Succinct Dynamic Data Structures*

Rajeev Raman[1], Venkatesh Raman[2], S. Srinivasa Rao[2]

[1] Department of Mathematics and Computer Science
University of Leicester, Leicester LE1 7RH, UK.
r.raman@mcs.le.ac.uk.

[2] Institute of Mathematical Sciences, Chennai, India 600 113,
{vraman,ssrao}@imsc.ernet.in

Abstract. We develop succinct data structures to represent (i) a sequence of values to support *partial sum* and *select* queries and *update* (changing values) and (ii) a dynamic array consisting of a sequence of elements which supports *insertion*, *deletion* and *access* of an element at any given index.

For the partial sums problem on n non-negative integers of k bits each, we support *update* operations in $O(b)$ time and *sum* in $O(\log_b n)$ time, for any parameter b, $\lg n/\lg\lg n \leq b \leq n^\epsilon$ for any fixed positive $\epsilon < 1$. The space used is $kn+o(kn)$ bits and the time bounds are optimal. When $b = \lg n/\lg\lg n$ or $k = 1$ (i.e., when we are dealing with a bit-vector), we can also support the *select* operation in the same time as the *sum* operation, but the update time becomes amortized.

For the dynamic array problem, we give two structures both using $o(n)$ bits of extra space where n is the number of elements in the array: one supports lookup in constant worst case time and updates in $O(n^\epsilon)$ worst case time, and the other supports all operations in $O(\lg n/\lg\lg n)$ amortized time. The time bounds of both these structures are optimal.

1 Introduction

Recently there has been a surge of interest in the study of *succinct* data structures [1–3, 9–13]. The aim is to design data structures that are asymptotically optimal with respect to operation times, but whose space usage is optimal to within lower-order additive terms. Barring a few exceptions [2, 13], most of these are static structures. In this paper we look at succinct solutions to two classical interrelated dynamic data structuring problems, namely maintaining *partial sums* and *dynamic arrays*. We assume a RAM model with word size $\Theta(\lg n)$ bits, where n is the input size. In this model, reading and writing $O(\lg n)$ consecutively stored bits, arithmetic and bit-wise boolean operations on $O(\lg n)$-bit operands can be performed in constant time. In more detail, the problems considered are:

* Research supported in part by UK-India Science and Technology Research Fund project number 2001.04/IT and in part by UK EPSRC grant GR L/92150.

Partial Sums This problem has two positive integer parameters, the *item size* $k = O(\lg n)$, and the *maximum increment* $\delta_{max} = \lg^{O(1)} n$. The problem consists in maintaining a sequence of n numbers $A[1], \ldots, A[n]$, such that $0 \le A[i] \le 2^k - 1$ under the operations:

- *sum*(i): return the value $\sum_{j=1}^{i} A[j]$.
- *update*(i, δ): set $A[i] \leftarrow A[i] + \delta$, for some integer δ such that $0 \le A[i] + \delta \le 2^k - 1$ and $|\delta| \le \delta_{max}$.

We also consider adding the following operation:

- *select*(j): find the smallest i such that *sum*$(i) \ge j$.

In what follows, we refer to the partial sums problem with *select* as the *searchable* partial sums problem.

Dietz [4] has given a structure for the partial sums problem that supports *sum* and *update* in $O(\lg n / \lg\lg n)$ worst-case time using $\Theta(n \lg n)$ bits of extra space, for the case $k = \Theta(\lg n)$. As the information-theoretic space lower bound is kn bits, Dietz's data structure uses a constant factor extra space even when $k = \Theta(\lg n)$, and is worse for smaller k. We modify Dietz's structure to obtain a data structure that solves the searchable partial sums problem in $O(\lg n / \lg\lg n)$ worst case time using $kn + o(kn)$ bits of space. Thus, we improve the space utilisation and add the *select* operation as well.

For the partial sums problem we can trade off query and update times as follows: for any parameter $b \ge \lg n / \lg\lg n$ we can support *sum* in $O(\lg_b n)$ time and *update* in $O(b)$ time[1]. The space used is the minimum possible to within a lower-order term.

