An In-Place Min-Max Priority Search Tree[*]

Minati De[†] Anil Maheshwari[‡] Subhas C. Nandy[†] Michiel Smid[‡]

Abstract

One of the classic data structures for storing point sets in $\mathbb{R}^2$ is the priority search tree, introduced by McCreight in 1985. We show that this data structure can be made in-place, i.e., it can be stored in an array such that each entry stores only one point of the point set and no entry is stored in more than one location of that array. It combines a binary search tree with a heap. We show that all the standard query operations can be answered within the same time bounds as for the original priority search tree, while using only $O(1)$ extra space.

We introduce the min-max priority search tree which is a combination of a binary search tree and a min-max heap. We show that all the standard queries which can be done in two separate versions of a priority search tree can be done with a single min-max priority search tree.

As an application, we present an in-place algorithm to enumerate all maximal empty axis-parallel rectangles amongst points in a rectangular region R in $\mathbb{R}^2$ in $O(m \log n)$ time with $O(1)$ extra-space, where m is the total number of maximal empty rectangles.

1 Introduction

Let P be a set of n points in $\mathbb{R}^2$. A *priority search tree*, as introduced by McCreight [14], is a binary tree T with exactly one node for each point of P and that has the following two properties:

- For each non-root node u, the point stored at u has a smaller y-coordinate than the y-coordinate stored at the parent of u.
- For each internal node u, the x-coordinate of every point in the left subtree of u is less than the x-coordinate of any point in the right subtree of u.

The first property implies that T is a max-heap on the y-coordinates of the points in P. The second property implies that T is a binary search tree on the x-coordinates of the points in P, except that there is no relation between the x-coordinate of the point stored at u and that of any of its children. We use MaxPST to denote this priority search tree with the max-heap property on the y-coordinates. Similarly, MinPST denotes the priority search tree with the min-heap property on the y-coordinates.

In order to use T as a binary search tree on the x-coordinates, McCreight stored at each internal node u one additional point p_u of P, called *splitting point*, which is the point in the left subtree of u whose x-coordinate is maximum. Thus, both MaxPST and MinPST use $O(n)$ space in addition

[*]Research supported by NSERC and the Commonwealth Scholarship Program of DFAIT. A preliminary version of this paper appears in the proceedings of the 23rd Canadian Conference on Computational Geometry, 2011

[†]Indian Statistical Institute, Kolkata, India. Part of this work was done while M.D. was visiting Carleton University, Ottawa, Canada.

[‡]School of Computer Science, Carleton University, Ottawa, Canada.

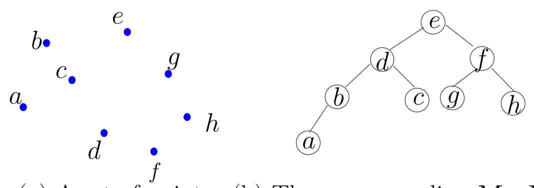
to the space required for storing the coordinates of the members in P. Since both MAXPST and MINPST are balanced trees, the following range queries can be answered efficiently:

1. LEFTMOSTNE(x_0, y_0): report the leftmost point of P in the north-east quadrant $[x_0, \infty) \times [y_0, \infty)$ of the query point (x_0, y_0).
2. RIGHTMOSTNW(x_0, y_0): report the rightmost point of P in the north-west quadrant $(-\infty, x_0] \times [y_0, \infty)$ of the query point (x_0, y_0).
3. HIGHESTNE(x_0, y_0): report the highest point of P in the north-east quadrant $[x_0, \infty) \times [y_0, \infty)$ of the query point (x_0, y_0).
4. HIGHESTNW(x_0, y_0): report the highest point of P in the north-west quadrant $(-\infty, x_0] \times [y_0, \infty)$ of the query point (x_0, y_0).
5. HIGHEST3SIDEDUP(x_0, x_1, y_0): report the highest point of P in the 3-sided query range $[x_0, x_1] \times [y_0, \infty)$.
6. ENUMERATE3SIDEDUP(x_0, x_1, y_0): report all points of P in the 3-sided query range $[x_0, x_1] \times [y_0, \infty)$.
7. LEFTMOSTSE(x_0, y_0): report the leftmost point of P in the south-east quadrant $[x_0, \infty) \times (-\infty, y_0]$ of the query point (x_0, y_0).
8. RIGHTMOSTSW(x_0, y_0): report the rightmost point of P in the south-west quadrant $(-\infty, x_0] \times (-\infty, y_0]$ of the query point (x_0, y_0).
9. LOWESTSE(x_0, y_0): report the lowest point of P in the south-east quadrant $[x_0, \infty) \times (-\infty, y_0]$ of the query point (x_0, y_0).
10. LOWESTSW(x_0, y_0): report the lowest point of P in the south-west quadrant $(-\infty, x_0] \times (-\infty, y_0]$ of the query point (x_0, y_0).
11. LOWEST3SIDEDDOWN(x_0, x_1, y_1); report the lowest point of P in the 3-sided query range $[x_0, x_1] \times (-\infty, y_1]$.
12. ENUMERATE3SIDEDDOWN(x_0, x_1, y_1): report all points of P in the 3-sided query range $[x_0, x_1] \times (-\infty, y_1]$.

Using MAXPST, the queries 1-5 can be answered in $O(\log n)$ time, whereas the query 6 takes $O(\log n + m)$ time, where m is the number of points of P that are in the query range. Similarly, using MINPST, the queries 7-11 can be answered in $O(\log n)$ time, whereas the query 12 takes $O(\log n + m)$ time.

We define the Min-Max Priority Search Tree (MINMAXPST) for a set P of n points in $\mathbb{R}^2$. It is a binary tree T with the following properties:

- For each internal node u, all points in the left subtree of u have an x-coordinate which is less than the x-coordinate of any point in the right subtree of u.
- The y-coordinate values of nodes on even (resp. odd) levels are smaller (resp. greater) than the y-coordinate values of their descendants (if any), where the root is at level zero.



(a) A set of points (b) The corresponding MinMaxPST

e	d	f	b	c	g	h	a

(b) The array containing the MinMaxPST

Figure 1: Example of a MinMaxPST.

The first property implies that T is a binary search tree on the x-coordinates of the points in P, except that there is no relation between the x-coordinates of the points stored at u and any of its children. The second property implies that T is a min-max heap [5] on the y-coordinates of the points in P.

We show that using the single structure MinMaxPST, we can answer all the above mentioned queries with the same time complexities.

In this paper, we show that these results can also be obtained without storing the *splitting point* p_u at each internal nodes u of the tree. Thus, any node of the tree stores exactly one point of P and, as a result, we obtain an *in-place* implementation of the priority search tree. We take for T a binary tree of height $h = \lfloor \log n \rfloor$, such that the levels $0, 1, \ldots, h-1$ are full and level h consists of $n - (2^h - 1)$ nodes which are aligned as far as possible to the left. This allows us to store the tree, like in a standard heap, in an array $P[1, \ldots, n]$; the root is stored at $P[1]$, its left and right children in $P[2]$ and $P[3]$, etc. See Figure 1, where a set of points and the corresponding MinMaxPST is shown.

In Sections 2 and 3, we will present algorithms for constructing the in-place MinMaxPST and answering the above queries. Each of these algorithms use, besides the array $P[1, \ldots, n]$, only $O(1)$ extra space, in the sense that only a constant number of variables are used as the working space, and each one being an integer of $O(\log n)$ bits. The main result of this paper is the following:

Theorem 1.1. *Let P be a set of n points in $\mathbb{R}^2$.*

1. *The in-place* MinMaxPST *can be constructed in $O(n \log n)$ time using $O(1)$ extra space.*

2. *Each of the queries,* HighestNE*,* HighestNW*,* LowestSE*,* LowestSW*,* LeftMostNE*,* RightMostNW*,* LeftMostSE*,* RightMostSW*,* Highest3SidedUp *and* Lowest3SidedDown *can be answered in $O(\log n)$ time using $O(1)$ extra space.*

3. *The queries* Enumerate3SidedUp *and* Enumerate3SidedDown *can be answered in $O(\log n + m)$ time using $O(1)$ extra space, where m is the number of points of P that are in the query range.*

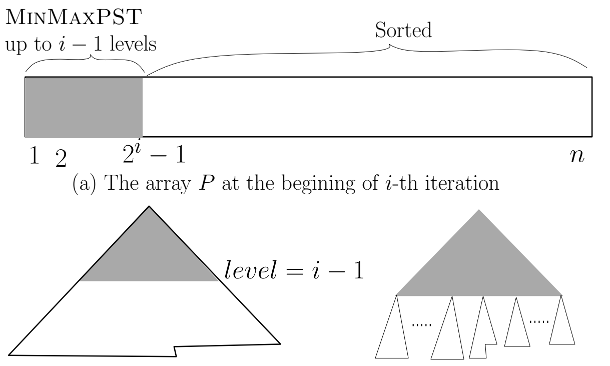
In Section 4, as an application of the in-place priority search tree, we show how one can enumerate all the maximal empty axis-parallel rectangles among a set of n points inside a rectangular region R in $\mathbb{R}^2$ in $O(m \log n)$ time, where m is the number of all maximal empty axis-parallel rectangles. Here we need to build an in-place MaxPST data structure on the set P of points and use the procedures LeftMostNE, RightMostNW and Highest3SidedUp to achieve the desired time complexity using $O(1)$ extra space. MaxPST can be constructed in an exactly similar manner as MinMaxPST.

