Dynamic Convex Hulls Under Window-Sliding Updates

Haitao Wang(✉)

School of Computing, University of Utah, Salt Lake City, UT 84112, USA
haitao.wang@utah.edu

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 all standard binary-search-based queries on the convex hull of S can be answered in $O(\log |S|)$ time, and the convex hull itself can be output in time linear in its size.

Keywords: Convex hulls · Dynamic update · Window-sliding

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 [5,26], 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 [9,21], 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 [14].

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 [24] 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 [27] gave a simpler method with the same performance. If only insertions are involved, then the approach of Preparata [25] 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 [10], 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

This research was supported in part by NSF under Grants CCF-2005323 and CCF-2300356. A full version is available at http://arxiv.org/abs/2305.08055.

P. Morin and S. Suri (Eds.): WADS 2023, LNCS 14079, pp. 689–703, 2023.
https://doi.org/10.1007/978-3-031-38906-1_46

update time to $O(\log n \log\log n)$. Finally, Brodal and Jacob [6] 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] can support $O(\log n)$ amortized time update. Schwarzkopf [28] 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 et al. [13] 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., [12,17].

In most applications, the reason to maintaining $CH(S)$ is to perform queries on it efficiently. As discussed in Chan [11], 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 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 [6,7,10,13,20] can support $O(\log n)$-time non-decomposable queries. More specifically, Chan's data structure [10] can be modified to support each non-decomposable query in $O(\log^{3/2} n)$ time but the amortized update time also increases to $O(\log^{3/2} n)$. Later Chan [11] gave a randomized algorithm that can support each non-decomposable query 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 [18,24]. Unfortunately, most other data structures are not able to do so, e.g., [6,7,10,13,19,20,27], although they can output $CH(S)$ in a slightly slower $O(h \log n)$ time. In particular, Bus and Buzer [8] 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 [13]. 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 [8]. However, no other queries on $CH(S)$ were considered in [8].

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 the *window-sliding updates*.

To solve the problem, one can apply any previous data structure for arbitrary point updates. For example, the method in [6] 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. [13] 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, as the data structure [18] explicitly stores the edges of $CH(S)$ in a balanced binary search tree, it can support both decomposable and non-decomposable queries each in $O(\log n)$ time as well as output $CH(S)$ 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. All decomposable and non-decomposable queries on $CH(S)$ mentioned above can be answered in $O(\log n)$ time each. 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 $O(k + \log n)$ 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 $O(\log n)$ 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 (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. [2] 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 [2]). The subproblem is solved using subpath convex hull query data structure in [15], 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 [29] recently improved the space of the result of [15] to $O(n)$, which also leads to an $O(n)$ space solution for the algorithm of [2]. However, the approach of Wang [29] is much more complicated.

Becker et al. [1] 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 [1]). Similarly, our result can reduce the space of their algorithm to $O(n)$ while the runtime is still $O(n\alpha(n) \log n)$.

Outline. After introducing notation in Sect. 2, we will prove Theorem 1 gradually as follows. First, in Sect. 3, we give a data structure for a special problem setting. Then we extend our techniques to the general problem in Sect. 4. The data structures in Sect. 3 and 4 can only perform the first operation in Theorem 1 (i.e., output $CH(S)$), we will enhance the data structure in Sect. 5 so that other operations can be handled. Due to the space limit, all lemma proofs are omitted but can be found in the full paper.

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 follow 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. Specifically, 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. We want to maintain the upper convex hull $\mathcal{U}(S)$ of a consecutive subsequence S of $L \cup R = \{p_1, \ldots, p_n, q_1, \ldots, q_n\}$, i.e., $S = \{p_i, p_{i+1}, \ldots, p_n, q_1, q_2, \ldots, q_j\}$, with $S = L$ initially, such that a deletion will delete the leftmost point of S and an insertion will insert the point of R right after the last point of S. Further, deletions only happen to points of L, i.e., once p_n is deleted from S, no deletion will happen. Therefore, there are a total of n insertions and n deletions.

Our result is a data structure that supports 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 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 Sect. 4, and the data structure will be enhanced in Sect. 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 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 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 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.

