ISSN 2614-0047

Effects of using problem-solving guide and explanatory support in program visualization tool on reducing students' misconceptions in learning data structure concepts

Adam B. Mtaho [a,1,*]

[a] Arusha Technical College, Arusha, Tanzania
[1] abasigie@yahoo.com
* corresponding author

ARTICLE INFO

Article history
Received August 9, 2023
Revised August 18, 2023
Accepted November 4, 2023

Keywords
Misconceptions
Program visualization tool
Data structures
Explanatory support
Problem solving guide

ABSTRACT

The tendency of novice programmers to hold misconceptions when learning data structures is one of the challenges that novice programmers face in computer science education. Holding misconceptions can result in students' demotivation and high failure rates in learning the subject. This article presents the findings of an experimental study that was conducted to examine effects of using problem solving guide and supportive explanations in program visualization tool on reducing students' misconceptions in learning data structure concept. The subject of the study 83 students pursuing a C122 data structures at the College of Informatics and Virtual Education of the University of Dodoma. The design chosen was a single factor between the experiment designs, with the number of errors committed by the students when writing programs as a dependent variable. The experimental group l were ins instructed by using CeliotM, while the control group received instruction using the conventional lecture method. Results show that the use of CeliotM significantly reduced student's misconceptions in learning data structures courses compared to the conventional lecture method. The study's important findings suggest that applying sufficient guidance and explanatory support within program visualization an effective teaching strategy for reducing students' misconceptions in learning data structures concepts.

1. Introduction

Data Structures and Algorithms (DSA) course is essential in Computer Science (CS) and engineering disciplines. DSA is an advanced level programming course that is mandatory for any CS student. It enables students to develop conceptual, strategic, problem-solving, and analytical thinking skills. To achieve this goal, the subject demands the learner's possession of a multitude of basic programming skills, such as knowledge of language syntax, program planning, problem solving, and algorithm analysis. However, learning data structures is very challenging and frustrating for novice programmers [1]. Due to its complexity, learning data structures causes students misconceptions [2]–[7]. Such misconceptions exhibit in using various data structures elements such as pointers [8]–[10], heap [11]–[13], recursions [14], [15], and linked lists [15], [16]. Misconceptions in learning DSA are attributed to the abstract nature of the course, high elementary activity, and dynamic nature of the data tructures concepts. The consequence of such misconceptions is high failure and dropout rates in computer science education.

The authors in [17] asserts that many misconceptions in learning to program occur because students do not see how program elements are executed inside the computer memory. Nevertheless, most conventional approaches such as lectures that are currently used in teaching programming do not

effectively engage students in dealing with aspects that are hidden within the execution-time world of the notional machine, leading to students misconceptions [18]. Studies have revealed that there exist different types of misconceptions that students tend to hold when learning DSA [1]–[6] [19]. Prior studies have shown that the teaching strategies that engage students towards achieving all four levels of learning programming i.e., syntactic, semantic, schematic, and strategic can help reduce students misconceptions in learning programming [20]. However, most strategies that are used in teaching data structures apply the use algorithm visualizations (AV). Such tools have less engaging features that guides student for successfully learning DSA concepts [21]–[24].

To address this study gap, this study delineates a novel strategy that utilizes a problem-solving guide and supportive explanations within the CeliotM program visualization tool as an attempt to reduce students' misconceptions in learning data structures among undergraduate students enrolled in the DSA course at the University of Dodoma. To the best of my knowledge, no study so far has examined the impact of using the proposed strategy in teaching DSA course. The remainder of the study is structured as follows: A literature review is presented in Section 2. Section 3 presents the proposed teaching strategy. Study methodology is discussed in section 4. The research findings and discussions are presented in section 5. The conclusion is presented in section 6.Literature review.

1.1. Misconceptions in Learning programming

The authors in [25] define the term "misconception" as an incorrect understanding of a concept or a set of concepts that leads to mistakes in writing or reading programs. According to [26], misconceptions occur when a programmer commits an error on a programming task. Misconceptions in learning programming dominate when the students lack syntactic, schematic, strategic, and conceptual knowledge [20], [27]. Syntactic knowledge is concerned with specific facts regarding a programming language and deals with the rules governing its use [4]. It just focuses on the program that will compile, not necessarily that it will provide a valid and viable solution to the problem [27]. A student who fails to use the correct syntax when writing a program is said to commit syntax errors. An example of a syntax error is a reference to undeclared variables, that is, writing cin>>x; without int x in C++. Schematic knowledge is defined as one's ability to recognize patterns in codes developed for previous problems, also known as programming plans, and apply them to form a solution for the current problem [20].

A student who fails to identify which method or what constructs to use to solve a given problem is said to commit schematic error. An example of a schematic error is using 'for loop' instead of nested for loop or using while loop instead of for loop. Strategic knowledge is defined as one's ability to solve programming problems that are more complex and beyond those that have been encountered before [20]. More specifically, strategic knowledge of programming refers to expert-level knowledge about planning, writing, and debugging programs for solving novel problems using syntactic and conceptual knowledge [27]. A student who fails to interpret the problem or question that he or she has been asked to solve is said to commit strategic errors. A strategic errors occurs when a programmer fails to include certain program component, e.g., the absence of a 'for loop' when this is an explicit requirement of the question/problem to be solved.

1.2. Causes of the misconceptions in Learning DSA course

DSA is reported as one of the most difficult curse to learn in CS education. Unlike introductory programming, the understanding of DSA depends on the learner's possession of a multitude of basic programming skills such as knowledge of problem-solving approach, program-planning, and syntax skills [28]. According to [28], due to the task complexity, there is a diverse range of misconceptions in learning data structures. For example, there are some misconceptions that students commit when trying to understand how different algorithms work [29] misconceptions on how to use such algorithms to manipulate data structures within the program [30] misconceptions in planning for a solution, and problems in debugging and tracing [28]. There are also misconceptions in understanding some of the fundamental programming concepts, such as nested loops, recursion, and linked lists, which are sources of difficulty [1]. There are also misconceptions in implementing pointers [10], recursion [15], [31], [32], linked lists, and heaps [33]. The misconceptions in learning DSA exist during the entire learning period, and they manifest themselves in the syntax, semantic, schematic, and strategic domains.

The difficulty of understanding the dynamic nature of algorithm states is another cause of a high misconception [30], [34] Such difficulty relies on the algorithm itself which is derived from the

dynamic step-by-step processes [34]. A dynamic view of a program brings together program codes, the state of the program, and the process that changes it, as well as the computer on which the program runs [if not the actual hardware, at least a notional machine [35]. Studies have also found that students who learn DSA have misconceptions in planning solutions. Even if they can plan, they end up developing an abstract plan that does not solve a given problem [28], [36]. Some students cannot form a plan from scratch; they try to wrongly use previously used plans to solve new problems. The authors in [37] claim that when learning programming, inexperienced programmers without strategic knowledge occasionally produce code that functions as intended in certain common but not all circumstances. In some circumstances, novices may become disoriented or completely fail to recognize where to begin. Some students could attempt to plan a potential course of action, but crucial components or interactions might be missed.

1.3. Empirical studies on students' misconceptions in learning DSA

Several studies have examined misconceptions in learning DSA. Some of these studies are those of [38]–[40], and [33]. The authors in [39] conducted the study to determine misconceptions held by students related to heaps and binary search trees. They found that some students showed passive knowledge of the formal definition of a heap. They failed to differentiate heaps with binary search trees. The authors in [40] also conducted the survey to examine students perceived level of difficulty in learning CS2 at West Coast University, USA. They found that students tend to confuse DSA concepts, even if they can conceptually state them, but in reality they can implement the same thing ´differently. Some students also have difficulties comprehending how some algorithms, such as merge sort and quick sort, work [40]. The author in [38] examined the types of errors mostly committed by students whom they called "advanced novice programmers".