For ease of presentation, we assume that no two points in the set P have the same x and y-coordinates. If they have the same x-coordinate, then the ties are broken by the y coordinates. The x- and y-coordinates of a point p in $\mathbb{R}^2$ will be denoted by $x(p)$ and $y(p)$, respectively. Most of the algorithms presented in this paper have been implemented and the code is available at [18].

1.1 Related work

In computational geometry, designing in-place algorithms have been studied only for few problems. For the convex hull problem in both 2D and 3D, space efficient algorithms are available. In 2D, the best known result is an $O(n \log h)$–time algorithm with $O(1)$ extra space by Brönnimann et al. [9]. Brönnimann et al. [8] also showed that the upper hull of a set of n points in 3D can be computed in $O(n \log^3 n)$ time using $O(1)$ extra space. This leads to the fact that the Voronoi diagram of a set of n points in 2D can be constructed in $O(n \log^3 n)$ time and $O(1)$ extra space. In the same paper it is also shown that for a parameter s satisfying $c \log^2 n \leq s \leq n$, (for some constant $c > 0$), if $O(\frac{n}{s})$ space is allowed, then the convex hull for a set of n points in 3D can be computed in $O(ns)$ time. Vahrenhold [19] proposed an $O(n^{\frac{3}{2}} \log n)$ time and $O(1)$ extra space algorithm for Klee's measure problem, where the objective is to compute the union of n axis-parallel rectangles of arbitrary sizes. Bose et al. [7] used an in-place divide and conquer technique to solve the following three problems in 2D using $O(1)$ extra space: (i) a deterministic algorithm for the closest pair problem in $O(n \log n)$ time, (ii) a randomized algorithm for the bichromatic closest pair problem in $O(n \log n)$ expected time, and (iii) a deterministic $O(n \log n + k)$ time algorithm for the orthogonal line segment intersection computation problem. For the general line segments intersection problem, two algorithms are available by Chen and Chan [12]. If the input array can be used for storing intermediate results, then the problem can be solved in $O((n+k) \log n)$ time and $O(1)$ extra space. If the input array is not allowed to be destroyed, then the time complexity increases by a factor of $\log n$, and it also uses $O(\log^2 n)$ extra space. Later Vahrenhold [20] improved the time complexity of this problem to $O(n \log^2 n + k)$. Finally, Chan and Chen [10] proposed a randomized algorithm for the same problem whose expected running time is $O(n \log n + k)$. In the same paper, they also proposed a randomized algorithm for computing the convex hull of a set of points in 3D that runs in $O(n \log n)$ time.

Asano et al. presented the following results in [3]: if a set of n points in 2D is given in a read-only array, then its Delaunay triangulation and Euclidean minimum spanning tree can be computed in $O(n^2)$ and $O(n^3)$ time, respectively. In the same paper, they showed that the shortest path between a pair of points s and t in a simple polygon can be computed in $O(n^2)$ time if the vertices of the polygon are given in a read-only array in counter-clockwise order. All these algorithms take $O(1)$ work-space in addition to the input array. This, in turn, recognizes the largest empty circle among a set of points with the same time complexity. Recently, Asano et al. [2] showed that given a planar straight-line graph with n vertices in a read-only array, it can be triangulated in $O(n^2)$ time and $O(1)$ extra space. This technique is applied to solve the shortest path query problem in a simple polygon. Here an $O(n^2)$ time pre-processing step is executed to create a data structure in k memory locations ($k << n$), and a shortest path query between a pair of points s and t inside the polygon can be answered in $O(\frac{n^2}{k})$ time.



(a) The array P at the begining of i-th iteration

(b) The tree representation of the array P

(c) the split of the unshaded subtree into 2^i blocks

Figure 2: Demonstration of the invariants of MinMaxPST

2 Constructing the in-place MinMaxPST

Let P be a set of n points in the plane. In this section, we present an algorithm for constructing the in-place MinMaxPST.

Let $h = \lfloor \log n \rfloor$ be the height of the MinMaxPST. Our algorithm constructs the tree level by level. After constructing the $(i-1)$-th level of the tree, it maintains the following invariant:

- The subarray $P[1, \dots, 2^i - 1]$ stores levels $0, 1, \dots, i-1$ of the tree, and the points in the subarray $P[2^i, \dots, n]$ are stored in sorted order by their x-coordinates (see Figure 2(a)).

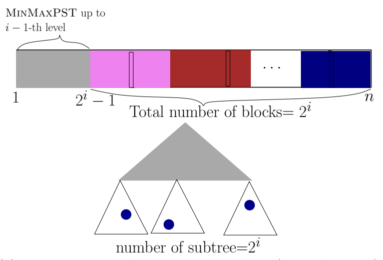
The algorithm starts by sorting the array $P[1, \dots, n]$ by their x-coordinates. After this sorting step, the invariant holds with $i = 0$.

Let i be an index with $0 \leq i < h$, and consider the i-th iteration of the algorithm. Let $A = n - (2^h - 1)$ be the number of nodes at level h of the tree, and let $k = \lfloor A/2^{h-i} \rfloor$. Level i consists of 2^i nodes (see Figure 2(b)). If $k = 2^i$, then each of these nodes is the root of a subtree of size $2^{h+1-i} - 1$. Otherwise, we have $k < 2^i$, in which case level i consists of, from left to right,

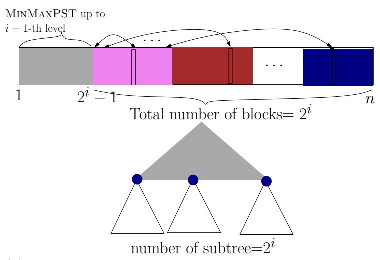
1. k nodes, which are roots of subtrees, each of size $K_1 = 2^{h+1-i} - 1$,
2. one node, which is the root of a subtree of size $K_2 = 2^{h-i} - 1 + A - k2^{h-i}$,
3. $2^i - 1 - k$ nodes, which are roots of subtrees, each of size $K_3 = 2^{h-i} - 1$.

See Figure 2(c) for an illustration. We divide the subarray $P[2^i, \dots, n]$ into 2^i *blocks*: If $k = 2^i$, then there are k blocks, each of size $2^{h+1-i} - 1$. Otherwise, there are, from left to right, (i) k blocks of size K_1, (ii) one block of size K_2, and (iii) $2^i - 1 - k$ blocks of size K_3.

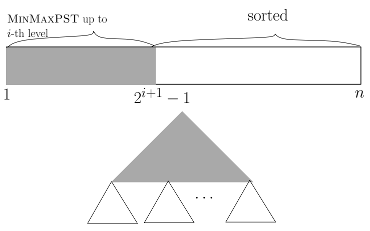
The algorithm scans the subarray $P[2^i, \dots, n]$ and in each of the 2^i blocks, finds the highest or lowest point depending on the parity of *level* (Figure 3(a)). These highest or lowest points are swapped with the subarray $P[2^i, \dots, 2^{i+1} - 1]$ (Figure 3(b)); note that these are the nodes of the i-th level of the MinMaxPST. At this moment, the invariant has not been restored yet, because the elements in the subarray $P[2^{i+1}, \dots, n]$ may not be sorted by their x-coordinates. Therefore, the algorithm runs the heapsort algorithm on this subarray. The final output after i-th iteration is shown in Figure 3(c).



(a) In each block of the colored portions (non-PST part) of the array, the element having maximum (or minimum) y-coordinate are marked. The corresponding elements in the tree are marked by blue dots.



(b) The marked elements are moved in the appropriate position to get the MinMaxPST up to the i-th level



(c) Shaded portion corresponds to MinMaxPST up to i-th level. The unshaded portion is sorted with respect to their x-coordinates

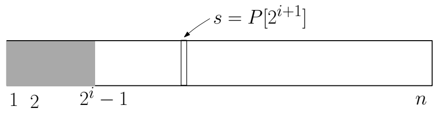
Figure 3: Construction of i-th level of MinMaxPST

The complete algorithm for constructing the in-place min-max priority search tree is given in Algorithm 1. It uses algorithm HeapSort(m, n), which runs the heapsort algorithm on the subarray $P[m, \ldots, n]$.

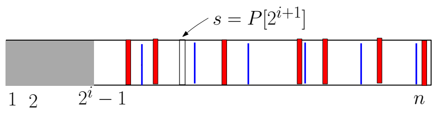
The correctness of this algorithm follows by observing that the invariant is correctly maintained. The initial sorting in line 3 takes $O(n \log n)$ time using $O(1)$ extra space. Each of the $h = \lfloor \log n \rfloor$ iterations of the outer-most for-loop takes $O(n \log n)$ time and $O(1)$ extra space. We can use one extra variable to maintain the value 2^i, so that it does not have to be recomputed during the execution of the outer-most for-loop. Thus, the entire algorithm ConstructMinMaxPST takes $O(n \log^2 n)$ time and uses $O(1)$ extra space.

