this project is visualized in figure 3.2:

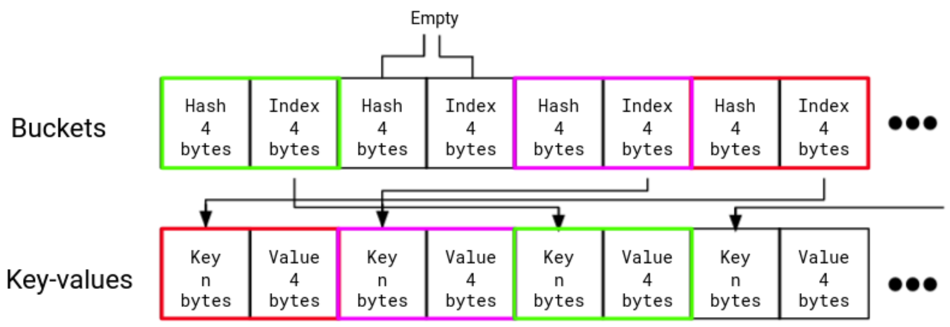

Figure 3.2: The memory layout of the hash table implementation. Each bucket consists of a hash and an index. Here 4 buckets of the hash table are shown with 4 key-value pairs inserted into the key-value array of the hash table.

The bucket array holds pairs of hashes and indices. Every hash has a respective index that comes right after in the array. The index points to the string key that is located in the key-value array. Strings are usually pointers to characters on the heap, meaning that every key may have its own allocation and place in memory. With this design, every key is stored continuously in memory. When the array resizes, we allocate more than we need, such that allocations are not needed every time a new entry is inserted. This should ensure faster insertion and offer excellent memory savings if the growth factor is set to a low number. This was inspired by CPython's move to ordered dictionaries[9]. Notice that info bytes are not used; instead, we use the 4-byte hash. The idea of this table is to store the hashes in ascending order. The most significant bit is 1 if the bucket is empty:

31	30 - 0
empty?	hash
0/1	$0 \ldots 2^{31} - 1$

Keeping 31-bits from the hash should save us many key comparisons and make it such that we never have to rehash the key. This naturally requires a lot more memory than using a single byte for the probe count, but there are various ways we can save memory, which will be explored later. For example, 32 bits for indices is a little high because the tables will rarely have close to 2^{32} entries. Also, the performance of the table is the primary focus. So how exactly would this table work? Firstly, have a look at how the keys are hashed:

```
uint32_t mask = ~((uint32_t)1 << 31);
uint32_t hash = hashfn(key, len, seed) & mask;
```

The mask is a 32-bit unsigned integer with all bits 1, except for the most significant bit. Next, we hash the key using a hash function. Notice that the hash produced from the hash function is ANDed with the mask, meaning that the most significant bit of the hash becomes 0. Recall that 0 means non-empty.

How do we go from the hash to the index? If the table has space for m entries, no entry should go beyond index $m - 1$. As shown earlier, we can ensure this by doing `hash %` m. Unfortunately, modulo is extremely slow because it uses division. Instead, we make sure that the size of the table is always a power of two. This ensures that we can do: `hash` » $\log_2 m$. In other words, we take the $\log_2 m$ most significant bits to get an index between $0 \ldots m - 1$. Why do we want to use the most significant bits instead of the least significant bits? The reason for this is that it ensures that entries with higher hashes have higher indices. For example, keys that hash to index 0 will always have smaller hashes than keys that hash to index 1. Since the hash is 32 bits, we right-shift it by `h->shift` which is equal to $\log_2 m$. Also, since the table has the hashes and indices interleaved, we multiply by 2. The index below should not be confused with the index in figure 3.2. It points to a bucket, but in the figure, it points to a key-value pair.

```
uint32_t index = (hash >> (h->shift)) * 2;
```

If the hashes are in ascending order, we get the same effect as Robin Hood hashing but amplified. With Robin Hood, we never have to search past entries with larger probe counts, but with ascending hashes, we never have to search past larger hashes. Say we have keys, a, b, and c with hashes $h(a) > h(b) > h(c)$ that all have the same original index. We Insert a and then b. With Robin Hood hashing, a would come before b in the table. However, with the proposed method of ascending hashes, b would come before a because its hash is smaller. When searching for c, which has not been inserted, Robin Hood would have to probe past both a and b, but the proposed method would not have to probe past b because $h(c) < h(b)$. We compare until a larger hash is reached. If we assume that the hashes are uniform, we can compute the expected percentage of saved probes. We have a table with m buckets; let two keys a and b have the same original index. They will have $\log_2 m$ bits that match, and since 1 bit is used for determining whether an entry is empty or not, we have that the keys will have $32 - 1 - \log_2(m)$ remaining random bits that have not been used to find the index. Let r and q denote the numbers that the remaining bits represent for a and b, respectively. The expected percentage of saved probes is the probability that q is less than r. We have:

$$Pr[q < r] + Pr[q > r] + Pr[q = r] = 1$$

Because of symmetry

$$Pr[q < r] = Pr[q > r]$$

Therefore

$$Pr[q < r] = \frac{1 - Pr[r = q]}{2}$$

$$Pr[q < r] = \frac{1 - \frac{1}{2^{32-1-\log_2(m)}}}{2}$$

If we plot this for every m (power of two) in $0 \ldots 2^{31}$ we get:

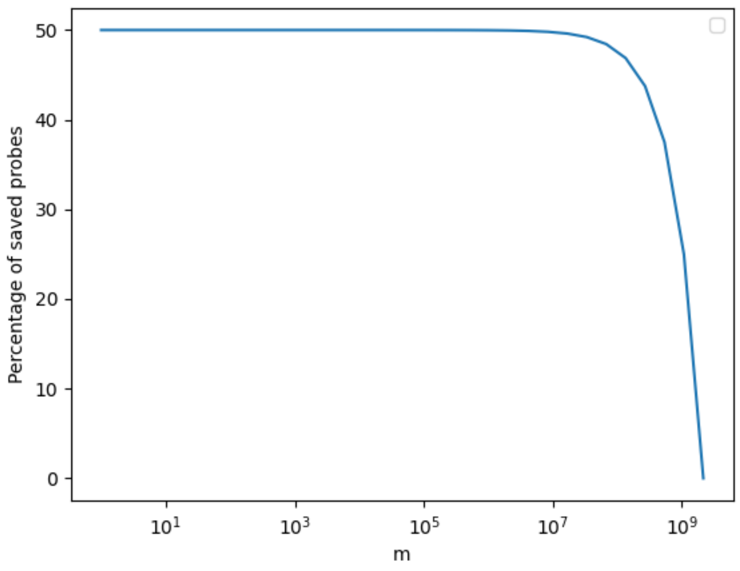

Figure 3.3: Percentage of saved probes.

As you can see from figure 3.3 we can expect to save nearly 50% of probes when the original indices are equal. What if we could fast-forward past buckets with a lower original index. We can achieve this by keeping fast-forward data at every bucket b. This number has information about where the entries with an original index at b start. This is inspired by how hopscotch hashing works[8]. If using fast-forward, we would thus save nearly 50% of the expected probes for unsuccessful search if we don't count the fast-forward as a probe. The memory layout for this is similar to the structure shown in figure 3.2 and both designs have been implemented.

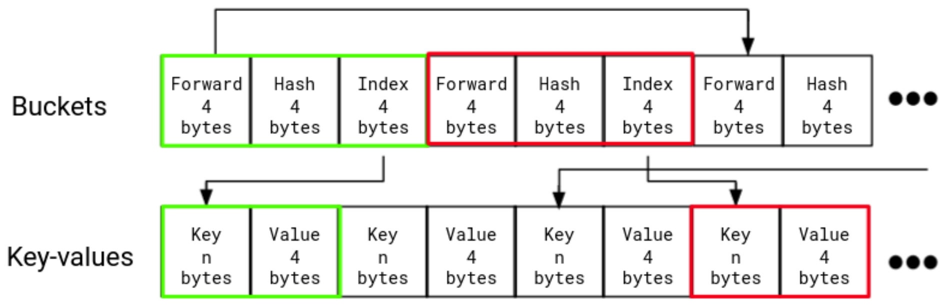

Figure 3.4: Memory layout with fast-forward.

3.2 Implementation

In this section, certain parts of the implementation are explained. The implementation shown is without fast-forwarding, but it has a lot in common with the fast-forward implementation. The hash and index can be set to 4 or 8 bytes.

