ABSTRACT

Is there consensus on what students should learn in CS2? Should they learn to use data structures, understand their specific implementation details, or both? Finally, has the computing education community's answer to the second question changed over time? In this paper, we begin to explore these questions based on an analysis of a key artifact instructors use to assess their students' performance: their final exams. Specifically, we look at two CS2 concepts as covered in those exams: stacks and hashtables.

Our dataset includes 76 exams from 14 institutions around the world spanning 1973-2009 that were gathered as part of the DCER project, which is investigating the feasibility of a repository for computing education research data; to our knowledge this is a novel dataset in computing education. We begin by giving a general feel for this extensive dataset by describing the formats and difficulty level of the stack and hashtable questions and the computing skill students must possess to answer them. Next, we look at the questions' assessment of implementation knowledge versus interface or application knowledge. Despite a number of calls for modern CS2 to focus more on application than implementation, we found no evidence of such a trend. We note, however, that there are institutional differences in the data, and that there are alternative ways in which application may be assessed in a course.

1. INTRODUCTION

Most computer science departments have an introductory data structures course where students learn about the classic structures: stacks, queues, hash tables, trees, graphs, and the like, as well as learning about abstract data types, basic efficiency of algorithms, and the use of these structures as components to build larger programs. Students are assessed in most of these classes by some combination of writing programs and taking exams. The course is a bridge between introductory programming and analysis of algorithms — and a place where students may struggle. There has been some discussion about what sort of programs should be assigned in this course (see Section 2), but little discussion about the written examinations. What sorts of exams are used to assess students in CS2? What, if anything, can they tell us?

In this study, we examine 76 data structures exams from a total of 14 institutions, given between 1973 and 2009. The collection of these exams stems from the DCER project [15], which is investigating the feasibility of a repository for computing education research data. In future work, we will be reflecting on the challenges and advantages of using repository data in this project.

Eventually, we would like to use this dataset as evidence to inform the question: Is there a consensus on what students should learn in CS2? The use of exam questions complements the existing work on programming competencies.

In this first report we begin by providing an overview of the stack and hashing questions on the exams: their format, how difficult they are, and what cognitive computing skills are required to answer them. The stack questions represent an average of 13% of the points per exam in the exams we analyzed; the hash questions, 16%.

With that background, we turn our focus to the following research question:
What is the relative emphasis on implementation, interface, and application (for hashing and stacks) in CS2, and has it changed over time?

In Section 2, we discuss related work on CS2 and exams; in Section 3, we describe our methodology; we present our analysis of the data structures exams in Section 4, discuss the results in Section 5, and conclude in Section 6.

2. RELATED WORK

2.1 Programming skills

While this study analyzes exam questions to learn more about what instructors expect of CS2 students, others have assessed programming competencies more directly. For example, McCracken et al. [14] investigated the programming performance of 216 introductory CS students drawn from four universities. On a set of programming exercises requiring the use of a stack to evaluate arithmetic expressions, they found students’ average scores were less than 23%.

2.2 Evolution of CS2

Before the advent of frameworks such as the Standard Template Library in C++ and the Java Collections Framework, CS2 courses necessarily emphasized the implementation of data structures over their application. The ACM’s 1984 CS2 guidelines explicitly listed the data structure implementation details for which students should be responsible [9], and the story had changed little by the time the Computing Curricula 2001 recommendations were published [8]. In 2003, Tenenberg [18] argued that students would be better served by learning how to use frameworks, since “software development is increasingly driven by the need to design larger and more complex software systems through the integration of existing software components.” Lister et al. [12] acknowledged the growing “implement” versus “apply” debate in 2004, urging instructors to first consider why they teach the data structures course. Their phenomenographic study identified five “categories of intent” among a group of educators.

The 2007 Liberal Arts Computer Science model curriculum [10] recommends that liberal arts students should learn to both apply and implement data structures: “[CS2] focuses on classic data structures from several perspectives, including implementing basic data structures, utilizing standard library classes as building blocks, and analyzing the appropriateness and performance of various data structures in different contexts.” By the time Yarosh and Guzdial described their media computing approach to CS2 [21], the idea of emphasizing the application of data structures over their implementation was treated as mainstream — they had moved on to asserting that one needs a compelling context in which to apply data structures.

