DESIGNING ALGORITHMS

Robert E. Tarjan

CS-TR-069-86

December 1986

Designing Algorithms

Robert E. Tarjan

Department of Computer Science
Princeton University
Princeton, NJ 08544

and

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

The quest for efficiency in computational methods yields not only fast algorithms, but also insights into the problems being solved. Such insights can produce problem-solving methods that combine speed with elegance, simplicity, and generality. This theme is illustrated with two examples: an algorithm for testing planarity of graphs, and a self-adjusting form of binary search tree.

December, 1986

Designing Algorithms

Robert E. Tarjan

Department of Computer Science

Princeton University

Princeton, NJ 08544

and

AT&T Bell Laboratories

Murray Hill, NJ 07974

I was surprised and delighted to learn of my selection as the co-recipient of the 1986 Turing Award. My delight turned to discomfort, however, as I realized that with this great honor comes a great responsibility: to speak to the computing community about some topic of one's own choosing. Many of my friends, when they learned of the award, suggested that I preach a sermon of some sort. But as I am not the sermonizing kind, I decided upon reflection just to share with you some thoughts about the work I do and what its relevance might be to the real world of computing.

Most of my research concerns the design and analysis of efficient computer algorithms. The goal in this field of study 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 based on 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 something deeper is true. The approach of designing for theoretical efficiency forces one to concentrate on the important aspects of the problem, to avoid redundant computations, and to design data structures that represent exactly the information needed to solve the problem. If this approach is successful, the result is not only an efficient algorithm, but also 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 the real-world problems.

I shall illustrate algorithm design with two examples. One is a graph algorithm, for testing the planarity of a graph, developed by John Hopcroft and me. The other is a data structure, a self-adjusting form of search tree devised by Danny Sleator and me. Let me tell you how we obtained these results.

I graduated from CalTech in June of 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 the latter and enrolled as a graduate student at Stanford in the fall of 1969. I thought that as a computer scientist I could use my mathematical skills to solve problems of more immediate prac-

tical 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 advisor at Stanford was Don Knuth, and I think he had other plans for my future than the study of artificial intelligence: his first advice to me was to read Volume 1 of his book, *The Art of Computer Programming* [27].

By June of 1970, I had successfully passed my Ph.D. qualifying examinations, and I began to cast around for a thesis topic. In 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.[1] Graphs serve as an important combinatorial model for problems in a variety of fields [10].

As a measure of computational efficiency, we settled on using the worst-case running time of an algorithm, as a function of the input size, on a sequential random-access machine (an abstraction of a sequential general-purpose digital computer). We chose to ignore constant factors in running time. This measure is independent of both the machine model and the details of the algorithm implementation, because constant factors are ignored. It is realistic: an algorithm efficient by this measure tends to be efficient in practice. It is also analytically tractable: we could actually derive interesting results about it.

Other approaches we considered have weaknesses of various kinds. Edmonds in the mid-1960's stressed the distinction between polynomial-time and non-polynomial-time algorithms [12], and though this distinction led in the early 1970's to the theory of NP-completeness [11,25], which now plays a central role in

[1] A *graph* is a collection of *vertices* and a set of unordered pairs of vertices called *edges*.

complexity theory [16,26], it was too weak to provide much guidance in the choice of 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 [27,28], but such detailed analysis was very hard to do for the sophisticated algorithms we wanted to study and sacrificed 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.

The state of algorithm design in the late 1960's 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 [8], 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 beautiful theorem of Kuratowski [29] states that a graph is planar if and only if it does not contain as a subgraph a subdivision[1] of one of the two graphs in Figure 2, the complete graph on five vertices (K_5), and the complete bipartite graph on two sets of three vertices ($K_{3,3}$).

[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 necessary to specify the geometry of an embedding; a specification of the topology of the embedding will do. For example, it is enough to know the clockwise ordering around each vertex of its incident edges.[2]

Auslander and Parter [4] in 1961 formulated a planarity algorithm that I shall call the *path*[3] *addition method* (after Even [13]). The algorithm easy to state recursively. Find a simple cycle[4] in the graph. Removal of this cycle breaks the rest of the graph into pieces, called *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

---

[1] A graph G_1 is a *subgraph* of another graph G_2 if the vertex set of G_1 is contained in that of G_2, and the edge set of G_1 is contained in that of G_2. A *subdivision* of a graph G is formed by repeating the following step an arbitrary number of times: Add a new vertex u to G, and replace some edge $\{v, w\}$ by the pair of edges $\{v, u\}$ and $\{u, w\}$.

[2] An edge $\{v, w\}$ is said to be *incident* to v and to w.

