COMPREHENSION OF LINKED LIST DIAGRAMS: THE EFFECTS ON CODE WRITING SKILLS

U H Obaidellah[1], Erma Rahayu Mohd Faizal[2], Aznul Qalid Md. Sabri[3], Chiam Yin Kia[4]*

[1,2,3]Department of Artificial Intelligence
Faculty of Computer Science and Information Technology
University of Malaya
50603 Kuala Lumpur, Malaysia

[4]Department of Software Engineering
Faculty of Computer Science and Information Technology
University of Malaya
50603 Kuala Lumpur, Malaysia

E-mail: unaizah@um.edu.my[1*](corresponding author), erma@um.edu.my[2], aznulqalid@um.edu.my[3], yinkia@um.edu.my[4]

DOI: https://doi.org/10.22452/mjcs.vol34no2.5

ABSTRACT

The use of graphics tends to aid reasoning in program solving by improving novice programmers' ability to read and write code. This study extends existing work in computer programming on the use of diagrammatic representation for students undertaking the fundamental data structure course (CS2) in Malaysia. Students were tested on comprehension of diagrams followed by the composition of code with respect to the linked list topic. The data was assessed using the inter-rater agreement test and showed a high degree of consistent ratings. Results showed a moderate correlation between students' ability to analyze list operations in the form of notation and performance on code writing. Students assessed the diagrams differently according to the complexity level. The result can be generalized to conclude that the use of diagrams alone may not fully support reasoning and program solving. However, some types of diagrams are potentially more effective to support code composition and more emphasis should be given to evaluating the effectiveness of diagrams in organizing cues to facilitate novice programmers in program solving. Further investigation on a combination of activities related to comprehension of diagrams, including code reading and explanation prior to code writing, is recommended.

Keywords: Data structures, computer science education research, diagrams comprehension, code writing, inter-rater agreement

1.0 INTRODUCTION

Computer programming is increasingly becoming an important subject. This is partly contributed by the demand of technological related use to support our daily lives. Therefore, technical knowledge and skills in computer programming are vital among (producers in developing) computer systems. However, grasping the concepts and mastery of computer programming has been identified as difficult by novice learners such as students in their first year of an undergraduate computer science program (Gomes & Mendes, 2007; Lahtinen, Ala-Mutka & Jarvinen, 2005; Robinsons, Rountree & Rountree, 2003). Among the factors identified to contribute to this difficulty are lack of problem-solving skills and poor conceptual understanding (Milne & Rowe, 2002). As a result, students often fail the course due to reduced motivation. It is felt that this failure has some relationship to skills required to write functional computer codes. Therefore, there is a (definite) need for a study to determine ways to enhance learning (and developing code).

The use of visuals in learning computer programming has the potential to facilitate learners' conceptual understanding as graphics reduces the complexity of abstract ideas while motivating longer attention for learners. Different types of graphical representations (i.e., animations, flowcharts, games) have been studied to evaluate its effectiveness in promoting computing programming skills at the fundamental and advanced level of a computing course (Berry & Kolling, 2016; Hooshyar, Ahmad, Md Nasir, Shamshirband, & Horng, 2015; Jin, Jin, & Xue, 2010). Each produces different degree of assistance for learners, depending on the complexity of the concept (Segura et al., 2008; Adamchik, 2011; Stasko, Badre, & Lewis, 1993). However, limited studies have investigated the assessment on the level of comprehending concepts in composing program codes for advanced programming subjects such as data structure using diagrams. Interaction with diagrams often gives flexibility in facilitating concept understanding through sketches instead of advanced technologies like animations or videos. This reduces student reliance on practicing concept understanding using other forms of technologies yet promotes mental reflection of the concepts. This evaluation is important as results can provide suggestions on students' problem-solving skills that impact breaking down and organizing problems into sub-problems on comprehension and composition skills in computer programming.

In this paper, we investigate the level of concept comprehension of advanced programming through diagrammatic representations among novice undergraduate computer science students. Given the high failure rate among first-time takers, (Milne & Rowe, 2002; Lahtinen, Ala-Mutka & Jarvinen, 2005) this evaluation is critical to better understand their level of knowledge on advanced programming concepts via diagrammatic intervention, and, as a consequence, to developing suggestions for teaching and learning approaches for novice programmers.

Considering this motivation, the proposed study aims to investigate the effectiveness of using visualization, or more specifically static pictures, in facilitating novice programmers to identify solution strategies in programming problems. More specifically, we tested undergraduate computer science students' comprehension of diagrams (used to represent concepts) followed by the composition of codes with respect to the singly linked list topic of data structure. The linked list topic is considered appropriate to represent an advanced programming topic, as it is one of the most fundamental topics taught at the advanced programming level (e.g., courses in data structure) (Duchowski & Davis, 2007). In our proposed study, data are analyzed via inter-rater agreement analysis by adopting the Structure of Observed Learning Outcomes (SOLO) taxonomy model, which can inform students' learning performance and understanding of materials tested in increasing complexity. The SOLO taxonomy model is a systematic approach of identifying and classifying students' understanding of a subject (or topic) according to the levels of increasing complexity of the task performed (Biggs & Collis, 1982) (see Section 3.2: SOLO taxonomy for more details). This assessment describes the students' levels of understanding which develops from simple to complex as they learn topics of increasing difficulty. Therefore, using this method would produce a more accurate judgement that can be made on the quality of the student's response on the tasks tested. Hence, this approach is considered appropriate for the proposed work. Furthermore, this study contributes to a better understanding on the procedural aspects of novice programmers during problem solving of the linked list, an advanced data structure topic. The findings presented in this work is crucial for the development of improved instructional design including a better guided teaching and learning methods for both instructors and learners.

2.0. Related work

This section will give an overview of the related literature to the proposed work. We begin by discussing the common difficulties of learners in advanced programming topics such as data structure prior to outlining a summary on the use of visualization elements to assist in learning computer programming. This is followed by a discussion on the use of diagrams to facilitate reasoning before a review on the use of diagrams to support learning the linked list topic is given.

2.1 Difficulties in learning Data Structure

The data structure (DS) course is usually taught to computer science students in the second semester (or term) of their first year. This subject is critical for these students as its content helps to develop thinking and problem-solving skills, improves programming skills, teaches the implementation of APIs or libraries, and stimulates students to apply the best practices of software engineering principles (Lister et al., 2007). The DS syllabus commonly covers linear structure topics such as linked lists, stacks, queues, and fundamental non-linear topics such as binary search trees. Typically, these topics are taught one at a time in increasing difficulty. Linear data structure topics precede non-linear topics. The linked list topic is often taught before stacks, queues, and binary search trees. Data structures are known to consist of heavy abstract concepts, with highly dynamic algorithms and programs. Thus, the nature of these topics which differ from those covered in fundamental programming topics makes it challenging for novice programmers to grasp its essential principles.

The standard approach of teaching the DS topics occurs by first introducing the motivation, importance, and application of the structure, followed by the algorithms that are discussed in terms of pseudocode. This process is facilitated by visualization to simulate the behavior of these algorithms. During learning, students are encouraged to code on their own in an IDE to support and verify their understanding. Their knowledge is further assessed in tutorial and lab assignments. However, there seems to be a problem among undergraduate students in learning the data structure course (CS2). Although the concepts of a data structure are easy to describe and comprehend, showing how these structures work by writing fully functional codes does not seem to be a straightforward task for these students. A few factors contribute to this. Firstly, programming textbooks, especially those on data structures, often consist of dense textual descriptions where algorithms and pseudocode are presented in a less organized structure that would not facilitate learning for novice programmers. In other words, these representations do not consider the limited capacity of working memory to hold a limited number of concepts at a time (Caspersen & Bennedsen, 2007). Secondly, beginner programmers can be considered novices compared to more experienced programmers who have been coding consistently for at least three years. Thus, the knowledge structures of these novices are less comprehensive than expert programmers causing difficulties in comprehending advanced concepts and unfamiliar algorithms.

Our observation on our students' coding performances suggests that they encounter problems writing fully functional data structure programs. If successful, their solutions or program codes often closely resemble those found in the lecture notes and textbook. A common approach novice programmers' practice to deal with this problem is by memorizing program code so as to reproduce them in assessments by adopting minimal or no revision from the original code learned from textbooks or lecture notes. This observation is consistent with the kind of learning strategies demonstrated by novice programming learners showing a superficial learning approach compared to those of experienced programmers (Bateson, Alexander, & Murphy, 1987; McKeithen, Reitman, Rueter, & Hirtle, 1981; Vessey, 1988; Weiser & Shertz, 1983). Hence, the replication of existing program codes reveals weak comprehension of problems due to their limited ability in translating conceptual understanding into a practical solution. Consequently, this reduces students' motivation to excel in writing computer programs.

