Informatics in Education, 2024, Vol. 23, No. 1, 57–99
© 2024 *Vilnius University*
DOI: 10.15388/infedu.2024.05

Empirical Evaluation of a Differentiated Assessment of Data Structures: The Role of Prerequisite Skills

Marjahan BEGUM[1], Pontus HAGLUND[2], Ari KORHONEN[3], Violetta LONATI[4,5], Mattia MONGA[4,5], Filip STRÖMBÄCK[2], Artturi TILANTERÄ[3]
[1]*City, University of London, London, UK*
[2]*Linköping University, Linköping, Sweden*
[3]*Aalto University, Helsinki, Finland*
[4]*Università degli Studi di Milano, Milan, Italy*
[5]*Laboratorio Nazionale CINI 'Informatica e Scuola', Rome, Italy*
e-mail: marjahan.begum@city.ac.uk, pontus.haglund@liu.se, archie@cs.hut.fi, lonati@di.unimi.it, mattia.monga@unimi.it, filip.stromback@liu.se, artturi.tilantera@aalto.fi

Received: April 2023

Abstract. There can be many reasons why students fail to answer correctly to summative tests in advanced computer science courses: often the cause is a lack of prerequisites or misconceptions about topics presented in previous courses. One of the ITiCSE 2020 working groups investigated the possibility of designing assessments suitable for differentiating between fragilities in prerequisites (in particular, knowledge and skills related to introductory programming courses) and advanced topics. This paper reports on an empirical evaluation of an instrument focusing on data structures, among those proposed by the ITiCSE working group. The evaluation aimed at understanding what fragile knowledge and skills the instrument is actually able to detect and to what extent it is able to differentiate them. Our results support that the instrument is able to distinguish between some specific fragilities (*e.g.,* value vs. reference semantics), but not all of those claimed in the original report. In addition, our findings highlight the role of relevant skills at a level between prerequisite and advanced skills, such as program comprehension and reasoning about constraints. We also suggest ways to improve the questions in the instrument, both by improving the distractors of the multiple-choice questions, and by slightly changing the content or phrasing of the questions. We argue that these improvements will increase the effectiveness of the instrument in assessing prerequisites as a whole, but also to pinpoint specific fragilities.

Keywords: data structures and algorithms, prerequisite skills, prerequisite knowledge, misconceptions, CS2, computer science education, qualitative content analysis.

1. Introduction

Many students of Computer Science (CS) programs often struggle with courses focused on programming, algorithms, or data structures and sometimes fail to achieve desired learning outcomes (Zingaro *et al.*, 2018). Teachers have a responsibility to assess and investigate why their students possibly fail. However, there are numerous different reasons why students fail to answer an assessment question correctly including a slip or, more frequently, lack of prerequisites, misconceptions (*i.e.,* incorrect or incomplete mental model of the related topics, see Qian and Lehman (2017); Margulieux *et al.* (2021)), or not having internalized the topic to a high enough level. In this paper, we call all of these *fragilities*, or more specifically *fragile knowledge* and *skills*, following the terminology proposed by Perkins and Martin (1985). They claim that students' difficulties in programming "stem from knowledge that is *fragile* in several ways, *i.e.*, partial knowledge, inert knowledge, lack of a critical filter, misplaced knowledge, and conglomerated knowledge" (ibid. p. 1).

Considering all these reasons why a student may answer incorrectly, understanding why a student is struggling is a challenging proposition. Even more so in advanced courses, when the cause for wrong answers could be unrelated to the new topics, and instead concern topics covered in previous programming courses. This is illustrated by Valstar *et al.* (2019) who found a correlation between a student's score on the final exam in an upper-level data structures course, and their ability to answer questions on pointers and tracing recursion. This observation shows that it is also important to consider the impact of knowledge and skills from previous programming courses, which we will refer to as *prerequisites*.

To explore this, a 2020 ITiCSE working group (Nelson *et al.*, 2020) explored what programming-related prerequisites a student needs to be proficient with in order to correctly answer questions in a latter course. The working group focused on questions that involve some amount of code (*i.e.,* not purely theoretical questions), and concluded that many require proficiency with both prerequisites and the topics of the current, advanced, course (*advanced topics*). This means that if a student answers a question incorrectly, it is difficult to locate the cause (or causes) of the incorrect answer. To address this problem, the working group (Nelson *et al.*, 2020) proposed assessments which are able to *differentiate* between weaknesses with programming prerequisites and with advanced topics. They also discussed a collection of design principles and strategies that can be used to construct differentiated assessments.

In this paper, we empirically evaluate one of the differentiated assessment proposed in Nelson *et al.* (2020), which the authors claim is able to differentiate between weaknesses in programming prerequisites and weaknesses related to advanced topics in a data structures and algorithms course. Specifically, the assessment (Nelson *et al.*, 2020, Appendix M.1.4) consists of the listing of a program and a set of related questions, concerning the implementation of a queue as an expandable circular array. More precisely, here we consider a slightly modified version (henceforth referred to as "the Instrument", see Section 3) of that set of questions.

To empirically evaluate the Instrument, we conducted semi-structured interviews and analyzed them with qualitative content analysis – following the methodology described in Schreier (2014). We pose the following research questions:

RQ1 What fragile knowledge and skills is the Instrument able to detect?

RQ2 To what extent is the Instrument able to differentiate between fragilities, both between prerequisites and advanced topics, and among specific prerequisites?

Note that instead of the term *weakness*, used by Nelson *et al.* (2020), we use the term *fragile* in accordance with the terminology of Perkins and Martin (1985).

In order to address these RQs, we administered the Instrument to 18 students at two institutions in two different countries, and interviewed them with the aim of uncovering fragilities related to their incorrect answers. Relevant transcribed excerpts both from the interviews and the written answers were then analyzed. The findings show that the Instrument is able to detect most of the "weaknesses" (fragilities) expected by Nelson *et al.* (2020), though not all of them. Hence we advise some changes to improve the Instrument.

For many questions from the Instrument, the cause of wrong answers can be tracked back to difficulties in high-level skills such as program comprehension and reasoning about constraints. Indeed they are typically not well developed at the end of introductory programming course(s) or sometimes not explicitly taught at the introductory level (Lister *et al.*, 2006a). Yet, they are typically not considered to be advanced topics either. They could therefore be seen as *middle-ground* skills. Even though such skills were discussed by Nelson *et al.* (2020) among the prerequisites, our findings show that they clearly have a different role than the knowledge of basic programming, such as control flow constructs or arrays. Overall, our findings about the role of these middle-ground skills suggest that they should be more explicitly recognized, and incorporated into the CS curriculum.

The remainder of this paper is structured as follows: Section 2 presents related work, Section 3 introduces the Instrument in detail, and Section 4 presents the methods used. After that, Section 5 presents the results from the written answers as well as the findings from the interviews, and Section 6 relates them to each other and discusses our findings. Finally, we conclude the paper in Section 7.

2. Related Work

In this section, we consider previous work related to this study. In Section 2.1 we explore differentiated assessments. Then, we present prerequisite skills that students learn during introductory programming courses in Section 2.2, and how these relate to the concept of abstraction in Section 2.3. Finally, we present research done on data structures and algorithms courses in Section 2.4.

2.1. *Differentiated Assessments*

In previous ITiCSE Working Groups, at least two approaches to differentiated assessments have been explored. In this context, a *differentiated assessment* is an assessment that aims to identify whether or not a student understands all topics in addition to providing a summative score. One of these approaches was explored by Luxton-Reilly *et al.* (2018), who aimed to create a set of small and self-contained questions that could be used together to explore a student's understanding of individual concepts. Another approach was explored by a 2020 Working Group (Nelson *et al.*, 2020). Rather than creating small self-contained questions, the 2020 Working Group examined to what extent it would be possible to identify individual areas of fragile knowledge from incorrect answers to questions that cover larger combinations of topics. The report first considers several CS questions on advanced CS topics, including the questions from the BDSI, which is a validated concept inventory on data structures (Porter *et al.*, 2019), and studies which prerequisite skills these questions depend on and whether they can diagnose difficulties with prerequisite skills. Principles are then discussed that help design differentiated assessments. Finally, examples of differentiated assessments are developed based on those principles. All such examples can be seen as *applied* questions (similar to our research), as they include a piece of code related to the advanced topics to be assessed, and they ask students to answer questions based both on the analysis of the code and their knowledge of advanced topics. For these reasons, these assessments are suitable to expose possible fragilities in prerequisites.

2.2. *Prerequisites from Basic Programming Courses*

In this paper we empirically evaluate the Instrument, therefore this section introduces research on prerequisites from basic programming courses and defines those prerequisites. Even though there is no clear definition of what introductory programming courses should cover (*e.g.,* the inclusion of object-orientation is often debated), there is a general consensus that they should include basic knowledge of programming constructs. This is reflected in the list of prerequisites proposed by Nelson *et al.* (2020), which is based on the ACM 2013 Curriculum Guide (Joint Task Force, 2013) and the list of core concepts taught in CS1 by Goldman *et al.* (2008).

Difficulties related to these skills can be explained by *misconceptions*, for example "errors in conceptual understanding" of programming constructs (Qian and Lehman, 2017). Misconceptions in introductory programming have been studied extensively (see *e.g.,* Fisler *et al.*, 2017; Sorva, 2013; Qian and Lehman, 2017). One particular finding by Fisler *et al.* (2017) is that some of these misconceptions are not overcome by students themselves, thus highlighting the importance of teaching them continuously throughout the education. However, as not all errors made by students are caused by a specific incorrect mental model, we use the more generic expression *fragile knowledge* suggested by Perkins and Martin (1985). The authors use the term to add nuance about students' knowledge in programming. The idea is that it is not necessarily clear cut whether students know or do not know something in the realm of programming, rather they may "sort of know" something, but not enough to solve a problem. They define fragile knowledge as knowledge that is partial, hard to access, and that is often misused. Specifically the authors present four categories of frailties. First is partial knowledge, where students' knowledge gaps impair their functioning. Second is inert knowledge, which is knowledge that the students possess but fail to muster when they need it. Third is misplaced knowledge, where the students' knowledge of something that is unsuitable for the current task impairs the students' ability to solve the current problem. Fourth is conglomerated knowledge, where the students' fail to follow the strict semantic and syntactic rules for the programming language, treating it more like a natural language.

Besides the skills that specifically relate to some programming construct, Nelson *et al.* (2020) identified also some broader skills related to reading and understanding code:

- **Tracing** is the ability to trace a piece of code, that is to simulate its execution, step by step, on a given input/instance, while keeping track of the state of the computation. This can be done with the support of some external representation such as a trace table to keep track of the variable values. Tracing requires correct knowledge of the programming language syntax and semantics of the constructs used in the code to be traced, that is, it requires an accurate mental model of the machine (Izu *et al.*, 2019). Moreover this skill includes the ability to rigorously follow the steps as specified by the program, without skipping parts, guessing the outcome or jumping to hasty conclusions.

In addition to *Tracing*, there are other skills that still relate to reading and understanding code, but further demand a high-level cognitive effort. They could thus be placed at the *Analyze* or *Evaluate* levels in the revised Bloom's taxonomy (Anderson *et al.*, 2001). They are as follows:

- **Metatracing** refers to the ability of realizing when tracing some specific portions of code, on some specific input, would be helpful in order to understand the code's behavior and purpose (Nelson *et al.*, 2020). In some ways it is hypothesis/goal driven tracing, either by a weak or strong hypothesis or goal. Identifying relevant and significant inputs for an algorithm/function/method is also covered by this skill. Although metatracing clearly requires tracing skills, it goes further than that. We consider tracing to be located in the *Apply* level and metatracing at the higher *Evaluation* level in the revised Bloom's taxonomy (Anderson *et al.*, 2001). Therefore, metatracing also has a crucial role in program comprehension tasks.
- **Program comprehension** is defined as the "process in which an individual constructs their *mental model* of a program" (Izu *et al.*, 2019, p. 28). As such, it is a broad and articulate process, which requires various skills (including basic knowledge of programming language syntax, tracing, and metatracing skills). However, in this paper we use program comprehension to refer specifically to the higher level skills that enables a student "to see the forest, not only the trees" (Lister

et al., 2006b, p. 122). In other words, to put together many single pieces of understanding into a consistent integrated model of what the program does and how it does it.