Our time bounds are optimal in the following sense. Fredman and Saks [5] gave lower bounds for this problem in the *cell probe* model with logarithmic word size, a much stronger model than ours. For the partial sums problem, they show that an intermixed sequence of n updates and queries requires $\Omega(\lg n / \lg\lg n)$ amortized time per operation. Furthermore, they give a more general trade-off [5, Proof of Thm 3′] between the number of memory locations that must be written and read by an intermixed sequence of updates and queries. Our data structure achieves the optimal trade-off between reads and writes, for the above range of parameter values. If we require that queries be performed using read-only access to the data structure—a requirement satisfied by our query algorithms—then the query and update times we achieve are also optimal.

Next, we consider a special case of the searchable partial sums problem that is of particular interest.

Dynamic Bit Vector Given a bit vector of length n, support the following operations:

[1] In the partial sum and bit-vector results, other trade-offs that allow expensive queries and cheap updates are possible. These are mentioned in the appropriate sections.

- $rank(i)$: find the number of 1's occurring before and including the ith bit
- $select(j)$: find the position of jth one in the bit vector and
- $flip(i)$: flip the bit at position i in the bit vector.

A bit vector supporting *rank* and *select* is a fundamental building block for succinct static tree and set representations [2, 11]. Given a (static) bit vector, we can support the *rank* and *select* operations in $O(1)$ time using $o(n)$ bits of extra space [3, 10].

As the dynamic bit vector problem is simply the searchable partial sums problem with $k = 1$, we immediately obtain a data structure that supports *rank*, *select* and *flip* operations in $O(\lg n/\lg\lg n)$ worst case time using $o(n)$ bits of extra space. For the bit vector, however, we are able to give a trade-off for all three operations. Namely, for any parameter $b \geq \lg n/\lg\lg n$ we can support *rank* and *select* in $O(\lg_b n)$ time and *update* in amortised $O(b)$ time. In particular, we can support *rank* and *select* in constant time if we allow updates to take $O(n^\epsilon)$ amortised time for any constant $\epsilon > 0$.

If we remove the *select* operation from the dynamic bit vector problem, we obtain the *subset rank* problem considered by Fredman and Saks [5]. From their lower bound on the subset rank problem, we conclude that our time bounds are optimal, in the sense described above.

Next we consider another fundamental problem addressed by Fredman and Saks [5].

Dynamic Array Given an initially empty sequence of records, support the following operations:

- $insert(x, i)$: insert a new record x at position i in the sequence
- $delete(i)$: delete the record at position i in the sequence and
- $index(i)$: return the ith record in the sequence.

Dynamic arrays are useful data structures in efficiently implementing the data types such as the Vector class in Java and C++. The dynamic array problem was called the *List Representation* problem by Fredman and Saks, who gave a cell probe lower bound of $\Omega(\lg n/\lg\lg n)$ time for this problem, and also showed that $n^{\Omega(1)}$ update time is needed to support constant-time queries. For this problem, Goodrich and Kloss [8] obtained a structure that supports *insert* and *delete* operations in $O(n^\epsilon)$ amortized time while supporting the *index* operation in $O(1/\epsilon)$ worst case time. This structure uses $O(n^{1-\epsilon})$ words of extra space, (besides the space required to store the n elements of the array) for any fixed ϵ, $0 < \epsilon < 1$. Here n is the size of the current sequence.

We first observe that the structure of Goodrich and Kloss can be viewed as a version of the well-known implicit data structure: the 'rotated list' [7]. Using this connection, we observe that the structure of Goodrich and Kloss can be made to take $O(n^\epsilon)$ worst case time for updates while maintaining the same storage ($o(n)$ additional words) and $O(1)$ worst case time for the *index* operation. Then using this structure in small blocks, we obtain a dynamic array structure that supports *insert*, *delete* and *index* operations in $O(\lg n/\lg\lg n)$ amortized time using $o(n)$ bits of extra space. Due to the lower bound result of Fredman and Saks, both our results above are on optimal points of the query time-update time trade-off while using optimal (within lower order term) amount of extra space.

