Efficient Applicative Data Types†

Eugene W. Myers

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

1. Introduction

Applicative programming has long been advocated on theoretical grounds because the formal properties of such programs are simple and elegant. Recently, there has been a trend to use the applicative approach in software development tools [3] and programming languages [2,7]. Unfortunately, the requirement that operations be free of side-effects makes it difficult to achieve efficient implementations [7]. To date, there are only two published algorithms [5,8], which treat the applicative manipulation of queues and stacks.

This work presents $O(lgN)$ time and space algorithms for the applicative manipulation of linear lists. A generalization of an AVL tree, called an AVL dag, is used. While the result is simple, its consequences are far reaching. Since almost every non-scalar data type can be modeled with lists, the results presented here are a powerful method for improving the implementations of applicative languages. The results also provide a fast and space efficient method for constructing history systems such as editors with unlimited "undos" and version control systems. Finally, lists can be realized in value-semantic programming languages, such as PASCAL, with worst case performance superior to any previously proposed solution.

Space is at a premium in history systems. An enhancement that improves absolute space performance by a factor of two for applicative editor operations (e.g. move, transfer, etc.) is also presented. The $O(1)$ time and space results in [5], suggest the possibility of yet more efficient applicative algorithms for the simpler sub-abstractions of list. Algorithms are presented for arrays that perform in $O(KN^{1/K})$ time and $O(K)$ space where K may be chosen arbitrarily.

†This work was supported by the National Science Foundation under Grant MCS-8210096.

2. Motivation

In many applications it is desirable to view a list data structure as an instance of a "linear list". For example, programming languages manipulate "integers", "arrays", "strings", etc. Formally, we assume that a *value-semantic linear list data abstraction* consists of: an arbitrary and time varying number of *objects* of type linear list; a fixed finite collection of *operators* that access and manipulate objects; and a time varying set of *variables*, each of which *refers* to an object. Variables can be created, destroyed, and their reference relation can be modified by *assignment*. A variable denotes the *value* of the object to which it refers. Only assignment can change the value of a variable.

The *state* of the data abstraction is the collection of objects referred to by the current set of variables. A typical linear list operation repertoire [6] consists of:

(1) LEN(L):Integer
Determine the number of elements in list L.
(2) SEL(L,k):X
Select the k^{th} element of list L.
(3) RNK(L,x):Integer
Determine the rank of x in an *ordered* list L.
(4) ADD(L,k,x):List_of_X
Insert x after the k^{th} element of list L.
(5) DEL(L,k):List_of_X
Delete the k^{th} element of list L.
(6) CON(L_1,L_2):List_of_X
Concatenate L_1 and L_2.
(7) SUB(L,i,j):List_of_X
Select the i^{th} through j^{th} elements of L.

The primitives have been formulated as functions. The constraint that only an assignment can change the value of a variable forces an implementation in which an operator may not modify the values con-

tained in its operands, i.e. it must operate *applicatively*. Such implementations are *applicative data types*.

Height-balanced models such as AVL trees [1,6] are frequently used to model linear lists because of their "smooth" $O(lgN)$ worst-case behavior. They are, however, unsuitable in a value-semantic framework because of their procedural formulation. For example, the concatenation algorithm is usually formulated as a procedure, $CATENATE(L_1,L_2,L_3)$, which places the concatenation of L_2 and L_3 in L_1. The procedure doesn't consume space, but it destroys its operands L_2 and L_3 through rebalancing and join operations. The presumed origin of this semantic choice was the assumption that the preservation of the operand trees would require that copies be made at an intolerable cost of $O(N)$ time and space. As shown in the next section, this assumption is false.

3. AVL Dags

The observation that led to the concept of an AVL dag and its applicative algorithms is illustrated by an example. Consider the AVL tree at the left in Figure 1. Suppose that the vertex labeled X is to be added as the right child of vertex 5. The procedure-oriented AVL algorithm modifies vertices 4, 5, and 6 to produce the result shown at the upper right. The list represented by their ancestor, 3, is also indirectly affected. Let Δ be the set of vertices directly modified by an AVL operation. Let Δ^* be the set consisting of Δ and every ancestor of a vertex in Δ. It follows that Δ^* is exactly the collection of vertices, v, whose list values are modified by the operation. Thus, the result of the operation can be represented (while still preserving the original tree) by using copies of just the vertices in Δ^*. In the example, this leads to the *dag* shown at the lower right of Figure 1.

Note that in Figure 1, Δ^* is exactly the set of vertices on the search path from the root to the point of insertion. For other AVL operations, Δ^* more generally depends on the balances of search path vertices and their children and grandchildren. Nonetheless, the cardinality of Δ must be $O(lgN)$ for any AVL algorithm because it operates in $O(lgN)$ time. Moreover, Δ^* is $O(lgN)$ because AVL algorithms modify a tree along just one or two search paths.

The tree resulting from an AVL operation is to be represented augmentatively and consequently it may refer to unaltered vertices in its operand trees. Thus the representation for the state of an AVL data type is a dag with the following special properties.

