# Orthogonal Range Searching in 2D with Ball Inheritance

Mads Ravn, 20071580

Master's Thesis, Computer Science

June 2015

Advisor: Kasper Green Larsen

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

# Abstract

This thesis studies orthogonal range searching which is one of the most well-studied problems in computational geometry[2]. We introduce the Ball Inheritance Search data structure. This data structure requires $\mathcal{O}(n)$ space and can return $k$ points with a query time of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ for any fixed constant $\epsilon > 0$. We compare the Ball Inheritance Search data structure to the kd-tree. The kd-tree requires $\mathcal{O}(n)$ space and can return $k$ points with a query time of $\mathcal{O}(\sqrt{n} + k)$.

We compare the query time of the two data structures with queries of different shapes. We show that both data structure have both best-case and worst-case shapes for their queries, and given a search query of their best-case shape they perform better than their counterpart.

We also look at the performance with a small amount of results. A search query of the worst-case shape to the Ball Inheritance Search data structure does not perform that much worse than the best-case shape to the kd-tree, while a search query of the best-case shape the Ball Inheritance Search performs much better than a search query of the worst-case shape to the kd-tree. The difference between the best-case and worst-case shape is least on the Ball Inheritance Search data structure, and thus it has a more stable performance. Finally we look at the space usage of the Ball Inheritance Search data structure and how we are able to leverage performance with space usage.

# Acknowledgements

First and foremost, I would like to thank my thesis advisor, Kasper Green Larsen. He shared his idea for the thesis with me, and he has been extremely helpful and engaged in the process of my writing. It has been a great experience to work with Kasper. I would also like to thank Eva Frederiksen for creating the figures in chapters 1, 2 and 3. It obviously helps a lot when the figures make sense. Johan Abildskov was kind enough to read an early draft of my thesis and give me some very good feedback and ideas.

Some of the figures in this thesis are inspired by the figures in Computational Geometry: Algorithms and Applications[1].

*Mads Ravn,*
*Aarhus, June 11, 2015.*

# Contents

**Abstract** iii

**Acknowledgments** v

**1 Introduction** 3

**I Theory** 7

**2 Related Work** 9
- 2.1 kd-trees 9
- 2.2 Range trees 13
- 2.3 Composite-number space 15
- 2.4 Summary 16

**3 Primary Work** 17
- 3.1 Ball Inheritance Search 18
  - 3.1.1 Preliminaries 18
  - 3.1.2 Solving the ball-inheritance problem 20
  - 3.1.3 Solving range reporting 21
- 3.2 Original Ball Inheritance Search 24
  - 3.2.1 Preliminaries 25
  - 3.2.2 Solving range reporting 26
- 3.3 Summary 30

**II Practical** 33

**4 Implementation** 35
- 4.1 Language 35
- 4.2 Design choices 35

**5 Analysis** 37
- 5.1 Square search queries 38
  - 5.1.1 Setup 38
  - 5.1.2 Data 39
- 5.2 Vertical and horizontal slices 42

5.2.1 Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.2.2 Vertical slices . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2.3 Horizontal slices . . . . . . . . . . . . . . . . . . . . . . . 46
5.3 Slices with small $k$ . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.4 Different BIS data structures with $B$ varying . . . . . . . . . . . 51
5.5 Vertical slices explained . . . . . . . . . . . . . . . . . . . . . . 53
5.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**6 Conclusion** **57**

**Primary Bibliography** **59**

Chapter 1

# Introduction

*"If this goes badly and I make a crater, I want it named after me!"*
— Iain M. Banks*, Against a Dark Background*

*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 points lie within a given axis-aligned query rectangle $\mathbb{Q} \subseteq \mathbb{R}^2$.

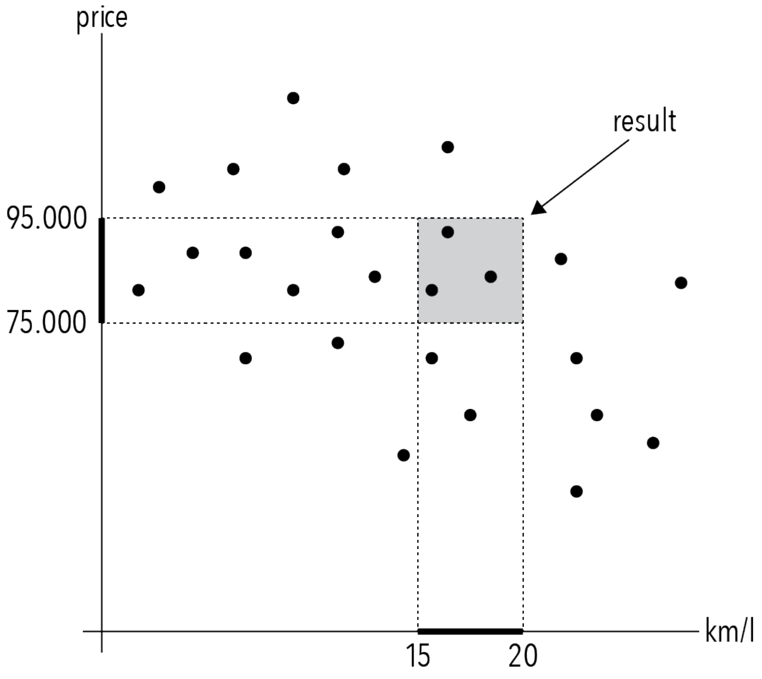


Figure 1.1: Example of an orthogonal range query

To motivate the problem, consider a database of vehicles for sale. Each vehicle has measurable attributes like price, the year the model was released, engine size, amount of doors, gasoline consumption in kilometers per liter, size and maximum speed. Perhaps a buyer is interested in finding cars which cost between $75,000$ and $95,000$ DKK and can drive between 15 and 20 kilometers per liter of gasoline. We can see such a search on figure 1.1, where each point within the gray area represents a car which fits the criteria, i.e. a search on two parameters is equivalent to finding all points within a 2-dimensional orthogonal range query. A point in the graph represents the ID of the car with which the car can be looked up in the database to find the other attributes of the car. When performing a search, two attributes are picked and a range for both attributes is chosen, giving a 2-dimensional search query. We can think of each car as a point with one coordinate per attribute. Given the ranges of two attributes we want to find those of the cars in the database which lie within the search query. In the example on figure 1.1 the search range is the 2d rectangle $[15; 20] \times [75,000; 95,000]$ returning three cars as the result.

The objective of this thesis is to study a variety of *orthogonal range searching* data structures. The main focus will be to introduce the *Ball Inheritance Search* data structure. It is a simplification of an orthogonal range searching data structure by Chan et al. [2], which will be referred to as the *Original Ball Inheritance Search* data structure. We are going to describe the kd-tree which will be the reference data structure in our analysis of the Ball Inheritance Search data structure. We are going to describe the range tree which shares some of properties of the Ball Inheritance Search and Original Ball Inheritance Search data structures.

We will look at the best-case and worst-case range queries for both the Ball Inheritance Search data structure and the kd-tree. We will show that the Ball Inheritance Search data structure is able to compete with the kd-tree, and even strongly outperform the kd-tree in cases where the shape of the query is a long thin slice through the attribute area. We will look at how resilient the data structures are to changes in shapes by looking at the best-case shaped search queries to both data structures compared to the worst-case shaped search queries. Some of these experiments will be performed with search queries with a small amount of results in order to emulate actual user interaction. Finally we are going to explore how much space the Ball Inheritance Search data structure uses and how we can leverage performance with space usage. We are going to compare the space of the Ball Inheritance Search data structure to the space of the kd-tree.

The model of computation used in this thesis is the $w$-bit word-RAM model by Fredman and Willard [4]. In the word-RAM model of computation, the memory is divided into words of $w$ bits. Given a set $P$ of $n$ points with integer coordinates from a universe $[U] = \{0, \dots, U-1\}$, we assume a word will have enough bits to store the integer address of any index into $P$ and enough bits to store any element from $U$. Thus, $w = \Omega(\lg n)$ and $w = \Omega(\lg U)$. Under the word-RAM model all standard word operations take constant time. This includes standard word operation from modern programming languages such as integer addition, subtraction, multiplication, division, shifts and the bit-wise operators AND, OR and XOR. Reading a single word from memory or writing a single word to memory also takes constant time. The number of bits in a word is found by the largest element which has to fit into a word. This means that it is often possible to divide the word into smaller logical blocks which can fit more than one integer.

**Outline.** In Chapter 2 we introduce related work. This covers the kd-tree and the range tree. In Chapter 3 we introduce the primary work of this thesis, the Ball Inheritance Search data structure, followed by the original work by Chan et al. [2]. Some implementation specifics are described in Chapter 4. The experiments performed on the Ball Inheritance Search data structure and its comparison to the kd-tree will be discussed in Chapter 5. Finally Chapter 6 will be the conclusion.

**Notation.** The set of integers $\{i, i+1, \ldots, j-1, j\}$ is denoted by $[i, j]$. When no base is explicitly given logarithm will have base 2. $\epsilon$ is an arbitrary small constant greater than 0. 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 $A$, including both $A[i]$ and $A[j]$. $A[1..n]$ denotes an array $A$ of size $n$ with entries 1 to $n$. Throughout the thesis the successor of $x$ in a set will be meant as the smallest number which is greater or equal to $x$ in that set - symmetrically, the same applies for predecessor of $x$ which is the biggest number less or equal to $x$. 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

This chapter will describe two well-known *static* data structures for orthogonal range searching: the kd-tree and the range tree. The kd-tree is the current de facto standard for orthogonal range searching because of its low space complexity. It has a space complexity of $\mathcal{O}(n)$ and a running time of $\mathcal{O}(\sqrt{n}+k)$ where $k$ is the number of results reported. In practice it uses the exact same amount of words as it holds elements[1]. The range tree uses $\mathcal{O}(n \lg n)$ words of space and has a running time of $\mathcal{O}(\lg^2 n + k)$ without *fractional cascading* and a running time of $\mathcal{O}(\lg n + k)$ with fractional cascading. The difference between the space complexities of the kd-tree and the range tree is a factor $\mathcal{O}(\lg n)$ which can become an issue when dealing with very large datasets and a limited amount of main memory. This factor can grow big for large datasets and is the reason why kd-trees are preferred in practice.

The kd-tree will be used as a reference in the comparison against the Ball Inheritance Search data structure since they have the same space complexity. This property makes the Ball Inheritance Search data structure quite attractive. The range tree will be used to show how much the running time can be decreased by increasing the space complexity by a factor of $\mathcal{O}(\lg n)$. The range tree will also be used to introduce some of the ideas behind the Ball Inheritance Search and Original Ball Inheritance Search data structures, which is where Chan et al. [2] drew some of their inspiration.

The query time of a search query to the main data structures in this chapter are all *output-sensitive*, meaning that their running time depends on the amount of results found. The data structures themselves are static: After the initial construction of the data structures they will not be altered by insertions or deletions.

## 2.1 kd-trees

The current standard of range reporting using linear space is the kd-tree. This data structure will be used as a reference point when evaluating the results of the primary work of the thesis. With linear space it is a fitting data structure for range reporting on the RAM, and a practical solution. The kd-tree with $n$ points can be represented as an array $A[1..n]$.

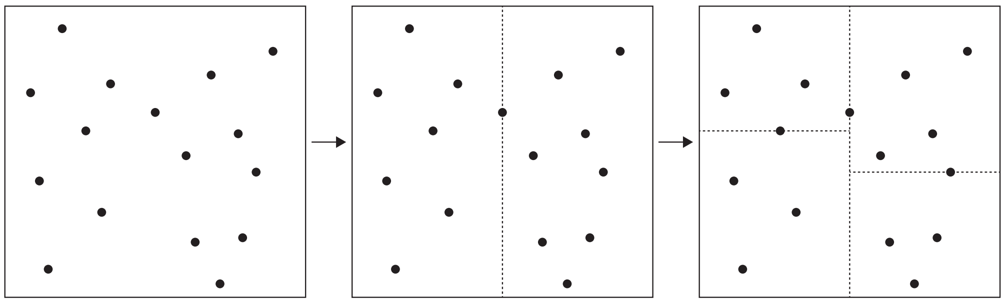


Figure 2.1: Showing the subdivision of points in a node: First dividing the points by the x-median, and then by the y-median

A kd-tree is constructed recursively: Given $n$ points, the median of the points with respect to x is 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 goes to the left child. Conceptually the median belongs to the left child, but in the array implementation the median will be stored in the middle, between the points going to the left child and the points going the right child. At the next level the points of each node will be divided in a similar fashion, this time using the y-median and the y-coordinates instead. This is shown on figure 2.1. When dividing $n$ points, the median will be chosen as the $\lceil n/2 \rceil$-th smallest number. Therefore we can think of a node as containing 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 uses $\mathcal{O}(n)$ words of space.

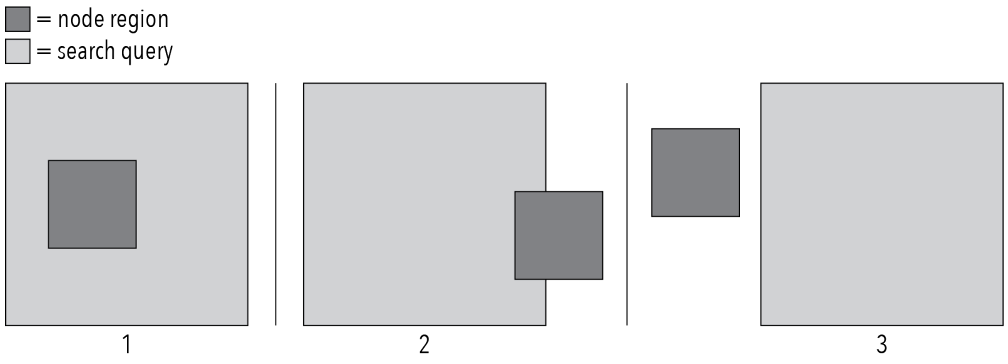


Figure 2.2: The three different situations which can occur between a search query and the region of a node

In order to search in this tree, we introduce the term *region*. The region of the root node is $\mathbb{R}^2$. Given a node $v$ with the region $R$, the region of the children is $R$ restricted to one side of the splitting line at $v$. The region of the left child of $v$ is the left side of $R$ split by the splitting line. The region of the right child of $v$ is the right side of $R$ split by the splitting line. The root contains all points and has the biggest region. Since each node contains a line dividing its points between both of its children, we can use this line to decrease the size of the regions of both children. Doing this halves the amount of points lying within each child region. As shown on figure 2.2, given an axis-aligned rectangular search query $q = [x_1, x_2] \times [y_1, y_2]$ and a node $v$ in the kd-tree the following can occur:

1. The region of the node can be fully contained in the search query, in which case all of the points stored in the leaves of the subtree rooted at $v$ are returned as part of the result.

2. The search query can overlap, but not fully contain, the region of the node $v$. In this case the search will check each child node of $v$ as to which of these three cases occur.

3. Finally the region of the node and the search query can have nothing in common in which case nothing happens and this branch of the search stops.

The search starts at the root of the kd-tree. In the kd-tree, the root has no special properties, so it just acts like any given node $v$. Here all three cases can occur. If case 1 occurs, all the points of the kd-tree will be part of the result. If case 3 occurs, the result of the search will be empty and the search will stop. If case 2 occurs, it will be checked which of the three cases above occurs at each of the two children of the root. Thus, case 2 is used to travel recursively down the kd-tree to determine which points will be returned as the result.
If a leaf is visited in the search, the point stored in the leaf is reported as part of the result if it lies within the search query. Also note that internal nodes of the kd-tree are the root of a smaller kd-tree.

Given a node whose region is fully contained, the time to report the points stored in the subtree of that node is linear in the number of points reported. Thus, it takes $\mathcal{O}(k_v)$ time to report all the $k_v$ points stored in the subtree of a node $v$ which is fully contained in the search region.

