Dynamic Planar Range Maxima Queries

Gerth Stølting Brodal[1] and Konstantinos Tsakalidis[1]

MADALGO[⋆],
Department of Computer Science,
Aarhus University, Denmark.
{gerth, tsakalid}@madalgo.au.dk

Abstract. We consider the dynamic two-dimensional maxima query problem. Let P be a set of n points in the plane. A point is *maximal* if it is not dominated by any other point in P. We describe two data structures that support the reporting of the t maximal points that dominate a given query point, and allow for insertions and deletions of points in P. In the pointer machine model we present a linear space data structure with $O(\log n + t)$ worst case query time and $O(\log n)$ worst case update time. This is the first dynamic data structure for the planar maxima dominance query problem that achieves these bounds in the worst case. The data structure also supports the more general query of reporting the maximal points among the points that lie in a given 3-sided orthogonal range unbounded from above in the same complexity. We can support 4-sided queries in $O(\log^2 n + t)$ worst case time, and $O(\log^2 n)$ worst case update time, using $O(n \log n)$ space, where t is the size of the output. This improves the worst case deletion time of the dynamic rectangular visibility query problem from $O(\log^3 n)$ to $O(\log^2 n)$. We adapt the data structure to the RAM model with word size w, where the coordinates of the points are integers in the range $U=\{0,\ldots,2^w-1\}$. We present a linear space data structure that supports 3-sided range maxima queries in $O(\frac{\log n}{\log \log n}+t)$ worst case time and updates in $O(\frac{\log n}{\log \log n})$ worst case time. These are the first sublogarithmic worst case bounds for all operations in the RAM model.

1 Introduction

Given a set P of n points in the plane, a point $p = (p_x, p_y)$ *dominates* another point $q = (q_x, q_y)$ if both $p_x \geq q_x$ and $p_y \geq q_y$ hold. The m points that are not dominated by any other point in the set are called *maximal*. The maximal points are also called the *staircase* of the set, since when they are sorted by increasing x-coordinate they are also sorted by decreasing y-coordinate. We consider the problem of designing a data structure that supports insertions and deletions of points in the set, and allows reporting the maximal points that dominate a given query point q (*maxima dominance* query). Actually we consider the more

[⋆] Center for Massive Data Algorithmics - a Center of the Danish National Research Foundation

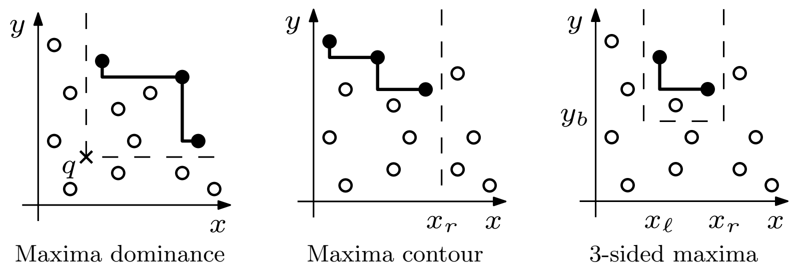

Fig. 1. Different range maxima queries. Black points are reported.

general *3-sided range maxima* queries. That is, given parameters x_ℓ, x_r and y_b, to report the maximal points of the points in the set $P \cap ([x_\ell, x_r] \times [y_b, +\infty[$. The special case where the query range is $]-\infty, x_r]\times]-\infty, +\infty[$ is also known as *maxima contour* queries.

Previous Results The maximal points of a static set of two-dimensional points can be computed in optimal $O(n \log n)$ time [2].[1] In the pointer machine linear space dynamic data structures have been presented that support reporting all maximal points. They achieve $O(m)$ worst case [8, 9, 3, 13] or amortized [15] query time. They support deciding whether a given query point lies above or below the staircase (*maxima emptiness* query) in $O(\log n)$ [8, 9, 15] or $O(\log m)$ [3, 13] worst case time. Maxima contour queries are supported by the structures of [3] and [9] in $O(\log n + t)$ worst case time, where t is size of the output.

Regarding updates, the structure of Overmars and van Leeuwen [3] supports the insertion and deletion of an arbitrary point in $O(\log^2 n)$ worst case time. Frederickson and Rodger [8], and independently Janardan [9], improve only the worst case insertion time to $O(\log n)$, preserving the other complexities. D'Amore et al. [13] improve both the insertion and deletion time to $O(\log n)$ worst case, under the assumption that the updated point has the maximum or minimum x-coordinate among the points in the set (boundary updates). Kapoor [15] shows how to support both the insertion and deletion of an arbitrary point in $O(\log n)$ worst case time. The deletion time is improved at the expense of worst case query time $O(m+\mathit{chng} \cdot \log n)$, where *chng* is the total number of changes that the update operations have caused to the staircase reported by the latest query operation. Kapoor modifies the structure such that a sequence of queries, and n insertions and d deletions of points requires $O(n \log n + d \log n + r)$ worst case time, where r is the total number of reported maximal points. The worst case update time remains $O(\log n)$, however the query needs $O(r)$ amortized time.

