1987

A New Approach to the Dynamic Maintenance of Maximal Points in a Plane

Greg N. Frederickson
Purdue University, gnf@cs.purdue.edu

Susan Rodger

Report Number:

87-658

TR 658

A New Approach to the Dynamic Maintenance of Maximal Points in a Plane*

Greg N. Frederickson and Susan Rodger

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA

Abstract. A point $p_i = (x_i, y_i)$ in the x–y plane is *maximal* if there is no point $p_j = (x_j, y_j)$ such that $x_j > x_i$ and $y_j > y_i$. We present a simple data structure, a dynamic contour search tree, which contains all the points in the plane and maintains an embedded linked list of maximal points so that m maximal points are accessible in $O(m)$ time. Our data structure dynamically maintains the set of points so that insertions take $O(\log n)$ time, a speedup of $O(\log n)$ over previous results, and deletions take $O((\log n)^2)$ time.

1. Introduction

Given a set S of points in the x–y plane, $p_i = (x_i, y_i) \in S$ is a maximal point if there is no $p_j = (x_j, y_j) \in S$ such that $p_i < p_j$ ($p_i < p_j$ iff $x_i < x_j$ and $y_i < y_j$). The set of maximal points of S form what we call an *m-contour*. In this paper we present a dynamic contour search tree, a data structure to represent contour information for a search tree. This data structure is quite natural, and is simpler than previous data structures for solving the dynamic version of the m-contour problem. The dynamic contour search tree stores the points of the set S in the leaves of the tree and stores various search information in the internal nodes. The m-contour is maintained as a linked list embedded in this structure, so that it can be accessed in $O(m)$ time where m is the number of maximal points. Our data structure dynamically maintains the set of points so that insertions take $O(\log n)$ time, a speedup of $O(\log n)$ over previous results, and deletions take $O((\log n)^2)$ time.

In addition, the query of whether a new point lies inside or outside of the m-contour can be determined in $O(\log n)$ time.

For a static set of n points, Kung et al. [1] have shown that the m-contour can be computed in $O(n \log n)$ time. One method for constructing the m-contour proceeds by examining all the points, one at a time, and building the m-contour from the points examined thus far. If a point is not on the m-contour then the point is discarded.

In the dynamic version of this problem, those points in the set S which are not maximal points must be saved, because deleting a point on the m-contour can result in a point which was not maximal becoming maximal. Overmars and van Leeuwen [2] use a balanced binary search tree in which the points from S are stored in the leaves of the tree and each internal node has a concatenable queue associated with it. The concatenable queue at the root contains all the maximal points and the other concatenable queues contain the remaining points. In the data structure of Overmars and van Leeuwen, the m-contour can be listed in $O(m)$ time, and the query of whether a new point lies inside or outside of the m-contour can be determined in $O(\log n)$ time. However, Overmars and van Leeuwen's insertions and deletions consist of tearing apart concatenable queues and rebuilding them at each internal node along a path, causing insertions and deletions to take $O((\log n)^2)$ time. With our data structure, insertions and deletions consist of updating $O(1)$ search information at each internal node along a path, causing insertions to take $O(\log n)$ time.

Willard and Lueker [4] have shown how to perform range queries in k dimensions on trees of bounded balance with worst-case times of $O((\log n)^{k-1})$ for update and query operations. Applying their data structure to the maximal point problem in two dimensions results in efficient worst-case update and query times of $O(\log n)$. However, since their data structure is designed specifically for range queries, it has no scheme for representing the m-contour. There appears to be no efficient way to list out the m-contour.

The dynamic contour search tree must be balanced in order to get the $O(\log n)$ search time. We use the balancing scheme of the red–black binary search trees of Tarjan [3] since they use the minimum number of rotations to keep the tree balanced. Each rotation in the dynamic contour search tree requires $O(\log n)$ time to update the search information at the three nodes possibly affected by the rotation. Thus it is crucial to use the red–black binary search trees which have $O(1)$ rotations per update in order to achieve the $O(\log n)$ time for insertions.

