Modern Data Structures: Experiences with a Flexible Approach to a Data Structures Course

Richard M. Stillman
rstillma@nova.edu
School of Computer and Information Sciences
Nova Southeastern University
Fort Lauderdale, FL 33314, USA

Alan R. Peslak
arp14@psu.edu
Information Sciences and Technology
Penn State University
Dunmore, PA 18512, USA

Abstract

Data structures retain a major place in the 2002 IS (Information Systems) Model Curriculum, but debate about teaching abstract data structures to computer and information systems students continues. The discussion generally centers on the relative merits of teaching how to program data structures versus how to use them. We propose a compromise approach in which students are introduced to both aspects. The capstone of the course is a final project where students are given the latitude to focus on developing and/or applying abstract data structures. Grades are based upon creativity and complexity. This approach allows each student to shape the educational experience to his or her own talents and professional needs. Experience with a group of 38 students of diverse backgrounds is presented. The validity and value of this final project are supported by the following trends that emerged from analyzing this experience. Students' grades on the final project correlated with their grades on other traditional assignments. Interestingly, those students in the upper one-third of the class tended to select the more difficult data structures to implement in their final project. Also, the 19 students with professional experience beyond entry-level employment were more likely to submit creative, rather than routine, final projects. The approach presented is seen as a success, ensuring that all students comprehend the basics of data structures, yet encouraging the more devoted students to excel.

Keywords: data structures, higher education, capstone, final project, information systems, active learning environment.

1. INTRODUCTION

The understanding of basic data structures still plays a central role in information systems curricula. In the 2002 model curriculum, IS (Information Systems) 2002.5 (Data, File and Object Structures) is a specified course for both IS majors and minors.

The course description (Gorgone et al, 2002) includes (*emphasis* added):

> **"IS 2002.5.** Programming, Data, File and Object Structures (Prerequisite: IS 2002.1)
>
> **Catalog:** This course presents object oriented and procedural software engineering methodologies in data definition and measurement, *abstract data type construction* and use in developing screen editors, reports and other IS applications using data structures including indexed files.
>
> **Scope:** This course provides an exposure to algorithm development, programming, computer concepts, and the *design and application of data and file structures*. It includes the use of logical and physical structures for both programs and data.
>
> **Topics:** *Data structures and representation:* characters, records, and files; precision of data; information representation, organization, and storage; algorithm development; programming control structures; program correctness, verification, and validation; file structures and representation. Programming in traditional and visual development environments that incorporate event-driven, object-oriented design.
>
> **Discussion:** *Specific data structures including arrays, records, stacks, queues, and trees will be created and used.* The course will provide an introduction to the use of predefined user interface components."

Data structures thus remain an important component of an IS curriculum. But there is some debate on how to teach data structures to undergraduate and beginning graduate students. Much has been written about the role of teaching abstract data structures in the curricula of computer science and information systems. The importance of introducing students to abstract data structures early in CS education has been discussed for several decades (Lang and Maruyama, 1989; Friedman and Koffman, 1976; Tremblay & Manohar, 1974).

Clearly, today's practitioners need to understand how to use abstract data structures, yet there is some question as to how deeply an undergraduate or early graduate-level CS or CIS course should delve into the implementation details. A description of an early data structures course that stressed implementation (Feldman, 1984) reported that many students actually used their data structure projects in their employment. Today, this is less likely. Modern object-oriented languages such as Java, C++ and C# come packaged with libraries of well-documented data structures which can be easily used without ever viewing the underlying code. The object oriented paradigm stresses modularity and code re-use. If we have students in essence re-write Java's collections classes from scratch, we may be continuing to create information technologists who are most comfortable rebuilding code that is already in place. Raymond Lister (Collins et al, 2003) suggests, "Software Engineering is moving away from emphasis on the creation of code, toward emphasis on components and code reuse. The teaching of data structures needs to adjust to that change." On the other hand, by building code from the ground up, students can gain a better appreciation of the packaged data structures they are using.

