Purely Functional Random-Access Lists

Chris Okasaki
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, Pennsylvania, USA 15213
(e-mail: cokasaki@cs.cmu.edu)

Abstract

We present a new data structure, called a *random-access list*, that supports array *lookup* and *update* operations in $O(\log n)$ time, while simultaneously providing $O(1)$ time list operations (*cons*, *head*, *tail*). A closer analysis of the array operations improves the bound to $O(\min\{i, \log n\})$ in the worst case and $O(\log i)$ in the expected case, where i is the index of the desired element. Empirical evidence suggests that this data structure should be quite efficient in practice.

1 Introduction

Lists are the primary data structure in every functional programmer's toolbox. They are simple, convenient, and usually quite efficient. The main drawback of lists is that accessing the ith element requires $O(i)$ time. In such situations, functional programmers often find themselves longing for the efficient random access of arrays. Unfortunately, arrays can be quite awkward to implement in a functional setting, where previous versions of the array must be available even after an update.

Since arrays are such an important data structure, tremendous effort has been invested in integrating arrays into purely functional languages. In recent years, however, most of this effort has focused on analyses and language restrictions to make destructive array updates safe. Crucial to these efforts is the assumption that arrays are typically used in a *single-threaded* manner, that is, after every update, the old array becomes garbage and may safely be modified in-place to yield the new array.

But what happens when this assumption is violated? Or what if we wish to describe arrays at the user level of a purely functional language, as opposed to the implementation level, so that destructive updates are not available?

In this paper, we present a new data structure, called a *random-access list*, that supports non-single-threaded array lookups and updates in $O(\log n)$ time.[1] This data structure is purely functional and is quite simple to implement in virtually any functional language.

The novel aspect of this data structure is that it provides random access *while simultaneously supporting all normal list primitives in $O(1)$ time*. Thus, random-access lists may be substituted for standard lists in any application with no loss of efficiency (up to a small constant factor).

We begin by describing random access for functional arrays represented first as complete binary trees, and then as collections of trees. We then show how to add or remove a single element from the front of such a collection in $O(1)$ time, providing the basis for efficient list operations. Next, we describe several variations on random-access lists, such as random-access queues and deques, and show how to support *min* queries in a random-access structure with no penalty. We then discuss related work and present experimental comparisons of our data structure with several other purely functional data structures. We conclude with some brief remarks.

All code fragments in this paper are written in Standard ML [23], but are easily adaptable to any other functional language. All source code is available via anonymous `ftp` from `ftp.cs.cmu.edu`, in directory `/afs/cs/project/fox/ftp` and file `randomaccess.tar.gz`.

2 Complete Binary Trees in Preorder

We begin by focusing on the issue of random access. We wish to implement purely functional arrays supporting the following operations:

- *lookup A i*: Return the ith element of A. If i is not a valid index, raise the *Subscript* exception.
- *update A i x*: Return a new array identical to A except that the ith element has been changed to x. If i is not a valid index, raise the *Subscript* exception.

(For now, we ignore the question of how arrays are created.)

How might such a data structure be represented? The obvious first approach is to use balanced binary trees. Any balanced binary tree representation will do (*e.g.*, AVL trees or red-black trees), but we adopt the rather extreme position of representing arrays as *complete* binary trees. Recall that complete binary trees are totally balanced — every internal node has two children of exactly the same size and depth. Unfortunately, complete binary trees are only available in sizes of the form $2^k - 1$, but we will deal with this problem shortly.

Once the shape of the tree has been determined, we must next decide on what order to store the elements in the tree. Typical balanced search trees, such as AVL trees or red-black trees, store the elements in symmetric order, that is, in the order of a left-to-right inorder traversal. However, we reject this choice on the grounds that the first element is stored in the leftmost node and requires $O(\log n)$ time

[1] All logarithms in this paper are base 2.

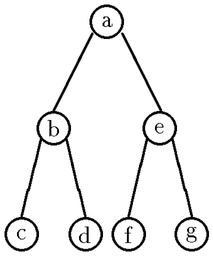

Figure 1: An array represented as a complete binary tree in preorder.

to access. Since we will eventually want to access the first element in $O(1)$ time (to support efficient list operations), we instead choose to store the elements in *preorder*, that is, in the order of a left-to-right preorder traversal (see Figure 1). In this scheme, the first element (element 0) is stored at the root. Elements $1 \ldots \lfloor n/2 \rfloor$ are stored in the left subtree, and elements $\lceil n/2 \rceil \ldots n-1$ in the right subtree.[2]

Given this organization, it is quite simple to write lookup and update functions, as shown in Figure 2. Note that each function takes the size of the tree as an extra parameter. Since the height of a complete binary tree is logarithmic in the size of the tree, the running time of each function is clearly $O(\log n)$.

