Evaluation of a data type in a Manufacturing Execution System

Peter Berglund
Abstract

Evaluation of a data type in a Manufacturing Execution System

Peter Berglund

Sandvik Coromant is currently using a Manufacturing Execution System named GSS-II used for planning, preparation, and production in the process of manufacturing hard metal inserts. The system is using an own developed data type, ParameterStack, which is used for holding the data from the database to the client. ParameterStack was developed in the 90’s and has not been reconsidered since then. This thesis evaluates the data type and investigates if it is possible to implement a data type with better complexity.

The ParameterStack was analyzed and parts that should be further investigated were identified. After that, the big O-notation for the selected parts was determined both in theory and by writing a program. This program was used to compare the data types. The complexity was determined for ParameterStack’s Add, AddFirst, Get, Delete, DeleteAll and update functions. A scenario where the client is starting up the system was also measured.

The result shows that the operations Add, Get, Delete and Update have a complexity of $O(\log(n))$ while the operations AddFirst and DeleteAll have a complexity of $O(n)$. The conclusion was to replace a binary search tree in the algorithm with a hash table, namely CMap, which has a better complexity and is faster to use. The tests show that some operations will take half the time with the new data type.
Contents

1 Introduction 2

2 Background 4
2.1 Storing data in ParameterStack 4
2.2 ParameterStack’s methods 5

3 Analysis of ParameterStack 6
3.1 Methodology for the analysis 6
  3.1.1 Distribution of ParameterStack’s methods 6
  3.1.2 Overview over function calls made in GSS-II 6
  3.1.3 Data in ParameterStack 7
  3.1.4 Generating test data and scenarios 7
3.2 Results from the analysis 7
  3.2.1 Distribution in GSS-II and data volume in ParameterStack 7
  3.2.2 Generated test data 8

4 Measuring Performance 9
4.1 The \(O\) - notation for ParameterStack in theory 9
  4.1.1 Add 9
  4.1.2 Get 10
  4.1.3 AddFirst 11
  4.1.4 DeleteAll 12
  4.1.5 Update 13
4.2 Measurements 13
4.3 Result 14

5 Review of Data Structures 18
5.1 A Data Structure 18
5.2 Analysis of Data Structures 18

6 Evaluation of new Data Structures 19
6.1 Theory 19
6.2 New data structures 20
6.3 Results 20

7 Conclusion and Future Work 28

Bibliography 30

Appendix A Table of distribution 31
Chapter 1

Introduction

Sandvik Coromant is currently using a MES (Manufacturing Execution System)[4] named GSS-II, developed in C++. The system was constructed in the nineties and is currently used all over the world on Sandvik Coromant’s production units. GSS-II is still administered and improved by Sandvik IT (Sandvik’s own IT company).

This system is an order handling system, which supports planning, preparation, and production in the process of manufacturing hard metal inserts. The system enables the staff to plan and carry out their work in an efficient manner. GSS-II stores information about an order, thereby enabling detailed tracking of the order, down to individual operations. The general functionality of the system is: order handling, operation handling, report handling and message handling.

GSS-II substantially uses a data type, named ParameterStack, which inherits from Microsoft standard class CObArray\(^1\). ParameterStack is used for holding data from the database and is partly used by the client applications that can call the GSS-II server functions, but also as input and output variables when GSS-II server classes call its own public functions.

The system is a client/server solution where data is sent from a server to a client (see Figure 1.1). When a client wants to do a task, in-parameters are sent from the client to the server. These in-parameters are stored in a ParameterStack. After this, the servers functions’ will perform the requesting task by getting from or adding data to a database and then do the task. The output from the servers’ function is a ParameterStack, which will be converted and sent through a network socket, and then converted back to a ParameterStack in the client.

The data type is as mentioned old and it might be possible to make an improvement on it. The goal of this thesis is to evaluate the performance of the data type and see if it still is a good

---

implementation after so many years. Some questions that will be answered in the process are:

- Is ParameterStack a good solution for holding the data in GSS-II?
- What difference will it make to substitute the data type with one with better performance?
- What improvements can be made to optimize the data type?