2. Definitions and Searching Operations

The m-contour of a set of points S in the plane is composed of the maximal points of S. These maximal points form a "staircase" in which all the remaining points of S lie to the left and below the staircase. An example set of points, and the staircase for them, are given in Fig. 1(a). The *inside region* of the m-contour is the region dominated by the m-contour. That is, a point p_i is inside the m-contour
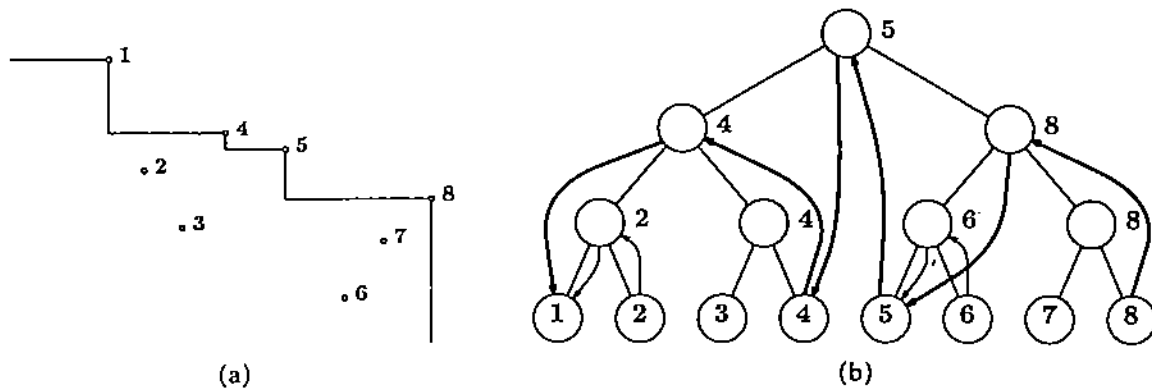
Fig. 1. (a) The m-contour. (b) The corresponding dynamic contour search tree without the *next_down* pointers shown. The embedded m-contour is emphasized.

if and only if there is a maximal point p_j such that $p_i < p_j$. The *outside region* of the m-contour is the region that is not dominated by the m-contour.

The data structure we propose to maintain the m-contour dynamically is a balanced binary tree with all the points of the set S sorted by the x-coordinate and stored in the leaves of the tree. Each leaf node contains three fields: x, y, and *transfer*. The internal nodes of the tree are used to search for points and to represent the m-contour. Each internal node v contains six fields: *right*, *left*, *max_x*, *max_y*, *next_down*, and *next_in_contour*. *Right*(v) is the right child of internal node v and similarly *left*(v) is the left child of v. *Max_x*(v) points to the leaf containing the point with the largest x-coordinate of all the points in v's subtree. We use *max_x*(v) in the search for a point via its x-coordinate. *Max_y*(v) points to the leaf containing the point with the largest y-coordinate of all the points in v's subtree. We use *max_y*(v) when updating certain information after an insertion or deletion. *Next_down*(v) of node v points to the closest descendant node w that has a *max_y*$(right(w)) = max\_y(right(v))$. If no such w exists, then *next_down*(v) points to the leaf node w that *max_y*$(right(v))$ points to. We use *next_down*(v) in Sections 3 and 4 to reset *transfer* pointers in $O(1)$ time.

Next_in_contour(v) for internal node v contains a pointer to the descendant leaf in the tree which represents the nearest point that is to the left and above of *max_y*$(right(v))$ in the plane. If *max_y*$(right(v))$ points to the leaf for $p_i = (x_i, y_i)$ then *next_in_contour*(v) will point to some point $p_j = (x_j, y_j)$ which has the largest x-coordinate x_j such that $x_j < x_i$ and $y_j > y_i$. It is possible for several internal nodes to have the same *max_y*$(right(v))$ values. Of these nodes, only the node v closest to the root will have its *next_in_contour*(v) pointer set to the next contour point; the others will set their *next_in_contour* pointers to *NULL*. *Transfer*(w), where w is a leaf containing the point $p_i = (x_i, y_i)$, is a pointer to the highest ancestor v that has a $y(max\_y(right(v)))$ value equal to y_i. *Transfer*(w) together with the *next_down* pointers of all the nodes v whose *max_y*$(right(v)) = p_i$ form a circularly linked list in the tree.

