Dynamic Convex Hulls under Window-Sliding Updates*

Haitao Wang
Kahlert School of Computing, University of Utah, Salt Lake City, Utah 84112, USA

Abstract

We consider the problem of dynamically maintaining the convex hull of a set S of points in the plane under the following special sequence of insertions and deletions (called *window-sliding updates*): insert a point to the right of all points of S and delete the leftmost point of S. We propose an $O(|S|)$-space data structure that can handle each update in $O(1)$ amortized time, such that standard binary-search-based queries on the convex hull of S can be answered in $O(\log h)$ time, where h is the number of vertices of the convex hull of S, and the convex hull itself can be output in $O(h)$ time.

Keywords and phrases Dynamic convex hulls, data structures, insertions, deletions, sliding window

Digital Object Identifier 10.57717/cgt.v4i1.53

Acknowledgements The author would like to thank Joseph S.B. Mitchell for posing the question at WADS 2023 about whether the $O(\log n)$ query time in the preliminary version of the paper can be improved to $O(\log h)$, and thank Michael T. Goodrich for suggesting the idea of using finger search trees to achieve this. This research was supported in part by NSF under Grants CCF-2005323 and CCF-2300356.

1 Introduction

As a fundamental structure in computational geometry, the convex hull $CH(S)$ of a set S of points in the plane has been well studied in the literature. Several $O(n \log n)$ time algorithms are known for computing $CH(S)$, e.g., see [5, 27], where $n = |S|$, and the time matches the $\Omega(n \log n)$ lower bound. Output-sensitive $O(n \log h)$ time algorithms have also been given [10, 21], where h is the number of vertices of $CH(S)$. If the points of S are already sorted, e.g., by x-coordinate, then $CH(S)$ can be computed in $O(n)$ time by Graham's scan [15].

Due to a wide range of applications, the problem of dynamically maintaining $CH(S)$, where points can be inserted and/or deleted from S, has also been extensively studied. Overmars and van Leeuwen [25] proposed an $O(n)$-space data structure that can support each insertion and deletion in $O(\log^2 n)$ worst-case time; Preparata and Vitter [28] gave a simpler method with the same performance. If only insertions are involved, then the approach of Preparata [26] can support each insertion in $O(\log n)$ worst-case time. For deletions only, Hershberger and Suri's method [18] can support each update in $O(\log n)$ amortized time. If both insertions and deletions are allowed, a breakthrough was given by Chan [11], who developed a data structure of linear space that can support each update in $O(\log^{1+\epsilon} n)$ amortized time, for an arbitrarily small $\epsilon > 0$. Subsequently, Brodal and Jacob [7], and independently Kaplan et al. [20] reduced the update time to $O(\log n \log\log n)$. Finally, Brodal and Jacob [8] achieved $O(\log n)$ amortized time performance for each update, with $O(n)$ space.

Under certain special situations, better and simpler results are also known. If the insertions and deletions are given offline, the data structure of Hershberger and Suri [19]

* A preliminary version of this paper appears in *Proceedings of the 18th Algorithms and Data Structures Symposium (WADS 2023)*.

can support $O(\log n)$ amortized time update. Schwarzkopf [29] and Mulmuley [23] presented algorithms to support each update in $O(\log n)$ expected time if the sequence of updates is random in a certain sense. In addition, Friedman, Hershberger, and Snoeyink [14] considered the problem of maintaining the convex hull of a simple path P such that vertices are allowed to be inserted and deleted from P at both ends of P, and they gave a linear space data structure that can support each update in $O(\log |P|)$ amortized time (more precisely, $O(1)$ amortized time for each deletion and $O(\log |P|)$ amortized time for each insertion). There are also other special dynamic settings for convex hulls, e.g., [13, 17].

In most applications, the reason to maintaining $CH(S)$ is to perform queries on it efficiently. As discussed in Chan [12], there are usually two types of queries, depending on whether the query is *decomposable* [4], i.e., if S is partitioned into two subsets, then the answer to the query for S can be obtained in constant time from the answers of the query for the two subsets. For example, the following queries are decomposable: find the most extreme vertex of $CH(S)$ along a query direction; decide whether a query line intersects $CH(S)$; find the two common tangents to $CH(S)$ from a query point outside $CH(S)$, while the following *line-intersection* queries are not decomposable: find the intersection of $CH(S)$ with a vertical query line or more generally an arbitrary query line. It seems that the decomposable queries are easier to deal with. Indeed, most of the above mentioned data structures can handle the decomposable queries in $O(\log n)$ time each. However, this is not the case for the non-decomposable queries. For example, none of the data structures of [8, 7, 11, 14, 20] can support $O(\log n)$-time non-decomposable queries. More specifically, Chan's data structure [11] can be modified to support certain non-decomposable queries (such as the above line-intersection queries) in $O(\log^{3/2} n)$ time but the amortized update time also increases to $O(\log^{3/2} n)$. Later Chan [12] gave a randomized algorithm that can support certain non-decomposable queries in expected $O(\log^{1+\epsilon} n)$ time, for an arbitrarily small $\epsilon > 0$, and the amortized update time is also $O(\log^{1+\epsilon} n)$.

Another operation on $CH(S)$ is to output it explicitly, ideally in $O(h)$ time. To achieve this, one usually has to maintain $CH(S)$ explicitly in the data structure, e.g., in [18, 25]. Unfortunately, most other data structures are not able to do so, e.g., [8, 7, 11, 14, 19, 20, 28], although they can output $CH(S)$ in a slightly slower $O(h \log n)$ time. In particular, Bus and Buzer [9] considered this operation for maintaining the convex hull of a simple path P such that vertices are allowed to be inserted and deleted from P at both ends of P, i.e., in the same problem setting as in [14]. Based on a modification of the algorithm in [22], they achieved $O(1)$ amortized update time such that $CH(S)$ can be output explicitly in $O(h)$ time [9]. However, no other queries on $CH(S)$ were considered in [9].

