

---

# 2D Orthogonal Range Search

Mads Ravn, 20071580

---

Master's Thesis, Computer Science

February 2015

Advisor: Kasper Green Larsen



AARHUS  
UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE



# Abstract

► in English... ◄



# Resumé

► in Danish . . . ◄



# Acknowledgements




*Mads Ravn,  
Aarhus, February 4, 2015.*





# Contents

Abstract	iii
Resumé	v
Acknowledgments	vii
1 Introduction	3
I Theory	5
2 Related Work	7
2.1 kd-trees . . . . .	7
2.2 Simplified Range Searching . . . . .	8
2.2.1 Preliminaries . . . . .	8
2.2.2 Solving the ball-inheritance problem . . . . .	9
2.2.3 Solving range reporting . . . . .	10
2.3 Orthogonal Range Searching . . . . .	12
2.3.1 Preliminaries . . . . .	12
2.3.2 Solving range reporting . . . . .	12
3  . . . . .	17
4 Conclusion	19
Primary Bibliography	19



# Chapter 1

## Introduction

►**Describe primary work**◄ In this thesis we are going to explore data structure we believe to be faster at orthogonal range search than a kd-tree. We are going to introduce the a range reporting data structure by Chan et al. [2]. A simplified model of their data structure will be made and implemented. We wish to show that this simplified model is able to compete with a kd-tree.

**Orthogonal Range Searching.** *Orthogonal range searching* is one of the most fundamental and well-studied problems in computational geometry. Even with extensive research over three decades a lot of questions remain. In this thesis we will focus on  $2D$  orthogonal range searching: Given  $n$  points from  $\mathbb{R}^2$  we want to insert them into a data structure which will be able to efficiently report which  $k$  points lie within a given query range  $Q \subseteq \mathbb{R}^2$ . This query can be defined as two corners of a rectangle, the lower left corner and the upper right corner, seeing as the query range is orthogonal to the axes.

**RAM, I/O, pointer models** ►**Skriv om Word RAM Model sammenlignet med de andre typer her. Når vi pakker bits ned i et word kan vi stadig tilgå det hele da access er constant time og operations er constant time**◄

**Rank Space Reduction.** Given  $n$  points from a universe  $U$ , the rank of a given point is defined as the amount of points which precede it in a sorted list. Given two points  $a, b \in U : a < b$  iff  $rank(a) < rank(b)$ . Expanding this concept to 2 dimensions we have a set  $P$  of  $n$  points on a  $U \times U$  grid. We compute for the  $x$ -rank  $r_x$  for each point in  $P$  by finding the rank of the x-coordinate amongst all the x-coordinates in  $P$ . The  $y$ -rank  $r_y$  finds the rank of y-coordinate amongst all of the y-coordinates in  $P$ . Using *rank space reduction* on  $P$  a new set  $P^*$  is constructed where  $(x, y) \in P$  is replaced by  $(r_x(x), r_y(y)) \in P^*$ . Given a range query  $q = [x_1, x_2] \times [y_1, y_2]$ , a point  $(x, y) \in P$  is found within  $q$  iff  $(r_x(x), r_y(y))$  is found within  $q^* = [r_x(x_1), r_x(x_2)] \times [r_y(y_1), r_y(y_2)]$ . ►**Noget med at deres ordered property remains intact.**◄ Computing the set  $P^*$  from  $P$  using rank space reduction,  $P^*$  is said to be in rank space. While the  $n$  points could be represented by  $\lg U$  bits in  $P$ , they can now be represented by

$\lg n$  bits in  $P^*$  with  $\lg n \ll \lg U$  which saves memory. ► **We have essentially created a mapping between ...** ◀

