

Execution Efficiency of the use of Array and Linked-List Implementations of a Stack Abstract Data Types Containing Complex Numbers in Methods of an Android Application

Igor Košťál¹, Martin Mišút¹

¹ *University of Economics in Bratislava, Faculty of Economic Informatics, Dolnozemska cesta 1, Bratislava, Slovakia*

Abstract – Abstract data types (ADTs) provide a way to define data structures and the operations allowed on them, independent of the specific implementation details. Choosing the appropriate data type is for many applications the most important step in their development that affects their performance. To investigate the most suitable stack implementation for evaluating arithmetic expressions with complex numbers, we developed an Android application. These expressions may contain a larger number of complex numbers, for example, 25, which can be enclosed in parentheses arbitrarily. The Android application uses an array and linked-list implementation of a stack ADT to evaluate these expressions in its methods, as well as a simple stack implementation that uses none ADT. We determined a more efficient implementation of a stack ADT and the most efficient implementation at all by analysing the execution times of these methods, which were evaluating the same arithmetic expressions with complex numbers.

In the paper we also analyse the functioning of both these stack ADT implementations and a simple stack implementation that does not use an ADT.

Keywords – Abstract data type, stack ADT, array implementation and linked-list implementation of a stack ADT, Android application.

1. Introduction

Calling is just one of the many functions offered by today's smartphones. An user can run often very specific applications in a smartphone operating system that do not relate to calling, e.g., an interactive Android mobile application for improving the communication skills of Arab children with autism [1], or an Android smart parking mobile application [2] that helps pre-book a parking space in a university campus swiftly and easily. Thereby, today's smartphone becomes a sophisticated computer with a full-fledged operating system that enables to execute specific applications of various kinds.

Our application, the Calculator, expands the possibilities of using an Android smartphone, because it enables evaluating even extensive arithmetic expressions with complex numbers. These expressions may contain a larger number of complex numbers, for example, 25, which can be enclosed in parentheses arbitrarily. The calculator in the current standard package of applications supplied with the Android 11, 12, or 13 operating systems on new Android smartphones does not include a module for evaluating arithmetic expressions with complex numbers.

As we mentioned above, a programmer fundamentally affects the performance of an application and its memory requirements by choosing a suitable data type during the development of an application.

DOI: 10.18421/TEM132-75

<https://doi.org/10.18421/TEM132-75>

Corresponding author: Igor Košťál,
University of Economics in Bratislava, Faculty of Economic Informatics, Bratislava, Slovakia


Email: igor.kostal@euba.sk

Received: 28 December 2023.

Revised: 08 January 2024.

Accepted: 08 April 2024.

Published: 28 May 2024.

 © 2024 Igor Košťál & Martin Mišút; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDeriv 4.0 License.

The article is published with Open Access at <https://www.temjournal.com/>

These facts highlight the high importance of this choice. Therefore, we also dealt with creating and selecting a suitable data type to work with complex numbers in our Android application.

In terms of programming and the application itself a data type is a set of values and a collection of operations on those values. Procedural and object-oriented programming languages have built-in data types that developers can use immediately. They can also create their own simple data types (structures, enumerations, and so on) or ADTs. We have created our own object-oriented ADTs, three classes, for our Android application that allow it to perform arithmetic operations with real and complex numbers. One class represents a linked-list implementation of a stack ADT that works with real numbers. The next two classes represent array and linked-list implementations of the stack ADT that works with complex numbers. Our main aim was to compare an execution efficiency of the use of array and linked-list implementations of a pushdown stack ADT containing complex numbers in methods of a mobile Android application. Expressions that evaluate these methods can be in a postfix or infix form. We have wondered whether a form of complex expressions influences an execution efficiency of these methods. To investigate and measure this impact, we have created four methods. Two methods evaluate postfix arithmetic expressions with complex numbers. They first convert the infix expressions to postfix expressions by the conversion method and then evaluate them. The next two methods evaluate input infix complex expressions, but without calling the conversion method. We have also wondered whether the use of the objects of ADTs by member methods for evaluating expressions with complex numbers has an impact on their execution efficiency. To investigate and measure this impact, we have created the next two methods that evaluate input infix complex expressions without using objects of ADTs. Otherwise, they use a stack to evaluate complex expressions, but in a simple form of an array. In Section V we deal with the experiment, using which we want to determine a more efficient implementation of a stack ADT and the most efficient implementation at all by analysing the execution times of all methods, which were evaluating the same arithmetic expressions with complex numbers.

The contribution of this paper as compared to other relevant works is:

- true ADTs were created, two classes, for working with complex numbers in methods of an Android application. One class represents an array implementation, and the next class represents a linked-list implementation of the stack ADT.

All member variables of these classes are private. In this case client methods can access the instance variables of objects of these classes using only instance methods, they cannot access them directly. The member methods of these two classes representing ADTs create interfaces of ADTs, and client methods can access the data of particular ADTs only through these interfaces.

- A comparison of an execution efficiency of the use of array and linked-list implementations of a pushdown stack ADT was carried out containing complex numbers in methods of a mobile Android application.
- the study has also compared an execution efficiency of an Android application methods with array and linked-list implementations of a pushdown stack ADT that were evaluating postfix expressions with complex numbers to similar methods that were evaluating infix expressions with the same complex numbers.
- we have also compared an execution efficiency of an Android application methods with an array and linked-list implementation of a pushdown stack ADT that were evaluating expressions with complex numbers to methods that do not use ADTs, and which were evaluating expressions with the same complex numbers.
- the authors have created Android application methods, which are able to evaluate arithmetic expressions with complex numbers, while these expressions may contain a larger number of complex numbers, for example, 25, which can be enclosed in parentheses arbitrarily. These methods recognize operations precedence correctly and calculate the value of such a long expression with parentheses correctly.

The next parts of the paper are structured as follows. Section 2, contains a summary of a research pertaining to ADTs and a comparison of an execution efficiency of the use of array and linked-list implementations of a stack ADT in applications. We briefly theoretically deal with ADTs, a stack ADT, complex numbers, and arithmetic operations with complex numbers in Section 3. We deal with our ADTs and also with our Android application itself and its architecture briefly in Section 4. In Section 5, we deal with the experiment, using which we want to determine the most execution efficient implementation of the stack. Conclusion, in which we briefly evaluate our experiment, is found in Section 6. Section 3 provides a brief theoretical overview of abstract data types (ADTs), focusing specifically on stack ADTs. It also covers complex numbers and the arithmetic operations applicable to them.

Section 4 offers a concise introduction to the chosen ADTs and the Android application itself, outlining its architecture. The focus of Section 5 is the experiment designed to identify the most execution-efficient stack implementation. The concluding section (Section 6) presents a brief evaluation of the experiment's findings.

2. Related Work

We did not find studies or papers dealing with our topic *A Comparison of an Execution Efficiency of the Use of Array and Linked-list Implementations of a Pushdown Stack ADT Containing Complex Numbers in Methods of an Android Application* or similar to our topic on the Web of Science, and the Scopus. In these sources, we have found works that deal only with parts of our topic.

Gutttag [3] deals with the algebraic specification of the semantics of an abstract data type. He also defines an ADT: the term "abstract data type" refers to a class of objects defined by a representation-independent specification. He emphasizes the role of ADTs in design of software system architecture. He argues that ADTs provide unambiguous specifications that lead to more efficient implementations chosen after more is known about the behavior of the system [4].

Eliëns [5] regards abstract data types as an essential constituent of object-oriented modelling. He relates an encapsulation, one of the features of object-oriented programming, to abstract data types (classes), because their elements are usually created using a hidden state.

