Dynamic Stabbing Queries with Sub-logarithmic Local Updates for Overlapping Intervals

Elena Khramtcova[1(✉)] and Maarten Löffler[2]

[1] Computer Science Department, Université Libre de Bruxelles, Brussels, Belgium
elena.khramtsova@gmail.com

[2] Department of Information and Computing Sciences,
Utrecht University, Utrecht, The Netherlands
m.loffler@uu.nl

Abstract. We present a data structure to maintain a set of intervals on the real line subject to fast insertions and deletions of the intervals, stabbing queries, and *local updates*. Intuitively, a local update replaces an interval by another one of roughly the same size and location. We investigate whether local updates can be implemented faster than a deletion followed by an insertion.

We present the first results for this problem for sets of possibly overlapping intervals. If the maximum depth of the overlap (a.k.a. *ply*) is bounded by a constant, our data structure performs insertions, deletions and stabbing queries in time $O(\log n)$, and local updates in time $O(\log n/\log\log n)$, where n is the number of intervals. We also analyze the dependence on the ply when it is not constant. Our results are adaptive: the times depend on the current ply at the time of each operation.

1 Introduction

Preprocessing a set of objects for fast containment queries is a classic data structure problem. One of the most basic variants is maintaining a set S of one-dimensional intervals on the real line $\mathbb{R}^1$, subject to *stabbing queries*. Given a point $q \in \mathbb{R}^1$, the stabbing query for q aims to find all the intervals in S that contain q. Maintaining a set of intervals subject to stabbing queries is well understood both in the static [9] and in the dynamic setting [6,10]. In particular, it is well known how to maintain a set of n intervals subject to insertions and deletions in time $O(\log n)$, and stabbing queries in time $O(\log n + k)$, where k is the size of the output [10]. Data structures for stabbing queries remain an active research area and many variations of the problem have been studied: reporting the number of the stabbed intervals [1], finding the maximum priority stabbed interval [7,12], or considering different computational models and tradeoffs between memory requirements and query time [3,13].

E. Khramtcova was partially supported by F.R.S.-FNRS, and by the SNF grant P2TIP2-168563 under the Early PostDoc Mobility program.

M. Löffler was partially supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 639.021.123 and 614.001.504.

P. Weil (Ed.): CSR 2017, LNCS 10304, pp. 176–190, 2017.
DOI: 10.1007/978-3-319-58747-9_17

In certain applications, for example involving moving or uncertain data, a special kind of update is frequently performed, called *local update* by Nekrich [11]; see also Löffler *et al.* [8] and references therein. Intuitively, a local update replaces an interval by another interval *similar* to it: the new interval has roughly the same size and location as the old interval (we make this definition precise in the next section). The particular nature of the local update suggests that it should be possible to perform such updates strictly faster than in logarithmic time, as in that much time one could delete an interval and insert another interval that would not need to be similar to the deleted one. In this paper we show that this is indeed the case for stabbing data structures in $\mathbb{R}^1$.

Sub-logarithmic local updates for containment queries in a set of *disjoint* intervals have already been studied in Löffler *et al.* [8]. However, the condition that intervals are and remain pairwise disjoint at all times is unrealistic in many applications. The method in [8] is hard to generalize to overlapping intervals even if the depth of the overlap is constant (see also Fig. 4). Therefore designing a new data structure that would handle overlapping intervals is posed as an open problem in [8]. Here we address this problem.

In this paper, we present a data structure to store a set of possibly overlapping intervals, allowing fast insertions, deletions, and local updates. The performance of the data structure is measured in terms of the number of intervals in the set and the *ply* of the set: the maximum number of intervals containing any point in $\mathbb{R}^1$. If the ply is bounded by a constant, then the operations of insertion, deletion, and answering a stabbing query require $O(\log n)$ time each, and a local update requires $O(\log n / \log\log n)$ time.

We conclude this section with the directions for further research. First direction is extending our data structure to operate with two-dimensional objects, which are very important in many applications. Another open question concerns the fact that the performance of our data structure depends linearly on the current ply of the interval set, see Theorem 3. Thus one would desire to be able, if ply is too large, to quickly transit from our data structure to a more efficient one, e.g., the interval tree [4,6]. Clearly, such transition should be made in $o(n \log n)$ time, i.e., faster than building an interval tree from the scratch.

1.1 The Problem Statement and Our Result

Given a set S of intervals in $\mathbb{R}^1$, we aim to maintain S subject to fast *stabbing queries* and *local updates*, as well as fast insertions and deletions of the intervals. We assume that at all times the intervals in S are contained in a bounding box,[1] i.e., they are contained in a large interval $B \subset \mathbb{R}^1$.

To state the problem formally, we need some definitions.

Definition 1 (Stabbing query). *Given a query point q, return all the intervals in S that contain q, or report that there is no such interval.*

[1] If an update of S violates this bounding box condition, B can easily be enlarged. Thus our assumption does not restrict the setting, but rather simplifies the description.

