UNIVERSIDAD NACIONAL DE SAN AGUSTIN

MAESTRIA EN CIENCIAS DE LA COMPUTACIÓN

CURSO: ALGORITMOS Y ESTRUCTURA DE DATOS

PROFESORES: Lizeth Joseline Fuentes Perez

Luciano Arnaldo Romero Calla

BRIGGI RIVERA GUILLÉN

JUAN MARQUINHO VILCA CASTRO

2018

Layered Range Tree

1. Búsqueda en rango ortogonal (Orthogonal range searching)

Dado un conjunto de puntos en cualquier espacio d-Dimensional, una consulta del tipo rango ortogonal utiliza una caja que encierra a un subconjunto de puntos del conjunto inicial, las consultas pueden ir dirigidas a la cantidad de puntos en la caja de consulta u obtener todos los puntos dentro de la caja o una cantidad finita K de puntos dentro de ésta.

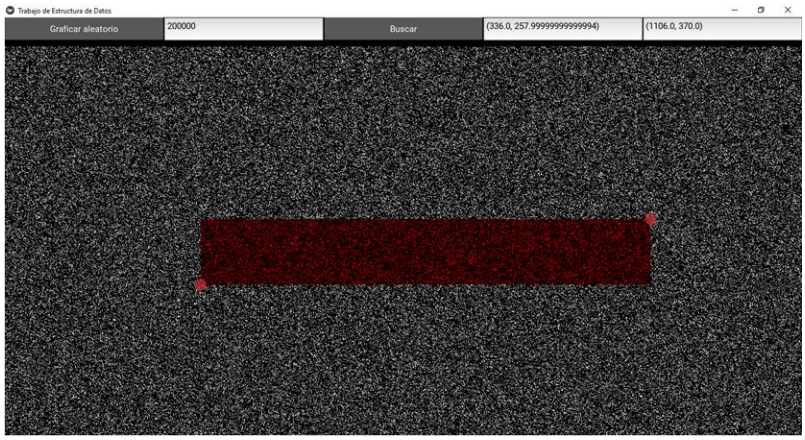


Figura 1. Búsqueda en rango ortogonal para un espacio 2-Dimensional

Una consulta en rango ortogonal puede hacerse en tiempo lineal, es decir, consultando que puntos están dentro de la caja. Esta técnica nos entrega un tiempo **O(n)** siendo n el número total de puntos en el espacio a consultar. Existen numerosas técnicas que nos permiten obtener una consulta en rango ortogonal en un mejor tiempo.

1.1. Range Trees

Este tipo de árboles fueron introducidos por **[1]**, siendo estructuras de datos ordenadas que almacenan un conjunto de puntos de un espacio d-dimensional, esta estructura de datos surge como una alternativa para los Kd-Trees, comparados a estos los Range Trees ofrecen un tiempo de consulta más rápido, siendo este $\mathbf{O(\log^d n + k)}$, pero el espacio utilizado para almacenar la estructura de datos es menos eficiente que la usada por el Kd-Tree $\mathbf{O(n\log^{d-1} n)}$, siendo **d** la dimensión del espacio de puntos, **n** el número de puntos en el espacio y **k** la cantidad de puntos reportados por la consulta**.**

Para el caso 1-D, se tiene un árbol primario (que puede ser un árbol binario balanceado) que permite la indexación de los puntos utilizando la primera componente (componente x), en la **Figura 2.** podemos apreciar cómo se generarán los índices del árbol primario.