An application of the maxima contour query is the rectangular visibility query problem. A point $p \in P$ is *rectangularly visible* from a point q if the orthogonal rectangle with p and q as diagonally opposite corners contains no other point in P. The structure of Overmars and Wood [5] supports reporting

[1] $\log n = \log_2 n$

Table 1. Running times for updates and the following query operations. Parameter t is the number of reported points. "All": Report all maximal points. "Emptiness": Is the query point above or below the staircase? "Dominance": Report the maximal points that dominate the query point. "Contour": Report the maximal points among the points that lie to the left of a given vertical query line. All structures occupy linear space. (†only boundary updates, ‡amortized bounds)

	Model	All	Emptiness	Dominance	Contour	Insertion	Deletion
[3]	PM	$O(m)$	$O(\log m)$	$O(\log m + t)$	$O(\log n + t)$	$O(\log^2 n)$	$O(\log^2 n)$
[8]	PM	$O(m)$	$O(\log n)$	$O(\log n + t)$	$O(\min\{\log^2 n + t, (t+1)\log n\})$	$O(\log n)$	$O(\log^2 n)$
[9]	PM	$O(m)$	$O(\log n)$	$O(\log n + t)$	$O(\log n + t)$	$O(\log n)$	$O(\log^2 n)$
[13]	PM	$O(m)$	$O(\log m)$	$O(\log m + t)$	-	$O(\log n)$	$O(\log n)^{\dagger}$
[15]	PM‡	$O(m)$	$O(\log n)$	$O(\log n + t)$	-	$O(\log n)$	$O(\log n)$
New	PM	$O(m)$	$O(\log n)$	$O(\log n + t)$	$O(\log n + t)$	$O(\log n)$	$O(\log n)$
New	RAM	$O(m)$	$O(\frac{\log n}{\log\log n})$	$O(\frac{\log n}{\log\log n} + t)$	$O(\frac{\log n}{\log\log n} + t)$	$O(\frac{\log n}{\log\log n})$	$O(\frac{\log n}{\log\log n})$

the t points that are rectangularly visible from a given query point in $O(\log^2 n+t)$ worst case time, insertions and deletions of points in $O(\log^3 n)$ worst case time and uses $O(n \log n)$ space, when the structure of [3] is applied. Only the insertion time is improved to $O(\log^2 n)$ by applying the structure of [9]. Both the insertion and deletion time can be improved to $O(\log^2 n)$ worst case, using $O(n \log n)$ space, at the expense of $O(t \log n)$ worst case query time [5, Theorem 3.5].

Our Results In the pointer machine model we present a linear space data structure that supports maxima dominance queries, and more general 3-sided range maxima queries in $O(\log n+t)$ worst case time. An arbitrary point can be inserted and deleted in $O(\log n)$ worst case time. This is the first dynamic data structure that achieves the update times of [15], and supports more general range maxima queries with worst case complexities. Our structure can be generalized to solve *4-sided range maxima* queries, namely to report the maximal points of the point set $P \cap ([x_\ell, x_r] \times [y_b, y_t])$, in $O(\log^2 n+t)$ worst case time. Updates take $O(\log^2 n)$ worst case time and the space usage is $O(n \log n)$. Using our 4-sided range maxima structure we can solve the dynamic rectangular visibility query problem with the same bounds. In the word-RAM model we present a linear space data structure that supports 3-sided range maxima queries in $O(\frac{\log n}{\log\log n} + t)$ worst case time, and updates in $O(\frac{\log n}{\log\log n})$ worst case time. This is the first dynamic data structure in the RAM model that supports these operations in sublogarithmic worst case time. Both the pointer machine and the RAM data structures support reporting all maximal points in $O(m)$ worst case time.

Outline of Solution We follow the basic approach of previous structures. We store the points sorted by x-coordinate at the leaves of a tree, and maintain a tournament in the internal nodes of the tree with respect to their y-coordinates. An internal node u is assigned the point with maximum y-coordinate in its subtree. We observe that the nodes in the subtree of u that contain this point form a path. We also observe that the maximal points among the points assigned to the nodes hanging to the right of this path are also maximal among the points in the subtree of u. We store these points in node u in order to allow the query algorithm to recursively access only points to be reported.

