Data Structure and Algorithm

Apurba Sarkar

August 1, 2017

1 Arrays

The first data structure that we will be discussing here is the ***Array***. The simplest form of an array is ***single dimensional array***, which may be defined as a finite, ordered set of homogeneous elements.

- By ***finite***, we mean that however large/small the array is, there exists specific number of elements in the array.
- By ***ordered***, we mean that the elements of the array is arranged so that there is a first element, second element, third element and so on.
- By ***homogeneous***, we mean that all the elements in the array must be of the same type.

To completely specify the array data structure, We must also specify how the structure is accessed. An array has two types associated with it.

- The first type is the type of the elements called the ***base type*** (or ***component type***) of the array.
- The second type is the type of the values (indexes) used to access individual elements in the array, called the ***index type***.

The number of elements in the array is called the ***size*** or the ***length*** of the array. Unless otherwise specified, we will assume the the ***index set*** of the array consists of the index $1, 2, 3, \ldots, n$. In general the length or the number of elements of the array can be obtained from index set by the formula

$$\textbf{\textit{length}} = \text{UB} - \text{LB} + 1.$$

Where UB is the largest index called ***Upper Bound*** and LB is the lowest index called the ***Lower Bound***. Note that ***length*** = UB if LB = 1.

The elements of an Array may be be denoted by

- *subscript notation:* as $A_1, A_2, A_3, \ldots, A_n$
- *parentheses notation:* as $A(1), A(2), A(3), \ldots, A(n)$ (in Pascal, Basic) or
- *bracket notation:* as $A[1], A[2], A[3], \ldots, A[n]$ (in C, C++, Java)

The number k in A_k or $A(k)$ or in $A[k]$ is called *subscript* or *index* and A_k, $A(k)$, $A[k]$ is called the subscripted variable.
Each programming language has its own *syntax* to declare an ***Array***. But, regardless of the syntax, each declaration must give three informations

- the ***name*** of the array,
- the ***data type*** of the array, and
- the ***index set*** of the array

2 Representation of Linear Arrays in Memory

Let A be the linear array in the memory. Recall that memory of the computer is simply a sequence of addressed locations as pictured bellow (Fig. 1). Let us use the notation LOC($A[k]$) to denote the address of the element $A[k]$ in the array A.

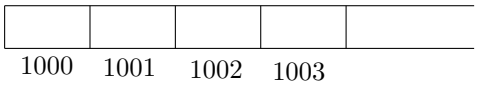


Figure 1: Representation of linear array A

As elements of array A is stored in consecutive memory cells, the computer does not need to keep track of address of every elements of the array A. It only needs to keep track of the first elements of the array A which is called the ***Base*** address of A and denoted by ***base[A]***.

The computer uses this base address ***base(A)*** of the array A to calculate the address of any element of the array by the following formula.

LOC($A[k]$) = ***base[A]*** + $w(k - \textit{Lower bound})$ where w is the size (in bytes) of each element of A. It should be noted that the time to calculate LOC($A[k]$) is essentially the same for any value of k. Most importantly, given any subscript k one can locate and access $A[k]$ without traversing any other element of A.

Example 1. *Consider the array PLACED which records the number of students get placed in each year from* 2003 *though* 2016. *Suppose PLACED appears in memory as pictured in Fig. 2. As pictured in the figure,* ***base[PLACED]*** = 200 *and* $w = 4$ *bytes per elements of PLACED.*

Then, LOC(PLACED[2003]) = 200, LOC(PLACED[2004]) = 204
LOC(PLACED[2005]) = 208 ...

The address for the array element for $k = 2010$ can be obtained from the formula
LOC(PLACED[2010]) = ***base[PLACED]*** + $w(2010 - \textit{lower bound})$
$= 200 + 4(2010 - 2003) = 228$
It is to be noted again that the content of this element can be accessed without accessing any other element of the array PLACED.

