

AN ABSTRACT OF THE DISSERTATION OF

Eman Almadhoun for the degree of Doctor of Philosophy in Computer Science
presented on December 17, 2021.

Title: Exploratory Study to Uncover Student Mental Models of Singly Linked Lists in
the C Programming Language

Abstract approved: _____

Jennifer Parham-Mocello

In computer science, learning abstract fundamental programming concepts requiring students to understand memory management can be very difficult and lead to misunderstandings that carry on into advanced topics. This is especially true in data structures with abstract data types. Understanding how novice students think and reason about data structures is important for improving teaching and learning in computer science. Most studies focus on student misunderstandings of advanced algorithms and data structures related to topics such as heaps, binary search trees, hash tables, dynamic programming, and recursion. Whereas, fewer studies focus on more elementary data structures, such as arrays and linked lists.

Since linked lists serve as a bridge to understanding more advanced data structures, we believe that it is critical to identify students' conceptual and procedural misunderstandings earlier rather than later. Therefore, directly after learning about linked lists using the C language, we conduct semi-structured, think-aloud interviews with 11 undergraduate students to uncover their mental models about singly linked lists in C, and how they apply their knowledge about singly linked lists to understand other types of linked lists. To determine the factors that might contribute to their understanding about linked lists, at the end of the interview, we ask students about the difficulties they had while learning about linked lists, and we give students a 10-minute visual-spatial reasoning test. Using rubrics to code responses to the interview questions, we quantify the accuracy of

their mental models and reveal common misunderstandings, such as confusion between the node containing a node pointer and being a node pointer, failure to create a node structure, lack of knowledge regarding typecasting malloc, and lack of attention to the importance of the NULL value.

We find that none of the participants have an accurate mental model of a singly linked list in C, after learning about and implementing them in their data structures course. Even though students struggle with expressing their conceptual understanding with enough detail in their verbal responses to interview questions, their performance on the coding questions is much better. The majority of students have a good procedural understanding of how to create and use the pieces of a linked list, and they understand how to implement most operations on a singly linked list in C. However, many students continue to struggle with C syntax and the prerequisite knowledge needed to understand linked lists in C, such as pointer manipulation and memory management. While we do not find a relationship between students' visual-spatial ability and their conceptual or procedural understanding of linked lists in C, students report that the abstract nature of pointers and relating linked lists to the real world and prior knowledge about dynamic arrays contribute the most to their difficulties with linked lists in C.

©Copyright by Eman Almadhoun
December 17, 2021
All Rights Reserved

Exploratory Study to Uncover Student Mental Models of Singly
Linked Lists in the C Programming Language

by

Eman Almadhoun

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 17, 2021

Commencement June 2022

Doctor of Philosophy dissertation of Eman Almadhoun presented on
December 17, 2021.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Eman Almadhoun, Author

ACKNOWLEDGEMENTS

First of all, I would like to thank Allah, the Almighty, for His generosity of blessings and giving me the power and patience throughout my research to complete it successfully.

Then, I would like to express my deep and sincere gratitude to my advisor, Professor Jennifer Parham-Mocello, for being my advisor and providing invaluable guidance throughout my dissertation. It was a great privilege and honor to work under her supervision. Her vision, sincerity, and motivation have deeply inspired me. I would also like to thank her for her friendship, kindness, and empathy. I am extending my deepest thanks to her family for their acceptance and patience during the discussions and the Zoom calls I had with her.

I also thank my committee members, Professors Margaret Burnett, Mike Bailey, Christopher Sanchez, and Maggie Niess for their helpful discussions, insights, and feedback.

I extend my thanks to all graduate writing center consultants for their help and support in scheduling and writing, especially Adam Haley for working with me during my whole academic journey. I also thank Aiden Nelson, Kaitlin Hill, Paris Kalathas, Mahsa Saeidi, and Abdullah Azzouni for helping me review my writing and giving me valuable feedback.

I am very thankful to my parents for their love, prayers, and care. I am extremely grateful to my beloved husband, Hasan, and my wonderful kids, Tamara, Yara, and Amer, for their love, understanding, prayers, and continued support to complete this research. Also, I express my thanks to all my sisters and brother for their support and valuable prayers.

Finally, thank you to all my friends and neighbors who supported and encouraged me in the toughest situations that I have ever experienced before, especially to Abrar Fallatah and Mahsa Saeidi.

TABLE OF CONTENTS

	<u>Page</u>
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Statement	2
1.3 Research Questions	3
1.4 Proposed Contributions	3
2 Background and Literature Review	5
2.1 Learning Abstract Concepts	5
2.2 Why Mental Models	6
2.3 Misconception Research	7
2.4 Motivation for a Linked List Concept Inventory	8
2.5 Reasoning and Spatial Visualization	10
3 Research Method	12
3.1 Categorization of Linked List Concepts:	12
3.1.1 Linked List Concepts:	14
3.2 Linked List Framework	18
3.3 The Survey and Semi-Structured Interview Questions:	20
3.3.1 Collect Expert Feedback and Revision:	21
3.3.2 Mapping Survey and Interview Questions to the Linked List Concepts	22
3.4 Data Collection:	25
3.5 Evaluation and Data Analysis:	28
4 Results and Discussions	34
4.1 RQ1: What are students' mental models of linked lists in the C programming language, and how accurate are their mental models?	35
4.1.1 RQ1.1: How accurate are students' mental models about the types and pieces of linked lists and operations on linked lists?	36
4.1.2 RQ1.2: What are students' misunderstandings or gaps in knowledge about the types and pieces of linked lists and operations on linked lists?	42

TABLE OF CONTENTS (Continued)

	<u>Page</u>
4.2 RQ2: What difficulties do students face while learning about linked lists in the C programming language?	103
4.3 RQ3 & RQ4: The Purdue Visualization of Rotations Test (ROT)	106
4.3.1 RQ3: What is the relationship between students' understanding about linked lists and their visual-spatial reasoning?	107
4.3.2 RQ4: What is the relationship between drawing pictures while reasoning about linked lists and students' visual-spatial reasoning?	108
5 Threats to Validity	111
6 Conclusion	112
7 Future Work	114
Bibliography	115
Appendices	126
A Survey Materials	127
B Verbal Questions	143
C Coding Questions	159
D Recognition Questions	186
E Participants' Grading in the Survey	191
F Scores Per Categories in the Survey for Interviewed Participants	194

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
3.1 Linked List Concept Categories	13
3.2 Linked List Framework illustrating the relationship among prerequisite knowledge, a singly linked list, and applying this knowledge to different linked lists.	19
3.3 Demographics and backgrounds of 40 consenting participants in the survey.	27
3.4 Example rubric for an open-ended survey question with coding for participants Joe and Bob.	29
3.5 Example rubric for a verbal interview question with coding for participant Joe.	31
3.6 Grading participant Xeng on the coding interview question about creating an empty list using the generated coding rubric.	32
3.7 Grading Chemi's explanation about code changing when adding a tail pointer variable using the explanation code rubric.	33
4.1 Percentage of students with the correct answer on each survey question. Note: Q21 is out of 10 participants, and Q22-28 are out of 27 participants.	36
4.2 Survey Question 23: Identify the meaning of a pointer variable declaration.	44
4.3 Question 24: Identify the legality of pointer variable assignments in the survey.	45
4.4 Question 25: Identify the legality assignments of the pointer variable in the survey.	45
4.5 Question 28: Identify the code's output about pointer variable manipulations and dereferencing using asterisk and ampersand operators in the survey.	46
4.6 Survey Question 1: Identify a node from a picture.	49
4.7 Survey picture of a singly linked list with a head and a tail pointer variables used in Q4-Q7.	49
4.8 Suzy (left) and Feng (right) node structure.	52

LIST OF FIGURES (Continued)

<u>Figure</u>	<u>Page</u>
4.9 Ecer's node structure.	52
4.10 Suzy (top) and Nate (bottom) responses to adding a new node at the beginning operation.	57
4.11 Feng's response to inserting a new node at the beginning of a list.	58
4.12 Suzy's response to creating an empty list.	61
4.13 Joe's response to creating an empty list.	62
4.14 Chemi's response in coding empty list operation.	63
4.15 Ecer's response to creating an empty list.	63
4.16 Chemi's response to adding a new node at the beginning operation.	64
4.17 A visual representation of an empty list. Note that the electrical ground symbol at the end represents the NULL value.	66
4.18 Nate's response to checking if a list is empty operation.	67
4.19 Suzy's response to checking if a list is empty operation.	67
4.20 The diagram of a singly linked list presented in the survey and used in Q21.	68
4.21 Question 21: Recognizing the steps for clearing a singly linked list.	68
4.22 Recognition question 4 for clearing the linked list nodes.	69
4.23 Joe's response to adding a new node at the beginning operation.	71
4.24 Xeng's response to adding a new node at the beginning operation.	71
4.25 A singly linked list provided to the participants in coding question 6.	72
4.26 Joe's response to adding a node at the end of a singly linked list.	73
4.27 Suzy's response to adding a node at the end of a singly linked list.	73
4.28 Feng's response to adding a node at the end of a singly linked list.	74
4.29 Joe's response to adding a new node at a specific location in a singly linked list.	75

LIST OF FIGURES (Continued)

<u>Figure</u>	<u>Page</u>
4.30 Max's response to adding a new node at a specific location in a singly linked list.	76
4.31 Suzy's response to adding a new node at a specific location in a singly linked list.	76
4.32 Ecer's response to adding a new node at a specific location in a singly linked list.	77
4.33 Nate's response to deleting a node at the beginning operation.	78
4.34 Joe's response to deleting the last node in a singly linked list.	78
4.35 Phil's response to deleting the last node in a singly linked list.	79
4.36 Suzy's response to deleting a node in the middle of a singly linked list. . .	80
4.37 Ecer's response to deleting a node at the middle of a singly linked list. . .	81
4.38 A singly linked list provided to the participants in recognition questions. .	83
4.39 Recognition question 1 for swapping two adjacent nodes in a singly linked list.	84
4.40 Bill's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.	85
4.41 Xeng's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.	85
4.42 Chemi's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.	87
4.43 Suzy's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.	87
4.44 Max's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.	88
4.45 Ecer's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.	89
4.46 Feng's response to finding the length of a linked list.	91

LIST OF FIGURES (Continued)

<u>Figure</u>	<u>Page</u>
4.47 Suzy's response to finding the length of the linked list.	92
4.48 Chemi's response to finding the length of the linked list.	92
4.49 Recognition question 2 for finding the x value in a singly linked list.	93
4.50 Recognition question 3 for printing the values stored in the singly linked list nodes.	94
4.51 Question 4: Identify a singly linked list type.	97
4.52 Question 12: Identify a singly circular linked list type.	99
4.53 Question 13: Identify a doubly linked list type.	101
4.54 Question 11: Identify a doubly circular linked list type.	102
4.55 Suzy's sketches while describing a doubly circular linked list type.	103

LIST OF TABLES

Table	Page
3.1 A comprehensive list of prerequisite knowledge and linked list concepts with their definitions.	17
3.2 Mapping linked list concepts to the corresponding survey and interview questions in which they appear. Note that S-Qx refers to survey question, IV-Qx refers to interview verbal question, IC-Qx refers to interview coding question, IE-Qx refers to interview explanation question, IR refers to interview recognition, SLL refers to singly linked list, SCLL refers to singly circular linked list, DLL refers to doubly linked list, and DCLL refers to doubly circular linked list.	25
3.3 Demographics of 11 Consenting Interview Participants. Note: PID (Participants' ID), LL (Linked List), CS (Computer Science), Chem. Eng. (Chemical Engineering), ECE (Electrical & Computer Engineering), and PSU (Portland State University).	28
4.1 Each participants' percentage of points earned on each question from the verbal, coding, and recognition sections of the interview. Note: Light-red (< 80%), blue (80-89%), gray (90-99%), and green (100%).	38
4.2 Scores per linked list concept based on conceptual (light-purple) and procedural (orange) understandings.	40
4.3 Overall score for each linked list category.	41
4.4 Spatial Visualization Test Score. Note that N is the sample size, M is the mean, Min. is the minimum score, and Max. is the maximum score. . . .	106
4.5 Students accuracy score per section, and ROT score (ROT percentage score in the parenthesis). Note that the presented data is organized based on the high to low scores in the overall linked list concepts.	108
4.6 The Number of students' drawings per section, and ROT score (ROT percentage score in the parenthesis). Note that the presented data is organized based on the high to low scores in the overall linked list concepts.	109

LIST OF APPENDIX TABLES

<u>Table</u>		<u>Page</u>
E.1	Participants' scores on each question from in the survey. The green check-mark for the correct answer and the red cross-mark is for the wrong answer. Note that the highlighted results are the results for the 11 participants we interviewed in semi-structured interview.	193
F.1	Scores per linked list concept in the survey based on conceptual (light-purple) understanding.	195

Chapter 1: Introduction

1.1 Motivation

Learning and teaching data structures is 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 [32], and yet, understanding algorithms and data structures are some of the most important courses for computer science (CS) undergraduate students because they serve as a foundation for their upper-division courses.

There are prior studies advancing educators' knowledge on student misconceptions about advanced data structures, such as heaps, binary trees, and hash tables [22, 64, 85], but much of the prior work is missing the important subject of basic data structures, like arrays and linked lists [76, 120]. We focus on linked lists because they serve as a bridge to understanding more advanced data structures [89]. 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 pointer variables, as with doubly linked lists, a binary tree has a node with left and right pointer variables.

To address this gap in the literature, we conduct semi-structured, think-aloud interviews with 11 students to uncover their reasoning and misunderstandings about singly linked lists. This initial exploratory case study aims to empirically identify students' lack of knowledge and difficulties with understanding and implementing singly linked lists, which is typically the first data structure taught after arrays. This includes students' struggles with syntax and the prerequisite knowledge needed to conceptually and procedurally understand and implement 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 [4].

In particular, our study focuses on singly linked lists in the C programming language, which is the language used at 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 [67]. Typically, the simplest singly linked list is implemented with a head pointer variable on the stack that either points to a node on the heap that begins the list or is NULL if the list is empty. Each node is an instance of a user-defined struct that contains list data of an appropriate type (int for a list of integers, char * for a list of strings, etc.) and a node pointer, which either points to the next node in the list or is NULL if the node is the last one in the list. In addition to the concepts of a node and head pointer variable, students' need to conceptually and procedurally understand the node's two members for a total of four basic pieces of a list: 1) the head pointer variable and 2) the node with 3) the list data and 4) a node pointer. We recognize that many implementations include a tail pointer variable, but we only ask students conceptual questions about the tail pointer variable because they do not implement a list in this way at their university.

Cognitive science uses the term "mental models" to describe a cognitive representation of an abstract structure or a part of an abstract structure of the real-world, situations, events, tasks, problems, procedures, concepts, activities, or phenomena and the relations between them [43, 81, 49]. Mental models are very important for learning a new concept, especially abstract concepts [121]. According to Johnson-Laird, mental models help us understand problems, find a suitable solution, and predict outcomes based on actions [59]. 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 [49] and reasoning [59].

Understanding how students reason about a concept provides insight into their mental model about the concept [59]. However, exploratory research on students' mental models of linked lists in CS is not well-known. Therefore, 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.

1.2 Thesis Statement

Based on the lack of research on students' mental models about linked lists in CS, we construct the following thesis statement as the basis for our research in this exploratory study. *Computer science undergraduate students struggle with linked lists in the C language, due to having inaccurate mental models with misunderstandings about linked lists*

concepts and low visual-spatial reasoning to visualize abstract concepts.

1.3 Research Questions

In order to support our thesis statement, we answer the following specific research questions.

1. What are students' mental models of linked lists in the C programming language, and how accurate are their mental models?
 - (a) How accurate are students' mental models about the types and pieces of linked lists and operations on linked lists?
 - (b) What are students' misunderstandings or gaps in knowledge about the types and pieces of linked lists and operations on linked lists?
2. What difficulties do students face while learning about linked lists in the C programming language?
3. What is the relationship between students' understanding about linked lists and their visual-spatial reasoning?
4. What is the relationship between drawing pictures while reasoning about linked lists and students' visual-spatial reasoning?

To answer our research questions, we use a mixed-methods approach with quantitative and qualitative data. The results of this study help us determine how students reason about linked lists in a data structures course and identify the misunderstandings that students have that lead them to struggle while learning about linked lists in the C language. This includes students' struggles with syntax and the prerequisite knowledge needed to conceptually and procedurally understand linked lists in C. In addition, we investigate correlations between students' reasoning about linked lists and their visual-spatial reasoning.

1.4 Proposed Contributions

In this thesis, we present the following key contributions from this exploratory research study:

- Develop a categorization and a comprehensive list of linked list concepts.
- Develop a set of questions along with example rubrics for assessing a person's conceptual and procedural understandings about singly linked lists in C.
- Measure the accuracy of students' mental models about linked list.
- Identify misunderstandings and gaps in knowledge about linked list concepts.
- Explore connections between students' mental models about linked lists and visual-spatial reasoning.

Chapter 2: Background and Literature Review

In this research study, we evaluate students' conceptual and procedural understanding to measure the accuracy of their mental model and explore how they reason about linked lists in a data structures course. We believe that constructing correct mental models of algorithms and data structures, as well as knowing how to use them, are vital cognitive processes that help students efficiently solve problems on their own. This work builds upon prior research on learning abstract concepts, mental models, misconceptions, concept inventories, and visual-spatial reasoning to gain a better understanding of students' conceptual and procedural understanding of linked list concepts in the C programming language.

2.1 Learning Abstract Concepts

Learning computer programming for CS undergraduate novice students is challenging [12, 29, 69, 58, 86, 87]. This is because computer programming consists of many abstract concepts, instructions, and processes that one needs to follow [103]. Students may feel frustrated and unable to continue to learn to program because they do not have a good visualization of the flow of these processes, and past research shows that visualization is especially critical for algorithms and data structures [29].

Data structures are abstract containers used to store, organize, and access data efficiently, while an algorithm contains the processes or steps that are followed to solve a specific problem [29]. 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 thinking needed in order for a learner to understand the concept [24]. "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 [4]. 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” [4]. Conceptual and procedural understanding are not independent, and many philosophers studying theories of knowledge claim that procedural understanding is based on conceptual understanding [35, 43], and likewise, conceptual knowledge is built upon factual knowledge. Therefore, we ask students’ about their conceptual understanding of the pieces that make up a singly linked list, and then we observe their procedural understanding of using these pieces in common linked list operations to better understand students’ mental models.

2.2 Why Mental Models

Norman defines mental models as internal representations built on individual experiences in the real world [83]. Mental models can be naïve or immature [57] and can be inaccurate and sometimes incomplete [59, 43]. Henderson and Tallman [49, 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 [59].

Novices who have limited experience and knowledge in solving problems construct partial mental models or map prior mental models to the new situation [49]. For example, when novice students want to find errors in a program, they will try all possible sequences to find the errors, 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 [80, 5]. Therefore, we believe it is important to build accurate mental models of abstract concepts to help individuals correctly solve problems.

Many methods elicit students’ mental models by asking students to talk about their understandings or explain them correctly [56]. These methods include having students think-aloud while performing a task [94, 114], writing a “Task Reflection” after solving a coding problem [33, 73, 91], and diagramming tasks to create a visualization based on

their prior knowledge of the process [55]. All of these research methods produce qualitative data, but some methods gain a richer understanding of the participants' mental models and the way they are reasoning and thinking than others.

2.3 Misconception Research

In CS, many studies identify students' misconceptions in areas outside data structures, including operating systems [110], computer architecture [88], discrete mathematics [3], digital logic [51, 53], and algorithms [11, 27, 62, 117, 46]. Other studies focus on students' misconceptions of advanced topics in data structures, such as heaps, binary trees, and hash tables [22, 85, 64]. However, there are not many studies on misconceptions about linked lists, except for one recent study identifying student difficulties with learning about basic data structures including ArrayLists, singly and doubly linked lists, and binary search trees [120].

Zingaro et al. focuses on students' procedural understanding of specific operations on lists, and the authors collect data from 249 students in an exam study session for a Java CS2 course. When the authors ask students to add a node to the end of a list, they find that 16% of the students did not update the tail reference referring to the last node in the list, 12% wrongly attached the new node to the list, and 10% iterated through all the nodes in the list to find the last node, rather than simply using the tail reference [120]. Zingaro et al. also reveals that some students believed that the doubly linked list can be searched in both directions in parallel, and students memorized the concepts of the data structures, rather than recalling and knowing how to use these concepts for each data structure [120].

Even though Zingaro et al. show that students make common mistakes when working with linked lists, they do not measure the dynamic process of the students thinking and reasoning about linked lists in real time [120]. We believe that observing students' thinking related to linked lists, pointer variables, and memory management in C can provide researchers with a deeper understanding of students' mental models. In addition, Zingaro et al. research on students' understanding of linked lists does not ask the students fundamental conceptual questions about linked lists in C, such as what a node and a node pointer are, what the benefits of having a tail pointer variable are, etc., and we believe that students have misunderstandings or are unsure about the basic pieces of linked

lists, which are needed to understand operations on linked lists and more complex data structures.

There are various methods for identifying misconceptions, which include analysis of 1) exam papers with think-aloud interviews from the instructor and/or students' [3, 11, 22, 64], 2) general interviews with students [53], 3) think-aloud interviews with students [51, 62, 117] 4) the combination of think-aloud interviews and a pilot of the Concept Inventory (CI) with students [52], and 5) final exam study sessions [120]. In this research study, we examine students' misunderstandings about the basic concepts of linked lists using semi-structured think-aloud interviews asking students to verbally define these concepts, as well as implement them. Since we do not consider students' preconceived notions or mistaken beliefs, we avoid using the term "misconceptions" in this dissertation, and instead use the term "misunderstandings".

2.4 Motivation for a Linked List Concept Inventory

Concept inventories (CI) are one type of assessment tool used to evaluate students understanding of concepts in a particular topic [37]. A CI is a standardized multiple-choice test used to help instructors identify concepts that students do not understand and what their misconceptions are [100]. Only one option is correct in each question, and the other options are distractors from student misunderstandings identified by a long history of qualitative studies, such as the Force Concept Inventory (FCI) [54]. Hestenes' FCI is used in physics to examine post-secondary students' understanding about force and motion concepts. The researchers find that most students could state Newton's Third Law and only a few of them fully understood it [54].

There are CIs in CS for discrete mathematics [3], digital logic [51, 53], and data structure topics related to heaps and binary search trees [22, 108], and more recently there is a CI developed for basic data structures including lists, trees, stacks, and sets with a heavy emphasis on lists and trees [90]. While there is a need to develop a comprehensive CI for linked lists in general and specific to languages, there is a lack of qualitative studies revealing student conceptual and procedural misunderstandings about linked lists to create such inventories. Within the computer science education literature, there are many research papers on visualization tools [115, 23, 70, 95, 96, 34, 31, 79, 10, 41, 68] and pedagogical techniques [47, 42, 9, 99, 119, 7, 38, 19, 48, 63, 93] to help students learn

linked lists better. However, there are very few research studies focusing on how students think about linked lists, such as their difficulties, misunderstandings, misconceptions, and mental models [120, 109, 15, 76, 90].

Some of the research studies on linked lists measure how students' performance, understanding, or motivation improves after using a visualization tool or new pedagogical technique, such as active learning, multimedia, or games, with pre- and post-tests, homework, or student evaluations [115, 31, 119, 63, 93, 48, 47], but these studies do not discuss the concepts students have trouble understanding. In addition, most of the tools and techniques are for helping students learn linked lists in Java [23, 119, 70, 95, 96, 34, 31], Python [79, 68, 41], or without any programming using schematic diagrams [38]. While there are some tools focused on learning linked lists in C [115, 10] or a combination of C and Python [68, 41], the developers of the tools do not provide students' misunderstandings about linked lists in C that provide the foundation for creating the tool.

There are some research papers that discuss students' difficulties or misunderstandings about linked list concepts, but they do not provide a comprehensive view of students' mental models about linked lists in C [120, 109, 15, 76, 90]. One study introduces the idea of exploring students' mental models about linked lists and suggests that students understand the general concept of linked lists and struggle with pointers [15], but the study is only one page with little detail. A study the year before compared students views of using arrays and linked lists in C versus Java [109], but the author only measures the time students spend on their implementations and student perspectives of what they thought was different and similar between their Java and C implementations. Another study repeating a study from 1996 shows how students think about recursive versus iterative code for searching and copying linked lists in Java and Pascal [76], but the study does not use C and only reports on how easily students recognize iterative versus recursive code for two operations.

Zingaro et al. is the closest related research describing students' difficulties thinking about a tail pointer, runtime, and iterating when adding a node to the end of a linked list [120], which later they use to create questions for their Basic Data Structures Inventory (BDSI) [90, 111]. However, in all of these examples of research on linked lists in CS education, none 1) present a comprehensive list of prerequisite knowledge and linked list concepts in the C programming language and 2) systematically measure students' conceptual and procedural understanding of all the concepts and prerequisite knowledge about

pointers and memory management to gain insights into students' mental models about linked lists in C. Therefore, the broader goal of this research study is to contribute qualitative information about students' conceptual and procedural understanding of linked lists in C that CS education researchers can use for the creation of a linked list CI.

2.5 Reasoning and Spatial Visualization

Teaching and learning CS involves drawing and sketching to support understanding through visualization of the program execution [104]. Many students start solving programming problems by drawing pictures or drawing code traces [21]. The pictures represent how the students think and the strategy used to solve the problems [50]. We believe drawing pictures is related to the students' spatial ability.

Researchers show that spatial reasoning is the cognitive ability to understand the relationship between objects in space [60], and spatial skills are important for navigating the real world and abstract information [113, 65]. In computer science education, researchers show that spatial ability is an important factor in program comprehension [25, 20, 61]. When programmers are trying to comprehend a program, they construct very high cognitive skills and structure [17] to support understanding by adapting and combining existing skills [16]. Programmers seem to use mental models and spatial imagery when coding programming concepts [39] and to describe the code's purpose, operation, and abstractions [30].

Spatial skills also play a very important role in success in science, technology, engineering, and mathematics (STEM), including biology [98], chemistry [102, 8], physics [66], mathematics [14, 105], computer programming [60], design [72], engineering graphics [74], geometry [107], and engineering [2, 106]. One spatial skill is spatial visualization [40], which is an individual's ability to mentally rotate, fold or unfold flat objects and manipulate two- and three-dimensional stimulus objects (e.g. the performance of serial operations) in short-term memory [40, 78]. There is a well-known test in chemistry called the Purdue Visualization of Rotations (ROT) test for measuring an individual's visual-spatial ability [8].

The ROT test has 20 multiple-choice questions, and it must be completed in 10 minutes to restrict analytic processing. At the top of each question, there is an example picture of an object that is rotated in a specific direction, and the student is asked to

rotate a different object in the same manner. The test creators show a highly significant correlation between students' performance in introductory chemistry courses and this visual-spatial test [8]. The researchers find that students who did well on the visual-spatial test did well on the exam questions that required problem-solving skills and mental manipulations of two-dimensional representations of a molecule. Therefore, they claim that the test might be used to determine students who may have difficulty learning abstract concepts, solving spatial tasks, and visualizing a three-dimensional structure from a two-dimensional space, such as a drawing [8].

Several CS education studies also research the connection between spatial ability and programming success [18, 28, 97, 30, 60, 61, 75, 112]. Fincher et al. find a small positive correlation between CS students' grades and spatial skills across eleven post-secondary educational institutions, mainly in Australia [28]. Jones and Burnett conduct a study with students in a Masters IT course in the UK, and they find that students with high spatial ability completed code comprehension exercises faster than students with lower spatial ability [61]. Also, they find a strong correlation between spatial ability and results in programming modules, and one year later, Jones and Burnett find a correlation between visualization (mental rotation) skills and programming success [60], which aligns with many prior studies.

In 1984, Webb finds that spatial ability in students aged 11-14 was a predictor of how well a student could use basic Logo commands and create graphical Logo programs, after learning Logo programming for one week [112]. Two years later, Mayer et al. show that "success in learning Basic was related to general intellectual ability, especially logical reasoning, and spatial ability" [75]. Two decades later, Fisher, Cox, and Zhao conduct a study with both undergraduate and graduate students from Engineering, Science, and Computer Science with experience in Java, and they find that "similar cognitive skills are used for spatial cognition and program comprehension/development" [30].

Since a person's visual-spatial reasoning contributes to their ability to think abstractly [118], this may impact reasoning about linked lists. In this research study, we use the ROT test to assess the relationship between students' reasoning about linked lists and their ROT test scores. Using the ROT test will help us identify whether students' visual-spatial reasoning plays a role in how well they are able to understand and visualize linked lists.

Chapter 3: Research Method

The purpose of our study is to 1) examine how students think and reason about linked lists in data structures, 2) explore how well students understand the linked lists, 3) examine what misunderstandings students have about linked list concepts, and 4) determine if there is a relationship between understanding linked lists and visual-spatial reasoning to provide more insights into factors that play a role in students' reasoning and learning about linked list. We use observations of students solving problems in a semi-structured, think-aloud interview and a survey of their conceptual understanding to identify the quality of a student's mental model of linked lists in C. In the following sections, we provide a comprehensive inventory of linked list concepts in C and a framework for how the concepts relate to one another, and then, we present how we collected and evaluated the data.

3.1 Categorization of Linked List Concepts:

Before we discuss the linked list concepts, we present a categorization of the linked lists concepts in the C language. These four categories include 1) the prerequisite knowledge about pointer variables and memory that is required before learning about linked lists, 2) the pieces (or parts) of linked lists, 3) the different types of linked lists, and 4) the operations performed on linked lists (see Figure 3.1).

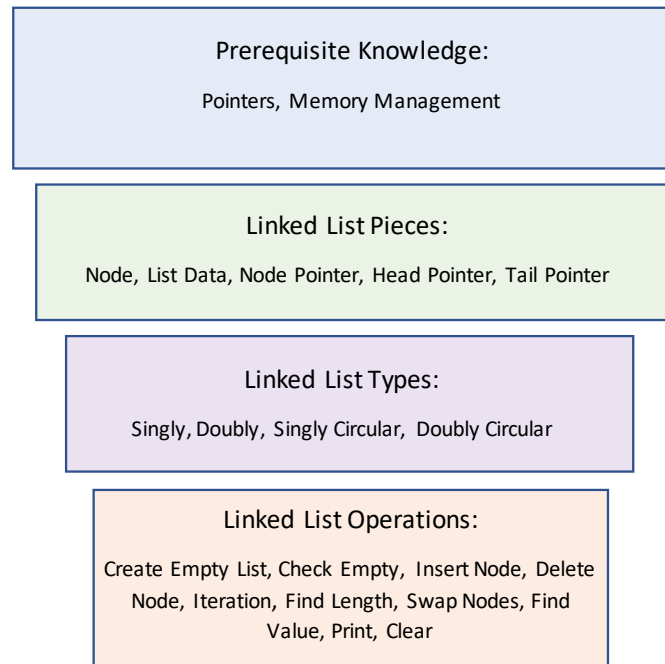


Figure 3.1. Linked List Concept Categories

The pieces of a linked list include 1) the node with 2) the list data and 3) the node pointer members, as well as 4) the head pointer variable and 5) the tail pointer variable. We realize that the tail pointer variable is not required, but we believe it is an important concept 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 to assess in the survey and interview, and we list each of these concepts within their corresponding category in Figure 3.1 and expand on each concept in detail in the following Section 3.1.1.

3.1.1 Linked List Concepts:

After categorizing a set of linked list concepts, we define what it means for a student to understand each concept to uncover how students reason about these concept after learning them in class. We consider every linked list concept in our study to help us fully examine students' mental models of linked lists.

Table 3.1 presents each concept with the concept number we use when referring to and analyzing the data, along with the definition of what it means for a student to understand each concept. Since students must understand every concept to have a correct mental model [59], we evaluate their prerequisite knowledge (PK1-PK10) and linked list concepts (C1 - C28) in the survey and interview.

Concept No.	Linked List Concept	Definitions of Student Understanding
Prerequisite Knowledge		
PK1	Pointer Variable Declaration	The student understands how to declare a pointer variable in memory.
PK2	Pointer Variable Assignment	The student understands how to assign a value to the pointer variable.
PK3	Pointer Variable Initializing	The student understands a node pointer variable initialization and the importance of doing that.
PK4	Pointer Variable Operators	The student understands how to use the asterisk (*) and ampersand (&) operators
PK5	Dereferencing Pointer Variable	The student understands the meaning of dereferencing a pointer variable and how the pointer variable is dereferenced.
PK6	Pointer Variable Manipulation	The student understands how to manipulate the pointer variable.
PK7	Freeing Memory	The student understands the benefit of using free function and how to use it.

PK8	Memory Storage	The student understands the memory model and how the data is stored and arranged in the memory.
PK9	Malloc() Command	The student understands what malloc function does and what it returns.
PK10	Coding with Functions	The student understands the how to pass the linked list through function parameters and arguments.
Linked List Pieces		
C1	Node	The student is able to identify the two members of the node (list data and node pointer) and what is the type of data that can be stored.
C2	Node Pointer Variable	The student is able to identify the node pointer variable's data type, the type of data that pointer variable can store, and the different forms that it can come in.
C3	NULL Pointer	The student is able to identify the meaning of NULL pointer, where it can be used in the context of the linked list, and the importance of including it.
C4	List Data	The student is able to identify the type of data that stores in the node's data section.
C5	Head Pointer Variable	The student knows the importance of the head pointer variable and the benefit of including it.
C6	Tail Pointer Variable	The student knows the importance of the tail pointer variable and the benefit of including it.

C7	Node Pointer vs. Node for Head & Tail	The student is able to identify the head and tail as node pointer variable and knowing the importance of being node pointer variable vs. node.
Linked List Types		
C8	Singly Linked List	The student is able to identify all different pieces of a singly linked list type.
C9	Doubly Linked List	The student is able to identify all different pieces of a doubly linked list types.
C10	Singly Circular Linked List	The student is able to identify all different pieces of a singly circular linked list types.
C11	Doubly Circular Linked List	The student is able to identify all different pieces of a doubly circular linked list types.
Linked List Operations		
C12	Node Allocation	The student understands how to allocate a new node in memory and assign its memory to a pointer variable.
C13	Accessing Node Members	The student understands how to access the list data and node pointer members that are stored in a node.
C14	Create an Empty List	The student understands the steps for creating an empty list.
C15	Check Empty	The student understands the steps to check for an empty list.
C16	Insert a Node at the Beginning of the List	The student understands the steps for adding a new node at the beginning of different types of linked lists.
C17	Insert a Node at the End of the List	The student understands the steps for adding a new node at the end of different types of linked lists.

