CC4102 Diseño y Análisis de Algoritmos

Informe Tarea II

Agustín López Q.
Matías Cisterna M.

5 de noviembre de 2014

Índice

1. Introducción

En esta tarea 2 de Diseño y Análisis de Algoritmos se busca analizar el desempeño de búsqueda en diferentes estructuras de datos. Las estructuras a analizar son las siguientes: Árbol Binario de Búsqueda, Árbol AVL, Árbol de Van Emde Boas, Splay Tree y Árbol Optimo.

Un árbol binario de búsqueda (ABB) es una estructura de datos basada en nodos en forma de árbol. Es un árbol binario (solo puede haber hasta dos nodos hijos por padre), en el que todo a la izquierda del nodo padre es menor que el y todo a la derecha de el es mayor.

Un árbol AVL, es un árbol que cumple las características de un árbol binario de búsqueda. Sin embargo este tiene la particularidad de auto balancearse. Esto quiere decir que que la altura de las ramas del lado izquierdo no difieren en mas de una unidad de altura de la derecha.

Un árbol de Van Emde Boas (vEB tree), es una estructura de datos basada en arreglos asociativos en forma de árbol. En que todas sus operaciones son en el orden $(\log(\log M))$ siendo M el numero máximo de elementos que la estructura puede contener.

Un Splay tree es un árbol con las mismas características que un árbol AVL y con la singularidad de que los elementos accesados recientemente tienen mejor tiempo de acceso que el resto. Esto se logra mediante haciendo *splaying* al elemento del árbol. Esto consiste en reordenar el árbol para que ese elemento quede mas arriba, logrando así menores tiempos de acceso.

Un árbol de búsqueda optima es un árbol de búsqueda binaria. Con la excepción de que cuando se genera el árbol uno sabe de antemano cuales son las frecuencias de acceso de cada elemento. Con dicha información uno puede generar un árbol en que los elementos mas accesados estén antes, mejorando así los tiempos de acceso.

Características del Computador Usado:

- Cpu: Intel I7 x980 3.33Ghz
- Ram: 8gm ddr3 800Mhz
- Disco duro: Sata II

2. Hipótesis

Se cree que tanto *ABB*, *AVL Trees* y *Splay Trees* se comportaran de manera similar en su tiempo promedio de búsqueda. Sin embargo, en estas tres estructuras se espera diferentes tiempos de búsqueda en su peor caso. Por otro lado, para *Van Emde Boas tree* Su tiempo de búsqueda va a depender estrictamente de el tamaño M con que se cree el árbol. Aun así este árbol, a menos que $M >>> N$ (siendo n el numero de elementos), debería tener mejores tiempos de acceso que el resto de los mencionados anteriormente. Finalmente se espera que el *Optimal Binary Search Tree* posea el menor tiempo de acceso de búsqueda de todos los arboles.

3. Diseño Experimental

3.1. Implementación

Para la implementación se usó el lenguaje de programación Java. El diseño se dividió en 9 clases:

- *ABBTree*: Representa al árbol binario de búsqueda.
- *AVLTree*: Representa al árbol AVL.
- *AVLNode*: Representa los nodos del árbol AVL.
- *IAVLNode*: Representa los nodos internos del árbol AVL.
- *NullAVLNode*: Representa los nodos nullos del árbol AVL.
- *SplayTree*: Representa al SplayTree.
- *vEBTree*: Representa al árbol de Van Emde Boas.
- *OpABBTree*:
- *StopWatch*: Clase creada para medir tiempos de forma simple.
- *Main*: Clase con método *main*, es desde donde se obtiene los datos.

Para la representación de cada estructura de datos: *ABBTree*, *AVLTree*, *SplayTree*, *vEBTree* y *optimum ABBTree* se genero una clase que los representa. Dentro de cada clase se encuentra un método de inserción y búsqueda. Para las estructuras de datos Splay tree y AVL existen otros métodos que garantizan sus propiedades especificas de auto balanceo y nodos recién utilizados.

- Búsqueda: Los algoritmos de búsqueda respectivamente para las 5 estructuras de datos reciben como parámetro un dato y retorna si se encuentra en la estructura o no y el tiempo que se demoro.
- Inserción: Los algoritmos de inserción dependen exclusivamente de la estructura en que se este ejecutando. Aun así todos reciben un nuevo dato a insertar y estos retornan la nueva estructura de datos con la nueva inserción.

