Implementation of Abstract Data Types with Arrays of Unbounded Dimensions

by Alejandro Barrero

The type array is the most widely used structured type in programming languages. Most programming languages have the type array, and many languages have incorporated this type after their creation; case in point Lisp. Arrays are very useful to model some important mathematical concepts like n-tuples, vectors and matrices, and non-mathematical concepts like tables and lists of limited size. The implementation of arrays proved to be easy from the earlier compilers on. Actually, even before the advent of high-level languages, programmers were using tables of fixed size taking advantage of index registers (and before index registers doing arithmetic on the instructions that accessed the elements). The reason that arrays have been part of programming languages since the earliest times is that they are easy to implement because they have a fixed size. Therefore, arrays are easier to allocate in static storage or in stack storage. Furthermore, arrays are perfect when the number of elements needed is fixed as in the primary colors or the days of the week. However, many of the abstract concepts that we model with arrays (sequences, lists, stacks, queues, heaps, etc.) are of unbounded capacity. We have learned to develop programs taking into account the finiteness of the computers that execute them; we have learned to accept the existence of the largest integer that can be used, and only in special circumstances we use packages for infinite-precision arithmetic. But our programs exceed the dimensions of our arrays much more often than our integer variables exceed the value of the largest integer in a machine. Also, it is desirable to design programs that treat lists, stacks, etc., as unbounded entities.

Another problem with arrays of fixed dimensions are programs that crash, or refuse to execute, because the dimensions or an array are exceeded. If the source code is available it is possible to increase the dimensions and recompile the program; it is a hassle, but it is still possible to use the program. The situation is much worse when the program is only available as an executable file; then it is necessary to reformulate the problem and reorganize the data, or to use a more powerful program. This is not an uncommon situation; it is present in many utility programs (Unix’s sort truncates lines longer than an implementation dependent maximum and Unix’s lex has an input buffer limited to 200 characters), we suffer its consequences when we reach the limit on the number of files that can be open in an operating system, and some times we are embarrassed when demonstrating the capabilities of our programs.

What happens is that arrays of fixed dimensions are one of the limitations of earlier computer technology that we still carry as a burden; we had to use fixed dimensions because we had to lay out our data on real memory. Of course, modern programming languages have dynamic range; arrays whose dimensions are specified at execution time (created in the activation record of program units). But dynamic arrays are not the solution to all problems; the dimensions still can be exceeded and the programs will crash. With few exceptions (Common Lisp and AWK have arrays that may grow dynamically, and SETL has the type sequence) programming languages do not support arrays with dimensions that are increased dynamically. Also, the conventional use of arrays carries with it a subtle inefficiency that passes unnoticed very easily; programs are designed with arrays sufficiently large for most uses; e.g., if an array is used to hold lines of text most programmers would consider that the usual length on terminals is 80 characters, and that commercial programs produce output on lines of 130 characters, and would chose a capacity of 200 characters for added safety. The programs then execute correctly most of the time, but storage goes allocated and unused. Suppose that a program uses an array of fixed dimensions. Then if in the execution of the program all the sizes up to the dimensions of the array are equally likely to be used, on the average only half of the array is used (50% efficiency). On the other hand, if the likelihood of using small sizes is large the situation is even worst. Granted, this is not a great problem in times of large memories and virtual storage, but such inefficiency should be avoided if possible.

The conventional alternative to using arrays in the implementation of Stacks and Queues has been using linked lists, but the implementation of the operations takes more machine instructions, and the overhead of memory management may become important; it is linear on the number of elements created unless the method presented at the end of this paper is used. In the methods presented here the overhead of memory management is logarithmic on the number of elements created in a data structure (a distinct advantage if this number is large). The
purpose of this communication is to present efficient methods to implement arrays of unbounded dimensions and elementary ADTs (stacks, queues, and heaps). The ideas are also used to create a very efficient scheme for dynamic allocation of records of fixed size. The complexity of the implementations is analyzed, both in the worst case sense and in the amortized sense. The effect of unbounded dimensions is achieved with a judicious use of storage allocation, and it can only be used in programming languages that allow dynamic creation of arrays. The appendix contains code for the implementations. A question that may arise at this point is: Since the research in Computer Science is directed towards improving the complexity of algorithms and data structures, why be concerned with a more efficient implementation of elementary ADTs that will not change the complexity of those algorithms and data structures? The answer is that elementary ADTs are used in many complex algorithms and data structures, and any inefficiency in the elementary ADTs translates to the algorithms or data structures that use them.