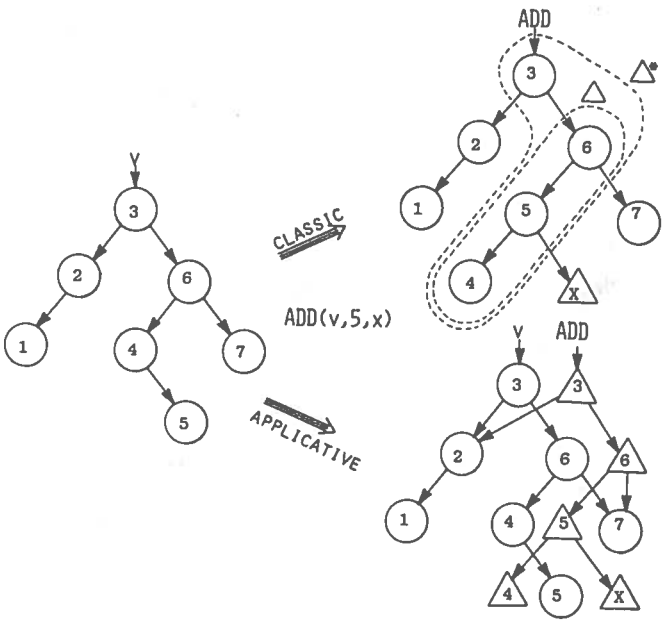

FIGURE 1: THE CENTRAL IDEA

(a) The dag is a *binary dag*. There is a special vertex Λ that has out-degree 0. Every other vertex has out-degree two and its two children are further ordered into "left" and "right".

(b) Let the *height* of a vertex, v, in the dag be the length of the longest path from v to Λ. The dag satisfies the height-balance property: the heights of the left and right children of every vertex (except Λ) differ by at most 1.

A dag satisfying these properties is an *AVL dag*. Figure 2 gives an example.

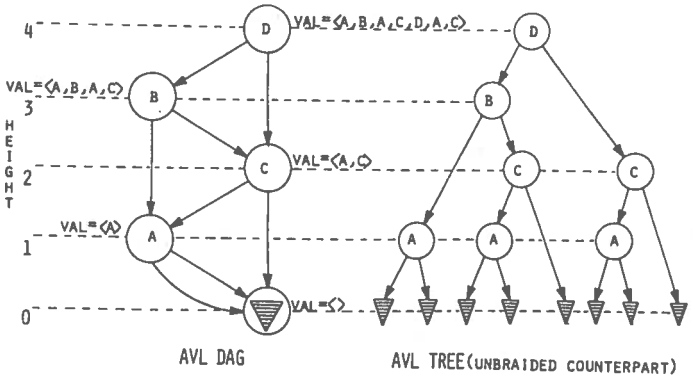

FIGURE 2: AVL DAG EXAMPLE

An AVL dag encodes a collection of linear lists as follows. As for AVL trees, each vertex v (except Λ) is labeled with a list element. The *value of* v, Val(v), is

the list of labels encountered in the symmetric and unmarked traversal of the subdag dominated by v. Equivalently, one can imagine "unbraiding" this subdag by replicating every vertex with in-degree greater than one until a tree rooted at v is obtained. This tree is an AVL tree and its symmetric label list is Val(v). Thus every vertex in the AVL dag represents an AVL tree whose symmetric order list is the linear list being modeled. It then follows [1] that the length of every path from v to Λ is $\Theta(lgN)$ where $N = |Val(v)|$.

4. Applicative AVL Algorithms

An AVL dag models the state of the list data type. Each variable refers to a dag vertex v; the list denoted by the variable is Val(v). The list operations require new vertices and cause current state vertices to become unused, i.e. the vertex cannot be reached from any vertex referenced by a variable. Thus a storage management scheme that allocates and garbage collects vertices is needed. An incremental reference counter method [10] provides the $O(1)$ allocation and collection algorithms -- NEW, INC, DEC. References to vertex v are created and destroyed with INC(v) and DEC(v). NEW(l,x,r) generates an initial reference to a new vertex with label x and left and right sons l and r. The efficiency of the method in [10] permits the *on-line cost* of storage management to be *included* in subsequent complexity claims (see Appendix A).

Each AVL list space vertex is modeled by the record:

```
Type vertex =
   Record
      RC : integer  (* Reference Count *)
    LN,H : integer  (* Length and Height *)
     L,R : ↑vertex  (* Left and Right Siblings *)
       V : X        (* Label (List Element) *)
   End

Type list = ↑vertex
```

The LN field contains |Val(v)|+1 and is used in searching for a list element. The fields LN and H are maintained by the list space primitive NEW. For simplicity, the INC and DEC calls needed to maintain reference counts will be ignored.