The implementation of our structures is based on the fact that we can obtain the set of points we store in a node from the set stored in its child node that belongs to the same path. In particular if that is the left child, we need to delete the points that are dominated by the point assigned to the right child and insert this point into the set. Sundar [7] shows how to implement a priority queue that supports exactly this operation (*attrition*) in $O(1)$ worst case time. This allows us to complete the insertion and deletion of a point to the construction in $O(\log n)$ worst case time. To achieve linear space we implement every path that contains the same point as a *partially persistent* priority queue with attrition. The priority queue with attrition abides by the assumptions for the technique of Brodal [12] that makes a pointer-based data structure partially persistent, since it can be implemented as a doubly linked list pointed by a constant number of additional pointers. We adapt the above construction to the word-RAM by increasing the degree of the tree to $\Theta(\log^{\varepsilon} n)$ for some $0<\varepsilon<1$. To search and update a node in $O(1)$ time, we make use of precomputed lookup-tables and the *Q-heap* of Fredman and Willard [10].

2 Preliminaries

Partial Persistence Driscoll et al. [6] show a general technique to make pointer based data structures *partially persistent.* Suppose that an update operation on a dynamic data structure creates a new version of the data structure. A dynamic data structure is called partially persistent when only the latest version can be updated, and any previous version can be queried given a pointer to it. We obtain a list of versions called the *version list*, by ordering the versions such that an update to version i creates version $i+1$. A *rollback* discards the latest version by reversing the update that created it, and sets its preceding version in the version list as the new updatable version.

Let D be a dynamic data structure that supports queries in worst case time q and updates in worst case time u. Under the assumption that D can be modeled by a graph where the in- and out-degree of each node is bounded by a constant, Brodal [12] presented a method to make D partially persistent, such that a query to a particular version can be supported in $O(q)$ worst case time, and an update to the latest version in $O(u)$ worst case time. This improves the original partial persistence technique of Driscoll et al. [6], that only achieves amortized $O(u)$ update time, and enables the rollback operation in $O(u)$ worst case time. Moreover after performing a sequence of s atomic update operations, the partially persistent data structure occupies $O(s)$ space.

Priority Queue with Attrition Sundar [7] introduces the *priority queue with attrition (PQA)* that supports the following operations in $O(1)$ worst case time on a set of elements drawn from a total order: DELETEMAX deletes and returns the maximum element from the PQA, and INSERTANDATTRITE(x) inserts element x into the PQA and removes all elements smaller than x from the PQA. The PQA uses space linear to the number of inserted elements, and is implemented as a doubly linked list with a constant number of additional pointers. Given a value x, the elements in the PQA that are larger than x can be reported in sorted order in $O(t+1)$ time, where t is the size of the output.

Red-black Tree A *red-black tree* [1] is a balanced binary search tree that maintains a set of n elements drawn from a total order. It supports the following operations in $O(\log n)$ worst case time, using $O(n)$ space: INSERT(x) inserts element x into the tree, DELETE(x) deletes element x from the tree, and PREDECESSOR(x) returns the largest element in the tree that is smaller or equal to element x.

Q-Heap A *Q-heap* [10] stores a subset of at most $\log^{1/4} n$ integers from the set $U=\{0,\ldots,2^w-1\}$. It operates in the RAM with word size $w \geq \log n$, and supports the operations INSERT(x), DELETE(x) and PREDECESSOR(x) in $O(1)$ worst case time. It uses $O(\log^{1/4} n)$ space, and utilizes an $O(n)$ space global precomputed lookup-table that needs $O(n)$ preprocessing time.

3 Pointer-Based Data Structure

In this section we present our pointer based data structure that supports 3-sided range maxima queries in $O(\log n + t)$ worst case time, where t is the number of reported points. The insertion and deletion of a point are supported in $O(\log n)$ worst case time. The structure occupies $O(n)$ space. Comparisons are the only allowed computation on the coordinates of the points.

3.1 Data Structure

We store the points sorted by increasing x-coordinate at the leaves of a red-black tree T. We maintain a *tournament* on the internal nodes of T with respect to the y-coordinates of the points. In particular, every internal node u contains the points $p_{x-\max}(u)$ and $p_{y-\max}(u)$ that are respectively the points with maximum x- and y-coordinate among the points in the subtree of u. The points $p_{x-\max}(u)$ allow us to search in T with respect to the x-coordinate. The points $p_{y-\max}(u)$ define n disjoint *winning paths* in T whose nodes contain the same point.

Let u be an internal node. Let u belong to a winning path π. We denote by π_u the suffix of π that starts at node u and ends at the leaf that stores $p_{y-\max}(u)$. We denote by R_u the set of right children of the nodes in π_u that do not belong to π_u themselves. We denote by MAX(u) the points that are maximal among $\{p_{y-\max}(v) \mid v \in R_u\}$. We can obtain MAX($u$) from the set of points MAX(c), where c is the child of u that belongs to π_u. In particular, if that is the right child u_R then $R_u = R_{u_R}$ and thus MAX(u) = MAX(u_R). Otherwise if that is the left child u_L, then $R_u = R_{u_L} \cup \{u_R\}$. Point $p_{y-\max}(u_R)$ belongs to MAX(u)

since it has the largest x-coordinate among all points in R_u. Moreover, all the points in $\mathrm{MAX}(u_L)$ with y-coordinate at most $p_{y-\max}(u_R)_y$ should be excluded from $\mathrm{MAX}(u)$ since they are dominated by $p_{y-\max}(u_R)$. See Figure 2 for an illustration of R_u and $\mathrm{MAX}(u)$.

