Resizable Arrays in Optimal Time and Space

Andrej Brodnik*† Svante Carlsson† Erik D. Demaine‡
J. Ian Munro‡ Robert Sedgewick§

Abstract

We present simple, practical and efficient data structures for the fundamental problem of maintaining a resizable one-dimensional array, $A[l \ldots l+n-1]$, of fixed-size elements, as elements are added to or removed from one or both ends. In addition to these operations, our data structures support access to the element in position i. All operations are performed in constant time. The extra space (i.e., the space used past storing the n current elements) is $O(\sqrt{n})$ at any point in time. This is shown to be within a constant factor of optimal, even if there are no constraints on the time. If desired, each memory block can be made to have size $2^k - c$ for a specified constant c, and hence the scheme works effectively with the buddy system. The data structures can be used to solve a variety of problems with optimal bounds on time and extra storage. These include stacks, queues, randomized queues, priority queues, and deques.

1 Introduction

The initial motivation for this research was a fundamental problem arising in many randomized algorithms [14, 17, 20]. Specifically, a *randomized queue* is to maintain a collection of fixed-size elements, such as word-size integers or pointers, and support the following operations:

1. Insert (e): Add a new element e to the collection.
2. DeleteRandom: Delete and return an element chosen uniformly at random from the collection.

That is, if n is the current size of the set, DeleteRandom must choose each element with probability $1/n$. We assume our random number generator returns a random integer between 1 and n in constant time.

*Department of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, Jadranska 19, 1111 Ljubljana, Slovenia, email: `Andrej.Brodnik@IMFM.Uni-Lj.SI`

†Department of Computer Science and Electrical Engineering, Luleå University of Technology, S-971 87 Luleå, Sweden, email: `svante@sm.luth.se`

‡Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, email: {`eddemaine`, `imunro`}`@uwaterloo.ca`

§Department of Computer Science, Princeton University, Princeton, NJ 08544, U.S.A., email: `rs@cs.princeton.edu`

At first glance, this problem may seem rather trivial. However, it becomes more interesting after we impose several important restrictions. The first constraint is that the data structure must be theoretically efficient: the operations should run in constant time, and the extra storage should be minimal. The second constraint is that the data structure must be practical: it should be simple to implement, and perform well under a reasonable model of computation, e.g., when the memory is managed by the buddy system. The final constraint is more amusing and was posed by one of the authors: the data structure should be presentable at the first or second year undergraduate level in his text [20].

One natural implementation of randomized queues stores the elements in an array and uses the *doubling technique* [9]. Insert (e) simply adds e to the end of the array, increasing n. If the array is already full, Insert first resizes it to twice the size. DeleteRandom chooses a random integer between 1 and n, and retrieves the array element with that index. It then moves the last element of the array to replace that element, and decreases n, so that the first n elements in the array always contain the current collection.

This data structure correctly implements the Insert and DeleteRandom operations. In particular, moving the last element to another index preserves the randomness of the elements chosen by DeleteRandom. Furthermore, both operations run in $O(1)$ amortized time: the only part that takes more than constant time is the resizing of the array, which consists of allocating a new array of double the size, copying the elements over, and deallocating the old array. Because $n/2$ new elements were added before this resizing occurred, we can charge the $O(n)$ cost to them, and achieve a constant amortized time bound. The idea is easily extended to permit shrinkage: simply halve the size of the structure whenever it drops to one third full. The amortization argument still goes through.

The $O(n)$ elements of space occupied by this structure is optimal up to a constant factor, but is still too much. Granted, we require at least n units of space to store the collection of elements, but we do not require $4.5n$ units, which this data structure occupies while shrinkage is taking place. We want the *extra space*, the space in excess of n units, to be within a constant factor of optimal, so we are looking for an $n + o(n)$ solution.

1.1 Resizable Arrays

This paper considers a generalization of the randomized queue problem to (one-dimensional) resizable arrays. A *singly resizable array* maintains a collection of n fixed-size elements, each assigned a unique index between 0 and $n - 1$, subject to the following operations:

1. Read (i): Return the element with index i, $0 \leq i < n$.
2. Write (i, x): Set the element with index i to x, $0 \leq i < n$.
3. Grow: Increment n, creating a new element with index n.
4. Shrink: Decrement n, discarding the element with index $n - 1$.

As we will show, singly resizable arrays solve a variety of fundamental data-structure problems, including randomized queues as described above, stacks, priority queues, and indeed queues. In addition, many modern programming languages provide built-in abstract data types for resizable arrays. For example, the C++ vector class [21, sec. 16.3] is such an ADT.

Typical implementations of resizable arrays in modern programming systems use the "doubling" idea described above, growing resizable arrays by a constant factor c, where c can be a user-defined parameter. This implementation has the major drawback that the amount of wasted space is linear in n, which is unnecessary. Optimal space usage is essential in modern programming applications with many resizable arrays each of different size. For example, in a language such as C++, one might use compound data structures such as stacks of queues or priority queues of stacks that could involve all types of resizable structures of varying sizes. Modern programming applications might involve a few huge resizable arrays, or a huge number of small resizable arrays. For example, in a language such as C++, one might use compound data structures such as stacks of queues or a queues of deques that could involve all types of resizable structures of varying sizes. For example, multidimensional arrays in C and C++ fall into this category. Optimal space usage is essential in such applications.

In this paper, we present an optimal data structure for singly resizable arrays. The worst-case running time of each operation is a small constant. The extra storage at any point in time is $O(\sqrt{n})$, which is shown to be optimal up to a constant factor.[1] Furthermore, the algorithms are simple, and suitable for use in practical systems. While our exposition here is designed to prove the most general results possible, we believe that one could present one of the data structures (e.g., our original goal of the randomized queue) at the first or second year undergraduate level.