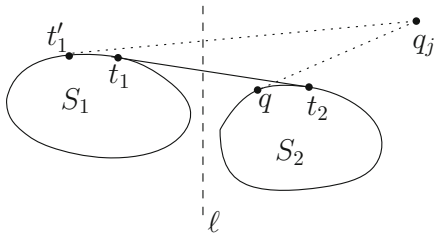

Fig. 1. Illustrating the insertion of q_j.

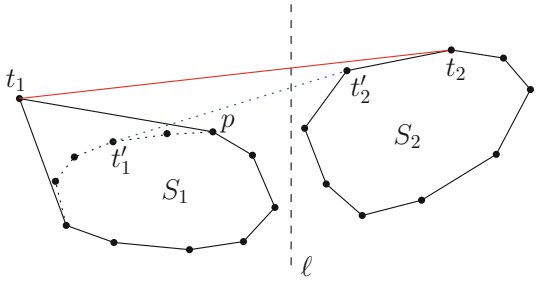

Fig. 2. Illustrating the deletion of p_i where $p_i = t_1$. $\overline{t_1' t_2'}$ are the new tangent of $\mathcal{U}(S_1)$ and $\mathcal{U}(S_2)$ after p_i is deleted.

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 $j > 1$.

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 the amortized cost of the procedure is $O(1)$.

Lemma 1. *Each point $p \in L \cup R$ can be involved in the insertion-type tangent searching procedure at most once in the entire algorithm.*

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 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. 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 amortized cost of the procedure is $O(1)$.

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

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 in Sect. 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. We will enhance the data structure in Sect. 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 Sect. 3. In addition, ℓ always contains a point of S. Let S_1 be the subset of S to the left of ℓ (including the point on ℓ), and $S_2 = S \backslash S_1$. 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 $\overline{t_1 t_2}$ 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.

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 Sect. 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 $\overline{t_1 t_2}$. Otherwise, we also need to update $\overline{t_1 t_2}$, by the deletion-type tangent searching procedure as before. Lemma 2 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 $|S_1|$ time. We call it the *left-hull construction procedure*. The following lemma implies that the amortized cost of the procedure is $O(1)$.

Lemma 3. *Every point of U is involved in the left-hull construction procedure at most once in the entire algorithm.*

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 got 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 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 Sect. 3. Lemma 1 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 Sect. 4 to support logarithmic time convex hull queries as stated in Theorem 1. This is done by incorporating an interval tree into our data structure. Below, we first describe the interval tree in Sect. 5.1. We incorporate the interval tree into our data structure in Sect. 5.2. The data structure can support $O(\log |U|)$ time queries, where U is the universe of all points that will be inserted, under the assumption that the size $|U|$ is known initially when $S = \emptyset$. We finally lift the assumption in Sect. 5.3 and also reduce the query time to $O(\log n)$, with $n = |S|$.

5.1 The Interval Tree

We borrow an idea from Guibas et al. [15] and use interval trees. We build a complete binary search tree T whose leaves from left to right correspond to the indices from 1 to $|U|$. So the height of T is $O(\log |U|)$. For each leaf, if it corresponds to index i, then we assign i as the *index* of the leaf. For each internal node v, if i is the index of the rightmost leaf in its left subtree, then we assign $i + 1/2$ as the index of v, although it is not an integer. In this way, the sorted order of the indices of all nodes of T follows the symmetric order of the nodes. For a line segment $\overline{p_ip_j}$ connecting two points p_i and p_j of U, we say that the segment *spans* a node v, if the index of v is in the range $[i, j]$. Comparing to the interval tree in [15], which is defined with respect to the actual x-coordinates of the points, our tree is more abstract because it is defined on indices only.