We should also point out that the resizable arrays of Brodnik et al. [1] can be used to support inserting and deleting elements at either ends of an array and accessing the ith element in the array, all in constant time. The data structure uses $O(\sqrt{n})$ words of extra space if n is the current size of the array. Brodnik et al. do not support insertion into the middle of the array.

For the dynamic array problem, we assume a memory model in which the system returns a pointer to the beginning of a block of requested size. I.e. any element in a block of memory can be accessed in constant time given the block pointer and an integer index into the block. This is the same model used by Brodnik et al. [1]. We count the time as the number of word operations, and space as the number of bits used to store the data structure. To simplify notation, we ignore rounding as it does not affect our asymptotic analysis.

In Section 2, we describe our space efficient structures for the partial sum problem. In Section 3, we look at the special case of the partial sum problem when the given elements are bits, and give the details of a structure that supports full tradeoff between queries (*select* and *rank*) and update (*flip*). Section 4 addresses the problem of supporting the dynamic array operations. We conclude with some open problems in Section 5.

2 Partial Sums

2.1 Searchable Partial Sums

In this section we describe a structure for the searchable partial sums problem that supports all operations in $O(\lg n/\lg\lg n)$ time. The space used is $kn+o(kn)$ bits. We begin by solving the problem on a small set of integers in $O(1)$ time, by adapting an idea of Dietz [4].

Lemma 1. *On a RAM with a word size of w bits, we can solve the searchable partial sum problem on a sequence of $m = w^\epsilon$ numbers, for any fixed $0 \le \epsilon < 1$, with item size $k \le w$, in $O(1)$ worst-case time and using $O(mw)$ bits of space. The data structure requires a precomputed table of size $O(2^{\epsilon' w})$ for any fixed $\epsilon' > 0$.*

Proof. Let $A[1], \ldots, A[m]$ denote the sequence of elements for which the partial sums are to be calculated. We store another array $B[1], \ldots, B[m]$ which contains the partial sums of A, i.e. $B[i] = \sum_{j=1}^{i} A[i]$. As we cannot hope to maintain B under the *update* operation, we use Dietz's idea of letting B get slightly 'out of date'. More precisely, B is not changed after each *update*; instead, after every m *update*s B will be *refreshed*, or brought up to date. Since the cost of refreshing is $O(m)$, the amortized cost is $O(1)$ per *update*.

To answer queries, we maintain an array $C[1], \ldots, C[m]$ in addition to A and B. C is set to all zeros when B is refreshed. Otherwise, when an *update* changes $A[i]$ by δ, we set $C[i] \leftarrow C[i]+\delta$. Since $|C[i]| \leq m\delta_{max} = w^{O(1)}$ always, the entire array C occupies $O(m \lg w)$ bits, which is less than $\epsilon' w$ bits for sufficiently large w. As observed by Dietz, $sum(i)$ can be computed by adding to $B[i]$ a corrective term obtained by using C to index into a pre-computed table.

We now show how to perform *select* in $O(1)$ time. For this, we use the *Q-heap* structure given by Fredman and Willard [6], which solves the following dynamic predecessor problem:

Theorem 1. *[6] For any $0 < M < 2^w$, given a set of at most $(\lg M)^{1/4}$ integers of $O(w)$ bits each, one can support the operations insert, delete, predecessor and successor operations in constant time where predecessor(x) (successor(x)) returns the largest (smallest) element y in the set such that $y < x$ ($y \geq x$). The data structure requires a precomputed table of size $O(M)$.*

By choosing $M = 2^{\epsilon' w}$, we can do predecessor queries on sets of size $m' = w^{\epsilon'/4}$ in $O(1)$ time, using a table of size $O(M)$. By using this data structure in a tree with branching factor m', we can support $O(1)$-time operations on sets of size m as well. We store the elements of B in the Q-heap data structure. Note that changes to the Q-heap caused by refreshing have $O(1)$ amortised cost.

