Categorical Range Searching

Troels Thorsen, 20113820

Master's Thesis, Computer Science
June 2016
Advisor: Kasper Green Larsen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In this thesis, we implement multiple solutions to the *two-sided one-dimensional categorical range search* problem, where we discuss which solutions are the best. We will introduce two theoretical solutions; an *augmented binary search tree* from [SJ05], and the *priority search tree* from [BCKO08, ch.10]. The priority search tree was originally intended to solve the *three-sided two-dimensional range search* problem, but we make a reduction, such that the one-dimensional problem can be solved with this structure. All solutions will be compared with each other and a simpler *search and scan* method.

We will see how small input sizes result in faster query times for a simple *search and scan* method, and how the augmented binary search tree will have the best query time in general. Additionally, we will observe how the priority search tree will have slightly worse query time than the augmented binary search tree, but a space usage factor of $\log c$ smaller, which matters for larger input sizes.

Acknowledgements

Tobias Brixen and Anders Fog Bunzel, for being great office buddies. Søren Elmely Petterson, for the `minunit.h` and `dbg.h` files for C, which I somewhat converted for use in C++. Kasper Green Larsen, for being a really great advisor. My lovely girlfriend Anna Bruun Pedersen, for being such a great support as always.

Troels Thorsen,
Aarhus, June 13, 2016.

Contents

Chapter 1

Introduction

Imagine that you are browsing your favourite car seller's website, and you want to find a suitable car with a price less than 100.000 DKK. You enter the search query, and you are presented with the results. You notice that sellers offer cars from the Toyota brand, but you wonder if there are other brands in this prize category. The query you entered, is known as a *one-sided one-dimensional range query*, which formally is defined as follows. Let A be an array of ordered numbers, and let x be a query parameter. Given x, report all numbers $a \in A$ which satisfy $a < x$.

You enter a new query, now searching for suitable car brands with prices less than 100.000 DKK, and different brands pop up on the screen this time. As a user, you are naturally astonished by the amazing speed in which the request was made. This second query you entered, is known as a *one-sided one-dimensional categorical range query*, which is similar to the first. The only difference in the formal definition is that the array A will contain *coloured* numbers, but the query parameter will be the same. Instead, we wish to report all distinct coloured numbers where any number $a \in A$ satisfy $a < x$.

You wish to be more specific, as you realise the cheapest cars are all in bad shape, so you enter a new query, now searching for cars with a price between 80.000 DKK and 100.000 DKK, which must have at least 4 doors, as you do not want a small car. Again, cars in the range pop up on your screen at the speed of light! This third query you entered, is known as a *three-sided two-dimensional orthogonal range query*, which also is similar to the first. Instead, we now have a two-dimensional array of points, where given a query with parameters x_0, x_1 and y, we wish to report all points that lie within the x-coordinates, and above the y-coordinate.

This thesis will cover the *two-sided one-dimensional categorial range searching* problem, where we query for colours in a range like $[x_0 : x_1]$, and the *three-sided two-dimensional orthogonal range searching* problem, where we query for points in a range like $[x_0 : x_1] \times [\infty : y]$.

The two-sided one-dimensional range query without colours can be solved in $\mathcal{O}(\log n + k)$ time, by using a balanced binary tree on top of the array, and executing a simple range query from x_0 to x_1, where x_0 is found by searching the tree, reporting the k numbers in the array.

The coloured version of the problem can also be solved in $\mathcal{O}(\log n + c)$ time, where c is the number of colours reported. The solution for the latter will be presented later as the *augmented binary search tree*.

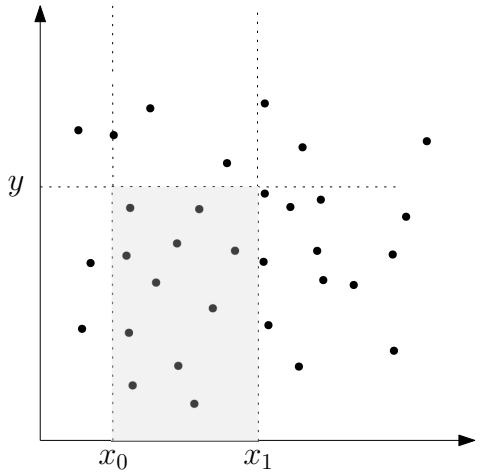

Figure 1.1: Example of a two-dimensional three sided query.

The three-sided two-dimensional range query can be solved in $\mathcal{O}(\log n + k)$ time, but we require a more sophisticated structure for that. The solution for the latter will also be presented later as the *priority search tree*.

Throughout this thesis, the efficiency of the data structures used will be measured according to their output and the time of query, along with the space usage, all denoted in worst case. Space will be expressed as a function of n (i.e. the input size), whereas the query time will be expressed as a function of both n and the output size (either expressed as k elements or c colours). A query of this form is known as *output-sensitive*, and will be denoted with bounds as $\mathcal{O}(f(x) + \text{output size})$ for some function f.

Mainly, we will implement the pointer machine categorical one-dimensional range query solution by [SJ05], and this solution will be compared with a simpler, but less efficient solution.

Additionally, a *priority search tree* from [BCKO08, ch.10] will be implemented, which will solve the three-sided two-dimensional orthogonal range query problem. As the priority search tree uses $\mathcal{O}(n)$ space, it will be interesting to see if it has just as good bounds for reporting colours in a one-dimensional context. We will compare the priority search tree with the solution for the one dimensional categorical solution. We will reduce a coloured one-dimensional set of points to a two-dimensional set of points, such that we can compare the two structures, and still keep the space bound for the priority search tree.

The *orthogonal range searching* problems have always been interesting, as many database systems use these kinds of queries. Orthogonal range queries can be found in many dimensions and deviations, and therefore we can also explain these problems more formally.

A general orthogonal range query can be described in the following way. Given a set of N points in a $d-$dimensional space, represent the points, such that given an axis-aligned query "rectangle", report the points therein.

By the query example with cars, it can be seen how the solutions to this kind of problems can be applied in database systems. We see that higher dimension categorical range searching can be done in a database system by executing queries with multiple parameters.

Ranged queries have a myriad of uses, and databases are just scratching the surface.

Another example can be to determine usage patterns, for instance in telemarketing. Suppose a company has database over phone calls, containing duration and time of the day, grouped by the customers calling. To determine the usage patterns, we can query which customers call during a specific time during the day, and for certain duration. This can be accomplished by solving an instance of a general range query problem.

As this is a well-studied field, it is expected that many of the theoretical solutions might be defined in different models of computation. The next section will introduce three common models, and afterwards some results in these models, mentioned in the future work section. Additionally, we will also have a quick overview of the inner workings of a computer, as we will use words as *cache miss* and *branch comparisons* in the later chapters.

The later chapters will study the two structures by giving a theoretical overview, implementation and lastly some experiments to analyze how well each implementation runs on a real world computer.

1.1 Models of Computation

When defining problems theoretically, we usually express things as query time and construction time in terms of the input size. In the simple theoretical model, we assume that we can do a single instruction in a constant amount of time, disregarding real-world advantages and disadvantages of the computer as we know it. In this section, we present commonly used models of computation, which have been used to reason closer to the hardware level of algorithms and datastructure theory. Many of the previously designed data structures have used the pointer machine model to describe range search problems. This section will give a quick overview of three well known models, that have been used to prove different bounds in range search problems. These models are a prerequisite for understanding some previous papers mentioned in Section 1.3.

1.1.1 Word Ram Model

This model allows indirect addressing of resources, which is similar to virtual memory in practice. We also assume that the memory available is divided into words of size w each. Given integer input points N from a universe $U = \{0, \ldots, U-1\}$, we assume that to address any input point, we will need a word of size $w = \theta(\log N + \log U)$ bits. Since the memory is divided into bits of w, we can manipulate a single word in $\mathcal{O}(1)$ time, and a word may represent arbitrary information about the input. E.g. we use the first m (where $m < w$) bits of a word to contain a value, and the last $w - m$ bits to contain a pointer to the next value. We may use any standard operations on the words in constant time, for instance subtraction, multiplication, division, addition and bit-wise operations. Using this model, it is possible to use small bit-wise tricks in constant time, as the one in the latter example. Additionally, it is assumed that a single word can be retrieved in $\mathcal{O}(1)$ time, as the word can be addressed.

1.1.2 I/O Model

The I/O model was first introduced in 1988 [AV88], where it was used to prove upper and lower bounds for sorting problems. In most real-life applications of the range search problems, we use large input sizes. Naturally, a database server will attempt to use as much memory

as possible, never quenching its thirst for more resources. Large data sets cause the disk to become a bottleneck. On hardware level, the disk is slow for random access, but it performs better doing sequential reads. A hard disk needs to spin up if it was idle for a while, then search for the data, and finally read the data into a buffer. In a single transfer, it is possible to send the data in the buffer to the program. The disk will repeat reading and searching for data, filling the buffer again.

To more accurately define performance using large data sets where the disk is the bottleneck, the I/O model was introduced. This model defines a machine to contain a main memory of size M words, and an infinite sized disk, partitioned into blocks of size B words. We assume that a data structure under this model resides on the disk, and that any parts used are moved from the disk to the memory in blocks of size B words. A single transfer of a block B is referred to as an *I/O*. Again, a word is defined as in Section 1.1.1.

Take notice that if our input set $N < M$, then we can solve a problem in-memory, spending $\mathcal{O}(M/B)$ I/Os, if the data was located on the disk in the first place. If the data was in memory on program start, it would only require $\mathcal{O}(1)$ I/Os.

1.1.3 Pointer Machine

The pointer machine model was first introduced in 1979 by Tarjan [Tar79]. A pointer machine consists of a memory and a finite amount of registers, just as on a real computer. The registers are either *pointer registers* or *data registers*. The memory consists of a finite amount of *records*. Each record consists of a finite amount of named *fields*, each containing either a *data field* or a *pointer field*. Each pointer register or pointer field will store a record, or contain *null*. Data registers or fields can store different types of data, like integers, strings, vectors, etc.

A pointer machine program is defined as a list of instructions, which will be executed sequentially. In the pointer machine, it is possible to access fields in storage, creating new elements or erasing them, and doing some simple operations, such as arithmetics. It is not possible to manipulate pointers, as it is in the word ram model, and therefore it is easier to prove lower bounds, as there are no nifty bit tricks available. Running times of programs in this model are defined in an obvious way – we charge a unit of time for each instruction performed.

Additionally in this model, we define the notion of *access steps* and *update steps*, where access steps define following pointers, and update steps define creating, modifying and erasing data or pointers. Therefore, random access is not possible, as we can only follow pointers.

E.g. in a binary tree of size n, we can access a leaf in $\mathcal{O}(\log n)$ access steps. If we were to insert a new leaf and balance our tree, we might have to spend $\mathcal{O}(\log n)$ access steps to find the leaf, and an additionally $\mathcal{O}(\log n)$ update steps to insert and bubble upwards in the tree to balance it.

In the introductory paper by Tarjan [Tar79], it is mentioned, that because of the lack of address arithmetic (bit-tricks and alike), it is impossible for a pointer machine to store hash tables, perform radix sort, and access dense matrices. When having downsides, the pointer machine model is an excellent tool to give good theoretical bounds in a higher level model, and to prove lower bounds.

1.2 The Hardware Model

We test practical implementations, and not theoretical models, so we must be aware of how the internal machinery of a computer works. This section will give us a quick overview of how the hardware of the computer has influence on performance.

A picture of the hardware model can be seen on Figure 1.2. We have been ignoring the virtual address space on purpose, as the operating system is responsible for translation between that and the physical memory and swap space on disk. The figure is very stripped compared to actual hardware, but it should help us understand the inner workings of a computer better.

1.2.1 Caching

In today's computer architecture, the CPU is the main computing force of the entire machine. If the CPU has to perform fast, it must have fast data and instruction access.

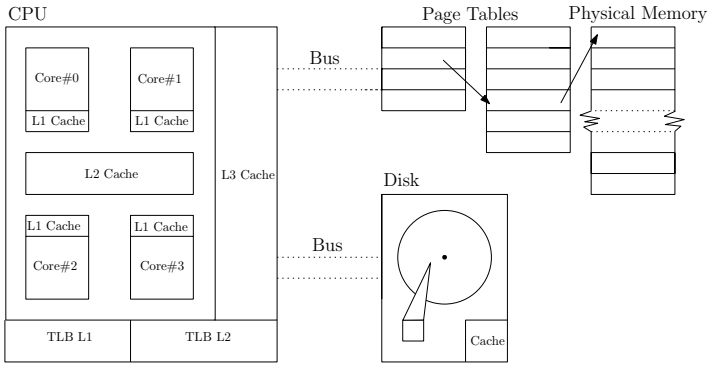

Figure 1.2: The hardware model.
Direct Memory Access (DMA) and other items are omitted to clarify the model.

To utilize retrieval and mutation of recurring data, we use a caching scheme in most hardware, where the most local level $L1$ is the smallest, but fastest, and the least local level (usually) $L3$ is the slowest, but biggest. Usually, the CPU will have a various amount of registers available for the fastest computation. In addition to registers, a CPU core usually has a L1 data cache and a L1 instruction cache of small size. Outwards, we have a L2 cache, usually shared between the CPU cores, if there are any available, and finally a L3 cache of greater size. On our testing machine (see Appendix, Section 5.2.1), we have a L1 data and instruction cache of 32 kilobytes each, a shared L2 cache of 256 kilobytes, and finally a L3 cache of 3 megabytes.

Every time the CPU attempts to get data or instructions from the memory, it will search in the cache. If an item is not found, a *cache miss* occurs, and the CPU will have to retrieve the item from the memory or disk. Whenever we fetch items from outside the cache, we bring in more than we need. Usually, we retrieve a block of data consisting of the item we need and proceeding items. The block of data we retrieve is called a *cache line*, and the size of this will vary according to the cache. On our testing machine, we have a cache line size of 64 bytes, which is room enough for $\frac{64}{4} = 16$ integers of 4 bytes each. Sequential data structures, such

as arrays, behave well in cache as the next elements are predictable, which usually means better performance for iteration.

Whenever we retrieve data from the memory (RAM), we use the *Translation Lookaside Buffer* (TLB) for faster memory page lookups. Usually a TLB is welded as a parallel circuit, letting us retrieve a memory page fast. Just as with cache, multiple levels are available, and access to these buffers can also *miss*. As we have a lot of memory (RAM) in the physical address space that can be addressed in the 64bit architecture on our machine, there are multiple TLBs that can be used.

If we process large data sets exceeding our physical memory, we use the swap space allocated on the hard drive or SSD. When loading data from a disk, we retrieve a large amount of sequential data to avoid spending additional I/Os retrieving data. Reading and writing data to the disk is the slowest of the latter methods of storage, and the speed can be compared with walking (querying the disk) to a destination instead of using a car (retrieving data from RAM).

1.2.2 Branching

As we will use the notion of branch mispredictions in some later experiments, we will introduce it shortly here.

A branch predictor is a digital circuit in modern CPU's, which allows the CPU to determine which branch is most plausible to continue execution on, e.g. in an `if-else` statement. Whenever we meet an conditional jump statement as an `if` instruction, the branch predictor will have to take a choice – should it pre-load the code inside the statement, or assume the path is not taken, pre-loading the code after the statement. Branch predictors are implemented in many ways, and different heuristics have been used for predicting the right move.

Whenever we predict the right branch, code execution will go smoothly, and will never have to slow down fetching a mispredicted branch of code. Branching will therefore have an influence on performance.

We can think of the branch predictor being a railroad junction that either goes to the harbor or the train repair shop. As the harbor is visited often, the railroad junction will always be fixed to that, assuming nothing else will happen. Whenever the train goes to the harbor, we "predict" correctly.

When a train wants to visit the repair shop instead, and the railroad junction leads the train to the harbor, the train must reverse, wait and let the junction switch tracks before taking the right way. This is the same as mispredicting a branch, and it will usually punish the CPU in the same way.

1.3 Previous Work

This section will introduce previous solutions from different papers addressing the *range searching* problem, and other deviations. The two-dimensional problems are relevant, as we later show a reduction from a coloured one-dimensional point set to a non-coloured two-dimensional one.

The *generalized one-dimensional categorical searching* problem was introduced by [JL93]. A solution with optimal $\mathcal{O}(n)$ space requirements and $\mathcal{O}(\log n + c)$ query time was devised

at that time. In this paper, *geometric intersection searching* problems are also introduced, along with the *generalized orthogonal range searching* problems. Here, two data structures are presented, solving the *four-sided two-dimensional orthogonal range searching* problem. One data structure used $\mathcal{O}(n \log n)$ space, and answered queries in $\mathcal{O}(\log^2 n + c)$ time. The other structure used $\mathcal{O}(n \log u)$ space, where u is the universe where $n \subseteq u$, and $\mathcal{O}(\log n + c)$ optimal query time.

The pointer machine solution by [SJ05] from 2005, solving the *one-dimensional categorical searching* problem, which will be implemented in this thesis, was proven to use $\mathcal{O}(n)$ space and $\mathcal{O}(\log n + c)$ query.

In [Mor03] from 2003, a reduction of the *generalized one-dimensional categorical searching* problem on an integer grid to a special case of a *tree-sided two-dimensional range searching* problem was given. The solution had a $\mathcal{O}(n)$ space bound, and optimal $\mathcal{O}(\log n + c)$ query time as well, which suddenly made solutions for the two-dimensional problems interesting, as they could also solve a one-dimensional problem.

