Student Performance on the BDSI for Basic Data Structures

KEVIN C. WEBB, Swarthmore College

DANIEL ZINGARO, University of Toronto Mississauga

SOOHYUN NAM LIAO, Princeton University

CYNTHIA TAYLOR, Oberlin College

CYNTHIA LEE, Stanford University

MICHAEL CLANCY, University of California, Berkeley

LEO PORTER, University of California, San Diego

A Concept Inventory (CI) is an assessment to measure student conceptual understanding of a particular topic. This article presents the results of a CI for basic data structures (BDSI) that has been previously shown to have strong evidence for validity. The goal of this work is to help researchers or instructors who administer the BDSI in their own courses to better understand their results. In support of this goal, we discuss our findings for each question of the CI using data gathered from 1,963 students across seven institutions.

CCS Concepts: • **Social and professional topics** → **Computing Education**;

Additional Key Words and Phrases: Concept inventory, assessment, data structures

ACM Reference format:

Kevin C. Webb, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Cynthia Lee, Michael Clancy, and Leo Porter. 2021. Student Performance on the BDSI for Basic Data Structures. *ACM Trans. Comput. Educ.* 22, 1, Article 8 (October 2021), 34 pages.

<https://doi.org/10.1145/3470654>

1 INTRODUCTION

Concept Inventories (CIs) are designed to measure student conceptual knowledge for a particular topic. They are valued by education researchers and instructors, as they provide a common benchmark for student knowledge, and they support instructors looking to diagnose areas of strength (or weakness) in their particular courses. The development process for a CI ensures

This work was supported in part by NSF award 1505001.

Authors' addresses: K. C. Webb, Computer Science Department, Swarthmore College, 500 College Avenue, Swarthmore, PA 19081; email: kwebb@cs.swarthmore.edu; D. Zingaro, University of Toronto Mississauga, 3359 Mississauga Road, Mississauga, ON, L5L 1C6; email: daniel.zingaro@gmail.com; S. N. Liao, Halıcıoğlu Data Science Institute, University of California San Diego, 9500 Gilman Drive MC 0555, La Jolla, CA 92093; email: soohyunl@cs.princeton.edu; C. Taylor, Oberlin College, 10 N Professor St, Oberlin, OH 44074; email: ctaylor@oberlin.edu; C. Lee, Gates Computer Science Building, 353 Jane Stanford Way, Stanford, CA 94305; email: cbl@stanford.edu; M. Clancy, Computer Science Division, University of California, Berkeley CA 94720; email: clancy@cs.berkeley.edu; L. Porter, Department of Computer Science and Engineering University of California San Diego 9500 Gilman Drive MC 0404 La Jolla, CA 92093-0404; email: Lporter@cs.ucsd.edu.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

© 2021 Copyright held by the owner/author(s).

1946-6226/2021/10-ART8 \$15.00

<https://doi.org/10.1145/3470654>

that various measures of validity are used to ensure that the questions on the CI are meaningful to instructors, that they are properly interpreted by students, and that they measure student understanding [1].

In computer science, there have been few instruments created for use in such a manner (for example, the Second CS1 instrument [23] and the Digital Logic Concept Inventory [14]). Our previous work complemented the field by adding the **Basic Data Structures Inventory (BDSI)**, a CI that can be used in conjunction with a CS2 course (or the corresponding course that teaches basic data structures) [27].

Our previous work first reported on the establishing of learning goals [26] and common difficulties with basic data structures [40]. We then reported on our extensive development and validation process in which we reported on our claims of validity and evidence we gathered to support those claims [27].

What remains unanswered for instructors and researchers using the instrument is how to interpret their findings. If a large section of students answered a particular distractor for a given question, how often do students at similar schools select this distractor and what might have led students to select that distractor? In this work, we seek to provide this context to users of the instrument by reporting on the frequency of student responses to each question and use data from our student interviews to help with interpreting the source of particular distractors. The data used to provide this context includes data from 99 interviews at seven institutions as well as data from large-scale pilots of the instrument to 1,963 students. For each question, we provide the full content of the question, a box plot that characterizes the frequency of student answer selections, and our interpretation of what the results imply about student understanding of the underlying concepts. By providing our analysis of these results, we hope that CS instructors, CS education researchers, or anyone else interested in utilizing the BDSI will feel more confident in adopting it and be better able to understand its results.

2 BACKGROUND

A CI is a multiple-choice assessment designed to measure student understanding of the core concepts of a course. The goal of a CI is not to summatively assess a particular student, but rather to create a standardized instrument that can be used to compare across instructors, courses, and pedagogies. Adams and Wieman [1] list several requirements for a useful CI:

- It must be easy to administer and grade without any training in the context of a normal course schedule.
- The instrument must test “expert thinking” of obvious value to teachers.
- It must be demonstrated that it measures what it claims (evidence of validity).

2.1 CIs in STEM and Computer Science

CIs arose from observations by physics educators in the mid-1980s that conventional instruction in Newtonian physics was not working. It appeared that everyday beliefs about motion and force were incompatible with Newtonian concepts; moreover, conventional instruction produced little change in these beliefs [15]. This led to the design of the **Force Concept Inventory (FCI)** [15] to measure these student misconceptions. The FCI spurred the adoption of Peer Instruction and other active learning techniques within physics education [4]. A later study [12] administered the FCI to 6,542 students in 62 different courses and found that traditional courses had lower learning gains than those using active learning. The impact of the FCI on the physics community led to the design of other CIs both in physics [3, 21] and in numerous other scientific areas including chemistry [19], genetics [32], statistics [34], calculus [8], geoscience [20], oceanography [2], nursing [29], astronomy [30, 39], and biology [7].

Within computer science, there have been fewer CIs developed [36]. Tew and Guzdial developed the FCS1, an assessment for CS1 that uses a pseudocode to measure students' basic programming knowledge [37]. Due to the unavailability of the FCS1, the **Second CS1 Assessment (SCS1)**, an isomorphic version of the FCS1, was developed and made available to instructors [23]. CIs have also been developed for digital logic [14] and computational thinking for elementary-level [24] and middle-grades [28] students. Preliminary work towards CIs is available for a number of other computer science topics, including algorithms [6, 9], cybersecurity [31], recursion [13], advanced data structures [18], architecture [25], and operating systems [38]. For additional evaluation instruments in CS, see Reference [5] and for a broad survey of CI availability in computer science, see Reference [36].

2.2 BDSI Prior Work

This article represents the culmination of a multi-year project to develop, validate, and understand the BDSI. Throughout the project, we published several other papers along the way. To provide context, we briefly summarize our prior work below. Note that while we summarize some of this prior work in other sections (e.g., BDSI development in Section 3 and evidence of statistical validation in Section 4), the prior work contains many more details. We encourage interested readers to refer to these papers for additional information about the BDSI:

- The first publication [26] describes the development of the BDSI's learning goals. It recounts how, with the help of a panel of CS2 experts, we determined a set of topics and question coverage goals for the BDSI.
- The next paper [40] identifies common student misunderstandings in basic data structures. It represents a precursor to the present article with a substantially smaller student sample size. It reports only a subset of the concepts on the BDSI and analyzes earlier, less-refined versions of the questions.
- The third publication [27] details our work toward validating the BDSI, and the present article analyzes the dataset presented in this validation work. The paper evaluates several validity claims for the BDSI, including topics such as *the questions are meaningful to instructors*, *students properly interpret questions*, and *the CI is internally consistent*. It analyzes student think-aloud interviews and a large dataset of several BDSI runs using the methods of Classical Test Theory and Item Response Theory. Ultimately, we believe that the results illustrate strong evidence of statistical validation for the BDSI.
- Finally, the most recent paper [35] characterizes the full development process for the BDSI. It chronicles the question development and refinement process with the intent of providing a CI development roadmap. We hope that the lessons it offers will lower the barrier to entry for future CI development in computer science.

The present article complements this prior work by analyzing a large set of student responses that we collected during the BDSI development process. Rather than development processes [35] or statistical validation [27], it primarily focuses on understanding the student learning implications of student selections on the CI. We expect that this article will be especially helpful in interpreting results for CS2 instructors or CS researchers who adopt the BDSI.

3 BDSI DEVELOPMENT AND ADMINISTRATION

Our findings are based on data collected during and after the development process of the BDSI. As such, we detail the completed development process and these sources of these data here. The BDSI development team consists of faculty with teaching backgrounds from both large research-intensive universities and small liberal arts colleges. We met frequently during the

development of the BDSI, authored questions, conducted (and analyzed) interview data from students, and analyzed results from pilot runs of the instrument. Throughout the CI development process, we consulted with an expert panel: CS2 instructors from a broad set of North American institutions including research-intensive universities, public regional universities, liberal-arts colleges, and community colleges. The expert panel provided feedback periodically during the development process and were extremely helpful in contributing their institutional context to the work. While the full details of our four-year CI development process are documented in prior studies [27, 35], it can be briefly summarized as:

- (1) Establish topics that are important to teachers.
- (2) Develop a set of open-ended questions on these topics.
- (3) Interview students and offer open-ended practice exams to discover misconceptions to convert questions into a forced-choice or select-all format.
- (4) Conduct validation interviews and question trials at a variety of institutions.
- (5) Statistically validate the CI.

