TR-511-96
Purely Functional Representations of Catenable Sorted Lists.

Haim Kaplan[1] Robert E. Tarjan[2]

Abstract

The power of purely functional programming in the construction of data structures has received much attention, not only because functional languages have many desirable properties, but because structures built purely functionally are automatically *fully persistent*: any and all versions of a structure can coexist indefinitely. Recent results illustrate the surprising power of pure functionality. One such result was the development of a representation of double-ended queues with catenation that supports all operations, including catenation, in worst-case constant time [21].

This paper is a continuation of our study of pure functionality, especially as it relates to persistence. For one purposes, a purely functional data structure is one built only with the LISP functions car, cons, cdr. We explore purely functional representations of sorted lists, implemented as finger search trees. We describe three implementations. The most efficient of these achieves logarithmic access, insertion, and deletion time, and double-logarithmic catenation time. It uses one level of structural bootstrapping to obtain its efficiency.

The bounds for find,insert, and delete are the same as the best known bounds for an ephemeral implementation of these operations using finger search trees. The representations we present are the first that address the issues of persistence and pure functionality, and the first for which fast implementations of catenation and split are presented. They are simple to implement and could be efficient in practice, especially for applications that require worst-case time bounds or persistence.

[1]Department of Computer Science, Princeton University, Princeton, NJ 08544 USA. Research supported by by the NSF, Grant No. CCR-8920505, the Office of Naval Research, Contract No. N00014-91-J-1463 and a United States-Israel Educational Foundation (USIEF) Fulbright Grant. hkl@cs.princeton.edu.

[2]Department of Computer Science, Princeton University, Princeton, NJ 08544 USA and NEC Institute, Princeton, NJ. Research at Princeton University partially supported by the NSF, Grant No. CCR-8920505 and the Office of Naval Research, Contract No. N00014-91-J-1463. ret@cs.princeton.edu.

1 Introduction

A *finger search tree* is a type of balanced search tree in which access in the vicinity of certain preferred positions, indicated by *fingers*, is especially efficient. Finger search trees were introduced by Guibas, McCreight, Plass and Roberts [17] and further developed by many other researchers [4, 18, 22, 32, 33, 34].

A common type of finger search tree, called a *heterogeneous finger search tree* in [31], is an ordinary balanced search tree (like an a,b-tree for example) in which each node along the left path points to its parent instead of its left child, and each node along the right path points to its parent instead of its right child. Access to the tree is by two fingers pointing to the leftmost and rightmost external nodes.

By maintaining a sorted list in a heterogeneous finger search tree, one can search for the dth item in the list in $O(\log(\min\{d, n-d\}+1)+1)$ time. Insertion,deletion, or splitting at the dth item takes $O(\log(\min\{d, n-d\}+1)+1)$ amortized time. To catenate a list L_1 with n_1 elements and a list L_2 with n_2 elements takes $O(1+\log(\min\{n_1, n_2\}+1))$ amortized time. For the analysis see [31, 20].

A couple of versions of finger search trees have been designed for which the bounds mentioned above for insert and delete are worst-case instead of amortized [32, 33, 34, 22, 18].

A *persistent* data structure is one in which a change to the structure can be made without destroying the old version, so that all versions of the structure persist and can be accessed or (possibly) modified. In the functional programming literature, persistent structures are often called *immutable*. Purely functional[1] programming, without side effects, has the property that every structure created is automatically fully persistent (accessible and modifiable). Persistent data structures arise not only in functional programming but also in text, program, and file editing and maintenance; computational geometry; and other algorithmic application areas. (See [8, 10, 11, 12, 13, 14, 15, 16, 19, 24, 25, 26, 27, 28, 29, 30].)

Several papers have dealt with the problem of adding persistence to general data structures in a way that is more efficient than the obvious solution of copying the entire structure whenever a change is made. In particular, Driscoll, Sarnak, Sleator, and Tarjan [13] described how to make pointer-based structures fully persistent using a technique called *node-splitting*, which is related to fractional cascading [9] in a way that is not yet fully understood. Dietz [12] described a method for making array-based structures persistent. Additional references on persistence can be found in those papers.

The general techniques in [12] and [13] fail to work on data structures that can be combined with each other rather than just be changed locally. Perhaps the simplest and probably the most important example of combining data structures is catenation of lists.

Consider the following operations for manipulating lists:
makelist(x): return a new list consisting of the singleton element x.
push(x, L): return the list that is formed by adding element x to the front of list L.
pop(L): return the pair consisting of the first element of list L and the list consisting of the second through last elements of L.

[1] For our purposes, a "purely functional" data structure is one built using only the LISP functions car, cons, cdr. Though we do not state our constructions explicitly in terms of these functions, it is routine to verify that our structures are purely functional.

inject(x, L): return the list that is formed by adding element x to the back of list L.
eject(L): return the pair consisting of the last element on list L and the list consisting of the first through next-to-last elements of L.
catenate(K, L): return the list formed by catenating K and L, with K first.

In accordance with convention, we call a list subject only to *push* and *pop* a *stack* and a list subject to all four operations *push*, *pop*, *inject*, and *eject* a *double-ended queue*, or *deque* (pronounced "deck").

A straightforward use of balanced trees gives a representation of persistent catenable deques in which an operation on a deque or deques of total size n takes $O(\log n)$ time. Driscoll, Sleator, and Tarjan [14] combined a tree representation with several additional ideas to obtain an implementation of persistent catenable stacks in which the k^{th} operation takes $O(\log\log k)$ time. Buchsbaum and Tarjan [6] used a recursive decomposition of trees to obtain two implementations of persistent catenable deques. The first has a time bound of $2^{O(\log^* k)}$ and the second a bound of $O(\log^* k)$ for the k^{th} operation, where $\log^* k$ is the iterated logarithm, defined by $\log^{(1)} k = \log_2 k, \log^{(i)} k = \log\log^{(i-1)} k$ for $i > 1$, and $\log^* k = \min\{i \mid \log^{(i)} k \leq 1\}$.