We say that a node visited by case 2 has a *non-empty parent*. From figure 2.2, case 1 takes $\mathcal{O}(k_v)$ time when $v$ has a non-empty parent. From a non-empty parent, case 3 takes constant time. In order to bound the time of a search query to the kd-tree, we need to bound the time spend on case 2. For this, we need to obtain a bound on the amount of nodes visited which are not fully contained in the search region. These are the nodes where an edge of the search query passes through their regions. Consider a search query where one of the edges passes through the region of the root. This edge can be thought of as infinitely long. Without loss of generality, we pick it to be a vertical line, as seen on figure 2.3.

We thus want to bound the number of regions of nodes where one of the query edges passes through. Let $Q(n)$ describe the amount of regions this

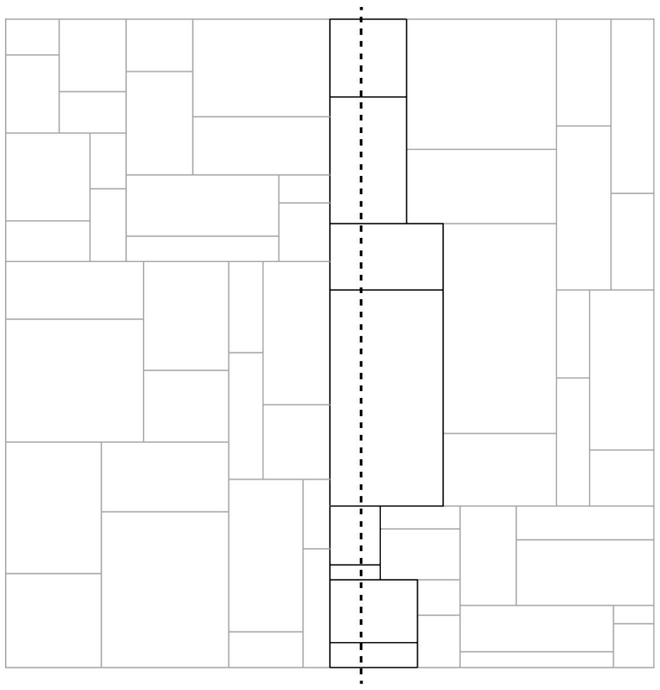


Figure 2.3: Example of a vertical edge through the entire region of the root node

infinitely long vertical line intersects. In order to bound the amount of regions intersected by the line, we need to recall how the kd-tree is built. Consider a region that is intersected by the line. When constructing the kd-tree, this region is split in one dimension and then in the other dimension, resulting in four regions for every two levels of the kd-tree. The vertical line can at most intersect two of these regions. Thus, when a vertical line intersects the region of a node, it intersects at most two of the four regions of the its descendants two level down the tree. The running time of a query to the kd-tree with $n$ points can thus be described by the recurrence:

$$Q(n) = \begin{cases} \mathcal{O}(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1 \end{cases}$$

Solving this recurrence gives the solution $Q(n) = \mathcal{O}(\sqrt{n})$. Since the vertical line is infinitely long, it intersects as least as many regions as the edge of a search query. The vertical line can intersect at most $\mathcal{O}(\sqrt{n})$ regions of the kd-tree, This is an upper bound for the amount of regions a vertical line can intersect, thus also bounding the amount of regions of nodes the edge of a search query can pass through. The exact same argument can be made for a horizontal line.

Searching the kd-tree thus takes $\mathcal{O}(\sqrt{n} + k)$ time to report $k$ points as a result. When the amount of points reported as result of the search query is low, the query time per point is relatively high. Another thing to notice is that there is no time penalty per point reported. Just searching through the data structure costs $\mathcal{O}(\sqrt{n})$ time, but the time to report points is linear in the number of points reported.

## 2.2 Range trees

The range tree is another data structure which supports range queries. The space complexity of this data structure is $\mathcal{O}(n \lg n)$. A range query to the range tree takes $\mathcal{O}(\lg^2 n + k)$ time to report $k$ points. This time can be optimized to $\mathcal{O}(\lg n + k)$ without changing the space complexity using *fractional cascading.* We will first look at how the data structure is built and how it is used for range reporting. Then we will introduce fractional cascading and see how that will change the query time. With a space complexity of $\mathcal{O}(n \lg n)$ words this data structure is not going to replace the kd-tree. Instead the range tree will serve as a way to introduce some of the ideas behind the Ball Inheritance Search and Original Ball Inheritance Search data structures.

Consider a balanced binary search tree with $n$ keys for a 1-dimensional query on the x-coordinates. These $n$ keys are sorted lowest-to-highest in the leaves of the tree, from left to right. In order to answer the query $q = [x_1, x_2]$ the following is done: From the root, travel to the *least common ancestor* of $x_1$ and $x_2$. This is the node whose subtree contains both $x_1$ and $x_2$, and $x_1$ lies in the left subtree and $x_2$ lies in the right subtree. From the least common ancestor, travel to both $x_1$ and $x_2$. While traveling to $x_1$, the first step is the left child of the lowest common ancestor of $x_1$ and $x_2$. From here, every time a left child is chosen as the next step in the path, the subtree in the right child will only contain points between $x_1$ and $x_2$. The nodes of this entire subtree are reported as results. Symmetrically, the same is done with the path to $x_2$. When a right child is chosen as the next step, the nodes of the entire subtree in the left child are reported as results. In a 1-dimensional search, when a node is the root of a subtree which only contains points in the search range, the node is said to be *fully contained.*

A balanced binary search tree has a space complexity of $\mathcal{O}(n)$. Reporting the points stored in a subtree requires time linear to the amount of points in the subtree. Travelling from the root to $x_1$ and $x_2$ requires $\mathcal{O}(\lg n)$ time. Hence, the query time of a 1-dimensional search query is $\mathcal{O}(\lg n + k)$.

Range reporting in a 2-dimensional space on the range tree is done by using 1-dimensional sub-queries where it separates the dimensions. Given a search query $q = [x_1, x_2] \times [y_1, y_2]$, it will first find the points which lie in the range of $[x_1, x_2]$. Among those points, it will find the points which lie in the range of $[y_1, y_2]$. This leaves us with all the points lying in the search query.

Doing the first 1-dimensional search is exactly what is accomplished using a balanced binary search tree. A balanced binary search tree is built to support range search on the x-axis of all of the points. We will call this tree the primary tree. Then for each internal node in the primary tree a new balanced binary search tree is built on the y-coordinates of all points in the leaves of the subtree rooted at that node. We call these balanced binary search trees for auxiliary trees. The primary tree holds pointers to the auxiliary tree for each node.

A range query $q = [x_1, x_2] \times [y_1, y_2]$ on the range tree is answered in the following way. From the least common ancestor of $x_1$ and $x_2$, the search travels down to $x_1$ and $x_2$. On the way to $x_1$ and $x_2$, each node that is fully contained in $[x_1, x_2]$ will be flagged. Using the auxiliary tree of each node that is flagged, a search will be done to find the points in the range $[y_1, y_2]$.

Each leaf in the primary tree of the range tree stores a point. The height of a balanced binary search tree containing $n$ points is $\lg n$. Each point $p$ in the primary tree is only stored in the auxiliary trees of nodes on the path to the leaf containing the point $p$. This means that each point $p$ is only stored once per level in the primary tree. Each auxiliary tree uses space linear to the amount of points it holds. Thus, the space complexity of a range tree is bounded by $\mathcal{O}(n \lg n)$.

The query time for each auxiliary tree that is searched is $\mathcal{O}(\lg n + k_v)$, where $k_v$ is the amount of points that is reported back by the auxiliary tree at the node $v$ in the primary tree. The amount of auxiliary trees which will be searched is bounded by the length of the path from the least common ancestor of $x_1$ and $x_2$ to the leaves containing $x_1$ and $x_2$. This path can at most visit two nodes per level of the primary tree, and the length is thus bounded by $\mathcal{O}(\lg n)$. The query time of a range search in the range tree is then

$$\sum_v \mathcal{O}(\lg n + k_v) = \mathcal{O}(\lg^2 n + k)$$

where $v$ are the nodes flagged on the path to $x_1$ and $x_2$ from their least common ancestor.

Fractional cascading can be used to speed up the query time without changing the space complexity of the data structure. Instead of using a balanced binary search tree as the auxiliary data structure, we are going to use an array. This array will contain the same points as the auxiliary balanced binary search tree did. The points in the array will be sorted by their y-coordinate. At the node $v$, each entry in the array $A_v$ will contain a point and two pointers. One pointer will be pointing to an entry in the auxiliary array of the left child of $v$, while the other pointer will be pointing to an entry in the auxiliary array of the right child of $v$. We call these the left pointer and the right pointer, respectively. Suppose that $A_v[i]$ stores a point $p$. Then the left pointer at $A_v[i]$ will be pointing to the first entry in the left child's auxiliary array containing a point with a y-coordinate greater or equal to $p_y$. The same applies to the right pointer of $A_v[i]$, pointing to the right child instead of the left child.

Searching the range tree with fractional cascading starts by finding the least common ancestor of $x_1$ and $x_2$. At this node, a binary search is done in order to find the first entry in the auxiliary array which y-coordinate is greater or equal to $y_1$. At any given node $v$, we call the position of this entry $\tau_v$. We walk from the least common ancestor of $x_1$ and $x_2$ to the leaves $x_1$ and $x_2$, finding all

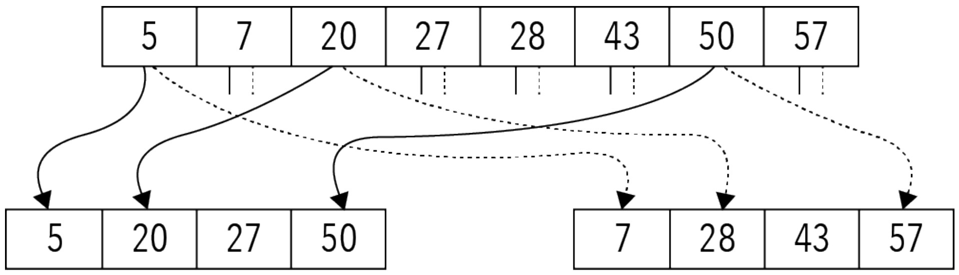


Figure 2.4: Example of an fractional cascading

the nodes which are fully contained in $[x_1, x_2]$. Each time a left child is visited on the path to $x_1$ or $x_2$, the left pointer is used to update $\tau$. The entry at position $\tau_v$ is the first element in $A_v$ which is greater or equal to $y_1$. Finding the index of this element at a child-node is a constant-time operation using the left pointer. Symmetrically, when a right child is visited on the path to $x_1$ and $x_2$, the position of $\tau$ is updated using the right pointer. This can be seen on figure 2.4. When a fully contained node is found, we look in the auxiliary array from the position of $\tau$ and $k_v$ entries forward in order to report $k_v$ points back as result. This is done by incrementing the position of $\tau$ until the point at that entry is no longer within the range of $[y_1, y_2]$. This takes $\mathcal{O}(1 + k_v)$ time. The total query time now becomes

$$\sum_v \mathcal{O}(1 + k_v) = \mathcal{O}(\lg n + k)$$

where $v$ are the nodes flagged on the path to $x_1$ and $x_2$ from their least common ancestor.

## 2.3 Composite-number space

In order to ensure all points have unique x-coordinates and unique y-coordinates, the points are translated into *composite-number space*[1]. A composite number of two numbers $x$ and $y$ is denoted by $(x \mid y)$. A total ordering on the composite-number space is defined by using lexicographic order. Given two composite numbers $(x_1 \mid y_1)$ and $(x_2 \mid y_2)$, we define the order as

$$(x_1 \mid y_1) < (x_2 \mid y_2) \iff x_1 < x_2 \text{ or } (x_1 = x_2 \text{ and } y_1 < y_2)$$

Given a set P of $n$ distinct points from $\mathbb{R}^2$, we translate each point $(x, y) \in P$ into composite-number space by assigning the point new set of coordinates: $(x, y) := ((x \mid y), (y \mid x))$. No two points will have the same x-coordinate unless the points are identical. The same holds for the y-coordinate.
In order to perform a range query $q = [x_1, x_2] \times [y_1, y_2]$ in composite-number space, the query will have to be transformed. This transformed range query will be $\hat{q} = [(x_1 \mid -\infty), (x_2 \mid +\infty)] \times [(y_1 \mid -\infty), (y_2 \mid +\infty)]$. It follows that

$$(x, y) \in q \iff ((x \mid y), (y \mid x)) \in \hat{q}$$

## 2.4 Summary

The kd-tree is a data structure using $\mathcal{O}(n)$ words of space and supports a range query in $\mathcal{O}(\sqrt{n} + k)$ time. It is built by continually subdividing smaller and smaller regions of the tree until a region only contains one point. A range search query will then match the query region to the region of a node to see if there is any overlap or full containment.

The range tree is a data structure using $\mathcal{O}(n \lg n)$ words of space and support a range query in $\mathcal{O}(\lg^2 n + k)$ time. It is built by constructing a tree with $n$ leaves and dividing the points to the leaves, such that all the leaves to the left of a leaf contain points with a smaller x-coordinate than the point at the leaf. All the internal nodes of the tree contain an auxiliary tree which has the same property just with the y-coordinate of the points contained in the subtree. This property allows a search query to quickly locate the subtrees containing only points between $[x_1, x_2]$ and $[y_1, y_2]$. Using fractional cascasding we can speed up the query time of the range tree to $\mathcal{O}(\lg n + k)$.

The $\mathcal{O}(\lg n + k)$ running time of the range tree is faster than the $\mathcal{O}(\sqrt{n} + k)$ running time of the kd-tree. However, the $\mathcal{O}(\sqrt{n})$ part is based on a rather pessimistic idea that a range query will overlap, but not fully include, a lot of regions stretching over the two extremities in one dimension. In practice, the $\mathcal{O}(\lg n)$ penalty on space is prohibited for most applications and is the main reason why kd-trees are preferred.

# Chapter 3

# Primary Work

*"My name?" said the old, and the same distant sadness came into his face again. He paused. "My name," he said, "is Slartibartfast."*
—Douglas Adams*, Hitchhiker's Guide to the Galaxy*

This chapter will introduce the Ball Inheritance Search and Original Ball Inheritance Search data structures. The Original Ball Inheritance Search data structure by Chan et al. [2] is a tree-like data structure with auxiliary data structures. It has a space complexity of $\mathcal{O}(n)$ and a supports search queries in $\mathcal{O}(\lg\lg n + (1 + k) \cdot \lg^{\epsilon} n)$ time, where $k$ is the amount of results reported and $\epsilon$ is an arbitrarily small constant greater than 0.

The Ball Inheritance Search data structure is a simplification of the Original Ball Inheritance Search and therefore they have the same underlying data structure. The Ball Inheritance Search data structure has some different auxiliary data structures and fewer of them. The data structure has a space complexity of $\mathcal{O}(n)$ and supports search queries in $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ time, where $k$ is the amount of results reported and $\epsilon$ is an arbitrarily small constant greater than 0. Going forward, *OBIS* will be used as shorthand for *Original Ball Inheritance Search* and *BIS* will be used as shorthand for *Ball Inheritance Search*.

The BIS data structure has a time complexity of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ which is greater than the time complexity of the OBIS data structure with $\mathcal{O}(\lg\lg n + (1 + k) \cdot \lg^{\epsilon} n)$. However, the BIS data structure is far more simple - both in code and the auxiliary data structures used. The difference between the running time constant hidden in $\mathcal{O}(\lg\lg n)$ and $\mathcal{O}(\lg n)$ is far greater than the difference between $\lg\lg n$ and $\lg n$. This makes the BIS data structure faster than the OBIS data structure in practice.

Each section will start with a *preliminaries* subsection. This subsection will describe some of the auxiliary data structures used in the section. As in Chapter 2, the main data structures in this chapter are static and the query time of the range queries are output-sensitive.

## 3.1 Ball Inheritance Search

This section will introduce the primary work of this thesis. It will show how the BIS data structure is built and how range reporting is done using the data structure. The data structure uses $\mathcal{O}(n)$ space and supports search queries in $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ time. This is the same space complexity as the kd-tree. The query time is different in that $\mathcal{O}(\lg n)$ is smaller than $\mathcal{O}(\sqrt{n})$, but there is a cost of $\mathcal{O}(\lg^{\epsilon} n)$ per point reported.