In [McC85] from 1985, the *priority Search Tree* was devised, solving the *three-sided two-dimensional range searching* problem, with space bounds of $\mathcal{O}(n)$ and query times of $\mathcal{O}(\log n + k)$ time.

In [AGM02] from 2002, we saw a study of the one and two-dimensional problem in the word-RAM model. The one-dimensional problem was solved using $\mathcal{O}(n \log U)$ space, and $\mathcal{O}(\log \log U + c)$ query time. For the two-dimensional problem, it was assumed the points resided on a $U \times U$ integer grid. We saw a data structure using $\mathcal{O}(n \log^2 U)$ space, which had an optimal $\mathcal{O}(\log \log U + c)$ query time. Additionally, a data structure for three-sided queries was also devised, using $\mathcal{O}(n \log n)$ space, with a query time of $\mathcal{O}(\log \log U + c)$.

In [LvW13] both the I/O and word-RAM model are studied, and multiple data structures are presented. For the I/O model, we saw two different structures for answering three-sided two-dimensional queries. The first has an optimal query time of $\mathcal{O}(\log_B n + c/B)$ I/Os, alongside a space usage of $\mathcal{O}(n \log^* n)$, where $\log^*$ is the iterated logarithm. The second structure uses $\mathcal{O}(n)$ space, and has a query time of $\mathcal{O}(\log_B n + \log^{(h)} n + c/B)$ I/Os for some constant integer h, where $\log^{(h)} = \log(\log^{(h-1)} n)$. For the word-RAM model, optimal data structures are obtained for the three-sided problem.

Chapter 2

Two-sided One-dimensional Range Search with Colours

This chapter will contain a theoretical section about the two-sided one-dimensional range search problem, and a theoretical solution. Additionally, analysis and experiments will be performed on two implementations which solve the problem in practice.

2.1 Theory

In this section we will delve into the theoretical definition and solution of the problem, before we begin on the implementation. Therefore, we will introduce the definition of a generalized one-dimensional range search problem with colours, and its theoretical solution. The solution from the paper by [SJ05] is implemented on a pointer machine, and the theoretical bounds given are of this model. Instead of using a universe U as stated in the paper, we will use a set S, and build a static solution for just answering queries.

2.1.1 Definition

Definition 1. *Let $S \in \mathbb{R}$ be a set of real points and let $|S| = n$. Let the set of colours $C \subseteq S$ WLOG, as we cannot have more colours than points. Any point $p \in S$ can be described with an x-coordinate, and a colour c, and written respectively like $p.x$ and $p.c$.*

The problem can be described in the following way.

Definition 2. *Given a set S, and a query range $[x_0 : x_1]$, where $x_0, x_1 \in \mathbb{R}$, report the colours of the points $p \in S$ where $x_0 \leq p.x \leq x_1$.*

The problem can also be represented as an array of S elements, where each element has an x value and a colour c, as shown on Figure 2.1.

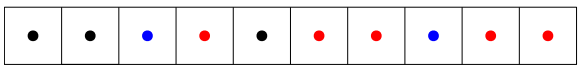

Figure 2.1: A one-dimensional array with colours.

2.1.2 Solution

We will use the solution from [SJ05], where the general idea was to build a balanced binary tree on the universe U. In contrary, the paper also refers to [Mor03], who built a balanced binary tree on the set S instead. This solution will build a balanced binary tree on top of S, and will only refer to the static solution with queries. The solution will use $\mathcal{O}(n \log c)$ space, and $\log n + c$ query time, which will be addressed and proven later in the implementation section.

Structure

First, we will build a balanced binary tree T on the set of S, where all elements are sorted in increasing order by their x-coordinate. For each node v, we will define a set S_v, which denotes the values stored in the sub tree T_v of v. Each node v will also have pointers to their *leftmost* and *rightmost* leaf nodes, respectively denoted by $left(v)$ and $right(v)$. Each node will also have a height in the tree, counting upwards. This means, that leaf nodes have height 0, and the root node will have height $\log_2 n$. The height of a node will be denoted by $height(v)$.

In addition to the binary tree structure, we will define two sets for each node v in the tree. Let v be an internal node, and let R_v denote the points of unique colours in S_v, where any point $p \in R_v$ will only be in the set if and only if it is the rightmost point of its colour in the set of S_v. Denoted more formally,

$$p \in R_v \quad \iff \quad \forall p' \in S_v \mid p' \neq p \wedge p'.c = p.c \mid p'.x > p.x$$

The set R_v will have a symmetric case, which will contain the leftmost points, denoted L_v. Additionally, these sets will be sorted according to the order of $p.x$, as we will need this property for queries. Luckily, ordering will come naturally, as we will iterate an already ordered set. An example figure of how the left and right sets are constructed, can be seen on Figure 2.2.

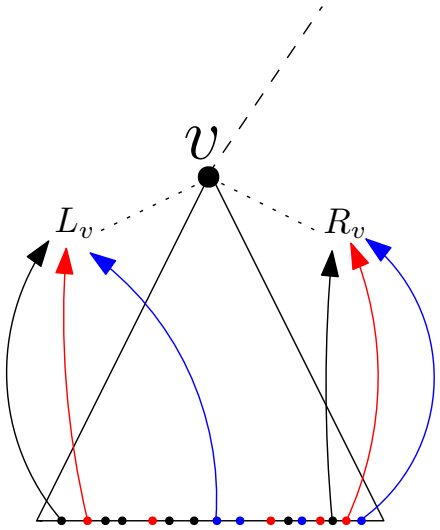

Figure 2.2: Construction of the R_v and L_v sets.

Query

By using the structure from last section, we can search the tree in $\mathcal{O}(\log n)$ time. The colour query will be of the form $[x_0 : x_1]$, and this query will first search for the successor of x_0, and then the predecessor of x_1, both located in S.

After finding the successor and predecessor leaf nodes, respectively u and v, we find the nearest common ancestor w of u and v in $\mathcal{O}(\log n)$ time[1]. Let l be the left child of w, and r the right child of w. The two children define sub trees that are guaranteed to contain all the points in S covered by the interval $[x_0 : x_1]$.

We now have two types of queries which must be answered. A right query, denoted by $(x_0, right(l))$ reporting the colours of the points in the sub tree of l, using the set R_l, which are all greater than x_0. A left query, denoted by $(left(r), x_1)$, reporting the colours of the points in the sub tree of r, using the set L_r, which are all smaller than x_1. Now we have a way of reporting all the specific colours in a sub tree, either leftmost or rightmost of some parameter x.

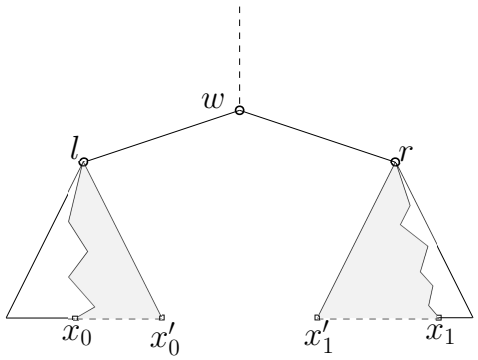

Figure 2.3: Query in the one-dimensional structure.

To summarize, we search for the successor u and v predecessor leaf nodes corresponding to x_0 and x_1. Then, we find the nearest common ancestor w of u and v. Finally, we answer two queries $(x_0, right(l))$ and $(left(r), x_1)$, on the left child l of w and the right child r of w. As the sets R_v and L_v for a node v can contain the same colours, there is a chance we report colours twice. An explanatory drawing can be seen on Figure 2.3, where x'_0 denotes the result of $right(l)$, and x'_1 denotes the result of $left(r)$.

2.2 Implementation

This section will introduce how two different solutions to the problem described in this chapter were implemented in practice. The projects have been implemented in *C++*, and the *PAPI* library has been used to measure accurate time used while executing. Further technical reading can be found in the appendix Section 5.1, which covers project files, how to compile and run, etc. In the following sections, we will introduce how the implementations were done.

[1]As this is the pointer machine model, it cannot be done using bit tricks and other address manipulation as we can only follow pointers, but if we were using the RAM model, it could be done in $\mathcal{O}(1)$ time.

2.2.1 Notes

To achieve fair experiments, the inorder layout for traversal of arrays has been chosen for the following implementations. The inorder layout for binary searching is the most inefficient one in practice [BFJ02], comparing to the Breadth First Search, Depth First Search and van Emde Boas array layouts. Although giving mediocre results for cache, it is expected that the search will have small if no importance in a coloured query, as the one-dimensional trees only search $\mathcal{O}(1)$ times, before scanning coloured sets or sorted arrays. The inorder layout is a tree layout, where the initial root node is accessed in the middle of the array. Knowing the size $|A|$ of the array, the height h of the root node is found to be $h = \log_2(|A|)$. Traversing to a left or right child depends on the height of the current node. The left and right children of a node v at index i are at positions $i - 2^{h(v)-2}$ and $i + 2^{h(v)-2}$ respectively, where $h(v)$ is the height of a node v. Being mediocre for look-ups, the inorder layout excels in scan times, as it retains the array in a sorted state.

2.2.2 Correctness

To ensure correctness, a small header file named `minunit.hpp` containing logic for unit testing has been used. Allowing the use of unit tests has been a great advantage, as most bugs have been found early in the tests, rather than in large experiments, making them harder to replicate. Unit tests have also been used to ensure partial correctness of individual auxiliary functions. The unit tests can be found in the `test` folder of the project, and their compiled versions will be in the `bin` folder of the project.

2.2.3 Sorted Array

To solve a one-dimensional query, where reporting distinct colours is required, we will use an array containing points with colours, ordered by their coordinates. As we assume the input set is unordered, it will be sorted using `std::sort` before being saved to the internal representation.

Instead of reporting each colour we ever meet, we maintain a set of distinct colours found while scanning the array. To keep references of whether or not we found a colour, a `std::vector` is maintained, with all entries set to -1 initially. This allows us to check for existence in constant time. Unfortunately we have to reset this set after each query, using $\mathcal{O}(c)$ time, which is paid by the query. In comparison, a `std::set` could have been used, with $\mathcal{O}(\log c)$ insertion and look up times instead.

Analysis

The structure of the solution is a simple array, containing the input size of points, making the space bound $\mathcal{O}(n)$. When querying, a list is used to keep track of the unique elements in the array. The list takes up $\mathcal{O}(c)$ space, where c is the amount of unique colours in the input set. In total, the space bound is $c + n = 2n = \mathcal{O}(n)$, as $c \leq n$.

To construct the sorted array, we sort it with a worst case performance of $\mathcal{O}(n \log n)$. Therefore, the worst case construction time of the array is $\mathcal{O}(n \log n)$.

When querying the sorted array with $[x_0 : x_1]$, we find the element corresponding to x_0 using a binary search. The query time of the search is at most $\mathcal{O}(\log n)$. After finding x_0, we

scan forward in the array, while inserting new colours into the result array. To see if a colour exists in the set or not, $\mathcal{O}(1)$ time is used, and insertions of new points are done in $\mathcal{O}(1)$ worst case time. As we in the worst case have to scan the entire array while maintaining the unique colour array, we see that the worst case query time will be $n + c = \mathcal{O}(n)$, as we are guaranteed to do c insertions and c resets in the coloured array.

2.2.4 Augmented Binary Search Tree

In this subsection the take on implementing the theoretical solution mentioned in Section 2.1.2 will be explained thoroughly. the augmented binary search tree will be denoted as T in the following explanation.

The data structure used to implement the solution, is a simple array denoted `_A`, containing array nodes. The nodes are traversed in the inorder layout, which allows us to just keep a sorted array as representation, as previously mentioned in Section 3.3.1. Each node v contains a point p, which contains a colour $p.c$ and a coordinate $p.x$, a height $v.h$, a size of its sub tree $|v|$, a reference to the parent index $v.parent$ of v, and two sets L_v and R_v containing points like p, each denoting the sets mentioned in the theoretical solution. The height is used for inorder search, and the parent is used to determine successors and predecessors. The sets are implemented using the STD library class `std::vector`, as this allows for fast range iteration, low space requirements, and preserving order of the elements inserted.

Initialization

To initialize T, its method `makeStructure` must be provided with the coloured points it should represent. First, the input set will be sorted using `std::sort`, which gives us the inorder tree structure. The method will then iteratively initialize all nodes with a height, a size of its subtree, a parent index, and corresponding point. In the beginning of the implementation, a recursive method was used instead. Using recursion resulted in slow construction, as the call stack grew unnecessarily big with large n. To implement an iterative way of visiting the nodes, a `std::vector` has been used as a stack to remove recursive calls.

For each node, the sets L_v and R_v will be initialized in almost the same way. To initialize L_v, we call the function `initialize_leftset`, in which the following parameters are provided. A range $[x_0 : x_1]$ of the sub tree corresponding to the set L_v will refer to, along with the index of the node v corresponding to L_v. This range will be calculated using the two methods `leftmost` and `rightmost`, which each calculates the spanning range of a sub tree, using the size of the sub tree. To keep references of whether or not we found a colour, a `std::vector` is maintained, with all entries set to -1 initially. This allows us to check for existence in constant time. Iterating from left to right, we find the first of each colour represented in the range $[x_0 : x_1]$, and insert them into L_v, which is a represented as a `std::vector` for each node v. If we have seen all the known colours, we stop. After finding the unique colours, the vector of maintained colours is reset by iterating over the colours in L_v, resetting each index in the vector to be -1 again. For the R_v set, we iterate from right to left, instead of iterating left to right.

Query

Given a range $[x_0 : x_1]$, the function `colorQuery` finds the unique colours present in that range, and reports them. First, we find the successor of x_0 and the predecessor of x_1, allowing us to find the indices which are closest to our range. If either x_0 or x_1 already exists in our array, we use those as indices instead of the predecessor or successor.

Then, the nearest common ancestor of x_0 and x_1 is found in $\mathcal{O}(\log n)$ time, and we use this to do a left and right colour query, gathering colours in a `std::vector`, using the same trick with the colour vector as in the initialization, as colours might be reported twice. A right and left colour query are almost done the same way, so only a thorough explanation for a left colour query will be given.

As each node has points in them, they can also contain a unique colour. Whenever we have a query that spans a common ancestor, we make sure to add the colour of the common ancestor to the result as well. To do a left colour query on a nearest common ancestor w, we first find the left child v of w, and secondly iterate from the end to the beginning of R_v, which is the set of rightmost colours from earlier. While the points are in the range of x_0 and x_1, we continuously insert the points into the given vector of colours, until we find a point not in the range $[x_0 : x_1]$. The right colour query is done in a similar fashion, although using the right child of w, and iterating the L_v vector from the beginning to the end.

Analysis

We use an array as the underlying structure. This allows us to place nodes in the array, which will represent the tree T mentioned in the theoretical Section 2.1.2. The array contains n nodes, which correspond to all input points. In addition to this array, we use an array of size equal to the total amount of colours c. It is assumed WLOG, that the amount of colours c are in $0 \dots c-1$, as this makes creating the colour array trivial. This array is used as a set when creating the left and right sets with colours. For each node in the array, we store a point, and the left and right set with colours.

We will now let s_i denote the size of the sub tree of a node, at a level i, and we will let the height of the tree be 0 at the root. The root node will contain at most $\min(s_0, c)$ colours, where $s_0 = n$, as it covers the entire tree.

A node can at most contain either c colours each if $c > s_i$, or contain s_i colours if $s_i \leq c$. It can now be seen that we can always pick $\min(s_i, c)$ as the total amount of colours for some level i. We now know the tree has $\log n$ levels, and each level has $\min(s_i, c)$ colours. We see that $s_i = 2^i$, as the size of a sub tree is defined by the level. It can also be seen, that at some level j, the size of the sub tree will be smaller or equal to c, i.e. $s_j \leq c$. This means, that after level j, the size of the tree will be dominating space. For a more illustrative measure, please refer to Figure 2.4.

As each node at a level i down to level j can contain at most $c \geq s_i$ colours each, the levels 0 to j will occupy the following space:

$$\sum_{i=0}^{j-1} c \cdot 2^i = 2^j c - c \tag{2.1}$$

Here 2^i denotes the amount of nodes we have in a single level i. We know that $j \leq \log n$, so the best case of c being the smallest, i.e. $j = \log n$ gives us $c \cdot n - c$ space usage, but as c is

the smallest at all levels, it implies that $c = 1$ and the space usage is therefore $\mathcal{O}(n)$.

If $j = 0$, we see how the equation gives us $c - c = 0$, and if $j = 1$, how it gives us $1c - c = cn$. We now see, that whenever the amount of unique colours c is smaller than the size of the sub tree at a given level, we will have linear space usage at that level and the previous ones, up to the root node.

If $j < \log n$, we know that at some level i where $s_i \leq c$, the space will be bounded by the sub tree instead. In these cases, we also know that from level j to $\log n$, we will use the following space:

$$\sum_{i=j}^{\log n-1} s_i \cdot \frac{n}{2^i} = \sum_{i=j}^{\log n-1} 2^i \cdot \frac{n}{2^i} = n \log n - jn \tag{2.2}$$

Here, $\frac{n}{2^i}$ denotes the size of a subtree at a given level i. As $j < \log n$, we see how the latter equation gives us a $\mathcal{O}(n \log n)$ space bound. We also see, that if $j = \log n$, then $c = 1$, and space usage resolves to 0, as c is the smallest everywhere, which means this equation is not applicable at any level.