1.2 Deques

A natural extension is the efficient implementation of a *deque* (or double-ended queue), which supports inserts and deletions at both ends. While we cannot implement deques using singly resizable arrays, they are the natural consequence of a variant that can resize on both ends. Specifically, a *doubly resizable array* which maintains a collection of n fixed-size elements. Each element is assigned a unique index between ℓ and u (where $u - \ell + 1 = n$ and ℓ, u are potentially negative), subject to the following operations:

1. Read (i): Return the element with index i, $\ell \leq i \leq u$.
2. Write (i, x): Set the element with index i to x, $\ell \leq i \leq u$.
3. GrowForward: Increment u, creating a new element with index $u + 1$.
4. ShrinkForward: Decrement u, discarding the element with index u.
5. GrowBackward: Decrement ℓ, creating a new element with index $\ell - 1$.
6. ShrinkBackward: Increment ℓ, discarding the element with index ℓ.

An extension to our method for singly resizable arrays supports this data type in the same optimal time and space bounds.

1.3 Connection to Hashing

Resizing has traditionally been explored in the context of hash tables [9]. Knuth [14, vol. 3, p. 540] traces the idea of resizing, in conjunction with hashings, back at least to Hopgood in

[1] For simplicity of exposition, we ignore the case $n = 0$ in our bounds; the correct statement for a bound of $O(b)$ is the more tedious $O(1 + b)$.

1968. Since Knuth, it has appeared in many basic textbooks on algorithms, e.g. [1, 20].

For hashing, changing the size of the table involves rehashing all the keys, and that cost is amortized, so the basic method is effective only when we grow the table by a constant factor. The insight of this paper is that, for many simpler data structures, we only need to amortize the resizing cost, and can change the size of the array in smaller increments, thus wasting far less space.

1.4 Outline

The rest of this paper is organized as follows. Section 2 describes our fairly realistic model for dynamic memory allocation. In Section 3, we present a lower bound on the required extra storage for resizable arrays. Section 4 presents our data structure for singly resizable arrays. Section 5 describes several applications of this result, namely optimal data structures for stacks, queues, randomized queues, and priority queues. Finally, Section 6 considers deques, which require us to look at a completely new data structure for doubly resizable arrays.

2 Model

Our model of computation is a fairly realistic mix of several popular models: a transdichotomous [10] random access machine in which memory is dynamically allocated. Our model is *random access* in the sense that any element in a block of memory can be accessed in constant time, given just the block pointer and an integer index into the block. Fredman and Willard [10] introduced the term *transdichotomous* to capture the notion of the problem size matching the machine word size. That is, a word is large enough to store the problem size, and so has at least $\lceil \log_2(1+n) \rceil$ bits (but not many more). In practice, it is usually the case that the word size is fixed but larger than $\log_2 M$ where M is the size of the memory (which is certainly at least $n+1$). Our model of dynamic memory allocation matches that available in most current systems and languages, for example the standard C library. Three operations are provided:

1. Allocate (s): Returns a new block of size s.
2. Deallocate (B): Frees the space used by the given block B.
3. Reallocate $(B,\, s)$: If possible, resizes the block B to the specified size s. Otherwise, allocates a block of size s, into which it copies the contents of B, and deallocates B. In either case, the operation returns the resulting block of size s.

Hence, in the worst case, Reallocate degenerates to an Allocate, a block copy, and a Deallocate. It may be more efficient in certain practical cases, but it offers no theoretical benefits.

In our analysis we ignore the running time of these three memory operations, effectively assuming that they run in constant time. This is not practical, however. In particular, as we just mentioned, Reallocate can often take a linear amount of time. Furthermore, Allocate (s) often takes $\Theta(s)$ time in order to zero out the memory being allocated, for security reasons. However, our point of view is justified because we prove that whenever our algorithms allocate a block of size s, the next call to Allocate or Deallocate is $\Omega(s)$ units of time away. Hence our worst-case time bounds convert directly into amortized time bounds if the cost of memory allocation is linear instead of constant.

A memory block B consists of the user's data, whose size we denote by $|B|$, plus a header of fixed size h. In many cases, it is desirable to have the *total size* of a block equal to a power of two, that is, have $|B| = 2^k - h$ for some k. This is particularly important in the binary buddy system [14, vol. 1, p. 435], [8] which would otherwise round to the next power of two. If the blocks contained user data whose sizes were powers of two, half of the space would be wasted.

The amount of space occupied by a data structure is the sum of total block sizes, that is, it includes the space occupied by headers. Hence, to achieve $o(n)$ extra storage, there must be $o(n)$ allocated blocks.

3 Lower Bound

Theorem 1 $\Omega(\sqrt{n})$ *extra storage is necessary in the worst case for any data structure that supports inserting elements, and deleting those elements in some (arbitrary) order. In particular, this lower bound applies to resizable arrays, stacks, queues, randomized queues, priority queues, and deques.*

Proof: Consider the following sequence of operations:

$$\mathsf{Insert}\ (a_1), \ldots, \mathsf{Insert}\ (a_n), \underbrace{\mathsf{Delete}, \ldots, \mathsf{Delete}}_{n \text{ times}}.$$

Apply the data structure to this sequence, separately for each value of n. Consider the state of the data structure between the inserts and the deletes: let $s(n)$ be the size of the largest memory block, and let $k(n)$ be the number of memory blocks. Because all the elements are about to be reported to the user (in an arbitrary order), the elements must be stored in memory. Hence, $s(n) \cdot k(n)$ must be at least n.