In 2009, Ernst et al. [5] summarized the history of CS2 before proceeding to argue that it is time for a third “generation” of CS2 (one in which the emphasis is on applying and extending frameworks):

The first major generation of this course focused on describing the basic data structures (e.g. list, stack, queue), implementing these data structures in various ways (e.g. array-based structures or pointer-based structures), and then implementing some simple examples using these structures.

The second generation of this course focused on using these basic data structures to solve meaningful problems, either implementing the data structures or using them from libraries such as the Standard Template Library (STL) for C++, the Java Collections Framework (JCF), or the Microsoft .NET Framework.

2.3 Alignment

Biggs’ theory of constructive alignment [1] advocates an educational approach in which student assessment activities connect with the intended outcomes of the course. If educators are practicing constructive alignment, assessed work should therefore reveal an educator’s objectives for the course. Even if faculty are not employing constructive alignment, assessed work sheds light upon what students are implicitly being told is important. Final exams are a significant part of this assessed work.

Boud [3] observed that “Assessment methods and requirements probably have a greater influence on how and what students learn than any other single factor”, and Gibbs et al. [6] found similar results when surveying studies of student learning: “Whether or not what it is that assessment is trying to assess is clearly specified in documentation, students work out for themselves what counts — or at least what they think counts, and orient their effort accordingly. They are strategic in their use of time and ‘selectively negligent’ in avoiding content that they believe is not likely to be assessed.”

2.4 Difficulty

Determining the difficulty of exam questions in computer science can be quite problematic. The authors of a study attempting to map multiple-choice questions to levels on the Bloom [2] and SOLO [1] taxonomies reported that the task was “challenging even to an experienced group of programming educators” [20]. The difficulty of applying Bloom’s taxonomy to computer science problems motivated Thompson et al. [19] to publish an interpretation of the taxonomy as it applies to introductory programming exams. Ultimately, they discovered that they could not make an adequate determination without knowing how a course had been taught: “We concluded that in order to effectively analyse a question the person undertaking the analysis should have an in-depth knowledge of the course as a whole.”

Johnson and Fuller [7] question whether Bloom’s taxonomy is appropriate for computer science after encountering similar disagreement between educators — particularly between the instructors responsible for the design and delivery of a course and those simply analyzing exam questions. A final complication arises from the fact that students can experience a given exam question in different ways depending upon their background and preparation. A question requiring nothing more than recall for one student may require more complex cognitive processes for another. Similarly, cognitive load theory [16, 17] implies that a problem’s phrasing and construction can influence the difficulty as experienced by students, independent of the problem’s intrinsic difficulty. One must determine the level of complexity at which most students will experience a problem.
3. METHODOLOGY

In early 2009, in response to postings on mailing lists and requests to colleagues, we collected 90 final exams from CS2 classes from 14 different institutions, 12 in the United States and 2 in Australasia, spanning the time period from 1973 to 2009 (inclusive). Figure 1 shows the types of institution from which the exams were collected (based on the classification system at http://classifications.carnegiefoundation.org/). Each institution contributed at least one exam; the maximum was 16.

Two of the exam files were unreadable. Of the remaining 88, 76 exams were analyzed. Each of the 12 exams that were omitted was from a very-high-research university from which we did include at least five other exams, and for all but one of the omitted exams, we included another exam from the same institution from the same year.

We chose two topics to examine, stacks and hashtables, on the grounds that they are core data structures topics that are likely to occur on most examinations. We divided the 76 exams among five of the researchers, and each researcher identified the questions in his or her assigned exams that related to stacks or hashtables. We examined questions that mentioned stacks and hashing explicitly and also looked for those that, while perhaps not using the terms stack or hash, were assessing those topics. Given the length and complexity of some of the questions, however, it is possible that we may not have identified all questions on these topics. Finally, many questions had more than one part. When that was the case, we considered each part as a separate question.

Once we had identified the questions, we read and re-read them and inductively developed a set of tags, which are discussed individually in the following sections. Each tag was assigned to a different researcher, who read and analyzed all the questions with respect to that particular tag. For example, the same researcher read all the stacks questions to determine whether they were easy, medium, or hard. (See Section 4.1.2 for definitions of easy, medium, and hard.) Then a second researcher looked at 20% of the questions and tagged them independently. In each case we reached at least 80% agreement. Disagreements were resolved by discussion.

For purposes of analysis, we recorded the number of points assigned to each question, and then normalized them so that each exam was considered to have a total of 100 points. When an exam did not indicate points, we arbitrarily gave each question an equal number of points. When an exam did not indicate separate point counts for individual parts of questions, we divided the points for that question evenly among the different parts.

