Concatenable Segment Trees

Marc J. van Kreveld and Mark H. Overmars

RUU-CS-88-36
November 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Concatenable Segment Trees

Marc J. van Kreveld and Mark H. Overmars

Technical Report RUU-CS-88-36
November 1988

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

Concatenable Segment Trees

Marc J. van Kreveld and Mark H. Overmars

Abstract

In this paper a variant of a segment tree is devised on which, in addition to insertions, deletions and stabbing queries, the operations concatenate and split can be performed efficiently. Insertions, concatenations and splits take $O(\log n)$ time, deletions take $O(\log^2 n)$ time, stabbing queries take $O(k+\log n)$ time (where k is the number of answers to the query), and the structure uses $O(n \log n)$ space to store. If we are interested in stabbing counting queries instead of stabbing queries, all operations can be performed in $O(\log n)$ time and the amount of storage used by the structure is reduced to $O(n)$. The technique is based on a new a general data structure that stores sets of objects, the union-copy structure, on which the operations union (of two sets), copy (of a set), insert (of an object in one or more sets), delete (of an object from all sets in which it occurs) and enumerate (of a set) can be performed efficiently.

1 Introduction

In computer science, data structures are used to store a collection of objects and to answer certain questions, called queries, about this collection in an efficient way. Besides queries, many data structures also allow for the operations insert and delete. In some situations we need even more operations on data structures, in particular split and concatenate operations. We say that a collection C of objects is split in two collections A and B with splitting object x when C is partitioned in A and B, such that A contains all objects less than or equal to x, and B contains all objects greater than x. Concatenation is the inverse operation to split. Two collections A and B can only be concatenated if all objects in A are less than all objects in B. In that case, the result of the concatenation of A and B is a collection C containing all objects of A and B. Also, we can speak of splitting and concatenating data structures if the collections of objects they represent are split and concatenated.

For some types of trees storing simple objects, efficient algorithms exist for splitting and concatenating. A structure which allows for these operations (in addition

Dept. of Computer science, University of Utrecht, P.O.Box 80.089, 3508TB Utrecht, The Netherlands.

to insert, delete and member query) is called a concatenable queue, and usually a 2-3 tree is used to represent it.

For most other types of data structures, especially the multi-dimensional data structures, split and concatenation methods are not available. The problem is that in most such structures, with each node there is an associated structure that stores some set of objects. Splitting then normally means rebuilding such associated structures, which takes too much time. Hence, more sophisticated techniques are required. In this paper we devise such a method for segment trees (see [3, 4, 8, 9, 11, 14]). This variant of the structure we design works efficiently and might be useful in a number of geometric applications (it was recently used for solving some robotics problem [6]).

Segment trees are used to store intervals on a line. Such intervals might overlap. When splitting a segment tree we assume that no interval contains the splitting point (otherwise the split would not be well defined). When we concatenate two segment trees, we assume that there is no point that is contained in the interior of an interval in both of the two segment trees. Thus, the intervals of the two trees must be separated by a point.

The concatenable segment tree will have the 2-3 tree as underlying structure. Both the standard segment tree and the 2-3 tree are descibed briefly in section 2.

To maintain the associated structures at the internal nodes of the segment tree, a new general data structure for storing sets of objects is devised that supports union of two sets, copy of a set, insertion of an object in one or more sets, deletion of an object from all sets, and enumeration of a set in an efficient way. This structure is called the union-copy structure. The union-copy structure is interesting in its own right and might have other applications as well. It will be described in section 3.

In section 4, the concatenable segment tree itself is described, and methods for performing insertions, deletions, concatenations, splits and stabbing queries are given, using the main tree and the union-copy structure.

In a number of applications we are not interested in reporting all segments that contain x but we only want to know their number, a so-called stabbing counting query. In section 5 we will give an optimal structure for stabbing counting queries that does support splits and concatenates.

Conclusions and open problems are given in section 6.

2 Preliminaries

In this section we briefly describe the segment tree and the 2-3 tree with operations, which will be the underlying structure of our modified segment tree. Readers familiar with segment trees and 2-3 trees can skip this section.

The *segment tree* is a one-dimensional data structure used for solving many two-dimensional problems, such as finding rectangle intersections and finding the contour of the union of rectangles. It was introduced by Bentley in 1977 [3] (see

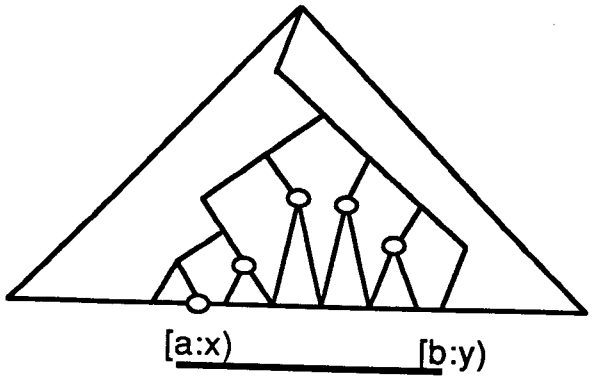

Figure 1: Insertion in a segment tree

also [4, 8, 9, 11, 14]). Suppose n segments (intervals) $[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]$ on the real line are given. Segments may intersect or overlap, and left and right endpoints of two segments may be equal. We sort all different endpoints, obtaining an ordered sequence $x_1, x_2, \ldots, x_m$, where $m \leq 2n$. These endpoints give rise to a number of consecutive intervals on the real line, called elementary intervals. These are $(-\infty : x_1), [x_1 : x_2), [x_2 : x_3), \ldots, [x_m : \infty)$, and they are stored in the leaves of a balanced binary tree. Each internal node is associated with an interval, being the union of the intervals of its sons. Now the segments themselves must be stored in the tree. A segment is stored with those nodes in the tree for which the associated interval is contained in the segment, but this does not hold for its father. All segments that are stored with a node δ are put in some structure (depending on what the segment tree is used for) that is stored with δ.

Often the segment tree is used for *stabbing queries*: given a point p on the real line, report all the segments that contain p. To perform a stabbing query, we follow the path from the root to the leaf storing the elementary interval in which p lies. For each node on the path, we report all segments that are stored with that node. Observe that, from the way segments are stored with nodes, it follows that all segments that contain p are reported exactly once, and no other segments are reported.

A segment can be inserted in the following way, assuming that the endpoints a and b already are endpoints of elementary intervals in the segment tree. We follow the path to a and b until it splits at some node. Here the path to a will turn left and the path to b will turn right. We continue the path to a, storing the segment at each node that is the root of the right subtree of a node where the path to a goes left. Also, we continue the path to b, storing the segment at each node that is the root of the left subtree of a node where the path to b goes right (see figure 1, the segment is stored with the encircled nodes). Since $O(1)$ time is spent at each level of the tree, an insertion can be performed in $O(\log n)$ time, assuming that the depth of the tree is $O(\log n)$ and an insertion in an associated structure can be done in constant time.

Theorem 1 *In a segment tree storing n segments, an insertion and deletion can be performed in $O(\log n)$ time, and a stabbing query takes $O(k + \log n)$ time, where k is*

the number of answers. It takes $O(n \log n)$ time to build the structure and $O(n \log n)$ space to store.

For a more detailed description of segment trees, see the references above.

The *2-3 tree*, introduced by Hopcroft in [1], is a one-dimensional tertiary tree satisfying the following three conditions: all objects are stored in the leaves; all leaves have the same depth; every internal node has two or three sons. It is clear that the depth of a 2-3 tree storing n objects is bounded by $O(\log n)$.

To insert an object, we search in the tree until the last internal node δ on the path to the object is found. If δ has two sons, the object can simply be added. If δ has three sons, then a new internal node δ' is made with two sons, one of δ and one with the new object. Thus δ and δ' now both have two sons. Then δ' is inserted as son of the father of δ in the same way the leaf with the object was inserted. This continues until we either find a node with two sons, or we arrive at the root node, in which case a new root node is made with two sons.

To delete an object, we search in the tree until the last internal node on the path is found, and we remove the son that contains the object. If δ has two sons now, we are ready. Else, we try to take a son from the brother of δ. If this brother has three sons, we can give one to δ and we are ready. Otherwise, the only son of δ is added to the brother (which has three sons now), and δ is removed from the tree in the same way the leaf was removed (ending when we find a node with three sons, or when we reach the root node, which is deleted in this case, and its only son becomes the root node).

