A Gentle Introduction to Linked Lists

Michael H. Goldwasser
Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007
goldwamh@slu.edu

ABSTRACT

We consider a pedagogy for introducing linked lists in the context of a non-programming, breadth-first introductory course. In short, linked lists are presented based upon their direct embedding in an underlying memory configuration. Though this approach to teaching linked lists is not original, it is surprisingly rare in a breadth-first context. Yet it affords a rich exploration of many key aspects of linked data structures. Furthermore, the coverage can be spiraled with many common aspects of a breadth-first introduction, effectively integrating topics such as memory management, data representation and algorithmic analysis.

Our treatment is coupled with newly developed software that allows students to fully investigate the depth of the subject via hands-on, non-programming experiences. Students can set and modify the contents of a displayed memory configuration, viewing the effect of those changes on a schematic diagram of the embedded linked list.

Categories and Subject Descriptors

E.2 [**Data Storage Representations**]: *linked representations*; E.1 [**Data Structures**]: *arrays; lists, stacks and queues; trees*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.4.2 [**Operating Systems**]: Storage Management—*main memory*

General Terms

Algorithms

1. INTRODUCTION

The context for this work is that of a breadth-first introduction to computer science course, akin to CS100B in the CC2001 Computer Science report [10]. The stated design for CS100B is "to provide students with an appreciation for and an understanding of the many different aspects of computer science." Ideally, the course implementation should offer students opportunities to interact with and explore such concepts in a meaningful way. Yet for a course which does not include a significant programming component, designing these activities is a challenge [13].

A linked list is a wonderful example of an aspect of computer science for inclusion in such a breadth-first course. It serves as an example of a non-trivial data structure. When contrasted with sequential storage of data in an array, linked lists can be used to illustrate the tradeoffs which arise when considering alternate data organizations. Algorithmic design can be introduced when formulating common list operations, as can preliminary algorithmic analysis when discussing their efficiencies. Furthermore as a dynamic, referential structure, linked lists naturally reinforce other common course topics such as that of memory addressing and management.

Therefore, we consider a pedagogy for introducing linked lists in a non-programming, breadth-first introductory course. We also introduce software which provides students a hands-on environment for investigating the depth and complexity of the subject. The paper is organized as follows. Section 2 contains a presentation of the pedagogy for linked lists, including an overview of the supporting software. In Section 3, we compare this pedagogy to existing treatments based upon a survey of common classroom texts as well as articles in the computer science education literature. A discussion of the pedagogy's benefits and a list of detailed learning objectives is provided in Section 4. We conclude in Section 5, with a discussion of further potential uses.

2. A PEDAGOGY FOR LINKED LISTS

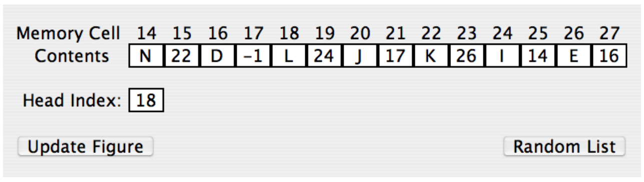
2.1 A Gentle Introduction

We present linked lists based upon their direct representation embedded in an underlying memory configuration. Students are shown a memory configuration, told the address of the cell which contains the first piece of data (i.e., the list "head"), told that each piece of data is followed, in the successive memory cell, by the explicit memory address of the cell containing the next piece of data (i.e., the "next pointer"), and finally that the list ends whenever "-1" is given as the next pointer (i.e., a "nil" reference).

For ease of presentation characters are used as data, to be clearly differentiated from references which are represented as integers. Figure 1 shows a sample memory configuration, representing the sequence L $\rightarrow$ I $\rightarrow$ N $\rightarrow$ K $\rightarrow$ E $\rightarrow$ D. Given a ten-minute introduction, students can generally tra-

Memory Cell:	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Contents:	N	22	D	-1	L	24	J	17	K	26	I	14	E	16

Figure 1: Memory configuration which includes a linked list headed at cell 18.



Memory Cell	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Contents	N	22	D	-1	L	24	J	17	K	26	I	14	E	16

Head Index: 18

Figure 2: Software display of the memory configuration corresponding to Figure 1.

