Umeå University
Institution för Datavetenskap
Rapport obligatorisk uppgift

DV1: Datavetenskapens byggstenar 7.5 p
DV160HT15

OU3 Tables

Submitted	2015-12-17
Author:	Lorenz Gerber (`dv15lgr@cs.umu.se`)
Instructor:	Lena Kallin Westin, Johan Eliasson, Emil Marklund, Lina Ögren

Contents

1 Introduction

The aim with this laboration was to understand and a set of given composite data types and then built on those implement two new ones. Test code for correctness and speed was given. The given datatypes were directed *list*, *array*, and a *table*, implemented using the aforementioned directed list. The used programming language was *C*. The datatypes to be implemented were table using a *move-to-front list* and *table* using *array*.

The implementation of the datatype *table* is according to the defintion in the coursebook [1, pp. 117 – 132]. In Brief: a *table* is a finite mapping of arguments to values. The textbook states a dictionary as a model where the lookup words are the arguments and the textual explanations the values. The datatype *table* however, does not need to be ordered by definition.

The operations found in the interface of a table are *empty*, to create an new empty *table*, *insert* to put new argument/value pair into an exisiting *table*, *isEmpty* to check whether an existing *table* has any content, *lookup* to access the value associated to a potentially available argument in an existing table and *remove* to discard a potentially existing argument/value pair from an existing table.

The assigment included also testing and benchmarking of both the given and the requested table implementations. The benchmarking was ment to allow discussing the different implementations from a number of different aspects such as lookup speed of existing and non-existing arguments, insertion and removal speed as well as skewed or biased lookup. As a starting point, a number of mandatory questions and discussion topics were given in the lab instructions.

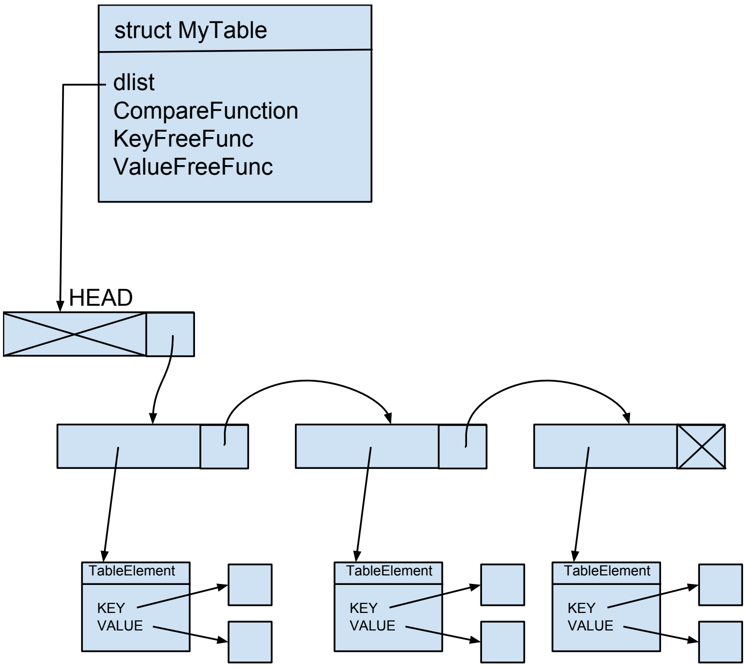

Figure 1: *Construction of the datatype dlist table and move-to-front list table.*

2 Material and Methods

The description of the datatype *table* in the introduction applies to the actual implementations which will be described below. A small difference is that the coursebook [1], uses the wording *argument* while the obtained implementation of a directed list table uses the term *key* in the code.

2.1 Directed List Table and Move-to-Front List Table

The structure of a *directed list table* and *move to front list table* is the same and can be seen in *figure 1*. The given *dlist* was constructed with a *HEAD*, using physical and logical position. The physical position is always one element before the logical. That means, when requesting content of the first list element, the pointer is directed to *HEAD*. Then *head->next* will be used to access the content of the first cell. This setup is needed in a single directional list mostly for the *insert* operation: The new element is inserted before the current element. If physical and logical position would be the same, it would be needed to travers the list again up to the position where the new element shall be inserted.

