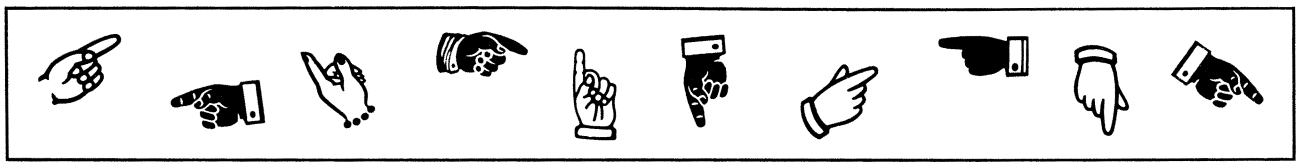
# In Search of Speedier Searches



Tables or lists that rearrange themselves in just the right way within a running computer program can save a lot of time

By IVARS PETERSON

Microseconds matter. Every extra step that a computer takes to find a bit of data or to do a simple operation takes time. Repeated thousands of times, the extra steps add up to long delays in computer processing that may stretch into minutes or hours.

With this in mind, many computer scientists and applied mathematicians have thought long and hard about ways to make a computer's search as simple and efficient as possible. This has led to the invention of a variety of "data structures." In general, a data structure — whether in the form of a list, table, array or tree — specifies the items involved (perhaps a set of names, telephone numbers or key words) and what can be done to those items. A list of names and telephone numbers constructed so that specific telephone numbers can be found, new numbers added and old numbers deleted is one example of a simple data structure.

In the case of a telephone directory, the most naive way of programming a computer to find a specific number is to ask it to start at the top of the list and keep on going, checking each entry one by one, until it finds the requested item. If frequently called numbers appear near the end of a list, the computer could spend an unnecessarily long time searching for the numbers. The answer may be to use a dynamic data structure that changes as it is used.

"People tend to program things in a very naive way," says Robert E. Tarjan of AT&T Bell Laboratories in Murray Hill, N.J. Yet, "there are reasonably simple data structures they could be using that conceivably would improve things tremendously."

One method that could speed things up is to allow items within a list to reorganize themselves according to how often they are called upon. For example, each accessed item can be automatically moved by swapping places with its predecessors until it gets to the front of the list. Because the most frequently accessed items will end up close to the front, the computer will find them more quickly. Although the "move-to-front" operation itself takes some time, the hope is that over the long run, there will be a net saving.

Alternatively, a single exchange, in which the accessed item is swapped only with its immediate predecessor, may be good enough to decrease computer search times. Another approach is to know ahead of time or keep track of how often items come up. In this "frequency-count" case, the items are listed according to how often they are likely to be accessed, and the items don't change their order later.

Working with Daniel D. Sleator, Tarjan has been studying the mathematical properties of these schemes, especially the "move-to-front" idea, to see how efficient they really are. "We'd like to prove that this thing is efficient in some general-purpose way," says Tarjan. "The idea is that on any sequence [of operations] whatsoever, this data structure performs as well as any other data structure within the same class.

"The proofs are hard to come by," says Tarjan. The intriguing property of these data structures is that they are extremely simple to program, yet they're very difficult to analyze because the structure is constantly changing over time.

"It's hard to analyze in the same way that strange attractors [SN: 4/26/84, p. 328] are hard to analyze," he says. "Here, we have an algorithm on a discrete structure, which we are iterating, and in some sense, we want to know the steady-state behavior of this thing. You can run computer simulations, and you seem to get the answer. The mathematical question is, how do you prove it?"

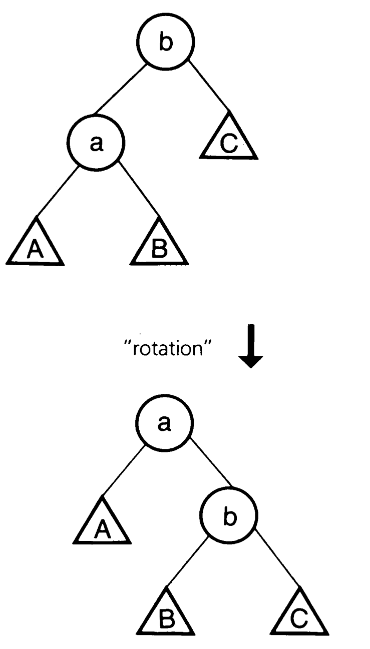
So far, both theoretical results and experiments performed on real data indicate that the "move-to-front" scheme works efficiently in a wide variety of situations. For instance, John L. Bentley, now with Bell Labs after working at Carnegie-Mellon University in Pittsburgh, tried the methods on English-language text and the names of variables in computer programs written in languages like Pascal. He and his students found that the "move-to-front" technique worked as well as and sometimes better than the "frequency-count" method. The "single-exchange" method adjusted too slowly to improve search times significantly.

Tarjan cites two advantages in using "self-adjusting" data structures. "They're very simple, which makes them easy to program," he says. Although other, more efficient algorithms for searches have been invented, most turn out to be "so hard to program that people are not willing to take the time to do it."