C18	Insert a Node at the Specific Location of the List	The student understands the steps for adding a new node at the specific location of different types of linked lists.
C19	Delete a Node at the Beginning of the List	The student understands the steps for deleting a node at the beginning of different types of linked lists.
C20	Delete a Node at the End of the List	The student understands the steps for deleting a node at the end of different types of linked lists.
C21	Delete a Node at the Specific Location of the List	The student understands the steps for deleting a node at the specific location of different types of linked lists.
C22	Iteration	The student is understands the steps to iterate through a linked list.
C23	Find List Length	The student understands the steps to find the list length.
C24	Swap: Nodes vs. Data	The student is able to differentiate between swapping only the data or the node as a whole and know the benefit of using each of them.
C25	Swap Nodes	The student understands the steps for swapping nodes.
C26	Find Value	The student understands the steps to find the node that stored the intended value.
C27	Print List	The student understands the steps to print the values stored in a linked list.
C28	Clear List	The student understands the steps to clear the entire linked list.

Table 3.1. A comprehensive list of prerequisite knowledge and linked list concepts with their definitions.

3.2 Linked List Framework

We measure students' mental models of a singly linked list by examining students' conceptual understanding of the singly linked list pieces and their procedural understanding of how to implement these pieces and operations on lists. We also evaluate how students apply or use this knowledge to understand other types of linked lists, such as a doubly linked list and a circular linked list.

Figure 3.2 shows a complete view of the framework used to examine students' mental models of a singly linked list in C. We use the C language in this research because it is the language used in the data structures class at the participating university. The figure shows that knowledge about singly linked lists (middle 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 a singly linked list in C, and the prerequisite knowledge and singly linked list knowledge are required for understanding other types of linked lists (right rectangle).

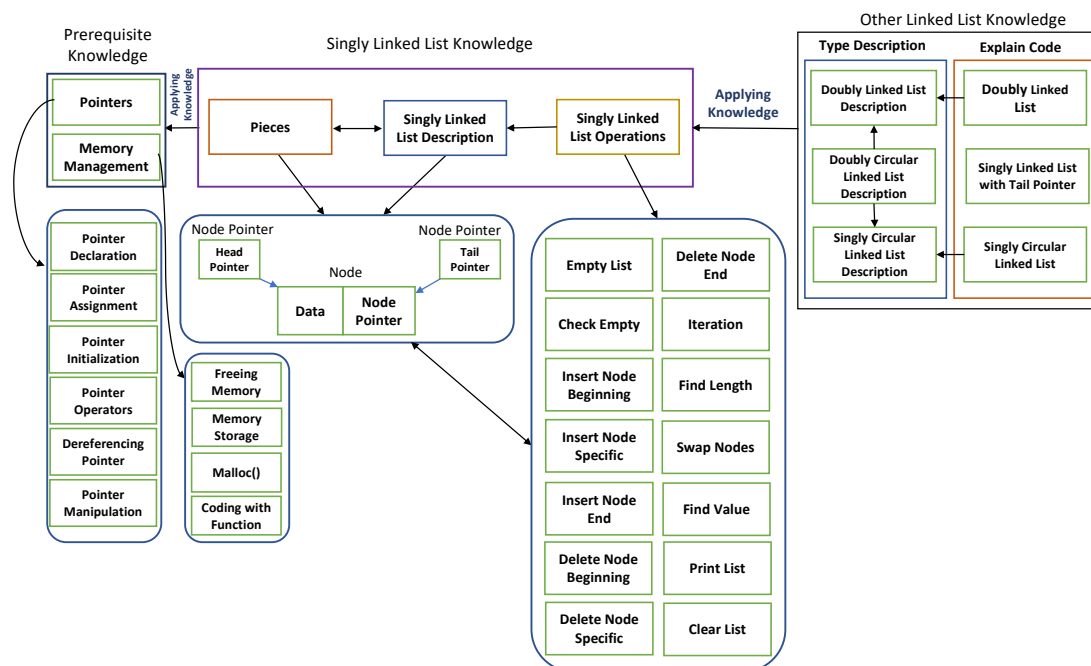


Figure 3.2. Linked List Framework illustrating the relationship among prerequisite knowledge, a singly linked list, and applying this knowledge to different linked lists.

First, students must generally understand what a linked list is, which requires knowledge of the pieces (parts) of the linked list but not necessarily the operations. Next, students must understand the five pieces 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 pieces, we measure their procedural understanding of 14 operations on singly linked lists (see Figure 3.2). We expand 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 these four operations. However, since students think aloud while implementing the linked lists, we are able to infer their conceptual understanding.

There is a strong dependency between understanding the pieces of a linked list and the ability to describe a singly linked list. If students have a deep understanding of the linked list pieces, they can correctly combine these pieces to describe a singly linked list.

Therefore, after identifying students' reasoning and understanding of individual linked list pieces, we explore how students combine their reasoning about these pieces when broadly describing a singly linked list and implementing operations on it.

Even though we do not directly measure all aspects of students' prerequisite knowledge about pointers and memory management, we do ask students some questions about pointer assignment and freeing memory. However, we primarily evaluate their prerequisite knowledge from their interview responses and code for a singly linked list. Then, we measure how students apply their knowledge of a singly linked list to describing different types of linked list and explaining how their code changes for other types of linked lists.

3.3 The Survey and Semi-Structured Interview Questions:

For students to have a correct mental model of linked lists in C, they need to understand all the linked list concepts, including the prerequisite knowledge. We use the linked list categories, inventory of linked list concepts, and framework of linked list concepts as a guide for building the survey and constructing the interview questions. The interview and survey questions come from a variety of sources including Kruse's "Data Structures and Program Design in C" textbook [67], lecture notes and book created by one of Oregon State University's faculty members, online resources "GeeksforGeeks"¹, classroom discussions from a summer 2019 lecture, and the researchers based on linked list concepts in Table 3.1. The observations in the classroom led to some questions being directly based on summer 2019 students' thinking at the participating university, which is helpful for detecting the same misconceptions other students may have [1].

The **survey** mainly focuses on examining students' conceptual understanding of a subset of the linked list concepts identified in Table 3.1. We use the results from the survey as a guide for our preliminary understanding of students' mental models about linked lists. The survey contains 28 questions with 12 multiple-choice questions, 5 open-ended questions, and 11 multiple choice with a justification for the choice (see Appendix A). The first part of the survey collects demographic and background information from the participants; whereas, the rest of the survey is about linked lists and pointers.

Only the fundamental concepts are covered in the linked lists survey to keep the time limit to 15 minutes. We ask students to recognize linked list pieces in drawings,

¹<https://www.geeksforgeeks.org/>

identify the benefit of specific linked list pieces, recognize different types of linked list by a drawing, and answer basic questions about pointers. While it is likely that the students are reasoning about their choices on most of the survey questions, especially for those where they must provide a justification for their answer, this survey primarily aims to assess their conceptual understanding for a subset of concepts and not their reasoning or procedural understanding.

Whereas, the **semi-structured interview** questions focus on measuring students' conceptual and/or procedural understandings of the identified concepts in Table 3.1 and the relationship between these concepts (see Appendices B - C). The interview questions are open-ended and consist of three main types: verbal, coding, and recognition. In the 13 verbal type questions, we ask students for definitions and the importance of different linked lists types and pieces. In the 9 coding questions, we ask students to write code or pseudocode for operations on a singly linked list.

Even though students in their data structures class are taught about singly and doubly linked lists with head and tail pointer variables, they only implement a singly linked list with a head pointer variable by themselves. Therefore, we only ask students to write code or pseudocode for a singly linked list with a head pointer variable. However, we do ask the participants in the last 3 coding questions of the interview to explain any modifications to their code if a tail was added to the end or if they changed the type of linked list to a doubly or circular linked list. Based on Hoffman, we believe that students who have accurate mental models of a singly linked list can apply this knowledge to unseen problems, which is important to extract from students' mental models [56]. The students only need to mention the name of the functions that need to change and how they are going to change them for the different types of lists.

In the last four questions, we ask students to recognize four functions about linked list operations and the purpose of each function. Only two interview questions directly ask students about their prerequisite knowledge, but the answers to the other questions about linked lists depend on a good understanding of the prerequisite knowledge.

3.3.1 Collect Expert Feedback and Revision:

We collect feedback by three experts who previously taught linked lists (2 graduate students and one professor) to pilot and revise the linked list categories, concept list,

survey questions, and interview questions. The experts agree on the importance of all concepts listed in Table 3.1, and they did not have any additional questions for the survey and the interview. However, the experts did suggest fixing misspelled words and adding clarity to questions that were confusing or needed to be split into two questions. We revised the survey and the interview questions many times, and the survey and interview questions in Appendices A - D reflect the final version.

3.3.2 Mapping Survey and Interview Questions to the Linked List Concepts

Since each survey and interview question assesses a specific linked list concept identified in Table 3.1, we map the concepts to the corresponding survey and interview questions to make sure we have coverage, as well as to see the dependencies concepts have with other concepts (see Table 3.2).

The questions have a letter identifying which area of the study we asked the question. The questions prefaced with a 'S' are from the survey, and the questions prefaced with an 'I' are from the interview. Then, we separate the interview questions into those that ask students for a verbal response (IV), to write code or pseudocode (IC), or to recognize what a piece of code does (IR). Within the coding questions, we label questions with an 'IE' to indicate when students are asked to explain their code or how it might change given a new concept. The prerequisite knowledge at the end of the table is not directly related to the linked list knowledge, but it is essential knowledge that the students need in order to successfully understand linked lists. Even though we only have a few questions to directly assess prerequisite knowledge, we analyze students' prerequisite knowledge as part of the rubric for all interview questions.

As seen in Table 3.2 mapping, some concepts are assessed using multiple questions from the survey or interview, which allows us to get multiple perspectives of how students understand individual concepts needed for an accurate mental model of linked lists, and we have complete coverage of the concepts in the interview. We recognize there are dependencies between the 28 linked list concepts identified in Table 3.1, such as when answering questions about operations on linked lists which require the understanding of linked list pieces. We show the mapping of the dependencies in the survey and interview in the "Concepts" column on the far right of Table 3.2.

	Survey	Interview			
Pieces	Questions	Verbal Questions	Coding Questions	Recognition Questions	Concepts
Node	S-Q1, Q7	IV-Q1, Q12	IC-Q1, Q3, Q6.1, Q6.2		PK1-5,PK9,C1, C2,C3,C4,C6,C12, C13
Node Pointer	S-Q2,Q5, Q8,Q10	IV-Q1, Q2, Q4, Q5, Q6, Q7	IC-Q3, Q6.1,Q6.2		PK1-6,PK10, C1,C2,C3,C6,C7
Data	S-Q15,Q16, Q17	IV-Q9	IC-Q6.1, Q6.2		PK1-5, C3,C12,C13
Head	S-Q6	IV-Q3, Q3.1,Q3.2	IC-Q1,Q3		C1,C2,C5,C7,C8
Tail	S-Q9	IV-Q3,Q3.1,Q3.2	IE-Q7		C1,C2,C6,C7,C8
Types					
SLL	S-Q4 S-Q18	IV-Q4			C1,C2,C3,C4, C5,C8
SCLL	S-Q12	IV-Q6	IE-Q8		C1,C2,C3,C4, C5,C10
DLL	S-Q13, Q19	IV-Q5	IE-Q9		C1,C2,C3,C4, C5,C9
DCLL	S-Q11	IV-Q7			C1,C2,C3,C4, C5,C11
Operations					
Create Empty List	S-Q3		IC-Q1		PK1-5,PK8, PK10,C1,C3,C5, C8,C13,C14,C17
Check Empty			IC-Q2		PK1-5,PK8, PK-10,C2,C5, C13,C15
Insert Beginning			IC-Q3		PK1-5,PK7-9, C1,C5,C8,C12, C13,C16
Insert Specific Location			IC-Q6.2		PK1-6,PK8-10, C1,C4,C5,C8, C12, C13,C18,C22,C26

Insert End		IC-Q6.1	PK1-6,PK8-10, C1,C3,C4,C5, C8,C12,C13,C17 C22	
Delete Beginning	S-Q20	IC-Q4	PK1-7,PK10, C5,C8,C13, C15,C16,C19	
Delete Specific Location		IC-Q6.4	PK1-7,PK10, C5,C8,C13, C21,C22,C26	
Delete End		IC-Q6.3	PK1-7,PK10, C3,C5,C8, C13,C20,C22	
Iteration		IC-Q5, 6.1, 6.2, 6.3, 6.4	PK1- 5,PK8,PK10, C5,C8,C13,C22	
Find Length		IC-Q5	PK1- 5,PK8,PK10, C5,C8,C13, C22,C23	
Swap Nodes		IV-Q13	IR-Q1	PK1-5,PK8, C5,C8,C13, C22,C24,C25
Find Value			IR-Q2	PK1-5,PK8, C5,C8,C13, C22,C26
Print list			IR-Q3	PK1-5,PK8, C5,C8,C13, C22,C27
Clear list	S-Q21		IR-Q4	PK1-8, C5,C8,C13, C22,C28
Prerequisite Knowledge				
Freeing Memory	S-Q22			PK7

Pointer Variable Declaration	S-Q23		PK1
Pointer Variable Assignment	S-Q24, Q25, Q26, Q27		PK1-2
Pointer Variable Manipulation	S-Q28		PK1-6,
Dereferencing Pointer Variable		IV-Q11, Q11.1	PK5

Table 3.2. Mapping linked list concepts to the corresponding survey and interview questions in which they appear. Note that S-Qx refers to survey question, IV-Qx refers to interview verbal question, IC-Qx refers to interview coding question, IE-Qx refers to interview explanation question, IR refers to interview recognition, SLL refers to singly linked list, SCLL refers to singly circular linked list, DLL refers to doubly linked list, and DCLL refers to doubly circular linked list.

3.4 Data Collection:

With permission from the Institutional Review Board (IRB) to recruit students, we visited the data structures class at Oregon State University (OSU) 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 out of the 40 students responded back to schedule the 2-hour, \$15/hr semi-structured think-aloud interview.

While the number of participants is small and the results are not generalizable to linked lists in other programming languages, 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 the C programming language that other researchers can leverage for future studies and educators can use to improve their instruction or assessment of linked lists in C. Since there is not much research on how students think about linked lists to use as a foundation, this research is similar to an initial, empirical study on 12 students' understanding of free fall [13], which was later

used by other researchers to develop the Force Concept Inventory, which is one of the most well-known misconception assessments in physics [77, 44].

Materials and Procedure: The online survey was constructed in Qualtrics, and the semi-guided interview, which was held in a Kelley Engineering Center conference room at OSU, followed a think-aloud technique that asks an individual to verbally express their thoughts as they answer a question or solve a problem [26]. We used the technique to capture the dynamic process of thinking, and throughout the semi-structured interview, we reminded the students to think-aloud.

All the interviews were audio and screen recorded using a Windows 7 touch screen tablet to capture what the student said, coded, and drew. Applications, such as Notepad++, MobaXterm, PuTTY, Microsoft Word, and the Google Chrome browser, were installed and saved as shortcuts on the desktop and taskbar for the participant's use. We gave all participants a printout of the interview questions, and there was no time limit for answering each question in the interview to reduce the students' stress level.

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 tablet with the stylus. While we were careful not to lead or bias student responses during the interview, when students did not say much, 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 to narrow their knowledge gap or clarify their misunderstandings. We also explained questions in more detail when they needed a question clarified or seemed confused by a question.

At the end of the interview, we gave the students a 20 item spatial visualization test called the Purdue Visualization of Rotations Test (ROT) [8]. A spatial visualization test measures the ability to mentally rotate, fold or unfold flat objects and manipulate two- and three-dimensional stimulus objects (e.g. performance of serial operations) in short-term memory [40][78]. The test had a 10-minute time limit to reduce the amount of analytic processing of the shapes, as suggested by the authors. We compare students' visual-spatial reasoning with how they reason about linked lists to provide more insight into their mental models about linked lists.

Demographics: Figure 3.3 shows the demographic and background information for the 40 survey participants, and Table 3.3 shows the demographics for the 11 interview

participants. The majority of the students self-identify 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 at the participating university to transition the students 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. In this research study, we give the interview students pseudonyms for their names that reflect their demographics, such as *Ecer* is the ECE student and *Chemi* is the chemical engineering student.



Figure 3.3. Demographics and backgrounds of 40 consenting participants in the survey.

PID	GPA	Gender	Race	Major	Take CS162 at OSU	Other School	LL Knowledge Prior CS261
Joe	3.54	Male	White	CS	Yes		Yes
Bob	3.2	Male	White	CS	No	PSU	Yes
Suzy	2.88	Female	Asian	CS	Yes		No
Bill	3.29	Male	White	CS	Yes		No
Max	2.97	Male	White	CS	Yes		Yes
Phil	3.5	Male	White	CS	Yes		No
Feng	3.16	Male	Asian	CS	Yes		Yes
Xeng	3.36	Male	Asian	CS	Yes		Yes
Chemi	3.98	Male	Asian	Chem. Eng.	Yes		Yes
Ecer	2.99	Male	White	ECE	Yes		No
Nate	3	Male	White	CS	Yes		Yes

Table 3.3. Demographics of 11 Consenting Interview Participants. Note: PID (Participants’ ID), LL (Linked List), CS (Computer Science), Chem. Eng. (Chemical Engineering), ECE (Electrical & Computer Engineering), and PSU (Portland State University).

3.5 Evaluation and Data Analysis:

Since we capture undergraduate students’ misunderstandings and knowledge gaps of linked list concepts using surveys and interviews, we use a mixed-method approach (quantitative and qualitative). We use the **survey** to gain a preliminary understanding of the students’ conceptual knowledge of basic linked list concepts, such as the pieces and types of linked lists, but to gain a deeper understanding of students’ mental models and factors that impact their understanding, we use the **semi-structured interview** to assess every linked list concept identified and mapped in Table 3.2, except for students’ prior knowledge of pointers and memory explicitly.

For the five open-ended questions in the survey (questions 6, 9, 10, 17, and 22), we generated a correct (or perfect) solution (see the blue answers in the rubrics in Appendix

A), and then, we assigned points to the details within the perfect solution for each question. This provided us an initial rubric to use for inter-rater reliability (IRR).

For each open-ended question, the main author of this study and a senior, undergraduate researcher independently coded 20% of the 40 participants (8 participants). To compute the IRR for each question, we used the average agreement between raters for the 8 participants.

For each question having an IRR below 80% agreement, the researchers 1) discussed their differences, 2) made changes to the rubric, and 3) graded a different set of 8 participants. The two researchers reached a 100% IRR for questions 9, 17, and 22, and they reached a 93.75% IRR for questions 6 and 10.

Given this reliability, each researcher used the agreed upon final rubrics in Appendix A to independently grade the remaining participants, as well as re-grade any participants where IRR was not reached earlier. Figure 3.4 shows the rubric used to grade question 6, which asks about the benefit of including the head pointer variable (labeled as A in the question 4 diagram) at the beginning of the list, and we provide an example coding for the first two participants.

Q6. From Q4, what is the benefit from including A (head) at the beginning of the list?

Answer:
To indicate the start/ beginning of the linked list

Rubric:
The question rewards one point:
1 pt (explicitly states indicate the start/ beginning/front of the list)
0.5 pt (knows it points to a linked list, but not explicitly state start/ beginning of the list)
0 pt (doesn't mention indicate start/beginning/point/front)

ID	Answer	Grade	Reason for deducting points
P1	the head pointer can point a link list	0.5	The student does not explicitly state start/ beginning of the list.
P2	We can access the beginning easily	1	
...			
P8			

Figure 3.4. Example rubric for an open-ended survey question with coding for participants Joe and Bob.

As with the open-ended survey questions, first, two researchers (one of them is an expert with many years of experience teaching linked lists in C) wrote the correct (or perfect) solutions for each question in the semi-structured interview (see the blue answers in the rubrics in Appendices B - D), and then we broke down the expert solutions into small pieces to create an initial rubric. Since most of the linked list concepts involve understanding other concepts, e.g. a node component involves understanding the list data, node pointer, and NULL pointer concepts (see Table 3.2), we make sure to separate these concepts in the rubric with their own individual points to allow us to analyze dependent concepts within concepts and other pieces of information.

After transcribing the audio data from the 11 participants for each interview question, the same two researchers for the survey questions independently coded 20% of the data using the initial rubrics, 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 same method mentioned above, i.e. the 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 an 80% or above IRR for each question, each researcher used the agreed upon final rubric in Appendices B - D to independently grade the remaining participants, as well as re-grade any participants where IRR was not reached earlier.

The final rubrics for the verbal and recognition questions are 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 Figure 3.5). We include the reasons for not being correct or partially correct, as well as any other observations we find surprising in the students' response. Figure 3.5 shows the final rubric used to score the first verbal question asking the student to describe a node in a linked list, and the figure includes an example coding for the first participant, Joe.

Q1. Describe a node in a linked list.
 A node in a linked list is an object that stores data of any type and pointer (memory address) of the next node or Null to indicate the end.

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:				
"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."				

Figure 3.5. Example rubric for a verbal interview question with coding for participant Joe.

For the coding questions, the final rubrics have two sections: the core section, which covers the understanding of the linked list concepts, and an additional/prerequisite knowledge section covering knowledge of other related concepts to the linked list (after the bold line in Figure 3.6). To have a complete procedural understanding of the singly linked list concepts, 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 and showing conceptual understanding, even if they lack procedural understanding. Because there are multiple ways to implement solutions, we had alternative wording in the rubrics to accommodate for multiple solutions.

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 Figure 3.6). Even if students write a function, we accommodate for the different ways a student might write this function. However, a student must be able to implement all pieces in the core section to successfully implement the operation. Figure 3.6 shows that the rubric for one solution to the first coding question and provides an example coding for participant Xeng. We use the results from coding questions and the survey questions as a triangulation for the missing or incorrect data in the verbal responses to interview questions. This provides more meaning and clarity to the data that is missing in these questions.

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:					

Figure 3.6. Grading participant Xeng on the coding interview question about creating an empty list using the generated coding rubric.

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. Points are not deducted 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.

The rubrics for the explanation questions in the coding section of the interview are updated to include additional details needed to make the change. Figure 3.7 is an example of the rubric used to grade Chemi's explanation about code modifications when adding a tail pointer variable. We consider the concept below the bold line as additional information because we provided the picture of the linked list for the participants. We did not expect that the participants would think about the case where inserting a node at a specific location would be after the last node. The rubric is very similar to the verbal and recognition question rubrics, but it has extra credit for stating what function needs to be changed (see Appendices A - D for more details about the rubrics).

Total Points: 3/15 0/3	Point Divvy: Gain points for mention of each of these topics:	States what needs to be change	Partially Correct to states how to change the what	Correctly states how to change the what	Comment: Reason for not getting full marks or anything else interesting about their conceptual and procedural understanding
3	Create an empty linked list: Set an additional tail pointer to point to NULL				Not present
3	Insert at the beginning: If the list is empty (both head and tail point to NULL), set head and tail to point to the new node.				Participant states does not change
3	Delete at the beginning: If there is only one node, set head and tail to NULL				Participant states does not change
3	Insert at the end: Access the end directly and update the tail to point to the new node			3	Participant states " <i>I could have just overwritten that tail with a tail pointer and then I would have to change the I would have to change the previous tail's node pointer.</i> " Also describes this operation in more detail later on
3	Delete at the end: Iterate the list to update the tail to point to the second last node and then free the last node				Not present
3	Insert after maybe: If the node after is the last node, then same as insert at the end, update the tail to point to the new node.				Doesn't think about this case
Other Interesting Observations/Comments Student Responses/Answers: States "Find length does not change"					

Figure 3.7. Grading Chemi's explanation about code changing when adding a tail pointer variable using the explanation code rubric.

Chapter 4: Results and Discussions

After coding all participant responses to the survey and interview questions using the final rubrics in Appendices A - D, we use the coded rubrics as the data for our analysis of student mental models and misunderstandings. In this chapter, we discuss our method for answering each research question and present the results from our analysis of the multiple-choice survey responses, coded rubrics of answers to open-ended questions, and visual-spatial reasoning test scores to answer our four main research questions.

- RQ1: What are students' mental models of linked lists in the C programming language, and how accurate are their mental models?
- RQ2: What difficulties do students face while learning about linked lists in the C programming language?
- RQ3: What is the relationship between students' understanding about linked lists and their visual-spatial reasoning?
- RQ4: What is the relationship between drawing pictures while reasoning about linked lists and students' visual-spatial reasoning?

In Section 4.1, we uncover students' mental models about linked lists and the accuracy of their mental models to answer RQ1. In Section 4.2, we address RQ2 by presenting the difficulties students' report having while learning about linked lists in C. In Section 4.3, we correlate students' mental models and drawing pictures of abstract concepts with their score on the Purdue Visualization of Rotations Test (ROT) [8] to gain a better understanding of how students' visual-spatial reasoning might impact their understanding of linked lists to answer RQ3 and RQ4.

4.1 RQ1: What are students' mental models of linked lists in the C programming language, and how accurate are their mental models?

We uncover students' mental models about linked lists and how accurate they are by addressing each sub-question individually. First, we measure the accuracy of students' mental models using the quantitative scores given on the coded rubrics, and then, we address the details of students' mental models by analyzing students' qualitative responses to the survey and think-aloud, semi-structured interview questions (see RQ1.1 and RQ1.2 below).

- *RQ1.1: How accurate are students' mental models about the types and pieces of linked lists and operations on linked lists?* We measure accuracy by calculating each student's overall score for each survey and interview question based on the correctness of their multiple choice answer or the total points received on a rubric for grading an open-ended question. We use the question scores and mapping to concepts in Table 3.2 to evaluate how accurate students' conceptual and procedural understandings are, which addresses how accurate students' overall mental models of linked lists are.
- *RQ1.2: What are students' misunderstandings or gaps in knowledge about the types and pieces of linked lists and operations on linked lists?* We classify students' misunderstandings as incorrect responses, and we define gaps in knowledge as a lack or absence of knowledge in the students' responses. To determine students' understanding of the types and pieces of linked lists, as well as the operations on linked lists, we analyze students' answers to multiple-choice survey questions and the coded rubrics with respect to each concept identified and mapped in Table 3.2. We present a content analysis for each linked list concept with descriptions/themes of students' misunderstandings and lack of knowledge, as well as an analysis of their prerequisite knowledge.

4.1.1 RQ1.1: How accurate are students' mental models about the types and pieces of linked lists and operations on linked lists?

We define accuracy as the correctness of the students' answers to survey and interview questions compared to an expert. We identify students' mental models by examining students' conceptual and procedural understanding of linked lists in the C programming language. Since students are not asked to implement other linked lists beyond a singly linked list with a head pointer variable, we do not measure the accuracy of students' procedural understanding of different types of linked lists or a tail pointer variable. However, we do measure students' conceptual understanding of all types and pieces of linked lists, in addition to their procedural understanding of singly linked lists.

Survey Performance: First, we calculate the percentage of students with the correct answer for each multiple-choice survey question (see Figure 4.1). The results from the 11 students who participated in the interview are compared with the other 29 participants who only took the survey. There are not many differences in the two groups of students, which indicates that the interview participants are largely representative of all the students who participated in the survey.

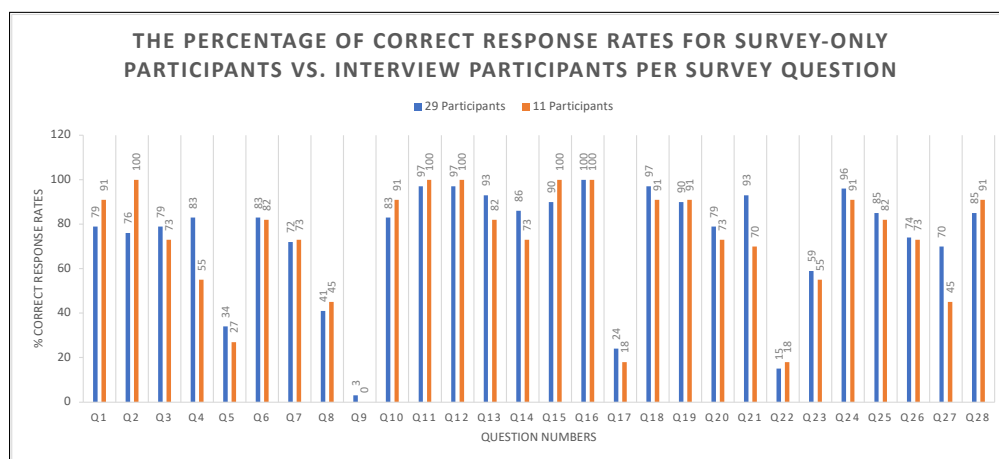


Figure 4.1. Percentage of students with the correct answer on each survey question. Note: Q21 is out of 10 participants, and Q22-28 are out of 27 participants.

In Q2 about the node pointer, the interview participants outperform those who only took the survey; whereas, the students who only took the survey outperform the interview

Q12	100%	63%	13%	50%	88%	38%	38%	63%	50%	38%	38%
Q13	100%	33%	33%	-	100%	100%	33%	67%	33%	67%	100%
Total	59%	48%	34%	53%	52%	40%	45%	36%	44%	41%	46%
Coding	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chem	Ecer	Nate
Q1	19%	100%	44%	94%	100%	63%	81%	100%	81%	25%	75%
Q2	43%	100%	0%	100%	100%	100%	71%	100%	71%	71%	43%
Q3	77%	100%	15%	100%	100%	92%	85%	85%	54%	69%	31%
Q4	81%	78%	6%	78%	100%	81%	94%	81%	75%	63%	0%
Q5	82%	100%	41%	82%	95%	91%	86%	100%	95%	91%	95%
Q6.1	82%	100%	25%	93%	100%	93%	86%	100%	96%	79%	93%
Q6.2	71%	100%	21%	82%	68%	93%	82%	100%	96%	82%	86%
Q6.3	74%	100%	11%	100%	58%	89%	58%	100%	95%	68%	89%
Q6.4	100%	100%	23%	100%	77%	100%	64%	100%	95%	86%	86%
Total	74%	98%	23%	91%	87%	89%	79%	97%	88%	73%	74%
Explain											
	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate
Q7	20%	40%	33%	53%	20%	20%	40%	53%	20%	33%	20%
Q8	33%	28%	0%	50%	0%	39%	11%	56%	56%	39%	33%
Q9	67%	19%	5%	57%	14%	71%	14%	57%	29%	57%	38%
Total	43%	28%	11%	54%	11%	46%	20%	56%	35%	44%	31%
Recognition											
	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate
Q1	50%	50%	0%	50%	50%	50%	50%	50%	50%	50%	50%
Q2	100%	100%	50%	100%	100%	100%	100%	100%	100%	100%	100%
Q3	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Q4	100%	100%	0%	100%	100%	100%	100%	100%	100%	100%	100%
Total	80%	80%	30%	80%	80%	80%	80%	80%	80%	80%	80%

Table 4.1. Each participants' percentage of points earned on each question from the verbal, coding, and recognition sections of the interview. Note: Light-red (< 80%), blue (80-89%), gray (90-99%), and green (100%).

Students' score much higher on coding and recognition questions about operations on linked lists in the interview (see Table 4.1). Overall, students do not do well on questions asking them to verbalize/explain their understanding of concepts. This might be because students can arrive at the correct solution through trial and error with code or lack enough detail in their responses to show a complete understanding of the concept.

For a more detailed view, we break down students' accuracy on interview questions by concept (see Table 4.2). The light-purple rows represent conceptual understanding, and the light-orange rows represent procedural understanding. In the interview, we only measure students' conceptual understanding of different types of linked lists and about dereferencing a pointer variable, but we measure both their conceptual and procedural understanding of the pieces of and operations on a singly linked list.

Only about 36% of the participants (Bob, Bill, Max, and Xeng) accurately code the node structure piece of a singly linked list in C, and only 27% of the participants (Bob, Bill, and Xeng) know how to correctly code the next node pointer variable as part of a node. Even fewer, only 18% of the participants (Bob and Max), accurately code the head pointer variable, and almost half of the participants (45%) do not have a complete understanding of the data piece of a linked list (Suzy, Bill, Phil, Ecer, and Nate).

The pieces of a linked list are fundamental concepts and required for understanding other linked list concepts. While none of the students have an accurate conceptual or procedural understanding of all the pieces of singly linked lists, there are many students who have an accurate procedural understanding of some pieces (see the top four orange rows in Table 4.2). However, most students do not have a good conceptual understanding of different types of linked lists (see the top four light-purple rows in Table 4.2), and overall, most students do not well on most conceptual questions in the interview.

Type Description	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate
SLL ^a	72	67	72	72	72	28	61	44	50	56	78
SCLL ^b	67	67	61	72	61	44	67	28	39	44	67
DLL ^c	68	64	50	64	68	27	55	18	50	32	55
DCLL ^d	82	45	45	73	59	36	64	32	45	45	64
SLL Pieces	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 Pointer	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 ^e	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
Prerequisite Knowledge	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate
Dereferencing Pointer	67	67	0	67	67	67	67	67	67	67	67
Pointer Operator	100	100	50	100	100	100	100	100	100	100	50

^aSLL refers to a singly linked list

^bSCLL refers to a singly circular linked list.

^cDLL refers to a doubly linked list

^dDCLL refers to a doubly circular linked list.

^eScore is calculated differently due to missing responses.

Table 4.2. Scores per linked list concept based on conceptual (light-purple) and procedural (orange) understandings.

Overall Accuracy: Lastly, we compare the 11 interview students' overall score for each linked list category identified in Figure 3.1 and evaluated in the survey and interview (see Table 4.3). One student did not answer a question in the survey, and another student did not answer a question in the interview due to time. Instead of counting these as incorrect answers, we excluded the points from the total points when calculating the average (see the overall score in red in Table 4.3).

PID	Survey					Interview				
	Overall Types	Overall Pieces	Overall Operations	Overall LL ^a	PK ^b	Overall Types	Overall Pieces	Overall Operations	Overall LL	PK
Joe	83	78	67	78	61	73	59	76	69	75
Bob	100	74	100	84	82	60	72	95	80	75
Suzy	50	44	33	44	39	56	21	23	29	13
Bill	100	70	100	82	68	70	67	91 ^c	77	75
Max	83	59	100	71%	75	65	61	87	74	75
Phil	100	78	100	87	100	34	63	90	70	75
Feng	100	74	67	80	46	61	65	77	69	75
Xeng	83	67	67	71	46	30	69	95	74	75
Chemi	83	85	100	87	100	46	63	87	71	75
Ecer	83	81	0	71	96	44	56	74	62	75
Nate	83	78	50 ^c	73	61	65	48	76	64	63

^aLL refers to linked list.