Insert

Now that we know how the `index` and `hash` is found given a `key` and a `value`, we can look at how entries are inserted. Recall that we want the hashes to be in ascending order, so when we insert a new entry, we start by checking if the hash at the `index` in the bucket array is smaller. If it is, we increment the `index` by two until the hash at the index is no longer smaller than the `hash` of the key we want to insert for. Notice that we only need to increment one number as opposed to listing 2.2 where we had to increment the probe count as well.

```
while (hash > h->buckets[index]) {
    index = (index + 2) & (m - 1);
}
```

Now that we know that the hash at the `index` cannot be smaller than `hash`, there's a possibility that it may be the same, so while the hashes are equal, we compare the keys, and if they match, we swap the value in the table with `value`. This is done until the keys match or the hashes no longer match:

```
while (hash == h->buckets[index]) {
    uint32_t kv_index = h->buckets[index + 1];
    uint32_t key_len = *(uint32_t*)(h->key_values + kv_index);
    if (len == key_len) {
      char* potential_match = (h->key_values + kv_index +
      ↪  sizeof(uint32_t));
      if (memcmp(potential_match, key, len) == 0) {
        // We have match, replace the value
        int* dest = (int*)(potential_match + len);
        *dest = value;
        return;
      }
    }
    index += 2;
}
```

If the keys don't match during the former step, we can be sure of two things. The hash at `index` and after now have higher values, and we know that the key we are inserting is not currently present in the table. Now the key-value pair is inserted into the ordered array of key values:

```
uint32_t key_value_size = sizeof(uint32_t) + len + sizeof(int);
uint32_t old_bytes = h->bytes;
h->bytes = old_bytes + key_value_size;
if (h->free_kv_space < (int32_t)key_value_size) {
    h->key_values = (char*)realloc(h->key_values, h->bytes * 2);
    h->free_kv_space = h->bytes;
} else {
    h->free_kv_space -= key_value_size;
}

char* dest = h->key_values + old_bytes;
uint32_t* dest_u32 = (uint32_t*)(h->key_values + old_bytes);
uint32_t kv_index = old_bytes;

*dest_u32 = len;
dest += sizeof(uint32_t);
memcpy(dest, key, len);
dest += len;
int* dest_int = (int*)dest;
*dest_int = value;
h->size++;
```

The key-value pair has been inserted at kv_index, but we still have to insert the hash along with the kv_index into the table so that we can find the key-value pair without linear search through the key-value array. If the bucket/entry is not empty, we swap the hash with the hash at index and increment index with two. We keep doing this until we reach an empty index, where we insert hash and index.

```
// If the bucket is equal to UINT32_MAX it is empty.
while (h->buckets[index] != UINT32_MAX) {
  uint32_t tmp_hash = h->buckets[index];
  uint32_t tmp_kv_index = h->buckets[index + 1];
  h->buckets[index] = hash;
  h->buckets[index + 1] = (uint32_t)kv_index;
  hash = tmp_hash;
  kv_index = (uint32_t)tmp_kv_index;
  index += 2;
}
```

Here a question may arise. What happens if we probe past the last index of the table? For example, if the table has space for eight entries and index 7 is taken. If we insert a new entry that also indexes to 7, there is no space for probing. A common solution to this is to make the table cyclic. In other words, every time an entry is probed, we increment the index like this:

```
index = (index + 1) & (m - 1)
```

If we probe past the last index, we go to the first index. This would not work with this table because if we probe past the last entry, then we know that the hash we are inserting must be bigger than any hash in the table, and it would be probed to the beginning of the table - violating that the hashes are in ascending order. Instead, we can do a simple but clever trick. We always keep one or more empty entries after the last index. If the table has space for eight entries, we add 1. No entry can directly index to 8, but if we probe to the 8'th index, we increase the size of the table such that an entry can never probe beyond the last index. There will always be an empty entry at the end of the table, and the hashes will always be in ascending order. This is also what ensures that we do index++ instead of index = (index + 1) & (m - 1) when probing:

```
uint32_t max_index = (h->capacity + h->extra_buckets) * 2 - 2;
if (index == max_index) {
    h->extra_buckets++;
    h->buckets =
    (uint32_t*)realloc(h->buckets, (h->capacity +
    ↪  h->extra_buckets) * 2 * sizeof(uint32_t));
    memset(h->buckets + max_index + 2, 255, 2 *
    ↪  sizeof(uint32_t));
}
```

Choosing a low max load factor will result in smaller probe chains, whereas a high max load factor will save lots of memory. Finding the optimal load factor is a balancing act because if the max load factor is too low, there will be a lot of wasted memory, and you have to allocate more memory when inserting. The max load factor has been chosen to 0.8 by default such that we can benchmark all of the hash tables with the same max load factor later. In figure 3.5 you see the average search time at different max load factors. Notice how higher max load factors are spikier.

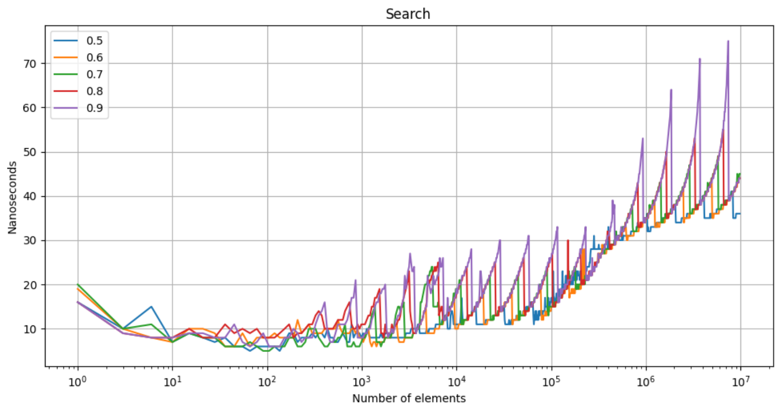

Figure 3.5: Search time per element with different max load factors.

If the load factor exceeds the max load factor, we have to resize the table and rehash every entry to find its new index. A low max load factor will cause more rehashings, which can be expensive, but there is no big difference with this table because all of the hashes are cached in the table. Caching the hash is one of the most significant optimizations we can do for insertion.

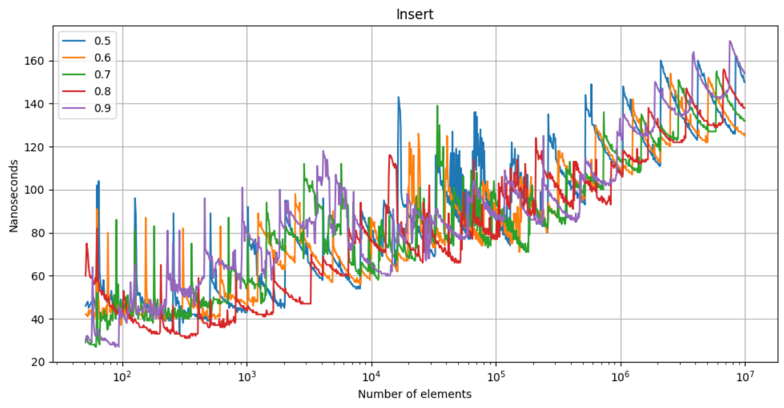

Figure 3.6: Insert time per element with different load factors.

Delete

`Delete` is perhaps the most difficult function to implement, especially for the memory layout of this table. Deletion has been implemented as a mix of back-substitution and tombstones. When an element is deleted, the succeeding elements in the `buckets` array get shifted back if they have been probed. When the elements have been back-shifted, we mark the respective key-value pair as deleted with a tombstone, by setting the length of the key to the max value. If more than half of the memory in the `key-value` array has been deleted, we start a clean-up procedure (as seen below) where the "holes" in memory get removed:

```
if (h->deleted_memory >= (h->bytes >> 1) && h->deleted_memory >
↪  100) {
    // clean up
    // move move key-value pairs back
    uint32_t free_mem = 0;
    char* at = h->key_values;
    char* end = h->key_values + h->bytes;
    while (at < end) {
      uint32_t key_len = *(uint32_t*)(at);
      uint32_t masked = (key_len & mask);
      uint32_t kv_size = masked + sizeof(uint32_t) +
      ↪  sizeof(int);
      if (key_len >= ((uint32_t)1 << 31)) {
        free_mem += kv_size;
      } else if (free_mem) {
        char* dest = at - free_mem;
        char* src = at;
        memcpy(dest, src, kv_size);
        free_mem -= kv_size;
      }
      at += kv_size;
    }
}
```

Search

The hash table has been designed around fast search. Luckily, search is also one of the most trivial functions. It can, however, be implemented in several different ways, and the difference in performance can be huge. After benchmarking multiple different versions, the fastest came out to be the one below. Notice that we have a do-while loop, and it has been unrolled. We check for equality twice in one iteration, and we only have one loop. This enabled the compiler to produce much faster code as opposed to the other versions. Benchmark results of the different versions can be seen in figure 6.2 and 6.1.

```c
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  uint32_t mask = ~((uint32_t)1 << 63);
  uint32_t hash = wyhash(key, len, 0, _wyp) & mask;
  uint32_t index = (hash >> (h->shift)) * 2;
  do {
    if (hash == h->buckets[index]) {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *potential_match = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(potential_match, key, len) == 0)) {
          // Replace value
          int *dest = (int *)(potential_match + len);
          return *dest;
        }
      }
    }
    index += 2;
    if (hash == h->buckets[index]) {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *potential_match = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(potential_match, key, len) == 0)) {
          // Replace value
          int *dest = (int *)(potential_match + len);
          return *dest;
        }
      }
    }
    index += 2;
  } while (hash >= h->buckets[index]);
  return 0; // No match
}
```

Listing 3.1: The implementation of the search operation.