At the time between the inserts and the deletes, the amount of extra storage is at least $hk(n)$ to store the memory block headers, and hence the worst-case extra storage is at least $k(n)$. Furthermore, at the time immediately after the block of size $s(n)$ was allocated, the extra storage was at least $s(n)$. Hence, the worst-case extra storage is at least $\max\{s(n), k(n)\}$. Because $s(n) \cdot k(n) \geq n$, the minimum worst-case extra storage is at least $\sqrt{n}$. $\square$

This theorem also applies to the related problem of vectors in which elements can be inserted and deleted at any position. Here constant-time updates and queries are not possible. Goodrich and Kloss [11] show that $O(n^{\varepsilon})$ amortized time suffices for updates, for any $\varepsilon > 0$, even when access queries must be performed in constant time. Their $\varepsilon = 1/2$ data structure uses $O(\sqrt{n})$ extra space, which as we see is optimal. It is an open problem whether $O(\sqrt{n})$ extra space is sufficient to achieve the same time bounds as in [11].

4 Singly Resizable Arrays

The basic idea of our first data structure is to store the elements of the array in $\Theta(\sqrt{n})$ blocks, each of size roughly $\sqrt{n}$. Now because n is changing over time, and we allocate the blocks one-by-one, the blocks have sizes ranging from $\Theta(1)$ to $\Theta(\sqrt{n})$. One obvious choice is to give the ith block size i, thus having $k(k+1)/2$ elements in the first k blocks. The number of blocks required to store n elements, then, is $\left\lceil(\sqrt{1+8n}-1)/2\right\rceil = \Theta(\sqrt{n})$.

The problem with this choice of block sizes is the cost of finding a desired element in the collection. More precisely, the Read and Write operations must first determine which element in which block has the specified index, in what we call the Locate operation. With the block sizes above, computing which block contains the desired element i requires computing the square root of $1+8i$. Newton's method [19, pp. 274–292] is known to minimize the time for this, taking $\Theta(\log\log i)$ time in the worst case. This prevents Read and Write from running in the desired $O(1)$ time bound.[2]

Another approach, related to that of doubling, is to use a sequence of blocks of sizes the powers of 2, starting with 1. The obvious disadvantage of these sizes is that half the storage space is wasted when the last block is allocated and contains only one element. We notice however that the number of elements in the first k blocks is $2^k - 1$, so the block containing element i is $\lfloor\log_2(1+i)\rfloor$. This is simply the position of the leading 1-bit in the binary representation of $i+1$ and can be computed in $O(1)$ time (see Section 4.3).

Our solution is to sidestep the disadvantages of each of the above two approaches by combining them so that Read and Write can be performed in $O(1)$ time, but the amount of extra storage is at most $O(\sqrt{n})$. The basic idea is to have conceptual *superblocks* of size 2^i, each split into approximately $2^{i/2}$ blocks of size approximately $2^{i/2}$. Determining which superblock contains element i can be done in $O(1)$ time as described above. Actual allocation of space is by block, instead of by superblock, so only $O(\sqrt{n})$ storage is wasted at any time.

This approach is described more thoroughly in the following sections. We begin in Section 4.1 with a description of the basic version of the data structure. Sections 4.2 and 4.3 prove the storage and time bounds, respectively. Section 4.4 shows how to modify the algorithms to make most memory blocks have total size a power of two, including the size of the block headers.

4.1 Basic Version

The basic version of the data structure consists of two types of memory blocks: one *index block*, and several *data blocks*. The index block simply contains pointers to all of the data blocks. The data blocks, denoted $DB_0, \ldots, DB_{d-1}$, store all of the elements in the resizable array. Data blocks are clustered into *superblocks* as follows: two data blocks are in the same superblock precisely if they have the same size. Although superblocks have no physical manifestation, we will find it useful to talk about them with some notation, namely $SB_0, \ldots, SB_{s-1}$. When superblock SB_k is fully allocated, it consists of $2^{\lfloor k/2\rfloor}$ data blocks, each of size $2^{\lceil k/2\rceil}$. Hence, there are a total of 2^k elements in superblock SB_k. See Figure 1.

[2] In fact, one can use $O(\sqrt{n})$ storage for a lookup table to support constant-time square-root computation, using ideas similar to those in Section 4.3. Here we develop a much cleaner algorithm.

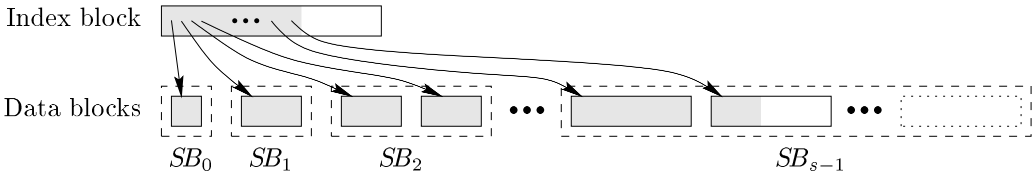

Figure 1: A generic snapshot of the basic data structure.

Grow:

1. If the last nonempty data block DB_{d-1} is full:
   (a) If the last superblock SB_{s-1} is full:
      i. Increment s.
      ii. If s is odd, double the number of data blocks in a superblock.
      iii. Otherwise, double the number of elements in a data block.
      iv. Set the occupancy of SB_{s-1} to empty.
   (b) If there are no empty data blocks:
      i. If the index block is full, Reallocate it to twice its current size.
      ii. Allocate a new last data block, and store a pointer to it in the index block.
   (c) Increment d and the number of data blocks occupying SB_{s-1}.
   (d) Set the occupancy of DB_{d-1} to empty.
2. Increment n and the number of elements occupying DB_{d-1}.

Algorithm 1: Basic implementation of Grow.

We reduce the four resizable-array operations to three "fundamental" operations as follows. Grow and Shrink are defined to be already fundamental; they are sufficiently different that we do not merge them into a single "resize" operation. The other two operations, Read and Write, are implemented by a common operation Locate (i) which determines the location of the element with index i.

