

ALGORITHMS
and
DATA STRUCTURES

Derek O'Connor

CONTENTS

Chapter 1	Introduction	1
1.1	Problems, Algorithms & Data Structures	
1.2	An Algorithmic Language	
1.3	Data Structures	
1.4	Application to <i>Sparse Matrices</i>	
Chapter 2	Basic Abstract Data Types	2
2.1	Introduction	
2.2	Memory Models	
2.3	Abstract Data Types	
2.4	Linked Lists	
2.5	Stacks, Queues, and Deques	
2.6	Dynamic Memory Emulation	
Chapter 3	Tables and Hashing	3
3.1	The Table ADT	
3.2	Hashing and Hash Functions	
3.3	Collision Resolution	
3.4	Applications of Hash Tables	
3.4.1	<i>Spelling Checkers</i>	
3.4.2	<i>Travelling Salesman Problem</i>	
3.4.3	Implementation Notes	
Chapter 4	Trees	4
4.1	Introduction and Theory	
4.2	Tree Data Structures	
4.3	General and Binary Trees	
4.4	Tree Traversal Algorithms	
4.5	Binary Search Trees	
4.6	Heaps	
4.7	Application of Heaps to Sorting	
4.8	Application of Trees to the <i>Network Simplex Method</i>	
4.8.1	Implementation Notes	
Chapter 5	Graphs	5
5.1	Introduction and Theory	
5.2	Graph Data Structures	
5.3	Graph Traversal Algorithms	
5.4	Applications of Graph Traversal Algorithms	
5.4.1	Connectivity	
5.4.2	Diameter of a Graph	
5.4.3	Topological Ordering	
5.4.4	CPM/PERT Networks	
5.4.5	Markov Chains	

	3
Chapter 6 The Union-Find Set Problem	6
6.1 Introduction	
6.2 Data Structure	
6.3 Height Balancing	
6.4 Path Compression	
6.5 Applications of UFSet ADT	
6.5.1 Online Graph Connectivity	
6.5.2 Generating Random Spanning Trees	
Chapter 7 Minimum Spanning Trees	7
7.1 Introduction and Theory	
7.2 Kruskal's Algorithm	
7.3 Prim's Algorithm	
7.4 Implementation Notes	
Chapter 8 Shortest Path Spanning Trees	8
8.1 Introduction	
8.2 Theory	
8.3 Prototype Shortest Path Algorithm	
8.4 Best First Algorithms	
8.5 Breadth First Algorithms	
8.6 Implementation & Testing	
Chapter 9 Sorting	9
9.1 Lower Bounds and Information Theory	
9.2 $O(n^2)$ Algorithms	
9.3 $O(n \log n)$ Algorithms	
9.4 $O(n)$ Algorithms	
9.5 Selection	
Chapter 10 Simulation	10
10.1 Introduction	
10.2 Mathematical System Theory	
10.3 The Structure of Digital Simulators	
10.4 Discrete Event Simulators	
Appendices	107
Appendix A Mathematical Review	107
Appendix B A Pascal Primer	108
Appendix C Software Design and Programming Standards	110
Appendix D Programming Assignments	113
References	103
Index	103

Chapter 1

■ INTRODUCTION

The study of algorithms and data structures is fundamental to all areas of computer science and indeed to any subject that uses a digital computer, be it chemistry, business, or engineering.

Whether you merely use software packages or design and implement software, a good understanding of algorithms, data structures and their implementation is vital. Two ‘war stories’ may help to emphasize this point :

The first was a large consultancy job whose purpose was to write a truck-dispatching package to speed up the manual system. When the package was tested by the dispatchers they found that it took longer than the manual method. An examination of the package by an outsider found that buried deep in it was a Shortest Path Algorithm that was used many times. The implementation of this algorithm had been lifted straight out of an elementary Operations Research textbook. When this implementation was replaced by a proper one the package ran *60 times* faster.