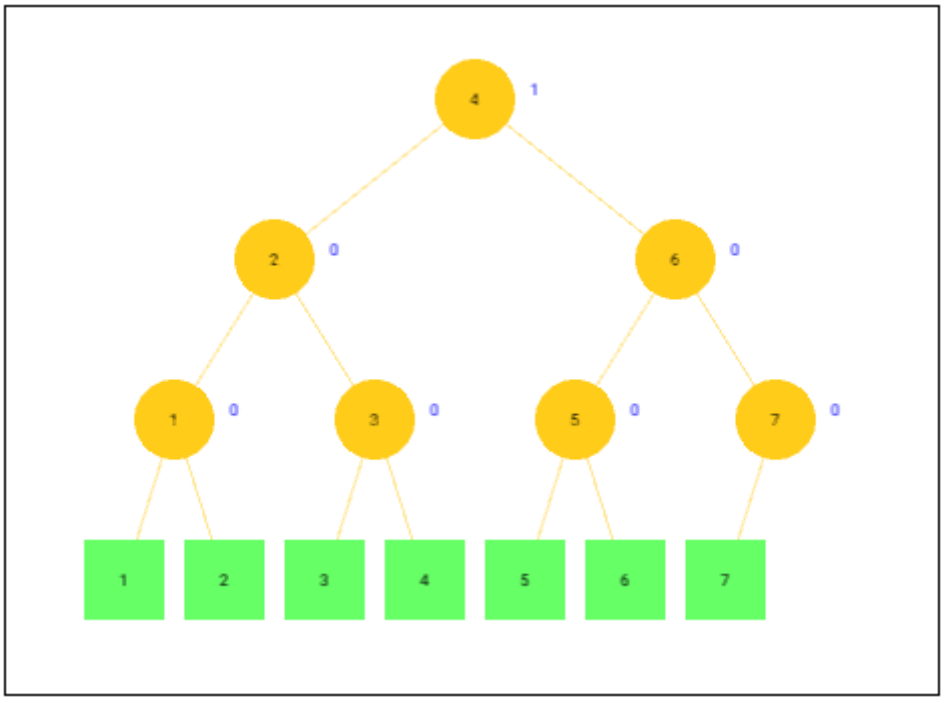


Figura 2. Generación de los índices para la primera componente del espacio D-dimensional

Como se aprecia en la **Figura 2.** Si utilizamos un árbol binario balanceado, entonces los sub-arboles izquierdos contendrán índices de claves inferiores o **iguales** a la raíz y los subárboles derechos contendrán índices de claves superiores a la raíz. Una diferencia resaltante en este tipo de estructura es que los nodos hoja (pintados de lila) contienen a las hojas mientras los nodos intermedios solo contienen índices referidos a la primera componente del espacio D-Dimensional.

Las consultas ortogonales sobre la estructura primaria radican en dos puntos (esto se cumple también para espacios de más dimensiones 2, 3, 4, etc.) estos puntos crean una caja la cual encerrará puntos del espacio de consulta, como se aprecia en la **Figura 3.** Mientras la entrada de la consulta son los puntos 3 y 16 la salida de la consulta es un conjunto M de puntos del espacio global.

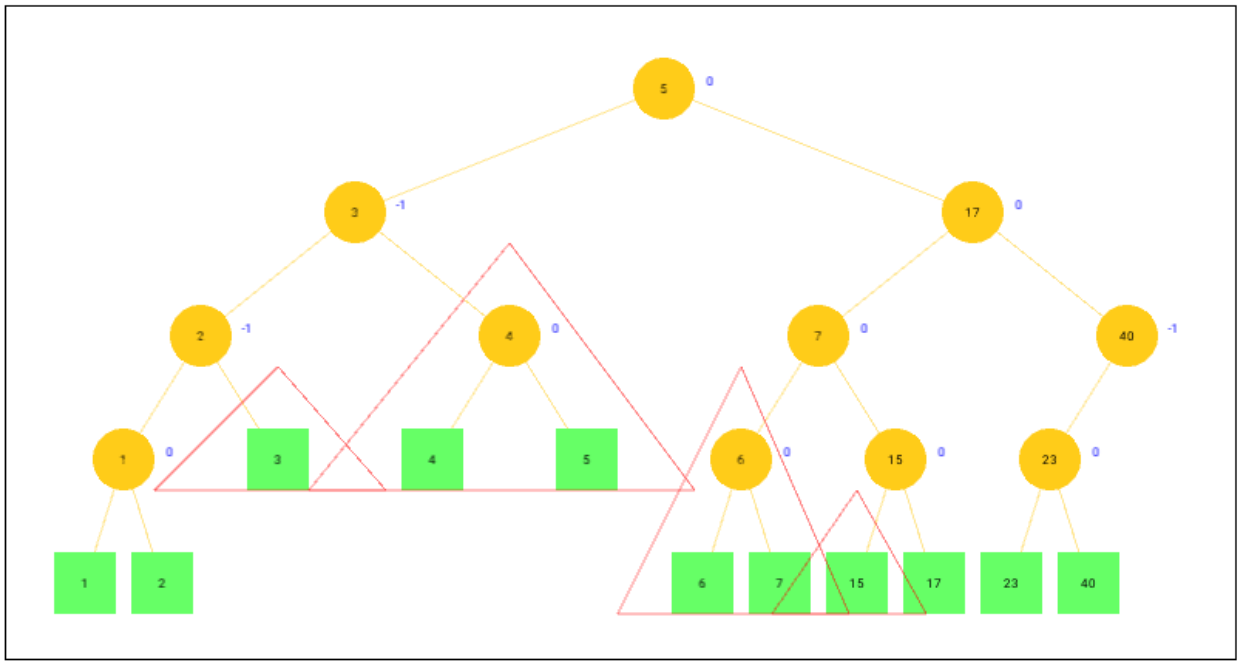


Figura 4. Consulta Ortogonal sobre la estructura primaria.

1.2. Layered Range Trees

Esta estructura de datos fue descrita por [4] y [5], conocida en su forma abreviada como LRT, soporta consultas de rango ortogonal en un mejor tiempo que el Range Tree, el costo algorítmico de consulta para el Range Tree es de $\mathbf{O(\log^{d-1} n + k)}$ donde **d** es el número de dimensiones, **n** es el número de puntos en la estructura de datos y **k** es el número de puntos en la consulta.