**Ball Inheritance.** Given a perfect binary tree with  $n$  leaves and  $n$  labelled balls at the root the idea is to distribute the balls from the root to the leaves. The balls at the root are contained in an ordered list and for each ball in a nodes list one of its children is picked to inherit the ball such that both children receive the same amount of balls. The level of a node is defined to be the height of the node. The root has the highest level, while each node is one level smaller than its parent. The leaves are at level 0. Each level of the tree contains the same amount of balls, and at level  $i$  each node contains  $2^i$  balls. Eventually each ball reaches a leaf of the tree and each leaf will contain exactly one ball. The identity of a given ball is a node and the index of the ball in that nodes list. Given the identity of a ball at any level, it is possible to follow this identity to its actual coordinates. The goal is to track a balls inheritance from a given node to a leaf and report the identity of the leaf. We call the identity of the leaf, where the actual coordinates reside, the *true identity* of a ball. When referring to ball inheritance, the words *ball* or *point* can be used interchangeably.

**Outline.** Has yet to be written.

**Notation.** The set of integers  $\{i, i + 1, \dots, j - 1, j\}$  is denoted by  $[i, j]$ . When no base is explicitly given logarithm will have base 2.  $\epsilon$  is an arbitrary small constant. Given an array  $A$ ,  $A[i]$  denotes the entry with index  $i$  in  $A$  and  $A[i, j]$  denotes the subarray containing the entries from  $i$  to  $j$  in  $A$ .  $A[1..n]$  denotes an array  $A$  of size  $n$  with entries 1 to  $n$ . Throughout the thesis the successor of  $x$  will be meant as the first number which is greater or equal to  $x$  - the same applies for predecessor of  $x$ . The work will be done under the assumption that no two points will have the same x-coordinate and no two points will have the same y-coordinate. This is a unrealistic assumption in practice, but it can easily be remedied by having the points lie in a *composite-number space* since we only need a total ordering of our points.

**Part I**

**Theory**



## Chapter 2

# Related Work

In this chapter a simplification of the work done by Chan et al. [2] is presented followed by their original idea. The simplification is the primary work of this thesis. Conceptually, the original structure by Chan et al. [2] can be thought of as an extension of the simplified data structure. This way the reader will introduced to the concepts at an incremental level.►rephrase◄ The theory behind a  $kD$ -tree will also be explained as it is the de facto standard of orthogonal range queries today and will be used to compare the practical results of the simplified model. ►rewrite◄

The simplified model has a time complexity of  $\mathcal{O}(\lg n)$  which is greater than the time complexity of the original model with  $\mathcal{O}(\lg \lg n)$ . However, the simplified model is far more simple - both in code and the data structures used. Because there is much more internal communication between the data structures in the original model and much more code to be executed, it is safe to assume that the running time constant of the original model far exceeds that of the simplified model, making the simplified model faster than the original.

►rephrase◄

Some sections will start with a *preliminaries* subsection. This subsection will describe some of the auxiliary data structures used in the section. It will make the reader acquainted with the data structures when they are referenced instead of having them explained in the middle of another explanation.

The main algorithms►data structures?◄ mentioned in this chapter are all *output-sensitive*, meaning that their running time depends on the amount of results found.

## 2.1 kd-trees

The current standard of range reporting is kd-trees. This technique will be used as a reference point when evaluating the results of the primary work in this thesis. With linear space it is an ideal data structure for range reporting on the RAM, and a practical solution.

A kd-tree is constructed recursively: Given  $n$  points, the median of the points with respect to  $x$  are found. All points which has an  $x$ -coordinate larger than the median goes to the right child, while points which has an  $x$ -coordinate

smaller than the median, and the median point, goes to the left child. At the next level the points will be divided in a similar fashion, this time using the y-median and the y-coordinates instead. When dividing  $n$  points, the median will be chosen as the  $\lceil n/2 \rceil$ -th smallest number [1]. A node will thusly contain the line dividing the points given to its left child from the points given to its right child. Alternating between focussing on the x-coordinates or the y-coordinates at each level, the points are divided until only one point remains in a node. This node will then be a leaf containing that point. Thus, we end up with  $n$  leaves. This data structure takes up  $\mathcal{O}(n)$  space.