We implement the sets $\mathrm{MAX}(u)$ of the nodes u along a winning path π as the versions of a *partially persistent priority queue with attrition* (PPPQA). That is, every internal node u stores the y-coordinates of the points in $\mathrm{MAX}(u)$ in a priority queue with attrition (PQA). If u is the i-th node of π, then it contains the i-th version of the PPPQA. The leaf of π contains version 0. The child of u that belongs to π contains the $i-1$-th version. If that is u_R, the i-th version and the $i-1$-th versions have the same content. Else if that is u_L, the i-th version is obtained by executing $\textsc{InsertAndAttrite}(p_{y-\max}(u_R)_y)$ to the $i-1$-th version. Moreover, for every point in $\mathrm{MAX}(u)$ we store a pointer to the node of R_u that stores the point. This node is the highest node of its winning path. Note that R_u is not explicitly maintained anywhere in the data structure.

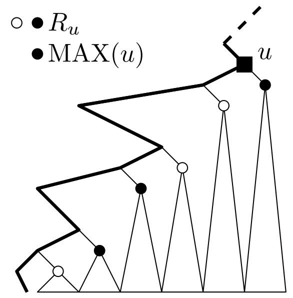

Fig. 2. The winning path π is bold. The circular nodes are in R_u. The black circular nodes are in $\mathrm{MAX}(u)$.

Let p be the point with maximum y-coordinate among the points $p_{y-\max}(u_L)$ and $p_{y-\max}(u_R)$. To *extend* the winning path that contains p from the child node to node u, we assign to u the points $p_{x-\max}(u_R)$ and p, and compute $\mathrm{MAX}(u)$.

Lemma 1. *Extending a winning path from an internal node to its parent node needs $O(1)$ time and $O(1)$ extra space.*

Proof. Points $p_{x-\max}(u)$ and $p_{y-\max}(u)$ can be computed from the children of u in $O(1)$ time. To extend the winning path that contains $p_{y-\max}(u_R)$, we create the reference to the new version of the PPPQA for the winning path in $O(1)$ time. To extend the winning path that contains $p_{y-\max}(u_L)$, we record persistently the operation $\textsc{InsertAndAttrite}(p_{y-\max}(u_R)_y)$ and create a reference to the new version in $O(1)$ worst case time using $O(1)$ extra space [12, 7]. $\square$

Lemma 1 implies that T can be constructed bottom-up in $O(n)$ worst case time, assuming that the n points are given sorted by increasing x-coordinate. The total space usage is $O(n)$.

3.2 Query

In the following we describe how to answer 3-sided range maxima queries. This immediately also gives us maxima dominance and maxima contour queries, since these are special cases where the query ranges are $[q_x, +\infty[\times[q_y, +\infty[$ and $]-\infty, x_r]\times]-\infty, +\infty[$ respectively.

Reporting the maximal points that lie above q_y We first show how to report all maximal points with y-coordinate larger than a given value q_y among the points that belong to a subtree T_u of T rooted at an internal node u. If $p_{y-\max}(u)_y \leq q_y$ we terminate since no maximal point of T_u has to be reported. Otherwise we report $p_{y-\max}(u)$ and compute the prefix $p[1], \dots, p[k]$ of $\mathrm{MAX}(u)$ of points with y-coordinate larger than q_y. To do so we report the elements of the PQA of u that are larger than q_y. Let u_i be the node of R_u such that $p_{y-\max}(u_i) = p[i]$. We report recursively the maximal points in T_{u_i} with y-coordinate larger than $p[i+1]_y$ for $i \in \{1, \dots, k-1\}$, and larger than q_y for $i = k$. The algorithm reports the maximal points in T_u in decreasing y-coordinate and terminates when the maximal point with the smallest y-coordinate larger than q_y is reported.

