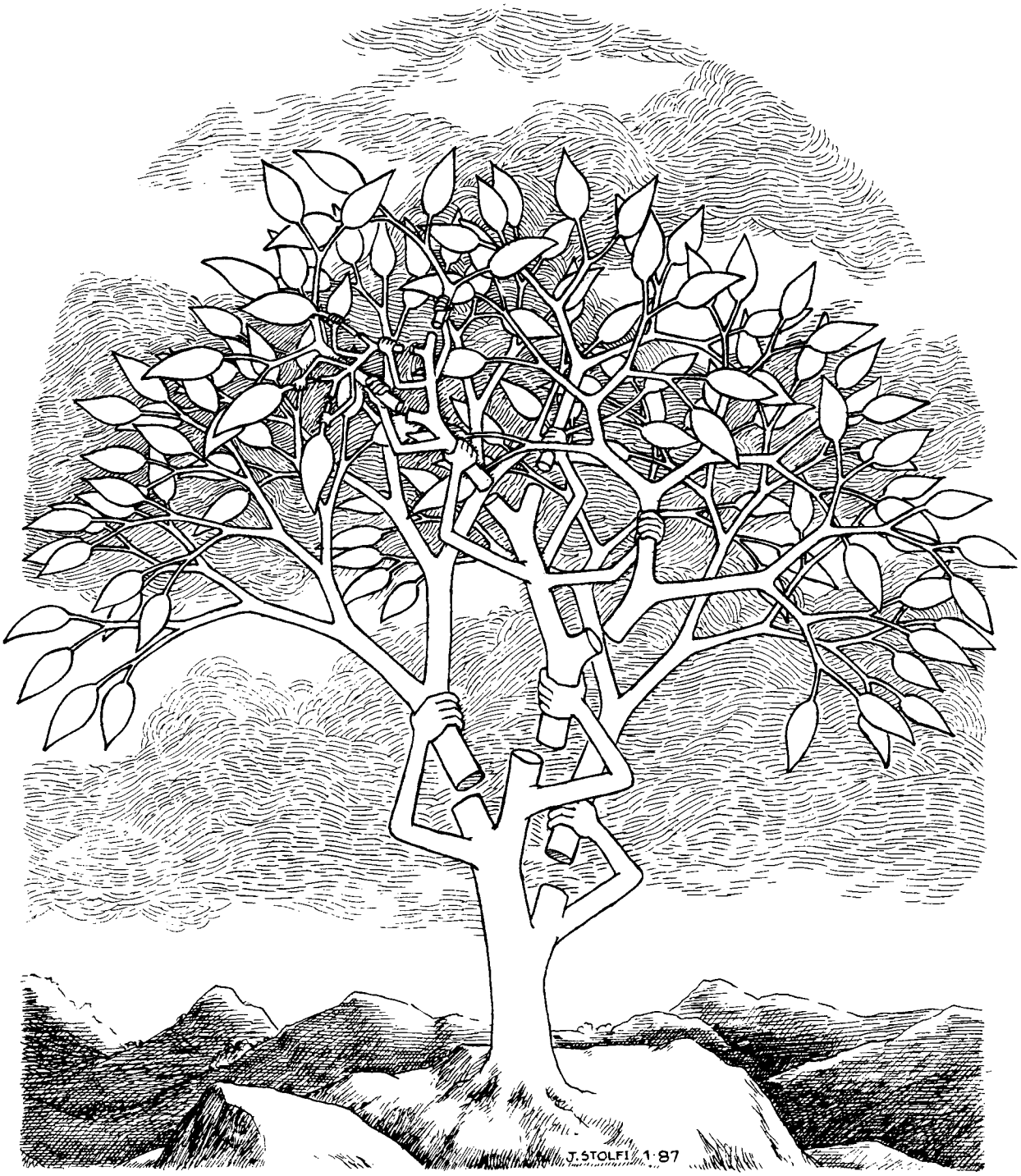


A Self-Adjusting Search Tree

ALGORITHM DESIGN

The quest for efficiency in computational methods yields not only fast algorithms, but also insights that lead to elegant, simple, and general problem-solving methods.

ROBERT E. TARJAN

I was surprised and delighted to learn of my selection as corecipient of the 1986 Turing Award. My delight turned to discomfort, however, when I began to think of the responsibility that comes with this great honor: to speak to the computing community on some topic of my own choosing. Many of my friends suggested that I preach a sermon of some sort, but as I am not the preaching kind, I decided just to share with you some thoughts about the work I do and its relevance to the real world of computing.

