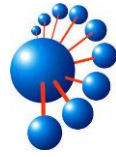




Universidad
de Concepción



Mini Proyecto 2

Implementación de QuadTree

Integrantes:

Alonso Isaías Bustos Espinoza

Sebastián García péndola

Ingrid Triviño Silva

Profesor: José Sebastián Fuentes Sepúlveda

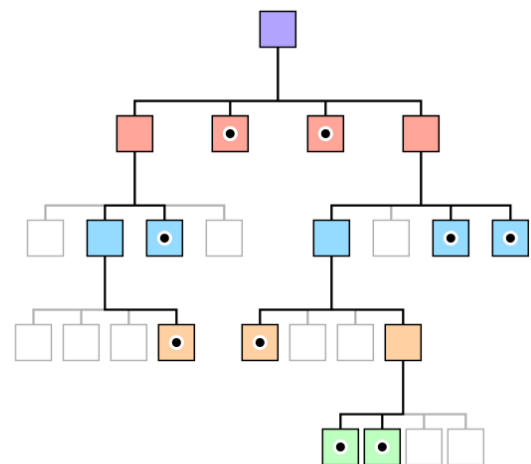
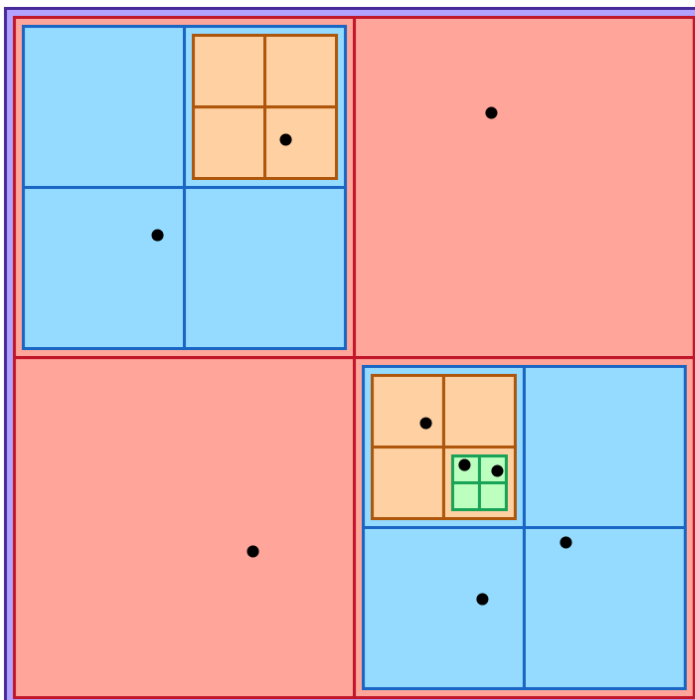
Miércoles, 28 de Junio de 2023, Concepción

1. Introducción

Para este miniproyecto buscaremos aplicar lo aprendido durante todo el curso, demostrando así que somos capaces de aprender cualquier tipo de estructura de dato que necesitemos durante nuestra carrera profesional.

Esta vez, la estructura de datos a estudiar e implementar será **QuadTree**, la cual es un tipo de árbol en que cada nodo tiene 4 hijos, permitiendo así operaciones sobre claves bidimensionales y siendo capaz de representar planos 2D sin desperdiciar espacio en memoria.

Esta implementación tendrá un análisis experimental en el que los datos utilizados serán extraídos del gigantesco dataset “*World Cities Population*”, el cual está compuesto por la información de más de 3 millones de ciudades del mundo y dispondremos de datos como su nombre, población estimada, coordenadas geográficas, entre otros campos.



