Dynamic planar convex hull

Riko Jacob

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Dynamic planar convex hull

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
PhD Degree

by
Riko Jacob
February 2002

Abstract

We determine the computational complexity of the dynamic convex hull problem in the planar case. We present a data structure that maintains a finite set of n points in the plane under insertion and deletion of points in amortized $O(\log n)$ time per operation. The space usage of the data structure is $O(n)$. The data structure supports extreme point queries in a given direction, tangent queries through a given point, and queries for the neighboring points on the convex hull in $O(\log n)$ time. The extreme point queries can be used to decide whether or not a given line intersects the convex hull, and the tangent queries to determine whether a given point is inside the convex hull. The space usage of the data structure is $O(n)$. We give a lower bound on the amortized asymptotic time complexity that matches the performance of this data structure.

Preface

This PhD-Thesis is the result of my studies at BRICS. It was handed in February 21st 2002 and defended May 31st 2002. During this process several small errors in the writing have been discovered and are corrected in this version. Additionally a conference version [BJ02] of the result is published by now.

In the process of writing the conference version, preparing talks about the result and mainly in the discussion with the evaluation committee we discovered several possibilities to explain the result in a different way and to restructure the exposition. We also realized that some of the notation introduced here is non-standard. I am grateful to everybody for the feedback, in particular Pankaj Agarwal and Mark de Berg.

This version does not attempt to restructure the exposition. We are currently writing on a journal version that (hopefully) gives a more concise, still precise and in structure easier to access description of the result.

When reading this version it might proof helpful to also take a look at the conference version [BJ02] for a concise account of the result, in particular the outline of the data structure.

Riko Jacob,
Munich, December 2002.

Acknowledgements

I want to thank my advisor Gerth Brodal for working with me on the subject, and for his support throughout the last two years. I am especially grateful for the time he spent proofreading the details of this thesis.

Sincere thanks go also to Olivier Danvy, Oliver Karch, Christa Mauer, Sven Oliver Krumke and Hans-Christoph Wirth for carefully proofreading drafts of this thesis and supporting me with criticism, comments and discussions.

Thanks to Peter Bro Miltersen and Rasmus Pagh for several discussions on previously known lower bounds.

Special thanks to Pankaj Agarwal and Mark de Berg for serving on the evaluation committee.

Riko Jacob,
Århus, 21st February 2002.

Contents

List of Figures

Chapter 1

Introduction

Planar geometry is one of the oldest branches of mathematics. It is concerned with the position of points relative to each other and to objects such as lines and circles. For example we can prove that all three perpendicular bisectors of a triangle meet in one point. Highly related, but different in flavor, is the question of how we can find this point, given only the corners of the triangle. The first is an existential statement, whereas the second is an algorithmic question, the type of question we are concerned about in this thesis.

Computational geometry is the branch of computer science that considers algorithmic problems in geometry. The name was coined in the late 1970's. Since then the field has attracted numerous scientists. Part of this interests stems from applications where the underlying objects are most naturally modeled as geometric objects. Application areas of this type are as diverse as robotics, pattern recognition, virtual reality, scheduling, geographical information systems, typesetting, computer games and Computer Aided Design (CAD). Besides these application, computational geometry developed a solid theoretical framework and lead to several powerful paradigms for geometric algorithms. The maturity of the field is documented in textbooks [PS85, dBvK$^+$97, Meh84b] and handbooks [GO97, SU00] on computational geometry.

One of the first questions computational geometry raises is that of the nature of an algorithm. One of the oldest type of algorithms is the construction with ruler and compass. There one can draw a line through two (already constructed) points, and draw circles around a point (with an arbitrary radius or with the radius defined by another already constructed point). The intersection of two lines, two circles or a line with a circle define a point. This framework allows to consider if and how fast certain geometric points can be constructed. In other situations one is only interested in classifying the input points, for example whether they all lie on a circle. For this it might be convenient to construct the center point, but the problem definition does not require to produce such a center point.

Generalizing the geometric primitives, we can express geometric construction in terms of algebraic computations on the (real) coordinates of points. We formalize the notion of an algorithm (the model of computation) in Section 2.1.

Having defined the precise primitives of algorithms, we can consider the running time of an algorithm and try to find fast (efficient) algorithms. We measure the speed of an algorithm as a function bounding the number of elementary steps depending on the input size. In our setting this is the number of algebraic definitions and comparisons depending on the number of input points in the plane that we process. The usual measure is asymptotic, leading to the well know $O(\cdot)$ notation. Specifically we ignore constant factors. On a Turing machine this is justified by the linear speed-up theorem. In the geometric setting this is more a matter of convenience and focus.

1.1 Static convex hull computation

Computing the convex hull of a finite set of points in the plane is one of the oldest problems considered in the setting of computational geometry. It is actually older than the term computational geometry itself. The convex hull of a set S of points is intuitively easy to describe: think of the shape of a rubber band that is snapped from the outside against the points of S. Figure 1.1 illustrates the concept. (Section 2.4.1 provides a precise definition.) Applications of convex hull computations range from pattern recognition, scheduling, statistics, and collision detection.

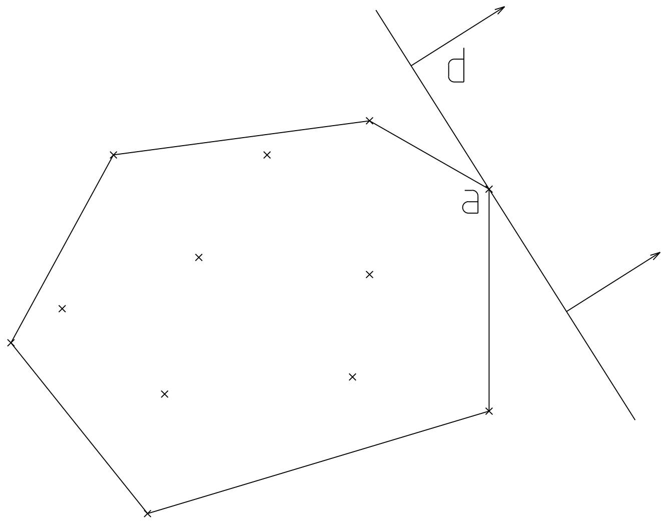

Figure 1.1: The convex hull of a set of points. The point a is the result of the extreme point query in direction d.

Several algorithms solve the convex hull problem in the plane. For a finite set S of points in the plane as input the algorithms compute the convex hull of S as a clockwise sorted list of the points of S that are vertices of the convex hull of S.

We only name a few such algorithms, in particular the ones that (in variants) will be used in the algorithm(s) presented in this thesis. Let $n = |S|$ be the number of points in the set and h the size of the convex hull, i.e., the number of vertices of the convex hull of S. There is Jarvis' march [Jar73] using time $O(n \cdot h)$, computing the vertices of the convex hull in clockwise order, finding the next vertex by exhaustive search. Then there is Graham's scan [Gra72] using time $O(n \log n)$, Andrew's vertical sweep line variant of Graham's scan [And79]). This algorithm first sorts the points in lexicographical order and then eliminates vertices that are not on the convex hull. There is a solution by Preparata and Hong [PH77] that achieves also $O(n \log n)$ time. It is a divide-and-combine algorithm based on a recursive bridge finding between vertically separated convex hulls. Later Kirkpatrick and Seidel [KS86] settled the asymptotic computational complexity of the problem by giving a recursive bridge finding algorithm that runs in $O(n \log h)$ time. This algorithm introduces the marriage-before-conquer paradigm. They also show a matching lower bound. Finally Chan [Cha96] gave a much simpler algorithm that achieves the same time bound. There the basic idea is to partition the set of input points, compute the convex hull of the parts with Graham's scan and to then compute the overall convex hull (merging the parts) in an efficient version of Jarvis' march.

1.2 Dynamic planar convex hull problem

The dynamic version of this problem is to keep track of the convex hull as the set of points S is changed by inserting and deleting points. There are applications asking for this kind of a data structure. Furthermore a dynamic planar convex hull data structure can be used as a black box inside an algorithm. Examples are the k-level problem in the plane [EW86, HPS01, Cha99b], the order-k Voronoi-diagram in the plane [CE87, AdB+98], the (connected) segment intersection problem in the plane [BGR96], and the two-dimensional linear programming problem with violated constraints [Mat95].

Usually the time complexity of a data structure is expressed as upper bounds on the execution time of a single operation. This is referred to as a worst-case bound. In an application, especially if the data structure is used as a black box inside an algorithm, the running time of a single operation is often not so important. It is more important how much time the algorithm spends inside the data structure. This is formalized as amortized analysis, where we average the execution time of the operation inside an arbitrary sequence of operations. (Section 2.2 provides a definition.)

The algorithms for the static problem can be used as a solution to the dynamic planar convex hull problem. After every change to the set S the convex hull of S is recomputed from scratch. The $O(n \log h)$ update time can easily be reduced to $O(n)$. This is achieved by keeping the set S in lexicographic order. Then Graham's scan (in Andrew's variant) runs in $O(n)$ time. A single insertion or deletion of one point can have a drastic impact on the convex hull of S. More precisely can a single insertion change the convex hull from containing all the points of S to containing only three points. The corresponding deletion can change the size of the convex hull from 3 to n. Given this, it is unclear in which sense an improvement over the linear update time is at all possible.

Instead of reporting the changes, we can require the data structure to maintain the points of the convex hull accessible in a specific data structure, which would naturally be a leaf-linked balanced search tree, storing the vertices of the convex hull in clockwise order. Such a tree representation allows to perform all kinds of queries in optimal $O(\log n)$ time. In this setting the data structure presented by Overmars and van Leeuwen [OvL81] achieves updates in worst-case time $O(\log^2 n)$. This is still the fastest solution providing an explicit representation of the convex hull (in a search tree as stated above). It also achieves the best known worst-case time bounds on the single update operations.

The problem gets somewhat easier if the data structure allows only specific queries to the convex hull of the set. One of the easiest such queries is the extreme point query in direction d. The task is to report the point $p \in S$ that maximizes the inner product $\langle d, p \rangle$. The *inner product* for 2-dimensional vectors (or points in the plane) is defined by $\langle (u, v), (x, y) \rangle := u \cdot x + v \cdot y$ for $u, v, x, y \in \mathbb{R}$. This query is related to the convex hull in the sense that the answer point p is a vertex of the convex hull. Such a situation is depicted in Figure 1.1 (p. 2). Another important query is the *gift-wrapping* query. The query consists of a point of the convex hull and asks for the two neighboring points on the convex hull. The gift wrapping query can be seen as a degenerate case of the *tangent query* that gives a point and asks for the two tangents on the convex hull that pass through this point (or the statement that the point is inside the hull).

Another interesting variant of the problem is the restriction of the updates to be either insertions only or deletions only, or if we assume that we know the complete sequence of updates in advance. For the insertion-only problem Preparata [Pre79] gives an $O(\log n)$ worst-case time algorithm to maintain the convex hull in a search tree. The deletion-only problem is solved by Hershberger and Suri in [HS92], where the authors give a construction where initializing the data structure (build) with n points and up to n deletions are accomplished in overall $O(n \log n)$ time.

Again Hershberger and Suri consider the off-line variant of the problem [HS96]. In this variant both insertions and deletions are allowed, but the times of all insertions and deletions are known a priori. In other words, the algorithm processes a list of insertions and deletions, and produces in $O(n \log n)$ time and space a data structure that can answer extreme point queries for any time using $O(\log n)$ time. Their data structure does not provide an explicit representation of the convex hull. They can reduce the space usage to $O(n)$ if the queries are also part of the off-line information. They consider the online version of the problem a long-standing open problem. In a survey paper Chiang and Tamassia [CT92] regard dynamic planar convex hull as one of the two most important problems in dynamic computational geometry. More precisely they ask, whether a data structure exists that has $O(\log n)$ update and query times.

Chan [Cha99a, Cha01] presents a data structure that achieves $O(\log^{1+\varepsilon} n)$ amortized update time for the fully dynamic problem. This construction does not maintain an explicit representation of the convex hull, but allows for extreme point (and other) queries. The construction is based on a general dynamization technique attributed to Bentley and Saxe [BS80]. Using the semidynamic deletions only data structure of Hershberger and Suri [HS92], and the right choice of parameters in a finite number of bootstrapping steps, this achieves update times of $O(\log^{1+\varepsilon})$ for an arbitrarily small constant $\varepsilon > 0$. To achieve the $O(\log n)$ extreme point queries, a construction based on an interval tree is used. Brodal and the author [BJ00] improve the amortized update time to $O(\log n \log \log n)$. This work can be seen as an intermediate result between Chan's construction and the data structure presented in this thesis. The improved update time is achieved by constructing a semidynamic data structure that is adapted better to the particular use. More precisely this data structure supports build in $O(n)$ time under the assumption that the points are lexicographically sorted. The deletions cost $O(\log n \log \log n)$ amortized time. This, together with a careful choice of the parameters for the interval tree and two bootstrapping steps, yields amortized $O(\log n \log \log n)$ update times and worst-case $O(\log n)$ query time.

All these data structure have an $O(n)$ overall space usage.

1.3 Duality, lower envelopes, parametric heaps

We can transform the (dynamic) convex hull question by a standard *duality transformation* (mapping points to lines and vice versa) into an (upper and lower) envelope question. The lower envelope of a set of lines H is the boundary of the region U in the plane that is the intersection of the lower half-planes defined by the lines in H. See Figure 1.2 (p. 5) for an illustration. The lower envelope consists of segments of the lines in H. The region U is a convex set, and we can basically describe it as the convex hull of some points in the plane. The simple extreme point query transforms (in the dual) to a vertical line query, where we take a vertical line l in the plane and ask for the input line that intersects l lowest. Section 4.1.3 gives a definition of the duality transformation.

The dual transformation is computationally trivial. It is more changing the point of view than changing the problem. If we are interested in an explicit representation of the convex hull or lower envelope, there is no algorithmic difference. We will

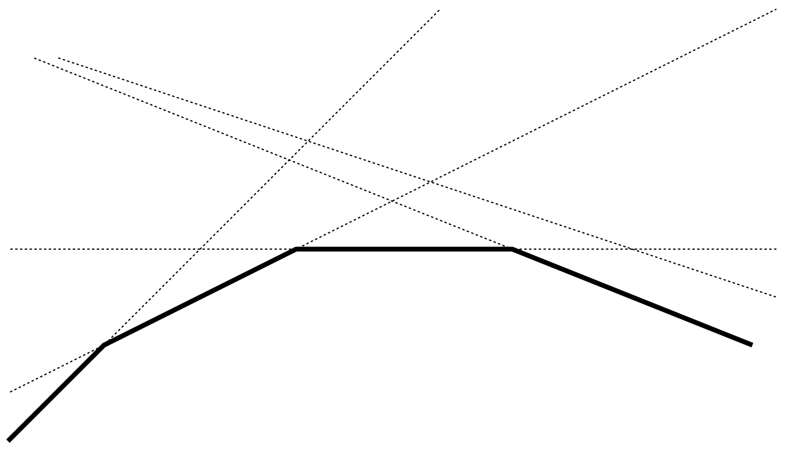

Figure 1.2: The lower envelope of a set of 5 lines. In the kinetic setting time elapses from left to right.

change the point of view from the primal to the dual whenever this makes the exposition easier.

The dual problem of maintaining the lower envelope is also known as parametric and kinetic heaps. The concept arises when generalizing heaps (priority queues). Instead of storing single values, a parametric heap stores linear functions. The intuition is that the values change (linearly) over time. A query is then asking for the minimal element at a certain point of time. This is precisely a vertical line query on a lower envelope. By duality this is equivalent to a extreme point query on a planar convex hull. The data structure can be updated by inserting and deleting linear functions. A kinetic heap is a parametric heap with the restriction that the argument (time) of a query has to advance (not decrease) between queries.

Kaplan, Tarjan and Tsioutsiouliklis [HTK01] consider several (restricted) variants of parametric and kinetic heaps. They (independently) give a construction that is very similar to the solution of Brodal and the author [BJ00]. They also consider several special cases of the problem where the update operations are restricted in various ways.

1.4 Known lower bounds

For the static convex hull computation there is a well known reduction to sorting, presented for example by Shamos in his PhD-thesis [Sha78] or in the textbook by Preparata and Shamos [PS85]. This establishes together with Ben-Or's [BO83] result an $\Omega(n \log n)$ lower bound on the real-RAM for computing the convex hull. This result is strengthened by van Emde Boas [vEB80], and Preparata and Hong [PH77]. They consider the task of identifying the vertices of the convex hull (without making their order explicit). Even this simpler problem requires time $\Omega(n \log n)$. This lower bound is tightened by Kirkpatrick and Seidel [KS86] to $\Omega(n \log h)$, matching their algorithm. Here the parameter h denotes the number of vertices on the convex hull. This immediately leads to that n insert and query operations require a total of $\Omega(n \log n)$ time.

The above lower bounds for the static problem imply some lower bound on the dynamic problem. We can use the dynamic data structure to compute the convex hull of a static set of points. To do so we insert the points one by one, and then perform Jarvis' march by performing gift-wrapping queries. This yields the lower bound that n insertions and n queries require a total processing time of $\Omega(n \log n)$. Insert and query together cannot be faster than $\Omega(\log n)$. As a query (intuitively and as we will show) requires $\Omega(\log n)$ time, this does not yield a non-trivial lower bound for insertions (or deletions).

In analogy to the reduction from sorting we can consider the predecessor problem. For this problem Miltersen (in a course) presents a lower bound in the comparison based setting. His proof is based on Fredman's use [Fre75] of Dilworth's theorem about chains and anti-chains. He achieves the same type of lower bound for insertions as we present in this thesis. The difference is that our bound holds in the stronger real-RAM model.

Ben-Amram and Galil [BAG01] study lower bounds for data structures on the real-RAM. Their methodology stems from the world of cell-probe complexity introduced by Fredman and Saks [FS89].

1.5 Contribution of this thesis

In this thesis we develop a data structure that achieves amortized $O(\log n)$ update time for the fully dynamic planar convex hull problem. The data structure allows worst-case $O(\log n)$ extreme point, gift-wrapping and tangent queries. The overall construction resembles that of Chan [Cha99a, Cha01] and Brodal and the author [BJ00].

The main improvement is a new semidynamic, deletions-only data structure. Instead of having an efficient build operation it allows for a linear-time merge operation of the data structures of two point sets. In the "combine" step of Bentley and Saxe's dynamization technique, we do not only reuse the lexicographic order of the points and construct the new sorting by a merging (sorting) step, but we actually reuse the already constructed data structures. We build a data structure that is capable of maintaining the convex hull of two convex hulls, given that the participating convex hulls can handle deletions correctly. This geometric merging data structure uses only $O(n)$ total processing time for building and handling up to n deletions. This new data structure, or the particular way of merging, requires $O(n \log n)$ space when used directly. The space usage can be reduced to $O(n)$ by introducing a geometric buffer. In a geometric buffer we maintain the convex hull of the set, but we eagerly delete the points that are on the convex hull from the recursive data structure. Placing a buffer above every merging level achieves that every point is stored in at most two data structures simultaneously. Together with a lazy movement strategy of the lines in the interval tree, we obtain the new result, summarized in the following theorem (this is a literal copy of the statement in Chapter 3):

Theorem 4.12

There exists a data structure for the fully dynamic planar convex hull problem supporting INSERT *and* DELETE *in amortized $O(\log n)$ time, and* EXTREME POINT QUERY, TANGENT QUERY *and* NEIGHBORING-POINT QUERY *in $O(\log n)$ time, where n denotes the size of the stored set before the operation. The space usage is $O(n)$.*

The new deletion-only data structure, where the build operation is replaced by a merge operation uses prominently a variant of a level-linked (2,4)-tree introduced by Hoffmann, Mehlhorn, Rosenstiehl, and Tarjan in [HM$^+$86], where in the amortized sense the split operation costs $O(1)$. We also use that this kind of tree allow for so called finger searches. The data structure also uses (geometric) elements from different static convex hull algorithms. Furthermore we use a variant of a B-tree with logarithmic degree as the underlying structure for the interval tree.

The data structure we present in this thesis improves over the general data structure for kinetic heaps presented in [HTK01]. It also matches the performance of the specialized data structures for arbitrary slopes. Section 4.6 gives the detail of how to use our semidynamic solution to achieve amortized $O(1)$ kinetic heap queries.

Rounding off the analysis of the data structure we consider lower bounds for the problem. Inspired by the sorting reduction for the static planar convex hull problem, we consider (as an intermediate problem) the membership problem in the algebraic decision tree model. We deduce the lower bound for the data structure solely by a reduction on the real-RAM. We do not argue with the state of the data structure at any point in time. This is in contrast to the arguments about comparison based (linear decision tree) data structure and the cell-probe complexity approaches (as for example by Ben-Amram and Galil [BAG01]). The following theorem is again a literal copy, the statement belongs to Chapter 5.

Theorem 5.6

Let $\mathcal{A}$ be a data structure implementing the SEMIDYNAMIC INSERTION-ONLY CONVEX HULL *problem on the real-RAM. Assume $\mathcal{A}$ supports extreme point queries in amortized $q(n)$ time, and* INSERT *in amortized $I(n)$ time for size parameter n. Assume that q and I are smooth functions. Then we have*

$$q(n) = \Omega(\log n) \qquad \text{and} \qquad I(n) = \Omega\Big(\log \frac{n}{q(n)}\Big) .$$

The main contribution of this thesis is hence that it establishes the amortized asymptotic computational complexity of the dynamic planar convex hull problem with extreme point queries.

1.6 Applications

There are several (geometric) problems where some of the algorithms proposed in the literature use a dynamic planar convex hull data structure (or a parametric/kinetic heap) as a black-box. In several cases this black box is clearly the bottleneck of the algorithm. In these cases our data structure immediately improves these algorithms.

We consider the example of the *k-level problem* in the plane. The problem is in the dual setting and is given by a set S of n non-vertical lines in the plane. For every vertical line we are interested in the k-th lowest intersection with a line of S. This is given by a collection of line-segments from lines of S. This generalizes the notion of a lower envelope ($k = 1$) and an upper envelope ($k = n$). The situation is exemplified in Figure 1.3 (p. 8).

As discussed by Chan [Cha99b] we can use two fully dynamic kinetic heaps (dual of the dynamic planar convex hull problem) to produce the k-level of a set of n lines. If we have m segments on the k-level (the output size), then the resulting algorithm completes in $O((n + m) \log n)$ time. This improves over the fastest deterministic algorithms, (Edelsbrunner and Welzl [EW86], using Chan's data structure achieving $O(n \log n + m \log^{1+\varepsilon} n)$ time). It is faster than the expected running time $O((n + m)\alpha(n) \log n)$ of the randomized algorithm of Har-Peled and Sharir [HPS01]. Here $\alpha(n)$ is the slow growing inverse of Ackerman's function.

1.7 Open Problems

The main problem remaining open is to achieve the same time bounds as worst-case bounds instead of amortized bounds. The data structure we develop in this thesis

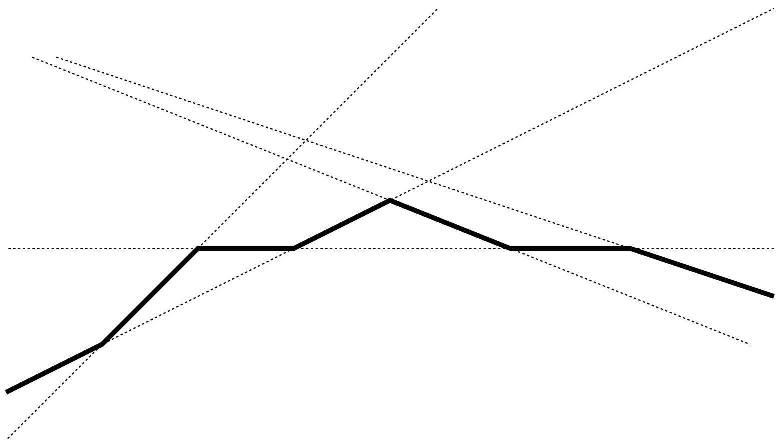

Figure 1.3: The 2-level of 5 lines in the plane. Note that the 2-level consists of 7 segments, two lines define two separate segments.

has a worst-case performance that we can basically only bound by the amortized analysis, that is an operation cannot take more than $O(n \log n)$ time. We have some preliminary ideas on how to reduce this to $O(n/\log^c n)$. These ideas are by no means close to achieving an $O(\log^2 n)$ worst-case performance as Overmars and van Leeuwen's data structure achieves.

This thesis gives basically one single new data structure for the dynamic planar convex hull problem. Given the size of the thesis, this naturally leaves the question of whether there is a simpler (in terms of the description) algorithm achieving the same performance.

In this work we consider primarily extreme point queries. This leaves the question open, how fast one can answer other types of queries. We have preliminary results indicating that we might be able to achieve $O(\log n \cdot (\log\log n)^{1+\varepsilon})$ bridge finding queries (linear programming queries in the dual). It remains open to find other concrete examples of efficient queries (on our data structure or on a different data structure), or an improved general framework that can achieve efficient queries.

1.8 Structure of the thesis

The second chapter contains basic definitions and considerations about the model of computation, and recapitulate the underlying data structures we use. This chapter contains Section 2.4 where we develop the necessary geometric framework. In the third chapter we develop the semidynamic, deletions only merging structure. In the fourth chapter we present the overall structure of the fully dynamic data structure and the interval tree that allows us to perform fast extreme point queries. We also discuss extensions to other queries there. The fifth and last chapter is devoted to the lower bound results.

Chapter 2

Definitions

We adopt the asymptotic notation as discussed in the textbook [GKP95]. In particular we write $f(n) = O(f(n))$ and $O(f(n)) = O(g(n))$ where the $=$ is not symmetric. Whenever we write $\log x$ we mean the (continuous) binary logarithm of x. As we are interested in the behavior of our algorithm for large values of n, we write terms like $\log\log n$ without explicitly stating that we require $n \geq 4$ for the term to be meaningful.

2.1 The model of computation

The concept of an algorithm is one of the most fundamental in computer science. If we want to make precise statements about the existence and efficiency of algorithms, we have to be precise about what an algorithm is. That is, we have to define a *model of computation*. An algorithm is usually formalized as a program on some kind of machine, for example a Turing machine or a random access machine (RAM). These models of computation are in the standard definition only capable of discrete inputs, expressed in a finite number of bits. This is not adequate to formalize geometric algorithms because the input are points in the plane, naturally modeled as $\mathbb{R}^2$. We could of course restrict our attention to fractional inputs, but this moves at least the focus of our attention away from the geometric intuition, towards the representation of numbers.

The classical algorithms to compute the convex hull of a static finite set of points use as geometric primitives only comparisons of coordinates, and for a line l defined by two input points the question if a third input point is above or below l. The data structure presented in this thesis uses also the construction of auxiliary points by the intersection of two lines, where lines are defined by already constructed points. We can restrict the algorithm to only construct auxiliary intersection points from lines defined by input points. We also use the technique of delayed (lazy) deletions, which means that the data structure uses input points even though they might already be deleted from the set of points to be maintained. The non-geometric part of our data structure is built entirely on search trees. All the variants of search trees we use can be implemented entirely pointer based. The parameter management of the data structure has to be able to count points, and determine the functions log, $\log\log$ and rounding to the next power of 2 for integers between 1 and n, the number of input points. We assume that it is possible to compute these functions in time $O(\log n)$. This is sufficient in our setting, as we do not need to compute these functions more than once per update operation.

The above assumptions already give the framework for the presented algorithm and data structure. We can implement our data structure on any model of computation that allows us to implement the above primitives. The time and space bound we give are on the level of these primitives. Space is measured in the number of auxiliary points and number of leaves of search trees. The running times reflect the number of geometric constructions and the (pointer and comparison) operations in the search trees.

In the remaining of this section we consider several models of computation that clearly satisfy the above assumptions. The real-RAM is what is usually used in the computational geometry literature. We also present the restricted version of it, that reflects our reduced need for geometric constructions. We also consider the algebraic computation tree model. This is a non-uniform model of computation that is clearly stronger than the real-RAM. We use this only for the lower bound result.

2.1.1 The real-RAM

We use as a model of computation the unit cost (algebraic) real-RAM. This kind of model is discussed in some detail in the textbook by Preparata and Shamos [Pre79]. We recapitulate the important details here.

The unit cost (algebraic) real-RAM consists of memory and a central processing unit that allows the manipulation of the memory according to a goto-program. The memory consists of two, a priori unbounded sized, arrays of cells, one storing integers, the other storing real numbers. The integer cells can be used as addresses into both types of memory (indirect addressing). The basic operations that can be carried out in constant time on cells holding reals are copying, addition, subtraction, multiplication, division and extraction of a square root. For the integer part we have integer versions of the above algebraic operations. We do not have floor or ceiling on the cells holding reals. Our program can branch depending on whether a (integer or real) cell is negative, zero or positive. We can use integer constants when manipulating real numbers. There is no automatic or implicit transformation from an integer to a real or vice versa.

We assume that input and output (especially in the data structure setting) is provided by storing the value in an (otherwise unused) dedicated register or part of the memory.

The running time of a program is measured by the number of instructions carried out, and the space usage is given by the largest address of a memory cell used during its execution.

This model can easily carry out construction of points with ruler (and compass) and geometric above/below decisions.

In general it is a strong assumption that the model can manipulate real numbers in constant time. It is a research topic of its own right to investigate what a reasonable formalism for computations over the real numbers could be, and how different formalisms relate to each other. An introduction to this area is for example the textbook by Weihrauch [Wei00]. He also describes (a variant) of the real-RAM and how it relates to other models. The main problem with the definition of the real-RAM is that a real number has arbitrary precision. Our algorithms actually does not really depend on this behavior. We can make this statement precise by introducing a somewhat weaker model of computation.

2.1.2 The order-k branching pointer machine

This is more a programming policy where we restrict the use of the integer part of the real-RAM. We do not use advanced address calculations, that is, for integers that stand for addresses we allow only dereferencing, addition and subtraction of 1. The use of the algebraic part of the real-RAM is restricted to branch on the value of a constant degree polynomial of constantly many input points. This means in particular that the real-RAM can simulate the order-k branching pointer machine without any slow-down.

With these restrictions the unit cost assumption is somehow more realistic. If we for example use an arbitrary precision floating point package, the performance of the single operations only depends on the size (accuracy) of the input points, but not on how many points constitute the input. In the same spirit we can assume that the input consists of b-bit integer numbers (the points stem from a square $2^b \times 2^b$). As it takes time $O(b)$ to evaluate a fixed degree polynomial over b-bit integers, a branching decision takes $O(b)$ time. Again assuming that the number n of input points is reasonably large in comparison to b, our running time analysis is meaningful.

The order-k branching pointer machine can be simulated by the BSS-model of Blum, Shub and Smale, introduced in [BSS90], and explained in more detail in the textbook [BC+98]. The overhead to do so is polynomial in the running time of the pointer machine. The BSS-model itself is not suited in our context because its program structure is very close to a Turing machine and not tuned for running times of $O(\log n)$. This is not surprising as the BSS-model was introduced to investigate complexity classes where a polynomial imprecision in the running times is irrelevant.

The order-k branching pointer machine is the model for which we formulate our algorithms.

2.1.3 Algebraic computation tree

The algebraic computation tree is a standard model, for example explained in the textbook by Blum, Cucker, Shub and Smale [BC+98]. It is a non-uniform model of computation that can simulate the real-RAM, the order-k branching pointer machine and the BSS-model with no slowdown. In its standard version it does not allow for the measurement of space. As it is one of the strongest reasonable models of algebraic computation, a lower bound in this model holds in many reasonable models of computation (both non-uniform and uniform models), in particular the ones we consider here. In Chapter 5 we give a precise definition of the algebraic computation tree, following the definitions in the article by Ben-Or [BO83]. There we also argue in detail that this model of computation can efficiently simulate the real-RAM (in our definition).

2.2 Amortized analysis

Often we are not so much interested in the running time of a single operation, but rather in the overall running time of a sequence of operations. To make this precise, we consider *amortized* time bounds. This concept is explained in detail in an article by Tarjan [Tar85] and in the textbook by Mehlhorn [Meh84a]. Assume for example that we have a data structure supporting the operations I, D and Q. For a sequence of operations consisting of i, d and q operations of the respective type. Let n be an upper bound for the input size parameter of this data structure, for example the number of points stored in the data structure. If we in this situation have a time bound of the type $i \cdot I(n) + d \cdot D(n) + q \cdot Q(n)$, we say that $I(n)$, $D(n)$ and $Q(n)$ are the respective amortized time bounds.

We usually use amortized analysis in opposition to worst-case analysis, where we determine how long a single operation can take in the worst-case. It should be noted that amortized analysis makes a statement about all sequences of operations, and is therefore most interested in the sequences that make the data structure perform worst. The amortized analysis is particularly appropriate if the data structure is to be used as a black box inside another algorithm. Then we are typically only interested in the overall time spent using the data structure, and not in the performance of the single operations.

The intuition behind the amortized analysis is that we allow the data structure to have "saved up" time from previous use of the data structure, and then use this saved time to perform a more complicated rearrangement of the representation. We follow this intuition when we perform the amortized analysis with the help of a potential function ϕ. There we take the data structure (and sometimes the history of it in the form of annotations) and assign a potential to it. We maintain the invariant $\phi(S_i) \geq 0$. Assume that the operation I on data structure S_i resulting in data structure S_{i+1} takes worst-case $I'(n)$ time. If we have $I'(n)+\phi(S_{i+1})-\phi(S_i) \leq I(n)$, we get that I takes amortized time $I(n)$. If we have several operations, we have to use the same potential function for all of them. We will see at the example of (2,4)-trees in Section 2.3 a well known example of amortized analysis.

2.3 Data structure basics

For a finite subset S of a completely ordered universe U we say that $p \in S$ is the *predecessor* of $u \in U$ if we have $p \leq u$ and there exists no $q \in S$ such that we have $p < q \leq u$. We say that p is the predecessor of u in S. Note that if we have $u \in S$, then u is its own predecessor. Let $S_{\leq u} = \{x \in S \mid x \leq u\}$ and $S_{>u} = \{x \in S \mid x > u\}$.

Definition 1
The Semidynamic Predecessor *problem asks for a data structure implementing the following operations, maintaining a finite set S of real numbers on the real-RAM. Initially $S = \emptyset$:*

Insert(x) *For $x \in \mathbb{R}$ change the set $S := S \cup \{x\}$.*

Predecessor(y) *For $y \in \mathbb{R}$ report the predecessor of y in S.*

Definition 2
The Semidynamic Membership *problem asks for a data structure implementing the following operations, maintaining a finite set S of real numbers on the real-RAM. Initially $S = \emptyset$:*

Insert(x) *Change the set $S := S \cup \{x\}$*

Element(y) *Report whether we have $y \in S$.*

We can use a data structure for the semidynamic predecessor problem to solve the semidynamic membership problem: We have that $x \in S$ if and only if x is its own predecessor in S. We use this observation in Chapter 5 to derive a lower bound for both problems in the algebraic branching tree model.

For upper bounds, i.e., algorithms, we actually are interested in more operations. A *level-linked (2,4)-tree* is a data structure T that can maintain a finite set S of elements from a completely ordered universe. Let size parameter be $n = |S|$, before the execution of the operation. It allows the following operations:

Extend(T, p, u) Extend the set S stored in T by the element u, that is, set $S := S \cup \{u\}$ under the assumption that p is a pointer to the predecessor (or successor) of u in S. Returns a pointer to the representation of u as an element of T. This operation takes amortized $O(1)$ time.

SEARCH(T, u) $(u \in U)$ Returns a pointer p to the predecessor of u in the set S stored in T. This operation takes $O(\log \min\{|S_{\leq u}|, |S_{>u}|\})$ worst-case time.

SPLIT(T, p) (p a pointer to some element u of the set S stored in T). Split the set T into two data structures, one for $S_{\leq u}$ and another for $S_{>u}$. The operation destroys the data structure for S. This operation takes amortized time $O(\log \min\{|S_{\leq u}|, |S_{>u}|\})$.

DELETE(T, p) (p a pointer to some element u of the set S stored in T). Remove u from S, i.e., set $S := S \setminus \{u\}$. This operation takes amortized $O(1)$ time.

The operations EXTEND, SPLIT, and DELETE take worst-case $O(\log n)$ time. This is a typical example where the amortized analysis shows that a data structure actually performs better than the worst-case bounds suggest. The operations INSERT and ELEMENT take $O(\log n)$ time, both in the worst-case and the amortized analysis.

In the textbook by Mehlhorn [Meh84a] the general (2,4)-tree is described and analyzed. The particular version of the (2,4)-tree that achieves the above performance is a level-linked (2,4)-tree due to Hoffmann, Mehlhorn, Rosenstiehl, and Tarjan [HM+86]. There the search operation is stated as a *finger search* in the following way: given any element u stored in the tree, we can find another element v of the tree in time $O(\log k)$, where k is the size of the set $S_{>\min\{u,v\}} \cap S_{\leq \max\{u,v\}}$. We can use this finger search to perform the SEARCH operation if we have a pointer to the leftmost and rightmost element stored in the tree. We then start such a search by comparing u with one of the keys r stored at the root of the tree. Then we can perform a standard finger search from the leftmost or rightmost element of the tree. So called level-links in the (2,4)-tree play in an important role for general finger searches. Our application uses only finger searches from the leftmost or rightmost element, for which the level links are actually not necessary.

Intuitively the good amortized performance of the EXTEND operation relies on the fact that most of the rebalancing of the tree happens close to the leaves. For this to stay true even though we might have interleaved EXTEND, DELETE and SPLIT operations, it is important that balance requirements are not too strict, namely that one node might have 2, 3 or 4 children. This is in contrast to (2,3)-trees, where we can have the situation that alternating insert and delete operations require (cascading) rebalancing on every level of the tree.

A B-tree T is a search tree where all leafs are on the same level, every internal node has, for some parameter B, between B and $2B - 1$ children, the root node has between 2 and $2B - 1$ children. At each node of T we have a (2,4)-tree holding the keys that separate the subtrees of the children. See the textbook by Mehlhorn [Meh84a, page 199] for a detailed exposition of B-trees. The balance requirements are maintained by performing (cascaded) node splittings, fusions and sharings. If the tree is monotonic, that is, we only insert new keys, we only need the split operation for nodes, the fuse and share operations are not needed. A *search path* in T for value u is a path from the root to a leaf v in T such that the leaf v stores the predecessor of u in T. We have that the *length* of a search path (the number of visited nodes in the B-tree) is $O(\log_B n)$.

2.4 Geometric preliminaries

A *slab* of $\mathbb{R}^2$ is the set of points between and including two vertical lines. Two points $u, v \in \mathbb{R}^2$ of a finite set of points P are said to be *neighboring* if there is no further point of P between the two vertical lines defined by u and v. For two points $u, v \in \mathbb{R}^2$ we denote with $\overline{u, v}$ the line segment from u to v, i.e. the set $\{p \in \mathbb{R}^2 \mid p = \lambda u + (1-\lambda)v,\ 0 \leq \lambda \leq 1\}$. A point p is above (below) a geometric object $Q \subset \mathbb{R}^2$ if the vertical line through p contains a point of Q and none of these points has a y-coordinate that is equal or larger than (smaller than) that of p.

2.4.1 Convex hull

Let $S \subseteq \mathbb{R}^2$, and $\{p_1, p_2, \ldots, p_n\}$ a finite subset of S. Let $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ be real numbers, $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i = 1$. Then $p = \sum \lambda_i p_i$ is a *convex combination* of points in S. The *convex closure* $\mathrm{CC}(S)$ of S is the set of points that can be written as a convex combination of the points in S. We define the *convex hull* $\mathrm{CH}(S) \subseteq S$ by the rule that a point $p \in S$ is in $\mathrm{CH}(S)$ if and only if $\mathrm{CC}(S) \neq \mathrm{CC}(S \setminus \{p\})$.

Property 2.1
Let H be the set of all half planes h that contain S, i.e., $h \in H \iff S \subseteq h$. Then we have

$$\mathrm{CC}(S) = \bigcap_{h \in H} h\ .$$

Instead of working with the convex hull itself we split the construction into an *upper hull* $\mathrm{UH}(S)$ and (symmetrically) a *lower hull* $\mathrm{LH}(S)$ as described in the following and illustrated in Figure 2.1. We define the *upper closure* $\mathrm{UC}(S)$ to be all the points $p \in \mathbb{R}^2$ such that there exists a point $q \in \mathrm{CC}(S)$, such that p and q are on one vertical line and p is not above q. Symmetrically we define the *lower closure* $\mathrm{LC}(S)$ to be all points of $\mathbb{R}^2$ that are on a vertical line not below a point in $\mathrm{CC}(S)$. For point p we have $p \in \mathrm{UH}(S)$ if we have $\mathrm{UC}(S) \neq \mathrm{UC}(S \setminus \{p\})$. Symmetrically we have $p \in \mathrm{LH}(S)$ if we have $\mathrm{LC}(S) \neq \mathrm{LC}(S \setminus \{p\})$.

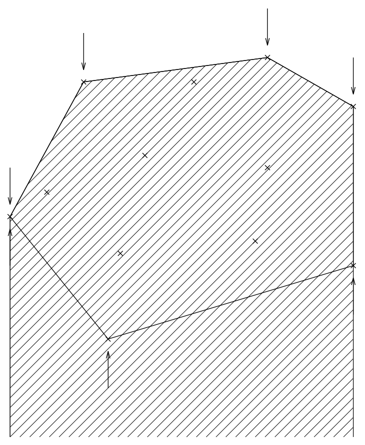

Figure 2.1: Splitting the convex hull of a set S into upper and lower hull; the leftmost point of S is on both upper and lower hull; the shaded region is the upper closure $\mathrm{UC}(S)$; the lines limiting the shaded region are the upper boundary $\mathrm{Bd}(S)$.