Therefore, there must be a point where c is the factor that determines the space usage. More formally, when $s_i \leq c$ then j will define the level where the sub tree size becomes dominant. The level j can then be found when a size of a sub tree is $\frac{n}{2^j} \leq c$, giving us that $j \geq \log(\frac{n}{c})$. We can now bound the space of the structure by using equations 2.1 and 2.2:

$$\begin{aligned} 2^j c - c + n \log n - jn = \\ 2^{\log((n)/c)} c - c + n \log n - n \log \frac{n}{c} = \\ n - c + n \log n - (n \log n - n \log c) = \\ n - c + n \log c \end{aligned}$$

We now see that the above equation will be bound by $\mathcal{O}(n \log c)$.

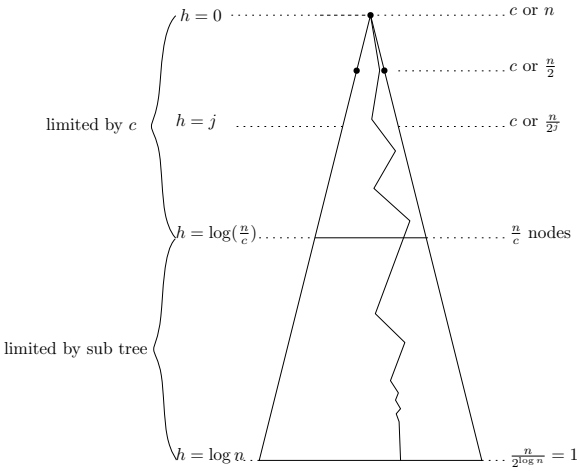

Figure 2.4: Calculating the amount of space used.
Left side of tree denotes height, right side denotes amount of colours.

As seen earlier in Section 2.2.4, the tree T is initialized using a stack instead of recursion, and each time we visit a node, we generate the two coloured sets of the node. First, the input

set is sorted, using `std::sort` with the worst case time complexity of $\mathcal{O}(n \log n)$. Afterwards, each node is visited once, and only once, where the left and right sets are constructed. For each set, we use the same array C with size of the distinct colours in the input set. This array is used as a set, such that we can make $\mathcal{O}(1)$ look-ups to see if we have visited a colour before. The array of colours C will be reset according to the visited colours after each set is initialized, so we might spend and additional $\mathcal{O}(c)$ time at the root level.

As we build each set according to the sub tree, we can at most iterate n elements at the root, $\frac{n}{2}$ at the children of the root, continuing until we reach the leaves, iterating a single time. The latter resolves into spending at most $(c + n) \log n = 2n \log n = \mathcal{O}(n \log n)$ time to scan and create the coloured sets. We can now see, that sorting takes $\mathcal{O}(n \log n)$ time, iterating the array visiting nodes takes $\mathcal{O}(n)$ time, and building the two coloured sets for all nodes take $\mathcal{O}(n \log n)$ time, resulting in a $\mathcal{O}(n \log n)$ worst case construction time.

The tree T can be queried using a coloured query, which will search for the predecessor and successor of the query points x_0 and x_1. We search inorder in the array, using a binary search with time complexity $\mathcal{O}(\log n)$. After the predecessor and successor have been found, we search for their nearest common ancestor, also using $\mathcal{O}(\log n)$ iterations. We now do a left or right query on the two respective children of the nearest common ancestor, where each set can contain at most c colours. Additionally, we must reset the colour set when we are done with the query, costing c time. In total $3 \log n + 3c = \mathcal{O}(\log n + c)$ time is spent querying T.

As a note for later, we can see that $3 \log n + 3c$ instructions are spent, compared to the $\log n + n$ of the sorted array with set. This implies that this augmented tree will have a larger constant overhead than the sorted array with set.

2.3 Experiments

In this section, experiments for the latter structures will be presented and discussed. The experiments have all been done on the same machine - specifications can be found in the appendix, at Section 5.2 and more technical info on the CPU can be found at Section 5.2.1. Each section regarding an experiment will include a description of the experiment, an expectation, and finally a result subsection. The data from the experiments has been plotted with *gnuplot*.

When referring to the simple range query implementation, the word "naive" will be used. Additionally, when referring to the implementation of [SJ05], where left and right sets have been used, we will use the wording "augmented BST". In the following experiments, the amount of colours is denoted as c, and the input size is denoted with n.

To ensure a good average, tests have been repeated 10 times, and random point sets with random colours have been used. For good randomness of points, the STD library `<random>` has been used, along with their implementation of the Mersenne Twister, which is a pseudo-random number generator (PRNG). To generate new random numbers each time, the PRNG was seeded with the current time. Random colours have also been used, which implies we are using a well-distributed, but not entirely unique amount of colours. This can be seen in experiments with $c = n$, where the augmented BST is slightly better than the naive solution. The augmented BST stores unique colours, which is why it is faster doing large scans.

Random queries have also been generated using the same method as above, to test the average query time. For each test iteration, 100 queries have been used, to get a good distribution. Exponentially increasing numbers have been used for testing the implementation, as they gave a good layout for the tree, which seemed most fair when using random queries. Random queries are not fair, as most of them span a large range, and we therefore have faster searches, as we almost never reach leaf nodes with that configuration. It can later be seen how the query range has an influence on performance.

Query times are calculated as average time for a single query, if not stated otherwise. Construction times are also calculated as average time for a single instantiation, if not stated otherwise.

2.3.1 Construction Time

In this experiment, we wanted to verify the construction times for the implementations. In the analysis of the naive implementation and augmented BST we argued that they both had a construction time of $\mathcal{O}(n \log n)$.[2] We therefore expect our graphs to show construction times close to $n \log c$, although the augmented BST may have a slight increase in time spent for larger c, as construction of left and right sets depend on c. The following graphs on Figure 2.7 show construction times for the two implementations.

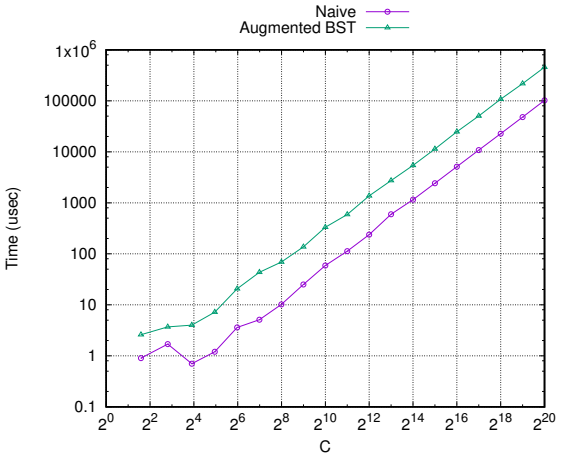

Figure 2.5: $c = 1$, varying n from 1 to 2^{20}

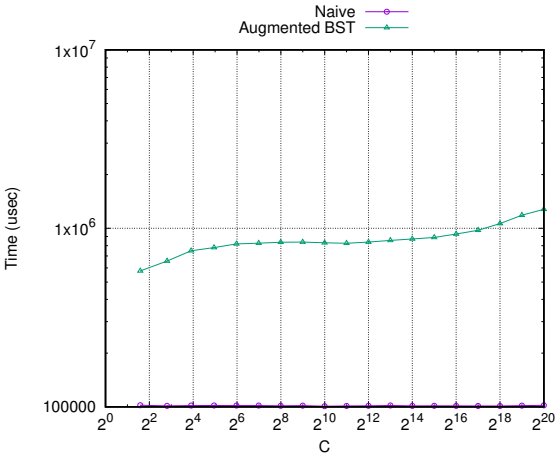

Figure 2.6: $n = 2^{20}$, varying c from 1 to 2^{20}

Figure 2.7: Construction time, using random queries.

Results

Our expectations were right and as can be seen on Figure 2.5, construction times are linear increasing according to c. On Figure 2.6, we see how there is a slight increase in construction time for the augmented BST with increasing c, whilst the naive implementation keeps the same construction time, regardless of amounts of colours. We also note how the augmented BST is about 10 times slower than the naive implementation. This is due to the fact, that the naive construction sorts the input set as the only operation, compared to the augmented BST, which also sorts the input set, and then afterwards iterates the sorted input once to create left and right sets. The augmented BST also represents its tree with nodes containing more than just a point, but also a height, a parent index, sub tree size, and the left and right sets. The augmented BST must therefore allocate at least three times as much memory as the naive implementation, which could be the explanation for the large constant slowdown in construction time.

[2] However, a tighter bound of $(n + c) \log n$ suits our observations more.

2.3.2 Fixing amount of colours

In this experiment, the amount of colours used was fixed to $c = 1$. The expectation was, that this decision would put the naive implementation in a bad standing compared to the augmented BST. Since the naive implementation in worst case would scan $\mathcal{O}(n)$ points regardless of the amount of colours, we would expect the augmented BST to be better, the greater the n. As we only spend 3 queries, costing in total $3 \cdot \log n$ time reporting 1 colour in the augmented BST, it is expected to outperform the naive implementation.

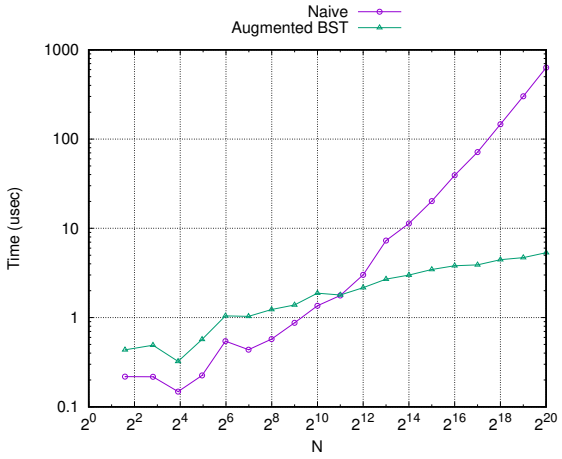

Figure 2.8: Query time, varying n from 2 to 2^{20}, and setting $c = 1$, using random queries.

Results

The results from this experiment can be found on Figure 2.8. It can be seen, that the naive implementation performs better at scanning sets of n smaller than 2^{10}. However, when the sets increase in size, it is clear that the augmented BST outperforms the naive solution. As the axis on time is on a logarithmic scale, it can be seen that the query time increases with a small constant factor for the augmented BST, and in a linear manner (if not worse) for the naive implementation.

Since it was expected that the naive solution would be faster with smaller sets, it is not a surprise that it is slightly better than the augmented BST with small n. This leads to the expectation, that the naive implementation should be used with smaller query ranges or input sets. In a later experiment, we will test and see if it matters switching strategies according to query range.

2.3.3 Fixing amount of points

In this experiment, the amount of points was fixed to a large size, and the amount of unique colours was increased from one until it was the same as the amount of points. The expectation was, that with small c, the augmented BST would outperform the naive implementation, as it would spend 3 queries, and then scan the exact amount of colours in the set, compared to the naive that would scan the entire input set. The greater c, the more equal the two solutions would become, as the R_v and L_v sets of the augmented BST would increase to be at most c big. The output set would in the end increase to be n, as $c = n$, where the expectation was that both implementations would do equally good.

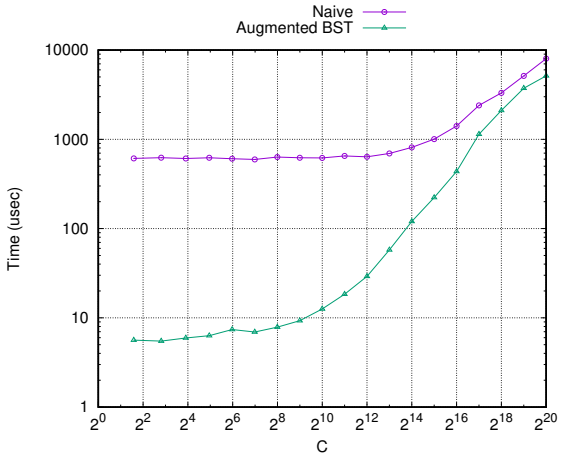

Figure 2.9: Query time, varying c from 2 to 2^{20}, and setting $n = 2^{20}$, using random queries.

Results

As seen on Figure 2.9, it is clear that small amounts of c give a great performance with the augmented BST. As the graph is logarithmic on the vertical axis, the gap of power is far greater than it can be perceived. When c increases, we also see how the naive implementation has almost the same speed with all c. When c reaches 2^{14} and larger, we see how the naive implementation has a large slowdown, as it visits more colours, implying that the coloured vector it maintains will have to be reset, spending more time with greater c. The output set is not sorted according to colours, which means that resetting the colour vector will not be in sequential access, but in random access, which implies this drastic slowdown for many colours. The two curves would be expected to be equal when $c = n$, but as we used random queries, there is a 2/3 chance (input points can either lie in the left or right children or the root), that the augmented BST queries a subtree instead of the entire set. The augmented BST is therefore a safe pick, when dealing with random colours, large input sets, and queries spanning a large range.

2.3.4 Fixing amount of points to be amount of colours

In this experiment, we fix $c = n$ for varying sizes of n. As seen in Section 2.3.3, we expected the two implementations to do equally good, if $c = n$. We expected this to be true for large n, but we were unsure whether or not this would hold for smaller values of n. The smaller n, the more we can fit in a single memory page, the cpu cache, and so forth. Therefore, it is expected that a simple scan would be better for this, but as n will grow, there is a probability that the augmented BST will perform better than the naive. As we are querying randomly on the structures, and not the entire point set, we will expect the augmented BST to perform better than the naive in the average case, as the augmented BST eliminates redundant scans.

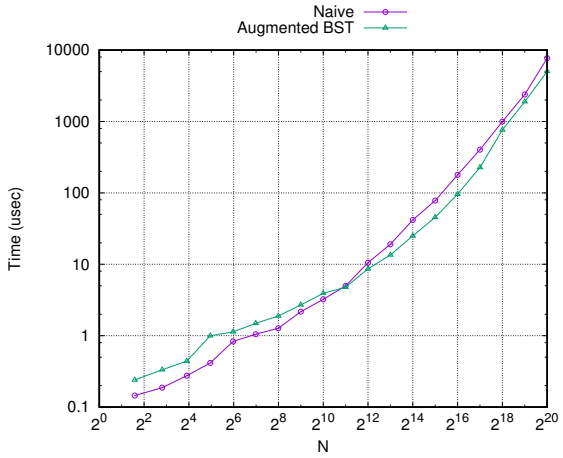

Figure 2.10: Query time, varying c from 2 to 2^{20}, and setting $n = c$, using random queries.

Results

The results of this experiment can be seen on Figure 2.10. This is one of the worst configurations for the augmented BST, as it excels in performance with small c, as seen on Figure 2.9. We expected the two implementations to be equally good, and as can be seen on the figure, we can confirm that expectation. There is a peculiar difference of the implementations, if we focus on the data points from $n = 2^0$ to $n \approx 2^{10}$, where the naive implementation is better than the augmented BST. As mentioned in the analysis Section 2.2.4, the augmented BST has a larger constant overhead than the naive implementation, and this might be the reason for this difference. When n grows beyond this point, the gap between the two curves increases slightly, and comes to a halt with large n.

2.3.5 Querying entire structure, varying amount of colours

In this experiment, we increase the amount of unique colours in the point set from one to n, but instead of using random queries, we will do queries that span the entire structure each time. Doing the spanning queries will put the naive structure in a worse light, as it will directly scale with n, without cutting any elements off. The spanning queries will also have an influence on the augmented BST, as the nearest common ancestor always will be the top node, which will contain all unique colours. Therefore, with big c, we expect the naive implementation to be just as good as the augmented BST. In addition, we expect the augmented BST being the best for smaller c, as the output size dictates how many elements will be scanned, and not the input size, as it does with the naive implementation.

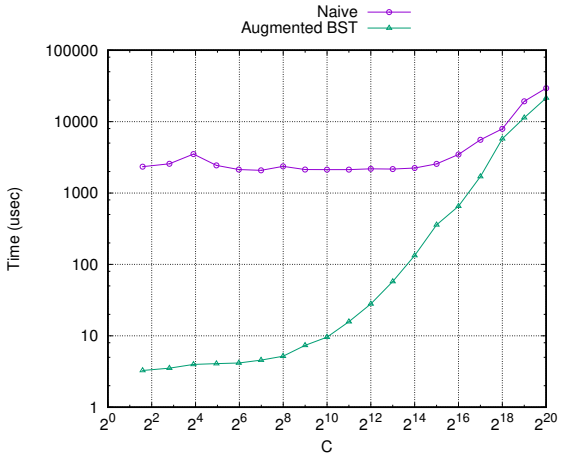

Figure 2.11: Query time, varying c from 2 to 2^{20}, and setting $n = 2^{20}$, querying from 0 to 2^{20}.

Results

The results of this experiment can be seen on Figure 2.11. Just as before, we see that the augmented BST outperforms the naive implementation on small c. When $c = n$, the augmented BST is just as fast as the naive implementation. We see how the query time of the augmented BST increases almost in a linear fashion from $c = 2^8$ to $c = 2^{17}$. For $c < 2^7$, we see how the constant overhead of the augmented BST comes in play, but gets barely noticeable as c grows. The slight gap comes from $c = 2^{20} \neq 2^{20}$ unique colours.

2.3.6 Fixing query range to be $\sqrt{n}$, varying amount of colours

In this experiment, we expected to make searching count more than usual. Compared to the experiment where we generated queries spanning the entirety of the structure, we instead fix our queries to be of such small size, that we will search deeper in the tree.

