Exploratory Study on Accuracy of Students' Mental Models of a Singly Linked List

Eman Almadhoun
School of EECS
Oregon State University
Corvallis, USA
almadhoe@oregonstate.edu

Jennifer Parham-Mocello
School of EECS
Oregon State University
Corvallis, USA
parhammj@oregonstate.edu

***Abstract*—This Research Full Paper presents a study on the accuracy of computer science (CS) novices' mental models about linked lists in the C programming language. In CS, learning abstract fundamental concepts that require students to understand memory management can be very difficult and lead to misunderstandings that carry on into the advanced topics. This is especially true for linked lists data structures because they serve as a bridge to understanding more advanced data structures. Therefore, it is important to understand how students think about linked lists for improving teaching and learning.**

Exploratory research on mental models in CS is not as well-known as in other disciplines, such as Psychology, Education, Chemistry, Physics, and Mathematics. Since CS is based on abstract concepts like in Chemistry, Physics, and Mathematics, we believe the CS education community can benefit from more research on mental models and student reasoning, especially in fundamental areas such as data structures and algorithms. Hence, we conducted 2-hour semi-structured, think-aloud interviews with 11 undergraduate students to uncover the accuracy of their mental models, including their conceptual and procedural understanding, about singly linked lists.

Our results suggest that none of the participants have an accurate mental model of a singly linked list, after learning about them and implementing them in their data structures course. Students struggle with expressing their conceptual understanding in their verbal responses to interview questions, while their scores for the coding questions are much better. Overall, the majority of students have a good procedural understanding of how to use the components of a linked list and implement the operations on a singly linked list, but most students cannot express their conceptual understanding of the components in a singly linked list. The results from this research suggest that educators need to spend more time covering linked lists in their data structures courses and make sure that students understand how their prerequisite knowledge of pointers and memory management integrate with the new knowledge about linked lists.

***Index Terms*—Knowledge retention, Computer science, Sophomore, Doctoral institution, Interviews, Mixed methods research, Expert-novice**

I. Introduction

Algorithms and data structures are some of the most important courses for computer science (CS) undergraduate students. Yet, learning and teaching data structures can be difficult, due to the abstraction and data structure manipulation that students cannot see or touch in order to build a model of the dynamic process [1]. In this research study, we explore students' understanding of a singly linked list in the context of the C language, which is the first data structure taught after arrays and the language used in the participating university.

In C, a linked list is a linear structure that has nodes stored at non-contiguous memory locations and linked using pointers [2]. There are many studies on students' misconceptions of advanced data structures, such as heaps, binary trees, and hash tables [3]–[5]. However, little attention has been paid to the fundamental, basic data structures, such as arrays and linked lists [6], [7].

We focus on linked lists because they serve as a bridge to understanding more advanced data structures [8]. For example, the concept of a node from linked lists is important for understanding a binary tree later. Instead of having a node with next and previous pointers, as in the case of a doubly linked list, a binary tree has a node with left and right pointers.

A recent study shows that undergraduate students do not reason well about linked lists [6], and we believe that identifying students' conceptual and procedural difficulties when working with linked lists in C can be helpful to educators and students. In C, a linked list builds upon many concepts, such as dynamic memory management, pointers, and user-defined structs, in addition to the specific linked list concepts. For the simplest linked list, students need to understand two primary components: 1) a user-defined struct type called a *node*, which has data of any type and a pointer to the next node, and 2) a *head pointer* that points to the first node in the list. Implementing a linked list not only involves understanding how to define and create a node and a head pointer, it also involves understanding operations on linked lists, such as inserting nodes into the list, managing nodes linked together in memory, and the syntax for writing the code in C.

Cognitive science uses the term "mental models" to describe a cognitive representation of an abstract structure or a piece of the abstract structure of the real-world, situations, events, tasks, problems, procedure, concepts, activities or phenomenon and the relations between them [9]–[11], and mental models are very important for learning a new concept, especially abstract concepts [12]. According to Johnson-Laird, mental models help us understand problems, find a suitable solution, and predict outcomes based on actions [13]. If learners construct correct mental models, know how to use them well, and understand the network of connections between them,

then they will save time on understanding some concepts and solving problems, as well as improve their overall learning [11] and reasoning [13].

Understanding how students reason about a concept provides insight into their mental model about the concept [13]. However, exploratory research on mental models in CS is not as well-known as it is in other disciplines, such as Psychology, Education, Chemistry, Physics, and Mathematics. Since CS is based on abstract concepts, we believe the CS community benefits from more research on mental models and students reasoning, especially in fundamental areas such as data structures and algorithms. Therefore, we address the following question in this research study.

RQ: How accurate are students' mental models of a singly linked list?

We measure the accuracy of students' understanding by grading each student's response to interview questions using a rubric. The total points received on the rubric determines their overall score for the question. We evaluate how accurate students' conceptual and procedural understandings are using the overall scores and a mapping to linked list concepts, which addresses the accuracy of students' mental models of singly linked lists. While it is important for students to have factual knowledge, which includes the terminology and specific details about linked lists, we are more interested in students' conceptual and procedural knowledge that leverage their factual and metacognitive knowledge [14].

