Mini-proyecto 2: QuadTree

Integrantes: Ignacio José Barrias Concha
Nicolas Andrés Pino Leal
Matias Ignacio Urrea Moya
Asignatura: Estructura de Datos
Grupo 16

Introducción

En el presente proyecto se lleva a cabo la implementación de la estructura QuadTree con el propósito de demostrar lo que se aprendió durante todo el semestre en el curso.

En el trabajo se ponen a prueba los conocimientos de los estudiantes respecto a análisis teórico de algoritmos, tipos abstractos de datos, uso de contenedores, árboles y colas de prioridad, presentes en diversas partes del código. También se pone en práctica el leer archivos externos, el saber hacer un buen análisis experimental con grandes cantidades de datos y poder representar datos obtenidos con el código.

QuadTree

El QuadTree es una estructura de datos que pretende representar de forma compacta y reducida los puntos de un espacio bidimensional; se omite completamente el almacenamiento de un espacio si en este no hay ningún punto. El QuadTree sigue la estructura de un árbol, como se puede inferir de su nombre, teniendo una raíz y nodos con relación padre-hijo.

Cada nodo equivale a una zona del plano, con el nodo raíz representando al plano completo. Los nodos se componen de los siguientes elementos:

- Una lista que guarda 4 nodos hijos, que representan los cuadrantes del plano o subdivisiones del espacio que el nodo representa.
- Un par de coordenadas correspondiente al límite superior-izquierdo del nodo.
- Un par de coordenadas correspondiente al límite inferior-derecho del nodo.
- Información relevante al cuadrante. En el próximo testeo, esta información corresponde a ciudades del mundo, y se almacena su población.
- Una variable que indica si el nodo es blanco o negro. Un nodo negro representa un espacio en el que hay al menos un punto, mientras que un nodo blanco simboliza un espacio completamente vacío. Un nodo blanco, al no tener puntos, no tendrá nodos hijos, puesto que no es necesario ocupar almacenamiento en múltiples subdivisiones que también estarán vacías

Mientras un nodo sea negro, tendrá a sus 4 hijos que representan sus 4 cuadrantes, los cuales también tendrán sus propios hijos, y asi sucesivamente hasta llegar a la división más mínima: en que las hojas del QuadTree representan a un solo punto.

La implementación de la estructura, si bien es en gran parte original, fue hecha en base a la implementación recursiva encontrada en la página GeeksforGeeks (https://www.geeksforgeeks.org/quad-tree/), propuesta como referencia al momento de entregar el proyecto. Las funciones que esta incluye y que se usan en nuestra implementación son inboundary(), search() e insert(). También se toman las estructuras de los Nodos y del QuadTree en sí.

Las funciones que se encuentran en el QuadTree son las siguientes:

- totalPoints(): Función que retorna un valor entero que describe la cantidad total de puntos dentro del plano.

- totalNodes(): Función que retorna un entero que entrega la cantidad total de Nodos negros y blancos en todo el plano 2D.

- countRegion(): Función que recibe un punto y una distancia que describen un área de una región del plano, su objetivo es retornar la cantidad de puntos dentro de dicha zona.

- aggregateRegion(): Función semejante a countRegion(), su objetivo es retornar el total de población dentro de un área específica.

Todas estas funciones descritas, tienen un tiempo de O($log_4(n)$) gracias a que todo el proceso es hecho por la siguiente función.

- search_inRegion(): Función que se encarga de realizar el trabajo de contar un total entero y retornarlo dependiendo del caso solicitado, los casos son:

Caso 1: Solicita la cantidad de puntos (Nodos negros en las hojas) existentes de una región determinada.
Caso 2: Solicita el total de población ubicada en una región determinada
Caso 3: Solicita la cantidad de tanto de Nodos blancos y negros en una region determinada.

Para ello recibe el quadtree principal y viaja por los hijos 4 hijos dependiendo si está dentro del área de búsqueda y distinto de *NULL*, lo primerot se verifica gracias a la función auxiliar inRect().

El caso 1 fue diseñado para cumplir el rol de entregar la cantidad entera pedida para countRegion(), mismo comportamiento tiene el caso 2 para realizar el trabajo de aggregateRegion(). Finalmente, tanto el caso 1 como el caso función 3 llegan a ser utilizadas como contratación de las funciones de totalPoints() y totalNodes() para entregar los datos en todo el plano existente. search_inRegion() tiene un rendimiento de tiempo logarítmico (O($log_4(n)$)).

