Identifying Student Difficulties with Basic Data Structures

Daniel Zingaro
University of Toronto Mississauga
daniel.zingaro@utoronto.ca

Cynthia Taylor
Oberlin College
ctaylor@oberlin.edu

Leo Porter
University of California, San Diego
leporter@eng.ucsd.edu

Michael Clancy
University of California, Berkeley

Cynthia Lee
Stanford University

Soohyun Nam Liao
University of California, San Diego

Kevin C. Webb
Swarthmore College

ABSTRACT

To be effective instructors and CS education researchers, we must identify and understand student difficulties surrounding core computing topics. This study examines student difficulties with the basic data structures commonly found in CS2 courses. Initial exploration of student thinking began with think-aloud interviews with students. These interviews centered on open-ended questions that were iteratively improved upon based on analysis of interview transcripts. The revised open-ended questions were then posed to 249 students during an end-of-term final exam study session. Using the explanations and justifications included by students, responses to the questions were coded and summarized. This work characterizes the difficulties revealed by student responses, and provides details of their prevalence among the examined student population.

CCS CONCEPTS

• **Social and professional topics → Computing Education**;

KEYWORDS

CS2, data structures, difficulties

ACM Reference Format:
Daniel Zingaro, Cynthia Taylor, Leo Porter, Michael Clancy, Cynthia Lee, Soohyun Nam Liao, and Kevin C. Webb. 2018. Identifying Student Difficulties with Basic Data Structures. In *ICER '18: 2018 International Computing Education Research Conference, August 13–15, 2018, Espoo, Finland.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3230977.3231005

1 INTRODUCTION

It is important for instructors to be aware of common student difficulties and errors related to the content domain being learned [20]. For example, a recent study of middle school physical science teachers and students found that teacher familiarity with student wrong answers was positively correlated with student learning gains [18].

Prior research has discussed both student misconceptions and student difficulties [4, 16]. A student **conception** describes a belief, theory or explanation previously developed to explain some behavior observed in the world [1]. When these beliefs are in conflict with accepted scientific theories, they become **misconceptions** [1]. A **difficulty** refers to an observable error committed by students [4]. The present paper focuses on student difficulties, which through additional study and corroboration may illuminate upstream misconceptions.

There are many studies that examine student difficulties and misconceptions in CS1 [17, 22]. In contrast, the relevant literature for data structures is in its infancy; for example, linked lists have received very little attention [10].

This work aims to further our understanding of student difficulties regarding Basic Data Structures, including ArrayLists, singly- and doubly-linked lists, and binary search trees. We began by recruiting students to participate in think-aloud interviews about Basic Data Structures problems. The interview results informed our authoring of a series of questions aimed to elicit student difficulties. To gather a larger data set, we then presented the questions to 249 students during a final exam study session. The students responded with an answer and a justification for that answer, which we coded to identify common difficulties. We explore these difficulties here.

2 BACKGROUND

There is a vast literature around student difficulties in introductory programming (CS1). That work has led to the discovery of surprising student misconceptions and has informed concept inventory development [5]. For example, researchers have found that students infer unwarranted relationships between variables, and believe that memory is reserved for uninstantiated objects [5]. Even fundamental concepts typically taught at the start of CS1, such as primitive/object variables, value/reference assignment, and parameter passing, are associated with considerable variation in student understanding [7, 8, 21]. Several comprehensive literature reviews demonstrate the extent to which researchers over the years have studied CS1 misconceptions and difficulties [17, 22]. In fact, one such literature review calls for the community to move away from identifying additional misconceptions for CS1 and to instead begin working to foster changes in CS1 in response to those that have already been identified [15].

Recent work highlights two core components of typical CS2 courses: Recursion and Basic Data Structures [14]. A large number of difficulties and misconceptions associated with recursion have been documented [4, 9]. By comparison, there are fewer reports of difficulties for introductory data structures topics. Tenenberg and Murphy tested students on data structures as part of a project on student self-assessment, and found that students performed best on questions about stack, queue and tree interfaces, and worst on questions concerning the runtime efficiency of different searches [24]. Some have studied student misconceptions of heaps, including the ways that heaps can be represented or constructed [13, 19]. Particularly relevant to the present study is work by Karpierz and Wolfman [6], who used interviews and exam/project analysis to identify student misconceptions of binary search trees (BSTs) and hash tables. The discovered BST misconceptions are: (1) a separate search through the tree is required to ensure that the element is not already in the tree before inserting it, (2) all keys in the tree must be inspected before inserting a new key, and (3) a BST is balanced by default. Interestingly, some have found that students conflate binary search trees and heaps [2, 13], while others have not replicated this [6].