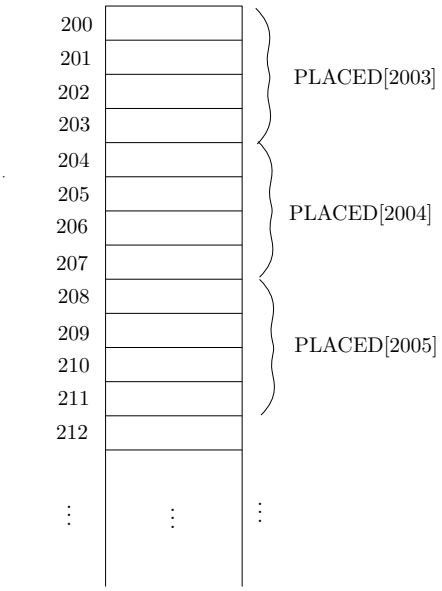


Figure 2: Representation of the array PLACED

Remark 1. *A collection of data element* ***A*** *is said to be* ***indexed*** *if any element of* ***A****,* A_k*, can be located and processed in a time that is independent of* k*. The above observation indicates that linear array is* ***indexed****. This is very important property of linear arrays. In fact,* ***Linked Lists*** *which we will cover next do not have this property.*

3 Operations on Linear Arrays

Let ***A*** be an array with lower bound **LB** and upper bound **UB**, typically following operations are performed on ***A***

1. Traversal
2. Insertion and Deletion

We will discuss algorithms to do all the above operations.

3.1 Traversal

The following algorithm (Alg. 1) traverses the linear array ***A***. The simplicity of the algorithm comes from the fact that ***A*** is a linear structure. Other linear structure, such as ***Linked List*** can also be easily traversed. On the other hand, the traversal of nonlinear structure, such as trees, graphs are considerably more involved.

Algorithm 1: TRAVERSE

```
1 Set k = LB ;                                  // initialize k
2 repeat
3 |   PROCESS A[k] ;                            // PROCESS A[k]
4 |   Set k = k + 1 ;                            // update k
5 until K > UB;
6 Exit;
```

3.2 Insertion and Deletion

Let $\boldsymbol{A}$ be a linear array with N elements. Inserting an element into $\boldsymbol{A}$ refers to the operation of addition of another element into $\boldsymbol{A}$. Similarly, referring to the operation of removing of an element from the array. The following algorithm (Alg. 2) inserts ITEM at into the k^{th} position in $\boldsymbol{A}$. the linear array $\boldsymbol{A}$.

Algorithm 2: INSERT

```
   Input: A: the array;
   N: the number of elements in A;
   k: is the positive integer ≤ N;
   ITEM: is the item to be inserted;
1  Set j = N ;                                  // Initialize counter
2  repeat
3  |   Set A[j+1] = A[j] ;                     // Move j^th element downward
4  |   j = j − 1 ;                              // decrease counter
5  until j < k;
6  Set A[k] = ITEM ;                            // Insert element
7  Set N = N + 1 ;                              // Reset N
8  Exit;
```

The following algorithm (Alg. 3) deletes k^{th} element from the array $\boldsymbol{A}$ and assigns it to a variable ITEM.

Algorithm 3: DELETE

```
   Input: A: the array;
   N: the number of elements in A;
   k: is the positive integer ≤ N;
1  Set ITEM = A[k] ;                            // Initialize counter
2  for j = k to N − 1 do
3  |   Set A[j] = A[j+1] ;                     // Move j + 1st element upward
4  Set N = N − 1 ;                              // Reset N
5  Exit;
```

4 Multidimensional Arrays

The linear arrays discussed so far are also called ***one dimensional arrays*** since each element of the array is referenced by a single subscript. Most programming languages provide two-dimensional and three dimensional arrays, i.e. arrays where elements are referenced respectively by two and three subscripts.