The BIS data structure relies heavily on the *ball-inheritance* data structure. The ball-inheritance structure is a tree with $n$ labelled balls at the root. In $\lg n$ steps it will distribute the balls from the root of the tree to $n$ leaves in the tree. Solving the ball-inheritance problem is to follow a ball from any given node to its leaf. We formally define the problem below in section 3.1.1. Succinct rank queries are an important part of data structure, playing a key role in solving the *ball-inheritance problem.*

The BIS data structure supports search queries in a manner similar to the range tree. A balanced binary search tree is used to locate the subtrees which is fully contained in $[x_1, x_2]$. From here the range tree uses balanced binary search trees or fractional cascading to locate which of those points are in $[y_1, y_2]$, while the BIS data structures uses the ball-inheritance structure to decode the y-ranks to actual points. The OBIS data structure will also use a balanced binary search tree, but only to locate the least common ancestor of $x_1$ and $x_2$. From here it will perform ball-inheritance queries to find the points within $[x_1, x_2] \times [y_1, y_2]$.

### 3.1.1 Preliminaries

#### Rank Space Reduction

Given $n$ points from a universe $U$, the rank of a given point in a sorted list of points is defined as the amount of points which precedes it in the 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 the *x-rank* $r_x$ for each point in $P$ by finding the rank of the x-coordinate among all the x-coordinates in $P$. The *y-rank* $r_y$ finds the rank of y-coordinate among 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 $q^* = [r_x(x_1), r_x(x_2)] \times [r_y(y_1), r_y(y_2)]$. 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 $\lg n \ll \lg U$ when $n \ll U$ which saves memory. Given $n$ points from $U$ we can translate them to rank space by inserting them in an array $A[1..n]$ and sort $A$. The index of a point in $A$ is now its rank. In order to translate a search query $q = [x_1, x_2]$ to rank space, we will look up the successor of $x_1$ in $A$ and the predecessor of $x_2$ in $A$. The indices of these two points delimits the search query in rank space. The array $A$ acts as a mapping to and from rank space. When a rank space reduction has been applied and there exists an array $A[1..n]$ mapping the elements to and from rank space, the algorithms used will only use RAM operations on integers of $\mathcal{O}(\lg n)$ bits.

### 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 greater than or equal to 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 *rank 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 report how many elements in $A[1..i]$ are equal to $A[i]$. This is called a *rank query*. Thus, the function is $rank(k) = \Sigma_{i \leq k} A[i]$. We want to be able to compute a 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 $\mathcal{O}(\Sigma \lg n)$ bits of space. By placing the checkpoints $\Sigma \lg n$ entries apart of each other, all of the checkpoints use $\mathcal{O}(\frac{n}{\Sigma \lg n} \cdot \Sigma \lg n) = \mathcal{O}(n)$ bits of space.

Each entry in $A$ contains a character from the alphabet $\Sigma$. At $A[i]$ we also store the amount of times the character at $A[i]$ occurs 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 $x - 1$. This approach fits the required space bound if $\Sigma \geq \lg^{\epsilon} n$, because there $\Sigma$ will dominate the complexity. Hence, storing $n$ entries uses $\mathcal{O}(n \cdot \lg(\Sigma \lg n)) = \mathcal{O}(n \cdot \lg \Sigma)$ bits. Working under the word-RAM model of computation, we are able to pack the $n$ integers into $n \cdot \lg m$ bits, where $m$ is the maximum number which needs to be stored.

### Ball Inheritance

The input to the ball-inheritance problem is a perfect binary tree with $n$ leaves and $n$ labelled balls at the root. The balls have been distributed from the root to the leaves in $\lg n$ steps. All balls are mapped to distinct leaves and follow the path from the root to that leaf. Following this path from a node to its child, we say that a ball is inherited by that child. The balls at the root are contained in an ordered list, and they will retain this order at all internal nodes. Each node thus has an associated list of those balls passing through it. The level of a node is defined to be the height of the node from the leaves. 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. A ball can be identified by a node and the index of the ball in the list of that node. Given the identity of a ball at any level, it is possible to follow this ball down the tree to a leaf. 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 the *true identity* of a ball.

### 3.1.2 Solving the ball-inheritance problem

Consider an input to the ball-inheritance problem, i.e. a perfect binary tree with $n$ leaves and $n$ balls following root-to-leaves path. At level $m$, each node has a bit vector $A_v[1..2^m]$ used to indicate which of its children a ball is inherited by: If $A_v[i]$ is 0 it means that the ball at index $i$ in that node's list is inherited by the left child and 1 means that it is inherited by the right child. The identity of a ball is a node and an index into the list of that node. Given a node $v$ and an identity of a ball, we can now calculate the ball's identity in the child node which inherits the ball. The node can answer the query $rank_v(k) = \Sigma_{i \leq k} A_v[i]$. If a ball is inherited by the right child node its new identity at that node is $rank_v(i)$ because that is how many 1s that precede it in the current node, and thus the number of balls inherited by the right child that precedes the ball in the ball-ordering. If a ball is inherited by the left child node the new identity is then $i - rank_v(i)$. 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 represented by the bit vectors of the nodes on that level. 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 as described in section 3.1.1 about succinct rank queries.

#### Faster Queries

In the solution above, 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 children's children, $\Sigma = \{0, 1, 2, 3\}$, the children's children's 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 for each ball 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. Just like with a parent-to-child step, bigger steps are supported by succinct rank queries. In order to determine the identity of a ball $b$ at $\Sigma$ levels below, we need to know the destination node and how many balls before $b$ chose the same destination node. Using succinct rank queries to determine the rank of a ball $\Sigma$ levels down is thus a constant time query.

Using this concept, we pick $B$ such that $2 \leq B \leq m$, where $m = \lg n$ is the height of the tree. All levels that are a multiple of $B^i$ expand their alphabet such that the balls can also 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. This gives us a query time of $\mathcal{O}(B \lg_B \lg n)$ because we need at most $B$ jumps at most $\mathcal{O}(\lg_B \lg n)$ times.

Storing the expanded alphabets at each level that is a multiple of $B^i$ costs $B^i$ bits per ball. The total cost is then

$$\sum_{i=1}^{\lg_B \lg n} \frac{\lg n}{B^i} \cdot \mathcal{O}(B^i) = \mathcal{O}(\lg n \cdot \lg_B \lg n) \tag{3.1}$$

bits per ball, summed over all levels. With $n$ balls, this is $\mathcal{O}(n \lg_B \lg n)$ words of space with query time of $\mathcal{O}(B \lg_B \lg n)$. If we pick a $B \geq \lg^\epsilon n$ we can reduce $\lg_B \lg n$ to $\lg_{\lg^\epsilon n} \lg n = \frac{1}{\epsilon}$. Thus, the space complexity for the ball-inheritance data structure is $\mathcal{O}(\frac{1}{\epsilon} \cdot n) = \mathcal{O}(n)$ words of space. By picking $B = \lg^{\epsilon/2} n = \Omega(\lg \lg n)$ we can upper bound the query time as follows:

$$\begin{aligned} \mathcal{O}(B \lg_B \lg n) &= \mathcal{O}(B \lg \lg n) = \mathcal{O}(\lg^{\epsilon/2} n \cdot \lg \lg n) \\ &= \mathcal{O}(\lg^{\epsilon/2} n \cdot \lg^{\epsilon/2} n) = \mathcal{O}(\lg^\epsilon n) \end{aligned} \tag{3.2}$$

The ball-inheritance problem can thus be solved in $\mathcal{O}(\lg^\epsilon n)$ time using $\mathcal{O}(n)$ words of space for an arbitrary small constant $\epsilon > 0$. The upper limit of $\lg_B \lg n$ in equation 3.1 and equation 3.2 is chosen because that is the largest $i$ where $B^i \leq \lg n$.

### 3.1.3 Solving range reporting

Consider a perfect binary tree with $n$ leaves. The root contains $n$ points in 2-d rank space. The elements in the ball-inheritance structure are points, so the words *ball* and *point* can be used interchangeably. The $n$ balls at the root of the tree 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. The actual coordinates of the balls are only stored at the leaves. This is how the ball-inheritance data structure was described in the previous section. The ball distribution has been specified. With this distribution, some facts about the tree can be stated. We know that the x-coordinates of the balls in the leaves are sorted from left to right - smallest to highest. The way the balls are distributed from the root, the x-coordinates are responsible for the inheritance path. Because the nodes are sorted by their y-rank in the root node and that they keep this order, the balls in a node list at any given node is ordered by their y-rank. These two facts will be used to solve the range reporting.
Since the actual coordinates of the 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 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 now have our query $q$ in rank space: $\hat{q} = [\hat{x}_1, \hat{x}_2] \times [\hat{y}_1, \hat{y}_2]$. We use $\hat{x}_1$ and $\hat{x}_2$ to find the least common ancestor of $\hat{x}_1$ and $\hat{x}_2$, $LCA(\hat{x}_1, \hat{x}_2)$. The subtree rooted at this node contains at least all the points with an x-coordinate between $x_1$ and $x_2$.

At the root of the BIS data structure we mark the positions of $\hat{y}_1$ and $\hat{y}_2$ on the bit vector. This range indicates which balls lie in the range $[y_1, y_2]$. $i_v$ and $j_v$ will denote this range in the bit vector of the node $v$. When searching for points which lie within this range, a node will update this range to fit its children. The updated range at the left child $l$ will be $i_l = i_v - rank_v(i_v)$ and $j_l = j_v - rank_v(j_v)$. The updated range at the right child $r$ will be $i_r = rank_v(i_v)$ and $j_r = rank_v(j_v)$. This is the same way the rank query was used in section 3.1.2. Now instead of just following a given ball, we keep track of a range of balls. This concept is seen on figure 3.1.

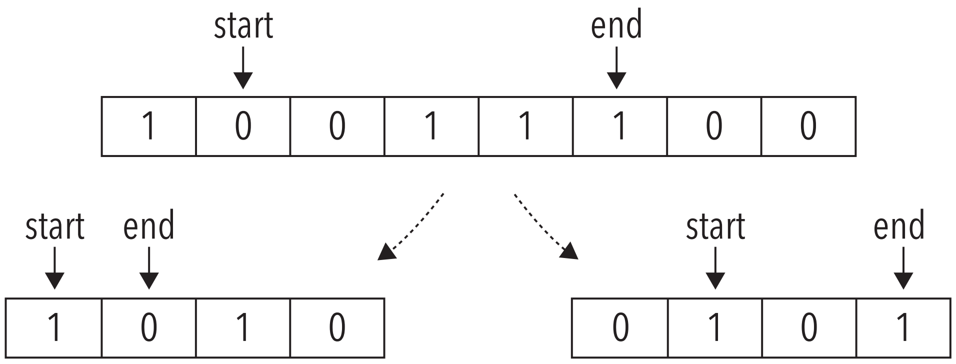


Figure 3.1: Example of nodes inheriting their bit vector ranges from their parent.

Traversing from the root to the $LCA$, this y-range will be maintained accordingly. 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 only contains points with x-coordinates 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 only contains points between $x_1$ and $x_2$. Such a subtree is said to be fully contained. When a subtree is fully contained, we also say that the node at which the subtree is rooted is fully contained. This concept is seen on figure 3.2. This is the same concept used with the range tree.

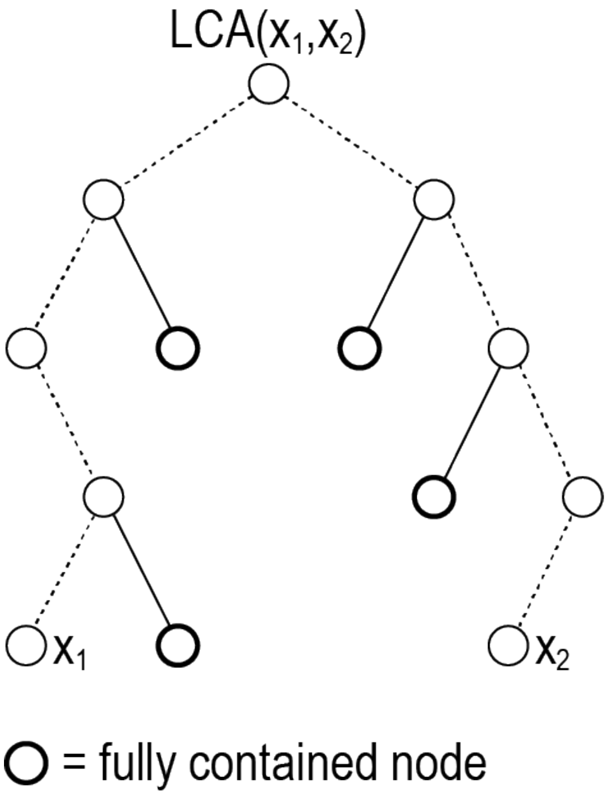


Figure 3.2: Traversing left from the LCA, each right subtree contains x-coordinates between $x_1$ and $x_2$. Traversing right from the LCA the same holds for left subtrees. Dotted line represent the path from the lowest common ancestor of $x_1$ and $x_2$ to $x_1$ and $x_2$

Each time a fully contained node $v$ is found, we want to follow all the balls lying in the y-range of the root of the subtree rooted at $v$ to their leaves. This is exactly what the solution to the ball-inheritance problem provides: We are given a list of ball identities and want to find which leaf stores the actual coordinates of the point we are looking for. Finding the coordinates of a ball takes $\mathcal{O}(\lg^{\epsilon} n)$ time per ball. The $\mathcal{O}(\lg^{\epsilon} n)$ thus describes, and bounds, how many jumps are needed from a node to a leaf.

The range tree from section 2.2 and the BIS data structures have some similarities. A range tree have a 1-d binary tree over the x-coordinates and an auxiliry data structure to look up the y-coordinates of fully contained nodes. We can think of the range reporting of the BIS data structure in terms of how the range tree works. Conceptually, there is a 1-d binary tree over the x-coordinates in the BIS data structure. When a fully contained node is found from the least common ancestor of $x_1$ and $x_2$, the ball-inheritance structure is used as the auxiliary data structure to decode the location of the leaf by using the rank of the y-coordinate. So in essence, the range tree and BIS data structure are quite alike, just with a different auxiliary data structure.

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

## 3.2 Original Ball Inheritance Search

This section describes the OBIS data structure. While the theoretical work of this thesis is a simplification of this data structure, the OBIS can also be viewed as an extension of the BIS. The OBIS data structure will <u>not</u> be implemented, but serves as a theoretical background for the BIS. The main property the BIS and OBIS share is that the underlying data structure is the ball-inheritance data structure and solving the range reporting heavily relies on solving the ball-inheritance problem.

Utilizing the ball-inheritance structure, Chan et al. [2] propose a theoretically better solution for orthogonal range search queries than the one of the Ball Inheritance Search data structure:

**Theorem 2.1** *for any $2 \leq B \leq \lg^{\epsilon} n$, we can 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 $q = [x_1, x_2] \times [y_1, y_2]$.

### 3.2.1 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 $2n + \mathcal{O}(n)$ bits of space with constant query time. The construction requires that the array is ordered, which we will see fits into our scheme. Note that $\mathcal{O}(n)$ is less than the bits needed to store $A$. Thus, only the index of the minimum key in $A[i, j]$ can be returned, not the actual key. Using the range minimum query data structure, we can build a range maximum query data structure. This is a range minimum query data structure on the mirrored input, i.e. the biggest element becomes the smallest.

#### Rank space predecessor search

In order to look up the rank space predecessor of a given coordinate, another succinct data structure will be used. Given a sorted array $A[1..n]$ of $\omega$-bit integers, predecessor search queries in $\mathcal{O}(\lg \omega)$ time is supported using $\mathcal{O}(n \lg \omega)$ bits of space with *oracle access* to the entries in the array. Since $\mathcal{O}(n \lg \omega)$ bits of space is not enough to store $n$ $\omega$-bit integers, the data structure will only store part of each $n$ elements and defer the look-up operation to an *oracle machine*. For this purpose a Patricia trie [5] is used to store some parts of the $n$ $\omega$-bit integers and find which entries in the array $A$ which has to be looked up by the oracle machine. The oracle machine is some data structure that, given an index $i$, can return $A[i]$.