Recently, Kaplan and Tarjan [21] have shown how to implement persistent catenable deques in $O(1)$ worst-case time per operation. The structure used is recursive. The operations are implemented so that only **one** operation on the recursive substructure is needed for every **two** on the top level structure. Kaplan and Tarjan called this technique *recursive slow-down*. In independent work, Okasaki has obtained a persistent implementation of catenable stacks with a constant time bound per operation [23]. His structure is simpler than Kaplan and Tarjan's, but it has two technical drawbacks: It is not purely functional but uses memoization, and the time bound is amortized rather than worst case. It is still open whether his approach can be extended to the double-ended problem.

This paper is a continuation of the study of the power of functional programming and persistence and in particular efficient functional implementation of data structure combination such as catenation. We use the results of [21] and new ideas to implement a powerful set of operations on sorted lists. The main result of this paper is a data structure for representing sorted lists which supports a **purely functional** (and thus fully persistent) implementation of the following operations:
Find(x, L) : Find the element with key x in L.
Insert(x, L) : Insert an element with key x into the list L in its position according to the linear order.
Delete(x, L) : Delete the element with key x from L.
Catenate(L_1, L_2) : Return the list formed by catenating L_1 and L_2. We assume that the first element in L_2 is greater than the last element of L_1.
Split(x,L) : Return two lists, one containing all the elements in L with keys smaller than x, and the other containing all the elements with keys greater than x.

Our purely functional implementation has time bounds as good as those of any known ephemeral representation that support find, insert, and delete, namely $O(\log(\min\{d, n-d\}+1)+1)$ for an operation at the dth item. Catenation and Split are not explicitly discussed in any of the papers on finger search trees we have cited. We describe an implementation for splitting at the dth item that takes $O(\log(\min\{d, n-d\}+1)+1)$ time. Furthermore, using a bootstrapping idea similar to the one used by Buchsbaum and Tarjan in [6], we show how to implement catenate to run in $O(\log\log(n_s+1)+1)$ time, where n_s is the size of the smaller

list being catenated. We also believe that using the techniques developed here, the structures described in [32, 33, 34] could be modified to be purely functional and support fast catenation.

Using the data structure we describe one can implement a purely functional merge operation for sorted lists which runs in $O(m \log(\frac{n}{m}))$ steps on lists of sizes m and n with $m \leq n$. This matches the lower bound for any comparison-based algorithm for the problem. The details were described by Brown and Tarjan in [4]. Our structure gives a way to turn their implementation into a purely functional one. Another application is a version of purely functional heaps that support delete-min in constant time and delete/insert of the dth item in $O(\log(d))$ time.

We actually propose three different implementations in this paper. Our first representation, presented in Section 2, is a structure closely related to the noncatenable deque implementation presented in [21]. Its main advantage is its simplicity. This simplicity comes from the fact that a structure like the one in [21] is essential for the other representations we suggest as well. The second and the third representations are based on 2-3 finger search trees in which we relax the degree constraint on the spines and represent them as lists with a specific structure. The basic representation is presented in Section 3. The first two representations have similar performance. Find,delete,insert and split at the dth item take $O(\log(\min\{d, n-d\}+1)+1)$ time. Catenate takes $O(\log(n_s+1)+1)$ time, where n_s is the size of the smaller list being catenated. The third representation is similar to the second. The difference is in the way we represent the lists which are used to represent the spines. We show that if one of the first two structures is used to represent these lists then a faster implementation of catenation is obtained. Specifically, catenate runs in $O(\log\log(n_s+1)+1)$ time. The details of this data structure bootstrapping idea are in Section 4. In Section 5 we summarize and point out a very interesting relation between our data structures and redundant binary number systems.

2 Generalized deques

In this section we describe an implementation based on the the non-catenable deques of [21].

Let a *6-list* be a list that can contain no more then 6 elements. For the purposes of this section *prefix* and *suffix* will denote 6-lists. Consider the following recursive representation of a list L for elements from a universe U. If $|L| \leq 6$ it consists of a single suffix and/or a prefix of elements from U denoted by suffix(L) and prefix(L) respectively. Otherwise it is a triplet: a prefix of elements from U denoted by prefix(L), a list denoted by $c(L)$ each of whose elements is either a pair or a triple of elements from U, and a suffix of elements from U denoted suffix(L). L is ordered as follows. First are the elements of prefix(L) (ordered as they are in the prefix). Next are the elements stored in $c(L)$ ordered consistently with the order of $c(L)$. The elements of each tuple are ordered according to their position in the tuple. Last are the elements of suffix(L) (ordered as they are in the suffix).

Note that one can think of an element x in the i-th level list as 2-3 tree of height i with elements of the list stored at its leaves. If x is not stored in the prefix or suffix then it is a subtree of a higher-level 2-3 tree.

Let q be an element stored at some level of the recursive structure. If $q = (x_1, \ldots, x_r)$ is an r-tuple ($r \in \{2,3\}$) then it has $r-1$ keys stored with it. The i-th key is an element not less then any leaf in the tree represented by x_i and not greater any key in the tree represented by x_{i+1}. Suppose q is stored in the prefix or suffix of some level. Let $x_i, x_{i+1}, \ldots, x_j$ be the

elements stored at the leaves of q. In addition to the keys we store with q two elements defining the *range* covered by it, $q_l \leq q_r \in U$ such that $x_{i-1} \leq q_l \leq x_i$ and $x_j \leq q_r \leq x_{j+1}$.

