Arrays

Introduction

Abstract Arrays

An **abstract array** A is a theoretical structure with the following properties.

1. There is a set of *indices*, which guide access to the data contained in A. For convenience, we assume indices are integers or tuples of integers, although they could be characters, or members of any other ordered type.
2. There is an operator **fetchA** which returns the datum of A indicated by the parameter of fetch, which is an index.
3. There is an operator **storeA** whose parameters are an index and a datum.
4. A stored datum can be retrieved by fetch. Specifally, if `storeA(i,x)` is executed, where i is an index, `fetchA(i)` will return `x`.

Modern programming languages have a built-in implementation of array. In this case `x = fetchA(i)` is implemented as $x = A[i]$, and `storeA(i,x)` is implemented as $A[i] = x$. But there are many other important implementations.

Sparse Arrays

An abstract array is called *sparse* if there are relatively few indices that point to an actual datum. If there is no datum whose index is i, `fetchi` returns a default value, such as `zero` or `nil.` We implement a sparse array using space only for the actual data.

Example. A hospital's data includes a variable of type *record* for each patient. A record contains the medical history of one patient, and also includes the patient's social security number. We declare the pointer type *patient* as follows: `typedef record*patient`

Our data form a virtual array P, where `patient fetchP(i)` returns a pointer to the record of the patient whose social security number is `i`, or a nil pointer if that record does not exist.

The virtual array can be implemented as a set of ordered pairs of the form `(i,p)`, where `i` is an index and tt p is a pointer to the record of the patient whose social security number is `i`. We do not store data for unused social security numbers. This set of pairs is stored in a search structure where the key of each pair is the index.

Memoization

In a dynamic programming problem, where $F(n)$ is computed with using previously computed values of F, it may be impractical, or even impossible, to predict which values are needed, or to compute these values in topological order. Memoization uses recursion to precompute thse predecessor values as they are needed.

Example: Collatz Stopping Time. For any positive integer n, define $f(n) = n/2$ if n is even, and $3n+1$ if n is odd. The Collatz conjecture states that, starting with a positive integer n and repeatedly applying f, you will eventually reach 1. The number of steps is called the *stopping time* of n, defined recursively as $S(1) = 0$, $S(n) = 1 + S(f(n))$ otherwise. For example, $S(13) = 9$. 40, 20, 10, 5, 16, 8, 4, 2, 1 is the Collatz sequence of 13. To compute $S(n)$ dynamically, we must know the values of $S(m)$ for all m in the Collatz sequence of n. Some of these could be larger than n, thus $S(i)$ is not a function of $S(j)$ for $j < i$. The following pseudocode describes the computation using memoization.

```
int S(int n)
  if n = 1 return 0;
  else if there is a memo (n,s)
    return s;
  else
    s = 1+S(f(n));
    store the memo (n,s);
    return s;
```

The *Collatz Conjecture* (Open) is that every positive integer has a finite stopping time. The conjecture is know to be true for all numbers up to more than a trillion. (I haven't kept up with the latest.)

Problem 1: If $S(i)$ is computed for all i up to 100, will 100 memos be stored, or more, or fewer?

Storing Arrays

If an array A[n] is declared, in a program, where, in the main memory of the machine, are the terms of A stored? We assume that main memory consists of *words*, each of which has an unsigned integer *address*. If each term of A requires one word of storage, then we expect that A[0] is stored at some address *base*, and that each $A[i]$ is stored at the address *base* $+i$.

But what about multi-dimensional arrays? Two standard methods for mapping that array onto main memory are *row major order* and *column major order*. The row-major order of A[4][3] is

A[0][0] A[0][1] A[0][2] A[1][0] A[1][1] A[1][2] A[2][0] A[2][1] A[2][2] A[3][0] A[3][1] A[3][2]

and the column-major order is

A[0][0] A[1][0] A[2][0] A[3][0] A[0][1] A[1][1] A[2][1] A[3][1] A[0][2] A[1][2] A[2][2] A[3][2].

Offset. If each term of A[n][m] is stored in one word of memory, The entry A[i][j] is stored at address base + offset(i,j), where the offset is the number of predecessors of A[i][j]. For row major order, A[i][j] has $m * i + j$ predecessors, for column major order, $n * j + i$ predecessors. If each entry of A uses d words of

storage, offset(i,j)= $d * (m * i + j)$ for row major order.

3 Dimensions. Row-major and column-major order are defined for higher dimensional arrays. Of course, there is no definition of "row" or "column" in those cases. For row-major order, the last index is incremented first, and incrementation is from right to left, like the odometer of your car. For example, the array A[2][3][2] is stored in the order:

A[0][0][0] A[0][0][1] A[0][1][0] A[0][1][1] A[0][2][0] A[0][2][1] A[1][0][0] A[1][0][1] A[1][1][0] A[1][1][1] A[1][2][0] A[1][2][1].

In column major order:

A[0][0][0] A[1][0][0] A[0][1][0] A[1][1][0] A[0][2][0] A[1][2][0] A[0][0][1] A[1][0][1] A[0][1][1] A[1][1][1] A[0][2][1] A[1][2][1].

Problem 2. What is the formula for the offset of A[i][j][k] if the array A[n][m][ℓ] is stored in row-major order, and each term takes d words of storage?

Problem 3. A 3-dimensional array $X[5][8][6]$ is stored in main memory in column-major order, with base address *base*. Each term of the array requires 4 words of storage. What is the address in memory of $X[3][5][4]$?

Triangular Arrays

The most well-known example of an abstract triangular array is Pascal's Triangle. We write $C[n][k]$ or $\binom{n}{k}$, for $0 \le k \le n$, for entry k in row n of Pascal's Triangle. $\binom{n}{k}$ is frequently called "n choose k," or "combinatorial n over k," and is the number of subsets of order k of a set of order n. Numbering of the rows, as well as entries within rows, begins with 0. For example, $\binom{6}{2} = 15$. Here are rows 0 through 6:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
```

Subsequences. $\binom{n}{k}$ is the number of strictly increasing subsequences of length k of the sequence $0, 1, \ldots, n$. On the other hand, the number of increasing (not necessarily strictly increasing) subsequences of length k of the sequence $0, 1, \ldots, n$ is $\binom{n+k-1}{k}$. We will make use of this formula later.

Triangular arrays are not stored in main memory, rather in a user declared linear array. In row-major order, entries of Pascal's Triangle will be written: 1 1 1 1 2 1 1 3 3 1 1 4 6 4 1 1 5 10 10 5 1 1 6 ... C[n][k] has $\binom{n+1}{2}$ predecessors, for example $\binom{5}{3} = 10$ and has $\binom{6}{2} + 3 = 18$ predecessors.

H igher Dimensional Triangular Matrices. An array A with 2 parameters is triangular if it every entriy $A[i][j]$ satisfies the rule $i \ge j \ge 0$. In row-major order, $A[i][j]$ has $\binom{i+1}{2} + \binom{j}{1}$ predecessors. (Note that

$\binom{j}{1} = j$.) A is a 3-dimensional triangular array if $A[i][j][k]$ satisfies the rule $i \geq j \geq k \geq 0$. The rule is that $A[i][j][k]$ has $\binom{i+2}{3} + \binom{j+1}{2} + \binom{k}{1}$ predecessors.

Exercise: How many predecessors does $A[i][j][k][\ell]$ have if A is a 4-dimensional triangular array?