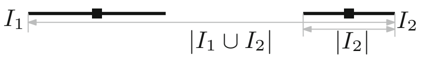

Fig. 1. 4-similar intervals I_1, I_2: the diameter $|I_2|$ is less than $|I_1|$ and $|I_1 \cup I_2| = 4|I_2|$.

For a closed, bounded, possibly disconnected region $R \subset \mathbb{R}^1$, the diameter of R is $|R| = \max_{p,q \in R} |q - p|$ (The diameter of an interval $I = [a, b]$ is $|I| = |b - a|$.)

Definition 2 (**[8]**). *For a pair I_1, I_2 of intervals in $\mathbb{R}^1$, and a real number $\rho > 0$, intervals I_1, I_2 are called* ρ-similar*, if $|I_1 \cup I_2| \le \rho \min\{|I_1|, |I_2|\}$. See Fig. 1.*

Definition 3 (Local update [8]). *Given an interval $I \in S$ and a pointer to an interval $I' \subset B$ that is ρ-similar to I for some constant ρ, replace I by I'.*

Definition 4 (Ply). *For a point $p \in \mathbb{R}^1$, the* ply *of S at p is the number of intervals in S that contain p. The* ply *of S is the maximum ply of S at any point in $\mathbb{R}^1$.*

Now we are ready to formulate our main problem.

Problem 1. Given a set S of n intervals in $\mathbb{R}^1$ that can possibly overlap, a bounding interval $B \in \mathbb{R}^1$ that contains each interval in S, and a real constant $\rho > 0$, preprocess S subject to fast stabbing queries, insertions and deletions of intervals, and local updates with parameter ρ.

We show how to solve Problem 1, such that the resulting data structure requires $O(n)$ space, and the following holds:

– If the ply of S is always at most some constant number, then stabbing queries, insertions and deletions of intervals require $O(\log n)$ time; local updates require $O(\log n / \log\log n)$ time.
– Otherwise, stabbing queries, insertions and deletions of intervals require $O(\log n + k \log n / \log\log n)$ time; local updates require $O(k \log n / \log\log n)$ time. Here k is the ply of S at the moment when the operation is performed.

In both cases, the time complexity bound for insertion of an interval is amortized; all other time bounds are worst-case.

We begin by reviewing an existing solution to Problem 1 for disjoint intervals [8], see Sect. 1.2. In Sect. 2 we give an alternative solution for disjoint intervals, that contains the ideas important for our data structure for overlapping intervals, which we present in Sect. 3. Section 4 discusses compression of the quadtree, an additional detail deferred to the end of the paper to ease the exposition.

1.2 A Data Structure for Disjoint Intervals (Ply = 1) [8]

Below we review the data structure for disjoint intervals by Löffler *et al.* [8]. It consists of two trees: one for performing updates, and one for performing queries.

The first tree, further referred to as the *quadtree*, is a one-dimensional compressed balanced quadtree on the center points of the intervals in S. In particular, an interval I is stored in the largest quadtree cell C such that C contains the center point of I and does not contain the center point of any other interval in S. Such cell C is a quadtree leaf. Further, since the intervals in S are disjoint, $|I| \leq 4|C|$. The quadtree is additionally augmented with level links, i.e., each quadtree cell has a pointer to its adjacent cells of the same size (if they exist). The quadtree is compressed, i.e., it contains a-compressed cells for some large constant a. An a-compressed cell C has only one child C', the size of C' is at most $|C|/a$, and $C \backslash C'$ contains no central points of intervals in S. Each non-compressed cell has zero or two children (in the former case it is a leaf).

The second tree, referred to as the *query tree*, is a balanced binary tree over the subdivision of $\mathbb{R}^1$ induced by the leaves of the quadtree. The leaves of the query tree store pointers to the corresponding leaves of the quadtree.

Stabbing Queries. Given a query point $q \in \mathbb{R}^1$, we must return the interval in S that contains q (if it exists), see Definition 1. To do this, we use the following:

Property 1 ([8]). For a quadtree leaf C, any interval that intersects C is either stored in C, or it is stored in the closest to C non-empty quadtree leaf[2] either to the left, or to the right of C.

The stabbing query is performed by checking whether q is contained in one of the intervals stored in the three candidate cells from Property 1. The leaf C is found in $O(\log n)$ time by a binary search for q in the query tree. To find the other two leaves in $O(1)$ time, we store with each leaf the pointers to the two closest non-empty leaves from both sides. Checking if a given interval contains q takes $O(1)$ time. Thus the stabbing query requires $O(\log n)$ time in total.