- **Reasoning about constraints** is the last prerequisite skill mentioned by Nelson *et al.* (2020). It refers to the ability of distinguishing what is known and what is not known about the program or its specification, identifying properties (*e.g.,* invariants, pre/post-conditions) and relationships among different parts of the program, and considering their implications on its behavior.

One of the main differences between basic prerequisite skills and these higher cognitive levels skills (metatracing, program comprehension, reasoning about constraints) is that such tasks involve some sort of *abstract thinking*.

2.3. *Prerequisite Skills and Abstraction*

While conducting research into differentiated assessments, especially when using the type of questions examined by Nelson *et al.* (2020), it is important to take abstraction into account due to the complexity in the questions. The role of abstraction in CS education is broadly acknowledged (Mirolo *et al.*, 2021; Aharoni, 2000). However, this is a complex "soft concept" that is hard to define formally, and which is difficult to characterize in this context, making it difficult to understand how to teach and assess it (Hazzan, 2008). Relating to programming and algorithms, Perrenet *et al.* (2005) proposes a hierarchy (called PGK hierarchy) of four abstract levels for the concept of algorithms: understanding the execution on particular inputs on a specific machine (Execution level); grasping the algorithm as described in a programming language by a program (Program level); envisioning the algorithm as an abstract object independent from any specific language (Object (algorithm) level); and finally perceiving the algorithm as a strategy to solve the implied problem (Problem level). In the program comprehension process, all these levels need to be considered. For instance, tracing pertains to the Execution level, abstract tracing to the Program level, describing what a method does is at the Object level, whereas summarizing the purpose of a program is at the Problem level.

Statter and Armoni (2020) explored abstraction in programming education further. Based on the PGK hierarchy, they have identified six operational dimensions that define CS abstractions: 1) the use of an algorithm, 2) working only at the algorithmic level (Object level), 3) using black boxes, 4) the ability to distinguish between different levels of CS abstraction, 5) the ability to move freely between different levels of CS abstraction, and 6) the ability to decide at which levels of CS abstraction to work. They conclude that since abstraction is a fundamental idea of CS, it should probably be reflected in the learning goals of CS curricula. However, teaching CS abstractions is a very challenging task. Students tend to perceive problems and solutions at the lower levels of PGK hierarchy, and they often have problems working on higher levels such as black boxes. Further, several authors identify the need for computer scientists to work with creating abstractions at different levels (Hartmanis, 1994) and being able to think at different levels of abstraction (Wing, 2006).

2.4. *Research on Data Structure Courses*

Students' performance in data structure courses have been investigated previously. Corney *et al.* (2014) asked students in their second programming course to explain in plain English object-oriented data structures problems that involve recursion. The results showed that many students struggle with this task, and the authors found a strong correlation between students' ability to read and explain code at abstract level and their performance in writing code on data structures. While this study did not investigate the causes of such difficulties, results from a study conducted by Valstar *et al.* (2019) reported a strong correlation between answering questions on prerequisite knowledge and the students' scores on the final exam. More than 30% of students could not answer questions on pointers and trace recursion, which underlines the importance of creating differentiated assessments.

Students' performance has also been investigated without focusing on the impact of prerequities (Danielsiek *et al.*, 2012; Tenenberg and Murphy, 2005; Zingaro *et al.*, 2018). Student misconceptions concerning heaps, hashtables, and with recursive structures (*e.g.,* linked lists, trees, and binary search trees) (Danielsiek *et al.*, 2012; Zingaro *et al.*, 2018). Further, Tenenberg and Murphy (2005) found issues with students grasp of tracing recursive algorithms and computing the run-time efficiency of algorithms. They also found that students performed best on questions related to the interface of stacks, queues, and trees.

Aharoni (2000) took a qualitative approach, investigating students' cognitive processes through semi-structured interviews. The author investigated their perception of data structures (*e.g.,* arrays), arguing that understanding develops over a continuum of levels of abstractions, which is rooted in constructivism, and introduces the concept of *programming-context thinking* vs. *programming-free thinking*. Programming-context refers to levels of abstractions that are closely aligned to the underlying notional machine, hence explicitly linked to the programming language. In other words, understanding occurs with the reference to implementation. Instead, programming-free thinking occurs when understanding refers to the abstract form of a data structure without reference to any particular implementation. This paper, like the aforementioned research, also uses a qualitative approach, but with the aim of exploring fragile knowledge in the context of a data structures, rather than their perception of the data structure.

3. The Instrument

In this section we describe the Instrument under evaluation, which can be found in its entirety in Appendix A. The Instrument is a slightly varied version of the one proposed by Nelson *et al.* (2020, Appendix M.1.4). It consists of the listing of a Java program and a questionnaire of related questions.

The Instrument's program (identical to the original by Nelson *et al.* (2020)) implements a *queue* as an expandable circular array, equipped with two indices, `lo` and `hi`, that identify the head and tail of the queue respectively. The variables are partially obscured, and the circular nature of the data structure means that `lo` is sometimes a number higher than `hi`. An integer `N` keeps track of the number of items currently in the queue. Hence, the *circular distance* between `hi` and `lo` is always equal to `N`. The class contains two methods (`insert` and `remove`) that implement the typical operations of the queue, and a `rebuild` method that is used by `insert` to double the size of the circular array whenever it is filled. Hence, the array length is always greater than the queue size, and is always a power of two. The `rebuild` method also rearranges all the elements that are currently in the queue so that, after the method is executed, they are all placed at the beginning of the array. To implement this, the method uses modulo operator.

The Instrument's questionnaire contains seven multiple-choice questions (Q1–Q3, Q6, Q8–Q10). In addition, there are two open-ended questions (Q4a and Q11) that ask students to write their answers in natural language . The rest of the questions require to trace a fragment of code that calls methods defined in the listing and to provide some representation of the resulting state of the data structure (Q4b, Q5, and Q7). Questions 1 to 7 aim at assessing the students' comprehension of the given code. In order to answer correctly, students need to understand both the mechanics of the implementation (*i.e.,* the circularity of the array and the rebuild strategy) and to comprehend the resulting behavior (*i.e.,* the FIFO policy). The first question asks the student to identify which data structure is implemented by the class. Questions 2–6 aim at verifying that the answer to the first question was not a guess, as well as the student's understanding of various details of the implementation. Question 6 was not present in the original assessment proposed in Nelson *et al.* (2020), but we added it to address the circularity of the queue and the fact that the size of the array is always a power of two.

In the context of the PGK hierarchy (Perrenet *et al.*, 2005) described in Section 2.3, questions Q1 to Q7 can be situated at different levels. For instance, Q1 (identifying that the code implements a FIFO data structure) is at the Problem level, Q4b (trace the `rebuild` method) is at the Execution level, and Q6 (which states are possible?) is at the Object (algorithm) level.

The remaining questions 8–11 focus on algorithm analysis, which is an advanced topic in this context. Thus, in this study, we focus on questions 1–7 that we hypothesize are able to reveal fragile knowledge in prerequisites.

3.1. *Prerequisite Skills Assessed by Each Question*

Nelson *et al.* (2020) discusses the prerequisites and advanced skills assessed by each question, and which individual fragilities would be made visible from the students' answers. This is done in two separate parts of the WG report, namely Appendix M.1.2 and M.1.5. We summarize here the claims related to questions 1–5 and 7 for future reference in the rest of the paper. It is worth mentioning that no formal empirical evaluation was carried out by Nelson *et al.* (2020), and the claims are based on the authors' argumentation.

Q1 A wrong answer could be attributed to bad knowledge of advanced topics (understanding data structures and ability to distinguish them) or to bad program comprehension skills. However, the correct answer could be guessed by doing cursory examination of the code since the wrong options can be excluded easily (Nelson *et al.*, 2020, p. 38).

Q2 A student who does not understand the circular nature of this queue implementation might wrongly choose option (d). The question, however, does not assess circularity explicitly and the correct answer might also be guessed by looking at some surface features of the code. The question also addresses the possible confusion between the length of the array and the number of elements currently in the queue.

Q3 Understanding the circularity and the rebuilding strategy are essential for answering this question (with full awareness). A student who does not understand the circular nature of this queue implementation might wrongly choose option (b). However, as for Q2, this question does not assess circularity explicitly and the correct answer might also be guessed by looking at some surface features of the code.

Q4 Part (a) requires students to reason about constraints. A relevant observation to answer this question is that `N` should always be equal to the *circular distance* between `hi` and `lo`, which does not hold in the state given in the question. Part (b) assesses knowledge on operators (modulus) and arrays, and requires tracing code on a specific instance.

Q5 Understanding the circularity and the rebuild strategy are essential for answering this question as well. Different approaches can be used to answer the question. One could trace the code line by line (but it would be a long, tedious, and possibly error-prone task), or trace the code at a higher level, relying on their understanding of how the queue is implemented. The second part of the question (how many times the rebuild method has been called) also assesses prerequisite knowledge on conditionals, and ascertains that the rebuild strategy has been understood. To understand the rebuild strategy, one needs knowledge about conditionals and arrays.

Q7 This question assesses knowledge about values and references.

Beside the above remarks on specific questions, the WG report (Nelson *et al.*, 2020, p. 23) claims that questions 1, 2, 3, and 4 "require close inspection of the code to figure out how the data structure works, which in turn require skills related to code comprehension" (Nelson *et al.*, 2020, p. 23). The WG report further claims Q2, Q3, and Q4 to be connected, in that answering correctly to Q2 and Q3 is a "good step toward" answering Q4 correctly (Nelson *et al.*, 2020, p. 38).

Many of the claims reported above state that understanding both the circular nature of the array and the rebuilding strategy are important steps towards answering the questions correctly. It is not clear, however, whether these skills are considered prerequisites (related to code comprehension) or as advanced skills related to data structure knowledge. Nevertheless, since there was not any specific question to assess circularity, as also noted by Nelson *et al.* (2020), we added a new question for this, with the following rationale:

Q6 This question assesses the understanding of circularity. It also relates to the possible length of the circular array implementing the queue. Answering the question requires reasoning about constraints.

In this paper, we evaluate the Instrument's validity according to Kane's framework (Kane and Bejar, 2014). We do this by using the questions in actual learning settings in an empirical study. Thus, we not only used the question set in actual courses, but also interviewed the learners to understand how well the questions are able to differentiate between prerequisite skills and advanced skills. We did this by probing on students approach to solving the problems in the Instrument.

4. Study Methods and Design

Section 4.1 describes how students were invited to participate in the study, how the written answers were collected, and how the interviews were collected. Section 4.2 then describes the theory and details behind Qualitative Content Analysis (QCA) (Schreier, 2014), which was used to analyze the collected data. After that, Section 4.3 provides background information about the learners and the learning context used in the study, and finally Section 4.4 describes the modifications made to the Instrument in order to make it suitable for the learners.

4.1. *Data Collection*

We collected data from a total of three courses, two held in Sweden and one in Finland. Students were invited to participate in the study by an e-mail sent to their university e-mail account 2–3 weeks before the final exam in each course. The invitation instructed students to answer the questions of the Instrument (attached to the e-mail), and send back their answers (either typed, or handwritten and scanned) along with any notes. Students who did so within the 7 allotted days were invited to an interview.

In the e-mail, students were informed that participation was voluntary and would not affect their grading of the final exam. Students were motivated to participate in the study by highlighting that the participation would be helpful to prepare for the final exam, as they would get feedback on their answers.

Before the interviews, we marked students' written answers either as correct or incorrect. If a written answer included incorrect reasoning, it was graded as incorrect to support investigating the student's reasoning further in the interview. Fig. 1 (in the Appendix) shows an overview of the written answers.

Interviews. A total of 18 students were interviewed as depicted in Table 1. An additional 10 students submitted solutions, but were either unavailable for interview, or

Table 1
Details of the three courses examined in this paper

	Sweden		Finland
Course	A	B	C
Programming language	C++	Java	Python
Completed the Instrument	8	4	16
Interviewed	7	1	10

submitted their answers too late. The interviews were conducted by a teacher at the same institution as the course was given, in the local language. Since interviews were conducted before the final exam, we ensured that students were interviewed by a teacher that was not involved in the course the student was taking. This is to avoid a situation where students are hesitant to express their uncertainties due to fear of affecting their grade. Before each interview, the interviewee's answers were graded in order to identify questions that were answered incorrectly. The interviewees were not informed of whether their answers were correct or not until after the interview, at which point they received written feedback to their answers. Each interview lasted for about 30 minutes, and was recorded with the consent of the interviewees.