Consider the set S maintained by our algorithm, which is a subset of U. We can store its upper hull $\mathcal{U}(S)$ in T as follows [15]. For each edge of $\mathcal{U}(S)$ connecting two vertices p_i and p_j, we store $\overline{p_ip_j}$ at the lowest common ancestor of leaves i and j in T (i.e., the highest node spanned by $\overline{p_ip_j}$; e.g., see Fig. 3). By also storing the lower hull of S in T as above, all queries on $CH(S)$ as specified in Theorem 1(3) can be answered in $O(\log |U|)$ time, by following a path from the root to a leaf [15] (the main idea is that the hull edge spanning a node v is stored either at v or at one of its ancestors, and only at most two ancestor edges closest to v need to be remembered during the search in T). In fact, our problem is slightly more complex because we need to store not only the edges of $\mathcal{U}(S)$ but also some historical hull edges. In the following, we incorporate this interval tree T into our data structure.

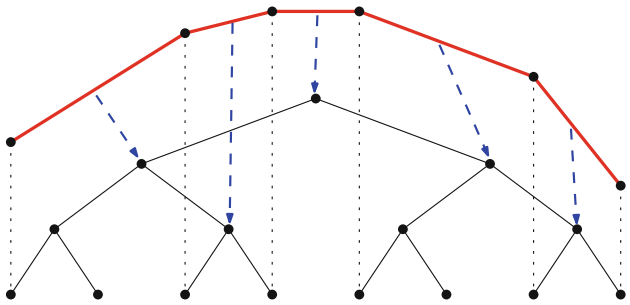

Fig. 3. Illustrating an upper hull and an interval tree that stores all hull edges: the (blue) dashed lines with arrows show where edges are stored. (Color figure online)

5.2 The Enhanced Data Structure

Unless otherwise stated, we follow the same notation as that in Sect. 4.

In addition to the data structure for storing S_1 and S_2 described in Sect. 4, we initially build the interval tree T. Then, we preprocess T in $O(|U|)$ time so that given any two nodes of the tree, their lowest common ancestor can be found in $O(1)$ time [3,16]. The amortized cost of this preprocessing is $O(1)$, for there will be $|U|$ insertions. In addition, we associate each node v of T with a stack, which is $\emptyset$ initially. For any two points p_i and p_j of U, we use $lca(p_i, p_j)$ to refer to the lowest common ancestor of the two leaves of T corresponding to the indices of the two points, respectively.

During the left-hull construction procedure for computing the list-stack data structure for S_1, we make the following changes. Whenever a vertex p_i is processed and an edge (p_i, p_j) is added as an edge to the current upper hull, in addition to setting $l(p_j) = p_i$, $r(p_i) = p_j$, and pushing p_i into the stack $Q(p_j)$ as before, we also push the edge (p_i, p_j) into the stack associated with the node $lca(p_i, p_j)$ of T. Thanks to the $O(1)$-time query performance of the lowest common ancestor query data structure [3,16], this change only adds constant time to

each step in the original algorithm, and thus does not affect the time complexity asymptotically.

We make similar changes in the right-hull updating procedure for computing $\mathcal{U}(S_2)$. Specifically, assume that we are inserting a point p_j. We run Graham's scan on the vertices of $\mathcal{U}(S_2)$ from right to left. Suppose that we are scanning an edge $\overline{pp'}$ of $\mathcal{U}(S_2)$ and we find that it needs to be removed from the current upper hull. Then, we also pop $\overline{pp'}$ out of the stack associated with the node $lca(p, p')$ of T. After the Graham's scan is done, let p_k be the vertex that connects to p_j in the new hull (i.e., $\overline{p_k p_j}$ is the new edge). Then, we push $\overline{p_k p_j}$ into the stack associated with the node $lca(p_k, p_j)$ of T. Again, these changes do not change the time complexity of the algorithm asymptotically.

In addition, we store the common tangent $\overline{t_1 t_2}$ at the top of the stack associated with the node $lca(t_1, t_2)$ of T.

Deletions and Insertions. 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.