The upper hull and the lower hull can at most share the rightmost and leftmost point. They do not share the leftmost point only if there are two different points with minimal x-coordinate in the set.

Property 2.2
For a finite set $S \subseteq \mathbb{R}^2$ of points in the plane we have $\mathrm{CC}(S) = \mathrm{UC}(S) \cap \mathrm{LC}(S)$, $\mathrm{CH}(S) = \mathrm{UH}(S) \cup \mathrm{LH}(S)$, *and* $|\mathrm{UH}(S) \cap \mathrm{LH}(S)| \leq 2$.

This justifies that from now on we will only work with the upper hull. The lower hull is handled completely symmetrically.

Property 2.3
For a finite set $S \subseteq \mathbb{R}^2$ and a set $Q \subseteq \mathbb{R}^2$ assuming $\mathrm{UH}(S) \subseteq Q \subseteq \mathrm{UC}(S)$ *we have* $\mathrm{UC}(S) = \mathrm{UC}(Q)$ *and* $\mathrm{UH}(S) = \mathrm{UH}(Q)$. *We also have* $\mathrm{UH}(\mathrm{UH}(S)) = \mathrm{UH}(S)$ *and* $\mathrm{UH}(\mathrm{UC}(S)) = \mathrm{UH}(S)$.

Property 2.4
A point $p \in S$ is on the upper hull ($p \in$ UH(S)) if and only if there exists a vector $d = (a, b)$ with $a \in \mathbb{R}$ and $b > 0$ such that p is the unique extreme point *of S in the direction d, i.e., $\langle p, d\rangle > \langle q, d\rangle$ for all $q \in S \setminus \{p\}$.*

We say a line l is a *tangent* of S, if $l \cap S \neq \emptyset$ and all points of S are on or below l. If l is vertical all points of S have to be on the same side of l or on l. The set of all tangent lines on S is denoted by $\mathrm{Tg}(S)$.

The *slope* of a tangent line l is the parameter a in the representation $l = \{(x, y) \mid y = ax + b\}$. If l is a vertical line, no such representation exists, and we set the slope to be $+\infty$ if all of S is to the right of (or on) l. Symmetrically we set the slope to be $-\infty$ if S is to the left of l. If for a tangent line l the set $T = l \cap \mathrm{UC}(S)$ has more than one element, then T is a *segment* of the upper hull of S. The endpoints of the segment are two points in $\mathrm{UH}(S)$. The slope of the segment is the slope of the tangent line l. The vertical half-line below the highest leftmost (and rightmost) point of S are also considered segments of the upper hull of S.

Property 2.5
For a nonempty finite set $S \in \mathbb{R}^2$ we have $p \in$ UH(S) if and only if p is the intersection of two segments of the upper hull of S.

1. *The segments of S form a left to right path (with adjacency by shared points of* $\mathrm{UH}(S)$*), the slope of the segments decreases strictly from left to right.*

2. *If a tangent line l has only one point in the intersection $\{p\} = l \cap$ UC(S), then $p \in$ UH(S), and the slope of l is between the slopes of the segments incident with p.*

3. *For every slope $a \in \mathbb{R}$, there is precisely one tangent line l on S with slope a.*

For a set of points S we define $\mathrm{UC}_0(S)$ to be the topological interior of $\mathrm{UC}(S)$, and the *upper boundary* of S to be $\mathrm{Bd}(S) = \mathrm{UC}(S) \setminus \mathrm{UC}_0(S)$.

Property 2.6
For a finite set S the point-wise union of all segments of the upper hull of S is precisely $\mathrm{Bd}(S)$.

Lemma 2.7
Let l be a vertical line in the plane and S a finite set of points. Then the intersection $l \cap$ Bd(S) is either a single point or l is a vertical tangent.

Proof: For a vertical line l we have that there is at most one intersection point c with the segments of the upper hull. Otherwise we would have two overlapping segments. □

Lemma 2.8
Let l be a line in the plane, S a nonempty finite set of points. Then the intersection $l \cap$ Bd(S) consists either of at most two points, or l is a tangent line defining a segment.

Proof: First we note that if a point x on l is outside of the hull of A, then all the points to the left or to the right of x are also outside. Therefore the points inside form an interval on l. The two endpoints of the interval (if it is nonempty) are intersected by a tangent line or the vertical line through the leftmost or rightmost point. $\square$

2.4.2 Constructs with rays

We now consider properties of specific tangents on a convex hull. The monotonicity properties we discuss here are the geometric heart of the semidynamic upper-hull construction of Chapter 3. In the following we denote with S a finite nonempty set of points in the plane.

Let $l \in \mathrm{Tg}(S)$ be a tangent line. If $i = l \cap UC(S)$ is a single point $i = \{p\}$, then p is the *anchor* of l. If i is a segment, the anchor is the midpoint of the segment, i.e., for $\{p_1, p_2\} = i \cap \mathrm{UH}(S)$, we take $p = \frac{1}{2} \cdot (p_1 + p_2)$ as the anchor of i.

Property 2.9 (Monotonicity of tangents)
Let S be a finite set in the plane, l_1 and l_2 two tangent lines on $\mathrm{UH}(S)$ with distinct anchor points. Then the slope of l_1 is smaller than the slope of l_2 if and only if the anchor of l_1 is to the left of the anchor of l_2.

A *ray* r is a subset of a tangent line $l \in \mathrm{Tg}(S)$. It consists of one of the up to two unbounded connected regions of $l \setminus \mathrm{UC}(S)$. If the slope of l is not vertical, it is immediately meaningful to say that r is *directed* to the left or to the right. If l is vertical, the ray has to point upwards. If l is the right vertical tangent line, then we consider r to be directed to the left, and vice versa. There is always a point $p \in \mathrm{UH}(S) \cap l$ such that $r \cup \{p\}$ forms a closed set. This point p is called the *root* of ray r. This is illustrated in Figure 2.2.

The following lemma is formulated for a right directed ray. The symmetric lemma for a left directed ray holds just as well.

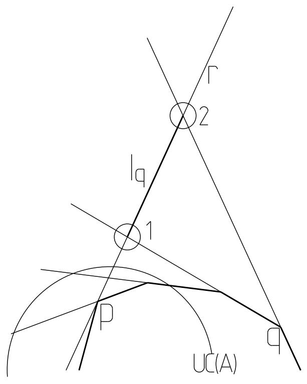

Figure 2.2: The situation of the ray monotonicity Property 2.10; the point p is the root of ray r; the interval I_q is depicted only for one example point; the circles marked with 1 and 2 are the intersections of i_1 and i_2 with r.

Property 2.10 (Monotonicity of rays)
Let S be a finite set in the plane. Let r be a right directed ray rooted at $p \in \mathrm{UH}(S)$. For every point $q \in \mathrm{UH}(S)$ that is to the right of p we define the subset $I_q \subseteq r$ of points that are an intersection of r with a left directed ray rooted at q.

Then all sets I_q are convex (line segments) and they partition r. The left to right order of the sets I_q is given by the order of the points q.

Specifically if $q_2 \in \mathrm{UH}(S)$ is to the right of $q_1 \in \mathrm{UH}(S)$ and the root $p \in \mathrm{UC}_0(A)$ for some finite set A and the intersection i_1 of some ray rooted at q_1 with r is such that $i_1 \notin \mathrm{UC}_0(A)$ then we can conclude that for any ray rooted at q_2 the intersection i_2 with r we have $i_2 \notin \mathrm{UC}_0(A)$.

Let S be a nonempty finite set of points in the plane. Let r_l and r_r be two rays rooted at $p \in \mathrm{UH}(S)$, where r_l is directed to the left and r_r is directed to the right. The rays r_l and r_r form a *valid pair* of rays, if the clockwise angle from r_l to r_r at p is of less than 180 degrees. Equivalently we can demand that the set $r_l \cup r_r$ is a subset of the interior of some upper half-plane, whose defining line is a tangent of S at p. The important consequence of this geometric configuration is that for a line l that intersects both r_l and r_r, the line l and the half-plane above it are disjoint from $\mathrm{UC}(S)$. See Figure 2.3 for an illustration of the situation.

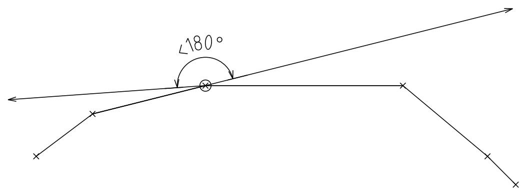

Figure 2.3: The geometry of a valid pair of rays

Assume there is a valid pair of rays a, b rooted at p (from now on writing the pair this way shall imply that a is directed to the left and b is directed to the right). Let q be the left neighbor of p and o the right neighbor on $\mathrm{UH}(S)$. If o is on the same tangent as a and q on the same tangent as b, then the pair of rays a, b is called *canonical*. The canonical pair of rays has the smallest possible angle between the rays. If p is the rightmost or leftmost (or only) point of S, the corresponding ray is vertical.

2.4.3 The geometry of merging two hulls

Now we consider the situation where we have two finite nonempty sets A and B of points in the plane and we are interested in $\mathrm{UH}(A \cup B)$. We call this situation the *merging* of A and B. To simplify the description, we assume that A and B are in upper-hull position, i.e., $A = \mathrm{UH}(A)$ and $B = \mathrm{UH}(B)$.

Property 2.11
For two finite sets of points A and B in the plane we have that $\mathrm{UH}(A \cup B) \subseteq (\mathrm{UH}(A) \cup \mathrm{UH}(B)) \setminus (\mathrm{UC}_0(A) \cup \mathrm{UC}_0(B))$.

The *symmetric difference* $\mathrm{SD}(A, B)$ of two finite sets of points A and B is given as by their upper closures, $\mathrm{SD}(A, B) = \big(\mathrm{UC}(A) \cup \mathrm{UC}(B)\big) \setminus \big(\mathrm{UC}(A) \cap \mathrm{UC}(B)\big)$. If p is a point in $\mathrm{Bd}(A) \cap \mathrm{Bd}(B)$, we say that p is an *equality point*. A vertical line through an equality point p does not intersect $\mathrm{SD}(A, B)$.

Property 2.12
Let $E \subseteq \mathbb{R}$ be the set of all x-coordinates of equality points of two finite sets of points. Then E is a finite collection of closed intervals. $\mathbb{R} \setminus E$ is a finite collection of open intervals.

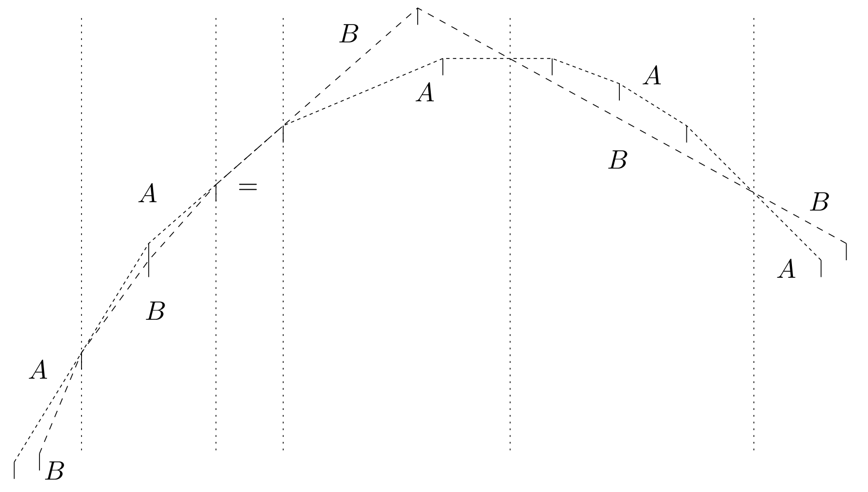

Figure 2.4: Strips and equality points of two upper hulls. The leftmost equality point is degenerate in the sense that the polarity of the neighboring strips is the same. Then we have an equality stretch. The second to the right strip of polarity A over B is trivial in the sense that B consists of a single segment.

If we have non-isolated equality points, we say that we have an *equality stretch*, that is we have a polyline, several consecutive segments (or parts of segments) of $\mathrm{Bd}(A)$ that are also part of $\mathrm{Bd}(B)$.

Let $W \subseteq \mathbb{R}^2$ be a maximal slab not containing an equality point in its interior. Then W is called a *strip* of the merging. We refer to the topological interior of W by W_0.

Property 2.13
Let l_1 and l_2 be two vertical lines in the interior of one strip of the merging of A and B. Then the intersection of l_1 with $\mathrm{Bd}(A)$ is above the intersection of l_1 with $\mathrm{Bd}(B)$ if and only if the intersection of l_2 with $\mathrm{Bd}(A)$ is above the intersection of l_2 with $\mathrm{Bd}(B)$.

A strip therefore has a *polarity*, we say that $\mathrm{Bd}(A)$ is locally above (outside) $\mathrm{Bd}(B)$ (or vice versa). It additionally has the two defining equality points, the so called *delimiter points*, unless it is the rightmost or leftmost strip.

Chapter 3

Semidynamic upper hull

In this chapter we develop a data structure that maintains the upper hull of a set of points when points can be deleted. In the constructions of [Cha99a, Cha01] the important feature of such a data structure is that it allows a fast build operation, that is given a set of points the data structure can be created efficiently from scratch. In [BJ00] one of the important observations is that we can improve the efficiency of the build operation, if we assume that the set of points is given in lexicographical order. In the overall setting of [BJ00], it is easy to compute the lexicographic ordering of the points, as the points stem from $\log n$ sets that are already lexicographically ordered, we only have to perform a generalized merging step. But apart from this ordering of the points the data structure and the geometric information it contains is destroyed when we build a new one. Here we go a step further, we continue to use the existing instances of the data structures and create the new instance by maintaining a geometric merging. In some sense we swallow the existing instances and continue to use them. We achieve overall linear time spent in one level of merging, which yields an overall amortized cost of $O(\log n)$ for inserting (and deleting) an element in the fully dynamic solution.

This approach (inherently) leads to an overall space usage of $O(n \log n)$. If we want to avoid this, we can use the semidynamic upper hull data structure of [BJ00], which yields an overall space usage of $O(n)$, at the cost that a deletion then takes amortized $O(\log n \log\log n)$ time. As this step makes the construction of this chapter superfluous, we do not discuss this possibility here any further.

Instead of a completely modular point of view that would allow us to consider a single merging data structure, we will work on the resulting complete data structure. This makes the analysis rather direct because we do not have to consider complexity that is hidden in the form of passing big arguments between modules. This is one of the many choices we can take without really changing the algorithm. Here we always choose the solution that (we think) is easiest to explain.

3.1 The interface

In the *semidynamic* upper hull problem we ask for a data structure with the following interface:

Definition 3 (Semidynamic Merging Structure)
is a data structure that supports the following operations

CREATE_SET(p) *The point p in the plane is given by its (x, y) coordinates. Creates a set A which contains only the point p; Returns a pointer to the data structure* *representing A and a pointer to the representation of the point p, its* base record.

MERGE(A, B) *The sets A and B are given by a pointer to their merging data structure. Creates a new merging data structure for the set $C = A \cup B$ and returns a pointer to C. Through C the upper hull of the points stored in C can be accessed in left to right order as they are stored in a doubly linked list. The data structures representing A and B are from now on only accessible from inside the data structure for C. We say that from now on the merging structure C owns all the points of $A \cup B$.*

DELETE(r) *The point r is given by a pointer to its base record. Removes r from all the sets it is stored in. Determines the merging structure M that owns r. It is assumed that r is on the upper hull of M. Returns the list L of points that replace r on the upper hull of M. The list L is represented as a doubly linked list between the points a and b, where a and b are the neighbors of r on the upper hull of M before the deletion. The function returns pointers to a and b.*

Two points are considered to be different, even though they have the same coordinates. The identity of points is solely given by the pointer to its base record.

Our semidynamic merging structure maintains a forest of rooted trees, where the leafs are (singleton) set of points in the plane, and the root nodes allow access to the upper hull of the union of the sets stored in the leafs. Additionally we can delete points from the leaf sets and combine two trees by merging the root nodes (creating a new root and make two old root nodes the children). A deletion of a point at a leaf propagates to the root of the current tree. The binary nodes on this paths are also referred to as *merging levels*. The overall time spent in the data structure is linear in the number of points stored at the leaves for every (binary merging) node in this forest. If we merge only data structures that contain the same number of merging levels, all merging trees are complete perfectly balanced binary trees. One merging operation costs amortized $O(n)$ time, where n is the size of the participating sets. This achieves an amortized $O(\log n)$ time bound for inserting a point, namely $O(1)$ for every merging level it participates in. The deletion of a point r also costs amortized $O(1)$ time per merging r participates in. This yields an overall amortized cost of $O(\log n)$ per deletion.

The assumption that we can only delete points that are on the overall upper hull of the merging tree they are stored in, is convenient for our geometric construction. As we are only interested in amortized time bounds, this is an assumption we can easily enforce. We merely check whether the point r is on the overall upper hull, and if it is not we delay the deletion of r until it surfaces.

In some sense the word "semidynamic" is not completely correct for the situation. After all the sets can get bigger (by merging), even though we already deleted some of the points. On the other hand the data structure is not directly fully dynamic, it does not allow an arbitrary insert operation. It only creates explicit representations of the upper hulls of the merged sets (one for each merging tree), not for the overall set (all the points stored in the merging forest). As the data structure replaces what is a semidynamic data structure in [Cha99a, Cha01, BJ00], we think this small inaccuracy is not too misleading.

3.2 The static geometric algorithm

To introduce and give an intuition about the geometric concepts needed for the semidynamic upper hull data structure (and their geometric properties that ultimately imply the correctness) we consider in this section a complicated way to compute the upper hull $\mathrm{UH}(A \cup B)$ of two finite point sets A and B where $\mathrm{UH}(A)$ and $\mathrm{UH}(B)$ are already (recursively) computed. As a simplification of the notation we assume that A and B are in upper convex position, i.e., $A = \mathrm{UH}(A)$ and $B = \mathrm{UH}(B)$. The focus of this section is to define several geometric constructions that are also used by the semidynamic data structure presented in Section 3.5 and Section 3.6. To simplify the exposition, we first show their important properties in the static algorithm. This static algorithm is in fact very closely related to the dynamic data structure, as it produces a geometric situation that corresponds to a valid state of the data structure. In this sense it can be seen as an initialization algorithm.

In the following we try to motivate why the constructs as introduced here might ultimately be useful. Usually it suffices to point at a prominent feature of the construct to make clear what the function of the construct in the overall construction is. Unfortunately these one line descriptions might only make sense when looking back, i.e., after having seen the complete construction.

We use the left to right ordering on the points of $C = A \cup B$, as it carries a lot of useful geometric information, e.g., about segments and tangent lines, but we will disregard representation and data structure issues for now.

3.2.1 Bridges

A segment of $\mathrm{UH}(A \cup B)$ is called a *bridge*, if it has one endpoint in A and the other in B, or if it is a segment of $\mathrm{Bd}(A)$ and contains a point of $\mathrm{UH}(B)$ (or with A and B exchanged). This situation is illustrated in Figure 3.1.

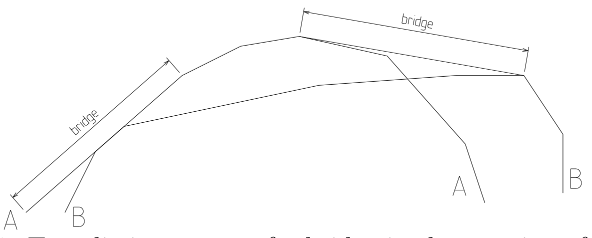

Figure 3.1: Two distinct types of a bridge in the merging of A and B.

Property 3.1
Every bridge s of the merging of A and B contains or is above one equality point of A and B.

Property 3.1 hints at the importance of finding the equality points. Algorithmically it will be important to have certificates that show that segments contain no equality points. Consider a strip of polarity A below B. For a segment s of $\mathrm{Bd}(A)$ such a certificate is that both endpoints of s are inside $\mathrm{UC}_0(B)$, i.e., below the segment formed by two points of $\mathrm{UC}(B)$. For a segment t of B such a certificate can be a tangent line l of $\mathrm{UC}(A)$ such that t is (strictly) above l.

Property 3.2
Let A and B be two finite sets of points in the plane. Then two points $a \in A$ and $b \in B$ form a segment of $\mathrm{Bd}(A \cup B)$ if and only if the line l through a and b is (simultaneously) a tangent on A and on B and the set $l \setminus \overline{a,b}$ contains no point of A or B, i.e. $(l \setminus \overline{a,b}) \cap (A \cup B) = \emptyset$

The algorithmic consequence of this last property is that even though it might be hard to find bridges, it is only a local condition to verify that (a, b) forms a bridge, if we already computed the upper hulls UH(A) and UH(B) in left to right order. We merely have to check that the neighbors of a on UH(A) and the neighbors of b on UH(B) are below the line defined by a and b.

Let us assume that we already know all the equality points of the two hulls, namely the set $\mathrm{Bd}(A) \cap \mathrm{Bd}(B)$. In this section we consider the geometric and algorithmic task of finding the overall upper hull $\mathrm{UH}(A \cup B)$. In some sense this amounts to an algorithmic version of Property 3.1 (p. 21), namely how to compute bridges starting from singular equality points.

The following two lemmas establish that we have a one to one connection between equality points of Bd(A) and Bd(B) and bridges.

Lemma 3.3
Let A be a finite set of points with $A = \mathrm{UH}(A)$ and let $u, v \in \mathrm{Bd}(A)$, be two points, u to the left of v, and l the non-vertical line defined by u and v. Then there is no point of A above l to the left of u or to the right of v.

Proof: Assume there is a point $p \in A$ to the left of u above l. Then the line segment $\overline{p,v} \subset \mathrm{UC}(A)$, implying $u \in \mathrm{UC}_0(A)$, a contradiction to $u \in \mathrm{Bd}(A)$. □

Lemma 3.4
Let S be a strip of the merging of $A = \mathrm{UH}(A)$ and $B = \mathrm{UH}(B)$ into $C = A \cup B$ of polarity Bd(B) above Bd(A). Then we have $B \cap S \neq \emptyset$, and $\mathrm{UH}(C) \cap S \neq \emptyset$, i.e., at least one of the points of $B \cap S$ is on the overall upper hull UH(C).

Proof: Let u and v be the left and right equality points of Bd(A) and Bd(B) limiting S. Then we know that the segment $\overline{u,v}$ is subset of both UC(A) and UC(B). If there is no point of B inside S_0, then $\overline{u,v}$ is part of Bd(B), which implies that we have $\mathrm{Bd}(B) \cap S_0 \subset \mathrm{UC}(A)$. This contradicts the definition of S being a strip where B is above A. So we have $B \cap S_0 \neq \emptyset$.

Assume now $\mathrm{UH}(C) \cap S_0 = \emptyset$. Then we have a segment $\overline{x,y}$ of Bd(C), such that x is to the left of S_0 and y is to the right of S_0. Let $p \in B$ be a point in $B \cap S_0$. The line l defined by u and v is then below p. By Lemma 3.3 (p. 22), we get that x and y are on or below l, implying the contradiction that p is above $\overline{x,y}$.

If S is the leftmost (or rightmost) strip of the merging, then the leftmost (and highest) point $p \in C$ is in B. We have $p \in \mathrm{UH}(C)$. □

We say that a bridge (a, b) is a *degenerate bridge* if it is completely covered by Bd(A) and Bd(B), i.e., we have $\overline{a,b} \cap (\mathrm{Bd}(A) \cup \mathrm{Bd}(B)) = \overline{a,b}$. We say that an equality point e is a *degenerate equality point* if it is part of the merged upper hull, i.e., we have $e \in \mathrm{UH}(A \cup B)$. We can detect degenerate bridges and equality points by considering constantly many neighboring points on UH(A) and UH(B).

Lemma 3.5
Let A and B be two finite sets of points, S_1 and S_2 two consecutive strips of the merging of A and B, with A above B in S_1 and B above A in S_2. Assume that there is a non-degenerate equality point e between S_1 and S_2. Then there is precisely one bridge of $\mathrm{UH}(A \cup B)$ between a point of A in S_1 and a point of B in S_2.

Proof: As e is non-degenerate, it is not on $\mathrm{Bd}(A \cup B)$. Let $\overline{x,y}$ be the segment of $\mathrm{Bd}(A \cup B)$ that is directly above e. (There cannot be a point of $\mathrm{UH}(A \cup B)$ directly above e.) W.l.o.g. we assume that S_1 is to the left of S_1 and x to the left of y. By Lemma 3.4 we have $x \in S_1$ and $y \in S_2$. By definition of a strip we have $\mathrm{UH}(A \cup B) \cap S_1 \subseteq \mathrm{UH}(A) \cap S_1$ and $\mathrm{UH}(A \cup B) \cap S_2 \subseteq \mathrm{UH}(B) \cap S_2$ Hence (x, y) is a bridge of $\mathrm{UH}(A \cup B)$. □

Lemma 3.6 (Bridges and equality points are one-to-one)
Under every non-degenerate bridge is precisely one non-degenerate equality point, and above every non-degenerate equality point is precisely one non-degenerate bridge.

Proof: By Lemma 3.4 one bridge cannot span over a strip of the merging, in particular not over more than one equality point. By Lemma 3.5 there always exists such a bridge. □

Considering $A \cap S_1$ and $B \cap S_2$ we look at a horizontally separated bridge finding task, as considered in the following.

3.2.2 Bridge finding

Assume we have two finite sets of points A and B that are separated by a vertical line l. Assume further that we already know the upper hulls of A and B, and for simplicity that we have $\mathrm{UH}(A) = A$ and $\mathrm{UH}(B) = B$. We are interested in finding the bridge between A and B. This bridge is defined as the segment of $\mathrm{Bd}(A \cup B)$. It is also given as the only common tangent on A and B. The situation is illustrated in Figure 3.2. This bridge finding task is at the core of the dynamic planar convex hull data structure of Overmars and van Leeuwen [OvL81]. It is also described in detail in the textbook by Preparata and Shamos [PS85, page 127].

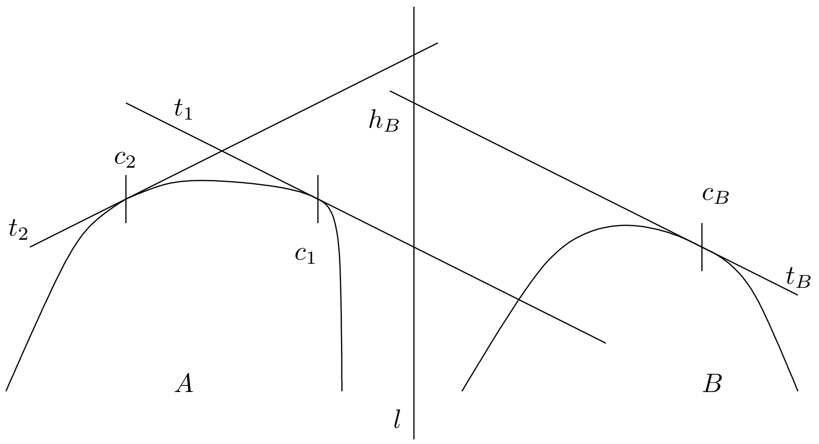

Figure 3.2: Classical bridge finding. The vertical line l separates A and B. Given the current candidate $c_B \in B$, we can argue that $c_1 \in A$ is too far to the right, whereas c_2 is too far to the left.

Assume that we have a current candidate c_A and c_B on both upper hulls. These candidates define a tangent t_A and t_B (actually they naturally define two tangents, but this simplifying assumption does not change the picture). We focus on the candidate c_A. In the situation where c_B is above the tangent t_A (in Figure 3.2 $c_A = c_1$), we can conclude that all the points to the right of c_A cannot be the touchdown point of the bridge on A, because they cannot be tangents of B (they do not have all of B on or below the tangent line).

For the other case we have to define an over-approximation of B. This is given by the current candidate c_B and its tangent t_B. If the current tangent of t_A of A is such that all the points to the right of l and below t_B (this includes all points of B) are strictly below t_A ($c_A = c_2$ in Figure 3.2), then all the points to the left of c_A cannot have a common tangent with B because their tangent lines cannot pass through a point of B.

These two geometric reasons allow us to search for the bridge. Unless both candidate points are the endpoints of the bridge, one of the two candidates is found to be either too far to the left, or to far too the right. The case analysis proceeds by drawing the segment $c = \overline{c_A, c_B}$ and comparing it to the two tangents that are defined by c_A on A and the two tangents defined by c_B on B. This yields on each side either a reflex (c_1 in Figure 3.2), concave (c_2 in Figure 3.2), or supporting (the segment c defines a tangent line on A or B) situation. Then we can consider all combinations of cases on A and B and conclude that in all of them (but the supporting-supporting case that identifies the bridge) at least one of the geometric reasonings is applicable.

We can use this geometric insight to perform an interleaved binary search (or a search given by a search tree) on A and B simultaneously. We can think of this as suspending both searches after they produced the first candidate point, and then advancing one of them according to the current geometric situation. The process stops when we find the bridge.

We will use similar concepts as the building block of our data structure: we use dangling searches (suspended searches), an over-approximation of the upper hull (as given by t_B) and an under-approximation of the upper hull (as given by c_B).

When we perform bridge finding as part of our data structure, we are in a slightly different situation. Instead of performing binary searches we perform (interleaved) linear scans. We start at an equality point of the the boundary of S_1 and S_2. Then we always see a reflex or supporting situation. In fact we end up performing a variant of the "remove non-convex point" (Andrew's variant of Graham's scan, [And79, Gra72]) algorithm to compute the upper hull of a point set that is already in left to right order.

When considering the dynamic data structure, we will have to make sure to not search over the same points several times. This makes it necessary to also have a phase of the bridge finding, where we perform a linear scan towards the intersection point (responding to a deletion on the other side of the intersection). There we will also meet the "concave vs. reflex" and "concave vs. supporting" situations.

3.2.3 Finding equality points

Before we can use the partition of the plane into strips as in the previous section, we have to find all the equality points of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. In the static setting we might employ the following algorithm, assuming that $A = \mathrm{UH}(A)$ and $B = \mathrm{UH}(B)$ are accessible in left to right order. We perform a left to right sweep line algorithm. For every point x of $A \cup B$, we define the vertical line l_x through x and determine the intersection points of l_x with $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. Without loss of generality (only as a naming convention) we assume assume $x \in A$. Then it is clear that $\mathrm{Bd}(A) \cap l_x = \{x\}$ and that $\mathrm{Bd}(B) \cap l_x$ depends only on one segment, namely from the rightmost point of B to the left of l_x and the leftmost point of B to the right of l_x. There is an equality point (intersection) of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ if we have that for two consecutive vertical lines l_x and l_y the above/below order of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ changes. As $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ are straight lines between l_x and l_y, we are also sure that we find all equalities of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. This algorithm identifies all strips of the merging of A and B in linear time.

Let us state the above algorithm in the form we want to use it later as part of our data structure. It is a modification, as we assume only that we have A and B accessible as two (doubly) linked lists. The sweep line gets replaced by a *sweep segment* $\overline{a, b}$ with $a \in A$ and $b \in B$. Let a' be the right neighbor of a and b' the right neighbor of b. We maintain the *slab invariant* that either a lies in the slab formed by b and b', or b lies in the slab formed by a and a'. If the segments $\overline{b, b'}$ and $\overline{a, a'}$ intersect, we report an equality point. To move the sweep line, we determine which of a' and b' is further to the left, say it is a'. Then we change the sweep segment to be (a', b). This leaves the slab invariant intact. If the sweep segment reaches the end of A or B, we are sure to have reported all the equality points.

To initialize the sweep segment and the slab invariant, we place the sweep segment at the leftmost point of A and B. This segment might not fulfill the slab invariant, say because a' is to the left of b. But then we can safely advance to (a', b), as $\overline{a, a'}$ certainly does not intersect $\mathrm{Bd}(B)$.

Let us reflect about the proof of correctness of the above algorithm (in the first description), and whether we can use it in a dynamic situation as well. The vertical lines l_x and their intersection with $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ are the atomic observations of our algorithm, geometric certificates that lead to its correctness. Observing that the algorithm actually did not exploit the fact that A and B are in upper convex position, we cannot leave out a single line l_x, without risking an incorrect result. In this sense the correctness of the algorithm depends on all these certificate lines.

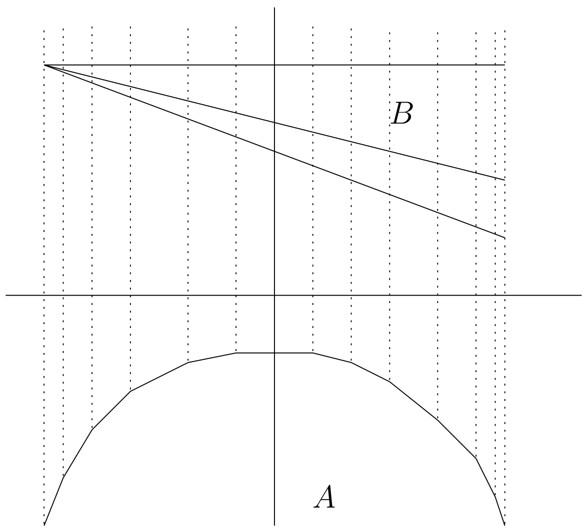

Figure 3.3: The example where maintaining vertical line certificates is expensive.

Let us consider the situation depicted in Figure 3.3. There we have $B = \{(-1, 1), (1, 1), (1, 0.5), (1, 0.25)\}$, and A is a set of size n with $A = \mathrm{UH}(A)$ and all points of A are below the x-axis and their x-values are between -1 and 1. Then the sweep line algorithm has n certificates of the above explained type. If we delete the point $(1, 1)$ from B, we have to update all n certificates, and we have to do this again when we delete $(1, 0.5)$ from B. This sequence of deletions shows that the particular set of certificates is not easy to maintain. In the example, it would be no problem to maintain completely different certificates, for example if we would have separated the task and maintain that all points of A are below the x-axis and all points of B are above the x-axis.

There is another interesting case, in which the above type of certificates is easy to maintain, and that is if the lines l_x alternate between being defined by a point of A and a point of B. In this case every segment would be part of precisely one certificate, and a single deletion would require only the update of two certificates.

What we actually use in the dynamic data structure does both: It does some kind of sampling (the sampling density depending on the geometric situation) from A and B in a way that ensures something in the spirit of alternation, and it introduces auxiliary lines that separate the geometric situation.

Somewhat more precisely we build two intermediate upper boundaries, one called $\hat{A}$ that is based upon points and segments of $\mathrm{Bd}(A)$ with the property $\mathrm{UC}(A) \subseteq \mathrm{UC}(\hat{A})$. Then we have another one, B' that is based upon points of B (and lines defined by $\hat{A}$), with the property $\mathrm{UC}(B') \subseteq \mathrm{UC}(B)$. To make all this useful we also make sure to have $\mathrm{UC}(\hat{A}) \cap S \subseteq \mathrm{UC}(B') \cap S$, for a strip S.

This is the vague and imprecise outline of our construction. We present the details in the next sections, exemplified on a complicated static algorithm to find the intersections of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. We start by presenting some of the concepts that are fundamental for the construction.

3.2.4 Selected points

In the following we restrict our attention to one strip of polarity $\mathrm{Bd}(B)$ above $\mathrm{Bd}(A)$. First we define what it means for a point $p \in A$ to be part of the hinted at sampling. A point $p \in \mathrm{UH}(A) \cap \mathrm{UC}_0(B)$ can be selected. In this case there is a pair of valid rays that are rooted at p, the so called *strong rays*.

Requirement 1 (Strong ray separation)
Let r and t be two different strong rays rooted at different points of $\mathrm{UH}(A)$, *not necessarily in the same strip. Then we require that the intersection point of r and t is outside of* $\mathrm{UC}(B)$.

The significance (for the dynamic setting) of strong rays is that it is sufficient to maintain the intersection of the strong rays rooted at A with $\mathrm{Bd}(B)$. Over time we will have to be able to deal with deletions of points in B. After any number of deletions the current set B' will be a subset of B, and we also have $\mathrm{UC}(B') \subseteq \mathrm{UC}(B)$, which means that the intersection of a strong ray with the $\mathrm{Bd}(B)$ move monotonically towards the selected points and that the strong ray separation requirement will stay valid.

3.2.5 Finding more points to select

Even though we describe the static case, the motivation clearly is the semidynamic setting, where we have to be prepared that the upper hulls "shrink". In this sense it is also meaningful to talk about the (unknown) future, meaning after some more deletions.

Now that we have defined what a selected point on A is, we can already imagine one approximation of A, namely the upper hull of the intersections of neighboring strong rays, that is from two neighboring selected points p left of q we take the intersection of the right directed strong ray rooted at p with the left directed strong ray rooted at q. This indeed contains all of $\mathrm{UC}(A)$, but by the invariants we imposed, it will be too big in the sense that every single point defining it is outside of $\mathrm{UC}(B)$. On the other hand we did not even define a condition that would enforce selected points to be close to each other, such that we can expect the approximation to resemble $\mathrm{UH}(A)$ in any sense. What we can do is to try to select as many points of $\mathrm{UH}(A)$ as possible without violating the strong ray separation, Requirement 1 (p. 26). Assume that we already have selected some points, among them $p \in \mathrm{UH}(A)$ and $q \in \mathrm{UH}(A)$, p to the left of q, where there is no further selected point of $\mathrm{UH}(A)$ between them. We consider the possibility of selecting another point $x \in \mathrm{UH}(A)$ between p and q. Let us assume that such a point x exists and that we try to find it using a comparison based (binary/exponential) search. Such a situation is depicted in Figure 3.4. Let us say that r is the right directed strong ray rooted at p and s is the left directed strong ray rooted at q. We have to be in the position to decide for a candidate point $c \in \mathrm{UH}(A)$, whether c

is to the right or the left of x. Let us say that e is the left directed canonical ray rooted at c, and f is the right directed one. Let a be the intersection point of r and e, and b be the intersection point of f and s.

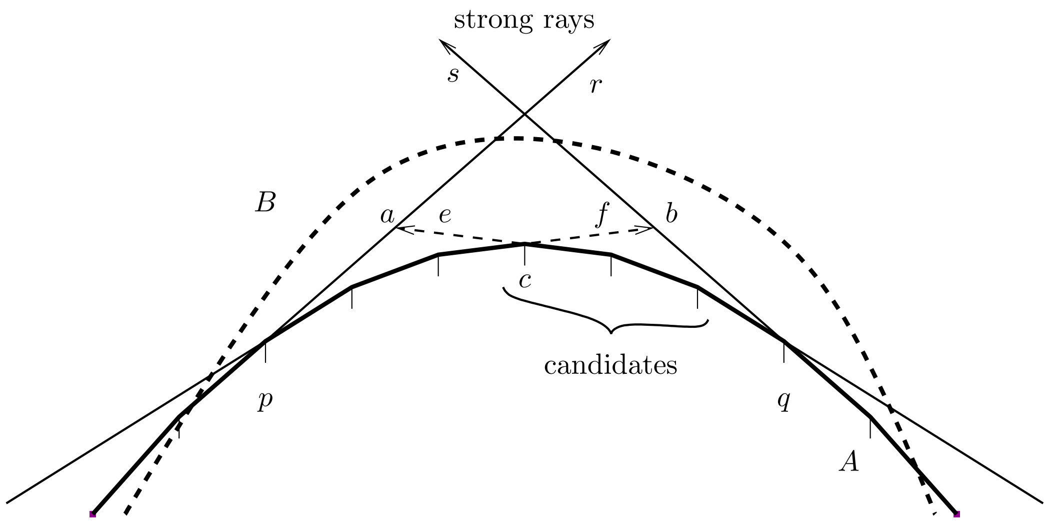

Figure 3.4: A relaxed dangling search: The upper hull B is depicted as smooth line (even so it is not) to stress the fact, that we do not care about the single segments of it here. The points p and q are the only selected points in the depicted strip. The point c is the candidate of the (relaxed) dangling search.

Now if both a and b are outside UC(B), we can select c as it fulfills the strong ray separation requirement. Even though we did not find the particular point x that could be selected by assumption, we found c instead, which is just as good.

If a is inside UC(B) and b is outside UC(B), we know by the monotonicity Property 2.10 (p. 16) that all points to the left of c will not meet the strong ray separation invariant to the left. So we should search to the right of c. In the dynamic setting we would have to say "c does not (yet) meet the strong ray invariant, but it might in the future." In this form it sounds like the evaluation of c as a candidate might be wasted effort and the branching we take can become wrong. If we instead state that "c and all points to the left of c already meet the strong ray condition with s," we have a statement that remains valid even in the setting of monotonically shrinking hulls. To be able to refer to this fact we call c the (new) *left guard* of this search.

If we see the symmetrical situation, namely that a is outside and b is inside, we know that we have to continue the search to the left of c, and make c the new *right guard* of the search.

It can also happen that both a and b are inside UC(B). Then we cannot continue the search, but instead we have c as a certificate that there cannot be any further selected point between p and q, i.e., our assumption about the existence of x is wrong. We call such a situation a *relaxed dangling search*. The point c is called the *candidate* point of the search and the canonical rays e and f are *weak rays*. The selected points p and q are called the *anchors* of the dangling search. We think of it as a search process of a data structure that can be suspended if we cannot (yet) decide whether we should go to the left or to the right. In the dynamic setting we will use this, as further deletions of points in B might eventually lead to one of the previous cases.

