An In-Place 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 only stores one point of the point set. We show that 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.

1 Introduction

Let P be a set of n points in $\mathbb{R}^2$. A *priority search tree*, as introduced by McCreight [2], 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, 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 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-coordinates of the points stored at u and any of its children.

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, viz., the point in the left subtree of u whose x-coordinate is maximum. Thus, the data structure uses $O(n)$ space and, by taking for T a balanced tree, several types of range queries can be answered efficiently:

- HIGHESTNE(x_0, y_0): report the highest point of P in the north-east quadrant of the query point (x_0, y_0).
- LEFTMOSTNE(x_0, y_0): report the leftmost point of P in the north-east quadrant of the query point (x_0, y_0).
- HIGHEST3SIDED(x_0, x_1, y_0); report the highest point of P in the 3-sided query range $[x_0, x_1] \times [y_0, \infty)$.
- ENUMERATE3SIDED(x_0, x_1, y_0); report all points of P in the 3-sided query range $[x_0, x_1] \times [y_0, \infty)$.

The first three queries can be answered in $O(\log n)$ time, whereas the fourth query takes $O(\log n + m)$ time, where m is the number of points of P that are in the query range.

In this paper, we show that these results can also be obtained without storing the "splitting" points p_u at the internal nodes 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[1] $0, 1, \ldots, h-1$ 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.

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

Theorem 1 *Let P be a set of n points in $\mathbb{R}^2$.*

1. *The in-place priority search tree can be constructed in $O(n \log n)$ time using $O(1)$ extra space.*
2. *Each of the queries* HIGHESTNE, LEFTMOSTNE, *and* HIGHEST3SIDED *can be answered in $O(\log n)$ time using $O(1)$ extra space.*
3. *The query* ENUMERATE3SIDED *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.*

*Research supported by NSERC and the Commonwealth Scholarship Program of DFAIT.

†Indian Statistical Institute, Kolkata, India. Part of this work was done while M.D. was visiting Carleton University, Ottawa, Canada. minati.isi@gmail.com

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

[1]The root is at level 0.

For ease of presentation, we assume that no two points in the set P have the same x-coordinates and no two points in P have the same 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.

2 Constructing the in-place priority search tree

Let $h = \lfloor \log n \rfloor$ be the height of the priority search tree. Our algorithm constructs the tree level by level and 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 sorted by their x-coordinates.

Algorithm 1: ConstructPST

Input: An array $P[1 \dots n]$ of points in $\mathbb{R}^2$.
Output: The priority search tree of those points stored in P.

```
1  h = ⌊log n⌋; A = n − (2^h − 1);
2  HeapSort(1, n);
3  for i = 0 to h − 1 do
4  |  k = ⌊A/2^{h−i}⌋;
5  |  K1 = 2^{h+1−i} − 1;
6  |  K2 = 2^{h−i} − 1 + A − k2^{h−i};
7  |  K3 = 2^{h−i} − 1;
8  |  for j = 1 to k do
9  |  |  ℓ = index in {2^i + (j−1)K1, ..., 2^i + jK1 − 1} such that
   |  |  y(P[ℓ]) is maximum;
10 |  |  swap P[ℓ] and P[2^i + j − 1] ;
11 |  if k < 2^i then
12 |  |  ℓ = index in {2^i + kK1, ..., 2^i + kK1 + K2 − 1} such that
   |  |  y(P[ℓ]) is maximum;
13 |  |  swap P[ℓ] and P[2^i + k];
14 |  |  m = 2^i + kK1 + K2;
15 |  |  for j = 1 to 2^i − k − 1 do
16 |  |  |  ℓ = index in {m + (j−1)K3, ..., m + jK3 − 1} such
   |  |  |  that y(P[ℓ]) is maximum;
17 |  |  |  swap P[ℓ] and P[2^i + k + j];
18 |  HeapSort(2^{i+1}, n);
```

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

Let i be an index with $0 \le i < h$, and consider the i-th step 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. 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 $K_3 = 2^{h-i} - 1$.

We divide the subarray $P[2^i \dots n]$ into 2^i *blocks*: If $k = 2^i$, then there are k blocks 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 point. These highest points are swapped with the subarray $P[2^i \dots 2^{i+1} - 1]$. At this moment, level i of the tree has been constructed, but 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 complete algorithm for constructing the in-place priority search tree is given in Algorithm 1. It uses algorithm HeapSort(m, n), which runs the heapsort algorithm on the subarray $P[m \dots n]$.