We identify with these claims about ADTs in [3], [4], [5], and with an ADT definition in [3]. This an ADT definition and a purpose of ADTs in object-oriented modelling that is specified in [5], satisfy implementations of the stack ADT in the *LinkedList*, *cplxArray* (Fig. 4), and *cplxLinkedList* (Fig. 5) classes of our Android application.

Fürst *et al.* [6] in their study emphasize an important advantage of using ADTs in the development of software systems: if we specify systems using ADTs, then they are more abstract and thereby easier to verify than systems designed directly without ADTs.

To this advantage we can add other properties of such systems that use ADTs. These systems are easier to extend, because their ADTs are easier to extend, and they are better to maintain than systems without ADTs. Also, from these reasons, we have created ADTs, the *LinkedList*, *cplxArray* (Fig. 4), and *cplxLinkedList* (Fig. 5) classes, in our Android application, which create the basic building components of its object-oriented architectural design.

Zhong, Ishizuka, and Enari [7] emphasize a very strong link between Object-Oriented Programming (OOP) and an ADT: OOP design is the construction of software as structured collections of ADTs implementations.

The *LinkedList*, *cplxArray*, and *cplxLinkedList* classes of our object-oriented Android application are created exactly like this, a stack ADT is implemented using an array in the *cplxArray* (Fig. 4) class, and a stack ADT is also implemented using a linked list in the *LinkedList*, and *cplxLinkedList* (Fig. 5) classes.

In other sources - books, and the Internet, we have found studies or works that dealt with similar topics to our topic.

For example, Agostini [8] has implemented a stack ADT using a linked list and an array in the Swift 3.0 programming language developed by Apple Inc. He has compared performances between these two implementations. A linked list implementation of his stack ADT was much faster than an array implementation of this stack. However, his stack contained only simple integer data.

Our stack in both implementations in our Android application, in a linked list and an array implementation, contains more complicated data, expressions with complex numbers that have operands and operators. Therefore, a comparison of performances of both our implementations of a stack ADT in our Android application that is written in the C# programming language can have different results from his results.

Rajput-Ji [9] has implemented a stack ADT using a linked list in the C# programming language. He has dealt with benefits of implementing a stack using a linked list, such as is an efficient memory usage and a dynamic memory allocation of this implementation of a stack. His stack contained only simple integer data.

Duggempudi [10] has implemented a stack ADT using an array and linked list in the C++ programming language. He did not compare performances between these two implementations. His stack also contained only simple integer data.

K Hong [11] has implemented a non-generic stack ADT using an array of integer numbers, a generic stack ADT using an array of floating-point numbers and an array of strings, and a non-generic stack ADT containing integer numbers using a linked list in the C++ programming language. He did not compare performances between array and linked implementations. His stacks also contained only simple data: integer numbers, floating-point numbers, and strings.

Sedgewick [12] has created an ADT, the *Complex* class, to work with complex numbers.

However, this ADT does not use a stack ADT to store complex numbers, their real and imaginary parts are stored in private member variables of the float type, and this ADT contains one arithmetic operation, the multiplication of complex numbers. This operation is executed by the overloaded operator *, the *Complex* class contains the *operator** method.

The *cplxArray* (Fig. 4), and *cplxLinkedList* (Fig. 5) classes of our Android application for working with complex numbers use an array and linked-list implementation of a stack ADT, and methods of these classes can perform all arithmetic operations with complex numbers. Both these implementations of a stack ADT contain more complicated data (expressions with complex numbers that consist of operands and operators) than stacks ADTs in [8], [9], [10], [11], and [12].

The next section theoretically deals with ADTs, a stack ADT, complex numbers, and arithmetic operations with complex numbers.

3. ADTs, a Stack ADT, Arithmetic Operations with Complex Numbers

An abstract data type (ADT) is a data type (a set of values and a collection of operations on those values) that is accessed only through an interface [13], [14]. We refer to a program or a method that uses an ADT as a client, and a class that specifies the data type as an implementation [13].

Fundamental ADTs are stacks and queues that are implemented using classical data structures (arrays, linked lists, and strings).

A *pushdown stack* is an ADT that comprises two basic operations: insert (push) a new item, and delete (pop) the item that was most recently inserted [13]. A pushdown stack is a LIFO (Last In, First Out) data structure [15]. It is the ideal mechanism for evaluating postfix arithmetic expressions, but also for converting an infix expression, for example, $(2 + 3i) * ((-4 - 3i) + (2 + 4i))$, to the postfix expression $2\ 3i\ -4\ -3i\ 2\ 4i\ +\ *$. In a postfix expression that does not need parentheses each operator appears after its two arguments. This is an important property of a postfix expression for which this expression is suitable for evaluating by a stack (Fig. 8 and Fig. 10). This implies that the pushdown stack appears to be a very suitable data structure that can be used for evaluating postfix arithmetic expressions with complex numbers. However, which implementation of the pushdown stack, array or linked-list, is more execution efficient? This is the main subject of our research. For this purpose, we have created the Calculator as an Android application that can evaluate internal postfix arithmetic expressions with complex numbers using the array and linked-list implementation of the

pushdown stack ADT as well as using a simple stack implementation without the use of an ADT.

A complex number z is any expression of the form $z = a + bi$, where a (the *real* part of z) and b (the *imaginary* part of z) are real numbers and i is the imaginary unit that satisfies $i^2 = -1$ [16]. The expression $a + bi$ is called the algebraic representation (form) of the complex number z [17].

Using algebraic representations of complex numbers, the arithmetic operations, the addition, subtraction, multiplication, and division, with complex numbers $z_1 = a + bi$ and $z_2 = c + di$ can be performed by the following formulas [17], [18], [19], [20], [21]

$$z_1 \pm z_2 = (a \pm c) + (b \pm d)i \quad (1)$$

$$z_1 \cdot z_2 = (ac - bd) + (ad + bc)i \quad (2)$$

$$\frac{z_1}{z_2} = \frac{z_1 \cdot \overline{z_2}}{z_2 \cdot \overline{z_2}} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} i \quad (3)$$

These formulas are written into the source code of the *evaluate_CPLXexp* method of our Android application, which evaluates elementary arithmetic expressions with complex numbers, and which is called by all methods that evaluate the input infix arithmetic expressions with complex numbers. We deal with our Android application, its ADTs and its important methods in the next chapter.

4. An Android Application that Uses ADTs

We have created the Calculator as the Android application, which is able to evaluate arithmetic expressions with complex numbers. These expressions may contain a larger number of complex numbers, for example, 25, which can be enclosed in parentheses arbitrarily (Fig. 15). Our Android application recognizes operations precedence correctly and calculates the value of such a long expression with parentheses correctly. The calculator in the current standard package of applications supplied with the Android 11, 12 or 13 operating system on new Android mobile phones does not include a module for evaluating arithmetic expressions with complex numbers.

We have developed our Android application in the C# programming language and in the development environment Microsoft Visual Studio 2019 Enterprise. It was developed as a single-page Xamarin.Forms application. The target operating system of this application is the Android 9 Pie and higher versions.

We have created a skeleton of our cross-platform Xamarin.Forms solution by a solution template.