II. Related Work

A. Knowledge and Learning Abstract Concepts

Learning computer programming for CS undergraduate novice students is challenging [15]–[20]. This is because computer programming consists of many abstract concepts, instructions, and processes that one needs to follow [21]. Students may feel frustrated and unable to continue learning computer programming because they do not have a good visualization of the flow of these processes. Visualization is especially critical for algorithms and data structures [16]. Data structures describe abstract containers that are used to store, organize, and access data efficiently, while an algorithm contains the processes or steps that are followed to solve a specific problem [16]. Many educational institutions teach algorithms and data structures using code and syntax with definitions and drawing pictures for illustration. However, some students may struggle to understand the code, the pictures, or the link between the two.

The more abstract a concept is, the more it needs thinking in order for a learner to understand [22]. "A Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives" presents 4 types of knowledge, which are: factual, conceptual, procedural, and metacognitive [14]. These types of knowledge are ordered from more concrete (factual) to more abstract (metacognitive).

In this research study, our focus is on the conceptual and procedural knowledge needed to have a deep understanding of linked lists. Conceptual knowledge is defined as "the interrelationships among the basic elements within a larger structure that enable them to function together", while procedural knowledge is defined as "how to do something, methods of inquiry, and criteria for using skills, algorithms, techniques, and methods" [14]. Conceptual and procedural understanding are not independent, and many philosophers studying theories of knowledge claim that procedural understanding is based on conceptual understanding [9], [23], and likewise, conceptual knowledge is built upon factual knowledge. Therefore, we first ask students' about their conceptual understanding of the components in a singly linked list, and then, we observe their procedural understanding of implementing these components and operations on lists.

B. Mental Models and Reasoning

Norman defined mental models as internal representations built on individual experiences in the real world [24]. Mental models can be naïve or immature [25] and can be inaccurate and sometimes incomplete [9], [13]. Henderson and Tallman [11, p. 36] argue that "Errors can occur because we do not activate all our knowledge at one time", which can come from building several mental models in the short-term memory due to thinking limitations [13].

Novices have limited experience and knowledge in solving problems, and they construct partial mental models or map prior mental models to the new situation. For example, when novice students want to find errors in a program, they will try all possible sequences to find the errors [11], but this is impractical and time-consuming. If students know how to build a correct mental model of the situation and strategically map the concepts with other similar mental models, they can check one possible sequence and discard all irrelevant ones [26], [27]. Therefore, it is important to build accurate mental models of abstract concepts to help individuals correctly solve problems.

Measuring the accuracy of students' mental models helps educators explore students' understanding of a concept. Many methods elicit students' mental models by asking students to talk about their understandings or explain them correctly [28]. These methods include having students think-aloud while performing a task [29], [30], writing a "Task Reflection" after solving a coding problem [31]–[33], and diagramming tasks to create a visualization based on their prior knowledge of the process [34]. All methods are qualitative, and some methods gain a richer understanding of the participants' mental models and the way they are reasoning and thinking than others.

III. Research Method

In this research study, we use observations of students solving problems in a semi-structured, think-aloud interview to identify the accuracy of a student's mental model of singly linked lists, which is the exploratory research needed to develop a concept inventory that measures students' commonsense beliefs about linked list concepts. While we recognize there are different types of linked lists, this initial research study only focuses on students' conceptual and procedural

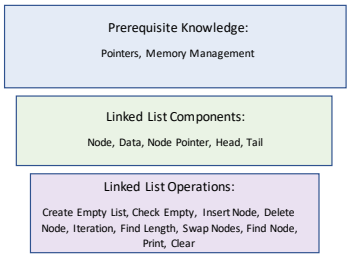

Fig. 1. Singly Linked List Concept Categories

knowledge of a singly linked list and how well students can manipulate the components in a subset of operations.

A. Categorization of Linked List Concepts:

We present three categories required for understanding and programming singly linked lists in the C language. These three categories include 1) the prerequisite knowledge about pointers and memory that is required before linked lists, 2) the components (or parts) of linked lists, and 3) the operations performed on linked lists (see Fig. 1).

The components of a singly linked list include 1) the node with 2) the data and 3) the node pointer members, as well as 4) the head pointer and 5) the tail pointer. We realize that the tail pointer is not required, but we believe it is an important component for students to conceptually understand. We identify ten operations on linked lists that include 1) creating an empty list, 2) checking if it is empty, 3) iterating through a list, 4) clearing a list, 5) finding the length, 6) finding a value, 7) printing the list, 8) swapping nodes, and 9) inserting and 10) deleting nodes. These categories provide a comprehensive set of prior knowledge and concepts required to have a complete mental model.

B. Linked List Framework

We measure students' mental models of a singly linked list by examining students' conceptual understanding of the singly linked list components and their procedural understanding of how to implement these components and operations on lists. Fig. 2 shows a complete view of the framework used to examine the students' mental models of a singly linked list. The figure shows that singly linked list knowledge (top-right rectangle) requires prior knowledge of pointers and memory management (left rectangle). Students must have a deep understanding of this prerequisite knowledge to successfully understand and implement singly linked list concepts in C.