We refer to step 3 as the *interview phase*, and it encompasses one-on-one think-aloud interviews with students on broad questions aimed to elicit thinking about basic data structures. Based on the interview results, we developed an initial set of CI candidate questions. As our goal was to gain a nuanced understanding of student conceptual knowledge of data structures, we opted to include select-all questions to capture cases where students frequently selected both a correct answer and one or more distractor answers, despite potential complications for scoring.

For step 4, the *refinement phase*, we iteratively refined the candidate questions via both individual student interviews and trial runs of the instrument. We organized the trials as “final exam review sessions” to be held in conjunction with the end of the term for courses that teach basic data structures. These sessions were led by course instructors, TAs, or other faculty and involved students individually taking a one-hour “practice exam” that was the current draft of the BDSI. During this phase, we asked students to explain their answers for early versions of multiple-choice questions, and these responses proved integral for finding patterns of erroneous student reasoning. Once complete, the exam booklets were collected by the staff administering the exam, and the staff then went over the answers to the BDSI as well as other course topics not addressed by the CI. Students who attended the final exam review session were allowed to opt-in or opt-out from participating per our approved Human Subject Board protocol (if they opted out, then they could complete the practice exam, but we did not use their data).

Finally, in step 5, the *validation phase*, we interviewed students to ensure that they were interpreting our questions correctly. We then administered the finalized CI via similar “final exam review sessions” to collect large-scale response data. The quantitative results presented in Section 4, in which we present how frequently students select each answer choice, come from these validation runs. The combination of the quantitative interviews and a statistical analysis of the finalized CI response data demonstrates strong evidence for BDSI validity [27].

For multiple-choice and select-all style questions, we did not randomize the order of the answer choices. The BDSI has a variety of question types (e.g., data structure analysis and code fill-in-the-blank), and though some types of questions may have benefited from randomization, others would not [10].

Steps 3–5 are particularly important for the question analysis we provide in Section 4, as each phase uniquely contributed to our understanding of common student misconceptions. Throughout that analysis, we describe whether our insights into patterns of student thinking came from an interview or an open-ended practice exam.

Table 1. BDSI Learning Goals (Adopted from Reference [26])

Goal#	Learning Goal
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.

3.1 Learning Goals

The final version of the BDSI contains 13 questions, which cover a variety of data structures topics, including lists, trees, stacks, and sets, with a heavy emphasis on lists and trees. While all of the questions are multiple-choice, they are presented in two formats—some ask that students select only one option, whereas others ask students to select all the options that apply. Each question has a primary learning goal according to the goals described in Reference [26]; the relevant table is reproduced in Table 1.

3.2 Assumed Student Background

We assume students have already taken a course in basic data structures when they are given the instrument. In developing the questions for the BDSI, we faced a challenge due to variations in course content, particularly surrounding the choice of programming language. To make the BDSI accessible to as many courses as possible, we aimed to develop questions that are agnostic to the choice of programming language. Thus, we augmented the pseudocode language originally used in the FCS1 [37] and SCS1 [23] with minor modifications to include objects. When presenting the BDSI to students, we provide a short reference packet that encompasses:

- Question answering instructions that highlight the differences between select-one and select-all question styles.
- Brief definitions for the interfaces and data structures used in the BDSI. We provide these definitions not to introduce students to these concepts (e.g., for a pre-test), but to help account for students who may have learned different terminology depending on their programming language (e.g., array lists in Java, lists in Python, and vectors in C++). For reference, we reproduce the BDSI's definitions, instructions, and pseudocode reference in Appendix A.
- A concise summary of the BDSI's pseudocode language with several examples.

Regardless of the language in which students learned the material (Java, Python, or C++), students rarely identified programming language differences, the pseudocode language, or data structure terminology as sources of confusion in the course of our validation work. While our pseudocode was tested with students using a variety of imperative coding languages, we do not address basic data structures in functional paradigms such as Scheme, whose teaching and learning may be different.

4 RESULTS AND ANALYSIS

In this section, we present each question, illustrate the selection rate of each answer choice, and offer our analysis of the results. The analysis draws from observations of student reasoning in the development phases described in Section 3. When possible, we offer our conjecture as to how faculty might work to remedy the identified misconception for students. To aid instructors in

Table 2. Summary of the Institutions That Participated in Data Collection

	I-A	I-B	I-C	I-D	I-E	I-F	I-G
School Type	CC	PUI	LAI	RIU	LAI	RIU	RIU
School Size	M	L	S	L	S	L	L
# Participants	16	17	12/57	14/11	63	82/408/840	110/164/169

Some institutions participated multiple times, as reflected by the number of participants separated by slashes. The type of each institution is listed as one of community college (CC), primarily undergraduate institution (PUI), liberal arts institution (LAI) or research intensive university (RIU). We define *small* as less than 5,000 students, *medium* as 5,000 to 10,000 students, and *large* as greater than 10,000 students.

Table 3. Summary of BDSI's Questions, Their Learning Goals, the Data Structures They Cover, and Their Median and Interquartile Correctness Rates

	Learning Goal	Structures	25 th	Median	75 th	Question Description
q1	4	LL	49%	68%	78%	Correctly add an element to the tail of a linked list.
q2	1	LL	9%	14%	30%	Compare list performance with and without a tail pointer.
q3	1	LL	73%	76%	81%	Without looking at the code, experimentally determine how an ADT was implemented.
q4	1	LL	75%	79%	86%	Analyze performance of implementing an interface with a linked list.
q5	5	BST	24%	33%	44%	Identify BST insertion orders that lead to balanced trees.
q6	5	LL	51%	61%	74%	Explain the behavior of a function that uses a linked list.
q7	4	BT	32%	34%	46%	Write a function to compute a property of a binary tree.
q8	3	Many	55%	64%	76%	Compare the performance of satisfying an abstract interface with several structures.
q9	5	BST	10%	21%	33%	Analyze performance and effect on tree properties of adding elements to a BST.
q10	2	LL, Stack	47%	54%	60%	Evaluate the performance of implementing "undo" functionality with a linked list.
q11	6	BST	68%	71%	74%	Determine appropriate cases for testing the correctness of a BST implementation.
q12	4	BT	42%	48%	50%	Compute the width of a binary tree.
q13	2	List, Set	78%	81%	89%	Compare solving a problem with a list interface versus a set interface.

comparing their findings to those of others at comparable institutions, we provide student response rates per answer choice. These additional results add nuance to interpreting the results, particularly for select-all questions, which may be masked when only examining question correctness.

In the interest of space, we only comment on results that we believe are notable, and our analysis ignores infrequently-chosen answers if they are not indicative of an interesting result. We note that even the most infrequently chosen answers *were* observed in student responses during earlier CI development phases (e.g., open-ended interviews).

Table 2 describes the institutions from which we collected data. Due to the variance in sample sizes between large R1 institutions and small liberal-arts colleges, our question analysis presents box plots and the median correctness rate *across each independent run of the instrument*, regardless of the number of students who participated. Reporting the median by independent run prevents the larger datasets from drowning out the smaller ones and characterizes differences in performance between institutions, which is one of the goals of a concept inventory. Table 3 summarizes the questions, their learning goals, and their correctness rates. The remainder of this section presents and analyzes each question in detail. We analyze the dataset that we collected as part of the BDSI validation process. For validation details (including CTT and IRT results), see Reference [27].

Fig. 1. Box plot results for Q1 and Q2. Correct answer choices are marked with green rectangles on each chart. Q2 asked students to mark all correct answer choices, although there was a single correct choice.

Preamble for Questions 1 and 2

Questions 1 and 2 refer to the `MyLinkedList` and `MyListNode` classes defined below.

```

CLASS MyLinkedList
  DEFINE initialize MyLinkedList ()
    head = nil // MyListNode head refers to nothing
    tail = nil // MyListNode tail refers to nothing
  ENNDEF
  DEFINE boolean isEmpty()
    RETURN head == nil
  ENNDEF
ENDCLASS

CLASS MyListNode
  DEFINE initialize MyListNode (node_item)
    item = node_item // assign MyListNode item
    next = nil // MyListNode next refers to nothing
  ENNDEF
ENDCLASS

```

4.1 Question 1: Linked List Append

Question 1 examines students' ability to author code to complete a linked list method when the singly linked list is modified to include a tail pointer. Students are expected to use the tail to insert a new element and then update the tail. This question addresses learning goal 4, "Design and modify data structures capable of insertion, deletion, search, and related operations." During interviews, the pseudocode language was not a barrier to students, although they sometimes looked to the

Question 1

The `MyLinkedList` and `MyListNode` classes defined on the prior page implement a singly linked list that maintains an internal reference named `tail`, which refers to the last node of the list.

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

```
DEFINE addAtEnd(e)
    IF tail == nil THEN
        head = tail = new MyListNode(e)
    ELSE
        // MISSING CODE
    ENDIF
ENDDEF
```

Select **ONE** of the following code segments to complete the `addAtEnd` method:

- A. `tail.next = e`
`tail = e`
- B. `temp = new MyListNode(e)`
`tail.next = temp`
- C. `temp = new MyListNode(e)`
`tail = temp`
- D. `tail.next = new MyListNode(e)`
`tail = tail.next`
- E. `temp = head`
`WHILE temp.next != nil DO`
 `temp = temp.next`
`ENDWHILE`
`temp.next = new MyListNode(e)`

question preamble for an example of how to create a new node. The correct response is option D. Figure 1(a) provides the frequency of student responses for each answer choice.