1.1 Our results

We consider a special sequence of insertions and deletions: the point inserted by an insertion must be to the right of all points of the current set S, and a deletion always happens to the leftmost point of the current set S. Equivalently, we may consider the points of S contained in a window bounded by two vertical lines that are moving rightwards (but the window width is not fixed), so we call them *window-sliding updates*.

To solve the problem, one can apply any previous data structure for arbitrary point updates. For example, the method in [8] can handle each update in $O(\log n)$ amortized time and answer each decomposable query in $O(\log n)$ time. Alternatively, if we connect all points of S from left to right by line segments, then we can obtain a simple path whose ends are the leftmost and rightmost points of S, respectively. Therefore, the data structure of Friedman et al. [14] can be applied to handle each update in $O(\log n)$ amortized time

and support each decomposable query in $O(\log n)$ time. In addition, although the data structure in [18] is particularly for deletions only, Hershberger and Suri [18] indicated that their method also works for the window-sliding updates, in which case each update (insertion and deletion) takes $O(\log n)$ amortized time. Further, the data structure [18] can support binary-search-based queries in $O(\log n)$ time and report $CH(S)$ in $O(h)$ time.

In this paper, we provide a new data structure for the window-sliding updates. Our data structure uses $O(n)$ space and can handle each update in $O(1)$ amortized time. Standard binary-search-based queries on $CH(S)$ can be answered in $O(\log h)$ time each.[1] More specifically, our data structure maintains a balanced binary search tree of height $O(\log h)$ storing the vertices of $CH(S)$; using the tree, binary-search-based queries on $CH(S)$ can be answered in $O(\log h)$ time in a standard way. Further, after each update, the convex hull $CH(S)$ can be output explicitly in $O(h)$ time. Specifically, the following theorem summarizes our result.

▶ **Theorem 1.** *We can dynamically maintain the convex hull $CH(S)$ of a set S of points in the plane to support each window-sliding update (i.e., either insert a point to the right of all points of S or delete the leftmost point of S) in $O(1)$ amortized time, such that the following operations on $CH(S)$ can be performed. Let $n = |S|$ and h be the number of vertices of $CH(S)$ right before each operation.*

1. *The convex hull $CH(S)$ can be explicitly output in $O(h)$ time.*
2. *Given two vertical lines, the vertices of $CH(S)$ between the vertical lines can be output in order along the boundary of $CH(S)$ in $O(k + \log h)$ time, where k is the number of vertices of $CH(S)$ between the two vertical lines.*
3. *Each of the following queries can be answered in $O(\log h)$ time.*
   a. *Given a query direction, find the most extreme point of S along the direction.*
   b. *Given a query line, decide whether the line intersects $CH(S)$.*
   c. *Given a query point outside $CH(S)$, find the two tangents from the point to $CH(S)$.*
   d. *Given a query line, find its intersection with $CH(S)$, or equivalently, find the edges of $CH(S)$ intersecting the line.*
   e. *Given a query point, decide whether the point is in $CH(S)$.*
   f. *Given a convex polygon of $O(h)$ vertices (represented in any data structure that supports binary search), decide whether it intersects $CH(S)$, and if not, find their common tangents (both outer and inner).*

Comparing to all previous work, albeit on a very special sequence of updates, our result is particularly interesting due to the $O(1)$ amortized update time as well as its simplicity.

Applications. Although the updates in our problem are quite special, the problem still finds applications. For example, Becker et al. [3] considered the problem of finding two rectangles of minimum total area to enclose a set of n rectangles in the plane. They gave an algorithm of $O(n \log n)$ time and $O(n \log \log n)$ space. Their algorithm has a subproblem of processing a dynamic set of points to answer queries of Type 3a of Theorem 1 with respect to window-sliding updates (see Section 3.2 [3]). The subproblem is solved using subpath convex hull query data structure in [16], which costs $O(n \log \log n)$ space. Using Theorem 1, we can reduce the space of the algorithm to $O(n)$ while the runtime is still $O(n \log n)$. Note that Wang [31] recently improved the space of the result of [16] to $O(n)$, which also leads to

[1] In the preliminary version of the paper at WADS 2023, the query time was $O(\log n)$. In this version, we improve the query time to $O(\log h)$.

an $O(n)$ space solution for the algorithm of [3]. However, the approach of Wang [31] is much more complicated.

Becker et al. [2] extended their work above and studied the problem of enclosing a set of simple polygons using two rectangles of minimum total area. They gave an algorithm of $O(n\alpha(n)\log n)$ time and $O(n\log\log n)$ space, where n is the total number of vertices of all polygons and $\alpha(n)$ is the inverse Ackermann function. The algorithm has a similar subproblem as above (see Section 4.2 [2]). Similarly, our result can reduce the space of their algorithm to $O(n)$ while the runtime is still $O(n\alpha(n)\log n)$.

We believe that our result may find other applications that remain to be discovered.

Outline. After introducing notation in Section 2, we will prove Theorem 1 gradually as follows. First, in Section 3, we give a data structure for a special problem setting. Then we extend our techniques to the general problem setting in Section 4. The data structures in Section 3 and 4 can only perform the first operation in Theorem 1 (i.e., output $CH(S)$), we will enhance the data structure in Section 5 so that other operations can be handled. Section 6 concludes with some remarks.

2 Preliminaries

Let $\mathcal{U}(S)$ denote the upper hull of $CH(S)$. We will focus on maintaining $\mathcal{U}(S)$, and the lower hull can be treated likewise. The data structure for both hulls together will achieve Theorem 1.

For any two points p and q in the plane, we say that p is to the *left* of q if the x-coordinate of p is smaller than or equal to that of q. Similarly, we can define "to the right of", "above", and "below". We add "strictly" in front of them to indicate that the tie case does not happen. For example, p is strictly below q if the y-coordinate of p is smaller than that of q.

