

UNIVERSIDAD CATOLICA SAN PABLO - AREQUIPA  
FACULTAD DE CIENCIAS DE LA COMPUTACIÓN



## Estructura de Datos Avanzados

---

### Informe

### R - Tree

---

Advisor: Erick Mauricio Gomes Nieto  
Carlos Eduardo Atencio Torres

Student: Loayza Huarachi Angel Josue

AREQUIPA, MAY 2023





# Contents

<b>1</b>	<b>Introducción</b>	<b>4</b>
<b>2</b>	<b>Metodología y Logica</b>	<b>4</b>
2.1	Clase Point . . . . .	4
2.2	Clase Pokemon . . . . .	6
2.3	Clase Record . . . . .	7
2.4	Clase Node . . . . .	9
2.5	Clase Rtree . . . . .	25
<b>3</b>	<b>Resultados</b>	<b>27</b>
3.1	Usando el Data Set . . . . .	27
3.2	Seguimiento del codigo . . . . .	29
3.3	Comentarios finales . . . . .	39



## 1 Introducción

Dentro de una época como la nuestra, donde la administración eficiente de grandes conjuntos de datos son una gran necesidad para diversas aplicaciones, como sistemas de información geográfica, motores de búsqueda, etc.; una de las estructuras de datos más importantes que nos ayuda a resolver esta problemática es el R-Tree.

*“Requerimos de un mecanismo de indexación que nos permita acceder rápidamente a información de acuerdo a su localización espacial. El R-tree es muy buena para datos espaciales o espacios multidimensionales”*<sup>[1]</sup>

Para este trabajo, se presentará la implementación de dicha estructura en el lenguaje de programación C++. Donde detallaremos las clases y métodos usados para su respectivo desarrollo. El trabajo estará limitado a explicar la lógica de sus funciones, más no a explicar como se crea la estructura, pues su procedimiento ya se encuentra bien detallado en su libro<sup>[1]</sup>. Aun así, se compartirá el link del repositorio de GitHub para futuras pruebas y mejoras: <https://github.com/angel452/R-Tree-and-VTK>

## 2 Metodología y Logica

A continuación, se describirá como se llevó a cabo esta implementación, detallando las funciones, clases, técnicas, etc.

En el código presentado, contamos con 5 clases primordiales para recibir el “Data Set” de los Pokemones, como: La clase Point, la clase Pokemon, la clase Record, la clase Node, y la clase RTree.

### 2.1 Clase Point

La clase Point nos servirá para guardar la longitud de las dimensiones de los “records” obtenidos. Sin embargo, también le daremos un uso general como el de guardar puntos como el “BottomLeft” y el “UpperRight”, los cuales servirán para sacar el MBR (Mínimum Bouding Rectangles), o calcular el área de expansión para otras funciones.

```
1 // ##### CLASE PUNTO #####
2 // DIMENSIONES DEL POKEMON
3 template <int ndimension>
4 class Point{
5     private:
```

```
6     float datosPokemon[ndimension]; // height_m, hp,
7     weight_kg_std
8
9     public:
10    // ##### CONSTRUCTOR #####
11    Point(){
12        for(int i = 0; i < ndimension; i++){
13            datosPokemon[i] = 0;
14        }
15
16    Point( vector<float> _datosPokemon){
17        for(int i = 0; i < ndimension; i++){
18            datosPokemon[i] = _datosPokemon[i];
19        }
20    }
21
22    // ##### METODOS #####
23    float &operator[](int posicion){
24        return datosPokemon[posicion];
25    }
26};
```

### Variables privadas:

- datosPokemon: Array de tipo float que se usara para 2 motivos: Guardar las dimensiones de un dato/record (height\_m, hp, weight\_kg\_std); y también para guardar el bottomLeft y el UpperRight de dichos datos/records

### Constructores:

- Point(): Usado para asignar el valor 0 a todos los espacios guardados en el array “datosPokemon”.
- Point(vector <float>): Usado para asignar un valor específico, dado por el vector que le pasamos por parámetro, al array “datosPokemon”.

## Métodos:

- Operator: Usado para devolver el valor de una posición en específico, controlado por el parámetro “posición”.

## 2.2 Clase Pokemon

La clase Pokemon cumple en gran medida la misma función de la clase Point; en vez de guardar la longitud de sus dimensiones, esta guardará las coordenadas de los “récords”. Pues cada vez que queramos referirnos a un punto en el espacio tridimensional, usaremos de esta clase.

```
1 // ##### CLASE POKEMON #####
2 // COORDENADAS DEL POKEMON
3 template <int ndimension>
4 class Pokemon{
5     private:
6         float cordsPokemon[ndimension];
7
8     public:
9         // ##### CONSTRUCTOR #####
10        Pokemon(){
11            for(int i = 0; i < ndimension; i++){
12                cordsPokemon[i] = 0;
13            }
14        }
15
16        Pokemon( vector<float> _cordsPokemon ){
17            for(int i = 0; i < ndimension; i++){
18                cordsPokemon[i] = _cordsPokemon[i];
19            }
20        }
21
22        // ##### METODOS #####
23        float &operator[](int posicion){
24            return cordsPokemon[posicion];
25        }
26
27    }
```



26 };

### Variables privadas:

- cordsPokemon: Array de tipo float que se usará para guardar las coordenadas (x, y, z) de un dato.

### Constructores:

- Pokemon(): Usado para asignar el valor 0 a todos los espacios guardados en el array "cordsPokemon".
- Pokemon(vector <float>): Usado para asignar un valor específico, dado por el vector que le pasamos por parámetro, al array "cordsPokemon".

### Metodos:

- Operator: Usado para devolver el valor de una posición en específico, controlado por el parámetro “posición”.

## 2.3 Clase Record

La clase Record es una clase “general” que va a guardar e integrar a las 2 clases anteriores. Esta se encargará netamente de guardar todo tipo de información proveniente de un Pokemon; ya sea las coordenadas (usando la clase Pokemon), o la longitud de las dimensiones (usando la clase Point)

```

1 // ##### CLASE RECORD #####
2 // ALL DATA OF POKEMON
3 template< typename T, int ndim >
4 class Record {
5     private:
6         T object; // Coordenadas del pokemon
7         Point<ndim> tuple; // Dimensiones del pokemon
8
9     public:
10        // ##### CONSTRUCTOR #####
11        Record(float attack = 0, float defense = 0, float speed = 0,
12               float height_m = 0, float hp = 0, float weight_kg_std = 0){
```



```
12
13     // COORDENADAS
14     vector<float> auxCoordenadas;
15     auxCoordenadas.push_back(attack);
16     auxCoordenadas.push_back(defense);
17     auxCoordenadas.push_back(speed);
18     Pokemon<ndim> _object(auxCoordenadas);
19     object = _object;
20
21     // DIMENSIONES
22     vector<float> auxDimensiones;
23     auxDimensiones.push_back(height_m);
24     auxDimensiones.push_back(hp);
25     auxDimensiones.push_back(weight_kg_std);
26     Point<ndim> _tuple(auxDimensiones);
27     tuple = _tuple;
28 }
29
30 // ##### METODOS #####
31 T getObject(){
32     return object;
33 }
34
35 Point<ndim> getTupla(){
36     return tuple;
37 }
38 };
```

### Variables privadas:

- Objetc: Variable del tipo/clase Pokemon (T) donde le pasaremos las coordenadas (x, y, z) de un dato.
- Tuple: Variable del tipo/clase Point, donde le pasaremos la longitud de las dimensiones de un dato

### Constructores:



- Record(float, ...): Recibirá 6 datos del tipo float por parámetro (datos que sacaremos del “Data Set”) los cuales serán los que insertaremos. En este mismo constructor, daremos valores a las variables privadas ya mencionadas. Desde la 13 hasta la 19 para las coordenadas; y desde la 21 hasta la 27 para las longitudes de las dimensiones. Como podemos ver, la manera en que asignamos valores a estas variables son por medio de vectores auxiliares; pues como bien lo explicamos en la clase Pokemon y la clase Point, sus respectivos constructores reciben como parámetro un vector.

**Metodos:** Los métodos presentes en esta clase son únicamente para retornar las variables privadas; pues por su misma naturaleza, al ser privadas no podemos acceder a ellas directamente. Entonces, la manera en como accederemos a estos será por medio de dichas funciones de la siguiente manera:

```
Pokemon < ndim > coordsA = grupoPokemones[0].getObject()
```

```
Point < ndim > dimensionesA = grupoPokemones[0].getTupla()
```

- getObject: Retorna la variable privada Object de tipo Pokemon.
- getTupla: Retorna la variable privada Tuple de tipo Point.

