Union-copy structures and dynamic segment trees

Marc J. van Kreveld, Mark H. Overmars

RUU-CS-91-5
February 1991

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Union-copy structures and dynamic segment trees

Marc J. van Kreveld, Mark H. Overmars

Technical Report RUU-CS-91-5
February 1991

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

Union-Copy Structures and Dynamic Segment Trees*

Marc J. van Kreveld Mark H. Overmars

February 13, 1991

Abstract

A new data structure – the union-copy structure – is introduced, which generalizes the well-known union-find structure. Besides the usual union and find operations, the new structure also supports a copy operation, that generates a duplicate of a given set. The structure can enumerate a given set, find all sets that contain a given element, insert and delete elements, etc. All these operations can be performed very efficiently. The structure can be tuned as to obtain different trade-offs in the efficiency of the different operations.

As an application of the union-copy structure we introduce a dynamic version of the segment tree. Contrary to the classical semi-dynamic segment trees, the dynamic segment tree is not restricted to a fixed universe, from which the endpoints of the segments must be chosen. The tree allows for insertions, splits and concatenations in $O(\log n)$ time each. Deletions can be performed in slightly more time.

1 Introduction

A well-known data structure problem is the union-find problem, introduced by Tarjan [13]. A data structure for the union-find problem stores a collection of disjoint sets $\{S_1, \ldots, S_n\}$, such that the following two operations can be performed efficiently: UNION(S_i, S_j), which takes two (disjoint) sets as its arguments, and unites them into a single set, and FIND(x), which takes an element x as its argument, and returns the name of the set that contains x. Tarjan showed that, starting with n sets which all contain one unique element, a mixed sequence of $n-1$ union operations and $m \geq n$ find operations can be performed in $\Theta(m \cdot \alpha(m,n))$ time (where $\alpha(m,n)$ is the the extremely slowly growing functional inverse of Ackermann's function). Later, Tarjan showed that this result is optimal in the pointer machine model [14] (see also [9]). Since then, the union-find problem has received considerable attention (see e.g. [3, 4, 8, 9, 12, 16]), and several different solutions and variations have been described.

*Authors address: Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. This research was partially supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM).

In this paper we study a generalization of the union-find problem. The problem can be formulated as follows. Given a collection of (possibly not disjoint) sets $\{S_1, \ldots, S_n\}$ and a collection $\{x_1, \ldots, x_m\}$ of elements, construct a data structure that supports the following operations:

SET-UNION(S_i, S_j)	given two disjoint sets, unite them into a single set.
SET-COPY(S_i, S_j)	given a set S_i and an empty set S_j, turn S_j into an exact copy of S_i.
ELEMENT-FIND(x)	return all sets that contain x.

The structure should also support *dual* operations. Dual in the sense that the roles of sets and elements are reversed. Such operations are ELEMENT-UNION(x_i, x_j), which takes two elements that don't share a set, and turns them into a single element that is member of all the sets x_i or x_j were in, ELEMENT-COPY(x_i, x_j), which copies an element and SET-FIND(S) which enumerates all the elements in set S. To make updates the structure should support the operations SET-CREATE(S) which creates an empty set S, SET-DESTROY(S) which empties a set S (and removes it) and SET-INSERT(S,x) which inserts element x into set S. Also the dual operations should exist. For a precise description of the operations, see the next section. Note that a structure for the normal union-find problem only supports the operations SET-UNION and ELEMENT-FIND.

The structure can be visualized as a representation of a bipartite graph with two node sets V_1 and V_2 and edges running between a node of V_1 and a node of V_2 only. The nodes in V_1 form the sets, the nodes in V_2 from the elements. An edge between an element and a set indicates that the element is a member of the set. The operations can now be described as follows:

SET-CREATE	create a new node in V_1.
SET-INSERT	add an edge between a node in V_1 and a node in V_2.
SET-DESTROY	remove a node from V_1 together with its edges.
SET-FIND	return all nodes in V_2 that are connected to the node in V_1.
SET-UNION	take two nodes in V_1 (not connected to the same node in V_2) and join them into a single node.
SET-COPY	make a copy of a node in V_1 connected to the same nodes in V_2.

The dual operations are exactly the same for nodes in V_2. Note that it is not possible to delete an individual edge. This would highly increase the time bounds.

For these operations we introduce the union-copy structure, which is based on a mixture of two distinct union-find structures for its implementation. The efficiency of the operations depends on the union-find structures chosen. For example, using the structure of Tarjan [13] we obtain the following performance: SET-CREATE and ELEMENT-CREATE, SET-INSERT and ELEMENT-INSERT (which are in fact the same), SET-UNION and ELEMENT-UNION all take constant time, SET-FIND, ELEMENT-FIND, SET-DESTROY and ELEMENT-DESTROY take time $O(1 + k \cdot \alpha(m,n))$, where k is the number of reported answers or deleted edges, and SET-COPY and ELEMENT-COPY take time $O(\alpha(m,n))$. Another version achieves constant time SET-COPY operations at the

cost of an increase in the time bound for ELEMENT-UNION. A precise statement of the results is given in Theorem 1.

Union-copy structures are useful, for instance, for persistent set maintenance. If we make a copy of a set prior to each update, we can always retrieve and use old versions of the sets. With the appropriate implementation, such a copy takes only constant time and constant additional storage.

Another application lies in the maintenance of bipartite graphs, describing the relations between two types of data, e.g. suppliers and products or funds and tasks. The operations are as indicated above. Union operations occur when funds or tasks get merged. Copy operations are necessary when funds or tasks get split.

A third application of union-copy structures is in dynamic graph algorithms. The sets and elements of the structure both correspond to the vertices, where an element vertex w is in a set vertex v when (v, w) is an edge of the graph. The copy operations corresponds to multiplication of vertices (see [5]); add a new vertex and let it have all neighbors of some specified vertex. A union operation on two vertices results in one new vertex, which has all the neighbors of the former two vertices. It can be performed on two vertices of distance at least three, because then their sets of neighbors are disjoint.

One important application, that will be described in detail in this paper, is an almost fully dynamic version of the segment tree. Segment trees were introduced by Bentley in [2], and have been used for solving a variety of problems in computational geometry. A segment tree stores a set of possibly overlapping segments on a line and can answer stabbing queries efficiently (i.e., which segments contain a given query point). In the classical description of segment trees (see e.g. [11]), a finite subset $U = \{x_1, \ldots, x_N\}$ (universe) of $\mathbb{R}$ is defined, from which all endpoints of the segments must be chosen. Such a segment tree, storing n segments, requires $O(n \log N)$ storage, updates take $O(\log N)$ time, and stabbing queries can be performed in $O(k+\log N)$ time, where k is the number of segments reported.