As of these novice learners, it is likely that is a lack of smooth interaction between their declarative knowledge (i.e. facts or knowledge about a concept or algorithm) with their procedural knowledge (i.e. how to implement the codes). In other words, they could experience difficulties in associating the potentially limited amount of knowledge possessed with the kind of procedures that should be taken to write a piece of working codes effectively. Commonly, students could accurately describe the general functions of an algorithm. However, similar ability is not demonstrated when they are required to describe the procedures of an algorithm step by step. This difficulty could be attributed to the large amount of cognitive load (Green & Petre, 1996) required for processing various tasks when solving a programming problem such as knowing how to best represent the information that describes the problem, consider the steps to convert information between different representations and plan the solution before writing the code using correct syntax.

2.2 Visualization in Computer Programming

In an effort to produce effective and interesting learning outcomes, various visualization-based approaches have been proposed in teaching and learning the DS subject. Three common approaches are 1) interactive multimedia systems (Andrade, Mercado, & Reynoso, 2008) such as websites and intelligent tutoring systems that show dynamic visual representations (i.e., animations and videos) and require students' active participation to gain its benefits, 2) problem-based-learning using computer graphics and image processing techniques such as that which requires the implementation of ray trace which enables real-time verification of visual feedback and allows the assessment of object oriented principles (Duchowski & Davis, 2007), 3) game-based learning that provides a competitive environment that requires students to code against other students' and their instructor's codes, which indirectly exposes these students to data structure concepts such as linked lists, stacks and queues (Lawrence, 2004).

Visualization or presentation of information in different graphical forms such as static pictures and dynamic animations have been studied to improve programming comprehension and composition (Adamchik, 2011; Meisalo, Sutinen, & Tarhio, 1997; Segura et al., 2008). However, a simple graphical representation is insufficient to support effective learning of the dynamic nature of algorithms (Stasko, Badre, & Lewis, 1993). For example, simply observing animations of an algorithm without understanding its notations and syntax would not necessarily improve a learner's understanding of the content. Hundhausen, Douglas, & Stasko (2002) suggest that interactive activities such as answering questions would promote better learning outcomes. A survey on software visualization (Price, Baecker, & Small, 1993) categorizes tools into those that 1) visualize the execution of code (i.e. code visualisation tools such as Jeliot and jGRASP)(Jain, Cross, Hendrix, & Barowski, 2006; Moreno, Myller, Sutinen, & Ben-Ari, 2004), and 2) visualize the concepts or animation of an algorithm without any code (i.e. algorithm animation tools such as JHAVE and MatrixPro) (Karavirta, Korhonen, Malmi, & St, 2003; Naps, 2005). The code visualization tool visualizes the static structure of a program which focuses more on a program's development, while the algorithm animation tool allows for a better understanding of dynamic program behaviour. Another type of visualization tool, known as ‘algorithm simulation’ (Hundhausen & Douglas, 2002; Karavirta et al., 2003), animates a particular activity or procedure within an algorithm (e.g. insert a value into a binary search tree by dragging a key from an array to that portion of the tree). This type of animation would help novice programmers to implement an algorithm and its visual graphics (Hundhausen & Douglas, 2002). In another review, Ragonis & Ben-Ari (2005) discussed the pros and cons of using static visualization and proposed that integrating it with dynamic visualization would amplify the benefits of learning programming in visual forms. jGRASP has both static and dynamic visualization functions.

2.3 Diagrams in DS to Facilitate Reasoning

Considering the advantages of graphical representation, we argue that learning DS with the aid of visualizations could potentially help students to understand encapsulation and abstraction in programming problems. Humans generally think of ideas from a “high-level” perspective. For example, we can perform a quick estimate for “measuring the area and perimeter of a square of known side lengths” without paying too much attention on whether the calculation of either the area or perimeter first would affect the final results. However, computer programs require detailed ‘low-level’ instructions, often represented by hundreds of lines of code. In the previous example, a detailed specification of whether the calculation for either an area or perimeter first needs to be written. Bauer & Johnson-Laird (1993) discussed that visual or graphical forms of programs can improve reasoning if used appropriately. In this context, *appropriately* refers to the use of graphical forms to complement and supplement reasoning, augment mental imagery, show how programs behave, and show the relationship between concepts and technical aspects (Petre, Blackwell, & Green, 1996). Visualization highlights the behavior of a program that is otherwise filled with texts. Given the limited internal knowledge among novice programmers, the graphical representation is expected to serve as a guide, “scaffold”, or structure for their reasoning process. It is estimated that this process would describe an algorithm more clearly. Furthermore, it is expected that this approach would reinforce problem comprehension and program composition tasks by strengthening the relationship between conceptual understanding and technical implementation. Petre et al. (1996) postulates that effective visual or diagrammatic representation may facilitate a student’s reasoning.

In studies related to the use of diagrams in data structures, Aharoni (2000) reported that students lack the ability to deal with abstraction, while Patel (2014) identified features of an intelligent tutoring system to increase learning efficiency. In a different study more related to ours, Davies (2008) evaluated students’ composition of diagrams and coding of a data structure operation (i.e., stack) depicted in the diagram. Students studied a description of a ‘stack’ problem before they were asked to draw a sequence of diagrams showing the state of the data structure operation. This was followed by writing code that described the operations. Davies (2008) reported that although students were able to translate the problems into pictorial representations correctly, many had problems translating them into code. The kinds of error produced were related to concepts, syntax, and misuse of computer memory allocation. In Davies’s study, students regarded writing code for the operations as a form of memorization rather than a conversion mediated by the diagrams. This discontinuity between diagrams and code suggests that the ability to interpret problems and produce correct pictorial representations is not enough to confirm students’ ability in writing code. Davies (2008) further proposed that for diagrams to be useful in aiding students in translating each change of a diagram into a logical line of code, an effective pedagogical method that involves a formal diagrammatic approach has to be developed. This means there should be formal rules about what constitutes a correct or incorrect diagram to represent a data structure. In this way, we expect that students would have a plan to support their code writing while solving a programming problem.

The proposed study differs from Davies’s by focusing on comprehension rather than composition of diagrams (2008). Students are evaluated on their ability to study given diagrams that illustrate data structure operations before translating them into program codes. We are interested to evaluate whether students' reasoning and code writing skills improve if cues are provided on diagrams. This was not tested by Davies (2008). We assume that most data structure topics such as linked lists are dynamic in nature where the allocated memory size is flexible and efficient, enabling operations such as memory space expansion and reduction (i.e. addition of new elements and removal of existing elements). This dynamicity enables us to evaluate how much students can think abstractly, allowing us to evaluate how students examine and interpret a given programming problem prior to code writing.

2.4 Learning Linked Lists with Diagrams

Linked lists are a type of data structure that stores information in a sequential manner. A linked list is composed of a set of nodes. In a singly linked list, each node is comprised of two portions of information: a value that stores the data of interest (such as integers or string objects), and a reference that links a node to the subsequent node of the list. The linking between the nodes is attained by the use of pointers, which reference the memory location of the nodes. A diagrammatic example of a node and linked list is shown in Figure 1.

The linked list is one of the most common topics taught for students taking the “Data Structure and Algorithm” course. In addition, this topic is chosen because it is one of the simplest topics in the DS course. However, our experiences in teaching and assessing this topic indicate that students often fail to grasp the correct concept which enable them to write their own program codes without any reference. Most students struggle to understand the linked list algorithm more than other potentially complex data structure topics such as binary search tree and stack. In learning the fundamental concepts of linked lists, students are expected to understand the relationship between pointers, links between the nodes, allocation or deletion of objects and traversal of lists to display values. These processes are fundamental for the more complex data structures. Thus, linked lists serve as a relatively easy foundation for students to begin learning data structures. This is because students would not be overloaded with other complicated tasks found in more advanced topics (e.g. stack, queue, binary search tree). In addition, each statement of the program code can usually be represented as a notation. Thus, it is likely that learning from diagrams is useful given that the notation and relationship between elements is understood. However, given the large number of processes that might appear too basic for most students to breakdown during their reasoning in performing common list operations such as addition or deletion of an element at a specified position in the list, the correct thinking processes required to structure a coherent solution for these problems may not be straightforward.