Most of my research deals with the design and analysis of efficient computer algorithms. The goal in this field is to devise problem-solving methods that are as fast and use as little storage as possible. The efficiency of an algorithm is measured not by programming it and running it on an actual computer, but by performing a mathematical analysis that gives bounds on its potential use of time and space. A theoretical analysis of this kind has one obvious strength: It is independent of the programming of the algorithm, the language in which the program is written, and the specific computer on which the program is run. This means that conclusions derived from such an analysis tend to be of broad applicability. Furthermore, a theoretically efficient algorithm is generally efficient in practice (though of course not always).

But there is a more profound dimension to the design of efficient algorithms. Designing for theoretical efficiency requires a concentration on the important aspects of a problem, so as to avoid redundant computations and to design data structures that exactly represent the information needed to solve the problem. If this approach is successful, the result is not only an efficient algorithm, but a collection of insights and methods extracted from the design process that can be transferred to other problems. Since the problems considered by theoreticians are generally abstractions of real-world problems, it is these insights and general methods that are of most value to practitioners, since they provide tools that can be used to build solutions to real-world problems.

I shall illustrate algorithm design by relating the historical contexts of two particular algorithms. One is a graph algorithm, for testing the planarity of a graph that I developed with John Hopcroft. The other is a data structure, a self-adjusting form of search tree that I devised with Danny Sleator.

I graduated from CalTech in June 1969 with a B.S. in mathematics, determined to pursue a Ph.D., but undecided about whether it should be in mathematics or computer science. I finally decided in favor of computer science and enrolled as a graduate student at Stanford in the fall. I thought that as a computer scientist I could use my mathematical skills to solve problems of more immediate practical interest than the problems posed in pure mathematics. I hoped to do research in artificial intelligence, since I wished to understand the way reasoning, or at least mathematical reasoning, works. But my course adviser at Stanford was Don Knuth, and I think he had other plans for my future: His first advice to me was to read Volume 1 of his book, *The Art of Computer Programming*.

By June 1970 I had successfully passed my Ph.D.

qualifying examinations, and I began to cast around for a thesis topic. During that month John Hopcroft arrived from Cornell to begin a sabbatical year at Stanford. We began to talk about the possibility of developing efficient algorithms for various problems on graphs.

As a measure of computational efficiency, we settled on the worst-case time, as a function of the input size, of an algorithm running on a sequential random-access machine (an abstraction of a sequential general-purpose digital computer). We chose to ignore constant factors in running time, so that our measure could be independent of any machine model and of the details of any algorithm implementation. An algorithm efficient by this measure tends to be efficient in practice. This measure is also analytically tractable, which meant that we would be able to actually derive interesting results about it.

Other approaches we considered have various weaknesses. In the mid 1960s, Jack Edmonds stressed the distinction between polynomial-time and non-polynomial-time algorithms, and although this distinction led in the early 1970s to the theory of NP-completeness, which now plays a central role in complexity theory, it is too weak to provide much guidance for choosing algorithms in practice. On the other hand, Knuth practiced a style of algorithm analysis in which constant factors and even lower order terms are estimated. Such detailed analysis, however, was very hard to do for the sophisticated algorithms we wanted to study, and sacrifices implementation independence. Another possibility would have been to do average-case instead of worst-case analysis, but for graph problems this is very hard and perhaps unrealistic: Analytically tractable average-case graph models do not seem to capture important properties of the graphs that commonly arise in practice.

Thus, the state of algorithm design in the late 1960s was not very satisfactory. The available analytical tools lay almost entirely unused; the typical content of a paper on a combinatorial algorithm was a description of the algorithm, a computer program, some timings of the program on sample data, and conclusions based on these timings. Since changes in programming details can affect the running time of a computer program by an order of magnitude, such conclusions were not necessarily justified. John and I hoped to help put the design of combinatorial algorithms on a firmer footing by using worst-case running time as a guide in choosing algorithmic methods and data structures.

The focus of our activities became the problem of testing the planarity of a graph. A graph is planar if it can be drawn in the plane so that each vertex becomes a point, each edge becomes a simple curve joining the appropriate pair of vertices, and no two edges touch except at a common vertex (see Figure 1).