Implementation of Arrays of Unbounded Dimensions

The main limitation of arrays of fixed dimensions is that they cannot be used to implement the mathematical concept of a sequence, which implies an unbounded number of elements, the linguistic concept of a string, or the common notion of a list, which connotes an arbitrary number of elements. In fact, only languages specifically designed to process strings, like SNOBOL, provide built-in operations on strings, while general purpose programming languages are lacking in this respect. The result has been non-existing or very poor support for operations on character strings. Surprisingly, very few people have indicated the need for arrays of unbounded dimensions. Dijkstra defines arrays of unbounded dimensions in his language of guarded commands, and uses them in his algorithms [1]. Also, Tarjan defines a type list, which is an array of unbounded dimensions, as an elementary object and uses it in his algorithms [4].

Arrays with dimensions that grow dynamically cannot be implemented in static or in stack storage because to have efficient direct access to the elements all must be in contiguous space, and the extra space contiguous to the array is in general in use (the exception is the top of the stack). That this space is free cannot be guaranteed even in heap storage. If this space is not free it is necessary to allocate space for the new size, and copy the array to this new space. The time involved in this operation has two parts: in the first place there is the overhead of memory management, and in the second place there is the time necessary to copy the array to the new location. Of course, this idea can be implemented in a number of ways with varying degrees of efficiency, and if the number of times that the size of the array is increased is small (say one or two times) efficiency is not an issue. But a truly general method should be efficient also in the asymptotic case where the number of times that the size is increased is large. Another important consideration is the manner in which the size of the array is increased from an original value \( n_0 \) to a final value \( n_m \), where \( m \) is the number of times that memory management is used to allocate new space and return the old one. In the best case the first element referenced that requires increasing the size of the array corresponds to the largest index used, and the array grows to the final size in one step. In the worst case the size of the array needs to grow by one element at a time (this corresponds to the use of the array as a sequence or as a stack). We will consider several methods for increasing the size of the array, and will analyze their behavior in the worst case.

The first method that may come to mind to increase the size of the array is to use a fixed size increment \( \Delta n \) each time. Then the values of size are: \( n_0, n_0 + \Delta n, n_0 + 2\Delta n, \ldots, n_m = n_0 + m\Delta n \). The overhead in memory management is proportional to the value of \( m \) which is \((n_m - n_0)/\Delta n\). The first time that the size of the array is increased it is necessary to copy \( n_0 \) elements, the second time \( n_0 + \Delta n \) elements, and so on, until for the last time \( n_0 + (m - 1)\Delta n \) elements are copied. This is an arithmetic progression with sum \( n_0m + \Delta n(m - 1)m/2 \). The number of cells in the array is \( n_m \), but this is not the important variable because an array of this size may have been created to accommodate just one more element, so in the very worst case the number of cells used is \( c = n_{m-1} + 1 = n_m - \Delta n + 1 \). Now, in terms of the number of elements used the number of times that elements are copied is \( \Theta(c^2) \). Thus, the overhead for memory management is linear on the number of cells used, and the number of copy operations is quadratic. Clearly this is a poor method on both counts. The only advantage is that on the average the unused space is small, \( \Delta n/2 \). On the other hand, if the value of \( \Delta n \) is very large the number of times, \( m \), that memory management is invoked is very low, and the degradation in performance would not be perceived. This is the method chosen in some systems as the GNU EMACs editor that recognize the need reallocate arrays, and are satisfied with a simple solution. However, even these systems would benefit from the methods proposed here.
The second idea is to double the size of the array every time with a view to gain a logarithmic advantage on the overhead for memory management. We will call this method the binary method. In this case the values of size are: \( n_0, 2n_0, 4n_0, \ldots, n_m = 2^m n_0 \). Again, the last doubling of the dimensions may have been done to add only one element, thus the number of cells used is \( c = 2^{m-1} n_0 + 1 \). Here the number of times that memory management is invoked is \( m = \log_2((c - 1)/n_0) + 1 \), and the number of times that elements are copied corresponds to the sum of the elements of a geometric progression, giving \( 2^m n_0 - n_0 = 2(c - 1) - n_0 \) (see Figure 1). This is a much better result; now the overhead for memory management is logarithmic on the number of cells used (which is negligible asymptotically), and the overhead for copying elements is linear (the constant is 2). The net effect of this method is that if the elements of the array are written only once, as in the construction of a sequence, the cost of each write operation is three times as much as usual; one time for writing and two times for copying. Notice, however, that this is a worst case analysis taking into consideration only operations to write elements in the array. If cells in the array are used to hold different data items at different times (if there are several write operations to the same cell), or if the cells are written to once but the data is read many times the efficiency of the implementation is much better. The analysis of space overhead is simple; if all sizes of the sequence are equally possible, then on the average half of the final size. This compares favorably with the use of a large array of fixed dimensions where in the average half of the total array will go unused.