In our observation, one of the common mistakes students demonstrate while learning linked lists is attempting to begin writing program code right after they are being taught. This occurs without enough consideration to produce a logical solution. These students claimed that it is relatively easy to comprehend the concept, thus writing codes would not become a problem. Nevertheless, many students gain little success following this approach. As expected, novice students are too eager to solve problems before careful consideration in planning the structure of their programs. Our experience in teaching novice students by using the conventional approach, such as by explaining the relationship between concepts and corresponding program code, indicate that it is not very effective as proven by the high number of failure rates among students. Therefore, we adopt a strategy that employs diagrams as part of our teaching method for linked lists. Using this approach, we encourage students to evaluate their understanding by associating a particular list operation code with its diagrammatic representation. Our goal is to assess whether the diagrammatic approach is effective in improving students’ performance. If the proposed technique is effective, it can be generalized for other topics in data structure.

We expect that this approach might be useful for novice learners by facilitating them to better visualize the abstract concepts of a linked list. We further expect that students who are trained to read the diagrams, which represent the steps of an algorithm, would learn the proper way of thinking through this problem. This is because better understanding of the processes involved in a particular operation guides learners’ attention to the next most relevant and logical step in producing a cohesive solution. For example, in an operation to add an item to the beginning of a list, a problem solver must first consider whether the specified list is empty. If the list is empty, pointers such as the head and the tail must refer to the newly created element that will serve as the first item on the list. Otherwise, the new element will be linked to the first existing node before the head pointer is re-pointed to the newly added element. This means that a learner who is able to coherently relate one concept with another during the problem-solving process could potentially secure improved procedural skills in writing codes. These relationships between portions of the diagrams are defined as *advance organizers*. *Advance organizers* create structure in a learner’s thinking process (Macfarlane & Mynatt, 1988). Therefore, this approach may benefit novice programming students by way of assisting them in developing comprehensive knowledge in computer programming, both technically and conceptually.

3.0 THE STUDY

In the work reported in this paper, we aim to compare three types of diagrammatic representations and evaluate its usefulness for novice programmers to solve programming problems involving linked lists. The problem posed in a quiz setting required students to analyze given diagrams and answer questions by naming corresponding operations before writing the program codes. Three versions of diagrams represented in the standard linked list notational format were designed (Figure 1, 2 and 3), each differing by the amount of complexity and information provided. The diagrams are expected to assist students’ reasoning processes for solving a given problem. Each diagram is presented as a main question of the quiz. The diagrams illustrate a variation of either complete or partial algorithm procedures for common linked list operations (i.e. add an element first in the list, add an element at an index location, and remove the last element in the list). These concepts taught in the classroom are similar to those found in textbooks. Each portion of the illustration can be translated into corresponding program codes. Respectively, the following variation of diagrams represents these operations: add an element first in the list, add an element at a particular index and remove the last element from the list.

As shown in Figure 1, the first version (Q1), called a “partial and labelled” diagram, only showed a portion of the notation (i.e., the nodes and arrows to represent elements of a list and their relationship) with labels to describe part of an unnamed operation (i.e., operation Z). The second version (Q2) shown in Figure 2, known as a “complete and unlabelled” diagram, showed a full notation without labels for a different operation (i.e., operation Y). The third version (Q3) shown in Figure 3, called a “complete and labelled” diagram, also showed a full notation supported with labels describing another type of list operation (i.e., operation W).

Students were asked to name the operations depicted in the diagrams prior to writing the corresponding program code. No “partial and unlabelled” diagram was given as information provided by this representation is considered insufficient and less meaningful to support effective reasoning.

3.1 Hypothesis

We predicted that the “complete and unlabelled” (Q2) diagram (Figure 2) would assist the low performing students more than diagram versions 1 and 3 due to the additional fill in the blanks task that would serve as an advance organizer for the task that follows. Diagram version 3 is expected to be the most difficult while diagram version 1 is expected to be moderately difficult in facilitating novices in solving programming problems associated with linked lists. We are interested to further evaluate the extent to which diagrams improved comprehension or students’ conceptual understanding when learning about linked lists. Thus, the assessment will analyze the correlation between students’ ability of naming an operation illustrated in the diagram and writing code for the respective operation. This evaluates students’ competence in code comprehension in the form of graphical representations and code composition upon analyzing the diagrams.

3.2 Structure of the Observed Learning Outcome (SOLO) Taxonomy

Given the nature of the test material used in this study that increases in complexity, the SOLO taxonomy is considered as the most useful model used in evaluating students’ learning performance. The SOLO taxonomy is a model that describes students’ understanding of a learned topic in levels of increasing complexity (Biggs & Collis, 1982). This type of model is useful in evaluating the quality of learning as it progressively becomes complex. The taxonomy is divided into five categories, namely 1) pre-structural (p): a student’s response indicates bits of unconnected information that are commonly illogical or unreasonable, 2) uni-structural (u): a student’s response focuses on one relevant aspect that emphasizes on basic concepts about the topic being tested, 3) multi-structural (m): a student’s response focuses on several relevant independent aspects showing that learners understood several concepts about a topic, but the ideas are disconnected, 4) relational (r): a student’s response indicates adequate understanding of a topic as demonstrated by the integration of different aspects into a coherent structure. Students demonstrate mastery of a topic by their ability to relate many parts together, 5) extended abstract (ea): a student’s response indicates the ability to create new ideas based on the mastery of a topic. This is the result of the abstraction of a coherent structure to a new topic. Therefore, each category of the SOLO taxonomy can be considered as an extended progression from the previous category in the manners of teaching and learning. Bloom’s taxonomy (Bloom, Englehard, Furst, Hill, & Krathwohl, 1956) may not necessarily indicate this association clearly. The evident advantages of the SOLO taxonomy in this study are - 1) the categories can be interpreted relative to the students’ proficiency of the learned topic, 2) the evaluation of how students learn is more closely related to what they have been taught, 3) each category can be a marker to establish the number of ideas which reflects a student’s learning performance.

The SOLO taxonomy used in this study (Table 1) is adapted from (Murphy, Fitzgerald, Lister, & McCauley, 2012; Whalley et al., 2006). Only four categories are used in the evaluation. The fifth category (extended abstract) describing students’ ability to extend their knowledge in solving other problems is not discussed further because there are no given problems with respect to this assessment. However, the multi-structural category is divided into two sub-categories as there was a trend of errors occurring at different complexity levels, 1) Multi-structural-A: answer contains two or more correct portions of the solution, but code consists of very minor errors such as incorrect use of variable names; 2) Multi-structural-B: answer contains two or more correct portions of the solution with some parts of the code in the wrong order or missing.

Table 1: The SOLO taxonomy for evaluating students' responses on diagrammatic linked list problems shown in ascending structural complexity

Category	Description
Pre-structural (P)	Provided answers show lack of knowledge or the answers given are unrelated to the question.
Uni-structural (U)	The answer contains a correct description for only one portion of the code.
Multi-structural (M)	The answer contains two or more correct portions of the code with some irrelevant or unrelated portions.
Relational (R)	Provides complete and correct answers for all code statements and their association.

4.0 METHOD

Prior to the assessment reported in this work, students were taught the concepts of standard linked list operations during lectures with the aid of diagrams such as those shown in Figure 1. All the students are familiar with the corresponding notations and syntax used in the test. During lectures, the relationship between parts of the diagram and the corresponding program code was described and shown to students using diagrammatic sketches on the board and an integrated development environment (IDE). The students learned other operations besides those tested in the assessment.

4.1 Participants

A total of 96 undergraduate computer science students (58 male, 38 female), aged between 19-24 years old (Mean age=20.7 yr.; SD=1.02 yr.), from a large publicly funded university participated in the assessment in partial fulfillment of a course requirement. The participants, who were in their second semester, enrolled in their second programming related course titled "Data Structure". The course offered for a duration of 14 weeks was taught using Java. The students also programmed in Java in their previous programming course (CS1). Most students spoke English as their second language.

4.2 Procedure

All data were collected during a written quiz that had a total of 65 points and made up 15% of the final assessment for the course. In four out of the five exam questions posed, students were required to analyze diagrammatic representations of linked list algorithms (i.e. each diagram specifies a different linked list operation), provide an appropriate name for the algorithm and write the corresponding code. The questions posed common data structure algorithms that they had already learned in a standard classroom setting one week prior to the quiz. The quiz lasted for an hour during a lab session. The students were informed about the quiz one week in advance. Access to a compiler, learning resources or peer discussions were not allowed during the quiz.