For several reasons, we argue for the continued study of student difficulties with introductory data structures. First, echoing arguments from other researchers [25], student difficulties serve as sources of pedagogical content knowledge (PCK), helping instructors anticipate likely struggles and how to guide student understanding in the context of those struggles. Second, introductory data structures are a core component of many CS2 courses, and CS2 itself often serves as the entrypoint to a CS major [14]. Finally, difficulties with some data structures (e.g., linked lists, trees) are studied in only a small number of papers [10, 11]. That is, while the community has been urged to stop itemizing CS1 difficulties [15], there is much that we do not know about post-CS1 difficulties.

3 METHODOLOGY

We began by developing open-ended questions designed to cover a set of learning goals and topics found to be important to the teaching of Basic Data Structures [14]. In order to ensure that these questions highlight a range of student understanding, we conducted interviews in which students were asked to think aloud while solving these problems. Through the course of these interviews, we frequently modified and adapted the questions to better reflect and surface student difficulties.

A total of 65 interviews were conducted at three different North American schools: one private and two public research-intensive universities. Students were recruited for participation from each institution at the end of the course that taught Basic Data Structures. Participation was approved by the Human Subjects Board for each institution, and students were compensated for their time with gift cards.

After completing and analyzing our student interviews, we presented the open-ended questions to students at a public, research-intensive university as part of a final exam study session for a Java-based CS2 course. We describe the test as "open-ended", even though some questions included multiple choice options, because every question asked students to provide written justifications for their answers. The justifications often provided insight into difficulties, sometimes even when students selected correct answers.

Question 1

The Node and LinkedList class defined above implement a singly-linked list with head and tail references. Head refers to the first node of the list, and tail refers to the last node of the list.

Given below is a method in the LinkedList class to add an element to the end of the list.

```
DEFINE addEnd(n)
   IF tail == nil THEN
      head = tail = new ListNode(n)
   ELSE
      //MISSING CODE
   END IF
ENDDEF
```

Supply the missing code:

Two hundred forty-nine students gave consent for their responses to be used in this research project. We coded the responses for each question using an open coding technique in which categories were developed from common patterns in student answers. Each question was coded by one member of the research team.

4 STUDENT DIFFICULTIES

In this section, we present our questions, characterize the students' answers, and highlight common difficulties. To avoid distractions that may stem from the low-level details of any particular programming language, we developed our questions using pseudocode. Our pseudocode is modeled after that of [12], with a few small additions to better support data structures (e.g., a nil value).

4.1 Adding to the Tail of a Linked List

We designed Question 1 to test students' ability to implement linked list methods. A correct answer to this question is the following:

```
tail.next = new ListNode(n)
tail = tail.next
```

In our open-ended test, 67% of students answered this question correctly. We identify three common errors, with each coded separately (as such, percentages may add to greater than 100%): 16% of students failed to update the tail pointer, 12% failed to correctly attach the new node, and 10% unnecessarily looped through the list to find the tail.

4.1.1 Failure to Update the Tail Pointer. A representative example of a student response that included this mistake appears below. In this response, the student correctly creates a new node and connects that new node to the end of the list, but then fails to set the tail to point to the new end of the list.

```
ListNode end = new ListNode(n)
tail.next = end
```

For this error and the next, it is difficult to discern whether the mistake reflects a misconception (e.g., students do not believe they are responsible for maintaining the invariant that the tail points to the last element in this list), or whether they simply forgot to update the pointer.

4.1.2 Failure to Attach the New Node. For this error, students fail to add the new element to the end of the list. An example of a student response expressing this mistake appears below:

```
tail = new ListNode(n,nil)
```

In this response, the student assumes that there exists a constructor that takes both the element and the next node as parameters. While this is not consistent with the provided sample code, we did not mark this as incorrect. However, they fail to correctly set the old tail to point to the new tail.

We note that this error was slightly more common in conjunction with the "unnecessary loop" mistake that we discuss next. Similar to the prior mistake, we are unclear as to whether the students fundamentally misunderstood the need to correctly point the second-to-last element to the last element or instead that they made a careless mistake.