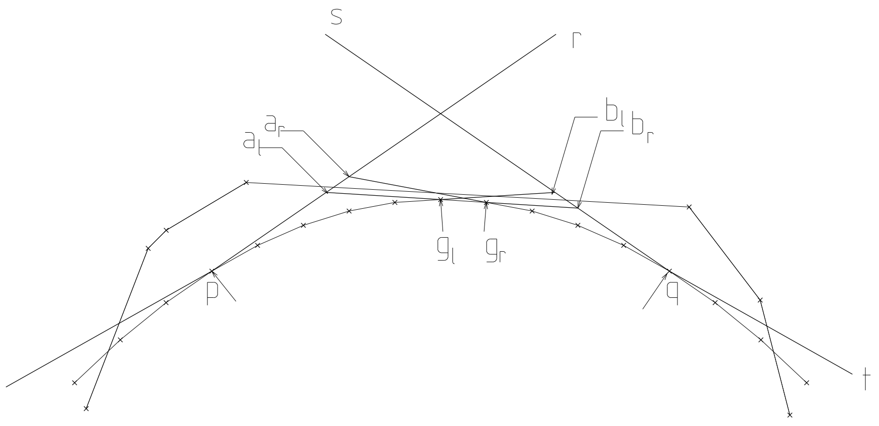

Figure 3.5: A dangling search with a candidate segment: The points p and q are the only selected points in the depicted slab. The points g_l and g_r are the neighboring guards of the relaxed dangling search with the line defined by a_l and b_r as the candidate.

As a special case it can actually happen that the search ends (there are no further candidate points), but neither the left guard g_l nor the right guard g_r can be selected. Such a situation is depicted in Figure 3.5. Here we allow the tangent line l on A that contains both guards to act as a candidate, defining a collinear pair of weak rays rooted at the midpoint of the segment $\overline{g_l, g_r}$. As this particular pair of rays is not valid (the angle between them is not less than 180 degrees) we have to be ready to detect the special case that l is also a tangent line of B.

To determine the position of a and b relative to $\mathrm{Bd}(B)$ it is sufficient (by Monotonicity Property 2.10 (p. 16)) to know the intersection of the strong rays with $\mathrm{Bd}(B)$.

Requirement 2 (Valid guards)
Let $p, q \in B$ be two neighboring selected points of B with a dangling search Q anchored between p and q. Let e be the right directed strong ray rooted at p and f be the left directed strong ray rooted at q. Let g_l and g_r be respectively the left and right guard of Q. Let e' be the right directed canonical ray rooted at g_l and f' the left directed canonical ray rooted at g_r. Then both intersection points $f \cap e'$ and $e \cap f'$ are outside of $\mathrm{UC}_0(A')$ (i.e. $f \cap e' \cap \mathrm{UC}_0(A') = \emptyset$).

This construction of a certificate, by selected points and dangling searches, shows its strength when we have to maintain it under deletions. Even though we perform searches, we do not necessarily use more than amortized constant time per element. The data structure that makes this all work out is a splitter as presented in Section 3.4.

3.2.6 Half open searches

In the discussion of dangling searches in the previous section, we only tried to select more points between already selected points of one strip. But of course it can also be the case that we can select more points toward the left or right end of that strip. We focus on the right end of a strip of polarity B above A, the left end and the other polarity are completely symmetrical.

We try to find a point x that can be selected, where x is to the right of the rightmost selected point q. Let us say that t is the right directed strong ray rooted at q, as depicted in Figure 3.6. We define the left directed weak ray w that is rooted at the right delimiting equality point v of the strip. We choose w such that it has the highest possible slope, which we can achieve by taking the left neighbor i of v on the hull $\mathrm{UH}(A)$ and choose the direction of w such that it contains i. We call this geometric situation a *half open search*, and i its *endpoint*.

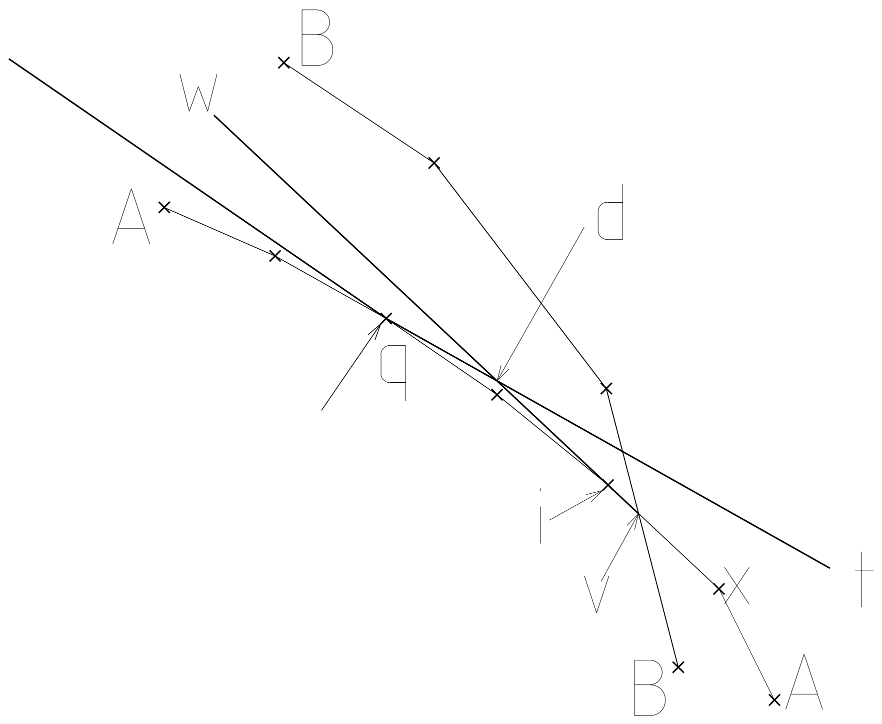

Figure 3.6: A half open search: q is the rightmost selected point in the strip, v is the equality point, t the relevant strong ray, and i is the point defining the direction of the weak ray. The intersection d of t and the weak ray shows that the half open search is relaxed

Now we determine the intersection point d of the strong ray r and the weak ray w. If $d \in \mathrm{UC}(B)$, we can conclude that selecting i would violate the strong ray condition with p and that therefore no point x can be selected between p and v. We say that we have a *relaxed half open search*, which means that it is a certificate of the fact that no further point can be selected.

It can easily happen that $i = q$, which means that the half open search is empty and trivially relaxed.

If $d \notin \mathrm{UC}(B)$, the point i meets the strong ray separation (Requirement 1, p. 26) to the left. Additionally i is neighboring the equality point v, which implies that the right directed canonical strong ray rooted at i leaves $\mathrm{UC}(B)$ at v. As v and its right neighbor x on $\mathrm{UH}(A)$ are not allowed to be selected because they are not inside $\mathrm{UC}_0(B)$, the point i meets the strong ray separation requirement to the right.

In this situation we select the point i. This creates a new, relaxed empty half open search between i and v, and gives rise to a new dangling search between q and i.

3.2.7 Shortcuts

Selecting points of A in a strip S where B is above A, we achieved some sampling on A. Now it remains to introduce the sampling on B, which is done by creating *shortcut*s. Remember that one intuitive goal of the combined sampling process is to make sure that we have for every slab roughly as many sampled points from B as we have from A. At this stage it is hard to say precisely why this is an advantage. As it is usual already, the names A and B can (and have to) be interchanged to yield the symmetric statements.

Geometrically a shortcut is defined by a line l in the plane and consists of the line segment defined as $l \cap \mathrm{UC}(B)$. Intuitively it is understood as a chord of $\mathrm{Bd}(B)$ that "cuts away" all the points above l. An intuitive candidate for such a shortcut is the line defined by two consecutive intersections of $\mathrm{Bd}(B)$ with strong rays rooted at neighboring selected points of $\mathrm{UH}(A)$. Sometimes we actually use precisely this shortcut, but we have to be a little more careful.

For a non-vertical line l we define h_l to be the closed half plane below l. Now we can define for a set of lines L the *shortcut version* of B with respect to L by the extreme points of some well structured convex region in the plane,

$$\mathrm{SC}_L(B) = \mathrm{UH}\Big(\mathrm{UC}(B) \cap \bigcap_{l \in L} h_l\Big).$$

If the set L consists of shortcuts, we can define the under-approximation (sampled version) B' of B by setting $B' = \mathrm{SC}_L(B)$. We also define $s_l = l \cap \mathrm{UC}(B)$, the segment of $\mathrm{Bd}(B')$ defined by l. The points of B' on l (the points introduced by l) are called *cutoff points*. This situation is depicted in Figure 3.7.

We think of applying shortcuts once and for all times, that is, the next time we have to find intersections of strong rays with $\mathrm{Bd}(B)$ we work with $\mathrm{Bd}(B')$ instead. Now the monotonicity of the shrinking of $\mathrm{UC}(B)$ immediately implies that $\mathrm{UC}(B')$ is also monotonically shrinking.

Property 3.7
Let B_1 and B_2 two finite subsets of the plane such that $\mathrm{UC}(B_2) \subseteq \mathrm{UC}(B_1)$. Let l be a line (a shortcut) in the plane. Then we have that the shortcut defined by l on B_2 is a subset of or equal to the shortcut defined by l on B_1.

We would like to not remove shortcuts. Unfortunately this could lead to auxiliary points whose coordinates depend on arbitrarily many input points. To avoid this (to achieve an algorithm in the order-k-branching machine) we delete shortcuts if they are no longer relevant for the other parts of the construction. More precisely we will never delete shortcuts that are intersected with strong rays, which ensures that the intersection points on strong rays still move downwards monotonously over time.

Even though B' is defined for all kinds of sets of lines L, it only leads to something useful if we insist on some requirements.

The first condition we have to require from a shortcut is that it does not introduce intersections with A.

Requirement 3 (Conservative Shortcuts)
A shortcut l on B may not introduce an equality point with $\mathrm{Bd}(A)$, that is, $l \cap \mathrm{UC}_0(B) \cap \mathrm{UC}(A) = \emptyset$.

Lemma 3.8
Let L be a set of shortcuts where every $l \in L$ satisfies Requirement 3 (p. 30). Then we have $\mathrm{Bd}(A) \cap \mathrm{Bd}(B) = \mathrm{Bd}(A) \cap \mathrm{Bd}(\mathrm{SC}_L(B))$, i.e., there are the same equality points between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ as there are between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(\mathrm{SC}_L(B))$.

Proof: An equality point between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(\mathrm{SC}_L(B))$ that is not an equality point between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ has to lie on a line $l \in L$ and on $\mathrm{Bd}(A)$. This is impossible by Requirement 3. An equality point between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ is also an equality point e between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$, as a line conforming with Requirement 3 cannot cut away e. □

We do not want to carry around shortcuts that have no effect.

Requirement 4 (Effectiveness)
Let l be a shortcut of B. Then we require $l \cap \mathrm{UC}_0(B) \neq \emptyset$

If a shortcut l is no longer effective in the sense of Requirement 4 (p. 31), we remove l from the set of shortcuts for B. This implies in particular we do not have a segment of $\mathrm{Bd}(B)$ and a shortcut of B that are on one line, the intersection of a segment and a shortcut is always a point or empty.

Requirement 5 (Shortcut Separation)
Let l and f be two shortcuts on B. Then we require that $l \cap f \cap \mathrm{UC}_0(B) = \emptyset$.

This requirement will ensure that a single deletion can affect at most 3 different shortcuts.

Lemma 3.9
Let l be an effective shortcut on B, i.e., l complies with Requirement 4 (p. 31). Then the two cutoff points of l, i.e., the intersections of l with $\mathrm{Bd}(B)$ are on two different segments of $\mathrm{Bd}(B)$.

Two consecutive segments of $\mathrm{Bd}(B)$ can intersect at most 3 different shortcuts.

Now we have to anticipate some problems that only occur in the dynamic setting, when we continue to create more and more shortcuts. Even though our algorithm will only introduce shortcuts that do not intersect strong rays, it can easily happen that the intersection of a new strong ray with B' lies on a shortcut.

Sampling usually implies that B' has fewer points than B. With our definition this is not necessarily true, but we achieve $|B'| = O(|B|)$ by the requirements we impose on shortcuts, in particular Requirement 5 (Separation).

It remains to make sure that the shortcuts actually simplify the locally outer hull enough to be useful in the (accounting of) the data structure.

Requirement 6 (Aggressive shortcutting)
Let s be a consecutive sequence of segments from $\mathrm{Bd}(B')$, such that s does not intersect a strong ray rooted at A or contain an equality point. Then s consists of at most 4 segments.

This requirement can always be achieved by placing one more shortcut between two consecutive intersection/equality points. We only have to be careful and respect the shortcut separation requirement, which is possible if there are sufficiently many segments. We will discuss how to create such shortcuts in Section 3.6.15.

3.2.8 The truss

The geometric construction we used as a certificate that there are no further equality points of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ is called the *truss*, since it resembles (vaguely) the appearance of an old style iron truss bridge, where iron rods are used to hold the supporting beams apart.

More precisely the truss stands for the data structure and construction of selected points, strong rays, half open and dangling searches, candidates, equality points and shortcuts. If we refer to the situation where all searches are relaxed (cannot be advanced and certify that we identified all equality points), we talk about the *relaxed truss*. We say that we have a relaxed truss for the merge of $\mathrm{UH}(A)$ and $\mathrm{UH}(B)$ in the dynamic setting, if the following requirement is met. We will augment this geometric requirement in Section 3.5.3 by some requirements about how the geometric situation is represented in the data structure. All together this has the flavor of an induction hypothesis: The requirements describe the state of the data structure before we process the deletion of a point, and the state we hence should reach after processing the deletion. Even though this usage of the requirements is in the dynamic setting, the requirement itself is static. In particular it is a description of the state of the truss achieved by the static algorithm.

Requirement 7 (Relaxed truss)
Let A and B be two finite sets of points in the plane. Assume we have sets of selected points $Q_A \subseteq \mathrm{UH}(A)$ and $Q_B \subseteq \mathrm{UH}(B)$ with the sets of strong rays R_A and R_B, and the sets of shortcuts H_A and H_B and the set of identified equality points E. This construction forms a relaxed truss if the following conditions are met:

1. *$E \subseteq \mathrm{Bd}(A) \cap \mathrm{Bd}(B)$, and every strip between two neighboring equality points in E has a polarity.*

2. *R_A forms valid pairs of rays on $\mathrm{Bd}(A)$ and these pairs of rays are rooted at the points of Q_A.*

3. *R_B forms valid pairs of rays on $\mathrm{Bd}(B)$ and these pairs of rays are rooted at the points of Q_B.*

4. *H_A and H_B conform to Requirement 3 (p. 30) (Conservative), Requirement 4 (p. 31) (Effectiveness), Requirement 5 (p. 31) (Separation), and Requirement 6 (p. 31) (Aggressive shortcutting).*

5. *$Q_A \subset \mathrm{UC}_0(B)$ and $Q_B \subset \mathrm{UC}_0(A)$.*

6. *R_A and R_B and conform to the strong ray separation requirement (Requirement 1, p. 26), with reference to $A' = \mathrm{SC}_{H_A}(A)$ and $B' = \mathrm{SC}_{H_B}(B)$.*

For one strip of polarity A below B between the identified equality points $u, v \in E$ we have:

1. *Let L be the list of points of $\mathrm{UH}(A)$ between u and v. If L is not empty, at least one of the points of L is selected.*

2. *Between two selected points there is a relaxed dangling search.*

3. *Between u (v) and the leftmost (rightmost) selected point $p \in \mathrm{UH}(A)$ is a relaxed half open search.*

The symmetrical conditions are valid for all strips of polarity B below A.

Lemma 3.10 (Effectiveness of the truss)
Let A, B be two finite sets of points in the plane and $Q_A \subseteq \mathrm{UH}(A)$, $Q_B \subseteq \mathrm{UH}(B)$, R_A, R_B, H_A, H_B, and E form a relaxed truss as described in Requirement 7. Then all equality points of A and B are identified, i.e. $\mathrm{Bd}(A) \cap \mathrm{Bd}(B) = E$.

Note that we only use some of the statements of Requirement 7 (p. 32) in the proof of the lemma.

Proof: We consider the hinted at approximations of the two upper hulls that result from the sampling. As we already introduced the under-approximation B' of B, we only have to define the over-approximation $\hat{A}$ of A. More precisely we define $\hat{A}$ such that we have $\mathrm{UC}(A) \subseteq \mathrm{UC}(\hat{A})$ and for a strip S of polarity A below B, we get $\mathrm{UC}(A) \cap S \subseteq \mathrm{UC}(\hat{A}) \cap S \subset \mathrm{UC}(B') \cap S \subseteq \mathrm{UC}(B) \cap S$. The situation is depicted in Figure 3.7.

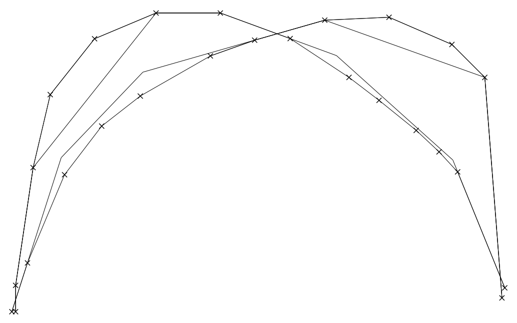

Figure 3.7: Approximate versions of the two merged hulls.

Now we have to define the points that define $\hat{A}$ such that $\hat{A}$ is in upper convex position. Every point of $\mathrm{UH}(A) \setminus \mathrm{UC}_0(B)$ (on A, outside $\mathrm{UC}_0(B)$) are part of $\hat{A}$. These are all the points of $\hat{A}$ that are in strips of polarity A over B. For a strip S of polarity B over A we have one point a of $\hat{A}$ for every left (right) directed strong ray r rooted at a point p on A inside S. This a is given by the intersection of r with the right (left) directed weak ray e, where e is the weak ray of the dangling or half open search next to r. This completes the definition of $\hat{A}$.

Now we should argue that we indeed have $\mathrm{UC}(\hat{A}) \supseteq \mathrm{UC}(A)$. Let $x, y \in \hat{A}$ be neighbors in $\hat{A}$. Then the lower half plane defined by the line l through x and y contains A: If x and y are on a pair of valid rays (rooted at a selected point or a candidate) this follows from the validity condition of the rays. If x and y are both in $\mathrm{UH}(A)$ then x and y are both not selected. If x are in the same strip of polarity A above B, and they are neighboring in $\mathrm{UH}(\hat{A})$ they are also neighboring in $\mathrm{UH}(A)$. If there is another strip of polarity A below B between x and y, the upper border of A in this strip consists of only one segment and no points of $\mathrm{UH}(A)$ (Requirement 7 (p. 32), first statement about strips). This segment is $\overline{x}, y$ and x and y are neighboring on $\mathrm{UH}(A)$. If x is in $\mathrm{UH}(A)$ and y is not they are in different, but neighboring strips, x in an equality strip or a strip of polarity A over B, and y in a strip W of polarity A below B. Let u be the equality point between x and y (left to right) that bounds the strip W. Now there is no point of $\mathrm{UH}(A)$ between x and u. The point y is defined by a half open search, the weak ray defining y is a tangent on x. In particular $\overline{x, y}$ is part of a tangent on A. So we conclude $\mathrm{UC}(A) \subseteq \mathrm{UC}(\hat{A})$.

As we have $\hat{A} \subset \mathrm{Bd}(\hat{A})$, and $\hat{A} \cap S$ contains the two equality points and otherwise only weak ray versus strong ray intersections of relaxed searches, we get $\hat{A} \cap S \subset \mathrm{UC}(B')$.

If a strip extends to infinity and does not contain points from one of the hulls, we look at a trivial special case.

Note that we have for the vertical line l through an equality point $\mathrm{UC}(A) \cap l = \mathrm{UC}(\hat{A}) \cap l = \mathrm{UC}(B') \cap l = \mathrm{UC}(B) \cap l$. □

Even though $\hat{A}$ (and $\hat{B}$) can easily be extracted from the data structure we maintain, it is in our intuition only necessary to reason about the certificate, we do not work with it as explicitly as with shortcuts forming the under-approximations A' and B'.

3.3 The geometry of the dynamic algorithm

In this section we consider the geometric properties that are relevant for the semidynamic merging structure that implements a data structure according to the interface defined in Section 3.1.

3.3.1 Deletions on the overall upper hull

We will impose the condition that points are only deleted, when they are on the overall upper hull. We have to argue that this is no loss of generality. This is done in Section 3.5. What we gain with this constraint is stability of the locally inside upper hull of a strip. We do not have to think about what to do when a selected point gets deleted, and how to reestablish something in its vicinity.

3.3.2 Continuous changes

Now, as we have seen the static geometric construction of our data structure, we have to focus on how to maintain the construction when points get deleted. As already pointed out, we assume that only points of the overall convex hull get deleted. This section is meant to provide the intuition why the truss is a reasonable construction, and identifies all equality points that appear. Therefore we keep the focus on the “standard” cases, only later we will explain the data structure in all details.

In this section our focus is still not on the data structure aspects, but on an efficient geometric construction. In this spirit we for example make sure that we select every point at most once. More precisely the life cycle of a point p in our merging structure is that it becomes part of hull A (say), but is inside $\mathrm{UC}(B)$. Then because of some deletions on B we choose to select p. Only when p *surfaces*, i.e., when B changes further such that $p \notin \mathrm{UC}(B)$, we deselect p. Later p can become part of $\mathrm{UH}(A \cup B)$ and finally it can be deleted.

Geometrically we can think of the deletion of a point $r \in B_1$, leading to the set $B_2 = B_1 \setminus r$, as one smooth change of $\mathrm{Bd}(B)$. To make this precise we take some point O inside of $\mathrm{UC}_0(B_2)$, and move r linearly towards O. This assures that r is no longer part of $\mathrm{UH}(B)$ at the end of the move. More precisely we define the linear function on $[1, 2]$ by $r(t) = (2-t) \cdot r + (t-1) \cdot O$, and $B(t) = B \setminus \{r\} \cup \{r(t)\}$. This defines also $\mathrm{UH}(B(t))$, and more importantly $\mathrm{Bd}(B(t))$. Now we have $B_1 = B(1)$ and $B_2 = B(2)$. For $t_1 < t_2$ we have $\mathrm{UC}(B(t_1)) \supseteq \mathrm{UC}(B(t_2))$. Note that the continuous change of the motion is restricted to two segments at a time, revealing more and more new points of B, until finally the two moving segments level out and become one segment. We describe how to maintain (the interesting part of) a valid truss during this motion.

We use the truss as a certificate that all intersections between $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ are identified (Lemma 3.10). The truss is only a valid certificate if all dangling and half open searches are relaxed and the weak ray intersections (the points a and b in Figure 3.4 and the point d in Figure 3.6) are inside the other hull. If we maintain the intersections of $\mathrm{Bd}(B)$ with strong rays rooted at $\mathrm{UH}(A)$, we are not only in the position to argue for the fact that these intersection points are inside $\mathrm{UC}(B)$, we also get a certificate that a selected point r is inside $\mathrm{UC}(B)$ (or $\mathrm{UC}(A)$ for the other type of half open search; in this situation it is d that is moving). As long as the continuous change of $B(t)$ does not result in any of the weak-ray intersections or a selected point to no longer be inside $\mathrm{UC}(B(t))$ (the point surfaces), there is no need to change the truss. It will continue to be a valid certificate that we identified all equality points. In return, we have to discuss how the truss can be adjusted in the case when one of these points changes from being inside the other upper hull to being outside. For the simplicity of this discussion (it is for the intuition only anyway), we assume that it never happens that two points of the truss are affected by such a change at the same time. We will use t_1 to denote a time directly before this single change and t_2 for a time directly after this change. In other words between t_1 and t_2 happens precisely one change to the truss. In the following we will discuss how to maintain the truss during these changes.

During the motion we might also see some change to a strip where $\mathrm{Bd}(B)$ is (locally) below $\mathrm{Bd}(A)$. As $\mathrm{Bd}(B(t))$ moves downwards, we know that during the continuous motion no new equality point can arise in such a strip. In general, such a strip increases it horizontal extent. So there is no point in continuously maintaining the truss there. Instead we will extend or establish a new truss just like in the static case described in Section 3.2.3 after the motion is finished. We will not consider this type of strip in the remainder of this discussion.

Advancing searches

The easiest case is, if the intersection point a of a weak ray stemming from a dangling search s surfaces. The names used here are taken from Figure 3.4. Assume w.l.o.g. that a is the intersection point of the right directed strong ray of the left anchor of s, and the left directed weak (canonical) ray of the candidate point c of s. In this situation we advance the dangling search by making the candidate c the new right guard of s. This complies with the valid guard requirement (Requirement 2, p. 28). In general we might have to continue advancing the dangling search (as described in Section 3.2.5) until it is relaxed again. If the candidate of s already was a segment g_l, g_r of $\mathrm{Bd}(A)$ (Figure 3.5, p. 28) and hence s cannot be advanced further, we select the left guard g_l. We have to argue that this is allowed by the strong ray separation requirement (Requirement 1, p. 26). To the left this is clear by the event we currently react to. To the right it follows by g_l being a left guard of s (Requirement 2, p. 28). We instantiate two new dangling searches between respectively p and g_l, and g_l and q. We advance these dangling searches until they are relaxed, just like in the static setting.

A similar case is if the intersection point d of a half open search s surfaces. The names to describe this situation are taken from Figure 3.6 (p. 29). Let $q \in A$ be the selected point of s, let t be the strong ray of s rooted at q, and w the weak ray of s, rooted at the equality point v. We select the endpoint i of the half open search s. As the canonical ray towards the next selected point q is the weak ray w, this is allowed by the strong ray separation requirement (Requirement 1, p. 26) because we now have that the ray intersection d is outside $\mathrm{Bd}(B(t_2))$. We also instantiate a new, empty half open search with the selected point i and the equality point v. This half open search is trivially relaxed. We instantiate a new dangling search between the selected points i and q. We have to advance this dangling search until it is relaxed.

New equality points

For a selected point $p \in A$ the motion of $\mathrm{Bd}(B(t))$ can result in p surfacing. If none of the neighbors of p in A are outside of $\mathrm{UC}(B(t))$, this means that we look at a new strip of polarity A over B. As argued before, we will not establish a truss of this polarity as long as $\mathrm{Bd}(B(t))$ is moving. After that we are back to the static case that is already described. For the strip that gets split, the changes to the truss are just the same as if there was an equality point moving over p (one in each direction, making the strip smaller). This case is discussed in the next paragraph.

Moving equality points

Another typical event is that a point $x \in A$ surfaces, that is, $x \in \mathrm{UC}(B(t_1))$ and $x \notin \mathrm{UC}(B(t_2))$, because an equality point moved over it. This means that the strip of polarity B over A shrinks horizontally. This is easy if x is not selected. Then we merely shrink a half open search in the following way. In the situation and naming of Figure 3.6 (p. 29), this means that the former endpoint i ($x = i$) of the half open search surfaces. Then by our definitions the left neighbor of i becomes the new endpoint. The new weak-ray intersection point d is by monotonicity (Property 2.10, p. 16) closer to the selected point q. By our assumptions about the continuous motion this new half open search is relaxed at time t_2.

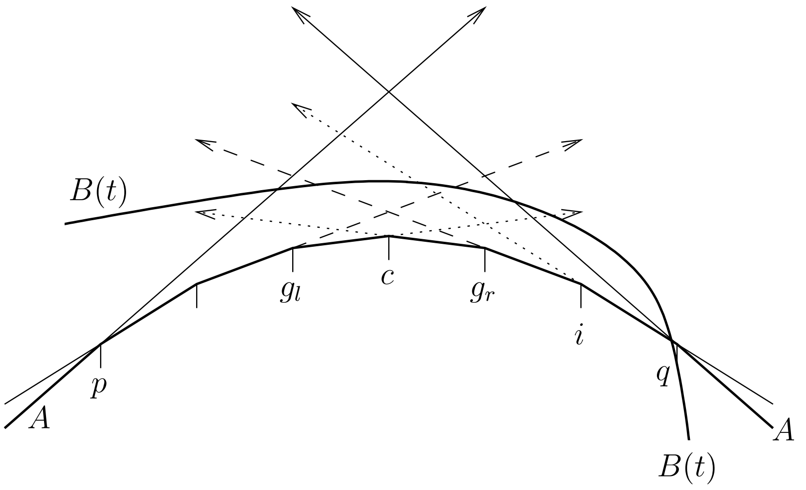

Figure 3.8: The situation of deselecting q. The dangling search was relaxed before q surfaced. After deselecting q we are allowed and required to select g_r.

If $x = q$ is selected, we have to deselect it. This situation is exemplified in Figure 3.8. Deselecting q, we can no longer have a dangling search s having q as the root of one of its strong rays. Let us assume that q is the left endpoint of s, and let i be its right neighbor on $\mathrm{UH}(A)$. By our assumption that the changes to the truss happen one at a time, we have $i \in \mathrm{UC}(B(t_2))$. Now we select the right guard g_r of s. This meets the strong ray condition with the left anchor (selected point) p of s. We instantiate a new dangling search between p and g_r. We instantiate a half open search between g_r and the equality point of $\mathrm{Bd}(B(t_2))$ and $\mathrm{Bd}(A)$ which is now between q and i. If this is should not be relaxed, we select i. Now we are sure to have a relaxed truss again. Note that there is a slightly different situation if $\mathrm{Bd}(B(t))$ moves differently, namely such that the intersection of $\mathrm{Bd}(B(t))$ moves on the left directed strong ray rooted at q towards q and finally over q. Then we would also have first selected g_r after advancing half open searches, and then i also as a result of advancing a dangling search.

If q is the only selected point of its strip W, we select its neighbor i inside W, if such a point exist. Now the half open search of i in the direction of q is empty and therefore relaxed, the other half open search is relaxed because it was with q as the anchor point. If no such point exist, we look at what we call a *trivial strip*, which is always considered relaxed.

3.3.3 The geometry of losing a pair of equality points

A deletion of point $r \in B$ can cause that we lose a strip with polarity B over A, and have to join two strips S_l and S_r (S_l to the left of S_r) of polarity A over B. Here the crucial observation is that the rightmost selected point $p \in S_l \cap B$ and the leftmost selected point $q \in S_l \cap B$ are far enough apart that they meet the strong ray separation requirement. This is where we make use of the global character of Requirement 1 (p. 26). Now we can join the half open searches of q and p into one dangling search. Luckily the geometric properties and the requirements of the data structure implementing dangling searches fit together nicely. We focus here on the geometric properties. The data structure aspects of the situation are considered in Section 3.6.7. It is crucial for the amortized analysis, that this is the only situation where we have to join two strips. The geometry allows us to join two half open searches (those have trivial candidates) into one dangling search with the promise that we will eventually select one of the points between the two points that become guards. The geometric situation (in the continuous setting) is illustrated in Figure 3.9.

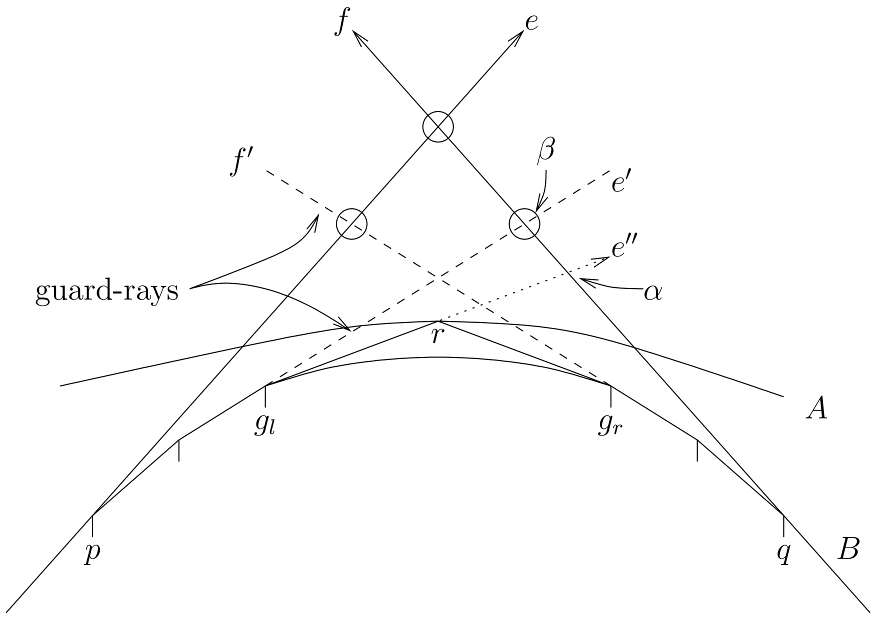

Figure 3.9: Illustrating the situation of joining two strips into one dangling search. The names are used in the proof that β is outside $\mathrm{UC}_0(A')$.

We have to argue that this joining of two half open searches complies with Requirement 2 (p. 28). It will be important for the data structure (the splitter) that we only join in this way, namely with the promise to eventually split again between g_l and g_r.

Lemma 3.11 (Geometric join)
Let $p, q \in B$ be two neighboring selected points of B with a dangling search Q anchored between p and q. Assume there is a point r on $\mathrm{Bd}(A')$ such that $\overline{r, g_l}$ and $\overline{r, g_r}$ are tangents on $\mathrm{Bd}(B)$. Then g_l is a valid left guard of the dangling search Q and g_r a valid right guard of Q.

Proof: By the global character of the strong ray separation Requirement 1 (p. 26) (and because of r), we have that the intersection point of e and f is outside hull A', i.e., we have $e \cap f \cap \mathrm{UC}(A') = \emptyset$.

The situation is depicted in Figure 3.9. We consider the right directed ray e'' rooted at g_l with the direction toward the deleted/moving point r. This is a valid ray (on a tangent line) as we are in the geometric situation that g_l and r are neighbors on $\mathrm{UH}(B \cup \{r\})$. As we have $r \notin \mathrm{UC}_0(A')$, we conclude that the intersection point α of e'' and f outside $\mathrm{UC}_0(A')$. Let e' be the canonical right directed ray e' rooted at g_l and β be the intersection point of e' and f (the strong ray intersection). By ray-monotonicity (Property 2.10, p. 16) the fact $r \in \mathrm{Bd}(A')$ implies $\alpha \notin \mathrm{UC}_0(A')$. One more application of the same lemma yields $\beta \notin \mathrm{UC}_0(A')$, the point g_l is a valid left guard of Q. By a symmetrical reasoning we get that g_r is a valid right guard of Q. □

We stop to consider the continuous motion here, as we know that it is impossible that more equality points come into existence. We already have seen how to relax the newly created dangling search in Section 3.2.5.

There is actually one more aspect to this join of two strips. Not only do we have to join data structures representing the lower hull, we also have to join those data structures holding the upper hull. We need to store the upper hull in a data structure that allows searches. When we select a point $p \in A$, we create strong rays, one of them t. Now we need to be in the position to determine the intersection of t with $\mathrm{Bd}(B)$. This amounts to a search on the segments of $\mathrm{Bd}(B)$. To avoid a possibly expensive join operation, we use the aggressive shortcutting requirement (Requirement 6, p. 31). It guarantees that we have to extend the search data structure for the surfaced points of $\mathrm{UH}(A)$ only by a constant number of points of A' that are already part of the truss. With this modification the refining searches implemented by a splitter (a data structure we will discuss in detail in Section 3.4) cause only linear work. The precise analysis how shortcuts and splitter achieve this goal is part of the detailed exposition of Section 3.5.

3.3.4 Geometry of a deletion

Algorithmically it would be a waste to simulate the smooth movement presented in Section 3.3.2. Instead we investigate how much change to the truss can be necessary because of a single deletion. This sets the stage for the algorithm of Section 3.5 that adjusts the data structure in one go to the change. This also means that we will not necessarily find the same state of truss that would be reached by the smooth movement. The goal of this algorithm is to establish a relaxed truss as introduced in the previous sections and made precise in Requirement 7 (p. 32). Again this discussion focuses on the geometry of the situation. We still ignore data structure aspects for now, and take it for granted that we can easily find all kinds of intersection points. We will address the data structure aspects in Section 3.5.

Lemma 3.12
Let S_1 be a finite set of points in the plane, $r \in \mathrm{UH}(S_1)$, and $S_2 = S_1 \setminus \{r\}$. Then we have the following statements:

1. $H_1 := \mathrm{UH}(S_1) \setminus \{r\} \subseteq \mathrm{UH}(S_2) =: H_2$.

2. *Considering H_1 and H_2 as left to right ordered sequences we have that the point r is replaced by a (possibly empty) sequence of points $p_1, \ldots, p_k$. The points $p_1, \ldots, p_k$ are called* fresh points.

3. *Let l be a line in the plane. Let s_l and s_r be the two segments of $\mathrm{Bd}(S_1)$ that contain the point r. Then the shortcut defined by l on S_1 can only be different from the shortcut defined by S_2, if l intersects s_l or s_r.*

4. *Let H be a set of lines, such that the shortcuts defined on S_1 are separated by vertical lines. Then at most 3 consecutive shortcuts of H are changed from $\mathrm{SC}_H(S_1)$ to $\mathrm{SC}_H(S_2)$.*

5. *Let H be a set of shortcuts that complies with the shortcut separation requirement (Requirement 5, p. 31) on S_1 (and therefore on S_2 as well). Let f be a ray that is rooted at some point inside $\mathrm{SC}_H(S_1)$. Assume that f leaves $\mathrm{UC}(S_1)$. Then the intersection of f with $\mathrm{Bd}(\mathrm{SC}_H(S_2))$ can be different from the intersection of f with $\mathrm{Bd}(\mathrm{SC}_H(S_1))$ only if f intersects the segment s_l or s_r.*

Proof: Statement 1 of the lemma amounts to that no other point but r is deleted from $\mathrm{UH}(S_1)$, i.e., $\mathrm{UH}(S_1) \setminus \{r\} \subseteq \mathrm{UH}(S_2)$. By the characterization of points in $\mathrm{UH}(S_1)$ as extreme points in some direction (Property 2.4, p. 15) this is trivial.

Statement 2 follows from the observation that two points $q_1, q_2 \in \mathrm{UH}(S')$ that are consecutive in $\mathrm{UH}(S_1)$ are also consecutive in $\mathrm{UH}(S_2)$. This follows from the fact that segments of $\mathrm{UH}(S_1)$ exist also in $\mathrm{UH}(S_2)$ as long as both endpoints are also in S_2. This is again a consequence of the extreme point lemma (Property 2.4 (p. 15)).

Statement 3 follows again from the fact that all other segments of $\mathrm{Bd}(S_1)$ exist as well in $\mathrm{Bd}(S_2)$; Statement 4 is an immediate consequence of the previous statement.

For Statement 5 let a be the intersection point of f with $\mathrm{Bd}(\mathrm{SC}_H(S_1))$, and assume that the intersection of f with $\mathrm{Bd}(\mathrm{SC}_H(S_2))$ is different from a (or does not exist). Then we can conclude that $a \in \mathrm{UC}(S_1) \setminus \mathrm{UC}(S_2)$, which implies that a is inside the triangle formed by s_l and s_r. This immediately implies that f intersects s_l or s_r. □

Additionally to these statements about the deletion of one point and its effect on a single upper hull, we want to investigate how much change a single deletion can cause to the truss between $\mathrm{UH}(A)$ and $\mathrm{UH}(B)$.

One of the design goals of the truss is locality. A single change at one place should not require any global adjustments. The following lemma makes this vague statement precise for the situation of a single deletion.

Lemma 3.13 (Limited impact of a deletion)
Let A, B_1 be two finite sets of points in the plane and $Q_A \subseteq \mathrm{UH}(A)$, $Q_B \subseteq \mathrm{UH}(B)$, R_A, R_B, H_A, H_B, and E form a relaxed truss as described in Requirement 7 (p. 32). Let $r \in \mathrm{UH}(B_1)$, and $B_2 = B_1 \setminus \{r\}$.

Then there are at most 4 consecutive rays R' of R_A that have different intersections with $\mathrm{SC}_H(B_2)$ than with $\mathrm{SC}_H(B_1)$. The root points $Q' \subseteq Q_A$ of the rays in R' are at most 5 consecutive (selected) points in Q_A.