4.1.1 Correct Choice. Option D correctly adds a new element to the end of the list by using the tail reference, then updates the tail. The majority of students (median of 68%) answer this question correctly.

4.1.2 Distractor E: Using a Loop (Ignoring the Tail Pointer). Option E is the most popular incorrect response with a median response of 13%. Although the loop is correct for appending the new element, it fails to update the tail reference and is hence incorrect. This response was observed during an open-ended practice exam in which students could provide any answer. Option E surprisingly ignores the tail reference, which is provided in the question and appears in other option choices. Unfortunately, this approach was not observed during interviews so our ability to interpret students' thinking is limited. However, we suspect this is caused by students having memorized (or becoming familiar with) the looping approach to traversing a linked list and recognizing that this looping approach is often what is needed when working with linked lists. Potential

remedies might include emphasizing data structure algorithm development beyond examples from class and providing more problems to students that have them author LinkedList methods that do not require looping through the list.

4.1.3 Distractor B: Correct Append but Fails to Update Tail. Students who selected option B correctly create a new node and add it to the end of the list, but they miss the update of the tail. This was observed during interviews and during open-ended practice exams. In general, students who made this mistake either forgot that they needed to update the tail themselves or believed that they were not responsible for updating the tail. We suspect instruction that emphasizes data structure invariants (e.g., the tail reference always points to the last element in the list) and the importance of every method maintaining those invariants might help with mitigating this error.

4.1.4 Distractor C: Correct Update of the Tail but Fails to Append. Students who selected option C correctly create a new node and set the tail to point to that new node. However, this option fails to update the original `tail.next` so the list does not include the new element and the tail now refers to an element not in the list. This response was observed during interviews and during open-ended practice exams. We believe the source of this mistake is similar to Distractor B in that they again fail to maintain all invariants of the list.

4.1.5 Distractor A: Misunderstanding of Objects. Option A is the least common response. The code correctly recognizes the need to update `tail.next` and the tail reference, but it does not create a `MyListNode` containing the element. In a strongly typed language, this code would not compile. Again, this response was observed both during interviews and during open-ended practice exams. During free-response interviews, we suspect this might have occurred due to misunderstanding the pseudocode or through carelessness. Upon adding multiple-choice options that all correctly create a new node, we did not see this response in our closed-form interviews. As such, we believe students selecting this option misunderstand types and objects. Future work could see if this is more common among students who take the course in one programming language over another. Potential remedies would be reviewing differences in object types and/or more emphasis on Object-oriented Programming techniques.

4.2 Question 2: Advantages of a Tail Pointer

Question 2 asks students to determine the runtime implications of adding a tail pointer to the singly linked list from the previous question. Students are expected to see that the operation to add an element at the end, as coded in the previous question, would be fastest and that no other operations would be faster. This question addresses learning goal 1, “Analyze runtime efficiency of algorithms related to data structure design.” The correct response is option D. Figure 1(b) provides the frequency of student responses for each answer choice. During interviews, the incorrect response B was often initially selected by students, but as they talked through their answer as part of the think-aloud process, they would catch themselves and recognize that although they could access the last element using the tail, they would be unable to point the tail to the second-to-last element without iterating through the entire list. At scale, however, a large number of students appear to make this mistake, causing this question to be consistently the hardest question on the BDSI.

4.2.1 Correct Choice. Option D correctly identifies that adding a new element to the end of the list is faster using the tail reference, as there is no longer a need to iterate through the list. The majority of students (median of 94%) correctly select this option. However, as this is a “select all that apply” question, only a median of 14% correctly select this as the only answer.

4.2.2 Distractor B: Removal of the Last Element. As mentioned above, Distractor B is selected by a majority of students (median of 70%) and is incorrect. We believe that students selecting

Question 2

In the `MyLinkedList` class in the previous exercise, we have a reference to both the head and the tail of the list. Compare this implementation against a singly linked list that does not have a reference to the tail of the list (i.e., it only contains a reference to the head of the list).

For which of the following linked list operations would a singly linked list with a tail reference have better execution time (i.e., achieve a faster worst-case time complexity) relative to a singly linked list without a tail reference?

Select **ALL** that apply:

- A. return True if the linked list contains a given element
- B. remove the last element from the linked list
- C. add a given element to the beginning of the linked list
- D. add a given element to the end of the linked list

this option are not carefully thinking through the detailed steps of how to perform the removal including how to: (1) update the second-to-last node to point to `nil` and (2) update the tail to point to the second-to-last node (or `nil` if the list is empty after the removal). We suspect that teaching students to create short pseudocode of solutions before reasoning about their runtime might help reduce this error.

4.2.3 Distractor C: Faster Addition of Elements to the Start. Adding an element to the beginning of the list is already fast for a singly linked list. This mistake did not appear in interviews but did appear during the open-ended run. Only a small group of students (median of 7%) select this option, but they may have considerable confusion about when linked-lists, without a tail pointer, are efficient.

4.3 Question 3: Runtime of LinkedList Operations

Question 3 asks students to design an experiment to determine if a list implementation is singly or doubly linked. To correctly answer the question, they must understand how the runtime of operations on the two list implementations will differ. This question addresses learning goal 1, “Analyze runtime efficiency of algorithms related to data structure design.” The correct response is option B. The distribution of student answers for this question is shown in Figure 2(a).

4.3.1 Correct Choice. The correct answer is choice B. This option requires students to understand that adding a node to the end of a singly linked list will take longer than adding to the start of the list, since to add at the end one must traverse the entire list, while adding at the start only requires following a single pointer and then updating several pointers. In a doubly linked list with a tail pointer, neither of these operations will require traversing the list, so they will both take approximately the same amount of time. The median student correctness for this question was 75%.

4.3.2 Distractor C: *addAtEnd* vs. *removeAtEnd*. This distractor compares the timing of two linked list operations, just like the correct answer, but the operations it compares are adding a node at the end of the list and removing a node from the end of the list. These two operations will both involve traversing the entire list in the singly linked list and avoid traversing it in the doubly linked list. Thus, both operations will be $O(n)$ in the singly linked list and $O(1)$ in the doubly

Question 3

You are given a new `LinkedList` class that implements the `List` interface, but you do not know whether the `LinkedList` is implemented using a singly linked list or a doubly linked list and do not have access to the source code. In this question, the singly linked list has a reference to its head, and each node in the list has a reference to the next node. The doubly linked list has references to the head and tail, and each node in the list has references to the next and previous node.

You only have access to the `LinkedList` through the `List` interface methods below:

<code>size()</code>	return the number of elements in the <code>List</code>
<code>contains(e)</code>	return <code>True</code> if <code>e</code> is in the <code>List</code> , <code>False</code> if not
<code>insert(e, ind)</code>	add <code>e</code> to the <code>List</code> at index <code>ind</code>
<code>get(ind)</code>	returns, without removing, the element at index <code>ind</code>
<code>removeAtStart()</code>	remove and return the first element from this <code>List</code>
<code>removeAtEnd()</code>	remove and return the last element from this <code>List</code>
<code>addAtStart(e)</code>	add <code>e</code> to the beginning of this <code>List</code>
<code>addAtEnd(e)</code>	add <code>e</code> to the end of this <code>List</code>

Indicate which of the following experiments would determine whether the unknown implementation is a singly linked list or a doubly linked list. If multiple approaches work, select the best approach.

Select **ONE**:

- A. Author a singly linked list class of your own and test the timing of it against the timing of the unknown `List` for all `List` methods. If the timing between your singly linked list and the unknown `LinkedList` is exactly the same for all methods, the unknown `LinkedList` implementation is a singly linked list, otherwise it is a doubly linked list.
- B. Execute `N` `addAtEnd` operations followed by `N` `addAtStart` operations; if the `addAtEnd` operations take much longer than the `addAtStart` operations, we have a singly linked list, otherwise it is a doubly linked list.
- C. Execute `N` `addAtEnd` operations followed by `N` `removeAtEnd` operations; if the `removeAtEnd` operations take much longer than the `addAtEnd` operations, we have a singly linked list, otherwise it is a doubly linked list.
- D. Create two instances of the `LinkedList` and test the timing of the first against the timing of the second `LinkedList` for all `List` methods. If the timings between the first and second instances are close for all methods, the unknown `LinkedList` implementation is a singly linked list, otherwise it is a doubly linked list.
- E. None of the above experiments would be able to answer this question. You would need to be able to examine the code to determine if the implementation uses a previous reference.

linked list. This means that comparing the times of these two operations on a single implementation will not reveal any differences between them. Students choosing this answer may think that they should test these two operations because they have a different time complexity in each implementation. This was the most popular distractor answer, with the median response rate of 11%.

4.3.3 Distractor E: Source Code Required. This distractor claims that it is impossible to tell which list implementation is being used without looking at the source code. Its median response

Fig. 2. Box plot results for Q3 and Q4. Correct answer choices are marked with green rectangles on each chart.

rate is 5%, making it the second-most-popular distractor. During interviews on the free-response version of this question, we frequently had students answer that they would look at the source code to determine the implementation, despite our attempts to make clear that this was a situation in which you would not have access to the source code. However, when we converted the question to multiple choice, validation interviews made it clear that there was no way to include looking at the source code as a potential answer without the students interpreting it as meaning they had access to the source code. Instructors should be aware of the student misconception that source code will always be available.

