An Empirical Evaluation of Extendible Arrays

Stelios Joannou and Rajeev Raman

University of Leicester, Department of Computer Science, University of Leicester, University Road, Leicester, LE1 7RH.

Abstract. We study the performance of several alternatives for implementing *extendible* arrays, which allow random access to elements stored in them, whilst allowing the arrays to be grown and shrunk. The study not only looks at the basic operations of grow/shrink and accessing data, but also the effects of memory fragmentation on performance.

1 Introduction

Dynamic (internal-memory) data structures are ubiquitous in computing, and are often used in on-line, continuously running, software that responds to external events (such as "daemons"). Many classical data structures (heaps, dynamic trees etc), are developed in the *pointer machine* model [17]; this paper is not primarily concerned with these, but with the rapidly increasing number of RAM dynamic data structures (e.g [1]) that have been recently proposed, particularly *succinct data structures* [12, 4, 11, 5, 13]. An important feature of these data structures is that they repeatedly allocate and deallocate variable-sized pieces of memory. The memory usage can be measured in two ways:

- In the *memory manager* model the algorithm calls built-in "system" procedures *allocate* and *free*. The procedure *allocate*(x) returns a pointer to the start of a sequence of *contiguous* (unused) memory locations, and increases the memory usage of the algorithm by x units. The procedure *free*(p) takes as an argument a pointer p to a contiguous block of memory locations that was previously *allocated*, and frees the entire block; the memory usage of the algorithm reduces by the requisite number of units.
- In the classical RAM memory model, the algorithm has access to memory words numbered $0, 1, 2 \ldots$. The space usage at any given time is simply $s+1$ where s is the highest-numbered word currently in use by the algorithm [6].

Although many dynamic succinct data structures [4, 9, 2] work in the memory manager model, this model does not charge the data structure for space wastage due to *external fragmentation* [16, Ch 9]. It is known that if N is the maximum total size of blocks in use simultaneously, any memory manager needs $\Omega(N \lg N)$ words of memory to serve the requests in the worst case [14, 15, 10]. (If data is always allocated in fixed-size chunks, there is no serious issue with fragmentation; we also do not consider situations where the memory manager can move an already allocated chunk of memory to a different address.)

Fragmentation is problematic for a number of reasons. If the memory allocator is directly allocating physical memory, then fragmentation results in significant underuse of physical memory. Of course, most computing devices run operating systems that provide per-process virtual memory, but this is not universal: operating systems such as Android do not support virtual memory, and this appears to be relatively widespread when the secondary storage is based on solid-state technology, due to the current tendency for upgrades to degrade solid-state memory[1]. Even when virtual memory is supported, it is not axiomatic that virtual memory is unlimited — a notable example is the Java VM, which is limited to 2GB of virtual memory. Finally, when virtual memory is effectively unlimited (as it would be on a 64-bit machine), when the data being stored is close to the physical internal memory on a machine, fragmentation may lead to "thrashing", and on smaller input sizes, poor usage of TLB.

Unfortunately, the memory-manager model is the *only* memory allocation method available for normal application programmers, and it is inconvenient (as in [11, 5, 13]) to simulate the RAM memory model through the memory-manager model (in practice such simulation is impossible if the data structure is to be used as part of a large, complex application). Our aim, therefore, is to find *fragmentation-friendly* dynamic data structures, which (ideally) achieve fragmentation-friendliness through *self-tuning*, and not by means of parameters that the user sets, since these parameters may be highly data-dependent (e.g. they may depend upon the relative amount of textual and markup data in an XML document, the distribution of keys to be hashed in a dictionary etc.).

