UMEÅ UNIVERSITY **May 7, 2020**
Department of Computing Science
Assignment 4 report

5DV149 - Assignment 4, v1.0

Data Structures and Algorithms (C) Spring 2020, 7.5 Credits

Comparison of table implementations

Name Elias Olofsson

user@cs.umu.se `tfy17eon@cs.umu.se`

Note to the graders:

This version of the report should be graded. The reason for the failed submission test is due to the additional file `hashtable.c`, my hash table implementation. See the report down below for more details.

Teacher
Anna Jonsson

Graders
Anna Jonsson, Lena Kallin Westin, Hannah Devinney

Contents

1 Introduction

In this lab we take a look at the abstract datatype *table* and make comparisons between four different implementations of the datatype in the programming language C. The first implementation is given from a supplied code-base while the remaining three have programmed by myself, using underlying implemented data-types such as *directed list* and *array*, which were given from the same aforementioned code-base. All the implementations of *table* use the same identical interface and can as such be easily tested and compared using the same test program. The aim of this lab is to help to understand the usage of abstract data-types, compare and analyse the benefits and disadvantages of different implementations of a certain abstract datatype, and to practice general programming skills, including correctly implementing dynamic memory allocation.

2 Description of abstract interface

Before going into the language specific implementations of the abstract datatype *table*, we need to describe its abstract and general interface. The interface is given below.

```
abstract datatype Table(arg, val)
  Empty    ()                                  → Table(arg, val)
  Insert   (x:arg, y:val, t:Table(arg, val))   → Table(arg, val)
  Isempty  (t:Table(arg, val))                 → Bool
  Lookup   (x:arg, t:Table(arg, val))          → (Bool, val)
  Remove   (x:arg, t:Table(arg, val))          → Table(arg, val)
```

The first function in the interface of *table* is the `Empty` function which creates a new and empty table. It takes no arguments and returns the newly created table, which itself is comprised of arguments of arbitrary type `arg` and values of arbitrary type `val`. Each argument in the table is unique and maps to a singular corresponding value.

The second function `Insert` takes an argument `x` and a value `y` and inserts both of them into the table `t`. At insertion, the argument gets mapped to the corresponding value and thus, the two become paired with each other within the table. After completed insertion, the function returns the updated table, now with the new key and value pair in it.

The third function `Isempty` takes a table `t` as an argument and checks whether the table in question is empty or not. If the table has existing key

and value pairs in it, the function returns false, otherwise true.

The fourth function `Lookup` takes an argument `x` and searches for that key within the table `t`. If a matching key is found, the function returns true while also returning the corresponding element value. If no matching key is found in the table, the function returns false.

The fifth and last function of the abstract interface is `Remove` which takes an argument `x` and searches for that key in the table `t`. If a matching key is found, both, the key and the corresponding value entry are removed from the table. The function returns the updated table, now having one key/value pair less in it. If no matching key is found in the table, the function does nothing.

3 Implementation details

The three different implementations I have constructed within this lab are all quite similar and contrasting at the same time. While the first of them uses a directed list as the underlying data structure, the same as the given example of implementation from the code-base, the other two utilize the array datatype as the fundamental structure.

The first implementation, `mtftable.c`, is constructed using almost the same exact code as the given implementation `table.c`. Thus, both of them are utilizing a directed list in the background, where each key/value pair is inserted as a `struct` first in the list at the time of insertion. My implementation only differs in two additional code lines (line 123-124, `mtftable.c`) within the function `Lookup`. The effective difference for my implementation is, when a matching key is found using `Lookup`, not only the corresponding value is returned, the key/value pair in question is also moved to the front of the list.

It might seem like a very minor change to implement the table as a *move-to-front* table but for certain conditions the small behaviour difference yields some performance improvements. Since `Lookup` searches linearly through the list for a matching key, the complexity becomes dependent on the length of the list. If the most sought-after keys come from a skewed set of the total available and existing keys, then you can effectively shorten the search time by constantly moving the recently referenced keys and values to the front of the list. Although this is great for these certain skewed conditions, there is no general performance improvement in random lookup speed when all available keys are equally likely to be chosen.