Lemma 1 *In a 2-3 tree storing n objects, insertions, deletions and member queries can be performed in $O(\log n)$ time.*

Proof: We only follow one path in the tree and spend constant time at each level, and a 2-3 tree storing n objects has $O(\log n)$ levels. □

To concatenate two 2-3 trees, we insert the smaller of the two trees at a node of the other tree which has the same depth plus one (on the leftmost or rightmost path, depending on which contains the larger objects). Thus the same technique that has been used for the insertion of an object can be used here. If the two trees have same depth, we can simply make a new root with the two trees as subtrees.

To split a 2-3 tree with a given object b in two trees, one with the objects less than or equal to b and one with the objects greater than b, we divide the tree in subtrees as big as possible and of which we know that all objects belong to only one of the resulting trees. To this end, two lists of subtrees are made, one for each resulting tree. We follow the path from the root to b, and add every subtree of a node on the path which lies to the left of the subtree in which b lies, to one list, and every subtree that lies to the right to the other list. Then we construct one tree of each list, using the concatenate algorithm repeatedly and beginning with the smallest subtrees of a list (see figure 2).

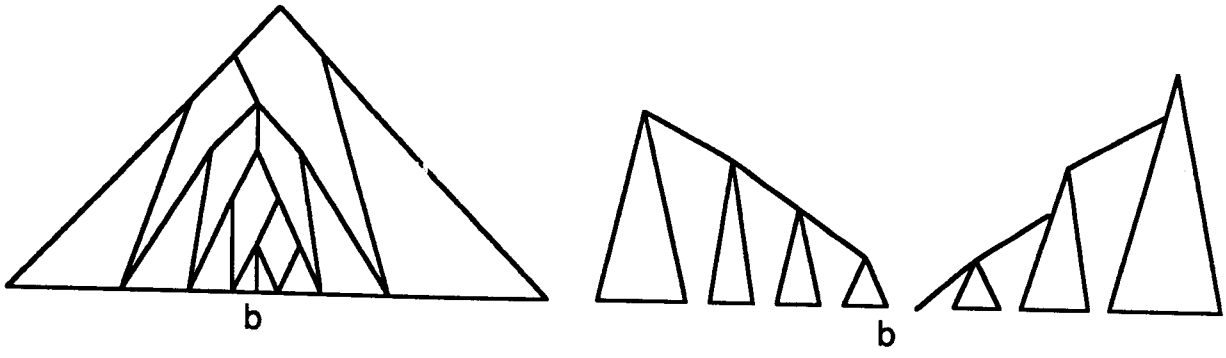

Figure 2: Split on a 2-3 tree

Theorem 2 *In a 2-3 tree storing n objects, insertions, deletions, concatenations, splits and member queries can be performed in $O(\log n)$ time each.*

Proof: It is clear that the concatenate algorithm takes $O(|\text{difference in depth of the two trees}|) = O(\log n)$ time if the deepest tree has $O(n)$ objects, since we only descend in the deeper tree until we reach a level with the same depth as the other tree, and $O(1)$ time is spent at each level we reach.

Dividing in subtrees for the splitting algorithm can be done in $O(\log n)$ time, since we walk down one path and spend $O(1)$ time at each node. As we have just seen concatenating takes time $O(\text{difference in depth}|)$. Let $d_1, d_2, \ldots, d_m$ be the depths of the subtrees in increasing order. Then the time taken to construct a tree of all these subtrees is $O(\sum_{i=1}^{m-1}((d_{i+1} - d_i) + 1)) = O(\log n)$, and the time bound for splitting follows. □

For a more extensive treatment of 2-3 trees and the split and concatenate algorithms, see [1].

3 Union-copy structures

The union-copy structure is related to the union-find structure (see [1, 7]), but differs from it in some important ways. Most important, the union-copy structure supports a copy operation and not a find operation, and the union-find structure supports a find operation and not a copy operation. Two sets in the union-copy structure are not necessarily disjoint, but if they are united, they must be disjoint. In the union-find structure, all sets are disjoint. The union-copy structure supports the following operations.

UNION(S_1, S_2): after this operation the set S_1 must contain all objects of S_1 and S_2. The sets S_1 and S_2 have to be disjoint. Set S_2 must be empty after the operation.

COPY(S_1, S_2): the set S_2 must contain the same objects as S_1, thus S_1 and S_2 are

equal after the operation and S_1 does not change.

INSERT(b, S): the object b must be added to the set S.

DELETE(b): the object b must be deleted from every set in which it occurs.

ENUMERATE(S): all objects in the set S must be reported.

For efficiency reasons, a new operation is defined to insert an object in possibly more than one set.

COMBINED INSERT($b, \{S_1, \ldots, S_m\}$): the object b must be added to all m sets S_j, where $1 \leq j \leq m$.

Notice that it is not possible to delete an object from fewer sets than it occurs in, except by first deleting it from all sets and then inserting it again in some of the sets.

3.1 The structure

The union-copy structure consists of a number of so-called set nodes, each representing a set, and a number of object nodes, each representing an object. Every set node is the root of a tree-like structure with the object nodes of the objects it contains in its leaves. Also, every object node is the root node of a tree-like structure with the set nodes of the sets the object is in, in its leaves. The two kinds of trees are intertwined, forming a graph with set nodes, internal nodes and object nodes. The meaning of an internal node is that all sets that can reach this internal node, contain all objects that can be reached from this internal node. There will only be one path from a set node to an object node of an object the set contains. The internal nodes are of one of two types, internal nodes that branch to the object nodes only (normal nodes), and internal nodes that branch to the set nodes only (reversed nodes). For an example of a union-copy structure, see figure 3, where set nodes are drawn as triangles, object nodes as squares and other nodes as circles.

Summarizing, there are four kinds of nodes.

- Set nodes: these are nodes that have no fathers and only one son (or zero when the set is empty). Via this son all the objects in the set can be reached. In this section we will sometimes use the name of the set when we refer to a set node.

- Object nodes: these are nodes that have no sons and only one father. They correspond to one object. For each object that occurs in some sets, there is exactly one object node. All sets containing this object will have a path from their set node to this object node.

- Normal nodes: these are intermediate nodes that branch from sets to objects. They have one father and arbitrary many — but at least two — sons.

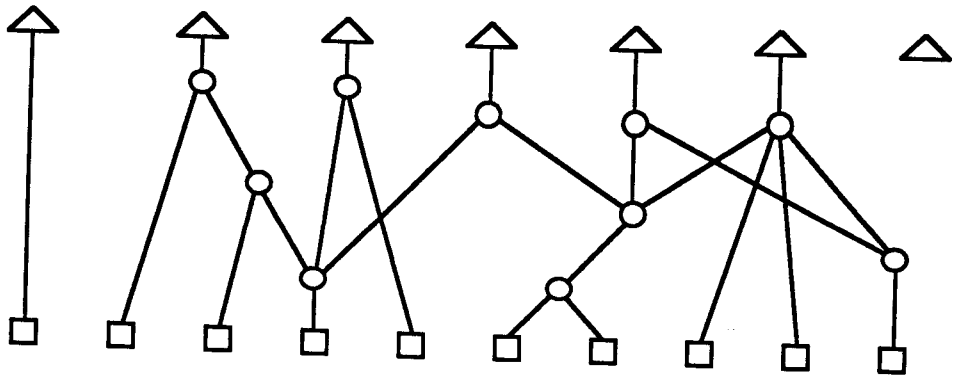

Figure 3: Example of a union-copy structure

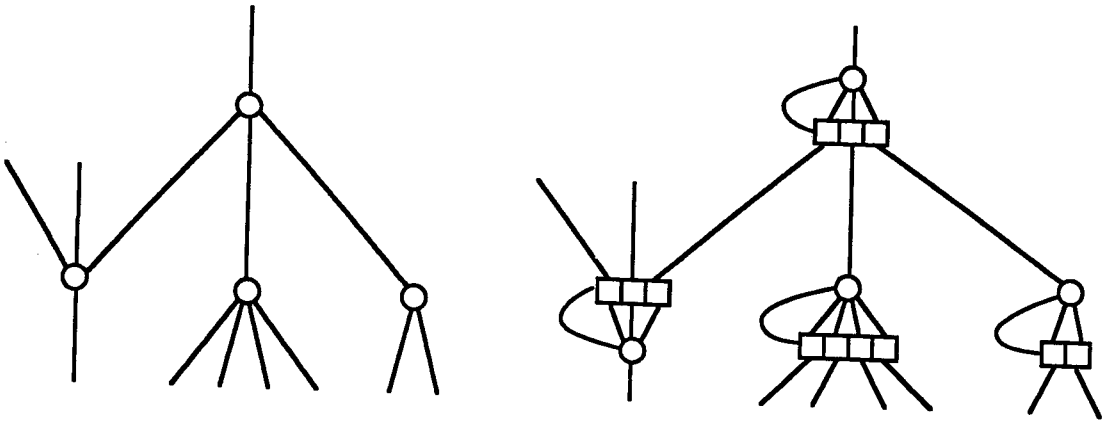

Figure 4: Implementation of nodes

- Reversed nodes: these are intermediate nodes that branch from objects to sets. They have arbitrary many — but at least two — fathers and only one son.