Conferring an understanding of what's under the hood of abstract data structures does have several advantages. A data structures course typically is the first course to follow the student's first object-oriented programming course. If nothing else, coding data structures helps sharpen a beginner's programming skills. But beyond this, building data structures illuminates one of the layers of opacity that separate the new programmer from the machine. The student who has coded even a simple abstract data structure develops an appreciation for the difficulty of making abstract code function reliably, the significance of edge cases, and the beauty of encapsulation and polymorphism. Lister et al (2004) point out that "data structures are a vehicle for developing thinking skills that are important and transferable beyond their immediate application to data structures… [and build an] awareness that the obvious or straightforward way to do things is often markedly inferior to clever ways that have been discovered…"

2. METHODS

The core master's-level data structures course at our major southeastern University lies at the root of a common pathway to graduate training in computer and information sciences. Students entering this course are assumed to have at least basic Java pro-

gramming skills, and students completing this course advance into classes in software engineering, object oriented applications, and client-server computing. Many of our students are professionals seeking to advance their knowledge and careers, and thus enrollees in the data structures course vary widely in their background knowledge and goals for their graduate education. Werth (1986) found that students' prior work experience demonstrated some correlation with performance in computer courses, and suggested that this may be due to motivation.

Our data structures course includes a major programming assignment that attempts to accommodate this diversity. This practical application approach is a compromise between those educators who feel that students should be taught only how to use data structures and those educators who would have their students create data structures.

The project requires the student to select any five abstract data structures, program them, and then program applications that use them. The exact nature of each application is entirely at the discretion of the student, as is the level and nature of the implementation of the data structure. Students are graded based upon the complexity and originality of their submissions.

This approach enables each student to mold the educational experience to his or her own skill set, and perhaps even more significantly, to his or her professional needs. For example, those students whose career plan will focus on developing systems from components can hone these skills by building applications. Those students who prefer to develop components would spend more time creating the data structures. And each of these students will do at least a little of both types of design during this course.

This final project is the capstone of the course, but the educational experience also includes five programming assignments taken from a standard data structures textbook, and two descriptive essay questions. In addition, a large portion of the course is traditional data structures text material and specific short answer assignments that explore students' knowledge of data structures concepts. As a result, it has been possible to compare students' grades on the traditional assignments with the grades achieved on the final project.

3. RESULTS

Based upon 38 submissions reviewed by the instructor, we have discerned clear qualitative differences among the students' approaches to this project.

Project Categories

Naturally, there was some variability within each submission, but in general, most submissions appeared to fit nicely into one of two categories:

Basic: These submissions included mostly data structures that rehashed the design of those provided by the instructor during the didactic portion of the course, especially those given in the textbook. The applications were rudimentary uses of the data structures.

Creative: Submissions in this group emphasized creative design of the ADTs, as, for example, building a class hierarchy, or using a sort algorithm different from the ones presented during the course. Some of these submissions effectively dropped assumptions about the data types that had been presented during the didactic portion of the course. These submissions therefore offered additional flexibility. For example, some of these ADTs permitted duplicate or null objects, intelligently handled exceptions rather than just throwing them back to the client, or allowed arbitrary object ordering methods. (We shall not discuss here the real-world problems with such relaxed constraints.) See the Appendix of this paper for excellent examples of what we considered creative applications.

Correlate of ADT Choice

Figure 1 shows the relationships between the students' specific choice of abstract data types for the final project and their overall standing in the class. The fractions add to more than 1 because each student submitted five ADTs.

Those students who were in the upper 1/3 of the class were more likely than the remainder of the class to have selected priority queues and recursive binary search implementations, which are the more abstract and

difficult ADTs. (The numbers in each row were too small to infer any other significant differences.) The project flexibility allowed the better students to select the more challenging ADTs.

Correlates of Performance

In addition to reviewing their grades in the two components of the data structures course, we also explored the students' experience levels. When we classify the quality of the students' submissions and consider the students employment backgrounds, interesting trends emerge.

Table 1 relates the project and assignment grades to the nature of the project and the student's professional background.

Project Category Versus Grade: The average grade on the 16 projects considered creative was 96.6 ± 4.7 compared to an average grade of 84.9 ± 16.1 on the 22 projects considered basic. Therefore, not unexpectedly, creative projects earned significantly higher grades (P = 0.003 by Student's *t*-test).

Project Category Versus Performance on Other Assignments: Of more interest, those students who submitted creative projects achieved significantly higher grades on the other assignments as well (93.7 ± 4.5 versus 87.4 ± 7.3 with P = 0.002).