Algorithm 1: ConstructMinMaxPST

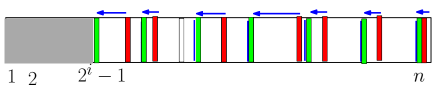
```
   Input: An array $P[1, \ldots, n]$ of points in $\mathbb{R}^2$.
   Output: The min-max priority search tree of those points stored in $P$.
1  $h = \lfloor \log n \rfloor$; $A = n - (2^h - 1)$;
2  $level = 0$;
3  HeapSort$(1, n)$;
4  for $i = 0$ to $h - 1$ do
5      $k = \lfloor A/2^{h-i} \rfloor$;
6      $K_1 = 2^{h+1-i} - 1$;
7      $K_2 = 2^{h-i} - 1 + A - k2^{h-i}$;
8      $K_3 = 2^{h-i} - 1$;
9      for $j = 1$ to $k$ do
10         $\ell$ = index in $\{2^i + (j-1)K_1, \ldots, 2^i + jK_1 - 1\}$ such that $y(P[\ell])$ is maximum or minimum
           depending on the parity of $level$;
11         swap $P[\ell]$ and $P[2^i + j - 1]$ ;
12     if $k < 2^i$ then
13         $\ell$ = index in $\{2^i + kK_1, \ldots, 2^i + kK_1 + K_2 - 1\}$ such that $y(P[\ell])$ is maximum or minimum
           depending on the parity of $level$;
14         swap $P[\ell]$ and $P[2^i + k]$;
15         $m = 2^i + kK_1 + K_2$;
16         for $j = 1$ to $2^i - k - 1$ do
17             $\ell$ = index in $\{m + (j-1)K_3, \ldots, m + jK_3 - 1\}$ such that $y(P[\ell])$ is maximum or
               minimum depending on the parity of $level$;
18             swap $P[\ell]$ and $P[2^i + k + j]$;
19     HeapSort$(2^{i+1}, n)$;
20     $level = level + 1$;
```



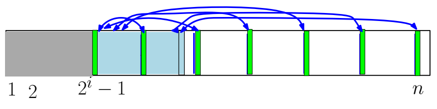
(a) Gray portion of the array corresponds to the MinMaxPST constructed so far and white portion is sorted according to the x-coordinate



(b) Each red colored element is the point with the maximum (or minimum) y-coordinate value among elements in their respective block.



(c) Each green element corresponds to the first element in its block. Each red colored element is moved to the green position of its block by successive swapping



(d) Green elements are the points with maximum (or minimum) y-value in their respective blocks. These are placed in the light-blue positions by swapping

Figure 4: Improving the construction time of MinMaxPST - the i-th iteration

2.1 Improving the construction time

We now show that the in-place MinMaxPST can be constructed in $O(n \log n)$ time. Consider again the i-th iteration of Algorithm 1. At the beginning of this iteration, (i.e., immediately after line 4), assign $s = P[2^{i+1}]$. In Figure 4(a) this is demonstrated, where the gray portion of the array corresponds to the MinMaxPST constructed up to the $(i-1)$-th level, and white portion is sorted according to the x-coordinate. As we have seen, immediately after line 18, level i of the tree has been constructed, but the elements in $P[2^{i+1}, \ldots, n]$ may not be sorted by their x-coordinates. The reason is that the *newly swapped* elements in the subarray $P[2^{i+1}, \ldots, n]$ have their x-coordinates smaller than $x(s)$. Note that the x-coordinates of all other elements in $P[2^{i+1}, \ldots, n]$ are greater than $x(s)$.

Assume that we have sorted the *newly swapped* elements by their x-coordinates. For any j with $2^{i+1} \leq j \leq n$, define $f(j) = 0$ if $x(P[j]) < x(s)$, and $f(j) = 1$ if $x(P[j]) \geq x(s)$. Then sorting the subarray $P[2^{i+1} \ldots n]$ by x-coordinates is equivalent to a stable sorting[1] of the bit-sequence $f(j)$, $2^{i+1} \leq j \leq n$. Katajainen and Pasanen [13] have shown that this is possible in $O(n)$ time using $O(1)$ extra space.

Thus, we modify lines 9–18 of the Algorithm 1 to sort the elements in $P[2^{i+1}, \ldots, n]$ as follows:

Step 1: In each of the 2^i blocks, find the highest (or lowest) point (see red colored elements in Figure 4(b)) and move it to the first position of its block by successive swapping. Note that all other points in this block remain sorted by their x-coordinates (see Figure 4(c)).

Step 2: For $j = 1, 2, \ldots, 2^i$, swap $P[2^i + j - 1]$ and the point at the first position of the j-th block (see Figure 4(d)).

Step 3: Run the heapsort algorithm on those elements that are the first elements in their block in the subarray $P[2^{i+1}, \ldots, n]$.

Step 4: Run Katajainen and Pasanen's algorithm on the subarray $P[2^{i+1}, \ldots, n]$.

Lemma 2.1. *Given a set of points in $\mathbb{R}^2$ in an array P, the in-place construction of* MinMaxPST *in the same array P uses $O(n \log n)$ time and $O(1)$ extra work-space.*

[1]Given a set of unsorted records having key values 0 and 1, where each record consists of some other information, a sorting of these records is called stable if it keeps the information of all the records with equal keys in the same relative order in the output as they were in the input.

Proof. Steps 1, 2, and 4 take $O(n)$ time. In Step 3, the heapsort algorithm is run on a sequence of size at most 2^i, where the heap is stored in the first positions of those blocks that are involved in the sorting step (i.e., in the subarray $P[2^{i+1}, \dots, n]$). Thus, if the algorithm wants to access the m-th element in the heap, it computes, in $O(1)$ time, the first position of the m-th block in the subarray $P[2^{i+1}, \dots, n]$. It follows that Step 3 takes $O(i \cdot 2^i)$ time.

The running time of the i-th iteration is

$$O\left(n + i \cdot 2^i\right) = O(n + 2^i \log n)$$

and, therefore, the total time of the algorithm is

$$O\left(\sum_{i=0}^{h-1}(n + 2^i \log n)\right) = O(n \log n).$$

The algorithm uses $O(1)$ extra space. □

Remark 1. *The* MaxPST *and* MinPST *data structures can be constructed in a similar manner with the same time and space complexities, where we compute either the highest or the lowest point at all levels of the tree instead of alternating.*

3 Queries on the in-place MinMaxPST

In this section, we present three query algorithms mentioned in Section 1. For ease of presentation, we describe the algorithms using the terminology of trees. We denote by T the MinMaxPST that is implicitly stored in the array $P[1, \dots, n]$, obtained by running the algorithm ConstructMin-MaxPST. Recall that the root of T, denoted by $root(T)$, is stored at $P[1]$. Consider a node whose index in P is i. If $2i \le n$, then the left child of this node is stored at $P[2i]$. If $2i + 1 \le n$, then the right child of this node is stored at $P[2i + 1]$. This node is a leaf if and only if $2i > n$. For any element $p \in P$, we denote by T_p the subtree rooted at p. Furthermore, the left and right children of p (if they exist) are denoted by p_l and p_r, respectively.

3.1 LeftMostNE(x_0, y_0)

For two given real numbers x_0 and y_0, let $Q = [x_0, \infty) \times [y_0, \infty)$ be the north-east quadrant of the point (x_0, y_0), and P_Q be the set of points of P that are in the region Q. If $P_Q \cap P \neq \emptyset$, define p^* to be the leftmost point of P_Q. If $P_Q \cap P = \emptyset$, define p^* to be the point (∞, ∞). Algorithm LeftMostNE(x_0, y_0) returns the point p^*.

As in the standard priority search tree [14], the search starts at the root of T. Since the x-coordinate of the partitioning line is not available in each node, two index variables p and q are required for navigation along the search path. Another variable *best* is used to store the leftmost point in Q obtained so far. At the end of the execution *best* will contain the final result. At any instant of time, the variables *best*, p, and q satisfy the following invariant:

- If $P_Q \neq \emptyset$, then $p^* \in \{best\} \cup T_p \cup T_q$.
- If $P_Q = \emptyset$, then $p^* = best$.
- p and q are at the same level of T and $x(p) \le x(q)$.

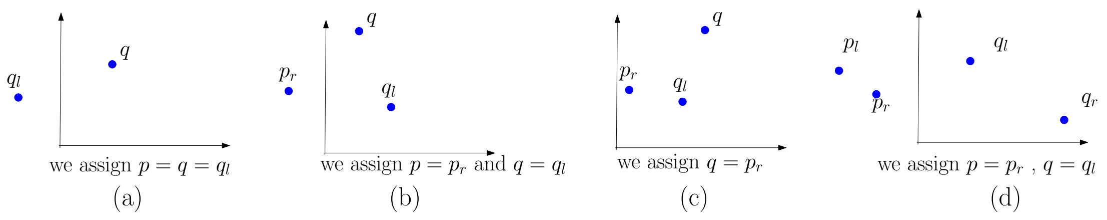


Figure 5: Different cases for the procedure LeftMostNE(x_0, y_0) and the corresponding assignments of p and q

The algorithm starts by initializing *best* $= (\infty, \infty)$ and $p = q = root(T)$. The algorithm uses a procedure UpdateLeftMost(t), which takes an input point $t \in P$, and updates *best* as follows: *if* $t \in Q$ *and* $x(t) < x(best)$ *then assign* $best = t$.