1. If p_i is not the only point in S_1, then we do the same processing as before with the following changes. First, we pop the common tangent $\overline{t_1 t_2}$ out of the stack associated with the node $lca(t_1, t_2)$ of T. Second, when we pop p_i out of the stack $Q(p)$ with $p = r(p_i)$, we also pop the edge $\overline{p_i p}$ out of the stack at the node $lca(p_i, p)$ in T. Third, we push the new tangent (t_1, t_2) into the stack of the new $lca(t_1, t_2)$ of T. Note that the push and the pop operations of the common tangent $\overline{t_1 t_2}$ are always needed following the above order even if it does not change (otherwise, assume that we do not perform the pop operation in the first step, then in the second step when we attempt to pop $\overline{p_i p}$ out of the stack at the node $lca(p_i, p)$, $\overline{p_i p}$ may not be at the top of the stack because $\overline{t_1 t_2}$ may be at the top of the same stack).
2. If p_i is the only point in S_1, then according to our algorithm, we need to run 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. Here we make the following changes. First, for each edge $\overline{p_k p_{k'}}$ of $\mathcal{U}(S_2)$, we pop $\overline{p_k p_{k'}}$ out of the stack associated with the node $lca(p_k, p_{k'})$ of T. Then, we run the left-hull construction procedure with the changes discussed above.

Consider the insertion of a point p_j. We first pop the tangent $\overline{t_1 t_2}$ out of the stack at the node $lca(t_1, t_2)$. Then, we run the right-hull updating procedure with the changes discussed above. Finally, we push the new tangent $\overline{t_1 t_2}$ (which may be the same as the original one) into the stack at the node $lca(t_1, t_2)$.

As discussed above, due to the $O(1)$ query time of the lowest common ancestor data structure, the amortized time of each insertion/deletion is still $O(1)$.

Queries. Next we discuss how to answer convex hull queries using the interval tree T. One difference between our interval tree T and that used in [15] is that there are stacks associated with the nodes of T, which may store edges not on $\mathcal{U}(S)$. Therefore, we cannot directly use the query algorithm in [15]. Rather, we need to make sure that non-hull edges in the stacks will not give us trouble. To this end, we first prove the following lemma (which further leads to Corollary 1).

Lemma 4. *Suppose a stack associated with a node of T contains more than one edge, and let e_1 and e_2 be any two edges in the stack such that e_1 is above e_2 in the stack (i.e., e_1 is stored closer to the top of the stack than e_2 is). Then, e_2 is vertically below e_1.*

Corollary 1. *The stack at any node of T can store at most one edge of $\mathcal{U}(S)$, and if it stores such an edge, then the edge must be at the top of the stack.*

Lemma 5 provides a foundation that guarantees the convex hull queries on $CH(S)$ can be answered in $O(\log |U|)$ time each. It resembles Lemma 4.1 in [15], but its proof relies on Lemma 4 to handle the non-hull edges in stacks.

Lemma 5. *We can walk in $O(\log |U|)$ time from the root to any leaf in T, at each node knowing which edge of $\mathcal{U}(S)$ spans the current node, or if none, to which side $\mathcal{U}(S)$ lies.*

With the algorithm in Lemma 5 as a "template", the convex hulls queries in Theorem 1(3) can all be answered in $O(\log |U|)$ time each. For example, consider the following query: Given a vertical line l, find the edge of $\mathcal{U}(S)$ that intersects l. If l is strictly to the left of the leftmost point of S or strictly to the right of the rightmost point of S, then the answer to the query is $\emptyset$. Otherwise, the query algorithm starts from the root of T. For each node v, we apply the algorithm in Lemma 5 to determine the edge e of $\mathcal{U}(S)$ that spans v (if there is no such a spanning edge, then the algorithm in Lemma 5 can determine to which side of v $\mathcal{U}(S)$ lies, and we proceed on the child of v on that side). If e intersects l, then we report e. Otherwise, if l is to the left (resp., right) of e, then we proceed on the left (resp., right) child of v. The time of the query algorithm is $O(\log |U|)$. Other queries can be handled similarly (see [15] for some details).