Throughout the rest of the paper, when we indicate the percentage of questions that fall into one category or another, the percentage will be based on the points allocated to those questions.

4. ANALYSIS

In this analysis, we do not claim that our sample is representative of the data structures exams given worldwide during 1973–2009. There are no exams from Europe, South America, or India, for example.

The dataset does include a broad sample over time and space, however. There are exams from several types of institution (see Figure 1); from the United States, New Zealand and Australia (see Figure 2); and from 27 different years: one exam from the 1970s (1973), 8 exams from the 1980s, 35 from the 1990s, and 32 from the 2000s (see Figure 3).

As we hoped, most of the exams included stack and/or hashtable questions. Out of the 76 exams, 70 (92.11%) included either a stack question, a hashtable question, or both (sometimes more than one). All 14 institutions were represented. Overall, we found 91 stack questions (≈510 points) and 215 hashing-oriented exercises (≈904 points, after normalization). The proportions of stack and hash questions for each year are shown in Figures 4 and 5.

While all institutions were represented, there is a great deal of variation. On average, stack questions counted for 13% of each exam and hash questions, 16%. For individual exams, stack percentages range from 0-50 and hash percentages from 0-60.

The time allowed for the exams ranged from 2 hours to 3 hours and 10 minutes, when specified; almost a third of the exams did not state the amount of time. Some of the exams were closed book; others were completely open book. A few exams were somewhere in between, allowing certain specified materials to be used such as a notecard or lab notes in the student’s own handwriting. Almost half of the exams did not specify whether they were open or closed book.

Some exams were language independent. Others used various programming languages, including assembly language, C, C++, Java, Lisp, Macro-10, MIIX, Pascal, and Postscript.
4.1 Basic Analyses

4.1.1 Question Formats

In addition to the standard true/false, multiple choice, short answer, and essay questions, we found two categories that are more characteristic of computing: writing code and drawing diagrams. The distribution of each question format by topic is shown in Figure 6.

The true/false and multiple-choice questions are the easiest to identify. Both types of questions involve recognition rather than recall of information (that is, the student must select the right answer from a set of given choices, rather than remembering it without a prompt [13]). Borderline cases between true-false and multiple-choice had two answers that weren’t “true” and “false” (for example “legitimate” and “illegitimate”); when there were exactly two choices, we classified these as true/false questions.

The remaining question types all require students to recall information, or perhaps synthesize new information. Coding questions were the most common category for stacks, where they accounted for 58% of the points, and the second most important category for hashing, accounting for 26% of the points.

Only a small proportion of the points were allocated to questions where the students had to draw a diagram (1% for stacks and 3% for hashing). These numbers would have been larger if we had included the questions that required the student to fill in the blanks in a pre-drawn diagram (e.g., to indicate the location of items in a hashtable after a series of inserts and/or deletions). These questions seemed more comparable to fill-in-the-blanks questions than to diagrams, so we classified them as short-answer questions.

Short answers and essays accounted for a substantial proportion of the points. Most of these were clearly short answer. Examples include filling in blanks in a sentence or filling in cells in a diagram. Generally short answers were no more than a couple of sentences.

Essays were longer than short answers, but also deeper. Questions that we considered to be essays test higher Bloom level skills such as synthesis, analysis, and evaluation. Demonstrating these skills generally requires a longer answer, but there’s no precise boundary between the two.

We found several questions that, while they could be answered in a paragraph or two, seemed to require a response at a higher Bloom level. For example

> Get back! Your company, Nocturnal Avionics, is designing a new web browser. Your task is to reimplement its “Back” and “Forward” capabilities. These should work as follows:

> [details omitted]

>(a) Explain your design in general: What data do you store, what sort of structure(s) will you use, and when will you store and retrieve data? When do you enable and/or disable the buttons? How do you detect when this should be done?

These questions we tagged as essays; they accounted for 11% of the points for stack questions and 15% of the points for hashing questions.

4.1.2 Difficulty of questions

We realized the difficulty in classifying the questions based on Bloom’s taxonomy or the SOLO structure (see Section 2.3). Given that the researchers were not the instructors for all the exams collected, we could not know the “course as a whole” to be able to “effectively analyse” the question [20]. Therefore, we chose to classify the questions as simply easy, medium, or hard as described below.