3 Collections of Trees

As noted in the previous section, complete trees only come in sizes of the form $2^k - 1$, so how do you represent an array of arbitrary size? One rather unsatisfying approach is to simply use the smallest complete tree that is sufficiently large. However, up to half of the elements might be wasted in this scheme. An alternative approach, adopted here, is to represent an array of arbitrary size as a *collection* of complete trees that sum to the appropriate size (see Figure 3). Clearly, this is always possible since you could have a collection of n trees of size 1. However, we show that in fact one can always represent an arbitrary array using only a logarithmic number of trees.

We first make several definitions. A *skew-binary term* is an integer of the form $2^k - 1$ for $k > 0$. Note that if t is a skew-binary term, then $2t+1$ is the next larger skew-binary term. A *skew-binary decomposition* of a integer n is a multiset T of skew-binary terms $\{t_1, \ldots, t_m\}$ such that $n = t_1 + \cdots + t_m$. We call such a decomposition *greedy* if the largest term is as large as possible and if the remainder of the decomposition is also greedy. More formally, a decomposition T is greedy if no subset of terms sums to $\geq$ a larger term, *i.e.*,

$$\forall S \subseteq T : \sum_{t \in S} t < 2(\max S) + 1$$

Call this the *subset-sum restriction.*

Greedy skew-binary decompositions have several useful properties, including:

[2] Braun trees [19, 28] also store the first element at the root, but alternate the remaining elements between the left and right subtrees.

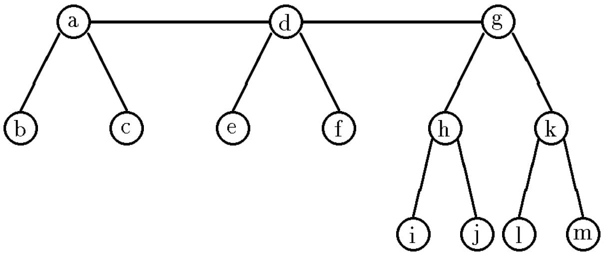

Figure 3: An array represented as a collection of complete binary trees.

Property 1 *Every integer $n \geq 0$ has a unique greedy decomposition, denoted $G(n)$.*

Proof: We give a constructive proof (*i.e.*, an algortihm). Clearly, $G(0) = \{\}$, and this is unique. Now, assume that $n > 0$ and consider the largest skew-binary term $t \leq n$. $G(n)$ must include t since no set of smaller terms may sum to $\geq t$. The remainder of $G(n)$ must sum to $n - t$, and must obey the subset-sum restriction. But this is just $G(n-t)$! Finally, note that no subset of $\{t\} \uplus G(n-t)$ may sum to a term larger than t since otherwise t would not be the largest term $\leq n$. Thus, $G(n) = \{t\} \uplus G(n-t)$. Clearly, this algorithm is deterministic and always terminates. □

Property 2 *A skew-binary decomposition is greedy if and only if every term in the decomposition is unique, except that the smallest two may be the same. In other words, if $t_1, \ldots, t_m$ are the (sorted) terms of a skew-binary decomposition T, then T is greedy if and only if*

$$t_1 \leq t_2 < \cdots < t_m$$

Proof: ($\Rightarrow$) Suppose in a greedy decomposition there existed two terms of the form $2^k - 1$, and another term of the same or lesser size. Then these three terms would violate the subset-sum restriction because they would sum to $\geq 2^{k+1} - 1$.

($\Leftarrow$) By induction on k, no collection of terms in which only the smallest is repeated can possibly sum to a larger term. Thus, the decomposition is greedy. □

Property 3 $|G(n)| \leq \lceil \log(n+1) \rceil$

Proof: By inspection for $n = 0$. Now, suppose $n > 0$ and $G(n)$ contains more than $k = \lceil \log(n+1) \rceil$ terms. By Property 2, it must contain a term at least as large as $2^k - 1 \geq n$. When combined with any of the remaining k or more terms, this strictly exceeds n. □

Now, to represent an array of size n, we simply choose tree sizes according to the greedy decomposition of n. Property 3 guarantees that lookups and updates on a collection of trees still require only $O(\log n)$ time: $O(\log n)$ to find the right tree in the collection, and $O(\log n)$ to find the right element in the tree. Code for these functions is presented in Figure 4.

