Umeå University
Institution för Datavetenskap

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

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 course-book [1, po. 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 pairs 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 a 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, number of mandatory questions and discussion topics were given in the lab instructions.

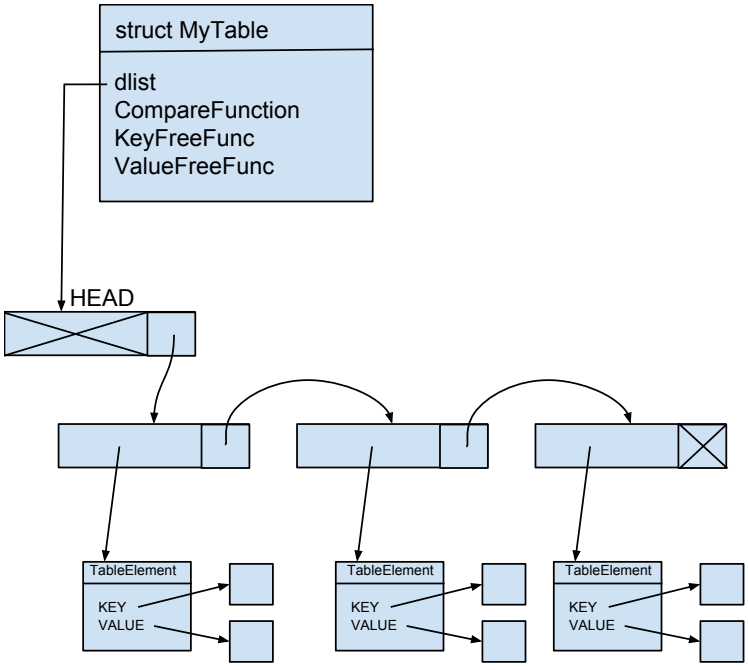

Figure 1: *Construction of the datatype dlist table and move-to-front list table.*

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’ and ‘value’ 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 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 where the physical is always one element before the logical. That means, when the requesting the 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 and array table implementation but also of the given directed list.

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 results in 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.

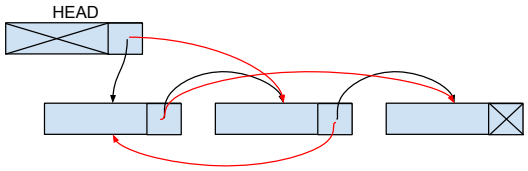

Figure 2: *move to front operation in a directed list.*

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, the void pointers of the array field is then set to point to the new table element. The current implementation traverses the array from low to high until an array field with 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.

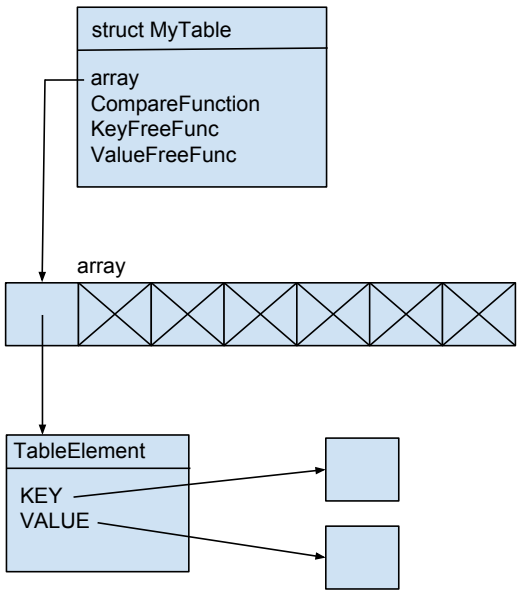

Figure 3: *Construction of the array table data type.*

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 would certainly 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

Present the results of the testing (1-2 pages, graphs/tables not included). Include the testparameters with the presentation of the results.

Two test sets were provided, one to asses correctness of implementation, the other to benchmark the speed for some typical operations.

