Assessing Novices' Program Comprehension based on Linked List Diagrams

Unaizah Obaidellah
Faculty of Computer Science & IT
University of Malaya, Malaysia
unaizah@um.edu.my

Abstract
Learning data structure can be difficult for low performing or novice students. Using diagrams during problem solving may benefit their program and algorithm understanding. We aim to evaluate the effectiveness of different types of linked list diagrams in assisting these students to solve specified problems in terms of naming operations and writing code for selected list operations of the data structure. Twenty eight novice computer science undergraduate students took part in this assessment. The metric used was students' accuracy to name the list operations and to write their corresponding code. Findings revealed that the problem solving process was best supported in the presence of some amount of diagrammatic notation that describes the operation under consideration.

Keywords: diagrams, novices, problem-solving, data structure, program comprehension

1. Introduction
First year computer science students frequently encounter difficulties to master computer programming courses. The difficulty is most evident when they have little or no prior experience of computer programming. Among others, this problem is caused by one or a combination of inadequate experience, practice, motivation and/or problem solving skills. Winslow (1996) suggested that it takes around 10 years to become an expert in computer programming. In most cases, reaching this level of expertise is unlikely during their undergraduate degree program, as the commonly practiced duration for a CS undergraduate program range within 3-4 years across universities internationally.

The programming courses are offered in various levels and complexity throughout the degree program. Often, it is compulsory for the students to undertake a few of these courses, each designed to prepare them with specific necessary skills. Generally, students are required to accomplish the fundamental programming courses prior to registering the more advanced courses such as algorithm design and data structures. In some institutions, these courses offered during a term of 3-4 months are considered a short duration to enable these students acquire adequate mastery. As a result, many of these students (considered novices due to their limited programming proficiency) fail to excel even in the fundamental programming course. However, practices in some universities allow students with marginal passing grades to register for the more advanced courses. Considering their weak understanding, problem solving and practical skills, efficient learning method is necessary to facilitate competency and continued motivation. Thus, using diagrams is a potential approach for this purpose due to its characteristics, being visually stimulating, reduces complexity of abstract ideas and motivates longer attention for most learners.

In the programming domain, learning with diagrams has been demonstrated useful to support program and algorithm understanding. The concept described as *program visualization* is defined by Myers (1986) as a conventional textual approach of specifying programing with the aid of graphics used to illustrate some aspect of the program or its run time execution. Myers (1986) further described that 'code visualization' illustrates actual program text in the form of graphics (i.e. flowchart) or adding graphics to show the algorithm of a program. The use of graphics to show algorithm also exhibit an abstract (i.e. conceptual and theoretical) form of how a program operates. For example, 2D displays such as flowcharts, block structured programs have been used to facilitate program comprehension Good, 1996; Smith, 1977). Some of these studies used diagrammatic representation for data structures (Brown & Sedgewick, 1984; Myers, 1980).

Novice programming students often claim that they acquire strong conceptual understanding but fail to write fully functional code. Various studies (Adelson, 1981; Pea & Kurland, 1984) remarked this as a result of poor ability in problem solving where students are unable to breakdown a problem into systematic and well-designed sub-problems. Our experience assessing weak (low performing) programming students on their ability to write computer programs show that they often memorise and recite codes and reproduce (with or without slight modification) them in the exam that tests similar questions. Their reproduction reflects superficial (weak) understanding as assessments of their knowledge, requiring higher level thinking skills, produce poor results. On the contrary, practicing "deep" learning strategies (involves close attention to meaning of the content being learned and relate them to learner's existing knowledge) may potentially reduce cognitively passive learning behaviour (i.e., review or highlight class notes and program examples; listen to lecture) among novices or low performing programming students. More active learning behaviours associated with deep learning enable students to breakdown complex processes in gradual steps, an approach that facilitates the problem solving process. Considering this advantage, we estimate that the use of diagrams in learning programming has some potential to enhance novices understanding in computer programming.

The work reported in this paper builds on the above and extends research in the programming domain with the use of diagrams. We focus on using diagrams to evaluate the effectiveness of learning linked list (a topic of the data structure course) among novices. In this context, we define novices as low performing programming students who marginally passed the fundamental programming course (scored 50% in their final examination assessment) and those who are re-taking the course due to a previous failure in the data structure course. Most of these students demonstrate insufficient fundamental conceptual knowledge and programming skills. We aim to test the extent of using the linked list diagrams in assisting low-performing students in solving a data structure topic. The data structure topic on linked list is chosen because the topic is relatively easy to be conceptually represented in a notational approach that shows the relationship contained between the elements. Each statement of the program can be represented as a notation. It is argued that learning from diagrams is useful given that the notational and relationship between the elements is understood by the learner.