Stacks

Easy questions for stacks were those that simply asked the student to have basic knowledge of the data structure requiring memorization skill or knowledge of the interface of the data structure. An example of this would be a question asking the efficiency of the pop operation on a stack (memorization) or the contents of the stack after a series of pushes and pops.
Medium questions required knowledge of standard implementations of a stack (array-based or linked-list based), defining a new stack interface routine (e.g. writing a copy-Stack method using only the standard operations of push, pop, top, and isEmpty), and recalling applications they had previously seen. Reverse Polish Notation (RPN) is a common application presented when teaching stacks and is included in most CS2 textbooks. Assuming that students had encountered this application prior to the exam, questions involving RPN were classified as medium difficulty.

Hard stack questions involve using stacks to solve a new application problem that students most likely had not encountered before or implementing standard stack operations in a new way. One example of this required students to implement a stack in assembly language storing all values in a half-word as opposed to a full word. Questions involving RPN with new operators (perhaps exponentiation or exchange operator) are also included in this category.

Using this classification, we found that the Easy questions represented 18%, Medium questions were 40%, and Hard questions were 42% of the total points (Figure 7).

**Hashing**

There were several categories of easy exercises for hashing. One involved merely recalling a term's definition or recalling a term with a given definition (e.g. "collision" or "load factor"). Another category asked for the run-time behavior of hashing — constant-time access with a good hash function and relatively empty table, more sluggish performance with a bad hash function or a full table — in order to compare a hash table to some other search structure. Remaining easy exercises all involved straightforward hashing of integers or strings. Typical exercises in this category were

- predict the contents of the table after a small number of insertions;
- design a hash function for dictionary words.

Manipulation of more complicated table element data types generally put an exercise into the "medium" category, for example, finding a hash function for inventory entries. Determining if a given hash function was likely to be a good one also earned a "medium" rating. While the task of inserting by hand, resolving collisions with linear probing, was regarded as easy, we classified the task of deleting an entry with linear probing as medium difficulty.

A multiple choice question generally earned an easier rating than the corresponding free response version.

Hard exercises involved more complicated uses of more complicated data, for example, debugging a .hashCode/.equals pair where the element data type was more than just a string or integer. The context could also increase the rating; an example would be a question about the details of using a hash table to keep track of positions in a maze. More cases to deal with also contributed to a "hard" rating, for example, comparing and contrasting several hashing algorithms or data structure choices. Finally, a few exercises involved some significant variation on the standard algorithms, for example by keeping track in the table of a stack of values for each key in order to handle variable declarations using static scope.

Using this classification, we found that the Easy questions represented 17%, Medium questions were 47%, and Hard questions were 36% of the total points (Figure 7).

### 4.1.3 Cognitive Skills Required

Instructors design programming questions to assess different kinds of data structure understanding — related to different ways in which computing professionals may need to employ their cognitive skills. Based on analysis of the stack and hash questions we developed a categorization of the skills tested:

- **Basic (B):** Basic knowledge of how DS works, define API
- **Basic New (BN):** Apply basic knowledge of how DS works to decide if appropriate for new problem
- **Tracing (T):** Trace code and give output, trace/draw DS changes
- **Analysis (A):** Algorithmic runtime, hash function efficiency
- **Writing (W):** DS methods implementations, write code using API, implement defined interfaces
- **Writing New (WN):** Code or pseudo-code to solve new problems using DS, switch code from using one DS to another
- **Other (O):** Find bugs, Translate from one language to another, design and draw new DS to meet new problem requirements

While many of these categories will be recognizable to the computing educator, some notable variations existed. For example, code tracing was common, but sometimes the task was altered to tracing pseudocode:

Consider the following pseudocode: What is written to the screen for the input "carpets"?

```java
declare a stack of characters
while (there are more characters in the word to read) {
    read a character
    push the character on the stack
}
while (the stack is not empty) {
    pop a character off the stack
    write the character to the screen
}
```
4.2 Implementation, Interface, or Application?

As discussed in Section 2.2, in theory CS2 has evolved from a purely implementation emphasis, to one including applications and context. We took the abstract data type [11] or interface as a key dividing point, and considered whether the questions had to do with the implementation (below the interface), the interface, or the application (above the interface). We define four categories of questions based on these perspectives: implementation, interface, apply-given, and apply-select.