4 Random-Access Lists

We have to this point ignored the issue of array creation. Typically, the size of an array is specified at the time of creation and is never changed. However, in many situations,

```
datatype 'a tree = Leaf of 'a | Node of 'a * 'a tree * 'a tree

exception Subscript

fun tree_lookup size (Leaf x) 0 = x
  | tree_lookup size (Leaf x) i = raise Subscript
  | tree_lookup size (Node (x,t1,t2)) 0 = x
  | tree_lookup size (Node (x,t1,t2)) i =
        let val size' = size div 2
        in
            if i <= size' then tree_lookup size' t1 (i-1)
                          else tree_lookup size' t2 (i-1-size')
        end

fun tree_update size (Leaf x) 0 y = Leaf y
  | tree_update size (Leaf x) i y = raise Subscript
  | tree_update size (Node (x,t1,t2)) 0 y = Node (y,t1,t2)
  | tree_update size (Node (x,t1,t2)) i y =
        let val size' = size div 2
        in
            if i <= size' then Node (x,tree_update size' t1 (i-1) y,t2)
                          else Node (x,t1,tree_update size' t2 (i-1-size') y)
        end
```

Figure 2: Lookup and update operations on complete binary trees.

```
type 'a functional_array = (int * 'a tree) list

fun lookup [] i = raise Subscript
  | lookup ((size,t) :: rest) i = if i < size then tree_lookup size t i
                                              else lookup rest (i-size)

fun update [] i y = raise Subscript
  | update ((size,t) :: rest) i y = if i < size then (size,tree_update size t i y) :: rest
                                                else (size,t) :: update rest (i-size) y
```

Figure 4: Lookup and update operations on collections of trees. Each tree is paired with its size.

the desired size is not known ahead of time, making fixed-size arrays awkward. Dijkstra [13] instead proposed a system of flexible arrays that are initially empty, but that can grow or shrink by one element at a time. We call a flexible array that grows and shrinks only at the front of the array, and is indexed from the first element, a *random-access list.*

We wish to provide the standard list operations *cons*, *head*, and *tail*, as well the empty random-access list *empty*, and a predicate *isempty*. It is in these operations that complete binary trees and greedy decompositions prove their worth. As the following property demonstrates, changing n by ± 1 only affects at most the two smallest terms of the greedy decomposition of n, suggesting that *cons* and *tail* can be implemented in $O(1)$ time, as opposed to $O(\log n)$ time as required by virtually every other tree-based representation of arrays.

Property 4 *(a) If $G(n)$ contains no repeated terms, then*

$$G(n+1) = \{1\} \uplus G(n)$$

(b) If $G(n)$ contains a repeated term $2^k - 1$, then

$$G(n+1) = \{2^{k+1} - 1\} \uplus G(n) - \{2^k - 1, 2^k - 1\}$$

Proof: (a) Since $G(n)$ contains no repeated terms, $\{1\} \uplus G(n)$ may contain repeated 1 terms, but otherwise all terms are unique. Thus, by Property 2, $\{1\} \uplus G(n)$ is the greedy decomposition of $n+1$.

(b) By Property 2, $G' = G(n) - \{2^k - 1, 2^k - 1\}$ is composed entirely of unique terms larger than $2^k - 1$. Thus, $\{2^{k+1} - 1\} \uplus G'$ may contain repeated $2^{k+1} - 1$ terms, but otherwise all terms are unique. Therefore, $\{2^{k+1} - 1\} \uplus G'$ is the greedy decomposition of $n+1$. □

Property 4 suggests the following implementation of list operations: To *cons* a new element onto a random-access list, check if the two smallest trees in the existing list are the same size. If not, add the new element as a singleton tree. Otherwise, join the two trees into a new, larger tree with the new element as the root. Provided the trees are maintained in increasing order of size, this can be accomplished in $O(1)$ time. To take the *tail* of a list, simply remove the leading singleton tree, if one exists. Otherwise, remove the root of the smallest tree, and add its two subtrees to the collection. This also requires only $O(1)$ time. These operations are illustrated pictorially in Figure 5. The remaining list operations (*head*, *empty*, *isempty*) are trivial to implement in $O(1)$ time. Figure 6 contains code for all five list operations.

The fact that all list primitives can be implemented in only $O(1)$ time is remarkable since it implies that any application using standard lists can be rewritten to use random-access lists with no loss of efficiency (up to a constant factor) *even if the application never takes advantage of random access.* In practice, slowdowns on list-intensive applications that do not use random access are less than about a factor of 2. And, of course, applications that do take advantage of random access can observe arbitrary speedups.

5 Analysis

Recall that lookups and updates require $O(i)$ time for standard lists. For small values of i, this is better than $O(\log n)$. However, since every step through the collection or down a tree decreases i by at least one, a more accurate bound on the worst-case running time for lookups and updates in random-access lists is $O(\min\{i, \log n\})$, showing that random-access lists are never less efficient than standard lists. This bound is tight (consider an array represented as a single tree).