Every normal node has a doubly linked list with edges to all its sons, where an edge between two nodes consists of a pointer from the first node to the second and a pointer from the second node back to the first. Additionally, a normal node has an edge to its father. A reversed node has an edge to its son and a list with edges to its fathers. Each element in a list has a pointer to the node to which it belongs. (See figure 4, in which the leftmost node is a reversed node and the other nodes are normal nodes. We assume that the set nodes lie above the nodes drawn and the object nodes lie below the nodes drawn.) This organization allows for addition and removal of an edge in $O(1)$ time, and going from one node to the father(s) or son(s) in time O(number of fathers (sons)).

Reversed nodes appear to make copies possible, because an extra father can easily be added to reversed nodes, and normal nodes appear to make unions possible, because an extra son can easily be added to normal nodes.

To be able to locate object nodes when they must be deleted, all objects are also stored in the leaves of a balanced tree D (dictionary). Every object in a leaf gives

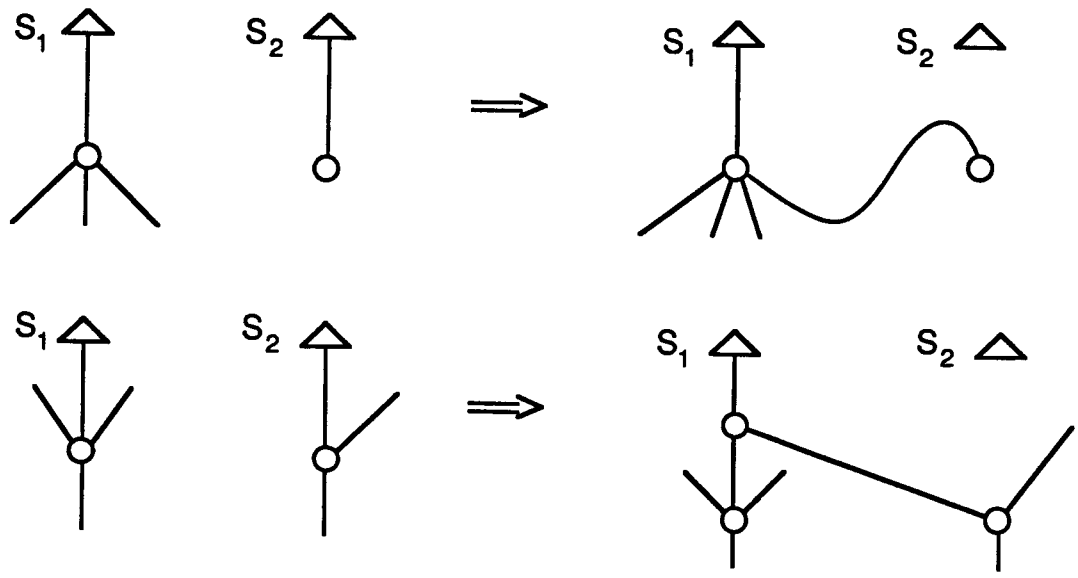

Figure 5: Union

access to the corresponding object node in the structure.

Sets are assumed to be directly accessible, for example with an index in an array. (When we use this structure for the concatenable segment tree, the algorithms for the operations will find the correct sets, and we will have direct access through an extra pointer stored at each node of the segment tree.)

3.2 Algorithms

The algorithms for the operations defined can be described now. We aim at constant time union and copy operations, enumeration in optimal time, thus linear in the number of objects in the set, and insert and delete as efficiently as possible. Insertions and deletions will take at least $O(\log n)$ time, but possibly more when we insert in or delete from more than one set.

UNION(S_1, S_2) (see figure 5)

- If S_2 is empty then do nothing.
- If S_1 is empty then turn the son of S_2 into a son of S_1 and make S_2 empty.
- If at least one of the set nodes has a normal son, then make the son of the other to be son of the normal son. Make the normal son son of S_1 and make S_2 empty.
- Else, add a normal node with the sons of S_1 and S_2 as its sons, and S_1 as its father. Make S_2 empty.

COPY(S_1, S_2) (see figure 6)

- If S_1 is empty, then make S_2 empty.

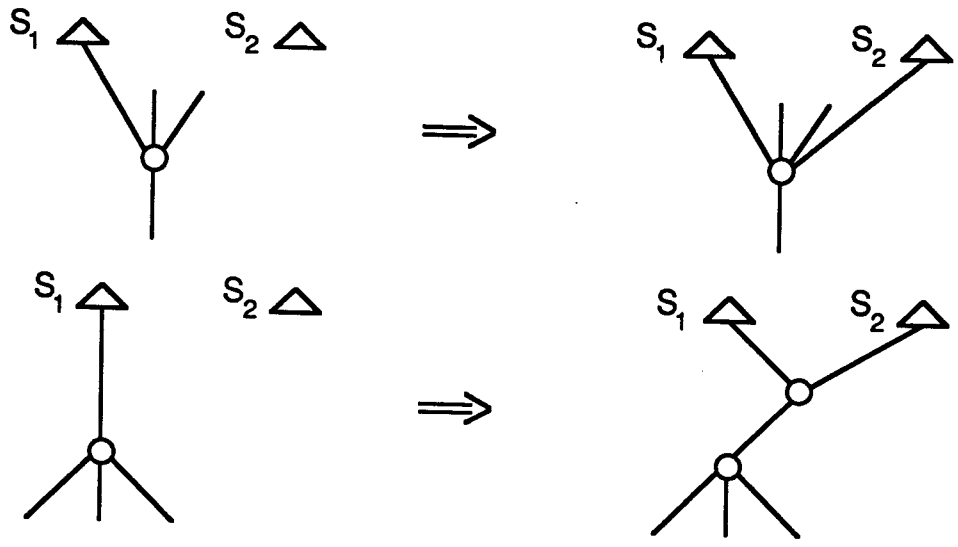

Figure 6: Copy

- If the son of S_1 is a reversed node, then make this node also the son of S_2 by adding S_2 as a father to the reversed node.
- Else, make a new reversed node with the son of S_1 as son and S_1 and S_2 as fathers.

Lemma 2 *The union and copy operations both take constant time on a union-copy structure.*

Proof: This follows easily because edges can be added and removed in constant time, and because sets are directly accessible. □

Since an insert is a special case of a combined insert, it will not be treated seperately. For simplicity, we assume that the objects to be inserted do not occur in the structure yet. During the algorithm we introduce two temporary sets S and T that must be removed afterwards.

COMBINED INSERT($b, \{S_1, \ldots, S_m\}$)

- Insert the object b in the dictionary tree D, and construct S as a set which only contains the object node corresponding to b.
  - If $m = 1$, then do union(S_1, S).
  - Else, for every j, $1 \leq j \leq m-1$, do copy(S, T); union(S_j, T). Then do union(S_m, S).

Lemma 3 *A combined insert in m sets in a union-copy structure with $O(n)$ objects takes $O(m + \log n)$ time.*

Proof: Insertion in D can be done in $O(\log n)$ time, and m unions and copies take $O(m)$ time by lemma 2. □

The union-copy structure described thusfar has a useful property, given in the following lemma.

Lemma 4 *Every reversed node is son of a set node or son of a normal node.*

Proof: It is obvious that a reversed node is never the son of an object node, because object nodes have no sons.

From the algorithms for union and copy it can easily be seen that whenever a new father-son edge is made, it is never between two reversed nodes. Consequently, this also holds for combined insert. □

The importance of this lemma is twofold. First, it makes the delete algorithm simpler. Second, and more important, it guarantees an optimal time bound for the enumeration of sets, as will be shown below.

In the delete algorithm, to be given next, we will make use of this lemma and restore the property whenever necessary.

DELETE(b) (see figure 7)

- Search for b in the tree D and remove b from D. Discern the following cases for the father δ of the object node containing b.
  - If δ is a set node, then make this set empty.
  - If δ is a normal node with at least 3 sons, then remove the one son that leads to b (see the upper part of figure 7).
  - If δ is a normal node with 2 sons, then let α be the father of δ and let β be the son of δ that doesn't lead to b. Remove δ and do the following.
    * If α and β both are reversed nodes, then if β has more fathers than α, let β have all the fathers of α and remove α. Else (if α has at least as many fathers as β), let α have all the fathers of β, make the son of β to be the son of α and remove β (see the middle part of figure 7).
    * Else, let β be the son of α (see the lower part of figure 7).
  - If δ is a reversed node, then discern the previous cases for all fathers of δ (which cannot be reversed nodes), and remove δ.