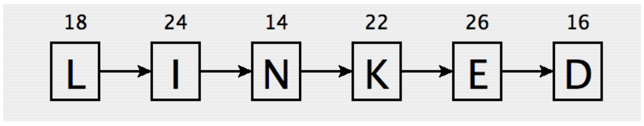


Figure 3: Software display of the schematic diagram corresponding to Figure 2.

verse such a list with ease. With additional instruction, students can learn the process for deleting a given item from the list, or inserting a new item into a desired place in the sequence.

Though we use this pedagogy in a breadth-first curriculum, it clearly draws upon the philosophies associated with the "hardware-first" strategy as outlined by CC2001 [10], namely to require "as little mystification as possible." This treatment could easily apply to the course CS111H, whose syllabus includes the category "Fundamental data structures: Primitive types; arrays; records; string and string processing; data representation in memory; pointers and references."

2.2 Supporting Software

To support this pedagogy, we introduce a software tool titled *A Gentle Introduction to Linked Lists*[1]. It is available, as a Java Applet, at `http://euler.slu.edu/~goldwasser/demos/linked`. The software could be used by the instructor when initially presenting the material, and later by students as an independent, active learning experience.

One panel of the software interface displays a given range of integer-indexed memory cells, the contents of which are editable. Figure 2 shows a sample such configuration, matching that of Figure 1. In a second panel, the software displays the traditional schematic view of a linked list based upon the current memory settings. The schematic is annotated with the underlying memory address for each node of the list. Such a diagram is shown in Figure 3.

The software affords students the opportunity to manipulate linked lists without reliance on the syntax of a programming language. At the same time, the interaction allows

[1]Our choice of name was directly inspired by Andrew Cumming's *A Gentle Introduction to SQL* [3], which also provides a great, interactive tool appropriate for use in a breadth-first context.

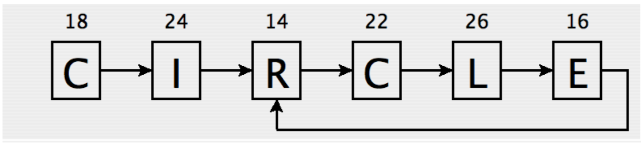


Figure 4: A schematic diagram including a circular reference.

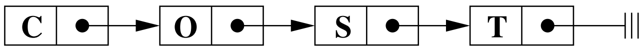


Figure 5: Typical schematic diagram of a linked list.

students to make common mistakes, while seeing the tangible outcome of such errors. The software design includes the following features:

- The schematic is only updated upon explicit request by the user. This allows for a period of self-test, where the user can predict the results before having it diagramed by the software. It also avoids the distraction of inconsistent intermediate stages, which would be seen during a real-time display of common list operations.
- Between successive updates, all edited cells are highlighted, drawing attention to the current set of manipulations.
- The cell contents are represented as strings, thus allowing for a mix of characters and integers, if desired.
- Any string which is not a well-defined memory reference is treated as a "nil" pointer.
- Circular references are detected and displayed, as shown in Figure 4.
- There is support for a pseudo-random generation of memory configurations, for convenient creation of further exercises.

3. SURVEY OF EXISTING PEDAGOGIES

In this section, we survey existing approaches for the teaching of linked lists, as well as existing software systems for supporting the learning experience.

In the context of a programming course, linked lists are generally introduced using schematic diagrams, such as those in Figures 5–6, together with a formal *syntactic* definition in a given programming language. The schematic figures often provide intuition and a shorthand for discussion, with a programming syntax used for concreteness. Through programming assignments, students gain insight into the beauty and complexity of linked structures.

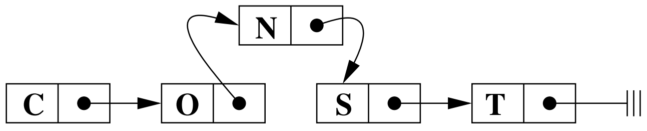


Figure 6: Typical schematic diagram after an insertion operation.

In the context of a breadth-first introductory course, most common texts avoid the topic entirely [6, 11], use a programming syntax for concreteness [17], or else rely purely on the schematic view of lists [5, 8, 15]. Unfortunately, the purely schematic treatment of the topic is too abstract and overly simplistic. We imagine an exam question which asks students to insert "N" between the "O" and "S" in Figure 5. Students will almost surely give the "correct" solution, as in Figure 6, because there is little room for mistakes. The view of a "pointer" is too loosely connected to the underlying memory configuration.