The implementations of the three fundamental array operations are given in Algorithms 1–3. Basically, whenever the last data block becomes full, another one is allocated, unless an empty data block is already around. Allocating a data block may involve doubling the size of the index block. Whenever two data blocks become empty, the younger one is deallocated; and whenever the index block becomes less than a quarter full, it is halved in size. To find the block containing a specified element, we find the superblock containing it by computing the leading 1-bit, then the appropriate data block within the superblock, and finally the element within that data block.

Note that the data structure also has a constant-size block, which stores the number of elements (n), the number of superblocks (s), the number of nonempty data blocks (d), the number of empty data blocks (which is always 0 or 1), and the size and occupancy of the last nonempty data block, the last superblock, and the index block.

In the next two sections, we prove the following theorem:

Theorem 2 *This data structure implements a singly resizable array using $O(\sqrt{n})$ extra storage in the worst case and $O(1)$ time per operation, on a random access machine where memory is dynamically allocated, and binary shift by k takes $O(1)$ time on a word of size*

Shrink:

1. Decrement n and the number of elements occupying the last nonempty data block DB_{d-1}.
2. If DB_{d-1} is empty:
   (a) If there is another empty data block, Deallocate it.
   (b) If the index block is a quarter full, Reallocate it to half its size.
   (c) Decrement d and the number of data blocks occupying the last superblock SB_{s-1}.
   (d) If SB_{s-1} is empty:
      i. Decrement s.
      ii. If s is even, halve the number of data blocks in a superblock.
      iii. Otherwise, halve the number of elements in a data block.
      iv. Set the occupancy of SB_{s-1} to full.
   (e) Set the occupancy of DB_{d-1} to full.

Algorithm 2: Basic implementation of Shrink.

Locate (i):

1. Let r denote the binary representation of $i+1$, with all leading zeros removed.
2. Note that the desired element i is element e of data block b of superblock k, where
   (a) $k = |r| - 1$,
   (b) b is the $\lfloor k/2 \rfloor$ bits of r immediately after the leading 1-bit, and
   (c) e is the last $\lceil k/2 \rceil$ bits of r.
3. Let $p = 2^k - 1$ be the number of data blocks in superblocks prior to SB_k.
4. Return the location of element e in data block DB_{p+b}.

Algorithm 3: Basic implementation of Locate.

$\lceil \log_2(1+n) \rceil$. *Furthermore, if* **Allocate** *or* **Deallocate** *is called when $n = n_0$, then the next call to* **Allocate** *or* **Deallocate** *will occur after $\Omega(\sqrt{n_0})$ operations.*

Some of our applications, namely stacks and queues, do not need general Read and Write operations. For these applications, the data structure does not use the transdichotomous model. In addition, our second data structure (presented in Section 6) does not use the transdichotomous model even for general Reads and Writes, and furthermore supports doubly resizable arrays.

4.2 The Space Bound

In this section we show that the worst-case extra storage in the basic data structure is $O(\sqrt{n})$, which by Theorem 1 is optimal. We need some preliminary lemmas:

Lemma 1 *The number of superblocks (s) is $\lceil \log_2(1+n) \rceil$.*

Proof: The number of elements in the first s superblocks is

$$\sum_{i=0}^{s-1} 2^i = 2^s - 1.$$

If we let this equal n, then $s = \log_2(1+n)$. For slightly smaller n, the same number of superblocks is required, and hence we get a ceiling operator. $\square$

Lemma 2 *At any point in time, the number of data blocks is $O(\sqrt{n})$.*

Proof: Because there is always at most one empty data block, and we assume $n > 0$, it suffices to consider just the nonempty data blocks. In the worst case, the number of data blocks in each superblock SB_i is the maximum allowed number $2^{\lfloor i/2 \rfloor}$. Hence, the maximum number of nonempty data blocks is

$$\sum_{i=0}^{s-1} 2^{\lfloor i/2 \rfloor} \leq \sum_{i=0}^{s-1} 2^{i/2} = \frac{2^{s/2}-1}{\sqrt{2}-1},$$

but $2^s = O(n)$ by Lemma 1, so this is $O(\sqrt{n})$ as desired. $\square$

Lemma 3 *The last (empty or nonempty) data block has size $\Theta(\sqrt{n})$.*

Proof: The last superblock is SB_{s-1}, where $s = \lceil \log_2(1+n) \rceil$ by Lemma 1. By the construction, it contains data blocks of size

$$2^{\lceil \frac{s-1}{2} \rceil} = \Theta(1) 2^{s/2} = \Theta(1)\sqrt{2^{\lceil \log_2(1+n) \rceil}} = \Theta(1)\sqrt{2^{\log_2 n}} = \Theta(\sqrt{n}),$$

as desired. If there is an empty data block, it has the same size or twice the size, which is also $\Theta(\sqrt{n})$. $\square$

We can now argue that the extra storage at any point in time is $O(\sqrt{n})$. The size of the index block is at most four times the number of data blocks, so by Lemma 2 has size $O(\sqrt{n})$. The wastage from block headers is $O(\sqrt{n})$ by Lemma 2. All data blocks except the last nonempty data block, and possibly an empty data block saved for later use, are full of elements; by Lemma 3, these two blocks have size (and so maximum wastage) $O(\sqrt{n})$.

4.3 Time Bound

To prove the time bound, we first show a bound of $O(1)$ for Locate, and then show how to implement Reallocate first in $O(1)$ amortized time and then in $O(1)$ worst-case time.

The key issue in performing Locate is the determination of $k = \lceil \log_2(1+i) \rceil$, the position of the leading 1-bit in the binary representation of $i+1$. Many modern machines include this instruction; newer Pentium chips do it as quickly as an integer addition. Brodnik [3] gives a constant-time method using only basic arithmetic and bitwise boolean operators. Another very simple method is to store all solutions of "half-length," that is for values of i up to $2^{\lfloor (\log_2(1+n))/2 \rfloor} = \Theta(\sqrt{n})$. Two probes into this lookup table now suffice. We check for the leading 1-bit in the first half of the $1 + \lfloor \log_2(1+n) \rfloor$ bit representation of i, and if there is no 1-bit, check the trailing bits. The lookup table is easily maintained as n changes. From this we see that Algorithm 3 runs in constant time.