Several competing data structures, notably Braun trees [19, 28] and finger search trees [21, 33], improve the worst-case bounds on lookups and updates slightly to $O(\log i)$. We now show that random-access lists match this bound, albeit in the expected case rather than worst case.

First, consider a random-access list containing exactly one tree of every size up to $2^{\lfloor \log(n+1) \rfloor} - 1$. Notice that the kth tree has depth k, and that every element i is contained in approximately the $(\log i)$-th tree. Thus, every element can be accessed in $O(\log i)$ time: $O(\log i)$ to find the right tree in the collection, and $O(\log i)$ to find the right element in the tree. In fact, this bound holds whenever there are no "gaps" in the collection where trees of more than c consecutive sizes are missing, for some constant c (Kosaraju [21] calls this the *gap property*). The intuition behind the $O(\log i)$ expected bound is that large gaps are unlikely for randomly chosen n.

Let $k = \lceil \log(i+2) \rceil$ and notice that $2^k - 1 > i$ so that if there is a tree of size $2^k - 1$ in the collection, then element i occurs no later in the collection than that tree. If element i occurs in an earlier tree, then it certainly requires only $O(\log i)$ time to access. Otherwise, it requires at most $O(k')$ time, where $2^{k'} - 1$ is the size of the first tree of size $\geq 2^k - 1$ that is present in the collection. For randomly chosen n, the probability that $k' = k$ is $1/2$. The probability that $k' = k+1$ is $1/4$, and, in general, the probability that $k' = k+m$ is $1/2^{m+1}$. Thus, the expected running time of an access is

$$O\left(\frac{k}{2} + \frac{k+1}{4} + \cdots + \frac{k+m}{2^{m+1}} + \cdots\right)$$

This series converges to $O(k) = O(\log i)$.

6 Variations

Sometimes arrays that are extensible at the rear rather than the front are more convenient. Or you may wish to extend the array at the front, but without shifting the existing indices. Both of these variations are quite simple to simulate using random-access lists by associating a total size or starting index with each list.

Alternatively, you may wish to extend the array at the rear, but remove elements from the front, yielding a random-access queue. This can be implemented by substituting random-access lists for standard lists in the purely functional queues of Hood and Melville [18]. Similarly, arrays that can grow or shrink at either end (random-access deques) can be implemented by substituting random-access lists for standard lists in the purely functional deques of Hood [17] or Chuang and Goldberg [10]. Note that the purely functional queues and deques of Okasaki [27] are not suitable for this modification since they depend on lazy lists, and random-access lists are (spine) strict.

A *min-list* is a list data structure that supports an additional operation *min* that returns the minimum element. (but note that the operation *delete-min* is not supported). Min-queues and min-deques are defined similarly. Such data structures have numerous applications, including VLSI river routing [11], all-pairs shortest-path algorithms [15], and computing external farthest neighbors for simple polygons [2]. We now show how to implement random-access min-lists with the same bounds as random-access lists. Just as above, random-access min-queues and min-deques may be built from