- _list(): Es una función que simplifica el proceso de "**List()**" pues sirve para agilizar la llamada inicial a la función, termina creando el contenedor.

- List(): Es una operación la cual retorna un contenedor con todos los puntos almacenados en el QuadTree. Por cada punto retorna sus coordenadas y su valor asociado. Esto se realiza adentrándose al quadtree de manera recursiva, teniendo en cuenta las condiciones que han de tener los cuadrantes, como serían no ser punteros a nulo, ni ser considerados "**blancos**"(los cual es no tener un punto-ciudad).

Después, al encontrar un punto-ciudad, se guarda en un archivo "**Prueba.txt**" que separa los elementos de la línea de la forma "**posición x. posición y; población**", al finalizar el proceso, se insertan los puntos-ciudad gracias a "**search(Point p)**" y se almacenan en un vector de punteros a nodo.

La complejidad de todo list() recae en la búsqueda inicial de los nodos, pues este se desplaza por los cuatro cuadrantes, y al encontrar uno lo guarda en el .txt, realizando esto en O(log n). Después se busca el punto-ciudad guardado en el .txt y se almacena en un contenedor, esto último se hace 'n' veces. Por tanto la complejidad final sería de O(log(n) + nlog(n)) por ende quedaría O(n log(n)).

- Escribir(): función auxiliar que ayuda a la conservación de los datos escribiendolos en el .txt que luego se utilizaran en List(). Es convocada por _list().

- Search(): Es una función que busca la existencia de un punto en específico, esta baja por el árbol comparando los límites del cuadrante en que está con el punto a encontrar. Finalmente, pregunta si el cuadrante en el que se encuentra es nulo si es así pues devuelve "NULL", si no, preguntará si el nodo existe, si no es así, entonces retorna "NULL" de nuevo, pero sí existe el nodo retorna el puntero al nodo. Esta función tiene un complejidad O(log n) pues baja todo el árbol en el peor caso y la altura del árbol es 'log n'.

- Insert(): Es una función que va creando cuadrantes en los cuales irá un nodo, los cuadrantes inician con límites conocidos como "topLeft" y "botRight". La función va buscando entre estos cuadrantes y se va introduciendo más a fondo en ellos hasta llegar a valores unitarios en los cuales inserta el nodo, este tendrá una posición dada y cantidad de población. Esta función tiene una complejidad O(log n) pues tiene que bajar (sumergirse) en los cuadrantes hasta llegar al final del mapa (valores unitarios de mapeado).

- inRect(): Función auxiliar que retorna una variable **bool** encargada de verificar si una región está dentro de otra, para verificarlo solamente verificamos ciertas condiciones que declaran falsa la proposición, en el caso

de que ninguna es aprobada significa que chocan entre sí, por ende retorna como verdadero. Su complejidad de trabajo se resume en O(1). Esta función fue elaborada gracias a investigar por Internet, en una página que presenta la detección de colisiones en dos dimensiones (Link en bibliografía).

Los datos utilizados son aportados por el dataset World Cities Population el cual nos da más de 3 millones de ciudades con la información necesaria para el proyecto.
Todo los experimentos se realizaron en un computador hp Pavilon con 8 gigas de ram y intel core i5. El repositorio con la implementación se encuentra en https://github.com/Pinox084/Proyecto2_Datos

Análisis experimental de las inserciones

Estos datos son analizados en cantidades distintas de inserción que incrementan hasta ser la cantidad total de ciudades para obtener una mejor referencia del tiempo empleado en distintas áreas de espacios.

Intento	Cantidad	Tiempo de inserción
1	200000	551.115 ms
2	200000	540.53 ms
3	200000	567.42 ms
4	200000	560.21 ms
5	200000	569.1 ms
6	200000	540.73 ms
7	200000	550.56 ms
8	200000	523.12 ms
9	200000	542.55 ms
10	200000	560.58 ms
11	200000	579.96 ms
12	200000	589.98 ms
13	200000	579.44 ms

14	200000	649.34 ms
15	200000	570.75 ms
16	200000	589.42 ms
17	200000	529.25 ms
18	200000	529.63 ms
19	200000	540.58 ms
20	200000	543.57 ms
	Promedio	551.115 ms

.

