Maintenance of the Convex Hull of a Dynamic Set

Jose Alberto Cisneros

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2007

Abstract

This report describes the development and implementation of the Overmars and Van Leeuwen algorithm to maintain the convex hull of a dynamic set. The details of the data structures mentioned by Overmars and Van Leeuwen were worked out, before starting on the implementation. A comparison was made between the Overmars and Van Leeuwen approach and a naive approach to maintain the convex hull of a dynamic set.

Acknowledgements

I would like to thank my supervisor, Mary Cryan, for all the help, advice, and support she offered me throughout the life-cycle of this project.

I also want to thank my parents, who have always been there for me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jose Alberto Cisneros*)

Table of Contents

1 Introduction **1**

1.1 The Convex Hull 3

1.1.1 Off-line Algorithms 3

1.1.2 On-line Algorithms 4

1.2 Dynamic Set of Points 4

1.3 Overview 5

2 The Overmars and Van Leeuwen Algorithm **7**

2.1 Prelude 7

2.2 Organising the Points of the Convex Hull 9

2.2.1 Storing the Points of the Convex Hull 9

2.3 Construction of the lc-hull 10

2.3.1 Building the bridge 12

2.4 Structuring the lc-hulls 13

2.5 Maintaining the Convex Hull 17

2.6 Running Time Analysis 17

3 Data Structures **19**

3.1 Concatenable Queues 19

3.1.1 2-3 Trees 19

3.1.2 Insertion of a Point 20

3.1.3 Concatenating Two Queues 22

3.1.4 Splitting a Queue 22

3.2 Top-Level Tree 23

3.2.1 Node's Structure 24

3.2.2 Insertion 25

3.2.3 Deletion 27

3.2.4 Keeping the Tree Balanced 29
3.3 Managing the *lc*-hull 33
3.4 Building the lc-hulls during the DOWN routine 35
3.5 Building the lc-hulls during the UP routine 35
3.5.1 Determining the bridge between two lc-hulls 36
3.5.2 Updating the lc-hull 39

4 Implementation of the Algorithm 41
4.1 Preliminary Design Decisions 41
4.2 Concatenable Queue 42
4.3 Top-Level Tree 42
4.4 DOWN Procedure 43
4.5 UP Procedure 44
4.6 User Options 46
4.6.1 Displaying the Top-Level Tree 47
4.7 Debugging the code 48

5 Testing the Implementation 51
5.1 Testing the Data Structures 51
5.2 A non-random test 52
5.3 Tests with random set of points 54
5.3.1 Comparing Running Times 61

6 Conclusions 65

Bibliography 67

List of Figures

2.1 Deleting an extreme point of a convex hull. 8
2.2 The *rc* and *lc*-hull of a set of points. 10
2.3 Constructing the *lc*-hull of a set of points. 11
2.4 Cases 1-8 for determining the bridge 14
2.5 Cases 9 and 10 for determining the bridge 15
2.6 Storing the *lc*-hulls in the height-balanced tree. 16

3.1 Insertion of a new point whose y-coordinate is bigger than the one of the point stored in the right leaf. 27
3.2 Clockwise-Anticlockwise Rotation. 29
3.3 Anticlockwise Rotation. 30
3.4 Anticlockwise-Clockwise Rotation. 30
3.5 Clockwise Rotation. 31

4.1 Display of a top-level tree. 48

5.1 Top-level tree storing $S1$ and its *lc* and *rc*-hull. 53
5.2 The Convex Hull of $S1$. 54
5.3 The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 4.5 and 13. 55
5.4 The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 8 and 17. 56
5.5 The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 7 and 12. 57
5.6 The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 11 and 17 after being updated. 59
5.7 Top-level tree after deleting the point $(5, 8)$ from $S1$. 62

5.8 The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 4.5 and 13 after being updated. 63
5.9 The Convex Hull of $S1'$. 63
5.10 The Convex Hull of $S1'$ and the Convex Hull after deleting p' from $S1'$. 64

Chapter 1

Introduction

Computational geometry deals with problems that take place in n-dimensional space, that is, the space of points represented as n-tuples, $(x_1, x_2, \ldots, x_n)$ where all x_i are real numbers. These problems concern sets of points, lines, and polygons. In fact, we can consider any objects in n-dimensional space that can be specified as a finite string of parameters.

In this project, we dealt with a particular problem of computational geometry: computing the convex hull of a set in 2D. We are interested in this problem because it is central in practical applications in areas such as pattern recognition, image processing, and stock cutting and allocation [4]. Also, the convex hull problem arises in the context of other computational geometric problems (which may not seem to be related), like the problem of the maximal element of a set or the diameter and width of a planar set of points.

The purpose of this project was to study and implement the algorithm presented by Overmars and Van Leeuwen [2]. The characteristic that makes this algorithm different from other algorithms is that it efficiently maintains the *convex hull* of a *dynamic* set. The concepts of convex hull and dynamic set will be explained in the sections below.

We now briefly describe the steps of this project. They are presented in the order that they were undertaken.

Understanding the Overmars and Van Leeuwen Algorithm This step consisted of studying the algorithm presented in [2]. This paper is a high-level technical paper in Algorithms and Data Structures. Time was needed to fully understand the data structures and the algorithm presented in Overmars and

Van Leeuwen's paper. Since it is a high-level paper, there were details that had to be worked out in order to implement the data structures and the algorithm.

Implementation First, we quickly had to decide on a language which we could use to construct and easily manipulate the data structures needed to implement the algorithm. We will see that Overmars and Van Leeuwen's algorithm requires the implementation of more than one type of tree. We chose C++ because it is an object-oriented language that supports where we could the use of pointers, which allow us to work easily with trees. Any other imperative language would have been ok, but we one of the reasons for choosing C++ was because I am familiar with it. The first and essential step before implementing the algorithm was implementing the data structures that are used in the algorithm. Before continuing to implement the algorithm, we tested individually the implementation of the data structures. The next step was implementing the procedures of the Overmars and Van Leeuwen algorithm to update the convex hull after the insertion or deletion of a point.

Checking the Correctness of the Implementation To test our implementation, we first used sets of points whose convex hull was known. The second step was to check our implementation involved using random sets of points and we benchmarked the results against the results obtained from an implementation of Graham's Scan. So each time we performed an insertion or deletion of a point from our set of points, we recomputed the convex hull with Graham's Scan implementation and compared this against the results obtained from our implementation of the Overmars and Van Leeuwen algorithm.

Write-up After most of the work outlined before was completed, the final part of the project was to write the report. The report consists of the description of the Overmars and Van Leeuwen algorithm and of the work we did to complete a correct implementation of it. The overview of the report is described in section 1.3.

1.1 The Convex Hull

A *convex set* in $\mathbb{R}^2$ is a region such that for every pair of points in the set, the entire line segment connecting those points is inside the set. Formally, it is defined as follows:

Definition 1.1 Let $C \subseteq \mathbb{R}^2$. C is said to be *convex* if for all p, $q \in C$ and $t \in \mathbb{R}$ such that $0 \leq t \leq 1$, the point $(1-t)p + tq$ is in C.

Definition 1.2 Let $S \subseteq \mathbb{R}^2$. The *convex hull* of S is the smallest convex set that contains S.

Intuitively, imagine surrounding the set S by a stretch rubber band. When the rubber band is released, it will take the shape of the contour of the convex hull of S. When S is a finite set of points, the shape of the convex hull is a polygon. It is well known that a polygon can be described by the list of the points that are its vertices.

The problem of the computing the convex hull has been restricted to finite sets of points. Henceforth, we will only talk about finite sets of points.

The problem of computing the convex hull of a set $S \subset \mathbb{R}^2$ has been studied extensively. Van Emde Boas [7] showed that solving this problem has an $\Omega(n \log n)$ worst-case running time, for a set of points of size n.

Different algorithms to compute the convex hull of S have been proposed. These algorithms can be divided in two major groups depending on how they process S, which is the input: *off-line* and *on-line* algorithms.

1.1.1 Off-line Algorithms

An algorithm that works off-line is one that requires that all inputs are read and stored before any processing can begin. Graham's Scan [4] is an example of an off-line algorithm to compute the convex hull of a set of n points.

Graham's Scan first orders all points of the set by their polar angle with respect to the point p_0 with the minimum y-coordinate. If there is more than one point with the minimum y-coordinate, it chooses the leftmost one to be p_0. After the points have been ordered, the algorithm starts at p_0 and walks around the points in order of increasing polar angle. At the current point, the algorithm checks if it has to turn right or left to get to the next point. If it has to turn left, the current point is added to the convex hull. If it has to turn left, it discards the current point.

The output of Grahams's Scan is an ordered set containing the vertices of the convex hull. Graham's Scan's running time matches the lower bound given by Van Emde Boas.

1.1.2 On-line Algorithms

An algorithm that works on-line is one that, without having the entire input from the start, processes the input piece by piece. A general feature of this type of algorithm is that a new item is input on request as soon as the update relative to the previously input item has been completed. Preparata [3] proposed an on-line algorithm to compute the convex hull. Preparata's algorithm finds the convex hull of a set of points in $O(n \log n)$ and performs the update of the convex hull after the insertion of a point in $\Theta(\log n)$ time.

Briefly, Preparata's algorithm [3] is as follows. When a new point p is presented, first it is determined whether it lies inside or outside the existing convex hull. When p lies inside, no update is needed. When p lies outside the convex hull, the tangents $\overline{rp}$ and $\overline{qp}$ to the current convex hull must be determined. To form the new convex hull, p must be inserted in place of the points between r and q.

1.2 Dynamic Set of Points

A dynamic set of points in the plane S is a finite set of points of $\mathbb{R}^2$, where throughout the course of time new points arrive and old points are deleted. A naive and inefficient approach to maintain the convex hull of S is to recompute it each time the set changes, perhaps using Graham's Scan or a similar algorithm.

In the setting of a dynamic set, an efficient algorithm should not need to recompute the convex hull from scratch. Each time the set is modified, the algorithm should only perform a little "extra work" to get the new convex hull. Note that a *dynamic* algorithm is more general than an on-line algorithm because it mus process deletions efficiently as well as insertions.

1.3 Overview

This dissertation continues in Chapter Two with a description of the Overmars and Van Leeuwen algorithm. This description includes the way the data structures are used to maintain the convex hull, how these structures are organised, and the analysis of the running time of the algorithm. Chapter Three presents the data structures that we use in our implementation and the gaps we filled from [2] in our design of these data structures. These are presented with the assumption that the reader is familiar with the concept of balanced tree schemes. In Chapter Four, we describe in detail our implementation of the Overmars and Van Leeuwen algorithm. The tests of the correctness of our implementation are outlined in Chapter Five. Finally, in Chapter Six we present the conclusions of the project.

My source code (and executable) is available in my student account, on DICE in the following directory:∼/s0671978/PROJECT/.

Chapter 2

The Overmars and Van Leeuwen Algorithm

2.1 Prelude

Before Overmars and Van Leeuwen proposed their algorithm [2], there were no known algorithms that could efficiently maintain the convex hull of a dynamic set of points. Only on-line algorithms, like the one of Preparata (described in section 1.1.2 of Chapter 1), were able to efficiently maintain the convex hull of a dynamic set where only insertions were permitted. Preparata's algorithm [3] can find the convex hull of a set of n points in $\Theta(n \log n)$ time and perform the update in $\Theta(\log n)$ time.

Preparata's algorithm cannot deal with deletions because it goes by the principle that points found inside the current convex hull will not be needed in the future and are discarded. Other algorithms that follow this principle are: Graham's Scan, Jarvis's March, and Quickhull [4]. Discarding a point is a problem because when a point is deleted from the convex hull, old points of the interior may become part of the new convex hull (see Figure 2.1). Also, the number of points added to the convex hull can be rather large. Hence, it is not enough just to store internal points for future use, we also need to store them in a manner which allows efficient updates even when a large number of points must be added to the convex hull. Overmars and Van Leeuwen [2] show how to maintain the convex hull of a set of n points in $O(\log^2 n)$ time after inserting a point into the set or deleting a point from it.

To explain Overmars and Van Leeuwen's idea, it helps to follow their paper

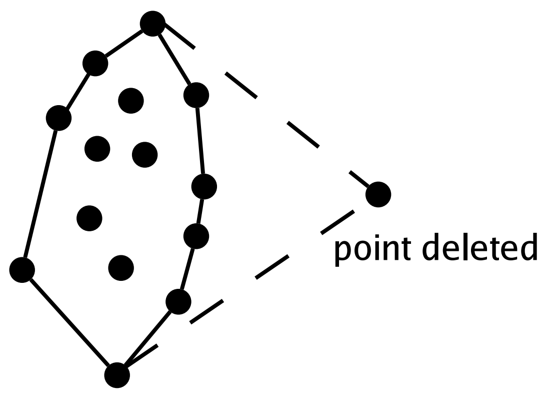


Figure 2.1: Deleting an extreme point of a convex hull.

and to first represent the convex hull as the union of two structures: the *lc*-hull and the *rc*-hull. The *lc*-hull is a convex arc that consists of the left faces of the convex hull. On the other hand, the *rc*-hull is a convex arc that consists of the right faces of the convex hull. Instead of updating the complete convex hull, Overmars and Van Leeuwen update the *lc* and *rc*-hulls by means of a divide-and-conquer framework. We focus at the *lc*-hull now. This divide-and-conquer framework consists of constructing the *lc*-hull of a set of points by joining (in the way that will be described in section 2.3) the *lc*-hulls of the two halves of the set. The points of the set are divided according to their y-coordinate. The base case of this divide-and-conquer approach is when the set has one point. The *lc*-hull of a set of points containing only one point consists only of that point.

Observation 2.1 Since the set of points is divided each time in half (approximately), this divide-and-conquer approach would terminate after $O(\log n)$ steps.

Despite the fact that Overmars and Van Leeuwen use this divide-and-conquer framework, it is not a recursive algorithm. It uses the divide-and-conquer framework to update the *lc*-hull (and *rc*-hull).

The recursive structure of the *lc*-hull (and *rc*-hull) allows the Overmars and Van Leeuwen algorithm to keep track of all points. All points can be considered to update the *lc* and *rc*-hull of the set, and hence the convex hull. This allows the Overmars and Van Leeuwen algorithm to overcome the problem Preparata's algorithm (and other algorithms that followed the same principle) had with deletions.

After a point is deleted or added to the set, Overmars and Van Leeuwen's algorithm updates the *lc*-hull (and *rc*-hull) of the set by first updating the *lc*-

hull of the base case at the point where the insertion or deletion occurred. The place where the insertion or deletion occurred was found by going down a special height-balanced binary search tree where the points of the set are stored. Then the algorithm goes up the recursion tree described before until it updates the lc-hull (and rc-hull) of the whole set.

Observation 2.2 Overmars and Van Leeuwen assume throughout their paper that the points of the set have different y-coordinate.

In the following sections, we will describe in detail how the lc-hull (and rc hull) are constructed and explain why the data structures that store them allow the Overmars and Van Leeuwen algorithm to achieve doing an update time of $O(\log^2 n)$ time for insertions and deletions. Also, we will explain how the set of points and the lc and rc hulls of the "halves" of the set are organised.

2.2 Organising the Points of the Convex Hull

To store the convex hull, Overmars and Van Leeuwen take in consideration that the convex hull can be seen as the "trivial" union of two convex arcs: the lc-hull and the rc-hull (lc for left convex and rc for right convex). Instead of maintaining the whole convex hull, they maintain these two arcs and then join them to form the convex hull. The lc-hull is considered separate from the rc-hull in order to enable the recursive decomposition described in section 2.1 for each of the half-hulls.

The lc-hull consists of the left faces of the convex hull. Likewise, the rc-hull consists of the right faces of the convex hull. The lc and rc-hull of a set of points are illustrated in Figure 2.2. In the following sections, we will focus on the lc-hull of a set, as its rc-hull is treated in a symmetric way.

2.2.1 Storing the Points of the Convex Hull

As mentioned in section 2.1 of this chapter, we can think of lc-hull as being constructed by splitting the lc-hulls of the halves of the set, and then joining one of the resulting parts of each half. The rc-hull is constructed in the same way. To date, we have said very little about how we will join or split these convex arcs. However, we will see that in order to achieve the $O(\log^2 n)$ running time, the following operations must be implemented efficiently (to take $O(\log N)$ time,

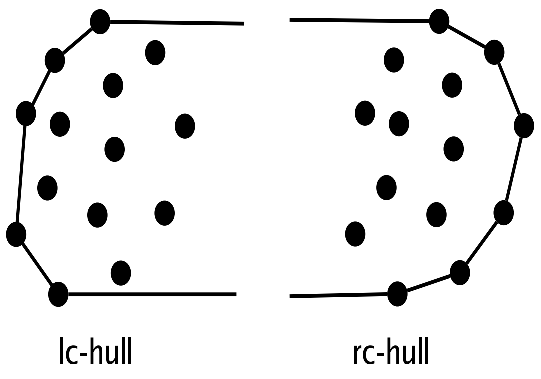


Figure 2.2: The *rc* and *lc*-hull of a set of points.

where N is the number of points in the current convex hull) for the lc-hull and rc-hull:

- Search for a point.
- Split into two sub lc-hulls.
- Join two lc-hulls.
- Insert a point.