Proof: We call a point a a *strong ray certificate* of A if it is the intersection point of two strong rays of opposite direction, rooted at two neighboring selected points q_1 and q_2. ($q_1, q_2 \in Q_A$ and no point of $\mathrm{UH}(A)$ between q_1 and q_2 is selected). By the strong ray separation requirement (Requirement 1, p. 26) we have $a \notin \mathrm{UC}(B_1')$, where $B_1' = \mathrm{SC}_{H_B}(B_1)$. Let X be the set of all segments of the type $\overline{q_1, a}$ or $\overline{q_2, a}$. The segments of X form a connected polyline (across all strips) and X has a natural left to right ordering. Let l be a tangent line on $\mathrm{UC}(B_1)$. By the definition of a selected point we have $Q_A \subset \mathrm{UC}_0(B_1)$. Hence we have $l \cap \mathrm{UC}_0(Q_A) = \emptyset$. Let h be the tangent on $\mathrm{UH}(Q_A)$ that is parallel to l. Let $q \in Q_A$ be the (single) touching point of h. Let q_l and q_r be the left and right neighbor of q in Q_A. This situation is exemplified in Figure 3.10. Let t be a right directed valid ray on q_r. Now the slope of h is higher than the slope of $\overline{q, q_r}$ which in turn is higher than the slope of r. It is therefore impossible that the ray t intersects the line l. Symmetrically no left directed valid ray on q_l can intersect l. If there is a segment of $\mathrm{Bd}(Q_A)$ that is parallel to l, then we define q_r and q_l to be the endpoints of this segment. Because selected points of A are strictly inside $\mathrm{UC}_0(B_1)$, the line l cannot be a tangent line

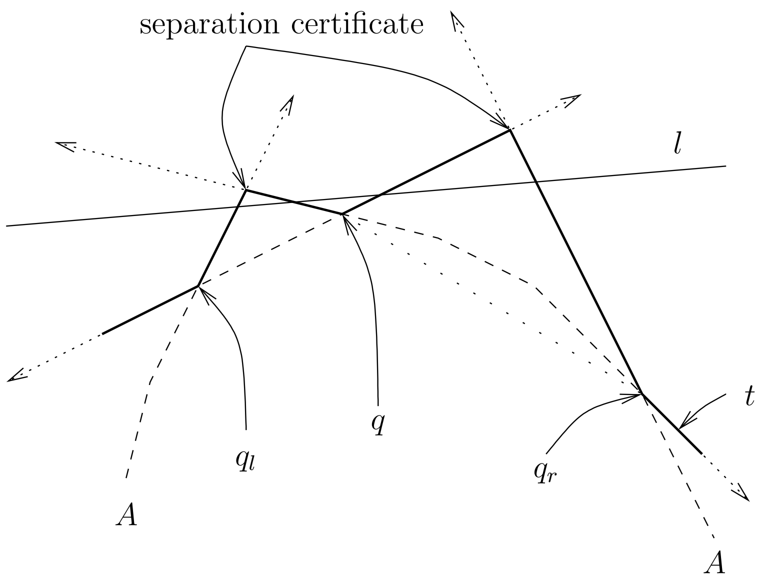

Figure 3.10: An example for the situation in the proof of the limited impact lemma (Lemma 3.13). The set X is drawn with solid segments. $\mathrm{Bd}(A)$ is drawn in dashed line-segments, $\mathrm{Bd}(Q_A)$ in dotted segments. The strong rays are depicted as dotted arrows.

of $\mathrm{UH}(Q_A)$. Again we conclude that no left directed ray of q_l and no right directed ray of q_r can intersect l. By the ray monotonicity lemma (Property 2.10, p. 16) we conclude that no right directed strong ray to the right of q_r and no left directed strong ray rooted to the left of q_l can intersect l. Therefore at most two strong ray certificates of A are above the line l.

Let s_l and s_r be the two segments of $\mathrm{Bd}(B_1)$ that are incident to r. The line segments of X that are intersected by s_l and s_r are consecutive: otherwise there would be a strong ray certificate c such that c is below s_l and s_r which means that c is inside $\mathrm{UC}_0(B_1)$. We conclude that only 4 consecutive strong ray certificates are above the lines defined by s_l and s_r.

By Lemma 3.12 (p. 38), Statement 5 only strong rays that intersect s_l or s_r can have a different intersection with $\mathrm{Bd}(B_2)$ than with $\mathrm{Bd}(B_1)$. We conclude that at most 8 strong ray intersections and 5 selected points (2 with only one of their rays, 3 with both rays) can be affected by the deletion of r. □

3.3.5 Requirements on the development of the truss

Here we formalize the intuition that the truss is "moving downward" only. If there where no shortcuts this would be a trivial as the upper hull of A, and that of B can only move down because of deletions. Even if we have shortcuts, but we do not delete them, this monotonicity is basically trivial. Only when we allow the algorithm to remove shortcuts, we need the following requirement to ensure that this non-monotonicity is invisible to the data structure.

In the following the names A and B are completely symmetrical, a statement or requirement we formulate for A and B is also true or required for B and A.

Requirement 8 (Monotonicity on strong rays)
Let f be a strong ray that is rooted at the selected point $p \in A$. Let $t_1 < t_2$ be two points in time. Let point i_1 and i_2 be the intersection of f with $\mathrm{Bd}(B')$ at time t_1

and time t_2 respectively. Then i_2 is closer to p than i_1, i.e., the boundary moves inward on the strong ray.

Property 3.14
Let e and f be two strong rays that are part of the data structure at time t. Assume e and f fulfill the strong ray separation Requirement 1 (p. 26) at time t. Assume the changes to A' and B' fulfill Requirement 8 (p. 40). Then we have that at all times after t the strong rays e and f fulfill Requirement 1 (p. 26).

Property 3.14 implies that it is sufficient to check for the strong ray separation requirement when we select the point. If we make a point c a guard in a dangling search N between the selected points p and q, it is sufficient to check the weak ray versus strong ray intersection when we make c a guard. If we do not select further points between p and q, the point c will stay a valid guard, as hinted at in the outlook to the dynamic setting in Section 3.2.5.

3.4 Splitter

A *splitter* is a data structure that contains a list of elements from a totally ordered universe. We assume that evaluating the order relation between two elements can be performed in $O(1)$ time. As the name suggests, the main purpose of a splitter is that we can split it into two smaller splitters. The interface of the data structure that we actually want to use is defined in detail in Section 3.4.2. Let us first get some intuition what a splitter is and how it might be useful in our context.

Hoffmann, Mehlhorn, Rosenstiehl, and Tarjan describe in [HM$^+$86] an algorithm that sort Jordan sequences in linear time. One of the core components of this algorithm is a data structure that can also be used as a splitter. To give some impression, why such a data structure is at all feasible, we consider in the following the implementation of a restricted version of a splitter, i.e., a data structure that only supports the operations build and split.

We can implement the restricted splitter using a (2,4)-tree, performing "exponential" searches from the first or last element of the current set. More precisely we start the search by deciding whether the position p we are searching for is to the left or right of the root. Then the search path starts at the leftmost (or rightmost) leaf of the tree, and goes upward in the tree, until we look at an element e that is to the right (left) of p. From there the search path continues to a leaf l, following the standard rules of predecessor searching in a (2,4)-tree. Now we split the (2,4)-tree at l. This combined search and split operation takes time proportional to the logarithm of the smaller resulting splitters.

For the amortized analysis we charge every search and split operation entirely to the smaller resulting splitter, distributing its cost equally over the elements. Now we account for every single element, how much search cost accumulated for it. Between two consecutive searches that get charged partly to an element, we know that the size of the resulting splitter is smaller than half of what it was before. That is, if the search costs $O(i)$ units of time, then the splitter had size roughly 2^i and we look at the following type of expression for the total amortized cost per element:

$$\sum_{i=1}^{\infty} \frac{i}{2^i} \leq 2.$$

As the splitter we want to use is somewhat more complicated (restricted insertion and deletion, a very special join, suspending searches) there is no point in making this analysis more precise here.

3.4.1 Dangling searches

As we have already seen in Section 3.2.5, it can happen that we cannot finish a search because the geometric situation is not (yet) conclusive. We can neither afford to finish the search immediately and perform the split (this would conflict with the geometric requirements we want to keep) nor can we afford to forget about this partial search (this would spoil the amortized analysis of the splitter). So we have to really suspend the search and encapsulate the state of the partially performed search on a splitter S. We do this by introducing the current candidate c, and the current right guard g_r and left guard g_l. We used these names already in the setting of geometric searches of Section 3.2.5. The guards mark elements that are already excluded from being the element we search for: everything to the left of g_l and everything to the right of g_r cannot be the element of S we are searching for. For the analysis this is mainly the promise to eventually split between (including) g_l and g_r, as these are the extreme outcomes of possible continuations of the search. We can declare the current candidate c to be the new left (or right) guard of the search. This is the only way the search can make progress. A search finishes if there are no further elements between g_l and g_r that could be the next candidate. Alternatively we can abort the search if we keep the promise of splitting between g_l and g_r. We do this typically because c, g_l or g_r are the element (or one of the elements) we are looking for.

To give some motivation for the definitions we give here, we shortly remind the geometric significance of a guard when we want to achieve a dynamic version of the algorithm from Section 3.2.5: the left guard is a point that would meet the strong ray invariant (requirement) against the next selected point to the right, but is not (yet) allowed to be selected because of the selected point to the left. If we later find out that (after a deletion) the left strong ray invariant is met, we can select the left guard without any further examination of the geometry, and fulfill the promise of splitting S.

3.4.2 The real splitters

A splitter consists of elements drawn from a completely ordered universe, stored in a level-linked (2,4)-tree. Additionally it has three pointers to elements, namely the *candidate* and the *left guard* and the *right guard*. We think of the splitter having an atomic operation SEARCH that can be suspended to achieve a dangling search. That means that a splitter can be in a state, where it has no active dangling search. In this state all three pointers (guards and candidate) are nil. If the two guards point to neighboring elements, the candidate pointer is nil. We keep the invariant that the left guard points to an element that is smaller or equal to the element the candidate points to, which in turn is smaller or equal to the element the right guard points to. Additionally the left and right guard have to point to distinct elements. The left (right) guard can be nil, which is understood as pointing to some special element of the universe that is smaller (bigger) than all the elements in the dictionary.

All operations we describe are destructive, the data structure is permanently changed by the execution of the operation, the previous state of the data structure is no longer accessible.

BUILD $(e_1, \ldots, e_k)$ Returns (a pointer to) a new splitter containing the elements $e_1, \ldots, e_k$, provided $e_1 < e_2 < \cdots < e_k$. There is no dangling search active on this new splitter.

EXTEND (S, e) Extends the splitter S that contains the elements $e_1, \ldots, e_k$ to the splitter $e, e_1, \ldots, e_k$ or $e_1, \ldots, e_k, e$, provided that either $e < e_1$ or $e_k < e$, and that there is no dangling search active on S.

Shrink_left/right (S) The splitter S is changed by deleting the leftmost/rightmost element that is stored in S, provided that there is no dangling search active on S.

Instantiate_dangling_search (S) It is required that the splitter S has no active dangling search before this procedure is called. The guard pointers remain at value nil. The candidate pointer is set to the element c of S according to the search algorithm of the dictionary. (one of the elements stored at the root-node of the (2,4)-tree).

Advance_dangling_search_left/right (S) It is required that the splitter S has an active dangling search. The left (or right) guard is changed to point to the element the candidate pointer is currently pointing to. A new candidate element is determined according to the search procedure in the (2,4)-tree. I.e., we disallow all elements to the right (or left) of c as possible outcomes of the dangling search.

Split (S, w) The splitter S is split into two splitters S_1 and S_2 according to the value of w, which is either *left guard*, *candidate* or *right guard.* The pointed to element is usually not member of any of the resulting splitters. Only if the guards point to neighboring elements and the candidate pointer is nil, every element of S is moved to either S_1 or S_2. The splitters S_1 and S_2 have no active dangling search.

Join $(S_1, (e_1, \ldots, e_k), S_2)$ The splitters S_1 and S_2 are required to not have active dangling searches. The elements $e_1, \ldots, e_k$ (possibly an empty list) are ordered, and e_1 is to the right of the rightmost element a stored in S_1, and e_k is to the left of the leftmost element b stored in S_2. In particular a is to the left of b. That is, we have $a < e_1 < e_2 < \cdots < e_k < b$. The splitters S_1 and S_2 are destroyed and a new splitter S is created. The splitter S holds all elements from S_1, S_2 and the new elements $e_1, \ldots, e_k$. It has an active dangling search, where the left guard points to a, the right guards points to b, and the candidate is chosen according to a (binary) search over $e_1, \ldots, e_k$. It is allowed that S_1 or S_2 or both are empty. In this case the corresponding guard pointer is nil.

This join operation seems to spoil the principle of splitters being in some sense monotonous, which is crucial to the analysis. Therefore we do not implement this operation literally as a join of the (2,4)-trees. We rather take it as a wrapper for a delayed extension of S_1 and S_2. Remember that instantiating the dangling search in the situation of the join has the promise built in that we will split at one of the elements $a, e_1, \ldots, e_k, b$ before we perform another join operation with this splitter. What we actually do is to place $e_1, \ldots, e_k$ in an auxiliary dictionary and use this to guide the dangling search. Only when this search is settled with a split operation, we extend S_1 (and S_2) with the elements left (and right) of the split point. So we are in the position to meet the interface of the split operation. We say that the splitter is in the forked state as opposed to the normal state. Note that the requirement that S_1 and S_2 have no active dangling search immediately implies that it is sufficient to have two types of splitters (two explicit states of a splitter) with an active dangling search.

Note that it seems not impossible to allow for extend and shrink on splitters that have an active dangling search. Because we do not need this functionality in our application, we also omitted it from the interface here.

3.4.3 Amortized analysis

Theorem 3.15
The operations of the splitter incur the following amortized execution times:

- BUILD, EXTEND, *and* JOIN *take amortized $O(k)$ time where k is the number of participating elements.*
- INSTANTIATE DANGLING SEARCH *takes amortized $O(1)$ time.*
- SHRINK *and* SPLIT *take amortized $O(1)$ time.*
- ADVANCE DANGLING SEARCH *takes a negative constant time in the amortized sense, i.e., it can pay for analyzing a constant sized geometric situation.*

This theorem is a variation of the construction given and analyzed in [HM+86]. For the sake of completeness, we give the parts of the amortized analysis that are beyond the assumptions given in Section 2.3.

Before starting the proof of the theorem, we consider a function that turns out to be helpful when defining a potential function. Consider $f\colon \mathbb{N}^+ \to \mathbb{R}$ given by

$$f(n) = n \cdot \int_0^{\ln n} x \cdot e^{-x} dx\ .$$

Lemma 3.16 (Properties of $f(n)$)

1. *f is monotonically increasing, i.e., for $n' > n$ we have $f(n') > f(n)$.*
2. *f is linearly bounded, i.e., $f(1) = 0$ and $f(n) < n$ for all $n \in \mathbb{N}^+$.*
3. *for all $n \in \mathbb{N}^+$ we have $f(n+1) - f(n) \leq 2$.*
4. *for $n_1, n_2, n \in \mathbb{N}^+$ where $n_1 + n_2 \leq n$ and $n_1 \leq n_2$,*
   we have $f(n_1) + f(n_2) + \frac{\ln 2}{2} \cdot \ln n_1 \leq f(n)$

Proof: We will use several times the following estimate on the value of a definite integral:

$$(b-a) \cdot \min_{x\in[a,b]} g(x) \quad \leq \quad \int_a^b g(x)dx \quad \leq \quad (b-a) \cdot \max_{x\in[a,b]} g(x)$$

Statement 1 follows immediately from the formulation as an integral over a positive function.

We define $p(n) = \int_0^{\ln n} x \cdot e^{-x} dx$, and observe that $P(x) = (-1-x)e^{-x}$ is an indefinite integral of $x \mapsto x \cdot e^{-x}$ which means we have $p(n) = (P(n) + 1)$. This representation of p together with $P(x) < 0$ and $P(0) = -1$ immediately imply Statement 2.

For Statement 3 we observe that $p(n+1) - p(n) < (\ln(n+1) - \ln n) \ln n \cdot e^{-\ln n} < 1/n \cdot \ln n/n < 1/n$. We get $f(n+1) - f(n) = (n+1) \cdot p(n+1) - n \cdot p(n) = n \cdot (p(n+1) - p(n)) + p(n+1) \leq 1 + 1 = 2$. In the situation of Statement 4 we immediately see that $n_1 \leq n/2$, implying $\ln n \geq \ln n_1 + \ln 2$. We observe $f(n) = np(n) \geq (n_1 + n_2)p(n) \geq n_1 p(n) + n_2 p(n_2) \geq f(n_2) + f(n_1) + n_1 \int_{\ln n_1}^{\ln n_1 + \ln 2} x \cdot e^{-x} dx \geq f(n_1) + f(n_2) + \frac{\ln 2}{2} \ln n_1$, since $n_1 \int_{\ln n_1}^{\ln n_1 + \ln 2} x \cdot e^{-x} dx \geq n_1 \ln 2 (\ln(n_1) + \ln 2) e^{-\ln n_1 - \ln 2} \geq n_1 \ln 2 \ln n_1 \frac{1}{2n_1} = \frac{\ln 2}{2} \ln n_1$. □

Proof of Theorem 3.15:. We rephrase the analysis of the data structure of Hoffmann, Mehlhorn, Rosenstiehl, and Tarjan in [HM+86]. For the ease of writing we approximate constants according to the inequality $\ln 2/2 > 1/4$.

Let d' be the constant amortized cost of an insertion into the (2,4)-tree such that deletion are amortized for free, and c' such that a search followed by a split in the (2,4)-tree costs $c'/4 \cdot \ln k$ where k is the size of the smaller subtree resulting from the split operation. This c' is calculated based upon the assumption that one comparison has cost c and includes the overhead of evaluating a comparison with a call to ADVANCE DANGLING SEARCH. We make sure that c' is big enough to also allow us to initialize a search, specifying the first element to compare with. We also define an average internal comparison cost $\hat{c}$ where we know $\hat{c} \geq c$, one comparison step cost in total less than $\hat{c}$ and if s the number of comparisons during a search we have that $\hat{c} \cdot s \leq \frac{c'}{4} \cdot \ln k$.

We define the amortized insertion cost for elements as $d = 2d' + 3c'$.

Let S be splitter containing n elements. Let s be the number of calls to ADVANCE DANGLING SEARCH we performed after instantiating a dangling search. If there is no active dangling search, we set $s = 0$. Then we define the potential of the splitter S to be

$$p(S) = c' \cdot f(n) - \hat{c} \cdot s \; .$$

If the splitter S is in the FORKED state with the sizes n_1, n_2 elements in the former splitters and k fresh elements,

$$p(S) = c'(f(n_1) + f(n_2)) + k(d + 5c') - \hat{c} \cdot s \; .$$

Now we have to argue that with this potential function the operations indeed have the claimed amortized costs.

For the BUILD operation this stems from the fact that inserting the elements into the dictionary costs amortized d' and by Lemma 3.16, Statement 3 we need at most c' per element for the potential function.

The SHRINK is also no problem, there is no cost from the dictionary operation, and the potential function decreases (Lemma 3.16, Statement 1).

For the EXTEND operation we start with a splitter S of size n. We insert the new element into the dictionary at cost d'. The difference in $p(S)$ is given by $c'(f(n+1) - f(n)) < 2c'$ (Lemma 3.16, Statement 3). The total cost of an EXTEND is smaller than d as it is bounded by $2c' + d'$. We will need this tighter bound later.

The INSTANTIATE DANGLING SEARCH OPERATION does not change the potential. The only action that takes place is to make the candidate pointer point to the element that is stored at the root of the (2,4)-tree. By choice of c' this action takes less then c' time, which is by definition less then d.

The operation ADVANCE DANGLING SEARCH can, by the definition of the potential function(s), pay an amount c. Be aware that the functionality of this operation depends slightly on whether the splitter is in the NORMAL or FORKED state.

The SPLIT operation and its analysis depends on the state of the splitter. If we have a NORMAL splitter, we know that the dangling search so far performed (s steps) is bounded from above by $c' \ln n_1$, where n_1 is the size of the smaller resulting splitter. More formally if we split S into two parts S_1 of size n_1 and S_2 of size n_2, where we assume $n_1 \leq n_2$, we have that $\hat{c} \cdot s \leq \frac{c'}{4} \ln n_1$. By the definition of $\hat{c}$ the term $\hat{c} \cdot s$ bounds the cost of searching and splitting the (2,4)-tree. By Lemma 3.16, Statement 4 we know that $c'(f(n_1) + f(n_2) - f(n)) \geq c' \cdot \ln n_1 \cdot \ln 2/2 > \hat{c}s$. This immediately implies $p(S_1) + p(S_2) + \hat{c} \cdot s \leq p(S)$.

If the splitter is in FORKED state, we observe that we have $\hat{c} \cdot s \leq c' \ln k \leq c'k$. This means we have still $d' + 2c'$ potential per element left after we paid for the search. This is sufficient to perform an EXTEND operation for the single elements.

The JOIN OVER DANGLING SEARCH operation puts the k new elements into a dictionary at cost d per element. The remaining cost of $d + 3c'$ per element goes entirely into the potential function. □

3.5 The merging data structure

From now on we want to explicitly allow to have different points that happen to have the same coordinates, we consider multisets of points instead of sets. This is reasonable if we say that a point is no longer identified by its coordinates, but by a pointer that we assign at creation time. Allowing degenerate cases in this way does not make the construction significantly more complicated. It allows us to formulate the variant of the merging data structure we need to achieve linear space as a special case of the general merging data structure, see Section 3.7 (p. 68).

If we prefer identity by coordinates, we can maintain a (2,4)-tree of the lexicographically ordered points on the level of the fully dynamic data structure. There it does not change the asymptotic performance of the data structure.

Theorem 3.17 (Semidynamic Merging Structure)
There exists a data structure that implements the operations as described in Definition 3 (p. 19). Let n be the number of points stored in the set $C = A \cup B$ (not only the size of $\mathrm{UH}(A) \cup \mathrm{UH}(B)$).

The operation CREATE_SET(p) *takes amortized and worst case $O(1)$ time. The operation* MERGE(A, B) *takes amortized and worst-case time $O(n)$. The operation* DELETE(r) *takes amortized $O(1)$ time, and worst-case $O(n)$ time. These times do not include time spent in the data structures storing A and B.*

The space usage of the data structure is $O(|\mathrm{UH}(A) \cup \mathrm{UH}(B)|)$. This space usage does not include the space used in the data structures storing A and B.

The statement about the worst-case times follows immediately from the amortized statements: There is only $O(n)$ potential in the data structure, this can in the worst-case be used at a single deletion. The data structure has no potential before we perform the merge operation.

As already stated, we assume that for a call to DELETE(r) that r is on the upper hull of all the merging data structure it is stored in. This is the case if it is on the hull of the set C, where C is the set containing r that was not part of a call to MERGE (yet). This can be easily achieved without changing the amortized performance of the data structure. If we find that point r is not on the upper hull of the set C, which we are able to detect in our data structure, we only mark it to be deleted and leave it in the data structure. If later in the algorithm r is part of the point set returned as a result from a subsequent delete operation, we perform the delayed operation DELETE(r). We continue to perform such delayed calls to DELETE until all points on the overall hull are not marked deleted. All the calls to DELETE for marked points are already paid for. To not spend too much time searching for points that are marked deleted, we scan all the returned points and maintain a stack of marked points on the overall upper hull. Alternatively we can think of this as recursive calls to DELETE.

3.5.1 Data structure

For every set of points that was created with a CREATE_SET or MERGE operation, we keep a *set record*. As a result of a $C = $ MERGE(A, B) operation, we create a double link between A and C, and also between B and C. Every set record of a set C has two special records holding the so called right-infinity (left-infinity) point of UH(C). We will now describe the data structure holding UH(C) and the truss (as described in Section 3.2) between A and B.

For every input point p we have a record that holds the (x, y) coordinates. This is the *base record* of this point. For every layer of merges where the point p is on the upper hull, we have a record representing it in relation to the other points on that upper hull. This record is called the *local representative* of the point. The local representatives at consecutive merging levels are doubly linked. That is, the local representative of p on hull A is doubly linked to the local representative of p on hull C. All the local representatives of p have a pointer to the base record of p. At the base record of p we additionally store the information whether p is part of the overall upper hull. Remember that p identifies a leaf in the rooted forest of semidynamic merging structures. This leaf is part of one tree that has the root set D. By overall hull we here mean whether or not $p \in$ UH(D).

Additional to the local representatives of input points, we have similar records for *introduced points*, namely for cutoff points of a shortcut, ray-intersections and equality points that are not in the inner of an equality stretch. These records consist of a constant number of pointers to base records of points that are needed to construct the coordinates of the point. This is feasible, as we ensure that none of these points depends on more than constantly many (here 4) input points (Section 3.6.16, p. 65). In order to deal with degenerate cases correctly, we allow to create an introduced point (for example an equality point) e at the position of an input point $p \in A$. The points e and p are in the same place of the left to right ordering that is basis for all of our doubly linked lists. We prepare for this possibility by having one "universal" type of local representative for points. If for example a point p is both an input point and an equality point (similar situation arise for a cutoff and a strong ray intersection point), both the fields corresponding to an equality point (layer 3) and corresponding to an input point (layer 1) of the representation are used. All other parts of the record are left blank, i.e. the ones corresponding to cutoff and ray-intersection points are set to nil. Introduced points are only necessary and meaningful for one instance of merging. They are not pointed to by other data structures, we can delete and change them freely.

At cutoff points we have an additional entry for a double link to a record representing the shortcut. Such a record for a shortcut holds a representation of the line defining the shortcut. This yields an explicit representation of Bd(A') and Bd(B'). It also identifies the points hidden by the shortcut. The points hidden by a shortcut are additionally marked to be hidden. They do not have a pointer to the shortcut that currently hides them (this would be too expensive to maintain).

We maintain three different doubly linked lists of points on Bd(A), the *layer* 1, 2 and 3 as described in the following and illustrated in Figure 3.11. We also have the 3 layers for Bd(B), they are defined analogously. Additionally we have as the 4. layer a doubly linked list of the points of UH(C).

Layer 1 connects all the local representatives of points on UH(A). This is basically a copy of the 4. layer of recursive merging data structure.

Layer 2 connects all the local representatives of points of UH(A'), that is, points of A and some cutoff points.

Layer 3 connects all the local representatives of points of UH(A') and all the strong ray intersection points (intersections of strong rays rooted at B and segments of Bd(A')) and explicit equality points of Bd(A'). An equality point has a *twin* of local representatives, one for the layer 3 of hull A, and another for layer 3 of hull B. A strong ray intersection point has a pointer to the root of the strong ray, the selected point of B. We will collect subsequences of points on layer 3 into (3 types of) splitters. These splitters are disjoint from each other. There are links from the extreme points in the splitter to the splitter. All the points of layer 3 are referred to as DS(A').. The set DS(A') is left-to-right ordered. This order coincides with

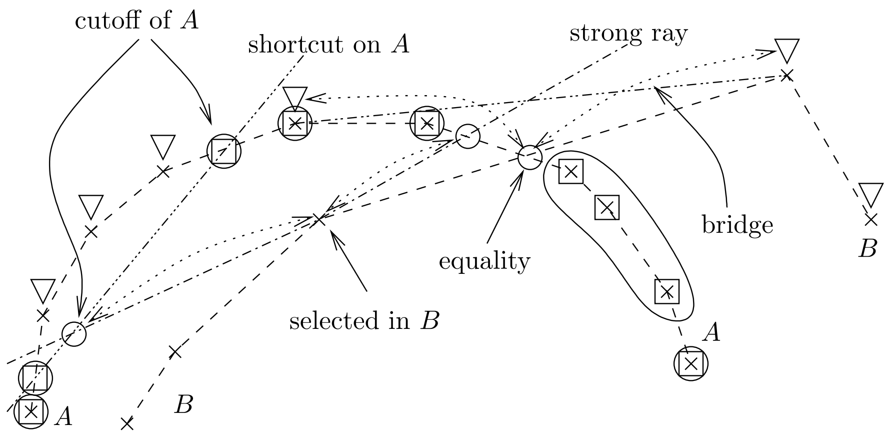

Figure 3.11: Illustration of the 4 layers points of UH(A) participate in. The crosses mark points of layer one in A and in B, the squares are layer two of A, the circles and encircled (in a splitter) points are layer three of A. The triangles indicate layer four. The dotted arrows show pointers aligning A and B

the path order induced by layer 3. We identify equality strips (stretches) by having consecutive equality points on DS(A') and DS(B').

Layer 4 represents the points of UH(C) (identifying bridges). If we have a point $p \in A$ and a point $q \in B$ that have identical coordinates and are on UH(C), we say that $p \in$ UH(C) and $q \notin$ UH(C). If a point is endpoint of a bridge, it additionally has a double link to one of the twin of equality points it is hiding. By Lemma 3.6 (p. 22) this equality point is unique.

The local representative of a point $p \in A$ can be marked as selected. Then on layer 3 there is a double link to the splitter holding the points up to the next selected or equality point to the right, and one to the splitter to the left. Additionally we store the slopes of the strong rays rooted at p (literally or by reference to a defining point), and a double link to the introduced ray-intersection point of Bd(B'). If one of the splitters is empty, the double link goes directly to the next selected or equality point of A.

An equality point p that marks the end of one strip has a double link to the splitter that is part of the half open search of p. If a record is not part of one of the above mentioned doubly linked lists, the according pointers are nil, for example if the point is not selected. Note that we have explicit alignment between DS(A') and DS(B') via selected points and strong ray intersections, and via equality points.

During the run of the algorithm some of the above mentioned connections between A and B might not yet be established. This can happen for the connection between a twin pair of equality points and between a selected point and a strong-ray-intersection point. Such not yet established link is called *broken* link. By Lemma 3.13 (p. 39) this is always only a local problem, that is, we never miss more than constantly many links, and the algorithm can easily keep track of the links that are currently broken.

3.5.2 The data structure in the reestablishing phase

To control the progress of the algorithm that reestablishes the (relaxed) truss after a deletion, we need some more concepts and constructs. Geometrically these are not really new. As in Section 3.3 we describe the situation of the merging of A and B. To be able to talk about the two versions of B, namely before and after the deletion, we say that a point $r \in B_1$ gets deleted, leaving us with $B_2 := B_1 \setminus \{r\}$. We then have to establish a new truss between A and B_2. We keep the shortcuts of the truss, which define the shortcut version of the sets A', B'_1 and B'_2. We continue to talk about B if the distinction between B_1 and B_2 is irrelevant in the situation.

Different types of splitters in the reestablishing

While the algorithm examines the geometric situation it reestablishes more and more constructs of the truss. The novel idea in this activity is to use dangling searches and half open searches. The points of A and B that are part of dangling and half open searches that might no longer be valid (a strong ray intersection or equality point became obsolete) is called the *active stretch* of the reestablishing phase. (One of the first steps in the reestablishing is to determine the active stretch.) To be able to perform these searches the points of the active stretch are stored in splitters. Selecting a point (as a result of a completed dangling search) on B (for example) relies on an efficient way to determine the intersections of the newly created strong rays with $\mathrm{Bd}(A')$. We use again a splitter to search for these intersection points. We store the points of both participating stretches of the upper hulls in three different types of splitters. The type of splitter a point p is stored in reflects the stage of the life-cycle of p, as introduced in Section 3.3.2. The general situation in which splitters are used is schematically depicted in Figure 3.12.

replacement splitter for fresh material from B'_2, initially all of it in one replacement splitter M. Replacement splitters store points p for which $p \in \mathrm{UC}(A)$ is not yet determined. We speculate (according to the already identified equality points) that these points are outside $\mathrm{UC}(A)$, at least they replace a point outside $\mathrm{UC}(A)$ that was deleted. The search operation is never suspended, we do not keep searches dangling on a replacement splitter. Replacement splitters are temporary in the sense that they are destroyed when the truss is relaxed again.

lasting splitters storing points from A' (old lasting splitters), or B'_2 (new lasting splitters). For an old lasting splitter we know that it covers a stretch of A where it coincides with A'. This stems from the fact that the points of the splitter are inside $\mathrm{UC}(B_1)$ (before the deletion of r occurred). During the reestablishing/relaxation phase it is fair to say that we conjecture (according to the already identified equality points) these points to be inside. If this conjecture is true the points stay packaged in the lasting splitter, which allows us to efficiently work with the points. If we store points (in a new splitter) from B'_2 (again coinciding with B_2), we are already sure that the points are currently inside $\mathrm{UC}(A)$. The searches of a lasting splitter can remain dangling.

surfacing splitters for freshly surfaced points of $\mathrm{UH}(A)$, points of A we found out that they are outside $\mathrm{UC}(B_2)$. As an exception we also store points of A' that are known to be outside of $\mathrm{UC}(B)$ in this kind of splitter. The search operation is never suspended, we do not keep searches dangling on a surfacing splitter. These are also temporary in the sense that all surfacing splitters are destroyed in the shortcut phase after the relaxed truss is reestablished.

In the following sections we move the points one by one from one type of splitter to the other, usually doing some kind of linear scan (a geometric sweep line). For the amortized analysis it is important that we have only 3 types of splitters, and that the movement of points between splitters can pay for the linear scan. All types of splitters are part of the doubly linked list that forms layer 3 of the truss.

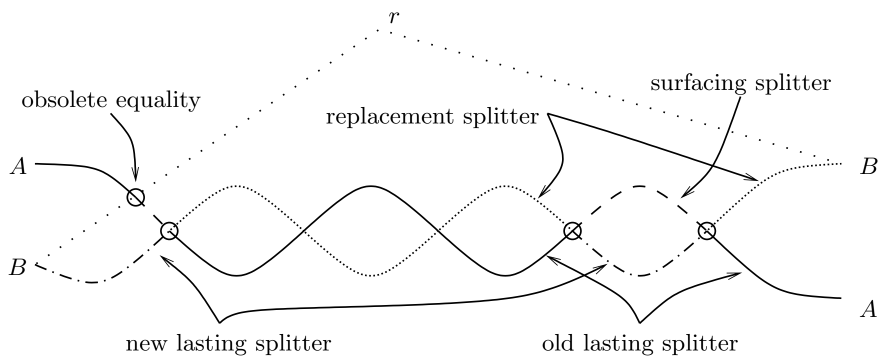

Figure 3.12: Schematic illustration of different types of splitters during the reestablishing phase after deleting r from B. The depicted hulls A and B are by no means convex. There is a geometric situation having this topology, only the size of the drawing gets really big compared to the distance between the hulls. The circles denote equality points the algorithm already detected. There are two more undetected equality points.

Alignment points

One of the important functions of the truss is to align UH(A) with UH(B). The key achievement of the construction is, that this alignment is made explicit only for a few points, just enough that the truss can still be a certificate of the fact that all equality points of Bd(A) and Bd(B) are identified. A point p is called an *alignment point* if it is

1. a selected point,
2. the endpoint of a half open search,
3. an introduced equality point,
4. a strong ray intersection point.

Fences

In the reestablishing phase we make the alignment of the two hulls explicit by so called *fences*. A fence connects two alignment points $a \in A$ and $b \in B$. We refer to a also as the *twin* of b and vice versa. The standard fences are depicted in Figure 3.13.

A fence has always precisely two endpoints, $a \in \mathrm{DS}(A')$ and $b \in \mathrm{DS}(B')$. Geometrically a fence $f = (a, b)$, drawn as a straight line segment, is always inside the symmetric difference of UC(A) and UC(B_2) (but the inner endpoint). This gives f the direction from a to b (by belonging to the sets A and B), which we call outward (upward) if $a \in \mathrm{UC}_0(B_2)$ and inward (downward) if $b \in \mathrm{UC}_0(A)$. If $a = b$, which is the case for an equality point, f does not have a direction.

A fence $f = (a, b)$ can be of 4 different types. If a and b are the same equality points, f is an *equality fence*. If a or b is selected, and the fence lies on a strong ray, f is a *selection fence*. If a and b are on a vertical line and a or b is a selected point we have a *vertical fence*. At the beginning of the reestablishing we see another type of alignment, namely a *tangent fence* as further described in Section 3.6.6 (p. 58), if an existing equality point is affected by the deletion. A situation of this type is depicted in Figure 3.14. This is the case if a is the endpoint of a half open search and b is an equality point that does, as a result of the deletion, no longer exist.

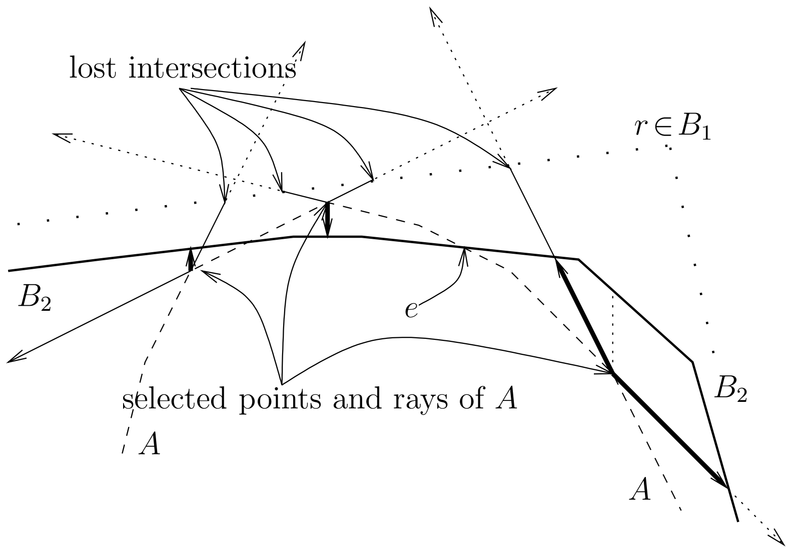

Figure 3.13: After a deletion of the point r from B *vertical fences* are used to place the previously selected points of A against the new part of B. The left selected point has an outward fence and we will scan for the new intersections of the strong rays with $\mathrm{Bd}(B)$. The right selected point had first an outward vertical ray (now dotted) and we performed the scans, leading to two outward *selection fences*. The middle selected point is outside $\mathrm{UC}(B)$, we will deselect it, explore the new strip of opposite polarity, and build a new truss. When exploring to the right we will find the equality point e and make it an *equality fence*

The algorithmic significance of the direction is that in the neighborhood of an inward fence we have fresh material on the inside hull and freshly surfacing material on the outside hull. So we can afford to perform a *linear exploration*, i.e., a linear scan for the new relevant equality points. If we instead have an outward fence, we cannot afford to perform a linear scan on the surrounding of the fence on $\mathrm{UH}(A)$, a problem we overcome by working with dangling searches on splitters and half open searches.

3.5.3 Data structure requirements

During the course of our algorithm, we keep several geometric invariants. Most importantly these are invariants of a dynamic version of the static truss introduced in Section 3.2. Additionally there are some requirements about the data structure representing the geometry correctly.

Requirement 9 (All equality points)
All equality points of $\mathrm{Bd}(A)$ *and* $\mathrm{Bd}(B)$ *are identified and represented as a pair of introduced equality points in the data structure.*

Requirement 10 (All strong ray intersections)
All intersections between a strong ray and $\mathrm{Bd}(A')$ *or* $\mathrm{Bd}(B')$ *are represented as an introduced point.*

Requirement 11 (Representation of dangling and half open searches)
Between two selected points there is a lasting splitter with an active dangling search. This dangling search is geometrically relaxed. The splitter might be the empty

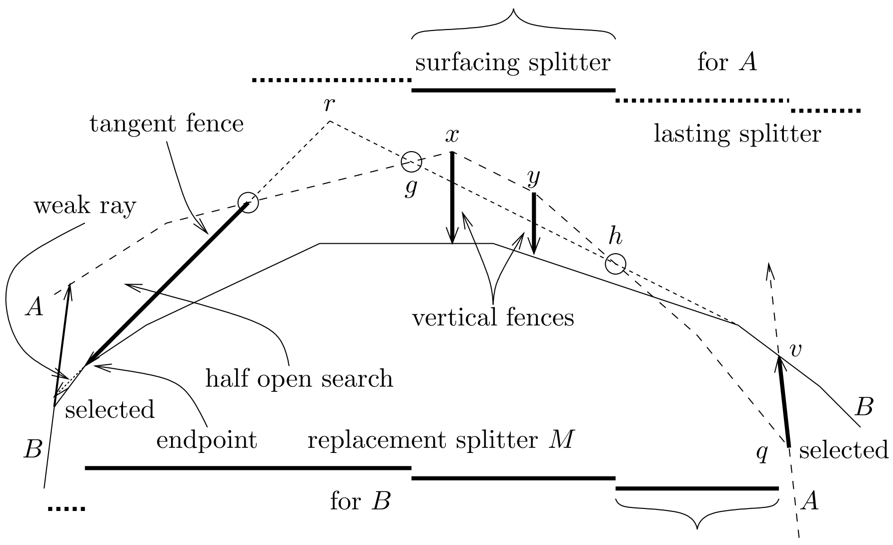

Figure 3.14: The situation of destroyed equality points (marked with circles). The partitioning of the two upper hulls into splitters (and their type) is given by the two staircase lines. The fences are depicted as thick arrows. Note the alignment of splitters via fences. The curly braces denote sections where the aggressive shortcutting is crucial.

splitter which has a trivial (and trivially relaxed) dangling search.

The points of UH(A) *between a selected point p and the neighboring equality point u (a half open search) are stored in a lasting splitter that has no active dangling search.*

Requirements during the Fencing phase

As it will be important for the analysis, we note beforehand that we will only create a constant number of fences when initializing the reestablishing after a deletion (paid by the deletion) and a constant number of fences when selecting a node (paid by the node, it only gets selected once). This is due to Lemma 3.13 (p. 39).

That the algorithm we present terminates is implied by the analysis that shows that it takes amortized constant time per participating element. The key idea for this is that every step of the algorithm moves at least one point from one of the three classes of splitters to the next, advancing the life cycle of the point into the next stage.

The fencing stage will establish the following requirements (Requirement 12 to Requirement 16). This happens in the order they appear here. As soon as one of the requirements is established the algorithm keeps it as an invariant.

Requirement 12 (Stability of A' and B')
After determining B'_2, the double linked lists of layer 1 and 2 do not change.

Requirement 13 (Alternation)
In the active stretch, on both hulls, layer 3 alternates between alignment points (endpoints of fences) and splitters, and lists of points that are to be hidden by a shortcut.

Requirement 14 (Alignment)
In the active stretch, every alignment point p is aligned by a fence to another alignment point q, its twin, on the other hull.

During the reestablish phase we will eventually achieve that stretches of the truss have relaxed dangling searches and we know that there cannot be any equality points in these stretches. We still need to create new shortcuts, such that the final truss complies with the aggressive shortcutting policy (Requirement 6, p. 31). We will keep a list of stretches of points that might be in conflict with Requirement 6. We refer to such a stretch as to be shortcut.

Requirement 15 (Shortcut list)
The replacement or surfacing splitter above a relaxed dangling or half open search, and the material between two strong rays r and t that are rooted at the same selected point, are included in the list of to be shortcut stretches.