The applicative nature of AVL dags suggests that a recursive and functional "bottom-up" approach be used in the exposition of the algorithms. It has long been recognized that the functional style of programming leads to terse and conceptually clear algorithms.† Consider the operation ADD(v,k,x) which inserts x after the k^{th} element of Val(v). For the moment, ignore the constraint that the result be height-balanced. If $v = \Lambda$ then Val(ADD(v,k,x)) = <x> and thus NEW(Λ,x,Λ) generates the desired reference. Proceeding inductively, if $v \neq \Lambda$ and $k < v.L.LN$ then Val(ADD(v.L,k,x)) = Val(ADD(v,k,x))[1..v.L.LN] implying that NEW(ADD(v.L,k,x),v.V,v.R) generates the correct refereence. Similarly, when $k \geq v.L.LN$, the desired reference is generated by NEW(v.L,v.V,ADD(v.R,k−v.L.LN,x)). These facts lead to the algorithm:

```
Function ADD(v: list; k: integer; x: base) : list
1. If v = Λ Then
2.    ADD ← NEW(Λ,x,Λ)
3. Else If k < v.L.LN Then
4.    ADD ← NEW(ADD(v.L,k,x),v.V,v.R)
5. Else
6.    ADD ← NEW(v.L,v.V,ADD(v.R,k−v.L.LN,x))
```

The recursion descends along a search path to Λ and produces its result by adding new vertices to the AVL dag as it proceeds back up this path.

While applicatively producing the correct list value, ADD does not retain the height balance property because the heights of search path vertices may be incremented producing a number of locally unbalanced sights. These local imbalances are rectified by a central utility algorithm, BAL(l,x,r), which performs the applicative equivalent of the single and double rotations used in conventional AVL algorithms. The function BAL(l,x,r) has the same effect as the primitive NEW (i.e. returns Val(l)*<x>*Val(r)), but in the event that $l.H-r.H \in [-2,2]$, BAL guarantees a height-balanced result.

```
Function BAL(l: list; x: base; r: list) : list
1. If l.H-r.H ∈ [-1,1] Then
2.    BAL ← NEW(l,x,r)
3. Else If l.H > r.H Then
4.    If l.L.H ≥ l.R.H Then
5.       BAL ← NEW(l.L,l.V,NEW(l.R,x,r))
6.    Else
7.       BAL ← NEW(NEW(l.L,l.V,l.R.L),
                   l.R.V,NEW(l.R.R,x,r))
8. Else
9.    If r.R.H ≥ r.L.H Then
10.      BAL ← NEW(NEW(l,x,r.L),r.V,r.R)
11.   Else
12.      BAL ← NEW(NEW(l,x,r.L.L),r.L.V,
                   NEW(r.L.R,r.V,r.R))
```

† The entire AVL dag implemented list abstraction (including storage management) was written in 193 lines of C.

OPERATOR	TIME	Δ^*:ORDER	Δ^*:ABSOLUTE	Δ^*:EXPECTED
SEL(V,K)	O(LG N)	0	0	0
RNK(V,X)	O(LG N)	0	0	0
ADD(V,K,X)	θ(LG N)	θ(LG N)	≤ V.H+1	.995 LG N + 1.23
DEL(V,K)	θ(LG N)	θ(LG N)	≤ $\frac{3}{2}$(V.H-1)	.998 LG N - .275
CON(L,R)*	θ(LG M)	θ(LG M)	≤ L.H + $\frac{1}{2}$(R.H-1)	.966 LG M + .520
SUB(V,I,J)	θ(LG N)	θ(LG R)	≤ 3V.H-5	1.94 LG R - .170

WHERE N = |VAL(V)| M = |VAL(L)| R = J-I

* WLOG ASSUME L.H ≥ R.H

TABLE I: AVL DAG PERFORMANCE SUMMARY

BAL is $O(1)$ and it adds a maximum of three new vertices to the dag. To maintain the height-balance property in the algorithm ADD above, simply use use BAL instead of NEW.

The algorithms for deletion, concatenation, and substring selection are similarly obtained by formulating the applicative equivalents of their procedural AVL tree counterparts (see Appendix A). The selection and rank primitives are elementary binary search algorithms. By analogy with the classic AVL algorithms, these AVL dag algorithms require $O(lgN)$ time. Moreover, it follows that BAL is called at most $O(lgN)$ times and thus Δ^* is $O(lgN)$ for every operator. A more detailed analysis yields the performance bounds presented in Table I. The issue of space performance is new to AVL dag algorithms. Table I includes tight upper bounds for the absolute value of Δ^* and its experimently determined expected value. The derivations of the tight upper bounds are given in [9]. The expected performance statistics were obtained by running a linear regression on the average value of Δ^* for a geometric series (5 to 2134 in steps of 1.4) of 19 parameter values. Each average was obtained by running 500 experiments on randomly constructed AVL trees of the required parameter size.

The augmentative approach employed here is not specific to AVL trees, but can be applied to any tree-based method yielding new data structures such as B dags, 2-3 dags, heap dags, and dynamic binary search dags. The corresponding applicative algorithms have the same time performance as their procedural counterparts; the space performance of a transformational primitive (e.g. ADD vs. SEL) is the same as its time performance.