1. *ABBTree*: El método de inserción es bastante simple el árbol inserta el nuevo dato en su posición correspondiente fijándose en cumplir que todo lo que este a su izquierda sea menor a el y a su derecha mayor a el. Esto se cumple para todo nodo en el árbol.
2. *AVLTree*: El método de inserción cumple las mismas condiciones que el *ABBTree* pero ademas este tiene que estar balanceado en su altura. Es decir la diferencia de altura de la rama mas larga con la rama mas corta es a lo mas una unidad.
3. *SplayTree*: El método de inserción cumple las mismas condiciones que el *AVLTree* pero ademas este reorganiza el elemento que tuvo el ultimo accesados para que así si se busca ese mismo elemento sea mas rápido.
4. *vEBTree*: El método de inserción de esta estructura difiere al resto. Esto se debe a que el árbol esta concedido por arreglos y no por nodos. La inserción funciona de la siguiente manera: Si el arreglo esta vació los valores min y max son iguales al nuevo elemento. Si no, si $x < T.min$ entonces insertamos T.min al sub-árbol i responsable de T.min y T.min es ahora igual a x.Si no, si $x > T.max$ insertamos x en el sub-árbol i y T.max ahora es x. Finalmente si $T.min < x < T.max$ insertamos x en el sub-árbol i.
5. *OpABBTree*:

3.2. Generación de Instancias

Se generan conjuntos de tamaños $n \in \{2^{10}, ..., 2^{20}\}$. Ademas cada llave sera un numero natural que esta en el rango $U = \{2^{10}, ..., 2^{20}\}$. Para cada n, se genera un conjunto al azar K_n de n llaves en el rango U. Finalmente se generan secuencias de tamaño $100n$ sobre estas llaves de tres maneras distintas al azar:

- **Distribución 1**: Se escogen $100n$ números al azar entre el conjunto de llaves K_n.
- **Distribución 2**: Se escoge que la i-esima llave aparece con probabilidad c/i^a, donde c es una constante de normalización. $a \in \{1.2, 1.5, 2.0\}$.
- **Distribución 3**: Se escoge que la i-esima llave aparece con probabilidad c/a^i, donde c es una constante de normalización. $a \in \{1.2, 1.5, 2.0\}$.

Finalmente el bloque de código de la creación de las 3 distribuciones queda así:

```
1  for(int n=1024;n <= 1024*1024; n*=2)
2  {
3      potencia = (int)(Math.log(n)/Math.log(2));
4      Kn = new int[n];
5      seq1 = new int[100*n];
6      seq2 = new int[3][100*n];
7      seq3 = new int[3][100*n];
8      freq1 = new double[100*n];
9      freq2 = new double[3][100*n];
10     freq3 = new double[3][100*n];
11     for (int i = 0; i < n; i++)
12     {
13        Kn[i] = r.nextInt(U) + 1;
14     }
15     // secuencia 1
16     for (int i = 0; i < 100*n; i++)
17     {
18        seq1[i] = Kn[r.nextInt(n)];
19        freq1[i] = 1.0/(100*n);
20     }
21     // secuencia 2
22     int i = 0;
23     int k = 0;
24     for(int index=0;index<3;index++)
25     {
26        double c = getC2(a[index],n);
27        while(i <= 100*n)
28        {
29           double freq_k = c/Math.pow(k+1,a[index]); //
                 frecuencia de Kn[k]
30           int cant_k = (int)Math.ceil(100*n*freq_k); //
                 cantidad de Kn[k]
31           for(int j=i;j<(100*n) && j<(i+cant_k);j++)
32           {
33              seq2[index][j] = Kn[k];
34              freq2[index][j] = freq_k;
35           }
36           k++;
37           i += cant_k;
38        }
39     //     secuencia 3
40        i = 0;
41        k = 0;
```

```
42          c = getC3(a[index],n);
43          while(i <= 100*n)
44          {
45              double freq_k = c/Math.pow(a[index],k+1); //
                    frecuencia de Kn[k]
46              int cant_k = (int)Math.ceil(100*n*freq_k); //
                    cantidad de Kn[k]
47              for(int j=i;j<(100*n) && j<(i+cant_k);j++)
48              {
49                  seq3[index][j] = Kn[k];
50                  freq3[index][j] = freq_k;
51              }
52              k++;
53              i += cant_k;
54          }
55      }
56  }
```

3.3. Medidas de Rendimiento

Las medidas de rendimiento usadas son dos en base al tiempo:

- Primero es el tiempo de demora en la construcción de cada estructura en cuestión.
- Segundo es el tiempo de demora en la búsqueda de un elemento en cada estructura.

Finalmente estas medidas, tanto de búsqueda como de inserción, son almacenadas en un nuevo archivo. Creado con el objetivo de almacenar los datos y luego analizarlos.

4. Presentación de Resultados

4.1. Datos Reales

4.1.1. Inserciones

A continuación se presentan los resultados obtenidos según la distribución de creación de datos usada.