On the previous method the size of the array grows exponentially on base 2. An extension of this method would be to use a large base \( b \). In this case very similar lines of analysis lead to \( n_m = b^m n_0, c = b^{m-1} n_0 + 1, m = \log_b((c - 1)/n_0) + 1 \), and the number of copies of elements is \((c - 1) p/(p - 1) - n_0/(p - 1)\). The overhead of memory management is logarithmic on base \( b \), and the number of elements copied is still linear with a constant of proportionality \( p/(p - 1) \), which means that in the worst case every write operation has an overhead of \( p/(p - 1) \) additional write operations —this value is only 1.25 for \( b = 5 \). We conclude then that the larger the value of \( b \) the lower the time overhead. On the other hand the space overhead gets worse because the size of the last increment grows with \( b \), the fraction of unused space is \((b - 1)/2b\). This is yet another example of trading time for space.

The last method is based on the use of Fibonacci numbers. The idea is that every time that the size of the array must be increased the new size is the sum of the previous two sizes. This method will be called the Fibonacci method. The values of size are: \( n_0, 2n_0, 3n_0, \ldots, f_m n_0 \), with \( f_k \) the \( k \)th Fibonacci number. The number of cells used in the worst case is \( c = f_m + 1 \), and the number of times that elements are copied is \( n_0(1 + f_3 + f_4 + \ldots + f_m) = n_0(f_m + 3 - 1) \). In this case too the overhead for memory management is logarithmic, with base the golden ration \( \phi = 1.61803 \), and the number of times that elements are copied can be written as \( n_0(2f_m + 1 + f_m^{-1})/n_0 f_m + 1 + 1 \) to the number of cells used. This value has a very interesting behavior; assuming that \( n_0 > 1 \) the ratio is 2 for \( m = 1 \) and \( m = 2 \), it is 2.333 for \( m = 3 \), and for larger values approaches \( \phi^2 \approx 2.61803 \) as \( m \) approaches infinity. This means that the method is initially equivalent to the method of doubling the size, but later the overhead of copying increases and approaches 2.61 operations per used cell. The average of unused cells can be computed under the same assumptions as before; this number is the difference between the last two sizes (the last increment) divided by 2, that is \((f_m + 2n_0 - f_m + 1 n_0)/2 = f_m n_0/2 \). The percentage of unused space is then \( f_m f_m + 2 \). This number starts at 0.25 just as in the case of doubling the size, drops to 0.1666 at the next step, and then stays around .19. The conclusion is that the Fibonacci method is equivalent to the binary method for small values of \( m \) (the size is increased only a few times), and that for larger values the time overhead is larger 2.6 versus 2, while the space overhead is smaller .2 versus .25. Here the trading of time for space is very even.
Implementation of a Stack with an Array of Unbounded Dimensions

