Purely Functional Worst Case Constant Time Catenable Sorted Lists

Gerth Stølting Brodal[1], Christos Makris[2], Kostas Tsichlas[2]

[1] Department of Computer Science, University of Aarhus,
BRICS, Basic Research in Computer Science, www.brics.dk,
funded by the Danish National Research Foundation.
e-mail address: gerth@brics.dk
[2] Department of Computer Engineering and Informatics, University of Patras,
26500 Patras, Greece.
e-mail address:{makri, tsihlas}@ceid.upatras.gr

Abstract. We present a purely functional implementation of search trees that requires $O(\log n)$ time for search and update operations and supports the join of two trees in worst case constant time. Hence, we solve an open problem posed by Kaplan and Tarjan as to whether it is possible to envisage a data structure supporting simultaneously the join operation in $O(1)$ time and the search and update operations in $O(\log n)$ time.
Keywords: data structures, sorted lists, purely functional programming

1 Introduction

The balanced search tree is one of the most common data structures used in algorithms and constitutes an elegant solution to the *dictionary* problem. In this problem, one needs to maintain a set of elements in order to support the operations of insertion, deletion and searching for the predecessor of a query element. In a series of applications [9, 10, 13, 16] search trees are also equipped with the operations of *join* and *split*. The most efficient search trees use linear space and support insertion, deletion, search, join and split operations in logarithmic time; the most prominent examples of them are: AVL-trees, red-black trees, (a, b)-trees, BB[α]-trees and Weight Balanced B-trees.

In commonly used data structures, update operations such as insertions and deletions change the data structure in such a way, that the old version (the version before the update) is destroyed. These data structures are called *ephemeral*. A data structure that does not destroy its previous versions after updates, is called *a persistent* data structure. Depending on the operations allowed by a persistent structure, the following types of persistence can be distinguished:

- *Partial* persistence: only the latest version of the structure can be updated and all the versions can be queried.
- *Full* persistence: all the versions can be updated and queried.

– *Confluent* persistence: all the versions can be updated and queried and additionally, two versions can be combined to produce a new version. Note that in this case it is possible to create in polynomial time an exponentially sized structure by repeatedly joining it with itself.

The history of a persistent data structure is represented by a *version graph* $G = (V, E)$; a node $v \in V$ corresponds to a version of the data structure while an edge $e \in E$ between nodes v and v' denotes that one of the structures involved in the operation creating v' was v. For the partial persistence case the version graph is a linear list, while for the full persistence case it is a tree. In the case of confluently persistent data structures the version graph is a directed acyclic graph.

Notably, persistent data structures are also met under the name *purely functional* data structures. This term indicates data structures built using operations that correspond to the LISP commands *car, cdr, cons*. These commands create nodes which are immutable and hence fully persistent. However, a full persistent data structure is not necessarily purely functional.

The problem of devising a general framework for turning ephemeral pointer-based data structures into their partial and full persistent counterparts was successfully handled in [7]. The proposed construction works for linked data structures of bounded in-degree and allows the transformation of an ephemeral structure into a partial or full persistent one with only a constant amortized time and space cost. The amortized bounds for the partial persistent construction were turned into worst case in [2]. Additionally, in [16] Okasaki presented simpler constructions by applying the lazy evaluation technique met in functional languages.

The aforementioned general techniques fail to apply in the confluent persistence setting; in this setting the version graph becomes a DAG making the navigation method of [7] to fail. Fiat and Kaplan presented efficient methods to transform general linked data structures to confluently persistent and have showed that if the total number of assignments is U then the update creating version v will cost $O(d(v) + \log U)$ and the space requirement will be $O(d(v) \log U)$ bits, where $d(v)$ is the depth of v in the version graph.

The aforementioned general framework was surpassed in practice by ad hoc solutions for specific data structures. In particular in [8, 4, 11, 12] a set of solutions for constructing confluent persistent deques was presented leading to an optimal solution that could handle every operation in worst case constant time and space cost. The supported set of operations included push, pop, inject, eject and catenate. One of the most interesting examples of purely functional data structure is sorted lists implemented as finger trees. Kaplan and Tarjan described in [13] three implementations, the more efficient of which achieved logarithmic access, insertion and deletion time, and double-logarithmic catenation time. In fact, they supported the search and update operations in $O(\log d)$ time, where d is the number of elements between the queried element and the smallest or the largest element in the tree. They asked whether the join operation can be implemented in $O(1)$ worst-case time even in an ephemeral setting,

while supporting searches and updates in logarithmic time. They sketched a data structure supporting the join operation in $O(1)$ time but the time complexity of search and update operations was $O(\log n \log\log n)$.