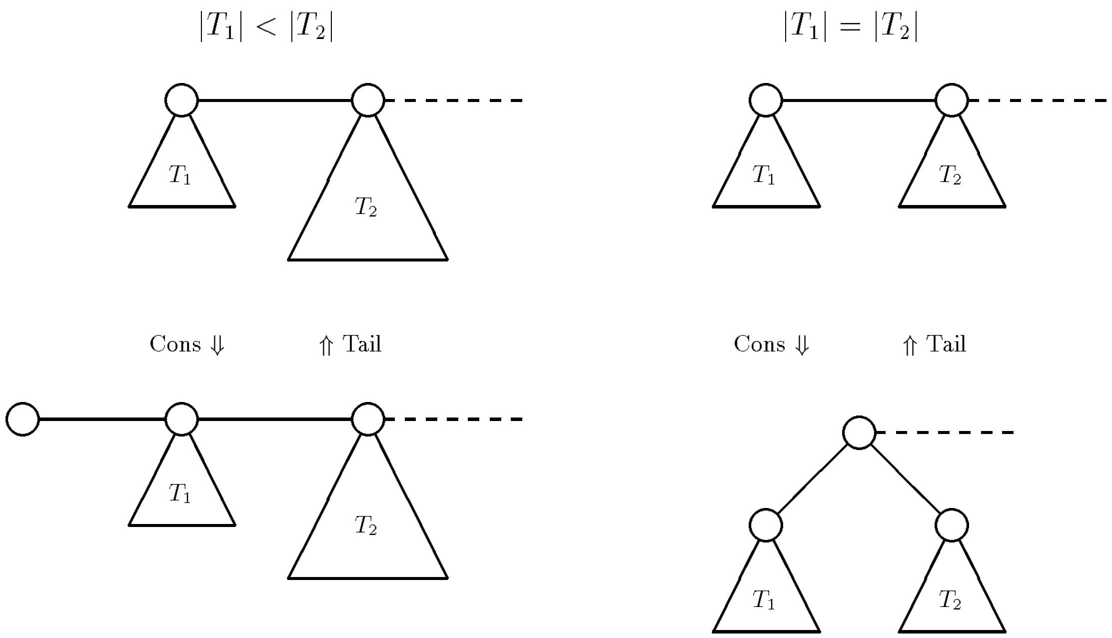

Figure 5: The two cases of *cons* and *tail* on random-access lists.

```
type 'a random_access_list = (int * 'a tree) list

exception Empty

val empty = []

fun isempty [] = true
  | isempty ((size,t) :: rest) = false

fun cons x (xs as ((size1,t1) :: (size2,t2) :: rest)) =
        if size1 = size2 then (1+size1+size2,Node (x,t1,t2)) :: rest
                         else (1,Leaf x) :: xs
  | cons x xs = (1,Leaf x) :: xs

fun head [] = raise Empty
  | head ((size,Leaf x) :: rest) = x
  | head ((size,Node (x,t1,t2)) :: rest) = x

fun tail [] = raise Empty
  | tail ((size,Leaf x) :: rest) = rest
  | tail ((size,Node (x,t1,t2)) :: rest) =
        let val size' = size div 2
        in
            (size',t1) :: (size',t2) :: rest
        end
```

Figure 6: Operations on random-access lists.

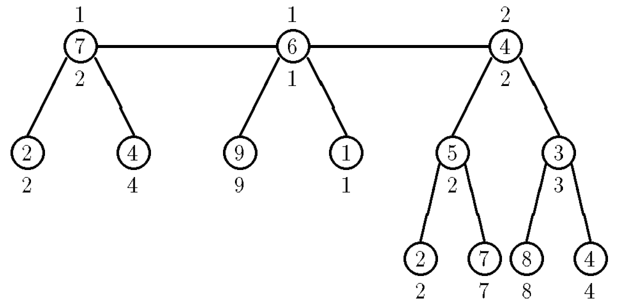

Figure 7: A random-access min-list. Tags on nodes are written below the node. The additional tags on roots are written above the node.

such lists. Gajewska and Tarjan [16] first described efficient min-deques, but without supporting random access.

It is very simple to associate a global minimum with each list. However, when the minimum element is removed or updated, we do not wish to search the entire list for the new minimum. Instead we tag every tree node with the minimum element in the subtree rooted at that node, and every root in the collection of trees with the minimum element from this and all subsequent trees (see Figure 7). Then the overall minimum is the tag on the first root. Maintaining these tags, even during updates, is not difficult since any given tag can be recomputed in $O(1)$ time with strictly local information. Tags on tree nodes depend only on the value at the node and the tags of its two children. Tags on roots in the collection depend only on the tag of the current root and the tag of the next root in the collection. The only tags that need to be recomputed during an update are those for nodes that are reconstructed by the update anyway, so there is no penalty for maintaining this information.

7 Related Work

A series of array accesses (lookups or updates) is called *single-threaded* if every operation refers to the most recent version of the array, never to a previous version [31]. Single-threaded array accesses may safely be implemented destructively, requiring only $O(1)$ time. The vast majority of research on functional arrays has focused on detecting or enforcing single-threadedness through such techniques as static analysis [7, 26, 30], linear types [36], or monads [29].

An array that supports non-singled-threaded access is called *fully persistent* [14]. A number of data structures have been proposed to represent fully persistent arrays. One technique, known as version arrays [1] or shallow binding [4, 5], maintains a cache of elements along with trees of differential nodes indicating differences from the cache. This technique yields $O(1)$ performance for single-threaded accesses, but may degrade to $O(\#\text{updates})$ for non-single-threaded accesses. Chuang [9] uses randomization to improve the expected performance for non-single-threaded accesses. Unfortunately, version arrays rely on destructive operations internally, and so cannot be implemented in a purely functional language (although they may be provided as primitives for such a language). Furthermore, version arrays are limited to a fixed size (although Chuang's randomized method could be adapted to flexible arrays).

Dietz [12] exploits the fact that indices of fixed-size arrays are bounded integers to provide an implementation requiring $O(\log\log n)$ expected amortized time per operation.

Various balanced binary tree schemes have been used to implement fully persistent arrays purely functionally, including AVL trees [25] and Braun trees [19, 28]. Such trees easily support $O(\log n)$ lookups and updates ($O(\log i)$ in the case of Braun trees), but require $O(\log n)$ time to add or remove an element.