We use the C language in this research because the participating university uses it. Using C to implement linked lists requires students to have an accurate mental model of pointers and memory management, including pointer declaration, assignment, initialization, pointer manipulation, and writing code with functions and pointers, in order to apply this knowledge correctly to a singly linked list (see Fig. 2). In this research study, we assume students already have the prerequisite knowledge needed to understand linked lists, since it was required knowledge in the prerequisite to the data

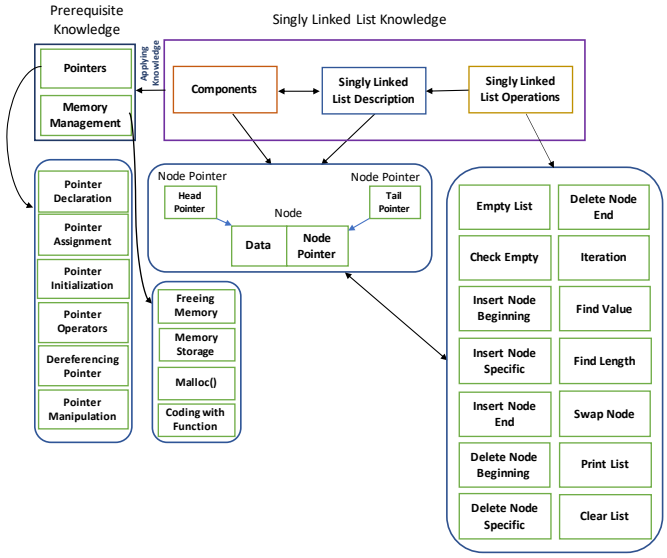

Fig. 2. Singly Linked List Framework illustrating the concept requirements among prerequisite knowledge and a singly linked list.

structures course. Therefore, we do not directly assess students understanding of pointers and memory management.

First, students must generally understand what a linked list is, which requires knowledge of the components but not necessarily the operations. Next, students must understand the five components of a singly linked list in order to perform operations on the list. In addition to measuring students conceptual and procedural understandings of linked list components, we measure their procedural understanding of 14 operations on linked lists (see Fig. 2). We expanded inserting and deleting nodes to include the beginning, the end, and a specified location in the list. Due to time constraints, we only ask students to recognize the code for swapping nodes, finding a value, printing the list, and clearing the list, and we do not ask student to explain the operations. However, since students thought aloud while implementing the linked lists, we are able to infer their conceptual understanding.

There is a strong dependency between understanding the linked list components and the ability to describe a singly linked list. If students have a deep understanding of the linked list components, they can correctly combine these components when describing a singly linked list, but students need to completely understand each component to accurately describe a list. Therefore, after identifying students' reasoning and understanding of individual linked list components, we explore how students' combine their reasoning about these components when broadly describing a singly linked list and the operations on them. Even though we do not directly measure students' prerequisite knowledge (pointers and memory management) in the interview, we are able to evaluate this information from their responses and code.

C. The Semi-Structured Interview Questions:

We used the categories and framework of singly linked list concepts as a guide for constructing the interview questions. The interview questions came from a variety of sources including Kruse's "Data Structures and Program Design in C" textbook [2], online resources "GeeksforGeeks" [1], classroom discussions, and the researchers. The interview questions were open-ended consisting of three main types: verbal, coding, and recognition.

First, we asked students to verbally respond to the following questions about each of the five components and a singly linked list.

- Describe a singly linked list? Draw a picture if needed.
- Describe a node in a linked list.
- Describe a node pointer in a linked list.
- What kind of data can be stored in a linked list?
- Describe the head and tail pointers in a linked list.
- What are the pros and cons of having a head and tail pointer?
- Do you create the head of a linked list as a node or node pointer?

The questions were very vague to find out what knowledge students use to answer the questions without probing them too much. While we were careful not to lead or bias student responses during the interview, we asked students follow-up questions, such as "What do you mean?", "Why do you think that?", "Can you elaborate more?", etc., or to express additional comments/thoughts to obtain more information, when students do not say much.

Next, we asked students to write code/pseudocode to create an empty linked list, check if the list is empty, insert and delete nodes at the beginning, end, and specific location in the list, and find the length. Then, we asked students to recognize code for swapping adjacent nodes, finding a value in the list, printing a list, and clearing a list.

D. Data Collection:

With permission from the Institutional Review Board (IRB) to recruit students, we visited the data structure class, which the researchers were not teaching, in the fourth week after covering linked lists, to ask students to participate in this study. After the class visit, we emailed the 250 students with a link to the consent and survey to determine who was interested in the interview and if we had variation among participants. Out of the 40 students who took the survey and expressed interest in the interview, only 11 committed to the 2-hour, $15/hr semi-structured think-aloud interview.

While the number of participants is small, we believe the rich data yielded from these participants provides meaningful, initial insights about patterns of students conceptual and procedural knowledge about singly linked lists in C that other researchers can leverage for future studies and educators can use to improve their instruction and/or assessment of linked lists. Since there is not much research on how students think about linked lists to use as a foundation, we find this research similar to an initial, empirical study on students' understanding of free fall conducted by Champagne et al. in 1980 with only 12 participants interviewed and observed [35], which is cited by McDermott's work on mechanics that Hestenes uses in the development of the Force Concept Inventory [36], [37].

