Resizable Arrays in Optimal Time and Space, Revisited

Jakub Łopuszański

Department of Computer Science and Mathematics, University of Wroclaw

Abstract. In this paper we analyze a resizable array—a fundamental and ubiquitous data structure with \texttt{GET}(i), \texttt{SET}(i,e), \texttt{PUSHBACK}(e), and \texttt{POPBACk}() operations.

We focus on the analysis of space required to store the data structure in memory between the queries, giving more freedom during the queries themselves. We define a new, realistic model in which data structure is allowed to use temporary buffers during operation, which must be freed before finishing it.

In this model we prove a trade off: any algorithm which performs \( n \) \texttt{PUSHBACK} operations wasting no more than \( O(f(k)\sqrt{n}) \) cells of memory for some natural \( k \) and function \( f \), must take at least \( \Omega(kn) \) to complete.

We match this lower bound, by providing for each \( k \in \mathbb{N} \) an algorithm which requires only \( O(k\sqrt{n}) \) additional memory and performs each operation in \( O(k) \) amortized time, where constants hidden by the \( O \) notation do not depend on \( k \) nor \( n \).

This result surpasses the lower bound of \( \sqrt{n} \) for additional memory needed to perform \( n \) \texttt{PUSHBACK} operations given in [2], where the proof relied on a strong requirement that the space consumption is measured not only between operations, but also while performing a query—in particular during reallocation.

1 Introduction

A resizeable array, also known as the vector from the STL, or the ArrayList in .NET and the Java language, is a ubiquitous abstract data structure used as a building block for more complex components in many applications. It provides operations such as \texttt{GETSIZE}(), \texttt{GET}(i), \texttt{SET}(i,e), \texttt{PUSHBACK}(e), and \texttt{POPBACk}() all of which programmers expect to work in an amortized \( O(1) \) time.

A realistic assumption is that an algorithm allocates memory in so called buffers of size specified by the algorithm. A buffer of size \( x \) contains \( x \) cells, each capable of storing one element or one pointer, and algorithm may access any cell of the buffer by indirect addressing in a constant time.

The most common implementation of a resizeable array simply allocates a twice larger buffer whenever the currently allocated buffer is too small, see [3, 1] for explanation. Implementations vary in ways in which they handle a case of too large buffer, in particular the STL's vector [4] never shrinks. One of the conceivable ways is to replace the buffer with twice smaller one when the buffer is
four times too big. This, in worst case, may lead to a situation where \( n \) elements are stored in a buffer of size roughly \( 4n \), or in other words wasting \( 3n \) cells of memory. This is one of the introductory examples of a data structure, on which students learn about amortized time analysis and the use of \( O \) notation for memory consumption.

There are areas where amortized constant time, or asymptotic linear space consumption is not good enough, though. In particular succinct data structures are a field of computer science in which the aim is to store information in a memory of size comparable to the information theoretic lower bounds, allowing only a sublinear amount of wasted space. As resizable arrays are a building block for many other data structures, space wasted or gained at this low level of abstraction can have a great impact on the size of a whole data structure.

In their paper [2] Brodnik, Carlsson, Demaine, Munro and Sedgewick shown that any data structure capable of performing insert and read\(^1\) operations must at some point waste at least \( \sqrt{n} \) cells of memory during a sequence of \( n \) inserts, either for pointers or for unneeded buffer space. In particular, this result applies to every resizable array. The title of their paper, “Resizable Arrays in Optimal Time and Space” might wrongly suggest that their result was restricted only to data structures trying hard to reach the amortized \( O(1) \) time for inserts, or that there is some trade-off between speed and size – but, none of this is the case.

Their proof starts from an observation that to hold \( n \) elements one either has more than \( \sqrt{n} \) pointers to buffers, or at least one buffer of size \( \sqrt{n} \). But, in the later case, immediately after the allocation of this large buffer, a data structure was wasting whole of it, as it was an uninitialized memory at that time.

This model and consequently the proof might seem a little bit too conservative, as it is common for data structures to use a temporary buffer during computations, which is then discarded before the control is returned to the caller. Even if such a buffer takes \( \Theta(n) \) words of memory, it might be bearable, as long as this is not a significant portion of the whole available system memory and there are not too many concurrently existing temporary buffers. It is reasonable to assume that both conditions hold in an environment with bounded number of threads concurrently executing a query and many small or mid-size resizable arrays. As resizable arrays are a fundamental building block of many other data structures it is easy to believe that there are many of them in the memory, and only a few of them are concurrently being reallocated.

This calls for a different model in which the algorithm pays only for memory consumed to store the data structure between queries and not for temporary buffers (i.e. the call stack, or a helper buffer during reallocation). In such a model one can not prove a lower bound using techniques from [2]. In particular if we not restrict the time algorithm has to complete a \texttt{PUSHBACK} operation, then it can simply perform reallocation after each operation, wasting no memory at all, except for the single pointer to the only buffer.

\(^1\) Actually they used delete instead of read, but they simply needed an operation that requires the data structure not to forget any single element
Intuitively, the potential lays in the omnipresent trade-off between speed and size. Adding the requirement that insert operation should complete in $O(1)$ amortized time seems reasonable, expected by developers, and as we shall see – fruitful.

Since inner workings of a hypothetical data structure, such as movement of elements between buffers, or computing their indexes, are quite difficult to analyze, to establish the lower bound we will concentrate on a cost that we are certain each data structure has to pay – the cost of allocating and filling a buffer. This way our time analysis will be strongly coupled with the analysis of memory consumption, as we will infer time costs from the sole fact that the algorithm has allocated a buffer of a particular size. This coupling will provide a trade-off, as the more often algorithm reallocates buffers the longer it takes to execute, and doing it less often will lead to wasting memory. Moreover the analysis becomes more abstract as it does not have to deal with low level world of RAM machine, but instead concentrates on pure mathematical objects such as a multiset of lengths of buffers the analyzed algorithm has allocated.