4.3 Stimulus

Only problems (i.e. questions) 1, 2 and 3 are evaluated in this study as these questions use diagrams. Thus, all students are required to answer questions 1, 2 and 3. Problems posed in question 4 and 5 required students to analyze a piece of functional code given in incorrect order and write them in the correct order and write code from a given method signature following a textual description about the function of the method. Thus, they were not directly related to the present study.

The marks are distributed as follows: Question 1 (11 points), Question 2 (28 points) and Question 3 (11 points). Every part of each main question was worth one point. As for the question that requires them to write code for list operations, no additional deductions were inflicted for any wrong statements written. As code writing was done on paper, minor syntax errors (i.e. missing a semicolon or closing brace) that do not affect the evaluation were disregarded. The final score was calculated as the total number of correct statements for each main question.

Generally, all the problems require students to analyze the algorithm depicted in a diagrammatic representation. The diagrams are conventional graphical representations (i.e., nodes) used to describe a singly linked list (Newell & Shaw, 1957). Students were expected to trace the pseudocode and provide an appropriate name for the corresponding linked list operations shown in the diagrams. This task is followed by code writing, where students are required to fill in the blanks provided.

Question 1: Diagram shows a portion of a list operation

Figure 1 shows the problems posed in Question 1 with the following description "The diagram above shows a portion of an operation Z". A node with links to a pointer is given. Only a portion of the full operation is posed.

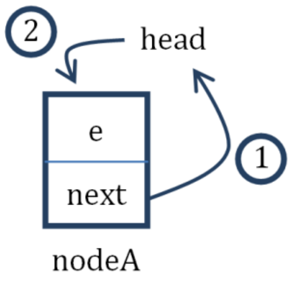


Figure 1: Diagram representation of operation Z with 'partial and labelled' specification (Q1)

Question 1 is divided into three parts. Each part requires students to – a) Name operation Z; b) Write a line of code for each numbered circle (1 and 2); c) Complete the program that described operation Z by filling in the blanks as shown in Figure A (Appendix).

Question 2: Diagram shows semi-labelled specification of a list operation

Figure 2 shows a collection of linked nodes that specifies the algorithm of a list operation by the numbered circles. Question 2 is divided into four parts that require students to a) label the diagrams; b) write code for the corresponding numbered circles; c) name the list operation; d) complete the program by filling in the blanks similar to Q1(c). Instructions in Q2(d) are as follows "Write the code you completed in Q2(b) in the correct arrangement for operation Y. Write the correct method name to replace operation Y."

Question 2 (a): Assume that a test program calls the following method: operation (3, f).

a) Label all the nodes shown in the diagram.
b) Write the code/statements for the numbered circles.

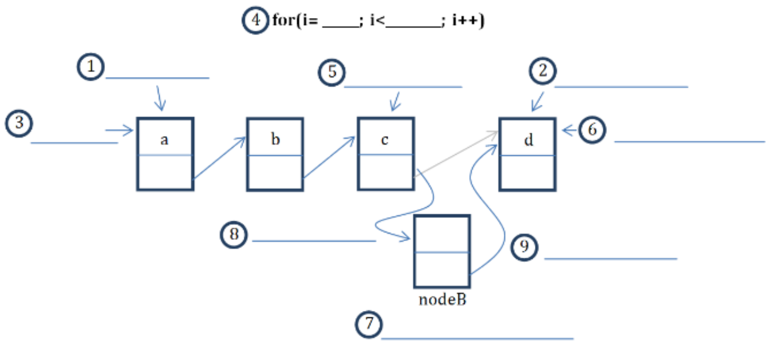


Figure 2: Diagram representation of operation Y with 'complete and unlabelled' specification (Q2)

Question 3: Diagram shows labelled specification of a list operation

Question 3 as shown in Figure 3 is different from Question 2 in a way that all labels are provided for the complete diagrammatic representation of a different list operation (i.e. remove the last element from a list). Question 3 is divided into two parts where students are required to a) name the operation; b) write the code for the named method in the space provided.

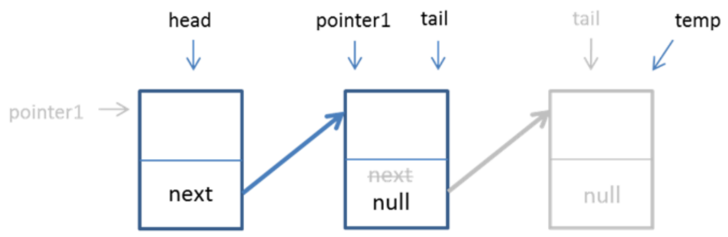


Figure 3: Diagram representation of an operation W with 'complete and labelled' specification (Q3)

Instructions for Q3(b) are given as "Write code to represent the given diagram. Write the correct method name to replace operation W."

4.4 Data Coding

All of the authors (hereafter referred as analysts) independently classified all 96 students' responses to questions 1, 2 and 3 according to the SOLO taxonomy described in Table 1. Thus, this study adopts the fully crossed design for assigning the analysts to rate the questions (i.e. each data point was rated by multiple analysts). Except for the first author, all analysts did not teach the course for the collected data. However, all are experienced researchers and academics, some of them having taught programming courses at the same institution. Thus, all of the analysts are considered to have acquired the appropriate skills to interpret the rating rubrics acceptably.

The rating rubrics prepared by the first author were discussed and agreed upon among all analysts. Prior to the coding, the Frame of Reference (FOR) training session involving all analysts to code at least five data sets were performed to ensure common understanding for a consistent rating system. The initial rating calibration acquired 80% adjacent agreement for the data assessed. This was considered to have achieved excellent range for the interrater agreement analysis. The duration of the training was 2 hours.

On completion of the FOR session, the analysts began coding the data individually. All answers were coded blind to conditions. After that, they met to discuss the ratings and converged to solutions for any differences in their coding results. Details of these discussions are covered in Section 5.2: Quantitative Results for statistical description.

5.0 RESULT

The answers to questions 1, 2 and 3 are given in Appendix A. Table 2 shows examples of typical correct responses which are considered acceptable for each SOLO category discussed in Section 3.2: SOLO Taxonomy.

As shown in Table 1, responses were classified as relational (R) when answers were complete and accurate. However, a response was considered pre-structural (P) when the answers were entirely incorrect or unrelated to the question. Answers which were 10-20% correct compared to the answer scheme (or when only one portion of the code was correct) were assigned the uni-structural (U) category. The percentage of correctness was used as a rule-of-thumb in the assessment as the number of lines and complexity of the code varies. Responses classified as multi-structural-a (Ma) were mainly correct except for very minimal errors (90% correct) such as one missing or incorrect statement out of seven lines of code, as shown in Table 2. Answers which consist of 50% - 90% incorrect lines were classified as multi-structural-b (Mb). An example of the Mb category is when the answers contain partially incorrect statements either written in the wrong order or due to missing codes.

Table 2: Example responses for some questions

	Example Responses	**Question**
Pre-structural (P)	.getFirst *//incorrect*	1b
	Linkedlist *//incorrect*	3a
Uni-structural (U)	line 1 : nodeA.next = head; line 2 : head=tail; *//incorrect*	1b

	line 1 : public void removeLast()
	line 2 : Node<E> temp = new Node<E>();
	line 3 : temp = tail;
	line 4 : temp.next = null; *//incorrect*
	line 5 : size--;
		3b

Multi-structural-a (Ma)	line 1 : addFirst (E e)
	line 2 : Node<E> nodeA = new Node<E>(e);
	line 3 : node.next = head;
	line 4 : head = nodeA;
	line 5 : *no answer* *//incorrect*
	line 6 : tail;
	line 7 : tail = head ;
		1c

	line 1 : next line 2 : next line 3 : next line 4 : f line 5 : next line 6 : next *//incorrect*	2a

Multi-structural-b (Mb)	line 1 : addFirst (E e)
	line 2 : Node<E> nodeA = new Node<E>(e);
	line 3 : head = node.next;*//incorrect*
	line 4 : head = nodeA;
	line 5 : *no answer* *//incorrect*
	line 6 : head; *//incorrect*
	line 7 : head = nodeA; *//incorrect*
		1c

	line 1 : head line 2 : tail line 3 : temp line 4 : for (i=1 ; i<3; i++) line 5 : temp line 6 : temp *//incorrect* line 7 : temp *//incorrect* line 8 : new node is insert*//incorrect* line 9 : after a new node is inserted *//incorrect (line9)*	2b
Relational (R)	addFirst(int index, E e)	2c

	line 1 : public E removeLast()
	line 2 : Node<E> pointer1 = head;
	line 3 : for(int i=0; i<size-2; i++)
	line 4 : pointer 1 = pointer1.next;
	line 5 : Node<E> temp = tail;
	line 6 : tail = pointer1;
	line 7 : tail.next = null;
	line 8 : size--;
	line 9 : return temp.element;
		3b

Most of the analysts rated the answers by closely following the appropriate SOLO category. In most cases, ratings to determine whether an answer falls in the uni-structural (U) or pre-structural (P) category were relatively consistent. However, during the post-analysis discussion, it was reported that several responses received mixed ratings between the two multi-structural categories (i.e. Ma and Mb). These mixed ratings occurred mostly for answers on ‘write the program code’ tasks (i.e., 1c, 2b, 1d, 3b) due to longer answers and varied interpretations in evaluating the code among the analysts (see below for details). On the contrary, the analysts agreed that classifying students’ answers for the ‘name the operation’ tasks (i.e., 1a, 2c, and 3a) were straightforward. The coherent rating assessments acquired from these responses suggests a collective understanding of the rubrics among the analysts. Thus, each category was measured individually for statistical analyses.