If the array B were up-to-date, then we can answer *select* queries in $O(1)$ time by finding the successor of j in the Q-heap. However, again we face the problem that B may not be up-to-date. To overcome this, we let $D[1], \ldots, D[m]$ be an array where $D[i] = \min\{A[i], m\delta_{max}\}$. As with C, D is also stored in a single word, and is changed in $O(1)$ time (using either table lookup or bitwise operations) whenever A is changed by an *update*. To implement *select*(j), we first consult the Q-heap data structure to determine an index t such that $B[t-1] < j \leq B[t]$. By calculating $sum(t-1)$ and $sum(t)$ in $O(1)$ time we determine whether t is the correct answer. In general, the correct answer would be an index $t' \neq t$; assume for specificity that $t' > t$. Note that t' is the smallest integer such that $A[t+1]+\cdots+A[t'] \geq j - sum(t)$. Since $j \leq B[t]$ and $B[t] - sum(t) \leq m\delta_{max}$, it follows that $j - sum(t) \leq m\delta_{max}$. By the definition of D, it also follows that t' is the smallest integer such that $D[t+1] + \ldots + D[t'] \geq j - sum(t)$. Given D, $j - sum(t)$ and t, one can look up a table to calculate t' in $O(1)$ time. A similar procedure is followed if $t' < t$.

Finally, we note that amortization can be eliminated by Dietz's incremental refreshing approach [4]. □

For larger inputs, we choose a parameter $m = (\lg n)^\epsilon$, for some positive constant $\epsilon < 1$. We create a complete m-ary tree, the leaves of which correspond to the entries of the input sequence A. We define the *weight* of a node (leaf or internal) as the sum of the sequence elements under it. At each internal node we store the weights of its children in the data structure of Lemma 1. The tree has $O(n/m)$ internal nodes, each occupying $O(m)$ words and supporting all operations in constant time. From this we get:

Lemma 2. *There is a data structure for the searchable partial sums problem that supports all operations in $O(\lg n/\lg\lg n)$, and requires $O(n)$ words of space.*

We now modify this structure reducing the space complexity of the structure to $kn + o(kn)$ bits. For this, we need the following:

Lemma 3. *For any parameter $b \geq 4$, there is a data structure for the searchable partial sums problem that supports update in $O(\log_b n)$ time and sum and search in $O(b \log_b n)$ time. The space used is $kn + O(((k + \lg b) \cdot n)/b)$ bits.*

Proof. We construct a complete b-ary tree over the elements of the input sequence A. At each internal node we store the sum of the elements of A under it. Clearly *update* takes time proportional to the height of the tree, and *sum* and *select* can be implemented within the claimed bounds by traversing the tree from the root to a leaf, looking at all the children of a node at each level. The space bounds follow after a straightforward calculation. $\square$

We take the input and divide it into groups of numbers of size $(\lg n)^2$ each. The groups are represented internally using Lemma 3, with $b = (\lg n)^{1/2}$. This requires $kn+o(kn)$ bits, and all operations within a group take $O((\lg n)^{1/2})$ time, which is negligible. The $n/(\lg n)^2$ group sums are stored in the data structure of Lemma 2, which requires $o(n)$ bits now. The precomputed tables (required in Lemma 1) also require $o(n)$ bits. Thus we have:

Theorem 2. *There is a data structure for the searchable partial sums problem that supports all operations in $O(\lg n/\lg\lg n)$ worst-case time and uses $kn+o(kn)$ bits of space.*

2.2 Trade-offs for Partial Sums

We now observe that one can trade off query and update times for the partial sums problem, and show that for any parameter $2 \leq b \leq n$, we can support *sum* in $O(\log_b n)$ and *update* in $O(b \log_b n)$ time, while still ensuring that the data structure is space-efficient. As these bounds are subsumed by Theorem 2 for $b \leq (\lg n)^2$, we will assume that $b > (\lg n)^2$ in what follows.