In order to explore students' approach to answering the questions, the interviews were conducted as semi-structured interviews (Adams, 2015). As such, the interviews were structured around a set of pre-determined questions with the aim of leading the interviewee into the right topic. For each question, the interviewer asked the student to describe how the interviewee arrived at their answer (reminding them of their written answer if necessary). Depending on the interviewee's answer, the interviewer then asked follow-up questions with the goal of identifying any fragile knowledge or skills the student had. As we were particularly interested in any fragilities that caused incorrect answers, the interviewer focused on questions where the student answered incorrectly. Thus, questions with incorrect answers were discussed more in depth during interviews, while questions answered correctly were sometimes skipped altogether due to a lack of time. During the interview, interviewers were careful not to reveal correct answers or otherwise influence participants' thinking for subsequent questions. As some of the questions depend on each other, the questions were discussed in the order they appear in the Instrument.

As previously stated, the follow-up questions aimed at clarifying students' reasoning and at uncovering misconceptions. This follows the guidelines mentioned in Adams (2015). To illustrate these follow-up questions, we provide some examples:

- "What is the difference between a queue and a priority queue?" – Question 1
- "Why did you choose this option and none of the others?" – Question 3
- The interviewer shares a notepad and writes the initial state given in the question. "Please write below, if you executed `rebuild` step by step, what would happen?" – Question 4b
- "What operations lead to this particular case?" – Question 6
- "On which line does the length of the array `A` increase?" – Question 6

4.2. *Qualitative Content Analysis*

To analyze the interviews, we used QCA as described in Schreier (2014). The purpose of the method is to systematically describe the meaning of qualitative interview data. Researchers split the data into mutually exclusive *segments*. Each segment is given a single *code*, which represents one particular meaning. Moreover, the codes form a hierarchy (a *coding frame*), typically with two main groups. The analysis is conducted by following a predefined series of steps. The objective of the analysis is to describe the data through coding in a compact way. The results of qualitative analysis may include representation of code frequencies, but the coding frame might be the end result as well. Like *quantitative* content analysis, QCA uses a systematic approach to content analysis. However, instead of merely coding segments based on objective criteria, QCA also considers the latent and context-dependent meaning of the content (*i.e.,* the researcher's interpretation) (Schreier, 2014). Note that onwards, we use the terms *label* and *codebook* for the corresponding terms *code* and *coding frame* introduced by Schreier (2014). This is to use the words *code* and *coding* only in the connotation of program code.

Codebooks in QCA have three main requirements (Schreier, 2014). *Unidimensionality* means that one codebook, the hierarchical category of labels, should cover only one concept. *Mutual exclusiveness* has two subrequirements: the labels of the same codebook must be mutually exclusive, and one segment can have only one label from the same codebook. This is to ensure that a segment is not assigned two opposite meanings. *Exhaustiveness* means that all relevant aspects of the material are covered by a label.

In our work, the codebook requirements of QCA are met as follows. Unidimensionality is met, because we have only one main concept: the fragility, meaning the hierarchy of skills and knowledge. Mutual exclusiveness of labels is taken as an assumption, starting with the deductively predesigned codebook in Nelson *et al.* (2020). Mutual exclusiveness of segments means that a student's answer to a particular Instrument question can have multiple labels only if the answer is split into different segments, each labeled as a different fragility. As each segment is given only one label, we are able to pinpoint which exact pieces of the interview recordings are the supporting evidence for one specific label. Moreover, by mutual exclusivity we ensure that two labels indeed represent different phenomena. Finally, we assume that the predesigned codebook is designed to be exhaustive, but we reserve the right to modify the codebook to fulfil this requirement.

We believe QCA is a suitable method for analyzing the Instrument empirically for the first time while being open for alternative hypotheses. In our research setting, the codebook is needed to represent students' different fragilities with prerequisite and advanced skills. Moreover, latent and context-dependent analysis of meaning, in other words, interpretation of students' answers, is needed for two purposes. First, the initial codebook, proposed by the designers of the Instrument (Nelson *et al.*, 2020), describes fragilities on a generic level, which requires situation-dependent interpretation, *i.e.*, there are no objective decision rules for labelling categories. Second, the initial codebook does not yet have empirical support for its labels and structure. Thus, the qualitative evidence in our work would support quantitative testing of the Instrument later.

The steps of our analysis process are described below and correspond to those described in (Schreier, 2014). The process itself is divided into two consecutive phases. The *pilot phase* tests the codebook with a subset of the material, while the *main phase* analyzes all the material with an established codebook. While the order of the steps describes a deductive process, some steps were iterated to utilize inductive reasoning.

- **Selecting material.** A diverse subset of data was selected to build an initial codebook in the pilot phase. Each team of two researchers in Sweden and Finland selected three interviews in the local language for the pilot phase. At least one of these interviews were used by both researchers in the team.
- **Building a codebook.** This involves building the hierarchy of categories which are used to describe the pieces of the material. As Nelson *et al.* (2020) had proposed a codebook for general prerequisite skills, the goal was to build a codebook for frailties that were excluded by Nelson *et al.* (*e.g.,* related to data structures and algorithms). Our hierarchy is made of the frailties that students display, *i.e.*, we build the codebook in a concept-driven way similarly to the method explicitly proposed by Nelson *et al.* (2020). At this point, the researchers from Sweden and Finland discussed their proposed additions to the codebook from the previous phase, so that they could be merged into one.
- **Segmentation.** The two teams of researchers from Sweden and Finland each analyzed the previously selected set of interviews independently. They selected segments, i.e., pieces of discussion and activity, so that each segment could later be assigned a single label. The Swedish pair selected segments from transcribed interviews. The Finnish pair instead viewed their video recordings and noted quotes, descriptions of activities, and short hypotheses in the local language along with timestamps. Due to mutual exclusiveness, the segments could not overlap. The length of a single segment varied from a single phrase to a couple of lines of dialogue. Primarily, we used a thematic criterion for segmentation: a student's written or spoken answer to each question of the Instrument formed a segment. If a segment seemed to contain evidence for multiple labels, the segment was further split to indicate which phrases or passages of dialogue represent a single label.
- **Trial labelling.** After segmenting the interviews, the researchers independently labelled each segment with a single label from the codebook. This was done in the pilot phase to study to which extent the original codebook by Nelson *et al.* (2020) with the proposed additions could be applied to the interview material. To verify the consistency of the labelling, the researchers in each pair compared their labelling and discussed any discrepancies until an agreement was reached. These discussions were partially conducted in-person in Sweden, but both countries relied heavily on online meetings.
- **Evaluating and modifying the codebook.** The codebook was evaluated against the following criteria: A codebook should be consistent so that two labelers assign the same label to the same segment. Moreover, the validity of the codebook means exhaustiveness (extensive coverage of the material) and

relevance to the research questions. In this phase, the main analysis phase, the Swedish and Finnish researchers discussed together how well the given codebook (Nelson *et al.*, 2020), with the proposed additions, applied to the data and which cases (segments) were difficult to assign a label. All of these discussions were conducted through a series of online meetings. The labelling was also available to all researchers through shared documents. Recordings were, however, not shared outside of the country they were recorded in for privacy reasons. As a result of this phase, the codebook was revised to meet the abovementioned criteria.

- **Main analysis.** In the main phase, the rest of the material was labeled with the revised codebook. We used two labelers per interview similarly to the trial labelling. The Swedish and Finnish results were then combined for the next step.
- **Presenting and interpreting the findings.** The labeled segments from all students were translated in to English, and sorted first by the question, and second by the label (specific fragility). Then, the codebook was presented with illustrative quotes forming a text matrix. Each column of the matrix was a student's interview answers to a particular Instrument question. Correspondingly, rows corresponded to labels and each cell contained the English-translated segment (direct quote or description of activity) that acts as evidence for the label. All 7 researchers in the team examined the text matrix, and discussed the contents through online meetings until an agreement was reached. In particular, the focus of these discussions was if a segment contains enough evidence for the assigned label and whether or not the label was correctly assigned. In some cases, these discussions led to further revisions of the codebook, which meant that this and the previous step had to be re-visited iteratively.

Finally, a note about transcription. In Sweden, one interviewer transcribed the interviews fully in the local language as intelligent verbatim transcription. The transcripts contained literal meaning of the dialogue between the interviewer and the interviewee, omitting nonverbal communication and utterances, with timestamps locating discussion on each Instrument question. In Finland, transcription was done selectively after the analyzing pair agreed on interview quotes that represent the evidence for a single label. However, when one research team member requested more information on a labeled segment in the *Evaluating and modifying the codebook* step, another team member reviewed the video or transcript representing the segment, and provided longer translation with more context.

4.3. *Context*

As mentioned previously, data was collected from three courses, two held in Sweden and one in Finland. All three courses were large programming courses with 85–250 students attending each one. The courses were all given during the second year of their programs, and aimed at teaching data structures and algorithms to students who already knew programming from a previous CS1 course. The textbook for the courses was OpenDSA (Shaffer *et al.*, 2011), an open, online textbook for Data Structures and Algorithms. In addition to textual descriptions of the material, OpenDSA also contains interactive components such as detailed visualizations and animations of many data structures and algorithms. In particular, OpenDSA covers queues, both at an abstract level (*i.e.*, the details of the ADT), and different implementations of this ADT. One of the implementations covered in detail is a circular array implementation that is similar to the one in the Instrument. In addition to the material covered by OpenDSA, the courses also involve a number of computer lab assignments where students implement and use various data structure. While none of the assignments require students to implement a circular queue like the one in the Instrument, there are assignments that require utilizing queues (*e.g.,* for graph traversals). All courses ended with a final exam.

Even though one course was given for CS programs, and the remaining two were given for non-CS programs, all courses focus on introducing data structures and algorithms so that students can select and use appropriate data structures in their work. They also focus on the ability to reason about the time-and space-complexity of programs. As a way to reach these goals, the courses expose students to various implementations of common data structures. This serves two purposes: first, they are used as examples to practice algorithm analysis, and second, a rough understanding of the implementation makes it easier to remember the characteristics of the data structure, which in turn makes it easier to reason about them in the context of larger programs. For this reason, the Instrument is relevant to the courses, even though none of them have the explicit goal that students need to know how to implement a circular queue. Rather, the Instrument is relevant because it focuses on analyzing a small implementation of an ADT (a queue) that should be familiar to the students. The focus is thus on the analysis and reasoning, rather than having previously memorized the implementation of a circular queue.

Sweden. Course A in Sweden is given early in the second year for two bachelor CS programs. Students from both programs have studied programming for a year before the course. Students from one of the programs start out programming in Python, transition to C++, and later work in other languages such as Ruby during their first year. Students from the other program are initially taught Ada, but quickly transition to C++ and continue working with C++ during the remainder of their first year. As all students have worked with C++, the course is taught in C++.

Course B is similar to course A, but is given for non-CS majors attending a five year program. These students have previously taken two introductory courses in Java, and this course is therefore taught in Java. Apart from this difference, the courses are very similar in nature, including the final exam. Both courses are given in Swedish. The final grade in both courses is entirely based on the result from the final exam.

Finland. The course in Finland is targeted to the second year students in several five year non-CS bachelor engineering programs. All students have previously taken at least one introductory programming course in Python, and therefore the course is taught in Python. The final grade is the weighted average of weekly assignments and the final exam.

4.4. *Adapting the Instrument to Different Programming Languages*

The questions were translated into the local spoken languages of the two institutions and the original Java program was translated into C++ (see Appendix B) for course A in Sweden, and into Python (see Appendix C) for the course in Finland to correspond the programming language used in the courses.

The C++ and Python implementations were not designed to exploit all the features of a specific language or even to be idiomatic for the target language. Instead we tried, in each language, to preserve the abstract description we presented in Section 3. The given implementation should present to the student a clear model of the mechanics of the data structure, without resorting to language features not necessarily known to beginners. We deliberately avoided using any high level features available to professional programmers, which might obscure the educational goals (namely, analyzing a specific implementation of the queue ADT).

C++ version. Although the C++ translation almost directly maps to the Java implementation, there are three significant differences:

1. The Java version makes explicit that the `Key` parameter for the generic type refers to a `Comparable` type, which is not possible in C++ (prior to C++20). This is not utilized in the Java implementation, but it suggests the data structure is a container for ordered objects, thus making it superficially suitable for a priority queue.
2. Similarly to the Java version, the C++ version uses a fixed size array (allocated using `new`) to implement the circular array. Contrary to the Java version, however, this requires storing the length of the array separately. This is done in a variable called `A_length`.
3. The C++ version explicitly frees the memory in the rebuild, where the Java version relies on the implicit garbage collector.