The correctness of this algorithm follows by observing that the invariant is correctly maintained. The initial sorting in line 2 takes $O(n \log n)$ time using $O(1)$ extra space. Each of the $h = \lfloor \log n \rfloor$ iterations of the main 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 for-loop. Thus, the entire algorithm ConstructPST takes $O(n \log^2 n)$ time and uses $O(1)$ extra space.[2]

3 Queries on the in-place priority search tree

In this section, we present the algorithms for the query problems mentioned in Section 1. For ease of presentation, we describe the algorithms using the terminology of trees. We will denote by T the priority search tree that is implicitly defined by the array $P[1 \dots n]$ that results by running algorithm ConstructPST. 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 this node has a left child, which is stored at $P[2i]$. If $2i + 1 \le n$, then this node has a right child, which is stored at $P[2i + 1]$. This node is a leaf if and only if $2i > n$. We will identify each node in T with the point of P stored at that node. For any p in P, we denote by T_p the subtree rooted at p. Furthermore, the left and

[2] Using the in-place algorithm of Katajainen and Pasanen [1] that stably sorts a sequence of n bits in $O(n)$ time, the running time can be improved to $O(n \log n)$ with $O(1)$ extra space. The details will be given in the full paper.

right children of p (if they exist) are denoted by p_l and p_r, respectively.

3.1 HIGHESTNE(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). If $Q \cap P \neq \emptyset$, define p^* to be the highest point of P in Q. If $Q \cap P = \emptyset$, define p^* to be the point $(\infty, -\infty)$. Algorithm HIGHESTNE(x_0, y_0) will return the point p^*.

The algorithm uses two variables *best* and p, which satisfy the following invariant:

- If $Q \cap P \neq \emptyset$, then $p^* \in \{best\} \cup T_p$.
- If $Q \cap P = \emptyset$, then $p^* = best$.

The algorithm initializes $best = (\infty, -\infty)$ and $p = root(T)$. During the algorithm, p moves down the tree according to the relative positions of p, its children, and the quadrant Q. The algorithm is given in Algorithm 2. It uses the procedure UPDATEHIGHEST(t), which takes as input a point t and does the following: If $t \in Q$ and $y(t) > y(best)$ then it assigns $best = t$.

Algorithm 2: HIGHESTNE(x_0, y_0)

Input: Real numbers x_0 and y_0 defining the north-east quadrant Q.
Output: The highest point p^* in $Q \cap P$, if it exists; otherwise the point $(\infty, -\infty)$.

```
1  best = (∞, −∞); p = root(T);
2  while p is not a leaf do
3      if p ∈ Q then
4          UpdateHighest(p); p = p_l;
5      else if y(p) < y_0 then
6          p = p_l;
7      else if p has one child then
8          p = p_l;
9      else if x(p_r) ≤ x_0 then
10         p = p_r;
11     else if x(p_l) ≥ x_0 then
12         p = higher among p_l and p_r;
13     else if y(p_r) < y_0 then
14         p = p_l;
15     else
16         UpdateHighest(p_r); p = p_l;
17 UpdateHighest(p);
18 return best;
```

The correctness of this algorithm follows from the fact that the invariant is correctly maintained. Since in each iteration, p moves down the tree, the while-loop makes $O(\log n)$ iterations, each one taking $O(1)$ time. Thus, the total time for algorithm HIGHESTNE is $O(\log n)$. It follows from the algorithm that it uses $O(1)$ extra space.

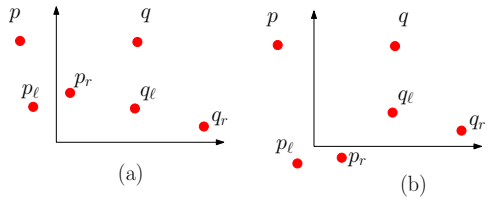

Figure 1: Two cases for LEFTMOSTNE(x_0, y_0).

3.2 LEFTMOSTNE(x_0, y_0)

As before, let $Q = [x_0, \infty) \times [y_0, \infty)$ be the north-east quadrant of the point (x_0, y_0). If $Q \cap P \neq \emptyset$, define p^* to be the leftmost point of P in Q. If $Q \cap P = \emptyset$, define p^* to be the point (∞, ∞). Algorithm LEFTMOSTNE(x_0, y_0) will return the point p^*.