#### Finding the lowest common ancestor

In the Ball Inheritance Search, the lowest common ancestor of $x_1$ and $x_2$ was found by following the path from the root of the tree to both $x_1$ and $x_2$. The last node both paths shared was then the lowest common ancestor. This way of finding the lowest common ancestor takes $\mathcal{O}(\lg n)$ time which is too big for the Original Ball Inheritance Search. In order to look up the lowest common ancestor of $x_1$ and $x_2$ in constant time, we are going to look at the binary representation of $x_1$ and $x_2$. First we look at $x_1 \oplus x_2$. The number of zero bits from to the start to the first one bit describes the amount of nodes the two paths have in common. We denote this number $b$. This will tell us to look on level $\lg n - b$. The number that the first $b$ bits of $x_1$ describes is the identity of the lowest common ancestor of $x_1$ and $x_2$ on level $\lg n - b$. Modern machines have an instruction for finding the most significant set bit of a number in constant time. Thus, the lowest common ancestor of $x_1$ and $x_2$ can be found in constant time.

### 3.2.2 Solving range reporting

With a solution to the ball-inheritance problem, Chan et al. [2] propose the following:

**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 $\mathcal{O}(\lg\lg n + (1 + k) \cdot \tau)$.*

The ball distribution scheme of this data structure is the same as the Ball Inheritance Search of section 3.1.2. 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 RMQ 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.

In order to support predecessor (and successor) search for the y-rank in the data structure, the rank space predecessor search data structure is added to the tree. This data structure works on an array of the y-ranks, which is already sorted. 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. 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 node which has a predecessor structure in order to perform the search there. Searching takes $\mathcal{O}(\lg\lg n)$ time plus $\mathcal{O}(1)$ queries to the ball-inheritance structure. The ball-inheritance structure is exactly the oracle machine that the rank space predecessor search needs: At any given node the balls are sorted by their y-rank and given the identity (the index) of a ball at that node, it can look up the y-coordinate by decoding the y-rank to a leaf storing a point. 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 $1, 2, ..n$ so we can easily locate a given range in the 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 $LCA$.

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 $x_1$ and the rank space predecessor of $x_2$. We call them $\hat{x}_1$ and $\hat{x}_2$. 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 whose subtree contains 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.

2. From the least common ancestor of $\hat{x}_1$ and $\hat{x}_2$, we walk at most $\lg \lg n$ steps up to find the nearest parent with a predecessor search structure for the y-rank. We then translate $y_1$ and $y_2$ into rank space coordinates. We then walk down to the LCA again while maintaining the $\hat{y}_1$ and $\hat{y}_2$ at each step. The rank space coordinates are then translated into both the left and right child of the LCA. At each of the two children of the least common ancestor, this range indicates which balls have a y-coordinate between $y_1$ and $y_2$, i.e. which leaves that stores a point with a y-coordinate between $y_1$ and $y_2$. This step is illustrated on figure 3.3

3. We now descend into the right child of $LCA(\hat{x}_1, \hat{x}_2)$ and use the range minimum query structure at this node to the find the index $m$ (the y-rank) of the point with the smallest x-rank in the range $[\hat{y}_1, \hat{y}_2]$. The y-rank of a point at a node is exactly the identity of the ball going to the leaf storing the point. Knowing the identity of the ball we can use the ball-inheritance structure to follow the path to the leaf to find the actual x-coordinate of the point. If the x-coordinate is less or equal to $x_2$ 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. Otherwise we terminate. When this is done we apply the same concept to the left child of $LCA(\hat{x}_1, \hat{x}_2)$ using the range maximum query to find points greater or equal to $x_1$.

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 3.1.2. 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 at each child of the least common ancestor will be twice the number of results found at that child plus one. 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. For the final result found, two ranges are recursed into which reports no results.

Conceptually, $LCA(\hat{x}_1, \hat{x}_2)$ describes a point between $x_1$ and $x_2$, more precisely a point with an x-coordinate greater than all the points in the left child and smaller than all the points in the right child. 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. We can see an example of how step 3 works on figure 3.4. Looking at the right child of the

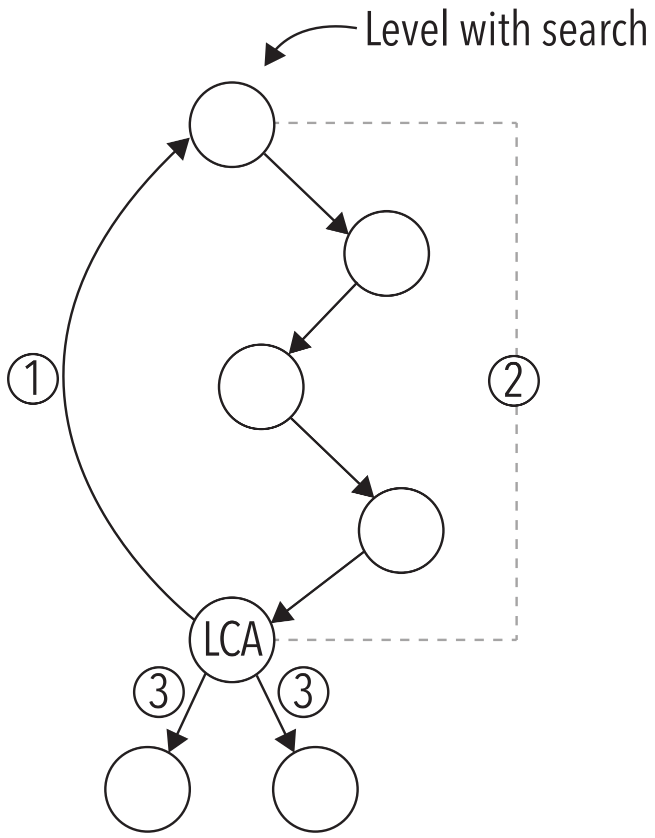


Figure 3.3: Illustration of how step 2 of the OBIS works.

figure, the leftmost point in the range $[\hat{y_1}, \hat{y_2}]$ will be found first. If that point has an x-coordinate less or equal to $x_2$ it will be part of the result. Then we recurse into the ranges $[p_7, p_8]$ and $[p_9, p_9]$. When the leftmost point in a range has an x-coordinate which is greater than $x_2$ then all other points in that range will also have an x-coordinate greater than $x_2$. Thus, the recursion in that range stops. Symmetrically, same applies for the left child of the least common ancestor.

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 $\mathcal{O}((1+k) \cdot \tau)$ time is used for walking from the children of the $LCA$ to the leaves solving the ball inheritance problem for the $k$ results. This gives a query time of $\mathcal{O}(\lg \lg n + (1+k) \cdot \lg^{\epsilon} n)$ to report $k$ points as results.

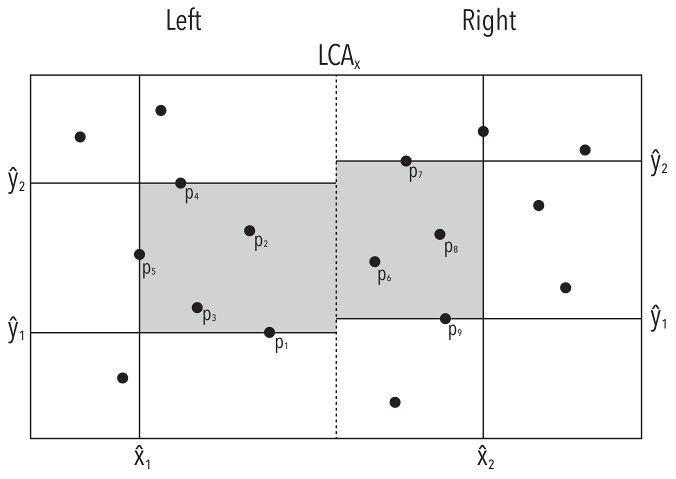


Figure 3.4: Illustration of how step 3 of the OBIS works. The dotted line divides the points in the left and right child of the least common ancestor.

## 3.3 Summary

The BIS and OBIS data structures both uses $\mathcal{O}(n)$ words of space. Both rely heavily on the ball-inheritance data structure in order to store and retrieve the actual coordinates of points. Both data structures support retrieval of points through the ball-inheritance data structure in $\mathcal{O}(\lg^{\epsilon} n)$ time. The main difference between the BIS data structure and the OBIS data structure is the rest of the query time: $\mathcal{O}(\lg n)$ for the BIS and $\mathcal{O}(\lg \lg n)$ for the OBIS. The OBIS data structure relies on more auxiliary data structures of greater complexity than the BIS data structure. Thus, the theoretical running time of a range query to OBIS is smaller than a range query to the BIS. In practise, it is safe to assume a range query to the BIS is faster than a range query to the OBIS. With much more code to be executed and using more advanced auxiliary data structures, the running time constant hidden in $\mathcal{O}(\lg \lg n)$ can be quite large. Also, the factor between $\lg n$ and $\lg \lg n$ is not that big. Given a very large dataset of $2^{64}$ elements as input, $\lg 2^{64} = 64$ and $\lg \lg 2^{64} = 6$, which has a factor $\sim 10$ difference.

Both the BIS data structure and the kd-tree use $\mathcal{O}(n)$ words of space. This is an attractive property when working on the RAM. Like all the main data structures mentioned in this thesis, the kd-tree and BIS data structure are output-sensitive. A range query to the kd-tree has a running time of $\mathcal{O}(\sqrt{n} + k)$ and a range query to the BIS data structure has a running time of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$. When $n$ grows, $\mathcal{O}(\sqrt{n})$ grows at a faster rate than $\mathcal{O}(\lg n)$. For each point reported as a result from a range query to the BIS data structure there is a cost of a factor $\mathcal{O}(\lg^{\epsilon} n)$ while each point reported as a result from a range query to the kd-tree has a cost of a factor $\mathcal{O}(1)$. The $\mathcal{O}(\sqrt{n})$ running time of the range query to the kd-tree is due to a pessimistic idea that a range query will overlap, but not fully include, a lot of regions in the kd-tree. So dependent on the shape of the range query to the kd-tree, the running time can vary a lot. The $\mathcal{O}(\lg n)$ part of the range query to the BIS data structure is due to the initial binary search and to follow the path from the root of the tree to $x_1$ and $x_2$. Thus, the running time of a range query to the BIS data structure will behave more stable (fluctuate less) than the running time of a range query to the kd-tree.

Another aspect of range querying is testing for emptiness. Given a range query $q = [x_1, x_2] \times [y_1, y_2]$, an emptiness test to a data structure will either answer "yes" or "no" to whether the range query contains any points. Thus, a range query can stop the moment a single result is found. The BIS data structure will be able to determine emptiness of a range query with no ball-inheritance queries. Hence, an emptiness query to the BIS data structure can be checked in $\mathcal{O}(\lg n)$ time. The initial binary search to find the rank space query might indicate that there are no points in $[x_1, x_2]$ or $[y_1, y_2]$. If either of the two rank space ranges are empty, the range query will also be empty. If the initial binary search does not indicate an empty range, the range query will continue normally. The moment a query to the ball-inheritance structure is made, the range is not empty and the emptiness test can report back without doing the actual ball-inheritance query. An emptiness query to the kd-tree can checked in $\mathcal{O}(\sqrt{n})$ time. The same argument as before applies: The query might overlap with a lot of regions which it does not fully contain, and within those regions none of the points are within the range. On the other hand, when a fully contained region is found the emptiness test can report back with a result. Given a range query $[x_1, x_2] \times [y_1, y_2]$, the BIS data structure will have a more stable running time for its emptiness test than the kd-tree. The running time of a query made to the kd-tree will fluctuate a lot depending on the shape of the query.

# Part II

# Practical

# Chapter 4

# Implementation

This chapter will explain some of the choices made as to the implementation of the BIS data structure.

## 4.1 Language

C++11 was chosen to implement the BIS data structure and the kd-tree. C++11 is a recent release of C++. C++11 combines the advantages of a modern programming language with the stability and support of a mature language which have been around for more than three decades. C++11 was chosen for several reason: It does not have garbage collection, it is a high level language with support for low level operation and contains libraries for everything needed in this project. The chrono header gives access to a high resolution clock to measure time. The vector class of the standard template library is very straight-forward container to work with and the underlying structure is a continuous block of memory making it ideal for cache purposes. The algorithm header gives access to convenience functions for generating and sorting data. The random header gives access to random numbers through a Mersenne twister engine with uniform integer distribution. The random numbers are used to generate the input data for the two data structures, thus giving a different data set every time. The data set is generated by two lists containing the numbers $[0, n-1]$ and using the standard library's shuffle function with the Mersenne twister engine.

## 4.2 Design choices

The theory describes the ball inheritance data structure as a binary tree with internal nodes and leaves. Each node has a bit vector representing the ball inheritance. In practice all the bit vectors at each level have been concatenated together to one, resulting in $\lg n$ bit vectors in all. Instead of being an actual entity, a node is just defined as which level it is from, where on that levels list its bit vector starts and how many balls its bit vector holds. Thus, a vector of bit vectors represents the ball inheritance tree. Since each ball only uses a single bit per level there would have been a lot of space wasted when nodes only held two or four balls in their bit vectors. By having a single unified bit vector per level we can pick one data type to work with independent of how many balls needs to be stored per node. The data type chosen is an unsigned fixed width integer type of 32 bits.

We are going to describe how to support constant-time succinct rank queries for a unified bit vector. First a checkpoint is added to every 32nd entry storing the amount of 1s seen in the bit vector so far. Since the bit vectors consist of entries of either 0 and 1 we only need count the amount of 1s. The rank of an entry with index $i$ storing a 1 bit is exactly the amount of 1s between index 1[1] and $i$. The rank of an entry with index $i$ storing a 0 bit is $i$ subtracted by the amount of 1s between index 1 and $i$. This is because we know that an entry not storing a 1 bit must be storing a 0 bit. A table is computed which given a 16 bit unsigned integer is able to answer how many 1 are in the binary representation of the integer. The table is a flat array and supports look-up in constant time.

Given an index $i$ in the bit vector, the closest checkpoint previous to $i$ is found. The data type storing the bits in the bit vector is a 32 bit unsigned integer. We can thus only retrieve data from the bit vector in blocks of 32 bits. We use a *binary and* to mask away the bits after $i$ and divide the 32 integer into two 16 bits integers. Using the precomputed table from before, we look up how many 1s the binary representation of the two 16 bit integer contain. The sum of the major checkpoint and the two table look-ups is the amount of times a 1 occurs before index $i$ in the bit vector.

However, we are seldom interested in knowing how many 1s there are in a bit vector between index 1 and $i$. We want to know how many 1s there are between $j$ and $i$, where $j$ is the start position of a node and $i$ is an entry in that node. We know that with the ball inheritance structure, each node gives each of its children half of its balls. Thus, half of the entries in the bit vector of a node are 1s. And thus, if $j$ is the start position of a node in the unified bit vector, there will be $\frac{j}{2}$ entries in the bit vector between 1 and $j$ which are 1s. Since one of the ingredients for a virtual node is the start position in the unified bit vector, we already know $j$ when looking up the rank of $i$. Using the unified bit vectors instead of an actual tree then poses no problems.

The source code for the BIS data structure will be made available online at http://www.madsravn.dk/BIS.

[1]Arrays are 1-based

# Chapter 5

# Analysis

*"The trouble with writing fiction is that it has to make sense, whereas real life doesn't"*

— Iain M. Banks

The purpose of this chapter is to compare the BIS data structure to the kd-tree. We are going to perform a variety of experiments on the BIS data structure and the kd-tree in order to determine the run-time properties of both, mainly looking at when the BIS data structure performs better than the kd-tree.

When performing the experiments, random data will be generated and given as input to the data structures. Both data structures will be given the same random data. This way they operate on the some data and the comparison will be fair. The random data is generated by making two lists, $X$ and $Y$ with the integers $[0, n-1]$ and shuffling them both randomly. The $n$ points given to the data structures as input are found by taking the $i$th entry of both $X$ and $Y$ and generating a point with those coordinates. This ensures that all x-coordinates are unique and all y-coordinates are unique. When running a specific experiment, different data sets will be generated and given as new input to the data structures such that an experiment is not only performed on a single data set. The different experiments will check how the shape of a given search query impacts the running time of the query to the data structures. When the shape and size of the search query has been determined, the search will performed with a different displacement on the x-axis and y-axis such that the queries are performed all over the data structure and not only in a best-case or worst-case position. The search query is performed a lot of times and then the average time per query is returned.