By defining a query range of size $\sqrt{n}$, we guarantee that the naive structure will scan at most $\sqrt{n}$ elements. Since we search two more times in the augmented BST, there is a possibility that the naive structure will perform even better at greater colour sets. The query size of $\sqrt{n}$ assures that we will reach a subtree containing $\sqrt{n}$ nodes, or two subtrees containing less. We know we will spend $\mathcal{O}(\log n)$ steps to reach a leaf node, but usually this is not the case with big queries. A subtree of size $\sqrt{n}$, requires at least $\log n - \log \sqrt{n} = \log n - \frac{\log n}{2} = \frac{\log n}{2}$ steps to reach, which is half of the steps used to reach a leaf node.

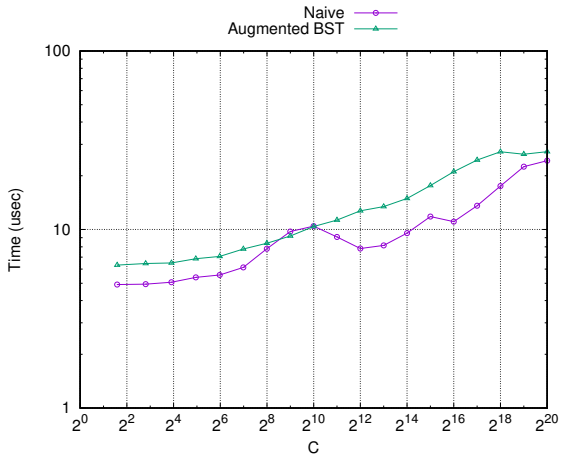

Figure 2.12: Query time, varying c from 2 to 2^{20}, and setting $n = 2^{20}$, querying with $\sqrt{n}$.

Results

The results of this experiment can be seen on Figure 2.12. As can be seen on the results, the augmented BST is slightly worse than the naive implementation. The naive implementation has improved greatly on short query ranges. It seems the two implementations are on par with each other, except at $2^8 \leq c \leq 2^{11}$, where it seems they are equal.

When we do a $\sqrt{n}$ query, we scan approximately $\sqrt{n} = \sqrt{2}^{20} = 1024$ coloured points. When we reach an amount of 2^8 unique colours, there is approximately $\frac{c}{\sqrt{n}} = \frac{2^8}{\sqrt{2^{20}}} = 0.25$ of each unique colour in a query range. At 2^{10} unique colours, there is approximately $\frac{2^{10}}{\sqrt{2^{20}}} = 1$ of each unique colour in a query range. Having to scan approximately all unique colours with the augmented BST is just as good as just scanning with the naive implementation, which can be the reason for the strange behaviour of the graph.

When $c < \sqrt{n}$, we are almost certain we will make redundant scanning with the naive implementation, as the query range will contain $\sqrt{n}$ points that will be scanned regardless, but less colours. In all cases, the augmented BST has a larger constant overhead, as it performs three searches, compared to one search in the naive implementation. With small

c, we tend to check if the same colour has been visited many times, and it is assumed cache behaves well in these cases.

When $c > \sqrt{n}$, we see as the amount of unique colours increases, and the query range remains fixed, there will be an even greater chance that we scan unique colours, which could be why the naive implementation turns out to be fast.

Generally, we know that the augmented BST will perform well with small c, even when spending three searches compared to one as the naive implementation does. We learn that even with a large input set $n = 2^{20}$, the naive implementation performs slightly better with shorter query ranges and large c.

In the next experiment in Section 2.3.7, we will see if the naive implementation performs better with shorter ranges.

2.3.7 Fixing query range to be $\log n$, varying amount of colours

Just as in the latter experiment, we have tried to reduce the size of the query range to provoke deeper tree traversal. This time, the query ranges are $\log n$ short, and will almost guarantee to go close to the leaves. To be exact, it will require at least $\log n - \log\log n$ steps to reach a leaf, which with our input sizes of 2^{20} numbers will be $\log 2^{20} - \log 20 \approx 16$ steps. It is expected that the naive implementation will have a smaller constant overhead than the augmented BST, as it only searches once. It is possible that the naive implementation will perform well compared to many of the other experiments.

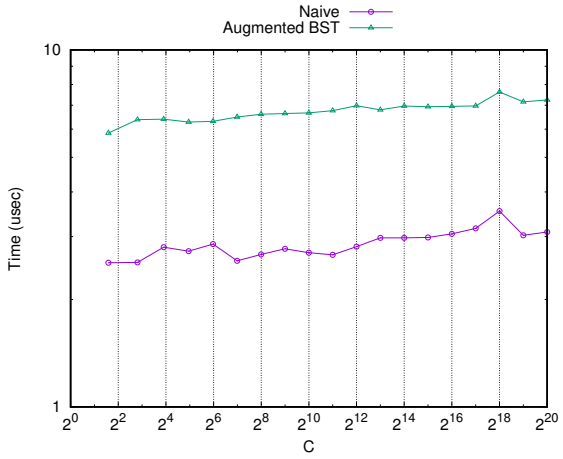

Figure 2.13: Query time, varying c from 2 to 2^{20}, and setting $n = 2^{20}$, querying with $\log N$.

Results

The results of this experiment can be seen on Figure 2.13. In this experiment, we see that the naive implementation gives us the best performance! We see that the range of $\log 2^{20} = 20$ numbers gives us fast traversal times for both structures, as the size of the query dictates how many elements will be scanned for the naive implementation, and the amount of colours in the query (at most 20) dictates how many elements will be scanned in the augmented BST.

The augmented BST will perform three searches, visiting at least $16 \cdot 3 = 48$ nodes in total and scanning at most 20 uniquely coloured elements, compared to the naive implementation, which will search once, visiting at least 16 nodes, and scanning at most 20 non-uniquely coloured elements. This gives a total of $48 + 20 = 68$ steps for the augmented BST, and $16 + 20 = 36$ steps for the naive implementation. We see that there is a $68 - 36 = 32$ additional steps taken by the augmented BST, which which is almost double the amount of the naive implementation. We see how the constant overhead of the augmented BST is larger than the naive implementation, and this is probably why the naive implementation is better in this case.

The reader must take note, that the query times on the graph are small compared to other graphs we have already analyzed, which represent more general behaviour.

2.3.8 Why Query Ranges Matter

The following graphs are a comparison of the query ranges already presented from earlier experiments. The graphs can be found on Figures 2.14 and 2.15. These two graphs give an overview of why query ranges matter, especially when creating reasonable experiments.

For queries with a short query range, we have seen how the naive implementation is a decent pick, even compared to the augmented BST. Depending on the amount of unique colours c in the set, we have also seen how the naive implementation is just as good as the augmented BST, when dealing with c being close to n. For small amounts of unique colours reported, and query ranges in all sizes, the augmented BST is the best pick, as the query time scales linearly with the amount of colours, rather than the input size. For short query ranges, the naive and augmented BST are close in performance, but with larger queries, the augmented BST excels in performance.

We see how query ranges matter, and therefore we have implemented the *augmented BST with limits*, where we will switch between the augmented BST and the naive implementation according to a given limit *lim*, which we will test according to query ranges and different query limits in the next experiments.

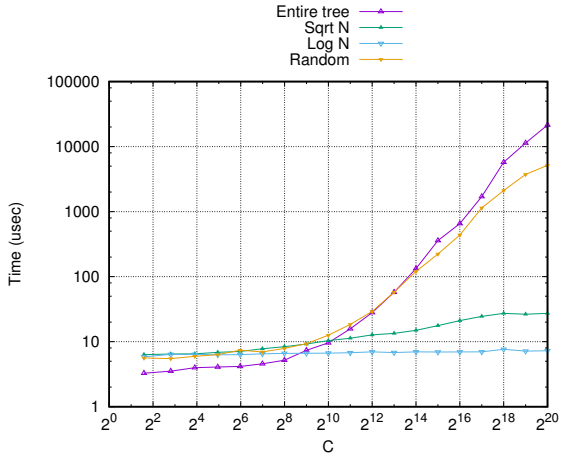

Figure 2.14: Augmented BST.

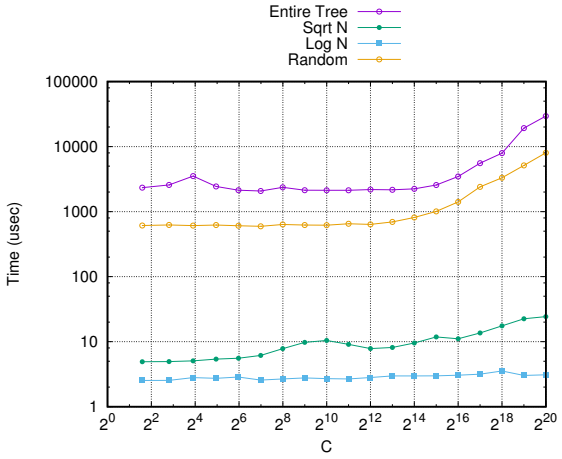

Figure 2.15: Naive implementation.

Figure 2.16: Querying with different ranges.

2.4 Augmented Binary Search Tree with Limits

After experiments with large amounts of data, it was noticed how the augmented binary search tree was slower on query times with short queries of n. This gave room for an idea, in which we could improve the running time even further by picking the best strategy for querying according to a query range limit.

The implementation of the tree was just like the original tree in Section 2.2.4, but with one minor difference. We let lim be a variable, defining the maximal size a query range may be, for it to be scanned with the naive implementation. When doing a query of the form $[x_0 : x_1]$, we check if $x_1 - x_0 < lim$, which describes whether or not we should scan the subset between x_0 and x_1, instead of utilizing the query technique used in the augmented BST.

We do not reason about any general theoretical bounds, as the performance increase is given from an optimization according to the implementation on the computer the experiments were run on.

We see that for a query $[x_0 : x_1]$ with an input set of size n, and with c unique colours, if $x_1 - x_0 < lim$ (i.e. the query range is small enough), we will spend $\mathcal{O}(\log n + lim)$ query time, else we will spend $\mathcal{O}(\log n + c)$ time. In theory it will be expected that the augmented tree will be the fastest regardless of lim.

On a real computer, the CPU has a cache, which makes retrieving continuous data and/or instructions faster. When the CPU attempts to load data from the cache, which is not present, a *cache miss* occurs, and the CPU is forced to get data from the RAM instead. When data from the RAM is loaded, a chunk of continuous data (also known as a *cache line*) is loaded into the CPU cache for easier continuous access. Scanning could therefore perform well for query ranges, as it is expected that the query set will fit into cache, and we will load the data faster this way.

2.4.1 Testing the limited implementation

As seen in the previous experiments with the query ranges set to shorter ranges, we attempted to let the queries go deep in the tree, to see if the constant of two additional queries actually mattered. It was confirmed that there was a difference in query time when selecting different query ranges. Therefore, the augmented BST with limits was implemented, and compared against the naive and the original augmented BST.

In the plots of the following sections, we will see how this modified version of the augmented tree gives us an edge in terms of query time for short query ranges. The implementation has been tested with different amounts of colours, query ranges, and limits on when to scan. As the following plots will show, the amount of colours does not matter for the short query ranges, but increasing the limit to and above 2^{10} will cause the augmented BST with limits to be slower in most cases.

Each of the three implementations have been tested with amount of colours, limit and query range, which have produced a lot of data. It applies to all following sections, that some of the plots have been omitted as they show nothing more than what was expected.

Picking Three Colours

The following plots on Figure 2.19 show the results of experimenting with three colours. Limits of 2^8 and 2^{12} have been picked from a test run of limits from the set $\{1, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}\}$, as these show two bad configurations.

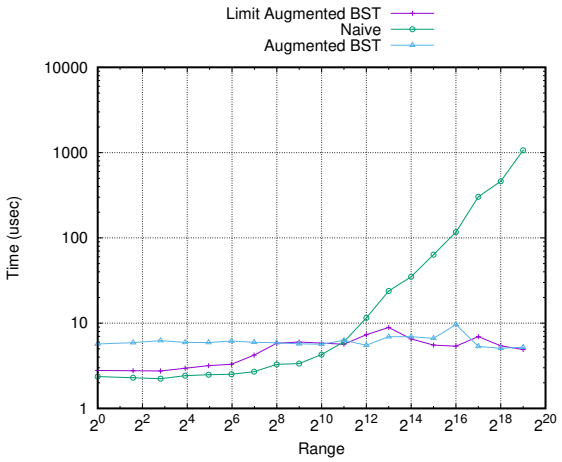

Figure 2.17: Limit 2^8

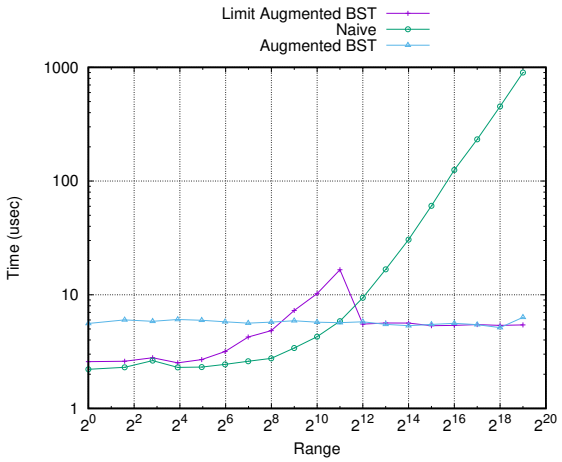

Figure 2.18: Limit 2^{12}

Figure 2.19: Query time, $n = 2^{20}$, 3 colours, varying ranges.

On Figure 2.17, we see how the augmented BST with limits has almost the same performance as the naive implementation. This was the best configuration that could be found from the test results. Earlier, as seen in the experiment Section 2.3.2, where we fixed $c = 1$ and varied n, we saw that the naive implementation was the better choice for $n \leq 2^{10}$. Luckily, we also see a similar tendency for the augmented BST with limits on this figure, as we picked our limit to be 2^8.

On Figure 2.18, we see how a large limit results in scanning large sets, which implies a slower query time. If we compare the graphs, we see how the limit has an impact on performance. Naturally, the amount of colours may have an impact on running times, and we will examine that with the two next sections.

Picking 2^{10} Colours

The following plots on Figure 2.22 show the results of experimenting with 2^{10} colours, which is $\approx \frac{n}{2}$. Limits of 2^{10} and 2^{12} have been picked from a test run of limits from the set $\{1, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}\}$, with the same reasoning as the experiment with three colours.

On Figure 2.20, we again see how a limit of 2^{10} was a good pick for performance, making the augmented BST with limits show good query time for both small and large query ranges.

On Figure 2.21, we see how a large limit, in this case 2^{12}, has bad impact on query time. It seems from this, that the amount of colours still has an impact on the running time – but it is expected when we perform output-sensitive queries.

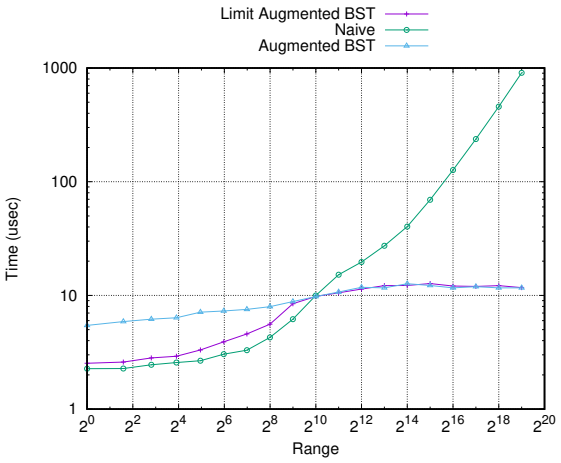

Figure 2.20: Limit 2^{10}

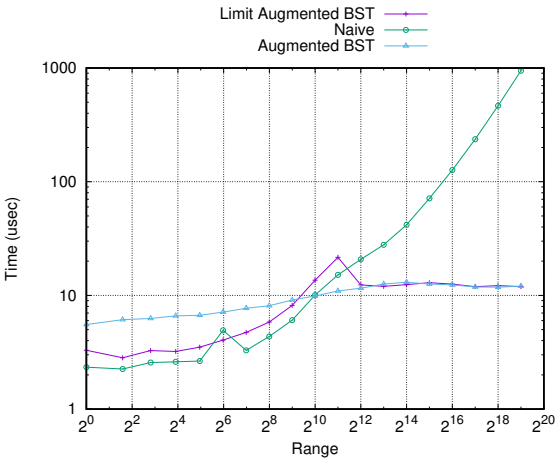

Figure 2.21: Limit 2^{12}

Figure 2.22: Query time, $n = 2^{20}$, 2^{10} colours, varying ranges.

Picking n Colours

The following plots on Figure 2.25 show the results of experimenting with $\approx n$ colours. Limits of 2^{10} and 2^{12} have been picked from a test run of limits from the set $\{1, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}\}$, with the same reasoning as the two previous experiments.

On Figure 2.23, we again see how a limit of 2^{10} was a good pick for query times. We also see how the naive and augmented BST are similar in performance with a large amount of colour.

On Figure 2.24, we see how a large limit, in this case 2^{12}, has slightly worsened the query time of the augmented BST with limits. We know from earlier, that the naive implementation scans n points, as $n = c$ does not imply that all colours are unique, compared to the augmented BST with guaranteed $\mathcal{O}(\log n + c)$ query time, scanning at most $2c$ colours. This implies, that if we used an input set of uniquely coloured points, the scanning times should amount to the same.