4.1.3 Unnecessary Loop to Find the Tail. A representative example of a student response demonstrating this inefficiency appears below:

```
ListNode newNode = new ListNode(n)
ListNode curr = head

WHILE curr.next != tail DO
      curr = curr.next
ENDWHILE

curr.next = newNode
tail = newNode
```

This difficulty is of interest as it embodies both **using and not using** the tail pointer. Students do understand that the tail pointer is present, and that it points to the last node: in 23 of the 25 cases where the student uses a loop, they also use the tail pointer somewhere in their code. In the sample response above, the student uses the tail pointer to find the node just before the end of the list in the while-loop, and updates the tail pointer to point at the new last node (unfortunately discarding the old last node). Other student responses correctly loop to the last node and then properly update the tail. None of these students, however, makes the leap to use the tail pointer to avoid the costly iteration through the loop. They may have simply memorized that looping through the list is always how the last node in a singly-linked list is accessed. This indicates that instructors may want to encourage students to thoroughly consider the impact of modifications to a data structure.

4.2 Fast Access to an Index

Question 2 was designed to test students' ability to choose an appropriate data structure when implementing a program. The correct answer to this problem is *d*, an ArrayList, since an ArrayList allows $O(1)$ access time to a specific position in the list, while either a singly- or doubly-linked list will take time $O(N)$ to access an element based on its position in the list.

Question 2

Suppose that your program is initialized with a set of numbers that is each added to a list. The user is then permitted to query for the number at a given position (index) in the list, and can make as many queries as they wish.

Which List data structure implementation would provide the best performance for the initialization and the user queries? You may assume that the number of user queries is many times more than the number of elements in the list.

a. a singly-linked list

b. a sorted doubly-linked list

c. an unsorted doubly-linked list

d. an ArrayList

Briefly explain your reasoning:

Seventy-five percent of students correctly chose option *d*, and 165 of the 188 students who answered correctly discussed fast access by index in their justification.

4.2.1 Binary Search on the Sorted Doubly-Linked List. Ten percent of students incorrectly chose *b*, the sorted doubly-linked list. Of the 26 students who chose this answer, five specifically discussed using binary search on a doubly-linked list, despite the impossibility of accessing the list by index. It may be that these students have some fundamental misconceptions about either the requirements of binary search or how elements are accessed in a linked list.

4.2.2 Binary Search on an ArrayList. Five percent of students correctly selected the ArrayList, but then justified it by saying that an ArrayList is searchable in $O(logN)$ time with binary search. These students are likely missing that search is not necessary to solve this problem, perhaps due to typical comparisons of data structures that focus on search time.

4.2.3 Searching from Both Ends of a Doubly-Linked List. Four percent of students claimed that a doubly-linked list would be faster to search because one could search from either end, depending on where the index was closest. For example, "The doubly link[sic] list will allow you to determine the start of the search. If the index is on the right half of the list, then you would want to start at the tail & move backwards." This justification was used for selecting both the sorted and unsorted doubly-linked list.

Regardless of whether the list is sorted or unsorted, searching from both ends will not give any sort of performance optimization. Students using this justification with a doubly-linked list appear to believe they can somehow search from both directions in parallel, or can instantly figure out from which end to start the search.

Question 3

Suppose that we wanted to insert the numbers 1 to 15 into a binary search tree. In what order should we insert the elements so that the tree is as balanced as possible?

4.3 Inserting into a BST

Question 3 probes students' ability to reason about the shape of a binary search tree (BST). In particular, students are expected to choose an insertion order for a range of numbers such that it ultimately produces a desirable BST structure: a fully balanced tree. Overall, 44% of students provided at least one correct insertion order. An additional 10% described a correct procedure to generate the order without specifying a sequence of integers, and another 5% drew a fully balanced tree containing all the values without indicating how such a tree might be constructed.

4.3.1 Starting in the Middle, Alternating Outwards. A common and interesting difficulty surfaced in which 6% of the students applied the correct logic to identify the root node, but then abandoned that logic for subsequent nodes under the root. That is, they identified that 8 should be the root of the tree, which produced two equally-sized trees below the root, but then generated a long chain of nodes for each of the root's subtrees. After inserting 8, these students provided sequences that alternated inserting values that were one larger or smaller than what was inserted previously. For example,

8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15.

