Oxford Brookes University

# A formally proven data structures library intended for Formal Methods teaching

P00998 - MSc Dissertation

Supervisor: Ian Bayley

Alexis Chevalier (15055343)

September 28, 2016

# Acknowledgments

I would like to thank my supervisors, Ian Bayley, for the support and the helpful answers he gave me throughout my dissertation and for the very interesting lectures and advices I received from him and David Lightfoot on Formal Methods which motivated me to choose this subject for my dissertation.

I am also grateful to Chris Cox, for his lectures on academic writing and other subjects, and his kind advices regarding my writings in this dissertation.

Finally, I would like to thank my father, Jean-Marc Chevalier, for this advices on my dissertation and my whole family for their unconditional love and support throughout my life and studies.

# Abstract

Formal methods are mathematically based validation techniques used in software engineering to prove the validity of a given system towards a specification. They are currently taught at Oxford Brookes University to MSc students in Computer Science and Software Engineering. In this dissertation we design, specify and implement a formally proven data structure library intended to be used as a teaching material for the Formal Methods module given at Oxford Brookes University. We base our implementation and design choices on a literature survey of the fields of Formal Methods and a specific subset related to the module currently taught at the university, their presence in the education and the data structures and algorithms. The Dafny Language has been chosen as our specification language for the library and a specific survey is also proposed. The final version of the library provides 10 different data structures and algorithms, 9 of them are fully proven and successfully tested, however, the last one (a HashMap) is only partially proven. Our library provides a large set of detailed and commented examples of use cases for Formal Methods and we strongly believe that it will be very useful for students to have a better understanding of the Formal Methods and how to use them.

In this report we introduce our research and its relation with our work on the design, specification and implementation of the library, the methodologies used and the technical approach is then discussed before explaining the results and the testing process undertaken to assess our work. Finally a critical discussion of the project is proposed along with a conclusion and some recommendations for future work.

# Contents

# List of Figures



# List of Tables



# Listings

# 1 Introduction

Formal Methods are a set of mathematically based verification system used, in our case, for software engineering projects. They have been applied for many years in the industry and in various projects and domains. J. Woodcock, P. Larsen, J. Bicarregui et al. made a survey picturing the usage of those methods in [1], they concluded that a large number of various techniques are used by different teams with a rather positive outcome regarding the project quality. In order to expand their usage to a broader panel of software engineers, an important point is to provide an efficient training to students attending software engineering courses, including lectures explaining the fundamentals of those methods, and teaching materials like a library of verified programs. In this project, we will design, specify and implement a verified data structures library intended for formal methods teaching. Data structures are usually well-known by students taking software engineering courses, which makes it a useful example for them to understand formal methods.

This report will describe the process followed to achieve the goal of providing a useful library for educational purposes. The literature regarding the main fields related to the subject (Formal methods and the specific method chosen for the implementation, their presence in education, and data structures) will be reviewed and analysed before discussing the methodology applied to the project itself. The whole process will be detailed, from the requirements to the implementation followed by the evaluation of the results and the testing process. Finally, a critical evaluation of the work achieved throughout the project will be proposed before the conclusion.

# 2 Research

## 2.1 Subject overview

During this project, the study of formal methods will focus on Behavioural Interface Specification Languages since Spec#, which is taught at Oxford Brookes University, falls into that category [2]. The first part of the study will consist in a survey of this category in order to find a suitable language for the implementation of the library (Spec# is not necessarily the most suitable since the community is no longer active as visible on their website: http://specsharp.codeplex.com/discussions). Also, Oxford Brookes University provides a short paper describing the module of Formal Software Engineering (P00401) [3], including the expected learning outcomes and the syllabus of the module, it is important to fully understand the course goals and key points. In a specific part, a detailed survey of the literature discussing the chosen language, Dafny, will be realized in order to provide an overview of the research and the capabilities of the language.

Since the library has an educational purpose, it is important to take in account the existing work in the field of formal methods in education in order for the library to contribute efficiently to the courses using it. Many different aspects can be relevant, from the known points to improve [4] to the directions given by experts in the field regarding support material for teaching [5], we will go through the literature discussing this field and draw useful conclusions for our project.

Implementing a data structure library will give the students and the lecturers larger examples (multiple files and classes) to practice in the classroom, also data structure can be seen as a great subject for a formally specified library since some of them are maintained by 'invariants' [6]. Invariants are rules defining the state of the structure, as long as they are respected, we can assert that the data structure is in a good state and works correctly. Since invariants are, in a way, operation contracts describing how the structure should interact with the external interfaces, it is similar to the 'Design by contract' system described in [7]. This system is used in many Behavioural Interface Specification Languages [2]. A data structure library implemented using one of those languages could then be a very suitable support resource for formal methods teaching, it is therefore important to study

the data structures and their usage in software libraries in particular in order to design a useful library.

## 2.2 Literature review

### 2.2.1 Formal methods and Behavioural Interface Specification Languages

Formal methods have been the subject of many research projects and used in various industrial projects. Applied to the field of software engineering, J. Woodcock, P. Larsen, J. Bicarregui et al. defines them as a set of methods using mathematical models to improve and ensure the quality of a given software engineering project through analysis and verification [1]. In the same paper, they made a survey of the usage of formal methods in the industry, pointing out that they are used in various domains (transport, financial, defense, etc.) on various project types (real-time, distributed, big data, etc.) with different kinds of methods (Model-based, refinement, proofs, etc.). Their survey revealed a rather positive outcome on the project quality and a medium outcome on the duration and cost of the projects.

As mentioned earlier, our project has for context a module of Formal Methods taught at Oxford Brookes University [3] which uses Spec# as specification language for the part that concerns the data structure library. That is the reason why we will restrict the scope of the literature survey to the Behavioural Interface Specification Languages, a category of formal methods described by J. Hatcliff, G. Leavens, K. Leino et al. as languages providing code-level annotations used to describe the intended behaviour of the program [2]. Spec# [8] is considered as a member of this category along with other languages like JML [9], Dafny [10] and Spark. In [2], they introduce the main languages and their common features like the pre/post conditions, program termination conditions, frame conditions, etc. through different code examples using, turn by turn, the various languages to illustrate the different points. Some of the authors are also contributors on the mentioned languages, K. Leino is one of the lead researchers on the Dafny project and previously on the Spec# while G. Leavens contributed to the JML project, they also wrote extensively about their work and research in the field. They concluded that those languages are an interesting trade-

off between their capabilities to specify a given program, complex or not, their simplicity which is important for students or programmers with a small mathematical background and their accessibility since they are accessible through some automated verifiers and can therefore be integrated into large projects.

On a technical level, many papers have been written about the particularities of the Behavioural Interface Specification Languages, they all follow the same structure and describe the language feature by feature with code samples used to illustrate the main points. Dafny has been extensively discussed in several papers, including [11], [12], [10] and [13], JML in [9] and [14] and Spec# in [8] and [15] (we voluntarily discarded Spark since it concerns Ada, which is not a commonly used language at the MSc level). Most of the languages share a lot of common features at the verification level. However some differences were identified at different levels. Syntactically speaking, JML is close to the Java language and is actually usable only through Java comments in the Java code (or in an external file), Spec# is a layer of the C# language and borrows some parts of its syntax, Dafny is the only one to be a completely new language designed from scratch for verification purposes, it also borrows some syntax elements from C# but also from some functional languages for the verification parts. Since Spec# and JML are parts of an existing programming language, the programs can be compiled to an executable; Dafny offers a compiler which translates the verified Dafny code to usable .NET code which can also be executed. As noticed by K. Leino and R. Monahan in [16], the support of generic types and object oriented features is limited in Dafny and prevents the implementation of some particular cases like a generic sorting algorithm for instance, this is not the case with JML and Spec# since their respective languages (Java and C#) have a great support for this. Dafny proposes an alternative to the lack of strong inheritance features, the traits, a way to share common behaviour between classes which has been described by R.Ahmadi, K. Leino and J. Nummenmaa in [13].

We understand that the formal methods are a large and continuously growing field, we decided to narrow our research to a particular category of techniques which seems the most suitable ones for our main goal: a teaching material for the Oxford Brookes University module. Our research on this category helped us to define a set of potential languages for

the library, but also to understand the concepts behind Behavioural Interface Specification Languages in order to implement the library in an efficient way.

### 2.2.2 Formal Methods in Education

In order to fully understand the context of the project, we need to research about the current situation of Formal Methods in Education. A survey made by V. Almstrum, C. Dean, D. Goelman et al. [4] shows that the presence of the subject in education is limited, only some software engineering courses are teaching the subject, and sometimes the subject is only partially or tangentially taught as mentioned in the survey under the term of 'semi-formal methods'. This paper was written as a report from the ITiCSE 2000 Working Group of Formal Methods Education, and includes a list of proposed improvements themes in the domain of Formal Methods that were gathered by the working group members through their experience. A large part of those themes directly concerns the tools and how they block or improve the student's experience while using them, they explain that tools could be improved with a better user experience and some of them could propose a better educational support in order not overwhelm the student with features. S. Liu, K. Takahashi, T. Hayashi et al. describe a teaching paradigm for formal methods in [5] and they also recommend simplicity along with visualization and preciseness for formal methods in education in order for the students to be interested and motivated by such modules.

Since formal methods are a very large field, it is important to provide the necessary support and the convenient modules to the students wishing to take this subject, in [17], G. Tremblay discusses the position of Formal Methods in a university cursus, whether it should be part of a computer science, mathematics or software engineering course. From his experience, he provides some related subjects that could be useful depending on the student's cursus and which part of the formal methods subject could be the most relevant.

Finally, the industry has a role to play in the teaching of formal methods, T. Mailbaum explains in [18] that the software engineering industry currently focuses on computer skills more related to a technician than to an engineer by putting a higher importance on skills as knowledge of programming languages than to mathematical or analytical abilities. If the industry were to change its priorities in terms of software engineering project management

and recruitment, the subject of Formal Methods could gain in popularity and therefore in better tooling.

The presence of Formal Methods in the education is not currently very bright, and the industry is not promoting them a lot currently. It is interesting to understand which points are important and that it is possible to improve the quality of the materials through better tools providing a better user experience and precision. We will do our best to apply those guidelines while working on the library in order to provide a great teaching material for Formal Methods education.

### 2.2.3 The Dafny Language

Following the studies made in the sections 2.2.1 and 2.2.2, the Dafny language has been chosen as our specification language, a detailed explanation of the choice is discussed in 3.6.3. Dafny is presented by K. Leino in [10] as an imperative, sequential language supporting basic object features and dynamic allocations. The verifier uses an SMT (Satisfiability modulo theories) based solver, which means that it will make an attempt to prove a given specification using a fixed set of decision procedures. A schema picturing the system realized by L. Herbert, K. Leino and J. Quaresma in [11] is available in figure 1.

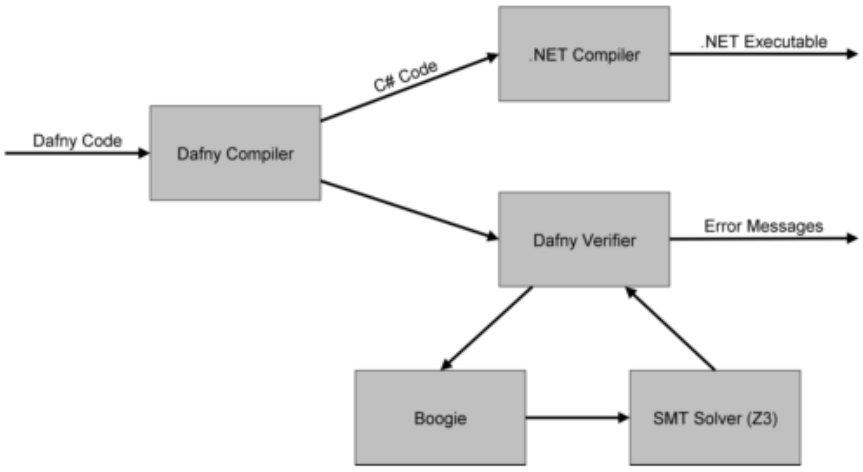



Figure 1: The Dafny system.

They explain that the Dafny compiler first goes through the verifier, which uses *Boo-*

*gie 2*, an intermediate verification language, to generate first-order verification conditions. Those conditions are then passed to the SMT solver, *Z3*, which tries to prove those conditions based on a defined set of decision procedures as mentioned earlier. If the SMT solver is unable to prove one or multiple conditions, it will return errors to the Dafny compiler. Dafny also compiles the Dafny code to the .NET platform and generates a .NET executable. It is also possible to generate a DLL (Dynamic-Link Library), which can be used to call Dafny methods from other .NET programs.

Through different papers, the feature of Dafny are described in a detailed way, K. Leino, L. Herbert, and J. Quaresma give a large overview in [10] and [11]. Starting with the pre and post conditions, inspired from languages like *Eiffel*, *JML* and *Spec#* and designed to specify the requirements for the caller (preconditions) and the responsibilities of the callee (post-conditions). The ghost variables are also explained, they are variables which are only accessible from the verification, which allows Dafny's verifications not to impact the performance of the final implementation. The authors explain that Dafny provides some built-in data structures like the sets, multi-sets, sequences and maps, which can be helpful when specifying a particular data structures.

Dafny provides loop invariants and termination conditions [11] through the keywords *invariant* and *decreases*. As the pre/post-conditions, they expect an expression supposed to be true for each iteration of the loop, the termination condition expect that the expression decreases at each iteration. The language also supports functions (methods with a single return type and containing a single mathematical expression), usually only used on the verification side, they are, by default, like ghost variables and cannot be accessed from the implementation except if the developer used a 'function method'. Some particular subtypes of functions are available, like the predicates which can only returns a boolean.