The situation where α and β both are reversed nodes is the situation where we have to do extra work to maintain the property of lemma 4.

Lemma 5 *In a union-copy structure with $O(n)$ objects, the deletion of an object which occurs in m sets can be performed in time $O(m + \log n)$.*

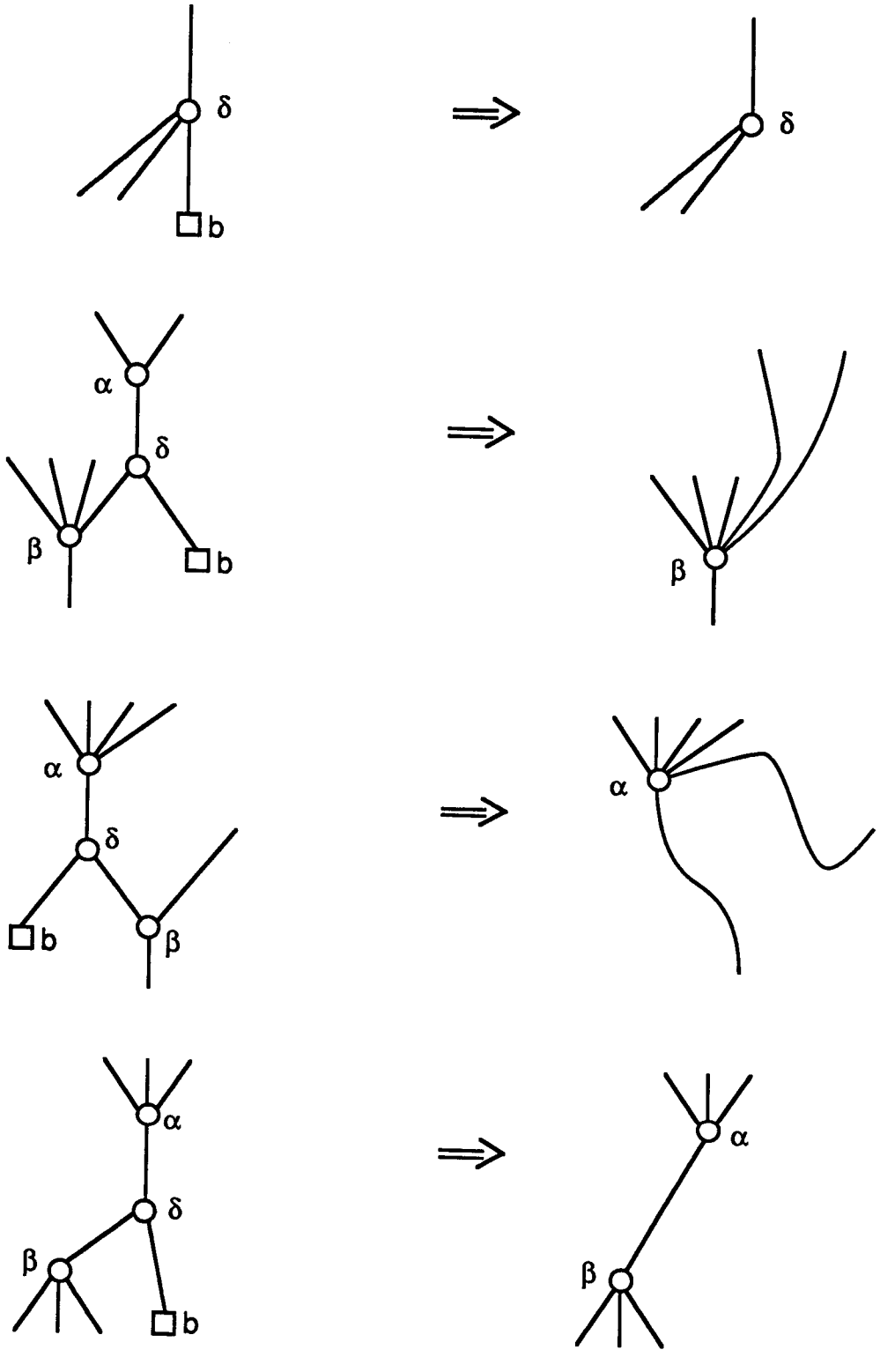

Figure 7: Delete

Proof: It takes $O(\log n)$ time to delete the object b from D.

Now consider the four possible cases. In the first two $O(1)$ time suffices. For the third case, assume α and β are both reversed nodes with x and y fathers, respectively. Then, obviously, b is in at least x sets. By the algorithm, we spend time bounded by $O(\min(x, y))$. (When not both α and β are reversed, the step takes $O(1)$ time.) In case 4, let δ have i fathers, each of which corresponds to at least one set containing b. For each such father one of the previous cases applies, thus for every father we can charge the time spent to delete a son to the number of occurences of b in sets. As every path from a set node to another node is unique, the total time spent to delete b from m sets is bounded by $O(m)$. □

In the delete algorithm we have taken some trouble to be more efficient than the above lemma claims. When using the union-copy structure for the concatenable segment trees, one can guarantee that the total number of edges in the union-copy structure is $O(n \log n)$. With this knowledge another bound on the time taken for deletions can be given.

Lemma 6 *In a union-copy structure with $O(n)$ sets and $O(n)$ objects, $O(n)$ deletions take time $O(n \log^2 n)$, if there have been $O(n \log n)$ edges in total in the structure.*

Proof: To prove the bound, we will charge time spent to the edges. There are two situations in which we charge costs to an edge during a deletion of an object. First, the deletion of an edge, which can happen only once to every edge, and takes constant time. Second, when the edges leading to two reversed nodes are taken together, to maintain the property that no reversed node has another reversed node as father. When an edge leads to such a reversed node, and it is in the smaller group of edges to a reversed node (i.e., the other reversed node has more edges leading to it), the edge is set to lead to the other reversed node, and we charge constant time to this edge. Afterwards it will be in a group of edges leading to a reversed node that is at least twice as large as its original group. Any reversed node can have at most $O(n)$ fathers, because any object can be in at most all $O(n)$ sets. Consequently, an edge can be set to lead to another node at most $O(\log n)$ time. Then it will be in a group with $O(n)$ edges, and it will never be in a smaller group. Thus we charge $O(\log n)$ time to every edge, and as there are $O(n \log n)$ edges, we charge at most $O(n \log^2 n)$ time. Furthermore, the time spent on the deletions from the dictionary tree is $O(n \log n)$. □

The last operation to be described is the enumeration of a set.

ENUMERATE(S)

- If the current node is an object node, then report the object.
- Else, enumerate all its sons.

Lemma 7 *An enumeration of a set S in a union-copy structure takes $O(1 + |S|)$ time, where S has $|S|$ objects.*

Proof: Observe that as every normal node has at least two sons, visiting it during an enumeration gives rise to an extra object in the set and, hence, the amount of time spent can be charged to this new answer. When visiting a reversed node, there does not appear a new answer but, due to lemma 4, its only son will be a normal node or an object node. Hence, the number of reversed nodes visited is not larger than the number of normal nodes visited plus the number of answers found. The bound follows. □

3.3 Time and space complexity

Let $f(n)$ be $\sum_{S \text{ set}} |S|$, the number of occurences of objects in sets for a given union-copy structure.

Lemma 8 *Given $O(n)$ objects, $O(n)$ sets, and with each object the sets in which it needs to be stored, then a union-copy structure can be built in time $O(n \log n + f(n))$, and it takes $O(f(n))$ space to store.*

Proof: The structure can easily be built by performing a combined insert operation for every object. The construction time follows immediatly.

To prove the storage bound, observe that the combined insert operation spends $O(1)$ time for each occurence of an object in a set. Additionally, some time is used for the dictionary tree, which uses $O(n)$ storage space. Thus for every occurence of an object in a set, $O(1)$ space is used. This proves the storage bound. □

Theorem 3 *There exists a structure that stores $O(n)$ sets, $O(n)$ objects and with $O(\sum_{S \text{ set}} |S|) = O(f(n))$ occurences of objects in sets such that*

- *a union takes $O(1)$ time;*
- *a copy takes $O(1)$ time;*
- *a combined insertion in m sets takes $O(m + \log n)$ time;*
- *a deletion from m sets takes $O(m + \log n)$ time;*
- *enumeration of a set S takes $O(1 + |S|)$ time.*

The structure can be built in $O(f(n) + n \log n)$ time and it takes $O(f(n))$ space to store.