Of course this technique only makes sense in the proof of the lower bound. For a sincere analysis of the upper bound, we have to count the time used not only for memory allocation, but also for every other detail, such as the time required to locate the $i$-th element among many buffers, moving the elements between buffers, etc.

1.1 Outline

The rest of this paper is organized as follows. Section 2 describes a model for dynamic memory allocation and time measure used through the paper. Section 3 contains a lower bound for any data structure which can handle insert and read operations. Section 4 presents an implementation of resizable array in an incremental manner – starting from a data structure suited precisely to deal with the input used in the lower bound, we subsequently add support for other operations and more complicated inputs, finally presenting a full, tested implementation of a resizable array.

2 Model

We aim to model data structures and algorithms which hold elements. The basis of the model is a RAM machine with cells capable of storing either a single element or a pointer. Apart from a constant number of cells which are always available (think: registers and some statically pre-allocated memory), an algorithm can only use cells in explicitly allocated buffers. That is an algorithm may request allocation of a buffer of arbitrary chosen size, which always succeeds and returns a pointer to the newly created buffer. We allow and use indirect addressing – given a pointer to a buffer and an integer $x$, an algorithm can access the $x$-th cell of the buffer in $O(1)$.
Even though, there exist allocators that take sublinear time to allocate \( x \) cells, we will assume it takes \( x \) units of time. In fact our lower bound analysis will not measure time used for any operations other than allocation. The rationale behind this is that for the purpose of the lower bound we will use an input sequence known in advance, so an algorithm will never have to overallocate space which will never be written to. So, one can think about this cost as being charged to those writes which will happen in the future. This makes the analysis simpler as we focus only on allocations, instead of tracking each write, read, deallocation, etc. Readers uncomfortable with such a prepaid setting, are invited to simply assume that an allocator requires linear time to zero out memory for security reasons.

Consequently, a buffer can be deallocated for free, as the cost can be charged to the linear cost of allocating it in the first place.

We measure the memory consumption in cells. As it is difficult to accurately analyze an arbitrary data structure, we perform the following simplification. We require the algorithm to store elements only in buffers and assume that the memory consumption is at least equal to the total length of all buffers holding at least one element plus the number of these buffers. We call the difference between this number and the number of elements the \textit{wasted memory}.

This neglects the fact that an algorithm might (and often does) store elements in the pre-allocated part of the memory (think: a single element can be stored directly in the union with pointer), or store pointers to buffers in another buffers (think: linked lists). This two oversimplifications can be justified, though. First, an algorithm which uses the common pre-allocated memory to store elements will not be suitable for a data structure used in multiple copies in the system, as sooner or later the pre-allocated memory will drain. Second, an algorithm can be charged for each pointer to a buffer at most twice in our model: once for the pointer itself, and potentially again for storing it in a cell of another buffer. This may increase the estimated amount of wasted memory at most by a factor of 2. This constant, similarly to the assumption that both pointer and element fit in a single cell, is of no particular importance when looking for data structures with sublinear amount of wasted memory. Skeptical reader is invited to count extra one cell as a metadata the allocator has to append to each buffer, or redo the math in the following chapters dividing relevant numbers by 2.

We use \( M \) to denote the current set of memory buffers allocated by the algorithm. As this can change over time, we use \( M_t \) to denote this set after performing \( t \)-th operation of the input sequence. We put \( M_0 = \emptyset \) to ensure that algorithm has to explicitly allocate all buffers.

\textbf{Definition 1.} Let \( |b| \) be the number of cells in a buffer \( b \in M \).

\textbf{Definition 2.} Let \( a_t \) be the total size of allocated buffers after \( t \)-th operation, that is \( a_t \equiv \sum_{b \in M_t} |b| \).

\textbf{Definition 3.} Let \( \Delta_t \) be the change in the cumulative size of allocated buffers during \( t \)-th operation, that is \( \Delta_t \equiv a_t - a_{t-1} \).
Definition 4. Let \( c_t \) be the cost of allocating new buffers during \( t \)-th operation, that is \( c_t \overset{\text{def}}{=} \sum_{b \in M_t \setminus M_{t-1}} |b| \).

3 Lower Bound

Similarly to [2] we show the lower bound by analyzing performance of an algorithm on a sequence of \( n \) insert operations. This result applies to any data structure which does not forget inserted elements, in particular to a resizable array with \textsc{pushBack}, but also a tree with insert, or singly a linked list with \textsc{prepend}. By forgetting we mean a situation when there is no cell in the memory which holds element \( e \) inserted previously.

Definition 5. Let \( c = \sum_{t=1}^{n} c_t \) be the total cost of all allocations done by the algorithm on this input sequence.

Definition 6. Let \( w_t \overset{\text{def}}{=} a_t + |M_t| - t \) be the amount of memory wasted by the algorithm after \( t \) inserts. Let \( w \overset{\text{def}}{=} \max_{t=1}^{n} w_t \).

Definition 7. Let \( \kappa = c/n \), be the amortized cost of allocations per single operation.

The following theorem captures the trade-off between \( \kappa \) and \( w \).

Theorem 1. For any sequence \( (M_t)_{t=0}^{n} \), for which \( M_0 = \emptyset \) and \( a_t \geq t \), we have \( w \geq \frac{1}{4} \left( 1 + \sqrt{n} \right) \) which is equivalent to \( \kappa \geq \frac{1}{\log_2 4w} - 1 \).

The theorem says some truth about sequences, sets and numbers, – pure mathematical objects – without diving into details such as how this sequences got generated in the first place. We abstracted the algorithm away, and focus on the sequence \( (M_t) \) which determines all other sequences and numbers in the theorem.

