9.1 INTRODUCTION: BREAKING THE lg n BARRIER

In Chapter 7 we showed that, by use of key comparisons alone, it is impossible to complete a search of n items in fewer than $\lg n$ comparisons, on average. But this result speaks only of searching by key comparisons. If we can use some other method, then we may be able to arrange our data so that we can locate a given item even more quickly.

In fact, we commonly do so. If we have 500 different records, with an index between 0 and 499 assigned to each, then we would never think of using sequential or binary search to locate a record. We would simply store the records in an array of size 500, and use the index n to locate the record of item n by ordinary table lookup.

table lookup

Both table lookup and searching share the same essential purpose, that of *information retrieval.* We begin with a key (which may be complicated or simply an index) and wish to find the location of the entry (if any) with that key. In other words, both table lookup and our searching algorithms provide *functions* from the set of keys to locations in a list or array. The functions are in fact one-to-one from the set of keys that actually occur to the set of locations that actually occur, since we assume that each entry has only one key, and there is only one entry with a given key.

functions for information retrieval

Contents
Index
Help

In this chapter we study ways to implement and access tables in contiguous storage, beginning with ordinary rectangular arrays and then considering tables with restricted location of nonzero entries, such as triangular tables. We turn afterward to more general problems, with the purpose of introducing and motivating the use first of access arrays and then hash tables for information retrieval.

tables

We shall see that, depending on the shape of the table, several steps may be needed to retrieve an entry, but, even so, the time required remains $O(1)$—that is, it is bounded by a constant that does not depend on the size of the table—and thus table lookup can be more efficient than any searching method.

The entries of the tables that we consider will be indexed by sequences of integers, just as array entries are indexed by such sequences. Indeed, we shall implement abstractly defined tables with arrays. In order to distinguish between the abstract concept and its implementation, we introduce the following notation:

table indices

Convention

The index defining an entry of an abstractly defined table is enclosed in parentheses, whereas the index of an entry of an array is enclosed in square brackets.

Thus $T(1,2,3)$ denotes the entry of the table T that is indexed by the sequence 1, 2, 3, and `A[1][2][3]` denotes the correspondingly indexed entry of the C++ array A.

example

9.2 RECTANGULAR TABLES

Because of the importance of rectangular tables, almost all high-level languages provide convenient and efficient 2-dimensional arrays to store and access them, so that generally the programmer need not worry about the implementation details. Nonetheless, computer storage is fundamentally arranged in a contiguous sequence (that is, in a straight line with each entry next to another), so for every access to a rectangular table, the machine must do some work to convert the location within a rectangle to a position along a line. Let us take a closer look at this process.

1. Row-Major and Column-Major Ordering

row-major ordering

A natural way to read a rectangular table is to read the entries of the first row from left to right, then the entries of the second row, and so on until the last row has been read. This is also the order in which most compilers store a rectangular array, and is called ***row-major ordering***. Suppose, for example, that the rows of a table are numbered from 1 to 2 and the columns are numbered from 5 to 7. (Since we are working with general tables, there is no reason why we must be restricted by the requirement in C and C++ that all array indexing begins at 0.) The order of indices in the table with which the entries are stored in row-major ordering is

(1, 5) (1, 6) (1, 7) (2, 5) (2, 6) (2, 7).

Contents
Index
Help

FORTRAN: column-major ordering

This is the ordering used by C++ and most high-level languages for storing the elements of a two dimensional array. Standard FORTRAN instead uses ***column-major ordering***, in which the entries of the first column come first, and so on. This example in column-major ordering is

(1, 5) (2, 5) (1, 6) (2, 6) (1, 7) (2, 7).

287

Figure 9.1 further illustrates row- and column-major orderings for a table with three rows and four columns.