Our experience teaching low performing students using the conventional approach by explaining the conceptual relationship with the corresponding code are less effective for this group of students as proven by high number of failure rates among some of them. Therefore, we devise a strategy that adopts the classical linked list diagrammatic notation as part of our method in teaching the linked list topic. Students are taught the relation between the algorithms of list operations with the linking operations that can be shown in the diagrams. In this study, we intend to compare three variations of linked list diagrams and evaluate its usefulness among these novices in solving a particular problem. The problems posed in a quiz setting required the students to study the given diagrams and answer questions by naming the corresponding operation before writing their code represented in a standard linked list notation (i.e., nodes and links).

Three versions of diagrams were designed (Fig. 1) which differ in the amount of complexity and information provided. The diagrams are expected to assist the process of solving the given programming problem. However, each diagram posed as a main question is posited to provide different level of assistance in helping the novices to solve the problem posed. We are interested to evaluate if a particular structure of diagrams (if any) is more beneficial for the novices. Better understanding on which type(s) of linked list structure is most useful in teaching the novices would improve the existing teaching method.

The diagrams illustrate complete or partial procedures (algorithms) for common operations of standard linked list concepts (i.e., add an element first in the list, add element at an index location, remove the last element in the list), taught in the classroom and similar to those found in textbooks. Each portion of the illustration can be translated into their corresponding code. Respectively, the following versions of the diagrams represent these operations: add an element first in the list, add an element at a particular index and remove the last element from the list.

The first version, "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., operationX). The second version, "complete and unlabelled" diagram showed a full notation only without meaningful labels to describe another type of list operation. Students were required to label the diagrams prior to writing the corresponding program code. The third version, "complete and labelled" diagram showed a full notation with labels describing a different operation. No "partial and unlabelled" diagram was given as information provided by this type of representation is considered insufficient and less meaningful to support effective cognitive processes.

We predicted that the "complete and unlabelled" (version 2) diagram would assist the low performing students more than version 1 and 3 diagrams due to the additional fill in the blanks task that would set as an advance organizer for task that follows (i.e., writing code). The version 3 diagram is expected most difficult and version 1 as intermediate in facilitating novices in solving programming problems associated with the linked list of the data structure course. The version 3 diagram is expected to provide least code cue support to assist the students in writing the code unlike version 1 and 2, due to the missing fill in the blanks activities.

2. Method

Participants. Twenty eight undergraduate computer science students (6 male, 22 female), aged between 19-24 years old (Mean age=20.4 yr.; SD=1.63), from a public university in Malaysia participated in the assessment in partial fulfilment of a course requirements. The participants who registered for the Data Structure course were mostly low-performing students as they were either re-taking the subject due to previous failure to pass with a minimum of 50% of the course assessment (C-) or taking the subject first time upon similar marginal passing of the previous fundamental programming course (i.e., C-).

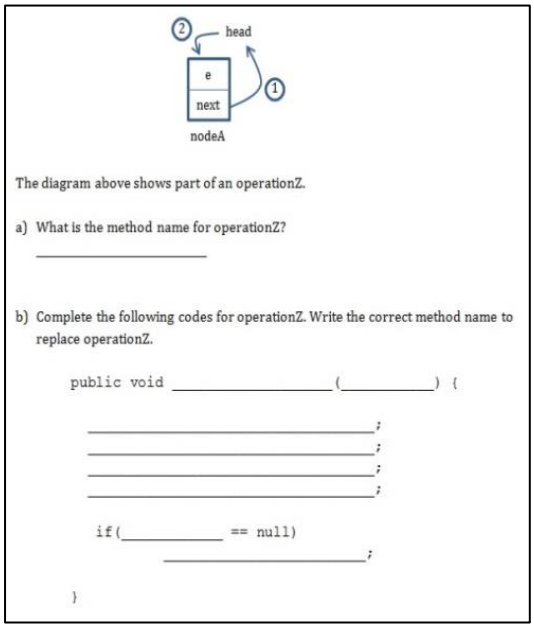

Fig 1(a). Version 1 (addFirst operation)