Proof: From lemmas 2, 3, 5, 7 and 8. □

Note that the $O(\log n)$ in the bounds for insertions and deletions comes from the updates in the dictionary D. When we have direct access to the objects this tree can be avoided and, hence, both insertions and deletions take time $O(m)$.

When using the union-copy structure for designing a concatenable segment tree, $f(n)$ will be bounded by $O(n \log n)$. In that case we get the following results for the union-copy structure, using lemma 6 to estimate the deletion time.

Corollary 1 *In a union-copy structure with $O(n)$ sets, $O(n)$ objects, $f(n) = O(n \log n)$ occurences of objects in sets, and a combined insertion involves $O(\log n)$ sets, then*

- *a union takes $O(1)$ time;*
- *a copy takes $O(1)$ time;*
- *a combined insertion takes $O(\log n)$ time;*
- *a deletion takes $O(\log^2 n)$ time amortized, if at most $O(n \log n)$ unions and copies are performed, and at most $O(n)$ combined insertions;*
- *enumeration of a set S takes $O(1 + |S|)$ time.*

The structure can be built in $O(n \log n)$ time and it takes $O(n \log n)$ space to store.

4 Concatenable segment trees

In this section we will describe the main result of this paper, the concatenable segment tree and its operations. On concatenable segment trees, we want to perform the following operations efficiently:

INSERT$(T, [a : b])$: the segment $[a : b]$ must be inserted into the structure T.

DELETE$(T, [a : b])$: the segment $[a : b]$ must be deleted from the structure T.

CONCATENATE(T_1, T_2, T) : the structures T_1 and T_2 must be concatenated to T. For the segment $[a : b]$ of T_1 with b maximal and the segment $[c : d]$ of T_2 with c minimal the inequality $b \leq c$ must hold. Otherwise, concatenating is not possible.

SPLIT(T, T_1, T_2, a) : the structure T must be split into T_1 and T_2, where T_1 contains all segments $[x : y]$ with $y \leq a$ and T_2 contains all segments $[x : y]$ with $a \leq x$. T may not contain any segments $[x : y]$ with $x < a < y$. Otherwise, splitting is not possible.

STABBING QUERY(T, p) : all segments in T that contain p must be reported.

4.1 The structure

As has been mentioned in the introduction, we will use a 2-3 tree as underlying structure of the segment tree. Recall that for a standard segment tree the following rule holds: a segment is stored with each node δ whose corresponding interval is contained in the segment, but the interval corresponding to the father of δ is not contained in the segment. To be able to perform split and concatenate operations on the segment tree, we will weaken this rule. It is not difficult to see that stabbing queries will be performed equally well when a segment is not stored with a node like δ in the rule, but with all sons of δ. Or, exactly once with a node on every path decending from node δ.

Definition 1 *A weak segment tree is a segment tree where any segment* $[a : b]$

- *is stored exactly once on each path ending in an elementary interval that is contained in the segment* $[a : b]$*;*
- *is not stored on any path ending in an elementary interval that is not contained in the segment* $[a : b]$*.*

In a weak segment tree, stabbing queries can be performed in the standard way. The concatenable segment tree will consist of the following two parts:

1. A weak segment tree as main tree, with a 2-3 tree as underlying structure;
2. A union-copy structure to represent the associated structures of the nodes in the main tree. Every node in the main tree corresponds to one set in the union-copy structure (every node in the main tree has an extra pointer to a set node to give direct access to sets), and every segment corresponds to an object in the union-copy structure. The dictionary tree D of the union-copy structure will contain the segments in lexicographical order, first with respect to increasing left endpoint and secondly with respect to increasing right endpoint. The dictionary tree will also be a 2-3 tree.

4.2 Insertions and deletions

A segment must be inserted in two steps. First, the endpoints that are not already present in the main tree T must be inserted in T. This is actually the division of an elementary interval and thus of a leaf. Notice that for all nodes of which this leaf is a descendant, the corresponding interval doesn't change. Consequently, the sets of segments stored with these nodes do not need to change. Second, the segment itself must be inserted in the sets of the correct nodes of the main tree. First, an algorithm for the first step is presented, where T is the main tree and x is an endpoint of the segment to be inserted.

DIVIDE ELEMENTARY INTERVAL(T, x)

1. Go down the tree T to the leaf δ with the elementary interval $[x_i : x_{i+1})$ in which x lies. If $x = x_i$ then the algorithm is finished. Else, make two leaves δ_1 and δ_2 with elementary intervals $[x_i : x)$ and $[x : x_{i+1})$. Rename the set of segments stored with δ, S_δ, to be S_{δ_1} and copy$(S_{\delta_1}, S_{\delta_2})$. Replace δ by δ_1 and δ_2. Let γ be their father.

2. If γ has two or three sons, then the algorithm is finished. If γ has four sons, split γ into γ_1 and γ_2, each with two sons, rename the set S_γ to be S_{γ_1} and copy$(S_{\gamma_1}, S_{\gamma_2})$. Replace γ by γ_1 and γ_2 and move up the tree by repeating step 2 with their father, if it exists. If the father does not exist, then γ was the root. Make a new root with pointers to γ_1 and γ_2 and with a set node representing the empty set.

In this algorithm, we use the weakness of the segment tree, because in some cases, segments could be stored higher in the tree. Also notice the use of the copy operation to obtain two representations of a set of objects, where there only was one at first.

Now the insert algorithm itself is presented. It is basically equal to the insert algorithm for standard segment trees, except for the way the associated sets are updated. T is the main tree and $[a : b]$ is the segment to be inserted.

INSERT$(T, [a : b])$

1. Call divide elementary interval(T, a) and divide elementary interval(T, b).

2. Search with a and b in the main tree and select a collection K of nodes at which the segment must be stored in the following way. At a certain node δ the paths to a and b will split. From this node δ to a, select all the sons of nodes on the path that are more to the right than the son that leads to a, and add these sons to K. From δ to b, select all the sons of nodes on the path that are more to the left than the son that leads to b, and add these sons to K.

3. For all nodes γ in K, insert $[a : b]$ in the set S_γ with a combined insert $([a : b], \{S_\gamma \mid \gamma \text{ in } K\})$. (Here the extra pointers of the nodes in the main tree to set nodes in the union-copy structure are used.)

Lemma 9 *The insertion of a segment in a concatenable segment tree storing n segments takes $O(\log n)$ time.*

Proof: The divide elementary interval algorithm follows only one path in the main tree and at each level, it spends $O(1)$ time in the tree and at most $O(1)$ time to copy a set. Thus the time taken by step 1 of the insert algorithm is $O(\log n)$. In step 2, at most $O(\log n)$ nodes are selected for K, because at each level at most two nodes are selected on the path to a and at most two on the path to b. Thus step 2 takes $O(\log n)$ time. From corollary 1 it follows that step 3, the combined insert, takes $O(\log n)$ time, proving the time bound. □

The deletion of a segment is also done in two steps. First, the segment must be deleted from all sets in which it occurs. Second, if the endpoints of the removed segment aren't used as endpoints of other segments, then elementary intervals must be united.

The first step is performed on the union-copy structure, where the segment is deleted from dictionary tree, and from every set it occurs in. Notice that this removes the segment from the concatenable segment tree.

To perform the second step, we need to know how many times an endpoint is used. At each leaf, corresponding with the elementary interval $[x_i : x_{i+1})$, two integers are stored, one (nb) for the number of times x_i is used as left endpoint of a segment, and one (ne) for the number of times x_i is used as right endpoint of a segment. When both integers are zero, then this leaf is united with the leaf directly to the left. At first it seems that one integer storing the sum of the two would suffice, but this introduces difficulties during the split operation. (When performing a split with x_i, we would not know which part of the integer must be in what tree.)

When we unite two elementary intervals, and thus perform a deletion in a 2-3 tree, we sometimes pass one node to the brother of its father. But then the interval corresponding to this brother changes (increases) and hence, the segments stored there might no longer cover the whole interval. To avoid this we pass down the associated set to the sons of this brother in order to keep the tree correct. The following algorithm shows how this is done.

UNITE ELEMENTARY INTERVAL($T, [a : b]$)

1. Go down the tree T to a and subtract one from nb in the leaf. If both nb and ne are zero, then go to step 2, otherwise go to step 3.