[3] A *path* in a graph is a sequence of edges $\{v_1, v_2\}$, $\{v_2, v_3\}$,..., $\{v_{k-1}, v_k\}$. Vertices v_1 and v_k are the *end vertices* of the path.

[4] A *simple cycle* in a graph is a sequence of edges $\{v_1, v_2\}$, $\{v_2, v_3\}$,..., $\{v_{k-1}, v_k\}$, $\{v_k, v_1\}$ such that $v_i \neq v_j$ for $i \neq j$.

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; if not, it is nonplanar.

[Figure 3]

As noted by Goldstein [17] in 1963, one must 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. Shirey [33] in 1969 proposed a version of the algorithm running in $O(n^3)$ time [1] 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. In this we eventually succeeded.

The first step was to settle on an appropriate graph representation, one that takes advantage of sparsity. We used a very simple one, 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, called *biconnected components*.

[1] The notation "$f(n)$ is $O(g(n))$," used of two functions f and g of n, means that for some positive constant c and all sufficiently large n, $f(n) \leq cg(n)$.

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 [22,38]. 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 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 a path at a time to a growing planar embedding. Each path is disjoint from earlier paths except for its end vertices. When a path is embedded, there may be several ways to embed it. 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 such a search [22]. Our initial method for keeping track of alternative embeddings used a complicated and ad hoc nested structure whose correct working requires that paths be generated in a specific order determined dynamically. Fortunately, our path generation algorithm was flexible enough to meet this requirement, and we obtained a planarity algorithm running in $O(n \log n)$ time [20].

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 paths are all generated first, then a graph representing their pairwise embedding constraints is constructed, and finally the graph is colored 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 explicitly computing only $O(n)$ such constraints, but enough so that if these are satisfied so are all the rest. Working out this idea lead to an $O(n)$-time planarity algorithm, which become the subject of my Ph.D. dissertation [37]. 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 twelve seconds. This was about eighty times faster than any other claimed time bound I could find in the literature. The program runs about 500 lines long, not including comments.

This is not the end of the story, however. In preparing a description of the algorithm for journal publication, we discovered how to avoid the task of constructing a graph representing pairwise embedding constraints and trying to color it. 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 a new path. This led to a simpler algorithm, still with an $O(n)$ time bound [23]. Don Woods, a student of mine, programmed the simpler algorithm, which tested graphs with 7000 vertices in eight 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 for actually constructing 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

[14,15,21,22,31,38,39,41,46]; the pile of twin stacks has been used to solve a sorting problem and a problem of recognizing codes for planar self-intersecting curves [32].

Our algorithm is not the only way to test planarity in $O(n)$ time. Another approach, due to Lempel, Even, and Cederbaum [30], 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 me [14] and by Booth and Lueker [9] 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 for keeping track of possible embeddings, the PQ-tree of Booth and Lueker. This data structure itself has several other applications [9].

There is even a third $O(n)$-time planarity algorithm. Only recently I discovered that a novel form of search tree called a *finger search tree*, invented in 1977 by Guibas, Plass, McCreight, and Roberts [18], can be used in the original planarity algorithm that John and I developed to improve the running time from $O(n \log n)$ to $O(n)$ [19]. 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 the realms of sorting [19] and computational geometry [47] in addition to graph algorithms.

What conclusions can we draw from these developments? Choosing the correct data structures is crucial to the design of efficient algorithms. Sometimes these structures are complicated. It can take a long time — years — to distill a complicated data structure or algorithm down to its essentials. But a good idea,

even if complicated, has a way of eventually becoming simpler, and of providing solutions to problems different from those for which it was intended. Just as in mathematics, there are rich and deep connections among diverse parts of computer science.

I want to shift the emphasis now from graph algorithms to data structures. Let us reflect a little on whether worst-case running time is an appropriate measure of the efficiency of a data structure. 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, not the time taken by a single operation. (Exceptions occur in real-time applications in which 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 these 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 situation is the *amortized time*, defined to be the time of an operation averaged over a worst-case sequence of operations. (See my survey paper

[44] for a thorough discussion of this concept.) 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 [40,43]. In this case the worse-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 [1].)

These two examples illustrate the use of amortization to give 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, one is led to a different approach than if one were trying to reduce worse-case running time. Instead of carefully crafting a structure with exactly the right properties to guarantee a good worst-case time bound, one 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 that 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 [36], 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*: 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.)

[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 to represent static or dynamically changing sorted sets. Although in many situations a hash table [28] 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 information among items is important. For example, in a search tree one can in a single search 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. For further discussion of the properties of search trees, see any good

book on data structures [3,28,42].