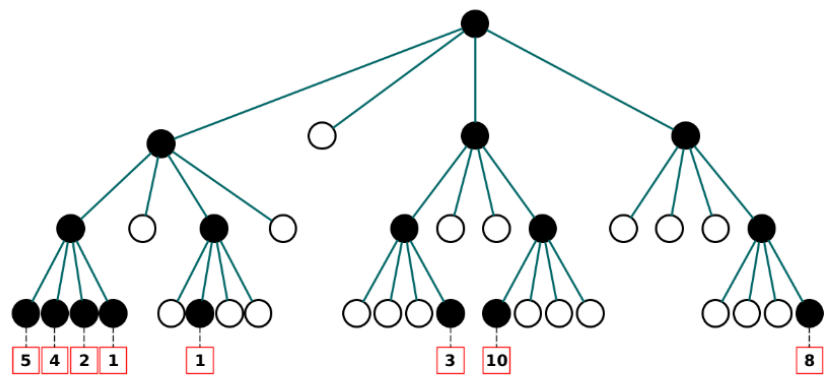
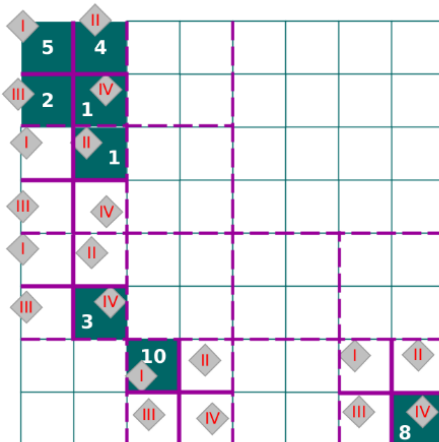
Representacion de Quadtree en forma de cuadrantes y árbol asociado a cada uno de estos

2. Descripción de la implementación

Como su nombre lo indica, el QuadTree es un árbol con padres de 4 hijos, por ende, su principal componente serán **Nodos**, los cuales están compuestos por:

- Una lista de **4 enlaces a nodos hijos**, representando cuadrantes del plano.
- **Dos pares de puntos** que representan los límites del cuadrante representado por el nodo. Un punto corresponde a la esquina superior-izquierda del cuadrante y el otro la inferior-derecha.
- Un campo para almacenar **información adicional relacionada al cuadrante** (Por ejemplo: el nombre de una ciudad, su población, etc.).
- Un campo que representa el **tipo de nodo**, el cual puede tomar los valores **blanco o negro**. Si un nodo es blanco, indica que el cuadrante que representa no contiene puntos, mientras que un nodo negro indica que el cuadrante que representa contiene al menos un punto.

En cuanto a la construcción del QuadTree, esta consistirá en un **nodo raíz que representará el cuadrante total del plano**, el cual solo será dividido en el caso de posteriores inserciones de puntos al plano, las cuales **dividirán al plano solo en los cuadrantes que contienen al punto**. Los cuadrantes que contengan puntos y los que no serán distinguidos por tipo “Negro” (contiene puntos) y “Blanco” (no contiene puntos). Estos pasos se aplican de forma recursiva hasta encontrar celdas individuales y se almacena la información asociada a cada punto del plano.



Nodos juntos sus colores correspondientes (Blanco/Negro), según los puntos contenidos



En más detalle, a continuación, se hará una revisión general de la implementación escrita en código:

- **QuadTree.hpp:** Archivo que contiene declaración de la clase QuadTree

Class QuadTree: Representa los nodos de QuadTree

Enum RegionType: Representa el tipo de región de los nodos de QuadTree

RegionType type: variable tipo **“Enum”** que indica si el nodo es blanco o negro.

vector<CityNode> nodes:* un vector el cual almacena **nodos “posición”** llamados **CityNode** que contienen el punto, población y nombre de cada ciudad.

QuadTree topL_QT, topR_QT, botL_QT, botR_tQT:* variables que representan los nodos hijos del QuadTree según la posición de su cuadrante.

Point topL_point, _botR_point, botL_point: puntos que representan los límites del cuadrante representado por el nodo actual de QuadTree. Uno de ellos corresponde a una modificación para facilitar los cálculos dentro de las funciones.

int count, dataSum: Variables para almacenar la cantidad de puntos ingresados, y suma de los datos de los hijos de cada QuadTree respectivamente.

- **QuadTree.cpp:** Archivo que contiene implementación/definición de métodos de QuadTree

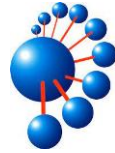
QuadTree(Point _topLeft, Point _botRight, bool first): Constructor de la clase QuadTree que recibe 2 puntos los cuales son **topL_point** y **_botR_point** respectivamente. Luego, **si first es verdadero**, se calcula **botR_point** de forma de que el plano tome un tamaño el cual es **potencia de 2** y sea posible así la **división exacta de cuadrantes**.