A dynamic contour search tree for the set of points in Fig. 1(a) is given in Fig. 1(b). Downward arrows are *next_in_contour* pointers, upward arrows are *transfer* pointers, and a label beside an interior node v is the *max_y*$(right(v))$
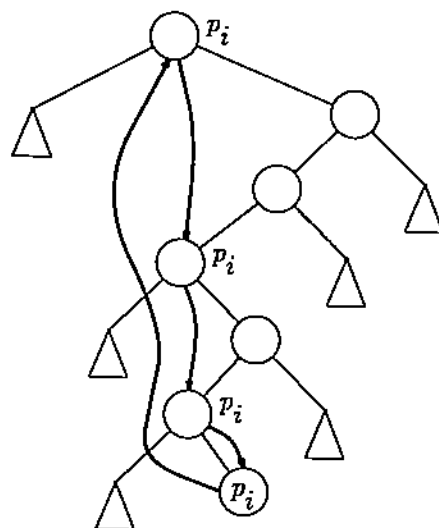
Fig. 2. A circular linked list for p_i formed by a *transfer* pointer and three *next_down* pointers.

value. *Next_down* pointers are not shown. Figure 2 shows the circularly linked list of the *transfer* pointer for the point p_i and the *next_down* pointers of the nodes v whose $max\_y(right(v)) = p_i$.

The m-contour can easily be produced from this data structure by traversing a linked list consisting of alternating *next_in_contour* and *transfer* pointers. Note that the rightmost point on the m-contour will always be a point $p_i = (x_i, y_i)$ with the largest x-coordinate. Starting from p_i, we can trace through the m-contour that is represented in the tree by alternately following *transfer* and *next_in_contour* pointers. This list is emboldened in Fig. 1(b).

Theorem 2.1. *The points of the m-contour can be listed using the dynamic contour search tree in $O(m)$ time where m is the number of maximal points.*

Proof. The rightmost leaf w in the search tree contains the rightmost point $p_i = (x_i, y_i)$ on the m-contour since the leaves are sorted by x-coordinates and x_i is the largest x-value of any point. *Transfer*(w) points to its highest ancestor v that has a $y(max\_y(right(v)))$ value of y_i. No point in v's right subtree has a larger y-value. To find the next maximal point, examine *next_in_contour*(v). It points to the leaf u containing the point $p_k = (x_k, y_k)$ such that $x_k = (max\ x_j$ such that $x_j < x_i$ and $y_j > y_i)$. Thus by starting with the rightmost leaf and tracing the *transfer* and *next_in_contour* pointers, the m-contour can be listed in $O(m)$ time.   □

An arbitrary point $p_j = (x_j, y_j) \notin S$ can be tested easily to see whether p_j lies inside or outside of the m-contour. Any point that lies inside the m-contour is dominated by some maximal point. A simple search of the dynamic contour search tree would find a maximal point p_i that dominates p_j if p_j lies inside the m-contour. Otherwise, p_j will be found to lie outside the m-contour. Initialize v to the root. Let $p_i = (x_i, y_i)$ be the point at the leaf pointed to by $max\_y(right(v))$. Figure 3 shows four quadrants with respect to p_i. The point p_j must lie in one

IV   I

p_i
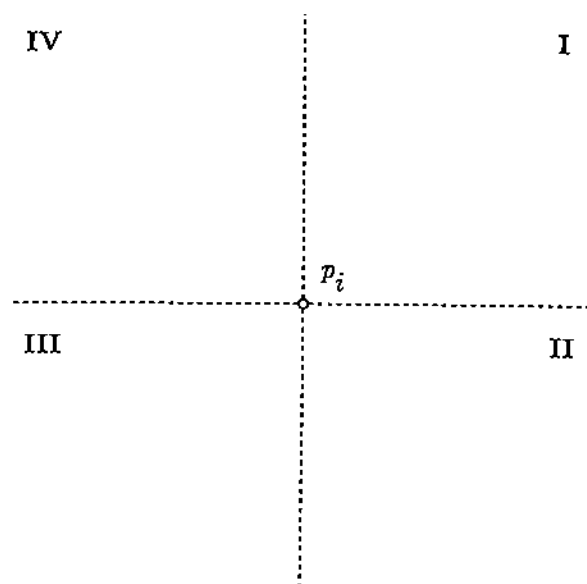
III   II

Fig. 3. Quadrants with respect to the point p_i.

of these quadrants. If $x_j > x_i$ and $y_j > y_i$ then the point p_j lies outside of the m-contour (quadrant I), as we shall show. If $x_j < x_i$ and $y_j < y_i$ then the point p_j is dominated by p_i and lies inside the m-contour (quadrant III). If $x_j < x_i$ and $y_j > y_i$ then recursively apply the search to v's left subtree (quadrant IV). If $x_j > x_i$ and $y_j < y_i$ then recursively apply the search to v's right subtree (quadrant II). If a leaf is reached, then p_j was not dominated by any point and thus lies outside the m-contour.