Local Updates. For an interval $I \in S$, we need to replace I with a new interval I' that is $O(1)$-similar to I, see Definitions 2 and 3. To do this, we follow pointers in the quadtree to find the cell that must store I'. Since I and I' are $O(1)$-similar, such cell is at most a constant number of cells away from the one that stores I, and thus can be determined in $O(1)$ time. We remove I from the old cell and insert I' into the new cell, performing the necessary compression, decompression, and balancing in the quadtree. This requires $O(1)$ worst-case time. The corresponding deletion and insertion in the query tree is done in $O(1)$ time, since the pointers are given and no search is needed. After each insertion or deletion the balance in the search tree is restored in worst-case $O(1)$ time [5]. Thus, a local update operation can be completed in $O(1)$ worst-case total time.

[2] Such leaves are not necessarily adjacent to C, as the adjacent ones might be empty.

Classic Updates. Intervals can be inserted or deleted from the data structure in $O(\log n)$ time: First, the insertion/deletion in the query tree is performed. This provides a pointer to the place in the quadtree where the insertion/deletion should be done.

We conclude this overview with stating the result.

Theorem 1 ([8]). *A set of n non-overlapping intervals can be stored in a data structure of size $O(n)$, subject to stabbing queries, insertion and deletion of intervals in $O(\log n)$ worst-case time, and local updates in $O(1)$ worst-case time.*

2 An Alternative Data Structure for Disjoint Intervals

Below we describe another solution to Problem 1 for disjoint intervals. This solution is less efficient than the one summarized in Sect. 1.2: it ignores Property 1, and it rather can be seen as a version of the data structure for 2-dimensional disjoint fat regions [8]. The main goal of this section is to simplify the latter data structure as much as possible, still making sure it contains the ideas, useful for our solution for overlapping intervals.

The data structure, similarly to the one of Sect. 1.2, contains the *quadtree* and the *query tree* built on its leaves, but they are now defined differently. Moreover, we use an additional type of structure: a *marked-ancestor tree* built on top of the quadtree cells.

The Quadtree. We maintain a compressed (but not balanced) quadtree that stores the intervals in S. Notice that sometimes we need to create the cells that would automatically exist, should the quadtree be balanced or non-compressed. We defer the discussion on handling this to Sect. 4, and until then we assume that the quadtree does not contain compressed nodes. The quadtree cells store intervals according to the following:

Condition 1. *An interval $I \in S$ is stored in a cell C if and only if C is the largest cell that contains I's center point and is entirely covered by I. See Fig. 2.*

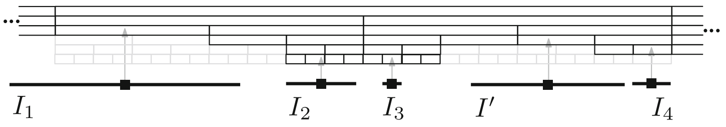

Fig. 2. A quadtree storing a set of disjoint intervals according to Condition 1

The above condition implies that if an interval I is stored in a quadtree cell C, then $|C| \le |I| < 4|C|$. In Fig. 2, the diameter of interval I_3 equals the size of its cell, and the diameter of interval I_2 is almost four times the size of its cell.

For the purpose which will be evident soon, we add more cells to the quadtree: For each interval I stored in a cell C according to Condition 1, we make sure that all (at most four) the cells of size $|C|$ intersected by I exist in the quadtree.

Figure 2 shows the quadtree for a set of five intervals. Black lines indicate the existing quadtree cells, and the light-gray lines indicate their further subdivision.

Marked-Ancestor Trees. We maintain three marked-ancestor trees built on cells of the quadtree. We denote these trees *L-MAT*, *R-MAT*, and *C-MAT*, standing for the left, the right, and the center marked-ancestor tree.

The quadtree cells are marked according to the following criteria:

Condition 2. *A quadtree cell C is marked in one of the marked-ancestor trees, if there is an interval $I \in S$ such that I intersects C, and the cell that stores I has size $|C|$. Specifically:*

(i) If C contains I's right endpoint (and thus I's center is to the left of C), then C is marked in L-MAT.
(ii) If C contains I's left endpoint then C is marked in R-MAT.
(iii) If I covers C entirely, then C is marked in C-MAT.

We say that C is marked by I, or that I marks C. See Fig. 3.

Each interval I marks either three or four quadtree cells: the cell C that stores I, the cells of size $|C|$ that contain respectively the left and the right endpoint of C, and possibly C's neighbor of size $|C|$ entirely covered by I. Note that, by the way we have defined the quadtree in the beginning of this section, such quadtree cells are always present in it.

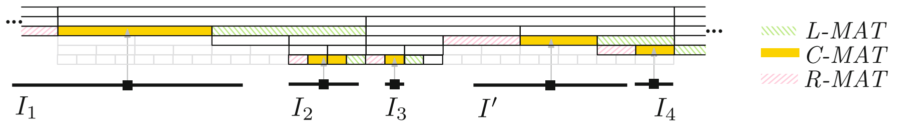

Fig. 3. Marking the cells of the quadtree from Fig. 2 in *L-MAT*, *C-MAT*, and *R-MAT*

Below Lemmas 1 and 2 are useful properties of the marked-ancestor trees, which are not hard to see. Lemma 3 provides an implementation of these structures.