Dado un conjunto de puntos S, el árbol binario principal en el LRT es construida usando la coordenada x de los puntos S, este permite búsquedas binarias de puntos usando solamente la coordenada X, la estructura asociada a cada nodo en un sub-árbol contiene todos los puntos contenidos en la estructura debajo del nodo, pero ordenados por la siguiente coordenada y**.**

En la **Figura 5.** Se aprecia la estructura de datos asociada al árbol de puntos ordenado por la componente X, en las hojas del árbol podemos ver que se almacenan los puntos.

En la **Figura 6.** Se aprecia la estructura de datos asociada al árbol de puntos ordenado por la componente Y, los puntos están almacenados en cada nivel del árbol.

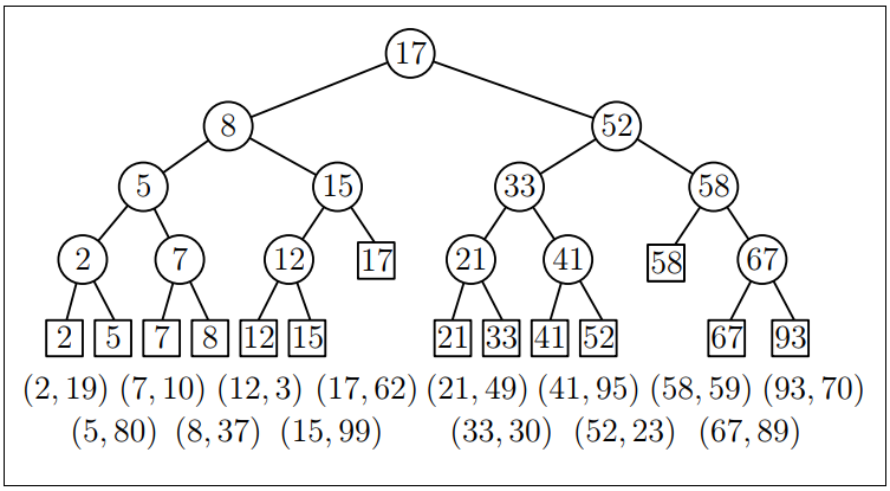


Figura 5. Árbol principal indexado por la componente X, los puntos están en las hojas

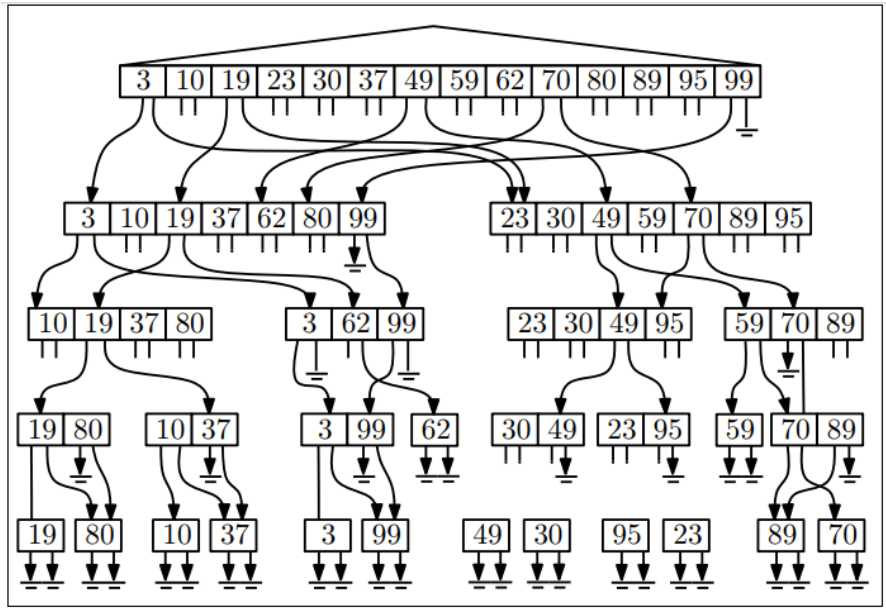


Figura 6. Estructura de Fractional Cascading asociada al árbol de la Figura 5.

2. Detalles de implementación

Se ha utilizado el lenguaje de programación Python 3.6 para la implementación de la estructura de datos:

El código fuente del demo está almacenado en el repositorio Git en el siguiente link:

https://github.com/juanuy1985/TareaEstructuraDeDatos

Se tienen dos demos:

1. **Demo 1-D:** muestra cómo funciona un Range Tree de una sola dimensión.
2. **DemoPrincipal**: muestra cómo funciona un Layered Range Tree de dos dimensiones.