The second story concerned a crude but important data processing system that took 30 mins to sort a few thousand records. The system was written in Basic. Five minutes work by someone who understood the implementation of algorithms and the importance of *types* reduced this time to 2 mins, even though the sorting algorithm was not changed and the person was only vaguely familiar with Basic. (Basic is only vaguely familiar with types).

These two ‘war stories’ are Irish. They have nothing to do with ‘bit-twiddling’ or code tuning. See Bentley (1986, 1989) for American stories and Bentley (1982) for code-tuning.

1.1 PROBLEMS, ALGORITHMS, AND DATA STRUCTURES

Problem-solving on a computer may be considered to be the application of an algorithm to a problem to obtain a solution. This process is shown in Figure 1.



Figure 1.1. Problem Solving.

A more detailed view is shown in Figure 2. We will explain the various pieces of this diagram as we develop the subject. In chapters 2, 3, 4 and 5 we will concentrate on parts 3 and 4 of Figure 2 in which we develop the basic abstract data types (ADTs) for lists, tables, trees, and graphs. In addition we will discuss various implementations of these ADTs as data structures in Pascal, along with simple applications programs that use these data structures. In the remaining chapters we will look at certain important applications in more detail : we will start with a mathematical description of the problem along with an outline of any theory needed to develop algorithms and ADTs. We will then implement these with appropriate data structures and procedures to obtain running programs which we will test on real data.

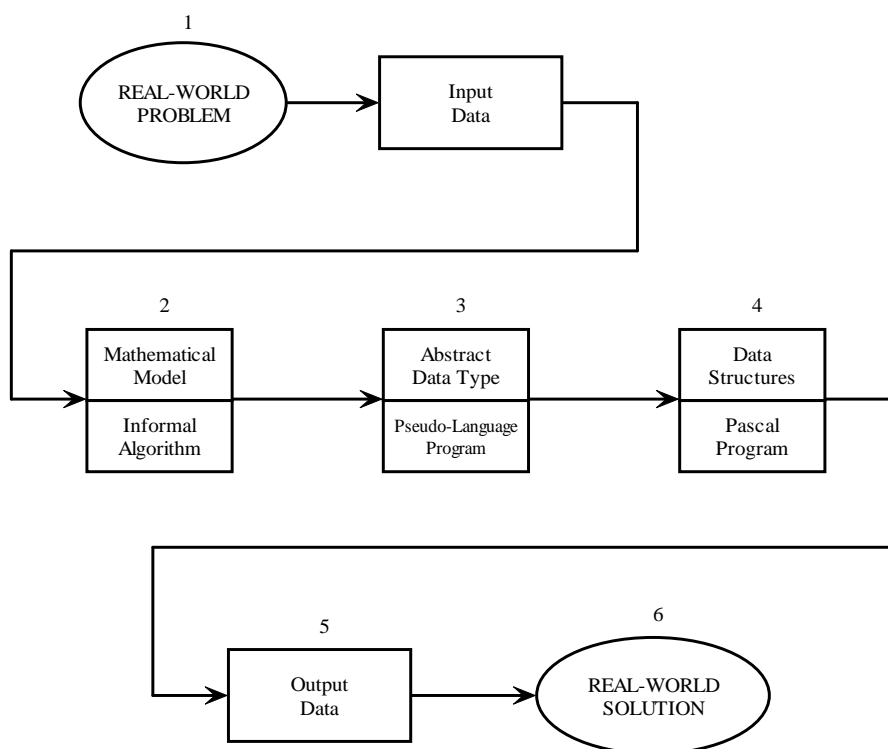


Figure 1.2. Detailed Problem Solving.

1.2 AN ALGORITHMIC LANGUAGE

We will not use a programming language as such. Instead we will use a Pascal-like pseudo-language which has the following features .

The data types are : INTEGER, REAL, BOOLEAN, POINTER (scalars); ARRAY, RECORD (non-scalar).