5. Applications

A. A Value Semantic List Data Abstraction

The immediate consequence of AVL dags is a superior implementation of list algorithms in a value-semantic framework.

Corollary 1: Let A be a list algorithm whose complexity is $O(F)$ when list operations are assumed to be atomic (i.e. $O(1)$). Further suppose that A uses I elements of input and constant data and that A outputs O elements of output. Using the AVL dag algorithms, the real-time complexity of algorithm A is $O(FlgL+I+O)$ where L is the average length of a list operand.

Any program must spend time proportional to the size of its input and output. $O(N)$ algorithms to build and write AVL dag encoded lists of length N can be easily constructed. Using AVL dags all operations require $O(lgN)$ time and space. Variables can be introduced and removed with $O(1)$ INC and DEC operations. Of paramount importance is the fact that assignment is also $O(1)$: DECrement the current variable reference and INCrement a new one to the desired value.

The most pathological possibility for the parameter L is represented by the algorithm that reads a list of length I and then doubles its length in the remaining F-1 steps. That is, L can be $\Omega(I2^F)$. Thus, in terms of F, I, and O, the AVL dag implementation guarantees that a list algorithm's real time complexity is $O(F(F+lgI)+I+O)$. The best performance of previous applicative list algorithms is a real time complexity of $O(FL+I+O)$ or $O(FI2^F+I+O)$.

Let the size of the current state S be the sum of the lengths of every variable's value. The correct maintenance of reference counts implies that the

number of vertices in the AVL dag is less than S. Moreover, the size of the dag may be as small as lgS. Thus the method is asymptotically space optimal. In practice, it may be *more* space efficient than conventional methods if the incidence of sharing is high enough to compensate for the incumbent overhead of the AVL dag model (~20 bytes per vertex).

The fundamental improvement expressed in Corollary 1 immediately leads to a plethora of new results for a variety of specific data abstractions. An applicative *ordered* list data type is obtained by employing the primitive RNK. As a result, the method provides applicative $O(lgN)$ implementations of the numerous sub-abstractions of list or ordered list, such as arrays, tables and any complex structure encoded as lists. In practical terms, the method applies to structures such as text files (a list of strings), program states (an array of words), and relational data bases (a finite set of ordered lists of ordered pairs).

B. Maintaining Histories

A *history tree* of an object is a rooted oriented tree in which each vertex denotes a value (called a *version*) of the object. The original version of the object is denoted by the root. The version of every other vertex is assumed to have been obtained by applying a transformational operator (called an *update*) to the version of its parent. The application of a passive operation (i.e. one that does not change the value of the object) is called a *query*. The problem is to maintain a history tree that permits efficient on-line algorithms for: adding a new leaf version through the application of an update to an existing version; deleting leaf versions; and querying an arbitrary existing version. Any applicative implementation solves this problem since the invocation of an update is guaranteed not to affect the version being operated upon. Thus, the AVL dag method provides the corollary:

Corollary 2: A history tree of lists can be maintained in $O(lgN)$ time per query or update and $O(lgN)$ space per update, where N is the size of the version upon which the operator is applied. Versions can be deleted in $O(1)$ time.

The generality and efficiency of the AVL dag method is illustrated by contrasting it with the earlier work of [4]. Their history problem constrains the history tree to be a line-graph and their list abstraction only permits the operations ADD, DEL, RNK, and SEL (i.e. a table). Their update and query algorithms require $O(lg^2 N)$ time. In [4], these algorithms were applied to a number of geometric intersection problems. The use of AVL dags immediately gives better results and holds promise for other problems in computational geometry.

Practical and efficient history systems can be constructed with AVL dag history trees. For example, an editor with complete history maintains a history line-graph of versions where each version is a text file; an update operation is a line-oriented editor command that changes the text; and a query operation is a passive editor command on a version in the current history. A text file is a list of lines and is implemented as an AVL dag. Each line-oriented editor command can be implemented as a finite composition of the AVL dag algorithms. This history editor performs updates in $O(lgN)$ time and space and queries in $O(lgN)$ time where N is the number of lines in the text file.

Another class of history systems currently of practical interest are version control systems. For example, a source code control system maintains a history tree of versions where each version is a source code text file; an update operation is the action of an editing session on a version; and a query operation searches or accesses an arbitrary version of the source code. Modeling text file as above, an update operation is realized by using the history editor above to compute the AVL dag representing the final text file of the editing session. This requires $O(ElgN)$ time and $O(min(N, \Delta lgN))$ space where E is the number of edit commands in the session; N is the maximum number of lines in the text file; and Δ is the number of lines changed by the session. Query operations can access K contiguous lines of an N line version in $O(K+lgN)$ time.