Notice that we have `_likely_` around the comparison of key length and around the `memcmp` of the characters. `_likely_` is a branch prediction hint for the compiler. It tells the compiler that the branch is very likely to be true, which can cause a more negligible probability for the CPU to fetch instructions that will not be executed. This is how the macro is defined:

```
#if defined(__GNUC__) || defined(__INTEL_COMPILER) ||
↪  defined(__clang__)
  #define _likely_(x)  __builtin_expect(x,1)
  #define _unlikely_(x)  __builtin_expect(x,0)
#else
  #define _likely_(x) (x)
  #define _unlikely_(x) (x)
#endif
```

An attempt to beat memcmp

When comparing the keys, we use `memcmp`, but can we do better? We know that the keys are very likely to be equal when their hashes are equal, so we don't need to have branches to cut off the comparison before all bytes have been compared. Also, we `memcmp(s1, s2)` returns an integer less than, equal to, or greater than zero if the first n bytes of `s1` are less than, match, or greater than the first n bytes of `s2`, respectively. However, when comparing the keys, we are only interested in a match or no match. We can also make sure that the keys are multiples of 8 to boost performance. Furthermore, if we develop our own `memcmp`, we can make sure that the function is inlined such that there is no function call. Unfortunately, even with all of these assumptions, this task is far from easy. `memcmp` is exceptionally optimized and takes full advantage of your specific CPU. I did not let that stop me. I've implemented a `memcmp_eq` function that uses Advanced Vector Extensions (AVX) when the strings are 256 bits or above but uses longs when below. This function is able to beat `memcmp` consistently for strings below 40 in length. Past that point, `memcmp_eq` seems slower (see figure 6.3). The difference is not big enough for me to want to use it in the table, but I definitely think that it is possible to make something faster than `memcmp` given the formerly explained assumptions, but this can be a time consuming and tedious process.

```
inline int memcmp_eq(const void *str1, const void *str2, size_t
↪  count) {
  const __m256i_u *s1 = (__m256i_u*)str1;
  const __m256i_u *s2 = (__m256i_u*)str2;
  __m256i_u neq = _mm256_set1_epi64x(0);
  const uint64_t *s3 = (uint64_t*)str1;
  const uint64_t *s4 = (uint64_t*)str2;
  uint64_t neq2 = 0;
  while (count >= 4) {
    count -= 4;
    __m256i_u item1 = _mm256_lddqu_si256(s1++);
    __m256i_u item2 = _mm256_lddqu_si256(s2++);
    __m256i_u result = _mm256_cmpeq_epi64(item1, item2);
    neq = _mm256_add_epi64 (neq, result);
  }
  while (count--) {
    neq += (*s3++ != *s4++);
  }
  return !neq2 && _mm256_testz_si256(neq, neq);
}
```

Listing 3.2: An attempt at optimizing the `memcmp` function.

User guide

The hash table can be used in C and C++. All you have to do is include the header file in your program. You can choose between two versions - with- or without fast-forward. However, fast-forward does not have a function for deletion yet.

```
#include "fastforward.h"
// or
#include "weihe.h"
```

You can initialize a new hash-table by calling the `new_hashmap` function:

```
struct Hashmap* h = new_hashmap();
```

Inserting elements is simple. You call `hset` with arguments: pointer to hash-table, key, key length, and value.

```
// c
char name[] = "Kasper";
hset(h, name, strlen(name), 21);

// c++
string name = "Kasper";
hset(h, name.data(), name.size(), 21);
```

Now we can retrieve that value using `hget` with arguments: pointer to hash-table, key, and key length.

```
// c
char name[] = "Kasper";
hget(h, name, strlen(name));

// c++
string name = "Kasper";
hget(h, name.data(), name.size());
```

A specific key can be deleted by calling `hdelete` with arguments: pointer to hash-table, key, and key length.

```
// c
char name[] = "Kasper";
hdelete(h, name, strlen(name));

// c++
string name = "Kasper";
hdelete(h, name.data(), name.size());
```

Chapter 4

Benchmarking

These benchmarks evaluate the performance of 7 different hash table implementations, each with `std::hash` set as their hash function. The benchmarks compare `search`, `insert`, and `delete` of each table. The benchmarks have been performed on Arch Linux with a minimal number of background programs - no desktop environment and such. GCC version 11.0.0 has been used, and the CPU is an i5-10400f 2.9 GHz (4.3 GHz Turbo) with 12GB DDR4-2400 RAM. The benchmarks have been built with `-O3 -march=native`. These are the hash tables that have been benchmarked:

- Google's absl::flat_hash_map from Abseil version 20210324.1-1.
- Facebook's folly::F14ValueMap from Folly version 2021.05.17.00-1.
- std::unordered_map. The standard implementation of unordered_map included in g++. This is the libstdc++ implementation.
- greg7mdp's phmap::flat_hash_map version 1.33.
- Martinus's robin_hood::unordered_map version 3.11.2.
- Malte Skarupke ska::flat_hash_map version 1.0.
- Kasper Weihe's hash table (weihe): this is the hash table I've built for this project. This is the version without fast-forward. Benchmarks for fast-forward can be seen in figure 6.10 and 6.11.

4.1 Search

As previously mentioned, `search` has been considered as the most essential operation and has been the main focus of the implemented hash table. The goal of this benchmark was to make it as unbiased as possible. Therefore, the benchmarks try to do the following:

- `search` is performed with three different levels of probability of a key being found in the table. Thus, we have three levels: 0%, 50% and 100%.
- `search` is performed with varying levels of elements in the table. This is extremely important since the fullness of a table greatly influences the time it takes to find an element.
- Since this project does not focus on hash functions, the same hash function has been used for every hash table in every benchmark such that the hash tables get compared on a level playing field.
- The string-keys are very short (8 bytes) such that the hash function has less influence on the performance. The hash function of all tables has been set to `std::hash`, but other hash-functions have also been benchmarked, and the results are very similar. Benchmarks with 30 byte keys can be seen in Appendix 6.3.

This is a short outline of how the benchmarks have been performed:

1. Insert x random key-value pairs into the table.
2. Perform 10+ million `search` operations in random order and take the average time in nanoseconds.
3. Increase x and go to step 1 until the table contains 10 million elements.

Results

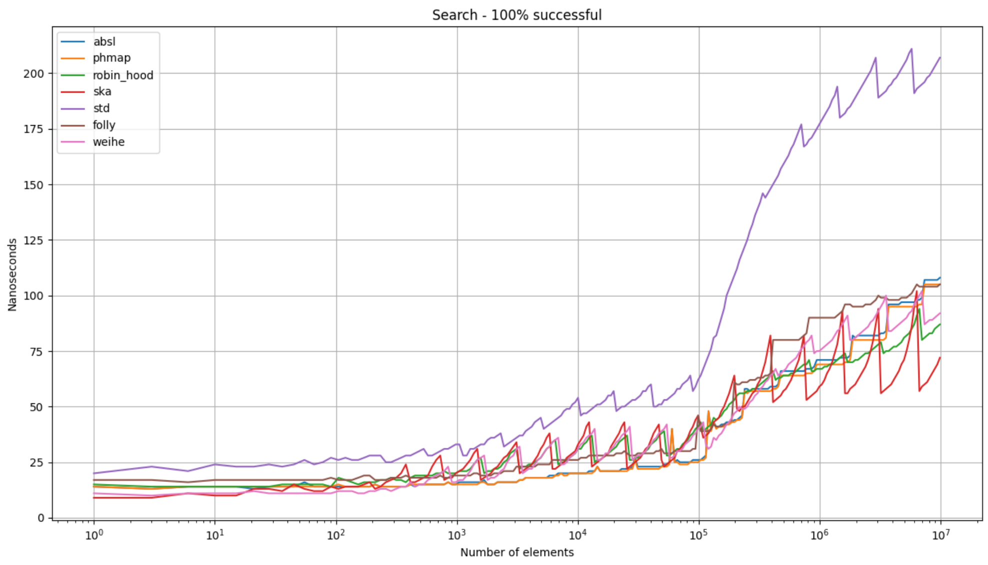

Figure 4.1: Average search time with 100% successful lookup.

