Advanced Programming and Data Structure
Course 2022-2023

Second Semester Project – Non-linear Data Structures

Knights of the Hash Table

	Login	Name
Students	miguelnicolas.palau	Miguel-Nicolás Palau Lessa
	adrianjorge.sanchez	Adrián Sánchez López
	alex.ciria	Alex Ciria Roca
	m.soto	Marc Soto José

Date	26/05/2023

Table of Contents

Programming language choice

Primarily we have chosen Java over other programming languages to practice in Java, although this is the main reason it is also a plus that Java, specially in IntelliJ, lets you take shortcuts and has a lot of prebuilt methods that you can use to speed up the time it takes you to program.

It is true however that compilation in C can be optimized to be faster than Java since it doesn't need to be interpreted and gets translated to machine code straight after compiling. Although this seems like an advantage when one of the aspects we want to test with this project is time efficiency, we are only analyzing the efficiency of each algorithm, so this doesn't really matter as long as you can appreciate how effective one is compared to another and how it varies with the input. C is also more flexible with memory usage however, once more, this doesn't contribute to the objective of the investigation.

Moreover, Java makes it easier to work with lists of objects. In our case, we knew we were going to deal with a large number of boats, sailors, and centers; and Java lets you add objects to a list, swap them, and remove them in an easy, intuitive, and fast way.

Custom List

In order to help ourselves and ease the managing of data during the development of the different algorithms, we decided to implement our own custom auxiliary data structure, similar to a List but also with some functionalities of a Queue.

To start with a skeletal implementation, our CustomList<E> extends AbstractList<E>, which provides us a template with the most common functionalities a List provides. Our class has three attributes: MIN_SIZE, arr and size. The first one describes the minimum size of the array, used on creation, the second one is the base array where elements will be stored, it is of type Object[] and finally, size holds the current size of the array. It is useful to know at all times the size to avoid constantly looping till the end to count the total elements.

Our CustomList has the following functionalities, enabled by different methods:

- public E get(int index)
- public int size()
- public void addLast(E obj)
- public void addFirst(E obj)
- public E pop()
- public E popLast()
- public E getAndRemove(int index)
- public boolean remove(Object obj)
- public void removeByIndex(int index)
- public void clear()
- public CustomList<E> clone()
- private void shiftRight()
- private void resize()

Graphs

Structure Design

To decide between implementing the graph using an adjacency list and an adjacency matrix we looked at the dataset provided, which stores the number of vertices and edges we will load in our graph. To see how dense will be the graphs we are going to be working with. By comparing the number of edges each dataset has, to the theoretical maximum ($e = v * (v - 1) / 2$).

	Vertices	Edges	Maximum	Capacity used
graphsXXS	8	22	28	78 %
graphsXS	16	80	120	66,66 %
graphsS	32	374	496	75,40 %
graphsM	64	1284	2016	63,69 %
graphsL	128	5494	8128	67,59 %
graphsXL	256	24546	32640	75,20 %
graphsXXL	512	104538	130816	79,91 %

As it can be seen all the graphs are pretty dense, as they have , approximately, 3/4 out of the maximum number of edges.

Even though they are not sparse graphs, which will be better implemented in an adjacency list (because it will waste a lot of space in a matrix), we have decided to implement it with an adjacency list. Because we will still be saving space(1/4 of the space compared to the adjacency matrix) and it is easier to look for the adjacent nodes of a given node. Which is better than being able to check whether 2 nodes are adjacent in θ(1). As in most of the algorithms, rather than checking if 2 nodes are adjacent, we will be looking for the best node from the adjacents in terms of weight and/or given some restrictions.

Finally, to implement it in an adjacency list, apart from the vertex class / structure which stores the points of interest data and a list of its edges. As each edge has different weights, we have created another class / structure for the edges. Which stores the different weights as well as a reference to the adjacent node.

Implemented Algorithms

Kingdom Exploration

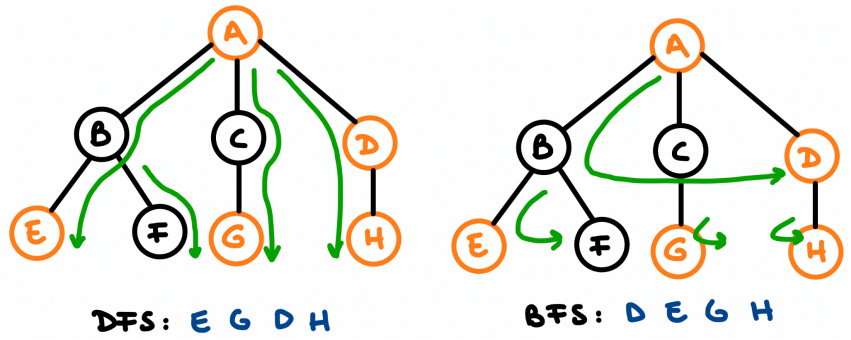
The following functionality gives the user the option to select a point of interest out of a given dataset to find all the nodes that belong to its same kingdom, prioritizing those that can be reached without crossing its border.

Having already organized our data in a graph, we then had to decide which algorithm would be the most appropriate to traverse it, as we have to search for all the points of interest's kingdoms in the graph to compare them to the input one.

We considered the DFS (Depth First Search) and the BFS (Breadth First Search) algorithms to perform this task. However, after thoroughly analyzing both of them, we concluded the BFS algorithm would suit our program better. This is because the BFS algorithm consists in visiting all the adjacent nodes of a point of interest rather than expanding all the nodes found as the DFS algorithm does.

As mentioned above, we want to prioritize the nodes that can be reached within the same kingdom. In other words, we want to first analyze all the nodes that are connected to the input one (to ensure the connection between them). If we find adjacencies with the same kingdom, their connections will then become the next priority (as we know they can be reached from the input without leaving its kingdom). Hence, the BFS algorithm was the most appropriate to perform this task.

To prove this, below you can find an image of an example graph which represents a possible scenario.



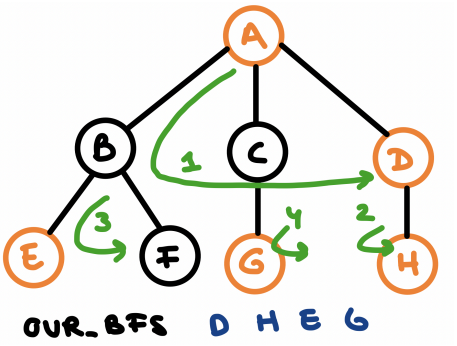
Given an input node A and knowing the orange nodes belong to the same network, we can intuitively see how nodes D and H should be the ones we prioritize (as they are the only orange nodes directly connected to A).

However, if we followed a DFS approach, we would expand all the nodes until we get to a point with no other connections (like node E). At this point, we would recursively go back

and explore the next possible path (F). This method would lead us into finding the nodes from the same kingdom in the order specified above: E, G, D, H. Clearly, this is not what we want, as the nodes D and H should be the first ones we display, not the last.

On the other hand, if we followed a BFS approach we would first go through all the adjacent nodes of A, finding D as the first orange node. Once all the adjacencies are checked, the "next level" would be explored. Notice how this simple implementation of the BFS algorithm is still not giving us the desired output, as node H should have been prioritized.

To design our final algorithm, we took a BFS approach while also adding priorities, ensuring that the adjacent nodes from an orange one were checked before the adjacent to black one. The following image represents how our algorithm would traverse the graph:



Once we had already decided how our algorithm would work, we defined the following pseudocode which we then used to code it.

```
function our_bfs(g: Graph, v:vertex)
    list: List<Vertex>
    v= list.addFirst()
    v.visited := TRUE
    while ¬list.empty do
        current := list.getFirst()
        for adj in g.adjacents(current) do
            if ¬adj.visited then
                adj.visited := TRUE
                if adj.same kingdom(current) then
                    list.addFirst(adj)
                else
                    list.addLast(adj)
                end
            end
        end
    end
end
```

As you can see, our code is based on a list of nodes where the input is first added. While there are still elements to check, the algorithm takes the first node on the list (marking it as visited) and gets its adjacent points of interest. It analyzes all of them and, if they haven't been visited yet, it checks whether their kingdom matches the input one. If it does, it prioritizes their evaluation by adding them to the beginning of the list. On the contrary, if they aren't from the same kingdom, it adds them to the end of the list. It repeats this process until there are no elements left on the list (assuming the getFirst() method removes them from the list).

The pseudocode for the BSF algorithm proposes to use a queue to perform this task. However, as we had already implemented the above-mentioned CustomList we decided we could use it, as we had defined the addFirst(), addLast(), and pop() methods.

We are aware of the problems the previous loop condition would give for disconnected graphs, getting stuck on the loop rechecking all the connections without finding the unvisited nodes (as these ones are only taken from the adjacent relations). However, as all of the nodes on our datasets have at least one connection, we believe this won't be an issue for our case. Even though it should be taken into account for different scenarios.

Common Journeys

We are asked to find the common journeys followed in aviary messaging without considering bird types. For this, we must find the journeys that will leave all points of interest interconnected, minimizing the distance between them, as the points of interest may have many interconnections between them with many different distances associated to each connection.

We are working with a weighted, undirected graph and we want to minimize the weights, thus, we consider finding the Minimum Spanning Tree of our graph will give us exactly what we want. The outcome will be another graph with all the nodes or vertices of the original graph but not with the same connections or edges. All vertices will be connected in a way that minimizes the number of edges required to do so, describing a Spanning Tree. A Minimum Spanning Tree has the added particularity of selecting the edges with the minimum available weights resulting in the minimisation of the sum of all the edge weights together.

In our case, we will obtain a graph with all the points of interest interconnected using the minimum number of edges, these with the lowest distance values possible. This can be visualized as a set of points interconnected with the shortest paths and no visible cycles or loops.

We've studied two different Greedy approaches to find a single MST, there may be multiple combinations for the same graph that apply as MST but we will focus on finding simply one that follows the conditions explained above. We have to choose between the Prim algorithm and the Kruskal algorithm, the first is preferable, when it comes to prioritizing efficiency, when vertices are heavily interconnected, as the cost is majorly affected and increased by the number of vertices. The Kruskal algorithm, on the other hand, is more notably efficient with few edges, as the cost mainly increases with the number of edges.

As we've calculated in the Structure Design section above, the edge quantity, taking into account the theoretical maximum, is around 70% for all datasets, meaning that our nodes will be highly interconnected. Knowing this, we cannot doubt choosing Prim as the algorithm to implement.

Our implementation starts with an empty graph and adds the first node in the vertex list. From this node, its edges are retrieved and the edge with the lowest distance is selected to add it, together with the destination, to the new graph. Then, another point that has not yet been visited will be selected as the next node to check. This is looped until all vertices from the original graph are present in the MST graph.

Looking at our code, to get into detail on how the implementation works, we work with two CustomLists, one holding all the points of interest of the original graph, and the other one holding the POIs of the new Minimum Spanning Tree graph. The algorithm consists of a while loop that will run until the size of both lists match, that is, when all points from the original graph are present in the MST also.

Inside the while loop, we get a point of interest from the original list, we will sequentially traverse the list through the loop execution, meaning that we will start with index 0 and after each iteration it will be increased to get the next point. With our point, we check if it has been visited, if so, it means that it is already in the MST so we ignore it and skip to the next iteration, otherwise, we continue with that point. Next, we run a for loop through the point's edges to find the edge with lowest distance value.

We now have the next edge to add to the MST, and whether the destination has been visited or not, we will add the vertex. Before adding anything, we will clear the edges of the current point and leave only the shortest edge we found. We will then add the point to the MST marking it also as visited. Next, we will check if the destination point should be added, if it has not been visited, we will do the same as previously, clear its edges to leave the one we found to be the shortest, mark it as visited and then add it to the MST. This loop will finally add all the points interconnecting them with minimized edges.

Premium Messaging

For this functionality we want to find the fastest possible route from one point of interest (POI) to another in order to deliver the user's message as fast as possible. In order to do so we have to also choose the bird which will complete this route the fastest and we will have to take into consideration if the bird will carry a coconut. Due to the fact that the bird's time to fly from one place to another is not directly proportional to the other we have to run the algorithm for each bird separately and select the best result which will be a list of POIs.

Essentially birds fly in a straight line from one POI to another, not being able to go from one POI to another if they are not connected. Due to us using a graph to represent each POI (node) and each connection (edge) we have deemed it most appropriate to approach this problem with Dijkstra's algorithm.

Dijkstra's algorithm consists of starting at a node and looking at all its adjacent nodes that have not been visited. In our case we will consider an adjacent node one that is , firstly, connected (they share an edge in common), secondly if the node we are going to has a climate that suits the bird, and lastly, if the bird is carrying a coconut an adjacent node needs to be at most 50 km away as it cannot fly more distance at a time due to the weight. For each adjacent node we will go to the node whose path has least cost (in our case the cost is the time taken to fly from one node to another).

Now that we have moved to this node we will once again look at all its adjacent nodes but we do not decide which one to go to by selecting the smallest cost from the current node to the adjacent ones, but instead from the starting node to the adjacent nodes, we do this by keeping track of the journeys to each node, this is why before starting on the first node we will have created two different arrays, one that records the cost from the first node to the adjacent ones we are looking each time we look at an adjacent node that we previously had never looked at or have now looked at by using a path with a smaller cost, we will update the details of the 'journey' to that node.

We will then, as we said, move on to the node with the shortest time journey from the initial node and repeat this process, slowly building a list of routes to all or most nodes in the graph until hopefully ending up in the final node, knowing the time/cost it took and the list of nodes that we needed to visit in order to get to the destination as fast as possible. The actual way in which we will end the process of going from one node to an adjacent one is by checking if there are any more nodes that are unvisited that we know how to get to, this means that if the we don't have a journey established for it we cannot use that as a current node as we do not know how to get there, this is either because we haven't evaluated enough nodes yet, because there is no existing path from the starting node or because we have pruned it in some way.

Once we have a route for the destination (but we do not know if we could possibly get a better one) we can decide not to go to a node if this means the cost of going to that node is higher than the cost we have of getting to the destination, this could potentially prune a lot of routing in large graphs.

Result Analysis

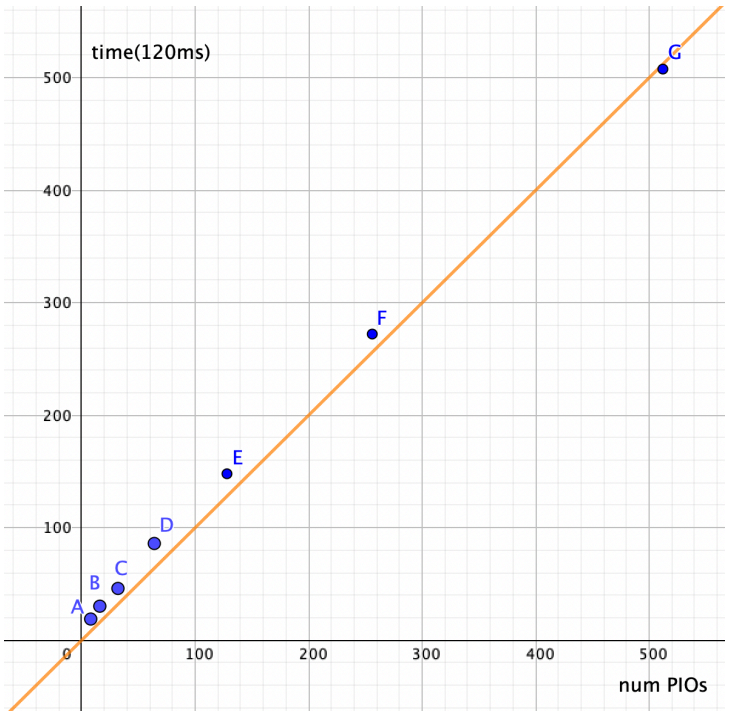
We measured the average time that it took for our program to execute the different algorithms for each of the given datasets. We did this by calling the function `System.nanoTime()` before running the algorithm and after this one was executed. We stored those values, subtracted the final time minus the initial one, and got our measurements, from which we then did an average to obtain a more reliable value. This is what we observed:

Kingdom Exploration

We obtained the following measures by checking in the different datasets how long our algorithm took to find all the points that belong to the same kingdom as the input:

Dataset	Number of POIs	Time taken
XXS	8	0.147 ms
XS	16	0.236 ms
S	32	0.359 ms
M	64	0.671 ms
L	128	1.154 ms
XL	256	2.123 ms
XXL	512	3.96 ms

As you can see in the table above, the time taken to execute the algorithm grows proportionally to the number of points of interest of the dataset. Therefore, if we plot these in a graph, we will get the following:



This graph proves that our function has a cost of O(n), as it increases proportionally to the number of points of view in our dataset. This is because this algorithm traverses the graph by going through all its positions and comparing their kingdom name. Even if we had chosen a different type of traversal such as DFS instead of BFS, we would have still gotten the same output.