4.4 Question 4: LinkedList Runtime

This question tests students knowledge of how to implement data storage in a LinkedList and the runtime of related LinkedList operations. It evaluates learning goal 1, “Analyze runtime efficiency of algorithms related to data structure design.” Students are asked to store key-value pairs in the LinkedList, where keys are integers, including negative integers. Pairs will be inserted once and accessed many times. The question asks for the best possible time complexity for a *get* method that takes in a key and returns a value. The correct response is option C. The breakdown of student responses for this question is shown in Figure 2(b).

This question was originally part of a longer question, in which we also asked students to implement the same data store, but using an ArrayList. We expected the correct answer to be $\log N$ —students would insert pairs in a sorted order and then perform binary search on the list. However, we found that both very high- and very-low-performing students were selecting the distractor answer: constant time. In interviews, we found that high-performing students devised a solution in which they found the most negative integer key and then biased all of the keys by that amount to insert them into a potentially very large array, allowing them to be found in $O(1)$ time. Low-performing students simply did not take into account the fact that they could not insert based on integer value if the integers were negative. To avoid this problem with the question, we used only the part of the question that focused on LinkedLists in the final CI.

Question 4

Consider the design of a new interface that will allow for storage and access to pairs of integer keys and values. The keys may be any collection of integers and not just a contiguous range (0,1,2,...) of integers. For example, one might set up an instance of the interface named `data` to have three keys—say, -5, 42, and -71; one may then access `data.get(-5)`, `data.get(42)`, and `data.get(-71)` to retrieve their corresponding values. An attempt to access the new data structure using any other key would be illegal.

You have control over how you implement the interface (e.g., which data structure is used and the order that elements are inserted). Assume that the set of keys is defined before the program is executed and does not change during the program run. N is the number of keys.

Consider implementing the new data type with a `LinkedList` of key/value pairs, ordered by key. Assuming you use the `LinkedList` as efficiently as possible, the `get` method should run in time proportional to:

Select ONE:

- A. N^2
- B. $N \log N$
- C. N
- D. $\log N$
- E. constant time

4.4.1 Correct Choice. The correct answer is option C, time complexity of N . Since this is a `LinkedList`, the method will need to traverse the `LinkedList` to find the key, an operation that has a time complexity of N , the length of the `LinkedList`. A median of 79% of students correctly chose this answer.

4.4.2 Distractor D: $\log N$. This was the second-most commonly chosen answer, with a median value of 7% of students selecting it. In this answer, students believe they can find a value in a `LinkedList` in time $\log N$. This answer may indicate that students think they can perform a binary search on the `LinkedList`.

4.5 Question 5: Binary Search Trees

Question 5, which is based on work by Karpierz and Wolfman [18], 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. This question addresses learning goal 5, "Trace through and predict the behavior of algorithms (including code) designed to implement data structure operations." Options B and D are both correct answers to this question. Figure 3(a) illustrates how frequently students selected each answer choice.

4.5.1 Correct Choices. The sequence in Option B, with a median selection rate of 80%, starts at the root and then immediately begins building both of the root's subtrees. Fewer students (median of 53%) correctly selected D, which initially spends more time constructing the root's right subtree. This resulted in a median correctness of only 33% on the problem, making it one of the harder problems on the BDSI. We suspect that option B more closely matches the way in which balanced

Fig. 3. Box plot results for Q5 and Q6. Correct answer choices are marked with green rectangles on each chart. Q5 asked students to mark all correct answer choices.

Question 5

Suppose that we wanted to insert the numbers 1 to 1,023 into a standard `BinarySearchTree`. Which of the following could be the **initial five values inserted**, in the order given, to produce a **fully balanced tree** after all 1,023 values are inserted?

Select **ALL** that apply:

- A. 1, 2, 3, 4, 5
- B. 512, 256, 768, 640, 384
- C. 512, 511, 513, 510, 514
- D. 512, 768, 640, 896, 256
- E. 1023, 512, 768, 256, 640
- F. Inserting values randomly will always produce a fully balanced tree.

BSTs are typically taught in the classroom, and students more easily recognized the pattern. We recommend that instructors demonstrate that multiple insertion orders might ultimately produce the same BST.

4.5.2 Distractor C: Starting in the Middle, Alternating Outwards. Option C was easily the most popular distractor choice (median 35%). In selecting C, students apply the correct logic to identify the root node, but then abandoned that logic for subsequent nodes under the root. That is, they identified that 512 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 512, the sequence alternates inserting values that were one larger or smaller than what was inserted previously.

Question 6

Here is a possible method for a `SinglyLinkedList` class. Assume that `head` is a variable in the `SinglyLinkedList` class that refers to the first node in the list.

```
DEFINE mystery(value)
    current = head
    temp = nil
    WHILE current != nil AND current.item != value DO
        temp = current
        current = current.next
    ENDWHILE
    RETURN temp
ENDDF
```

Which of the following best explains the “purpose” of the `mystery` method? (That is, what is the overall goal of the `mystery` method?)

Select **ONE**:

- A. It returns the node before the one containing `value`, or it returns `nil` if `value` is in the head, or it returns the last node of the list if `value` is not found.
- B. It returns the node containing `value`, or it returns the last node of the list if `value` is not found.
- C. It returns the node containing `value`, or it returns `nil` if `value` is not found.
- D. It returns the node before the one containing `value`, or it returns `nil` if `value` is in the head, or it returns `nil` if `value` is not found.

Students selected this distractor in interviews and during the open-ended practice exams. They seem to 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. Concerningly, they do not consider applying the same criterion recursively, despite many properties of BSTs often being defined recursively.

4.5.3 Distractor F: Randomness. A median of 7% of the responses asserted that inserting values randomly into a BST will always produce a balanced tree. 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. From interviews and the open-ended practice exams, it seemed as though students had learned that the average case for BSTs is $O(\log n)$ and that random insertion can often provide this average case. Some students may also have been confused about the definition of a “fully balanced tree.”

4.6 Question 6: Linked List Traversal

Question 6 tests students’ ability to trace a linked list traversal and reason about its behavior. The function to analyze, `mystery`, uses a `while` loop to search for a provided key without running off the end of the linked list (by testing that `current != nil`). This question addresses learning goal 5, “Trace through and predict the behavior of algorithms (including code) designed to implement data structure operations.” The correct response is option A. Figure 3(b) provides the frequency of student responses for each answer choice.

An important insight is that there are two temporary variables that step through the linked list during the traversal, with the variable `temp` always one step (or link) behind the variable `current`.

This means that when the function returns `temp`, it returns the node *before* the node containing the desired key (or `nil` if the desired key is in the head of the list).

We recognize that, for this question (and others), student errors could be caused by misunderstandings related to general programming (e.g., the challenges of loops [16, 33]). In these subsections, we speak to the data structures errors we were able to discern from interviews in the CI refinement and validation phases.

4.6.1 Correct Choice. Option A correctly describes the function's behavior in each of three important cases: when the key is found in the first node of the linked list, when the key is found elsewhere in the linked list, and when the key is not found. The remaining distractors each describe an incorrect behavior for one or more of these cases. A median of 61% of students correctly chose this answer.

4.6.2 Distractors B and C: Misunderstanding What Is Returned When the Key Is Found. Distractors B and C both resemble the behavior of returning the current variable when the key is found, ignoring the role of the trailing `temp`. This results in believing that the function returns the node containing the key, not the one before it. Distractors B and C differ in what students believe happens if the key is not found, which depends on whether they correctly trace the ending condition of the `WHILE` loop. A median of 14% of students chose distractor B, and 12% chose distractor C.

4.6.3 Distractor D: Misunderstanding What Is Returned in Special Cases. Distractor D reflects a correct understanding of the main role of the `temp` variable and how it causes the function to return the node before the one containing key, but incorrectly analyzes what happens in the edge case when the key is not found. This distractor was chosen by a median of 10% of students.

4.7 Question 7: Recursion on a Binary Tree

Question 7 asks students to reason about recursive operations on a binary search tree. This corresponds to learning goal 4, "Design and modify data structures capable of insertion, deletion, search, and related operations." Students are expected to both know to use recursion and be able to correctly implement recursive functions. The question asks them to correctly select the function that returns the sum of the values of all of the leaf nodes of a binary tree. They are given the class definition for the nodes of the binary tree and told that their function will be called on the root node of the tree. Although this is a "select all" style question, it has only one correct answer: Option B.

4.7.1 Correct Choice. In this answer, the function first checks if the node is `nil`, and if so, returns zero. If both children of the node are `nil`, then the tree is a leaf node, and so the value of the node is returned. Otherwise, the function returns the sum of the recursive call of the function on both children of the node. The majority of students (median correctness of 77%) recognized that this answer was correct; however, most of them failed to recognize it was the only correct answer, with a median of 44% correctly choosing *only* this answer.

4.7.2 Distractor A: One-sided Recursion. In this distractor, the function first returns the node's value if the child is a leaf node. Next, it checks if a left child exists, and if so, returns the value of the current node plus a recursive call to the left. Last, the function checks if a right child exists, and if so returns the current node's value plus the recursive call to the right. This distractor is wrong in two ways: It both includes the values of non-leaf nodes in the sum, and it recurses only to the left side if a node has two children. This was our least popular distractor answer, with a median of only 13% of students choosing it. This answer was more popular as an answer in the open-ended practice exam phase than when the question was converted to multiple choice. It may

be that while only recursing to one side is an easy mistake for students to make while writing code, it is easy for them to recognize the mistake when they can also see correct code.