Intento	Cantidad	Tiempo de inserción
1	500000	1232.79 ms
2	500000	1213.82 ms
3	500000	1178.63 ms
4	500000	1194.05 ms
5	500000	1197.87 ms
6	500000	1181.88 ms
7	500000	1188.19 ms
8	500000	1189.34 ms
9	500000	1189.04 ms
10	500000	1234.31 ms
11	500000	1171.4 ms
12	500000	1161.77 ms
13	500000	1200.01 ms
14	500000	11.67.85 ms

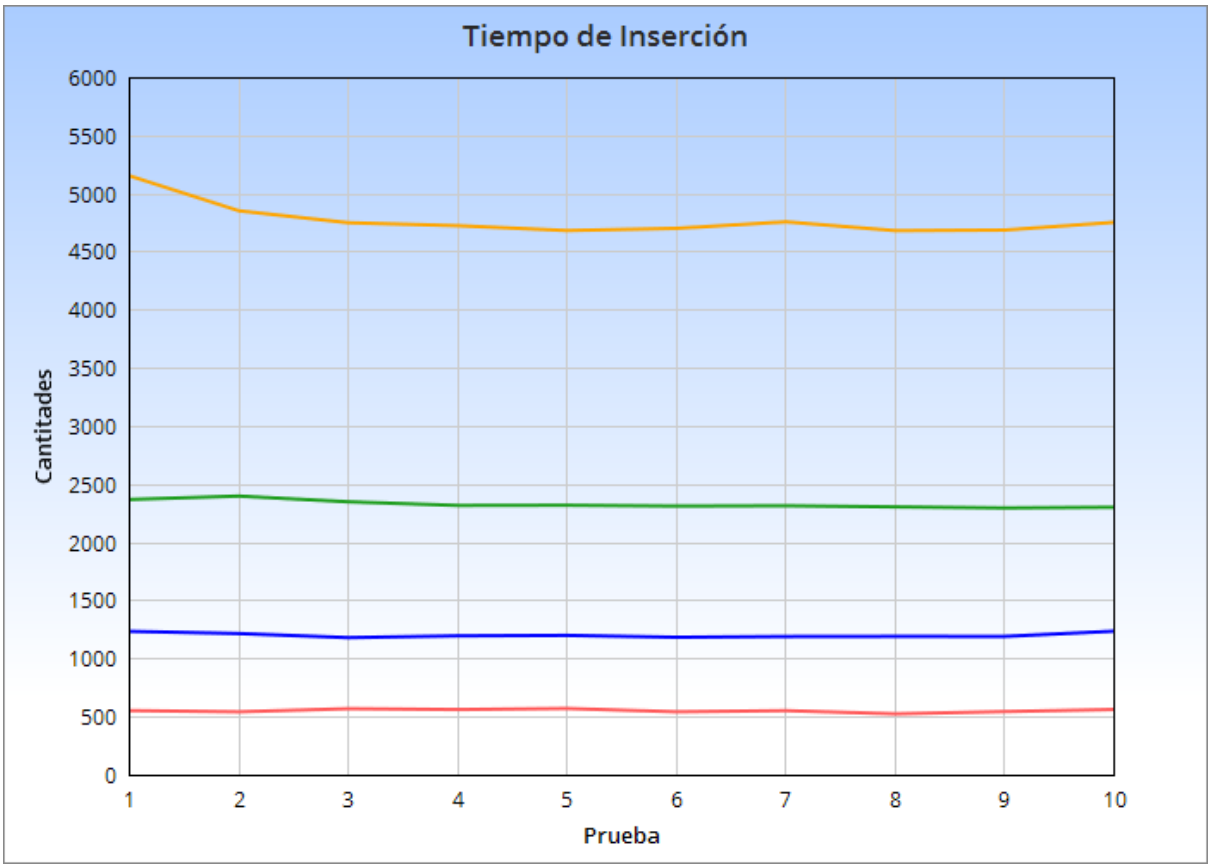
15	500000	1214.31 ms
16	500000	1181.3 ms
17	500000	1200.14 ms
18	500000	1167.04 ms
19	500000	1178,43 ms
20	500000	1164,31 ms
	Promedio	1171,37 ms

Intento	Cantidad	Tiempo de inserción
1	1000000	2367.18 ms
2	1000000	2396.18 ms
3	1000000	2347.16 ms
4	1000000	2315.84 ms
5	1000000	2317.36 ms
6	1000000	2310.55 ms
7	1000000	2313.12 ms
8	1000000	2303.57 ms
9	1000000	2294.09 ms
10	1000000	2300.38 ms
11	1000000	2319.71 ms
12	1000000	2324.92 ms
13	1000000	2317.38 ms
14	1000000	2318.36 ms