For a line segment s and a point p, we say that p is *vertically below* s if the vertical line through p intersects s at a point above p ($p \in s$ is possible). For any two line segments s_1 and s_2, we say that s_1 is vertically below s_2 if both endpoints of s_1 are vertically below s_2.

Suppose $\mathcal{L}$ is a sequence of points and p and q are two points of $\mathcal{L}$. We will adhere to the convention that a subsequence of $\mathcal{L}$ *between* p *and* q includes both p and q, but a subsequence of $\mathcal{L}$ *strictly* between p and q does not include either one.

For ease of exposition, we make a general position assumption that no two points of S have the same x-coordinate and no three points are collinear.

3 A special problem setting with a partition line

In this section we consider a special problem setting. Let $L = \{p_1, p_2, \ldots, p_n\}$ (resp., $R = \{q_1, q_2, \ldots, q_n\}$) be a set of n points sorted by increasing x-coordinate, such that all points of L are strictly to the left of a known vertical line ℓ and all points of R are strictly to the right of ℓ. Let $L \cup R = \{p_1, ..., p_n, q_1, ..., q_n\}$ denote the left-to-right sorted sequence of all the points. Throughout, we maintain a consecutive subsequence of $L \cup R$, denoted S. Initially $S = L$. Each subsequent operation is either an insertion or a deletion. An insertion adds the leftmost point of R that is not already in S, and a deletion removes the leftmost point of L that is in S. Overall, there are a total of at most n insertions and n deletions, which may be interspersed arbitrarily. More specifically, $S = \{p_i, p_{i+1}, ..., p_n, q_1, q_2, ..., q_j\}$ for some i and j. If the next operation is an insertion (implying that $j < n$), the point q_{j+1} is added at the right end of S. If the next operation is a deletion (implying that $i < n$), the

point p_i is removed from the left end of S. Let $\mathcal{U}(S)$ denote the upper convex hull of the current sequence S. We want to maintain $\mathcal{U}(S)$ subject to insertions and deletions.

Our result is a data structure that can support each update in $O(1)$ amortized time, and after each update we can output $\mathcal{U}(S)$ in $O(|\mathcal{U}(S)|)$ time. The data structure can be built in $O(n)$ time on $S = L$ initially. Note that the set L is given offline because $S = L$ initially, but points of R are given online. We will extend the techniques to the general problem setting in Section 4, and the data structure will be enhanced in Section 5 so that other operations on $CH(S)$ can be handled.

3.1 Initialization

Initially, we construct the data structure on L, as follows. We run Graham's scan to process points of L leftwards from p_n to p_1. Each vertex $p_i \in L$ is associated with a stack $Q(p_i)$, which is empty initially. Each vertex p_i also has two pointers $l(p_i)$ and $r(p_i)$, pointing to its left and right neighbors respectively if p_i is a vertex of the current upper hull. Suppose we are processing a point p_i. Then, the upper hull of $p_{i+1}, p_{i+2}, \ldots, p_n$ has already been maintained by a doubly linked list with p_{i+1} as the head. To process p_i, we run Graham's scan to find a vertex p_j (with $j > i$) of the current upper hull such that $\overline{p_i p_j}$ is an edge of the new upper hull. Then, we push p_i into the stack $Q(p_j)$, and set $l(p_j) = p_i$ and $r(p_i) = p_j$. The algorithm is done after p_1 is processed.

The stacks essentially maintain the left neighbors of the vertices of the historical upper hulls so that when some points are deleted in the future, we can traverse leftwards from any vertex on the current upper hull after those deletions. More specifically, if p_i is a vertex on the current upper hull, then the vertex at the top of $Q(p_i)$ is the left neighbor of p_i on the upper hull. In addition, notice that once the right neighbor pointer $r(p_i)$ is set during processing p_i, it will never be changed. Hence, in the future if p_i becomes a vertex of the current upper hull after some deletions, $r(p_i)$ is the right neighbor of p_i on the current upper hull. Therefore, we do not need another stack to keep the right neighbor of p_i.

The above builds our data structure for $\mathcal{U}(S)$ initially when $S = L$. In what follows, we discuss the general situation when S contains both points of L and R. Let $S_1 = S \cap L$ and $S_2 = S \cap R$. The data structure described above is used for maintaining $\mathcal{U}(S_1)$. For S_2, we only use a doubly linked list to store its upper hull $\mathcal{U}(S_2)$, and the stacks are not needed. In addition, we explicitly maintain the common tangent $\overline{t_1 t_2}$ of the two upper hulls $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$, where t_1 and t_2 are the tangent points on $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$, respectively. We also maintain the leftmost and rightmost points of S. This completes the description of our data structure for S.

Using the data structure we can output $\mathcal{U}(S)$ in $O(|\mathcal{U}(S)|)$ time as follows. Starting from the leftmost vertex of S_1, we follow the right neighbor pointers until we reach t_1, and then we output $\overline{t_1 t_2}$. Finally, we traverse $\mathcal{U}(S_2)$ from t_2 rightwards until the rightmost vertex. In the following, we discuss how to handle insertions and deletions.

3.2 Insertions

Suppose a point $q_j \in R$ is inserted into S. If $j = 1$, then this is the first insertion. We set $t_2 = q_1$ and find t_1 on $\mathcal{U}(S_1)$ by traversing it leftwards from p_n (i.e., by Graham's scan). This takes $O(n)$ time but happens only once in the entire algorithm (for processing all $2n$ insertions and deletions), so the amortized cost for the insertion of q_1 is $O(1)$. In the following we consider the general case $j > 1$.

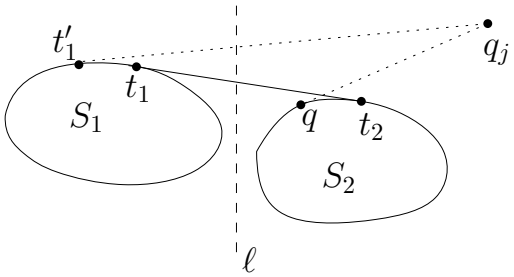

Figure 1 Illustrating the insertion of q_j.