To capture the dynamic process of thinking, we followed a think-aloud verbal protocol asking students to verbally express their thoughts as they answer a question or solve a problem [38]. We audio and video recorded all interviews using a Windows touch screen tablet to capture what the student said, coded, and drew. There was no time limit for answering each question in the interview to reduce students' stress level. We used software to initially transcribe the audio recordings, and the researchers reviewed the software transcription for any mistakes. Even though students knew that they could run the code they wrote on the computer, we reminded them of this when they were not sure about the correctness of their code. We also reminded students that they could draw on the computer.

Demographics: All the participants were ages 18 to 24, and Table I shows that the majority of the students were 'B' students, self-identifying as white male majoring in computer science, and most students had knowledge of linked lists before the data structures class, which is typically covered at the end of the second programming class to help students transition from C++ to pure C before the data structures course. Even though the two programming courses prior to the data structures course are in C++, it is worth noting that the participating university begins teaching pointers halfway into the first programming class, and the curriculum continues to stress pointers and memory management throughout the second programming class.

Most participant demographics closely represent the demographics in the data structures course at the participating university. Even though there are about 30% ECE majors in the data structures class, the number of female students is sadly not much higher with only approximately 15% of the course identifying as female. We give the students pseudonyms for their names that reflect their demographics, such as Ecer is the ECE student and Chemi is the chemical engineering student.

IV. Evaluation and Data Analysis

In the semi-structured interview, we explicitly assess every linked list concept identified in Fig.s 1 and 2 , except for students' prior knowledge of pointers and memory. To identify how accurate students' mental models are from the qualitative interview data, we use quantitative and qualitative analysis. First, an expert with many years of experience teaching linked lists in C wrote a correct (or perfect) solution for each question in the semi-structured interview, and then we broke down the expert solutions into small pieces to create an initial rubric.

Using the initial rubric, two researchers independently coded 20% of the data, which was two participant responses for each question. To compute the inter-rater reliability (IRR) for applying the rubric to student responses, we used the

[1]https://www.geeksforgeeks.org/

average agreement between raters for each piece of the solution in the rubric for both participants. For each question having an IRR below 80%, the researchers discussed their differences, made changes to the rubric, and graded two different participants with the updated rubric. After reaching above 80% IRR for each question, the researchers used the agreed upon final rubric to independently grade the remaining participants, as well as re-grade any participants less than 80%.

The final rubric for the five verbal questions about the four basic linked list components have the same format with two points for each correct piece of the expert solution, and one point for partial credit in the cases where a student only mentions part of what is in the expert solution (see Fig. 3). We include the reasons for not being correct or partially correct, as well as any other observations we find surprising.

For the two coding questions, the final rubrics have two sections: the core section and the additional/prerequisite knowledge section, which contains knowledge of other related concepts (see the bold line in Fig. 4). To have a complete procedural understanding of the components, students must correctly implement each piece of the solution in the core section. Each piece of the solution is worth three points for correctly implementing it, and students get two points for trying to implement a piece of the solution or expressing it in pseudocode. Students get one point for stating what needs to be done, even if they lack procedural understanding. For example, the first coding question asks students to create an empty list, and since we do not state that students must make a function to do this, creating a function is not part of the core section (see Fig. 4). However, a student must implement all pieces in the core section to successfully create an empty list.

During the interview, only two students compile their code. In the situation when the compiler fixes their errors, we do not give them full points, but we give them full points when they fix their bugs in later questions by themselves as a result of the earlier compiler message. There is no point deduction when students perform minor errors in coding, such as missing semicolons, missing brackets, or do not typecast the return memory address by `malloc` when a new node is created. This is because these syntax errors do not directly correlate to misunderstanding the linked list concepts.

TABLE I
DEMOGRAPHIC AND BACKGROUND INFORMATION

PID	GPA	Gender	Race	Major	LL Knowledge Prior DS Class
Joe	3.54	Male	White	CS	Yes
Bob	3.2	Male	White	CS	Yes
Suzy	2.88	Female	Asian	CS	No
Bill	3.29	Male	White	CS	No
Max	2.97	Male	White	CS	Yes
Phil	3.5	Male	White	CS	No
Feng	3.16	Male	Asian	CS	Yes
Xeng	3.36	Male	Asian	CS	Yes
Chemi	3.98	Male	Asian	Chem. Eng.	Yes
Ecer	2.99	Male	White	ECE	No
Nate	3	Male	White	CS	Yes

TABLE II
OVERALL SCORE FOR EACH LINKED LIST CATEGORY.

PID	SLL[a] Description	SLL Components	SLL Operations	Overall SLL
Joe	72	59	76	69
Bob	67	72	95	80
Suzy	72	21	23	29
Bill	72	67	91[b]	77
Max	72	61	87	74
Phil	28	63	90	70
Feng	61	65	77	69
Xeng	44	69	95	74
Chemi	50	63	87	71
Ecer	56	56	74	62
Nate	78	48	76	64

[a]SLL refers to a singly linked list.
[b]Score is calculated differently due to missing responses.

V. ACCURACY OF STUDENTS' MENTAL MODELS OF SINGLY LINKED LIST

We define accuracy as the correctness of the students' answers to interview questions compared to an expert. We identify students' mental models by examining students' conceptual and procedural understanding of the concepts of a singly linked list. We consider students with a perfect overall score for all the linked list concepts as having an accurate mental model. Participants with a 90%-99% have a very close to accurate mental model. Those getting an 80-89% are above average, and those with a score below 80% indicate a concern.