6. Further Results

A. Saving Space in Applicative Editor Operations

The line-oriented editor of the last section manipulates applicative "text files". A typical operator repertoire consists of:

(1) LEN(T):Integer
Return the number of lines in text file T.
(2) FIND(T,a):Line
Return the a^{th} line of T.
(3) SCROLL(T,a,Φ)
Starting with the a^{th} line, pass successive lines of T to the "handler" procedure Φ until it returns a halt signal.
(4) REPLACE(T,a,s):Text_File
Replace the a^{th} line of T with line s.
(5) DELETE(T,a,b):Text_File
Delete lines a through b of T.
(6) INSERT(T,c,Φ):Text_File
Insert the sequence of lines returned by the procedure Φ immediately after line c of T.

(7) MOVE(T,a,b,c):Text_File
Move lines a through b of T immediately after line c.

(8) TRANSFER(T,a,b,U,c):Text_File
Place a copy of lines a through b of T immediately after line c of U.

While these operations can be expressed in terms of the list operations given in Section 2 (e.g. DELETE(T,a,b) ≡ CON(SUB(T,1,a−1), SUB(T,b+1,LEN(T)))), a closer examination of the underlying mechanism reveals an approach that uses half as much space. Although this improvement is only by a constant factor, it is of great practical value because it doubles the size of histories that can be maintained.

The basic action used to achieve the list functions CON and SUB is embodied in the applicative function JOIN(l,x,r), which has the same effect as BAL (i.e. returns Val(l)*<x>*Val(r)), but is correct *regardless* of the heights of l and r. The procedural version of JOIN is treated in [6] where x is called the juncture value (see also Appendix A). The applicative version of JOIN adds a maximum of |l.H-r.H|+1 vertices to the AVL dag.

Consider the operator DELETE. The portion of the tree to be retained is represented by the height-monotone *slices* depicted in Figure 3. The desired result is obtained by successively joining the subtrees of the slices together with their interspersed juncture values. When formulated as above, DELETE first joins the subtrees of the left slice together in height increasing order. DELETE then joins the subtrees of the right slice. A final application of JOIN on the results of the proceeding two steps produces the desired list. The sum of the space bounds of each application of JOIN telescopes to give a total worst-case bound of 2H+V vertices, where H is the height of the heighest subtree in either slice and V is the number of subtrees in both slices.

Since concatenation is associative, any order of joins is correct. The improvement is obtained by using a better "join order" than the one above. Specifically, the subtrees in both slices are simultaneously joined in order of increasing height. This merged joining is analogous to the sorting technique of merging two ordered lists. With this order, only H+V vertices are added in the worst case.

For DELETE, the *height profile* of the slices form a "V". For other operations the height profiles are more complex and merging opposing slices is more subtle. For example, the operator TRANSFER requires the joining of four slices as depicted in Figure 4. For this "W"-profile the optimum join procedure is as follows: (1) in a

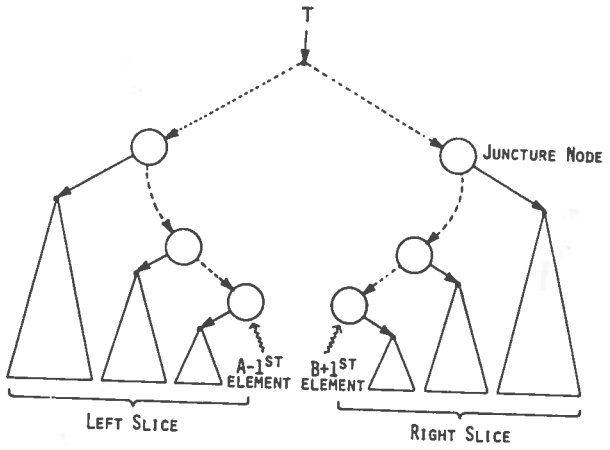

FIGURE 3: DELETE(T,A,B) SLICES

merged fashion join the slice $R_{a,b}$ and the portion of slice $U_{c+1,\infty}$ whose subtrees have height less than $lca.H$; (2) as in (1) join the lower portion of $U_{0,c}$ and all of $L_{a,b}$; (3) join the results of (1) and (2) with juncture value $lca.V$; and (4) in a merged fashion join the upper portions of $U_{0,c}$ and $U_{c+1,\infty}$ using the result of (3) to "seed" the merge. With this join order, at most H+L+V vertices are used where V is the number of subtrees in all the slices, H is the height of the heighest subtree, and L is the height of the vertex *lca*.

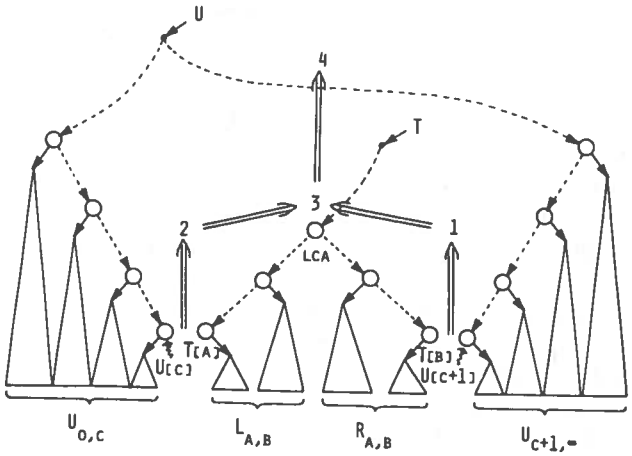

FIGURE 4: TRANSFER (T,A,B,U,C) SLICES

The number of subtrees in any slice is less than the difference in height between the lowest and highest subtrees of the slice. Thus V ≤ 2H in the analysis of DELETE. This implies an upper bound of 3H vertices when joins are merged and 4H vertices otherwise. However, these bounds are somewhat coarse as the space efficiency of the JOIN operations is not independent of the parameter V. As V

