Previous Chapter Return to Table of Contents Next Chapter

PART V: Advanced Data Structures

Introduction

This part returns to the examination of data structures that support operations on dynamic sets but at a more advanced level than Part III. Two of the chapters, for example, make extensive use of the amortized analysis techniques we saw in Chapter 18.

Chapter 19 presents B-trees, which are balanced search trees designed to be stored on magnetic disks. Because magnetic disks operate much more slowly than random-access memory, we measure the performance of B-trees not only by how much computing time the dynamic-set operations consume but also by how many disk accesses are performed. For each B-tree operation, the number of disk accesses increases with the height of the B-tree, which is kept low by the B-tree operations.

Chapters 20 and 21 give implementations of mergeable heaps, which support the operations INSERT, MINIMUM, EXTRACT-MIN, and UNION. The UNION operation unites, or merges, two heaps. The data structures in these chapters also support the operations DELETE and DECREASE-KEY.

Binomial heaps, which appear in Chapter 20, support each of these operations in $O(\lg n)$ worst-case time, where n is the total number of elements in the input heap (or in the two input heaps together in the case of UNION). It is when the UNION operation must be supported that binomial heaps are superior to the binary heaps introduced in Chapter 7, because it takes $\Theta(n)$ time to unite two binary heaps in the worst case.

Fibonacci heaps, in Chapter 21, improve upon binomial heaps, at least in a theoretical sense. We use amortized time bounds to measure the performance of Fibonacci heaps. The operations INSERT, MINIMUM, and UNION take only $O(1)$ actual and amortized time on Fibonacci heaps, and the operations EXTRACT-MIN and DELETE take $O(\lg n)$ amortized time. The most significant advantage of Fibonacci heaps, however, is that DECREASE-KEY takes only$O(1)$ amortized time. The low amortized time of the DECREASE-KEY operation is why Fibonacci heaps are at the heart of some of the asymptotically fastest algorithms to date for graph problems.

Finally, Chapter 22 presents data structures for disjoint sets. We have a universe of n elements that are grouped into dynamic sets. Initially, each element belongs to its own singleton set. The operation UNION unites two sets, and the query FIND-SET identifies the set that a given element is in at the moment. By representing each set by a simple rooted tree, we obtain surprisingly fast operations: a sequence of m operations runs in $O(m\,\alpha(m, n))$ time, where $\alpha(m, n)$ is an incredibly slowly growing function--as long as n is no more than the estimated number of atoms in the entire known universe, $\alpha(m, n)$ is at most 4. The amortized analysis that proves this time bound is as complex as the data structure is

simple. Chapter 22 proves an interesting but somewhat simpler bound on the running time.

The topics covered in this part are by no means the only examples of "advanced" data structures. Other advanced data structures include the following.

- A data structure invented by van Emde Boas [194] supports the operations `MINIMUM`, `MAXIMUM`, `INSERT`, `DELETE`, `SEARCH`, `EXTRACT-MIN`, `EXTRACT-MAX`, `PREDECESSOR`, and `SUCCESSOR` in worst-case time $O(\lg \lg n)$, subject to the restriction that the universe of keys is the set $\{1, 2, \ldots, n\}$.

- ***Dynamic trees***, introduced by Sleator and Tarjan [177] and discussed by Tarjan [188], maintain a forest of disjoint rooted trees. Each edge in each tree has a real-valued cost. Dynamic trees support queries to find parents, roots, edge costs, and the minimum edge cost on a path from a node up to a root. Trees may be manipulated by cutting edges, updating all edge costs on a path from a node up to a root, linking a root into another tree, and making a node the root of the tree it appears in. One implementation of dynamic trees gives an $O(\lg n)$ amortized time bound for each operation; a more complicated implementation yields $O(\lg n)$ worst-case time bounds.

- ***Splay trees,*** developed by Sleator and Tarjan [178] and discussed by Tarjan [188], are a form of binary search tree on which the standard search-tree operations run in $O(\lg n)$ amortized time. One application of splay trees simplifies dynamic trees.

- ***Persistent*** data structures allow queries, and sometimes updates as well, on past versions of a data structure. Driscoll, Sarnak, Sleator, and Tarjan [59] present techniques for making linked data structures persistent with only a small time and space cost. Problem 14-1 gives a simple example of a persistent dynamic set.

Go to Chapter 19 Back to Table of Contents