The control structures are

```
IF <logical expr> THEN
  S1
ELSE
  S2
ENDIF
```

```
WHILE <logical expr> DO
  S1;
  S2;
  .
  .
  Sk;
ENDWHILE
```

```
FOR <index> := <start> TO <finish> DO
  S1;
  S2;
  .
  .
  Sk;
ENDFOR
```

```
PROCEDURE <name>(< arg-list > )
  <procedure body>
ENDFUNC <name>
```

```
FUNCTION <name>(< arg-list > ):<f-type>
  <function body>
ENDFUNC <name>
```

1.3 DATA STRUCTURES

We now review the elementary data structures which are used in the construction of more complicated data structures. We then use some of these to implement some fundamental algorithms. These algorithms, in turn, will be used to implement more complicated algorithms.

Virtual Machine Model We assume that the machine on which the algorithms will be implemented has a single processor and that memory is an array of contiguous cells indexed by a single number

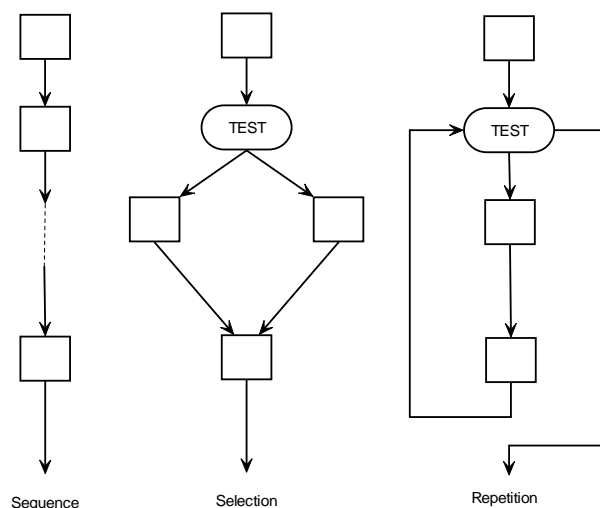


Figure 1.3. Control Structures.

(address). We further assume that address arithmetic can be performed on these addresses and that a cell can be accessed in $O(1)$ (constant) time, given its address. This is a *Random Access Machine*

1.3.1 Elementary Data Structures

Apart from scalars, there are two fundamental ways to store structures in a machine : *contiguous* or array storage, and *linked* or pointer storage.

Contiguous Storage Each element of the structure is stored beside the next as shown in Figure 1.1. Once the address of the first element is known then the address of all other elements can be calculated. This is because each element is in a fixed position relative to the first. Of necessity, the size of such a structure is *static* (fixed). The address of any element is calculated as follows :

$$Addr(a[i]) = Addr(a[1]) + i - 1$$

Access to any element requires $O(1)$ time.



Figure 1.4. Contiguous Storage.

Linked Storage Each element of the structure may be stored anywhere in memory. For this reason we must explicitly store in each element the address of the next element. This is shown in Figure 1.2. Once the address of the first element is known then the address of all other elements can be calculated. The size of such a structure is *dynamic* (variable). The address of any element is calculated as follows :

$$Addr(a_i) = NextAddr(a_{i-1})$$

Access to any element requires $O(n)$ time, where n is the number of elements in the structure.

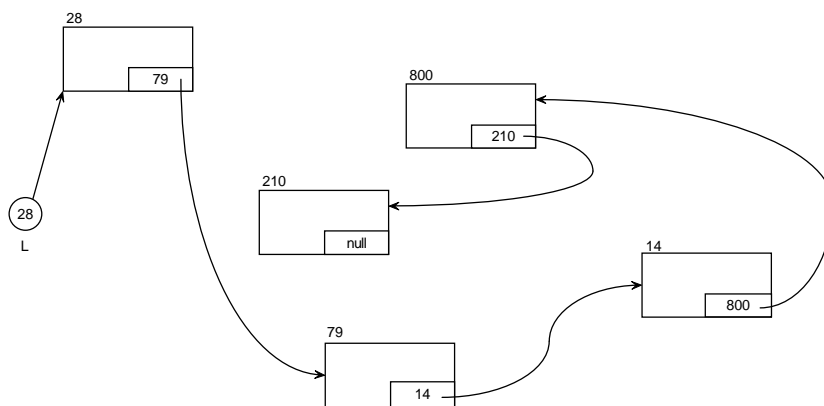


Figure 1.5. Linked Storage.