A concatenable queue can perform these operations in just $O(\log N)$ time, where N is the number of elements stored in the queue [1]. These are the structures that Overmars and Van Leeuwen chose as a data structure for the lc-hull. The points of the lc-hull are stored in a concatenable queue sorted by their y-coordinate. There are different ways to implement a concatenable queue. In section 3.1 of Chapter 3, we describe how we implemented a concatenable queue.

2.3 Construction of the lc-hull

The lc-hull is a structure that is decomposable in the following sense. Split the set of points into two halves, A and C, by a horizontal line parallel to the y axis. The lc-hull of the complete set is then composed of a section of the lc-hull of A, a section of the lc-hull of C, and a *bridge* b.

Definition 2.1 Let A be the upper half of a set of points and C the lower half. Let s and t be the points where the common tangent to the lc-hulls of A and C intersects the lc-hull of A and C respectively. The *bridge* is the line segment $\overline{st}$.

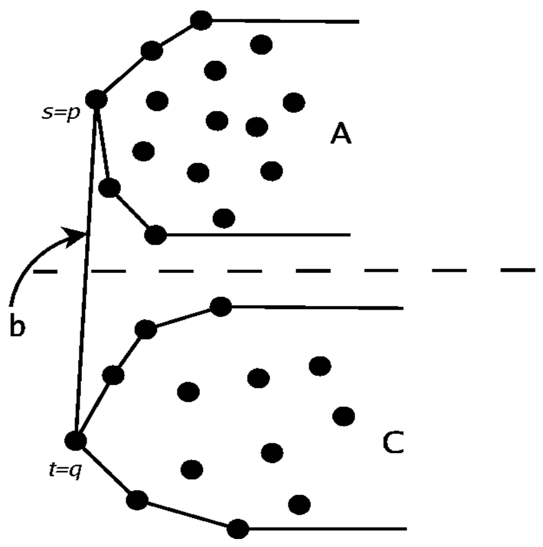


Figure 2.3: Constructing the *lc*-hull of a set of points.

Let $\{p_1, p_2, \ldots, p_m\}$ be the set of points of the lc-hull of a set of points, ordered by y-coordinate, and let p_1 be the point with smallest y-coordinate.

Definition 2.2 The *upper part* of a lc-hull w.r.t. a point p_i of the lc-hull is the set of points $\{p_i, p_{i+1}, \ldots, p_m\}$.

Definition 2.3 The *lower part* of a lc-hull w.r.t. a point p_i of the lc-hull is the set of points $\{p_1, p_2, \ldots, p_i\}$.

Let s, in the lc-hull of A, and t, in the lc-hull of C, be the points where the bridge starts and ends respectively. To build the lc-hull of the complete set, the lc-hulls of A and C are split at s and t, respectively. Then the upper part of the lc-hull of A w.r.t. s is joined with the lower part of the lc-hull of C w.r.t. t. This is illustrated in Figure 2.3.

Overmars and Van Leeuwen make the following observation. The lc-hulls of A and C can also be built in the way previously described. This gives a recursive decomposition of the lc-hull of the complete set of points. However, note that Overmars and Van Leeuwen's algorithm is not a divide and conquer algorithm, it is a dynamic algorithm (in the sense described in section 1.2 of Chapter 1). So, Overmars and Van Leeuwen deal with a recursive data structure and not with a recursive algorithm.

2.3.1 Building the bridge

To build a bridge to connect two disjoint lc-hulls, we need to find the tangent T to the lc-hulls of A and C, and also find the points s, in the lc-hull of A, and t, in the lc-hull of C. The points s and t are the points where T intersects the lc-hulls of A and C respectively. The task of finding the tangent T is done by doing a binary search along both of the hulls. Let $p_1, p_2, \ldots, p_m$ be the points of the lc-hull of A ordered by their y-coordinate. Let $q_1, q_2, \ldots, q_k$ be the points of the lc-hull of C ordered by their y-coordinate. We start with the points $p = p_{\lceil \frac{m}{2} \rceil}$ and $q = q_{\lceil \frac{k}{2} \rceil}$. We then check if the line that passes through p and q is tangent to the lc-hulls. If it is, we have found s and t. If the line is not a common tangent, Overmars and Van Leeuwen use the rules 1-10 below to determine if we continue the search in the upper part or lower part of the lc-hulls of A and C w.r.t. p and q respectively. This procedure is carried out until we find the tangent.

Before enumerating the cases that can occur when $\overrightarrow{qp}$ intersects the lc-hull of A and C, we will define some points that will be used to describe the cases.

- Let p^+ be the point in the lc-hull of A such that $p^+_y > p_y$ and p^+_y is the closest such y-value.

- Let p^- be the point in the lc-hull of A such that $p^-_y < p_y$ and p^-_y is the closest such y-value.

- Let q^+ be the point in the lc-hull of C such that $q^+_y > q_y$ and q^+_y is the closest such y-value.

- Let q^- be the point in the lc-hull of C such that $q^-_y < q_y$ and q^-_y is the closest such y-value.

The cases are the following:

1. In this case, q^-, q^+, p^-, and p^+ are right w.r.t. $\overrightarrow{qp}$. Hence, all points in the lc-hulls of the upper and lower half are to the right of $\overrightarrow{qp}$. So $\overrightarrow{qp}$ is a tangent to the lc-hulls. Hence, we have found s and t. See Figure 2.4(a).

2. In this case, q^+, p^-, and p^+ are right w.r.t. $\overrightarrow{qp}$ and q^- is left w.r.t. $\overrightarrow{qp}$. The lower part of the lc-hull of A w.r.t. p^- is eliminated. The upper par of the lc-hull of C with y-coordinate bigger than q is eliminated. See Figure 2.4(b).

3. In this case, q^-, p^-, and p^+ are right w.r.t. $\overrightarrow{qp}$ and q^+ is left w.r.t. $\overrightarrow{qp}$. The lower parts of the *lc*-hulls of A and C w.r.t. p^- and q^- respectively are eliminated. See Figure 2.4(c).

4. This case is similar to case 2, but the point that is left w.r.t. $\overrightarrow{qp}$ is p^+. The same parts of the *lc*-hulls as in case 2 are eliminated. See Figure 2.4(d).

5. This is similar to case 3, but the point that is left w.r.t. $\overrightarrow{qp}$ is p^-. The upper parts of the *lc*-hulls of A and C w.r.t. p^+ and q^+ respectively are eliminated. See Figure 2.4(e).

6. In this case, p^- and q^+ are right w.r.t. $\overrightarrow{qp}$ and p^+ and q^- are left w.r.t. $\overrightarrow{qp}$. The lower part of the *lc*-hull of A w.r.t. p^- is eliminated. The upper part of the *lc*-hull of C w.r.t. q^+ is eliminated. See Figure 2.4(f).

7. In this case, p^+ and q^+ are right w.r.t. $\overrightarrow{qp}$ and p^- and q^- are left w.r.t. $\overrightarrow{qp}$. Only the upper part of the *lc*-hull of C w.r.t. q^+ is eliminated. See Figure 2.4(g).

8. In this case, p^- and q^- are right w.r.t. $\overrightarrow{qp}$ and p^+ and q^+ are left w.r.t. $\overrightarrow{qp}$. Like in case 7, only one part of one *lc*-hull is eliminated. The part that is eliminated is the lower part the *lc*-hull of A w.r.t. p^-. See Figure 2.4(h).

9. In this case, p^+ and q^- are right w.r.t. $\overrightarrow{qp}$ and p^- and q^+ are left w.r.t. $\overrightarrow{qp}$. The tangents to the *lc*-hull of A and the tangent to the *lc*-hull of C that pass through p and q respectively intersect at a point whose y-coordinate is smaller than the one of the point of the *lc*-hull of C with biggest y-coordinate. The lower part of the *lc*-hull of C w.r.t. q^- is eliminated. See Figure 2.5(a).

10. In this case, q^-, q^+, p^-, and p^+ are the same w.r.t. $\overrightarrow{qp}$ as in case 9. But the point where the tangents intersect has a y-coordinate that is bigger than the one of the point of the *lc*-hull of A with the smallest y-coordinate. The upper part of the *lc*-hull of A w.r.t. p^+ is eliminated. See Figure 2.5(b).

2.4 Structuring the lc-hulls

The data structure that holds the *lc*-hulls and contains the information needed to deal with insertions and deletions is organised as follows. Its skeletal component is

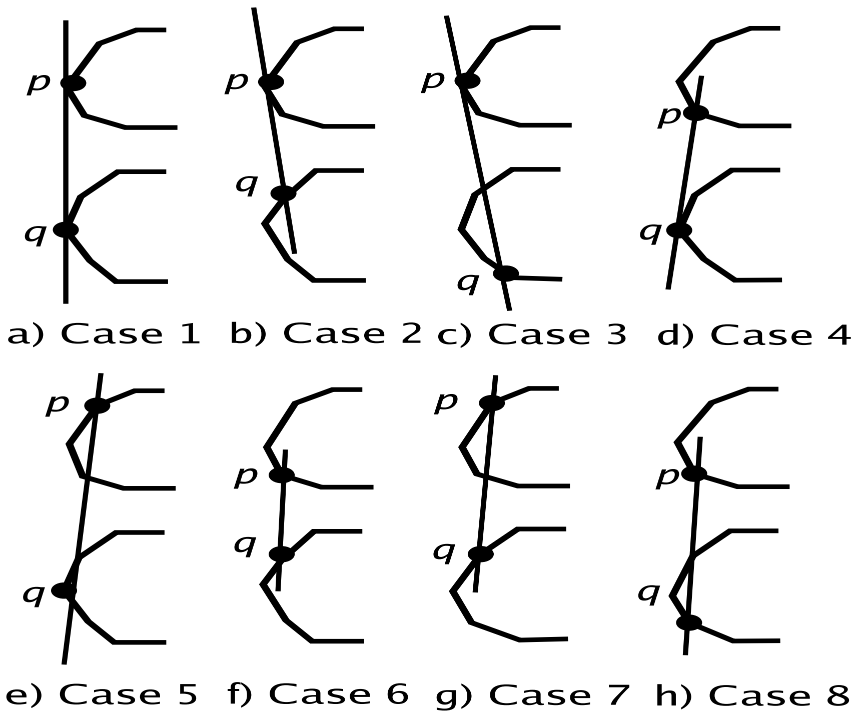


Figure 2.4: Cases 1-8 for determining the bridge

a height-balanced binary search tree whose leaves store the points of the current set, sorted by y-coordinate. The tree also stores in the internal nodes the lc-hull (and rc-hull) of the set of points in the subtree below that node. Keeping the points ordered by their y-coordinate allow Overmars and Van Leeuwen to recursively split the set in halves (the points stored in the left subtree of a node and the ones stored in the right subtree of a node) to build the lc-hull of the entire set. This process was described in section 2.3. To decide the type of height-balanced tree is left to the reader by Overmars and Van Leeuwen [2]. The one we used in our implementation and the reasons of our choice will be explained in section 3.2 of Chapter 3. Throughout this report, we will refer to the height-balanced tree structure used by Overmars and Van Leeuwen as the *top-level tree.*

Observation 2.3 The depth in a height-balanced tree is $O(\log N)$, where N is the total number of nodes in the tree. Since $N \leq 2n$, where n is the number of leaves in the tree, hence the height is $O(\log n)$.

Observation 2.4 The number of points stored in the right subtree and in the left subtree of a node in the top-level tree are not the same by a value of even 2

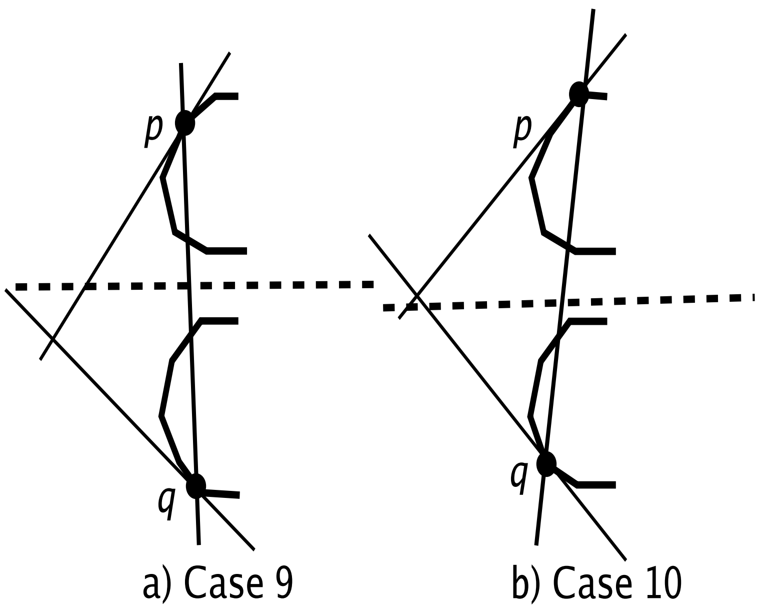


Figure 2.5: Cases 9 and 10 for determining the bridge

or 3 away. The “halves” in which the set is divided need not be equal. *However, since the tree is balanced, observation 2.3 implies that the depth of all the leaves is $O(\log n)$, where n is the number of points in the set.*

The lc-hulls are organised as follows. A node v of the height-balanced binary search tree will have a concatenable queue that stores the lc-hull of the set formed by points stored in leaves in its subtree. A leaf will have the lc-hull of the set formed only by the point stored in it, which is a trivial lc-hull consisting of just that point.

The set of points P stored in the subtree of v is divided into two halves: the set of points stored in the left subtree, C, and the set of points stored in the right subtree, A. Since the points are ordered by y-coordinate, the points in the left subtree correspond to the points below some line parallel to the y axis. On the other hand, the points in the right subtree are above this line. As described in section 2.3, we can build the lc-hull of P from the lc-hulls of C and A which are respectively stored in the left and right children of v. See Figure 2.6 for an example.

Overmars and Van Leeuwen [2] do not retain the complete lc-hulls of the child nodes of v. Instead, they simply store the segment that is not part of the lc-hull

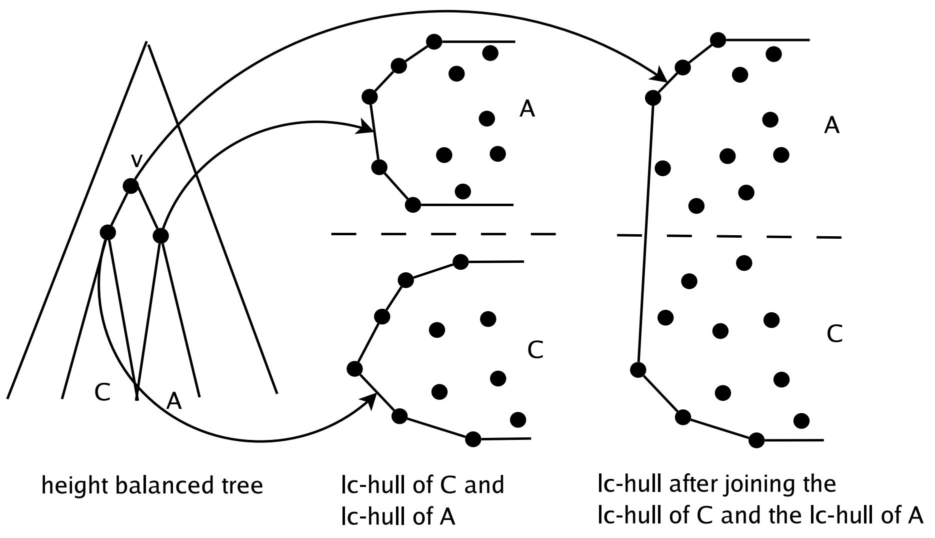


Figure 2.6: Storing the *lc*-hulls in the height-balanced tree.

stored in v. They do this to achieve the $O(\log^2 n)$ running time because it will take more than $O(\log n)$ time to copy the segments back to form the complete *lc*-hulls of the children. After splitting the *lc*-hulls of the children, the part of the *lc*-hull of C that is not used to form the *lc*-hull of P is stored in a concatenable queue in v's left child. Likewise, the part not used of the *lc*-hull of A is stored in v's right child. In node v, they also store the points of the bridge used to form the *lc*-hull. To fully reconstruct the *lc*-hulls of the children of v, they just split the *lc*-hull stored in v at the bridge and join each piece with the part stored in the corresponding children: the upper part of the *lc*-hull w.r.t. the bridge with the part stored in the right child and the lower part of the *lc*-hull w.r.t. the bridge with the part stored in the left child (recall definitions 2.2 and 2.3).