Python version. The Python translation was intended to reproduce the mechanics of the data structure in the syntax most familiar with the students, even though a professional Python programmer would likely have used a different approach. It has three significant differences:

1. Python has no syntactic way to reduce the visibility of methods, thus the rebuild method is not different in any way from the insert and remove operations. We decided not to use the conventional naming with the underscore (`_rebuild`) because, even if it is an explicit convention of the language[1], it could be a source of confusion for Python novices.
2. Python does not have static typing, thus the methods have a signature which does not explicitly state the intended type of the parameters and return values. We avoided the use of type hints since this aspect was considered not relevant for the Instrument (it concerns a generic collection of data) and their use is still uncommon.

[1] `https://docs.python.org/3/tutorial/classes.html#tut-private`

3. The underlying array of values was implemented by using a Python list, a basic data type in that language, but in fact a dynamic structure in itself. Using actual (statically allocated) array types would have increased the complexity of the code, possibly obscuring the key parts. The explicit allocation of the slots of the array was simulated by using `None` elements.

5. Results

This section presents the results of the analysis conducted as described in Section 4. The first subsection presents an overview of the written answers to the Instrument. This is then followed, in the second subsection, by the results of the qualitative analysis of the interviews.

The remainder of this paper refers to individual students using a single letter followed by an integer. The letter corresponds to the programming language used in the course that the student attended (C for C++, J for Java and P for Python). As different programming languages were used in all three courses, this uniquely identifies the course that the student attended. The integer then identifies individual students within each course. For example, C2 is student number 2 in the course that used C++ (i.e., course A in Sweden).

5.1. *Written Answers*

Table 2 presents the written answers to questions 1–7 of the Instrument, given by the 18 students that were interviewed. The answers from the 10 students that were not interviewed are left out due to space limits. Only 4 students answered questions 4b and 6 correctly. The other questions mostly got correct answers. We next summarize the mistakes found in the written answers, question by question.

Each of Q1, Q2, and Q3 had a frequent incorrect answer. In Q1, the only incorrect answer which the interviewed students selected was option (c), that is, the data structure would be a priority queue. In the answers for the 10 students who were not interviewed, one picked (option a, Stack) and another picked (option d, Union find). Similarly, all incorrect answers to Q2 indicated that the number of elements in the data structure was given by the length of the array (option b), rather than the value of the variable `N` (option a). For Q3, all students who answered incorrectly were interviewed, and all incorrect answers are thus present in Table 2. Three of these answers incorrectly established `lo < hi` as an invariant (option b), one believed `hi == N` to be an invariant (option d), and another answered that none of the proposed options were an invariant.

In Q4a, most students correctly found that the state was invalid by counting the number of non-zero elements in the array. Some students (*e.g.,* C7) also identified that the modular difference between `lo` and `hi` did not match the value of `N`. Interestingly enough, three students provided a sequence of insertions and removals that

Table 2

Written answers to the questions 1–7 of the Instrument, given by the 18 students that were interviewed. The line in *italic* contains the correct answer to each question and the last line contains the total number of correct answers. The other rows represent each interviewed student. Notation: ✓ correct answer (otherwise student's answer); ∅ none of the options in a multiple-choice question; ✗ incorrect reasoning; * answer matches closely the indicated item

	1	2	3	4a	4b			5					6	7	
					A	lo	hi	A	lo	hi	N	rebuild		A	B
Correct	***b***	***a***	***a***	***-***	***1, 3, 0, 0, 0, 0, 0, 0***	***0***	***2***	***3***	***0***	***1***	***1***	***1***	***a, c***	***3***	***2***
J1	c	b	✓	-	-	-	-	3	✓	✓	✓	3	a	1	✓
C1	✓	✓	✓	✓	1, 3	✓	✓	✓	✓	✓	✓	-	a, b, c, d	✓	✓
C2	✓	✓	✓	✓	3, 8, 4, 1, 0, 0, 0, 0	✓	4	✓	✓	✓	✓	✓	✓	1	✓
C3	✓	✓	✓	✗	unclear	✓	✓	✓	✓	✓	✓	✓	a	✓	✓
C4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C5	c	b	✓	✗	3, 8, 4, 1	✓	4	✓	✓	✓	✓	✓	e*	1	✓
C6	c	✓	✓	✓	1, 3, 8, 4	✓	✓	3	✓	3	✓	✓	c, d, e	✓	✓
C7	✓	✓	✓	✓	1, 3, 8, 4, 0, 0, 0, 0	✓	4	✓	✓	✓	✓	✓	a, b, c, d	✓	✓
P1	✓	✓	b	✗	✓	✓	✓	✓	✓	✓	✓	✓	∅	✓	✓
P2	✓	✓	✓	✓	1, 3, 8, 4, 0, 0, 0, 0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
P3	c	✓	d	✓	8, 4, 3, 1	✓	✓	3	✓	✓	✓	3	a, b, e	[3]	[]
P4	✓	✓	∅	✗	3, 8	✓	✓	✓	✓	✓	✓	✓	a	1	✓
P5	✓	✓	✓	✗	8, 3, 8, 0, 0, 0, 0, 0	✓	✓	✓	✓	✓	✓	✓	a, b	1	✓
P6	c	✓	b	✓	1, 3, 8, 0, 0, 0, 0, 0	✓	✓	✓	✓	✓	✓	✓	c	✓	✓
P7	✓	✓	✓	✓	1, 3, 8, 0, 0, 0, 0, 0	✓	✓	✓	✓	✓	✓	✓	a, b, c, d	1	✓
P8	✓	b	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	a, b, c, d	1	✓
P9	✓	b	b	-	3, 8, 4, 0, 0, 0, 0, 0	2	✓	✓	✓	✓	✓	✓	✓	`None`	`None`
P10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	c	✓	✓
Total	**13**	**14**	**13**	**11**	**4**			**14**					**4**	**9**	

allegedly would produce the state in the question. In the second part of the question, Q4b, most students failed to trace the behavior of `rebuild` from the (invalid) state presented in the first part. Only six out of the 28 students answered the question correctly. All the incorrect answers failed to compute the contents of the array, but most managed to predict the values of `lo` and `hi`. A number of patterns are visible from the contents of the array. Some students (*e.g.,* C1 and P3) answered with an array with less than eight elements. Some students copied three (*e.g.,* P5) or four (*e.g.,* C2) rather than two elements. Finally, some students (*e.g.,* C6 and P6) reordered the elements according to the logic in `rebuild`, while others (*e.g.,* C2 and P9) incorrectly preserved the original order.

While Q4a seems easier than Q4b (11 vs. 4 correct written answers), one student's answer was graded as incorrect for Q4a but correct for Q4b. P1 identified an incorrect invariant `lo < hi` in Q3. Superficially, they answered correctly in Q4a that the state is impossible, but used two arguments, the first false and the second true: (i) the invariant `lo < hi` does not hold for the state, and that (ii) the value of `N` should be 4, not 2, because it is the number of elements. Because the student's answer included incorrect reasoning, it was graded as incorrect. For Q4b, P1 only provided the correct end state.

Most students answered Q5 correctly. All six incorrect answers had an array with an incorrect length (either 1, 3, or 4 elements instead of 2). Only five students in total answered Q6 correctly. Here, students were asked to determine which of the five states were possible after a partially unknown sequence of insertions and removals have been executed. Fourteen of the incorrect answers included one of the options where the length of the array would not be possible (options b and d). A less frequent mistake was not to select either option (a) (4 students) or (c) (6 students). Finally, student C5 found no option to be viable and instead provided another state that was similar to option (e). Student P1 believed that no option was reachable since elements are inserted in the beginning of `A`.

Finally, for Q7, the most common answer (8 of 11 incorrect answers in total) was that `a == 1` (instead of `a == 3`) after executing the code. The remaining incorrect answers stated that `a` and `b` contains arrays (P3 and P16), or that `a == b == None` (P9).

5.2. *Codebook of Fragilities*

The labelling of the excerpts from the interviews resulted in seven labels describing different fragilities. Each label is based on the observations from previous work (Nelson *et al.*, 2020), with minor alterations to align with the data. We group the labels into three major groups.

The first group consists of three labels that represent misconceptions about specific aspects of the notional machine. Thus they cover skills that would be considered prerequisites to a data structures and algorithms course. These closely match the codebook from Nelson *et al.* (2020) and are defined as follows:

- **Operators** Excerpts with this label show fragile "skills related to operators [which] cover both being able to use arithmetic and comparison operators. Examples of these include questions related to operator precedence and Boolean logic" (Nelson *et al.*, 2020, Table 1).
- **Arrays** Excerpts with this label show fragile knowledge about "declaring and indexing arrays [including] what happens when the index is out of bounds" (Nelson *et al.*, 2020, Table 3). The label also includes fragile skills in using "loops to iterate arrays, either using regular loops and indices, or any dedicated syntax for the task" (Nelson *et al.*, 2020, Table 2).
- **References** Excerpts with this label show fragile skills in differentiating "between values and references (or pointers) to values", and identifying "differences between making copies of a value and a reference to a value" (Nelson *et al.*, 2020, Table 3).

The second group of two labels represent higher level skills. These are skills that are in some way introduced early on in CS education, but which typically take time to become proficient in. As described in Section 2.2, these are partially based on the higher-level skills described in Nelson *et al.* (2020), but also on overarching statements made by the authors of the Instrument. The labels are defined as follows:

- **Reasoning about constraints** As defined in (Nelson *et al.*, 2020), this label describes fragile knowledge when reasoning (semi-formally) about constraints enforced in the code. For example, the student may reason incorrectly, because they fail to identify some constraint enforced by the code; fail to utilize identified constraints; or establish contradictory or false constraints.
- **Program comprehension** Excerpts with this label show fragile knowledge when building a complete, consistent and integrated understanding of the behavior of the program by means of an accurate inspection of the code. In particular, we use this label for claims of students who make an educated guess on a particular program behavior without basing their argument on the program code, or specifically motivates their claims. For example, a student may guess the purpose of the program based on surface features, such as the names of functions or variables. While this skill is not formally present in the codebook proposed by Nelson *et al.* (2020), the authors informally noted that program comprehension was required to correctly answer some questions in the Instrument. As this label is closely related to the previous one, we distinguish them based on the detail of the observations. Detailed and more formal observations were tagged as *Reasoning about constraints*, while informal observations (*i.e.,* some level of guessing) were tagged as *Program comprehension*.

The last group of two labels refer to concepts that are specific to the advanced topic studied in this paper, data structures and algorithms. Therefore, these are not present in (Nelson *et al.*, 2020). The labels are defined as follows:

- **DS knowledge** Excerpts with this label show fragile factual knowledge about different Data Structures (DS), including their semantics or how they are implemented. Difficulties in distinguishing between different data structures is also marked with this label. The excerpts with this label typically show students making incorrect statements about some data structure. This was mentioned as an issue in (Nelson *et al.*, 2020), but not formally defined in detail, as the focus was on the prerequisite skills and not on the advanced topics.
- **ADT vs. implementation** Excerpts with this label show confusion between the abstraction provided by an Abstract Data Type and its implementation. For example, in the context of the Instrument, a student may confuse the queue (an abstract container of elements) with the array (that is used to store the elements in the queue). In particular, the former abstraction never contains "empty" elements, while the latter has to deal with empty positions in the array.

5.3. *Interview Excerpts Revealing Fragile Knowledge*

Table 3 summarizes what fragilities were found for each of the 18 interviewed students. The table is based on the text matrix discussed in Section 4.2. Notice that this table is a high-level representation of the data: a single student's answer to a single question may consist

Table 3

Outcome of the qualitative analysis of interviews and open answers. Labels 1–3, 4a, 4b, and 6–7 indicate some fragility related to the respective question for a particular student. Legend: * Python-specific issues with array iteration, † relying on the surface features of the program, ‡ correct written answer.