The algorithm uses three variables *best*, p, and q, which satisfy the following invariant:

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

The algorithm starts by initializing $best = (\infty, \infty)$, $p = root(T)$, and $q = root(T)$. During the algorithm, p and q move down the tree according to the relative positions of their children and the quadrant Q. The algorithm is given in Algorithm 3. It uses the procedure UPDATELEFTMOST(t), which takes as input a point t and does the following: If $t \in Q$ and $x(t) < x(best)$ then it assigns $best = t$.

It follows by a careful case analysis that the invariant is correctly maintained, implying 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.

3.3 HIGHEST3SIDED(x_0, x_1, y_0)

The three real numbers x_0, x_1, and y_0 define the three-sided range $Q = [x_0, x_1] \times [y_0, \infty)$. If $Q \cap P \neq \emptyset$, define p^* to be the highest point of P in Q. If $Q \cap P = \emptyset$, define p^* to be the point $(\infty, -\infty)$. Algorithm HIGHEST3SIDED(x_0, x_1, y_0) returns the point p^*.

The algorithm uses two bits L and R, and three variables *best*, p, and q. As before, *best* stores the highest point in Q found so far. The bit L indicates whether or not p^* may be in the subtree of p; if $L = 1$, then p is to the left of Q. Similarly, the bit R indicates whether or not p^* may be in the subtree of q; if $R = 1$, then q is to

Algorithm 3: 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 (∞, ∞).

```
1  best = (∞,∞); p = root(T); q = root(T);
2  while p is not a leaf do
3  |  UpdateLeftMost(p); UpdateLeftMost(q);
4  |  if p = q then
5  |  |  if p has one child then
6  |  |  |  q = p_l; p = p_l;
7  |  |  else
8  |  |  |  q = p_r; p = p_l;
9  |  else
10 |  |  // p ≠ q
11 |  |  if q is leaf then
12 |  |  |  q = p;
13 |  |  else if q has one child then
14 |  |  |  if y(q_l) < y_0 then
15 |  |  |  |  q = p_r; p = p_l;
16 |  |  |  else if y(p_r) < y_0 then
17 |  |  |  |  p = p_l; q = q_l;
18 |  |  |  else if x(q_l) < x_0 then
19 |  |  |  |  p = q_l; q = q_l;
20 |  |  |  else if x(p_r) < x_0 then
21 |  |  |  |  p = p_r; q = q_l;
22 |  |  |  else
23 |  |  |  |  q = p_r; p = p_l;
24 |  |  else
25 |  |  |  // q has two children
26 |  |  |  if x(p_r) ≥ x_0 and y(p_r) ≥ y_0 then
27 |  |  |  |  q = p_r;p = p_l; // Fig. 1(a)
28 |  |  |  else if x(p_r) < x_0; then
29 |  |  |  |  if x(q_l) < x_0 then
30 |  |  |  |  |  p = q_l; q = q_r;
31 |  |  |  |  else if y(q_l) < y_0 then
32 |  |  |  |  |  p = p_r; q = q_r;
33 |  |  |  |  else
34 |  |  |  |  |  p = p_r; q = q_l;
35 |  |  |  else
36 |  |  |  |  // x(p_r) ≥ x_0 and y(p_r) < y_0
37 |  |  |  |  if y(p_l) < y_0 then
38 |  |  |  |  |  p = q_l; q = q_r;  // Fig. 1(b)
39 |  |  |  |  else
40 |  |  |  |  |  p = p_l;
41 |  |  |  |  |  if y(q_l) ≥ y_0 then
42 |  |  |  |  |  |  q = q_l
43 |  |  |  |  |  else
44 |  |  |  |  |  |  q = q_r
45 UpdateLeftMost(p); UpdateLeftMostq;
46 return best;
```

the right of Q. More formally, the variables satisfy the following invariant:

- If $L = 1$ then $x(p) < x_0$.
- If $R = 1$ then $x(q) > x_1$.
- If $Q \cap P \neq \emptyset$, then $p^* \in \{best\} \cup \big(\cup_{z \in \mathcal{I}} T_{N(z)}\big)$, where $\mathcal{I} = \{z \in \{L, R\} | z = 1\}$ and

$$N(z) = \begin{cases} p & \text{if } z = L, \\ q & \text{if } z = R. \end{cases}$$

- If $Q \cap P = \emptyset$, then $best = (\infty, -\infty)$.

The algorithm is given in Algorithm 4. In the initialization, the variables L, R, $best$, p, and q are assigned depending on the position of the root of T with respect to the query region Q.

At any moment during the algorithm, if $L = 1$, then we say that p is an *observing point*. Similarly, if $R = 1$, we say that q is an observing point.

