Binary Trees: A Challenge Problem For Separating Concerns

Scott M. Pike
The Ohio State University
Columbus, OH 43210-1277, USA
pike@cis.ohio-state.edu

ABSTRACT
The explosive growth in hardware performance over the last 50 years has buttressed a belief in Moore's Law that storage, bandwidth, and computing power will double approximately every 18 months. This unparalleled rate of progess in hardware technology has fueled an ever-growing industry of software applications. Unfortunately, the scale of current software has out-paced the development of tools and technologies for bringing software complexity within the tractable range of our limited human intellect. Research in separation of concerns has addressed this problem on many fronts, but it stands in need of challenge problems to motivate and evaluate new and existing approaches. This position paper outlines a formidable set of cross-cutting concerns pertaining to binary trees. The centrality of this data structure in diverse and conflicting application domains makes it an ideal challenge problem for current research.

1 INTRODUCTION

March 2001 witnessed the ACM1 conference "Beyond Cyberspace" held in San Jose, California (http://www.acm.org/acm1/). In a word, the conference was visionary. With 15 plenary speakers forecasting the next 50 years of computing, it was a veritable pep-rally for the knights-errant of technology looking to scale the castle walls of innovation. Like most crusades, there was a motivating belief behind which the movers and shakers could unite. In the case of ACM1, the belief was in Moore's Law: the unparalleled doubling of storage, bandwidth, and computing power roughly every 18 months. The innevitably smaller, faster, and cheaper computers of the future were reified as a holy grail within our reach. Astrophysicists, oceanographers, biogeneticists, and AI researchers alike galvanized the bandwagoneers of Moore's Law, drowning the naysayers of scale and software complexity in a wake of enthusiasm.

Conspicuously understated was the so-called "God's Law" expressed by William Buxton as the fact that human capacity for understanding is limited. Developing applications at the pace of Moore's Law dwarfs our raw ability to manage their scale. As Fred Brooks has observed, problems of scale in software cannot be solved simply by adding more people. Consider the humorous, yet apt, metaphor courtesy of Gerald Weinburg: you can't make a baby in one month by putting nine women on the job. The complexity of large-scale software outstrips the capacity of our unaided intellectual abilities, an asymmetry that underscores the importance of research in separation of concerns in any engineering discipline.

Following the spirit of quotation, let us not forget the words of Robert Browning: one's reach must exceed one's grasp, else what is heaven for? This certainly holds true for computational complexity; witness the halting problem. But what about intellectual complexity? Is it our fate to conceive of systems with practical solutions that we can neither manage nor grasp intellectually? The underlying goal in separation of concerns is to minimize the intellectual distance between the problem and the product. But how do we measure this distance?

Many innovative techniques to aid our understanding have been proposed, but the jury is still out with respect to their effectiveness. In addition to new software-engineering technologies, we need a basis of comparison: a suite of challenge problems to serve as an evaluation benchmark. Ultimately, we need *solved* problems, and enough of them so that powerful composition mechanisms can keep pace with the scale and complexity of future software. In service of this goal, we propose the following challenge problem, the centrality of which makes it an excellent touchstone for current and future research in advanced separation of concerns.

2 BINARY TREE APPLICATIONS

Binary trees are the most basic nonlinear structure in computer science. Their wide range of applicability derives from their fundamentally *hierarchical* nature, a property induced by their recursive definition. We can define a binary tree formally as "a finite set of nodes that either is empty, or consists of a root and the elements of two disjoint binary trees, called the left and right subtrees of the root" [5].

Draft submitted to Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001. Not for public distribution.

Binary trees have two primary application categories: representing hierarchical structures, and implementing efficient data storage and retrieval.

For applications that are *intrinsically* hierarchical, the structure of a binary tree is constrained. For example, the abstract parse tree for an arithmetic expression must capture both the syntax and the order of operator precedence. Restructuring the tree hierarchy is visible on the application level in the form of evaluation errors. In this category, binary trees must capture the structure of the application domain itself, so the tree structure is constrained by the hierarchical nature of the problem it models. Other applications in this category include attribute grammars, taxonomic classification systems, and game-theoretic strategies such as minimax and alpha-beta pruning.

The application category of data storage and retrieval is fundamentally different. Although the *data* in a binary tree are subject to application constraints, the *structure* of the tree itself is independent. Applications can index data *implicitly* by address, or *explicitly* by value, so data constraints depend only on the mode of access. For example, text files and virtual memories access data implicitly by line or word, whereas sets and look-up tables access data explicitly by comparing key values. In either case, the data are ordered linearly on the application level, rather than hierarchically, so the actual tree structure is invisible to the application. Various mechanisms – such as AVL trees [1], red-black trees [4], and splay trees [7] – exploit this fact by restructuring trees to support efficient access and update operations.

3 CONCERNS

We now turn to several concerns pertaining to binary trees. Many of these concerns are in conflict and crosscut several different domains. The challenge is not to satisfy every concern simultaneously; such is not possible. Rather, the challenge is to distill the design and implementation of binary trees so that different concerns are identified and encapsulated in distinct generic components that can be composed non-invasively into application-specific systems.

(1) Like many containers, binary trees should be able to separate data from structure, so that the basic operations on a binary tree are independent of the data it contains. The same underlying implementation should be able to support binary trees of integers, binary trees of payroll records, and binary trees of any other user-defined type.

(2) Binary trees are used in many diverse applications. Sometimes the structure of the tree must capture the structure of the problem domain, like parse trees. Other times, the structure can be manipulated to support efficient search. When the search mechanism is based on key values, as in a set or a look-up table, a binary search tree is appropriate. But binary search trees do not support all modes of search; for example, consider rank and order applications, or random-access sequences that index data by position. Despite these conflicting concerns of use, the underlying data structure is still a binary tree. So what is the least common denominator? That is, how can we specify a common underlying interface for binary trees that is independent of particular application concerns?