4.1 Representation of Multidimensional Arrays

Even though multidimensional arrays are provided as a standard object in most high level languages, it is interesting to see how they are represented in memory. Recall that memory of a computer system may be regarded as one dimensional array of bytes numbered from 1 to m. So, we are concerned with how to represent n dimensional arrays in a one dimensional memory. There are may be many plausible representation but we must select one in which the location of any arbitrary element say $A(i_1, i_2, i_3, \ldots, i_n)$, can be determined efficiently. This is necessary, since array element are, in general, accessed in a random order. In addition to that, it is also necessary to be able to determine the amount of memory space to be reserved for a particular array. Assuming that each element of array requires one word of memory, the number of words needed is the number of elements of in the array. If an array is declared as $A(l_1 : u_1, l_2 : u_2, \ldots, l_n : u_n)$, then it is easy to see that the number of elements is

$$\prod_{i=1}^{n}(u_i - l_i + 1)$$

There are two most common ways of to represent multidimensional arrays.

1. Row Major Order, (first row first)
2. Column Major Order (first column first)

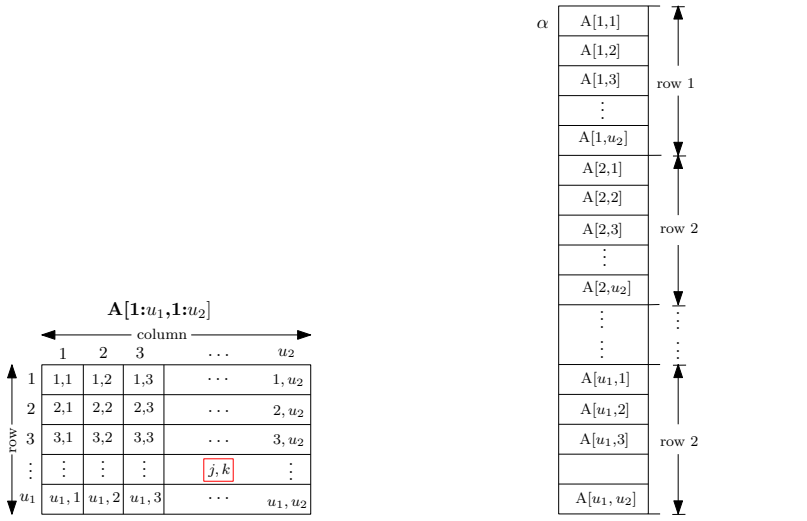
In the following sections we will first deal with representation of two dimensional array, then three dimensional array and finally n dimensional array.

4.1.1 Representation of Two Dimensional Array

Let A[$1 : u_1$, $1 : u_2$] be a two-dimensional array as pictured in Fig. 3a, the said two dimensional array may be interpreted as u_1 rows each row consisting of u_2 elements. The corresponding mapping in physical memory in row-major order is shown in Fig. 3b. It is to be observed that the elements of the first row i.e. A[1,1], A[1,2], A[1,3], ..., A[1,u_2] are stored sequentially before the elements (A[2,1], A[2,2], A[2,3], ..., A[2,u_2]) of the second row and so on. As compiler only keeps track of the ***base address*** of the 2-D array, the address of any arbitrary element A$[j, k]$ as marked in a red box in Fig. 3a is calculated as follows.

Let α be the address of the first element i.e. A[1,1] and let the size of each element be w bytes. To calculate the the address of A$[j, k]$, we need to calculate the number of elements that precedes A$[j, k]$. We see that there are $(j-1)$ rows above the j^{th} row each containing u_2 elements and there are $(k - 1)$ elements in the left of A$[j, k]$ in j^{th} row. So the total number of elements that precedes A$[j, k]$ is $(j - 1)u_2 + (k - 1)$. Each of this element is of w bytes in size. The address of the element A$[j, k]$ is then finally calculated as