A prefix or suffix is *green* if it has two to four elements. It is *yellow* if it has one or five elements. It is *red* if it has zero or six elements. Order the colors such that $red < yellow < green$. The color of a list $L \neq \emptyset$ for which $c(L) \neq \emptyset$ is defined as the minimum of its prefix and suffix colors. The color of a list $L \neq \emptyset$ for which $c(L) = \emptyset$ is defined as the minimum of its prefix and suffix colors if both are not empty. Otherwise it is the color of the one which is not empty.

The representation suggested above for a list L is actually a stack $S(L)$ in which the i-th element stores pointers to prefix($c^i(L)$) and suffix($c^i(L)$).

The representation of L that a persistent implementation will actually use is a stack $S'(L)$ constructed from $S(L)$ as follows: Every maximal sequence $s_i, s_{i+1}, \ldots, s_{i+k}$ of elements corresponding to yellow lists in $S(L)$ is replaced by a single element $s_{i,k}$ in $S'(d)$. $s_{i,k}$ will contain a pointer to a length k secondary stack containing the yellow lists $s_i, \ldots, s_{i+k}$, with s_i on top. This representation allows us to access the topmost non-yellow list in $S'(L)$ easily in the persistent setting.

We will maintain the following invariant for every list L:

Invariant 2.1 *If $c^i(L)$ is red list then $i > 0$ and there exists a green list $c^j(L)$, $j < i$, such that any list $c^k(L)$, $j < k < i$, is yellow.*

While doing operations we may create intermediate lists that violate Invariant 2.1.

Let L be a list that might be violating Invariant 2.1 by having its first non-yellow level red. The following *fix* step will restore the invariant.

algorithm FIX(L)
Let $x = c^i(L)$ be the topmost non-yellow list in $S(L)$.
If x is red make x green by doing at most two operations on $c(x)$
according to the following cases:
a) $c(x)$ is empty.
  a.1) $6 \leq |x| \leq 8$. Balance the prefix and the suffix
  such that there are two to four elements in each by
  moving elements from one to the other.
  a.2) $8 < |x| \leq 12$. If the prefix is of length 5-6
  then pack its last three elements in a tuple.
  Push the tuple into $c(x)$.
  If the suffix has length 5-6 then pack its first
  three elements in a tuple.
  Inject the tuple into $c(x)$.
b) $c(x)$ is not empty.
  If the prefix length is 0-1 then pop a tuple from $c(x)$ into the prefix.
  If the prefix length is 5-6 then push a tuple from the prefix into $c(x)$.
  If the suffix length is 5-6 inject a tuple from the suffix into $c(x)$.
  If the suffix length is 0-1 and $c(x)$ is not empty then
  eject a tuple from $c(x)$ into the suffix.

We continue with a description of the operations insert and catenate. The implementation of find is straightforward and the implementation of delete and split is provided in the Appendix.

Insert: In order to insert an item x into a list L first search for the root of the 2-3 tree whose range includes x. If x is not contained in the range of any tree then locate the tree whose upper bound is closest to x among all 2-3 trees in the structure. One can do the search by linearly traversing the prefixes and suffixes of the lists at the different levels. We start at the first level and continue in increasing level order. We go in first-to-last order in the prefixes and in last-to-first order in the suffixes.

Denote by P the 2-3 tree which was found. Assume it was found at the prefix or suffix of $c^i(L)$. Fix $c^i(L)$. Then, insert x into P. If the height of P did not increase as a result of the insertion the insert is finished by increasing the range of P to include x if that was not the case to begin with.

Suppose that the height of P has increased by inserting x into it. Its root has two children. Replace P in its buffer by the two trees obtained when discarding P's root. Define their ranges using P's range and the extra key stored at its root. The level containing P may have its color degraded. If such a degradation happens then fix $c^i(L)$.

Remark: $Push(x, L)$ is a special case of $insert(x, L)$ in which x is known to be less the the first element in L. $Inject(x, L)$ is a special case of $insert(x, L)$ in which x is known to be greater than the last element in L. We will use these special cases of insert to describe the implementation of catenate and split.

Catenate: Let L_1 and L_2 be the two lists to be catenated to obtain a list L. Let h_1 and h_2 be the number of levels in each of them respectively. Assume that $h_1 \leq h_2$. For each level i, $0 \leq i \leq h_1$ do the following. Push prefix($c^i(L_1)$) and suffix($c^i(L_2)$) into a temporary stack. Arrange the elements in suffix($c^i(L_1)$) and prefix($c^i(L_2)$) into a list of tuples (of length 2-3). The elements of suffix($c^i(L_1)$) should precede those in prefix($c^i(L_2)$). Fix $c^{i+1}(L_2)$. Push the tuples in the list into $c^{i+1}(L_2)$ in last-to-first order.

At the end we get to the bottom of L_1's stack. In the temporary stack the prefixes of the first h_1 levels of L_1 and the suffixes of the first h_1 levels of L_2 are stored in pairs such that the topmost pair consists of prefix($c^{h_1}(L_1)$) and suffix($c^{h_1}(L_2)$).

Initialize $c^{i+1}(L) = c^{i+1}(L_2)$. The following iteration will finish the catenation by fixing the top h_1 levels of L. Repeat the following step for i runing down from h_1 to 0 (when the temporary stack gets empty). Pop the temporary stack to get prefix($c^i(L_1)$) and suffix($c^i(L_2)$). Let $c^i(L)$ be a new list which has prefix($c^i(L_1)$) as its prefix, suffix($c^i(L_2)$) as its suffix, and $c^{i+1}(L)$ as its recursive list of tuples. Fix $c^i(L)$.

Purely Functional Implementation: We use $S'(L)$ in order to be able to perform the fix operation defined in section 2 in constant time. The result of the fix is a new list L' represented by a stack $S'(L')$. A detailed description of a fix step in a persistent setting appears in [21].

Inserting or deleting an item located at level i in a list L persistently will require:
1) Inserting or deleting an item from/to a height i 2-3 tree. This can be done persistently by path copying in $O(i)$ time.
2) A constant number of fix steps.
3) Copying at most i levels at the top of $S(L)$. Note that the prefixes and suffixes of the levels could be shared by the new version and L as long as they do not change.