	Basic prerequisites			High-level prerequisites		Advanced topics	
Student	Op.	Array	Ref.	Reasoning constr.	Program compr.	DS knowl.	ADT vs. impl.
J1	–	–	7	6	1†, 4b, 5†	1	2
C1	–	4b	–	4a‡, 6	4b†	–	–
C2	–	–	7	–	4b	–	4a‡
C3	–	–	–	6	1‡†, 4a, 6†	1‡	–
C4	–	–	7‡	–	–	–	–
C5	–	–	7	–	1†, 2, 4a, 6‡	–	–
C6	–	–	–	6	1†, 4b	1	–
C7	–	–	–	6	–	–	–
P1	–	6*	–	4a	–	–	6
P2	–	4b*	–	–	–	–	–
P3	–	4b*	–	3, 4b	1†	–	–
P4	4b	–	–	–	6	–	–
P5	–	–	7	4a, 6	4b, 6	–	–
P6	–	4b*	–	–	1†	1	2‡
P7	–	4b*	7	6	–	–	–
P8	–	–	7	6	–	–	–
P9	4b	4b*, 7*	–	3	–	–	2, 3
P10	–	–	–	6	–	–	–
Total	**2**	**7**	**7**	**12**	**10**	**4**	**5**

of multiple segments, and thus have multiple labels (*e.g.,* student C1 is labeled both *Arrays* and *Program comprehension* in Q4b). The mutual exclusiveness requirement discussed in Section 4.2 still holds, as each code was assigned to a different segment in the text matrix.

The remainder of this section contains a detailed description of the fragilities found, and how they appeared in the interviews. The notation in the students' quotes is as follows: square brackets [] denote transcriber's clarification, dots [...] indicate omitted speech, and the `monospace` font is our post-transcript choice to emphasize when the student refers to the program code.

5.3.1. *Basic Prerequisites: Operators, Arrays, and References*

Two students showed fragile knowledge concerning *Operators*, and in particular the modulo operator, when tracing the `rebuild` function in Q4b. This evidence is in the form of students who made incorrect statements about the meaning of the operator, which is illustrated by quotes from P4 and P9:

> *Isn't that percent sign that the integer is taken into account, so that when that becomes less than one then it is like zero.* [Student P4]

I would remember that it [referring to the modulo operation] is a truncating division. [Student P9]

The most common fragility among the basic prerequisites concerns *Arrays*, and was found in seven students. One such issue was uncertainty about array indexing. For example, P9 implied that in the assignment `tmp[i] = A[j]`, that `tmp[i]` would remain at its current value if `A[j]` is out of bounds. This is visible when the student traces `rebuild` in Q7 during the interview. After loop iteration `i == 1` the student makes the following statement.

`tmp = [3, None]`, because at that location in `A` [index 1 at Python list `A == [3]`] there is nothing. [Student P9]

From Table 3 we can also see that many of the Finnish students had Python-specific issues with arrays. One such an example is P1 who did not understand the syntax for creating arrays in Python. When asked about the line: `tmp = [None] * (2 * len(self.A))` they answered:

Will that become like... it multiplies that `None` with that double length. I'm not quite sure. [Student P1]

Aside from this occurrence, most of the language-specific issues with this label belong to students who incorrectly assumed that the `range` statement in the `for` loop, used for iterating through the array, would yield one extra iteration (*i.e.,* three instead of two). This is, for example, illustrated by P2 when asked about their answer to Q4b:

I found right away a mistake I made, so it does not take the first four values but two, no, three. [Student P2]

This issue was sometimes also visible in the written answers for Q4b. For example, students P5, P6, P7, and P9 had three nonzero elements in their written answers to Q4b in Table 2.

Finally, we found seven students who had fragile knowledge of *References*. This was revealed by Q7, where variables `w` and `z` refer to the same instance of the queue. The students failed to realize this, and instead believed that a copy was made:

When object `z` of class `Y` was made, then the `z` copied into the variable `w`, then an insert was made on `w`, but it will not change that `z`. [Student P7]

When `w = z`, I'm unsure whether `z` changes if `w` changes. [Student P5]

`w` will be, like, a data structure that is assigned `z`'s contents, or something like that. [Student J1]

5.3.2. *Fragilities in Prerequisite High-Level Skills*

From Table 3, we can see that having fragilities in high-level prerequisite skills were the most common issues found in the interviews.

We found evidence that 11 students had issues related to *Reasoning about constraints* in Questions 3, 4a, 4b, and 6. For example, students have difficulties in recognizing that `hi` can be less than `lo`, as illustrated by P1 when asked about their answer to Q3, and C1 when asked about their answer to Q4a:

> *`lo` should be less than `hi`* [Student P1]

> *I thought that they were linked, that `hi` was always going to be a higher position* [Student C1]

Another example from Q3 is P3, who concluded that `hi` equals `N` is an invariant since `rebuild` assigns `hi` to `N`:

> *I have looked at that point that always if one adds something there it will do that `rebuild` [...] In `rebuild` it sets `self.hi` to `self.N`.* [Student P3]

The question with most evidence of fragility in reasoning about constraints was Q6. The excerpts show that students incorrectly conclude that the inserted elements need to be consecutive:

> *The only thing I know is that 1, 2 and 3 should be after each other.* [Student C3]

> *[...] and the three were after each other, then we should have a sequence of 1, 2, 3.* [Student J1]

> *there couldn't be empty slots between the elements.* [Student P5]

> *So I arrived at the conclusion that it should be possible to have empty slots in the middle, but not in the beginning.* [Student C6]

Another property that was problematic in Q6 was the length of the array `A`. In particular, some students believed that the number of elements in the queue had to coincide with the variable `A_length`:

> *Its size should be dependent on the `A_length` variable.* [Student C6]

Others failed to consider that the length had to be a power of two due to the fact that `rebuild` always doubles the size of the array:

> *So the size... These three must follow each other, regardless of size, as long as the size is larger than three values.* [Student C1]

We found evidence that 11 students had fragile knowledge and skill in *Program comprehension*. As described above, excerpts in this category indicate that students did not conduct a careful analysis of the code necessary to understand its behavior, but drew incorrect conclusions based on some amount of guessing. The following excerpts illustrate these kinds of statements, which all led to some kind of incorrect conclusion about the data structure:

> *I didn't feel it was relevant [...] It looked like a stack* [Student C6]
>
> *I answered to this by quite a gut feeling* [Student P5]
>
> *I was very uncertain here. [...] So I think I guessed if it was that [...]* [Student C4]
>
> *I thought it may [...] I might not understand exactly what the code does in detail but I kind of know [...]* [Student J1]

Furthermore, for the cases marked with a dagger (†) in Table 3, we have evidence that the observations are based on surface level features of the code, such as names of functions and variables. In this case, the names `insert` and `remove` lead to correct conclusions regarding the functionality of the respective functions, as illustrated by J1:

> *We have an* `insert` *function and a* `remove` *function. And it stands to reason that they should insert and remove.* [Student J1]

However, the conclusions drawn from the name `rebuild` were often not correct:

> *I thought it may be a priority queue and that's why* `rebuild` *is there.* [Student J1]
>
> *I think that the* `rebuild` *function changes the structure according to when some new element is added, so that it changes the order according to that priority.* [Student P3]

Similarly, the presence of `None` (in the Python version) was incorrectly seen as indicating leaves in a tree structure:

> *[...] a data structure which is built in a tree-like way and because there were those* `None` *values, I thought that they are those kinds of branches.* [Student P6]

As for tracing and metatracing, some students mentioned in the interview that they did trace (or tried to trace) the code. In particular this occur for questions Q4b, which asks the state of variables after the execution of `rebuild`. It was difficult to identify clearly when students have difficulties in tracing or meta-tracing. When scaffolded to trace the code closely during the interviews, all students were able to simulate the step-by-step execution of code while keeping track of the state of the computation. When errors occurred, they were due to slips, or to specific misconceptions, e.g, relating to the Python `range`.

Finally, there was one inconclusive segment, P1's interview answer to Q3. It was excluded from Table 3, because we could label it both as Reasoning About Constraints and Program Comprehension, which would have violated the mutual exclusiveness criteria:

> *I kind of deduced that lo would be the smallest, or first element in the list, for example, zero or whatever it usually is; and then ..., or the first location where there is not* `None`*. So, for example, if there is* `None`*, if the list is* `None, 1, 2, None`*, then the first would be zero, then* `lo` *would be 1 and* `hi` *would then be 3. So* `hi` *depicts the the first None in the list, its location in there, and* `lo` *depicts the first actual element.* [Student P1]

To illustrate qualitative content analysis in detail, we have provided a further example of segmenting and labeling an interview excerpt. Table 4 shows a piece of dialogue: segments S0, S1, and S2 are adjacent in the interview transcript. The interviewer asks student P6's reasoning behind their answer to Q1. The interviewer's question deliberately asks to elaborate the difference between the correct answer (queue) and the answer option that the student selected (priority queue). Segments S1 and S2 form together the student's entire answer to the interviewer's question.

P6's interview answer in Table 4 is an example of interview data which has multiple student's claims open to interpretations. In S1, the student implies that a priority queue is a tree, although it is actually an abstract data type. Based on this observation, we have labeled it as a fragility in *Data structure knowledge*. An alternative interpretation is that the student tries to explain an implementation of a priority queue, a binary heap, which is indeed based on a binary tree. This interpretation would give the possibility to code S1 as an *ADT versus implementation* issue. However, it is equally possible that by "tree-like way", the student means any tree structure they have encountered on their course. Therefore the alternative interpretation is weaker.

To illustrate how a single answer to a single question can be labelled with multiple labels, consider student P6's answer in Table 4. Segment S2 shows the student's further confusion. One could speculate that by tree branches and `None` values, the student

Table 4

An example of segment splitting in qualitative content analysis

Segment		Label
Context: Student P6's written answer to Q1 is "priority queue"		
S0	Interviewer: "What do you think is the difference between a queue and a priority queue?"	
S1	Student: "A priority queue is a data structure which is built in a tree-like way ..."	DS knowl.
S2	"...and because there were those None values, I thought that they are those kinds of branches. When I thought about how it could be a queue, somehow those None values made me think about the queue, that can it be a queue."	Program compr.

had meant a binary tree with some children set to `None` (similar to null pointers in C++). However, the evidence here is too scarce for us to pinpoint a confusion between certain data structures. The stronger evidence we see here is more implicit. When they try to recognise the data structure, they are relying on surface features of the program code, that is, the `None` values. It is likely that when answering the first Instrument question, the student has not traced the code at all. Therefore, S2 is labeled as *Program comprehension*. As such, using this approach we never had a case where we wished to assign more than one label to a single segment.

5.3.3. *Fragile Knowledge of the Advanced Topics*

From Table 3, we can see that the first seven questions also revealed fragile knowledge of the advanced topics assessed by the Instrument, that is data structures and algorithms.

We found evidence of fragilities with data structure knowledge, and we thus used the label DS Knowledge to mark excerpts that show lack of factual knowledge of data structures. As can be seen in Table 3, the evidence for this was only found in answers to Question 1. This evidence was mostly regarding the union-find data structure. For example, J1 did not know what union-find was, while C3 believed it to be an algorithm instead of a data structure:

> *About union-find, I was like, no. I haven't even encountered that one.* [Student J1]

> *I can see right away that it can't be union-find, since that is an algorithm.* [Student C3]

A number of students also confused the common illustration of a data structure and its typical implementation. For example, union-find is often illustrated as a graph where nodes have a single link to a parent node, but is implemented as an array:

> *[...] and I don't know, union find [...] I think union find has something to do with graphs, so I didn't feel it was relevant.* [Student C6]

Similarly, a priority queue is often implemented as a heap, but it has other implementations as well. Furthermore, heaps are typically illustrated as trees, but they are typically stored in an array as the heap property allows that efficiently. This confused C3 and P6 in different ways:

> *Then we have a priority queue, that is, I think it's the same thing as a heap. Since it has nodes and pointers and such. So it's either a stack or a queue.* [Student C3]

> *A priority queue is a data structure that is built in kind of tree-like way, so that when there was those `None` values, I though that they are like branches.* [Student P6]

We found seven students who showed confusion with *ADT vs. implementation*. For example, in Q2, some students confused the length of the array with the length of the data structure (queue):

> *Return* `A.length`*, right. I thought that since, well, I thought of* `A` *as the data structure itself.* [Student J1]

> *I think the command takes the length of* `A`*, that is, how many elements there are in the list.* [Student P9]

When asked, P9 also revealed that they were aware that such an answer would include the `None` elements in the underlying Python list. Another such example is P6 who was unsure what elements are supposed to be included in the data structure in the interface:

> *I was not sure whether all elements here are wanted, that could it only be those values given by the user. I was not sure which one it was meant.* [Student P6]

A similar confusion was also visible in Q4a, where the students were asked if a particular state is possible:

> `A` *holds four elements, but* `N` *only holds the value two, and I thought that should not be possible.* [Student C2]

5.4. *Programming Language Specific Issues*

We also detected three language-specific issues in the interviews. The issue with the `range` keyword in Python was already described above, as it is closely related to arrays. The remaining issues are described here, as they are not entirely in line with the research objective of the Instrument, and we do not have much conclusive evidence for them.