Lemma 1. *Let C and C_a be two quadtree cells, such that C_a is an ancestor of C, and C_a is marked in C-MAT by some interval I_a. Then I_a entirely covers C.*

Lemma 2. *Let C and C_a be two quadtree cells that are both marked in L-MAT by an interval I and by an interval I_a, respectively. If C_a is an ancestor of C, then I_a lies to the left of I. A symmetric property holds for R-MAT.*

Lemma 3 **([2]).** *For any rooted tree T, a data structure over the nodes of T can be maintained, supporting insertions and deletions of leaves in $O(1)$ time, marking and unmarking nodes in $O(\log\log n)$ time, and $O(\log n/\log\log n)$-time lowest marked-ancestor queries. The time bound for insertions is amortized; other bounds are worst-case.*

If T is a path, then marking, unmarking, and answering a marked-successor (or a marked-predecessor) query require $O(\log\log n)$ time each.

The Query Tree. The query tree is a balanced binary tree whose leaves correspond to leaves of the quadtree, ordered as they appear on $\mathbb{R}^1$. Unlike Sect. 1.2, the leaves of the query tree do not have pointers to their non-empty neighbors.

Stabbing Queries. Given a query point q, we need to return an interval in S that contains q, if such interval exists. By locating q in the query tree, we find the quadtree leaf C that contains q. If C is marked in C-MAT by some interval I (C may or may not store I), then we report I. Otherwise, we find the lowest marked ancestor C_a of C in L-MAT, and check whether the interval that marks C_a contains q. We do the same in R-MAT.

Lemma 4. *The above procedure is correct and requires $O(\log n)$ time.*

Proof. Let C be the quadtree leaf that contains q. If C is marked by an interval I in C-MAT then by definition I covers C, and thus it contains q. Let I be an interval in S that intersects C, but does not cover it. Then I intersects one of the borders of C, say, the left border. Thus I marks an ancestor C_a of C in L-MAT. Moreover, C_a is the lowest marked ancestor of C in L-MAT: suppose that some $C'_a \neq C_a$ is the lowest marked ancestor of C; let I' be the interval that marks C'_a. Then both I and I' contain the left border of C'_a. We obtain a contradiction to the disjointness of the intervals in S. The argument for R-MAT is symmetric.

By Lemma 3, the lowest marked ancestor of C in L-MAT and the one in R-MAT can be determined in $O(\log n/\log\log n)$ time. Therefore, the total time required for the stabbing query is dominated by the time required for point location in the query tree, and thus it is $O(\log n)$. □

Local Updates. Given a pointer to an interval I stored in cell C, we need to replace it with a $O(1)$-similar interval I'. Let C' be the quadtree cell that needs to store I', see Condition 1. Assume that the pointer to C' is available.

Deleting I and inserting I' then reduces to repairing the data structures. Modifying the quadtree and the query tree is done as in Sect. 1.2. Deleting I requires unmarking the quadtree cells marked by I. The latter cells are easy to find in constant time as they are the cells of size $|C|$ that intersect I. If these unmarked cells are leaves, they may get deleted from the quadtree. Inserting I' requires the reverse manipulations: marking the cells of size $|C'|$ intersected by I, and adding them if they do not yet exist.

Lemma 5. *A local update requires $O(\log n/\log\log n)$ time.*

Proof. Intervals I and I' mark at most four quadtree cells each; such cells can be found in constant time. To insert and to delete a quadtree leaf is a constant-time operation. Thus the time required by the local update (after C' is available) is dominated by the time for a constant number of marking and unmarking operations, each of which by Lemma 3 requires $O(\log\log n)$ time. The overhead due to compression of the quadtree for finding C' is $O(\log n/\log\log n)$, see Lemma 8. The claim follows. □

Classic Updates. Insertion (resp., deletion) of an interval requires an insertion (resp., deletion) in the query tree, and an update to marked-ancestor structures. The former two operations require worst-case $O(\log n)$ time (including the overhead due to compression, see Lemma 8). The latter operation requires $O(1)$ time, amortized for insertions and worst-case for deletions. Therefore we say that the classic updates require $O(\log n)$ time per operation, amortized for insertion and worst-case for deletions.

3 A Data Structure for Overlapping Intervals

In this section we present our solution to Problem 1 for sets of intervals that may overlap. Section 3.1 considers the case when the ply of the interval set is always at most two. Generalization of the data structure to the case of higher ply is quite intuitive, and we sketch it briefly due to the space constraints.

3.1 Intervals with Ply ≤ 2

Now we are given a set S of intervals that may overlap, such that the ply of S is guaranteed to be at most two at any moment. We solve Problem 1 for such S.