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* of Adelson-Velskii and Landis [2] 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 performing 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 nodes, and preserves the symmetric order of the items. (See Figure 5.)

[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 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 [5], a tree with in general many more than two children per node, suitable for storing very large sets of data on secondary storage media such as disc memories. 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 1970's. 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 to retrieve an item depends on its weight, heavier items being easier to access than lighter ones. The hardest requirement to meet was the need to perform drastic update operations fast; each tree was to represent a list, and we needed the ability to do 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* [7], which become the subject of Sam's Ph.D. thesis [6]. Danny and I combined biased search trees with other ideas to obtain the efficient maximum flow algorithm we had been seeking. This algorithm was the subject of Danny's Ph.D. thesis [34]. The data structure that forms the heart of this algorithm, called a *dynamic tree*, has a variety of other applications [35].

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 had not been 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 simpli-

fying 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, which would be as efficient as balanced trees but only in the amortized sense. Having formulated this problem, it was only a few weeks before we had a candidate data structure, although it took us much longer to analyze it. (And, as I shall discuss below, 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*[1] operation. 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 only a little harder to access, if that.

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

[Figure 6]

Splay Step: Apply the appropriate one of the following three cases:

> *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-

[1] Splay, as a verb, means "to spread out".

versa, rotate at x and then again at x.

[Figure 7]

As suggested by the example in Figure 7, a sequence of costly retrievals in an originally very unbalanced splay tree will quickly drive it into a reasonably balanced state. Indeed, the amortized retrieval time in an n-node splay tree is $O(\log n)$ [36]. 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, one expressly constructed to minimize the total retrieval time for the given sequence [36]. 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 [36], our original goal.

On the basis of these results, Danny and I have made the 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 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. I invite you to think about it. 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 [45].

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 [36]. Preliminary experiments by Doug Jones [24] 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 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 not only practical algorithms for specific problems, but broad approaches and general techniques useful in a variety of problems.

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; we understand how to do it. This is not true of experimental analysis. We need a disciplined, systematic, scientific approach. Experimental analysis is much harder than theoretical

analysis in a way because it requires writing programs. Comparing algorithms experimentally without introducing bias through the coding process or through the choice of sample data is not easy.

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, one 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 the possibility of obtaining tremendous speedups in the solution of certain kinds of problems. But it is important to realize that parallelism is no panacea. Parallel hardware will not make theoretical analysis unimportant. Indeed, as the size of solvable problems increases, asymptotic analysis becomes more, not less important. New algorithms are being developed and will be developed that exploit parallelism, but many of the ideas developed for sequential computation will transfer to the parallel setting, and not all problems are amenable to parallel solution. Understanding the impact of parallelism is a central goal in much research in algorithm design today.

I do research in algorithm design not only because it offers the possibility of 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.

References

[1] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," *Math. Ann.*, Vol. 99, No. 1 & 2 (1928), pp. 118-133.

[2] G. M. Adelson-Velskii and Y. M. Landis, "An algorithm for the organization of information," *Soviet Math. Dokl.*, Vol. 3, No. 5 (September, 1962), pp. 1259-1261.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA, 1974.

[4] L. Auslander and S. V. Parter, "On imbedding graphs in the plane," *J. Math. and Mech.*, Vol. 10, No. 3 (May, 1961), pp. 517-523.

[5] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica,* Vol. 1, No. 3 (1972), pp. 173-189.

[6] S. W. Bent, "Dynamic Weighted data structures," Ph.D. thesis, Department of Computer Science, Stanford University, May, 1982.

[7] S. W. Bent, D. D. Sleator, and R. E. Tarjan, "Biased Search Trees," *SIAM J. Comput.*, Vol. 14, No. 3 (August, 1985), pp. 545-568.

[8] J. L. Bentley, *Writing Efficient Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[9] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *J. Comp. Sys. Sci.*, Vol. 13, No. 3 (December, 1976), 335-379.

[10] R. Busacker and T. Saaty, *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill, New York, NY, 1965.

[11] S. A. Cook, "The complexity of theorem proving procedures," *Proc. Third Annual ACM Symp. on Theory of Computing*, Shaker Heights, Ohio, May 3-5, 1971, Association for Computing Machinery, New York, NY, 1971, pp. 151-158.

[12] J. Edmonds, "Paths, trees, and flowers," *Canadian J. Math.*, Vol. 17 (1965), pp. 449-467.

[13] S. Even, *Graph Algorithms* Computer Science Press, Potomac, MD, 1979.

[14] S. Even and R. E. Tarjan, "Computing an *st*-numbering," *Theoretical Computer Science*, Vol. 2, No. 3 (September, 1976), pp. 339-344.