Los algoritmos implementados son los siguientes:

Algoritmo 1.0: Bulk (points)

```
1.  pointsX  = Ordenar points por su coordenada X
2.  pointsY  = Ordenar points por su coordenada Y
    internalNodes = Por cada punto en pointsX crear un nodo interno
3.  que contenga como
                clave a la coordenada X del
                punto
    leaves = Por cada punto en pointsY crear un nodo hoja que contenga
4.  como clave a la
                coordenada X y como valor al punto en
                cuestión
5.  listaNodosParaInsertarHojas = Lista vacia
    root = makeTree( internalNodes,
6.  listaNodosParaInsertarHojas)
7.  index = 0
8.  Para cada nodo en InternalNodes
        Si el hijo Izquierdo de nodo
9.      es nulo
                hijo izquierdo de nodo =
10.             leaves[index]
                index =
11.             index + 1
        Si index < Largo(leaves) e hijo derecho de nodo es
12.     nulo
                hijo derecho de nodo =
13.             leaves[index]
                index =
14.             index + 1
15. Fin Para
    creamos la primera capa del Fractional Cascading a la cuál apuntará
16. el nodo root
    makeFractionalCascadeRecursive(
17. root )
```

Algoritmo 2.0: makeTree (internalNodes, listaNodosParaInsertarHojas)

```
1
.       newRoot  = Creamos un Internal Node sin Key
2
.       makeTreeRecursive( internalNodes, 0, Largo(internalNodes)-1, newRoot, listaNodosParaInsertarHojas)
3
.       retornar newRoot
```

Algoritmo 3.0: makeTreeRecursive (internalNodes, a, b, node, listaNodosParaInsertarHojas)

```
1.      m =  ceil ((a+b)/2)
        childrenLeft = 0 // Altura máxima del hijo izquierdo, esto permitirá
2.      actualizar el factor de balanceo
        childrenRight = 0 // Altura máxima del hijo derecho, esto permitirá
3.      actualizar el factor de balanceo
4.      Si a < b
5.          si a < m
                hijo izquierdo de nodo = Creamos
6.              InternalNode sin clave
                childrenLeft = makeTreeRecursive (internalNodes, a, m-1, Hijo izquierdo de nodo,
7.              listaNodosParaInsertarHojas)
8.          si b > m
                hijo derecho de nodo = Creamos
9.              InternalNode sin clave
                childrenRight = makeTreeRecursive (internalNodes, m+1, b, Hijo derecho de nodo,
10.             listaNodosParaInsertarHojas)
        clave de nodo =
11.     internalNodes[m].clave
        Si el hijo izquierdo o el derecho de
12.     nodo son nulos
            listaNodosParaInsertarHojas.add(
13.         nodo )
        factor de nodo = childrenRight -
14.     childrenLeft
        retornar max( childrenRight,
15.     childrenLeft)
```

Algoritmo 4.0: search (p1, p2)

```
            lca = getLCA(
1.          p1, p2 )
            Si lca es nodo
2.          hoja
                Si la clave de LCA != p1.x o  la clave
3.              de LCA != p2.x
                        retornar
                        conjunto
4.                      vacio
5.              return conjunto formado por el punto
```

```
                almacenado por lca
        S1 = getLeftChildren(lca, p1,
6.      p2)
        S2 = getRightChildren(lca, p1,
7.      p2)
        retornar S1 U
8.      S2
```

Algoritmo 5.0: getLCA (p1, p2)

```
        node =
1.      root
2.      Mientras nodo != nulo y el nodo no es hoja
            Si la clave del nodo > p1.x y la clave del
3.          nodo > p2.x
                nodo = hijo
4.              izquierdo del nodo
            Si la clave del nodo < p1.x y la clave del
5.          nodo < p2.x
                nodo = hijo derecho
6.              del nodo
7.          O Si No Salir de Mientras
8.      retornar nodo
```

Algoritmo 6.0: getLeftChildren (lca, p1, p2)

```
        S = Lista de puntos
1.      vacia
2.      nodo = hijo izquierdo de lca
        pos1 = binary_search( Arreglo asociado al nodo LCA, p1.y ) // Buscamos posición de y en el
3.      arreglo de coordenadas Y
        pos2 = binary_search( Arreglo asociado al nodo LCA, p2.y ) // Buscamos posición de y en el
4.      arreglo de coordenadas Y
        Mientras nodo no sea nulo y no sea una
5.      hoja
6.          Si la clave del nodo >= p1.x
                insertar en S la sublista obtenida del arreglo asociado al hijo derecho de
7.              nodo con limites pos1 y pos2
                pos1 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
8.              hijo izquierdo de pos1
                pos2 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
9.              hijo izquierdo de pos2
                nodo = hijo
10.             izquierdo de nodo
            O Si
11.         no
                pos1 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
12.             hijo derecho de pos1
                pos2 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
13.             hijo derecho de pos2
14.             nodo = hijo derecho
```

```
                    de nodo
15.     Fin Mientras
        Si el nodo es Hoja y la clave del nodo <=
16.     p1.x
            insertar en S la sublista obtenida del arreglo asociado al nodo con
17.         limites pos1 y pos2
        retornar
18.     S
```

