Arrays and Linked Nodes

As mentioned in Handout 1, an abstract data type (ADT) provides a programmer with an interface or set of operations. An ADT hides from the programmer the details of how the operations are implemented and how the data within objects of that type are represented. The terms data structure and concrete data type refer to the internal representation of an ADT's data. This handout covers operations on two data structures that are used to implement abstract data types: ***arrays*** and ***linked nodes***. Later, we use both of them to implement several ADTs and assess the time/space trade-offs in using these two kinds of data structures.

Characteristics of Arrays

The array is probably the most commonly used data structure in programs. In this section, we discuss several characteristics that programmers must keep in mind when using an array to implement an ADT.

Contiguous Memory and Random Access

As you know, an array represents a sequence of items that can be accessed by index position. The index operation makes storing or retrieving an item at a given position easy for the programmer. The index operation is also ***very fast***. ***Indexing*** is a random-access operation, During random access, the computer obtains the location of the i^{th} item by performing a constant number of steps. Thus, no matter how large the array, it takes the same amount of time to access the first item as it does to access the last item.

The computer supports random access for arrays by allocating a block of contiguous memory cells for the array's items. One such block is shown in Figure 5.1

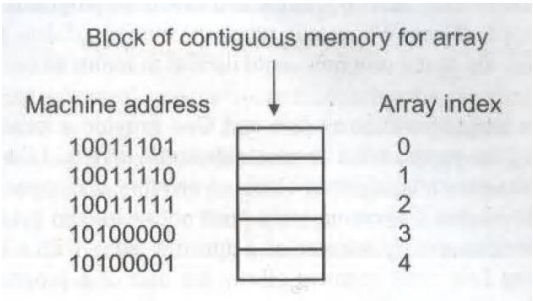


Figure 5.1 A block of contiguous memory

For simplicity, the figure assumes that each data item occupies a single memory cell, although this is not often the case.

Because the addresses of the items are in numerical sequence, the address of an array item can be computed by adding two values: the array's base address and the item's offset. The array's base address is the machine address of the first item. An item's offset is equal to its index multiplied by a constant representing the number of memory cells required by an array item. To summarize, the index operation has two steps:

```
Fetch the base address of the array's memory block
Return the result of adding the index * k to this address
```

For example, suppose the base address of an array's memory block is 10011101 and each item requires a single cell of memory. Then the addresses of the first three data items are 10011101, 10011110, and 1001111, respectively.

The important point to note about random access is that the computer does not have to search for a given cell in an array, where one starts with the first cell and counts cells until the i^{tch} cell is reached. Random access in constant time is perhaps the most desirable feature of an array. However, this feature requires that the array be represented in a block of contiguous memory. As we will see shortly, this requirement leads to some costs when we implement other operations on arrays.

In linked object, only sequential access is possible, so no fast indexing is possible.

For multidimensional arrays or triangular arrays, the mapping of multidimensional address to linear address of the memory cells is achieved through appropriate mathematical formulae.

Static Memory and Dynamic Memory

Arrays in older languages such as FORTRAN and Pascal were static data structures. The size of the array was determined at the compile time, so the programmer needed to specify this size with a constant. Because the size of an array could not be changed at run time, the programmer needed to predict how much array memory would be needed by all applications of the program. If the program always expected a known, fixed number of items in the array, there was no problem. But in the other cases, where the number of data items varied, programmers had to ask for enough memory to cover the cases where the largest number of data items would be stored in an array. Obviously, this requirement resulted in programs that wasted memory for many applications. Worse still, when the number of data items exceeded the size of the array, the best a program could do was to return an error message.

Modern languages such as Java and C++ provide a remedy for these problems by allowing the programmer to create dynamic arrays. Like a static array, a dynamic array occupies a contiguous memory and supports random access. However, the size of a dynamic array need not be known until run time. Thus, the programmer can specify the size of a dynamic array with a variable. For example, the following Python code segment allows the user of a program to input the size of an array from the console:

```
size = input("Enter the size of array")
array = [0] * size
```

In addition, the programmer can readjust the size of an array to an application's data requirements at run time. These adjustments can take three forms:

1. Create an array with a reasonable default size at program startup.
2. When the array cannot hold more data, increase its size.
3. When the array seems to be wasting memory (some data have been removed by the application), decrease its size.