Another important part of a verification language concerns the modifications made by the methods, this is called the 'frame problem' and described by I. Kassios with the question 'when formally describing a change in a system, how do we specify what parts of the state of the system are not affected by that change?' [19]. Dafny uses a system called dynamic frames which is described in [19] and [20], it involves of a set containing all the objects that the method is allowed to modify, this set is then specified through the *modifies*

and will be used to ensure that the method only modifies the elements in this set.

Another way of specifying a system through abstract and refined modules is available with Dafny, J. Koenig and K. Leino introduce the concept in [21] through multiple examples of refined and abstract systems made with Dafny, they explain the methodology and the usefulness of such a system.

Dafny has been compared to other similar languages like JML and Spec# about their features and design by K. Leino in [10], and a benchmark on the performance and some weak points of Dafny has been made in [16], they concluded that Dafny is a decent verification language, it has a weak object oriented support when compared to JML and Spec# but offers a more precise set of verification features and manages to prove complex specifications relatively quickly. This survey of the work made on the language will be a great help during our work on the library in order to have a better understanding of the proof systems and how to use Dafny's features and syntax.

### 2.2.4 Data structures and their usage

Data structures are used to store data in a way that optimizes their treatment by a computer as defined by C. Leiserson, R. Rivest, T. Cormen et al. in [6]. Their book, Introduction to Algorithms (Third edition), is considered as a standard in the field of data structures and algorithms (all the editions combined have more than 3000 citations according to the ACM website). They introduce a large set of structures and algorithms while covering a large part of the field (Sorting algorithms, Algorithm design and analysis, Graph algorithms, Linear programming, etc.). They also present some side topics like the mathematical background required for the algorithms in the book, the asymptotic growth of functions or the divide and conquer principle. In our research we were mainly interested by the data structures, they introduce multiple ones (stacks, queues, trees, sets, etc.) by providing a detailed explanation including pseudo-code, mathematical explanations and a list of exercises for every chapter. R. Sedgewick and K. Wayne propose a similar content in Algorithms (Fourth edition) [22] with a different approach, they provide a more practical content including more code samples to illustrate the algorithms, along with schemes and less mathematics.

The two books present the state invariants (if applicable), the rules that must be valid at any times if the implementation is valid, for each data structure and algorithm. Those invariants guarantee the validity of the system's state.

The concept of state invariants can be summarized are a set of rules in a contract that specifies what should be modified and what should not be modified in a given system. This concept can be related to the notion of design by contract introduce by B. Meyer in [7]. R. Sedgewick et al. even refers to this notion in [22] when describing some elements of the Java language used to verify the correctness of a system, like the assertions, for instance.

In industrial programs, data structures are rarely implemented in their raw forms, programmers tend to used object oriented implementations like a Queue or a List for instance. A raw linked list is less useful than a Queue because the Queue hides the linked list implementation and provides useful methods directly for the programmer to use, B. Long gives some concrete examples of such interfaces in [23]. The documentation of common programming languages like Java [24] and .NET (C#/VB) [25] also describe their own interfaces for the data structures they provide, we can see some similarities between those interfaces and the ones provided by B. Long.

Data structures are essential to many programs and we saw that they can be easily related to a formal specification through Design by Contract, a formally proven data structures library is therefore a great subject for the project. It will be important to choose convenient interfaces and to correctly implement the state invariant for the project to be useful though.

## 2.3 Existing approaches related to the project and relevant software

At the time of writing, we were not able to find a similar project (a data structure library proven with some kind of formal method) during our research which is the reason why we extended our research to a broader set of projects. We inspired ourselves from common languages data structures library for our programming interfaces like those from C# and Java.

On the formal methods side, multiple examples of approaches were discovered during the literature review: A Dafny implementation of the Schorr-Waite algorithm has been proposed by K. Leino in [10], multiple data structures and algorithms specified with Dafny have been proposed by K. Leino and R. Monahan in [16], some examples of inheritance in Dafny with traits were discussed by R. Ahmadi, K. Leino and J. Nummenmaa in [13] and an example of a Queue has been introduced by K. Leino in [20]. Many other examples are available in the literature discussed in the previous literature review and we inspired our work from those.

## 2.4 Discussion of relevant professional issues

Since this project has an educational goal, some ethical points are important and should be carefully treated. The first issue concerns the neutrality of the choices made during the project, the main choice being the specification language for the library, it is important not to make a personal choice for a language over another one, the choice should be strongly justified by comparisons based on quality sources. To ensure that this point hasn't been treated lightly in our project, we first made a literature survey of the existing Behavioural Interface Specification Languages in 2.2.1 in order to define a limited set of languages to compare. A comparison has then been made, and is detailed in the section 3.6.3 of this report. Finally, a literature survey of the Dafny language has been made in section 2.2.3 to validate the previously made choice and to help us during the implementation.

A second but smaller issue concerns the correctness of the library, it is important for the students that the library doesn't show bad or invalid examples in order to be a useful support material, we believe that this issue will be prevented thanks to the use of formal methods, however, a bug could exist in both the specification and the implementation, that is the reason why we added a testing phase including a test case in order to put the library in a real use case.

# 3 Methodology

The project was completed in three main phases (Preparation, Implementation and Reporting). The first phase was mostly composed of the requirements elicitation, the initial research, the feasibility study and the writing of the proposal report, during this phase, the work time was shared between the analysis of existing solutions and academic literature, and writing the proposal report while defining the project more and more precisely. At the end of the first phase, the project was nearly completely defined and a large part of the literature review was achieved. The second phase was the longest and the most important one, the library design, specification and implementation were made during that part, also, the literature review was completed during this phase. The final report had not been started at this time, but many notes were taken during the work on the library and the research. Finally, the last phase was dedicated to the reporting and the final testing, the final report and the short paper were completed during this part, using the many notes taken during the previous ones. Even if most of the testing was made during the second phase, some particular cases were tested once the entire library was completed.
In this section we will detail the initial project requirements and analysis, the important choices that were made and the design, specification and implementation will be discussed.

## 3.1 Requirements

The requirements have not changed much since the proposal report, except for some rewordings and a modification regarding the time and space complexity of the algorithms, a detailed report of the work done will be made in the Results section. The time an space complexity of the implementations will be less tested than initially planned because of a lack of time, only a quick analysis will be made. We believe that this is enough for the current scope of the project since the main goal is to help students understand the formal methods, not to optimise algorithms. It is important to know that the requirement have not been updated with the choice of Dafny as the specification language, because some functional requirements about the language are important to explain this choice.

The non-functional requirements are listed below:

1. The data structures library must include the following data structures : Stack, Queue, TreeMap, HashMap
2. The data structures library source code must be clean, extensible and commented
3. The data structures library must be entirely verified using the program verifier for the chosen specification language

The following functional requirements are listed below:

1. The data structures library must be developed using a specification language
2. The chosen specification language syntax must be close to a commonly used programming language (object oriented like C#, Java, etc... and/or procedural like C)
3. The correctness of the data structures library must be verifiable using an external tool
4. The complexity of the implemented algorithms and data structures must match the current industry standards

A few extended requirements were also proposed:

1. The data structures library should be compilable and/or reusable into other projects
2. The data structures library should include other algorithms and data structures if they are relevant
3. A library usage guide should be provided
4. A specification language configuration guide should be provided

## 3.2 Constraints and limitations

The most important limitation was business-related, it was the time allocated to project, which helped to define the scope since the project has to been delivered on the 30th of September, which creates an important time-based limitation. This constraint led us to define a precise set of data structures to implement in the library, in order to have enough time to dedicate to every single implementation.
The second limitation was technical and discovered later during the project while working with the chosen language, Dafny. Even if we knew at the time of choosing the language that Dafny had a poor object oriented support, we didn't expected that it would completely prevent some generic implementations like the generic sorting algorithms. A mitigation measure was taken and is detailed in section 3.4.

## 3.3 Testing and assessment methods

For the data structure library source code, this project will feature a testing process different from classical software engineering projects since it involves the usage of formal methods. The first part of the testing involves the automated verifier provided by Dafny, before writing any test case, we want the verifier to accept the specification and the implementation. If it is not the case, we keep on working on the data structure using assertions and assumptions to understand the problems and correct them. When the verifier validates the data structure, we write a single test method (multiples one if deemed useful), using all the methods of the data structure interface in a different order along with assertions to ensure that we have written a specification that satisfies the expected usage. To summarise, the automated validation verifies the correctness of the program towards the specification and the test case verifies the correctness of the specification towards a real usage. In order to consider a data structure as tested and validated, we require both the automated validation and the test case to be valid.

The code quality requirements (cleanliness, extensibility and comments) are discussed with detailed explanations of the process followed through all the project is proposed in sections 3.8 and 3.7. Regarding the specification language, Dafny, a complete explanation

of the choice and how it meets the non-functional requirements is made in section 3.6.3. A quick analysis of the implementations (regarding the nested loops, recursivity, etc...) will be made to ensure that no major mistakes regarding the performance have been made.

Regarding the extended requirements, the two first ones are validated by design since Dafny can be compiled to a format usable as an external Library from some .NET programs, also, the List and the sorting algorithms have been added to the library design. The third one, about the usage guide has not been realised since a very good tutorial is available on the Dafny online IDE (http://rise4fun.com/Dafny/tutorial/Guide) and a paper has been written by K. Leino and V. Wüstholz about the Dafny IDE in [26]. Finally, the configuration guide has not been realised because of a lack of time, but the recently created Github page (https://github.com/Microsoft/dafny) provides many links and details about the subject.

## 3.4 Risk and issues analysis

Different risks were expected during this project, the first one being that the chosen specification language, Dafny, would not suitable for the project scope. The risk didn't occurred during the research and the initial design phase, however, it occurred later during the specification and implementation phases of the library. Dafny actually caused some problems related to the genericity and reusability of the data structures and algorithms, it was impossible to use generic comparable objects (for instance to compare generic objects in a sorting algorithm). The mitigation applied to that issue was simply to ignore it, we decided to make this choice because the first goal of the project is educational and not industrial, which means that the most important point is to give to the students a decent set of examples in order for them to have a better understanding of formal methods, the genericity of the implementation is therefore not an essential point.

Regarding the data structures library, a second risk concerning the difficulties encountered during to the specification and implementation of the chosen data structures, the risk wasn't expected at the beginning, but it occurred during the implementations of the HashMap and the TreeMap (the HashMap is not completely validated and we didn't man-

aged to provide a size() operation for the TreeMap, our critical thoughts for the cause is that we missed an important part of the specification in both cases). The mitigation plan here was first to try some workarounds to completely validate both the structures (Extended chaining and linear probing for the HashMap, and different specifications for the TreeMap), but it wasn't sufficient. The final mitigation measure was to provide another kind of data structure to the project, in order for the students to have enough materials. The List interface was chosen with an ArrayList implementation and a LinkedList mixed with key-value features were proposed (the linked list had a key-value system because it was supposed to be used as extended chaining for the HashMap, it helped but didn't completely solved the initial problem). Both of the List implementations are completely validated.

A third risk was detected, regarding the data structures and their implementation, it was important that they are correctly implemented in order to avoid showing a bad example to the students. At the time of writing, no errors were detected by the test cases.

Another risk concerns the fact that a project with an educational goal should not try to influence the students to use a specific tool or system without solid references and arguments. We were fully aware that this was a risk and we think that we avoided it. Dafny is the most important choice made during this project but it has been made after a deep literature review and a comparison between different solutions competing in the same category (the comparison is detailed in section 3.6.3). The other technologies and products mentioned during in this report are not as important as the language, and are purely personal choices by habit or preference.

Finally, the last risk, which we can unfortunately difficultly assess at the time of writing, is that the project doesn't provide enough support for the students or the lecturers. This could occur for instance if the language, Dafny, is too complicated for the students or is not close enough to a more classical language. We think that the only way to assess the quality of the project on this point is to use it in a teaching module.

## 3.5 Impact of the literature and the existing technology

The literature review led to multiple choices and modifications on the project at different levels, the first one being of course the specification language, which has been chosen after the literature analysis as explained in section 3.6.3. The coding style and standards (described in section 3.7) are also influenced from the literature since many papers features code samples from the Dafny contributors [11] [12] [13]. Finally, some data structures specifications were modified to be lighter and easier to understand following some good practices found in the literature like the usage of dynamic frames detailed in [19].

The existing technology also helped to shape the project to its current state, the most influential projects were widely used programming language (Java and C#), the interfaces they provide for their own data structures implementation were really helpful in order to determine which methods are the most useful for each data structure. The structural diagrams available in section 3.8 (see figure 4) are inspired from the languages and the organisation of the collections library found in common languages (http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html for Java and https://msdn.microsoft.com/en-us/library/mt654013.aspx for C# were the most used ones during the project). We thought it was a interesting way to create interest for the student towards the library by using some similarities with some common elements of a programming language they probably used during their experience as software engineers.

## 3.6 Methods, language and tools choices

### 3.6.1 Methods and planning

Since the project team was composed of a single developer, we decided to apply a custom project management strategy. The plan was to follow the initial Gantt chart (see appendix A) by implementing the library step by step while continuing the literature review and writing the report at the same time. The method was inspired from the Agile methodologies since it involves a continuous evolution and testing of the software and a task-based plan (using a Kanban board). The method was planned so that a writing time was allocated after each

implementation period.