(3) A closely-related point is the desirability of separating the interface from both the representation and the implementation. In the obvious representation, each node in the tree contains three fields: a data field for the key, a pointer field to the root node of the left subtree, and a pointer field to the root node of the right subtree. In this representation, null pointer values indicate empty subtrees. This memory-level view of binary trees is so pervasive that it is commonly mistaken for being ubiquitous. It has been observed, however, that this conventional representation is not space-efficient, since most of the pointers are null [6]. An alternative representation known as a *threaded binary tree* utilizes this space more effectively. Instead of pointing to null, a leaf-node pointer gets linked to its inorder successor or predecessor. Variations of this representation scheme use preordering or postordering successorship to determine linkage. How can we decouple higher-level tree features and operations from different representation models so that changing the representation won't break the client algorithms?

(4) For the moment, let's consider a searching application. Balancing mechanisms for binary trees are complicated. A developer may initially choose a naive implementation for searching that ignores the performance gains of balanced trees in favor of a solution that is easy to verify with respect to correctness. Later, the developer may chose to tweak the performance of the overall system by substituting a clever balancing scheme. If the system is small enough to keep all data in main memory, the developer may choose AVL trees to achieve this desired gain in performance. Suppose now that the system is sucessful and amasses a large data set that exceeds the capacity of main memory. The AVL-tree performance plummets when the data set spills over into secondary storage. Consequently, the developer wants to change the implementation to use red-black trees instead, because they perform better in secondary storage by exploiting data locality. The foregoing maintenance activities are typical of successful systems. How can they be supported without invasive changes to the existing software?

(5) Although randomly built trees tend to be balanced on average [3], tree restructuring mechanisms must be employed to prevent degenerate performance. Many techniques for restructuring binary trees have b een developed over the years, including AVL-trees [1], splay trees [7], and symmetric binary B-trees (*a.k.a.*red-black trees) [2]. The definition of "balanced" varies from one technique to another. In the case of splay trees, where good performance is only guaranteed in an amortized sense, trees may actually become *less* balanced! Another problem is that restructuring mechanisms are complex enough to be conflated with — or identified

as — different *kinds* of data structures; that is, as "AVL" trees or "splay" trees or "red-black" trees. Thus far, the community has lacked a uniform characterization of what these techniques have in common. Consequently, practice has grafted tree algorithms directly onto particular balancing mechanisms. This instance of code coupling is a stumbling block to reusability. Is there a general characterization that separates a client's concern for efficient search from any particular restructuring mechanism?

(6) Sorting applications require a total ordering on key values to define a relevant notion of "less than of equal to" per application. Since this ordering may change from client to client, it should not be hard-coded into the implementation. Even basic types, like character strings, do not have obvious orderings; witness the differences between alphabetic, lexicographic, and favorite-words-first orderings. Similarly, binary search trees require a total ordering on key values to direct the path of search and update operations. The ordering must be a function of the key values, but it is not required for sorting. Still, the ordering cannot be hard-coded as the total ordering on memory-representations; several bit-configurations may represent the same *abstract* key value, in which case the correctness of the search and update operations would be coupled to a partcular representation scheme. How can implementations of sorting, search, and update operations be separated from the total ordering used to direct the path through the binary tree?

(7) Binary tree operations may have preconditions. During development, it is often advantageous to check preconditions, but where does the code belong: in the client or in the implementation? Sometimes the checking code is too costly (with respect to performance) to ship in the final product. How can checking code be included or excluded without resorting to invasive techniques?

(8) Successfully separating many of the concerns above will promote substitutability of new design decisions. How will substitution affect system correctness? In particular, will local changes require global re-analysis to determine overall system correctness? Reasoning about correctness should be porportional to the magnitude of the change to avoid invasive reasoning.

(9) Binary trees are often used in shared databases that need to maximize concurency for good performance. When a binary tree is being shared by multiple clients, update operations must lock parts of the tree. At the same time, balancing operations change the local structure of the tree. How can balancing and update operations be decoupled to maximize the currency of shared trees?

4 CONCLUSIONS

We propose the above set of conflicting concerns as a challenge problem to be analyzed, augmented, and eventually solved. At the workshop we will outline our own approach that leverages new template idioms to decouple independent concerns. Part of our approach is based on mechanism. In particular, we use template parameterization to separate concerns, and template instantiation to compose generic components into application-specific systems. The other part of our approach is based on new insights into the design and specification of binary tree components. Specifically, we characterize the least common denominator of all restructuring mechanisms used to support efficient search with binary trees. Our approach shows that some concerns can be separated simply by reconceptualizing the nature of the problem, without recourse to fancy and complicated tools that differ from cut-and-paste only in degree rather than in kind.

Our design approach will be useful to other researchers and practitioners in separation of concerns because it is independent of the particular technology (templates) that we have used to implement binary tree components. In addition formal specifications and abstract composition diagrams, we will include links to source code and use-case scenarios which illustrate the separation of concerns achieved. And, alas, there are other concerns to be considered; we look forward to attending the workshop where different viewpoints can be exchanged and triangulated to plot a common course.

REFERENCES

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[2] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[3] P. Flajolet and A. Odlyzko. The average height of binary trees and other simple trees. *Journal of Computer and System Sciences*, 25:171–213, 1982.

[4] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, 16–18 October 1978. IEEE.

[5] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Third Edition*. Addison–Wesley, Reading, MA, 1997.

[6] A. J. Perlis and C. Thornton. Symbol manipulation by threaded lists. *Communications of the ACM*, 3(4):195–204, April 1960.

[7] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.