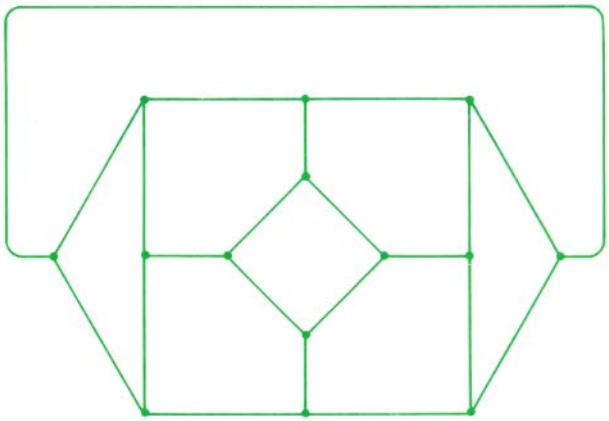


FIGURE 1. A Planar Graph

A beautiful theorem by Casimir Kuratowski states that a graph is planar if and only if it does not contain as a subgraph either the complete graph on five vertices (K_5), or the complete bipartite graph on two sets of three vertices ($K_{3,3}$) (see Figure 2).

Unfortunately, Kuratowski's criterion does not lead in any obvious way to a practical planarity test. The known efficient ways to test planarity involve actually trying to embed the graph in the plane. Either the embedding process succeeds, in which case the graph is planar, or the process fails, in which case the graph is nonplanar. It is not neces-

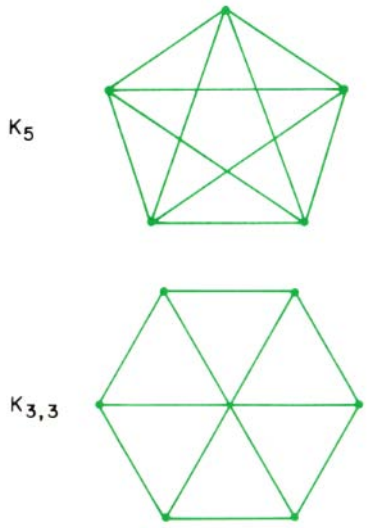


FIGURE 2. The "Forbidden" Subgraphs of Kuratowski

sary to specify the geometry of an embedding; a specification of its topology will do. For example, it is enough to know the clockwise ordering around each vertex of its incident edges.

Louis Auslander and Seymour Parter formulated a planarity algorithm in 1961 called the path addition method. The algorithm is easy to state recursively: Find a simple cycle in the graph, and then remove this cycle to break the rest of the graph into segments. (In a planar embedding, each segment must lie either completely inside or completely outside the embedded cycle; certain pairs of segments are constrained to be on opposite sides of the cycle. See Figure 3.) Test each segment together with the cycle for planarity by applying the algorithm recursively. If each segment passes this planarity test, determine whether the segments can be assigned to the inside and outside of the cycle in a way that satisfies all the pairwise constraints. If so, the graph is planar.

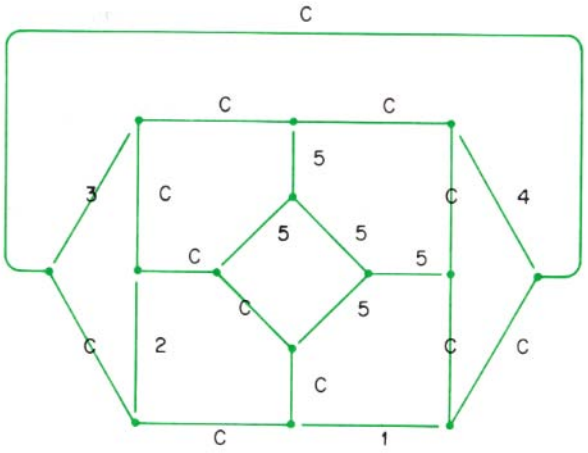
As noted by A. Jay Goldstein in 1963, it is essential to make sure that the algorithm chooses a cycle that produces two or more segments; if only one segment is produced, the algorithm can loop forever. Rob Shirey proposed a version of the algorithm in 1969 that runs in $O(n^3)$ time on an n-vertex graph. John and I worked to develop a faster version of this algorithm.

A useful fact about planar graphs is that they are sparse: Of the $n(n-1)/2$ edges that an n-vertex graph can contain, a planar graph can contain only at most $3n-6$ of them (if $n \geq 3$). Thus John and I hoped (audaciously) to devise an $O(n)$-time planarity test. Eventually, we succeeded.

The first step was to settle on an appropriate graph representation, one that takes advantage of sparsity. We used a very simple representation, consisting of a list, for each vertex, of its incident edges. For a possibly planar graph, such a representation is $O(n)$ in size.