We can still output $\mathcal{U}(S)$ in $O(|\mathcal{U}(S)|)$ time as before. Hence, the performance of the first operation in Theorem 1 can be achieved. For the second operation in Theorem 1, we consider the upper hull first. Let l_1 and l_2 be the two query lines. Without loss of generality, we assume that both lines intersect $\mathcal{U}(S)$ and l_1 is to the left of l_2. Using T, we first find in $O(\log |U|)$ time the edge $\overline{p_i p_j}$ of $\mathcal{U}(S)$ that intersects l_1. Without loss of generality, we assume that p_j is to the right of p_i. Then, following the right neighbor pointer $r(p_j)$ of p_j, we can output the vertices of $\mathcal{U}(S)$ to the left of l_2 in $O(1)$ time per vertex in a way similar to that for outputting vertices of $\mathcal{U}(S)$. The lower hull can be treated likewise. Hence, the second operation of Theorem 1 can be performed in $O(k + \log |U|)$ time.

5.3 A Further Improvement

We further improve our data structure to remove the assumption that $|U|$ is known initially and reduce the query time from $O(\log |U|)$ to $O(\log |S|)$. The idea is to still use an interval tree T, but instead of building it initially, we periodically rebuild it during processing updates so that the number of leaves of T is always no more than $4|S|$, and thus the height of T is $O(\log |S|)$, which guarantees the $O(\log |S|)$ query time. As will be seen later, our algorithm maintains an invariant

that whenever T is rebuilt, the number of its leaves is equal to $2|S|$, where S is the set when T is being rebuilt.

Initially when $S = \{p_1, p_2\}$, we build T with 4 leaves corresponding to indices $\{1, 2, 3, 4\}$. Let $T.max$ denote the index of the rightmost leaf of T. Initially $T.max = 4$. Let $|T|$ denote the number of leaves of T. During processing the updates, we keep track of the size of $|S|$ using a variable σ.

For each deletion, we decrease σ by one. If $\sigma = |T|/4$, we claim that at least σ deletions have happened since the current tree T was built. Indeed, let m be the size of S when T was just built. According to the algorithm invariant, $|T| = 2m$. Now that $\sigma = |T|/4$, at least $m - \sigma = |T|/4$ points have been deleted from S since T was built. The claim thus follows; let P denote the set of points in those deletions specified in the claim. We rebuild a tree T of $2 \cdot \sigma$ leaves corresponding to the indices $i, i+1, \ldots, i+2 \cdot \sigma - 1$, where i is the index of the leftmost point of S, and set $T.max = i + 2 \cdot \sigma - 1$. Note that rebuilding T also includes adding the edges in the data structure for S to the stacks of the nodes of the new T, which can be done by running the left-hull construction procedure on S_1 and running the right-hull updating procedure on S_2. The total time for building the new tree is $O(\sigma)$. We charge the $O(\sigma)$ time to the deletions of P, whose size is at least σ by the above claim. In this way, each deletion is charged at most once, and thus the amortized cost for rebuilding T during all deletions is $O(1)$.

Consider an insertion of a point p_j (i.e., this is the j-th insertion). We first increment σ by one. If $j = T.max$, then we rebuild a new tree T with $2 \cdot \sigma$ leaves corresponding to the indices $i, i+1, \ldots, i+2 \cdot \sigma - 1$, where i is the index of the leftmost point of S, and set $T.max = i + 2 \cdot \sigma - 1$. The time for building the new tree is $O(\sigma)$. We charge the time to the insertions of the points of the second half of S (i.e., the rightmost $\sigma/2$ points of S; note that $\sigma = |S|$). Lemma 6 implies that each insertion will be charged at most once in the entire algorithm and thus the amortized cost for rebuilding T during all insertions is $O(1)$.

Lemma 6. *No point in the second half of S was charged before.*

References

1. Becker, B., Franciosa, P., Gschwind, S., Leonardi, S., Ohler, T., Widmayer, P.: Enclosing a set of objects by two minimum area rectangles. J. Algorithms **21**, 520–541 (1996)
2. Becker, B., Franciosa, P.G., Gschwind, S., Ohler, T., Thiem, G., Widmayer, P.: An optimal algorithm for approximating a set of rectangles by two minimum area rectangles. In: Bieri, H., Noltemeier, H. (eds.) CG 1991. LNCS, vol. 553, pp. 13–25. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54891-2_2
3. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000). https://doi.org/10.1007/10719839_9
4. Bentley, J.: Decomposable searching problems. Inf. Process. Lett. **8**, 244–251 (1979)
5. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry – Algorithms and Applications, 3rd edn. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-77974-2