Our interpretation of this result is that some students believe that only the root needs to contain an equal number of children on the left and right for the tree to be considered balanced. They do not consider applying the same criterion recursively, despite properties of BSTs often being defined recursively.

4.3.2 Starting at the Wrong Root (7). Another 6% of the student responses make the mistake of starting with 7 to be the root rather than 8. This error indicates that some students have difficulty choosing the root of the tree, particularly when the maximum element (15) is not evenly divisible by 2. We saw similar misunderstandings during our think-aloud interview phase. In those interviews, when students made the initial choice of 7 as the root, they repeatedly struggled to balance the tree rather than reevaluate their choice of root.

4.3.3 Randomness. Four percent of the responses asserted that, to be balanced, the values must be inserted in random order. This response suggests that some students may equate "random order" with "good order", despite a random order not being guaranteed to produce a balanced tree.

4.3.4 Min- or Max-Heap. Prior to collecting our student responses, we expected that some students might respond to Question 3 with answers that conflate binary search trees with heaps, as shown by Danielsiek et al [2]. Like Karpierz and Wolfman [6], we found that very few student responses—only 1%—seemed to exhibit this confusion.

Question 4

Consider a binary search tree containing N values whose root is presently the median value in the tree. Suppose that N more values are added to the collection using the standard tree insertion algorithm, with all N larger than the largest value currently stored. Which of the following is true? (There may be more than one correct answer.)

a. The root still has a median value in the tree.

b. There are more values in the right subtree of the updated tree than in the left subtree.

c. Adding the Nth new value to the collection can take time proportional to log N.

d. Adding the Nth new value to the collection can take time proportional to N.

e. Adding the Nth new value to the collection can take time proportional to N^2.

Briefly explain your reasoning:

4.4 BST Changes After Insertions

Question 4 was designed to test students' ability to predict the behavior of an algorithm that operates on a data structure.

The correct answer to this problem is *b*, *c*, and *d*. Option *a* is no longer true, since the root will stay the same but the median value of the tree will change as all the added values are larger than the largest value currently in the tree. Option *b* is true, since all of the new values will be added to the right subtree. Option *c* will be true if the new values have been added in an order that makes the right subtree a balanced tree. Option *d* will be true if the new values have been added in an order that makes the right subtree unbalanced. Option *e* cannot be true, since there are fewer than N^2 values in the tree. Only 12% of students answered this question correctly.

4.4.1 Adding Cannot Take $O(N)$ Time. Forty-five percent of students did not select *d*, indicating they did not think the time to insert could be $O(N)$. This may be an example of the "default balanced" misconception identified by Karpierz and Wolfman [6], in which students assume that BSTs will automatically have a balanced shape. The large number of students who failed to select this answer indicates that instructors should emphasize the worst-case results of insertion into a BST.

4.4.2 Adding Cannot Take $O(logN)$ Time. Twenty-nine percent of students did not select *c*, indicating they did not think the time to insert could be $O(logN)$, with the majority of these students indicating they had interpreted the question to mean that every inserted value was larger than every previously inserted value, rather than just being larger than the original values in the tree. (The question has since been clarified.)

Question 5

Consider the design of a new data structure we will call a *generalized array*. A *generalized array* is like a list, except that its subscript values may be any collection of integers and not just a contiguous range of integers. For example, one might set up a *generalized array* named vals to have three subscripts—say, -5, 42, and -7001—and then to assign to or access vals[-5], vals[42], and vals[-7001]. An attempt to access the *generalized array* using any other subscript values would be illegal.

An implementation of a *generalized array* should allow fast access to its elements for large subscript sets. You have control over how you create the data structure (e.g., which data structure, insertion order of elements, etc.) Which of the following data structures best satisfy this requirement? **(There may be more than one correct answer.)** Assume that the set of subscript values is defined before the program is executed and does not change during the program run.

a. an ArrayList of subscript/value pairs, ordered by subscript

b. an ArrayList of subscript/value pairs, ordered by value

c. a linked list of subscript/value pairs, ordered by subscript

d. a linked list of subscript/value pairs, ordered by value

e. a binary search tree of subscript/value pairs, ordered by subscript

f. a binary search tree of subscript/value pairs, ordered by value

Briefly explain your reasoning:

4.4.3 The Root Remains the Median. Twenty-one percent of students chose *a*, indicating that they thought the root would stay the median. Of the 53 students that selected *a*, 32 of these students explicitly offered the fact that the root would stay the same as a justification for their answer. (Of the remaining 21 students, only two offered an alternate justification for their answer, with the rest not explaining their answer in any way.) This indicates that students have internalized the idea that once in a BST, a node will stay in the same position unless nodes are removed; however, they have memorized this idea to the extent that it did not occur to them that while the root value will not change, the median value will.