At each iteration, UpdateLeftMost(p) and UpdateLeftMost(q) are executed to update the variable *best*; p and q move down the tree according to the relative positions of their children and the query region Q as stated below. As soon as p reaches the leaf level, the algorithm terminates.

If $\mathbf{p} = \mathbf{q}$, then irrespective of whether or not $p \in Q$, we proceed along both left and right children of p; thus, we assign $p = p_l$ and $q = p_r$. If p has only one child then we assign $p = q = p_l$.

If $\mathbf{p} \neq \mathbf{q}$, then we need to consider the following situations:

q is a leaf: Here p may or may not be a leaf. If p is not a leaf, then the search proceeds to the children of p.

q is not a leaf: Here p is also not a leaf, and p has two children.

Now, if **q has only one child** then according to our inplace structure of the MinMaxPST, p, q lies in one level above the leaf level of T. Here three situations need to be considered: (i) $x(q_l) < x_0$, (ii) $x(p_r) < x_0$ but $x(q_l) > x_0$, and (iii) $x(p_r) > x_0$. The assignments of p and q for executing the next iteration in these three situations are shown in Figures 5(a),5(b) and 5(c) respectively. Note that, here we need not have to consider the y-coordinate of p_r or q_l since they are in the last level, and if they are not inside Q then *best* will not be updated by the procedure UpdateLeftMost.

If **q has two children**, we need to consider the y-coordinate of the children of both p and q to update the values of p and q for the next iteration.

Note that, here x_0 (defining the vertical line $L : x = x_0$ of the region Q) may lie in one of the five intervals defined by $I_0 = [-\infty, x(p_l)]$, $I_1 = [x(p_l), x(p_r)]$, $I_2 = [x(p_r), x(q_l)]$, $I_3 = [x(q_l), x(q_r)]$, $I_4 = [x(q_r), \infty]$.

- If $\mathbf{x_0} \in I_0$, then the x-coordinates of all the members in the set $V = \{p_l, p_r, q_l, q_r\}$ are greater than x_0. In a MaxPST, we consider the members in V in increasing order of their x-coordinates. For each $\theta \in V$, we test whether $\theta \in Q$ (i.e., $y(\theta) > y_0$). If so, we set $p = q = \theta$ and do not consider the other members of V; otherwise, we consider the next member of V. Since we are working with a MinMaxPST, while considering $\theta \in \{p_l, p_r, q_l, q_r\}$ in this order, if either $\theta \in Q$ or $\theta \notin Q$ but any one of its children lies in Q, then we set $p = q = \theta$ and do not check the other members in the list. If no such θ is found then the algorithms stops[2].

[2]In the pseudocode, we assigned $p = q = p_l$ (the right end-point of I_0). This will not cause any problem; here the algorithm continues up to the leaf level but *best* is not updated by the procedure UpdateLeftMost any more.

- If $\mathbf{x}_0 \in I_i$ for some $i = 1, 2, 3$, then we set p to the left end point of I_i. We construct the ordered set V with the right end point of I_i and all the end-points of I_j, $i < j \leq 4$, and update q with the leftmost element $\theta \in V$ as we did earlier (see Figure 5(d)). If no such θ is found then we set q to the right-end of I_i.
- If $\mathbf{x}_0 \in I_4$, then we assign $p = q = q_r$ since if there is any further update of *best* it will be any of the successors of q_r. Note that here we do not have to consult the y-coordinate of q_r since if it or its successors are not inside Q then *best* will not be updated by the procedure UPDATELEFTMOST.

The detailed steps of the algorithm are given in Algorithm 2. The following lemma states the correctness and the complexity result of this query algorithm.

Lemma 3.1. *Given an in-place* MINMAXPST *with a set of points in $\mathbb{R}^2$ in an array P, the proposed* LEFTMOSTNE(x_0, y_0) *query algorithm correctly reports the resulting point in $O(\log n)$ time with $O(1)$ extra work-space.*

Proof. By a careful case analysis it can be observed that the invariant is correctly maintained. This implies the correctness of the algorithm. In each iteration of the while-loop, p and q move down the tree, except in line 12. In the latter case, however, p will become a leaf in the next iteration. As a result, the while-loop makes $O(\log n)$ iterations. Since each iteration takes $O(1)$ time, the total time for algorithm LEFTMOSTNE is $O(\log n)$. It follows from the algorithm that it uses $O(1)$ extra space. □

3.2 HIGHEST3SIDEDUP(x_0, x_1, y_0)

Given three real numbers x_0, x_1, and y_0, the three-sided query range is defined by $Q = [x_0, x_1] \times [y_0, \infty)$. As in the earlier section, here also we will use P_Q to denote the subset of points in P that lie inside Q. If $P_Q \neq \emptyset$ then the highest point of P in Q is p^* satisfying $y(p^*) = \max\{y(p) | p \in P_Q\}$. If $P_Q = \emptyset$, then define $p^* = (\infty, -\infty)$. Algorithm HIGHEST3SIDEDUP(x_0, x_1, y_0), described in Algorithm 3, returns the point p^*.

As before, *best* will store the highest point in Q found so far. Since Q is bounded by two vertical sides, we use four index variables p, p', q and q' to guide the search path. In addition, we use four bits L, L', R and R'; these correspond to the subtrees of T rooted at the nodes p, p', q, and q', respectively; if a bit is equal to one, then the corresponding node is referred to as an *active node* (for example, if $L = 1$, then p is an active node), and the subtree rooted at that node may contain a candidate point for p^*. So the search is required to be performed in the subtree rooted at all active nodes. More formally, at any instant of time the variables satisfy the following invariant:

- If $L = 1$ then $x(p) < x_0$.
- If $L' = 1$ then $x_0 \leq x(p') \leq x_1$.
- If $R = 1$ then $x(q) > x_1$.
- If $R' = 1$ then $x_0 \leq x(q') \leq x_1$.
- If $L' = 1$ and $R' = 1$ then $x(p') \leq x(q')$.
- If $P_Q \neq \emptyset$, then either $p^* = best$ or p^* is in the subtree rooted at any one of the active nodes. In other words, $p^* \in \{best\} \cup \left(\cup_{z \in Active} T_{\tau(z)}\right)$, where

$$Active = \{z | z \in \{L, L', R, R'\} \text{ and } z = 1\}$$

Algorithm 2: LeftMostNE(x_0, y_0)

```
Input: Real numbers $x_0$ and $y_0$ defining the north-east quadrant $Q$.
Output: The leftmost point $p^*$ in $Q \cap P$, if it exists; otherwise the point $(\infty, \infty)$.
best = $(\infty, \infty)$; $p = root(T)$; $q = root(T)$;
while $p$ is not a leaf do
    UpdateLeftMost($p$); UpdateLeftMost($q$);
    if $p = q$ then
        if $p$ has one child then
            $q = p_l$; $p = p_l$;
        else
            $q = p_r$; $p = p_l$;
    else
        // $p \neq q$
        if $q$ is leaf then
            $q = p$ ;
        else if $q$ has one child then
            if $x(q_l) < x_0$ then
                $p = q_l$; $q = q_l$;
            else if $x(p_r) < x_0$ then
                $p = p_r$; $q = q_l$;
            else
                $q = p_r$; $p = p_l$;
        else
            // $p$ and $q$ both have two children
            $c_1 = p_l$; $c_2 = p_r$; $c_3 = q_l$; $c_4 = q_r$;
            if $x_0 < x(c_1)$ then
                $V = \{c_t | 1 \leq t \leq 4$ and $c_t$ or one of its children is in $Q\}$;
                if $V = \emptyset$ then
                    $q = c_1$;
                else
                    $q$ = leftmost point in $V$;
                $p = q$;
            else if $x_0 \geq x(c_4)$ then
                $p = q = c_4$;
            else
                $i$ is the index such that $x(c_i) \leq x_0 < x(c_{i+1})$;
                $p = c_i$;
                $V = \{c_t | (i+1) \leq t \leq 4$ and $c_t$ or one of its children is in $Q\}$;
                if $V = \emptyset$ then
                    $q = c_{i+1}$;
                else
                    $q$ = leftmost point in $V$;
UpdateLeftMost($p$); UpdateLeftMost($q$);
return best;
```

and $\tau(z)$ is defined as follows:

z	L	L'	R	R'
$\tau(z)$	p	p'	q	q'

In the initialization, we set $best = (\infty, -\infty)$. The other variables are assigned depending on the position of the root of T with respect to the query region Q. More specifically,

- if $x(root(T)) < x_0$ then initialize $L = 1$, $L' = R = R' = 0$ and $p = root(T)$.
- if $x_0 < x(root(T)) \leq x_1$ then initialize $L' = 1$, $L = R = R' = 0$ and $p' = root(T)$.
- if $x(root(T)) > x_1$ then initialize $R = 1$, $L = L' = R' = 0$ and $q = root(T)$.

We use *level*(t) to denote the level of node $t \in T$. The *root*(T) is assumed to have level 0. At each iteration, we choose an active node having minimum level. If more than one such node is available, then we choose one of them arbitrarily. We define a variable $\lambda = min_{z \in \mathcal{A}ctive} level(z)$ and $z^* = \{z | z \in \mathcal{A}ctive, level(z) = \lambda\}$. In other words, z^* is the active node having minimum level. If no *active node* is available then the algorithm stops. Otherwise, we call the procedure CheckLeft, CheckLeftIn, CheckRight and CheckRightIn, depending on whether z^* is equal to p, p', q, or q', respectively. These procedures are used to decide the possible directions of the search paths for the next iteration. We describe the details of the procedures CheckLeft and CheckLeftIn in Sections 3.2.1 and 3.2.2. The other two procedures are symmetric to these.