[15] M. R. Garey, "A linear-time algorithm for finding all feedback vertices," *Inform. Process. Lett.*, Vol. 7, No. 6 (October, 1978), pp. 274-276.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco, CA, 1979.

[17] A. J. Goldstein, "An efficient and constructive algorithm for testing whether a graph can be embedded in a plane," Graph and Combinatorics Conference, Office of Naval Research Logistics Project, Department of Mathematics, Princeton University, Princeton, NJ, May 16-18, 1963, 2pp.

[18] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, "A new representation for linear lists," *Proc. Ninth Annual ACM Symposium on Theory of Computing*, Boulder, CO, May 2-4, Association for Computing

Machinery, New York, NY, 1977, pp. 49-60.

[19] K. Hoffman, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan, "Sorting Jordan sequences in linear time using level-linked search trees, *Information and Control*, Vol. 68, Nos. 1-3 (January/February/March, 1986), pp. 170-184.

[20] J. E. Hopcroft and R. E. Tarjan, "Planarity testing in V log V steps: Extended abstract," *IFIP Congress 71: Foundations of Information Processing, TA-2*, Ljubljana, Yugoslavia, August 23-28, 1971, North-Holland, Amsterdam, The Netherlands, pp. 18-22.

[21] J. Hopcroft and R. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Comput.*, Vol. 2, No. 3 (September, 1973), pp. 135-158.

[22] J. E. Hopcroft and R. E. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Comm. ACM,* Vol. 16, No. 6 (June, 1973), pp. 372-378.

[23] J. Hopcroft and R. Tarjan, "Efficient planarity testing," *J. Assoc. Comput. Mach.*, Vol. 21, No. 4 (October, 1974), pp. 549-568.

[24] D. W. Jones, "An empirical comparison of priority-queue and event-set implementations," *Comm. ACM*, Vol. 29, No. 4 (April, 1986), pp. 300-311.

[25] R. M. Karp, "Reducibility among combinatorial problems," *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, NY, 1972, pp. 85-104.

[26] R. M. Karp, "Combinatorics, complexity, and randomness," *Comm. ACM,* Vol. 29, No. 2 (February, 1986), pp. 98-109.

[27] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.

[28] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[29] C. Kuratowski, "Sur le probleme des corbes gauches en topologie," *Fundamenta Mathematicae,* Vol. 15 (1930), pp. 271-283.

[30] A. Lempel, S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," *Theory of Graphs: International Symposium*, Rome, Italy, July, 1966, P. Rosenstiehl, ed., Gordon and Breach, New York, NY, 1967, pp. 215-232.

[31] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flow graph," *Trans. Prog. Lang. and Sys.*, Vol. 1, No. 1 (July, 1979), pp. 121-141.

[32] P. Rosenstiehl and R. E. Tarjan, "Gauss codes, planar Hamiltonian graphs, and stack-sortable permutations," *J. Algorithms*, Vol. 5, No. 3 (September, 1984), pp. 375-390.

[33] R. W. Shirey, "Implementation and analysis of efficient graph planarity testing algorithms," Ph.D. thesis, University of Wisconsin, June, 1969.

[34] D. D. Sleator, "An $O(nm \log n)$ algorithm for maximum network flow," Technical Report No. STAN-CS-80-831, Department of Computer Science, Stanford University, December, 1980.

[35] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Comput. Sys. Sci.*, Vol. 26, No. 3 (June, 1983), pp. 362-391.

[36] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. Assoc.*

Comput. Mach., Vol. 32, No. 3 (July, 1985), pp. 652-686.

[37] R. E. Tarjan, "An efficient planarity algorithm," Technical Report STAN-CS-244-71, Department of Computer Science, Stanford University, December, 1971.

[38] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, Vol. 1, No. 2 (June 1972), pp. 146-160.

[39] R. E. Tarjan, "Testing flow graph reducibility," *J. Computer Sys. Sci.*, Vol. 9, No. 3 (December, 1974), pp. 52-53.

[40] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. Assoc. Comput. Mach.*, Vol. 22, No. 2 (April, 1975), pp. 215-225.

[41] R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica*, Vol. 6, No. 2 (1976), pp. 171-185.

[42] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[43] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. Assoc. Comput. Mach.*, Vol. 31, No. 2 (April, 1984), pp. 245-281.

[44] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Math.*, Vol. 6, No. 2 (April, 1985), pp. 306-318.

[45] R. E. Tarjan, "Sequential access in splay trees takes linear time," *Combinatorica*, Vol. 5, No. 4 (1985), pp. 367-378.