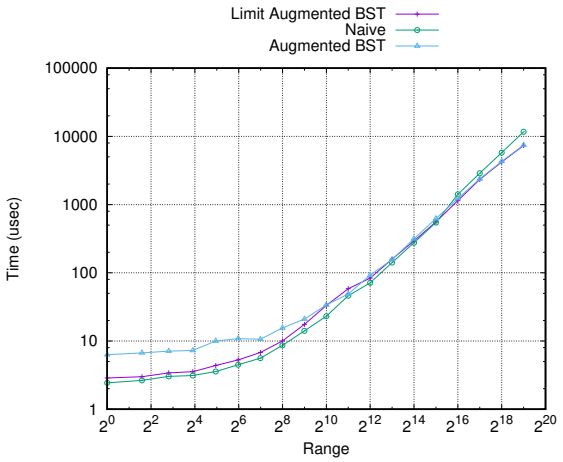

Figure 2.23: Limit 2^{10}

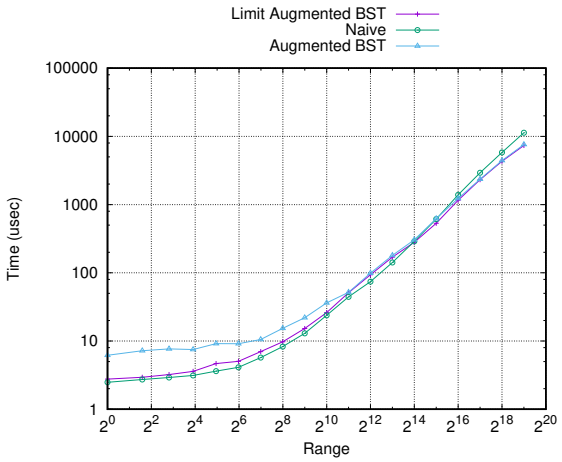

Figure 2.24: Limit 2^{12}

Figure 2.25: Query time, $n = 2^{20}$, 2^{20} colours, varying ranges.

2.5 Wrapping up

This section concludes the experiments for the naive implementation and the augmented BST with and without limits, and we will now summarize the results we have discovered.

We have in the previous experiments varied on input size n, amount of colours c and query range for all implementations. In addition, we have also experimented with *lim* of the augmented BST with limits. Mainly, query times were our focus, but construction times were verified in the first section.

The augmented BST proved to have great performance in most cases, but the naive implementation showed us good results with small input, regardless of the amount of unique colours. With short query ranges, we also noticed how the naive implementation was better. This inspired us to build an augmented BST with limits, which was then compared to the naive implementation and the augmented BST. It was then discovered, that picking a limit

of $\approx 2^{10}$ gave the best performance in comparison to both the naive implementation and the augmented BST.

So why exactly 2^{10}? Earlier when analysing in Section 2.2.4, we argued that the augmented BST had a larger overhead when searching before scanning. With our later experiments, testing the augmented BST with limits, we would spend $3 \cdot \log n$ time on searches. This would give us additional search time compared to the naive implementation, using only $\log n$ time for a search. At a query range of 2^{10}, searching once, and scanning afterwards would be just as good as searching three times and scanning unique colours. It is expected that this number is not magical, but rather dependant on the specifications of the computer the tests were run on. The specifications of the computer we ran experiments on can be found in the appendix, Section 5.2.

Let us first estimate how much memory our augmented BST uses. Implementation-wise, we are using an array filled with nodes. Each node contains a point p, a height, a sub tree size, a parent index, and two vectors defining the left and right sets. Points are implemented with C-structs of two integers, defining a colour and a point. Height, sub tree size, and parent index are integers as well, and the two vectors are referenced with integer pointers. In total, we use 5 integers and two pointers at least, which each uses up 4 bytes[3] and 8 bytes respectively, giving us up to a total 36 bytes per node. When testing, we used input sizes of $n = 2^{20}$, which would mean we had an array of at least $36 \cdot 2^{20}$ bytes size, roughly resulting in 38 megabytes size. The naive solution contains an array with two integers for each entry, which results in $8 \cdot 2^{20} \approx 2$ megabytes size.

A query range of 2^{10} points would result in a query size of $36 \cdot 2^{10}$ bytes size, which is roughly 36 kilobytes, for the augmented BST. For the naive implementation, we will use a query size of $8 \cdot 2^{10}$ bytes size, which is roughly 8 kilobytes. As seen in the appendix, our testing machine has a 32 kilobyte large *L1 data cache*. We know that having good locality of data is a benefit, and if we can keep our data in the $L1$ cache, it results in increased performance, as can be seen on our results. On our testing machine, we had ≈ 3 megabytes of L3 cache, definitely having enough room for the naive implementation, but not for the augmented BST.

We also see that short query ranges of 2^{10} or less might fit into the cache for the augmented BST, giving us fast performance. Additionally, we see the same for the naive implementation, with query ranges of $8 \cdot 2^{12} \approx 32$ kilobytes or less. Finally, we could have measured the L2 and L3 cache to see if this statement was correct. There is a disadvantage in searching and then scanning an individual vector as we do in the augmented BST, opposed to the naive implementation where we keep a single structure for both scanning and searching.

We see how the cache affects performance, but we also notice that points are stored in the same way in both the naive and augmented BST. Points in the left and right sets in the augmented BST are represented the same way as the array in the naive implementation, which means scanning would have the same performance. Therefore, we expect searching to be a costly operation as well.

The limit of a 2^{10} query range for scanning can therefore be viewed as the time where it is feasible for the augmented BST to perform three searches, compared to the naive implementation.

[3]Using GCC 5.3.0, `sizeof(int)` is equal to 4 bytes, and `sizeof(int*)` is equal to 8 bytes (this was compiled on a 64 bit architecture).

Chapter 3

Introducing the Priority Search Tree for lower space bounds

In this chapter, the *priority search tree* from the book [BCKO08, ch. 10] will be introduced, as this implementation has a smaller memory footprint, namely a space bound of $\mathcal{O}(n)$. We will sometimes refer to the priority search tree as PST. In addition, the priority search tree promises an optimal $\mathcal{O}(\log n + k)$ query time for three-sided two-dimensional queries, where k is elements and not colours. First, we will show how we can solve a three-sided two-dimensional query using the priority search tree, and afterwards use the tree to solve a two-sided one-dimensional query with colours, just as we did in the previous chapter.

We transform the input points of the one-dimensional coloured input set into a two-dimensional point set, such that we can answer the problem with the PST, which should give us the $\mathcal{O}(n)$ space bound, and the optimal $\mathcal{O}(\log n + c)$ query time. The priority search tree is interesting, as we can save some space and keep the query time, compared to the augmented BST with limits from the earlier chapter.

Additionally, we will introduce the definition of a generalized three-sided two-dimensional orthogonal range search problem, and its theoretical solution.

3.1 Theory

Just as in the previous chapter, we will introduce the problem and structure in this theory section.

3.1.1 Definition

Let us first define where our points come from.

Definition 3. *Let $S \in \mathbb{R}^2$ be a set of real points and let $|S| = n$. Any point $p \in S$ can be described with an x and a y-coordinate, and written respectively like $p.x$ and $p.y$.*

The query problem can be described in the following way.

Definition 4. *Given a set S, and a query range $[x_0 : x_1] \times [-\infty : y]$, where $x_0, x_1, y \in \mathbb{R}^2$, report the points $p \in S$ where $x_0 \leq p.x \leq x_1 \wedge p.y \leq y$.*

3.1.2 The Priority Search Tree

To solve the three-sided two dimensional orthogonal range search problem, we now introduce the *Priority Search Tree* from [BCKO08, ch.10], along with its theoretical analysis. The priority search tree was originally designed in the paper [McC85], and is noticeable for its good query time and linear space usage.

The priority search tree uses the property from definition 4, namely that queries are unbounded to one side, such that the tree only takes up $\mathcal{O}(n)$ space. The primary idea of the priority search tree is to use the unbounded side of the query to achieve $\mathcal{O}(\log n + k)$ query time.

3.1.3 Structure

The structure of the priority search tree can be thought of as a heap-structured binary search tree. To construct the tree, we do the following.

We let $P := \{p_0, p_1, \dots p_n\}$ define the point set used for input, where each point p has an x and y coordinate, addressed by $p.x$ and $p.y$. We pick the point with the smallest y-coordinate, $p_{min} \in P \mid \forall p \in P, p_{min}.y < p.y$, and create the root node with this point. Afterwards, we pick the remaining points $P \setminus \{p_{min}\}$, and split them according to the median x_{mid} of the remaining points. More formally, we split the remaining points into two sets as follows.

$$P_{left} := \{p \in P \setminus \{p_{min}\} \mid p.x \leq x_{mid}\},$$
$$P_{right} := \{p \in P \setminus \{p_{min}\} \mid p.x > x_{mid}\}$$

We will then process each set recursively, in the same way as we did for the root. Any node v in the tree will then either be empty, or contain the point with the smallest y-coordinate p_{min} of the sub tree. Additionally, v will also contain x_{mid} which will be used for queries later on. The left and right sub trees of v will be P_{left} and P_{right} respectively.

As mentioned earlier, using this structure also gives us $\mathcal{O}(n)$ space usage, as we never duplicate a point, or do anything additionally than creating a heap.

3.1.4 Query

To answer a query of the form $[x_0 : x_1] \times [-\infty : y]$, we utilize that the query is unbounded in the negative direction of the y-coordinates. The intuition is to search with x_0 and x_1 as in a regular binary search tree, and use the min-heap ordering of the y-coordinates to report points.

To answer the query we first search down in the tree, reporting points each time we visit a point that lies in the query range. At any node v, we go to the left child if the median x_{mid} of v is smaller than x_1, and right if x_{mid} is greater or equal to x_0. At some node v_{split}, we might encounter that we can go both ways, i.e. that both child nodes are contained in the query, as indicated by Figure 3.1, taken from [BCKO08]. We will search the sub trees contained by the two search paths by y-coordinate only, as we are ensured they will be contained by the x-coordinates of the query. As seen on the figure, the grey areas will be searched and points will be reported using a simple recursive function, as if we would query a heap. Given by the min-heap order, we will stop when the y-coordinates of the child nodes are larger than y.

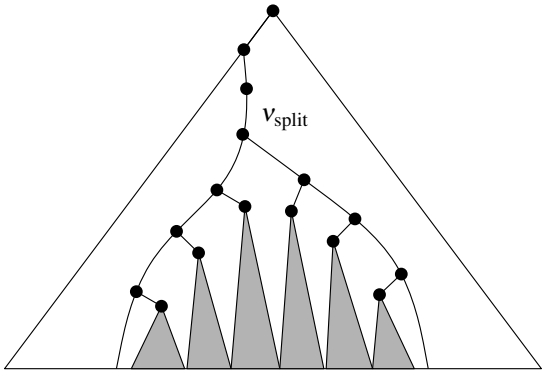

Figure 3.1: Query in a priority search tree.

We will in the worst case iterate $\mathcal{O}(\log n + k)$ points, resulting in a single splitting search path done in $\mathcal{O}(\log n)$, and reporting points in each sub tree of a visited node v with the recursive function in $\mathcal{O}(1 + k_v)$ time. As explained earlier, the min heap structure allows us to report everything we need and nothing more, as the ordering of the points allows us to spend no unnecessary time.

3.2 Reduction

As mentioned earlier, the priority search tree can solve two-dimensional three-sided range queries. Now, we will give a reduction from the one-dimensional two-sided coloured range queries to three-sided two-dimensional queries, such that we can solve the one-dimensional problem with the priority search tree.

The general idea of this reduction is to arrange the coloured points $p \in P \in \mathbb{R}$, such that provided a specifically crafted query, we will only report unique coloured points. As the priority search tree will answer queries of the form $[x_0 : x_1] \times [-\infty : y]$, we must first give our points more dimension. First, we will provide a more intuitive reduction to answer queries of the form $[x_0 : x_1] \times (y : \infty]$, and then make it work with our priority search tree.

We let a point $p \in P$ with an x-coordinate $p.x$, and colour $p.c$ be transformed to a coloured point $p' \in P' \in \mathbb{R}^2$, where $p'.x = x$, $p'.c = c$ and $p'.y = x_{succ}$. Here, x_{succ} will denote the x-coordinate of the *next* point with the same colour as p'. If there is no succeeding point of p', the x_{succ} is set to ∞.

In this way, we will form a "chain" for each set of uniquely coloured points, which will have an increasing order on the second coordinate. As illustrated on Figure 3.2, we see how the increasing coordinates for each unique coloured set of points will give us this arrangement. We can use this arrangement of points to our advantage, as we can decide the parameters for our query, especially the y-axis.

We will now let any query of the form $[x_0 : x_1]$ be translated into the query $[x_0 : x_1] \times (x_1 : \infty]$. As we have arranged the points $p \in P$ without modifying the x-coordinate, we are ensured to correctly report only the points within the $[x_0 : x_1]$ range. Additionally, we see how the query range $[x_0 : x_1] \times (x_1 : \infty]$ will contain only the points $p \in P$ with $p.y > x_1$. As each coloured point has been given a y-coordinate according to its succeeding points x-coordinate, we see how querying with $y = x_1$ will exclude all previously coloured points, as

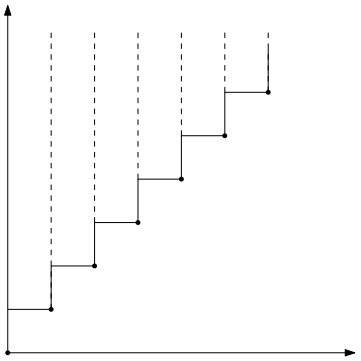

Figure 3.2: Arrangement of the linked coloured points.

their successors will be less than x_1, and therefore not a part of the query. We also see how a point with a successor greater than x_1 will be included in the query.

The only problem we have now, is including a colour more than once. We see how this can only be the case, if the query $[x_0 : x_1] \times (x_1 : \infty]$ will contain more than one successor of a single colour.

If we query with $x_1 = x_{max}$ where x_{max} is the largest value of the x-coordinates in the input set, we will only get points with $x_{succ} > x_{max}$, which will be the ends of all the coloured "chains", containing only a unique set of colours. If we set x_1 to be anything else, we are guaranteed that only points with successors greater than x_1 are reported, and since we can only access points in $[x_0 : x_1]$, we are guaranteed that only one colour of each will have a successor greater than x_1, meaning only a unique set of colours will be reported.

An example query, where the query $[3 : 5] \times (5 : \infty]$ is used, can be found on Figure 3.3. Two different colours have been used, where the first point set is $\{1, 2, 3\}$ and the second set $\{4, 5, 6\}$.

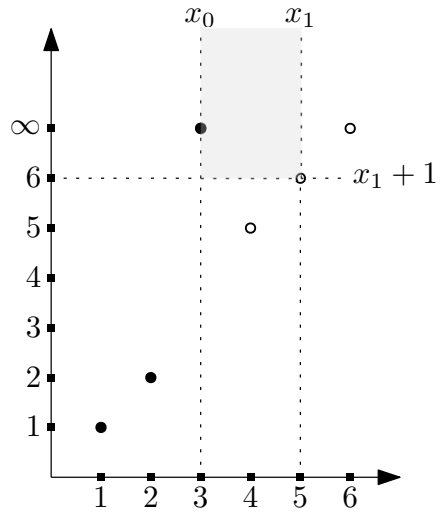

Figure 3.3: Query example of the linked coloured points, with two colours depicted as circles and dots.

Lastly, to support queries of the form $[x_0 : x_1] \times (-\infty : y]$, we simply transform the points as earlier, but invert the y-coordinate, such that for a point p, $p.y = -x_{succ}$. This arrangement of points will give a decreasing order instead. The query $[x_0 : x_1] \times (x_1 : \infty]$ from earlier, will simply be translated into $[x_0 : x_1] \times (-x_1 : \infty]$, which will now support the change in the points.

3.3 Implementation

This section will introduce how the priority search tree was implemented in practice. This structure has also been implemented in *C++*, and the *PAPI* library has been used to measure accurate time used while executing. Further technical reading can be found in the appendix Section 5.1, which covers project files and how to compile and run.

3.3.1 Notes

Pointer based traversal has been chosen for the *priority search tree* implementation. Pointer based implementations are in general less cache-aware, and it should result in slower running times compared to the array layouts used in the implementations solving the one dimensional problem specifically. The pointer based traversal is reference based, so each node v will have children $v.l$ and $v.r$, which can be followed by their pointer. A pointer-based solution has been used, as the priority search tree most likely will be unbalanced, which would imply more space usage if using an array. Additionally, the pointer-based solution is easier to implement, as it seems more intuitive according to the theoretical design.

3.3.2 The Priority Search Tree

In this subsection, the implementation of the priority search tree will be covered. The tree priority search tree is defined as a single root node. Each node v contains a two-dimensional point $v.p$ with a colour $v.p.c$, a $v.x_{mid}$ variable, and pointers to left and right child nodes, respectively $v.l$ and $v.r$.

Initialization

To initialize the tree, a list of two-dimensional points P must be given as an argument to the `makeStructure` function. This function will sort the list by x-coordinate with `std::sort`, and call the recursive function `initialize_tree` with the now ordered list. We have ordered the list to quickly split it in smaller pieces, as we will split it on x-coordinate.