After the set and its *lc*-hull have been updated (after an insertion or deletion), the nodes in the balanced tree store the following. The root stores in a concatenable queue the complete *lc*-hull of the set and the points where the bridge connects the root's children *lc*-hulls. All the other nodes store the part of the *lc*-hull of the set of points stored at the leaves of its subtrees that does not form part of the *lc*-hull of set of points corresponding to its parent. If the *lc*-hull of the set of points stored in the leaves of the subtrees of a node is all part of the *lc*-hull of the set of points corresponding to the parent, then the concatenable queue stored in that node is an empty one. The nodes also store the points of the bridge that connect the *lc*-hulls of the set of points stored in the leaves of its left and right subtree.

2.5 Maintaining the Convex Hull

To insert or delete a point of the set of points and update the *lc*-hull of the set the process is the following. Overmars and Van Leeuwen [2] use two routines for it: **DOWN** and **UP**. They use the **DOWN** procedure to go down the tree in search of the leaf that stores the point to be deleted or the node where the point should be inserted. As they go down, at each node they explicitly build the *lc*-hulls of its two children (as described in section 2.4). Observe that this can be done with the operations described in section 2.2.1. Once the point is inserted/deleted, they use the routine **UP** to go up the tree updating the *lc*-hulls all the way to the root, that has the complete *lc*-hull. To update the *lc*-hull at each node, they determine the bridge to join the *lc*-hull of the set of points stored in the leaves left subtree and the one of the set of points stored in the leaves of the right subtree as described in the previous section.

The way to keep the tree balanced after an insertion/deletion, and to manage the concatenable queues that store the pieces of *lc*-hulls after the rebalancing operations are left to the reader in Overmars and Van Leeuwen [2]. Part of the reason this is left to the reader is because Overmars and Van Leeuwen do not specifically choose a particular data structure for the top level tree or the *lc* and *rc*-hulls. In sections 3.2.4 and 3.3 of Chapter 3, we will explain how we dealt with these issues for our implementation.

As mentioned before, the *rc*-hull is treated in the same way as the *lc*-hull. So during the **DOWN** and **UP** routines, the *rc*-hull is also updated. To update the convex hull of the current set, it is just the trivial join of the points that belong to the *lc* and *rc*-hulls after performing the **DOWN** and **UP** routines.

2.6 Running Time Analysis

Let the current set of points have n points and the current *lc*-hull N points. N is at most equal to n, so $N = O(n)$. Since the top-level tree is a height-balanced tree, observation 2.4 implies that the procedure **DOWN** takes at most $O(\log n)$ steps to get to the leaf node that stores the point that is deleted or where the leaf node with new point must be inserted. At each step, we built the *lc*-hulls of the children of the current node v. To do that, as explained before, we must split the *lc*-hull stored in v at the bridge. Since the *lc*-hull is stored in a concatenable queue, that operation takes $O(\log N)$. Then we must join the parts resulting after splitting the lc-hull with the ones stored in the children. Each of those operations take $O(\log N)$. At each step of the **DOWN** routine, we perform work that costs $O(\log N)$ time. Since $N = O(n)$, then the time spent at each step is $O(\log n)$. The **DOWN** routine finishes in $O(\log^2 n)$ time.

After using **DOWN**, they use **UP** to update the lc-hull. It will take $O(\log n)$ steps to reach to the root. At each step it updates the lc-hull of the node v it is visiting. To do this it first determines the bridge to join the two lc-hulls stored in v's children. This is done in $O(\log N)$ time, since it is a binary search over the two concatenable queues that store the lc-hulls of the children. Then it splits those concatenable queues taking $O(\log N)$ time. Finally, it joins the corresponding parts from the lc-hulls of the children and that operation takes $O(\log N)$. So each step of **UP** does work that takes $O(\log N)$. Like in **DOWN**, it is $O(\log n)$, and the total time that it takes to finish **UP** is $O(\log^2 n)$.

Since to update the rc-hull they do the same operations needed to update the lc-hull, both hulls are updated in $O(\log^2 n)$. To join the lc and rc-hulls it takes constant time.

If we use Overmars and Van Leeuwen's algorithm to compute the convex hull of set of n, it will be done on-line by first getting the convex hull of a set of one element. Then updates the convex hull of the set of one element after the insertion of the second point. Then it will update the convex hull of the set of two elements with the insertion of the third point, and so on. So when inserting the $i+1$ point, it will take $O(\log i)$ time to update the existing convex hull. After this update, we will have the convex hull of the first $i+1$ elements. So the total time to compute the convex hull of the n points is:

$$O(\log^2 2) + O(\log^2 3) + O(\log^2 4) + \ldots + O(\log^2 (n-1))$$

From this we get that the total running time is:

$$\sum_{k=2}^{n-1} O(\log^2 k) = O(n \log^2 n)$$

So, to build the convex hull of a set of n points, we are paying a higher price than less powerfull techniques, like Graham's Scan, that take $\Theta(n \log n)$ time to compute the convex hull of a set of n points. But, if we are dealing with a dynamic set, it is clear that each update is faster than computing again the convex hull of the new set with another algorithm.

Chapter 3

Data Structures

In this chapter, we will describe how we implemented the data structures used to store the points of the set and the points of the *lc* and *rc*-hulls. The data structure used to store the *lc* and *rc*-hulls is a concatenable queue (as mentioned in section 2.2.1 of Chapter 2). The data structure used to store the points of the set and organise the *lc* and *rc* hulls is a height-balanced search tree.

3.1 Concatenable Queues

A concatenable queue is a data structure that can process the following instructions: insert, delete, find, concatenate, and split. These instructions take $O(\log N)$ time, where N is the number of elements stored in the queue [1].

We decided to do our own implementation of a concatenable queue because we wanted to store points from the plane (with x and y-coordinates) indexed by their y-coordinate. Also, at each node we need to store information that is not normally stored in implementations of a concatenable queue. Our implementation was done following the description of the concatenable queue data structure made by Aho, Hopcroft, and Ullman [1].

In the following sections, we will describe how we implemented the concatenable queue, and the three instructions (insert, split, and concatenate) that are used in our implementation of the algorithm of Overmars and Van Leeuwen.

3.1.1 2-3 Trees

To implement a concatenable queue, we used a 2-3 Tree. A linearly ordered set can be represented by a 2-3 Tree by assigning elements of the set to the leaves of the tree. In our case, the elements of the set are the points of the contour of the lc/rc-hull. The points are indexed by their y-coordinate and are assigned to the leaves in left-to-right order.

Definition 3.1 A *2-3 Tree* is a tree in which every node that is not a leaf has 2 or 3 children, and every path from the root to a leaf is of the same length.

In order to be able to keep the points ordered by their y-coordinate, we need to store values L and M at each node v, which is not a leaf. L is the value of the highest y-coordinate of the points stored in the leaves of the subtree rooted at the left child of v. M is the value of the biggest y-coordinate of the points stored in the leaves of the subtree rooted at the second child of v (right child in case of 2-nodes and middle child in case of 3-nodes).

All nodes store pointers to its child and parent nodes. Also, each node v, which is not a leaf, stores pointers to the leaf nodes that store the points with the highest and lowest y-coordinates among the points stored in the leaves of the subtree rooted at v. We will explain why we need these pointers and how we use them in section 3.5.1.

3.1.2 Insertion of a Point

To insert a point $p = (p_x, p_y)$ in a concatenable queue Q, we first create a new leaf node v and store p in it. Then we must locate the position for v. We have three cases:

1. If Q is empty, we set v to be the root of Q.

2. If Q consists of a single leaf u, we create a new 2-node r and set it to be the root of Q. We make v the left child and u the right child of r if $p_y < q_y$, where q is the point stored in u. Otherwise, we make v the right child and u the left child of r.

3. If Q consists of more than one node, need to search for the inner node u, which has leaves as children, that will be the parent of v. We do this by going down the tree, starting at the root, as follows. At the current node z, we have three cases:

    i. If $p_y < z.L$, we go to $z.left$.

ii. If z is a 3-node and $z.L \leq p_y < z.M$, we go to $z.middle$.

iii. Otherwise, we go to $z.right$.

This continues until we reach u. If u is a 2-node, let $u.middle \leftarrow u.left$ and $u.left \leftarrow v$ if $p_y < u.L$. If $u.L \leq p_y < u.M$, let $u.middle \leftarrow v$. Otherwise, let $u.middle \leftarrow u.right$ and $u.right \leftarrow v$. If u is a 3-node, then we set the left child and middle child of u to be the two leaf nodes (from $u.left$, $u.middle$, $u.right$, and v) that store the points with smaller y-coordinate. Also, we set the right child to be the leaf node with the point with highest y-coordinate. Let v' be the leaf node that stores the point with the third highest y-coordinate. Then we call the procedure *addChild* with parameters u and v'.

Algorithm *addChild*(v, aux)

Input: v, aux are pointers to nodes of a concatenable queue.

```
1.  Create a new vertex $v'$;
2.  $v'.left \leftarrow aux$;
3.  $v'.right \leftarrow v.right$;
4.  $v.right \leftarrow v.middle$;
5.  if $v$ has no father
6.    then Create a new node $r$ and set it as the root;
7.         $r.left \leftarrow v$;
8.         $r.right \leftarrow v'$;
9.    else if $v.parent$ has only 2 children
10.          then if $v$ is a left child
11.                 then $v.parent.middle \leftarrow v'$;
12.                 else $v.parent.middle \leftarrow v.parent.right$;
13.                      $v.parent.right \leftarrow v'$;
14.          else if $v$ is a left child
15.                 then $aux1 \leftarrow v.middle$;
16.                      $v.middle \leftarrow v'$;
17.                 else if $v$ is a middle child
18.                        then $aux1 \leftarrow v'$
19.                        else $aux1 \leftarrow v.parent.right$;
20.                             $v.parent.right \leftarrow v'$;
21.               addChild($v.parent$, $aux1$);
```

3.1.3 Concatenating Two Queues

The concatenate instruction takes as input two concatenable queues $Q1$ and $Q2$ such that all the points stored in $Q1$ have a smaller y-coordinate than the ones stored in $Q2$. The output is a new concatenable queue Q.

The trivial cases are when either $Q1$ or $Q2$ is empty. If $Q2$ is the empty queue, then $Q \leftarrow Q1$; otherwise, $Q \leftarrow Q2$.

Now we consider the cases when both $Q1$ and $Q2$ are not empty. Let $h1$ be the height of $Q1$, and let $h2$ be the height of $Q2$. Let $r1$ and $r2$ be the root of $Q1$ and the root of $Q2$ respectively. The cases are the following:

1. If $h1 = h2$, we create a new 2-node r and set it to be the root of Q. Then, we make $r1$ and $r2$ be the left and right child of r respectively. Let $r.L$ be $r1.M$ and let $r.M$ be $r2.M$.

2. If $h1 > h2$, let v be the rightmost node of $Q1$, which has depth $h1 - h2$ and $r1$ the root of Q. If v is a 2-node, let $v.middle \leftarrow v.right$ and $v.right \leftarrow r2$. If v is a 3-node, we call the procedure *addChild* with parameters v and $r2$.

3. If $h1 < h2$, let v be the leftmost node of $Q2$, which has depth $h2 - h1$ and $r2$ the root of Q. If v is a 2-node, let $v.middle \leftarrow v.left$ and $v.left \leftarrow r1$. If v is a 3-node, let $aux \leftarrow v.middle$, $v.middle \leftarrow v.left$, and $v.left \leftarrow r1$. Then we call the procedure *addChild* with parameters v and aux.

3.1.4 Splitting a Queue

The result of splitting a concatenable queue Q w.r.t. a point $p = (p_x, p_y)$, stored in Q, is two concatenable queues $Q1$ and $Q2$. The points stored in $Q1$ have a y-coordinate smaller or equal to the y-coordinate of p, so p is stored in $Q1$. The points stored in $Q2$ have a y-coordinate higher than the one of p.

To split Q w.r.t. p, we follow the path from the root to the leaf node u that has p stored. Observe that $Q1$ and $Q2$ start being empty queues. Let v be the current node in the path to u. We have two cases:

1. If v is a 2-node, we have two cases:

   i. If $p_y \leq v.L$, let $Q2'$ be a concatenable queue such that it is rooted at $v.right$. Then we concatenate $Q2'$ with $Q2$ into $Q2$, and we continue to the node $v.left$.

  ii. Otherwise, let $Q1'$ be a concatenable queue such that it is rooted at $v.left$. Then we concatenate $Q1$ with $Q1'$ into $Q1$, and we continue to the node $v.right$.

2. If v is a 3-node, we have three cases:

   i. If $p_y \leq v.L$, we join the concatenable queues rooted at $v.middle$ and $v.right$ into $Q2'$. Then we concatenate $Q2'$ with $Q2$ into $Q2$, and we continue to the node $v.left$.

   ii. If $v.L < p_y \leq v.L2$, let $Q1'$ and $Q2'$ such that their roots are $v.left$ and $v.right$ respectively. Then we concatenate $Q1$ with $Q1'$ into $Q1$ and $Q2'$ with $Q2$ into $Q2$. We continue to the node $v.middle$.

   iii. Otherwise, we concatenate the concatenable queues rooted at $v.left$ and $v.middle$ into $Q1'$. Then we concatenate $Q1$ with $Q1'$ into $Q1$, and we continue to the node $v.right$.

When we reach u, we concatenate $Q1$ with the concatenable queue formed just by the leaf node u into $Q1$.

We also split Q w.r.t. p into $Q1$ and $Q2$ such that p is stored in $Q2$. The only difference with the procedure described above is that when we reach u. Instead of concatenating $Q1$ with u, we concatenate u with $Q2$. Henceforth, we will say *split left* when we split Q w.r.t. p and p is stored in $Q1$,and *split right* when we split Q w.r.t. p and p is stored in $Q2$.

3.2 Top-Level Tree

The top-level tree we chose for our implementation of the Overmars and Van Leeuwen algorithm is similar to an AVL tree. The data stored in the tree are points in $\mathbb{R}^2$. The points are indexed according to their y-coordinate. The inner nodes of the tree store the information needed to build the *lc*-hull (and *rc*-hull) of the points stored in the leaves of their subtrees. We will see that, like an AVL tree, the difference between the heights of the right subtree and left subtree of any node, is at most one.

The difference between our tree and an AVL tree is that the data (in our case points of the plane) is stored only in the leaves. In AVL trees, every node (either internal or leaf) contains key values. For our tree, internal nodes represent the

lc-hull (and rc-hull) of sets of points, rather than points themselves. This change will affect how we re-balance the top-level tree (see section 3.2.4).

Since our tree is balanced and the the points of the set are stored only in the leaves, observation 2.4 holds for our tree. Hence, going down the tree to any leaf will take $O(\log n)$ steps, where n is the number of points stored in the leaves of the tree. This will help our implementation of the Overmars and Van Leeuwen algorithm to handle insertions and deletions in $O(\log^2 n)$ time.

Note that the details of the top-level tree are not spelled out in the Overmars and Van Leeuwen paper [2], they just mentioned that the tree should be balanced. Part of this project was to work out the details to do the implementation of the top-level tree.

In the following sections, we will describe the information stored in each node of the top-level tree. Also, we will outline the operations to insert and delete a point from the tree, to keep the tree balanced, and to manage the lc and rc-hulls. We will only focus on the lc-hull because the rc-hull is treated in a symmetric way.

3.2.1 Node's Structure

Before describing the the structure of the top-level tree nodes, we will give the following definition.

Definition 3.2 Let v be a node of the top-level tree. Let S_v be the set of points stored in the subtree under v.

The information that is stored in the each node v is the following:

- Pointers to its child and parent nodes. If these exist, let w be the left child node, let x be the right child node, and let u be the parent node.

- A concatenable queue Q_l where we store the part of the lc-hull of S_v that does not form part of the lc-hull of S_u (as described in section 2.4 of Chapter 2).

- A concatenable queue Q_r where we store the part of the rc-hull of S_v that does not form part of the rc-hull of S_u (as described in section 2.4 of Chapter 2).

- The points s_l and t_l where the bridge that joins the lc-hulls of S_w and S_x intersects them. s_l is in the lc-hull of S_x and t_l is in the lc-hull of S_w.