4.5 Generalized Array

We designed Question 5 to assess students' ability to choose an appropriate implementation of a data structure. The correct answer is both *a* and *e*, since the design should optimize for quick access by subscript, and does not need to optimize for insertion or deletion time. Both a sorted ArrayList and a BST ordered by subscript will enable finding a subscript in time $O(logN)$, since binary search can be used on the sorted ArrayList and elements can be inserted appropriately into the BST to make it balanced. Only 22% of students answered this question correctly.

4.5.1 Binary Search Tree is Always Fastest. The most popular answer to this question, chosen by 24% of students, was *e*, a binary search tree ordered by subscript. Student answers specifically mentioned the quick time to find an element in a BST; e.g., "A BST allows for the fastest search time complexity of the three, at $O(logN)$." However, these answers overlook the fact that an ArrayList can also be searched in $O(logN)$ time using binary search. This error may be caused by instructors focusing on the ability to quickly find an element in a BST, while not emphasizing the same ability for sorted ArrayLists.

4.5.2 Accessing by Negative Index. Twelve percent answered only option *a*. Of the 30 students who answered this way, 19 of them justified their answer by saying that an ArrayList would offer direct or fast access to an index. This suggests that they believe that accessing an ArrayList with a negative index would be possible. This may partially have been prompted by our using array style references in the problem description; we have since rewritten this problem to avoid that array-like syntax. However, even with this problem phrasing, these students believing that vals[-5] is a valid way to access an ArrayList is troubling, and may point to a misconception about valid ArrayList indices.

4.5.3 Ignoring Performance. Twelve percent of students' answers included either of the linked list responses (*c* or *d*), with the most popular of these answers being *a*, *c*, and *e* (answered by 4% of students). The justifications for these answers generally did not mention performance, and focused on the fact that an implementation **could** be built using any of the data structures. (A sample student answer is "All three data structures should work as long as they arrange the elements by subscript meaning that if the subscript can't be found in all three of the structures, it wouldn't exist in general and you can't access its value unless you know the subscript.") As we discuss in Section 4.6.1, this indicates that instructors may wish to enhance their coverage of performance tradeoffs.

4.6 Implementing Undo

We designed Question 6 to test students' ability to correctly use existing data structures when implementing program functionality. The correct answer to this problem is *d*, to add and remove from the head, as this will maintain a LIFO ordering and both of these operations can be performed in constant time.

Seventy-two percent of the students correctly chose *d* for this problem, and of these, 60% mentioned the LIFO property in their explanation of their answer. Seven percent chose *a* or *b*, choices that do not provide a LIFO property.

4.6.1 Ignoring Performance. Thirteen percent of students chose both *c* and *d*, and an additional 4% chose only *c*. (Four percent did not answer the question.) The relatively large percentage of

Question 6

Imagine you are implementing "undo" functionality in a program: you want to save the user's actions with the ability to undo them in reverse order. For example, if a user performs operations a, b, and then c, activating your undo function would undo first c, then b, then a. **With the goal of providing the best performance for your undo operation, which of the following data structures would you use to store the user's actions? Check all that apply.**

For each of the following choices, assume the Singly-linked List implementation has a reference only to its head (but not its tail).

a. A singly-linked List that adds to tail and removes from head.

b. A singly-linked List that adds to head and removes from tail.

c. A singly-linked List that adds to tail and removes from tail.

d. A singly-linked List that adds to head and removes from head.

Briefly explain your reasoning:

students who chose *c*, either in addition to *d* or instead of it, indicates that students may not well understand the performance differences between the two, or may have not explicitly considered performance in their answers. Of the 180 correct answers, 138 of them mentioned time complexity as a justification for their answer. Only one student who selected *c* as an answer mentioned time complexity, and they appeared to erroneously believe that *c* and *d* would take the same amount of time.

Instructors may want to spend more time discussing performance and time complexity when covering how to choose an existing data structure for use in a larger program. That is, while multiple data structures may each be "correct", implementation choices should also depend on performance considerations given the methods that will be frequently called.