Table II shows the overall score for each student based on their general description of a singly linked list, conceptual and procedural understanding of the components, and procedural understanding of the operations. None of the participants have a completely accurate mental model of a singly linked list. Bob has the highest overall score (80%), while Suzy, who could not write in code and preferred pseudocode, has the lowest overall score (29%). Even though Suzy selected not having seen linked lists prior to the data structures class in the demographic survey, she states during the interview that this is her second time taking the data structures class, and she is still struggling.

Total Points: 8/**12**	Point Divvy: Gain points for mention of each of these topics:	**Partially Correct**	**Correct**	**Comment: Reason for not getting full marks or anything else interesting** about their conceptual understanding
2	**A node stores data**		2	
2	The **data** can be **any type**		2	
2	**A node stores a pointer or memory address**		2	
2	**The pointer points** to a **different/next node**		2	
2	The **pointer** points to **Null**			Not present
2	**Null pointer indicates the end (or last node) of the linked list**			Not present
Other Interesting Observations/Comments Student Responses/Answers: Participant says, "a node in a linked list is kind of a box that contains both a pointer to another node which contains kind of an address of another one of these boxes and a variable which can contain any sort of information."				

Fig. 3. Example rubric for verbal question 1.

Total Points: **16/16 6/6**	Point Divvy: Gain points for mention of each of these topics:	States **what** needs to be done (conceptual)	**Attempts** to execute (or states) **how** to implement the what (procedural)	**Correctly** executes **how** to implement the what (procedural)	**Comment: Reason for getting/not getting full marks or anything else interesting** about their conceptual and procedural understanding
3	**Defines a node**			3	
3	Node **has data part**			3	Use int data type
3	Node **has pointer part**			3	
3	**Creates a node pointer** (creates head)			3	
3	**Assigns** node pointer (**head**) to **NULL**			3	
1	**The list** is **successfully created**	1			
3	Creates a **function**			3	
3	Returns head pointer			3	
Other Interesting Observations/Comments Student Responses/Answers:					

Fig. 4. : Example rubric for coding question 1.

Students do not provide a detailed description of a singly linked list and its components. We explicitly mention that students can draw when describing a singly linked list because research shows that drawing pictures helps students reason better while solving problems [39]. Phil and Xeng perform the lowest on this question, and they did not draw or visualize the linked list on the tablet while describing it. However, we find that over half have above average understanding of programming singly linked list operations in the C language. Bob, Bill, Phil, and Xeng have a very close to correct mental model of the operations on linked lists, and Max and Chemi score above average on their procedural understanding of linked list operations.

Table III shows a detailed view of students' conceptual and procedural understandings of singly linked list components and students' procedural understanding of the linked list operations. We calculate students' scores using the final rubrics that measure individual concepts in all interview responses. These results reflect the accuracy of each linked list concept, as well as the overall accuracy of their mental model.

Linked list components are fundamental concepts of a linked list. While none of the students have accurate conceptual and procedural understandings of all the components of singly linked lists, there are many students who have an accurate procedural understanding of some components (top four orange rows in Table III). However, most students do not have an above average conceptual understanding of the components (top four light-purple rows in Table III).

Almost half of the participants (45%) do not have a complete understanding of the data component (Suzy, Bill, Phil, Ecer, and Nate), and Suzy again scores the lowest (20%). This is because she does not conceptually understand that the data stored in a node can be of any type or how to create the data member of the node. On the other hand, some students struggle to store the data in a node using a void pointer or they use a dot operator instead of dereferencing the node pointer with an arrow.

Students are better at procedurally understanding how to use the linked list components (top four orange rows in Table III) than they are at providing complete verbal descriptions of the components (top four light-purple rows in Table III). However, only about 36% of the participants (Bob, Bill, Max, and Xeng) accurately code a node component, and only 27% of the participants (Bob, Bill, and Xeng) correctly code a node pointer as part of a node. Even fewer, only 18% of the participants (Bob and Max), accurately code a head pointer, which is independent of a node.

Overall, students perform better on the operations than they do on components. Even though Table II shows that none of the students have a perfectly accurate procedural understanding of the overall operations, there are many students who correctly recognize code for some of the linked list operations, except for swapping adjacent nodes. For example, all students can correctly identify code for printing a singly linked list, and all students except Suzy recognize code for finding a value and clearing a list. It is interesting that only Bob and Xeng can write correct code for finding the length of a list, but yet, there are many students who can correctly iterate through a list. This is because many students forget about checking for an empty list.

Very few participants outside of Bob, Bill, Phil, and Xeng, who have a very close to accurate procedural understanding of all operation concepts, have an accurate understanding of how to code most of the operations that manipulate a linked list, with the exception of inserting and deleting from the beginning. We find that the majority of students conceptually understand the operations, but they lose points because they fail to correctly write the syntax in C for the operations due to misunderstandings about some components or prerequisite knowledge.

VI. Factors Affecting the Accuracy of Students' Mental Models of Singly Linked List

There are many factors that lower students' scores in the interview. In the following enumerated list, we elaborate on the five factors affecting the accuracy of student's mental models of singly linked lists.