OPERATOR	TIME	Δ^*:ORDER	Δ^*:EXPECTED (UNOPTIMIZED)	Δ^*:EXPECTED (OPTIMIZED)
FIND(v,A)	O(Lg N)	0	0	0
SCROLL(v,A,P)	O(Lg N + K)	0	0	0
REPLACE(v,A,S)	θ(Lg N)	θ(Lg N)	.99 Lg N + .28	.95 Lg N - .28
DELETE(v,A,B)	θ(Lg N)	θ(Lg (N-R))	1.89 Lg (N-R) - .99	1.04 Lg (N-R) + .28
INSERT(v,C,P)	θ(Lg N + K)	θ(Lg N + K)	1.85 Lg N + K - .89	1.00 LG N + K + .23
TRANSFER(w,A,B,V,C)	θ(Lg N + Lg M)	θ(Lg N + Lg R)	1.83 (Lg N + Lg R) - .47	1.07 (Lg N + Lg R) + 1.11
MOVE(v,A,B,C)	θ(Lg N)	θ(Lg N)	1.95 (Lg N + Lg RS) - 1.19	1.13 (Lg N + Lg RS) + .41

WHERE N = |VAL(v)| M = |VAL(w)| R = |B-A| S = |C-B|
K = No. OF LINES PROCESSED BY P.

TABLE II: APPLICATIVE EDITOR PERFORMANCE SUMMARY

approaches 2H, the height differences between successive subtrees in each slice (and their merged sequence) become smaller until almost every such difference is 0 or 1. JOIN(l,x,r) requires |l.H-r.H|+1 vertices only when |l.H-r.H|≥2; otherwise it uses just |l.H-r.H| vertices. Thus as V approaches 2H, the joins become more space efficient. It can be shown that at most 2H vertices are used when joins are merged and 3H otherwise. These bounds are tight and imply that merging joins improves space performance by 1/3 in the worst case.

Table II lists the asymptotic performance bounds for the applicative editor operators and shows the results of experiments designed to determine the expected-case space usage of both optimized and unoptimized operators. The experiments reveal that merging improves space utilization by roughly 45% in the expected case. A prototype editor using these primitives has been built. Its speed is comparable to that of "ed" under UNIX. With a megabyte of memory, a history of 1,750 to 3,500 versions of a 10,000 line text file can be maintained.

B. Applicative Arrays

An array is a fixed length list with the operator repertoire:

(1) SEL(A,k):X
Select the k^{th} element of array A.
(2) ASN(A,k,x):Array_of_X
Assign x to the k^{th} element of array A.

While arrays could be modeled applicatively by encoding them as lists, a more space efficient scheme is desirable as the length of arrays are frequently very large. (For example, consider a history system that maintains versions of a program's state modeled as an array of words.)

First consider modeling an array as a reference to a singly-linked list in which each cell encodes an element of the array as an <index,value> pair. ASN(A,k,x) is easy: push a cell containing the pair <k,x> onto the front of the linked list referenced by A. This is applicative (the original linked list is not modified) and requires only $O(1)$ time and space. SEL(A,k) is achieved by searching A for the first cell containing index k and returning the associated value. This requires time proportional to the length of the linked list which can, unfortunately, be made arbitrarily large through the repeated use of ASN operations.

The next refinement guarantees a worst-case time of $O(N)$ for SEL where N is the size of the array. Let each cell further contain an auxiliary <index,value> pair with the following properties. The auxiliary index of a cell is one more (modulo N+1) then the auxiliary index of its successor in the linked list. The auxiliary value of a cell v is SEL(v,a) where a is the auxiliary index of v. From these properties it follows that SEL(A,k) can be achieved by returning the value associated with the first auxiliary or regular index that matches k. Moreover, at most N cells must be searched before a cell with auxiliary index equal to k is found. The one drawback is that now ASN must engage in an $O(N)$ SEL in order to establish the auxiliary value of the cell it pushes onto the linked list. Thus this second approach requires $O(1)$ space and $O(N)$ time for *both* operators.

