UMEÅ UNIVERSITY
Department of Computing Science
Assignment report

March 2, 2020

5DV149 - Assignment 4

Data Structures and Algorithms (C) Spring 2019, 7.5 Credits

Comparing time complexity among some implementations of tables

Name	Rasmus Lyxell
user@cs.umu.se	`c19rll@cs.umu.se`
Version	`1.0`

Teacher
Niclas Börlin

Graders
Niclas Börlin, Ola Ringdahl, Anna Jonsson, Fredrik Peteri, Lennart Steinvall

Contents

Disclaimer: All the following sections can be broken down further (e.g. "Intro -> Intro + Theory").

1 Introduction

In this report the time complexity of three different implementations of the data type "table" will be compared. Two of the three tables will be implemented using directional lists while the third will use one dimensional arrays. The relevant parts of the interfaces for the relevant data types are described in this section.

1.1 Table interface

The table interface is taken from the header file of the provided code base for the assignment. It is defined through the following functions

1.1.1 table_empty

```
//Function declaration
table *table_empty(compare_function key_cmp_func, free_function key_free_func,
```

Takes three functions for comparing keys, freeing keys, and freeing values respectively as input and creates an empty table that uses said functions for freeing memory allocated by the user (when using functions such as table_remove() and table_kill()) and for comparing keys. Optionally takes NULL as a free_function to not deallocate memory outside of the implementation. Returns a table pointer to a new table.

1.1.2 table_is_empty

```
//Function declaration
bool table_is_empty(const table *t);
```

Takes a table pointer as a constant and checks if said table contains no key/value pairs. Returns True if the table is empty and False otherwise.

1.1.3 table_insert

```
//Function declaration
table *table_insert(table *t, void *key, void *value);
```

Takes a table pointer and two pointer two a key and a value. Inserts said key/value pair into the table. Controlling duplicate entries with the same key is defined by the implementation of the table.

1.1.4 table_lookup

```
//Function declaration
table *table_lookup(const table *t, const void* key);
```

Takes a table pointer and a pointer to a key. Uses the key_cmp_func that is defined on table creation to compare the input key with keys in the table. Returns a pointer to the corresponding value of the most recently inserted pair if the key matched any key in the table. Otherwise it returns NULL.

1.1.5 table_choose_key

```
//Function declaration
void *table_top(table *t);
```

Takes a table pointer and returns an arbitrary key from the table.

1.1.6 table_remove

```
//Function declaration
table *table_remove(table *t, void *key);
```

Takes a table pointer and a pointer to a key and removes all pairs associated with said key in said table. Uses key_free_func and value_free_func to free both the key pointer and the value pointer if the functions were specified on table creation.

1.1.7 table_kill

```
//Function declaration
void table_kill(table *t);
```

Takes a table pointer and returns all allocated memory used by said table and its elements. If a free_function was defined at creation then table_kill calls it for each key and value in the table to free user-allocated memory used by the element values.

1.1.8 table_print

```
//Function declaration
void table_print(table *t, inspect_callback print_func);
```

Takes table pointer and a printing function and prints the key/value pairs of said table using the provided function.

2 Methods

The implementations that are going to be tested are:

- An implementation using a directional list which handles duplicates on inspect and remove referred to as Dlisttable.
- An implementation using a directional list which moves entries to the start of the internal list when looked up and handles duplicates on inspect and remove referred to as MTFtable.
- An implementation using a one dimensional array which handles duplicates on insertion referred to as Arraytable.

2.1 Testing the implementations

All three implementations will be run through a test provided by the course. The provided test tests five different things.

1. Insertion speed
2. Removal speed
3. Lookup speed with...
   (a) Random keys from the table
   (b) Non-existing keys
   (c) A smaller set of the same keys being looked up repeatedly.

Each test tests the relevant operation a given amount of times. For this assignment it will be run with element amounts running from 4000 to 40000 in 4000 element amount steps with 5 runs each to ensure better results. The results averages will be in the form of a function table for the time elapsed as a function of element amounts eg. $T(5000) = 2000ms$ etc. It will be done with a script which automates the testing process and sorts the results in a pleasant way. The script is run using the salt and pepper servers owned by the Computer Science institution at Umeå University running one at a time to ensure minimum amount of performance difference between test run by the script.

3 Results

The results are divided into two parts; asymptotic analysis and experimental analysis. The asymptotic analysis predicts the big O notation of each operations time taken as a function of the amount of elements in the table. The experimental analysis provides data points for approximating a function for time taken for each operation and gives real results tied to the hardware it was performed on.

3.1 Asymptotic analysis

Asymptotic analysis is analysing the code time complexity without regards to hardware or similar factors. For this assignment the analysis is only made on a surface level to judge what the big O for each part will be.

3.1.1 Insertion

The Dlisttable simply pushes the key pair to be inserting on the first position of the list.