In order to search in this tree, we introduce the term *region*. A region of a node is the area of which its point lie within. The root contains all points and thusly has the biggest region. Since each node contains a line dividing its points between both of its children, we can use this line to narrow the region of both children. Doing this halves the amount of points lying within the region. Now given a search query  $q = [x_1, x_2] \times [y_1, y_2]$  and a kd-tree one of three things can happen. The region of a node contain can be fully contained in the search query, in which case the entire node and its points are returned as part of the result. The region of a node and the search query can overlap in which case we continue the search down both of the children of the node. Finally the node of a node and the search query has nothing in common in which case the search stops. If a leaf is visited in the search, the point of the leaf is reported as part of the result if it lies within the search query. Searching the kd-tree takes  $\mathcal{O}(\sqrt{n} + k)$  time to give  $k$  points as a result.

The tree with  $n$  points can be represented as flat array with  $n$  entries. The  $\lceil n/2 \rceil$ -th element in the array is the root of the tree. **►uddyb◄**

## 2.2 Simplified Range Searching

This section will introduce the primary work of this thesis. It will show how the data structure is built and how range reporting is done using the data structure. Succinct rank queries is an important part of this chapter, playing a key role in solving the ball-inheritance problem.

### 2.2.1 Preliminaries

#### Predecessor search using binary search

In order to find the rank space successor or rank space predecessor of a point, a binary search is used on a sorted array of points. This data structure uses  $\mathcal{O}(n)$  space and have a query time of  $\mathcal{O}(\lg n)$ . By locating the first key in the array that is bigger than the search query, the index of that key is the rank space successor. Similarly, by locating the last key that is smaller in the array, the index of that key is the ranks space predecessor.



### Succinct rank queries

Consider an array  $A[1..n]$  with elements from some alphabet  $\Sigma$ . Given an index  $i$  in the array, we can find out how many elements in  $A[1..i]$  are equal to  $A[i]$ . This is called a *rank query*. We want to be able to answer the rank query in constant time using a data structure of  $\mathcal{O}(n \lg \Sigma)$  bits. In order to do this checkpoints are created. For each character in the alphabet  $\Sigma$ , the checkpoint contains the number of times that character appears in  $A[1..i]$ , where  $i$  is the checkpoint location. Such a checkpoint takes up  $\mathcal{O}(\Sigma \lg n)$  space. By placing each checkpoint  $\Sigma \lg n$  entries apart of each other, all of the checkpoints use  $\mathcal{O}(\frac{n}{\Sigma \lg n} \cdot \Sigma \lg n) = \mathcal{O}(n)$  space.

Furthermore, for each entry in the array  $A$ , a smaller sum will be stored. Each entry in  $A$  contains a character from the alphabet  $\Sigma$ . At  $A[i]$  we store the amount of times the character at  $A[i]$  appears since the last checkpoint. This is a smaller number and can be stored using  $\mathcal{O}(\lg(\Sigma \lg n))$  bits per entry in  $A$ . This is because we only need  $\lg x$  bits to store a number which has a maximum value of  $x - 1$ . Since there is only  $\Sigma \lg n$  entries between each checkpoint, this adds up. This approach fits the required space bound if  $\Sigma \geq \sqrt{\lg n}$ , because there  $\Sigma$  will dominate the complexity.

For smaller alphabets, another scheme is used. Checkpoint are still stored at every  $\Sigma \lg n$  positions. In addition to this, minor checkpoints are added. These minor checkpoints are added at every  $\lg \lg n$  positions and contain the amount of times each character is seen since the last major checkpoint. These minor checkpoints take up  $\mathcal{O}(\Sigma \lg \lg n)$  bits each. In order to answer the rank query, a query to the last major checkpoint and the last minor checkpoint has to be made. Given that  $\Sigma \lg \lg n \leq \sqrt{\lg n} \cdot \lg \lg n$ , the array entries holding the amount of times  $A[i]$  is seen since the last minor checkpoint fits into  $\mathcal{O}(\sqrt{\lg n} \cdot \lg^2 \lg n)$  bits. Therefore it is possible to store these array entries in plain form, and with the help of a precomputed table of  $n^{\mathcal{O}(1)}$  space we can answer rank queries between minor checkpoints in constant time. **►Plain form?◄ ►rephrase◄**