We are mainly going to focus on two different kinds of experiments. First we are going to test how a search query shaped like a square performs in both the BIS data structure and the kd-tree. This type of query will be good for the kd-tree since it will get to fully include many regions. In the second experiment the configuration of the search query is going to one where the worst-case scenario for the kd-tree happens. It will cover the entirety of the search area in a thin slice either in a vertical or horizontal direction. In the third section we are going to look closer at how much better the search query to the BIS data structure compares to the search query to the kd-tree when the search query is a vertical or horizontal slice and $k \leq 200$. The limit of 200 is chosen because it is small and a reasonable upper bound on the amount of results for human interaction.

The experiments have been performed on data structures with data sets of size $2^{\lg n}$ where $\lg n = [17, 25]$. 17 was chosen as the smallest because it would still be big enough to show something interesting on the graphs. 25 was chosen because the current initialization of the BIS data structure requires a bit of work and thus takes up nearly all of the main memory. Future work includes an idea for a faster and less memory requiring setup phase.

In all the experiments we have chosen $B = \lceil \frac{1}{2} \lg^{\frac{1}{3}} n \rceil$. Recall from section 3.1.2 that we must have $B = \Omega(\lg^{\epsilon} n)$ to obtain linear space. Thus, we have chosen $\epsilon = \frac{1}{3}$. Section 3.1.2 also states that $B$ is responsible for the big jumps in the ball inheritance structure: where the jumps should be placed and how big they should be. Unless otherwise stated the graphs in this chapter show results from a BIS data structure configured with $B = \lceil \frac{1}{2} \lg^{\frac{1}{3}} n \rceil$. When $n \in [2^{17}, 2^{25}]$, $B$ will be 2. We will use the variables $k$ and $size$ interchangeably in this analysis. The reason for this will be explained later.

The experiments were performed on a machine with an Intel i7-3770 CPU with 3.40GHz and with 32 GB RAM. The machine was running Ubuntu 14.04.2 LTS with clang version 3.4. The machine was provided by MADALGO.

## 5.1 Square search queries

This section will show the running time of square search queries to both data structure. With this configuration we expect to see that kd-tree performs better than the BIS data structure. We are interested in seeing just how much worse the BIS data structure performs.

### 5.1.1 Setup

The kd-tree has a query time of $\mathcal{O}(\sqrt{n} + k)$. The $\sqrt{n}$ part is based on a pessimistic notion that an edge of a query will pass through the entire search area of the kd-tree. This is not always the case. We also note that when $k > \sqrt{n}$, $k$ will dominate the expression and thus the query time will be linear in the output size. In order to fairly compare the running time of a search query to both data structure, we are going to generate queries finding the points within a rectangle with the area of $\sqrt{size} \cdot \sqrt{n} \times \sqrt{size} \cdot \sqrt{n}$ where size will increase. As previously described, this search query will be made with random displacements in order to query arbitrary places in the structures. So given two random numbers $x$ and $y$ a query will be $q = [x, x + \sqrt{size} \cdot \sqrt{n}] \times [y, y + \sqrt{size} \cdot \sqrt{n}]$. A query of this shape will not invoke the worst-case scenario for the kd-tree and will thus give an idea of how the BIS data structure performs in contrast to the kd-tree under circumstances where the kd-tree performs well. We thus expect the kd-tree to perform better than the BIS data structure in this experiment.

The points generated for the data structures lie in the range of $[0, n-1] \times [0, n-1]$, which gives an area of $n^2$. If the search query has an area of $A$, each point has a $\frac{A}{n^2}$ chance of being in that search query. With $n$ points we thus expect to find $n \cdot \frac{A}{n^2}$ points in a search query with the area of $A$. In this experiment we set $A = \sqrt{n} \cdot \sqrt{size} \times \sqrt{n} \cdot \sqrt{size}$, where $n$ is constant to the data structure and $size$ will increase during the experiment. We then expect to find $n \cdot \frac{A}{n^2} = \frac{n \cdot n \cdot size}{n^2} = size$ points. This obviously depends a lot on how the points are distributed in that specific case. When generating 10 different data sets for the data structures in the experiments and picking the displacements for the search query at random each search, we will expect the average amount of points returned by the search query $q = [x, x + \sqrt{size} \cdot \sqrt{n}] \times [y, y + \sqrt{size} \cdot \sqrt{n}]$ to be $size$.

### 5.1.2 Data

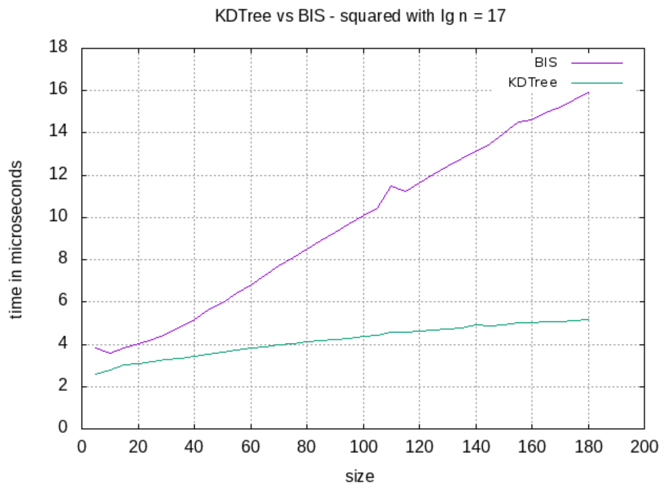
On figure 5.1a and figure 5.1b we see that the running time of a query to the BIS data structures, for a fixed $n$, increases linear to the amount of points returned. This agrees with the theoretical running time of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$. The theoretical $\mathcal{O}(\lg^{\epsilon} n)$ is the worst-case amount of jumps from a any given node to a leaf. On figure 5.1b we notice that around $size \approx 150$ the graph changes its slope. The slope decreases which means that the running time per point decreases. This means that the average amount of jumps per point reported can change, but will naturally always be bounded by the worst-case of $\lg^{\epsilon} n$. Other factors may have a say in the change of slope as well. Some of these factors will be mentioned in section 5.5.
As expected, the average amount of points reported from a lot of search queries with $q = [x, x + \sqrt{size} \cdot \sqrt{n}] \times [y, y + \sqrt{size} \cdot \sqrt{n}]$ were $size$ or $size - 1$. When $\sqrt{size} \cdot \sqrt{n}$ is a floating point number it will be rounded down to the nearest integer.

The graphs on figure 5.1a and figure 5.1b show the time of a search query to the BIS data structure compared to the time of a search query to the kd-tree. The shape of the search query is a square. The variable $size$ increments in levels of 5 per iteration of the experiment and will have a maximum of $\frac{\sqrt{n}}{2}$. The x-axis of the graphs describes the $size$ variable in the expression $A = \sqrt{n} \cdot \sqrt{size} \times \sqrt{n} \cdot \sqrt{size}$. Examining figure 5.1a we see that when the shape of the search query is a square, the search query to the kd-tree is always performing better than the search query to the BIS data structure. Looking closer at the figure we notice that while the search query to the BIS data structure performs worse, it is not that bad. At $size = 180$ we have window of size $\sqrt{2^{17}} \cdot \sqrt{180} \times \sqrt{2^{17}} \cdot \sqrt{180} = 4857.26 \times 4857.26$. The time to perform the search query on the BIS data structure at $size = 180$ is 15.9 microseconds, while the search to the kd-tree takes 5.2. This search query includes 180 points. This is a relatively big query and the time to perform the search query on the BIS data structure is only a factor 3 worse than the time of the search query on kd-tree.

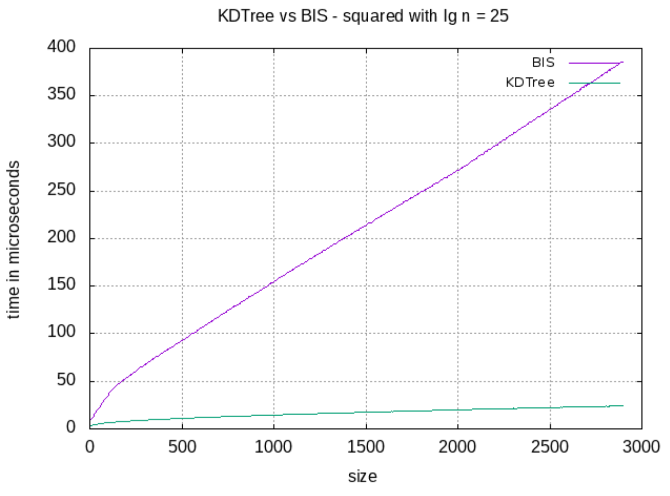
Examining figure 5.1b with $n = 2^{25}$ the largest search window has become quite large with a size of $\sqrt{2^{25}} \cdot \sqrt{2895} \times \sqrt{2^{25}} \cdot \sqrt{2895} = 311673 \times 311673$. The biggest window now returns 2895 points. A search query with this size to the BIS data structure and the search query to the kd-tree has a factor $\frac{386}{24} = 16.1$ difference. This a quite big difference in the search time, but it is also a big search query.

It is obvious from the graphs that the kd-tree performs better when the search queries are square windows. Since $\lg^{\epsilon} n$ and $\lg n$ are fixed, the time of a search query to the BIS data structure grows linearly to the size of the window, just like a query to the kd-tree, but with a bigger slope. This means we will not run into an unexpected growth when asking for more points from the BIS data structure.

The biggest search query at $n = 2^{17}$ is $\sqrt{180} \cdot n \times \sqrt{180} \cdot \sqrt{n}$. We now fix $size = 100 \leq 180$ and see how the ratio between the search query to the BIS data structure and kd-tree will evolve when $n$ grows. We see this on figure 5.2. The graph is growing and thus the running time of a search query to the BIS data structure grows faster than same search query to the kd-tree when $n$ is growing. This is to be expected since we pay $\mathcal{O}(k \cdot \lg^{\epsilon} n)$ compared to $\mathcal{O}(k)$. The ratio grows from 2.3 to 5.5 when $\lg n$ grows from 17 to 25. This is not a huge increase in ratio taking into account that $n$ grows by a factor of $\frac{2^{25}}{2^{17}} = 2^8 = 256$.



(a) $n = 2^{17}$ with $\sqrt{n} = 362$.



(b) $n = 2^{25}$ with $\sqrt{n} = 5792$.

Figure 5.1: Square search on BIS and kd-tree.

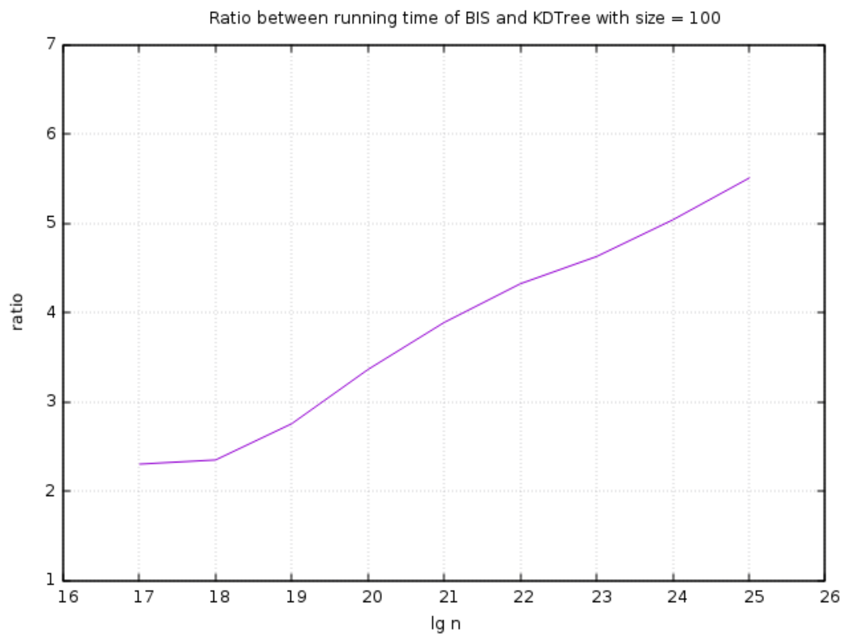


Figure 5.2: Ratio between a square search query to the kd-tree and BIS data structure with constant $size = 100$. Ratio describes how much better the kd-tree performs.

## 5.2 Vertical and horizontal slices

In this section we are going to compare the performance of the BIS data structure and the performance of the kd-tree when the search query is in the shape of a *slice*. We define a *slice* to be a search query which covers the entirety of the search space in one dimension while only covering a small part of the search space in the other dimension. Since a slice is a search query with the form of either $q_h = [0, n-1] \times [y_1, y_2]$ or $q_v = [x_1, x_2] \times [0, n-1]$ we omit $[0, n-1]$ and define the *size of the slice* to be $|y_2 - y_1|$ or $|x_2 - x_1|$. A slice which extends through the entirety of the x-dimension is called a *vertical slice*. A slice which extends through the entirety of the x-dimension is called a *horizontal slice*. Both the x-coordinates and y-coordinates of the points generated are unique which means that a slice of size $k$ will always return $k$ points as the result.

### 5.2.1 Setup

A search query to the kd-tree has a query time of $\mathcal{O}(\sqrt{n}+k)$ and a search query to the BIS data structure has a query time of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$. We expect the query time of a slice to the BIS to be faster than the query time of a slice to the kd-tree when $k$ is small. When $k$ grows the query to the kd-tree will eventually become faster. When increasing the size of a slice, we expect the query time of the two search queries to be roughly equal at $k \approx \frac{\sqrt{n}-\lg n}{\lg^{\epsilon} n - 1}$. This originates from $\sqrt{n} + k = \lg n + k \cdot \lg^{\epsilon} n \Leftrightarrow k = \frac{\sqrt{n}-\lg n}{\lg^{\epsilon} n-1}$. We call this theoretical intersection point between the running times for $k_t$. The $\mathcal{O}(\lg^{\epsilon} n)$ bounds the amount of jumps needed to perform in order to find the identity of a leaf given the identity of a ball in the ball inheritance structure. $\lg^{\epsilon} n$ intuitively describes the amount jumps a ball uses to travel from a node to a leaf. Thus, in this analysis $\lg^{\epsilon} n$ will be the average amount of jumps per result, as measured by the results in the experiments. For example, if the ball-inheritance structure has used 10 jumps to locate 4 leaves, $\lg^{\epsilon} n = \frac{10}{4} = 2.5$. We use this measured $\lg^{\epsilon} n$ instead of the theoretical $\lg^{\epsilon} n$ because the theoretical is used to bound the amount of jumps, not count them. Thus, in $k_t$ we will let $\lg^{\epsilon} n$ describe the average amount of jumps taken by each point reported back as a result of the query, measured by the experiments.

Recall that the BIS data structure treats the two dimensions very differently. Given a rank space search query $q = [x_1, x_2] \times [y_1, y_2]$, the search query will find the least common ancestor of $x_1$ and $x_2$. From there, it will find the path to both $x_1$ and $x_2$ and all leaves between them will be in the range $[x_1, x_2]$. The search algorithm now has to determine which of these leaves contain a point with y-coordinate in $[y_1, y_2]$. This is done by using the ball inheritance structure from each of the fully contained nodes which were found on the path from the least common ancestor to $x_1$ and $x_2$.

A horizontal slice includes all x-coordinates of the search space, and thus the least common ancestor of $[x_1, x_2]$ will be the root of the tree. The path from the least common ancestor to $x_1$ will only go left which means each level will have one fully contained node. The path from the least common ancestor to $x_2$ will only go right, also yielding one fully contained node per level. The experiments measure the difference in performance between the BIS data structure and the kd-tree when the search query is a slice. A slice will start out being very small, $k = 5$. Thus, many of the fully contained nodes will have no ball inheritance work when the slice is a horizontal slice. As the slice grows, eventually more and more node will have one or more balls to follow. The amount of ball inheritance each node is responsible for varies a lot, and therefore the ball inheritance will become somewhat sporadic. But the nature of the ball distribution asserts that if node has two or more balls for ball inheritance, these balls will be right next to each other.