## 2.4 Clase Node

La clase Node es de las más importantes, pues se encargará de darle el cuerpo a esta estructura de datos. Nos ayudará a formar el “árbol” que estaremos creando con los hijos creados a partir de sus funciones como: Insert, split, etc.; como también guardar información necesaria para cada nodo, desde la raíz hasta el último nodo hoja. Como por ejemplo: un contador de “records” guardados, el máximo de “records” por nodo, etc.

```

1 // ##### CLASE NODO #####
2 template< typename T, int ndim >
3 class Node {
4     private:
5         vector< Record<T,ndim> > grupoPokemones;      // Vector of
records

```



```
6     vector< Node<T,ndim> > grupoHijos;           // Vector of
7     childrens type Node
8         Point<ndim> bottomLeft;                  // Point x,y,z
9         Point<ndim> upperRight;                 // Point x,y,z
10        int maxRecords;                         // Limit of
11        records per node
12        int countRecords;                      // Data counter
13        per node
14        bool isLeaf;                           // Flag
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
```

```
childrens type Node
Point<ndim> bottomLeft;
Point<ndim> upperRight;
int maxRecords;
records per node
int countRecords;
per node
bool isLeaf;
```

```
public:
// ##### CONSTRUCTOR #####
Node(){
    maxRecords = 0;
    countRecords = 0;
    isLeaf = true;

    bottomLeft = Point<ndim>();
    upperRight = Point<ndim>();
}

Node(int _maxhijos){
    maxRecords = _maxhijos;
    countRecords = 0;
    isLeaf = true;

    bottomLeft = Point<ndim>();
    upperRight = Point<ndim>();
}

// ##### METODOS #####
// ----- Functions to print -----
void printRecord(Node*pntN);
```



```
38     void printRecord2(Record<T, ndim> record);
39
40     void printBottomLeft(Node *pntN);
41
42     void printUpperRight( Node *pntN );
43
44 // ----- Functions to recalculate and get MBR
45 -----
45     Point<ndim> recalculateBottomLeft(Node *pntN);
46
46     Point<ndim> recalculateUpperRight(Node *pntN);
47
48
49     pair< Point<ndim>, Point<ndim> > get_BL_UR_Record( Record<T,
50     ndim> record );
51
51 // ----- Functions to PeekSeeds -----
52     float calculateArea(Point<ndim> bottomLeft, Point<ndim>
53     upperRight);
54
54     pair< Point<ndim>, Point<ndim> > getDimensionsJ( Record<T,
55     ndim> E1I, Record<T, ndim> E2I);
56
56     pair< Point<ndim>, Point<ndim> > getDimensionsE( Record<T,
57     ndim> E_NI);
58
58 // ----- Functions to PeekNext -----
59     float calculateEnlargment( Record<T, ndim> &record, Node
60     *hijo);
61
61 // ----- Functions for Cuadratic Split -----
62     pair< Node, Node > peekSeeds_nextPeek( Node *pntN );
63
63     pair< Node, Node > splitCuadratico(Node *pntN);
64
64 // ----- Functions for Equidistant Split -----
65
```



```
67     float distance2Pokemons( Pokemon<ndim> pntBottomLeft,
68         Point<ndim> pntUpperRighth );
69
70
71     pair< Node, Node > splitEquisdistantes(Node *pntN);
72
73
74     // ----- INSERT FUNCTION -----
75
76     bool insertNode(Record<T,ndim> &record, Node *pntN);
77
78
79     // ----- DELETE FUNCTION -----
80
81
82     // ----- SEARCH FUNCTION -----
83
84
85     // ----- PRINT VTK FUNCTION -----
86
87     void getAllSizesNode(Node *pntN);
88
89
90 };
```

### Variables privadas:

- grupoPokemones: Vector que guarda variables del tipo/clase Record. En este vector estaríamos guardando todos los datos que le pertenecen a un nodo específico.

De esta manera es tan simple insertar un dato dentro de un nodo como:

```
pntN->grupoPokemones.push_back(record)
```

- grupoHijos: Vector que guarda variables del tipo/clase Nodo. Este vector se estaría usando sola y únicamente cuando ejecutamos un “Split()”, pues es señal de que a partir de ahora, dicho nodo tendrá hijos. Por eso, es que este vector va a guardar datos también de tipo Nodo, ya que existe la posibilidad de que estos hijos también tengan más hijos, y así creamos el árbol esperado y característico del R-Tree.

En lo posterior se explicará como y que retorna la función split, pero al igual que la variable “grupoPokemones”, la ventaja de usar “vector” es que podremos añadirlo de la siguiente manera:

```
pntN->grupoHijos.push_back(auxSplit.first)
```

```
pntN->grupoHijos.push_back(auxSplit.second)
```

- bottomLeft: Variable del tipo/clase Point que guardará el punto izquierdo-abajo del



“record”. Para obtener estos 2 puntos (si fuera un espacio 2D), bastaría únicamente con asignarle las coordenadas de un record, las cuales se encuentran guardadas en la clase Pokemon.

- `upperRight`: Variable del tipo/clase Point que guarda el punto derecha-arriba del “record”. Estos puntos se obtiene mediante la siguiente operación:

$$\begin{aligned} \text{upperRight}[0] &= \text{coordenada}[0] + \text{crecimiento}[0] \\ \text{upperRight}[1] &= \text{coordenada}[1] + \text{crecimiento}[1] \\ \text{upperRight}[2] &= \text{coordenada}[2] + \text{crecimiento}[2] \end{aligned}$$

Para comprender tanto el `bottomLeft` como el `upperRight`, nos podemos guiar por el siguiente ejemplo:

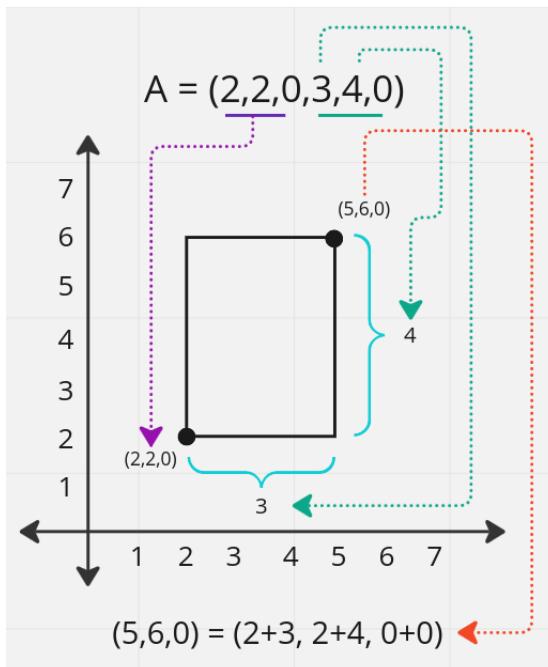


Figure 2.1: Ejemplo de "Record" en un espacio 2D

- `maxRecords`: Número máximo de “Records” que podemos guardar en un nodo.
- `countRecords`: Contador de “Records” por nodo.
- `isLeaf`: Variable booleana que nos permite identificar si un nodo es un nodo intermedio - raíz o un nodo hoja.



### Constructores:

- Node(): Inicializa todas las variables anteriormente vistas por 0 o puntos vacíos, a excepción del booleano isleaf, pues cuando se crea un nuevo nodo, por predeterminado, será un nodo hoja.
- Node(maxhijos): Inicializa de la misma manera que el anterior constructor, solo que ahora le pasamos la variable `_maxhijos` para guardar el número máximo de datos que un nodo puede llevar.

**Métodos:** Dentro de todas las funciones que se implementó, las clasificamos de la siguiente manera: Funciones de impresión, funciones para recalcular y obtener el MBR, Funciones para el PeekSeeds, funciones para el PeekNext, funciones para el “Cuadratic Split”, funciones para el "Split con Equidistantes", función de Insert, y la función de apoyo para graficar en VTK.

- Funciones de impresión: En el código compartido, estas funciones están comentadas; si se desea ver el paso a paso de como van cambiando estos valores tras ciertos inserts, podría des-comentarlos y hacerle un seguimiento.
  - printRecord: Esta función imprime en consola todos los records/datos guardados dentro de un nodo. Por eso es que se pide, como parámetro, un puntero que apunte al nodo que queremos ver su contenido.
  - printRecord2: Esta función, a diferencia de la anterior, imprime en consola únicamente la información que está guardando la variable del tipo Record. Por eso, el parámetro que se pide es de ese tipo.
  - printBottomLeft: Esta función imprime en consola el conjunto de puntos guardado dentro de la variable bottomLeft de un nodo en específico. Por ello, es que se pide, en el parámetro, un puntero a ese nodo.
  - printUpperRight: Esta función imprime en consola el conjunto de puntos guardado dentro de la variable upperRight de un nodo en específico. Tiene el mismo parámetro que de la función anterior.
- Funciones para recalcular y obtener el MBR: Las 2 primeras funciones correspondientes se llaman tras la inserción de un nuevo dato, pues luego de añadirlo dentro de nuestro vector “grupoPokemones”, es necesario recalcular el bottomLeft y el UpperRight para que este contenga a los anteriores nodos junto con el nuevo. Técnicamente, sería como actualizar el MBR.



- recalculateBottomLeft y recalculateUpperRight: Recalculan el bottomLeft y el upperRight del nodo actual.

Lógica:

Para este ejemplo tenemos 2 datos (A y B) en un espacio bidimensional mostrados por sus BottomLeft y UpperRight. A continuación, mostraremos el procedimiento de como el algoritmo calcula del MBR de estos 2 datos.