The special feature about the move-to-front list table is the lookup operation: each time, an element is looked up, it is moved to the front of the directed list. A schematic of this operation can be seen in *figure 2*. The arrows in black show the initial situation and the red arrows new situation after moving the looked up cell to the front. In *figure 2*, the cell to be moved to the front is the one in the middle.

The request to remove a non-exisiting key result for both the given *directed list table* and the move to front list table in a traversal of the whole list, altough without any feedback. In case of duplicate key values on an insertion, the old key/value pair is overwritten. In more abstract terms, duplicates are handled at insertion time.

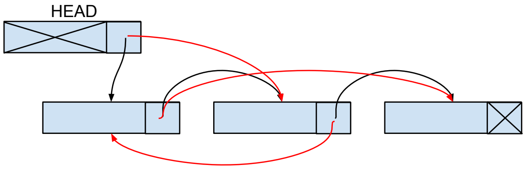

Figure 2: *move to front operation in a directed list.*

2.2 Array Table

The structure of the array table can be seen in *figure 3*. Instead of the *directed list* it is now an *array* of *void pointers*. The table is initialized to *NULL pointers*. On insertions, a *void pointer* of the array field is then set to point of the new table element. The current implementation traverses the array from low to high until an array field with a *NULL pointer* or the same *key* as the one to insert is found. On removal of a key/value pair, the whole *array* is traversed to discard eventual duplicate *key* entries. In more abstract terms, duplicates are handled partly at insertion time and partly but definite at removal time. If a non-existing *key* is given as argument to the remove operation, the whole *array* will be traversed. In the current implementation no feedback is given whether a key was found and removed or not.

An important feature to mention is that by definition an array is a static datatype while a table is dynamic. This will be discussed in more detail. Also by definition, they index of an array is usually a natural number or integer which does not match with the definition of the argument/key in a table. A direct mapping from argument/key to array index is therefore in the most general case not possible.

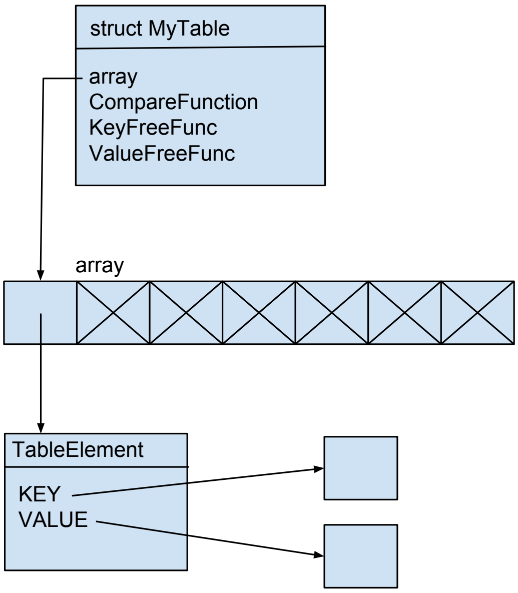

Figure 3: *Construction of the array table data type.*

2.3 Proposed changes to the interface

Currently, the interface specification does not implement any return value for the operations *remove*,*insert* and *free*. For those three, a boolean feedback whether the operation succeeded or not could make sense. This would for example provide the information whether a *key* requested for removal was available in the table or not. For the *insert* operation, it could provide a signal in the *table array* implementation when the underlying array is full with values.

3 Results

3.1 Correctness of datatype implementations

To assess the correctness of the implemented datatypes a set of eight tests was provided: Correctness of the data types was assessed by eight

- isEmtpy on a new created table yields TRUE
- insert a single element
- lookup a single element
- insert and lookup multiple elements with non identical keys table
- insert and lookup multiple elements with identical keys
- remove a single element
- remove elements with different keys
- remove elements with the same key

The tests followed to a large extent the axiomatic table defintion given in the course book [1, p 122]. All the implemented tables fulfilled the test criteria.

3.2 Performance tests

The performance of the implemnted datatypes was assed by five different tests that represent typical operations and use cases for the datatype *table*.