Unfortunately the data structure library required much more time than expected on the Gantt chart, which led us to concentrate at nearly full time on the specification and implementations. This caused a deep change in the method, the continuous evolution system and the Kanban board were kept but the continuous writing of the final report was abandoned; only quick notes were taken before starting the next implementations. This caused a modification of the planning, replacing the small phase indented for the completion of the reports at the end by a larger phase used to combine all the notes taken the previous phases into the final reports. The implementation and specification phases were stopped in late august in order to leave enough time for the final reporting phase. An updated Gantt chart is available in appendix B and offers a more accurate representation of how the project was actually realised.

A Kanban board was used in order to have a clear overview of the project's advancement and state, the figure 2 shows the board at an early time in the project, the ”to do” column was used to list the next tasks to achieve (usually in the next two to three weeks), the ”ongoing” column contained the tasks that were currently in progress, and the ”done” column hold all the tasks that have been completed.

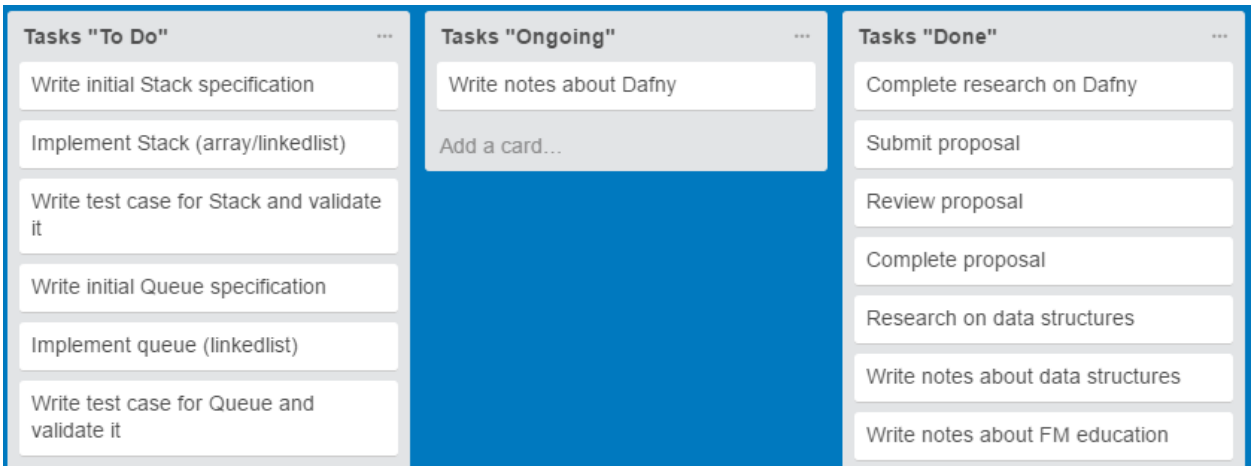



Figure 2: Kanban board.

### 3.6.2 Tools and resources

Different tools were used to complete this project, the following list gives a decent overview of them:

- Dafny (https://github.com/Microsoft/dafny/) as specification language and automated verifier
- ShareLatex.com (http://specsharp.codeplex.com/discussions) as online LaTeX IDE used to write the reports
- Git as versionning system used to manage the source code (Bitbucket (https://bitbucket.com) as provider)
- Trello (https://trello.com) as kanban board to have a clear view of the past, current and remaining tasks
- Visual Studio 2012 with the Dafny extension as our Integrated Development Environment
- Mendeley (https://www.mendeley.com), used as bibliography manager

Regarding the resources, all the papers and books mentioned in the literature review and/or in the references were used throughout the project in order to reach the expected goal, some particular websites were also used:

- Oxford Brookes Discover (https://www.brookes.ac.uk/library/resources/e-resources/discover/) and Google Scholar (https://scholar.google.co.uk/) as academic content gateways
- Rise4Fun Dafny Tutorial (http://rise4fun.com/Dafny/tutorial/Guide) as learning tool for Dafny
- The Dafny community website (https://dafny.codeplex.com/discussions) and the StackOverflow Dafny tag (http://stackoverflow.com/tags/dafny) for the community questions and responses

- The Dafny repository on Github (https://github.com/Microsoft/dafny/) for the issues and source code analysis

### 3.6.3 The specification language choice

After the initial literature review, three Behavioral Interface Specification Languages (See section 2.2.1) were selected based on their closeness to the Oxford Brookes University's module of Formal Software Engineering (P00401) course description [3] to be compared in order to choose the most suitable one for the implementation of the library, those languages were:

- Spec#, an extension of C# for API contracts offering several specification features and an automated program verifier [8]
- JML, a specification language for Java including a large set of specification possibilities and multiple automated tools (including a verifier) [14]
- Dafny, a specification language built for the .NET platform with specification in mind, it also includes an automated verifier [10]

At the time of writing, Spec# is currently used for teaching at Oxford Brookes University, the language syntax is close to common languages like C# or Java, has a decent object oriented support and even offers and online IDE in order to quickly validate a specification through a browser. The language provides many of the essential specification features (pre/post conditions, loop invariants, frames, etc...) and can be used on a computer, either through a command line interface or inside Visual Studio 2012. However, the project's community coordinator (K. Rustan M. Leino) described it, in a forum thread (https://specsharp.codeplex.com/discussions/569610), as and old project which is not actively researched anymore, he also suggests the use of Code Contracts (a tool included in .NET which provides basic verification features but no automated verifier) or Dafny, which will be detailed more later.
JML provides a syntax similar to the Java language, except that all the specification instructions have to be written inside comments in the Java file. Since the specification

describes the interfaces and behaviour of Java-compliant code, it offers all the features of the Java language including its object oriented features. JML offers the basic specification features and some specific one like specification-only variables and built-in data structures like sequences and sets, which can be helpful with some specification cases, also, thanks to the refinement features, it is possible to start with an abstract system and refine it later for implementation. Unfortunately, no online version of JML were found during this project, which limits the usage to a computer, either by installing a command line version, or with an Eclipse plugin (I didn't managed to get the plugin to work during the project under different configurations).

Dafny is the only one in the selection to be a complete programming language with specifications features at the core of its design. The syntax of the language shares aspects from object oriented languages (classes, traits, etc..) and functional languages (mathematical formulations, quantifiers, etc...). Dafny includes the basic specification features mentioned earlier, plus some advanced ones like JML (specification only variables and data structures, and refinement) Like Spec#, it provides a web-based IDE (http://rise4fun.com/Dafny) and can be installed on a computer using Visual Studio 2013 (The plugin has been tested and provides useful help including a debugger). Unfortunately, Dafny provides a light object oriented support (no interfaces nor classical inheritance), it only provides "Traits", which can be used to extend classes with methods, abstract methods or variables, however, those Traits doesn't support generic types at the time of writing. Based on this summary of the comparison that has been made and the research on formal methods in the education (see section 2.2.2), Dafny has been chosen as the implementation language since it is easy to use either from a web browser or a computer, it is an active project which keeps on being updated, and it provides a large set of useful and precise specification features, as demonstrated in [10] with the Schorr-Waite Algorithm specification (the specification is an interesting example of many features available in the language). During the literature review, a particular interest has been given to Dafny, as shown in section 2.2.3.

## 3.7 Coding Standards

It is important to define and use a common coding standard when working on a project, in this subsection, the main guidelines used during the implementation of the library will be described.

```
1  /*
2   * The class Stack
3   */
4  class {:autocontracts} Stack<T> {
5    //Implementation variables
6    var top: Node<T>;
7    var stackSize: int;
8
9      //Specification variables
10   ghost var Repr: set<object>;
11 }
```

Listing 1: Coding style for class declaration.

The first part that has been standardised in the project is the class definition, as visible in the listing 1, it is similar to many languages, the class name, generic type if any, and the opening curly bracket are on the same line, followed by a new line where the class content may start. Some properties can be included (:autocontracts in the example), they will concern all the class and are directives for the dafny verifier, this one specifies that all the method will require the predicate Valid() to be valid before and after every method call, if there is a predicate with the name Valid in the class. At the end of the class, the closing curly bracket must be put on a new line, alone. The class variables (and the static ones if required) will be declared at the top of stack, with a separation between the implementation ones (non-ghost) and the specification ones (ghost). Comments are mostly single-line to prevent an abusive usage, but if an explanation or a list require a multi-line comment, it is better to use them.

```
1  //Removes the current node and returns the value
2  method doSomething(node: Node<T>) returns (value: T)
3    requires node != null && this.stackSize != 0;
4    modifies this;
5    ensures node == old(node).next && value == old(node).value;
6    ensures this.stackSize == old(this.stackSize) - 1;
7  {
8    value := node.value;
9    node := node.next;
10   this.stackSize := this.stackSize - 1;
11 }
```

Listing 2: Coding style for methods.

The second most important point concerns the methods, the listing 2 gives an example with the doSomething(). The first line is the declaration, which including the name, input and output parameters. The next part is very different from classical languages, since it includes all the specification elements. A Tabulation will always be put before them in order to differentiate them from the declaration and from the implementation. The first elements are the pre-conditions, followed by the frame conditions and finally the post-conditions. In some specific case, when the method is recursive, some termination conditions (decreases keyword) will be added after the post-conditions. The opening curly bracket, marking the beginning of the implementation is put, alone, on a new line, then the implementation can follow, with one instruction per line (exception be made for multiple variable assignation/declaration). All methods are terminated with the single curly bracket on a new line.

```
predicate Valid()
  reads this, this.Repr;
{
  (this in this.Repr)
  &&
  (this.next != null ==>
    (
      (this.next in this.Repr)
    )
  )
  &&
  (this.next == null ==>
    (this.QueueContent == [this.value])
  )
}

function Size(): int
  reads this.Repr;
{
  |this.QueueContent|
}
```

Listing 3: Coding style for predicates and functions.

The last main elements used in the data structures library are the predicates and the functions, a function is only usable on the specification side (opposite of a method, but there is a way to mix them), and a predicate is a function that will always returns a boolean. Those functions and predicates can only return a single output variable, but accept many input variable, they follow the same specification rules than the methods for the pre, post, frame and termination conditions. In the listing 3, we show our way of using them, since functions and predicates can only be composed of a single expression (it can be very long), the best way to represent it is by breaking every sub-part of the expression on a different line and at a different indentation level when sub-parts are nested. the Valid()

predicate shows a very great example and it is easy to insert a single line comment in order to explain a complex part. The indentation system helps to understand the level of nesting for a better understanding. Every sub part must be put into parenthesis, we noticed multiple times that dafny handle the expression better this way.

There are many other elements in Dafny, but those are the main ones that have been codified for our project, we believe that they optimise the readability of the source code and help to insert comments, which is important for a project with an educational purpose.

## 3.8 Design process

The design process changed during the project, it was divided into two parts, an initial design step intended for the structural design and the first version of the data structures interfaces. The initial phase started during the writings of the intermediate report, the structural design pictured in figure 3 has been realised at this time. At this time, the specification and implementation of all the elements present in this schema was planned, and the proposed interfaces were inspired from the common programming languages Java and C# (as explained in section 3.5). During the design phase, the structure of the project was also defined, the schema shows large boxes containing the algorithms and data structures, ”Library” was supposed to be the main folder holding all the library, ”Data structures” and ”Algorithms” were intended for all the specification and implementation files, and ”Common” for all the shared behaviours and classes.

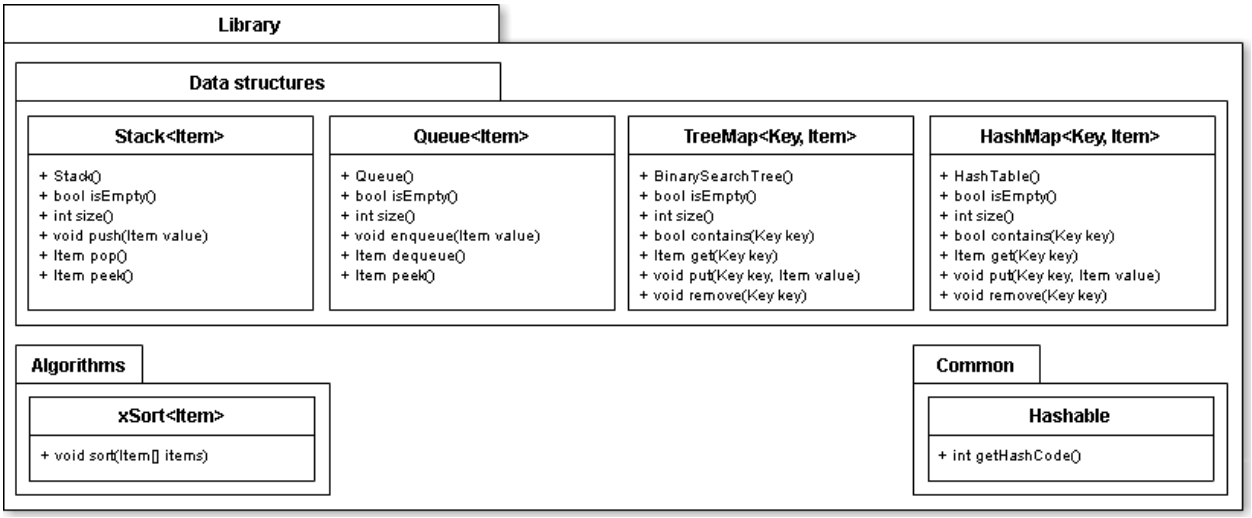



Figure 3: Initial structural design.

