FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor Thesis

Yu Brian She & Peter Alexander Garnæs

Hopscotch Hashing in Rust

Jyrki Katajainen

June 2014

Contents

Hopscotch Hashing on Rust

Yu Brian She[1] Peter Alexander Garnæs[2]

Department of Computer Science, University of Copenhagen
Universitetsparken 5, DK-2100 Copenhagen East, Denmark
`nyamo70@gmail.com`
`petergarnaes@gmail.com`

Abstract. For a long time, C++ has been the dominant force in the field of programming, be it low level programming or high level programming. Many different alternatives have been made, but none have yet succeeded. We have looked at a new alternative called Rust. Rust is a new, and not officially released programming language with a new approach to programming and safety. While in C++, high level code can be executed with very low runtime cost, it usually suffers from segmentation faults. In Rust, code can be executed without the fear of segmentation faults with minimal runtime cost. This paper is an attempt to implement Hopscotch in Rust, is a hashing scheme that uses open addressing to resolve hash collisions. It focuses on good cache locality, and performs well on high load factors. In our series of tests and benchmarkings, we not only test Rust up against C++, we also test the performance of Hopscotch compared to another cache locality friendly algorithm called Robin Hood. Our results looks appealing, as not only does Hopscotch outperform Robin Hood in nearly every aspect, Rust performs almost as well as C++, and it may even be further improved, if memory safety guarantees is sacrificed.

June 9 2014

1. Introduction

Rust is a programming language designed by Mozilla[1], which has been steadily growing in popularity in the programmer community on the internet. To test Rust, we implement a dynamically resizable hashing scheme in Rust called Hopscotch, which is introduced to us in Herlihy[3]. We compare the runtime of a Rust implementation to two other hashmap implementations: Robin Hood, the standard Hashmap implemented in the Rust standard library and a C++ implementaion of Hopscotch made by Henrik Thorup Andersen[1]. The goal of this paper, is to compare Rust to C++. The comparison will focus around the performance of our Hopscotch implementation to Robin Hood and Henrik[1].

2. Hopscotch Hashing

Hopscotch Hashing is an open addressing scheme for collision resolution in hash tables. This scheme is was first introduced in the paper by Maurice Herlihy, Nir Shavit and Moran Tzafrir[3]. The approach is based on the concept of neighbourhood, where the neighbourhood consist of itself and H-1 following buckets, where H is a constant usually corresponding to the word size of the machine. The neighbourhood is what defines what we call a 'virtual bucket', where a value hashed to a bucket, always can be found in the corresponding virtual bucket.

Each bucket contains a bitmap representing the neighbourhood of the bucket. We call this bitmap the hop-information. The hop-information for Bucket A contains data on which buckets within the neighbourhood contains values which are hashed to Bucket A. This hop-information is useful when looking through the hash table, as we are able to look for hash collisions with limited linear probing, becuase we only need to look at the buckets indicated by the hop-information. In worst case, a lookup through the hopinformation takes constant time.

Adding an item into the hash table takes three steps to execute:

1. Starting at index i, using linear probing to find the closest empty bucket at index j.
2. If the emptry entry's index j is within H-1 of i, place the item there, update hop-information and return.
3. Otherwise, j is too far from i, create an empty entry closer to i, by finding an index y that lies between i and j, but within H-1 of j. This y must be able to move to the free index found in step 2, without leaving its virtual bucket. Displacing y to j creates a new empty slot closer to i. If this index isn't within the virtual bucket of i we repeat. If no such item exists, or if the bucket i already contains H items, we resize and rehash the table.

[1] `http://www.mozilla.org/en-US/research/projects/`

What stands out in hopscotch hashing, is the idea of moving the empty slot closer towards the desired bucket, compared to f.ex. linear probing, where it would just leave it where it was found. In the paper from Herlihy, Shavit and Tzafrir[3] it is shown that the insertion with rehashing and resize can be done in linear time and that insertion without rehashing and resize can be done in constant time, thus resulting in an amortized constant time when inserting in the hash table.
Removing an item in Hopscotch hashing is as simple as removing the reference in the hopinformation of the bucket. Other traits which Hopscotch gives, is great cache locality and parallelism. The great cache locality of Hopscotch is reasoned by the fact, that lookups at most has 2 cache invalidations[3, p. 3]. Hopscotch is good for parellel programming, as each operation only uses a small segment of the hash table, which makes synchronization simpler.

We want to compare hopscotch to another cache friendly algorithm called Robin Hood, and look at its concurrency. We are doing this in a new system programming language, that as of spring 2014 is still being developed, called Rust. Rust was chosen because it introduces new and exciting aspects and philosophies that are defitinely worth exploring, while still being a low level systems programming language. Rust standard library also conveniently contains a Robin Hood hashmap that we can compare our implementation too.

3. Rust introduction

Rust is a programming language designed by Mozilla[2], which has been steadily growing in popularity in computer science community on the internet. As a systems programming language Rust is aimed to be an alternative to C++, and its main features are:

- Compiler checked memory safety
- Optional garbage collection
- Compiler checked concurrent memory safety
- The performance of low level compiled machine code

Furthermore Rust have a ton of secondary features including but not limited to:

- Rust code can make C bindings, and by use of certain libraries even be written like C (with malloc, null and copyable pointers)
- Rust offer object oriented features like traits and inheritance
- Rust offer functional programming features like higher order functions, algebraic data types and pattern matching
- Cleaner syntax because of type inference

To implement the compiler-checked memory safety, the compiler enforces a set of rules which makes writing Rust programs different from other lan-

[2] `http://www.rust-lang.org`

guages. One of these main rules is explicit mutability. With explicit mutability as part of the type system, the compiler can make sure you are not referencing the same variable in two different places, unless it is immutable (read only). It also ensures you can never alter an immutable variable. To ensure further memory safety Rust also implements a compiler checked lifetime and ownership system. Ownership and borrowing ensures that a piece of memory can only be referenced by one vairable at a time. This ensures memory safety because it makes it possible to track a variables lifetime (to know when it is valid to use) and prevents segmentation faults and accessing wrong memory from occuring.

To ensure full memory safety the language also requires pointers to always point to valid memory. With this rule the compiler avoids null pointers and segmentations faults, and thanks to the lifetime and ownership rules, the compiler knows what memory is valid to point to.

These rules sometimes force you to write ineffecient code, because we cannot simply null check or manipulate pointers directly. This is why Rust allows you to write explicitly unsafe blocks of code, where you can clone pointers and call unsafe functions in the library that can be used for more optimal code. The philosophy is that these blocks should be small and provide simple features that can easily be reasoned about and debugged. This way potential failures are fewer, because the unsafe code is carefully written, and even in case of segmentation faults, less code is needed to be looked through.