Correctness of the data types was assessed by eight differnt tests:

- '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

The benchmarks included five different cases:

- Insert speed

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

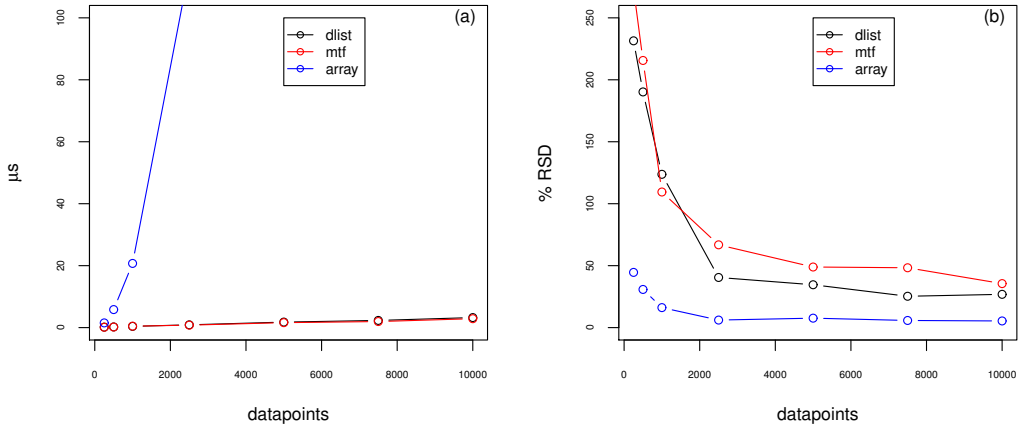

Figure 4: *Figure 1a, shows the element **'insertion'** times. Figure 1b shows the RSDs from the measurments in figure 1a. n=10.*

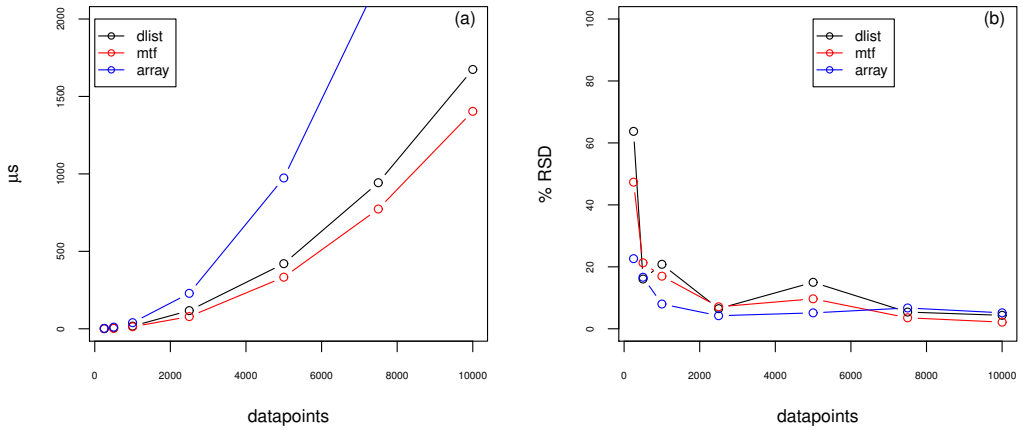

Figure 5: *Figure 2a, shows the times for the **'Random Existing Lookup'** benchmark. Figure 2b shows the RSD's from the measurments in figure 2a. n=10.*

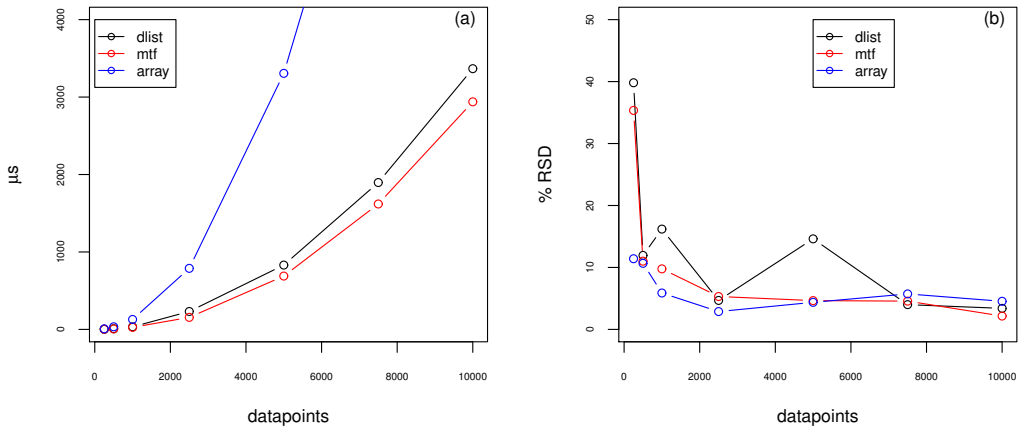

Figure 6: *Figure 3a, shows the times for the **'Random Non-Existing Lookup'** benchmark. Figure 3b shows the RSD's from the measurments in figure 3a. n=10.*