The only breadth-first textbook we find that broaches the topic of linked lists as represented in the underlying memory configuration is that of Brookshear [1]. Even it uses a purely schematic treatment of linked lists in the chapter body. However a series of chapter review problems introduces the connection between a linked list and an underlying memory configuration, asking students to traverse and even modify such lists. For more advanced data structure and algorithms courses, several texts [2, 4, 16] include treatment of linked lists embedded in arrays, generally based on its value as a legitimate implementation technique.

The impact of visualization and interaction in a learning environment is discussed often in the education literature (see [12] for example). In the context of algorithm visualization, Grissom *et. al.* [9] denote levels of student engagement, distinguishing between "simply viewing visualizations for a short period in the classroom" and "interacting directly with the visualizations for an extended period outside of the classroom."

Rambally [14] described an early system for automatically generating schematic views of linked lists based upon a *syntactic* description of an algorithm. Many subsequent such systems have been developed [7, 13, 18], each of which generates visualizations of linked data structures as they are manipulated using the syntax of a given programming language. A system described by Wu, Lee and Lin [18] included a "system memory window" which displayed the underlying memory configuration as well as a schematic view. The student manipulates the schematic view directly with mouse events, seeing the resulting changes in the memory configuration.

4. POTENTIAL LEARNING OBJECTIVES

Despite its simplicity, this pedagogy affords a very rich treatment of many common introductory concepts, and if desired, even some more advanced lessons. We detail many such potential learning objectives below, grouped according to major sub-disciplines and ordered generally from introductory to advanced.

Subtleties of Linked Structures

- Students can recognize the need for an explicit reference to the head of a list.
- Students can recognize the need for a conventional "nil" reference, which is clearly distinguishable from a legitimate memory reference.
- Students can appreciate the fragility of a linked structure, namely that a single corrupted reference can effectively cause the loss of the remainder of a structure.
- Students can recognize the special care required when operations are performed at the front of the list (or alternatively, students can explore the use of a sentinel node as the head of the list).
- Students can interpret the existence of a circular reference (i.e., back pointer) on the underlying representation of a list.
- Students might explore how two or more independent lists can be intertwined in memory.

Memory Usage and Management

- Students will have a clear understanding of a reference pointer represented explicitly as a memory address.
- Students must explore the issue of memory allocation in some fashion. As an item is inserted into the list, underlying memory must be found for a new node. When simulating many insertions and deletions, more formal treatments of memory management can be explored, such as
  - whether the memory for a deleted node could be reset to some recognizable 'garbage' state.
  - how garbage collection might be performed.
  - how a free list can be used to organize the available nodes.
- Students might explore the issue of memory fragmentation. Since a node is a two-cell memory structure, a single isolated memory cell cannot be effectively utilized.
- Because a node is represented as a multi-cell structure, students must understand alternatives for how such a multi-cell structure is referenced (e.g., perhaps the memory address of the *first* cell is used).
- Students might consider how a system should respond in the case that no available memory exists for an insertion, or more specifically, no consecutive pair of available cells.
- Students may explore the low-level encoding of pointers versus data as well as how context allows one to differentiate between the two.

Design and Analysis of Algorithms

- Once students have mastered the ad hoc simulation of basic operations, such as traversal, insertion and deletion, they can attempt to formalize the algorithm specification. In doing so, they can more carefully consider the precise order of operations and whether their ad hoc techniques rely on their own "local" memory.
- Students should have a clear understanding that a single insertion or deletion operation can be performed in constant time, that is, independent of the length of the list (though formal asymptotic notation need not be introduced).
- Students should consider the relative advantages and disadvantages of differing data organizations. In particular, a linked list can be contrasted to the sequential storage of a sequence in a traditional array. Points of contrast may include the $\Theta(1)$ vs. $\Omega(n)$ insertion and deletion times, the varying memory usage, and the effects of corrupted memory.
- Students can examine how a large piece of a sequence can be "cut" as a whole in constant time, as opposed to the linear performance if individually deleting each item. Furthermore, as the relative links of the removed portion remain intact, that piece can then be "pasted" as a whole elsewhere, again in constant time.
- If binary search were introduced in the context of arrays, students can explore why such a technique cannot be applied to searching a linked list.