A. Missing Details in Verbal Responses

Students may build incomplete mental models while learning linked lists or maybe they do not believe or understand everything in their current mental model. A linked list is an abstract data structure requiring prior knowledge of pointer manipulation and memory management, in addition to an understanding of 4-5 components and 14 operations. It is hard for students to verbalize every aspect of the linked list components, but it is much easier for them to program the components and operations.

The majority of the participants understand most of the linked list concepts when explicitly asked. However, when we asked broad open-ended verbal questions about a singly linked list and its components, they do not verbalize all the important information that an expert would. Maybe they cannot recall the information relative to other components or connect ideas to other linked list concepts to complete their response. In any

TABLE III
SCORES PER SINGLY LINKED LIST CONCEPT BASED ON CONCEPTUAL (LIGHT-PURPLE) AND PROCEDURAL (ORANGE) UNDERSTANDINGS.

SLL Components	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate
Node	80	65	15	50	65	45	65	55	50	55	55
	33	100	38	100	100	93	87	100	93	84	91
Overall Node	60	80	25	71	80	66	74	74	69	68	70
Data	100	100	0	50	100	100	100	100	100	100	100
	100	100	33	67	100	67	100	100	100	67	67
Overall Data	100	100	20	60	100	80	100	100	100	80	80
Node Pinter	40	30	50	30	60	30	20	20	40	20	20
	74	100	21	100	72	97	87	100	95	77	77
Overall Node Pointer	67	86	27	86	69	84	73	84	84	65	65
Head & Tail	43	33	13	30	15	30	30	33	28	28	10
	60	100	13	93	100	73	93	93	67	67	33
Overall Head & Tail	47	51	13	47	38	42	47	49	38	38	16
SLL Operations	**Joe**	**Bob**	**Suzy**	**Bill**	**Max**	**Phil**	**Feng**	**Xeng**	**Chemi**	**Ecer**	**Nate**
Create Empty List	19	100	44	94	100	63	81	100	81	25	75
Check Empty	43	100	0	100	100	100	71	100	71	71	43
Insert Beginning	77	100	15	100	100	92	85	85	54	69	31
Insert Specific Location	71	100	21	82	68	93	82	100	96	82	86
Insert End	82	100	25	93	100	93	86	100	96	79	93
Delete Beginning	81	78	6	78	100	81	94	81	75	63	0
Delete Specific Location	100	100	23	100	77	100	64	100	95	86	86
Delete End	74	100	11	100	58	89	58	100	95	68	89
Iteration	100	100	9	100	100	97	45	100	100	88	97
Find Length	82	100	41	82	95	91	86	100	95	91	95
Swap Nodes	100	33	33		100	100	33	67	33	67	100
	50	50	0	50	50	50	50	50	50	50	50
Overall Swap Nodes	80	40	20	50[a]	80	80	40	60	40	60	80
Find Value	100	100	50	100	100	100	100	100	100	100	100
Print List	100	100	100	100	100	100	100	100	100	100	100
Clear List	100	100	0	100	100	100	100	100	100	100	100

[a]Score is calculated differently due to missing responses.

case, asking novice general questions in interviews and for assessment is an important consideration. Instructors need to teach students to be more detailed in their verbal responses to questions, or interviewers and instructors should use more directed questions to elicit more details from novice.

B. Lack of Attention to NULL

When we ask the participants to describe a node pointer in a linked list, the majority of the participants state that the node pointer points to the next node in the list, but 82% of them do not say the value can be `NULL` when at the end of the list. Some of these same participants, such as Ecer, also do not mention or draw `NULL` when describing a singly linked list, which carries over as a problem when inserting and deleting a new node at the end of the list.

Although only Ecer fails to assign the new node's next node pointer to `NULL` when inserting a node at the end, Joe, Suzy, Max, and Ecer fail to set the new last node to `NULL` when deleting the last node in the list, and Joe believes that the new last node will automatically point to `NULL`. These students, except Max, also fail to check whether the list is empty when deleting at the beginning.

We see this issue again when asked to describe the head pointer. We find that no one explicitly mentions how the head points to NULL when the list is empty, which is a misunderstanding that continues to be seen when coding an empty list. We find that Joe, Suzy, and Phil fail to assign the head pointer to NULL. Phil creates a head node pointer but does not set the node pointer to NULL. In contrast, Joe and Suzy create a head node pointer and set it to point to one node for an empty list. This indicates that these students do not deeply understand the benefit of `NULL`.

C. Confusion Around Data as a Void Pointer

There are some students who unnecessarily create a void pointer or void type for the data member of a node, which only needed to store an integer. This causes confusion about how to store an integer value in the void pointer. For example, Bill, Ecer, and Nate try to assign a literal integer value to the void pointer instead of an address. Bill thinks that it would work better if he typecasts the void pointer to match the data type of the information being stored by writing the following.

```
(int)new_node->value = 6;
```

Even though Bill understands that these two types need to match, it is not correct to typecast the operand on the left side of the assignment operator. The student should assign the address of where an integer is in memory to the void pointer variable. These students confuse themselves more by creating a void pointer for the data component, when they only needed

to store an integer value. It is a waste of memory to use this style of implementation for storing the address of a single value.