^bPK refers to prerequisite knowledge.

^cScore is calculated differently due to missing responses.

Table 4.3. Overall score for each linked list category.

None of the participating students have a completely accurate mental model of linked lists evaluated in the survey or the interview (see Table 4.3). Phil and Chemi have the highest score (87%) in the survey, but they do not have the highest scores in the interview. However, Bob has the next highest score (84%) in the survey and the highest score (80%) in the interview. Suzy, who could not write in code and preferred pseudocode, has the lowest score on both the survey (44%) and the interview (29%) questions. 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.

Overall, students perform better on the survey than in the interview, but the survey is not as comprehensive as the interview. For example, the four out of the five students

receiving a 'B' grade on the survey do not make above an 80% in the interview. We also notice that students are better at identifying types of linked lists in the survey (four receive a 100%) than describing them in the interview. This is because describing an answer to a question requires the student to express more detail, and none of the students accurately describe different types of linked lists with as much detail as an expert does (see Appendix F for more details).

Also, students perform better on questions about the operations on linked lists than they do on the pieces of linked lists in both the survey and interview. Table 4.3 shows that five students have accurate mental models of the operations covered in the survey, and these same students are some of the higher performers on the operation questions in the interview. In the survey, we only ask the students three multiple-choice questions about checking if a linked list is empty, deleting a node at the beginning of a list, and clearing a list. Whereas, in the interview, we ask about all the operations on a linked list identified in Table 3.1. 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 pieces of a linked list or prerequisite knowledge.

4.1.2 RQ1.2: What are students' misunderstandings or gaps in knowledge about the types and pieces of linked lists and operations on linked lists?

In order for us to address students' conceptual and procedural understanding of linked lists in the C programming language, we must first analyze students' prerequisite knowledge about pointers and memory management that are required for working with linked lists in C. To answer RQ1.2, we begin with an evaluation of students' prerequisite knowledge in Subsection 4.1.2.1 followed by their understanding of the pieces of and operations on singly linked lists in Subsections 4.1.2.2 and 4.1.2.3. Lastly, in Subsection 4.1.2.4, we discuss students conceptual understanding of different types of linked lists, and we organize student misunderstandings and lack of knowledge as themes within the four categories identified in Section 3.1 (see Figure 3.1).

4.1.2.1 Prerequisite Knowledge

Given that linked lists have pointer variables and require many pointer variable manipulations in C, we ask students 7 background questions in the survey (Q22-28) about memory management and pointer variable declaration and assignment to assess students' general prior knowledge about pointers and memory. Later, in the interview, we ask them to explain what dereferencing a pointer means and how a pointer is dereferenced.

4.1.2.1.1 Freeing Memory

In the survey, we ask an open-ended question (Q22) about the benefit of using the `free()` function in the C language. We look for students to say something about deallocating (or freeing) the memory space allocated by the function `malloc()`, as well as that memory can be reused by a subsequent `malloc` function calls. We find that only two interview participants (Phil and Chemi) correctly answer this question, while four participants (Bob, Suzy, Bill, and Ecer) almost answer the question correctly by mentioning that the `free` function is responsible for deallocating memory space, but they say the benefit is to prevent memory leaks, rather than allowing the memory to be reused. Other participants (Max, Feng, Xeng, and Nate) only mention what the `free` function does, which is deallocating the memory assigned by `malloc`. Joe, on the other hand, says the benefit is to prevent memory leaks. The `free` function can prevent memory leaks, but this is due to being able to reuse the freed memory space. Joe continues to show an understanding of the benefit of the `free` function during the interview, which is not the case for others who correctly state what the function does. Many students either forget to use the `free` function or answer questions about deleting a node at the beginning of the list incorrectly because they do not understand when to free the node.

4.1.2.1.2 Declaring a Pointer Variable

After asking students about the `free` function, we ask students to identify the meaning of `char *p` (the l-value in the first assignment statement in Figure 4.2). The correct answer is to "Create a pointer to a character" (second option). Six students out of the 11 correctly answer the question, while the other students incorrectly select the "Create an array of characters" option (Bill, Max, Feng, Xeng, and Nate). While `p` can be used to

point to the address of the first element in an array of characters, the declaration itself is not used to create an array of characters. The array is created using the `malloc()` command.

<p>Q23. Suppose you have the following:</p> <pre>char *p = (char) malloc(sizeof(char) *10); char *q = (char) malloc(sizeof(char) *10);</pre> <p>What does the part <code>char *p</code> mean?</p> <p><input type="radio"/> Create a character variable</p> <p><input type="radio"/> Create a pointer to a character</p> <p><input type="radio"/> Create an array of characters</p> <p><input type="radio"/> Other</p> <input type="text"/>
--

Figure 4.2. Survey Question 23: Identify the meaning of a pointer variable declaration.

4.1.2.1.3 Pointer Variable Assignment

The next multiple-choice question (Q24) asks students the meaning of the `p = q` assignment (see Figure 4.3). The correct answer is “Make p point to the same thing as q” (second option), and we find that students do very well on this question. Only one student (Joe) chooses “Make q point to the same place as p”. However, in the interview, Joe seems to have a good understanding of pointer variable assignment, which means this answer could have been a mistake.

<p>Q24. What does $p = q$ mean? Please write why you choose this answer?</p>
<p><input type="radio"/> Make q point to the same place as p <input type="text"/></p> <p><input type="radio"/> Make p point to the same place as q <input type="text"/></p> <p><input type="radio"/> All of the above <input type="text"/></p> <p><input type="radio"/> Other <input type="text"/></p>

Figure 4.3. Question 24: Identify the legality of pointer variable assignments in the survey.

In the following three questions (Q25-27), we ask students to identify whether a variety of assignment statements between two character pointer variables (p and q) are legal or illegal (see Figure 4.4 as an example). First, we ask students if $*p = *q$ is legal, and we find that 9 out of 11 students correctly answer that the statement is legal, while Suzy and Feng choose that it is illegal. Feng claims that “ $*p$ is a dereference”, but Suzy does not provide any reasoning for her answer.

<p>Q25. Is the following operation legal or not? Please write why you choose this answer?</p> <p>$*p = *q;$</p>
<p><input type="radio"/> Legal <input type="text"/></p> <p><input type="radio"/> Illegal <input type="text"/></p>

Figure 4.4. Question 25: Identify the legality assignments of the pointer variable in the survey.

The second two questions (Q26 and Q27) ask students if $p = *q$ and $*p = q$ are legal. We find that only 3 of the 11 participants (Suzy, Feng, and Xeng) incorrectly believe that it is legal to assign a character value to a place that stores addresses, while 6 of the 11 participants (Joe, Bob, Suzy, Bill, Xeng, and Nate) incorrectly believe that it

is legal to assign an address of a memory location to a place that stores character values. Regardless of what the compiler allows, one should not do either of these assignments.

When assigning a character value to a place that stores addresses ($p = *q$), Feng believes that this is legal because “they are both pointer(s)”, but Suzy and Xeng do not mention any reason for their choice. On the other hand, for assigning a memory address to a character ($*p = q$), some students believe that it is legal to overwrite the memory address with a value and state that the compiler is not going to throw any errors. Bill states, “Now, p is pointing to a memory address”, and Bob claims, “You will set the data value of p to a pointer, but the compiler will work with this as it will set p ’s new data value to the q ’s address.” These students do not seem to truly understand the importance of matching types and what the dereference operator ‘ $*$ ’ means.

The last question in the survey (Q28) is a synthesis of understanding pointer variable declaration and assignment using the asterisk and ampersand operators (see Figure 4.5). Out of the 11 students interviewed, Suzy is the only participant who incorrectly answers this question in the survey and who struggles the most with pointer manipulation in the interview. While most students understand how to follow code with the asterisk (dereference) and ampersand (address of) operators in the context of single number values, most students do not explain the purpose of dereferencing a pointer variable in the interview.

<p>Q28. What is the output after the execution of the following code:</p> <pre> double *p; double pi, e; p = &e; *p = 2.71828 p = &pi; *p = 3.14159; printf("%p %g %g %g\n", p, *p, pi, e); </pre>
<p><input type="radio"/> eebff768 2.71828 3.14159 3.14159</p> <p><input type="radio"/> eebff768 3.14159 3.14159 2.71828</p> <p><input type="radio"/> 3.14159 3.14159 2.71828 eebff768</p> <p><input type="radio"/> 3.14159 3.14159 eebff768 2.71828</p>

Figure 4.5. Question 28: Identify the code’s output about pointer variable manipulations and dereferencing using asterisk and ampersand operators in the survey.

4.1.2.1.4 Dereferencing a Pointer Variable

To understand students' conceptual knowledge about dereferencing pointer variables, we ask the students two closely related questions:

- “What does dereferencing a pointer mean?”
- “How is a pointer dereferenced or which operation can be used to dereference a pointer?”

We find that most of the participants know how to explain what dereferencing a pointer variable is, except Suzy. Suzy begins by saying, “*it has to do with the star and ampersand. So dereferencing, I cannot remember.*” Then, she continues to say that it is the ampersand for the background stored in the heap. It is clear that this student does not know what dereferencing a pointer variable is.

All other students describe it as fetching the information (or a value) at the memory location stored in the pointer variable. However, it is interesting that they do not state that dereferencing is needed to store information, and as Feng writes code to show what dereferencing a pointer variable is, the student writes `int* = a; a = 1;`. This shows a misunderstanding in the pointer variable declaration and assignment to the pointer variable, similar to what Caceffoe et al. found [11].

Contradictory to our expectation that all participants would state that the asterisk is used to dereference a pointer variable in the follow-up question, we find that two students mix up the asterisk and ampersand operators for dereferencing a pointer variable, and most students write code to explain how to dereference a pointer variable. After declaring a pointer to an integer, `int *a;`, Nate claims that “`a[0];` would go to the memory address of `a`”. This will work, but it is not needed if `'a'` does not point to an array. After that, Nate writes `a&;` to claim another way to dereference a pointer variable, which is not correct. This is similar to Suzy's confusion about the ampersand when asked to describe what dereferencing means.

4.1.2.2 Pieces of a Linked List

It is clear that students' prerequisite knowledge about pointers and memory in C is lacking, which influences their understanding of linked lists in C. This section presents

students' reasoning and misunderstandings about the basic pieces of a singly linked list in C. We begin by discussing students' understandings of the concept of a node structure containing a data member and next node pointer variable followed by details about the head and tail node pointer variables outside the node structure.

4.1.2.2.1 The Node

Besides the prerequisite knowledge needed for understanding linked lists in C, the first concept of a singly linked list for students to understand is the node structure. The node structure contains two members: the list data of any type and a pointer to a node. Students must understand how to define a node structure, how to allocate a node in memory, and the importance of the NULL pointer for a complete understanding.

Conceptual Understanding

To examine students' conceptual understanding of a node, we purposefully ask the students, in the interview, a general question, "*Describe a node in a linked list*", to see what they say on their own. We are looking for students to specifically talk about 1) what the node contains, 2) the purpose of each part of the node, and 3) the need for setting the last node's node pointer variable to NULL. While most students are not as detailed as an expert in their description, when we triangulate their responses with their responses to survey and other interview questions, we find that the majority of the students are not lacking an understanding of what a node is. However, there are themes of misunderstandings that emerge about the node and node pointer, types of list data stored in the node, and the NULL pointer value.

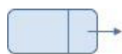
Confusion around the node containing a node pointer and being a node pointer: Even though Suzy is the only student who does not talk about the node pointer as a member of the node structure when describing a node, we find that her misunderstanding shows up in the survey and again when she defines a node. Suzy states that there is a data piece and a hidden key in the node, and she says that there is an additional piece outside the node, which is a pointer to connect the nodes. When we follow up with this student about the hidden key, the student states, "*like the information is stored in a heap. So, it's ... I forget the word. It's like the ID of the, ID of the node where it's stored.*" We believe Suzy is using the word 'key' to mean pointer because the

student drew an arrow in her picture on the tablet, but she draws the arrow outside the node. We think Suzy is mixing up linked list concepts with other data structures, such as a hash/look-up table (or dictionary) that has a key. We also think she is struggling with the concept of a node containing a node pointer and not being a node pointer.

We see this same confusion between a node and node pointer in the survey distributed before the interview, where we ask students to identify a node in two multiple-choice questions (Q1 & Q7). Q1 asks students to identify a node independent of a list (see Figure 4.6), and Q7 asks students to identify a node (item B) within a picture of a singly linked list (see Figure 4.7).

In the following questions, \equiv represents NULL

Q1. What does the following drawing represent? Please write why you choose this answer?



-
- Node Pointer
- Node
- Data
- NULL Pointer
- Other

Figure 4.6. Survey Question 1: Identify a node from a picture.

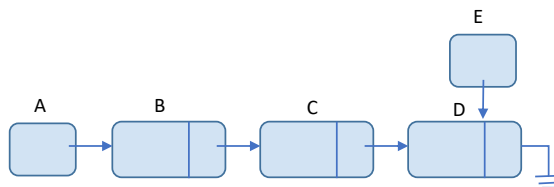


Figure 4.7. Survey picture of a singly linked list with a head and a tail pointer variables used in Q4-Q7.

All students correctly answer Q1, except for Suzy, who chooses “Node Pointer” without

stating a reason. Suzy also selects “Node Pointer” as the answer for Q7 without a reason, but Max also selects “Node Pointer” as the first node in the singly linked list. Max states, “*It contains data (even if NULL), and has a pointer to the next node.*” It seems that this student understands that a node contains the list data and a pointer to the next node because he answers survey Q1 correctly and states that a node has a node pointer in the interview. However, it is interesting that Max identifies a node by itself as a node but refers to the first node in the linked list as a node pointer. Even though the confusion between the node and node pointer is not directly seen in more students, there is further evidence that other students confuse these concepts in other questions.

Unsure about any type of list data in the node: In addition to students knowing that one member in a node is the list data, it is important for students to know that the list data in a node can be of any type, such as an integer, double, character, pointer, array, structure, etc. When asked to describe a node, Suzy, Bill, Feng, Xeng, Chemi, and Ecer do not talk about the types of information associated with the list data in a node, much less that the type can be anything. Even when asked “*What kind of data can be stored in a linked list?*”, we find that Suzy and Bill continue to be unsure about the possible types of list data in a node.

At first, Suzy says that the data can be everything but then concludes that it can only be numbers and letters. This is similar to Bill who says, “*I feel like a lot of kinds of data can be stored. You can make the value field of the node in a linked list kind of anything.*”, but then he goes on to say he is unsure of this. Whereas, Nate begins unsure and then concludes that it can be anything, i.e. “*what kind of data! I’m not fully sure about that one. I would assume that most if not every type of data could be stored in the linked list.*”. Even though everyone recognizes that the list data linked together can be an integer, character, or pointer in the survey, they express uncertainty about this in the interview.

Lack of knowledge about typecasting malloc: Not only does creating a linked list in C require understanding how to define a node structure, it requires understanding how to allocate a node on the heap. To first measure students’ conceptual understanding of how to allocate a node in memory in C, we ask students what the following line of code means/does.

```
struct node *new_node = (struct node*) malloc(sizeof(struct node));
```

All students are able to give a generic response to this question and state that this line of code allocates a new node in memory, and Joe draws on the tablet after explaining the syntax. However, very few students talk about other aspects of the code, such as the typecasting or `sizeof()` function. When creating a new node on the heap 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. Unexpectedly, we find that nine participants do not mention what the typecast does in the code, and even more striking, Max, Phil, and Ecer mention that they had never seen the `(struct node*)` typecast syntax before. This explains why none of the participants initially typecast when writing code to allocate a new node. Two students (Bill and Phil) do typecast their `malloc` after receiving an error message using a C++ compiler, instead of the C compiler.

Procedural Understanding

Not only do students need to understand the concept of a node, they should know how to define a node structure and allocate a new node in memory using the C syntax from their data structures class.

Failure to define a node structure: In C, understanding how to define the node structure requires understanding the concept of a node, which contains the list data of any type and a pointer to a node. The first coding question asking students to “*Write code/pseudocode to create an empty linked list*” requires students to define a node structure as the first step.

Suzy, Phil, Feng, and Ecer do not use the correct syntax to define the node structure. This could be because the node structure definition is provided for the students in the code template for the linked list assignment in their data structures class. However, students should be able to recreate the structure on their own.

Interestingly, Suzy and Feng use pointer syntax with the node structure definition, rather than defining a `struct node` (see Figure 4.8). These students know they need to make a node, but they do not understand when they should use the pointer syntax and when they should not.

```

1  struct ls* (){
2      int val;
3      void key;
4      cur, pre, next;
5  }

```

```

1  struct node*{
2      int val;
3      *next;
4  }

```

Figure 4.8. Suzy (left) and Feng (right) node structure.

Not only do Suzy and Feng define a node pointer structure, instead of a node structure, but they do not define the node pointer member in the node correctly either. Suzy writes three variables names, i.e. `cur`, `pre`, and `next`, without stating their type, as well as includes a `key`, and Feng writes `*next`; for the node pointer member, instead of `struct node *next`;. Suzy's misunderstanding of the node containing a key carries over into her procedural understanding, when trying to define a node structure. In addition, Suzy does not define pointer variables, which suggests that her confusion between the node containing a node pointer is also carrying over. Whereas, Feng knows the node has a next pointer variable, but he does not include the `struct node` type. This could be because he defines the node structure incorrectly and gets confused.

Similarly, Phil and Ecer do not include the keyword `struct` when writing the syntax for their node pointer variable in the node, which is required in C but not in C++ (see Figure 4.9 for an example). We do not know whether they do not include this because they do not know it is needed or because they are mixing up the syntax with C++. In any case, students are reminded that they can compile and run their code to check for errors, but only Bill and Phil try compiling their code during the interview. Since Phil uses a C++ compiler, he does not see this as an error in C. In addition to leaving off the keyword `struct` before the the node pointer variable in the node, Ecer defines the list-data member in the node as `void`, but he states that the list-data member should be a void pointer (see Figure 4.9).

```

37  struct node {
38      void data;
39      node* next;
40  }

```

Figure 4.9. Ecer's node structure.

Misunderstanding of how to use the `sizeof()` function to allocate a node:

Students need to know how to allocate a node in order to create a linked list. To measure students' procedural understanding of allocating a node in memory, we look at how students dynamically allocate the node on the heap using `malloc` when asked to “*Write code/pseudocode for inserting a new node at the beginning of a singly linked list.*”. While typecasting the address returned by `malloc` is not required in C, understanding that you are allocating memory for the size of a node in memory is required for correctly allocating a node.

When we ask students to explain the code for allocating a new node in memory, most students recognize that `malloc` allocates the size of a new node in memory. However, Suzy might have a misunderstanding about the role of `malloc()` versus the `sizeof()` function because she states that the `malloc` command is used to “find the size of the node”, rather than stating it allocates memory for the size of a node. Other students either ignore talking about the `sizeof()` function or they simply read the syntax rather than explaining it. Suzy, Joe, and Ecer continue to show a misunderstanding about the use of the `sizeof()` function when allocating a new node, but their misunderstandings are about whether to pass a node or a node pointer type to `sizeof()`.

Joe and Ecer allocate memory for the size of a node pointer, rather than the size of a node, which further indicates students' misunderstanding of types and memory allocation.

```
malloc(sizeof(struct node*));
```

Although Joe explains the node's allocation perfectly, he could not write the syntax to allocate a new node successfully. In comparison, Ecer expresses uncertainty about putting an asterisk, ‘*’, after the size of `struct node` or not and states, “*I don't know if I can make it the size of that. I think that's the size of the node.*” These participants understand they need to allocate a new node (not a node pointer), but they do not know how to write the syntax.

4.1.2.2.2 The Node's List Data

The data stored in a node is an essential concept of linked lists. As stated in the conceptual part of Section 4.1.2.2.1, understanding that the list data in a node can be of any type is important, which some students are unsure about. However, having the procedural understanding of how to store and access list data in a node is also crucial.

Procedural Understanding

To examine students' procedural understanding of the node's data member, we analyze how students declare the data member in the node and store and access the list data. We observe two major procedural misunderstandings when students try to store information in the node's data member they declared as a void pointer variable and then try to access the information referenced by the data member.

Confusion about how to store data in a void pointer variable: In the node structure, some students define a void or void pointer variable as the list-data type in the node. We find that the students defining a void pointer variable have trouble using the variable when storing an integer value. For example, Bill, Ecer, and Nate try to assign an integer value to the void pointer variable, which is supposed to hold addresses not integers. Bill thinks that it would work better if he casts the void pointer variable to an integer 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. Actually, these students confuse themselves more by defining the list-data member as a void pointer, when they only need 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.

Misunderstanding about how to access the list data: In the coding question requiring students to allocate a new node and access the list-data member of the node, Phil thinks that accessing the list-data member in a node is different than accessing the node pointer member. The student uses the dot '.' operator whenever accessing the node's list-data member on the heap without dereferencing the node pointer to access the node's data, but he accesses the node's node pointer member correctly using the arrow ->.

It seems that this student associates the arrow and dot operators with the type of a node's member, rather than how they are being accessed. He is able to fix his error because he compiles his code and gets an error message that fixes the code for him. Even though Phil writes correct syntax for accessing the list-data member of a node moving

forward, we believe that he is still carries some form of misunderstanding about what is being dereferenced.

4.1.2.2.3 The Node's Node Pointer

The node's node pointer member refers to (or points to) a node by storing a node's memory address, and a node pointer can be a member of a node or independent of a node. Students seem to conceptually understand a node pointer as a member of a node (see Section 4.1.2.2.1), but understanding that a node pointer is independent of a node is critical for accessing the node that begins a nonempty list. We ask students to describe a node pointer in the interview, and we ask them to identify a node pointer in the survey. We find that students struggle with similar issues as they do when describing a node.

Conceptual Understanding

In the interview, we ask students to *“Describe a node pointer in a linked list”*. We explicitly look for students to talk about 1) what the node pointer type (`struct node*`) is, 2) how a node pointer is used to define either a pointer variable inside a node that points to the next node or declare a pointer variable independent of a node, and 3) the benefit of having NULL as a possible node pointer value.

Probably because students are not implementing a node pointer variable, we find that no one mentions the node pointer variable's type (`struct node*`). Knowing the node pointer's type and being able to distinguish it from a node's type, (`struct node`), is important to understand when implementing linked lists in C, and we see students procedurally struggle with these two types many times throughout the interview, which we discuss in Section 4.1.2.2.1 and this section.

While all participants state that a node pointer in a node refers to the next node or end of a list, only Joe, Bill, and Max state that a node pointer can be independent of the node, and Bill draws on the tablet to help him explain his answer. Similar to describing a node in Section 4.1.2.2.1, only Bill and Nate explicitly state that a node has a node pointer referring to the next node. Most students only mention that it refers to the next node without mentioning anything about addresses or how a node points to the next node. Even though the majority of students seem to understand that a node pointer variable can be independent of a node and refers to an address when creating the head

pointer variable, they do not explicitly talk about these concepts in the detail that an expert would, when asked to generally describe a node pointer. This is not the same for the concept of NULL. There are students who lack a conceptual understanding of the benefits of NULL that continue into their procedural understanding.

Lack of attention to the importance of NULL: Assigning NULL to a pointer variable means that the pointer is not valid, and in the context of linked lists, this means that a node pointer does not refer to a node. When a node's node pointer variable is NULL, it signifies that there are no more nodes after (or before, as in the case of doubly linked lists) the current node. When the head node pointer variable of a linked list is assigned NULL, it signifies an empty list. These two concepts are extremely important for successfully implementing linked lists in C.

Surprisingly, we find that the majority of the students do not give details about the need for NULL when asked about the node pointer. We understand students not talking about NULL when describing the node, but it is interesting that most students do not mention NULL when asked to describe the node pointer.

All students correctly identify a picture of a pointer pointing to NULL in the survey as a NULL pointer, which means that all students recognize what a NULL pointer is. All students, except Suzy, are able to say something about NULL being used to know you are at the end of the list or last node in other questions, but they do not talk about it when asked to describe a node pointer. For some students, we see this lack of attention to NULL carry over into their procedural understanding of the last node's node pointer member, as well as when they when they fail to assign the head pointer variable to NULL when creating an empty list (see the procedural part of Section 4.1.2.2.4).

Procedural Understanding

In addition to understanding what a node's node pointer member is, students must be able to declare, access, and manipulate the node pointer variable inside a node. We address a node pointer variable outside the node in Section 4.1.2.2.4 with students' understanding of the head and tail node pointer variables..

When we ask students to insert a node at the beginning of the list, students need to first make the node pointer variable in the new node point to the same as the head pointer variable, regardless of whether the list is empty or not. Of course, this is only true if the head node pointer variable was properly set to NULL for the empty list. We

find that Feng and Xeng still check for an empty list, and these students unnecessarily repeat almost the same line of code when inserting a new node in a empty and non-empty list, and Bill and Max begin drawing on the tablet before writing their syntax.

All participants writing code know to use an arrow to access the node pointer variable in a node. This suggests that they understand how to access the node pointer member in a node using a pointer to the node, but it is not clear they know why they are using the arrow or which node pointer they are dereferencing when using the arrow, as seen when accessing the list-data member in a node. However, there are students who do not mention or set the node pointer variable in the node to the head pointer variable's value before changing what the head pointer variable points to.

Misunderstanding of how to set a new node's node pointer variable to the head pointer variable: When inserting a node to the beginning of the list, we need to set its node pointer to the same as head pointer points, whether it is to the first node in an existing list or the NULL value when the list is empty. Suzy conceptually understands that the new node is connected to the previous first node, but she does not explain how the node refers to the previous first node (see top of Figure 4.10). Even though we ask the students to write in pseudocode, they need to be very specific about how the node pointer within the node connects to the existing first node. On the other hand, Nate does not even mention needing to set the new node's node pointer to the old first node. Instead, he sets the head pointer variable to the new node (see bottom of Figure 4.10), and he confuses himself when he assumes that the node to be inserted was created. He passes the node's pointer to the function without ever using malloc(), rather than passing a new node to the function.

```

32 int ls_insert(){
33     //create a new node for insert vlaue
34     //get val
35     //find size of whole lineked list and subtract 0
36     //connect the node to the beginning by pointing node to the preveious first element
37     //return new linked list
38 }

22 void insert(struct node * head, struct node * nodePutIn) {
23     head->next = nodePutIn;
24 }

```

Figure 4.10. Suzy (top) and Nate (bottom) responses to adding a new node at the beginning operation.

Another student, Feng, correctly assigns the head to the new node inserted at the

beginning of the list, but he assigns the new node's next to the second node in the list (see Figure 4.11 on lines 61 & 62), rather than the first node in the list using `temp->next=list->head`. This causes a memory leak by losing the first node on the list each time.

```

51 void insert(struct link_list* list, int val){
52     if(list->head == NULL){
53         struct node* temp = malloc(sizeof(struct node));
54         temp->val = val;
55         temp->next = NULL;
56         list->head = temp;
57     }
58     else{
59         struct node* temp = malloc(sizeof(struct node));
60         temp->val = val;
61         temp->next = list->head->next;
62         list->head = temp;
63     }

```

Figure 4.11. Feng's response to inserting a new node at the beginning of a list.

Continued lack of attention to the importance of NULL in operations at the end of a list: Some students who did not pay attention to the importance of NULL in their conceptual understanding continue to disregard the importance of NULL in operations at the end of the list. Although only Ecer fails to assign the new node's next node pointer variable 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.

4.1.2.2.4 The Head & Tail Pointer Variables

In C, a head pointer variable is an important concept that every nonempty linked list on the heap has. It is used to locate the first node that begins the linked list in memory. A tail pointer variable is another important concept. It points to the last node in the linked list, which allows for increased efficiency when doing certain operations. We do not evaluate students' procedural understanding of the tail pointer variable because they do not implement a linked list with a tail on their own in their data structures class. However, we evaluate them conceptually about the tail pointer variable.

Conceptual Understanding

In the interview, we ask the students to describe the difference between the head and the tail pointer variables in a linked list. In their responses, we are looking for them to talk about 1) what the head and tail pointer variables are, 2) how they point to the linked list, and 3) what they point to in the case of an empty list. All but Nate state that the head pointer variable points to the first node in the list, and Suzy and Nate are the only ones to incorrectly describe the tail pointer variable. Suzy says that the linked list points to the tail and the tail points to `NULL`, and Nate thinks the last node's next node pointer variable is the tail pointer variable that helps identify the last node in the list. As with the node pointer, no one explicitly mentions how the head and tail pointer variables point to the first and last nodes or point to `NULL` when the list is empty.

Then, we ask the students a follow-up question, “*What are the pros and cons of having a head and tail pointer?*”. We want the students to describe trade-offs and time complexity with regard to operations on linked lists. Most students ignore the performance achieved by head and tail pointers. They talk about the accessibility rather than time complexity, but they do say that it “helps” without saying how.

While all participants mention that the head pointer variable provides access to the beginning, participants did not mention that the tail is used to access the end of the list, and only Bill directly discusses the time complexity when accessing the end of the list with and without a tail pointer variable. It was unexpected to find that second-year students do not talk about time complexity or space when discussing the pros and cons to changing a data structure. This indicates students may not be able to differentiate between choosing the suitable data structure to use for better performance, as Zingaro et al. found with singly linked lists [120].

We find that Joe and Bob explicitly mention that the head pointer variable allows for new nodes to be inserted into the list, but only Bob mentions that the head allows for deleting a node at the beginning of a linked list. Joe and Bob also talk about the operations impacted by having a tail pointer variable, but Bob continues to believe that the tail pointer variable helps to insert and delete nodes at the end of the linked list, which is similar to other findings [90]. Since you cannot go backwards in a singly linked list, deleting a node from the end that has a tail pointer variable still requires iterating through the list to set the new last node to `NULL` and update the tail pointer variable to the new last node in the list. If the student is thinking about a doubly linked list, then they are correct; otherwise, they have a misunderstanding.

Students do not talk about the disadvantage of having a tail pointer variable. It would waste space if a node is never added to the end of the list. Xeng believes there are no disadvantages of having the head or tail pointer variable, and Chemi and Ecer believe that the disadvantage of having a head and a tail pointer variable is the complexity of managing two pointers in the code. However, the latter students have more difficulty with pointers in coding than some others, as shown in the node and node's node pointer sections (see 4.1.2.2.1 and 4.1.2.2.3).

Misunderstandings around the tail pointer variable: Contradictory to our expectation, Max mistakenly believes that we can have only a tail pointer variable without a head pointer variable in a doubly linked list or a circularly linked list. While students do not program a doubly or circularly linked list, they talk about doubly linked lists in class. So, it is interesting that students do not think that the head pointer variable is an essential part of every kind of linked lists, and it is even more interesting that Max and Phil think that the tail pointer variable has to exist in a doubly linked list. They may have learned that using a tail pointer variable with a doubly linked list improves performance, as Joe mentions, but they cannot elaborate on the use of the tail pointer variable.

Head or tail pointer variables are a node: Before asking the students to write code for creating a head pointer variable, we ask students “*Do you create the head or tail of a linked list as a node or node pointer?*” to find out if they think about the head pointer variable as being independent of a node or within a node. This is because we believe some students have a misunderstanding about the head pointer variable being within a node with an unused data member, rather than a pointer independent of a node. Nate continues to state that the head pointer variable is the first node in the list, which we see show up again in his procedural understanding when he declares and uses the head pointer variable.

Interestingly, Nate's response to the open-ended survey question asking about the benefit of the head pointer variable is that “A is the head, and it helps act as a pointer to the rest of the list”, and while in the interview, the student states that the head is the first node in the list. Nate also seems to be confused as to whether the tail pointer variable is either the last node or part of the last node. Whereas, Suzy believes that the tail is a node.

Procedural Understanding

Even though we assess students' conceptual understanding of the tail pointer variable, we do not assess their procedural understanding of this piece of a linked list because students do not implement a list in this way in their class. On the other hand, students need to know how to declare the head pointer variable and set it to NULL or the address of the first node that begins the list. Since you cannot have a linked list without a pointer to the first node in the list, then the head pointer variable should be created on the stack in the main function in C. We first discuss how most students waste memory by either putting the pointer to the first node of a list on the heap or creating the head pointer variable as a node. Then, we discuss how many students fail to assign the head pointer variable to NULL when creating an empty list or update the head pointer variable when inserting a node to the beginning of a list.

Wasting memory when creating the head pointer variable: Carried over from his conceptual misunderstanding that the head pointer variable is a node, Nate not only wastes memory by creating a node for a pointer to the beginning of the list, but he creates the node on the heap, which requires an additional pointer to the node, i.e. `struct node *head = malloc(sizeof(struct node))`. Then, he assigns the node's next pointer variable to NULL with `head->next = NULL`. The student does not think about the wasted memory for the empty data in the node and the additional pointer to the node, as well as the added complexity of accessing the head pointer. Similarly, Joe and Suzy create a head pointer variable, and then, they put a node on the heap when creating an empty list (see Figures 4.12 and 4.13).

```

7 void create_ls(){
8     struct ls* new_ls = (struct ls)malloc(sizeof(sturct ls));
9     //set up cur, pre next
10
11     //if creat_ls is null do not return else return ls
12     if(new_ls == NULL){
13         return NULL;
14     }
15     return new_ls;
16 }

```

Figure 4.12. Suzy's response to creating an empty list.

```

1 //initialize head pointer to linked list node on heap
2 struct linked_list_node* node = malloc(sizeof(struct linked_list_node*));
3 struct linked_list_node* link_list_head = node;

```

Figure 4.13. Joe’s response to creating an empty list.

All other students, except for Nate, Joe, and Suzy, define two user-defined structures. One is a list structure for storing the head pointer variable (and possibly a tail pointer variable), and the other is a node structure for storing the data and next node pointer members. This is an excellent coding style that allows developers to send one list structure to functions when there is a head and tail, and it avoids the head pointer variable from being a double pointer to a node, if you pass it and need to change it inside a function.

However, just as the students using only a node structure, all students using a list structure as a container for the head pointer variable waste memory by storing their list structure on the heap using `malloc(sizeof(struct list))`, instead of the stack with `struct list l`, but at least they do not waste more memory by making a node with a data member that is never used. Since Bob, Max, Feng, Xeng, and Ecer create their list structure in a function, like Suzy with the node (see Figure 4.12), they have to store the list structure on the heap to be able to access it after the function is executed. Even though Ecer defines the node and creates his list structure when inserting a node, neither Ecer nor Feng returns the address of the list on the heap.