We remark that partitioning S into a constant number of layers, such that at each layer the intervals are disjoint, does not seem to work.

Should we do this, we would need to restore the properties of the layers when after a local update two intervals of the same layer start overlapping. A natural way to handle this, i.e., to assign the interval that just has been updated the next possible layer, does not work: Fig. 4 shows three intervals and a sequence of local updates, where every local update causes a change of the layer for the updated interval. Thus we need a more involved data structure for solving our problem.

Our starting point is the data structure of Sect. 2. Note that a single *L-MAT* (and a single *R-MAT*) is not enough for our setting: It can happen that the quadtree leaf that contains a query point q has linearly many marked ancestors in *L-MAT* before the one marked by the interval that actually contains q. Figure 5 shows such situation with three marked ancestors; the construction can be continued to increase this sequence arbitrarily, using intervals and cells of smaller size. To overcome this issue, we use marked-ancestor trees of two levels. Below we discuss the modifications to the data structure of Sect. 2 in detail.

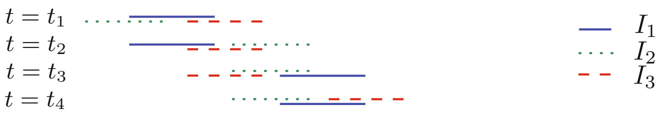

Fig. 4. Subdividing intervals in two layers of disjoint intervals is not efficient: A set of three intervals $\{I_1, I_2, I_3\}$, and a sequence of local updates that would cause linear number of layer changes. We illustrate here the first three updates in this sequence, i.e., the segment set in each of the first four time moments $t_1, \ldots, t_4$.

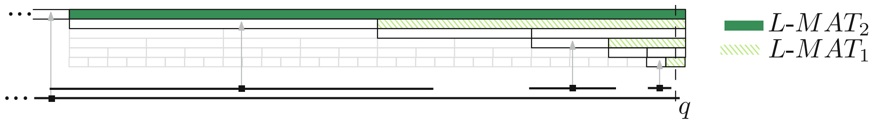

Fig. 5. A set of intervals with ply two, and a query point q, such that the cell of q has many marked ancestors in *L-MAT* (the shaded cells), but only the largest one is marked by the interval that contains q. This cell must be marked in $L\text{-}MAT_2$.

The Quadtree. The rule to store intervals in the quadtree cells is still provided by Condition 1. Since intervals are now allowed to overlap, they may be stored in intermediate cells of the quadtree. If an interval $I \in S$ does not overlap any other intervals, then I is stored in a quadtree leaf. The inverse is not true. Notice that since ply of S is at most two, one cell C stores at most two intervals. If C stores two intervals, then C is a leaf.

Marked-Ancestor Trees. We maintain two levels of the left and the right marked-ancestor trees, denoted by $L\text{-}MAT_i, R\text{-}MAT_i$, $i \in \{1, 2\}$. The center marked-ancestor tree ($C\text{-}MAT$) is unique, and is defined in the same way as previously.

Marking the quadtree cells in the left marked-ancestor trees is now done according to the following (marking in the right trees is symmetric):

Condition 3. *For a cell C, if there is an interval I such that Condition 2(i) holds for C and I, then C is marked in one of the left marked-ancestor trees. Specifically (See Figs. 5 and 6):*

- *C is marked in $L\text{-}MAT_2$ if there is a descendant C' of C and an interval I' such that Condition 2(i) holds for C' and I', and I entirely covers I'.*
- *C is marked in $L\text{-}MAT_1$, otherwise.*

Since *C-MAT* is defined in the same way as in Sect. 2, Lemma 1 still holds. The following lemma generalizes Lemma 2.

Lemma 6. *Let C and C_a be two quadtree cells that are both marked in $L\text{-}MAT_i$ for some level $i \in \{1, 2\}$ respectively by interval I and by interval I_a. If C_a is an ancestor of C, then the right endpoint of I_a lies to the left of the right endpoint of I. A symmetric property holds for $R\text{-}MAT_i$.*

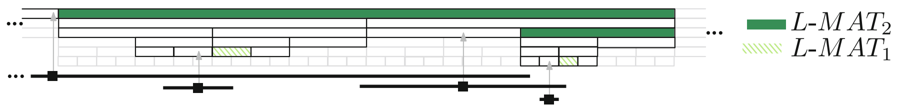

Fig. 6. A set of four intervals with ply 2; marking in $L\text{-}MAT_1$ and $L\text{-}MAT_2$ induced by them

Proof. Suppose for the sake of contradiction, that there are two quadtree cells C and C_a that are both marked in $L\text{-}MAT_1$ or both in $L\text{-}MAT_2$ by intervals I and I_a respectively, such that C_a is an ancestor of C, and the right endpoint of I_a is to the right of the right endpoint of I. Then I is covered entirely by I_a. It is not possible that both C and C_a are marked in $L\text{-}MAT_1$: by definition C_a must be marked in $L\text{-}MAT_2$, since I_a covers I. If both C and C_a were marked in $L\text{-}MAT_2$, then there would be an interval I' entirely covered by I. This would contradict the ply ≤ 2 condition for S. □

Marked-Descendant Trees. To update the marked-ancestor trees efficiently, we need a data structure built on the quadtree, that would quickly answer the following queries.