D. General Lack of Prerequisite Knowledge

Some participants lack the prerequisite knowledge about pointers and memory management needed to reason well about linked list concepts. For example, creating a linked list requires understanding how to dynamically allocate memory for a node, which requires a good understanding of how to use the `malloc()` function in C. The majority of the participants lack knowledge regarding typecasting the address returned by malloc and struggle with what to send to the sizeof() function.

When dynamically allocating a new node in C, it is good practice to typecast the address of the allocated node returned by malloc to a node pointer, i.e. `struct node*`, even though it is not required. None of the participants initially typecast when writing code to create a new node. Two students (Bill and Phil) add a typecast to their malloc after receiving an error message using a C++ compiler, instead of the C compiler. When we show students the syntax for typecasting malloc to a node pointer, we find that Max, Phil, and Ecer mention they have never seen the `(struct node*)` typecast before.

In addition to understanding how to allocate memory, students must understand pointers well. Many students do not seem to have a deep understanding of pointers. Just as students lack knowledge about typecasting the void pointer returned by malloc, students do not assign the address returned by malloc to a pointer.

For example, Chemi and Ecer send the list pointer to the insert function, but they do not make the parameter a pointer to a list (see Figure 5). Chemi and Ecer know to set the head pointer variable to the new node, but they do not make the parameter a `struct list *l`, which is similar to what you see again with forgetting to make a pointer variable to store the address returned by malloc on line 21 (see Figure 5)

While we did not ask the students to write functions for their operations, most students chose to write functions. When adding or deleting a node at the beginning of a linked list, the head value needs to be updated to point to the new node, and when writing a function for these operations, the function needs to either receive the address of the head pointer or return the new node's address to update the head pointer. However, all the students store the head node pointer on the heap, rather than the stack. Therefore, the students do not need to pass the address of the list or head pointer variable. Passing the pointer variable passes the address of where their head pointer variable is.

```
20 void insert_front (struct list l, int value){
21     struct node new_head= malloc(sizeof(struct node));
22     new_head->val = value;
23     new_head->next = l->head;
24     l->head = new_head;
25 }
```

Fig. 5. Chemi's response to adding a new node at the beginning operation.

However, we find that Suzy, Chemi, and Ecer fail to update the head pointer variable because of a mismatch between the type a function is supposed to return or receive as input and the type of value being returned or provided as an argument. Suzy returns the address of the new node inserted in the linked list, but the function return type is int, which means it returns an integer value, instead of a node pointer. When Suzy starts coding, she asks whether the function parameters are essential to include. We state that they need to have it, but she totally dismisses the reply and does not include them. These students struggle with making their function return types and argument/parameter types match.

We also see that another student (Feng) has a misunderstanding of types used in function calls. The student writes `insert_end(struct link_list* list, int val);` when calling the insert at the end function. Surprisingly, Feng also struggles with writing a condition for comparing the head to `NULL`. Feng writes `if(list->head) printf(''empty'');`. Since NULL is zero, then this is false when the list is empty and true when not empty. Sadly, students at this stage of studying CS are still not completely thinking through basic programming concepts, which can cause other serious problems moving forward.

VII. Conclusion

In this study, we measure the accuracy of undergraduate students' mental models of a singly linked list by assessing and evaluating their conceptual and/or procedural understanding of five components and 14 operations. In a semi-structured think-aloud interview, we ask students to describe a singly linked list and five components, and then we ask students to code and recognize fourteen essential singly linked list operations.

We find that none of the students have an accurate mental model of a singly linked list. Students do not address every aspect of the linked list concepts in their verbal responses to general questions. They highlight common knowledge and give high-level information, but they do not provide very detailed information. In addition, students do not focus on edge cases, such as an empty list and the importance of NULL, and this carries across coding questions for some students. Students need to think more broadly and in more detail like an expert. On the other hand, students still struggle with prerequisite knowledge related to allocating memory with `malloc`, pointers, and matching types in functions that impact their ability to understand and implement a singly linked list.

The education philosophers suggest that educators make sure that the materials they teach are correct and do not have any misleading aspects [40], but it is hard to do this without an inventory of concepts required for completely understanding linked lists. This research contributes a categorization of essential linked list concepts and a framework for assessing and evaluating students' mental models about a singly linked list. As we can see from this initial investigation, above average students face difficulties with understanding and implementing linked lists in C, which is primarily due to misunderstanding about pointers and memory management.

REFERENCES

[1] E. Fouh, M. Akbar, and C. A. Shaffer, "The role of visualization in computer science education," *Computers in the Schools*, vol. 29, no. 1-2, pp. 95–117, 2012.

[2] R. L. Kruse, C. L. Tondo, and B. P. Leung, *Data Structures and Program Design in C*. USA: Prentice-Hall, Inc., 1996.

[3] 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*. ACM, 2012, pp. 21–26.

[4] W. Paul and J. Vahrenhold, "Hunting high and low: instruments to detect misconceptions related to algorithms and data structures," in *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 2013, pp. 29–34.

[5] K. Karpierz and S. A. Wolfman, "Misconceptions and concept inventory questions for binary search trees and hash tables," in *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 2014, pp. 109–114.

[6] D. Zingaro, C. Taylor, L. Porter, M. Clancy, C. Lee, S. Nam Liao, and K. C. Webb, "Identifying student difficulties with basic data structures," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 2018, pp. 169–177.