5.1 Interrater Reliability of Data Coding

In general, the analysts’ ratings were similar for most of the data. However, the analysts disagreed on certain responses. The discussion focused on the accuracy of the responses and the interpretation of the SOLO taxonomy. As for naming the operation task, analysts initially disagreed on answers such as “add” to question 2(c). The correct answer is “addFirst”. This is because adding an item in a linked list operation can be described in at least three locations such as add first, add last and add at a particular position. In question 2(c), the students were expected to specify their answers accurately (i.e., add(index) or adding an item at a particular position in the list). Two of the four analysts considered this answer as M(a), while the others as relational (R). However, after deliberating on the vague meaning of the word “add”, all analyst later agreed on accepting this statement as multi-structural-a, M(a).

The analysts reached a consensus to accept several styles of answers such as those which used different terminologies as long as the answers are conceptually correct (e.g. student’s answer was “deleteLast” and answer in the scheme was “removeLast”). In some cases, a student’s written code does not use the same variable names as that shown in the diagram (e.g. variable in the diagram is nodeA, but student wrote newNode). All analysts accepted the inconsistent usage of variables if this does not largely affect the provided answer. However, statements that consist of logic errors such as “node A = head” were not accepted. Although the syntax of this kind of answer is correct, this answer is logically incorrect. The correct answer is “head=nodeA”. Some analysts perceived the interpretation of the SOLO taxonomy in a strict manner against the answers observed. The analysts looked at the meaning of the answers in general. One analyst opined that answers written in plain English deserve correct marks (e.g. stop iterating at c; point c.next to new; create node). However, the others argued that this does not demonstrate a technical understanding of the tested concepts. Furthermore, accepting this kind of answer would be difficult to evaluate the students’ full understanding with succinct clarity. There were 13 cases of answers where plain English descriptions were used. Five of these cases were classified as pre-structural (P) when prior and subsequent parts of the same question were incorrect. Otherwise, this issue was resolved by classifying the answers as uni-structural (U) or multi-structural-a (Ma). As a consequence of the discussion, all analysts agreed on the changes before the final data was submitted for statistical analysis.

Table 3: The interrater agreement for the ICC analysis using type consistency and two-way random model
Note. M = mean. ICC = Intraclass Correlation Coefficient

ICC	**Q1**			**Q2**				**Q3**		**M**
	Q1a	**Q1b**	**Q1c**	**Q2a**	**Q2b**	**Q2c**	**Q2d**	**Q3a**	**Q3b**	
Single measures	.938	.909	.819	.611	.496	.676	.515	.926	.823	.746
Average measures	.984	.976	.948	.862	.797	.893	.810	.980	.949	.911
Percentage of agreement (%)	91.4	78.1	76.4	67.7	57.7	67.7	55.1	91.2	63.5	72.1

5.2 Quantitative Result

The test consists of five questions, out of which only three questions were studied as the goal was to evaluate students' performance in solving diagrammatic notational problems. The SOLO categories of P, U, Ma, Mb and R were each coded into an integer as P→1, U→2, Ma→3, Mb→4, and R→5. An average SOLO rating was also computed for each question across all analysts' responses. The rating process took approximately one week for each analyst. They rated all datasets at once on completion of data collection as a result of a single measure. Each problem was analyzed as a single category. Due to the nature of the exam score and the SOLO ratings that are not uniformly distributed along with the small sample size used in the study, non-parametric statistical tests were adopted in the analysis, using the SPSS package.

5.3 Interrater Reliability of Data Coding

The sample data calculated for reliability analysis was the same as the full sample acquired in the study. A total of 3456 points of data (96 datasets x 4 analysts x 9 variables) were rated. There were 4 analysts and the data collected was on an ordinal scale. Thus, the interrater agreement was computed using Intraclass correlation coefficient (ICC). ICC gives an estimate of the proportion of variation in ratings that is due to the rate of performance rather than the raters' errors in evaluating the data.

The level of interrater agreement is calculated using the Intraclass correlation coefficient with the Two-Way Random model specification as 1) all analysts consistently rated every data point, and 2) the analysts were a part of other potential analysts. As a result, this method models the effects of analysts and rating assuming that both effects are acquired randomly from larger populations (i.e. random effects model). Table 3 shows the ICC analysis results using the consistency type for the interrater agreement correlation analysis.

The results shown in Table 3 can be interpreted as follows: the interrater agreement for Q1 (a) for the mean scores for all analysts (average measures) is estimated to be 0.984, indicating that 98% of the observed variance is due to the true variance. The cumulative average measure for all questions is 91%. The average measures showed an agreement between the averages of all analysts' rating with the average ratings computed by another similar group of analysts. In other words, this result shows a high similarity in ratings between the analysts. Equally, the single measures record 94% for Q1 (a), indicating high agreement between one analyst with another analyst. In other words, the evaluation rating of one analyst is almost a perfect agreement with ratings recorded by another analyst. Generally, the ICC results show that the reliability score for a single analyst is 75% and the mean measures for all analysts (i.e. average measures) is 91%.

The percentage of agreement was computed to evaluate the consistency between evaluators in terms of the ordering of performance ratings, rather than the absolute value of their rating. This shows how the analysts interpret the SOLO rubrics and rate the data. The percentage of absolute agreement shown in Table 3 is calculated using (Fleiss, 1971). This analysis indicates how often the analysts agree on the exact rating. Generally, the analysts' agreement reached substantial coherence in their levels of agreement. The highest percentage recorded was 91.4% (Q1a) and the lowest was 55.1% (Q2d) with an average percentage of agreement of 72.1%.

This consistency was also evaluated using the percentage of exact and adjacent agreements and results confirm our findings. In summary, analysts reached a reliable analysis as shown in Table 3 with relatively accurate analysis of 91% agreement (on average) between them and absolute agreement of 72% (on average). The high-level agreement among these analysts is coherent and ratings are consistent according to interpretations of the SOLO scale.

5.4 Correlation Tests Within and Between Questions

Table 4 shows the mean rating of the questions calculated from all analysts' ratings. Question 2(a) showed the lowest mean score (2.90) while the highest mean score was recorded for Question 3a (4.58). The lowest mean score for Q2a which is on labelling a diagram could indicate that many students did not manage to answer this particular question, potentially due to misunderstandings on the instructions posed. The highest mean score for Q3a on naming the operation shows that the depicted operation is well-defined allowing for easy interpretation. The means for each question were 3.99 (Q1), 3.42 (Q2) and 3.96 (Q3) respectively.

Table 4: The mean (M) and standard deviation (SD) of all questions and parts

Question	**M**	**SD**
Q1a	4.53	1.20
Q1b	3.74	1.61
Q1c	3.69	1.12
Q1	3.99	1.08
Q2a	2.90	1.38
Q2b	3.19	0.67
Q2c	4.20	0.91
Q2d	3.39	0.78
Q2	3.42	0.59
Q3a	4.58	1.13
Q3b	3.34	1.26
Q3	3.96	1.00

Prior to statistical analysis, a normality test was done and we found that data was not normally distributed. Therefore, Spearman's rho coefficient correlation test was computed to assess the relationship between ratings for different parts of each question (e.g., Q1a – Q1b, Q1a – Q1c, Q1b – Q1c). Table 5, 6 and 7 show the correlation results for Q1, Q2 and all questions (i.e., Q1-Q2-Q3).

Table 5: Spearman’s rho correlation results for part a, b and c of Question 1

	Q1a	Q1b
Q1b	.396**	-
Q1c	.466**	.564**

Note. **. Correlation is significant at the 0.01 level (2-tailed).