Algorithm 3: HIGHEST3SIDEDUP(x_0, x_1, y_0)

```
Input: Real numbers $x_0$, $x_1$, and $y_0$ defining the region $Q = [x_0, x_1] \times [y_0, \infty)$.
Output: The highest point $p^*$ in $Q \cap P$, if it exists; otherwise the point $(\infty, -\infty)$.
best = $(\infty, -\infty)$;
if $x(root(T)) < x_0$ then
    $p = root(T)$ ; $L = 1$; $L' = R = R' = 0$
else if $x(root(T)) < x_1$ then
    $p' = root(T)$; $L' = 1$; $L = R = R' = 0$
else
    $q = root(T)$; $R = 1$; $L = L' = R' = 0$
while $L = 1 \vee L' = 1 \vee R = 1 \vee R' = 1$ do
    $\mathcal{A}ctive = \{z \in \{L, L', R, R'\} | z = 1\}$;
    $z$ = element of $\mathcal{A}ctive$ for which $level(\tau(z))$ is minimum;
    if $z = L$ then
        CHECKLEFT$(p)$;
    else if $z = L'$ then
        CHECKLEFTIN$(p')$;
    else if $z = R$ then
        CHECKRIGHT$(q)$;
    else
        CHECKRIGHTIN$(q')$;
return best;
```

Each of these procedures evokes UpdateHighest and ExUpdateHighest. UpdateHighest(t) updates *best* as follows: if $t \in Q$ and $y(t) > y(best)$ then $best = t$ is set. For a node t, ExUpdateHighest(t) is executed only when it is certain that all the elements in the subtree rooted at t are within the two vertical lines at x_0 and x_1. The procedure ExUpdateHighest(t)

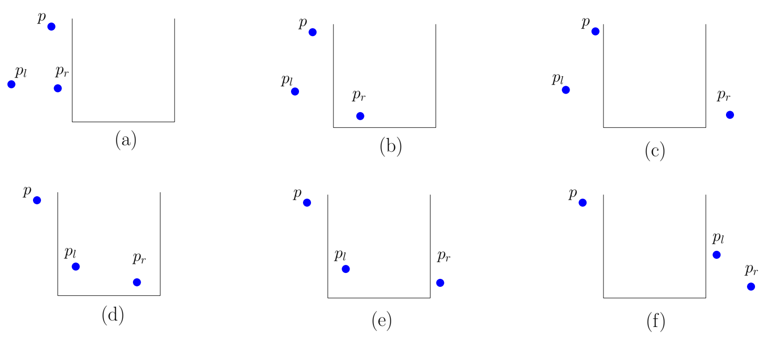


Figure 6: Different cases for the procedure CheckLeft(p).

also updates *best* by checking t and its children (if any one of them exists); in other words, it evokes UpdateHighest(t), UpdateHighest(t_l) and UpdateHighest(t_r).

3.2.1 CheckLeft

This procedure is called with the node p as argument, provided it is active, i.e., $L = 1$ and $level(p)$ is minimum among other active nodes. Depending on the value of the x-coordinates of the two children of p, the following situations may arise:

$x(p_l), x(p_r) < x_0$: See Figure 6(a). Here, we set $p = p_r$.

$x(p_l) < x_0$, $x_0 < x(p_r) < x_1$: See Figure 6(b). Here, we evoke UpdateHighest(p_r), and set $p' = p_r$ and $p = p_l$ copying the existing value of p' in a temporary location *temp*. Now, if $L' = 1$ and $R' = 1$, then we evoke ExUpdateHighest(p'); else if $L' = 1$ and $R' = 0$, then we set $q' = temp$ and $R' = 1$.

$x(p_l) < x_0$, $x(p_r) > x_1$: See Figure 6(c). We set $q = p_r$, $p = p_l$ and $R = 1$. Note that the existing values of L' and R' are both zero.

$x_0 < x(p_l) < x(p_r) < x_1$: See Figure 6(d). Here we set $L = 0$ and $p' = p_l$ copying the existing value of p' in *temp*. Here, depending on the value of L' and R', four situations may happen :

- **$L' = 0, R' = 0$:** We evoke UpdateHighest(p_l) and UpdateHighest(p_r), and set $q' = p_r$ and $L' = R' = 1$.
- **$L' = 0, R' = 1$:** We evoke UpdateHighest(p_l) and ExUpdateHighest(p_r), and set $L' = 1$.
- **$L' = 1, R' = 0$:** We evoke UpdateHighest(p_l) and ExUpdateHighest(p_r), and set $q' = temp$ and $R' = 1$.
- **$L' = 1, R' = 1$:** We evoke UpdateHighest(p_l), ExUpdateHighest(p_r), ExUpdateHighest($temp_r$) and ExUpdateHighest($temp_l$).

$x_0 < x(p_l) < x_1$, $x(p_r) > x_1$: See Figure 6(e). We evoke UpdateHighest(p_l), and set $p' = p_l$, $q = p_r$, $L = 0$, $L' = 1$, and $R = 1$. Note that the existing value of $R' = 0$ will remain unchanged in this situation.

$x(p_l), x(p_r) > x_1$: See Figure 6(f). We set $q = p_l$, $L = 0$ and $R = 1$. Note that the existing values of L' and R' are both zero.

Algorithm 4: CheckLeft(p)

```
Input: A node $p$ such that $x(p) < x_0$.
if $p$ is a leaf then
    $L = 0$
else if $p$ has one child then
    /* the only child is $p_l$ */
    if $x_0 \leq x(p_l) \leq x_1$ then
        UpdateHighest($p_l$);
        if $L' = 1 \wedge R' = 1$ then
            ExUpdateHighest($p'$);
        else if $L' = 1$ then
            $q' = p'$; $R' = 1$
        $p' = p_l$; $L' = 1$; $L = 0$;
    else if $x(p_l) < x_0$ then
        $p = p_l$
    else
        $q = p_l$; $R = 1$; $L = 0$
else
    /* $p$ has two children */
    if $x(p_l) < x_0$ then
        if $x(p_r) < x_0$ then
            $p = p_r$
        else if $x(p_r) \leq x_1$ then
            UpdateHighest($p_r$);
            if $L' = 1 \wedge R' = 1$ then
                ExUpdateHighest($p'$);
            else if $L' = 1$ then
                $q' = p'$; $R' = 1$
            $p' = p_r$; $p = p_l$; $L' = 1$
        else
            $q = p_r$; $p = p_l$; $R = 1$
    else if $x(p_l) \leq x_1$ then
        if $x(p_r) > x_1$ then
            UpdateHighest($p_l$);
            $q = p_r$; $p' = p_l$; $L = 0$; $L' = R = 1$;
        else
            UpdateHighest($p_l$);
            if $R' = 1 \wedge L' = 1$ then
                ExUpdateHighest($p_r$);  ExUpdateHighest($q'$);
            else if $L' = 1$ then
                ExUpdateHighest($p_r$);  $q' = p'$; $R' = 1$
            else if $R' = 1$ then
                ExUpdateHighest($p_r$);  $L' = 1$
            else
                UpdateHighest($p_r$);  $q' = p_r$; $L' = R' = 1$
            $p' = p_l$; $L = 0$
    else
        $q = p_l$; $L = 0$; $R = 1$
```

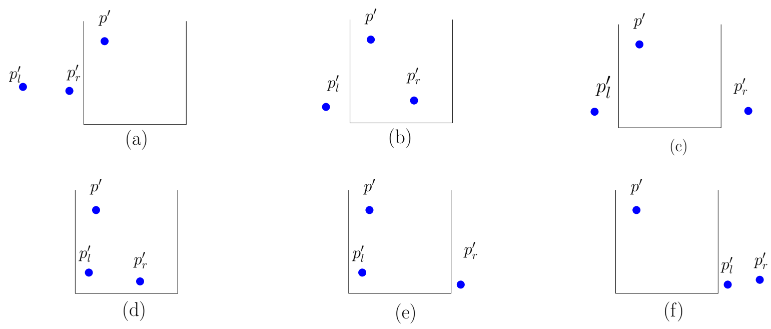


Figure 7: Different cases for the procedure CheckLeftIn(p').

Observe that in each case the variables satisfy all the invariants. The detailed steps of the algorithm are given in Algorithm 4, where the special cases, when p is a leaf or p has only one child, are taken care of appropriately.

3.2.2 CheckLeftIn

This procedure is called with the node p' as argument, provided it is active, i.e., $L' = 1$ and $level(p')$ is minimum among other active nodes. If we work on MaxPST, then in this situation nothing needs to be done. Since we are working on MinMaxPST, we may have $p' \notin Q$ (i.e., $x_0 \leq x(p') \leq x_1$ and $y(p') < y_0$), but the children of p' may lie in Q. This is handled by setting the variables as follows for the next iteration.

$x(p'_l), x(p'_r) < x_0$: We set $p = p'_r$, $L = 1$ and $L' = 0$ (see Figure 7(a)).

