Assignment 3 **May 19, 2025**

5DV149 Datastrukturer och algoritmer

Assignment 3 — Comparison of Table implementations

version 1.0

Marc Meunier Marc Meunier
tfy22mmr tfy22mmr

Contents

1 Introduction

In this lab the goal was to become more familiar with datatype, especially the abstract datatype table and understanding why abstract datatype can be useful. We also trained on how to implement them and trained on our written presentation form. We implemented the abstract datatype table with two different methodes. The first one was using a method where we moved every looked up entry to the first position. Useful for items that are often looked up. Second method we implemented was an array implementation that have the benefit of allocating the array at the start.

2 Datatypes

2.1 The Table user interface

- **table_empty(key_cmp_func,key_kill_func,value_kill_func)** — Returns an empty table and takes kill functions as input.
- **table_is_empty(table t)** — Returns true if table **t** is empty, false if table is nonempty.
- **table_insert(t, key, value)** — Adds **key** and **value** to the table **t**. Returns nothing.
- **table_lookup(t, key)** — Returns the value stored in the table **t** associated with the **key**.
- **table_chose_key(t)** — Returns arbitrary key from the table **t**.
- **table_remove(t, key)** — Removes the **key** and associated value stored in the table **t**. Returns nothing.
- **table_kill(t)** — Kills and deallocated the memory used for the table **t**. Calls all kill functions inserted at table creation. Returns nothing.
- **table_print(t)** — Prints all elements in the table **t**. Returns nothing **print_func**

2.2 Implementation details

2.2.1 Table

In the table implementation, the datatype dlist is used. Inserted values are stored in the top of the list, meaning the lookup function will return the upper most value first, thus always the latest value. No switching is made for looked up values meaning the same lookup time when searching for the same value 2 times in a row.

Table 1: A good caption, e.g., asymptotic complexity for the table operations for the different table implementations.

	`table`	`mtftable`	`arraytable`	...
`empty`	$O(1)$	$O(1)$	$O(1)$	
`isempty`	$O(1)$	$O(1)$	$O(1)$	
`insert`	$O(1)$	$O(1)$	$O(n)$	
`lookup`	$O(n)$	$O(n)$	$O(n)$	
`remove`	$O(n)$	$O(n)$	$O(n)$	
`choose_key`	$O(1)$	$O(1)$	$O(1)$	
`kill`	$O(m)$	$O(m)$	$O(m)$	
`print`	$O(m)$	$O(m)$	$O(m)$	

2.2.2 MTFTable

In MTFTable, the datatype dlist is used for manipulating values. It's similar to the Table 2.2.1 but differs in the lookup function. Looked up values are placed in the top of the list and removes the looked up value for performance gains. This improves the speed if the same values are looked up 2 or more times in a row.

2.2.3 ArrayTable

In the arraytable, for inserting values, i first checked that the keys weren't already assigned to the table. Afther this, if the key had matched i replaced the old information with the new so that the old value wouldn't be returned. And if the key did not match, the value + key was inserted at the first empty location. The values inserted into the list was counted and placed in the table.
For removing, the matching value was found in the array and then deleted. Lookups are performed by comparing each entry key with the search key returning the value associated in the position of the matching key. To kill the table, all positions was searched through and deallocated.

2.3 Complexity analysis

A simple complexity analysis was performed on all of the methods with only regards to the large complexity $O(n)$. *arraytable* differs from *mtftables* and *table* because of the different implementations and underlying datatype used. The *arraytable* has a more complex insert function as it has to be certain that the inserted value/key wont affect the already inserted entries. This can be especially important because of the preallocated space used for the array, that can be seen as a upper limit of entries saved.
One could think the *mtfarray* would have a more complex lookup function compared to *table* but the only difference between them is moving the value to the top, witch requires only one operation $O(1)$ and because $O(n)$ is much larger, the function will therefore still be of $O(n)$.

3 Experiments

Using a bash script, five tests where run using the provided tabletest.c file. Each test was rerun 6 different times and the mean time of all the runs was taken. The tests where run on a laptop running Fedora Workstation 42 (Linux), with kernel 6.14.5. The laptop has a AMD Ryzen 7 5800H CPU. The graphs where created using MATLAB version R2024b. The CPU was unlocked and not specified to run on a specific frequency or core, meaning that the computing speed could differ during the length of time of running the tests. Thermal throttle could also occur lowering the speed of the CPU after some time.