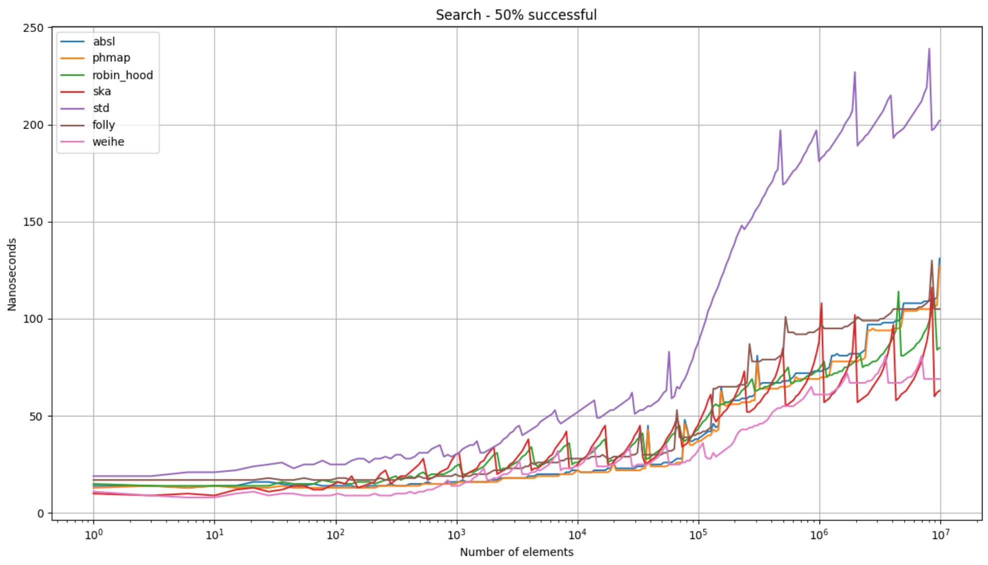

Figure 4.2: Average search time with 50% successful lookup.

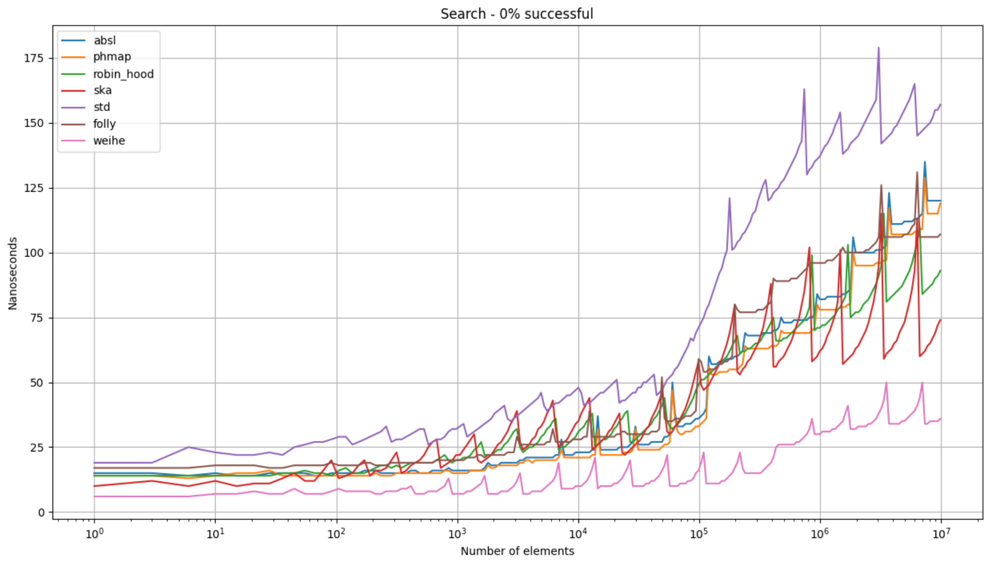

Figure 4.3: Average search time with 0% successful lookup.

Reflection on the results

There is a lot of interesting stuff to deduce from these results. First of all, we see that all the tables become quite a lot slower when they reach 10^5 elements. This is because the tables no longer fit in the L3 cache, and it now has to fetch memory from RAM. We also see a difference from 10^3 onward, which is likely because the tables no longer fit in the L2 cache.

We also see that `std::unordered_map` performs extremely bad. This can be expected from a separate chaining hash table because they have terrible cache characteristics. It is the only separate chaining hash table of the bunch. We can also see that most of the hash tables get slower as they become more full. This is expected because more collisions will happen as the table fills up, so more comparisons are needed to find the key we are looking for. Despite that, we see that this effect is less prominent with `phpmap`, `absl` and `folly`. These tables are all variants of Swiss hash tables, and they employ SSE-instructions to lookup 16 entries simultaneously with instruction parallelism. With these hash tables, it takes roughly the same time to look up an element that has not been probed and one that has been probed 15 times. These tables can search very deep probe chains inexpensively, but these instructions may have a higher constant cost, which is why they don't necessarily perform better than the tables with Robin Hood hashing. Only when the table has between 10^2 and 10^5 elements, the Swiss tables perform better - and not by much.

`ska`, `robin_hood`, and `weihe` all use Robin Hood hashing of some sort. They have nearly the same performance characteristics for 100% successful search, but this changes as we get more unsuccessful lookups. `weihe` gets quite a bit faster. This is expected due to a couple of reasons. Firstly, `weihe` never compares two keys unless their 32-bit hashes match. Some of the other tables do something similar, but for example, `robin_hood` only stores 1-5 bits from the hash, which means that the hashes will have a much higher chance of being equal, which in turn will result in the expensive comparisons of strings. Note that this effect has been minimized from this benchmark using 8 character strings. Another difference, and perhaps a more important difference, is that the other tables have to compare with every key with the same original index as the key we are searching for. With my table, we only have to compare keys with the same original index and a less or equal hash to the hash we are searching for. This greatly eliminates many comparisons, hence why it beats the other hash tables by quite a lot for 50% and 0% successful lookups.

4.2 Insert

As mentioned earlier, `insert` has been prioritized as the second most important operation for the hash table. For this benchmark, the keys are 8 characters long again. This benchmark is quite simple and it follows these steps:

1. Initialize a table and insert x amount of elements and measure the average insertion time.
2. Do step 1 again where x is a larger number until we reach a table of 10 million elements.

Results

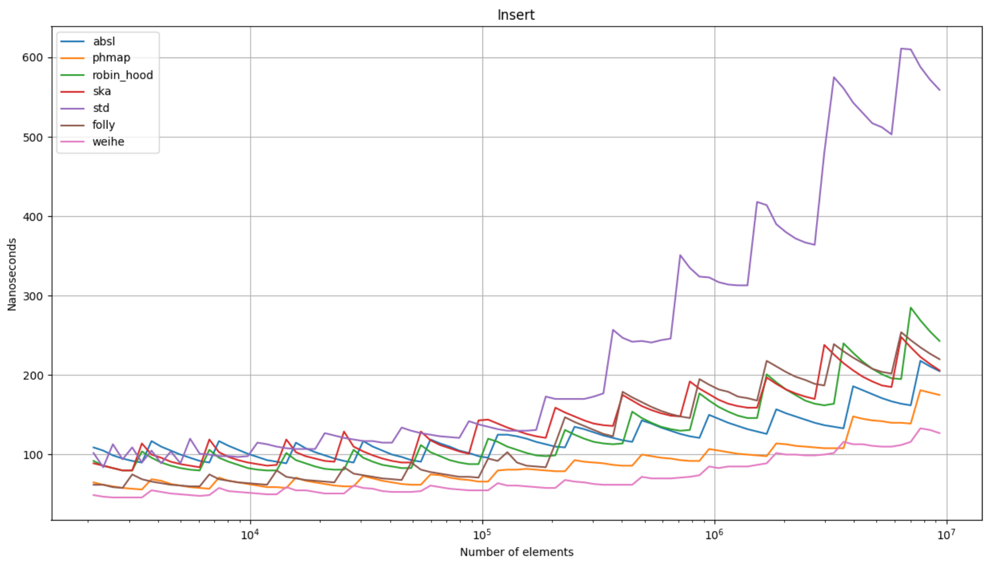

Figure 4.4: Average insertion time.

Reflection on the results

We also see this periodic hill pattern for this benchmark, but this time it rises quick but falls slowly, whereas it rose slowly and fell quick with search. The reason for the hills is the resizing of the table and making room for new elements. So why is my table so much faster in this regard? There might be multiple reasons for this. The primary reason is probably that my table never rehashes because all of the hashes are cached in the table. Rehashing all elements in the table is extremely expensive, so a lot of time is saved. Another big reason is that less memory is moved around - the other tables are unordered, meaning that if they move their metadata around, they also have to move the key-value pairs. Only the hash and index are moved around with my table, and all of the key-value pairs stay untouched. This is a massive improvement in comparison to the other tables. Now you probably think that my table uses a lot more memory than the others, which will be explored later.