[46] R. E. Tarjan, "Two streamlined depth-first search algorithms," *Fundamenta Informaticae*, Vol. IX (1986), pp. 85-94.

[47] R. E. Tarjan and C. J. van Wyk, "An $O(n \log \log n)$-time algorithm for triangulating simple polygons," *SIAM J. Comput.*, to appear.

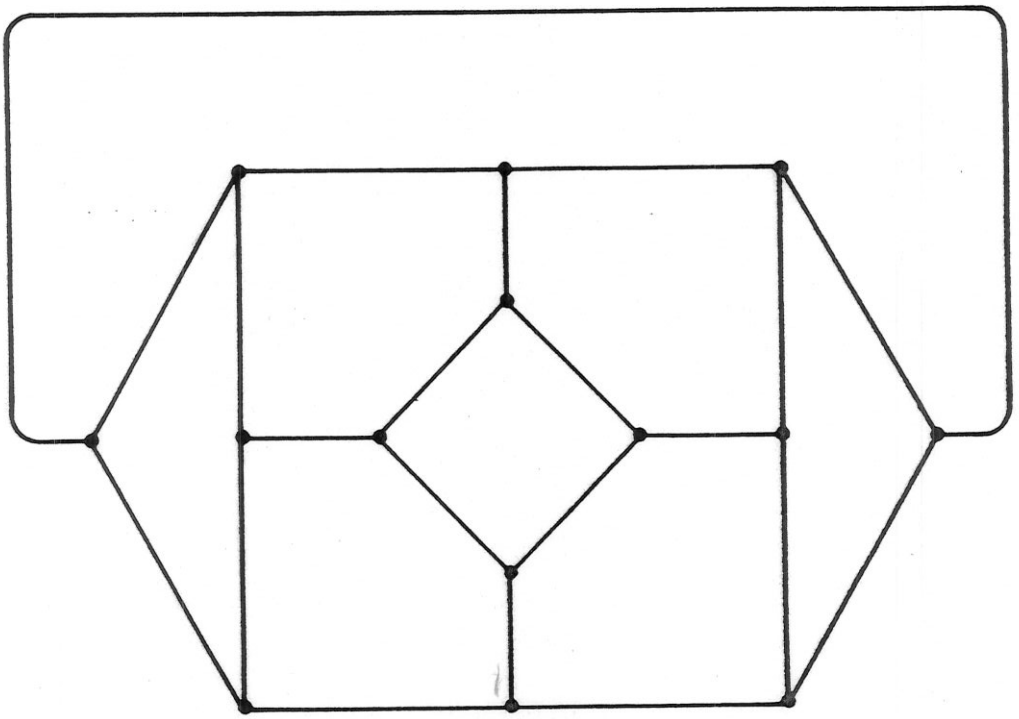
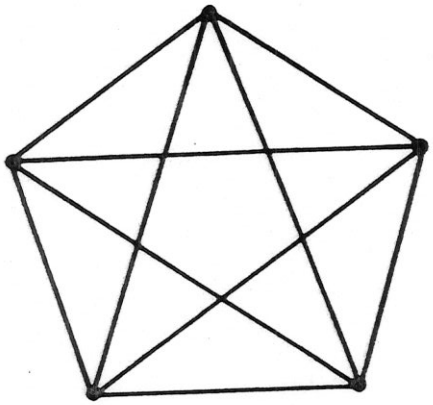


Figure 1. A planar graph.

K_5



$K_{3,3}$

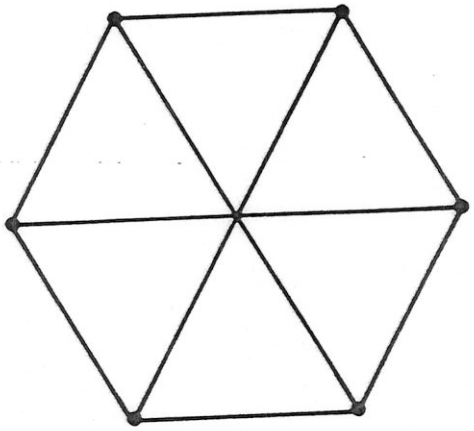


Figure 2. The forbidden subgraphs of Kuratowski.