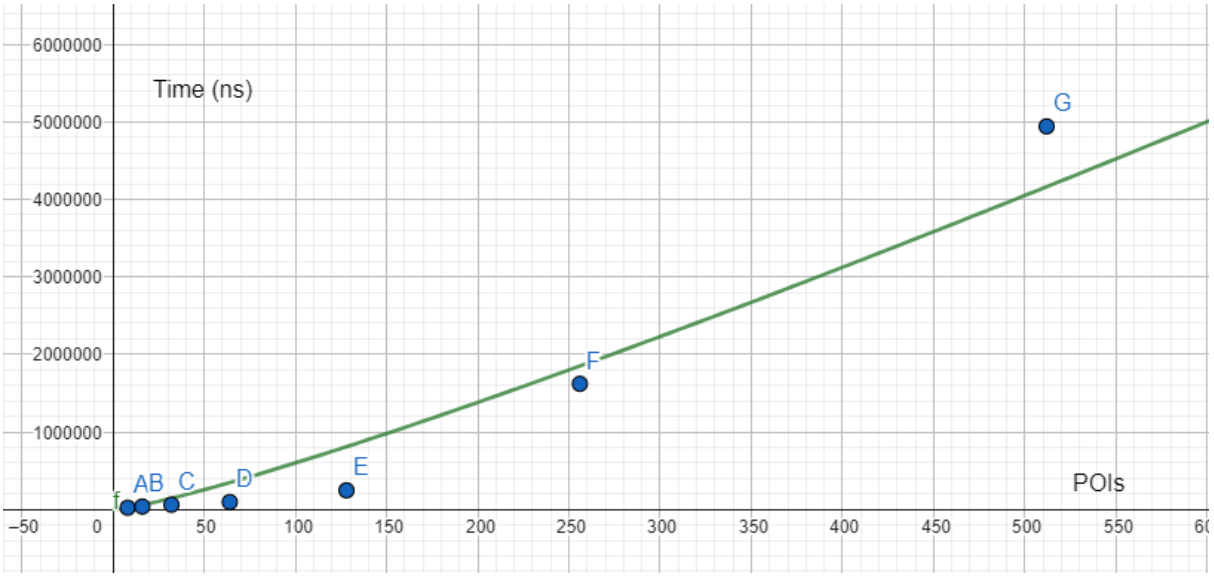
Common Journeys

We've used Prim's algorithm over Kruskal because our datasets hold nodes that are heavily interconnected, as you can see in the table. Prim's algorithm is more efficient whenever the graph is heavily interconnected because its cost increases significantly mainly when the number of vertices increases, thus, the impact of a high number of edges is lower than, for instance, Kruskal's algorithm.

To effectively analyze the cost of the Common Journeys algorithm we've tracked the execution time for each dataset:

Dataset	Number of POIs	Number of edges	Time taken
XXS	8	22	22.10 us
XS	16	80	37.20 us
S	32	374	60.80 us
M	64	1284	96.00 us
L	128	5494	246.00 us
XL	256	24546	1.62 ms
XXL	512	104538	4.94 ms

We can observe that the time increases quite linearly, as expected. Prim's algorithm has an upper bound of $O(|E| + |V| \cdot \log(|V|))$, we can discard the number of edges component as it does not contribute much to the calculation. The equation $|V| \cdot \log(|V|)$ gives us a quite linear plotting as we can see in the following graph:



In this graph we can see that our tracked times follow quite similarly the trace that the upper bound draws. However, we can distinguish that our times follow a trace a bit more curved, having a rather small exponential look. As mentioned previously, we haven't considered the number of edges because its impact is insignificant.

Premium Messaging

When analyzing the cost of the Dijkstra algorithm, it is not like analyzing other algorithms as it is based on finding the shortest path in a graph, the execution time is highly dependent on the number of vertex in the graph, but the average number of edges a vertex has also has a significant impact. This is because in the algorithm, we start at a vertex, we calculate the time for all adjacent vertices which is the same amount as edges that vertex has and then move on to the next vertex until you have finally visited the destination. In a scenario where every vertex is connected to another, meaning that the first iteration the origin will loop through all its adjacent vertices and find the destination in an approximated cost of average number of edges per vertex (m). In another scenario where each vertex only has one edge it where a vertex is an 'o' and an edge is a '-' the graph will look like so:

o-o-o-o-o-o-o-o-o

Finally the time to find the destination will generally be of the number of vertices (n).

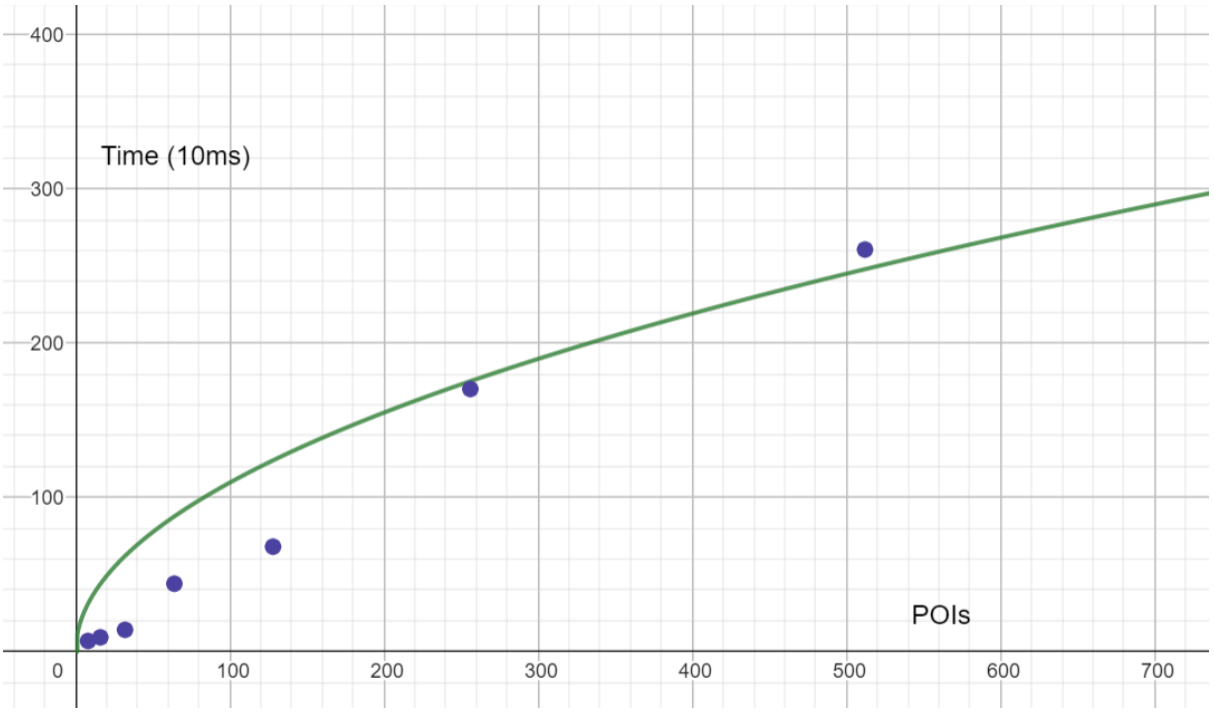
Anywhere in the middle it will be a mix of both, however we don't know in what proportionality until we further analyze it.

Due to the fact that the cost of finding the shortest path greatly depends on how far the two POIs are, apart from the size of the graph, we decided to do the testing in this for two points on two extremes of the graphs, be the dataset XS or XL. This is mainly down to three factors, the most important one is that we want to compare the datasets on a similar basis and

taking points that are proportionally more or less as far from each other in different datasets allows for a fair comparison. The second factor is that when making an algorithm, one of the most important things to consider is how well does it perform in the worst case scenario, and thirdly the further away the points are, the longer it takes for the program to run, and the smaller the compilation time differences from one run to another will be.

It should also be important to mention that we decided that the swallow wouldn't carry a coconut for these tests as this would cause many cases where there wasn't a possible path, leading to more unfair execution times. For each dataset we have run them 5 times and have made an average keeping time of the mil, the results are the following:

Dataset	Number of Vertex	Time taken
XXS	8	0.69 ms
XS	16	0.92 ms
S	32	1.41 ms
M	64	4.40 ms
L	128	6.80 ms
XL	256	17.02 ms
XXL	512	26.06 ms



In this graph the time units are 10ms, and the closest function to satisfy the points was

$$f(x) = a(b\sqrt{cx})$$

In our case the function we chose to portray the points was

$$f(x) = 2(\sqrt{30x})$$

Although this function doesn't accurately represent the theoretical cost since there is no clear root relation between the dataset size and the execution time this could be down to the dataset limitations we have, which will be further explained in observed problems.

Observed Problems

As explained before the anomalous result of the function in Premium messaging could be down to the limitations of the dataset, first of all the theoretical costs are costs that refer to the execution time as the input approaches very big numbers. Our datasets, although some are big, are quite small in comparison to the large numbers these costs are accurate for, being so small the numbers are prone to having a relatively high margin error which might cause for a function that actually describes the relationship to be unrecognizable from the points we have. It is also impossible in this case to duplicate the dataset to get higher numbers as it wouldn't make sense to have POIs with the same id and exactly the same attributes which is extremely unrealistic and would therefore most likely give unrealistic execution times.

Following along the path of the dataset in the premium messaging option, as discussed before the execution time depends highly on the average adjacent vertex each vertex has, in order to truly be able to make a function out of the data, this should be an independent variable together with the number of POIs which is given in the dataset. And then ideal scenario would be for every number of vertex try different averages of adjacent edges, while doing the points as discrete as possible while making a proper average between each point, making execution time be a bivariant function of these two variables, however this would require careful testing and lots of time which is why this is another limitation.

The problem with this part was while implementing the minimum spanning tree. While developing this part, we had some troubles as while we tried to create the MSP we ended up modifying the original graph. We tried to do it in several ways so as to not touch the original graph. But in one way or another due to the references we always keep modifying something in the original graph. So to solve this we made a deep clone of the original graph.

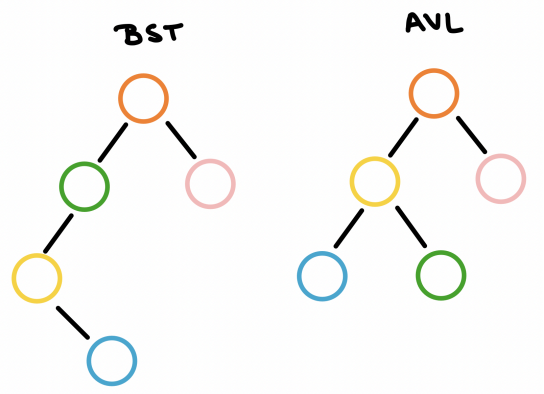
Binary Search Trees (BST)

Structure Design

Similarly to the previous data structure, trees allow us to connect nodes between each other. However, we now define a hierarchy between these, separating nodes in different *levels* (that determine their *height)* and establishing a parent-child relation between them. We define the *degree* of a node as the number of children it has, considering *leaves* those that have a degree of 0. In our case, we will define the maximum degree of a node as two, making our data structure a Binary Tree.

Specifically, we will implement a type of binary tree known as Binary Search Tree (BST). This one defines a specific way of storing information: smaller nodes to the left and greater to the right. This implementation is most useful for optimizing search operations (as we will see next), achieving a logarithmic cost for traversals. This criteria can lead to various *unbalanced* situations that worsen the performance of the data structure. For instance, if several smaller nodes are added consecutively, these will all be placed to the left and our tree would start to resemble a list making our traversing costs increase up to O(n) in the worst case scenario (nodes entered in a sorted manner).

To fix this problem, we have implemented an *AVL Tree*, a particular type of self-balancing BST. This defines the *balance factor* of each node in the tree to detect and fix situations where our structure becomes too unbalanced. The balance factor *(BF)* of a node is determined by the subtraction of heights of its children (considering a non-existing child as BF = -1). If this one is zero it means the tree is completely balanced (same number of nodes on both sides). However, if this one is equal to 1 or -1 it means it has one more node to its right/left. This is already an unbalanced situation, but it cannot be fixed (as there will always be an extra node that doesn't fit), so we will ignore it. Lastly, if a node has a balance factor greater or equal to +-2, this means that the tree can be rearranged to become balanced again. We will face this situation when inserting and deleting nodes in our tree, so we will explain how we fix those on their corresponding sections below.



To represent our tree we considered a sequential implementation (storing nodes in a specific position of an array: starting from parents and adding their children to its right (1st or 2nd cell depending on left/right child)). However, we ended up choosing to create an entity Node that would hold all the required information regarding the residents (id, name, weight, kingdom) as well as three pointers: one to its parent node and the others to its left and right children. We thought this approach represented a tree more clearly and made it easier to restructure the tree when inserting, deleting, and rebalancing; not having to modify the entire array when doing so and improving the cost of these operations.

Lastly, we had to decide which attribute of the residents we wanted to take into account for sorting our tree. As we want to be able to distinguish witches among the population, and we know they can be identified by their weight, we decided to use this as the sorting criteria. Therefore, we would be able to achieve the desired logarithmic costs when traversing our structure using weight as reference.

Implemented Algorithms

Basic Functionalities

The user is able to insert or delete residents from our system. To do so, we followed the process explained below:

Insertion:

We designed this as a recursive function that starts at the root of our tree and keeps exploring the correct path until it finds a leaf to insert the new node. We determine the right path by comparing the weight of the newly inserted node with the current one. If this one is greater, we check if the node has a left child. If it does, we add it as its child by pointing the *left* attribute of the current node to the new node and setting the *parent* of the new node to the current one. However, if there already is a left child, we call the function recursively on it to keep looking for a leaf to insert down the left subtree.

We follow the same steps for when the value of the current node is smaller than the new one, but with the right child. We took this approach as we believed it was a very optimal way of traversing the tree and determining the position of a node by discarding half of the tree in every evaluated node.

Also, we have to keep track of the node's height every time a new node is inserted by calling the *updateHeight* function. Therefore, we recalculate this one on the way up of our recursive function, ensuring we update those nodes that now increased their height because of the new leaf. Once the heights are updated, we also check if the insertion has made the tree unbalanced by calling the *checkBalance* function, which will rearrange the tree if necessary. We will explain this function in the section below.

We also had to consider the possibility of two nodes having the same weight. We had to decide if we wanted them to be inserted on the right or on the left. As we don't believe this affects us in any way (even though we have to be conscious about it when traversing the tree), we opted for inserting them to the left.

Deletion:

Once the user has entered the id of the node that they want to delete, we call the *getNodeByID* function to retrieve it (repeating the process by showing the appropriate error message if no node with that id exists in our tree). This function traverses the tree by checking if the current node (starting at the root) has the same id. If it doesn't, it recursively calls itself on the left and right children until it finds it. This traversal isn't as optimal as the one we did in insertion, because we are doing it by *id* when our residents are sorted by their *weight*. Therefore, there is no way for us to discard half of the tree in each decision and achieve a cost of O(log(n)). Instead, this function has a cost of O(n).

Having found the node we want to remove, we send it as a parameter to the *remove* function. As opposed to insertion, this function doesn't use recursion as we already pass it the node to be removed and don't need to traverse the tree to look for it. It simply disconnects the desired node from the tree and adjusts its parent/children if necessary.

This deletion process required us to account for different cases, as it can't be generalized like insertion. To differentiate between these, the first thing we do in our *remove* function is get the *children* and parent from the node to be removed. Then, we distinguish between two cases: when we want to delete the root or a regular node.

If we want to delete a regular node, we need to identify the following three cases: depending on if it has zero, one, or two children.

Removing a node with no children (leaf) is a simple process, as we just need to disconnect it from its parent (checking if it was its right or left node) and update the parent's height by calling the *updateHeights* function. It is worth mentioning that this function is different from the *updateHeight* we used for insertion. The previous one recalculated the height of a specific node without recursion (as we were already calling it from a recursive function in every node). However, as the *remove* function doesn't use recursion, the *updateHeights* function recalculates the height of a specific node and calls itself recursively on the node's parent (if it has one) to ensure all the corresponding heights get decreased now that we deleted a node. Lastly, once the parent's height has been updated, we check if it is unbalanced by calling the *checkBalance* function, as after performing several tests we saw that this type of deletion could cause situations that needed a RR or LL rotation (further explained on the balancing section).

The second case considers removing a node with one child, in which case we have to connect the child to the node's parent (and the parent to the child). After doing so, we call the *updateHeights* function. This time though, there is no need for us to check if the tree is unbalanced, as we concluded (after several tests) that this situation would never cause an unbalancing worth fixing (never BF >= 2).

The third case involves removing a node which has two children. In this situation, we need to substitute the node for an appropriate one. This could either be the smallest next number or the greatest previous number. We choose to go for the predecessor (greatest previous), which we find with the *getPredecessor* function, called on the node to remove. This one takes the left child of the current node and keeps looping through the right childs until it finds a node that doesn't have a right child (meaning it has reached the node that's farthest to the right in the subtree). We have to account for the case in which the predecessor is not a leaf, as it could have a left child. When this happens, we connect the child to the

predecessor's parent (and the parent to the child). Then, we connect the predecessor to the parent of the node to be removed (and vice versa). At this point, we update the height of the predecessor (and the nodes above this one). Notice that if the predecessor was a leaf, we would only update the heights starting from the position of the node we just deleted. However, if it had a child we would have updated the heights starting from the predecessor's child and upwards.