We now have an $O(1)$ time bound if we can ignore the cost of dynamic memory allocation. First let us show that Allocate and Deallocate are only called once every $\Omega(\sqrt{n})$ operations as claimed in Theorem 2. Note that immediately after allocating or deallocating a data block, the number of unused elements in data blocks is the size of the last data block. Because we only deallocate a data block after two are empty, we must have called Shrink at least as many times as the size of the remaining empty block, which is $\Omega(\sqrt{n})$ by Lemma 3. Because we only allocate a data block after the last one becomes full, we must have called Grow at least as many times as the size of the now full block, which again is $\Omega(\sqrt{n})$.

Thus, the only remaining cost to consider is that of resizing the index block and the lookup table (if we use one), as well as maintaining the contents of the lookup table. These resizes only occur after $\Omega(\sqrt{n})$ data blocks have been allocated or deallocated, each of which (as we have shown) only occurs after $\Omega(\sqrt{n})$ updates to the data structure. Hence, the cost of resizing the index block and maintaining the lookup table, which is $O(n)$, can be amortized over these updates, so we have an $O(1)$ amortized time bound.

One can achieve a worst-case running time of $O(1)$ per operation as follows. In addition to the normal index block, maintain two other blocks, one of twice the size and the other of half the size, as well as two counters indicating how many elements from the index block have been copied over to each of these blocks. In allocating a new data block and storing a pointer to it in the index block, also copy the next two uncopied pointers (if there are any) from the index block into the double-size block. In deallocating a data block and removing the pointer to it, also copy the next two uncopied pointers (if there are any) from the index block into the half-sized block.

Now when the index block becomes full, all of the pointers from the index block have been copied over to the double-size block. Hence, we Deallocate the half-size block, replace the half-size block with the index block, replace the index block with the double-size block, and Allocate a new double-size block. When the index block becomes a quarter full, all of the pointers from the index block have been copied over to the half-size block. Hence, we Deallocate the double-size block, replace the double-size block with the index block, replace the index block with the half-size block, and Allocate a new half-size block.

The maintenance of the lookup table can be done in a similar way. The only difference is that whenever we allocate a new data block and store a pointer to it in the index block, in addition to copying the next two uncopied elements (if there are any), compute the next two uncomputed elements in the table. Note that the computation is done trivially, by monitoring when the answer changes, that is, when the question doubles. Note also that this method only adds a constant multiplicative factor to the extra storage, so it is still $O(\sqrt{n})$. The time per operation is therefore $O(1)$ in the worst case.

4.4 The Buddy System

In the basic data structure described so far, the data blocks have user data of size a power of two. Because some memory management systems add a block header of fixed size, say h, the total size of each block can be slightly more than a power of two ($2^k + h$ for some k). This is inappropriate for a memory management system that prefers blocks of total size a power of two. For example, the (binary) buddy system [14, vol. 1, p. 540], [8] rounds the total block size to the next power of two, so the basic data structure would use twice as much storage as required, instead of the desired $O(\sqrt{n})$ extra storage. While the buddy system is rarely used exclusively, most UNIX operating systems (e.g., BSD [16, pp. 128–132]) use it for small block sizes, and allocate in multiples of the page size (which is also a power of two) for larger block sizes. Therefore, creating blocks of total size a power of two produces substantial savings on current computer architectures, especially for small values of n.

This section describes how to solve this problem by making the size of the user data in every data block equal to $2^k - h$ for some k. As far as we know, this is the first theoretical algorithm designed to work effectively with the buddy system. To preserve the ease of finding the superblock containing element number i, we still want to make the total number of elements in superblock SB_k equal to 2^k. To do this, we introduce a new type of block called an *overflow block*. There will be precisely one overflow block OB_k per superblock SB_k. This overflow block is of size $h2^{\lfloor k/2 \rfloor}$, and hence any waste from using the buddy system is $O(\sqrt{k})$.

Conceptually, the overflow block stores the last h elements of each data block in the superblock. We refer to a data block DB_i together with the corresponding h elements in the overflow block as a *conceptual block* CB_i. Hence, each conceptual block in superblock SB_k has size $2^{\lceil k/2 \rceil}$, as did the data blocks in the basic data structure.

We now must maintain two index blocks: the *data index block* stores pointers to all the data blocks as before, and the *overflow index block* stores pointers to all the overflow blocks. As before, we double the size of an index block whenever it becomes full, and halve its size whenever it becomes a quarter full.

The algorithms for the three fundamental operations are given in Algorithms 4–6. They are similar to the previous algorithms; the only changes are as follows. Whenever we want to insert or access an element in a conceptual block, we first check whether the index is in the last h possible values. If so, we use the corresponding region of the overflow block, and otherwise we use the data block as before. The only other difference is that whenever we change the number of superblocks, we may allocate or deallocate an overflow block, and potentially resize the overflow index block.

We obtain an amortized or worst-case $O(1)$ time bound as before. It remains to show that the extra storage is still $O(\sqrt{n})$. The number s of overflow blocks is $O(\log n)$ by Lemma 1, so the block headers from the overflow blocks are sufficiently small. Only the last overflow block may not be full of elements; its size is at most h times the size of the last data block, which is $O(\sqrt{n})$ by Lemma 3. The overflow index block is at most the size of the data index block, so it is within the bound. Finally, note that the blocks whose sizes are not powers of two (the overflow blocks and the index blocks) have a total size of $O(\sqrt{n})$, so doubling their size does not affect the extra storage bound.

Hence, we have proved the following theorem.