$x(p'_l) < x_0$, $x_0 < x(p'_r) < x_1$: We evoke UpdateHighest(p'_r), and set $p = p'_l$, $p' = p'_r$ and $L = 1$ (see Figure 7(b)).

$x(p'_l) < x_0$, $x(p'_r) > x_1$: We set $p = p'_l$, $q = p'_r$, $L = 1$, $L' = 0$ and $R = 1$. Note that the existing value of R' is zero (see Figure 7(c)).

$x_0 < x(p'_l) < x(p'_r) < x_1$: We evoke UpdateHighest(p'_l), and set $p' = p'_l$. If $R' \neq 1$, we evoke UpdateHighest(p'_r), and set $q' = p'_r$, $R' = 1$. If $R' = 1$, then we execute ExUpdateHighest(p'_r) (see Figure 7(d)).

$x_0 < x(p'_l) < x_1$, $x(p'_r) > x_1$: We evoke UpdateHighest(p'_l), and set $q = p'_r$, $p' = p'_l$, and $R = 1$ (see Figure 7(e)).

$x(p'_l), x(p'_r) > x_1$: We set $q = p'_l$, $L' = 0$ and $R = 1$. Note that the existing value of R' is zero (see Figure 7(f)).

We describe the pseudocode for CheckLeftIn(p') in Algorithm 5.

Lemma 3.2. *(i) In each iteration of algorithm* Highest3SidedUp*, the value of λ (i.e., the minimum level among active nodes) increases or remains same. (ii) In each iteration, the difference of levels among the active nodes is at most 1. (iii) There may exist at most four consecutive iterations during which λ remains the same.*

Algorithm 5: CHECKLEFTIN(p')

```
Input: A node $p'$ such that $x_0 \leq x(p') \leq x_1$.
if $p'$ is a leaf then
    $L' = 0$
else if $p'$ has one child then
    /* assume that the only child is $p_l$ */
    if $x_0 \leq x(p'_l) \leq x_1$ then
        UpdateHighest($p'_l$);
        $p' = p'_l$;
    else if $x(p'_l) < x_0$ then
        $p = p'_l$; $L' = 0$; $L = 1$
    else
        $q = p'_l$; $R = 1$; $L' = 0$
else
    // $p'$ has two children
    if $x(p'_l) < x_0$ then
        if $x(p'_r) < x_0$ then
            $p = p'_r$; $L = 1$; $L' = 0$
        else if $x(p'_r) \leq x_1$ then
            UpdateHighest($p'_r$);
            $p = p'_l$; $p' = p'_r$; $L = 1$;
        else
            $q = p'_r$; $p = p'_l$; $R = 1$; $L = 1$; $L' = 0$
    else if $x(p'_l) \leq x_1$ then
        if $x(p'_r) > x_1$ then
            UpdateHighest($p'_l$);
            $q = p'_r$; $p' = p'_l$; $R = 1$;
        else
            UpdateHighest($p'_l$);
            if $R' = 1$ then
                ExUpdateHighest($p'_r$);
                $p' = p'_l$
            else
                UpdateHighest($p'_r$);
                $q' = p'_r$; $p' = p'_l$; $R' = 1$
    else
        $q = p'_l$; $L' = 0$; $R = 1$
```

Proof. (i) In each iteration we choose an active node $z^* \in \{p, p', q, q'\}$ having minimum level λ. After the execution of the corresponding procedure, either z does not remain active or it is assigned to a child of z (in the next level). Now, the minimum level either increases or remains the same depending on whether there exists one or more than one node having level λ.

(ii) [By induction] Assume that the result holds at the beginning of an iteration. Let $z^* \in \mathcal{A}ctive$ be the node chosen with minimum level, say λ. Let $active^* = \mathcal{A}ctive \setminus \{z^*\}$. If z^* remains active after the iteration, its level increases; otherwise it does not remain active. If $active^* = \emptyset$ then the difference becomes 0; otherwise, the difference becomes 1.

(iii) We show that in each iteration either the minimum level λ is increased by 1 or the number of active nodes having minimum level is decreased by 1.

Let $z^* \in \mathcal{A}ctive$ be chosen with minimum level λ in an iteration. If z^* remains active after the iteration, its level has been increased by 1. So, we need to consider the case where z^* does not remain active. In such a situation,

- if z^* corresponds to a leaf, it no longer remains active.
- otherwise, some other variable $z' \in \{p, p', q, q'\} \setminus \{z^*\}$ is assigned to a child of z^*. Note that, if $level(z') = \lambda + 1$ prior to the iteration, then its level is not increased. But the number of members in the set $\mathcal{A}ctive$ having minimum level λ is reduced by 1.

Thus, if there is an instance having four members in the set $\mathcal{A}ctive$ each having level λ, at most four consecutive iterations are executed by choosing a node from $\mathcal{A}ctive$ with level λ. $\square$

Lemma 3.3. *Given an in-place* MinMaxPST *with a set of points in $\mathbb{R}^2$ in an array P, algorithm* Highest3SidedUp(x_0, x_1, y_0) *correctly reports the point having maximum y-coordinate in the query region , in $O(\log n)$ time with $O(1)$ extra work-space.*

Proof. The space complexity trivially follows since we have used $O(1)$ working variables (4 index variables, 4 bit variables and 4 λ-values). The time complexity follows from Lemma 3.2 and the fact that the height of MinMaxPST is $O(\log n)$. $\square$

3.3 Enumerate3SidedUp(x_0, x_1, y_0)

Here the query region Q is same as that in Highest3SidedUp, and the objective is to report all the points of P in the region Q. This algorithm uses the same approach as algorithm Highest3SidedUp, and is presented in Algorithm 6. It uses the procedures EnumerateLeft, EnumerateLeftIn, EnumerateRight and EnumerateRightIn which are similar to CheckLeft, CheckLeftIn, CheckRight and CheckRightIn, respectively. The only difference is that all these procedures call the procedure Explore$(t, y_0, level(t))$ instead of ExUpdateHighest(t) and reporting of t is done instead of UpdateHighest(t).

The algorithm Explore$(t, y_0, level(t))$ reports all points in T_t whose y-coordinates are at least y_0. An in-order traversal is performed in T_t. At each node, the corresponding point is reported if it lies in Q. If two consecutive nodes on a path do not lie in Q, then the traversal along that path does not progress further. A variable *level* is used during the traversal that indicates the level of the current node. This is needed since we are using MinMaxPST. Thus observing whether the current node is a t even or odd level becomes important. The input parameter *level*(t) is used to initialize the variable *level* The procedure Explore is described in Algorithm 7. It uses two variables *current* and *state* satisfying the following invariant:

Algorithm 6: ENUMERATE3SIDEDUP(x_0, x_1, y_0)

```
Input: Real numbers $x_0$, $x_1$, and $y_0$ defining the region $Q = [x_0, x_1] \times [y_0, \infty)$.
Output: All elements of $Q \cap P$.
if $x(root(T)) < x_0$ then
    $p = root(T)$ ; $L = 1$; $L' = R = R' = 0$
else if $x(root(T)) < x_1$ then
    $p' = root(T)$; $L' = 1$; $L = R = R' = 0$
else
    $q = root(T)$; $R = 1$; $L = L' = R' = 0$
while $L = 1 \vee L' = 1 \vee R = 1 \vee R' = 1$ do
    $\mathcal{I} = \{z \in \{L, L', R, R\} | z = 1\}$;
    $z$ = element of $\mathcal{I}$ for which $level(N(z))$ is minimum;
    if $z = L$ then
        EnumerateLeft($p$);
    else if $z = L'$ then
        EnumerateLeftIn($p'$);
    else if $z = R$ then
        EnumerateRight($q$);
    else
        EnumerateRightIn($q'$);
```

- *current* is a node in T_p.
- $y(current) \geq y_0$.
- $state \in \{0, 1, 2\}$.
- If $state = 1$, then all elements of the set $Q \cap (\{current\} \cup T_{current_l})$ have been reported, where $T_{current_l}$ is the left child of *current* in the tree T.
- If $state = 2$, then all elements of the set $Q \cap T_{current}$ have been reported.

As in Section 3.2, it can be shown that the number of iterations of the while-loop of algorithm ENUMERATE3SIDEDUP is at most four times the height of T. Thus, we have the following result.

Lemma 3.4. *Given an in-place* MINMAXPST *in an array P and a query range $Q = [x_0, x_1] \times [y_0, \infty)$, the algorithm* ENUMERATE3SIDEDUP(x_0, x_1, y_0) *correctly computes all elements of P in the region Q in $O(\log n + |P_Q|)$ time using $O(1)$ extra work space, where P_Q is the set of points in the region Q.*

Remark 2. *The algorithms* LEFTMOSTSE RIGHTMOSTNW*and* RIGHTMOSTSW *are similar to algorithm* LEFTMOSTNE. *Similarly,* LOWEST3SIDEDDOWN *and* ENUMERATE3SIDEDDOWN *are analogous to* HIGHEST3SIDEDUP *and* ENUMERATE3SIDEDUP*, respectively. The algorithms* HIGHESTNE, HIGHESTNW *can be formulated using* HIGHEST3SIDEDUP*. Similarly,* LOWESTSE *and* LOWESTSW *can be formulated using* LOWEST3SIDEDDOWN.