Theorem 2.2. *Using the above procedure, an arbitrary point p_j will be found to lie inside or outside the m-contour in $O(\log n)$ time.*

Proof. Suppose p_j lies outside the m-contour. No point dominates p_j so either p_j is found to lie clearly outside of the m-contour (quadrant I) or a leaf is reached during the search indicating that no points dominate p_j.

Suppose p_j lies inside the m-contour. Then there is at least one point on the m-contour which dominates p_j. Let p_k be the maximal point to the right of p_j that has the largest y-value. There is an internal node w in the dynamic contour search tree with its $max\_y(right(w))$ equal to p_k such that the leaf position in the tree that would be reached in searching for p_j is in w's left subtree. Thus, there is an internal node along the search path from the root to p_j whose $max\_y(right(w))$ value would determine that p_j is inside the m-contour. Figure 4(a) shows the point p_j dominated by three maximal points and Fig. 4(b) shows the node w in the dynamic contour search tree.

Starting at the root of the tree, the search proceeds toward the node w. Let v be the current internal node in our search and let $p_i = (x_i, y_i)$ be the point at the leaf pointed to by $max\_y(right(v))$. If p_i dominates p_j then our search stops. Otherwise, if p_i is a contour point, then w is either in v's left subtree if $x_k < x_i$ or in v's right subtree if $x_k > x_i$ and the search continues accordingly toward w.
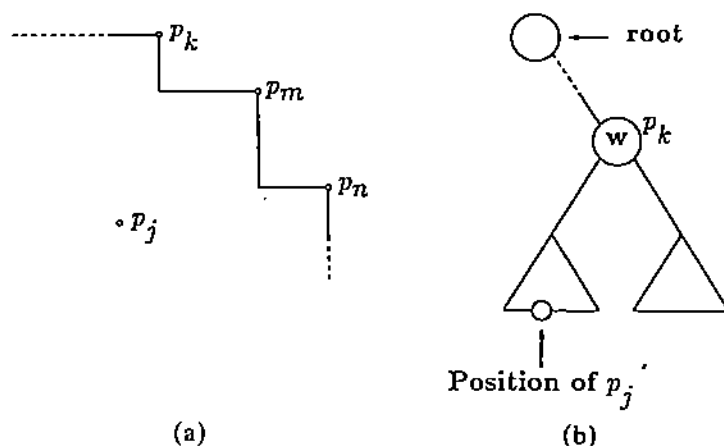
Fig. 4. (a) The m-contour. (b) The corresponding dynamic contour search tree.

If p_i is not a contour point, then there are no contour points in v's right subtree. The node w is not in v's right subtree since if it was then p_i would be a contour point. Thus w is in v's left subtree and p_j's position in the tree is to the left of p_i. The search continues via the left child of v. It cannot happen that $p_i < p_j$, so the point p_j cannot lie in quadrant I with respect to p_i. Thus either the search halts before reaching w, or it halts at w.

The search clearly takes $O(\log n)$ time.   □

3. Insertions

Given the dynamic contour search tree for a set of n points, suppose we want to add an additional point p_j to the set. The point p_j will be inserted into the dynamic contour search tree. If p_j is outside the current m-contour, then p_j will become a new point on the m-contour, otherwise p_j will become an interior point. To maintain balance in the dynamic contour search tree after each insertion, we use the balancing scheme of the red–black binary search trees of Tarjan [3]. Each rotation performed during balancing will take $O(\log n)$ time to update the fields of the node involved. Thus is it crucial to use the red–black trees since they have $O(1)$ rotations per insertion.

To insert the point $p_j = (x_j, y_j)$ into the dynamic contour search tree, we use the insertion procedure for red–black trees, and update other fields as the insertion is performed. First search for the leaf position of p_j, a leaf node containing the point $p_i = (x_i, y_i)$. On the way down to find the insertion position, update $max\_x(v)$ and $max\_y(v)$ for each node v on the path by comparing it with p_j. Also on the way down, identify each node v such that $max\_y(right(v))$ will change to point to p_j. Each of these will have its $next\_in\_contour$ pointer set to $NULL$ except for the one closest to the root.