Their studies focused on nested loops, arrays, and recursion. They found that even after being taught in the classes, such students fail to understand basic constructs such as nested loops and recursions, which they have studied in previous classes [38]. Also, [33] examined students' procedural comprehension of particular linked list operation for 249 students who were enrolled in Java CS2 course. Students performance results were as follows: 16% of those students failed to update the tail pointer, 12% incorrectly attached the new node, and 10% needlessly looped through the list to find the tail when asked to add a node to the end of a list. Additionally, they discovered that some students were misinformed about double-linked lists and thought that a double-linked list could be simultaneously searched in both directions.

Most students relied mainly on memorization of data structures principles and lacked the practical skills necessary to apply plan and implement the data structures programs. Studies show that teaching programming using program visualization (PV) tools that actively engage learners help reduce student's misconceptions in learning to program and hence improves students programing comprehension [41]–[46].

1.4. Visualization in learning programming

Visualization are concrete models that are used in various discipline mostly in science and engineering fields. They are used to provide a clear visual metaphor for explaining conception or operations that are taking place behind the "scene" from the viewers' point of views eye. More specifically, visualization are used to make visible actions that are taking place in the invisible world. In Computer science such visualizations are categorized as AV and PV). Both AV and PV tools have been used in teaching and learning programming. But they differ in terms of scope, design and use. According Bergin et al., [46]. PV tools focus on the graphical representation of an executing program and its data. In PVs, data, code and events of interest are visualized at the low level of abstraction. AVs on other hand show operations fundamental to an algorithm, as opposed to just code and data. Despite the significant differences between program and algorithm animation systems, both cases, each tool is being used to visualize the dynamic execution of a static description [46].

In early days of computer science education PVs were used to teach introductory programming concepts, while algorithm AVs were mainly used to teach data structures. However, studies have shown that the use of AVs for teaching DSA is not pedagogically effective since such teaching strategy has put much emphasis on conceptual knowledge [22], [47] , with less emphasis on strategic, syntactic and schematic skills, resulting in student's misconceptions and frustrations in writing programs [22], [48], [49]. For that matter, recent studies have recommended the use of PV tools instead of just AVs as a strategy to help students effectively understand the DSA course [22]. To ensure that PV help

students clearly understand data structure concepts earlier studies have recommended the designed PV tools to ensure that (i) they provide resources that help learners interpret the graphical representation. (ii) Provides supportive explanation that will help map a visualization to the underlying algorithm it is designed to represent (iii) Ensure that learners are provided dynamic feedback on their activities rather static ones (iv) Complement visualizations with explanations for better understanding [43], [46], [47]. This studiy aimed at examining the effects of using problem-solving guide and explanatory support within PV tool on reducing students' misconceptions in learning data structure concepts.

2. The Teaching Strategy

The proposed teaching strategy employs the use CeliotM PV tool [10]. CeliotM is a PV tool designed to help new programmers learn and understand data structures in C++ [10]. The tool functions as a compiler, AV, and PV. In contrast to pure animation software, CeliotM offers a comprehensive learning and programming environment, making it more enjoyable and suitable for novice programmers. In this section, we briefly describe three learners' engagement features in CeliotM that played crucial roles in this study. Such features are (i) problem solving guide, (ii) an informative error message support, (iii) system generated and user's defined explanations, and (iv) memory address explanations.

2.1. Problem-solving guide

Research reveals that novices cannot solve problems well unless they are well guided. However, according to [52], guiding students on how to effectively program has always been ignored by many instructors. In order to help students understand programming problems, interpret them, and implement problem solutions correctly, prior to attempting any programming problem, CeliotM provides a general guide for solving the problem-based exercises. Thus, the framework guides the instructors on how students should deal with programming questions or problems. The feature prompts automatically when the exercise feature of the tool is opened. Each instructor and student were supposed to use this framework. Fig. 1 shows a snapshot of the CeliotM with guidelines for solving programming problems.

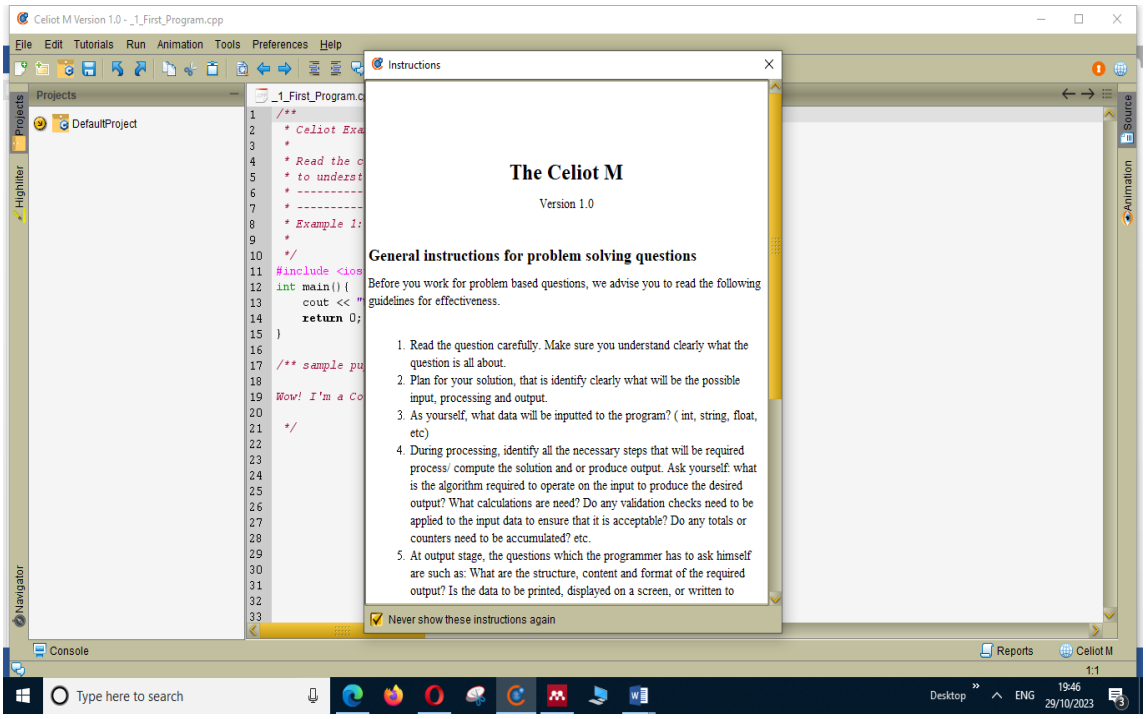

Fig. 1. Problem solving guide

Fig. 2 shows the general instructions for problem-solving questions.

General instructions for problem-solving questions

Before you work on problem-based questions, we advise you to read the following guidelines for effectiveness:

1. Read the question carefully. Make sure you understand clearly what the question is all about.
2. Plan for your solution, that is, identify clearly what the possible input, processing, and output will be.
3. Ask yourself, what data will be inputted into the program? (*int, string, float, etc*.)
4. During processing, identify all the necessary steps that will be required to process, compute the solution, or produce output. Ask yourself: What is the algorithm required to operate on the input to produce the desired output? What calculations are needed? Do any validation checks need to be applied to the input data to ensure that it is acceptable? Do any totals or counters need to be accumulated? etc.
5. At the output stage, the questions that the programmer has to ask himself are such as: What are the structures, content, and format of the required output? Is the data to be printed, displayed on a screen, or written to secondary storage? etc.
6. Identify the appropriate constructs (such as for loop, if, array, pointer, etc.) to be used in a given problem.
7. Adhere to the programming syntax and semantic rules.
8. Include all necessary header files and libraries (if any) that will be used in the program.
9. Test your program for both syntax, semantic, and logical errors.
10. Document your program. Include static comments and dynamic comments (user explanations for CeliotM) if required.
11. By using the C++ compiler, compile any program the written program.
12. Using visualization to compile and visualize the program.

Fig. 2. General instructions for problem-solving questions

According to the above guide, students were supposed to follow this guide when attempting to solve a problem. For example, given the question: Write a program that prompts users to write their first name, middle name, last name, age, and salary, and then returns the output in the ascending order of the salary. Given the problem that demands the use of data structures and algorithms, the students were guided step by step by using this guide.