For the correctness of the above observe that the point $p_{y-\max}(u)$ is the leftmost maximal point among the points in T_u. The x-coordinates of the rest of the maximal points in T_u are larger than $p_{y-\max}(u)_x$. The subtrees rooted at nodes of R_u divide the x-range $]p_{y-\max}(u)_x, +\infty[$ into disjoint x-ranges. The point p' with maximum y-coordinate of each x-range is stored at a node u' of R_u. The points in $\mathrm{MAX}(u)$ are maximal among the points in T_u. Let $p[i+1]$ be the leftmost maximal point in T_u to the right of p'. If p' does not belong to $\mathrm{MAX}(u)$ then none of the points in $T_{u'}$ are maximal, since $p'_y \leq p[i+1]_y$ and $p[i+1]_x$ is larger than the x-coordinate of all points in $T_{u'}$. Otherwise p' belongs to $\mathrm{MAX}(u)$ and more precisely $p' = p[i]$. Point $p[i]$ is the leftmost maximal point among the points in $T_{u'}$. The maximal points among the points in T_u with x-coordinate at least $p[i]_x$ and y-coordinate larger than $p[i+1]_y$ belong to $T_{u'}$. In particular, they are the maximal points among the points in $T_{u'}$ with y-coordinate larger than $p[i+1]_y$.

Lemma 2. *Reporting the maximal points with y-coordinate larger than q_y among the points of a subtree of T takes $O(t+1)$ worst case time, where t is the number of reported points.*

Proof. If $p_{y-\max}(u)$ is not reported we spend in total $O(1)$ time to assess that no point will be reported. Otherwise $p_{y-\max}(u)$ is reported. We need $O(k+1)$ time to compute the k points in $\mathrm{MAX}(u)$ with y-coordinate larger than q_y. If $k = 0$ we charge the $O(1)$ time we spent in node u to the reporting of $p_{y-\max}(u)$. Otherwise we charge the time to compute $p[i]$ and to access node u_i to the reporting of $p[i]$. In total every reported point is charged $O(1)$ time. □

In order to report all maximal points of P we apply the above algorithm to the root of T with $q_y{=}-\infty$. The total time is $O(m)$.

3-sided Range Maxima Queries We show how to report all maximal points of the point set $P \cap ([x_\ell, x_r] \times [y_b, +\infty[)$. We search for the leaves ℓ and r of T that contain the points with the largest x-coordinate smaller than x_r and smallest x-coordinate larger than x_ℓ, respectively. Let π_ℓ and π_r be the root-to-leaf paths to ℓ and r, respectively. Let R denote the set of right children of nodes of $\pi_\ell \setminus \pi_r$ that do not belong to π_ℓ themselves, and the set of left children of nodes of $\pi_r \setminus \pi_\ell$ that do not belong to π_r themselves. The subtrees rooted at the nodes u

of R divide the x-range $]x_\ell, x_r[$ into disjoint x-ranges. We compute the maximal points $p[1], \ldots, p[k]$ among the points $p_{y-\max}(u)$ in R with y-coordinate larger than y_b using [2]. Let u_i be the node of R such that $p_{y-\max}(u_i) = p[i]$. We report recursively the maximal points in T_{u_i} with y-coordinate larger than $p[i+1]_y$ for $i \in \{1, \ldots, k-1\}$, and larger than y_b for $i = k$.

Lemma 3. *Reporting the maximal points for the 3-sided range maxima query takes $O(\log n + t)$ worst case time, where t is the number of reported points*

Proof. There are $O(\log n)$ nodes u in R. The points $p_{y-\max}(u)$ are accessed in decreasing x-coordinate. Therefore we can compute the maximal points $p[i], i \in \{1, \ldots, k\}$ in $O(\log n)$ time [2]. Let $p[i] = p_{y-\max}(u_i)$ for a particular node u_i of R, and let there be t_i maximal points to be reported in the subtree T_{u_i}. Since $p[i]$ will be reported we get that $t_i \geq 1$. By Lemma 2 we need $O(t_i)$ time to report the t_i points. Therefore the total worst case time to report the $t = \sum_{i=1}^{k} t_i$ maximal points is $O(\log n + t)$. □

In order to answer whether a given query point $q = (q_x, q_y)$ lies above or below the staircase (maxima emptiness query) in $O(\log n)$ time we terminate a 3-sided range maxima query for the range $[q_x, +\infty[\times [q_y, +\infty[$ as soon as the first maximal point is reported. If so, then q lies below the staircase. Else if no point is reported then q lies above the staircase.

3.3 Update

To insert (resp. delete) a point $p = (p_x, p_y)$ in the structure, we search for the leaf ℓ of T that contains the point with the largest x-coordinate smaller than p_x (resp. contains p). We traverse the nodes of the parent(ℓ)-to-root search path π top-down. For each node u of π we discard the points $p_{x-\max}(u)$ and $p_{y-\max}(u)$, and rollback the PQA of u in order to discard MAX(u). We insert a new leaf ℓ' for p immediately to the right of the leaf ℓ (resp. delete ℓ) and we rebalance T. Before performing a rotation, we discard the information stored in the nodes that participate in it. Finally we recompute the information in each node u missing $p_{x-\max}(u)$, $p_{y-\max}(u)$, and MAX(u) bottom-up following π. For every node u we extend the winning path of its child u' such that $p_{y-\max}(u) = p_{y-\max}(u')$ as in Section 3.1. Correctness follows from the fact that every node that participates in a rotation either belongs to π or is adjacent to a node of π. The winning path that ends at the rotated node will be considered when we extend the winning paths bottom-up.