When we finish a dangling search on a lasting splitter and select a candidate or guard point, we need this splitter aligned with precisely one replacement or surfacing splitter on the other hull. Otherwise we could not efficiently select the point that is the result of the search. On layer 3 of the data structure, splitters are always neighbors of alignment points. We say that a splitter G is *aligned to splitter H* if the left neighboring alignment point of G is aligned to the left neighboring alignment point of H, and the two right neighboring alignment points are also aligned.

Requirement 16
Every active splitter is aligned by two fences to one other active splitter on the other hull. One of the two splitters is a lasting splitter.

This also gives rise to a natural linear ordering of the fences from left to right. This ordering coincides with the geometrical one. Fences do not cross each other as geometric objects.

3.6 A deletion

The interface of the overall semi-dynamic data structure assumes for the DELETE(r) operation the the point r is on the overall convex hull. This assumption is justified in Section 3.5.

As a naming convention we assume that the instance of merging we are considering merges the sets A and B into $C = A \cup B$, and $r \in B$. To capture the changes to set B in our notation, we define B_1 and B_2 by $r \in B = B_1$, and $B_2 = B_1 \setminus \{r\}$. To be able to report correctly the changes to C, we remember the two neighbors of r on UH(C) until we finished processing the deletion of r on this level of the merging. By the definition of the interface of one merging data structure, we get from the participating merging structure a list L of fresh points on B_2. As by Lemma 3.12 (p. 38) this describes the change on the upper hull caused by the deletion. $L = \mathrm{UH}(B_2) \setminus \mathrm{UH}(B_1)$ is a list of points ordered from left to right, forming a connected subsequence of UH(B_2). Now we have to create the same type of list for $\mathrm{UH}(A \cup B_2) \setminus \mathrm{UH}(A \cup B_1)$.

If in the following discussion we use the apparently imprecise notion of B (or B' and alike), we implicitly state that in the particular usage there is no difference between B_1 and B_2, the particular part of B we are referring to is not affected by the deletion.

Intuitively the algorithmic task is to repair the truss. We can think of the deletion of r as being a destructive move that our data structure has to be able to deal with. We process a deletion just like a repair task. We start by determining the extent of the impact, that is, we determine all the points of our construction that become *obsolete* because they are defined by one of the segments that are incident to r. Here we crucially use that *the geometry of the truss limits the size of the impact to be constant.* Before we delete the obsolete points, we have to identify the *active point*s of the truss that (geometrically or in the data structure) depend on obsolete points. For the active points we create some auxiliary constructs that we later use to place the active points into the new truss. This is prominently the case for some of the selected points of A. A selected point $p \in A$ is *preselected* if for one of the strong rays f rooted at p the intersection of f with $\mathrm{Bd}(B_1')$ is given by an obsolete segment and therefore possibly different from the intersection of f with $\mathrm{Bd}(B_2')$. By Lemma 3.13 (p. 39) there are at most 5 preselected points. Then we start to process the fresh points that replace r on $\mathrm{UH}(B_2)$. As a first step we apply the shortcuts we have for B_1. This modifies the list of fresh points. Now we place the fresh points into the data structure, and reestablish the pointer structure of the truss. Finally we start to analyze the geometric position of the fresh points. We advance dangling searches and perform linear scans over new (parts of) strips (establishing points in a new stage of their life-cycle, so we can afford to scan). This eventually leaves us with a relaxed truss. Finally we create new shortcuts and adjust old and create new bridges.

The algorithm we present here takes the priority on the simplicity of the exposition not on the fine scaled efficiency. Keep this in mind, some of the operations seem to (unnecessarily) throw away information. This is purpose, it unifies to the situation where we have the smallest amount of knowledge.

Both segments that are incident to r on $\mathrm{Bd}(B_1)$ might be intersected once or twice by $\mathrm{Bd}(A)$. This yields 5 different cases of numbers of intersections (identifying symmetric cases): (0,0), (1,0), (1,1), (1,2), and (2,2). Our first goal is to unify this situation without making too many cases explicit.

3.6.1 Deleting obsolete points and links

Triggered by the deletion of point r, the segments s_l and s_r of $\mathrm{Bd}(B)$ incident to r become obsolete. This makes cutoff points located on s_l and s_r obsolete, which in turn makes some shortcuts active. We additionally have to delete all equality points and ray intersection points that are located on obsolete segments of $\mathrm{Bd}(B_1')$.

To reflect the fact that the mentioned pieces of the construction are no longer valid in the data structure, we do the following. We walk along layer 2 of B, starting at the local representative of r until we find the next point of B itself, that is a point of layer 1. For the cutoff points we find on this way, we create entries in the list of *active shortcut*s. Then we walk along layer 3 of B, away from r or from one of the (determined) obsolete cutoff points, until we find the next (unaffected) cutoff point of B or a point of B itself. This determines all the (introduced) points of B_1 that are obsolete. We delete all the obsolete points of B_1, break the double link to their twins and include the twin brothers into the list of active alignment points of A' that lost their twin.

By the geometry of the situation (Lemma 3.13, p. 39) the list of obsolete and active points has constant length, the ordering is mere convenience. The necessary walk-along on the data structure of B is of constant length.

As a first step we restore the doubly linked list of local representatives of $\mathrm{UH}(B_2)$. We create local representatives for the points of L and connect them with each other as layer 1 of the data structure for $\mathrm{UH}(B_2)$. We connect this representation of L with the neighbors of r on $\mathrm{UH}(B_1)$. This establishes the representation of layer 1 of $\mathrm{UH}(B_2)$.

3.6.2 Applying shortcuts

This step is going to establish layer 2 of the representation of $\mathrm{UH}(B_1')$ in the data structure. After this step we can forget about the obsolete cutoff points.

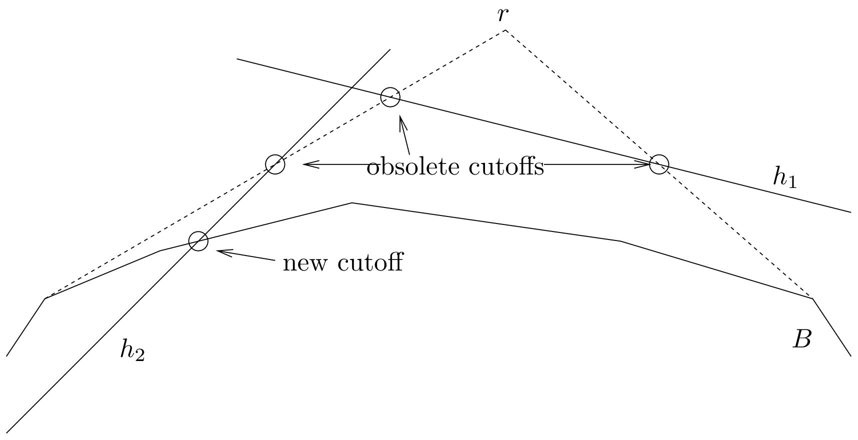

Figure 3.15: Applying shortcuts to B after the deletion of r. The active shortcut defined by line h_1 becomes futile and obsolete. The active shortcut defined by h_2 shortens and changes one of its cutoff points.

Let H be the set of lower half-planes defining the shortcuts that are defined to transform B_1 into B_1'. Recalling the definition

$$\mathrm{SC}_H(B) = \mathrm{UH}\Big(\mathrm{UC}(B) \cap \bigcap_{h\in H} h\Big) ,$$

we have $B_1' = \mathrm{SC}_H(B_1)$, and want to get our hands on $B_2' = \mathrm{SC}_H(B_2)$. By the interface of the data structure, the change from B_1 to B_2 is that the point r is replaced by a (possibly empty) list of points. The change from B_1' to B_2' is in general more complicated. There we replace a list R' of at most 4 points (r and some cutoff points, Lemma 3.12 (p. 38), Statement 3) with a list L' of points. More precisely we define $R' = B_1' \setminus B_2'$ and $L' = B_2' \setminus B_1'$. Be aware that we in general do not have $L \subseteq L'$ or $L' \subseteq L$, since shortcuts remove points of L and introduce cutoff points.

To compute L' in our data structure, we only have to determine the new positions of the cutoff points on active shortcuts. It can happen that a shortcut of H becomes empty and gets deleted. We do this with a left to right sweep line over the points of L.

The shortcuts of H are vertically separated (Requirement 5, p. 31), and because of the deletion of r shortcuts (as segments) only shrink (Property 3.7, p. 30). Hence the slab defined by an active shortcut $h \in H$ on B_1 bounds where we have to expect the new cutoff points of h on B_2, and the slabs of the active shortcuts are disjoint. A left to right scan over the points of L allows us therefore to determine L'.

More precisely we have two modes of the scan: expecting the beginning of the new shortcut (a left cutoff point) or expecting the end of the new shortcut (a right cutoff point). If the left neighbor of r on B_1 is cut away by shortcut $h \in H$, we start searching for the corresponding right cutoff point. Similarly we do not have to search for the right cutoff point of a shortcut $j \in H$ that cuts away the left neighbor of r on B_1.

When we walk along L looking for a left cutoff point, we merely copy the points to L'. If we find such a cutoff point, we include it into L' and switch mode. In the other mode we do not copy points, but mark them to be hidden by some shortcut. If we find the next right cutoff point, we include it into L' and switch back to the other mode. If we reach the end of the slab where the new shortcut h might be without finding a cutoff point, we conclude that h is no longer effective. In this case we delete h.

After this procedure we do not only have computed L', but we also have the data structure in a state that represents B'_2 and the points that are hidden by a shortcut. This establishes that layer 2 of B represents correctly B'_2. We will not change this layer before the next deletion.

3.6.3 The initial replacement splitter

As a start for the reestablishing phase of layer 3 we build a replacement splitter M containing the elements of L'. Unfortunately an endpoint of L' (as identified in Section 3.6.2) is not necessarily an alignment point of B'_2. We extend the splitter M to the left and right such that it includes all points of B'_2 up to the next alignment point. This is only necessary if the (left or right) endpoint p of L' is locally outside, i.e., $p \notin \mathrm{UC}(A)$. Then we extend M up to the next equality or strong ray intersection point of B'. This extension is only by constantly many points, as the truss meets the aggressive shortcutting Requirement 6 (p. 31). Now we have a splitter M with (fresh) points of B'_2 that is enclosed by the active alignment points u and v.

This gets the data structure of B already in a state that conforms to Requirement 13 (p. 52). The only thing that we really miss is the alignment with A, namely the preselected points and lost equality point (twins), Requirement 14 (p. 52).

3.6.4 Initial Fences

The next step to achieve a valid data structure of the truss as defined in Section 3.5.1 (to establish the truss opposed to relaxing it), is to align the alignment points. To do so, we introduced fences. The situation is symmetrical for the alignment points u and v that enclose the replacement splitter M. Remember that by construction we have $u, v \in \mathrm{UH}(B_1) \cap \mathrm{UH}(B_2)$. If we discuss and illustrate the situation for u the symmetrical construction applies to v and vice versa. If u is an equality point, u has a twin u_A on $\mathrm{DS}(A')$, and we create an equality fence (u, u_A). If v is an intersection of a strong ray r rooted at a selected point $q \in A$, we create the (outward) selection fence (q, v). This situation is illustrated as part of Figure 3.14 (p. 52) and also Figure 3.16.

If u is the endpoint of a half open search, we instantiate a tangent fence as illustrated in Figure 3.16. Let i be the obsolete equality point, j the neighbor of i on A', such that $j \notin \mathrm{UC}(B)$. Assume $p \in B$ is the selected point, and e is the strong ray intersection with $\mathrm{Bd}(A')$ of this half open search. Then we create the tangent fence (j, u). We delete i and remove it from layer 3 of B. We change the link of the lasting splitter N_A of A from i to j. Now we can remove the obsolete equality point i from layer 3 of A. We create a surfacing splitter N_S that we initialize with the $O(1)$ many points of A' between e and j. We link N_S on layer 3 between e and j. We remember u as a *potential guard* (for the case that the strip that contains p has to be joined with its right neighbor strip). We create a selection fence between p and e. Note that the obsolete equality point i cannot be part of a trivial half open search, as u is a point inside $\mathrm{UC}(A)$.

Note that there is no such thing as an outward directed tangent fence. If one of the deleted segments contains two (now obsolete) equality points, the corresponding initial fence is only a selection fence. We treat the obsolete equality points similar to preselected points of $\mathrm{UH}(A)$, as explained in Section 3.6.8.

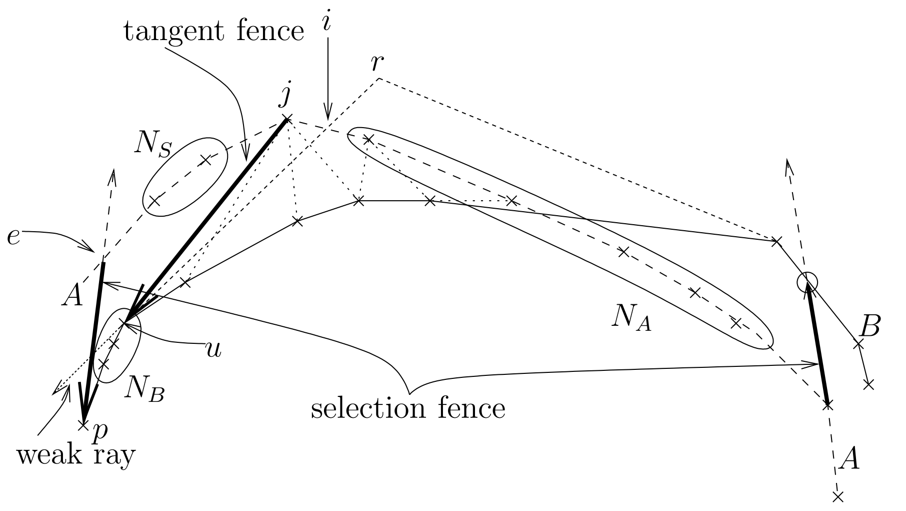

Figure 3.16: The situation of creating the tangent fence (j, u). We also see two selection fences and all the initial splitters.

The fence on the left that has as one endpoint u is called $f_l = (u, ?)$, similarly we have on the right $f_r = (v, ?)$. With this we align the endpoints of the replacement splitter M (that holds the fresh points). Unfortunately it might be, that on Layer 3 of A we have more than one splitter between the two fences f_l and f_r. Be aware that f_l is not necessarily the leftmost fence we instantiated, and f_r not necessarily the rightmost, namely if they are tangent fences.

This operations achieve the alternation Requirement 13 (p. 52) and the alignment Requirement 14 (p. 52) for the active stretch of B'_2.

3.6.5 Finishing dangling searches to unblock splitters

We will need the following procedure in Section 3.6.6 and Section 3.6.9. The geometry of this situation is described and illustrated in Section 3.3.2. There we considered the situation where an equality point literally moved over a selected point p. The similarity to the situation now is, that our linear scan moves over the selected point p, telling our algorithm to deselect the point. The similarity is, that we know the side of p that is outside the other hull (and this is where both the equality point and the sweep line come from).

Assume we have a lasting splitter N that has as left neighbor the selected or preselected point $p \in \mathrm{UH}(A)$ and as right alignment the (pre-)selected point $q \in \mathrm{UH}(A)$. A point d of $\mathrm{UH}(A)$ surfaces, if we have $d \in \mathrm{UC}(B_1)$ and $d \notin \mathrm{UC}(B_2)$. (Here B and B' are locally equivalent.) If p or q surfaces we have to finish the dangling search on N, which we do as described in the following.

Finishing a dangling search means, that we have to keep the promise of splitting between the guards of the dangling search. Let g_l, g_r, c by the points standing respectively for the current left guard, right guard and candidate of N.

Assume now that p has surfaced. (The case for q surfacing is symmetrical.) We preselect g_l, splitting N into N_l and N_r. This finishes the dangling search, keeping the promise to split between the guards. The strong ray intersection property to the left is assured as we have $p \notin \mathrm{UC}(B_2)$ and there are no preselected or selected points between p and g_l on $\mathrm{UH}(A)$. The strong ray intersection property to the right follows from g_l being the right guard of a dangling search that is delimited by q, which currently is the next (pre)selected point of $\mathrm{UH}(A)$ to the right of g_l.

We will describe in further detail how to select a point in Section 3.6.11. If we already have established the initial set of fences, the splitter alignment requirement is met (Requirement 16, p. 53). Then there is a splitter M on B that is aligned to N via p and q. Then we split M at the vertical line defined by the selected point and introduce a vertical splitter. This maintains the splitter alignment requirement. Otherwise we are in a phase of the reestablishing that allows to have preselected points. In this case we consider the to be selected point as preselected.

3.6.6 Extending

We have to perform the following procedure first, as we have to detect a lost pair of equality points correctly. Additionally this is the only way to process tangent fences. Instead of making this detection phase explicit as we do here, we could incorporate it into the normal exploration phase as described in Section 3.6.9, and handle it there as a special case.

Let f_l be the fence as defined in Section 3.6.4, and assume it is a tangent fence, i.e., one of the points of f_l is the endpoint of a half open search (left of f_l). If f_r is a tangent fence we perform the symmetric procedure. If none of them is a tangent fence, we continue as described in Section 3.6.8. We perform the extending procedure before we instantiate any other fences between f_l and f_r. If we find that we have to join two strips, the sweep we perform here cleans up (removes from layer 3) all alignment points of A. If there are more than two obsolete equality points, we know that there is no joining of two strips. We still perform the extending procedure to process tangent fences. Otherwise we instantiate vertical fences after the sweep segment found a new equality point.

We start by investigating the geometry, ignoring Layer 3 of both hulls for a moment. We will describe how to establish a good structure of Layer 3 later, depending on the geometric situation we find. We instantiate a sweep segment $\overline{b,a} = f_l$ with $b \in B_2$, and $a \in \mathrm{Bd}(A')$ that we use for finding the first equality point. We move this sweep segment as described in Section 3.2.3.

A first difficulty we face here is, that the sweep segment does not necessarily meet the slab invariant: Let a' be the right neighbor of a and b' the right neighbor of b, then the slab invariant is that either a lies in the slab formed by b and b', or b lies in the slab formed by a and a'. Because the segment that was just deleted had the equality point i on the segment $\overline{a,a'}$, we know that b is to the left of a'. So the only problem can be that b' is to the left of a. But we also know that all points of $\mathrm{UH}(B'_2)$ to the right of b and to the left of a are below the segment $\overline{b,i}$, which in turn is completely inside $\mathrm{UC}(A')$. Hence there can be no equality point to the left of a, we can blindly advance the sweep segment until the slab invariant is met. This situation is illustrated in Figure 3.17.

Now we advance the sweep segment until we either find an equality point x of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$, or we find an endpoint of the other tangent fence f_r. (If f_r is not a tangent fence, we cannot lose a pair of intersections. We know in advance, that we will find an intersection.)

If we find the tangent fence f_r, we know that we lose a pair of intersections. In this case we proceed as described in Section 3.6.7. Otherwise we have to adjust Layer 3 for both A and B to reflect the geometric situation we just discovered.

For hull A this means that we walk from the fence f_l to the point x, guided by Layer 2. As we go, we extend the surfacing splitter N_S (from Section 3.6.4, Figure 3.16 and also Figure 3.17). Let m be such a point we want to move. Then m might currently be the leftmost point of a splitter that has no active dangling search. In this case we shrink the splitter to free m. Or it can be a selected point. Then we finish the dangling search on the splitter to the right of m as described in Section 3.6.5. This preselects one of the guards of the splitter. When we arrive

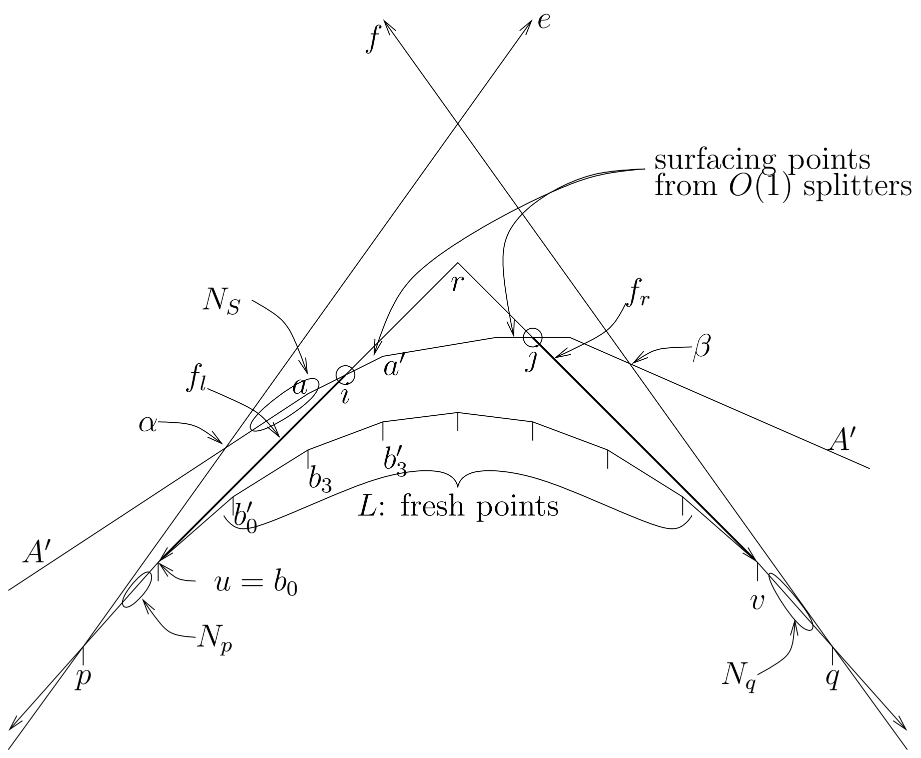

Figure 3.17: The geometric situation when detecting that two strips have to be joined. We also see the data structure aspects of joining the strip.

at x, we create x_A on Layer 3 of A and make it the left neighbor of the (shrunk) lasting splitter, and the right neighbor of the surfacing splitter. This establish the alternation Requirement 13 (p. 52) on A. If we find an obsolete equality point that is not part of a tangent fence, we destroy it and continue. The aggressive shortcutting policy (Requirement 6, p. 31) assures that we will only include constantly many points of A'. This limits the cost that is induced by reseting (turning back) the life-cycle of these points.

For hull B we walk from f_l to x, moving points from the replacement splitter into the lasting splitter of the half open search that defined f_l. We create x_B and make it the right neighbor of the lasting splitter and the left neighbor of the replacement splitter. We establish the twin link between x_A and x_B.

This establishes a fresh half open search with the same selected point and strong ray as the half open search defining f_l.

We perform the symmetric sweep starting from f_r, if this is also a tangent fence. As we did not lose a pair of equality points, we are sure to find an equality point y. It can happen that $x = y$, i.e., we find the same equality point from both sides, a degenerate case we do not have to treat separately.

This establishes alternation Requirement 13 (p. 52) on both A and B, and the alignment Requirement 14 (p. 52) for the active stretch of B. For the swept over points we also have the alignment of splitters (Requirement 16, p. 53).

After having taken care of extending strips as described in this section, there are no tangent fences left in the construction. Only a next deletion can create new tangent fences.

3.6.7 Losing a pair of equality points

Assume that we detected the loss of a pair of equality points by the procedure in Section 3.6.6. More precisely we have two points u, v of $\mathrm{UH}(B_2) \cap \mathrm{UH}(B_1)$, such that all of the points between u and v are a list L of fresh points, as depicted in Figure 3.17. We detected that u, v and all points of L are inside $\mathrm{UC}(A)$. Even though L is already stored in the replacement splitter M, we consider it as a list of points only. We know that we have a situation where $\mathrm{Bd}(B_1)$ has one pair of equality points more with $\mathrm{Bd}(A)$ than $\mathrm{Bd}(B_2)$, i.e., we lose a pair of equality points because of the deletion of r. We know that the fences $f_l = (i, v)$ and $f_r = (j, v)$ as defined in Section 3.6.4 are both tangent fences.

In the following we use names as illustrated in Figure 3.17 (p. 59). For the half open search that is delimited by f_l, we call p be the selected point of $\mathrm{UH}(B)$, e the right directed strong ray, α the intersection of e with $\mathrm{Bd}(A')$, N_p the lasting splitter and u the endpoint of this half open search. Analogously, for the half open search delimited by f_r, we call q the selected point of $\mathrm{UH}(B)$, f the right directed strong ray, β the intersection of f with $\mathrm{Bd}(A')$, N_q the lasting splitter and j the endpoint of this half open search. As already pointed out in Section 3.6.4 are the points u and v remembered as potential guards.

In Section 3.3.3 we already argued that we achieve a geometrically valid (not yet relaxed) dangling search , when joining the splitter N_p and N_q over the list L of fresh points. We perform the operation $N = \textsc{Join}(N_p, L, N_q)$. On layer 3 of B we make p the left neighbor of the lasting splitter N, and q its right neighbor. We collect all the points of A' between α and β into a surfacing splitter S. On layer 3 of A' we link S to the alignment points α and β (they are both part of a selection fence). All the points between i and j are surfacing, and there are only constantly many points between α and i (and j and β). Hence this operation is affordable. We delete the fences f_l and f_r and instantiate two selection fences (p, α) and (q, β). That is, we have one pair of a lasting and a surfacing splitter, properly aligned by fences.

This establishes alternation Requirement 13 (p. 52) on both A and B, and the alignment Requirement 14 (p. 52) for the active stretch of B. We continue and advance this dangling search as described in Section 3.6.13.

3.6.8 Preselected points and obsolete equality points

In general we will now look at a situation where we have the replacement splitter M on B between f_l and f_r. In contrast, on the active stretch of $\mathrm{Bd}(A')$ there can still be not aligned alignment points, i.e., alignment points that are not part of a fence yet. By the construction of layer 3 of $\mathrm{UH}(A')$ it is easy to scan over those points.

Let p be such a point. We know that p is either an equality or a preselected point. We furthermore know that it is located between two fences f and g that are connected on $\mathrm{Bd}(B')$ with a replacement splitter M. (In the first step f_l and f_r.) We take the vertical line through p and search for its intersection with M (using the fact that M is a splitter), where we split M into M_l and M_r, introducing a point $b \in \mathrm{Bd}(B')$ that we link into Layer 3 of B as the neighbor of M_l and M_r. The new splitters M_l and M_r inherit the left and right alignment (fence) from M, and we instantiate the vertical fence (p, b).

There might still be one or two pairs of obsolete equality points on $\mathrm{Bd}(A)$ that delimited a trivial strip S, i.e., a strip of polarity A over B where $\mathrm{Bd}(B)$ consists of a single segment and there are no points of B within S. Between two such points there is not yet a splitter. Let g and h be such a pair of equality points. This situation is part of the illustration in Figure 3.14 (p. 52). Let x be the neighbor of g on A' (Layer 3) that is outside $\mathrm{UC}(B)$; if $q \in A'$ we have $x = g$. Similarly we define y as the outside neighbor of h on A'. We delete the equality points g and h, x and y inherit the lasting splitter they delimited. Then we take the (constantly many) points on A' between x and y (Layer 2) and build a surfacing splitter N out of it. We link N on Layer 3 with x and y. This achieves the the alternation Requirement 13 (p. 52) for the active stretch of A. Now we split the replacement splitter M at the two vertical lines defined by x and y, and introduce two auxiliary points b_x and b_y on layer 3 of B. We create two vertical fences (x, b_x) and (y, b_y). Those get processed as described in Section 3.6.9.

This achieves the alignment Requirement 14 (p. 52) for the active stretch of A and B, the alternation Requirement 13 (p. 52) for A and B, and the splitter alignment Requirement 16 (p. 53) for the active stretch of both A and B. In other words the active stretch of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ is represented in splitters that are properly aligned via fences.

3.6.9 Exploring

The process of exploring a new (part of) a strip should be regarded as part of the relaxation phase. There it happens after a point (of A) is selected, and we detected that it surfaced. It is very similar to the extending of a strip as described in Section 3.6.6. As part of the reestablishing we also initiate an exploring process because of a preselected point that surfaces, or to extend a (previously trivial) strip where A is above B, where both defining equality points were on the two deleted segments of $\mathrm{Bd}(B)$. The boundary of an exploration is always indicated by a vertical inward fence.

As a special case we can also explore equality regions. In such a case we move from point to point until we find that the two hulls are no longer identical. Then we either find a new strip of polarity A over B, so we continue exploring. Otherwise we find a strip of polarity B over A which means that we stop exploring and process that strip by initiating and advancing dangling and half open searches.

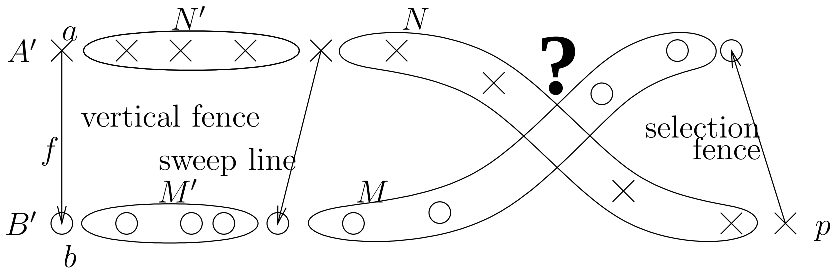

Figure 3.18: Exploring a new strip

Let $f = (a, b)$ be a vertical inward (A above B) fence. This situation is schematically illustrated in Figure 3.18. Then on one side of f, say to the right (there is the symmetrical case), we have that point a delimits a lasting splitter N holding points of A that is aligned with the replacement splitter M holding points of B, delimited by b. If the other side of f already has a surfacing splitter N' for points of A and a lasting splitter M' for points of B, we will extend them. If we do this we destroy the fence, including the auxiliary points. Otherwise we create M' and N' with empty content, and delimit it by a and b respectively. The main goal of the process of exploring is to find the corresponding equality alignment point for M' and N'.

As described in Section 3.2.3, we move a sweep segment. As we move over points we shrink and extend splitters as described in the previous paragraph. If the sweep segment moves over selected points, we unblock the splitters by finishing a dangling search. This we do as described in Section 3.6.5.

If we find another unprocessed (vertical inward) fence as we move along, we destroy it. If it is partly processed we join the freshly created lasting and surfacing splitters and destroy the fence. We can avoid this situation by always continuing to explore at partly processed vertical fences.

When we find the equality point e, we create the twin representation for Layer 3 of both A and B, and connect it respectively to the splitters M' and N'. We instantiate an equality fence at e. This maintains the alternation Requirement 13 (p. 52) and the alignment Requirement 14 (p. 52) for the active stretch of both A and B. We also maintain the splitter alignment (Requirement 16, p. 53).

3.6.10 Establishing

If we cannot explore any new strips, we have to establish the truss by instantiating and advancing dangling searches.

Assume we are in a situation where all alignment points and splitters are aligned. Assume further that there are two neighboring equality fences f and g, i.e., there are no further fences between f and g. If we locally at f or g have the situation (in one direction) that $\mathrm{Bd}(A)$ is above $\mathrm{Bd}(B)$ (or they are equal), and the upper splitter is lasting and the lower replacement, then we explore this (new) strip like in Section 3.6.9.

If this is not the case, we know that locally at f and g the lower splitter N is lasting and is paired up via f and g with a replacement or surfacing splitter M. If N is empty, we know that both equality points (given by f and g) are on one segment (of $\mathrm{Bd}(A)$ or $\mathrm{Bd}(B)$), which means that we look at a trivial strip with a degenerate but relaxed half open search. Otherwise we select any of the points in the lasting splitter as described in Section 3.6.11. This eventually establishes Requirement 11 (p. 51) that there is at least one selected point in every strip that contains a point of the locally lower hull.

3.6.11 Selecting a point

Assume that the current truss consists of aligned splitters (Requirement 16, p. 53). Let p be a point of a lasting splitter N such that the canonical pair of rays rooted at p meets the strong ray separation Requirement 1 (p. 26) with all other strong rays. This can be the case as a result of advancing a dangling or half open search or of establishing a region between two equality points. By the fences N is paired up with a replacement or surfacing splitter M. We know that the vertical line defined by p as well as the canonical rays rooted at p will intersect the other boundary ($\mathrm{Bd}(A)$ or $\mathrm{Bd}(B)$) between the alignments of M. The situation is illustrated in Figure 3.19.

We split N at p, which results in two splitters N_l and N_r, none of which contains p. N_l inherits in Layer 3 of A the left neighbor from N, N_r the right neighbor. The point p is the new right neighbor of N_l and the new left neighbor of N_l. We search and split M at the intersection point y of $\mathrm{Bd}(M)$ with the vertical line through p into the splitters M_l and M_r (not removing a point). This results in a vertical fence, just as if p was preselected. This maintains the alternation Requirement 13 (p. 52) and the alignment Requirement 14 (p. 52) for the active stretch of both A and B

We proceed either with "Establishing strong rays" (Section 3.6.12) or "Exploring" (Section 3.6.9), depending whether or not y is above x.

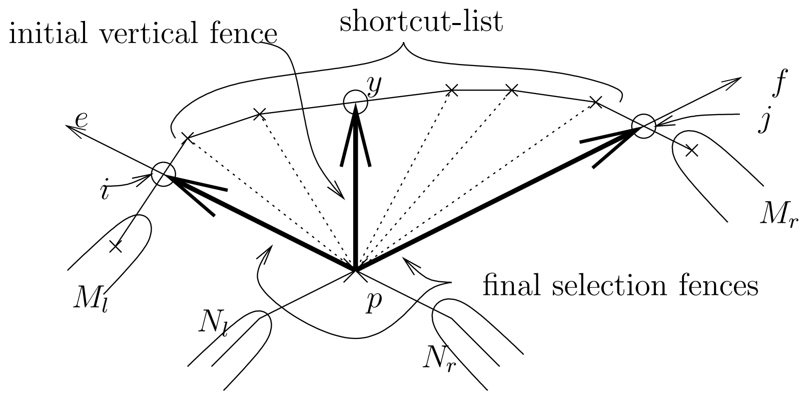

Figure 3.19: Selecting the point p that meets the strong ray requirement with all other strong rays. In this case p does not surface, and we establish the strong rays e and f together with the strong ray intersections i and j.

3.6.12 Establishing strong rays

Assume we have a vertical fence f with the two endpoints $p \in A$ and $y \in B$, where the point p is delimiting the lasting splitters N_l and N_r, and is below the point y delimiting the replacement splitters M_l and M_r.

We assume that the splitters M_l and M_r have no active dangling search. This stems from the fact that only lasting splitters have dangling searches, on surfacing and replacement splitters the searches are not suspended. There is a symmetric case with $p \in B$ and $y \in A$ where M_l and M_r are surfacing splitters.

We instantiate the canonical pair of rays e, f at p as the strong rays of the selected point p. Remember that we do not change the strong rays of p later. Now we perform a linear scan in M_l for the intersection i of e with $\mathrm{Bd}(B')$, which is promised by the strong ray separation (Requirement 1, p. 26) to be within M_l. We shrink M_l and insert the points into a list, that we remember to be shortcut, as required by Requirement 15 (p. 53). The steps of the linear scan for the strong ray intersection are straightforward, we can decide whether to move on by examining the rightmost element of M_l. When we find the intersecting segment (or point), we make it an introduced point of $\mathrm{Bd}(Y)$. This establishes under M_l either a dangling search (we instantiate the dangling search of the splitter) or a half open search, depending on the fence at the left alignments of M_l and N_l. We perform the symmetric scan over for M_r to find the intersection j of f with $\mathrm{Bd}(B')$.

This maintains the alternation Requirement 13 (p. 52) and the alignment Requirement 14 (p. 52) for the active stretch of both A and B.

3.6.13 Advancing a dangling search

We have a pair of selection fences f_l and f_r, with the lower, lasting splitter N on A, and the upper replacement splitter M on B. There is also a symmetric case where we have a new lasting splitter N on B and a surfacing splitter M on A.

If this is not already the case, we instantiate a dangling search on N. From the candidate point (segment) c of N we consider the pair of canonical rays e, f. This defines the intersection a of e with r and the intersection b of f with s. As described in Section 3.2.5 we can decide upon the search by examining the relative position of the pair of canonical rays rooted at c, f_l, f_r and the strong ray intersections.

If we can advance the search to the right or the left, we do this and continue advancing the dangling search. If c is a point that meets the strong ray separation requirement (Requirement 1, p. 26), we select c as described in Section 3.6.11. This fulfills the promise to split M between the guards. If c is a geometrically valid candidate, the dangling search is relaxed, and the part of the truss between f_l and f_r is reestablished. We make M part of the list of splitters to be shortcut.

This maintains the splitter alignment (Requirement 16, p. 53).

3.6.14 Advancing a half open search

We describe one case here. There are also symmetric cases (both with A and B interchanged and with left and right interchanged), where the same names and concepts are applicable.

Let s be a half open search, delimited by the selection fence $f = \overline{q, d}$ and the equality fence v. Let $q \in \mathrm{UH}(A)$ be the selected point of f, and i the endpoint of the half open search s. Let N be the splitter between (delimited by) q and v. By examining only the geometry of q, v and i we can decide whether or not s is relaxed as described in Section 3.2.6. If it is not relaxed, we select i as described in Section 3.6.11.

This procedure maintains the splitter alignment (Requirement 16, p. 53).

3.6.15 Creating new shortcuts

As a result of the of the relaxation phase, the situation of the data structure is that we have several relaxed dangling and half open searches. On the side we established a list of replacement and surfacing splitters (or just lists of points), that have to be shortened to comply with the aggressive shortcutting policy (Requirement 6, p. 31). Let L be such a list of points that have to be shortcut. We know (have a certificate for by the alignment points) that all points of L are surfaced. We destroy the surfacing or replacement splitter that holds the points of L, reducing it to a list.

If L consists of less than 4 segments in the beginning, then there is in need to introduce a new shortcut.

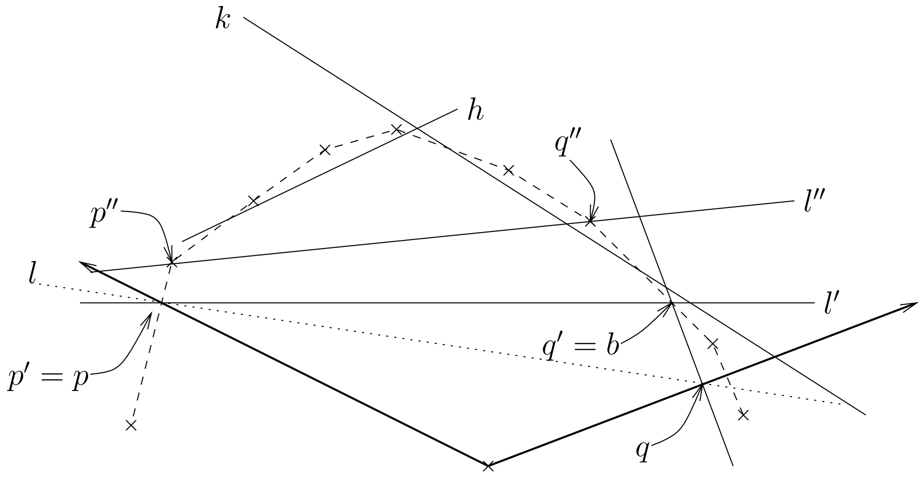

Figure 3.20: The situation of creating a new shortcut. The most aggressive choice l is forbidden by shortcut separation. The line l' is suitable and maintains the monotonicity of $\mathrm{UC}(A')$ but has the disadvantage of depending on a previous shortcut. The line l'' provides a shortcut that results in monotonic strong ray intersections and is defined directly by two input points.

To meet the aggressive shortcutting Requirement 6 (p. 31), we have to create new shortcuts (on A or B) that cuts away almost all the points of L. The situation is illustrated in Figure 3.20. Naively we would take the line l defined by the alignment points p and q of L. This might introduce a shortcut that is is not separated (Requirement 5, p. 31), if p or q are themselves on shortcuts. To avoid this we define $p' = p$ if p is not on a shortcut, and $p' = b$ if p is on a shortcut with the cutoff points a and b, where b is the cutoff above l. Similarly we define q'. Now we define the real shortcut line l' by p' and q' and make p' and q' a twin pair of cutoff points (on Layer 2), defining this new shortcut. We walk along the elements of L, marking them as being hidden by a shortcut (and remove pairs of cutoff points, effectively removing superseded shortcuts).

3.6.16 Avoiding high precision arithmetic

Now we might still have the problem that the points p' and q' that define the line l' can themselves be defined by a shortcut. In Figure 3.20 this is the case for q'. In this situation the parameters defining l' can depend on all the input points the two former shortcut lines dependent upon, plus four more points. This dependency on many input points can in principle accumulate, we have no a priori bound on the number of input points the parameters of l' depends upon. As explained in Section 2.1.2 we prefer shortcuts that are defined only by input points and not by auxiliary, computed points. We observe that all shortcuts (as segments) that are completely above l are irrelevant when it comes to the monotonicity of intersections on strong rays (Requirement 8, p. 40). Ignoring this kind of shortcut we walk one more step beyond q' and take q'' as the next input point, that is the first input point in L above l. We define p'' in analogy. We define a new shortcut s by the line l'' through p'' and q''. The shortcut s fulfills the aggressive shortcutting Requirement 6 (p. 31), between p and q are at most 4 points, that is at most 3 consecutive segments not intersected by a strong ray.

3.6.17 Bridges

After having reestablished the truss, it remains to establish bridges. Most of the geometric and algorithmic aspects of this are discussed in Section 3.2.1 and Section 2.4.3, we only have to adjust the algorithm to respect our representation of the data structure and the accounting.

A *protector* is a double link between the local representatives of two points of A (or B). Its importance is that it allows us to not use the points "behind" the protector twice for finding a bridge. The situation is depicted in Figure 3.21.