~QuadTree(): Destructor de la Estructura QuadTree, que elimina todos los cuadrantes y CityNodes almacenados en QuadTree.

int totalPoints(): Retorna la cantidad de puntos (CityNodes) almacenados en el QuadTree la cual se encuentra almacenada en la variable **count**.

int totalNodes(): Función recursiva que retorna la cantidad de nodos de QuadTree, tanto blancos como negros en el QuadTree, estos **se calculan de recorriendo de manera postorder** el QuadTree.

void insert(Point p, int data, string city): Es una **función recursiva que busca la región correspondiente a la posición ingresada**, de manera que a medida que se recorren los nodos internos del QuadTree estos se actualizan, **creando los subcuadrantes correspondientes** y actualizando el conteo de nodos junto a la suma de los datos.



vector<CityNode*>list(): Retorna una lista almacenada en un vector con todos los CityNodes de QuadTree. Por cada nodo contiene las coordenadas, su población asociada y su nombre de su ciudad. Para esto se recorre recursivamente el QuadTree.

int countRegion(Point p, int d): Retorna la cantidad de puntos en una región del plano, tomando como centro el punto **p** y una distancia **d**. Primero se verifica que el nodo se encuentre dentro de la región para luego llamar a la **función recursiva auxiliar** **countRegionAux()**.

int aggregateRegion(Point p, int d): Retorna la población total estimada dentro de una región del plano, tomando como centro el punto **p** y una distancia **d**. Primero se verifica que la región está contenida por el cuadrante de la raíz, para luego hacer la llamada a la **función recursiva auxiliar** **aggregateRegionAux()**.

Además, se agregaron los siguientes métodos que nos fueron de utilidad en la implementación:

getType(): retorna el **type** del nodo de QuadTree

inBounds(Point p): Sirve **verificar si el punto p está dentro de la región** delimitada por **topL_point**, **_botR_point**.

aggregateRegionAux(Point pTL, Point pBR, QuadTree* QT): función recursiva que calcula de manera pre-order la población de la zona asignada.

countRegionAux(Point pTL, Point pBR, QuadTree* QT): función recursiva que calcula de manera pre-order la cantidad de nodos de la zona asignada.

search(Point p): Busca un punto dentro del plano escogiendo el cuadrante donde pertenece y de manera recursiva realiza **search(p)** en ese cuadrante hasta encontrar el punto.

dataSumChange(Point p, int res, int sum): Modifica el dataSum en caso de alguna alteración de la data de un CityNode almacenado anteriormente, en este caso resta **res** del **dataSum** de cada subcuadrantes al que pertenece el punto y les suma **sum**.



- **Point.hpp:** Archivo que contiene declaraciones de la clase Point
Class Point: Representa posiciones en QuadTree (Contenido en CityNode)
int x, y: variables para manejar las ubicaciones en el plano y realizar cálculos
double dx, dy: variable que almacenan posiciones originales con decimales
- **Point.cpp:** Archivo que contiene implementación/definiciones de métodos de Point

Point(double u, double v): Constructor que da los valores a *dx* y *dy* respectivamente y mediante una ecuación la cual es multiplicar por mil y transformar a int se da valor a *x* e *y* respectivamente

Point(int u, int v): Constructor el cual da valor solamente a *x* e *y* con valor *u* y *v* respectivamente

setPoint(double u, double v): Editar valores de posición (decimales)

setPoint(int u, int v): Editar valores de posición (enteros)

getX(): obtiene posición en *x*

getY(): obtiene posición en *y*
- **CityNode.hpp:**
Class CityNode: Representa conjunto de información de datos de ciudades

Point point: posición de ciudad asociado al nodo en el plano

int data: dato (población) asociado al nodo

string city: nombre de ciudad asociado al nodo
- **CityNode.cpp:**

CityNode(Point p, int d, string c): constructor de *CityNode* recibe como parámetro las variables *point*, *data* y *city* respectivamente