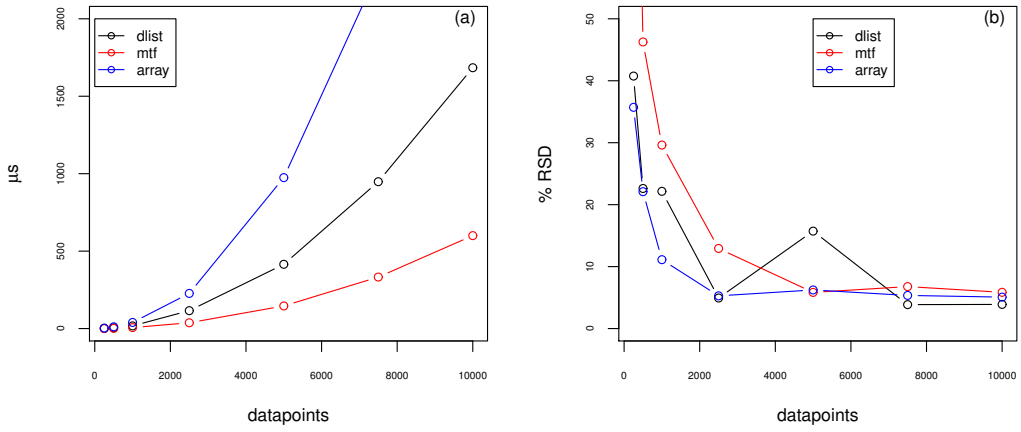

Figure 7: *Figure 4a, shows the times for the **'Skewed Lookup'** benchmark. Figure 4b shows the RSD's from the measurments in figure 4a. n=10.*

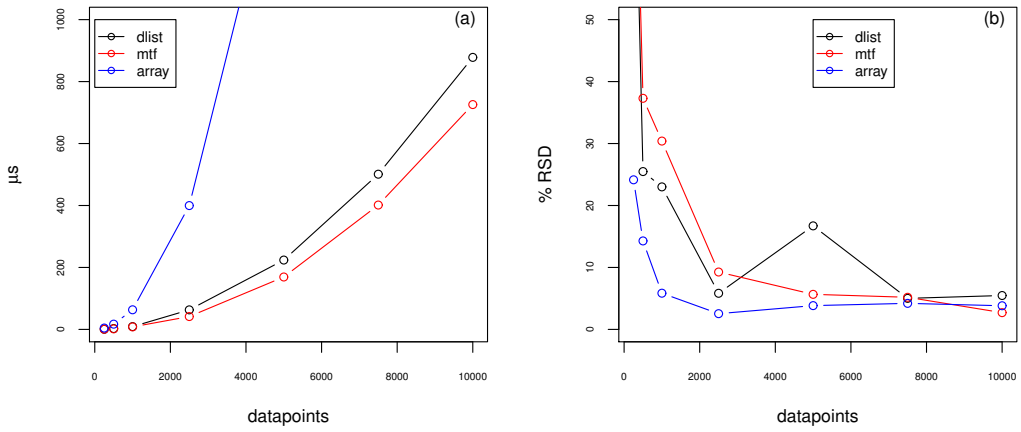

Figure 8: *Figure 5a, shows the times for the **'Removal'** benchmark. Figure 5b shows the RSD's from the measurments in figure 5a. n=10.*

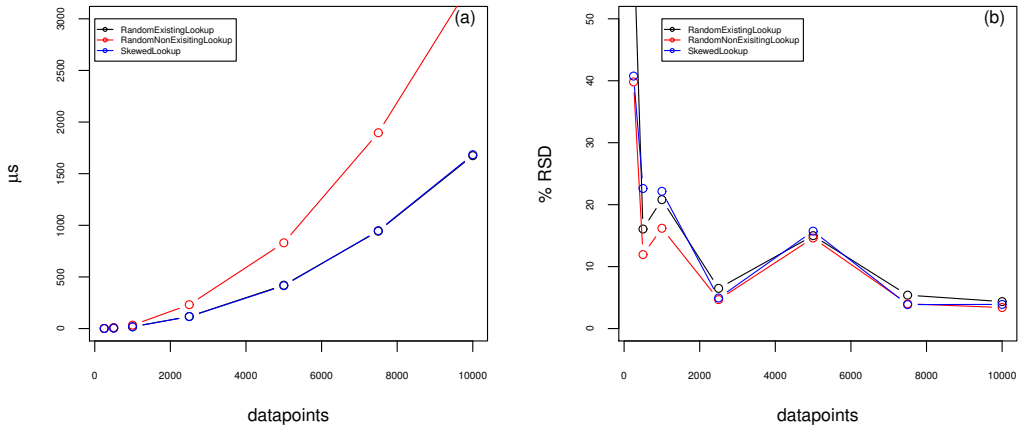

Figure 9: *Figure 6a, shows the times for the **'Random Existing Lookup'**, **'Random Non-Exisiting Lookup'** and **'Skewed Lookup'** benchmark of the datatype **'dlist'**. Figure 6b shows the RSD's from the measurments in figure 6a. n=10.*