The outline of the proof is that we will define a potential function \( \Phi \), which on the one hand will be bounded, and on the other hand will increase with each insert. The only way to decrease potential will be to perform reallocations of blocks, and this will give us a lower bound on the cost of the algorithm.

Definition 8. Let \( \Phi(M) \) be the potential of a set of buffers \( M \) defined as:

\[
\Phi(M) \overset{\text{def}}{=} \sum_{b \in M} |b| \lg \frac{a_n}{|b|}.
\]

There are several intuitions behind this potential function. One is that an algorithm will have to “tidy up” the set of allocated buffers from time to time in order to limit the number of pointers, and entropy is quite a good measure of disorder. Another one is that reasonable algorithms concatenate and replace smaller buffers with larger ones, so that an element travels from smaller buffers
into larger ones, each time participating in the cost of reallocation. A logarithm of a block’s length seems to be a good measure of how large the block already is, and thus $\log \frac{b}{b'}$ tries to capture the future cost of further concatenations in which element will participate.

The proof of Theorem 1 is rather technical and lengthy as it has to work around dividing by, and taking logarithms of, zero. Therefore it can be found in the Appendix A.

**Corollary 1.** If there is a constant $k$, and function $f$, such that for every $n$ the space wasted by the algorithm during $n$ inserts is $O(f(k) \sqrt{n})$ then amortized time of an insert must be at least $\Omega(k)$.

**Proof.** If $w \leq C f(k) \sqrt{n}$ for some constant $C$, then the Theorem 1 implies that $4C f(k) \geq n^{1+\kappa-\frac{1}{k}}$, which holds for all values of $n$ only if $\kappa \geq k - 1$.

## 4 Upper Bound

We will present the data structure in an incremental manner. We start with a simple algorithm crafted precisely for the task of performing $n$ pushBack operations, just to demonstrate how one can get close to the lower bound presented in Section 3. Later, we modify the data structure so that it can handle popBack operations quickly. Finally we show how to dynamically adapt some parameters of the data structure so that it does not have to know $n$ in advance.

### 4.1 A Data Structure for $n$ pushBack Operations

Let us fix some $n, k \in \mathbb{N}$, such that $s = \sqrt[2k]{n}$ is a natural number. We assume a very simple subset of input sequences, namely those which consist of $n$ pushBack operations, some get or set operations, but no popBack at all. The number of buffers of particular size will be determined by representation of the current number of elements $t$ in $s$-ary. That is if $t$ can be represented as $\sum_{i=0}^{k} \alpha_i s^i$ with $\alpha_i < s$, then there will be exactly $\alpha_i$ buffers of size $s^i$ forming so called $i$-th block. We store pointers to all these buffers in a single buffer of size $(s - 1)k$. We index cells of these buffers so that the first element is in the largest buffer and the last element is in the smallest buffer. The pushBack algorithm simply creates a buffer of size 1 and tries to push it to the 0-th block. Whenever a block contains $s$ buffers it has to be merged into a larger buffer and recursively pushed into the next block.

This structure wastes most memory for $t = n - 1$ when the buffer of size $(s - 1)k$ is filled with pointers. At this point $w = w_{n-1} = 1 + 2(s - 1)k < 2k \sqrt{n}$.

From Corollary 1 we know that an algorithm with so small footprint can not perform inserts faster than $\Omega(k)$. In our algorithm the amortized cost of pushBack is $O(k)$, as each element starts its life in 0-th block and then gradually gets merged into next blocks, each time participating in the cost of allocation, moving and deallocation.
4.2 A Data Structure for Known n

We will now allow the input sequence to contain \texttt{POpBACK} operations, but still assume that the maximal number of elements in the vector is known in advance and is equal to \( n \). Moreover we will still compare \( w \) to \( n \), even though the actual number of elements, which we will denote by \( t \), can be much smaller than \( n \). This is still unrealistic, but allows us to concentrate on the issue of amortized time analysis in the presence of repeated and expensive merge and split operations.

The key observation is that there are two situations which are stressful and we would like to avoid them: pushing a buffer into an already full block, and popping from an empty block. First requires merging all buffers of the full block into a larger one, possibly causing a recursion. Second requires us to split a large buffer, possibly after recursive split of even larger ones.

We say that the data structure is in \textit{the state of balance} if each block is far from these two extremes. We redefine a block to consist of at most \( 2s - 1 \) buffers, and say it is in a balance iff it has exactly \( s \) buffers.

Since we allow more than \( s - 1 \) buffers in a block, there are many possible ways in which \( t \) elements can be divided into buffers. This makes accessing \( i \)-th element more tedious operation, as the representation of \( t \) in \( s \)-ary is no longer helpful. We can locate the block containing \( i \)-th element in \( O(k) \) time, and then the buffer and the position within it in \( O(1) \). By keeping updated suffix sums of the sequence \( (\alpha_i s^i) \) we can speed it up to \( O(\lg k) \) using binary search. As most elements are in the largest buffers, the naive linear search might actually outperform binary search in practice, though.

Each time we need to push a buffer into a full block, we compact \( s \) buffers into one larger and push it recursively to the next block. The remaining \( s \) buffers result in the state of balance for the current block.

Each time we need to pop a buffer from an empty block, we first recursively pop a big buffer from the next block, and split it into \( s \) smaller buffers. After this temporary state of balance, we pop a small buffer.

To analyze the amortized cost of operations, we use a potential function of the data structure which depends on the number of buffers in each block. We define it so that it is minimal when blocks are in the state of balance:

\[ \phi(\alpha) \overset{\text{def}}{=} [\alpha_r > s](\alpha_r - s)\lambda_r + \sum_{i=0}^{r-1} |\alpha_i - s| \lambda_i, \]

where \( \alpha_i \) is the number of buffers in the \( i \)-th block, \( r \leq k \) is the largest index of nonempty block, and \( \lambda_i \) is a constant defined to be:

\[ \lambda_i \overset{\text{def}}{=} (k - i)s^i. \]

The sum goes only up to \( r - 1 \) and treats \( r \) individually, to make sure that an empty data structure has potential \( \phi(0) = 0 \).