Algoritmo 7.0: getRightChildren (lca, p1, p2)

```
        S = Lista de puntos
1.      vacia
2.      nodo = hijo derecho de lca
        pos1 = binary_search( Arreglo asociado al nodo LCA, p1.y ) // Buscamos posición de y en el
3.      arreglo de coordenadas Y
        pos2 = binary_search( Arreglo asociado al nodo LCA, p2.y ) // Buscamos posición de y en el
4.      arreglo de coordenadas Y
        Mientras nodo no sea nulo y no sea una
5.      hoja
6.          Si la clave del nodo <= p1.x
                insertar en S la sublista obtenida del arreglo asociado al hijo izquierdo de
7.              nodo con limites pos1 y pos2
                pos1 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
8.              hijo derecho de pos1
                pos2 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
9.              hijo derecho de pos2
                nodo = hijo derecho
10.             de nodo
            O Si
11.         no
                pos1 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
12.             hijo izquierdo de pos1
                pos2 = Acceder a la estructura del Fractional Cascading para obtener la posición en el
13.             hijo izquierdo de pos2
                nodo = hijo
14.             izquierdo de nodo
15.     Fin Mientras
        Si el nodo es Hoja y la clave del nodo >=
16.     p1.x
            insertar en S la sublista obtenida del arreglo asociado al nodo con
17.         limites pos1 y pos2
        retornar
18.     S
```

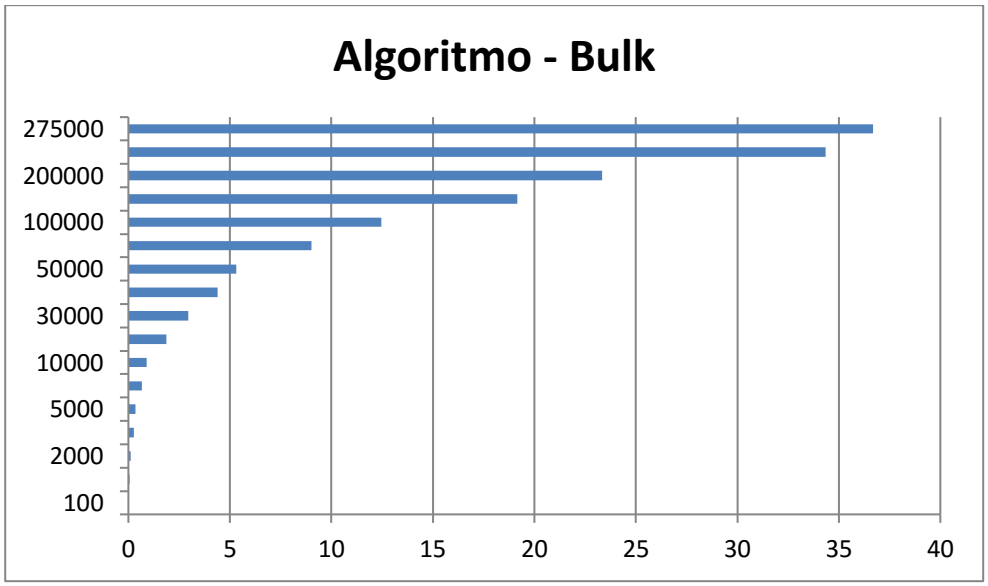
3. Experimentos

Se utilizaron las siguientes computadoras para realizar las pruebas de desempeño de la estructura de datos:

	Sistema Operativo	Procesador	Memoria
Computadora I	Windows 7 Enterprise - 64 bits	Core i-5 - 2500, 3.30 Ghz	4.00 Gb
Computadora II	Windows XP Professional - 32 bits	Pentium Dual Core E5800 - 3.20 Ghz	2.00 Gb
Computadora III	Windows 10 Pro - 64 bits	Core i-7 - 5500 U CPU 2.4 Ghz	8.00 Gb

Tabla 1. Caracteristicas de las computadoras utilizadas para el experimento.