2. Suppose the endpoint x_i of the elementary interval $[x_i : x_{i+1})$ has to be deleted, and the elementary interval to the left is $[x_{i-1} : x_i)$. Then change $[x_{i-1} : x_i)$ into $[x_{i-1} : x_{i+1})$ and remove the leaf with $[x_i : x_{i+1})$ from T. The tree T is restored to a 2-3 tree with the standard algorithm for deletions in 2-3 trees. See figure 8 for the operations that have to be performed when rebalancing the tree. The figure shows parts of the main tree, and $\Rightarrow$ denotes the transition that has to take place, $\cup$ denotes (disjoint) set union, and $\varnothing$ denotes the empty set.

3. Go down the tree T to b and subtract one from ne in the leaf. If both nb and ne are zero, then perform step 2, else the algorithm is finished.

The delete algorithm is very simple now.

DELETE($T, [a : b]$)

1. Delete the segment $[a : b]$ from the union-copy structure.

2. Call unite elementary interval($T, [a : b]$).

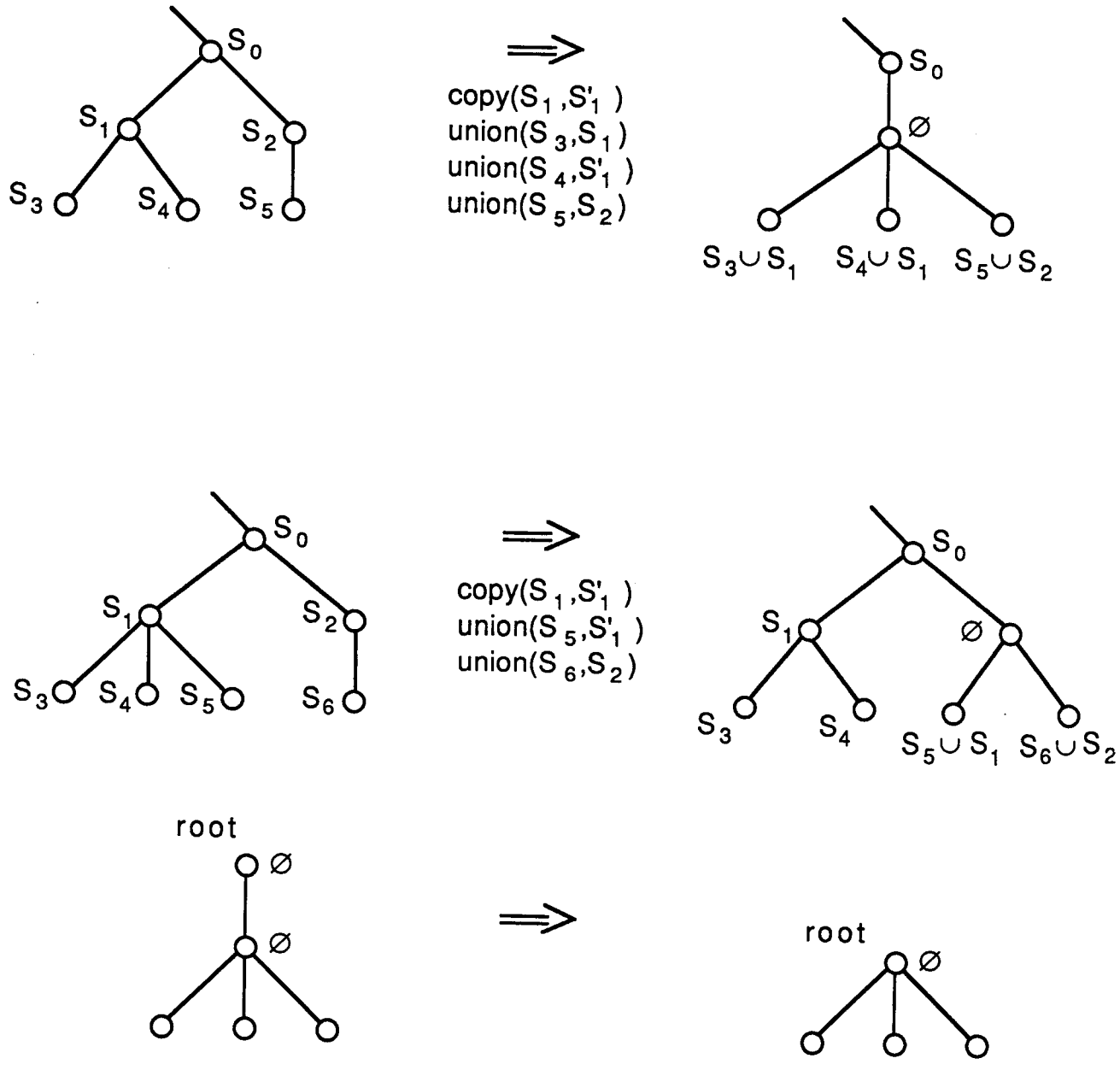

Figure 8: Deletion in the main tree

Lemma 10 *The deletion of a segment from a concatenable segment tree storing n segments takes $O(\log^2 n)$ time amortized.*

Proof: Deleting a segment from the union-copy structure takes $O(\log^2 n)$ time amortized, which has been proved in corollary 1, and unite elementary interval takes $O(\log n)$ time, because at each level in the tree a constant number of copies, unions and other work is done (and we have proved in lemma 2 that a copy and a union take $O(1)$ time). □

Because two integers have been added to the leaves of the main tree, some minor changes have to be made in the insert algorithm. These changes are trivial and are left to the reader.

4.3 Concatenations and splits

Concatenation of two concatenable segment trees would be simple if there were no associated structures. Then the elementary intervals could be adjusted and the standard concatenate algorithm for 2-3 trees could be used on the main trees and the dictionary trees.

Notice that in a segment tree, the nodes on the leftmost and rightmost paths do not contain any segments, because no segment contains minus infinity or infinity. Also notice that, since sets of segments are only passed down to sons (and never go up), the sets of nodes on the leftmost and rightmost paths will remain empty after the removal of the leftmost or rightmost leaf in a tree (except for the new leftmost or rightmost leaf). This implies that we do not need to give attention to the associated structures, and the concatenate algorithm becomes straightforward.

CONCATENATE(T_1, T_2, T)

1. Assume without loss of generality that T_1 contains the segments with the lower endpoints and T_2 the segments with the higher endpoints, and that T_1 has at least as many levels as T_2 (all other cases are similar). Let $[x : \infty)$ be the rightmost elementary interval in T_1, and $(-\infty : y)$ and $[y : z)$ the two leftmost elementary intervals in T_2. Remove the leaf with $(-\infty : y)$ from T_2, using the rebalancing of the unite elementary interval algorithm of the previous subsection.

2. If $x = y$, then remove the leaf with $[x : \infty)$ from T_1 (using the rebalancing of unite elementary interval) and set $ne_{[y:z)}$ to be $ne_{[x:\infty)}$. Else, change the elementary interval $[x : \infty)$ in $[x : y)$.

3. Concatenate T_1 and T_2, the two main trees, using the standard concatenate algorithm for 2-3 trees.

4. Concatenate D_1 and D_2, the two dictionary trees, using the standard concatenate algorithm for 2-3 trees.

5. Set T to be T_1.

Lemma 11 *Concatenation of two concatenable segment trees, both storing $O(n)$ segments, takes $O(\log n)$ time.*

Proof: Steps 1 and 2 of the algorithm take $O(\log n)$ time, because two paths are followed and one or two leaves deleted. Steps 3 and 4 also take $O(\log n)$ time by theorem 2. Step 5 finally takes $O(1)$ time. □

The split operation is more complicated, although it is basically equal to the standard split algorithm for 2-3 trees. Again we try to move the segments in sets to the sons whenever they are in the way. This can happen when we split the segment tree in subtrees, and when we concatenate the subtrees to form a complete tree. We also have to split the union-copy structure.

Lemma 12 *Suppose T is a concatenable segment tree in which no segment contains a given point p in its interior. Then the union-copy structure of T consists of two unconnected parts, one containing all segments to the left of p (with the sets containing them) and one containing all segments to the right of p (with the sets containing them).*

Proof: Suppose the two parts are connected in some way. Then there must be a normal node leading to one segment of either part. Also there must be a set S leading to this normal node, and there is a node δ in the main tree of which S is the associated structure. The interval corresponding to δ must be contained in both mentioned segments. But neither of them contain p in their interior, and they lie on opposite sides of p. The only possibility is that δ corresponds to the interval $[p:p]$. A contradiction, because no node in the main tree corresponds to an interval which only contains one point. □

Lemma 13 *Suppose D is the dictionary tree of a concatenable segment tree in which no segment contains a point p in its interior. Then D can be split with $[p:p]$, using the standard algorithm for splitting 2-3 trees, in two trees, one with the segments to the left of p and one with the segments to the right of p.*

Proof: Trivial due to the ordering chosen on D. □

The two lemmas guarantee that the union-copy structure with the dictionary tree can be split correctly, and in fact the splitting of the union-copy structure is done by splitting the segment tree and the dictionary tree. Next we present the complete split algorithm, where the structure T has to be split with point a into two structures T_1 and T_2.