We can now formally show that the amortized cost of \texttt{PUSHBACK} and \texttt{POPBACK} is \( O(k) \).
Proof that amortized cost of \texttt{PUSHBACK} is $O(k)$ We can divide a \texttt{PUSHBACK} into two phases and analyze them separately:

- \textit{the actual push} to the 0-th block,
- zero or more \textit{merges}, each of which merges $s$ buffers from one block into a larger buffer and pushes it to the next block.

**Fact 2.** The actual push requires $O(1)$ operations, and increases potential at most by $\lambda_0 = k$, so the amortized cost is $O(k)$.

**Fact 3.** A merge of $s$ buffers of size $s^i$ into a single buffer of size $s^{i+1}$ requires $O(s^{i+1})$ operations. If performed at the moment when $i$-th block contains precisely $2s$ buffers, it reduces $|\alpha_i - s|$ by $s$ and increases $|\alpha_{i+1} - s|$ by at most 1, so the potential drops at least by $s\lambda_i - \lambda_{i+1} = s(k - i)s^i - (k - i - 1)s^{i+1} = s^{i+1}$.

An important consequence of Fact 3 is that exact number of merges does not matter as they are for free. For those concerned with the cost of recursion itself, we can pay additional $O(k)$ for handling that.

Proof for \texttt{POPBACK} is analogous, and can be found in Appendix B. It may look odd, that potential drops by $s^{i+1}$ during both: merging and splitting. This is however by design, as we have purposely chosen the size of blocks, so that both operations restore the state of balance.

4.3 The Final Implementation of Resizable Array

In previous subsections we have fixed value of $s$ to be $\sqrt[3]{n}$ where $n$ was the maximal number of elements stored in the data structure. Of course whenever number of elements is much smaller than $n$, such approach wastes too much memory. In this section we use $n$ to denote the current number of elements in the data structure, and algorithm no longer knows what to expect.

So far, our data structure wasted at most $O(sk)$ cells for pointers to buffers, the buffer containing them, and some auxiliary counters. In what follows, we present a way to ensure that an invariant $s \leq 3 \sqrt[3]{n}$ always holds, which means that wasted memory will be $O(k\sqrt[3]{n})$. We will adapt $s$ to the current value of $n$ according to the rules explained below.

**Definition 9.** We say that $s$ is too small if the combined space in all buffers is not enough to hold $n$ elements, that is iff $$ (2s - 1)(1 + s + \ldots + s^k) < n . $$

**Definition 10.** We say that $s$ is too large iff $$ s > \max\{2, 3 \sqrt[3]{n}\} . $$

The threshold in Definition 10 is somewhat arbitrary, but ensures that wasted space is $O(k\sqrt[3]{n})$ unless $s$ is too large.

**Definition 11.** We say that $s$ is appropriate if it is not too small and not too large.
Initially let $s = 2$, which is appropriate for $n = 0$.

After each `pushBack` and each `popBack` we check if $s$ is appropriate. If it is not then we will compute a new value $s' = \lceil \sqrt[2]{n} \rceil$, allocate a new data structure built using $s'$ instead of $s$, move all elements to the new data structure and finally drop the old data structure. Observe that $s'$ is appropriate for $n$, as:

$$(2s' - 1)(1 + s' + \ldots + s'^k) \geq s'^k = \lceil \sqrt[2]{n} \rceil^k > n,$$

and

$$s' = \lceil \sqrt[2]{n} \rceil < \sqrt[2]{n} + 1 < 3\sqrt[2]{n}.$$

It is easy to see that this whole rebuilding requires $O(n)$ low level operations. However, we must also pay for the potential of the new data structure, which depends on the way in which $n$ will be decomposed into blocks.

It can be shown that the ratio between potential and the number of elements is never greater than $O(k)$ no matter how we partition $t$ into blocks. This suggest a simple and practical implementation of resizing in which we use the $s'$-ary representation of $k$ as our new $\alpha$, and fill the new data structure by using $O((s + s')k)$ `memcpy` calls.

Although, to make the presentation of the proof simpler, we implement resizing as $n$ `pushBack` operations on the new data structure. Clearly the resulting potential would be at most the total amortized cost of $n$ `pushBack` operations, which is $O(kn)$. In order to be able to pay for such careless rebuilding we need to somehow accumulate $\Omega(kn)$ credits before $s$ becomes too small or too large. We will keep totally separate account for this purpose – otherwise, we could end up with an infinite loop in our reasoning in which each `pushBack` would have to pay not only for resizing but also for `pushBack` operations performed during it. Actually the analysis is quite simple due to the following lemma:

**Lemma 1.** *Between the moment when $s$ is set to $\lceil \sqrt[2]{n} \rceil$, and the moment when $s$ becomes too small, or too large for $n'$ elements, at least $\Omega(n')$ `popBack` or `pushBack` operations must be performed.*

*Proof.* We will now consider the two cases separately:

1. Assume the number of elements has shrunk from $n$ to $n'$, and $s$ became too large. It must be that $s > \max\{2, 3\sqrt[n']{n'}\}$, so

$$2\sqrt[n']{n} > 1 + \sqrt[n']{n} > \lceil \sqrt[n']{n} \rceil = s > \max\{2, 3\sqrt[n']{n'}\} \geq 3\sqrt[n']{n'},$$

and thus $n > 1.5^{k'n'} \geq 1.5n'$ which means that we had to perform at least $n'/2$ `popBack` operations between resizings.