Grades Versus Professional Experience: Half of the students had professional experience beyond entry-level employment. The experienced professionals performed marginally better in the assignments (91.7 versus 88.3), but significantly better on the final project (92.7 versus 87.0). Experience correlated significantly with the student's decision to implement (self-select) creative projects. But experience alone was not required for achieving higher grades. Students with little experience who exerted the effort to develop creative projects did similarly well in both project and course grades. We suggest that there is most likely a higher learning curve for those with less experience, but with self-motivation this learning curve was overcome. Those with less experience who did not make this effort excelled neither in the project nor in the other assignments.

4. DISCUSSION

In the varied environments of today's object oriented languages, programmers have to learn to recognize and adjust to subtle differences among data structure implementations (Fekete, 2002). Each textbook tends to focus on one particular implementation. Our capstone project allows the student to work out the subtleties of at least one other implementation.

Jarc (1994) recognized that students are often taught abstract data structures individually and in isolation. He suggested a more unified approach. Goldweber et al (1997) looked at the computer science curriculum from several different vantage points and, among other recommendations, emphasized the importance of pedagogic innovations that help students develop mature problem-solving strategies. Haddad (2002) found that newly hired, recent CS graduates, despite good theoretical knowledge, often lacked the ability to apply the principles in practice. Maurer (2002) pointed out that at the conclusion of a data structures course, students might have acquired nothing more than a set of disjoint bits of information about abstract structures without a clue about how all of it fits together. Solving real world problems and participating in hands-on exercises improve computer science students' reasoning skills (Parham, 2003). CS students require an active learning environment (Briggs, 2005). Budd (2006) talked of the benefits of active learning in a data structures course.

We present a practical and flexible approach to a final project for a basic data structures course (a required course in the 2002 Model Curriculum). Students are given wide latitude to develop and to apply abstract data structures. Grades are based upon creativity and complexity. This approach allows each student to shape the educational experience to his or her own talents and expectations.

It also provides a valid yardstick for grading. It could be suggested that the project grades only reflect the bias of the instructor, but grades on the final project correlated strongly with grades on the other assignments ($p < .001$). These assignments were purely objective, thus validating the knowledge transfer which has taken place during the semester and the allowing of students to create not only working data structures but

practical applications as well. Students in the upper one-third of the class were more likely to select the more abstract and difficult ADTs to implement for the final project.

Interestingly, those students with employment experience beyond entry-level jobs were more likely to submit creative final projects. We believe that the combination of traditional data structures course materials combined with a flexible capstone project allows for a better more comprehensive approach to teaching data structures.

Our approach compels the student to select data structures and apply them. Addressing Jarc (1994), our data structures are not taught in isolation. We believe we have adapted a problem-solving approach as suggested by Goldweber et al. (1997) and Parham (2003). We allow for applying principles in action (Haddad, 2002). Finally, we address Maurer's (2002) perceived need for a capstone, and Budd (2006) and Briggs (2005) active learning models. Data structures remains a vital component of IS education. Our approach is suggested to allow for a modern, flexible approach to meet the needs to the current professional environment.

5. REFERENCES

Bogoiavlenski, I. A., Clear, A. G., Davies, G., Flack, H., Myers, J. P., and Rasala, R. 1997. "Historical perspectives on the computing curriculum." *SIGCUE Outlook* 25, 4 (Oct. 1997), 94-111.

Briggs, T. 2005. "Techniques for active learning in CS courses." *J. Comput. Small Coll.* 21, 2 (Dec. 2005), 156-165.

Budd, T. A. 2006. "An active learning approach to teaching the data structures course." In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, Texas, USA, March 03 - 05, 2006). SIGCSE '06. ACM Press, New York, NY, 143-147.

Collins, W., Tenenberg, J., Lister, R., and Westbrook, S. 2003. "The role for framework libraries in CS2." In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, Navada, USA, February 19 - 23, 2003). SIGCSE '03. ACM Press, New York, NY, 403-404.

Fekete, A. 2002. "Teaching data structures with multiple collection class libraries." In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky, February 27 - March 03, 2002). SIGCSE '02. ACM Press, New York, NY, 396-400.

Feldman, M. B. 1984. "Abstract types, ADA packages, and the teaching of data structures." In *Proceedings of the Fifteenth SIGCSE Technical Symposium on Computer Science Education* L. N. Cassel and J. C. Little, Eds. SIGSCE '84. ACM Press, New York, NY, 183-189.