This memory safety philisophy is also used in the concurrency system, which are lightweight memory isolated threads, called tasks. Tasks share data through message passing, and transfering mutable data between tasks is a matter of passing a message with a reference to the data, which the memory safety system ensures is the only reference to the data. This way the lifetime system philosophy carries over into the concurrency system, because it makes sure only one task at a time possesses the data, by either lending it out or transfering its ownership.

If you wish to have several references to a mutable piece of data, which you can only do through unsafe code blocks, Rust also provides locks. But even the locks in Rust make use of the lifetime and ownership model to ensure memory safety. In Rust, a mutex lock owns some datastructure, which means that to get a reference to it, you must take the lock, which will return you that reference. Thanks to the lifetime system, when you unlock, your reference becomes invalid, and the compiler has made it impossible to access the datastructure without locking.

As with all programming languages, Rusts features comes with a tradeoff, and in the case of Rust, these rules might limit you in the things you want to do. One example is the way multithreading is handled. Instead of adhearing to the pthreads system, Rust uses lightweight threads, that Rust distributes on the available operating system threads threads.

Rust often has different ways of doing things, so learning Rust is not trivial even with some background in programming languages. Rusts syntax and features is still being changed and expanded, and there are much to be done, and much left out intentionally.

4. Implementation

For our implementation, we have two sections: One for our single threaded implementation and one for our multithreaded implementation. We are sticking to writing only safe code in Rust, to assess its usefulness and effeciency when using it as intended. The multithreaded implementation is incomplete and which will be discussed in a later section.

4.1 Single threaded Hashmap implementation

This section will mostly discuss what choices and changes that were made compared to the given Hopscotch Hashing algorithm[3]. As mentioned, Rust will sometimes enforce inefficient coding compared to C/C++, which as well, will be addressed. Our single threaded implementation of Hopscotch hashing consists of two parts: A rawtable, where the data is stored, and the hashmap implementaion, which executes the hopscotch algorithm.

4.1.1 Rawtable

Our rawtable stucture for the single threaded implementation is the following:

Listing 1 : Rawtable structure

```
#[deriving(Show,Clone,Eq)]
pub struct RawTable<K,V>{
    capacity: uint,
    buckets: Vec<Bucket>,
    keys: Vec<Option<K>>,
    vals: Vec<Option<V>>
}
```

Note that the buckets, the keys and the data are stored in seperate arrays, called vectors in Rust. This choice is made for optimizing cache locality of each bucket, so that fewer cache lines are needed to scan a virtual bucket. The downside is that when we call `find_closer_bucket` the keys and values that must be moved do not have cache locality.

Buckets in this implementation are only used to store neighbourhood information and a hash. When finding the right bucket our key resides in, the array index of the bucket is used to find the value in the value array.

Listing 2 : Bucket structure

```
#[deriving(Show,Clone,Eq)]
pub struct Bucket{
    pub hop_info: u32,
    pub hash: u64
}
```

Because of word allignment, we are wasting 32 bits of memory in each bucket on 64 bit machines. This is because we have decided to use a virtual bucket capacity of 32, although 64 is definitely possible.

4.1.2 Hashmap

In our Hashmap implementation, we tried to deviate as little as possible from the implementation of Hopscotch hashing[3]. Some of the deviations are choices we have made, in an effort to optimize the algorithm and maybe suit the Rust philosophy better, and some are simply enforced by the Rust compiler. Most notably we use arrays to and indexes to access the hashmap, instead of just incrementing pointers as the given algorithm does[3]. We'll describe the four main functions: remove, lookup and insert one by one, and how they have been implemented.

lookup
When performing a lookup with hopscotch, we use the hop information as an indicator of which buckets we need to look at in the hash table.

We then use the hashed key and the hash stored in the bucket and compare them to each other. If the hop information indicates that the bucket contains an item, and that the hashes matches each other, we have a hit. If we don't get a hit, we simply look at the next bucket and checks again. In case we don't get a hit, the lookup function returns None, if the lookup function gets a hit, we return value in a wrapper as an Option: Option<value>. Options are used, because Rust doesn't have null.

remove
The remove function removes the reference in the hopinformation and , and the data is returned. Subsequent inserts in removed buckets will overwrite the values previously there. To do this, we use four steps:

1. step: Loop through the virtual buckets hop information, to check if the current bucket isn't empty.
2. step: When a bucket in the neighourhood that isn't empty is found, we check if the hash is corresponds, and if it does not, we continue.
3. step: If the hash corresponds, we remove the reference of the bucket in the virtual bucket, and return the removed value as an Option<value>.

insert
When inserting values in the hopscotch table, we probe through the virtual bucket to compare the keys, to find an empty spot. This is arguably the

weakpoint of hopscotch, because insertion uses linear probing to find a free bucket, and must furthermore swap buckets untill it lies within the right virtual bucket (neighbourhood). Insertion in the hash table is done in the following steps:

1. step: Check whether or not the key is already inserted in the hash table, if not we continue to step 2, if it is we return false.
2. step: Use linear probing to find an empty bucket X to perform an insertion on.
3. step: Then check if the bucket X is a within ADD_RANGE of current bucket. If not, we resize the hash table and retry the insertion.
4. step: If step 3 is a success, check if the empty bucket is within H range of the current bucket. If so, insert the bucket into the hash table. If not, try to displace the empty bucket closer X to the current bucket.
5. step: If step 4 is not successful, resize the hash table and retry the insertion again.

Our implementation can be seen in the appendix. Overall our implementation matches the algorithm of the original implementation by Herlihy, Shavit and Tzafrir[3]. Here's a list of choices we've made in our implementation:

1. We've splitted the hash table into three different vectors, to gain beter memory locality.
   There were two reasons for this implementation choice to be made. First, by splitting up the hash table into three different vectors, we achieve better cache locality for the buckets, and we can fit in more buckets in a cache line. The Rust compiler also forbids us from having pointers to keys and values in our buckets, that we can increment. Our compromise was to let corresponding buckets, keys and values have the same index in their respective vectors.
2. Our vectors for our keys and data are Vec<Option>.
   Hopscotch hashing uses linear probing when performing an insertion. This requires that empty buckets points towards NULL, as to check if the bucket is empty. NULL-pointers aren't supported in Rust. By using Options for our keys and data, it's possible for us to check if the bucket is empty by pattern matching. Though this solution increases overhead of our Rawtable[3] by having to pattern match, and increases memory usage by one data-word for each key and value, it is our only alternative.
3. When initiating a Hashmap, we always set the capacity to 2^n.
   Our hashmap implementation is a circular array. To make the indexing wrap around when reaching the end of the array we can use the bitwise operator AND if the table size is 2^n. This is much more effecient than modulo operations, and doubling the size of the hashmap at each resize seems a good approach.