We first update $\mathcal{U}(S_2)$ by Graham's scan. This procedure takes $O(n)$ time in total for all n insertions, and thus $O(1)$ amortized time per insertion. Let q be the vertex such that $\overline{qq_j}$ is the edge of the new hull $\mathcal{U}(S_2)$ (e.g., see Fig. 1). If q is strictly to the right of t_2, or if $q = t_2$ and $\overline{t_1t_2}$ and $\overline{t_2q_j}$ make a right turn at t_2, then $\overline{t_1t_2}$ is still the common tangent and we are done with the insertion. Otherwise, we update $t_2 = q_j$ and find the new t_1 by traversing $\mathcal{U}(S_1)$ leftwards from the current t_1, and we call it the *insertion-type tangent searching procedure*, which takes $O(1 + k)$ time, with k equal to the number of vertices on $\mathcal{U}(S_1)$ strictly between the original t_1 and the new t_1 (and we say that those vertices are *involved* in the procedure). The following lemma implies that the total time of this procedure in the entire algorithm is $O(n)$, and thus the amortized cost is $O(1)$.

▶ **Lemma 2.** *Each point of $L \cup R$ can be involved in the insertion-type tangent searching procedure at most once in the entire algorithm.*

Proof. We use t_1 to refer to the original tangent point and use t_1' to refer to the new one (e.g., see Fig. 1). Let p be any point on $\mathcal{U}(S_1)$ strictly between t_1' and t_1. Then, we have the following *observation*: all points of L strictly to the right of p must be *vertically below* the segment $\overline{pq_j}$.

Assume to the contrary that p is involved again in the procedure when another point q_k with $k > j$ is inserted. Let t_1'' be the tangent point on $\mathcal{U}(S_1)$ right before q_k is inserted. As p is involved in the procedure, p must be on the current hull $\mathcal{U}(S_1)$ and strictly to the left of t_1''. According to the above observation, t_1'' is vertically below the segment $\overline{pq_j}$ (and $t_1'' \notin \overline{pq_j}$ due to the general position assumption). Since q_j is in the current set S_2 and p is in the current set S_1, and t_1'' is vertically below the segment $\overline{pq_j}$, t_1'' cannot be the left tangent point of the common tangent of $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$, incurring a contradiction. ◀

3.3 Deletions

Suppose a point $p_i \in L$ is deleted from S_1. If $i = n$, then this is the last deletion. In this case, we only need to maintain $\mathcal{U}(S_2)$ for insertions only in the future, which can be done by Graham's scan. In the following, we assume that $i < n$.

Note that p_i must be the leftmost vertex of the current hull $\mathcal{U}(S_1)$. Let $p = r(p_i)$ (i.e., p is the right neighbor of p_i on $\mathcal{U}(S_1)$). According to our data structure, p_i is at the top of the stack $Q(p)$. We pop p_i out of $Q(p)$. If $p_i \neq t_1$, then p_i is strictly to the left of t_1 and $\overline{t_1t_2}$ is still the common tangent of the new S_1 and S_2, and thus we are done with the deletion. Otherwise, we find the new tangent by simultaneously traversing on $\mathcal{U}(S_1)$ leftwards from p and traversing on $\mathcal{U}(S_2)$ leftwards from t_2 (e.g., see Fig. 2). Specifically, we first check whether $\overline{pt_2}$ is tangent to $\mathcal{U}(S_1)$ at p. If not, then we move p leftwards on the new $\mathcal{U}(S_1)$ until $\overline{pt_2}$ is tangent to $\mathcal{U}(S_1)$ at p. Then, we check whether $\overline{pt_2}$ is tangent to $\mathcal{U}(S_2)$ at t_2.

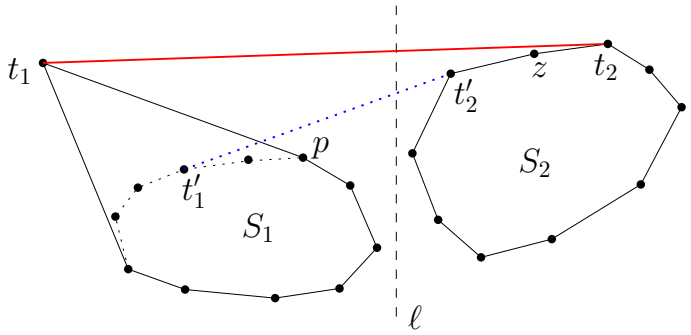

Figure 2 Illustrating the deletion of p_i where $p_i = t_1$. $\overline{t'_1 t'_2}$ are the new tangent of $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$ after p_i is deleted.

If not, then we move t_2 leftwards on $\mathcal{U}(S_2)$ until $\overline{pt_2}$ is tangent to $\mathcal{U}(S_2)$ at t_2. If the new $\overline{pt_2}$ is not tangent to $\mathcal{U}(S_1)$ at p, then we move p leftwards again. We repeat the procedure until the updated $\overline{pt_2}$ is tangent to $\mathcal{U}(S_1)$ at p and also tangent to $\mathcal{U}(S_2)$ at t_2. Note that both p and t_2 are monotonically moving leftwards on $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$, respectively. Note also that traversing leftwards on $\mathcal{U}(S_1)$ is achieved by using the stacks associated with the vertices while traversing on $\mathcal{U}(S_2)$ is done by using the doubly-linked list that stores $\mathcal{U}(S_2)$. We call the above the *deletion-type tangent searching procedure*, which takes $O(1 + k_1 + k_2)$ time, where k_1 is the number of points on $\mathcal{U}(S_1)$ strictly between p and the new tangent point t_1, i.e., the final position of p after the algorithm finishes (we say that these points are involved in the procedure), and k_2 is the number of points on $\mathcal{U}(S_2)$ strictly between the original t_2 and the new t_2 (we say that these points are involved in the procedure). The following lemma implies that the total time of this procedure in the entire algorithm is $O(n)$, and thus the amortized cost is $O(1)$.