In applications where N – the size of the universe – is $O(n)$, these bounds are good, but when N is considerably larger than n, or when U is not known at all, the classical solution is not applicable. Using the union-copy structure, we develop dynamic segment trees that can store any collection of segments. Our structure requires $O(n \log n)$ space. Furthermore, insertions take $O(\log n)$ time, deletions take $O(\log n \cdot a(i, n))$ time, where $a(i, n)$ is related to the functional inverse of Ackermann's function (see the appendix for a precise definition), and stabbing queries take $O(k + \log n)$ time. Our structure also allows for split and concatenate operations, which take $O(\log n)$ time each. This dynamic segment tree, using the split and concatenate operations, has recently been proven useful for solving a motion planning problem (see [7]).

The paper is organized as follows. In Section 2 we give a precise description of the operations we wish to perform, and describe the union-copy structure and the algorithms in detail. The complexity of the operation depends on the choice of two underlying union-find structures. Theorem 1 summarizes the main result of the paper. In Section 3 we describe the dynamic segment tree and the operations that can be performed on it. Finally, in Section 4 we give some conclusions and directions for further research.

2 Union-copy structures

In this section we will give a precise desciption of the operations we wish to perform on a union-copy structure, and describe the data structure and algorithms that carry out the operations.

Let $\mathcal{S} = \{S_1, \ldots, S_m\}$ be a collection of sets. Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a collection of elements. The sets in $\mathcal{S}$ are subsets of $\mathcal{X}$ (or are empty). Elements in $\mathcal{X}$ are contained in zero or more sets in $\mathcal{S}$. Let $\mathcal{S}_x$ denote the collection of sets that contain element x. Our goal is to represent $\mathcal{S}$ and $\mathcal{X}$ such that the following operations can be supported efficiently:

SET-CREATE(S)	Create a new, empty set in $\mathcal{S}$ with name S.
SET-INSERT(S_i,x_j)	Make $S_i \leftarrow S_i \cup \{x_j\}$. (Defined only when $x_j \notin S_i$.)
SET-DESTROY(S_i)	Remove S_i from $\mathcal{S}$.
SET-FIND(S_i)	Report all elements in S_i.
SET-UNION(S_i,S_j)	This operation is only defined when S_i and S_j are disjoint. Its effect is that $S_i \leftarrow S_i \cup S_j$ and $S_j \leftarrow \emptyset$.
SET-COPY(S_i,S_j)	This operation is only defined when S_j is empty. It makes S_j equal to S_i (i.e., it duplicates S_i).
ELEMENT-CREATE(x)	Create a new element in $\mathcal{X}$ with name x, and $\mathcal{S}_x \leftarrow \emptyset$.
ELEMENT-INSERT(x_i,S_j)	Make $\mathcal{S}_{x_i} \leftarrow \mathcal{S}_{x_i} \cup \{S_j\}$. (Defined only when $x_i \notin S_j$.) Note that this is exactly the same operation as SET-INSERT(S_j,x_i).
ELEMENT-DESTROY(x_i)	Remove x_i from $\mathcal{X}$, and, hence, from all sets that contain it.
ELEMENT-FIND(x_i)	Report all sets that contain x_i. (I.e., report $\mathcal{S}_{x_i}$.)
ELEMENT-UNION(x_i,x_j)	This operation is only defined when no set contains both x_i and x_j. It puts x_i in all sets in $\mathcal{S}_{x_j}$ and removes x_j from all those sets. Hence, after the operation $\mathcal{S}_{x_i}$ has become $\mathcal{S}_{x_i} \cup \mathcal{S}_{x_j}$ and $\mathcal{S}_{x_j}$ has become empty.
ELEMENT-COPY(x_i,x_j)	This operation is only defined when x_j is in no set. It puts x_j in all sets containing x_i, i.e., $\mathcal{S}_{x_j}$ becomes $\mathcal{S}_{x_i}$.

Note that the ELEMENT- operations are the exact duals of the SET- operations. As we shall see, also the data structure will be symmetric. This data structure we will call a union-copy structure.

2.1 The structure

The union-copy structure basically consists of four ingredients: set nodes, element nodes, normal nodes and reversed nodes. The set nodes represent the different sets in $\mathcal{S}$. The element nodes represent the different elements in $\mathcal{X}$. The normal and reversed nodes connect sets with the elements they contain and the elements with their sets. Normal nodes branch from sets towards elements and reversed nodes branch from elements towards sets. For the ease of description, we say that edges between nodes are always directed from sets towards elements (although we represent them by two pointers such

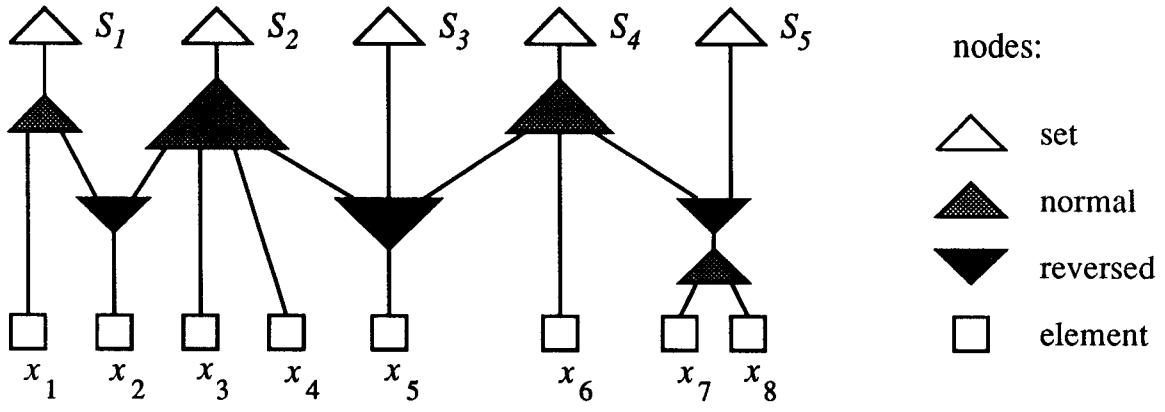

Figure 1: A union-copy structure

that we can traverse edges in both directions). The whole network forms a directed acyclic graph. Whenever there is a path from a set node to an element node, the corresponding element is in the corresponding set. If an element is in a set we will take care that there is one unique path from the set node to the element node. We impose the following restrictions on this structure. A set node has one outgoing edge. An element node has one incoming edge. A normal node has one incoming edge, and at least two outgoing edges. A reversed node has one outgoing edge and at least two incoming edges. No edge may go from a reversed node to another reversed node, and finally, no edge may go from a normal node to another normal node. Consequently, any path in the union-copy structure from a set node to an element node consists of an alternating sequence of normal and reversed nodes. See Figure 1 for an example. In this example e.g. the set $S_4 = \{x_5, x_6, x_7, x_8\}$ and x_5 is element of S_2, S_3 and S_4, i.e. $\mathcal{S}_{x_5} = \{S_2, S_3, S_4\}$. Note that if we disregard the direction of the edges, the structure is completely symmetric.