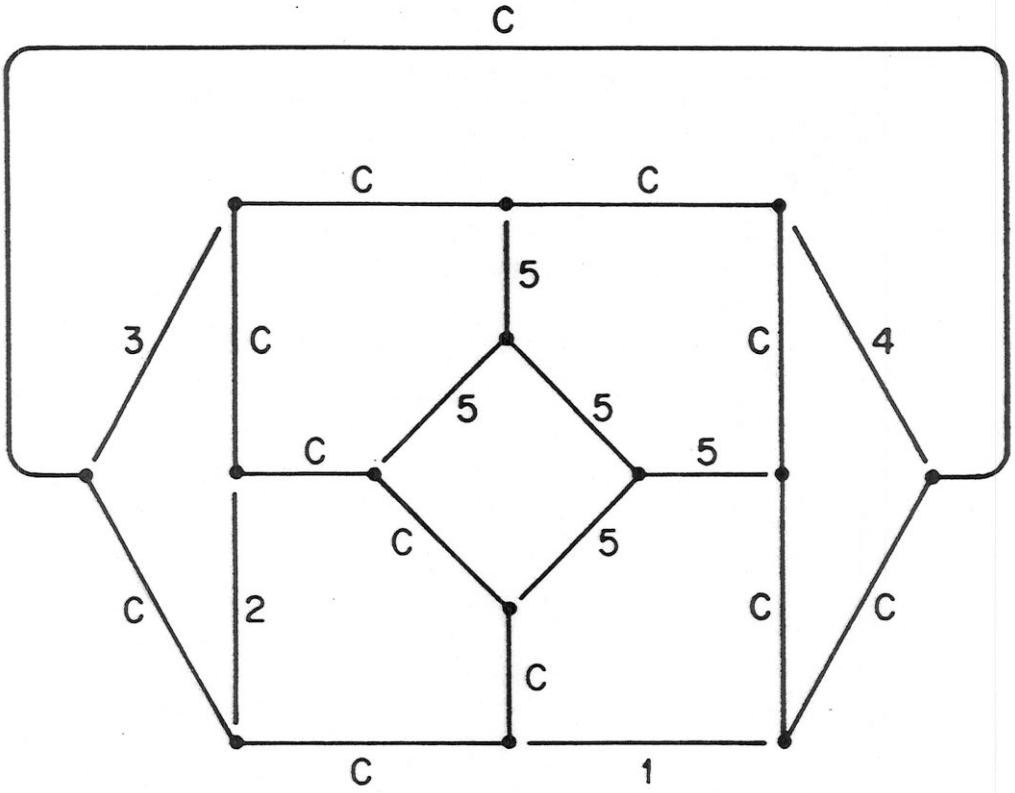


Figure 3. Division of the graph of Figure 1 into five segments by removing cycle C. Segments 1 and 2 must be embedded on opposite sides of C, as must 1 and 3, 1 and 4, 2 and 5, 3 and 5, and 4 and 5.

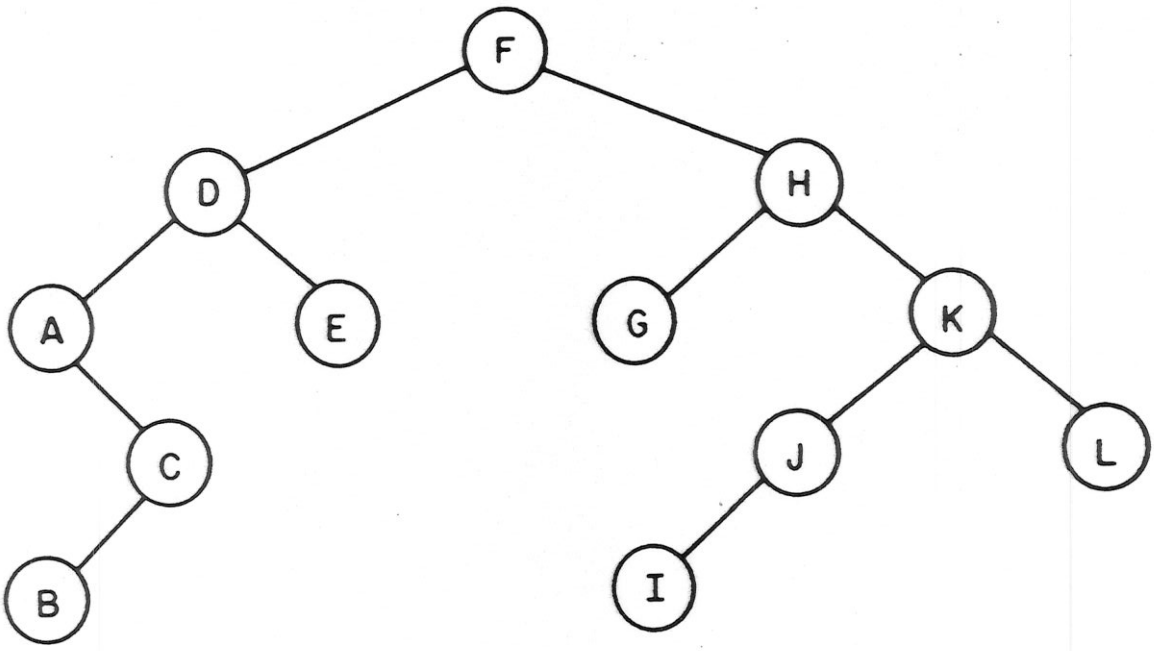


Figure 4. A binary search tree. The items are letters, ordered alphabetically.