These are the steps we take when deleting a regular node. However, if we want to delete the root, as we are calling the *remove* function on itself, we have to change some processes. Firstly, we check if the root has a child on its left. If it does, we do the same as we did when a regular node had two children, look for the predecessor and substitute it (also works if the root had a right child). However, if the root doesn't have a left child we need to check if it has a right one. If this is the case, we just set the right child as the new node. If it doesn't (meaning the root is the only node left in the system) we display an error message as we don't want to allow the user to have an empty tree.

Also, we are calling these functions from within the root itself so it is not possible for us to delete it. It is also worth mentioning that to switch content between the root and other nodes we had to use functions such as *copyCode*, *duplicate*, and *switchRoot*; which copy the contents of a given node onto the root (as we cannot change the reference to the root node from where we are running the methods).

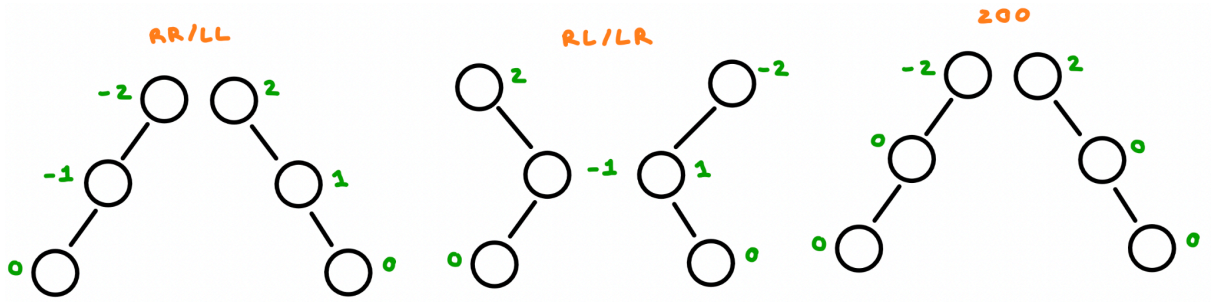
Balancing:

To detect when our tree is unbalanced we use the *checkBalance* function that checks if the balance factor of the node is greater or equal than 2. As we mentioned before, if this one is +-1 we cannot fix the unbalancing, as there will always be an extra node to the side. However, once the balance factor exceeds two, this means we can rearrange our tree to make it balanced again.

Firstly, we identify if the tree is unbalanced on its right or left side by checking the sign of its balance factor. If this one is positive, it means we need to balance the right subtree of the node, so we call the *balance* function on it. On the contrary, if its balance factor is negative, we call the *balance* function on its left subtree.

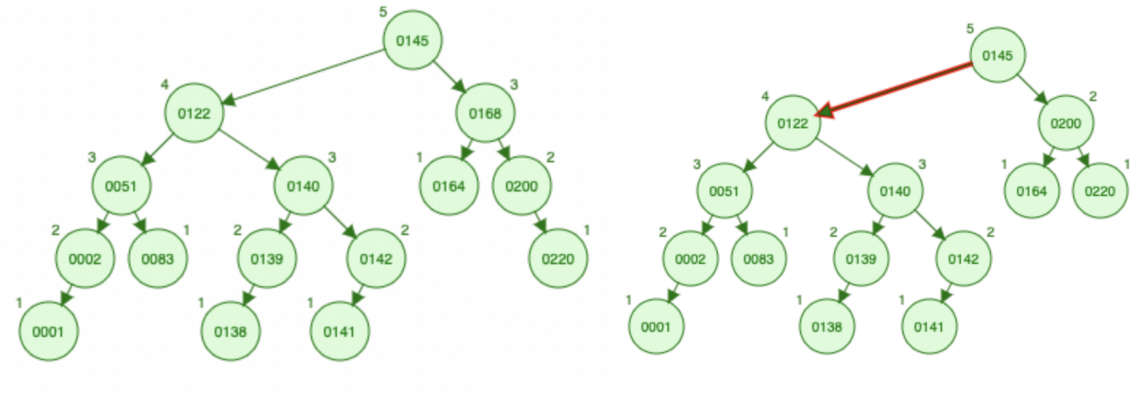
Moreover, we realized that in some deletion cases the fix of balancing on a node could still not fix the balance factor of its parent (in insertion as we use recursion we make sure all the nodes are balanced on the way up), so we had to account for this case in the *checkBalance* function.

We reach the *balance* function when we know for sure that the node is unbalanced. At this point, we need to identify the type of unbalancing the node has, to fix it accordingly. We have accounted for the following three types:

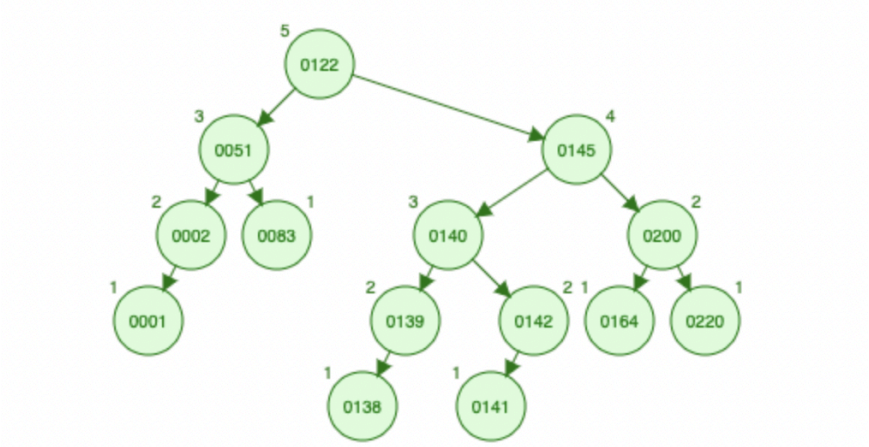


The first case is when we encounter a right-right or left-left rotation, meaning that the balance factor of the unbalanced node is caused by an accumulative unbalance of its child with a balance factor of +-1. We solve this situation by performing a left rotation on the node with a balance factor of +-2. Secondly, we can find a right-left or left-right case in which the balance factor of the unbalanced node is of the opposite sign as its child. This case is solved by performing a right rotation on the node with a balance factor of +-1. Lastly, after doing exhaustive tests on the deletion process we figured out this last case could happen when deleting a node and fixing its unbalanced situation (insertion will only cause RR or RL situations). In this one, the right/left subtrees of the unbalanced node are balanced, but within different heights that cause the node to have a balance factor of +-2.

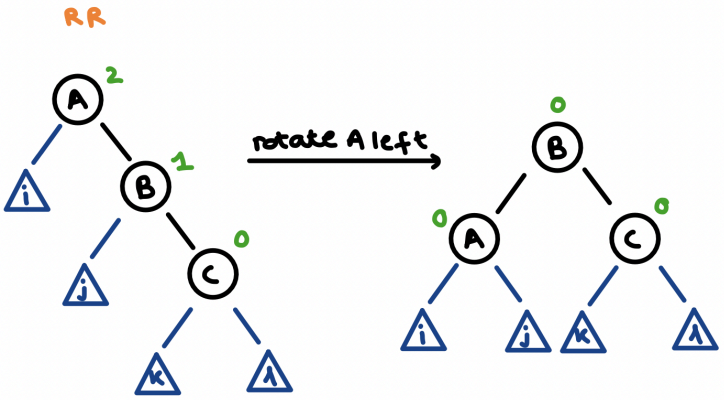
Here's an example of this particular scenario, in which if we try to delete the node 168, we will encounter a balance factor of -2 on the root (as the height of the right subtree is 1 and the left one is 3), but its children will be balanced.



We fix this situation by rotating the unbalanced node left and achieving the following balance tree:



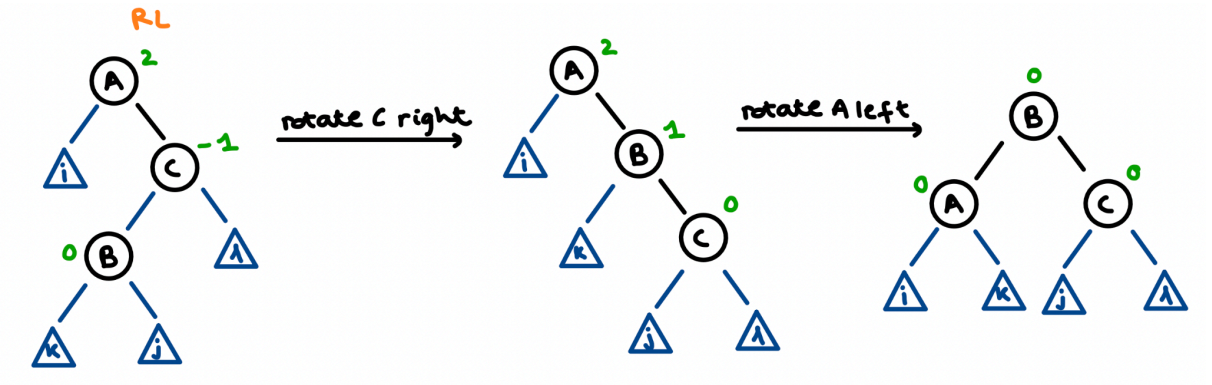
To perform the left rotation we mentioned before, we will use the mechanism specified below. This one consists in making the node with the balance factor equal to 1 the new middle node that connects both the unbalanced node and the one with BF = 0. To do so, we need to connect the parent of the unbalanced node (if it has one) to its right child (B), and vice versa. Then, connect the unbalanced node to its right child as if it was its parent and connect itself as the new left node of its right node. Then, if the previous right child (B) had a left child we substituted, we need to connect this as the right child of the unbalanced node (A), as we know for sure this one will be greater than A because of its position in the tree. Same process but mirrored for when we have a LL situation.



When performing any rotation, we must ensure we update the heights of the nodes to validate the balancing has been done correctly. Also, we need to account for the case of rotating the root. When this happens, as we cannot simply change the pointers of the nodes (as we are performing the operation on the root node) we use the function *copyContent* and *switchRoot* to change the contents of this one with the desired node.

On the other hand, to perform a right rotation we will need to connect the left child (B) of the node we are rotating (C) to its parent (A), and vice versa. Then, we need to connect C as the new right child of B (and set B as the new parent of C), and if this one had a right child,

we need to connect it as the left child of C (knowing this one is for sure smaller than C so the tree will be consistent). Once we have done this, we are in the exact same situation as before, in which we need to perform a left rotation on the top node to fix the unbalancing of the tree. We perform the exact same process but mirrored for the LR case.



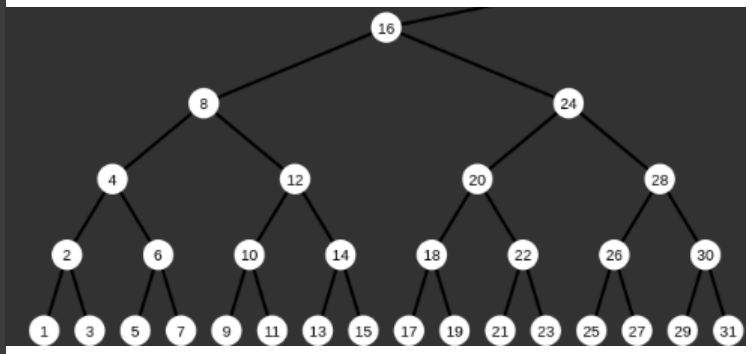
For RL or LR rotations, there is no need for us to account for rotating the root, as this will never happen (notice how we rotate C and B, node A stays the same). If this type of unbalancing situation required us to rotate the root (A), we would do it when rotating left, not right (as we call the same function used for the previous rotation).

Visual Representation

We are asked to develop a way to visualize the graph, either following the format presented or in a 2D graphical representation.

```
Which functionality doth thee want to run? C

     |--- Rabbit of Caerbannog (666, Kingdom of Aaargh): 1.5kg
     |
|--- Dingo (736, Kingdom of Anthrax): 43.85kg
|    |
|    |--- Brother Maynard (7, Kingdom of Antioch): 49.9kg
|
* Tim the Enchanter (1, Kingdom of Aaargh): 69.42kg
|
|--- The Old Man from Scene 24 (24, Kingdom of Mercia): 72.4kg
     |
     |    |--- Roger the Shrubber (42, Kingdom of Nni): 75.19kg
     |    |    |
     |    |    |--- A Famous Historian (1, Kingdom of UK): 77.7kg
     |    |
     |--- King Arthur (5, Kingdom of Mercia): 78.1kg
```



Idea of a 2D graphical representation.

Presented format to represent a tree.

As the whole project is a console program (where all the interaction happens in the console). We found it better to develop the visual representation functionality by following the presented format, which shows in the console the tree following the format specified before, rather than using swing and creating a window that graphically shows the tree instead of showing it through the console.

Looking at the specified format, it can be seen how the tree is printed from right to left, being the rightmost node in the first line and the leftmost node in the last line. Therefore, to represent it that way, a reverse inorder traversal is needed. Instead of traversing left subtree, root and right subtree, it traverses right subtree, root and left subtree. So the structure of the implemented function will be the following: an if to check if the node is not null. If it is not null, then the function is called recursively passing as a parameter the right child of the current node. Then the current node information is printed and, finally, the function is called recursively passing now as a parameter the left child. With that it will traverse the whole tree and print it in the order we want(from right subtree to root to left subtree).

Once the whole tree is printed, now, it has to be printed with the correct format. To do so, a new block of code is added to the code after checking that the current node is not null and before calling the function recursively for the right child. Where the proper verifications will be done to add the spaces/columns or remove spaces/columns, needed to follow the specified format, in a string that will later be passed as a parameter when recursively calling for the function(to accumulate the correct spacing). Firstly, it will be checked if the node is the root or not, if it is, then no verifications will be done, as nothing needs to be printed.

But, if it is, the proper spaces/column will be added or removed depending on whether: the node is the child of the root or not, the node is in the right or the left subtree, the node's parent is a right or left child, the node is the right or left child. Note that to know if the node is the right or left child a boolean is passed when the function is called, storing true if it is the left child or false if it is not. And to know if a node is in the left or right subtree, when the root is printed a boolean that is initialized to false at the start of the program is set to true. Finally, to know whether the parent is the left child or not, it is manually checked if the left child of the parent of the parent of the current node is equal to the parent of the current node. Therefore, storing *TRUE* if the parent is a left child and false if it is not.

Finally, as between one node and another it needs to be a space and some columns or not depending on the case that the node to be printed is. Before printing the node with its correct spacing and format before it, a previous print will be made with the line that separates the node to be printed and the node previously printed. To do so, some minor checks are made before printing. As the line that is printed before the node does not need to have the exact same spacing and format that the node needs.

Scientific Identification

This functionality will implement search logic. It will prompt the user for a weight and a category, the search it will perform will depend on the category chosen. The first search type will display those nodes with the selected weight, the second type will display the first node that weighs less than the input and, finally, the last search will show the first node that weighs more.

When the search has to find the nodes with a particular weight, our algorithm will start at the root and will recursively check whether the weight to find is higher or lower than the current node's weight. If the current weight is higher than the weight to find, we will continue to the left, that is, we will continue traversing down the left node as this subtree contains values lower than the current. If the current weight is, on the other hand, lower than the weight to find, the traversion will continue towards the right, as the right subtree has values larger than the parent node. When a node with the weight we are searching is reached, we will consider it and we will continue searching as our BST can have duplicate values, we will go traverse both sides because, if we want to maintain the tree balanced, we cannot simply leave the duplicated nodes next to each other to the same side, thus, because we balance our tree, we won't know where duplicate values can be. We will continue searching until we reach a leaf node.

Now, for the next search, as we understood the statement, we must traverse the tree until we find a node with a weight lower than the specified weight. In order to do this, we start at the root and we compare in each iteration, if the current node's weight is less than the specified weight, if so, we've found our node, otherwise we must continue down to the left as this subtree will always have weights lower than the current node.

For the last search, we must do the same but going down the right side and comparing if the current node's weight is greater than the specified weight. However, we realized there was a slight mistake and the last two searches should have the getter method to get the left/right node from the current node swapped.

Another way of understanding the statement for these two last searches is to find the first weight considering all weights available, without just stopping when the condition, is lower than or greater than, is satisfied. To do this, as an extension to the logic mentioned above, after finding a node that satisfies our condition we must continue towards the subtree opposite to the side we previously constantly traversed, as there can be a node that has a weight closer to the specified weight, representing the first node that would go before or after the specified weight, if the weights were sorted linearly.

Witch Hunt

This functionality allows the user to identify all the residents whose weight is in between a given range, which can be extra useful for knights who are planning to organize special raids for finding witches.

Once we have prompted the user to enter the desired minimum and maximum weight, we call the recursive function *getNodesInRange* from our root. This one checks whether the current node is in between the range. If it is, we add it to a custom list of witches passed as a parameter. Then, we check if its weight is greater than the minimum required (meaning that its left subtree could contain other nodes that are in the desired range) and call the same function on its left child (if it has one). Conversely, if the current node's weight is smaller than the maximum defined, we call the function on its right child (if it has one) to find if there are any nodes with a greater weight than the current one but smaller or equal than the maximum.