The second step was to discover how to do a necessary bit of preprocessing. The path addition method requires that the graph be biconnected (that is, have no vertices whose removal disconnects the graph). If a graph is not biconnected, it can be divided into maximal biconnected subgraphs, or biconnected components. A graph is planar if and only if all of its biconnected components are planar. Thus planarity can be tested by first dividing a graph into its biconnected components and then testing each component for planarity. We devised an algorithm for finding the biconnected components of an n-vertex, m-edge graph in $O(n+m)$ time. This algorithm uses depth-first search, a systematic way of exploring a graph and visiting each vertex and edge.

We were now confronted with the planarity test



Segments 1 and 2 must be embedded on opposite sides of C, as must segments 1 and 3, 1 and 4, 2 and 5, 3 and 5, and 4 and 5.

FIGURE 3. Division of the Graph in Figure 1 by Removal of Cycle C

itself. There were two main problems to be solved. If the path addition method is formulated iteratively instead of recursively, the method becomes one of embedding an initial cycle and then adding one path at a time to a growing planar embedding. Each path is disjoint from earlier paths except at its end vertices. There is in general more than one way to embed a path, and embedding of later paths may force changes in the embedding of earlier paths. We needed a way to divide the graph into paths and a way to keep track of the possible embeddings of paths so far processed.

After carefully studying the properties of depth-first search, we developed a way to generate paths in $O(n)$ time using depth-first search. Our initial method for keeping track of alternative embeddings used a complicated and ad hoc nested structure. For this method to work correctly, paths had to be generated in a specific, dynamically determined order. Fortunately, our path generation algorithm was flexible enough to meet this requirement, and we obtained a planarity algorithm that runs in $O(n \log n)$ time.

Our attempts to reduce the running time of this algorithm to $O(n)$ failed, and we turned to a slightly different approach, in which the first step is to generate all the paths, the second step is to construct a graph representing their pairwise embedding constraints, and the third step is to color this constraint graph with two colors, corresponding to the two possible ways to embed each path. The problem with

this approach is that there can be a quadratic number of pairwise constraints. Obtaining a linear time bound requires computing explicitly only enough constraints so that their satisfaction guarantees that all remaining constraints are satisfied as well. Working out this idea led to an $O(n)$-time planarity algorithm, which became the subject of my Ph.D. dissertation. This algorithm is not only theoretically efficient, but fast in practice: My implementation, written in Algol W and run on an IBM 360/67, tested graphs with 900 vertices and 2694 edges in about 12 seconds. This was about 80 times faster than any other claimed time bound I could find in the literature. The program is about 500 lines long, not counting comments.

This is not the end of the story, however. In preparing a description of the algorithm for journal publication, we discovered a way to avoid having to construct and color a graph to represent pairwise embedding constraints. We devised a data structure, called a pile of twin stacks, that can represent all possible embeddings of paths so far processed and is easy to update to reflect the addition of new paths. This led to a simpler algorithm, still with an $O(n)$ time bound. Don Woods, a student of mine, programmed the simpler algorithm, which tested graphs with 7000 vertices in 8 seconds on an IBM 370/168. The length of the program was about 250 lines of Algol W, of which planarity testing required only about 170, the rest being used to actually construct a planar representation.

From this research we obtained not only a fast planarity algorithm, but also an algorithmic technique (depth-first search) and a data structure (the pile of twin stacks) useful in solving many other problems. Depth-first search has been used in efficient algorithms for a variety of graph problems; the pile of twin stacks has been used to solve problems in sorting and in recognizing codes for planar self-intersecting curves.

Our algorithm is not the only way to test planarity in $O(n)$ time. Another approach, by Abraham Lempel, Shimon Even, and Israel Cederbaum, is to build an embedding by adding one vertex and its incident edges at a time. A straightforward implementation of this algorithm gives an $O(n^2)$ time bound. When John and I were doing our research, we saw no way to improve this bound, but later work by Even and myself and by Kelly Booth and George Lueker yielded an $O(n)$-time version of this algorithm. Again, the method combines depth-first search, in a preprocessing step to determine the order of vertex addition, with a complicated data structure, the PQ-tree of Booth and Lueker, for keeping track of possible embeddings. (This data structure itself has several other applications.)

There is yet a third $O(n)$-time planarity algorithm. I only recently discovered that a novel form of search tree called a finger search tree, invented in 1977 by Leo Guibas, Mike Plass, Ed McCreight, and Janet Roberts, can be used in the original planarity algorithm that John and I developed to improve its running time from $O(n \log n)$ to $O(n)$. Deriving the time bound requires the solution of a divide-and-conquer recurrence. Finger search trees had no particularly compelling applications for several years after they were invented, but now they do, in sorting and computational geometry, as well as in graph algorithms.