The first such example is a student who incorrectly believed that the expression `[None] * 8` in Python would create 8 separate lists (P3). This is not something that was covered in detail earlier the data structures and algorithms course. Similarly, some students (*e.g.,* P4) did not understand what public and private functions mean in the context of an object. Again, this is at least partially due to not covering these topics in detail in prior courses, and since Python does not differentiate between public and private functions apart from naming conventions.

A similar issue emerged for students taking the C++ version of the Instrument. In this particular case, student C5 expressed that they did not know the meaning of the statement `delete []tmp`:

> *And then that* `delete` *operation came, that I did not entirely... understand what I should do. [...]* [Student C5]

Again, it is likely that the student has not seen that particular syntax for a `delete` operation before, as modern C++ recommends working with higher level abstractions rather than arrays in the style of the C programming language.

6. Discussion

In this study, a theoretically designed Instrument (described in Section 3) is evaluated empirically for the first time. The research questions in Section 1 reflect the following hypotheses. First, if a student answers incorrectly to the Instrument questions related to data structures and algorithms, they have fragile knowledge or skills relating to prerequisites and/or to data structures and algorithms topics. Second, the questions in the Instrument can detect particular fragilities based on a student's answer. To test the hypotheses, we used qualitative content analysis introduced by Schreier (2014).

In this section we discuss the results presented in Section 5. First, we consider each Instrument's question in lights of Research Question 1: "What fragile knowledge and skills is the Instrument able to detect?" and discuss how this compares to the prior analysis by Nelson *et al.* (2020). In addition, we give suggestions about what changes are advisable to further improve the ability of individual questions to detect fragile knowledge of prerequisites. In Section 6.2 we address Research Question 2: "To what extent is the Instrument able to differentiate between fragilities, both between prerequisites and advanced topics, and among specific prerequisites?" Finally, in Section 6.3 we discuss threats to validity, and the extent to which the results presented in this paper apply.

6.1. *Analysis of Instrument Questions*

Q1–Q4a. First, some options of these multiple-choice questions worked as was expected. The most common incorrect answer in Q1 was that the class behaved like a priority queue. Interviewed students with this answer showed fragile knowledge with program comprehension. Q2 revealed either difficulties with program comprehension or confusion between an ADT and its implementation when a student answers (b) there. This was confirmed in four interviews. Four interviewed students answered Q3 incorrectly, and three of them had fragile skills when reasoning about constraints. Similarly, an incorrect answer to Q4a reveals mostly fragile skills in reasoning about constraints. These findings were consistent with the design intention (Nelson *et al.*, 2020).

Second, Q1–Q4a had unreliability with the distractor options. Table 2 shows that some answer options were rarely or never selected. In addition, the interviews indicate that if a student answers Q1 correctly, they still might have difficulties with program comprehension or fragile data structure knowledge. The correct answer in Q3 seems too easily identifiable as correct. A correct answer to Q4a may hide fragilities that a student has, because not all the invariants are needed to be identified to answer the question correctly.

We recommend altering these questions as follows. Q1 would have a free text answer instead of multiple choices. Q2 and Q3 would ask selecting all options that apply. Furthermore, convert option `lo = hi` to `lo <= hi` in Q3, as some students excluded the original options due to equality, and option `hi < N` to `hi < A.length`, as that is less obvious. Omitting the contents of `A` in Q4a might steer the students to think about the circular distance.

Q4b: Trace rebuild. The interviews revealed fragile knowledge of *operators*, *arrays*, and in *program comprehension* as expected in Nelson *et al.* (2020), as our *program comprehension* label includes program tracing.

We discovered the following patterns between written answers and identified weaknesses. In Table 2, P6, P7 and P9 have copied three array elements due to an off-by-one error with Python-specific `range` syntax, coded as *array iteration*. P4 and P9 have difficulties with operators, typically the modulo operator, resulting in incorrect order of elements. In contrast, P5, C2, and C5 have similar answers, but did not show fragile knowledge of operators in particular. It is possible that other issues masked this.

P1 is an outlier, as they answered correctly to Q4b, but incorrectly to Q4a. They identified an incorrect invariant `lo < hi` in Q3. As mentioned in Section 5.3.2, it is unclear whether this was caused specifically by a weakness in identifying program constraints or more generally program comprehension, therefore P1 has no label for Q3 in Table 3. The student only provided the end state in Q4b, leaving the details of their reasoning unknown. Unfortunately, the interviewer did not notify the student of the inconsistency between their answers to Q3, Q4a, and Q4b. One explanation for the inconsistency is that the student answered the Instrument questions in numerical order, also meaning Q4a before Q4ab. In Q4b, they began to apply detailed program tracing first time. However, tracing only the rebuild function does not contradict with the assumed `lo < hi` invariant. This lack of detail in the answers is a property of the instrument: if the Instrument had requested a detailed answer to Q4b, that might itself give a hint to apply program tracing, masking the fragility in metatracing.

Finally, some students found it confusing to trace `rebuild` from a state they had found to be inconsistent in Q4a. Some of them tried to fix the state before tracing. Others either could not, or did not trace the code closely enough, while others had issues understanding the modulo operator. As such, we recommend providing a different state for Q4b than the one used in Q4a, which students have already deemed to be invalid.

Q5: Trace insert + remove. Only three of 18 interviewed students answered incorrectly that the array `A` would have size of 1 and contain the value 3. Of those, two students showed fragile skills in with *program comprehension*. J1 relied on the surface features of the code, assuming the behavior of the data structure on the names `insert` and `remove`. C5 was unsure what the code does, but managed to trace it correctly in the interview. P3 was not asked about Q5 due to time constraints. These cases match Nelson *et al.* (2020): students either trace the code in detail or at a high level. We suggest keeping Q5 as is, because it explicitly assesses when the array is rebuilt.

Q6: Circularity. Students struggled with three aspects of Q6 in the interview. Three students, C1, P7, and P8, included the options having an array length of five, indicating a failure to identify the constraint that the size of the array is a power of two. Meanwhile, excluding (c) `[2, 3, -, 1]` and (d) `[2, 3, -, -, 1]` likely indicates not having understood the circularity: interviews of C3, P4, and P5 revealed a belief that the last three elements should be located consecutively in the array. Finally, some students struggled with how to approach the problem, because the unknown sequence could not be traced.

Q6 does not distinguish between the constraint cases $2k$ versus 2^k for the length of the array. We recommend adding an option with an array with length of 6. Also, as this question was relatively difficult, some students might benefit from a hint that the unknown sequence of operations only includes inserts and removes.

Q7: References. The major finding of Q7 supports the intended design that this question would assess references. Eight of eleven incorrect answers to Q7 was that `a = 1, b = 2`. All seven corresponding interviews revealed issues with references. The remaining students (3 incorrect answers, 2 interviewed) provided a nonnumerical answer, suggesting other fragilities.

However, the authors of Nelson *et al.* (2020) also hypothesized that the answer `a = 3, b = 2` would indicate that students were unable to distinguish between different *instances* of the data structure. We found no such answers in our data set, which could simply mean that this fragility is rare among the students that answered the Instrument. Another possibility is that most students traced the code at a high-level (indicated in some interviews), and since students did not need to trace the `insert` and `remove` operations in details, difficulties with separating different instances would not be visible.

Moreover, a student might arrive at the incorrect answer `a = 1, b = 2` with a correct understanding of references. This might happen if the student assumes that the data structure is a minimum priority queue and traces `insert` and `remove` at a high level. It might be useful to add an additional element to `y` in order to detect if this is the case, as a similar issue would then be visible in the value of b.

6.2. *Differentiated Assessment*

Our findings suggest that the questions in the Instrument are indeed useful to differentiate fragile prerequisite knowledge and skills from fragilities in advanced topics. The incorrect answers to questions Q1 and Q3–Q7 were often caused by fragile prerequisite knowledge, as can be seen in Table 3. Although the Instrument is able to distinguish between prerequisites and advanced topics, it cannot always differentiate among the *specific* fragilities in prerequisites. For instance, Nelson *et al.* (2020) introduce the label Basic Notational Machine that separate skills such as *Array iteration* and *Arrays* (declaring and indexing arrays). We were not able to differentiate these from each other. Instead, the categories were merged to form a single fragility (Array).

In comparison, fragilities related to the advanced topic (data structures and algorithms) were found to directly contribute to errors only for questions Q1 and Q2 (and, in a single case, for Q3, Q4a, and Q6 each). In particular, students who struggled to distinguish between abstract data types and their implementation failed either Q2 or Q3, therefore these two questions together have the potential to highlight fragilities with data structures and algorithms.

We found evidence that Q4b indicate fragile knowledge of operators (modulo in particular), and two Python-specific issues: array iteration (off by one with `range`) and creating an array with a predetermined number of empty cells. Nelson *et al.* (2020) added Q7 (originally numbered in the paper as Question 6) to detect specific issues with references. Our results support the claim that a particular incorrect answer to this question does indeed reveal fragilities with values and references.

Fragile skills in reasoning about constraints were observed in Q3, Q4a and Q6. This is not unexpected, since these are the three questions that, more evidently, address the understanding of circularity and the rebuilding policy.

What is most evident from Table 3, however, is that due to fragile skills, 16 students struggled with either *program comprehension*, *reasoning about constraints*, or both. It can be argued that these high-level skills are not typically taught in an introductory programming course explicitly, as part of the syllabus, but these are still expected to be learned (Izu *et al.*, 2019). These prerequisites were those we qualified as being *high-level* (Section 5.3.2), and were indeed grouped as such by Nelson *et al.* (2020), even though they were still considered to be prerequisites. Regardless of this, our findings show that they have a different role compared to the knowledge of basic programming notions such as control flow constructions, operators, or arrays.

In this way, the aforementioned high-level skills are similar to the findings of Fisler *et al.* (2017) regarding scope, aliasing and mutation. The authors observed that these concepts are expected to be taught in an introductory course, but yet students struggle with them throughout their education. The authors hypothesize that this is because they are not taught explicitly enough in the introductory course, and after that students are expected to learn the details themselves. As such, these high-level skills are not considered to be advanced topics covered by the Instrument. Therefore, they might most appropriately be seen as *middle-ground* skills, located between introductory programming and advanced skills, as they represent skills that can not be expected to be fully developed after an introductory programming course.

As discussed in Section 2.3, it is worth connecting these middle-ground skills to abstraction skills, which are notoriously difficult to teach and assess (Mirolo *et al.*, 2021). In particular, they align well with the higher levels of the hierarchy proposed by Perrenet *et al.* (2005) and the operational dimensions of Statter and Armoni (2020), since they require understanding the code at a higher level, as understanding what the program does for some particular input is not enough.

This ability to switch between different levels of abstraction (reasoning on high-level behaviors *vs.* tracing the code in detail) was also visible for students who struggled to differentiate between the ADT and its implementation. Abstraction has a significant role also in relation with another high-level skill that was included by Nelson *et al.* (2020) among the prerequisites skills, namely meta-tracing. See, *e.g.,* Statter and Armoni (2020) about the importance of knowing *when* it is feasible to use the high-level reasoning, and when it is necessary to trace the code closely. From both the written answers and the interviews, it is difficult to say whether the students were actually able to trace the code autonomously when answering the questions, and whether or not they even tried to trace it. The interviews did not help us understand which meta-tracing activities students conducted: when, and on which inputs, they decided to trace portions of code in order to understand its behavior. Nevertheless, the fact that students were often found to guess what this behavior would be, lead us to hypothesize that they either failed to recognize the need to trace the code closely, or did not have the skills to do it. That is, even if we did not find ultimate evidence for this, the impression is that there are fragile meta-tracing skills besides program comprehension issues and the ability to reason about constraints.

As has been shown in this paper, the Instrument seems to be able to reveal fragile high-level skills as well as highlight some particular more fundamental fragilities (unless they are hidden by lack of high-level skills). Overall, our results emphasize the importance of the high-level skills, and validate some of the hypotheses in Nelson *et al.* (2020). We suggest modifications for the questionnaire, which could be then used as a part of preliminary test for a data structures and algorithms course. However, the Instrument requires revisions and more empirical research to ensure its reliability. In addition, we suggest that the skills labelled as *high-level skills* should be considered to be *middle-ground* skills rather than prerequisites, and that the creators of assessments should distinguish them to better account for their importance.