4 Enumerating Maximal Empty Rectangles

Let $R = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$ be an axis-parallel rectangular region containing a set P of n points. An *empty rectangle* is an axis parallel rectangle inside R that contains no point of P inside it. A *maximal empty rectangle* (MER) is an empty rectangle that is not properly contained in any

Algorithm 7: EXPLORE$(p, y_0, level)$

```
Input: A node $p$ and its $level$ in $T$ and a real number $y_0$.
Output: All points $q$ in $T_p$ for which $y(q) \geq y_0$.
1  $current = p$;
2  $state = 0$;
3  while $current \neq p$ or $state \neq 2$ do
4      if $state = 0$ then
5          if $y(current) \geq y_0$ then
6              report $current$;
7          if ($current$ has a left child) $\bigwedge$ $(y(current_l) \geq y_0 \bigvee (y(current) \geq y_0 \bigwedge level \mod 2 = 0))$ then
8              $current = current_l$; $level = level + 1$;
9          else
10             $state = 1$;
11     else
12         if $state = 1$ then
13             if ($current$ has a right child) $\bigwedge$ $(y(current_r) \geq y_0 \bigvee (y(current) \geq y_0 \bigwedge level$
               $\mod 2 = 0))$ then
14                 $current = current_r$; $level = level + 1$; $state = 0$;
15             else
16                 $state = 2$;
17         else
18             // $state = 2$ and $current \neq p$
19             if $current$ is the left child of its parent then
20                 $state = 1$;
21             $current = parent(current)$; $level = level - 1$;
```

other empty rectangle. Each edge of an MER either contains a point in P or coincides with the boundary of R. We will consider the problem of designing an in-place algorithm for enumerating all the MERs using $O(1)$ extra-space. It is to be noted that we can use this algorithm for optimizing any measure among all the MERs.

Namaad et al [15] proposed an algorithm for this problem which runs in $O(\min(n^2, m \log n))$ time, where m is the number of all possible MERs. They showed that in the worst case m may be $O(n^2)$; but if the points are randomly placed, then the expected value of m is $O(n \log n)$.

Atallah and Frederickson [4] and Orlowski [17] proposed simple algorithms for enumerating all the MERs that run in $O(m + n \log^2 n)$ and $O(m + n \log n)$ time respectively. If the objective is only to identify the largest MER, then efficient algorithms are available. Chazelle et al. [11] proposed an $O(n \log^3 n)$ time algorithm for this problem. Later Aggarwal and Suri [1] improved the time complexity to $O(n \log^2 n)$. The same time complexity result holds for the recognition of the largest MER among a set of arbitrary polygonal obstacles [16]. Boland and Urrutia [6] gave an $O(n \log n)$ time algorithm for finding the largest MER inside an n-sided simple polygon. None of these algorithms is in-place; each of them uses $O(n)$ work-space apart from the input array.

4.1 Algorithm

We broadly divide MERs into the following three categories:

Type-A: The MERs whose top and bottom boundaries are aligned with the top and bottom boundaries of R respectively.

Type-B: The MERs whose top boundary is aligned with the top boundary of R, but bottom boundary passes through a point in P.

Type-C: The MERs whose top boundary passes through a point in P. Its bottom boundary may or may not pass through a point in P.

Clearly, each MER belongs to exactly one of the above three types. Now, we will show that the MaxPST defined in Section 1, can be used for solving our problem in an in-place manner using only $O(1)$ extra-space.

Algorithm EnumerateMER, given in Algorithm 8, enumerates all MERs inside R among the set P of points. It calls the procedures EnumerateTypeA, EnumerateTypeB and EnumerateTypeC to enumerate all *Type-A*, *Type-B* and *Type-C* MERs respectively. The EnumerateTypeB and EnumerateTypeC procedures need the MaxPST data structure. So, before executing these two procedures, we need to invoke the ConstructMaxPST procedure for building MaxPST. We describe the three procedures separately in the following subsections.

```
Algorithm 8: EnumerateMER
  Input: An axis-parallel rectangular region $R = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$ containing a set $P$ of $n$ points.
  Output: All axis-parallel maximal empty rectangles inside $R$.
1 EnumerateTypeA($R, P$);
2 ConstructMaxPST($P$);
3 EnumerateTypeB($R, P$);
4 EnumerateTypeC($R, P$);
```

4.1.1 EnumerateTypeA(R, P)

Type-A MERs are easy to obtain. First sort the points in P in increasing order of their x-coordinate values in an in-place manner. Now each pair of consecutive points defines a *Type-A* MER. Algorithm EnumerateTypeA, given in Algorithm 9, computes all the *Type-A* MERs.

```
Algorithm 9: EnumerateTypeA($R, P$)
  Input: an axis-parallel rectangular region $R = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$ containing a set $P$ of $n$ points.
  Output: All the Type-A MERs inside $R$
1 HeapSort($P$);
2 $top = y_{max}$; $bottom = y_{min}$; $left = x_{min}$;
3 for $i = 1$ to $n$ do
4     $right = x(P[i])$;
5     report rectangle $(left, right, top, bottom)$ ;
6     $left = right$;
7 $right = x_{max}$;
8 report rectangle $(left, right, top, bottom)$ ;
```

4.1.2 EnumerateTypeB(R, P)

A *Type-B* MER is an MER whose top boundary coincides with the top boundary of R but bottom boundary passes through a point in P. The number of *Type-B* MERs is at most n. The reason is as follows: Let the bottom boundary of a *Type-B* MER pass through a point $p \in P$. Its left and right boundaries are uniquely defined by two points $\textsc{RightMostNW}(x(p), y(p))$ and $\textsc{LeftMostNE}(x(p), y(p))$ respectively. The algorithm $\textsc{EnumerateTypeB}$ is stated below. For a given point $p \in P$, it uses $\textsc{MaxPST}$ for getting the points $\textsc{RightMostNW}(x(p), y(p))$ and $\textsc{LeftMostNE}(x(p), y(p))$ in $O(\log n)$ time using $O(1)$ extra space (see Section 3.1). Thus, the time required for computing all the *Type-B* MERs present on R is $O(n \log n)$.

Algorithm 10: $\textsc{EnumerateTypeB}(R, P)$

```
Input: An axis-parallel rectangular region $R = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$ and the MaxPST of the
       set $P$ of $n$ points.
Output: All the $Type$-$B$ MERs inside $R$.
1  $top = y_{max}$;
2  for $i = 1$ to $n$ do
3      $bottom = y(P[i])$;
4      $l = \textsc{RightMostNW}(x(P[i]), y(P[i]))$;
5      if $l = (\infty, -\infty)$ then
6          $left = x_{min}$;
7      else
8          $left = x(l)$;
9      $r = \textsc{LeftMostNE}(x(P[i]), y(P[i]))$;
10     if $r = (\infty, -\infty)$ then
11         $right = x_{max}$;
12     else
13         $right = x(r)$;
14     report $rectangle$ $(left, right, top, bottom)$ ;
```

4.1.3 $\textsc{EnumerateTypeC}(R, P)$

Let $p^* \in P$ be at the root of the $\textsc{MaxPST}$. We now describe the method of computing all the *Type-C* MERs with the point $p^* \in P$ on its top boundary. Note that, while computing the *Type-C* MERs with top boundary passing through the point p^*, all the points $\{q | y(q) \geq y(p^*)\}$ can be ignored. Thus, we delete the point p^* from $\textsc{MaxPST}$.

Let us now consider a drop-down curtain whose top is fixed at p^*, and its $left$ and $right$ sides are initialized by x_{min} and x_{max} respectively. Imagine lowering the curtain till either it reaches a point of P or reaches the bottom boundary of R. If it reaches a point of P, say p, then p satisfies the following:

(i) $left < x(p) < right$
(ii) $y(p) < y(p^*)$
(iii) $y(p)$ is maximum among all the points in P satisfying (i) and (ii).

Note that, if $\textsc{MaxPST}$ contains all the points $\{q | y(q) < y(p^*)\}$, then p can be computed by performing the $\textsc{Highest3SidedUp}(left,\ right, y_{min})$ query in $\textsc{MaxPST}$. If $p \neq NIL$, then we report a *Type-C* rectangle, and adjust $left$ or $right$ as follows: if $x(p) < x(p^*)$ then $left = x(p)$, otherwise $right = x(p)$. We further pull the curtain down till it meets a point of P or the bottom boundary of R, and report the next *Type-C* MER. By repeating the above steps, we report all *Type-C* MERs whose top boundary is anchored at p^* (see Figure 8).

The iteration continues until $\textsc{MaxPST}$ becomes empty. In Subsection 4.1.4, we show that deletion of the root from a $\textsc{MaxPST}$ can be performed in $O(\log n)$ time with $O(1)$ extra storage, where n

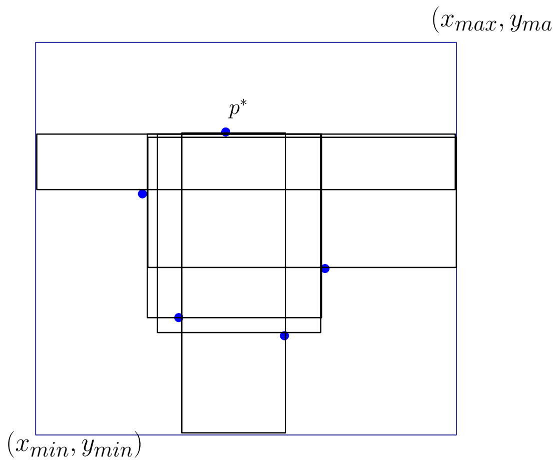


Figure 8: Enumerating *Type-C* MER keeping the point p^* at the top boundary.

is the number of points in the MaxPST. In Subsection 4.1.5, we mention the changes required in the algorithm Highest3SidedUp to handle the dynamically changing MaxPST.