Friedman, F. L. and Koffman, E. B. 1976. "Some pedagogic considerations in teaching elementary programming using structured FORTRAN." In *Proceedings of the ACM SIGCSE-SIGCUE Technical Symposium on Computer Science and Education* (February 01 - 01, 1976). R. Colman and P. Lorton, Eds. ACM Press, New York, NY, 1-10.

Gorgone, J., Davis, G., Valacich, J., Topi, H., Feinstein, D., and Longenecker, H. (2002). "IS 2002 Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems." Accessed 5-25-2006. http://www.aisnet.org/Curriculum/IS2002-12-31.doc

Haddad, H. 2002. "Post-graduate assessment of CS students: experience and position paper." *J. Comput. Small Coll.* 18, 2 (Dec. 2002), 189-197.

Jarc, D. J. 1994. "Data structures: a unified view." *SIGCSE Bull.* 26, 2 (Jun. 1994), 2-4.

Lang, J. E. and Maruyama, R. K. 1989. "Teaching the abstract data type in CS2." In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education* (Louisville, Kentucky, United States, February 23 - 24, 1989). R. A. Barrett and M. J. Mansfield, Eds. SIGCSE '89. ACM Press, New York, NY, 71-73.

Lister, R., Box, I., Morrison, B., Tenenberg, J., and Westbrook, D. S. 2004. "The dimensions of variation in the teaching of data structures." In *Proceedings of the 9th Annual SIGCSE Conference on innovation and Technology in Computer Sci-*

ence Education (Leeds, United Kingdom, June 28 - 30, 2004). ITiCSE '04. ACM Press, New York, NY, 92-96.

Maurer, W. D. 2002. "A capstone unit for a data structures class." *J. Comput. Small Coll.* 17, 5 (Apr. 2002), 104-109.

Parham, J. R. 2003. "An assessment and evaluation of computer science education." *J. Comput. Small Coll.* 19, 2 (Dec. 2003), 115-127.

Tremblay, J. P. and Manohar, R. 1974. "A first course in discrete structures with applications to computer science." In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education* SIGCSE '74. ACM Press, New York, NY, 155-160.

Werth, L. H. 1986. "Predicting student performance in a beginning computer science class." In *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Ohio, United States, February 06 - 07, 1986). J. C. Little and L. N. Cassel, Eds. SIGCSE '86. ACM Press, New York, NY, 138-143.

Table & Figure

Table 1. Relationship Among Students' Professional Experience, Fundamental Approach to the Final Programming Project, and Performance in the Course.

Employment background	Approach to final project	N	Grades	
			Final Project	Assignments
Entry-level	Basic	16	85.5	86.8
	Creative	3	95.0	96.3
	Total	19	87.0	88.3
Experienced	Basic	6	83.3	88.9
	Creative	13	97.0	93.1
	Total	19	92.7	91.7