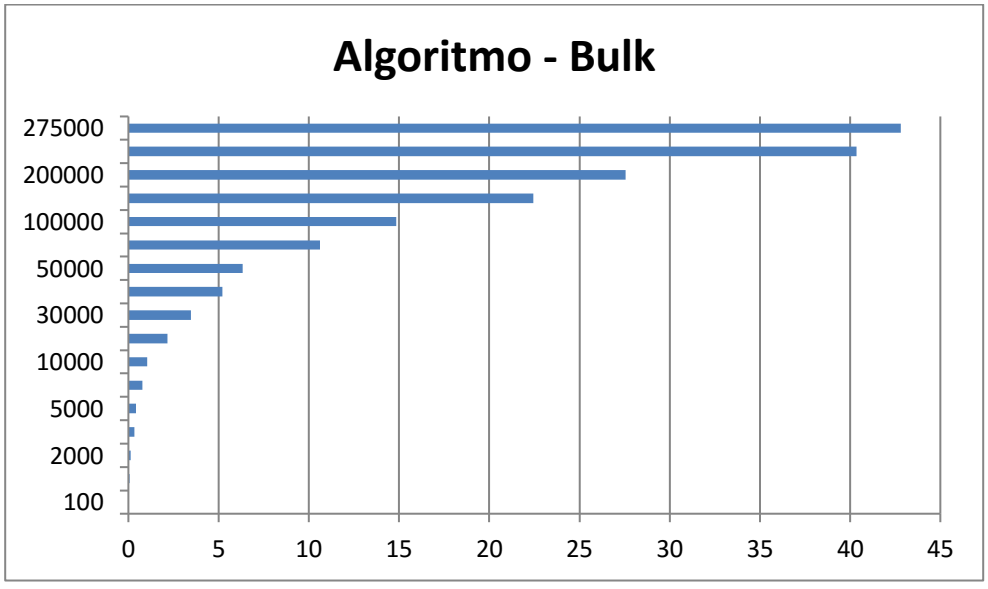
3.1. Algoritmo de Bulk



Cuadro 1. Algoritmo Bulk – para la computadora 1. Se muestra cantidad de nodos contra segundos.

Algoritmo - Bulk	
100	0.003982216
1000	0.047360853
2000	0.110718366
4000	0.281406019
5000	0.361140223
8000	0.653361824
10000	0.898656713
20000	1.871913199
30000	2.958611073
40000	4.403469893
50000	5.310203306
80000	9.018977803
100000	12.46314498
150000	19.17195524
200000	23.32587685
250000	34.3556239
275000	36.6816127

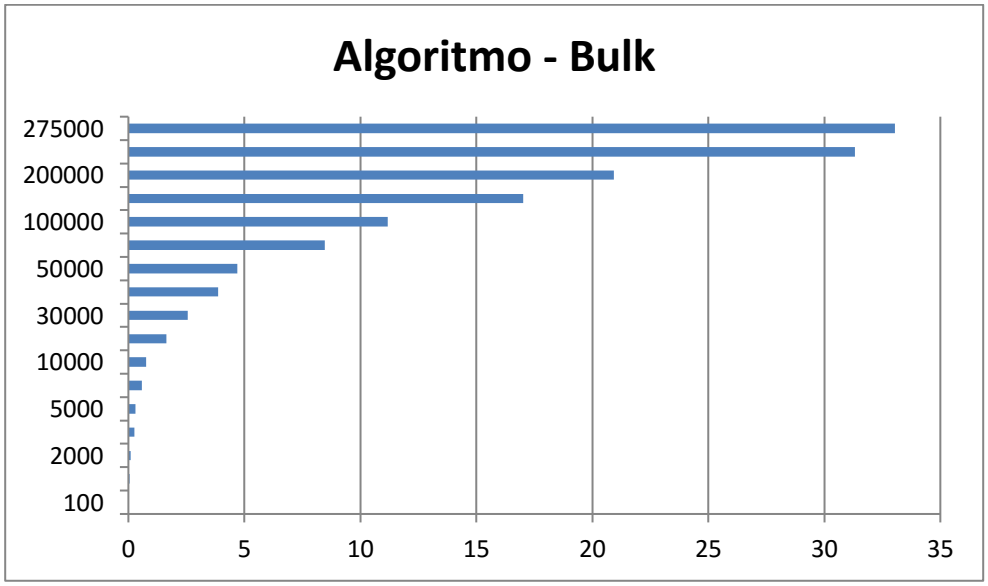
Tabla 2. Algoritmo Bulk – para la computadora 1. Se muestra cantidad de nodos contra segundos.



Cuadro 2. Algoritmo Bulk – para la computadora 2. Se muestra cantidad de nodos contra segundos.

Algoritmo - Bulk	
# elementos	Tiempo (segundos)
100	0.004598099
1000	0.055600306
2000	0.130016489
4000	0.327597515
5000	0.416891651
8000	0.780107372
10000	1.053403646
20000	2.163915833
30000	3.471521714
40000	5.213302668
50000	6.331291812
80000	10.62885088
100000	14.84438544
150000	22.44168116
200000	27.54978313
250000	40.34324304
275000	42.82157676

Tabla 3. Algoritmo Bulk – para la computadora 2. Se muestra cantidad de nodos contra segundos.