3.1 User instructions

The written code was placed in the datastructures-v2.2.2.2.zip extracted folder, where i also included a bash script for compiling the code. The bash script calls on 3 different commands to compile the three different table examples that are used, commands are shown below. The GCC version used was "gcc (GCC) 15.1.1 20250425 (Red Hat 15.1.1-1)".

```
gcc -Wall -I include/ -g -o tabletest tabletest.c src/table/table2.c
src/dlist/dlist.c
gcc -Wall -I include/ -g -o mtftabletest tabletest.c mtftable.c
src/dlist/dlist.c
gcc -Wall -I include/ -g -o arraytabletest tabletest.c arraytable.c
src/array_1d/array_1d.c
```

3.2 Test runs

Here is an example of output of the arraytabletest program. Test was performed with 40000 entries.

```
tabletest v1.10 (2023-02-17)
Code base version v2.2.2.2.

Testing...
Isempty returns true directly after a table is created. - OK
Isempty false if one element is inserted to table. - OK
Test of looking up non-existing key in a table with one element - OK
Looking up existing key in a table with one element - OK
Looking up three existing keys-value pairs in a table with three elements - OK
Looking up existing key and value after inserting the same key three times with different values - OK
Inserting one element and removing it, checking that the table gets empty - OK
Inserting three elements and removing them, should end with empty table - OK
Inserting three elements with the same key and removing the key, should end with empty table - OK
All correctness tests succeeded!

Insert 40000 items                 : 9455 ms.
Remove all items                   : 6576 ms.
40000 lookups with non-existent keys : 14194 ms.
40000 random lookups               : 7096 ms.
40000 skewed lookups               : 7122 ms.
Test completed.
```

The test performed inserted 40000 entries, while taking the time, it took 9455 ms. Then after all entries where removed, taking 6576 ms. 40000 lookups where then performed using non existing keys taking 14194 ms. then 40000 random lookups where performed, taking 7096 ms. The last test was 40000 skewed lookups taking 7122 ms.

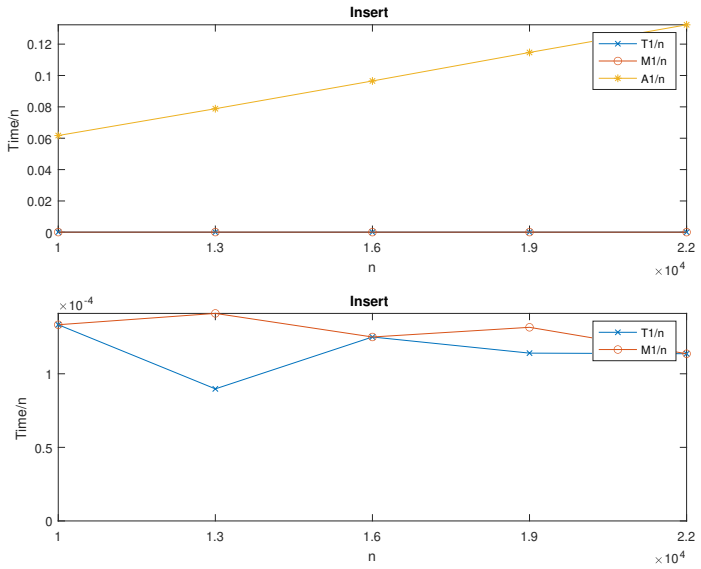

Figure 1: One

3.3 The actual experiments

The actual experiment used 5 different entries; 10000, 13000, 16000, 19000 and 22000 where each was rerun 6 different times and output was saved in a separate text file for every table tested. An example of the output from the code can be seen in 3.2.
Time was plotted in matlab where we the results where divided by the number of entries used, meaning the graphed time shows mean time for one operation (example inserts) during a test with n entries (example 10000).

4 Results

As can be seen in the plots 1, 2, 3 the different test did not take the exact same amount of time to run. For figure 1 we see that the insert function of *arraytable* increases linearly, while the other tests stay constant over increasing n. We can also see that *Table* and *MTFTable* does not differ all that much in insert speed. In the second figure, 2 we see that the *arraytable* is faster then the other implementation at removing.
The third test, 3 shows how the *arraytable* has faster failed lookups, faster random lookups and slightly faster skewed lookups. Both *Table* and *MTFTable* have very similar failed and random lookups, whereas *MTFTable* has faster Skewed lookups.