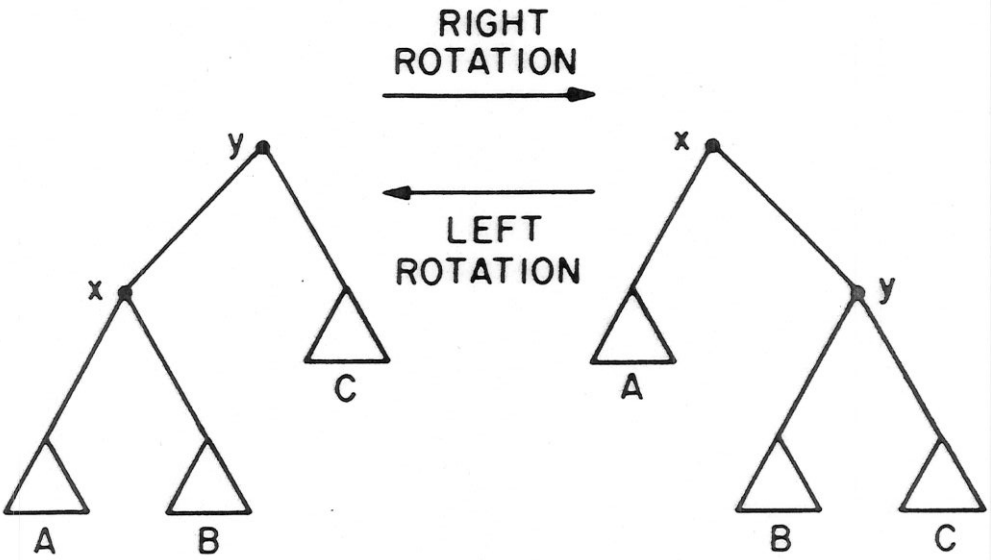


Figure 5. A rotation in a binary search tree. The right rotation is at node x, the inverse left rotation is at node y. The triangles denote arbitrary subtrees, possibly empty. The tree shown can be part of a larger tree.