In this paper we focus on the problem of efficiently implementing purely functional sorted lists from the perspective of search trees and not finger trees as in [13]. We present a purely functional implementation of search trees that supports join operations in worst-case constant time, while simultaneously supporting search and update operations in logarithmic time. In Figure 1 we provide the complexities of our data structure and compare them with previous results. In Section 2, we introduce the reader to the problem as well as to some basic notions used throughout the paper. In Section 3, we present the main structural elements of the construction and depict how to make the structure purely functional, and finally we conclude in Section 4 with some final remarks.

	Traditional (e.g. AVL, (a,b)-trees)	Kaplan & Tarjan (STOC '96 [13])	This Paper
Search	$O(\log n)$	$O(\log n)$ - $O(\log d)$	$O(\log n)$
Join	$O(\log n)$	$O(\log\log n)$	$O(1)$
Insert/Delete	$O(\log n)$	$O(\log n)$ - $O(\log d)$	$O(\log n)$

Fig. 1. Comparison of the complexities of our data structure with previous results. Here n denotes the number of stored elements, while d denotes the number of elements between the queried element (defined by the insert, delete or search operations) and the smallest or largest element.

2 Definitions

A *biased tree* T [15], [1] is a leaf-oriented search tree storing elements equipped with weights. The *weight* $w(v)$ of a node v in T is the sum of the weights of all the leaves in its subtree. The *weight* $w(T)$ of the tree T is the weight of the root of T. The left (right) spine of the tree T is the path from the root of T to the smallest (largest) element of the tree.

In this paper, we consider the problem of maintaining a set of elements each of weight 1, represented as a collection of trees, subject to the following operations:

1. **Insert**(T_i,x), inserts element x in the tree T_i.
2. **Delete**(T_i,x), deletes the element x, if it exists, from tree T_i.
3. **Search**(T_i,x), returns the position of x in the tree T_i. If x does not exist in T_i, then it returns the position of its predecessor.
4. **CreateTree**(T_i,x), creates a new tree T_i with element x. A tree T_i ceases to exist when it has no elements.
5. **Join**(T_i,T_j), joins the two trees in one tree. Trees T_i and T_j are ordered in the sense that all elements of T_j are either smaller or larger than the smallest or largest element of T_i. Assume without loss of generality that

$w(T_i) \geq w(T_j)$. In this case, tree T_j is attached to tree T_i, and the result of this operation is the tree T_i. T_j is attached to a node on the spine of T_i.

There exist various implementations of biased trees differing in the used balance criterion; our construction is based on the biased $2, b$ trees presented in [1] which are analogous to 2,3 trees and B-trees. A $2, b$ tree is a search tree with internal nodes having degree at least 2 and at most b; in a biased $2, b$ tree the rank $r(v)$ of a leaf v is equal to $\lfloor \log w(v) \rfloor$, while the rank of an internal node is one plus the maximum of the ranks of its children. The rank $r(T)$ of the tree is the rank of the root. In [1] it is described how to maintain a $2, b$ tree so that accessing a leaf v takes $O(\log(w(T)/w(v))$ query time; inserting an item i takes $O(\log(w(T)/(w_{i-} + w_{i+})) + \log(w'(T)/w_i))$ time and deleting an item i takes $O(\log(w'(T)/(w_{i-} + w_{i+})) + \log(w(T)/w_i))$ time, where $w(T)$, $w'(T)$, are the weights of the tree before and after the operation and $i-$, $i+$ are the largest item smaller than i and the smallest item larger than i respectively. These bounds are achieved by maintaining a balance criterion termed *global bias*. A *globaly biased* $2, b$ tree is a $2, b$ tree with the property that any neighboring leaf of a node v whose parent w has rank larger than $r(v)+1$, has rank $r(w)-1$.

We will employ in our construction biased $2, 4$ trees, using the same definitions of weights, ranks and balance (global bias) as in [1].

3 The Fast Join-Tree

In this section we provide a description of our structure, called the *Fast Join-tree*. Initially we present an overview, introducing the building components of the structure and the way these various components are linked together. Then, we discuss how these components are combined when joining trees in order to produce a worst-case constant time implementation. Finally, we provide the necessary machinery for the update operations as well as the necessary changes that have to be performed in the structure in order to make it purely functional.