Each element of a linked structure must contain the address (link) of the next element. To avoid confusion we call this combined element a *node*, i.e., a node contains the element and the link. A node can be easily implemented as a 2-field record as follows :

```

nodetype = RECORD
    element : elemtype;
    link   : ^nodetype;
END
    
```

If $node_p$ is the address of some node then $node_p \wedge link$ is the address of the next node.

Basic Abstract Data Types The concept of *Abstract Data Type* has proved to be one of the most useful in the design and implementation of software. It was mainly a research topic in the 1970's and early 1980's and started to appear in textbooks around 1985.

There are many definitions of abstract data type but most of them agree on the essential details. Roughly speaking, an *abstract data type* is a set of objects and the operations that may be performed on the objects. The objects and operations are defined in abstract terms without specifying how the objects are represented or how the operations are performed.

EXAMPLE : INTEGERS. The set is the integers in the range $[-maxint \dots + maxint]$ and the operations are $+$ $-$ $*$ DIV MOD . These operations map pairs of integers into an integer. Notice that no details are given about how the integers are stored (e.g., 1 sign bit and 31 bits for the magnitude) or how the operations are performed.

Usually there is some structure on the set, i.e., some relationship between the objects or elements of the set. For example, the integers are totally ordered, i.e., for each pair of integers (i, j) only one of the following three relations hold : $i < j$, $i = j$, or $i > j$.

We now give a more complete definition of an abstract data type.

DEFINITION (*Abstract Data Type*).

- A set of *elements*
- A defined *structure*
- A set of *operations* defined on the structure.

The proper implementation of an abstract data type should be (1) *modular* so that different ADT's can be compiled separately; (2) *encapsulated* so that all implementation details are hidden from the user (*information hiding*) and cannot be altered by the user. This is to ensure that *the only way the user can access or manipulate the data is through the specified operations*.

The benefits of using properly implemented abstract data types are :

- Precise specification.
- Modularity
- Information hiding.
- Simplicity.
- Integrity.
- Implementation independence.

Very few programming languages have the ability to implement abstract data types properly. Ada and Modula-2 were designed with ADT's in mind. Pascal and C have *Object Oriented* extensions which can be used to implement ADT's, but these languages have not been standardized.

1.4 APPLICATION TO SPARSE MATRICES

Most physical and social structures are sparse in the sense that the elements of these structures are loosely connected. For example, the atoms of very large molecules are directly connected only to a few other atoms; towns are directly connected to 2 or 3 other towns; a person in an organization of 10,000 probably communicates with less than 10-20 people in any week.

A mathematical model of these connections is the *Adjacency Matrix* A , where $a_{ij} = 1$ if element i is connected directly to element j , and $a_{ij} = 0$ or ∞ otherwise. A more general model is : $a_{ij} \neq 0$ if element i is connected directly to element j , and $a_{ij} = 0$ or ∞ otherwise. This allows us to represent a road network by an adjacency matrix, where a_{ij} is the distance or travel time between towns i and j directly connected by a road, and $a_{ij} = \infty$ otherwise. Such adjacency matrices occur in many applications and they all have the characteristic that most of their elements are 0 (or ∞).

DEFINITION (*Sparse Matrix*). An $n \times n$ matrix is called *dense* if it has $O(n^2)$ non-zero elements, and *sparse* if it has $O(n)$ non-zero elements.

Figure 1.6 shows an example of a sparse matrix (the dots represent non-zero elements).