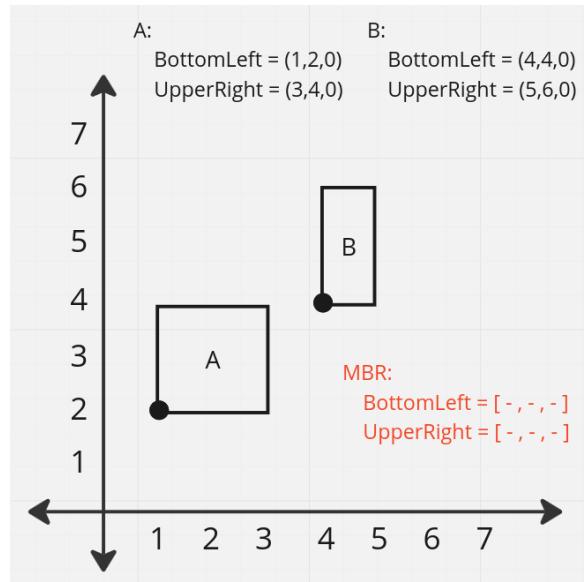


Figure 2.2: Ejemplo - Recalcular BottomLeft y UpperRight

1. Paso 1: Seleccionamos el BottomLeft[0] de A y de B; y colocamos dentro del BottomLeft[0] del MBR el menor de estos. En este caso, el número 1:

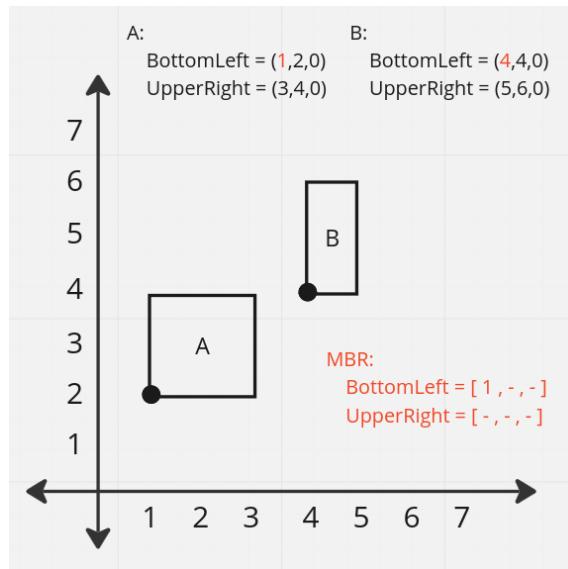


Figure 2.3: Ejemplo - Recalcular BottomLeft y UpperRight 1

2. Paso 2: Ahora, seleccionamos el  $\text{UpperRight}[0]$  de A y de B; y colocamos dentro del  $\text{UpperRight}[0]$  del MBR el mayor de estos. En este caso el número 5:

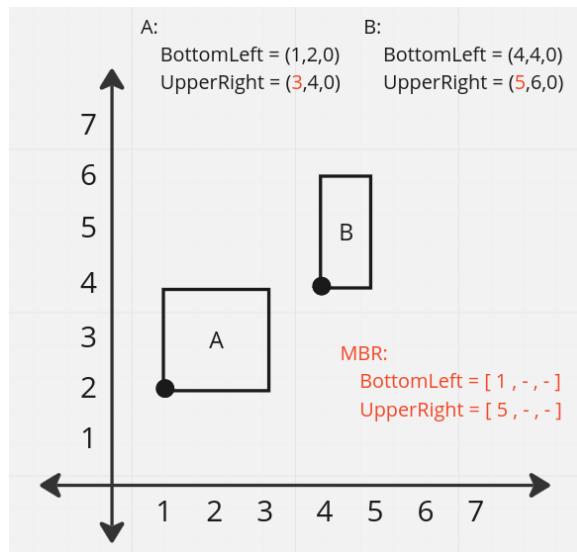


Figure 2.4: Ejemplo - Recalcular BottomLeft y UpperRight 2

3. Paso 3: Repetimos desde el paso 1, pero en esta ocasión ya no en las posiciones [0], sino, ahora será en las posiciones [1], y luego en la [2], y así sucesivamente hasta que culminen todas las dimensiones que tenemos. Quedando al final de la siguiente manera:

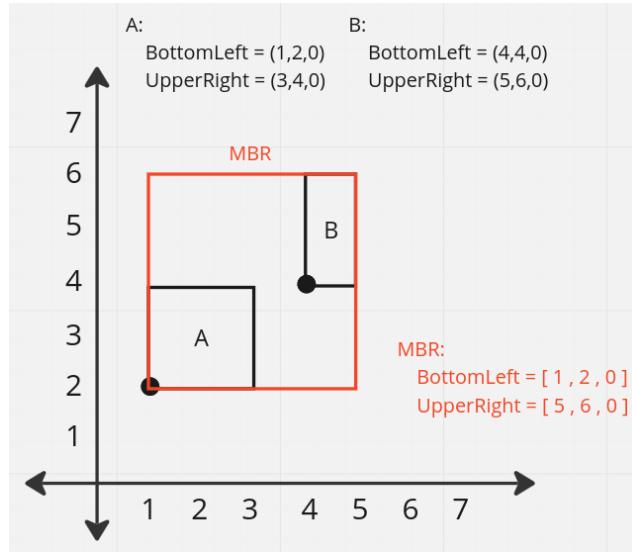


Figure 2.5: Ejemplo - Recalcular BottomLeft y UpperRight Final

- `get_BL_UR_Record`: Es una función de apoyo que me permite guardar en una estructura `pair< Point, Point >` los `bottomLeft` y el `UpperRight` de un Record en específico. Si vemos el código, usamos justamente las funciones `getObject` y `getTupla` propias de la clase `Record`, que ya vimos.
- Funciones para el “Cuadratic Split”:
  - `splitCuadratico`: Esta función se ejecuta cuando insertamos un nuevo dato dentro de un nodo que está completamente lleno (superá la granularidad). Tras su ejecución, se espera que todos los datos almacenados en el padre, se distribuyan en alguno de estos 2 nodos (L y LL) que retornará esta función.  
Podemos usar la técnica de los equidistantes, o la técnica que usa funciones como `peekSeeds` y `PeekNext` (en este caso, se usó este último método).  
Tal como indica el libro "R-trees: a dynamic index structure for spatial searching" [1], esta función retornará dichos 2 nodos mediante la estructura “pair”, los cuales contendrán todos los datos del nodo padre, distribuidos lógicamente.  
(Ver Algorithm 1: `splitCuadratico`)



```
1 Input: Node *pntN
2 Output: pair <Node , Node> resultSplit
3 pair <Node, Node > resultPeekSeeds = peekSeeds_nextPeek(pntN)
4 L = resultPeekSeeds.first
5 LL = resultPeekSeeds.second
6 resultSplit.first = L
7 resultSplit.second = LL
```

**Algorithm 1:** splitCuadratico

- peekSeeds\_nextPeek: Como vimos en el Algoritmo 1: splitCuadratico, esta función se llama en la línea 3, y hace en una sola las 2 funciones que Guttman recomienda: PeekSeeds y NextPeek.
  - \* En la primera parte (PeekSeeds), nos encargamos de buscar los 2 datos más lejanos para convertirlas en nuestras semillas; se usó la lógica de la típica implementación para buscar el menor dato dentro de un array (Ver Algorithm 2: sortArray). Pero esta vez, orientada a buscar el máximo desperdicio que genera el MBR de 2 datos.

Recordemos que para el PeekSeeds usamos esta fórmula:

$$d = \text{Area}(J) - \text{Area}(E_1I) - \text{Area}(E_2I)$$

Donde Área(J) es el área que se crea al juntar en un MBR un par de nodos E1 y E2; el Área(E\_1I) es el área que genera el dato E1; y Área(E\_2I) es el área que genera el dato E2. (Ver “Algorithm 3: peekSeeds”).

```
1 Input: arr[]
2 Output: float menorDato
3 minDato = arr[0]
4 for i = 1 : arr.size() do
5   | minDatoAux = arr[i]
6   | if minDatoAux < minDato then
7   |   | minDato = minDatoAux
8   | end
9 end
```

**Algorithm 2:** sortArray



```

1 Input: Node *pntN
2 Output: pair < Node, Node > resultPeekSeeds_NextPeek
3 posSemilla1 = 0
4 posSemilla2 = 1
5 areaJ = calculateArea(...)
6 areaE1 = calculateArea(...)
7 areaE2 = calculateArea(...)
8 d = areaJ - areaE1 - areaE2
9 j = 1
10 for i = 0 : countRecords do
11   for j : countRecords do
12     areaJAux = calculateArea(...)
13     areaE1Aux = calculateArea(...)
14     areaE2Aux = calculateArea(...)
15     dAux = areaJAux - areaE1 - areaE2
16     if dAux < d then
17       posSemilla1 = i
18       posSemilla2 = j
19     end
20   end
21   j = i + 2
22 end

```

### Algorithm 3: peekSeeds

Hasta este punto, tendríamos en las variables “posSemilla1” y “posSemilla2” las posiciones de los datos más lejanos; simplemente tocaría ahora insertarlos a estos, en nodos distintos de la siguiente manera:

<i>insertNode(grupoPokemones[posSemilla1], pntNL)</i>
-------------------------------------------------------

<i>insertNode(grupoPokemones[posSemilla2], pntNLL)</i>
--------------------------------------------------------

Como podemos ver, primero estamos insertando el Record ubicado en la posición [posSemilla1] del vector grupoPokemones donde el puntero pntNL apunta, es decir, al nodo hijo L; y en el segundo, insertamos el Record ubicado en la posición [posSemilla2] del mismo vector, en el nodo LL, ya que el puntero pntNLL apunta ahí.