6. Brodal, G., Jacob, R.: Dynamic planar convex hull. In: Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS), pp. 617–626 (2002). Full version available at arXiv:1902.11169
7. Brodal, G.S., Jacob, R.: Dynamic Planar Convex Hull with Optimal Query Time and O(log $n \cdot \log \log n$) Update Time. In: SWAT 2000. LNCS, vol. 1851, pp. 57–70. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44985-X_7
8. Bus, N., Buzer, L.: 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)
9. Chan, T.: Optimal output-sensitive convex hull algorithms in two and three dimensions. Discrete Comput. Geom. **16**, 361–368 (1996). https://doi.org/10.1007/BF02712873
10. Chan, T.: Dynamic planar convex hull operations in near-logarithmic amortized time. J. ACM **48**, 1–12 (2001)
11. Chan, T.: Three problems about dynamic convex hulls. Int. J. Comput. Geom. Appl. **22**, 341–364 (2012)
12. Chan, T., Hershberger, J., Pratt, S.: Two approaches to building time-windowed geometric data structures. Algorithmica **81**, 3519–3533 (2019). https://doi.org/10.1007/s00453-019-00588-3
13. Friedman, J., Hershberger, J., Snoeyink, J.: Efficiently planning compliant motion in the plane. SIAM J. Comput. **25**, 562–599 (1996)
14. Graham, R.: An efficient algorithm for determining the convex hull of a finite planar set. Inf. Process. Lett. **1**, 132–133 (1972)
15. Guibas, L., Hershberger, J., Snoeyink, J.: Compact interval trees: a data structure for convex hulls. Int. J. Comput. Geom. Appl. **1**(1), 1–22 (1991)
16. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**, 338–355 (1984)
17. Hershberger, J., Snoeyink, J.: Cartographic line simplification and polygon CSG formula in $O(n \log^* n)$ time. Comput. Geom. Theory Appl. **11**, 175–185 (1998)
18. Hershberger, J., Suri, S.: Applications of a semi-dynamic convex hull algorithm. BIT **32**, 249–267 (1992). https://doi.org/10.1007/BF01994880
19. Hershberger, J., Suri, S.: Offline maintenance of planar configurations. J. Algorithms **21**, 453–475 (1996)
20. Kaplan, H., Tarjan, R., Tsioutsiouliklis, K.: Faster kinetic heaps and their use in broadcast scheduling. In: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 836–844 (2001)
21. Kirkpatrick, D., Seidel, R.: The ultimate planar convex hull algorithm? SIAM J. Comput. **15**, 287–299 (1986)
22. Melkman, A.: On-line construction of the convex hull of a simple polygon. Inf. Process. Lett. **25**, 11–12 (1987)
23. Mulmuley, K.: Randomized multidimensional search trees: lazy balancing and dynamic shuffling. In: Proceedings of the 32nd Annual Symposium of Foundations of Computer Science (FOCS), pp. 180–196 (1991)
24. Overmars, M., van Leeuwen, J.: Maintenance of configurations in the plane. J. Comput. Syst. Sci. **23**(2), 166–204 (1981)
25. Preparata, F.: An optimal real-time algorithm for planar convex hulls. Commun. ACM **22**, 402–405 (1979)
26. Preparata, F., Shamos, M.: Computational Geometry. Springer, New York (1985). https://doi.org/10.1007/978-1-4612-1098-6

27. Preparata, F.P., Vitter, J.S.: A simplified technique for hidden-line elimination in terrains. In: Finkel, A., Jantzen, M. (eds.) STACS 1992. LNCS, vol. 577, pp. 133–146. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55210-3_179
28. Schwarzkopf, O.: Dynamic maintenance of geometric structures made easy. In: Proceedings of the 32nd Annual Symposium of Foundations of Computer Science (FOCS), pp. 197–206 (1991)
29. Wang, H.: Algorithms for subpath convex hull queries and ray-shooting among segments. In: Proceedings of the 36th International Symposium on Computational Geometry (SoCG), pp. 69:1–69:14 (2020)