4.7.3 Distractor C: No Recursion. This distractor answer uses a while loop to iterate over nodes in the binary tree, summing the values of the leaf nodes. However, it always travels down to the left child of a node (unless the node does not have a left child), leaving most of the tree unexplored. This was our second-most-popular distractor answer, with a median of 18% of students choosing it. To address this common error, instructors may wish to stress the recursive nature of data structures like BSTs.

Question 7

Here is a class for a node in a BinaryTree.

```
CLASS TreeNode
  DEFINE initialize TreeNode(new_item)
    item = new_item // assign TreeNode item
    left = nil // TreeNode left refers to nothing
    right = nil // TreeNode right refers to nothing
  ENDDF
ENDCLASS
```

Our goal is to write a function that returns the sum of the values of all of the leaf nodes (i.e. nodes with no children). For example:

This tree would return 10.

This tree would return 12.

Our function, `sum_leaves`, will be passed the root node of the tree.

Select **ALL** that apply:

- A. `DEFINE sum_leaves(node)`
 `IF node.left == nil AND node.right == nil THEN`
 `RETURN node.item`
 `ENDIF`
 `IF node.left != nil THEN`
 `RETURN node.item + sum_leaves(node.left)`
 `ENDIF`
 `IF node.right != nil THEN`
 `RETURN node.item + sum_leaves(node.right)`
 `ENDIF`
 `ENDDF`
- B. `DEFINE sum_leaves(node)`
 `IF node == nil THEN`
 `RETURN 0`
 `ELSE IF node.left == nil AND node.right == nil THEN`
 `RETURN node.item`
 `ELSE`
 `RETURN sum_leaves(node.left) + sum_leaves(node.right)`
 `ENDIF`
 `ENDDF`

Question 7 (continued)

```

C. DEFINE sum_leaves(node)
    value = 0
    WHILE node != nil DO
        IF node.left == nil AND node.right == nil THEN
            value = value + node.item
        ENDIF
        IF node.left != nil THEN
            node = node.left
        ELSE IF node.right != nil THEN
            node = node.right
        ENDIF
    ENDWHILE
    RETURN value
ENDDDEF

D. DEFINE sum_leaves(node)
    value = 0
    IF node.left == nil AND node.right == nil THEN
        value = value + node.item
    ENDIF
    IF node.left != nil THEN
        sum_leaves(node.left)
    ENDIF
    IF node.right != nil THEN
        sum_leaves(node.right)
    ENDIF
    RETURN value
ENDDDEF

```

4.7.4 Distractor D: Not Returning the Recursive Call. This distractor defines a new variable and sets it to 0 at the beginning of the function. If the node it is called on is a leaf, then it adds the value of the node to this accumulator variable, and it returns the variable at the end of the function. It also recursively calls itself on the left and right children of the node, but does not return the results of these recursive calls or add them to the variable. As a result, it will only return the value of the first leaf node it finds, not the sum of all the leaf nodes. This was the most commonly chosen distractor answer. A median of 40% of students chose this answer. Similar student answers have been documented in previous studies of recursion, and the misconception of variables being treated as global in recursion has been described in the literature as the “looping” mental model of recursion; see References [11, 17, 22]. The popularity of this answer indicates that instructors should explicitly address when students must return the result of a recursive call of a function and how to properly pass variables when using recursion.

4.8 Question 8: Random-access List Implementation

Question 8 tests students’ ability to choose an appropriate data structure, given a high-level description of the data to be stored and the planned access pattern. It corresponds to learning goal 3, “Compare data structure tradeoffs to select the appropriate implementation for an abstract data type.” The correct response is option A, and Figure 4(b) provides the frequency of student responses for each answer choice.

Our interviews revealed that this question captures student misconceptions that essentially derive from having memorized atomized facts about performance of arrays and linked lists without the conceptual understanding of how to apply those facts to engineering design choices. It was

Fig. 4. Box plot results for Q7 and Q8. Correct answer choices are marked with green rectangles on each chart. Q7 asked students to mark all correct answer choices, although there was a single correct choice.

Question 8

Suppose that your program stores a list of Strings. The user is permitted to access the string at a given position (index) in the list, and can make as many accesses as they wish. N is the number of strings in the list.

Which List data structure would provide the best performance for the user accesses and why?

Select **ONE**:

- A. ArrayList is best as it guarantees constant-time access.
- B. ArrayList is best as it guarantees access time proportional to $\log N$ using binary search.
- C. Unsorted DoublyLinkedList is best as it guarantees constant-time access.
- D. Unsorted DoublyLinkedList is best as it guarantees access time proportional to $\log N$ using binary search.
- E. Sorted DoublyLinkedList is best as it guarantees constant-time access.
- F. Sorted DoublyLinkedList is best as it guarantees access time proportional to $\log N$ using binary search.

particularly important to the members of our expert panel that students be able to make such engineering design choices that go beyond simply memorizing particular data structure facts.

Some of the atomized facts that students verbalized during interviews for this question included the following (these are all true but *not* all applicable to this situation, and their misapplication leads students to the various distractor choices):

- Inserting into the front or “middle” of an `ArrayList` can be time-consuming, since all elements following the inserted one need to be moved over to create space (students may correctly understand this to be a reason that we may sometimes prefer a linked-list to an array).
- Doubly linked lists with both a head and tail pointer allow both forward and reverse traversal of the list.
- Binary search is an efficient search mechanism.
- To use binary search, data must be in sorted order.

Two key insights for this question are that a linked list’s insert/remove flexibility is not advantageous for an access pattern that does not include insertion/removal and that binary search is not advantageous when we are given the index location of the element to access. Thus, the only relevant focus of analysis is the access time when the element’s index is given.

4.8.1 Correct Choice. Option A correctly identifies the `ArrayList` as the correct data structure for storing N strings and allowing access to them according to their position (index). This was chosen by a median of 64% of students.

4.8.2 Distractor B: Unwarranted Assumptions about Access Pattern and Sorting. Distractor B proposes using binary search to access elements of the list, but this errs in two ways. First, there is no need to perform binary search if the index is given, as it is in this question. Second, the binary search algorithm only works on sorted data, and this answer choice does not specify that the data would be sorted. This distractor was chosen by a median of 10% of students.

4.8.3 Distractors C and E: Incorrect Analysis of `DoublyLinkedList` Traversal Cost. Distractors C and E propose using a linked data structure, despite the fact that, as mentioned above, list editing is not required for this application.

These distractors also incorrectly suggest that a list, whether unsorted (Distractor C) or sorted (Distractor E), can support constant-time access to arbitrary elements. In fact, an element at a given index must be accessed by an $O(N)$ traversal from either the head or the tail. In our interviews, some students thought that this traversal of the doubly linked list would be better than linear time, because it could proceed simultaneously forwards from the head and backwards from the tail. While both directions of traversal are possible, the access remains linear time cost. The median percentage of students who chose these options was 0% for C and 3% for E.

4.8.4 Distractors D and F: Incorrect Application of Binary Search to a Linked Structure. Distractors D and F propose using a linked data structure, despite the fact that, as mentioned above, list editing is not required for this application.

These distractors also propose using binary search. However, a $\log(N)$ binary search algorithm is impossible on `DoublyLinkedList` structures, since it requires random access ability. Sorting is required for binary search, making choice D incorrect in an additional dimension. The median percentage of students who chose these answers was 0% for D and 17% for F.

4.9 Question 9: Binary Search Tree Properties

Question 9 asks students to predict the properties of a data structure (BST) after a series of operations (insertions) is performed on it. It requires students to identify which properties of the tree change and to consider how long the insertions will take (both in good and worst-case scenarios). It corresponds to learning goal 5, “Trace through and predict the behavior of algorithms (including code) designed to implement data structure operations.” The correct answer combination is B, C, and D. Figure 5(a) illustrates how frequently students selected each answer choice.

Question 9

Consider a BinarySearchTree T_0 , containing N numerical values, whose root is the median value in the tree. (The median value is the value such that half of the values are less and the other half are greater.)

Suppose that N additional values are added to the tree T_0 , using the standard insertion algorithm, to create an updated tree T_1 . The newly added values are all larger than the largest value that T_0 stored, and they are added in an unknown order.

Note: N refers to both the original number of nodes in T_0 and the number of nodes added to T_0 to create T_1 . Hence, T_1 contains $2N$ nodes. You may assume $N > 0$.

Which of these statements is/are true about the creation of T_1 ?

Select **ALL** that apply:

- A. The root of T_1 holds the median value of T_1 .
- B. There are more values in the right subtree of T_1 than in the left subtree.
- C. Adding the last new value **could** take time proportional to $\log N$.
- D. Adding the last new value **could** take time proportional to N .
- E. Adding the last new value **could** take time proportional to $N \log N$.

4.9.1 Correct Choices. The correct selections are B, C, and D. Option B is true because, with the median at the root, the tree initially contains an equal number of nodes in the left and right subtrees. All new insertions are larger than any existing value in the tree, thus they must all appear in the right subtree, making it larger than the left. A median of 79% of students chose option B. Options C and D depend on the order in which the insertions are made, which is unspecified by the question. If the new values are added in an order that balances the right subtree, then the final insertion will take $\log N$ time (C). However, if the values are added in the worst-case order (increasing numerical order), then the right subtree devolves into a list, and the final insertion requires time proportional to N . A median of 61% of students chose C, and 46% of students chose D. Only 21% of students correctly chose all three answers.