Definition 5 (Leftmost/rightmost marked-descendant query). *The* leftmost marked-descendant query *for an intermediate quadtree cell C is as follows. If the subtree of C contains cells marked in $L\text{-}MAT_1$, among these cells, return the one that comes first after C in the pre-order traversal of the quadtree.*[3] *Otherwise return* nil. *The* rightmost marked-descendant query *for C is: If the subtree of C contains cells marked in $R\text{-}MAT_1$, among these cells, return the one that precedes C in the post-order traversal of the quadtree.*[4] *Otherwise return* nil.

The following lemma justifies the use of marked-descendant queries to manipulate the marked-ancestor trees.

Lemma 7. *Suppose a quadtree cell C and an interval I satisfy Condition 2(i). Cell C is marked in $L\text{-}MAT_2$ if and only if (1) the leftmost marked descendant C_d of C exists, and (2) $I_d \subset I$, where I_d is the interval that marks C_d.*

Same holds for a cell marked in $R\text{-}MAT_2$ and its rightmost marked descendant.

Proof. If items (1) and (2) hold, then C is marked in $L\text{-}MAT_2$ by Condition 3.

Suppose C is marked in $L\text{-}MAT_2$. Then there is a descendant C' of C marked in $L\text{-}MAT_1$ by an interval I' such that $I' \subset I$. First note that C' is a marked descendant of C, thus the leftmost marked descendant C_d of C exists. Suppose $C_d \neq C'$. Since C_d appears before C' in the pre-order traversal of the quadtree,

[3] The pre-order traversal of a binary tree first visits the root, then it recursively visits the left subtree, and finally it recursively visits the right subtree.

[4] The post-order traversal of a binary tree first recursively visits the left subtree, then it recursively visits the right subtree, and finally it visits the root.

the left border of C_d either coincides with left border of C', or lies to the left of it. Thus I_d is completely to the left of I'. Since I_d is stored in a cell whose size is less than $|C|$, the left border of I_d cannot be to the left of the left border of I, and thus $I_d \subset I$. □

To efficiently answer the above queries, we maintain a pair of data structures, which we call the *left* and the *right marked-descendant tree*, respectively. The left marked-descendant tree is implemented by maintaining the path induced by the pre-order traversal of the quadtree, and the marked-successor data structure of Lemma 3 on top of this path for the cells marked in $L\text{-}MAT_1$. Maintaining the path can be done by augmenting each node with a pointer to its successor in the traversal. The leftmost marked-descendant query is then performed by querying the marked successor of C in the pre-order, and checking whether the returned cell is in the subtree of C. If no cell is returned, or if the returned cell is not a descendant of C, then we return *nil.* Otherwise we return that cell. The marked-successor queries require worst-case $O(\log\log n)$ time per query by Lemma 3.

The right marked-descendant tree is symmetric: we maintain the reversed post order, (i.e., determine the pointers for the post-order traversal and reverse all of them), and the marked-successor data structure on top of this path.

Stabbing Queries. Given a query point q, we need to report all intervals in S that contain q. To do that, we should first find the quadtree leaf C that contains q. If C or C's ancestor(s) are marked in $C\text{-}MAT$, we report the interval(s) that mark them, similar to Sect. 2. After that we query left and right marked-ancestor trees: We check the intervals that mark the lowest and the second lowest marked ancestor of C in $L\text{-}MAT_1$ (in case C is marked in $L\text{-}MAT_1$, we check its marking interval too). We report those of the above intervals that contain q. We repeat the procedure for $L\text{-}MAT_2$, $R\text{-}MAT_1$, and $R\text{-}MAT_2$.

The above procedure requires $O(\log n)$ time, as it is dominated by the time for point location in the query tree.

Local Updates. We need to move an interval I stored in a cell C so that it becomes an interval I', $O(1)$-similar to I. As in Sect. 2, we assume that we have a pointer to the cell C' that must store I'. Below we describe updating of the left marked-ancestor and marked-descendant trees only, as all other structures are either symmetric to those, or are updated exactly as in Sect. 2.