My second implementation of the table datatype, `arraytable.c`, uses the underlying array structure as implemented in `array_1d.c`, given in the codebase. Thus, the table implementation is going to inherit some of the characteristics associated with the datatype *array*, different from the previous two implementations, which instead had properties reminiscent of the datatype *directed list*. Perhaps most notably, the table implementation as an array is going to be static in table size due to the fact that arrays are static in nature. In contrast, a directed list is dynamic and changes size as new elements are inserted into the structure, thus making the table implementations dynamic in size as well.

Other noteworthy implementation details of `arraytable.c` are the duplicate-handling and the element removal process, which differ from the previous two table implementations as directed lists. In `table.c` and `mtftable.c` no checks for duplicate keys are preformed at insertion but since new elements are placed first in the list, the most recent element value is always going to be correctly referenced by `Lookup` which linearly searches the list from the front and stops when a match is found. It is only at removal that potential duplicate keys get removed from the list. Conversely, in `arraytable.c` the check for a key duplicate is performed at each key/value pair insertion. For a new and empty table, the first element is placed in the first position of the array. Each following insertion is going to successively check each array position linearly for a key match until an empty position is found. Unless a key duplicate is found, for which then the corresponding element value is updated, each new key/value pair is added to the end of the populated part of the array. Thus, the table is always going to contain unique keys, and also have the first part of the array be consecutively populated.

However, a problem arises at the removal of table entries since no "holes" can be left in the array without creating problems for later. If `Lookup` among other function searches through the array linearly until reaching the first empty array position, then elements past this position can be lost if this happened to be the hole left from a previously removed table entry. My solution to this issue is to keep a counter of the number of elements in the array which allows me to efficiently index and move the last element in the array to fill the hole left open after the removal process. As a result, the chain of array elements is never broken and linear searches continue to work.

Perhaps a more exotic table implementation is my `hashtable.c`, which partly came to be due to a misunderstanding of the specification of this lab but also since I was intrigued by the concept of hash-functions and wanted to try to implement one. As the name suggests, it is a implementation of a hash table and is somewhat similar to the array implementation in `arraytable.c` since it uses the same underlying resources `array_1d.c`

found in the code-base. Likewise, it is static in table size and also checks for duplicate keys at insertion, rather than at removal. Nevertheless, this implementation is at the same time also way different due to the usage of a hash-function to efficiently process a key into an index in the array.

My hash-function is a simple one, built around the given pseudo code from the lecture notes of this course, and credited to Brian Kernighan and Dennis Ritchie, appearing in the book "The C Programming Language"[1]. It accepts a key either given as an `int` or as a `char` array of at least four characters and returns a valid index within the array. String literals with at least three characters are also fine as key input due to their null termination. When called, the hash-function typecasts the input `void*` to an `int32_t*`, thus only dereferencing the four first bytes of memory pointed to, and converts the integer to a string of numbers using the standard library function `sprintf`. The string is then scrambled and hashed using the BKDR hashing algorithm and shrunk down to an integer corresponding to a valid index in the array using the modulus function of the table size.

The introduction of the hash-function makes the behaviour of this table implementation rather different than the previous, more straight-forward, array implementation. While `arraytable.c` searches for a matching key or an empty spot in the array by always starting at the beginning of the array, `hashtable.c` calls the hash-function at the beginning of every search and is in return given a starting index in the array, determined by the key in question. The search is then commenced from the starting index and continued through the array until an empty position or matching key is found. Also adding to the differences, while the normal array table implementation searches the array linearly, I have implemented the hash table to use a quadratic search strategy. Thus, I also had to allow the search to "wrap around" the array once the quadratically incremented indices reach out of bounds of the table size.

However, we still come across the same problem here as we did in the regular array implementation. Naively removing table entries from the hash table can create later problems where array elements past this "hole" may be lost by the searching algorithm if no counter measure is taken. The previous solution to this problem, by always reshuffling the array entries to continuously fill one end of the array, does unfortunately not work here. Since the hash-function tries to uniformly spread out the table entries over the entire array, we will naturally have an abundance of "holes" in any partially filled array, and thus have to modify the searching behaviour. The solution is to add a "removed element" marker, signifying that the current position in the array previously has been occupied but currently is empty and available. Practically, I achieved this by letting the "removed element" marker

be defined as a `table_entry` struct left in its position in the array, but its key and value struct members both set to `NULL`. So once the `Remove` function has gotten this modification, the `Insert` and `Lookup` functions can be modified accordingly to either skip past or stop the search at a "removed element" marker and henceforth, no entries should be lost in the table due to incorrect removal.