Ultimately, choosing the correct data structures is crucial to the design of efficient algorithms. These structures can be complicated, and it can take years to distill a complicated data structure or algorithm down to its essentials. But a good idea has a way of eventually becoming simpler and of providing solutions to problems other than those for which it was intended. Just as in mathematics, there are rich and deep connections among the apparently diverse parts of computer science.

I want to shift the emphasis now from graph algorithms to data structures by reflecting a little on whether worst-case running time is an appropriate measure of a data structure's efficiency. A graph algorithm is generally run on a single graph at a time. An operation on a data structure, however, does not occur in isolation; it is one in a long sequence of similar operations on the structure. What is important in most applications is the time taken by the entire sequence of operations rather than by any single operation. (Exceptions occur in real-time applications where it is important to minimize the time of each individual operation.)

We can estimate the total time of a sequence of operations by multiplying the worst-case time of an operation by the number of operations. But this may produce a bound that is so pessimistic as to be useless. If in this estimate we replace the worst-case time of an operation by the average-case time of an operation, we may obtain a tighter bound, but one that is dependent on the accuracy of the probability model. The trouble with both estimates is that they do not capture the correlated effects that successive operations can have on the data structure. A simple example is found in the pile of twin stacks used in planarity testing: A single operation among a sequence of n operations can take time proportional to n, but the total time for n operations in a sequence is always $O(n)$. The appropriate measure in such a situ-

ation is the amortized time, defined to be the time of an operation averaged over a worst-case sequence of operations.[1] In the case of a pile of twin stacks, the amortized time per operation is $O(1)$.

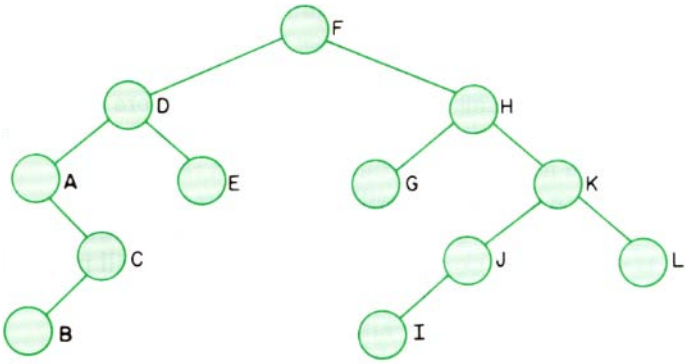
A more complicated example of amortized efficiency is found in a data structure for representing disjoint sets under the operation of set union. In this case, the worst-case time per operation is $O(\log n)$, where n is the total size of all the sets, but the amortized time per operation is for all practical purposes constant but in theory grows very slowly with n (the amortized time is a functional inverse of Ackermann's function).

These two examples illustrate the use of amortization to provide a tighter analysis of a known data structure. But amortization has a more profound use. In designing a data structure to reduce amortized running time, we are led to a different approach than if we were trying to reduce worst-case running time. Instead of carefully crafting a structure with exactly the right properties to guarantee a good worst-case time bound, we can try to design simple, local restructuring operations that improve the state of the structure if they are applied repeatedly. This approach can produce "self-adjusting" or "self-organizing" data structures that adapt to fit their usage and have an amortized efficiency close to optimum among a broad class of competing structures.

The splay tree, a self-adjusting form of binary search tree that Danny Sleator and I developed, is one such data structure. In order to discuss it, I need first to review some concepts and results about search trees. A binary tree is either empty or consists of a node called the root and two node-disjoint binary trees, called the left and right subtrees of the root. The root of the left (right) subtree is the left (right) child of the root of the tree. A binary search tree is a binary tree containing the items from a totally ordered set in its nodes, one item per node, with the items arranged in symmetric order; that is, all items in the left subtree are less than the item in the root, and all items in the right subtree are greater (see Figure 4).

A binary search tree supports fast retrieval of the items in the set it represents. The retrieval algorithm, based on binary search, is easy to state recursively. If the tree is empty, stop: The desired item is not present. Otherwise, if the root contains the desired item, stop: The item has been found. Otherwise, if the desired item is greater than the one in the root, search the left subtree; if the desired item is less than the one in the root, search the right subtree. Such a search takes time proportional to the depth of the desired item. defined to be the number of nodes examined during the search. The worst-case search time is proportional to the depth of the tree, defined to be the maximum node depth.

Other efficient operations on binary search trees include inserting a new item and deleting an old one (the tree grows or shrinks by a node, respectively). Such trees provide a way of representing static or dynamically changing sorted sets. Although in many situations a hash table is the preferred representation, since it supports retrieval in $O(1)$ time on the average, a search tree is the data structure of choice if the ordering among items is important. For example, in a search tree it is possible in a single search to find the largest item smaller than a given one, something that in a hash table requires examining all the items. Search trees also support even more complicated operations, such as retrieval of all items between a given pair of items (range retrieval) and concatenation and splitting of lists.