[3] section 3.1.1: Rawtable

4. We've set H to 32 and ADD_RANGE to 256.
   Both variables, H and ADD_RANGE influence how long it takes to insert and lookup keys, because they influence how many buckets are searched. The greater these values are, the more buckets are searched in order to find. We've chosen to set H to 32, as this can hold all keys hashed to that bucket, even for very big hash tables. We have set ADD_RANGE to 256, because we found it decreased the need for resizing on larger tables. This meant that even large tables could operate with high load factors, which we found was where hopscotch really excels. These tests are further discussed in Section 5.2.1 Resize-ADD_RANGE test.

4.2 Multi threaded Hashmap Implementation

In out attempt to implement a concurrent Hopscotch in Rust, we have concluded that this was not a possibility.

Shared mutable memory is barely supported in Rust because of its unsafe nature. As explained in section 3, Rusts lifetime and ownership system prevents you from accessing memory concurrently in safe mode, unless they are guarded by locks.

The Rust safe mode locking system is very strict, because the data to be accessed must be guarded. This led us to another fundemental problem, where we could not read data without acquiring any locks. This is done in in the hopscotch algorithm when doing a lookup, because hopscotch instead uses timestamps to assess if data was changed while doing the lookup. We then had to use Read-Write locks on every bucket, which is far more ineffecient than what was described in the original paper[3].

We then ran into a huge problem, because in order to access data guarded by a lock, we create a reference. In our specific case, when we lock a bucket, we create a reference into our hashmap. Because of the lifetime system, we can only hold one reference into a datastructure at a time. This prevents us from locking two buckets at the same time, because they are both references inside the hashmap. Two references are not allowed, because it would make it impossible for the compiler to track the lifetime of the data. This fundemental problem makes it impossible for us to create a concurrent hopscotch implementation in safe code.

Specifically, the problem arises when we both insert and remove, because we need to lock both the virtual bucket with the hop information, and the individual bucket at the same time. This would create two references into the same datastructure, which Rust prohibits.

To avoid this problem, one would have to utilize raw pointers, which only works in unsafe code. Raw pointers have no lifetime, and can be cloned, making two references to the same data. This would make it possible to refer to two buckets at once, and even make abstract locks, ie. locks that does not guard any data.

The safe Rust approach is to run everything as separated as possible, and mostly use message passing to transfer ownership or data. Rust also provides Futures, which allows you to dispatch a subroutine and recieve the result later. In case of the hashmap, one could dispatch a subroutine to access the hashmap, and simply receive the data at a later time.

The Hopscotch algorithm provided in Herlihy[3] can not be safely implemented in Rust. Our assesment is that the safe concurreny system in Rust is developed as support for systems programming, where the software contains many modules that require a low amount of communication.

5. Performance Evaluation

In this section we empirically compare our implementation of Hopscotch hashing against two other hashmap implementations, namely Thorup's[1] implementation of Hopscotch . C++ and the current hashmap implementation in Rust, which uses the Robin Hood algorithm.

5.1 Benchmark Setup

We ran our benchmarks on the following machine:

- i7-4700MQ CPU 2.40GHz
- 8 CPUs
- 8 GB RAM
- OS: Linux Ubuntu v13.10

Our code was compiled with the Rust compiler 0.11.0-pre, which can be found on GitHub.

5.2 Single-threaded Hashmap performance evaluation

5.2.1 Resize-ADD_range test

Hopscotch is a resizable scheme, which use an ADD_RANGE indicator to know when to resize. If we can not find a free bucket within ADD_RANGE, we do not bother trying to find a closer bucket, but instead resize and rehash the table. To optimize our implementation, we needed to test which ADD_RANGE resulted in fewest resizes, when you randomly hash your keys. To test this, we tested what percentage of hashmaps resized when hashing keys randomly at three different add ranges on four different sized hashmaps. We got the following results.

It can be seen on Figure 1-3, that the amount of resizes performed by our implementation doesn't differ much between an ADD_RANGE of 256 and 512. It can be seen in the test results, that if the add range is set too low compared to the table size, its amount of resizes scales dramatically, as it can be seen when the add range is set to 128. As mentioned in section 4.1.2, the add range is set to 256, as it has the best performance. As 256 and 512 performed an equal in all the test scenarios, the lower add range is better suited, as it has a lower linear probing ceiling when inserting compared to a higher add range.

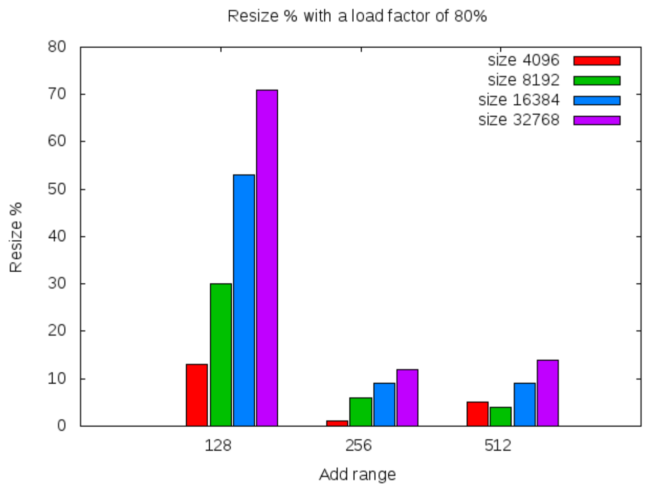

Figure 1: Test 1: Resize test with a 80% load factor.

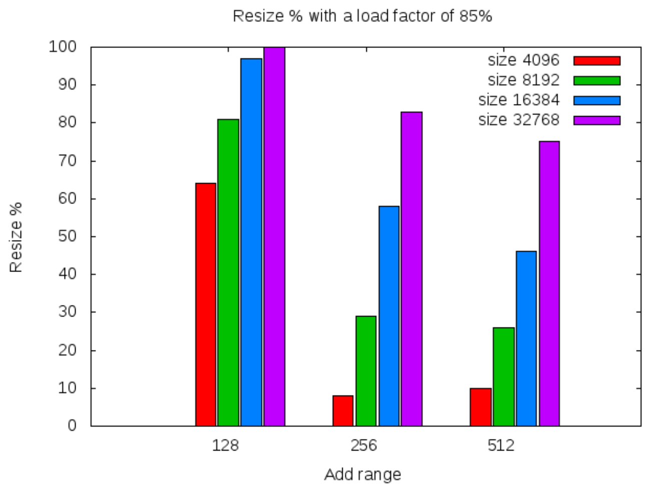

Figure 2: Test 1: Resize test with a 85% load factor.