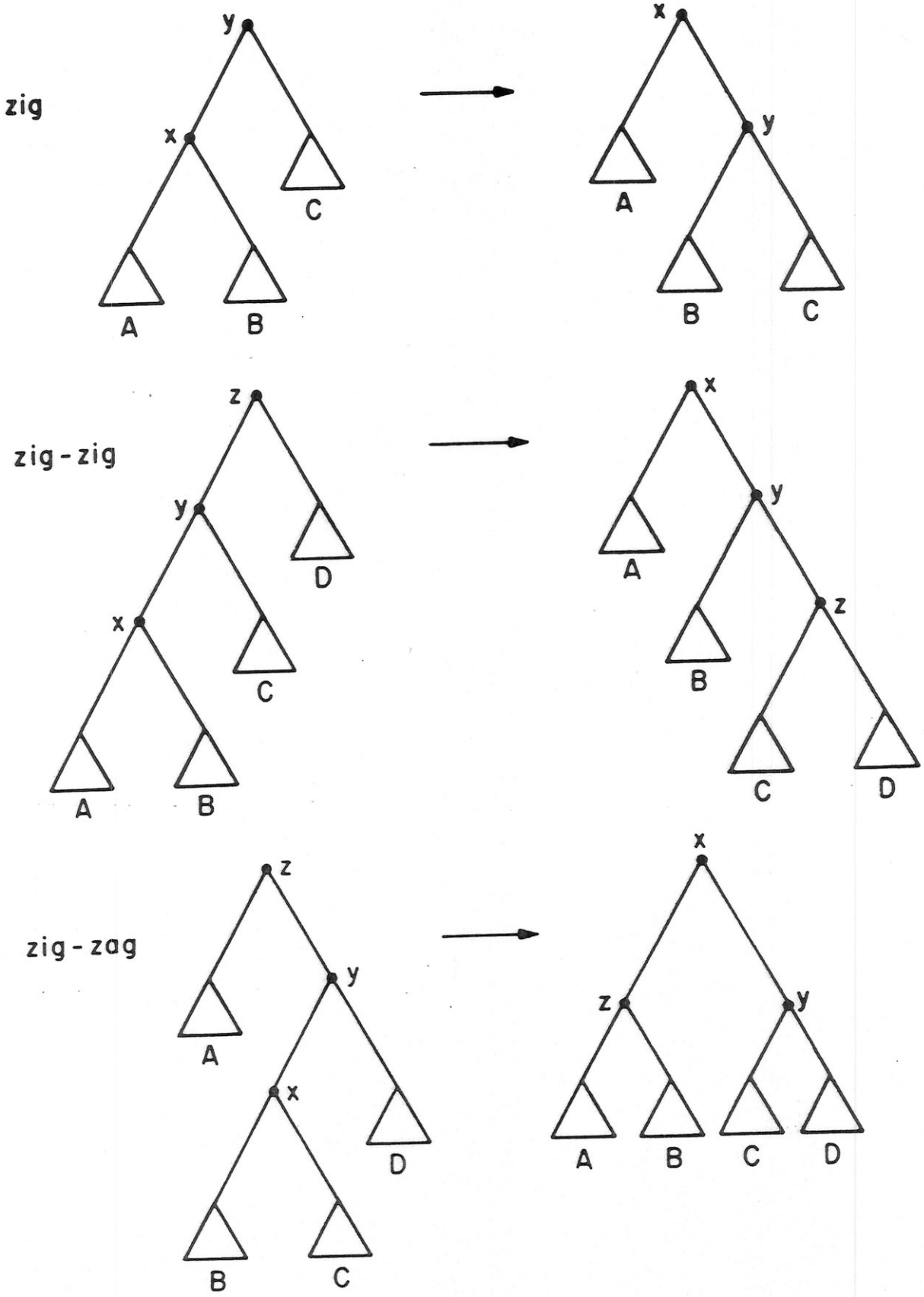


Figure 6. The cases of a splay step. In the zig-zig and zig-zag cases, the tree shown can be part of a larger tree.

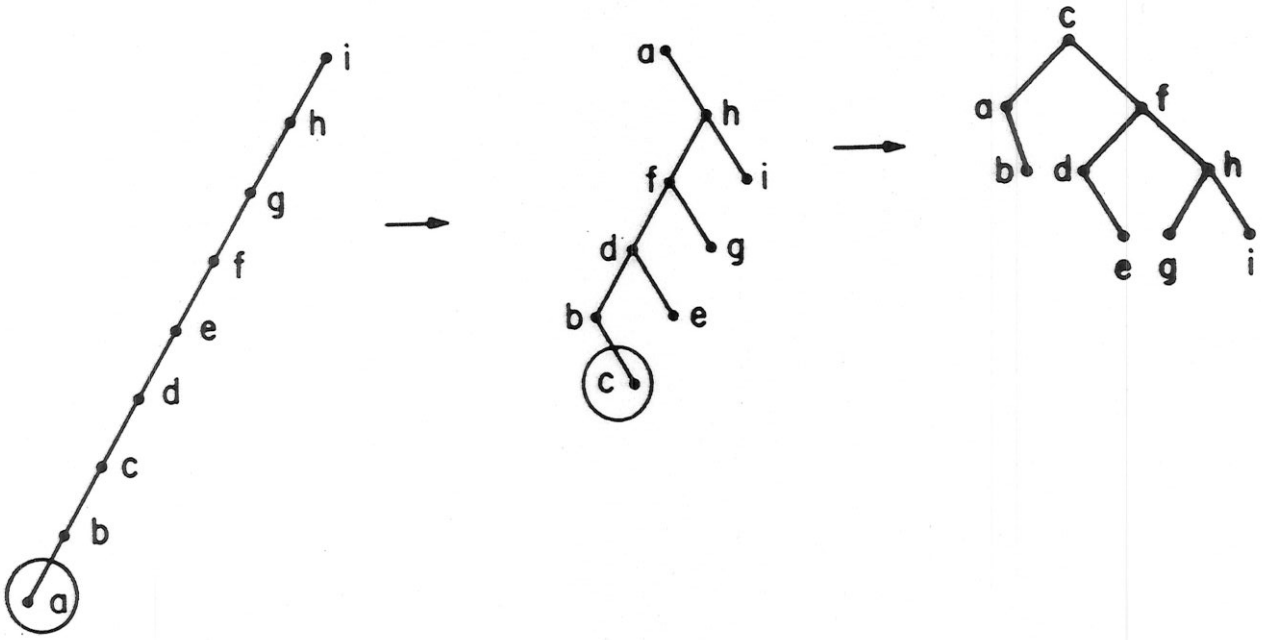


Figure 7. A sequence of two costly splays on an initially unbalanced tree.