4.3 Delete

As previously mentioned, `delete` has been considered as the least important operation for the implemented hash table. However, it is still interesting to see how the table compares to the others, despite not being designed for fast deletion. Therefore, this benchmark follows these steps:

1. Insert x elements and delete every element. Measure the average deletion time in nanoseconds.
2. Increase x and repeat step 1 until the table reaches 10 million elements.

Results

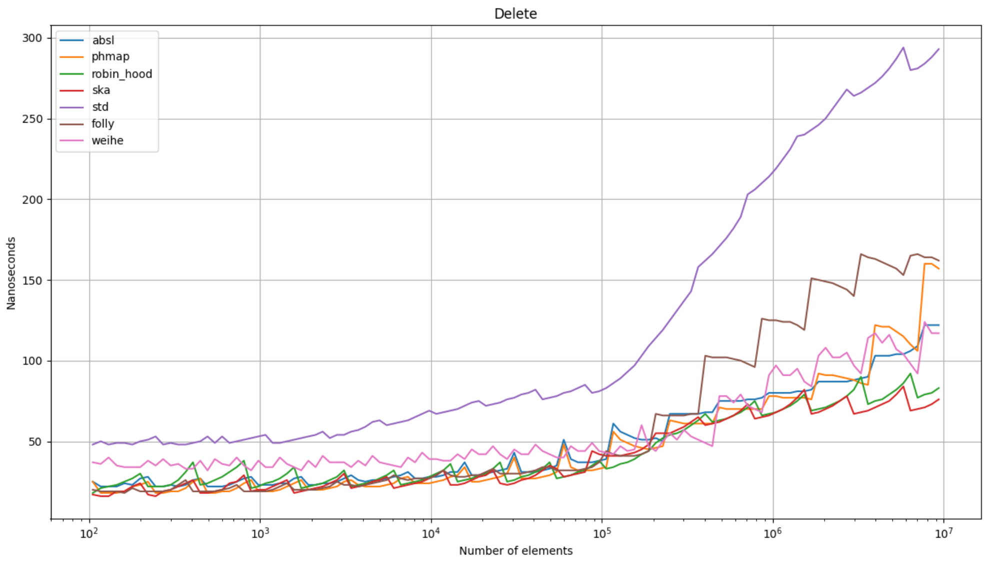

Figure 4.5: Average deletion time.

Reflection on the results

The curves for most of the tables look a lot like they did for search. This is because deletion is essentially search where you back-shift the elements. My implementation back-shifts and marks key-value pairs with tombstones, meaning that when the number of deleted key-values exceeds 50% memory, they get wiped out. This is the reason why the graph for `weihe` is a bit jumpier. While my table does not have that good deletion time, these results are still very satisfactory.

4.4 Memory usage

This benchmark has been performed with the following steps:

1. Initialize the hash table and measure the resident set size (RSS) - memory held in RAM.
2. Insert 100 elements and measure the difference in RSS.
3. Repeat step 2 until table reaches 10 million elements.

Results

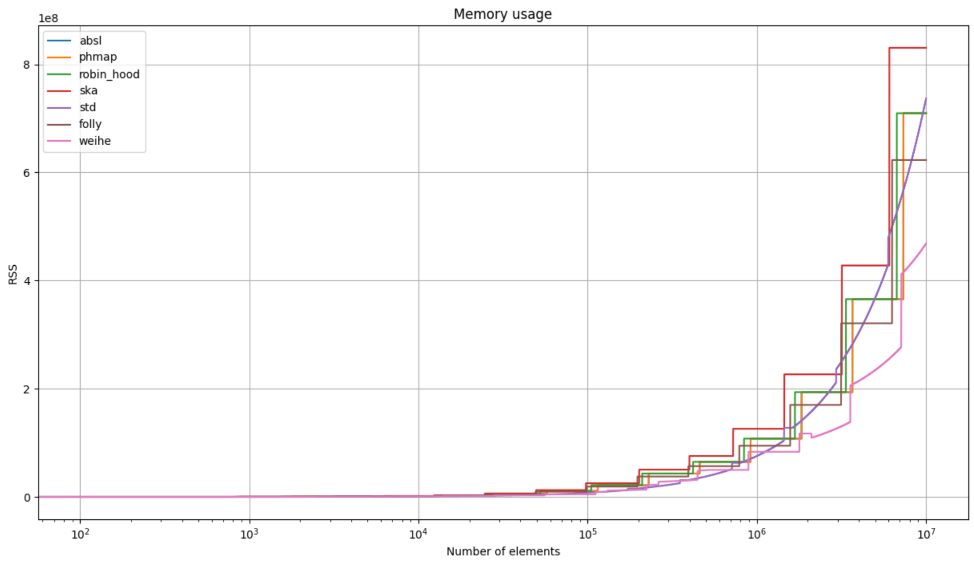

Figure 4.6: Memory usage of the hash tables.

Reflection on the results

For this benchmark, we again see that the curves go up when the tables resize. The reason why we don't see a rise in RSS for most of the tables between the points where they resize is that `std::string` has a size of 32 bytes in GCC, meaning that when keys of 8 bytes are inserted - nothing gets allocated. This is good for the performance of the set, but these maps have to store 32 bytes for every entry in the table. If the table has a load factor of 0.8 and a growth factor of 2, then 20-60% would be empty entries, and if the strings take up 32 bytes while being empty, then a lot of space gets wasted. This is only the case if the tables store the entries without indirection/pointers, and most of these tables have equivalents with node indirection. While `weihe` stores the 32-bit hash, it actually doesn't use that much memory because the key-values are stored consecutively in an array with a low growth factor.

Chapter 5

Future work and conclusion

5.1 Future work

The implementation is not close to having feature parity with `std::unordered-_map`, so this is an area where much work could be put in. In detail, this would be making it generic such that it works with all types of keys and values, and methods like `reserve`, `clear`, etc., should also be added. It would also be appropriate to make a C++ wrapper such that the syntax is more similar to `std::unordered_map`. Ideally it should be a drop-in replacement for `std::unordered_map`.

The implementation is fast, but we can always optimize. One optimization could be not to store indices as `uint32_t`, but instead, use the smallest type possible for every size of the table. For example, use `uint8_t` if the table has capacity for less than 256 elements. There are naturally other places that possible could be optimized. It would also be very interesting to use some of the knowledge gained from this project to build a hash-table with memory usage as low as possible. The keys could, for example, be stored consecutively in memory with only null bytes in-between instead of storing the length of every key.

The most exciting future prospect for this implementation is that it will be used in the standard library in the V programming language - a new language, but one of the most prominent open-source programming languages on GitHub with 424 contributors and 20 paid developers in the V organization. I am part of the V organization, and it has been decided that the current implementation should be replaced by this implementation when it has feature parity and passes all tests, reviews and benchmarks.

5.2 Conclusion

This project aimed to investigate and devise several optimization techniques for hash tables and examine how established solutions have been implemented. Several popular and very fast hash-tables, including implementations by Google and Facebook, have been investigated. The general trend is that these hash-tables are either implemented as Robin Hood or Swiss tables. A new hash table with a new technique based on Robin Hood hashing has been implemented in C/C++. The hash table has been compared to some of the fastest hash-tables, but also `std::unordered_map`. `search`, `insert`, `delete` and memory usage have been benchmarked and measured. While my implementation is limited in functionality compared to the others, it performs very well in some benchmarks due to its alternative design. It is the fastest for unsuccessful lookups and insertions by a good margin. It also has quite good memory consumption; however, it is second to slowest for deletion. The focus has been `search`, `insert`, `delete` - in that order of priority. While this project's goal was not to build a faster hash table than other implementations, it is still a big feat to beat the established implementations in some benchmarks, and lots of interesting information could be deduced from the results. The primary goal of building this hash table was to understand how hash tables work and how they are implemented in popular libraries - this has proven effective.

Bibliography

[1] greg7mdp (Gregory Popovitch). *The Parallel Hashmap*. 2021. URL: https://github.com/greg7mdp/parallel-hashmap.

[2] Martin Ankerl. *Hashmaps Benchmarks - Find 1 - 1M std::string*. 2019. URL: https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-04-05-result-RandomFindString_1000000/.

[3] Sam Benzaquen et al. *Swiss Tables Design Notes*. 2021. URL: https://abseil.io/about/design/swisstables.

[4] *Breaking Murmur: Hash-flooding DoS Reloaded.* https://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/. Accessed: 2021-06-01.

[5] Pedro Celis, Per-Ake Larson, and J. Ian Munro. “Robin Hood Hashing”. In: *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. SFCS ’85. USA: IEEE Computer Society, 1985, pp. 281–288. ISBN: 0818608444. DOI: 10.1109/SFCS.1985.48. URL: https://doi.org/10.1109/SFCS.1985.48.

[6] Facebook. *F14 Hash Table*. 2021. URL: https://github.com/facebook/folly/blob/master/folly/container/F14.md.