Lemma 4. *Inserting or deleting a point from T takes $O(\log n)$ worst case time.*

Proof. The height of T is $O(\log n)$. Rebalancing a red-black takes $O(\log n)$ time. We need $O(1)$ time to discard the information in every node of π. By Lemma 1 we need $O(1)$ time to extend the winning path at every recomputed node. Thus we need in total $O(\log n)$ time to update the internal nodes of T. □

4 4-sided Range Maxima Queries and Rectangular Visibility

In the following we describe how to support 4-sided range maxima queries in $O(\log^2 n + t)$ worst case time and updates in $O(\log^2 n)$ worst case time, using $O(n \log n)$ space, by borrowing ideas from [5].

For the rectangular visibility query problem, where we want to report the points that are rectangularly visible from a given query point $q = (q_x, q_y)$, we note that it suffices to report the maximal points p that lie to the lower left quadrant defined by point q (namely the points where $p_x \leq q_x$ and $p_y \leq q_y$ holds). The rectangularly visible points of the three other quadrants can be found in a symmetric way. This corresponds exactly to a 4-sided range maxima query for the range $]-\infty, q_x] \times]-\infty, q_y]$. The rectangular visibility query problem can be solved in the same bounds, using four 4-sided range maxima structures.

To support 4-sided range maxima queries, as in [5], we store all points sorted by increasing y-coordinate at the leaves of a weight-balanced B-tree S [16]. In every internal node u of S we associate a secondary structure that can answer 3-sided range maxima queries on the points that belong to the subtree S_u of S.

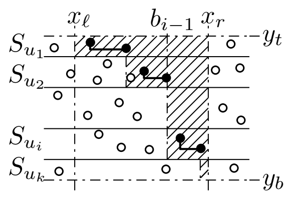

Fig. 3. 4-sided range maxima query.

To perform a 4-sided range maxima query we first search for the leaves of S that contain the point with the smallest y-coordinate larger than y_b and the point with largest y-coordinate smaller than y_t, respectively. Let π_b and π_t be the root-to-leaf search paths respectively. Let L denote the set of nodes of S that are left children of the nodes in $\pi_t \setminus \pi_b$ and do not belong to π_t themselves, and the set of nodes of S that are right children of the nodes in $\pi_b \setminus \pi_t$ and do not belong to π_b themselves. The subtrees rooted at the nodes of L divide the y-range $]y_b, y_t[$ into $O(\log n)$ disjoint y-ranges. We consider the nodes $u_1, \ldots, u_k$ of L in decreasing y-coordinate. In the secondary structure of u_1 we perform a 3-sided range maxima query with the range $]x_\ell, x_r] \times]-\infty, +\infty[$. In the secondary structure of every other node u_i of L we perform a 3-sided range maxima query with the range $]b_{i-1}, x_r] \times]-\infty, +\infty[$, where b_{i-1} is the x-coordinate of the last point reported from the secondary structures of $u_1, \ldots, u_{i-1}$, i.e. the rightmost point that lies to the left of x_r and lies in y-range spanned by the subtrees $S_{u_1}, \ldots, S_{u_{i-1}}$. See Figure 3 for the decomposition of a 4-sided query.

To insert (resp. delete) a point in S, we first search S and create a new leaf (resp. delete the leaf) for the point. Then we insert (resp. delete) the point to the $O(\log n)$ secondary structures that lie on the search path. Finally we rebalance the weight-balanced B-tree S and reconstruct the secondary structures at rebalanced nodes.

Theorem 1. *Given a set of n points in the plane, the 4-sided range maxima query problem can be solved using $O(n \log n)$ space, $O(\log^2 n + t)$ worst case query time, and $O(\log^2 n)$ worst case update time, where t is the output size.*

Proof. There are $k{=}O(\log n)$ nodes in L where we execute a 3-sided range maxima query. By Lemma 3 each such query takes $O(\log n + t_i)$ worst case time, where t_i is the number of reported points. In total the worst case time to report the $t = \sum_{i=1}^{k} t_i$ points is $O(\log^2 n + t)$. Each point p occurs once in every secondary structure of a node of S that is an ancestor of the leaf that contains p. There are $O(\log n)$ such ancestor nodes, thus the total space is $O(n \log n)$. By Lemma 4 we need $O(\log n)$ time to insert and delete a point p from each secondary structure at an ancestor node. The worst case time to insert and delete p to the secondary structures that contain it is $O(\log^2 n)$. When the incremental rebalancing algorithm of [16] is applied, then for each insertion and deletion to S, $O(1)$ updates are applied to the secondary structures at each of $O(\log n)$ levels of S. By Lemma 4 each such update needs $O(\log n)$ worst case time, thus the rebalancing costs $O(\log^2 n)$ worst case time. □

5 3-sided Range Maxima in the RAM Model

In this section we consider the RAM model of computation, where the coordinates are represented by a word of $w \geq \log n$ bits and belong to the range of integers $U = \{0, 1, \ldots, 2^w - 1\}$. We present a data structure that supports 3-sided range maxima queries in $O(\frac{\log n}{\log \log n} + t)$ worst case time, where t is the number of reported points. The insertion and deletion of a point are supported in $O(\frac{\log n}{\log \log n})$ worst case time. The occupied space is linear to the number of stored points.