### 2.2.2 Solving the ball-inheritance problem

Consider a perfect binary tree with  $n$  leaves. At each level a bit vector  $A[1..n]$  is used to indicate which of a node's children a ball is inherited by: If  $A[i]$  is 0 means that the ball with identity  $i$  in that node was inherited by its left child and 1 means that it was inherited by its right child. Given a node and an identity of a ball we can know calculate the ball's identity in the node it is inherited by. The node can answer the query  $rank(k) = \sum_{i \leq k} A[i]$ . If a ball is inherited by the right child node its new identity at that node is  $rank(i)$  because that is how many 1's that precede it in the current node. If a ball is inherited by the left child node the new identity is then  $i - rank(i)$ . With this information it is possible to traverse down the tree following a ball from any given node to a leaf. There are  $n$  balls per level which is represented by a bit vector of size  $n$  bits per level. Even though conceptually this bit vector is divided out amongst the nodes of that level, we can interchangeably think of it as a bit vector per

level or a bit vector per node. **►Rephrase and replace◄** Each level in the tree uses  $\mathcal{O}(n)$  bits to store the bit vectors. This adds up to  $\mathcal{O}(n \lg n)$  bits, or  $\mathcal{O}(n)$  words in all. This trivial solution to the ball-inheritance problem uses  $\mathcal{O}(\lg n)$  query time, given that it follows a ball  $\mathcal{O}(\lg n)$  steps down to its leaf. The rank function is a constant time query. **►henvis til prelim◄**.

A bit vector is an array with entries from the alphabet  $\Sigma = \{0, 1\}$ , where each entry is used to indicate whether a left or right child has been chosen to inherit a given ball. By expanding the alphabet we can point to the childrens children,  $\Sigma = \{0, 1, 2, 3\}$ , the childrens childrens children,  $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$ , and so forth. Expanding the alphabet will use  $\mathcal{O}(n \lg \Sigma)$  bits per level. Storing a pointer from level  $i$  to level  $i + \Delta$  increases the storage space by  $\Delta$  bits per ball, but also enables the ball to be inherited by  $2^\Delta$  descendants. By expanding the alphabet the query time can be lowered since it is possible to take bigger steps down the tree.

Using this concept, we pick  $B$  such that  $2 \leq B \leq m$ , where  $m$  is the height of the tree. All levels that are a multiple of  $B^i$  expand their alphabet such that the balls reach  $B^i$  levels down. If a target level does not exist, the ball points to its leaf. We need at most visit  $B$  levels that are multiple of  $B^i$  before reaching a level that is multiple of  $B^{i+1}$ , making it possible to jump down the tree with bigger and bigger steps.

Storing the expanded alphabets at each level that is multiple of  $B^i$  costs  $B^i$  bits per ball. The total cost is then  $\sum_i^{\lg_B \lg n} \lg n \cdot \frac{\lg n}{B^i} \cdot \mathcal{O}(B^i) = \mathcal{O}(\lg n \cdot \lg_B \lg n)$  bits per ball, at all levels. With  $n$  balls, this is  $\mathcal{O}(n \lg_B \lg n)$  words of space with query time of  $\mathcal{O}(B \lg_B \lg n)$ . **►Det er fordi  $\lg_B \lg n$  er det største  $i$  som beskriver  $B^i \leq \lg n$  ◄**

### 2.2.3 Solving range reporting

Consider a perfect binary tree with  $n$  leaves. The root contains  $n$  points in 2-d rank space. These  $n$  balls are sorted by their y-rank. When distributing the balls for inheritance, a node will give both its children half of its balls: the lower half sorted by the x-rank to its left child and the upper half by x-rank to its right child. The order of the balls in a child node will be the same as the parent node. **►rephrase◄** The actual coordinates of the balls are only stored at the leaves and we know that x-coordinates of the balls in the leaves are sorted from left to right - smallest to highest. Since the actual coordinate points are only stored once, this data structure uses linear space.