▶ **Lemma 3.** *Each point of $L \cup R$ can be involved in the deletion-type tangent searching procedure at most once in the entire algorithm.*

Proof. Let p' be a vertex on $\mathcal{U}(S_1)$ strictly between p and the new tangent point t_1 (which is t'_1 in Fig. 2). We argue that p' cannot be involved in the same procedure again in the future. Indeed, p' was not a vertex of $\mathcal{U}(S_1)$ before p_i is deleted because it was vertically below the edge $\overline{p_i p}$ of $\mathcal{U}(S_1)$. After p_i is deleted, since p' is involved in the procedure, p' must be a vertex of $\mathcal{U}(S_1)$. Further, p' will always be a vertex of $\mathcal{U}(S_1)$ until it is deleted. Hence, p' will never be involved in the procedure again in the future (because to do so p' cannot be a vertex of $\mathcal{U}(S_1)$ right before the deletion).

Let t'_2 be the new t_2 and t_2 be the original one. Let z be a vertex on $\mathcal{U}(S_2)$ strictly between the two (e.g., see Fig. 2). We argue that z will never be involved in the same procedure again in the future. Let t''_2 be the tangent point on $\mathcal{U}(S_2)$ right before an update in the future. We assume inductively that t''_2 is either at t'_2 or strictly left of t'_2. Note that initially we have $t''_2 = t'_2$ and therefore the assumption holds. Depending on whether the update is a deletion or an insertion, there are two cases.

- If the update is a deletion, then according to our deletion algorithm, every point of R involved in the deletion-type tangent searching procedure must be to the left of t''_2. As z is strictly to the right of t'_2 and thus is strictly to the right of t''_2, z cannot be involved in the procedure during this deletion operation. Further, according to the deletion algorithm, after the deletion, the tangent point on $\mathcal{U}(S_2)$ is either still at t''_2 or strictly to the left of t''_2; this establishes the assumption.

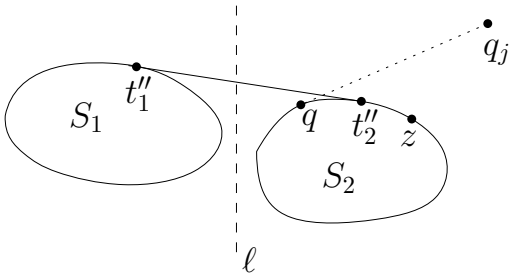

Figure 3 Illustrating the case where a point q_j is inserted for the proof of Lemma 3.

- If the update is an insertion, say, inserting a point p_j, then according to our insertion algorithm, we first find the edge $\overline{qq_j}$ of the new hull $\mathcal{U}(S_2)$ (e.g., see Fig. 3). Let t_1'' be current tangent point on $\mathcal{U}(S_1)$, i.e., $\overline{t_1''t_2''}$ is the common tangent.
  According to our insertion algorithm, if q is strictly to the right of t_2'', or if $q = t_2''$ and $\overline{t_1''t_2''}$ and $\overline{t_2''q_j}$ make a right turn at t_2'', then $\overline{t_1''t_2''}$ is still the common tangent; in this case, since t_2'' is still the tangent point, the assumption still holds. Otherwise, q is t_2'' or strictly to the left of t_2'', and thus all points of S_2 strictly to the right of t_2'' must be vertically below $\overline{qq_j}$. As z is strictly to the right of t_2'', z is vertically below $\overline{qq_j}$, which implies that z cannot be on the upper hull $\mathcal{U}(S_2)$. Since only insertions will happen to S_2 in the future, z can never be on the upper hull $\mathcal{U}(S_2)$ in the future. Because only points on $\mathcal{U}(S_2)$ can be involved in the deletion-type tangent searching procedure, z will never be involved in the procedure in the future.

The above proves that z will never be involved in the deletion-type tangent searching procedure again in the future. The lemma thus follows. ◀

As a summary, in the special problem setting, we can perform each insertion and deletion in $O(1)$ amortized time, and after each update, the upper hull $\mathcal{U}(S)$ can be output in $|\mathcal{U}(S)|$ time.

4 The general problem setting

In this section, we extend our algorithm given in Section 3 to the general problem setting without the restriction on the existence of the partition line ℓ. Specifically, we want to maintain the upper hull $\mathcal{U}(S)$ under window-sliding updates, with $S = \emptyset$ initially. We will show that each update can be handled in $O(1)$ amortized time and after each update $\mathcal{U}(S)$ can be output in $O(|\mathcal{U}(S)|)$ time. The high-level framework is somewhat similar to the idea of implementing a queue using two stacks/lists [24]. We will enhance the data structure in Section 5 so that it can handle other operations on $CH(S)$.

During the course of processing updates, we maintain a vertical line ℓ that will move rightwards. At any moment, ℓ plays the same role as in the problem setting in Section 3. In addition, ℓ always contains a point of S. Let S_1 be the subset of S that lies to the left of ℓ (including the point on ℓ), and $S_2 = S \setminus S_1$. For S_1, we use the same data structure as before to maintain $\mathcal{U}(S_1)$, i.e., a doubly linked list for vertices of $\mathcal{U}(S_1)$ and a stack associated with each point of S_1, and we call it *the list-stack data structure*. For S_2, as before, we only use a doubly linked list to store the vertices of $\mathcal{U}(S_2)$. Note that $S_2 = \emptyset$ is possible. If $S_2 \neq \emptyset$, we also maintain the common tangent $\overline{t_1t_2}$ of $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$, with $t_1 \in \mathcal{U}(S_1)$ and $t_2 \in \mathcal{U}(S_2)$. We can output the upper hull $\mathcal{U}(S)$ in $O(|\mathcal{U}(S)|)$ time as before.