Therefore, we recursively traverse our tree by starting at the root and going down through the leaves, discarding those subtrees we already know fall outside of our boundaries to optimize the performance of our function. This approach follows a preorder traversal, as the order in which the nodes will be inserted to our list is first the parent and then the preorder of its children.

This is the perfect example of a type of search that is very optimal in this tree structure, as we are able to search for values in a range while only traversing the parts of the tree that hold potential solutions as a result of the sorting criteria used for inserting new nodes in BST.

Result Analysis

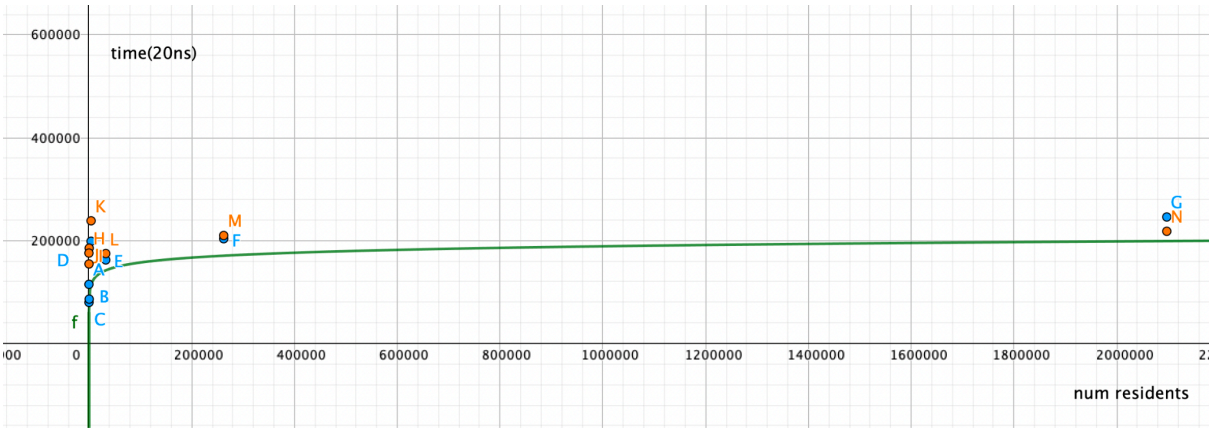
To analyze the different algorithms and see their performance when running in the different datasets, we took the same approach as we previously mentioned: calculating an average of the time they took to execute on each case. We observed the following:

Basic Functionalities

To properly measure the cost of our ***insertion*** algorithm, we checked the time that this one took to insert a new resident in all the datasets both for when the tree auto-balances itself and when it doesn't. These are the results we obtained:

Dataset	Number of residents	Time taken (no balance)	Time taken (*balance*)
XXS	8	5.75 us	8.79 us
XS	64	3.99 us	7.73 us
S	512	4.31 us	9.27 us
M	4096	9.94 us	11.93 us
L	32764	8.12 us	8.74 us
XL	262144	10.18 us	10.5 us
XXL	2097152	12.3 us	10.91 us

As you can see, this algorithm increases as the input does (but not significantly). As we mentioned in the explanation of insertion, we are traversing the tree to find the position where to place our element, meaning we should somehow get close to a O(long(n)) cost, as we will be discarding half the tree in each decision. If we plot points above into a graph we get the following, being orange the points for when the tree balances itself and blue for when it doesn't.



Notice how both achieve a similar logarithmic cost, increasing as the input does but not as fast. The reason why these inputs are similar is because they counteract each other. That is, whenever our tree is not balanced, traversing through it gets less efficient, as we are no longer ensuring we discard half of the tree in each decision (as we are allowing nonoptimal shapes), which increases the cost of inserting an element. On the other hand, the balanced

version of our tree is much more efficient to traverse through and can find the position to insert the node much faster. However, the cost of balancing the tree is quite high and contrasts the optimal traversal, making both algorithms have a similar cost.

Moreover, the order in which the residents are structured in the datasets also affects our algorithm when our tree isn't balanced (blue case). This is, the less optimal the insertion process is, if no balancing happens, the less optimal traversing through the tree would be afterwards. As the worst case scenario, if we were given a sorted dataset all the nodes would be structured resembling an array (all to one side) and when the user tried to insert a new node (smaller for when sorted upwards or larger for sorted downwards) we would achieve a cost of O(n) as we would have to visit all the nodes of our tree to insert it. This isn't the case for our datasets, although they aren't placed in the best optimal way for searches. Hene, increasing the cost of our blue algorithm.

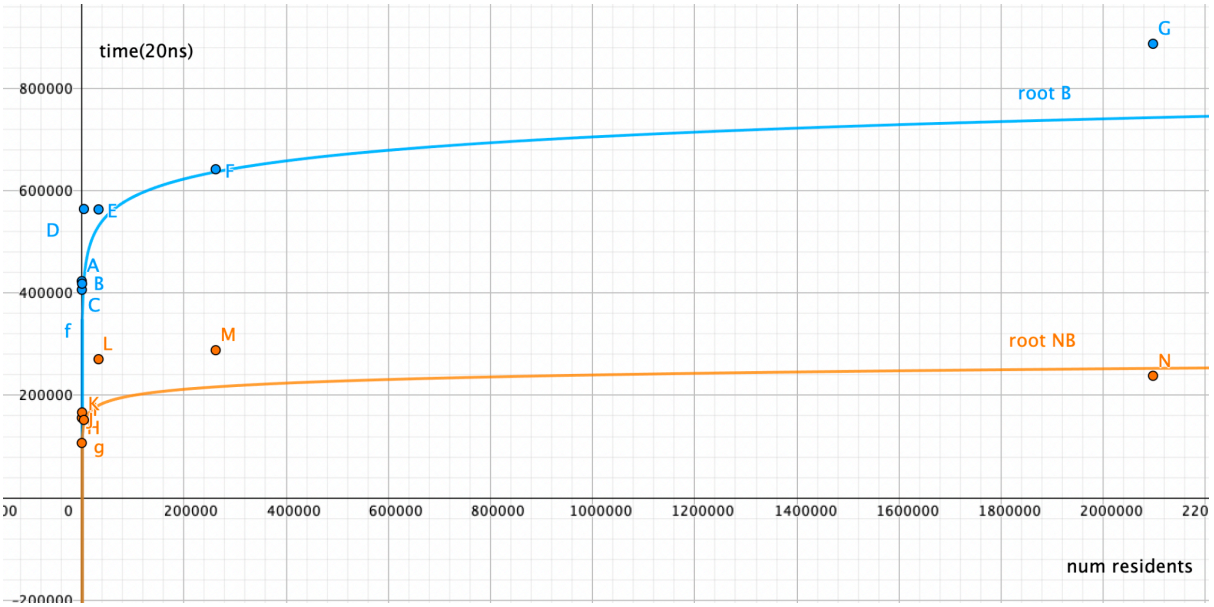
We would only be able to achieve a perfect O(log(n)) cost when inserting an element in an already balanced tree without needing to balance it. This case is more likely to happen in the orange algorithm, as none of the datasets provided give a perfectly balanced tree for the blue case to be able to insert a node with the least cost.

We will now evaluate our ***deletion*** algorithm by how much time it takes to delete the root and a leaf in each dataset. We will compare these two cases as they are usually the ones with higher and lowest cost. This is what we observed:

Dataset	Number of residents	Time taken (*root NB*)	Time taken (*root B*)
XXS	8	5.36 us	21.17 us
XS	64	7.81 us	20.32 us
S	512	8.34 us	20.92 us
M	4096	7.61 us	28.21 us
L	32764	13.54 us	28.17 us
XL	262144	14.42 us	32.09 us
XXL	2097152	11.92 us	44.35 us

We can see in the table that it takes significantly less time to delete the root of our tree whenever this one is not balanced (almost half of the time). We believe this is due to the fact that the cost of checking for unbalanced nodes and fixing them is very big, and this will happen a lot when removing the root of the tree. This cost compensates the optimization of the traversal we must do to find the predecessor node we need to substitute the root for.

If we plot these points in a graph we get the following:



As we can see in the graph, both cases give us more or less a logarithmic cost because of the tree traversal we do to find the predecessor node. The difference between this is due to the extra cost of balancing whenever is necessary. The deviation in some of the non-balanced datasets may be due to the non-optimal shape of the tree this one acquires by not balancing it when necessary, making it harder to reach the desired predecessor node in some cases. The balanced version of this algorithm also scales quicklier as the datasets grow and deviates from the O(log(n)) cost because of the two traversals required for this task. As the algorithm not only looks for the predecessor node, but also updates the heights of all the nodes above this one when it is removed. This process takes more time as the tree grows, making it harder to reach a logarithmic cost.

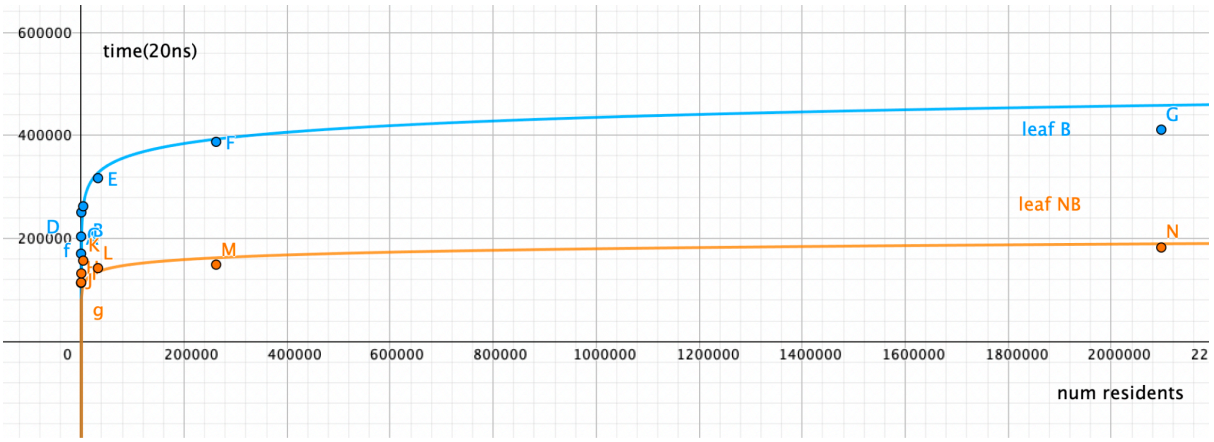
Note that this process will be very similar when deleting a node that has two children, as the algorithm will still have to traverse the tree to find the predecessor node of this one (even though it would take less time as it won't be as high as the root and will find the predecessor in less steps). However, this process will change a bit whenever there is no need to traverse the tree to find a node: deleting a leaf node (if you recall, our *deletion* function gets the node to remove as a parameter and doesn't need to find it in the tree). However, we will still need to update the heights of all the nodes above the deleted one.

This are the measurements that we took for this last case of deleting a leaf node:

Dataset	Number of residents	Time taken (leaf NB)	Time taken (leaf B)
XXS	8	5.69 us	8.53 us
XS	64	5.73 us	10.19 us
S	512	6.61 us	12.53 us
M	4096	7.85 us	13.12 us
L	32764	7.12 us	15.86 us
XL	262144	7.46 us	19.38 us
XXL	2097152	9.13 us	20.57 us

As you can observe, the time taken for this is significantly smaller than the one to delete the root in both the balanced and not balanced cases. This is because the way to delete this type of node is way simpler by just connecting the parent of the deleted node to its children and there is no need to traverse the tree to find the predecessor. Moreover, this deletion tends to cause less unbalancing situations than deleting the root, taking less time to perform the balanced version of the algorithm.

If we plot these points in a graph we get the following:



Notice how both functions give us a logarithmic cost. This is because of the *updateHeights()* function we call in this method. We don't traverse the tree to find a node, but we do the inverse process for updating the heights of those nodes above the deleted one. Therefore, achieving the same logarithmic cost. The difference between these two cases (balancing and not balancing) is due to the extra cost of balancing the node's parent when required.

Overall, we can see how balancing our tree gives us a way bigger cost in both insertion and deletion cases. However, this increase of cost is the price we pay for achieving very optimal ways of traversing our tree, as you will see below.

Visual Representation

The following table shows the time taken to graphically show the tree unbalanced and balanced:

Dataset	Number of residents	Time taken (b-*trees*)	Time taken (*b-trees balanced*)
XXS	8	11.6 ms	12.1 ms
XS	64	13.8 ms	13.2 ms
S	512	25.6 ms	28 ms
M	4096	61.3 ms	53 ms
L	32768	286.7 ms	239 ms
XL	262144	1632 ms	1429 ms
XXL	2097152	10652 ms	9525 ms

As it can be seen it takes the same time to graphically show the b-tree when the tree is balanced or unbalanced. It does not matter if it is balanced or not because to show the whole tree, the program has to traverse it all, from the rightmost node to the left most node. Therefore, as balancing a tree does not add or remove nodes, the number of residents stays the same. Therefore, the time to traverse it is the same.

Moreover, compared to other functionalities, this one takes way more than the other. That is not only because compared to others, this functionality traverses the whole tree always. But it also does different calculations for each node to display the tree well.

Scientific Identification

The execution timings we obtained for each search with the tree unbalanced are as follows:

Dataset	Number of residents	Equal Weight	First Less Weight	First Last Weight
XXS	8	13.8 us	2.5 us	2.3 us
XS	64	16.2 us	3.0 us	2.0 us
S	512	15.2 us	2.6 us	2.5 us
M	4096	16.9 us	2.1 us	3.1 us
L	32768	19.7 us	3.2 us	2.4 us
XL	262144	19.0 us	2.6 us	2.1 us
XXL	2097152	26.4 us	2.9 us	3.3 us

On the other hand, with the tree balanced, we get the following times:

Dataset	Number of residents	Equal Weight	First Less Weight	First Last Weight
XXS	8	11.8 us	2.8 us	2.9 us
XS	64	14.2 us	2.5 us	2.4 us
S	512	14.3 us	3.1 us	3.1 us
M	4096	15.2 us	3.2 us	2.5 us
L	32768	19.1 us	2.8 us	2.4 us
XL	262144	18.6 us	3.7 us	3.6 us
XXL	2097152	24.1 us	3.1 us	2.9 us

As we can clearly observe, only the equal weight search increases with a dataset change to more data. The other searches seem to maintain a linear trend whereas the equal weight search depicts a shape slightly similar to a logarithmic function.

The cost we expect for all searches is logarithmic, we can appreciate this only for one search. The other searches seem to perform better than expected as they maintain a rather small cost linearly. The upper bound is n for all searches, as would happen if the elements were inserted in a sorted manner without balancing. In our case, elements are added randomly, thus, we don't see that upper bound in our testing.

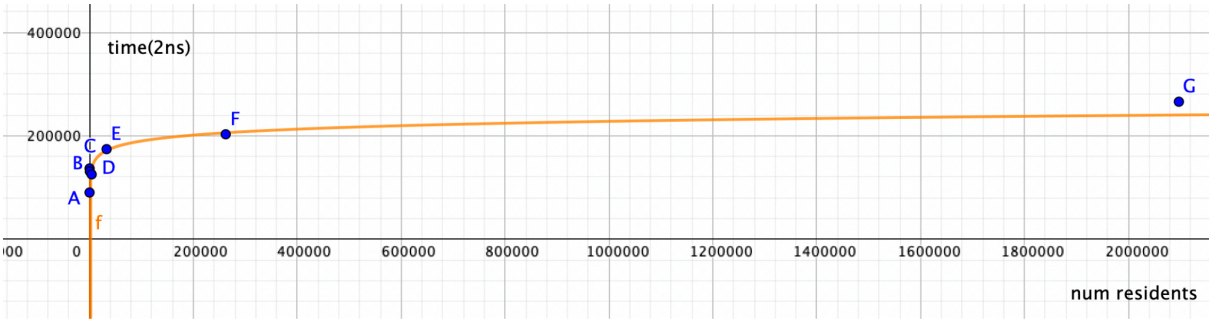
When it comes to balancing, we see no appreciable difference between the tests executed in a balanced tree and those executed in an unbalanced tree.

Witch Hunt

We have decided to analyze the cost of this search both when our tree balances itself and when it doesn't to compare the behavior of our algorithm in this case. These are the results we obtained:

Dataset	Number of residents	Time taken (*no balance*)	Time taken (*balancing*)
XXS	8	7.57 us	4.41 us
XS	64	13.56 us	6.68 us
S	512	37.3 us	6.40 us
M	4096	98.1 us	6.118 us
L	32764	7.78 ms	8.5 us
XL	262144	134.67 ms	9.9 us
XXL	2097152	7.45 s	12.97 us

Firstly, as you can see, the time taken to execute the code grows as the input does, but not as quickly; achieving a O(log(n)) cost. If we plot the points obtained when our tree balances itself we obtain the following graph in which we can see how our function behaves logarithmically.