The questions will be answered in two steps. First an analysis how ParameterStack is currently used will be performed, i.e. by creating test data that can be used to measure the complexity on some of the main operations for the data type. The analysis of the data type will show which main operations are appropriate to measure the complexity on. Secondly, there will be an evaluation of the complexity for the chosen operations. The last part is then to evaluate new data structures with the same interface as ParameterStack with improved complexity.

The ParameterStack is described in Chapter 2. In Chapter 3, an analysis of ParameterStack and the test data used when measuring performance are selected. Chapter 4 describes the complexity of selected operations of ParameterStack while in Chapter 5 a new data structure is selected and compared to the existing one for ParameterStack.

Finally, a review of data structures is described in Chapter 6 and the conclusion and future work in Chapter 7.
Chapter 2

Background

The name ParameterStack can be a bit misleading since this is not an actual stack. The datatype is a combination between an std::map and a CObArray. The std::map is implemented as a red-black binary search tree [2] while a CObArray is similar to C arrays, except a CObArray can shrink and grow dynamically [7].

![Binary search tree connected with an array](image)

**Figure 2.1**: An overview of how the binary search tree is connected with an array.

2.1 Storing data in ParameterStack

ParameterStack saves data in two steps. First the name, type and value of an item is stored in a struct (shown in the right column in figure 2.2), and a pointer to the struct is stored in a CObArray. The name of the item is then stored together with the index in the CObArray where the pointer is stored (see Figure 2.1) and the item name will be the key. Elements with the same name can be separated by adding an extra parameter to the end, for example, OrderName_1_1, OrderName_1_2, Ordername_1_3, etc.

Maps store elements with a key and a mapped value. The key is usually used to find a element and the mapped value stores the content associated with the key.
An example on this is when we want to store all orders in GSS-II in a ParameterStack. In this example the information about an order will contain the number of the order (OrderNumber), the name of the order (OrderName) and the status of the order (OrderStatus).

At element 0 in the CObArray the order number for the first order will be stored. The name will become OrderNumber_1 since this is the first order number. That name is then stored on element 0 in the binary search tree. Next, the actual name on the order will be stored in element 1 in the array (‘hard steel batch’) and the tree adds OrderName_1 into element 1 for the tree. Same goes for order status in element 2.

Now all information about the first order has been stored and we continue with adding the order number for the second order in element 3, but since this is order number 2, the suffix becomes _2 instead (see Figure 2.2).

2.2 ParameterStack’s methods

First of all, there are several ways to add an item to the ParameterStack. An item can be added first, last or at a given index in the stack. There is also possible to add a whole block of data. Items that can be added are integers, long, const CString, double and const Decimal.

Secondly, there is a number of methods to get data from the ParameterStack. An item can be retrieved by searching for the item’s name. It is also possible to get a data block starting from a specific index, and also to get the name of a specific data block. A data block contains several elements from a ParameterStack. There is also a method to update an Item in a ParameterStack.

There are also methods to check if an item is in the ParameterStack, in the ParameterStack at a specific index or in the ParameterStack with a specific index and with a specific value. Items can be removed in three ways, one by deleting an item, delete an item at a specific row or delete a whole data block. Delete a whole stack is also an option.
Chapter 3

Analysis of ParameterStack

ParameterStack is used in different servers and clients. This analysis will cover the usage of a particular server, named ‘general server’. This server is the main server the operator connects to from a computer (client) when he or she wants to do a task, process orders for example.

3.1 Methodology for the analysis

The analysis of ParameterStack is divided into four parts:

1. Get an overview over the distribution of ParameterStack’s methods.
2. Get an overview over which functions are called in GSS-II during 24 hours.
3. Check what type and quantity of data ParameterStack usually holds.
4. Generate test data and several scenarios from the results in steps 1-3.

3.1.1 Distribution of ParameterStack’s methods

The main reason to investigate the distribution of ParameterStack’s methods is to choose which methods to run individual tests on. This will also help understanding how the methods in ParameterStack are used. To achieve this, a plugin to Visual Studio[5], named Visual Assist[10], was used. Visual Assist provides the feature to find all references to a function call and copy them. The result from this was copied into an Excel document and structured to display all files calling each method and all functions that call the methods. From this it is possible to count how many places in GeneralServer the method is used.