In any case, all students should create their list structure with the head pointer variable on the stack in the main function. Even students not using a function store the list structure with the head pointer variable on the heap (e.g., Phil), like Nate and Joe do with a node (see Figure 4.13). Beyond wasting memory, we also notice that students have many other errors with their code, such as returning a value in a void function, creating unnecessary pointers, and typecasting malloc, which are issues with prerequisite knowledge.

Misunderstanding of how to create an empty list: Joe, Suzy, and Phil fail to assign the head pointer variable to NULL when creating an empty list. Phil creates a head pointer variable but does not set the node pointer to NULL or mention anything about setting it to NULL. In contrast, Joe and Suzy declare a head pointer variable and set it to point to one node for an empty list (see Figures 4.12 and 4.13). This is likely because Joe and Suzy treat the next node pointer variable in the first node of the list as the head pointer variable, like Nate. However, they do not set the node’s next pointer variable to

NULL.

Like Phil, Chemi creates a list structure with a head pointer variable and does not assign it to anything. He believes that if the list has nothing inside, then it is empty. He says, “there would be nothing inside that would be reasonably empty.” However, after performing a check for an empty list, Chemi, unlike Phil, realizes the need to go back and store a NULL in the head pointer variable, and he fixes his code (see Figure 4.14).

```
50 int main (char** argv, int argc){
51     struct list l = malloc(sizeof(struct list));
52     list->head = NULL;
```

Figure 4.14. Chemi’s response in coding empty list operation.

On the other hand, Ecer believes that the list structure with a head pointer variable set to NULL is an empty list, but he incorrectly assigns the head pointer variable to NULL inside the list structure definition (see Figure 4.15). Not only does he forget the keyword “struct”, but it is illegal to assign a member a value in a struct definition in C.

```
1 struct list {
2     node* head = NULL;
3 };
```

Figure 4.15. Ecer’s response to creating an empty list.

Failure to update the head pointer variable: When inserting a new node to the beginning of the list, the head pointer variable’s value needs to be updated to point to the new first node. When using a function, the function should either return the new node’s address or receive the address of the head pointer variable (a double pointer to a node or a pointer to a list structure with a head pointer variable) to update what the head pointer variable on the stack in main points to. 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.

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. 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 4.16). These students struggle with making their function return types and argument/parameter types match, as discussed previously in Section 4.1.2.2.3. 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 4.16).

```

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 }
```

Figure 4.16. Chemi’s response to adding a new node at the beginning operation.

4.1.2.3 Operations on a Linked List

Assessing students’ conceptual understandings of operations on linked list is very important for examining their ability to recognize when to use the operations and know which operations to use. However, since time is a limitation in this study, we created the survey to be short to increase the completion rate, and we limited the interview to two hours. Therefore, in the survey, we only ask students about their conceptual understanding of three essential linked list operations: checking if a list is empty, deleting a node, and clearing the entire list. Whereas, in the interview, we ask them about all the operations on linked lists identified in Table 3.1.

In the interview, we only ask them to implement five operations with three variations of inserting and deleting nodes in a list. These operations include 1) create an empty list, 2) check if the list is empty, 3) insert at the beginning, after a specific location, and at the end of the list, 4) delete at the beginning, after a specific location, and at the end of the list, and 5) find the length. For assessing students’ understanding of the other operations in the interview, we ask the students to recognize or explain how to swap nodes, find a value in a list, print a list, and clear a list. When recognizing the code for an operation, we ask students to “*use one sentence to explain what the following function does*”, similar

to studies asking students to recognize recursive functions [6, 76].

Even though we do not specifically ask students to write functions when implementing operations, most of the participants (everyone but Joe and Bill) do. Joe and Bill are really good students, but they do not even include a main function in their code for the operations. Whereas, Chemi writes the first half of the operations as functions and the rest in the main function. Suzy tries to write the C syntax in the first two questions about creating an empty list and checking if the list is empty, but after that, she switches to writing pseudocode or comments. This is likely due to her weakness in coding and misunderstandings about linked list concepts, pointers, and memory management.

In the following subsections, we present students' reasoning and misunderstandings about specific singly linked list operations that students struggle with in the C language. We divide this subsection into five categories of operations. These categories include 1) working with empty lists, 2) inserting nodes, 3) deleting nodes, 4) swapping nodes, and 5) iterating through a list. In the empty lists category, we include the questions about checking if a list is empty and clearing the entire list. We do not cover creating an empty linked list in this section because we cover this with students' procedural understanding of the head pointer variable concept in Section 4.1.2.2.4. The inserting and deleting nodes categories include all three questions about inserting or deleting a node at the beginning, a specific location, and the end of a list. The swapping nodes category includes a question about why swapping nodes is better than swapping values and a question asking students to recognize the code for swapping adjacent nodes. Instead of manipulating lists, the last category of operations only requires iterating through the list, such as finding the length of a list, finding a value in a list, and printing a linked list.

4.1.2.3.1 Empty Lists

The simplest linked list is an empty linked list, which is a node pointer variable on the stack pointing to the NULL value (see Figure 4.17 for the visual representation we use for an empty list). As mentioned before, we discuss creating an empty list with students' procedural understanding of the head pointer variable (see Section 4.1.2.2.4). In this section, we measure students' procedural understanding of both checking whether the list is empty and clearing a list by deleting all nodes from an existing list.

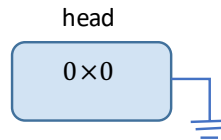


Figure 4.17. A visual representation of an empty list. Note that the electrical ground symbol at the end represents the NULL value.

Checking for an Empty List

Since many operations require checking for an empty list, we ask students to “*Write code/pseudocode to check if a linked list is empty.*” To check whether the list is empty, students only need to check if the head pointer variable is equal to the NULL value. We find that the four students who correctly created an empty list (Bob, Bill, Max, and Xeng), in addition to Phil, correctly check whether the list is empty. Ecer and Chemi understand how to check if a list is empty, but they forget the asterisk in the function parameter to make a pointer to the list. Similarly, Feng also knows to check if the head pointer variable is NULL, but his logic is backwards when he writes, “`if (list->head) print("empty") else print("not empty")`”. In this case, the empty message will print when the head pointer variable is not pointing to NULL. This leaves three other students who have misunderstandings about how to check if a list is empty, such as checking the data member of the node or using a loop to check all nodes in the list.

Misunderstandings about how to check if a list is empty: Suzy and Nate incorrectly compare the node’s data value member to NULL (see Figures 4.18 & 4.19), and Suzy believes that she needs a loop to check whether the list is empty. She says, “if checking if it’s empty, it should loop through everything.” While it is the programmer’s choice to return 1 or 0 for an empty list, a function checking if a list is empty or not usually returns true when it is empty and false when it is not. However, Nate returns true if it is not empty.

```

14 //return 1 if not empty, return 0 if empty
15 int empty(struct node * head) {
16     if (head->data != NULL) {
17         return 1;
18     }
19     return 0;
20 }

```

Figure 4.18. Nate’s response to checking if a list is empty operation.

```

18 void is_empty(){
19     while(ls->val != NULL){
20         //check element if empty
21
22         //travel
23         curr=curr->next;
24         return val;
25     ]
26     return NULL;
27 }

```

Figure 4.19. Suzy’s response to checking if a list is empty operation.

Interestingly, we find that Joe chooses to use an `assert()` function to check if a list is empty, rather than constructing a function himself. He writes the following line of code as his response to checking if a list is empty:

```
assert(link_list_head); //check that head points to a node
```

The `assert()` function can check if a pointer is `NULL`, but the program will error and stop running if it is. Since it is not an error to be an empty linked list, then it is a poor coding choice to use this method for checking if a list is empty.

Clearing a Linked List:

Clearing a linked list means deleting all nodes in a linked list and setting the head pointer variable to the `NULL` value. In the survey, we ask about deleting (clearing) the entire singly linked list (see Figures 4.20 and 4.21). The correct answer is the third option “while A is not pointing to `NULL`, assign a temporary node pointer to point to the same place as A, assign A to point to the next node in the list, and free the node pointed to by the temporary node pointer.”

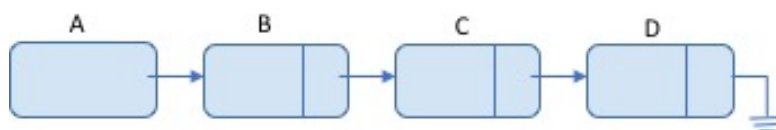


Figure 4.20. The diagram of a singly linked list presented in the survey and used in Q21.

<p>Q21. From Q20, suppose you want to delete the entire linked list, please select the best answer:</p> <p><input type="radio"/> Free A</p> <p><input type="radio"/> While A is not pointing to NULL, free what A points to and assign A to the next node.</p> <p><input type="radio"/> While A is not pointing to NULL, assign a temporary node pointer to point to the same place as A, assign A to point to the next node in the list, and free the node pointed to by the temporary node pointer.</p> <p><input type="radio"/> Both a and b are correct</p> <p><input type="radio"/> None of the above</p>
--

Figure 4.21. Question 21: Recognizing the steps for clearing a singly linked list.

We find that Suzy, Xeng, and Ecer wrongly answer this question. Xeng incorrectly selects the fourth option that states that the first and the second options are correct, and Suzy and Ecer mistakenly select the second option. The first and second options cause memory leaks by only deleting the first node in the list; therefore, nodes 'C' and 'D' are lost. We are unsure whether these students incorrectly believe that deleting what the head pointer variable points to would delete all the nodes in the list or not. Students might not understand the question wording because the majority can identify the code for clearing a linked list in the interview. It could also be the style of coding used in the class with a linked list structure containing the head pointer variable that confuses them.

In the interview, we ask students to identify the purpose of the provided code for clearing a list (see Figure 4.22). We find that ten participants give the correct answer, even Xeng and Ecer who wrongly answer the survey question about clearing the entire list. On the other hand, Suzy gets confused and thinks that there is an error in the code when seeing the syntax of setting the current pointer variable to the next pointer variable, after freeing the node that the current pointer variable points to. She believes

that the current pointer variable no longer exists, which indicates a misunderstanding of what freeing memory means and how the free function works.

4. Use one sentence to explain what the following function does.

```
void D(struct node *head)
{
    struct node* current = head;
    struct node* next;

    while(current != NULL)
    {
        next = current->next;
        free(current);
        current = next;
    }
    head = NULL;
}
```

Figure 4.22. Recognition question 4 for clearing the linked list nodes.

4.1.2.3.2 Inserting Nodes

To examine students' procedural understanding of inserting a new node in a singly linked list in the interview, we ask them coding questions about inserting a new node at the beginning of a singly linked list, at the end of the list, and after a specific location (in the middle of the list), and we report students' understanding of each of these operations in following paragraphs.

Inserting a New Node at the Beginning

After asking students to create an empty list and check if a list is empty, we ask students to “*Write code/pseudocode for inserting a new node at the beginning of a singly linked list*”. In performing this operation, the students need to 1) create a new node, 2) join the node to the beginning of the list, and 3) update the head pointer variable to point to the new beginning node.

We find that eight participants fail to successfully insert a new node at the beginning of the list. We find that most of the issues students have are problems with the basic

pieces of a list and not with the actual operation. For example, students 1) fail to create a new node (see Subsection 4.1.2.2.1), 2) misunderstand how to access the data member of a node (see Subsection 4.1.2.2.2), 3) fail to set the new node's node pointer variable to what the head pointer variable points to (see Subsection 4.1.2.2.3), 4) fail to update the head pointer variable value (see Subsection 4.1.2.2.4). The other issues students have are with misunderstanding how to access the first node of a list or making small syntax errors regarding pointers.

Misunderstanding of how to access the first node of a list: Suzy cannot describe the operation nor code; instead, she writes in pseudocode without a specific explanation. Suzy is the only one who believes that finding the first node in the list requires finding the list size and subtracting by zero. She claims, “subtract size by zero to get the beginning node linked list element node” The student also reasons that accessing a node in a linked list is the same as accessing a value in an array. She states, “I think subtract 0 because if we think of an array, it should be the position is 0.”

Suzy mixes up linked list and array data structures. Suzy even wonders which side the linked list starts by saying, “So, for this, where is the front would be in like (the) linked list? Would it be at the left side or the right side?” Eventually, she concludes that it starts from the left, but it is clear that the student is struggling with how to access the beginning of the list and ignores the importance of the head pointer variable. However, she knows that the head pointer variable points to the start of the list in the survey and verbal interview questions. However, she cannot apply this knowledge in coding questions because she relies more on recalling information from her memory without understanding it.

Syntactical Errors: Some students have minor syntactical errors that they do not notice, due to not tracing or compiling their code. For example, Joe understands that a new node must point to the old first node, but he incorrectly writes `node->ptr;`, instead of `node->next = ptr;`, to update the node pointer variable to point to the same node as the head pointer variable (see Figure 4.23 line 10).

```

6 //inserting new node
7 struct linked_list_node* ptr = link_list_head;
8 struct linked_list_node* node = malloc(sizeof(struct linked_list_node*));
9 node->val = value;
10 node->ptr;
11 link_list_head = node;

```

Figure 4.23. Joe’s response to adding a new node at the beginning operation.

Checking for an empty list is not required when inserting a new node at the beginning of a linked list with only a head pointer variable. Students need to make the new node point to what the head pointer variable points to, no matter whether the variable points to a node or the NULL value. Interestingly, Xeng, in the empty case, dereferences the head pointer variable, rather than the new node pointer variable, to store the data and the address to the next node (see Figure 4.24 in lines 28 & 29). In the case of non-empty list, the student correctly writes the code, but he uses `l_head` instead of `l->head` when attempting to update the head pointer variable (see in Figure 4.24 in line 34).

```

25 void list_insert(struct list* l,int val){
26     struct node* new_node = malloc(sizeof(struct node));
27     if(l->head == NULL){
28         l->head->val = val;
29         l->head->next = new_node;
30     }
31     else{
32         new_node->val = val;
33         new_node->next = l->head;
34         l_head = new_node;
35     }

```

Figure 4.24. Xeng’s response to adding a new node at the beginning operation.

Inserting a New Node at the End

To examine students’ understanding of adding a new node at the end, we provide a picture of a singly linked list to the participants (see Figure 4.25) and ask them to “*add item (6) to the end of the list*”. We want students to add a node that stores the value ‘6’ to the end of the list, which we clarify when students are confused. Since there is no tail pointer variable in the provided linked list, participants must iterate to the end of the list to add the new node to the end of the list. The student participants should 1) create a new node on the heap, 2) store the data and NULL values in the node, 3) iterate to the end of the list, and 4) connect the last node to the new node.

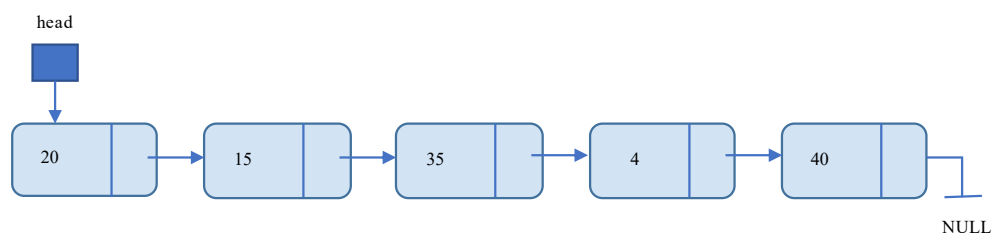


Figure 4.25. A singly linked list provided to the participants in coding question 6.

Similar to inserting a new node at the beginning of the list, most student participants procedurally understand how to insert a node at the end of a list. However, some students continue to struggle with using the pieces of a list, as discussed in Subsection 4.1.2.2.

Inefficient code for inserting a node at the end: Although Bill is the only participant to directly discuss the time complexity when accessing the end of the list with and without a tail, this participant and Suzy find the list's size to get to the end of the list rather than finding the last node pointing to NULL. This is not efficient because they go through the list twice to find the length and use it to get to the end. This could be due to asking them to find the list length directly before this question, but iterating twice through the list is not necessary for traversing the list.

Since Joe does not use functions, he does not directly use the length function to determine how many nodes to traverse to the end like Bill and Suzy. However, he continues to use the same node pointer variable from previously finding the list's length to directly connect the new node to the end of the list (see lines 28-30 in Figure 4.26). This would work if he had correctly assigned the pointer variable to the last node in the previous question and created a new node the size of a node, instead of a node pointer (see Subsection 4.1.2.2.1). Regardless, students should implement the operations independent of one another.

```

18 //find length
19 int count = 0;
20 struct linked_list_node* curr;
21 while(curr->next != NULL) {
22     count++;
23     curr = curr->next;
24 }
25 count++;
26
27 //introducing new item(6)
28 curr->next = malloc(sizeof(struct linked_list_node*));
29 curr->next->value = 6;
30 curr->next->next = NULL;

```

Figure 4.26. Joe’s response to adding a node at the end of a singly linked list.

Storing data before creating the node: According to Suzy’s psuedocode in lines 53 and 54 in figure 4.27, she seems to struggle with sequencing steps when adding a node to the end of the list. She adds the value to the node before creating it, but she does not do this when inserting an new node at the beginning of a list. The student understands the need to connect the new node to the end of the list, but she also gets confused between the pointer variable and struct member names (see lines 56 & 57 in Figure 4.27).

```

52 int ls_add(){
53     //initlize val to 6
54     //create a new node
55     //find total length of ls while cur->next is not NULL
56     //pre->cur = curr
57     //cur->next = NULL;
58 }

```

Figure 4.27. Suzy’s response to adding a node at the end of a singly linked list.

Misunderstanding how to manipulate the node pointer variable: In response this question, Feng says, “*I forgot how to get to the last last node*”. He fails to reach the end of the list because he does not use a loop. Instead, he uses an if-else statement and incorrectly manipulates the node pointer variables (see Subsection 4.1.2.2.3). In every coding question that requires traversing the list, Feng declares two temporary pointer variables separated by a comma operator, and only the second pointer variable is initialized (see line 20 in Figure 4.28). This student may have a misunderstanding of

the comma operator, and he may think that an assignment at the end is for both node pointer variables. Each pointer variable needs its own initialization. Feng's if statement may never be executed because the `curr` pointer variable may not ever be `NULL`. When the `else` statement is executed, the `curr` and `next` pointer variables are set to the second node, since he cannot remember how to get to the end of the list.

```

16 void insert_end(struct link_list* list, int val){
17     struct node* last = malloc(sizeof(struct node));
18     last->val = val;
19     last->next = NULL;
20     struct node* curr, * next= list->head;
21     if(curr==NULL)
22         curr->next = last;
23     else{
24         next=list->head->next;
25         curr = next;
26     }
27 }
```

Figure 4.28. Feng's response to adding a node at the end of a singly linked list.

Inserting a New Node at a Specific Location

To measure students' understanding of adding a new node to the middle of the list after a particular node, we ask the student participants to *"add item (50) after the node that stores the value 35"*. Students must 1) create a new node on the heap with the data value of '50', 2) iterate to the node that stores the value '35', 3) connect the new node to what the node with '35' is connected to, and 4) connect the node with '35' to the new node. Since we provide the picture of the linked list with a node that has the value '35', no one checked whether a node with '35' existed or not, but we did not deduct points for this in the rubric. We only noted it as additional information.

Mixing up the order for inserting in the middle: Three students, Joe, Max, and Suzy, attach the new node to the node with the value '35' before attaching the new node to what the node with the value '35' is linked to. This shows that students are confused by pointer manipulation, and this prevents these students from correctly inserting a new node to the middle of the linked list.

Joe successfully traverses the list to the node that stores the value '35', but he assigns the new node's address to a temporary pointer variable that was used to iterate to the

node storing '35' in the list. Therefore, he loses the node's location with the '35' value and does not join the new node to the list or mention how to connect the new node (see Figure 4.29). Joe could be confused because he does not use a different node pointer variable to point to the new node. Even though he is able to delete a node at a specific location, he seems to get confused by not having two different node pointer variables to keep track of where the new node and the node with the '35' value are.

```

32 //store item(50) after item(35)
33 curr = link_list_head;
34 while(curr->next->value != 35) {
35     curr = curr->next;
36 }
37 curr = curr->next;
38 curr = malloc(sizeof(struct linked_list_node*));
39 curr->value = 50;

```

Figure 4.29. Joe's response to adding a new node at a specific location in a singly linked list.

While Max did very well on adding a new node to the beginning and the end of the list, he seems to struggle with adding a node to a specific location. He uses an undeclared and uninitialized node pointer variable to the head pointer variable in the loop, and he gets confused by the sequence of steps needed to join the node to the list using the two node pointer variables (see lines 79 & 80 in Figure 4.30). Instead of setting the next variable in the new node to point to what the node at a specific location points to first, Max points the pointer to the specific node to the new node, which should have been the specific node's next variable, i.e. `curr->next = node;`. This is similar to Joe, who also set the pointer to the specific node, rather than the specific node's next variable. However, Joe was missing the first step altogether, instead of mixing the order. At least, Max understands how to make the new node's next point to what the specific node's next variable points to.

```

71 void list_insert(struct list* list, int data, int pos){
72     struct node* node = malloc(sizeof(struct node));
73     node->data = data;
74     int curr_pos = 0;
75     while(curr_pos != pos-1){
76         curr = curr->next;
77         curr_pos++;
78     }
79     curr = node;
80     node->next = curr->next;
81 }

```

Figure 4.30. Max’s response to adding a new node at a specific location in a singly linked list.

Suzy shows some understanding of iterating through the list to reach the node with the value ‘35’ and connecting the new node after the node that stores the value ‘35’ in the list (see lines 62, 64, & 65 in Figure 4.31). However, lines 61 and 63 in the same figure show a misunderstanding of creating the new node, and Suzy does not explicitly state the need to create the node or state that temp is a node pointer variable referring to the new node. In addition, she also mixes up the order of operations by connecting the specific node to the new node before using it to connect the new node to the list.

```

60 int ls_add_middle(){
61     //set val to 50
62     //loop though the linked list with a while (ls->val !=35)
63     //create a temp value to store 50
64     //connect element holding 35 to temp
65     //then connect temp to next value with pointer to next
66     //return linked list
67 }

```

Figure 4.31. Suzy’s response to adding a new node at a specific location in a singly linked list.

Creating an infinite loop: Not only does Ecer define a function without a return type, but he also creates a list structure parameter and uses it as a pointer to a list structure. Ecer’s bigger issue is that he creates an infinite loop inserting nodes with the value ‘50’, if a node with the value ‘35’ is found in the list (see Figure 4.32). Ecer should either move the pointer variable to the next node inside the if and else or move the pointer variable regardless of the if.

```

62 add_50(struct list linked){
63     struct node* n = linked->head;
64     while(n->next != NULL){
65         if(n->data == 35){
66             struct node* i = malloc(sizeof(struct node));
67             i->next = n->next;
68             n->next = i;
69             i->data = 50;
70         }
71         else n = n->next;
72     }
73 }

```

Figure 4.32. Ecer’s response to adding a new node at a specific location in a singly linked list.

4.1.2.3.3 Deleting Nodes

Similar to inserting a new node, we examine students’ procedural understanding of deleting a node in a singly linked list. We ask coding questions to delete a node at the beginning of the list, at the end of the list, and after a specific location (in the middle of the list). We report students’ struggles with each of these operations in the following paragraphs.

Deleting a Node at the Beginning

For deleting a node at the beginning of a singly linked list, the students need to 1) assign a temporary pointer variable to point to the first node to save the first node’s memory address, 2) update the head pointer variable to point to the second node, and then 3) delete the first node. It is also essential to check for an empty list when deleting a node in a linked list to prevent a segmentation fault.

Lack of checking whether the list is empty: We find that eight participants do not check for an empty list (Joe, Bob, Suzy, Bill, Phil, Xeng, Ecer, and Nate). Although Xeng checks for the empty list when inserting at the beginning, he does not check if the list is empty when deleting a node at the beginning, which is necessary in this case. These students do not fully understand the importance of handling the empty list case when deleting a node from the beginning of the list.

Non-generalizable solution for deleting nodes at the beginning: Beyond not checking whether the list is empty, Nate unnecessarily comments whether the last pointer

variable is `NULL`. He does not make a generalizable solution and tries to delete the node added from question 3, which inserted a node to the beginning of the list. If the list is empty or there is only one node, then the check to see if the second node's next pointer variable is `NULL` will cause the program to crash (see Figure 4.33 line 27). If there are more than two nodes, then this solution will cause a memory leak when freeing the second node before updating the head pointer variable (see Figure 4.33 line 29). Nate is missing knowledge about pointers and memory management needed to successfully add and delete nodes at the beginning of the list.

```

26 void delete(struct node * head) {
27     head->next->next == NULL; //check so we don't lose memory
28
29     free(head->next);
30 }

```

Figure 4.33. Nate's response to deleting a node at the beginning operation.

Deleting a Node at the End

Question 6.3 in the interview asks students to “*delete item (6)*”. To do this, students need to delete the last node at the end of the list. If students were confused by the question, we explained the question in more detail. We will frame it more clearly in the future. Students, in response to this question, need to 1) traverse to the second to last node, 2) delete the last node, and 3) set the new last node's next pointer variable to `NULL`.

Deleting or dereferencing `NULL` when deleting the last node :

When deleting the last node in the list, we find that Joe provides a hard-coded answer specific to the provided linked-list picture, rather than creating a generalized solution (see Figure 4.34 line 42). Joe uses the node pointer variable's value from the previous question and writes the code for iterating over three nodes and deleting the fourth. By doing this, Joe deletes `NULL`, instead of deleting the last node.

```

41 //free item(6)
42 curr = curr->next->next->next;
43 free(curr);

```

Figure 4.34. Joe's response to deleting the last node in a singly linked list.

Unlike Joe, Phil creates a generalized function for deleting a node at the end and specific location of a list. Phil was successfully able to delete the node at the specific location, but his code causes a segmentation fault after deleting the node at the end of the list. After the last node is deleted, Phil updates the current pointer variable to point to NULL (see Figure 4.35 line 97). In line 87, the compiler will stop and cause a segmentation fault because the student checks ‘‘current -> next != NULL’’, and there is no node’s next pointer anymore (see Figure 4.35 line 87). Phil needs to check after deleting the last node whether there is a node or not to stop advancing the current node pointer variable.

```

83 void deletenode(struct list* mylist, int value) {
84
85     struct node* current = mylist->head;
86
87     while (current->next != NULL) {
88
89         if (current->next->data == value) {
90
91             //delete node
92             struct node* temp = current->next;
93             current->next = current->next->next;
94
95             free(temp);
96         }
97         current = current->next;
98     }
99     return;
100 }
```

Figure 4.35. Phil’s response to deleting the last node in a singly linked list.

Deleting a Node at Specific Location

We also ask the student participants to “*delete item (50)*”, which is a node they inserted in the middle of the list. They must 1) use two temporary pointer variables to traverse the list: one is to point to the node before the one being deleted and the other is to delete the specific node that has the value ‘50’, 2) connect the nodes before and after the one to delete together, and 3) delete the specific (middle) node.

Trouble manipulating two pointer variables: Suzy seems to have a conceptual understanding of deleting a specific node in the middle of the list. She understands the need to traverse the list searching for the value ‘50’, and she has some understanding of

how to connect the node before and after the node with the '50' together (see Figure 4.36). Even though Suzy can state what dereferencing a pointer variable is, she does not understand the difference between the node pointer variables inside and outside of a node being referenced. Suzy connects the node members with the arrow operator in a way that matches her linked list drawing. For example, when she draws a linked list to solve one of the questions, she draws three nodes and labels them `pre`, `cur`, `next`. Suzy understands the need to connect the previous node with the next node, and she understands the need to delete the current node. However, she does not use the 'next' node pointer variable member in the node to connect the nodes, indicating a weak understating about how to use pointers to structs to access struct members.

```

76 int ls_delete_mid(){
77     //while (ls->val != 50)
78     //once out of loop
79     pre->cur->next = pre->next
80     free(cur)
81 }
```

Figure 4.36. Suzy's response to deleting a node in the middle of a singly linked list.

Ecer solves this question similar to how he solves the previous questions asking him to insert a node with the value '50' and delete a node with the value '6'. Instead of using a while loop, he uses an if-else statement. He uses two temporary pointer variables: one pointer variable to iterate through the list finding the node with the value '50' and another one to point to the node just before the node with the '50' value. However, just as with adding a node at a specific location, the node pointer variable that traverses the list is not updated. However, unlike before, Ecer is not even updating the pointer variable used to traverse the list in the case when the node with the value '50' is not found. This shows that this student is having a hard time manipulating two node pointer variables in one operation.

```

84 void remove_50(struct list linked){
85     struct node* n = linked->head;
86     struct node* p = n->next;
87
88     while(p != NULL){
89         if(p->data == 50){
90             p = p->next;
91             free(n->next);
92             n->next = p;
93         }
94         else n = n->next;

```

Figure 4.37. Ecer’s response to deleting a node at the middle of a singly linked list.

4.1.2.3.4 Swapping Nodes

Swapping nodes can be achieved efficiently by exchanging the memory addressees stored inside the node’s node pointers of the intended nodes. In the interview, we ask the students a verbal question to examine their conceptual understanding of why swapping nodes is better than just the data inside a node. Then, we asked a recognition question to measure their procedural understanding of swapping adjacent nodes.

Swapping Nodes vs. Swapping the Data

The swapping operation in a linked list can be done between adjacent nodes or non-adjacent nodes, and some people swap data in the nodes rather than the nodes themselves. Even though linked list nodes do not have to be physically adjacent in memory, they are connected through a node’s next node pointer variable. Swapping nodes is better than swapping list data values because swapping node data can be expensive when the data is large. Whereas swapping nodes is always constant by exchanging the node pointer values.

In this study, we want to measure students’ understanding of swapping nodes and why this is better than swapping node data values. In the interview, we ask students the following verbal questions, “*If you want to swap two numbers, what do you think is better, swapping the node or swapping the data?*” and subsequently “*Why do you think that?*”. Bill is the only student not able to answer these questions due to time limitations in the interview. However, we find that only four participants (Joe, Max, Phil, and Nate) give a correct answer to this question.

Incorrect reason for why swapping nodes is better: We find that six out of the ten participants (Joe, Suzy, Max, Phil, Ecer, and Nate) mention that swapping nodes is better than swapping data, but they give a wrong reason. For example, Ecer thinks that “swapping the nodes would be the least amount of steps.” Although Suzy knows that swapping nodes is better, she shows a misconception of the logic of swapping node pointers vs. values. She claims that swapping data requires searching for pointers, and swapping pointers requires searching for values. Suzy states, “if you remove the number try switch the numbers then you have to find the key (means pointer) of the linked list. And usually, when you’re just switching the element of linked list elements like the box things, you just look for the value instead.” Both swapping types need to look for the values and pointers to swap or sometimes need to search only for pointers when the list is reversed, or two adjacent nodes are swapped.

Misunderstanding that swapping data is better: However, Bob, Feng, Xeng, and Chemi believe incorrectly that swapping the data is better than swapping nodes. For example, Bob thinks that swapping data takes less time than swapping the pointers. The student reasons, “because you would need a lot more written out code to have to re-update all the pointers if you’re shifting entire node. Whereas if you’re just taking what the next what next value is equal to and setting it to the current and then what current value is setting it to next. That’s a lot quicker and less time cumbersome than what changing all the pointers values are.”

At first, Xeng also believes that swapping nodes increases time complexity “because it required moving the whole node”. However, Xeng, in the end, concludes “if it’s swapped the data, that’s also fine. And if it’s where the nodes that’s also fine, like both ways we can do things”. After this answer, we repeated the question to the student and clarified that we wanted the reason to be in terms of efficiency. Then, Xeng changed his answer to claim, “the efficient way, maybe the swapping the nodes because it could completely change the nodes, both nodes and it will be more efficient. And if we just swap the data, at some point, like at some edge cases.....because there are some edge cases in the linked list where that swapping data can be like bad for that.”

Feng and Chemi believe that swapping the data is easier than swapping pointers. Chemi explains that swapping data requires iterating the list two times to grab and swap the data between the two nodes. The student states, “you can just traverse the list once, find whichever node you want to the first node that you want to swap the data out. Pull

that out, continue traversing, find the next node, swap out the the data value, go back to the beginning of the list, and then swap in the data value that you store from the second node, which I think would be better than trying to find the pointers. It would just you would have to do like maybe half as much swapping.”

The majority of the students do not think that the data stored in a node could be significantly large. We may need to be very specific when asking this question to make sure students think beyond integer and singular values as data in the nodes. The students do not think broadly when answering this question.

Swapping Two Adjacent Nodes

We provide a picture of a singly linked list (see Figure 4.38), along with the C code for constructing this list, for the students to use when answering the recognition questions (see Appendix D). The first function in the recognition section is for swapping adjacent nodes of the singly linked list (see Figure 4.39). After executing this function, the new order of the list will be 2, 1, 4, 3, 5.

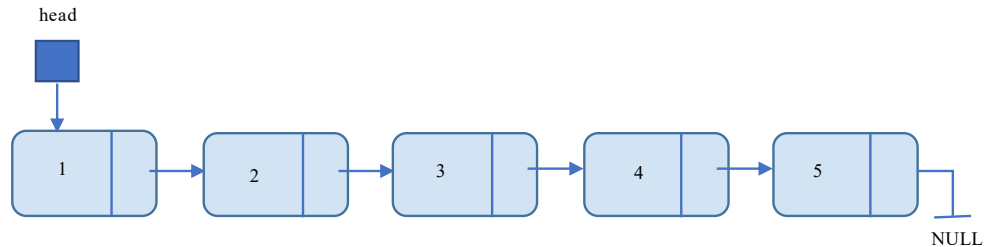


Figure 4.38. A singly linked list provided to the participants in recognition questions.

1. Use one sentence to explain what the following function does.

```

struct node* A(struct node* head)
{
    if (head == NULL || head->next == NULL)
        return head;

    struct node* curr = head->next->next;
    struct node* prev = head;
    head = head->next;
    head->next = prev;

    while (curr != NULL && curr->next != NULL)
    {
        prev->next = curr->next;
        prev = curr;
        struct node* next = curr->next->next;
        curr->next->next = curr;
        curr = next;
    }

    prev->next = curr;

    return head;
}

```

Figure 4.39. Recognition question 1 for swapping two adjacent nodes in a singly linked list.