2. Assume the number of elements has grown from $n$ to $n'$, and $s$ became too small. It must be that

$$n' > (2s - 1)(1 + s + \ldots + s^k) > 2s^k > 2n,$$

which means that at least $n'/2$ `pushBack` operations were performed. □
By charging additional $O(k)$ credits to each `popBack` and `pushBack` (excluding those caused by resizing itself) we can amortize the cost of resizing.

**Theorem 4.** For every $k \in \mathbb{N}$ there exists a data structure which can perform `pushBack`, `popBack`, `get`, `set` operations in $O(k)$ amortized time and at any time wastes no more than $O(k \sqrt{n})$ cells of memory to store $n$ elements, where the constants hidden by $O$ notation do not depend on $k$ nor $n$.

## 5 Experimental Evaluation

Figures 1-4 present the experimental evaluation of the data structure and algorithms demonstrated in this paper for $k = 2$ and $k = 5$ compared to STL’s `vector<int>` and the Singly Resizable Array from [2]. Please consult Appendix C and [5] for tables containing numerical data and details about the way the tests were conducted. Drops visible in Figure 1 are caused by resizing which for our data structure triggers at various points dependent on the pattern of operations. Timing of operations in Figures 2-4 is comparable between all data structures and all sizes. For $n > 2^{18}$ data no longer fits in L2 cache and random access slows down for all data structures in Figure 4.

## 6 Conclusion

We have shown that for any $k \geq 2$ we can implement a resizable array, which wastes no more than $O(k \sqrt{n})$ cells of memory for storage, and performs all operations in $O(k)$. We have matched this result with a lower bound, which states, that any data structure which wastes no more than $O(k \sqrt{n})$ can not perform $n$ inserts faster than in $\Omega(k)$ amortized time. Moreover the data structure was implemented and tested for correctness and performance, achieving better memory consumption and speed comparable to alternative implementations.

## References


Fig. 1. Which root of $n$ is the amount of wasted memory for $n$ elements? Amount of wasted memory should be comparable with $\sqrt[n]{n}$ for some $k$, therefore we present at $Y$-axis a value $y$ such that the wasted memory is $y \sqrt{n}$ – the higher the value the better, and we should expect convergence to $k$. For example STL’s implementation converges to 1, Singly Resizable Array from [2] converges to 2, while our implementation for $k = 5$ very slowly converges to 5, but outperforms other implementations even for relatively small number of elements. The lines appear bold, while they are actually very densely oscillating.

Fig. 2. How long does it take to PUSHBACK an element on average if we push $n$ elements to an empty data structure?
Fig. 3. How long does it take to \textsc{popBack} an element on average if we pop all elements from a data structure initially containing \( n \) elements?

Fig. 4. How long does it take to \textsc{get} a random element if a data structure contains \( n \) elements?
A Proof of the Theorem 1

Let us start with some technical inequalities required in the proof. We use a convention that $0 \log 0$ is equal to 0.

**Fact 5.** Let $x_1, \ldots, x_m$ be a sequence of nonnegative numbers, and $x = \sum_{i=1}^{m} x_i$, then:

$$x \log \frac{x}{m} \leq \sum_{i=1}^{m} x_i \log x_i \leq x \log x$$

**Lemma 2.** Let $x_1, \ldots, x_m$ be a sequence of nonnegative numbers, $x = \sum_{i=1}^{m} x_i$, and $x > 0$. For fixed $y$, a sequence $y_1, \ldots, y_m$ of nonnegative numbers, which maximizes $\sum_{i=1}^{m} x_i \log y_i$ subject to $\sum_{i=1}^{m} y_i = y$ is given by $y_i = x_i \frac{y}{x}$.

**Proof.** If $x_i = 0$ then $y_i = 0$ as otherwise we could increase $y_j$ for some $j$ where $x_j > 0$. Assume the interesting case when there are at least two indexes $i, j$ for which $x_i, x_j > 0$. Let us fix the sum $s = y_i + y_j$ for a moment. By manipulating $y_i$ we can optimize the a small part of the goal function, namely

$$x_i \log y_i + x_j \log y_j = x_i \log y_i + x_j \log (s - y_i)$$

without affecting the rest of the sum. By differentiating this term by $y_i$ we get that the minimum is reached when the ratio between $y_i$ and $y_j$ is equal to the ratio of $x_i$ to $x_j$. If a sequence does not have this property then it cannot be optimal. The only sequence which sums up to $y$ and has correct ratios is the one proposed in the lemma. \qed

Now, assume to the contrary that there is a sequence $(M_t)$ for which we can compute $(c_t)$ and $(w_t)$ which contradict the theorem, that is assume that $w < \frac{1+c/n}{\sqrt{c/n}}$.

We will not try to analyze the algorithm itself, but rather the sequence $(M_t)$. In order to do so, we need to normalize it a little bit, in a way that will not increase $w$ nor $c$, thus the inequality will still hold for the new sequence. We do not care if there is any algorithm which could generate such normalized sequence – the proof says some truth about sequences of numbers. In particular, it may happen that some changes to the sequence we propose would actually require moving elements between blocks in a very inefficient way – this does not matter, as the theorem which we are proving does not depend on the inner workings of the algorithm.