For this node v, we compute its $next\_in\_contour$ pointer as follows. The $next\_in\_contour$ value of v will point to a leaf in v's left subtree. It cannot point to a leaf in v's right subtree because then it would be pointing to a y-value which is larger than $y(max\_y(right(v)))$. To calculate v's $next\_in\_contour$ pointer,

traverse down a path in v's left subtree. At each internal node w, if $y(max\_y(right(w)))$ is greater than $y(max\_y(right(v)))$, then traverse down w's right subtree, else traverse down w's left subtree. If a leaf containing the point p_k is reached such that $y_k > y(max\_y(right(v)))$, then set v's *next_in_contour* pointer to point to this leaf. Otherwise set v's *next_in_contour* pointer to *NULL* because there is no point in the subtree that is above and to the left in the plane.

As mentioned above, there may be several nodes v such that the value $max\_y(right(v))$ will change to point to p_j. If $max\_y(right(v))$ was previously pointing to p_k, then *transfer*(w) for the leaf node w containing the point p_k is pointing to v as v must be the node closest to the root with a $max\_y(right(v)) = p_k$. *Next_down*(v) points to the next node u closest to the root with a $max\_y(right(u)) = p_k$, so when $max\_y(right(v))$ changes to point to p_j, *next_down*(v) is used to update *transfer*(w) quickly. *Next_down*(v) will be updated as soon as the node it will point to is reached.

For nodes on the path between the root and the new leaf for p_j, their *next_in_contour* pointer could possibly change to p_j if p_j is in quadrant IV with respect to them and p_j is closer than the point they are currently pointing to. A simple comparison at each node on the way down the path should tell whether the *next_in_contour* pointer should be changed to point to p_j.

Upon reaching p_i, replace p_i by an internal node and two leaves, one for p_j and one for p_i. The *transfer* pointer of the leaf p_j should be set to point to the internal node v which is the closest internal node to the root with a $max\_y(right(v))$ value of p_j.

Once p_j is inserted in a leaf, the insertion path is retraced updating the balance information. This will halt at some node with $O(1)$ rotations performed. These rotations may affect three internal nodes that would need to recalculate their fields at a cost of $O(\log n)$.

In Fig. 5(a) the change to the m-contour is shown when point p_9 is inserted into the set of points from Fig. 1. The corresponding dynamic contour search tree is shown in Fig. 5(b).
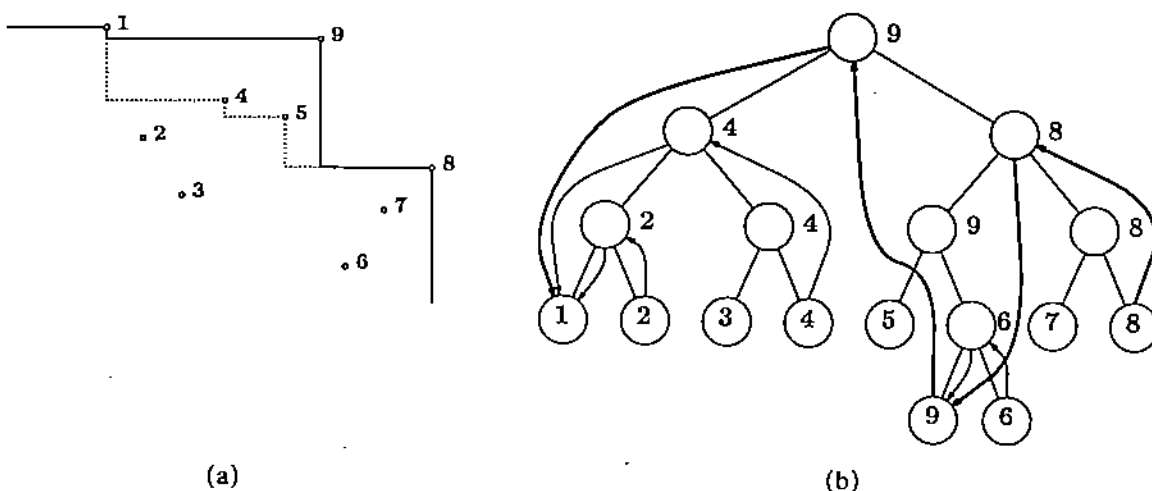

(a)   (b)

Fig. 5. (a) The m-contour after inserting point 9. (b) The corresponding dynamic contour search tree with the embedded m-contour emphasized.