- \* Para la segunda parte, hacemos el NextPeek. Su procedimiento es más sencillo: seleccionamos todos los datos restantes (todos los datos diferentes a las semillas) y calculamos cuanto “Enlargment” tendría el dato si se lo añadiera en el nodo L o en el nodo LL; obviamente, seleccionamos el nodo



con menor área de expansión, pues ahí mismo es donde será insertado. (Ver "Algorithm 4: NextPeek")

```
1 Input: Node *pntN
2 Output: pair <Node, Node> resultPeekSeeds_NextPeek
3 for i = 0 : countRecords do
4     crecimiento1 = calculateEnlargment(record[i], hijo1)
5     crecimiento2 = calculateEnlargment(record[i], hijo2)
6     if crecimiento1 < crecimiento2 then
7         | insertNode(record[i], hijo1)
8     end
9     else
10        | insertNode(record[i], hijo2)
11    end
12 end
```

**Algorithm 4:** NextPeek

- Funciones para el PeekSeeds: Estas funciones son llamadas cuando entramos a la función peekSeeds\_nextPeek; estas nos ayudaran en todo este cálculo matemático que tenga que ver con áreas, distancias, etc.

A continuación explicaremos el objetivo de cada una de estas.

- calculateArea: Esta función recibe como parámetro 2 variables del tipo Point, que no son nada más que el bottomLeft y el UpperRight de un nodo. Mediante cálculos matemáticos no muy complicados, obtenemos el área que crean estos puntos.
- getDimensionsJ: Esta función toma como parámetros 2 variables del tipo Record, pues recordemos que tenemos que calcular el Área(J), y requerimos necesariamente de un bottomLeft y un upperRight del MBR que crean dichos Records. Por eso es que esta función retorna una estructura del tipo pair <Point, Point>.

Mediante esta estructura, retornaremos en un Point el nuevo bottomLeft del MBR; y en el otro, el upperRight también del mismo MBR.

La lógica usada es idéntica a las funciones recalculateBottomLeft y recalculateUpperRight. La única diferencia es que esta calcula el MBR de 2 Records, y no de todos los Records de un nodo, como en dichas 2 funciones; técnicamente solo cambio el parámetro de la función.



- `getDimensionsE`: Esta función es más simple, pues no requerimos hacer una operación adicional; simplemente tenemos que retornar el `bottomLeft` y el `UpperRight` de un dato en la misma estructura `pair < Point, Point >`; para esto, nos apoyamos de la función `get_BL_UR_Record`.

- Funciones de `PeekNext`:

- `calculateEnlargment`: Esta función también es llamada cuando entramos a la función `peekSeeds_nextPeek`, específicamente la podemos ver en la línea 4 y 5 del Algoritmo 4: `NextPeek`.

Lo que hace es calcular el crecimiento que tendría un nodo si aceptara como suyo a un dato.

- Funciones para el “Split con Equidistantes”: Esta técnica es diferente a la que Guttman menciona<sup>[1]</sup>. Tiene una implementación más sencilla que la anterior; y para su desarrollo solo necesitamos de 2 funciones:

- `splitEquisdistantes`: Al igual que el anterior `Split`, esta también se divide en 2 partes:

- \* La primera parte es de obtener equidistantes. “*Se dice que un punto es equidistante de un conjunto de figuras geométricas si las distancias entre ese punto y cada figura del conjunto son iguales*”.

Ya que estos rectángulos/cubos están en distintos lugares, incluso muy lejos de los ejes ( $x$ ,  $y$ ,  $z$ ); determiné que la forma más fácil y que no requiera hacer muchas operaciones era jugar con promedios. Así, no importa donde se ubique, siempre el promedio entre 2 puntos me dará la posición central en el eje específico que le dé. (Ver Algorithm 5: `getEquidistantes`).

```

1 Input: Node *pntN
2 Output: pair < Node, Node > resultSplit
3 for i = 0 : ndimensiones do
4   mitad = promedio(bottomLeft[i], upperRight[i])
5   equidistante1 = promedio(bottomLeft[i], mitad)
6   equidistante2 = promedio(mitad, upperRight[i])
7 end
```

**Algorithm 5:** `getEquidistantes`

Como podemos ver, hacemos un recorrido de todos los ejes disponibles y sacamos 3 datos: variable `mitad` (promedio entre el `bottomLeft` y el



UpperRight); luego la variable equidistante1 (promedio entre el bottomLeft y la mitad); y por último la variable equidistante2 (promedio entre la mitad y el upperRight)

- \* Para la segunda parte, procedemos con el “Reinsert” de los datos. Esta parte es de las más simples, pues recorremos todos los records y calculamos la distancia entre las coordenadas de un record y el equidistante 1 y 2; luego lo insertamos en el que más cerca se encuentre. (Ver Algorithm 6: ReInsert)

```
1 Input: Node *pntN
2 Output: pair <Node, Node> resultSplit
3 for i = 0 : countRecords do
4     crecimiento1 = distance2Pokemons(record[i], equidistante1)
5     crecimiento2 = distance2Pokemons(record[i], equidistante2)
6     if crecimiento1 < crecimiento2 then
7         | insertNode(record[i], hijo1)
8     end
9     else
10        | insertNode(record[i], hijo2)
11    end
12 end
```

**Algorithm 6:** ReInsert

- distance2Pokemons: Tiene el objetivo de calcular la distancia entre 2 puntos/coordenadas del Record que estamos tratando. Usamos la fórmula matemática “Distancia entre 2 puntos”:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Pero como en este caso tratamos un espacio tridimensional, se modifica a la siguiente manera:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

(Ver Algorithm 7: distance2Pokemon).



```
1 Input: Pokemon data1, Point data2
2 Output: float distance
3 distance = 0
4 for i = 0 : ndimensiones do
5   | aux = data2[i] - data1[i]
6   | distance = distance + pow(aux, 2)
7 end
8 distance = sqrt(distance)
```

**Algorithm 7:** distance2Pokemon

- Funcion de Insert:

- InsertNode: La función InsertNode es la que engloba a todas las funciones anteriores para conseguir adjuntar nuevos datos a nuestro árbol y poder hacer Split's o cálculos de Enlargment sobre la marcha. El siguiente pseudocódigo explica la lógica de como se usó la recursividad para poder navegar en el árbol tras cada nuevo nodo.



```
1 Input: Record dato, Node *pntN
2 Output: true or false
3 if pntN->isLeaf then
4   if countRecords < maxRecords then
5     // Normal Insert
6     countRerords++
7     groupPokemones.push_back(dato)
8     recalculateMBR()
9   end
10  else
11    // Normal Insert
12    countRerords++
13    groupPokemones.push_back(dato)
14    recalculateMBR()
15    // Split
16    splitCuadratico(pntN) OR splitEquidistante(pntN)
17    grupoHijos.push_back(split.first)
18    grupoHijos.push_back(split.second)
19  end
20 end
21 else
22   bool flag = dato.isInSomeChild()
23   if flag == true then
24     | insert(record, pntNAux)
25   end
26   else
27     | chooseLeaf()
28   end
29 end
```

**Algorithm 8:** InsertNode

- Función para graficar en VTK
  - getAllSizesNode: Esta función es muy parecida al search() de un árbol binario. De por sí, la función es recursiva, teniendo como “caso base” el momento donde el puntero visita un nodo hoja; esto indica que llegamos al límite del árbol y tenemos que pasar al siguiente hijo. La gracia de esta función es que vamos recolectando en una “variable global” llamada “allSizes” todos los bottomLeft y lo UpperRight de cada nodo que visita, pues estos al final serán los usados para graficar con VTK. El pseudocódigo es el siguiente:



```

1 Input: Node *pntN
2 Output: -
3 if pntN.isleaf == true then
4   | allSizes.push_back(bottomLeft and upperRight)
5 end
6 else
7   | allSizes.push_back(bottomLeft and upperRight)
8   | for i : pntN.countRecords do
9     |   | pntNAux = pntN.gruposHijos[i];
10    |   | getAllSizesNode(pntAux);
11   | end
12 end

```

**Algorithm 9:** getAllSizesNode

También quisiera comentar que la forma en que se imprime en consola tras ejecutar esta función se interpreta de la siguiente manera:

En primer lugar, aparecerán 6 números seguidos por guiones. Los 3 primeros son las coordenadas del “Record” y los otros 3 son las dimensiones que un eje se alarga. Para entenderlo mucho mejor pueden ver la Figura 2.1: Ejemplo de “Record” en un espacio 2D.