For each $i \geq 1$, let p_i denote the i-th inserted point. Let U denote the universal set of all points p_i that will be inserted. Note that points of U are given online and we only use U for the reference purpose in our discussion (our algorithm has no information about U beforehand). We assume that S initially consists of two points p_1 and p_2. We let ℓ pass through p_1. According to the above definitions, we have $S_1 = \{p_1\}$, $S_2 = \{p_2\}$, $t_1 = p_1$, and $t_2 = p_2$. We assume that during the course of processing updates S always has at least two points, since otherwise we could restart the algorithm from this initial stage. Next, we discuss how to process updates.

4.1 Deletions

Suppose a point p_i is deleted. If p_i is not the only point of S_1, then we do the same processing as before in Section 3. We briefly discuss it here. If $p_i \neq t_1$, then we pop p_i out of the stack $Q(p)$ of p, where $p = r(p_i)$. In this case, we do not need to update $\overline{t_1t_2}$. Otherwise, we also need to update $\overline{t_1t_2}$, by the deletion-type tangent searching procedure as before. Lemma 3 is still applicable here (replacing $L \cup R$ with U), so the procedure takes $O(1)$ amortized time.

If p_i is the only point in S_1, then we do the following. We move ℓ to the rightmost point of S_2, and thus, the new set S_1 consists of all points in the old set S_2 while the new S_2 becomes $\emptyset$. Next, we build the list-stack data structure for S_1 by running Graham's scan leftwards from its rightmost point, which takes $O(|S_1|)$ time. We call it the *left-hull construction procedure*. The following lemma implies that the amortized cost of the procedure is $O(1)$.

▶ **Lemma 4.** *Every point of U is involved in the left-hull construction procedure at most once in the entire algorithm.*

Proof. Consider a point p involved in the procedure. Notice that the procedure will not happen again before all points of the current set S_1 are deleted. Since p is in the current set S_1, p will not be involved in the same procedure again in the future. ◀

4.2 Insertions

Suppose a point p_j is inserted. We first update $\mathcal{U}(S_2)$ using Graham's scan, and we call it the *right-hull updating procedure*. After that, p_j becomes the rightmost vertex of the new $\mathcal{U}(S_2)$. The procedure takes $O(1 + k)$ time, where k is the number of vertices that were removed from the old $\mathcal{U}(S_2)$ (we say that these points are *involved* in the procedure). By the standard Graham's scan, the amortized cost of the procedure is $O(1)$. Note that although the line ℓ may move rightwards, we can still use the same analysis as the standard Graham's scan. Indeed, according to our algorithm for processing deletions discussed above, once ℓ moves rightwards, all points in S_2 will be in the new set S_1 and thus will never be involved in the right-hull updating procedure again in the future.

After $\mathcal{U}(S_2)$ is updated as above, we check whether the upper tangent $\overline{t_1t_2}$ needs to be updated, and if yes (in particular, if $S_2 = \emptyset$ before the insertion), we run an insertion-type tangent searching procedure to find the new tangent in the same way as before in Section 3. Lemma 2 still applies (replacing $L \cup R$ with U), and thus the procedure takes $O(1)$ amortized time. This finishes the processing of the insertion, whose amortized cost is $O(1)$.

As a summary, in the general problem setting, we can perform each insertion and deletion in $O(1)$ amortized time, and after each update, the upper hull $\mathcal{U}(S)$ can be output in $|\mathcal{U}(S)|$ time.

5 Convex hull queries

In this section, we enhance the data structure described in Section 4 so that it can support $O(\log h)$ time convex hull queries as stated in Theorem 1, where h is the number of vertices of the convex hull $CH(S)$.

The main idea is to use a red-black tree T (or other types of finger search trees [6]) to store the vertices of $\mathcal{U}(S)$ in the left-to-right order with two "fingers" (i.e., two pointers) at the leftmost and rightmost leaves of T, respectively.[2] During the course of the algorithm, T will be updated with insertions and deletions. We will show that each insertion/deletion must happen at either the leftmost or the rightmost leaf, which takes $O(1)$ amortized time with the help of the two fingers [30]. Using T, we can easily answer binary-search-based queries on $CH(S)$ in $O(\log h)$ time each.

Unless otherwise stated, we follow the same notation as those in Section 4. In addition to the data structure for storing S_1 and S_2 described in Section 4, we assume that T stores the vertices of $\mathcal{U}(S)$ in the left-to-right order. In what follows, we discuss how to update T during the algorithm in Section 4.

5.1 Deletions

Consider the deletion of a point p_i. As before, depending on whether p_i is the only point of S_1, there are two cases.

p_i is the only point of S_1. If p_i is the only point in S_1, then according to our algorithm, we need to perform the left-hull construction procedure on the points $\{p_{i+1}, p_{i+2}, \ldots, p_j\}$, where p_j is the rightmost point of S_2, after which the above set of points becomes the new S_1. In addition to the algorithm described in Section 4.1, we update T as follows.

Recall that the left-hull construction procedure process vertices of $\{p_{i+1}, p_{i+2}, \ldots, p_j\}$ from right to left. Suppose a vertex p_k is being processed (i.e., points $p_{k+1}, \ldots, p_j$ have already been processed and $\mathcal{U}(S)$ is thus the upper hull of these points; T store the vertices of $\mathcal{U}(S)$). Then, using Graham's scan, we check whether the leftmost vertex p_g of $\mathcal{U}(S)$ needs to be removed due to p_k. If yes, we delete p_g from $\mathcal{U}(S)$; note that p_g must be at the leftmost leaf of T. We continue this process until the leftmost vertex p_g of the current $\mathcal{U}(S)$ should not be removed due to p_i. Then, we insert p_k into T as the leftmost leaf. Since each insertion and deletion of T only happens at its leftmost leaf and we already have a finger there, each such update of T takes $O(1)$ amortized time. Further, due to Lemma 4, the amortized cost of the left-hull construction is still $O(1)$.

p_i is not the only point of S_1. Suppose p_i is not the only point in S_1. Then we preform the same processing as before in Section 4 and update T accordingly. The details are discussed below. There are two subcases depending on whether $p_i = t_1$.