More precisely we address the situation that the deleted point r is the endpoint of a bridge (r, a). W.l.o.g. we assume that r is to the left of a. Then we have an equality point e of the merging that is hidden by (r, a). Now we instantiate a protector on A between a and the right neighbor v of e on $\mathrm{UH}(A)$, $e \notin \mathrm{UC}(B_1)$ (unless $v = a$). Let s be the right neighbor of r on $\mathrm{UH}(B_1)$, i.e., s is below the bridge (r, a). Let u the left neighbor of the equality point e on $\mathrm{UH}(B_2)$, i.e., $u \notin \mathrm{UC}(A)$. If u is to the right of s, we instantiate a protector between u and s. We include e into a list of fresh equality points, and remember the protector. We can think of the bridge (r, a) as deleted. If there is another bridge (to the left of r) having r as an endpoint, we perform the same steps.

Now we have to consider every equality point e out of the list of fresh equality points, that is, e was either marked fresh when instantiating protectors, or was found in the reestablishing phase reacting to the deletion of r.

We begin searching for a bridge (u, v). We set a candidate bridge (a, b) to the neighbors of e. For the candidate bridge we can by the case analysis of Section 3.2.2

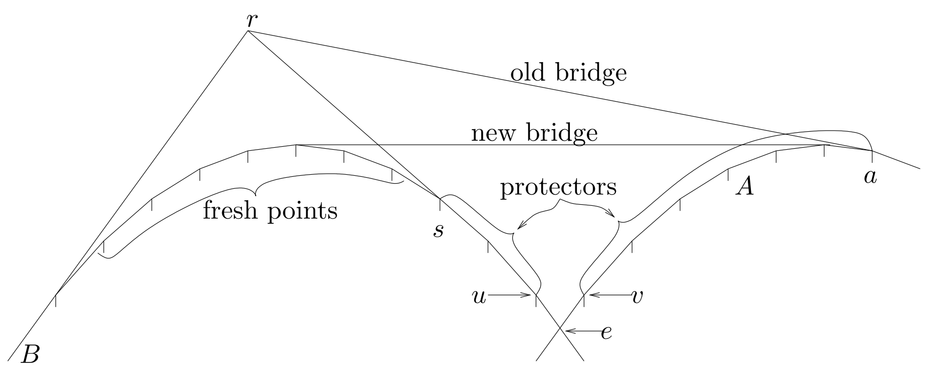

Figure 3.21: The situation of placing protectors. When processing the deletion of point r we have to protect parts of the upper hulls of A and B from being searched over again, when we search for the new bridge.

decide what we should do next. We either find out that the bridge is at the correct place, or we get the information that we have to move (at least) one of a or b in a particular direction. Now we only have to try the points of A and B in a particular order (of course respecting the decisions the search already took), and we are sure to find the bridge (u, v). The order is to go along UH(A) away from e. Only if we meet the endpoint of a protector we "jump" to the other end of it. There we either continue (linearly) away from e or linearly back towards e. When we find the bridge, we establish the links on layer 4 of our data structure representing it, including the link to e.

After having considered all fresh equality points we delete all unused protectors. By the correctness of the case analysis for the bridge finding, a point is considered at most once while searching for the bridge (u, v). Additionally we only search over points "under" the protector towards e. This means, that points we move away from must now be on UH(C), they will never be searched over by a bridge finding of this merging level again. Only the endpoints of the protectors might be searched over already (and again later). As there are at most 4 protectors for every deletion, the process of bridge finding is achieved in amortized $O(1)$ time per element participating in the merging.

3.6.18 Interface to the next merging level

Starting from the (remembered) old neighbors of r on UH($A \cup B_1$), we follow the new upper hull UH($A \cup B_2$) on layer 1, using bridges if they are present. We update Layer 4 while we go along. This works also for the left and right infinity points.

3.6.19 Analysis of the fencing phase

Now we have to argue, that all operations of the locating stage require only amortized constant time per merge. The key ingredient to this analysis is that a single point can basically participate in every step of the above process at most once, and the processes like exploring, extending, establishing either happen only once or twice per deletion or every point can initiate such a process at most once.

Here is the complete life cycle of a point p of A in the merging of A and B into $C = A \cup B$.

1. (because of a deletion of $r \in A$) the point $p \in A$ becomes part of UH(A). This is the first time p is processed on this merging instance. We insert p into a

replacement splitter.

2. During the reestablishing phase after the deletion of r we realize that we have $p \in \mathrm{UC}_0(B)$ and include p into a lasting splitter.
3. In the relaxation phase (of some later deletion on A or B), we decide to select p. We still have $p \in \mathrm{UC}_0(B)$.
4. We find that we have $p \in \mathrm{Bd}(B)$, the point p is explored as part of an equality stretch.
5. Because of a deletion on B we realize that we now have $p \notin \mathrm{UC}(B)$. We insert p into a surfacing splitter.
6. The point p gets cut away by a shortcut.
7. During the bridge finding we realize that p is hidden by some bridge.
8. As a result of a deletion (on A or B) we realize in the bridge finding stage that p is no longer hidden by a bridge, i.e., we have $p \in \mathrm{UH}(C)$.
9. The point p gets deleted.

Note that hiding by a shortcut (Statement 6) and a bridge (Statement 7) are independent of each other; both can happen at most once. This linear life cycle is only disturbed by the fact that a deletion can move constantly many points backwards in their life cycle, namely include them into a replacement or surfacing splitter again. This already accounts for all actions that move points from one splitter into another. As the call to ADVANCE DANGLING SEARCH of the splitter frees a constant amount of potential, the comparisons needed to perform this call are also paid for.

Another thing we have to account for are the operations that determine intersections between strong rays and segments. These operations additionally create introduced points. The observation is that we determine these points only once for a selected or preselected point. So the point getting selected (and the deletion) can pay for this. This includes the constant cost induced by determining the intersection of a vertical fence, and for setting up a dangling search that is delimited by the new strong ray intersection.

We also have to account for the equality points we create. Realizing that we only start to search for equality points for tangent fences and after selecting a point (and finding it on the surface), we can charge the creation cost of the equality point to the (pre-)selection of a point.

The linear cost for creating a shortcut and the cutoff points is paid for from the life cycle of the cut-away points. The constant setup cost to create a shortcut is charged to creating the strong ray intersection or equality point.

The cost for applying shortcuts is paid for by the points that are found to be cut away by the shortcut. The setup cost is paid for by the deletion, there are only constantly many shortcuts affected by one deletion. This constant setup cost include paying for the life cycle of the new cutoff points.

The other important observation is about fences, namely that we only create situations that we also know how to deal with. This process is not as linear as the life cycle of a point. Roughly we start by introducing tangent fences, vertical fences, selection fences and equality fences. First we eliminate the tangent fences, then we explore and establish until we are left with unrelaxed dangling and half open searches. When relaxing the searches we again select points, explore and establish. As all the work we do is accounted for as described above, we are sure to eventually find a situation where we have only relaxed dangling and half open searches.

Now we can charge all the cost induced by a point, including the deletion cost, to the participation in the MERGE operation. This cost is $O(1)$.

3.7 Linear space: Separators

If we use the data structure of Section 3.5 as it is described so far, we might get a poor space efficiency. More precisely if the points are all on the upper hull, then every point is stored on every of the $\log n$ merging levels. This totals in a space requirement of $\Theta(n \log n)$. In this section we describe an extension of the data structure that reduces the total space usage to $O(n)$. The general idea is to avoid storing a point in more than two semidynamic merging structures. The data structure used to achieve this is called a *separator*. We place a separator above every instance of a semidynamic merging data structure.

The interface of the semidynamic data structure requires us to make the upper hull of the stored set explicit in a doubly linked list. Hence we have to store the points on the upper hull ourselves. On the other hand, the recursive data structure gives us access to the upper hull of the set of points it stores. What we can do is to use the ability of the recursive data structure to delete points. We can delete all the points on the upper hull from the recursive data structure. In other words the separator makes the first two convex layers of the set of points it stores explicit: The points on the convex hull of the stored set are deleted from the recursive data structure form the 1. layer, the upper hull of the remaining points (stored recursively) form the 2. layer. We will use the notation that the separator stores the semidynamic set C and partitions it into A and B such that $B = \mathrm{UH}(C)$. We call this policy *eager deletions* as we delete a point as soon as we can. This achieves that all points that might be stored in data structures using the separator are not stored in the recursive data structure the separator owns. The space usage of a separator is proportional to the number of points it stores explicitly, that is the number of points of the first and second convex layer of the set it stores.

The problem we have to address is how we can support deletions of points. This is the algorithmic task we consider in this section.

Recall (Theorem 3.17, p. 46) that the semidynamic merging structure uses space proportional to the number of points on the recursive upper hulls. If we place a separator above every internal node of the merging forest, every point is stored explicitly in one merging structure and in one separator. This implies an overall $O(n)$ space usage. Instead of thinking of separators as unary nodes in the merging forest, we can think of the separator as augmenting the binary nodes, namely the merging data structures.

A separator uses recursively one semidynamic merging data structure and is itself a semidynamic merging structure, both interfaces fitting to the interface of the semidynamic merging structure (Definition 3, p. 19) in the following sense

INIT(A) Initializes a new data structure B that wraps around the existing semidynamic merging data structure A. Let C be the set stored in A. Then B has the interface of a semidynamic merging structure storing set C. We delete points from A such that no point is stored simultaneously in A and in B.

DELETE(p) Delete the point $p \in C$ from the set of points C stored in B. As a result the explicit representation of the upper hull of B is correctly updated. Points from A are deleted to guarantee that points are not stored twice.

3.7.1 Geometry with tangent oracle

Let us consider the geometry of the situation. Assume we want to provide a semidynamic data structure for the set C. The task of providing a separator requires us to partition C into two sets A and B. Let A be the set of points stored recursively, and B the points stored at the separator. We maintain the upper hull $B = \mathrm{UH}(C)$ and maintain $A = C \setminus B$. From the recursive data structure we have access to UH(A). The algorithmic challenge of the situation is to correctly react to a deletion of a point $r \in B$. More precisely we have the situation $C_1 = A_1 \cup B_1$ before the deletion. Now we delete $r \in B_1$. This defines $C_2 = C_1 \setminus \{r\}$. The algorithm has to determine $B_2 = \mathrm{UH}(C_2)$ and $A_2 = C_2 \setminus B_2$.

Assume (unrealistically) that we have access to a tangent oracle on UH(A): For any point $x \in \mathbb{R}^2$ the oracle tells us in constant time the tangents on UH(A) through x or that $x \in \mathrm{UC}(A)$. Then the task becomes easy: Let x and y be the left and right neighbor of r on B_1. The situation is exemplified in Figure 3.22. We use the oracle to determine the (right and left) tangents on UH(A_1) passing through x and y. This allows us to distinguish two cases: If the segment $\overline{x, y}$ is outside $\mathrm{UC}_0(A_1)$, we have $\mathrm{UH}(C_2) = \mathrm{UH}(C_1) \setminus \{r\}$. Setting $B_2 = B_1 \setminus \{r\}$ and $A_2 = A_1$ is sufficient to restore the separator. Otherwise the tangents through x and y identify a list L of points on UH(A_1) that replace r on the upper hull UH(C_2), i.e. $L = \mathrm{UH}(C_2) \setminus \mathrm{UH}(C_1)$ and we have $L \subseteq \mathrm{UH}(A_1)$. Setting $B_2 = (B_1 \setminus \{r\}) \cup L$ and $A_2 = A_1 \setminus L$ restores the separator. Note that we in both cases have $A_2 \subseteq A_1$, we can achieve the change from A_1 to A_2 by deleting points, which is conforming to the interface of the recursive data structure storing A.

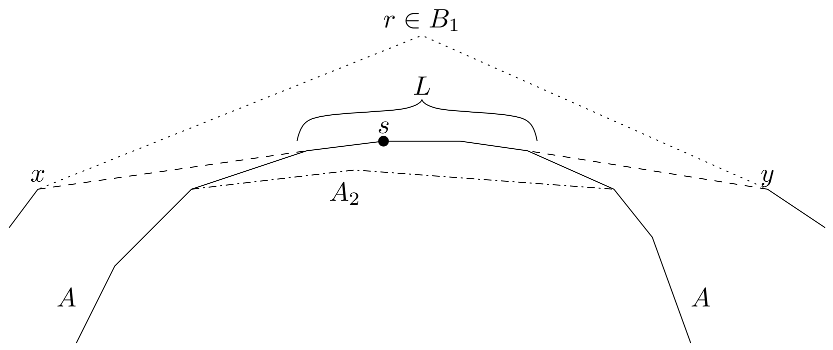

Figure 3.22: Geometric separator with a tangent oracle; The point $r \in B_1$ gets deleted. Its neighbors x and y determine the set $L \subseteq \mathrm{UH}(A_1)$ that surfaces, and becomes part of B_2. We delete L recursively, yielding UH(A_2)

This discussion showed that the task we are aiming at geometrically complies with the definition of our interfaces. We only lack an algorithm identifying the points of A_1 that become part of the upper hull of C_2.

Note that there is an apparently weaker oracle that allows us the above operations: It is sufficient to know one point s of A_1 that surfaces if such a point exists. We can locally confirm whether or not s really surfaces, and we can scan along UH(A_1) to identify all the surfacing points. A point can only surface once in the separator. This means that there is no asymptotic slowdown for the merging of point sets, assuming we have the surfacing point oracle.

3.7.2 Using the truss

A tangent oracle would also help a lot in the setting of a semidynamic merging structure. There we developed the truss that we can use instead. The situation of a separator is similar, even somewhat simpler. The truss is defined between two upper hulls that are allowed to change by deletions of points (if the point is on the overall upper hull). We first have to argue that we can phrase the situation of a separator in a way that conforms to this specification. We consider an algorithm that performs the macroscopic changes to the truss, namely the changes to the sets of points. For the simplicity of the exposition we first describe an algorithm that has access to a tangent oracle or a surfacing point oracle. Only later we will describe how the same changes to the truss can be achieved even without access to an oracle.

We maintain a truss between the sets A and B, just as if we were merging A and B. This is reasonable, the truss as a certificate allowed us to conclude for a strip that indeed B is above A (or vice versa). We have the same situation here, the truss is the certificates that assures that we identified the upper hull (as the set B) and no point of A is part of the upper hull of C.

To comply with the interface to the recursive data structures for A and B we have to pretend to have a recursive data structure for B. We do this in the following way. After the deletion of $r \in B_1$ we take the list $L \subseteq \mathrm{UH}(A_1)$ as the replacement of r on B_2, i.e., $B_2 = (B_1 \setminus \{r\}) \cup L$. We reestablish the truss between B_2 and A_1. This computes the new upper hull of $A_1 \cup B_2$. As we assumed that we know L already, this computation is somewhat futile. Then we delete the points of L one after another from A. After every step we reestablish the truss, using the information we get from the recursive merging structure storing A. This eventually results in a truss between A_2 and B_2.

We will maintain a truss between A and B that is adopted to the special geometric situation. Instead of using an oracle, the truss determines surfacing points itself.

3.7.3 Augmented dangling search

By the definition of A and B the truss consists of one strip of polarity B over A (before processing the deletion of r, after that we might have precisely on equality stretch in the middle). This means in particular that we cannot loose equality points, and that the reestablishing phase only advances dangling searches to possibly detect an equality point. As soon as we found one equality point we are in the same situation as if we had an oracle telling one surfacing point. After that we continue reestablishing the truss in the standard way.

Recall that the reestablishing phase after a deletion as described in Section 3.6 starts by placing initial fences and investigating (unifying) the situation around the deletion. In our special geometric situation, the initial fences will be selection fences, and we start to investigate the position of the surfacing points of B (the points we do not know because we do not have an oracle) as part of a dangling search as described in Section 3.6.13. The geometric situation of a dangling search is explained in Section 3.2.5.

The difference to a standard dangling search is, that we can not assume to know the new strong ray intersections. We start by considering one specific geometric situation. After having seen how we can work in this specific situation we will consider how to achieve it.

Assume that we have a dangling search on A with the left selected point p, the right directed strong ray e rooted at p. The right selected point is q with the left directed strong ray f. Assume that the deletion of $r \in B_1$ affected the strong ray intersections on e and f. Let x and y be the two neighbors on $\mathrm{UH}(B_1)$ of r. Then we are in the situation that the segments $\overline{x,r}$ and $\overline{r,y}$ intersect e and f. This situation is illustrated in Figure 3.23 (p. 71). Now we are in the situation where we have to advance the dangling search between p and q, but we do not have access to the strong ray intersections of e and f with $\mathrm{Bd}(B_2)$.

If we assume that no point is surfacing, i.e., $B_2 = B_1 \setminus \{r\}$ and $L = \emptyset$, then the only segment of B_2 is $\overline{x,y}$. This segment defines the intersection points with e called x' and with f called y'. These points are also depicted in Figure 3.23 (p. 71) (the figure illustrates a different case where some points surface). We always have that $x', y' \in \mathrm{UC}(B_2) \setminus \mathrm{UC}_0(A_1)$ because x' and y' are located on a strong ray. In the standard situation and if we knew that no point is surfacing, the points x' and y' are the strong ray intersections, and we can advance the dangling search. As we simultaneously have to address the case that there are surfacing points, we have to advance the dangling search in a slightly more complicated way. We will still use x' and y' in the role of strong ray intersections, which is incorrect if there are points surfacing. It is even incorrect in the dangerous direction, the real strong ray intersections will be further away from p and q, so we are in the danger of violating the monotonicity requirement on strong rays (Requirement 8, p. 40). The analysis of the situation shows that this is never the case.

We need the strong ray intersections mainly to define the right and left guard criterion. Recall that the guard criterion allows us to advance a dangling search: if we find out that the current guard c of the dangling search satisfies the left guard criterion, we can advance the search to the right and make c the new left guard. Before we formulate the modified guard criterion, we investigate the geometric situation if there are surfacing points. We can detect surfacing points locally, as formalized by the following lemma.

Lemma 3.18
Let $t \in \mathrm{UH}(A)$ such that t is above $\overline{x,y}$ and let e and f be the pair of canonical rays rooted at t. If e is above x and f is above y then t surfaces, i.e. $t \in \mathrm{UH}(B_2)$.

Proof: As t is above $\overline{x,y}$, some points of $\mathrm{UH}(A)$ surface. If t is not surfacing there needs to be a point $s \in \mathrm{UH}(A)$ such that t is below $\overline{x,s}$ (w.l.o.g.). But then the left directed canonical ray of t would be below x, because all the segments of $\mathrm{Bd}(A)$ between s and t are already below $\overline{x,s}$. □

We identify the following version of the monotonicity lemma.

Property 3.19
Let x be a point in the plane, $x \notin \mathrm{UC}(A)$. Let M subset of $\mathrm{UH}(A)$ defined by the canonical right directed ray from the point being above x. Then M consists of all points of $\mathrm{UH}(A)$ to the left of some vertical line.

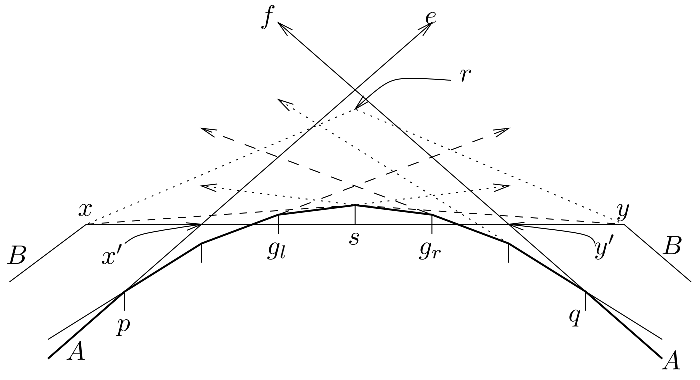

Figure 3.23: Situation of an augmented dangling search; The point $r \in B_1$ gets deleted. Its neighbors x and y determine x' and y'. These points guide the dangling search. Eventually a surfacing point like s will be found.

We consider a point c to be an *augmented left guard* if we have that the right directed canonical ray at c is above y and above y'. Symmetrically we have an *augmented right guard* if we have that the left directed canonical ray at c is above x and above x'. This definition allows us to advance a dangling search.

We have to argue that an augmented left guard g_l is a point that meets the strong ray separation requirement to the right, towards the selected point q. If no point is surfacing this is given by the definition of x'. Otherwise let h be the right directed canonical ray rooted at c, and i the intersection of the left strong ray f and h. The situation is illustrated in Figure 3.24. Then we know that $i \notin \mathrm{UC}(A)$. We also know that $i \notin \mathrm{UC}(B_2)$ because i is above $\overline{x,y}$. Assume that $i \in \mathrm{UC}(C_2)$. Then i must be below the line defined by one point $a \in A$ and a point $b \in B$. By the observations how a deletion of a point in C can change the upper hull, we know that $b = x$ or $b = y$. We also know that $a \notin \mathrm{UC}(B)$. Because c is between p and q we know that points above $\overline{x,y}$ and in $\mathrm{UC}(A)$ have to be inside the triangle formed by $\overline{x,y}$, h and f. This contradicts the assumption that i is below $\overline{a,b}$.

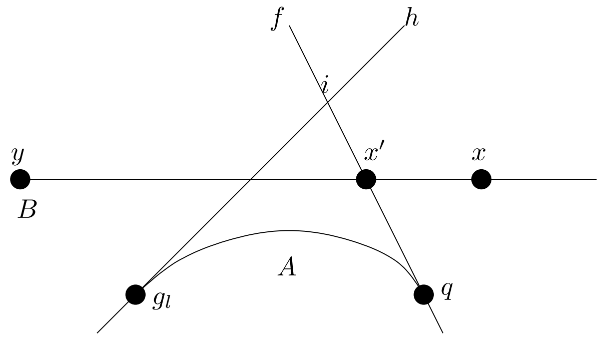

Figure 3.24: Situation of an augmented left guard; We have to argue that in any case the intersection point i is outside $\mathrm{UC}(B_2)$.

With this argument we can conclude that we may select a point if it is simultaneously augmented right and an ordinary left guard (or vice versa). If we find a point that is simultaneously augmented right and left guard, this point surfaces and we can continue as if we had a surfacing oracle. As we do not yet know all of B_2, we can not determine strong ray intersections. We advance the dangling searches as augmented dangling searches. We might look at another case of an augmented dangling search that we will consider later.

If the current candidate c is neither a left nor a right augmented guard, we have two possible situations. If c is below $\overline{x,y}$, it means that the dangling search is relaxed according to the standard definition of a dangling search. In this case it provides us with a certificate that there is no point surfacing, and we can continue to reestablish the truss as if the oracle told us that there are no points surfacing.

If c is above $\overline{x,y}$ and not an augmented guard, we can conclude by Lemma 3.18 that c surfaces. Then we can continue reestablishing the truss as if c was the answer of the oracle, namely we scan along $\mathrm{UH}(A)$ starting from c and identify all the surfacing points of $\mathrm{UH}(A)$. In this case none of the guards we decided upon is part of the truss as a guard anymore, the possible non-monotonicity on the strong rays cannot lead to selecting a point or to a guard that would not meet the ordinary guard criterion.

Any augmented dangling search has to have at least one of the strong rays intersected by $\overline{x,y}$. We already discussed the case when both strong rays are intersected. This leaves us only with the case where one strong ray is intersected and the other one is not. W.l.o.g. we assume that e is not intersected by $\overline{x,y}$. In this case we know the strong ray intersection of e as it is not affected by the deletion of r. Requiring for a right guard to have the left directed ray above x is then clearly a stronger requirement than the ordinary right guard criterion. Advancing the dangling search as an augmented dangling search with $x' = x$ will therefore only select points that meets the strong ray separation requirement. If the search relaxes without finding a surfacing point, we can conclude that there is none. Hence this asymmetric augmented dangling search is also correctly replacing the surfacing oracle.

By the analysis of the standard truss we get that the amortized time spend in a separator is $O(1)$ per point. We use one separator above every semidynamic merging structure. This new construction has precisely the same amortized time performance. The space usage is reduced to $O(n)$.

Chapter 4

Fully dynamic data structure

In this chapter we develop a fully dynamic planar convex hull data structure. The semidynamic data structure presented in Chapter 3 maintains a doubly linked list that stores the points currently on the upper hull in left-to-right order. In contrast to this explicit representation the fully dynamic data structure only allows for queries to the convex hull. There are all kinds of interesting queries: the extreme point in a certain direction, the tangent lines on the convex hull through a given point, whether a point is inside the convex hull, given a point on the convex hull the right (left) neighbor on the convex hull, the number of points on the convex hull between two points on the hull, just to name a few. When we consider the dual of the problem, the lower envelope (as defined in Section 4.1.3), we also transform the queries. Then we ask questions like which line segment of the lower envelope intersects a vertical line (dual to the extreme point query), which segment of the envelope intersects an arbitrary line (tangent query) and the extreme point on the lower envelope in a certain direction (dual to a segment intersection/bridge finding query). We will take this point of view on the problem throughout this chapter.

When we develop the data structure we focus on the extreme point query. In Section 4.5 we will see that we can use the data structure as it is for other queries as well. In (most of) this chapter we will change our point of view to the dual. This seems to be the most natural way to describe the concepts we are using.

We summarize the goal of this chapter in the following definition:

Definition 4 (Fully dynamic Merging Structure)
is a data structure that maintains a multiset S of points in $\mathbb{R}^2$ (points are identified by pointers, not by coordinates) under the operations

INSERT(p) *The point p in the plane is given by its (x, y) coordinates. Inserts p into the multiset S. Returns a pointer to the representation of the point p, its* base record.

DELETE(p) *The point p is given by a pointer to its base record. Removes p from the multiset S.*

QUERY(a) *(extreme point) Returns (a pointer to) a point of $p \in S$ such that the line l given by $y = ax + b$ for some b contains p and all points of S are on or below l.*

Two points are considered to be different, even though they have the same coordinates. The identity of points is solely given by the pointer to its base record.

4.1 Overview over the data structure

The semidynamic planar convex hull data structure we developed in Chapter 3 is particularly suited as a part of a fully dynamic data structure. It supports insertions in the sense that it can easily be used in a standard dynamization technique. Such a technique can turn a deletion only data structure into a fully dynamic data structure. The technique we use is attributed to Bentley and Saxe [BS80] and is suited for decomposable search problems. We describe this technique (and the generalization we use) in Section 4.1.1. The key feature of the technique is to maintain a partition of the stored set. Every subset is stored in a semidynamic (deletion-only) data structure. Insertions are handled by merging of subsets (and reestablishing semidynamic data structures for them).

Even though the extreme point queries on a convex hull is a decomposable search problem, the dynamization technique alone does not achieve fast queries. We will use a variant of an interval-tree (Section 4.1.4) to combine the queries on the different subsets. We have to maintain the interval-tree when subsets are merged and points are deleted. To achieve this, we use as a black box several secondary structures, namely fully dynamic planar convex hull data structures. It is important that the secondary structures store significantly less points at a time and therefore do not need to be as efficient. This meta-recursion is called *bootstrapping*: We first define a suitable inefficient data structure and use it as a secondary structure to increase its efficiency. We discuss the details of the bootstrapping in Section 4.4.

To be in the position to pay for the insertions into the secondary structures and the navigation in the interval-tree, we have to use the mentioned techniques, but not with a constant degree (tree nodes, merging), but we have to carefully choose parameters. We discuss the role and choice of these parameters in Section 4.1.11.

We continue this chapter with several sections on isolated solutions to problems that we have to solve. The calculations about the overall data structure in these sections are to be understood as motivating the technique. We show how the different constructions and parameters fit together into the construction of the real data structure in Section 4.2. There we will also present the analysis of the amortized performance of the data structure.

This chapter of the thesis is based upon the already published work of the author and Gerth Brodal [BJ00]. There are several ideas stemming from different sources. We use the dynamization technique of Bentley and Saxe [BS80]. We consider the situation in the dual setting. We use an interval-tree to achieve fast queries. These basic ideas stem from Chan's [Cha99a] data structure. We store segments closer to the root of the interval-tree than they have to be. This idea goes back to [BJ00]. Here we additionally perform lazy deletions (actually in the form of lazy movements). This requires a more sophisticated analysis, namely the annotation of the data structure with location justifiers and barriers.

4.1.1 Dynamization technique

A *decomposable search problem* arises in the situation where we search for an element in a set, the global answer. The problem is decomposable if for any partition of the set the answers for the subsets contains the global answer and we can easily identify the global answer among the answers for the subsets. Finding the element of the set maximizing an objective function (that is query dependent) is a typical example of a decomposable problem: The global maximum must also be the maximum within the set, and the maximum of the maxima of the subsets is the global maximum. This kind of consideration is useful if the set is partitioned into a few big subsets, that support fast searches for local maxima. Finding the global maximum among the local maxima is easy as there are not too many of them.

More concretely the dynamization technique stores the set S in $e = O(\log |S|)$ different semidynamic data structures for the sets $S_1, \ldots, S_e$. To answer a query on S we perform the queries on the sets S_i, and combine the answers. To be capable of inserting points, we perform some kind of binary counting on the sets. More precisely every set has a rank. We insert a point by creating a singleton set of rank 0. If we have two sets of the same *rank* i, we merge them into one set of rank $i+1$. As an invariant we get that a set of rank i has size 2^i if we assume that there are no deletions of points. This immediately implies that every point can participate in at most $O(\log n)$ merge operations, and at any point in time we have at most $O(\log n)$ different sets. Using the semidynamic data structure of Chapter 3, we already achieve that the insertion and deletion costs are amortized $O(\log n)$. The only problem is the performance of the query operation. We query all the sets, and every query takes $O(\log n)$, which already is $O(\log^2 n)$, a $\log n$ factor slower than what we aim at.

We can easily generalize this technique by introducing a parameter r, the *merging degree*. Now instead of merging as soon as we have two sets of the same rank, we wait until we have r sets of the same rank, and merge only then. Instead of the binary counting, this process resembles counting to the base r. Every set of rank i then has r^i elements and each element participates in at most $\log_r n = \frac{\log n}{\log r}$ merge operations. The total number of sets that we can have simultaneously is bounded by $e = \frac{r \cdot \log n}{\log r}$.

As our semidynamic data structure is only capable of binary merging, we will choose r as a power of two. Then we can simulate merging r sets by binary merges according to a complete binary tree.

4.1.2 Doubling Technique

It is easier to construct a data structure, if we know in advance an upper bound on the number N of elements that will be inserted into the data structure. Assume that the performance of the (operations of the) data structure depends on this parameter N. In general a data structure will not have access to this kind of information. We also have to deal with the fact that the number of insertions does not necessarily correspond to the number of simultaneously stored elements.

As it turns out we can easily cope with all of the above problems. We can guess the value N reasonably good, and we can afford to rebuild the whole data structure if the number of insertions and stored elements should differ significantly. This process is referred to as a *doubling technique* and was systematically investigated by Overmars [Ove83]. The name stems from the rule to rebuild the data structure whenever we find that our guess is off by a roughly a factor of two.

Definition 5
A function $f : \mathbb{N} \to \mathbb{N}$ is called smooth *if we have that f is monotonic and $f(2n) = O(f(n))$.*

All polynomials are smooth functions, so is $\log n$ and $n \log n$, because the product and the sum of two smooth functions is smooth.

Lemma 4.1 (Guessing the size)
Let A_N be a data structure that performs insertions in amortized time $i(N)$, deletions in amortized time $d(N)$ and a query operations in time $q(N)$. The parameter N has to be specified at creation time of the data structure. The data structure allows a total of at most N insert operations. The space usage of A_N is $s(N)$. Assume that i, d, q and s are smooth functions.

Then there exists a data structure B that supports insert in time $O(i(n))$, deletions in amortized time $O(d(n))$, and the query operation in time $O(q(n))$. The *space usage of the data structure is $O(s(n))$. Here n is the number of elements stored in the data structure before executing the operation.*

Proof: We describe how to build data structure B using several instances of A_N. When initializing B set the current guess for the number of inserts to some constant $N = c$. We initialize A_N and process the first N insertions. We perform deletions and other operations without counting them.

From then on we repeat the following process as long as necessary. We rebuild the data structure: We set N' to be twice the number of elements currently stored in A_N. Then we initialize a new $A_{N'}$ and insert all the elements currently stored in A_N. We destroy the old A_N. We set $N = N'$ (and $A_{N'}$ takes the role of A_N). We perform operations until we have processed $N/2$ insertions or $N/4$ deletions (whatever happens first). Then we rebuild the data structure again.

We observe that with this strategy one data structure A_N will not be required to process more than N insertions. None of the data structures A_N (but the very first) contains less than $N/4$ elements (but when initializing A_N). Hence (as q and s are smooth) the query operations take $O(q(n))$ time, and the space usage is $O(s(n))$.

We define a potential function to argue that we can achieve insertions in amortized $3 \cdot i(2N)$ time and deletions in amortized $3 \cdot i(2N) + d(N)$ time, where the parameter N is $\Theta(n)$. Every freshly inserted element has potential $2 \cdot i(2N)$. For every deleted element we keep $3 \cdot i(2N)$ potential. With this potential we achieve the claimed amortized time bounds. Now we have to argue that we can afford to rebuild when we do it. After $N/2$ insertions we have $N \cdot i(2N)$ potential and at most N elements. We initialize a new data structure with parameter $N' \leq 2N$. The potential can pay for the rebuild. After $N/4$ deletions we have at least $\frac{3}{4}N \cdot i(2N)$ potential and at most $\frac{N}{2} + \frac{N}{2} - 1 - \frac{N}{4} < \frac{3}{4}N$ elements. We initialize a new data structure with parameter $N' \leq 2N$. Again the potential can pay for the rebuild.

This completes the proof if we have $i(n) = O(d(n))$. If this is not the case we have to argue that we can pay for the $O(i(n))$ term in the amortized deletion time already when inserting the element. We use this data structure as a black box, we wrap another layer of analysis around it. Let c be a constant such that the $O(i(n))$ term of the deletion is bounded by $c \cdot i(n)$. The choice of c depends on the constant of the smoothness condition of $i(n)$. We define the potential function

$$P(n) = c \cdot \sum_{i=1}^{n} i(n) \; .$$

With this potential function we get deletions in amortized $O(d(n))$ time and insertions in $O(i(n))$ time. □

We can use the way of constructing a data structure that we used in the proof of Lemma 4.1 (p. 77) also in a slightly different setting. If we have a data structure that has an a priori known *nominal size* N, meaning that by our use there are never more than N elements stored in the data structure simultaneously. We only have to transform the data structure in a way that the performance of the data structure depends on N and not on the number of insertions already performed. Additionally we distinguish between two kinds of deletions from the data structure. If we perform a *lazy deletion* for an element l currently stored in the data structure, we can leave it in the data structure (and consider it for queries) as long as we choose to. When the data structure decides to actually delete l (when rebuilding) it might have to perform a function call with l, usually inserting l in a different data structure. In contrast to this is a *strong deletion*, that requires the data structure to immediately delete the element and not consider it in future queries.

We use basically the same technique as in Lemma 4.1, after a certain number of insertions and deletions we rebuild the data structure. This is when we *execute* the lazy deletions, and the lazily deleted elements are no longer part of (queries on) the data structure.

Lemma 4.2 (Periodic rebuilding)
Let A_N be a data structure that performs insertions in amortized time $i(N)$, strong deletions in $d(N)$, lazy deletions in $l(N)$. It supports queries in time $q(N)$. The parameter N has to be specified at creation time of the data structure. The data structure allows a total of at most N insert operations. The space usage of the data structure is $s(N)$. Assume that i, d, l and q are smooth functions.

Then there exists a data structure B that can store up to N elements simultaneously and supports insertions in time $O(i(N))$, forced deletions in $O(d(N))$, lazy deletions in $O(l(N))$, and queries in $O(q(N))$. The space usage of the data structure is $O(s(N))$. The data structure B can trigger a function call when the lazy deletions get executed.

We use this doubling technique in two settings. On the top level we have that the size of the interval tree we build and the merging technique dependent on the number of insertions only. That is, after many insertion and deletions the performance of the data structure decreases, even though the number of simultaneously stored elements stays roughly constant. Additionally we assume that we know the number of insertions in advance. All this is no restriction as we have Lemma 4.1. In a somewhat different setting we have the secondary structures. There we will move the lines between different secondary structures lazily. This is precisely the situation of Lemma 4.2.

4.1.3 Duality transformation

For this part of the thesis we change the point of view of the exposition to the dual problem and consider upper envelopes instead of upper hulls. This duality, as explained, e.g., in [dBvK+97, p. 167], maps points to lines and vice versa in a way, that preserves above/on/below relations between points and lines (relations between points on one vertical line get inverted). We define the dual transform of point $p = (a, b) \in \mathbb{R}^2$ to be the line $p^* := (a \cdot x - b = y)$. For a line $l = (c \cdot x - d = y)$ we define the dual of the line to be the point $l^* = (c, d)$. The situation is exemplified in Figure 4.1.

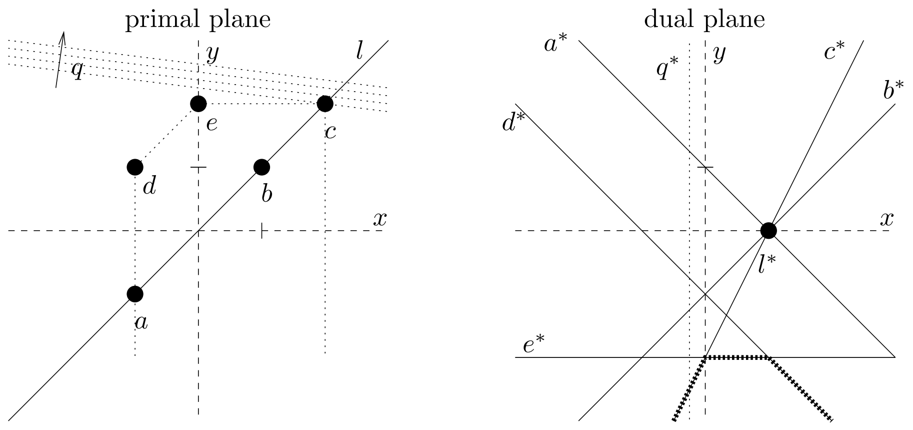

Figure 4.1: An example of the duality transformation. The points d, e and c form the upper hull in the primal, the lines c^*, e^* and d^* form the lower envelope in the dual. The extreme point query for slope q transform into a vertical line query.

Property 4.3
Let l be a non vertical line and $p \in \mathbb{R}^2$ a point. We have

$$p \in l \qquad \Longleftrightarrow \qquad l^* \in p^*$$

and

$$p \text{ lies above } l \qquad \Longleftrightarrow \qquad l^* \text{ lies above } p^*$$

In analogy to the upper hull of a set of points we define the lower envelope of a set of lines. We can also view a set of lines L as a collection of linear functions that have as graphs the lines in L. In this setting the lower envelope is the function $m: \mathbb{R} \to \mathbb{R}$ that achieves point-wise minimum of the linear functions, i.e. $\min_L(x) = \min_{l \in L} l(x)$. The vertical line query amounts to evaluating this minimum at q. We say that a line l is on the *lower envelope* $l \in \mathrm{LE}(L)$ if it contributes with more than a single point to the minimum. This is the case if there exists a $x_1 < x_2 \in \mathbb{R}$ such that $l(x_1) = \min_L(x_2)$ and $l(x_2) = \min_L(x_2)$ (this is in analogy of not considering points on a segment of the upper boundary to be part of the upper hull). For every line $l \in \mathrm{LE}(L)$ we can define its *activity interval* on the lower envelope by

$$I_l^L = \{x \in \mathbb{R} \mid l(x) = \min_L(x)\} \ .$$

For a line $l \in L \setminus \mathrm{LE}(L)$ we set $I_l^L = \emptyset$.

Property 4.4
An extreme point query in the primal setting provides a slope q and asks for the point of the upper hull that has a tangent of slope q. By duality this query transform into a vertical line query. Given a vertical line with x-coordinate q, report the line of L that achieves $\min_L(x)$. The interval I_l^L identifies all the vertical line queries for which l is the correct answer.

The dynamic data structure problem that is dual to the dynamic convex hull allows to insert and delete lines. Queries ask for the minimum of the stored lines at a given vertical line. This data structure problem is also known as *parametric heap*.

We say that a set L of lines is in lower envelope position if all lines in L have a segment on the lower envelope, i.e. $L = \mathrm{LE}(L)$.

Note that the dual transformation is computationally trivial. It is more a change in the point of view.

4.1.4 Interval tree

An interval tree is a data structure that can store intervals in a way that allows efficient containment queries. More precisely for a set J of intervals, the query consists of $x \in \mathbb{R}$ and the answer consists of all intervals $I \in J$ that contain x, i.e. $x \in I$. This data structure is described in detail in the textbook [dBvK+97, page 210]. It is due to Edelsbrunner [Ede80] and McCreight [McC80].

The tree structure of an interval tree $\mathcal{T}$ is that of a search tree (possibly with degree greater than two, for example a B-tree). This tree stores the key values in the leafs. To be able to search in $\mathcal{T}$ every node v stores keys that separate the key values of the subtrees rooted at the children of v. The standard search procedure for value x starts at the root. It determines the predecessor of x among the keys stored at a node and descends to the corresponding child. The search finishes when it reaches a leaf of the tree. The visited nodes of $\mathcal{T}$ form the *search path* for x in $\mathcal{T}$. Every internal node v of $\mathcal{T}$ defines an interval I_v by the elements stored at the leafs of the subtree rooted at v. Let $I \subseteq \mathbb{R}$ be an arbitrary interval I. Let v_I be a node of $\mathcal{T}$ such that $I \subseteq I_v$ and for all of the children u of v we have $I \not\subseteq I_u$. There is precisely one such node v_I, we call it the *canonical node* v_I of I. In other words v_I is the node furthest away from the root such that $I \subseteq I_v$ holds.