Collections of extendible arrays. We focus on the above issues in a *collection of extendible arrays* (CEA), which is arguably the simplest dynamic random access data structure, but can be used to build complex data structures [11, 13]. A CEA maintains a collection of *extendible arrays* (EAs); each EA in the collection is a sequence of n records. Each record is stored in a single word and is assigned a unique index between 0 and $n-1$. The operations supported are:

- $\mathsf{create}(r)$: create an new empty EA and return its name,
- $\mathsf{destroy}(A)$: destroy the EA A, and
- $\mathsf{access}(i, A)$: access (read/write) the record with index i in the EA A,
- $\mathsf{grow}(A)$: if the EA A currently has n records, add a new record to the end of A with index n.
- $\mathsf{shrink}(A)$: if the EA A currently has n records, delete the record in A with index $n-1$ (the last record).

Although there have been several studies of memory fragmentation in general [3, 8, 7], we believe this the first study where the effect on fragmentation of a series of allocations/deallocations by a *specific* data structure is studied.

[1] This is especially problematic when the amount of secondary memory is limited, as memory locations will be written to repeatedly by the virtual memory system.

2 Data Structures

We now describe our data structures. A CEA is represented by a vector (as described below) which contains pointers to the individual EAs; the handle of an EA is simply the index in this vector that contains the pointer to the EA. We consider the following implementations of an individual EA:

Vector. This is the standard data structure, which stores an EA with n records in an array of size at most $2^{1+\lceil \log_2 n \rceil}$ records. To handle an intermixed sequence of grow and shrink operations, a rule for resizing the array might be as follows: double the array size whenever there is no more room to accommodate a grow and halve the array size whenever a shrink causes it to become less than 1/4 full.

Remarks. The time for access is worst-case $O(1)$, grow and shrink take $O(1)$ amortized time each and and create/destroy take $O(1)$ time each. However, a vector of size n may have internal fragmentation of $\Theta(n)$ words[2]. Furthermore, assuming a first-fit allocator, it is easy to come up with a sequence of operations that yields n vectors of total size $O(n)$ records that occupy a range of memory addresses spanning $\Theta(n \log n)$ words (details omitted).

Simple. To reduce the internal fragmentation, the simplest idea is to choose a fixed integer parameter $b > 1$ (ideally a power of 2). Records are stored in fixed-size *data blocks* of b words each. For each EA with size n, we store a vector (called the *index block*) that contains $\lceil n/b \rceil$ pointers to each data block; to perform access(i), we access the $(i \bmod b)$-th entry in the $\lfloor i/b \rfloor$-th data block.

Remarks. This gives $O(1)$ worst-case time for access and $O(1)$ amortized time for grow and shrink, and $O(1)$ time to create and destroy empty EAs. The use of equal-sized data blocks means that a CEA built upon this EA is less susceptible to external fragmentation. The index block occupies $O(n/b)$ words, this overhead can be minimized by choosing a large value of b. However, if the collection contains a significant proportion of small (size $\ll b$) EAs, there could $\Theta(b)$ words of internal fragmentation per EA, and the internal fragmentation could be even more than for the vector CEA. Thus, the parameter b must be chosen based upon knowledge of the way the DS will be used (which may not be available), and this DS is not "self-tuning". Furthermore, from an asymptotic viewpoint, the fact that index blocks are $\Theta(n)$ in size may mean that external fragmentation caused by index blocks is relevant.

Brodnik. In [4] a vector of size n is divided into consecutive (conceptual) *superblocks* of size $1, 2, 4, \ldots, 2^{\lfloor \log_2 n \rfloor}$. A superblock of size 2^k is represented as up to $2^{\lceil k/2 \rceil}$ data blocks of size $2^{\lfloor k/2 \rfloor}$ each, and memory is only allocated for

[2] By *internal fragmentation* we mean memory allocated by a data structure but not used, analogous to the operating systems term [16])

non-empty data blocks. An index block contains pointers to all data blocks and is represented as a vector. The access(i) function is a little complex:

access(i):
1. Let r denote the binary representation of $i+1$, with all leading zeros removed.
2. The desired element i is element e of data block b of superblock k, where:
   (a) $k = \lfloor \log_2(i+1) \rfloor$,
   (b) b is the $\lfloor k/2 \rfloor$ bits of r immediately after the leading 1-bit, and
   (c) e is the last $\lfloor k/2 \rfloor$ bits of r.
3. Let[3] $p = 2^{\lfloor k/2 \rfloor} + 2^{\lceil k/2 \rceil} - 2$.
4. Return the e-th element of the $(p+b)$-th datablock.