Arrays provide an excellent means for the implementation of stacks, queues, and heaps. As an example, let us consider the implementation of a stack. With an array the code is simple and efficient (Figure 2(a) reviews the basic ideas). The only drawback is that if the dimensions of the array are exceeded the program halts with the message: “STACK OVERFLOW”, which is absolutely incorrect (stacks cannot be overflowed). Before the invention of linked lists programs that used several stacks could crash when one of the stacks overflowed even if there was free space in the other stacks. The trick for implementing two stacks in a single array was quickly discovered, and a very ingenious method was developed for implementing three or more stacks in a single array and moving them around to allocate more space to a stack that needed it [2, 6]. However, this method is very complicated, and the performance deteriorates when the array comes close to being full. The conventional solution to this problem is to use a linear linked list instead of an array, and to add and remove records at the head of the list to implement push and pop operations (see Figure 2(b)). This is the method proposed in elementary books on data structures, but if implemented naively creating and destroying records as needed the overhead of storage allocation is very large. A good solution has been known since the earliest times as expounded by Knuth [3], which consists in creating a pool of free records with the elements of an array. The method is excellent as long as the size of the array is not exceeded, but the problem remains as to what to do when this happens. Exactly the same problems occurs in the implementation of a queue, and again the solution is to use a linked list. On the other hand, a linear linked list is not a good idea for the implementation of a heap because both the sorted list and the unsorted list are inefficient. Of course, very efficient implementations of heaps have been developed using linked lists, e.g. Binary Heaps, but the code is much more complicated.

The problem with the previous implementation of a stack is trying to push an element when the array is full. The first solution that we will consider is to use an array of unbounded dimensions. Anyone of the methods discussed in the previous section could be used, but we will concentrate the discussion on the binary method (see Figure 3).

![Figure 2. Elementary implementations of a stack](image_url)
The first array created has twice the size of the original array, the second has four times the size, and the $m$th has $2^m$ times the size. Now, since memory management is invoked $m$ times, and the size of the last array is $5 \times 2^m \times$ size of original array, then

$$m = \log_2 \left( \frac{\text{size of last array}}{\text{size of original array}} \right).$$

Given that the size of the array is less than or equal to the number of push operations since the creation of the stack (with equality holding in the absence of pop operations), and this is less than or equal to the total number of stack operations, then

$$m = O(\log_2 \text{stack operations}).$$

To show the efficiency of this implementation of a stack we will use the bankers’ view of amortized complexity [5], and we will define a token as the cost (time) of writing an element into an array. We assign three tokens to every push operation; one token to pay for the writing of data (the cost of the operation), and two tokens to be saved, and pay for the cost of rewriting data later on. We assign no tokens to operations that examine the top of the stack, or pop elements from it, because no writing of elements takes place in these operations. The first time that the array is copied there are two tokens in the account for each element in the array, more than enough to pay for the cost of copying. On consecutive times there are two tokens in the account for each element in the top half of the array, and this is exactly the number required to pay for the copy of all elements (Figure 4). Therefore, the amortized cost of a sequence of $k$ operations is less than or equal to $3k$ (the number of tokens that must be assigned to $k$ push operation) because the number of tokens in the account never becomes negative. Thus, the time complexity of a sequence of $k$ operations is $O(3k)$, and the actual time taken by the $k$ operations is less than or equal to three times the time required if an array of sufficient size had been used.

This implementation of a stack and the analysis using amortized complexity seem to be part of the folk-
lore of Computer Science; many people are aware of the ideas but there are no written references (at least the author could not find any). Actually, it was this idea that prompted the development of this paper, but the extension to larger bases, the Fibonacci method, and the remainder of the paper are original.

**Implementation of Heaps and Queues**

The implementation of a heap follows the same ideas as the implementation of a stack; as soon as an array is full we create a new one of larger size, and we copy the old array into the new one. The overhead of memory management, and the cost of copying array elements are exactly as before. Even the amortized analysis is similar, but since there is so much data movement in the implementation of heap operations, it is necessary to use a standard trick in amortized analysis to avoid considering those operations: compare the behavior of the amortized data structure and another data structure. Here we compare the implementation creating and copying arrays to an implementation using an array with more elements than are needed. What we want to compute is the extra time required by the new implementation. The analysis is done assigning two tokens to every insert operation, to be saved and pay for the cost of copying the array (the cost of all the operations is the same as in the other implementation). Then it is clear that every time that the array needs to be copied the top half of the elements are associated with two tokens each, and that is enough to pay for the cost of copying both halves (Figure 4 again). It is noteworthy of this implementation that since the complexity of a sequence of $k$ operations is usually $k \log k$ (the exception is the creation of a heap which is linear in $k$), the overhead of rewriting the array, being linear, is asymptotically unimportant.