The MTFtable behaves identically as the Dlisttable on insertion and also has a big O of.

The Arraytable differs as it checks for duplicates to the inserted key in the insertion. As such it has to in the worst case search every key in the table and in the best case search one element into the array.

Therefore the big O's should be as shown in Table 1.

Table type	Best case big O	Worst case big O
Dlisttable	$O(1)$	$O(1)$
MTFtable	$O(1)$	$O(1)$
Arraytable	$O(1)$	$O(n)$

Table 1: Big O's from asymp. analysis for insertion

3.1.2 Removal

The Dlisttable and MTFtable behave identically on removal having to traverse the entire table to search for all entries matching the key to be removed. This is because they allow duplicates to be stored in the table.

The Arraytable on the other hand only has to traverse the table until it encounters a single matching key. This is because it guarantees that only one entry exists per key.

As such the big O's for removing an entry is as shown in 2.

Table type	Best case big O	Worst case big O
Dlisttable	$O(n)$	$O(n)$
MTFtable	$O(n)$	$O(n)$
Arraytable	$O(1)$	$O(n)$

Table 2: Big O's from asymp. analysis for removal

3.1.3 Lookup

For lookup both Dlisttable and MTFtable are implmented such as to return the most recently added entry matching the given key. Because of that they both have to traverse the table until they find a matching key.

The difference between the two is that MTFtable moves the most recently looked up entry to the front of the internal dlist. As such, MTFtable will be slightly slower when looking up random keys but will excel in workloads that include looking up smaller groups of keys from the table, so called skewed look ups.

The Arraytable simply traverses the table and looks for a matching entry in all cases.

Because of the different kind of lookup loads that are placed on the table, different tables should perform better or worse in specific tests.

According to these facts the results should look like those in Table 3.

	Random lookup		Non-existing lookup		Skewed lookup	
Table type	Best	Worst	Best	Worst	Best	Worst
Dlisttable	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
MTFtable	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Arraytable	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$

Table 3: Big O's from asymp. analysis for lookup

Worth noting is that identical big O notations does not mean that they necessarily perform the same.

3.2 Experimental analysis

The experimental analysis of the time complexities comes from the tests mentioned in section 2.1. Using the different points of data we can interpolate a graph like shown in Figure 1 showing $T(n)$ for the the tests themselves. The $T(n)$ for each operation (insertion, lookup, etc.) is obtained by dividing the tests $T(n)$ by the amount tested (n). The resulting graphs for each implementation is shown in Figure 2.

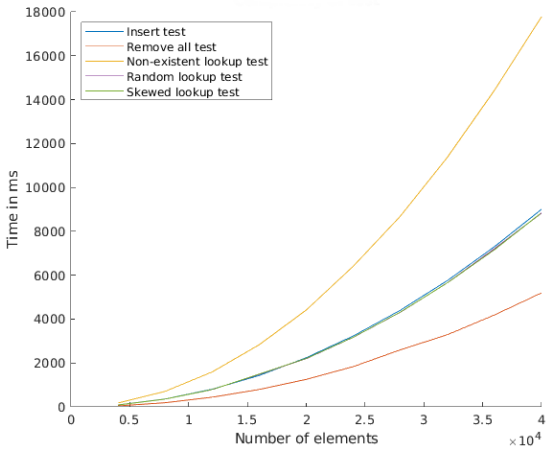
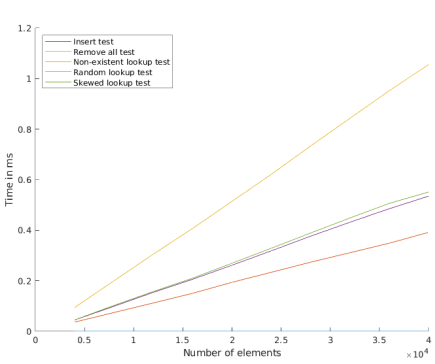
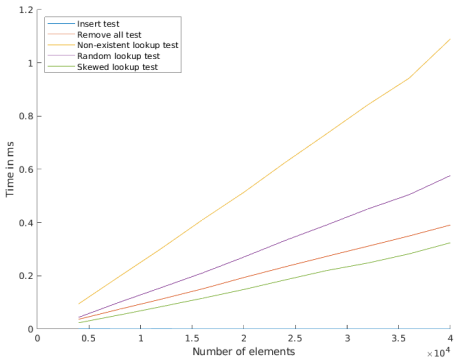


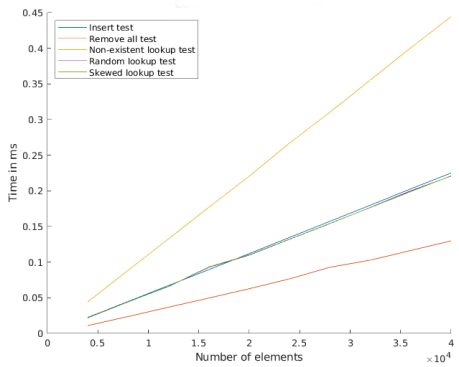
Figure 1: Time complexity graph for the provided test using Arraytable implementation