The second part was realised before starting every new specification in order to ensure that the interface initially proposed was still feasible and should not be modified. Before starting a specification, some tests were made along with some reading regarding the data structures and the required features of Dafny, if a problem was detected, then a mitigation measure was taken. For instance, we noticed that it would be impossible to specify a generic sorting algorithm using Dafny, we decided to simply provide the algorithm for a single data type (an integer). Later, some problems occurred during the implementation of the TreeMap and HashMap structures regarding the size calculation and the validation of the HashMap, we decided to remove the size element from those structures and to use assumptions to prove the HashMap, since this was a large alteration to the initial design, we decided to add the List interface to our design, in order to provide enough material for the students, that is the reason why the final structural design (visible in figure 4) differs from the first one, because the mitigation measures we took durign the project impacted the design of the library.

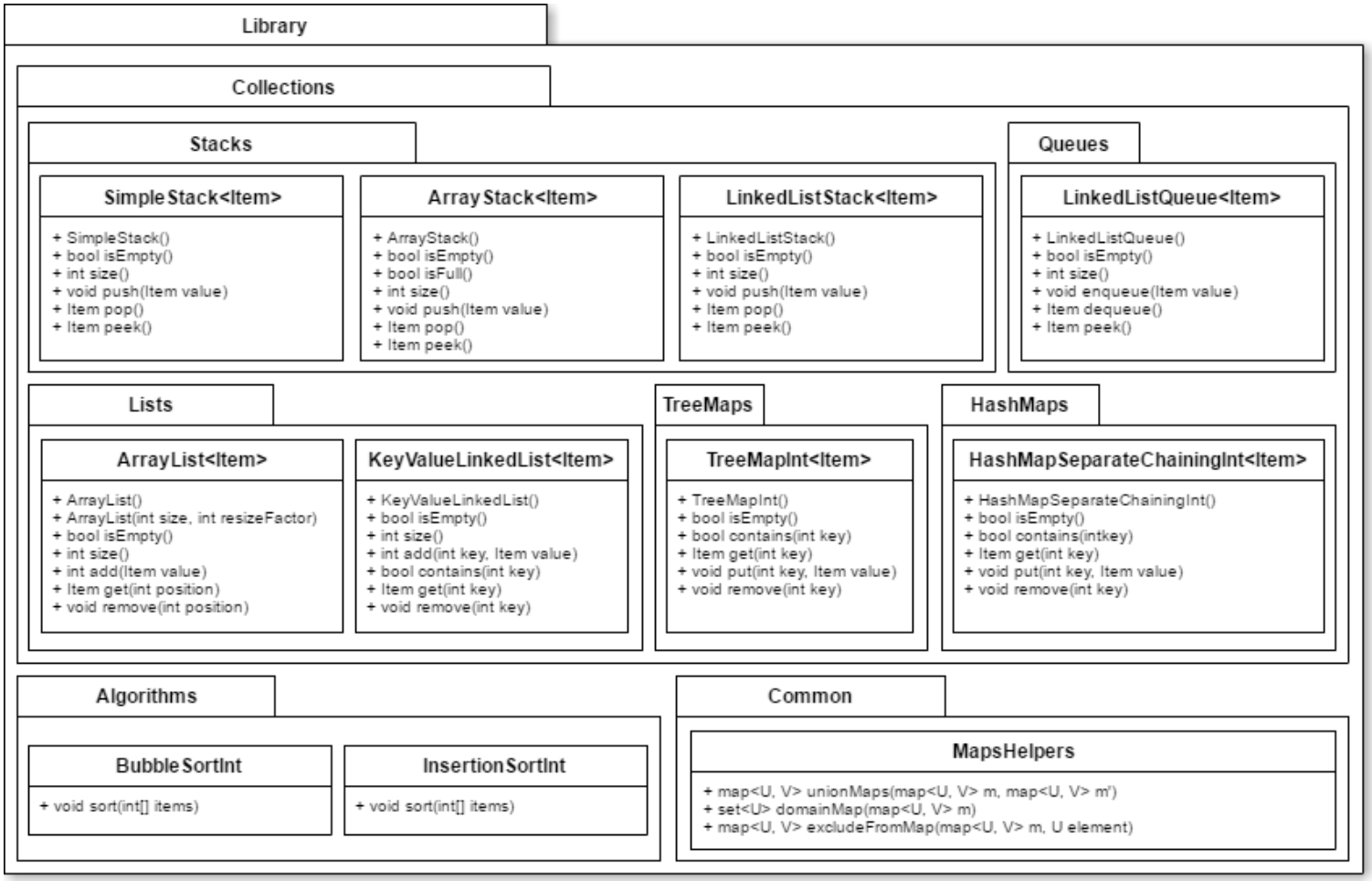



Figure 4: Final structural design.

The figure 4 pictures the final design of the library, it is clear that there is many differences between this version and the initial one (see figure 3). The interfaces and their methods haven't changed a lot, the List interface has been added to the library and an additional level of nesting has been added to the folders to separate the interfaces and their implementation (for instance ArrayList is an implementation of the List interface). Depending on the interface, we decided to implement different versions, sometimes based on arrays, sometimes based on linked lists because we wanted to give a broad set of examples for the students.

Only two sorting algorithms have been implemented and provide the same interface, only accepting integer arrays. The common folder has been updated since the implementation of the common behaviour *Hashable* (Providing a common hashing method for all

objects) isn't possible with Dafny (because of the lack of generics support for the Traits), it has been replaced by an utility class providing helper methods for the Maps (maps are built-in specification elements in Dafny). Finally, it is important to note that even if the HashMap and TreeMap support generic values, they doesn't support generic keys, which is why their interface has been changed to replace the key with an integer type.

## 3.9 Implementation process

In order to complete this section on methodology, this last subsection will describe the implementation and specification process for every data structure and algorithm in the library, some generic concepts of Dafny will also be detailed in each part. The process was roughly the same for all the implementations, starting with a short design study in order to validate the initial design, then write the specification in Dafny without the implementation, and finally write the implementation. We never had a program completely proven at the first time after using this process a single time, so once this was done, we were using assertions and assumptions to improve the specification to help dafny prove it or sometimes to fix our implementation if it was incorrect.

### 3.9.1 Stacks and state validation

The stacks were the first data structures to be implemented because they seemed to be the most straightforward to implement. We decided to start with the most simple version possible, a version only validated at the operations level (without validating the state of the stack). This version is available in the file SimpleStack.dfy, and is based on a linked list. We first defined the specification without the implementation (visible in listing 4, we decided to display this one as example, but most of the other data structures are too large to be inserted in the report). The advantage of Dafny here was that it allowed us to fist deal with the specification and validate it without having to deal with the implementation, of course the implementation is required in order to fully validate the data structure but it is better and easier to separate the specification from the implementation when dealing with complex programs.

```
class SimpleStack<T> {
  var top: Node<T>;
  var stackSize: int;

  constructor()
    modifies this;
    ensures this.stackSize == 0 && this.top == null;

  method size() returns (s: int)
    ensures this.stackSize == s;

  method isEmpty() returns (empty: bool)
    ensures empty <==> this.top == null;

  method peek() returns (value: T)
    requires this.top != null && this.stackSize != 0;
    ensures value == this.top.value;

  method push(value: T)
    modifies this;
    ensures fresh(top) && top.value == value && top.next == old(top)
    ;
    ensures this.stackSize == old(this.stackSize) + 1;

  method pop() returns (value: T)
    requires this.top != null && this.stackSize != 0;
    modifies this;
    ensures top == old(top).next && value == old(top).value;
    ensures this.stackSize == old(this.stackSize) - 1;
}
```

Listing 4: Simple stack Dafny specification.

The invariant in a stack is that the last element inserted is the first to come out, followed by the element inserted just before and so on, in that case we can see that the specification fully respects this rule through the post conditions of the operations. Once the specification was validated, we implemented the missing parts, added the method's bodies and filled them according to our specification, after a few corrections and adjustments the first stack was valid.

The problem with this stack is that if a new operation is added to the data structure then there is no way to tell that the inner state of the stack object will not be altered and that we won't lose or gain unexpected nodes in the linked list. In order to solve this problem, two other versions of the stack were built, using an array and a linked list, in order to show different ways of validating using Dafny. A new element was introduced here, the use of the Valid predicate, this predicate is supposed to be true before and after all the public operations of a data structures, this ensures that all the provided operations are guaranteed to keep the state in a valid state. This concept regarding the state validation in Dafny has been discussed in [10] by Leino with the concept of Dynamic Frame, which will be explained later with the Queue.

The listing 5 gives an example of the Valid predicate used for the linked list version of the fully validated stack, we can see that the expression handles many different cases, whether the stack is empty or not, it checks that the inner state is always valid. In order to be useful, this predicate must be required (pre-condition) and ensured (post-condition) for every single public operation of the data structure (not necessarily for private ones since they can be used sometimes for recursive purposes, and in that case it is difficult to maintain the state after every recursive call). It is generally a good practice to use a Valid predicate for every class involved in the program since they all have their own state, for instance, the Node class used in the linked list stack also have its own Valid predicate.

```
predicate Valid()
  reads this, this.Repr;
{
  (this.top == null ==>
    (this.stackSize == 0)
    &&
    (isEmptyPredicate()))
  )
  &&
  (this.top != null ==>
    (this.top in this.Repr)
    &&
    (this.top.Repr == this.Repr)
    &&
    (this.top.CurrentNodes == this.CurrentNodes)
    &&
    (this.stackSize == getStackSizeFunction())
    &&
    (this.top.Valid())
  )
}
```

Listing 5: Valid predicate for the linked list stack.

The three different stack versions were not complicated to implement, the linked list based one is very simple since it only removes and changes the top node at every method, the state verification predicate checks that all the nodes that are supposed to be in the stack are still really in the stack by recursively checking that each node holds its own value followed by the value of all its successors (every node uses a specification sequence holding all the successors nodes inside it).

The array based one is completely different and only involves a single class, the stack itself, and works thanks to the combination of an array and a pointer, pointing to the top element in the stack. This version also provides an isFull method, since the array does not

expand automatically. The array simply holds the values that have been pushed onto the stack (the last inserted value being at the index corresponding to the pointer). All the elements before the pointer are the elements considered as in the stack, all the elements after the pointer are considered as undefined, which means they could be previously popped values, or empty values since it is not possible to set a generic type to null or to empty an array element using Dafny, so we just leave the popped elements in the array waiting to be replaced by a new pushed value.

### 3.9.2 Queue and dynamic frames

When the Stack implementations were completed, we decided to move onto the Queue since it shares a data structure with one of the stack: the linked list. The management of a queue is different from a stack since this time the first element inserted has to come out first, and the last one in last, which means that two pointers to the linked list have to be maintained at all times. We decided to call them the tail, where the elements were inserted (enqueued) and the head, where the elements are taken (dequeued). The figure 5 gives a detailed overview of the specification and implementation of the Queue in the library. We can see on the left part that the queue class is separated between the implementation variables (the head and tail pointers and the size) and the specification variables (a set of nodes in the queue, and an ordered sequence of values representing the queue content). On the right part, we can see the linked list, currently holding 4 items. A node of the linked list (represented in blue) contains a value variable and a pointer to the next node as implementation variables, and a sequence of the all next nodes in the list for the specification.

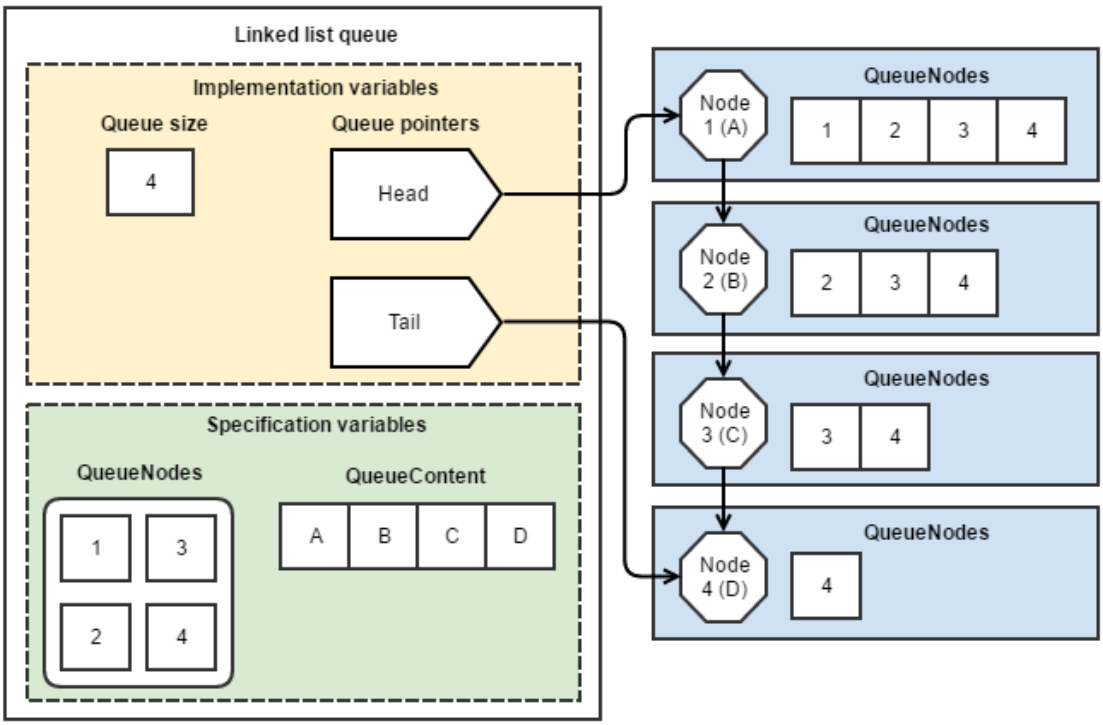



Figure 5: Schema of the queue specification and implementation variables.