[7] Free Software Foundation. *hash_bytes.cc*. https://github.com/gcc-mirror/gcc/blob/master/libstdc++-v3/libsupc++/hash_bytes.cc. 2021.

[8] Maurice Herlihy. *Hopscotch Hashing*. 2008. URL: https://people.csail.mit.edu/shanir/publications/disc2008_submission_98.pdf.

[9] Raymond Hettinger. *[Python-Dev] More compact dictionaries with faster iteration*. 2012. URL: https://mail.python.org/pipermail/python-dev/2012-December/123028.html.

[10] Intel. *Intrinsics Guide*. 2021. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[11] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel. 2020. URL: `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html`.

[12] Matt Kulukundis. *Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step*. YouTube. 2017. URL: `https://www.youtube.com/watch?v=ncHmEUmJZf4&t=217s`.

[13] M. Thorup. "High Speed Hashing for Integers and Strings". In: *ArXiv* abs/1504.06804 (2015).

[14] Mikkel Thorup. *On the k-Independence Required by Linear Probing and Minwise Independence*. 2014. arXiv: `1302.5127 [cs.DS]`.

Chapter 6

Appendix

6.1 Search versions

Version 1

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  uint32_t mask = ~((uint32_t)1 << 33);
  uint32_t hash = wyhash(key, len, 0, _wyp) & mask;
  uint32_t index = (hash >> (h->shift)) * 2;
  while (hash > h->buckets[index]) index += 2;
  do {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *pmatch = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(pmatch, key, len) == 0)) {
          // Replace value
          int *dest =
              (int *)(h->key_values + kv_index +
              ↪  sizeof(uint32_t) + len);
          return *dest;
        }
      }
    index += 2;
  } while (hash == h->buckets[index]);
  // No match
  return 0;
}
```

Version 2

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  uint32_t mask = ~((uint32_t)1 << 33);
  uint32_t hash = wyhash(key, len, 0, _wyp) & mask;
  uint32_t index = (hash >> (h->shift)) * 2;
  while (hash > h->buckets[index]) index += 2;
  while (hash == h->buckets[index]) {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *pmatch = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(pmatch, key, len) == 0)) {
          // Replace value
          int *dest =
              (int *)(h->key_values + kv_index +
              ↪  sizeof(uint32_t) + len);
          return *dest;
        }
      }
    index += 2;
  }
  // No match
  return 0;
}
```

Version 3

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  uint32_t mask = ~((uint32_t)1 << 33);
  uint32_t hash = wyhash(key, len, 0, _wyp) & mask;
  uint32_t index = (hash >> (h->shift)) * 2;
  do {
    if (hash == h->buckets[index]) {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *pmatch = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(pmatch, key, len) == 0)) {
          // Replace value
          int *dest =
              (int *)(h->key_values + kv_index +
              ↪  sizeof(uint32_t) + len);
          return *dest;
        }
      }
    }
    index += 2;
  } while (hash >= h->buckets[index]);
  // No match
  return 0;
}
```

Version 4