4 Analysis of complexity

4.1 Asymptotic analysis

In the following is a simplified analysis of the complexity of the functions in the interface of the four table implementations, analogous to the functions in the abstract interface seen above in Sec.(2).

4.1.1 `Empty`

This function is identical for the first two list-based implementations and can be estimated to have a relative complexity of order $\mathcal{O}(1)$ due to only performing operations and allocations to create the table structure itself, which is independent of the table size and the number of table entries. The same is true for both the regular array table implementation and my hash table implementation.

4.1.2 `Isempty`

The `Isempty` function is also identical for the first two list-based table implementations, and since this function is only comprising of a single function call to the list interface to check whether the list is empty, this has a relative complexity of order $\mathcal{O}(1)$.

For the other two array-based implementations, this function, in a fully general sense, would have to be dependent of the size of the table since the only way to truly check if an array is empty, is to manually inspect each position. However, I can circumvent this limitation of the array datatype by implementing an element counter that keeps track of the current number of table entries and updates it after each insertion and removal. Thus, it becomes trivial to check whether the table is empty, and the relative complexity for this function for both array implementations becomes of order $\mathcal{O}(1)$.

4.1.3 `Insert`

The `Insert` function is yet another function that is identical for both of the list-based implementations. Accordingly, both functions will also be of $\mathcal{O}(1)$, since it always places new elements first in the list and thus only requires the same sets of operations and allocations, independent on the table size and number of table entries.

For the regular array implementation, the complexity of the `Insert` function becomes a bit different. Since a linear search for potential duplicates is performed before the actual element insertion itself, this function gets a relative complexity of the order $\mathcal{O}(n)$, where n is the number of table entries in the table.

An asymptotic analysis of the complexity of `Insert` for the hash table implementation becomes a bit more tricky than the preceding two estimates. One thing to note is, the hash function itself can, to good approximation, be estimated to be of order $\mathcal{O}(1)$. It should technically be weakly dependent on the key itself, depending on how many characters long the converted string of numbers ends up being, adding extra loops to the hashing process. Since the highest number an `int32_t` can take is 10 numerals long in decimal form, the for-loop is upper limited to 10 total loops, making it in the context essentially constant for our usage in the table implementation.

For the actual searching part of the `Insert` function of the hash table, the relative complexity of the function should be dependent, not necessarily on the number of table entries, but on the degree to which the table is currently filled. Since we have chosen to implement a quadratic search strategy, there is to my understanding no satisfactory theoretical analysis of the complexity and trying to pursue this is past the expectancy of this course. However, in practice we can expect massive improvements in insertion speed compared to the normal array implementation. Since for a table that is relatively empty, we normally only have to check a fraction of the table entries before placing a new key/value pair into the array.

4.1.4 `Lookup`

For this function, the code is for once not exactly identical to the first two list-based implementations, but the two additional lines of code that make one of them behave as a *move-to-front* table does not alter the relative complexity in any meaningful way. Both of the functions linearly searches through the list for the requested key, thus making the function have a relative complexity of order $\mathcal{O}(n)$, where n is the number of table entries in

the table.

For the regular array-based implementation, the story is very similar. The function searches through the array linearly until the key is found or not, making the relative complexity of order $\mathcal{O}(n)$, where n is again the number of table entries.

For the hash table implementation, this function is also a bit more tricky to estimate the complexity, but we can reason in a very similar way as we did in the preceding subsection regarding the function `Insert`. We can approximate the hash-function as being of constant order, using the same arguments as previously stated. We can also expect the `Lookup` function to be dependent on the degree to which the table is currently filled, but no complete analytical analysis of complexity can be found, see preceding subsection. Nonetheless, we should for a normal case of a reasonably filled table expect the effective random lookup-speed to be largely improved compared to all the other table implementations due to only having to check a fraction of the table entries.

