Resizable Arrays in Optimal Time and Space

Andrej Brodnik[1,2], Svante Carlsson[2], Erik D. Demaine[3], J. Ian Munro[3], and Robert Sedgewick[4]

[1] Dept. of Theoretical Computer Science, Institute of Mathematics, Physics, and Mechanics, Jadranska 19, 1111 Ljubljana, Slovenia, `Andrej.Brodnik@IMFM.Uni-Lj.SI`

[2] Dept. of Computer Science and Electrical Engineering, Luleå University of Technology, S-971 87 Luleå, Sweden, `svante@sm.luth.se`

[3] Dept. of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {`eddemaine`, `imunro`}`@uwaterloo.ca`

[4] Dept. of Computer Science, Princeton University, Princeton, NJ 08544, U.S.A., `rs@cs.princeton.edu`

Abstract. We present simple, practical and efficient data structures for the fundamental problem of maintaining a resizable one-dimensional array, $A[l..l+n-1]$, of fixed-size elements, as elements are added to or removed from one or both ends. Our structures also 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 [6, 8, 10]. Specifically, a *randomized queue* maintains a collection of fixed-size elements, such as word-size integers or pointers, and supports 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.

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 [10].

One natural implementation of randomized queues stores the elements in an array and uses the *doubling technique* [3]. 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 $O(n)$ space occupied by this structure is optimal up to a constant factor, but 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, $0 \le i < n$.
2. Write (i, x): Set the element with index i to x, $0 \le i < n$.
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 [11, 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 any constant factor c. 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.

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.

A natural extension is the efficient implementation of a *deque* (or double-ended queue). This leads to the notion of 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.

The rest of this paper is outlined 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 [4] 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 [4] 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

[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 $O(1 + b)$.

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.

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 [6, vol. 1, p. 435], which otherwise rounds to the next power of two. If all the blocks contained user data whose size is a power 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:

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

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 $f(n)$ be the size of the largest memory block, and let $g(n)$ 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, $f(n) \cdot g(n)$ must be at least n.

At the time between the inserts and the deletes, the amount of extra storage is at least $hg(n)$ to store the memory block headers, and hence the worst-case extra storage is at least $g(n)$. Furthermore, at the time immediately after the block of size $f(n)$ was allocated, the extra storage was at least $f(n)$. Hence, the worst-case extra storage is at least $\max\{f(n), g(n)\}$. Because $f(n) \cdot g(n) \geq n$, 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 anywhere. Goodrich and Kloss [5] show that $O(\sqrt{n})$ amortized time suffices for updates, even when access queries must be performed in constant time. They use $O(\sqrt{n})$ extra space, which as we see is optimal.

4 Singly Resizable Arrays

The basic idea of our first data structure is storing 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 $\lceil(\sqrt{1+8n}-1)/2\rceil = \Theta(\sqrt{n})$.

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 [9, 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 4.1).

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. Section 4.2 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

[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 4.1. Here we develop a much cleaner algorithm.

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; 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.

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 Fig. 1.

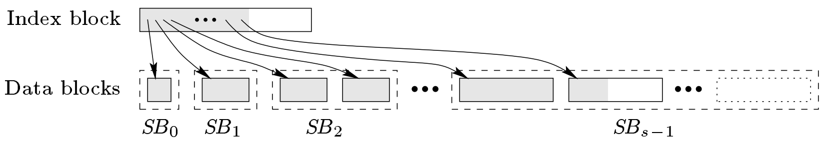

Fig. 1. A generic snapshot of the basic data structure.

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.

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.

In the rest of this section, we show the following theorem:

Theorem 2. *This data structure implements singly resizable arrays 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 $\lceil \log_2(1+n) \rceil$. 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.*

The space bound follows from the following lemmas. See [2] for proofs.

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

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

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

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 [1] 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 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 factor to the extra storage, so it is still $O(\sqrt{n})$. The time per operation is therefore $O(1)$ in the worst case.

4.2 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 [6, vol. 1, p. 540] 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 [7, 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;

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.

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 singly resizable arrays 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.*

5 Applications of Singly Resizable Arrays

This section presents a variety of fundamental abstract data types 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. Please refer to [2] for details of the algorithms.

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

Furthermore, the general Locate operation can be avoided by using the pointer to the last element in the array. Thus the computation of the leading 1-bit is not needed. This result can also be shown or the following data structure by keeping an additional pointer to an element in the middle of the array [2].

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

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.

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.*

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

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). It is easy to achieve an amortized time bound by storing the queue in two stacks, and flipping half of one stack when the other becomes empty. To obtain a worst-case time bound, we use a new data structure that keeps the blocks all roughly the same size (within a factor of 2). By dynamically resizing blocks, we show the following result; see [2] for details.

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.

Resizing has traditionally been explored in the context of hash tables [3]. Knuth traces the idea back at least to Hopgood in 1968 [6, vol. 3, p. 540]. 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. A. Brodnik. Computation of the least significant set bit. In *Proceedings of the 2nd Electrotechnical and Computer Science Conference*, Portoroz, Slovenia, 1993.
2. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. Technical Report CS-99-09, U. Waterloo, 1999.
3. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SICOMP*, 23(4):738–761, Aug. 1994.
4. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *JCSS*, 47(3):424–436, 1993.
5. M. T. Goodrich and J. G. Kloss II. Tiered vector: An efficient dynamic array for JDSL. This volume.
6. D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.
7. M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.
8. R. Motwani and P. Raghavan. *Randomized Algorithms*. Camb. Univ. Press, 1995.
9. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Camb. Univ. Press, 2nd ed., 1992.
10. R. Sedgewick. *Algorithms in C*. Addison-Wesley, 3rd ed., 1997.
11. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd ed., 1997.