- The points s_r and t_r where the bridge that joins the rc-hulls of the points stored in the subtrees of v intersects them. s_r is in the rc-hull of the points stored in the right subtree and t_r is in the rc-hull of the points of the left subtree.

- A balance factor that indicates if the heights of the subtrees are equal (it has the value 0) or which subtree's height is bigger by one (-1 if the left subtree is bigger and 1 if the right subtree is bigger).

- If v is a leaf node, it stores a point of the set and the label is equal to the point's y-coordinate.

- Otherwise, the label is equal to the y-coordinate of the point stored in the rightmost leaf of the left subtree of v. Observe that the label of inner nodes is the maximum y-value among the points in S_w.

Observation 3.1 The root node stores in Q_l and Q_r the complete lc and rc-hulls of the set of points stored in the top-level tree. During the **UP** and **DOWN** routines, some inner nodes will temporarily store in Q_l and Q_r the complete lc and rc-hull of the points in the subtree below them.

3.2.2 Insertion

When a point $p = (p_x, p_y)$ is inserted into the set of points, first we create a leaf node v to store it. We store the lc-hull of the set $\{p\}$ in $v.Q_l$. Observe that the lc-hull (and rc-hull) of $\{p\}$ is the trivial one containing only p. To insert v into the top-level tree, we proceed down the tree. At each internal node u we explicitly build the lc-hulls (and rc-hulls) of the set of points stored at the left subtree and right subtree, and store them in the Q_l (and Q_r) structures of the left child node and right child node respectively. The operation of building the lc-hull is done in $O(\log n_l)$, where n_l is the number of points in Q_l. The process to build the lc-hulls (and rc-hulls) of the sets of points stored in the left and right subtrees will be explained in section 3.4. We descend down the left subtree if the label u is greater than p_y. This is process is done until we reach a leaf v'. Observe that this is our implementation of the **DOWN** routine described in section 2.5

of Chapter 2.

Let v' be the leaf reached after going down the tree to insert point p, which is stored in v, and let u be v''s parent. After reaching v', we create a new inner node u'. We have two cases:

1. If v' is a right child of u, we set u' to be u's right child. Then we have two cases:

    i. If p_y is bigger than v''s label, let v be $u'.right$ and let v' be $u'.left$.

    ii. Otherwise, let v be $u'.left$ and let v' be $u'.right$.

2. If v' is a left child of u, let u' be $u.left$, let v be $u'.left$, and let v' be $u'.right$.

See Figure 3.1 to see how v is inserted when v' is a right child and when p_y is bigger than the label of v'. We store in $u'.Q_l$ the lc-hull of $\{p, q\}$, where q is the point stored in v'. Observe that the lc-hull (and rc-hull) of $\{p, q\}$ is formed by the points p and q. We set $v.Q_l$ and $v'.Q_l$ to be empty queues because the complete lc-hull of the sets $\{p\}$ and $\{q\}$ forms part of the lc-hull of $\{p, q\}$.

After inserting v, we go up the tree with the **UP** routine described in section 2.5 of Chapter 2. We implemented this routine (in the insertion case) as follows in order to update the balance factor of the nodes. Every time we walk one step up the top-level tree, let z be the current node. We have the following cases.

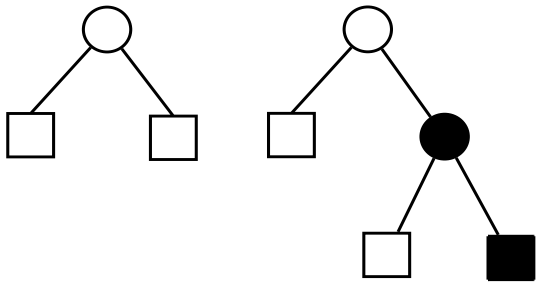
- If we reached z from its right child and we have not stopped updating the balance factor, we add one to the balance factor of z.

- If we reached z from its left child and we have not stopped updating the balance factor, we subtract one to the balance factor of z.

If after this change is made z's balance factor gets a value of zero or z was rebalanced, the changes to the subtree rooted at z will not change the height of the subtrees of any upper node.

At every node z (whether we modify its balance factor or not), we have the following cases:

- If z's balance factor gets the value -1 or 1, we explicitly build the lc-hull (and rc-hull) of S_z and store it in $z.Q_l$.



The squares are leaf nodes and the ones that are colored are the new nodes inserted.

Figure 3.1: Insertion of a new point whose y-coordinate is bigger than the one of the point stored in the right leaf.

- If z's balance factor gets the value 0, we explicitly build the lc-hull (and rc-hull) of S_z, store it in $z.Q_l$, and stop updating the balance factor of the nodes above z.
- If z's balance factor gets the value -2 or 2, we re-balance z (this will be described in section 3.2.4) and then we rearrange the lc-hulls (and rc-hulls) and update them (this will be described in section 3.3). We also stop updating the balance factor of the node above z.

This procedure to build the lc-hull (and rc-hull) of the points in the subtree beneath the current node will be described in section 3.5.

Observation 3.2 We do not need to update after the insertion of a point the labels of the inner nodes because when we go down the top-level tree to insert the node we never go down to the left subtree of a node unless the y-coordinate of the new point is smaller than the label. Hence, the point with higher y-coordinate stored in the left subtree of any node does not change.

3.2.3 Deletion

To delete a node p from the set of points, we need to proceed down the tree in search of the leaf node v that stores p. We go down the tree in the same manner as we do for an insertion (described in section 3.2.2). Let u be the parent of v. After we have reached v, we replace u with v's sibling as the corresponding child node of u's parent. Then we go up the tree with an implementation of the **UP** routine described in section 2.5 of Chapter 2. With this implementation of **UP**, we update the balance factor of the nodes, the labels, keep the tree balanced, and update the lc-hull (and rc-hull).

Observation 3.3 The inner node whose label is updated after a point is deleted from the set is the one who is the root of the subtree where v is the rightmost leaf of the left subtree.

At each step we walk up the top-level tree, let z be the current node. We have two cases:

- If we reached z from its right child and we have not stopped updating the balance factor, we subtract one to the balance factor of z.
- If we reached z from its left child and we have not stopped updating the balance factor, we add one to the balance factor of z.

Observe that if the balance factor of z becomes non-zero and z is balanced the height of the subtree below z did not change after the deletion of the point. Hence, we do stop updating the balance factor of the nodes above z because the changes of the subtree rooted at z will not affect the height of the subtrees of any upper node.

At every node z (whether we modify its balance factor or not), we have the following cases:

- If z's balance factor gets the value -1 or 1, we explicitly build the lc-hull (and rc-hull) of S_z, store it in $z.Q_l$, and stop updating the balance factor of the nodes above z.
- If z's balance factor gets the value 0, we explicitly build the lc-hull (and rc-hull) of S_z and store it in $z.Q_l$.
- If z's balance factor gets the value -2 or 2, we re-balance z (this will be described in section 3.2.4) and then we rearrange the lc-hulls (and rc-hulls) and update them (this will be described in section 3.3).

This procedure to build the lc-hull (and rc-hull) of the points in the subtree beneath the current node will be described in section 3.5.

Observe that if we had to re-balance z we still continue updating the balance factor of the nodes above z. This is because after re-balancing z, the height of the subtree rooted at z is reduced.

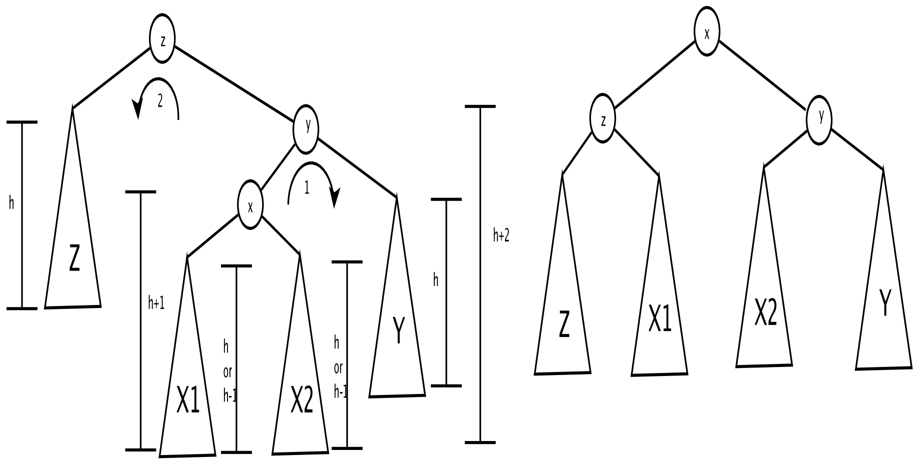


Figure 3.2: Clockwise-Anticlockwise Rotation.

3.2.4 Keeping the Tree Balanced

We say that a node is out of balance if one of its subtree has height h and the other has height $h + 2$. To re-balance the tree, we use local rotations similar to the ones used by AVL trees to balance the lowest node who is out of balance. In the case of an insertion, after balancing the lowest unbalanced node all the nodes above it become balanced. In case of a deletion, we still need to go up the tree checking if an upper node got out of balance after the rotation. We do this in our implementation because after we delete or add a point we go up the tree updating the balance factor; and when we find the first one out of balance, we re-balance it. This process of going up after an insertion and deletion has been described in sections 3.2.2 and 3.2.3 respectively.

Let z be the first node out of balance such that the height of the right subtree is $h + 2$ and the height of the left subtree h. Let y be z's right child. We have two cases:

- If y's left subtree has a height h and y's right subtree has a height $h - 1$, then we use the *clockwise-anticlockwise* double rotation.
- Otherwise, we re-balance z with the *anticlockwise* rotation. Observe that before the rotation is performed, y's right subtree's height is equal or bigger than y's left subtree.

Figures 3.2 and 3.3 show the z node and its subtrees before and after being rebalanced with the clockwise-anticlockwise double rotation and the anticlockwise rotation respectively.

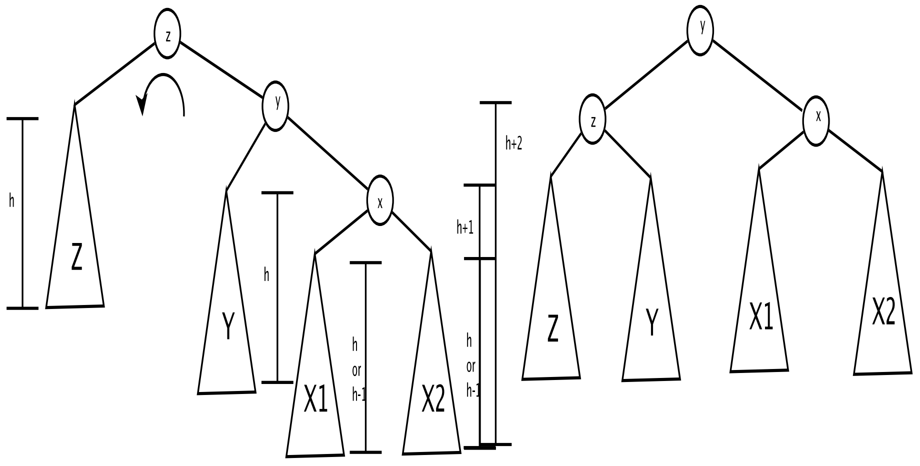


Figure 3.3: Anticlockwise Rotation.

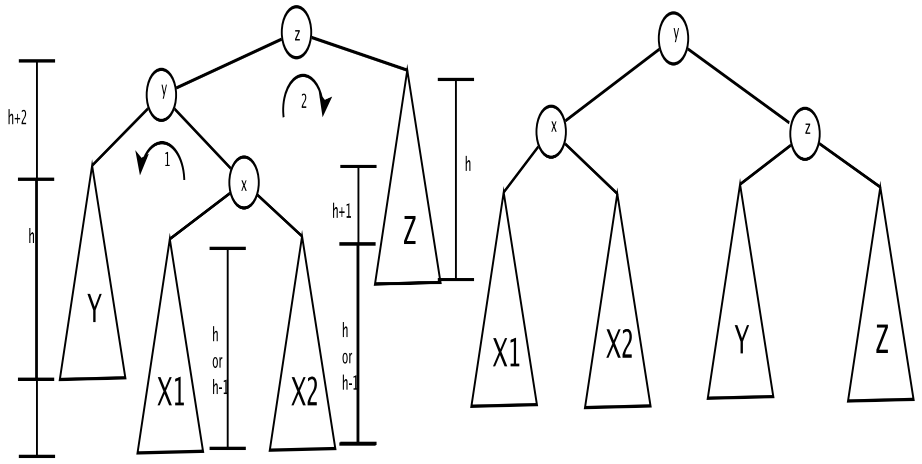


Figure 3.4: Anticlockwise-Clockwise Rotation.

Let z be the first node out of balance such that the height of the right subtree is h and the height of the left subtree $h + 2$. Let y be z's left child. We have two cases:

- If y's right subtree has a height h and y's left subtree has a height $h - 1$, then we use the *anticlockwise-clockwise* double rotation.

- Otherwise, we re-balance z with the *clockwise* rotation.

Figures 3.4 and 3.5 show the z node and its subtrees before and after being rebalanced with the anticlockwise-clockwise double rotation and the clockwise rotation respectively.

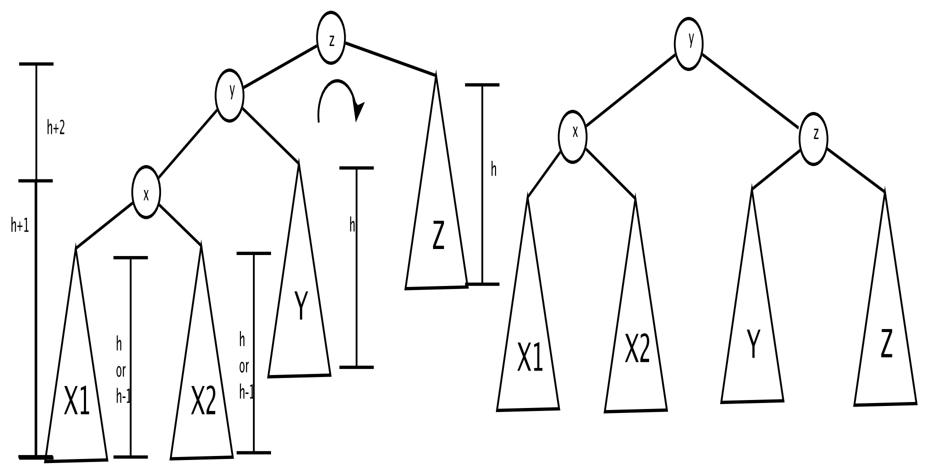


Figure 3.5: Clockwise Rotation.

Rotation Algorithms

The following are the algorithms for the Clockwise Rotation, Anticlockwise Rotation, Clock_Anticlockwise Double Rotation, and Anticlockwise_Clockwise Double Rotation.