3.1.2 Overview over function calls made in GSS-II

The current system logs activities during run time. One of the things it logs is when a function is called and when it returns from it. This makes it easy to get an overview over which functions are called during a 24 hour range. The log file contains other information as well which means the function calls need to be extracted. A method to do this is to use findstr\(^1\) in the command

\(^1\)findstr is a Command-line utility to search for a specific text in a text file and then extract it from the text file. By using the command findstr "Called" /logfile.txt > results.out we search for the string "Called" in the file logfile.txt and print the result in result.out.
prompt to filter out every called function. The resulting line looks similar to this:

```
2017-04-23 23:50:45.603 INFO <General Server><CL_123> Called OrderNo=123121
```

where <CL_123> is the called function. The function name is then cut out and placed into an Excel document. In the Excel document it is possible to count how many times function is called and thereby get the distribution of function calls and how often they are called during a 24-hour period.

### 3.1.3 Data in ParameterStack

Something that is not logged in the system is the content of the ParameterStack. The client calls a specific function with a ParameterStack and what type of operation the server shall do.

The function then calls the function responsible for that operation. The function called by the client also sends back the result to the client in form of a ParameterStack. So to get the data in the ParameterStack a log file was created and we can put the value of the ParameterStack in it.

Sandvik has a test system for GSS-II which can be used to implement the log file in. Now it is time to run the client for the test system and see the outcome in the log file.

### 3.1.4 Generating test data and scenarios

When generating test data some things need to be considered. The test data for measuring each method in ParameterStack should be similar to what is usually put in the stack. This means that if a stack usually holds 1000 to 2000 elements, the measurements should be between that. Scenarios should reflect something that actually can happen in the system. This can be, for example, to add new orders or list all orders in the production.

To generate scenarios we need to look into which function calls are frequently made and what those functions do. We also need to see what kind of data and the number of elements ParameterStack has in that situation. The two previous sections show frequently used functions and the result from the analysis can be used to make test scenarios. The content in the created log file describes the number of elements in the ParameterStacks and what each ParameterStack contains.

### 3.2 Results from the analysis

#### 3.2.1 Distribution in GSS-II and data volume in ParameterStack

The intuition before the analysis was that Add and Get are the most common methods seen in the code base. Figure A.2 displays the result from the distribution analysis of ParameterStacks’ methods. The methods of ParameterStack that are most frequently seen in GSS-II:s methods are Add (6411 times), Get (4093 times), DeleteAll (197), AddSubItem (102 times), Update (92 times), GetSubItem (63 times), Delete (60 times), TranslateParameterStackToString (60 times), MoveNext (44 times), AddFirst (43 times). From these methods are Add, AddFirst, Get, Delete, DeleteAll and Update chosen to be tested separately and determine the complexity of each of them.
In next part of the analysis, we get all function calls on a 24-hour period, described in Chapter 3.1.3. After evaluating which methods in the ParameterStack each function is calling, we see that Add, AddFirst, Delete, Deleteall and Update are the most commonly used methods.

3.2.2 Generated test data

Some conclusions can now be drawn from the result of the analysis and can work as a support when generating test data. First, we can see that we should focus on testing the performance on Add, AddFirst, AddSubItem, Get, DeleteAll, and Update separately. The average stack is between 1 to 40,000 elements, which means that the test data should be in that range. When testing these functions the time it takes to do each of the chosen operations on a stack with elements between 1 and 40,000 elements will be tested with a step on 10 in order to get sufficient number of measuring points. This is for example to add an element into a stack with first 1 element, then to a stack with 10 elements, then 20 elements and continue until the stack reaches 40,000 elements.

Scenarios to be tested are to add 40,000 elements, get one element from a stack with 40,000 elements, delete all elements and delete one element from a stack with 40,000 elements. The reason to test these individually is because they are commonly used and the most interesting methods to evaluate for big stacks.

