**Umeå University**
Institution för Datavetenskap

# DV2: Algorithms and Problemsolving 7.5 p
# DV169VT16

## OU4 Data Representation

Submitted 2016-03-03
Author: Lorenz Gerber (dv15lgr@cs.umu.se)
Instructor: Lena Kallin Westin / Erik Moström / Jonathan Westin

## Contents

# 1 Introduction

In this assignment the aim was to specify three different possible data representations for a spreadsheet application. The representations were to be described such that they could be implemented from the descriptions. Several criteria to judge the suitability of the chosen representations were discussed in the mandatory seminar *OU2*. Three of those criteria were to be chosen and applied to judge the three selected data representations.

## 1.1 Interpretation of the Problem Description

The problem description from the course homepage defines a spreadsheet as a 'table'. Hence, a spreadsheet can be seen seen as a potentially infinite table. A requirement given in the description is that the implementation shall be more economic than an plain rectangular structure covering all non-empty table elements. From those descriptions it can be deduced that a more concise description of the problem is to find an *efficient data representation of a sparse matrix*.

## 1.2 Typical Use Cases of a Spreadsheet

During a user session, text, numbers, formulas and links to other elements are stored in the table elements. In many use cases the number of filled elements will be very low compared to the virtual rectanglular set of elements that surrounds the outermost non-empty table elements, hence the table is said to have a low *fill ratio* or represent a sparse matrix. This is the main reason why the potential data structure should store only non-empty elements.

Another property of a spreadsheet is that the data structure is represented in the graphical user interface. Due to the size, there is just one part of the data visible, hence blockwise value lookup for scrolling over the data table is one of the most common operations. Spreadsheets are often used to prepare sorted lists. Hence rearranging the order of whole rows or columns is another operation of importance. When spreadsheets are used for calculation purposes, extensive linking between memeber elements of the spreadsheet is common.

In most graphical user interfaces of spreadsheets, there are usually more rows than columns visible. This is just based on the simple fact that numbers and text extend horizontally on the screen. However, this will probably influence how the user arranges the data. It seems likely that there will be more block operations along the column direction. If the data representation is asymetrical in terms of structure or operations, this should be kept in mind for the implementation.

## 1.3 Spreadsheet as a Datatype

It was assumed that the data representation of a spreadsheet is very similar to an array, hence it makes sence to have at least the set of operations available as in an array (see listing 1) [3, pp. 92-95].

The most significant differences to a standard array is the *Remove* operations and the behaviour of the *low* and *high* operations: They will return the lowest respective hightes indices of a non-empty spreadsheet element.

**Listing 1** *shows the most basic operations that the data representation of a spreadsheet needs to implement*

```
Create(xy) -> s
Has-Value(s,xy) -> bool
Set-value(s,xy, v) -> s
Low(s) -> xy
High(s) -> xy
Inspect-value(s,xy) -> v
Remove(s, xy) -> s
```

## 2 Chosen Criteria

### 2.1 Time Complexity

The speed of specific operations is an important criteria. In some cases, it could decide whether a certain construction is feasible or not. Here 'speed' was define as the processing time needed at a realistic use case size of the instance in question. Hence, sometimes a bad complexity can be accepted as the typical realworld instance size happens to be in a very narrow bandwith. If possible, also the time complexity of individual operations will be asessed.

The following operations of interest were chosen:

- *Block Lookup* - Lookup values for a block of cells. This operation is called when the user scrolls or jumps to another place in the spreadsheet to update the graphical user interface. In a *Block lookup* a sequence of elements from both rows and columns is accessed.
- *Remove* - Removing elements of the spreadsheet.
- *Search* - Traversing the data structure to search for a user entered value in the spreadsheet elements.

### 2.2 Ease of Implementation

During OU2 discussion, most groups agreed that a lower complexity of the implementation is generally desirable as it decreases the susceptibility for bugs and code maintenance. How complex a certain datatype is to implement is not an objective measure. The presented evaluation is based on the authors current level of experience in C programming and data structures.