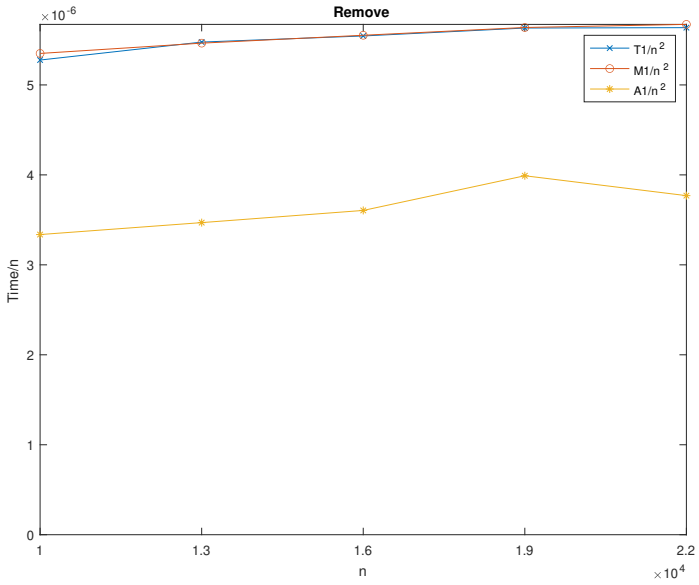

Figure 2: Two

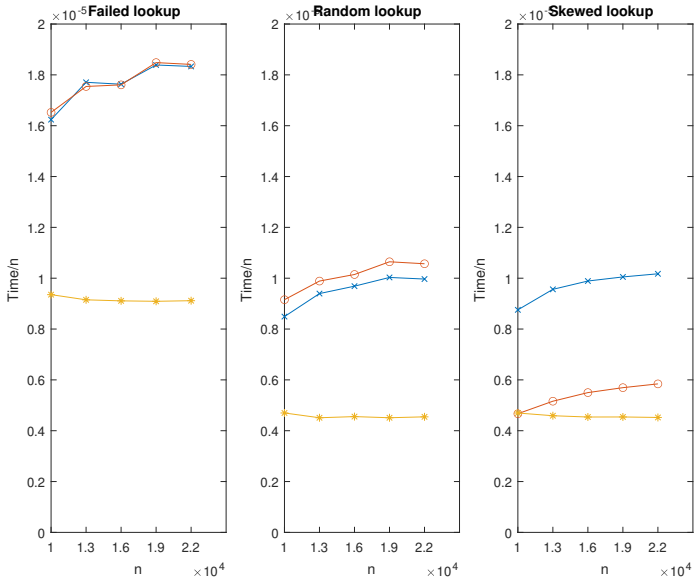

Figure 3: Three

5 Discussion

5.1 Discussion

The results make sense, the insert function of *arraytable* has to find the first empty location in the array to insert whereas the other implementations just have a constant insert speed because of putting the values first. If the same key is inserted, we also overwrite the old key, meaning the insertion has some more operations to perform.
The remove function on *arraytable* is faster, because of only needing to remove the first found entry matching with the key. This is because of the implementation in the insertion part where it removes duplicates.
I did not think *arraytable* would be the fastest in all lookup tests. Especially in the Skewed lookup, as it does not move the lookup to the front. We can see that the *MTFTable* has some performance gain in that test because of moving the entries to the front, but it is still behind *arraytable* on larger datasets. Arrays seems to be faster to search through then the linear implementation in *Table* and *MTFTable*.
The first two implementation *Table* and *MTFTable* will be best in cases where a lot of insertions are necessary and data isn't often removed. If more frequent data that is inserted is accessed *MTFtable* would be the best. But the *arraytable* implementation is best suited for cases with many lookup, and where entry deletion is often done, and where insertion time is not the priority. It will also be more efficient when overwriting entries with same key, as it wont use more memory on a overwrite whereas other implementations still saves the old keys until the delete function runs.

5.2 Conclusion

The implementations can be usefull in different cases depending on what operations are most common. For tables that often change entries, the *arraytable* is best used because of it's speed when removing and also for using less data when overwriting. *Table* and *MTFTable* both are usefull when inserting speed is critical, and access and removing is less often done.

5.3 Reflections

This was interesting coding. It reminds me of different databases where each one is better at different tasks. I got a lot of memory error in the beginning because i found it hard to work with memory. This took a lot of time fixing and was not the funniest part. I also started on the code a bit late, took a while to figure out what exactly we were meant to do, witch has made my week very stressful (a lot of late coding nights and mornings). I hope i won't have that many faults in the implementation, as I have a lot of recap in other courses to do.