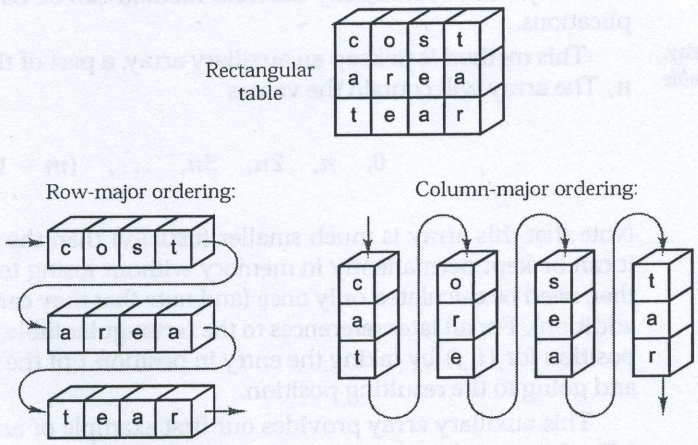

Figure 9.1. Sequential representation of a rectangular array

2. Indexing Rectangular Tables

In the general problem, the compiler must be able to start with an index (i, j) and calculate where in a sequential array the corresponding entry of the table will be stored. We shall derive a formula for this calculation. For simplicity we shall use only row-major ordering and suppose that the rows are numbered from 0 to $m - 1$ and the columns from 0 to $n - 1$. The general case is treated as an exercise. Altogether, the table will have mn entries, as must its sequential implementation in an array. We number the entries in the array from 0 to $mn - 1$. To obtain the formula calculating the position where (i, j) goes, we first consider some special cases. Clearly (0, 0) goes to position 0, and, in fact, the entire first row is easy: $(0, j)$ goes to position j. The first entry of the second row, (1, 0), comes after $(0, n - 1)$, and thus goes into position n. Continuing, we see that $(1, j)$ goes to position $n + j$. Entries of the next row will have two full rows (that is, $2n$ entries) preceding them. Hence entry $(2, j)$ goes to position $2n + j$. In general, the entries of row i are preceded by ni earlier entries, so the desired formula is

index function, rectangular array

Entry (i, j) in a rectangular table goes to position $ni + j$ in a sequential array.

A formula of this kind, which gives the sequential location of a table entry, is called an ***index function***.

3. Variation: An Access Array

The index function for rectangular tables is certainly not difficult to calculate, and the compilers of most high-level languages will simply write into the machine-language program the necessary steps for its calculation every time a reference is made to a rectangular table. On small machines, however, multiplication can be relatively slow, so a slightly different method can be used to eliminate the multiplications.

access array, rectangular table

This method is to keep an auxiliary array, a part of the multiplication table for n. The array will contain the values

$$0, \quad n, \quad 2n, \quad 3n, \quad \ldots, \quad (m - 1)n.$$

Note that this array is much smaller (usually) than the rectangular table, so that it can be kept permanently in memory without losing too much space. Its entries then need be calculated only once (and note that they can be calculated using only addition). For all later references to the rectangular table, the compiler can find the position for (i, j) by taking the entry in position i of the auxiliary table, adding j, and going to the resulting position.

This auxiliary array provides our first example of an ***access array*** (see Figure 9.2). In general, an access array is an auxiliary array used to find data stored elsewhere. An access array is also sometimes called an ***access vector***.

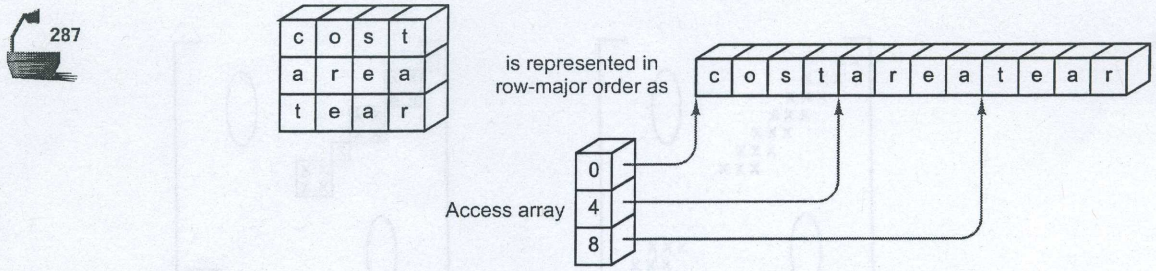

Figure 9.2. Access array for a rectangular table

Exercises 9.2

IRM

E1. What is the index function for a two-dimensional rectangular table whose rows are indexed from 0 to $m - 1$ and whose columns are indexed from 0 ot $n - 1$, inclusive, under column-major ordering?