Remarks. Brodnik et al. [4] show how to implement access in $O(1)$ worst-case time. The amortized time for grow and shrink is clearly $O(1)$, and $O(1)$ time is needed to create and destroy empty EAs. It is easy to see that "wasted" space (internal fragmentation plus the index block) is $O(\sqrt{n})$ words. Brodnik et al. [4] show that this level of wasted memory is optimal. However, it is possible to give a sequence of grow and shrink operations that creates $O(n)$ vectors of total size $O(n)$, but occupying $\Theta(n \log\log n)$ words of memory (details omitted).

Modified Brodnik A modification of Brodnik et al.'s data structure is as follows. All data blocks in a given EA are of the same size b (which is a power of 2), initially $b = 2$. There is initially an index block of size i (also a power of 2), initially $i = 1$. A grow or shrink adds/deletes elements from the last data block, allocating a new data block or freeing a newly-empty one, as needed. Consider now a sequence comprising solely of grow operations. If the index block is full, we alternate between two courses of action: doubling i and doubling b; in the latter case we take pairs of existing data blocks, and copy their data into a newly allocated data block of size $2b$, and free the existing data blocks (this has the effect of making the index block half-full). For a mixture of intermixed grow and shrink operations, if the index block occupancy drops below $1/4$ after a shrink we undo the last "adjustment" operation (i.e. we halve b or i, whichever variable was doubled most recently). The access operation works as in Simple.

Remarks. This gives $O(1)$ worst-case time for access and $O(1)$ amortized time for grow and shrink, and $O(1)$ time to create and destroy empty EAs. However, the CPU cost of the access instruction is significantly lower. Again, the wasted space is $O(\sqrt{n})$ words and, as with Brodnik, it is possible to give a sequence of grow and shrink operations that creates $O(n)$ vectors of total size $O(n)$, but occupying $\Theta(n \log\log n)$ words of memory.

Global Brodnik. Both Brodnik-style data structures above potentially suffer from external fragmentation when used in a CEA. This is because different EAs

[3] The formula $p = 2^k - 1$ in [4] is (clearly) wrong: there are $O(\sqrt{n})$ data blocks.

in the CEA will have different data block sizes (we ignore external fragmentation due to index blocks since the index blocks are typically a small overall component), so a mixture of block sizes will typically be in the process of allocation/deallocation. We now use some ideas from [13] to "self-tune" block sizes. If t is the number of EAs currently created, N is their total size, and b the current block size, then the worst-case internal fragmentation is $O(bt)$, and that due to the index blocks is $O(t + N/b)$. Balancing the two gives the optimal block size as $b = \Theta(\sqrt{N/t})$. The algorithm tries to maintain an ideal block size of $c\sqrt{N/t}$ for some constant $c > 0$, and whenever the real block size is more than a factor of two away from this "ideal" value, it is either doubled or halved, resulting in a re-organization of all EAs in the CEA.

Remark. The time for access is clearly $O(1)$, and in [13] it is shown that the amortized time for grow, shrink and create is $O(1)$; however, this analysis assumes that the number of EAs in existence at any given time is within a constant factor of the maximum number of EAs that were ever in existence in the past. The internal fragmentation is clearly $O(\sqrt{Nt})$ words; representing each EA individually using Brodnik would lead to internal fragmentation of $O(\sum_{i=1}^{t} \sqrt{n_i})$ words, where n_i is the size of the i-th EA, which is better than $O(\sqrt{Nt})$ unless all EAs have roughly the same size.