(a) Dlisttable



(b) MTFtable



(c) Arraytable

Figure 2: Each implementations time complexity per operation

Insertion is $O(1)$ for Dlisttable and MTFtable as predicted in section 3.1.1 while Arraytable seems to be $O(n)$ which matches the same prediction. Another noteworthy result is that when we compare the MTFtable with the Arraytable (Figure 3) we see that the Arraytable is considerably faster than both MTFtable and Dlisttable at each operation excluding insertion.

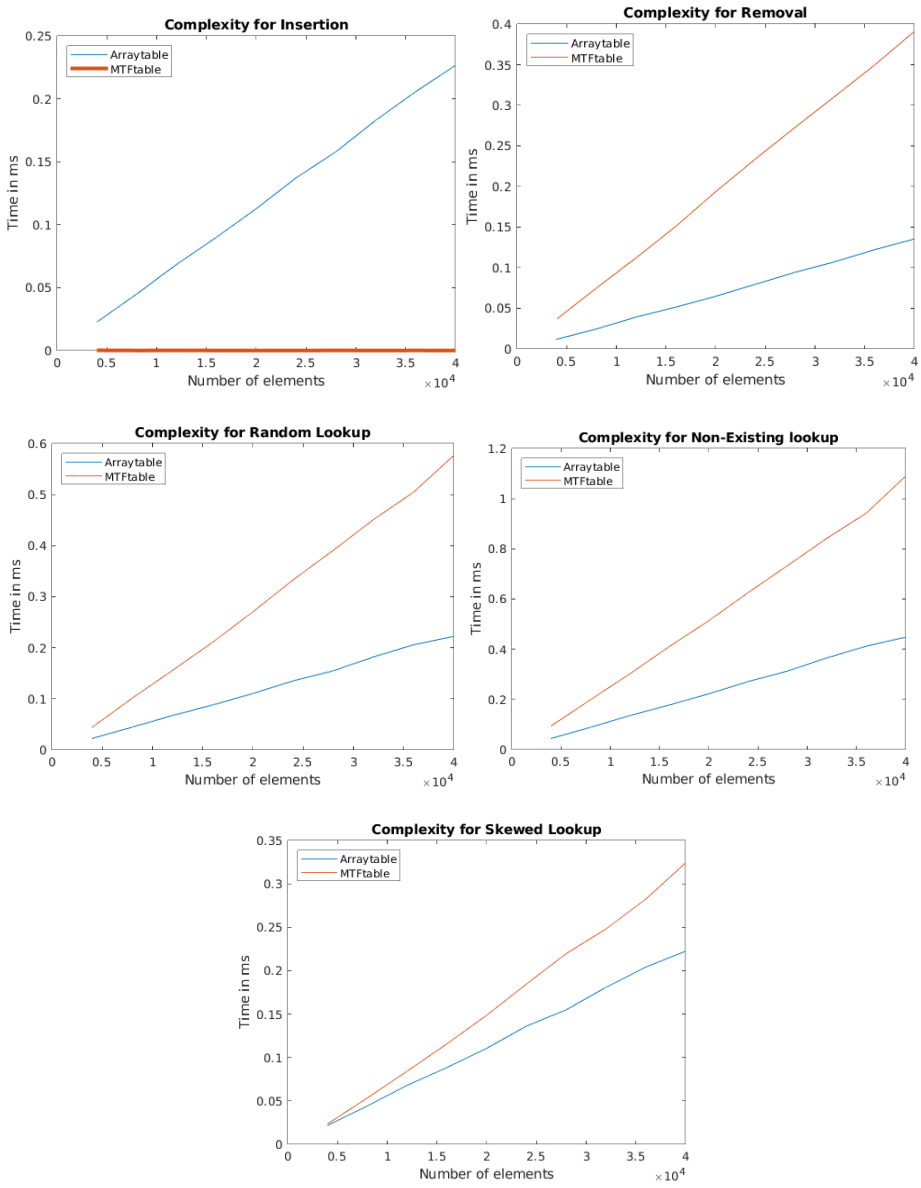
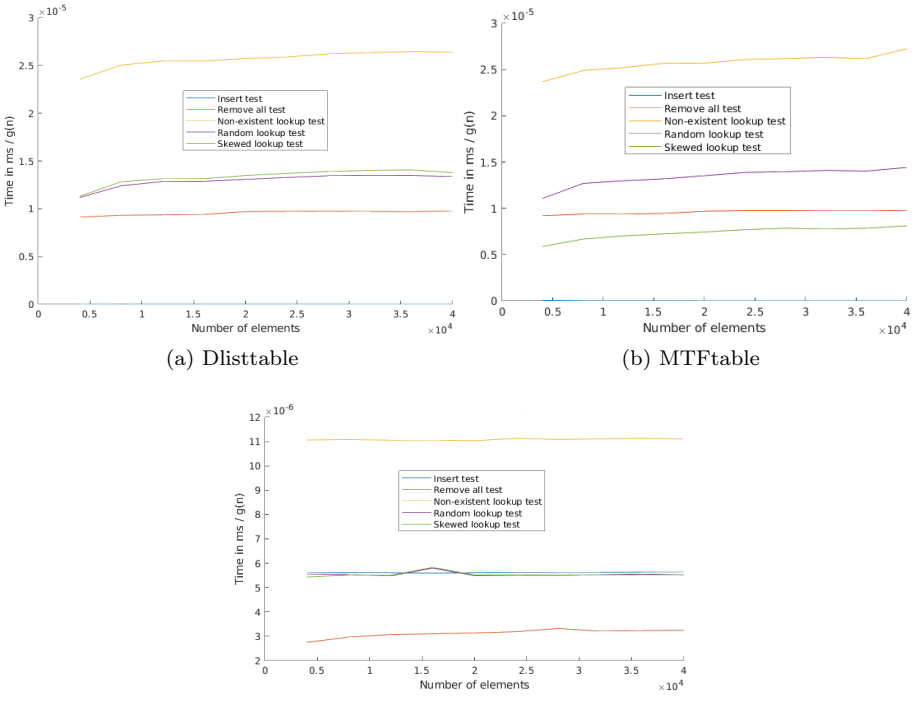