- Insertion speed
- random existing lookup speed
- random non-existing lookup speed
- skewed lookup speed
- remove speed

All above mentioned tests took as parameter the number *n* of operations to be run for each separate test. This allows to assess the time-complexity when performing the test for a series of increasing *n*. The test data generation procedure was also provided and was based on the C *rand* function. To account for worst/best case scenarios stemming from the random numbers, each test was run 10 times. Results below are presented as scatter plots (a) with arithmetic means of ten runs for each *n* operations case. The (b) plots show the relative standard deviation (RSD) for the 10 replicate runs in each *n* operations point.

To simplify the whole procedure, an additional block of code was added to each test function which caused to append the test results to a text file with the name *benchmark.txt* (listing 1).

Listing 1: code block for writing benchmark data to file

```
FILE *f = fopen(‘‘benchmark.txt’’, ‘‘a’’);
fprintf(f, ‘‘%lu",end−start);
fprintf(f,"\t");
fclose(f);
```

The text files where then imported into the statistical programming language ‘R’ [2]. Data aggregation, statistics and plotting was done in the aforementioned language according to a separate protocol (attachement).

The five first result figures (*Figures 4 to 8*) present data on a per test basis. The *(a)* plots show the average benchmark time for 7 datapoints (*n = 250, 500, 1000, 2500, 5000, 7500, 10’000*). Each plot contains three data series for the benchmarked datatypes.

The *(b)* plots present the relative standard deviation (RSD) of the experiments shown in *(a)*.

Figure 9 to 11 show average test time and RSD per data type for 3 selected benchmarks (*Random Existing Lookup*, *Random Non-Existing Lookup* and *Skewed Lookup*). Those three figures show now new data but allow more convenient intpretations compared to the per test figures.

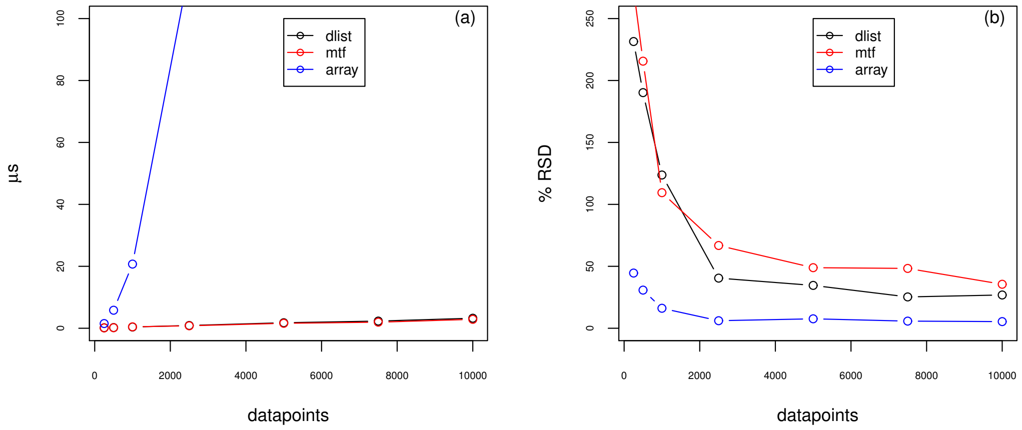

Figure 4: *Figure 1a, shows the element* ***'insertion'*** *times. Figure 1b shows the RSDs from the measurments in figure 1a. n=10.*

Figure 4 shows the results for the *insertion* benchmark. The speed for *dlist* and *mtf* datatype were about the same while the *array* implementation was much slower. It should be noted that for all conducted tests on the *array* implementation, the size of the *array* was always adapted to the needed size. RSD's for the *array* implementation were lower than for the other two data types altough with a similar behaviour: High RSD's for small *n's* and lower for high *n's*. The curves decreases quick at low towards higher *n's* and becomes very flat and stable after about 5000 *n*.