Quick to respond without reading and tracing all of the code: Although all participants but Suzy recognize that the code swaps nodes, it is surprising that none of the participants realize that the code is explicitly swapping adjacent nodes. Interestingly, we find that seven participants (Joe, Bob, Bill, Max, Feng, Xeng, and Nate) conclude that the function reverses the nodes in the list, and almost all of them (except Bill and Max) fail to reason about the while loop. These students stop tracing the code after recognizing that the first two nodes are swapped, instead of looking over all the code. The students seem to link this function to one of the assignments they did with reversing the list, rather than reading and tracing the entire code for confirmation.

Failure to redraw or start with the same provided picture of the linked list: All participants (except Joe, Bob, and Nate) draw pictures while reading the function (except Feng, who initially draws). However, they do not redraw or start with the same provided list. Bill draws more nodes than what is in the provided figure 4.38 (see Figure 4.40), and four students (Max, Phil, Feng, and Xeng) draw fewer nodes than the list has,

which makes them incorrectly solve the problem (see Figure 4.41 for Xeng's sketches). We also find that some students (Bill, Phil, Feng, Xeng, Chemi, and Ecer) do not write the integer values inside the nodes to help them visualize the node swapping and correctly find the function purpose (see Figures 4.40 4.41 for examples).

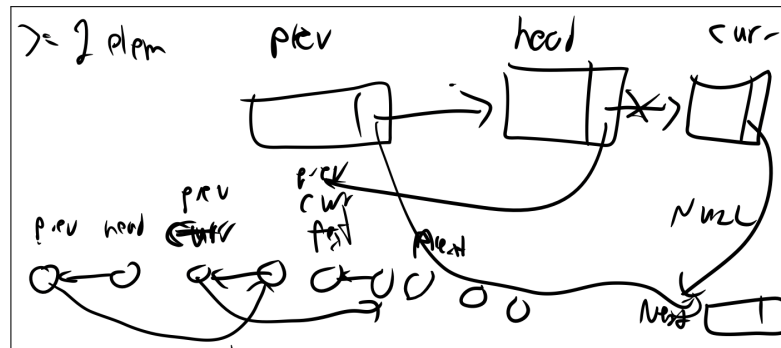


Figure 4.40. Bill's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.

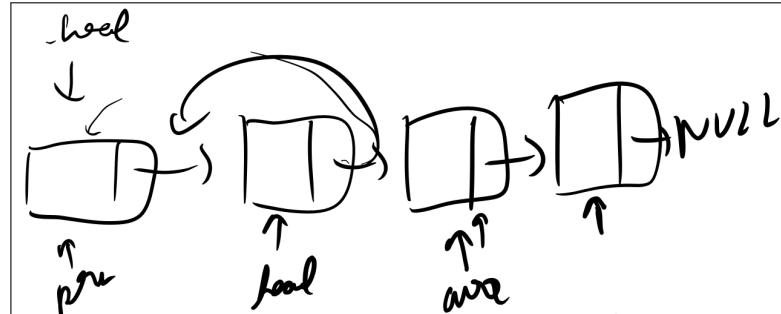


Figure 4.41. Xeng's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.

Confusion around the purpose of returning the head pointer variable from the function: Only Joe, Bill, Ecer, and Nate state that the purpose of returning the new head pointer variable value is to update the head pointer variable pointing to the beginning of the list. Suzy, Phil, and Chemi claim that the return is for returning one node, even though Suzy wonders why to return only the head pointer variable. These students also think that the return type at the end of the function is a node type, rather

than the memory address of the node at the beginning of the linked list. It seems that they do not think about the head pointer variable needing to be returned if the first two nodes are swapped. This is similar to some of the participants not thinking about updating the head pointer variable value when inserting or deleting at the beginning of the list, due to structuring their code in such a way that automatically updates the head pointer variable inside a list structure containing the head pointer.

Confusion around reassigning a next pointer variable using a pointer to the previous node: Suzy and Chemi get confused with reassigning the next node pointer variable to the previous node using the curr pointer variable that points to the previous node, i.e. `curr ->next -> next = curr;`. In their drawing, they move the position of the curr pointer variable to `curr -> next -> next`, rather than moving the `curr -> next -> next` pointer to what the curr pointer variable points to. This indicates students may not understand the reassignment of a node's next pointer variable using the next pointer variable of the previous node (see Figures 4.42 and 4.43). Due to the mistakes with the node pointer variable assignment with the `curr -> next -> next` statement mentioned above, Chemi gets confused when the fourth node gets lost without any pointer variable pointing to it. After that, Chemi asks, "Is this removing the middle node of the list?" The student rethinks the question again and wrongly concludes, "In the case where you don't do a while, ummm, the first and second node swap places. And in the case that you go into the while, I'll just say like, you leak every other node."

Although Suzy draws the list with values correctly, she fails to read or assign the node pointer variables in the drawing. For example, on the lines in Figure 4.39 where the `curr` and `prev` pointer variables are created and assigned values, Suzy sets the `curr` node pointer variable to the second node rather than the third node, and she sets the `prev` node pointer variable on the top of the head pointer variable (see Figure 4.43), which we assume means it points to the head pointer variable. The student does not seem to understand that `prev = head` means setting the `prev` node pointer variable to what the `head` pointer variable points to (i.e., to the first node). At the end, Suzy concludes that the function is for "checking if the head is the only element in there or while if it isn't, then they're changing the positions of the elements of current, previous and next." Suzy not only struggles in coding, as we demonstrated previously, but she also struggles with pointer variable manipulation.

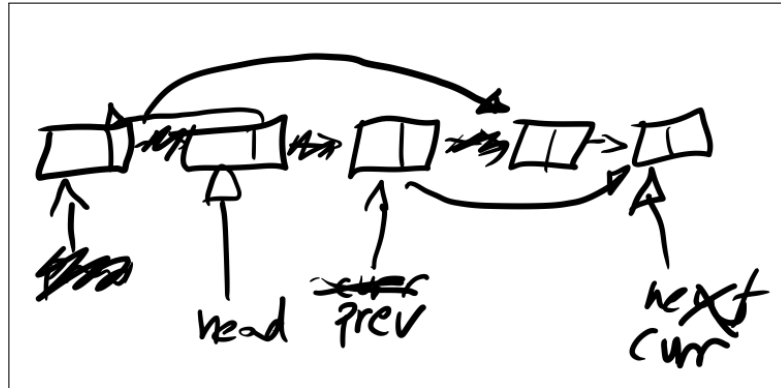


Figure 4.42. Chemi’s sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.

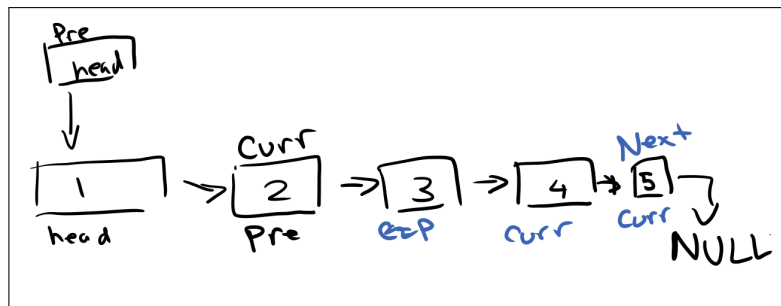


Figure 4.43. Suzy’s sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.

Unaware of good strategies for reading and tracing code: Max is able to trace the code correctly, but he draws only four nodes, which makes him initially include the NULL in one of the nodes and then make the `curr` and `next` pointer variables point to the NULL (See Figure 4.44). When Max manipulates the node pointer, he does not draw step by step with the code to reduce the confusion of swapping nodes. When he redraws a list for clarity and sees the nodes change position, he makes the `head` pointer variable point to nothing. This is because he does not read the line where the `next` pointer variable pointed to by the `head` pointer variable is set to point to the same as the previous pointer variable, making him believe the code is removing the node that is at the beginning of the list. He concludes eventually that the function “seemingly removes the second element

and reverses the list.” This suggests that some students do not use good strategies to read and trace the code to successfully identify the code’s purpose.

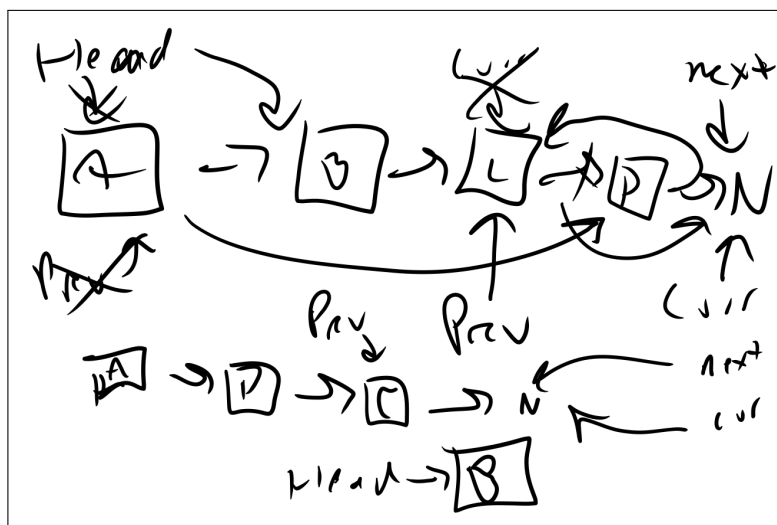


Figure 4.44. Max’s sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.

In the beginning, Ecer reads the code without drawing or writing anything and reasons, “this is doing a lot of going forward into the node and swapping the current next next equal current. I think that’s what it’s doing. I think this function returns the end of the node basically, or the end of the list, I mean.” Then, he draws while tracing the code again. He can correctly trace the code and manipulate the pointer variable, but he does not make the changes when assigning the head’s next to point as the previous pointer and the previous pointer to point as the current pointer points to. Eventually, Ecer wrongly concludes, “So, I think this function is making a single circular, linked list. I think that’s what it’s doing and then returns the head of that list, I think” (See Figure 4.45).

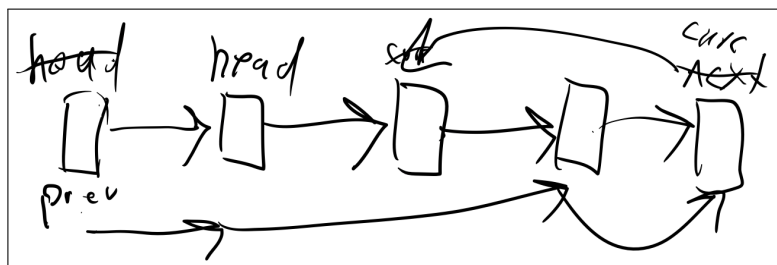


Figure 4.45. Ecer's sketches while solving recognition question 1 for swapping two adjacent nodes in a singly linked list.

Lack of Knowledge about passing a double-pointer value to the function:

Chemi is the only student who reads the code to construct the linked list, which uses the push function, before trying to figure out what the code does that swaps nodes. In the push function, Chemi gets confused about passing a double-pointer value to the function to (see Appendix D). He states, "... Slightly different implementation from mine ... I was just a little confused when I saw that..." This result suggests that students may not have experienced using this style of coding or understand how and why to pass the address of a head pointer variable to a function.

Confusion around the node pointer named 'next': Some students may get confused about the name of the `next` node pointer variable. For example, when Suzy, Bill, Feng, Chemi, and Ecer see the declaration and initialization of the `next` pointer variable, i.e. `struct node* next = curr -> next -> next;`, they state that it is making a new node called `next`. We also notice that Suzy, Bill, Phil, Ecer do not draw the node pointer variables as a separate component than the node. Instead, they put just the name on the top of the node, and it seems like nodes' names. All of these students, except Bill, show misunderstandings around the pointer variable and memory management. It would be better in the future to differentiate between the name of the node pointer variable pointing to a node in the list and the name of the node pointer member in the node structure. Instructors also need to teach students to draw a correct representation of a linked list and its pieces to build an accurate mental model.

4.1.2.3.5 Iterating

The following operations do not require any memory management, but there is pointer manipulation to update a node pointer variable for traversing a list. Students need to iterate through a list to 1) find the length of the list, 2) find a particular value stored in the list, and 3) print the list values. We ask students to write a code/pseudocode to find the length of a list; whereas, we ask students to recognize the code for finding a particular value in the list and printing the list values.

Finding the length

Coding question 5 in the interview asks participants to *"Write code/ pseudocode to find the length of the linked list"*, i.e. find the number of nodes in the linked list. To do this, participants need to 1) create a loop to go through all the nodes in the list, 2) traverse the list from one node to the next node, and 3) increment a counter that counts the number of the nodes in the list.

Looping until the next node is NULL: Interestingly, we find that five participants (Joe, Suzy, Max, Ecer, and Nate) check if the node's next in the loop is NULL, rather than the node itself (i.e., `while(current -> next != NULL)`). This causes the students to write an extra line of the code outside the loop to count the last node (Joe) or not count the last node (Suzy, Max, Ecer, and Nate), which leads these students to incorrectly count the number of nodes in the list. In the case the list is empty with a head pointer variable referring to NULL, this implementation will not work because they try to access a node that does not exist. Therefore, they cause a segmentation fault for dereferencing a pointer pointing to NULL.

Continued misunderstanding of how to use a node's next pointer variable to traverse a list: Students need to traverse a list when finding a list's length. As with inserting a new node to the end of a list (see Section 4.1.2.3.2, we find that Feng struggles to iterate through the list. He writes `next = list->head->next;`, instead of `next = next->next;`, to update the next pointer variable used to iterate through the list (see Figure 4.46 line 34). Feng fails to move the pointer to the end of the list; instead, he moves the pointer to the second node each time in the loop. This confusion may be because the student uses the name 'next' for both the node pointer variable used to traverse the list and the node pointer member in the node, or they are getting confused

between the list pointer variable and the node pointer variable.

```

29 int get_size(struct link_list* list){
30     int count = 0;
31     struct node* curr, * next= list->head;
32     while(temp != NULL){
33         count+=1;
34         next=list->head->next;
35         curr = next;
36     }
37     return count;
38 }

```

Figure 4.46. Feng’s response to finding the length of a linked list.

Unaware of using uninitialized/undeclared node pointer variable:

We find that some students use an undeclared (Suzy, Bill and Feng) or an uninitialized node pointer variable (Joe, Suzy, Bill, and Feng) to iterate through the list. Undeclared node pointer variables will be caught by the compiler, but the students do not compile their code. However, uninitialized pointers can lead to a segmentation fault when it points to somewhere in memory that cannot be accessed or could lead to incorrect results when the pointer points to some accessible data different than the desired list.

If the students trace or run their code, they may detect their errors and correctly find the length. For example, Feng declares two node pointer variables with a comma in between 'curr' & 'next', but he only initializes 'next' (see Figure 4.46). Actually, the 'curr' pointer variable has no real purpose, but Feng uses temp, which is never declared. When the student declares node pointer variables with a comma in between, we are unsure whether they intend the initialization value for both node pointer variables. This could be the case because the student uses this style in all the following coding questions in which they need to traverse the list.

Incorrectly checking for the last node: Feng at least knows to check if the node pointer variable used to traverse the list is NULL. Five students (Joe, Suzy, Max, Ecer, and Nate) used `while(curr->next != NULL)` to find the length of a list, which will cause a runtime error if the list is empty, and if the list is not empty, then the length will be one less than it should be. To fix this problem, Joe adds one at the end, but this does not fix the runtime error with an empty list.

Although Suzy correctly writes the iteration syntax when checking for an empty list operation, she fails to iterate to find the length of the list. The student shows an understanding of the meaning of finding the length, but she writes in comments and misses essential concepts like passing the list to the function, declaring and initializing the node pointer variable, and iterating through the list.

```

46 int ls_length(){
47     //initilize int i = 0
48     //while cur->next != NULL add 1 evey time to i
49     return i
50 }

```

Figure 4.47. Suzy’s response to finding the length of the linked list.

Misunderstanding of using the recursive method to find the length of the linked list: Chemi has a misconception of finding the length of the linked list using only the recursive method. The student believes that “Yeah, can’t do it iteratively, I wouldn’t make sense. You’d have to do it recursively. I’m fairly sure.” Moreover, the student uses an unnecessarily intermediate function to pass only the list’s head and another function for finding the length of the list (see Figure 4.48). However, by introducing this unnecessary function, Chemi fails to include a return in front of the call to the `node_length()` function inside the `list_length()` function.

```

27 int list_length(struct list l){
28     node_length(l->head);
29 }
30
31 int node_length(struct node* n){
32     if(n == NULL){
33         return 0;
34     }else{
35         return 1+node_length(n->next);
36     }
37 }

```

Figure 4.48. Chemi’s response to finding the length of the linked list.

Finding a Value

We ask students to recognize a function for finding a desired integer, represented by

'x', in the linked list (see Figure 4.49). The function loops to the end of the list and compares each node's data member with an integer number, 'x'. The function returns 'true' if 'x' exists in the list; otherwise, it returns 'false'. We expected everyone to correctly answer this question because it is similar to, yet simpler than, adding and deleting a node at a specific location.

2. Use one sentence to explain what the following function does.

```
bool B(struct node* head, int x)
{
    struct node* current = head;

    while (current != NULL)
    {
        if (current->data == x)
            return true;
        current = current->next;
    }
    return false;
}
```

Figure 4.49. Recognition question 2 for finding the x value in a singly linked list.

We find that everyone but Suzy correctly recognizes what the function does. Suzy gives the correct solution at the beginning, but then she changes her answer and mentions, "I might have to revise what I said, because I forgot it was inside a while." The student mentions the function returns 'true' if the "current data" is equal to the integer 'x' and returns 'false' if current is NULL (which means empty list). Suzy says, "So, if the current value is an actual value it's not NULL, then inside the if statement it checks if the current data is equal to the integer x and then it returns true. But inside of the while, it keeps on looping until it meets the requirements of the if statement, and then it will return true and if the current is NULL return false." The student does not state if the 'current' node pointer variable reaches NULL and integer x is not found, then return 'false'. The student traces the code line by line rather than correctly giving the overall purpose of the function. This shows that Suzy struggles not only in writing code but also in reading and tracing coding.

Moreover, we find that two participants (Suzy and Nate) claim that they have not seen a 'Boolean' data type in the C language before when they saw the function returns 'bool' data type. They may come to this conclusion because they have never seen the `stdbool.h` header included and the `bool` data type used in C.

Printing a List

The third function that the students need to recognize is for printing the entire data stored in the linked list nodes (see Figure 4.50). As we expected, we find that all participants correctly answer this question.

3. Use one sentence to explain what the following function does.

```
void C(struct node* head)
{
    struct node* current = head;

    while (current!= NULL)
    {
        printf("%d ", current->data);
        current = current->next;
    }
}
```

Figure 4.50. Recognition question 3 for printing the values stored in the singly linked list nodes.

4.1.2.4 Types

In the survey and interview, we ask students about four types of linked lists: singly linked list (SLL), singly circular linked list (SCLL), doubly linked list (DLL), and doubly circular linked list (DCLL). Each type of lists has characteristics and features that differentiate it from the other types of lists, such as the number of node pointer members in the node, the value of the next pointer variable in the last node, the value of the previous variable in the first node, and the visual representation of the list. Even though we only ask students to write and recognize code for SLLs, we ask students conceptual questions about all four types of linked lists in the survey and interview.

In the survey, we provide a picture of each type of linked list types mentioned above, and we asked, “*What type of list does the following drawing represent?*”. We find that all participants can identify a picture of the SCLL and DCLL, even though they did not cover them in the class. However, some students cannot correctly identify the types they have learned about in class. Only six students recognize the picture of the SLL, and nine students identify the image of the DLL. Students may not recognize a detail in the picture or the pictures look unfamiliar to them due to the tail pointer variable in the picture of the SLL type.

In the interview, we ask participants to go beyond recognizing pictures of linked lists, and we ask them to describe each type of linked list. We explicitly mention that they are allowed to draw in all the questions because some students may think that they are not allowed to draw anything on the paper or on the tablet, and drawing pictures helps students reason better while solving problems [36].

We are looking for students to describe the structure of the linked list, i.e. whether it is linear or circular, and they need to mention 1) how each node connects to another node, 2) what each member of the node stores, and 3) the benefit from including the NULL pointer value at the end of a non-circular list and at the head of any empty list. Students need to express the following scenarios when describing a linked list: the zero node case (empty list), one node, and more than one node. If students miss one of these scenarios, especially the empty list case, they miss a very significant part of the linked list, such as creating an empty linked list in the code, as we discussed earlier in Section 4.1.2.2.4.

We find that Joe is the only one who successfully describes the DCLL in the interview. Students who performed the lowest on this question are the students who did not draw or visualize the linked list on paper or the tablet as they were describing it. This is not that surprising, since a doubly circular linked list is the most complex type of linked list.

Most students can correctly discuss the details when asked directly about individual pieces of a linked list, but they are not detailed enough in these broader questions about the types of linked lists. For example, they do not describe the node structure or the benefit of the NULL value. They may not think to be as specific as the rubric requires when describing a type of linked list. Sometimes they talk about certain pieces of a list in one linked list type and forget to talk about these pieces again when describing a different type of linked list, or they draw a box to represent a node without dividing it into data

and one or two pointer sections. We also find that only Joe and Nate mention the node's memory address that is stored in the pointer variable/s to point to the next/previous node, while the majority of others neglect mentioning how the nodes are connected with addresses.

None of the participants mention that the head pointer variable is pointing to NULL to represent an empty list when describing any of the linked list types, and Phil and Ecer do not mention the head pointer variable at all. They may think that the head pointer variable is not an essential component that needs to be discussed when describing linked lists, but it is the second most essential component after a node structure. Even when describing linked list types, Nate continues to believe that the head is the first node without data in it, and he states that there is "a pointer to it from somewhere else." The confusion could be that they call the first node a head, which is usually the term used to refer to the pointer to the first node and not the first node itself. In addition to Nate, we see this with Bob and Xeng when they call the first and last nodes the head and tail, but unlike Nate, Bob and Xeng understand that the head and tail are separate pointer variables to the beginning and the end of the list. It seems that many students use the head and tail node terminologies for the first and last nodes in the list, in addition to the head and tail pointer variables, but Nate gets confused and believes that the first node is the head pointer variable.

The majority of the students draw or say something about the NULL pointer at the end of a singly and doubly linked list, but the majority of the students do not state the benefit of including the NULL pointer at the end of singly or doubly linked list. Only Bob and Ecer (in a singly linked list) and Xeng and Ecer (in a doubly linked list) do not draw or state that the last node is pointing to NULL when explaining singly and doubly linked lists.

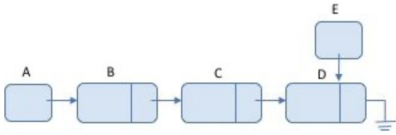
In the following subsections, we describe students' specific reasoning and misunderstandings about each different type of linked list assessed in the survey and interview questions. We begin with the singly linked list, followed by the singly circular linked list, doubly linked list, and then doubly circular linked list.

4.1.2.4.1 Singly Linked List

We define a singly linked list in C as a unidirectional, linear structure of nodes that are connected together with pointers that refer to the next node in the list. Each node has two members for the data, which can be of any type, and the node pointer, which is an address to the next node or NULL in the case of the last node to indicate the end of the list. There is a node pointer variable called the head (or head pointer) that points to the first node in the list or NULL in the case of an empty list. Students must include all pieces in their description to show a complete understanding of a singly linked list in the think-aloud interview.

Incorrectly identify picture of SLL as SCLL: As discussed in Section 4.1.1 on the accuracy of students' mental models of linked lists, we provide a picture of a singly linked list with head and tail pointer variables in the survey, and we ask the students to identify the type of the list given five multiple-choice options with the four types of linked lists and an "Other" option (see Figure 4.51 in Section 4.1.1 above). We find that only six participants correctly answer this question. Joe and Ecer incorrectly choose a singly circular linked list, and Max, Chemi, and Nate mistakenly select the "Other" option, while stating that it is a circular linked list.

Q4. What type of list does the following drawing represent?



Singly Linked List
 Doubly Linked List
 Singly Circular Linked List
 Doubly Circular Linked List
 Other

Figure 4.51. Question 4: Identify a singly linked list type.

It seems that most students have misunderstandings about the picture representing

a circular linked list, and maybe this is because there is a tail pointer variable in the picture that makes it confusing. Here are some of the reasons for why Max, Chemi, and Nate, as well as a survey-only participant, chose the "Other" option:

- "I cannot say it's a doubly linked list as there aren't pointers in a backwards manner (D->C->B->A). There appears to be both a head (A) and tail (E) pointer, so it's not a standard Singly Linked List. I'm not sure what a Singly Circular Linked List is, but this appears to be it."
- "Singly Linked List with a tail pointer. I'm not sure if that's the same as a circular linked list"
- "It is close to a singly linked list, but the addition of E makes it not."
- "Does not point backwards"

If students notice the NULL pointer at the end of the list and truly understand why it is there, then they should not identify this as a circular linked list, even if they have not talked about the circular linked list in the class. Comments also suggest that students do not understand that the type of linked list does not change if the tail component exists or not. The tail component is optional in all linked list types.

Failure to mention NULL or update nodes with NULL value: In the interview, we ask the students to *"Describe a singly linked list, and draw a picture if needed"*. We find that Ecer, who chooses the singly circular linked list option in the survey, fails to mention the NULL pointer value when describing the node's node pointer member in both singly and doubly linked lists. He also fails to make the last and second to last node point to NULL when adding and deleting a node at the end of a singly linked list in the coding questions. This shows that Ecer does not have a deep procedural understanding of the NULL concept, even though he correctly states the benefit of NULL when explicitly asked in the survey.

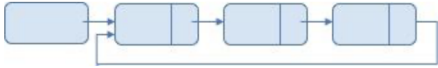
4.1.2.4.2 Singly Circular Linked List

Like a singly linked list, we define a singly circular linked list in C as a unidirectional, circular structure of nodes that are linked together with pointers that refer to the next node in the list. Just as with a singly linked list, each node has two members, the data

and an address to the next node in the list, with the exception of the last node that points back to the first node in the list making it circular. There is still a node pointer variable named head, which points to the first node in the list or NULL in the case of an empty list.

In the survey, we provide a picture of a singly circular linked list with only a head pointer variable, and we ask students to identify the type of list (see Figure 4.52). All students correctly identify the picture of the singly circular linked list. It seems that the students can correctly identify the type when the drawing looks like a circle, which may confirm the students' confusion when seeing the tail pointer variable at the end of the 'singly linked list' picture.

Q12. What type of list does the following drawing represent?



Singly Linked List
 Doubly Linked List
 Singly Circular Linked List
 Doubly Circular Linked List
 Other

Figure 4.52. Question 12: Identify a singly circular linked list type.

Similar to the singly linked list, in the interview, we ask participants to “*Describe a singly circular linked list, and draw a picture if needed*”. The students can recognize a circular linked list structure when presented with a picture of the last node going back to the beginning of the list, but the major problem is the students are not specific and detailed when describing or/and drawing the list, as mentioned previously.

Incorrectly thinks the last node and the first node points to NULL: Max, who incorrectly identifies the singly linked list as a singly circular linked list, believes that the last node of a singly circular linked list points to NULL and points to the first node in the list. The student claims that this helps to traverse back to the beginning of the list by stating the following, “I would just have a conditional statement saying if

we're at you know if we're at the NULL, then allow us to go to the head.”

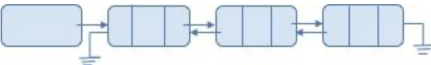
Confusion about which node to begin with in the SCLL: On the other hand, Suzy confuses a circular linked list with an array. When explaining the circular linked list, the student starts iterating from the middle of the list to the beginning of the list without drawing the head pointer variable. Although the student knows the head pointer variable points to the beginning of the list when describing the head concept, she gets confused about the beginning of the list from the left or right side when creating an empty list. In either case, it is clear that this student lacks the conceptual understanding of a singly circular linked list.

4.1.2.4.3 Doubly Linked List

We define a doubly linked list in C as a bidirectional, linear structure of nodes that are linked together with two pointers that refer to the next node and the previous node. Each node has three members: the data, the address of the next node, and the address of the previous node, with the exception of the first and the last node that point to NULL to indicate the beginning and end of the list. There is still a node pointer variable called the head that points to the first node in the list or NULL for an empty list.

Misidentify a DLL as a circular list: In the survey, we provide a picture of a doubly linked list with only a head pointer variable, and we ask students to identify the type of list (see Figure 4.53). We find that nine students correctly identify the doubly linked list picture. However, Suzy selects a doubly circular linked list, and Xeng chooses a singly circular linked list.

Q13. What type of list does the following drawing represent?



Singly Linked List
 Doubly Linked List
 Singly Circular Linked List
 Doubly Circular Linked List
 Other

Figure 4.53. Question 13: Identify a doubly linked list type.

Both Suzy and Xeng ignore the NULL pointer at the end of the list and select a circular linked list type. These students have no issues explaining a DLL in the interview and show conceptual understanding of a doubly linked list, but they do ignore the NULL value in their description during the interview. In this survey question, they either accidentally choose the wrong type or the two electrical ground symbols for NULL confuse them, even though we provide an explanation of this symbol.

Incorrectly assume a DLL must have a pointer variable to the last node: Just as with the singly linked list types, we ask participants to “Describe a doubly linked list, and draw a picture if needed” in the verbal section of the interview. We find that Max and Phil have a misconception that a doubly linked list has to have a tail pointer variable. They may have learned that using a tail with a doubly linked list improves performance, as Joe mentions when describing a doubly linked list. Joe says, “I guess that might be a good idea and also have a tail pointer variable, which points to the last node in the list.” Due to students’ limited thinking, they cannot elaborate on the use of the tail pointer variable with any other linked list.

Lack of attention to the importance of NULL for the first node’s previous pointer variable: In a doubly linked list, to indicate the end of the list in both directions, the first node’s previous pointer variable and the last node’s next pointer variable point to NULL. We find that five participants out of 11 (Joe, Suzy, Xeng, Chemi, and Ecer) do not mention the first node ever pointing to NULL when describing a doubly linked list, but that

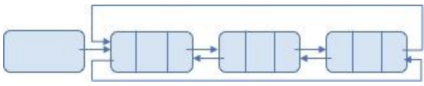
is not surprising because Joe, Suzy, Chemi, and Ecer show misunderstandings or missing knowledge about the NULL concept in other questions. For example, Joe and Suzy forget about NULL when creating an empty singly linked list, and Ecer does not mention the NULL pointer value in most of the interview questions, as discussed in Section 4.1.2.2.3.

4.1.2.4.4 Doubly Circular Linked List

Lastly, we define a doubly circular linked list in C as a bidirectional, circular structure of nodes that are linked together in two directions with two pointers that refer to the next node and the previous node. Each node stores three parts: the data, the address to the next node, and the address to the previous node, except for the first and last nodes that point to the last node as the previous and the first node as the next respectively. Just as with all the previous linked lists types we defined, there is a node pointer variable called the head, which points to the first node in the list or NULL for an empty list.

In the survey, we provide a picture of a doubly circular linked list, and we ask the students to identify the type of the list (see Figure 4.54). As with the singly circular linked list, we find that all participants correctly choose a doubly circular linked list. It seems that students can easily recognize a circular linked list in a picture, regardless of whether it is a singly or doubly linked list.

Q11. What type of list does the following drawing represent?



Singly Linked List
 Doubly Linked List
 Singly Circular Linked List
 Doubly Circular Linked List
 Other

Figure 4.54. Question 11: Identify a doubly circular linked list type.

As with the other types of lists, we ask the students to “*Describe a doubly circular*

linked list, and draw a picture if needed". Suzy and Feng mention uncertainty when answering this question. They state that this is the first time they have talked about a doubly circular linked list. Feng correctly draws the list, while Suzy is not confident when describing it. Suzy mentions that it is like the doubly linked list, but it can circle from the front to the end and end to the front. However, she only shows and states that the last and first nodes are connected together in both directions, and she does not draw two pointers out of each node pointing to the next and previous nodes (see Figure 4.55). Suzy correctly identifies the doubly linked list and doubly circular linked list in the survey, but she cannot correctly describe them in the interview.

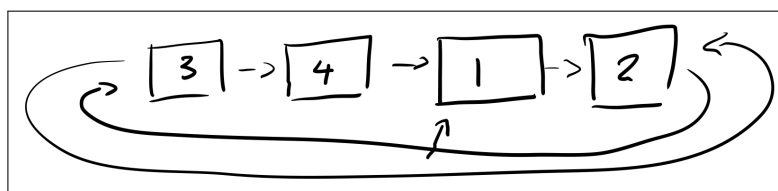


Figure 4.55. Suzy's sketches while describing a doubly circular linked list type.

4.2 RQ2: What difficulties do students face while learning about linked lists in the C programming language?

After assessing students understanding of linked lists in C, we ask students in the survey and the interview about any difficulties they had learning and understanding linked lists while taking the data structures class, CS 261, at Oregon State University. We find that linked lists are not only difficult for students learning about them for the first time in CS 261 but also for those who learned about them before CS 261. Students find that the conceptual understanding of linked lists is easy, but they struggle the most with their implementation due to pointer manipulation. They are also confused between linked lists and arrays and cannot differentiate when and where the two are more or less useful.

From the survey results, we find that a few students, who learned about linked lists before CS 261, state having a hard time understanding abstract data structures because they do not have any examples of their uses in the real world. For example, one of the survey responses notes, "Understanding drawings aren't hard but the code is because we aren't shown the code or a real life application of the linked list. It would be really cool

if we could do a real life scenario with code and we would walk step by step through it instead of drawing it all” In addition, other students comment on pointer manipulation making it hard to implement linked lists, regardless of whether they had knowledge about linked lists before CS 261. One student, who learned about linked lists prior to CS 261, makes the following comment about reversing a list, “Learning how to reverse a linked list properly – the pointer management and setting the head of the list correctly at the end is difficult.”

Some students find linked lists concepts more difficult than arrays. This is because you can directly access the items in an array with square brackets, which do the address arithmetic for you without the need for direct pointer manipulation. In addition, as one student says, “they didn’t understand the value linked lists had over an array or what situations you would use them for.” Students seem to be exposed to arrays more than linked lists in the first-year classes and have not been taught the advantages and disadvantages between arrays and linked lists in their data structures class.