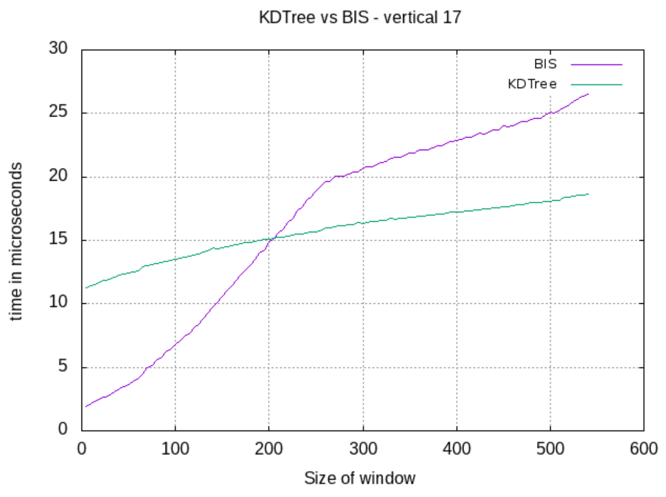
On the other hand we have the vertical slices. The vertical slice includes all y-coordinates of the search space, and thus the location of the least common ancestor of $[\hat{x_1}, \hat{x_2}]$ varies a lot dependent on the search query. But the nodes which are marked as fully contained on the path from the least common ancestor to $\hat{x_1}$ and $\hat{x_2}$ will all only contain leaves with points with y-coordinates in $[y_1, y_2]$ which means we have to do ball inheritance on all the balls belonging to fully contained nodes. Thus, the ball inheritance in vertical slices are much more batched together. Comparing a vertical and horizontal slice of the same size, the vertical slice will have good chances of being faster than the horizontal slice. With a least common ancestor closer to the leaves, the ball inheritance in the vertical slice will have to jump from a lower level than the ball inheritance in the horizontal and the balls are more batched together in the vertical.

We are going to look at how well the vertical and horizontal slices perform on both the BIS data structure and the kd-tree. We are interested in seeing how big the slices can become before the search query to the BIS data structure performs worse than the search query to the kd-tree. We are going to look at the vertical slices first. Below are some graphs showing the running time of a search query to both the BIS and the kd-tree dependent on the size of the slice.

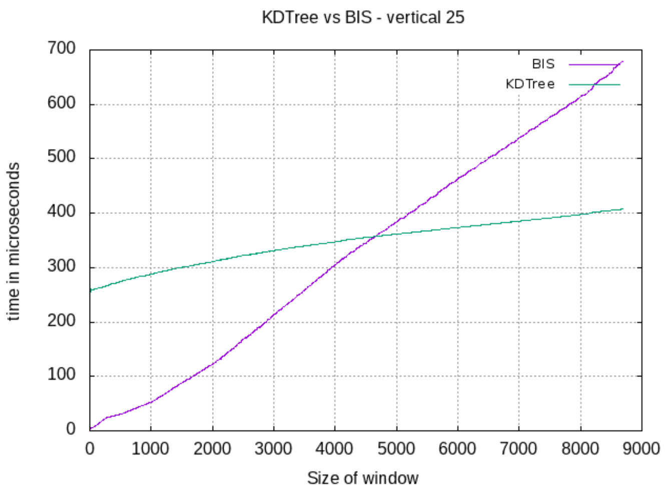
### 5.2.2 Vertical slices

Figure 5.3a and figure 5.3b show the performance of a search query to the BIS data structure compared to a search query to the kd-tree, where the search query is a vertical slice. Figure 5.3a shows that the BIS data structure is faster than the kd-tree up until the size of the slice is 200. That is quite significant. Not only is the BIS data structure faster when $k \leq 200$, but at $k = 100$ it is approximately twice as fast as the kd-tree. There is a sudden change in the slope of the graph at around $k \approx 250$ which will be addressed in section 5.5. On figure 5.3b we see that the BIS data structure is faster than the kd-tree up to a size of 4660. That is big query. At $k = 100$ the BIS data structure is 27 times faster than the kd-tree. At $k = 200$ the BIS data structure is 14 times faster than the kd-tree. These are significant differences in performance.

We just pointed out that the graphs on figure 5.3a intersect at $k = 200$ and the graphs on figure 5.3b intersect at $k = 4660$. Figure 5.4a shows the size ($k$) of the slice at the point of intersection between the running-time of a search to the BIS and the kd-tree for each $n$ tested. Recall the theory above where it



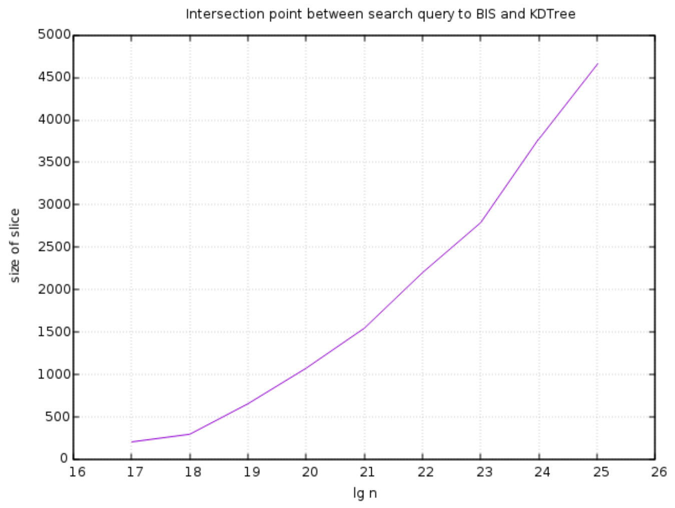
(a) $n = 2^{17}$



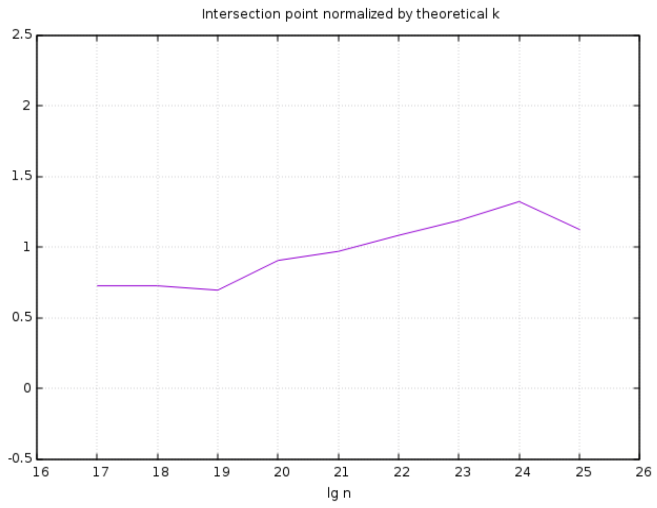
(b) $n = 2^{25}$

Figure 5.3: Vertical slice on BIS and kd-tree.

was described how the intersection point should theoretically be $k_t = \frac{\sqrt{n} - \lg n}{\lg^{\epsilon} n - 1}$. Figure 5.4b shows $\frac{k_m}{k_t}$, where $k_m$ is the amount of results measured by the experiments. The graph is plotted using $\lg^{\epsilon} n$ as the average jumps per result as measured by the experiments. We want this graph to as close to a flat line as possible. It does not really matter where on the y-axis the flat line lies, because both the numerator and denominator of $k_t$ has hidden constants in the O-notation. We notice that the graph lies steadily around 1 meaning that it is a stable relationship. Figure 5.4a has a exponential tendency. That is to be expected since the x-axis is $\lg n$ which means that $n$ grows by a factor of 2 each step on the x-axis and then the point of intersection between the two running times increases by approximately $\sqrt{2}$. This is because of the $\mathcal{O}(\sqrt{n})$ part from the running time of a search query to the kd-tree. Obviously, $\sqrt{n}$ grows by a factor of $\sqrt{2}$ when $n$ grows by a factor of 2, which is exponential. A search query in the shape of a slice is the worst-case scenario for the kd-tree.



(a) Point of intersection between BIS and kd-tree.



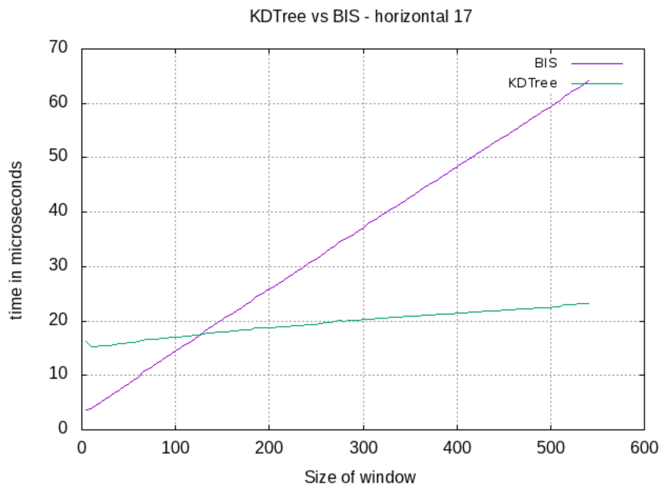
(b) Point of intersection normalized by theoretical $k$

Figure 5.4: Vertical slice on BIS and kd-tree.

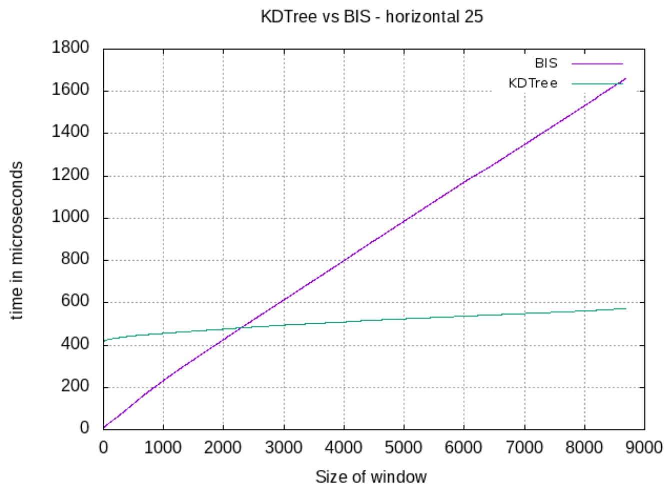
### 5.2.3 Horizontal slices

We are now going to look at the graphs for the horizontal slices. Recall that when a search query is a horizontal slice, the least common ancestor will be the root of the tree. In $[x_1, x_2]$, $x_1$ will be the leftmost leaf and $x_2$ will be the rightmost leaf. From the root to $x_1$ and $x_2$ there will many fully contained nodes, 2 per level to be exact, which means there will be many different nodes performing small amount of balls inheritance look-ups. The y-coordinates of the points are uniformly distributed. Thus, unlike the vertical shape, we cannot say anything definitively about where the ball-inheritance takes place.

Figure 5.5a and figure 5.5b show the performance of a search query to the BIS data structure compared to a search query to the kd-tree, where the search query is a horizontal slice. Figure 5.5a shows that the BIS data structure is faster than the kd-tree until the size of the slice becomes 125. While not as good as the vertical slice, it is still a decent size. At $k = 100$ the BIS data structure is only a factor 1.2 faster than the kd-tree. On figure 5.5b we see that the BIS data structure is faster than the kd-tree up until $k = 2290$. At $k = 100$ the BIS data structure is 13 times faster than the kd-tree, and at $k = 200$ it is 8 times faster. Again, this is not as good as the vertical slice, but it is still quite significant. Given a query with 200 results in a horizontal slice, the BIS data structure outperforms the kd-tree by a factor of 8. It is obvious for both the horizontal and the vertical slice, that the smaller the window, the bigger the ratio between the performance of the two data structures.



(a) $n = 2^{17}$

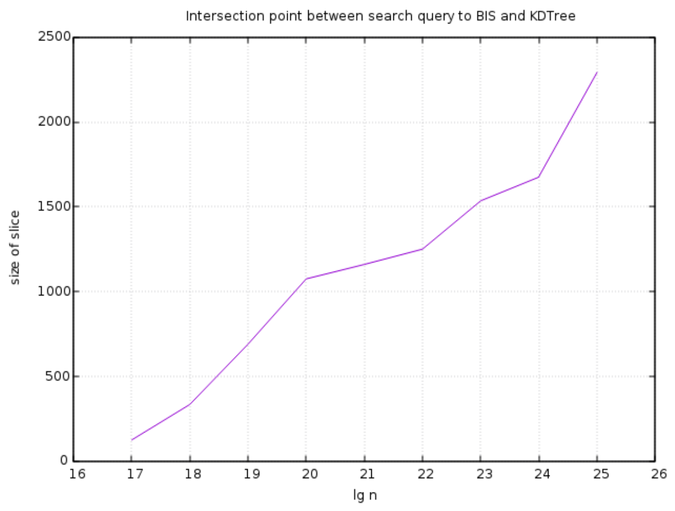


(b) $n = 2^{25}$

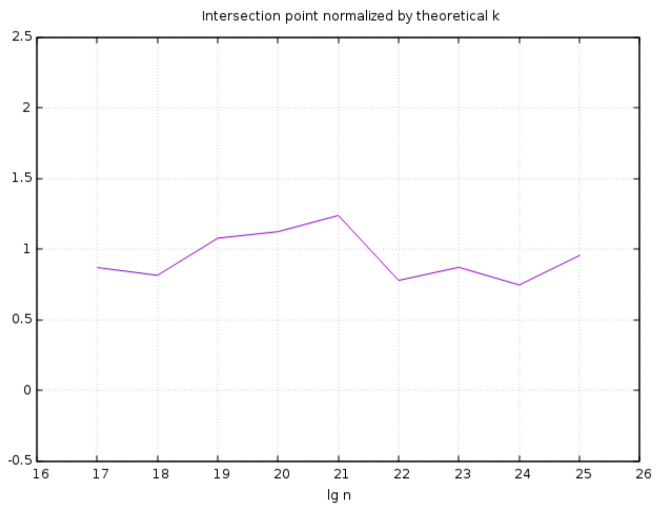
Figure 5.5: Horizontal slice on BIS and kd-tree.

Figure 5.6a shows the size $(k)$ of the slice at the point of intersection between the running-time of a search to the BIS and kd-tree for each $n$ tested. Figure 5.6a is a little more messy than figure 5.4a. At figure 5.6b the graph crosses above and below 1, but keeps rather close to 1. Again, we are looking for a graph that is as flat as possible in order to show that is a stable relationship.

In the big perspective, this graph certainly shows a stable relationship around 1. Just as before, the $\lg^{\epsilon} n$ in $k_t$ is the average amount of jumps per result measured by the experiments. While the horizontal slice is not as good as the vertical slice, it is still very good. It definitely outperforms the kd-tree up to a good size.



(a) Point of intersection between BIS and kd-tree.

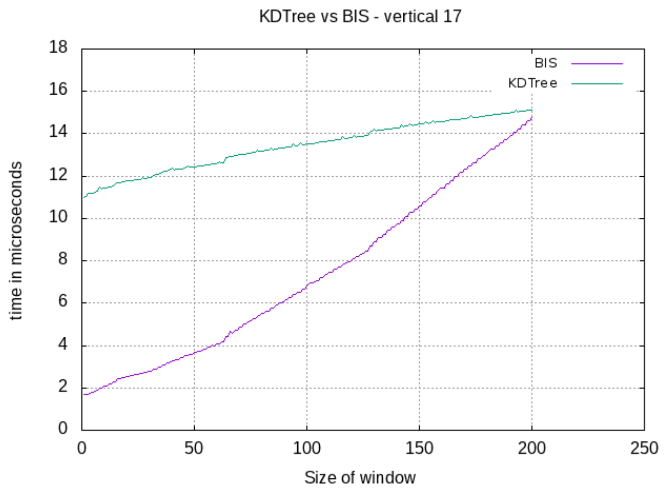


(b) Point of intersection normalized by theoretical $k$

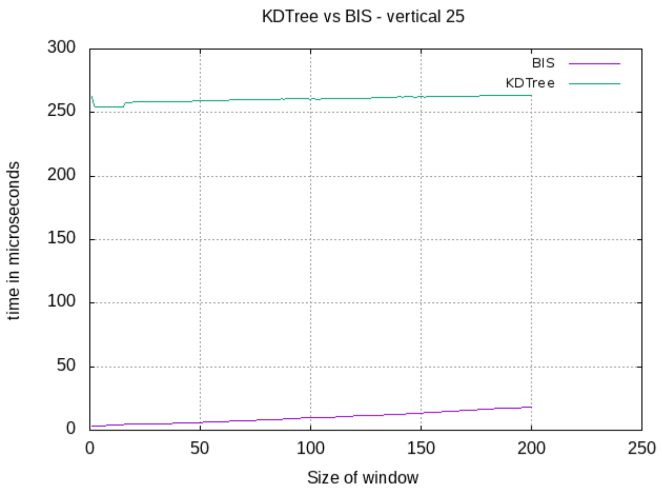
Figure 5.6: Horizontal slice on BIS and kd-tree.