3. Análisis teórico

A continuación, se describe la complejidad de cada operación según el análisis teórico realizado:

int totalPoints(): $O(1)$ como retorna el `count` de QuadTree que es una variable es solo acceso por lo que es constante

int totalNodes(): $O(n \log n)$ ya que recorre recursivamente todos los nodos pertenecientes al QuadTree

void insert(Point p, int data, string city): $O(n)$ primero verifica `inBound(p)` que es $O(1)$, luego realiza operación asignaciones y comparaciones en cuanto a ciclo, el primero `for()` sirve para manejar colisiones y lo cual en el peor caso lo hace $O(n)$ por cada inserción, luego seguiría con operaciones constantes hasta que llega la parte en la que se llama recursivamente a `insert()` en el cuadrante al que pertenezca a `p` lo cual es $O(\log n)$ pero como el peor caso anterior solo se hace en la última hoja entonces el peor caso para el `insert()` es $O(n)$ pero por lo general tendrá un tiempo de $O(\log n)$

vector<CityNode*> list(): $O(n \log n)$ recorre recursivamente el QuadTree para encontrar todos los nodos que han sido ingresados

int countRegion(Point p, int d): $O(n \log n)$ se realizan operaciones constantes de comparación y asignación además de `inBounds()` hasta llegar a `countRegionAux()` el cual es $O(n \log n)$

int aggregateRegion(Point p, int d): $O(n \log n)$ se realizan operaciones constantes de comparación y asignación además de `inBounds()` hasta llegar a `aggregateRegionAux()` el cual es $O(n \log n)$

QuadTree(Point _topLeft, Point _botRight, bool first): $O(1)$ ya que solo se hacen operaciones de asignación y en caso de ser el primer QuadTree algunas comparaciones, pero esta llega a un tiempo constante que no depende de la entrada

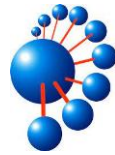
~ QuadTree(): $O(n)$ Destructor de QuadTree, debido a que elimina todos los sub-Quadtree y todos los CityNode.

Además, se agregaron los métodos:

getType(): $O(1)$ ya que solo retorna la variable `type`

inBounds(Point p): $O(1)$ ya que solo retorna si se cumple o no operaciones entre variables la cuales son constantes

aggregateRegionAux(Point pTL, Point pBR, QuadTree* QT): $O(n \log n)$ se recorre recursivamente el todo QuadTree para encontrar todos los `dataSum` útiles, en el peor de los casos es $O(n \log n)$ aunque por lo general es mejor.



countRegionAux(Point pTL, Point pBR, QuadTree* QT): $O(n \log n)$ se recorre recursivamente el todo QuadTree para encontrar todos los **count** útiles, en el peor de los casos es $O(n \log n)$ aunque por lo general es mejor.

search(Point p): $O(\log n)$ se hace comparación de **p** con los cuadrantes de QuadTree y luego se procede realizar **search(p)** recursivamente en el cuadrante al que **p** pertenece haciendo el recorrido $O(\log n)$ hasta llegar a una rama de QuadTree que contenga el punto .

dataSumChange(Point p, int res, int sum): $O(\log n)$ se resta **res** y suma **sum** $O(1)$ luego se hace comparación de **p** con los cuadrantes de QuadTree y luego se procede realizar **dataSumChange(Point p, int res, int sum)** recursivamente en el cuadrante al que **p** pertenece haciendo el recorrido $O(\log n)$ hasta llegar a una rama de QuadTree que contenga el punto.

4. Análisis resultados experimentales

El software utilizado fue:

- **Sistema Operativo:** Windows 11
- **IDE:** Visual Studio Code
- **Compilador:** g++ (GNU Compiler Collection)

El hardware utilizado fue:

- **CPU:** AMD Ryzen 5 5600G (4.20 GHz)
- **RAM:** 16 GB DDR4 (3200 MHz)

El enlace al repositorio con el código fuente es: [Ado-do/MiniProyecto2-ED](https://github.com/Ado-do/MiniProyecto2-ED)

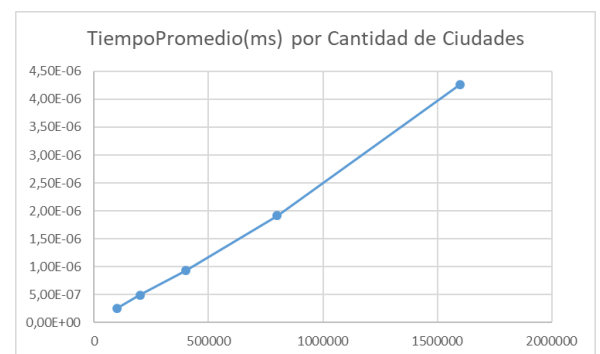
Para realizar los experimentos de prueba de la implementación se usará un dataset, en este caso, utilizaremos el **dataset World Cities Population**, el cual está compuesto por la información de más de 3 millones de ciudades del mundo. Para cada ciudad disponemos de datos como su nombre, población estimada, coordenadas geográficas, entre otros campos. Este dataset se encuentra disponible en la web.