Catenation of two lists the smallest of which has i levels may require $O(i)$ fix steps and $O(i)$ pushes, each of which takes constant time. Split on an item located at level i may also require $O(i)$ fix steps as well as splitting a 2-3 tree of height i. It is easy to verify that the latter operation can be done persistently in $O(i)$ time.

3 Modified 2-3 finger search trees

In this section we consider a representation based on 2-3 finger search trees in which the nodes on the left and right spines are allowed to have 1-4 children. Denote by *x-node* a node with x children and by $p(v)$ the parent of a node v in the tree.

Our modified 2-3 finger search tree is represented as a triplet which includes a *right spine*, a root, and a *left spine*.

A spine is represented as a list called a *spine list*. Each of its elements is either an x-node with $x \in \{1, 4\}$, a list storing a maximal sequence of consecutive 3-nodes, called a *3-list* or a list storing a maximal sequence of consecutive 2-nodes, called a *2-list*. Denote by *{2,3}-list* a list which is either a 2-list or a 3-list. Similarly, denote by *{1,4}-node* a node which is either a 1-node or a 4-node and by *{2,3}-node* a node which is either a 2-node or a 3-node.

The nodes are ordered within the {2,3}-lists according to their order on the spine, lower nodes preceding higher ones. The {2,3}-lists and the {1,4}-nodes are similarly ordered in the spine list. The *height of a {2,3}-list* is defined to be the height of its first node. The *keys* of such a list are the keys of the first node.

Trees hanging off the spines are represented as ordinary 2-3 trees.

The representation for the spine list and the {2,3}-lists should support the operations **makelist**(x), **push**(x, L), **pop**(L), **inject**(x, L), **eject**(L), **catenate**(K, L), described in Section 1. It should also support the following operation:
split(x, L): Assumes that x is an element in L. Returns two lists, one containing the elements which precede x in L, the other the elements which follow x in L.

Any representation which supports makelist,push,pop,inject,eject in $O(1)$ time, catenation in $O(\min\{|K|, |L|\})$ time, and split at the dth item in $O(\min(d, |L| - d))$ time suffices to achieve the bounds claimed in this section. For example we could use either the catenable or the noncatenable structure of Kaplan and Tarjan [21]. Naive persistent implementations for catenate and split can always be added to a structure which supports the operations push,pop,eject and inject. Moreover, if push,pop,eject and inject can be carried out in constant time then the bounds for catenate and split are as required. Note that one can traverse the lists of [21] in constant time per element.

In this section we will assume that one of the representations above is used both for the spine lists and the {2,3}-lists.

Remark: In the algorithms described below for the various operations a list is first traversed to locate a particular item and then split at that item. With a naive implementation for split one may wish to combine these two stages in order to reduce the constant factor in the running time.

We will maintain the following invariant about the right and left spines.

Invariant 3.1 *1) There are no two 4-nodes with a 3-list between them.*
2) There are no two 1-nodes with a 2-list between them.

In order to describe the implementation of the various operations we need to define the following two operations on a node in a spine. A *split* operation applies to a 4-node. It replaces it by two 2-nodes one of which remains on the spine. A *fuse* operation applies to a 1-node v. It fuses v with its sibling (That sibling is guaranteed to exist by Invariant 3.1). After the fuse v's

degree is 3 or 4. The degree of $p(v)$ decreases by one. Note that the operation split is defined both on nodes and on lists. It would be clear from the context which split is referred to.

Let S be a spine list in a representation of some list L. Let v be a node on the spine. If v is a {2,3}-node denote by y the {2,3}-list which contains it; otherwise, consider y to be empty. The operation $split(v, y, S)$ cuts S at v and returns two lists S_u and S_l. S_u contains the ancestors and S_l the descendants of v on the spine, correctly packed into {2,3}-lists and ordered according to their heights. Its definition follows.

algorithm SPLIT(v, y, S)
if $y \neq emptyset$ **then**
  $(y_u, y_l) :=$ split(v, y); $(S_u, S_l) :=$ split(y, S);
  if $y_u \neq \emptyset$ **then** push(y_u, S_u); **if** $y_l \neq \emptyset$ **then** inject(y_l, S_l)
else $(S_u, S_l) :=$ split(v, S)
endif
return(S_u, S_l)

Split/Fuse at v causes a change in the degrees of v and $p(v)$. Thus the partition of the list R which contains it into {2,3}-lists has to be fixed. A 1-node will be fused only when it is preceded by an optional 2-list and another 1-node on the spine. A 4-node will be split only when it is preceded by an optional 3-list and another 4-node on the spine. This implies that it will always be possible to fix the partition by doing a constant number of pushes and pops on {2,3}-lists. Node v could change its degree to 3 only when it is fused with a degree two sibling. In that case it could be preceded in S_u by a 2-list or a 1-node only. Similarly, it could change its degree to 2 only when it splits and in that case it could be preceded by a 3-list or a 4-node only. We assume that the above mentioned fixes to S_u are always carried out as part of the split/fuse operation.

A 2-node v is pushed (injected) into a spine list (or some piece of it) R by pushing it into a 2-list which appears first (last) on R. If there isn't such 2-list a new 2-list is created with v in it and pushed (injected) into S_u. A 3-node is pushed or injected similarly. In order to safely push a {1,4}-node v one has to guarantee first that by pushing v Invariant 3.1 will not be violated. The procedure *Fix&Push* pushes a {1,4}-node v into a spine list (or part of a spine list) R without violating Invariant 3.1. W.l.o.g. assume that R is part of a right spine list. Pseudo-code for this procedure is provided in the Appendix. Note that a regular push would do whenever one knows that a violation of 3.1 cannot occur.