If $p_i \neq t_1$, recall that our algorithm pops p_i out of the stack $Q(p)$, where $p = r(p_i)$. In this case, the common tangent $\overline{t_1 t_2}$ does not change. We update T as follows. We first delete p_i from T. Observe that p_i must be the leftmost leaf of T. Therefore this deletion costs $O(1)$ amortized time. Let $\mathcal{U}(S)$ refer to the upper hull after p_i is deleted. Let S' denote the set of vertices of $\mathcal{U}(S)$ that were not on $\mathcal{U}(S)$ before p_i is deleted. It is not difficult to see that

[2] In the preliminary version of the paper at WADS 2023, we used interval trees and achieved $O(\log n)$ query time. In this version, following Michael T. Goodrich's suggestion, we resort to finger search trees instead, which help reduce the query time to $O(\log h)$ and also simplifies the overall algorithm.

points of S' are exactly the vertices of $\mathcal{U}(S)$ strictly to the left of p. Starting from p, these vertices can be accessed from right to left using the list-stack data structure. We insert these points into T in the right-to-left order. Observe that each newly inserted point becomes the leftmost leaf of the new tree (note that before these insertions, p was at the leftmost leaf of T). Therefore, each such insertion on T takes $O(1)$ amortized time. We refer to this process as *deletion-type tree update procedure*; we say that points of S' are *involved* in this procedure. The following lemma implies that the amortized time of this procedure is $O(1)$ and so is the amortized time of deleting p_i.

▶ **Lemma 5.** *Each point can be involved in the deletion-type tree update procedure at most once in the entire algorithm.*

Proof. Let p' be any point of S'. We argue that p' cannot be involved in the deletion-type tree update procedure again in the future. Indeed, p' was involved in the procedure due to the deletion of p_i. Since $p_i \neq t_1$, p' must be a point in S_1. Right before p_i is deleted, p' was not on the upper hull of S_1. After p_i is deleted, since no points will be inserted into S_1 (before S_1 becomes empty), p' will always be on the upper hull of S_1 until it is deleted. This implies that p' cannot be involved in the procedure again in the future. ◀

If $p_i = t_1$ (see Fig. 2), recall that our algorithm finds the new tangent $\overline{t'_1t'_2}$ using the deletion-type tangent searching procedure as described before. We update T accordingly as follows. Observe that p_i, which is t_1, is the leftmost vertex of $\mathcal{U}(S)$ and thus is at the leftmost leaf of T. We delete p_i from T, which takes $O(1)$ amortized time. Then, we insert the points of $\mathcal{U}(S_2)$ from t_2 to t'_2 in this order so that each newly inserted points becomes the leftmost leaf of T, and thus each insertion takes $O(1)$ amortized time. Note that the above points can be accessed one by one using their left neighbor pointers, starting from t_2. Next, we insert t'_1 to T at the leftmost leaf and then insert the vertices of $\mathcal{U}(S_1)$ from t'_1 to its leftmost vertex (which is p_{i+1}) in this order; these points can be accessed one by one using the list-stack data structure, starting from t'_1. Again, each newly inserted point becomes the leftmost leaf of T and thus each such insertion takes $O(1)$ amortized time. Observe that each of the above newly inserted point p_k of T belongs to one of the following three cases: (1) p_k is a vertex of $\mathcal{U}(S_1)$ strictly between t'_1 and p_{i+1}; (2) p_k is is a vertex of $\mathcal{U}(S_2)$ strictly between t'_2 and t_2; (3) p_k is either t'_1 or t'_2. For Case (1), by the same analysis as in Lemma 5, p_k will not be involved in this process again in the future. For Case (2), since p_k is involved in the deletion-type tangent searching procedure (for computing $\overline{t'_1t'_2}$), by Lemma 3, p_k will not be involved in this process again in the future. Therefore, the amortized time of this process is $O(1)$, so is the amortized time of deleting p_i.

5.2 Insertions

Suppose we insert a point q_j. In addition to our processing as described in Section 4, we update T as follows. Recall that our algorithm first runs a right-hull updating procedure on $\mathcal{U}(S_2)$ by Graham's scan. During the procedure, if we need to remove a point q_k from $\mathcal{U}(S_2)$ and q_k is to the right of t_2 (including the case $q_k = t_2$), then q_k must be the rightmost vertex of the current $\mathcal{U}(S)$ and thus is the rightmost leaf of T. In this case, we delete q_k from T. If $q_k = t_2$, then the rightmost leaf of T becomes t_1. Recall that in this case our algorithm will perform an insertion-type tangent searching procedure to find the new tangent point t'_1 on $\mathcal{U}(S_1)$ (see Fig. 1). During the tangent searching procedure, we keep deleting the rightmost leaf of T until t'_1 is found, after which we insert q_j to T as the rightmost leaf. As such, each deleted point q_k of T as above belongs to one of the following three cases: (1) q_k is a vertex

of the original $\mathcal{U}(S_2)$ before q_j is inserted; (2) q_k is a vertex of $\mathcal{U}(S_1)$ strictly between t'_1 and t_1; (3) q_k is either t_2 or t_1. For Case (1), as discussed in Section 4.2, q_k will not be involved in this procedure again in the future. For Case (2), since q_k is involved in the insertion-type tangent searching procedure, by Lemma 2, q_k will not be involved in this procedure again in the future. Since each of the above update operations on T happens at either the leftmost or the rightmost leaf of T, the amortized cost of handing the insertion of q_j is $O(1)$.