3.1 An Overview

The main goal of the proposed data structure is to support the join operation in $O(1)$ worst-case time. The main obstacle in achieving this complexity for the join operation is the location of the appropriate position on the spine (see Section 2, definition of Join operation). This problem can be overcome by performing joins in a lazy manner. In particular, if two trees T and T' such that $r(T) \geq r(T')$ are joined, then T' is inserted in a temporary structure along with other trees that have been joined with T. This temporary structure is called a *tree-collection*. Thus, a tree-collection is a set of elements structured as a forest of trees. During the insertion of trees in the tree-collection, the spine is traversed so that the tree-collection is finally inserted in the appropriate position. The tree-collection is implemented as a simple linked linear list of trees.

A tree-collection can be considered as a weighted element to be inserted in a tree structure. The weight of a tree-collection x is the number of leaves of

the trees in x. A tree-collection can be inserted in a fast Join tree in worst-case constant time by employing a preventive top-down rebalancing scheme on the spines.

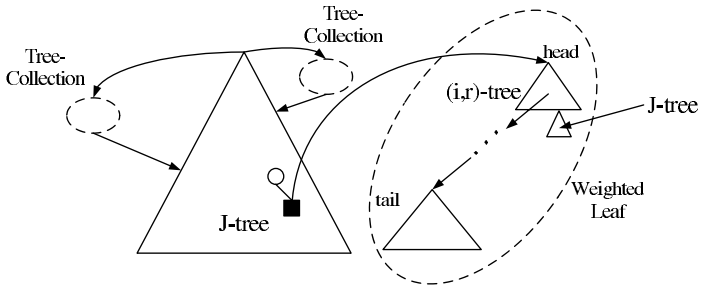

Fig. 2. The structure of the fast Join-tree at one recursive level.

The fast Join-tree is better understood as a recursive data structure. A biased search tree constitutes a recursive level of the fast Join-tree, called henceforth a *J-tree*. As a result, the fast Join-tree is a J-tree whose leaves are tree-collections. A tree-collection, as implied earlier, is a linear list of J-trees. These J-trees constitute the second recursion level. The recursion ends when the tree-collection is a simple item. Figure 2 depicts an instance of our structure.

3.2 The Biased Tree and the Tree-Collections

The J-tree is a biased $2,4$-tree with top-down rebalancing at its spines. This tree structure is subject to insertions of *weighted elements* at its spines as well as decrements by one of the rank of one of its leaves. A weighted element can be a simple item equipped with a weight or a tree-collection, which is a forest of trees on weighted elements.

A tree-collection is structured as a simple linear linked list. The insertions of new elements in the tree-collection take place always at the tail of the linked list. The virtual rank $vr(x)$ of a tree-collection x is the number of J-trees contained in x. Its real rank $r(x)$ is equal to the logarithm of the sum of the weights of the participating trees. It will become clear below how these quantities are related to each other. We describe the weighted insertion operation as a one step procedure, but it will in fact be implemented incrementally.

Assume a weighted element x_i with weight $w(x_i)$. This element must be inserted as a weighted leaf of the last (from top to bottom) node on the spine of the tree (either left or right) that has rank larger or equal to $\lfloor \log w(x_i) \rfloor + 1$.

During the traversal of the spine all nodes with 4 children are split, so when the weighted element is inserted there are no cascading splits on the path to the root. It may be the case that the spine is too short, meaning that the weighted element should be inserted at the spine deeper than the length of the spine.

In this case, unary nodes must be created in order to put the new element at the appropriate level. However, these nodes can be introduced on demand by attaching the leaf to the last node of the spine. If some other element is inserted with small weight then we attach it as leaf to the last node or introduce new nodes if the number of weighted elements attached to this node is 4.

Consider now an arbitrary leaf l. Our biased tree must support an operation that decreases the rank of the leaf by one. We call this operation *Demote(ℓ,T)*. This decrement by one of the rank of the leaf (which in term of weights is equivalent to reducing by half its weight) can be implemented by simply moving from this leaf to the root of the tree, and using similar techniques as those described in [1].

The following lemma summarizes the properties of the described biased tree.