Theorem 3 *This data structure implements a singly resizable array in $O(\sqrt{n})$ worst-case extra storage and $O(1)$ time per operation, on a $\lceil \log_2(1+n) \rceil$ bit word random access machine where memory is dynamically allocated in blocks of total size a power of two, and binary shift by k takes $O(1)$ time. Furthermore, if* Allocate *or* Deallocate *is called when $n = n_0$, then the next call to* Allocate *or* Deallocate *will occur after $\Omega(\sqrt{n_0})$ operations.*

Grow:

1. If the last nonempty conceptual block CB_{d-1} is full:
   (a) If the last superblock SB_{s-1} is full:
      i. Increment s.
      ii. If s is odd, double the number of data blocks in a superblock.
      iii. Otherwise, double the number of elements in a conceptual block.
      iv. Set the occupancy of SB_{s-1} to empty.
      v. If there are no empty overflow blocks:
         - If the overflow index block is full, Reallocate it to twice its current size.
         - Allocate a new last overflow block, and store a pointer to it in the overflow index block.
   (b) If there are no empty data blocks:
      i. If the data index block is full, Reallocate it to twice its current size.
      ii. Allocate a new last data block, and store a pointer to it in the data index block.
   (c) Increment d and the number of data blocks occupying SB_{s-1}.
   (d) Set the occupancy of CB_{d-1} to empty.
2. Increment n and the number of elements occupying CB_{d-1}.

Algorithm 4: Buddy implementation of Grow.

5 Applications of Singly Resizable Arrays

This section describes a variety of fundamental data structures that are solved optimally (with respect to time and worst-case extra storage) by the data structure for singly resizable arrays described in the previous section. Sections 5.1 and 5.2 begin with the simple data structures of stacks and queues, respectively. Sections 5.3 and 5.4 examine the more interesting cases of randomized queues and priority queues, respectively.

5.1 Stacks

A stack is trivial to implement using a singly resizable array. The operation Push (e) calls Grow, and Writes e to the last element of the array. Top just Reads the last element of the array, and Pop simply calls Shrink. We thus have proved

Corollary 1 *Stacks can be implemented in $O(1)$ worst-case time per operation, and $O(\sqrt{n})$ worst-case extra storage.*

Note however that the Read and Write operations are unnecessary for stacks when using the data structure presented in Section 4. As described above, the only accessed element of the array is the last one. This location is already maintained throughout the updates. In the basic data structure, it is the last occupied element of the last nonempty data block DB_{d-1}. In the buddy data structure, it is the last occupied element of either the last nonempty data block DB_{d-1} or the last nonempty overflow block OB_{s-1}; which of the two can be maintained by monitoring when DB_{d-1} becomes full.

As a result, the Locate algorithm can be avoided in our solution to stacks.

Shrink:

1. Decrement n and the number of elements occupying CB_{d-1}.
2. If CB_{d-1} is empty:
   (a) If there is another empty data block, Deallocate it.
   (b) If the data index block is a quarter full, Reallocate it to half its size.
   (c) Decrement d and the number of data blocks occupying the last superblock SB_{s-1}.
   (d) If SB_{s-1} is empty:
      i. If there is another empty overflow block, Deallocate it.
      ii. If the overflow index block is a quarter full, Reallocate it to half its size.
      iii. Decrement s.
      iv. If s is even, halve the number of data blocks in a superblock.
      v. Otherwise, halve the number of elements in a conceptual block.
      vi. Set the occupancy of SB_{s-1} to full.
   (e) Set the occupancy of DB_{d-1} to full.

Algorithm 5: Buddy implementation of Shrink.

Locate (i):

1. Let r denote the binary representation of $i+1$, with all leading zeros removed.
2. Note that the desired element i is element e of conceptual block b of superblock k, where
   (a) $k = |r| - 1$,
   (b) b is the $\lfloor k/2 \rfloor$ bits of r immediately after the leading 1-bit, and
   (c) e is the last $\lceil k/2 \rceil$ bits of r.
3. Let $j = 2^{\lceil k/2 \rceil}$ be the number of elements in conceptual block b.
4. If $e \geq j - h$, element i is stored in an overflow block:
   Return the location of element $bh + e - (j - h)$ in overflow block OB_k.
5. Otherwise, element i is stored in a data block:
   (a) Let $p = 2^k - 1$ be the number of data blocks in superblocks prior to SB_k.
   (b) Return the location of element e in data block DB_{p+b}.

Algorithm 6: Buddy implementation of Locate.

5.2 Queues

A queue can be implemented with two stacks using a well-known method often described in the functional-languages community (e.g., [18]), discovered independently by several researchers [4, 12, 13]. Call the two stacks the "enqueue" and "dequeue" stacks, respectively. The Enqueue operation always simply Pushes the given element onto the enqueue stack. The Dequeue operation first checks whether the dequeue stack is empty. If so, Dequeue repeatedly Pops an element off the enqueue stack and immediately Pushes it back on the dequeue stack; in other words, it flips over the entire enqueue stack, placing the result on the dequeue stack. Once this process is complete (or if the dequeue stack was originally nonempty), Dequeue Pops an element from the dequeue stack and returns it.

This results in a constant amortized time bound. For a constant worst-case time bound, we use more of the flexibility of singly resizable arrays. As we will see, though, the use of Read and Write will be sufficiently restricted to continue avoiding the use of Locate.