The *deletion* of I causes the following modifications. Let C_ℓ be the cell such that $|C_\ell| = |C|$ and C_ℓ contains the right endpoint of I. Cell C_ℓ is marked by I in either $L\text{-}MAT_1$ or in $L\text{-}MAT_2$. In the latter case C_ℓ gets unmarked, and nothing else should be done. In the former case, in addition to unmarking C_ℓ we must check whether such unmarking causes some cell marked in $L\text{-}MAT_2$ to change the marking level and start being marked in $L\text{-}MAT_1$ instead. Observe, that the only cell that could possibly change the marking level is the lowest marked ancestor C_a of C_ℓ in $L\text{-}MAT_2$. To check whether C_a changes the marking level, we perform the leftmost marked-descendant query for C_a. If that query returned a cell C_d, and I_a entirely covers I_d, where I_a and I_d are the intervals that mark C_a and C_d respectively, then C_a stays marked in $L\text{-}MAT_2$. Otherwise (i.e., if I_a does not cover I_d, or if C_a does not have the leftmost marked descendant), we unmark C_a in $L\text{-}MAT_2$ and mark it in $L\text{-}MAT_1$.

The *insertion* of I' causes the following modifications. The quadtree cells of size $|C'|$ intersected by I' get marked. Let C'_ℓ be the quadtree cell such that $|C'_\ell| = |C'|$ and C'_ℓ contains the right endpoint of I'. Cell C'_ℓ should be marked in one of the levels of the left marked-ancestor trees. To decide in which, we perform the leftmost marked-descendant query for C'_ℓ. If the query returns a cell C'_d, we check whether I' covers the interval I'_d that marks C'_d. If this is the case, C'_ℓ should be marked in $L\text{-}MAT_2$. Otherwise, C'_ℓ gets marked in $L\text{-}MAT_1$. In the latter case, we also check for the lowest ancestor C'_a of C'_ℓ in $L\text{-}MAT_1$, whether the marking of C'_ℓ causes a change of marking level of C'_a. Namely, if I' is contained in the interval I'_a that marks C'_a, then C'_a gets unmarked in $L\text{-}MAT_1$ and marked in $L\text{-}MAT_2$ instead.

Whenever a cell C is marked or unmarked in $L\text{-}MAT_1$, it is marked or unmarked in the left marked-descendant tree.

Correctness of the above procedure follows from Lemma 7. The time required by the procedure is dominated by the time for a constant number of lowest marked-ancestor queries and thus is $O(\log n/\log\log n)$, including the overhead due to compression, see Lemma 8. Classic updates are exactly the same as in Sect. 2. We conclude.

Theorem 2. *A set S of n intervals in $\mathbb{R}^1$, such that the ply of S is at most two, can be stored in a data structure of size $O(n)$, subject to stabbing queries, insertion and deletion of intervals in $O(\log n)$ time, and local updates in $O(\log n/\log\log n)$ time. The bound is amortized for the insertion, and worst-case for all other operations.*

3.2 Intervals with Higher Ply

For a set of intervals with ply $k \geq 2$, the above data structure can be generalized, resulting in the following.

Theorem 3. *A set S of n intervals in $\mathbb{R}^1$, that may overlap, can be stored in a data structure of size $O(n)$, subject to stabbing queries, insertion and deletion of intervals in $O(\log n + k\log n/\log\log n)$ time, and local updates in $O(k\log n/\log\log n)$ time, where k is the ply of S at the time of the operation. The bound is amortized for the insertion, and worst-case for all other operations.*

Proof. (sketch). We maintain k levels of marked-ancestor trees. For a quadtree cell C, if there is an interval I such that Condition 2(i) is satisfied for C and I, then C is marked in $L\text{-}MAT_j$, for some $1 \leq j \leq k$. Cell C is marked in $L\text{-}MAT_i$, $1 < i \leq k$, if there is a sequence $C_1, \ldots, C_{i-1}$ of descendants of C and a sequence $I_1, \ldots, I_{i-1}$ of intervals in S such that $I_1 \subset I_2 \subset \ldots \subset I_{i-1} \subset I$ and Condition 2(i) is satisfied for each pair C_j, I_j, $1 \leq j \leq i-1$. Otherwise, C is marked in $L\text{-}MAT_1$.

We also maintain $k-1$ marked-descendant trees, where the left marked-descendant tree of level i, $1 \leq i < k$ is defined as in Sect. 3.1 for the cells marked in $L\text{-}MAT_i$.

The operations and queries are simple generalizations of the ones from Sect. 3.1. For example, while doing the stabbing queries for a point q, we search in each marked-ancestor tree for at most k lowest marked ancestors of C, where C is the quadtree leaf containing q; the total number of marked-ancestor queries performed is $O(k)$ due to a generalization of Lemma 6. □

4 Compressing the Quadtree

The compressed quadtree contains a-compressed cells for some large constant a. An a-compressed cell C has only one child C', such that $|C'| \leq |C|/a$, and for any interval I in S whose center point is contained in $C \backslash C'$, I entirely covers C. The compressed nodes cut the quadtree into several *regular components* that are smaller (uncompressed) quadtrees. We need to modify our data structure from Sects. 2 and 3 to handle the presence of compressed cells in the quadtree. Lemma 8 discusses the complexity of such modifications. In particular, each update operation needs additional time to restore the properties of a compressed quadtree. We refer to such additional time as *overhead*.[5] Notice that stabbing queries do not modify the quadtree, therefore they do not cause any overhead.