4.1.5 `Remove`

For this function, we once more have identical code for the first two list-based table implementations. Since both of them linearly search through the list for the key to be removed, the relative complexity becomes of order $\mathcal{O}(n)$, where n is the number of table entries.

For the regular array-based implementation, the same approach is true. The function linearly searches through the array for the key in question, making the relative complexity be of order $\mathcal{O}(n)$, where n is the number of entries in the table.

For the hash table implementation, we hold a similar argument for the analysis of complexity as for the previous two functions, `Insert` and `Lookup`, seen in the two preceding subsections. The complete analytical complexity cannot fully be determined, but the effective behaviour is expected to have improved considerably compared to all the other table implementations, given that the table is filled to a reasonably degree. Since only a fraction of the table entries has to be checked, the random remove speed in a table based on the hash table implementation should typically result in large improvements given normal conditions.

4.2 Experimental analysis

In this lab, we got supplied an complimentary testing program `tabletest-1.9.c` to verify that we have achieved correct functionality of the different table implementations. Also included in the test program was a suite of speed tests to experimentally quantify the complexity and compare the different functions in the table interface. The five speed tests are of different character and aimed to evaluate the performance of three select functions in the table interface, namely `Insert`, `Lookup` and `Remove`. At each execution of the test suite, the program takes an input parameter, specifying the number of table entries that should be involved in the tests, and subsequently performs the five speed tests in rapid succession. The results of the timed executions are measured in milliseconds and printed in the terminal, but can also be saved to a file for more convenient analysis later.

If we let n be the number of table entries used in the test suite, the five speed tests performed are in order; a test of n insertions into the table, a test of n random removals from the table, a test of n unsuccessful random lookups, a test of n successful random lookups and finally, a skewed test of n successful random lookups. Each table instance is completely destroyed and randomly regenerated between each test, and the lookup tests all are performed on a table populated with n table entries.

To investigate the complexity, I let the test program be compiled with each of the four table implementations and subsequently ran at table entry sample sizes of 1000, 2000 and up to 20000 table entries, in increments of 2000 entries. For each sample size, the test suite was ran five times for each table implementation. The averaged result for each of the different speed tests, for each of the different implementations, can be seen in Appendix A, Fig.(1). Here, the total mean run-time is plotted as a function of the number of table entries involved in each of the speed tests. Thus, this is a representation of the absolute complexity for the three different table interface functions, for the given sets of operating conditions.

Since in the preceding section, Sec.(4.1), we were discussing the complexity of the table interface functions in relative terms, as in relative to the underlying data structure interface, we need to further analyze the experimental results in order to compare it with the simplified asymptotic analysis of the previous section. The underlying structures we have utilized in these table implementations have been from the abstract datatypes *directed list* and *array*. Thus, we need to factor out the complexity associated with the functions in their interfaces, corresponding to the usage of them within the table implementations.

Since all of the functions in the interface of both underlying datatypes are of order $\mathcal{O}(1)$, the corresponding math becomes easy. To make the experimental time complexity relative, we only need to factor out the linearity by which the total run-time increases as we let the code involve more table entries. Hence, by dividing the total run-time seen in Appendix A, Fig.(1), by the sample size of table entries used in each test, we can make an estimate of the relative behaviour for the time complexity of the table interface functions. The results can be seen in Appendix A, Fig.(2), where run-time per table entry is plotted as a function of the total number of table entries involved in the test.

Taking a closer look at the set of figures in Appendix A, Fig.(2), we can verify the analytical reasoning we held in the previous section, Sec.(4.1). Firstly, one can note that for the insertion test in Fig.(2a) we see essentially constant behaviour for all table implementations, except for the regular array implementation, where the behaviour is linear. This was correctly predicted by our analysis in the preceding section.

In the following figure, Fig.(2b), we can note linear behaviour for the `Remove` functions of all normal table implementations with the only outlier being the hash table implementation, which for this table size and degree to which it is filled, is essentially constant. The linearity of the removal functions for the first three table implementations was correctly predicted, but it is interesting to note the differences in slope between the list-based and array-based implementations.