Given a range query  $q = [x_1, x_2] \times [y_1, y_2]$  the rank successors of  $x_1$  and  $y_1$  and the rank predecessors of  $x_2$  and  $y_2$  are looked up. We know from section **►ref◄** that a range query can be translated to a rank space query. We call these  $\hat{x}_1, \hat{y}_1, \hat{x}_2$  and  $\hat{y}_2$ . We use  $\hat{x}_1$  and  $\hat{x}_2$  to find the lowest common ancestor,  $LCA(\hat{x}_1, \hat{x}_2)$ , containing all the points between  $x_1$  and  $x_2$ .

In the root we mark the position of  $\hat{y}_1$  and  $\hat{y}_2$  on the bit vector used to indicate which descendant a balls goes to. This range indicates which balls lie

within the range of  $[y_1, y_2]$ . Going forward we will use  $i$  and  $j$  to indicate this range in the bit vector of any given node. When a node inherits this range from a parent, the rank function is used to query how many points before  $i$  went to this node, and how many points before  $j$  went to this node. We refer to section 2.2.2 **►refer to corret place◄** on how to update the positions using the rank query. This works because the points in a node are sorted by their y-rank and we keep track of how many points in a given node falls within the range of  $[y_1, y_2]$  updating the range of  $[i, j]$  everytime a node inherits balls. **►Skriv afsnit om◄**

We know the positions of the leaves containing  $\hat{x}_1$  and  $\hat{x}_2$  so we can traverse from the *LCA* down to each of them. Traversing to  $\hat{x}_1$ , the first stop is the left child of the *LCA*. From here, each time a node selects its left child as the path to  $\hat{x}_1$  we know that the subtree contained in the right child contains points between  $x_1$  and  $x_2$ . Symmetrically, the same applies when going right from the *LCA*: Each time a node selects a right child on the path to  $\hat{x}_2$  the subtree contained in the left child contains points between  $x_1$  and  $x_2$ . This concept is seen on figure 2.2.

While keeping the y-rank range updated, we travel from the root to the *LCA*. From here we travel down to both  $\hat{x}_1$  and  $\hat{x}_2$ . Each time a subtree between  $\hat{x}_1$  and  $\hat{x}_2$  is visited, we know that it is fully included on the x-dimension. We can then look in its bit vector range to determine which of the balls referred to from this node is included in the y-dimension. For all the balls included in the y-dimension, we use the ball-inheritance structure to look up their actual coordinates in  $\mathcal{O}(\lg^\epsilon n)$  time.

The data structure is utilized as follows:

1. Use a binary search to find the rank space predecessors and successors of  $x_1, y_1, x_2$  and  $y_2$ . At this point the algorithm will terminate if either rank space range is empty.
2. From  $LCA(\hat{x}_1, \hat{x}_2)$ , both  $\hat{x}_1$  and  $\hat{x}_2$  will be visited. The range y-rank range found in step 1 will be updated from the root to this *LCA* and updated from the *LCA* to  $\hat{x}_1, \hat{x}_2$  and all the subtrees between them.
3. Each time a fully included subtree is visited, we determine which balls falls within the y-range and use the ball-inheritance structure to travel to its leaf. When a leaf is visited, its actual coordinates will be reported back as a result.

The actual coordinates of the points are only stored at the leaves which then takes up  $\mathcal{O}(n)$  words of space. The rest of the tree contains  $\lg n$  levels of bit vectors of  $n$  bits taking  $\mathcal{O}(n \lg n)$  bits,  $\mathcal{O}(n)$  words. Looking up the rank-space predecessor and successor of  $x_1, x_2, y_1$  and  $y_2$  using a simple binary search at the root requires  $\mathcal{O}(n)$  space and  $\mathcal{O}(\lg n)$  time. Summing it up, the entire data structure uses  $\mathcal{O}(n)$  words of space.