3 Experimental evaluation

The aforementioned data structures have been implemented in C++ and a variety of tests were conducted to study the speed as well as the memory usage/fragmentation of the implementations together with the C++ STL vector, which we now describe. The test machine that was used to run these tests was a Intel Core 2 Duo 64-bit machine with 4GB of main memory, 3.16GHz CPU and 6MB L2 cache, running Ubuntu 10.04.1 LTS Linux. The compiler version was g++ 4.4.3 with optimization level 3. The CEAs all stored 4-byte integer records; note that pointers are 8 bytes each. To measure the memory, both virtual memory (VM) and resident memory (RES) that was used by the tests, `/proc/file/stat` was used. For the speed tests `clock()` method was used to measure CPU time and the `/usr/bin/time` command for wall time.

3.1 Implementation Details

For all of these DS, except Global Brodnik, a common collection manager class was used, to allow multiple instantiations of EAs, choosing the DS using compiler options. The collection manager uses an array of pointers to store the memory locations of each instance of the DS. Every time the array is full it doubles its size. The EAs are allocated in memory using the `new` keyword.

Vector. We used the standard STL implementation, which uses doubling if the underlying array is full, but when elements are removed, the underlying array size does not change in any way.

Brodnik. This implementation of this DS is based on the original paper. To optimize the speed of access (and also shrink and grow), a number of values are stored in the header block of this DS, giving a relatively large header size of 41 bytes. Further, to optimize access(i), some operations (e.g. $\lfloor x/2 \rfloor$, $\lceil x/2 \rceil$) were written using bitwise operations. To compute $\lfloor \log_2 x \rfloor$ (the left-most set bit in an integer x) the folklore trick of casting x to a float is used. We access this memory as an integer, use bitwise operations to extract the exponent, then subtract the bias, and the result is the position of the left-most set bit. Finally, a table of size at most 64 integers was used to map the number of the superblock that the i-th record is located in, to the number of data blocks prior to the specified superblock. These optimizations greatly increased the speed of access(i).

Simple. This DS is implemented with the data block size (which must be a power of 2) being a constructor parameter. In the access function, operations such as division by b and modulo b are implemented by shifts and masks, respectively. We used $b = 2^6 = 64$ throughout in our tests. Again a number of header variables are used and the header block size is 29 bytes.

Modified Brodnik. The access(i) operation is similar to Simple, it uses masking and shifting to get the location of element in a data block and the location of that data block in a super block. Since the size of these data blocks changes over time as elements are added or removed, a static array of masks was used. Since growing the index block and data block alternates, a boolean was used to check what was doubled last (the index block size or the data block size). This DS has a header size of 30 bytes.

When the data blocks need to double, every two data blocks are merged into a new one with double the size. This new data block is stored in the already existing index block starting from the beginning (thus avoiding the creation of a new index block). Similarly where access is worst-case $O(1)$, grow and shrink would take $O(1)$ amortized time each and and create/destroy applied to a new EA/empty EA would take $O(1)$ time each. However, a vector of size n may have internal fragmentation of $\Theta(n)$ words. Furthermore, assuming a first-fit allocator, it is easy to come up with a sequence of creates, grows and shrinks that yields n vectors of total size $O(n)$ records, occupying a range of memory addresses spanning $\Theta(n \log n)$ words in total. When shrinking, either the size of the index block or the data block will be halved. When data blocks need to shrink, one data block is split into two and this is done by storing the new bigger data block into a new index block of the same size as the original one. The old data blocks and index blocks are subsequently deleted.

Global Brodnik. Each individual EA has a header size of 25 bytes. The collection maintains the total number of elements t in all the instances of the EAs that it contains. We derive from the current data block size b (a power of 2) an upper bound $U = 2b$ and lower bound $L = b/2$. After each grow/shrink we

use calculate an ideal block size $\hat{b} = \sqrt{t/N}$, where N is the number of EAs. We maintain the condition that $L < \hat{b} < U$: if this condition is violated then the data block size is doubled/halved, along with U and L, to restore this condition. We avoid doing a square root calculation every time there is a grow or shrink by checking if $t/N \geq U^2$ for the upper bound and similarly for shrink. The values U^2, L^2 are recomputed every time U and L change.