In the interview, we ask the students again about any difficulties in understanding and learning linked list while taking the CS 261 data structures class. Surprisingly, we find that 55% of the students (Joe, Bob, Max, Phil, Feng, and Xeng) claim that they did not have any difficulties while learning linked lists. This could be because all of them, except for Phil, have prior knowledge about linked list before the data structures class and may have faced difficulties in the class before the data structures class, as Joe mentions. However, Joe, Phil, and Feng show procedural misunderstandings about some linked list concepts, such as using node pointer variables and accessing data members in a node as discussed in Section 4.1.2.2, even though Feng states that linked lists are very easy to learn and understand.

Other students (Suzy, Chemi, and Nate), who have experience with linked lists in a programming language that does not require direct pointer manipulation, like Java and Python, claim that they have difficulties learning linked lists in C, and their experience might have affected their performance in solving problems in C. Suzy, who retok the class, finds linked lists to be one of the most challenging topics in the data structure class when she took the class for the first time. She finds learning linked lists hard because the instructor only spends one class explaining them, and the linked list has too much information to absorb during one lecture. Suzy, instead of trying to understand linked list concepts, finds that memorization is the main challenge of coding linked lists, as well

as the challenge with the previous instructor's testing style.

In the interview, Chemi, Ecer, and Nate mention that they do not have an issue with the conceptual understanding of linked lists, but they find it hard to code them. Bill also finds learning linked list challenging, and he struggles the most with pointer variables, as with many other students. He finds that the most challenging concept is iterating through the list to the n th node and keeping track of the previous node pointer when deleting at the end of a singly linked list without a tail pointer variable. Bill spends time understanding his weaknesses while learning linked lists in CS261, and he is one of the students who get the higher coding scores. Bill also claims that there is less focus on linked lists than on arrays in previous classes, as well as in CS 261. As a result, we see some students find the list length to iterate to the end of a list, rather than taking advantage of the NULL pointer at the end of the list to reduce the time complexity of traversing the list twice.

In addition, Nate says there is no need to use the linked lists because "we have access to dynamic arrays." He mentions the advantage of direct access versus linear time added by the complication of iterating in linked lists, but the student does not seem to understand the limitations of the dynamic array over the importance of linked lists in some situations and where linked lists are helpful in real applications.

As explained previously, they struggle the most with pointer manipulation, and Nate finds that deleting and reversing functions are the most challenging. All students show misunderstandings of pointers, memory management, and some pieces of a linked list.

From the survey and the interview results, we find that students conceptually understand linked lists, but they struggle with C syntax and the prerequisite knowledge about pointers and memory management. They also fail to understand the reasons for using linked lists over arrays. It could be very helpful for the students to visualize how linked lists are used in the real-world and walking through code with the examples. In addition, educators need to pay more attention to explaining the advantages and disadvantages between linked lists and dynamic arrays and apply their uses to different types of lists correctly when needed.

4.3 RQ3 & RQ4: The Purdue Visualization of Rotations Test (ROT)

In the previous sections, we discuss the accuracy of students' mental models of linked lists, their conceptual and procedural misunderstandings, and the difficulties that students face while learning linked lists. This section shifts the focus to investigating visual-spatial reasoning as a contributing factor to student struggles with abstract data structures. We use the Purdue Visualization of Rotations Test (ROT) to measure students' visual-spatial reasoning, and we correlate students' understanding about linked list concepts and drawing pictures with their ROT scores. We hypothesize that there is a relationship between students' ROT scores and both their understanding of linked list and drawing pictures while solving problems. We explain the Purdue Visualization of Rotations Test and individually address each question below.

Research shows that students who score at least 60 percent or higher (12 points out of 20) on the ROT generally have the necessary spatial visualization skills to succeed in engineering, chemistry, math, and physics courses, and students with a score less than 60 percent (below 12 points) need to develop better visual-spatial skills to be successful in the courses within these majors [105, 84]. Out of the 11 students from the interview, the highest ROT score in this study is 20 points, and the lowest score is 13 points (see Table 4.4). The average score obtained by the participating university students on the ROT is 16 points.

	N	M	Min.	Max.
ROT	11	16 (80%)	13 (65%)	20 (100%)

Table 4.4. Spatial Visualization Test Score. Note that N is the sample size, M is the mean, Min. is the minimum score, and Max. is the maximum score.

None of the participants score below 60%, which indicates that all participants have at least the basic visual-spatial skills. It is interesting to note that the Chemical Engineering student (Chemi) scored a perfect score on the ROT, and Suzy, who seems to struggle the most with linked lists, is one of the lowest ROT scores. It is also interesting that Chemi has the highest GPA (3.98), and Suzy and Max, who receive the lowest ROT scores (65%), have the lowest GPAs (2.88 & 2.97, respectively). This is not the case for all participants, such as Xeng, who has an above average GPA (3.36) and receives a below average ROT score (70%). In the following subsections, we analyze correlations between

students' ROT scores and their understanding of linked lists and drawing pictures while working with linked lists to determine if visual-spatial reasoning is a contributing factor to students successes or struggles with understanding linked lists in C.

4.3.1 RQ3: What is the relationship between students' understanding about linked lists and their visual-spatial reasoning?

To answer this research question, we use the students' overall scores on the linked list concepts measured in RQ1.1 in Section 4.1.1 with their ROT scores (see Table 4.5). While the highest overall score on linked list concepts is 80%, we notice that the student with this highest score does not have the highest ROT score, and the student with the highest ROT score has an "Overall LL" score of 71. Also, there does not seem to be a pattern with the lowest ROT scores and the lowest overall linked list scores, except with the case of Suzy. At this time, we do not see a relationship between students' reasoning about linked lists and their spatial visualization ability.

	Accuracy Score Per Section			Overall LL	ROT
	Overall Types	Overall Pieces	Overall Operations		
Bob	60	72	95	80	15 (75)
Bill	70	67	91	77	19 (95)
Max	65	61	87	74	13 (65)
Xeng	30	69	95	74	14 (70)
Chemi	46	63	87	71	20 (100)
Phil	34	63	90	70	19 (95)
Joe	73	59	76	69	17 (85)
Feng	61	65	77	69	17 (85)
Nate	65	48	76	64	18 (90)
Ecer	44	56	74	62	16 (80)
Suzy	56	21	23	29	13 (65)

Table 4.5. Students accuracy score per section, and ROT score (ROT percentage score in the parenthesis). Note that the presented data is organized based on the high to low scores in the overall linked list concepts.

4.3.2 RQ4: What is the relationship between drawing pictures while reasoning about linked lists and students' visual-spatial reasoning?

To answer RQ4, we compare the number of times students draw pictures when answering questions about linked lists and their reasons for drawing with their ROT test scores (see Table 4.6). Since research shows that visualization or drawing pictures helps students reason better while solving problems [36], we want to see if there is a relationship between drawing pictures while reasoning about linked lists in C and students visual-spatial reasoning, which could be a factor into why some students draw more or less than others and for what reasons they are drawing pictures.

	# Drawing Per Section				Total Drawing	ROT
	Verbal	Coding	Explain Code	Recognition		
Bob	4	0	0	0	4	15 (75)
Bill	7	3	1	1	12	19 (95)
Max	6	6	0	1	13	13 (65)
Xeng	0	4	2	2	8	14 (70)
Chemi	2	0	0	1	3	20 (100)
Phil	0	0	0	1	1	19 (95)
Joe	6	0	0	0	6	17 (85)
Feng	5	0	0	1	6	17 (85)
Nate	6	3	2	2	13	18 (90)
Ecer	4	1	0	1	6	16 (80)
Suzy	6	0	0	1	7	13 (65)

Table 4.6. The Number of students’ drawings per section, and ROT score (ROT percentage score in the parenthesis). Note that the presented data is organized based on the high to low scores in the overall linked list concepts.

We notice that students who have a higher spatial visualization score draw fewer pictures when coding, which could be because they are able to correctly visualize the process of the operation in their head (see Table 4.6). Chemi and Phil, who receive a 95% or higher ROT scores, do not draw in the coding questions or use drawing to explain the coding process to the interviewer, like Bill.

Bill, in the delete at the beginning operation, mistakenly deletes the first node without using a temporary pointer variable to hold the memory address of that node. Then, he notices the problem during his think-aloud process without drawing any pictures, and he fixes his code accordingly. Chemi seems to have a good spatial ability to visualize things due to his field of study and thinking about linked lists as a “chain of atoms” that bond together. He finds that linked lists are easy to understand conceptually, but he struggles with the C language syntax, especially when dealing with pointers, memory management, and using the pointer operator ‘*’ when defining the list.

We explore the purpose of the students’ drawing while solving coding questions, and we find that only five participants (Bill, Max, Xeng, Ecer, and Nate) draw in the cod-

ing questions. We observe that these students use drawing pictures while solving the coding questions in four different ways: 1) draw first before writing the code to help them create/construct the code (Max, Ecer, Nate), 2) draw after being stuck in writing code (Ecer), 3) draw after writing the code to find bugs and check if the code works perfectly (Max, Ecer, Nate), or 4) draw after writing the code to explain the process of the operation (Bill and Xeng).

We also find that all students, who draw pictures in coding questions (Bill, Max, Xeng, Ecer, and Nate), draw a picture when deleting a node after a specific location. Only Bill and Xeng, who purposely draw after coding this operation to explain, have a complete understanding of this operation. Others, who draw to construct the code, have errors related to the pointer variable. However, we do not find any relationship between why the students draw pictures and their ROT score.

Chapter 5: Threats to Validity

We identify five threats to the validity of this study. First, we did not pilot the questions with students prior to the study, which means there are likely issues with the questions used. We asked experts to review the questions, but some experts might have overlooked some issues, such as ambiguous wording, that could have led to confusion for the student. For example, we used 'head pointer' and 'head' for the 'head pointer variable', but we were consistent with the language used in the course. Second, there is a possibility that we interpret small syntax mistakes as misunderstandings, but we triangulate issues with other data to minimize this threat. Third, we interview only A and B students. Students with lower grades might have different misunderstandings. Fourth, this study reveals the misunderstandings from these 11 participants. It is not likely that the small sample size represents all the misunderstandings or lack of knowledge students have about basic linked list concepts in C. If we conducted this same study with other students, we may or may not see the same misunderstandings, and we would likely see new ones. Finally, this study is limited to misunderstandings about linked lists in C. Some misunderstandings may or may not be generalized to other languages.

Chapter 6: Conclusion

Identifying students' misunderstandings is one of the most important aspects of CS educators' competence [92], and understanding students' reasoning about a concept can provide insight into their mental models about that concept [59]. Most previous studies for discovering students' misconceptions in CS focus on procedural understanding by asking students to write code or fill in a missing piece of code. However, lacking conceptual understanding could make learning the procedural knowledge difficult. This study contributes to educators understanding of how students reason and think about the linked list concepts in C. If students struggle with the basic data structures, then they cannot be expected to understand more complex data structures or manipulating data structures.

The education philosophers suggest that educators make sure the materials they teach are correct and do not have any misleading aspects [82], but it is hard to do this without an inventory of concepts required for a complete understanding of a specific topic. This research contributes a categorization of essential linked list concepts and a framework for assessing and evaluating students' mental models about linked list. As we can see from this initial investigation, some students face difficulties understanding linked lists in C, even if they learn about them before their data structures class. This is largely due to their misunderstandings about pointers, which is a fundamental prior concept needed to fully understand linked lists in the C programming language. For example, the concept of a node pointer variable, inside and outside the node structure, appears to be confusing, which might be because students are still struggling with pointer variable concepts in general. Not only are participants have difficulties with learning linked lists, but other participants in other studies have found linked lists to be a challenging topic in data structures [101, 116, 71].

The interview results demonstrate that these A and B students have significant conceptual and procedural misunderstandings about basic linked lists concepts in C, after learning about linked lists in their data structures class. While GPA is not directly correlated to these students' misunderstandings, the participant with the lowest GPA is the lowest performing participant in the interview, which we learn is retaking the data

structures class.

We also 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 would. On the other hand, students still struggle with prerequisite knowledge related to allocating memory with `malloc`, pointers variables and memory management concepts, and matching types in functions that impact their ability to understand and implement a singly linked list in C.

In addition to investigating students' understanding about linked lists in C, this research also explores relationships between students' performance on the linked list concepts and drawing pictures with their performance on the Purdue Visualization of Rotations Test (ROT) [8]. Even though all our participants receive a score higher than 60% on the ROT, which indicates that they have the necessary spatial visualization skills, we do not see that the students with higher ROT scores perform better on the linked list questions in the survey or interview. In fact, we see that some students who have high overall linked list scores have lower ROT scores. It could be due to their struggles thinking abstractly about 2-D or 3-D shapes or being exhausted by the end of the interview. However, we notice that students who have high visual-spatial skills draw less when coding or explaining the coding process, but this might be due to being able to visualize well in their head.

Chapter 7: Future Work

This research study contributes a categorization and comprehensive list of linked list concepts with a set of questions and example rubrics for assessing students' mental models about a singly linked list in C, but more importantly, it contributes to the future work in creating a linked list concept inventory using the misunderstandings and gaps in knowledge about linked list concepts identified in this exploratory research. As the force concept inventory authors [54] suggest, teaching students about their misconceptions is not effective in improving learning and overcoming these misconceptions. Using "a well-designed and tested instructional method is essential" to obtain visible conceptual change.

Halloun and Hestenes in [45] modified the initial diagnosis physics (mechanic) test many times over three years while administering the different versions on more than 1000 college students in introductory physics courses to validate the instrument and use the students' misconceptions in the final version of the multiple-choice test. It also took them years to interview students to build a well-known validated force concept inventory in physics that helps educators assess the commonsense beliefs of their students. However, our data are only from one data structures class. We are in the first stage in the whole process, and our plan is to collect more data from more students in different universities to build a concept inventory for linked lists.

Following Hestenes et al. method of validating the (mechanic's) test, we want to perform a content validity to our survey and interview questions using the following steps:

1. More discussion of the test questions with data structure professors and graduate students.
2. Give the questions to different graduate students to see how well they solve the questions and make changes until they all agree on the correct answers.
3. Provide the questions to more students and follow-up with students about the questions to ensure they understood them.

Bibliography

- [1] Wendy K Adams and Carl E Wieman. Development and validation of instruments to measure learning of expert-like thinking. *International Journal of Science Education*, 33(9):1289–1312, 2011.
- [2] Maizam Alias, Thomas R Black, and David E Gray. Effect of instruction on spatial visualization ability in civil engineering students. *International Education Journal*, 3(1), 2002.
- [3] Vicki L Almstrum, Peter B Henderson, Valerie Harvey, Cinda Heeren, William Marion, Charles Riedesel, Leen-Kiat Soh, and Allison Elliott Tew. Concept inventories in computer science for the topic discrete mathematics. In *ACM SIGCSE Bulletin*, volume 38, pages 132–145. ACM, 2006.
- [4] Lorin W Anderson, David R Krathwohl, et al. A revision of bloom’s taxonomy of educational objectives. *A Taxonomy for Learning, Teaching and Assessing*. Longman, New York, 2001.
- [5] Yuichiro Anzai and Herbert A Simon. The theory of learning by doing. *Psychological review*, 86(2):124, 1979.
- [6] Alan C Benander, Barbara A Benander, and Howard Pu. Recursion vs. iteration: An empirical study of comprehension. *Journal of Systems and Software*, 32(1):73–82, 1996.
- [7] Stephen Bloch. Teaching linked lists and recursion without conditionals or null. *J. Comput. Sci. Coll.*, 18(5):96–108, may 2003.
- [8] George M Bodner and Roland B Guay. The purdue visualization of rotations test. *The Chemical Educator*, 2(4):1–17, 1997.
- [9] Elizabeth Boese. Linked-list vs array in memory: An unplugged active learning experience (abstract only). In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE ’18, page 1106, New York, NY, USA, 2018. Association for Computing Machinery.
- [10] Sarah Buchanan, Brandon Ochs, and Joseph J LaViola Jr. Cstutor: a pen-based tutor for data structure visualization. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 565–570, 2012.

- [11] Ricardo Caceffo, Steve Wolfman, Kellogg S Booth, and Rodolfo Azevedo. Developing a computer science concept inventory for introductory programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 364–369. ACM, 2016.
- [12] Jason Carter and Prasun Dewan. Contextualizing inferred programming difficulties. In *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering*, pages 32–38, 2018.
- [13] Audrey 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.
- [14] Yi-Ling Cheng and Kelly S Mix. Spatial training improves children’s mathematics ability. *Journal of Cognition and Development*, 15(1):2–11, 2014.
- [15] Harrison Chotzen, Alasdair J. Johnson, and Parth M. Desai. Exploring the mental models of undergraduate programmers in the context of linked lists. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE ’19*, page 1261, New York, NY, USA, 2019. Association for Computing Machinery.
- [16] Eric Chown, Stephen Kaplan, and David Kortenkamp. Prototypes, location, and associative networks (plan): Towards a unified theory of cognitive mapping. *Cognitive Science*, 19(1):1–51, 1995.
- [17] Andy Clark. *Microcognition: Philosophy, cognitive science, and parallel distributed processing*. MIT Press, 1991.
- [18] Stephen Cooper, Karen Wang, Maya Israni, and Sheryl Sorby. Spatial skills training in introductory computing. In *Proceedings of the eleventh annual international conference on international computing education research*, pages 13–20, 2015.
- [19] Felicia-Mirabela Costea, Ciprian-Bogdan Chirila, and Vladimir-Ioan Crețu. Auto-generative learning objects for learning linked lists concepts. In *2020 International Symposium on Electronics and Telecommunications (ISETC)*, pages 1–4, 2020.
- [20] Anthony Cox, Maryanne Fisher, and Philip O’Brien. Theoretical considerations on navigating codespace with spatial cognition. In *PPIG*, page 9, 2005.
- [21] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 164–172, 2017.

- [22] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 21–26. ACM, 2012.
- [23] Herbert L Dershem, Ryan L McFall, and Ngozi Uti. Animation of java linked lists. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 53–57, 2002.
- [24] John Dewey. *How we think*. Courier Corporation, 1997.
- [25] Chris R Douce, Paul J Layzell, and Jim Buckley. Spatial measures of software complexity. 1999.
- [26] K Anders Ericsson and Herbert A Simon. *Protocol analysis: Verbal reports as data*. the MIT Press, 1984.
- [27] Mohammed F Farghally, Kyu Han Koh, Jeremy V Ernst, and Clifford A Shaffer. Towards a concept inventory for algorithm analysis topics. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 207–212. ACM, 2017.
- [28] Sally Fincher, Bob Baker, Ilona Box, Quintin Cutts, Michael de Raadt, Patricia Haden, John Hamer, Margaret Hamilton, Raymond Lister, and Marian Petre. Programmed to succeed?: A multi-national, multi-institutional study of introductory programming courses. 2005.
- [29] Sally Fincher and Marian Petre. *Computer science education research*. CRC Press, 2004.
- [30] Maryanne Fisher, Anthony Cox, and Lin Zhao. Using sex differences to link spatial cognition and program comprehension. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 289–298. IEEE, 2006.
- [31] Davide Fossati, Barbara Di Eugenio, Christopher W Brown, Stellan Ohlsson, David G Cosejo, and Lin Chen. Supporting computer science curriculum: Exploring and learning linked lists with ilist. *IEEE Transactions on Learning Technologies*, 2(2):107–120, 2009.
- [32] Eric Fouh, Monika Akbar, and Clifford A Shaffer. The role of visualization in computer science education. *Computers in the Schools*, 29(1-2):95–117, 2012.
- [33] Shane Frederick. Cognitive reflection and decision making. *Journal of Economic perspectives*, 19(4):25–42, 2005.

- [34] David Furcy. Jhavepop: Visualizing linked-list operations in c++ and java. *Journal of Computing Sciences in Colleges*, 25(1):32–41, 2009.
- [35] Rochel Gelman and Charles R Gallistel. *The child's understanding of number*. Harvard University Press, 1978.
- [36] Dedre Gentner and Albert L Stevens. *Mental models*. Psychology Press, 1983.
- [37] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey L Herman, Lisa Kaczmarczyk, Michael C Loui, and Craig Zilles. Setting the scope of concept inventories for introductory computing subjects. *ACM Transactions on Computing Education (TOCE)*, 10(2):5, 2010.
- [38] Michael Goldwasser. A gentle introduction to linked lists. In *Proceedings of the Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education: SIGCSE 2003*, page 413, 2003.
- [39] Thomas RG Green and R Navarro. Programming plans, imagery, and visual programming. In *Human—Computer Interaction*, pages 139–144. Springer, 1995.
- [40] Joy Paul Guilford and John I Lacey. Printed classification tests: Report no. 5. 1947.
- [41] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.
- [42] Mark Guzdial and Barbara Ericson. Listening to linked lists: Using multimedia to learn data structures (abstract only). In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, SIGCSE '12*, page 663, New York, NY, USA, 2012. Association for Computing Machinery.
- [43] Graeme S Halford. *Children's understanding: The development of mental models*. Hillsdale, NJ: Erlbaum, 1993.
- [44] I. A. Halloun and D. Hestenes. The initial knowledge state of college physics students. *American Journal of Physics*, 53(11):1043–1048, 1985.
- [45] Ibrahim Abou Halloun and David Hestenes. The initial knowledge state of college physics students. *American journal of Physics*, 53(11):1043–1055, 1985.
- [46] Sally Hamouda, Stephen H. Edwards, Hicham G. Elmongui, Jeremy V. Ernst, and Clifford A. Shaffer. A basic recursion concept inventory. *Computer Science Education*, 27(2):121–148, 2017.

- [47] B Hardiyana, L Fadilah, and D Effendi. Application of linked list algorithm based on multimedia. In *IOP Conference Series: Materials Science and Engineering*, volume 879, page 012087. IOP Publishing, 2020.
- [48] Sarah S. Heckman. An empirical study of in-class laboratories on student learning of linear data structures. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research, ICER '15*, page 217–225, New York, NY, USA, 2015. Association for Computing Machinery.
- [49] Lynette D Henderson and Julie Tallman. *Stimulated recall and mental models: Tools for teaching and learning computer information literacy*, volume 2. Scarecrow Press, 2006.
- [50] Geoffrey L Herman and Dong San Choi. The affordances and constraints of diagrams on students' reasoning about state machines. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 173–181, 2017.
- [51] Geoffrey L Herman, Lisa Kaczmarczyk, Michael C Loui, and Craig Zilles. Proof by incomplete enumeration and other logical misconceptions. In *Proceedings of the fourth international workshop on computing education research*, pages 59–70. ACM, 2008.
- [52] Geoffrey L Herman, Michael C Loui, and Craig Zilles. Creating the digital logic concept inventory. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 102–106. ACM, 2010.
- [53] Geoffrey L Herman, Craig Zilles, and Michael C Loui. Work in progress-students' misconceptions about state in digital systems. In *2009 39th IEEE Frontiers in Education Conference*, pages 1–2. IEEE, 2009.
- [54] David Hestenes, Malcolm Wells, and Gregg Swackhamer. Force concept inventory. *The physics teacher*, 30(3):141–158, 1992.
- [55] Robert R Hoffman and Peter A Hancock. Measuring resilience. *Human factors*, 59(4):564–581, 2017.
- [56] Robert R Hoffman, Shane T Mueller, Gary Klein, and Jordan Litman. Metrics for explainable ai: Challenges and prospects. *arXiv preprint arXiv:1812.04608*, 2018.
- [57] Christine Howe, Andrew Tolmie, Anthony Anderson, and Mhairi Mackenzie. Conceptual knowledge in physics: The role of group interaction in computer-supported teaching. *Learning and Instruction*, 2(3):161–183, 1992.

- [58] Cruz Izu, Amali Weerasinghe, and Cheryl 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*, pages 251–259, 2016.
- [59] Philip Nicholas Johnson-Laird. *Mental models: Towards a cognitive science of language, inference, and consciousness*. Number 6. Harvard University Press, 1983.
- [60] Sue Jones and Gary Burnett. Spatial ability and learning to program. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 2008.
- [61] Sue Jane Jones and Gary E Burnett. Spatial skills and navigation of source code. *ACM SIGCSE Bulletin*, 39(3):231–235, 2007.
- [62] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111. ACM, 2010.
- [63] Vinayak Teoh Kannappan, Owen Noel Newton Fernando, Anupam Chattopadhyay, Xavier Tan, Jeffrey Yan Jack Hong, Hock Soon Seah, and Hui En Lye. La petite fee cosmo: Learning data structures through game-based learning. In *2019 International Conference on Cyberworlds (CW)*, pages 207–210. IEEE, 2019.
- [64] Kuba Karpierz and Steven 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*, pages 109–114. ACM, 2014.
- [65] RJ Keeble and Robert D. Macredie. Assistant agents for the world wide web intelligent interface design challenges. *Interacting with computers*, 12(4):357–381, 2000.
- [66] Maria Kozhevnikov, Michael A Motes, and Mary Hegarty. Spatial visualization in physics problem solving. *Cognitive science*, 31(4):549–579, 2007.
- [67] Robert L. Kruse, Clovis L. Tondo, and Bruce P. Leung. *Data Structures and Program Design in C*. Prentice-Hall, Inc., USA, 1996.
- [68] NS Kumar, PN Revanth Babu, KS Sai Eashwar, MP Srinath, and Sreyans Bothra. Code-viz: Data structure specific visualization and animation tool for user-provided code. In *2021 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, pages 1–8. IEEE, 2021.
- [69] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *Acm sigcse bulletin*, 37(3):14–18, 2005.

- [70] Ah-Fur Lai, Ting-Ting Wu, Gon-Yi Lee, and Horng-Yih Lai. Developing a web-based simulation-based learning system for enhancing concepts of linked-list structures in data structures curriculum. In *2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, pages 185–188. IEEE, 2015.
- [71] Lucas Layman, Yang Song, and Curry Guinn. Toward predicting success and failure in cs2: A mixed-method analysis. In *Proceedings of the 2020 ACM Southeast Conference*, pages 218–225, 2020.
- [72] Hanyu Lin. Influence of design training and spatial solution strategies on spatial ability performance. *International Journal of Technology and Design Education*, 26(1):123–131, 2016.
- [73] Katherine D Lippa, Helen Altman Klein, and Valerie L Shalin. Everyday expertise: cognitive demands in diabetes self-management. *Human Factors*, 50(1):112–120, 2008.
- [74] Gordana Marunic and Vladimir Glazar. Spatial ability through engineering graphics education. *International Journal of Technology and Design Education*, 23(3):703–715, 2013.
- [75] Richard E Mayer, Jennifer L Dyck, and William Vilberg. Learning to program and learning to think: what’s the connection? *Communications of the ACM*, 29(7):605–610, 1986.
- [76] Renee McCauley, Brian Hanks, Sue Fitzgerald, and Laurie Murphy. Recursion vs. iteration: An empirical study of comprehension revisited. In *Proceedings of the 46th ACM technical symposium on computer science education*, pages 350–355. ACM, 2015.
- [77] L. McDermott. Research on conceptual understanding in mechanics. *Physics Today*, 37(7):24–32, 1984.
- [78] Mark G McGee. Human spatial abilities: Psychometric studies and environmental, genetic, hormonal, and neurological influences. *Psychological bulletin*, 86(5):889, 1979.
- [79] Pedro Moraes and Leopoldo Teixeira. Willow: A tool for interactive programming visualization to help in the data structures and algorithms teaching-learning process. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 553–558, 2019.
- [80] Allen Newell, Herbert Alexander Simon, et al. *Human problem solving*, volume 104. Prentice-Hall Englewood Cliffs, NJ, 1972.

- [81] Douglas P Newton. Causal situations in science: a model for supporting understanding. *Learning and Instruction*, 6(3):201–217, 1996.
- [82] Nel Noddings. *Philosophy of education*. Routledge, 2018.
- [83] Donald A Norman. Some observations on mental models. In *Mental models*. Psychology Press, 1983.
- [84] Richard M Onyancha, Matthew Derov, and Brad L Kinsey. Improvements in spatial ability as a result of targeted training and computer-aided design software use: Analyses of object geometries and rotation types. *Journal of Engineering Education*, 98(2):157–167, 2009.
- [85] Wolfgang Paul and Jan 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*, pages 29–34. ACM, 2013.
- [86] Martinha Piteira and Carlos Costa. Computer programming and novice programmers. In *Proceedings of the Workshop on Information Systems and Design of Communication*, pages 51–53, 2012.
- [87] Martinha Piteira and Carlos Costa. Learning computer programming: study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication*, pages 75–80, 2013.
- [88] Leo Porter, Saturnino Garcia, Hung-Wei Tseng, and Daniel Zingaro. Evaluating student understanding of core concepts in computer architecture. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 279–284. ACM, 2013.
- [89] Leo Porter, Daniel Zingaro, Cynthia Lee, Cynthia Taylor, Kevin C Webb, and Michael Clancy. Developing course-level learning goals for basic data structures in cs2. In *Proceedings of the 49th ACM technical symposium on Computer Science Education*, pages 858–863. ACM, 2018.
- [90] Leo Porter, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Kevin C Webb, Cynthia Lee, and Michael Clancy. Bdsi: A validated concept inventory for basic data structures. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 111–119, 2019.
- [91] N Praetorius and KD Duncan. Verbal reports: a problem in research design. In *Tasks, errors, and mental models*, pages 293–314. 1988.

- [92] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1, 2017.
- [93] Sarika Rajeev and Sharad Sharma. Motivational game-theme based instructional module for teaching binary tree and linked list. In *Proceedings of 28th International Conference*, volume 64, pages 31–40, 2019.
- [94] Jens Rasmussen, Annelise Mark Pejtersen, and Len P Goodstein. Cognitive systems engineering. 1994.
- [95] Robert Ravenscroft. An html5 browser application for modeling and teaching linked lists. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 1106–1106, 2018.
- [96] Robert A Ravenscroft Jr. Dynamic data structures, a web based tool for teaching linked lists and binary trees. *Journal of Computing Sciences in Colleges*, 33(6):97–106, 2018.
- [97] Anthony Robins, Ken Sutton, Denise Tolhurst, and Jodi Tutty. Programmed to succeed?: A multi-national, multi-institutional study of introductory programming courses.
- [98] K Rochford. Spatial learning disabilities and underachievement among university anatomy students. *Medical education*, 19(1):13–26, 1985.
- [99] Timothy J. Rolfe. Classroom exercise demonstrating linked list operations. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '06, page 83–84, New York, NY, USA, 2006. Association for Computing Machinery.
- [100] Mary A Sadowski and SA Sorby. Engineering graphics concepts: A delphi study. In *ASEE Annual Conference Proceedings*, 2015.
- [101] Beth Simon, Mike Clancy, Robert McCartney, Briana Morrison, Brad Richards, and Kate Sanders. Making sense of data structures exams. In *Proceedings of the Sixth international workshop on Computing education research*, pages 97–106, 2010.
- [102] Melinda Y Small and Mary E Morton. Research in college science teaching: Spatial visualization training improves performance in organic chemistry. *Journal of College Science Teaching*, 13(1):41–43, 1983.
- [103] 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)*, pages F3G–1–F3G–3, Oct 2010.
- [104] Amber Solomon. The role of spatial representations in cs teaching and cs learning. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 237–238. IEEE, 2019.
- [105] Sheryl Sorby, Beth Casey, Norma Veurink, and Alana Dulaney. The role of spatial training in improving spatial and calculus performance in engineering students. *Learning and Individual Differences*, 26:20–29, 2013.
- [106] Sheryl A Sorby. Educational research in developing 3-d spatial skills for engineering students. *International Journal of Science Education*, 31(3):459–480, 2009.
- [107] K Suzuki. Improvement of spatial ability through descriptive geometry education. *Journal of the Graphic Science of Japan*, 49:21–28, 1990.
- [108] Cynthia Taylor, Daniel Zingaro, Leo Porter, Kevin C Webb, Cynthia Bailey Lee, and Michael Clancy. Computer science concept inventories: past and future. *Computer Science Education*, 24(4):253–276, 2014.
- [109] Tammy VanDeGrift. Compare and contrast in data structures. *J. Comput. Sci. Coll.*, 34(1):195–201, oct 2018.
- [110] Kevin C Webb and Cynthia Taylor. Developing a pre-and post-course concept inventory to gauge operating systems learning. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 103–108. ACM, 2014.
- [111] Kevin C Webb, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Cynthia Lee, Michael Clancy, and Leo Porter. Student performance on the bdsi for basic data structures. *ACM Transactions on Computing Education (TOCE)*, 22(1):1–34, 2021.
- [112] Noreen M Webb. Microcomputer learning in small groups: Cognitive requirements and group processes. *Journal of Educational Psychology*, 76(6):1076, 1984.
- [113] SJ Westerman and T Cribbin. Navigating virtual information spaces: Individual differences in cognitive maps. *Proceedings of the UK Virtual Reality Special Interest Group*, 1999.
- [114] Michael D Williams, James D Hollan, and Albert L Stevens. Human reasoning about a simple physical system. In *Mental models*, pages 139–162. Psychology Press, 2014.

- [115] Chen-Chih Wu, Greg C Lee, and Janet Mei-Chuen Lin. Visualizing programming in recursion and linked lists. In *Proceedings of the 3rd Australasian conference on Computer science education*, pages 180–186, 1998.
- [116] Lucy Yeomans, Steffen Zschaler, and Kelly Coate. Transformative and troublesome? students’ and professional programmers’ perspectives on difficult concepts in programming. *ACM Transactions on Computing Education (TOCE)*, 19(3):1–27, 2019.
- [117] Shamama Zehra, Aishwarya Ramanathan, Larry Yueli Zhang, and Daniel Zingaro. Student misconceptions of dynamic programming. In *Proceedings of the 49th ACM technical symposium on Computer Science Education*, pages 556–561. ACM, 2018.
- [118] Philip David Zelazo. *The Oxford Handbook of Developmental Psychology, Vol. 1: Body and Mind*, volume 1. Oxford University Press, 2013.
- [119] Jinghua Zhang, Mustafa Atay, Elvira R Caldwell, and Elva J Jones. Reinforcing student understanding of linked list operations in a game. In *Frontiers in Education Conference (FIE), 2015 IEEE*, pages 1–7. IEEE, 2015.
- [120] Daniel Zingaro, Cynthia Taylor, Leo Porter, Michael Clancy, Cynthia Lee, Soohyun Nam Liao, and Kevin C Webb. Identifying student difficulties with basic data structures. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 169–177. ACM, 2018.
- [121] Rolf A Zwaan. Situation models, mental simulations, and abstract concepts in discourse comprehension. *Psychonomic bulletin & review*, 23(4):1028–1034, 2016.

APPENDICES

Appendix A: Survey Materials

CS Concept Inventory Consent

128

WHAT IS THE PURPOSE OF THIS FORM?

This form contains information you will need to help you decide whether to be in this research study or not. Please read the form carefully and ask the study team member(s) questions about anything that is not clear.