Catenating two pieces of the spine list back together could also result in a violation of 3.1. The procedure *Fix&Catenate* catenates two lists R_1 and R_2, assumed to satisfy 3.1, so that the result also satisfies 3.1. W.l.o.g. assume that R_1 and R_2 are parts of a right spine list. Pseudo-code for Fix&Catenate is also provided in the Appendix.

The root needs to be split when its degree is three and one of its children (either on the spine or not) splits. Denote by v_1 and v_2 the two nodes obtained while splitting the root. Assume that v_1 roots a subtree of elements which are less than the elements in v_2's subtree. Both v_1 and v_2 have degree two. Node v_1 is injected into the left spine list and v_2 into the right spine list. The new root has degree two.

If the root has only two children, v_1 and v_2, and one of them, say v_1, has degree one and has to be fused then v_1 and v_2 are ejected from the spine lists then v_1 is fused. The node that results is the new root. If its degree is 4 it has to be split.

We continue with a description of the operations insert and catenate. The implementation of find is straightforward and the implementation of delete and split is provided in the Appendix.

Insertion: Let x be the element to be inserted into a list L. The spines are traversed to locate a node v which points to the 2-3 tree T into which x has to be inserted. If v is a {2,3}-node the {2,3}-list y which contains it is also located. This can be done by going up the right spine and the left spine simultaneously. On the right spine we look for an element z such that x is smaller than its rightmost key but greater than the rightmost key in its successor. On the left spine we look for an element z such that x is greater than its leftmost key but smaller than the leftmost key at its successor. If z is a 4-node then $v := z$ and $y := \emptyset$. Otherwise, z is a {2,3}-list containing v. We set $y := z$ and further traverse z in a similar fashion to locate v. W.l.o.g. assume v and y have been located on the right spine S. S is split at v into S_u and S_l by using the Split algorithm described earlier in this section.

We insert x into T using a regular insertion algorithm for 2-3 trees with one exception. If the height of T has to increase by one, the insertion algorithm instead of splitting the root and adding a new degree two root on top of it, returns two trees T_1, T_2 of the same height as T, and a key k. All the elements in T_1 are not greater than k and all the elements in T_2 are not less than k.

In case the insertion into T described above returns a single tree, a pointer to that tree is stored in v. Node v is pushed into S_u, and S_u is catenated with S_l.

The case in which the insertion returns two trees is slightly more complicated. Since v has to point to both of them its degree increases by one. The spine list has to be reconnected so that it contains only nodes of degrees 1 to 4 and satisfies Invariant 3.1. We use the operations Fix&Push and Fix&Catenate to reconnect the spine lists in one of the following cases according to the change in the degree of v.

$4 \rightarrow 5$: Split v into two nodes v_1,v_2. Node v_1 has degree 2; v_2 has degree 3 and stays on the spine. Push(v_2, S_u) and catenate(S_u, S_l).

$3 \rightarrow 4$: Fix&Push(v, S_u). Fix&Catenate(S_u, S_l). Note that only one of the two calls will actually fix.

$2 \rightarrow 3$: Push(v, S_u). Fix&Catenate(S_u, S_l).

Catenate: Let L_1 be the list to be catenated to the front of list L_2. Let T_1 and T_2 be the trees representing L_1 and L_2 respectively, with h_1 the height of T_1 and h_2 the height of T_2. Assume that $h_1 > h_2$; the case in which $h_1 < h_2$ is symmetric.

The right spine S of L_1 is traversed to locate an element z whose height is $h_2 + 1$. First an element z is located whose height is $\leq h_2 + 1$ and the height of its successor is greater than $h_2 + 1$. If z is a {1,4}-node then $v := z$; $y := \emptyset$. Otherwise z is a {2,3}-list $y := z$ and z is further traversed to locate v. After setting v and y, split(v, y, S) is performed to obtain S_u and S_l.

In order to obtain a tree representing the result, the root r_2 of T_2 must be added as the rightmost child of v. Care should be taken that the right spine list of the result will satisfy Invariant 3.1.

Node v's degree increases by one. The upper part of the new spine is fixed according to the following case analysis on the degree change of v.

$4 \rightarrow 5$: Split v into two nodes v_1,v_2 of degrees 3 and 2 respectively. Let v_2 be node which stays on the spine. Push(v_2, S_u).

$3 \rightarrow 4$: Fix&Push(v, S_u).
$2 \rightarrow 3$, $1 \rightarrow 2$: Push(v, S_u).

Next r_2, whose degree is either 2 or 3, is pushed into S_u. Let R be the right spine of T_2. Fix&Catenate(S_u, R) finishes the catenation.

Remark: Note that S_l is stored as the next-to-rightmost child of v. Doing this we get paths packed into lists in the 2-3 trees hanging off the nodes on the spine. These paths are converted into regular 2-3 nodes as they are traversed during later searches or updates. Alternatively, one can convert S_l to regular 2,3 nodes as part of the catenate procedure and keep the trees hanging of the spine as regular 2,3 trees.

We summarize this section with the following theorem. Its proof is straightforward and will be provided in the full version.

Theorem 3.2 *The list representation described in this section supports catenation in* $O(\log(\min\{n_1, n_2\}+1)+1)$ *time where* n_1 *and* n_2 *are the sizes of the two lists being catenated. The time bounds for find,split,insert and delete are* $O(\log(\min\{d, n-d\}+1)+1)$ *if the operation is carried out at the* d*th item.*

Purely Functional Implementation: In order to get a purely functional implementation of the modified 2-3 finger search trees described, the following ingredients are needed:
1. A purely functional implementation for the spine lists and the {2-3}-lists.
2. A purely functional implementation for the operations insert,delete and split on 2,3-trees.

The first can be found in [21]. All implementations discussed there are purely functional. Addition of naive split and catenate can also be done in a purely functional way. The second is achieved by implementing the operations on 2-3 trees so that the whole access path is copied.