15	1000000	2333.65 ms
16	1000000	2335.07 ms
17	1000000	2333.29 ms
18	1000000	2325.03 ms
19	1000000	2393.02 ms
20	1000000	2328.34 ms
	Promedio	2329.71 ms

Intento	Cantidad	Tiempo de inserción
1	2000000	5151.28 ms
2	2000000	4850.23 ms
3	2000000	4747.64 ms
4	2000000	4722.35 ms
5	2000000	4680.46 ms
6	2000000	4699.26 ms
7	2000000	4755.43 ms
8	2000000	4679.78 ms
9	2000000	4684.56 ms
10	2000000	4751.22 ms
11	2000000	4680.81 ms
12	2000000	4854.37 ms
13	2000000	4751.86 ms

14	2000000	4702.93 ms
15	2000000	4709.07 ms
16	2000000	4682.86 ms
17	2000000	4748.13 ms
18	2000000	4770.04 ms
19	2000000	4780.35 ms
20	2000000	4796.27 ms
	Promedio	4759.94 ms



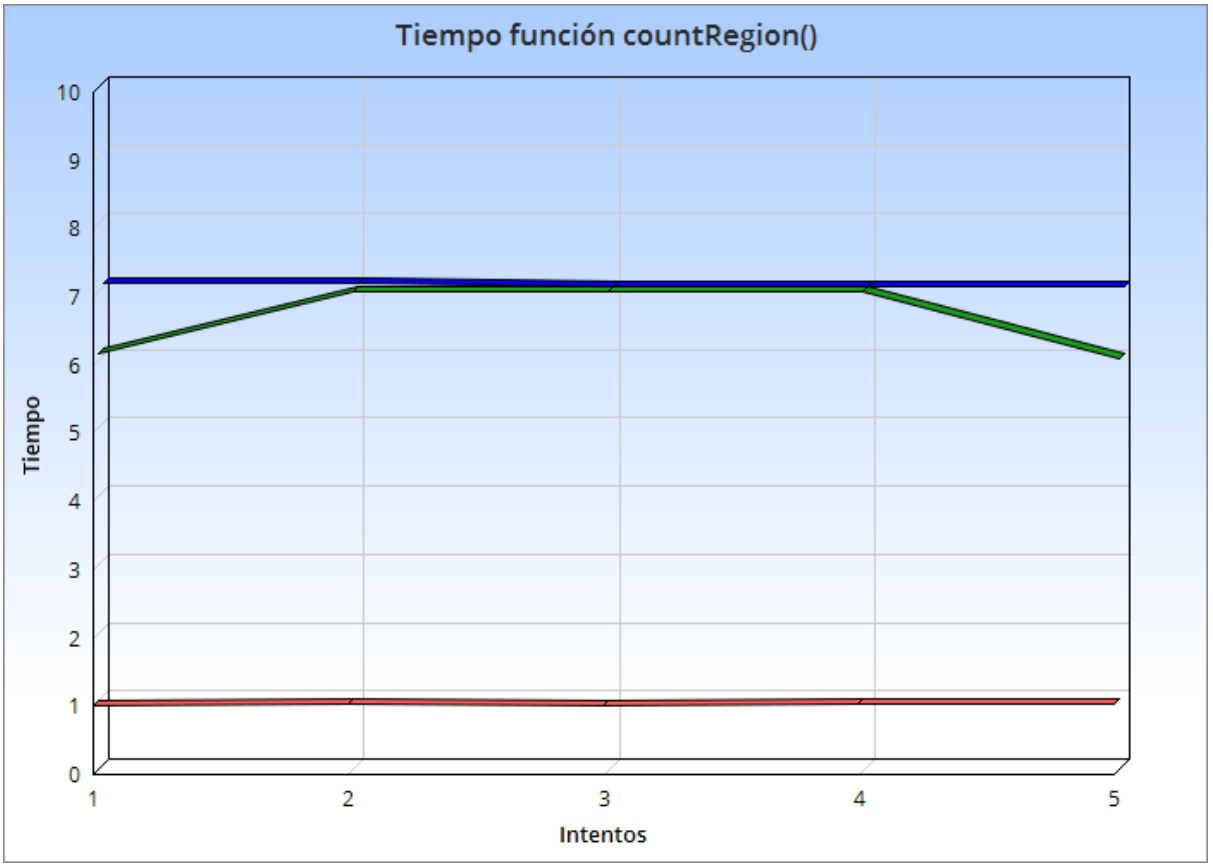
- Rojo: 200000 elementos
- Azul: 500000 elementos
- Verde: 1000000 elementos