As we mentioned in the explanation of this function, we are able to achieve this logarithmic cost by the way in which we traverse the tree, discarding almost half of it in each decision. Notice that the cost of this type of search depends on the input given, reaching a more logarithmic cost for smaller and more precise ranges (some that retrieve very few residents, such as the examples we used to obtain these points). However, if we keep on increasing this range the cost will get bigger, reaching O(n) on its worst case scenario: whenever the input range is the minimum and maximum resident's weight of our tree. As in this case, the algorithm won't be able to discard any branch and will have to visit all their nodes.

On the other hand, when we deactivate the balancing of our tree our algorithm increases its cost significantly. This is because we are allowing our tree to have a not optimal arrangement of branches that makes it harder to traverse through it, discarding branches that may be empty or way smaller than they should. Therefore, we visit a way bigger number of nodes than we do when the tree is balanced. This makes its cost increase significantly, especially when the datasets are not provided in the best order.

Moreover, the behavior of the algorithm is much more unpredictable when the tree isn't balanced. As you can see in the table above, we get very big deviations between some datasets, even decreasing their cost for bigger datasets. This is because its performance completely depends on the datasets and the specific order in which the residents are provided to us, which will affect in the shape of three and therefore in the number of residents that the algorithm has to go through.

Observed Problems

When we delivered the code of the project, we had forgotten to add to it one of the cases we mentioned before in the deletion process when removing the root. We had wrongly assumed that there would always be at least three elements in our tree (hence the root would have two children) and when this one was deleted, we changed it by its predecessor. We did account for all the different cases when the node to delete wasn't the root (0,1, or 2 children). However, we forgot this could also happen in the root if the user started deleting every resident in the dataset and eventually ended up with three or less.

To fix this, we simply added an if condition that checked how many children the root had. If these were greater than two, we executed the same code. If not, we either substitute it by its child (if it had only one) or prompt an error message to the user when the root has no children (as we cannot remove the object we are running the function to).

Moreover, when we realized the above-mentioned issue, we revised the code thoroughly and found we had forgotten to update the *name* and *kingdom* attributes whenever we switched nodes with the root. That is, in the *switchRoot*, *copyNode* and *duplicate* functions. We had overlooked this mistake as before delivering the project we ran several tests on our internal *printDebug* function which displayed the *id* and *weight* of a node (which are the important attributes that really tell us if a node has been rotated or not) and we saw the code was working as expected. Again, this only happens whenever we delete or rotate the root, for the rest of the nodes this isn't a problem as we didn't have to copy the content from one node to another, we were able to simply change the pointers between them.

Lastly, in our *Scientific Identification* function, for the second search (*WOOD*) we made a mistake and specified in the *else* condition that the *currentNode* we wanted to move to was the one to the right of the one we were at. However, as we are looking for smaller nodes, we would have had to choose *left* instead. The same confusion happened for the third search (STONE), in which we choose *left* when we would have had to take *right* to find larger values.

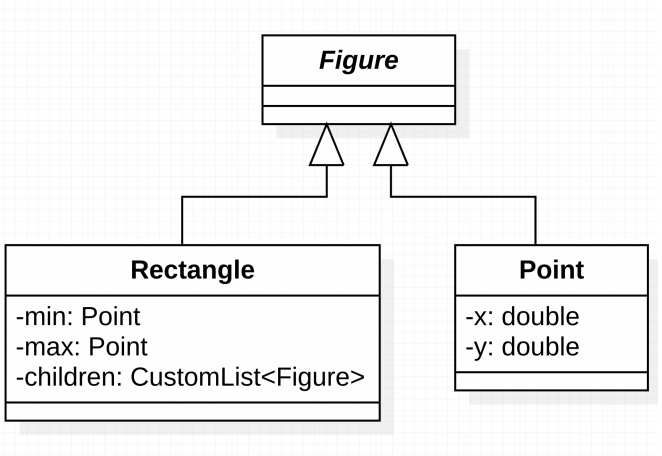
R-trees

Structure Design

This kind of tree is most used for representing spatial data, perfect for storing the coordinates of shrubs for optimizing their search. Hence, we will use a two dimensional tree for performing this task and be able to execute proximity operations efficiently.

R-trees group the given points in rectangles to organize the information and be able to find it easily. We define the order of the tree (O) as the maximum children that can be stored in a rectangle, in our case, we choose this to be three. By trying to keep the rectangles as small as possible while still containing the points, we will achieve the best performance of the data structure.

Therefore, we identify two types of nodes: internal (rectangles that store other up to O nodes inside them) or external (leaves, storing up to O points inside of them). To represent these, we decided to create the entities Rectangle and Point, respectively; which both extend from the abstract class Figure. We chose this approach as we believed defining some common methods for both classes on its parent would come in handy for the implementation of these (such as *getDistanceTo,* which can be implemented in the same way for both entities). Each of these entities then overrides the methods we define in Figure and implements them accordingly.



We defined a Rectangle to have a minimum and maximum points to keep track of the area it contains (reusing the entity Point for these attributes), as well as a custom list of Figures to represent its children, as these can be either other rectangles or points. On the other hand, we defined a Point to have x and y coordinates (representing the longitude and latitude of the point). Having these defined, we then implemented methods such as *inRange*, *getCenter*, or *getPerimeter*; which will explain below with more detail.

Implemented Algorithms

Basic Functionalities

The addition first checks if the root has no children, if this is true it just adds the point to the root's children, if this is not the case, the children could perfectly be rectangles, so we check if the children of that rectangle, in this case the root, are rectangles if they are rectangles, we loop through the rectangles children trying to see which rectangle would expand the least, in this case we decide to calculate this by doing the perimeter the new rectangle would have if this point was introduced, every time we find a rectangle that will expand less we save the rectangle and when the loop is finished we have found the rectangle's (in this case the root) child rectangle which has to expand the least to include that point, in the case where multiple children are drawn, we chose the first one as this will have the same impact on them. The reason why we chose the perimeter is that the area describes less accurately if a rectangle has to expand more than another, for example if a rectangle's x coordinates are all the same and you add a point which is extremely far but has the same x coordinates, the rectangle's area will remain 0 which is not useful for us.
Once we have selected the child rectangle which will have to expand the least, we expand the rectangle to what we calculated (which means changing its max and min point, which are two points that represent the smallest x and y coordinates in the rectangle and the largest x and y coordinates in the rectangle) and we run the add function once again, but this time with the rectangle chosen as the root, and we will repeat the same process over and over again until we have run the function on a rectangle whose children are points, or shrubs in this project, and we simply just add the point.
Once we have added the point, if the rectangle we have added a point to's size is not bigger than the max children allowed for a rectangle, the stack will clear up from all the recursive calls and nothing else will happen. In the other case we will go through all of the function calls reordering through all the rectangles we have added to. By reordering, we mean popping all the children from the overflowed rectangle and keeping them in our custom list. The parent of the rectangle we have just popped from will have two rectangles added to its child list for the one it has just lost, the popped points will be divided into the two rectangles. The way we do this is by putting the two furthest figures (rectangle or point) into a respective new rectangle, then we will add the rest of the figures into the rectangles by checking which will make the rectangle expand the least like we did previously and when adding the figures we expand these rectangles. Now that we have added two rectangles where there were only one we have to check if these two rectangles are children of a rectangle that has just overflowed, and if this is the case we repeat the process. We do this process recursively utilizing the recursive calls we made before when trying to find a rectangle to add a point to. This process however can lead to overflowing the root's children list, which is why before starting all the addition process we created a rectangle and added

the root to its children meaning that if it overflows the roots children can be divided into two rectangles in the new parent's children and then we will return that parent as the new root.

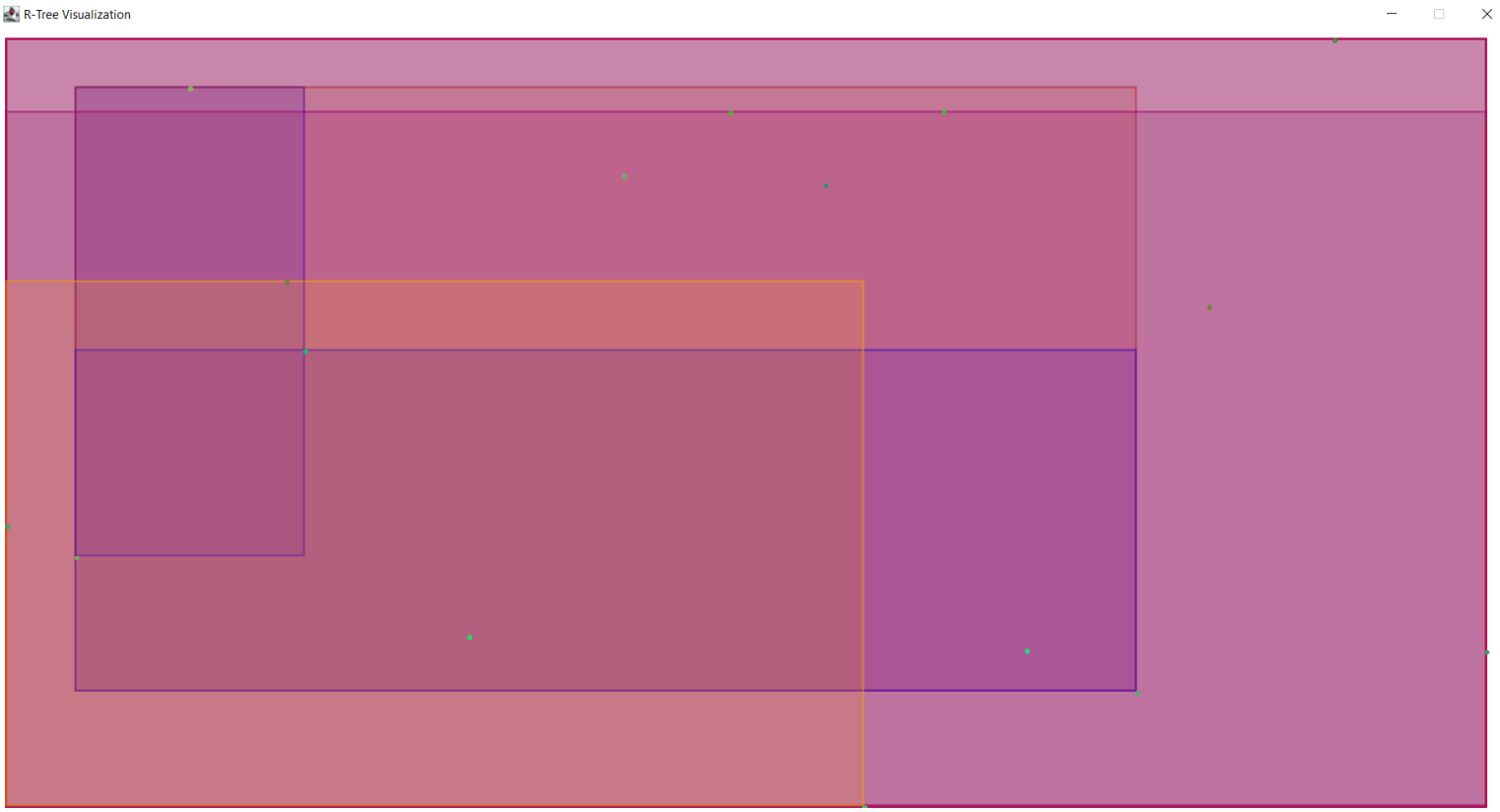
The deletion functionality allows the removal of given points by traversing the R-tree in an efficient manner in search of the particular point to remove and performing post-deletion processing to leave the R-tree structure in a way that will enable further operations to execute efficiently. These processing techniques involve: resizing the deleted point's parent rectangle, if possible, to the minimum possible size to follow the rule of having the rectangles as small as possible to avoid overlapping as much as possible, and ensuring the parent rectangle still has a predefined minimum number of children, to avoid having unnecessary rectangles.

The algorithm works recursively entering rectangles where the point to delete could be inside, this is checked by comparing the point's coordinates to the rectangle's boundaries, defined as minimum and maximum points, which are placed the farthest as possible from each other. If the algorithm finds a point it simply compares its coordinates with the point to delete. This search is done until the point is found or all rectangles where the point could be inside have been recursively searched with no match.

If the point is found, it's removed from its parent rectangle and the R-tree update process starts. The update process executes also recursively, it gets the parent rectangle's children and checks if there is a minimum number of children, 30% of the maximum number of children, by default. If there are less, the parent rectangle is removed and the points are reinserted from the root. A recursive call is done to ensure the parent rectangle of the deleted rectangle still meets the minimum requirement. The recursion will continue until the requirement is met for all traversed rectangles. It is then when the rectangle resizing occurs, ensuring that the last rectangle is the smallest as possible considering its children.

Visualization

The visualization functionality allows the user to graphically see the R-tree in the following way:



To display it like that java swing has been used. The process is pretty simple, given a rectangle and the 2 points that define it, we calculate the width and the height and use the left top point as the start x and y of the rectangle and we draw it with the draw function. For the points we just take the coordinate of the point and draw a doot there.

Knowing that, now the process of printing a whole R-tree is pretty straight forward. A queue is created and the root is added. After that a loop that will stop when the queue is empty will start. This loop pops the first element of the queue and draws it into our frame, if it is a doot it draws a dot if not a rectangle. After drawing it, if the element has children, meaning that it is not a dot, it will add those childrens to the queue and start again popping the next element of the queue. If the element is a dot it will just return to the start and pop the next element without adding the childrens as it does not have.

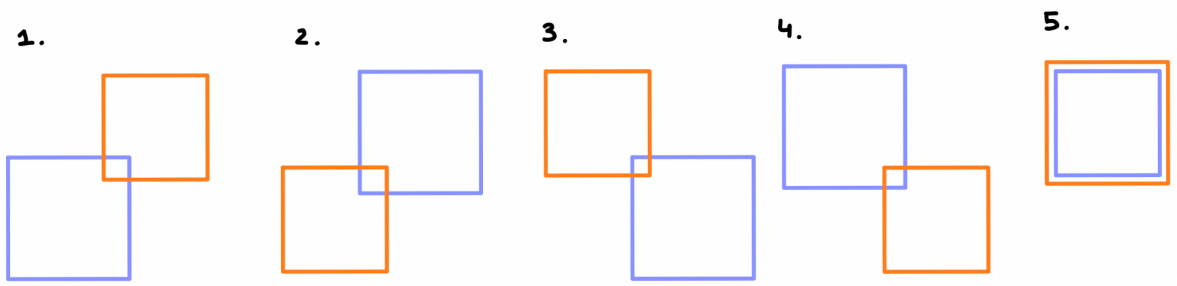
That way we will traverse the whole R-tree in a breadth first search, but a breadth first search from start to end, we are not locking for a node, we are just locking to traverse the whole r-tree.

Search by Area

This functionality allows the user to find all the shrubs contained in a specific area determined by two points. These two coordinates could also be considered the maximum and minimum points of a rectangle that represents the area we want to search in. Therefore, we will traverse our tree and find which rectangles overlap with the given one and the points that could fit in its boundaries.

Once we have prompted the user to enter the set of points, we will start traversing the tree from its root. For each of the nodes we visit, we will go through their children and check if they are in the given range. We should keep in mind that the figures we encounter in our tree can be either a rectangle or a point, so we will have to treat them differently. Hence, we defined the *inRange()* function on our Figure class, which is overridden by both the Rectangle and Point entities.

To check whether a rectangle fits in the given range, we need to consider any overlaps between these two, as the rectangle could contain points throughout its whole area. Therefore, if we discarded any overlap (even if it was just a corner) we could potentially be losing points that would be of our interest. To ensure this situation doesn't happen, we defined the following five possible overlap cases (where the orange rectangle represents the one entered by the user and the purple the one we are comparing it to):



The first four cases define an overlap with one of the sides of the rectangles. We had to separate these cases as we only have the minimum and maximum points of each, and the comparisons vary depending on the side they are (top/bottom and left/right). We defined thee to also include the greatest and smallest overlaps. Meaning that if the top right corner of the purple rectangle and the bottom left corner of the orange one meet in a single point, the first case will be triggered. However, if they are fully overlapping this case will also be triggered, as it is still true that the bottom left of the orange triangle is inside the purple one. Therefore, all of the four first conditions would be true in a full overlap or if the orange rectangle is inside the purple one.

Nonetheless, we still had to consider the case of the evaluated rectangle being contained inside the area we want to search in. This case wouldn't trigger any of the others as none of the corners of the orange rectangle meet the purple one. That is why we defined the fifth case, in which we check that all the latitude and longitude of the rectangle evaluated are greater than the minimum and less than the maximum of the given points. It is worth mentioning that we considered the possibility of the first input point being greater than the second one, so we do our comparisons using the *max()* and *min()* functions from the *Math* library to ensure we get the right measures.

If we find that the rectangle we are checking satisfies any of the cases specified above, we know it could potentially contain the points we are looking for so we recursively call the function on itself to keep traversing the tree down its path. Conversely, if we find that the rectangle doesn't satisfy any of the conditions, we ignore it and don't evaluate its children, as we know they cannot contain points in the desired range. This way we are able to discard those branches that are not of our interest and traverse the tree more efficiently.