Many students selected B and C without 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 [18], 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.

Also, many students selected B and D but failed to select option C, indicating they did not think the time to insert could be proportional to $\log N$. Based on open-ended practice exams in which we asked students to explain their selections, students who did not select C often indicated that 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. Despite clarifying the text of the question many times, we believe this may still primarily be an issue of reading comprehension or students not reading the question carefully enough.

4.9.2 Distractor A: Root Remains the Same. A median of 9% of students chose A, indicating that they thought the root would continue to hold the median value in the tree, even after new nodes

Fig. 5. Box plot results for Q9 and Q10. Correct answer choices are marked with green rectangles on each chart. Both Q9 and Q10 asked students to mark all correct answer choices, but only Q9 had multiple correct answer choices.

were added. In our open-ended practice exams, more than half of the students who selected A justified their answer by indicating that the root of a BST remains the same. Selecting this option indicates that students have internalized the idea that once in a BST, a node will stay in the same position unless nodes are removed (correct); however, they seem to have memorized this idea to the extent that it did not occur to them that while the root node's *value* will not change, properties related to it (like the median) can change.

4.10 Question 10: Linked Lists for Stack Functionality

Question 10 tests students' ability to correctly apply standard data structures when implementing prescribed program functionality, which is closely aligned with learning goal 2, "Select appropriate abstract data types for use in a given application." The question asks students to build "undo" functionality (which behaves as a LIFO stack) using a singly linked list, taking performance into account. The correct answer for this problem is option D. Figure 5(b) illustrates how frequently students selected each answer choice.

4.10.1 Correct Choice. While options C and D both meet the correctness requirements (LIFO functionality), only option D offers the best performance ($O(1)$). A median of 84.2% of students correctly chose option D, making it by far the most popular answer.

4.10.2 Distractor C: $O(N)$ Performance. Option C is the second-most frequently selected choice (median of 29%), and while it correctly solves the problem, its $O(N)$ performance is substantially worse than option D. Most of the students who selected C selected both C and D, which suggests that they understand the correctness requirements and data structure behavior without accounting for performance. In response to these results, we recommend that instructors explicitly highlight concrete examples in which different "correct" applications of data structures exhibit disparate performance results.

Question 10

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.

You have a `SinglyLinkedList` whose implementation has a reference only to its head. **With the goal of providing the best performance for your undo operation, how would you best use the `SinglyLinkedList` to store the user’s actions?**

Select **ALL** that apply:

- A. Add to the end of the `SinglyLinkedList` and remove from the start.
- B. Add to the start of the `SinglyLinkedList` and remove from the end.
- C. Add to the end of the `SinglyLinkedList` and remove from the end.
- D. Add to the start of the `SinglyLinkedList` and remove from the start.

4.10.3 Distractors A and B: Incorrect. Neither option A nor B correctly solve the stated problem, and students select them at approximately the same rate (a median of 9.8% and 9.5%, respectively). During open-ended practice exams, there was no clear pattern in students’ explanations for selecting these choices.

4.11 Question 11: Testing BST Search

This question is designed to measure students’ ability to properly test their code to identify possible mistakes in the implementation. The learning goal associated with this question is #6, “Identify and remedy flaws in a data structure implementation that may cause its behavior to differ from the intended design.” Based on our interviews with students, testing appears to not always be taught extensively, so possible improvements to student performance on this question would be to teach students how to design good tests, perhaps discussing code coverage. The correct answer for this question is E. Figure 6(a) provides the frequency of student responses for each answer.

4.11.1 Correct Choice. Option E searches for an element at each level of the tree, both in right and left subtrees, and for an item not in the tree. The median correct response rate is 71%.

4.11.2 Distractor B: Only Inspect Leaf Nodes. The most popular incorrect answer (12% median response rate) is to search for each leaf node. Although this ensures the search is capable of going down either the right or left branches of the tree, it does not consider searches for the root, levels other than the leaves, nor an unsuccessful search.

4.11.3 Distractor D: Inspect All Except Leaf Nodes. The second-most-popular incorrect answer (6% median response rate) searches for the root node, down the left and right subtrees, and for an element not present. However, it does not search for any elements in leaf nodes.

4.12 Question 12: Binary Tree Traversal

Question 12 asks students to recursively reason about information stored in a binary tree by computing a width function. The question corresponds to learning goal 4, “Design and modify data structures capable of insertion, deletion, search, and related operations,” and the correct response is option C. Figure 6(b) provides the frequency of student responses for each answer choice. The

Fig. 6. Box plot results for Q11 and Q12. Correct answer choices are marked with green rectangles on each chart.

Question 11

Suppose you just wrote the search method for your `BinarySearchTree` implementation. The specification is below.

`search(e)`

Input: An element `e`

Description: Determines if `e` is in the tree

Return: True if `e` is in the tree; False if not.

To test your new method, you build the tree below.

You are not sure if your implementation has flaws. Please select the four integer values that you should search for in this tree to gain as much confidence as possible in the correctness of your algorithm.

Select **ONE**:

- A. 10, 15, 20, 30
- B. 15, 25, 35, 45
- C. 15, 20, 30, 40
- D. 20, 30, 40, 50
- E. 25, 30, 40, 50

original, open-ended, version of this question asked students to write code for the entire function, but we found that doing so took students a considerable amount of time. As such, we reduced the question to only require code for the core recursive component.

Question 12

The *width* of a `BinaryTree` is the maximal number of nodes on a path from one node in the tree to another. (The width of an empty `BinaryTree` is 0.) Here are some examples, with the relevant path indicated in bold. Note in particular that the relevant path does not necessarily include the root of the `BinaryTree`.

<i>width</i>	1	3	5	7
<i>tree</i>

The *height* of a node in a `BinaryTree` is the number of nodes on the longest path from the node down to a leaf. (The height of a leaf is 1.) Assume that each `BinaryTree` node contains its height in an instance variable named `height`. You may also assume that you have a function, `max`, which returns the greater of two values.

Assume that you are writing a function named `width` that, when given a node as argument, returns the width of the `BinaryTree` rooted at that node. In authoring this function, consider a node with both left and right children. What is the width of the `BinaryTree` rooted at that node? We've started the answer for you.

The width of a node is the maximum of:

- `width(node.left)`
- `width(node.right)`
- _____

Select **ONE** of the following to fill in the blank:

- A. `node.left.height + node.right.height`
- B. `width(node.left) + width(node.right)`
- C. `1 + node.left.height + node.right.height`
- D. `1 + width(node.left) + width(node.right)`
- E. `1 + max( width(node.left) + node.right.height, node.left.height + width(node.right) )`

A challenge in designing the recursive `width` function is that the maximal length path could either include the current node, or not, and the handling of these two cases looks very different. The case where the maximal path is located somewhere among the current node's left or right descendants (and does not include the current node) is handled in the provided code lines numbered 1 and 2, respectively.

4.12.1 Correct Choice. Option C correctly accounts for the case where the maximal length path does include the current node, by gathering the information from left and right children, and remembering to add 1 to account for the current node itself. A median of 46% of students correctly chose C.

4.12.2 Distractors B and D: Incorrect Gathering of Left and Right Path Lengths. Distractors B and D attempt to gather the path lengths from the left and right children by recursively calling the `width` function. This is incorrect, because it will count paths that start and end in the subtrees, not extending up to the current node. Distractor D was the second-most commonly chosen distractor (median of 13%), because it shares the same structure as the correct answer: gathering information

Fig. 7. Box plot results for Q13. The correct answer choice is marked with a green rectangle on the chart.

from left and right and adding 1 to account for the current node. A median of 7% of students chose distractor B. We suspect that students answering B or D believe that recursion is necessary to solve this question but struggle to do so properly.

4.12.3 Distractor E: Incorrectly Conflates Completed Paths and Continuing Paths. Like Distractors B and D, Distractor E attempts to gather the path lengths from the left and right children by recursively calling the width function. The code correctly adds 1 to account for the current node, but incorrectly mixes the maximal paths of the subtrees with the heights of the subtrees, essentially creating new branches off of subtree paths from the current node. Students may be drawn to this option because they understand that implementing the width function requires finding the max of some options, while neglecting to observe that the problem asks them only to list the three options. Distractor E was the most commonly chosen distractor with a median of 21%.

4.13 Question 13: Abstract Interfaces

Question 13 asks students to compare two abstract interfaces and select the most appropriate one for solving a particular problem (recording e-mail addresses). It corresponds to learning goal 2, “Select appropriate abstract data types for use in a given application.” The BDSI assumes that students are already familiar with an abstract List interface (at the level described in the BDSI data structure definitions in Appendix A), whereas it does not assume that students have previously seen the Set interface. The correct answer is option C. Figure 7 illustrates how frequently students selected each answer choice.

4.13.1 Correct Choice. The correct answer is option C—if the goal is to recognize unique addresses, then storing duplicates does not help, since a duplicate will cause the same address to be counted multiple times. Overall, students tend to do very well on this question, which has median correctness rate of 80%.

4.13.2 Distractor B. Option B is the most commonly selected distractor, with a median rate of 7%. In interviews, students were drawn to this response because a list most closely matched the

Question 13

You are given two interfaces for Set and List. A Set is a new interface for a collection of elements with no ordering and no duplicates.