SPLIT(T, T_1, T_2, p)

1. Follow the path to p in the main tree, making lists A and B of subtrees. At every node δ, do the following.

   - If δ is a leaf, let $[x_i : x_{i+1})$ be the elementary interval with δ. Add a new leaf with $[x_i : \infty)$, $nb_{[x_i:\infty)} = 0$, and $ne_{[x_i:\infty)} = ne_{[x_i:x_{i+1})}$ to A. If $x_i = p$ then set $ne_{[x_i:x_{i+1})}$ to be 0, add δ to B, and add a leaf with $(-\infty : x_i)$ to B, else add a leaf with $(-\infty : x_{i+1})$ to B. Go to step 2.
   - If δ is an internal node, move the set associated to δ down to all its sons (if the set isn't empty) and do the following.
     - If δ has two sons and the path to p goes left, then add the right subtree of δ to B. Continue with the left son of δ.
     - If δ has two sons and the path to p goes right, then add the left subtree of δ to A. Continue with the right son of δ.
     - If δ has three sons and the path to p goes left, then add a node to B with as sons the two rightmost subtrees under δ. Continue with the left son of δ.
     - If δ has three sons and the path to p takes the middle, then add the left subtree of δ to A, add the right subtree of δ to B, and continue with the middle son of δ.
     - If δ has three sons and the path to p goes right, then add a node to A with as sons the two leftmost subtrees under δ. Continue with the right son of δ.

2. Now A is a list with subtrees of ascending height and decreasing elementary intervals, and B is a list with subtrees of ascending height and increasing elementary intervals. To form a complete tree of the subtrees in, say A, take the first two subtrees and combine them. Add this new subtree to the front of A and repeat until A contains only one tree. Two subtrees t_1 and t_2 of A are combined with the following steps (suppose t_1 was the first in the list and t_2 the second):

   - If t_2 and t_1 are equally deep, then make a new node with t_2 as left subtree and t_1 as right subtree.
   - Otherwise, go down the rightmost path of t_2, copy the set of every node on the path and unite it with the sets of its sons, until a node is reached whose subtree is equally deep as t_1. This will make the sets of the upper nodes on the rightmost path in t_2 empty. Now concatenate t_2 and t_1 using the standard algorithm for concatenating 2-3 trees (see figure 9).

3. Split the dictionary tree D with $[p : p]$, using the standard splitting algorithm for 2-3 trees.

Lemma 14 *A concatenable segment tree storing n segments can be split in $O(\log n)$ time.*

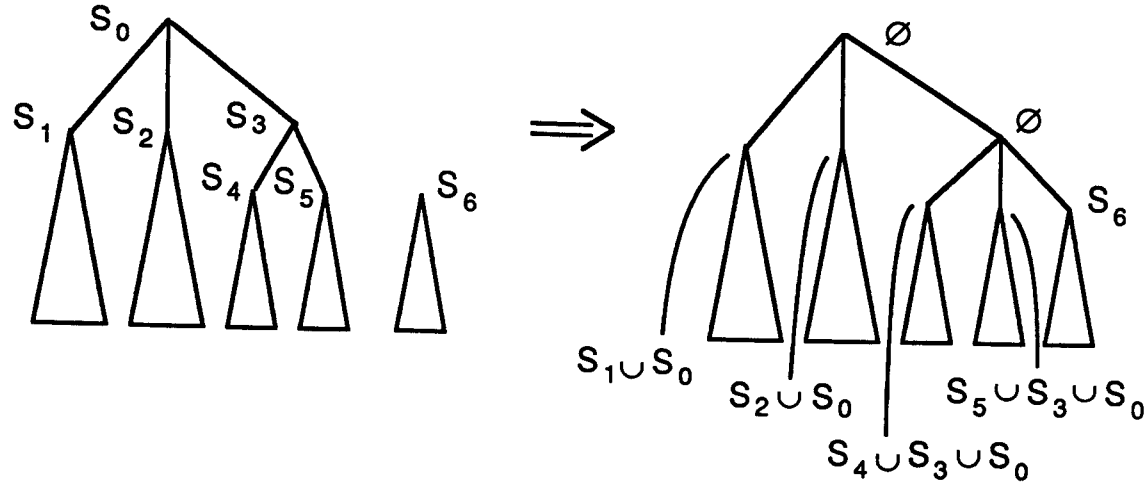

Figure 9: Concatenating subtrees and moving sets

Proof: The first step of the algorithm follows only one path and spends only constant time at each node for dividing the tree, copying and uniting. The second step takes $O(\log n)$ time by theorem 2 and lemma 2, and the third step $O(\log n)$ time by theorem 2. □

4.4 Stabbing queries

The last operation to be described is the stabbing query.

STABBING QUERY(T, a)

1. Follow the path in T to a until we end in a leaf. For every node δ on the path, report all segments in the corresponding set with enumerate(S_δ).

Lemma 15 *A stabbing query on a concatenable segment tree storing n segments takes $O(k + \log n)$ time, where k is the number of answers.*

Proof: We only follow one path and enumerate the associated set, which takes $O(\log n) + O(\sum_{\delta \text{ on path}}(1 + |S_\delta|))$ by lemma 7. Obviously $O(\sum_{\delta \text{ on path}} 1) = O(\log n)$ and $O(\sum_{\delta \text{ on path}} |S_\delta|) = O(k)$. The total stabbing query time is thus $O(k + \log n)$. □

4.5 Construction and storage considerations

Suppose n segments $[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]$ are given, and we have to construct a concatenable segment tree with these segments. The following straightforward

algorithm shows how this is done.

BUILD(T)

1. Sort all the endpoints $a_1, b_1, a_2, \ldots, a_n, b_n$, remove all duplicates, resulting in a list $x_1, x_2, \ldots, x_m$ with $m \leq 2n$. These endpoints give rise to $m+1$ elementary intervals $(-\infty : x_1), [x_1 : x_2), \ldots, [x_m : \infty)$ which partition the real line.
2. Build a 2-3 tree with these elementary intervals ordered in the leaves. We now have a skeleton of the structure, the main tree without associated structures.
3. For every segment $[a_i : b_i]$ with $1 \leq i \leq n$, insert it in the structure with the insert algorithm described before.

Lemma 16 *A concatenable segment tree storing $O(n)$ segments can be built in $O(n \log n)$ time and it takes $O(n \log n)$ space to store.*

Proof: Sorting the endpoints takes $O(n \log n)$ time, the construction of the 2-3 tree takes $O(n)$ time because the objects are given in sorted order, and the insertion of a segment takes $O(\log n)$ time by lemma 9. This proves the building time. The storage bound follows directly from the construction time, because in $O(n \log n)$ time no more than $O(n \log n)$ space can be used. □

When we perform operations on the concatenable segment tree, like insert, delete, concatenate and split, we often pass down the associated structure of a node to its sons. The associated structure itself is not copied, but an extra reference to it is made. This implies that a little extra storage is used each time an associated structure is passed down and thus copied. Ultimately we could get a situation where only leaf nodes have segments in their associated set, and if the segments are long enough, it could happen that $\Omega(n)$ segments have occurences in $\Omega(n)$ sets. In such a situation the structure would take $\Omega(n^2)$ space to store, and we obviously do not want this. To keep the total size of the structure $O(n \log n)$, we rebuild it completely after $n/2$ operations.