The implementation of the union-copy structure consists of a collection of set nodes, a collection of element nodes, a union-find structure denoted N-UF, and a union-find structure denoted R-UF. Every set node in the collection stores the name of the set, and one pointer for the outgoing edge. Similarly, every element node in the collection stores the name of the element, and one pointer for the incoming edge. The structure N-UF represents the normal nodes, where the sets in N-UF correspond one-to-one to the normal nodes. The elements of N-UF correspond one-to-one with the outgoing edges of these normal nodes. Similarly, the structure R-UF represents the reversed nodes and the incoming edges of these nodes. To avoid confusion between sets of the union-copy structure and sets of N-UF and R-UF, the latter will simply be called normal nodes (resp. reversed nodes). Furthermore, the elements of N-UF and R-UF will be called edges rather than elements. Notice that the edges are actually objects in the structure. An edge e from a node v (parent) to a node w (child) in the union-copy structure stores the type of v, denoted ptype(e), and the type of w, denoted ctype(e). The type of any node u is either set, element, normal, or reversed.

The following relation with union-find structures exists. A union-copy structure with only set nodes, element nodes and normal nodes is a union-find structure in the usual way, where a normal node represents the relation between its parent (a set node) and its children (outgoing edges to element nodes). A union-copy structure with only set nodes, element nodes and reversed nodes is also a union-find structure, but now the elements play the role of the sets, and the sets play the role of the elements (the dual relation of sets and elements).

Since we have the structure N-UF to represent the normal nodes, it is possible to let one normal node v take over all outgoing edges of another normal node w. This operation is simply a union in N-UF, and is denoted by N-UNION(v, w). Such a union clearly is a disjoint union. Furthermore, for an edge e in the union-copy structure with ptype(e) = normal, it is possible to retrieve the parent node by the operation N-FIND(e). If for an edge e ptype(e) = reversed or set, parent(e) is a primitive function which returns this parent node. In a similar way, the structure R-UF represents the reversed nodes, and the operations R-UNION and R-FIND operate the same way on the reversed nodes.

Note that the implementation of N-UF and R-UF need not to be the same. In fact, in our main application, it will be important to use different structures. There are though some restrictions we have to impose upon the structures. First of all, it must be possible to add an edge (element) to a particular node (set) in constant time. Secondly, the deletion of an edge should be possible in constant time without increasing time bounds for future operations, and without actually retrieving the node to which the edge belongs. Thirdly, it should be possible for an edge in the R-UF structure to decide in constant time to whether this edge is the only incoming edge of the (reversed) node. Finally, given a normal or reversed node, it should be possible to enumerate all its edges in time linear in their number. Known union-find structures, as in [8, 13, 16], satisfy these restrictions. We call these operations on N-UF N-INSERT, N-DELETE and N-ENUMERATE, and similar for R-UF.

2.2 The operations

We are now ready to describe the different operations in detail. We will only describe the SET- operations. The ELEMENT- operations are the exact duals. All operations turn out to be simple and straightforward. To descibe the procedures we use the following notations. For a set node representing a set S_i we denote by e_i the outgoing edge (being **nil** if the set is empty). For a normal node v we denote by parent(v) the one incoming edge. By children(v) we denote the set of outgoing edges (that can be found using N-ENUMERATE(v)). Similarly, for a reversed node v we denote by child(v) the one outgoing edge and with parents(v) the set of incoming edges. If ctype(e) is normal or element we let Child(e) denote this child node. When ctype(e) is reversed we can retrieve the child using R-FIND(e) (which, depending on the implementation, might take more than constant time). Similarly, when ptype(e) is reversed or set we let Parent(e) denote this parent node. If ptype(e) is normal we can use N-FIND(e) for this.

To indicate the use of this information, let us first describe the procedure SET-FIND which reports all elements in a set S_i. The idea is that, starting with e_i we traverse the structure, only going from parents to children, until we reach the element nodes. If we

reach a normal node in this process we recur at all its children. If we reach a reversed node we continue at its only child. The following piece of pseudo-code describes this process in detail:

```
SET-FIND(S_i)
    if e_i ≠ nil then TRAVERSE(e_i)

TRAVERSE(e)
    case ctype(e) of
        element:  report the element as an answer
        normal:   forall e' ∈ children(Child(e)) do TRAVERSE(e')
        reversed: TRAVERSE(child(R-FIND(e)))
```

Note that the part of the structure we traverse is a multiway tree with every other node having at least two children (because normal and reversed nodes alternate). As a result, the number of nodes we visit is of the same magnitude as the number of answers found. This will be important in the analysis of the complexity, provided in the next subsection.

The operation SET-UNION takes two set nodes and unites them into a single set. The sets have to be disjoint. The procedure checks the types of the children of the nodes and takes appropriate actions. If the sets are empty or contain a single element some simple edge operations suffice. If both children are normal nodes we unite them. If one is normal and the other reversed we simply add the reversed node as a son to the normal node. If both are reverse we create a new normal node with the two reversed nodes as children. It can easily be seen that in this way the properties of the union-copy structure are maintained. See Figure 2 for an example.

```
SET-UNION(S_i,S_j)
    if e_i = nil or ctype(e_i) ≠ normal then exchange e_i and e_j
    if e_j = nil then ready
    elsif ctype(e_i) = normal and ctype(e_j) = normal then N-UNION(Child(e_i),Child(e_j))
    elsif ctype(e_i) = normal then N-INSERT(Child(e_i),Child(e_j))
    else { both are reversed or elements}
        make a normal node v
        N-INSERT(v,e_i); N-INSERT(v,e_j)
        Child(e_i) ← v; e_j ← nil
```

The SET-COPY operation also tests the type of the child of the set to be copied. To a reversed node, a new parent is added for the new set. If the child is a normal node, then a new reversed node is made with the two sets as parents. See Figure 2 for an example.

```
SET-COPY(S_i,S_j)
    if e_i = nil then ready
    elsif ctype(e_i) = reversed then R-INSERT(R-FIND(e_i),e_j))
    else { ctype(e_i) = normal or element }
        make a reversed node v
        child(v) ← e_i
        R-INSERT(v,e_i); R-INSERT(v,e_j)
```

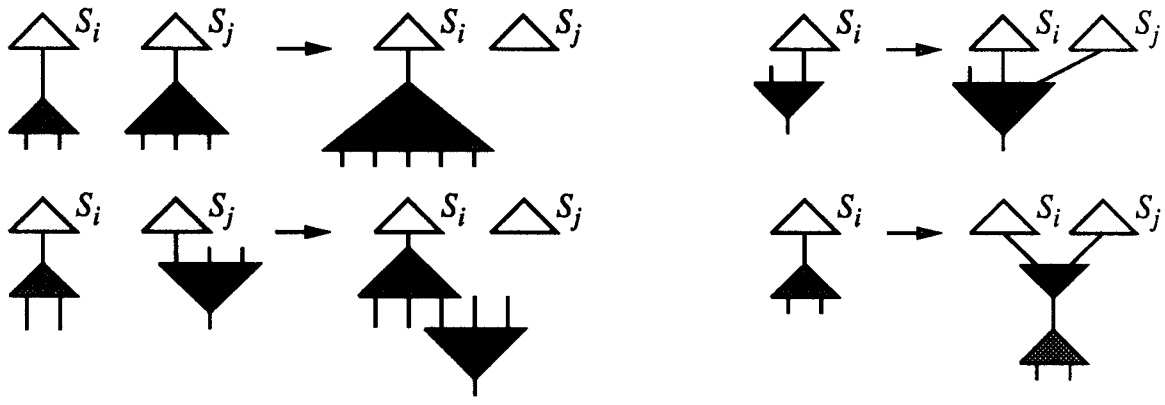

Figure 2: SET-UNION (left) and SET-COPY (right)

The operation SET-CREATE is trivial. We simply create a new set node with the given set name and a **nil** pointer. The SET-INSERT operation is also simple. It checks the types of the child of the set node and the parent of the element node, and takes actions accordingly. Let e_j denote the incoming edge of the element node.