Lemma 4.5
Let $\mathcal{T}$ be an interval tree as defined above. Let $x \in \mathbb{R}$ with $x \in I$ for an interval I. Then the search path p for x in $\mathcal{T}$ contains the node v_I.

Proof: Let $I = [a, b]$. Then v_I is (or is above) the least common ancestor of the search paths to a and b. Hence the search path to $a \leq x \leq b$ has to use v_I. □

For the containment queries application we use a balanced binary interval tree. We store every interval at its canonical node. A node v defines the set of intervals L_v stored at v. An interval $I \in L_v$ contains the key stored at v. We maintain two endpoint-lists of the elements of L_v. One list is ordered by increasing right endpoint, the other by decreasing left endpoint. Following the search path for x we can report the set $C \subseteq J$ of intervals containing x in time $O(\log |J| + |C|)$.

We will use the interval tree in a setting where higher degrees make sense. In particular we do not report all containing intervals. The next sections explain why this is advantageous and what precisely we do.

4.1.5 Fast queries only

The construction we present now is going to achieve fast queries. Unfortunately it does not achieve good enough performance in the update operations. This data structure allows us to focus on how the different queries are combined into a single search.

Assume we do a binary merging in the dynamization technique. That is, we have at most $\log n$ different semidynamic sets, for which we have to perform a simultaneous query. One can also see this as performing an "on the fly merge" of the semidynamic sets as part of the query.

As explained in Section 4.1.3 the answer to the query for u is given if we find all the intervals I_l that contain u. In this simple construction we know a priori that we have at most $\log n$ such intervals (one interval from every envelope). To achieve such a query we use a binary interval tree, as described in Section 4.1.4 with the endpoints of the intervals as search keys at the leafs. For a query for x, we perform the containment query on the interval tree for x. This produces with $O(\log n)$ work precisely the up to $\log n$ relevant segments. We can determine the overall answer to the query by inspecting the relevant segments one by one.

The problem of this construction is the deficiency of the updates. Consider for example the changes in this interval tree that are necessary for one merge operation. Deleting the segments/intervals from the tree can be easy if we maintain pointers from the intervals to the node where it is stored. The real problem is that the nodes, where k new segments are to be stored are at arbitrary locations in the tree. Additionally we have to maintain the two ordered endpoint-lists of the segments stored at one node. Using standard techniques it seems impossible to achieve all this in the $O(1)$ amortized time per element that the merging is allowed to take.

This section is not to be misunderstood as a statement that this or a similar construction cannot work. It is only that we could not find techniques that would allow a data structure as outlined above.

Our approach is to change the merging degree of the dynamization technique to $\log n$ (as we use the semidynamic binary merging data structure of Chapter 3, this is rounded to the next power of two, as mentioned in Section 4.1.1). The increased merging degree will allow us to use $O(\log \log n)$ time per line that is moved during a merge operation of $\log n$ semidynamic sets. This allows us in particular to insert the participating line into a different secondary structure. As we will see this change of parameter will propagate, forcing us to introduce several new concepts. The immediate problem we have to address is that we now have $O(\log^2 n)$ semidynamic sets. A query can no longer consider all the containing intervals in the interval tree.

4.1.6 Secondary structures and queries

Instead of considering all the segments that are stored at a particular node v of the interval tree, it is sufficient to find the segment s stored at v that has the lowest intersection with the vertical query line q. This is correct because the query is decomposable. To be in the position to find this segment, we store the lines at the node in a *secondary structure*, that is, we take some fully dynamic lower envelope data structure for every node of the tree. For a secondary structure we assume that the insertion and query time to be $O(\log k)$ for k elements, i.e., we assume that the we already have a good data structure where only the deletion is not as fast as we wish it to be.

To perform a query we consider again the search path for q in the interval tree, collecting the answers from the queries for q on the secondary structures, determining the overall answer as we go along. Assuming that there are only $O(\log^2 n)$ lines stored in a secondary structure, every query takes $O(\log(\log^2 n)) = O(\log \log n)$ time. As we want to achieve $O(\log n)$ queries, we have to change the degree of the interval tree to $B = \log n$, such that the height of the interval tree is $O(\frac{\log n}{\log \log n})$. Then it takes us additionally $O(\log \log n)$ time to determine the next node on the search path for q. So we achieve an overall search time of $O(\frac{\log n}{\log \log n} \cdot \log \log n) = O(\log n)$.

It is a crucial observation that the correctness of a query, only depends upon the fact that all intervals that contain q are stored somewhere on the search path of q. This allows us to store a line anywhere on the path between the root and the canonical node of the interval. In other words we may store an interval I at any node v of the interval tree if we have $I \subseteq I_v$. We can use this freedom to reduce the work necessary for placing the lines into a secondary structure. In return it will increase the size of the secondary structures. This increase in size is tolerable if it is only by a factor $\log^{O(1)} n$ as the performance of the secondary structures is logarithmic in their size (ignoring deletions for a moment).

4.1.7 Insertions into secondary structures

We motivated the increased degree of the merging to $\log n$, to allow every line to be inserted into a new secondary structure as the result of a merge operation. The $O(\log \log n)$ time we charge the participating line at the merge suffices to pay for the insertion into the secondary structure itself. Before we can insert the line into a new secondary structure we have to find an appropriate node of the interval tree. We will later address the problem of deleting the line from the secondary structure it is currently stored in.

Locating the canonical node for an arbitrary interval I takes time $O(\log n)$. Instead of locating the node for the interval for every single segment, we use one common interval for a *chunk* of roughly $b \approx \log n / \log \log n$ consecutive segments of one semidynamic envelope. This chunk size is big enough that locating the node of the interval tree costs $O(\log \log n)$ time per participating segment, which is precisely what we can afford. By storing all segments of one chunk at the same secondary structure we increase the size of the secondary structures by a factor b. The size bound for the secondary structures is given by the number e of semidynamic sets, the degree B of the B-tree and the chunk size b. We calculate it to be $eBb = O(\frac{\log^2 n}{\log \log n} \log n \frac{\log n}{\log \log n}) = O(\frac{\log^4 n}{\log^2 \log n}) = O(\log^{O(1)} n)$. Hence the insert operation on a secondary structure takes still $O(\log \log n)$ (amortized) time.

Dividing a lower envelope into chunks is an easy task. It is more complicated to maintain the partitioning when the envelope changes because of a deletion. When a segment s gets deleted, then s and its neighbors are replaced by a list of new segments. As a result of this change we might have to change some chunks and their interval (join or split). Hence we need to move the segments of a chunk to a different secondary structure. Now the choice of the chunk-size is small enough that moving a chunk in this way induces an overall amortized cost of $O(\log n)+O(\log\log n)\cdot\frac{\log n}{\log\log n}=O(\log n)$. Here the $O(\log n)$ term reflects finding the appropriate node of the interval-tree, and the other term stands for inserting the segments of one chunk into the secondary structure at that node. We can charge the movement of constantly many chunks to the deletion. If the replacement list is so long that we need more chunks, we charge the cost of creating the chunk to the participating lines. This is similar to when creating chunks directly after a merge operation. The lines were not inserted into the interval tree since they participated in a merge of semidynamic envelopes.

Summarizing we can conclude that we could adjust the construction and the parameters in a way that allows us to insert the segments into secondary structures whenever their interval changes.

4.1.8 Changes to intervals

So far we only addressed the problem of inserting the lines into a different secondary structure. We completely ignored the need to also delete a line from the secondary structure it is currently stored in. We will perform these deletions lazily, actually delaying the insertion of the line into the new data structure as well. We call this concept a *lazy movement*. Before we consider how lazy movements work precisely, we have to investigate how the interval (the segment on the lower envelope) of a line changes depending on the operations on the semidynamic envelopes.

If a line l participates in a merging of semidynamic envelopes, its interval shrinks. This is intuitive, the line l is now competing with more lines for a place on the lower envelope. More precisely we have the following lemma.

Lemma 4.6 (Changes to intervals when merging)
Let $S_1,\ldots,S_k$ be sets of lines that get merged to $S=S_1\cup\cdots\cup S_k$. For a line $l\in S_i$ we have $I_l^S\subseteq I_l^{S_i}$.

Proof: Let $l\in S_i$ be line. Let I be the interval of the x-axis for which l is the lowest of all lines in S. Then for all points in I the line l will also be the lowest among the lines of $S_i\subset S$. $\square$

This means that the canonical node of the interval of a line will in general be closer to the leafs. This means that we do not really need to move the lines, they are still stored on the path from the root to their canonical node. Even if the line l is no longer on the lower envelope (but is still not deleted from S, i.e. $I_l^S=\emptyset$) we can store l at *any* node in the interval tree without compromising the correctness of the queries.

Assume we delete a line $l\in S$, leaving us with the semidynamic set $S'=S\setminus\{l\}$. Then the two neighboring segments on $\mathrm{LE}(S)$ extent, i.e., lines may need to be moved to their ancestors in the interval tree. Additionally all the lines that replace l on the lower envelope change their interval. More precisely for such a line $h\in\mathrm{LE}(S')\setminus\mathrm{LE}(S)$ we have $I_l^S=\emptyset$, whereas $I_l^{S'}$ is a nonempty interval. This situation requires us potentially to move lines, the node a line is currently stored at might no longer be allowed for this line.

For every line l we have three nodes in the interval tree: The canonical node v_l that is given by the current interval of the segment this line forms on a lower envelope in a semidynamic set. For the queries we need to store l at a node on the path from v_l to the root. Then we have the node v where l is currently stored. Finally there is the node v_c that is determined by the interval of the chunk. This is where we insert l if it is no longer at another valid node. This is the target of the delayed lazy moves.

Note that we are not concerned with the relative position of v and v_c. It is sufficient that both are above v_l. We already investigated how I_l changes over the life-time of a line l. This in turn tells us how v_l can change.

We observe one inconsistency in this: When $I_l = \emptyset$ the line stored at node v (which is fine), but we do not have $I_l \subseteq I_v$. This is actually not only a formality, but it tells that our definition of I_l does not reflect the possibility of lazy movements precisely enough. We overcome this in the analysis by introducing location justifiers.

4.1.9 Location justifiers

To be in the position to analyze the movement of lines in the interval tree more precisely, we introduce the concept of a *location justifier* for a line that is currently not on the lower envelope of the semidynamic set it is stored in. Let $l \in S_i$ be a line that participates in a merge operation of semidynamic sets $S_1, \ldots, S_k$ into S. Assume that $l \in \mathrm{LE}(S_i)$ and $l \notin \mathrm{LE}(S)$, i.e., because of the merging l is no longer on the lower envelope and we have $I_l^S = \emptyset$. We will leave the line l in the secondary structure where it is currently stored. This does certainly no harm as long as we have $I_l^Q = \emptyset$ for the semidynamic set Q that currently holds l. The set Q can be S, or the result of further mergings where the points in S participate. If l becomes part of the lower envelope of Q, chances are that its new activity interval is a subset of the old one, i.e. $I_l^Q \subseteq I_l^{S_i}$. We can argue that this is the case if both neighbors of l on $\mathrm{LE}(S_i)$ are also in Q (none of the neighbors is deleted).

We define a *location justifier* J to consist of a *base* line l and two *anchor* lines l_l and l_r. The lines $M = \{l_l, l, l_r\}$ are in lower envelope position, the resulting interval $I_J := I_l^M$ is the reason for defining justifier J. It is possible that one of the anchors is missing, in this case the interval of J is unbounded. We require that all the defining lines of J are in the same semidynamic set. If one of the anchor lines gets deleted from the fully dynamic data structure, we destroy the location justifier J.

We will not make location justifiers explicit in our data structure. They are only used as an annotation of the data structure when analyzing its performance.

4.1.10 Deletions from secondary structures

We already discussed movements of lines in the interval tree. So far we only considered how to locate appropriate secondary structures and how to account for inserting the lines there. Now we will discuss how to delete lines from secondary structures and what the hinted at lazy movements precisely are.

We have to distinguish two types of deletions. Assume the line l is the parameter of a DELETE operation of the overall fully dynamic data structure. Then we immediately have to delete l from the secondary structure it is currently stored in. Otherwise we risk incorrect outcomes of query operations. This is not a real problem, deleting a single line from a secondary structure is not too expensive as the secondary structures have only size $O(\log^4 n)$. We will come back to this when we discuss the bootstrapping of the data structure in Section 4.1.11.

The other reason for deleting a line l from a secondary structure at node u is that we decided to store it in a different secondary structure, at a different node v of the interval tree. This happens either because of a merge operation or because of a deletion of a line in a semidynamic set. Given that l can be stored on the path between its old canonical node and the root, it can very well be that u is also a feasible node for the new canonical node. For a merging operation Lemma 4.6 (p. 83) guarantees that this is the case. For lines becoming part of the lower envelope again as a result of deleting a line, the location justifiers guarantee this.

This suggests that we can perform lazy deletions. Only when we anyway rebuild the secondary structure (Lemma 4.2, p. 79), we actually execute the deletions. We do not want to store lines in several secondary structures (this might cause expensive deletions from the fully dynamic data structure). Therefore we adopt a *lazy movement* strategy. We do not only delay the deletion of the line, but we also delay the insertion of the line at its new location. Only when we decide to rebuild the secondary structure and execute the lazy deletions, we also perform the delayed insertions.

For a line l that becomes part of the lower envelope of the set it is currently stored in, it can happen that the activity interval of l changes so much that it is no longer stored at a feasible node of the interval tree. In this case we insert l at its canonical node. Because we do not allow lines to be stored in more than one secondary structure, we perform an (expensive) explicit deletion of l from the secondary structure l is currently stored in, and insert l it at its canonical node. We say that we perform a *forced move*. The location justifiers will play an important role when analyzing how many forced moves can be necessary.

4.1.11 Parameters, bootstrapping, and analysis

Even though we mentioned the choice of suitable parameters when introducing the corresponding concepts, we summarize the interaction of the different parameters here.

For the dynamization technique we choose the degree parameter $r = \Theta(\log n)$. This allows us to use $O(\log\log n)$ amortized time to insert the lines into a secondary structure after every merge. We choose the degree of the interval tree to be $B = \Theta(\log n)$. This is necessary for fast queries. Remember that querying a secondary structure takes $O(\log\log n)$ time since secondary structures have size $\log^{O(1)} n$. We choose the chunk-size parameter $b = \Theta(\log n/\log\log n)$. This is big enough that the $O(\log n)$ cost of locating the canonical node for a chunk results in an $O(\log\log n)$ cost per element. It is small enough that locating a chunk and inserting all its elements into new secondary structures (lazily) costs $O(\log n)$.

All this is under the assumption that we already have a data structure that achieves insertions in amortized $O(\log n)$ time and extreme point queries in worst-case $O(\log n)$ time. Such a data structure exist. We take Preparata's insertion only data structure [Pre79]. This data structure maintains a balanced search tree of the upper hull (lower envelope) and uses it to react to insertions by finding tangents. This takes $O(\log n)$ time. We handle deletions by rebuilding the data structure in $O(n)$ time, reusing that the points are already lexicographically sorted.

In the first step of the bootstrapping we use Preparata's data structure for secondary structures. We pay for a forced move of the base line of a location justifier J whenever we delete an anchor line of J. The forced move of the base of J can happen now, later or not happen at all. A line can become an anchor line of a location justifier (we create a location justifier) only once per merging operation. Therefore the deletion of a line destroys at most $O(\log n/\log\log n)$ location justifiers. The deletion of a location justifier requires us to pay for a forced move. A deletion from the fully dynamic data structure costs therefore the deletion of a point from the semidynamic data structure and $O(\log n/\log\log n)$ deletion from secondary structures. By the size of the secondary structure and the linear deletion time of Preparata's data structure this totals to an amortized cost of $O(\log^5 n)$.

In the second bootstrapping step we use the above achieved data structure for the secondary structures. With the same analysis as before we see that deletions cost no more than $O(\log n/\log\log n)\cdot O(\log^5(\log^4 n)) = O(\log n \log^5 \log n)$. The problem is that one deletion still has to pay for one forced move from each possible rank of the merging. This is not necessary. We can move some of the cost for forced moves to the insertion without changing the asymptotic performance of an insert operation. We do this by introducing *barriers* in the merging. More precisely we introduce a barrier parameter $P = \Theta(\sqrt{\log n})$. Every P^{th} level of the merging is a barrier level. When merging the sets $S_1 \ldots S_k$ into S at a barrier level, we pay every line $l \in S$ a forced move and are therefore allowed to destroy all the location justifiers with anchors (and base lines) in S. A line can only participate in $O(\frac{\log n}{\log\log n \cdot P}) = O(\sqrt{\log n})$ mergings on barrier levels. The amortized cost for the forced moves is $O(\sqrt{\log n})\cdot O(\log^5 \log n) = O(\log n)$. This is asymptotically no slow down for the insertions. Now any line can be the anchor of at most P location justifiers at a time. The amortized cost in the interval tree for a deletion is hence $P \cdot O(\log^5 \log n) = O(\log n)$.

4.2 Real construction

As all the constructs are now introduced, we order this exposition in a way that avoids forward references at the price of being unmotivated.

Theorem 4.7 (Speed up construction)
Let $U(n)$ be a nondecreasing positive function, with $U(n) \geq \log n$. Assume there exists a data structure for the dynamic lower envelope problem supporting INSERT *in amortized $O(\log k)$ time,* DELETE *in amortized $O(U(k))$ time, and* VERTICAL LINE QUERY *in $O(\log k)$ worst-case and amortized time, where k is the total number of lines inserted. Assume the space usage of this data structure is $O(k)$.*

Let $b(n)$ be a function such that $b(n)\cdot U(\log^4 n) = O(\log n)$. Then there exists a data structure for the dynamic lower envelope problem supporting INSERT *in amortized $O(\log n)$ time and* DELETE *in amortized $O(\frac{\log n}{\log\log n}\cdot\frac{U(\log^4 n)}{b(n)})$ time, and* VERTICAL LINE QUERY *in worst-case and amortized $O(\log n)$ time, where n is the total number of lines inserted. The space usage of this data structure is $O(n)$.*

The remainder of this section is devoted to prove Theorem 4.7.

Let N be an estimate on the number of points stored in the data structure and an upper bound on the number of inserts, just as in Lemma 4.1 (p. 77). We maintain that $N \geq 4$, such that $\log\log n$ is well defined and ≥ 1. Let S be the set of lines currently stored in the data structure.

We maintain a partition of S into (semidynamic) (sub)sets $S_1, \ldots, S_e$. Every S_i has a rank. We define the merging degree $r = 2^{\lceil \log\log N \rceil}$, an upper bound on the number of sets of equal rank. This leads to an upper bound e on the number of sets. The choice of r guarantees $e = O(\log^2 N)$. The points of a set S_i are stored in a semidynamic lower envelope data structure conforming to (the dual of) Definition 3 (p. 19). For every line $l \in S_i$ we determine its activity interval $I_l^{S_i}$. We say that l is *inactive* if $I_l^{S_i} = \emptyset$, otherwise it is *active*. For active lines we determine a *placement interval* $P_l \supseteq I_l$. For each S_i the segments on $\mathrm{LE}(S_i)$ are partitioned into a sequence of chunks of size $\Theta(b)$ with $b = \lceil \log n/\log\log n \rceil$. We determine a common placement interval P_l for all lines of a chunk. The boundaries of P_l are given by the rightmost and leftmost segments of the chunk. If $\mathrm{LE}(S_i)$ consists of less than b segments, all of $\mathrm{LE}(S_i)$ forms one chunk with placement interval $P_l = \mathbb{R}$ for all lines $l \in \mathrm{LE}(S_i)$.

We maintain an interval tree $\mathcal{T}$. The leaves of $\mathcal{T}$ are the endpoints of the intervals P_l. We do not delete leafs from $\mathcal{T}$, i.e., $\mathcal{T}$ is monotonically growing. The tree $\mathcal{T}$ is a monotonic B-Tree with degree parameter $B = \lceil \log N \rceil$, as described in Section 2.3. For every node $v \in \mathcal{T}$ we have a secondary structure Q_v, that is, a fully dynamic lower envelope data structure with the characteristics assumed in the theorem.

For a node v of the B-Tree, we have records necessary to perform the B-Tree operations and additionally the following information: A pointer to the secondary structure Q_v, the interval I_v, a balanced search tree for the search key values used at v. We say that it is *admissible* to store l in Q_v if $I_l \subseteq I_v$.

For a secondary structure Q_v we maintain a pointer to the representation of node v, and a counter C_v, holding the number of lines inserted into Q_v since the last rebuild. We maintain globally the *nominal* size N_s for secondary structures, that is, the value N in Lemma 4.2 (p. 79). We set $N_s = 4 \cdot \lceil \log N \rceil^4$. We additionally have the starting point C_v of a doubly linked list of chunks that are supposed to store their lines in Q_v. The list C_v identifies the (active) lines that should be stored in Q_v when we rebuild it.

Every active line l is stored in precisely one secondary structure Q_v, where v is on the path from the root to the canonical node of the interval I_l. An inactive line l can be stored in any (but only one) secondary structure, or it is not stored in a secondary structure. If a line l is deleted from S, it is not stored in any secondary structure.

For every line $l \in S$ we have a record holding the following information: a pointer to the semidynamic set S_i that contains l, a pointer to the secondary structure Q_v the line is currently stored, the interval $I_l^{S_i}$, and pointers to the neighbors of l in S_i. This information is always up-to-date.

For a chunk c we maintain a record holding the following information: The interval I_c that contains all the segments of the chunk (the placement interval P_l of the lines in the chunk), a pointer to the secondary structure Q_u that should store the lines contained in this chunk, a pointer to the first and last line of the chunk, and a counter C_c holding the current size of the chunk. We store the chunk size parameter $b = \lceil \log n / \log \log n \rceil$ globally. We maintain the invariant that chunks have sizes between b and $2b - 1$, unless the lower envelope of the semidynamic set has less then b lines. In this case we have one trivial chunk with $I_c = \mathbb{R}$.

For the semidynamic sets S_i, we have a pointer to the semidynamic data structure holding the elements of S_i, a pointer to the leftmost line on $\mathrm{LE}(S_i)$, and a pointer to a rank-record. The rank-record for rank j has a counter that reflects the number of semidynamic sets of rank j. It is the starting point of a list of the records for the semidynamic sets of rank j. It has a pointer to the rank-record for $j + 1$. This pointer is nil if j is the highest rank of a semidynamic set in this fully dynamic data structure.

4.2.1 Vertical line query

Let S be the set of all the lines currently stored in the overall fully dynamic data structure. A query for value x requires us to determine $q(x) = \min_{l \in S} l(x)$.

For every node v on the search path p in $\mathcal{T}$ for x we perform the query with value x at the secondary structure Q_v. This gives $|p| = O(\log n / \log \log n)$ answers. We return the minimum f of these answers as the result.

By definition we get $f \geq q(x)$. Let $l \in S$ be a line with $l(x) = q(x)$. Let I be the interval of l on $\mathrm{LE}(S)$. Let S' be the semidynamic set with $l \in S'$, and I' the interval of l on $\mathrm{LE}(S')$. Then we have $x \in I'$ and by the construction of the interval tree, l is stored in one of its allowed secondary structures, that is, at a node v on path p. Therefore the result of the query of the secondary structure at v is $q(x)$, and the overall result is $f = q(x)$.

We will consider other queries in Section 4.5.

4.2.2 Insert

To insert a point p into our data structure, we create a singleton set $S_o = \{p\}$ (and semidynamic data structure) of rank 0. We insert S_o into the list of semidynamic sets of rank 0. It might be that the number of sets of rank 0 is now r, our merging degree. In this case we merge the sets of rank 0 by applying Theorem 3.17 (p. 46) $r-1$-times in a tree-like fashion. As we create a new set of rank 1 this process can cascade. If we create a set of a rank that is higher than the highest rank we have so far in this fully dynamic data structure, we create a record for this rank.

Let $S_1, S_2, \ldots, S_r$ be semidynamic sets of the same rank that get merged into $S = S_1 \cup S_2 \cup \cdots \cup S_r$. By the construction of the semidynamic sets we have access to $\mathrm{LE}(S_i)$ before the merge and to $\mathrm{LE}(S)$ after the merge. We perform the merging along a complete binary tree of height $\log r$, which is by our definition of r the integer $\lceil \log \log N \rceil$.

Only for the analysis of the algorithm we create location justifier with base l for every line $l \in \mathrm{LE}(S_i)$, with the anchor lines given by the neighbors of l in $\mathrm{LE}(S_i)$.

First we mark all lines $l \in S_i$ to be inactive. This is done by following the links of $\mathrm{LE}(S_i)$, setting the interval $I_l := \emptyset$. We abandon all the chunk records, taking them out of the doubly linked list at the secondary structures. This means that line l is lazily deleted from the secondary structure Q_v it happens to be stored in. As soon as Q_v is rebuilt, l is actually no longer in Q_v. In the record of l we keep the pointer to Q_v as long as l is stored there.

We perform the merge operation of the semidynamic sets. As a result we have access to $\mathrm{LE}(S)$. We walk along the set $\mathrm{LE}(S)$, updating the interval I_l for every line $l \in \mathrm{LE}(S)$ and counting $m = |\mathrm{LE}(S)|$. If $m \leq 2b-1$ we create a single chunk. Otherwise we create chunks of size b as long as the number of remaining lines is greater than $2b-1$. With this policy the last chunk also has size between b and $2b-1$.

To create a chunk c, we instantiate a chunk record. Then we walk along the records of the lines to be stored in c and update their chunk pointer to c. We set the pointer to the first and last line of the chunk appropriately. We compute I_c. We insert the endpoints of I_c as leafs into $\mathcal{T}$. This might lead to restructuring of $\mathcal{T}$ which we discuss below. We perform a search in $\mathcal{T}$ determining the canonical node v for I_c. We set the pointer of c to Q_v. We walk along the lines of c. We insert line l into Q_v if l is not stored in another secondary structure Q_u. By the nature of a merge operation and Lemma 4.6 (p. 83) the node u is an admissible location for l.

If we have to split a sequence of nodes (along a path in the interval tree) because of the insertion of new leafs, we first perform all the split operations and update the intervals I_v of the affected nodes. If we split the node v, we move all the lines stored in the secondary structure Q_v into a (initially empty) list L_s. We destroy the old secondary structure. We include the chunk records stored at v into a list L_c of affected chunks. We create a new (empty) secondary structure and an empty list of chunks at the newly created nodes.

We walk along the list L_c and determine for a chunk $d \in L_c$ the new canonical node for I_d in the interval tree. By the nature of the split operation this will be one of the newly created nodes, but this is not important for the analysis. Then we consider all the lines stored in L_s. We insert the lines at the secondary structure given by the chunk they belong to. Apart from now executed lazy movements of lines, this are the newly created secondary structures. Again this is not important for the analysis.

During these operations (because of executed lazy moves), it can happen that a secondary structure reaches the size limit N_s. If this is the case, we detach the fully dynamic data structure from the node, and create a new one. Then we walk along all lines stored in the detached secondary structure and insert the lines l into the secondary structure that is given by the chunk. We update the pointer of l to the secondary structure it is stored in. If during this process another secondary structures reaches its size limit, we include it into a list L_o, and mark it as full. We do not perform insertions into full secondary structures, but merely insert the lines into a list L_v. The lines in L_v are inserted into a new secondary structure as soon as we rebuild Q_v. We continue to rebuild secondary structures until the list L_o is empty.

4.2.3 Delete

Assume we globally delete the line l that is currently stored in the semidynamic set S_i, i.e., $l \in S_i$. We have to adapt $\mathcal{T}$ to be based on the new set $S'_i = S_i \setminus \{l\}$.

For the analysis (and only there) we delete all the location justifiers where l is an anchor, each time paying a forced move to the base line.

We delete l from the secondary structure it is currently stored in. We perform the deletion on the data structure representing S_i. By the interface of the semidynamic data structure, this yields a new representation of LE(S'_i), where the former neighbors of l on LE(S_i) identify the list L' of fresh lines, i.e., lines that changed their activity interval from LE(S_i) to LE(S'_i) (most of them away from $\emptyset$).

We check whether we have to determine new chunks of LE(S'). If this is not the case because the deletion did not change the boundary of a chunk and the size restriction is obeyed, we insert the fresh lines into the secondary structure Q_u at the node u given by the chunk. Otherwise we have to rearrange some chunks. By the nature of deleting one line, we know that there are at most two chunks of LE(S_i) affected. If there is only one affected chunk and it is getting too small, we can take either neighbor and consider it affected. We delete the affected two chunks just as we deleted the chunks of sets that get merged in Section 4.2.2. We define an extended notion of fresh lines, defining the set $L'' \supset L'$ of all lines that currently are not in a chunk. The lines of L'' form a stretch on LE(S'_i), and we have counted them. We perform the steps of creating new chunks (most of them of size b) for the lines in L'' as described in Section 4.2.2. We also insert the new chunks into $\mathcal{T}$ as described there.

For all fresh lines $l \in L'$ we check whether they are stored at admissible locations, i.e., we have $I_l \subseteq I_v$ for the node v where l is stored. If this is not the case, we are required to perform a forced move. We delete l from the secondary structure it is currently stored in, and insert it into the secondary structure that is given by the chunk l it is (now) part of.

Be aware that our choice to insist on chunks having size between b and $2b - 1$ and creating many chunks of size b might induce some unnecessary work. As this apparently has no bad influence on the asymptotic analysis, we stick to this easy to explain scheme.

4.3 Analysis of the speed up construction

We analyze the performance of the data structure looking back in time, for an arbitrary point in time. The total work performed up to this time has to be bounded by the sum of the claimed amortized costs of the performed operations.

4.3.1 Number of chunks invented

Let M denote the number of chunks ever created up to the current point in time. Then the size of $\mathcal{T}$ is linear in M, as we introduce precisely two leafs in $\mathcal{T}$ for every chunk we create.

We note that most of the chunks we create contain only lines that are inserted into $\mathcal{T}$ the first time after participating in a merge operation (either at the merge operation or later as part of a deletion). As every line participates only in $\log_r N$ merge operations, we can have at most $N \cdot \log_r N/b = N \cdot \frac{\log N/\log\log N}{\log N/\log\log N} = N$ such chunks.

The remaining chunks are at most 5 per every deletion: two old chunks that are affected by the deletion, holding at most $4b-2$ lines. These lines might be (in the worst case) distributed into chunks of size b, 3 chunks on one end and 2 on the other end. We have in total at most 5 new chunks that contain lines that have been part of the lower envelope on the same merging level.

This discussion leads to the following lemma:

Lemma 4.8
Let N be the limit on the number of insertions of the fully dynamic data structure. Then the number M of leafs in $\mathcal{T}$ at time t is bounded by $M = O(N)$.

Lemma 4.9
The overall amount of work for inserting chunks into $\mathcal{T}$ is $O(M \log N)$, that is, amortized $O(\log N)$ per inserted line.

4.3.2 Work in splits

Now we have to bound the total work we spend in splitting nodes of $\mathcal{T}$ up to some point in time.

We charge the cost for splitting a node entirely to the newly created node. That is, if we split the node u and create a sibling v of u, we pay $O(\log N)$ to every chunk c that is currently supposed to be stored at u (before the split). This certainly pays for finding the new canonical node of c and also to pay for inserting all lines of c into Q_u or Q_v. This is again due to the choice of b and the insertion performance of the secondary structures.

If we split a node on the 4 levels closest to the leafs of $\mathcal{T}$, we know that the split can only have affected as many chunks as there are leafs below the node. On level 1 we have at most N/B nodes, each having at least B leafs below it, on level 2 at most N/B^2 nodes, each having at least B^2 leafs below it, and so on. As we only consider 4 levels in this way, we have $O(M)$ many chunks affected by split operations accounted for in these 4 levels. This gives a total of $O(M \log N)$ work.

There are at most $B^{\log_B N - 4} = O(N/B^4) = O(N/\log^4 N)$ nodes in $\mathcal{T}$ remaining to be accounted for. We can have at most $e \cdot B = O(\log^3 N)$ many chunks stored at one node at a time. We pay $O(\log N)$ per affected chunk, so the total work we have to pay for splitting nodes is $O(\frac{N}{\log^4 N} \cdot \log^4 N) = O(N)$.

Lemma 4.10
The total amount of time used to split nodes in $\mathcal{T}$ is bounded by $O(M \log N)$, that is, $O(\log N)$ amortized per inserted line.

4.3.3 Forced moves

In this section we will derive an upper bound on the number of forced moves that we have to perform in total up to some point in time.

The concept of a location justifier is already introduced in Section 4.1.9. Location certificates are not part of the data structure. They are to be understood as an annotation of the data structure. We maintain them only for the purpose of analyzing running times. In this respect they are very similar to a potential function for the amortized analysis.

In Section 4.2.2 we already stated when to instantiate a location justifier. Let line $l \in \mathrm{LE}(S_i)$ be on the lower hull of its semidynamic set S_i. Assume S_i gets merged with some other semidynamic sets into S. If l is not on $\mathrm{LE}(S)$ we instantiate a location justifier J with base l and the neighbors on $\mathrm{LE}(S_i)$ as the anchors. By the nature of merging, all lines of J are in S. The lines of J will from now on not be in different semidynamic sets.

A location certificate J becomes unnecessary if the base line gets deleted, removed from the secondary structure because of a clean-up, or it becomes part of a lower envelope again, with an active interval that is a subset of the interval of J. In these cases we just forget about J (remove it from our annotation of the algorithm). A location certificate J becomes invalid, if one of the anchor lines gets deleted. In this case we have to pay for a forced move of the base line.

Let l be a line that is inactive, but has a valid and necessary location certificate J. Then l is stored in a secondary structure at node v of $\mathcal{T}$, where v is an admissible node of $\mathcal{T}$ for I_J. If l becomes active again (as a result of a deletion), we can be assured that the new active interval I_l of l is a subset of I_J, which means that l is not part of a forced move.

It is sufficient in the amortized analysis to pay for a forced move when we make a location certificate invalid. That is, we charge the deletion of line l with the costs for forced moves for all location certificates where l is an anchor. The line l can be anchor for at most two location certificates for every merge operation it participated in.

As already hinted at in Section 4.1.9, we introduce the notion of a barrier to account for some of the forced moves as part of the insertion. Let $b(N)$ denote the number of barriers we decided upon (it can easily be that we have $b(N) = 1$).

We distribute $b(N)$ barriers equally over all our merging levels. That is, we have a distance between barriers of at most $O(\frac{\log N}{\log\log N \cdot b(N)}$. Whenever we perform a merge operation with the resulting set S of a barrier rank, we invalidate (delete) all location justifiers, and pay one forced move for the lines of S that are not in $\mathrm{LE}(S)$. The induced extra cost for the insertion is $b(n) \cdot U(\log^4 n) = O(\log n)$ by the assumption about b in the theorem.

At any point in time a line l is anchor for at most $O(\frac{\log N}{b(N) \log\log N})$ location justifiers. If we delete l we have to pay for every location justifier a forced move. Given the cost of a forced move, this totals for one deletion to $O(\frac{\log N \cdot U(\log^4 N)}{b(N) \log\log N})$.

Together with the doubling technique of Lemma 4.1 (p. 77), we achieve the amortized running times claimed in Theorem 4.7 (p. 86).

4.3.4 Space usage

Now we also have to analyze the space usage of the data structure. The interval tree and the chunks use $O(N)$ space. In the secondary structures every line uses $O(1)$ space as it is stored in at most one secondary structure. This totals to a space usage of $O(N)$. Together with Lemma 4.1 (p. 77) this is the claim of Theorem 4.7 (p. 86).

This completes the proof of Theorem 4.7.

4.4 Bootstrapping

Lemma 4.11
There exists a data structure for the fully dynamic lower envelope problem supporting Insert *in worst-case $O(\log n)$ time,* Delete *in $O(n)$ worst-case time and* Vertical line query *in worst-case $O(\log n)$ time. The space usage is $O(n)$. The parameter n denotes the size of the stored set before the operation.*

Proof: We basically take Preparata's data structure from [Pre79]. More precisely we keep the lower envelope in a (2,4)-tree. For the insertion of line l we search for the two intersection points u, v with the current lower envelope. If they exist, we remove all the segments between u and v from the (2,4)-tree. We keep a second (2,4)-tree storing the elements of the set in lexicographical order. For a deletion we rebuild the (2,4)-tree holding the lower envelope from scratch, using Graham's scan in Andrew's version. By the performance of (2,4)-trees we achieve the claimed time bounds. □

Theorem 4.12
There exists a data structure for the fully dynamic planar convex hull problem supporting Insert *and* Delete *in amortized $O(\log n)$ time, and* Extreme point query, Tangent query *and* Neighboring-point query *in $O(\log n)$ time, where n denotes the size of the stored set before the operation. The space usage is $O(n)$.*

Proof: We take Preparata's data structure as described in Lemma 4.11 and the semidynamic data structure described in Theorem 3.17. We apply Theorem 4.7 (p. 86) with $b(n) = 0$. This gives us a data structure with $O(\log n)$ amortized insert and query and $O(\log^5 n)$ amortized deletion time. We apply again Theorem 4.7 (p. 86) with $b(n) = \sqrt{\log n}$. This reduces the number of forced moves that are accounted to one deletion. The corresponding term in the amortized cost of a deletion is reduced to $O(\frac{\log n}{\log\log n} \cdot \frac{\log^5 \log^4}{\sqrt{\log n}}) = O(\log n)$, leading to the time bounds of the theorem. The space bounds come directly from Theorem 4.7 (p. 86). □

4.5 Tangent / arbitrary line query

If the only query we are interested in is extreme point / vertical line query, the presented data structure is sufficient. If we are interested in different queries, it is not obvious how useful the query data structure is.

In the primal setting we can imagine the following query: given a point $p \notin \mathrm{UC}(S)$ in the plane, what are the two common tangent lines of p and S? Translating the query into the dual setting, we ask for the two intersection points of an arbitrary line with the lower envelope of L. Given that we may perform vertical line queries, we can easily verify a hypothetic answer. Therefore it is sufficient to consider the situation under the assumption that the line intersects the lower envelope twice.

Let us focus on finding the right intersection of line l with the lower envelope $\mathrm{LE}(S)$ for a set of lines S, that is, we are in the dual setting. This is again an optimization task, we ask for the line of S with slope strictly greater than l, that intersects l furthest to the left.

Now we need the geometric argument that we can use such queries in the secondary structure to navigate in $\mathcal{T}$ in a way that leads to the correct answer.

We use the following fact about arbitrary line queries to navigate in the interval tree of our data structure.

Lemma 4.13
Let a and b be two vertical lines, a to the left of b. Let $S' \subseteq S$ be two sets of lines such that the lower envelope of S' at a and b coincides with the lower envelope

of S. Assume that an arbitrary line query for a line ℓ on S' results in the right intersection point t. If t lies between a and b then also the right intersection T of ℓ with S (if it exists) lies between a and b.

Proof: By the definition of the right intersection point as the leftmost intersection of ℓ with the lines of greater slope in S' and S we immediately have that T is not to the right of t and hence not to the right of b.

Assume that the left intersection of ℓ with LE(S') is also between a and b. If both intersections of ℓ with LE(S) exist, they are between the intersections of ℓ with LE(S'). In this case the lemma holds.

Otherwise we know that the intersection v of ℓ with a is below the intersection u of LE(S') with a. Assume that T is to the left of a. Let h be a line of $S \setminus S'$ that contains T and has greater slope than ℓ. Then the intersection of h with a is below v. But then the lower envelope of S intersects a at or below v, contradicting the statement that the two lower envelopes coincide on a. □

Using this lemma, we can process an arbitrary line query for ℓ in the following way: Starting at the root, we perform the query for ℓ at the secondary structure Q at the root. If we find that there are no intersections of LE(Q) with ℓ, we know that there is no intersection of ℓ with S. Otherwise let u be the found right intersection point. We find the slab that is defined by the keys stored at the root that contains u. This slab identifies a child c of the root. We continue the search at c in the same way, that is, we perform another arbitrary line query to the secondary structure. Now we take the leftmost of the two results (stemming from c and the root) and use it to identify a child of c. This process we continue until we reach the leaf level. There we take the leftmost of all answers we got from secondary structures, and verify it by performing a vertical line query. It is necessary to use the currently leftmost answer as we allow lines to be stored higher up in the tree. It is necessary to verify the outcome, we cannot exclude the case that all queries to secondary structures find two intersection points, whereas there is no intersection of ℓ and LE(S).

For the bootstrapping we have to argue that we can perform arbitrary line queries in $O(\log n)$ time. As this kind of search is the main ingredient to perform fast inserts, this does not require an additional algorithm.

As the arbitrary line query in $\mathcal{T}$ performs precisely one query to a secondary structure at every level of $\mathcal{T}$, we get an overall worst-case and amortized time bound of $O(\log n)$.

If we run this query for a point $p \in S$, we determine whether $p \in$ UH(S) and if this is the case, we find the neighbors of p in UH(S). We also realize if p is a point on a segment of Bd(S). Tangent queries allow us to report a stretch of k consecutive points on the upper hull of S in time $O(k \cdot \log n)$. This is by a $O(\log n)$ factor slower compared to an explicit representation of the convex hull.

4.6 Kinetic heaps

By the dual transformation the dynamic planar convex hull problem with extreme point queries transforms to the parametric heap problem, an extension of a priority queue as explained in Section 4.1.3. We can insert and delete lines $l_i := (y = ax + b)$ into the heap and ask for time t (which of the currently stored lines achieves) the minimum $\min_i l_i(t)$.