Now, we will split the list according to the median x_{mid} of the current points, and find the point p_{min} with the smallest y-coordinate. We pick the smallest y-coordinate, as we wish to create a min-heap ordering on the second axis. We find the median by picking the element in the middle of the list in constant time. Additionally, we scan the input list to find the smallest y-coordinate. The point p_{min} will be assigned to the current node, along with x_{mid}. We split the list by creating two new containers, and passing them as arguments to create the left and right children of the current node. Instead of creating new containers for each recursive step, we could have passed the original list along as a pointer, but we had to remove p_{min} from the container, spending $\mathcal{O}(n)$ time anyways.

Query

The query algorithm is simple, as it resembles the theoretical solution closely. The query function `query` takes three arguments, namely the integers x_0, x_1 and y. First, the function `walk_to_split` is called, which reports points on the search path to x_0 and x_1, if they are in the query range. At some point, the function halts and returns the node v_{split}, where the search path branches and we can go to both the left and/or right.

Now, the search path will be traversed both ways using the functions `walk_left` and `walk_right`. We are guaranteed that the search path traversed by `walk_left` will have all sub trees to the right of the path included in the range $[x_0 : x_1]$. The `walk_right` function will be the opposite of `walk_left`.

Each function searches down in the tree, and calls the recursive function `report_in_subtree` at each node included in the range $[x_0 : x_1]$. This means, that `report_in_subtree` will only have to use the min-heap ordering of the y-coordinates, to determine if a point is to be included in the reported points.

The illustration on Figure 3.1 from the theoretical section can be used as a reference for the search path, where the greyed out sub trees are the ones visited by the `report_in_subtree` algorithm.

Analysis

We remove one point at each recursion step, which makes the recursion linear in construction time. Additionally, $\mathcal{O}(n)$ time is used in all levels of the tree, as we find the point with the smallest y-coordinate by scanning. As we always split in the middle by the median, we will create a close to completely balanced tree. Having a balanced tree gives us $\log n$ levels, resulting in a $\mathcal{O}(n \log n)$ total construction time.

The query will in the worst case follow two search paths which will not include any points in the query. A single search depends on the height of the tree, which in our case is $\log n$ tall. We will spend $\mathcal{O}(1+k)$ time whenever calling the `report_in_subtree`, as we are guaranteed to only report points which lie in the range of $[x_0 : x_1]$ and using the min-heap ordered property, to only recursively visit children if they are smaller than the given y. In total, we will spend a $\mathcal{O}(\log n)$ search, and report $\mathcal{O}(k)$ points, giving us a $\mathcal{O}(\log n + k)$ time on a query as promised earlier.

On behalf of the reduction, we spend $\mathcal{O}(n \log n)$ time to sort the input set, and afterwards $\mathcal{O}(n)$ to scan and convert all points.

For good measure, we expect our PST to have a small memory footprint per node, as each node contains two pointers, one integer, and a two-dimensional point, giving a total of $2 \cdot 8 + 4 + 3 \cdot 4 = 32$byte space usage.

3.4 Experiments

In the following section, we have tested the priority search tree against the augmented BST with limits (now ABSTL) from the earlier chapter. It must be noted, that the limit of the ABSTL is set to the optimal configuration from earlier, which was 2^{10} in average. All earlier experiments we have covered are also used as cases in this section, and therefore some descriptions of expectations are omitted to avoid redundancy.

3.4.1 Construction

In this experiment, we were slightly worried about the construction time, as we were using a recursive construction algorithm for the priority search tree. In addition, we knew that the construction time for the PST was $\mathcal{O}(n \log n)$, compared to $\mathcal{O}(n \log c)$ of the ABSTL. We see on Figure 3.4, how increasing c will give slightly slower construction times for the ABSTL, while the PST remains unchanged. As the ABSTL has an overhead in construction, we see how it is slower than the PST. As we use the reduction from Section 3.2, there will be an overhead in converting the coloured one-dimensional points to two-dimensional points, but compared to the ABSTL, the PST is still faster. On Figure 3.5, we see that the construction times are similar when $c = n$, but we still see how the augmented BST is slower.

We can conclude that the recursive construction algorithm of the PST does not slow down construction time, compared to the ABSTL.

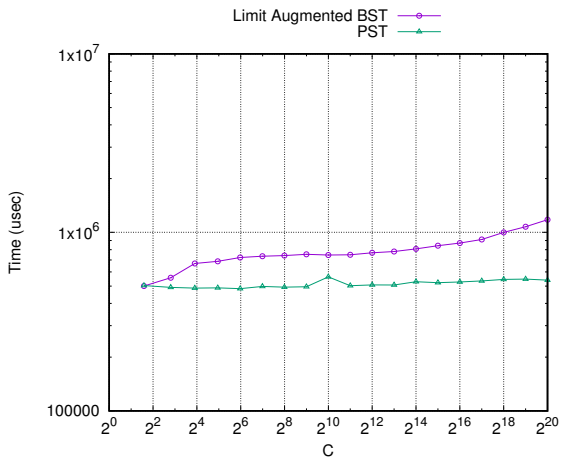

Figure 3.4: Setting $n = 2^{20}$ and varying c.

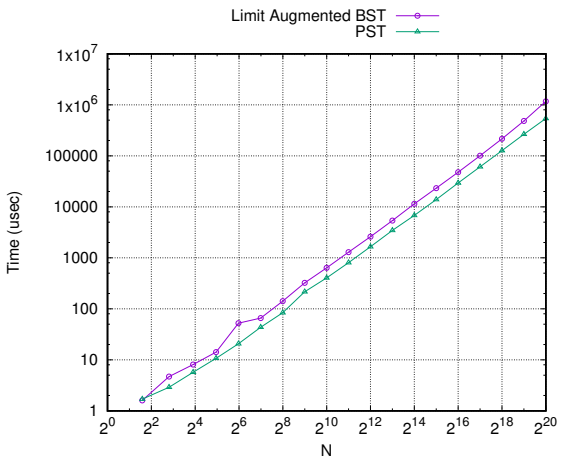

Figure 3.5: Setting $n = c$.

Figure 3.6: Construction times.

3.4.2 Querying entire structure, varying amount of colours

In this experiment, we increase the amount of unique colours in the point set from one to n, but instead of using random queries, we will do queries that span the entire structure each time. We expected the PST to perform worse in terms of cache, as we use a pointer structure, compared to the inordered array layout of the ABSTL.

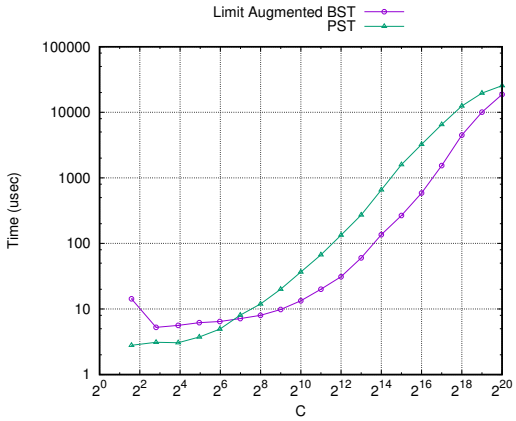

Figure 3.7: Querying entire tree, query time.

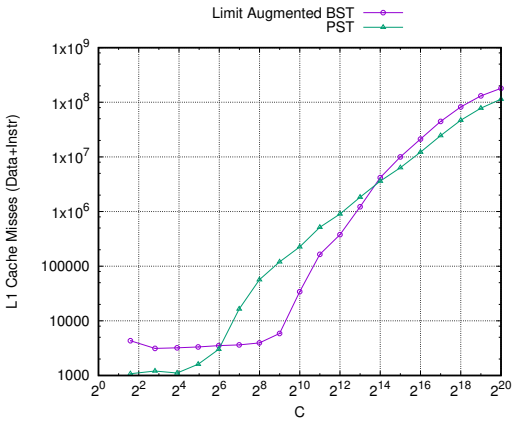

Figure 3.8: L1 cache misses.

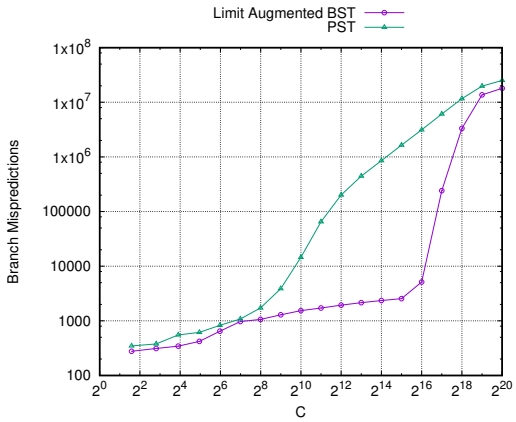

Figure 3.9: Branch mispredictions.

Figure 3.10: Querying entire tree.

We see on Figure 3.7, that the PST is slightly better at smaller amounts of unique colours, but is quickly overtaken by the ABSTL when we have more than $c = 2^7$ colours. Looking at Figure 3.8, we see how the slowdown is caused by a sudden increase in L1 Cache misses. As explained earlier, the PST is pointer-based, and therefore the children of a node in the PST could lie anywhere in memory, comparing with the array-based structure of the ABSTL, where the left and right sets have a sequential layout.

Let's take a look at the final graph, seen on Figure 3.9. This figure shows branch mispredictions for the query data of the ABSTL and PST. We see how mispredictions increase as

the amount of colours increases on both curves. The mispredictions increase close to linearly with c, as the more colours exist, the more nodes the PST has to visit.

There is a sudden increase in branch mispredictions for the ABSTL at colours $c > 2^{16}$ to $c = 2^{20}$, which can be explained by a larger percentage of unique colours. We see how 2^{16} colours will give us an approximately $\frac{2^{16} \cdot 100}{2^{20}} = 6.25\%$ chance that a point is a given colour. As we increase the amount of colours, we also increase the uniqueness, along with the chance of finding a new unique colour.

Having more unique colours will also have an impact on how the ABSTL handles reporting colours, as it allocates an array as big as the amount of unique colours, and uses it to remember if a colour has already been visited. This array will be accessed and written to, if the colour was not seen yet, and later reset for new queries. With the sudden increase of finding a new colour, the colour array will be written to more as the percentage increases. It can be seen how 100% unique colours flatten the curve once again.

The PST performed surprisingly well in our experiment, especially compared to the fast ABSTL.

3.4.3 Queries of size $\log n$

In this experiment, we have used query ranges of $\log n$ size, hoping to see similar search times for the PST and ABSTL. We see on Figure 3.11, how these small queries will result in the same search times for both the PST and the ABSTL, regardless of the amount of colours. It is interesting, how a single branching search in the PST is just as good as scanning with the ABSTL. It must be noticed that, as the query ranges of $\log 2^{20} = 20$ are smaller than the limit of 2^{10} configured in the ABSTL, we are practically comparing the PST to scanning.

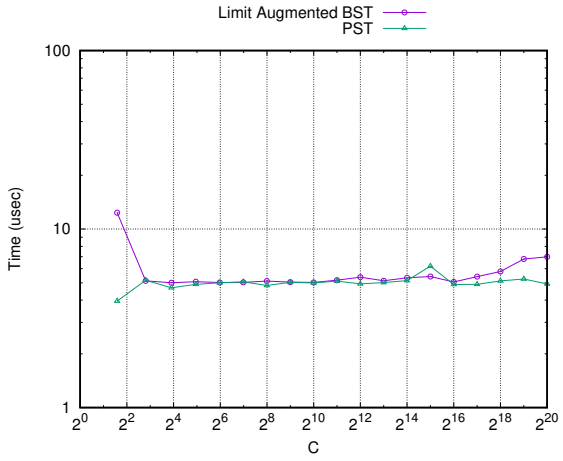

Figure 3.11: Queries of size $\log n$ with $n = 2^{20}$ varying colours.

3.4.4 Queries of size $\sqrt{n}$

We see on Figure 3.12, how slightly larger queries result in slower query times for the PST. Just as in the previous experiment, the ABSTL will only use the scanning strategy, as we never exceed 2^{10} elements.

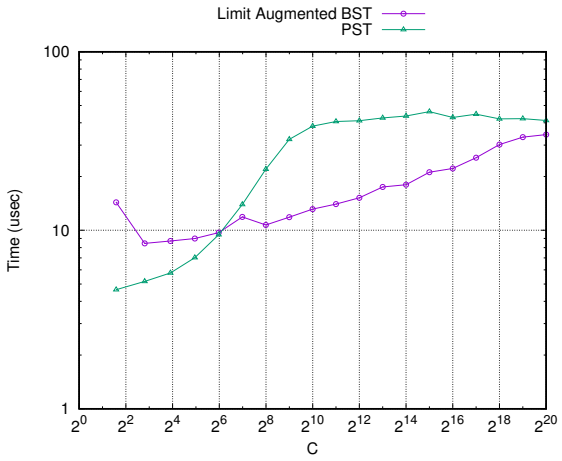

Figure 3.12: Queries of size $\sqrt{n}$ with $n = 2^{20}$ varying colours.

We see how the PST works well for smaller amounts of colours, and how the curve gets softly capped when we reach 2^{10} colours, meaning most of the colours are unique in the query. It can also be seen, how the ABSTL gets worse in performance as the uniqueness of colours increases, meaning the colour array for storing occurrences of colours becomes larger, resulting in more random look-ups and writes. Compared to the PST, we do not use a colour array, and we only report what we need. We see how the PST will have a small overhead on reporting a colour, as it needs to search in its heap structure, spending a pointer look-up on each report.

3.4.5 Varying colours

On the following figures, we have queried with a random query size, which should give us a glimpse of how the average performance for larger queries is. We see on Figure 3.13, how the PST is slightly slower than the ABSTL, for amounts of colours greater than 2^6. We already know that the pointer-based PST will be slower, as left and right pointers can sit anywhere in the virtual memory, compared to the sequential array of the ABSTL. Therefore, we can also see on Figure 3.15, how the ABSTL utilizes caches better, keeping cache misses to a minimal until the curve reaches $c = 2^8$ colours. Furthermore, we see how branch mispredictions increase as the amount of colours increases for the PST, as seen on Figure 3.14. Whenever we search and report in the PST, the branch predictor will have a 50% chance of guessing the correct path.

Combining the bad cache performance, and increasing branch mispredictions, we see how the PST is worse for reporting many colours.

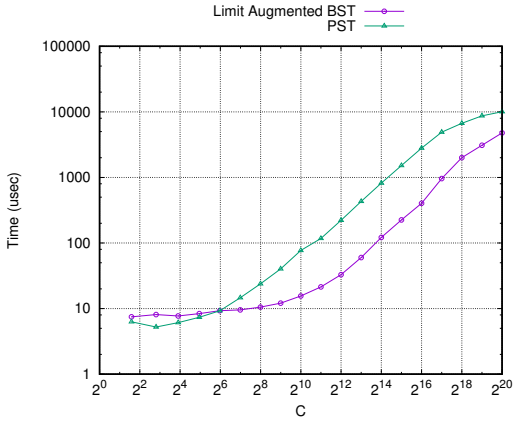

Figure 3.13: Varying colours, random query size, query time.

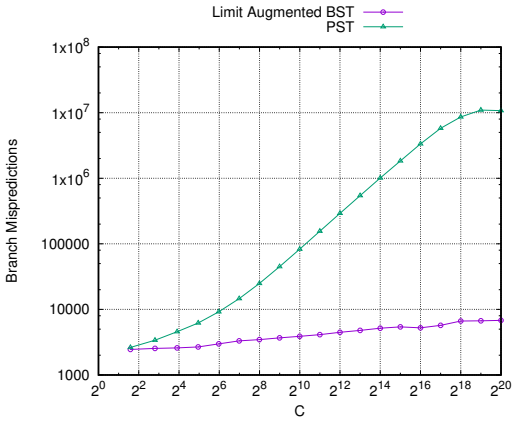

Figure 3.14: Branch mispredictions.

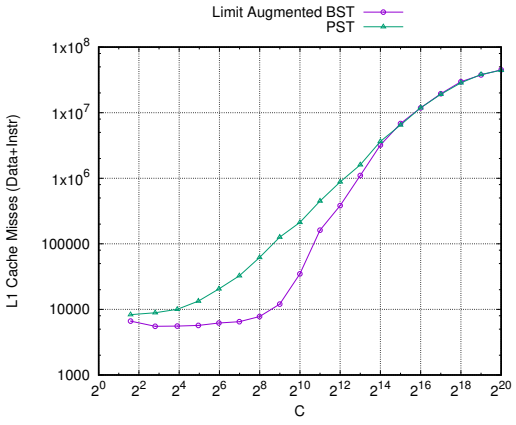

Figure 3.15: L1 cache misses.

Figure 3.16: Varying colours, random query size.

3.4.6 One colour, varying n

We created this experiment to see how much the search time influenced performance on large random query sizes. We see on Figure 3.17, that search times for both structures increase with n, as expected. We see how both structures are similar when searching. The most important thing to note, is that we observe how searching takes a tiny amount of time compared to reporting the colours, which we have seen earlier.