Theorem 3.1.   *Inserting a point p into a search tree with n leaves takes $O(\log n)$ time.*

Proof. To add the new point p_j involves a search from the root to a leaf which takes $O(\log n)$ time. At each internal node on the way down the path, updating max_x and max_y takes $O(1)$ time. There is one $next\_in\_contour$ pointer which must be calculated at a cost of $O(\log n)$ time. The remaining $next\_in\_contour$ pointers that change are set to *NULL* or are set to point to the leaf p_j and thus can be calculated in $O(1)$ time. *Transfer* and $next\_down$ pointers are updated in $O(1)$ time by adding or deleting a node from a linked list.

Balance information at each internal node can be calculated in $O(1)$ time. Each rotation affects at most three internal nodes. Their fields can be recalculated in $O(\log n)$ time. Since there are $O(1)$ rotations per insertion, the total time for balancing is $O(\log n)$ time. Thus the time to insert the point p_j into the search tree takes $O(\log n)$ time.   □

Corollary 3.1.   *The search tree for a set of n points can be built in $O(n \log n)$ time.*

Proof. Apply the insertion procedure above for each point.   □

4. Deletions

Given the dynamic contour search tree for a set of n points, suppose we want to delete a point p_j from the set. If p_j was a point on the m-contour then the new m-contour will be updated when p_j is deleted by updating the dynamic contour search tree. Even if p_j was not on the m-contour, some work may be needed to update the dynamic contour search tree.

To delete the point $p_j = (x_j, y_j)$ from the dynamic contour search tree, we use the deletion procedure for the red–black trees. First find the leaf w containing the point p_j. Leaf w will be the child of some internal node v. Let u be the other child of v and let $p(v)$ be the parent of v. Remove w and v and let u become a child of $p(v)$. Now proceed back up the path, updating values at the internal nodes. Updating $max\_x(v)$ and $max\_y(v)$ at each internal node v along the path takes $O(1)$ time per node. Updating *transfer* and $next\_down$ pointers also takes $O(1)$ time per node along the path.

Some $next\_in\_contour$ pointers may need to be recalculated. If any internal node v had $y(max\_y(right(v))) = y_j$ then this node would receive a new $max\_y(right(v))$ value. Nodes with $y(max\_y(right(v))) = y_j$ must lie along the search path from the root to p_j. If the $y(max\_y(right(v)))$ value of a node v changes, then the $next\_in\_contour$ pointer of the node could possibly change, so its $next\_in\_contour$ pointer must be recalculated. In addition, if any $next\_in\_contour$ pointer was pointing at p_j, then these $next\_in\_contour$ pointers need to be recalculated because p_j will be deleted. Only nodes along the path from the root to p_j can have $next\_in\_contour$ pointers pointing to p_j since $next\_in\_contour$ pointers must point to a descendant leaf. $Next\_in\_contour$ pointers should be recalculated on the way back up to the root.
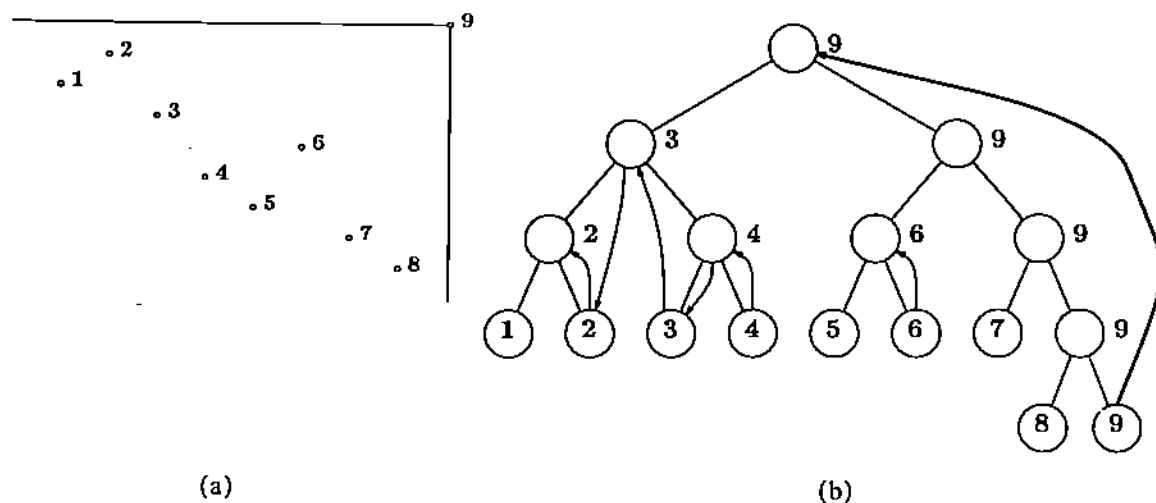
(a)   (b)

Fig. 6. (a) The m-contour before the deletion of point 9. (b) The corresponding contour search tree before deleting point 9.