- Amarillo: 2000000 elementos

- Análisis experimental función countRegion()

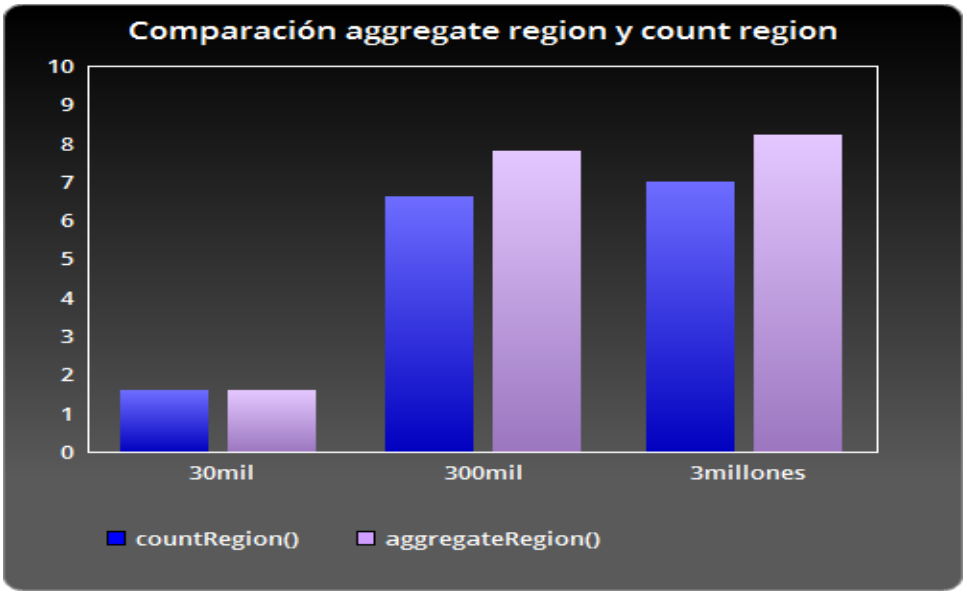
Esto extrae el tiempo empleado al buscar la cantidad de ciudades en distintos áreas de búsqueda.

Intento	Área	Tiempo
1	30000 X 30000	1 ms
2	30000 X 30000	1.01 ms
3	30000 X 30000	0.99 ms
4	30000 X 30000	1.015 ms
5	30000 X 30000	1.014 ms
6	300000 X 300000	6.09 ms
7	300000 X 300000	6.98 ms
8	300000 X 300000	6.98 ms
9	300000 X 300000	6.997 ms
10	300000 X 300000	6.005 ms
11	3 mill. X 3 mill.	7.04 ms
12	3 mill. X 3 mill.	7.03 ms
13	3 mill. X 3 mill.	6.98 ms
14	3 mill. X 3 mill.	6.98 ms
15	3 mill. X 3 mill.	6.98 ms



Áreas<:

- Rojo: 30000 x 30000
- Verde: 300000 x 300000
- Azul: 3 millones x 3 millones

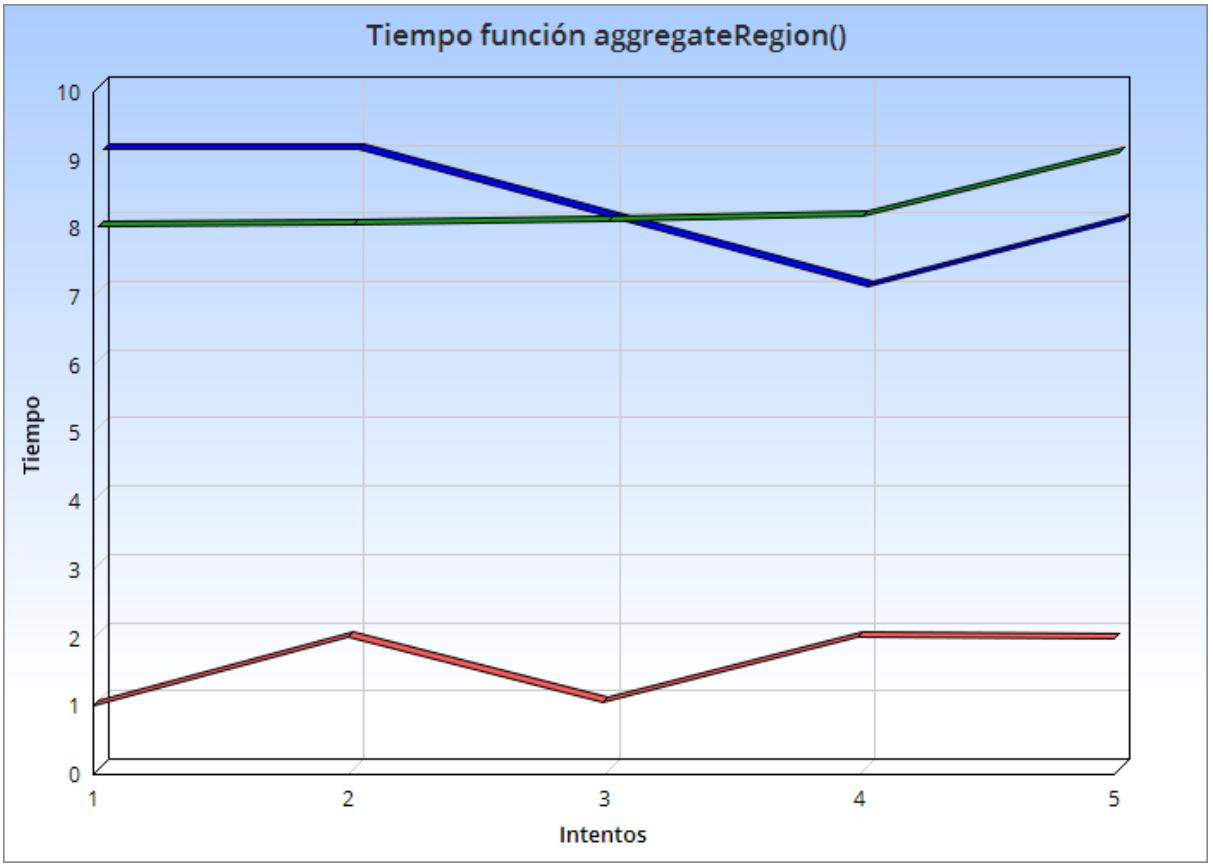


Comparación de los tiempos de ejecución en ‘ms’ de las funciones aggregateRegion y countRegion.

Análisis experimental función aggregateRegion()

Mismo experimento que la función del countRegion(), pero en este caso entregando el total de la población del área solicitada.

Intento	Área	Tiempo
1	30000 X 30000	0.99 ms
2	30000 X 30000	1.99 ms
3	30000 X 30000	1.043 ms
4	30000 X 30000	2 ms
5	30000 X 30000	1,98 ms
6	300000 X 300000	7.95 ms
7	300000 X 300000	7.97 ms
8	300000 X 300000	8.02 ms
9	300000 X 300000	8.1 ms
10	300000 X 300000	9.02 ms
11	3 mill. X 3 mill.	9.002 ms
12	3 mill. X 3 mill.	9.012 ms
13	3 mill. X 3 mill.	7.996 ms
14	3 mill. X 3 mill.	6.98 ms
15	3 mill. X 3 mill.	7.98 ms



- Rojo: 30000 x 30000
- Verde: 300000 x 300000
- Azul: 3 millones x 3 millones

Conclusión

La finalidad principal del QuadTree es la representación de un plano bi-dimensional, del cual uno puede insertar y extraer información más rápidamente que con un arreglo matriz convencional, esto se obtiene gracias a su comportamiento de árbol y funciones que llegan a obtener un mejor rendimiento de búsqueda, pero esta rapidez se obtiene en sacrificio de un mayor espacio de almacenamiento requerido para funcionar. Pero aun así resulta ser una buena herramienta al obtener datos en un área de datos muy grande.

Bibliografía

1. Base Quad Tree: https://www.geeksforgeeks.org/quad-tree/
2. Colisiones: https://ants.inf.um.es/staff/jlaguna/tp/tutoriales/colisiones/index.html
3. Lectura .csv: https://parzibyte.me/blog/2021/04/16/leer-csv-cpp/