Consider one iteration of the while-loop. The algorithm chooses an observing point that is closest to the root of T. (For ease of presentation, our pseudocode does not explicitly maintain the levels in T of p and q.) If this point is p, algorithm CHECKLEFT(p) is called; otherwise, algorithm CHECKRIGHT(q) is called.

Algorithm 4: HIGHEST3SIDED(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 $(\infty, -\infty)$.

```
1  best = (∞,−∞);
2  if x_0 ≤ x(root(T)) ≤ x_1 then
3  |  L = 0; R = 0;
4  |  if y(root(T)) ≥ y_0 then
5  |  |  best = root(T)
6  else if x(root(T)) < x_0 then
7  |  p = root(T); L = 1; R = 0;
8  else
9  |  q = root(T); L = 0; R = 1
10 while L = 1 ∨ R = 1 do
11 |  I = {z ∈ {L, R} | z = 1};
12 |  z = element of I for which level(N(z)) is
   |  minimum;
13 |  if z = L then
14 |  |  CheckLeft(p);
15 |  else
16 |  |  CheckRight(q);
17 return best;
```

We describe the procedure for CHECKLEFT(p) in Algorithm 5. The procedure for CHECKRIGHT(q) is symmetric and omitted from this paper. Both these procedures use algorithm UPDATEHIGHEST(t), which takes

as input a point t and does the following: If $t \in Q$ and $y(t) > y(best)$ then it assigns $best = t$.

Algorithm 5: CHECKLEFT(p)

```
Input: A node p such that x(p) < x_0.
1  if p is a leaf then
2  |  L = 0
3  else if p has one child then
4  |  if x_0 ≤ x(p_l) ≤ x_1 then
5  |  |  UPDATEHIGHEST(p_l); L = 0;
6  |  else if x(p_l) < x_0 then
7  |  |  p = p_l
8  |  else
9  |  |  q = p_l; R = 1; L = 0
10 else
11 |  // p has two children
12 |  if x(p_l) < x_0 then
13 |  |  if x(p_r) < x_0 then
14 |  |  |  p = p_r
15 |  |  else if x(p_r) ≤ x_1 then
16 |  |  |  UPDATEHIGHEST(p_r);
17 |  |  |  p = p_l;
18 |  |  else
19 |  |  |  q = p_r; p = p_l; R = 1
20 |  else if x(p_l) ≤ x_1 then
21 |  |  UPDATEHIGHEST(p_l); L = 0 ;
22 |  |  if x(p_r) > x_1 then
23 |  |  |  q = p_r; R = 1;
24 |  |  else
25 |  |  |  UPDATEHIGHEST(p_r);
26 |  else
27 |  |  q = p_l; L = 0; R = 1
```

Consider the set $\mathcal{I}$ and the value of $\ell = level(N(z))$ in lines 11 and 12 of algorithm HIGHEST3SIDED. Assume that algorithm CHECKLEFT(p) is called. During this algorithm, either p moves one level down in the tree T or the bit L is set to 0. In addition, the point q either stays the same or it becomes a child of (the original) p. Therefore, in one iteration of the while-loop in algorithm HIGHEST3SIDED, the value of $\ell = level(N(z))$ either increases, or ℓ does not change in which case the size of the set $\{z' \in \mathcal{I} | level(N(z')) = \ell\}$ decreases. It follows that the number of iterations of the while-loop of algorithm HIGHEST3SIDED is at most twice the height of T, i.e., $O(\log n)$. Since each iteration takes $O(1)$ time, it follows that the total time for algorithm HIGHEST3SIDED is $O(\log n)$. It follows from the algorithm that it uses $O(1)$ extra space.

3.4 ENUMERATE3SIDED(x_0, x_1, y_0)

Given three real numbers x_0, x_1, and y_0, define the three-sided range $Q = [x_0, x_1] \times [y_0, \infty)$. Algorithm ENUMERATE3SIDED(x_0, x_1, y_0) returns all elements of $Q \cap P$. This algorithm uses the same approach as algorithm HIGHEST3SIDED. Besides the two bits L and R, it uses two additional bits L' and R'. Each of these four bits L, L', R, and R' corresponds to a subtree of T rooted at the points p, p', q, and q', respectively; if the bit is equal to one, then the subtree may contain points that are in the query region Q.