## 5.3 Slices with small $k$

In this section we are going to look at slices with $k \leq 200$. We will only focus on vertical slices. We are interested in seeing how much faster the BIS data structure is than the kd-tree when looking at amount of results which seems reasonable to user interaction. Figure 5.7a figure 5.7b show the performance a search query to the BIS data structure compared to a search query to the kd-tree, where $k \leq 200$ for vertical slices.



(a) Data set of size $n = 2^{17}$.

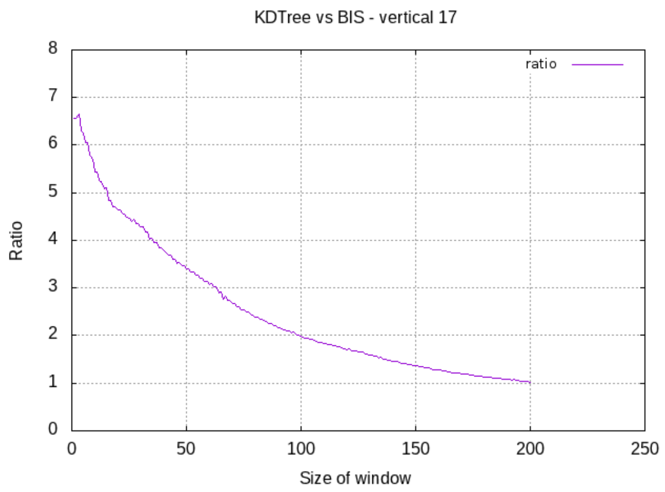


(b) Data set of size $n = 2^{25}$.

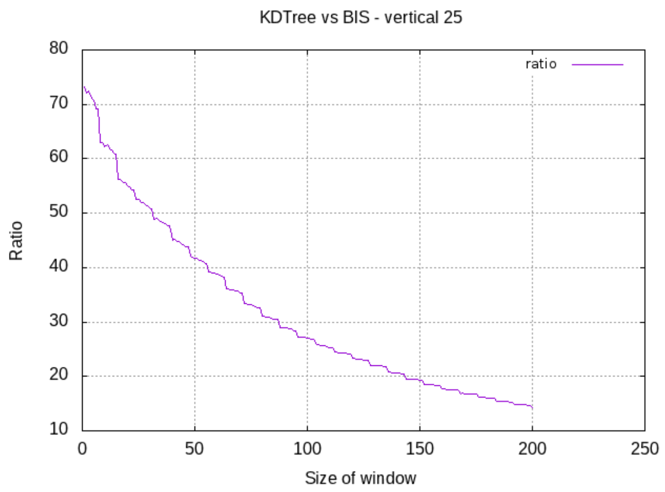
Figure 5.7: Vertical slice on BIS and kd-tree.

The running time of a slice to the BIS data structure where $k \leq 200$ is noticeably better than the running time of a slice to the kd-tree. There is a great gap between the graph of the two running-times. The figures show that the BIS data structure always performs better when the size of the slice is below 200. Seeing this tendency, we are then also interested in testing just how much better the BIS data structure performs. We are going to measure this by looking at the ratio between the performance of the BIS data structure and the performance of the kd-tree. We see this on figure 5.8a and figure 5.8b. The ratios describes how much better the BIS data structure is than the kd-tree in terms of running time: Bigger is better. As mentioned in section 5.2, the ratio between the performance of the BIS data structure and the kd-tree is strictly decreasing as the size of the slice is increasing. Thus, the smaller the slice, the bigger the ratio between performances of the data structures. On figure 5.2 we plotted the ratio between a square query to the kd-tree and a square query to the BIS data structure in order to find out how much better the kd-tree performed. Now the roles are reversed.

Looking at figure 5.8a and figure 5.8b we notice that the ratio between the two running times at $size = 100$ increases from 2 to 27 as $n$ increases from $2^{17}$ to $2^{25}$. A factor of 27 is the difference between the vertical slice to the BIS data structure and a vertical slice to the kd-tree with $size = 100$. With $size = 200$



(a) Data set of size $n = 2^{17}$.



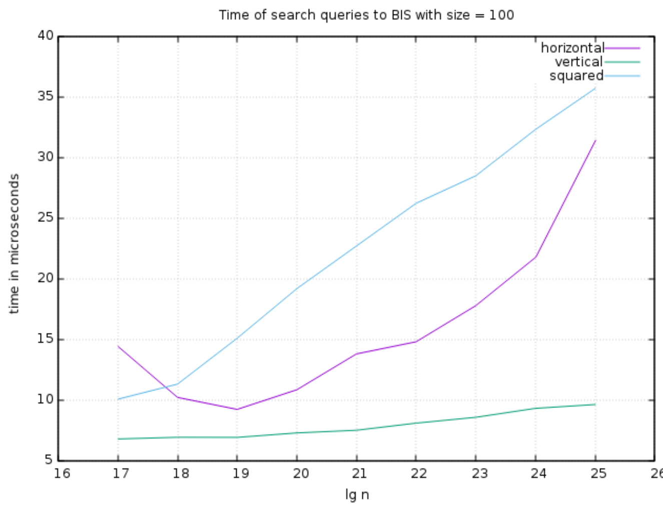
(b) Data set of size $n = 2^{25}$.

Figure 5.8: Ratio between running time of slice on BIS and kd-tree with fixed $n$.

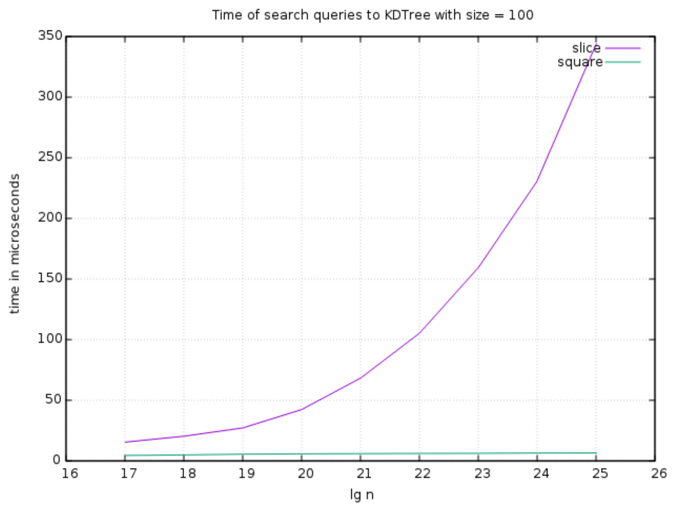
the ratio between the two data structures increases from just above 1 to 14. A factor 14 is a big difference. We see that when the size of the slice is constant and $n$ is growing, the ratio between a slice to the BIS data structure and the kd-tree increases. On figure 5.8b we also notice that when $1 \leq k \leq 50$ the ratio is between 73 and 42. Thus, given a big data set and a small vertical slice the BIS data structure will significantly outperform the kd-tree. This fits well with the theory where the kd-tree has the worst-case of $\mathcal{O}(\sqrt{n})$ and the BIS data structure is more stable in respect to its shape. We will investigate this further.

Looking back to the section about the square windows we remember that a window with the dimensions $\sqrt{n} \cdot size \times \sqrt{n} \cdot \sqrt{size}$ is expected to return $size$ points as result. This was based on the area of area of the search query. Thus, we can rewrite the expression as follows: $\sqrt{n} \cdot size \times \sqrt{n} \cdot \sqrt{size} = \sqrt{n} \cdot \sqrt{n} \times \sqrt{size} \cdot \sqrt{size} = n \times size$. This is exactly what vertical or horizontal slice looks like. Thus, we know that these two types of queries are expected to return the same amount of points and we can therefore compare the square search query with a horizontal or vertical slice to see how much of an impact the shape of the search has for the BIS data structure. Since both shapes of a search query is expected to return $size$ points when the size of the search query is $size$, which is why we can use the variables $k$ and $size$ interchangeably in this analysis.

We pick $size = \{50, 100, 150\}$ from both the square experiments and the slice experiments to see how changing the shape of the query affects the running time of the query to the BIS data structure. The graphs on figure 5.10a, figure 5.9a and figure 5.10b show some similarities. The vertical slice is clearly the fastest. The horizontal slice performs worse than the vertical slice. The squared search query seems to be the worst-case scenario for the BIS data structure. But looking at only the squared search across the three graphs we see that it growing with a rather linear tendency. We would expect as much from the theory because if we fix $k$ in $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ we only have $\lg n$ and $\lg^{\epsilon} n$ growing. Since $\lg n$ increases by 1 and $\mathcal{O}(\lg^{\epsilon} n)$ is a bound for the amount of jumps performed, this ties rather well to the theory. Looking at these three graphs, we suspect that the square search is the worst case of a search query to the BIS data structure, while both the horizontal and vertical slice searches are special cases performing even better. Thus, when we fix $k$ and let $n$ grow, the time required to find $k$ points with a search query to the BIS data structure seems to be bound by the running time of the square search.

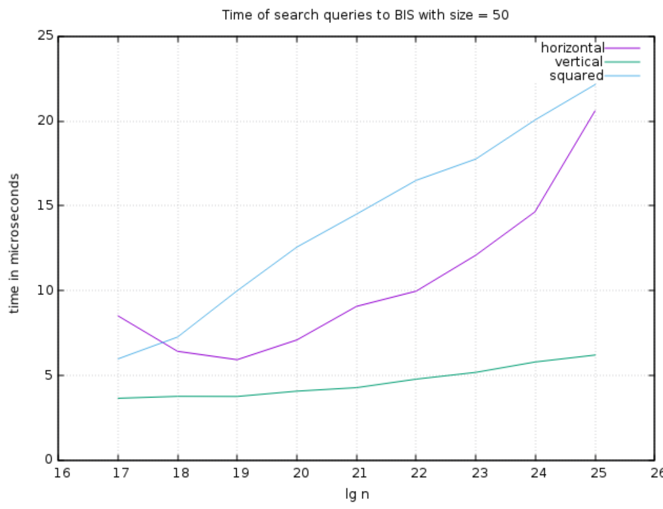


(a) Queries to the BIS data structure with $size = 100$.

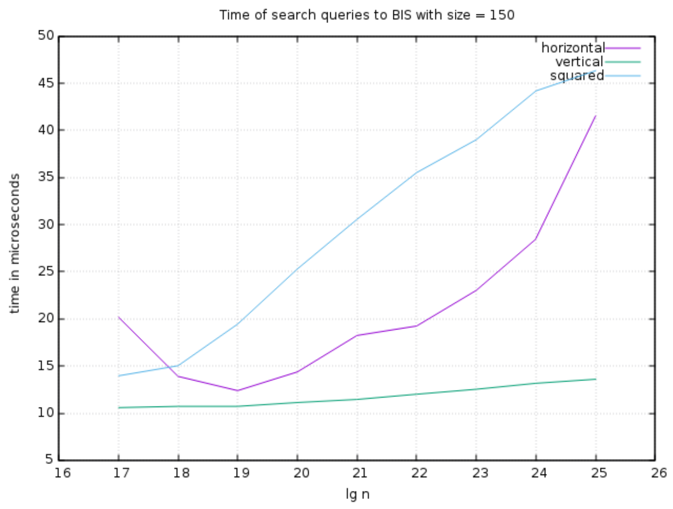


(b) Queries to the kd-tree with $size = 100$.

Figure 5.9: Comparison of shapes on BIS and kd-tree.



(a) Queries to the BIS data structure with $size = 50$.



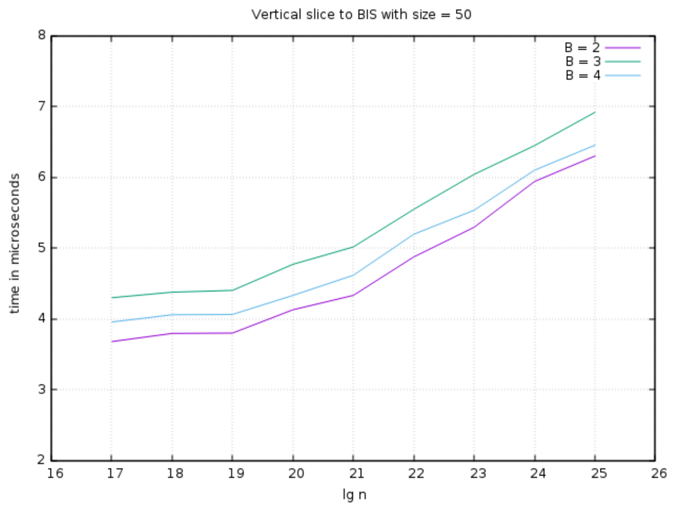
(b) Queries to the BIS data structure with $size = 150$.

Figure 5.10: Comparison of shapes on BIS.

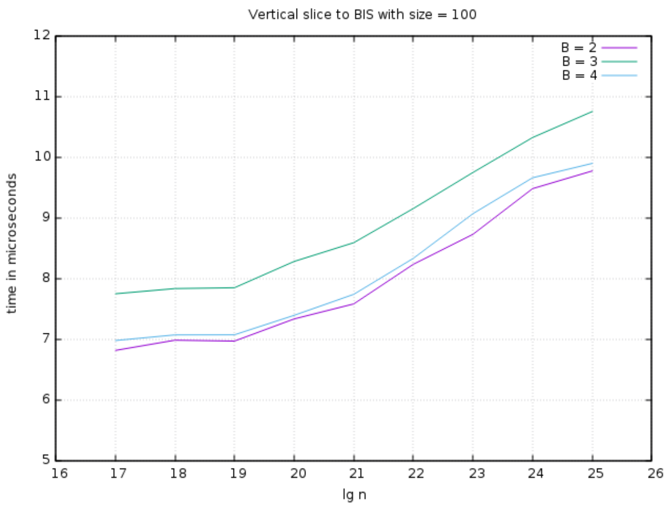
We look at figure 5.9b in order to confirm that changing the shape from a square to a slice impacts the running time of the search query to kd-tree very much. Comparing figure 5.9a and figure 5.9b we see that the difference between the two shapes of search queries are much, much bigger on the kd-tree than on the BIS data structure. The biggest ratio between the vertical slice and the square search on figure 5.9a is around 4 at $\lg n = 25$. The biggest ratio between the slice and square search on figure 5.9b is around 53 at $\lg n = 25$. That is really a noticeable difference. Thus, when a search query is performed on the BIS data structure we can expect a much more stable running time. The BIS data structure is more resilient to changes in the shape of the search query than the kd-tree. Figure 5.10a and figure 5.10b shows the same tendency as figure 5.9a: A stable relationship between the square search and slice search with a maximum factor of 4 in difference, at $\lg n = 25$. Notice that the ratio between best-case and worst-case search queries on all four figures grow as $n$ grows. The ratio between the worst-case and best-case search queries to the BIS data structures grows much slower than the kd-tree.

## 5.4 Different BIS data structures with $B$ varying

So far we have only looked at the BIS data structure with $B = \lceil \frac{1}{2} \lg^{\frac{1}{3}} n \rceil$. For $n \in [2^{17}, 2^{25}]$ this is $B = 2$. We have also not yet mentioned the important aspect of how much main memory the BIS data structure uses compared to the kd-tree. In this section we are going to look at the BIS data structure for $n \in [2^{17}, 2^{25}]$ and $B = 2, 3, 4$ and show how much space it uses and compare it to the space used by the kd-tree. When $B$ grows in the BIS data structure, we expect space to drop, but the query time of a range query to grow. We are going to briefly look at how the BIS data structure performs with $B = \{3, 4\}$.