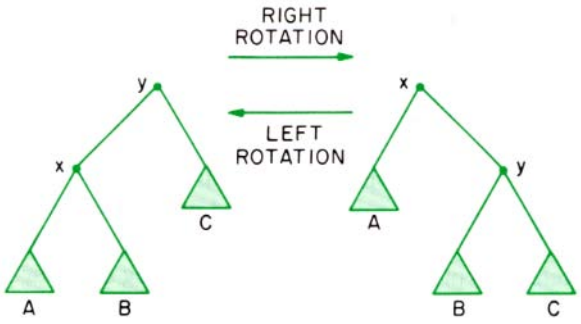


The items are ordered alphabetically.

FIGURE 4. A Binary Search Tree

An n-node binary tree has some node of depth at least $\log_2 n$; thus, for any binary search tree the worst-case retrieval time is at least a constant times $\log n$. One can guarantee an $O(\log n)$ worst-case retrieval time by using a balanced tree. Starting with the discovery of height-balanced trees by Georgiĭ Adelson-Velskii and Evgeniĭ Landis in 1962, many kinds of balanced trees have been discovered, all based on the same general idea. The tree is required to satisfy a balance constraint, some local property that guarantees a depth of $O(\log n)$. The balance constraint is restored after an update, such as an insertion or deletion, by performance of a sequence of local transformations on the tree. In the case of binary trees, each transformation is a rotation, which takes constant time, changes the depths of certain

[1] See R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Math. 6*, 2 (Apr. 1985), 306–318, for a thorough discussion of this concept.



The right rotation is at node x, and the inverse left rotation at node y. The triangles denote arbitrary subtrees, possibly empty. The tree shown can be part of a larger tree.

FIGURE 5. A Rotation in a Binary Search Tree

nodes, and preserves the symmetric order of the items (see Figure 5).

Although of great theoretical interest, balanced search trees have drawbacks in practice. Maintaining the balance constraint requires extra space in the nodes for storage of balance information and requires complicated update algorithms with many cases. If the items in the tree are accessed more or less uniformly, it is simpler and more efficient in practice not to balance the tree at all; a random sequence of insertions will produce a tree of depth $O(\log n)$. On the other hand, if the access distribution is significantly skewed, then a balanced tree will not minimize the total access time, even to within a constant factor. To my knowledge, the only kind of balanced tree extensively used in practice is the *B*-tree of Rudolph Bayer and Ed McCreight, which generally has many more than two children per node and is suitable for storing very large sets of data on secondary storage media such as disks. *B*-trees are useful because the benefits of balancing increase with the number of children per node and with the corresponding decrease in maximum depth. In practice, the maximum depth of a *B*-tree is a small constant (three or four).

The invention of splay trees arose out of work I did with two of my students, Sam Bent and Danny Sleator, at Stanford in the late 1970s. I had returned to Stanford as a faculty member after spending some time at Cornell and Berkeley. Danny and I were attempting to devise an efficient algorithm to compute maximum flows in networks. We reduced this problem to one of constructing a special kind of search tree, in which each item has a positive weight and the time it takes to retrieve an item depends on its weight, heavier items being easier to access than lighter ones. The hardest requirement to meet was the capacity for performing drastic update operations fast; each tree was to represent a list, and we needed to allow for list splitting and concatenation. Sam, Danny, and I, after much work, succeeded in devising an appropriate generalization of balanced search trees, called biased search trees, which became the subject of Sam's Ph.D. thesis. Danny and I combined biased search trees with other ideas to obtain the efficient maximum flow algorithm we had been seeking. This algorithm in turn was the subject of Danny's Ph.D. thesis. The data structure that forms the heart of this algorithm, called a dynamic tree, has a variety of other applications.

The trouble with biased search trees and with our original version of dynamic trees is that they are complicated. One reason for this is that we had tried to design these structures to minimize the worst-case time per operation. In this we were not entirely successful; update operations on these structures only have the desired efficiency in the amortized sense. After Danny and I both moved to AT&T Bell Laboratories at the end of 1980, he suggested the possibility of simplifying dynamic trees by explicitly seeking only amortized efficiency instead of worst-case efficiency. This idea led naturally to the problem of designing a "self-adjusting" search tree that would be as efficient as balanced trees but only in the amortized sense. Having formulated this problem, it took us only a few weeks to come up with a candidate data structure, although analyzing it took us much longer (the analysis is still incomplete).