We also test a system-specific task. A scenario where many ParameterStacks are used and is easy to compare against are when the client is starting the system. The log file shows exactly which functions are called when starting the system, by taking the time it takes for each of the test systems functions to perform ParameterStack’s methods with the corresponding stack size.
Chapter 4

Measuring Performance

The efficiency of an algorithm can be compared by measuring how the running time for the algorithm can grow. An algorithm that is asymptotically more efficient will be the best choice, if the input is not too small. [9] There are several notations used to describe the asymptotic running time of an algorithm and the one chosen in this analysis is the $O$-notation. The $O$-notation describes how the running time can grow as most.

4.1 The $O$-notation for ParameterStack in theory

The $O$-notation for each method is obtained by determining the $O$-notation for every operation in the methods’ source code. The $O$-notations are then compared to see which has the most effect for the running time and that one will reflect the $O$-notation for the method.

4.1.1 Add

The code analysis for Add is presented here. A pointer is created and holds the information we want to store (line 4 in Figure 4.1). Then the information is stored in the structure (line 5-8 in Figure 4.1). All of them take constant time since we only create and insert one element each time.

The structure is then inserted in a COBArray and the position in the COBArray is returned. This operation is constant-time[7]. The index is paired with the name of the item and inserted into a map (line 10-11 in Figure 4.1). This operation is logarithmic[2]. Lastly the function is returning an OK which is a constant time operation.

The O-notation for this method is $O(log(n))$. 

```c++
int ParameterStack::Add(const CString &inputX, int inputY)
{
    Item1 *item = new Item1();
    item->Id = InputX;
    item->type = _T(1);
    item->value.Format(_T("%d"), inputY);
    index = CObArray::Add(item); // O(1)
    m_map.insert(m_map::value_type(inputX, index)); // O(log(n))
    return OK; // O(1)
}

/* The O-notation is: O (log(n)) */
```

Figure 4.1: The O-notation for adding an element to the ParameterStack is $O(\log(n))$

### 4.1.2 Get

The get function starts with declaring a pointer. This pointer is used for storing the information from the specific item requested. A lookup if this item exists is done on line 5 in Figure 4.2 and the LookupItem-function is $O(\log(n))$[2]. If the item exists, the value of the Item is stored in *inputX and an OK is return, which both operations are constant. Else an error is returned, also in $O(1)$.

The $O$-notation for this method is $O(\log(n))$. 

---

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

/* The O-notation is: O (log(n)) */
4.1.3 AddFirst

AddFirst is a longer function. It starts with declarations of some variables and by storing the information in a structure (line 3 to 9 in Figure 4.3). Then this structure is added in the CObArray at index 0. All these operations are constant time. After that, the item’s name and index are stored in a map, with $O(\log(n))$ and the name together with the index are stored at the beginning of the map, $O(1)[2]$.

Since the item is added first, the map needs to be iterated from the first to the last element. This will take as long time as there are elements, hence $O(n)$ where $n$ is the number of elements in the ParameterStack. $O(n)$ beats $O(\log(n))$ and $O(1)$ which means the final $O$-notation is $O(n)$.

```cpp
int ParameterStack::AddFirst(const CString &inputX, const CString &inputY)
{
    CString temp;
    int index = 0;
    ItemI *item = new ItemI();
    item->id = inputX;
    item->type = _T(s);
    item->value = inputY;

    CObArray::InsertAt(index, item);          // O(1)
    map::insert(map::value_type(inputX, index)); // O(log(n))

    map::iterator iter;
    CString tmp;
    iter = map.begin();                      // O(1)
    while (iter != map.end())                // O(n)
    {
        tmp = (*iter).first;                   // O(1)
        if (tmp.CompareNoCase(inputX) == 0)    // O(n)
            { (*iter).second = index;         // O(1) } 
        else
            { (*iter).second += 1;            // O(1) }
            iter = map.upper_bound(tmp);      // O(1) 
    }

    return OK;                               // O(1)

    /* The O-notation is: O(n) */
}
```

Figure 4.3: The $O$-notation for adding an element first in the ParameterStack is $O(n)$
4.1.4 DeleteAll