$$\text{LOC(A}[j, k]) = \alpha + w[(j - 1)u_2 + (k - 1)]$$



(a) Logical view of the 2-D array (b) The 2-D array as stored in memory

Figure 3: Representation of 2-D array in row-major order

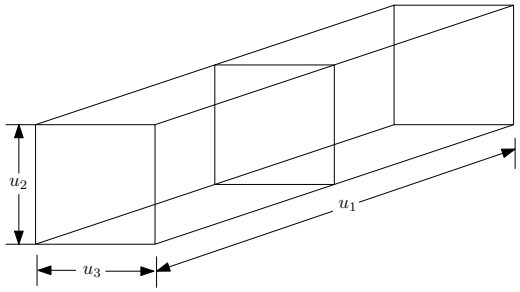


Figure 4: Logical view of the 3-D array

4.1.2 Representation of Three Dimensional Array

Let A[1 : u_1, 1 : u_2, 1 : u_3] be a three dimensional array. Array A[1 : u_1, 1 : u_2, 1 : u_3] is interpreted as u_1 2-dimensional array of size $u_2 \times u_3$. The logical view of the array A[1 : u_1, 1 : u_2, 1 : u_3] is shown in Fig. 4. To calculate the address of any arbitrary element A[i, j, k] we need to calculate the number of elements that precedes A[i, j, k] in row-major representation. We observe that there are $(i-1)$ 2-D array of size $u_2 \times u_3$ before i^{th} row and there are $(j-1)u_3+(k-1)$ (same as 2-D array discussed above) elements before A[i, j, k] in the i^{th} row. If α be the address of the element A[1,1,1] and w be the size of each element, the formula for calculating the address of the A[i, j, k] turns out to be

$$\text{LOC(A}[i, j, k]) = \alpha + w[(i-1)u_2u_3+(j-1)u_3+(k-1)]$$

4.1.3 Representation of n Dimensional Array

Generalizing the preceding discussion on 2-D and 3-D arrays, the addressing formula for any arbitrary element $\mathrm{A}(i_1, i_2, i_3, \ldots, i_n)$ in an n-dimensional array declared as $\mathrm{A}[1 : u_1,\ 1 : u_2,\ 1 : u_3,\ \ldots,\ 1 : u_n]$ can be easily obtained. If α is the address for $\mathrm{A}(1, 1, 1, \ldots, 1)$ then formula for address of the element $\mathrm{A}(i_1, i_2, i_3, \ldots, i_n)$ is calculated as follows

$$
\begin{aligned}
\mathrm{Loc}(\mathrm{A}(i_1, i_2, i_3, \ldots, i_n)) \quad &= \alpha \quad +(i_1 - 1)u_2u_3 \ldots u_n \\
&\qquad\quad +(i_2 - 1)u_3u_4 \ldots u_n \\
&\qquad\quad +(i_3 - 1)u_4u_5 \ldots u_n \\
&\qquad\quad \vdots \\
&\qquad\quad +(i_{n-1} - 1)u_n \\
&\qquad\quad +(i_n - 1) \\
&= \alpha \quad +\sum_{j=1}^{n}(i_j - 1)a_j \quad \text{with} \begin{cases} a_j = \prod_{k=j+1}^{n} u_k & 1 \le j < n \\ a_n = 1 \end{cases}
\end{aligned}
$$

5 Sparse Matrices

A matrix is a mathematical object that appears in many practical problems. As computer scientists, we are interested in studying ways to represent matrices so that the operation to be performed on them can be carried out efficiently. A general matrix consists of m rows and n columns. In general we write $m \times n$ (read m by n) to designate a matrix with m rows and n columns. Such a matrix has mn elements. When m is equal to n, we call it a square matrix. It is very natural to store a matrix in a two dimensional array, $\mathrm{A}(1 : m, 1 : n)$. We can work with any element by writing $\mathrm{A}(i, j)$; and this element can be found very quickly (we have already explained in earlier section). However, Consider the following matrix, we see that it has many zero entries. Such a matrix is called sparse. Although there is no precise definition of when a matrix is sparse and when it is not, but, if in a matrix 70% entries are zero, then it is intuitively considered as sparse. In the matrix bellow 8 out of 36 entries are zero and it is sparse.

$$
\begin{vmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{vmatrix}
$$

Figure 5: A sparse matrix

Sparse matrix requires us to consider an alternate form of representation. This is because many of the matrix we want to deal with in practice are large e.g. 1000×1000, but at the same time they are sparse. Even today, it is impossible to load a full 1000×1000 matrix at once in the memory of the most of the computers. In the alternate representation we only store non-zero elements.