### 2.3 Memory efficency

It is understood from the problem definition that the used construction shall store just non-empty spreadsheet elements. Hence the judgement of memory efficency focuses mostly on the amount of memory used for dynamically manage the non-empty elements.

## 3 Chosen Datatypes

After a short literature study the following three datatypes were selected to be evaluated in more depth.

1. Array
2. Binary Tree
3. Directed Acyclic Graph

The array representation was chosen based on the description in the course litterature about implementation of a sparse matrix [3]. Binary tree representation was chosen after the defining time complexity to be one of the most critical criteria in the data representation: Binary trees in general offer very good time complexity. Finally, Directed Acyclic (DAG) Graphs were chosen upon several indications that that DAGs are a common way how industry standard spreadsheets are implemented [4][5] [6].

### 3.1 Construction of Datatypes

#### Array

The abstract datatype *array* ressembles the description of how a spreadsheet is defined. However, the static datatype array was not suited as an*efficient representation of a sparse matrix*. Janlert [3, pp 101-103] suggest in such a case to construct the array from a vector of tables. In C, the array is a physical datatype. Typical for the datatype array, it can not be resized during runtime. Hence this construction interpretes the requirement of a infinite large spreadsheet by chosing a large enough length for the row dimension. This seems to be an accepted choice: *Microsoft Excel*, one of the arguably most popular spreadsheet applications on the market, has for example a row/column limitation of 1'048'576/16'384 [2].

There is no physical datatype *table* in C. In this case construction from a dynamic datatype such as *linked list* seems reasonable. As there is also no physical datatype *list* in C, the construction could be done through the complex datatype *struct* linked with *pointers*. An example of the datatypes *list* and *table* implemented as described above is available on the course web page [1].

In this construction, columns are accessed directly through the array index and rows by looking up the value associated to the key that represents the row number (the course litterature uses the term *argument* instead of *key*[3]).

#### Binary trees

*Binary search trees* offer a good time complexity [3, pp. 286-291]. Therefore they were also considered for construction of a spreadsheet. Here, the row index was represented in one *binary tree*. The label of each node contains the column index and a pointer to another *binary tree* for representing the row indices. Hence, for each node of the column tree, there is an additional row *binary tree*. The nodes in the row trees have a label that contains both the row index and the actual value. In C, the construction of a binary tree is by using pointers to link *structs*. Also here, an implementation according to Janlert [3] was available through the course homepage [1].

#### Directed Acyclic Graph and Hash Table

When thinking about the *link* or *reference* feature that most spreadsheets offer, a table constructed from a graph should be an intersting possibility. A directed acyclic graph or *DAG*

is an n-linked structure. The nodes can be constructed as cells which are created dynamically when needed. In C, a struct would be the natural choice. Edges represent links in the spreadsheet. They are also created dynamically, in C as pointers.

To access individual cells, a hash table with a continuous numbering system, either along the rows or columns of the table can be implemented. Alternatively, the keys for the hash table can be the concatenation of the spreadsheet row and column coordinates. In C, a datatype *hash table* is not available by default and has to be constructed. The underlying datatype is a table as described earlier. For simplicity, *closed* with *linear probing* and a simple hash function such as *modulo* was assumed here[3, p. 277].

## 4 Evaluation of Data Representations

All data representations are evaluated seperately and later compared in the discussion section.

### 4.1 Time Complexity

#### Array

Looking up elements in a spreadsheet is always a composite of both row and column lookup. Here, the column lookup is of $O(1)$ based on array index access for the row while the row lookup is of $O(n)$, a consequence of constructing the *table* as a (*linked list*). Hence, *Block Lookup* in this representation is in the order of $O(n)$. *Removal* of spreadsheet elements is also $O(n)$. *Searching* for values is rather slow as the whole data structure has to be traversed: the time complexity is in the order of $O(n)$.