We construct a complete tree with branching factor b, with the given sequence of n elements at the leaves. Clearly this tree has height $h = \log_b n$. At each internal node, we store the *weight* of that node, i.e. the sum of the leaves descended from it, and also store an array containing the partial sums of the weights of all its children. By using the obvious $O(b)$ time algorithm, the partial sum array at an internal node is kept up-to-date after each *update*. This gives running times of $O(b \lg_b n)$ and $O(\lg_b n)$ for *update* and *sum* respectively. Unfortunately, the space used to store this 'simple' structure is $O(kn)$ bits.

To get around this, we use one of two methods, depending on the value of k. If $k \geq (\lg n)^{1/2}$ then we divide the input values into groups of size $\lg n$. Within a group, we do not store the $A[i]$'s explicitly, but store only their partial sums. The sums of elements in each of the $n/\lg n$ groups are stored in the simple structure above, but the space required by that data structure is now $O((k + \lg\lg n)n/\lg n)$ (as the size of each sum could be $k + \lg\lg n$) which is $o(kn)$ bits. The space required by each group is $\lg n(k + \lg\lg n)$ bits; this sums up to $kn + n\lg\lg n = kn + o(kn)$ bits overall. Clearly the asymptotic complexity of *update* and *sum* are not affected by this change.

If $k < (\lg n)^{1/2}$ then we divide the given sequence of elements into groups of $\epsilon \lg n/k$ each. Again, group sums are stored in the simple structure, which requires $O(kn(k+\lg\lg n)/\lg n) = o(kn)$ bits. Noting that an entire group requires $\epsilon \lg n$ bits, we answer *sum* queries within a group by table lookup.

Finally, noting that given any parameter $b \geq (\lg n)^2$, we can reduce the branching factor from b to $b/\lg n$ without affecting the complexity of *sum*; however, *update* would now take $O(b)$ steps. Combining this with Theorem 2 we have:

Theorem 3. *For any parameter* $\lg n/\lg\lg n \leq b < n$*, there is a data structure for the partial sums problem that supports sum in* $O(\lg_b n)$ *time and update in* $O(b)$ *time, and uses* $kn + o(kn)$ *bits of space.*

Remark 1. Note that Lemma 3 combined with Theorem 2 also gives a trade-off whereby *update* takes $O(\log_b n)$ and *sum* takes $O(b)$ time, for any $b \geq \lg n/\lg\lg n$.

3 Dynamic Bit Vector

The dynamic bit vector problem operation is a special case of the searchable partial sum problem. The following corollary follows from Theorem 2.

Corollary 1. *Given a bit vector of length* n*, we can support the rank, select and flip operations in* $O(\lg n/\lg\lg n)$ *time using* $o(n)$ *bits of space in addition to the bit vector.*

Similarly, Theorem 3 immediately implies the following result (the only thing to observe is that of the two cases in Theorem 3, we apply the one that stores the input sequence explicitly):

Corollary 2. *For any parameter* $\lg n/\lg\lg n \leq b < n$*, there is a data structure for the dynamic bit vector problem that supports rank in* $O(\lg_b n)$ *time and flip in* $O(b)$ *time, using* $o(n)$ *bits of space in addition to the bit vector.*

3.1 Trade-off between *query* and *update* times

In this section we show that the trade-off's between *sum* and *update* for partial sums established in Section 2.2, also hold between *select* and *update* for the special case of the dynamic bit vector problem. We first note the following proposition.

Lemma 4. *The operations select and $flip$ can be supported in $O(1)$ time on a bit-vector of size $N = (\lg n)^{O(1)}$ on a RAM with word size $O(\lg n)$, using a fixed pre-computed table of size $O(n^\epsilon)$ bits for some constant $\epsilon < 1$. The space required is $o(N)$ bits in addition to the pre-computed table and the bit-vector itself.*

Proof. Simply store the values in a balanced tree with branching factor $\sqrt{\lg n}$, and stop the tree at the level when the number of leaves at the subtree rooted at the nodes is about $(\lg n)/2$. With each internal node, we keep the searchable structure of Lemma 1. At the leaf level, we will use a precomputed table to support flip and select in constant time.