On the other hand, to evaluate if a point is in the given range, we check if its longitude is greater or equal than the minimum of the given rectangle and less or equal than its maximum. Same with the latitude. If both fit the criteria, we can conclude we found one of the points we were looking for, we add them to the list passed to the function as a parameter and keep searching through our recursive function.

Aesthetic Optimization

First of all we initialize an array of points of size k, where each point's coordinates are Double.MAX_VALUE, we go down the tree through the root going into the children of that rectangle, the root in this case, choosing the child whose center is closest to the point entered. Before calling the function recursively on the next rectangle we mark the rectangle as visited, this is an array of rectangle.getchildren.size which exists in the function, because the function is recursive each function call will have an array corresponding to the rectangle.

We go down until the rectangle's children are points, we loop through the children, which we know are points and we add them to the array of k size if any of the points in the rectangle are closer to the point entered than the points in the array. We add as many as there should be and we go back up the tree to the previous call, in there the children will be rectangles again and if there is a chance the children can contain a point that is closer to the point entered than one of the points in the array we go down (we only go down if the rectangle is not marked as visited) when we go down we mark again, and so we repeat the process going up and down until we have an array where we are sure no other rectangle can contain a point closer to the point entered than any of the points in the array.

In order to know if there is a possibility of a rectangle containing a point that is closer than one in the array we take the smallest possible distance of the point to an edge of the rectangle, we do this with our method minDistanceBetween(Figure rectangle, Point point) in RTressFunctionalities.

Once we have the array finished we check if there are more circles or squares and display the most common.

Result Analysis

As previously mentioned, we have calculated the average time each algorithm took to run on the given datasets. The information we obtained is the following:

Basic Functionalities

First of all, before analyzing the algorithm it is expected that the execution time doesn't take very long as it is logarithmic.

When analyzing **addition** we decided to see how long it took to add a random point to the tree, in order to test this as accurately as possible we decided to add 10 different points and take the average for each dataset. The results are the following:

Dataset	Number of points	Time taken
XXS	8	0 ms
XS	16	0 ms
S	512	0 ms
M	8192	0 ms
L	65 536	0 ms
XL	262 144	0ms
XXL	524 288	0 ms

When seeing these results something looks wrong as they do not even reach a millisecond in execution time even in the XXL file. However we thought about it and the average number of iterations done to reach the bottom(i) of the XXL file when adding a shrub taking into account that our maximum children is 3 is the following:

$$3^i = 524\,288$$

$$\mathrm{i} = \log_3(524\,288)$$

Although this looks like a huge number when looked at like this, since we have the conception that the limit of a log of x when x tends to infinity is equal to infinity, we think the result will be very big, however the result of this is less than 12 iterations, by this we mean that the tree will have only about 11 levels, and for each level went down only three children will be looked at, which is very little in comparison to the amount of shrubs there are, this however still looks too small to be true. Which is why we ran some tests apart from the usual debugging, where we added a shrub to this dataset, checking with ctrl f that there

wasn't a shrub with the same coordinates, and then deleted it from the tree, if it didn't find the shrub we know it wasn't added, but in our case it deleted the shrub perfectly.

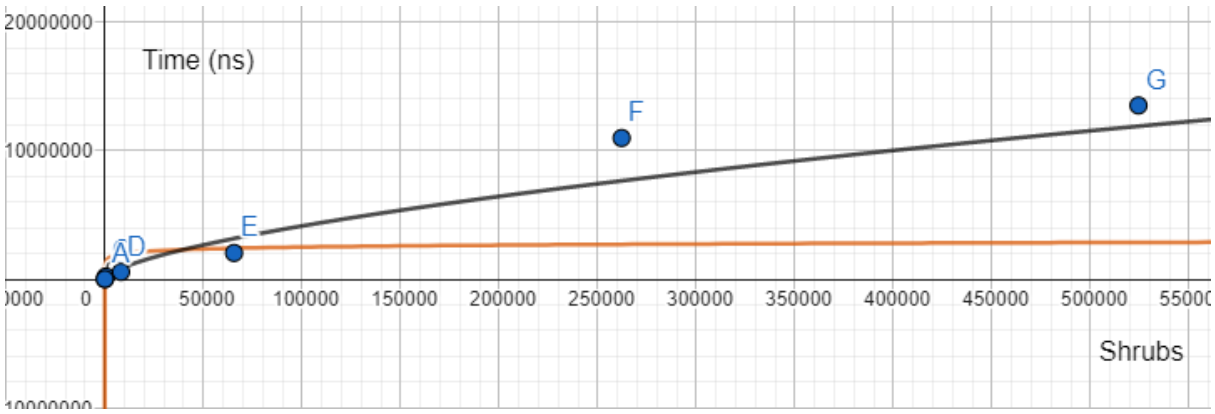
We also think this is due to the fact that in the majority of cases when adding it is rare that it overflows, so it only goes down 11 levels in the biggest dataset looking at only three children and then finishing all together.

The results we were getting lacked a correlation to an extreme where putting the execution in nanoseconds wouldn't bring any information and would only confuse making it look like a smaller dataset might take longer than a larger just because of compilation time varying from one to another.

When it comes to ***deletion***, we have tracked the time it takes to delete arbitrary points corresponding to different datasets:

Dataset	Number of shrubs	Time taken
XXS	8	10.40 us
XS	16	14.90 us
S	512	210.80 us
M	8192	563.30 us
L	65536	2.04 ms
XL	262144	10.98 ms
XXL	524288	13.51 ms

Plotting our data with shrubs in the X axis and the time taken in the Y axis together with the data regression curve, in black, and a logarithmic curve for reference, in orange, we see the following:



With the regression curve, which is a curve that tries to best fit the data, we can confirm our algorithm follows a logarithmic trend. Comparing it with the real logarithmic function curve, we notice a difference, of course, because it doesn't have a perfect logarithmic cost, however, the logarithm property is still there, as the time taken doesn't increase notably after a significant increase in the data to process. The expected cost is logarithmic but it depends on the current structure of the tree. In a deletion call, a point can be simply removed or this plus deleting around N rectangles and points and reinserting them continuously, so the performance can vary. Despite this, this deletion method is much more efficient in comparison to other data structures, which feature expensive deletions such as Binary Search Trees.

The reinsertion is done through the root, that is, starting from the top. This is done because we want to achieve an optimal distribution despite the higher cost. We could reinsert at the same level and achieve a cheaper cost, but the points would not be ideally distributed, increasing the cost of further operations, such as searches.

Visualization

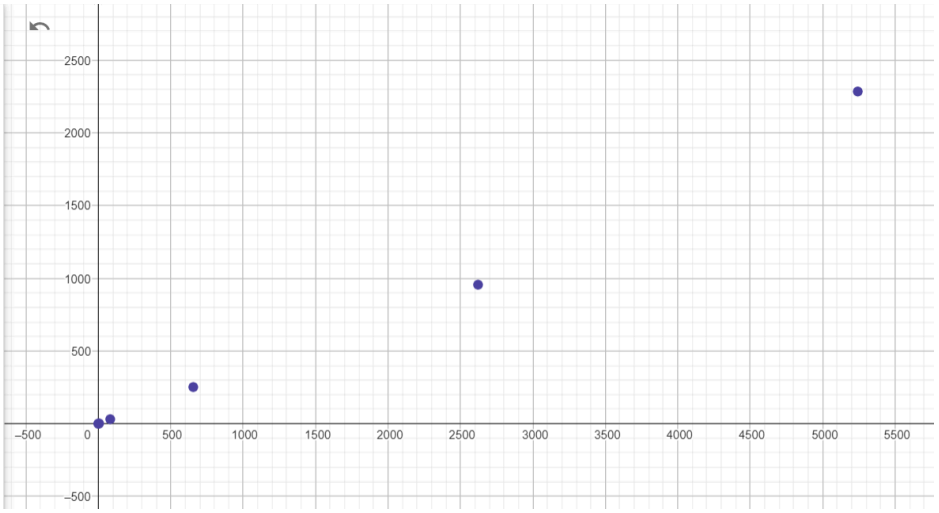
After measuring the time it took the program to traverse and draw all the figures we got the following table:

Dataset	Number of shrubs	Time taken
XXS	8	0.045 s
XS	16	0.108 s
S	512	1.869 s
M	8192	31 s
L	65536	252 s
XL	262144	956 s
XXL	524288	2285 s

The time of this visualization is much higher due to the laptop where it was run. In others the xxl lasts for approximately 1 or 2 mins.

As it can be seen the amount of time it takes compared to other functionalities is incredibly big. Not only that, but the time it increments as the number of shrubs grows is also incredibly high compared to the increment of others. It increases to the point where it is almost the same increment as the number of shrubs. Making this function has a cost of O(N). As it can be seen in the following graph where the table is plotted, but dividing by 100 the x axis as it grows at almost the same rate, but with much higher numbers, what difficulties to appreciate the relation.

x	y_1
0.08	0.045
0.16	0.108
5.12	1.869
81.92	31
655.36	252
2621.44	956
5242.88	2285



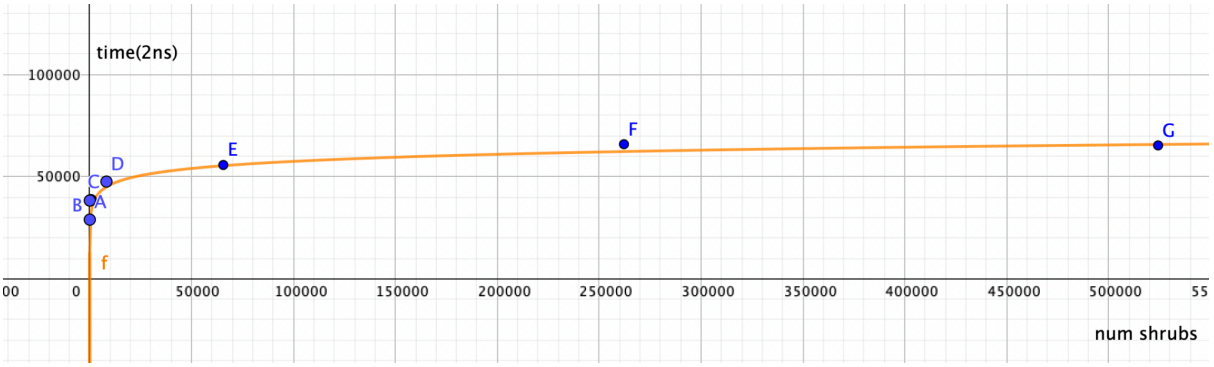
The reason as to why it takes that much is because this function, in contrast with others, traverses the whole r-tree. But that is not the worst, the worse is that it has to graphically draw figures through the graphics class of java swing which is pretty demanding. Moreover, as the coordinates of the points oscillate between 53.8 and 54 in terms of x and 4.14 and 4.15 in terms of y, a lot of calculations were needed to transform these coordinates into numbers between 0 and 1000 for the x and 0 and 500 for the y.

Search by Area

We measured the time that it took for our algorithm to obtain the shrubs located in a specified area between two points (trying to maintain a relationship between the number of total shrubs in the dataset and the output of our search). These are the results we obtained:

Dataset	Number of shrubs	Time taken
XXS	8	14.49 us
XS	16	19.15 us
S	512	19.26 us
M	8192	23.79 us
L	65536	27.86 us
XL	262144	32.94 us
XXL	524288	32.65 us

As you can see, the time taken for our algorithm to run increments as the datasets grow, but not significantly. Making it to so that when we plot these points into a graph, we get the following:



Notice how the shape of the graph is logarithmic, meaning that the cost of our function is O(log(n)). We are able to achieve this cost by the traversal we do in this type of data structure, allowing us to discard a set of points and rectangles whenever these ones don't overlap with the desired output one.

This cost can vary depending on the input we give to our algorithm. For instance, the best case scenario would be when we give it a small enough area to look for that only overlaps with a single rectangle that contains the points we are looking for, or even better: when we give it an area that doesn't return any output point as it doesn't overlap with any given rectangle. On the other hand, the worst case scenario of this algorithm would be when we give it a range that contains the entire set of points of our system. Meaning that it would need to traverse the entire tree, see that the searched area overlaps with all the rectangles and explore them all. Therefore, achieving a cost of O(n) by visiting all the points of our system.

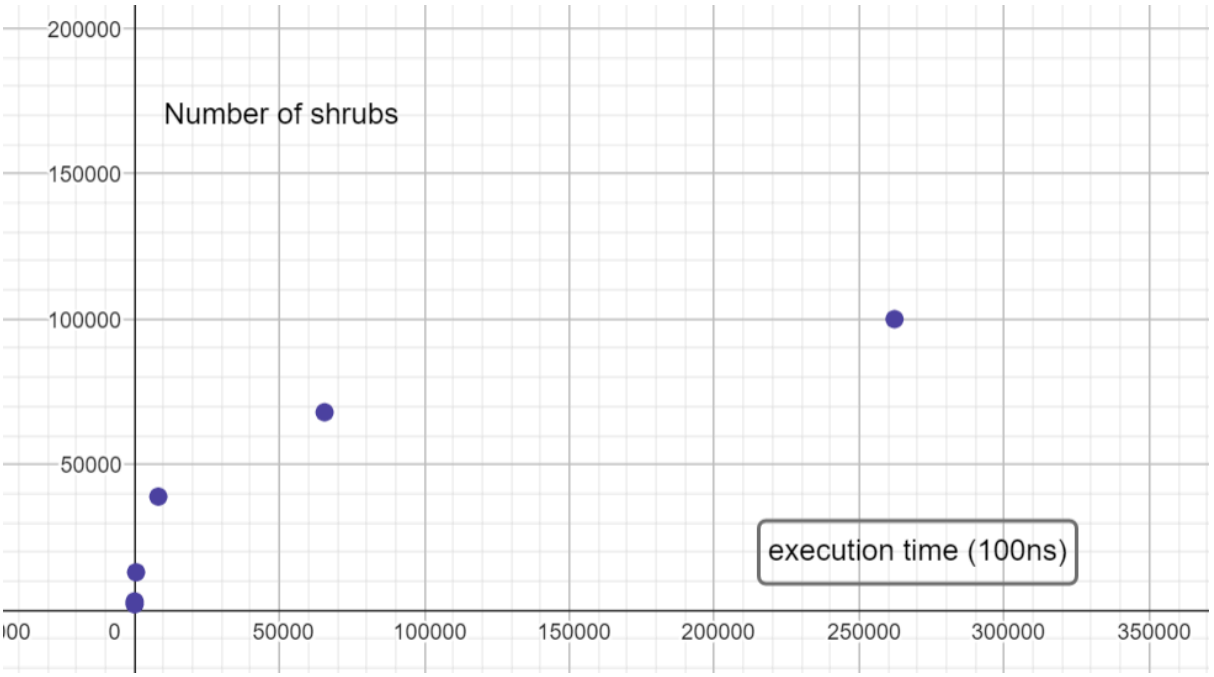
Nonetheless, for the average input that returns only a set of points in our tree, this search is able to achieve a very optimal performance and return them with a logarithmic cost, similarly to the other search by range we saw in the previous data structure.

Aesthetic Optimization

In order to analyze this functionality we decided to enter 5 different points for each dataset and get the average between them to get more accurate results and settle on k = 4 to measure all the datasets against the same standards.

The results we got are the following:

Dataset	Number of points	Time taken
XXS	8	0,2 ms
XS	16	0,3 ms
S	512	1,3 ms
M	8192	3.9 ms
L	65 536	6.8 ms
XL	262 144	10.0 ms
XXL	524 288	12.3 ms



As we can see in this graph, after manipulating the units (which is multiplying by a constant) we can clearly see the logarithmic shape of the graph. This makes sense as, when explained before in the addition analysis, we go down levels logarithmically, in this case we do pruning when going back up but it still allows us to see that the time complexity is in fact 0(log(n)).

Observed Problems

In the case of the addition we lack a perfect scenario where we have an infinite amount of shrubs as even extremely huge numbers make the execution time very small and this doesn't allow for a good time complexity analysis which is why no graph was included for this functionality. In this perfect scenario we would also have the rectangles set up so that we can have all of them about to overflow so that we can test how long it would take when

adding a point and all the way up we are performing operations, while in our current situation in almost the worst possible case scenario, two will overflow when going back up the tree.

Tables

Structure Design

We will now represent a completely different data structure, which can be (in certain cases) the most efficient one reaching a cost of *O(1)* as opposed to the current best cost we got which was *O(log(n))*. We achieve this by storing our data in an array of size *R*, as it can then be accessed with a cost of one.

This data structure is most used for pairs of values and keys (identifiers that represent the value). However, arrays limit this identifier to integers representing their position. To avoid this, we use a *hash()* function that associates a value (position) to each given key (not limited to any specific data type). Therefore allowing us to assign a position in our array to any given data we want to store.