Algorithm *Clockwise Rotation*(z)

Input: z is a pointer to a node of the top-level tree.

```
1.   $y \leftarrow z.left$;
2.   $z.left \leftarrow y.right$;
3.   $y.right \leftarrow z$;
4.   if $z = root$
5.      then $root \leftarrow y$;
6.      else  if $z$ is a left child
7.               then $z.parent.left \leftarrow y$
8.               else  $z.parent.right \leftarrow y$
9.   if $y.balance\_factor = -1$
10.     then $y.balance\_factor \leftarrow 0$;
11.          $z.balance\_factor \leftarrow 0$;
12.     else $z.balance\_factor \leftarrow -1$;
13.          $y.balance\_factor \leftarrow 1$;
```

Algorithm *Anticlockwise Rotation*(z)

Input: z is a pointer to a node of the top-level tree.

```
1.   $y \leftarrow z.right$;
```

```
2.  $z.right \leftarrow y.left$;
3.  $y.left \leftarrow z$;
4.  if $z = root$
5.      then $root \leftarrow y$;
6.      else  if $z$ is a left child
7.                then $z.parent.left \leftarrow y$
8.                else  $z.parent.right \leftarrow y$
9.  if $y.balance\_factor = 1$
10.     then $y.balance\_factor \leftarrow 0$;
11.          $z.balance\_factor \leftarrow 0$;
12.     else  $z.balance\_factor \leftarrow 1$;
13.           $y.balance\_factor \leftarrow -1$;
```

Algorithm *Clockwise_Anticlockwise Double Rotation*(z)
Input: z is a pointer to a node of the top-level tree.

```
1.  $y \leftarrow z.right$;
2.  $x \leftarrow y.left$;
3.  $z.right \leftarrow x.left$;
4.  $y.left \leftarrow x.right$;
5.  $x.right \leftarrow y$;
6.  $x.left \leftarrow z$;
7.  if $z = root$
8.      then $root \leftarrow x$;
9.      else  if $z$ is a left child
10.               then $z.parent.left \leftarrow x$
11.               else  $z.parent.right \leftarrow x$
12. if $x.balance\_factor = 0$
13.     then $y.balance\_factor \leftarrow 0$;
14.          $z.balance\_factor \leftarrow 0$;
15.     else  if $x.balance\_factor = 1$
16.               then $y.balance\_factor \leftarrow 0$;
17.                    $z.balance\_factor \leftarrow -1$;
18.               else  $y.balance\_factor \leftarrow 1$;
19.                     $z.balance\_factor \leftarrow 0$;
20. $x.balance\_factor \leftarrow 0$;
```

Algorithm *Anticlockwise_Clockwise Double Rotation*(z)

Input: z is a pointer to a node of the top-level tree.

```
1.  $y \leftarrow z.left$;
2.  $x \leftarrow y.right$;
3.  $z.left \leftarrow x.right$;
4.  $y.right \leftarrow x.left$;
5.  $x.left \leftarrow y$;
6.  $x.right \leftarrow z$;
7.  if $z = root$
8.     then $root \leftarrow x$;
9.     else if $z$ is a left child
10.             then $z.parent.left \leftarrow x$
11.             else $z.parent.right \leftarrow x$
12. if $x.balance\_factor = 0$
13.    then $y.balance\_factor \leftarrow 0$;
14.         $z.balance\_factor \leftarrow 0$;
15.    else if $x.balance\_factor = 1$
16.            then $y.balance\_factor \leftarrow -1$;
17.                 $z.balance\_factor \leftarrow 0$;
18.            else $y.balance\_factor \leftarrow 0$;
19.                 $z.balance\_factor \leftarrow 1$;
20. $x.balance\_factor \leftarrow 0$;
```

3.3 Managing the lc-hull

After the local rotation is performed to re-balance a node z, the organisation of S_z changes. So, we have to prepare in order to change the *lc*-hull stored in z. The operations we do, to construct the *lc*-hull of S_z, before and after a single or a double rotation to re-balance z (described in section 3.2.4) are different. First we will explain what we do before and after an anticlockwise rotation. Then, we will do the description of what we do before and after a clockwise-anticlockwise rotation. The operations done before and after the clockwise and anticlockwise-clockwise rotations can be described similarly and symmetrical.

Let z be the lowest unbalanced node in the top-level tree, and suppose that it needs an anticlockwise rotation to be rebalanced. Let y be z's right child, let w

be y's left child, and let x be y's right child. Observe that since we are going up the tree with the **UP** routine, we have already built the lc-hull of S_y and stored it in $y.Q_l$ by the time we have reached z. Before re-balancing z, we first use $y.Q_l$, $x.Q_l$, and $w.Q_l$ to explicitly build the lc-hulls of S_w and S_x (this procedure was described in section 2.4 of Chapter 2). We will store the lc-hulls of S_w and S_x in $x.Q_l$ and $w.Q_l$ respectively. After doing these operations, we perform the anticlockwise rotation. Observe that the set of points stored in y's left subtree has now changed. Before updating $y.Q_l$, we need first to update $z.Q_l$ because z is now y's left child. We explicitly have the lc-hull of the set stored in z's left subtree, since it was built when we went down the tree. Also, we have the lc-hull of the set stored in z's right subtree because that subtree is rooted at w and we explicitly built that lc-hull of S_w before the rotation. Using these two lc-hulls, we can update $z.Q_l$. Then we can update $y.Q_l$ because now we have the complete lc-hulls of the sets stored at its left and right subtrees.

Let z be the lowest unbalanced node and suppose that it needs a clockwise-anticlockwise rotation to be rebalanced. Let y be z's right child, let x be y's left child, and let w be y's right child. Observe that since we are going up the tree with the **UP** routine, we have already built the lc-hull of S_y and stored it in $y.Q_l$. Before doing the clockwise-anticlockwise rotation, we explicitly build the lc-hull of S_x, and we will store it in $x.Q_l$. We also build the lc-hull of S_w and store it in w. Then, we explicitly build the lc-hulls of the sets stored in x's left and right subtrees and store them in x's left and right child nodes respectively. After performing these operations, we perform the clockwise-anticlockwise rotation to balance z. Observe that the sets of points stored in z's and y's subtrees have now changed. The set stored in y's left subtree is the one that was stored in x's right subtree before the rotation was carried out. We explicitly built the lc-hull of x's right subtree and store it in the root of this subtree, which after the rotation is now y's left subtree. The set S_w did not change after the rotation, and we built its lc-hull before carrying out the rotation. Since we have the lc-hull of the set stored in y's left subtree and the lc-hull of the set stored in y's right subtree, we can update the lc-hull of the set stored in y's subtrees and store it in $y.Q_l$. Now, we can update $x.Q_l$ because we explicitly have the lc-hulls of the set stored in its left subtree, which is rooted at z, and the lc-hull of the set stored in its right subtree, which is rooted at y.

3.4 Building the lc-hulls during the DOWN routine

In this section, we will describe how we explicitly build the lc-hulls of the sets of points in the left and right subtrees of a node v that is visited during the **DOWN** routine.

Let v be the current node during the **DOWN** routine. Let v_l be v's left child and let v_r be v's right child. As mentioned in section 3.2.1 of Chapter 3, v stores the points, t_l and s_l, of the lc-hulls of S_{v_l} and S_{v_r} respectively, where the bridge intersects the lc-hulls of S_{v_l} and S_{v_r}. At this point of the **DOWN** routine, $v.Q_l$ stores the lc-hull of the set S_v. The node v_r stores, in $v_r.Q_l$, the upper part of lc-hull of S_{v_l} w.r.t. t_l. Likewise, v_l stores in $v_l.Q_l$ the lower part of lc-hull of S_{v_r} w.r.t. s_l (recall definitions 2.2 and 2.3). Observe that the upper part of the lc-hull of S_{v_l} w.r.t. t_l is the part of the lc-hull of S_{v_l} that does not form part of the lc-hull of S_v. Likewise, the lower part of the lc-hull of S_{v_r} w.r.t. s_l is the part of the lc-hull of S_{v_r} which is not part of the lc-hull of S_v.

To compute the lc-hulls of S_{v_l} and S_{v_r} first we split right $v.Q_l$ w.r.t. s_t (as defined in section 3.1.4 of Chapter 3). Let Q_C and Q_A be the resulting concatenable queues such that the Q_A is the upper part of $v.Q_l$ w.r.t. s_t and Q_C the lower part w.r.t. t_l. To form the lc-hull of S_{v_l}, we concatenate Q_C with $v_l.Q_l$. To form the lc-hull of S_{v_r}, we concatenate $v_r.Q_l$ with Q_A. The resulting lc-hulls of S_{v_l} and S_{v_r} are stored in $v_l.Q_l$ and $v_r.Q_l$ respectively.

As mentioned in observation 3.1, the root of the top-level tree stores the lc-hull of the current set of points. When we start the **DOWN** routine when we are going to insert or delete a point, we have the lc-hull of the current set, so we can split it and form the lc-hulls of sets stored in its left and right subtree, and continue this way until we reach the leaf node where the new leaf node is going to be inserted or the leaf node that stores the point that is going to be deleted.

3.5 Building the lc-hulls during the UP routine

In this section, we will describe how we explicitly build the lc-hull of S_v when a node v is visited during the **UP** routine.

Let v be the current node during the **UP** procedure. Let v_l be the left child of v and let v_r be the right child of v. Observe that at this step of the **UP** procedure, $v_l.Q_l$ stores the lc-hull of S_{v_l}. Likewise, $v_r.Q_l$ stores the lc-hull of S_{v_r}.

This notation will be used in sections 3.5.1 and 3.5.2.

3.5.1 Determining the bridge between two lc-hulls

As mentioned in section 2.3 of Chapter 3, to construct the lc-hull of the set S_v the first thing we need to find is the bridge (recall definition 2.1) that connects the lc-hulls of S_{v_l} and S_{v_r}. We search for the points t_l, in the lc-hull of S_{v_l}, and s_l, in the lc-hull of S_{v_r}, where the bridge starts and ends. The task of finding t_l and s_l is carried out by doing a binary search along $v_l.Q_l$ and $v_r.Q_l$.

The binary search starts at the roots of $v_l.Q_l$ and $v_r.Q_l$, which are concatenable queues. We go down the concatenable queues until we find t_l and s_l. The binary search is done as follows. Let z_{Q_C} be the current node of $v_l.Q_l$. Let z_{Q_A} be the current node of $v_r.Q_l$. Let the points p, p^-, p^+, q, q^-, and q^+ be as follows:

- Let q be the point with biggest y-coordinate stored in the left subtree of z_{Q_C}.
- Let q^- be the point with second biggest y-coordinate stored in the left subtree of z_{Q_C}. Observe that q^- is stored in a leaf node that is a sibling node of the leaf node that stores q.
- Let q^+ be the point with smallest y-coordinate stored in the second subtree of z_{Q_C} (right subtree if z_{Q_C} is a 2-node and middle subtree if z_{Q_C} is a 3-node).
- Let p be the point with smallest y-coordinate stored in the right subtree of z_{Q_A}.
- If z_{Q_A} is a 2-node, let p^- be the point with biggest y-coordinate stored in the left subtree of z_{Q_A}. Otherwise, let p^- be the point with biggest y-coordinate stored in the middle subtree of z_{Q_A}.
- Let p^+ be the point with second smallest y coordinate stored in the right subtree z_{Q_A}. Observe that p^+ is stored in a leaf node that is a sibling node of the leaf node that stores p.

Observe that to find the points q^-, q^+, p^-, and p^+ we do not have to go all the way down the concatenable queues $v_l.Q_l$ and $v_r.Q_l$, because every node stores a pointer to the leftmost leaf of its left subtree and a pointer to the rightmost leaf of its right subtree. This was described in section 3.1.1 of Chapter 3.

Checking whether a point lies to the left or right of a vector is a basic problem in Computational Geometry. It is well known that the solution for the problem of checking if the point p_2 is right or left w.r.t. the vector $\overrightarrow{p_0p_1}$ is to examine the cross product $(p_2 - p_0) \times (p_1 - p_0)$. We have three cases:

1. If $(p_2 - p_0) \times (p_1 - p_0) = 0$ then p_2 is collinear to $\overrightarrow{p_0p_1}$.
2. If $(p_2 - p_0) \times (p_1 - p_0) < 0$ then p_2 is left w.r.t. $\overrightarrow{p_0p_1}$.
3. If $(p_2 - p_0) \times (p_1 - p_0) > 0$ then p_2 is right w.r.t. $\overrightarrow{p_0p_1}$.

To check whether q^-, q^+, p^-, and p^+ lie right or left w.r.t. $\overrightarrow{qp}$ we use the following algorithm:

```
**Algorithm** *Right Turn*$(p0, p1, p2)$
**Input:** $p0, p1, p2$ are points in the plane.
**Output:** An integer with value 1 if $p2$ is right w.r.t. $\overrightarrow{p0p1}$, 0 if $p2$ is left w.r.t.
    $\overrightarrow{p0p1}$, and 2 if $p2$ is colinear to $\overrightarrow{p0p1}$.
1.  $cross\_prod \leftarrow ((p2_x - p0_x) \times (p1_y - p0_y)) - ((p2_y - p0_y) \times (p1_x - p0_x))$;
2.  if $cross\_prod = 0$
3.     then return 2;
4.     else if $cross\_prod < 0$
5.             then return 0;
6.             else return 1;
```

Now we refer to the 10 cases described in section 2.3.1 of Chapter 2 to see if we eliminate the lower part or the upper part of the lc-hulls of S_{v_l} and S_{v_r}.

1. If case 1 occurs, we have found t_l and s_l, which are q and p respectively.
2. If cases 2, 4, and 6 occur, we eliminate the lower part of the lc-hull of S_{v_r} w.r.t. to p^- and the upper part of the lc-hull of S_{v_l} w.r.t. q^+.
3. If case 3 occurs, we eliminate the lower parts of the lc-hull of S_{v_l} and S_{v_r} w.r.t. q^- and p^- respectively.
4. If case 5 occurs, we eliminate the upper parts of the lc-hull of S_{v_l} and S_{v_r} w.r.t. q^+ and p^+ respectively.
5. If case 7 occurs, we eliminate the upper part of the lc-hull of S_{v_l} w.r.t. q^+.

6. If case 8 occurs, we eliminate the lower part of the lc-hull of S_{v_r} w.r.t. p^-.

7. If case 9 occurs, we eliminate the lower part of the lc-hull of S_{v_l} w.r.t. q^-.

8. If case 10 occurs, we eliminate the upper part of the lc-hull of S_{v_r} w.r.t. p^+.

We have the following cases when we eliminate the upper part of the lc-hull of S_{v_l} w.r.t. q^+:

1. If q is stored in the rightmost leaf of the left subtree of z_{Q_C}, we go down to $z_{Q_C}.left$.

2. If q is stored in the rightmost leaf of the middle subtree of z_{Q_C}, we go down to $z_{Q_C}.middle$.

3. If q is stored in the leftmost leaf of the right subtree of z_{Q_C}, t_l can only be q or q^-.

4. If q is stored in the leftmost leaf of the middle subtree of z_{Q_C}, t_l can only be q or q^-.

If cases 3 or 4 occur, we go down to the leaf node that stores q and in the following steps we only check q and q^- to try to find t_l.

We have the following cases when we eliminate the lower part of the lc-hull of S_{v_r} w.r.t. p^-:

1. If p is stored in the leftmost leaf of the right subtree of z_{Q_A}, we go down to $z_{Q_A}.right$.

2. If p is stored in the leftmost leaf of the middle subtree of z_{Q_A}, we go down to $z_{Q_A}.middle$.

3. If p is stored in the rightmost leaf of the left subtree of z_{Q_A}, s_l can only be p or p^+.

4. If p is stored in the rightmost leaf of the middle subtree of z_{Q_A}, s_l can only be p or p^+.

If cases 3 or 4 occur, we go down to the leaf node that stores p and in the following steps we only check p and p^+ to try to find s_l.

We have three cases when we eliminate the lower part of the lc-hull of S_{v_l} w.r.t. q^-:

1. If q is stored in the rightmost leaf of the left subtree of z_{Q_C} or in the rightmost leaf of the middle subtree of z_{Q_C}, let $q^- \leftarrow q$ and $q \leftarrow q^+$. Let q^+ be the point stored in the sibling right next to the leaf node where q is stored.

2. If q is stored in the leftmost leaf of the middle subtree of z_{Q_C}, let q be the point with biggest y-coordinate of the middle subtree and let q^+ the point with smallest y-coordinate of the right subtree. Let q^- be be the point stored in the sibling right next to the leaf node where q is stored.

3. If q is stored in the leftmost leaf of the right subtree of z_{Q_C}, we go down to $z_{Q_C}.right$.

We have three cases when we eliminate the upper part of the lc-hull of S_{v_r} w.r.t. p^+:

1. If p is stored in the leftmost leaf of the right subtree of z_{Q_A} or in the leftmost leaf of the middle subtree of z_{Q_A}, let $p^+ \leftarrow p$ and $p \leftarrow p^-$. Let p^- be the point store in the sibling right next to the leaf node where p is stored.

2. If p is stored in the rightmost leaf of the middle subtree of z_{Q_A}, let p be the point with smallest y-coordinate of the middle subtree and let p^- the point with biggest y-coordinate of the right subtree. Let p^+ be be the point store in the sibling right next to the leaf node where p is stored.

3. If p is stored in the rightmost leaf of the left subtree of z_{Q_A}, we go down to $z_{Q_A}.left$.

We continue going down the concatenable queues z_{Q_C} and z_{Q_A} until case 1(of the cases described in section 2.3.1 of Chapter 2) occurs or we reach a leaf node in both concatenable queues. Observe that when we reach a leaf of z_{Q_C}, we have eliminated all other possibilities of other points to be t_l but the one stored in that leaf. Hence, we have found t_l. Likewise, we find s_l when we reach a leaf of z_{Q_A}.