Structure We store the points of P sorted by increasing x-coordinate at the leaves of an (a, b)-tree T, such that $a = \frac{1}{2} \log^{\frac{1}{4}} n$ and $b = \log^{\frac{1}{4}} n$. The height of T is $O(\frac{\log n}{\log \log n})$. Every node u is assigned the points $p_{y-\max}(u)$ and $p_{x-\max}(u)$. Define $C_y(u) = \{p_{y-\max}(u_i)_y \mid u_i \text{ is a child of } u\}$ and similarly let $C_x(u)$ be the x-coordinates of the points with maximum x-coordinate assigned to the children of u. In every internal node u we store $C_x(u)$ and $C_y(u)$ in two Q-heaps $X(u)$ and $Y(u)$, respectively. We store two global lookup-tables for the Q-heaps. We store a global look-up table S that supports 3-sided range maxima queries for a set of at most $\log^{\frac{1}{4}} n$ points in rank-space. Every node u stores a reference to the entry of S that corresponds to the permutation in $C_y(u)$. We adopt the definitions for the winning paths and MAX(u) from Section 3.1, with the difference that now R_u denotes the children of nodes v of π_u that contain the second maximal point in $c_y(v)$. As in Section 3.1, we implement the sets MAX(u) of the nodes u along a winning path as the versions of a PPPQA. For every point p in MAX(u) we store a pointer to the node v of π_u whose child is assigned p. The tree and the look-up tables use $O(n)$ space.

Query To report the maximal points in a subtree T_u with y-coordinate larger than q_y, we first compute the prefix $p[1], \ldots, p[k]$ of the points in MAX(u) with y-coordinate larger than q_y, as in Section 3.1. For each computed point $p[i]$ we visit the node v of π_u whose child is assigned $p[i]$. We use the $Y(v)$ and the reference to S to compute the maximal points $p'[j], j \in \{1,\ldots,k'\}$ among the points in $C_y(v)$ that lie in the range $]p[i]_x,+\infty[\times]p[i+1]_y,+\infty[$. Let v_j be the child of v such that $p'[j]=p_{y-\max}(v_j)$. We recursively report the maximal points in the subtree T_{v_j} with y-coordinate larger than $p'[j+1]_y$ for $j\in\{1,\ldots,k'-1\}$, and larger than $p[i+1]_y$ for $j=k'$. If $i=k$ and $j=k'$ we recursively report the points in T_{v_j} with y-coordinate larger than q_y. Correctness derives from the fact that $p[i]_x<p'[1]_x<\cdots<p'[k']_x<p[i+1]_x$ and $p[i]_y>p'[1]_y>\cdots>p'[k']_y>p[i+1]_y$ hold, since $p[i]$ and $p'[1]$ are respectively the first and the second maximal points among the points in $C_y(v)$. The worst case query time is $O(t)$.

To answer a 3-sided range maxima query, we first use the Q-heaps for the x-coordinates in order identify the nodes on the paths π_ℓ and π_r to the leaves that contain the points with smallest x-coordinate larger than x_ℓ and with largest x-coordinate smaller than x_ℓ, respectively. This takes $O(\frac{\log n}{\log\log n})$ worst case time. For each node on π_ℓ and π_r we identify the interval of children that are contained in the x-range of the query. For each interval we identify the child c with maximum $p_{y-\max}(c)$ using S in $O(1)$ time. These elements define the set R from which we compute the maximal points using [2] in $O(\frac{\log n}{\log\log n})$ worst case time. We apply the above modified algorithm to this set, ignoring the maximal points $p'[j]$ outside the x-range of the query. The worst case time for 3-sided range maxima query is $O(\frac{\log n}{\log\log n}+t)$.