The access method is the same as modified Brodnik and an array of masks is used. The index block of an individual EA is doubled when it gets full, either to accommodate one new data block, or because the size of the data blocks is halved. An index block is halved when its occupancy drops below a quarter of its capacity (by a shrink on an individual EA).

3.2 Speed Tests

We tested the time for access(i) (specifically a read — writes were not tested). In all cases EAs were created *sequentially*, i.e., the i-th EA was created and grown to its final size before creating and growing the $(i+1)$-st EA. We considered two access patterns: *sequential* and *random*. For the sequential access test elements were accessed in the order in which they were grown. In the random access test, we instead made uniform random accesses equal to the number of elements in the CEA. The random test was essentially run only in the case where all EAs in the CEA are equally sized, and the number of EAs and elements per EA are both powers of two. In this case we used one call to the `lrand48()` method in the C++ `cstdlib`. This generates a number in the range of $[0, 2^{31})$: we use the most-significant bits to select an EA and the least-significant bits to select an element within that EA. This avoids making two calls to `lrand48()` (which is relatively slow), but limits the total number of elements t that can be used for this test to 2^{31}. This was not a limitation as data sizes such as these would have exceeded the RAM of our machine.

A variety of values of N (the number of EAs) and k (the size of each EA) was used. The values used were $N = 16$ and $k = 16777216$ (a few large EAs), $N = k = 16384$ and $N = 2097152, k = 128$ (many small EAs, relevant to some succinct dynamic data structures). Each test was run five times and the average of these times is included in this paper. Table 1 gives the results.

As can be seen, all the data structures are significantly faster than Brodnik for sequential access. This is very much as expected (and Brodnik is not particularly "slow" in absolute terms). Also vector is slightly faster than other EAS in all the tests. The random tests show more interesting structure. In the first test, most data structures are similar except for Simple, which is a bit slower. All the data structures used for these tests except for the vector require two memory accesses to retrieve the required element due to indirection cause by the index blocks. This is the main reason why in general the vector is faster than the other DS that were tested. However, in the first test the size of the index blocks in all but Simple are very small (they grow as $\sqrt{n}$, where n is the size of an individual EA) and so fit in cache. However, this pattern is not repeated in the other tests, since the overall size of the index blocks (as a proportion of data blocks) increases

EAs x Elements	DS	Grow	Sequential	Random
16 x 16777216	Vector	2.38	0.25	22.65
	Brodnik	2.93	1.90	28.66
	Simple	1.87	0.31	40.53
	Modified Brodnik	1.69	0.29	20.63
	Global Brodnik	4.95	0.33	23.96
16384 x 16384	Vector	1.90	0.25	24.03
	Brodnik	3.12	1.87	57.46
	Simple	1.85	0.32	44.35
	Modified Brodnik	2.39	0.30	48.05
	Global Brodnik	4.93	0.34	44.46
2097152 x 128	Vector	3.12	0.29	44.69
	Brodnik	6.31	2.09	86.45
	Simple	2.11	0.43	56.28
	Modified Brodnik	6.21	0.58	54.04
	Global Brodnik	6.26	0.48	58.26

Table 1. Growing time, Sequential and Random access time test results (in seconds)

as n decreases. There is a slight advantage to the Global and Modified Brodnik in terms of access times, we believe that this may be because the regular rearrangement of data in Global and Modified results in more compact storage; but this requires further investigation.

3.3 Memory Usage Results

Worst Case for Brodnik DS. We now discuss a potential worst-case scenario for the Brodnik DS. The scenario is constructed assuming that there is some "first-fit-like" behavior in the memory manager and will be tested experimentally against the real-life Linux allocator.