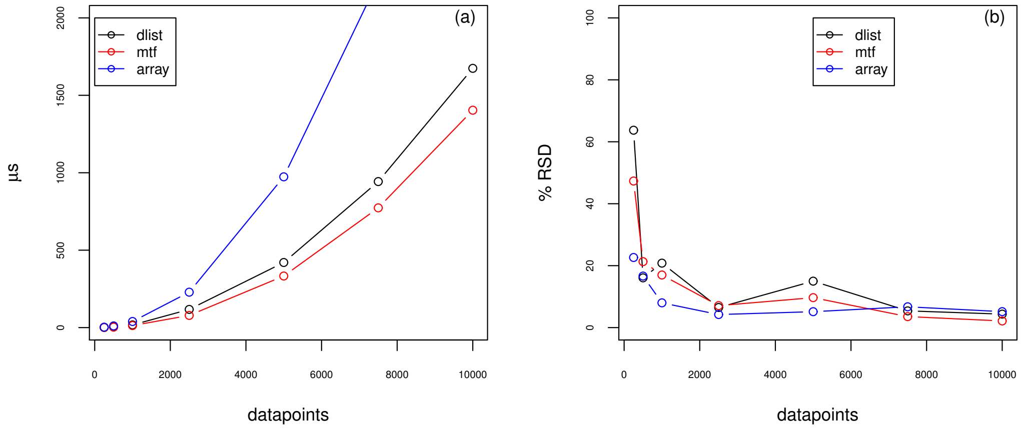

Figure 5: *Figure 2a, shows the times for the* ***'Random Existing Key Lookup'*** *benchmark. Figure 2b shows the RSD's from the measurments in figure 2a. n=10.*

Figure 5 shows the results for the *Random Existing Key Lookup* test. *dlist* and *mtf* datatypes show smimilar mean times while *array* is again the slowest. The RSD's are very similar for all benchmarked datatypes.

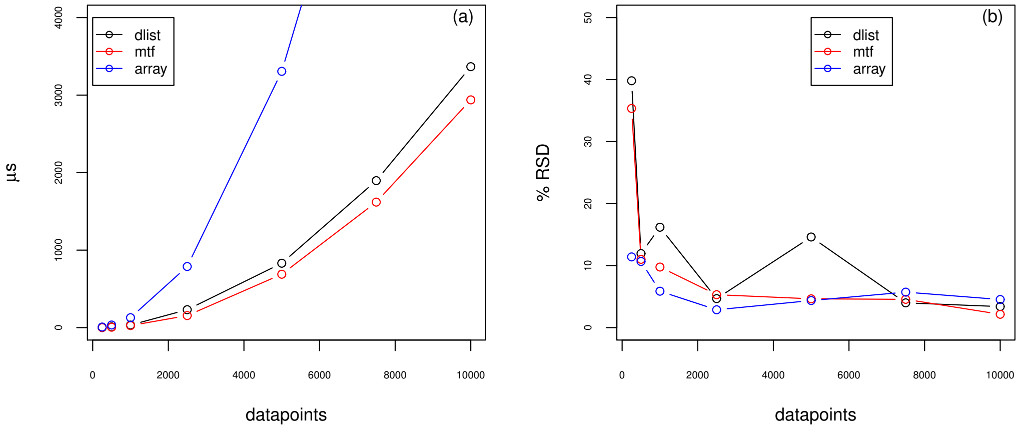

Figure 6: *Figure 3a, shows the times for the* ***'Random Non-Existing Key Lookup'*** *benchmark. Figure 3b shows the RSD's from the measurments in figure 3a. n=10.*

Figure 6 shows the results for the *Random Non-Existing Key Lookup* benchmark. The times are about double compared to the *Random Existing Key Lookup* test with the same ranking where *dlist* and *mtf* are about the same and the *array* implementation is the slowest. The RSD's for *mtf* and *array* datatype look similar as in the *Random Existing Key Lookup* however the *dlist* resulted in a peculiar irregular curve shape for the series of increasing *n*. On closer inspection, a similar curve shape, although less pronounced can be found for all RSD value series of the *dlist* datatype.