All those variables are required to prove that at any times the queue state is valid, we are using a hierarchical system in order to check that each node is valid towards the next node, this way we are sure that there is no unexpected values inserted in the queue. Dafny requires a reference to all the objects that are going to be modified by the program or accessed by the verifier, this is specified through the *modifies* clause and is called a frame condition. In such data structures, the *modifies* clause would be really complex to write if we had to make a reference to every node in the linked list. The dynamic frame concept is very useful in this matter, it helps with programs presenting *abstract aliasing* [19] (when a class have a reference to some secret nodes, like in a linked list for instance). In Dafny, dynamic frames are represented as a set of objects (usually called *Repr* or *Rep*), the type of the objects doesn't matter, the goal is to have all the required object referenced in a single set, this way the modifies clause is way easier to specify since we only have to write that the method will modify the dynamic frame.

We use the concept of dynamic frame with the Queue specification (it was also used in the linked list stack) because many nodes are not accessible directly (as visible for

the nodes 2 and 3 on the figure 5). Every node holds a specification sequence variable composed of its value followed by the next node's values, the queue itself holds a set of all the nodes in the linked list and a sequence of the values which should be identical to the top node's sequence. The verification is made by verifying that each node is actually composed of its value and all its successor's values, this way we can be sure that the queue's state is correct. The dynamic frame is used here but not represented on the schema by lack of space, it's a simple set of object composed of all the linked list's nodes and the queue object (in order to simplify the modify statement by not having to specify the current object and the dynamic frame every time).

The Queue was difficult to implement as the second data structure of the library, the first version wasn't completely specified by dafny because of a problem with specification of the *enqueue* operation, the specification wasn't strict enough for Dafny, we decided to go on with the other parts of the library and to come back after a few other implementations, we managed to fully validate the queue after working on the TreeMap. We spent some time to fully validate the TreeMap and the experience gathered from this data structure about linked lists and dynamic frames was really helpful and allowed us to complete the queue at this time.

### 3.9.3 Sorting algorithms and loops validation

In order to show some examples of loop based specification, we decided to add two sorting algorithms to the library, the Bubble Sort and the Insertion Sort were chosen for the simplicity of their invariants, they make an interesting example without having a too large solution. The bubble sort invariant is that all the elements after the last inserted element are sorted (we always put the largest elements at the end, and we never move them again). For the insertion sort, all the elements before the currently sorted elements are sorted between each other (there may be elements inserted later in that part).

The design of those algorithm was actually straightforward, the interface only provides a *sort* method which realises an in-place sorting of the array. Some helper functions are used, for instance a predicate telling if the array is sorted from a point to another, used in the loop invariants to shorten the size of the specification and validate the algorithm.

```
1  while (outerCounter != arr.Length)
2    invariant 0 <= outerCounter <= arr.Length;
3    invariant sorted(arr, 0, outerCounter);
4    invariant multiset(arr[..]) == multiset(old(arr[..]));
5  {
6    innerCounter := outerCounter;
7    while(innerCounter > 0 && less(arr[innerCounter], arr[innerCounter
       - 1]))
8      invariant 0 <= innerCounter <= outerCounter;
9      invariant forall a, b :: 0 <= a < b <= outerCounter && b !=
     innerCounter ==> lessOrEqual(arr[a], arr[b]);
10     invariant multiset(arr[..]) == multiset(old(arr[..]));
11   {
12     arr[innerCounter], arr[innerCounter -1] := arr[innerCounter -
     1], arr[innerCounter];
13     innerCounter := innerCounter - 1;
14   }
15   outerCounter := outerCounter + 1;
16 }
```

Listing 6: Sorting loop for the insertion sort algorithm.

The listing 6 show the two embedded loops used by the insertion sort algorithm. We can clearly see the loop invariants, starting with the keyword *invariant* and followed by the expression which must be satisfied before and after each loop iteration. We have three kind of invariants on each loop here, the first one setting the bounds of the loop, in order to ensure termination (In this case, Dafny is able to find the termination condition, so there is no need for a *decreases* clause), the second one is the data structure invariant, the first part of the array, before the currently sorted element, must be sorted at all times, and the last one simply ensures that the arrays is simply altered by a permutation, so we don't change the values (A multiset is a Dafny built-in data structure, it is a set which allows duplicates, the set keeps tracks of the number of occurrences for each item, this way we

can ensure that the array doesn't change except for the permutation).

The bubble sort specification is similar, except that it follows the invariants of the specific algorithms as mentioned earlier. Both of the algorithms didn't caused large problems during the specification and the implementation, since the invariants are well defined for those kind of algorithms, it is rather easy to implement them using Dafny. However those versions are not generic, because of the poor generic support of Dafny, but Koenig and Leino proposed in [21] a pseudo-generic solution using the refinement features of Dafny (pseudo-generic because a version including a very small modification is required, a fully generic version wouldn't require a separate version). Their implementation features multiples level of refinement, based on an abstraction of the total order and of the sorting algorithm, the refinement can then use a specific version of the total order for a given type, which will require a more precise version (refinement) of the algorithm using the typed total order. We decided not to use this solution in the project in order to keep the examples within the classical object oriented systems to avoid a too large scope which could lead to difficulties for the students to apprehend the whole library.

### 3.9.4 TreeMap

The TreeMap is one of the largest data structures implemented in the library, it is built on a binary tree and offers a symbol table-like interface. A binary tree has a very clear invariant: for every node *n*, all the keys of the nodes in the left subtree of *n* are smaller than the key of *n* and all the keys of the nodes in the right subtree of *n* are larger than the key of *n*. This single invariant defines a binary tree completely. The listing 7 shows a very small part of the *Valid* predicate for a node in the binary tree, there are some missing parts but we can see how the validation works, we first ensure that the whole tree is valid recursively by checking the subtrees of each node, and then we verify that all the nodes in the subtree are either larger or smaller depending on which subtree is checked. The dynamic frames conditions are not shown here but are required in order for the expression to be validated by dafny.

```
(this.left != null ==>
  (this.left.Valid())
  &&
  (forall y :: y in this.left.KeyValueMap ==> y < this.key)
)
&&
(this.right != null ==>
  (this.right.Valid())
  &&
  (forall y :: y in this.right.KeyValueMap ==> this.key < y)
)
```

Listing 7: Main invariant of the TreeMap (Extract only).

We were unable to validate a version including the *size* operation however, we tried to make a solid relation between the dafny map used to hold all the mappings in the tree and an integer variable supposed to hold the size, but the automated verifier wasn't able to validate it. We met the same problem with the list, but it was solved by using a sequence along with the map to hold the nodes, this is however a problem with the binary tree since there is no simple sequential way to represent it. We believe that the problem is related to the maps, there is probably a missing condition that could help make the automated verifier understand when the map size diminishes or increases because some side tries with simple maps operations were unable to be validated if we added some size conditions (That is the reason why there are no size conditions in the *MapHelpers* utilities class).

Aside of that problem, the specification didn't caused too many problems, the implementation was not as trivial as for the other data structures, the *remove* operation was particularly complex since it involves a rotation of the tree when removing a node with two non-null subtrees. In order to limit the difficulties on this point, we included many comments in the code to allow the student to focus on the formal methods. The TreeMap took nearly a week to complete, but the work made on this data structure was very helpful with the Linked lists used in one of the lists and the queue, it is only after this that we were able to completely validate the queue.

### 3.9.5 Lists

The Lists weren't initially planned to be integrated into the library, but were added as mitigation measure for the problems with the HashMap and the TreeMap. The ArrayList, the first one to be implemented, is a simple data structure, its interface provides an ordered list of elements which are stored in an array. The advantage of such a data structure is to hide the complexity of removing an item from the list without breaking the continuity of the indexes and the resizing of the array when the maximal capacity is reached. This version was very simple to specify and to implement since it doesn't involve any complex invariant or alteration, the most complex operation being the removal of an element which causes a left shift of the elements located at the right of its index. The specification uses a simple sequence to compare the values actually in the array and the values supposed to be in the array.

The second List implementation was added to the project to serve as extended chaining system for the HashMap, which is why it provides a "key-value" feature. Internally, the data structure is simply an linked list, but instead of only having a value, each node also have a key. It allows to provide a *contains*, a *get*, a *put* and a *remove* method that are key-based, which is required for an extended chaining. The figure 6 shows the internal variables used for the specification and implementation. We can see that this linked list is similar to the one used for the queue in section 3.9.2, except that it includes a mapping system, named *KeyValueMap*. The KeyValueMap is a dafny map, used to keep track of all the mappings present in the list. The *ListContent* and *QueueNodes* variables are only here to keep an ordered sequence of the nodes, without any key-value notion, they help during the validation. For the implementation variables, we simply have a top pointer for the linked list and the size on the linked list itself, and a key, a value, and a next pointer in each node.

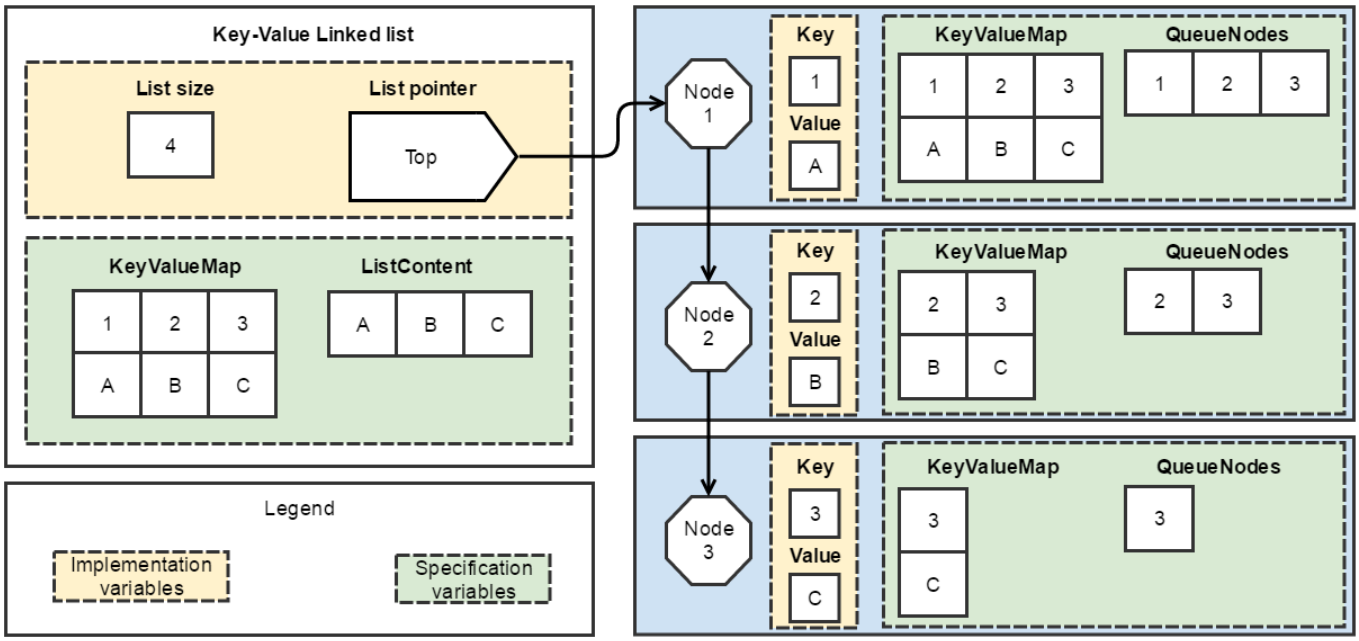



Figure 6: Schema of the key value linked list specification and implementation variables.

The state specification recursively checks that all the nodes are valid, and that the nodes are valid between each other, which ensures continuity and prevents loops in the linked list. We also verify that the nodes that are present in the queue strictly correspond to the key-value map of the specification. This data structure is very specific and has been made in order to be useful for a HashMap, it was complex to validate it completely, but the work realised on the TreeMap helped lot, this implementation also helped us to fully validate the Queue in the library.

### 3.9.6 HashMap and Assumptions

This last data structure is basically a HashTable and features, as for the TreeMap, a symbol table-like interface. This is the only data structure that we didn't managed to validate completely, the *put* and *remove* operations are validated thanks to assumptions. Assumptions are usually used in dafny to find the missing parts of a specification, we use them to assume that a given expression is true, and see if Dafny validates the system or not, when it validates, we need to replace the assumptions by pre/post conditions or assertions, a valid system in Dafny should not use assumptions in its final version.

The HashMap uses the key-value linked list detailed in the previous part as extended chaining system, which reduces a lot the size of the code and allowed us to focus on the invariants of the hash table more than on the linked list. A hash table has an important invariant: A key k can only appear in the array index i if and only if the result of the hash function for k is i. The hashing method used in our implementation is simply the modulo of the integer key by the capacity of the hash map.

The figure 7 pictures the state of the hashmap, we can see some similarities with the schema of the linked list shown in figure 6, that is the case the case because the *KeyValueLinkedList* is exactly the data structure from that figure. On the HashMap side, we are using a simple KeyValue Dafny map in order to keep a list of what is inside the map, and on the implementation side, an array is used as hash table and an integer holds the maximal capacity of the table. On the extended chaining part, we use the variables existing in the linked list.