Since the height of the tree is a constant, select and flip can be supported in constant time. The space used is $o(N)$ besides the $O(n^\epsilon)$ bits required for the precomputed table. □

We now show how to support *select* in $O(\lg_b n)$ time if $flip$ takes $O(b)$ time, for any parameter $(\lg n)^4 \leq b \leq n$. We divide the bit vector into *superblocks* of size $(\lg n)^4$. With each superblock we store the number of ones in it. The sequence of superblock counts is stored in the data structure of Theorem 3 with the same value of b. This enables us, $O(\lg_b n)$ time, to look up the number of ones to the left of any given superblock. We store each of the superblocks using the structure of Lemma 4. The space required is $o(n)$ bits.

In addition, we divide the ones in the bit vector into groups of $\Theta((\lg n)^2)$ successive ones each. A group's size varies between $0.5(\lg n)^2$ and $2(\lg n)^2$. We construct a weight-balanced B-tree (WBB tree) in the sense of Dietz [4], each leaf of which corresponds to a group leader. Roughly speaking, the branching of this tree is b. Some small modifications need to be made to Dietz's balance conditions: for example, the weight of an internal node needs to be redefined to be the sum of the sizes of the groups under it (we omit details of the WBB tree in this abstract). Given an integer j, using Dietz's ideas we can locate the group in which the j-th one lies in $O(\lg_b n)$ time, and support changes due to $flip$s in $O(b)$ amortised time.

With each group we store the index of the superblock in which the group's leader lies. Equally, with each superblock, we store all group leaders which lie in that superblock.

The *span* of a group is the index of the superblock in which the next group's leader lies minus the index of the superblock in which its group leader lies. If the span of a group is ≥ 2 we say it is *sparse*. With each sparse group, we store an array which gives the location of each 1 in the group. Since the maximum size of this array is $O((\lg n)^2)$, using either the implementation of Goodrich and Kloss or the implementation of Theorem 5, we get a dynamic array which allows insertions and deletions in $O(\lg n)$ time and accesses in $O(1)$ time. This requires $O((\lg n)^3)$ bits per sparse group, but there can only be $O(n/(\lg n)^4)$ sparse groups, so the total space used here is $O(n/\lg n)$.

To execute *select*(j) we first locate the group in which the j-th one lies in $O(\lg_b n)$ time, spending $O(1)$ time at each level of the tree. If this group is sparse then we look up the array associated with the group in $O(1)$ time. Otherwise, we look in the superblock in which the group leader lies, as well as the adjacent superblock, where we can answer the query in $O(1)$ time each by Lemma 4

To set bit j to 1, we first locate the group to which position j would belong. This is easy given the space bounds: recall that via *rank* and *select* one can move from the $i+1$st one bit to the ith one bit in $O(1)$ time. As we move along this "virtual linked list" of 1 bits, we check to see if we have reached a group leader (by looking to see if it is listed among the group leaders in the current superblock). Having thus located the group leader in poly-log (negligible) time, we then either insert into the appropriate array (if the group is sparse) and also into the data structure associated with the superblock. Group splits and merges are handled straightforwardly. Again for $b \geq \lg^4 n$, we can actually make the branching factor to be $b/\lg n$. For other values of b, using Corollaries 1 and 2, we have:

Theorem 4. *Given a bit vector of length n, we can support the rank and select operations in $O(\lg_b n)$ time and $flip$ in $O(b)$ amortised time for any parameter b, $b \geq \lg n/\lg\lg n$ using $o(n)$ bits of extra space.*

Remark 2. Note that one can also get a trade-off similar to that of Remark 1 whereby $flip$ takes $O(\log_b n)$ and *rank/select* takes $O(b)$ time, for any $b \geq \lg n/\lg\lg n$, using $o(n)$ bits of extra space.