In a splay tree, each retrieval of an item is followed by a restructuring of the tree, called a splay operation. (Splay, as a verb, means "to spread out.") A splay moves the retrieved node to the root of the tree, approximately halves the depth of all nodes accessed during the retrieval, and increases the depth of any node in the tree by at most two. Thus, a splay makes all nodes accessed during the retrieval much easier to access later, while making other nodes in the tree at worst a little harder to access.

The details of a splay operation are as follows (see Figure 6): To splay at a node x, repeat the following step until node x is the tree root.

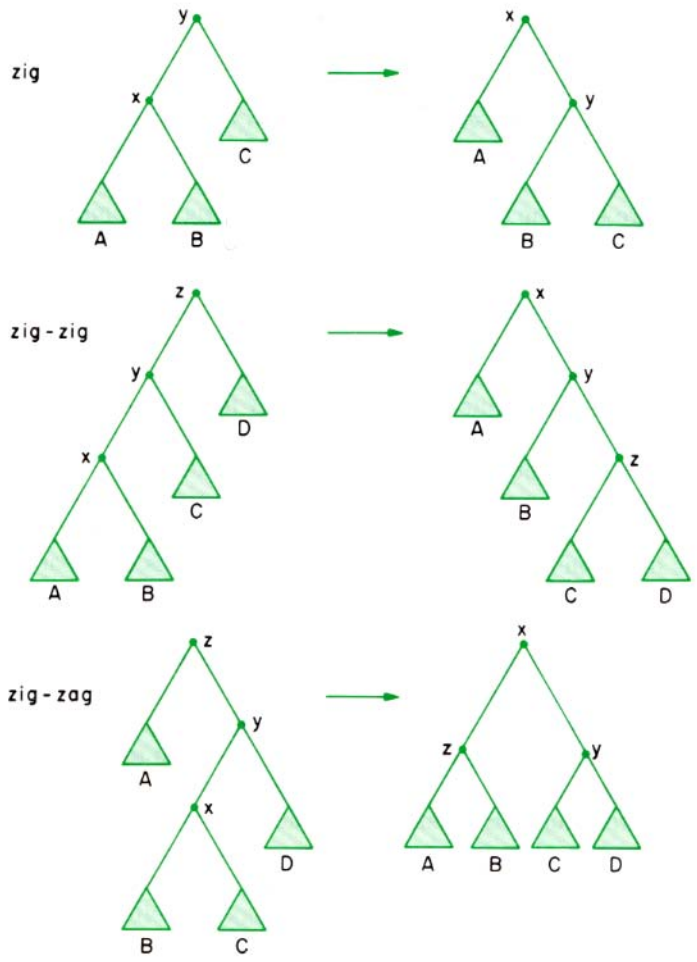
Splay Step. Of the following three cases, apply the appropriate one:

Zig case. If x is a child of the tree root, rotate at x (this case is terminal).

Zig-zig case. If x is a left child and its parent is a left child, or if both are right children, rotate at the parent of x and then at x.

Zig-zag case. If x is a left child and its parent is a right child, or vice versa, rotate at x and then again at x.

As Figure 7 suggests, a sequence of costly retrievals in an originally very unbalanced splay tree will



In the zig-zig and zig-zag cases, the tree shown can be part of a larger tree.

FIGURE 6. The Cases of a Splay Step

quickly drive it into a reasonably balanced state. Indeed, the amortized retrieval time in an n-node splay tree is $O(\log n)$. Splay trees have even more striking theoretical properties. For an arbitrary but sufficiently long sequence of retrievals, a splay tree is as efficient to within a constant factor as an optimum static binary search tree expressly constructed to minimize the total retrieval time for the given sequence. Splay trees perform as well in an amortized sense as biased search trees, which allowed Danny and I to obtain a simplified version of dynamic trees, our original goal.

On the basis of these results, Danny and I conjecture that splay trees are a universally optimum form of binary search tree in the following sense: Consider a fixed set of n items and an arbitrary sequence of $m \geq n$ retrievals of these items. Consider any algorithm that performs these retrievals by starting