SET-INSERT(S_i,x_j)
  if e_i = **nil then** $v \leftarrow$ the set node
  elsif ctype(e_i) = normal **then** $v \leftarrow$ Child(e_i)
  else { ctype(e_i) = reversed or element}
    make a normal node v
    N-INSERT(v,e_i)
    Child(e_i) $\leftarrow v$
  { now v is a normal or set node }
  if e_j = **nil then** $w \leftarrow$ the element node x_j
  elsif ptype(e_j) = reversed **then** $w \leftarrow$ Parent(e_j)
  else { ptype(e_j) = normal or set}
    make a reversed node w
    R-INSERT(w,e_j)
    Parent(e_j) $\leftarrow w$
  { now w is a reversed or element node }
  make an edge between v and w

Finally we describe the SET-DESTROY operation, which removes the set node and its outgoing edge. The actions performed for deletions are the straightforward actions to restore the properties of the union-copy structure (see Figure 3). There are a number of cases to identify. Firstly, when the outgoing edge of the set goes to a reversed node, this node might be left with only one parent, which is not allowed. Hence, we have to remove this reversed node as well. This can be done by uniting its only child with its only parent. If child and parent are both normal nodes, we enumerate the children of the child and add them to the parent (to make sure that no normal node is child of

another normal node). Secondly, when the edge goes to a normal node we remove this node, together with all its outgoing edges. This again might cause reversed nodes to loose a parent, and similar actions are taken. The following piece of code describes the procedure in detail. See also Figure 3.

```
SET-DESTROY(S_i)
    if e_i = nil then ready
    case ctype(e_i) of
        element:  remove the edge e_i
        reversed: RESTORE(e_i)
        normal:   forall e ∈ children(Child(e_i)) do
                      if ctype(e) = element then remove the edge e
                      else { ctype(e) = reversed } RESTORE(e)
    remove S_i


RESTORE(e) { ctype(e) = reversed }
    if Child(e) has more than two parents then
        R-DELETE(e) from the reversed node at Child(e)
    else { Child(e) looses one of its two parents and must be removed}
        v ← Child(e)
        R-DELETE(e) from the reversed node v
        e' ← the one remaining parent of v
        if ptype(e') = set or ctype(child(v)) = element then
            Child(e') ← Child(child(v))
        else { ptype(e') = normal and ctype(child(v)) = normal }
            w ← N-FIND(e')
            forall e'' ∈ children(Child(child(v))) do
                N-INSERT(w, e'')
            remove Child(child(v))
        remove v
```

The first test in the procedure RESTORE requires only constant time because of the restrictions we posed on the R-UF structure. As we shall see in the next section, the number of N-FIND operations is bounded by the size of the set S_i.

This finishes the description of the SET- operations. As noted above, the ELEMENT- operations are the exact duals (i.e., switch reversed and normal nodes, parent and child, R-FIND and N-FIND, etc.). Therefore, we won't give a detailed description.

2.3 The analysis

The time bounds of the different operations clearly depend on the choice of the N-UF and R-UF structure. Let $U_N(n)$ and $F_N(n)$ denote the time taken for a union and a find in the N-UF-structure with at most n elements. Similarly, $U_R(n)$ and $F_R(n)$ are defined. Let $|S_i|$ denote the number of elements in the set S_i. k denotes the number of elements or sets involved in a particular operation. I.e., for a SET-FIND the number of elements reported and for a SET-DESTROY the number of elements in the set.

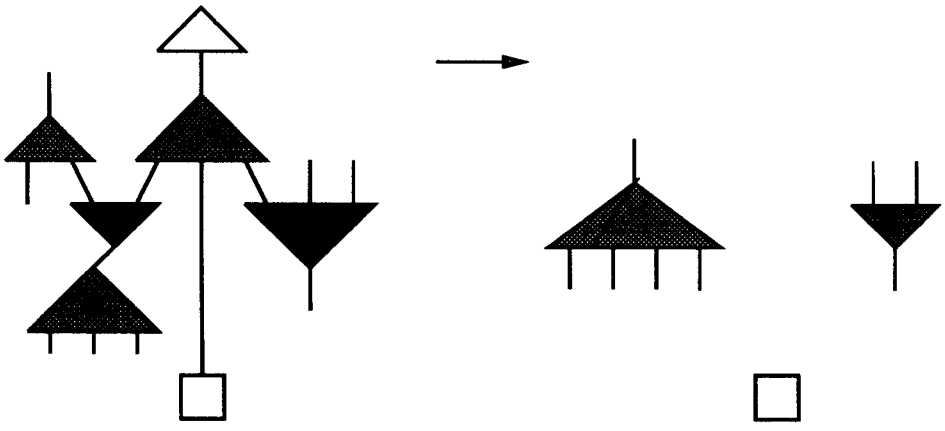

Figure 3: The operation SET-DESTROY

Theorem 1 *Given a collection $\mathcal{S}$ of m sets, and a collection $\mathcal{X}$ of n elements. A union-copy structure, based on the N-UF-structure and the R-UF-structure, has the following performance:*

SET-CREATE	$O(1)$	ELEMENT-CREATE	$O(1)$
SET-INSERT	$O(1)$	ELEMENT-INSERT	$O(1)$
SET-DESTROY	$O(1 + k \cdot F_N(n))$	ELEMENT-DESTROY	$O(1 + k \cdot F_R(m))$
SET-FIND	$O(1 + k \cdot F_R(m))$	ELEMENT-FIND	$O(1 + k \cdot F_N(n))$
SET-UNION	$O(U_N(n))$	ELEMENT-UNION	$O(U_R(m))$
SET-COPY	$O(F_R(m))$	ELEMENT-COPY	$O(F_N(n))$

where k is the size of the set in SET-DESTROY *and* SET-FIND*, or the number of occurrences of the element in* ELEMENT-DESTROY *and* ELEMENT-FIND*. Assuming that the N-UF-structure and R-UF-structure require linear space, the union-copy structure requires $O(n + \sum_{i=1}^{m} |S_i|)$ space.*

Proof. Most of the time bounds stated above are clear from the pseudo-code procedures that implement them, since any normal node has outdegree at most n, and any reversed node has indegree at most m. Hence, the operations N-FIND and N-ENUMERATE operate on sets of size $O(n)$, and R-FIND and R-ENUMERATE operate on sets of size $O(m)$. We prove the bounds for SET-FIND and SET-DESTROY.