3.5.2 Updating the lc-hull

Let T be the tangent to the lc-hulls of S_{v_l} and S_{v_r}. Let t_l and s_l the points in the lc-hulls of S_{v_l} and S_{v_r} respectively, which were found with the procedure described in section 3.5.1. Hence, T intersects the lc-hulls of S_{v_l} and S_{v_r} at t_l and

s_l. To update the lc-hull of S_v we do the following. Let Q_{C1} be the lower part and of the lc-hull of S_{v_l} obtained by doing a left split of $v_l.Q_l$ w.r.t. t_l. Let Q_{C2} be the upper part of the lc-hull of S_{v_l} obtained by doing a left split of $v_l.Q_l$ w.r.t. t_l. Let Q_{A1} be the lower part of the lc-hull of S_{v_r} obtained by doing a left split of $v_r.Q_l$ w.r.t. s_l. Let Q_{A2} be the upper part of the lc-hull of S_{v_r} obtained by doing a left split of $v_r.Q_l$ w.r.t. s_l. Then we concatenate Q_{C1} and Q_{A2} and store it in $v.Q_l$. We store Q_{C2} in $v_l.Q_l$ and Q_{A1} in $v_r.Q_l$.

Chapter 4

Implementation of the Algorithm

In this chapter, we will describe our implementation of a concatenable queue and the top-level tree. In sections 4.4 and 4.5, we will describe our implementation of the **DOWN** and **UP** routines.

In section 4.6, we will describe the user options of our implementation. This options allow the user to see the structure of the top-level tree and the *lc* and *rc*-hulls that are built as we go up and down the tree.

In section 4.7, we will describe the problems we encountered while doing the implementation and how we solved them.

Our source code (and executable) is stored on DICE in the directory ~/s0671978/PROJECT/. The source code is two C++ libraries (`cqueue.h` and `toplevel_tree.h`) and the C++ source code of the executable of the Overmars and Van Leeuwen algorithm (`updatehull.cc`).

4.1 Preliminary Design Decisions

The first important design decision was the choice of programming language. I decided to use an object-oriented programming language, and in particular C++. The main reason to take this choice was because of my knowledge and experience with C++.

The next decision we had to take was how to organise the implementation of the data structures. We decided to build two libraries: one with the implementation of the concatenable queue and one with the implementation of the top-level tree. This choice is now seen as having paid off; separating the two data structures into two libraries allowed us to test them separately and made the task of debugging easier.

4.2 Concatenable Queue

The library that contains our implementation of a concatenable queue is `cqueue.h`. Before implementing a concatenable queue, we defined the structure `node` that is used as a node of the concatenable queue. The structure `node` is defined as a node of a 2-3 Tree and it stores the information described in section 3.1.1 of Chapter 3. We implemented a concatenable queue with the class `ConcatenableQueue`. This class has as a public attribute a pointer to the root node. The operation to add a node is in the public procedure `ConcatenableQueue::add_node`.

The operations concatenate, split right, and split left are defined in the library `cqueue.h`. Since these are operations that will use objects of the class `ConcatenableQueue` we did not implement2 them as members of the class `ConcatenableQueue`. The functions that define this operations are:

- `ConcatenableQueue concatenate (ConcatenableQueue, ConcatenableQueue)`
- `void split_right(ConcatenableQueue, *ConcatenableQueue, *ConcatenableQueue, double)`
- `void split_left(ConcatenableQueue, *ConcatenableQueue, *ConcatenableQueue, double).`

4.3 Top-Level Tree

The library that contains our implementation of the top-level tree is `toplevel_tree.h`. Before implementing the top-level tree, we defined the structure `tree_node` that is used as a node of the top-level tree. This structure stores all the information described in section 3.2.1 of Chapter 3. To be able to store the concatenable queues Q_l and Q_r in the structure `tree_node` and use the operations concatenate and split, the library `toplevel_tree.h` includes the library `cqueue.h`.

We implemented the top-level tree with the class `Tree`. This class has as a public attribute a pointer to the root node. It has the public procedures `Tree::add_node` and `Tree::delete_node` that are used by the user to add a point to the current set of points or delete a point from it.

4.4 DOWN Procedure

The way we go down the top-level tree during the **DOWN** routine is defined in the procedure `Tree::add_node` in the case of an insertion. For a deletion, it is defined in the procedure `Tree::search_node`, which returns the leaf node that stores the point that is going to be deleted.

The procedure that builds the *lc*-hulls (and *rc*-hulls) as we go down the top-level tree in the way described in section 3.4 of Chapter 3 is the following:

```
void Tree::buildChildrensHulls(tree_node *parent)
{
    ConcatenableQueue upperLcHull, upperRcHull, lowerLcHull,
      lowerRcHull, completeUpperLcHull, completeUpperRcHull,
       completeLowerLcHull, completeLowerRcHull;
    split_right(parent->Ql,&lowerLcHull,&upperLcHull,
        parent->bridge2_lc.y_coord);
    split_right(parent->Qr,&lowerRcHull,&upperRcHull,
        parent->bridge2_rc.y_coord);
    completeUpperLcHull=
              concatenate(parent->rightchild->Ql,upperLcHull);
    completeUpperRcHull=
              concatenate(parent->rightchild->Qr,upperRcHull);
    completeLowerLcHull=
              concatenate(lowerLcHull, parent->leftchild->Ql);
    completeLowerRcHull=
              concatenate(lowerRcHull, parent->leftchild->Qr);
    parent->rightchild->Ql=completeUpperLcHull;
    parent->leftchild->Ql=completeLowerLcHull;
    parent->rightchild->Qr=completeUpperRcHull;
    parent->leftchild->Qr=completeLowerRcHull;
}
```

This procedure is called at every step as we go down the top-level tree and the node sent as a parameter is the current node.

4.5 UP Procedure

The **UP** routine is more complex than the **DOWN** routine. We decided to break it up into different procedures, all members of the `Tree` class. The procedures that go up the top-level tree updating the balance factor and calling the procedures with the rotation algorithms are `Tree::updateBalanceFactor` when an insertion has been done and `Tree::updateBalanceFactor_delete` when a deletion has been performed.

The process for updating the *lc*-hull (and *rc*-hull) was also broken into several procedures. The procedure that performs the binary search to find the points of the new bridge is `Tree::updateBridge_Hull`. This procedure calls the procedures that eliminate the upper or lower part of the *lc*-hull (and *rc*-hull) as the search continues (described in section 3.5.1 in Chapter 3). Part of the procedure that eliminates the upper part of the *lc*-hull (*rc*-hull) of the subtree stored in the left subtree of the current node is the following:

```
node *Tree::newSS_lh_right(node *current, point *q, point *q0,
point *q1, bool *onepoint, bool *out)
{
   .
   .
   .
 else if ( current->type == 2 )
 {
  x = q1->x_coord;
  y = q1->y_coord;
  if(q->y_coord == current->left->biggest_y->data2)
  {
   if (current->middle->smallest_y->parent->type == 2 )
   {
    q1->x_coord = current->middle->smallest_y->parent->middle->x_coord;
    q1->y_coord = current->middle->smallest_y->parent->middle->data2;
   }
   else
   {
    q1->x_coord = current->middle->smallest_y->parent->right->x_coord;
    q1->y_coord = current->middle->smallest_y->parent->right->data2;
   }
   q0->x_coord = q->x_coord;
   q0->y_coord = q->y_coord;
   q->x_coord = x;
   q->y_coord = y;
  }
  else if ( q->y_coord == current->middle->smallest_y->data2 )
  {
   q1->x_coord = current->right->smallest_y->x_coord;
   q1->y_coord = current->right->smallest_y->data2;
   q->x_coord = current->middle->biggest_y->x_coord;
   q->y_coord = current->middle->biggest_y->data2;
   if ( current->middle->biggest_y->parent->type == 2 )
   {
    q0->x_coord = current->middle->biggest_y->parent->middle->x_coord;
    q0->y_coord = current->middle->biggest_y->parent->middle->data2;
   }
   else
   {
    q0->x_coord = current->middle->biggest_y->parent->left->x_coord;
    q0->y_coord = current->middle->biggest_y->parent->left->data2;
   }
  }
  else
  {
   if ( current->right->smallest_y->parent->type == 2 )
   {
    q1->x_coord = current->right->smallest_y->parent->middle->x_coord;
    q1->y_coord = current->right->smallest_y->parent->middle->data2;
   }
   else
   {
    q1->x_coord = current->right->smallest_y->parent->right->x_coord;
    q1->y_coord = current->right->smallest_y->parent->right->data2;
   }
   q0->x_coord = q->x_coord;
   q0->y_coord = q->y_coord;
   q->x_coord = x;
   q->y_coord = y;
```

```
  }
 }
 else
 {
   x = q1->x_coord;
   y = q1->y_coord;
   if ( current->right->smallest_y->parent->type == 2 )
   {
    q1->x_coord = current->right->smallest_y->parent->middle->x_coord;
    q1->y_coord = current->right->smallest_y->parent->middle->data2;
   }
   else
   {
    q1->x_coord = current->right->smallest_y->parent->right->x_coord;
    q1->y_coord = current->right->smallest_y->parent->right->data2;
   }
   q0->x_coord = q->x_coord;
   q0->y_coord = q->y_coord;
   q->x_coord = x;
   q->y_coord = y;
  }
  return newcurrent;
}
```

The procedure that splits at the new bridge the *lc*-hulls (and *rc*-hulls) stored in the left and right subtrees and concatenate the corresponding parts as described in section 3.5.2 of Chapter 3 is `Tree::updateHull`.

4.6 User Options

The program that builds the executable to run our implementation of the Overmars and Van Leeuwen algorithm is in the file `updatehull.cc`. To build the executable `convexHull_OVL`, the user must compile `updatehull.cc` with the following command[1]: `make -f makefile`.

Our implementation works the following way. To start the application with the current set S empty, the user must type in the command line the name of the executable:

[1]This is the command for the Linux Operating System. The file `makefile` is also stored in the directory ~/s0671978/PROJECT/

`convexHull_OVL`. If the user wants to give a set $S = \{p_1, \ldots, p_n\}$, he must first create two text files: one with the x-values of S and another with the y-values of S. To start the implementation, the user must type `./convexHull_OVL [x-values file] [y-values file]`. The x and y-values must be one per line in the text files. After the convex hull of S is computed and displayed, the user can choose to see the structure of the top-level tree. Then, a menu is displayed with the following options:

1. Add a point.
2. Delete a point.
3. Exit.

In options 1 and 2, the user will be asked to input the x and y-coordinates of the point to be inserted or deleted. Then, the user can choose to see the structure of the top-level tree before and after the insertion or deletion of a point. Also, the user can choose to see the *lc* and *rc* hulls that are built during the **DOWN** and **UP** procedures. For the **DOWN** procedure, it will show the *lc*-hulls and *rc*-hulls of the set of points stored in the left and right subtrees of the current node. It will indicate in which level of the tree is the node (of the path) that roots the subtree that is split in two parts and *lc*-hulls (and *rc*-hulls) of each part is constructed. For the **UP** procedure, it will print the updated *lc*-hull and *rc*-hull of the nodes in the path all the way to the root.

The convex hull of the current set will be presented as the union of the *lc*-hull and *rc*-hull.

We followed the assumption of Overmars and Van Leeuwen noted in observation 2.2, that every point in the set of points S has a different y-coordinate. Also, if a new point is inserted, its y-coordinate must be different fro—m all the y-coordinates of the points in the current set.

4.6.1 Displaying the Top-Level Tree

As mentioned in the previous section, the user has the option to print out the top-level tree. We implemented this feature in a way that allows the user to see the structure of the tree. When we say we print a node, we refer to printing the label of the node if it is an inner node or, if it is a leaf node, printing the point stored in the node.

We print the nodes of the top-level tree as follows. Starting at the root, we print the node and then all the nodes in the left subtree and then we print all the nodes in the right subtrees. The deeper a node is, the more it will be indented before being printed. Nodes that are at the same level in the tree have the same indentation.

The parent of a node v is the one printed above v with the next smaller indentation.

```
-- 36
----- 15
-------- 14
----------- (32, 14)
----------- (6, 15)
-------- 28
----------- (32, 28)
----------- (68, 36)
----- 63
-------- 62
----------- (1, 62)
----------- (81, 63)
-------- 91
----------- (47, 91)
----------- (73, 94)
```

Figure 4.1: Display of a top-level tree.

The two child nodes of a node v (if they exist) are the next two nodes printed beneath v that have the next bigger indentation.

Figure 4.1 shows the output of the top-level tree when the current set is $\{(32,14)$, $(6,15)$, $(32,28)$, $(68,36)$, $(1,62)$, $(81,63)$, $(47,91)$, $(73,94)\}$. In the example shown in Figure 4.1, the parent of the node with label 28 is the node with label 15. The left and right child nodes of the node with label 36 are the nodes with labels 15 and 63 respectively.

4.7 Debugging the code

The main problem we had while doing the implementation was with $NULL$ pointers. The problem was because we were trying to call procedures or attributes of $NULL$ pointers. We overcame this problem debugging the code. To do this, we set breakpoints to detect where was this problem. Once we found where we where making the calls to attributes of $NULL$ pointers, we set the necessary conditions to avoid this calls. We had this problems while doing the procedures to balance the top-level tree and to update the labels of the inner nodes of the top-level tree after a point was deleted. We also had similar problems while implementing the operations to concatenate and split concatenable queues.

We also had difficulties while doing the binary search to find to bridge because we got into infinite loops. We had this problem because at some points we did not move to the next level of the concatenable queue that stored the *lc*-hull (and *rc*-hull). To find where we had this problem, we put break points along the procedure that implements the binary search to find where we where missing going down the concatenable queue. After finding where we were not going down, we just assigned the current node to be the proper child node of the current node.

Chapter 5

Testing the Implementation

In this chapter, we will describe the process we followed to check the correctness of our implementation. The testing phase was done in three phases: individual testing of the data structures, testing the correctness of our algorithm with small sets of points whose convex hull was known, and testing our implementation with random sets of points.

We benchmarked the results obtained with our implementation against the results obtained with an implementation of Graham's Scan [5]. The source code of this implementation was found in [6]. We modified the implementation so that we could check the running time of the execution. We also made a modification so that the input (the set of points) could be read from two text files, one with the x-coordinates and one with the y-coordinates, like in our implementation. The executable is stored on DICE in the folder in which our executable and source code are stored.

5.1 Testing the Data Structures

This phase was done after finishing the implementation of the concatenable queue and the top-level tree.

To test the concatenable queue, we used sets of random points. We generated the x and y-values of the set with the random number generator of Open Office Spreadsheet. To check the insertion operation, we started with an empty concatenable queue and insert one by one the points of the set. After each insertion, we printed out the concatenable queue and checked if the labels at the internal nodes were correct. Also, we checked if the pointers to the leaf nodes that stored the points with the smallest and biggest y-coordinates were correct in all the inner nodes of the concatenable queue. We checked the cases when the points added were to be inserted under a 2-node and under a 3-node. To check the concatenation of two concatenable queues $Q1$ and $Q2$ to form Q, we used different sizes of concatenable queues and checked all the cases described in section 3.1.3 of Chapter 3: $Q1$ and $Q2$ have the same height, $Q1$ is taller than $Q2$, and $Q2$ is taller than $Q1$. To verify the operation, we printed out Q and see if it was the expected concatenable queue.

To check the top-level tree, we also sets of points with random y-coordinates. The first thing we checked after an insertion and deletion was that the labels of the tree were updated correctly as well as the balance factor, but the tree was not still balanced. Then, after we implemented the rotation algorithms, we verified that the tree was correctly balanced and that after rebalancing a node we continued updating the labels of the tree. To check this, we printed out the nodes of the top-level tree (the output of printing the top-level tree was described in section 4.6.1 of Chapter 4).

5.2 A non-random test

In this section, we will present one of the tests we did with a set of points that were not randomly generated. We knew in advance what was the convex hull of this set. Also, we carried out a deletion, and an insertion, and we knew the convex hull of the set after these operations. We knew the convex hull of the sets, before and after the operations, by plotting the set of points with Matlab and looking at the the plot to find the convex hull. We also plotted the convex hull to check its correctness.

The initial set of points used was the following:

$$S1 = \left\{ \begin{array}{ccccccc} (13,20), & (9,4), & (6.5,6), & (10,13), & (12,5), & (5,8), & (14,14), \\ (17,12), & (13,4.5), & (18,7), & (6,18), & (16,0), & (19,7), & (5.5,11) \end{array} \right\}$$

We presented the set of points in this order. The files with the x-values and y-values of $S1$ are `ex1_xcoords` and `ex1_ycoords` respectively. The files are stored on DICE in the directory ∼/s0671978/PROJECT/, which is the same directory where our source code is stored. Recall that the Overmars and Van Leeuwen algorithm updates the convex hull of the set containing $\{(13,20)\}$ after the insertion of the point $(9,4)$, then it updates the convex hull of the set $\{(3,20),(9,4)\}$. It continues this way until the point $(5.5,11)$ is inserted in the current set. The top-level tree when $S1$ is the current set is illustrated in Figure 5.1. The inner nodes are represented with circles and show their label. The leaf nodes are represented as squares and show the point from the set that they store. The convex hull of $S1$ is represented by the following set of points: $\{(16,0),(9,4),(6.5,6),(5,8),(6,18),(13,20),\ (18,7),(19,17)\}$. Observe that this set of points is the union of the points that represent the lc-hull and rc-hull of $S1$, which are $\{(16,0),(9,4),(6.5,6),(5,8),(6,18),\ (13,20)\}$ and $\{(16,0),(18,7),(19,17),(13,20)\}$ respectively. The lc-hull and rc-hull of $S1$ are plotted in Figure 5.2. Here you can see in different types of lines the lc-hull and rc-hull of $S1$.