Once we have these ingredients it is straightforward to check that the algorithms described in the previous sections can be implemented to be purely functional.

4 Faster catenation via bootstrapping

In this section we show how to modify the representation of Section 3 to get a faster implementation for catenation. Specifically, catenation will run in $O(\log\log(\min\{n_1, n_2\}+1)+1)$ time where n_1 and n_2 are the sizes of the two lists being catenated. The time bounds for find,insert, delete and split will not change.

The data structure is similar to the one described in Section 3. The main difference is that both the spine lists and the {2,3}-lists are represented either by the structure of Section 2 or the structure of Section 3 instead of the deques of [21]. The search keys of the elements are their heights. The lists represented in this way are faster to split but slower to catenate and allow efficient searches for nodes with particular heights.

The implementation of all the operations is similar to the one described in Section 3.

While catenating two lists represented by trees T_1 and T_2 of heights h_1 and h_2 respectively (W.l.o.g. assume $h_1 \geq h_2$) a node of height h_2+1 has to be located in a spine list of T_1. In Section 3 this search was performed by linearly traversing the relevant lists. The new representation of the lists allow us to carry out this search faster. First, an element with largest possible key which is no bigger than h_2+1 is located in the spine list by using the **find** operation with the key h_2+1 on this list. If a {1,4}-node is located then its height is

exactly $h_2 + 1$ and it is the required node. Otherwise, a {2,3}-list has been located. One more application of find on this list with the key $h_2 + 1$ will provide the required node. The rest of the procedure continues as in Section 3. Note that in order to obtain fast catenation one must allow nodes packed into spine lists in the internal 2,3 trees hanging off the spines. Immediate conversion of the lower part of a spine list as described in the last remark of the previous section will degrade the time bounds. The following theorem summarizes the result.

Theorem 4.1 *The lists representation described in this section supports catenation in* $O(\log\log(\min\{n_1, n_2\} + 1) + 1)$ *time where* n_1 *and* n_2 *are the sizes of the two lists being catenated. The time bounds for find, split, insert and delete are as in Theorem 3.2.*

Note that the data structure can be implemented purely functionally as described in Section 3.

5 Concluding Remarks

All representations we have described in this paper are based on 2-3 trees. We have used 2,3 trees to simplify the presentation. The data structures described could be modified to use any kind of a,b-trees.

Particularly interesting is a relation between the structures we have presented and redundant binary counting systems as described by Clancy and Knuth in [7]. Clancy and Knuth describe two counters. The first uses three digits 0,1,2. Any representation used must satisfy the requirement that any pair of 2 digits is separated by at least one 0 digit and the rightmost digit which is not 1 is 0. One can be added to a number in constant time using the following algorithm:
1. Add one by changing a rightmost 0 to 1 or x1 to (x+1)0
2. Fix the rightmost two by changing x2 to (x+1)0
This binary counter is equivalent to the work allocation mechanism used in the representation of Section 2, where red, yellow, green correspond to the digits 2,1,0 respectively. The second counter suggested in [7] uses 4 digits -1,0,1,2. The representations used must not contain consecutive 1's between 2's or consecutive 0's between -1's and must contain either a 0 or -1 to the right of the rightmost 2 and a 1 or 2 to the right of the rightmost -1. With this counter one can either increment or decrement a number in constant time using the following algorithm:
1. Add or subtract 1 as desired
2. Fix rightmost 2 or -1 by changing x2 to (x+1)0 or x(-1) to (x-1)1.
This counter is similar to the spine list and the way it is maintained in Section 3, where 1,2,3,4-nodes correspond to the digits -1,0,1,2 respectively.

Recently, in independent and distantly related work Brodal [1, 2] and Brodal and Okasaki [3] have designed heaps which can be melded in constant time. Interestingly, an essential ingredient in all the structures they describe is a redundant counter similar to the ones we use here.

It is an intriguing open problem whether one can design a sorted list representation that supports even faster catenation. In particular can sorted lists be catenated in constant time while the bounds for find, insert and delete are still $O(log(d))$ or at least $O(log(n))$, where the operation occurs at the dth item and the list is of size n ? We believe that the answer to this question is yes but have no solution yet. It is possible to use a version of the structure

suggested by Buchsbaum and Tarjan in [6] to obtain a representation for sorted lists with constant time for catenate, but the search time degrades to $O(\log(n)\log(d))$ for the dth item.

Acknowledgements

We thank Ian Munro for pointing out to us the related redundant binary counting systems.

References

[1] G. S. Brodal. Fast Meldable Priority Queues. WADS'95 proceedings, LNCS 955, pp.282-290.

[2] G. S. Brodal. Worst Case Priority Queues. To appear in SODA'96 proceedings.

[3] G. S. Brodal and C. Okasaki. Optimal Purely Functional Priority Queues. Unpublished manuscript 1995.

[4] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists, *Siam J. of Computing*, 1980, 594–614.

[5] A. L. Buchsbaum, R. Sundar and R. E. Tarjan. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues, *Proceedings of the 33rd IEEE Symp. on Foundations of Computer Science*, 1992, 40–49.

[6] A. L. Buchsbaum and R. E. Tarjan. Confluently persistent deques via data structural bootstrapping, *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, 155–164.

[7] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. STAN-CS-77-606, Computer Science Department, Stanford University.

[8] B. Chazelle. How to search in history, *Information and Control* **64** (1985), 77–99.

[9] B. Chazelle and L. J. Guibas. Fractional cascading: I. A. data structure technique. *Algorithmica*, **1**(2):133–62, 1986.

[10] R. Cole. Searching and storing similar lists, *Journal of Algorithms* **7** (1986), 202–220.

[11] D. P. Dobkin and J. I. Munro. Efficient uses of the past, *Journal of Algorithms* **6** (1985), 455–465.