4 Dynamic Arrays

We look at the problem of maintaining an array structure under the operations of insertion, deletion and indexing. Goodrich and Kloss [8] have given a structure that supports (arbitrary) *insert* and *delete* operations in $O(n^\epsilon)$ amortized time and *index* operation in $O(1)$ worst case time using $o(n)$ bits of extra space to store a sequence of n elements. Here, first we describe a structure that essentially achieves the same bounds above (except that we can now support updates in $O(n^\epsilon)$ worst case time) using a well known implicit data structure called *recursively rotated list* [7]. Using this as a basic block, we will give a structure that supports all the dynamic array operations in $O(\lg n/\lg\lg n)$ amortized time using $o(n)$ bits of extra space.

We assume a memory model in which the system returns a pointer to the beginning of a block of requested size and hence any element in a block of memory can be accessed in constant time given its index within the block and the block pointer. This is the same model used in the resizable array of Brodnik et al. [1].

Rotated lists were discovered to support dictionary operations implicitly, on a totally ordered set. A (1-level) rotated list is an arbitrary cyclic shift of the sorted order of the given list. We can search for an element in a rotated list on n elements in $O(\lg n)$ time by a modified binary search, though updates (replacing one value with another) can take $O(n)$ time. However, replacing the largest (smallest) element with an element smaller (larger) than the smallest (largest) can be done in $O(1)$ time if we know the position of the smallest element in the list. A 2-level rotated list consists of elements stored in an array divided into blocks where the i-th block is a rotated list of i elements. It is easy to see that such a structure containing n elements has $r = O(\sqrt{n})$ blocks. In addition, all the elements of block i are less than every element of block $i+1$, for $1 \leq i < r$.

This structure supports searches in $O(\lg n)$ time and updates in $O(\sqrt{n})$ time, if we also explicitly store the position of the smallest element in each block (otherwise the updates take $O(\sqrt{n} \lg n)$ time). This is easily generalized to an l-level rotated list where searches take $O(2^l \lg n)$ time and updates take $O(2^l n^{1/l})$ time. See [7] for details.

To use this structure to implement a dynamic array, we do the following. We simply store the elements of the array in a rotated list based on their order of insertions. We also keep the position of the first element in each recursive block. Since we know the size of each block, $index(i)$ operation just takes $O(l)$ time in an l-level rotated list implementation of a dynamic array. Similarly inserting/deleting at position i can be done in a similar fashion as in a rotated list taking $O(2^l n^{1/l})$ time. Thus we have,

Theorem 5. *A dynamic array having n elements can be implemented using an l-level rotated list such that queries can be supported in $O(l)$ time and updates in $O(2^l n^{1/l})$ time using an extra space of $O(n^{1-1/l})$ pointers.*

Choosing l to be a small constant, we get

Corollary 3. *A dynamic array containing n elements can be implemented to support queries in constant time, and updates in $O(n^{\epsilon})$ time using $O(n^{1-\epsilon})$ pointers, where ϵ is any fixed positive constant.*

Using this structure, we now describe a structure that supports all the dynamic array operations in $O(\lg n / \lg \lg n)$ amortized time using $o(n)$ bits of extra space.

We divide the given list of length n into sub lists of length $\Theta(\lg^4 n)$. In particular, each sub-list will be of length between $\frac{1}{2} \lg^4 n$ and $2 \lg^4 n$. (We implement these leaves using the dynamic array structure of Theorem 5.) We construct a weight-balanced B-tree (WBB tree) in the sense of Dietz [4], each leaf of which corresponds to a sub-list. Some small modifications need to be made to Dietz's balance conditions: for example, the weight of an internal node needs to be redefined to be the sum of the sizes of the sub-lists under it. The space required to store this tree is $o(n)$ bits. Supporting insert and delete in $O(\lg n / \lg \lg n)$ amortized time is done as in [4]. To find the jth element of the list, we first find the block in which the jth element occurs, using the $select(j)$ operation on the WBB tree and then find the required element in that block. Thus we have:

Theorem 6. *A dynamic array can be implemented using $o(n)$ bits of extra space besides the space used to store the n records, in which all the operations can be supported in $O(\lg n / \lg \lg n)$ amortized time, where n is the current size of the array.*

5 Conclusions

We have given a succinct searchable partial sum data structure where *sum*, *select* and *update* can be supported in optimal $O(\lg n/\lg\lg n)$ time. We have also given structures in which *sum* can be supported in $(\lg_b n)$ time and *update* in $O(b)$ time for any $b \geq \lg n/\lg\lg n$. These tradeoffs also hold between *select*/*rank* and *update* (flip) for the dynamic bit vector problem. These structures use at most $o(n)$ extra words than necessary.