The Brodnik DS, as mentioned before, has a header block, an index block and data blocks grouped into virtual superblocks. The test proceeds in rounds $0, 1, 2 \ldots$. In round i, N_i EAs of size k_i are created sequentially (see beginning of 3.2); in rounds $i > 0$ this creation is accompanied by shrinking (in a round-robin manner) the EAs created in round $i - 1$. We choose j_0 to be an even integer and let $k_0 = 2^{j_0+1} - 1$; in subsequent iterations we take $j_{i+1} = 2j_{i+1} + 4$ and $k_{i+1} = 2^{j_{i+1}+1} - 1$. We always maintain $N_0 k_0 = N_1 k_1 = N_2 k_2$ and so on, so that the total number of elements in the CEA stays the same.

The reason why this pattern may result in fragmentation is as follows. We hope that if we sequentially allocate N_i EA of size k_i, the space between the header blocks of EAs will be approximately equal to k_i. Note that for EAs of size $2^{x+1} - 1$, the last superblock is of size 2^x, and the size of the data blocks in the last superblock is $2^{\lceil \frac{x}{2} \rceil}$. Thus, in the next round, the data blocks in the last two superblocks of the newly created EAs will be of size $2^{j_i+2} > 2k_i$. Thus, we hope that all these data blocks (which total 3/4 of the EAs created in the $i + 1$-st phase) will be allocated in "fresh" memory.

In the test, we chose $j_0 = 4$, giving $k_0 = 31, k_1 = 8191$ and $k_2 = 2^{29} - 1 = 536870911$. Assuming that $N_2 = 1$, this would imply that $N_0 = k_2/k_0$, but our machine was unable to allocate so many EAs. Hence we chose $N_0 = 2^{22}$, $N_1 = N_0 k_0/k_1$, $N_2 = 1$, and $k_2 = N_0 k_0 \approx 2^{27}$. The results are shown in Table 2.

i	N_i	k_i	**VM (GB)**	**RES (GB)**
0	4194304	31	2.46	2.45
1	15873	8191	3.03	2.99
2	1	130023424	3.53	3.08

Table 2. The results of the Brodnik DS worst case

Although the real size occupied with the data blocks should be close to 507MB, due to the headers of the data structure it reaches the 2.46GB initially as shown above. In subsequent phases, there is an increase of almost 570MB, showing that in each case most of the memory allocated for the data blocks is coming from "new" memory, not memory previously freed, even though the very last EA is not quite as large as needed by the formula.

Random. For this test we start with N EAs sequentially created, each of size k. Then we go through the CEA and shrink each EA once. We call this a pass. After each shrink we grow one EA. The EA to be grown is selected based on the following rule: the first 20% of the EAs should contain 80% of the elements. This rule is applied recursively so 20% of the first 20% of EAs should contain 80% of the total number of elements of the first 20% of the EAs. We go through all the EAs k times, so the EAs at the beginning of the CEA should be larger and the EAs which have not been grown will have 0 elements.

To run this test the values $N = 2^{16}$ and $k = 1024$ were used. The gradual increase after every pass is shown in Figure 1. Table 3 shows the initial memory usage after creation and growing of the N EAs of size k and the resulting memory usage after this test was run. The important thing to notice in this test is that there has been a significant memory increase in all of the data structures without adding new elements, just by redistributing the elements within the EAs.

DS	**Initial**		**Ending**	
	VM (KB)	**RES (KB)**	**VM (KB)**	**RES (KB)**
Vector	277648	266980	692040	603772
Brodnik	388228	377576	628456	523156
Simple	304276	293624	357080	343476
Modified Brodnik	328828	318264	577224	485612
Global Brodnik	328768	318208	372900	357440