To prove the time bound for SET-FIND, recall that normal and reversed nodes alternate on a path in the union-copy structure. If there are k answers to SET-FIND, then at most $k-1$ normal nodes and $2k-1$ reversed nodes have been visited by the procedure. By the assumption that N-ENUMERATE takes $O(k')$ time for a normal node of degree k', we have that the total time spent on normal nodes is $O(k)$. Clearly, the time used by the procedure to pass a reversed node is $O(F_R(m))$. Hence, at most $O(1 + k \cdot F_R(m))$ time is taken by SET-FIND.

The bound for SET-DESTROY can be proved in a similar way. The procedure RESTORE takes $O(k' + F_N(n))$ time if the normal node Child(child(v)) has k' children. Therefore,

at most $O(1 + k \cdot F_N(n))$ time is taken for all calls to RESTORE, and hence, for SET-DESTROY. □

3 Dynamic segment trees

In this section the union-copy structure will be used to solve a seemingly unrelated problem. It shows that union-copy structures have the potential to be advantageous in various problems. The application we describe is in dynamic segment trees.

The classical description of the segment tree is the following (see e.g. [11]). Given a universe $U = \{x_1, x_2, \ldots, x_N\}$ of points on the real line. The segment tree is a balanced binary tree, which stores in its leaves the *elementary intervals* of U, being $(-\infty : x_1), [x_1 : x_1], (x_1 : x_2), [x_2 : x_2], \ldots, (x_N : \infty)$ (round brackets denote open intervals, square brackets denote closed intervals). An internal node δ of the segment tree corresponds to an interval I_δ, where I_δ is the union of the intervals of the leaves in the subtree rooted at δ.

The segment tree can store a set S of n segments (or intervals), of which the endpoints are chosen from the universe U. Any such segment s is stored at those nodes δ of the segment tree, for which I_δ is contained in s, but $I_{\text{father of } \delta}$ is not contained in s. An immediate consequence is that any segment is stored at most twice on each level of the tree, and the (disjoint) union of I_δ for all nodes δ that store s, is precisely s itself.

The queries for which a segment tree is generally used are so-called *stabbing queries*. Such a query takes a real value y as its argument, and reports all segments in the segment tree that contain y. The search follows the path from the root of the segment tree to that leaf, of which the elementary interval contains y. All segments stored at the nodes on the path to this leaf are reported. Notice that these segments are precisely the segments of S that contain the query value y.

Segment trees are generally known as semi-dynamic data structures. New segments may only be inserted if their endpoints are chosen from the restricted universe. In this section we introduce segment trees that do not have this restriction. Segments may be inserted and deleted freely, and furthermore, split and concatenate operations are provided. The structure rests on two ideas. Firstly, the use of the union-copy structure to represent all associated structures, and secondly, the introduction of the weak segment tree. For the ease of description, we only consider closed segments. However, our solution also allows for open segments or half closed segments without any problem. We do allow segments to have endpoints with the same value.

Let T, T_1 and T_2 be dynamic segment trees, each storing a set of closed segments over the real numbers. Let $[x_1 : x_2]$ be an arbitrary segment, and let y be an arbitrary real number. As a slight abuse of notation, we let a dynamic segment tree T not only stand for the tree, but also for the set of segments it stores. The following operations are provided for by dynamic segment trees.

CREATE(T)	Defined only if T does not exist yet. This procedure creates an empty tree T, i.e. a tree consisting of one leaf that stores the elementary interval $(-\infty : \infty)$.
INSERT($T, [x_1 : x_2]$)	Defined only if $[x_1 : x_2] \notin T$. $[x_1 : x_2]$ is inserted into T.
DELETE($T, [x_1 : x_2]$)	Defined only if $[x_1 : x_2] \in T$. $[x_1 : x_2]$ is deleted from T.
CONCATENATE(T_1, T_2, T)	Defined only if the greatest (right) endpoint of a segment in T_1 is less than the smallest (left) endpoint of a segment in T_2, and T is empty. It makes $T \leftarrow T_1 \cup T_2$; T_1 and T_2 are returned empty.
SPLIT(T, T_1, T_2, y)	Defined only if for all segments $[x_1 : x_2]$ in T, either $y < x_1$ or $x_2 \leq y$ holds, and T_1 and T_2 are empty. It makes $T_1 \leftarrow \{[x_1 : x_2] \mid x_2 \leq y\}$; $T_2 \leftarrow \{[x_1 : x_2] \mid y < x_1\}$; T is returned empty.
STABBING-QUERY(T, y)	All segments $[x_1 : x_2]$ in T are reported for which $x_1 \leq y \leq x_2$.

Notice that CONCATENATE and SPLIT are inverse operations. Let $S = \{[x_1 : x_2], \ldots, [x_{2n-1} : x_{2n}]\}$ be a set of n segments, where the endpoints are chosen from the real numbers, and let $y_1 \ldots y_m$ be the ordered sequence of their distinct endpoints. S defines the following canonical partition of the real line into the elementary intervals $(-\infty : y_1), [y_1 : y_1], (y_1 : y_2), \ldots, (y_m : \infty)$. With the following definition, the description of our dynamic segment tree diverges from the traditional segment tree.

Definition 1 *A* weak segment tree *for the set S of segments consists of a rooted binary tree T, with ordered in its leaves the elementary intervals defined by S. Furthermore, each node represents a subset of S, such that for any segment $s \in S$ the following two conditions hold:*
1. s is represented exactly once on every path from the root to a leaf, of which the elementary interval is contained in s;
2. s is not represented on any other path from the root to a leaf.

As the name implies, a traditional segment tree is a weak segment tree, but not necessarily vice versa. A weak segment tree gives more flexibility in the nodes at which the segments should be stored. This flexibility is needed to perform the update operations INSERT, DELETE, SPLIT and CONCATENATE efficiently. Figure 4 shows a traditional segment tree and one possibility for a weak segment tree on the same set of segments.

Let T be a weak segment tree that stores a set S of segments. Let δ be a node in T. Denote by S_δ the subset of S stored at δ. We will represent all sets S_δ together as a union-copy structure. All segments form the elements, and the sets S_δ form the sets. This allows us to efficiently e.g. copy the associated set of a node. The flexibility of the weak segment tree as to where segments are stored, allows us to empty a set S_δ for a particular node δ by shifting its elements to both its children. This will be an important operation for our purposes. The procedure DOWN(δ) describes this process in detail, by making calls to the operations of a union-copy structure. For an internal node δ, its two children are denoted by lchild(δ) and rchild(δ).