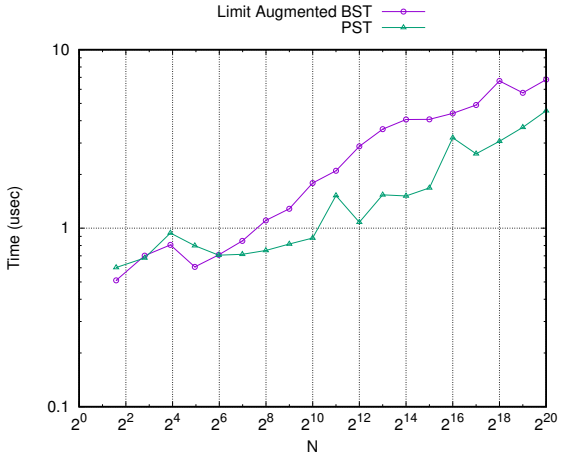

Figure 3.17: Setting $c = 1$, varying n, random query size.

We can conclude, that in terms of searching, the ABSTL and PST are close in performance.

3.4.7 Setting colours to be n

In Section 3.4.4, we argued that the uniqueness of colours affected the performance of the ABSTL. We have used a random query range in this experiment, just as in most of the others. With $c = n$, we expect that the structures will report only unique colours, which means we can see the performance of reporting points. The ABSTL is expected to be in a better position as the left and right sets are in a sequential layout, and the PST should perform slightly worse, as all reporting is done by traversing pointers.

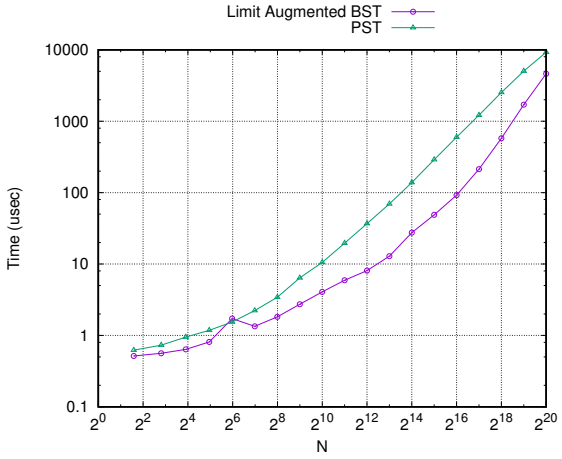

Figure 3.18: Setting $c = n$, varying n, random query size.

We see on Figure 3.18, how both implementations scale linearly according to the output size. And as expected, the ABSTL performs better than the PST.

3.4.8 Varying query range and colours

We know from earlier experiments in the last chapter, that query ranges mattered for the ABSTL. On the following graphs, we see whether or not it matters for the PST.

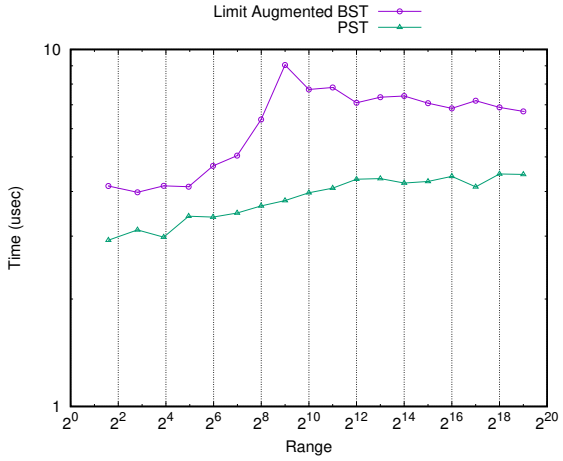

Figure 3.19: Setting $n = 2^{20}$, $c = 3$, varying range.

As we see on Figure 3.19, the PST performs well whenever there is a small amount of colours, as $c = 3$. We saw in Section 3.4.2, how picking $c \leq 2^7$ would let the PST perform better than the ABSTL. It is important to notice, that there is only ≈ 5 microseconds in difference of the two curves, which means both implementations are almost equally good. We also see how we should have picked a slightly smaller limit for the ABSTL in this case, as seen on the curve "peak" with a query range of 2^9.

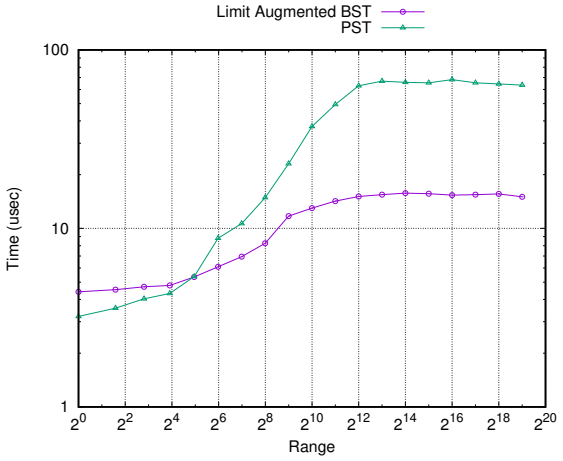

Figure 3.20: With $c = 2^{10}$.

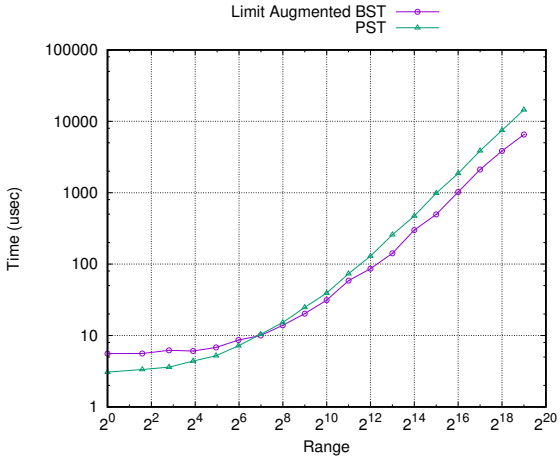

Figure 3.21: With $c = 2^{20}$.

Figure 3.22: Setting $n = 2^{20}$ and varying range.

We see on Figure 3.20, how setting $c = 2^{10}$ will decrease the performance of the PST. We assume that we report all unique colours, when we query a range of $\approx 2^{12}$, as the PST curve flattens out, and it has the same performance regardless of the query range afterwards. We

see how the same tendency appears on the curve of the ABSTL. It seems again, as we saw in Section 3.4.4, that the PST has an overhead on reporting colours, as it has to follow pointers, and might mispredict many branches while traversing.

On the last Figure 3.21, we see how the ABSTL performs slightly better than the PST. As the PST has an overhead on reporting colours, we see how it also affects the performance for larger ranges, as we are certain to report many unique colours, as $n = c$. However, the PST has a better performance than expected, as we assumed following pointers would give us a major decrease in terms of query time for large ranges and c.

3.4.9 Comparison of query times

In the following section, we will have a small comparison between the query times of the PST and the ABSTL. Just as in previous chapter, it makes sense to compare the different query ranges, to give an overview of how the implementations perform in a more practical situation.

We can see on Figure 3.25, that both the ABSTL and the PST handle small amounts of colours effectively, regardless of the query range. We recall, that the naive implementation was heavily dependent on the query range, in comparison to these two structures.

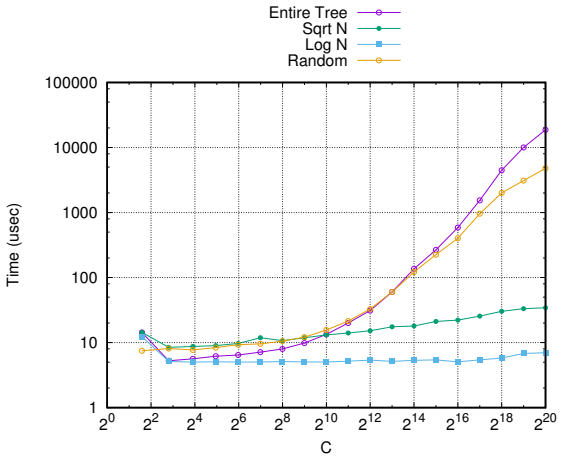

Figure 3.23: ABSTL.

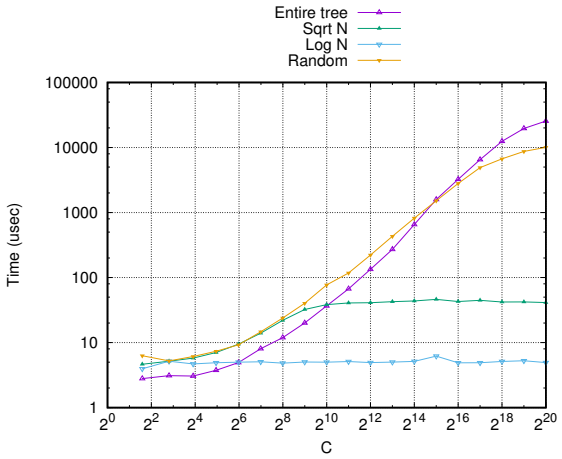

Figure 3.24: PST.

Figure 3.25: Query times with $n = 2^{20}$, showing the different ranges tested earlier.

We see how a short query range as $\log n$ gives the same performance, without depending on the amount of colours for both structures.

With greater ranges as $\sqrt{n}$, we see how the ABSTL performs better in general. We could conclude earlier that the PST has an overhead when reporting colours, and therefore we see how the PST performs worse already at this range. Additionally, we saw how the PST performed worse according to cache as well.

In the ABSTL, we see how querying randomly gives almost the same query times as querying the entire tree, as we will rarely traverse further down in the tree than the first levels, since the queries are random. We see how there is a $1/3$ chance[1] that the two query points respectively will belong to a different child node, causing the nearest common ancestor to be the root of the two children, and what we query. Further, this chance will decrease by $1/3$ each time we traverse to a child, as we traverse for a random query range.

It is notable how querying the entire tree and querying randomly have the same query times at the colour range from 2^{12} to 2^{14}, which is the turning point where the random query ranges get faster than querying the entire tree. As the ABSTL will use the root node to answer queries that span the entire tree, iterating the left and right sets, we know reporting small amounts of colours will be fast, as we do not search in the tree at all. When answering random queries, we generally traverse the tree a bit more, which explains the small difference

[1]The query $[x_0 : x_1]$ is either in the left sub tree, the right sub tree, or spans both.

in query times. We see how how 2^{12} is the turning point, where the occasional searches are paid with less colours to report, as the random ranges are smaller than the entire tree. It must still be noted that the size of the random queries will result in querying the root node in approximately 1/3 of all cases, which is why the query times seem very similar.

The PST has a similar tendency, and the turning point is located at $c = 2^{15}$. We can see how querying the entire tree is the fastest strategy for reporting points when $c < 2^6$. As querying the entire tree will result in a easy-to-predict path, it is expected that the branch predictor will perform well in this setting, and it could be an explanation for the great performance. Additionally, we expect that the turning point is the place where it becomes feasible to have smaller query ranges than querying the entire tree.

3.4.10 Comparison of branch mispredictions

As mentioned when we compared the query times, we expected the branch predictor of our test machine to predict the outcomes with better precision, when querying the entire tree. As we can see on Figure 3.28, it seems to be the case for both implementations, that when querying a predictable path, great performance is achieved.

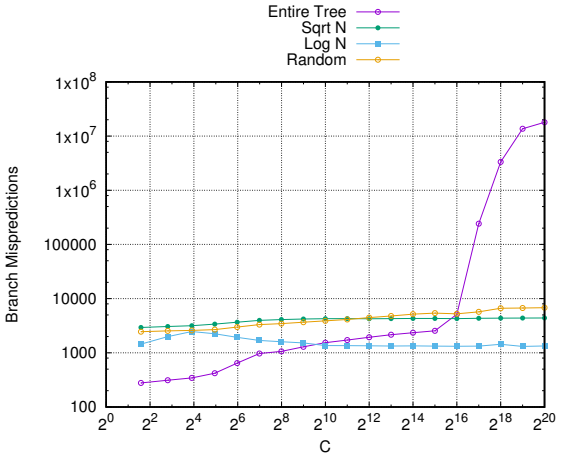

Figure 3.26: ABSTL.

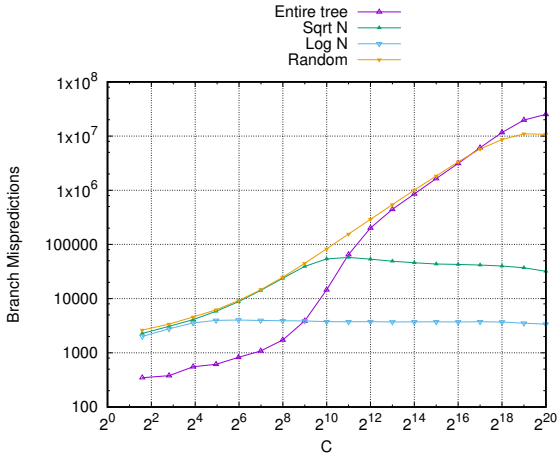

Figure 3.27: PST.

Figure 3.28: Branch mispredictions with $n = 2^{20}$, showing the different ranges tested earlier.

For the PST, we see how querying randomly gives us the most branch mispredictions for all cases, until we reach $c > 2^{16}$, where there will be more than $\frac{2^{16} \cdot 100}{2^{20}} = 6.25\%$ chance of getting a unique colour, meaning that the amount of duplicate colours decrease, and the random queries will report slightly less colours than queries on the entire tree. As we query more of the PST tree structure as the colour count increases, it follows that the random queries will report less colours than the queries on the entire tree, which also affects the graph.

On the other hand, we see how the branch mispredictions increase dramatically for $c > 2^{16}$. This is due to an `if` branch that checks if a colour already exists in our colour array, which at small c/n will be predicted correctly more often.

If we compare the smaller queries, we see how the PST has a larger amount of mispre-

dictions than the ABSTL. This is due to more traversal in the PST, as all data is stored in individual nodes.

We can in general conclude, that the PST has a larger amount of branch mispredictions than the ABSTL for all cases. As seen in our comparison of query times, the PST had the worst general query time as well, which could follow from the amount of branch mispredictions it had.

We see how the ABSTL generally behaves better according to the branch prediction, as the ABSTL does not have to recursively visit its sub tree like the PST does, but instead searches its tree and iterates two lists.

3.4.11 Comparison of cache misses

In this final section, we will compare how the two implementations behave according to cache. The L1 cache is the smallest cache of the CPU, disregarding the CPU registers, and therefore the second best option for storage of instructions and data for fast access. When dealing with experiments and large input sets, we usually see a break on the curve, where the input sets cannot fit into the cache anymore, and the CPU starts searching outside the cache to get the next data, resulting in cache misses. On the figures presented, we will see readings of L1 cache misses, which will be analyzed. As the L2 cache will be physically further away from the main computing power, it is expected that a L2 cache miss will be more punishing. However, L2 cache misses are omitted, as we are interested in how well scanning in the local ranges is handled.

On Figure 3.31, we can see breaks at $c \approx 2^{10}$ and $c \approx 2^6$ for the ABSTL and PST, respectively. We observe that the curve for the PST seems to break at smaller c, due to the non-sequential layout of the PST structure. This is due to searches taking random paths, and that the pointer structure of the PST is inefficient when searching in terms of cache.

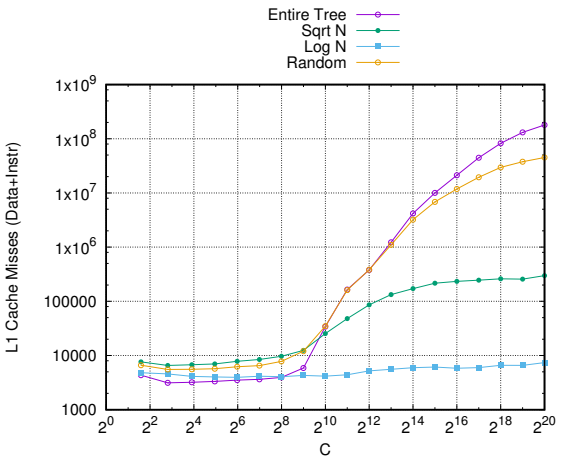

Figure 3.29: ABSTL.

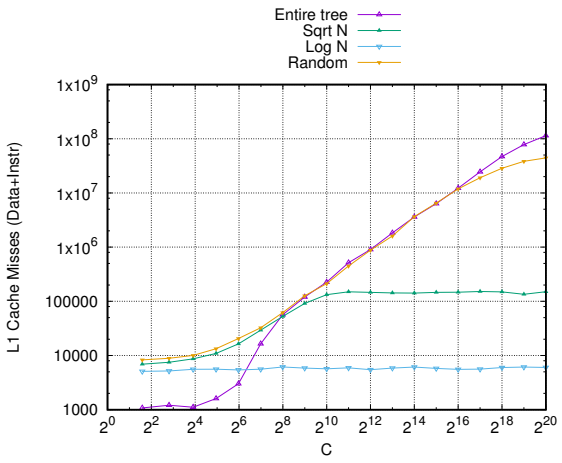

Figure 3.30: PST.

Figure 3.31: L1 cache misses with $n = 2^{20}$ showing the different ranges tested earlier.

Querying the entire tree seems to give the best performance for small c in the PST, which seems odd when comparing against the ABSTL with a good sequential layout. As we initialize

the tree, we always recursively initialize the left node first in the PST, which means that the path from the root to the leftmost node in the PST is expected to be allocated sequentially on the heap, just as in an array.