```c
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  uint32_t mask = ~((uint32_t)1 << 31);
  uint32_t hash = wyhash(key, len, 0, _wyp) & mask;
  uint32_t index = (hash >> (h->shift)) * 2;
  do {
    if (hash == h->buckets[index]) {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *potential_match = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(potential_match, key, len) == 0)) {
          // Replace value
          int *dest = (int *)(potential_match + len);
          return *dest;
        }
      }
    }
    index += 2;
    if (hash == h->buckets[index]) {
      uint32_t kv_index = h->buckets[index + 1];
      uint32_t key_len = *(uint32_t *)(h->key_values +
      ↪  kv_index);
      if (_likely_(len == key_len)) {
        char *potential_match = (h->key_values + kv_index +
        ↪  sizeof(uint32_t));
        if (_likely_(memcmp(potential_match, key, len) == 0)) {
          // Replace value
          int *dest = (int *)(potential_match + len);
          return *dest;
        }
      }
    }
    index += 2;
  } while (hash >= h->buckets[index]);
  // No match
  return 0;
}
```

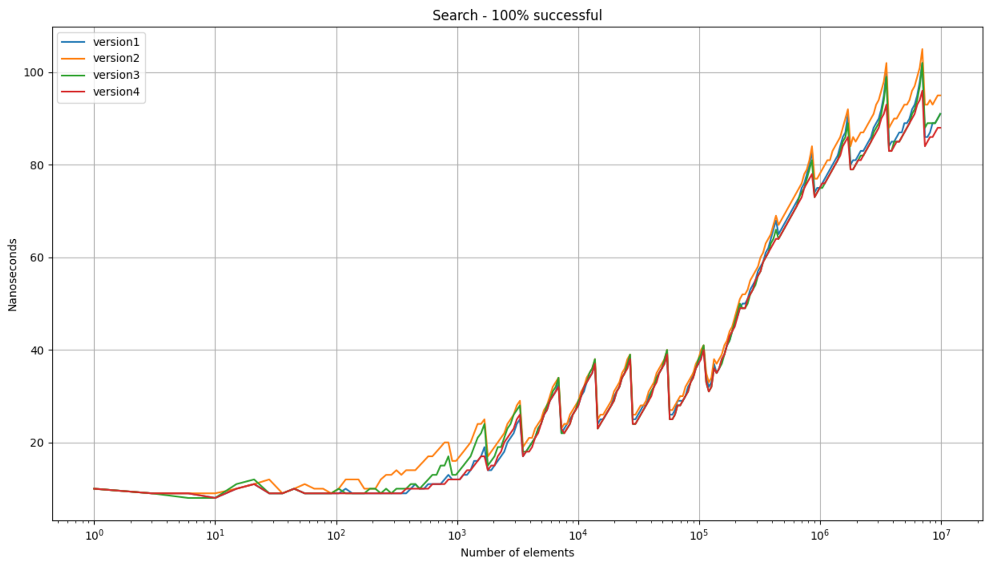

Figure 6.1: Average (100% successful) search time in nanoseconds with different search versions.

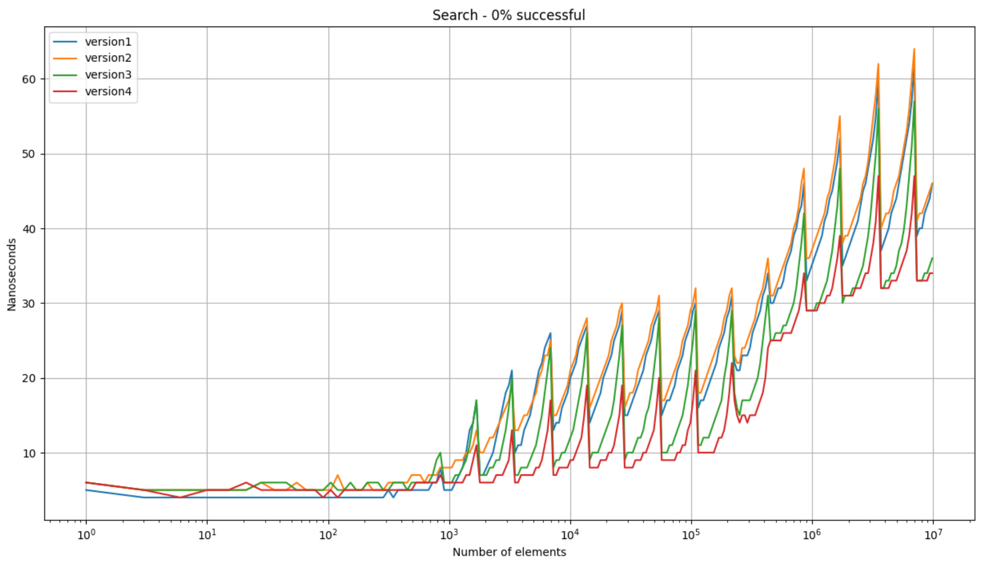

Figure 6.2: Average (0% successful) search time in nanoseconds with different search versions.

6.2 memcmp

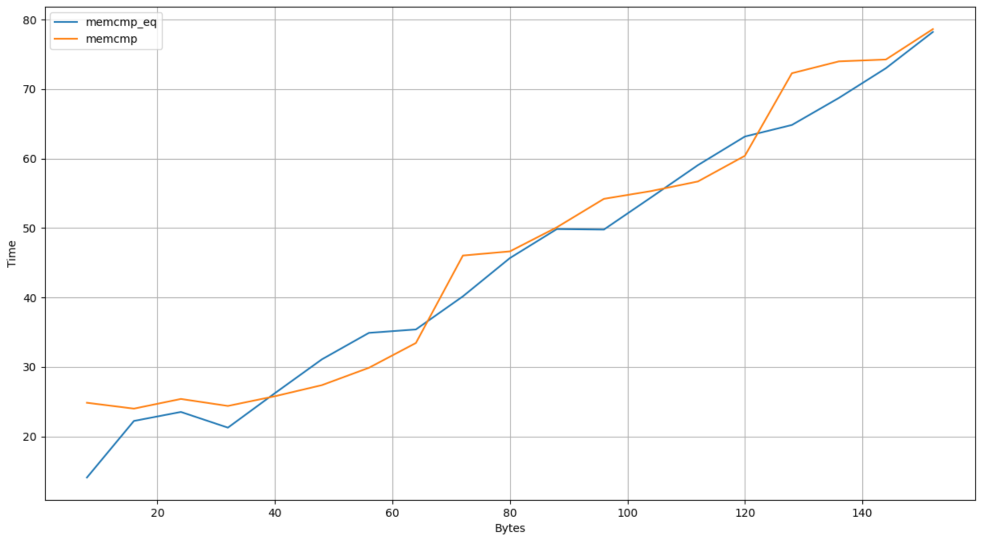

Figure 6.3: Benchmark of my `memcmp_eq` vs `memcmp` from libgcc. Lower time is better.

6.3 Benchmarks with 30 byte keys and std::hash

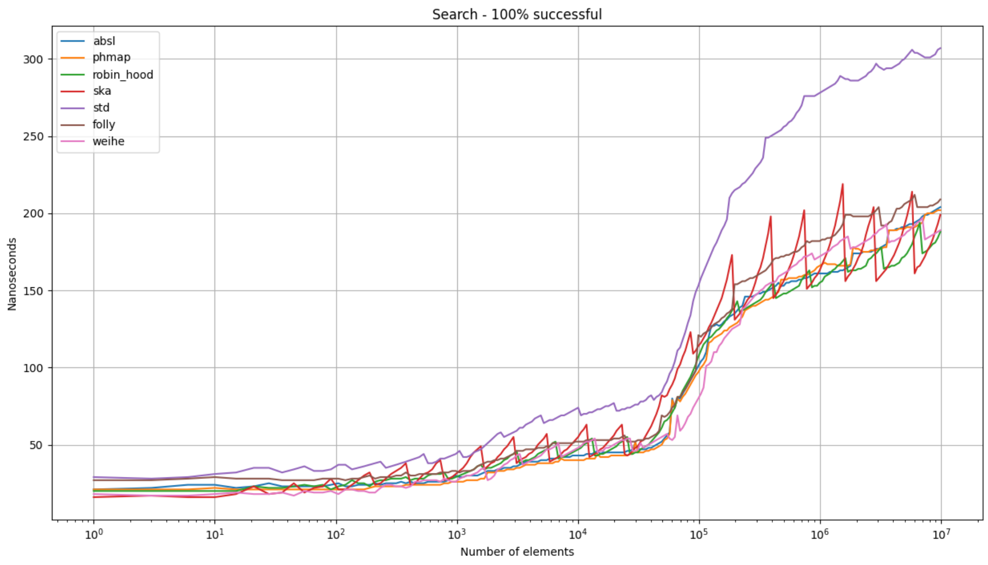

Figure 6.4: Average search time with 100% successful lookup. 30 byte keys.

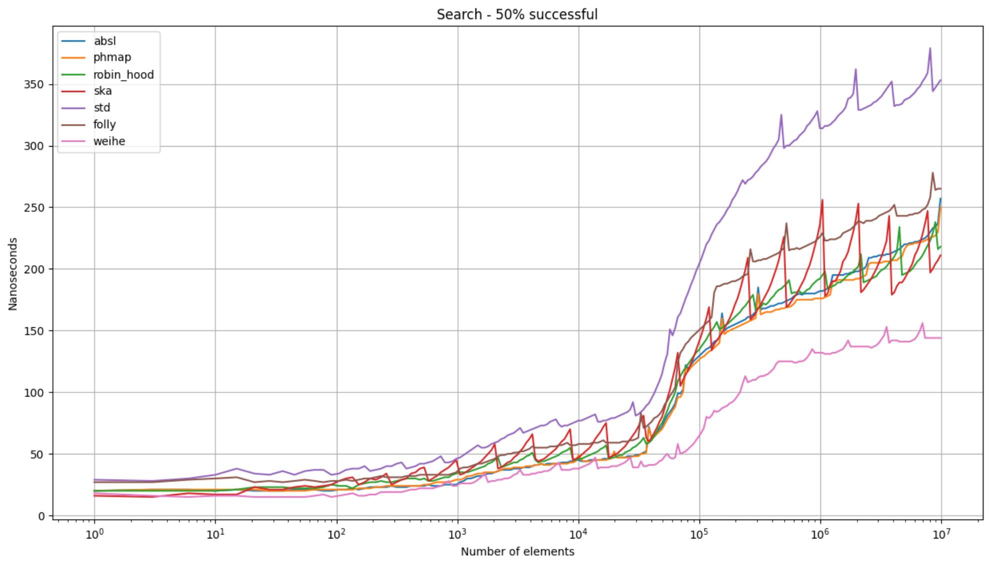

Figure 6.5: Average search time with 50% successful lookup. 30 byte keys.