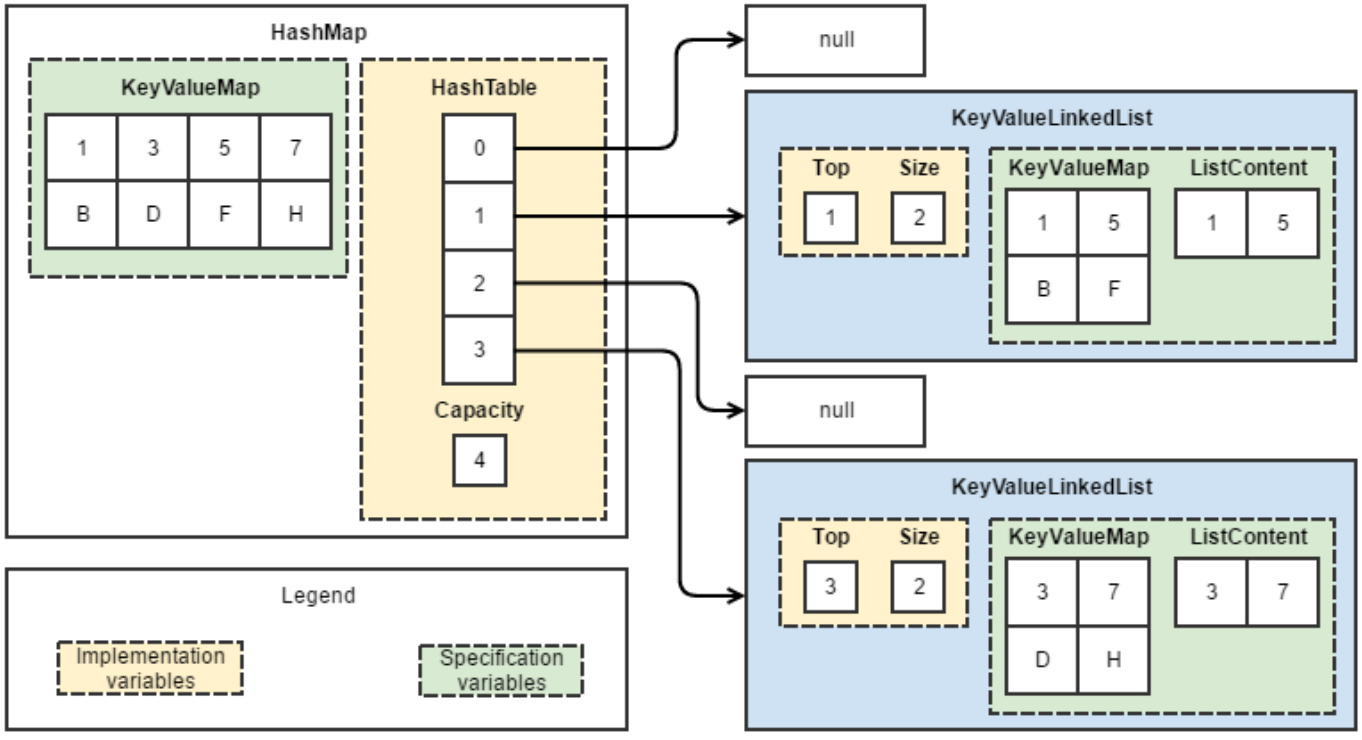



Figure 7: Schema of the HashMap specification and implementation variables.

The specification logic is less straightforward than for other structure here since an em-

bedded data structure is involved, we want to ensure that the invariant mentioned earlier is valid, but this require strict frame conditions, to ensure that the nodes are in a single instance of the linked list for instance, or that a key is only present in a single array index. Since we choose to initialise the array to null for every index, it is important to ensure that for every key in the Dafny map, the index where this key is supposed to be is not null and contains the given key.

The difficulties we had on this one were about the alterations made to the array (removing or inserting a key), from our understanding of the problems, Dafny think that we didn't only modified a single index in the array, the verifier seems to tell us that the array could be null or that other indices of the array could be invalid after that kind of operations, which should not be the case. We decided to use assertions and assumptions in the source in order to show where the problem is and to give an explanation of why we think it should be correct, the listing 8 pictures it.

```
var hashedKey := private_hashKey(keyToRemove);
// (A1) - All the linked lists in the hashtable are valid
assert (forall i :: 0 <= i < this.capacity && i != hashedKey && this
    .hashTable[i] != null ==> this.hashTable[i].Valid());
// (A2) - The program modifies a single entry of the array
this.hashTable[hashedKey].remove(keyToRemove);
// (A3) - The linked list modified in the entry corresponding to the
     hashed key is valid
assert (this.hashTable[hashedKey].Valid());
// Since (A3) is valid after the modification in (A2), knowing that
    (A1) was valid before the modification, then this assumption
    should not be required.
assume forall i :: 0 <= i < this.capacity && i != hashedKey && this.
    hashTable[i] != null ==> this.hashTable[i].Valid();
```

Listing 8: Validation problem with the HashMap.

The comments shows that if all the array elements were valid before the alteration, and if the altered element is valid after the alteration, then all the array should be valid after the

alteration (which is not the case). There is probably a side effect with the called method that we weren't able to correct.

Without the assertions, Dafny tells us that he is unable to prove that the part of the *Valid* predicate which ensures that all the hash table entries are valid will be respected after the removal of the element in the hashtable, the listing 9 shows us that Dafny can prove that the entry which has been modified is valid, but he can't prove that all the other entries are valid after the removal, the error returned by the Dafny verifier is "Error: assertion violation". The assertion have been inserted in our case to make it easier to show the problem in the report, but in without them, the verifier points out that the same expression is wrong in the *Valid* predicate and return the error "Error: A postcondition might not hold on this return path". The problem is the same with the *put* method.

```
1 //This assertion is valid:
2 assert (forall i :: 0 <= i < this.capacity && i != hashedKey && this
      .hashTable[i] != null ==> this.hashTable[i].Valid());
3 //Remove operation on a single array entry:
4 this.hashTable[hashedKey].remove(keyToRemove);
5 //This assertion is valid:
6 assert (this.hashTable[hashedKey].Valid());
7 //This assertion is now invalid:
8 assert (forall i :: 0 <= i < this.capacity && i != hashedKey && this
      .hashTable[i] != null ==> this.hashTable[i].Valid());
```

Listing 9: Verifier errors for the HashMap.

We believe that Dafny fails to understand that we only modify a single entry in the array, but we were not able to find the reason behind this issue, it may be related to the fact that we are using another data structure in each array entry since this is the only part of the library where we are doing this but again we do not see the reason why.

The HashMap implementation was complex and involved for the first time the use of another data structure in a new data structure, but even if we didn't succeeded to validate it, we believe that Dafny is perfectly capable to handle such cases with a correct specification.

# 4 Results

In this section we will present the results gathered from the project, starting with a description of the current project's completion in order to give an overall vision of the project state. A summary of the testing report will be detailed before discussing the quality of implementation and its position towards related work.

## 4.1 Overall completion of the library

The table 1 gives a detailed overview of the library state at the end of the project. We can see that all the data structures but one have been validated and tested.

| Data structure/Algorithm | Validated | Tested | Comments |
|---|---|---|---|
| SimpleStack | Yes | Yes | Generic |
| ArrayStack | Yes | Yes | Generic |
| LinkedListStack | Yes | Yes | Generic |
| LinkedListQueue | Yes | Yes | Generic |
| ArrayList | Yes | Yes | Generic |
| KeyValueLinkedList | Yes | Yes | Generic for values only, not keys |
| TreeMap | Yes | Yes | Generic for values only, not keys |
| HashMap | **Partly*** | **Yes*** | Generic for values only, not keys; * Validated and tested using assumptions |
| BubbleSort | Yes | Yes | Not generic |
| InsertionSort | Yes | Yes | Not generic |

Table 1: Overall completion of the library

The HashMap being fully validated and tested only when assisted with assumptions as explained in 3.9.6. Most of the implementation support generic values, except for the sorting algorithms which are limited because of the generic issue mentioned in section 3.4. The key-value-based data structures (TreeMap, Hashmap and KeyValueLinkedList)

are affected by the same issue for their keys since there is no possibility using the Dafny classes to compare generic keys at the moment.

Thanks to this table, we can say that a large part of the project has been completed (even if the two Lists were not planned at the beginning of the project). The genericity of some implementations is a problem, but we decided to accept the issue and not change the language because of this since we think that this version is useful enough for educational purposes.

## 4.2 Testing

A summary of the testing we made for every part of the library will be described in this part, we won't include the test cases for every implementation since they are not particularly interesting and just make calls to the available methods. A quick analysis of the performance will be proposed and compared to an existing reference if available.

### 4.2.1 Stacks

We decided to group the three stacks in the same section for the testing since they all share the same concepts, except that their specification is different, but this has been detailed in the methodology section.

Dafny fully validates the three implementations, even if the SimpleStack doesn't provide a complete state validation process. We decided to keep it in the library to serve as example in order to explain the concept of state validation.

The test case is similar for the three implementations, we use all the operations of the stack in different order in order to be sure that the state is valid before and after every operation, and we use assertions to verify that the output parameters are valid towards our use case. A sample test case for the linked list case can be seen in the listing 10.

Regarding the implementations, a quick asymptotic analysis has been made on the three stacks and we detected that no loops nor recursive methods were used for the implementations of every single methods in those data structures, we can therefore confirm that the algorithms will operate in a constant time regardless of the number of items present

in the stack which is compliant with the analysis proposed by T. Cormen et al. in [6].

The testing of the stacks did not reveal any problem, we can continue with the remaining parts of the library.

```
method Main() {
  var stack1 := new LinkedListStack<int>();
  var isEmpty: bool, poppedValue: int, peekedValue: int, stackSize:
   int;
  isEmpty := stack1.isEmpty();
  stackSize := stack1.size();
  assert isEmpty == true;
  assert stackSize == 0;
  stack1.push(10);
  stack1.push(11);
  stackSize := stack1.size();
  assert stackSize == 2;
  peekedValue := stack1.peek();
  assert peekedValue == 11;
  poppedValue := stack1.pop();
  assert poppedValue == 11;
  stackSize := stack1.size();
  assert stackSize == 1;
  //Test case voluntarily reduced for the report
}
```

Listing 10: Test case for the LinkedListStack.

### 4.2.2 Queue

The Queue has been fully validated by Dafny, and the test case has been successfully accepted. We only have a linked-list based implementation because we decided that an array-based implementation would not be a very valuable added element to the library. It is interesting to note that in the case of the queue, loops and recursive methods have

been used by the verification code, but they should not impact the performance of the generated executable since they only affect ghost variables.

Even if this report concerns the final testing only, the intermediate testing of this data structure helped us to understand better the concept of dynamic frames with the linked lists and helped to fix issues on the TreeMap and the key-value-based list.

Regarding the performance of the implementation, as mentioned earlier no loops or recursive methods are used in the implementation, we can therefore expect a constant time for every operation. Since a constant time is to be expected as explained in [6], we can conclude that the Queue has been successfully tested.

### 4.2.3 ArrayList

The array-based List implementation has been fully validated and tested.

Unfortunately we weren't able to find academic literature which could serve as reference for our implementation of a linked list since we don't provide some very specific methods (like *contains* for instance). Since the data structure works as an improved array, we can deduce that the *get* operation will have a constant time, which is the case in our implementation. The *add* method should also have a constant time, however, when the array is full, a larger array is created and all the values are migrated to the new array, therefore most of the time our implementation will take a constant time, but sometimes, it will take a time linear to the number of items in the array to increase it (O(n), n being the number of elements). Finally, the remove operation will actually remove in a constant time since it knows which index to remove, but it needs to shift all the next elements to the left, therefore, the remove operation will also take a time linear to the number of items in the array. The operations *size* and *isEmpty* and both the constructors will take a constant execution time.

### 4.2.4 Key-value-based list

This data structure was designed to be used as a basic linked list with key-value properties in order to be usable as an extended chaining system for the HashMap. Dafny validated

our implementation, and our test case was validated too. We decided to test the data structure as we did for the TreeMap and the HashMap in order to verify the key value system. The testing was also successful.

Regarding the performance of the implementation, our constructor, *size* and *isEmpty* methods have a constant running time but all the other methods (*get*, *put*, *remove* and *contains*) involve a search system since our interface is similar to a list, every operation will therefore run in a time linear to the number of elements in the linked list in order to find the corresponding element through a recursive search in the list. In [6], we can see that the linked list search operation is actually supposed to take a linear time, which makes our implementation compliant.

### 4.2.5 TreeMap

Like all the previous data structures, our TreeMap implementation has been successfully validated by Dafny and fully tested. The test case uses the map features built on top of the binary search tree.

Regarding the performance, the constructor and the *isEmpty* operations run in a constant time, all the other methods, however, are using a tree search operation, which, in our implementation, runs in a time linear to the height of the tree. The remove operation will sometimes run two search operations since it will require rotating the tree when a node with two children is removed.

Our implementation doesn't provide a system used to balance the tree at all times, which means that in the worst case, height of the tree can also be the number of elements in the tree, in that case, the running time will be linear to the number of elements (*O(n)*). Sedgewick et al. presented the same worst case values in [22].

### 4.2.6 HashMap

The HashMap has not been fully validated by Dafny even after many different approaches with the specification. We managed to get a version validated using assumptions to force Dafny to validate the part that we didn't manage to prove manually. However, this version,

with the assumptions, has passed the test case completely, we believe that the problem resides in our specification, we probably misunderstood a part of the Dafny language that could help us with the arrays. This data structure is therefore not considered as fully validated for the project and would require some improvements.