5.2.2 Our Hopscotch compared to Robin Hood

First we looked at the performance hashing to the same buckets, for both insert and lookup: We find that Robin Hood performs better when inserting into the same bucket, while Hopscotch performs better when all the values hashed to the same bucket are looked up.

To test Hopscotchs performance compared to Robin Hood, we looked at different mixes of operations under different load factors. With this information we can see how the hashmap performs overall, by looking at its

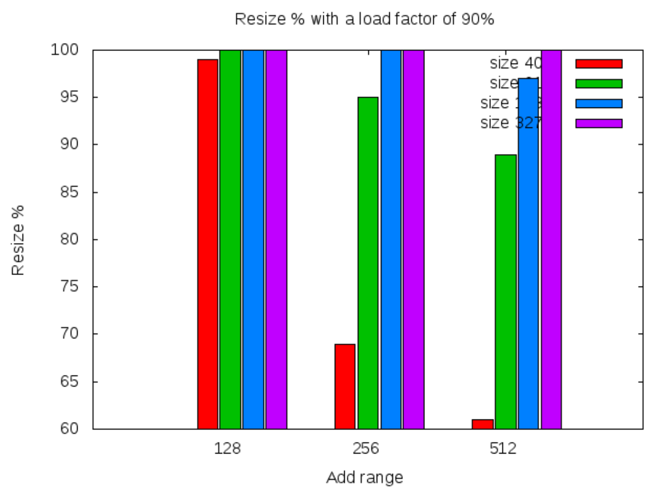

Figure 3: Test 1: Resize test with a 90% load factor.

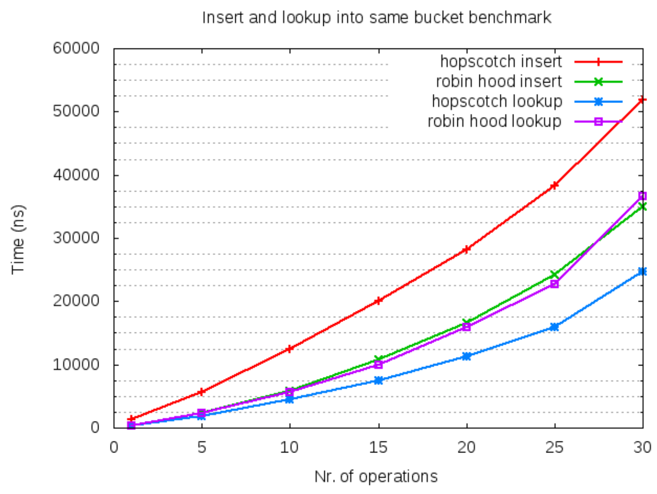

Figure 4: Same bucket test

performance in many different situations. Below we see the results for these benchmarks.

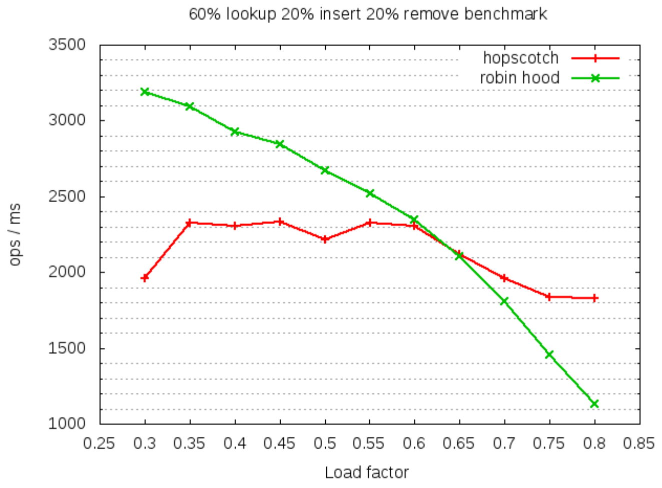

Figure 5: 60% lookups, 20% insertions and 20% removes.

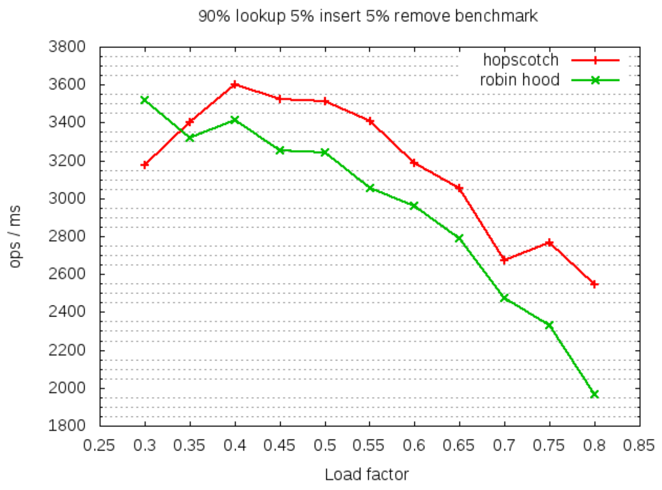

Figure 6: 90% lookups, 5% insertions and 5% removes.

We've chosen two types of action distributions, the first one can be seen in Figure 1, where the distribution is 60% lookups, 20% insertion and 20% removes and the second can be seen on Figure 2, 90% lookups, 5% insertion and 5% removes.

In Figure 1, we see that our implementation runs slower than the Robin Hood when the load factor is low, but with higher load factors, our hopscotch runs faster. We see that Hopscotch has a lower performance penalty on higher load factors, and as a result, manages to out perform Robin Hood on high load factors.

In Figure 2 we see that our implementation is superior from the beginning. By increasing the amount of lookups performed from Test 1 to Test 2, the performance of Hopscotch increased overall and exceeded the performance Robin Hood in all load factors from 30%-80%. Once again we see that Hopscotch suffers less on high load factors, and as a result performs even better on high load factors.

The only difference between the two tests, is the distribution of operations. Comparing the results from Figure 1 and Figure 2, we conclude that our implementations lookup performs better than Robin Hoods lookup overall. We also conclude that hopscotch performs better than Robin Hood on load factors higher than 60%.

5.2.3 Comparing Rust and C++ in singlethreaded environment

To compare our implementation in Rust to an implementation in C++, we have chosen Henrik Thorup ANdersens hopscotch implementation[1]. In order to compare our implementation to Thorups, we ran the same tests Thorup ran to benchmark his own implementation. Our tests find the average insert, lookup and remove time for both Robin Hood, Hopscotch written in Rust and Hopscotch written in C++.

To get a better comparison, we removed the time it takes to hash a key from all implementations, as we found the default hasher in Rust is severely slower than the default hasher in C++ which Thorup uses.