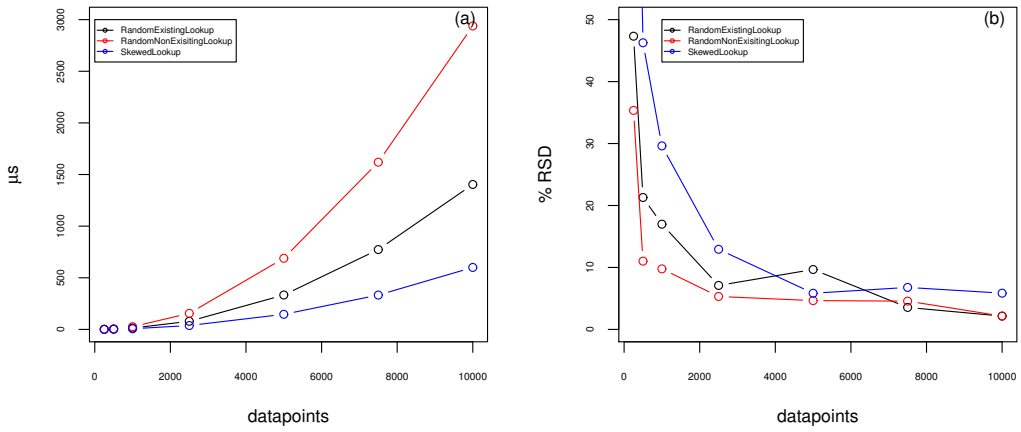

Figure 10: *Figure 7a, shows the times for the **'Random Existing Lookup'**, **'Random Non-Exisiting Lookup'** and **'Skewed Lookup'** benchmark of the datatype **'mtf'**. Figure 7b shows the RSD's from the measurments in figure 7a. n=10.*

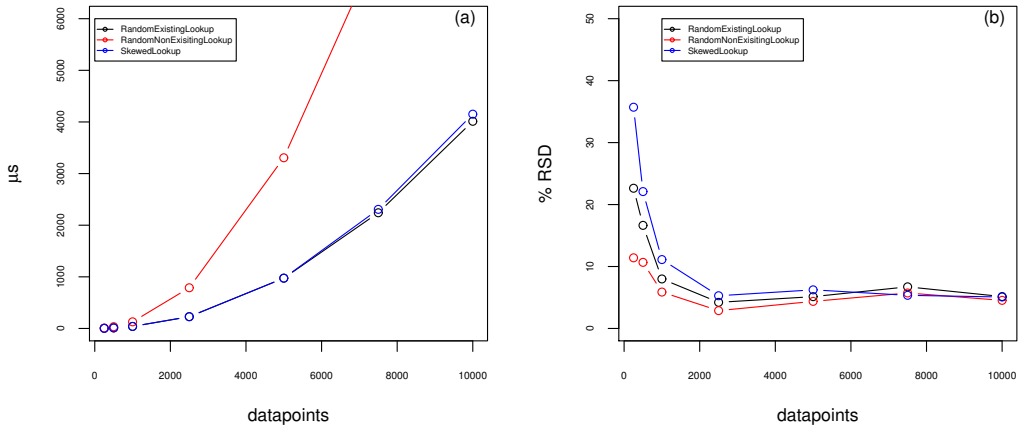

Figure 11: *Figure 8a, shows the times for the **'Random Existing Lookup'**, **'Random Non-Exisiting Lookup'** and **'Skewed Lookup'** benchmark of the datatype **'array'**. Figure 8b shows the RSD's from the measurments in figure 8a. n=10.*

4 Discussion

Analysis of the results (1-3 pages). Flowing text. Shall answer at least the following questions: Which datatype was fastest, which (dis)advantages do the different tabeltypes have? Reason for one advantage why the other implementation don't offer it. Reasoning about dynamic/static implementation. If done, reason why the direct indexing table is so fast.

References

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