Following this, we proceed to the three figures, Fig.(2c-2e) where the relative time complexity behaviour of the `Lookup` functions can be seen. The first two list-based implementations have exactly the same linear behaviour for the two tests where all table entries involved are equally likely to be referenced, but for the skewed lookup test, where only a subset of the total available keys are searched, the *move-to-front* table has a clear performance improvement. This behaviour was expected and has now been verified experimentally.

For the regular array-based implementation, the random lookup speed tests in Fig.(2c-2e) displays a linear behaviour, which was also predicted correctly in the simplified asymptotic complexity analysis in the previous section. Worth noting is that the regular array-based implementation performed at least on par, but mostly better, than both of the list-based implementations in the lookup tests. As such, if lookup speed is essential for the application of the table implementation, then the regular array implementation would be advantageous. On the other hand, if insertion speed is most important in the final table application, then one of the list based implementations would be favorable, but this prioritization would come at the expense of random remove and lookup speed.

However, all of the normal table implementations were completely crushed by the extreme performance of the hash table implementation. This should not be too much of a surprise, given the fundamentally different approach and more advanced theory that is hidden beneath the surface of this implementation. Considering the methodology and the scale at which this experimental analysis was conducted, it is not really a fair comparison between the hash table and the rest. Firstly, it is orders of magnitudes faster in execution speed, finishing faster than the timings of this speed test can register, and secondly, they should not really be compared by the same metric. The first three implementations have a performance dependency on the number of current table entries n, while, in contrast, the hash table is dependent on the degree to which the table is currently filled. This is normally described by a quotient between current table population and total table capacity, named the load factor.

In all of these test performed here, the load factor has never risen above 0.25, and since the performance generally drops down as the load factor increases, the results displayed here in this report should probably be viewed as a best case scenario for the hash table implementation. It would be interesting to do a deeper dive into how the complexity changes as the load factor approaches unity but since this is not part of the specifications for this lab and time has been a factor, no such investigation has been done. Still, it is a bit mesmerizing to see the power of the hash table implementation, and watch it run circles around the other implementations during normal testing conditions.

5 Discussion

In this lab we have investigated four different implementations of the abstract datatype *table*. Two of the implementations have been built on the underlying datatype *directed list*, while the other two have used the *array* datatype as the underlying structure. Even though all of the implementations have identical behaviour on the surface, as a result of the implementation details and the inheritance of properties from the underlying data structure, they all have differing characteristics and varying relative complexity of the functions in the table interface. Depending on the choice of implementation and the set of operating conditions, the performance of the interface functions differ.

As reasoned in the preceding section, Sec.(4), depending on whether either insertion speed, random lookup speed or skewed lookup speed is important for the final table application, different implementations should be selected