Our method will use two singly resizable arrays, called the enqueue array and the dequeue array. The dequeue array will now have a more complex form. The last element will represent the ith element to delete for some known value i, the element before that will be the $(i-1)$st element to delete, and so on; thus, the ith element from the end will be the first element to delete. Immediately before that element will be the $(i+1)$st element to delete, then the $(i+2)$nd element, and so on; thus, the first element in the array is the last element to delete. In other words, the order of elements in the dequeue array is as follows:

$$n,\ n-1,\ \ldots,\ i+2,\ i+1,\ 1,\ 2,\ \ldots,\ i-1,\ i.$$

The Dequeue operation now Reads the ith element from the end of the dequeue array (returning it later), replaces it with the last element of the dequeue array, shrinks the dequeue array by one, and decrements i if it is more than 1. The dequeue array now has the following order of elements:

$$n,\ n-1,\ \ldots,\ i+2,\ i+1,\ i,\ 2,\ \ldots,\ i-1,$$

which is of the desired form. Note that i will eventually become 1, in which case Dequeue degenerates to a Pop operation.

Two details remain to be mentioned. First, if the dequeue array becomes empty, Dequeue first swaps the enqueue and dequeue arrays, and initializes i to the number of elements. Second, the Enqueue operation is simple: it just adds the given element to the end of the enqueue array. This leads to the following result:

Corollary 2 *Queues can be implemented in $O(1)$ worst-case time per operation, and $O(\sqrt{n})$ worst-case extra storage.*

Note that by keeping a pointer to the ith element from the end of the dequeue array, and incrementing it as we execute Dequeue (which can be easily done in constant time), we can again avoid the use of Locate.

5.3 Randomized Queues

As mentioned above, our original motivating application, randomized queues, can be implemented as follows. Insert is the same as the stack Push operation: it calls Grow and Writes the given value to the last element of the array. DeleteRandom generates a random number i between 0 and $n-1$, calls Read (i), copies the last element of the array to index i, and finally calls Shrink.

Before we can claim exactly what time bound is achieved by this data structure, we need to discuss what facilities for random-number generation are provided. There are three standard operations:

1. Random (n): Returns a random integer between 0 and $n-1$.
2. RandomBit: Returns a random integer between 0 and 1.
3. RandomWord: Returns a random integer between 0 and 2^w-1, where w is the number of bits in a word.

The operation we desire is Random (n). However, often just RandomBit or RandomWord is provided, so it is important to consider how to implement Random (n) using one of them. If RandomBit is the only available operation, then Random-Word can be implemented by calling RandomBit w times, where w is the number of bits in a word. Hence, we can assume that Random-Word is provided, possibly with a running time of $\Theta(w)$.

A common implementation of Random (n) is to call RandomWord and take the result modulo n. However, this is an incorrect implementation, because the choice is biased towards values less than $2^w \bmod n$. This is only suitable when the user allows the randomized queue to have a small skew in the distribution of randomly chosen elements; the probability that a particular element will be chosen is always less than $2/n$, and it approaches the desired $1/n$ as n approaches infinity.

Perhaps the simplest correct implementation of Random (n) is the following standard one [15]. Let m be the smallest power of two that is at least n.[3] Then we call RandomWord, take the result modulo m (which can be done using a bitwise and operation), and repeat until we obtain a number less than n. Because of the randomness of RandomWord, the result is a truly random number between 0 and $n-1$. Because $m < 2n$, the probability that a particular iteration will succeed is greater than $1/2$, so the expected number of iterations is less than 2. Hence, the expected running time is $O(1)$. While the worst-case running time is potentially infinite, this is necessary for any correct implementation of Random based on RandomBit or RandomWord [15].

In conclusion, we have described implementations of Random (n) with running time $O(1)$ expected time if RandomWord is provided, and $\Theta(\log n)$ if RandomBit is provided. Both of these time bounds are optimal under these assumptions. In general, we have the following result:

Corollary 3 *Randomized queues can be implemented in $O(\sqrt{n})$ worst-case extra storage, where Insert takes $O(1)$ worst-case time, and DeleteRandom takes time dominated by the cost of computing a random number between 1 and n.*

5.4 Priority Queues

The heap data structure introduced by Williams [22] is well known for its elegant use of arrays to implement priority queues. Because the n elements of the heap always lie in the first n elements of the array, and the operations simply make $O(\log n)$ accesses to the array, we have the following immediate consequence:

Corollary 4 *Priority queues can be implemented in $O(\log n)$ worst-case time per operation, and $O(\sqrt{n})$ worst-case extra storage.*

Another interesting problem is double-ended heaps. A *double-ended priority queue* supports FindMax and DeleteMax in addition to the usual FindMin, DeleteMin, and Insert operations in the priority queue. There are several so-called “implicit” or “pointerless” data

[3] This value can be computed incrementally in $O(1)$ time per operation, because n only changes by at most 1 during each operation.

structures for double-ended priority queues, with the same properties as heaps (occupying only the first n elements of an array). Some examples are the min-max heap [2], deap [5, 6], and diamond deque [7]. While these data structures use $\Theta(n)$ extra space, we can use our efficient singly resizable arrays to obtain the following result:

Corollary 5 *Double-ended priority queues (which support both* DeleteMin *and* DeleteMax*) can be implemented in $O(\log n)$ worst-case time per operation, and $O(\sqrt{n})$ worst-case extra storage.*

6 Doubly Resizable Arrays and Deques

A natural extension to our results on optimal stacks and queues would be to support deques (double-ended queues). The amortized time bound is easy to extend: instead of flipping a whole stack when the other becomes empty, just flip half of the stack. However, deques cannot be efficiently implemented in the worst case with two stacks, unlike the case of queues. This leads us to examine *doubly resizable arrays* that can grow and shrink on both ends.

The main reason that the data structure presented in Section 4 cannot support doubly resizable arrays is its imbalance. More precisely, the blocks range in size from $\Theta(1)$ to $\Theta(\sqrt{n_0})$ when $n = n_0$, so if we start deleting from the low end (where the blocks are of constant size), we will eventually have a single block of size $\Theta(\sqrt{n_0})$, which means that we are potentially wasting $\Omega(\sqrt{n_0})$ storage even when $n = O(1)$.