[12] P. F. Dietz. Fully persistent arrays, Proceedings of the 1989 Workshop on Algorithms and Data Structures, Ottawa Canada, *Lecture Notes in Computer Science* 382, Springer-Verlag, 1989, 67–74.

[13] J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent, *Journal of Computer and Systems Science* **38** (1989), 86–124.

[14] J. Driscoll, D. Sleator, and R. Tarjan. Fully persistent lists with catenation. In *Proc. 2nd ACM-SIAM SYMP. on Discrete Algorithms*, pages 89–99, 1991. Submitted to J. ACM.

[15] M. Felleisen. The Theory and Practice of First-Class Prompts. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 180–190, 1988.

[16] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: a mathematical semantics for handling full functional jumps. *Proc. Conference on Lisp and Functional Programming*, 52–62, 1988.

[17] L. J. Guibas, E. M. McCreight, M. F. Plass and J. R. Roberts. A new representation for linear lists, *Proceedings of the 9th Annual ACM Symp. on Theory of comput.*, 1977, 49–60.

[18] S. Huddleston. An efficient scheme for fast local updates in linear lists, Dept. of Information and Computer Science, University of California, Irvine, CA, 1981.

[19] G. F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 158–168, 1988.

[20] K. Mehlhorn. Data structures and efficient algorithms, Volume 1: Sorting and Searching, Springer-Verlag, Berlin, 1984.

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

[22] S. R. Kosaraju. Localized search in sorted lists, in *Proc. 14th ACM Symp. on Theory of Computing*, pages 62-69, 1981.

[23] Chris Okasaki. Amortization, lazy evaluation and purely functional catenable lists, IEEE Symposium on Foundations of Computer Science, October 1995, pages 646-654.

[24] M. H. Overmars, Searching in the past, I. Technical Report RUU-CS-81-7, Department of Computer Science, University of Utrecht, Utricht, The Netherlands, 1981.

[25] M. H, Overmars, "Searching in the past, II: General Transforms," Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utricht, The Netherlands, 1981.

[26] T. Reps, T. Teitelbaum, and A. Demers, Incremental context-dependent analysis for language based editors, *ACM Transactions on Programming Languages and Systems* **5** (1983), 449–477.

[27] N. Sarnak. *Persistent Data Structures.* Ph.D. thesis, Department of Computer Sciences, New York University, 1986.

[28] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, **29**(7):669–79, 1986.

[29] D. Sitaram, M. Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*: An International Journal, **3** (1990), 67–99.

[30] Hans-Jörg Stoss. K-band simulation von k-Kopf-Turing-maschinen. *Computing*, **6**(3):309–317, 1970.

[31] R. E. Tarjan, and C. J. Van Wyk. An $O(n \log\log n)$-time algorithm for triangulating a simple polygon, *Siam J. of Computing*, 1988, 143–173.

[32] A. K. Tsakalidis. A simple implementation for localized search, Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece. 1990.

[33] A. K. Tsakalidis. A optimal implementation for localized search, A84/06 Fachbereich Angewante Mathematic und Informatik, Universitat des Saarlandes, Saarbrucken, West Germany, 1984.

[34] A. K. Tsakalidis. AVL-Trees for Localized Search, *Information and Control*, 67, 1985, 173-194.

A Appendix

A.1 Deletion and split for modified deques

In this section we describe the implementation of delete and split for the structure described in Section 2.

Delete: Let x be the item to be deleted from a list L. Locate the 2-3 tree, P, which includes x in its range. Let i be the level in which P is stored. Assume P is contained in $prefix(c^i(L))$ (the case in which it is located at the suffix is similar). Fix $c^i(L)$. Then, delete x from P. If the height of P did not change after deleting x then the deletion is done. Suppose that the height of P has decreased due to the deletion we split into one of the following cases:
1) If P is the first element in its prefix. Pop it from its prefix and inject it into the prefix of the level below. Fix $c^i(L)$ then fix $c^{i-1}(L)$.
2) Suppose P is not the first element in its prefix. Let P_l be P's left neighbor. a) If P_l has less two children then add P as P_l's rightmost child. Fix $c^i(L)$. b) If P_l has three children. Delete its rightmost child R. create a new node with R as its left child and P as its right child. Replace P in its prefix by this newly created tree.

Split: Let L be the list to be split at x to obtain L_l of elements less than x and L_r of elements greater than x. A search is made to locate the 2-3 tree, P, which includes x as a leaf, in the list L. Assume P is located at the prefix of $c^i(L)$ (The case in which P is located at the suffix is treated symmetrically). Split P at x and obtain two trees P_1 and P_2 whose heights are h_1 and h_2 respectively ($h_1, h_2 \leq i$). If there are elements before P in prefix($c^i(L)$) push them into a newly created prefix, p. Start constructing L_l by setting prefix($c^i(L_l)$) to be p and suffix($c^i(L_l)$) to either empty or to contain P_1 if $i = h_1$. Fix $c^i(L)$. Initialize $c^i(L_r) = c^i(L)$. Repeat the following step for each level, l running from $i-1$ down to 1. Create $c^l(L_l)$ by setting its prefix to be prefix($c^l(L)$) and its suffix to be either empty or to contain P_1 if $h_1 = l$. Fix $c^l(L_l)$. Create $c^l(L_r)$ by setting its suffix to be suffix($c^l(L)$) and its prefix to be either empty or to contain P_2 if $h_2 = l$. Fix $c^l(L_r)$.