Walking from the root to the *LCA* requires  $\mathcal{O}(\lg n)$  steps. Visiting  $\hat{x}_1$  and  $\hat{x}_2$  requires  $\mathcal{O}(\lg n)$  steps each. Visiting each of the  $k$  leaves in the subtrees

between  $\hat{x}_1$  and  $\hat{x}_2$ , containing the points which will be reported as a result, takes  $\mathcal{O}(k \cdot \lg^\epsilon n)$  time.

This adds up to  $\mathcal{O}(\lg n + k \cdot \lg^\epsilon n)$  query time to report  $k$  points as results. An empty range will be detected by the binary search at the root. If the binary search does not report an empty range, we proceed with the search. **►Forklar hvordan  $\lg^\epsilon n$  opstår fra “solving ball-inheritance”◄**

## 2.3 Orthogonal Range Searching

Utilizing the ball-inheritance structure, Chan et al. [2] propose a better solution for orthogonal range search queries. With the same space complexity as the  $kD$ -tree, but lower query time. Theorem 2.1 states that *for any  $2 \leq B \leq \lg^\epsilon n$ , we can solve 2-d orthogonal range reporting in rank space with  $\mathcal{O}(n \lg_B \lg n)$  space and  $(1+k)\mathcal{O}(B \lg \lg n)$  query time.*

In this section some supporting data structures will be introduced. Then we will show how the ball-inheritance is used in conjunction with these data structures to find the points within a search query  $q = [x_1, x_2] \times [y_1, y_2]$ .

### 2.3.1 Preliminaries

#### Range minimum queries

In order to find the smallest element in a range, a succinct data structure will be used. This data structure can solve the *range minimum query* problem and will be referred to as RMQ. Consider an array  $A$  with  $n$  comparable keys, this succinct data structure allows finding the index of the minimum key in the subarray  $A[i, j]$ . Fischer [3] introduces a data structure which solves this problem in  $2n + \mathcal{O}(n)$  space with constant query time. The construction requires that the array is ordered, which we will see fits into our scheme.

#### Rank space predecessor search

In order to look up the rank space predecessor of a given coordinate, another succinct data structure will be used. This data structure has a smaller space complexity than the RMQ, but has a bigger query time. Given a sorted array  $A[1..n]$  of  $\omega$ -bit integers, predecessor search queries in  $\mathcal{O}(\lg \omega)$  time is supported using  $\mathcal{O}(n \lg \omega)$  space which has oracle access to the entries in the array. **►reference◄**. The points in rank space of  $\mathcal{O}(\lg n)$  bits will use  $\mathcal{O}(n \lg \lg n)$  bits per level, with  $\omega = \lg n$ , and  $\mathcal{O}(n \lg n \lg \lg n)$  bits in all, which is  $\mathcal{O}(n \lg \lg n)$  words.

### 2.3.2 Solving range reporting

With a solution to the ball-inheritance problem, Chan et al. [2] proposes **Lemma 2.4** *if the ball inheritance problem can be solved with space  $S$  and query time  $\tau$ , 2-d range reporting can be solved with space  $\mathcal{O}(S + n)$  and query time  $\mathcal{O}(\lg \lg n + (1+k)\tau)$ .*

The ball distribution scheme of this data structure is the same as the simplified range search of section 2.2.2. Having distributed the  $n$  points from the root to the leaves, additional data structures are required in order to answer the range queries. For each node in the tree that is a right child a range minimum query structure is added. The indices are the y-rank and the keys are the x-rank that the given node contains. A range maximum query structure is added to all the nodes which are left children. Each data structure uses  $2n + \mathcal{O}(n)$  bits, making it  $\mathcal{O}(n)$  bits per level of the tree and  $\mathcal{O}(n \lg n)$  bits in all - i.e.  $\mathcal{O}(n)$  words of space.