The implementation of a queue is also similar; as long as it is not required to insert an element into a full array, the array is used as a circular array. When it is necessary to copy the array into a larger one the elements are not copied from the beginning of the old array, but from the front of the queue. The reason for this is that the rear of the queue plays the role of the first location of the circular array.

**Implementations of Stacks and Queues Using a Linked List of Arrays**

There is an even better way to implement stacks and queues with arrays of unbounded dimensions. The idea is to keep a linked list of records containing arrays. Here, instead of copying the elements of the old array into the new one, the new array is simply linked in front of the old one. The list is kept double linked to be able to move over

![Figure 5. Implementation of a stack with a linked list of arrays](image1)

![Figure 6. Implementation of memory management with a linked list of arrays](image2)
the array forwards and backwards. Increasing the size of the arrays by a factor, or with Fibonacci numbers, maintains the overhead of memory management logarithmic on the total size of the array (and on the number of operations on the ADT). But since there is no rewriting of elements, the complexity of the operations, actually the time of execution, is the same as if a very large array was used (except for the overhead to move from one record to another which is logarithmic on the number of operations). Figure 5 shows the ideas in the implementation of a stack. The secret to match the speed of the conventional implementation is to access the elements directly by pointers to the elements themselves, instead of indirectly through pointers to the records (to access the specific array and then use an index).

In the implementation of a queue we do not use a circular array, but instead, when an element must be added to the rear, and the current array is full, a larger array is created. When a pop operation is executed in the stack, we move backward over the array, and eventually we may leave a record with an empty array on the top of the stack (to be used later). In the queue we remove elements from the front (which would correspond to the bottom of the stack), and we move forwards when adding elements. The empty arrays left behind cannot be used in the normal course of operations. Therefore, when records with empty arrays are created it is necessary to move them to the rear of the list. The overhead is bounded by the logarithm of the total size of the array, and the execution of the operations takes exactly the same time as in the case of an array of fixed dimensions.

**Efficient Storage Allocation with Arrays of Unbounded Dimensions**

Many programs require the allocation and disposal of records of fixed size. The simplest way to code these programs is to invoke memory management to create new records when needed, and to dispose of them when no longer needed. But, the overhead for memory management may become substantial, and it is desirable to find ways to minimize it. The traditional solution is to declare an array of records and construct a pool of available records. Thus, records are removed from the pool and when no longer needed they are returned to the pool. This idea provides good results as long as the number of records needed at any one time does not exceed the size of the array. Therefore, programs would be designed using arrays (considered) large enough for all applications with the problems that been mentioned from the beginning of this paper. The solution is rather easy: extend the conventional idea with the use of a linked list of arrays of records, creating arrays of increasing sizes every time that a record is needed and the pool is empty. In a straightforward implementation of this idea the records of a new array would be linked to form a pool of available records, but there is a technique to avoid this overhead (also part of the folklore in Computer Science). In this method (shown in Figure 6) some of the records of the array are being used, some have been placed in the pool of available records, and others have never been used. All the records that have been used (that are currently in use, or are in the free list) are located at the beginning of the array. The records that have never been used are at the end of the array, and a pointer, appropriately called pristine, indicates the first unused record in the array. The procedure to obtain a record is a little more complicated than before: if the free list is not empty the record is removed from the free list, but if the free list is empty, and pristine points to a record in the array, that record is used, and pristine is incremented. If the free list is empty, and pristine points to a location beyond the last element of the array, then it is necessary to allocate a new array and start using its elements. At this point the resetting of the system is very simple: the free list remains empty, and pristine is set to the first element in the array.

**Conclusions**

Two efficient techniques of memory management to implement arrays of unbounded dimensions were studied. The efficiency results from the creation of arrays with increments in size larger than the previous increments used. In the first method the contents of the old array is copied into the new larger array, but it is shown that this process is not inefficient. In the second method a linked list of arrays is kept, and new arrays are appended to the list, thus avoiding the need to copy elements. The first method is very useful for the implementation of heaps, and in this case the overhead of copying elements is negligible. The second method is useful for the implementation of stacks or queues, and except for the overhead of memory management these implementations execute as fast as the conventional ones. Finally, the technique of a linked list of arrays was used to develop a very efficient memory management method for records of fixed size.
References


Alejandro Barrero (abarrero@pella.com) is principal software engineer at Pella Corp., Pella, Iowa. He submitted this article for consideration in 1990.