Table 3. Memory usage before and after 80-20 test

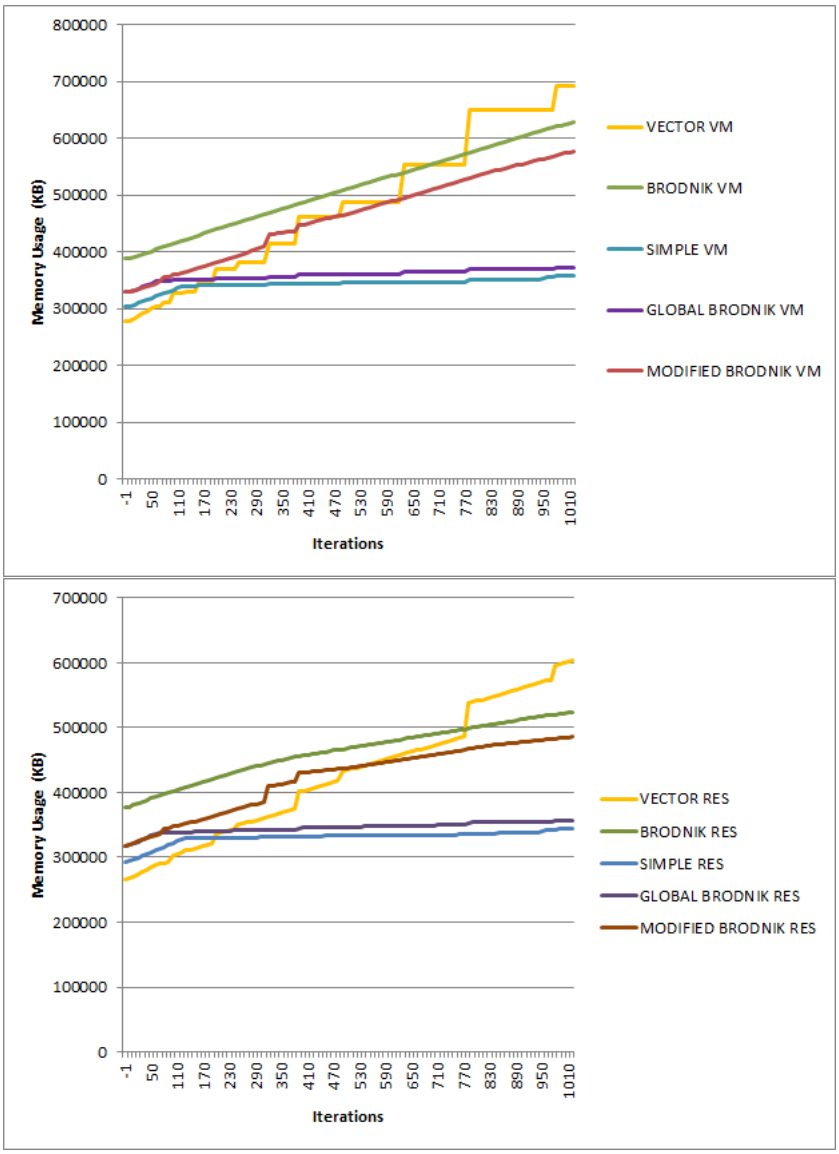

Fig. 1. Virtual Memory (Top) and Resident Memory (Bottom) results of the 80-20 test

Thrashing. For the thrashing test we created $N = 2^{19}$ EAs sequentially, each of size $k = 1200$, equating to about 2.4GB of useful data (recall that our machine has 4GB RAM). We performed random access tests immediately after creation and *after* growing and shrinking arrays as in the the 80-20 test described in Section 3.3[4]. We measured the following: (a) CPU time for the first random access test (b) VM/RES before and after the 80-20 test (c) CPU/elapsed time for the second random access test. The results for (b) and (c) are shown in in Table 4 – we do not report (a) because they are in line with Table 1, except that Brodnik was slower by a factor of 2 than expected (the initial VM was close to the physical memory of the test machine). For (c), Brodnik and Modified Brodnik fell foul of thrashing and took over 14 minutes to complete(thrashing was verified by inspecting CPU usage and page faults using `top`). Vector, despite a high VM, completed, albeit slowly, because it allocates contiguous chunks of memory. Simple and Global Brodnik performed the best in this case.