It has created three projects: one common project (the Portable Class Library project) and two platform projects - for iOS, and Android. The common project was built into a dynamic-link library (DLL) that is referenced by both the individual platform projects [22]. The source code of the common project is divided into four source files:

App.xaml - this XAML (Extensible Application Markup Language) file contains the *x:Class* specification, which indicates that the *App* class in the *calcCPLX* namespace derives from the *Application* class.

```
<?xml version="1.0" encoding="utf-8" ?>
<Application
  xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="calcCPLX.App">
  <Application.Resources></Application.Resources>
</Application>
```

Figure 1. The 'App.xaml' file

App.xaml.cs - this *.cs* file contains the *App* class definition. At run time, the constructor of the *App* class instantiates the *MainPage* class and sets an instance of this class to the *MainPage* property of the *App* class. The *MainPage* constructor (defined in the *MainPage.xaml.cs* code-behind file) calls the *InitializeComponent* method (defined in the *MainPage.xaml.g.cs* generated file), and the *InitializeComponent* method calls the *LoadFromXaml* method. It loads the *MainPage.xaml* file and parses it, instantiating and initializing all the elements in this XAML file except for the root element, which already exists [22]. The instance of the *App* class is an important part of the startup code of our Android application.

```
namespace calcCPLX {
  public partial class App : Application {
    public App() { MainPage = new calcCPLX.MainPage(); }
  }
}
```

Figure 2. The part of the 'App.xaml.cs' file

MainPage.xaml - this XAML file contains the definition of the Android application tree-structured user interface that was created by XAML. Objects of all controls (buttons and labels) are instantiating and initializing in a XAML code of this file.

MainPage.xaml.cs - this *.cs* file and the *MainPage.xaml* file contribute to the *MainPage* class that derives from the *ContentPage* class and that is defined in this *MainPage.xaml.cs* code-behind file. The *MainPage* class is used to create the underlying logic of the user interface of our Android application. The user interface consists of only a single page (Fig. 15).

The *MainPage* class includes event handlers of all controls (buttons and labels) of this user interface and helper methods. These event handlers and helper methods are member methods of this class. The *MainPage* class also contains two important nested structures and five important nested classes, which create ADTs.

The *App.xaml.cs* and *MainPage.xaml.cs* are code-behind files of the *App.xaml* and *MainPage.xaml* files.

During building the Xamarin.Forms solution all its projects were built.

When the iOS platform project was built, it needed to use the Apple compiler on the Mac to generate native iOS machine code from the C# Intermediate Language (IL) [22]. We did not use this compiler on the Mac to build the iOS platform project into the final iOS application.

When the Android platform project was built into the Android application, the Xamarin C# compiler generated IL, which runs on a version of Mono on the Android device alongside the Java engine, but the API calls from the application are pretty much the same as though the application were written in Java [22].

4.1. Nested Structures and Nested Classes (ADTs) within the MainPage Class

Two structures *cplx_str* and *cplx* are declared within the *MainPage* class.

The *cplx_str* structure type is used to create arrays of this type. Real and imaginary parts of complex numbers in the form of strings are stored into the inner variables of elements of such arrays by the *InfixToPostfixCPLX* member method of the *MainPage* class.

The *cplx* structure type is used in the constructor of the *cplxArray* class to create a dynamic array of this type that represents a stack containing complex numbers in an array implementation of a stack ADT. This structure type is also used in the constructor of the *cplxNode* class to create data parts of data elements of a linked list that represents a linked-list implementation of the stack ADT (the *cplxLinkedList* class) working with complex numbers. This structure type is also used to create local variables and return values of various member methods of the *MainPage* class, for example, *evaluate_CPLXexp*, *EvalCPLXPostfixByArrayStack*, *EvalCPLXPostfixByObjArrayStack*, *EvalCPLXPostfixByLinkedListStack* and *EvalCPLX_INFIFXByArray Stack*.

Five nested classes *LinkedList*, *Node*, *cplxArray*, *cplxLinkedList* and *cplxNode* are defined within the *MainPage* class (Fig. 3), too.

The *LinkedList* class represents a linked-list implementation of a stack ADT that works with real numbers. The *Node* class object represents one data element (node) of a linked list created using the constructor of the *LinkedList* class. The *cplxArray* class represents an array implementation, and the *cplxLinkedList* class represents a linked-list implementation of the stack ADT. Both these implementations work with complex numbers. The *cplxNode* class object represents one data element (node) of a linked list created using the constructor of the *cplxLinkedList* class.

All three ADTs have been created truly abstract. All member variables in the *LinkedList*, *cplxArray* and *cplxLinkedList* classes are private, in this case client methods can access the instance variables of objects of these classes using only instance methods, they cannot access them directly.

The member methods of these three classes representing ADTs create interfaces of ADTs, and client methods can access the data of particular ADTs only through these interfaces.

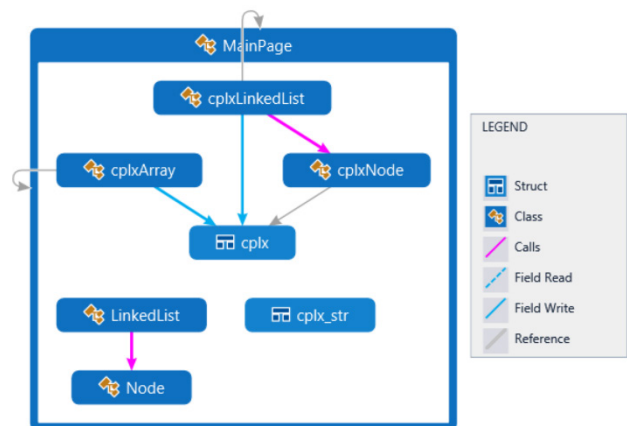


Figure 3. The class class

```

public struct cplx {
    public double re;
    public double im;
}

public class cplxArray {
    private cplx[] stackCplx;
    private int count;
    private int peek;
    private MainPage objMainPage;

    public cplxArray(int x) {
        stackCplx = new cplx[x];
        count = 0;
        peek = 0;
    }

    public void STACKpushArr(cplx data) {
        stackCplx[peek++] = data;
        count++;
    }

    public cplx STACKpopArr() {
        if (count > 0) {
            count--;
            return stackCplx[--peek];
        }
        else {
            objMainPage.result_Lbl.Text = "No
            element exists in this array.";

            cplx empty_value;
            empty_value.re = 0;
            empty_value.im = 0;
            return empty_value;
        }
    }
}
    
```

Figure 4. The declaration of the 'cplx' nested structure and the definition of the 'cplxArray' nested class (ADT)

```

public class cplxLinkedList
{
    private cplxNode head;
    private int count;
    private MainPage objMainPage;

    public cplxLinkedList()
    {
        head = null;
        count = 0;
    }

    public void STACKpush(cplx data)
    {
        cplxNode newNode = new cplxNode() { value = data };
        if (head == null)
            head = newNode;
        else
        {
            newNode.next = head;
            head = newNode;
        }
        count++;
    }

    public cplx STACKpop()
    {
        if (count > 0)
        {
            cplx removed_value = head.value;
            head = head.next;
            count--;
            return removed_value;
        }
        else
        {
            objMainPage.result_Lbl.Text = "No element
                                         exists in this array.";

            cplx empty_value;
            empty_value.re = 0;
            empty_value.im = 0;
            return empty_value;
        }
    }
}

public class cplxNode
{
    public cplx value;
    public cplxNode next;
}

```

Figure 5. The definitions of the 'cplxLinkedList' and 'cplxNode' nested classes (ADTs)