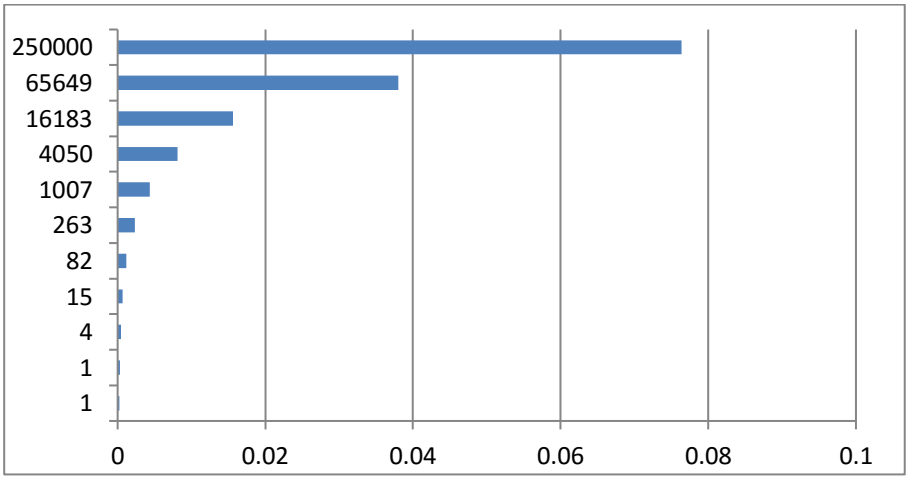


Cuadro 3. Algoritmo Bulk – para la computadora 3. Se muestra cantidad de nodos contra segundos.

Algoritmo - Bulk	
# elementos	Tiempo (segundos)
100	0.003447744
1000	0.041319465
2000	0.097685923
4000	0.256287257
5000	0.312501042
8000	0.587203492
10000	0.770124203
20000	1.6393195
30000	2.559860174
40000	3.85745535
50000	4.696071274
80000	8.455513973
100000	11.18795975
150000	17.01460651
200000	20.920122
250000	31.30962849
275000	33.04434469

Tabla 4. Algoritmo Bulk – para la computadora 3. Se muestra cantidad de nodos contra segundos.

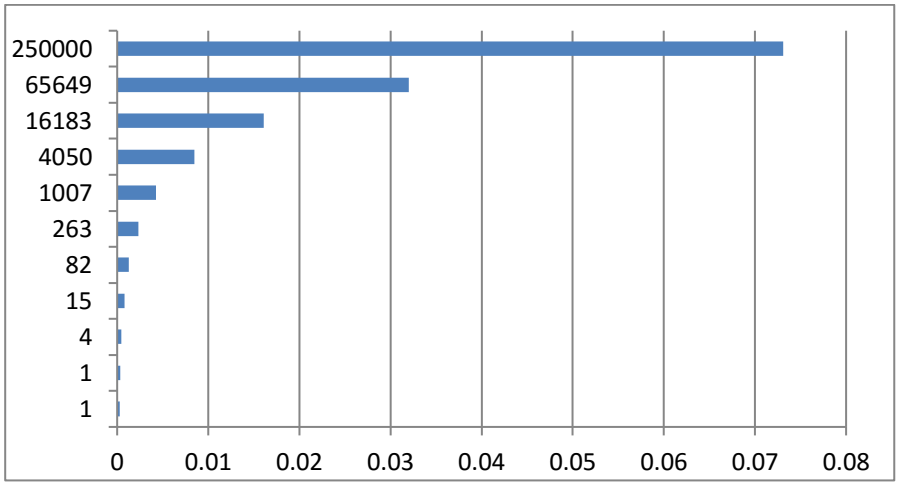
3.2. Algoritmo Search



Cuadro 4. Algoritmo Search – para la computadora 1. Se muestra cantidad de elementos obtenidos contra segundos.

Algoritmo - Search (árbol con 250000 elementos)		
# Rangos de Consulta	Tiempo (segundos)	Elementos obtenidos
[[0, 0], [1000, 1000]]	0.000237536	0
[[0, 0], [2000, 2000]]	0.000276192	1
[[0, 0], [4000, 4000]]	0.000433055	6
[[0, 0], [8000, 8000]]	0.00069412	15
[[0, 0], [16000, 16000]]	0.001159668	55
[[0, 0], [32000, 32000]]	0.002322698	255
[[0, 0], [64000, 64000]]	0.004328308	1019
[[0, 0], [128000, 128000]]	0.00812216	4185
[[0, 0], [256000, 256000]]	0.015612385	16555
[[0, 0], [512000, 512000]]	0.038018073	65958
[[0, 0], [1000000, 1000000]]	0.076391891	250000

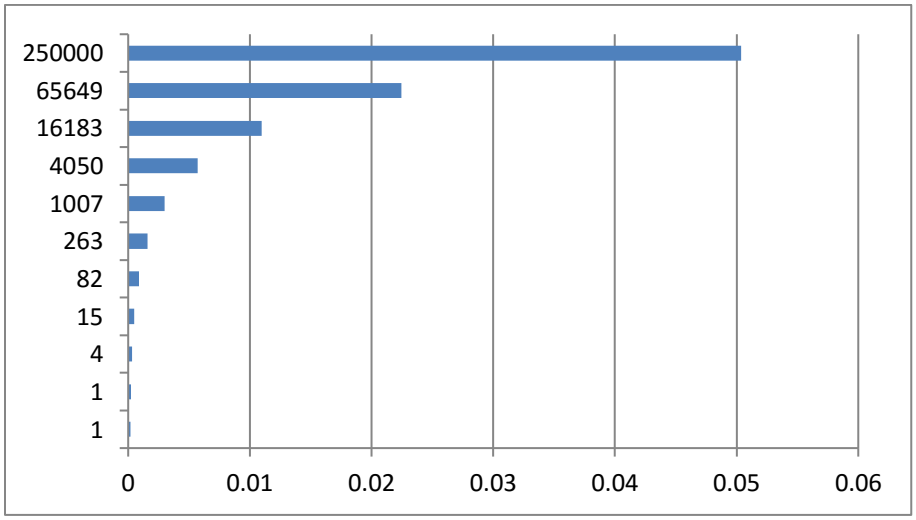
Tabla 5. Algoritmo Search – para la computadora 2. Se muestra cantidad de elementos obtenidos contra segundos.