The interface for Set is:

<code>add(e)</code>	add e to the Set (if e is already in the Set, do nothing)
<code>remove(e)</code>	remove e from the Set (if e is not in the Set, do nothing)
<code>contains(e)</code>	return True if e is in the Set, False if not
<code>size()</code>	return the number of elements in the Set

The interface for List is:

<code>addAtStart(e)</code>	add e to the beginning of this List
<code>addAtEnd(e)</code>	add e to the end of this List
<code>removeAtStart()</code>	remove and return the first element from this List
<code>removeAtEnd()</code>	remove and return the last element from this List
<code>contains(e)</code>	return True if e is in the List, False if not
<code>size()</code>	return the number of elements in the List

You are given the problem of determining the number of unique e-mail addresses you've received e-mail from. As input, you are given a file with the sender email address of all emails you've ever received in the order you received them.

You choose to use the Set interface rather than the List interface. Did you choose the interface that appropriately models the problem?

Select **ONE**:

- A. No, the List more appropriately models the problem because it maintains the order in which emails were received.
- B. No, the List more appropriately models the problem because it can contain duplicate email addresses.
- C. Yes, the Set more appropriately models the problem because it cannot contain duplicate email addresses.
- D. Yes, the Set more appropriately models the problem because it maintains the order in which emails were received.
- E. It doesn't matter, both interfaces appropriately model this problem.

form of the data as it was provided ("a file...of all emails you've ever received in the order you received them"). Preservation of the original data was prioritized over efficiency for the task at hand.

4.13.3 Distractor E. Selecting option E (median 4.9%) suggests that a student is having trouble understanding a new abstract interface and comparing it against a known interface. While we expect that many courses already do this, we recommend that instructors walk through examples in which they evaluate two or more abstract interfaces for fitness in solving a problem.

5 HOW TO USE THE BDSI

We now address how faculty can adopt the BDSI as an instrument to compare aggregate student knowledge of basic data structures across institutions, instructors, courses, and pedagogies.

The BDSI is available to faculty by joining the BDSI group online.¹ The format of the instrument is a PDF in two parts: the introductory material (including pseudocode guide) and the test itself. It should be printed for students to complete on paper. There are some diagrams in the BDSI, and an accessible, though not validated, version with text descriptions in place of the diagrams is also available for visually impaired students to complete on a device with screen reader software. Faculty should plan on giving students about one hour to complete the CI. It is important for faculty to give the entire CI, and not a subset of CI questions, as we have only demonstrated validity for the CI as a whole—results are not comparable for individual questions.

We recommend only using the BDSI as a post-test, as students will be unfamiliar with terminology prior to being taught basic data structures. Because CIs are not intended to be used for summative assessment of individual students, the BDSI should not be used on, or in place of, a comprehensive final exam, or as another graded assignment. Additionally, because CI questions cannot be altered, we caution against using the CI in any way that would incentivize students to either look for copies of the questions ahead of time or share questions with each other or publicly (i.e., using it as a graded exam). However, students may find it useful as a supervised “dress rehearsal” practice final exam, which faculty can use as a mutually beneficial opportunity to obtain student results.

6 CONCLUSION

In this work, we presented the results of administering the BDSI, a Concept Inventory for basic data structures, to 1,963 students across seven institutions. We discuss both the results and the implications of those results for instructors. We hope that this article will serve as a resource for instructors who administer the BDSI in their own courses, as well as inform instructors about common student misconceptions in basic data structures.

The value of a CI is not in the questions themselves, but in its capacity to drive pedagogical change. As such, along with reporting on our results, we have made the CI available to the community. Educational researchers may now use this assessment of student knowledge in their research, and instructors can use it to gauge strengths and weaknesses in their own curricula. Ultimately, we hope that this CI will be taken up and deployed by the CS education research community to further enhance student learning.

¹<https://groups.google.com/forum/#!forum/cs2-bdsi-concept-inventory>.

APPENDIX

A BDSI INSTRUCTIONS, DEFINITIONS, AND PSEUDOCODE SUPPLEMENT

Instructions

This assessment contains questions with two response types: **select one** and **select all**.

The **select one** answer choices are preceded by circles: ☐. For **select one** questions, you should select the one choice that best answers the question.

The **select all** answer choices are preceded by squares: ☐. For **select all** questions, you should select all answer choices that correctly answer the question.

Please clearly indicate your selections by completely filling in the choice's shape. For example:

Select one:

- ☐ Not selected
- ☐ Not selected
- ☒ Selected
- ☐ Not selected

Select all:

- ☒ Selected
- ☐ Not selected
- ☒ Selected
- ☐ Not selected

The following questions are designed to assess your knowledge of basic data structures, regardless of the programming language you typically use. As a result, the questions involving code are written in pseudocode. **A description of the pseudocode language can be found on the following pages.** Additionally, you may have also used slightly different terms for data structures than the ones used in these questions. As such, **this supplement also includes a summary of terms and definitions.**

Data Structure Terms and Definitions - Page 2

Pseudocode Language Guide - Pages 3 & 4

Interface and Data Structure Terms

Interfaces

List: A List can have an element added at any position and removed from any position, and you can access any position.

Stack: A Stack can have an element pushed to the top of the Stack, an element removed from the top, and you can inspect only the value of the element presently at the top of the Stack.

Queue: A Queue can have elements added to the back and removed from the front. You can inspect the contents of only the element at the front of the Queue.

Data Structures

ArrayList: Implements the List interface by storing elements in a contiguous block of memory to speed up access to requested indices. The implementation of this dynamic, resizable array manages resizing when adding and removing elements. This data structure is known as an array list in Java, list in Python, and vector in C++.

LinkedList: Implements the List interface by storing a series of objects that refer (link) to subsequent objects. You can typically access the head (first) element in the list. You may or may not be able to directly access the tail (last) element in the list.

A LinkedList may take the form of:

A SinglyLinkedList: A LinkedList for which each object, starting with the head, refers to the next object. The last object would refer to nothing (nil).

A DoublyLinkedList: Similar to a SinglyLinkedList except each object refers both to the next object and to the previous object.

BinaryTree: A linked structure of nodes that starts with an object called the root. Each node refers to a left child and a right child. Either or both those nodes can refer to nothing (nil).

BinarySearchTree: A BinaryTree with the following restriction: for each node, all the values in the left subtree are less than or equal to the node's value and all the values in the right subtree are greater than the node's value. A BinarySearchTree may, or may not, support having duplicate elements and does not automatically attempt to balance its structure.

Pseudocode Guide

Syntax	Examples
Statements <code>variable = expression</code> Math Operators: +, -, *, /, % Relational Operators: ==, <, <=, >, >=, != Conditional Operators: AND, OR, NOT Booleans: True/False	<code>total = 0</code> <code>x = 4 * 3</code> <code>name = "John Smith"</code> <code>flag = True</code> <code>answer = True AND (True OR False)</code>
If Statement (Conditional) <code>IF condition THEN</code> <code>statement(s)</code> <code>ELSE IF condition THEN</code> <code>statement(s)</code> <code>ELSE</code> <code>statement(s)</code> <code>ENDIF</code>	<code>IF testScore >= 90 THEN</code> <code>grade = "A"</code> <code>ELSE IF testScore >= 80 THEN</code> <code>grade = "B"</code> <code>ELSE IF testScore >= 70 THEN</code> <code>grade = "C"</code> <code>ELSE</code> <code>grade = "F"</code> <code>ENDIF</code>
For Loop (Definite) <code># counter from start up to but not</code> <code># including end</code> <code>FOR counter = start TO end BY # DO</code> <code>statement(s)</code> <code>ENDFOR</code>	<code>FOR x = 1 TO 5 BY 1 DO</code> <code>xSquared = x * x</code> <code>PRINTLN x, xSquared</code> $\Rightarrow$ <code>ENDFOR</code> <div style="float: right; text-align: right;"> Output 1 1 2 4 3 9 4 16 </div>
While Loop (Indefinite) <code>WHILE condition DO</code> <code>statement(s)</code> <code>ENDWHILE</code>	<code>count = 5</code> <code>WHILE count < 9 DO</code> <code>PRINTLN count</code> $\Rightarrow$ <code>count = count + 1</code> <code>ENDWHILE</code> <div style="float: right; text-align: right;"> Output 5 6 7 8 </div>
List Interface <code># two implementations of the list interface:</code> <code># ArrayList and LinkedList</code> <code># differences described in the prev section</code> <code>list_var = new list-type(item1, item2, ...)</code> <code># size of list</code> <code>list_var.size()</code> <code># access the first element</code> <code>list_var[0]</code> <code># add element at position index</code> <code>add(list_var, index, value)</code> <code># adding an element in the middle shifts the</code> <code># later items:</code> <code># delete element at position index</code> <code>del(list_var, index)</code>	<code>colors =</code> <code>new ArrayList("red", "blue", "pink")</code> <code>colors2 =</code> <code>new LinkedList("red", "blue", "pink")</code> <div style="float: right; text-align: right;"> Output 3 </div> <code>colors.size()</code> <code>colors[0]</code> $\Rightarrow$ "red" <code>add(colors, 3, "orange")</code> $\Rightarrow$ <code>ArrayList("red", "blue", "pink", "orange")</code> <code>add(colors2, 1, "green")</code> $\Rightarrow$ <code>LinkedList("red", "green", "blue", "pink")</code> <code>del(colors, 2)</code> $\Rightarrow$ <code>ArrayList("red", "blue", "orange")</code>

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of the following collaborators: Meghan Allen, Owen Astrachan, John Bell, Darci Burdge, Stephanie Chasteen, Rajendra Chattergoon, John Donaldson, Maureen Doyle, John Glick, Paul Hilfinger, Josh Hug, Marina Langlois, Kevin Lin, Lisa Meeden, Zach Palmer, Mahika Phutane, Joe Politz, Kate Rydberg, Samar Sabie, Kate Sanders, Jacqueline Smith, Marty Stepp, Paramsothy Thananjeyan, Steve Wolfman, Anna Zaitsev, and Christine Zhou.