A continuación, se describirá cada resultado obtenido de las pruebas de cada método:

De forma general, se realizaron pruebas con distintas cantidades de datos (ciudades), las cuales fueron 100.000, 200.000, 400.000, 800.000 y 1.600.000.

- **insert():** En el caso del insert se tiene un aumento de tiempo progresivo a medida que aumenta el número de entrada que, aunque la recta hace parecer que es lineal este no lo es ya que en caso de ser lineal los tiempos serían bastante mayores al igual que debería ir más hacia arriba en este caso se asume que ese debido a la cantidad de operaciones constantes que se puede ver esa diagonal

CantidadCiudades	TiempoPromedio(ms)
100000	2.56E-07
200000	4.92E-07
400000	9.27E-07
800000	1.91E-06
1600000	4.27E-06



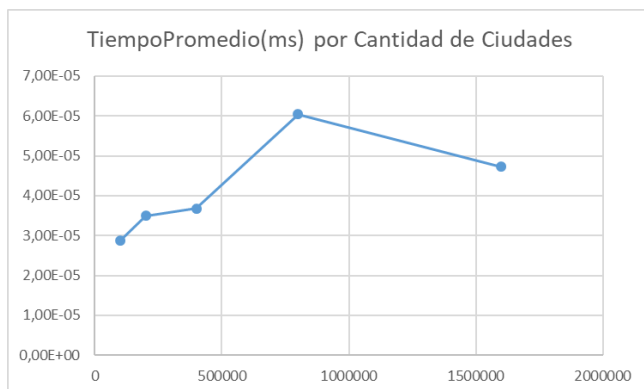
- **countRegion()** y **aggregationRegion()**:

En el caso de **countRegion()** y **aggregationRegion()**, se utilizó un radio **d** de $2^{((3*i)+2)}$ con **i** de 1 hasta 5, esto para tener una variación significativa entre cada uno de ellos.

Además, estos tuvieron valores muy similares y esto es debido a que sus funciones son casi idénticas, solo cambiando que le valor a sumar: en el primero es **count** y el segundo es **dataSum**. Pero en cuanto a todo lo demás, son iguales, haciendo que su recorrido a la misma **p** con la misma **d** sea de igual tiempo, además se puede apreciar el aumento logarítmico a medida que aumentan los datos.

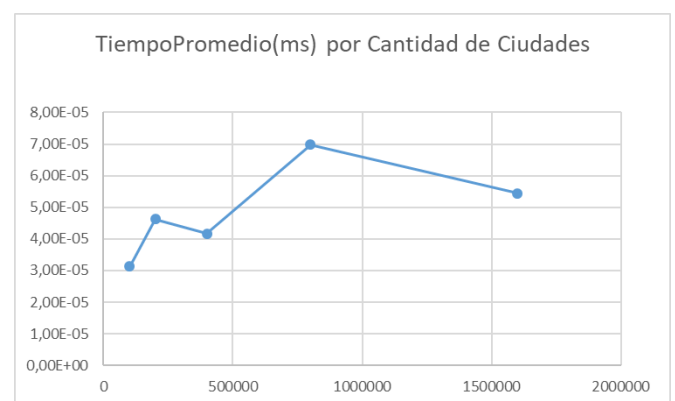
CantidadCiudades	TiempoPromedio(ms)
100000	2.88E-05
200000	3.50E-05
400000	3.68E-05
800000	6.04E-05
1600000	4.73E-05

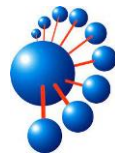
countRegion()



CantidadCiudades	TiempoPromedio(ms)
100000	3.13E-05
200000	4.62E-05
400000	4.16E-05
800000	6.98E-05
1600000	5.44E-05

aggregationRegion()