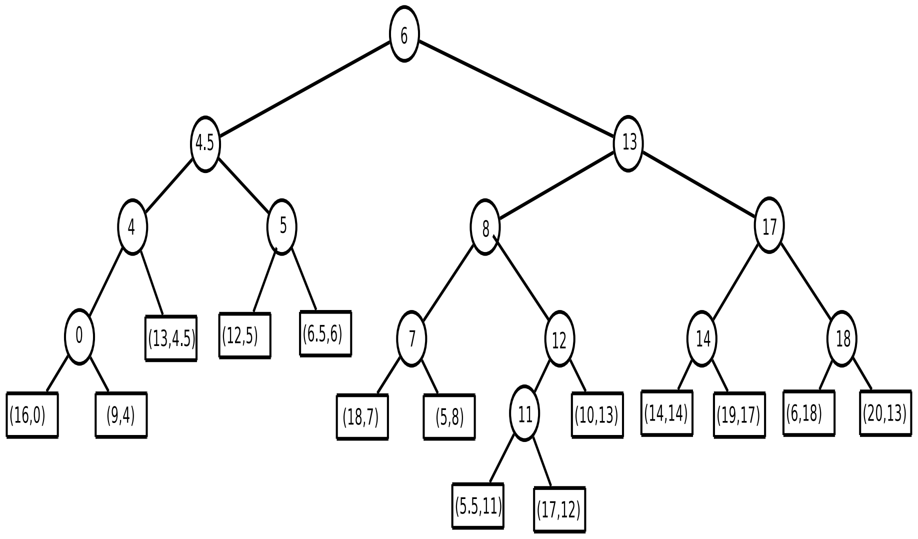


Figure 5.1: Top-level tree storing $S1$ and its *lc* and *rc*-hull.

$S1$ is our current set now. If we delete the point $p = (5, 8)$, we have to go down the tree with the procedure **DOWN**. While following the path towards the leaf that stores p, at each node, we explicitly build the lc-hulls (and rc-hulls) of the set of points stored in the left and right subtree. Figure 5.3 illustrates the lc-hulls of the set of points stored in left and right subtrees of the root. Figure 5.4 shows the lc-hulls that were built when the current node of the **DOWN** procedure was the one with label 13, and Figure 5.5 shows the lc-hulls built when the current node was the one with label 8. The dotted line in this figures, is just to show the division of the points stored in the left subtree (beneath the line) and right subtree (above the line). After deleting the leaf node that stores p, we go up the tree updating the labels and the lc-hulls (and rc-hulls), and keeping the top-level tree balanced. Figure 5.7 shows how the top-level tree looks after p was deleted. Figures 5.6 and 5.8 illustrate the updated lc-hulls of the set of points stored in the left and right subtrees of the nodes with label 13 and 6 respectively. Let $S1' = S1 \backslash \{p\}$. The points that represent the convex hull of $S1'$ are: $(16, 0)$, $(9, 4)$, $(6.5, 6)$, $(18, 7)$, $(5.5, 11)$, $(19, 17)$, $(6, 18)$, and $(13, 20)$. Figure 5.9 shows the convex hull of $S1'$ as the union of its lc and rc-hull. In this Figure, p is represented as a circle, while the points of $S1'$ are represented with crosses.

If we delete $p' = (16, 0)$ from $S1'$, the new convex hull will include an inner point of the convex hull of $S1'$. This is shown in Figure 5.10. The points that represent the new convex hull are: $(9, 4)$, $(13, 4.5)$, $(6.5, 6)$, $(18, 7)$, $(5.5, 11)$, $(19, 17)$, $(6, 18)$, and $(13, 20)$.

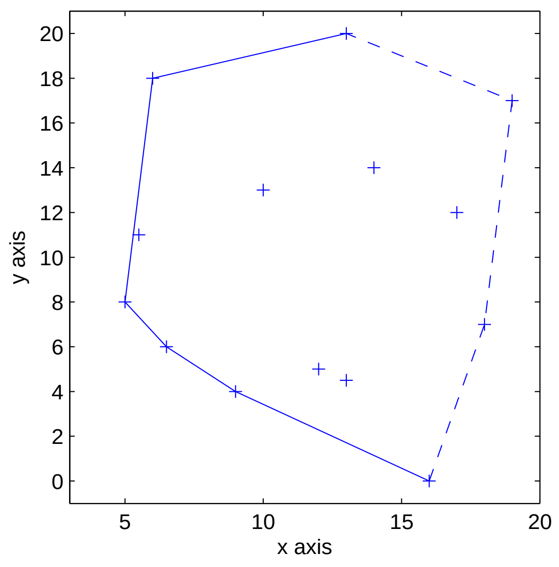


Figure 5.2: The Convex Hull of $S1$.

5.3 Tests with random set of points

In this section, we will show the results of four of the tests we ran with sets of points randomly generated. The x and y-coordinates of the points were generated using the random number generator of OpenOffice Spreadsheet. Then we tested adding new points and deleting points from the set. To check the convex hull computed with our implementation, we computed the convex hull using Graham's Scan. We worked with an implementation made by Skiena and Revilla [6], which we modified to input the set in a text file and check the running time of Graham's Scan.

The files with the x and y-coordinates of the sets of points of the following tests are stored on DICE in the same directory where our implementation is stored.

Test 1

The initial set for this test was the following:

$$S2 = \left\{ \begin{array}{lllll} (58.22, 42.88), & (30.04, 38.52), & (63.85, 11.51), & (51, 90.66), & (84.35, 77.78), \\ (61.52, 3.37), & (94.94, 27.02), & (82.14, 40.78), & (63.47, 27.79), & (7.43, 56.43), \\ (80.03, 58.72), & (9.2, 36.37), & (29.73, 53.53), & (47.09, 74.66), & (38.57, 43.34), \\ (31.25, 96.79), & (35.32, 50.07), & (72.83, 86.32), & (40.74, 57.18), & (64.1, 2.25), \\ (60.56, 59.03), & (29.28, 42.7), & (99.81, 92.75), & (70.49, 7.24), & (49.18, 50.52), \\ (65.96, 58.39), & (86.89, 95.69), & (10.92, 33, 98), & (70.35, 49.49), & (77.32, 1.63), \\ (46.29, 63.54) & & & & \end{array} \right\}$$

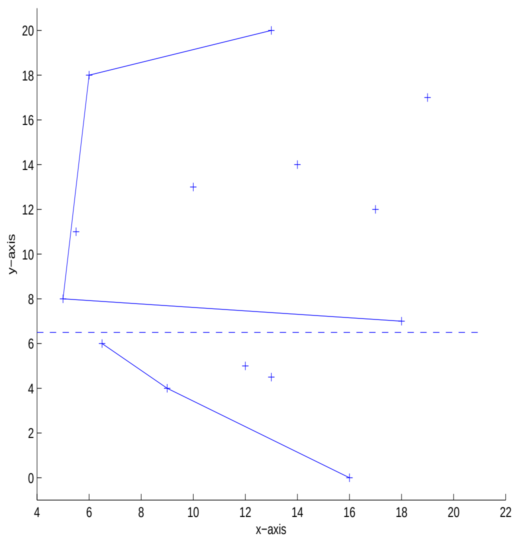


Figure 5.3: The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 4.5 and 13.

The convex hull of $S2$ is represented by the points $(7.43, 56.43)$, $(9.2, 36.37)$, $(10.92, 33.98)$, $(61.52, 3.37)$, $(64.1, 2.25)$, $(77.32, 1.63)$, $(94.94, 27.02)$, $(99.81, 92.75)$, $(86.89, 95.69)$, and $(31.28, 96.79)$. We performed the following operations on $S2$:

1. Add the point $(100, 105)$.
2. Delete the point $(94.94, 27.02)$.
3. Delete the point $(99.81, 92.75)$.

The results of performing this operations are the following:

- After operation 1, the current set is $S2^1 = S2 \cup \{(100, 105)\}$ and its convex hull is $(7.43, 56.43)$, $(9.2, 36.37)$, $(10.92, 33.98)$, $(61.52, 3.37)$, $(64.1, 2.25)$, $(77.32, 1.63)$, $(94.94, 27.02)$, $(99.81, 92.75)$, $(100, 105)$, and $(31.28, 96.79)$.
- After operation 2, the current set is $S2^2 = S2^1 \backslash \{(94.94, 27.02)\}$ and its convex hull is $(7.43, 56.43)$, $(9.2, 36.37)$, $(10.92, 33.98)$, $(61.52, 3.37)$, $(64.1, 2.25)$, $(77.32, 1.63)$, $(99.81, 92.75)$, $(100, 105)$, and $(31.28, 96.79)$.
- After operation 3, the current set is $S2^3 = S2^2 \backslash \{(99.81, 92.75)\}$ and its convex hull is $(7.43, 56.43)$, $(9.2, 36.37)$, $(10.92, 33.98)$, $(61.52, 3.37)$, $(64.1, 2.25)$, $(77.32, 1.63)$, $(100, 105)$, and $(31.28, 96.79)$.

The files with the points of the sets are the following:

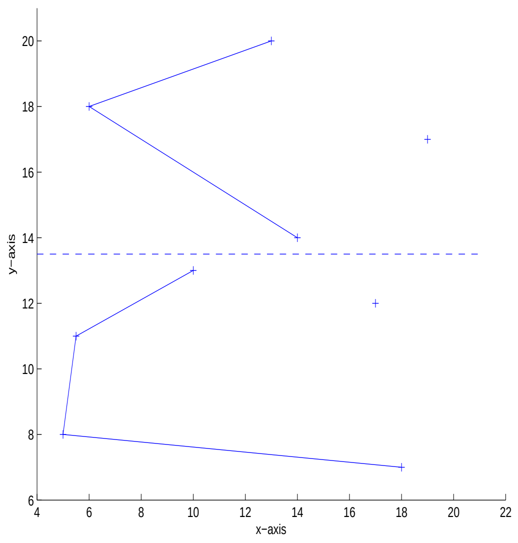


Figure 5.4: The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 8 and 17.

Set	x-coordinates file	y-coordinates file
$S2$	`ex2_xcoords`	`ex2_ycoords`
$S2^1$	`ex2_xcoords.1`	`ex2_ycoords.1`
$S2^2$	`ex2_xcoords.2`	`ex2_ycoords.2`
$S2^3$	`ex2_xcoords.3`	`ex2_ycoords.3`

The files of the sets $S2^1$, $S2^2$, and $S2^3$ where used as an input for the implementation of Graham's Scan to benchmark the results obtained with our implementation.

Test 2

The initial set for this test was the following:

$$S3 = \left\{ \begin{array}{lllll} (365,-184), & (328,255), & (-498,352), & (202,-447), & (-405,7), \\ (248,-367), & (14,-46), & (-387,-376), & (454,232), & (173,182), \\ (400,244), & (361,-470), & (401,-495), & (-344,-389), & (443,344), \\ (-238,-57), & (-327,-67), & (-112,-169), & (320,281), & (-186,115), \\ (-198,218), & (35,-226), & (-149,240), & (-420,-133), & (251,441), \\ (20,-240), & (-342,336), & (252,422), & (-218,-433), & (-100,340), \\ (117,-247), & (98,463), & (-272,95), & (120,-395), & (-201,-484), \\ (-176,-398), & (-132,-262), & (314,-30), & (-63,-274), & (-179,470), \\ (-13,-348), & (-163,471), & (-318,-500), & (-113,-342), & (-7,-419), \\ (-376,345), & (149,102), & (167,15), & (13,-325), & (-31,-118) \end{array} \right\}$$

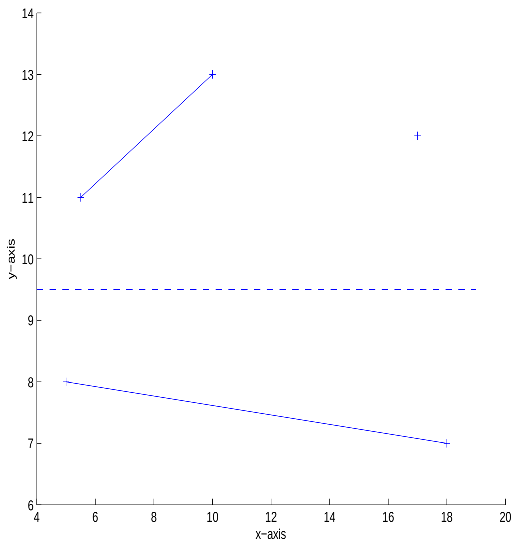


Figure 5.5: The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 7 and 12.

The convex hull of $S3$ is represented by the points $(-498, 352)$, $(-387, -376)$, $(-318, -500)$, $(401, -495)$, $(454, 232)$, $(443, 344)$, $(251, 441)$, $(98, 463)$, $(-163, 471)$, and $(-179, 470)$. We performed the following operations on $S3$:

1. Delete the point $(454, 232)$.
2. Delete the point $(-498, 352)$.
3. Add the point $(0, -510)$.
4. Add the point $(314, -372)$.

The results of performing this operations are the following:

- After operation 1, the current set is $S3^1 = S3\backslash\{(454, 232)\}$ and its convex hull is $(-498, 352)$, $(-387, -376)$, $(-318, -500)$, $(401, -495)$, $(443, 344)$, $(251, 441)$, $(98, 463)$, $(-163, 471)$, and $(-179, 470)$.
- After operation 2, the current set is $S3^2 = S3^1\backslash\{(-498, 352)\}$ and its convex hull is $(-420, -133)$, $(-387, -376)$, $(-318, -500)$, $(401, -495)$, $(443, 344)$, $(251, 441)$, $(98, 463)$, $(-163, 471)$, $(-179, 470)$, and $(-376, 345)$.
- After operation 3, the current set is $S3^3 = S3^2 \cup \{(0, -510)\}$ and its convex hull is $(-420, -133)$, $(-387, -376)$, $(-318, -500)$, $(0, -510)$, $(401, -495)$, $(443, 344)$, $(251, 441)$, $(98, 463)$, $(-163, 471)$, $(-179, 470)$, and $(-376, 345)$.

- After operation 4, the current set is $S3^4 = S3^3 \cup \{(314, -372)\}$ and its convex hull is the same as the one of $S3^3$.

The files with the points of the sets are the following:

Set	x-coordinates file	y-coordinates file
$S3$	`ex3_xcoords`	`ex3_ycoords`
$S3^1$	`ex3_xcoords.1`	`ex3_ycoords.1`
$S3^2$	`ex3_xcoords.2`	`ex3_ycoords.2`
$S3^3$	`ex3_xcoords.3`	`ex3_ycoords.3`
$S3^4$	`ex3_xcoords.4`	`ex3_ycoords.4`

The files of the sets $S3^1$, $S3^2$, $S3^3$, and $S3^4$ where used as an input for the implementation of Graham's Scan to benchmark the results obtained with our implementation.