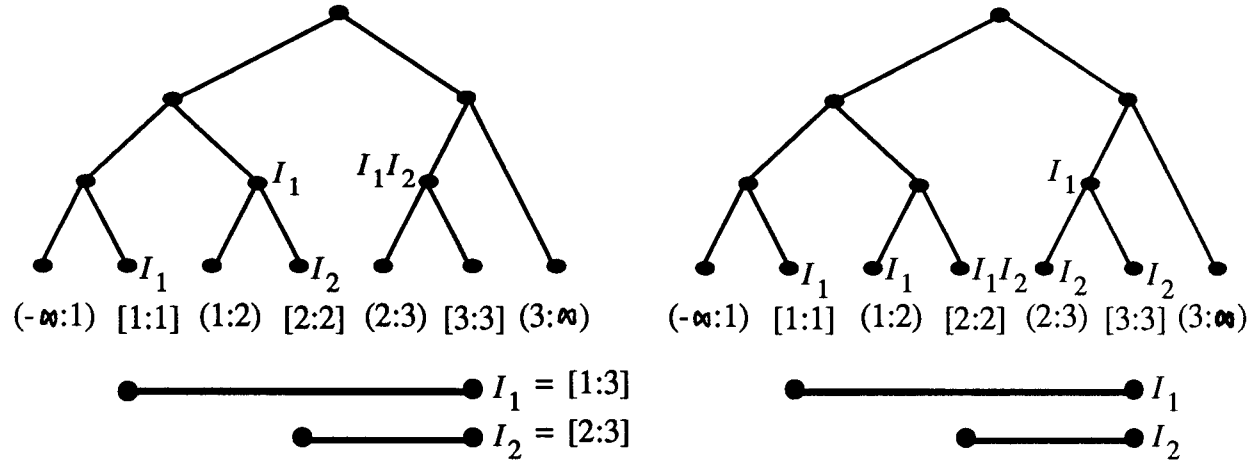

Figure 4: A traditional segment tree (left) and a weak segment tree (both)

DOWN(δ)
  SET-CREATE(S')
  SET-COPY(S_δ, S')
  SET-UNION($S_{\text{lchild}(\delta)}, S'$)
  SET-UNION($S_{\text{rchild}(\delta)}, S_\delta$)
  SET-DESTROY(S') { Remove the empty set S' }

With the appropriate choices of the underlying union-find structures, the operation DOWN(δ) can be performed in only $O(1)$ time!

3.1 The union-copy structure for dynamic segment trees

In this section we describe the union-copy structure that is needed for the dynamic segment tree in detail. Let S be a set of n segments, and let T be a weak segment tree which stores the segments of S. Every node in T corresponds to a set in the union-copy structure, and every segment corresponds to an element. Consequently, the union-copy structure has $O(n)$ sets and $O(n)$ elements. To perform the operations on dynamic segment trees, the following operations of the union-copy structure must be supported efficiently: SET-UNION, SET-COPY, SET-ENUMERATE and ELEMENT-DESTROY. Furthermore, the trivial operations SET-CREATE, ELEMENT-CREATE and SET-DESTROY (for an empty set only) are needed.

To perform these operations efficiently we can take for the N-UF-structure any union-find structure that allows for constant time N-UNION (because N-FIND is not performed by the operations above). For example, the tree structure suffices (see any good textbook on data structures, e.g. [1, 10]). For the R-UF-structure we take a structure in which R-FIND takes constant time. We use the UF(i)-structure of La Poutré [8], which allows for

R-FIND in $O(i)$ time and R-UNION in $O(a(i,n))$ time amortized (where $a(i,n)$ is a very slowly growing function related to the inverse Ackermann function, see the appendix for its definition). We choose i a constant. The analysis of the union-copy structure for this choice of N-UF and R-UF structures is postponed to the analysis of the dynamic segment trees.

3.2 The structure

The dynamic segment tree consists of the following three parts. Firstly, a weak segment tree, which is implemented using a balanced red-black tree (see [6, 10]). Every node is augmented with an extra pointer to a set node of the union-copy structure. Secondly, the union-copy structure as described above. Thirdly, a dictionary tree, which is a balanced red-black tree, storing the set S of segments in its leaves, ordered on increasing left endpoint, and with equal left endpoints on increasing right endpoint. Every leaf in the dictionary tree stores a segment, which is an element node of the union-copy structure. The dictionary tree is needed for deletions, because the element node in the union-copy structure must be located before it can be deleted. Futhermore, for every leaf of the weak segment tree that stores an endpoint $[x_i : x_i]$, an integer m is added, which represents the number of segments in the tree that have x_i as their endpoint. Let T be a dynamic segment tree. Then we denote by ST_T the weak segment tree of it, and by D_T the dictionary tree.

3.3 The operations

To update the red-black trees ST_T and D_T, we have the basic operations insert, delete, concatenate and split for red-black trees, which we consider to be known. Their description can be found in [6, 10]. If these operations are performed on ST_T, they are augmented in the following way. Just before a left-rotation takes place at a node δ, perform DOWN(δ); DOWN(rchild(δ)). Analogously, just before a right-rotation takes place at a node δ, perform DOWN(δ); DOWN(lchild(δ)) (see Figure 5). Just before a double rotation takes place, DOWN procedure is also performed at the proper nodes and in the proper order. As a result, the nodes involved in the rotation will have empty associated structures. This guarantees that the rotation maintains the weak segment tree property (see Definition 1).

In the description of the algorthms below, the following notational conventions are used. T, T_1 and T_2 are dynamic segment trees. δ denotes a node in ST_T or D_T, and γ denotes a leaf. If γ is a leaf in ST_T, then int(γ) denotes the elementary interval stored at γ. If int(γ) = $[x_i : x_i]$, then m(γ) denotes the number of segments that have x_i as their endpoints. Because the operation CREATE is trivial, we omit its description.

The procedure INSERT first inserts the segment in the dictionary tree. Then it inserts the new elementary intervals (if needed), and selects the nodes in the segment tree to which the segment must be added. This is done in the usual way for segment trees. Only, the insertion of a segment to a node is performed by SET-INSERT on the union-copy structure.