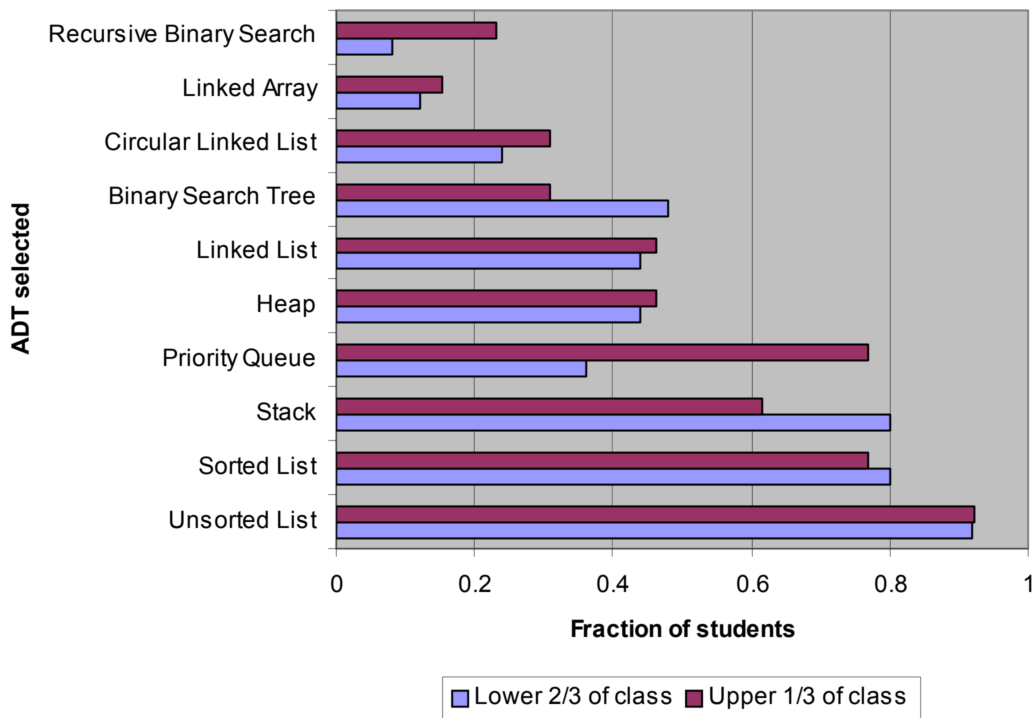

Figure 1. Relationship Between Students' Choices of Abstract Data Type (ADT) for the Final Project and Class Rank Based

Appendix

Here are two examples of what we classified as "creative" submissions for the final project.

Image Renderer

This Swing application graphically compares a variety of priority queue implementations. It displays a window consisting of several graphical components. Each graphical component is a dynamic display of an image as it is rendered pixel by pixel. Each Pixel object knows its color and position within the image. All 10,000 Pixel objects constituting each 100 x 100 image are shuffled randomly (using a static method in our List class) and stored in a Priority Queue data structure. They are then dequeued in order of their priorities and inserted into the developing image. In this application, pixels sequence in order of their rgb color, highest first. Because white is represented as 0ffffffh, it has highest priority and therefore white pixels render first, then red (0ff0000h), green (000ff00h), and finally blue (00000ffh). Intermediate colors (e.g., purple, 0ff00ffh) render in priority of their natural ordering. Each copy of the image is handled by a different priority queue implementation and all of these are processed in parallel, each by a different Thread running concurrently so that we can visually compare the performance of these data structures.

Figure 2 is a screen shot of ImageRenderer.java in action. Notice how sluggishly the Circular-LinkedQueue behaves compared to the others – it is still processing the image as this screen is captured. Also, notice that our original implementations compare favorably to those of Sun's ArrayList class. The balanced AVL tree seems to be the most efficient. (The picture is an Escher.)

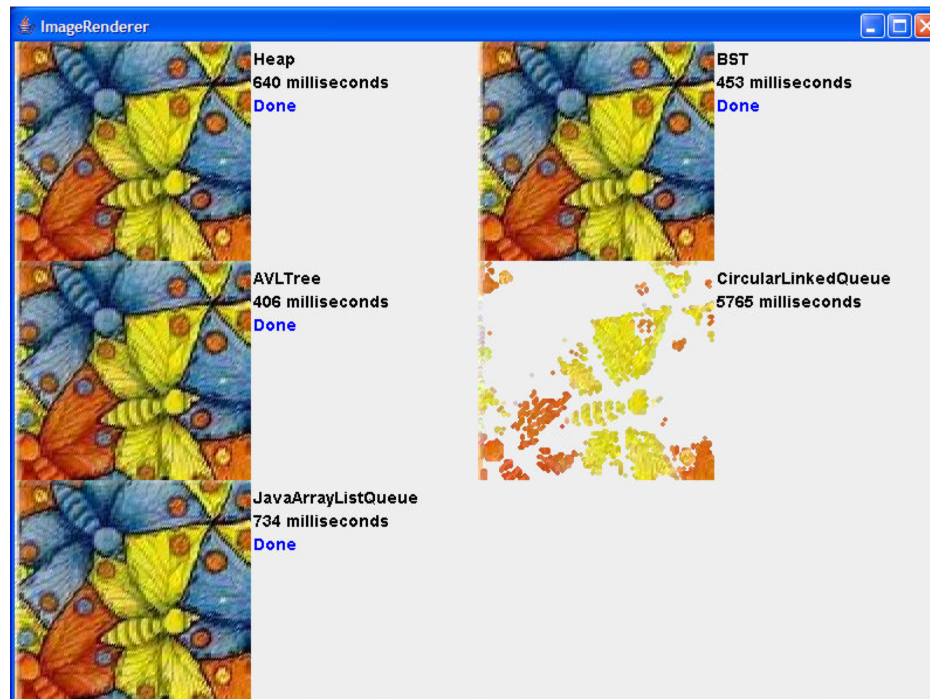

Figure 2. ImageRenderer, an application that graphically compares the performance of several implementations of a priority queue data structure.