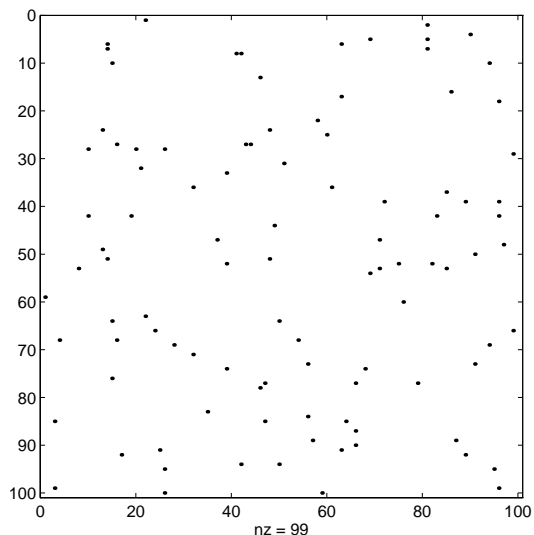


Figure 1.6. A Sparse Matrix.

In most applications involving sparse matrices the size of the matrix is very large. Typically n is in the range 1,000–1,000,000, with 2–20 non-zeros per row. Storing such large matrices is impossible, even on supercomputers. Besides, most of the storage would be wasted on zeros and, worse still, most of the calculations would be wasted on zeros. We will see in the next two sections that great savings in storage and calculations can be made if the matrix is stored in sparse form.

1.4.1 Storage of Sparse Matrices

There are many different schemes for storing sparse matrices. The choice of storage scheme will depend on the problem being solved and the algorithm used to solve it. We will concentrate on a storage scheme that is useful when using the *Successive Over-Relaxation* method for linear equations. Detailed discussions of other schemes can be found in George & Liu, (1978), and Tewarson, (1973).

A matrix A is stored in dense form by fixing the location of the first element a_{11} in memory, followed by the remaining elements of the first row, then the second row, etc., as shown in Figure 1.7. The location of any element a_{ij} is calculated as follows :

$$\text{loc}(a_{ij}) = \text{loc}(a_{11}) + n \times (i - 1) + (j - 1)$$

This calculation assumes that all n^2 elements of A are present and in a predetermined place relative to the first element.



Figure 1.7. Dense Matrix Storage.

We now describe a sparse storage scheme that stores the the non-zeros only, along with indexing information for each non-zero. The indexing information needed to locate each non-zero must be stored explicitly because the non-zeros can occur at random places in each row and so they are not in pre-determined positions relative to the first element.

The sparse storage scheme we use requires 3 arrays, assuming there are t non zeros in A : one array *Val* of length t holds the non-zeros in row order; one array *Col* of length t holds the corresponding column indices; one array *RowStart* of length $n + 1$ indicates where each row starts in the arrays above. An example of this storage scheme is shown in figure 1.8.

$$A = \begin{bmatrix} 21 & 0 & 0 & 12 & 0 & 0 \\ 0 & 0 & 49 & 0 & 0 & 0 \\ 31 & 16 & 0 & 0 & 0 & 23 \\ 0 & 0 & 0 & 85 & 0 & 0 \\ 55 & 0 & 0 & 0 & 91 & 0 \\ 0 & 0 & 0 & 0 & 0 & 41 \end{bmatrix}$$

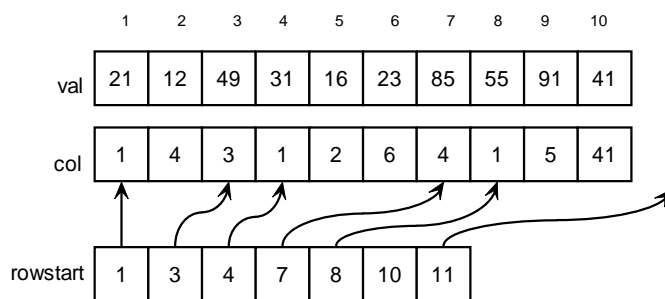


Figure 1.8. Sparse Matrix Storage.

This storage scheme requires $2t + n + 1$ boxes. It does not allow easy access to a randomly chosen value a_{ij} , but fortunately random access is rarely needed in matrix-vector computations. We now use this scheme to implement the successive over-relaxation method for linear equations.