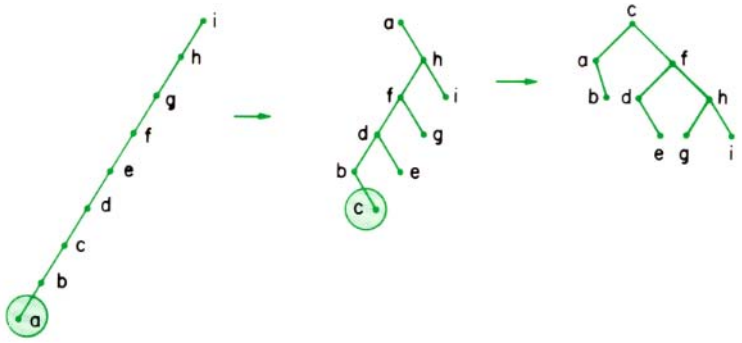


FIGURE 7. A Sequence of Two Costly Splays on an Initially Unbalanced Tree

with an initial binary search tree, performing each retrieval by searching from the root in the standard way, and intermittently restructuring the tree by performing rotations anywhere in it. The cost of the algorithm is the sum of the depths of the retrieved items (when they are retrieved) plus the total number of rotations. We conjecture that the total cost of the splaying algorithm, starting with an arbitrarily bad initial tree, is within a constant factor of the minimum cost of any algorithm. Perhaps this conjecture is too good to be true. One additional piece of evidence supporting the conjecture is that accessing all the items of a binary search tree in sequential order using splaying takes only linear time.

In addition to their intriguing theoretical properties, splay trees have potential value in practice. Splaying is easy to implement, and it makes tree update operations such as insertion and deletion simple as well. Preliminary experiments by Doug Jones suggest that splay trees are competitive with all other known data structures for the purpose of implementing the event list in a discrete simulation system. They may be useful in a variety of other applications, although verifying this will require much systematic and careful experimentation.

The development of splay trees suggests several conclusions. Continued work on an already-solved problem can lead to major simplifications and additional insights. Designing for amortized efficiency can lead to simple, adaptive data structures that are more efficient in practice than their worst-case-efficient cousins. More generally, the efficiency measure chosen suggests the approach to be taken in tackling an algorithmic problem and guides the development of a solution.

I want to make a few comments about what I think the field of algorithm design needs. It is trite to say that we could use a closer coupling between theory and practice, but it is nevertheless true. The world of practice provides a rich and diverse source of problems for researchers to study. Researchers can provide practitioners with not only practical algorithms for specific problems, but broad approaches and general techniques that can be useful in a variety of applications.

Two things would support better interaction between theory and practice. One is much more work on experimental analysis of algorithms. Theoretical analysis of algorithms rests on sound foundations. This is not true of experimental analysis. We need a disciplined, systematic, scientific approach. Experimental analysis is in a way much harder than theoretical analysis because experimental analysis requires the writing of actual programs, and it is hard to avoid introducing bias through the coding process or through the choice of sample data.

Another need is for a programming language or notation that will simultaneously be easy to understand by a human and efficient to execute on a machine. Conventional programming languages force the specification of too much irrelevant detail, whereas newer very-high-level languages pose a challenging implementation task that requires much more work on data structures, algorithmic methods, and their selection.

There is now tremendous ferment within both the theoretical and practical communities over the issue of parallelism. To the theoretician, parallelism offers a new model of computation and thereby changes the ground rules of theoretical analysis. To the practitioner, parallelism offers possibilities for obtaining tremendous speedups in the solution of certain kinds of problems. Parallel hardware will not make theoretical analysis unimportant, however. Indeed, as the size of solvable problems increases, asymptotic analysis becomes more, not less, important. New algorithms will be developed that exploit parallelism, but many of the ideas developed for sequential computation will transfer to the parallel setting as well. It is important to realize that not all problems are amenable to parallel solution. Understanding the impact of parallelism is a central goal of much of the research in algorithm design today.

I do research in algorithm design not only because it offers possibilities for affecting the real world of computing, but because I love the work itself, the rich and surprising connections among problems and solutions it reveals, and the opportunity it provides to share with creative, stimulating, and thoughtful colleagues in the discovery of new ideas. I thank the Association for Computing Machinery and the entire computing community for this award, which recognizes not only my own ideas, but those of the individuals with whom I have had the pleasure of working. They are too many to name here, but I want to acknowledge all of them, and especially John Hopcroft and Danny Sleator, for sharing their ideas and efforts in this process of discovery.

CR Categories and Subject Descriptors: A.0 **[General Literature]**: General—*biographies/autobiographies*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems; G.2.2 **[Discrete Mathematics]**: Graph Theory; K.2 **[Computing Milieux]**: History of Computing—*people*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: John E. Hopcroft, Robert E. Tarjan, Turing Award

Author's Present Addresses: Robert E. Tarjan, Computer Science Dept., Princeton University, Princeton, NJ 08544; and AT&T Bell Laboratories, Murray Hill, NJ 07974.