DeleteAll has as Addfirst the $O$-notation $O(n)$ where $n$ is the number of elements in the ParameterStack. This is because we need to iterate through all elements and delete them.

```cpp
int C_ParameterStack::DeleteAll(BOOL inputX)
{
    itemI *item;
    int i;
    for (i=0; i<GetSize(); i++) // O(n)
    {
        item = (itemI *) GetAt(i); // O(1)
        if (item != NULL) // O(1)
        {
            delete item; // O(1)
            item = NULL; // O(1)
        }
    }
    RemoveAll(); // O(n)
    if (!inputX) // O(1)
    {
        map::clear(); // O(n)
    }
    parameterIndex = 0; // O(1)
    return OK; // O(1)
}
/* The $O$ - notation is : $O(n)$ */
```

Figure 4.4: The $O$-notation for deleting all elements to the ParameterStack is $O(n)$
4.1.5 Update

Update uses LookupItemIndex (line 6) which is $O(\log(n))$. The rest are $O(1)$ which gives the $O$ - notation for the whole function $O(\log(n))$ where $n$ is the number of elements in the ParameterStack.

```c
int C_ParameterStack::Update(const CString &inputX, int inputY)
{
    int index;
    itemI *item = new itemI();
    if ((index = LookupItemIndex(inputX)) >= 0) // O(log(n))
    {
        itemI *item;
        item->id = inputX;
        item->type = _T(1);
        item->value.Format(_T("%d"), inputY);
        oldItem = (itemI *)CObArray::GetAt(index); // O(1)
        if (oldItem != NULL) // O(1)
        {
            delete oldItem; // O(1)
        }
        CObArray::SetAt(index, item); // O(1)
        return OK; // O(1)
    }
    else
    {
        if (item != NULL) // O(1)
        {
            delete item; // O(1)
            return Couldn'tFindItem; // O(1)
        }
    }
} /* The O - notation is: O (log(n)) */
```

Figure 4.5: The $O$-notation for updating an element in the ParameterStack is $O(\log(n))$

4.2 Measurements

The most common functionalities in ParameterStack were identified in the analysis and then evaluated in theory. Next step is to measure the complexity and compare it with the result we got in theory. These measurements are also going to be the guideline when looking into a new possible way to implement ParameterStack.

When comparing each method, we want to compare the time\(^1\) it takes in a certain range. The analysis gives a lower limit on 1 and an upper limit on 40,000, which means that is the range to measure the time on. This is implemented with a for-loop taking the time it takes for inserting 1 element into a ParameterStack with first 100 elements and increasing with 100 each time until it reaches 40,000. This gives us 400 measuring points. The measuring points are then inserted into an excel file where a graph is made. These graphs give a visualization on which type of complexity each method has.

\(^1\)Measuring the time is implemented with QueryPerformanceCounter. QueryPerformanceCounter retrieves a time stamp from the performance counter. The time stamp has a high resolution ($< 1 \text{us}$) [8]
For the scenarios are the time for running the methods Add, Get, Delete and Update on a stack with 40000 elements measured. There is also a scenario where the time it takes to perform a number of Add and Get operations similar to when one of the system’s client is starting up the program is measured. To be more specific, this is a measurement on how much time the system spends on doing operations on the ParameterStack when starting up the client.

4.3 Result

The empirical runtime complexity of the selected methods corresponds to the result we got in theory. Add, get, delete and update have logarithmic growth (see figure 4.1) while Addfirst and DeleteAll have linear growth (see figure 4.2). It takes 0.73 seconds to add 40,000 elements to a ParameterStack, 0.000007 seconds to get an element in a ParameterStack with 40000 elements, 0.000014 seconds to delete an element in a ParameterStack with 40,000 elements, 0.047 seconds to delete 40,000 elements in a ParameterStack and 0.23 seconds to perform the simulated startup of OPI (see figure 4.3).
Figure 4.6: Runtime when measuring the time it takes to add, get, delete, or update an element to a ParameterStack with 1 - 40000 elements.
Figure 4.7: Time it takes to delete all elements or add an element first to a ParameterStack with 1 - 40000 elements.
Figure 4.8: Time it takes to add or delete 40000 elements to a ParameterStack. Also the time it takes to get or delete an element in a ParameterStack with 40000 elements and the time it takes to start the OPI application.
Chapter 5

Review of Data Structures

Before describing data structures, the difference between a data type and an abstract data type needs to be defined. In programming languages, a data type is the set of values that the data type may have, for example a Boolean may have the values true or false. The basic data types vary between different programming languages. An abstract data type can be seen as a mathematical model with various defined operations. These operations can for example be to insert a variable or delete a variable in the abstract data type[3].