## 2.5 Clase Rtree

La clase R-Tree será la clase que permitirá llamar al insert de la clase Node, y será el encargado de crear el primer nodo de todos, el root. Además, también creará su puntero respectivo como lo veremos a continuación.

```

1 // ##### CLASE RTREE #####
2 template< typename T, int ndim >
3 class RTree {
4     private:
5         Node<T,ndim> root;
6         Node<T, ndim> *pntN;
7
8     public:
9         RTree(int _maxHijos){
10             Node<T,ndim> aux(_maxHijos);
11             root = aux;

```



```
12
13     pntN = &root;
14 }
15 void insert(Record<T,ndim> &rec ){
16     // INSERTAR
17     root.insertNode(rec, pntN);
18 }
19
20 void getAllSizes(){
21     root.getAllSizesNode(pntN);
22 }
23 };
```

### Variables privadas:

- root: Variable que inicializa y crea el primer nodo del árbol y de la estructura: el root o la raíz; la variable es de tipo Node, pues en este se guardarán nuevos records en el vector grupoPokemones, o también nuevos hijos en caso pase un Split, entre otros atributos propios de la clase Node ya explicados.
- \*pntN: También es una variable del tipo Node, pero a diferencia del root, esta será un puntero. Esta variable será la que nos ayude a navegar entre distintos nodos, e incluso funciones, pues será el que pasemos por parámetro en la mayoría de métodos usados en la clase Node.

### Constructores:

- RTree(maxHijos): Este constructor está relacionado con constructor de la clase Node, pues únicamente recibimos una variable de tipo entero MaxHijos, que indica la cantidad máxima de datos que puede contener un nodo. Por eso es que al root le pasamos ese dato en la línea 11.  
Adicionalmente, aquí es donde le indicaremos al puntero N, que debe apuntar al nodo root desde un comienzo. Por ello igualamos el puntero con la dirección de memoria de root.

**Métodos:** Las 2 funciones presentes, no realizan alguna modificación dentro del árbol; sino, simplemente actúan como “links directos” a las funciones “importantes” de la clase Nodo.



- **insert:** La función insert de la clase R-Tree recibe como parámetro un Record (todos los datos del Pokemon proporcionados por la Data Set). Su objetivo será simplemente llamar a la función insert de la clase Node, pasándole a esta dicho Record, y el puntero a nodo donde lo insertaremos, en este caso, como es el primero, al root.
- **getAllSizes:** Esta función se ejecutará una vez que todos datos del “Data Set” se hayan insertado, es decir, una vez que el árbol ya este creado y no se haga alguna modificación adicional. Esta función actúa como un simple “intermediario” que llama a la función getAllSizesNode de la clase Node, pasándole por parámetro el puntero al root, para que empiece a navegar por todos los hijos y a todos sus nodos hoja.

## 3 Resultados

### 3.1 Usando el Data Set

En las siguientes imágenes mostraremos capturas de como resultó la estructura de datos R-Tree con diferentes granularidades (maxRecords), frente al Data Set proporcionado.

En esta ocasión probaremos con granularidades de 10, 50, 100 y 400.

- Granularidad 10

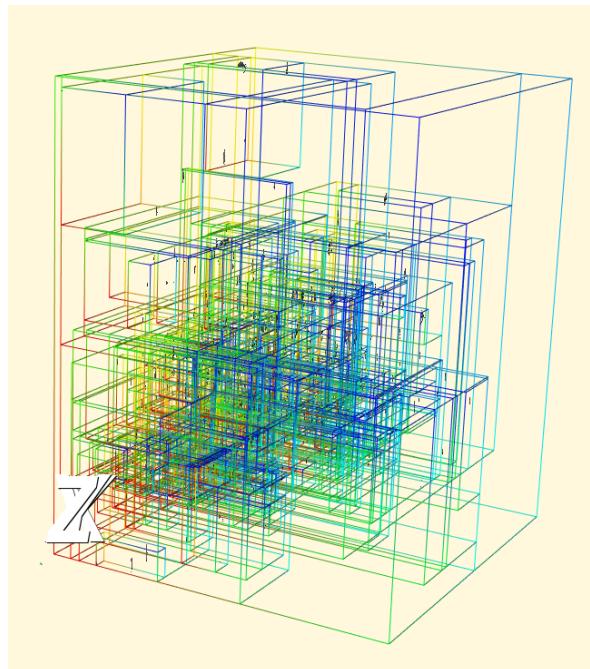


Figure 3.1: Rtree - Granularidad 10



- Granularidad 50

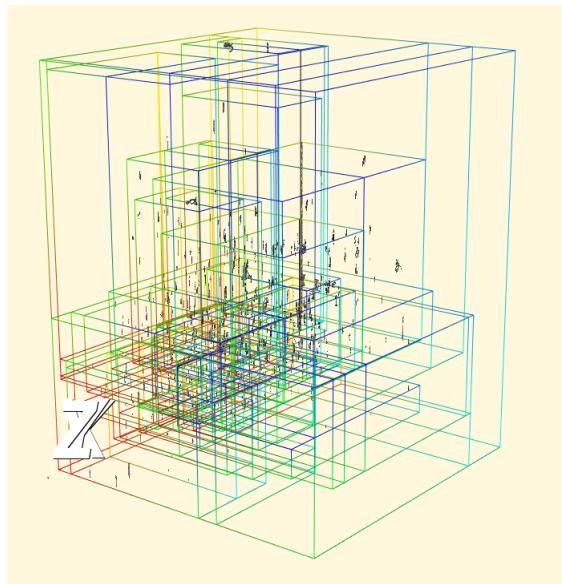


Figure 3.2: Rtree - Granularidad 50

- Granularidad 100

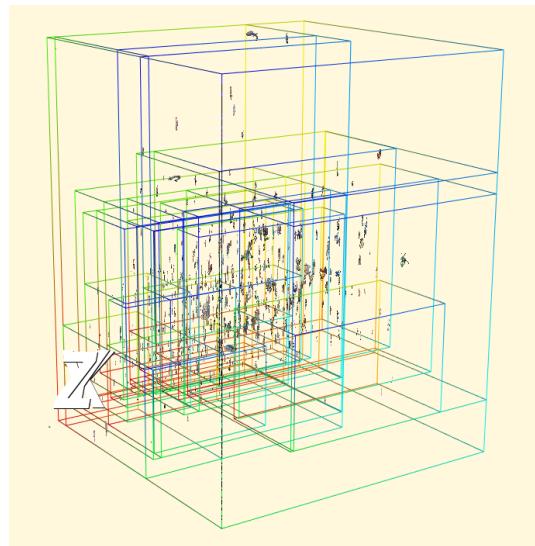


Figure 3.3: Rtree - Granularidad 100



- Granularidad 400

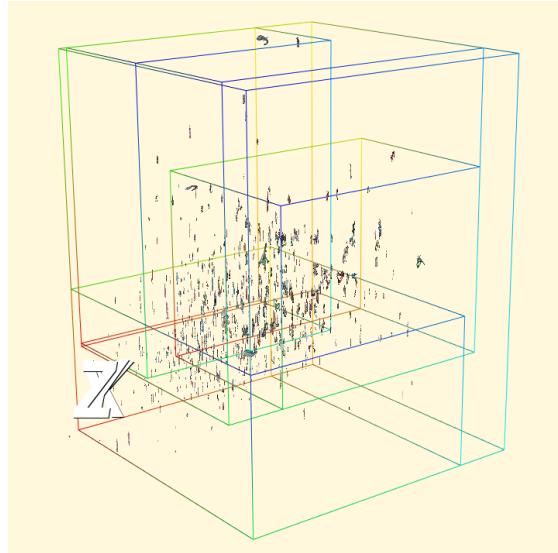


Figure 3.4: Rtree - Granularidad 400

### 3.2 Seguimiento del código

Para esta última parte, mostraremos el paso a paso cuando insertamos 10 datos uno por uno, para ver el comportamiento del nodo root, de los hijos, el MBR, y la granularidad. La granularidad será de máximo 3 datos por nodo; y serán los siguientes:

- Dato A: (2, 2, 0, 2, 2, 1)
- Dato B: (7, 9, 0, 2, 2, 1)
- Dato C: (9, 6, 0, 3, 2, 1)
- Dato D: (1, 7, 0, 2, 3, 1)
- Dato E: (2, 5, 0, 1, 1, 1)
- Dato F: (9, 2, 0, 2, 3, 1)
- Dato G: (2, 12, 0, 3, 3, 1)
- Dato H: (5, 11, 0, 1, 1, 1)
- Dato I: (1, 16, 0, 3, 3, 1)
- Dato J: (5, 17, 0, 3, 3, 1)



Para mejorar la visibilidad y apreciación de la inserción de los datos, estamos colocando en valor 0 todas las posiciones número [2], eso indica que las coordenadas de todos los datos serán únicamente en “X” e “Y”; y la posición [5] para también todos los datos será de 1, para que no haya problemas cuando calculemos el área de expansión.

### Seguimiento:

1. Paso 1: Insertamos el Dato A (Bulbasaur).

Como vemos en la consola, podemos obtener el MBR del único dato presente



Figure 3.5: Insert - Seguimiento 1



## 2. Paso 2: Insertamos el Dato B (Ivysaur).

No hay problema con la inserción, pues el nodo aún puede guardar más datos (máximo 3). Solo vemos que el MBR se actualiza y contiene a ambos.