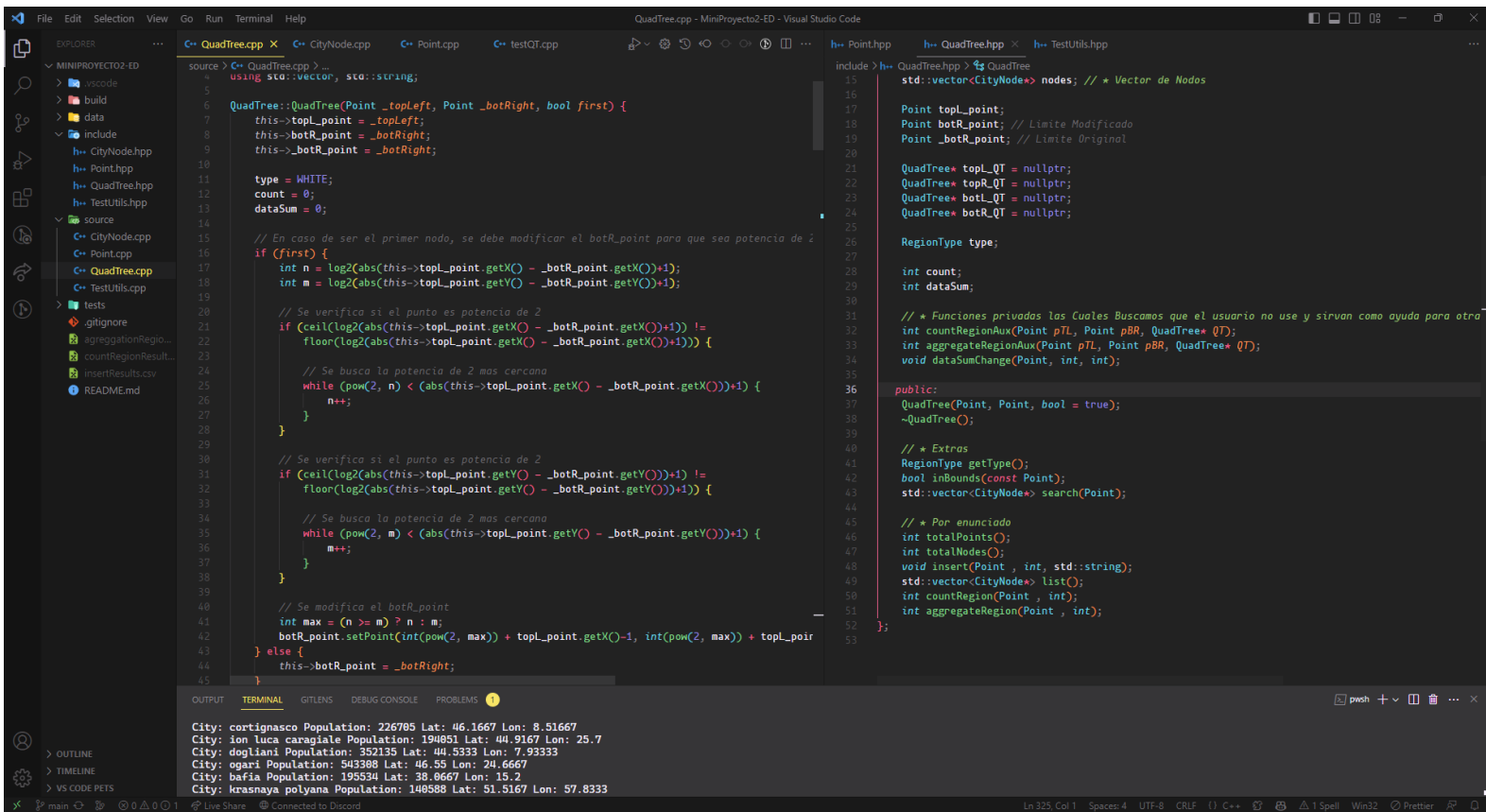


5. Conclusión

En conclusión, creemos que hemos podido implementar exitosamente la estructura de datos **QuadTree**, en la cual hicimos pruebas con una enorme cantidad de datos, pertenecientes al dataset “*World Cities Population*”, y logramos representar ciudades de un gran plano 2D de forma eficiente. De esta forma, nos creemos capaces de aprender e implementar cualquier estructura de datos que necesitemos en un futuro proyecto.