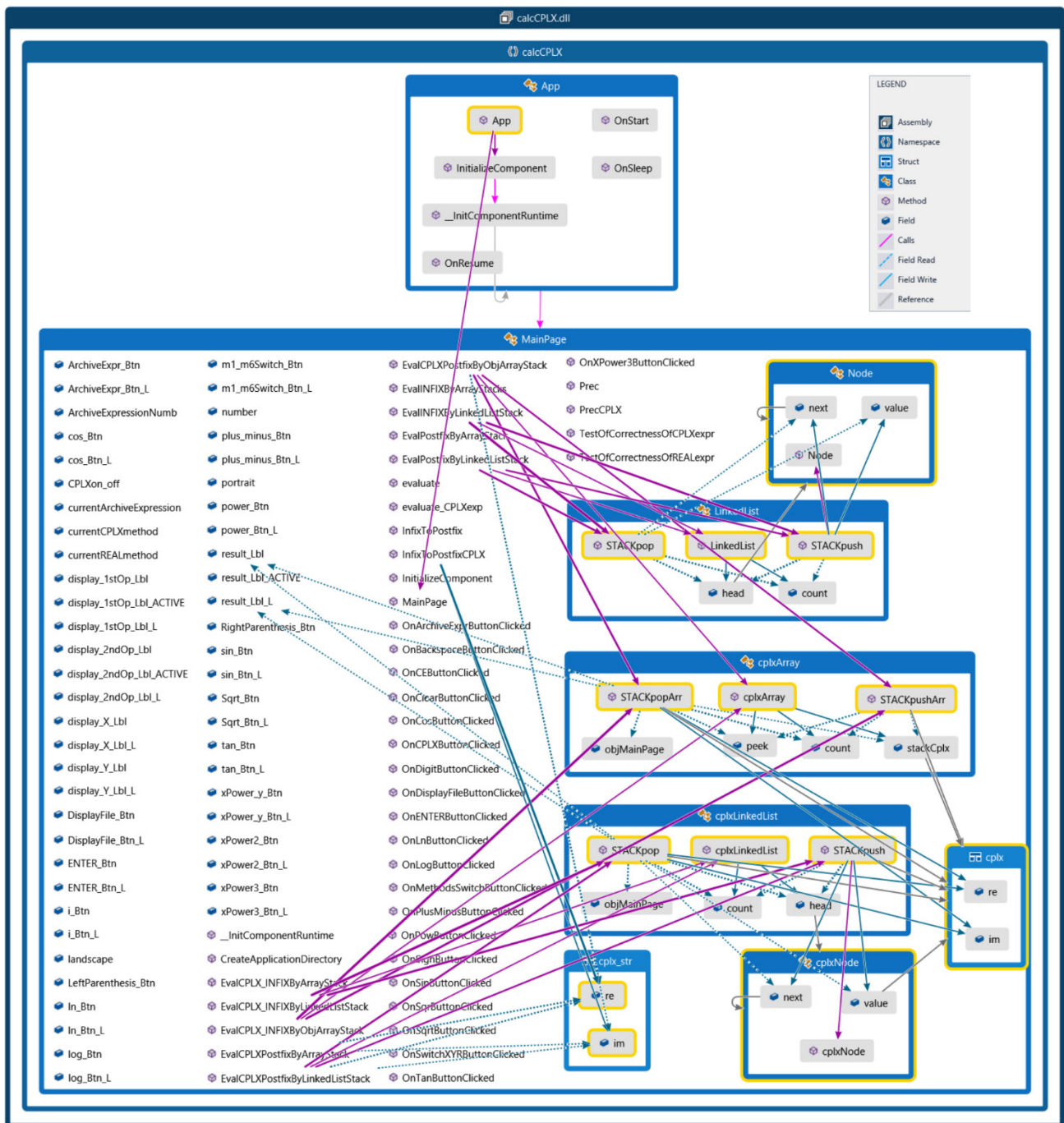


Figure 6. The code map of the Android application

4.2. Member Methods of the MainPage Class that use ADTs

The *EvalCPLXPostfixByObjArrayStack*, *EvalCPLXPostfixByLinkedListStack*, *EvalCPLX_INFIXByObjArrayStack* and *EvalCPLX_INFIXByLinkedListStack* member methods use objects of created ADTs for evaluating arithmetic expressions with complex numbers. The *EvalCPLXPostfixByObjArrayStack* and *EvalCPLXPostfixByLinkedListStack* methods evaluate postfix arithmetic expressions with complex numbers. Therefore they first convert the input infix expressions to internal postfix expressions by the *InfixToPostfix CPLX* method and then evaluate them.

The *InfixToPostfixCPLX* conversion method uses two stacks for converting an input infix arithmetic expression with complex numbers to a postfix expression. It uses the *CplxStrs* 100-element array of structure variables of the *cplx_str* type as a stack for storing operands. The index of the top of this stack is always the maximum index of the used element. The method uses the *stack_op* object of the *System.Collection.Generic.Stack<T>* is library class [23] as a stack for storing operators and parentheses. The top of this stack is always an element with index 0. The method assumes that particular complex numbers in an infix expression are enclosed in parentheses.

These parentheses always indicate the beginning and end of a given complex number for the *InfixToPostfixCPLX* method, but during a conversion they are not processed, they are ignored by this method. Similarly, all other member methods that work with the input infix expression assume that the particular complex numbers in this expression are enclosed in parentheses. These are processed by these methods in the same way as by the *InfixToPostfixCPLX* method. The Fig. 7 shows the states of both stacks of the *InfixToPostfixCPLX* method during converting the infix expression $(2 + 3i) * ((-4-3i) + (2 + 4i))$ to postfix. The method proceeds from left to right through the expression. If it encounters a complex number (operand), it writes it at the top of the *CplxStrs* stack. If it encounters an operator in an infix expression, it successively examines whether the priority of the operators in the *stack_op* stack is greater than or equal to the priority of the being processed operator. If such operator is found in the *stack_op* stack, it is removed from this stack and inserted at the top of the *CplxStrs* stack of operands. Then the being processed operator is inserted at the top of the *stack_op* stack. If the method encounters a left parenthesis, it pushes this parenthesis at the top of the *stack_op* stack of operators. If it encounters a right parenthesis, it removes all operators "above" the left parenthesis from the *stack_op* stack and successively stores them at the top of the *CplxStrs* stack. Then the method deletes the left parenthesis from the *stack_op* stack, removes all remaining operators from this stack, and successively stores them at the top of the *CplxStrs* stack. Finally, the method returns a reference to the *CplxStrs* stack, in which is stored the resulting postfix expression $2\ 3\ -4\ -3\ 2\ 4\ +\ *$, which the method created from the input infix expression $(2 + 3i) * ((-4-3i) + (2 + 4i))$.

The *EvalCPLXPostfixByObjArrayStack* (CM2) and *EvalCPLXPostfixByLinkedListStack* (CM3) methods, which have the most efficient written source codes from all our methods, and which use the objects of ADTs, the *cplxArray* and *cplxLinkedList* classes, for implementing a stack and operations in it, are the subject of our research, so we deal with them in more detail. As we mentioned above, these methods first convert an input infix expression to postfix using the *InfixToPostfixCPLX* conversion method. Then the methods read this postfix expression from left to right and evaluate it.

The CM2 method creates the *arrayStack* object of the *cplxArray* class. The *stackCplx* instance variable of this object is the 100-element array of the *cplx* type that represents the stack of the CM2 method.

The CM3 method creates the *Inklist* object of the *cplxLinkedList* class. This object is the linked list that represents the stack of the CM3 method.