We say that $(M_t)$ is normalized if $a_{t-1} \leq a_t$ for all $t$, that is if the data structure never shrinks. If it is not the case, let $t$ be the smallest index for which inequality does not hold. We can repair this inequality without affecting previous ones, by changing $M_t$ to be equal to $M_{t-1}$. This sets $c_t$ to 0, and might increase $c_{t+1}$, but the sum $c_t + c_{t+1}$ will not rise, as $\sum_{b \in M_t \setminus M_{t-1}} |b| + \sum_{b \in M_{t+1} \setminus M_t} |b| \geq \sum_{b \in M_{t+1} \setminus M_{t-1}} |b|$. This operation can also increase $w_t$, but not above $w_{t-1} - 1$, so the $w$ will not increase. Proceeding in this way from left to right we can normalize the whole sequence $(M_t)$. 
We will use the following observation which binds $\Delta_t$ and $w$, to simplify some terms later on:

$$\Delta_t = a_t - a_{t-1} \leq a_t + 1 - t \leq w_t \leq w.$$  

Observe that $\Phi(M_0) = 0$ and that by Fact 5 we have:

$$\Phi(M_n) = a_n \log a_n - \sum_{b \in M_n} |b| \log |b| \leq a_n \log a_n - a_n \log \frac{a_n}{|M_n|} \leq a_n \log w,$$

and thus we get an upper bound on the change of the potential:

$$\Phi(M_n) - \Phi(M_0) \leq a_n \log w.$$

**Lemma 3.** The minimum increase of potential at each step is:

$$\Phi(M_t) - \Phi(M_{t-1}) \geq \Delta_t \log \frac{a_n}{e} - (c_t - \Delta_t) \log w - \Delta_t \log c_t .$$

**Proof.** By definition of $\Phi$ we have:

$$\Phi(M_t) - \Phi(M_{t-1}) = \left( \sum_{b \in M_t \setminus M_{t-1}} |b| \log \frac{a_n}{|b|} \right) - \left( \sum_{b \in M_{t-1} \setminus M_t} |b| \log \frac{a_n}{|b|} \right)$$

$$= \left( c_t \log a_n - \sum_{b \in M_t \setminus M_{t-1}} |b| \log |b| \right) - \left( (c_t - \Delta_t) \log a_n - \sum_{b \in M_{t-1} \setminus M_t} |b| \log |b| \right)$$

There are two cases:

Case 1. $M_{t-1} \setminus M_t$ is empty. Then $c_t = \Delta_t$ and by Fact 5 we get

$$\Phi(M_t) - \Phi(M_{t-1}) \geq c_t \log a_n - c_t \log c_t = \Delta_t \log a_n - \Delta_t \log c_t .$$

Case 2. Otherwise $c_t > \Delta_t \geq 0$ and by Fact 5 we get:

$$\geq (c_t \log a_n - c_t \log c_t) - \left( (c_t - \Delta_t) \log a_n - (c_t - \Delta_t) \log \frac{c_t - \Delta_t}{|M_{t-1} \setminus M_t|} \right)$$

$$\geq \Delta_t \log a_n - c_t \log c_t + (c_t - \Delta_t) \log \frac{c_t - \Delta_t}{w}$$

$$= \Delta_t \log a_n - (c_t - \Delta_t) \log w - c_t \log c_t + (c_t - \Delta_t) \log (c_t - \Delta_t)$$

$$= \Delta_t \log a_n - (c_t - \Delta_t) \log w - \Delta_t \log c_t + (c_t - \Delta_t) \log \left( \frac{c_t - \Delta_t}{c_t} \right)$$

$$\geq \Delta_t \log a_n - (c_t - \Delta_t) \log w - \Delta_t \log c_t - \Delta_t \log e ,$$

where the last inequality holds trivially for $\Delta_t = 0$ and can be easily obtained for $\Delta_t > 0$ from:

$$(c_t - \Delta_t) \log \left( \frac{c_t - \Delta_t}{c_t} \right) = - \log \left( 1 + \frac{\Delta_t}{c_t - \Delta_t} \right)^{(c_t - \Delta_t)} = - \Delta_t \log \left( 1 + \frac{\Delta_t}{c_t - \Delta_t} \right)^{\frac{c_t - \Delta_t}{w}} .$$
Combining Lemma 3 with observation that $\sum_{t=1}^{n} \Delta_t = a_n$ we get

$$\Phi(M_n) - \Phi(M_0) = \sum_{t=1}^{n} \Phi(M_t) - \Phi(M_{t-1}) \geq a_n \frac{a_n}{e} - (c - a_n) \lg w - \sum_{t=1}^{n} \Delta_t \lg \Delta_t,$$

which by Lemma 2 is

$$\geq a_n \frac{a_n}{e} - (c - a_n) \lg w - \sum_{t=1}^{n} \Delta_t \lg \Delta_t \geq a_n \frac{a_n}{e} - (c - a_n) \lg w - a_n \frac{a_n}{e} - c \lg w.$$

So we have bound the potential change $(\Phi(M_n) - \Phi(M_0))$ from both sides. Putting this all together we get:

$$a_n \lg w \geq a_n \frac{a_n}{e} - c \lg w.$$

Dividing both sides by $a_n$ and regrouping gives:

$$(1 + \kappa) \lg w \geq \frac{a_n}{e\kappa}$$

and finally:

$$w \geq \sqrt[1+\kappa]{\frac{a_n}{e\kappa}} \geq \sqrt[n]{\frac{a_n}{e\kappa}} \geq \frac{3}{4} \sqrt[n]{\frac{n}{e}} \geq \frac{1}{4} \sqrt[n]{n}$$

$\square$

### B Proof that amortized cost of pop\text{\textsc{back}} is $O(k)$

We will analyze separately:

- zero or more splits, each of which splits one large buffer into $s$ smaller ones
- the actual pop from the 0-th block

**Fact 6.** A split of a single buffer of size $s^{i+1}$ into $s$ buffers of size $s^i$ requires $O(s^{i+1})$ operations. If performed at the moment when $i$-th block is empty, it reduces $|\alpha_i - s|$ by $s$ and increases $|\alpha_{i+1} - s|$ by at most 1, so the potential drops by at least $s\lambda_i - \lambda_{i+1} = s(k-i)s^i - (k-i-1)s^{i+1} = s^{i+1}$.

**Fact 7.** The actual pop requires $O(1)$ operations, and increases the potential by at most $\lambda_0 = k$, so the amortized cost is $O(k)$.