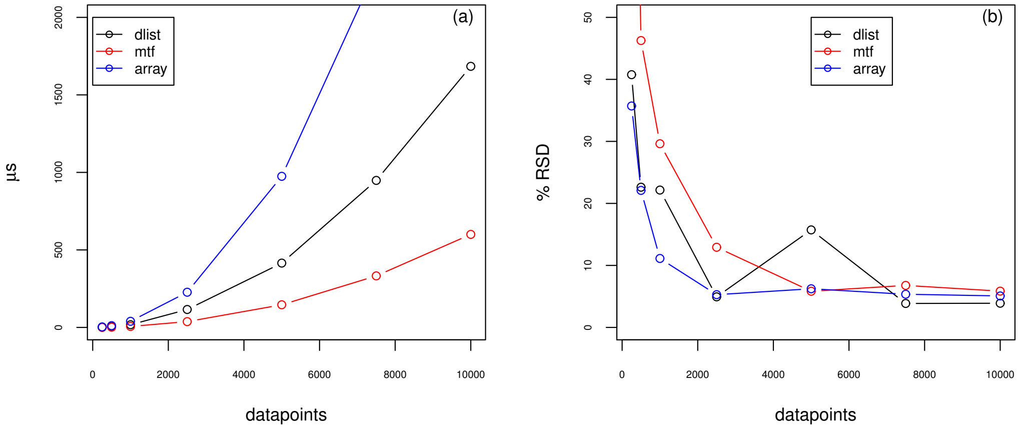

Figure 7: *Figure 4a, shows the times for the* ***'Skewed Lookup'*** *benchmark. Figure 4b shows the RSD's from the measurments in figure 4a. n=10.*

Figure 7 shows the results for the 'Skewed Lookup' benchmark where the some values are looked up with a higher frequency than random. Here the *mtf* implementation is by far the fastest followed by the *dlist* and slowest again the *array* implementation. The RSD's are similar as for the two affore mentioned benchmarks.

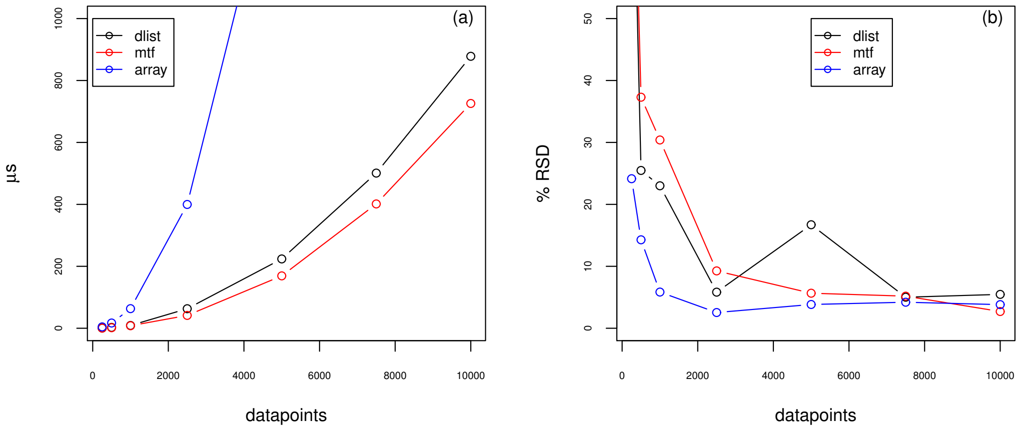

Figure 8: *Figure 5a, shows the times for the **'Removal'** benchmark. Figure 5b shows the RSD's from the measurments in figure 5a. n=10.*

Figure 8 shows the results for the *Removal* benchmark. The plot look similar like the *Random Non-Existing Key Lookup* however the times are a bit faster for all datatypes. The RSD's are in the same range as for the other benchmarks (excpet the insertion).