Protein Taxonomy

This submission demonstrates an application of a binary search tree using a bioinformatics backdrop. The student explains the underlying biology:

> The evolutionary similarities among organisms can be estimated by looking at the structure of their proteins. Here's how. If we look at a protein common to a number of organisms, we note similar but not identical amino acid sequences. For example, the amino acid sequence G A L V (glycine – alanine – leucine – valine) looks similar to G A I V (glycine – alanine – isoleucine – valine) but quite different from V L A S (valine – leucine – alanine – serine). The question is how to quantify the similarities and differences. One answer is to determine how much editing we have to do to convert one sequence into the other. For example, by inserting a gap into each of the first two sequences, they align as follows:
>
> ```
> G A L – V
> | | | | |
> G A – I V
> ```
>
> We might say that these two sequences have an "edit distance" of two.

He then goes on to describe the application:

> ProteinTaxonomy.java starts by reading an input file consisting of a list of proteins. It parses each record in the file into a Protein object. The Protein object has a name (such as "Human" or "Gorilla") and an amino acid sequence (here averaging about 150 amino acids.) It calculates a score for each protein based upon the edit distance from an index protein (the one at the top of the list). These Protein objects are then inserted into a BinarySearchTree using a Comparator based upon edit distances.

Figure 3 shows the result of running ProteinTaxonomy on a file having the structure of the beta-globin protein[1] from each of a number of organisms. The resultant hierarchy of life forms – arranged only by analyzing the structure of this one protein – seems remarkably parallel to our intuition.

[1] The amino acid sequences came from the protein database Entrez at http://www.ncbi.nlm.nih.gov/Database/index.html

Protein Taxonomy

File: Globins.txt

The following proteins are ranked in order of similarity of their amino acid sequences to the first protein
using global alignment with the Needleman-Wunsch dynamic programming algorithm for pairwise linear sequence alignment.