Lemma 1. *There exists an implementation of a biased tree T, such that the tree has height $O(\log w(T))$ and supports the operations of insertion of a weighted element at its spines and demotion of a weighted element in $O\left(\log \frac{w(T)}{w(x_i)}\right)$ time, where x_i is the element inserted or demoted.*

Proof. The insert operation places the elements always at the correct level which is never changed by the insertion operation. Hence by a similar line of arguments as that in [1] we can conclude that T has height $O(\log w(T))$. From the insertion algorithm an element x_i is inserted at a level such that the path from the root to this level has length at most $O(\log w(T) - \log w(x_i))$ which is equal to $O\left(\log \frac{w(T)}{w(x_i)}\right)$. Finally by using the analysis in [1] it is proved that the demote operation takes the same time complexity. □

3.3 The Join Operation

The fast Join-tree is a biased tree with one tree-collection attached to each of its spines and tree-collections at its leaves. The tree-collection is a weighted element that must be inserted at the appropriate position on the spine of the biased tree. This insertion is incrementally implemented during the future join operations. When the appropriate node for the tree-collection has been found, it is attached to this node as a weighted leaf and the process starts again from the root with a new and possibly empty tree-collection.

The tree-collection x_L (for the left spine) maintains a pointer p_L that traverses the spine of the biased tree T starting from its root v. Assume that T is involved in a join operation with some other tree T', such that $R = r(T) \geq r(T')$. Then, tree T' is inserted in the tree-collection x_L and p_L is moved one node down the spine. The choice of moving one node down the spine is arbitrary and any constant number would do. During the traversal of the spine a simple counter is maintained, which denotes the ideal rank of each node on the spine. This counter is initialized to R, and each time we traverse a node on the spine it is reduced by one. When a node is located such that the counter is equal to $r(x_L) + 1$ the tree-collection is inserted and the process starts again from the root with a new

tree-collection. The inserted tree-collection is inserted as a weighted leaf of this node.

Assume that T' and T are joined, where $r(T) \geq r(T')$, and that the tree-collection x_L points to a node u on the spine with rank $r(u)$. There are two cases as to the relation of $r(T')$ and $r(u)$:

1. $\lfloor \log(w(x_L) + w(T')) \rfloor + 1 < r(u)$: in this case the tree-collection x_L after the insertion of T' in it must be inserted somewhere down the spine hence the traversal must be continued.
2. $\lfloor \log(w(x_L) + w(T')) \rfloor + 1 \geq r(u)$: in this case the weight of T' introduced in x_L is too much and x_L should be inserted higher up the spine. If $\lfloor \log(w(x_L) + w(T')) \rfloor + 1 < r(u) + 1$ then x_L canbe safely inserted as a child of the father of node u. Otherwise, the tree-collection x_L without T' is made a child of u and a new tree-collection is created and initialized to tree T' that starts the traversal of the spine from the root. We call x_L the *stepchild* of node u.

For the second case the following lemma holds:

Lemma 2. $r(T') > r(x_L)$.

Proof. (by contradiction) Assume that $r(T') \leq r(x_L)$. Then, by the addition of T' the tree-collection will have rank at most $r(x_L) + 1$. However, this is not possible since x_L could have been attached without taken T' into account. □

A stepchild is a weighted element that was not inserted in its correct position but higher on the spine. We assume that the stepchild does not contribute to the out-degree of its father. The following property is essential:

Property 1. Each internal node of the fast Join-tree has at most one stepchild.

This property is a direct consequence of Lemma 2. Thus, node u is not anymore part of the spine and the property follows. A similar issue arises in the case when tree T' is merged with the larger tree T. As before, the two non-empty tree-collections of T' are made stepchildren of the nodes they currently point to. Tree T' is not part of the spine of T, thus Property 1 is maintained.

One problem that arises from the use of stepchildren is that Lemma 1 may not hold anymore. Fortunately, this is not the case because of Property 1 and the fact that the stepchildren cannot cause any splits on the spine.

The result of this discussion is that the tree-collections move only once the spine from root to leaf. The following property exploits this fact and relates the virtual rank $vr(x)$ of a tree-collection x with the corresponding pointer p with its rank $r(x)$ and the rank R of T:

Property 2. $R - r(x) > vr(x)$

Proof. We prove that $R - vr(x) > r(x)$. $R - vr(x)$ is the maximum rank of the node pointed by the pointer p of the tree-collection x. This is because, after $vr(x)$ insertions of trees in the tree-collection x, the pointer p has traversed $vr(x)$ nodes down the spine. Since, the tree-collection x has not been attached to any node yet, its rank must be less than the rank of the node pointed by p. The inequality follows and the property is proved. □

The join operation is performed in $O(1)$ worst-case number of steps since a constant number of nodes are traversed on the spine and a single insertion is performed in one tree-collection. We now move to the discussion of the search operation. The search starts from the root of the fast Join-tree T and traverses a path until a weighted leaf ℓ is reached. The search continues in the forest of J-trees in tree-collection ℓ. The search in a forest of J-trees is implemented as a simple scan of a linear linked list from head to tail. Note that the head of the list is the first element ever inserted in the tree-collection.