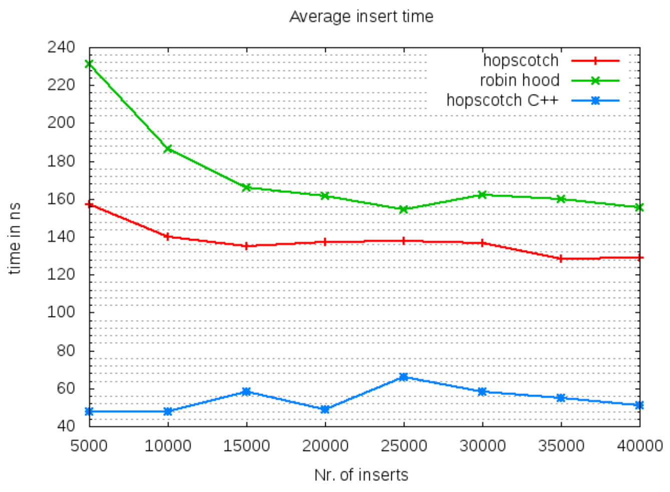

Figure 7: Insert test

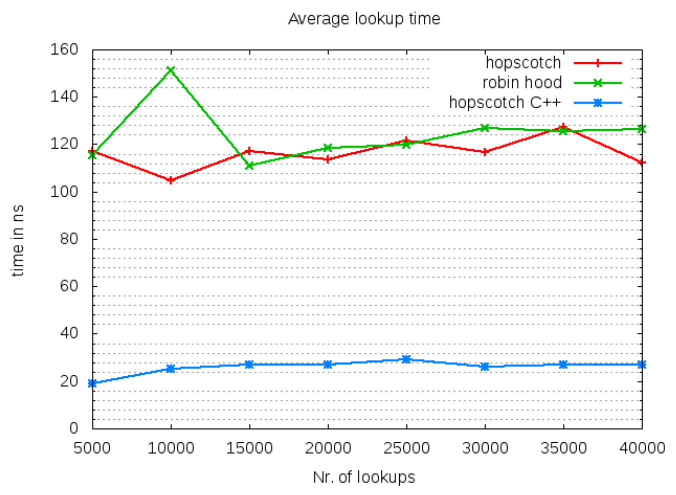

Figure 8: Lookup test

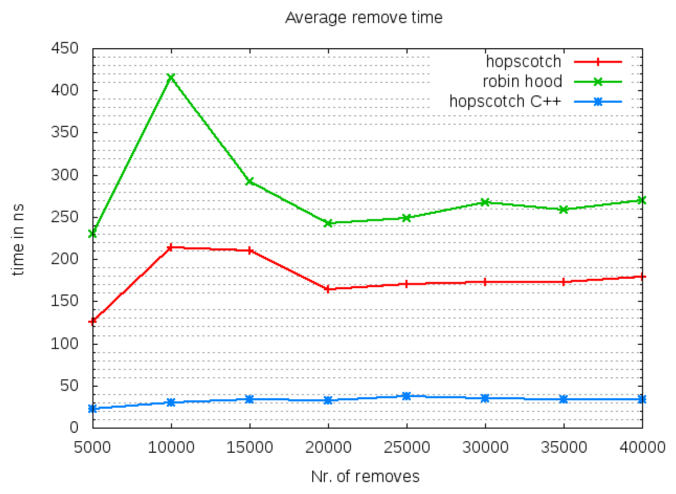

Figure 9: Remove test

All tests show the same result, the implementation in C++ performs better than the implementations in Rust. Even the Robin Hood hashing, which utilizes a lot of unsafe code to improve performance[4] performs significantly worse than the C++ implementation. The tests also suggest the Hopscotch algorithm performs better than Robin Hood.

[4] `https://github.com/mozilla/rust/blob/master/src/libstd/collections/hashmap.rs`

5.2.4 Evaluation of single threaded implementation tests

The most notable result of our tests, is how much better C++ performs pr. operation compared to Rust. As seen in section 5.2.3, C++ runs 5-10 times faster than the Rust implementation, and this may be caused by one of the main features of the Rust compiler: safe programming. This feature means that quick operations such as moving/incrementing pointers isn't possible and null pointers does not exist, as this may lead to segmentation faults or core dumped errors. This causes every lookup in the rawtable to be slower, and it causes overhead when linear probing, as we are forced to wrap our keys and values in an Option.

Based on our tests, programs written in safe Rust will not be the most effecient. Rust is still a work in progress, so improvements and optimizations of the languages features might improve before the official release. Rust still seems like a fast programming language, although it is still not near optimal. Rust is still very useful, and the promise of memory safe programs are for many developers so valueable that the performance cost is worth it, because of how much it helps when writing programs.

Our hopscotch implementation performs overall better than the Robin Hood implementation. In our tests, there were two scenarios where Robin Hood performed better than Hopscotch:

- In low load factors when performing: 60% lookups, 20% inserts and 20% removes,
- When inserting into the same bucket
- Robin hood only resized at load factor 90%, while Hopscotch often resizes when above 80%.

It can be concluded that Robin Hood performs better than Hopscotch, while dealing with smaller hashmaps. It performs better than Hopscotch if the hashmap performs many inserts and removes and less lookups, especially in lower load factors. But since hashmap is mostly used for lookups, Hopscotch seems like a superior hashmap algorithm compared to Robin Hood, and this is despite that the Robin Hood implementation uses unsafe code. THe greates advantage Robin Hood has over Hopscotch, is the fact that it is more memory efficient, as it only resizes when it reaches a load factor of 90%, whereas Hopscotch most likely will resize when above 80%. For a better comparison between the two implementations, it may be needed to make a new Hopscotch implementation with the use of unsafe code.

6. Conclusion

The Hopscotch algorithm performs very well when compared to the Robin Hood algorithm. Hopcotch perform lookups better on high load factors, and both insert and remove are generally faster. Robin Hood are better when when inserting into the same bucket, but this case is relatively rare, and as a result, Hopscotch comes out on top in most scenarios.

Hopscotch written in C++ performs generally better than Hopsotch written in safe Rust. This was to be expected, as safe code prevents some effective programming patterns. We found our implementation to be approximately 6 times slower for lookups, and removes, and about 5 times for inserts. Currently Rust provides no optimization levels when compiling code, and optimizations are not the focus of the language development at the moment, but in the future this will maybe change, which if happened, would increase the runtime of our implementation.

Rust safe code does not provide the neccesary features to implement a shared mutable memory Hopscotch hashmap. Because of this we were not able to implement a concurrenly accessed Hopscotch hashmap.