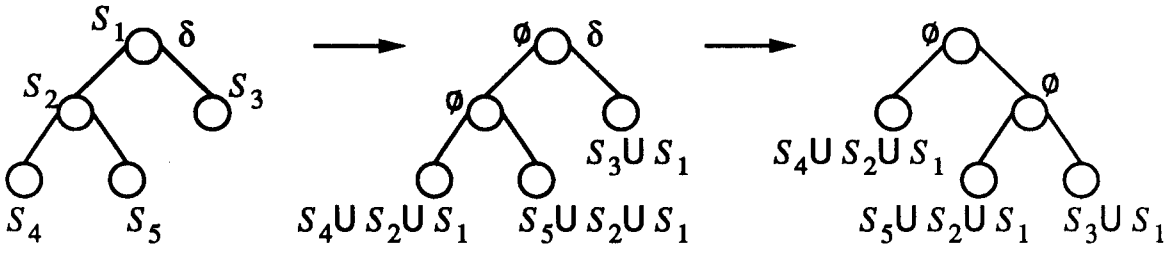

Figure 5: A right-rotation at a node δ

INSERT($[x_1 : x_2], T$)
insert $[x_1 : x_2]$ in the dictionary D_T
NEW-ENDPOINT(x_1); NEW-ENDPOINT(x_2)
$\delta \leftarrow$ the node in ST_T where the path to x_1 goes left and
the path to x_2 goes right
for δ_1 on path from lchild(δ) to the leaf that contains x_1 **do**
if x_1 is in left subtree of δ_1 **then** SET-INSERT($S_{\text{rchild}(\delta_1)}, [x_1 : x_2]$)
for δ_2 on path from rchild(δ) to the leaf that contains x_2 **do**
if x_2 is in right subtree of δ_2 **then** SET-INSERT($S_{\text{lchild}(\delta_2)}, [x_1 : x_2]$)

NEW-ENDPOINT(x)
$\gamma \leftarrow$ the leaf that contains x in ST_T
if int(γ) is an open interval $(a : b)$ **then**
int(γ) $\leftarrow [x : x]$
insert $(a : x)$ as a leaf γ' in ST_T
insert $(x : b)$ as a leaf γ'' in ST_T
SET-CREATE($S_{\gamma'}$); SET-CREATE($S_{\gamma''}$)
SET-COPY($S_\gamma, S_{\gamma'}$); SET-COPY($S_\gamma, S_{\gamma''}$)
m(γ) $\leftarrow$ 0
SET-INSERT($S_\gamma, [x_1 : x_2]$); m(γ) $\leftarrow$ m(γ)+1

The procedure DELETE removes the segment from the dictionary tree, and also from the union-copy structure by means of ELEMENT-DESTROY. Then it removes the leaves that correspond to unused endpoints.

DELETE($[x_1 : x_2], T$)
$\gamma \leftarrow$ the leaf in D_T that contains $[x_1 : x_2]$
ELEMENT-DESTROY(γ)
delete $[x_1 : x_2]$ from the dictionary D_T
REMOVE-ENDPOINT(x_1)
REMOVE-ENDPOINT(x_2)

REMOVE-ENDPOINT(x)
  $\gamma \leftarrow$ the leaf of ST_T that contains x
  $\mathrm{m}(\gamma) \leftarrow \mathrm{m}(\gamma) - 1$
  if $\mathrm{m}(\gamma) = 0$ **then**
    { let int(leaf left of γ) $= (a : x)$; let int(leaf right of γ) $= (x : b)$ }
    $\mathrm{int}(\gamma) \leftarrow (a : b)$
    delete from ST_T the leaf left of γ
    delete from ST_T the leaf right of γ

To concatenate two dynamic segment trees, all one has to do is concatenate the two ST_T trees and the two D_T trees with the standard red-black tree concatenate algorithm. The union-copy structure is concatenated implicitly; it follows from the concatenation of the two pairs of trees.

CONCATENATE(T_1, T_2, T)
  if T_1 is empty **then** $T \leftarrow T_2$; ready
  if T_2 is empty **then** $T \leftarrow T_1$; ready
  $\gamma_1 \leftarrow$ the rightmost leaf of ST_{T_1} { let $\mathrm{int}(\gamma_1) = (x : \infty)$ }
  $\gamma_2 \leftarrow$ the leftmost leaf of ST_{T_2} { let $\mathrm{int}(\gamma_2) = (-\infty : y)$ }
  $\mathrm{int}(\gamma_1) \leftarrow (x : y)$
  delete γ_2 from ST_{T_2}
  concatenate ST_{T_1} and ST_{T_2} to form ST_T
  concatenate D_{T_1} and D_{T_2} to form D_T

The SPLIT procedure basically splits the weak segment tree ST_T and dictionary tree D_T. Splitting a red-black tree goes in two phases. In the first, the tree is cut along the path to the splitting value, and in the second phase, the subtrees obtained from cutting are composed to the resulting trees. We refer to these phases as cutting and composing.

SPLIT(T, T_1, T_2, y)
  if T is empty **then** make empty trees T_1 and T_2; remove T; ready
  forall δ on the path in ST_T to y **do** DOWN(δ)
  { all nodes on the path to y have an empty set as associated structure }
  cut ST_T along the path to the leaf γ that contains y, giving
    a sequence S_l of subtrees left of the path and a sequence S_r of subtrees right of the path
  if $\mathrm{int}(\gamma) = [y : y]$ **then**
    add γ to S_l
    make a leaf γ' with $\mathrm{int}(\gamma') \leftarrow (y : \infty)$, and add it to S_l
    make a leaf γ'' with $\mathrm{int}(\gamma'') \leftarrow (-\infty : x')$, and add it to S_r
      { where x' is the smallest value of an endpoint in S_r }
  else { let $\mathrm{int}(\gamma) = (x : x')$ }
    $\mathrm{int}(\gamma) \leftarrow (x : \infty)$
    add γ to S_l
    make a leaf γ' with $\mathrm{int}(\gamma') \leftarrow (-\infty : x')$, and add it to S_r
  compose S_l to from a new weak segment tree ST_{T_1}
  compose S_r to from a new weak segment tree ST_{T_2}
  split D_T with $[y : y]$ into two new dictionary trees D_{T_1} and D_{T_2}

The procedure for STABBING-QUERY is quite simple. Its correctness follows directly from the definition of the weak segment tree. As for ordinary segment trees, the procedure follows the path to the query value, and reports all segments stored with nodes on the path by means of the union-copy procedure SET-FIND.

STABBING-QUERY(T, y)
forall nodes δ on the path in ST_T to y **do** SET-FIND(S_δ)

3.4 The analysis

The analysis of the dynamic segment tree, and the analysis of the union-copy structure used for it, go together. Performing updates causes changes in the structure, and, possibly, the dynamic segment tree requires additional space after any of the update operations. To bound the space requirements we take the following approach. It will be shown that a dynamic segment tree can be constructed in $O(n \log n)$ time (and space), and also that any operation adds at most $O(\log n)$ space to the structure. Thus after $n/2$ updates, the dynamic segment tree still requires $O(n \log n)$ space. Rebuilding the structure from time to time will keep the storage used within the $O(n \log n)$ bound. A second thing to analyze is the time bound for DELETE, for which it is necessary to know how much time SET-DESTROY in the union-copy structure may take. To get a bound on SET-DESTROY, we use a bound on the number of edges in the union-copy structure, which is again dependent on all operations on the weak segment tree. We begin the analysis with some simple lemmas.

Lemma 1 *The operations* INSERT, CONCATENATE *and* SPLIT *take* $O(\log n)$ *time on a dynamic segment tree that stores* n *segments, and these operations increase the space used by the structure with at most* $O(\log n)$.