E2. Give the index function, with row-major ordering, for a two-dimensional table with arbitrary bounds r to s, inclusive, for the row indices, and t to u, inclusive, for the column indices.

E3. Find the index function, with the generalization of row-major ordering, for a table with d dimensions and arbitrary bounds for each dimension.

Contents
Index
Help

9.4 TABLES: A NEW ABSTRACT DATA TYPE

At the beginning of this chapter we studied several *index functions* used to locate entries in tables, and then we turned to *access arrays*, which were arrays used for the same purpose as index functions. The analogy between functions and table lookup is indeed very close: With a function, we start with an argument and calculate a corresponding value; with a table, we start with an index and look up a corresponding value. Let us now use this analogy to produce a formal definition of the term *table*, a definition that will, in turn, motivate new ideas that come to fruition in the following section.

1. Functions

In mathematics a ***function*** is defined in terms of two sets and a correspondence from elements of the first set to elements of the second. If f is a function from a set A to a set B, then f assigns to each element of A a unique element of B. The set A is called the ***domain*** of f, and the set B is called the ***codomain*** of f. The subset of B containing just those elements that occur as values of f is called the ***range*** of f. This definition is illustrated in Figure 9.8.

domain, codomain, and range

292

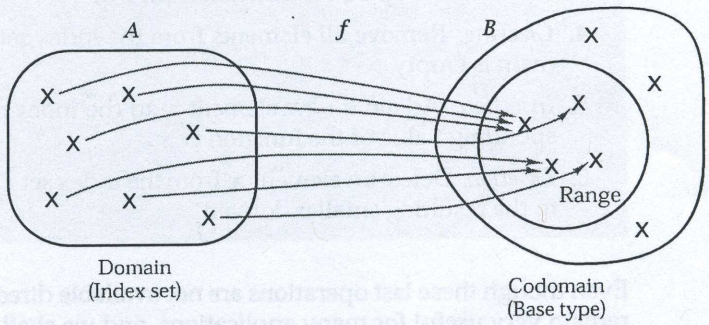

Figure 9.8. The domain, codomain, and range of a function

Contents
Index
Help

index set, value type

Table access begins with an index and uses the table to look up a corresponding value. Hence for a table we call the domain the ***index set***, and we call the codomain the ***base type*** or ***value type***. (Recall that in Section 4.6 a *type* was defined as a set of values.) If, for example, we have the array declaration

```
double array[n];
```

then the index set is the set of integers between 0 and n – 1, and the base type is the set of all real numbers. As a second example, consider a triangular table with m rows whose entries have type Item. The base type is then simply type Item and the index type is the set of ordered pairs of integers

$$\{(i, j) \mid 0 \leq j \leq i < \mathsf{m}\}.$$

2. An Abstract Data Type

293

We are now well on the way toward defining ***table*** as a new abstract data type, but recall from Section 4.6 that to complete the definition, we must also specify the operations that can be performed. Before doing so, let us summarize what we know.

Definition

A ***table*** with index set I and base type T is a function from I into T together with the following operations.

the ADT table

1. *Table access*: Evaluate the function at any index in I.
2. *Table assignment*: Modify the function by changing its value at a specified index in I to the new value specified in the assignment.

These two operations are all that are provided for arrays in C++ and some other languages, but that is no reason why we cannot allow the possibility of further operations for our abstract tables. If we compare the definition of a list, we find that we allowed insertion and deletion as well as access and assignment. We can do the same with tables.

3. *Creation*: Set up a new function from I to T.
4. *Clearing*: Remove all elements from the index set I, so the remaining domain is empty.
5. *Insertion*: Adjoin a new element x to the index set I and define a corresponding value of the function at x.
6. *Deletion*: Delete an element x from the index set I and restrict the function to the resulting smaller domain.