References

[1] H. T. Andersen, Project on hopscotch hashing, Bachelor project report, Department of Computer Science, University of Copenhagen (2014).
[2] P. Celis, Robin hood hashing, Technical report, Department of Computer Science, University of Waterloo (1986).
[3] N. S. Maurice Herlihy and M. Tzafrir, Hopscotch hashing, Technical report, Department of Computer Science, Brown University and Tel Aviv University (2008).

7. Appendix

7.1 Singlethreaded Hashmap Implementation code

7.1.1 Remove

Listing 3 : Remove

```
      pub fn remove<'a>(&'a mut self, key:K)->Option<V>{
             let new_hash = self.hasher.hash(&key);
             let mask = self.raw_table.capacity()-1u;
             let index_addr = (new_hash as uint) & mask;
             let hop_info =
                 self.raw_table.get_bucket(index_addr).hop_info.clone();
             let mut info = 1u32;
             //loops through the hop information
             for i in range(0u, VIRTUAL_BUCKET_CAPACITY){
                    if info & hop_info >= 1u32{
                    let addr = (index_addr+i) & mask;
                           let check_hash =
                               self.raw_table.get_bucket(addr).hash.clone();
                           //if we get a hit, remove the
                               reference and the key in the hash
                           //table
                           if new_hash == check_hash {
                 self.raw_table.get_bucket(index_addr).hop_info -=
                     info;
                 self.raw_table.delete_key(index_addr);
                 let ret_val =
                     self.raw_table.get_val(addr).clone();
                 self.decrement_size();
                 return Some(ret_val);
                           }
                    }
          info = info << 1;
             }
          None
   }
```

7.1.2 Lookup

Listing 4 : Lookup

```
   pub fn lookup<'a>(&'a self, key:K)->Option<&'a V>{
       let new_hash = self.hasher.hash(&key);
       let mask = self.raw_table.capacity()-1u;
             let index_addr: uint = (new_hash as uint) & mask;
       let mut hop_info = self.raw_table.get_i_bucket(index_addr).

             //Loops through the hop information.
```

```
                for i in range(0u, VIRTUAL_BUCKET_CAPACITY){
                        if (hop_info & 1) == 1{
                            let check_hash =
                                self.raw_table.get_i_bucket((index_addr
                                + i) &

                                //if we get a hit, return the found
                                    value.
                                if new_hash == check_hash {
                                        return
                                            Some(self.get_return_value((index_addr+i)
                                            & mask));
                                }
                        }
                        hop_info = hop_info >> 1;
                }
        None
    }
}
```

7.1.3 Insert

Listing 5 : Insert

```
    pub fn insert(&mut self, key:K, data:V)-> bool{
                if self.check_key(&key) {
                        return false;
                }
                let new_hash = self.hasher.hash(&key);
                let mask = self.raw_table.capacity()-1;
                let index_addr = mask & (new_hash as uint);
        let mut free_distance = 0u;
                let mut val = 1;
                //probes through the hash table, and checks if there
                    is an empty bucket
                //within ADD_RANGE.
        for i in range(0,ADD_RANGE){
            if !self.raw_table.get_key_option((index_addr+i) & mask)
                {
                break;
            }
            free_distance += 1;
        }
                //Checks if the empty bucket is within ADD_RANGE
                if free_distance < ADD_RANGE {
                        //if val = 0, we resize
                        while val != 0 {
                                //checks if the empty bucket is within
                                    H-range from the bucket.
                    //If it is, insert the bucket, if not,
                        we try and displace it
                    //closer.
                    if free_distance <
                        VIRTUAL_BUCKET_CAPACITY {
         self.raw_table.get_bucket(index_addr).hop_info


                            self.raw_table.get_bucket((index_addr
                                + free_distance) &
                                           mask).hash =
                                               new_hash;
                            self.raw_table.insert_key((index_addr
                                + free_distance) &
                                                mask,
                                                    key.clone());
                            self.raw_table.insert_val((index_addr
                                + free_distance) &
                                               mask,
                                                   data.clone());
                            self.size += 1;
                            return true;

                    }
                    //Tries to displace the empty bucket
                        closer to the bucket
                    self.find_closer_bucket(&mut
                        free_distance, index_addr,



                }
            }
        self.resize();
            self.insert(key.clone(), data.clone())
    }
}
```

7.1.4 *Find Closer Bucket*

Listing 6 : Find Closer bucket

```
       fn find_closer_bucket(&mut self, free_distance:&mut uint,
           index_addr:uint,
                        val:&mut int, mask:uint){
       let free_bucket_index = (index_addr + *free_distance) & mask;
       let mut move_bucket_index = ((index_addr + *free_distance) -
                       (VIRTUAL_BUCKET_CAPACITY-1)) & mask;
              let mut free_dist = VIRTUAL_BUCKET_CAPACITY-1u;
              while 0 < free_dist {
                let mut move_bucket_hop_info =
                    self.raw_table.get_bucket(
                                    move_bucket_index).hop_info.clone();
                    let mut move_free_distance = -1;
                    let mut mask2 = 1u32;
                    //Checks if there is a bucket we can possibly
                        displace with
                    for i in range(0, free_dist){
                            if mask2 & move_bucket_hop_info == 1{
                                    move_free_distance = i;
                                    break;
                            }
            mask2 = mask2 << 1;
                    }
                    //if we get a hit, displace the buckets
                if move_free_distance != -1 {
            let new_free_bucket_index = (move_bucket_index +
                                          move_free_distance) &
                                              mask;

            self.raw_table.get_bucket(move_bucket_index).hop_info
                                                        |=
                                                            (1<<free_dist);
            let new_free_bucket_val = self.raw_table.get_val(
                                                      new_free_bucket_index).clone();
            self.raw_table.insert_val(free_bucket_index,new_free_bucket_val);
            let new_free_bucket_key = self.raw_table.get_key(
                                                      new_free_bucket_index).clone();
            self.raw_table.insert_key(free_bucket_index,new_free_bucket_key);
            self.raw_table.get_bucket(move_bucket_index).hop_info
                &=
                                                        !(1u32<<move_free_distance);
            let new_free_bucket = self.raw_table.get_bucket(
                                                    new_free_bucket_index).clone();
            self.raw_table.get_bucket(free_bucket_index).hop_info
                =
                                                    new_free_bucket.hop_info;
            self.raw_table.get_bucket(free_bucket_index).hash =
                                                    new_free_bucket.hash;
                        *free_distance -= free_dist;
                        return;
                }
        move_bucket_index = (move_bucket_index + 1) & mask;
        free_dist -= 1;
        }
        *val = 0;
    }
```