### C Details of Tests

This section provides more details about the tests summarized in Section 5.
C.1 Details of Implementation

The STL’s implementation was augmented with halving the size of the buffer each time it is only quarter full.

The implementation of Singly Resizable Array was the one described as Basic in [2] with explicitly added handling the case of empty data structure by using index buffer of size 8, and fixing a small bug in their formula for offset \( p \) in the procedure \( \text{LOCATE}(i) \) which should be \( p = 2^{[k/2]} + 2^{[k/2]} - 2 \) and not \( p = 2^k - 1 \).

The implementation of our data structure was also modified a little. For the 0-th block we have a single buffer of size \( 2s - 1 \) instead of \( 2s - 1 \) buffers of size 1. Also we recompute and store the value of \( \left\lceil \frac{(s/3)^k}{k} \right\rceil \) each time \( s \) changes, in order to make checking if \( s \) is too small an \( O(1) \) operation. To ensure we never have more than \( 2s - 1 \) buffers in a block we perform some steps of \( \text{PUSHBACK} \) in the opposite order, that is we start with recursive step whenever compacting \( s \) buffers is necessary to make room for the one which is being pushed.

The source code for all implementations and tests can be found online [5].

C.2 The Test Machine

The test machine was running code compiled with gcc 3.4.2 mingw-special under Cygwin environment on a 32-bit version of Windows 7 with AMD Athlon 64 Processor 3000+ 2.00 GHz with 512 KB L2 cache and 1GB of RAM.

C.3 Details of Figure 1

For each value of \( n \) we first performed \( x \) \( \text{PUSHBACK} \) operations followed by \( x - n \) \( \text{POPBACK} \) operations, for various values of \( x > n \) and the largest amount of allocated memory was recorded. Actually, to make it faster, the outer loop goes through all possible values of \( x \), and for each \( x \) we insert \( x \) elements, and then remove all of them, updating worst case values for all \( n < x \). To speed it up, only values of \( x \) from the geometric sequence \( \lceil 1.005^i \rceil \) not larger than \( 5n \) were used.

We tried using more granular values of \( x \), but it did not provide any significant precision to our estimations and obviously slowed down everything. The test for \( n \leq 10^6 \) took around 30 min on the test machine. Table 1 summarizes the results.

C.4 Details of Figure 2

As pushing \( n \) elements is actually very fast, to make measurements we had to repeat the test at least \( \max\{10^6/n, 1\} \) times. To make sure that the creation of empty data structure or computing test data does not interfere with the test, each iteration was run on a separate copy of the data structure, all of which were created before starting the clock. The times reported are measured by \( \text{clock} \) function from C standard library and divided by the number of calls to \( \text{PUSHBACK} \). Each test was repeated 5 times and the median was reported. Tests
2, 3, 4 were actually run all together in a single pass for \( n \leq 10^7 \) from the sequence \([1, 1^2]\) in less than 2h on the testing machine. Table 2 summarizes results of this test.

### C.5 Details of Figure 3

Techniques used here were similar to testing pushing, except prior to starting clock we also had to fill all the instances with the data. Table 3 summarizes results of this test.

### C.6 Details of Figure 4

Accessing \( i \)-th element was the fastest operation, so to get relevant measurements we always performed \( 10^7 \) GET calls and recorded the average. The whole procedure was repeated 5 times and the median was reported. The pseudo random sequence was generated before starting the clock using uniform distribution with a fixed seed for all values of \( n \) and implementations. Table 4 summarizes results of this test.

**Table 1.** The table presents for each \( n \) the maximum amount of wasted memory which can be observed for each of four implementations when the number of elements is at most \( n \). The values of \( k \sqrt[4]{n} \) are provided as a reference. The memory is measured in 4-byte words capable of storing a single pointer or an element.

<table>
<thead>
<tr>
<th>( n )</th>
<th>STL's vector with halving</th>
<th>Singly Resizable Array [2]</th>
<th>Resizable Array for ( k = 2 )</th>
<th>Resizable Array for ( k = 5 )</th>
<th>( 2\sqrt[4]{n} )</th>
<th>( 5\sqrt[4]{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>21</td>
<td>20</td>
<td>46</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td>34</td>
<td>44</td>
<td>47</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>194</td>
<td>90</td>
<td>116</td>
<td>83</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>1000</td>
<td>1538</td>
<td>202</td>
<td>332</td>
<td>119</td>
<td>64</td>
<td>20</td>
</tr>
<tr>
<td>10000</td>
<td>24578</td>
<td>778</td>
<td>968</td>
<td>215</td>
<td>200</td>
<td>32</td>
</tr>
<tr>
<td>100000</td>
<td>196610</td>
<td>2570</td>
<td>2888</td>
<td>215</td>
<td>633</td>
<td>51</td>
</tr>
<tr>
<td>1000000</td>
<td>1572866</td>
<td>6154</td>
<td>8648</td>
<td>215</td>
<td>2000</td>
<td>80</td>
</tr>
<tr>
<td>4000000</td>
<td>6291458</td>
<td>12298</td>
<td>8648</td>
<td>215</td>
<td>4000</td>
<td>105</td>
</tr>
</tbody>
</table>
Table 2. The table presents for each $n$ the result of performing multiple iterations of a test consisting of $n$ pushBack operations. We present the total number of pushes performed during all iterations, and for each implementation the total time of the test and the average time necessary to push a single element measured in ticks and micro $(10^{-6})$ ticks respectively.