5.1 A Data Structure

A data structure is the implementation of an abstract data type. The data structure is a collection of variables, sometimes with different data types, and with several operations defined for it. One of the most basic data structures is an array. The array has one or several cells, which is the building block of a data structure. A value is stored in each cell and different operations handle the variables, for example insert or access a value to or from a cell[3].

5.2 Analysis of Data Structures

The running time of an algorithm is influenced by many different factors, such as the quality of the code generated by the compiler or the nature and speed of instructions on the computer executing it. When analyzing data structures, analyzing the time complexity is often used[3].

Big-Oh notation is used to describe the asymptotic upper bound. The asymptotic upper bound can be described as the maximum function growth. If you want to search for an element in the array you need to iterate every element until reaching the right one, which will take n number of steps where n is the number of elements in the array and the big-Oh notation will be O(n)[11].

The big-Oh notation can be used to compare different data structures operation, for example inserting an element into the end of an array and a stack. The big Oh notation for insertion in an array is O(n) since the element is placed at the end of the array and every element needs to be traversed before insertion. In a stack the element is placed on the top, meaning that nothing needs to be traversed. This means that the time complexity is constant-time, described as O(1). The time complexity when doing insertion into a stack is better compared to doing insertion into the end of an array. Figure A.2 in appendix A describes some data structures and the complexity of the basic operations for each data structure.
Chapter 6

Evaluation of new Data Structures

6.1 Theory

The complexity of the chosen methods in the current ParameterStack is:

- Add: $\log(n)$
- AddFirst: $n$
- Get: $\log(n)$
- Delete: $\log(n)$
- DeleteAll: $n$
- Update: $\log(n)$

Apart from AddFirst and DeleteAll, the complexity for the rest of the chosen methods is $\log(n)$, which is acceptable. As we know by now, the data type is divided into a COBarray and a map. The benefit having a COBarray instead of a normal array is the ability to insert and delete elements with complexity $O(1)$ instead of $O(n)$. The complexity for accessing an element is $O(1)$ for both a COBarray and a normal array. Insert, delete and access elements in the COBarray are the only operations needed, hence the complexity cannot be improved by replacing the COBarray since those operations have complexity $O(1)$. The map, on the other hand, uses the operations insert, search and delete. These are all performed with a complexity of $O(\log(n))$. The only data type with better complexity for these operations is a hash table and it is most probably the best data type to replace the map with.

There are a couple of ways to implement a hash table; either with an own implementation or use a hash table only defined for C++. The best option here is to use the standard hash tables, since it is already implemented, which reduces the risk of bugs. One standard hash table is STL’s unordered_map and has the same methods as the map. It is possible to change the declaration and leave the code as it is. The key for an unordered_map must be of a data type supported in STL and MFC’s data type CString is not. This means CString cannot be the key in the hash table. Fortunately, Microsoft Foundation Class has a hash table named CMap that allows the key to be a CString. The final decision is to change map to CMap.
6.2 New data structures

As previously stated, CMap is a hash table that supports CString as the key. It is required to include the afxtempl.h, which is part of the Microsoft Foundation Class Library, in order to use CString. The new data type with CMap is created as:

\[
\text{Typedef CMap\langle CString, LPCTSTR, int, int \rangle C\_ItemMap;}
\]

declaration means that CString and LPCTSTR are the key part and int, int are the value types. When CMap is created, the standard is to allocate 16 buckets in the table. This means there will collision when we reach over 16 elements\(^1\), and it will take more time to insert an element into the table [6]. There is a way to avoid the collision: using the InitHashTable – function. This function defines the number of elements in the hash table. The possibility for collisions will decrease if the initialization is 20 percent more than the actual number of elements and a prime number. So if 100 elements are expected to be in the hash table, the initiation will be initHashTable(121). 121 is around 20 percent more than 100 and a prime number.