To resize an array, the procedure is: a new desired size array is allocated, copy data from old to new array, delete the old array, and assign original name to new array.

In linked object, memory is allocated as and when required and deallocated when not required, so it is dynamic by the nature.

Physical Size and Logical Size

When working with an array, programmers must often distinguish between its **physical size** and its logical size. The physical size of an array is its total number of array cells, or the number used to specify its capacity when the array is created. The logical size of an array is the number of items in it that should be currently available to the application. In cases where the array is always full, the programmer need not worry about this distinction. However, such cases are rare. Figure 5.2 shows three arrays with the same physical size but different logical sizes. The cells currently occupied by data are shaded.

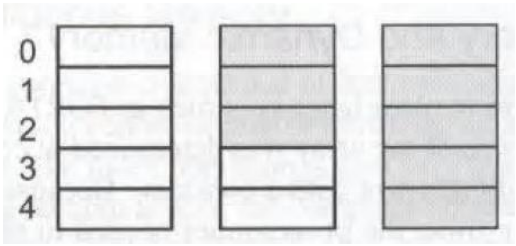
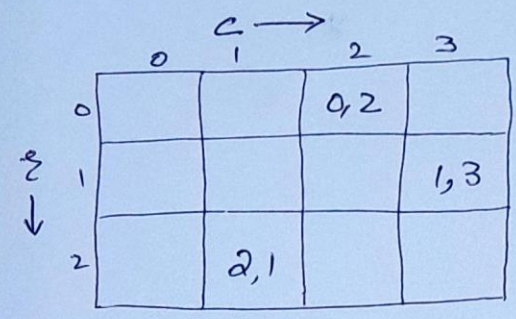


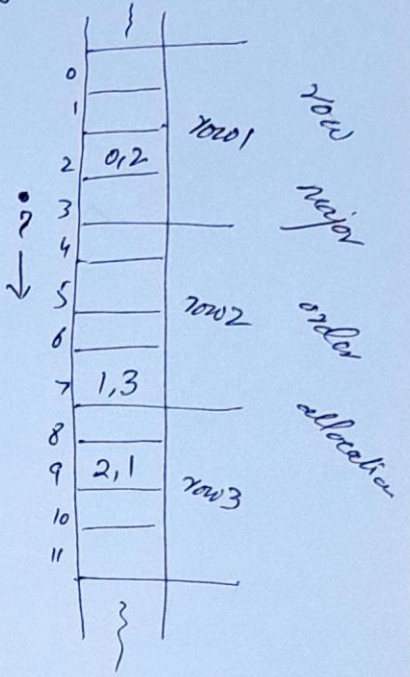
Figure 5.2 Arrays with different logical sizes

As you can see, it is possible to access cells in the first two arrays that contain garbage, or data not currently meaningful to the application. Thus, the programmer must take care to track both the physical size and the logical size of an array in most applications.

In linked object, the logical size is always proportional to the physical size of the collection, whereas in the arrays, the logical size <= physical size.

Let 3×4 size, 2D array as follow





$$i = r * \text{rowsize} + c$$

$$\Rightarrow \quad 0*4+2 = 2$$
$$1*4+3 = 7$$
$$2*4+1 = 9$$

1D: i^{th} element of array of size S1 mapped to i^{th} element of linear addressing

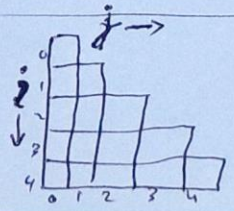
2D: i^{th}, j^{th} element of array of size S1, S2 mapped to $(i*S2+j)^{th}$ element of linear addressing

3D: i^{th}, j^{th}, k^{th} element of array of size S1, S2, S3 mapped to $(i*S2*S3+j*S3+k)^{th}$ element in linear add—

4D: ... $(i*S2*S3*S4+j*S3*S4+k*S4+l)$ ...

and we can generalize it.

---

for triangular array 

$\left(\frac{i(i+1)}{2}+j\right)^{th}$ is linear address of i^{th}, j^{th} coll.