A *kinetic heap* is a parametric heap with the additional restriction that the sequence of FIND-MIN(t) queries has non-decreasing t-values. In other words the *time* at which we are interested in the current minimum of the parametric heap is never going backward. Sometimes the requirement that time has to progress covers also the DELETE-MIN(t) operation. This is for example the case in the recent work on kinetic heaps by Kaplan, Tarjan and Tsioutsiouliklis [HTK01]. In our setting this is not necessary as the general DELETE operation already takes only $O(1)$ amortized time.

To not confuse the kinetic and the ordinary FIND-MIN operation, we introduce the new name KINETIC-FIND-MIN. In this section we describe a data structure that achieves KINETIC-FIND-MIN operations in amortized $O(1)$ time. This data structure is based on the same dynamization technique as the fully dynamic planar convex hull (lower envelope) data structure. It can even share the semidynamic data structures with such a fully dynamic data structure. All we do here is to describe an alternative (addition) to the interval-tree construction that is optimized for kinetic queries.

4.6.1 Kinetic queries in the semidynamic setting

We first introduce a search algorithm that achieves $O(1)$ amortized KINETIC-FIND-MIN operations for the semidynamic (merging) data structure as defined in Chapter 3. We use the linear list representing the lower envelope of the semidynamic set. We keep a pointer (finger) to the last answer to a KINETIC-FIND-MIN operation. For a new query we perform a linear scan on the segments currently forming the lower envelope until we find the correct answer. If a deletion removes the line our finger is pointing to, then we search the segment of the new lower envelope that achieves the minimum for the time of the last query.

Lemma 4.14 (Bounded advancing)
The linear scans on the current lower envelope of a semidynamic merging structure S, storing n lines, advances over $O(n)$ segments in total.

Proof: We perform the analysis backward in time. Let $E \subset \mathbb{R}$ be the set of all endpoints of segments on the lower envelope of S. Then E is not affected by considering the updates to the semidynamic set in inverted order. This means that we have to consider insertions only. When inserting a line, it can introduce at most two new endpoints of segments on the lower envelope. Hence we have $|E| = O(n)$.

Let $t_i \leq t_{i+1}$ be the times of two consecutive KINETIC-FIND-MIN operations. The number of segments scanned over when performing the query for t_{i+1} is bounded by the number of points in $E_i = \{x \in E \mid t_i < x < t_{i+1}\}$. The sets E_i are disjoint from each other. The total time spend in scans that advance over the boundary of a segment is hence bounded by $|E|$. $\square$

The $O(n)$ time spent advancing over segments is charged to the insertion of the point into the semidynamic data structure. This yields an amortized $O(1)$ time bound for the KINETIC-FIND-MIN operation.

4.6.2 Combining semidynamic sets: a query structure

The approach we follow to combine the semidynamic sets is somewhat similar to the interval tree approach for general queries, only a lot simpler. In contrast to the interval tree we only maintain one secondary structure. Again the $\log n$ merging degree allows us at every merging level to insert all lines into the secondary structure once. We also perform a bootstrapping that improves only the deletion time.

We augment the query to return the segment achieving the minimum at time t, including the right endpoint of this segment. This additional information can be seen as an expiration time: if the set is not changed and the next query is before this time, the answer will not be changed. In the semidynamic case we have this information immediately available.

To combine the results of the $O(\log^2 n)$ semidynamic data structures we use one secondary structure. This secondary structure we assume to allow insertions in $O(\log k)$ amortized time, and kinetic queries in $O(1)$ amortized time. For the current time t_c (the time of the last KINETIC-FIND-MIN operation) the secondary structure stores all lines that are currently answers on the semidynamic data structures. We set the nominal size of the secondary structure to $\Theta(\log^2 n)$, the number of semidynamic data structures. We perform lazy deletions and rebuilding according to Lemma 4.2 (p. 79). Additionally we keep a (2,4)-tree of the endpoints of the segments that are currently stored in the secondary structure (not including the lazily deleted lines). We also keep the smallest such endpoint explicit.

For a KINETIC-FIND-MIN query for time t we do the following: We first check with the smallest endpoint of a segment in the secondary structure, whether the current secondary structure is up-to-date. If this is not the case we delete all endpoints from the (2,4)-tree that are smaller than t. We lazily delete the corresponding lines from the secondary structure. For all semidynamic sets that are no longer represented in the secondary structure we perform a KINETIC-FIND-MIN query for time t. We insert the returned line into the secondary structure and insert the endpoints of the segments into the (2,4)-tree. We update the smallest endpoint when updating the (2,4)-tree. Now we perform the query on the secondary structure for time t. This gives both the line and the endpoint of the current minimal segment.

For a merge operation we remove the current endpoints from the (2,4)-tree and perform lazy deletions of the lines in the secondary structure.

For a delete operation we perform the delete on the semidynamic data structure, and if the deleted line is currently stored in the secondary structure we delete (not lazily) it from the secondary structure. We also delete the endpoint from the (2,4)-tree. We delay inserting the new result of a query for the current time until the the next query operation. Only then we perform a query in the semidynamic data structure and insert the result into the secondary structure and the (2,4)-tree.

4.6.3 Analysis

Every line has to pay for one insertion and deletion into the (2,4)-tree and for one insertion and lazy deletion in the secondary structure for every merging level. This totals to $O(\log n)$ amortized time, charged to the insertion of the line.

A deletion pays for the deletion of one line in the secondary structure and for reinserting one line into the (2,4)-tree and secondary structure. This amounts to $O(\log\log n + U(\log^2 n))$ where $U(k)$ is the amortized deletion time for a secondary structure.

Using Preparata's [Pre79] semidynamic insertion only data structure, we achieve insertions in $O(\log n)$ time and $O(1)$ kinetic heap queries. The $O(n)$ amortized deletions do not only rebuild the data structure, but also pay for advancing the kinetic search over all segments. Bootstrapping with this we get $O(\log n)$ amortized insertions, $O(1)$ amortized queries and amortized $O(\log^2 n)$ deletions. Bootstrapping one more time reduces the amortized deletion cost to $O(\log n)$.

We summarize the above discussion into the following theorem.

Theorem 4.15
There exists a data structure for the fully dynamic kinetic heap problem supporting INSERT *in amortized $O(\log n)$ time,* DELETE *in amortized $O(1)$ time and* KINETIC FIND MIN *in amortized $O(1)$ time. The space usage of the data structure is $O(n)$. The parameter n denotes the size of the stored set before the operation.*

Chapter 5

Lower bounds

The ultimate goal when designing a data structure is to find one that is as fast as possible. This is almost impossible to achieve in the literal sense. It is for example very hard to rule out the possibility that treating some situations as special cases will speed up the algorithm for some inputs. This is where an asymptotic worst case analysis allows us to focus on the big picture. By considering running times (and space usage) only up to a constant factor we abstract away from a lot of details that are hard to control.

In this chapter we consider lower bounds for the data structure problems we are concerned with. For a lower bound statement the model of computation plays a crucial role. An algorithm is not only a program on one machine, but rather an algorithmic idea that can be implemented in a variety of formalisms and on different models of computation. An efficient algorithm in one setting is likely to be an efficient algorithm in a similar setting. Or looking at things differently, we could say that a really good algorithm is efficient for all settings. Unfortunately this kind of quality considerations of an algorithm are not easy to capture formally.

What we can do formally is to consider one specific model of computation. There we can have a proof that our data structure performs asymptotically optimal in the sense that it uses only a constant factor more time than is necessary for a particular family of input values. This statement can hold with the same time bounds for different models of computation. If a stronger model can simulate a weaker model without any slow-down, and we have the lower bound on the stronger and the algorithm on the weaker model, the bounds match in both models. This is the case here: The data structures are formulated as order-k branching pointer programs, whereas the lower bounds are stated for the real-RAM and the algebraic decision tree model.

5.1 Complexity of data structure problems

There is actually one more complication when considering the amortized performance of a data structure. There we want to measure the (overall) running time of the data structure of the sequence of operations depending on the size of the problem (like the maximal number of simultaneously stored elements) and the number of the different operations in the sequence. Alternatively we could define the size by the number of insertions, or consider the size parameter directly before executing the operation. As long as the functions describing the performance are smooth, all this does not make an asymptotic difference (Lemma 4.1, p. 77). This is yet another indication that the asymptotic complexity is a good measure.

In the following we use the formulation with the *size parameter* n where we allow the data structure to store at most n elements at a time. The statement that an insertion (query) takes amortized time $I(n)$ then means that the total time spend in k insertions (queries) is $k \cdot I(n)$. This reflects the situation that we do not really have a bound for the running time of a single operation. What we show is that every general upper bound for all choices of n and k has to respect the lower bounds we give.

Another phenomenon we have to address is that the performance of the different operations are not independent. In a data storage setting we can easily achieve constant time insertions if we allow ourselves linear time to query. The other way around we can sometimes achieve constant time queries if we are willing to create and store a table with all possible answers. Both solutions are not necessarily interesting, but they are optimal in a naive sense. It might be more interesting to say that all data structure that achieve a certain query performance necessarily use a certain amount of time when processing an insertion. We will consider this kind of a trade-off between insertions and queries.

In this chapter we show that in the real-RAM our data structure for the dynamic planar convex hull problem is asymptotically optimal in the following sense: In any implementation on the real-RAM queries have to take time $\Omega(\log n)$. Our data structure achieves $O(\log n)$ queries and is therefore asymptotically optimal for queries. If we restrict our attention to implementations that achieve $O(\log n)$ queries (we can even allow $O(n^{1-\epsilon})$ queries), then we have a lower bound on the insert operation of $\Omega(\log n)$ per operation. Again our data structure matches this lower bound with an asymptotic amortized insertion (and deletion) time of $O(\log n)$. For the kinetic heap we show the same lower bound on the amortized insertion time as for the dynamic planar convex hull problem. This is matched by our data structure.

As intermediate steps we show lower bounds for the membership problem and the predecessor problem that are of interest in their own right.

5.2 Reductions

For the static case of computing the convex hull of a set of points, there is a reduction from sorting. We have the by now classical result that sorting takes $\Omega(n \log n)$ time on the real-RAM. More precisely the approach is to define a decision problem, in the example of sorting ELEMENT DISTINCTNESS. Then one shows a lower bound for this decision problem for algebraic decision trees. This lower bound is also a lower bound for a real-RAM algorithm solving ELEMENT DISTINCTNESS. Finally we *reduce* ELEMENT DISTINCTNESS to sorting in linear time. Specifically we use a (fast) sorting algorithm to solve ELEMENT DISTINCTNESS. This bounds the speed of the sorting algorithm, as time spend in the reduction is negligible (in the asymptotic analysis) compared to the time the overall algorithm has to take.

Our approach follows this by now classical argument. The conceptual difference is that we are interested in a data structure problem, and not with a decision problem or the task of computing a function. The general idea is to use calls to the data structure (much like an oracle) as part of the reduction algorithm. Assume the reduction algorithm executes in linear time if we count data structure operations as taking one unit of time. Then we can conclude that any implementation of the data structure needs to use the time that is required to solve the problem we reduced from.

More concretely we consider data structures for the membership problem, the predecessor problem, and for the dynamic planar convex hull problem. We define the variation DISJOINTSET$_{n,k}$ of ELEMENT DISTINCTNESS. Instead of requiring that there are no equal values in a multiset, we require that the values in one set are different from the values in the other set. We show a lower bound of $\Omega(n \log k)$

for $\textsc{DisjointSet}_{n,k}$. We then present reductions that use one of the data structures we are interested in to solve $\textsc{DisjointSet}_{n,k}$. The reduction itself does only take linear time and can therefore not substantially help in deciding $\textsc{DisjointSet}_{n,k}$. We conclude that the data structure has to perform the computations necessary to decide $\textsc{DisjointSet}_{n,k}$. We also give a lower bound for the kinetic heap. To accommodate the situation that the queries have to advance time, we define the variant $\textsc{DisjointSet}^{+}_{n,k}$, where we assume that one of the sets is already sorted.

The reduction we propose actually only use a very weak version of the data structure, namely what we could call off-line variants of the data structure. In this variant the data structure has access to the complete sequence of operations before it starts the processing. The off-line variant of the problem can only be easier to solve than the real data structure problem. Our reductions are almost not interacting with the data structure, they merely convert the input into a sequence of operations to perform. Then the data structure computes all the results of the operations (especially queries). Finally we use these results to give a correct answer to the original problem.

5.3 Algebraic computation trees

Our lower bound result for the algebraic computation tree model relies on a fundamental theorem by Ben-Or in [BO83, Theorem 3] which reads as follows:

Theorem 5.1 (Ben-Or)
Let $W \subseteq \mathbb{R}^n$ be any set, and let T be a computation tree that solves the membership problem for W. If N is the number of disjoint components of W, and h is the height of T, then

$$2^h 3^{n+h} \geq N.$$

As we want to apply this theorem, we follow the definitions of [BO83] closely. In contrast to the real-RAM and the order-k branching pointer machine, the algebraic computation tree is a non-uniform model of computation, that is we (have to) define a different computation tree for every possible input size, and we do not require the trees for different sizes to be similar in any sense, we do not even require that a tree for an arbitrary size can be computed. As the model describes a computation that is allowed to manipulate real numbers in constant time, it is in some sense a really powerful model. On the other hand it is restricted to perform algebraic operations, much in the same way as the real-RAM. This is actually why this model is useful in our setting, namely because it can simulate the program of a real-RAM without any slow down. We will argue for this fact after having given the definition of the model.

An *algebraic computation tree* over $\mathbb{R}^n$ is a labeled rooted tree T with three different types of vertices:

A simple vertex v has exactly one son and is labeled with an operational instruction of the form

$$f_v := f_{v_1} \circ f_{v_2} \quad \text{or} \quad f_v := f_{v_1} \quad \text{or} \quad f_v := \sqrt{f_{v_1}}$$

where v_1 and v_2 are other simple vertices and ancestors of v in the tree T, or $f_{v_1} \in \{x_1, \ldots, x_n, c\}$. We have $\circ \in \{+, -, \times, /\}$, and $c \in \mathbb{R}$ is a constant.

A branching vertex v has precisely two sons and is labeled with a test instruction of the form

$$f_{v_1} > 0 \quad \text{or} \quad f_{v_1} \geq 0 \quad \text{or} \quad f_{v_1} = 0$$

where v_1 is an ancestor of v, or $f_{v_1} \in \{x_1, \ldots, x_n\}$. The edges to the two sons are labeled to identify which son is corresponding to a positive outcome of the test.

A leaf v is labeled with YES or NO, understood as the result of the computation.

An input $x \in \mathbb{R}^n$ defines a (labeled) path $P(x)$ in the tree T. The path $P(x)$ starts at the root of T. Assume that we already defined $P(x)$ up to vertex v. If v is a simple vertex, the value $f_v \in \mathbb{R}$ is defined according to the operational instruction forming the label of the node v. By the definition of T the values f_v depends upon are already defined as labels of $P(x)$. If the operation is / and the second operand is 0, or if the operation is $\sqrt{\cdot}$ and the operand is negative, $P(x)$ ends and the result of $P(x)$ is undefined. We understand f_v as the label of v. The next vertex of $P(x)$ is the only son of v in T. If v is a branching vertex, the next vertex of $P(x)$ is the son of v that corresponds to the outcome of the test on the appropriate label f_{v_1} of $P(x)$. If v is a leaf vertex, $P(x)$ does not continue and the result of $P(x)$ is the label of the leaf.

We say that an algebraic computation tree T over $\mathbb{R}^n$ solves the membership problem for $W \subseteq \mathbb{R}^n$ if for all $x \in \mathbb{R}^n$ the outcome of $P(x)$ is defined, and YES if and only if $x \in W$. The height of the tree T is the maximal length of a computation path $P(x)$ where x ranges over all of $\mathbb{R}^n$.

Now we have to relate this definition of an algebraic computation tree to our definition of the real-RAM in Section 2.1.1.

Lemma 5.2 (real-RAM simulation)
Let P be a program for the real-RAM solving the membership problem for a set $W_n \subseteq \mathbb{R}^n$ in time $t(n)$.

Then there exists an algebraic computation tree T over $\mathbb{R}^n$ of height $t(n)$ solving the membership problem for W_n.

Proof: We define T according to a symbolic execution of P on all possible inputs $x \in \mathbb{R}^n$. We actually define an annotated version of T. Every node v of T is also annotated with a state S_v of the integer part of the real-RAM (integer cells and registers, program counter). To keep track of the already performed algebraic branching v is annotated with a subset $Q_v \in \mathbb{R}^n$. The set Q_v consists of all input vectors that lead the computation to the node v. The real-cells of the real-RAM are represented by an annotation R_v: R_v has for every real-cell an entry holding a variable x_i for the input, the name f_u for some predecessor u of v in T, or the constant 0. If the cell holds f_u, the annotation also contains a function $g_u\colon \mathbb{R}^n \to \mathbb{R}$. The function corresponding to the name x_i is given by the projection function $(x_1, \ldots, x_i, \ldots, x_n) \mapsto x_i$.

The root r of T is annotated with $Q_r = \mathbb{R}^n$ and S_r is the initial state of the integer part of the real-RAM. R_r has in the first n cells the variables $x_1, \ldots, x_n$ and in the remaining cells the constant 0. For a node v we create children of v depending on the next operation q of the real-RAM when in state S_v. If q is an operation that affects (changes) only the integer part of the real-RAM, like unconditional jumps, integer assignments, or branching on integer values, we make v a simple vertex with son u and the (void) definition $f_v := 1.0 \cdot x_1$. We annotate u by defining $Q_u := Q_v$ and $R_u = R_v$. S_u reflects the state of the integer part of the real-RAM after executing q. If q is a positive (or negative) halt-instruction, we declare v a leaf that is labeled YES (or NO). If q is an assignment of a real cell, say $c_k := a \circ b$ for $\circ \in \{+, -, \times, /, \sqrt{\cdot}\}$ where a and b are either a real-cell (possibly depending on an integer register of the real-RAM) or a constant. By R_v we can identify a variable f_a that corresponds to a and f_b that corresponds to b, and we set $f_u = f_a \circ f_b$. In the annotation R_v we define g_u by $g_u(x) := g_a(x) \circ g_b(x)$. We define $R_u = R_v[c_k := f_u]$, i.e., we take R_v with the cell c_k overwritten with the variable name f_u. We set $Q_u = Q_v$ unless the operation is $/$ or $\sqrt{\cdot}$. In this case we set Q_u to be the subset of Q_v where f_u is not 0 or not negative. We set S_u to the state of the real-RAM after executing q, that is, S_v with the program-counter advanced. If q is a branching operation, say a conditional jump if $c_k > 0$, then v is a branching vertex with children u and w. Let f_a be the function name that is the entry of R_u in cell c_k and g_a the corresponding function. We set the label (test instruction) of v to be $f_a > 0$ with u being the son for a positive outcome of the test. As annotation we set $Q_u := \{x \in Q_v \mid g_v(x) > 0\}$ and $Q_w = Q_v \setminus Q_u$, $R_u := R_v$ and $R_w := R_v$. The state S_u is S_v with the program-counter reflecting that the conditional jump was performed, and S_w is the state S_v with the program counter merely advanced, reflecting that the conditional jump was not performed.

By this procedure we define an algebraic computation tree, that is possibly infinite, even though the program of the real-RAM has bounded running time.

The following claim follows immediately from the above definitions:
Claim: For $x \in \mathbb{R}^n$ the set of all nodes v with $x \in Q_v$ forms a root to leaf path that corresponds to the execution of the real-RAM on input x.

As the height of the algebraic computation tree is defined by the deepest node v with $Q_v \neq \emptyset$, we get that the height of T is bounded by the running time of the real-RAM, $h(T) \leq t(n)$. Note that we say that a program on the real-RAM decides a set $W \subseteq \mathbb{R}^n$ in time $t(n)$ if it halts on all inputs $x \in \mathbb{R}^n$ after at most $t(n)$ steps with the correct answer. In particular the program does not attempt to divide by zero or extract a root of a negative number for any input. □

In particular if we have an algorithm that is order-k branching based, the height of the computation tree reflects the worst-case number of branchings. Hence we focus on the algebraic computation tree for lower bounds.

5.4 Decision problems

Definition 6
The problem $\text{DisjointSet}_{n,k} \subset \mathbb{R}^{n+k}$ *is defined by the rule that for the vector* $z = (x_1, \ldots, x_n, y_1, \ldots, y_k) \in \mathbb{R}^{n+k}$ *we have* $z \notin \text{DisjointSet}_{n,k}$ *if there exists* $i \in \{1, \ldots, n\}$ *and* $j \in \{1, \ldots k\}$ *such that* $x_i = y_j$*, otherwise we have* $z \in \text{DisjointSet}_{n,k}$.

Definition 7
For a vector $z = (x_1, \ldots, x_n, y_1, \ldots, y_k) \in \mathbb{R}^{n+k}$ *we have* $z \in \text{DisjointSet}^+_{n,k} \subset \mathbb{R}^{n+k}$ *if and only if* $y_1 \leq y_2 \leq \cdots \leq y_k$ *and for all* i *and* j *we have* $x_i \neq y_j$.

Lemma 5.3
For $81 < k \leq n$ *the depth* h *of an algebraic computation tree deciding the set* $\text{DisjointSet}_{n,k}$ *or* $\text{DisjointSet}^+_{n,k}$ *is lower bounded by* $h \geq c \cdot n \log k$ *for some* $c > 0$.

Proof: We exhibit a set of k^n input vectors that are in different connected components of $\text{DisjointSet}_{n,k}$.

Let $\mathcal{S} = S_0, \ldots, S_k$ be a named $(k+1)$-partition of $\{1, \ldots, n\}$. We define the vector $z_{\mathcal{S}} = (x_1, \ldots, x_n, 1, 2, \ldots, k)$ by the rule $x_i = j + \frac{1}{2}$ for $i \in S_j$. Then we clearly have that $z_{\mathcal{S}}$ is in $\text{DisjointSet}_{n,k}$ and $\text{DisjointSet}^+_{n,k}$.
Claim: For two different named $(k+1)$-partitions $\mathcal{S} = S_0, \ldots, S_k$ and $\mathcal{R} = R_0, \ldots, R_k$ we have that the vectors $z_{\mathcal{S}}$ and $z_{\mathcal{R}}$ are in different connected components of $\text{DisjointSet}_{n,k}$ and $\text{DisjointSet}^+_{n,k}$.

Let $i \in \{1, \ldots, n\}$ be such that $i \in S_a$ and $i \in R_b$ for $a > b$ (w.l.o.g.), and $\tau : [0, 1] \to \mathbb{R}^{n+k}$ continuous (a path) with $\tau(0) = z_{\mathcal{S}}$ and $\tau(1) = z_{\mathcal{R}}$. Let p :

$\mathbb{R}^{n+k} \to \mathbb{R}$, defined by $(x_1, \ldots, x_n, y_1, y_2, \ldots, y_k) \mapsto x_i - y_a$. Then we have $p(\tau(0)) = (a + \frac{1}{2}) - a = \frac{1}{2} > 0$ and $p(\tau(1)) = (b + \frac{1}{2}) - a < 0$. Hence there exists $t \in [0, 1]$ such that $p(\tau(t)) = 0$ and $\tau(t) \notin \textsc{DisjointSet}_{n,k}$, and $\tau(t) \notin \textsc{DisjointSet}^{+}_{n,k}$. As all connected components of $\textsc{DisjointSet}_{n,k}$ and $\textsc{DisjointSet}^{+}_{n,k}$ are open sets and therefore also path-connected, we conclude that $z_{\mathcal{S}}$ and $z_{\mathcal{R}}$ are in different connected components.

There are $(k+1)^n$ different ways of distributing n elements into $k+1$ sets, that is there are $(k+1)^n$ named $(k+1)$-partitions of $\{1, \ldots, n\}$, hence there are at least $(k+1)^n$ different connected components of $\textsc{DisjointSet}_{n,k}$. The same lower bound holds for the number of different connected components of $\textsc{DisjointSet}^{+}_{n,k}$.

Theorem 5.1 (p. 99) implies $2^h 3^{n+k+h} \geq (k+1)^n$ yielding

$$h \log 2 + (n + k + h) \log 3 \geq n \log(k+1),$$

$$h(1 + \log 3) \geq n \log k - n \log 3 - k \log 3 \geq n(\log k - 2 \log 3)$$

choosing $c = \frac{1}{2}(1 + \log 3)$ and assuming $\log k - 2 \log 3 \geq \frac{\log k}{2}$ we get

$$h \cdot 2c \geq n \frac{\log k}{2} \; .$$

Transforming the condition on k we get $\frac{\log k}{2} \geq 2 \log 3$, $k \geq 2^{4 \log 3} = 3^4 = 81$. □

5.5 Data structures with general queries

The running time functions in the following theorems are assumed to be non-decreasing with the size of the data structure. As these functions are used as upper bounds on running times, this is no loss of generality. Alternatively we could define $I(n)$ by the fact that $n \cdot I(n)$ is the time that is necessary to insert n objects into an initially empty data structure.

Theorem 5.4
Let $\mathcal{A}$ be a data structure implementing the Semidynamic Membership *problem on the real-RAM. Assume $\mathcal{A}$ supports* Element *queries in amortized $q(n)$ time, and* Insert *in amortized $I(n)$ time for size parameter n. Assume that q and I are smooth functions. Then we have*

$$I(n) = \Omega\left(\log \frac{n}{q(n)}\right) \; ,$$

and for any $I(n)$ we have

$$q(n) = \Omega(\log n) \; .$$

Proof: For the first bound we describe a reduction from $\textsc{DisjointSet}_{n,k}$. We choose the parameter $k = \lfloor n/q(n) \rfloor$. Let the vector $z = (x_1, \ldots, x_n, y_1, \ldots, y_k) \in \mathbb{R}^{n+k}$ be an input to $\textsc{DisjointSet}_{n,k}$. Then we first insert $x_1, \ldots, x_n$ into the data structure and then query for $y_1, \ldots, y_k$. If we get a negative answer for all queries $\textsc{Element}(y_i)$, we conclude $z \in \textsc{DisjointSet}_{n,k}$. Assuming that the Semidynamic Membership data structure is correct, we give the right answer to $\textsc{DisjointSet}_{n,k}$. The time that the resulting algorithm has to take is by Lemma 5.3 (p. 101)

$$I(n) \cdot n + q(n) \cdot k \geq c \cdot n \cdot \log k \; .$$

Using our choice of k we get

$$I(n) \cdot n + n \geq c \cdot n \cdot \log(\lfloor n/q(n) \rfloor).$$

Dividing by n and rearranging terms yields

$$I(n) \geq c \cdot \log(\lfloor n/q(n) \rfloor) - 1 \;.$$

For the second lower bound we use again a reduction from DISJOINTSET$_{n,k}$, but this time with the parameters n and k interchanged. We choose the parameter $n = k \cdot I(k)$. Now we observe that we can solve DISJOINTSET$_{n,k}$ also with k INPUT operations and n QUERY operations. Now by Lemma 5.3 (p. 101) we get for sufficiently large k and some constant c

$$k \cdot I(k) + n \cdot q(k) \geq c \cdot n \cdot \log k \;.$$

Using $k \cdot I(k) = n$ and dividing by n we get

$$q(k) = \Omega(\log k) \;.$$

□

Note that for $q(n) = O(n^{1-\varepsilon})$, Theorem 5.4 implies $I(n) = \Omega(\log n)$. Another example is that $I(n) = O(\log\log n)$ yields $q(n) = \Omega(n/(\log n)^{O(1)})$.

Corollary 5.5
Let $\mathcal{A}$ be a data structure implementing the SEMIDYNAMIC PREDECESSOR PROBLEM *problem on the real-RAM. Assume $\mathcal{A}$ supports* PREDECESSOR *queries in amortized $q(n)$ time, and* INSERT *in amortized $I(n)$ time for size parameter n. Assume that q and I are smooth functions. Then we have*

$$q(n) = \Omega(\log n) \qquad \textit{and} \qquad I(n) = \Omega\Big(\log \frac{n}{q(n)}\Big) \;.$$

Theorem 5.6
Let $\mathcal{A}$ be a data structure implementing the SEMIDYNAMIC INSERTION-ONLY CONVEX HULL *problem on the real-RAM. Assume $\mathcal{A}$ supports extreme point queries in amortized $q(n)$ time, and* INSERT *in amortized $I(n)$ time for size parameter n. Assume that q and I are smooth functions. Then we have*

$$q(n) = \Omega(\log n) \qquad \textit{and} \qquad I(n) = \Omega\Big(\log \frac{n}{q(n)}\Big) \;.$$

Proof: Reduction from SEMIDYNAMIC MEMBERSHIP. For an insert(a) operation we insert $(a, -a^2/2)$. For query(b) we ask the extreme point query for slope $-b$. If and only if the query returns with the point $(b, -b^2/2)$, we return the answer $b \in S$.

Define the line l by $y = -bx + b^2/2$. Then for all points $p = (a, -a^2/2)$ we have that p is not above l and p is on l if and only if $a = b$. We calculate the vertical position (signed distance) of p compared to l by the formula $p_y - l(p_x) = ba - b^2/2 - a^2/2 = -(a-b)^2/2$. This shows the correctness of the reduction. □

5.6 Kinetic heaps

The situation for kinetic heaps is somewhat different, we have a data structure that performs queries in amortized $O(1)$ time. This data structure performs insertions in amortized $O(\log n)$ time. In this section we show that this data structure is optimal in the sense that every data structure needs to use $\Omega(\log n)$ amortized time per insertion to solve the problem. The proof is almost the same as for the general situation of Theorem 5.6, only that the specific type of query does not immediately allow a reduction from the semidynamic membership problem. We could have unified the situation by considering a (somewhat unnatural) kinetic membership problem as intermediate.

Theorem 5.7
Let $\mathcal{A}$ be a data structure that implements a kinetic heap. For size parameter n assume that the amortized running time of the Insert *operation of $\mathcal{A}$ be bounded by $I(n)$ and the amortized running time for the* kinetic-find-min *query be bounded by $q(n)$. Then we have*

$$I(n) = \Omega\left(\log \frac{n}{q(n)}\right) .$$

Proof: Reduction from $\textsc{DisjointSet}^+_{n,k}$. We choose the parameter $k = \lfloor n/q(n) \rfloor$. Let the vector $z = (a_1, \ldots, a_n, b_1, \ldots, b_k) \in \mathbb{R}^{n+k}$ be an input to $\textsc{DisjointSet}^+_{n,k}$. We check in linear time whether we have $b_1 \leq b_2 \leq \cdots \leq b_k$. If this is not the case, we reject.

We insert the lines $l_{a_i} := (y = -a_i \cdot x + a_i^2/2)$ into the kinetic heap $\mathcal{A}$. Then we perform kinetic-find-min(b_i) queries (in the natural order). If for one of the queries kinetic-find-min(b_i) the answer is the line $l_{b_i} := (y = -b_i \cdot x + b_i^2/2)$, we reject. Otherwise we accept.

We reject precisely if we have $a_j = b_i$ for some i and j: We calculate the values of the find-min query at time b_i and compare it with the value if line l_{b_i} is member of the kinetic heap, $l_{a_j}(b_i) - l_{b_i}(b_i) = -a_j \cdot b_i + a_j^2/2 + b_i \cdot b_i - b_i^2/2 = (a_j - b_i)^2/2$. Hence we correctly solve the $\textsc{DisjointSet}^+_{n,k}$ problem.

The reduction takes linear time. The time that the combined algorithm has to use is by Lemma 5.3 (p. 101)

$$(I(n) + d) \cdot n + q(n) \cdot k \geq c \cdot n \cdot \log k ,$$

for some constants c and d. Using our choice of k we get

$$(I(n) + d) \cdot n + n \geq c \cdot n \cdot \log(\lfloor n/q(n) \rfloor) .$$

Dividing by n and rearranging terms yields

$$I(n) \geq c \cdot \log(\lfloor n/q(n) \rfloor) - 1 - d .$$

□

5.7 Trade-off

The above lower bounds apply for all kinds of functions $q(n)$ and $I(n)$. The standard data structures for membership queries on the real-RAM are balanced search trees. This establishes a matching upper bound only for the cases where insertions are required to take $\Omega(\log n)$ time, namely for $q(n) = O(n^{1-\varepsilon})$. We have the same situation for the dynamic planar convex hull problem. This raises the question, whether there are data structures that match the lower bound for other combinations of insertion and query times as well.

There is one simple idea for a trade-off between insertion times and query times: we simply maintain several (small) search structures and insert into one of them. In return the query operation has to query all the search structures. The situation we describe is valid for all decomposable search problems (as defined in Section 4.1.1) that can be solved with $O(\log n)$ amortized insertions and queries. We will describe the predecessor problem and use balanced search trees as the underlying data structure. We focus on the insertion only case. If we want to accommodate deletions we have to perform global rebuilding following a doubling technique. This does not change the (spirit of) the result, it only makes it more complicated to describe. The argument works for worst-case and amortized complexities.

We will choose a smooth parameter function $s(n)$ that tells the data structure how many elements might be stored in one search tree. We assume that $s(n)$ is easy to evaluate (one evaluation in $O(n)$ time suffices) and non-decreasing. The data structure keeps two lists of trees, one with the trees that contain precisely $s(n)$ elements and the other with the trees containing less elements. For an Insert(e) operation we insert e into one of the search trees that contains less than $s(n)$ elements. If no such tree exists, we create a new one. When $s(n)$ increases, we join the two lists (all trees are now smaller than $s(n)$) and create an empty list of full search trees. For a query operation we query all the search trees and combine the result.

The (amortized) insertion time is $I(n) = O(\log s(n))$, the query time is $q(n) = O(\frac{n}{s(n)} \log s(n))$. We consider the term

$$\log \frac{n}{q(n)} = \Omega\left(\log \frac{s(n)}{\log s(n)}\right) = \Omega\big(\log s(n) - \log\log s(n)\big) = \Omega(\log s(n)) \; .$$

This means that we achieve according to Theorem 5.4 (p. 102) optimal amortized insertions times.

If we are interested in a data structure for the membership, predecessor and convex hull problem that allows queries in $q(n)$ time for a smooth, easy to compute function q, then this technique allows us to have a data structure with asymptotically optimal insertion times.

Bibliography

[AdB$^+$98] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf, *Constructing levels in arrangements and higher order Voronoi diagrams*, SIAM J. Comput. 27 (1998), no. 3, 654–667 (electronic).

[And79] A. M. Andrew, *Another efficient algorithm for convex hulls in two dimensions*, Information Processing Letters 9 (1979), no. 5, 216–219.

[BAG01] A. M. Ben-Amram and Z. Galil, *Lower bounds for dynamic data structures on algebraic RAMs*, Algorithmica 32 (2001), no. 3, 364–395.

[BC$^+$98] L. Blum, F. Cucker, M. Shub, and S. Smale, *Complexity and real computation*, Springer-Verlag, New York, 1998, With a foreword by Richard M. Karp.

[BGR96] J. Basch, L. Guibas, and G. D. Ramkumar, *Reporting red-blue intersections between two sets of connected line segments*, Algorithms—ESA '96 (Barcelona) (Berlin), Springer, Berlin, 1996, pp. 302–319.

[BJ00] G. S. Brodal and R. Jacob, *Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log\log n)$ update time*, Proc. 7th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, vol. 1851, Springer, 2000, pp. 57–70.

[BJ02] G. S. Brodal and R. Jacob, *Dynamic planar convex hull*, Proc. 43rd Annual Symposium on Foundations of Computer Science, 2002, pp. 617–626.

[BO83] M. Ben-Or, *Lower bounds for algebraic computation trees*, Proc. 15th Annual ACM Symposium on Theory of Computing, 80 – 86, 1983.

[BS80] J. L. Bentley and J. B. Saxe, *Decomposable searching problems. I. Static-to-dynamic transformation*, J. Algorithms 1 (1980), no. 4, 301–358.

[BSS90] L. Blum, M. Shub, and S. Smale, *On a theory of computation over the real numbers: NP completeness, recursive functions and universal machines [Bull. Amer. Math. Soc. (N.S.)* **21** *(1989), no. 1, 1–46*, Workshop on Dynamical Systems (Trieste, 1988) (Harlow), Longman Sci. Tech., Harlow, 1990, pp. 23–52.

[CE87] B. Chazelle and H. Edelsbrunner, *An improved algorithm for constructing kth-order Voronoĭ diagrams*, IEEE Trans. Comput. 36 (1987), no. 11, 1349–1354.

[Cha96] T. M. Chan, *Optimal output-sensitive convex hull algorithms in two and three dimensions*, Discrete Comput. Geom. 16 (1996), no. 4, 361–368, Eleventh Annual Symposium on Computational Geometry (Vancouver, BC, 1995).

[Cha99a] T. M. Chan, *Dynamic planar convex hull operations in near-logarithmic amortized time*, IEEE Symposium on Foundations of Computer Science, 1999, pp. 92–99.

[Cha99b] T. M. Chan, *Remarks on k-level algorithms in the plane*, 1999, Manuscript.

[Cha01] T. M. Chan, *Dynamic planar convex hull operations in near-logarithmic amortized time*, Journal of the ACM 48 (2001), no. 1, 1–12.

[CT92] Y.-J. Chiang and R. Tamassia, *Dynamic algorithms in computational geometry*, Proceedings of the IEEE 80 (1992), no. 9, 1412–1434.

[dBvK$^+$97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: Algorithms and applications*, Springer-Verlag, Berlin, 1997.

[Ede80] Edelsbrunner, *Dynamic data structure for orthogonal intersection queries*, Tech. Report F59, Inst. Informationsverarb. Tech. Univ. Graz, Graz, Austria, 1980.

[EW86] H. Edelsbrunner and E. Welzl, *Constructing belts in two-dimensional arrangements with applications*, SIAM J. Comput. 15 (1986), no. 1, 271–284.

[Fre75] M. L. Fredman, *On computing the length of longest increasing subsequences*, Discrete Math. 11 (1975), 29–35.

[FS89] M. L. Fredman and M. E. Saks, *On the cell probe complexity of dynamic data structures*, Proc. 21$^{\text{st}}$ ACM STOC, (1989), 1989.

[GKP95] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete mathematics*, 2$^{\text{nd}}$ ed., Addison-Wesley, Reading, USA, 1995.

[GO97] J. E. Goodman and J. O'Rourke (eds.), *Handbook of discrete and computational geometry*, CRC Press, Boca Raton, FL, 1997.

[Gra72] R. L. Graham, *An efficient algorithm for determining the convex hull of a finite planar set*, Information Processing Letters 1 (1972), no. 4, 132–133.

[HM$^+$86] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan, *Sorting Jordan sequences in linear time using level-linked search trees*, Information and Control (now Information and Computation) 68 (1986), no. 1-3, 170–184.

[HPS01] S. Har-Peled and M. Sharir, *On-line point location in planar arrangements and its applications*, Proc. 12th ACM-SIAM Sympos. Discrete Algorithms, 2001, pp. 57–66.

[HS92] J. Hershberger and S. Suri, *Applications of a semi-dynamic convex hull algorithm*, BIT 32 (1992), no. 2, 249–267.

[HS96] J. Hershberger and S. Suri, *Off-line maintenance of planar configurations*, J. Algorithms 21 (1996), no. 3, 453–475.

[HTK01] K. H., R. Tarjan, and T. K., *Faster kinetic heaps and their use in broadcast scheduling*, Proc. 12th ACM-SIAM Symposium on Discrete Algorithms, 2001, pp. 836–844.

[Jar73] R. A. Jarvis, *On the identification of the convex hull of a finite set of points in the plane*, Information Processing Letters 2 (1973), no. 1, 18–21.

[KS86] D. G. Kirkpatrick and R. Seidel, *The ultimate planar convex hull algorithm?*, SIAM J. Comput. 15 (1986), no. 1, 287–299.

[Mat95] J. Matoušek, *On geometric optimization with few violated constraints*, Discrete Comput. Geom. 14 (1995), no. 4, 365–384, ACM Symposium on Computational Geometry (Stony Brook, NY, 1994).

[McC80] McCreight, *Efficient algorithms for enumerating intersecting intervals and rectangles*, Tech. Report CSL-80-9, Xerox Park Palo Alto Res. Center, Palo Alto, CA, 1980.

[Meh84a] K. Mehlhorn, *Data structures and algorithms 1: Sorting and searching*, Springer-Verlag, Berlin, 1984.

[Meh84b] K. Mehlhorn, *Data structures and algorithms 3: Multidimensional searching and computational geometry*, Springer-Verlag, Berlin, 1984.

[Ove83] M. H. Overmars, *The design of dynamic data structures*, LNCS, vol. 156, Springer-Verlag, Berlin, 1983.

[OvL81] M. H. Overmars and J. van Leeuwen, *Maintenance of configurations in the plane*, J. Comput. System Sci. 23 (1981), no. 2, 166–204.

[PH77] F. P. Preparata and S. J. Hong, *Convex hulls of finite sets of points in two and three dimensions*, Comm. ACM 20 (1977), no. 2, 87–93.

[Pre79] F. P. Preparata, *An optimal real-time algorithm for planar convex hulls*, Comm. ACM 22 (1979), no. 7, 402–405.

[PS85] F. P. Preparata and M. I. Shamos, *Computational geometry, an introduction*, Springer-Verlag, New York, 1985.

[Sha78] M. I. Shamos, *Computational geometry*, Ph.D. thesis, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1978.

[SU00] J.-R. Sack and J. Urrutia (eds.), *Handbook of computational geometry*, North-Holland, Amsterdam, 2000.

[Tar85] R. E. Tarjan, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods 6 (1985), no. 2, 306–318.

[vEB80] P. van Emde Boas, *On the $\omega(n \log n)$ lower bound for convex hull and maximal vector determination*, Inform. Process. Lett. 10 (1980), no. 3, 132–136.

[Wei00] K. Weihrauch, *Computable analysis. an introduction*, Springer-Verlag, Berlin, 2000.

Index