They were guided on how to interpret the question, prepare the algorithm and pseudocode, and then write the code, run the animation where necessary to get a full view of the programming logic, and debug the program. It was hypothesized that a student who uses problem-solving guidance along with supportive explanations in CeliotM will improve his or her programming writing skills and hence commit fewer errors in learning DSA courses.

2.2. Informative error message support

Unclear messages that the compiler provides to the users are sources of confusion for many novice programmers [53], [54]. CeliotM has managed to address this problem by introducing an informative error message. The informative error message feature enables users to automatically identify the line where a syntactical or semantic error has occurred. Fig. 3 illustrates how CeliotM clearly shows a syntax error at line 19, which has resulted after compiling the program with a missing semicolon (;) at the end of the code statement.

The message in the error panel reads: *Syntax error, at line 19, column 1, and a semicolon is missing in the return statement*. Thus, unlike the traditional compiler, which returns several error messages for just one syntax error, this feature reports just one error, while indicating the exact location where an error has occurred.

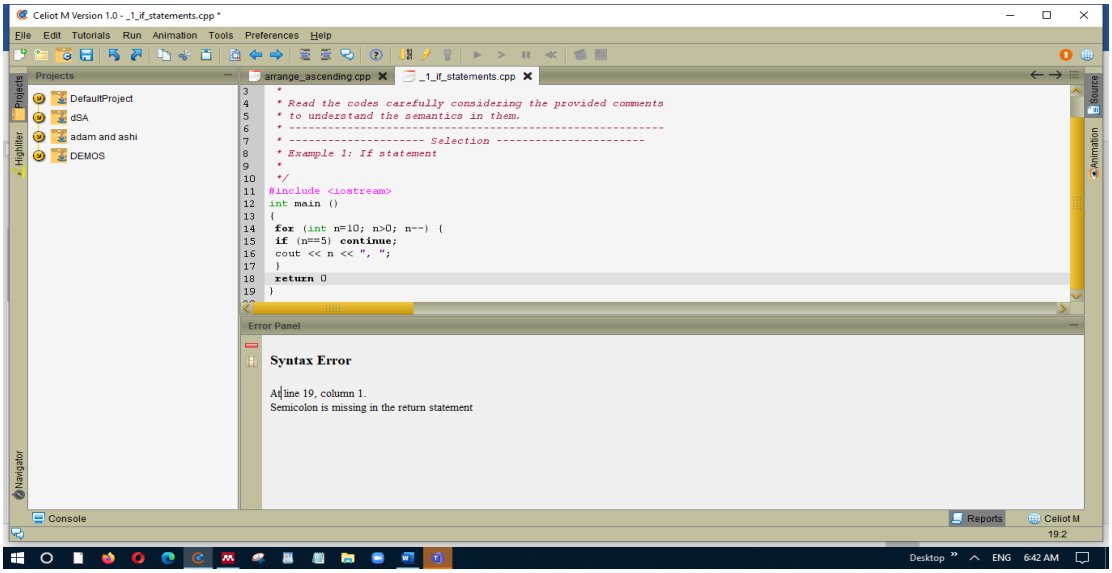

Fig. 3. The informative error message in CeliotM (Syntax error message)

Fig. 4 1also shows informative error message describing the occurrence of the semantic error in the program. The message in the error panel indicates the type of semantic error that has occurred, the exact position in the program (line and column number) where it has happened, and the reason why such an error has happened.

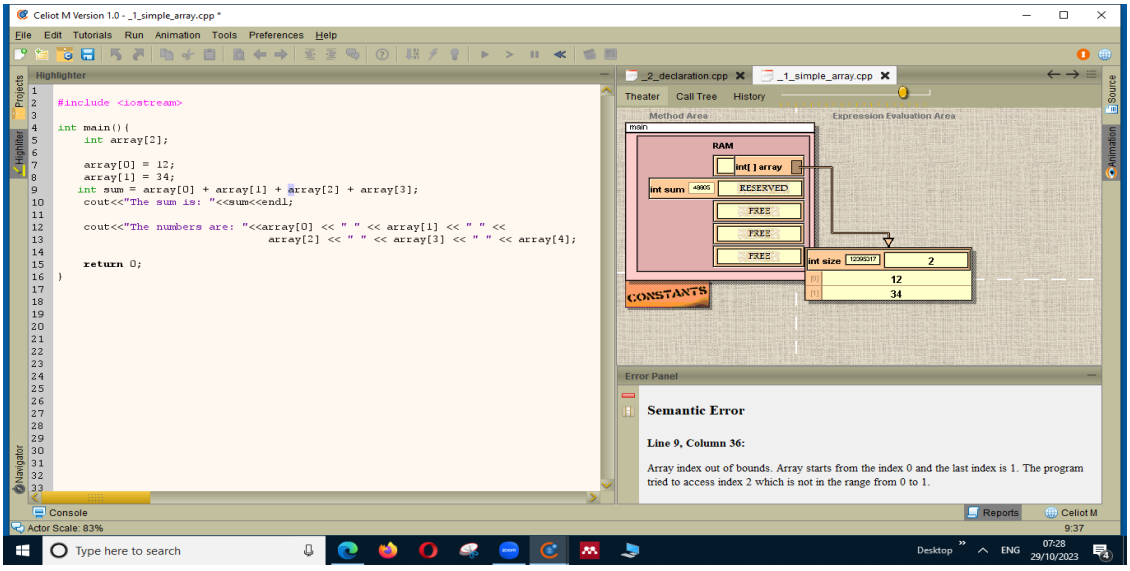

Fig. 4. The informative error message in CeliotM (Semantic error message)

It was expected that the provision of informative error messages in CeliotM would help build students' syntactic and semantic knowledge and enhance students' debugging skills.

2.3. System and user-defined explanations

Lack of conceptual understanding of how the program executes remains the biggest challenge for novices. It also leads to the inability of the students to imagine what happens when lines of code execute, thus affecting their ability to write correct programs. For this reason, the system and user-defined explanation features were provided within CeliotM. By using this feature, learners can visually see and interpret what happens when the program is executed. While system explanations are automatically generated by the system, user-defined explanations are user-generated. Thus, in this tool, when the system-defined explanation button is turned on, the learner can actively see and learn what is going on when each construct in the code is executed. Fig. 4 shows a system explanation

describing when variable assignment operations occur during program execution. As shown in Fig. 5, when line 14 is executed by the machine, a message "(int) 10 is assigned to a variable of type int" will appear to help the user actively understand the program execution.

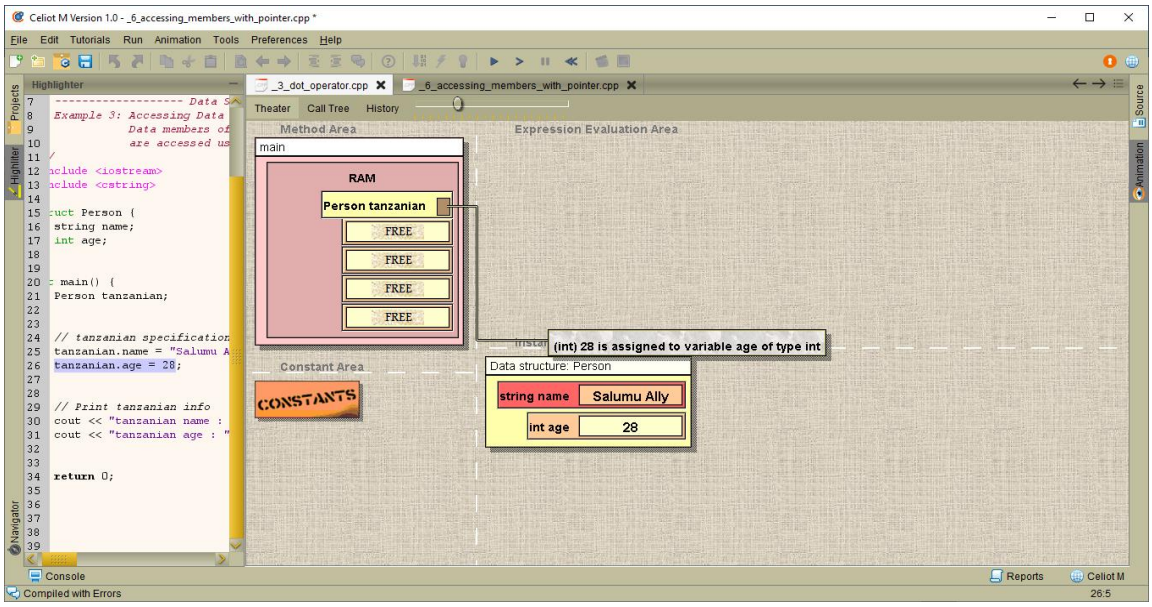

Fig. 5. System defined explanation.