DS	Initial		Final		CPU	Elapsed
	VM	RES	VM	RES		
Vector	4.23	3.73	7.34	3.74	40.12	780
Brodnik	3.66	3.65	6.06	3.73	40.19	872
Simple	2.83	2.82	3.20	3.17	28.2	150
Modified Brodnik	3.15	3.14	5.71	3.67	43.28	1988.4
Global Brodnik	3.15	3.14	3.51	3.47	25.52	134.4

Table 4. Memory usage (GB) before and after 80-20 EA modification in thrashing test; CPU and elapsed time for second random access test (s).

4 Conclusions

In this paper we have investigated a simple random-access dynamic data structure, the collection of extendible arrays. The standard solution would be to use a number of vectors, but this solution runs into memory fragmentation problems. We have demonstrated a sharp rise in virtual memory usage for the standard solution. We have also conducted some tests that demonstrate that for appropriate data sets that require memory close to the physical memory of the machine, after running the 80-20 test described in section 3.3 the memory requirements were greater than the physical memory of the machine, thus thrashing occured. Unfortunately, the same is true for the data structure proposed by Brodnik et al., which is aimed at solving this problem. We observe that the simple solution of using indirection, together with the so-called "Global Brodnik" seem to avoid this problem, but the simple solution requires parameter setting (which in turn

[4] With a minor modification: we never let a shrink reduce the size of an EA below 200.

requires knowledge of how the data structure is used) which would appear to preclude it as a general-purpose solution. However, "Global Brodnik" is relatively slow when supporting the grow and shrink operations, which should be investigated further. Another important task would be to compare their performance on real-life inputs.

References

1. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. J. ACM 54(3), 13 (2007)
2. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: FOCS. pp. 787–796. IEEE Computer Society (2010)
3. Brodal, G.S., Demaine, E.D., Munro, J.I.: Fast allocation and deallocation with an improved buddy system. Acta Inf. 41(4-5), 273–291 (2005)
4. Brodnik, A., Carlsson, S., Demaine, E.D., Munro, J.I., Sedgewick, R.: Resizable arrays in optimal time and space. In: WADS. LNCS, vol. 1663, pp. 37–48. Springer (1999)
5. Farzan, A., Munro, J.I.: Dynamic succinct ordered trees. In: ICALP (1). LNCS, vol. 5555, pp. 439–450. Springer (2009)
6. Hagerup, T., Raman, R.: An efficient quasidictionary. In: SWAT. LNCS, vol. 2368, pp. 1–18. Springer (2002)
7. Jr., P.W.P., Stigler, S.M.: Statistical properties of the buddy system. J. ACM 17(4), 683–697 (1970)
8. Knuth, D.E.: The Art of Computer Programming, Volume I: Fundamental Algorithms. Addison-Wesley (1968)
9. Lee, S., Park, K.: Dynamic rank/select structures with applications to run-length encoded texts. Theor. Comput. Sci. 410(43), 4402–4413 (2009)
10. Luby, M., Naor, J., Orda, A.: Tight bounds for dynamic storage allocation. In: SODA. pp. 724–732 (1994)
11. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: SODA. pp. 529–536 (2001)
12. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: WADS. LNCS, vol. 2125, pp. 426–437. Springer (2001)
13. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: ICALP. LNCS, vol. 2719, pp. 357–368. Springer (2003)
14. Robson, J.M.: An estimate of the store size necessary for dynamic storage allocation. J. ACM 18(2), 416–423 (1971)
15. Robson, J.M.: Bounds for some functions concerning dynamic storage allocation. J. ACM 21(3), 491–499 (1974)
16. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 7e. John Wiley & Sons, Inc. (2004)
17. Tarjan, R.E.: Data Structures and Network Algorithms. SIAM (1987)