Table 6: Spearman’s rho correlation results for part a, b and c of Question 2

	Q2a	Q2b	Q2c
Q2b	.161	-	
Q2c	-.067	.097	-
Q2d	.113	.410**	-.004

Note. **. Correlation is significant at the 0.01 level (2-tailed).

Table 7: Spearman’s rho correlation results for Question 1, 2 and 3

	Q1	Q2
Q2	.551**	-
Q3	.395**	.387**

Note. **. Correlation is significant at the 0.01 level (2-tailed).

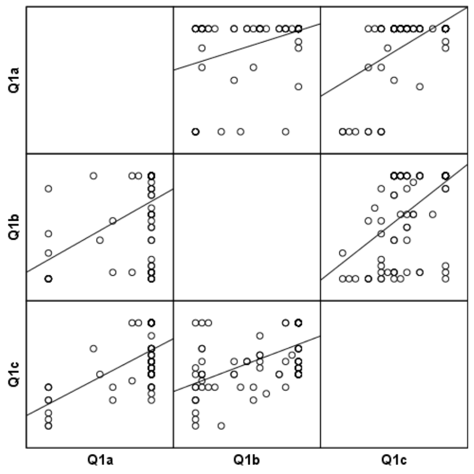


Figure 4: Scatterplot for parts a, b and c of Q1 which shows a moderately positive correlation between all parts

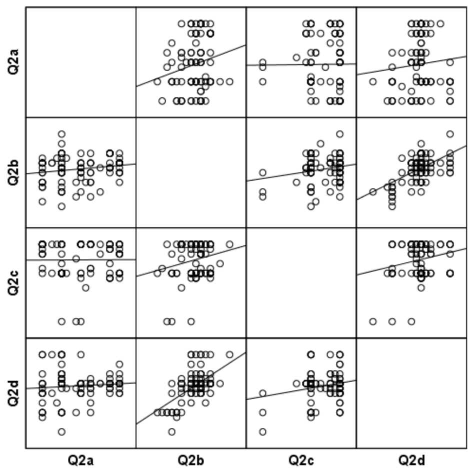


Figure 5: Scatterplot for parts a, b and c of Question 2 which shows a moderately positive correlation between parts 2b and 2d

In Question 1, it was shown that a correlation was found between all parts (a, b, and c) as follows: 1) Q1a and Q1b, r(96) = .396, p < .001; 2) Q1a and Q1c, r(96) = .466, p < .001; 3) Q1b and Q1c, r(96) = .564, p < .001. A scatterplot (Figure 4) shows the positive graphical relationship. Overall, there was a moderately positive correlation between parts a, b and c for Q1. Thus, students' ability to name the operation correlates with increased ratings in code writing for Q1.

As shown in Table 6, Question 2 only shows a correlation for Q2b-Q2d, r(96) = .410, p < .001. Figure 5 shows the scatterplot for these results. Generally, there was a moderately positive correlation for b-d in Q2. Thus, students' ability to fill in the blanks for the diagram (Q2b) was moderately correlated with the ratings for code writing (Q2d). However, the ability to name the operation in Question 2 did not correlate with ratings in code writing.

In Question 3, a moderately positive correlation was found between Q3a – Q3b, r(96) = .395, p < .001. It can be interpreted that students' ability to name the operation for Question 3 was proportional to ratings in code writing.

The relationship between all three questions is shown in Table 7. The overall score between the questions shows significant effect for 1) Q1-Q2, r(96) = .551, p < .001; 2) Q1-Q3, r(96) = .395, p < .001; 3) Q2-Q3, r(96) = .387, p < .001. This result shows a moderately positive correlation between Q1, 2 and 3 as shown in Figure 6. Thus, students answered the questions across different levels of complexity.

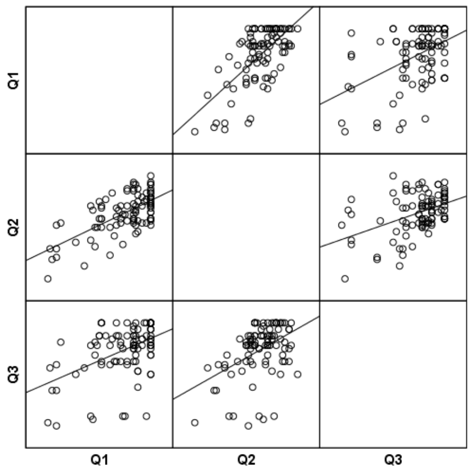


Figure 6: Scatterplot for Questions 1, 2 and 3 which shows a moderately positive correlation between all questions

5.5 Comparison between Questions

In order to evaluate whether there was any difference between tasks for different questions, the Kendall's coefficient of concordance test, Kendall's W, was conducted. This analysis was not carried out for the parts within a question because the nature of these parts across the question does not exhibit the same level of complexity.

The correlation for naming the operation tasks between Q1a, Q2c and Q3a was found to be significant, $p < .001$, with the average mean score (i.e. homogeneous subsets) higher for both Q1a(2.18) and Q3a(2.24) compared to Q2c(1.58). A pairwise comparison showed significant effects between 1) Q2c and Q1a, $r(96) = .594$, $p < .001$; and 2) Q2c and Q3a, $r(96) = .656$, $p < .001$. However, correlation for the code writing task between Q1c, Q2d and Q3b was found non-significant, $r(96) = .031$, $p = .05$.

Overall comparison between the questions showed a significant effect, $p < .001$. A similar pattern with that found in the operation naming task was found in this comparison, showing a higher mean score (i.e. homogeneous subsets) for both Q1(2.25) and Q3(2.32) compared to Q2(1.43). The pairwise comparison showed a significant correlation between 1) Q2 and Q3, $r(96) = .823$, $p < .001$, and 2) Q2 and Q1, $r(96) = .896$, $p < .001$. Overall, the comparison between all questions indicated that Q2 is significant or different as both Q1 and Q3 showed higher mean scores (i.e. there is non-significance between Q1 and Q3).

6.0 DISCUSSION

The aim of this study is to investigate the effectiveness of static diagrams usage in data structure education, especially in assisting problem solving. This measures the students' competence in comprehending an algorithm from a graphical representation and composing code upon analyzing the diagrams. Among the diagrams that support learning and problem solving, it was hypothesized that the notation presented in the Q2 provided an easier support than Q1, with Q3 being the most difficult of all. However, as will be discussed below, both Q1 and Q3 were found to be easier than Q2.

The inter-rater analysis results showed a high agreement between coders in the ratings. According to the guidelines for the interpretation of ICC measures (Cicchetti, 1994), all of the inter-rater agreement measures reached excellent levels of agreement among the analysts. This shows that our analysts were consistent in their coding of the SOLO category used.

6.1 Question Complexity and Difficulty

Further analysis involves evaluating the level of difficulty of the questions, translated from the students' scores in answering the diagrammatic questions. Students were required to evaluate the processes involved in the specified operations. This activity necessitates the use of at least three levels of the SOLO taxonomy including identifying the basic concepts of linked list operations (unistructural), describing the relationship between the processes considered for a particular operation (multistructural), interpreting the existing knowledge into the context of the question (relational) and analyzing the difference between processes involved in the operation (relational).

The difference in diagrammatic representation between Q1 and Q3 is that in Q1, a portion of various processes for adding a new item to an empty list was shown while in Q3 the complete process necessary for removing the last item was given. A more complex operation was shown in Q2 that requires students to label numbered parts before writing code. The code writing activity was expected to be derived from the labelling activity. Thus, among all questions, Q2 was assumed to be the easiest to answer as step-by-step cues are given as labels. Q3 was hypothesized to be the most difficult and Q1 was assumed to be of intermediate difficulty.

6.1.1 Q1 and Q3 are Easier than Q2

Generally, collective results suggest that Q1 and Q3 were easier than Q2. However, between Q1 and Q3, it is difficult to specify which was more evidently difficult than the other. Q1 depicts part of a process for an add operation of a singly linked list. This required students to first translate the numbered processes into code (Q1b) before using this to write the full program. Q1 was considered easier as confirmed by the correlation where students' ability to name the operation correctly also ensured that they were able to write the code well in increasing difficulty. Students effectively translated the diagram into code as demonstrated by the high mean ratings and positive correlation between parts a, b and c. Q1b is the translation of the diagram into code, while Q1c requires students to translate their knowledge from Q1b into a fully functional program. Competence in using conceptual data structure knowledge of the add operation in writing functional code at the application level of Bloom's taxonomy showed that students attained sufficient conceptual and reasoning skills for questions like Q1.