Proof. This follows immediatly from the procedures that implement these operations, the balancedness of the weak segment tree and dictionary tree, and the choice of the union-copy structure. □

Lemma 2 *Suppose the union-copy structure (as in Subsection 3.1) has* $O(n \log n)$ *edges. Then* $O(n)$ ELEMENT-DESTROY *operations take* $O(n \log n \cdot a(i, n))$ *time (where* $a(i, n)$ *is the row inverse of Ackermann's function, see the appendix).*

Proof. From the code of the procedure SET-DESTROY one can see that if $\Omega(1 + k \cdot F_R(m))$ time is used, then $\Omega(k)$ edges have been deleted from the structure. Because there are at most $O(n \log n)$ edges to be deleted, it follows that $O(n)$ SET-DESTROY operations take $O(n \log n \cdot a(i, n))$ time. □

The following theorem is the main result of this section, and presents the overall performance of dynamic segment trees.

Theorem 2 *There exists a structure – the dynamic segment tree – for storing a set of n segments on the real line, such that* INSERT, CONCATENATE *and* SPLIT *operations each take $O(\log n)$ amortized time. Furthermore,* DELETE *operations take $O(\log n \cdot a(i,n))$ amortized time (where $a(i,n)$ is the row inverse of Ackermann's function, for a constant i), and stabbing queries take $O(k + \log n)$ time, where k is the number of segments reported. The structure requires $O(n \log n)$ space, and it can be constructed in $O(n \log n)$ time.*

Proof. The dynamic segment tree can be built in optimal $O(n \log n)$ time with any straightforward algorithm, for instance, by constructing the weak segment tree first, and then inserting the segments. Then we clearly have a dynamic segment tree that uses $O(n \log n)$ space. From Lemma 1 it follows that any of the operations INSERT, CONCATENATE and SPLIT add no more than $O(\log n)$ edges to the union-copy structure. The same holds for the operation DELETE. Consequently, after any sequence of $n/2$ update operations on the dynamic segment tree, it still uses $O(n \log n)$ space. Any INSERT, CONCATENATE and SPLIT operation in the sequence takes $O(\log n)$ time by Lemma 1, and any DELETE operation takes $O(\log n \cdot a(i,n))$ amortized time, by Lemma 2. After $n/2$ update operations, the dynamic segment tree is reconstructed completely. This takes $O(n \log n)$ time in total, thus an additional $O(\log n)$ amortized time per update operation.

At all time in the process, a STABBING-QUERY takes $O(\log n + k)$ time worst case, where k is the number of segments reported. This follows from the bound on SET-FIND, and the balancedness of the weak segment tree. □

4 Concluding remarks

In this paper we have presented a new data structure for representing sets and elements. It is a generalization of the union-find structure that also allows for copy operations (and some others) in an efficient way. The main result is stated in Theorem 1 in Section 2.

As an application, we have given a dynamic version of the segment tree, which allows for insertions, splits and concatenations in $O(\log n)$ time, deletions in $O(\log n \cdot a(i,n))$ time, and stabbing queries in $O(k + \log n)$ time, where k is the number of answers. The structure requires $O(n \log n)$ space, and can be built in $O(n \log n)$ time. These results improve upon the semi-dynamic segment trees, which only allow for insertions of segments, of which the endpoints are taken from a fixed universe. The main advantage of our structure is the fact that the time and space bounds are independent of the size of the universe.

Still, the dynamic segment tree doesn't have optimal deletion time, although for all practical situations it does. Furthermore, rebuilding has to be done after a linear number of operations. Hence, improvements might be possible.

Another open problem concerns the union-copy structure. It is not clear whether its performance can be improved in some way, or extended. For instance, it is unknown how to delete an element from a single set in the structure.

Appendix

Ackermann's function $A(i,n)$ is defined as follows:

$$\begin{aligned} A(0,n) &= 2n && \text{for } n \geq 0 \\ A(i,0) &= 1 && \text{for } i \geq 1 \\ A(i,n) &= A(i-1, A(i,n-1)) && \text{for } i \geq 1,\ n \geq 1. \end{aligned}$$

The functional inverse of Ackermann's function is defined for $m, n \geq 1$ by:

$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, 4\lceil m/n \rceil \geq n)\}.$$

For a fixed constant i, $a(i,n)$ is called the row inverse of Ackermann's function, and it is defined as follows:

$$a(i,n) = \min\{j \mid A(i,j) \geq n\}.$$

For $i = 1$, $a(i,n) = \Theta(\log n)$, and for $i = 2$, $a(i,n) = \Theta(\log^* n)$, the iterated logarithm. The higher the index i, the slower the function $a(i,n)$ grows in n. For $i \geq 2$, the function $a(i,n)$ is constant for all practical situations, although it goes to infinity as n does.

References

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] Bentley, J.L., *Solutions to Klee's Rectangle Problems*, unpublished notes, Dept. of Computer science, Carnegie-Mellon University, 1977.

[3] Blum, N., On the Single-Operation Worst-Case Time Complexity of the Disjoint Set Union Problem, *SIAM J. Comput.* **15** (1985), pp. 1021-1024.

[4] Gabow, H.N. and R.E. Tarjan, A Linear-Time Algorithm for a Special Case of Disjoint Set Union, *J. of Comp. and Syst. Sci.* **30** (1985), pp. 209-221.

[5] Golumbic, M.C., *Algorithmic Graph Theory and Perfect Graphs*, Ac. Press, San Diego, 1980.

[6] Guibas, L.J. and R. Sedgewick, A Dichromatic Framework for Balanced Trees, *Proc. of the 19th Ann. Symp. on FOCS* (1978), pp. 8-21.

[7] Halperin, D. and M. Overmars, Efficient Motion Planning for an L-shaped Object, *SIAM J. Comput.*, to appear.

[8] La Poutré, J.L., New Techniques for the Union-Find Problem, *Proc. of the 1st Ann. Symp. on Discr. Algorithms* (1990), pp. 54-63.

[9] La Poutré, J.L., Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines, *Proc. of the 22th Ann. STOC* (1990), pp. 34-44.

[10] Mehlhorn, K., *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[11] Mehlhorn, K., *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[12] Mehlhorn, K., S. Näher and H. Alt, A Lower Bound on the Complexity of the Union-Split-Find Problem, *SIAM J. Comput.* **17** (1988), pp. 1093-1102.

[13] Tarjan, R.E., Efficiency of a Good But Not Linear Set Union Algorithm, *J. of the ACM* **22** (1975), pp. 215-225.

[14] Tarjan, R.E., A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets, *J. of Comp. and Syst. Sci.* **18** (1979), pp. 110-127.

[15] Tarjan, R.E., *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, Penn., 1983.

[16] Tarjan, R.E. and J. van Leeuwen, Worst-case Analysis of Set Union Algorithms, *J. of the ACM* **31** (1984), pp. 245-281.