Even though these last operations are not available directly for arrays in C++, they remain very useful for many applications, and we shall study them further in the next section. In some other languages, such as APL and SNOBOL, tables that change size while the program is running are an important feature. In any case, we should always be careful to program *into* a language and never allow our thinking to be limited by the restrictions of a particular language.

Contents
Index
Help

3. Implementation

index functions and access arrays

The definition just given is that of an abstract data type and in itself says nothing about implementation, nor does it speak of the index functions or access arrays studied earlier. Index functions and access arrays are, in fact, implementation methods for more general tables. An index function or access array starts with a general index set of some specified form and produces as its result an index in some subscript range, such as a subrange of the integers. This range can then be used directly as subscripts for arrays provided by the programming language. In this way, the implementation of a table is divided into two smaller problems: finding an access array or index function and programming an array. You should note that both of these are special cases of tables, and hence we have an example of solving a problem by dividing it into two smaller problems of the same nature. This process is illustrated in Figure 9.9.

divide and conquer

4. Comparisons

lists and tables

Let us compare the abstract data types *list* and *table*. The underlying mathematical construction for a list is the sequence, and for a table, it is the set and the function. Sequences have an implicit order; a first element, a second, and so on, but sets and functions have no such order. (If the index set has some natural order, then sometimes this order is reflected in the table, but this is not a necessary aspect of using tables.) Hence information retrieval from a list naturally involves a search

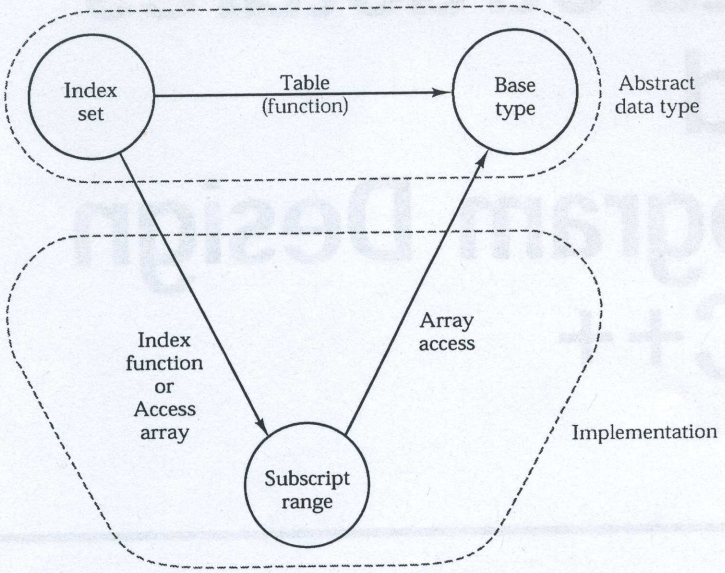

Figure 9.9. Implementation of a table

Contents
Index
Help

like the ones studied in the previous chapter, but information retrieval from a table requires different methods, access methods that go directly to the desired entry. The time required for searching a list generally depends on the number n of entries in the list and is at least $\lg n$ (see Theorem 7.6), but the time for accessing a table does not usually depend on the number of entries in the table; that is, it is usually $O(1)$. For this reason, in many applications, table access is significantly faster than list searching.

retrieval

traversal

On the other hand, traversal is a natural operation for a list but not for a table. It is generally easy to move through a list performing some operation with every entry in the list. In general, it may not be nearly so easy to perform an operation on every entry in a table, particularly if some special order for the entries is specified in advance.

tables and arrays

Finally, we should clarify the distinction between the terms *table* and *array*. In general, we shall use *table* as we have defined it in this section and restrict the term *array* to mean the programming feature available in C++ and most high-level languages and used for implementing both tables and contiguous lists.

9.5 APPLICATION: RADIX SORT

A formal sorting algorithm predating computers was first devised for use with punched cards but can be developed into a very efficient sorting method for linked lists that uses a table and queues. The algorithm is applied to records that use character string objects as keys.