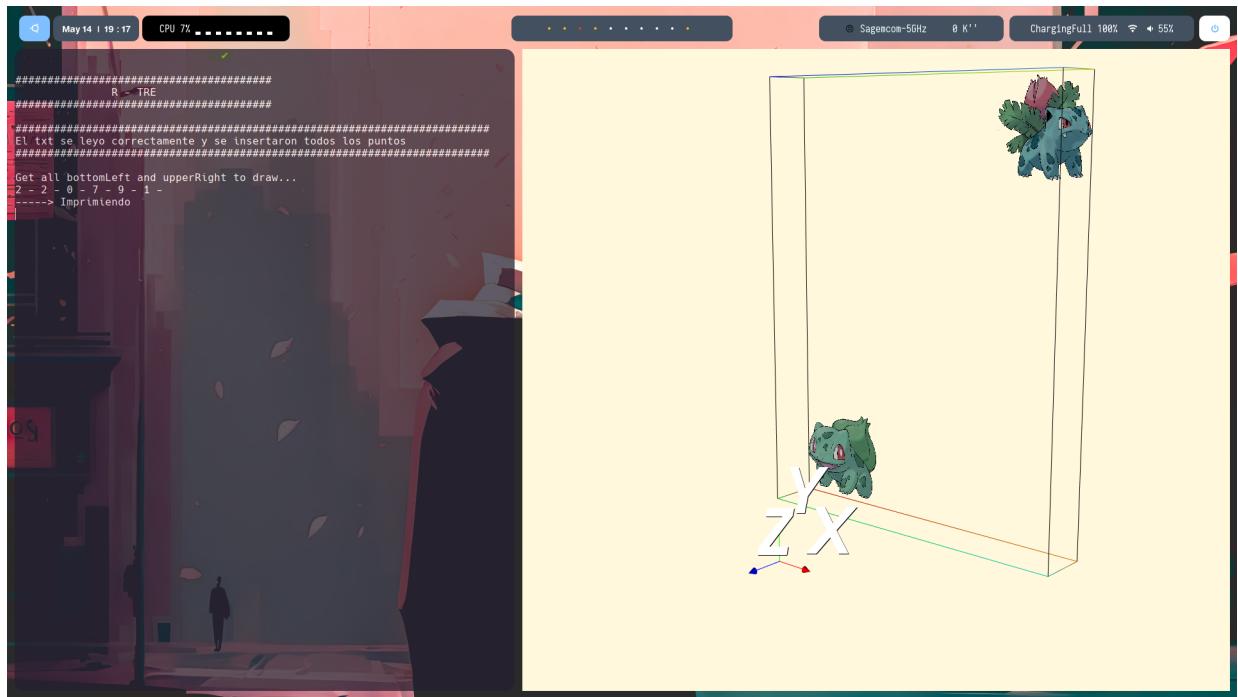


Figure 3.6: Insert - Seguimiento 2



3. Paso 3: Insertamos el Dato C (Venusaur). Tampoco hay problema con la inserción.  
El MBR nuevamente se actualiza.



Figure 3.7: Insert - Seguimiento 3



4. Paso 4: Insertamos el Dato D (Charmander). Como supera la granularidad, llamamos al split; el algoritmo seleccionara a los 2 datos más lejanos, en este caso al A - Bulbasaur y B - Ivysaur, y reinsertara a los otros 2 en el que más cercano se encuentre, creando así los 2 hijos que vemos en la imagen.

Se actualiza el MBR del padre, y también mostramos el MBR de los 2 hijos. Ello también se puede presenciar en lo impreso en la consola; como vemos, imprime primero del padre, y luego de los 2 hijos.

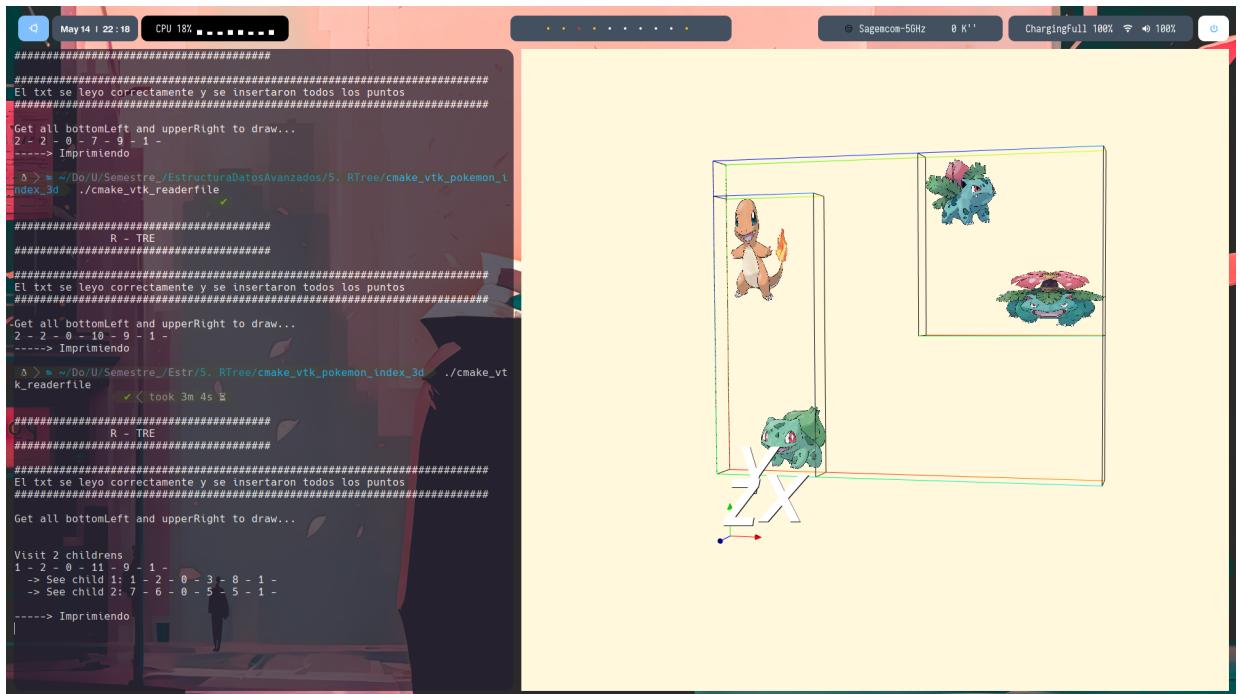


Figure 3.8: Insert - Seguimiento 4

5. Paso 5: Insertamos el Dato E (Charmeleon). Como ahora el padre ya no es un nodo hoja, tiene que visitar a sus 2 hijos. Pero primero verifica si este se encuentra totalmente dentro de uno de los MBR de los hijos. En este caso, si lo está, entonces nos ahorraremos operaciones de búsqueda y lo insertamos donde pertenece. No hay problema en ello, pues dicho hijo no supera la granularidad

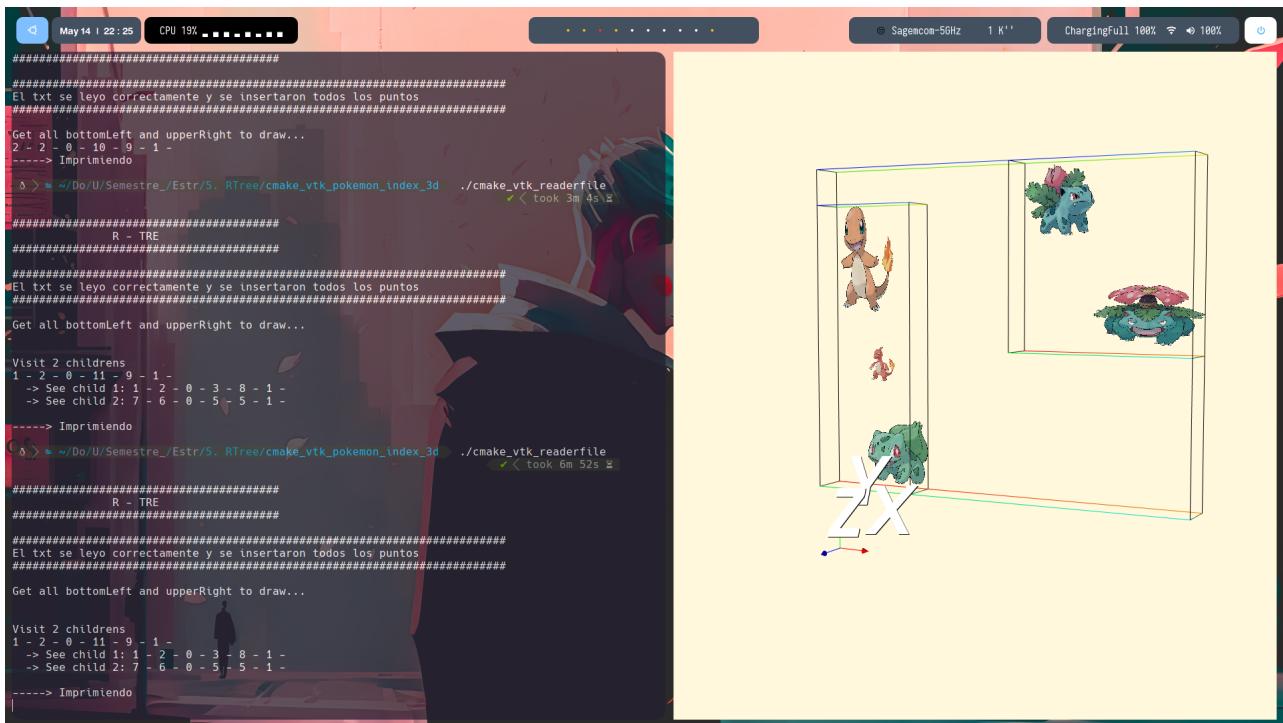


Figure 3.9: Insert - Seguimiento 5



6. Paso 6: Insertamos el Dato F (Charizard). Para este dato es el caso contrario de cuando insertamos al dato E - Charmeleon. Aquí, el nuevo dato no está completamente dentro de uno de los hijos, entonces tenemos que ejecutar el ChooseLeaf para ver cuál de los hijos tiene menos expansión en caso de insertarlo; para este caso es el hijo 2, que tienen al dato B - Ivysaur y C - Venusaur. Procedería a insertarlo y recalcular el MBR.