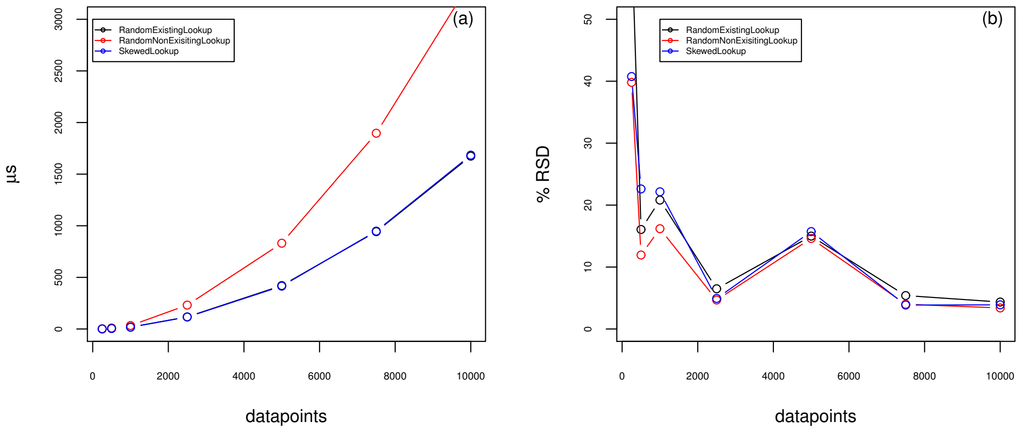

Figure 9: *Figure 6a, shows the times for the **'Random Existing Lookup'**, **'Random Non-Exisiting Lookup'** and **'Skewed Lookup'** benchmark of the datatype **'dlist'**. Figure 6b shows the RSD's from the measurments in figure 6a. n=10.*

Figure 9 shows the results for the benchmarks *Random Existing Key Lookup*, *Random Non-Existing Key Lookup* and *Skewed Lookup* for the datatype *dlist*. *Random Existing Key Lookup* and *Skewed lookup* are exactly equal fast while *Random Non-Existing Key Lookup* is a bit slower. Note the peculiar shape of the RSD data curves.

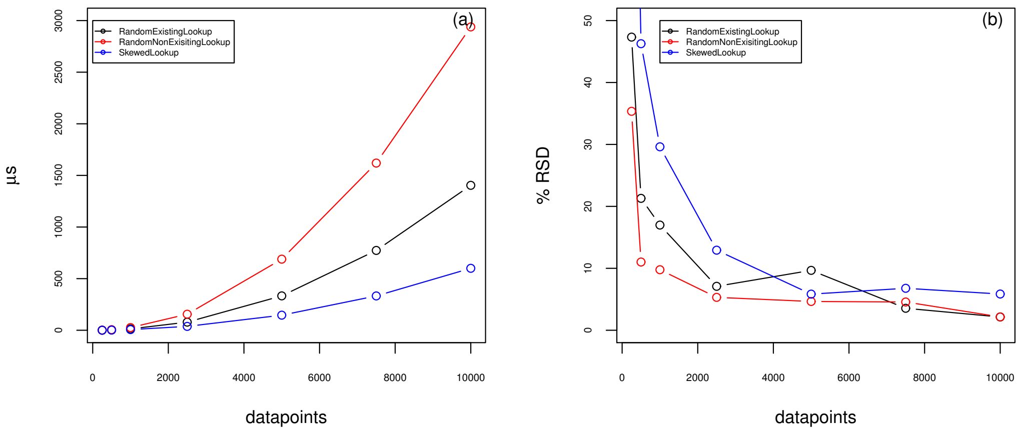

Figure 10: *Figure 7a, shows the times for the* ***'Random Existing Lookup'****,* ***'Random Non-Exisiting Lookup'*** *and* ***'Skewed Lookup'*** *benchmark of the datatype* ***'mtf'****. Figure 7b shows the RSD's from the measurments in figure 7a. n=10.*

Figure 10 shows the results for the benchmarks *Random Existing Key Lookup*, *Random Non-Existing Key Lookup* and *Skewed Lookup* for the datatype *mtf*. *Skewed lookup* here is by far the fastest followed by *Random Existing Key Lookup* and *Random Non-Existing Key Lookup*. RSD's are similar for the three cases with the common pattern, high and quick decreasing for very low *n* while low and almost flat for higher *n* values.