Fig. 6 shows CeliotM with the user-defined explanation feature enabled. A user-defined explanation feature allows the user to write comments to explain what the code does. In this example, the instructor has written a comment on line 14 of the program that states that: "This line declares the array object that can store five integers". This comment is not visible in the code view but will appear in the animation view when line 14 is executed. Through the use of such explanatory features, a learner is helped to map and interpret how a given program statement or algorithm executes in the notional machine.

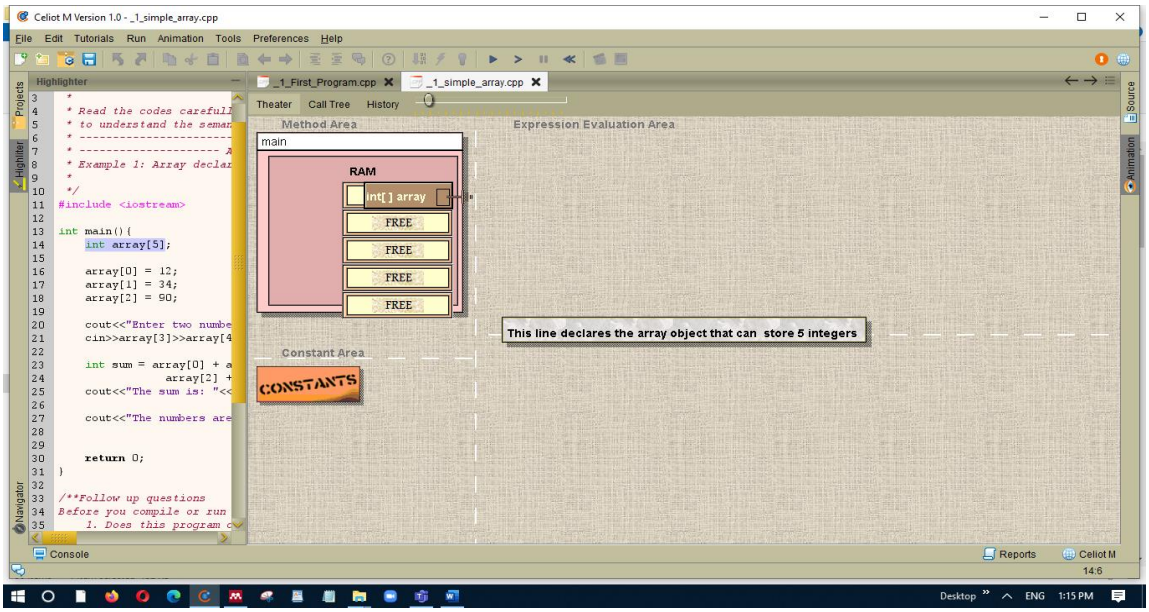

Fig. 6. User- defined explanation

It was hypothesized that including the system and use-defined explanation inside the tool and allowing instructors to apply such features when teaching the students would help increase students schematic and strategic skills in learning data structures, and hence reduce students misconceptions in planning and writing programs.

2.4. Memory address explanation

The function of memory addressing is to find the definite memory location that is used when the program is executed in the computer. This feature is an integral part of all pointer operations performed in any C++ program. Without it, pointer operations are not possible. Fig. 7 show that apart from viewing state changes and algorithm execution, CeliotM enable users to view both memory occupied as well as address. It is expected the student who view dynamically how memory address are assigned and referenced will clearly understand the logic behind the working behind pointers, linked list and other data structures operation that involve manipulation of memory address.

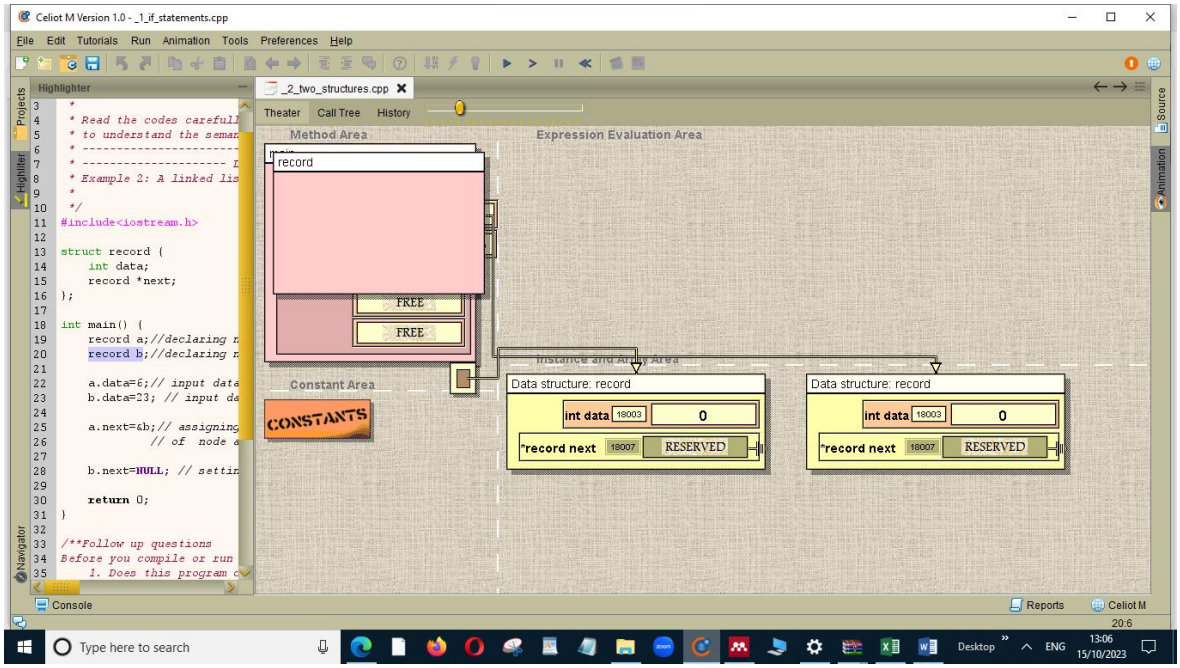

Fig. 7. Memory address explanation

2.5. Support of Data Structures Visualization

CeliotM supports all linear data structures such as linked lists, queues, and stacks. Fig. 8 shows the simple C++ linked list structure program. The source code view contains the source codes, while the animation view supports animations. The source code may also be compiled using the C++ compiler.