4.7 Shape of a BST

Question 7 was adapted from Karpierz and Wolfman [6] to test students' ability to predict how inserting into a binary search tree would change its shape. In the Karpierz and Wolfman question, there was an additional distractor answer:

This shape with either 1 or 7 at the root and other keys arranged appropriately:

```
   *
 *   *
* * * *
```

Question 7

What shape is a binary search tree that contains the keys 1, 2, 3, 4, 5, 6, and 7? (Keys were not necessarily inserted in that order.)

a. Exactly this shape:

```
      7
     6
    5
   4
  3
 2
1
```

b. Exactly this shape:

```
   4
 2   6
1 3 5 7
```

c. Exactly this shape:

```
1
 2
  3
   4
    5
     6
      7
```

d. There is not enough information to tell.

Briefly explain your reasoning:

We eliminated this distractor as it did not occur in our open-ended interviews for the question. In the pilot of the Karpierz and Wolfman concept inventory, students perform poorly on this question, with 42.3% answering correctly. In contrast, our students performed very well on this question, with 87% correctly choosing *d*.

4.7.1 Default Balanced. Karpierz and Wolfman [6] identify a "default balanced" misconception, where students assume that binary search trees will automatically be balanced. Thirty-six percent of their students demonstrate this misconception by selecting either *b* or their other balanced distractor answer. Only 6% (15) of our students chose *b*. Twelve of these students mentioned the tree being balanced in their answer, suggesting that the default balanced misconception is present in our students, but to a lesser extent. It may be that this misconception is population-dependent, or depends in some way on sequencing or coverage of related course content. Another possible explanation is that there is a priming effect of Question 3, as that question asks in what order elements should be inserted to form a balanced BST.

At the end of a course on Basic Data Structures, students should be able to:

(1) Analyze runtime efficiency of algorithms related to data structure design.
(2) Select appropriate abstract data types for use in a given application.
(3) Compare data structure tradeoffs to select the appropriate implementation for an abstract data type.
(4) Design and modify data structures capable of insertion, deletion, search, and related operations.
(5) Trace through and predict the behavior of algorithms (including code) designed to implement data structure operations.
(6) Identify and remedy flaws in a data structure implementation that may cause its behavior to differ from the intended design.

Figure 1: Course-Level Learning Goals for Basic Data Structures [14]

4.7.2 Root Decides Shape. Among students who correctly chose *d* for the answer, 7 students (2% of total answers) provided explanations specifying that one would have to know the value of the root node, rather than the insertion order. This implies that some students believe the final shape of the tree is entirely dependent on the root, and that the insertion order of the other elements does not matter. Instructors may want to specify that all nodes, not just the root, affect the shape of the final tree.

5 DISCUSSION

In this section we discuss the implications and limitations of our work.

5.1 Question Relevance to Learning Goals

Recent work by Porter et al. [14] identified six course-level learning goals for Basic Data Structures. These goals were reached through a lengthy consensus process involving multiple experts at a variety of institution types, and can be found in Figure 1. We mapped our questions to those learning goals, and found that our questions address four of the six goals. As our questions are aligned to well-accepted goals for a core component of CS2, we hope that the questions will be of use to instructors seeking to provide formative feedback to their students.

A summary of our questions, student difficulties, and applicable learning goals appears in Table 1. Although our questions address the majority of the learning goals for Basic Data Structures, we believe based on ongoing interviews that we have uncovered only a small subset of relevant student difficulties. Further work, using a variety of additional questions and students, is warranted.

5.2 Overmemorization

Many of the difficulties that we identified could be interpreted as students trying to memorize facts about data structures, rather than engaging core data structure concepts. For example, students may memorize that they should use a loop to find the last node in a singly-linked list, and so they do this even when there is a pointer to the tail node. They may have memorized that the root of a BST will never change, and so they claim the root will stay the median value, even though the median changes. They may have memorized that BSTs are used to quickly find an element, and so they overlook that the same performance can be achieved using binary search on an ArrayList.

Overall, this points to the need to augment how data structures are taught. It seems that some students can excel in data structures simply by memorizing a set of facts about each data structure covered. Rather than focusing on recall, instructors should focus on how students can use the facts about a given data structure to predict data structure behavior and evaluate tradeoffs between data structures.