WHY IS THIS RESEARCH STUDY BEING DONE?

This project aims at understanding how students reason about pieces of and operations on linked list and the connections this has to visual-spatial reasoning. This research seeks to uncover students mental models and misunderstandings about linked list to develop a concept inventory to improve learning. Some findings of the study will be used for a student's thesis/dissertation. The study team members include Jennifer Parham-Mocello and Eman Almadhoun.

WHY AM I BEING INVITED TO TAKE PART IN THIS STUDY?

You are being asked to take part in this study because you are a student in CS 261.

What will happen if I take part in this research study?

You will participate in a 15-20 minute survey on linked lists, and depending on your responses to the survey, you may be recruited to participate in a semi-structured interview for \$15/hour. At this time, we are asking your permission to use the survey data for research purposes only.

Study duration: The survey will take no longer than 20 minutes.

WHAT ARE THE RISKS AND POSSIBLE DISCOMFORTS OF THIS STUDY?

The security and confidentiality of survey information collected cannot be guaranteed. Information collected online or sent by email can be intercepted, corrupted, lost, destroyed, arrive late or incomplete, or contain viruses. The research team will do its best to keep data secure and confidential.

WHAT ARE THE BENEFITS OF THIS STUDY?

This study is to advance our understanding of how students reason about linked lists and improve learning through qualitative research.

WHO WILL SEE THE INFORMATION I GIVE?

The information that you give us will only be used for this study. We will not share information about you with others or use it in future studies without your consent. There is still a chance that someone could figure out that the information is about you.

The information you provide during this research study will be kept confidential to the extent permitted by law. Research records will be stored securely and only researchers will have access to the records. Federal regulatory agencies and the Oregon State University Institutional Review Board (a committee that reviews and approves research studies) may inspect and copy records pertaining to this research. Some of these records could contain information that personally identifies you. If the results of this project are published your identity will not be made public, unless permission is given to share audio publicly.

What other choices do I have if I do not take part in this study?

If you decide to participate, you are free to withdraw at any time without penalty. If you choose to withdraw from this project before it ends, the researchers may keep information collected about you and this information may be included in study reports. Your decision to take part or not take part in this study will not affect your relationship with your instructor, researchers, or the University.

WHO DO I CONTACT IF I HAVE QUESTIONS?

If you have any questions about this research project, please contact: Jennifer Parham-Mocello at (541) 737-8895 or parhammj@oregonstate.edu. If you have questions about your rights or welfare as a participant, please contact the Oregon State University Institutional Review Board (IRB) Office, at (541) 737-8008 or by email at IRB@oregonstate.edu.

WHAT DOES MY AGREEMENT ON THIS CONSENT FORM MEAN?

Your agreement indicates that this study has been explained to you, that your questions have been answered, and that you agree to take part in this study.

-
- I agree to participate in this study.
- I do not agree to participate in this study.

Demographic/Background

What is your GPA?

129

To which gender identity do you most identify?

- Male
- Female
- Transgender Male
- Transgender Female
- Non-Conforming
- Not Listed
-
- Prefer not to answer

What is your age?

- 18 - 24
- 25 - 33
- 34 - 42
- 43 or more

Choose one or more races that you consider yourself to be:

- White
- Black or African American
- American Indian or Alaska Native
- Asian
- Native Hawaiian or Pacific Islander
- Other

What is your major?

- Computer Science
- Electrical Engineering
- Other:
-

Have you ever changed your program/major?

Yes: what was your previous program/major?

130

No

Are you studying in a double degree program?

Yes

No

Did you take CS 162 - Introduction to Computer Science II class at Oregon State University (OSU)?

Yes

No

Other

Which quarter did you take CS 162 at OSU?

Who was your instructor in CS 162 at OSU?

Did you have any knowledge of linked lists prior to CS 261?

Yes

No

Did you have any difficulties while learning linked lists in this class?

Yes

No

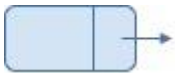
What difficulties did you have while learning linked lists? and why?

131

Linked Lists

In the following questions, \equiv represents NULL

Q1. What does the following drawing represent? Please write why you choose this answer?



Node Pointer

Node

BC the 2 parts: data and pointer

Data

NULL Pointer

Other

Q2. What does the following drawing represent? Please write why you choose this answer?



Node Pointer

Node

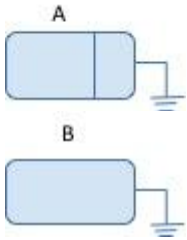
Data

NULL Pointer

Other

132

Q3. Which drawing is an empty list? Please write why you choose this answer?



A

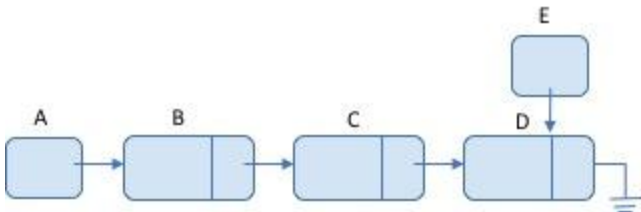
B

All of the above

None of the above

Other

Q4. What type of list does the following drawing represent? (Questions 5 - 23 are referred to this drawing)



Singly Linked List

Doubly Linked List

Singly Circular Linked List

Doubly Circular Linked List

Other

Q5. From Q4, what is the data type of A? Please write why you choose this answer? ¹³³

Node Pointer

This is the head which stores the address of the first node of the list

Node

Head

Tail

Other

Q6. From Q4, what is the benefit from including A at the beginning of the list?

To indicate the start/beginning of the linked list

Q7. From Q4, what is B? Please write why you choose this answer?

Node Pointer

Node

BC the 2 parts: data and pointer

Head

Tail

Other

Q8. From Q4, what is the data type of E? Please write why you choose this answer?

Node Pointer


Node

-
- Head
- Tail
- Other

134

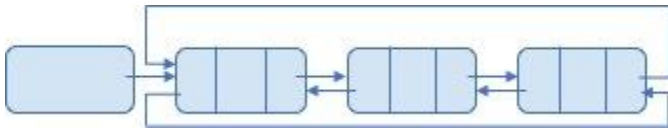
Q9. From Q4, what is the benefit from including E at the end of the list?

To allow access the end of the list in constant time $O(1)$ and ease add a new node to the end of a linked list.

Q10. From Q4, what is the benefit from including this element () at the end of the list?

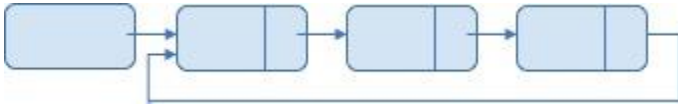
To indicate the end of a linked list

Q11. What type of list does the following drawing represent?



- Singly Linked List
- Doubly Linked List
- Singly Circular Linked List
- Doubly Circular Linked List
- Other

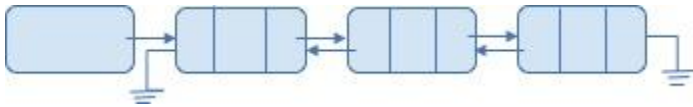
Q12. What type of list does the following drawing represent?



135

- Singly Linked List
 Doubly Linked List
 Singly Circular Linked List
 Doubly Circular Linked List
 Other

Q13. What type of list does the following drawing represent?



- Singly Linked List
 Doubly Linked List
 Singly Circular Linked List
 Doubly Circular Linked List
 Other

Q14. In a linked list, how are the nodes stored/arranged in memory?

- Contiguously
 Non-Contiguously
 Other

Q15. What type of data can be stored in a node of a linked list? Please write why you choose this answer? (You can Choose more than one answer)

- Integer

- Character
- Memory Address
- Null
- All of the above
- I do not know

136

Q16. A node in a singly linked list has two parts, what are they? (You can choose more than one answer)

- Data
- Pointer
- Other

Q17. From Q16, please explain what each part stores?

Data part: stores any value, such that int, char, double, memory address, object...
Pointer part: stores memory address of the next node point to or Null

Q18. In a singly-linked list, how many pointers are in a node?

- 0
- 1
- 2
- 3
- 4

Q19. In a doubly-linked list, how many pointers are in a node?

- 0
- 1
-

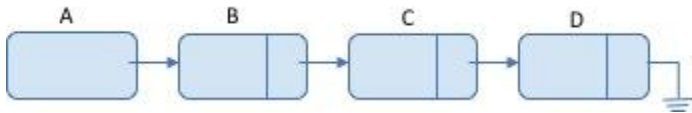
2

3

4

137

Q20. Suppose you have a singly linked list, you want to delete B from the list, please select the best answer to do so:



Assign A to point to C and then free B

Free B

Assign a temporary node pointer to point to B, assign A to point to C, and then free B

Free A

Q21. From Q20, suppose you want to delete the entire linked list, please select the best answer:

Free A

While A is not pointing to NULL, free what A points to and assign A to the next node.

While A is not pointing to NULL, assign a temporary node pointer to point to the same place as A, assign A to point to the next node in the list, and free the node pointed to by the temporary node pointer.

Both a and b are correct

None of the above

Pointers

Q22. In the C language, what is the benefit to using the free() function in your code?

To de-allocate the memory allocated by the functions malloc ()

To deallocate the memory space assigned by the functions malloc () and to be reused by a subsequent malloc()

138

Q23. Suppose you have the following:

```
char *p = (char) malloc(sizeof(char) *10);  
char *q = (char) malloc(sizeof(char) *10);
```

What does the part char *p mean?

- Create a character variable
- Create a pointer to a character
- Create an array of characters
- Other

Q24. What does p = q mean? Please write why you choose this answer?

- Make q point to the same place as p
-
- Make p point to the same place as q
-
- All of the above
-
- Other

Q25. Is the following operation legal or not? Please write why you choose this answer?

```
*p = *q;
```

- Legal

Same data type

- Illegal

Q26. Is the following operation legal or not? Please write why you choose this answer?

`p = *q;`

139

Legal

Illegal

Different data type, p is an address while *q is a character

Q27. Is the following operation legal or not? Please write why you choose this answer?

`*p = q;`

Legal

Illegal

Different data type, *p is a character while q is an address

Q28. What is the output after the execution of the following code:

```
double *p;
double pi, e;
p = &e;
*p = 2.71828
p = &pi;
*p = 3.14159;
printf( "%p %g %g %g\n", p, *p, pi, e);
```

eebff768 2.71828 3.14159 3.14159

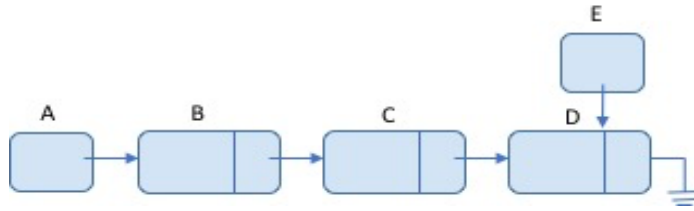
eebff768 3.14159 3.14159 2.71828

3.14159 3.14159 2.71828 eebff768

3.14159 3.14159 eebff768 2.71828

Open-Opened Survey Questions, Answer and Rubric

140



Q6. From Q4, what is the benefit from including A (head) at the beginning of the list?

Answer:

To indicate the start/ beginning of the linked list

Rubric:

The question rewards one point:

1 pt (explicitly states indicate the start/ beginning/front of the list)

0.5 pt (knows it points to a linked list, but not explicitly state start/ beginning of the list)

0 pt (doesn't mention indicate start/beginning/point/front)

ID	Answer	Grade	Reason for deducting points
P1			
P2			
...			
P40			

Q9. From Q4, what is the benefit from including E (tail) at the end of the list?

Answer:

To allow access the end of the list in constant time and ease add a new node to the end of a linked list.

Rubric:

The question rewards two points one point for each part:

1 pt (explicitly states constant and linear)

0.5 pt (knows it is faster, but not explicitly state constant/linear)

0 pt (doesn't mention complexity/time/faster)

1 pt (explicitly states only add and not delete)

0.5 pt (combines all operations on the end (add and delete))
0 pt (nothing about operations on the end)

141

ID	Answer	Grade	Reason for deducting points
P1			
P2			
...			
P40			

Q10. From Q4, what is the benefit from including this element () (NULL) at the end of the list?

Answer:

To indicate the end of linked list

Rubric:

The question rewards one point:

1 pt (explicitly states the indication of end of the list)

0.5 pt (knows it points to NULL, but not explicitly state indication of end)

0 pt (doesn't mention the indication of the end of the list)

ID	Answer	Grade	Reason for deducting points
P1			
P2			
...			
P40			

Q17. From Q16, please explain what each part stores (in a node)?

Answer:

Data part: stores any type of data or value, such that int, char, double, memory address, object...

...

Pointer part: stores memory address of the next node point to or NULL

Rubric:

The question rewards two points, one point for each part:

1 pt (explicitly states there is a data part and it stores any value/info/data)

.5 pt (states there is a data part, but doesn't explicit state that it stores any value/info/data)

0 pt (doesn't explicitly state that there is a data part or that it stores any value/info/data)

- 1 pt (explicitly states there is a pointer part and it stores memory address of the next node)
- 0.5 pt (mentions there is a pointer part, but doesn't state that it stores a memory address to next node (stating that it stores a pointer/reference to another node isn't the same as the address because it is a pointer/reference, and it holds/stores an address))
- 0 pt (doesn't mention pointer part stores memory address/pointer of the next node)

ID	Answer	Grade	Reason for deducting points
P1			
P2			
...			
P40			

Q22. In the C language, what is the benefit to using the free() function in your code?

Answer:

To deallocate the memory space assigned by the function malloc() to be reused by a subsequent malloc()

Rubric:

The question rewards 1 point:

- 1 pt (explicitly states deallocate/free/release memory space to be reused)
- 0.75 pt (does not explicitly state that memory can be reused, but they state that deallocate/free/release memory space to prevent memory leaks)
- 0.5 pt (does not explicitly state that deallocate/free/release memory space, but they state that memory can be reused or prevent memory leak)
- .25 pt (does not explicitly state that memory can be reused or prevent memory leak, but they state that it deallocate/free/release the memory space)
- 0 pt (doesn't mention that memory can be reused or deallocate/free/release or /prevent memory leaks)

ID	Answer	Grade	Reason for deducting points
P1			
P2			
...			
P40			

Appendix B: Verbal Questions

Interview Verbal Questions, Answer and Rubric

144

Q1. Describe a node in a linked list.

A node in a linked list is a container or an object that stores data of any type and pointer (or memory address) of the next node or Null to indicate the end (or last node) of the linked list.

Total Points: /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	The data can be any type			
2	A node stores a pointer or memory address			
2	The pointer points to a different/next node			
2	The pointer points to Null			
2	Null pointer indicates the end (or last node) of the linked list			
Other Interesting Observations/Comments Student Responses/Answers:				

Q2. Describe a node pointer in a linked list.

A node pointer points (refers) to the next node in a list by storing/using the memory address of the next node or Null to indicate the end (or last node). A node pointer also can come by itself as a head or tail. In a doubly linked list, there is an extra pointer points to the previous node.

Total Points: /10 /2	Point Divvy: Gain 1 point 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	The node pointer points (refers) to the next node in the list			
2	The node pointer holds the memory address of another node (explicitly mentions the term “ memory address ”)			
2	The value held by the node’s pointer can be Null			

2	Null pointer indicates the last (ending) node of the linked list				145
2	A node pointer can come by itself as a head or a tail				
2	In a doubly linked list, there is an extra pointer points to the previous node.				
Other Interesting Observations/Comments Student Responses/Answers:					

Q3. Describe the difference in a head pointer vs. a tail pointer in a linked list.

A head pointer:

- is a node pointer referring (pointing) to the first node in a linked list
- contains the memory address of the first node in a linked list or NULL for an empty list

A tail pointer

- is a node pointer that refers (points) to the last node in the list.
- contains the memory address of the last node in a linked list or NULL for an empty list.

Total Points: /12	Point Divvy: Gain 1 point 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
Head Pointer:				
2	points to or references the first node in a linked list			
2	stores the memory address of the first node			
2	can be assigned Null to represent an empty list			
Tail Pointer:				
2	points to or references the last node in a linked list			
2	stores the memory address of the last node			

2	can be assigned Null to represent an empty list			146
Other Interesting Observations/Comments Student Responses/Answers:				

- 3.1 What are the pros and cons of having a head and tail pointer?

Pros:

A head pointer:

- provides ability to access the beginning of a linked list in memory
- allows inserting or deleting a node at beginning in constant time $O(1)$.

A tail pointer:

- provides ability to access the end of a linked list in memory in constant time $O(1)$.
- allows inserting at the end of the list in a constant time $O(1)$.

Cons:

A tail pointer:

- more memory storage (4 bytes on 32-bit CPU / 8 bytes on 64-bit CPU) to store a reference to the end of the list
- deleting a node at the end with the tail in singly linked list still takes linear time $O(n)$

Having only head pointer (no tail pointer): **There are no cons for having a head pointer since it requires to access the list**

- Finding the end of the list is not constant time
- Add a new node to the end is linear time $O(n)$ (not constant time)

Total Points: /24	Point Divvy: Gain 1 point 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
Pros of Head Pointer:				
2	provides the ability to access the beginning of a linked list in memory			
2	allows for inserting or deleting a node at the beginning of a linked list			
2	allows for insertion and deletion operations to take constant time $O(1)$			

	Note: give a point for above cell when this is present				147
Pros of Tail Pointer:					
2	provides the ability to access the end of a linked list				
2	allows for access to the end of the linked list to take constant time $O(1)$ Note: give a point for above cell when this is present				
2	allows for inserting a node at the end of a linked list				
2	allows for inserting a node at the end of a linked list to take constant time $O(1)$ Note: give a point for above cell when this is present				
Cons of Tail Pointer:					
2	The tail pointer has a con of more memory storage				
2	more memory storage to store a reference to the end of the list (the last node)				
2	provides deleting a node at the end of the singly linked list in linear time $O(n)$				
Cons of having only Head Pointer (not having a tail pointer):					
2	Finding the end of the list is not constant time				
2	Add a new node to the end is linear time (not constant time)				
Other Interesting Observations/Comments Student Responses/Answers:					

- 3.2 Do you create the head or tail of a linked list as a node or node pointer?
 - Why?
 - Node pointer because less memory storage for avoiding unnecessary data storage.

Total Points: /4	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	Use node pointer to create head and tail.			
2	Reasoning: Storing only the memory address will take up less space than a full node would			
Other Interesting Observations/Comments Student Responses/Answers:				

Q4. Describe a singly linked list? Draw a picture if needed.

A singly linked list is a linear structure of nodes that are linked together in one direction (unidirectional) from beginning/front to last/end with a pointer that refers (points) to the next node. Each node stores two parts the data and address of the next node with the exception of the last node that points to NULL to indicate the end of the list, and there is a node pointer (or pointer to a node) called the head that points to the first node in the list or NULL for an empty list.

Total Points: /18	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 singly linked list is a linear structure			
2	A singly linked list consists of nodes linked together in a single direction (unidirectional)			
2	A singly linked list's nodes are linked together with a pointer that points (or refers) to the next node			
2	Each node stores two parts			
2	Each linked list node has a data value			

2	Each linked list node has a memory address of the next node				149
2	The last node points to or holds a memory address of NULL to indicates the end of the list				
2	A node pointer called the head points to the first node in a linked list				
2	A head can point to Null to indicate an empty list				
Other Interesting Observations/Comments Student Responses/Answers:					

Q5. Describe a doubly linked list? Draw a picture if needed.

A doubly linked list is a linear structure of nodes that are linked together in two directions (bidirectional) from beginning/front to last/end and last/end to beginning/front with two pointers that refer (point) to the next node and the previous node. Each node stores three parts: the data, the address of the next node and the address of the previous node with the exception of the first and the last node that points to NULL to indicate the end of the list (in the backward or forward direction), and there is a node pointer (or pointer to a node) called the head that points to the first node in the list or NULL for an empty list.

Total Points: /22	Point Divvy: Can still gain points if they are pointing out differences between their description of a singly linked list 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 doubly linked list is a linear structure			
2	A doubly linked list consists of nodes linked together in both directions (bidirectional)			
2	A doubly linked lists' nodes are linked together with pointers that points (refers) to the next node and the previous node			
2	Each node stores three parts			
2	Each linked list's node has a data value			

2	Each linked list node has a memory address of the next node				150
2	Each linked list node has a memory address of the previous node				
2	The first node point to or holds a memory address of NULL to indicates the end of the list				
2	The last node point to or holds a memory address of NULL to indicates the end of the list				
2	A node pointer called the head points to the first node in a linked list				
2	A head can point to Null to indicate an empty list				
Other Interesting Observations/Comments Student Responses/Answers:					

Q6. Describe a singly circular linked list? Draw a picture if needed.

A singly circular linked list is a circular a structure of nodes that are linked together in one direction (unidirectional) from beginning/front to last/end with a pointer that refers (points) to the next node. Each node stores two parts the data and address of the next node with the exception of the last node that points back to the first node in the list, which makes it circular, and there is a node pointer (or pointer to a node) called the head that points to the first node in the list or NULL for an empty list. The end of the list is known by checking to see if the address in the node pointer section of a node is the same address in the head node pointer.

Total Points: /18	Point Divvy: Can still gain points if they are pointing out differences between their description of other linked lists 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 singly circular linked list is a circular (circle, linked start to end, looped, etc.) structure			
2	A singly circular consists of nodes linked together in a single direction (unidirectional)			

2	A singly circular linked list's nodes are linked with a pointer that points (or refers) to the next node			151
2	Each node stores two parts			
2	Each linked list node has a data value			
2	Each linked list node has a memory address of the next node			
2	The last node points back to or holds a memory address of the first node			
2	A node pointer called the head points to the first node in a linked list			
2	A head can point to Null to indicate an empty list			
Other Interesting Observations/Comments Student Responses/Answers:				

Q7. Describe a doubly circular linked list? Draw a picture if needed.

A doubly circular linked list is a circular structure of nodes that are linked together in two directions (bidirectional) from beginning/front to last/end and last/end to beginning/front with two pointers that refer (point) to the next node and the previous node. Each node stores three parts: the data, the address of the next node and the address of the previous node with the exception of the last node that points back to the first node in the list and the first node that points to the last node in the list, which makes it doubly circular, and there is a node pointer (or pointer to a node) called the head that points to the first node in the list or NULL for an empty list.

Total Points: /22	Point Divvy: Can still gain points if they are pointing out differences between their description of other linked lists 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 doubly circular linked list is a circular (circle, linked start to end, looped, etc.) structure			
2	A doubly circular list consists of nodes linked together in both (bidirectional) directions			

2	A doubly circular linked list's nodes are linked together with pointers that points (or refers) to the next node and the previous node			152
2	Each node stores three parts			
2	Each linked list node has a data value			
2	Each linked list node has a memory address of the next node			
2	Each linked list node has a memory address of the previous node			
2	The last node points back to or has holds a memory address of the first node			
2	The first node points to or holds a memory address of the last node			
2	A node pointer called the head points to the first node in a linked list			
2	A head can point to Null to indicate an empty list			
Other Interesting Observations/Comments Student Responses/Answers:				

Q8. What are the advantages and disadvantages of using a linked list over a dynamic array?

Advantages:

- **Linked list is much more efficient** to insert at the front and back and delete in the front with a singly linked list [time complexity: $O(1)$], and delete from the back is constant time for a doubly linked list with tail node pointer vs dynamic array takes linear time always for insertion and deletion an item in the array
- **Linked list takes** half the time of an array $O(n/2)$ (find the middle and add/delete in constant time) in deletion/Insertion at the middle vs dynamic array takes linear time (need to copy over all elements to new array).
- **Linked list does not need to be copied over to a different place** in memory when its size/capacity is changed vs Dynamic Array has to be copied over to a different place in memory when its size/capacity is changed (deletion or

insertion causes array to be reconstructed) as the whole array needs to be copied to a different place in memory.

- **Linked list does not require specifying the size/capacity** of the list in advance unlike a dynamic array

Disadvantages:

- **Accessing items in a linked list are less efficient $O(n)$** than direct access to specific element location (e.g. 3rd item in the list) using address arithmetic in dynamic arrays.
- **Nodes are stored in non-contiguous** locations in memory, which can lead to cache misses when accessing data in the list vs items in dynamic arrays are stored in contiguous order which can be accessed through address arithmetic and lead to more cache hits for items right beside each other.
- **Requires more memory storage due to storing 1-2 memory addresses** in addition to the actual data vs dynamic array requires less memory storage due to storing just the actual data.
- **Same search time** for item of a specific value as linked list (linear time $O(n)$).

Total Points: /32	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
Advantages of Linked List Over Dynamic Array:				
2	A linked list is more efficient/faster/better time complexity in some cases OR A dynamic array is less efficient/slower/worst time complexity in some cases			
2	Linked list takes constant time ($O(1)$) for inserting a new node at the head OR Dynamic array takes linear time ($O(n)$) for inserting of an item at the beginning			
2	Linked list takes constant time ($O(1)$) for inserting a new node at the end with a tail OR			

	Dynamic array takes linear time ($O(n)$) for inserting a new item at the end when the capacity is exceeded			154
2	Linked list takes constant time ($O(1)$) for deleting a node from the front OR Dynamic array takes linear time ($O(n)$) for deleting an item from the front			
2	Linked list takes constant time ($O(1)$) in deleting a node from the back of a doubly linked list with a tail AND/OR Dynamic array takes linear time ($O(n)$) in deleting an item from the back			
2	Linked list takes constant time inserting a new node in the middle in addition of searching time ($O(n/2)$ time) OR Dynamic array takes linear time ($O(n)$) in inserting an item in the middle			
2	Linked list takes constant time ($O(1)$) in deleting a node in the middle in addition of searching time ($O(n/2)$ time) OR Dynamic array takes linear time ($O(n)$) in deleting an item in the middle			
2	Linked list does not need to be copied over to a different place in memory when its size/capacity is changed OR Dynamic Array has to be copied over to a different place in memory when its size/capacity is changed (deletion or insertion causes array to be reconstructed)			
2	Linked list does not require specifying the			

	size/capacity of the list in advance OR Dynamic array requires specifying the size/capacity of the array in advance			155
Disadvantages of Linked List Over Dynamic Array:				
2	A linked list is less efficient/slower/worst time complexity in some cases OR A dynamic array is more efficient/faster/better time complexity in some cases			
2	Accessing items in a linked list is less efficient (or $O(n)$) than an array OR A dynamic array is more efficient (or $O(1)$) for accessing specific element locations (direct access)			
2	Nodes in linked list are stored in non-continuous locations in memory OR A dynamic array has contiguous order to its elements			
2	The non-continuous locations of the nodes can lead to cache misses when accessing data in the list OR A dynamic array's contiguous elements can allow for more cache hits for items right beside each other			
2	Linked list requires more memory storage OR A dynamic array requires less storage			
2	Linked list requires storing 1-2 memory address(es) in addition to the data OR			

	A dynamic array requires less storage for storing just the actual data.				156
2	Linked list and dynamic array have the same search time for item of a specific value (linear time $O(n)$)				
Other Interesting Observations/Comments Student Responses/Answers:					

Q9. What kind of data can be stored in a linked list?

The data part in a linked list can be any type of data e.g. integer, char, double...

Total Points: /2	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	The data part in a linked list can be any type of data			
Other Interesting Observations/Comments Student Responses/Answers:				

Q10. What possible operations can be performed on a linked list?

Insert node (at the beginning, add at the end, add at specific location), delete node (at the beginning, add at the end, add at specific location), find, clear, swap, create an empty list, check empty, find the length of the list, reverse, and print...

Total Points: /6	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	Insertion (at the beginning, add at the end, add at specific location)			
2	Deletion (at the beginning, add at the end, add at specific location)			
2	Other operations			
Other Interesting Observations/Comments Student Responses/Answers:				

Q11. What does “dereferencing” a pointer mean?

Dereferencing a pointer means to go to the memory address that is stored in the pointer¹⁵⁷ to storing or fetching the information stored at that location.

Total Points: /6	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	Mentions go to the memory address that is stored in the pointer			
2	To store information at that location (memory address)			
2	To fetch the information stored at that location (memory address)			
Other Interesting Observations/Comments Student Responses/Answers:				

- How is a pointer “dereferenced”? (OR which operation can be used to do so).
Using the * operator

Total Points: /2	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	* (an asterisk)			
Other Interesting Observations/Comments Student Responses/Answers:				

Q12. What does this line of the code mean/do?

struct node* new_node = (struct node*) malloc(sizeof(struct node));

This line means during runtime, dynamically allocate memory on the heap the size of a node structure and type cast this address to a pointer to a node structure, then assign this address to new_node, which is a pointer to a node structure.

Total Points: /8	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	Allocates memory during runtime on the heap			
2	Size of a node structure			
2	Type casts the address of allocated memory returns by malloc to a node structure pointer			
2	Assigns this address to a pointer to a node structure (node pointer) called a new node			
Other Interesting Observations/Comments Student Responses/Answers:				

Q13. If you want to swap two numbers, what do you think is better, swapping the node or swapping the data?

- Why do you think that?

Swapping the Nodes is better. It should be swapped by changing the pointers. Swapping data of nodes is expensive when data is very large.

Total Points: /6	Point Divvy: Gain points for mention of each of these topics:	Partially Correct)	Correct	Comment: Reason for getting/not getting full marks or anything else interesting about their conceptual understanding
2	States that swapping nodes is better.			
2	Should be swapped by changing pointers (memory addresses).			
2	States that swapping data is expensive when data is large.			
Other Interesting Observations/Comments Student Responses/Answers:				

Appendix C: Coding Questions

Semi-Structured Interview Questions

Q1. Write code/pseudocode to create an empty linked list.

```

struct Node {
    int data;
    struct Node *next;
};

struct Node *createEmptyList() {
    struct Node *head = NULL;

    return head;
}
    
```

Total Points: /16 /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	Node has data part				
3	Node has pointer part				
3	Creates a node pointer (creates head)				
3	Assigns node pointer (head) to NULL				
1	The empty list is successfully created				
3	Creates a function				
3	Returns head pointer				
Other Interesting Observations/Comments Student Responses/Answers:					

Q2. Write code/pseudocode to check if a linked list is empty.

161

```
bool checkEmpty(struct Node *head) {
    if (head == NULL)
        return true;
    else
        return false;
}
```

Total Points: /7 /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	Uses an if statement (or similar logic structure)				
3	Checks the head against NULL .				
1	The empty list is successfully checked				
3	Creates a function with a pointer to beginning of list				
3	Return True/False				
Other Interesting Observations/Comments Student Responses/Answers:					

Q3. Write code/pseudocode for inserting a new node at the beginning of a singly linked list.

//Q3. Function for inserting a new node at the beginning of a singly linked list

```
struct Node *insertBeginning(struct Node *head, int x){

    //dynamic allocating on the heap for a new node and storing
    the memory address for that location in a node pointer called
    new_node
    struct Node *new_node = (struct Node*) malloc(sizeof(struct
    Node));

    //store the x value (inserted by the user) inside the data
    portion in the node pointed by new_node pointer
    Order does not matter for this line and the following line
    new_node -> data = x;

    //make the new node pointer portion points to same as head
    points to by storing the memory address as the head
    new_node ->next = head;

    // return the address of the new head
    return new_node;
}
```

//Alternative solution for Q3. Function for inserting a new node at the beginning of a single linked list using double pointer to modify on the head pointer

```
void insertBeginningDoublePointer(struct Node **head_ref, int x){

    //dynamic allocating on the heap for a new node and storing
    the memory address for that location in a node pointer called
    new_node
    struct Node *new_node = (struct Node*) malloc(sizeof(struct
    Node));

    //store the x value (inserted by the user) inside the data
    portion in the node pointer by new_node pointer
    Order does not matter for this line and the following line
    new_node -> data = x;

    //make the new node pointer portion points to same as head
    points to by storing the memory address as the head
    new_node ->next = *head_ref;

    //update the head pointer to point the same node as new_node
    pointer points to (the new node is just added)
    *head_ref = new_node;
}
```

Total Points: /13 /9	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	Creates a new node				
3	Assign to pointer/reference				
3	Make new node point to the old/previous first node				
3	Update head to point to new node				
1	A new node is successfully added at the beginning				
3	Store/Get the data inside the new node (you can infer that they will store if they refer to getting or putting a value into the new node)				
3	Creates a function with a pointer to beginning of list OR Creates a function with pointer to the head pointer of list				
3	Return the address of the new node to update the head				

	<p>OR Updates the head inside the function, rather than returning the address to new node.</p>				164
<p>Other Interesting Observations/Comments Student Responses/Answers:</p>					

Q4. Write code/pseudocode for deleting a node at the beginning of a singly linked list.

//Q4. Function for deleting a node at the beginning of a singly linked list

```

struct Node *deleteBeginning(struct Node *head){

    //declare a node pointer (temp) and set it to point the same
    //as what the head points to by storing the memory address of
    //the first node. You must have a temp in this function,
    //otherwise you will access something after freeing it, and
    //that should be a grade deduction!
    struct Node *temp = head;

    //If the list is not empty
    if(head != NULL) {
        //update the head to point to the second node by storing
        //the memory address that stored in head->next portion
        head = head->next;

        //delete temp (the memory allocation that set for the
        //first node)
        free(temp);
    }

    // return the address of the new head
    return head;
}

```

```
//Alternative solution for Q4. Function for deleting a node at
the beginning of a singly linked list using double pointer
```

```
void deleteBeginningDoublePointer(struct Node **head) {
```

165

```
//declare a node pointer (temp) and set it to point the same
as what the head points to by storing the memory address of
the first node. You must have a temp in this function,
otherwise you will access something after freeing it, and
that should be a grade deduction!
```

```
struct Node *temp = *head;
```

```
//If the list is not empty
```

```
if(*head != NULL) {
```

```
//update the head to point to the second node by storing
the memory address that stored in head->next portion
```

```
*head = (*head)->next;
```

```
//delete temp (the memory allocation that set for the
first node)
```

```
free(temp);
```

```
}
```

```
}
```

Total Points: /16 /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	Has a temp pointer .				
3	Set temp to point to what head points to (first node)				
3	Includes a check the make sure the list is not empty .				
3	Stores next node after head to indicate new beginning of the list.				
3	Remove original head node to prevent memory leak.				
1	The first node is successfully deleted				

3	Creates a function with a pointer to beginning of list OR Creates a function with a pointer to head pointer.				166
3	Return the address of the new node to update the head OR Correctly dereferences and updates the head pointer within the function				
Other Interesting Observations/Comments Student Responses/Answers:					

```

//Alternative solution for Q4. Function for deleting a node at
the beginning of a singly linked list
struct Node *deleteBeginning(struct Node *head){

    //declare a node pointer (temp) and set it to point the
    second node You must have a temp in this function, otherwise
    you will access something after freeing it, and that should
    be a grade deduction!
    struct Node *temp = head->next;

    //If the list is not empty
    if(head != NULL) {
        //delete the first node
        free(head);

        //update the head to point to the second node (store the
        memory address the same as temp
        head = temp;
    }

    // return the address of the new head
    return head;

}

```

```
//Alternative solution for Q4. Function for deleting a node at
the beginning of a singly linked list using double pointer
void deleteBeginningDoublePointer2(struct Node **head) {
```

167

```

//declare a node pointer (temp) and set it to point the
second node You must have a temp in this function, otherwise
you will access something after freeing it, and that should
be a grade deduction!
struct Node *temp = (*head)->next;
```

```

//If the list is not empty
if(*head != NULL) {
    //delete the first node
    free(*head);
```

```

//update the head to point to the second node (store the
memory address the same as temp
*head = temp;
```

```
}
```

```
}
```

Total Points: /16 /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	Has a temp pointer				
3	Set temp to point to head next (second node)				
3	Includes a check the make sure the list is not empty				
3	Remove original head node				
3	The new head is set to the original head's next node				
1	The first node is successfully deleted				
3	Creates a function with a pointer to beginning of list OR Creates a function with a pointer to head pointer.				