REFERENCES

- [1] Wendy Adams and Carl Wieman. 2011. Development and validation of instruments to measure learning of expert-like thinking. *Int. J. Sci. Educ.* 33, 9 (2011), 1289–1312.
- [2] Leilani Arthurs, Jennifer F. Hsia, and William Schweinle. 2015. The oceanography concept inventory: A semicustomizable assessment for measuring student understanding of oceanography. *J. Geosci. Educ.* 63, 4 (2015), 310–322.
- [3] Stephanie V. Chasteen, Rachel E. Pepper, Marcos D. Caballero, Steven J. Pollock, and Katherine K. Perkins. 2012. Colorado upper-division electrostatics diagnostic: A conceptual assessment for the junior level. *Phys. Rev. Special Top. - Phys. Educ. Res.* 8, 2 (2012).
- [4] Catherine H. Crouch and Eric Mazur. 2001. Peer instruction: Ten years of experience and results. *Amer. J. Phys.* 69, 9 (2001), 970–977.
- [5] csedresearch. Evaluation Instruments. Retrieved on April 2021 from <https://csedresearch.org/evaluation-instruments/>.
- [6] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. 21–26.
- [7] Charlene D'Avanzo. 2008. Biology concept inventories: Overview, status, and next steps. *BioScience* 58, 11 (2008), 1079–1085.
- [8] Jerome Epstein. 2007. Development and validation of the calculus concept inventory. In *Proceedings of the 9th International Conference on Mathematics Education in a Global Community*. 165–170.
- [9] Mohammed F. Farghally, Kyu Han Koh, Jeremy V. Ernst, and Clifford A. Shaffer. 2017. Towards a concept inventory for algorithm analysis topics. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 207–212.
- [10] Mark J. Gierl, Okan Bulut, Qi Guo, and Xinxin Zhang. 2017. Developing, analyzing, and using distractors for multiple-choice tests in education: A comprehensive review. *Rev. Educ. Res.* 87, 6 (2017), 1082–1116. DOI: <https://doi.org/10.3102/0034654317726529>
- [11] Tina Götschi, Ian Sanders, and Vashti Galpin. 2003. Mental models of recursion. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. 346–350.
- [12] Richard R. Hake. 1998. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *Amer. J. Phys.* 66, 1 (1998), 64–74.
- [13] Sally Hamouda, Stephen H. Edwards, Hicham G. Elmongui, Jeremy V. Ernst, and Clifford A. Shaffer. 2017. A basic recursion concept inventory. *Comput. Sci. Educ.* 27, 2 (2017), 121–148.
- [14] Geoffrey L. Herman, Craig Zilles, and Michael C. Loui. 2014. A psychometric evaluation of the digital logic concept inventory. *Comput. Sci. Educ.* 24, 4 (2014), 277–303.
- [15] David Hestenes, Malcolm Wells, and Gregg Swackhamer. 1992. Force concept inventory. *Phys. Teach.* 30, 1 (1992), 141–158.
- [16] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. DOI: <https://doi.org/10.1145/1734263.1734299>
- [17] Hank Kahney. 1983. What do novice programmers know about recursion. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 235–239.
- [18] Kuba Karpierz and Steven A. Wolfman. 2014. Misconceptions and concept inventory questions for binary search trees and hash tables. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. 109–114.
- [19] Stephen Krause, James Birk, Richard Bauer, Brooke Jenkins, and Michael J. Pavelich. 2004. Development, testing, and application of a chemistry concept inventory. In *Proceedings of the 34th Annual Frontiers in Education Conference*.
- [20] J. C. Libarkin, E. M. G. Ward, S. W. Anderson, G. Kortemeyer, and S. Raeburn. 2011. Revisiting the geoscience concept inventory: A call to the community. *GSA Today* 21, 8 (2011), 26–28.
- [21] David P. Maloney, Thomas L. O'Kuma, Curtis J. Hieggelke, and Alan Van Heuvelen. 2001. Surveying students' conceptual knowledge of electricity and magnetism. *Amer. J. Phys.* 69, S1 (2001), S12–S23.
- [22] Renée McCauley, Scott Grissom, Sue Fitzgerald, and Laurie Murphy. 2015. Teaching and learning recursive programming: A review of the research literature. *Comput. Sci. Educ.* 25, 1 (2015), 37–66.
- [23] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proceedings of the 12th ACM Conference on International Computing Education Research*.
- [24] Miranda C. Parker, Yvonne S. Kao, Dana Saito-Stehberger, Diana Franklin, Susan Krause, Debra Richardson, and Mark Warschauer. 2021. Development and preliminary validation of the assessment of computing for elementary studies (ACES). In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*.

- [25] Leo Porter, Saturnino Garcia, Hung-Wei Tseng, and Daniel Zingaro. 2013. Evaluating student understanding of core concepts in computer architecture. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. 279–284.
- [26] Leo Porter, Daniel Zingaro, Cynthia Lee, Cynthia Taylor, Kevin C. Webb, and Michael Clancy. 2018. Developing course-level learning goals for basic data structures in CS2. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 858–863.
- [27] Leo Porter, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Kevin C. Webb, Cynthia Lee, and Michael Clancy. 2019. BDSI: A validated concept inventory for basic data structures. In *Proceedings of the ACM Conference on International Computing Education Research (ICER'19)*. Association for Computing Machinery, New York, NY, 111–119. DOI: <https://doi.org/10.1145/3291279.3339404>
- [28] A. Rachmatullah, B. Akram, D. Boulden, B. Mott, K. Boyer, J. Lester, and E. Wiebe. 2020. Development and validation of the middle grades computer science concept inventory (MG-CSCI) assessment. *Eurasia J. Math., Sci. Technol. Educ.* 16, 5 (2020).
- [29] Catherine Y. Read and Linda D. Ward. 2016. Faculty performance on the genomic nursing concept inventory. *J. Nurs. Scholar.* 48, 1 (2016), 5–13.
- [30] Philip M. Sadler, Harold Coyle, Jaimie L. Miller, Nancy Cook-Smith, Mary Dussault, and Roy R. Gould. 2010. The astronomy and space science concept inventory: Development and validation of assessment instruments aligned with the K–12 national science standards. *Astron. Educ. Rev.* 8, 1 (2010), 010111.
- [31] Alan T. Sherman, Geoffrey L. Herman, Linda Oliva, Peter A. H. Peterson, Enis Golaszewski, Seth Poulsen, Travis Scheponik, and Akshita Gorti. 2020. Experiences and lessons learned creating and validating concept inventories for cybersecurity. In *National Cyber Summit Research Track, NCS 2020 (Advances in Intelligent Systems and Computing)*, Kim-Kwang Raymond Choo, Tommy Morris, Eric Imsand, and Gilbert L. Peterson (Eds.). Springer, 3–34. DOI: https://doi.org/10.1007/978-3-030-58703-1_1
- [32] Michelle K. Smith, William B. Wood, and Jennifer K. Knight. 2008. The genetics concept assessment: A new concept inventory for gauging student understanding of genetics. *CBE–Life Sci. Educ.* 7, 4 (2008), 422–430.
- [33] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26 (11 1983), 853–860. DOI: <https://doi.org/10.1145/182.358436>
- [34] Andrea Stone, Kirk Allen, Teri Reed Rhoads, Teri J. Murphy, Randa L. Shehab, and Chaitanya Saha. 2003. The statistics concept inventory: A pilot study. In *Proceedings of the 33rd Annual Frontiers in Education Conference*.
- [35] Cynthia Taylor, Michael Clancy, Kevin C. Webb, Daniel Zingaro, Cynthia Lee, and Leo Porter. 2020. The practical details of building a CS concept inventory. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. Association for Computing Machinery, New York, NY, 372–378. DOI: <https://doi.org/10.1145/3328778.3366903>
- [36] Cynthia Taylor, Daniel Zingaro, Leo Porter, Kevin C. Webb, Cynthia Bailey Lee, and Mike Clancy. 2014. Computer science concept inventories: Past and future. *Comput. Sci. Educ.* 24, 4 (2014), 253–276.
- [37] Allison Elliott Tew and Mark Guzdial. 2010. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. 97–101.
- [38] Kevin C. Webb and Cynthia Taylor. 2014. 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*. 103–108.
- [39] Michael Zeilik. 2002. Birth of the astronomy diagnostic test: Prototest evolution. *Astron. Educ. Rev.* 1, 2 (2002), 46–52.
- [40] 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 *Proceedings of the 14th ACM Conference on International Computing Education Research*. 169–177.

Received November 2020; revised July 2021; accepted July 2021