On the performance side, since we are not making a complete mathematical analysis it is difficult to give a precise time complexity here, we can assert that in the best case, the insert, search and remove operations will take a constant time (*O(1)*) since we are basically making an array access to an empty list. But in the worst case, if the hashing function is not efficient, then all the values could be located in the same extended chain, which would give an asymptotic time complexity linear *O(n)* to the number of elements in the HashMap because a search operation would occur in the linked list. Moreover, the hashing function used in our example is very basic (a simple modulo) and doesn't reflect a real life use of such a data structure. Our results for the best-case and worst-case analysis are the same than the ones proposed by Cormen et al. in [6].

### 4.2.7 Sorting algorithms

The bubble sort and insertion sort implementations are both validated by Dafny and tested through a very similar test case. Both of the algorithms only provide an in place *sort* method, which simply permutes the array. The test case used for the insertion sort is visible in the listing 11, as we can see, we simply make sure that a non-sorted array is well sorted after the method call.

Regarding the performance of the algorithms, our bubble sort implementation only uses two embedded loops, which means that the average time complexity will be $O(n^2)$, n being the number of elements in the array. The insertion sort uses the same logic, except that it handles the processing of the array in a different way, but the average time complexity will be the same than for the bubble sort implementation. We didn't manage to find a solid reference for those algorithms, we decided to use different Internet sources (http://bigocheatsheet.com/, http://www.cs.nott.ac.uk/~psznza/G52ADS/simplesorts1.pdf) to compare with existing results, and we found that our implementations complexities were coherent with the ones we found online.

```
method Main() {
  var arrInt := new int[4];
  arrInt[0] := 2;
  arrInt[1] := 3;
  arrInt[2] := 4;
  arrInt[3] := 1;
  assert arrInt[..] == [2,3,4,1];
  InsertionSortInt.sort(arrInt);
  assert arrInt[..] == [1,2,3,4];
}
```

Listing 11: Test case for the Insertion Sort.

## 4.3 Related work

During our literature review, we didn't find any study directly related to the formal specification of a software intended for education purposes, however, as mentioned in section 2.3 we did find some Dafny specifications in the academic literature. We found that most of the programs described in those papers were either lacking a useful aspect (a level of abstraction too high for being useful in the industry for instance the sorting algorithm in [21]), which makes it difficult for the student to work on its own projects, or only using Dafny's built-in data structures (sets, sequences and maps like the queue in [16]) which is not really useful for the student to understand how to relate to other programming paradigms. Some programs are also too complex to be used as an educational material like the Schorr-Waite algorithm in [10] which features many lines of specification.

We tried to do our best to avoid falling in those same problems by providing generic, non-abstract versions of the data structures as often as possible and using standard structures for the implementations (linked lists instead of Dafny's sequence for instance). Unfortunately some programs are complex and have a medium size, for instance the TreeMap, which is using a complex implementation. The choice of limiting the abstraction of our library implied a trade-off between the specification features used and the feasibility of the project, as mentioned in section 3.4, those risks were expected and had to be mitigated.

# 5 Discussion

In this report, we detailed section by section our research, methodology and results; we explained our choices and the reasons behind them. In this section we will critically review our work on the project by looking at the main goals and how did we fulfil them, assessing the strengths and weaknesses, examining the key issues that occurred during the project and discussing our contribution to the field of Formal Methods.

## 5.1 Critical analysis of the project

The main goal of our project, A formally proven data structure library intended for Formal Methods teaching, was to provide a useful resource for students working on Formal Methods, we believe that we mostly succeeded and created a (nearly complete) data structure library which is easy to understand and features a large set of various validation techniques, including pre/post-conditions, ghost variables, loop invariants, termination conditions and frame conditions. We applied ourselves to maintain a clean and clear coding style to help the reader to understand the specification line by line, along with the comments we provided.

Thanks to our effective literature review, we had at disposal a large set of academic literature which was proven very useful throughout the project, first to define the scope of the library, then later to understand how Dafny works and what are the best ways to specify a data structure with it. We believe that the literature review covers a large and nearly sufficient part of the project scope, except for the SMT solvers and particularly *Z3*, which is used by Dafny (see section 2.2.3), we think that it could have been helpful to include them in the literature review in order to write more efficient specifications (maybe to even fully validate the HashMap). Unfortunately we only looked at this domain at the very end of the project, and we decided not to include more than what was already present in the literature review since we would not have had the time to change the library according to those additional papers and we preferred take the time to write a decent and complete report.

Even if most of the project has been successfully completed and validated, we must

consider the main issue of the project, the specification of the HashTable that has not been fully completed. The main reason behind this issue was a lack of time at the end of the time frame allocated to the implementation of the library, as explained in section 3.6.1, we had to switch to the final writings by the end of August in order to be able to provide a complete and reviewed report by the 30th of September. The mitigation solution applied for this issue was to use Dafny assumptions in order to have a partly proven data structure since we were limited by the time. After consideration, we think that not writing enough of the final report during the research and implementation phases was a mistake which limited the time that we could dedicate to the implementation in case of issues at the end of the project. Since we did not expect so many issues with the HashTable, we were unable to allocate more time to solve it. This was probably our biggest weakness during the project since it also slowed down the final writing because some parts previously realized took more time to write than if it would have been written just after the work was done.

Minor issues were also presented throughout the sections 3.8 and 3.9, mostly regarding initially planned features that we had to remove because of the Dafny language capabilities. As mentioned earlier, we think that those issues are not important enough to switch to another language since the main goal of the library is educational. We did apply a mitigation solution for those issues by adding to the library the List data structure with two implementations. We believe that even with the missing features, the combination of the Dafny language and the current state of the library is a useful teaching material as explained throughout the whole report: the language is an active project which has been designed for validation purposes and the library proposes a large set of verification methods throughout various commented data structures implementations.

On the project management side, our choice of working on a data structure at a time was a good idea since it allowed us to fully concentrate on the specifications then on the implementations before working on another data structure. However we discovered that when we were stuck on a particular point, switching to another part to come back later on the issue was very helpful, for instance we were only able to fully validate the Queue after completing the TreeMap which helped us to have a better understanding of the dynamic frames. The fact of using a Kanban board (usually used in agile methodologies) was very

rewarding throughout the project since having on a single board all the tasks to do and the tasks that have been done on the project is highly motivating and helps to have an overview of the project.

## 5.2 Fulfilling the educational purpose

The library created in this project has to respond to a problem regarding the Formal Methods in the education, as explained in section 2.2.2, there is a need for more user friendly resources and a wider presence of the subject in universities. We believe that our project will contribute to improve those two points, first we provide a large, documented and useful teaching resource which can be used in Formal Methods courses to help students understand the good practices with behavioural interface specification languages and how to use them to specify complex systems. As explained throughout the report, we tried to make the library easy to understand and featuring various validation techniques, which should be helpful for students to get started with their own specifications. Secondly, we believe that this library can be used to motivate other universities to create Formal Methods courses by providing examples usable in the lectures and/or exercises, this would contribute to help spread the use and the popularity of Formal Methods in the Software Engineering world.

## 5.3 Professional issues

Regarding the professional issues related to our project, we explained in section 2.4 that the main professional issues were ethical. The most important one being that the project should not try to influence the student with arbitrary choices, we believe that we avoided this issue since all the main choices of this project have been explained and are based on the previously made literature review. The other minor issue that was expected concerns the correctness of the library, it is important to provide a correct implementation of the data structures if we want the project to be useful as a teaching material. As detailed in section 4, a complete testing phase was realized and we did not notice any problem regarding the correctness of the implementations (The correctness is tested with the test cases).

## 5.4 Contribution to the body of knowledge

We think that our project will contribute to the field of Formal Methods on several levels. First we believe that it will actually help students to have a better understanding and an increased interest towards the field by having access to the concrete examples provided in this library. Increasing the interest in Formal Methods could contribute to an increased usage and therefore an improvement for the formal methods usage in the education or the industry. Also, our literature review summarizes a set of writings about Formal Methods and more particularly Behavioural Interface Specification Languages and their usage in education, which can be helpful for someone looking for an overview of the field. Finally, our project features a nearly fully proven data structure library which is also compilable and reusable in .NET programs, as mentioned earlier, we did not find any paper mentioning discussing a similar project, it can therefore be an example that widely used libraries could use Formal Methods in order to increase their reliability level.

# 6 Conclusion

During this project we have created a formally proven data structure library intended for teaching purposes. A set of algorithms and data structures have been defined during the research phase, which were later specified and implemented using the Dafny language, which has been chosen as specification language for its ease of use, its closeness to Spec#, the currently taught specification language at Oxford Brookes University, and the fact that it has been designed as a verification language. All the modules of the library except the HashMap have been fully proven valid using the Dafny verifier program and their specification has been tested through our test cases.

A few issues came up during the development concerning the specification of the HashMap that we weren't able to complete and the object oriented features of Dafny which forced us to withdraw some generic design aspects from the library. We mitigated those issues by respectively adding the List data structure to the library and altering the design, which reduced the generic support. Ethically speaking, we took care of making justified and detailed choices in order to avoid forcing personal choices on the students using the library.

We managed to successfully specify all but one data structures and algorithms of the library as described in table 1. As explained before, we believe that the library will be useful in an educational context in order to help students to understand the course or lecturers by providing examples for their teaching materials. The issues mentioned earlier are not critical for an educational purpose since the student can still benefit from the formally proven data structures even if they do not support generic types.

However, if this library were to be used in an industrial context, then the lack of generic support would largely limit the usefulness of the library (mostly for the maps) and would require specific implementations for each type, which is not very great. This issue could be removed if Dafny releases support for generic object oriented features in the future and if the library were to be updated. We also want to believe that such projects will bring more interest to the field and contribute to a wider possibility of usages.

Working on this project was a fascinating experience and was highly rewarding on

a personal level, we learned many useful techniques and methods regarding the software validation field and we firmly believe that it will benefit us in our professional lives to produce safer and more reliable software. Working of formal specifications forced us to improve our analytical skills towards a software system and we now feel more confident in finding potential corner cases or exceptions in a system. Finally, we were used to implement the software and then write an unit test to validate our implementation and protect it against regression bugs, however we now understand that this pattern present a potential failure point since the test case may not cover every possible case. This is where the combination of Formal Methods and classical testing methods become useful, the formal verification will detect any case that is not accepted by the formal specification (the formal specification has to be strong enough, but that's another discussion), and the classical tests will ensure that the formal specification matches the expected business requirements.

# 7 Recommendations

Our research and work on the library made us consider potential extended improvements that didn't fit in the scope of this project. Regarding the Dafny Language, as we explained in this report, Dafny's limited object oriented features has been the main technical issue that occurred during this project, even the Traits system, proposed by R. Ahmadi, K. Leino and J. Nummenmaa in [13] doesn't solve our issue. They propose as future work an extension of this system supporting generic types for the Traits, such a system would improve the capabilities of the Dafny language in terms of object oriented design, and would allow the data structure library to support even more generic structures.

Secondly, as explained by V. Almstrum, C. Dean, D. Goelman et al. in [4], the field of formal methods doesn't provide very user friendly tools, we strongly agree on this point, since the tooling required for this project was an old version of Visual Studio and/or a very light (as in not very efficient in usage) web-based IDE. In order to improve the interest for formal methods in education and in the industry, it could be useful to provide a user friendly dedicated IDE for a specification language like Dafny (or even for many languages as long as they share common features, like Behavioural Interface Specification Languages). The IDE could provide automated verification, code completion, meaningful error messages and visualisation tools to have a better understanding of the issues within a specification.

# References


[1] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods. *ACM Computing Surveys*, 41(4):1–36, 2009.

[2] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):1–58, 2012.

[3] Oxford Brookes University - Department of Computing and Communication Technologies Postgraduate Modules. https://students.cct.brookes.ac.uk/PG_Modules. [Online; accessed 21-June-2016].

[4] Vicki L Almstrum, C Neville Dean, Don Goelman, Thomas B Hilburn, and Jan Smith. Support for teaching formal methods. *ACM SIGCSE Bulletin*, 33(2):71, jun 2001.

[5] Shaoying Liu, Kazuhiro Takahashi, Toshinori Hayashi, and Toshihiro Nakayama. Teaching formal methods in the context of software engineering. *ACM SIGCSE Bulletin*, 41(2):17, jun 2009.

[6] Ce Charles E Leiserson, RI Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[7] Bertrand Meyer. Applying ”Design by Contract”. *Computer*, 25(10):40–51, 1992.

[8] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec\# Programming System: An Overview. *International Conference in Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*, (October):49–69, 2004.

[9] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. *ACM SIGSOFT Software Engineering Notes*, 31(3):1, may 2006.

[10] K. Rustan M Leino. Dafny: An automatic program verifier for functional correctness. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6355 LNAI:348–370, 2010.

[11] Luke Herbert, K Rustan M Leino, and Jose Quaresma. Using Dafny, an Automatic Program Verifier. pages 156–181, 2012.

[12] K. Rustan M Leino. Developing verified programs with Dafny. *Proceedings - International Conference on Software Engineering*, pages 1488–1490, 2013.

[13] Reza Ahmadi, K. Rustan M. Leino, and Jyrki Nummenmaa. Automatic verification of Dafny programs with traits. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs - FTfJP '15*, pages 1–5, New York, New York, USA, 2015. ACM Press.

[14] Gt Leavens and Y Cheon. Design by Contract with JML. *Draft, available from jmlspecs. org*, 1:4, 2006.

[15] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# Experience. *Communications of the ACM*, 54:81, 2011.

[16] K. Rustan M. Leino and Rosemary Monahan. Dafny Meets the Verification Benchmarks Challenge. volume 6217, pages 112–126. 2010.