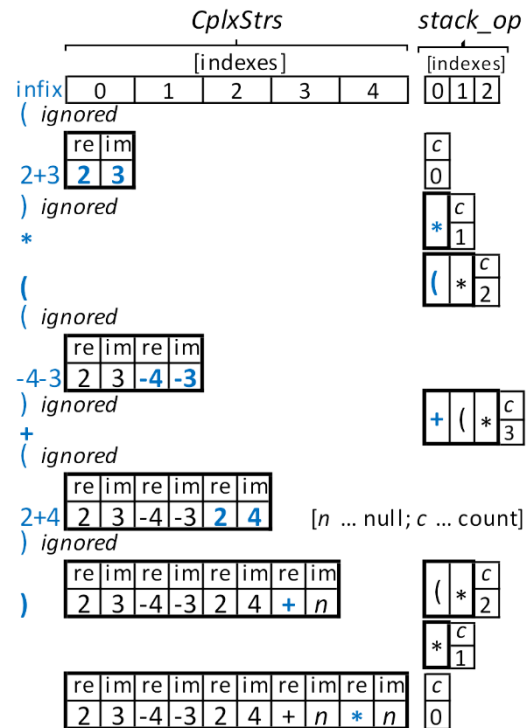


Figure 7. States of the 'CplxStrs' and 'stack_op' stacks of the 'InfixToPostfixCPLX' method during conversion of the input infix expression $(2 + 3i) * ((-4-3i) + (2 + 4i))$ to the output postfix expression $2\ 3\ -4\ -3\ 2\ 4\ +\ *$

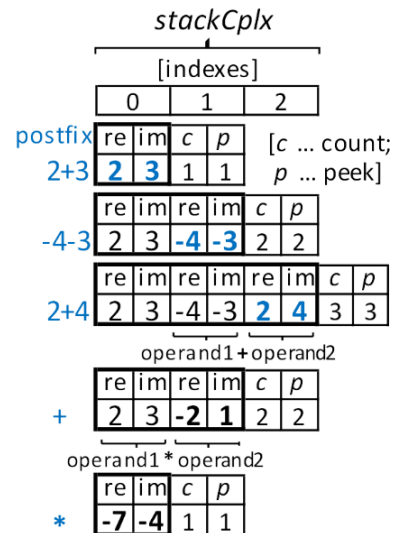


Figure 8. States of the 'stackCplx' stack (the instance variable of the 'arrayStack' object) of the 'EvalCPLXPostfixByObjArrayStack' (CM2) method during evaluating the postfix expression $2\ 3\ -4\ -3\ 2\ 4\ +\ *$. The figure shows only the used elements of the 'stackCplx' 100-element array (the stack) in particular phases of processing the expression by the CM2 method. The 'stackCplx' 100-element array is allocated throughout the lifetime of the 'arrayStack' object

The *STACKpushArr* and *STACKpopArr* instance methods of the *arrayStack* object created by the CM2 method and *STACKpush* and *STACKpop* instance methods of the *Inklist* object created by the CM3 method use the CM2 and CM3 methods to push and pop operands and results of particular operations at the top and from the top of their stacks.

The source codes of both methods are almost identical. They differ only in the used object of an ADT. Therefore, their functioning is very similar. We will describe how the *EvalCPLXPostfixByObjArrayStack* (CM2) method works. This method reads a postfix expression from left to right. If it encounters an operand, it pushes this operand onto the stack. If the method encounters an operator, it pops the top two operands from the stack, performs the corresponding operation on them, and pushes its result at the top of the stack. When all operators in the postfix expression have been processed, the result of the evaluation of the postfix expression that the method returns is at the top of the stack.

```

public cplx EvalCPLXPostfixByObjArrayStack(
    string[] InfixArr_strs)
/ public cplx EvalCPLXPostfixByLinkedListStack(
    string[] InfixArr_strs) {
// converting an input infix expression to postfix using
// the 'InfixToPostfixCPLX' conversion method
cplx_str[] PostFixArrCplx_strs =
    InfixToPostfixCPLX(InfixArr_strs);
// initializing an empty stack
cplxArray arrayStack = new cplxArray(100);
/ cplxLinkedList Inklist = new cplxLinkedList();
int i = 0;
while (PostFixArrCplx_strs[i].re != null) {
    string str = PostFixArrCplx_strs[i].re;
    double operand_re = 0;
    // If the scanned string is an operand,
    if (Double.TryParse(str, out operand_re)) {
        cplx operand; operand.re = operand_re;
        operand.im = Double.Parse(PostFixArrCplx_strs[i].im);
        // push it to the stack.
        arrayStack.STACKpushArr(operand);
        / Inklist.STACKpush(operand); }
    // If the scanned string is an operator,
    else if((str == "+") || (str == "-") || (str == "**") || (str == "/"))
    {
        // pop the top two operands from the stack,
        cplx operand2 = arrayStack.STACKpopArr();
        / cplx operand2 = Inklist.STACKpop();
        cplx operand1 = arrayStack.STACKpopArr();
        / cplx operand1 = Inklist.STACKpop();
        // perform the corresponding operation on them,
        // and push its result at the top of the stack.
        arrayStack.STACKpushArr(evaluate_CPLXexp(
            operand1, str, operand2));
        / Inklist.STACKpush(evaluate_CPLXexp(
            operand1, str, operand2)); } i++; }
return arrayStack.STACKpopArr();
/ return Inklist.STACKpop(); }
    
```

Figure 9. The source codes of the 'EvalCPLXPostfixByObjArrayStack' (CM2) and 'EvalCPLXPostfixByLinkedListStack' (CM3) methods. (Differences in the CM3 method source code are written in blue.)

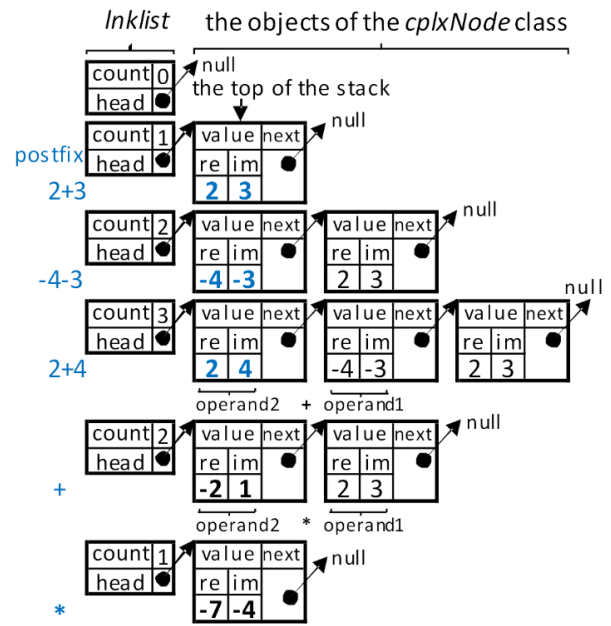


Figure 10. States of the 'Inklist' stack of the 'EvalCPLXPostfixByLinkedListStack' (CM3) method during evaluating the postfix expression $2\ 3\ -4\ -3\ 2\ 4\ +\ *$. The figure shows the actual occupation of memory by the 'Inklist' object, the linked list that represents the stack, in particular phases of processing the expression by the CM3 method. The stack occupies no additional memory space

We have wondered whether the conversion of the input infix complex expression of the CM2 and CM3 methods to the postfix by the *InfixToPostfixCPLX* method has an effect on their execution efficiency. To investigate and measure this impact, we have created the *EvalCPLX_INFIXByObjArrayStack* (CM5) and *EvalCPLX_INFIXByLinkedListStack* (CM6) methods, which evaluate input infix complex expressions, but without calling the *InfixToPostfixCPLX* conversion method.

The CM5 and CM6 methods also use the objects of ADTs, the *cplxArray* and *cplxLinkedList* classes, for implementing a stack and operations in it. These methods do not evaluate postfix expressions. They work directly with input infix expressions that successively convert to elementary postfix expressions, which they immediately evaluate using the stack. From the results of elementary postfix expressions they successively assemble the result of the evaluation of the input infix expression.