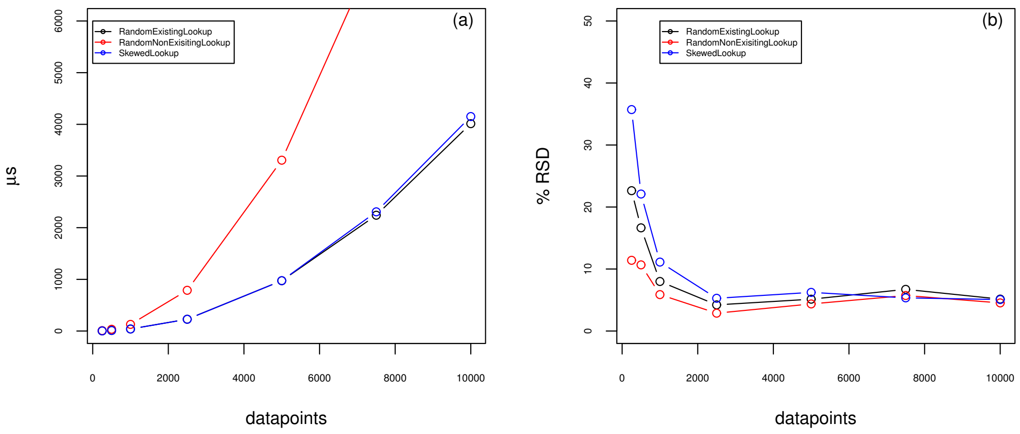

Figure 11: *Figure 8a, shows the times for the* ***'Random Existing Lookup'****,* ***'Random Non-Exisiting Lookup'*** *and* ***'Skewed Lookup'*** *benchmark of the datatype* ***'array'****. Figure 8b shows the RSD's from the measurments in figure 8a. n=10.*

Figure 11 shows the results for the benchmarks *Random Existing Key Lookup*, *Random Non-Existing Key Lookup* and *Skewed Lookup* for the datatype *array*. *Random Existing Key Lookup* and *Skewed lookup* are exactly equal fast while *Random Non-Existing Key Lookup* is a bit slower. The RSD's are very regular and stable among the three test cases.

4 Discussion

Ranking according speed for the different datatypes was obvious. The slowest implementation in every aspect by far is *array* while *dlist* and *mtf* are about equal, except for the *Skewed Lookup* test where the *mtf* implementation is fastest. The *Move-to-Front* strategy seems to be tailored for this situation. There should be quite a lot of real world applications where the distribution of lookup keys is not randomly distributed but rather skewed. A typical application could be webshops. There performance improvements can be obtained by *mtf* as there will usually be a number of *favourite* products accessed more often by customers than other products.

For operations where always or in the worst case a full traversal of the data whole structure is needed, it is obvious that the *array* implementation is slowest. However it is not obvious to me with this reasoning why the *array* implementation is slower for *insertion* and *removals*. It can be noted from the axis scale in *figure 4* and *8* that insertion is still quite a bit faster than removal. While the time critical operation in removal should be the time needed for a full traversal, this can not be the reason for slow performance on insertions. A hypothesis is that allocating a large amount of memory at once, as it is the case for *array*, takes a longer time than allocating several times very short address ranges.

An obvious disadvantage of the *array* implementation is also the static character which it forces onto the *table*. A table is by defintion a dynamic datatype. When implementing the *table* with an *array* this is then just valid for a limited operational envelope, the size of the array. A small advantage of the *array* implementation could be the very stable and well defined RSD's. They suggest that worst and best case are very similar for this implementation and as such the response times very predictable. However, whether this difference to the two other implementations is really significant would have to be tested statistically. One thing to note regarding the RSD's is the peculiar pattern found for the *dlist* datatype on a series of different *n* values. A simple statement describing this observed pattern could be: The RSD for replicates at 5000 operations is higher than the RSD at 2500 operations. This does not sound logica and at the moment, I don't have a reasonable explication for it, but the phenomena seems to replicable as it occoured in all three benchmarks *(Figure 9(b))*.

While most of the presented results suggest that implementing a *table* by an *array* is a bad choice, there are some exceptions: For a table implementation where keys are exclusively natural numbers/integers that do not spread over a large value range, an *array* should yield a very fast datatype. This comes from the fact that for the expensive operations *lookup*, *insert* and *remove* no traversal is needed. It will be quick direct acces. Also no expensive handling of duplicate values is needed in this case.

References

[1] L.E. Janlert and T. Wiberg. *Datatyper och algoritmer*. Studentlitteratur, 2000.

[2] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.