For the dynamic array, we have given two structures, both using $o(n)$ bits of extra space where n is the number of elements in the array: one supports lookup in constant worst case time and updates in $O(n^{\epsilon})$ worst-case time, and the other supports all operations in $O(\lg n/\lg\lg n)$ amortized time.

The following problems remain open:

1. In the searchable partial sums problem, we were able to support select in $O(\lg_b n)$ time and update in $O(b)$ time using $o(kn)$ bits for the special case of $k = 1$. When is this trade-off achievable in general?
2. For the dynamic array problem, are there tradeoffs (both upper and lower bounds) similar to those in the partial sum problem between query and update operations? In particular is there a structure where updates can be made in $O(1)$ time and access in $O(n^{\epsilon})$ time?
3. Another related problem looked at by Dietz, and Fredman and Saks is the List indexing problem which is like the dynamic array problem, but adds the operation *position*(x), which gives the position of item x in the sequence, and also modifies *insert* to insert a new element after an existing one. Dietz has given a structure for this problem that takes $O(n)$ extra words and supports all the operations in the optimal $O(\lg n/\lg\lg n)$ time. It is not clear that one can reduce the space requirement to just $o(n)$ extra words, and still support the operations in optimal time.

References

1. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro and R. Sedgewick, "Resizable Arrays in Optimal Time and Space", *Proceedings of Workshop on Algorithms and Data Structures*, **LNCS 1663**, 37-48 (1999).
2. A. Brodnik and J. I. Munro, "Membership in Constant Time and Almost Minimum Space", *SIAM Journal on Computing*, **28(5)**, 1628-1640 (1999).
3. D. R. Clark, "Compact Pat Trees", Ph.D. Thesis, University of Waterloo, 1996.
4. Paul F. Dietz, "Optimal Algorithms for List Indexing and Subset Rank", *Proceedings of Workshop on Algorithms and Data Structures*, **LNCS 382**, 39-46 (1989).
5. M. L. Fredman and M. Saks, "The Cell Probe Complexity of Dynamic Data Structures", *Proceedings of the* 21^{st} *ACM Symposium on Theory of Computing*, 345-354 (1989).
6. M. L. Fredman and D. E. Willard, "Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths", *Journal of Computer Systems Science*, **48**, 533-551 (1994).

7. G. N. Fredrickson, “Implicit Data Structures for the Dictionary Problem”, *Journal of Association of the Computing Machinery*, **30**, 80-94 (1983).
8. M. T. Goodrich and J. G. Kloss II, “Tiered Vectors: Efficient Dynamic Array for JDSL”, *Proceedings of Workshop on Algorithms and Data Structures*, **LNCS 1663**, 205-216 (1999).
9. R. Grossi and J. S. Vitter, “Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching”, *Proceedings of Symposium on Theory of Computing*, 397-406 (2000).
10. G. Jacobson, “Space Efficient Static Trees and Graphs”, *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 549-554 (1989).
11. J. I. Munro and V. Raman, “Succinct representation of balanced parentheses, static trees and planar graphs”, *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1997) 118-126.
12. J. I. Munro, V. Raman and S. S. Rao, “Space Efficient Suffix trees”, *Proceedings of the conference on Foundations of Software Technology and Theoretical Computer Science*, **LNCS 1530**, 186-196 (1998).
13. J. I. Munro, V. Raman and A. Storm, “Representing Dynamic Binary Trees Succinctly”, *Proceedings of Symposium on Discrete Algorithms*, 529-536 (2001).