

Datavetenskapens byggstenar 7.5 p
DV160HT15

OU4 Data Representation

Submitted: 2016-03-03
Author: Lorenz Gerber (dv151gr@cs.umu.se)
Instructor: Lena Kallin Westin / Johan Eliasson / Emil Marklund / Lina Ögren

Contents

1	Introduction	2
1.1	Interpretation of the Problem Description	2
1.2	Typical Usecases of a Spreadsheet	2
1.3	The datatype Spreadsheet	2
2	Chosen Criteria	3
2.1	Time Complexity	3
2.2	Ease of Implementation	3
2.3	Memory efficiency	3
3	Chosen Datatypes	3
3.1	Construction of Datatypes	4
4	Evaluation of Data Representations	5
4.1	Time Complexity	5
4.2	Ease of Implementation	5
4.3	Memory Efficiency	6
5	Discussion	6
5.1	Comparison of the different representations	6
5.2	Conclusions	7
	References	7

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 then applied to judge the chosen 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 as a potentially infinite table. A requirement given in the description is that the implementation shall be more economic than a plain rectangular structure covering all non-empty table elements. From those descriptions it becomes obvious that a more concise description of problem is the *efficient implementation of a sparse matrix*.

1.2 Typical Usecases of a Spreadsheet

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

Another property of a spreadsheet program is that the data structure is represented in the graphical user interface. Due to the size, there will usually just one part of the data be visible, hence blockwise value lookup for scrolling over the data table is a very common operation. 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 member elements of the spreadsheet is common.

In a normal spreadsheet, 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 most likely influence how the user arranges data. Most likely, there will be more operations involving just a single columns compared to just a single row. If the data representation is asymmetrical in terms of structure or operations, this should be kept in mind for the implementation.

1.3 The datatype Spreadsheet

As a consequence of the above considerations the interface of the hypothetical datatype *spreadsheet* needs a basic set of operations shown in listing 1.

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

```
create(xy) -> s
HasValue(s, xy) -> bool
set(s, xy, v) -> s
inspect-value(s, xy) -> v
remove(s, xy) -> s
```

The most significant difference to a standard array as shown in Janlert [3, pp. 92-95] is the non existence of *low* or *high* operations. This is the consequence of the requirement that the spreadsheet is of infinite size.

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. Which 'speed' I would define 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 bandwidth. If possible, also the time complexity of individual operations will be assessed.

The chosen operations of interest are given below:

- Block Lookup - Get values for a block of cells. This operation will usually be applied when the user scrolls or jumps to another place in the spreadsheet. Block lookup differs from a normal lookup that both elements from both various rows and columns have to be looked up.
- Deletion - Deleting elements of the spreadsheet.
- Value Search - Traversing the data structure to search for a user entered value in the spreadsheet elements.

2.2 Ease of Implementation

During OU2 discussion, it was 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 experience level.

2.3 Memory efficiency

It is understood from the problem definition that the used construction shall store just non-empty spreadsheet elements. Hence the judgement of memory efficiency focuses on the amount of memory used for 'administration' of the non-empty elements.

3 Chosen Datatypes

After a short literature study three datatypes were chosen to be evaluated in more depth.

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

Array was chosen as there was a description in the course literature about implementation of a sparse matrix [3]. The binary tree was chosen after deciding that time complexity

was the maybe most critical criteria in the data representation and that binary search trees offer a very good time complexity. Finally, Directed Acyclic (DAG) Graphs were chosen after having found indications on the web, 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* resembles the description of how a spreadsheet is defined. However, when looking at the problem description, it became obvious that the physical datatype array was not suited as representation 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 choosing 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 to be 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 literature uses the term *argument* instead of *key*[3]).

Binary trees

Binary search trees offer a good time complexity. Therefore they were also considered for construction of a table trees. 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*. 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 make heavy use upon, a table constructed from a graph should be an interesting possibility. A directed acyclic graph or *DAG* is an n-linked structure. The nodes can be constructed as cells which are created dynamically on demand. 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 used. Alternatively, the keys for the hash table can be the concatenation of the spreadsheet table 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, *open hashing* with *linear probing* and a simple hash function such as *modulo* was chosen [3, p. 277].

4 Evaluation of Data Representations

All data representations are evaluated separately 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 the chosen construction of the *table (linked list)*. Hence, *Block Lookup* in this representation is in the order of $O(n)$. *Deletion* of spreadsheet elements is also of $O(n)$. *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 deliberately chosen to have the vector of tables along the column indices such that one table contains all values for one column.

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 also $O(\log(n))$.

Deletions are of the same time complexity, however, significant structural reshuffling 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 value 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. This is a consequence of the fact that we are hashing for spreadsheet coordinates and not spreadsheet element user values. 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 on 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 rather simple as the underlying data types are available on the course homepage. However, the whole representation feels less intuitive because a tree does not really resemble a spreadsheet. A binary tree has a very rich set of operations. Mapping the spreadsheet interface to the binary tree is certainly possible but a concise treatment of this issue would go beyond the current 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 that is allocated when needed and removed on deletion. However, implementing a hash table in C is certainly not trivial and will require a fair amount of work. Also a number of design choices have to be taken such as whether *open* or *closed* hashing shall be used. Here, the possibilities start to fan out such that a concise treatment 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 complexity. Hence this will increase the complexity for implementation.

4.3 Memory Efficiency

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 on demand during the user session.

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 very memory efficient.

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 current hashtable fills above a certain fill ratio.

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 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 clearer if the application envelope would be specified in more details than it currently is.

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

I interpreted this exercise to be rather data structure centric and not so much functional. I realized this after I chose the data type DAG, mostly because I found it mentioned in literature. It seems like the only interesting property of the DAG in the spreadsheet respect is for linking spreadsheet elements. While this functionality is in my experience very important in a ‘real’ spreadsheet application, it was not really mentioned in the specifications for this exercise. 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 literature 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 pre-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 increased 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 basically 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 actual data structure does not resemble much the underlying data. For all use cases that were envisioned, the structure seems to work, but the implementation does not feel to be very intuitive.

5.2 Conclusions

As mentioned earlier, more details about the required functionality of the spreadsheet application would be needed to make a more concise evaluation. My first impression was that I would probably start with a very simplistic implementation using the *array* constructed through a *vector* of *tables*. However, after I’ve seen the code snippet in reference [5], I am in favour of the hash table combined with a DAG. I was impressed by the simplicity of this implementation. In fact, it seems that the initial work would concentrate mostly on implementing the hash table with links to structs that later will serve as nodes in a DAG. A simplistic hash table can be coded rather quick. When developing software, I prefer to have first a functioning framework of the whole application with simplistic versions of each part (could be hardcoded or dummy) already in place instead of perfecting individual parts prior to putting pieces together.

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.