There is still the matter of adding predecessor (and successor) search for the y-rank. For this purpose the rank space predecessor search is added to our tree. This data structure works on an array of the y-ranks, which is already sorted, while using  $\mathcal{O}(\lg \lg n)$  words. In order to reduce this to linear space we will only place this predecessor search structure at levels which are multiples of  $\lg \lg n$ . When using the predecessor search from the lowest common ancestor of  $\hat{x}_1$  and  $\hat{x}_2$ ,  $LCA(\hat{x}_1, \hat{x}_2)$ , we go up to the closest ancestor to use its search structure. Searching takes  $\mathcal{O}(\lg \lg n)$  time plus  $\mathcal{O}(1)$  queries to the ball-inheritance structure: using the ball-inheritance structure we walk at most  $\lg \lg n$  steps down while translating the ranks of  $y_1$  and  $y_2$  to the right and left child of  $LCA(\hat{x}_1, \hat{x}_2)$ .

The reason why this structure is necessary for the y-ranks and not the x-ranks, is because of the way the points have been distributed in the ball-inheritance tree: From left to right, the leaves have x-rank  $1, 2, \dots, n$  so we can easily locate a given range in x-dimension, but in order to keep track of the y-dimensional range we need to follow the balls down the ball-inheritance structure. Adding this structure to each  $\lg \lg n$  level saves us from going all the way from the root down to the  $LCA$ . ►rephrase◀

In order to use this data structure to report points in the range of  $q = [x_1, x_2] \times [y_1, y_2]$  we follow these steps:

1. We find the rank space successor of  $x_1$ ,  $\hat{x}_1$ , and the rank space predecessor of  $x_2$ ,  $\hat{x}_2$ . We use these to find the lowest common ancestor of  $\hat{x}_1$  and  $\hat{x}_2$ ,  $LCA(\hat{x}_1, \hat{x}_2)$ . This is the lowest node in the tree containing at least all the points between  $x_1$  and  $x_2$ . By knowing  $\hat{x}_1$  and  $\hat{x}_2$ , finding the lowest common ancestor is a constant time operation. Given that points are in an array, we can use xor as our  $LCA$  operation. ►rephrase◀
2. As in step 1 where we found the rank space of the x-coordinates, we find the rank space coordinates of the y-coordinates,  $\hat{y}_1$  and  $\hat{y}_2$ , inside the left and right child of  $LCA(\hat{x}_1, \hat{x}_2)$ . This step is precisely what the rank space predecessor structure mentioned above supports.
3. We now descend into the right child of  $LCA(\hat{x}_1, \hat{x}_2)$  and use the range minimum query structure to find the index  $m$  (the y-rank) of the point with the smallest x-rank in the range  $[\hat{y}_1, \hat{y}_2]$ . Solving the ball-inheritance problem, we follow the path to the leaf to find the actual x-coordinate of

the point. If the x-coordinate is smaller than  $x_2$  we return the point as a result and recurse into the ranges of  $[\hat{y}_1, m - 1]$  and  $[m + 1, \hat{y}_2]$  in order to find more points. When this is done we apply the same concept to the left child of  $LCA(\hat{y}_1, \hat{x}_2)$  using the range maximum query to find points above  $x_1$ .

►Insert figure to conceptually show we are working our way out from the inside◄

The time complexity of step 3 depends on the use of the ball-inheritance structure. The time to traverse this structure is dependent on the improvements made in 2.2.2. An empty range will result in two queries, one query to each child of  $LCA(\hat{x}_1, \hat{x}_2)$ . In the worst case the amount of queries to the ball-inheritance structure will not exceed twice the number of results reported. Each time a result is found, a recursion is made to both the left and right subrange of that result. If one of the sides constantly fails to find a result, at most two queries are made for each result found.