The CM5 method creates the *arrayStack* object of the *cplxArray* class. The *stackCplx* instance variable of this object is the 100-element array of the *cplx* type that represents the stack of the CM5 method. The CM6 method creates the *Inklist* object of the *cplxLinkedList* class. This object is the linked list that represents the stack of the CM6 method.

The *STACKpushArr* and *STACKpopArr* instance methods of the *arrayStack* object created by the CM5 method and *STACKpush* and *STACKpop* instance methods of the *lnklist* object created by the CM6 method use the CM5 a CM6 methods to push and pop operands and results of particular operations at the top and from the top of their stacks. Besides these stacks of operands, each method uses its own *operators_stack* stack of operators and parentheses created as the object of the *System.Collection.Generic.Stack<T>* library class [23]. These stacks have tops in elements with indexes 0. The methods use them to push and pop operators and parentheses of the input infix expression.

The source codes of both methods are almost identical. They differ only in the used object of an ADT. Therefore, their functioning is very similar. We will describe how the *EvalCPLX_INFIXByObjArrayStack* (CM5) method works.

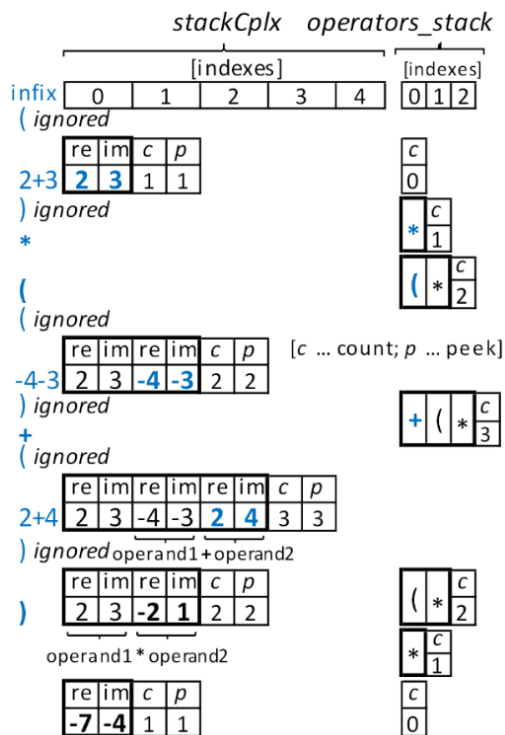


Figure 11. States of the 'stackCplx' and 'operators_stack' stacks of the 'EvalCPLX_INFIXByObjArrayStack' (CM5) method during evaluating the input infix expression $(2 + 3i) * ((-4 - 3i) + (2 + 4i))$

The method proceeds from left to right through the infix expression. If it encounters a complex number (operand), it writes this operand at the top of the *stackCplx* stack. If it encounters an operator in an infix expression, it successively examines whether the priority of the operators in the *operators_stack* stack is greater than or equal to the priority of the being processed operator.

If such operator is found in the *operators_stack* stack, it is popped from this stack, the method performs the corresponding operation on the top two operands from the *stackCplx* stack and pushes its result at the top of the *stackCplx* stack. Then the being processed operator is inserted at the top of the *operators_stack* stack. If the method encounters a left parenthesis, it pushes this parenthesis at the top of the *operators_stack* stack of operators. If it encounters a right parenthesis, it successively performs all the operations specified by the operators placed "above" the left parenthesis in the *operators_stack* stack, i.e., it successively reads and removes the operator from the top of this stack, it performs the corresponding operation on the top two operands from the *stackCplx* stack and writes its result at the top of this stack. Then the method deletes the left parenthesis from the *operators_stack* stack and performs the corresponding operations with the remaining operators in this stack in the same way as described in the previous sentence. Finally, the method returns an element from the top of the *stackCplx* stack that contains the result of evaluating the input infix expression.

4.3. Member Methods of the MainPage Class that do not use ADTs

We have also wondered whether the use of the objects of ADTs by member methods for evaluating expressions with complex numbers has an impact on their execution efficiency. To investigate and measure this impact, the *EvalCPLXPostfixByArrayStack* (CM1) and *EvalCPLX_INFIXByArrayStack* (CM4) methods that evaluate input infix complex expressions without using objects of ADTs were created. Otherwise, they use a stack to evaluate complex expressions, but in a simple form of an array.

The CM1 method, which evaluates a postfix expression, works very similar to the CM2 method. The difference how it works is that it does not use the object of an ADT. The method implements a stack of operands and results of operations in a postfix expression in the *stackCplx* 100-element array of the *cplx* type.

The CM4 method, which evaluates the input infix expression directly, works very similar to the CM5 method. The difference how it works is that it does not use the object of an ADT. The method implements a stack of operands and results of operations in elementary postfix expressions in the *stackCplx* 100-element array of the *cplx* type.

5. Experiment, its Results, and their Brief Analysis

The use of the array implementation of the stack ADT in the *EvalCPLXPostfixByObjArrayStack* (CM2) method should be more efficient than the use of the linked-list implementation of the stack ADT in the *EvalCPLXPostfixByLinkedListStack* (CM3) method, if both methods evaluate the same arithmetic expressions with complex numbers. Our assumption results from the fact that the list implementation uses more execution time for push and pop operations than the array implementation, because it allocates memory for each push and deallocates memory for each pop operation.

We have wondered what number of complex numbers in the expression is being evaluated will make a difference in execution efficiency of the use of these two implementations obvious. Verifying these assumptions was the main subject of our research. However, we have also wondered whether it is possible to create methods for evaluating complex expressions using a stack even with greater execution efficiency than have the CM2 and CM3 methods. We assume that it is possible to develop such methods. To verify this assumption, we have created the *EvalCPLX_INFIXByObjArrayStack* (CM5) and *EvalCPLX_INFIXByLinkedListStack* (CM6) methods that use the same ADTs as the CM2 and CM3 methods, but do not convert the input infix expression to postfix by calling the *InfixToPostfixCPLX* method. To verify this assumption, we have also created the *EvalCPLXPostfixByArrayStack* (CM1) method, which evaluates a postfix expression, works very

similar to the CM2 method, but does not use the object of an ADT, and the *EvalCPLX_INFIXByArrayStack* (CM4) method that directly evaluates an input infix expression, works very similar to the CM5 method, but does not use the object of an ADT.

To verify all these assumptions, we carried out an experiment using our Android application, the Calculator. This application was running on the Android 9.0 Pie emulator, and the emulator on the Microsoft Windows 10 Home operating system. The experiment was performed on the computer that was equipped with the following basic hardware:

Intel Core i7-4700MQ (6MB Cache, 2.40 GHz, 5 GT/s Bus Speed, 4 Cores, 8 Threads), RAM: 8 GB.

Our Android application includes all the CM1, CM2, CM3, CM4, CM5, and CM6 methods. Using these methods, the Calculator can evaluate expressions with complex numbers, accurately measure the execution times of each method, and store them into the *LogFileAndroidApp.txt* disk file. During the experiment, the Calculator evaluated each of the 24 expressions using its own CM1, CM2, CM3, CM4, CM5, and CM6 methods and stored all measured execution times into the *LogFileAndroidApp.txt* disk file. 24 expressions with complex numbers that were evaluated by Android application particular methods were created as follows: the E2 expression contained 2 complex numbers, the E3 expression contained 3 complex numbers... the E25 expression contained 25 complex numbers with any number of pairs of parentheses. The E25 expression is shown in an input-output text box of the Calculator in Fig. 15.