As explained earlier, it was delibarately chosen to have the vector of tables along the column indices such that one table contains all values for one column. This should help to speed up block wise operations along the column direction.

#### Binary Tree

Average value lookup in binary trees is in the order of $O(log(n))$. This implementation uses two binary search tree runs in sequence, first for the column and then for the row index. Hence the complexity for *Block Lookup* is still $O(log(n))$.

*Deletions* are of the same time complexity. This operation is usually implemented as a recursive function shifting the chain of childrens up one level. Hence, significant structural re-shuffeling is needed when nodes close to the root are deleted. *Searching* for values is in this case 'just' $O(n)$ as all nodes have to be traversed. This is because the order relation in the binary trees is based on their spreadsheet coordinates and not on the user values entered in spreadsheet elements.

#### Direced Acyclic Graph (DAG) with Hash Table

Lookup speeds in a *hashtable* are in the order of $O(1)$ which applies also to *Block Lookup* speed. The same is true for *Deletions*. *Searching* is on an avarage $O(n)$ as all stored values have to be looked up. The fact that the whole structure is implemented as a DAG has no inflence on the time complexity here.

### 4.2 Ease of Implementation

#### Array

Implementing a spreadsheet using *array* as data representation is straight forward as a spreadsheet basically is an array. In the present case the array needs to be constructed from a vector of tables, but these are still rather simple datatypes and they are all available through the course homepage.

Implementing the operations stated in listing 1 is also straight forward: column access is directly by indexing and the table uses a *lookup* operation. If the table is constructed from a linked list, this will result in a list traversal.

#### Binary Tree

Building the basic structure of the *binary tree* implementation is also rather simple in C by using *structs* and *pointers*. Also here, the underlying data types are available on the course homepage. However, the whole representation feels less intuitive because a tree does not really ressembles a spreadsheet. A binary tree has a rather rich set of operations. Mapping the spreadsheet interface to the binary tree is possible but a concise treatment of this issue would go beyond the aim of this report.

#### Directed Acyclic Graph (DAG) with Hash Table

The DAG purely as structure is simple to implement: The individual nodes are basically a dynamic resource (in C built from *structs*), that is allocated when needed and removed on deletion. The interface is more complicated as there are a range of operations to be implemented. However, in the beginning, they are not really needed as they are not crucial for the structural representation of a spreadsheet.

Implementing a hash table in C is less trivial and will require a fair amount of work. Also a number of design choices have to be take such as whether *open* or *closed* hashing shall be used. Here, the possibilites start to fan out such that a concise treatement of the topic would be beyond the scope of this report. Generally, it can be said that hash tables offer superior time complexity. But for real world application, they often have to be adapted for more balanced characteristics between time and space complecity. Hence this will increase the complexity for implementation.

### 4.3 Memory Efficency

#### Array

The described array representation for a spreadsheet is quite memory efficient. One dimension, in the current case the column index vector, is chosen to be static. while the tables for the row indices and the corresponding user values are created dynamically when needed during the user session. Despite one dimension beeing static, this implementation offers still a tremendous gain in memory efficiency.

#### Binary Tree

The whole representation is dynamically created during the user session and holds in both dimensions just elements for used row and column indices. As such, this representation is as memory efficient as an implementation can be.

**Directed Acyclic Graph (DAG) with Hash Table**

The amount of memory used depends on the chosen size for the hash table. If it is chosen large enough to cover most possible use cases, the overhead is rather big. A smaller, more memory efficient hash table will require more advanced algorithms, for example for *rehashing* in case the hashtable fills above a certain fill ratio. Alternatively, open hashing could be implemented, using a vector of tables as hash table. This would result in a partly static and a partly dynamic strucutre with somewhat better average memory efficency.

## 5 Discussion

### 5.1 Comparison of the different representations

Results from evaluating the three datatypes according to the defined criteria are summarized in table 1. There is no obvious clear winner. It seems like all three representations are feasible and could be chosen for a certain application envelope. Probably the situation would become clearar if the application envelope would be specified in more details than it currently is.