(a) Vertical query with $size = 50$.



(b) Vertical query with $size = 100$.

Figure 5.11: BIS with $B = \{2, 3, 4\}$.

Figure 5.11a and figure 5.11b show how $B$ impacts the running time of a vertical slice query to the BIS data structure with *size* fixed. Since $B$ directly impacts where in the ball-inheritance structure big jumps will be located and how big the jumps are, we expect the query time to be slower when $B$ grows. However, both graphs show that the query time of a vertical slice is faster when $B = 4$ than when $B = 3$. This is probably due to the big jump at level 4. But in general, when $B$ grows, the average amount of jumps per result is expected to grow. When $\lg^\epsilon n$ grows, the time to find $k$ results grows.

The most important reason why the range tree is not used as the standard range reporting data structure today is its space complexity. While we have shown theoretically that the space complexity of the BIS data structure is linear, we would also like to see it in practice. Figure 5.12 shows the actual space usage of the BIS data structure with $B = \{2, 3, 4\}$ and the space usage of the kd-tree. The size has been normalized by the amount of points the data structure holds. Thus, the normalized size of the kd-tree is 2, meaning that for each point in the kd-tree it uses $2 \cdot 32$ bits - one 32 bits integer for each coordinate. As we expected, the size of the BIS data structure decreases when $B$ grows.

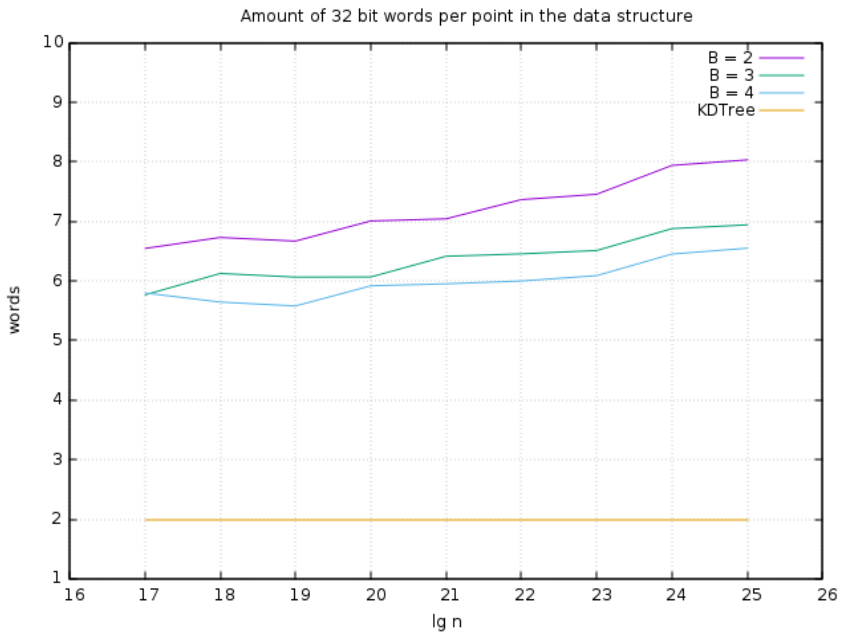


Figure 5.12: The normalized sizes of the BIS data structure with $B = \{2, 3, 4\}$ and the kd-tree.

An interesting thing to remember is that the size of the kd-tree is entirely dependent on the data type used to represent each coordinate. In our experiments we have used a 32 bit unsigned integer. If we were to change that to a 64 bit unsigned integer, the size of the kd-tree would increase by a factor of 2. As the kd-tree, the BIS data structure uses 2 words per point to store the coordinates. The BIS data structure also uses 1 word per point to store the y-coordinates in a sorted list as to allow for binary search to find $\hat{y_1}$ and $\hat{y_2}$. The rest of the BIS data structure are no dependent on the data type used to represent the coordinates. Thus, changing the data type from a 32 bit integer to a 64 bit integer would only increase the total space usage by 3 32-bit integers per point. On figure 5.12 the size of the BIS data structure with $B = 2$ and $\lg n = 25$ is 8. Increasing that to 11 would increase the space usage of the BIS data structure by a factor of $\frac{11}{8} = 1.375$.

The BIS data structure with $B = \lceil \frac{1}{2} \lg^{\frac{1}{3}} n \rceil = 2$ and $\lg n \leq 25$, we get a good performance with certain queries and a better stability when changing the shapes of the search queries compared to the kd-tree.

## 5.5 Vertical slices explained

In this section we are going to dive a deeper into the technical details of the results seen in some of the previous sections. The figures depicting the performance of the vertical slices showed some interesting tendencies. This is the last section before the summary.

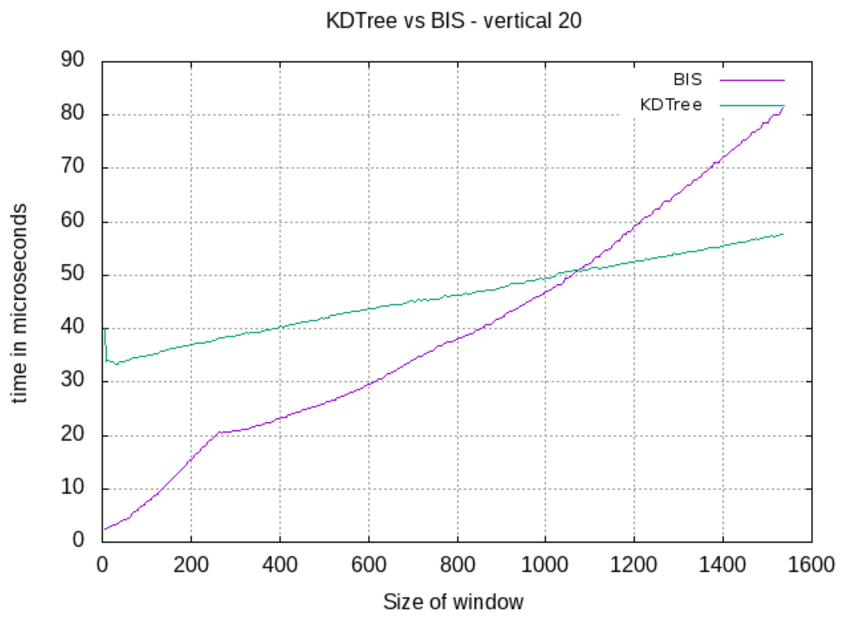
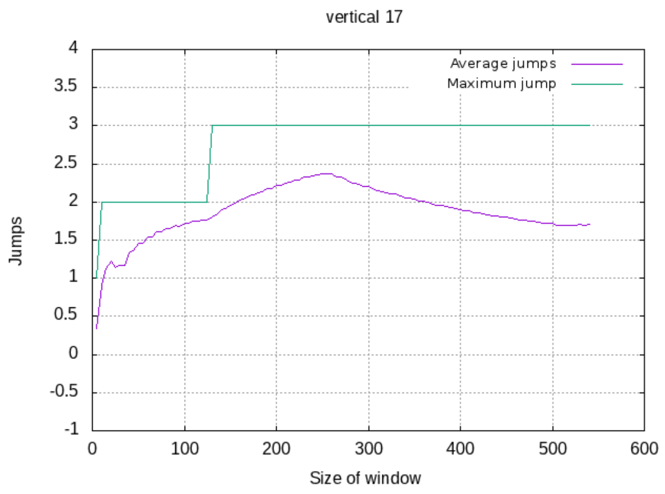
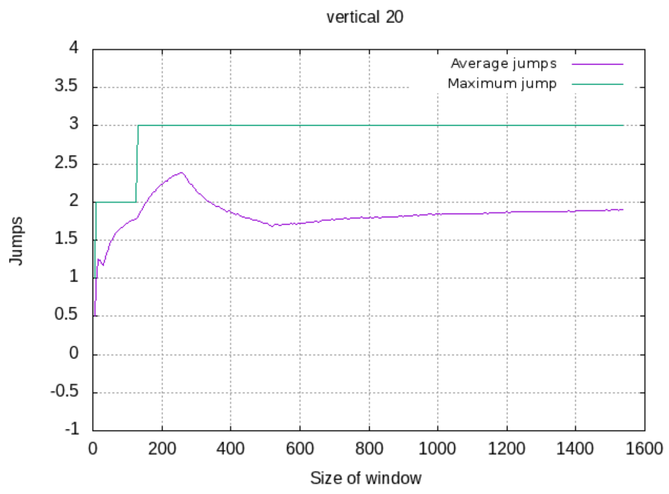


Figure 5.13: Vertical slice. data set size of $n = 2^{20}$.

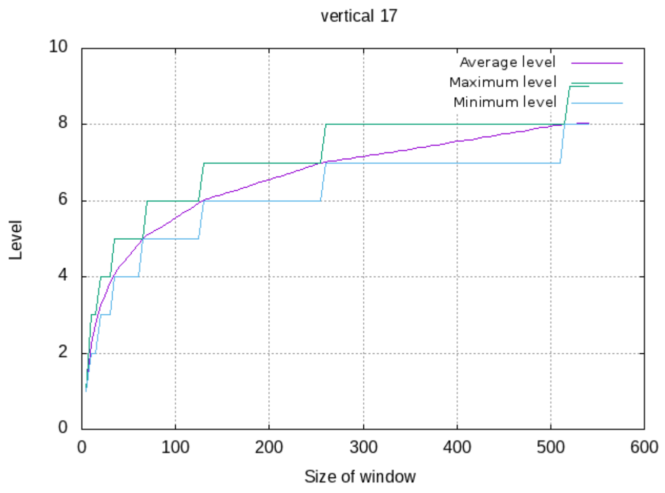


(a) Data set with $size = 2^{17}$.

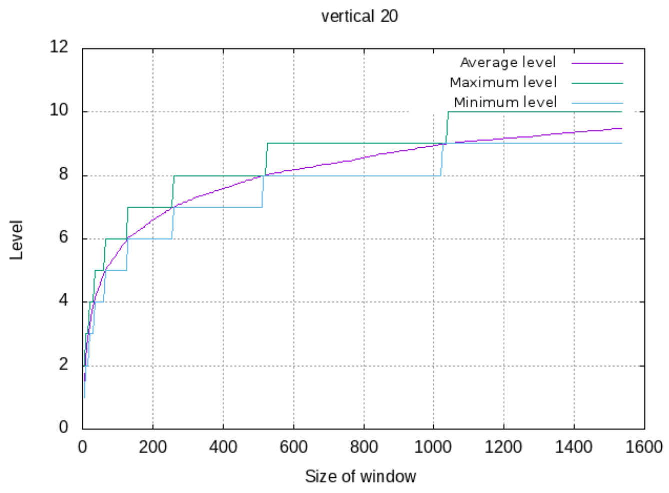


(b) Data set with $size = 2^{20}$.

Figure 5.14: Size of jumps. 'Average jumps' is the average of all the jumps performed normalized by the size of the slice.



(a) Data set with $size = 2^{17}$.



(b) Data set with $size = 2^{20}$.

Figure 5.15: Level of highest fully contained node.

On the graphs showing the performance of the vertical slices to the BIS data structure, there is a very noticeable change in slope at around $k = 256$. Since $B = 2$, we have a big jump at level $2^3 = 8$ which allows the ball inheritance structure to jump from level 8 to a leaf in one jump. When jumping from level 8 instead of 5, 6 or 7 the ball reaches the leaf in one jump instead of two or three. This means that the average amount of jumps per result will decrease and thus the running time will not increase as fast as before.

Figure 5.14a and figure 5.14b show the average amount of jumps per result, i.e. the sum of all jumps divided by $k$. We see how the graph has a local maximum at around $k = 256$ and then the average amount of jumps per result decreases until $k = 512$ where it starts increasing at steady level again. Figure 5.15a and figure 5.15b describes the highest level of a fully contained node. We see between $k = 256$ and $k = 512$ that the maximum is level 8 and the minimum is level 7 and that the average level increases meaning more and more fully contained node starts using level 8. Since we have $B = 2$, there is a 2-jump every 2 levels, a 4-jump every 4 levels, an 8-jump every 8 levels and a 16-jump every 16 levels. This means that at level 7 the ball inheritance structure needs 3 jumps to reach a leaf. This is why such a noticeable local maximum exists on figure 5.14a and figure 5.14b. The jumps per results eases off because from level 8 there is 1 jump, from level 9 there are 2 jumps and from level 10 there are 2 jumps. The graph on figure 5.14a also shows that when $2^i \leq k \leq 2^{i+1}$ the maximum level of the highest fully contained node is $i$ and the minimum level of the highest fully contained node is $i - 1$. Recall that the way the vertical slices work means that if a node is fully contained then all of the balls in the node's list will be followed.

We have seen a great increase in performance when the size of the slice is big enough to use the big jump at level 8. This will also happen if the big jump at level 16 is used, but that would require a big search query. We have omitted the technical details of the horizontal slices since that would be too technical. But it is interesting to note that the BIS data structure treats a vertical slice in a different way than the horizontal slice.

## 5.6 Summary

In this chapter we have presented the results of different experiments on the BIS data structure. We have compared it to the kd-tree. We have also confirmed that the practical use of the BIS data structure fits well with the theory.

The BIS data structure performs very well when a search query is shaped like a slice, most notably as a vertical slice. A square search query does not perform as well as a slice does on the BIS data structure. However the square search query is not a disaster on the BIS data structure. We noticed the difference in performance between the square and slice-formed search query to the BIS data structure was not as big as the difference in performance between the two shapes on the kd-tree. The worst kind of search query to the kd-tree was a slice and the best was a square - the exact opposite of the BIS data structure.

The kd-tree is much more dependent on the shape of its search query than the BIS data structure. This is most notable by looking at the difference between figure 5.9a and figure 5.9b.

# Chapter 6

# Conclusion

*"Hell... It's about time."*
— Tychus Findlay, *Starcraft II*

In this thesis we have presented the theory and performance of the BIS data structure. We have compared the performance of the BIS data structure to that of the kd-tree. The performance of the BIS data structure corresponds well with the theoretical running time of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$. It does not seem like the theoretical query time has any hidden big constants like we assumed the OBIS data structure had.

We have seen that, just like the kd-tree, the BIS data structure has a worst-case search query and a best-case search query. The difference in execution time between the worst-case and best-case search query to the BIS data structure is much less than that of the kd-tree. We have seen that the search query to the BIS data structure performs much more stable than a search query to the kd-tree when the shape of the search query changes.

Given a search query in the shape of slice, the BIS data structure will outperform the kd-tree up to good size. With $2^{17}$ points and a vertical search query, the BIS data structure will outperform the kd-tree when $k$ is less than 200. With $2^{25}$ points and a vertical search query, the BIS data structure will outperform the kd-tree when $k$ is less than 4660.

For smaller sizes of $k$, a vertical query to the BIS data structure will be several times faster than the kd-tree. With $2^{17}$ points and a vertical search query, the BIS data structure will be 3.5 times faster than the kd-tree when $k = 50$. With $2^{25}$ points the BIS data structure will be 42 times faster than the kd-tree when $k = 50$. In general the BIS data structure performs pretty well for when $k$ is small. The BIS data structure can definitely outperform the kd-tree in many cases.

There is a duality between the shapes of the search queries to the BIS data structure and the kd-tree. The BIS data structure performs best with slices and the kd-tree performs best with square searches. The BIS data structure performs worst with the square search and the kd-tree performs worst with a slice.

We have seen that the BIS data structure actually does compete with the kd-tree. The BIS data structure could be a realistic alternative to the kd-tree in practice. The BIS data structure uses several factors more space than the kd-tree. By picking an $\epsilon$ and a $c$ for $B = c \cdot \lg^{\epsilon} n$ we are able to leverage the performance of the BIS data structure with the amount of main memory it should be able to use.

# 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`.

[4] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(93)90040-4. URL `http://www.sciencedirect.com/science/article/pii/0022000093900404`.

[5] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. *CoRR*, abs/0902.2648, 2009. URL `http://arxiv.org/abs/0902.2648`.