5.3 Incorporating Student Difficulties

Awareness of common student difficulties is an important component of pedagogical content knowledge. We hope that the collection of difficulties identified during our interviews can both confirm and supplement such knowledge that faculty who teach CS2 may have acquired over time through their student interactions. This knowledge has immediate application to the classrooms of all CS2 instructors, because it suggests the need for targeted interventions, of whatever sort are customary for that instructor and class, to address these areas of potential gaps in student mastery. For example, an instructor who uses Peer Instruction in lectures could add a clicker question to target each difficulty. An instructor who uses an online platform for small coding exercises could similarly select or write more-targeted exercises. Durable and broadly-used curricular materials, such as textbooks and video-based online courses, could also be updated to better target these difficulties.

5.4 Threats to Validity

Interview and test subjects were drawn from classes at highly selective, research-focused schools. Although all students enrolled in CS2 at those schools were encouraged to participate, this student population may differ from CS2 students in other programs and types of institutions. Additionally, students who self-select to participate in interviews about CS2 or consent to have their work used in research projects may not reflect the general CS2 population.

To account for this, at-scale validation of future drafts will include a broader range of institutions. This will likely not contribute to further discovery of difficulties, a purpose served by the interviews and open-ended test, but will allow calibration on question difficulty and validation that the identified difficulties are shared by the new student populations.

Because this work was the first time these questions were administered to a large group, there are instances where ambiguous wording in the question led to confusion for students. We discussed these issues earlier in the text where they are relevant. Though we saw no evidence of language-related confusion, our choice to use a pseudocode language could cause students to incorrectly answer a question that they might otherwise have answered correctly had it been presented in their preferred programming language.

Table 1: Summary of Questions, Difficulties, and Applicable Learning Goals

Question	Goal	Data structure	Question Description	Difficulties
1	4	LL	Add to tail	Failing to update tail
				Failing to attach new node
				Unnecessarily looping to find tail
2	3	Many	Fast index access	Binary-searching doubly-linked list
				Searching instead of indexing
				Searching from both ends of a DLL
3	5	BST	Insertion order to balance BST	Starting in the middle, alternating outward
				Starting at the wrong root
				Inserting in random order
4	5	BST	Impact of BST insertions	Adding cannot take $O(N)$ time
				Adding cannot take $O(\log N)$ time
				Root remaining the median
5	3	Many	Generalized array	Binary search trees are always fastest
				Indexing by negative indices
				Ignoring performance
6	2	LL	Data Structure for undo	Ignoring performance
7	5	BST	Shape of a BST	Default balanced [6]
				Root determining shape

5.5 Future Work

This work is part of a long-term research project with the goal to create and validate a programming language-independent concept inventory (CI) for the Basic Data Structures component of CS2. A CI is a standardized test instrument designed to determine whether students correctly grasp core concepts of a topic area [23]. A reliable, valid CI can catalyze research and improvements in pedagogical practice for a subject by providing a common point of reference for measurements of student outcomes. Without an available CI, researchers are forced to cull questions from various sources and independently verify the effectiveness of the test that they construct [24].

Two early steps in the CI development process are the identification of student difficulties and the creation of an open-ended test [23]. Future work involves the creation of a test containing questions in multiple-choice format; validating these questions through additional rounds of interviews; and administering the questions to several large classes for the purposes of statistical analysis.

6 CONCLUSION

In this work, we explore student difficulties about Basic Data Structures, adding to the limited but growing work in this area. In particular, our work explores similar difficulties to those of Karpierz and Wolfman [6], Danielsiek et al [2], and Paul and Vahrenhold [13]. We offer a look at related, and in some cases the same, material, but with a different student population, and discover both similar and different student difficulties.

We hope that these difficulties provide instructors new insight into problems students may have when learning data structures. However, we caution against assuming that these are the only difficulties that students may have. Student difficulties may vary based on student population and the way course content is covered, both in Basic Data Structures and in courses they have previously taken, as the contrast between some of our results and the results of previous work with different student populations shows [2, 6, 13]. Our future work includes extending our measurement of these difficulties to a much broader student population across a variety of institutions and instructors.