To add an element to the table SetAt(X,Y) is used. X, in this case, is the key and Y is the value. To get an element from the table, the function lookup(X,Y) is used. X and Y are the key and the retrieved value. To remove an element is removekey(X) used and it removes the element at key X [6].

Since the method calls are not equivalent to STL’s map, some changes needed to be done in the ParameterStack. Most of the changes are in the add, get, and delete methods. Some more tests are created when map is replaced with CMap, to see if the new implementation is working as it should and lastly the complexity tests are performed on the new ParameterStack.

6.3 Results

The results from measuring each method are presented with the result for the original ParameterStack, CMAP when there is no initialization, and CMap with an initialization of 48001 elements.

The add-method is first analyzed (see figure 5.1). As in theory, the complexity is linear when not using initiation. It takes longer time compared to the original ParameterStack when the stack is over 26000 elements and using initiation is faster than the original ParameterStack.

In addFirst (figure 5.2), get-method (figure 5.3) and delete all, CMAP is faster than the original ParameterStack and it does not matter if we use initiation or not.

Delete (figure 5.4) and update (figure 5.6) look similar. It is faster to use the original ParameterStack when we have over 14000 elements.

The scenarios in figure 5.7 confirm what we already know, it does not matter using initiation on Get and delete all elements but it does on delete an element and add.

The last scenario is when starting the OPI (see figure 5.8). When using CMAP the startup time is half the time it takes with the original ParameterStack. We also see that it does not really matter using initiation or not.

\(^{1}\)There might still be collisions with fewer than 16 elements
Figure 6.1: Comparing the add-method with the original ParameterStack and when using an initiated and uninitiated hash table. The measurements are done on a ParameterStack with 1 to 40000 elements.
Figure 6.2: Comparing the addFirst-method with the original ParameterStack and when using an initiated and uninitiated hash table. The measurements are done on a ParameterStack with 1 to 40000 elements. The initiated and uninitiated hash table shows the same result.
Figure 6.3: Comparing the get-method with the original ParameterStack and when using an initiated and uninitiated hash table. The measurements are done on a ParameterStack with 1 to 40000 elements.
Figure 6.4: Comparing the delete-method with the original ParameterStack and when using an initiated and uninitiated hash table. The measurements are done on a ParameterStack with 1 to 40000 elements.
Figure 6.5: Comparing the deleteAll-method with the original ParameterStack and when using an initiated and uninitiated hash table. The measurements are done on a ParameterStack with 1 to 40000 elements. The initiated and uninitiated hash table shows the same result.
Figure 6.6: Comparing the update-method with the original ParameterStack and when using an initiated and uninitiated hash table. The measurements are done on a ParameterStack with 1 to 40000 elements.
Figure 6.7: Time it takes to add or delete 40000 elements to a ParameterStack and the time it takes to get or delete an element in a ParameterStack with 40000 elements. The comparison is between the original ParameterStack, a ParameterStack with an initiated hash table and a ParameterStack without a initiated hash table.

Figure 6.8: Time it takes to start OPI with the origin ParameterStack, a ParameterStack with an initiated hash table and a ParameterStack without an initiated hash table.
Chapter 7

Conclusion and Future Work

The final result from the evaluation of ParameterStack shows that it is possible to have a ParameterStack with better complexity if the map is changed to a hash table. MFC’s hash table cmap is chosen since it can keep CString as the key, compared to STL’s unordered_map that cannot.

The first question when starting this thesis was: is ParameterStack a good data type to use in the manufacturing execution system? Considering that most of the examined methods have a complexity of $O(\log(n))$ shows that it is a reasonably good implementation as it is now. Also, the current implementation of ParameterStack does not seem to slow down the system. The difference in changing to cmap is that certain operations will take shorter time to perform. That is the main reason to change to the new implementation and answer the question stated in the beginning: what difference will it make to rewrite the data type.