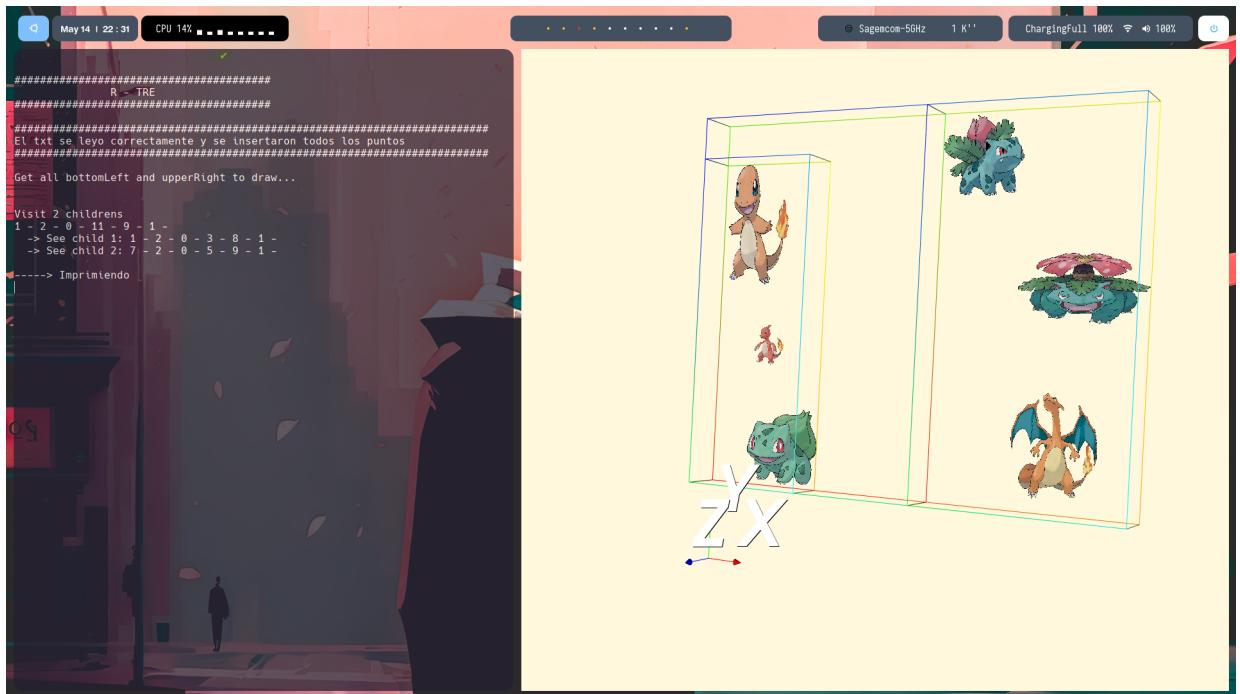


Figure 3.10: Insert - Seguimiento 6



7. Paso 7: Insertamos el Dato G (Squirtle). Nuevamente, caemos en un caso similar al paso 4, cuando recorremos los hijos y nos enteramos de que tenemos que insertarlo en un hijo que supera la granularidad, requerimos de hacer un split, pero esta vez al hijo correspondiente, luego continuamos con el mismo proceso.

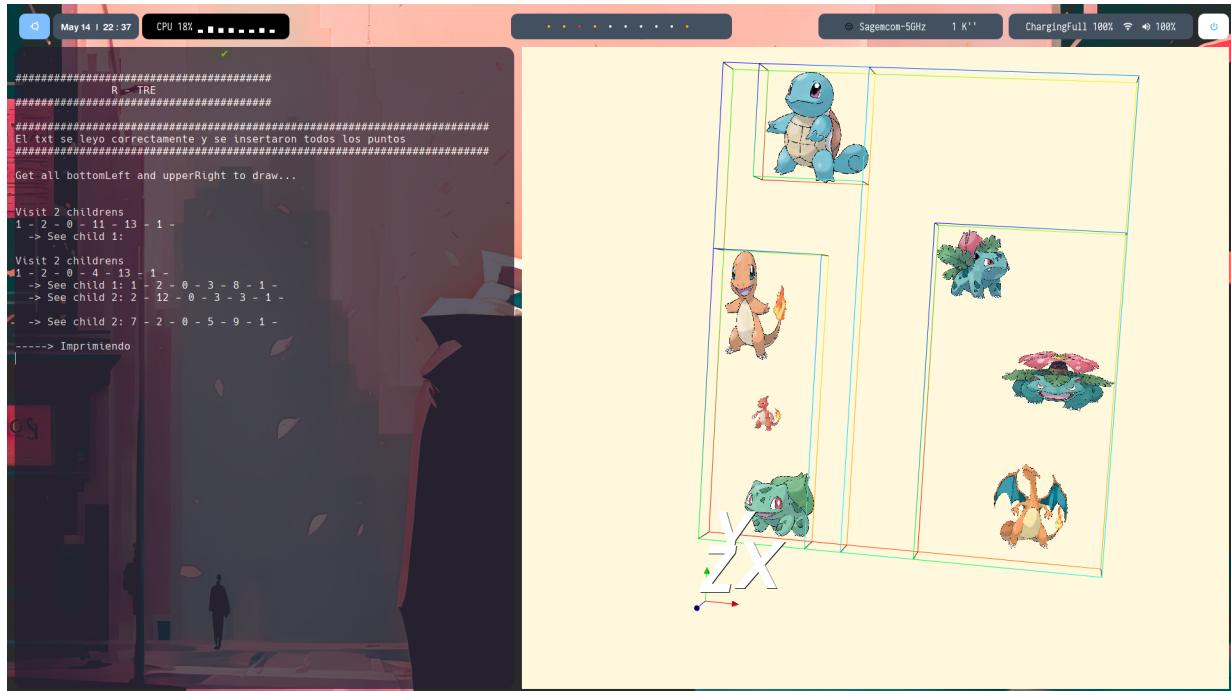


Figure 3.11: Insert - Seguimiento 7



8. Paso 8: Insertamos el Dato H (Wartortle). Seguimos recorriendo los hijos hasta encontrar donde insertarlo, ya que el Dato G - Squirtle está más cerca a él, y no supera su granularidad aún, con libertad podemos insertarlo ahí.

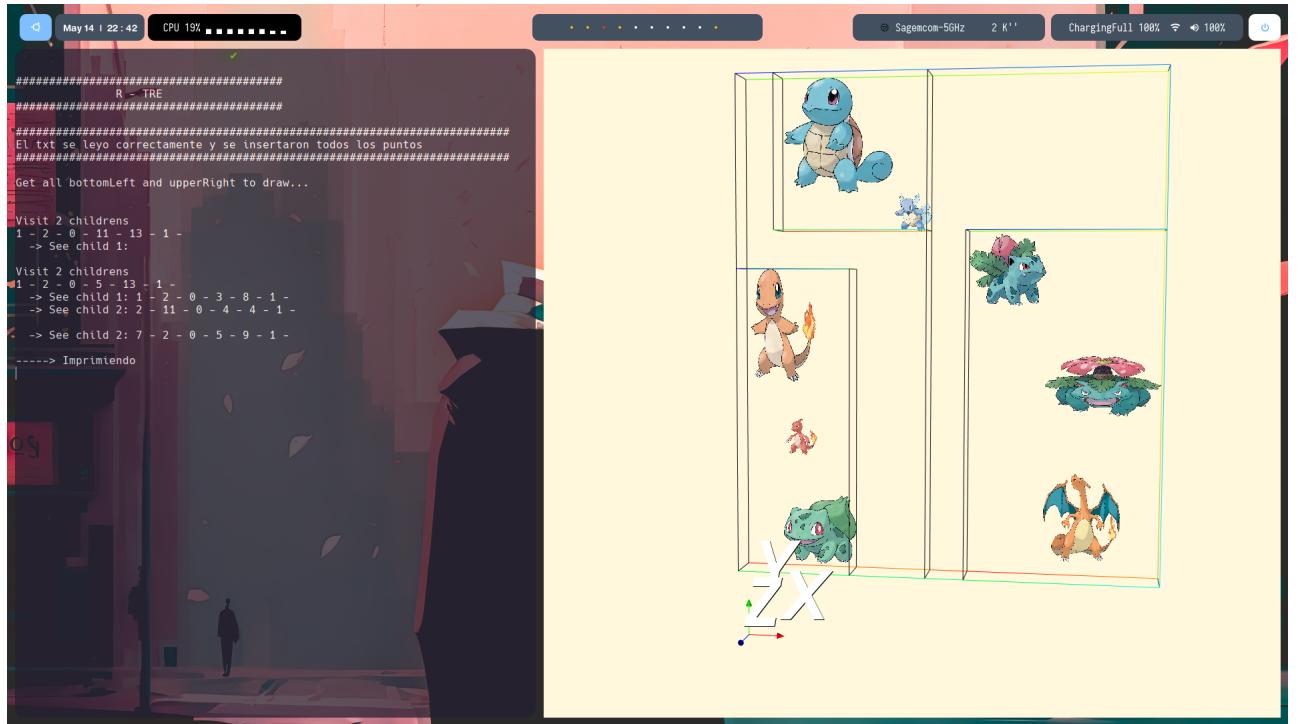


Figure 3.12: Insert - Seguimiento 8



9. Paso 9: Insertamos el Dato I (Blastoise). Sigue los mismos pasos que el paso 8. Solo lo insertamos para llenar con el máximo de datos por nodo.