Además, de acuerdo con los resultados presentados, podemos concluir que en general que las estructuras de árboles, en este caso **QuadTree**, nos permiten abstraer y manejar grandes volúmenes de información (claves bidimensionales) de manera bastante rápida y eficiente, permitiéndonos tener velocidades de búsqueda $O(\log n)$ o $O(n \log n)$ dependiendo de los datos utilizados.

¡Muchas gracias por leer!



```
source > C++ QuadTree.cpp > ...
1 using std::vector, std::string;
2
3 QuadTree::QuadTree(Point _topLeft, Point _botRight, bool first) {
4     this->topL_point = _topLeft;
5     this->botR_point = _botRight;
6     this->botR_point = _botRight;
7
8     type = WHITE;
9     count = 0;
10    dataSum = 0;
11
12    // En caso de ser el primer nodo, se debe modificar el botR_point para que sea potencia de 2
13    if (first) {
14        int n = log2(abs(this->topL_point.getX() - _botR_point.getX())+1);
15        int m = log2(abs(this->topL_point.getY() - _botR_point.getY())+1);
16
17        // Se verifica si el punto es potencia de 2
18        if (ceil(log2(abs(this->topL_point.getX() - _botR_point.getX())+1)) !=
19            floor(log2(abs(this->topL_point.getX() - _botR_point.getX())+1)) {
20
21            // Se busca la potencia de 2 mas cercana
22            while (pow(2, n) < (abs(this->topL_point.getX() - _botR_point.getX())+1) {
23                n++;
24            }
25        }
26
27        // Se verifica si el punto es potencia de 2
28        if (ceil(log2(abs(this->topL_point.getY() - _botR_point.getY())+1)) !=
29            floor(log2(abs(this->topL_point.getY() - _botR_point.getY())+1)) {
30
31            // Se busca la potencia de 2 mas cercana
32            while (pow(2, m) < (abs(this->topL_point.getY() - _botR_point.getY())+1) {
33                m++;
34            }
35        }
36
37        // Se modifica el botR_point
38        int max = (n >= m) ? n : m;
39        botR_point.setPoint(int(pow(2, max)) + topL_point.getX()-1, int(pow(2, max)) + topL_point.getY()-1);
40    } else {
41        this->botR_point = _botRight;
42    }
43 }
```

```
include > h++ QuadTree.hpp > QuadTree
15
16
17 Point topL_point;
18 Point botR_point; // Limite Modificado
19 Point _botR_point; // Limite Original
20
21 QuadTree* topL_QT = nullptr;
22 QuadTree* topR_QT = nullptr;
23 QuadTree* botL_QT = nullptr;
24 QuadTree* botR_QT = nullptr;
25
26 RegionType type;
27
28 int count;
29 int dataSum;
30
31 // * Funciones privadas las cuales buscamos que el usuario no use y sirvan como ayuda para otra
32 int countRegionAux(Point pTL, Point pBR, QuadTree* QT);
33 int aggregateRegionAux(Point pTL, Point pBR, QuadTree* QT);
34 void dataSumChange(Point, int, int);
35
36 public:
37 QuadTree(Point, Point, bool = true);
38 ~QuadTree();
39
40 // * Extras
41 RegionType getType();
42 bool inBounds(const Point);
43 std::vector<CityNode*> search(Point);
44
45 // * Por enunciado
46 int totalPoints();
47 int totalNodes();
48 void insert(Point, int, std::string);
49 std::vector<CityNode*> list();
50 int countRegion(Point, int);
51 int aggregateRegion(Point, int);
52
53 }
```

```
City: cortignasco Population: 226785 Lat: 46.1667 Lon: 8.51667
City: ion luca caragiale Population: 194851 Lat: 44.9167 Lon: 25.7
City: dogliani Population: 352135 Lat: 44.5333 Lon: 7.93333
City: ogari Population: 543388 Lat: 46.55 Lon: 24.6667
City: bafia Population: 195534 Lat: 38.0667 Lon: 15.2
City: krasnaya polyana Population: 140588 Lat: 51.5167 Lon: 57.8333
```