1. Introduction

I would like to talk about various problems I have worked on over the course of my career. In this lecture I'll review simple problems with interesting applications, and problems that have rich, sometimes surprising, structure.

Let me start by saying a few words about how I view the process of research, discovery and development. (See Figure 1.)

My view is based on my experience with data structures and algorithms in computer science, but I think it applies more generally. There is an interesting interplay between theory and practice. The way I like to work is to start out with some application from the real world. The real world, of course, is very messy and the application gets modeled or abstracted away into some problem or some setting that someone with a theoretical background can actually deal with. Given the abstraction, I then try to develop a solution which is usually, in the case of computer science, an algorithm, a computational method to perform some task. We may be able to prove things about the algorithm, its running time, its efficiency, and so on. And then, if it's at all useful, we want to apply the algorithm back to the application and see if it actually solves the real problem. There is an interplay in the experimental domain between the algorithm developed, based on the abstraction, and the application; perhaps we discover that the abstraction does not capture the right parts of the problem; we have solved an interesting mathematical problem but it doesn't solve the real-world application. Then we need to go back and change the abstraction and solve the new abstract problem and then try to apply that in practice. In this entire process we are developing a body of new theory and practice which can then be used in other settings.

A very interesting and important aspect of computation is that often the key to performing computations efficiently is to understand the problem, to represent the

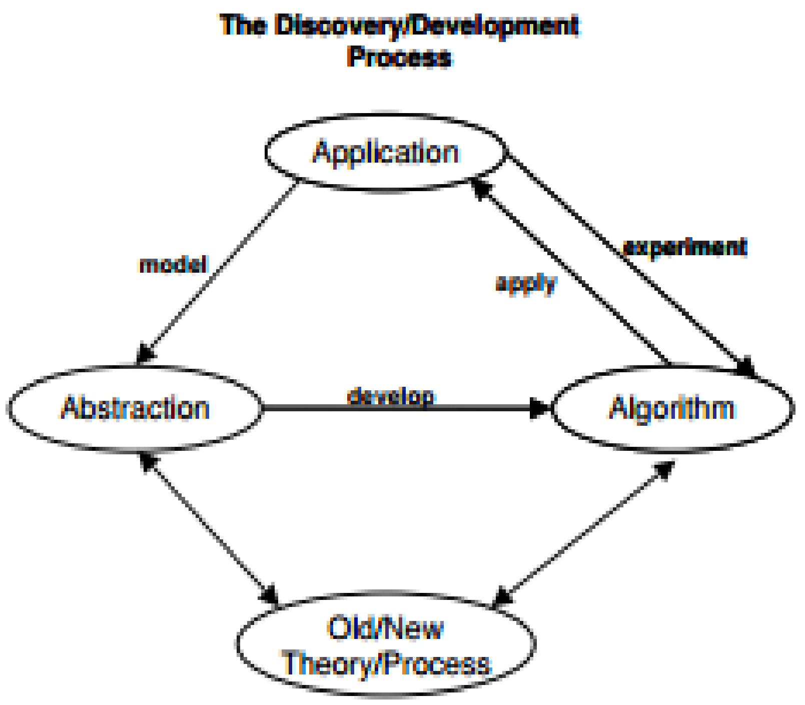

Figure 1.

problem data appropriately, and to look at the operations that need to be performed on the data. In this way many algorithmic problems turn into data manipulation problems, and the key issue is to develop the right kind of data structure to solve the problem. I would like to talk about several such problems. The real question is to devise a data structure, or to analyze a data structure which is a concrete representation of some kind of algorithmic process.

2. Optimum Stack Generation Problem

Let's take a look at the following simple problem. I've chosen this problem because it's an abstraction which is, on the one hand, very easy to state, but on the other hand, captures a number of ideas. We are given a finite alphabet Σ, and a stack S. We would like to generate strings of letters over the alphabet using the stack. There are three stack operations we can perform.

push (A)—push the letter **A** from the alphabet onto the stack,
emit—output the top letter from the stack,
pop—pop the top letter from the stack.

We can perform any sequence of these operations subject to the following well-formedness constraints: we begin with an empty stack, we perform an arbitrary series of ***push***, ***emit*** and ***pop*** operations, we never perform ***pop*** from an empty stack, and we

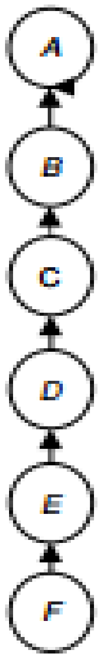

Figure 3.

the pointers to the root node, which names the set. The time of the ***find*** operation is proportional to the length of the path. The tree structure is important here because it affects the length of the ***find*** path. To perform a ***unite(x,y)*** operation, we access the two corresponding tree roots x and y, and make one of the roots point to the other root. The ***unite*** operation takes constant time.