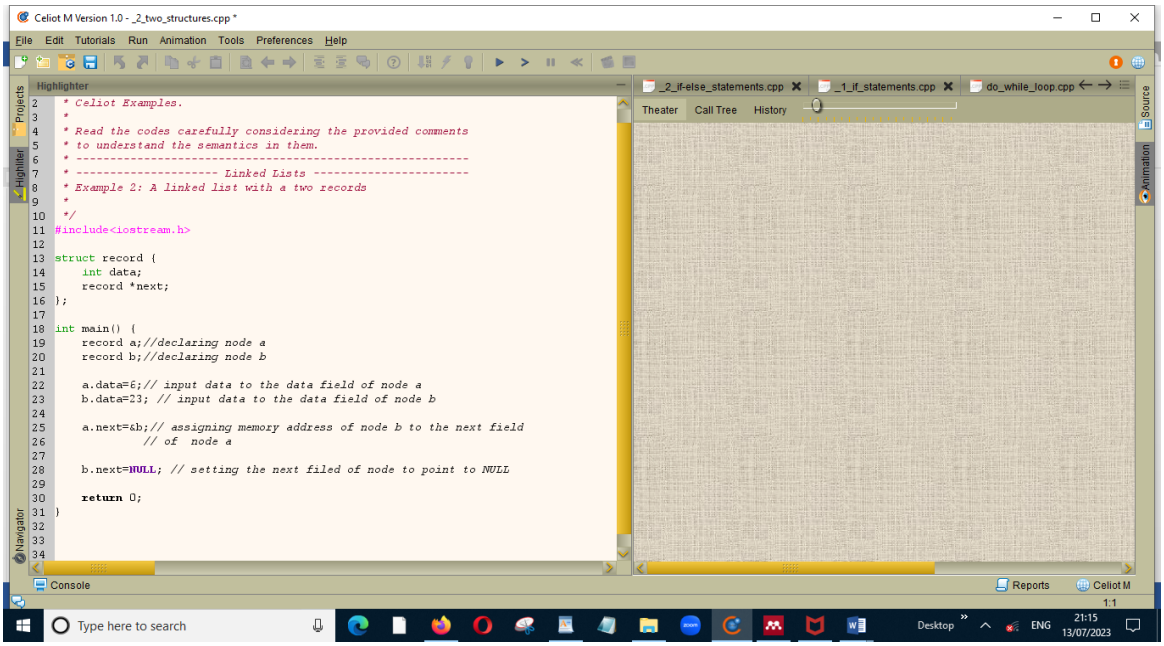

Fig. 8. Simple linked program before animation in CeliotM

Fig. 9 shows the animation of the simple liked list program at the instance when line 14 is executed.

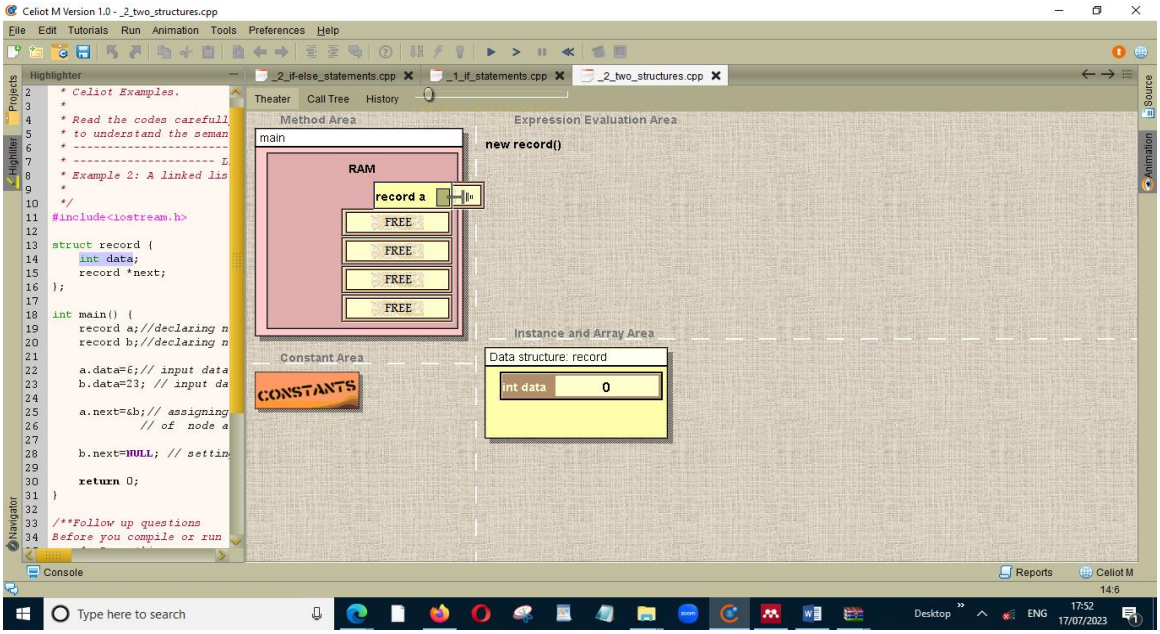

Fig. 9. Animation state of a linked list program during the execution of line 14

By using the tool, a user can view how the linked list works step by step, from node creation to node update, insertion, and deletion. The tool displays how the memory space is utilized and variable addresses are assigned in computer memory during program execution, thus helping the learner understand clearly how the pointer works in a linked list program. The tool shows the animation of a simple linked list program. It visually displays the entire execution cycle, starting from how data is input to the data fields of nodes a and b, how the memory address of node b is assigned to the next field, how node a is assigned to the next field, and how node a is set to NULL. Fig. 10 shows the final program animation state when the linked list program is fully executed.

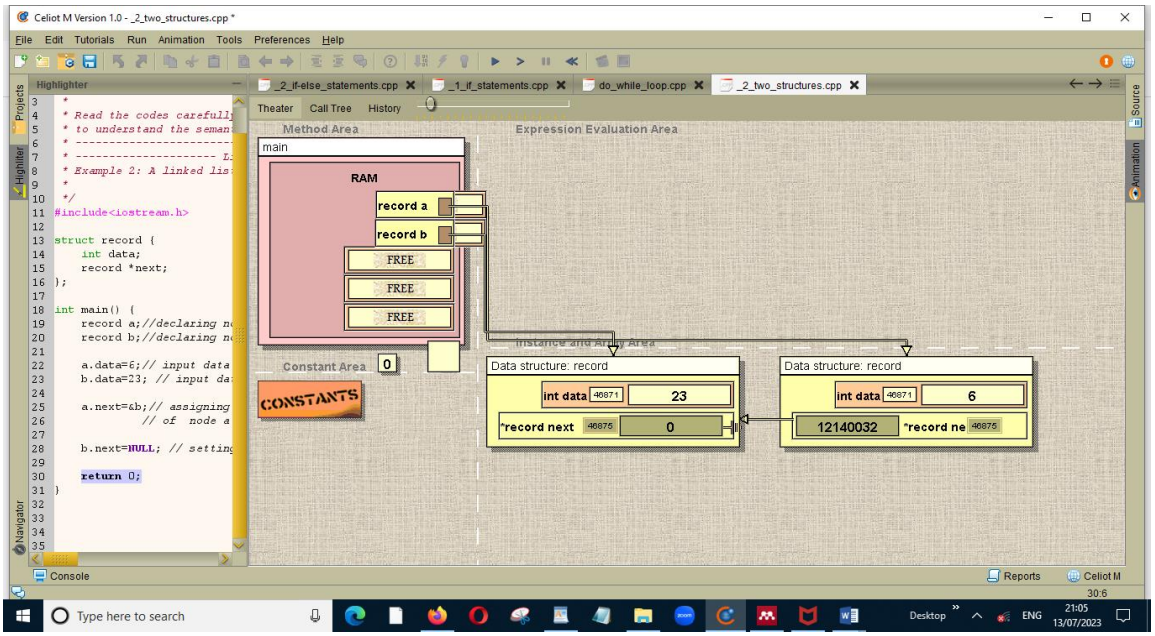

Fig. 10. Final program animation state in CeliotM

Through the use of CeloitM learners were able to plan, write, compile, visualize and debug various data structures concepts. The inclusion of exercises and examples in the CeliotM was expected to help minimize misconceptions in learning data structures, hence, improve students' programming comprehension.

3. Method

3.1. Study objective

The aim of this study was therefore to test if the use of CeliotM along with a problem-solving guide, an informative error message, a system, and user-defined explanations would reduce students' misconceptions (syntax, semantic, schematic, and strategic errors) and hence improve students' comprehension in learning data structure concepts compared to the traditional lecture method.

3.2. Research design

In this study, an experiment-research design was used. The experiment was conducted at the College of Informatics and Virtual Education of the University of Dodoma, in 2019. The experiment was set up as a single factor, with a number of errors the students made when coding programs as the dependent variable. The two independent variables were (i) teaching and learning data structures using a conventional approach; and (ii) teaching and learning data structures using a conventional lecture method combined with the CeliotM framework. The number of error committed by students in writing computer program was a dependent variable. The dependent variable was measured by counting the frequencies of each error committed by students and computing the average of its occurrence.