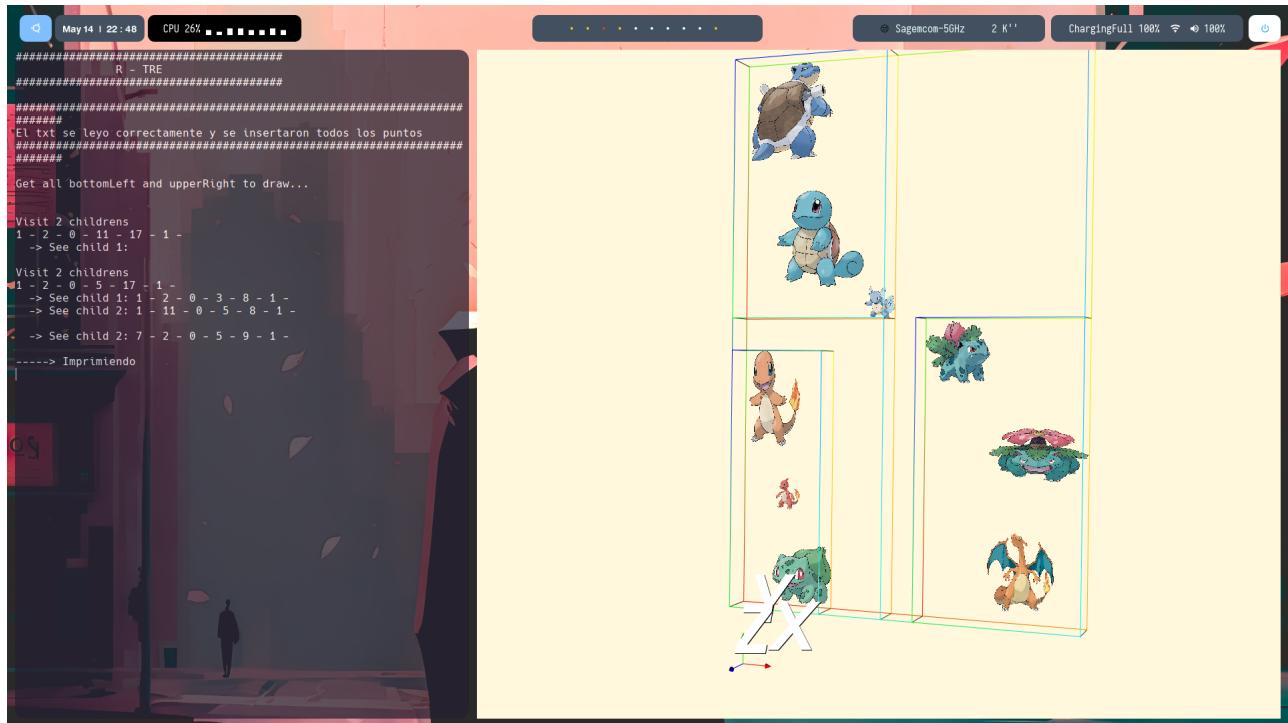


Figure 3.13: Insert - Seguimiento 9



10. Paso 10: Con este último dato, podemos presenciar una de las “limitaciones” más características del R-Tree, la sobre-posición.

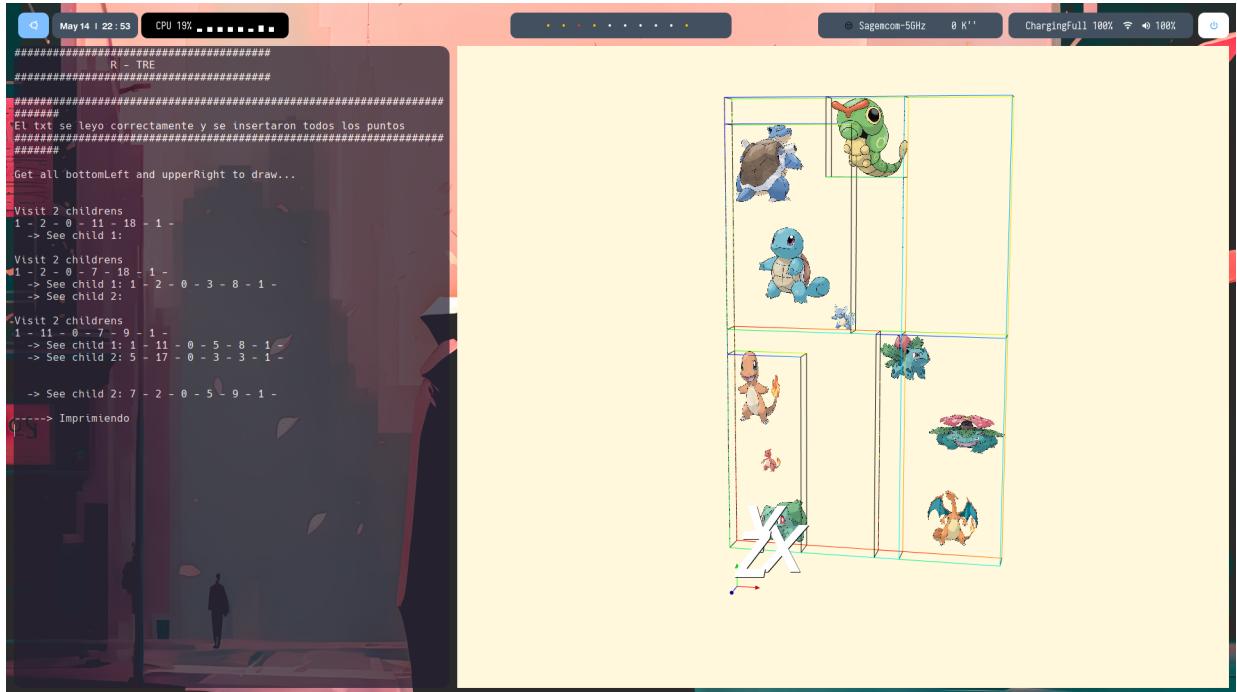


Figure 3.14: Insert - Seguimiento 10

### 3.3 Comentarios finales

Sobre la elección entre la técnica usando equidistantes o la propuesta por Guttman. En la medida que realizaba las pruebas, note que la técnica de equidistantes no era totalmente eficiente; pues en los peores casos, la forma que implemente caía en un bucle infinito.

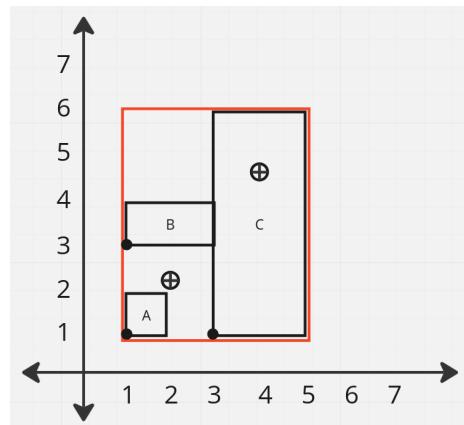


Figure 3.15: Peor caso - Equidistante



Ya que mi función distance2Pokemons compara únicamente las coordenadas de los 2 equidistantes con las coordenadas de cada record, aparentemente dichos 3 puntos están más cercanos al equidistante 1, tal como muestra la figura 3.15: Peor caso - Equidistante. Entonces, el algoritmo no tiene de otra que insertar en un solo nodo (L) dichos 3 datos; el problema radica en que como la granularidad es 2, a ese nodo L, tendría que aplicarse un nuevo Split, y al volver a recalcular los equidistantes caerían en las mismas posiciones, pues el MBR nunca se modificó, cayendo en un bucle infinito.

Considero que necesariamente tenemos que asegurar de alguna manera que tras un split, los nodos L y LL tengan al menos un dato; por eso es que opte por cambiar dicha técnica y usar la que recomienda Guttman, ya que su función peekSeeds me asegura ello.

Como último comentario, es importante resaltar que en el código compartido, por más que sus funciones y bucles usados, sean controlados por una variable “global” como es el ndim (que me indica con cuantas dimensiones trabajamos); su estructura está pensado para únicamente trabajar con 3 dimensiones. Sí se cambia dicho valor, lo más probable es que ocurran errores.



## References

- [1] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14:47–57, 1984.