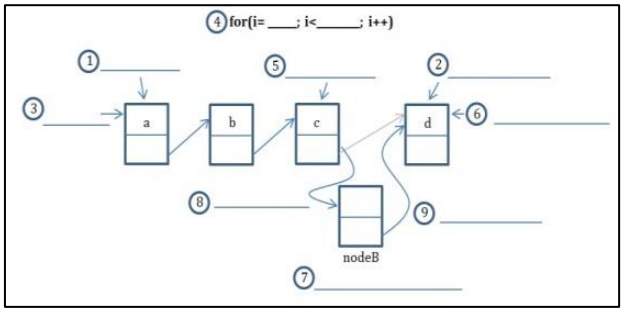

Fig 1(b). Version 2 (addAtIndex operation)

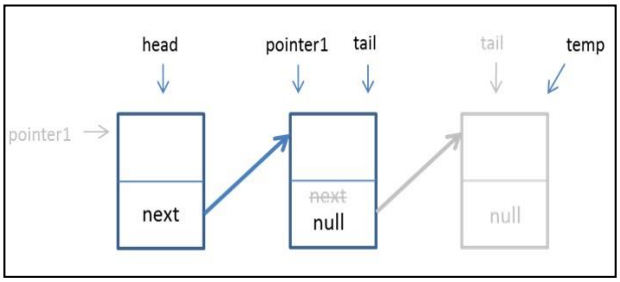

Fig 1(c). Version 3 (removeLast operation)

Fig. 1. The three diagrams used in the assessment

Materials. Two open ended questions follow each diagram to assess the student's comprehension level. These questions asked students to name the operation and write the complete program for the operation named. The instruction specified them to 'write the correct method name to replace operationX' in the method signature of their program. In the first and second version, students were asked to complete the program by filling in the blanks of the underlined missing statements (as of fill

in the blanks task). The second version requires students to label the numbered parts of the diagrams by filling in the blanks task prior to writing the full program. However, in the third version, students were required to write the complete code without any underlined cues given. Different operations were being tested for each version. These operations refer to adding an item at a particular position (i.e. addFirst for the first version, addAtIndex for the second version, removeLast for the third version).

Design. All students performed all conditions by completing all questions for the three versions of the diagram - "partial and labelled", "complete and labelled" and "complete and no labelled", once each.

Procedure. All students answered the questions in a quiz environment within one hour during a lab session in a paper-based test. They were informed about the quiz one week before it took place. No access to a compiler, learning resources or peer discussions was allowed during the quiz. Although there was no restriction in the program language used to write the code, all students wrote their programs using the Java language following that used in teaching. The students were familiar with all of these notations and diagrams as they were used in the teaching part of the course. Prior to the quiz, the students completed their exercises including tutorial and lab assignments using these diagrams.

Coding. All answers were coded blind to condition by a research assistant. The score given was one mark for each correct answer. The distribution of the full marks for each part of the question is as follows: 1) one mark if a method is named correctly, 2) 10 marks if all code statements are complete and correct. The total mark of the quiz is 33 marks (11mark for each question).

No additional minus mark (penalty) was deducted for any wrong statements written for questions on writing code. As writing the program code occurred on paper, minor syntax errors (i.e., missing semicolon or open brace) that do not affect the answers being evaluated were disregarded. A final score was calculated as the total number of correct statements on each main question.

3. Results

Table 1 shows the results of means and standard deviation for naming method, writing code and total overall score for each diagram version.

Type of diagram	N	Name operation		Write code		Total score	
		M*	SD	M*	SD	M*	SD
Partial and labelled (version 1)	28	0.78	0.42	6.58	3.12	7.36	3.54
Complete and unlabelled (version 2)	28	0.93	0.27	6.01	2.79	6.94	3.06
Complete and labelled (version 3)	28	1.00	0.00	3.32	3.59	4.32	3.59

**Total means scores (M) are out of 1 for name operation, 10 for write code and 11 for the total score.*

Table 1. *Descriptive statistic of the results.*

Generally, the results for version 1 and 2 are similar. The mean results of the total scores for version 1 and 2 diagrams may suggest that these types of diagrams are more effective in facilitating novices during the solution process. Similar pattern also showed for writing the code for the corresponding named operation. However, all 28 students correctly named the operation illustrated in the version 3 diagram (M=1.00, SD=0.00). A t-test comparison between the diagrams for the total marks was

significant, p<.001, between version 1 and 3, and version 2 and 3. Table 2 shows the number of students who scored full marks in each task (i.e., name operation and write code). Nine students wrote complete and correct code for version 1 diagram, while only two and three students for, respectively, version 2 and 3. The overall results indicate that the version 1 diagram is more effective than versions 2 and 3. One third of the students were able to write the code better in version 1 (Table 2). Version 3 diagram was the least effective for writing code although all students were able to name the operation correctly.