3.3. Participants

The target population was the undergraduate students studying DSA course at the College of informatics and virtual Education of the University of Dodoma. A total of eighty-three (83) students' examination CS 122 scripts served as the sample for the experiment. These scripts were chosen at random from the 2017/2018 and 2018/2019 exam scripts. A total of 43 exam scripts drawn at random from 972 exam script of 2019/2020 academic year constituted the experimental group, while the 40 scripts drawn at random from 894 exam scripts of academic year 2017/2018 constituted the control group. The proposed strategy together with lecture method was used to teach the students the CS 122 course during the second semester of the academic year 2019/2020 (the experimental group), while the traditional lecture method was used to teach the same course to the students in the academic year 2017/2018 (control group). Arrays, structures, pointers, linked lists, queues, stacks, trees, graphs, sorting algorithms, and searching algorithmic forms were all covered in the course syllabus. The course took 52 hours to complete, of which 26 were spent in lectures and 26 in labs. All of these students received the same instruction from the same instructors and were evaluated using the same evaluation standards.

3.4. Hypotheses to be validated

To validate if using a problem-solving guide and explanatory support in a PV tool would help reduce students' misconceptions in learning data structure concepts and hence improve students' comprehension of the learning data structure course compared to the traditional lecture method or not, the following conjectures were formulated: (i) The null hypothesis, which states that the mean number of errors committed by students from the two groups is equal: Hp0): $\mu control = \mu experimental$, implying that there is no statistical difference in the number of errors committed between the two groups.(ii). The alternative hypothesis states that the mean number of errors committed by students from all groups is not equal $\mu control \neq \mu experimental$, implying that there exists a significant statistical difference in the number of errors committed between the two groups. In this study, it was predicted that students who were instructed data structures using CeliotM (the experimental group) would commit fewer errors compared to the conventional lecture method (the control group.

3.5. Tools and Materials

The tools used were Borland C++ Compiler V.5.5 and CeliotM, while the materials used were examination scripts, an error protocol, a C++ textbook, and a user manual. All practical work and demonstrations involved lectures with visualizations. The manual covered the topics of arrays, structures, pointers, linked lists, queues, stacks, trees, graphs, sorting algorithms, and searching algorithms.

3.6. Error protocol

In order to In order to create an error protocol, it was necessary to review the typical programming mistakes, issues, and misunderstandings that novice programmers consistently commit when learning data structures [33], [38], [55]–[58]. Based on the early studies of [59] and [20].The protocol was

divided into seventeen (17) sorts of mistakes (misconceptions), which were then grouped into four categories of syntactic, semantic, schematic, and strategic errors.

3.7. Examinations settings and marking

The CS122 exams were created by instructors who taught the same course from 2017/2019. The examination panels designed the tests and the grading criteria. The exam and marking scheme were subsequently examined by external examiners to make sure they complied with both the curriculum and examination criteria. The same group of examiners marked the test scripts. When the findings were complete and available, the researchers randomly chose 43 examination scripts from the 2018/2019 scripts and compared them to 40 examination scripts from the 2017/2018 scripts. For the sake of the study, only errors committed in linked-list and quick-sort questions were examined in both the 2017/2018 and 2018/2019 exams. The questions tested the same level of programming cognition. The students' performance between the experimental and control groups was then considered for analysis.

3.8. Procedure

The study was carried out using lectures, tutorials, and laboratory work. Each week comprised 8 hours of teaching divided into 4 hours of lectures, 2 hours of tutorials, and 2 hours of laboratory sessions. The students were allowed to practice for 4 hours per week, that is, 2 hours of tutorials and 2 hours of supervised lab sessions. All students in participated the study. For each question provided students were guided as directed by the problem solving guide. From defining the problem, determining the algorithm, write the program, visualizing, debugging. Each program was written in CeliotM editor. The students were writing codes and visualizing them step by step, observing both animation and explanations per week. Students and visualized different programs and examine how program execution take place, debug and trace the program. After learning these topics for 52 weeks, students took the end-of-semester examination to verify if the use of the proposed approach had a positive impact on their cognition. After the end-of-semester examinations, the data pertaining to final examination performance was collected for analysis. The number of errors committed by the students in the experimental group was compared with those committed by the control group. The comparison of the errors was done by comparing the four most common errors that novices commit as defined by [59] and [20].

3.9. Data Analysis

The study employed quantitative data analysis methods. An independent t-test was used to determine if there exists a significant difference in the mean number of errors committed by students between the two groups. Descriptive statistics (percentages) was used to compare the number of errors committed between control and experimental group. The results of the study were presented using tables and figures.

4. Results and Discussion

Table 1 shows that the experimental group made an average of 11.41 errors, whereas the control group made an average of 36.65% errors, with standard deviations of 5.767 and 10.295, respectively. The results from the independent samples' test in Table 2 show that the significance value (p) is 0.0001. Since the p-value of 0.000 is less than 0.05, the alternative hypothesis is accepted, and hence there is a significant difference, as observed in the mean CS122 performance, between the number of errors committed between the control and the experimental groups. This implies that the number of errors committed by the students in the academic year 2019–2020 for the CS 122 examination (when the CeliotM framework was used) was less than those committed by students in 2017–2018 (when the traditional approach was used).

Table.1 Descriptive Statistics for errors committed between control and experimental group

	Treatment	N	Mean	Std. Deviation	Std. Error Mean
SCORE	Control	17	36.65	10.295	2.497
	Experimental	17	11.41	5.767	1.399

Table.2 Independent samples' test results

		Levene's Test for Equality of Variances		t-test for Equality of Means						
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
									Lower	Upper
SCORE	Equal variances assumed	1.494	.231	8.817	32	.000	25.235	2.862	19.406	31.065
	Equal variances not assumed			8.817	25.141	.000	25.235	2.862	19.343	31.128

These results confirm the hypothesis that students who used the CeliotM framework to learn data structures improved their comprehension as compared to those who used the conventional lecture method. As shown in Table 3, the total number of syntax, strategic, semantic, and schematic errors made in the academic year 2017-2018 was 363 (in the control group), while those made in 2017/2019 (in the experimental group) were 207. Table 3 also shows the percentage of errors reduced after using the CeliotM framework as follows: Semantic errors (75.0%), strategic errors (72.2%), schematic errors (7.3%), and syntax errors (67.2%). This implies that the use of the CeliotM framework was more helpful in reducing students' misconceptions when learning data structures as compared to the traditional method.

Table.3 Comparison of number of errors committed between control and experimental group

	Control Group	Experimental Group	% of Error Reduced
Syntax errors	119	39	67.2%
Strategic errors	108	30	72.2%
Semantic errors	28	7	75.0%
Schematic errors	108	31	71.3%
TOTAL	363	107	70.5%

Table 3 compares the total number of errors committed by the experimental group (2018/2019) versus those committed by the students in the control group (2017/2018). As shown in Table 4, the number of errors committed decreased after using CeliotM. The strategic error decreased by 72.2%, while the semantic error decreased by 75.0%. This implies that the students who used CeliotM experienced fewer misconceptions and hence deeply comprehended the data structures as compared to those in the traditional approach (2017/2018). The types of misconceptions that were committed with a frequency above 59% in linked were those related to failure to correctly update the head pointer, failing to update the next pointer, not handling an empty list properly, forgetting to update the list size, and mixing node references or data types. For the case of the merge sort, the following misconceptions were committed with a frequency up to 55%: failure to formulate the correct base case for recursion, not passing the correct array size, not updating indices properly, wrongly calculating the middle element calculation, and mixing up the order of recursive calls. These results concur with previous studies that learning data structure concepts is difficult, and thus the majority of the students who learn data structures commit misconceptions in learning the course [33], [60].

The authors in [20] and contend that novice programmers usually commit four types of programming errors as they try to learn to program: failure to interpret the problem or question that they have been asked to solve (strategic errors); failure to identify which method or what constructs

to use to solve a given problem (schematic); failure to determine the contribution of that programming construct to the solution (i.e., inability to know how the compiler interprets the code); and failure to follow the programming language rules (syntax or compiler errors). Therefore, novices require support at all four levels of programming, across program and problem formulation. Findings from this study have shown that the use of the CeliotM framework has managed to improve all types of errors. These results support previous studies that show that as you extend the capabilities of the learning tool to engage them, learners' cognitive resources become fully utilized, thus improving programming comprehension [61]. In general, the inclusion of new learner engagement features in CeliotM helped to enhance the comprehension of data structures and hence minimize learners' misconceptions in learning the data structure course.