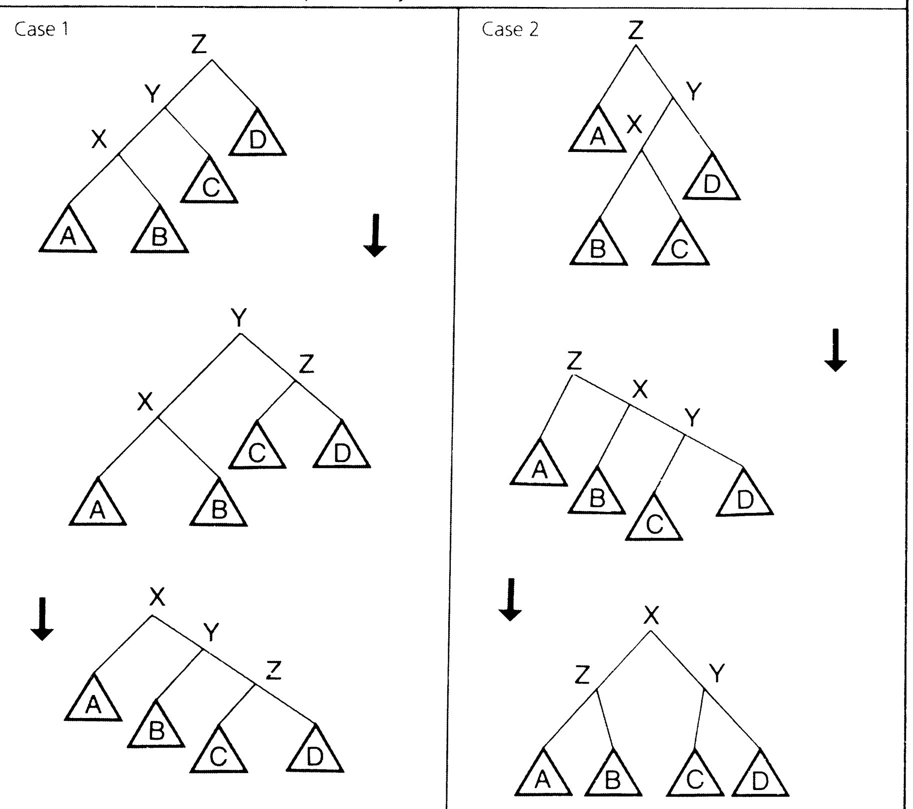
The other advantage from a purely practical point of view is that these structures

## Self-Adjusting Binary Search Trees

A single "rotation" rearranges a portion of a binary tree:



A "splay" step, involving several rotations, brings node X to the root of a simple binary tree:



adjust to fit input data or changing patterns of requests. With a fixed data structure, "if the data don't fit your model of what the data are supposed to look like, you can lose," says Tarjan. In the past, programmers have tended to design structures that may be efficient on the average or fit a certain situation or a particular pattern of input data. "With these self-adjusting structures, you're guaranteed to get pretty good behavior," he says.

Lately, Tarjan and his colleagues have been applying the same idea to binary search trees. In this case, items are organized so that they are linked in a definite pattern. The starting point (or root) branches into two nodes. These nodes, in turn, each branch to form two more nodes, and so on. The question is whether there is an equivalent of the "move-to-front" method for these binary search trees. In other words, if the computer often needs information stored at particular nodes, is there a way to move those items close to the root and so rearrange the tree to make searches more efficient?

Tarjan's answer is to use a move called a "rotation" that shifts the connections between the nodes in a specific way (as illustrated). A combination of rotations to give a sequence of steps he calls a "splay" moves the accessed item to the root of the tree, and as a bonus, also approximately halves the distance from the root of the tree (where the computer would start its search) to any particular node.

"Whenever the tree is accessed, certain adjustments take place along the access path," says Tarjan. Although making the adjustments takes time, the overall procedure is very fast in many situations for a long sequence of requests or accesses. This is especially true when the pattern of requests is nonuniform or somewhat unpredictable.

Self-adjusting search trees turn out to be very nice for allocating space within a computer's memory, says Tarjan. "We've done some experiments on that at Bell Labs, and it seems to work quite well in practice."

When working with large, shared computers, users are continually asking for blocks of storage space of a certain size. As the number of users and their needs change, the computer's operating system must have a way of allocating new requests for space. In the simplest approach, called "first fit," the computer scans, from front to back, a list of all the available memory blocks until it finds a "hole" or free space large enough to accommodate the request. The problem is that the front of the memory gets chopped up, usually leaving only tiny chunks free. The computer has to scan over an increasing number of these small holes until it finds one large enough to use. "The memory allocation routine itself starts to slow down eventually," says Tarjan.

If you use a self-adjusting binary search tree to represent the holes, says Tarjan, "then you can really speed up the allocation process." Each node corresponds to a block of memory. "The clever trick here is that you can store the tree information in the free blocks themselves," he says. The tree structure sits in the unused space, and as the unused space fills up, the tree gets smaller.

Now, Tarjan is thinking about looking at more complicated search trees and at arrays that contain several bits of data in each location. "The most general issue is to explore other situations in which this idea of self-adjustment would be applicable," he says. "It's a very general notion, and I think it should have very general applications."

The ideas are also simple and can easily be applied to programs for microcomputers. "Some of these very simple ideas may have potentially huge practical payoffs, especially in a limited resource situation like a personal computer," says Tarjan. "They deserve to be tried out." □