The following lemma states that searching in a tree-collection is efficient.

Lemma 3. *If the search procedure needs $O(p)$ steps to locate the appropriate J-tree T' in a tree collection in J-tree T, then the rank of T' is at most equal to the rank of T reduced by p.*

Proof. Assume that at some recursive level of detail the J-tree T has rank $R = r(T)$. In addition, let T' be the p-th tree in the tree-collection. Since T' is the p-th tree in the collection, its insertion must occured at the p-th step of the spine's traversal. Since when traversing the spine we visit nodes of reduced rank we get as a result that T', being the J-tree in the p-th position has rank smaller than T by at least p, and the lemma follows. □

The following theorem states the logarithmic complexity of the search operation by using Lemma 3.

Theorem 1. *The search operation in a fast Join-tree T is carried out in $O(\log w(T))$ steps.*

Proof. Assume that the search procedure enters a weighted leaf which is a child of a node u in the J-tree T. Additionally, assume that in the forest the search procedure explores the p-th J-tree T'.

We show that:

$$r(T') \leq \min\{r(T) - p, r(u)\} \tag{1}$$

Since the search has reached node u we get that $r(T') \leq r(u)$. This observation in conjunction with Lemma 3 proves Equation 1.

Equation 1 states that the rank of the search space is decreased by 1 after $O(1)$ steps. As a result, to find a single element in the J-tree T we need $O(\log w(T))$ steps. □

3.4 Supporting Update Operations

We now describe how the J-tree supports insertions and deletions of single elements. The update operations may cause the fast Join-tree to become unbalanced; thus rebalancing strategies must be employed in order to ensure that this is not the case. We first show how insertions are implemented and then we move to the case of deletions.

Insertions We implement insertions by using a two level data structure. The first level of the structure is the fast Join-tree while the second one is a traditional degree balanced $(2, 4)$-tree as described in [15]. Each leaf of the fast Join-tree is a degree balanced $(2, 4)$-tree. Hence, the structure can be seen as a forest of $(2, 4)$-trees over which a secondary structure is built to incorporate join operations. Consequently, the first level of the structure implements efficiently the join operation while the second level implements the insertion operation. Figure 3 depicts the structure.

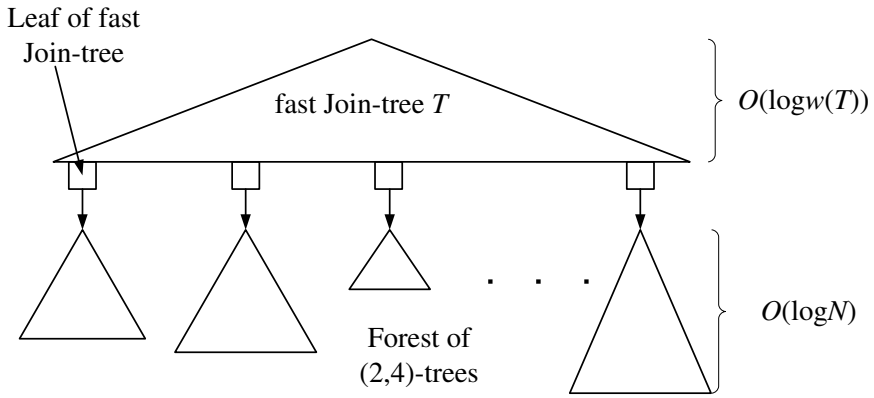

Fig. 3. A high level view of the structure for insertions.

When inserting an element we first need to locate its position in the structure. This is accomplished by searching the fast Join-tree for the appropriate leaf by using Theorem 1. The leaf represents a $(2, 4)$-tree in the second level of the structure, in which the element is inserted without affecting the fast Join-tree. This means that insertions do not affect the weights of the internal nodes of the fast Join-tree. The weight of the fast Join-tree is the number of leaves, that is the number of $(2, 4)$-trees in the second level.

