Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Depto. de Ciencias de la Computación
CC4102 - Diseño y Análisis de Algoritmos

Tarea 1

Integrantes : Rodrigo Delgado
Belisario Panay
Gabriel Sanhueza
Profesor : Gonzalo Navarro
Ayudante : Sebastián Ferrada
Auxiliar : Jorge Bahamondes

Índice

1. Introducción

1.1. Problema a resolver

Los R-trees son un tipo de árbol que se maneja en memoria secundaria, el cual contiene rectángulos como información. El problema a resolver consiste en implementar un R-Tree, una herramienta de búsquedas de rectángulos y 2 heurísticas de inserción de rectángulos en el R-Tree.
En particular, se busca evaluar el impacto entre distintas versiones de *inserción* que manejan una versión específica de manejo de *overflow* (para este caso, *Linear Split* y *Greene Split*.

1.2. Hipótesis

Especificaciones de la máquina utilizada

- Procesador: Intel ®Core ®i5-4570 @ 3.20GHz
- Arquitectura: x86_64
- Número de CPUs: 4
- Número de Threads: 4
- Memoria RAM: 8192 KB
- Tamaño de página de disco: M = 2048 bytes ($\ulcorner 40\,\%M \urcorner = 820$)
- Sistema Operativo: Elementary O.S. 0.4 Loki
- Lenguaje usado: C
- Compilador: gcc version 5.4.0

Usaremos siempre la misma semilla de aleatoriedad, para poder tener experimentos "aleatorios" repetibles.
Limitamos la cantidad máxima de rectángulos en un nodo con respecto al tamaño de página del disco.
Así, si *BLOCK_SIZE* = 2048 y tenemos un *struct rectangle* llamado *Rectangle*:

- M = *BLOCK_SIZE / sizeof(Rectangle)* = 2048/32 = 64
- m = 40 % de M. = $40\,\% * 64 = 25$

Por último, la idea es nunca tener más de dos archivos abiertos en un instante dado.
A partir de esto, consideramos que la inserción tome un tiempo corto con pocos rectángulos y se incremente notablemente a medida que se añaden unos cuantos más. Además, esperamos que la búsqueda se demore menos que la inserción.

2. Diseño Experimental

2.1. Metodología

Para cada $n \in \{2^9, ..., 2^{18}\}$, se harán 3 experimentos:

- Inserción de n rectángulos, control de *overflow* con Linear Split.
- Inserción de n rectángulos, control de *overflow* con Greene Split.
- Búsqueda de *n/10* rectángulos.

2.2. Structs

En nuestra implementación hicimos 2 *structs* para manejar los rectángulos en el R-Tree.

2.2.1. Rectangle

Esta estructura posee información sobre:

- Coordenada X.
- Coordenada Y.
- Ancho del rectángulo.
- Alto del rectángulo.
- Identificador (nombre) del rectángulo.
- Identificador del hijo de este rectángulo.

2.2.2. Node

Esta estructura posee información sobre:

- Arreglo dinámico de rectángulos.
- Número de rectángulos escritos en el arreglo.
- Nombre del nodo actual (para uso como nombre de archivo en disco).

2.3. Constantes

- **BLOCK_SIZE:** Tamaño del bloque en disco.
- **count**: Variable global para diferenciar nodos al escribirlos a disco.
- **M**: BLOCK_SIZE / sizeof(Rectangle) — Máximo número de rectángulos en un nodo.
- **m**: 40 % de M — Mínimo número de rectángulos en un nodo.