7.1.5 Resize

Listing 7 : Resize

```
    pub fn resize(&mut self){
        let new_capacity = self.raw_table.capacity() << 1;
        let old_table = replace(&mut self.raw_table,
                                    raw_table::RawTable::new(new_capacity));
        let old_capacity = old_table.capacity();
        let mut info = 0;
        for i in range(0,old_capacity){
            if old_table.get_key_option(i){
                self.insert(old_table.get_key(i).clone(),
                                        old_table.get_val(i).clone());
            }
        }
    }
```

7.1.6 Rawtable

Listing 8 : Rawtable

```
#![allow(dead_code)]
use std::clone::Clone;
use std::option::{Option,None,Some};
use std::mem::replace;
use std::cmp::max;
use std::num;
use std::vec::Vec;
//Our H-range
pub static VIRTUAL_BUCKET_CAPACITY: uint = 32;
//Sets our default size when creating a hash table
static INITIAL_LOG2_CAP: uint = 4;
pub static INITIAL_CAPACITY: uint = 1 << INITIAL_LOG2_CAP;

//Bucket structure, containt hop information and hash
#[deriving(Show,Clone)]
pub struct Bucket{
    pub hop_info: u32,
    pub hash:    u64
}

//Rawtable, contains the 3 arrays: Array of buckets, keys and
    values.
#[deriving(Show,Clone)]
pub struct RawTable<K,V>{
    capacity: uint,
    buckets: Vec<Bucket>,
    keys:    Vec<Option<K>>,
    vals:    Vec<Option<V>>
}

//Rawtable functions
```

```
impl<K: Clone, V: Clone> RawTable<K,V>{
        //Initiates a new hash table.
    pub fn new(cap: uint) -> RawTable<K,V>{
        let capacity =
            num::next_power_of_two(max(INITIAL_CAPACITY,cap));
        let bucket_vec =
            Vec::from_elem(capacity,Bucket{hop_info:0,hash:0});
        let keys_vec = Vec::from_elem(capacity,None);
        let vals_vec = Vec::from_elem(capacity,None);
        let ret = RawTable{
                    capacity: capacity,
                    buckets: bucket_vec,
                    keys: keys_vec,
                    vals: vals_vec
                };
        ret
    }
        //returns an immutable bucket at the given index
    pub fn get_i_bucket<'a>(&'a self,idx:uint)->&'a Bucket{
        self.buckets.get(idx)
    }
        //returns a mutable bucket at the given index
    pub fn get_bucket<'a>(&'a mut self,idx:uint)->&'a mut Bucket{
        self.buckets.get_mut(idx)
    }
        //returns an immutable key at the given index
    pub fn get_key<'a>(&'a self,idx:uint)->&'a K{
        match *self.keys.get(idx) {
            Some(ref k) => k,
            None => fail!("Getting the key at: {} fails",idx)
        }
    }
        //returns an immutable value at the given index
    pub fn get_val<'a>(&'a self,idx:uint)->&'a V{
        match *self.vals.get(idx) {
            Some(ref v) => v,
            None => fail!("Getting the value at: {} fails",idx)
        }
    }
        //return a mutable value at the given index
    pub fn get_mut_val<'a>(&'a mut self,idx:uint)->&'a mut
        Option<V>{
        self.vals.get_mut(idx)
    }
        //returns a key wrapped in an option.
    pub fn get_key_option(&self,idx:uint)->bool{
        match *self.keys.get(idx) {
            Some(_) => true,
            None => false
        }
    }
        //inserts a key at the given index
    pub fn insert_key(&mut self,idx:uint,elem:K){
        *self.keys.get_mut(idx) = Some(elem);
        //replace(self.keys.get_mut(idx),elem);
    }
        //deletes a key at the given index
    pub fn delete_key(&mut self,idx:uint){
        *self.keys.get_mut(idx) = None;
    }
        //inserts a value at the given index
    pub fn insert_val(&mut self,idx:uint,elem:V){
        replace(self.vals.get_mut(idx),Some(elem));
    }
        //returns the max capacity of the hash table.
    pub fn capacity(&self)->uint{
        self.capacity
    }
}
```

7.2 Multithreaded Hashmap Implementation Code

7.2.1 Remove

Listing 9 : Remove

```
        pub fn remove<'a>(&'a mut self, key:K)->Option<V>{
let new_hash = self.hasher.hash(&key);
let mask = self.raw_table.capacity()-1u;
let index_addr = (new_hash as uint) & mask;
        let lock_virtual_bucket =
            self.raw_table.get_bucket_lock(index_addr);
        let bucket = lock_virtual_bucket.write();
let hop_info = bucket.hop_info.clone();
let mut info = 1u32;

for i in range(0u, VIRTUAL_BUCKET_CAPACITY){
if info & hop_info >= 1u32{
let addr = (index_addr+i) & mask;
                let lock_remove_bucket =
                    self.raw_table.get_bucket_lock(addr);
                let remove_bucket = lock_remove_bucket.read();
let check_hash = remove_bucket.hash.clone();
                drop(remove_bucket);
if new_hash == check_hash {
                    bucket.hop_info -= info;
                    self.raw_table.delete_key(index_addr);
                    let ret_val =
                        self.raw_table.get_val(addr).clone();
                    self.decrement_size();
                    return Some(ret_val);
}
```

```
                //Unlocks the lock
}
            info = info << 1;
}
None
        // Because drop is called on all variables the function
            returns,
        // the lock we took on the virual bucket is automatically
            unlocked
    }
```

7.2.2 Lookup

Listing 10 : Lookup

```
  pub fn lookup<'a>(&'a self, key:K)->Option<&'a V>{
        let new_hash = self.hasher.hash(&key);
        let mask = self.raw_table.capacity()-1u;
let index_addr: uint = (new_hash as uint) & mask;
        let virtual_bucket_lock =
            self.raw_table.get_bucket_lock(index_addr);
        let virtual_bucket = virtual_bucket_lock.read();
        let mut hop_info = virtual_bucket.hop_info.clone();

for i in range(0u, VIRTUAL_BUCKET_CAPACITY){
if (hop_info & 1) == 1{
                let check_bucket_lock =
                    self.raw_table.get_bucket_lock(
                                                    (index_addr+i) &
                                                        mask);
                let check_bucket = check_bucket_lock.read();
let check_hash = check_bucket.hash.clone();
if new_hash == check_hash {
return self.get_val((index_addr+i) & mask);
}
                drop(check_bucket);
}
hop_info = hop_info >> 1;
}
        None
    }
```