6.3. *Limitations*

The sample size – 28 written answers and 18 interviews – was adequate for qualitative study. The limited number of subjects, however, have implications on what type of conclusions can be drawn from the results. While the data shows fragile knowledge and skills, it is not possible to interpret the absence of fragility for some question as a proof that the particular question do not assess that fragility. A qualitative study design should include a sampling strategy which evolves over time. In contrast to random sampling used in quantitative research, qualitative research may aim collecting information-rich samples which vary from each other, thus vastly representing the phenomenon to be studied (Rapley, 2014). Unfortunately, the study design did not include a choice of a sampling strategy. Due to the small and self-selected group of students that were interviewed, it is possible that a student with some particular fragility was not a part of the group of interviewees.

The study had differences in the ways of transcription for the two local interview languages (intelligent verbatim vs. selected quotes and descriptions). Verbatim transcripts are not always necessary, if one can label a time segment in a digital recording that can be easily retrieved (Barbour, 2014). Indeed, due to timestamps, it was possible to review certain segments of our interview videos and produce as verbatim transcripts translated in English. A greater concern is likely a case where the original analyzer of the interview produced too short a segment transcribed in English. We experienced the latter in the *Presenting and interpreting the findings* phase of the analysis with the whole research team (see Section 4.2), but if one researcher decided they need more context to verify a segment, another researcher provided a longer transcript or description for them from the original video recording.

Certain fragilities (*e.g.,* program comprehension) seem to hide more fundamental ones (*e.g.,* knowledge of array indexing) during the interviews. For example, it is difficult to reliably assess whether a student who is unable to trace programs also has fragile knowledge of the modulo operator or not. The small and self-selected data set could therefore be the reason why some expected fragilities were not found. See, for instance, the failure to differentiate between different instances in Q7. Finally, the small sample size might hide some infrequent patterns. A larger data set could give more insights on how to interpret the results for Q4b.

Some of the fragilities were detected during the interviews, although the student answered correctly to the question. This was partially found to be due to inadequate distractors. Some students might have guessed the answer, which is typical in multiple-choice questions. The small data set limits our ability to evaluate the extent of this type of issues.

Although the fact that the data set includes students from multiple countries and involve multiple programming languages is a strength of the data set, this does introduce potential sources of errors. In particular, the different spoken languages means that students from different countries were interviewed by different interviewers. Furthermore, different spoken and programming languages may highlight the same concept in different lights, and for spoken languages, some of this nuance might be lost when translated into English. In spite of these difficulties we found many similarities between the different countries, which suggests that these potential problems were at least not major enough to mitigate our conclusions, but the results are generalizable at least to institutions with similar academic culture. As previously mentioned, the results show that some of the fragilities are specific to a particular programming language. This means that, while many of the findings here are independent of the context, care needs to be taken when translating the code in the Instrument into different programming languages, as new types of problems may appear.

7. Conclusion

7.1. *Program Comprehension and Reasoning about Constraints as Key Middle-ground Skills*

The empirical evaluation supports many, but not all of the claims by Nelson *et al.* (2020) on the Instrument's capability of differentiating between skills related to prerequisite and advanced topics. We suggested ways to improve the Instrument's questions, both by improving the distractors of the multiple-choice questions, and by slightly changing the content or phrasing of the questions. Based on our data, we argue that these improvements will increase the Instrument's effectiveness in assessing prerequisites as a whole, but also to pinpoint specific fragilities.

Our data shows that the skills previously labelled as *high-level* skills (such as *program comprehension* and *reasoning about constraints*) seem to be the cause for many students answering incorrectly to many of the questions in the Instrument: these skills seem to bear a major responsibility in hindering the proficiency in more advanced skills. Therefore we suggested adding the category of *middle-ground* skills to the framework of Nelson *et al.* (2020), in order to highlight their special role. In fact these skills are often considered to be prerequisites in latter courses and are not usually taught explicitly in that context. However, our data shows that students still struggle with them in courses on data structures and algorithms. As such, the situation for these skills is similar to what Fisler *et al.* (2017) observed for concepts such as scope, mutation and aliasing. Latter courses expect students to have familiarity with them, but the introductory courses do not cover them explicitly enough. Our results highlight the importance of teaching these middle-ground skills explicitly at an early stage in the education, and to continue teaching them throughout the education to allow students to master them, as it is difficult to fully develop them only in one introductory course. Exercises that ask to analyze a portion of code and reflect on the behaviour that it determines, such as the ones proposed by the Instrument, can be fruitfully used as learning tools or to provide formative assessments for these middle-ground skills.

7.2. *Future Work*

Some of the fragile prerequisite skills that the Instrument was designed to identify, such as tracing and meta-tracing, in this report were folded under the label Program comprehension. Part of the reason for this is the difficulty of identifying precisely what specific fragility a student has, even when employing semi-structured interviews. Further studies are needed to address this finer level of granularity. A possible approach could be to conduct a think-aloud study to follow students' lines of reasoning, or to observe students working in pairs on the Instrument's questions, by analysing students' strategies and conversations during the problem solving process.

As already discussed, what we identified as middle-ground skills are in fact crucial to progress further, but it may be unreasonable to expect wide proficiency with these skills after introductory programming courses. However, how this should be dealt with within the constraints of introductory and advanced courses needs further research.

Given the Instrument's ability to differentiate between certain prerequisites, middle-ground skills, and advanced skills, it would be interesting to explore integrating the Instrument into a learning management system. This would allow the learning management system to tell the students not only if they passed or failed, but direct students towards the most effective remediation. For example, if a student is identified as having an issue with basic prerequisites about array indexing, they could be asked to review relevant fundamentals or pointed to follow-up questions dealing with this issue.

Finally, in this work we focused on one of the questions proposed by Nelson *et al.* (2020), but the report also discussed questions related to other advanced course topics. More work would be needed to generalize our results to other contexts, however we hypothesize that middle-ground skills play again a major role, whenever students are asked to answers questions and reflect about a given portion of code and the behaviour that it determines.

References

Adams, W.C. (2015). *Conducting Semi-Structured Interviews*. John Wiley & Sons, Inc., New Jersey, USA, pp. 492–505. Chap. 19. 9781119171386. `https://doi.org/10.1002/9781119171386.ch19`

Aharoni, D. (2000). Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures. In: *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '00. Association for Computing Machinery, New York, NY, USA, pp. 26–30. 1581132131. `https://doi.org/10.1145/330908.331804`

Anderson, L.W., Krathwohl, D.R., Airasian, P.W. (2001). *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Longman, New York, NY, USA. 0321084055.

Barbour, R.S. (2014). *Quality of Data Analysis*. SAGE Publications, London. Chap. 34. 978-1-4462-0898-4. `https://doi.org/10.4135/9781446282243`

Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R., Murphy, L. (2014). 'Explain in Plain English' Questions Revisited: Data Structures Problems. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. ACM, New York, NY, USA, pp. 591–596. 978-1-4503-2605-6. `https://doi.org/10.1145/2538862.2538911`

Danielsiek, H., Paul, W., Vahrenhold, J. (2012). Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. ACM, New York, NY, USA, pp. 21–26. 978-1-4503-1098-7. `https://doi.org/10.1145/2157136.2157148`

Fisler, K., Krishnamurthi, S., Tunnell Wilson, P. (2017). Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '17. Association for Computing Machinery, New York, NY, USA, pp. 213–218. 9781450346986. `https://doi.org/10.1145/3017680.3017777`

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M.C., Zilles, C. (2008). Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process. *SIGCSE Bull.*, 40(1), 256–260. `https://doi.org/10.1145/1352322.1352226`

Hartmanis, J. (1994). Turing Award Lecture on Computational Complexity and the Nature of Computer Science. *Commun. ACM*, 37(10), 37–43. `https://doi.org/10.1145/194313.214781`

Hazzan, O. (2008). Reflections on Teaching Abstraction and Other Soft Ideas. *SIGCSE Bull.*, 40(2), 40–43. `https://doi.org/10.1145/1383602.1383631`

Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C., Weeda, R. (2019). Fostering Program Comprehension in Novice Programmers -Learning Activities and Learning Trajectories. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '19. Association for Computing Machinery, New York, NY, USA, pp. 27–52. 9781450375672. `https://doi.org/10.1145/3344429.3372501`

Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society (2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA. 9781450323093.

Kane, M.T., Bejar, I.I. (2014). Cognitive frameworks for assessment, teaching, and learning: A validity perspective. *Psicologia Educativa*, 20(2), 117–123. `https://doi.org/10.1016/j.pse.2014.11.006`

Lister, R., Simon, B., Thompson, E., Whalley, J.L., Prasad, C. (2006a). Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITICSE '06. Association for Computing Machinery, New York, NY, USA, pp. 118–122. 1595930558. `https://doi.org/10.1145/1140124`

Lister, R., Simon, B., Thompson, E., Whalley, J.L., Prasad, C. (2006b). Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. *SIGCSE Bull.*, 38(3), 118–122. `https://doi.org/10.1145/1140123.1140157`

Luxton-Reilly, A., Becker, B.A., Cao, Y., McDermott, R., Mirolo, C., Mühling, A., Petersen, A., Sanders, K., Simon, Whalley, J. (2018). Developing Assessments to Determine Mastery of Programming Fundamentals. In: *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. ITiCSE-WGR '17. Association for Computing Machinery, New York, NY, USA, pp. 47–69. 9781450356275. `https://doi.org/10.1145/3174781.3174784`

Margulieux, L., Denny, P., Cunningham, K., Deutsch, M., Shapiro, B.R. (2021). When Wrong is Right: The Instructional Power of Multiple Conceptions. In: *Proceedings of the 17th ACM Conference on International Computing Education Research*. ICER 2021. Association for Computing Machinery, New York, NY, USA, pp. 184–197. 9781450383264. https://doi.org/10.1145/3446871.3469750

Mirolo, C., Izu, C., Lonati, V., Scapin, E. (2021). Abstraction in Computer Science Education: An Overview. *Informatics in Education*, 20(4), 615–639. https://doi.org/10.15388/infedu.2021.27

Nelson, G.L., Strömbäck, F., Korhonen, A., Begum, M., Blamey, B., Jin, K.H., Lonati, V., MacKellar, B., Monga, M. (2020). Differentiated Assessments for Advanced Courses That Reveal Issues with Prerequisite Skills: A Design Investigation. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '20. Association for Computing Machinery, New York, NY, USA, pp. 75–129. 9781450382939. https://doi.org/10.1145/3437800.3439204

Perkins, D., Martin, F. (1985). Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22. Technical Report IR85-22, Educational Technology Center, Cambridge, MA, USA. https://eric.ed.gov/?id=ED295618

Perrenet, J., Groote, J.F., Kaasenbrood, E. (2005). Exploring Students' Understanding of the Concept of Algorithm: Levels of Abstraction. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '05. Association for Computing Machinery, New York, NY, USA, pp. 64–68. 1595930248. https://doi.org/10.1145/1067445.1067467

Porter, L., Zingaro, D., Liao, S.N., Taylor, C., Webb, K.C., Lee, C., Clancy, M. (2019). BDSI: A Validated Concept Inventory for Basic Data Structures. In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER '19. Association for Computing Machinery, New York, NY, USA, pp. 111–119. 9781450361859. https://doi.org/10.1145/3291279.3339404

Qian, Y., Lehman, J. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.*, 18(1). https://doi.org/10.1145/3077618

Rapley, T. (2014). 4. *Sampling Strategies in Qualitative Research*. SAGE Publications, London. 978-1-4462-0898-4. https://doi.org/10.4135/9781446282243

Schreier, M. (2014). *Qualitative Content Analysis*. SAGE Publications, London. Chap. 12. 978-1-4462-0898-4. https://doi.org/10.4135/9781446282243

Shaffer, C.A., Karavirta, V., Korhonen, A., Naps, T.L. (2011). OpenDSA: Beginning a Community active-eBook Project. In: *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. Koli Calling '11. ACM, New York, NY, USA, pp. 112–117. 978-1-4503-1052-9. https://doi.org/10.1145/2094131.2094154

Sorva, J. (2013). Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.*, 13(2). https://doi.org/10.1145/2483710.2483713

Statter, D., Armoni, M. (2020). Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Trans. Comput. Educ.*, 20(1). https://doi.org/10.1145/3372143

Tenenberg, J., Murphy, L. (2005). Knowing what I know: An investigation of undergraduate knowledge and self-knowledge of data structures. *Computer Science Education*, 15(4), 297–315. https://doi.org/10.1080/08993400500307677

Valstar, S., Griswold, W.G., Porter, L. (2019). The Relationship between Prerequisite Proficiency and Student Performance in an Upper-Division Computing Course. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Association for Computing Machinery, New York, NY, USA, pp. 794–800. 9781450358903. https://doi.org/10.1145/3287324.3287419

Wing, J.M. (2006). Computational Thinking. *Commun. ACM*, 49(3), 33–35. https://doi.org/10.1145/1118178.1118215

Zingaro, D., Taylor, C., Porter, L., Clancy, M., Lee, C., Nam Liao, S., Webb, K.C. (2018). Identifying Student Difficulties with Basic Data Structures. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER '18. ACM, New York, NY, USA, pp. 169–177. 978-1-4503-5628-2. https://doi.org/10.1145/3230977.3231005

M. Begum Dr Marjahan Begum is a learning scientist in computing education interested in programming pedagogy, algorithmic thinking and software engineering education. Recently she has embarked on "women in computing" research specifically looking at barriers faced by women in the field. She is a Lecturer in the Department of Computer Science, University of Nottingham (0.8) and at the City University of London (0.2).