3	Return the address of the new node to update the head OR Correctly dereferences and updates the head pointer within the function				168
Other Interesting Observations/Comments Student Responses/Answers:					

Q5. Write code/pseudocode to find the length of the linked list.

//Q5. Function for finding the length of the linked list using while loop

```
int findLength(struct Node *head) {
```

```
    int count = 0;
```

```
    //declare a node pointer (temp) and set it to point the same as what the head points to by storing the memory address of the first node. The temp is not necessary, so we don't want to grade down for not having it. You could use head, since it is not changing the head outside the function.
```

```
    struct Node *temp = head;
```

```
    //traverse until temp reach the last node in the list by changing the location of temp pointer
```

```
    while(temp != NULL)
```

```
    {
```

```
        //add one each time temp visits a node
```

```
        count++;
```

```
        temp = temp->next;
```

```
    }
```

```
    //return the number of the node in the list
```

```
    return count;
```

```
}
```

//Alternative solution for Q5 function for finding the length of the linked list using for loop

```
int findLength_for(struct Node *head) {
```

```
    //declare a node pointer (temp) and set it to point the same as what the head points to by storing the memory address of the first node. The temp is not necessary, so we
```

don't want to grade down for not having it. You could use head, since it is not changing the head outside the function.

169

```
struct Node *temp = head;

    //traverse until temp reach the last node in the list by
    changing the location of temp pointer
    for(int count=0; temp != NULL; count++)
    {
        //add one each time temp visits a node
        temp = temp->next;
    }

    //return the number of the node in the list
    return count;
}

//Alternative solution for Q5 function for finding the length of the
linked list using double pointer
int findLengthDoublePointer(struct Node **head){

    int count = 0;

    //declare a node pointer (temp) and set it to point the same as
    what the head points to by storing the memory address of the first
    node
    struct Node *temp = *head;

    //traverse until temp reach the last node in the list by changing
    the location of temp pointer
    while(temp != NULL)
    {
        //add one each time temp visits a node
        count++;
        temp = temp->next;
    }

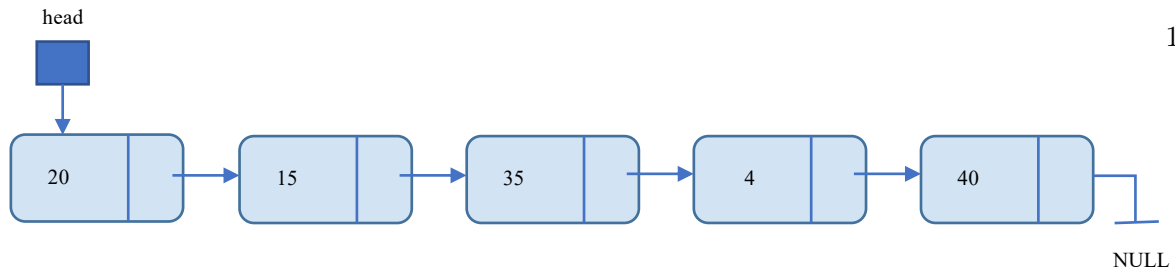
    //return the number of the node in the list
    return count;
}

//Alternative solution for Q5 function for finding the length of the
linked list using recursion
int findLengthRecursion(struct Node *head){

    if(head == NULL)
        return 0;
    else
        return 1+findLengthRecursion(head->next);
}
```


Total Points: /22 /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	Has a temp pointer				
3	Set temp to point to what head points to (first node)				
3	Includes a counter variable to keep track of length				
3	Set the counter to zero				
3	Includes a loop to go through the list				
3	Increment the counter by one				
3	Iterates from node to node in list				
1	The length of linked list is successfully counted				
3	Creates a function with a pointer to beginning of list OR Creates a function with pointer to the head pointer of list				
3	Returns/displays/give final count (return the number of nodes in the linked list)				
Other Interesting Observations/Comments Student Responses/Answers:					

Q6. Suppose we have a singly linked list as follows:



171

1) add item(6) to the end of the list.

```
//Q6_1. Function for inserting a new node that has value 6 at
the end of a singly linked list The student will not have a
return type if they are not thinking of the empty list.
void addLast(struct Node *head, int new_data)
{
    //dynamic allocating on the heap for a new node and
    storing the memory address for that location in a node pointer
    called new_node
    struct Node *new_node=(struct Node*)malloc(sizeof(struct
    Node));

    //store in the data portion of the new node the value that
    the user wants to insert
    new_node->data = new_data;

    //store in the pointer portion of the new node NULL value
    because this node will be the last node in the list. If
    student makes the new node next points to what the last node
    next in the list points to, they have to do that before assign
    the last node next to the new node
    new_node->next = NULL;

    //check if the list is empty. Students may not think about
    making this work for an empty linked list because the picture
    shows a list that isn't empty. They likely don't hard code
    the value 6, so they should make this function general. Deduct
    points if this is not checked.
    if (head == NULL) {
        // make head point to new node
        head = new_node;
        return;
    }
    else{
        //traverse until temp reach the last node in the list
        by changing the memory address stored in temp pointer
        while (head -> next != NULL)
            head = head ->next;
```

```

        //change the next of last node to point to the new node
        head ->next = new_node;
        return;
    }
}

//Alternative solution for inserting a new node that has value
6 at the end of a singly linked list using double pointers The
student will not have this alternative if they didn't think
about an empty list.
void addLastDoublePointer(struct Node **head, int new_data)
{
    //declare a node pointer (temp) and set it to point the
    same as what the head points to by storing the memory
    address of the first node. Student cannot use the head
    pointer to add the new node because the list will be
    updated only with the last 2 nodes.
    struct Node *temp = *head;

    //dynamic allocating on the heap for a new node and
    storing the memory address for that location in a node pointer
    called new_node
    struct Node *new_node = (struct Node*)malloc(sizeof(struct
Node));

    //store in the data portion of the new node the value that
    the user wants to insert
    new_node->data = new_data;

    //store in the pointer portion of the new node NULL value
    because this node will be the last node in the list. If
    student makes the new node next points to what the last node
    next in the list points to, they have to do that before assign
    the last node next to the new node
    new_node->next = NULL;

    //check if the list is empty. Students may not think about
    making this work for an empty linked list because the picture
    shows a list that isn't empty. They likely don't hard code
    the value 6, so they should make this function general. Deduct
    points if this is not checked.
    if ((*head) == NULL)
        // make head point to new node
        *head = new_node;

    else{
        //traverse until temp reach the last node in the list
        by changing the memory address stored in temp pointer
        while (temp->next != NULL)
            temp = temp ->next;

        //change the next of last node to point to the new node

```

```

temp->next = new_node;
    }
}

```

Total Points: /28 /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	Has a temp pointer				
3	Set temp to point to what head points to (first node)				
3	Creates a new node				
3	Assign to pointer/reference				
3	Store/Get the data (6) inside the new node (you can infer that they will store if they refer to getting or putting a value into the new node)				
3	Assigns the new node next pointer to NULL OR Make the new node next points to what last node points to.				
3	Includes a loop to find end of list				
3	Iterates from node to node in list				
3	Put new node in list by making the last node points to the new node				
1	New node is successfully added at the end				
3	Includes a check the make sure the list is not empty				
3	Creates a function with a pointer to beginning				

	of list and the new value to be added OR Creates a function with a pointer to head pointer and the new value to be added				174
Other Interesting Observations/Comments Student Responses/Answers:					

2) add item(50) after a node that stores value 35.

//Q6_2. Function for inserting a new node (store 50) after a node that stores value 35. Student can use the head pointer to add the new node or use a temporary pointer.

```

void insertAfter(struct Node* head, int new_data)
{
    //dynamic allocating on the heap for a new node and
    storing the memory address for that location in a node
    pointer called new_node
    struct Node *new_node = (struct Node*)
    malloc(sizeof(struct Node));

    //store in the data portion of the new node the value
    that the user wants to insert
    new_node->data = new_data;

    //traverse until temp reach the node that stores value
    35 by changing the memory address stored in temp pointer.
    while(head != NULL && head ->data != 35)
    {
        head = head ->next;
    }
    if(head != NULL) {
        //make next of new node points same as next of temp
        points to.
        new_node->next = head ->next;

        //move the next of temp to point to the new node
        head->next = new_node;
    }
    else
        //delete the new node if 35 not found. We might not
        need this line because students think that 35 node exists
        as a picture of linked listed with 35 on it is provided).
        free(new_node); or only create in the if

```

```

    }

//Alternative solution for inserting a new node that has value 50
//after a node that store value 35 of a singly linked list using double
//pointers
void insertAfterDoublePointer(struct Node** head, int new_data)
{
    //declare a node pointer (temp) and set it to point the same as
    //what the head points to by storing the memory address of the
    //first node. Student cannot use the head pointer to add the new
    //node because the list will be updated only with the last 2 nodes.
    struct Node *temp = *head;

    //dynamic allocating on the heap for a new node and storing
    //the memory address for that location in a node pointer called
    //new_node
    struct Node *new_node = (struct Node*) malloc(sizeof(struct
Node));

    //store in the data portion of the new node the value that the
    //user wants to insert
    new_node->data = new_data;

    //traverse until temp reach the node that stores value 35 by
    //changing the memory address stored in temp pointer
    while(temp != NULL && temp->data != 35)
    {
        temp= temp->next;
    }
    if((*head) != NULL) {
        //make next of new node points same as next of temp points
to
        new_node->next = temp->next;

        //move the next of temp to point to the new node
        temp->next = new_node;
    }
    else
        free(new_node); //or only create in the if
}

```

Total Points: /28 /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	Has a temp pointer				
3	Set temp to point to what head points to (first node)				
3	Creates a new node				
3	Assign to pointer/reference				
3	Store/Get the data (50) inside the new node (you can infer that they will store if they refer to getting or putting a value into the new node)				
3	Includes a loop to find the node that has value 35				
3	Iterates from node to node in list				
3	Make the new node next points to what 35 next points to				
3	Make 35 next points to the new node				
1	New node is successfully added after 35				
3	Delete the new node if 35 is not found (fails elegantly)				
3	Creates a function with a pointer to beginning of list and the new value to be added OR Creates a function with a pointer to head pointer and the new value to be added				
Other Interesting Observations/Comments Student Responses/Answers:					

3) delete item(6)

177

```
//Q6_3. Function for deleting a node at the end of a singly  
linked list (node has value 6) Student can use the head  
pointer to add the new node or use a temporary pointer.
```

```
void deleteLast(struct Node *head)  
{  
  
    //return if the list is empty (there is no node in the  
list to delete)  
    if (head == NULL)  
        return;  
  
    // If the Linked List has only one node  
    else if (head->next == NULL) {  
        free(head);  
    }  
  
    else{  
        //traverse until current pointer reaches the second  
to last node in the list by changing the memory address  
stored in the temp pointer. Students may not know they can  
reference the next from the next.  
        while((head->next)->next != NULL)  
        {  
            //move temp to next node  
            head = head ->next;  
        }  
        //delete the last node  
        free(head->next);  
        //make previous next points to NULL  
        head->next = NULL;  
    }  
}
```

```
//Alternative solution for deleting a node that has value 6 at the end  
of a singly linked list using double pointers
```

```
void deleteLastDoublePointer(struct Node **head)  
{
```

```
    //declare a node pointer and set it to point the same as what the  
head points to by storing the memory address of the first node  
    struct Node *temp = *head;
```

```
    //return if the list is empty (there is no node in the list to  
delete)
```

```
    if ((*head) == NULL)  
        return;
```



```

// If the Linked List has only one node
if ((*head)->next == NULL) {
    free(head);
}

else{
    //traverse until current pointer reach the last node in the
list by changing the memory address stored in current pointer
    while(temp->next->next != NULL)
    {
        temp = temp->next;
    }
}

//delete what temp next pointer to(the memory allocation that set
for the last node)
free(temp->next);

//make last next points to NULL
temp->next = NULL;
}

```

Total Points: /19 /12	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	Has a temp pointer				
3	Set the pointer to point to what head points to				
3	Include a loop to find the second to last node				
3	Iterates from node to node in list				
3	Delete the last node				
3	Set the preceding node's pointer to NULL				
1	Node that has 6 is successfully deleted at the end				
3	Includes a check the make sure the list is not empty				
3	Includes a check the make sure the list has only one node				

3	Case is handled to search for 6 in the list				
3	Creates a function with a pointer to beginning of list OR Creates a function with a pointer to head pointer				179
Other Interesting Observations/Comments Student Responses/Answers:					

4) delete item(50)

//Q6_4. Function for deleting a node at a specific location in a singly linked list (node has value 50)

```
void deleteAfter(struct Node *head, int x)
{
    //declare 2 node pointer (current and previous) and
    set them to point the same as what the head points to by
    storing the memory address of the first node. You must
    have two pointers to current and previous to delete in the
    middle, but you can also use head as one of those pointers,
    if you know you are not deleting at the beginning
    struct Node *current = head, *previous = head;

    //what about when 50 is the first node in the list or the
    last node? We will not grade down for not thinking of
    these situations.
    //return if the list is empty (there is no node in the
    list to delete)
    if (head == NULL)
        return;
    //search for x that needs to be deleted, keep track of
    the previous node as we need to change 'prev->next'
    else{
        while (current != NULL && current ->data != x)
        {
            previous = current;
            current = current->next;
        }
    }

    if(current != NULL){
        //make previous next points to same as current
        next points to
        previous->next = current->next;
```

```
        //delete current pointer (the memory allocation
that set for the required node to be deleted)
        free(current);
```

180

```
    }
    else{
        printf("\n 50 not found");
        return;
    }
}
```

```
//Alternative solution for deleting a node that has value at specific
location of a singly linked list using double pointers
```

```
void deleteAfterDoublePointer(struct Node **head, int x)
```

```
{
    //declare 2 node pointer (current and previous) and set them to
point the same as what the head points to by storing the memory
address of the first node
```

```
    struct Node *current = *head, *previous = *head;
```

```
    //return if the list is empty (there is no node in the list to
delete)
```

```
    if (*head == NULL)
        return;
```

```
    //search for x that needs to be deleted, keep track of the
previous node as we need to change 'prev->next'
```

```
    else{
        while (current != NULL && current ->data != x)
        {
            previous = current;
            current = current->next;
        }
    }
```

```
    if(current != NULL){
        //make previous next points to same as current next points to
        previous->next = current->next;
```

```
    //delete current pointer (the memory allocation that set for the
required node to be deleted)
```

```
        free(current);
```

```
    }
    else{
        printf("\n 50 not found");
        return;
    }
}
```

```
}
```

Total Points: /22 /9	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/not¹⁸¹ getting full marks or anything else interesting about their conceptual and procedural understanding
3	Have 2 node pointers				
3	Set at least one of them to point to what head points to and the other pointer to NULL				
3	Include a loop to find the node that store 50				
3	Make one pointer point to 50 or after 50				
3	Make another pointer point to the node previous to 50				
3	Make the previous pointer points to what 50 next points to				
3	Delete the node that store 50				
1	Node that has 50 is successfully deleted				
3	Includes a check the make sure the list is not empty				
3	Case is handled if 50 is not found (fails elegantly)				
3	Creates a function with a pointer to head pointer and the value to be deleted OR Creates a function with pointer to the head pointer of list and the value to be deleted				
Other Interesting Observations/Comments Student Responses/Answers:					

Interview Explanation about Coding Questions, Answer and Rubric

Q7. Which of the functions you wrote would change if you add a **tail** pointer?

- how would the functions change?

The functions/codes would change are:

1. Create an empty linked list: set an additional tail pointer to point to Null
2. Insert beginning: if the list is empty (both head and tail point to Null), head and tail need to point to the new node.
3. Delete beginning.: if there is only one node, set head and tail pointers to NULL
4. Insert after maybe would change: if the node after is the last node, then same as insert at the end. **(might not think about this if they didn't think about adding to the end with this function, so we will not take off points for this)**
5. Insert at the end: direct access the end through the tail and add the new node and then update the tail to point to the new node.
6. Delete at the end: We need to update the tail to point to the previous node by iterating the list to second last node, makes it points to NULL and then free the last node.

Total Points: /15 /3	Point Divvy: Gain points for mention of each of these topics:	States what needs to be change	Partially Correct to states how to change the what	Correctly states how to change the what	Comment: Reason for not getting full marks or anything else interesting about their conceptual and procedural understanding
3	Create an empty linked list: Set an additional tail pointer to point to NULL				
3	Insert at the beginning: <u>If the list is empty</u> (both head and tail point to NULL), set head and tail to point to the new node.				
3	Delete at the beginning: <u>If there is only one node</u> , set head and tail to NULL				
3	Insert at the end: Access the end directly through the tail and add the new node and update the tail to point to the new node				
3	Delete at the end: Iterate the list to update the tail to point to the second last node , make this node to points to NULL and then free the last node				
3	Insert after maybe would change: <u>If the node after is the last node</u> (same as insert at the end), add the new node and update the tail to point to the new node.				
Other Interesting Observations/Comments Student Responses/Answers:					

Q8. Which of the functions you wrote would change if this was a **circular** linked list?

○ how would the functions change?

1. Any while loop pointer checks will be for the head node instead of Null
2. Insert at beginning: If the insertion is for the first node in the list (list is empty), put this node to point to itself instead of null. If the list is not empty: Find the last node, and now make the last node and the head/first node point to the new node to be inserted.
3. Insert at end: Find the last node and make the last node point to the new node, and the new node will point to the head/first node.
4. Delete at beginning: Find the last node, and then make last node point to the same as the head pointer points to after deletion.
5. Delete at the end: Find the second to last node, delete the last node, and make the new last node point to the same as head.

Total Points: /18	Point Divvy: Gain points for mention of each of these topics:	States what needs to be change	Partially Correct to states how to change the what	Correctly states how to change the what	Comment: Reason for not getting full marks or anything else interesting about their conceptual understanding
3	Any function with a while loop: pointer checks will be for head/first node instead of NULL				
3	Insert at the beginning: <u>If the list is empty</u> , put this node to point to itself instead of null.				
3	Insert at the beginning: <u>If the list is not empty</u> , find the last node and make the last node and the head point to the new node to be inserted.				
3	Insert at the end: Find the last node and make the last node point to the new node, and the new node will point to the head/first node.				
3	Delete at the beginning: Find the last node, and then make last node point to the same as the head points to after deletion.				
3	Delete at the end: Find the second to last node, delete the last node, and make the new last node point to the same as head.				

Other Interesting Observations/Comments Student Responses/Answers:

Q9. Which of the functions you wrote would change if this was a **doubly** linked list?

184

o how would the functions change?

1. Create an empty linked list: Add previous pointer in node struct
2. Insert new node at beginning: If the list is empty, the new node previous and next are set to NULL. If the list is not empty, the previous of the new node is set to NULL and the previous of the old first node is set to the new node.
3. Insert new node after a specific node: If the specific node is the last node, set the previous of the new node points to the specific node to be inserted after. If the specific node is not the last node, then set the previous of the node, after the specific node to insert after, points to the new node and set the previous of the new node to point to the specific node to insert after.
4. Insert new node at end: If the list is empty, the new node previous and next are set to NULL. If the list is not empty, the previous of the new node is set to the last node.
5. Delete at the beginning: after deleting the node, set head->previous to point to Null.
6. Delete node after a specific node: the previous of node after the node being deleted needs to point to the same thing as the previous of the node being deleted

Total Points: /21 /6	Point Divvy: Gain points for mention of each of these topics:	States what function needs to change	Partially Correct to states how to change the what	Correctly states how to change the what	Comment: Reason for not getting full marks or anything else interesting about their conceptual understanding
3	Create an empty linked list: Add previous pointer in node struct				
3	Insert at the beginning: <u>If the list is empty</u> , set new node previous and next to NULL				
3	Insert at the beginning: <u>If the list is not empty</u> , set the previous of the new node to NULL and set the previous of the old first node to the new node .				
3	Insert at specific location: <u>If the specific node is not the last node</u> , make the previous of the node , after the specific node to insert after, points to the new node and set the previous of the new node to point to the specific node .				
3	Insert at the end: <u>If list is not empty</u> , Set the previous of the new node to the last node .				
3	Delete at the beginning: After deleting the node , set				

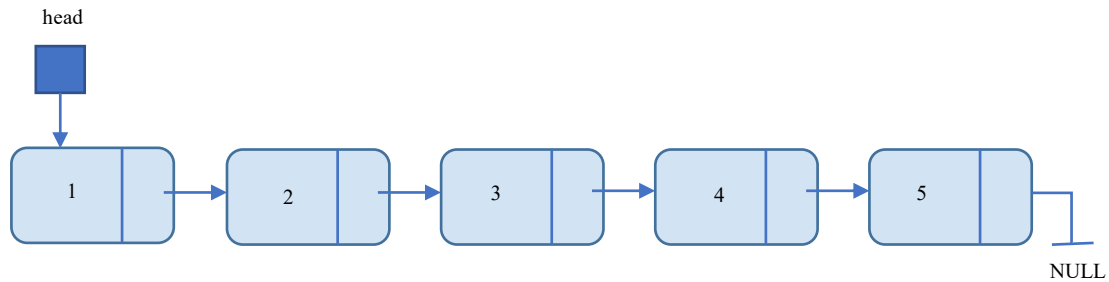
	head->previous to point to Null.				
3	Delete at specific location: Make the previous of node after the node being deleted needs to point to the same thing as the previous of the node being deleted.				185
3	Insert at specific location: <u>If the specific node is the last node</u>, set previous of the new node to point to the specific node to be inserted after				
3	Insert at the end: <u>If list is empty</u>, set the new node previous and next to NULL				

Other Interesting Observations/Comments Student Responses/Answers:

Appendix D: Recognition Questions

Interview Recognition Questions, Answer and Rubric

187



A singly linked list stores the following numbers “1, 2, 3, 4, 5”, is constructed using the following C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
    int data;
    struct node* next;
};

void push(struct node** head_ref, int new_data)
{
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

int main()
{
    struct node* head = NULL;

    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    return 0;
}
```

1. Using one sentence to explain what the following function does.

188

```

struct node* A(struct node* head)
{
    if (head == NULL || head->next == NULL)
        return head;

    struct node* curr = head->next->next;
    struct node* prev = head;
    head = head->next;
    head->next = prev;

    while (curr != NULL && curr->next != NULL)
    {
        prev->next = curr->next;
        prev = curr;
        struct node* next = curr->next->next;
        curr->next->next = curr;
        curr = next;
    }

    prev->next = curr;

    return head;
}

```

Swap adjacent nodes (2 1 4 3 5)

Total Points: /4	Point Divvy: Gain points for mention of each of these topics:	Partially Correct	Correct	Comment: Reason for getting/not getting full marks or anything else interesting about their conceptual understanding
2	Understands code is swapping nodes.			
2	Understands the code is swapping adjacent nodes.			
Other Interesting Observations/Comments Student Responses/Answers:				

2. Using one sentence to explain what the following function does.

```

bool B(struct node* head, int x)
{
    struct node* current = head;

    while (current != NULL)
    {
        if (current->data == x)

```

```

        return true;
        current = current->next;
    }
    return false;
}

```

189

Find an integer x in a linked list

Total Points: /2	Point Divvy: Gain points for mention of each of these topics:	Partially Correct	Correct	Comment: Reason for getting/not getting full marks or anything else interesting about their conceptual understanding
2	States the function is to find an integer x in a linked list			
Other Interesting Observations/Comments Student Responses/Answers:				

3. Using one sentence to explain what the following function does.

```

void C(struct node* head)
{
    struct node* current = head;

    while (current != NULL)
    {
        printf("%d ", current->data);
        current = current->next;
    }
}

```

Print the whole linked list's data

Total Points: /2	Point Divvy: Gain points for mention of each of these topics:	Partially Correct	Correct	Comment: Reason for getting/not getting full marks or anything else interesting about their conceptual understanding Notes
2	States the function is to print the entire elements in the linked list			
Other Interesting Observations/Comments Student Responses/Answers:				

4. Using one sentence to explain what the following function do.

```

void D(struct node *head)
{
    struct node *current = head;
    struct node *next;

    while(current != NULL)
    {
        next = current->next;
        free(current);
        current = next;
    }
    head = NULL;
}

```

190

Clear the entire linked list

Total Points: /2	Point Divvy: Gain points for mention of each of these topics:	Partially Correct	Correct	Comment: Reason for getting/not getting full marks or anything else interesting about their conceptual understanding
2	States the function is to clear (or free) the entire nodes in the linked list			
Other Interesting Observations/Comments Student Responses/Answers:				

Appendix E: Participants' Grading in the Survey

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
P1	✓	✓	✓	✓	✗	.5	✓	✓	.5	✓	✓	✓	✗	✓	✓	1.5	✓	✓	✓	✗	✓	.25	✓	✓	✗	✓	✓	
P2	✓	✓	✗	✓	✓	✓	✓	✓	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	.25	✓	✓	✓	✓	✓	
P3	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✓	✓	✗	✓	✓	1.5	✗	✗	✗	✗	✗	.75	✓	✓	✗	✗	✗	
P4	✓	✓	✓	✗	✗	✓	✗	✗	.5	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	.25	✓	✓	✓	✓	✓	
P5	✓	✓	✓	✓	✗	✓	✓	✗	1.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
P6	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✗	✓	✓	1.5	✓	✓	✓	✓	✗	.25	✓	✓	✗	✗	✓	
P7	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	1	1.5	✓	✓	✓	✓	✓	.25	✓	✓	✓	✓	✗	
P8	✓	✓	✓	✓	✓	✓	✗	✓	.5	✓	✓	✓	✓	✗	✓	1	✓	✓	✓	✓	✓	.75	✓	✓	✓	✓	✗	
P9	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	1	1.5	✓	✓	✓	✓	✓	✓	.25	✓	✓	✓	✓	✗	
P10	✓	✗	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
P11	✓	✓	✓	✓	✗	✓	✓	✗	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	.75	✗	✓	✓	✓	✗	
12	✓	✓	✓	✗	✓	✓	✓	✓	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
P13	✓	✓	✓	✓	✗	✓	✓	✗	.5	✓	✓	✓	✓	✗	✓	1	✓	✓	✓	✓	✗	.5	✗	✗	✓	✓	✗	
P14	✓	✓	✗	✗	✗	✓	✓	✗	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.5	✓	✗	✓	✓	✗	
P15	✓	✓	✓	✓	✗	✓	✓	✗	1	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	.5	✗	✓	✓	✓	✓	
P16	✓	✗	✓	✓	✗	✓	✓	✗	✗	.5	✗	✓	✗	✓	✓	1.5	✓	✓	✓	✓	✓	.75	✓	✓	✓	✓	✗	
P17	✓	✓	✓	✓	✗	✓	✓	✗	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	.25	✓	✓	✓	✓	✓	
P18	✓	✓	✓	✓	✗	✓	✓	✓	.5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
P19	✓	✓	✓	✓	✓	✓	✓	✓	1.5	.5	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	.5	✓	✓	✓	✓	✓	
20	✓	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	1	1	✓	✓	✓	✓	✓	.5	✓	✓	✗	✓	✓	
P21	✓	✓	✗	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	.25	✓	✓	✗	✗	✗	
P22	✓	✓	✗	✗	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.25	✗	✓	✓	✗	✓	
P23	✓	✓	✓	✓	✓	✓	✓	✗	.5	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
P24	✗	✓	✓	✓	✗	✓	✓	✗	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✗	✓	.75	✓	✓	✗	✗	✓	
P25	✗	✗	✓	✓	✗	✓	✗	✗	1.5	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	.25	✗	✓	✓	✗	✓	
P26	✓	✓	✓	✗	✓	✓	✗	✓	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	.5	✓	✓	✓	✓	✓	
P27	✓	✓	✓	✓	✗	✓	✓	✗	.5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.75	✗	✓	✓	✓	✓	
P28	✓	✓	✓	✓	✗	✓	✓	✗	.5	✓	✓	✓	✓	✓	✓	1.5	✓	✓	✓	✓	✓	.75	✓	✓	✓	✓	✓	

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	
P29	X	✓	X	✓	X	✓	X	X	✓	✓	✓	✓	✓	✓	✓	✓	1	✓	✓	X	✓								
P30	X	X	X	X	X	✓	✓	X	1	✓	✓	✓	X	✓	✓	✓	1.5	✓	✓	X	✓	✓	X	✓	X	X	✓	✓	
P31	✓	✓	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.75	✓	✓	✓	✓	✓	✓	✓
P32	✓	✓	X	✓	✓	✓	X	✓	.5	✓	✓	✓	✓	✓	✓	✓	✓	X	X	✓	✓	.25	✓	✓	✓	✓	✓	✓	✓
P33	✓	✓	X	✓	✓	✓	✓	X	X	✓	✓	✓	✓	✓	✓	✓	1.5	✓	✓	X	X	.75	✓	✓	✓	✓	✓	✓	✓
P34	✓	✓	X	X	X	✓	✓	X	X	X	✓	X	X	X	✓	✓	1	✓	X	X	X	.25	✓	✓	✓	✓	✓	X	✓
P35	✓	✓	✓	✓	✓	✓	✓	X	.5	✓	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	.25	✓	✓	✓	✓	✓	✓	✓
P36	X	X	✓	✓	X	✓	X	X	X	✓	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	✓	.75	X	✓	✓	X	✓	✓	✓
P37	✓	✓	X	X	X	.5	✓	✓	.5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	.25	X	✓	✓	✓	✓	X	✓
P38	X	✓	✓	✓	X	.5	X	X	.5	X	✓	✓	✓	✓	✓	✓	1	✓	✓	X	✓	.25	X	✓	✓	✓	✓	X	✓
P39	✓	✓	✓	✓	X	✓	✓	✓	.5	✓	✓	✓	✓	✓	✓	✓	1	✓	✓	✓	✓	.25	✓	✓	✓	✓	✓	✓	X
P40	✓	✓	✓	✓	X	✓	✓	X	X	✓	✓	✓	✓	✓	✓	✓	1.5	✓	✓	X	X	✓	✓	✓	✓	✓	✓	✓	✓

Table E.1. Participants' scores on each question from in the survey. The green check-mark for the correct answer and the red cross-mark is for the wrong answer. Note that the highlighted results are the results for the 11 participants we interviewed in semi-structured interview.

Appendix F: Scores Per Categories in the Survey for Interviewed
Participants

Type Description	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate	Total Points	# Correct Response
SLL ^a - Q4	0	1	1	1	0	1	1	1	0	0	0	1	6
Q18	1	1	0	1	1	1	1	1	1	1	1	1	10
SCLL ^b - Q12	1	1	1	1	1	1	1	1	1	1	1	1	11
DLL ^c - Q13	1	1	0	1	1	1	1	0	1	1	1	1	9
Q19	1	1	0	1	1	1	1	1	1	1	1	1	10
DCLL ^d - Q11	1	1	1	1	1	1	1	1	1	1	1	1	11
Pieces	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate	Total Points	# Correct Response
Node - Q1	1	1	0	1	1	1	1	1	1	1	1	1	10
Q7	1	0	0	1	0	1	1	1	1	1	1	1	8
Node pointer - Q2	1	1	1	1	1	1	1	1	1	1	1	1	11
Q5	0	1	0	0	0	0	0	0	1	1	0	1	3
Q8	0	1	0	0	0	0	1	0	1	1	1	1	5
Q10	1	1	0	1	1	1	1	1	1	1	1	1	10
List Data - Q15	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	11
Q16	1	1	1	1	1	1	1	1	1	1	1	1	11
Q17	2	1	1.5	1.5	1	1.5	1.5	1.5	1.5	1.5	2	2	2
Head - Q6	1	1	1	1	1	1	0.5	1	1	1	0.5	1	9
Tail - Q9	1	0.5	0	0.5	0.5	1.5	0.5	0	0.5	0	0.5	2	0
Operations	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate	Total Points	# Correct Response
Empty List - Q3	0	1	1	1	1	1	1	1	1	0	0	1	8
Delete Beginning Q20	1	1	0	1	1	1	0	1	1	0	1	1	8
Clear List - Q21	1	1	0	1	1	1	1	0	1	0		1	7
Prerequisite Knowledge	Joe	Bob	Suzy	Bill	Max	Phil	Feng	Xeng	Chemi	Ecer	Nate	Total Points	# Correct Response
Freeing Memory - Q22	0.25	.75	.75	.75	.25	1	.25	.25	1	.75	.25	1	2
Pointer Variable Declaration - Q23	1	1	1	0	0	1	0	0	1	1	0	1	6
Pointer Variable Assignment - Q24	0	1	1	1	1	1	1	1	1	1	1	1	10
Q25	1	1	0	1	1	1	0	1	1	1	1	1	9
Q26	1	1	0	1	1	1	0	0	1	1	1	1	8
Q27	0	0	0	0	1	1	1	0	1	1	0	1	5
Pointer Variable Manipulation-Q28	1	1	0	1	1	1	1	1	1	1	1	1	10

^aSLL refers to a singly linked list

^bSCLL refers to a singly circular linked list.

^cDLL refers to a doubly linked list

^dDCLL refers to a doubly circular linked list.

Table F.1. Scores per linked list concept in the survey based on conceptual (light-purple) understanding.