Conceptually,  $LCA(\hat{x}_1, \hat{x}_2)$  describes a point between  $x_1$  and  $x_2$ . Step 3 selects points that are in the range of  $[y_1, y_2]$  moving outwards from the point of  $LCA(\hat{x}_1, \hat{x}_2)$ , always picking the point closest to  $LCA(\hat{x}_1, \hat{x}_2)$  in its decreasing y-range. ►rephrase◄

Going back to Lemma 2.4, we see that the time complexity fits:  $\mathcal{O}(\lg \lg n)$  time is used for the predecessor search and  $\mathcal{O}((1 + k)\tau)$  time is used for walking from the  $LCA$  to the leaves solving the ball inheritance problem for the  $k$  results.

The three approaches described above all use a number of bits ►bits or bytes? Anden formulering?◄ linear to the amount of points. The original search structure by Chan et al. [2] is the fastest with its  $(1 + k)\mathcal{O}(B \lg \lg n)$  query while kd-trees are the slowest with  $\mathcal{O}(\sqrt{n} + k)$  query time.

►ADD CONSTANT TIME DIFFERENCE TALK◄

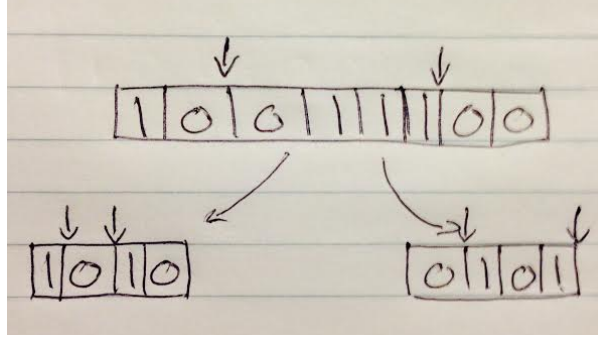


Figure 2.1: When nodes inherit their bit vector ranges from their parent, it can become obvious if the entire subtree is contained within the range of  $[y_1, y_2]$  or falls without the range.

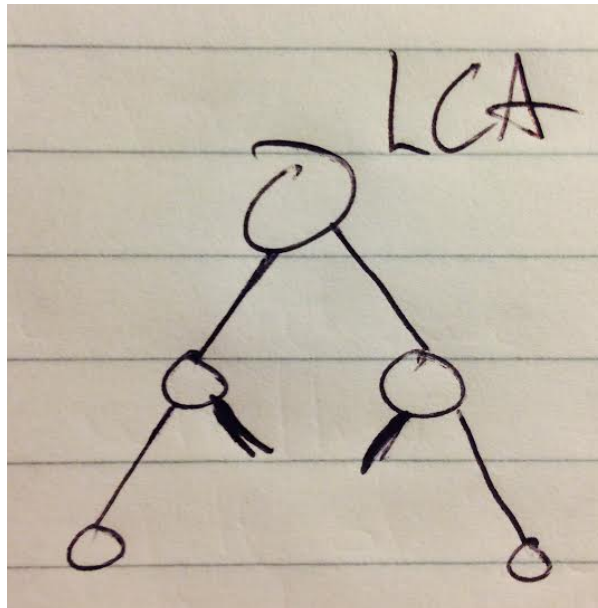


Figure 2.2: Traversing left from the LCA, each right subtree contains x-coordinates between  $x_1$  and  $x_2$ . Traversing right from the LCA the same holds for left subtrees.





## Chapter 3





## Chapter 4

# Conclusion





# Bibliography

- [1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. ISBN 3540779736, 9783540779735.
- [2] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the Twenty-seventh Annual Symposium on Computational Geometry*, SoCG '11, pages 1–10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0682-9. doi: 10.1145/1998196.1998198. URL <http://doi.acm.org/10.1145/1998196.1998198>.
- [3] Johannes Fischer. Optimal succinctness for range minimum queries. *CoRR*, abs/0812.2775, 2008. URL <http://arxiv.org/abs/0812.2775>.