When the position of the insertion is located in a $(2, 4)$-tree in the second level, the element is inserted. Finally, rebalancing operations are performed in the $(2, 4)$-tree and the insertion procedure terminates. Thus, the fast Join-tree is by no means affected by this insertion operation. As a result, there may be the case that a very light fast Join-tree has very large number of stored elements because of insertion operations, and we get the following property:

Property 3. The number of leaves of the fast Join-tree is a lower bound for the number of elements stored in the forest of $(2, 4)$-trees in the second level of the structure.

By Theorem 1, a leaf in a fast Join-tree T is located in $O(\log w(T))$ steps. When the leaf is located a second search operation is initiated in the $(2, 4)$-tree

attached to this leaf. If the number of elements stored in the forest of $(2,4)$-trees is N, then by Property 3 we get that $w(T) \leq N$. As a result, the total time complexity for the search operation is bounded by $O(\log w(T)) + O(\log N)$, which is equal to $O(\log N)$. Consequently, insertion operations are supported efficiently by applying this two level data structure.

Deletions The delete operation cannot be tackled in the same way as the insertion operation. This is because when a leaf of the fast Join-tree becomes empty, the weight of internal nodes is reduced by one. We devise a rebalancing strategy for the fast Join-tree in the case of deletions. Additionally, we introduce a *fix* operation, which incrementally joins leaves from the forest of $(2,4)$-trees.

As in the case of insertions, first the element must be located. The search procedure locates the appropriate leaf of the fast Join-tree and then locates the element in the $(2,4)$-tree attached to this leaf. This element is removed from the $(2,4)$-tree by applying standard rebalancing operations (fuse or share). If the $(2,4)$-tree is non-empty after the deletion, then the procedure terminates and the fix operation is initiated. If it is empty, then rebalancing operations must be forced on the fast Join-tree.

Since this leaf is empty it is removed from the fast Join-tree. If this leaf belonged to a tree-collection x then the virtual rank has decreased by one and potentially the rank of x has decreased by one. By employing a demote operation on the biased tree whose leaf is this tree-collection the change of the rank is remedied. This change may propagate up to the J-tree of the first recursive level. During this traversal, the weight of all nodes on the path to the root is updated accordingly. When the root of the fast Join-tree is reached, the deletion operation terminates. Note that this operation does not violate Property 2. This is because both the rank and the virtual rank of the tree-collection are reduced.

As shown above the deletion operation may reduce by one the virtual rank of the tree-collection. Thus, a tree-collection may be completely empty after a deletion operation. This means that the leaf is removed and rebalancing operations (fuse or share) must be performed in the biased tree to maintain its structural properties.

A final detail is how deletions interact with stepchildren. In this case, when a tree-collection is demoted then if one of its adjacent brothers is a stepchild, we also demote the stepchild. If the stepchild reached its correct level then it ceases to be a stepchild and it is inserted as an ordinary tree-collection. This procedure may be seen as an insertion of a weighted element in a J-tree, thus inducing rebalancing operations which may propagate up to the root of the fast Join-tree. The time complexity of the delete operation remains unaffected.

We now switch to the description of the fix operation. The fix operation is used as a means of globally correcting the data structure, that is it is not mandatory, however it is used to give a better shape to the structure. Each time an update operation is performed, the fix operation picks 4 leaves of the fast Join-tree and merges them together. Each one of these leaves represents a $(2,4)$-tree. The merge of these trees can be performed in $O(\log n)$ time, where n is the

number of elements stored in the forest of $(2,4)$-trees. From this merge three leaves of the fast Join-tree become empty. Thus, rebalancing operations must be employed on the fast Join-tree for these leaves. All in all, the time complexity for the delete operation is $O(\log n)$.

The problem posed by this delete operation is that Property 2 may be violated. As a result, the search bounds and consequently the update bounds will not be logarithmic anymore. Assume that N is the number of leaves of the fast Join-tree T before the fix operation is initiated, where $n \geq N$. Thus, the rank of T is $R = \log N$. During the fix operation, until all the leaves are merged into a single $(2,4)$-tree and the fast Join-tree is a single node, the number of elements in the forest of $(2,4)$-trees will never decrease below $\frac{n}{2}$. By Theorem 1, the time complexity for the search will be $O(R) + O(\log n)$, and since $R = O(\log n)$ the time complexity for the deletion follows.