This work represents a first step towards quantifying student difficulties about Basic Data Structures. Some instructors may feel that some of these difficulties are obvious (or perhaps outlandish, depending on what they have observed of their own students) and dispute the need for formal work in this area. However, rigorous steps towards eliciting and measuring student difficulties allow instructors to focus their limited time and resources to where students are most likely to struggle. It is known that instructors often fail to anticipate student difficulties or understand the misconceptions that lead students to generate incorrect and "confusing" interpretations of concepts [3]. We urge further exploration of Basic Data Structures difficulties so as to facilitate more productive discussion and learning of this critical material among our students.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of the following collaborators: Meghan Allen, Owen Astrachan, Darci Burdge, Stephanie Chasteen, Maureen Doyle, John Glick, Paul Hilfinger, Kate Sanders, Paramsothy Thananjeyan, and Steve Wolfman. This work was supported in part by NSF award 1505001.

REFERENCES

[1] J. Confrey. Chapter 1: A review of the research on student conceptions in mathematics, science, and programming. *Review of research in education*, 16(1):3–56, 1990.

[2] H. Danielsiek, W. Paul, and J. Vahrenhold. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, pages 21–26, 2012.

[3] M. Guzdial. Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6):1–165, 2015.

[4] S. Hamouda, S. H. Edwards, H. G. Elmongui, J. V. Ernst, and C. A. Shaffer. A basic recursion concept inventory. *Computer Science Education*, 27(2):121–148, 2017.

[5] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pages 107–111, 2010.

[6] K. Karpierz and S. A. Wolfman. Misconceptions and concept inventory questions for binary search trees and hash tables. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pages 109–114, 2014.

[7] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1):57–80, 2011.

[8] S. Madison and J. Gifford. Modular programming. *Journal of Research on Technology in Education*, 34(3):217–229, 2002.

[9] R. McCauley, S. Grissom, S. Fitzgerald, and L. Murphy. Teaching and learning recursive programming: a review of the research literature. *Computer Science Education*, 25(1):37–66, 2015.

[10] R. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy. Recursion vs. iteration: An empirical study of comprehension revisited. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 350–355, 2015.

[11] L. Murphy, S. Fitzgerald, S. Grissom, and R. McCauley. Bug infestation!: A goal-plan analysis of CS2 students' recursive binary tree solutions. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 482–487, 2015.

[12] M. C. Parker, M. Guzdial, and S. Engleman. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proceedings of the 2016 ACM conference on International Computing Education Research*, 2016.

[13] W. Paul and J. Vahrenhold. Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, pages 29–34, 2013.

[14] L. Porter, D. Zingaro, C. Lee, C. Taylor, K. C. Webb, and M. Clancy. Developing course-level learning goals for basic data structures in CS2. In *Proceedings of the 49th ACM technical symposium on Computer Science Education*, pages 858–863, 2018.

[15] Y. Qian and J. Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *Transactions on Computing Education*, 18(1):1:1–1:24, 2017.

[16] N. Ragonis and M. Ben-Ari. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3):203–221, 2005.

[17] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[18] P. M. Sadler, G. Sonnert, H. P. Coyle, N. Cook-Smith, and J. L. Miller. The influence of teachers' knowledge on student learning in middle school physical science classrooms. *American Educational Research Journal*, 50(5):1020–1049, 2013.

[19] O. Seppälä, L. Malmi, and A. Korhonen. Observations on student misconceptions—a case study of the Build-Heap Algorithm. *Computer Science Education*, 16(3):241–255, 2006.

[20] L. S. Shulman. Those who understand: Knowledge growth in teaching. *Educational researcher*, 15(2):4–14, 1986.

[21] J. Sorva. The same but different: Students' understandings of primitive and object variables. In *Proceedings of the 8th Koli Calling International Conference on Computing Education Research*, pages 5–15, 2008.

[22] J. Sorva. Notional machines and introductory programming education. *Transactions on Computing Education*, 13(2), 2013.

[23] C. Taylor, D. Zingaro, L. Porter, K. C. Webb, C. B. Lee, and M. Clancy. Computer science concept inventories: past and future. *Computer Science Education*, 24(4):253–276, 2014.

[24] J. Tenenberg and L. Murphy. Knowing what I know: An investigation of undergraduate knowledge and self-knowledge of data structures. *Computer Science Education*, 15(4):297–315, 2005.

[25] S. Zehra, A. Ramanathan, L. Zhang, and D. Zingaro. Student misconceptions of dynamic programming. In *Proceedings of the 49th ACM technical symposium on Computer Science Education*, pages 556–561, 2018.