A similar result as Q1 was reported in Q3 which illustrates a set of diagrams that represent a different operation. A moderate correlation found between the naming operation task (Q3a) and the code writing task (Q3b) suggests that students reasoning ability were moderate in evaluating the processes illustrated in the diagram of Q3. The complete description of processes depicted in the diagram appears influential in guiding students to write the program code. However, caution must be taken about the claims for Q1 and Q3 being considered easier than Q2. Among the reasons is that Q1 and Q3 are the more fundamental operations (add first and remove last) than that shown in Q2 (i.e. adding an item to an existing list). Therefore, much emphasis could have been given in introducing the linked list concepts for the add and remove last operations during teaching and learning prior to the test. This creates a more familiar association for these operations. In addition, students' ability in writing code in Q1 may be contributed by the simpler presentation of the diagram shown. Thus, the optional consideration for solving the problem is more focused and has a more straightforward answer for Q1. Another reason for better performance in Q3 may have been contributed by the presentation of the notation in Q3 that emphasizes the processes in the form of solid and lighter colour shades. This could have provided a strong cue during problem solving.

6.1.2 Why is Q2 Difficult?

In Q2, the lack of correlation between the naming operation (Q2c) and code writing (Q2d) suggests that students were weak at reasoning the processes involved in the 'add an item at a specific location in the list' operation to correctly identify their name, which in turn causes difficulties in writing the program code. It appears that the availability of structured numbered cues (given that each was numbered in a consecutive order) was not fully beneficial as a form to facilitate student's reasoning in guiding them to write the code for Q2. Hence, this type of question is the most difficult type to answer, as is supported by its lowest mean rating among all questions. However, the moderate correlation between the fill-in-the-blanks task (Q2b) and code writing task (Q2d) indicates that the ability to perform the fill in the blanks task means a higher possibility of being able to write the code. The lowest mean mark was recorded for Q2a. A contributing reason for this result could be that many students were potentially unaware of the requirement of this part of the question or could have mistakenly understood this part as requiring them to do similar tasks as the following part, which made the students focus on the fill in the blanks task, rather than completing the answers in the 'nodes'.

6.2 Comparison across the Operation Naming and Code Writing Tasks for all Questions

Further analysis compared the difficulty level of the questions with respect to operation naming, code writing and diagram difficulty. As for the operation naming task, Q2 was a more difficult question to answer as the results once more confirmed significant pairwise correlation between the same activity for Q1 and Q3. Again, the reasons for this could be due to the kind of cues (i.e. fill in the blanks) that were not seen effectively by the students or their conceptual knowledge for this operation is weak compared to that of the more fundamental operations shown in Q1 and Q3, respectively. The lack of ability to satisfactorily interpret their existing knowledge and examine the processes shown in Q2 shows limited proficiency in forming higher forms of thinking as that described by Bloom's taxonomy (Bloom et al., 1956). Perhaps, diagrams labelled with complete processes for a particular operation was a more useful cue for the student as shown by collective results of Q3 that was answered better than Q2. However, this claim needs further verification in future studies.

The non-significant results found between the questions for the code writing tasks may suggest that the complexity level of the operations posed in each question did not have a strong effect on the student's ability in writing code. However, this would be a weak claim. Perhaps, a more appealing argument is to discuss the complexity of the diagrams and its effect on students' performance. In other words, which level of diagrammatic complexity was considered most suitable to assist students in their reasoning and problem solving? Given the present results, students are more inclined to perform better in answering questions with less complicated diagrams such as Q1 and those with fully labelled cues (Q3). However, as the complexity of the diagrams increases (i.e. more notation, but lacking direct cues), this may raise further confusion among novice programmers and may well increase their cognitive load in answering Q2 effectively. Perhaps, there might be a limit for the amount of complexity for the diagrammatic notation used to support learning. Investigation for an ideal level of complexity for these diagrammatic representations to support learning is another potential future work.

The limitation of the present study is in the selection of operations as the diagrammatic stimulus which produces questions at different levels of complexity. For example, Q1 and Q3 could have been treated as easier than Q2, as the latter requires more detailed consideration. In addition, each diagram only represents one type of operation. It would be interesting to evaluate the results from a variation of these diagrams and operations. The present results are consistent with the findings reported by Davies (2008). A formal rule for specifying the benchmark of using the diagrammatic presentation to support code comprehension and writing warrants further investigation. The results also strengthen existing findings with respect to learning data structure which indicates that visualization in its standard representation may not be sufficient to guide and promote novice programmers' reasoning strategies (Colaso, Kamal, & Saraiya, 2002; Stasko et al., 1993).

6.3 Future Work

The data and results acquired from this study are inconclusive to describe whether the diagrammatic representation is useful in program solving, in particular learning highly conceptual programming topics such as linked lists. If so, further questions as to what level of complexity and number of cues are most suitable to assist novice programmers in both programming problem comprehension and code composition warrant further investigation. At least from the present findings, it is difficult to assure which type of diagram and level of cues were most useful. Further tests to verify the outcome from other linked list operations are necessary to confirm these results. In addition, the effectiveness of using diagrams in facilitating students in their reasoning and guiding them for solving programming problems can be better evaluated if compared with studies that include the use of diagrams against those without any use of diagrams in a controlled environment. Furthermore, given the present data and results, it is unlikely to state with great confidence whether each process shown on the diagrams was the key factor that helps students in strategizing their solution steps, especially in writing the program code. This is because the students could have memorized the program available from notes and textbooks upon training and learning, although they were unaware of the questions posed to them prior to the test. Prior access to existing materials and memorizing them is difficult to control as these are the fundamental operations available in text as full program code.

An experimental design which can separate testing between students' knowledge based on memorization and those acquired from knowledge transfer (based on interpretations of the diagrams or external representations) would be an added contribution to the computer science education domain. This is observed through some code that looked similar to those that can be found in textbooks. Although some students wrote the code correctly, some demonstrated a close description of what was found in the notes and textbook, their ability at replicating what they have learned suggests a weak form or low level of abstraction. Their capacity of answering based on 'programming-language oriented thinking' (Aharoni, 2000), are heavily influenced by their familiarity in learning DS using the Java programming language. It is imperative for students to develop high-level abstraction to ease problem solving particularly in DS topics. A potential avenue to develop increased levels of abstraction in solving DS problems among the students is to first provide an environment for them to partake in various activities including tutorials, labs, and assignments. The implementation of DS would follow once students reach a sufficient level of understanding of abstraction.

7.0 ACKNOWLEDGEMENT

This work has been funded by the following grants, FRGS (FP062-2014A, FRGS/1/2014/ICT07/UM/03/3) and UMRG (RP030C-12ICT). We thank Dr. Mahmoud Danaee for his support in the statistical consultation for this work. We also thank the students who participated in the study, and the reviewers and associate editor for their comments which improved this manuscript.

REFERENCES

[1] Adamchik, V., "Data structures and algorithms in pen-based computing environments", *2011 IEEE Global Engineering Education Conference (EDUCON)*, 2011, pp. 1211–1214.

[2] Aharoni, D., "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*, 2000, *32*(1), pp. 26–30.

[3] Andrade, E., Mercado, C., & Reynoso, J., "Learning Data Structures Using Multimedia-Interactive Systems" in *Communications of the IIMA*, 2008, *8*(3), pp. 25–32.

[4] Bateson, A. G., Alexander, R. A., & Murphy, M. D., "Cognitive processing differences between novice and expert computer programmers", in *International Journal of Man-Machine Studies*, 1987, *26*(6), pp. 649 – 660.

[5] Bauer, M. I., & Johnson-Laird, P. N., "How Diagrams Can Improve Reasoning" in *Psychological Science*, 1993, *4*(6), pp. 372–378.

[6] Berry, M., & Kolling, M., "Novis: A notional machine implementation for teaching introductory programming", in *Fourth International Conference on Learning and Teaching in Computing and Engineering (LATICE) 2016*, 2016, p. 4503.

[7] Biggs, J., & Collis, K. F., *Evaluating the Quality of Learning: The SOLO Taxonomy*. New York: Academic Press, 1982.

[8] Bloom, B. S., Englehard, M. D., Furst, E. J., Hill, W. H., & Krathwohl, D. R. (1956). Taxonomy of Educational Objectives: The Classification of Educational Goals: Handbook I Cognitive Domain. *New York*, *16*, 207. doi:10.1300/J104v03n01_03

[9] Caspersen, M. E., & Bennedsen, J., "Instructional design of a programming course: a learning theoretic approach." in *Proceedings of the Third International Workshop on Computing Education Research*, 2007, pp. 111–122.