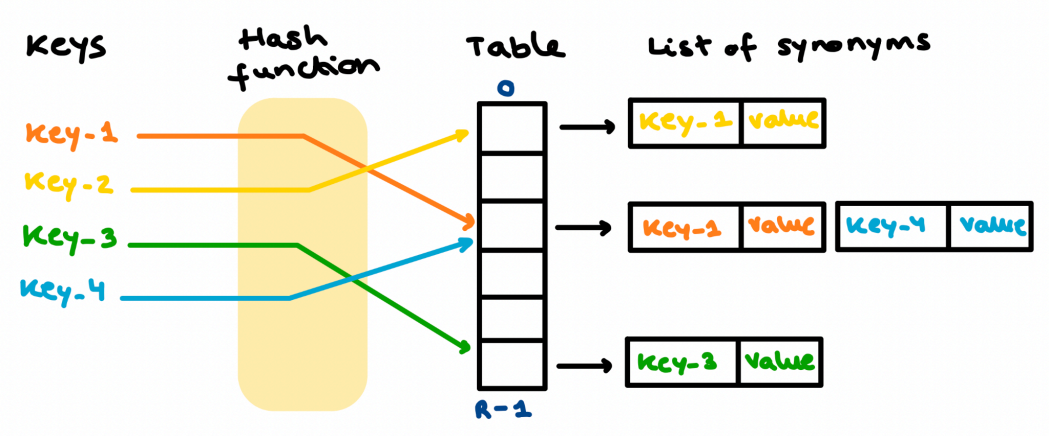
In our case, we believed that, given the datasets, it made most sense to define the key as the suspect's name (String), and their number of rabbits and occupation would be the values they represent (as we are assured the names of the suspects are unique). This way, we can determine with our *hash()* function the position in which each of the suspect's data is stored by only giving it their name (we explain this function in the section below).

As our hash function isn't perfect and can give us the same key for different input names, we needed to decide if we wanted our data structure to use open or closed hashing. Open hashing defines a synonyms list in each position of the array, allowing to store different elements (with the same key) together. On the contrary, closed hashing only allows one element per cell. Hence, it needs to use a second function for when two values fall on the same position (*rehashing*) to relocate them.

After thoroughly analyzing both possibilities, we opted for the first option, as we believed this one brought more benefits compared to closed hashing, especially for big datasets. That is, closed hashing can store a limited number of elements while in the synonyms list we could exceed the length of the array and don't run out of positions. This option does require more memory (as we define a list of elements for each cell), but it compensates by allowing a more flexible size of the array and increasing the performance of the data set.

Whenever the array starts to be quite full, inserting new elements can become a costly process in closed hashing because of the *rehashing* function trying to find empty positions. This also affects the searches, as each time we get a duplicate position we would have to apply the *rehashing* function as many times as needed to find the element we are looking for. However, with open hashing this process is significantly improved by checking the

different elements on each position, knowing the one we are looking for will for sure be there.



Overall, we believed that closed hashing's performance degraded way faster and wasn't worth the memory save, so we decided to take an open hashing approach by defining the array of type *Node*. Each node contains a Customlist of suspects and some methods that allow us to get a specific suspect from its list or remove it, among others.

Lastly, we had to determine the size *R* of our array used to implement the table. We finally decided to initialize this one to the number of suspects our dataset has, ensuring that even if our *hash* function worked incredibly well, we would still have enough space to fit all the given suspects in a different position. Also, as we use a list of synonyms we are not worried about new nodes not having space to be inserted.

Hash Function

We created a first version of the *hash()* function that summed the numerical value of the characters of the given name and took the modulus *R* of the resulting value. As we know, a *hash* function should be exhaustive, uniform, fast, and key independent. This first approach is fast, but not exhaustive as it doesn't generate all the possible values from 0 to *R*-1 (ASCII values are not very high and names aren't long). Also, it is too dependent on the key as, for instance, the names *Blake* and *Kaleb* would generate the same value (as they have the same letters). Moreover, it is not uniform as there are some values with a higher possibility of being generated (depending on the length of the word and use of common letters such as vowels).

Therefore, we kept this function for analysis purposes but decided to create a better one. *advancedHash()* satisfies the hash function required properties to a much better extent. To convert the data type to an integer value, instead of simply adding up the ASCII values of

each character, it adds the value but with a prior bit shifting, shifting to the left N times, where N is the character index. This shifting outputs a greater number than the original ASCII value, increasing the magnitude each iteration. The final outcome is typically a large number, we now have to prepare or generate a position in the table, as we're working with an array limited by its size, R, so we cannot directly use the outcome as an index. We obtain the index by squaring the output and obtaining the log(R) central digits. Because the central bits of a square number don't relate with the original value, we will achieve an even distribution throughout our array.

log(R) produces a decimal output, the typical procedure is to floor the value, in other words, round downwards, without the extra remainder the output will never be out of bounds with an array of size R. However, after doing some tests, we found that rounding up instead, and then simply doing the R modulus to avoid out of bounds, it outputs even more evenly distributed random numbers, so we decided to use this approach.

With this improved hash function, we ideally met the 4 requirements: exhaustivity, uniformity, key independence, and speed.

Implemented Algorithms

Basic Functionalities

We had to ensure our data structure allowed the basic operations of addition and deletion, so that the user could add or remove a suspect.

Adding a new suspect is an easy process in this dataset, as we only have to access the position of our table related to the key of our suspect (its name), which we get through the *hash* function. In terms of code: table[hash(name)]. As we had defined the array to be of type Node we can't directly add the suspect to this position. Therefore, we call the *addSuspect* function we created in the Node class that adds the Suspect instance we send as a parameter to the synonyms list of that node.

Regarding the deletion of a suspect, we repeat the process we did before of accessing the position of our table related to the key. Once we are in the correct node, we call the *deleteSuspect* function and we send it the key. This function searches in the list of suspects in that Node if one of them has the given key. If it finds it, it deletes it. However, if none of the suspects have the same key (meaning the user has entered a non-existent name), the function returns *FALSE* so that we can display the appropriate error message.

Edict of Grace

This functionality allows the user to mark a specific suspect as heretic or not. The system prompts the user to enter the corresponding name of a suspect and Y if yes and N if not to mark that suspect as heretic.

To do so, we access the position of our table related to the key of the suspect (its name), which is retrieved by the has function(table[has(name)]). Again, as our table is of type Node, we created a function *getSuspect* to which we send the key as a parameter and returns the desired suspect by looking for it through the list of synonyms of the Node on that position (returns *NULL* if suspect isn't found, indicating that the user entered a non-existent name and the corresponding error message gets displayed). If the user is found, then it will be marked or unmarked as heretic depending on whether the user wrote a Y or N.

Thanks to the hash table, we are able to retrieve the desired suspect having a cost of O(1), even though this can be increased depending on the size of our list of synonyms.

Final Judgment (One Suspect)

This functionality allows the user to search any suspect on our system by its name. The system prompts the user to enter the corresponding name and all the information from the suspect is displayed.

To do so, we access the position of our table related to the key of the suspect (its name), which is retrieved by the *hash* function (table[hash(name)]). Again, as our table is of type Node, we created a function *getSuspect* to which we send the key as a parameter and returns the desired suspect by looking for it through the list of synonyms of the Node on that position (returns *NULL* if suspect isn't found, indicating that the user entered a non-existent name and the corresponding error message gets displayed).

This is one of the example searches that make this dataset better than the rest, as we are able to retrieve the information of a specific element by achieving a (minimum) cost of O(1), even though this can be increased depending on the size of our list of synonyms.

Final Judgment (Range)

This functionality allows the user to find all the suspects that have seen a number of rabbits specified in a range. Once the user has input the minimum and maximum number of rabbits to use in our search, we call the *searchRange* function.

This time we do require a function to perform this task as opposed to the previous searches that could retrieve the information by directly accessing the desired position of the table. As we have to look through a range (and the suspects aren't sorted in any specific manner) we have no other choice but to traverse the entire table.

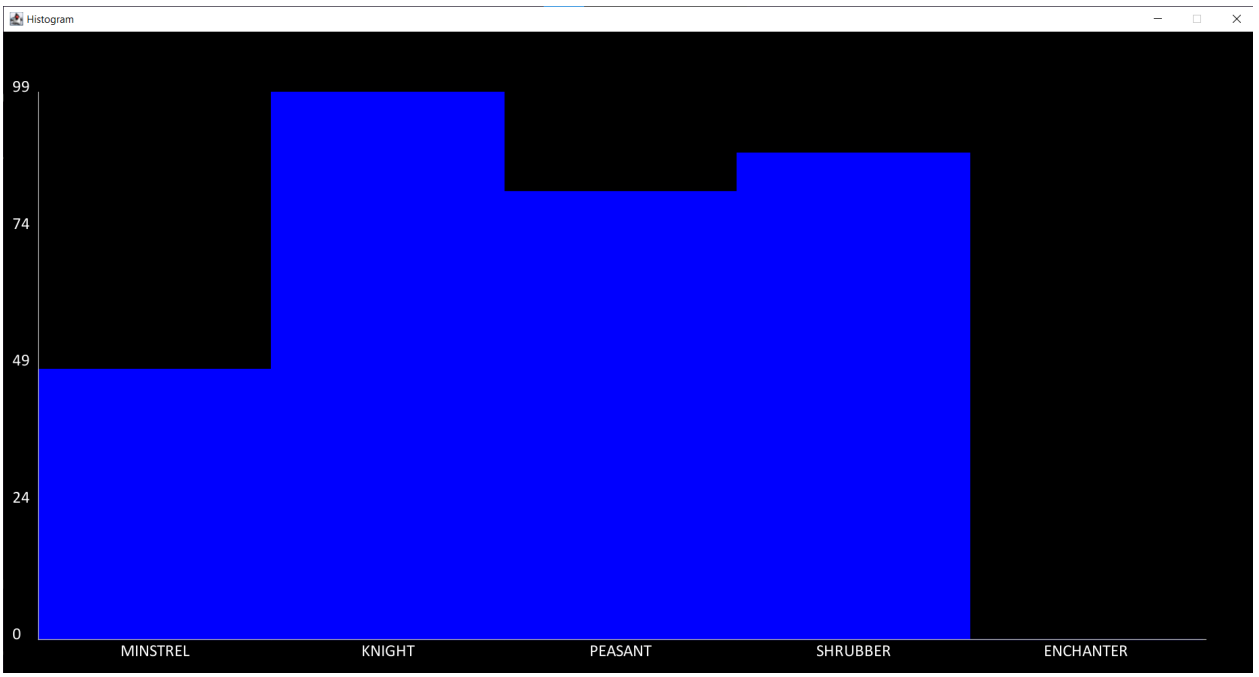
We do so by checking for every single position of our array if its node is empty (we created a method that checks if there is at least one suspect added to that node). If it has, we go through its suspects (list of synonyms) and check whether the number of rabbits they have seen fall inside our range. If they do, we add them to a custom list we then return after having checked all the positions of the table.

This is the perfect example of a non-optimal use of this data structure. As we can see, tables aren't handy for searches in a range (unless we implement them using some type of sorting), making this simple search have a cost of O(n), which other data structures could have achieved with O(log(n)).

Histogram

This functionality allows the user to graphically see how many heretics are there for each occupation, taking out the king, queen and clergyman occupations, as they can't be heretics.

To do so, we go through the whole table and count for each occupation how many heretics there are. After that, through java swing graphics class, a histogram is drawn with the data received. Drawing the histogram is pretty simple: 2 lines of the desired size are drawn as x axis and y axis, and then the height of the bar of the occupation with the most heretics is set as the length of the y axis, and the other occupations get a length proportional to the heretics they have. After that, we will get an histogram looking like that:



Result Analysis

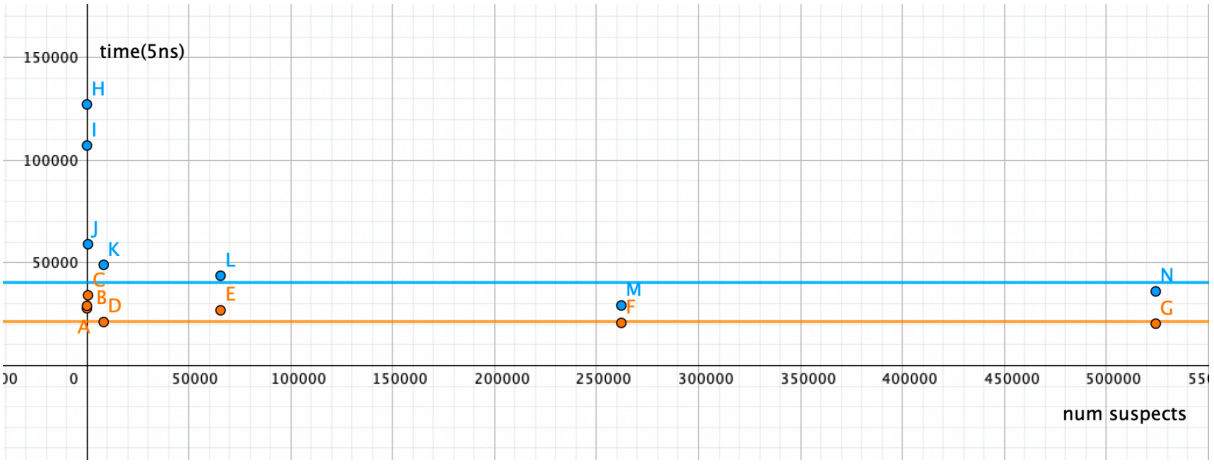
Once more, we have calculated the average time each algorithm took to execute on the given datasets and observed the following:

Basic Functionalities

We measured the time it took for our **insertion** algorithm to add a new suspect to our table in two cases: using our *hash* function and our *advancedHash* function. These are the results we obtained:

Dataset	Number of suspects	Time taken (*hash*)	Time taken (*advanced Hash*)
XXS	8	5.58 us	25.45 us
XS	16	5.825 us	21.44 us
S	512	6.865 us	11.83 us
M	8192	4.25 us	9.82 us
L	65536	5.39 us	8.76 us
XL	262144	4.16 us	5.86 us
XXL	524288	4.09 us	7.23 us

As you can see, this algorithm has almost a constant cost O(1), as it just consists of accessing a position of our table determined by the hash function. If we plot these points into a graph we get the following:



Notice how these show the constant cost of the algorithm (being the blue the measurements with *advanceHash* function and orange the one with its base version), as they don't increase as the datasets do, both remain more or less constant (except for the first dataset). The deviation of the first datasets is due to the fact that these are too small and our hash functions aren't good enough to provide a different position for all the given suspects, provoking more collisions.

Also, we believe this is affected by the initialization of the size of our table. We defined this one equal to the number of suspects in the given datasets. Therefore, for smaller datasets

there is a higher chance of collision that makes our algorithm slow down a bit when inserting new elements in the same position as others.

To understand the difference between the blue and orange lines, we should know that the *advanceHash* function has a lot more logical operations than the *hash* one, which makes it have a higher cost. This cost will then be compensated in searches when the access of information is optimized by reducing the amount of collisions. Even if this takes more time when inserting, it will make it easier and faster to find our information as we will see below.

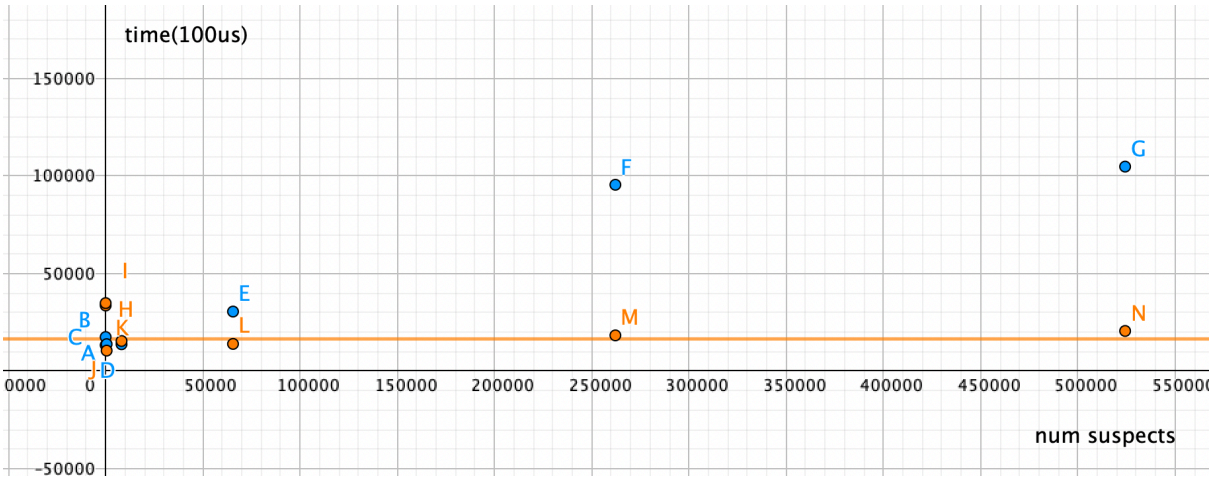
We then analyzed the cost of our **deletion** function by measuring the time that it took to delete a suspect from our table using the *hash* and *advancedHash* functions. We obtained the following:

Dataset	Number of suspects	Time taken (*hash*)	Time taken (*advanced Hash*)
XXS	8	133.12 us	336.12 us
XS	16	174.53 us	348.52 us
S	512	138.32 us	105.17 us
M	8192	137.75 us	154.5 us
L	65536	305.28 us	139.37 us
XL	262144	0.955 ms	183.12 us
XXL	524288	1.048 ms	205.71 us

As you can see in the table, our algorithm behaves quite differently when using one of the hash functions or the other one. The cost of this one increases significantly as the datasets do for the simple version of the hash function. However, when using the *advancedHash* functions the cost remains somewhat constant. This is because of the numerous collisions the simple version of the function creates and how they affect every time we want to delete a user.