A.2 Pseudo-code for Fix&Push and Fix&Catenate

algorithm FIX&PUSH(v, R)
/* The predicate *degree* applies to an element of a spine list and returns 1 or 4 */
/* if it is a 1-node or a 4-node respectively */
(x_1, R) = Pop(R)
if degree(v) = 1 **then**
  if x_1 is not a 2-list and degree(x_1) $\neq$ 1 **then** Push(x_1, R); Push(v, R); **return**; **endif**
  if x_1 is a 2-list **then**
    (x_2, R) = Pop(R);
    if degree(x_2) $\neq$ 1 **then** Push(x_2, R); Push(x_1, R); Push(v, R); **return**; **endif**
  endif
  /* At this point either degree(x_1) = 1 or x_1 is a 2-list and degree(x_2) = 1 */
  if degree(x_1) = 1 **then** $z = x_1$; $y = \emptyset$; **else** $z = x_2$; $y = x_1$; **endif**
  z = Fuse(z); Push(z, R); **if** $y \neq \emptyset$ **then** Push(y, R); **endif**
  Push(v, R);
else /* degree(v) = 4 */
  if x_1 is not a 3-list and degree(x_1) $\neq$ 4 **then** Push(x_1, R); Push(v, R); **return**; **endif**

```
  if x_1 is a 3-list then
    (x_2, R) = Pop(R);
    if degree(x_2) ≠ 4 then Push(x_2, R); Push(x_1, R); Push(v, R); return; endif
  endif
  /* At this point either degree(x_1) = 4 or x_1 is a 3-list and degree(x_2) = 4 */
  if degree(x_1) = 4 then z = x_1; y = ∅; else z = x_2; y = x_1; endif
  (z_1, z_2) = Split(z); Push(z_2, R); if y ≠ ∅ then Push(y, R) endif
  Push(v, R)
endif

algorithm Fix&Catenate(R_1, R_2)
x=Pop(R_1); y=Eject(R_2);
if x is a 2-list and y is a 2-list then
  x_1= Pop(R_1); y_1= eject(R_2);
  if both x_1 and y_1 are 1-nodes then x_1 = Fuse(x_1) endif
  Push(x_1, R_1); Inject(y_1, R_2); xy = Catenate(x, y); Push(xy, R_1); Catenate(R_1, R_2)
else
  if x is a 3-list and y is a 3-list then
    x_1= Pop(R_1); y_1= eject(R_2);
    if both x_1 and y_1 are 4-nodes then (z_1, x_1) = Split(x_1) endif
    Push(x_1, R_1); Inject(y_1, R_2); xy = Catenate(x, y); Push(xy, R_1); Catenate(R_1, R_2)
  else Push(x, R_1); Inject(y, R_2); Catenate(R_1, R_2)
  endif
endif
```

A.3 Deletion and split for modified 2-3 finger search trees

In this section we describe the implementation of delete and split for the structure described in Section 3.

Deletion: Let x be the element to be deleted from a list L. As in the implementation of insertion, the spines are traversed to locate a node v which points to the 2-3 tree T from which x has to be deleted. If v is a {2,3}-node the {2,3}-list y which contains it is located first. The spine is then split at v into S_u and S_l.

x is deleted from T using a regular deletion algorithm for 2-3 trees modified such that it always returns a tree of the same height as its input tree but it may return a tree in which the root has degree one.

If the deletion from T result in a valid 2-3 tree. A pointer to that tree has to be stored in v. v is pushed into S_u and S_u is catenated with S_l.

We now describe how to handle the case in which the deletion from T result in a tree with root r of degree one. W.l.o.g. assume that v is on the right spine. One of the following cases apply:

1. r has a left sibling z.

a) If z has degree three. Disconnect its rightmost child and make it the leftmost child of r. v is pushed into S_u and S_u is catenated with S_l.

b) If z has degree two. Fuse z and r to a node with three children. The degree of v decrease by one. The way by which the spine list is reconnected is described shortly.

2. r has no left sibling but a right sibling z. This case is exactly symmetric to the previous one. Special attention should be given to the case in which z is on the spine. If z's degree is

greater than two than by contributing a child it decreases its degree. v in this case does not change its degree. If z's degree is two then it is fused with r. The new node obtained is a degree three node on the spine replacing z just below v. v decreases its degree by one.

In any case at most one node on the spine decreases its degree by one. Denote that node w. Note that w is either v or its child on the spine. Assume that S_l and S_u have been fixed to contain all the spine nodes below w and above it respectively correctly partitioned into {2,3}-lists.

We continue according to the change of w's degree.
$4 \rightarrow 3$: Push(v, S_u). Catenate(S_u, S_l).
$3 \rightarrow 2$: Push(v, S_u). Fix&Catenate(S_u, S_l).
$2 \rightarrow 1$: Fix&Push(v, S_u). Fix&Catenate(S_u, S_l). Note that only one of the two calls could actually fix.

Split: Let L be a list to be split at an element x to obtain L_1 and L_2. As in deletion a node v on the spine which points to the tree T containing x is located. The spine is split at v to obtain S_u and S_l.

Let T_v be the subtree rooted at v. T_v is split at x using split algorithm for 2-3 trees.

Remark: T_v is represented by v and S_l. It should be converted to a regular representation first or the algorithm should be modified to handle it as it is represented (that might be the case anyway if fast catenation as described in Section 4 is implemented). It is easy to verify that the conversion could be carried out in time proportional to the length of S_l.

Let T_1 and T_2 be the two trees returned by the split. T_1 is the one with elements less than x.

T_2 stores exactly the elements in L_2. Its spines must be packed back into spine lists.

Let T' be the whole tree representing L without T_v. The list S_u is completed to be a valid right spine of T' as follows. Note that the degree of $p(v)$ is one less in T' than its degree at T. If its degree decrease from 2 to 1 there might be a violation to Invariant 3.1. It is fixed by poping $p(v)$ from S_u and pushing it back using Fix&Push.

At this point S_u satisfies Invariant 3.1. It has to be completed to a spine list of T' by packing the lower part of its rightmost spine (starting from $p(v)$'s rightmost child) into {2,3}-lists top-to-bottom while pushing them into S_u.

T_1 has to be converted from a regular 2-3 representation to the modified representation with spine lists. Then T' is catenated with T_1 to obtain a representation for L_1.