[10] Cicchetti, D. V., "Guidelines, criteria, and rules of thumb for evaluating normed and standardized assessment instruments in psychology" in *Psychological Assessment*, 1994, *6*(4), pp. 284–290.

[11] Colaso, V., Kamal, A., & Saraiya, P., "Learning and retention in data structures: A comparison of visualisation, text, and combined methods." in *Proc. ED-MEDIA*, 2002, pp. 1–2.

[12] Crescenzi, P., Malizia, A., Verri, M. C., Díaz, P., & Aedo, I., "Integrating algorithm visualisation video into a first-year algorithm and data structure course." in *Educational Technology and Society*, 2012, *15*(2), pp. 115–124.

[13] Davies, S., "Work in progress - Analyzing the gap between diagrams and code in computer science.", in *Proceedings - Frontiers in Education Conference, FIE*, 2008, pp. 16–17.

[14] Duchowski, A. T., & Davis, T. A., "Teaching Algorithms and Data Structures through Graphics.", *Eurographics 2007*, 2007.

[15] Fleiss, J. L., "Measuring nominal scale agreement among many raters.", *Psychological Bulletin*, 1971.

[16] Gomes, A., & Mendes, A. J., "Learning to program-difficulties and solutions.", in *International Conference on Engineering Education–ICEE*, 2007.

[16] Green, T. R. G., & Petre, M., "Usability Analysis of Visual Programming Environments: A “Cognitive Dimensions” Framework.", *Journal of Visual Languages & Computing*, 1996, *7*(2), pp. 131–174.

[17] Hooshyar, D., Ahmad, R. B., Md Nasir, M. H. N., Shamshirband, S., & Horng, S.-J., "Flowchart-based programming environments for improving comprehension and problem-solving skill of novice programmers: A survey.", in *International Journal of Advanced Intelligence Paradigms*, 2015, *7*(1), pp. 24–56.

[18] Hundhausen, C. D., Douglas, S. A., & Stasko, J. T., "A Meta-Study of Algorithm Visualisation Effectiveness.", in *Journal of Visual Languages and Computing*, 2002, *13*(3), pp. 259–290.

[19] Hundhausen, C., & Douglas, S. A., "SALSA and ALVIS: A Language and System for Constructing and Presenting Low Fidelity Algorithm Visualisations.", *Software Visualisation*, 2002, pp. 227–240.

[20] Jain, J., Cross, J. H., Hendrix, T. D., & Barowski, L. A., "Experimental evaluation of animated-verifying object viewers for Java.", in *SoftVis '06 Proceedings of the 2006 ACM Symposium on Software Visualisation*, 2006, pp. 27.

[21] Jin, B., Jin, M., & Xue, X., "Algorithm animation and its applications in instruction.", in *2010 3rd IEEE International Conference on Ubi-Media Computing*, 2010, pp. 272–276.

[22] Karavirta, V., Korhonen, A., Malmi, L., & St, K., "MatrixPro – A Tool for On-The-Fly Demonstration of Data Structures and Algorithms.", in *System*, 2003, pp.26–33.

[23] Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M., "A study of the difficulties of novice programmers." in *ACM Sigcse Bulletin* , 2005, Vol. 37, No. 3, pp. 14-18.

[24] Lawrence, R., "Teaching Data Structures Using Competitive Games.", in *IEEE Transactions on Education*, 2004, *47*(4), pp. 459–466.

[25] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., Seppälä, O. *A Multi-National Study of Reading and Tracing Skills in Novice Programmers*., 2007.

[26] Macfarlane, K. N., & Mynatt, B. T., "A study of an advance organizer as a technique for teaching computer programming concepts.", in *SIGCSE Bull.*, 1988, *20*(1), pp. 240–243.

[27] McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C., "Knowledge organization and skill differences in computer programmers." in *Cognitive Psychology*, 1981, *13*(3), 307–325.

[28] Meisalo, V., Sutinen, E., & Tarhio, J., "{CLAP}: teaching data structures in a creative way.", in *ITiCSE '97: Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education*, 1997, pp. 117–119.

[29] Milne, I., & Rowe, G., "Difficulties in learning and teaching programming—views of students and tutors.", in *Education and Information technologies*, 2002, 7(1), pp. 55-66.

[30] Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M., "Visualizing programs with Jeliot 3.", in *Proceedings of the Working Conference on Advanced Visual Interfaces - AVI '04*, 2004, pp. 373.

[31] Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R., "Ability to “explain in plain english” linked to proficiency in computer-based programming.", in *Proceedings of the Ninth Annual International Conference on International Computing Education Research - ICER '12*, 2012, pp. 111.

[32] Myers, B., "Visual Programming, Programming by Example, and Program Visu lization: A Taxonomy.", in *Proceedings SIGCHI'86: Human Factors in Computing Systems*, 1986, pp. 59–66.

[33] Myers, B., "Taxonomies of Visual Programming and Program Visualisation.", in *Visual Languages and Computing*, *1*(1), 1990, pp. 97–123.

[34] Naps, Thomas L., Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, A. V., "Exploring the role of visualisation and engagement in computer science education.", in *ACM SIGCSE Bulletin*, 2002, *35*(2), pp. 131–152.

[35] Naps, T. L., "JHAVE: Supporting algorithm visualisation.", in *IEEE Computer Graphics and Applications*, 2005, *25*(5), pp. 49–55.

[36] Newell, A., & Shaw, J. C., "Programming the Logic Theory Machine. *Western Joint Computing Conference*, 1957.

[37] Patel, S., "A Literature Review on Tools for Learning Data Structures", 2014, pp. 1–7.

[38] Pathiania, U. S. A., "Visualisation Tools of Data Structures Algorithms – A Survey.", in *International Journal of Advanced Research in Computer Science and Software Engineering*, 2014, *4*(3), pp. 338–341.

[39] Petre, M., Blackwell, A. F., & Green, T. R. G., *Cognitive Questions in Software Visualisation*, 1996, pp. 1–20.

[40] Price, B. A., Baecker, R. M., & Small, I. S., "A Principled Taxonomy of Software Visualisation.", in *Journal of Visual Languages and Computing*, 1993, 4, pp. 211–266.

[41] Ragonis, N., & Ben-Ari, M., "On understanding the statics and dynamics of object-oriented programs.", in *SIGCSE '05 Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005, 37(1), pp. 226.

[42] Robins, A., Rountree, J., & Rountree, N., "Learning and teaching programming: A review and discussion.", in *Computer science education*, 2003, 13(2), pp. 137-172.

[43] Segura, C., Pita, I., Saiz, A. I., Soler, P., del Vado Vírseda, R., Saiz, A. I., & Soler, P., "Interactive Learning of Data Structures and Algorithmic Schemes.", in *Computational Science – ICCS 2008 SE - 85*, 2008, pp. 800–809.

[44] Stasko, J., Badre, A., & Lewis, C., "Do Algorithm Animations Assist Learning? An Empirical Study and Analysis.", in *Interchi 03*, 1993, *5*, pp. 61–66.

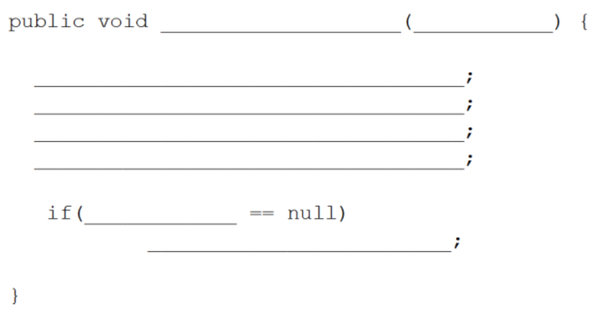
[45] Vessey, I., "Expert-novice knowledge organization: an empirical investigation using computer program recall.", in *Behaviour & Information Technology*, 1988, *7*(2), pp. 153–171.

[46] Weiser, M., & Shertz, J., "Programming problem representation in novice and expert programmers.", in *International Journal of Man-Machine Studies*, 1983, *19*, pp. 391–398.

[47] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K., & Prasad, C., "An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies.", in *Conferences in Research and Practice in Information Technology Series*, 2006, pp. 243–252.

APPENDIX

Question 1(c): Complete the following code for operationZ. Write the correct method name to replace operationZ.



```
public void ____________________(____________) {

  ____________________________________;
  ____________________________________;
  ____________________________________;
  ____________________________________;

   if(_____________ == null)
          __________________________;

}
```

Figure A: Fill-in-the-blanks task for Question 1(c)