6 Concluding remarks

We proposed a data structure to dynamically maintain the convex hull of a set of points in the plane under window-sliding updates. Although the updates are quite special, our result is particularly interesting because each update can be handled in constant amortized time and binary-search-based queries (both decomposable and non-decomposable) on the convex hull can be answered in logarithmic time each. In addition, the convex hull itself can be retrieved in time linear in its size. Also interesting is that our method is quite simple.

In the dual setting, the problem is equivalent to dynamically maintaining the upper and lower envelopes of a set S^* of lines in the plane under insertions and deletions. The window-sliding updates become inserting into S^* a line whose slope is larger than that of any line in S^* and deleting from S^* the line with the smallest slope. Alternatively, the lower envelope of S^* may be considered as a parametric heap [20]. A special but commonly used type of parametric heaps is kinetic heaps [1, 20]. With our result, we can handle each window-sliding insertion and deletion on kinetic heaps in $O(1)$ amortized time, and the "find-min" operation can also be performed in $O(1)$ amortized time, in the same way as those in [8, 20].

References

1 Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999. https://doi.org/10.1006/jagm.1998.0988.

2 Bruno Becker, Paolo G. Franciosa, Stephan Gschwind, Stefano Leonardi, Thomas Ohler, and Peter Widmayer. Enclosing a set of objects by two minimum area rectangles. *Journal of Algorithms*, 21:520–541, 1996. https://doi.org/10.1006/jagm.1996.0057.

3 Bruno Becker, Paolo G. Franciosa, Stephan Gschwind, Thomas Ohler, Gerald Thiem, and Peter Widmayer. An optimal algorithm for approximating a set of rectangles by two minimum area rectangles. In *Workshop on Computational Geometry*, pages 13–25, 1991. https://doi.org/10.1007/3-540-54891-2_2.

4 Jon L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8:244–251, 1979. https://doi.org/10.1016/0020-0190(79)90117-0.

5 Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational Geometry — Algorithms and Applications*. Springer-Verlag, Berlin, 3rd edition, 2008.

6 Gerth Stølting Brodal. Finger search trees. In *Handbook of Data Structures and Applications,* Dinesh P. Mehta and Sartaj Sahni (eds.). Chapman and Hall/CRC, 2004. https://cs.au.dk/~gerth/papers/finger05.pdf.

7 Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log \log n)$ update time. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 57–70, 2000. https://doi.org/10.1007/3-540-44985-X_7.

8 Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002. https://doi.org/10.1109/SFCS.2002.1181985.

9 Norbert Bus and Lilian Buzer. Dynamic convex hull for simple polygonal chains in constant amortized time per update. In *Proceedings of the 31st European Workshop on Computational*

Geometry (EuroCG), 2015. https://perso.esiee.fr/~busn/publications/2015_eurocg_dynamicConvexHull/eurocg2015_dynamicHull.pdf.

10 Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996. https://doi.org/10.1007/BF02712873.

11 Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmaic amortized time. *Journal of the ACM*, 48:1–12, 2001. https://doi.org/10.1145/363647.363652.

12 Timothy M. Chan. Three problems about dynamic convex hulls. *International Journal of Computational Geometry and Applications*, 22:341–364, 2012. https://doi.org/10.1142/S0218195912600096.

13 Timothy M. Chan, John Hershberger, and Simon Pratt. Two approaches to building time-windowed geometric data structures. *Algorithmica*, 81:3519–3533, 2019. https://doi.org/10.1007/s00453-019-00588-3.

14 Joseph Friedman, John Hershberger, and Jack Snoeyink. Efficiently planning compliant motion in the plane. *SIAM Journal on Computing*, 25:562–599, 1996. https://doi.org/10.1137/S0097539794263191.

15 Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972. https://doi.org/10.1016/0020-0190(78)90041-8.

16 Leonidas J. Guibas, John Hershberger, and Jack Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry and Applications*, 1(1):1–22, 1991. https://doi.org/10.1142/S0218195991000025.

17 John Hershberger and Jack Snoeyink. Cartographic line simplification and polygon CSG formula in $O(n \log^* n)$ time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998. https://doi.org/10.1016/S0925-7721(98)00027-3.

18 John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992. https://doi.org/10.1007/BF01994880.

19 John Hershberger and Subhash Suri. Offline maintenance of planar configurations. *Journal of Algorithms*, 21:453–475, 1996. https://doi.org/10.1006/jagm.1996.0054.

20 Haim Kaplan, Robert E. Tarjan, and Kostas Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 836–844, 2001. https://doi.org/10.5555/365411.365793.

21 David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986. https://doi.org/10.1137/0215021.

22 Avraham A. Melkman. On-line construction of the convex hull of a simple polygon. *Information Processing Letters*, 25:11–12, 1987. https://doi.org/10.1016/0020-0190(87)90086-X.

23 Ketan Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. In *Proceedings of the 32nd Annual Symposium of Foundations of Computer Science (FOCS)*, pages 180–196, 1991. https://doi.org/10.1145/109648.109662.

24 Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5:583–592, 1995. https://doi.org/10.1017/S0956796800001489.

25 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981. https://doi.org/10.1016/0022-0000(81)90012-X.

26 Franco P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22:402–405, 1979. https://doi.org/10.1145/359131.359132.

27 Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

28 Franco P. Preparata and Jeffrey S. Vitter. A simplified technique for hidden-line elimination in terrains. *International Journal of Computational Geometry and Applications,*, 3:167–181, 1993. https://doi.org/10.1142/S0218195993000117.

29 Otfried Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual Symposium of Foundations of Computer Science (FOCS)*, pages 197–206, 1991. https://doi.org/10.1109/SFCS.1991.185369.

30 Robert E. Tarjan and Christopher J. Van Wyk. An $o(n \log \log n)$ algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17:143–178, 1988. https://doi.org/10.1137/0217010.

31 Haitao Wang. Algorithms for subpath convex hull queries and ray-shooting among segments. In *Proceedings of the 36th International Symposium on Computational Geometry (SoCG)*, pages 69:1–69:14, 2020. https://doi.org/10.4230/LIPIcs.SoCG.2020.69.