If you recall, we previously mentioned that our *delete* function accesses the position of the table related to the given key inserted by the user, and checks throughout the whole list of elements of that position if one of them equals the one to delete (to ensure we are not trying to delete a non-existent suspect). Therefore, whenever we have multiple collisions in our table this process is slowed down significantly, which is exactly what causes the time increase on our algorithm when using the *hash* function. However, if there aren't many collisions in our table (using an improved version of the hash function), the cost of deleting a suspect remains constant almost independently on the input (just affecting the amount of time our *advancedHash* function takes to generate a key, which increases as the name gets longer).

If we plot the given points on the table onto a graph we obtain the following, with blue representing the performance of our algorithm with the *hash* function and orange the *advancedHash.*



We can clearly see how the *advancedHash* function lets us achieve a constant cost O(1) keeping it almost the same as the datasets increase (with a little deviation in the first dataset as we already explained in *insertion*). However, we can see how the cost of the algorithm using the *hash* function significantly increases in comparison to the orange one, quadrupling the time it takes to eliminate a suspect in the biggest dataset.

Overall, we have seen that the better our hash function is, the better our *deletion* algorithm works, achieving an homogeneous cost for any given dataset. However, the performance of our hash function doesn't affect our *insertion* algorithm that much, keeping its cost almost constant for any hash function. We can see how the basic functionalities on this dataset have a way smaller cost than in the previous ones, making this data structure very interesting to use in certain cases (not all, as we will see below).

Edict of Grace

Dataset	Number of suspects	Time taken (hash)	Time taken (*advanced hash*)
XXS	8	24.7 us	34 us
XS	16	22.1 us	33.1 us
S	512	27.3 us	38.8 us
M	8192	29.9 us	38.2us
L	65536	25.4 us	25. 7us
XL	262144	29 us	24.2 us
XXL	524288	23.5 us	24.8 us

In the table it can be seen how, as said before, the cost is O(1). Notice how, it does not matter how many suspects there are, the time the algorithm takes to find the suspect the user wanted is always the same(almost the same). This is because given a name, with function hash we directly access the position where that suspect is. We do not search until we find it, we just directly go to where that suspect is.

We wanted to compare the effect of the two *hash* functions we had designed on this scenario and counted the time it took to run in both cases. As you can see, the naive approach of the function works better for the small datasets and worse for the big ones. This is due to the fact that our advanced version of this function has a higher cost than the simple one. Therefore, everytime we access an element of our table, we accumulate this extra cost of the function. This cost is usually worth the time we save while finding a specific suspect (not dealing with that many elements in the same position), especially in larger datasets where there are a lot of suspects that could have similar keys. However, for smaller datasets, this implementation is unnecessary and a simpler approach works better because dealing with repetitions of a few elements doesn't take a big amount of time and doesn't compensate for the increased cost of the function.

Final Judgment (One Suspect)

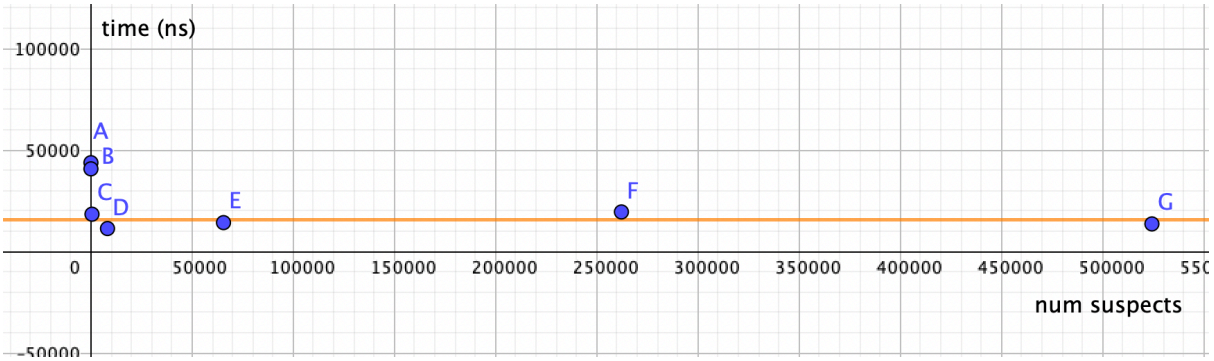
Dataset	Number of suspects	Time taken (*hash*)	Time taken (*advanced Hash*)
XXS	8	13.26 us	43.79 us
XS	16	17.25 us	40.75 us
S	512	10.06 us	18.374 us
M	8192	7.65 us	11.31 us
L	65536	14.03 us	14.23 us
XL	262144	22.97 us	19.51 us
XXL	524288	118.55 us	13.61 us

This table proves the point we made before and demonstrates it is possible to achieve a cost of O(1) for searches in tables. Notice how, as the size of the datasets increase significantly, the time the algorithm takes to run remains (approximately) constant. This is because with this data structure, we are able to directly access the element we want to retrieve by using the *hash* function, without having to search for it in our table.

We wanted to compare the effect of the two *hash* functions we had designed on this scenario and counted the time it took to run in both cases. As you can see, the naive approach of the function works better for the small datasets and worse for the big ones. This is due to the fact that our advanced version of this function has a higher cost than the simple one. Therefore, everytime we access an element of our table, we accumulate this extra cost of the function. This cost is usually worth the time we save while finding a specific suspect (not dealing with that many elements in the same position), especially in larger datasets where there are a lot of suspects that could have similar keys. However, for smaller datasets, this implementation is unnecessary and a simpler approach works better because dealing with repetitions of a few elements doesn't take a big amount of time and doesn't compensate for the increased cost of the function.

Also, notice how the first two datasets have a deviation compared to the other ones. As we previously explained, this is related to the small size of the datasets, having a higher chance

of collision that makes our algorithm slow down a bit when accessing a specific position of our table, and having to go through all the elements stored there.



If we plot the points from the table above (using the *advancedHash* function) in a graph we get the following, proving that our algorithm is behaving with a constant cost O(1), as it doesn't increase as the number of suspects do (without considering the deviation on the first datasets).

As we can see, this is the most optimal search we have been able to achieve in this project, proving the cases in which this data structure is more optimal than the others. However, this is not the case for any given type of search, as we will see next.

Final Judgment (Range)

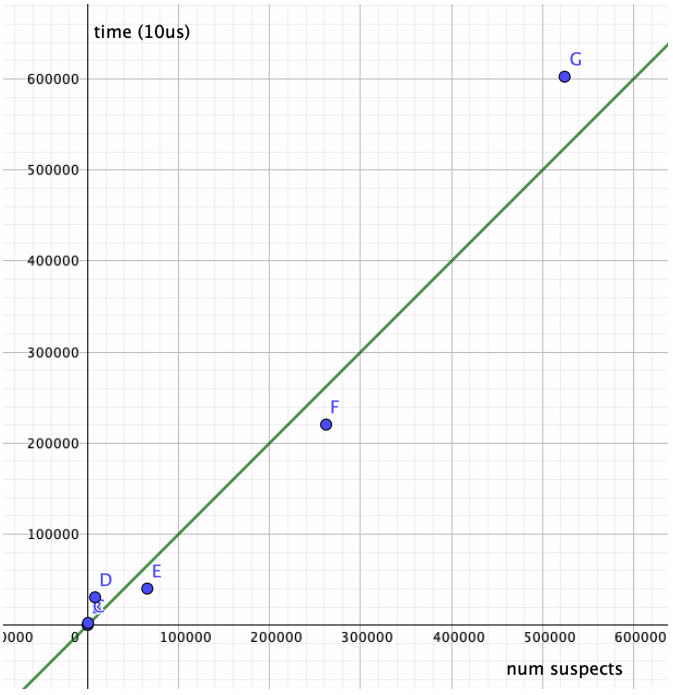
Dataset	Number of suspects	Time taken (*hash*)	Time taken (*advanced Hash*)
XXS	8	17.47 us	22.52 us
XS	16	31.43 us	38.72 us
S	512	0.295 ms	0.245 ms
M	8192	3.019 ms	3.06 ms
L	65536	3.742 ms	4.016 ms
XL	262144	20.42 ms	22.03 ms
XXL	524288	42.566 ms	60.252 ms

As you can see, this algorithm gives us different results from the previous one, as this one does significantly increase as the datasets grow. This is because, as we mentioned before, this data structure is not optimal for searches in a range. As our suspects are not sorted in any special matter, we have no way to predict where those that have seen a number of rabbits in a specific range will be. Therefore, we have to traverse our entire tree to find the suspects we are looking for. This makes our algorithm have a cost of O(n).

As we can see, for this specific algorithm, a simple implementation of the *hash* function gives better results. This is because in our code for this function we access our table in a sequential manner, going through all its positions. Therefore, we are not making use of the *hash* function to retrieve a suspect in a specific position, meaning we only use the function

when inserting all the suspects of our dataset. As for this specific function, we don't care if the positioning of the suspects in the table is optimal (as we will go through all the positions anyways), we get more or less the same search performance for both, but more cost for when we use the *advancedHash* function as it takes slightly longer to access the information if its is stored in multiple positions than in a single one. That is, it is somewhat faster to retrieve five elements of the same List than accessing the first element of five different Lists. Hence, as the simple version of the *hash* function gives us more collisions, this one performs faster when checking for all the elements of the table, having to check less Lists.

Notice how in this specific case, the worst possible implementation of our *hash* function: all keys generating the same position, would make this algorithm perform the best, giving us the smallest possible cost having all the elements in the same List.



If we plot the times we measured with our *advancedHash* function into a graph we obtain the one above. In this one, we can clearly see how there is a complete dependance between the time the algorithm takes to run and the number of suspects, proving that this one has a cost of O(n), seeing that when the number of suspects increases, so does the time (at approximately the same speed).

Overall, this case proves how the abuse of this data structure is not useful for all the scenarios. As we can see, this algorithm gives us the worst cost in a range search out of all the algorithms of this type we have seen in the different data structures. Therefore, the optimal cost of O(1) achieved on doing a single search is then compensated by the

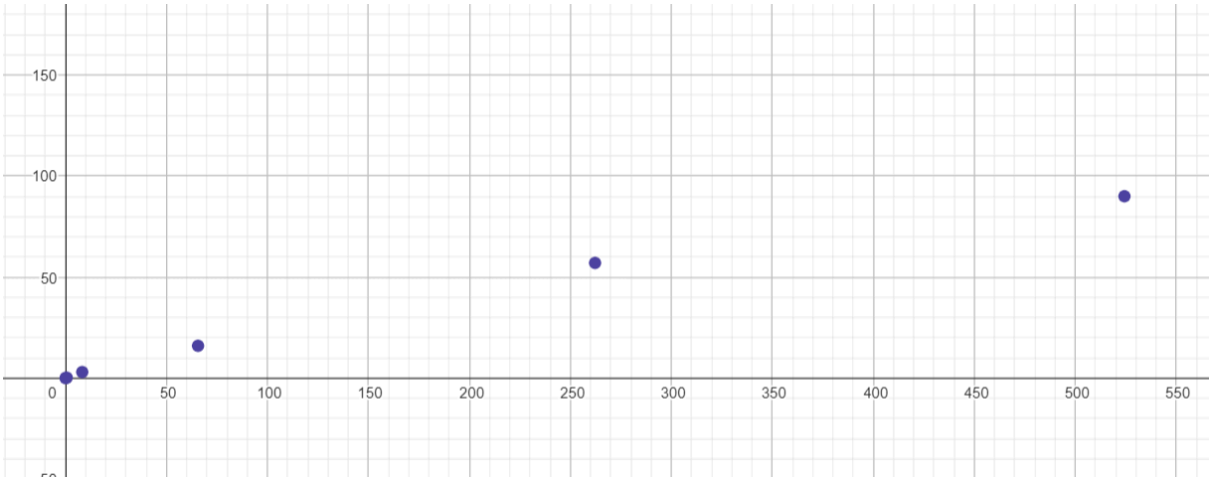
increasing cost obtained in this algorithm, proving it should only be used in specific cases when it can satisfy the requirements.

Histogram

Dataset	Number of suspects	Time taken
XXS	8	0.022 ms
XS	16	0.036 ms
S	512	0.225 ms
M	8192	3 ms
L	65536	16 ms
XL	262144	57 ms
XXL	524288	90 ms

As it can be seen in the table, here the times are not constant and are way bigger. That is because this function has nothing to do with hash tables. As in this function we are just going through the whole table and counting how many heretics for occupation there are. Therefore, no hash function or advancedHash has been used, as we just loop through the whole table and get the occupation of each suspect and if they are heretics or not.

Notice how as it loops through the whole table the cost of the algorithm is of O(N). The more suspects there are the more time it takes. It can be seen clearer in the function below:



Observed Problems

One of the main issues (as so to speak) we encountered while implementing this data structure was the size of the internal array. We initially decided to determine this one by the number of suspects in the given data structures. This approach seems to work for the biggest datasets. However, we noticed some anomalies in the cost of our algorithms for the two smallest datasets. After analyzing what could be the source of the time deviation we were getting, we concluded that our table's size was causing multiple collisions for the smallest sizes.

As we use the size of the array in our hash function, multiple of the keys generated had the same positions in the table. Therefore, causing more collisions and slowing down our algorithms a bit. To fix this, we believe it would be better to define a minimum size for our internal array (probably around 50 elements) to ensure that our hash function is exhaustive enough.

Apart from this little issue, this data structure was the most intuitive to implement and we did not face any problems in the process.

Testing Methods

We have used multiple tools to ensure our algorithms were working as expected before delivering the project. The most common one amongst the different data structures has been the debugger tool incorporated in IntelliJ, which is the IDE we have used to code this project. By making use of breakpoints we have been able to track the performance of our algorithms and see how they were acting in different situations. Also, this tool was extremely useful to check our memory during runtime to validate our algorithms were following the processes correctly.

Another simple but effective tool we have used on multiple occasions is a *printDebug* function that displays the data we deem necessary. For instance, as we previously mentioned, for BSTs we created a function of this type that displayed the tree by showing each node and their children with their ids, weights, heights, and balance factors. This was extremely useful while developing the balancing algorithm to confirm it was able to detect all the unbalanced situations.

Moreover, we sometimes created our own datasets to test the algorithms with more intuitive data. For instance, we created a dataset for RTrees that used points that were further away from each other (on non-decimal positions) to be able to clearly check that the search by ranges or the visualization algorithms were working as expected.

Obviously, we also validated, whenever it was possible, the outputs of our algorithms with the data provided to identify any potential mistake.

Lastly, to approve our algorithms were performing at an accepting cost, we counted the time it took for them to run on multiple occasions and computed an average, for all the given datasets. This allowed us to properly analyze their performance and validate our work.

Conclusions

The development of this project has helped us reinforce all the concepts learned in APDS classes and widened our knowledge regarding the use and optimization of data structures as well as their basic algorithms and traversals.

Working with these data structures with the different datasets and problems has helped us identify their strengths and weaknesses and recognize when it is more convenient to use each of them. Also, by implementing each data structure on our own, we have been able to comprehend how they work and more precisely predict how certain situations will affect their behavior.

For instance, we now know that the most optimal way to store one dimension data (if this will only be accessed individually, not performing any kind of range search) is a Table, as it can provide us a O(1) cost. However, if we need to perform searches that might include more than one output, then BSTs are the way to go, as these still provide a considerably low cost for traversals (O(long(n))).

On the other hand, if the data we want to store is two dimensional, RTrees might be the best option, as these provide optimal searches also achieving a logarithmic cost. Lastly, if we want to store data represented as a network of connections, a Graph is the appropriate dataset to use, as it allows its elements to share multiple links between them.

Furthermore, the analysis of the given data structures has shown us how the advanced features that improve our algorithms such as balancing in Binary Trees or having an improved hash function in Tables has a big cost. However, this cost is compensated by the improvements we see in traversals and search functions when these are implemented, especially for the largest datasets.

Lastly, implementing our own Custom List has provided us a deeper understanding of how a List works internally (especially ArrayLists) and the cost of each of their methods.

Overall, we have learned that there isn't a specific answer to the question: which is the most optimal data structure? As this one depends on both the data to be stored and how we need to access this data. As programmers, we believe we have gained enough knowledge as to identify these cases in the future and be able to optimize our data storage and representation.

Bibliography

These are the sources from where we obtained the information to develop this project:

Pol Muñoz. *APDS - 2nd Semester Notes*. September 2022

- https://estudy2223.salle.url.edu/mod/resource/view.php?id=11696

Pol Muñoz. *R-Trees Insertion*. September 2022

- https://estudy2223.salle.url.edu/mod/resource/view.php?id=11697

David Galles: Data Structure Visualizations. 2011

- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html