<table>
<thead>
<tr>
<th>$n$</th>
<th>pushes</th>
<th>STL's vector with halving</th>
<th>Singly Resizable Array [2]</th>
<th>Resizable Array for $k=2$</th>
<th>Resizable Array for $k=5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>avg</td>
<td>sum</td>
<td>avg</td>
<td>sum</td>
</tr>
<tr>
<td>10</td>
<td>1000000</td>
<td>141 140 140 140µ 156 156µ 172 172µ</td>
<td>156 156µ 172 172µ</td>
<td>157 157µ 166 166µ</td>
<td>209 209µ 249 249µ</td>
</tr>
<tr>
<td>100</td>
<td>1000000</td>
<td>47 47 47 47µ 63 63µ 109 109µ 156 156µ 172 172µ</td>
<td>157 157µ 166 166µ</td>
<td>209 209µ 249 249µ</td>
<td>250 250µ 290 290µ</td>
</tr>
<tr>
<td>1000</td>
<td>1000000</td>
<td>31 31 31 31µ 47 47µ 63 63µ 156 156µ 172 172µ</td>
<td>157 157µ 166 166µ</td>
<td>209 209µ 249 249µ</td>
<td>250 250µ 290 290µ</td>
</tr>
<tr>
<td>10000</td>
<td>1000000</td>
<td>31 31 31 31µ 47 47µ 63 63µ 156 156µ 172 172µ</td>
<td>157 157µ 166 166µ</td>
<td>209 209µ 249 249µ</td>
<td>250 250µ 290 290µ</td>
</tr>
<tr>
<td>100000</td>
<td>1000000</td>
<td>344 343 453 453µ 687 687µ 969 969µ 1000 1000µ</td>
<td>1000 1000µ 1000µ</td>
<td>1000 1000µ 1000µ</td>
<td>1000 1000µ 1000µ</td>
</tr>
</tbody>
</table>

Table 3. The table presents for each $n$ the result of performing multiple iterations of a test consisting of popping all elements from a data structure initially containing $n$ elements. We present total number of pops performed during all iterations, and for each implementation the total time of the test and the average time necessary to pop a single element.

<table>
<thead>
<tr>
<th>$n$</th>
<th>pops</th>
<th>STL's vector with halving</th>
<th>Singly Resizable Array [2]</th>
<th>Resizable Array for $k=2$</th>
<th>Resizable Array for $k=5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>avg</td>
<td>sum</td>
<td>avg</td>
<td>sum</td>
</tr>
<tr>
<td>10</td>
<td>1000000</td>
<td>156 156µ 79 79µ 156 156µ 172 172µ</td>
<td>156 156µ 172 172µ</td>
<td>125 125µ 282 282µ</td>
<td>187 187µ 282 282µ</td>
</tr>
<tr>
<td>100</td>
<td>1000000</td>
<td>63 63µ 78 78µ 125 125µ 282 282µ</td>
<td>125 125µ 282 282µ</td>
<td>187 187µ 282 282µ</td>
<td>282 282µ 282 282µ</td>
</tr>
<tr>
<td>1000</td>
<td>1000000</td>
<td>47 47µ 47 47µ 78 78µ 141 141µ</td>
<td>78 78µ 141 141µ</td>
<td>187 187µ 282 282µ</td>
<td>282 282µ 282 282µ</td>
</tr>
<tr>
<td>10000</td>
<td>1000000</td>
<td>47 47µ 31 31µ 78 78µ 141 141µ</td>
<td>78 78µ 141 141µ</td>
<td>187 187µ 282 282µ</td>
<td>282 282µ 282 282µ</td>
</tr>
<tr>
<td>100000</td>
<td>1000000</td>
<td>62 62µ 31 31µ 47 47µ 94 94µ 109 109µ</td>
<td>47 47µ 94 94µ</td>
<td>109 109µ 109 109µ</td>
<td>109 109µ 109 109µ</td>
</tr>
<tr>
<td>1000000</td>
<td>1000000</td>
<td>562 562µ 391 391µ 656 656µ 656 656µ 1000 1000µ</td>
<td>656 656µ 1000 1000µ</td>
<td>1000 1000µ 1000 1000µ</td>
<td>1000 1000µ 1000 1000µ</td>
</tr>
</tbody>
</table>

Table 4. The table presents for each $n$ and for each implementation the total time to perform all $10^7$ get operations and the average time of a single operation on a data structure containing $n$ elements.

<table>
<thead>
<tr>
<th>$n$</th>
<th>STL's vector with halving</th>
<th>Singly Resizable Array [2]</th>
<th>Resizable Array for $k=2$</th>
<th>Resizable Array for $k=5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
<td>avg</td>
<td>sum</td>
<td>avg</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td>11µ 297 29µ 578 57µ 1250 125µ</td>
<td>297 29µ 578 57µ</td>
<td>1250 125µ 125µ</td>
</tr>
<tr>
<td>100</td>
<td>125</td>
<td>12µ 297 29µ 531 53µ 766 76µ</td>
<td>297 29µ 531 53µ</td>
<td>766 76µ 76µ</td>
</tr>
<tr>
<td>1000</td>
<td>110</td>
<td>11µ 328 32µ 532 53µ 625 62µ</td>
<td>328 32µ 532 53µ</td>
<td>625 62µ 62µ</td>
</tr>
<tr>
<td>10000</td>
<td>110</td>
<td>11µ 328 32µ 375 37µ 641 64µ</td>
<td>328 32µ 375 37µ</td>
<td>641 64µ 64µ</td>
</tr>
<tr>
<td>100000</td>
<td>172</td>
<td>17µ 375 37µ 500 50µ 688 68µ</td>
<td>375 37µ 500 50µ</td>
<td>688 68µ 68µ</td>
</tr>
<tr>
<td>1000000</td>
<td>703</td>
<td>70µ 1094 109µ 1062 106µ 1109 110µ</td>
<td>1094 109µ 1062 106µ</td>
<td>1109 110µ 110µ</td>
</tr>
<tr>
<td>10000000</td>
<td>625</td>
<td>62µ 1203 120µ 1313 131µ 1266 126µ</td>
<td>1203 120µ 1313 131µ</td>
<td>1266 126µ 126µ</td>
</tr>
</tbody>
</table>