Algorithm 6: ENUMERATE3SIDED(x_0, x_1, y_0)

```
Input: Real numbers x_0, x_1, and y_0 defining the
       region Q = [x_0, x_1] × [y_0, ∞).
Output: All elements of Q ∩ P.
1  if y(root(T)) < y_0 then
2  |  L = L' = R = R' = 0
3  else if x(root(T)) < x_0 then
4  |  p = root(T) ; L = 1; L' = R = R' = 0
5  else if x(root(T)) < x_1 then
6  |  p' = root(T); L' = 1; L = R = R' = 0
7  else
8  |  q = root(T); R = 1; L = L' = R' = 0
9  while L = 1 ∨ L' = 1 ∨ R = 1 ∨ R' = 1 do
10 |  I = {z ∈ {L, L', R, R} | z = 1};
11 |  z = element of I for which level(N(z)) is
   |  minimum;
12 |  if z = L then
13 |  |  ENUMERATELEFT(p);
14 |  else if z = L' then
15 |  |  ENUMERATELEFTIN(p');
16 |  else if z = R then
17 |  |  ENUMERATERIGHT(q);
18 |  else
19 |  |  ENUMERATERIGHTIN(q');
```

The following invariant will be maintained:

- 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 $x(p') \leq x(q')$.
- All points in $(Q \cap P) \setminus \left(\cup_{z \in \mathcal{I}} T_{N(z)}\right)$ have been reported, where $\mathcal{I} = \{z \in \{L, L', R, R'\} | z = 1\}$ and

$$N(z) = \begin{cases} p & \text{if } z = L, \\ p' & \text{if } z = L', \\ q & \text{if } z = R, \\ q' & \text{if } z = R'. \end{cases}$$

The algorithm is given in Algorithm 6. In one iteration of the while-loop, the algorithm chooses an observing point that is closest to the root. Depending on this point, one of the procedures ENUMERATELEFT, ENUMERATELEFTIN, ENUMERATERIGHT, and

EnumerateRightIn is called. The first two procedures are given in Algorithms 7 and 8; the other two are symmetric and omitted from this paper.

Algorithm 7: EnumerateLeft(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
    if x_0 ≤ x(p_l) ≤ x_1 then
        if L' = 1 ∧ R' = 1 then
            Explore(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) ≤ x_1 then
            if L' = 1 ∧ R' = 1 then
                Explore(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) ≤ x_1 then
        if x(p_r) > x_1 then
            q = p_r; p' = p_l; L = 0; L' = R = 1;
        else
            if R' = 1 ∧ L' = 1 then
                Explore(p'); Explore(p_r);
            else if L' = 1 then
                Explore(p_r); q' = p'; R' = 1
            else if R' = 1 then
                Explore(p_r); L' = 1
            else
                q' = p_r; L' = R' = 1
            p' = p_l; L = 0
    else
        q = p_l; L = 0; R = 1
```

These procedures use algorithm Explore(t), which takes as input a node t in T and reports all points in T_t whose y-coordinates are at least y_0. This algorithm does an in-order traversal of T_t, using $O(1)$ extra space, and runs in time $O(1 + |Q \cap T_t|)$.

As in Section 3.3, it can be shown that the number of iterations of the while-loop of algorithm Enumerate3Sided is at most four times the height of T, i.e., $O(\log n)$. It follows that the total time for algorithm Highest3Sided is $O(\log n + |Q \cap P|)$. It follows from the algorithm that it uses $O(1)$ extra space.

Algorithm 8: EnumerateLeftIn(p')

```
Input: A node p' such that x_0 ≤ x(p') ≤ x_1.
if y(p') ≥ y_0 then
    report p';
if p' is a leaf then
    L' = 0
else if p' has one child then
    if x_0 ≤ x(p'_l) ≤ x_1 then
        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) ≤ x_1 then
            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) ≤ x_1 then
        if x(p'_r) > x_1 then
            q = p'_r; p' = p'_l; R = 1;
        else
            if R' = 1 then
                Explore(p'_r); p' = p'_l
            else
                q' = p'_r; p' = p'_l; R' = 1
    else
        q = p'_l; L' = 0; R = 1
```

4 Conclusion

Our motivation for creating an in-place priority search tree was for designing an in-place algorithm for finding the *maximum area axis-parallel empty rectangle* among a set of n point obstacles in a rectangular region. It can be shown that using our in-place priority search tree one can recognize the desired rectangle in $O(m \log n)$ time using $O(1)$ extra-space, where m is the number of all possible maximal empty rectangles. It will be worthwhile to find other applications where this tree may help in saving space.

References

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

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