From the problem description, it was understood that this assignment was rather data structure centric and not so much functional. This was found after already having chosen the data type *DAG*, mostly because it was mentioned in litterture as a common representation in spreadsheets. However, he only interesting property of the *DAG* in respect to spreadsheets is the possibility linking spreadsheet elements and the powerful algorithms available to manage the linking structure. While this functionality seems very important in a 'real' spreadsheet application, it was not mentioned in the specifications for this assignment. As such, it turned out that it was more the hash table that made large impact for the third datatype.

The hashtable turned out to be some sort of magic bullet that performs well in almost every respect. However, when searching the net and litterature about hash tables, it was found that there are quite a number of details that need to be decided and optimized for each application case. The time complexity advantage is basically bought with a worse space complexity. In the most simple case, the hash table size is static and has to be decided prior to runtime. There are certainly implementations described that allow 're-hashing' to adjust the hash table size during run time, but then the complexity of implementation increases considerably.

After comparing the different representations, an interesting observation was that the array implementation with a vector of tables could basically serve as hash table when using 'open hashing'. Hence, there could be a possibility to start the project with a very simple implementation, and later, when time complexity turns out to be an issue, step up to a more complex implementation on the same code basis.

The binary tree implementation seems to perform somewhere in between the array and the DAG/hash table. However, it feels a bit like a black box as the tree-in-tree data structure does not ressemble much the underlying data. For all use cases that where envisoned, the structure seemed to work, but the implementation does not feel intuitive. Maybe it could be an intersting alternative to use a binary search tree in representation one instead of the table.

**Table 1** Evaluation of the data representations according to the chosen criteria

| | | Array | Binary Tree | DAG /Hash Table |
|---|---|---|---|---|
| Time Complexity | | | | |
| | Block Lookup | slow | fast | very fast |
| | Deletion | fast | fast | very fast |
| | Value Search | slow | slow | slow |
| Ease of Implementation | | simple | intermediate | rather advanced |
| Space Complexity | | quite good | very good | rather low |

### 5.2 Conclusions

In a situation where a whole spreadsheet application was to be developed, the choice would be to first choose a data representation that is easy to implement but has the potential to be upgraded for a better performing one. Hence, the array constructed from vector of tables would be such a representation.

Later, the access to the table could be improved by the use of hashing where the initial vector of tables would function as the open hash table.

The implementation of *DAG* together with a *hash table* is not really relevant here as the *DAG* itself does not contribute to the actual sctructure, it is just a set of nodes. However, as in most other data representations the actual user data would be stored in a separate data container, *DAGs* could become intersting from a functional point of view. It should be possible to implement the edges of a *DAG* structure by pointers between the data containers of another data representation.

As mentioned earlier, more details about the required functionality of the spreadsheet application would be needed to make a more consicise evaluation.

## References

[1] DV169VT16, cambro course homepage, 'datatypes'. `https://www8.cs.umu.se/kurser/5DV169/datatypes/index.html`, 2016. accessed: 2016-02-28.

[2] Microsoft Excel number of rows and columns. `https://en.wikipedia.org/wiki/Microsoft_Excel#Number_of_rows_and_columns`, 2016. accessed: 2016-02-27.

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

[4] StackExchange, what are the data structures behind a spreadsheet? `http://programmers.stackexchange.com/questions/219298/what-are-the-data-structures-behind-a-spreadsheet`, 2013. accessed: 2016-03-01.

[5] ActiveState, spreadsheet (python recipe). `http://code.activestate.com/recipes/355045-spreadsheet/`, 2004. accessed: 2016-03-01.

[6] Wikipedia, directed acyclic graph, 'applications'. `https://en.wikipedia.org/wiki/Directed_acyclic_graph`, 2016. accessed: 2016-03-01.