1.4.2 Successive Over-Relaxation (SOR)

To solve $Ax = b$ iteratively we transform the equation into fixed-point form $x = Cx + d$ and use successive approximations to solve the fixed point equation thus:

$$x^{k+1} = Cx^k + d, \quad \text{given } x^0.$$

This generates a sequence of n -vectors $\{x^1, x^2, \dots, x^k, \dots\}$, which converges to the fixed point, under suitable conditions on C . The fixed point $x = Cx + d$ is a solution of $Ax = b$.

There are many ways of converting $Ax = b$ to fixed point form, e.g., *Jacobi*, *Gauss-Seidel*, and *Successive Over-Relaxation*. The Gauss-Seidel method splits A into $A = (L + D + U)$, where $L, D,$ and U are the lower-diagonal, diagonal, and upper-diagonal parts of A , respectively. This gives $(L + D + U)x = b$, which is easily transformed into fixed-point form as follows:

$$x^{k+1} := -D^{-1}Lx^{k+1} - D^{-1}Ux^k + D^{-1}b.$$

In vector component form this is

$$x_i^{k+1} := (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k) / a_{ii}$$

In correction form this is

$$x_i^{k+1} := x_i^k + (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^n a_{ij}x_j^k) / a_{ii}$$

or

$$x_i^{k+1} := x_i^k + \delta x_i^k$$

The Successive Over-Relaxation method modifies the correction term of the Gauss-Seidel method as follows

$$x_i^{k+1} := x_i^k + \omega(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^n a_{ij}x_j^k)/a_{ii}$$

When $\omega = 1$ we have the Gauss-Seidel method; with $\omega > 1$ it is called *successive over-relaxation*; with $\omega < 1$ it is called *successive under-relaxation*. The parameter ω is called the *relaxation parameter* and its purpose is to modify the spectral radius of the resulting C matrix so that convergence is faster – although still linear.

We now give a pseudo-Pascal procedure for both the dense and sparse forms of the Successive Over-Relaxation method.

algorithm *Dense-SOR*($a,b,n,itmax,\epsilon,\omega,x,flag$)

```

k := 0
while (not converged) AND k ≤ itmax do
  for i := 1 to n do
    sum := 0
    for j := 1 to n do
      sum := sum + a[i, j] * x[j]
    endfor
    x[i] := x[i] - ω * (b[i] - sum) / a[i, i]
  endfor
  k := k + 1
endwhile
endalg Dense-SOR

```

This algorithm requires $O(n^2)$ storage and $O(n^2)$ flops/iteration.

algorithm *Sparse-SOR*($a,b,n,itmax,\epsilon,\omega,x,flag$)

```

k := 0
while (not converged) AND k ≤ itmax do
  for i := 1 to n do
    sum := 0
    for j := Rowstart[i] to Rowstart[i + 1] do
      sum := sum + Val[j] * x[Col[j]]
    endfor
    x[i] := x[i] - ω * (b[i] - sum) / d[i]
  endfor
endwhile
endalg Sparse-SOR

```

This algorithm requires $O(2t+n)$ storage and $O(t+n)$ flops/iteration. For sparse matrices $t = O(n)$, and so the sparse SOR algorithm only requires $O(n)$ storage and $O(n)$ flops/iteration. This gives a substantial saving in storage and computation. Table 1.1 shows the savings for $n = 3000$ and $t = 9000$ non-zeros.

TABLE 1.1 – DENSE AND SPARSE SOR

	DENSE	SPARSE
STORAGE	36 MBytes	60 KBytes
FLOPS/ITER	9×10^6	6×10^3

A Turbo Pascal 5.5 implementation of this algorithm was tested on a 10MHz AT 80286/87 computer using a tri-diagonal matrix with $a_{i,i-1} = \alpha$, $a_{i,i} = \beta$, and $a_{i,i+1} = \gamma$. The results of this test are shown in Table 1.2.

TABLE 1.2 – SPARSE SOR 10MHZ AT

n	ITERS	TIME	TIME/ITER
500	40	10 secs	0.25 secs
1500	40	31 secs	0.75 secs
3000	40	60 secs	1.50 secs

This confirms that the time per iteration is $O(n)$.