Finally, consider modeling an N-element array A[0..N-1] with a K-dimensional array B[0..W−1]···[0..W−1] and for simplicity assume $N = W^K$. Let the i^{th} element of A correspond to the i^{th} element in the lexicographical order of elements in B. That is, let $A[i] \equiv B[i_1][i_2]\cdots[i_K]$ where $i_1 i_2 \cdots i_K$ is the K-digit representation of i in the radix-W number system. Now view B as a W-element array of (K-1)-dimensional arrays, each of which is a W-element array of (K-2)-dimensional arrays, and so on. Applicatively model each of the W-element arrays using the simple method above. Observe that:

$$SEL(A,i) = B_K$$

where $B_J = SEL(B_{J-1}, i_J)$ for J = 1 to K
and $B_0 = B$

Thus a selection into the N-element array requires K selections into the W-element arrays for a total of $O(KN^{1/K})$ time. Further observe that:

$$ASN(A,i,x) = C_1$$

where $C_J = ASN(B_{J-1}, i_J, C_{J+1})$ for J = 1 to K
and $C_{K+1} = x$

Thus applicatively assigning an element in the N-element array requires K applicative assignments and selections into the W-element arrays for $O(KN^{1/K})$ time and $O(K)$ space.

Observe that K can be chosen arbitrarily. Space consumption is constant when K is fixed at a small integer, e.g. $K = 3$ gives $O(N^{1/3})$ time and $O(1)$ space. Logarithmic search time and less than $O(lgN)$ space are simultaneously attained by choosing $K = lgN / lg\ lgN$ (giving $O(lg^2N / lg\ lgN)$ time). When $K = lgN$ performance coincides with that of the list algorithms.

Acknowledgements

The author would like to thank Chris Fraser, Gary Levin, and David Hanson for their many helpful suggestions.

References

1. Adel'son-Vel'skii, G.M. and Landis, E.M. "An Algorithm for the Organization of Information." *Dokl. Akad. Nauk SSSR* 146 (1962), 263-266 (Russian). English translation in *Soviet Math. Dokl.* 3 (1962), 1259-1262.
2. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Comm. ACM* 21, 8 (1978), 613-641.
3. Broy, M. and Pepper, P. "Combining Algebraic and Algorithmic Reasoning: An Approach to the Schorr-Waite Algorithm." *ACM Trans. on Prog. Languages and Systems 4, 3 (1982), 362-381.*
4. Dobkin, D.P. and Munro, J.I. "Efficient Uses of the Past." *Proc. 21st ACM Symp. on Foundations of Computer Science* (1980), 200-206.
5. Hood, R. and Melville, R. "Real-Time Queue Operations in Pure LISP." *Inform. Process. Lett.* 13, 2 (1981), 50-54.
6. Knuth, D.E. *The Art of Computer Programming (Vol. 3) Sorting and Searching.* Addison-Wesley, Reading, Mass. (1973), 451-468.
7. Morris, J.H., Schmidt, E. and Wadler, P. "Experience with an Applicative String Processing Language." *Proc. 7th ACM Symp. on the Princ. of Prog. Languages* (1980), 32-46.
8. Myers, E.W. "An Applicative Random-Access Stack" *Inform. Process. Lett.* (to appear).
9. Myers, E.W. "AVL Dags" TR 82-9, Dept. of Computer Science, U. of Arizona, Tucson, AZ (1982).
10. Weizenbaum, J. "Symmetric List Processor." *Comm. ACM* 6, 9 (1963), 524-536.

Appendix A: The List Algorithms

This appendix contains a complete specification of the applicative list algorithms discussed in Sections 2 through 4. The following conventions insure that reference counts are correctly maintained: (1) every function returns a new (INCremented) reference to its result, and (2) every reference passed as an argument to a function is consumed (DECremented) by the function. In what follows, the abbreviation "@<list>" denotes the expression INC(<list>).

The list data abstraction is assumed to be implemented in a work space consisting of some sufficiently large, say DAGMAX, array of vertex records. A special record is set aside to model Λ. Its H-field is 0, its LN-field is 1, and its RC-field is initially 2*DAGMAX−1. At the outset, every other record is in a free list with its RC-field set to 0 and its L- and R-fields referencing Λ.

The work space is manipulated through the primitives INC, DEC, and NEW. The absence of cyclic substructures guarantees that a reference counter strategy suffices to detect all unused vertices. If a vertex v becomes free in a call to DEC, it is added to the free list. The processing of v's internal references does not proceed at this time but is deferred until v is reallocated in a call to NEW. At the time of realloca-

tion, v's offspring are collected but the processing of their internal references is again deferred. This strategy yields the following $O(1)$ *on-line* implementations of NEW, INC, and DEC:

```
Function INC(v: list) : list
1. v.RC ← v.RC+1
2. INC ← v

Procedure DEC(v: list)
1. v.RC ← v.RC-1
2. If v.RC = 0 Then Push v onto the free list

Function NEW(l: list; x: base; r: list) : list
   Var v: list
1. If free list is empty Then
2.    Abort : "Overflow"
3. Else
4.    Pop v from the free list
5.    DEC(v.L)
6.    DEC(v.R)
7.    NEW ← INC(v)
8.    v.V ← x
9.    v.L ← l
10.   v.R ← r
11.   v.H ← max(l.H,r.H)+1
12.   v.LN ← v.L.LN + v.R.LN
```