Also on the way back up the path, update the balance information using the balancing scheme of the red–black binary search trees. There may be $O(1)$ rotations needed to balance the tree. At most three internal nodes will be affected by a rotation and need their fields recalculated. After performing a rotation, continue on up the path updating the remaining nodes on the path.

Figure 6(a) and (b) gives a set of points and an associated dynamic contour search tree. Figure 7(a) and (b) gives the set of points and the associated dynamic contour search tree after point p_9 is deleted.

Theorem 4.1. *Deleting a point from a search tree with n leaves takes $O((\log n)^2)$ time.*

Proof. To delete the leaf p_j from the dynamic contour search tree involves a search from the root to the leaf p_j which takes $O(\log n)$ time. After removing the point p_j, the same path must be traversed back up to the root, updating the fields at each internal node along the way. At most $O(\log n)$ nodes will need to
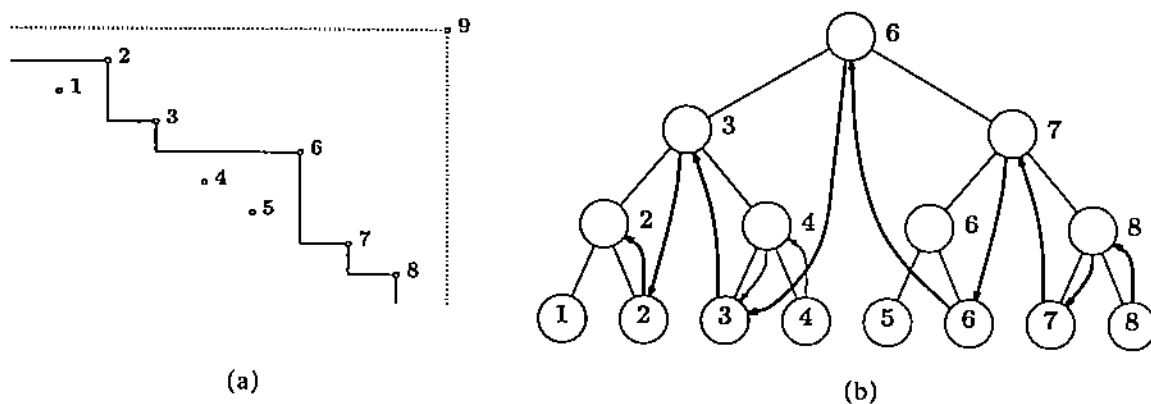

(a)   (b)

Fig. 7. (a) The m-contour after the deletion of point 9. (b) The corresponding contour search tree after deleting point 9.

recalculate *next_in_contour* pointers. It takes $O(\log n)$ time to recalculate one *next_in_contour* pointer so the total time in recalculating the *next_in_contour* pointers is $O((\log n)^2)$ time. Updating *max_x*, *max_y*, *transfer*, *next_down*, and the balance information takes $O(1)$ time at each node. There will be $O(1)$ rotations and it will take $O(\log n)$ time to recalculate the fields of the nodes affected by the rotation. Thus the total time to delete point p_j from the search structure is $O((\log n)^2)$ time.   □

References

1. H. T. Kung, F. Luccio, and F. P. Preparata, On finding the maxima of a set of vectors, *J. Assoc. Comput. Mach.* **22**, 4 (1975), 469–476.
2. M. H. Overmars and J. L. van Leeuwen, Maintenance of configurations in the plane, *J. Comp. System Sci.* **23** (1981), 166–204.
3. R. E. Tarjan, Updating a balanced search tree in $O(1)$ rotations, *Inform. Process. Lett.* **16** (1983), 253–257.
4. D. E. Willard and G. S. Lueker, Adding range restriction capability to dynamic data structures, *J. Assoc. Comput. Mach.* **32**, 3 (1985), 597–617.