- Distribucion 1:

n	**Tipo Árbol**	**Tiempo Promedio Construcción (milisegundos)**
10482	30296	191

A continuación se presenta en una tabla los resultados del experimento de inserción de rectángulos generados de datos reales en un R-Tree con el método de *LinearSplit*:

Tamaño	**Cantidad I/O's**	**Tiempo (milisegundos)**
10482	30058	58

4.1.2. Búsqueda

A continuación se presenta en una tabla los resultados del experimento de búsqueda de rectángulos generados de datos reales en un R-Tree en el arbol generado con el método de *QuadraticSplit*:

Tamaño	**Cantidad promedio de I/O's**	**Tiempo promedio (milisegundos)**
10482	0.011450381679389313	0.004770992366412214

A continuación se presenta en una tabla los resultados del experimento de búsqueda de rectángulos generados de datos reales en un R-Tree en el arbol generado con el método de *LinearSplit*:

Tamaño	**Cantidad promedio de I/O's**	**Tiempo promedio (milisegundos)**
10482	0.0	0.004770992366412214

4.2. Datos Generados

4.2.1. Inserciones

A continuación se presenta en una tabla los resultados del experimento de inserción de rectángulos aleatorios en un R-Tree con el método de *QuadraticSplit*:

Tamaño (2^n)	**Cantidad I/O's**	**Tiempo (milisegundos)**
9	755	43
12	10656	115
15	118832	1043
18	1193602	9624
21	11553820	84628
24	141142799	476308

Los gráficos de ambos datos, en escala logarítmica para su mejor visualización se encuentran en la Figura **??** y **??**.

Ahora se presenta en una tabla los resultados del experimento de inserción de rectángulos aleatorios en un R-Tree con el método de *LinearSplit*:

Tamaño (2^n)	**Cantidad I/O's**	**Tiempo (milisegundos)**
9	740	23
12	10471	36
15	113245	451
18	1192441	4683
21	11118402	46764
24	136824746	252177

Los gráficos de ambos datos, en escala logarítmica para su mejor visualización se encuentran en la Figura **??** y **??**.

4.2.2. Búsqueda

A continuación se presenta en una tabla los resultados del experimento de busqueda de rectángulos aleatorios en un R-Tree con el método de inserción *QuadraticSplit*:

Tamaño (2^n)	**Cantidad promedio de I/O's**	**Tiempo promedio (milisegundos)**
9	0.0	0.0
12	0.0	0.0
15	0.0	0.0
18	0.0	7.629510948348211E-5
21	0.0	3.337863290656367E-5
24	0.0	2.384186643667213E-5

A continuación se presenta en una tabla los resultados del experimento de busqueda de rectángulos aleatorios en un R-Tree con el método de inserción *LinearSplit*:

Tamaño (2^n)	**Cantidad promedio de I/O's**	**Tiempo promedio (milisegundos)**
9	0.0	0.0
12	0.0	0.0024449877750611247
15	0.0	9.157509157509158E-4
18	0.0	1.1444266422522316E-4
21	0.0	4.291538516558186E-5
24	0.0	2.7418146402172948E-5

4.3. Gráficos

5. Análisis e Interpretación de Datos

5.1. Inserciones

Podemos ver que para el caso de datos generados la cantidad de I/O's y el tiempo a medida que crece n, es lineal para ambos casos de inserción(*LinearSplit* y *QuadraticSplit*). Sin embargo, podemos notar que para el caso de *LinearSplit* este toma casi la mitdad del tiempo que *QuadraticSplit*. Ademas *LinearSplit* hace ligeramente menos I/O's que *QuadraticSplit*.

Por otro lado, en el caso en que los rectángulos son generados a partir de datos reales. Vemos que nuevamente el tiempo de inserción para el método *LinearSplit* toma bastante menos tiempo que textitQuadraticSplit. Sin embargo, en la cantidad de I/O's ambos algoritmos se comportan relativamente similar.

Dado la naturaleza de ambos algoritmos de inserción es fácil ver que para el caso de *LinearSplit* este iba a lograr menores tiempos de inserción en comparación a *QuadraticSplit*. Esto se debe a que la naturaleza de su algoritmo hace generar arboles mas irregulares. Es decir, genera arboles menos balanceados que *QuadraticSplit*. Esto se debe a que las estrategias de *LinearSplit* son mas laxas. Sin embargo, esto puede producir que al momento de realizar operaciones sobre un árbol generado por *LinearSplit* esta tome mas tiempo que en *QuadraticSplit*.

5.2. Búsqueda

Lamentablemente después de analizar ambos casos, Datos reales y generados. Se puede concluir que el método de búsqueda posee algún error. Se cree que este no estaba cargando los datos de disco por lo tanto al leerlos de memoria primaria arroja los datos adjuntos. Que son de costo prácticamente cero para ambos casos, que es de esperarse si los datos son leídos de memoria primaria.

Por otro lado se esperaba que *QuadraticSplit* tuviese mejores tiempos y menores I/O's que *LinearSplit*.