If m' be the number of MERs reported with top boundary passing through p^*, then the time complexity of this iteration is $O(m' \log n)$. The detailed steps of this procedure is given in Algorithm EnumerateTypeC.

Algorithm 11: EnumerateTypeC(R,P)

```
Input: An axis-parallel rectangular region $R = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$ and the MaxPST of the
       set $P$ of $n$ points.
Output: All the Type-C MERs inside $R$.
$i = 0$;
while $i \neq n$ do
    $p^* = $ DeleteTop($P$);
    $top = y(p^*)$; $left = x_{min}$; $right = x_{max}$;
    repeat
        $h = $ Highest3SidedUp($left, right, y_{min}$);
        if $h = (\infty, -\infty)$ then
            $bottom = y_{min}$;
        else
            $bottom = y(h)$;
        report rectangle ($left, right, top, bottom$);
        if $h \neq (\infty, -\infty)$ then
            if $x(h) < x(p^*)$ then
                $left = x(h)$;
            else
                $right = x(h)$;
    until $h = (\infty, -\infty)$;
    $i = i + 1$;
```

4.1.4 DeleteTop

The procedure DeleteTop works as follows. It starts from the root $P[1]$ by assigning $i = 1$. In each step, it considers both the children of $P[i]$, and choose the one, say $P[j]$ having maximum y-coordinate. The tie, if arises, is broken arbitrarily. It then swaps $P[i]$ and $P[j]$, and moves to the next step by setting $i = j$, provided $P[j]$ is not a leaf node. Note that, after the deletion of a point from MaxPST, it still remains in the array P maintaining the following invariants:

(i) MaxPST contains all the points $p \in P$ satisfying $y(p) < y(p^*)$, and

(ii) any point $p \in P$ is in MaxPST if and only if $y(p) < y(parent(p))$.

Thus, the above two invariants say that a node $P[j]$ has no left (resp. right) child if

- $2j > n$ (resp. $2j + 1 > n$), or
- y-coordinate of $P[2j]$ (resp. $P[2j + 1]$) is greater than the y-coordinate of the root of T,

Note that, in this dynamic setup under the deletion of the root of the MaxPST, the tree may not remain balanced after the deletion of some points. In other words, leaf nodes may appear in different levels. However, the length of the each search path will still be bounded by $O(\log n)$.

Algorithm 12: DeleteTop(P)

```
Input: The array $P[1, \ldots, n]$ containing the max priority search tree $T$
Output: The top of the $T$ and delete it from $T$
$i = 1$;
while $(2i \le n \wedge y(P[2i]) < y(P[i])) \bigvee ((2i + 1) \le n \wedge y(P[2i + 1]) < y(P[i])$ do
    if $2i + 1 \le n$ then
        if $y(P[2i]) < y(P[i]) \wedge y(P[2i + 1]) < y(P[i])$ then
            if $y(P[2i + 1]) \ge y(P[2i])$ then
                $j = 2i + 1$;
            else
                $j = 2i$;
        else if $y(P[2i + 1]) < y(P[i])$ then
            $j = 2i + 1$;
        else
            $j = 2i$;
    else
        $j = 2i$;
    Swap $P[i]$ and $P[j]$ and set $i = j$;
Return($P[j]$);
```

4.1.5 Highest3SidedUp

In Section 3.2, we described the procedure Highest3SidedUp; it is mentioned that if a node has only one child then that must be the left child. But, in the dynamic setup under the deletion, there may exist some node in MaxPST which has only the right child. Thus the necessary modification (in the case *p has only one child*) is needed in the procedures CheckLeft, CheckRight, CheckLeftIn and CheckRightIn, that are invoked by Highest3SidedUp presented in Algorithm 3. However, the time complexity of the Highest3SidedUp query remains $O(\log n)$ using $O(1)$ extra space.

4.2 Correctness and Complexity

The correctness of the Algorithm 9 is easy to observe. The correctness of Algorithm 10 follows from the correctness of the procedures LeftMostNE and RightMostNW. Finally, the correctness of the Algorithm 11 follows from the invariants (i) and (ii) maintained by MaxPST after the deletion of its root node, as mentioned in Subsection 4.1.4. The following theorem states the final result.

Theorem 4.1. *Given a rectangular region R containing a set of n points, all the maximal empty axis parallel rectangles can be enumerated in $O(m \log n)$ time with $O(1)$ extra space, where m is the number of all maximal empty rectangles.*

Proof. The algorithm EnumerateMER correctly enumerates all MERs exactly once as each of EnumerateTypeA, EnumerateTypeB and EnumerateTypeC enumerates correctly each of the *Type-A*, *Type-B* and *Type-C* rectangles respectively exactly once. The time complexity of each of the procedures EnumerateTypeA, ConstructMaxPST and EnumerateTypeB is $O(n \log n)$. We now analyze the time complexity of the Algorithm EnumerateTypeC.

In each iteration of the while loop of EnumerateTypeC, we first delete the point p^* at the *root* using the procedure DeleteTop. This needs $O(\log n)$ time since the root needs to move at most $\log n$ levels for the deletion. Each iteration of the inner loop reports a *Type-C* MER with top boundary defined by p. Here a point is identified that obstructs the corresponding curtain using the procedure Highest3SidedUp. This needs $O(\log n)$ time. Thus, the Procedure Enumerate-TypeC takes $O(m' \log n + n \log n)$ time, where m' is the number of *Type-C* rectangles present in R. Since, the number of *Type-A* and *Type-B* MERs are both $O(n)$, the time complexity result follows. From the algorithm it is obvious that, apart from the array P, the extra space used in the algorithm is $O(1)$. □

Corollary 4.2. *A maximum area/perimeter MER can be computed in $O(m \log n)$ time with $O(1)$ extra space, where m is the number of all possible MERs.*

5 Conclusion

An in-place algorithm for constructing a MinMaxPST tree with a set of points in $\mathbb{R}^2$ is presented in this paper. The worst case time complexity of this algorithm is $O(n \log n)$ and it uses $O(1)$ extra workspace, and stores the MinMaxPST in the same array that contains the input. All the standard query algorithms on the priority search tree [14] can be executed on this data structure in logarithmic time using $O(1)$ extra work-space. The implementation of this algorithm is available in [18]. As an application of this algorithm, we proposed an in-place algorithm of enumerating all the maximal empty axis-parallel rectangles (MER) among the point set P inside an axis-parallel rectangular region R. It runs in $O(m \log n)$ time and $O(1)$ extra workspace apart from the input array P, where m is the number of MERs present on R. A careful look into this algorithm may reduce the time complexity to $O(m + n \log n)$ which is best known for this problem with $O(n)$ extra workspace [17].

Acknowledgment: We thank Simon Pratt for his useful comments on a preliminary version of this paper.

References

[1] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Symposium on Computational Geometry*, pages 278–290, 1987.

[2] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *CoRR*, abs/1112.5904, 2011.

[3] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *JoCG*, 2(1):46–68, 2011.

[4] M. J. Atallah and G. N. Frederickson. A note on finding a maximum empty rectangle. *Discrete Applied Mathematics*, 13(1):87 – 91, 1986.

[5] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29(10):996–1000, 1986.

[6] R. P. Boland and J. Urrutia. Finding the largest axis aligned rectangle in a polygon in o(n log n) time. In *CCCG*, pages 41–44, 2001.

[7] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Comput. Geom.*, 37(3):209–227, 2007.

[8] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Symp. on Comput. Geom.*, pages 239–246, 2004.

[9] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theor. Comput. Sci.*, 321(1):25–40, 2004.

[10] T. M. Chan and E. Y. Chen. Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection. *Comput. Geom.*, 43(8):636–646, 2010.

[11] B. Chazelle, R. L. S. D. III, and D. T. Lee. Computing the largest empty rectangle. *SIAM J. Comput.*, 15(1):300–315, 1986.

[12] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *CCCG*, pages 68–71, 2003.

[13] J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32(4):580–585, 1992.

[14] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[15] A. Naamad, D. T. Lee, and W. L. Hsu. On the maximum empty rectangle problem. *Discrete Applied Mathematics*, 8(3):267 – 277, 1984.

[16] S. C. Nandy, A. Sinha, and B. B. Bhattacharya. Location of the largest empty rectangle among arbitrary obstacles. In *FSTTCS*, pages 159–170, 1994.

[17] M. Orlowski. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5(1):65–73, 1990.

[18] S. Pratt. Implementation of a heap-based priority search tree. *http://github.com/spratt/PrioritySearchTree.*

[19] J. Vahrenhold. An in-place algorithm for Klee's measure problem in two dimensions. *Inf. Process. Lett.*, 102(4):169–174, 2007.

[20] J. Vahrenhold. Line-segment intersection made in-place. *Comput. Geom.*, 38(3):213–230, 2007.