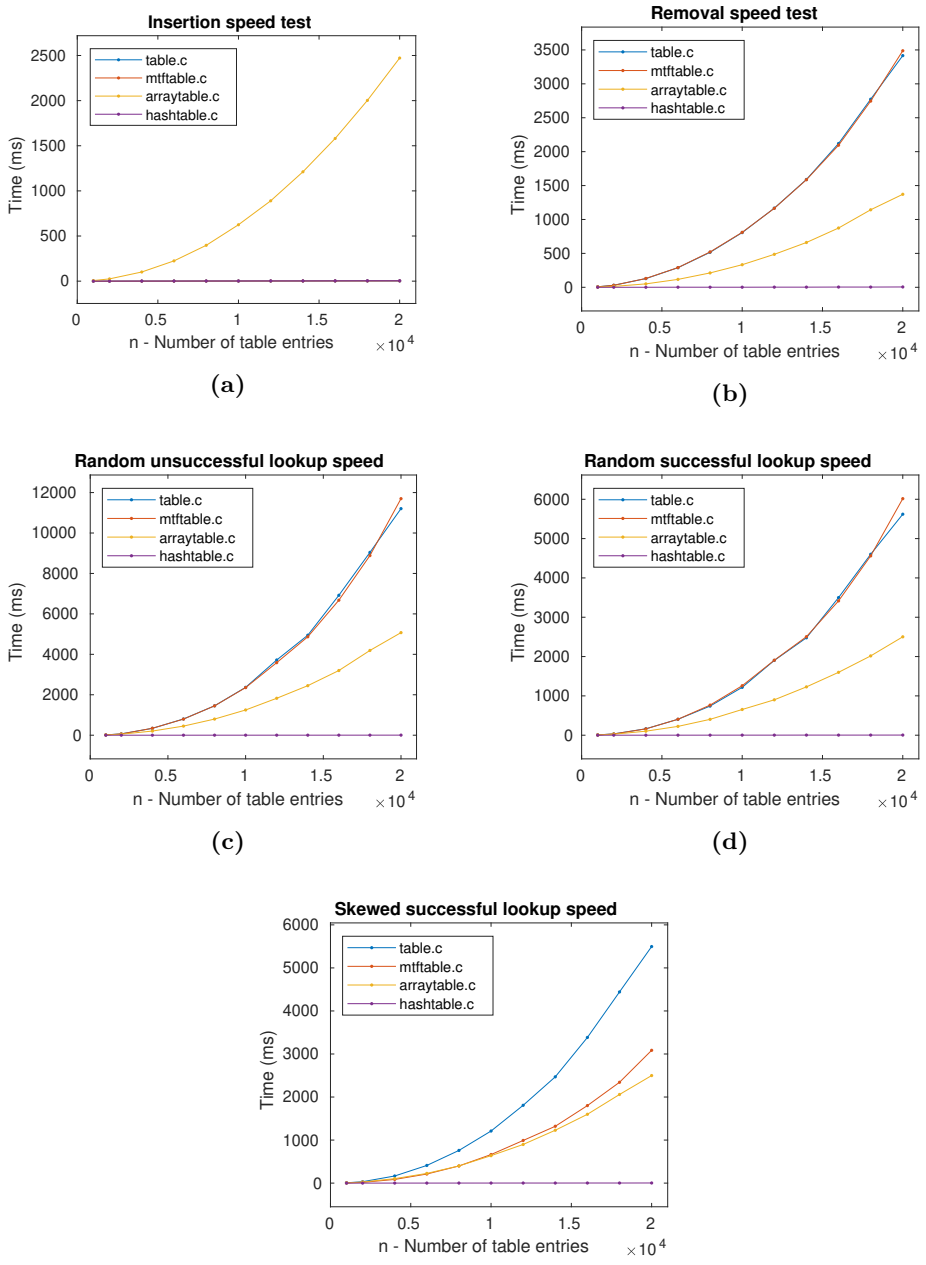
for each corresponding prioritized use case. While list-based implementations have great insertion speed, they lack the lookup speed of the regular array-based implementation. For certain operating conditions, where the most frequented keys are skewed relative to the total available keys, the list-based implementation can still make up for some of it in the lookup speed by utilizing a *move-to-front* strategy, then performing on par with the array-based implementation. If maximal possible performance is required for the final application, one should perhaps investigate in the usage of a hash table implementation. However, this will come with the trade off of largely increased code complexity, which in turn increases the development time and cost of deployment.

All in all, I personally found this to be a really interesting lab to complete. I have learnt a lot about abstract and implemented data structures, and I have also gained a great amount of general purpose programming skills. I have improved my ability to program in C while also having discovered a lot of other external, but incredibly useful, tools such as `make`, `bash` and `git`. I may have wasted some time going into how to implement a hash table, initially thinking it was part of the lab specification, but in retrospect I still see that as a worthwhile investment into my general programming skills. It has been a pleasure, and if I were to redo the lab with the knowledge that it was not part of the specification, I probably would still take the detour anyway.

References

[1] A. Partow, "General Purpose Hash Function Algorithms", Partow.net. [Online]. Available: http://www.partow.net/programming/hashfunctions/.

A Appendix



(a) (b) (c) (d) (e)

Figure 1 – Averaged result of the speed test suite, for all table implementations. Sample size tested, in number of table entries n, was 1000, 2000 and up to 20000 in 2000 increments. For each sample size, the average run-time of five test program executions was calculated.