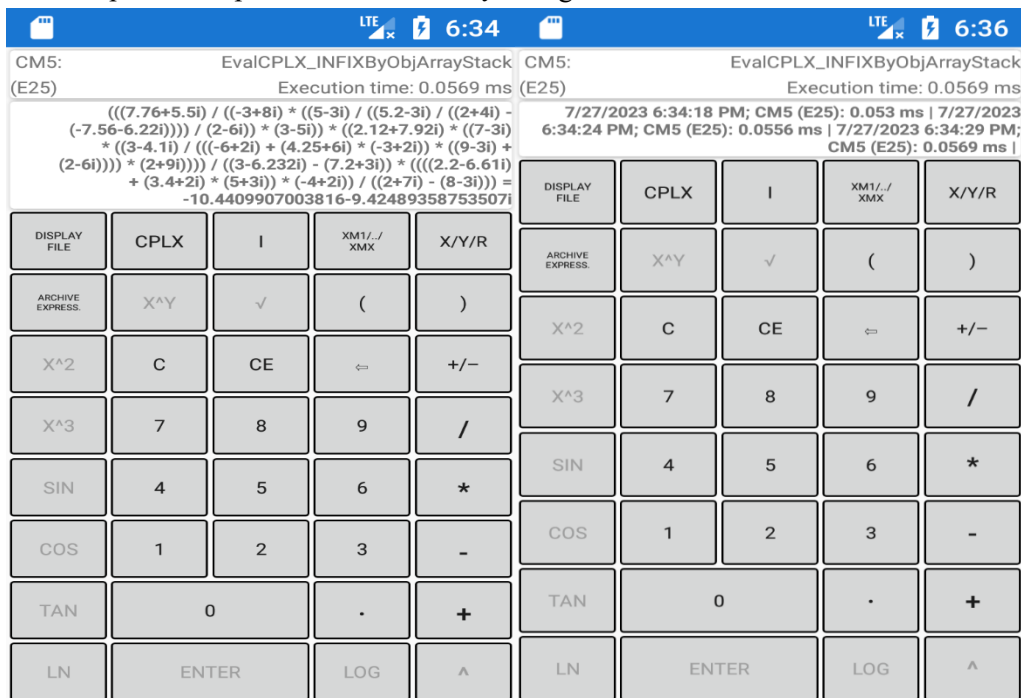


Figure 15. The Calculator showing the result of the E25 expression evaluation by the CM5 method (left) and the execution times from the 'LogFileAndroidApp.txt' disk file (right)

Execution times of these particular methods evaluating these expressions with complex numbers are shown in the following graphs.

6. Discussion

Several facts result from a comparison of the execution times of the particular methods (Fig. 12). The use of the array implementation of the stack ADT in the CM2 and CM5 methods is more execution efficient than the use of the linked-list implementation of the stack ADT in the CM3 and CM6 methods. The difference in execution efficiency between the array and linked-list implementation of the stack ADT seems to be more obvious with the increasing number of complex numbers in the expression is being evaluated (approximately from the number of 18) (Fig. 13). These facts confirm the main assumptions of our research. From the results of the experiment it is also clear that the CM4, CM5, and CM6 methods that evaluate the input infix expression directly and do not use the *InfixToPostfixCPLX* method to convert

this expression are more execution efficient than the CM1, CM2, and CM3 methods, which first convert this input infix expression to postfix by the *InfixToPostfixCPLX* method and then they evaluate it.

The most execution effective method is the *EvalCPLX_INFIXByArrayStack* (CM4) method, this is more visible from the number of 17 complex numbers in an expression. This method evaluates the input infix expression directly and does not use an ADT. However, at the same time, we have to add that the differences in the execution times of the CM4 method and the second most execution efficient the CM5 method, which uses the object of an ADT, are very minor, from 0.0003 ms to 0.0026 ms (Fig. 14). In addition, since the CM4 method does not use an ADT, it has to perform all operations with operands onto the stack by its own code. Therefore, it has more expansive and more complicated source code that is harder to modify and worse to maintain during developing newer versions of the Calculator than shorter and simpler the CM5 method source code.

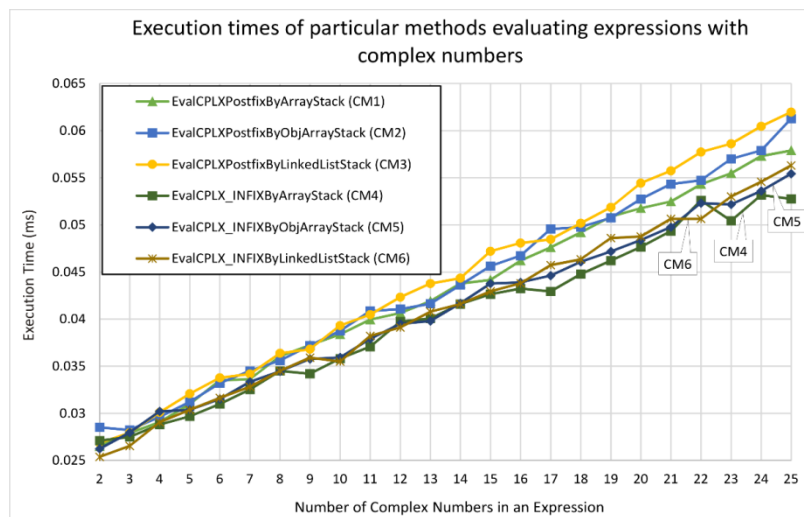


Figure 12. Execution times of Android application particular methods evaluating expressions with complex numbers

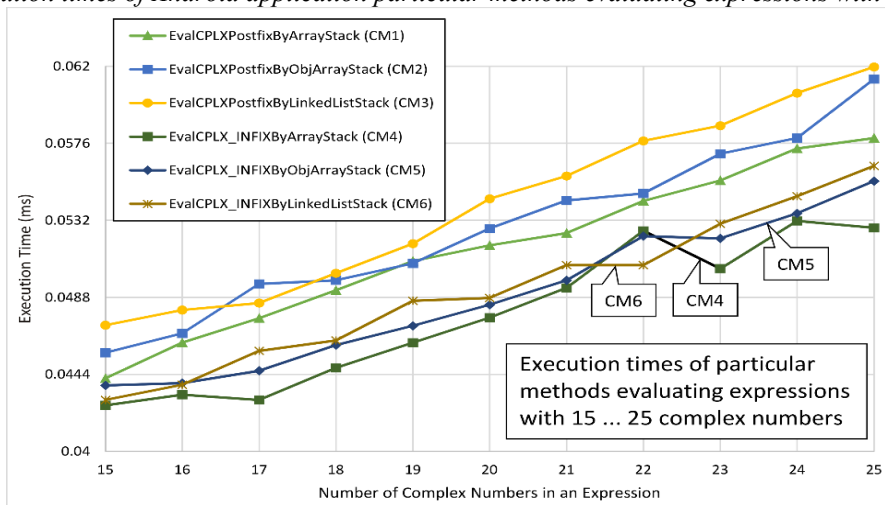


Figure 13. Execution times of Android application particular methods evaluating expressions with 15...25 complex numbers