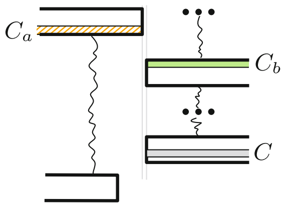

Fig. 7. Regular components of the quadtree (bounded by bold lines), and paths that connect the compressed nodes with their unique children (zigzag lines). Cell C_b is marked in $L'\text{-}MAT$ by the compressed cell C_a. For the cell C, its lowest marked ancestor in $L'\text{-}MAT$ is C_b.

Lemma 8. *Compression in the quadtree for intervals that possibly overlap, can be maintained in $O(n)$ space. The time overhead is $O(\log n/\log\log n)$ per local update operation, and $O(\log\log n)$ per classic update operation.*

Proof. Consider a local update that turns an interval I into an interval I', $O(1)$-similar to I. We need to quickly find the cell where I' should be stored. That cell should be a constant number of cells away from C, and in a non-compressed (or in a balanced) quadtree it (or the leaf where it should be inserted) could be

[5] Note that the analysis in Sects. 2 and 3 already takes Lemma 8 into account.

found by following the level links from C to the cells of size $|C|$ adjacent to C. In our setting, it may happen that such cell C'' does not exist, and the smallest cell C_a above C'' is a compressed cell. In that case, once C_a is found, the necessary decompression can be done in $O(1)$ time.

To perform the search for C_a efficiently, we maintain two additional marked-ancestor trees on top of the quadtree. These trees are denoted L'-MAT and R'-MAT, and correspond to the following marking rule. A cell C_a is marking a cell C_b in L'-MAT if C_a is the rightmost leaf in its regular quadtree component, and C_b is the largest cell adjacent to the right border of C_a, such that $|C_b| \leq |C_a|$, and C_b is to the right of C_a. See Fig. 7. Marking in R'-MAT is symmetric.

For a cell C (see Fig. 7), going to the lowest marked ancestor C_b of C in L'-MAT, and then to the cell that marks C_b, results exactly in the sought compressed cell C_a. Thus the time needed to restore the properties of a compressed quadtree after a local update is dominated by the time to find the lowest marked ancestor in L'-MAT or R'-MAT. By Lemma 3, this is $O(\log n / \log\log n)$.

A classic update causes $O(1)$ compression/decompression operations like in the case of non-overlapping intervals [8], which requires $O(1)$ time, and $O(1)$ marking/unmarking operations in L'-MAT and R'-MAT, which by Lemma 3 can be performed in $O(\log\log n)$ time in total. □

Acknowledgements. We wish to thank Irina Kostitsyna for helpful discussions.

References

1. Agarwal, P.K., Arge, L., Kaplan, H., Molad, E., Tarjan, R.E., Yi, K.: An optimal dynamic data structure for stabbing-semigroup queries. SIAM J. Comput. **41**(1), 104–127 (2012)
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: 39th Annual Symposium on Foundations of Computer Science, pp. 534–543 (1998)
3. Arge, L., Vitter, J.S.: Optimal dynamic interval management in external memory. In: 37th Conference on Foundations of Computer Science, pp. 560–569 (1996)
4. Berg, M., Cheong, O., Kreveld, M., Overmars, M.: Computational Geometry - Algorithms and Applications, 3rd edn. Springer, Heidelberg (2008)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
6. Edelsbrunner, H.: Dynamic data structures for orthogonal intersection queries. Report F59. Technische Universität Graz (1980)
7. Kaplan, H., Molad, E., Tarjan, R.: Dynamic rectangular intersection with priorities. In: 35th ACM Symposium on Theory of Computing (STOC), pp. 639–648 (2003)
8. Löffler, M., Simons, J.A., Strash, D.: Dynamic planar point location with sub-logarithmic local updates. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 499–511. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40104-6_43
9. McCreight, E.M.: Efficient algorithms for enumerating intersecting intervals and rectangles. report csl-80-9. Technical report, Xerox Palo Alto Res. Center (1980)
10. McCreight, E.M.: Priority search trees. SIAM J. Comput. **14**(2), 257–276 (1985)

11. Nekrich, Y.: Data structures with local update operations. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 138–147. Springer, Heidelberg (2008). doi:10.1007/978-3-540-69903-3_14
12. Nekrich, Y.: A dynamic stabbing-max data structure with sub-logarithmic query time. In: Asano, T., Nakano, S., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 170–179. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25591-5_19
13. Thorup, M.: Space efficient dynamic stabbing with fast queries. In: 35th ACM Symposium on Theory of Computing (STOC), pp. 649–658. ACM Press (2003)