Cuadro 5. Algoritmo Search – para la computadora 3. Se muestra cantidad de elementos obtenidos contra segundos.

Algoritmo - Search (árbol con 250000 elementos)		
# Rangos de Consulta	Tiempo (segundos)	Elementos obtenidos
[[0, 0], [1000, 1000]]	0.000259889	0
[[0, 0], [2000, 2000]]	0.000317174	0
[[0, 0], [4000, 4000]]	0.000466113	3
[[0, 0], [8000, 8000]]	0.000802582	21
[[0, 0], [16000, 16000]]	0.001272915	69
[[0, 0], [32000, 32000]]	0.002322121	258
[[0, 0], [64000, 64000]]	0.004288476	1038
[[0, 0], [128000, 128000]]	0.008485298	4079
[[0, 0], [256000, 256000]]	0.016063093	16359
[[0, 0], [512000, 512000]]	0.031976645	65701
[[0, 0], [1000000, 1000000]]	0.073103688	250000

Tabla 6. Algoritmo Search – para la computadora 2. Se muestra cantidad de elementos obtenidos contra segundos.



Cuadro 6. Algoritmo Search – para la computadora 3. Se muestra cantidad de elementos obtenidos contra segundos.

Algoritmo - Search (árbol con 250000 elementos)		
# Rangos de Consulta	Tiempo (segundos)	Elementos obtenidos
[[0, 0], [1000, 1000]]	0.000179187	1
[[0, 0], [2000, 2000]]	0.000218959	1
[[0, 0], [4000, 4000]]	0.000316464	4
[[0, 0], [8000, 8000]]	0.000511901	15
[[0, 0], [16000, 16000]]	0.000875407	82
[[0, 0], [32000, 32000]]	0.001606267	263
[[0, 0], [64000, 64000]]	0.002996997	1007
[[0, 0], [128000, 128000]]	0.005710031	4050
[[0, 0], [256000, 256000]]	0.010984283	16183
[[0, 0], [512000, 512000]]	0.022459941	65649
[[0, 0], [1000000, 1000000]]	0.050372465	250000

Tabla 7. Algoritmo Search – para la computadora 3. Se muestra cantidad de elementos obtenidos contra segundos.

4. Conclusiones y discusión

- La estructura Layered Range Tree puede preprocesar un espacio de puntos en un tiempo **$O(n*\log^{d-1}n)$**, mientras que el tiempo de consulta es de **$O(\log^{d-1}n+k)$,** siendo k el número de puntos obtenidos de la consulta, es decir, si la consulta tiene una caja muy grande el costo algorítmico crece.
- Para la estructura de datos que soportará el Fractional Cascading, la teoría dice que debe almacenarse solo los índices en las capas, pero para un acceso eficiente, se debe de almacenar todo el punto, de tal manera que se pueda extraer la información de puntos de una forma más eficiente.
- La estructura de Fractional Cascading solo se aplica en la última dimensión del espacio a indexar, es decir si tenemos un espacio con "d" componentes el LRT será igual a un Range Tree para las primeras d-1 coordenadas, pero en la última es dónde se aplica la búsqueda en cascada, es por eso la diferencia en sus tiempos de consulta: **$O(\log^{d}n+k)$** para el Range Tree y **$O(\log^{d-1}n+k)$** para el LRT.
- Algunas fuentes indican que solo la raíz del árbol debe de contener una estructura asociada para la siguiente coordenada, esto permitiría búsquedas en las que no participe X, pero como el propósito del LRT es poder realizar búsquedas Ortogonales esto no serviría mucho.

5. Referencias

[1] Bentley, J. L. (1979). "Decomposable searching problems". Information Processing Letters. **8** (5): 244–251.

[2] Chazelle, B., & Guibas, L. J. (1986). Fractional cascading: I. A data structuring technique. Algorithmica, 1(1-4), 133-162.

[3] Chazelle, B., & Guibas, L. J. (1986). Fractional cascading: II. applications. *Algorithmica*, *1*(1-4), 163-191.

[4] Dan E. Willard. New data structures for orthogonal range queries. SIAM J. Comput., 14(1):232–253, 1985.

[5] G.S. Lueker. A data structure for orthogonal range queries. Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science, pages 28–34, 1978.