Figure 3: Graph comparing the results of Arraytable and MTFtable

The rest of the time complexities for all implementations all seem to land on $O(n)$ which also matches the predictions from the asymptotic analysis. We can check each implementations hypotheses by dividing the obtained $T(n)$ for each operation with our hypothesised $g(n)$ from our $O(g(n))$. If they are correct the resulting graph should be more or less constant or approach a constant.



(a) Dlisttable

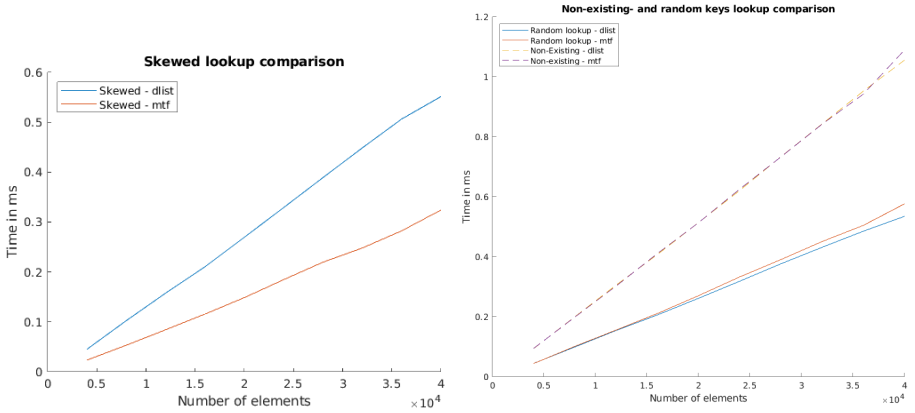
(b) MTFtable

(c) Arraytable

Figure 4: Each implementations time complexity per operation divided by their predicted $g(n)$

As seen in Figure 4 the graphs all stay constant within at most a few millionths. Therefore the hypotheses were correct.

As for the individual difference in constants (meaning the slope of the graph) for the operations complexities according to the analysis done earlier the MTFtable should outperform Dlisttable in skewed look ups and vice versa for random look ups. The speed for non-existing look ups should be identical.



(a) Skewed lookup (b) Random lookup

Figure 5

As shown in Figure 5 the speed of each operation is as predicted. There is a very noteworthy difference between the implementations in skewed lookup but the difference in random lookup is no where near as noticeable.

4 Discussion

In summary, the array implementation performed better in all tests except insertion and the MTFtable performed noticeably better in skewed look ups and slightly worse in random look ups compared to the Dlisttable. The Arraytables difference in speed most likely comes from the $O(1)$ complexity of inspecting and element in an array.

Something that the provided test did not test was the memory usage of each implementation. The array based implementation allocates a static size at the initialisation of each new table while the list based ones do not. As such the Arraytable uses more memory.

There were no noticeable anomalies in the tests which might be partly because of the script used for testing and partly because all tests were run on the same hardware. Using the results you could conclude that a table implemented to handle duplicates on removal and inspection would be favorable if the work loads placed on it mostly consisted of insertions. Otherwise, more variable workloads work better with implementations that handle duplicates on insertion.