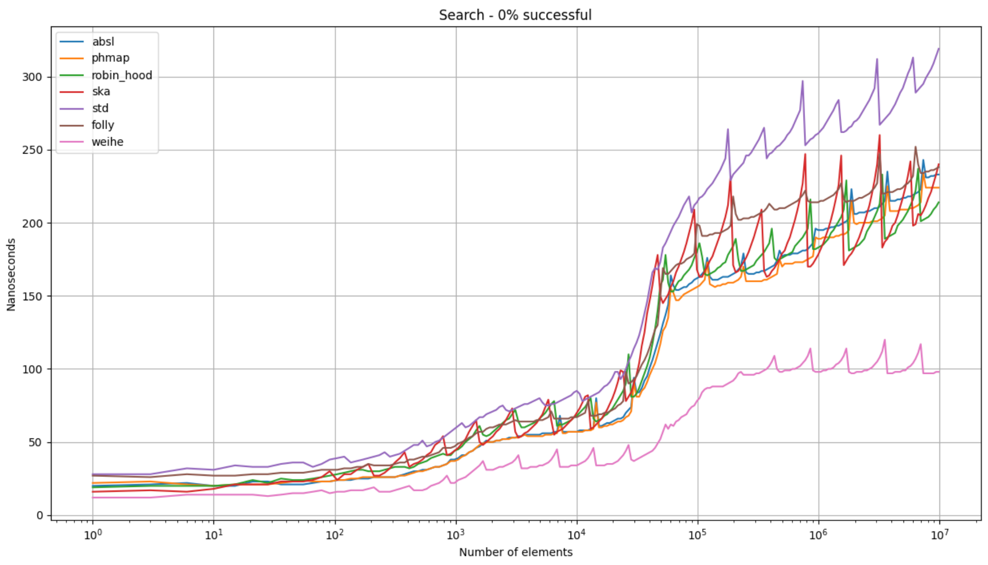

Figure 6.6: Average search time with 0% successful lookup. 30 byte keys.

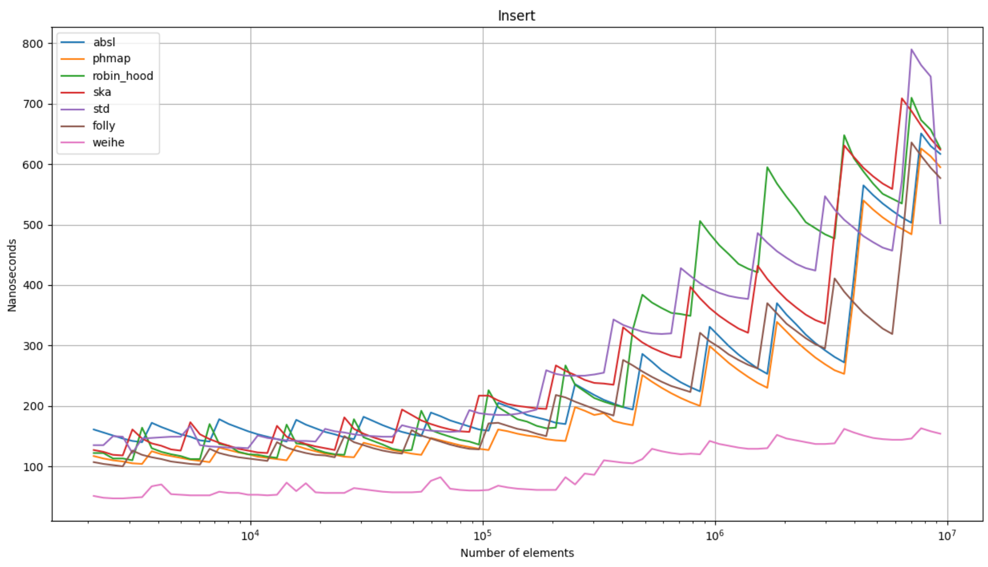

Figure 6.7: Average insertion time. 30 byte keys.

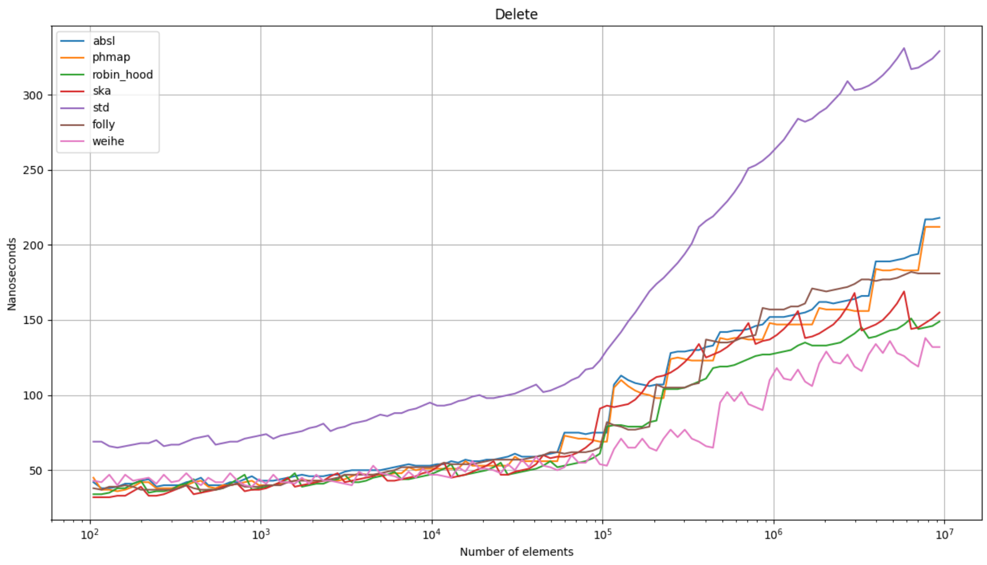

Figure 6.8: Average deletion time. 30 byte keys.

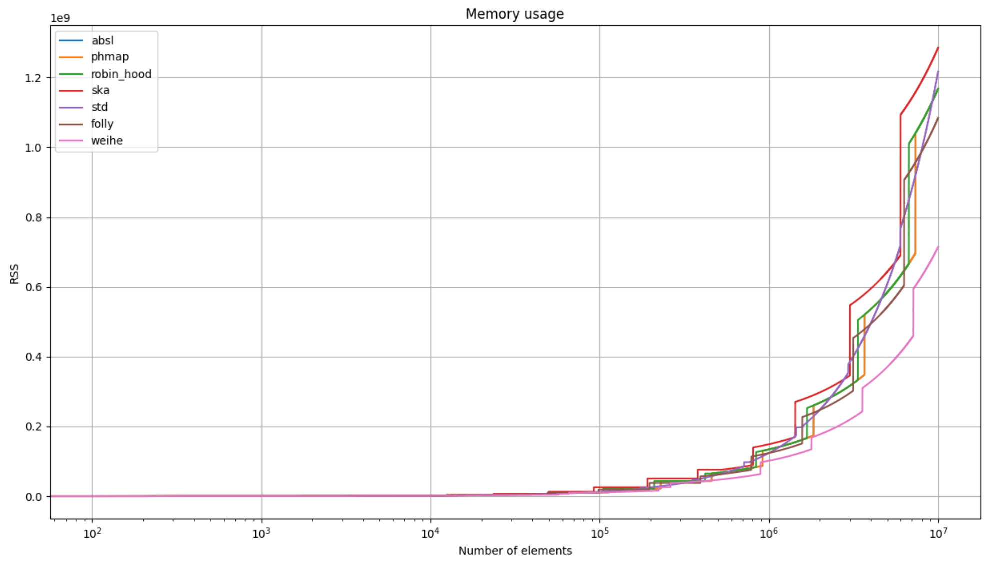

Figure 6.9: Memory usage (RSS). 30 byte keys.

6.4 Benchmarks with fast-forward

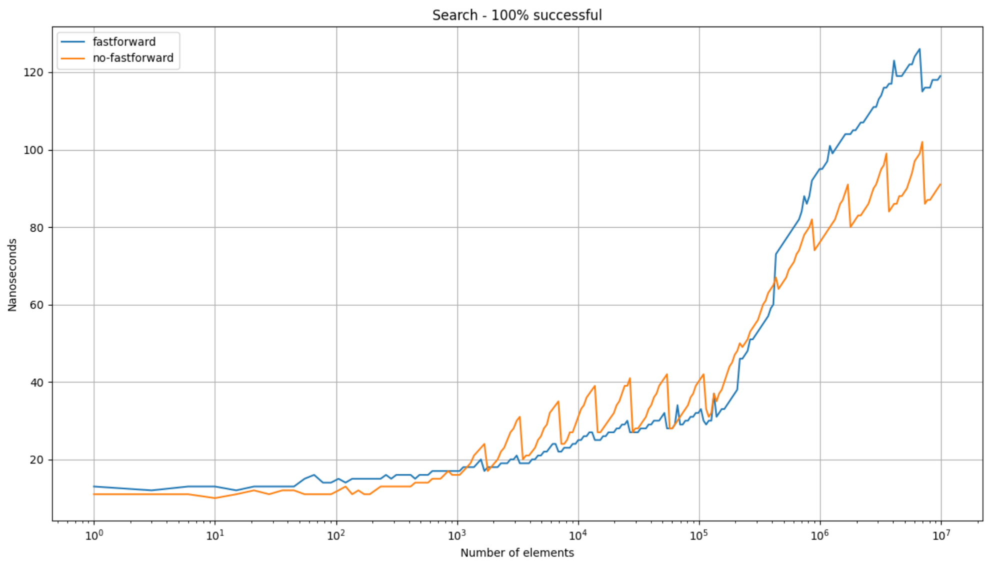

Figure 6.10: Average search time with 100% successful lookup, 8 byte keys. With- and without fast-forward.

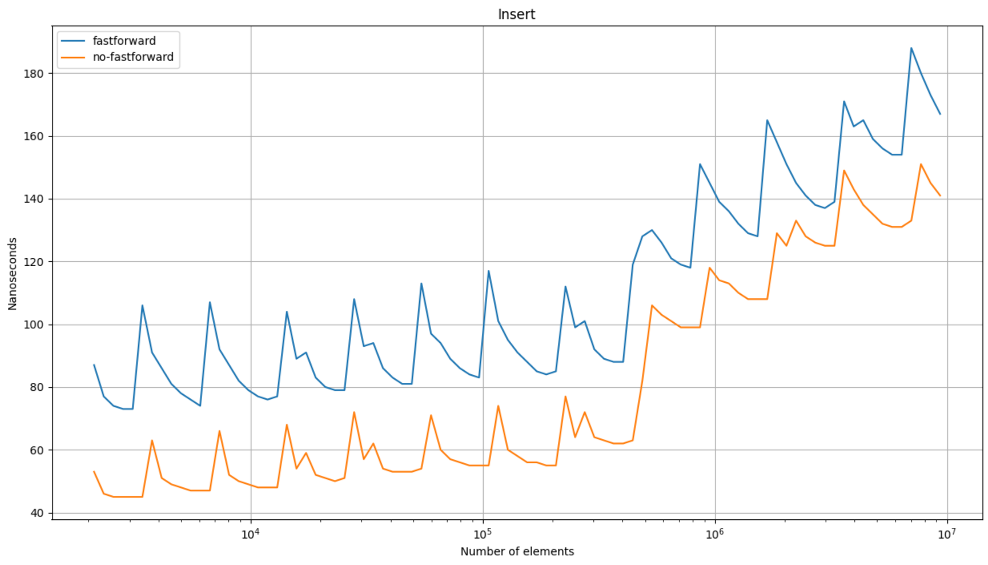

Figure 6.11: Average insert time. 8 byte keys, with- and without fast-forward.

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  // find hash and index
  // ...
  while (hash > h->buckets[index]) index += 2;
  do {
    // check for key equality
    // ...
    index += 2;
  } while (hash == h->buckets[index]);
  // No match
  return 0;
}
```

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  // find hash and index
  // ...
  while (hash > h->buckets[index]) index += 2;
  while (hash == h->buckets[index]) {
    // check for key equality
    // ...
    index += 2;
  }
  // No match
  return 0;
}
```

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  // find hash and index
  // ...
  do {
    // check for key equality
    // ...
    index += 2;
  } while (hash >= h->buckets[index]);
  // No match
  return 0;
}
```

```
inline int hget(struct Hashmap *h, char *key, uint32_t len) {
  // find hash and index
  // ...
  do {
    // check for key equality
    // ...
    index += 2;
    // check for key equality
    // ...
    index += 2;
  } while (hash >= h->buckets[index]);
  // No match
  return 0;
}
```