By Theorem 1 and the previous discussion, we get the following theorem:

Theorem 2. *There exists a search tree supporting search and update operations in $O(\log n)$ worst case time and meld operations in worst case constant time.*

3.5 Purely Functional Implementation

The nodes that compose our structure have all degrees that are bounded by a constant. Hence, in order to make our structure work efficiently in a purely functional setting we need the following ingredients:

- a purely functional implementation of the linear list with which we implement the tree collection. This structure is implemented purely functionally in [3, 16].
- a purely functional implementation of the left and right tree spines. The presence of the pointer designating the movement of the tree-collection is quite awkward, since in a functional setting explicitly assigning values is not permitted. However, the pointer movement can be modeled by partitioning each spine into two lists the border of which designates the pointer position. These lists should be implemented purely functionally and should support both catenation and split operations. Details of such an implementation can be found in [14].

By Theorem 2 and the previous discussion, we get:

Theorem 3. *There exists a purely functional implementation of search trees supporting search and update operations in $O(\log n)$ worst case time and join operations in worst case constant time.*

4 Conclusion

We have presented a purely functional implementation of catenable sorted lists, supporting the *join* operation in worst case constant time, the *search* operation in $O(\log n)$ time and the insertion and deletion operations in $O(\log n)$ time. This

is the first purely functional implementation of search trees supporting the *join* operation in worst case constant time.

It would be very interesting to implement efficiently the *split* operation. It seems quite hard to do this in the proposed structure because of the dependence of Property 2 on the rank of the tree. Splitting will invalidate this property for every tree-collection and will lead to $(\log n \log \log n)$ search and update times. It would also be interesting to come up with an efficient purely functional implementation of sorted lists, implemented as finger trees (as in [13]) that could support join in worst case constant time. In this structure, it is quite unlikely to implement finger searching due to the relaxed structure of the fast Join-tree.

References

1. Bent, S., Sleator, D., Tarjan R. Biased Search Trees, *SIAM Journal of Computing* 14:545–568, 1985.
2. Brodal, G.S. Partially Persistent Data Structures of Bounded Degree with Constant Update Time, *Nordic Journal of Computing*, 3(3):238-255, 1996.
3. Brodal, G.S., and Okasaki, C. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6(6):839-857, 1996.
4. Buchsbaum, A. and R. E. Tarjan. Confluently persistent deques via data structural bootstrapping. *Journal of Algorithms*, 18:513-547, 1995.
5. Dietz P.F. Fully Persistent Arrays. *In Proc. of Workshop on Algorithms and Data Structures (WADS)*, pp. 67-74, 1989.
6. Dietz, P. and Raman, R. Persistence. Amortization and Randomization. *In Proc. of the 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 78-88, 1991.
7. Driscoll, J.R., Sarnak, N., Sleator, D., and Tarjan, R.E. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86-124, 1989.
8. Driscoll, J.R., Sleator, D., and Tarjan, R.E. Fully Persistent Lists with Catenation. *Journal of the ACM*, 41(5):943-959, 1994.
9. Fiat, A., and Kaplan, H. Making Data Structures Confluently Persistent. *Journal of Algorithms*, 48(1):16-58, 2003.
10. Kaplan, H. Persistent Data Structures. *Handbook of Data Structures*, CRC Press, Mehta, D., and Sahni, S., (eds.), 2004.
11. Kaplan, H., Okasaki, C., and Tarjan, R. E. Simple Confluently Persistent Catenable Lists. *SIAM Journal of Computing*, 30(3):965-977, 2000.
12. Kaplan, H., and Tarjan, R.E. Purely Functional, Real-Time Deques with Catenation. *Journal of the ACM*, 46(5):577-603, 1999.
13. Kaplan H., and Tarjan, R.E. Purely Functional Representations of Catenable Sorted Lists. *In Proc. of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, 202-211, 1996.
14. Kaplan, H., and Tarjan, R.E. Persistent Lists with Catenation via Recursive Slowdown. *In Proc. of the 27th Annual ACM Symposium on Theory of Computing*, pp. 93-102, 1995.
15. Mehlhorn, K. Data Structures and Algorithms 1: Sorting and Searching. *EATCS Monographs on Theoretical Computer Science*, Springer-Verlang, 1984.
16. Okasaki, C. Purely Functional Data Structures, Cambridge University Press, 1998.
17. Okasaki, C. Purely Functional Random-Access Lists. *In Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 86-95, 1995.