The functions BAL(l,x,r) and JOIN(l,x,r) perform the fundamental operation of concatenating AVL subtrees (dags) in a manner that preserves the height-balance property. Both return a list whose value is Val(l)*<x>*Val(r). BAL produces a height-balanced result whenever $|l.H-r.H| \leq 2$. JOIN uses BAL to produce a height-balanced result regardless of the heights of l and r.

```
Function BAL(l: list; x: base; r: list) : list
1. If l.H-r.H ∈ [-1,1] Then
2.    BAL ← NEW(@l,x,@r)
3. Else If l.H > r.H Then
4.    If l.L.H ≥ l.R.H Then
5.       BAL ← NEW(@l.L,l.V,NEW(@l.R,x,@r))
6.    Else
7.       BAL ← NEW(NEW(@l.L,l.V,@l.R.L),
                   l.R.V,NEW(@l.R.R,x,@r))
8. Else
9.    If r.R.H ≥ r.L.H Then
10.      BAL ← NEW(NEW(@l,x,@r.L),r.V,@r.R)
11.   Else
12.      BAL ← NEW(NEW(@l,x,@r.L.L),r.L.V,
                   NEW(@r.L.R,r.V,@r.R))
13. DEC(l,r)

Function JOIN(l: list; x: base; r: list) : list
2. If l.H-r.H ∈ [-2..2] Then
2.    JOIN ← BAL(@l,x,@r)
3. Else If r.H > l.H Then
4.    JOIN ← BAL(JOIN(@l,x,@r.L),r.V,@r.R)
5. Else
6.    JOIN ← BAL(@l.L,l.V,JOIN(@l.R,x,@r))
7. DEC(l,r)
```

The remaining algorithms constitute the operators of the applicative list data type. They are simply applicative adaptations of their procedural counterparts. However, note that the change in perspective has permitted these algorithms, considered difficult by many, to be tersely and clearly expressed.

```
Function SEL(v: list; k: integer) : base
1. If k < v.L.LN Then
2.    SEL ← SEL(@v.L,k)
3. Else If k > v.L.LN Then
4.    SEL ← SEL(@v.R,k-v.L.LN)
5. Else
6.    SEL ← v.V
7. DEC(v)

Function RNK(v: list; x: base) : integer
1. If v = Λ Then
2.    RNK ← 0
3. Else If x < v.V Then
4.    RNK ← RNK(@v.L,x)
5. Else
6.    RNK ← v.L.LN + RNK(@v.R,x)
7. DEC(v)

Function ADD(v: list; k: integer; x: base) : list
1. If v = Λ Then
2.    ADD ← BAL(@Λ,x,@Λ)
3. Else If k < v.L.LN Then
4.    ADD ← BAL(ADD(@v.L,k,x),v.V,@v.R)
5. Else
6.    ADD ← BAL(@v.L,v.V,ADD(@v.R,k-v.L.LN,x))
7. DEC(v)

Function DEL(v: list; k: integer) : list
1. If v.L = Λ and v.R = Λ Then
2.    DEL ← @Λ
3. Else If k ≤ v.L.LN and v.L ≠ Λ Then
4.    If k = v.L.LN Then
5.       DEL ← BAL(DEL(@v.L,k-1),
                   SEL(@v.L,k-1),@v.R)
6.    Else
7.       DEL ← BAL(DEL(@v.L,k),v.V,@v.R)
```

```
8.  Else
9.     If k = v.L.LN Then
10.       DEL ← BAL(@v.L,SEL(@v.R,1),
                       DEL(@v.R,1))
11.    Else
12.       DEL ← BAL(@v.L,v.V,DEL(@v.R,k-v.L.LN))
13. DEC(v)

Function CON(l,r: list) : list
1.  If l.H ≤ r.H Then
2.     CON ← JOIN(DEL(@l,l.LN-1),
                     SEL(@l,l.LN-1),@r)
3.  Else
4.     CON ← JOIN(@l,SEL(@r,1),DEL(@r,1))
5.  DEC(l,r)

Function SUB(v: list; i,j: integer) : list

   Function Left(v: list; k: integer) : list
   1.  If v = Λ Then
   2.     Left ← @Λ
   3.  Else If k ≤ v.L.LN Then
   4.     Left ← JOIN(Left(@v.L,k),v.V,@v.R)
   5.  Else
   6.     Left ← Left(@v.R,k-v.L.LN)
   7.  DEC(v)

   Function Right(v: list; k: integer) : list
      · · · "Symmetric analog of Left" · · ·

1.  If j < i Then
2.     SUB ← @Λ
3.  Else If j < v.L.LN Then
4.     SUB ← SUB(@v.L,i,j)
5.  Else If i > v.L.LN Then
6.     SUB ← SUB(@v.R,i-v.L.LN,j-v.L.LN)
7.  Else
8.     SUB ← JOIN(Left(@v.L,i),v.V,
                     Right(@v.R,j-v.L.LN))
9.  DEC(v)
```