Update To insert and delete a point from T we proceed as in Section 3.1. To extend a winning path to a node v we first find the child u of v with the maximum $p_{y-\max}(u)_y$ using $Y(v)$, i.e. the winning path of u will be extended to v. Then we set $p_{x-\max}(v)=p_{x-\max}(u_k)$ where u_k is the rightmost child of v, we set $p_{y-\max}(v)=p_{y-\max}(u)$, insert $p_{x-\max}(u)_x$ and $p_{y-\max}(u)_y$ to $X(v)$ and $Y(v)$ respectively, we recompute the reference to the table S using $Y(v)$, and we recompute $\text{MAX}(v)$ from $\text{MAX}(u)$ as in Section 3.1. In order to discard the information from a node u we remove $p_{x-\max}(u)$ and $p_{y-\max}(u)$ from node u and from the Q-heaps $X(v)$ and $Y(v)$ respectively, and rollback $\text{MAX}(u)$. These operations take $O(1)$ worst case time. Rebalancing involves splitting a node and merging adjacent sibling nodes. To facilitate these operation in $O(1)$ worst case time we execute them incrementally in advance, by representing a node as a pair of nodes. Therefore we consider each Q-heap as two separate parts Q_ℓ and Q_r, and maintain the size of Q_ℓ to be exactly a. In particular, whenever we insert an element to Q_ℓ, we remove the rightmost element from Q_ℓ and insert it to Q_r. Whenever we remove an element to Q_ℓ, we remove the leftmost element from Q_r and insert it to Q_ℓ. In case Q_r is empty we remove the rightmost or leftmost element from the immediately left or right sibling node respectively and insert it to Q_ℓ. Otherwise if both sibling nodes have a elements in their Q-heaps, we merge them. The total worst case time for an update is $O(\frac{\log n}{\log\log n})$ since we spend $O(1)$ time per node.

Theorem 2. *Given a set of n planar points with integer coordinates in the range $U=\{0,1,\ldots,2^w-1\}$, the 3-sided range maxima query problem can be solved in* *the RAM model with word size w using $O(n)$ space, $O(\frac{\log n}{\log\log n}+t)$ worst case query time, and $O(\frac{\log n}{\log\log n})$ worst case update time, where t is the output size.*

6 Conclusion

In the comparison model, it can be shown that $O(\log n)$ update time is optimal for the attained query time, by reduction from the Insert/Delete/FindMax problem [11]. In the cell probe model, it can be shown that $O(\frac{\log n}{\log\log n})$ time for the maxima emptiness query of the RAM structure is optimal for the attained update time, by equivalence to the dynamic planar dominance emptiness problem [14]. Reporting maximal or rectangularly visible points of dimension larger than two, admits logarithmic factors on the output-sensitive part of the query complexity [5, 15]. Improving them even for the static case remains an open problem.

References

1. R. Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. Acta Inf. 1: 290-306 (1972)
2. H. T. Kung, F. Luccio, F. P. Preparata. On Finding the Maxima of a Set of Vectors. J. ACM 22(4): 469-476 (1975)
3. M. H. Overmars, J. van Leeuwen. Maintenance of Configurations in the Plane. J. Comput. Syst. Sci. 23(2): 166-204 (1981)
4. S. Huddleston, K. Mehlhorn: A New Data Structure for Representing Sorted Lists. Acta Inf. 17: 157-184 (1982)
5. M. H. Overmars, D. Wood. On Rectangular Visibility. J. Alg. 9(3): 372-390 (1988)
6. J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan. Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124 (1989)
7. R. Sundar. Worst-Case Data Structures for the Priority Queue with Attrition. Inf. Process. Lett. 31(2): 69-75 (1989)
8. G. N. Frederickson, S. H. Rodger. A New Approach to the Dynamic Maintenance of Maximal Points in a Plane. Discrete & Comp. Geom. 5: 365-374 (1990)
9. R. Janardan. On the Dynamic Maintenance of Maximal Points in the Plane. Inf. Process. Lett. 40(2): 59-64 (1991)
10. M. L. Fredman, D. E. Willard. Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. J. Comput. Syst. Sci. 48(3): 533-551 (1994)
11. G. S. Brodal, S. Chaudhuri, J. Radhakrishnan. The Randomized Complexity of Maintaining the Minimum. Nord. J. Comput. 3(4): 337-351 (1996)
12. G. S. Brodal. Partially Persistent Data Structures of Bounded Degree with Constant Update Time. Nord. J. Comput. 3(3): 238-255 (1996)
13. F. d'Amore, P. G. Franciosa, R. Giaccio, M. Talamo. Maintaining Maxima under Boundary Updates. Proc. 3rd Italian Conference on Algorithms and Complexity, LNCS 1203, 100-109 (1997), Springer
14. S. Alstrup, T. Husfeldt, T. Rauhe. Marked Ancestor Problems. Proc. 39th Foundations of Computer Science, 534-544 (1998), IEEE Press
15. S. Kapoor. Dynamic Maintenance of Maxima of 2-d Point Sets. SIAM J. Comput. 29(6): 1858-1877 (2000)
16. L. Arge, J. S. Vitter. Optimal External Memory Interval Management. SIAM J. Comput. 32(6): 1488-1508 (2003)