Type of diagram	Number of student	
	Name operation	Write code
Partial and labelled (version 1)	21	9
Complete and unlabelled (version 2)	25	2
Complete and labelled (version 3)	27	3

Table 2. *The number of students scoring full marks for each task and overall score*

4. Discussion

The aim of the work is to evaluate whether different types of linked list diagrams are useful to facilitate the process of writing program among novices. The problems posed in a quiz setting required the students to study the given diagrams before they answer questions by naming the corresponding operation before writing their program.

The higher mean scores (total marks) for the partial with label (version 1) and complete without label (version 2) diagrams suggests that these representation are potentially most useful in assisting novices during their problem solving process. In terms of writing code, both of these diagrams employed the fill in the blanks tasks, which could have cued the participants to think of the correct steps of the algorithm. This is because information on these diagrams such as the labelled nodes and their association may serve as advance organizers for the learner. A pattern of highest score for version 1 (total score, writing code and number of student scoring full marks) diagram may suggest its effectiveness more than the others. This could be due to less complicated logic and/or fewer lines for the code for the *add first* operation (i.e. adding an element first in the list) shown in version 1 diagram (Fig 1(a)).

Although all students correctly named the operation for the version 3 diagram, their ability to write the corresponding program were not equally demonstrated. This means the students were able to analyse the processes for the *remove last* operation (i.e. removing the last element in the list), but lack effective solution strategies. Another indication is that these students attained necessary conceptual understanding of the list operation considered. Fewer number of students who scored full marks in the questions on version 3 diagram and in scores for writing the code, indicated that writing the complete program for the operation illustrated in the diagram is more difficult without the presence of the underlined parts (as fill in the blanks question), unlike that available for version 1 and 2 diagrams. Thus, the lack of code cues might have made the task with diagrams 3 much more difficult. Although this may seem speculative, perhaps, the availability of the underline portion for the statements has some relation with the amount of confidence these novices attain. This is considered because the amount of complexity of logic involved can be considered similar with the *add first* operation (version 1).

Findings for the version 2 diagram follows our prediction, which hypothesised that requiring students to label the numbered parts of a diagram prior to writing the code would assist their problem solving

process. It is expected that this practice (i.e., parts labelling) would form a mental schema for the operation depicted in the diagram, where the labelled portion serves as fillers for the slots given as underlined spaces in the writing code task. However, a pattern of lower marks for the writing code task, the total marks, and reduced number of students scoring full scores than version 1 diagram do not fully support this prediction. A potential reason why version 2 diagram was not as effective as version 1 diagram may be due to more complicated logic these novice student have to evaluate, thus reducing their capability to prioritize solution strategies and/or eliminate irrelevant criteria when solving the question. Although two groups of low performing students participated in this study - retakers and non-retakers (low performing pre-requisite course students), the results suggests that the student's familiarity of the diagrams does not seem to affect their performance.

An alternative interpretation of the present results could be discussed in terms of 'which exercise was the easiest' with respect to the use of these diagrams. Diagram 1 was considered easiest to write code. Similarly, writing code for diagram 2 was easier than diagram 3. Although an opposite pattern showed for diagram 3, the complete and labelled notation to describe the *remove last* operation was easiest for the participants to evaluate for naming the operation.

The limitation of the present assessment only considers one version for each operation. Thus, it is useful to verify the present findings with different versions of the diagrams with variation of operations including control groups in the group of participants. Therefore, due to the no task variation, the present results could have been heavily influenced by the operation described in each diagram, as such, the results are less generalizable. In addition, the present work can be further improved in various ways, such as comparing the effectiveness of these diagrams between high and low performing students and comparing its effects with standard conventional program without any use of diagram.

5. Acknowledgement

The research described in this paper was funded by the FRGS 062-2014A and RP030C-14AET. We thank the reviewers and associate editor for their comments which improved this manuscript.

6. References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, *9*(4), 422–433.

Brown, M. H., & Sedgewick, R. (1984). A system for algorithm animation. *SIGGRAPH Comput. Graph.*, *18*(3), 177–186.

Good, J. (1996). *The "right" tool for the task : an investigation of external representations, program abstractions and task requirements*.

Myers, B. (1980). Displaying Data Structures for Interactive Debugging.

Myers, B.A. (1986). Visual programming, programming by example, and program visualization: a taxonomy. *ACM SIGCHI Bulletin*, *17*(4), 59–66.

Pea, R., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*.

Smith, D. C. (1977). Pygmalion: A Computer Program to Model and Stimulate Creative Thought.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22.