Finger search trees [21, 33] are similar to binary search trees. Instead of beginning searches at the root, however, finger search trees begin searches at the leftmost leaf, first searching upward to find an ancestor of the desired node and then searching downward in the normal fashion. Lookups and updates require $O(\log i)$ time, and adding or removing a leaf in the leftmost position can be accomplished in $O(1)$ time. Finger search trees can be made fully persistent using the (internally destructive) techniques of Driscoll *et al.* [14], but it is not clear whether they can be implemented purely functionally without suffering a degradation in performance.

Kaplan and Tarjan [20] recently introduced the algorithmic notion of recursive slowdown, and used it to design a new purely functional implementation of constant-time double-ended queues. A pleasant accidental property of their data structure is that it also supports random access in $O(\log d)$ time, where d is the distance from the desired element to the nearest end of the queue.

A very different technique was introduced by Myers [24] to implement random-access stacks. He augments a standard singly-linked list with auxiliary pointers allowing one to skip arbitrarily far ahead in the list. The number of elements skipped by each auxiliary pointer is controlled by the digits of a canonical skew-binary number (virtually identical to the greedy skew-binary decompositions described here). His scheme allows $O(1)$ list operations, and $O(\log n)$ lookups, but requires $O(i)$ time for updates. The difficulty with updates is that his scheme contains many redundant pointers. Removing these redundant pointers yields a structure isomorphic to the random-access lists described here (see Figure 8). Thus, random-access lists may be viewed as an improvement to Myers' representation to support efficient updates.

Finally, random-access lists share many similarities with binomial heaps [34]. Both are based on rigidly structured trees of certain, fixed sizes (2^k for binomial heaps and 2^k-1 for random-access lists). Both represent aggregates as collections of trees. Both build bigger trees by joining trees of identical size. Adding a single element to each structure is highly reminiscent of binary arithmetic. The difference is that binomial heaps use binary arithmetic, in which carries may propogate arbitrarily far, while random-access lists use skew-binary arithmetic, in which carries may propogate only a single position.

8 Measurements

To determine the effectiveness of our data structure, we compared the running times of six purely functional data structures on a variety of problems and problem sizes. In addition to random-access lists, we also measured the performance of standard lists, Myers' random-access stacks [24], AVL trees [25], Braun trees [19, 28], and Kaplan-Tarjan lists [20]. For both AVL trees and Kaplan-Tarjan lists, we optimized the implementations to support insertions and deletions only at the front. Otherwise, only minor tuning was performed, such as replacing division by two with a bitwise shift operation.

We used five benchmark problems:

- Sum: construct a list containing n integers, and then

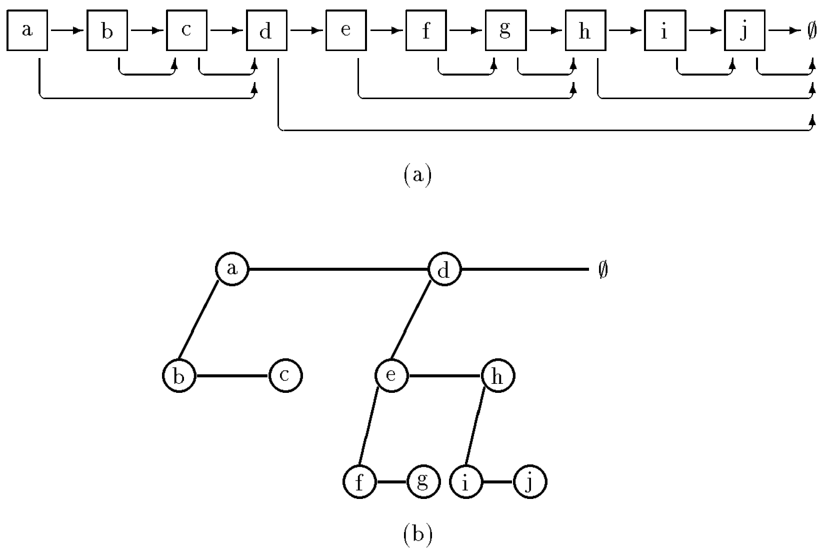

Figure 8: Myers' random-access stacks with redundant pointers (a) and without (b)

sum the elements

- Lookup: sequentially lookup each element of an existing list of size n
- Update: sequentially update each element of an existing list of size n
- Quicksort: sort a list of n random integers using the quicksort algorithm on lists
- Histogram: count the occurrences of each integer in a list of $5n$ integers chosen randomly from $0 \ldots n-1$

The first three benchmarks were run on problem sizes of 1000, 10,000, and 100,000; the latter two benchmarks on problem sizes of 1000 and 10,000. The Sum and Quicksort benchmarks measure the effectiveness of each data structure when used as a list. The Lookup, Update, and Histogram benchmarks measure the effectiveness of each data structure when used as an array.