5. Conclusion

This study aimed at examining the impact of using CeliotM on reducing students' misconceptions about data structures. More specifically, the study examined the effect of using a CeliotM learning environment on reducing syntax, semantic, schematic, and strategic errors. The tool's system- and user-defined explanations, informative error messages, and problem-solving guides all make the framework more engaging. The CeliotM integrated working environment makes it work as a compiler, visualizer, and practice working platform. The impact of using the CeliotM has been evaluated through experiments. Results from the experiment have shown that the use of this framework has significantly reduced students' misconceptions about learning data structures and hence improved programming comprehension. The positive results revealed in this study promise that further improvement, if done within this framework, can bring better results. Currently, the platform supports only text narration. Future research should consider transforming systems and user-defined text emanations into audio to enhance the understanding ability of animations. The tool also currently supports the C++ language. Extending it to support Python and C programming languages could be done to extend the applicability of the platform. The errors protocol used focused on the four main categories of errors; future studies should consider extending the range of errors protocols to include a wide range that is more specific when implementing data structures such as tree, graphs, and heaps.

References

[1] P. Perera, G. Tennakoon, S. Ahangama, R. Panditharathna, and B. Chathuranga, "A Systematic Mapping of Introductory Programming Languages for Novice Learners," *IEEE Access*, vol. 9, pp. 88121–88136, 2021, doi: 10.1109/ACCESS.2021.3089560.

[2] D. McCall and M. Kölling, "A New Look at Novice Programmer Errors," *ACM Trans. Comput. Educ.*, vol. 19, no. 4, pp. 1–30, Dec. 2019, doi: 10.1145/3335814.

[3] C. M. Kandemir, F. Kalelioğlu, and Y. Gülbahar, "Pedagogy of teaching introductory text-based programming in terms of computational thinking concepts and practices," *Comput. Appl. Eng. Educ.*, vol. 29, no. 1, pp. 29–45, Jan. 2021, doi: 10.1002/cae.22374.

[4] K. Kwon, "Novice programmer ' s misconception of programming reflected on problem-solving plans," vol. 1, no. 4, 2017, doi: 10.21585/ijcses.v1i4.19.

[5] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun. 2020, pp. 691–701, doi: 10.1145/3377811.3380352.

[6] R. Benatti, T. Aparecida, R. Azevedo, G. Gama, and T. Caldas, "An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in C," pp. 1-44, 2017. [Online]. Available at: https://ic.unicamp.br/~reltech/2017/17-15.pdf.

[7] A. L. C. Barczak, A. Mathrani, B. Han, and N. H. Reyes, "Automated assessment system for programming courses: a case study for teaching data structures and algorithms," *Educ. Technol. Res. Dev.*, pp. 1–24, Aug. 2023, doi: 10.1007/s11423-023-10277-2.

[8] M. Heinsen Egan and C. McDonald, "An evaluation of SeeC: a tool designed to assist novice C programmers with program understanding and debugging," *Comput. Sci. Educ.*, vol. 31, no. 3, pp. 340–373, Jul. 2021, doi: 10.1080/08993408.2020.1777034.

[9] N. Burow *et al.*, "Control-Flow Integrity," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–33, Jan. 2018, doi: 10.1145/3054924.

[10] M. Adam Basigie, "The Impact Of Combining Follow-Up Questions And Worked Examples In Program Visualization Tool On Improving Students' Held Mental Models Of Pointers' Value And Address Assignment," *Educ. Pedagog. J.*, no. 2(4), pp. 53–64, Dec. 2022, doi: 10.23951/2782-2575-2022-2-53-64.

[11] S. Su, E. Zhang, P. Denny, and N. Giacaman, "A Game-Based Approach for Teaching Algorithms and Data Structures using Visualizations," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, Mar. 2021, pp. 1128–1134, doi: 10.1145/3408877.3432520.

[12] C. O'Farrelly, A. Booth, M. Tatlow-Golden, and B. Barker, "Reconstructing readiness: Young children's priorities for their early school adjustment," *Early Child. Res. Q.*, vol. 50, pp. 3–16, 2020, doi: 10.1016/j.ecresq.2018.12.001.

[13] C. Izu *et al.*, "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*, Dec. 2019, pp. 27–52, doi: 10.1145/3344429.3372501.

[14] T. Scholtz and I. Sanders, "Mental Models of Recursion : Investigating Students ' Understanding of Recursion," pp. 103–107, 2010, doi: 10.1145/1822090.1822120.

[15] R. Mccauley, S. Grissom, S. Fitzgerald, and L. Murphy, "Teaching and learning recursive programming : a review of the research literature," *Comput. Sci. Educ.*, vol. 3408, no. May, pp. 1–30, 2015, doi: 10.1080/08993408.2015.1033205.

[16] E. Almadhoun and J. Parham-Mocello, "Identifying Student Misunderstandings About Singly Linked Lists in the C Programming Language," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2021, vol. 2010-Octob, pp. 1–9, doi: 10.1109/VL/HCC51201.2021.9576162.

[17] J. Sorva, "Notional Machines and Introductory Programming Education," vol. 13, no. 2, 2013, doi: 10.1145/2483710.2483713.

[18] T. Kohn and D. Komm, "Teaching Programming and Algorithmic Complexity with Tangible Machines," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11169 LNCS, Springer Verlag, 2018, pp. 68–83, doi: 10.1007/978-3-030-02750-6_6.

[19] E. Vrachnos and A. Jimoyiannis, "Secondary education students' difficulties in algorithmic problems with arrays: An analysis using the SOLO taxonomy," *Themes Sci. Technol. Educ.*, vol. 10, no. 1, pp. 31–52, Dec. 2017. [Online]. Available at: http://earthlab.uoi.gr/.

[20] N. J. Coull, "SNOOPIE : Development of a Learning Support Tool for Novice Programmers within a Conceptual Framework," University of St Andrews, p. 237, 2008. [Online]. Available at: https://research-repository.st-andrews.ac.uk/handle/10023/522?show=full.

[21] K. Shinohara, N. Jacobo, W. Pratt, and J. O. Wobbrock, "Design for Social Accessibility Method Cards," *ACM Trans. Access. Comput.*, vol. 12, no. 4, pp. 1–33, Dec. 2019, doi: 10.1145/3369903.

[22] K. Romanowska, G. Singh, M. A. A. Dewan, and F. Lin, "Towards Developing an Effective Algorithm Visualization Tool for Online Learning," in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Oct. 2018, no. March 2020, pp. 2011–2016, doi: 10.1109/SmartWorld.2018.00336.

[23] A. Spanier, S. W. Harms, and J. Hastings, "A Classification Scheme for Gamification in Computer Science Education: Discovery of Foundational Gamification Genres in Data Structures Courses," in *2021 IEEE Frontiers in Education Conference (FIE)*, Oct. 2021, vol. 2021-Octob, pp. 1–9, doi: 10.1109/FIE49875.2021.9637447.

[24] T. J. Burns, S. C. Rios, T. K. Jordan, Q. Gu, and T. Underwood, "Analysis and Exercises for Engaging Beginners in Online {CTF} Competitions for Security Education." p. 9, 2017. [Online]. Available at: https://www.usenix.org/conference/ase17/workshop-program/presentation/burns.

[25] W. W. Yang, Shi, Krupal, Shah, "Toward Semi-Automatic Misconception Discovery Using Code, " *Association for Computing Machinery* vol. 1, no. 1, p. 7 2021, doi: 10.1145/3448139.3448205.

[26] L. Gusukuma *et al.*, "Misconception-Driven Feedback : Results from an Experimental Study," no. 1, pp. 160–168, 2018, doi: 10.1145/3230977.3231002.