A question is at the implementation level if it requires the student to know and/or be able to reason about how the data structure (or one of its components) is implemented. Examples include writing the code to implement a stack using an array, or drawing a picture of a hash table using open addressing after very some number of puts and gets. Filling in the boxes of a drawing can be implementation or not: for hash tables, putting numbers in cells or drawing lists below the correct cell depends on the implementation being an array, while drawing the contents of a stack in a vertical sequence of boxes is not, as the boxes abstract away from whatever the implementation might be. Questions involving components of data structures (hash functions, for example) that are used similarly in multiple implementations are also considered implementation — having to know about things inside of the encapsulation is characteristic of an implementation question.

A question is in the interface category if it requires knowledge of the interface of a data structure: what its operations are, what they do, and their efficiencies (independent of implementation). A typical interface question would be one in which the student is asked to give the results of a series of pushes and pops on a stack. If there is a problem to solve, or code to trace, it can still be an interface question as long as the student is not required to perform any explicit task beyond executing structure operations. An efficiency question about a data structure operation (such as push on a stack, or get from a hashtable) falls here as well, if it is independent of implementation. By contrast, determining whether a character sequence has balanced parentheses using a stack would fall into the next category, apply-given.

A question in the apply-given category requires the student to know or explain how a given structure could be used to perform some task (the task is something beyond simply calling a given sequence of methods). Examples: using a stack to check for balanced parentheses, or using a stack to evaluate an infix expression, or using a hash table to implement a spell-checker. On the margin, but in apply-given, are questions that ask students to compare the suitability of different data structures for a given task: if the different structures to compare are stated explicitly, it is apply-given; if they are not (e.g. what is the best data structure to use for task x?), it is not — it would be apply-select. On the margin: efficiency of problem-task things that need to reflect underlying implementation. This category (and Apply-select) assumes the application need not break encapsulation; a question that requires encapsulation violations would be considered an implementation question.

A question in the apply-select category requires the student to pick an appropriate data structure for a given task as well as knowing or explaining how to apply it. Examples: give a design for a spell-checker, or explain what data structure you would use for sets to make membership tests as efficient as possible. The selection can not be from a given set of alternatives (which would make it apply-given).

There is one more category, none-of-the-above, for questions that fall outside of the above categories. As this accounted for less than 2% of the points we ignore these in our analysis.

We observe a pronounced difference in emphasis between stack and hashtable questions. As shown in Figure 9, a majority of hash table points are from implementation questions — those requiring knowledge below the interface, while a majority of the stack question points concern issues at or above the interface (interface, apply-given, apply-select). Additionally, there are very few interface questions for hashtables — perhaps because the behavior of hashtables in terms of puts, gets, and removes without a motivating task is too simple. Finally, while the hashtable questions give approximately equal weight to apply-given and apply-select questions, there are relatively few apply-select questions for which stack is the appropriate answer.

4.3 Whether the emphasis has changed

We found no clear increase in the ratio of exam points per year assessing application and interface, relative to implementation (Figures 10 and 11). By our data, it would seem that assessment practice has not yet followed various calls in the literature to emphasize application over implementation, at least for stacks and hashtables.

Although there does not appear to be any clear trend, av-
averages may hide a potentially important issue: different institutions may have strong opinions on the implement-apply debate. To explore this question, we selected four institutions (the four from which we had at least five exams each, including some from the last 5 years). In Figure 12, we see that there are dramatic institutional differences in emphasis of implementation versus application. Institutions G and J focus almost entirely on implementation; Institution H indicates a possible trend; and Institution L shows a much greater interest in applications throughout.

The results for stacks, although we do not present them for space reasons, are similar: notable institutional differences (Institution H tests only stack interface knowledge, though very little of that) with no overall trends over time.

Looking closely at the questions, we see certain changes over time. For example, after the early 2000s, as many institutions shifted to Java, we saw a noticeable decrease in questions about linear probing. This could be explained by the fact that Java’s built-in java.utilHashtable does not use linear probing.

5. DISCUSSION

There were examples of application questions in our data, but there weren’t many, and there was no clear increase over time. Given the emphasis in the literature on the desirability of moving away from focusing on implementation and towards a greater emphasis on applications, we expected to see a similar trend in these examinations. We did not, at least for the data structures examined.

Why isn’t the shift from implementation to application evident in these data? There are a number of possible reasons. One possible explanation is that while broad in scope, our sample is not necessarily representative. We have no examinations from Europe, for example, or anywhere outside the United States and Australasia, and we limited our focus to stacks and hashtables.