P. Haglund is a lecturer and PhD student in the Department of Computer Science at Linköping University, Sweden. He has taught at the university since 2018, primarily working with courses in programming and language design. His interest in research is focused around students' acquisition of subtle programming concepts. He has also worked with the Swedish National Agency for Education, developing courses for the national initiative to introduce younger students to programming. His efforts here have primarily been focused on teaching programming to K-12 teachers.

A. Korhonen is a Senior University Lecturer in Aalto University. He is also an Adjunct Professor at University of Turku. Korhonen has been working in the field of educational technology since 1997. He has been developing several systems capable of automatic assessment and feedback as well as visualizing and engaging students to digital learning environments. He is one of the founders of the LeTech (Learning + Technology) research group at Aalto University (`https://research.cs.aalto.fi/LeTech/`). Korhonen has been a PC member, chairing, and organizing conferences like ACM Conference on International Computing Education Research (ICER), Frontiers in Education (FIE), and Koli Calling International Conference on Computing Education Research.

V. Lonati is an assistant professor at the Computer Science Department of Università degli Studi di Milano (Italy). With a degree in mathematics and a PhD in computer science, she has been working in the field of computer science education since 2008. She is one of the founder of the ALaDDIn research group at University of Milan (`http://aladdin.unimi.it`). Her current research interests include introductory programming learning, computing education at K-12 level, constructivist strategies in computing education, professional development for teachers.

M. Monga is an Associate Professor at Università degli Studi di Milano, Milan, Italy with the Department of Computer Science. He holds a Ph.D. in Computer and Automation Engineering from Politecnico di Milano, Italy. He is one of the founders of ALaDDIn (`http://aladdin.unimi.it`) and the Deputy Director of the CINI National Laboratory on Computer Science and School. Further information are available on his web page at `http://homes.di.unimi.it/~monga`.

F. Strömbäck is a lecturer in the Department of Computer Science at Linköping University, Sweden. He has been working in the area of teaching and learning concurrent programming, and as a part of that work developed the visualization tool Progvis. In addition to concurrent programming, his research interests include teaching subtle programming concepts to novices, and programming language design and implementation.

A. Tilanterä is a Ph.D. student at Aalto University, Finland. They received B.Sc. and M.Sc. degrees in Computer Science from the same institution in 2016 and 2021, respectively. They are currently studying students' misconceptions related to Data Structures and Algorithms, while having research interests also in Algorithm Visualization, Automated Feedback, and Information Visualization.

Appendix A. The Instrument

Consider the following code when answering the questions below:

```java
public class Y<Key extends Comparable<Key>>
{
    private Key[] A = (Key[]) new Comparable[1];
    private int lo, hi, N;
    public void insert(Key in)
    {
        A[hi] = in;
        hi = hi + 1;
        if (hi == A.length) hi = 0;
        N = N + 1;
        if (N == A.length) rebuild();
    }
    public Key remove() // assumes this is not empty
    {
        Key out = A[lo];
        A[lo] = null;
        lo = lo + 1;
        if (lo == A.length) lo = 0;
        N = N - 1;
        return out;
    }
    private void rebuild()
    {
        // The line below is essentially:
        // Key[] tmp = new Key[2*A.length]
        // with keys being comparable.
        Key[] tmp = (Key[]) new Comparable[2*A.length];
        for (int i = 0; i < N; i++ )
            tmp[i] = A[(i + lo) % A.length];
        A = tmp;
        lo = 0;
        hi = N;
    }
}
```

1. Class Y behaves like which well-known data structure?
   (a) Stack
   (b) Queue
   (c) Priority queue
   (d) Union find

2. Write the body of a method `int size()` that returns the number of elements in the data structure.
   (a) `return N;`
   (b) `return A.length;`
   (c) `return A[N];`
   (d) `return hi - lo;`
3. Which invariant does the data structure maintain after every public operation? (An invariant is a condition that the data structure ensures is true after each operation)
   (a) `N < A.length`
   (b) `lo < hi`
   (c) `hi < N`
   (d) `hi == N`
4. Assume that:

   `A` holds

3	8	4	1

   `lo` holds 3
   `hi` holds 2
   `N` holds 2
   (a) Is the above situation something that can occur by calling a sequence of `insert` and `remove`? If yes, give such a sequence, otherwise explain why not.
   (b) What are the contents of `A`, `lo` and `hi` after executing `rebuild` in this state?
5. Draw the data structure (including the contents of `A` and the values of `hi`, `lo`, and `N`) after the following operations, and indicate how many times `rebuild` were called:

```
Y y = new Y();
y.insert(1);
y.remove();
y.insert(2);
y.remove();
y.insert(3);
```

6. Given the following partially known sequence of operations, what are the possible contents of `A`? Select all that apply. Empty boxes are considered empty by the data structure (i.e., they contain something that the data structure does not care about).

```
Y y = new Y();
// an unknown sequence of operations
y.insert(1);
y.insert(2);
y.insert(3);
```

(a)

	1	2	3

(b)

	1	2	3	

(c)

2	3		1

(d)

2	3			1

(e)

1	2		3

7. What are the values of `a` and `b` after executing the following piece of code?

```
Y y = new Y();
Y z = new Y();
Y w = z;
w.insert(3);
z.insert(1);
y.insert(2);
int a = z.remove();
int b = y.remove();
```

8. How many array accesses does a single call to `Y.remove` take in the worst case? (To make this well-defined, we assume that the compiler performs no clever optimisations. That is, every array access we've written in the code will actually be performed.)
   (a) $\sim 4N$
   (b) 2
   (c) $\sim 2N$
   (d) 7
9. How many array accesses does a single call to the most expensive public method of `Y` take in the worst case?
   (a) linear in k – $\Theta(k)$.
   (b) constant – $\Theta(1)$.
   (c) linearithmic in k – $\Theta(k \log k)$.
   (d) quadratic in k – $\Theta(k^2)$.
10. What is the number of array accesses per operation in the following sequence of $2k$ operations, starting from anempty data structure: `y.insert(1); y.remove(); y.insert(2); y.remove(); y.insert(3); y.remove(); ... y.insert(k); y.remove();`
    Note: The amortized case describes the average, or expected, runtime of an operation.
    (a) linear in k in the worst case and in the amortized case.
    (b) constant in the worst case.
    (c) constant in the amortized case, but linear in k in the worst case.
    (d) quadratic in k in the worst case.
11. True or false: The data structure `Y` uses space linear in `N`. Explain you answer on a separate piece of paper. (Be as formal *and short* as you can, but not shorter. If you use more than half a page of text you're on the wrong level of abstraction.)

Appendix B. The Code in C++

Below is the C++ version of the code from the Instrument for reference.

```
template <class Key>
class Y
{
public:
    void insert(Key in)
    {
        A[hi] = in;
        hi = hi + 1;
        if (hi == A_length) hi = 0;
        N = N + 1;
        if (N == A_length) rebuild();
    }
    Key remove() // assumes this is not empty
    {
        Key out = A[lo];
        A[lo] = Key{};
        lo = lo + 1;
        if (lo == A_length) lo = 0;
        N = N - 1;
        return out;
    }
private:
    Key *A{new Key[1]};
    int A_length{1};
    int lo{0}, hi{0}, N{0};
    void rebuild()
    {
        Key *tmp = new Key[2*A_length];
        for (int i = 0; i < N; i++ )
            tmp[i] = A[(i + lo) % A_length];
        delete []A;
        A_length = 2*A_length;
        A = tmp;
        lo = 0;
        hi = N;
    }
}
```

Appendix C. The Code in Python

Below is the Python version of the code from the Instrument for reference.

```python
class Y:

    def __init__(self):
        self.A = [None]
        self.lo = 0
        self.hi = 0
        self.N = 0

    def insert(self, input):
        self.A[self.hi] = input
        self.hi = self.hi + 1
        if (self.hi == len(self.A)):
            self.hi = 0
        self.N = self.N + 1
        if (self.N == len(self.A)):
            self.rebuild()

    def remove(self): # assumes self is not empty
        output = self.A[self.lo]
        self.A[self.lo] = None
        self.lo = self.lo + 1
        if (self.lo == len(self.A)):
            self.lo = 0
        self.N = self.N - 1
        return output

    def rebuild(self):
        tmp = [None] * (2 * len(self.A))
        for i in range(0, self.N):
            tmp[i] = self.A[(i + self.lo) % len(self.A)]
        self.A = tmp
        self.lo = 0
        self.hi = self.N
```

Appendix D. Overview of the Written Answers

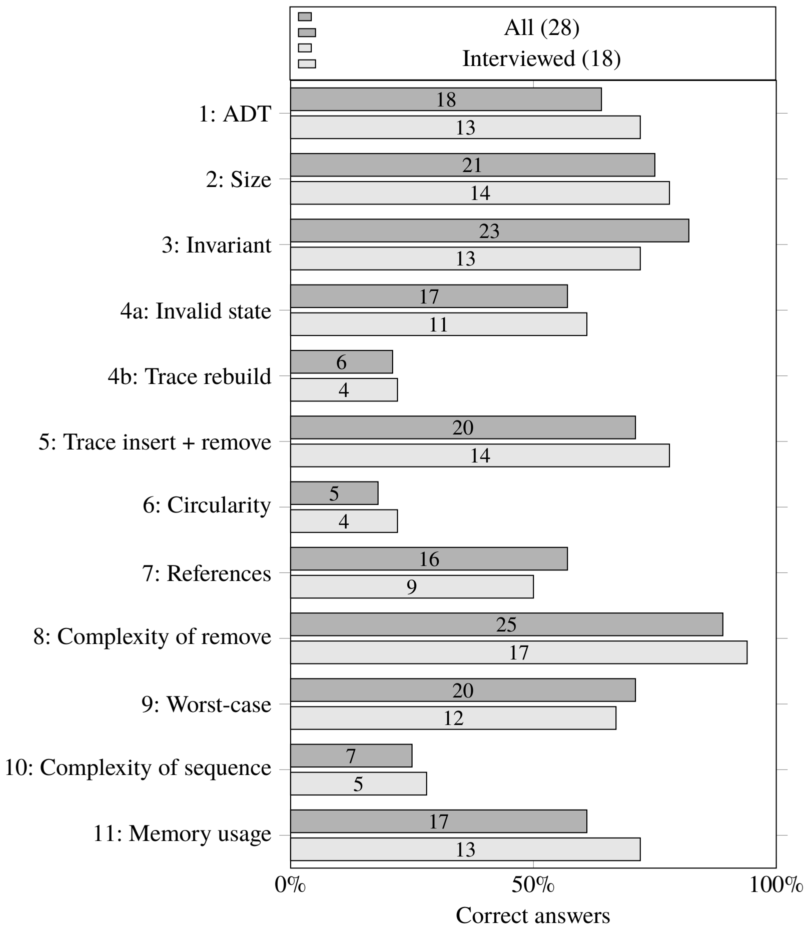

Fig. 1. Overview of the correct answers to each of the questions (1–11). The number inside each bar refers to the number of correct answers, while the length of each bar corresponds to the percentage of correct answers. Question 11 was considered correct as long as the justification for the answer was correct.