Test 3

The initial set for this test was the following:

$$S4 = \left\{ \begin{array}{lllll}
(98,146), & (-378,-55), & (-404,-442), & (19,76), & (-310,498), \\
(437,-15), & (273,183), & (-347,433), & (31,-459), & (378,182), \\
(-331,24), & (134,-32), & (-385,441), & (140,214), & (360,-483), \\
(-415,-201), & (293,-78), & (330,257), & (-414,92), & (451,403), \\
(412,149), & (431,-84), & (53,-240), & (-74,211), & (-394,374), \\
(435,234), & (57,-87), & (500,-309), & (92,-207), & (-73,408), \\
(-240,-162), & (-311,439), & (49,-148), & (-144,-104), & (208,15), \\
(240,-150), & (-207,381), & (-20,-303), & (393,-217), & (325,423), \\
(-142,380), & (-438,307), & (-42,-110), & (-26,322), & (202,21), \\
(-183,-92), & (59,-380), & (-6,443), & (-354,-336), & (213,345), \\
(314,-372), & (-396,25), & (-498,338), & (439,-305), & (-370,346), \\
(31,-219), & (-432,-307), & (-302,-211), & (428,476), & (83,8), \\
(48,-265), & (308,-18), & (281,451), & (331,-215), & (230,13), \\
(160,-100), & (138,-328), & (-380,-17), & (-19,-465), & (-342,-310), \\
(28,-203), & (102,-295), & (-399,-223), & (193,-190), & (-186,-198), \\
(446,448), & (6,371), & (-57,-293), & (-451,-101), & (-469,-238), \\
(-55,343), & (388,9), & (470,-208), & (139,98), & (-482,-399), \\
(315,-19), & (-330,382), & (-413,-206), & (-488,271), & (-403,358), \\
(171,302), & (61,-495), & (405,-161), & (-49,253), & (-108,-210), \\
(135,353), & (-390,-351), & (-471,-37), & (256,-165), & (92,-316), \\
(-312,154) & & & &
\end{array} \right\}$$

The convex hull of $S4$ is represented by the points $(-498, 338)$, $(-482, -399)$, $(-404, -442)$, $(61, -495)$, $(360, -483)$, $(500, -309)$, $(451, 403)$, $(446, 448)$, $(428, 476)$, $(-310, 498)$, and $(-385, 441)$. We performed the following operations on $S4$:

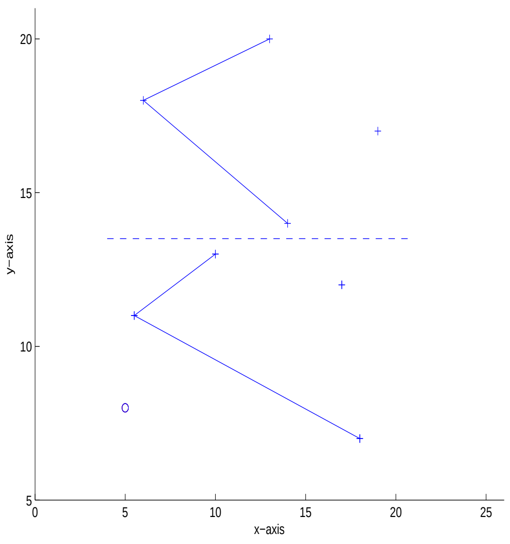


Figure 5.6: The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 11 and 17 after being updated.

1. Delete the point $(451, 403)$.

2. Delete the point $(-498, 338)$.

3. Add the point $(-550, 0)$.

The results of performing this operations are the following:

- After operation 1, the current set is $S4^1 = S4\backslash\{(451, 403)\}$ and its convex hull is $(-498, 338)$, $(-482, -399)$, $(-404, -442)$, $(61, -495)$, $(360, -483)$, $(500, -309)$, $(446, 448)$, $(428, 476)$, $(-310, 498)$, and $(-385, 441)$.

- After operation 2, the current set is $S4^2 = S4^1\backslash\{(-498, 338)\}$ and its convex hull is $(-488, 271)$, $(-482, -399)$, $(-404, -442)$, $(61, -495)$, $(360, -483)$, $(500, -309)$, $(446, 448)$, $(428, 476)$, $(-310, 498)$, and $(-385, 441)$.

- After operation 3, the current set is $S4^3 = S4^2 \cup \{(-550, 0)\}$ and its convex hull is $(-550, 0)$, $(-404, -442)$, $(61, -495)$, $(360, -483)$, $(500, -309)$, $(446, 448)$, $(428, 476)$, $(-310, 498)$, $(-385, 441)$, and $(-488, 271)$.

The files with the points of the sets are the following:

Set	x-coordinates file	y-coordinates file
$S4$	`ex4_xcoords`	`ex4_ycoords`
$S4^1$	`ex4_xcoords.1`	`ex4_ycoords.1`
$S4^2$	`ex4_xcoords.2`	`ex4_ycoords.2`
$S4^3$	`ex4_xcoords.3`	`ex4_ycoords.3`

The files of the sets $S4^1$, $S4^2$, and $S4^3$ where used as an input for the implementation of Graham's Scan to benchmark the results obtained with our implementation.

Test 4

The initial set for this test was the following:

$$S5 = \left\{ \begin{array}{lllll}
(265,-221), & (296,739), & (76,453), & (-319,-316), & (-86,746), \\
(625,484), & (-116,-368), & (-988,-702), & (-77,160), & (821,-547), \\
(-967,-79), & (-500,127), & (-648,-773), & (958,921), & (740,354), \\
(-285,303), & (809,511), & (413,667), & (-869,-43), & (-692,576), \\
(971,831), & (348,-577), & (-788,-191), & (825,-886), & (279,111), \\
(886,201), & (768,860), & (233,109), & (-307,-201), & (-734,324), \\
(409,-869), & (990,-463), & (343,-224), & (-911,-184), & (-952,169), \\
(-743,-478), & (714,-701), & (-731,-179), & (-363,460), & (-246,-746), \\
(-697,742), & (138,-413), & (106,-518), & (-740,663), & (-123,942), \\
(822,785), & (-931,173), & (-710,608), & (953,-259), & (-623,-252), \\
(-740,438), & (301,165), & (-410,558), & (-915,552), & (-421,-724), \\
(-525,-242), & (813,386), & (-881,-265), & (221,518), & (158,-906), \\
(462,512), & (311,-667), & (207,264), & (719,-967), & (25,-367), \\
(-860,65), & (989,-146), & (662,-907), & (-106,320), & (-708,-404), \\
(-201,-320), & (1000,802), & (-447,-742), & (677,-378), & (822,586), \\
(-378,431), & (967,-771), & (775,-673), & (0,-821), & (-773,668), \\
(75,492), & (590,737), & (-688,220), & (655,769), & (-935,-505), \\
(-836,-369), & (467,154), & (234,-947), & (-717,366), & (-313,-329), \\
(-609,-853), & (746,-122), & (999,5), & (599,412), & (464,-89), \\
(23,638) & & & &
\end{array} \right\}$$

The convex hull of $S4$ is represented by the points $(-988,-702)$, $(-609,-853)$, $(234,-947)$, $(719,-967)$, $(825,-886)$, $(967,-771)$, $(990,-463)$, $(999,5)$, $(1000,802)$, $(958,921)$, $(-123,942)$, $(-697,742)$, $(-915,552)$, $(-952,169)$, and $(-967,-79)$. We performed the following operations on $S5$:

1. Delete the point $(958,921)$.
2. Delete the point $(1000,802)$.
3. Add the point $(-900,500)$.

The results of performing this operations are the following:

- After operation 1, the current set is $S5^1 = S5 \setminus \{(958, 921)\}$ and its convex hull is $(-988, -702)$, $(-609, -853)$, $(234, -947)$, $(719, -967)$, $(825, -886)$, $(967, -771)$, $(990, -463)$, $(999, 5)$, $(1000, 802)$, $(971, 831)$, $(768, 860)$, $(-123, 942)$, $(-697, 742)$, $(-915, 552)$, $(-952, 169)$, and $(-967, -79)$.

- After operation 2, the current set is $S5^2 = S5^1 \setminus \{(1000, 802)\}$ and its convex hull is $(-988, -702)$, $(-609, -853)$, $(234, -947)$, $(719, -967)$, $(825, -886)$, $(967, -771)$, $(990, -463)$, $(999, 5)$, $(971, 831)$, $(768, 860)$, $(-123, 942)$, $(-697, 742)$, $(-915, 552)$, $(-952, 169)$, and $(-967, -79)$.

- After operation 3, the current set is $S5^3 = S5^2 \cup \{(-550, 0)\}$ and its convex hull is the same as the one of $S5^2$.

The files with the points of the sets are the following:

Set	x-coordinates file	y-coordinates file
$S5$	`ex5_xcoords`	`ex5_ycoords`
$S5^1$	`ex5_xcoords.1`	`ex5_ycoords.1`
$S5^2$	`ex5_xcoords.2`	`ex5_ycoords.2`
$S5^3$	`ex5_xcoords.3`	`ex5_ycoords.3`

The files of the sets $S5^1$, $S5^2$, and $S5^3$ where used as an input for the implementation of Graham's Scan to benchmark the results obtained with our implementation.

5.3.1 Comparing Running Times

We measured in microseconds, using the C++ library `time.h`, the time our implementation of the Overmars and Van Leeuwen algorithm took to construct the convex hull hull of the initial set. We also timed the insertion of a point into the current set and the deletion of a point from the current set (including maintaining the convex hull).

The following table shows the times we obtained while doing the tests presented in section 5.2. It presents the time (in microseconds) it took to update the convex hull of the sets (of the tests presented in the previous section) with our implementation and the time it took the implementation of Graham's Scan[5] to recompute it.

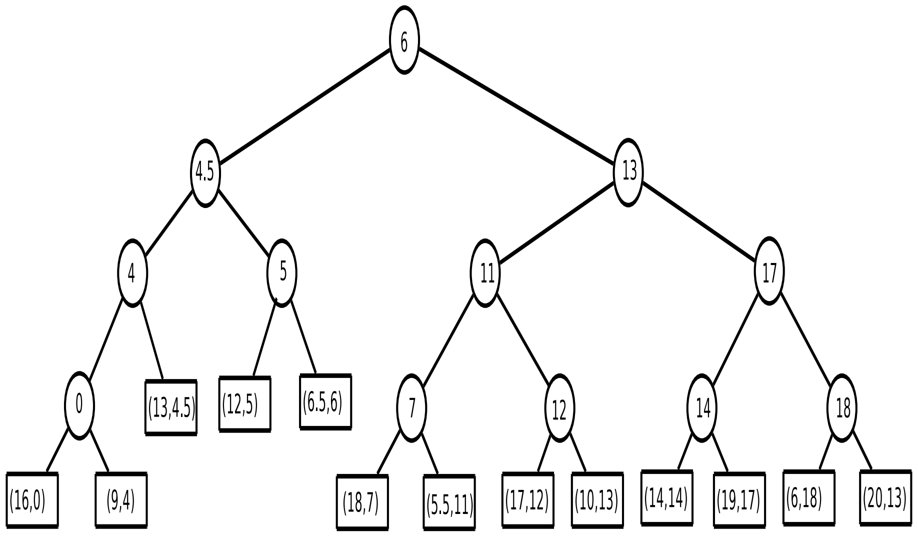


Figure 5.7: Top-level tree after deleting the point $(5, 8)$ from $S1$.

Set	Number of elements of the set	Overmars and Van Leeuwen	Graham's Scan
$S2$	31	2460	41
$S2^1$	32	70	41
$S2^2$	31	81	40
$S2^3$	30	80	39
$S3$	50	5013	70
$S3^1$	48	63	66
$S3^2$	48	68	64
$S3^3$	49	71	67
$S3^4$	50	60	69
$S4$	101	10412	256
$S4^1$	100	66	251
$S4^2$	99	77	250
$S4^3$	100	79	253
$S5$	96	9609	265
$S5^1$	95	83	233
$S5^2$	94	64	231
$S5^3$	95	79	268

As expected from the running time analysis made in section 2.6 of Chapter 2, Graham's Scan was faster to compute the convex hull of the initial set of points. As sets got bigger, Overmars and Van Leeuwen's algorithm was faster to maintain the convex hull than recomputing is with Graham's Scan.

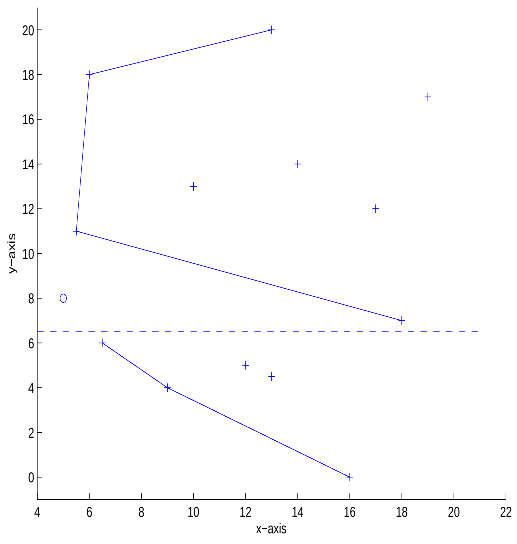


Figure 5.8: The *lc*-hulls of the sets of points stored in the subtrees rooted at the nodes with label 4.5 and 13 after being updated.

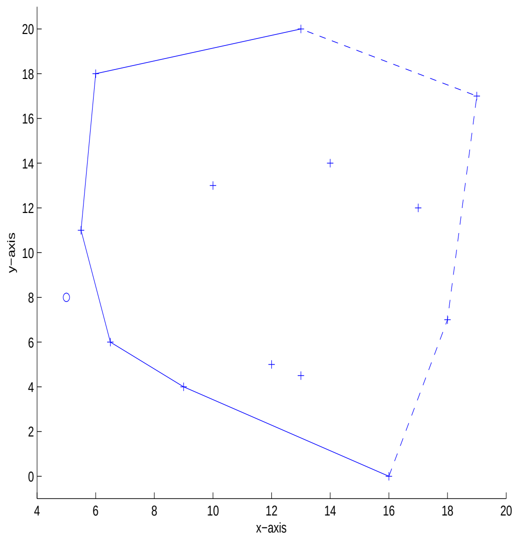


Figure 5.9: The Convex Hull of $S1'$.

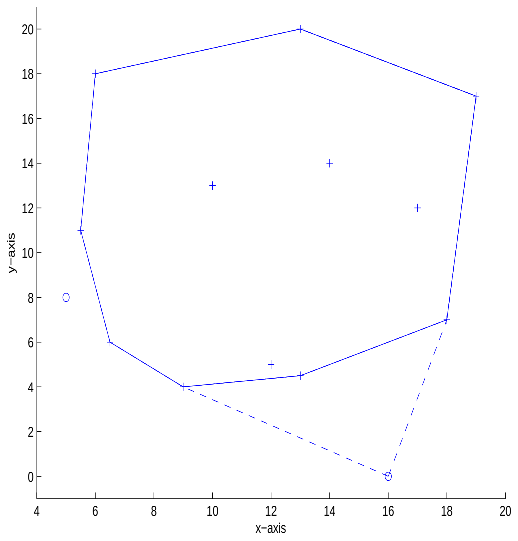


Figure 5.10: The Convex Hull of $S1'$ and the Convex Hull after deleting p' from $S1'$.

Chapter 6

Conclusions

The convex hull problem has been widely studied. Overmars and Van Leeuwen were the first to propose a dynamic algorithm to solve this problem. By a dynamic algorithm we mean an algorithm than can maintain the convex hull of a set after the insertion of a point into the set or the deletion of a point from the set, without recomputing the convex hull from scratch. In Overmars and Van Leeuwen [2], the authors describe data structures using a very high level description. We had to work out the details Overmars and Van Leeuwen omit in their paper to do the implementation from scratch.

We were able to implement the Overmars and Van Leeuwen algorithm so that it handles insertions and deletions in $O(\log^2 n)$ time, where n is the number of points in the set. We were able to see (without using very large sets of points) that updating the convex hull with the Overmars and Van Leeuwen algorithm is faster than with a naive approach (recomputing the convex hull with Graham's Scan each time the set changes).

It might be worth attempting to construct an implementation that uses an algorithm that computes the convex hull of a set in $O(n \log n)$ time to compute the initial set of points and then handle the insertions and deletions with Overmars and Van Leeuwen's algorithm in $O(\log^2 n)$ time, where n is the number of points in the set. This would allow the algorithm to compute the convex hull of the initial set more quickly, and also to be able to maintain it dynamically. Making this extension might be interesting because the implementation will have to be able to provide the recursive structures Overmars and Van Leeuwen maintain after computing the convex hull of the initial set.

The Overmars and Van Leeuwen algorithm is exclusively for maintaining the convex hull of dynamic sets in 2D. Their idea of maintaining the left and the right side of the convex hull by means of recursive structures cannot be used in a higher dimension. The reason is because in a higher dimension there is no notion of left and right as in 2D. If time could have permitted, we would have tried to study the case of dynamic sets in a higher dimension. Maintaining the convex hull of a set in a higher dimension than 2D with a dynamic algorithm is still an open problem. No dynamic algorithm has been proposed.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, first edition, 1974.

[2] M.H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166 – 204, 1981/10/.

[3] F.P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22(7):402 – 5, 1979/07/.

[4] F.P. Preparata and M.I. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, first edition, 1988.

[5] S. Skiena and M Revilla. *Programming Challenges: The Programming Contest Training Manual*. Springer-Verlag, first edition, 2003.

[6] S. Skiena and M Revilla. Programs for programming challenges. Webpage, 2003. `http://www.cs.sunysb.edu/∼skiena/392/programs/`.

[7] P. van Emde Boas. On the omega(n log n) lower bound for convex hull and maximal vector determination. *Inf. Process. Lett.*, 10(3):132–136, 1980.