The results of these experiments are shown in Figure 9. For each experiment, we also give the ratio of its average running time to that of the fastest data structure on that problem and problem size. Examining the ratios, we see that (not surprisingly) standard lists outperform all other data structures when used as lists, but are totally impractical as arrays. Myers' random-access stacks also perform quite well as lists, but their lack of efficient updates cripples their use as arrays. On the other hand, AVL trees and Braun trees both excel at array applications while faring poorly on list applications. Kaplan-Tarjan lists yield mediocre results on every experiment, but recall that they were not designed with random access in mind.

Of all the data structures, only random-access lists are competitive at every task. They are nearly as efficient as standard lists on list applications, and nearly as efficient as binary trees on array applications. In fact, they are always within about a factor of two of the fastest data structure on any given problem. This makes random-access lists the data structure of choice for applications featuring both patterns of usage (particularly when array accesses are non-single-threaded).

Finally, the erratic performance of Braun trees deserves some comment. Braun trees perform very well as arrays when either the array is small or access is random. However, they perform surprisingly poorly for sequential access on large arrays. Closer examination reveals an inordinate amount of time being spent collecting garbage. This is because Braun trees exhibit very poor locality — other than near the root, elements that are close in the array ordering will always be stored in distant segments of the tree. This interacts poorly with generational garbage collectors [22], which depend on locality for good performance.

9 Discussion

We have presented an implementation of random-access lists that supports $O(\log n)$ array operations and $O(1)$ list operations. For randomly chosen n, the bound for array operations can be improved to $O(\log i)$ expected time. This data structure is both simple to code, and efficient enough to be a practical alternative to standard lists even when random access is not required.

Although the $O(\log n)$ bounds are disappointing when compared to the $O(1)$ bounds of imperative arrays, Ben-Amram and Galil [6] have shown that this is optimal for any linked structure of bounded-width nodes, even allowing for destructive updates. Improving these bounds would require the use of substructures of unbounded width supporting random access. In order to maintain the $O(1)$ bounds on list operations, such substructures would need to be initializable in $O(1)$ time.

There is one sense in which the replacement of standard lists with random-access lists might be unpalatable, and that is pattern matching. To maintain the necessary structure, random-access lists would need to be provided as an abstract data type, and abstract data types are well known to clash with pattern matching. Pattern matching is an extremely convenient way of writing functions involving lists (and many other data structures), but it requires knowledge of the data structure's representation. Mechanisms for pattern matching on abstract data types have been pro-

Problem	N	Standard List				Myers Stack				Random-Access List			
		User Time	GC Time	Total Time	Ratio	User Time	GC Time	Total Time	Ratio	User Time	GC Time	Total Time	Ratio
Sum	1000	0.010	0.004	0.014	1.00	0.011	0.005	0.016	1.13	0.014	0.002	0.016	1.11
	10000	0.105	0.129	0.234	1.00	0.118	0.208	0.326	1.39	0.145	0.213	0.358	1.53
	100000	1.032	1.562	2.594	1.00	1.170	3.845	5.015	1.93	1.489	4.060	5.550	2.14
Lookup	1000	0.263	0.000	0.264	17.58	0.020	0.000	0.021	1.37	0.019	0.000	0.020	1.32
	10000	—	—	—	—	0.250	0.007	0.257	1.48	0.235	0.006	0.241	1.39
	100000	—	—	—	—	2.859	0.015	2.874	1.44	2.480	0.016	2.497	1.25
Update	1000	3.297	4.585	7.882	189.64	3.599	3.714	7.313	175.96	0.063	0.003	0.066	1.59
	10000	—	—	—	—	—	—	—	—	0.897	0.264	1.161	1.45
	100000	—	—	—	—	—	—	—	—	10.598	6.239	16.837	1.38
Quicksort	1000	0.191	0.010	0.201	1.00	0.201	0.016	0.217	1.08	0.236	0.021	0.257	1.28
	10000	2.686	1.203	3.890	1.00	2.677	1.832	4.509	1.16	3.121	1.968	5.089	1.31
Histogram	1000	17.859	19.795	37.654	46.70	18.305	22.619	40.924	50.75	0.502	0.340	0.842	1.04
	10000	—	—	—	—	—	—	—	—	7.233	7.869	15.102	1.46