Lemma 17 *Suppose T is a concatenable segment tree storing n segments that has just been built (rebuilt). Suppose $n/2$ operations are performed on it. Then all segment trees (there may be more because of split operations) together take $O(n \log n)$ space to store (expressed in the actual number of segments).*

Proof: Any of the operations insert, concatenate and split take $O(\log n)$ time, thus they cannot use more than $O(\log n)$ extra storage. A deletion takes $O(\log^2 n)$ time amortized, but it can easily be seen that only $O(\log n)$ extra storage can be created. ($O(\log^2 n)$ time is used to lead existing edges to a new father, thus no extra space is used.) Consequently, $O(n)$ operations cannot use more than $O(n \log n)$ space. Furthermore, after $n/2$ operations the segment trees together contain at least $n/2$ segments, which proves that the storage bound is expressed in the actual number of segments. □

Theorem 4 *There is a structure, the concatenable segment tree, on which insertions, concatenations and splits can be performed in $O(\log n)$ time amortized, deletions can be done in $O(\log^2 n)$ time amortized, and stabbing queries take $O(k+\log n)$ time, where k is the number of answers. The structure takes $O(n \log n)$ space to store and can be built in $O(n \log n)$ time.*

Proof: We rebuild the structure after $n/2$ operations. This takes $O(n \log n)$ time, thus every operation takes an additional $O(\log n)$ time amortized. The theorem now follows from lemmas 9, 10, 11, 14, 15, 16, 17 and corollary 1. □

5 Stabbing counting queries

In this section we will concentrate on stabbing counting queries that only ask for the number of segments that contain a given query value. A counting segment tree is used for this purpose. Instead of storing a set of segments with each node, only an integer is stored that represents the number of segments stored with that node (i.e. the cardinality of the set). Thus in a counting segment tree, a segment gives rise to incrementing an integer by one at each node δ whose corresponding interval is contained in the segment, but the interval corresponding to the father of δ is not contained in the segment. On this tree, stabbing counting queries, insertions and deletions can be performed in $O(\log n)$ time, and the structure uses $O(n)$ space to store. We will see that we can obtain a concatenable counting segment tree with equal bounds, and concatenations and splits can be performed in $O(\log n)$ time.

5.1 The structure

We again use a 2-3 tree as main tree with in the leaves the elementary intervals. With each node in the tree we store an extra integer, which counts the number of segments that are stored with this node. The segments aren't stored anywhere in the tree. For a concatenable counting segment tree, the rule stating with which nodes what segments are stored is too strong to make operations possible. We define a weak counting segment tree in the same way as the weak segment tree of the previous section. Consequently, in a weak segment tree we can pass down the integers, representing the number of segments, to all sons of a node.

5.2 The operations

The operations insert, concatenate and split are very much like the operations on a concatenable segment tree. Only the operations on the union-copy structure are different (because it isn't used now), and these operations are replaced by the corresponding actions on the cardinality of the set, stored in the integer with each node. Union of two (disjoint) sets becomes addition of their cardinalities, copying a set becomes duplication of its cardinality, and combined insert in m sets becomes

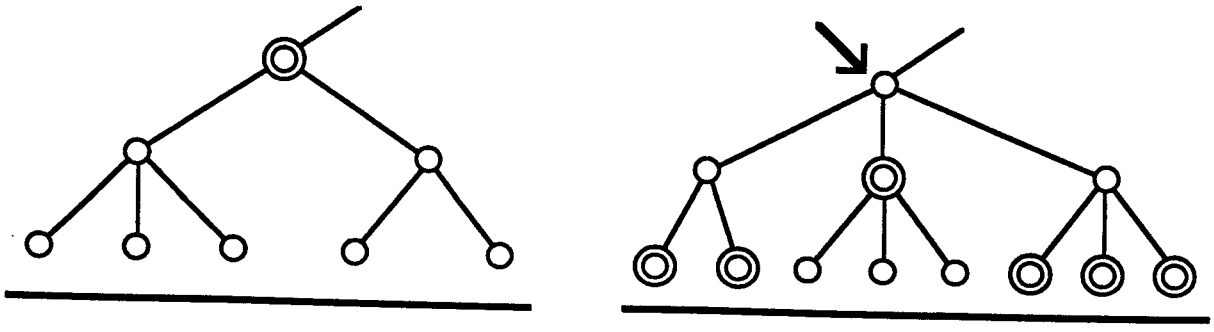

Figure 10: Deletion in a concatenable counting segment tree

incrementing m cardinalities by one. (For example, when we pass down a cardinality c from a node δ to its sons, we add the value of c to the cardinality of each son, and we set the integer stored with δ to be zero.)

The deletion is the tricky operation, although it is not difficult. After we have inserted a segment, we know at which nodes the integers have been incremented by one. But after other operations, these nodes may have passed down their integer, or duplicated it when a node is split. Thus we don't know where to find the nodes of which the integer must be decremented by one, and if we did know how to find them, there might be too many to decrement all these integers anyway.

The idea is that it does not really matter where we decrement the cardinalities as long as we do it on each path towards a leaf inside the segment exactly once. Hence, we can simply use the collection of nodes where the segment would be stored when inserting it. Thus a deletion finds the same collection and decrements the integers. So the segment is not really deleted, but it is compensated for in nodes at which the segment possibly isn't stored, but would be stored if it was a standard counting segment tree.

We will illustrate this with an example. Suppose some segment was stored at the encircled node of the left subtree of figure 10. This encircled node corresponds to the interval below the subtree. After some operations the subtree representing the same interval is shown in the right subtree, and the segment is stored at the encircled nodes. A deletion will decrement the integer stored with the node marked with an arrow. When a stabbing counting query reaches this tree part, values of the encircled nodes (which are one to high) will be compensated by the value of the node with the arrow (which is one to low). This will be the case independent of the path taken in this subtree. Notice that with this solution, the integers of nodes can become negative, and actually do not represent the cardinality of a set anymore.

Theorem 5 *There is a structure on which inserting, deleting, splitting, concatenating and stabbing counting queries can be performed in $O(\log n)$ time. The structure uses $O(n)$ space to store and it takes $O(n \log n)$ time to build.*

6 Concluding remarks and open problems

In this paper a variant of a segment tree was devised, on which we can perform insertions, concatenations and splits in optimal time $O(\log n)$, and deletions in time $O(\log^2 n)$. Stabbing queries take optimal $O(k + \log n)$ time, where k is the number of answers, and the structure uses $O(n \log n)$ space to store. If we are interested in stabbing counting queries instead of stabbing queries, the operations insert, delete, concatenate, split and stabbing counting query all take $O(\log n)$ time, which is optimal, and the structure uses $O(n)$ space, which is also optimal.

To obtain a concatenable segment tree, a new general data structure was devised, the union-copy structure, for operations on sets of objects. It allows for unions of sets, making copies, insertions, deletions and enumerate operations, all very efficiently. The structure is interesting in its own right and might be useful for other problems as well.

Some open problems do remain. First of all there is the question how deletions can be performed more efficiently, like in the standard segment tree. In our concatenable segment tree, this immediatly comes down to improving the deletion time in the union-copy structure. Another question we could ask about the union-copy structure is if there is a way we can delete from specific sets only, rather than first deleting it from all sets and then inserting it again in some sets. And a final question asks whether there is a way to perform the find operation efficiently on the structure, that is, can we report all sets a given object is in.

A more general question asks to perform concatenates and splits on other structures, for example interval trees (see [5]) and many two- and more-dimensional trees, as k-d trees (see [8] and [10]) and range trees (see [2, 11]). We recently obtained some results to split and concatenate k-d trees (see [12]) and a general method for decomposable searching problems (see [13]). But for individual problems better solutions might exist.

7 Acknowledgement

The authors would like to thank Mark de Berg for reading earlier drafts of this paper.

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., Decomposable searching problems, *Inform. Proc. Lett.* 8 (1979), pp. 244-251.

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

[4] Bentley, J.L., and D. Wood, An optimal algorithm for reporting intersections of rectangles, *IEEE Trans. on Computers* C-29 (1980), pp. 571-577.

[5] Edelsbrunner, H., *Dynamic data structures for orthogonal intersection queries*, Rep. F59, Tech. Univ. Graz, Institute für Informationsverarbeitung, 1980.

[6] Halperin, D., and M. Sharir, private communication.

[7] Mehlhorn, K., *Data structures and algorithms 1: sorting and searching*, Springer-Verlag, Berlin, 1984.

[8] Mehlhorn, K., *Data structures and algorithms 3: multi-dimensional searching and computational geometry*, Springer-Verlag, Berlin, 1984.

[9] Overmars, M.H., *The design of dynamic data structures*, Lect. Notes in Comp. Science Vol. 156, Springer-Verlag, 1983.

[10] Overmars, M.H., and J. van Leeuwen, Dynamic multi-dimensional data structures based on quad- and k-d trees, *Acta Inform.* 17 (1982), pp. 267-285.

[11] Preparata, F.P., and M.I. Shamos, *Computational geometry, an introduction*, Springer-Verlag, New York, 1985.

[12] van Kreveld, M.J., and M.H. Overmars, *Divided k-d trees*, Techn. Rep. RUU-CS-88-28, Dept. of Computer Science, University of Utrecht, 1988.

[13] van Kreveld, M.J., and M.H. Overmars, *Concatenable structures for decomposable searching problems*, Techn. Rep., Dept. of Computer Science, University of Utrecht, 1988, to appear.

[14] van Leeuwen, J., and D. Wood, The measure problem for rectangular ranges in d-space, *J. of Algorithms* 2 (1981), pp. 282-300.