[17] G Tremblay. Formal methods: mathematics, computer science or software engineering? *Education, IEEE Transactions on*, 43(4):377–382, 2000.

[18] Tom Maibaum. Formal methods versus engineering. *ACM SIGCSE Bulletin*, 41(2):6, jun 2009.

[19] Ioannis T Kassios. Dynamic Frames and Automated Verification. pages 1–26, 2011.

[20] K. Rustan M. Leino. Specification and verification of object-oriented software. *Engineering Methods and Tools for Software Safety and Security, lecture notes of the Marktoberdorf International Summer School 2008*, 22:231–266, 2009.

[21] Jason Koenig and K. Rustan M. Leino. Programming Language Features for Refinement. *Electronic Proceedings in Theoretical Computer Science*, 209:87–106, jun 2016.

[22] Robert Sedgewick and Kevin Wayne. Algorithms (4th Edition). In *Algorithms (4th Edition)*, pages 1–992. Addison-Wesley Professional, 4th editio edition, 2011.

[23] Brad Long. Towards the design of a set-based Java collections framework. *ACM SIGSOFT Software Engineering Notes*, 35(5):1, oct 2010.

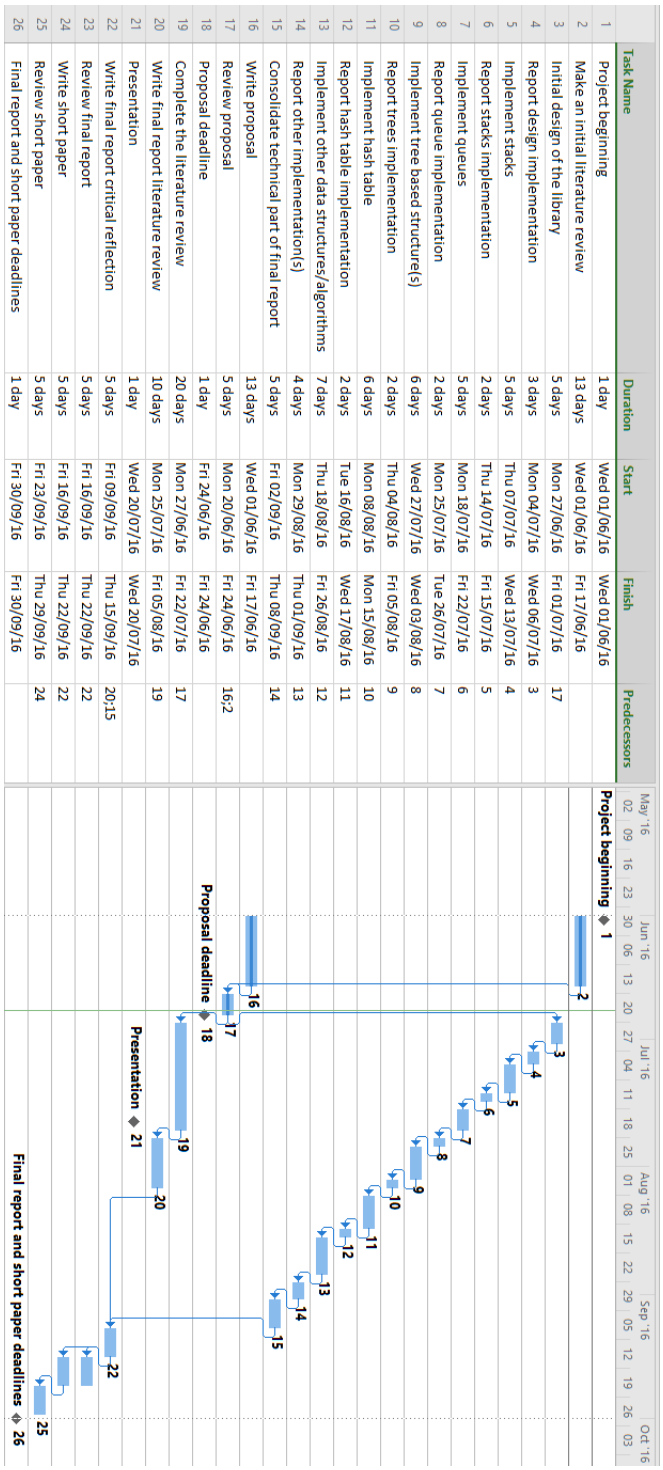
[24] Collections in Java - Oracle Java Documentation. http://docs.oracle.com/javase/tutorial/collections/. [Online; accessed 22-June-2016].

[25] Collections and Data Structures in .NET - Microsoft Developer Network. https://msdn.microsoft.com/en-us/library/7y3x785f(v=vs.110).aspx. [Online; accessed 22-June-2016].

[26] K. Rustan M. Leino and Valentin Wüstholz. The Dafny Integrated Development Environment. *Electronic Proceedings in Theoretical Computer Science*, 149:3–15, 2014.

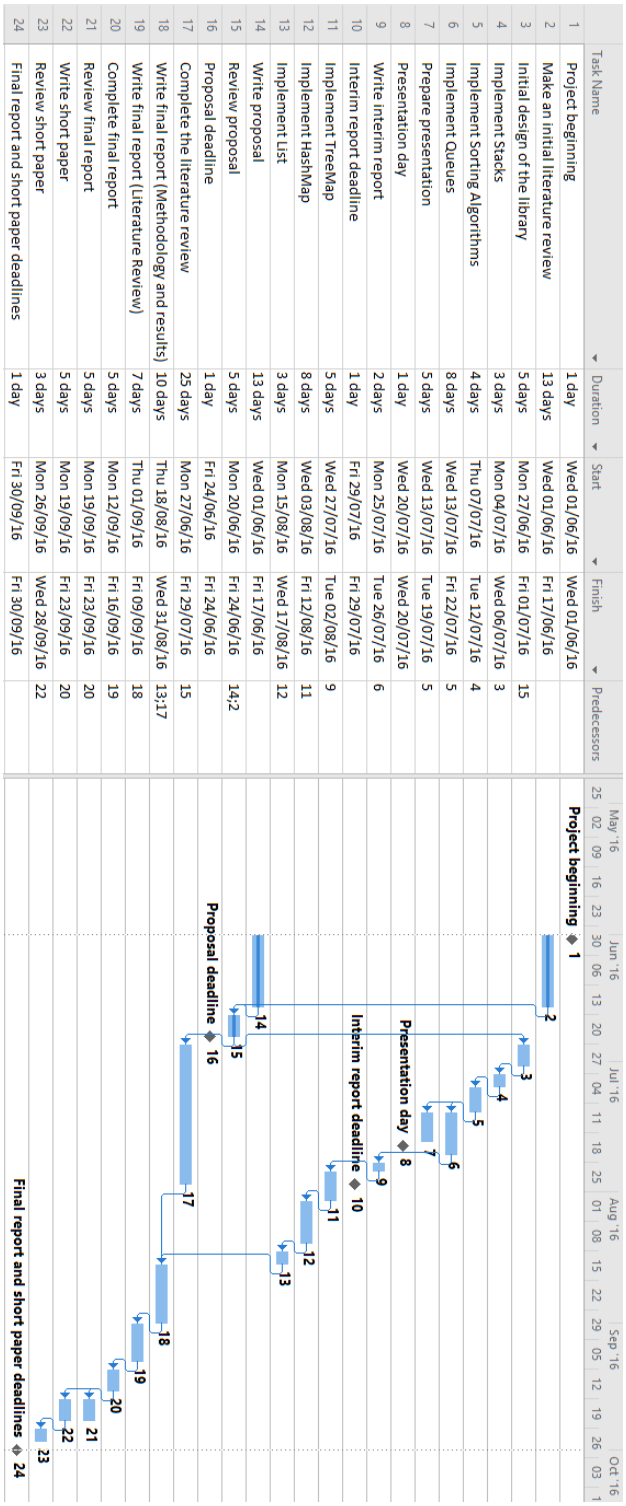
# Appendices

## A Initial Gantt chart



| | Task Name | Duration | Start | Finish | Predecessors |
|---|---|---|---|---|---|
| 1 | Project beginning | 1 day | Wed 01/06/16 | Wed 01/06/16 | |
| 2 | Make an initial literature review | 13 days | Wed 01/06/16 | Fri 17/06/16 | |
| 3 | Initial design of the library | 5 days | Mon 27/06/16 | Fri 01/07/16 | 17 |
| 4 | Report design implementation | 3 days | Mon 04/07/16 | Wed 06/07/16 | 3 |
| 5 | Implement stacks | 5 days | Thu 07/07/16 | Wed 13/07/16 | 4 |
| 6 | Report stacks implementation | 2 days | Thu 14/07/16 | Fri 15/07/16 | 5 |
| 7 | Implement queues | 5 days | Mon 18/07/16 | Fri 22/07/16 | 6 |
| 8 | Report queue implementation | 2 days | Mon 25/07/16 | Tue 26/07/16 | 7 |
| 9 | Implement tree based structure(s) | 6 days | Wed 27/07/16 | Wed 03/08/16 | 8 |
| 10 | Report trees implementation | 2 days | Thu 04/08/16 | Fri 05/08/16 | 9 |
| 11 | Implement hash table | 6 days | Mon 08/08/16 | Mon 15/08/16 | 10 |
| 12 | Report hash table implementation | 2 days | Tue 16/08/16 | Wed 17/08/16 | 11 |
| 13 | Implement other data structures/algorithms | 7 days | Thu 18/08/16 | Fri 26/08/16 | 12 |
| 14 | Report other implementation(s) | 4 days | Mon 29/08/16 | Thu 01/09/16 | 13 |
| 15 | Consolidate technical part of final report | 5 days | Fri 02/09/16 | Thu 08/09/16 | 14 |
| 16 | Write proposal | 13 days | Wed 01/06/16 | Fri 17/06/16 | |
| 17 | Review proposal | 5 days | Mon 20/06/16 | Fri 24/06/16 | 16;2 |
| 18 | Proposal deadline | 1 day | Fri 24/06/16 | Fri 24/06/16 | |
| 19 | Complete the literature review | 20 days | Mon 27/06/16 | Fri 22/07/16 | 17 |
| 20 | Write final report literature review | 10 days | Mon 25/07/16 | Fri 05/08/16 | 19 |
| 21 | Presentation | 1 day | Wed 20/07/16 | Wed 20/07/16 | |
| 22 | Write final report critical reflection | 5 days | Fri 09/09/16 | Thu 15/09/16 | 20;15 |
| 23 | Review final report | 5 days | Fri 16/09/16 | Thu 22/09/16 | 22 |
| 24 | Write short paper | 5 days | Fri 16/09/16 | Thu 22/09/16 | 22 |
| 25 | Review short paper | 5 days | Fri 23/09/16 | Thu 29/09/16 | 24 |
| 26 | Final report and short paper deadlines | 1 day | Fri 30/09/16 | Fri 30/09/16 | |

# B Final Gantt chart



| | Task Name | Duration | Start | Finish | Predecessors |
|---|---|---|---|---|---|
| 1 | Project beginning | 1 day | Wed 01/06/16 | Wed 01/06/16 | |
| 2 | Make an initial literature review | 13 days | Wed 01/06/16 | Fri 17/06/16 | |
| 3 | Initial design of the library | 5 days | Mon 27/06/16 | Fri 01/07/16 | 15 |
| 4 | Implement Stacks | 3 days | Mon 04/07/16 | Wed 06/07/16 | 3 |
| 5 | Implement Sorting Algorithms | 4 days | Thu 07/07/16 | Tue 12/07/16 | 4 |
| 6 | Implement Queues | 8 days | Wed 13/07/16 | Fri 22/07/16 | 5 |
| 7 | Prepare presentation | 5 days | Wed 13/07/16 | Tue 19/07/16 | 5 |
| 8 | Presentation day | 1 day | Wed 20/07/16 | Wed 20/07/16 | |
| 9 | Write interim report | 2 days | Mon 25/07/16 | Tue 26/07/16 | 6 |
| 10 | Interim report deadline | 1 day | Fri 29/07/16 | Fri 29/07/16 | |
| 11 | Implement TreeMap | 5 days | Wed 27/07/16 | Tue 02/08/16 | 9 |
| 12 | Implement HashMap | 8 days | Wed 03/08/16 | Fri 12/08/16 | 11 |
| 13 | Implement List | 3 days | Mon 15/08/16 | Wed 17/08/16 | 12 |
| 14 | Write proposal | 13 days | Wed 01/06/16 | Fri 17/06/16 | |
| 15 | Review proposal | 5 days | Mon 20/06/16 | Fri 24/06/16 | 14;2 |
| 16 | Proposal deadline | 1 day | Fri 24/06/16 | Fri 24/06/16 | |
| 17 | Complete the literature review | 25 days | Mon 27/06/16 | Fri 29/07/16 | 15 |
| 18 | Write final report (Methodology and results) | 10 days | Thu 18/08/16 | Wed 31/08/16 | 13;17 |
| 19 | Write final report (Literature Review) | 7 days | Thu 01/09/16 | Fri 09/09/16 | 18 |
| 20 | Complete final report | 5 days | Mon 12/09/16 | Fri 16/09/16 | 19 |
| 21 | Review final report | 5 days | Mon 19/09/16 | Fri 23/09/16 | 20 |
| 22 | Write short paper | 5 days | Mon 19/09/16 | Fri 23/09/16 | 20 |
| 23 | Review short paper | 3 days | Mon 26/09/16 | Wed 28/09/16 | 22 |
| 24 | Final report and short paper deadlines | 1 day | Fri 30/09/16 | Fri 30/09/16 | |