Description	% similarity	Amino acid sequence
Human		mvhltpeeksavtalwgkvnvdevggealgrllvvypwtqrffesfgdlstpdavmgnpkvkahgkkvlgafsdglahldnlkgtfatlselhcdklhvdpenfrllgnvlvcvlahhfgk
Human	100	mvhltpeeksavtalwgkvnvdevggealgrllvvypwtqrffesfgdlstpdavmgnpkvkahgkkvlgafsdglahldnlkgtfatlselhcdklhvdpenfrllgnvlvcvlahhfgk
Gorilla	99	mvhltpeeksavtalwgkvnvdevggealgrllvvypwtqrffesfgdlstpdavmgnpkvkahgkkvlgafsdglahldnlkgtfatlselhcdklhvdpenfkllgnvlvcvlahhfgk
Saki	70	mvhltgeekaavtalwgkvnvdevggealgrllvvypwtqrffesfgdlsspdavmnnpkvkahgkkvlgafsdglahldnlkgtfaqlselhcdklhvdpenfrllgnvlvcvlahhfgkeftpqvqaayqkvvagvanalahkyh
Capuchin	69	mvhltaeeksavttlwgkvnvdevggealgrllvvypwtqrffdsfgdlstpdavmnnpkvkahgkkvlgafsdglthldnlkgtfaqlselhcdklhvdpenfrllgnvlvcvlahhfgkeftpqvqaayqkvvagvanalahkyh
Monkey	69	mvhltgdekaavtalwgkvnvdevggealgrllvvypwtqrffesfgdlstpdavmhnpkvkahgkkvlgafsdglahldnlkgtfaqlselhcdklhvdpenfrllgnvlvcvlaqhfgkeftpqvqaayqkvvagvanalahkyh
Chimpanzee	68	mvhltpeektavnalwgkvnvdavggealgrllvvypwtqrffesfgdlsspdavmgnpkvkahgkkvlgafsdglahldnlkgtfsqlselhcdklhvdpenfrllgnvlvcvlarnfgkeftpqvqaayqkvvagvanalahkyh
Marmoset	68	vhltgeeksavtalwgkvnvdevggealgrllvvypwtqrffesfgdlstpdavmnnpkvkahgkkvlgafsdglthldnlkgtfahlselhcdklhvdpenfrllgnvlvcvlahhfgkeftpvvqaayqkvvagvanalahkyh
Tarsius	64	mvhltadekaavtalwskvnvedvggealgrllvvypwtqrffdsfgdlstpaavmsnakvkahgkkvlnafsdgmahldnlkgtfaklselhcdklhvdpenfrllgnvlvcvlahhfgkqftpqlqaayqkvvagvaaalahkyh
Rabbit	63	mvhlsseeksavtalwgkvnveevggealgrllvvypwtqrffesfgdlssanavmnnpkvkahgkkvlaafseglshldnlkgtfaklselhcdklhvdpenfrllgnvlvivlshhfgkeftpqvqaayqkvvagvanalahkyh
Rat	58	mvhltdaekaavnglwgkvnpddvggealgrllvvypwtqryfdsfgdlssasaimgnpkvkahgkkvinafndglkhldnlkgtfahlselhcdklhvdpenfrllgnmivivlghhlgkeftpcaqaafqkvvagvasalahkyh
Lemur	58	tflltpeenghvtslwgkvnvekvggealgrllvvypwtqrffesfgdlsspdaimgnpkvkahgkkvlsafseglhhldnlkgtfaqlselhcvalhvdpenfkllgnvlvivlahhfgndfspqtqaafqkvvtgvanalahkyh
Mouse	57	mvhltdaekaavsclwgkvnsdevggealgrllvvypwtqryfdsfgdlssasaimgnakvkahgkkvitafndglnhldslkgtfaslselhcdklhvdpenfrllgnmivivlghhlgkdftpaaqaafqkvvagvatalahkyh
Sheep	56	mltaeekaavtgfwgkvkvdevgaealgrllvvypwtqrffehfgdlsnadavmnnpkvkahgkkvldsfsngmkhlddlkgtfaqlselhcdklhvdpenfrllgnvlvvvlarhhgneftpvlqadfqkvvagvanalahkyh
Bovine	54	ltaeekaavtafwgkvkvdevggealgrllvvypwtqrffesfgdlstadavmnnpkvkahgkkvldsfsngmkhlddlkgtfaalselhcdklhvdpenfkllgnvlvvvlarnfgkeftpvlqadfqkvvagvanalahryh
Goat	52	ltaeekaavtgfwgkvkvdevgaealgrllvvypwtqrffehfgdlssadavmnnakvkahgkkvldsfsngmkhlddlkgtfaqlselhcdklhvdpenfkllgnvlvvvlarhhgseftpllqaefqkvvagvanalahryh
Kangaroo	49	vhltaeeknaitslwgkvaieqtggealgrllivypwtsrffdhfgdlsnakavmanpkvlahgakvlvafgdaiknldnlkgtfaklselhcdklhvdpenfkllgnilviclaehfgkeftidtqvawqklvagvanalahkyh
Chicken	48	mvhwtaeekqlitglwgkvnvaecgaealarllivypwtqrffasfgnlssptailgnpmvrahgkkvltsfgdavknldniknttfsqlselhcdklhvdpenfrllgdiliivlaahfskdftpecqaawqklvrvvahalarkyh
Carp	35	mvewtdaersaiiglwgklnpdelgpqalarclivypwtqryfasfgnlsspaaimgnpkvaahgrtvmggleraiknmdnikatyaplsvmhseklhvdpdnfrlladcitvcaamkfgpsgfspnvqeawqkflsvvvsalcrqyh
Frog	23	mvnltakerqlitgtwskicaktlgkqalgsmlytypwtqryfssfgnlssieaifhnaavathgekvltsigeaikhmddikgyyaqlskyhsetlhvdpynfkrfcsctiismaqtlqedftpelqaafeklfaaiadalgkgyh
Tortoise	11	vltagdkanvktvwskvgshledygsetlerlfvvypstktyfphfdlhhdspqvrahgkkvlsalgeavnhiddipgalsklsdlhaqnlrvdpvnfkllnlcfvvvvgrhhptiltpevhvsldkflsavaqnltskyr
Anteater	10	vltdaerkevtslwgkasghaedygaealerlflsfpttktyfshmdlskgsahvrahgkkvadalttavghfndmdgalsdlsdlhahklrvdpvnfkllahcflvvlarhhpeeftpsahaamdkflsrvatvltskyr

Figure 3. Protein taxonomy, an application that shows how effective an abstraction an object comparator can be.