7.2.3 Insert

Listing 11 : Insert

```
  pub fn insert(&mut self, key:K, data:V)-> bool{
if self.check_key(&key) {
```

```
return false;
}
let new_hash = self.hasher.hash(&key);
let mask = self.raw_table.capacity()-1;
let index_addr = mask & (new_hash as uint);
       let mut free_distance = 0u;
let mut val = 1;
       for i in range(0,ADD_RANGE){
           if !self.raw_table.get_key_option((index_addr+i) & mask)
               {
               break;
           }
           free_distance += 1;
       }
if free_distance < ADD_RANGE {
       while val != 0 {
              if free_distance < VIRTUAL_BUCKET_CAPACITY {
                  let virtual_bucket_lock =
                      self.raw_table.get_bucket_lock(index_addr);
                  let virtual_bucket = virtual_bucket_lock.write();
                  virtual_bucket.hop_info |= 1<<free_distance;
                  drop(virtual_bucket);
                  let insert_bucket_lock =
                      self.raw_table.get_bucket_lock((index_addr
                                          + free_distance) & mask);
                  let insert_bucket = insert_bucket_lock.write();
                  insert_bucket.hash = new_hash;
                  drop(insert_bucket);
                  self.raw_table.insert_key((index_addr +
                      free_distance) &
                                            mask, key.clone());
                  self.raw_table.insert_val((index_addr +
                      free_distance) &
                                           mask, data.clone());
                  self.size += 1;
                  return true;
                  }
                self.find_closer_bucket(&mut free_distance,
                    index_addr, &mut val, mask);
             }
       }
self.resize();
self.insert(key.clone(), data.clone())
}
```

7.2.4 Find Closer Bucket

Listing 12 : Find Closer Bucket

```
pub fn find_closer_bucket(&mut self, free_distance:&mut uint,
    index_addr:uint,
val:&mut int, mask:uint){
       let free_bucket_index = (index_addr + *free_distance) & mask;
       let mut move_bucket_index = ((index_addr + *free_distance) -
           (VIRTUAL_BUCKET_CAPACITY-1)) & mask;
let mut free_dist = VIRTUAL_BUCKET_CAPACITY-1u;
while 0 < free_dist {
           let move_bucket_lock =
               self.raw_table.get_bucket_lock(move_bucket_index);
           let mut move_bucket = move_bucket_lock.write();
let mut move_bucket_hop_info = move_bucket.hop_info.clone();
let mut move_free_distance = -1;
let mut mask2 = 1u32;
for i in range(0, free_dist){
if mask2 & move_bucket_hop_info == 1{
move_free_distance = i;
break;
}
               mask2 = mask2 << 1;
}
if move_free_distance != -1 {
               let new_free_bucket_index = (move_bucket_index +
                   move_free_distance) & mask;

               move_bucket.hop_info |= (1<<free_dist);
               // The Rust way would be if this method only required
                   an
               // immutable self, because the locks can return the
                   guarded
               // data as mutable. If this way of doing things
                   should be
               // complete, keys, and values should be locked as
                   well,
               // but that would be ridiculous!
               let new_free_bucket_val =
                   self.raw_table.get_val(new_free_bucket_index).clone();
               self.raw_table.insert_val(free_bucket_index,new_free_bucket_val);
               let new_free_bucket_key =
                   self.raw_table.get_key(new_free_bucket_index).clone();
               self.raw_table.insert_key(free_bucket_index,new_free_bucket_key);

               move_bucket.hop_info &= !(1u32<<move_free_distance);

               let new_free_bucket =
                   self.raw_table.get_bucket_lock(new_free_bucket_index);
               let free_bucket_lock =
                   self.raw_table.get_bucket_lock(free_bucket_index);
               let free_bucket = free_bucket_lock.write();
               free_bucket.hop_info = new_free_bucket.hop_info;
               free_bucket.hash = new_free_bucket.hash;
*free_distance -= free_dist;
return;
               // Again, buckets automatically unlock when returning
}
           drop(move_bucket);
           move_bucket_index = (move_bucket_index + 1) & mask;
           free_dist -= 1;
}
*val = 0;
    }
```

Listing 13 : Rawtable

```
#![allow(dead_code)]
extern crate sync;
use sync::RWLock;
use std::clone::Clone;
use std::option::{Option,None,Some};
use std::mem::replace;
use std::cmp::max;
use std::num;
use std::vec::Vec;

pub static VIRTUAL_BUCKET_CAPACITY: uint = 32;
static INITIAL_LOG2_CAP: uint = 4;
pub static INITIAL_CAPACITY: uint = 1 << INITIAL_LOG2_CAP; //2^5

//Is not boxed, like structures are in Rust
pub struct Bucket{
    pub hop_info: u32,
    pub hash: u64
}

pub struct RawTable<K,V>{
    // Available elements
    capacity: uint,
    // Occupied elements
    //buckets: Vec<Bucket>, //Contains hop info and hash
    buckets: Vec<RWLock<Bucket>>,
    keys: Vec<Option<K>>,
    vals: Vec<Option<V>>
}

impl<K: Clone, V: Clone> RawTable<K,V>{
    pub fn new(cap: uint) -> RawTable<K,V>{
        let capacity =
            num::next_power_of_two(max(INITIAL_CAPACITY,cap));
        let bucket_vec =
            Vec::from_fn(capacity,|_|{RWLock::new(Bucket{hop_info:0,hash:0})});
        let keys_vec = Vec::from_elem(capacity,None);
        let vals_vec = Vec::from_elem(capacity,None);
        let ret = RawTable{
                capacity: capacity,
                buckets: bucket_vec,
                keys: keys_vec,
                vals: vals_vec
            };
        ret
    }
    pub fn get_bucket_lock<'a>(&'a self,idx:uint)->RWLock<Bucket>{
        self.buckets.get(idx)
    }
    pub fn get_key<'a>(&'a self,idx:uint)->&'a K{
        match *self.keys.get(idx) {
            Some(ref k) => k,
            None => fail!("We suck at rawtable keys:{}",idx)
        }
    }
    pub fn get_val<'a>(&'a self,idx:uint)->&'a V{
        match *self.vals.get(idx) {
            Some(ref v) => v,
            None => fail!("We suck at rawtable values:{}",idx)
        }
    }
    pub fn get_mut_val<'a>(&'a mut self,idx:uint)->&'a mut
        Option<V>{
        self.vals.get_mut(idx)
    }
    pub fn get_key_option(&self,idx:uint)->bool{
        match *self.keys.get(idx) {
            Some(_) => true,
            None => false
        }
    }
    pub fn insert_key(&mut self,idx:uint,elem:K){
        *self.keys.get_mut(idx) = Some(elem);
        //replace(self.keys.get_mut(idx),elem);
    }
    pub fn delete_key(&mut self,idx:uint){
        *self.keys.get_mut(idx) = None;
    }
    pub fn insert_val(&mut self,idx:uint,elem:V){
        replace(self.vals.get_mut(idx),Some(elem));
    }

    pub fn capacity(&self)->uint{
        self.capacity
    }
}
```