Now we will review how much data we can have in the L1 cache in the best case. We know that each node of the ABSTL is 36 bytes large, in comparison to a PST node which was 32 bytes large. We therefore know that the ABSTL node is $\frac{36 \cdot 100}{32} = 11.25\%$ larger than the PST node, which implies a slightly larger overhead for how many nodes we can fit into the L1 cache. Our testing machine has a 32kb large L1 data cache, which can contain $\frac{32 \cdot 1024}{32} = 1024 = 2^{10}$ nodes in the best case for the PST, and $\frac{32 \cdot 1024}{36} \approx 889 \approx 2^{9.8}$ nodes for the ABSTL, disregarding the coloured sets. We must be aware, that the PST searches and reports using nodes only, and the ABSTL searches followed by two sequential iterations of the left and right sets. As we set $n = 2^{20}$, we will use at most 60 iterations (three searches) for the ABSTL before finding the left and right colour sets, whereas the PST will spend at most 40 iterations (two potential searchpaths) plus all the additional paths traversed when reporting colours between the searchpaths. We can conclude, that we can fit the search in cache, but we will likely experience a cache miss for each node we visit for both implementations. The ABSTL has the inorder cache layout, which was the worst layout for cache-efficieny, and the PST has a pointer based structure, where a cache miss is almost certain whenever the path of the right child is taken.

As we saw on the graphs, the ABSTL breaks at around $c \approx 2^{10}$ which has a correlation to the space usage. On the graph of the PST we notice an earlier break, as we must visit nodes according to the amount of colours, which the PST does slower. Furthermore, we see how cache misses increase for the PST as the searches become larger because the amount of unique colours increases.

On our testing machine, we have a cache line of 64 bytes size, which means that we can retrieve two consecutive PST nodes in one CPU cycle, compared to a single ABSTL node. Since we discussed earlier that the searches mean little in the ABSTL when dealing with larger ranges than 2^{10}, we can assume it is reporting the colours that takes up most time. The ABSTL left and right sets consist of small lists of points with colour of size $2 \cdot 4 = 8$ bytes each. We see how $\frac{64}{8} \approx 8$ points can fit in a single cache line, as compared to two in the PST if iteration was sequential. Iteration of these sets will therefore be quick compared to the searches in the PST. We note, that the PST does not guarantee a sequential layout on the heap at any time, but we expect that the leftmost nodes are allocated sequentially on the heap, which will result in the cache line to retrieve two consecutive PST nodes in one CPU cycle in most cases.

3.5 Wrapping up

To conclude the experiments, this section will give a quick overview of the differences between the ABSTL and PST.

We have seen how the ABSTL performed slightly worse in construction times, as the PST did not have an extra multiplier of $\log c$ overhead in space usage.

We also saw how the PST outperformed the ABSTL in query time, when querying the entire tree for colours up to $c = 2^6$, and we reasoned that it was branch mispredictions and bad cache locality, that caused the PST to slow down after this point. We also saw how

this was the case for queries of size $\sqrt{n}$, although the slowdown was only slight. When using random sized queries, the same tendency occurred, and we saw how the PST had an excess amount of branch mispredictions.

When we used a single colour and varied the input size, we saw how the PST and the ABSTL had similar query times.

When we started varying on the query ranges, we saw how the PST performed well for $c = 3$, but slower for $c = \sqrt{n}$. We also saw how the two implementations were almost equal, when we had to report a input set of unique colours.

In the two last sections, we compared query times for all experiments, and we concluded that the PST was better for small amounts of colours, but as the set of unique colours increased the ABSTL was the best match. By comparing branch mispredictions, we saw how the PST generally performed worse, which was one of the reasons why it was slower for larger amounts of unique colours. Finally, we compared L1 cache misses, and saw how the PST performed worse in cache, even though the ABSTL had 11.25% larger nodes.

It must however be noted, that the node size of the ABSTL only had an influence on searching, and not reporting colours.

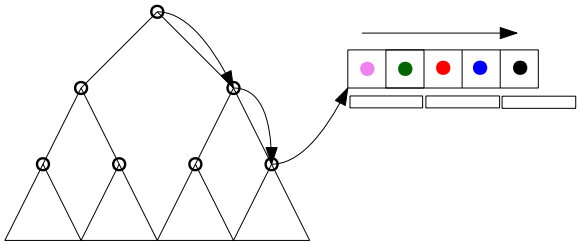

Figure 3.32: ABSTL.

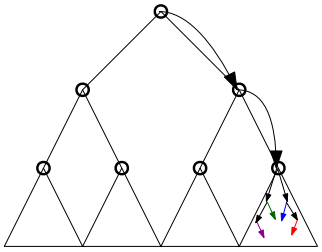

Figure 3.33: PST.

Figure 3.34: How the two structures search and report.

As we can see on Figure 3.34, the ABSTL will search and then report points according to the right and left sets in a sequential layout, whereas the PST will report points while searching. We mentioned earlier, that the PST relies on following pointers, as this is how all traversal and reporting is done in the tree. This gives a slowdown, compared to the ABSTL searching first and then reporting colours according to the array structure. The array structure of the ABSTL is more predictable, as we take the same branch while scanning for points, compared to the PST, where reporting nodes is done by traversing sub trees and taking random paths.

The PST performed better than expected for queries, especially considering that we used a pointer based layout. The ABSTL will still give a generally better query performance, but it will use more space.

As the PST uses $\mathcal{O}(n)$ space, it will be preferred in low memory implementations, as opposed to the ABSTL using $\mathcal{O}(n \log c)$ space. Implementation-wise, we used 32bytes per node in the PST, and 36 bytes per node in the ABSTL, giving us trees of size $32 \cdot 2^{20} \approx 33.5$ and $36 \cdot 2^{20} \cdot \log c \approx 37.7 \cdot \log c$ megabytes, respectively, for our experiments. We do see, how increasing c to 2^{20} will make the ABSTL use $37.7 \cdot 20 = 755$ megabytes of space in the worst case, which makes the PST better for small-memory environments such as a smartphone, a raspberry pi or an embedded system.

Chapter 4

Conclusion

This thesis addressed the two-sided one-dimensional categorical range query and three-sided two-dimensional orthogonal range query problems, solving the first problem using a simple solution based on searching and iterating an array, a more complex solution (the augmented BST) with and without optimizations, and finally the priority search tree for less space usage. The PST was originally designed for the second problem, but we made a reduction from one-dimensional points with colour to two-dimensional points without colour, without spending additional space or query time.

First, we defined the two problems formally, and gave the theoretical solutions for the augmented BST from the paper by [SJ05]. Then we implemented the augmented BST, along with the naive solution in C++, accompanied by multiple analyzis of the two implementations proving their space bounds and query bounds. The augmented BST was expected to have a query time of $\mathcal{O}(\log n + c)$, whereas the naive implementation would have a query time of $\mathcal{O}(n)$ in the worst case. The space consumed by the augmented BST was expected to be $\mathcal{O}(n \log c)$, whereas the naive implementation had a space usage of $\mathcal{O}(n)$.

Then, we experimented with the augmented BST and the naive implementation, to see how well the augmented BST performed against the trivial solution. We found that short query ranges made the naive implementation perform better in terms of query time, but at greater ranges, the augmented BST was the best. We saw how a small overhead in query time made the augmented BST slow for small input sizes of points, compared to the naive implementation. However, it did not matter for larger query ranges, and we saw how small amounts of unique colours would devastate the performance of the naive implementation, when compared against the augmented BST.

Afterwards, we saw how we could use the fast performance for short query ranges in a new implementation, the ABSTL. In the ABSTL, we could set a query limit of when to use the naive implementation, and when to use the augmented BST for querying. We saw a performance increase in the general case, but we still had a space usage of $\mathcal{O}(n \log c)$ of the augmented BST. We then concluded the second chapter of the thesis, with room to improve.

In the third chapter, we introduced the priority search tree from [BCKO08, ch.10], along with its theoretical solution. We also included a reduction, such that we could use the PST to solve the one-dimensional categorical range query problem. As we saw from earlier, the augmented BST had an optimal query time of $\mathcal{O}(\log n + c)$, but with a $\mathcal{O}(n \log c)$ space usage, that we wanted to dispose of. We saw how the PST had a promised query time of

$\mathcal{O}(\log n + k)$, and a space usage of $\mathcal{O}(n)$, which seemed as a good pick for further testing. Additionally, the reduction would give us a query time of $\mathcal{O}(\log n + c)$, as we could arrange the points such that we did not report duplicates.

We implemented the PST using a pointer based approach, instead of the array-based layout the ABSTL had, and expected the PST to perform slightly worse, as we had no control of how nodes were located in memory. Additionally, we implemented the reduction as a part of our construction step in the PST, where we could seamlessly translate one-dimensional points with colour, to two-dimensional points. In our experiments, we compared the ABSTL in its best configuration according to our testing machine and the PST.

We saw how the ABSTL worked better in the general case, although the PST was better for small amounts of unique colours that should be reported. Due to the tree-structure of the PST, we had no guaranteed sequential arrays to traverse, which meant that the ABSTL performed better for larger amounts of unique colours. We measured branch mispredictions and L1 cache misses, and most of our results implied that the PST performed worse in terms of cache, and that using a tree-structure to report points gave the branch predictor of our machine a hard time, resulting in a slowdown as well. The PST performed good for queries of small size, almost comparable to the ABSTL in most cases.

The PST did however have a smaller memory footprint. E.g. for $n = 2^{20}$, the ABSTL used approximately 755 megabytes of memory in the worst case, compared to the PST using only approximately 37.7 megabytes of memory in any case. We concluded that the $\log c$ of the theoretical bound mattered more than expected. To see a summary of query times and real space usage, refer to Figure 4.1.

In the end, we learned that we could trade some space usage for better query times using the ABSTL. On the other hand, we could use the PST for reasonable query times but better space usage. We saw how the PST and ABSTL were almost equally good when handling short query ranges or input size. Additionally, we saw how the naive implementation was fast for short query ranges as well.

Depending on the amount of data that must be processed, the naive solution will be preferred for sizes smaller than $n = 2^{10}$, as it will have the same or better performance than the augmented BST, and it will be faster to implement. The ABSTL and PST took the same time to implement, so the ABSTL will be preferred for solving the one-dimensional coloured problem if query time matters the most.

We learned how locality of data made a big difference when reporting the colours of the input set. We saw how searching in both the ABSTL and PST did not matter much for the rest of the query time. We also saw how the points in the PST were reported by traversing the tree, instead of an sequential structure, which made reporting expensive.

Table 4.1: Summary of complexities.

	Space	**Query**	**Space** ($n = 2^{20}$)
Naive (Search and Scan)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	8.38 MB
Augmented BST	$\mathcal{O}(n \log c)$	$\mathcal{O}(\log n + c)$	755 MB
Priority Search Tree	$\mathcal{O}(n)$	$\mathcal{O}(\log n + c)$	37.7MB

4.1 Future Work

This final section will cover subjects that would be interesting to research for possible improvements of performance in the current codebase, and other similar problems that would be exciting to study.

First, space usage is something we could improve on. A general concern was how we stored the colours which were reported in the ABSTL. We settled with an array, containing references to colours, spending $\mathcal{O}(c)$ space. As each colour was in a interval of $[0 : c]$, we could have used an array of booleans instead, also spending $\mathcal{O}(c)$ space, but with a smaller overhead. Additionally, it was mentioned in [SJ05], that it was possible to reduce space usage of the ABSTL to $\mathcal{O}(n)$, which hopefully would improve the large usage of memory the ABSTL has.

Secondly, for the PST, we could implement a "bucket" scheme, just as we usually do when in the I/O model, such that we can have better cache performance. We could additionally have implemented the *external priority search tree* from [ASV99] promising a query time of $\mathcal{O}(\log_B n + k/B)$ I/Os and linear space usage, and compared with our current solutions.

Thirdly, we could make small optimizations according to cache and the branch predictor. Whenever we added a colour to the array, we wanted to check if it existed already. We could have written to the array regardless, saving some branch mispredictions that was caused by guessing the `if` branch. Instead of only measuring L1 cache misses and Branch mispredictions, we could have measured L2 and L3 cache misses as well. As these caches are further away from the CPU cores, it is expected that we will see some other behaviour which could be optimized accordingly. We could have increased n for larger trees, such that the PST would not fit into the L3 cache, as we expect that the PST would have worse performance in that case.

Finally, search time is a subject we could improve on, but we only saw how searching mattered the most with small c. We use $\mathcal{O}(\log n)$ time when searching for the nearest common ancestor in the ABSTL. Instead, we could implement finding the nearest common ancestor using some bit-shifting tricks from [HT84], giving us $O(1)$ time to find the ancestor, still using the static tree. Additionally, we could have tried other array layouts as the *Breadth First Search* layout, and the *van Emde Boas Tree* for searching in the ABSTL, as in [BFJ02].

Chapter 5

Appendix

5.1 Project Structure

This section will present the folders with code briefly, to give the reader an overview of what has been done.

5.1.1 Dependencies

The project is built with CMake [cma16], and experiments depend on the PAPI library [pap16]. Additionally, to generate plots of the experiment files, gnuplot [gnu16] is required. The project compiles and runs without PAPI installed, but it is required for experiments.

5.1.2 Files

The code is available at `http://cs.au.dk/~tt1/files/thesis_code.zip`.
The source folder contains the following folders.

```
src
├── bin
├── cmake
│   └── modules
├── experiment
├── lib
│   ├── include
│   └── src
├── plotdata
├── plots
├── test
├── CMakeLists.txt
├── create_exps_plots.sh
└── make_dbg_then_test.sh
```

The folder *lib* is where the implementation source code resides. As the project is made with C++, header files and code are separated into two folders, *include* and *src*, respectively. Additionally, the folders *test* and *experiment* contain source code for the unit tests and the experiments shown in this thesis, respectively. The binary folder *bin*, is where all binaries

will be generated, regardless if it is a test or experiment. The *cmake* folder contains a small module, which determines if PAPI is installed. Lastly, the *plots* folder is for generated images, generated by the experiments with the data in *plotdata*, and gnuplot files residing there as well.

5.1.3 Compiling, building, testing

The project can be compiled using the bash file, `make_dbg_then_test.sh`, which removes the *build* folder if it exists, generates Make files using CMake, builds the code, and finally runs the unit tests. Running the latter file with the "q" argument will skip generating Make files, if there is a need for a faster build.

To run experiments, use the bash file, `create_exps_plots.sh`. Compared to the file used for testing, this one will not create a debug build, but rather compile the project with optimization flags. Additionally, it will run the experiments, and generate plots afterwards. Running this file the with "q" argument skips building files, and attempts to generate new plots, from already existent data.

5.2 Testing Environment

The computer used to run the tests have the following specifications.

Listing 5.1: Machine info

```
Model: Lenovo S540 Custom Laptop
OS: Antergos
Kernel: x86_64 Linux 4.2.5-1-ARCH
CPU: Intel Core i5-4210U CPU @ 2.7GHz
RAM: 8GiB SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
Compiler: GCC 5.2.0
Papi version: 5.4.0
Disk: 256GB SSD
```

5.2.1 CPU Info

Listing 5.2: CPU info

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 69
Stepping:              1
CPU MHz:               768.000
BogoMIPS:              4788.82
Virtualization:        VT-x
L1d cache:             32 KB 8-way set associative
L1i cache:             32 KB 8-way set associative
L2 cache:              256 KB 8-way set associative
L3 cache:              3072 KB 12-way set associative
Cache Line Size:       64B
64-byte Prefetching
Data TLB:              1-GB pages, 4-way set associative, 4 entries
Data TLB:              4-KB Pages, 4-way set associative, 64 entries
L2 TLB:                1-MB, 4-way set associative, 64-byte line size
NUMA node0 CPU(s):     0-3
```

In short, the interesting parts are the combined 64K L1 cache and the 256K L2 cache for each core. Our L3 cache is 3072 KB, which supports holding $\frac{3072 \cdot 1024}{4} = 786432 \approx 2^{20}$ integers, where 4 is the integer size in bytes. We also notice that this CPU has 2 cores, each core has two additional threads as of the hyperthreading technology that Intel uses. The cache-line size is 64 bytes, which is common for most of the new Intel chipsets. The CPU frequency is misleading as the clock speed varies according to workload on the computer.

Bibliography

[AGM02] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. *Algorithms — ESA 2002: 10th Annual European Symposium Rome, Italy, September 17–21, 2002 Proceedings*, chapter Range Searching in Categorical Data: Colored Range Searching on Grid, pages 17–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[ASV99] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 346–357, New York, NY, USA, 1999. ACM.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[BCKO08] 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.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 39–48. Society for Industrial and Applied Mathematics, 2002.

[cma16] Cmake, 2016. [Online; accessed at `https://cmake.org/`, 18-April-2016].

[gnu16] gnuplot, 2016. [Online; accessed at `http://www.gnuplot.info/`, 18-April-2016].

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[JL93] Ravi Janardan and Mario A. Lopez. Generalized intersection searching problems. *Int. J. Comput. Geometry Appl.*, 3:39–69, 1993.

[LvW13] Kasper Green Larsen and Freek van Walderveen. Near-optimal range reporting structures for categorical data. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 265–277, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics.

[McC85] Edward M McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[Mor03] Christian W Mortensen. Generalized static orthogonal range searching in less space. Technical report, Technical Report TR-2003-22, The IT University of Copenhagen, 2003.

[pap16] Papi — Performance Application Programming Interface, 2016. [Online; accessed at `http://icl.cs.utk.edu/papi/`, 18-April-2016].

[SJ05] Qingmin Shi and Joseph JaJa. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Information Processing Letters*, 95(3):382 – 388, 2005.

[Tar79] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110 – 127, 1979.