[7] R. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy, "Recursion vs. iteration: An empirical study of comprehension revisited," in *Proceedings of the 46th ACM technical symposium on computer science education*. ACM, 2015, pp. 350–355.

[8] 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," in *Proceedings of the 49th ACM technical symposium on Computer Science Education*. ACM, 2018, pp. 858–863.

[9] G. S. Halford, *Children's understanding: The development of mental models*. Hillsdale, NJ: Erlbaum, 1993.

[10] D. P. Newton, "Causal situations in science: a model for supporting understanding," *Learning and Instruction*, vol. 6, no. 3, pp. 201–217, 1996.

[11] L. D. Henderson and J. Tallman, *Stimulated recall and mental models: Tools for teaching and learning computer information literacy*. Scarecrow Press, 2006, vol. 2.

[12] R. A. Zwaan, "Situation models, mental simulations, and abstract concepts in discourse comprehension," *Psychonomic bulletin & review*, vol. 23, no. 4, pp. 1028–1034, 2016.

[13] P. N. Johnson-Laird, *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983, no. 6.

[14] L. W. Anderson, D. R. Krathwohl *et al.*, "A revision of bloom's taxonomy of educational objectives," *A Taxonomy for Learning, Teaching and Assessing. Longman, New York*, 2001.

[15] J. Carter and P. Dewan, "Contextualizing inferred programming difficulties," in *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering*, 2018, pp. 32–38.

[16] S. Fincher and M. Petre, *Computer science education research*. CRC Press, 2004.

[17] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers," *Acm sigcse bulletin*, vol. 37, no. 3, pp. 14–18, 2005.

[18] C. Izu, A. Weerasinghe, and C. Pope, "A study of code design skills in novice programmers using the solo taxonomy," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 2016, pp. 251–259.

[19] M. Piteira and C. Costa, "Computer programming and novice programmers," in *Proceedings of the Workshop on Information Systems and Design of Communication*, 2012, pp. 51–53.

[20] ——, "Learning computer programming: study of difficulties in learning programming," in *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, 2013, pp. 75–80.

[21] B. J. Smith and H. S. Delugach, "Work in progress 2014; using a visual programming language to bridge the cognitive gap between a novice's mental model and program code," in *2010 IEEE Frontiers in Education Conference (FIE)*, Oct 2010, pp. F3G–1–F3G–3.

[22] J. Dewey, *How we think*. Courier Corporation, 1997.

[23] R. Gelman and C. R. Gallistel, *The child's understanding of number*. Harvard University Press, 1978.

[24] D. A. Norman, "Some observations on mental models," in *Mental models*. Psychology Press, 1983.

[25] C. Howe, A. Tolmie, A. Anderson, and M. Mackenzie, "Conceptual knowledge in physics: The role of group interaction in computer-supported teaching," *Learning and Instruction*, vol. 2, no. 3, pp. 161–183, 1992.

[26] A. Newell, H. A. Simon *et al.*, *Human problem solving*. Prentice-Hall Englewood Cliffs, NJ, 1972, vol. 104, no. 9.

[27] Y. Anzai and H. A. Simon, "The theory of learning by doing." *Psychological review*, vol. 86, no. 2, p. 124, 1979.

[28] R. R. Hoffman, S. T. Mueller, G. Klein, and J. Litman, "Metrics for explainable ai: Challenges and prospects," *arXiv preprint arXiv:1812.04608*, 2018.

[29] J. Rasmussen, A. M. Pejtersen, and L. P. Goodstein, "Cognitive systems engineering," 1994.

[30] M. D. Williams, J. D. Hollan, and A. L. Stevens, "Human reasoning about a simple physical system," in *Mental models*. Psychology Press, 2014, pp. 139–162.

[31] S. Frederick, "Cognitive reflection and decision making," *Journal of Economic perspectives*, vol. 19, no. 4, pp. 25–42, 2005.

[32] K. D. Lippa, H. A. Klein, and V. L. Shalin, "Everyday expertise: cognitive demands in diabetes self-management," *Human Factors*, vol. 50, no. 1, pp. 112–120, 2008.

[33] N. Praetorius and K. Duncan, "Verbal reports: a problem in research design," in *Tasks, errors, and mental models*, 1988, pp. 293–314.

[34] R. R. Hoffman and P. A. Hancock, "Measuring resilience," *Human factors*, vol. 59, no. 4, pp. 564–581, 2017.

[35] A. B. Champagne *et al.*, "Interactions of students' knowledge with their comprehension and design of science experiments." *A Technical Report at the University of Pittsburgh*, 1980.

[36] L. McDermott, "Research on conceptual understanding in mechanics," *Physics Today*, vol. 37, no. 7, pp. 24–32, 1984.

[37] I. A. Halloun and D. Hestenes, "The initial knowledge state of college physics students," *American Journal of Physics*, vol. 53, no. 11, pp. 1043–1048, 1985.

[38] K. A. Ericsson and H. A. Simon, *Protocol analysis: Verbal reports as data.* the MIT Press, 1984.

[39] D. Gentner and A. L. Stevens, *Mental models*. Psychology Press, 1983.

[40] N. Noddings, *Philosophy of education*. Routledge, 2018.