The last question is about what improvements can be made to optimize the data type. As stated, the big improvement is to change to a hash table. The tests in the scenarios show that the time it takes to perform some operations will be halved. Now consider that ParameterStack is used a lot in the system and even though the decrease in time is small for one operation it might make a difference in the longer run.

The big issue when using cmap is the memory usage. To avoid collisions there needs to be reserved buckets in the hash table. As it is for the test in this thesis, 48001 buckets are reserved since it is approximately the max number of elements put into a ParameterStack. This is the max number, but the analysis shows that a stack normally contains 2-400 elements. This means that we will allocate a lot of extra memory, which might even lead to a slow down in the system. Also the scenario when OPI is starting shows that it does not make a big difference if we do the startup with reserved buckets or without. I do not think it is necessary to allocate memory, especially as much as for 40000 buckets, since it is rare to have 40000 elements in the ParameterStack.

There is some future work before the new ParameterStack can be implemented with an bucket initiation. A suggestion is to make a performance analysis on the whole system when the new ParameterStack is implemented with 48001 elements already reserved in the hash table. This will show if there is a bottleneck when allocating the memory. It is also possible to experiment with the number of elements that are reserved each time. If a normal stack does not hold more than 500 elements, then maybe 500 are good enough to reserve. With that being said, it is most likely faster to use cmap without initiation compared to the old ParameterStack and unnecessary to allocate memory because of the small sizes of the ParameterStack.

The new implementation has been tested to work properly for the methods Add, Get,
AddFirst, Delete, DeleteAll, and Update. Still, the new implementation might need to be tested even further before putting it into the system.
Bibliography


# Appendix A

## Table of distribution

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of files</th>
<th>Number of methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>237</td>
<td>6411</td>
</tr>
<tr>
<td>Addfirst</td>
<td>19</td>
<td>43</td>
</tr>
<tr>
<td>AddSubItem</td>
<td>10</td>
<td>102</td>
</tr>
<tr>
<td>AddBlock</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Get</td>
<td>298</td>
<td>4039</td>
</tr>
<tr>
<td>GetSubItem</td>
<td>4</td>
<td>63</td>
</tr>
<tr>
<td>GetDataBlock</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>HasItem</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>HasItemWithValue</td>
<td>11</td>
<td>24</td>
</tr>
<tr>
<td>HasDataBlock</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>GetDataBlockFromIndex</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>GetDataBlockNameFromIndex</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Delete</td>
<td>18</td>
<td>60</td>
</tr>
<tr>
<td>DeleteAll</td>
<td>54</td>
<td>197</td>
</tr>
<tr>
<td>DeleteDataBlock</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Update</td>
<td>31</td>
<td>92</td>
</tr>
<tr>
<td>DeleteRow</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>GetMaxRowCounter</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>GetMaxSubItemCounter</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>AddWithPrefix</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>GetWithPrefix</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MoveFirst</td>
<td>19</td>
<td>31</td>
</tr>
<tr>
<td>MoveNext</td>
<td>17</td>
<td>44</td>
</tr>
<tr>
<td>MovePrev</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>GetFirst</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>GetNext</td>
<td>20</td>
<td>41</td>
</tr>
<tr>
<td>GetNextAndVerify</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>GetCurrent</td>
<td>17</td>
<td>41</td>
</tr>
<tr>
<td>GetCurrentType</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>TranslateParameterStackToString</td>
<td>26</td>
<td>60</td>
</tr>
<tr>
<td>TranslateStringToParameterStack</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>TranslateStringToParameterStack_new</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>GetStatus</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SetSeparator</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GetSeparator</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Figure A.1: The distribution between ParameterStacks’ methods.
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Worst</td>
</tr>
<tr>
<td></td>
<td>Access</td>
<td>Search</td>
</tr>
<tr>
<td>Array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Stack</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Queue</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Singly-Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Doubly-Linked List</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Skip List</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Hash Table</td>
<td>N/A</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Cartesian Tree</td>
<td>N/A</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>B-Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Red-Black Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Splay Tree</td>
<td>N/A</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>KD Tree</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
</tr>
</tbody>
</table>

Figure A.2: Complexity for different data structure’s operations[1].