[27] T. J. McGill and S. E. Volet, "A conceptual framework for analyzing students' knowledge of programming," *J. Res. Comput. Educ.*, vol. 29, no. 3, pp. 276–297, 1997, doi: 10.1080/08886504.1997.10782199.

[28] Y. Qian and J. Lehman, "Students' Misconceptions and Other Difficulties in Introductory Programming," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, pp. 1–24, Mar. 2018, doi: 10.1145/3077618.

[29] E. Fouh, M. Akbar, C. A. Shaffer, and V. Tech, "The Role of Visualization in Computer Science Education," pp. 95–117, 2012, doi : 10.1080/07380569.2012.651422.

[30] A. A. Supli, "Critical Analysis on Algorithm Visualization Study Critical Analysis on Algorithm Visualization Study," no. October, pp. 18–22, 2016, doi: 10.5120/ijca2016911633.

[31] S. Hamouda, S. H. Edwards, H. G. Elmongui, J. V Ernst, and C. A. Shaffer, "A basic recursion concept inventory," *Comput. Sci. Educ.*, vol. 27, no. 2, pp. 121–148, 2017, doi : 10.1080/08993408.2017.1414728.

[32] L. Porter, D. Zingaro, C. Lee, C. Taylor, K. C. Webb, and M. Clancy, "Developing Course-Level Learning Goals for Basic Data Structures in CS2," pp. 858–863, 2018, doi : 10.1145/3159450.3159457.

[33] D. Zingaro, C. Taylor, L. Porter, M. Clancy, C. Lee, and K. C. Webb, "Identifying Student Difficulties with Basic Data Structures," no. 169, pp. 169–177, 2018, doi : 10.1145/3230977.3231005.

[34] E. Fouh *et al.*, "Investigating Difficult Topics in a Data Structures Course Using Item Response Theory and Logged Data Analysis ∗," pp. 370–375. [Online]. Available at : https://eric.ed.gov/?id=ED592711.

[35] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory," vol. 13, no. 4, 2013, doi : 10.1145/2490822.

[36] J. C. Spohrer, E. Soloway, and E. Pope, "Where The Bugs Are," no. April, pp. 47–53, 1985, doi : 10.1145/1165385.317465.

[37] J. C. Spohrer and E. Soloway, "Simulating Student Programmers.," in *IJCAI*, 1989, vol. 89, pp. 543–549. [Online]. Available at : https://www.ijcai.org/Proceedings/89-1/Papers/087.pdf.

[38] G. Yarmish and D. Kopec, "Revisiting Novice Programmer Errors," vol. 39, no. 2, pp. 131–137, 2007, doi : 10.1145/1272848.1272896.

[39] H. Danielsiek, W. Paul, and J. Vahrenhold, "Detecting and understanding students' misconceptions related to algorithms and data structures," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 2012, pp. 21–26, doi : 10.1145/2157136.2157148.

[40] A. Decker and D. Simkins, "Uncovering Difficulties in Learning for the Intermediate Programmer," *2016 IEEE Frontiers in Education Conference (FIE)*, pp. 1-8, 2016, doi : 10.1109/FIE.2016.7757446.

[41] A. Moreno, E. Sutinen, and C. Islas Sedano, "A game concept using conflictive animations for learning programming," in *2013 IEEE International Games Innovation Conference (IGIC)*, Sep. 2013, pp. 175–178, doi: 10.1109/IGIC.2013.6659161.

[42] M. H. Egan and C. Mcdonald, "Program visualization and explanation for novice C programmers," in *Proceedings of the Sixteenth Australasian Computing Education Conference (ACE2014)*, 2014, pp. 51–57. [Online]. Available at : https://dl.acm.org/doi/pdf/10.5555/2667490.2667496.

[43] M. J. Laakso, T. Rajala, E. Kaila, and T. Salakoski, "The impact of prior experience in using a visualization tool on learning to program," *IADIS Int. Conf. Cogn. Explor. Learn. Digit. Age, CELDA 2008*, no. January, pp. 129–136, 2008, [Online]. Available at: https://www.researchgate.net/profile/Teemu-Rajala/publication/31597980_.

[44] T. L. Naps *et al.*, "Exploring the role of visualization and engagement in computer science education," *ACM SIGCSE Bull.*, vol. 35, no. 2, pp. 131–152, Jun. 2003, doi: 10.1145/782941.782998.

[45] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Trans. Comput. Educ.*, vol. 13, no. 4, 2013, doi: 10.1145/2490822.

[46] L. J. Mselle and H. Twaakyondo, "The impact of Memory Transfer Language (MTL) on reducing misconceptions in teaching programming to novices," *Int. J. Mach. Learn. Appl.*, 2012, doi: 10.4102/ijmla.v1i1.3.

[47] P. Bellstrom and C. Thoren, "Learning how to program through visualization: A pilot study on the bubble sort algorithm," in *2009 Second International Conference on the Applications of Digital Information and Web Technologies*, 2009, pp. 90–94, doi : 10.1109/ICADIWT.2009.5273943.

[48] L. Mselle and F. Ishengoma, "Memory transfer language as a tool for visualization-based-pedagogy," *Educ. Inf. Technol.*, vol. 27, no. 9, pp. 13089–13112, Nov. 2022, doi: 10.1007/s10639-022-11165-7.

[49] T. Naps *et al.*, "Evaluating the Educational Impact of Visualization," pp. 124-136, 2003, doi : 10.1145/960875.960540.

[50] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *J. Vis. Lang. Comput.*, vol. 13, no. 3, pp. 259–290, 2002, doi: 10.1006/jvlc.2002.0237.

[51] J. Urquiza-fuentes, V. E. L. Azquez-iturbide, U. Rey, and J. Carlos, "A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems ∗," vol. 9, no. June, 2009, doi : 10.1145/1538234.1538236.

[52] E. De Corte, L. Verschaffel, and H. Schrooten, "Cognitive effects of learning to program in Logo: A one-year study with sixth graders," in *Computer-based learning environments and problem solving*, 1992, pp. 207–228, doi : 10.1007/978-3-642-77228-3_10.

[53] P. Denny *et al.*, "On Designing Programming Error Messages for Novices : Readability and its Constituent Factors," 1983, doi: 10.1145/3411764.3445696.

[54] B. A. Becker *et al.*, *Compiler Error Messages Considered Unhelpful : The Landscape of Text-Based Programming Error Message Research*. pp. 177 - 208, 2019, doi : 10.1145/3344429.3372508.

[55] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *J. Vis. Lang. Comput.*, vol. 16, no. 1-2 SPEC. ISS., pp. 41–84, 2005, doi: 10.1016/j.jvlc.2004.08.003.

[56] B. Du Boulay, "Some difficulties of learning to program," *J. Educ. Comput. Res.*, vol. 2, no. 1, pp. 57–73, 1986, doi : 10.2190/3LFX-9RRF-67T8-UVK9.

[57] D. Fossati, B. Di Eugenio, C. W. Brown, S. Ohlsson, D. G. Cosejo, and L. Chen, "Supporting Computer Science Curriculum : Exploring and Learning Linked Lists with iList," vol. 2, no. 2, pp. 107–120, 2009, doi : 10.1109/TLT.2009.21.

[58] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," *ACM SIGCSE Bull.*, vol. 35, no. 1, pp. 153–156, 2003, doi : 10.1145/792548.611956.

[59] G. Lund, "Quality aspects of the program development process used by learner programmers." University of Abertay Dundee, p. 242, 2002. [Online]. Available at: https://rke.abertay.ac.uk/en/studentTheses/quality-aspects-of-the-program-development-process-used-by-learne.

[60] E. Almadhoun and J. Parham-Mocello, "Exploratory Study on Accuracy of Students' Mental Models of a Singly Linked List," in *2021 IEEE Frontiers in Education Conference (FIE)*, 2021, pp. 1–9, doi : 10.1109/FIE49875.2021.9637318.

[61] J. Sweller, "Element Interactivity and Intrinsic , Extraneous , and Germane Cognitive Load," pp. 123–138, 2010, doi: 10.1007/s10648-010-9128-5.