A second possible explanation is that application questions are difficult for instructors to write. They require identifying a good application. In addition, if the application has not been covered in class, describing it may take 2-3 pages, even for a short-answer question.

Third, application questions — if they are short-answer, coding, or essay questions, like most of the questions on these exams — are difficult to grade. See Denny et al. [4], where the difficulty of developing and applying a code grading rubric is discussed. Well-designed multiple-choice questions might help to solve this problem.

Fourth, textbooks are biased towards implementation. In the typical book, there is a chapter for each data structure. Exercises at the end of the chapter, though possibly including application questions, use the data structure of the chapter and thus never get to the apply-select step. A capstone chapter that outlines (and perhaps solves) a problem for which the key data structure isn’t obvious might be a good solution for this problem.

Finally, it is possible that instructors have chosen to assess application skills using programming assignments. For example, when McCracken et al. [14] wanted to assess computing students at the end of their first year, they assigned programs (and analyzed the solutions) for the following tasks:

...
P1: evaluate a postfix expression

P2: evaluate an infix expression with no operator precedence and no parentheses

P3: evaluate an infix expression with parenthesis establishing precedence

Students were given 1 to 1.5 hours to write one of these programs in a closed lab, comparable to the amount of time they might have on an exam.

It is possible to assess application skills on exams, however, and in fact, in a greater variety of ways than permitted by programming assignments. Our exams included apply-select questions in a variety of formats, including multiple-choice, short answer, and essay questions, as well as code. One example, very closely related to the McCracken problem, was in multiple-choice format:

When evaluating an infix expression,

A. a stack is used to hold the operands until they are ready to be processed.
B. a stack is used to hold the operators until they are ready to be executed.
C. Two stacks are used one for the operators and another one for the operands.
D. Two queues are used one for the operators and another one for the operands.

An example of an apply-given question addressing the same type of problem is:

Show the steps involved in evaluating the following reverse-polish expression. For each step, show the operand stack and state the operation that is performed.

\[4 \ 1 \ 5 \ * \ 1 \ + \ 0 \ 5 \ + \ * \ +\]

The Nocturnal Avionics problem, described above in Section 4.1.1, has two parts, both of which are apply-select questions: one that requires the student to explain his or her design (an essay) and one that requires the student to write code.

6. CONCLUSIONS AND FUTURE WORK

We began by characterizing our dataset, an international repository of CS2 exams. These exams come from 14 institutions and span the years from 1973-2009. They include true-false, multiple-choice, short-answer, and essay questions, as well as questions where students are asked to draw diagrams or write code. We focused on the stack and hashing questions, about 19% of the questions (by points). Most of the hash questions involve analysis or tracing, while the largest group of stack questions (about half) involve writing code.

Based on these data, we then explored the question: What is the relative emphasis on implementation, interface, and application (for hashing and stacks) in CS2, and has it changed over time? We developed a classification scheme that sorted the questions based on their position relative to the interface, that is, whether they required knowledge of the implementation, only of the operations provided, or of the operations and how to apply them to solve a problem. Overall, we found that the greatest proportion of the questions (by points) involved implementation. Despite recent calls for a move to greater emphasis on applications, we found no clear trends in that direction. There are, however, clear institutional differences; one institution did exhibit a possible upward trend, and a second had an ongoing emphasis on applications.

In future work we will extend this research by looking at the other data structures (queues, graphs, trees, etc.) to see if they demonstrate the same patterns found with stacks and hashtable, especially in their use of application questions. We also plan to analyze what a typical data-structures exam looks like in terms of the structures covered, types of questions and difficulty presented. We intend to examine the exams for particularly interesting and novel types of questions that illustrate unique ways of testing concepts of data structures at the interface, implementation, and application levels. We will collect additional exams to broaden the representation of institutions and exams per year. Finally, we will report on the experience of analyzing a large dataset from a repository: lessons learned, assumptions needed, and improvements to the process.

ACKNOWLEDGMENTS

The authors would like all those who shared their final exams, and the NSF for supporting the DCER project. This material is based upon work supported in part by the National Science Foundation under grant number CNS 0734761. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[10] Liberal Arts Computer Science Consortium. A 2007 model curriculum for a liberal arts degree in Computer
[12] R. Lister, I. Box, B. Morrison, J. Tenenberg, and D. S. Westbrook. The dimensions of variation in the