5. FURTHER POSSIBILITIES

5.1 Additional Linked Structures

Though this paper specifically addresses singly-linked lists, the same pedagogy could be used to explore doubly-linked lists, binary trees, or other naturally linked data structures. For some such structures, e.g. doubly-linked lists, an issue is how best to visualize an *ill-formed* structure.

5.2 Use in Additional Courses

Though originally developed for a CS0 course, we have used the software as well when covering linked lists in a programming sequence. The treatment tends to provide a grounding intuition. Furthermore, it can be coupled with programming exercises involving linked lists embedded directly into an array.

Acknowledgement

We greatfully thank Dennis Bouvier for providing constructive feedback on the presentation of this paper.

6. REFERENCES

[1] J. G. Brookshear. *Computer Science: an Overview.* Addison-Wesley, Boston, Massachusetts, eighth edition, 2005.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, second edition, 2002.

[3] A. Cumming. A gentle introduction to SQL. http://sqlzoo.net/.

[4] N. Dale, D. T. Joyce, and C. Weems. *Object-Oriented Data Structures Using Java.* Jones and Bartlett Publishers, Sudbury, Massachusetts, 2002.

[5] N. Dale and J. Lewis. *Computer Science Illuminated.* Jones and Bartlett Publishers, Sudbury, Massachusetts, second edition, 2004.

[6] R. Decker and S. Hirshfield. *The Analytical Engine: An Introduction to Computer Science Using the Internet.* Brooks/Cole, Belmont, California, second edition, 2004.

[7] H. L. Dershem, R. L. McFall, and N. Uti. Animation of Java linked lists. In *Proc. 33rd SIGCSE Technical Symp. on Computer Science Education*, pages 53–57, Covington, Kentucky, Feb. 27–Mar. 3, 2002.

[8] B. A. Forouzan. *Foundations of Computer Science: From Data Manipulation to Theory of Computation.* Brooks/Cole, Pacific Grove, California, 2003.

[9] S. Grissom, M. F. McNally, and T. Naps. Algorithm visualization in CS education: comparing levels of student engagement. In S. Diehl and J. T. Stasko, editors, *Proc. 2003 ACM Symp. on Software Visualization*, pages 87–94, San Diego, California, June 2003.

[10] Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science Final Report.* IEEE Computer Society and the Association for Computing Machinery, Dec. 2001. http://www.computer.org/education/cc2001/final.

[11] K. F. Lauckner and M. D. Linter. *The Computer Continuum.* Prentice Hall, Upper Saddle River, New Jersey, second edition, 2001.

[12] T. Naps, G. Rößling, J. Anderson, S. C. W. Dann, R. Fleischer, B. Koldehofe, A. Korhonen, M. Kuittinen, C. Leska, L. Malmi, M. McNally, J. Rantakokko, and R. J. Ross. Evaluating the educational impact of visualization. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 124–136. ACM Press, 2003.

[13] M. J. Palakal, F. W. Myers, and C. L. Boyd. An interactive learning environment for breadth-first computing science curriculum. In *Proc. 29th SIGCSE Technical Symp. on Computer Science Education*, pages 1–5, Atlanta, Georgia, Feb. 26–Mar. 1, 1998.

[14] G. K. Rambally. Real-time graphical representations of linked data structures. In *Proc. 16th SIGCSE Technical Symp. on Computer Science Education*, pages 41–48, Mar. 1985.

[15] G. M. Schneider and J. L. Gersting. *An Invitation to Computer Science: Java Version.* Thomson Learning, Boston, Massachusetts, second edition, 2004.

[16] R. Sedgewick. *Algorithms in C.* Addison-Wesley, Reading, Massachusetts, third edition, 1998.

[17] R. L. Shackelford. *Introduction to Computing and Algorithms.* Addison-Wesley, Reading, Massachusetts, 1997.

[18] C.-C. Wu, G. C. Lee, and J. M.-C. Lin. Visualizing programming in recursion and linked lists. In *Proc. Third Australasian Conf. on Computer Science Education*, pages 180–186, University of Queensland, Australia, July 1998.