Umeå University
Institution för Datavetenskap

Datavetenskapens byggstenar 7.5 p
DV160HT15

OU4 Data Representation

Submitted 2016-03-03
Author: Lorenz Gerber (dv15lgr@cs.umu.se)
Instructor: Lena Kallin Westin / Johan Eliasson / Emil Marklund / Lina Ögren

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 then applied to judge the chosen data representations.

It was chosen to structure the report into five parts: First an *introduction* where the aim of the work is presented. Second, the *interpretation of problem description* including also a short reference to the chosen data representations and criterias. Third comes an *construction of the data types for each chosen data representation*. Then the *evaluation according the chosen criteria*. And finally *summarizing discussing* where also the decision for the most suitable data representation is revealed.

2 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 becomes obvious that a more concise description of problem is the *efficient implementation of a sparse matrix*.

2.1 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 rectanglular 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 memeber elements of the spreadsheet is common.

2.2 Chosen Criteria

Time Complexity

The speed of specific operations is an important criteria. In some cases, it could decide whether a ceratin construction is feasible at or not. Whith '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 bandwith. If possible, also the time complexity of individual operations will be asessed.

The chosen operations of interest are given below:

- Block Lookup - Get values for a block of cells. This operation will usally be applied

when the user scrolls or jumps to another place in the spreadsheet. If row/column wise scrolling is assumed, typicall lookup sizes would be around 10 to 50.

- Deletion - Deleting elements of the spreadsheet.
- Element Search - Traversing the data structure to search for a value

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. Below are indicated some features/operations for which the ease of implementation will be judged and compared across the different table constructions.

- Basic Structure - The data container as such
- Sort - Sorting larger blocks of non-empty elements

Memory efficency

It is understood from the problem definition that the used construction shall just store non-empty spreadsheet elements. Hence the judgement of memory efficency focuses on the amount of memory used for 'administratiion' of the non-empty elements, such as hash tables and non value bearing data structure.

2.3 Chosen Datatypes

The abstract datatypes that were chosen to be evaluated are:

- Array
- Binary Tree
- Directed Acyclic Graph

3 Construction of Datatypes

3.1 Array

The abstract datatype *array* ressembles 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 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 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, rows are accessed directly through the array index and columns by looking up the value associated to the key that represents the column number (the course litterature uses the word *argument* instead of *key*[3]).

3.2 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 row index and a pointer to another *binary tree* for representing the column indices. Hence, for each node of the row tree, there is an additional column *binary tree*. The nodes in the column trees have a label that contains both the column 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 was available through the course homepage [1].

3.3 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 becomes 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 on demand. In C, a struct would be the natural choice. Edges represent links in the spreadsheet. They are also created dynamically.

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 number to put in 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 seperately and later compared in the discussion section.

4.1 Time Complexity

Array

Looking up values in a spreadsheet is always a composite of both row and column lookup. Here, the column lookup is of O(1) while the column lookup is of O(n). Hence, *Block Lookup* in this representation is in the order of O(n) because of the *table*. *Deletion* of spreadsheet elements also of O(n). *Searching* for values is rather slow as the whole data structure has to be traversed, hence, the time complexity is in the order of O(n).

Binary Tree

Average value lookup in binary trees is in the order of O(log(n)), which is also the complexity for *Block Lookup*. *Deletions* are of the same time complexity, however, significant structural reshuffeling is needed when nodes close to the root have to be deleted. *Searching*

for values is 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 coordinate values and not on the user value entered in a spreadsheet cell.

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 consquence 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 current 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.

Binary Tree

Building the basic structure of the *binary tree* implementation is also 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 ressembles a spreadsheet.

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.

5 Discussion

And our results looked like this...

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.