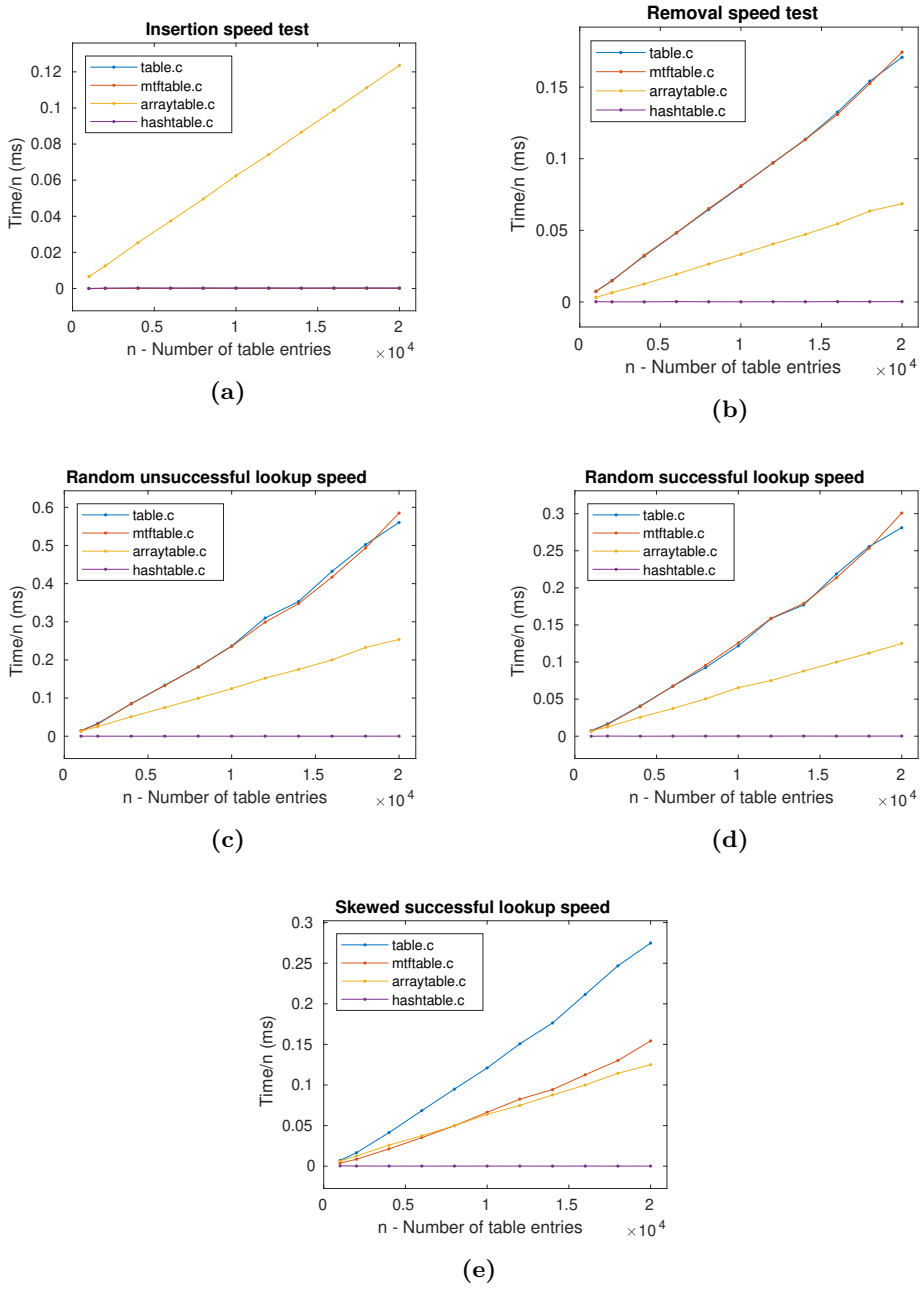


Figure 2 – Relative averaged result of the speed test suite, for all table implementations. Average run-time per call to the table interface plotted as a function of entries in table. The sample size tested, in number of table entries n, was 1000, 2000 and up to 20000 in 2000 increments. For each sample size, the average run-time of five test program executions was calculated.