PART III: Data Structures

Previous Chapter Return to Table of Contents Next Chapter

Introduction

Sets are as fundamental to computer science as they are to mathematics. Whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time. We call such sets ***dynamic***. The next five chapters present some basic techniques for representing finite dynamic sets and manipulating them on a computer.

Algorithms may require several different types of operations to be performed on sets. For example, many algorithms need only the ability to insert elements into, delete elements from, and test membership in a set. A dynamic set that supports these operations is called a ***dictionary.*** Other algorithms require more complicated operations. For example, priority queues, which were introduced in Chapter 7 in the context of the heap data structure, support the operations of inserting an element into and extracting the smallest element from a set. Not surprisingly, the best way to implement a dynamic set depends upon the operations that must be supported.

Elements of a dynamic set

In a typical implementation of a dynamic set, each element is represented by an object whose fields can be examined and manipulated if we have a pointer to the object. (Chapter 11 discusses the implementation of objects and pointers in programming environments that do not contain them as basic data types.) Some kinds of dynamic sets assume that one of the object's fields is an identifying ***key*** field. If the keys are all different, we can think of the dynamic set as being a set of key values. The object may contain ***satellite data*** , which are carried around in other object fields but are otherwise unused by the set implementation. It may also have fields that are manipulated by the set operations; these fields may contain data or pointers to other objects in the set.

Some dynamic sets presuppose that the keys are drawn from a totally ordered set, such as the real numbers, or the set of all words under the usual alphabetic ordering. (A totally ordered set satisfies the trichotomy property, defined on page 31.) A total ordering allows us to define the minimum element of the set, for example, or speak of the next element larger than a given element in a set.

Operations on dynamic sets

Operations on a dynamic set can be grouped into two categories: ***queries***, which simply return information about the set, and ***modifying operations*** , which change the set.

Here is a list of typical operations. Any specific application will usually require only a few of these to be implemented.

SEARCH(*S, k*) A query that, given a set *S* and a key value *k*, returns a pointer *x* to an element in *S* such that *key*[*x*] = *k*, or NIL if no such element belongs to *S*.

INSERT(*S, x*) A modifying operation that augments the set *S* with the element pointed to by *x*. We usually assume that any fields in element *x* needed by the set implementation have already been initialized.

DELETE(*S, x*) A modifying operation that, given a pointer *x* to an element in the set *S*, removes *x* from *S*. (Note that this operation uses a pointer to an element *x*, not a key value.)

MINIMUM(*S*) A query on a totally ordered set *S* that returns the element of *S* with the smallest key.

MAXIMUM(*S*) A query on a totally ordered set *S* that returns the element of *S* with the largest key.

SUCCESSOR(*S, x*) A query that, given an element *x* whose key is from a totally ordered set *S*, returns the next larger element in *S*, or NIL if *x* is the maximum element.

PREDECESSOR(*S, x*) A query that, given an element *x* whose key is from a totally ordered set *S*, returns the next smaller element in *S*, or NIL if *x* is the minimum element.

The queries SUCCESSOR and PREDECESSOR are often extended to sets with nondistinct keys. For a set on *n* keys, the normal presumption is that a call to MINIMUM followed by *n* - 1 calls to SUCCESSOR enumerates the elements in the set in sorted order.

The time taken to execute a set operation is usually measured in terms of the size of the set given as one of its arguments. For example, Chapter 14 describes a data structure that can support any of the operations listed above on a set of size *n* in time $O(\lg n)$.

Overview of Part III

Chapters 11--15 describe several data structures that can be used to implement dynamic sets; many of these will be used later to construct efficient algorithms for a variety of problems. Another important data structure--the heap--has already been introduced in Chapter 7.

Chapter 11 presents the essentials of working with simple data structures such as stacks, queues, linked lists, and rooted trees. It also shows how objects and pointers can be implemented in programming environments that do not support them as primitives. Much of this material should be familiar to anyone who has taken an introductory programming course.

Chapter 12 introduces hash tables, which support the dictionary operations INSERT, DELETE, and SEARCH. In the worst case, hashing requires $\Theta(n)$ time to perform a SEARCH

operation, but the expected time for hash-table operations is $O(1)$. The analysis of hashing relies on probability, but most of the chapter requires no background in the subject.

Binary search trees, which are covered in Chapter 13, support all the dynamic-set operations listed above. In the worst case, each operation takes $\Theta(n)$ time on a tree with n elements, but on a randomly built binary search tree, the expected time for each operation is $O(\lg n)$. Binary search trees serve as the basis for many other data structures.

Red-black trees, a variant of binary search trees, are introduced in Chapter 14. Unlike ordinary binary search trees, red-black trees are guaranteed to perform well: operations take $O(\lg n)$ time in the worst case. A red-black tree is a balanced search tree; Chapter 19 presents another kind of balanced search tree, called a B-tree. Although the mechanics of red-black trees are somewhat intricate, you can glean most of their properties from the chapter without studying the mechanics in detail. Nevertheless, walking through the code can be quite instructive.

In Chapter 15, we show how to augment red-black trees to support operations other than the basic ones listed above. First, we augment them so that we can dynamically maintain order statistics for a set of keys. Then, we augment them in a different way to maintain intervals of real numbers.

Go to Chapter 11 Back to Table of Contents