Our solution is to keep all the data blocks of roughly the same size. Indeed, we will maintain the invariant that every data block has size

$$s_1 = 2^{\left\lfloor \frac{\log_2(1+n)}{2} \right\rfloor} \quad \text{or} \quad s_2 = 2^{1 + \left\lfloor \frac{\log_2(1+n)}{2} \right\rfloor}$$

for the current value of n. Blocks of the two sizes are called *small* and *large*, respectively. We further maintain the constraint that all the small blocks are consecutive, followed by all the large blocks which are consecutive. Only the first small block and the last large block may be partially empty, and therefore the extra storage is $O(\sqrt{n})$ at all times.

We maintain these properties by dynamically merging and splitting data blocks. As with our description of the singly resizable array data structure, we first give a method which achieves a constant amortized time bound, and then describe modifications to achieve a constant worst-case time bound.

The basic idea is as follows. Before we create a new block in a Grow operation that ran out of room on the requested side, we first merge the last two small blocks (the ones adjacent to the large blocks) into one large block. If there is only one small block, this degenerates to a Reallocate operation; and if there are no small blocks, this degenerates to doing nothing. Next Grow checks whether $\left\lfloor \frac{1}{2}\log_2(1+n) \right\rfloor$ has increased, in which case we double s_1 and s_2. The block is then created, using these new values to determine its size: s_1 for GrowBackward and s_2 for GrowForward. Note that because we merged the two small blocks before changing the size, we will maintain that there are only two block sizes.

Before we mark a block as empty in a Shrink operation that discarded the last element in a block on a particular side, we split the first large block (the one adjacent to the small blocks) into two small blocks. If there are no large blocks, this operation degenerates to doing nothing. Next we check whether $\left\lfloor \frac{1}{2}\log_2(1+n)\right\rfloor$ has decreased, in which case we halve s_1 and s_2. Note that because we split the block before this change, the blocks will all be of small size when this occurs, as desired.

One trick is required in this data structure. We keep a copy of the unmerged or unsplit version of the block(s) we have just merged or split, on both ends of the array. Thus for example, if we decide that we should merge a block that we just split, this can be done in constant time. Similarly, if we decide that this split-then-remerged block should be split again, we revert to the already computed split version. We only waste a constant number of blocks, or $O(\sqrt{n})$ extra storage, to store these extra copies. The extra copies must be maintained when we perform updates to elements at the ends, which only requires an extra constant amount of work in the Write operation.

The Locate operation is quite simple. Because we know the number of small blocks, the size of each small block, and the occupancy of the first small block, we can determine whether the desired element is in the small blocks or the large blocks. Then we can determine which block contains the element by dividing by the block size, and the remainder is the element number within that block.

For a constant worst-case time bound, we need the following modifications to the data structure. During updates to the data structure, we simultaneously work on merging and splitting a constant number of blocks. This approach is similar to that of Section 4.3, where we deamortize the cost of resizing the index block. We work on merging the next pair of small blocks we might merge, during each Grow operation; and similarly work on splitting the next large block we might split, during each Shrink operation.

We have thus proved the following theorem:

Theorem 4 *A doubly resizable array can be implemented using $O(\sqrt{n})$ extra storage in the worst case and $O(1)$ time per operation, on a transdichotomous random access machine where memory is dynamically allocated.*

Note that this data structure avoids finding the leading 1-bit in the binary representation of an integer. Thus, in some cases (e.g., when the machine does not have an instruction finding the leading 1-bit), this data structure may be preferable even for singly resizable arrays.

7 Conclusion

We have presented data structures for the fundamental problems of singly and doubly resizable arrays that are optimal in time and worst-case extra space on realistic machine models. We believe that these are the first theoretical algorithms designed to work in conjunction with the buddy system, which is practical for many modern operating systems including UNIX. They have led to optimal data structures for stacks, queues, priority queues, randomized queues, and deques.

As mentioned in Section 1.3, resizing has traditionally been explored in the context of hash tables. An interesting open question is whether it is possible to implement dynamic hash tables with $o(n)$ extra space.

We stress that our work has focused on making simple, practical algorithms. One of our goals is for these ideas to be incorporated into the C++ standard template library (STL). We leave the task of expressing the randomized queue procedure in a form suitable for first-year undergraduates as an exercise for the fifth author.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, October 1986.

[3] Andrej Brodnik. Computation of the least significant set bit. In *Proceedings of the 2nd Electrotechnical and Computer Science Conference*, Portoroz, Slovenia, 1993.

[4] F. W. Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.

[5] Svante Carlsson. The Deap—a double-ended heap to implement double-ended priority queues. *Information Processing Letters*, 26:33–36, September 1987.

[6] Svante Carlsson, Jingsen Chen, and Thomas Strothotte. A note on the construction of the data structure "Deap". *Information Processing Letters*, 31:315–317, June 1989.

[7] S. C. Chang and M. W. Du. Diamond deque: A simple data structure for priority queues. *Information Processing Letters*, 46:231–237, July 1993.

[8] Erik D. Demaine and J. Ian Munro. Fast allocation and deallocation with an improved buddy system. In *Proceedings of the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 84–96, Chennai, India, December 1999.

[9] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[10] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[11] Michael T. Goodrich and John G. Kloss II. Tiered vector: An efficient dynamic array for JDSL. In *Proceedings of the 1999 Workshop on Algorithms and Data Structures*, Vancouver, Canada, August 1999.

[12] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[13] R. Hood and R. Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, November 1981.

[14] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.

[15] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428. Academic Press, Inc., 1976.

[16] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[17] Rajeev Motwani and Prabhaker Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] Chris Okasaki. Simple and efficient purely functional queues. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[19] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[20] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 3rd edition, 1997.

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[22] J. W. J. Williams. Algorithm 232. *Communications of the ACM*, 7(6):347–348, 1964.