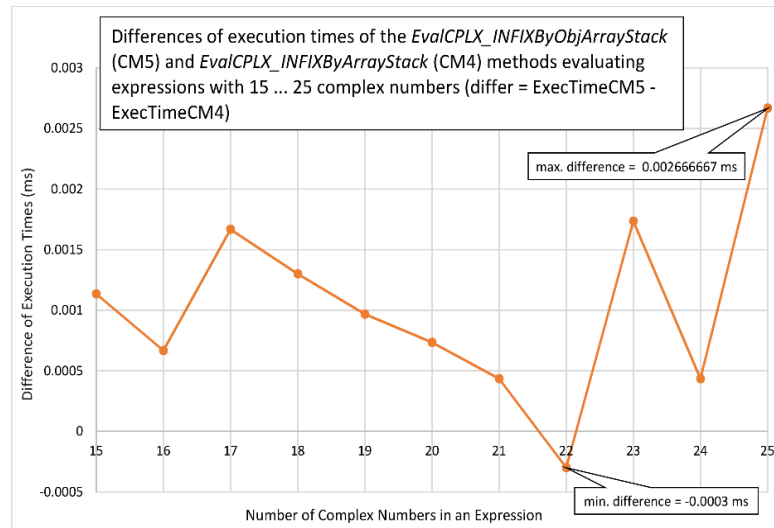


Figure 14. Differences of execution times of the 'EvalCPLX_INFIXByObjArrayStack' (CM5) and 'EvalCplX

7. Conclusion

The results of the experiment have confirmed our basic assumption. The use of the array implementation of the stack ADT in the Android application methods evaluating expressions with complex numbers is more execution efficient than the use of the linked-list implementation of the stack ADT in other methods of the same Android application evaluating the same complex expressions (Fig. 12). The difference in the execution efficiency of both implementations is more evident when expressions with a larger number of complex numbers is evaluating, from the number of 18 numbers in our case (Fig. 13).

It is also evident from the results of the experiment that the CM4, CM5, and CM6 methods that do not use the *InfixToPostfixCPLX* method to convert the input infix expression to postfix are more execution efficient than the CM1, CM2, and CM3 methods that perform this conversion by this method (Fig. 12). This effect is more evident from the number of 16 complex numbers in the expression (Fig. 13). The most execution efficient method in our experiment, although with minimal differences in execution times from the second most efficient the *EvalCPLX_INFIXByObjArrayStack* (CM5) method, is the *EvalCPLX_INFIXByArrayStack* (CM4) method (Fig. 14), which evaluates the input infix expression directly using the stack implemented by a simple array of the *cplx* structure type variables, without the use of an ADT and without the use of the *InfixToPostfixCPLX* conversion method.

Our conclusion is: if we need to evaluate expressions with a larger number of complex numbers in an Android application, and from this reason to work with a large stack, also if we want to evaluate these expressions with high execution

efficiency and if we also want to allow an efficient maintenance and expansion of this application, then it is advantageous to use the array implementation of the stack ADT in the methods of this application and evaluate infix expressions directly in them (the CM5 method in our case). However, if we want to evaluate expressions with a smaller number of complex numbers in such an Android application and if we also want to allow an efficient maintenance and expansion of this application, then it is advantageous to use the linked-list implementation of the stack ADT in methods of this application and evaluate infix expressions directly in them (the CM6 method in our case). The linked-list implementation of the stack ADT works with memory very efficiently, however, it has execution efficiency slightly less than the array implementation of the stack ADT.

References:

- [1]. Wali, A., Alfrihidi, M., Alasiri, N., & Alsabei, N. (2023). Aawn: An Interactive Mobile Application for Improving the Communication Skills of Arab Children with Autism. *TEM Journal*, 12(3), 1307-1315.
- [2]. Alkhuraiji, S. (2020). Design and Implementation of an Android Smart Parking Mobile Application. *TEM Journal*, 9(4), 1357-1363.
- [3]. Guttag, J. (1977). Abstract data types and the development of data structures. *Communications of the ACM*, 20(6), 396-404.
- [4]. Girard, J. F., & Koschke, R. (2000). A comparison of abstract data types and objects recovery techniques. *Science of Computer Programming*, 36, 149-181. Doi: 10.1016/S0167-6423(99)00035-0.
- [5]. Eliëns, A. P. W. (2000). *Principles of Object-Oriented Software Development*, (2nd ed.). Harlow, England: Addison-Wesley.

- [6]. Fürst, A., Hoang, T. S., Basin, D., Sato, N., & Miyazaki, K. (2016). Large-scale system development using abstract data types and refinement. *Science of Computer Programming*, 131, 59-75. Doi: <https://doi.org/10.1016/j.scico.2016.04.010>.
- [7]. Zhong, Y., S. Ishizuka, S., & Enari, R. (1988). Integrating abstract data types with object-oriented programming by specification-based approach. *Proceedings. 1988 International Conference on Computer Languages, Miami Beach, Florida, USA, October 9-13, 1988*, 202-209. Doi: 10.1109/ICCL.1988.13066.
- [8]. Agostini, D. (2017). *Implementing a stack using a linked list data structure*. Agostini.tech. Retrieved from: <https://agostini.tech/2017/01/04/implementing-a-stack-using-a-linked-list-data-structure/> [accessed: 18 October 2023].
- [9]. Rajput-Ji. (2023). *Implement a stack using singly linked list*. Geeks for Geeks. Retrieved from: <https://www.geeksforgeeks.org/implement-a-stack-using-singly-linked-list/> [accessed: 18 October 2023].
- [10]. Duggempudi, A. R. (2023). *Implementing a stack using an array and linked list*. Iq.opengenus. Retrieved from: <https://iq.opengenus.org/implement-stack-using-array-and-linked-list/> [accessed: 22 October 2023].
- [11]. K Hong. (2023). *Algorithms - stack data structure*. Bogo To Bogo Retrieved from: <https://www.bogotobogo.com/Algorithms/stacks.php> [accessed: 03 November 2023].
- [12]. Sedgewick, R. (1998). *Algorithms in C++ parts 1-4, Fundamentals, Data Structures, Sorting, Searching*, (3rd ed.) Addison-Wesley Publishing Company, Inc.
- [13]. Sedgewick, R. (1998). *Algorithms in C parts 1-4. Fundamentals, Data Structures, Sorting, Searching*, (3rd ed.). Addison-Wesley Publishing Company, Inc.
- [14]. Dale, N., & Walker, H. M. (1996). *Abstract Data Types: Specifications, Implementations, and Applications*. Jones & Bartlett Learning.
- [15]. Dale, N. (2003). *C++ Plus Data Structures*. Jones and Bartlett Publishers, Inc. Sudbury, Massachusetts, Boston Toronto London Singapore.
- [16]. Mokhithi, M., & Shock, J. (2020). *Introduction to complex numbers*. Open. Uct. Retrieved from: <https://open.uct.ac.za/server/api/core/bitstreams/a076364f-b68d-4685-922f-0245d9f248fb/content> [accessed: 07 November 2023].
- [17]. Andreescu, T., & Andrica, D. (2014). *Complex Numbers from A to ... Z*. Springer Science+Business Media New York. Doi: 10.1007/978-0-8176-8415-0.
- [18]. Craats, J. (2022). *An introduction to complex numbers*. Staff. Retrieved from: <https://staff.fnwi.uva.nl/j.vandecraats/ComplexNumbers.pdf> [accessed: 07 November 2023].
- [19]. Bartsch, H. J. (1987). *Mathematische Formeln*. VEB Fachbuchverlag, DDR-7031 Leipzig.
- [20]. Reade, J. B. (2003). *Calculus with Complex Numbers*. London and New York: Taylor & Francis.
- [21]. Roy, S. C. (2007). *Complex Numbers, Lattice Simulation and Zeta Function Applications*. Woodhead Publishing.
- [22]. Petzold, Ch. (2016). *Creating Mobile Apps with Xamarin.Forms*. Xamarin, Inc., Microsoft Press.
- [23]. Microsoft Corporation. (2022). *Stack<T> class*. learn Microsoft. Retrieved from: <https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1?view=net-7.0#code-try-4> [accessed: 09 November 2023].