Problem	N	AVL Tree				Braun Tree				Kaplan-Tarjan List			
		User Time	GC Time	Total Time	Ratio	User Time	GC Time	Total Time	Ratio	User Time	GC Time	Total Time	Ratio
Sum	1000	0.098	0.019	0.117	8.34	0.055	0.006	0.061	4.34	0.065	0.014	0.080	5.67
	10000	1.287	0.312	1.599	6.83	0.814	1.477	2.291	9.79	0.673	0.445	1.119	4.78
	100000	15.640	3.328	18.969	7.31	11.394	23.291	34.685	13.37	5.695	10.825	16.520	6.37
Lookup	1000	0.015	0.000	0.015	1.03	0.015	0.000	0.015	1.00	0.030	0.001	0.031	2.09
	10000	0.173	0.000	0.174	1.00	0.210	0.002	0.212	1.22	0.420	0.004	0.424	2.44
	100000	1.990	0.003	1.993	1.00	3.167	0.022	3.189	1.60	5.154	0.166	5.320	2.67
Update	1000	0.040	0.006	0.046	1.11	0.039	0.002	0.042	1.00	0.099	0.008	0.107	2.57
	10000	0.542	0.260	0.803	1.00	0.602	0.727	1.328	1.65	1.155	0.478	1.633	2.03
	100000	6.612	5.570	12.182	1.00	8.012	17.937	25.949	2.13	13.470	9.548	23.018	1.89
Quicksort	1000	0.953	0.168	1.121	5.58	0.508	0.037	0.546	2.72	0.956	0.257	1.213	6.04
	10000	16.645	3.014	19.658	5.05	10.260	7.548	17.809	4.58	10.495	4.372	14.867	3.82
Histogram	1000	0.423	0.399	0.821	1.02	0.367	0.439	0.806	1.00	0.721	0.642	1.363	1.69
	10000	5.695	6.508	12.204	1.18	4.852	5.478	10.330	1.00	10.092	10.899	20.991	2.03

Figure 9: Average running time (in seconds) for each combination of data structure, problem, and problem size. Times are broken down into user time and garbage collection time. Dashes (—) indicate tests that were too slow to measure. For each experiment, we also give the ratio of its total running time to that of the fastest data structure on the same problem and problem size. Note that only random-access lists are competitive on every task. All measurements were taken on a DecStation 5000/200 with 32MB of RAM running Mach 3.0 and SML/NJ 0.93.

posed [35, 8], but have yet to be incorporated into a major functional language. (Note that the more limited proposal of Aitken and Reppy [3] is not robust enough to cope with random-access lists.)

Finally, note that the techniques of Shao *et al.* [32] for reducing the space requirements of standard lists by storing multiple values per node apply equally well to random-access lists.

Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

References

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[2] Pankaj K. Agarwal, Alok Agarwol, Boris Aronov, S. Rao Kosaraju, Baruch Schieber, and Subhash Suri. Computing external farthest neighbors for a simple polygon. *Discrete Applied Mathematics*, 31(2):97–111, April 1991.

[3] William E. Aitken and John H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11, 1992.

[4] Henry G. Baker. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

[5] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, August 1991.

[6] Amir M. Ben-Amram and Zvi Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, July 1992.

[7] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, London*, pages 26–38, 1989.

[8] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.

[9] Tyng-Ruey Chuang. A randomized implementation of multiple functional arrays. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 173–184, 1994.

[10] Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 289–298, 1993.

[11] Richard Cole and Alan Siegel. River routing every which way, but loose. In *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 65–73, 1984.

[12] Paul F. Dietz. Fully persistent arrays. In *Proceedings of the First Workshop on Algorithms and Data Structures*, volume 382 of *LNCS*, pages 67–74. Springer-Verlag, 1989.

[13] Edsger W. Dijksta. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[14] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[15] Greg N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, January 1991.

[16] Hania Gajewska and Robert E. Tarjan. Deques with heap order. *Information Processing Letters*, 22(4):197–200, April 1986.

[17] Robert Hood. *The efficient implementation of very-high-level programming language constructs*. PhD thesis, Department of Computer Science, Cornell University, 1982.

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

[19] Rob R. Hoogerwoord. A logarithmic implementation of flexible arrays. In *Proceedings of the Second International Conference on the Mathematics of Program Construction, Oxford*, volume 669 of *LNCS*, pages 191–207. Springer-Verlag, 1992.

[20] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pages 93–102, 1995.

[21] S. Rao Kosaraju. Localized search in sorted lists. In *13th Annual ACM Symposium on Theory of Computing*, pages 62–69, 1981.

[22] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[23] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[24] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.

[25] Eugene W. Myers. Efficient applicative data types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, 1984.

[26] Martin Odersky. How to make destructive updates less destructive. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 25–36, 1991.

[27] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, October 1995. To appear.

[28] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[29] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[30] A. V. S. Sastry, William Clinger, and Zena Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 266–275, 1993.

[31] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[32] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 185–195, 1994.

[33] Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1–3):173–194, October 1985.

[34] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978.

[35] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, 1987.

[36] Philip Wadler. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, 1990.