The question is, how long can ***find*** paths be? Well, if this is all there is to it, we can get bad examples. In particular, we can construct the example in Figure 3: a tree which is just a long path. If we do lots of ***finds***, each of linear cost, then the total cost is proportional to the number of ***finds*** times the number of elements, $O(m \cdot n)$, which is not a happy situation.

As we know, there are a couple of heuristics we can add to this method to substantially improve the running time. We use the fact that the structure of each tree is completely arbitrary. The best structure for the ***find***s would be if each tree has all its nodes just one step away from the root. Then ***find*** operations would all be at constant cost. But as we do the ***unite*** operations, depths of nodes grow. If we perform the ***unites*** intelligently, however, we can ensure that depths do not become too big. I shall give two methods for doing this.

Unite* by size** (Galler and Fischer [16]): This method combines two trees into one by making the root of the smaller tree point to the root of the larger tree (breaking a tie arbitrarily). The method is described in the pseudo-code below. We maintain with each root x the tree size, ***size(x) (the number of nodes in the tree).

The situation becomes more interesting if we want to allow ***insertion*** and ***deletion*** operations, since the shape of the tree will change. There are standard methods for inserting and deleting items in a binary search tree. Let me remind you how these work. The easiest method for an ***insert*** operation is just to follow the search path, which will run off the bottom of the tree, and put the new item in a new node attached where the search exits the tree. A ***delete*** operation is slightly more complicated. Consider the tree in Figure 5. If I want to delete say "pig" (a leaf in the tree in Figure 5), I simply delete the node containing it. But if I want to delete "frog", which is at the root, I have to replace that node with another node. I can get the replacement node by taking the left branch from the root and then going all the way down to the right, giving me the predecessor of "frog", which happens to be "dog", and moving it to replace the root. Or, symmetrically, I can take the successor of "frog" and move it to the position of "frog". In either case, the node used to replace frog has no children, so it can be moved without further changes to the tree. Such a replacement node can actually have one child (but not two); after moving such a node, we must replace it with its child. In any case, an insertion or deletion takes essentially one search in the tree plus a constant amount of restructuring. The time spent is at most proportional to the tree depth.

Insertion and deletion change the tree structure. Indeed, a bad sequence of such operations can create an unbalanced tree, in which accesses are expensive. To remedy this we need to restructure the tree somehow, to restore it to a "good" state.

The standard operation for restructuring trees is the rebalancing operation called ***rotation***. A ***rotation*** takes an edge such as (f, k) in the tree in Figure 6 and switches it around to become (k, f). The operation shown is a right rotation; the inverse operation is a left rotation. In a standard computer representation of a search tree, a rotation takes constant time; the resulting tree is still a binary search tree for the same set of ordered items. ***Rotation*** is universal in the sense that any tree on some set of ordered items can be turned into any other tree on the same set of ordered items by doing an appropriate sequence of rotations.

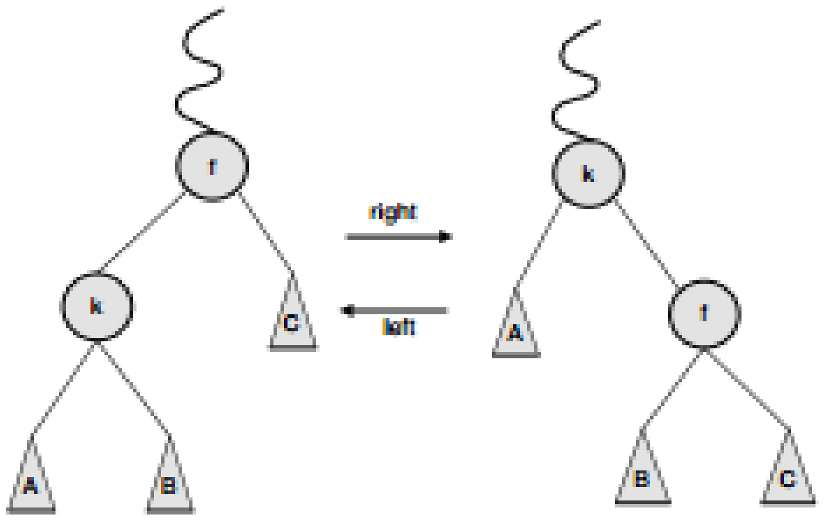

Figure 6.