2.4. Funciones

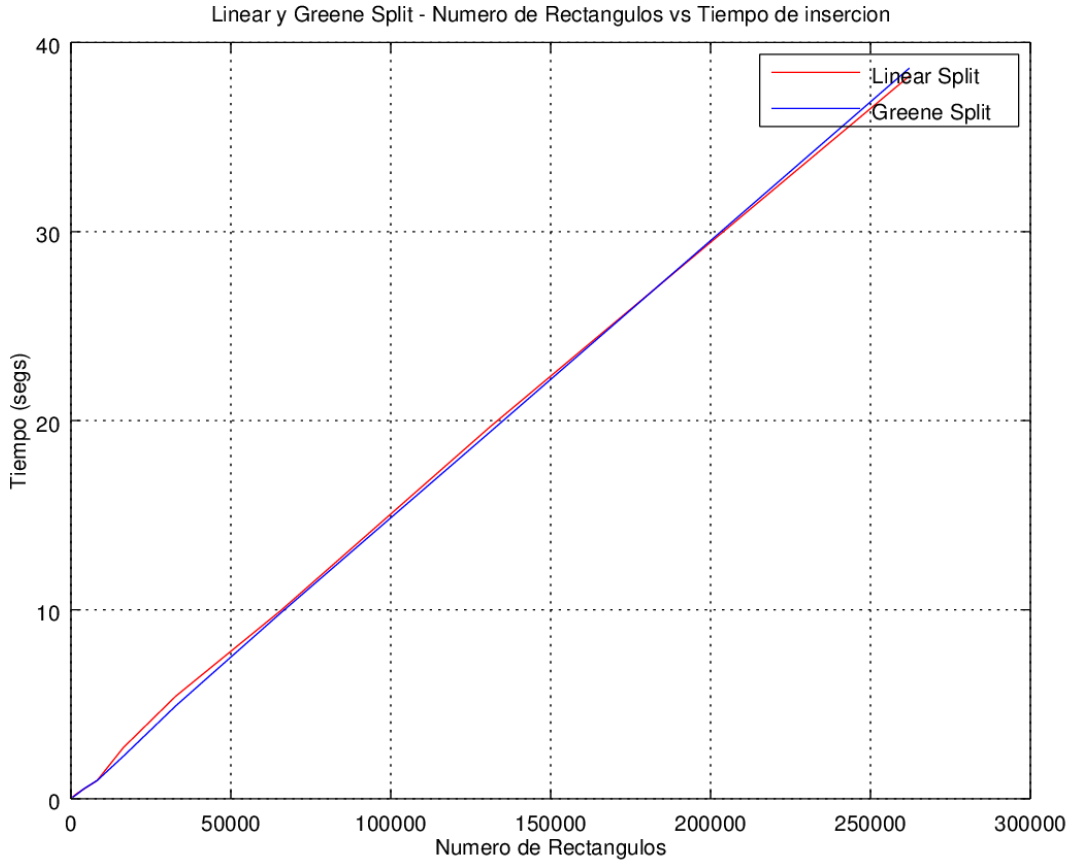
- *Rectangle* createRectangle(int x, int y, int w, int h, int id)*: Crea un rectángulo con coordenadas y nombre.
- *Node* createNode()*: Crea un nodo con un arreglo de rectángulos como información interna.
- *void setAccessToDisk(int n)*: Setter de cantidad de accesos al disco.
- *int getAccessToDisk()*: Getter para accesos al disco.
- *void freeMemory(Node *pNode)*: Libera memoria no usada.
- *writeToDisk(Node *data)*: Escribe los datos de un nodo a disco.
- *Node* loadFromDisk(char *filename)*: Carga un archivo del disco.
- *int intersect (Rectangle *r1, Rectangle *r2)*: Retorna un "booleano"que dice si los dos rectángulos se intersectan.
- *int MBR(Rectangle *r1, Rectangle *r2)*: Calcula la nueva Area si se agrega r2 a r1.
- *void mergeRectangle(Rectangle *r1, Rectangle *r2)*: Actualiza las coordenadas de r1 al agregarle r2. (Solo las actualiza, no añade r2 a r1).
- *int partitionX(Node *header,int inicio,int final)*: Funcion auxiliar para quicksort, eje X.
- *int partitionY(Node *header,int inicio,int final)*: Funcion auxiliar para quicksort, eje Y.
- *void quicksort(Node *header,int inicio,int final,int d)*: Quicksort para rectángulos.
- *Rectangle **makeRandom(Node pNode)*: Desordena el orden de los rectángulos de un nodo para aleatorizar el split.
- *void printRectangle(Rectangle *r)*: Imprime información de un rectángulo.
- *Rectangle **calculateXRectangles(Node *pNode)*: Calcula los rectangulos con mayor bajo y menor alto en un arreglo para el eje X e Y.
- *int *calculateBounds(Node *pNode)*: Calcula el rectángulo más grande de todos los que están en el nodo.
- *int randomNum(int max)*: Retorna un número aleatorio acotado.
- *Rectangle *generateRandomRectangle(int n)*: Crea un rectángulo de id n, de tamaño aleatorio.
- *Rectangle *copy(Rectangle *r)*: Copia un rectángulo.
- *Node *search(char *nodeName, Rectangle *rect)*: Busca en el nodo todos los rectángulos que intersectan a *rect.
- *void insertToRootLinear(char *nodeName, Rectangle *r)*: Inserción Linear Split con control de Overflow en Root.
- *void insertToRootGreene(char* nodeName, Rectangle *r)*: Inserción Greene Split con control de Overflow en Root.
- *void insertLinear(char *nodeName , Rectangle *r)*: Inserción Linear Split con control de Overflow en el resto de los nodos.

- *void insertGreene(char *nodeName, Rectangle *r)*: Inserción Greene Split con control de Overflow en el resto de los nodos.
- *Rectangle ** linearSplit(Node *header)*: Control de overflow usando Linear Split.
- *Rectangle ** greeneSplit(Node *header)*: Control de overflow usando Greene Split.

3. Presentación de los Resultados

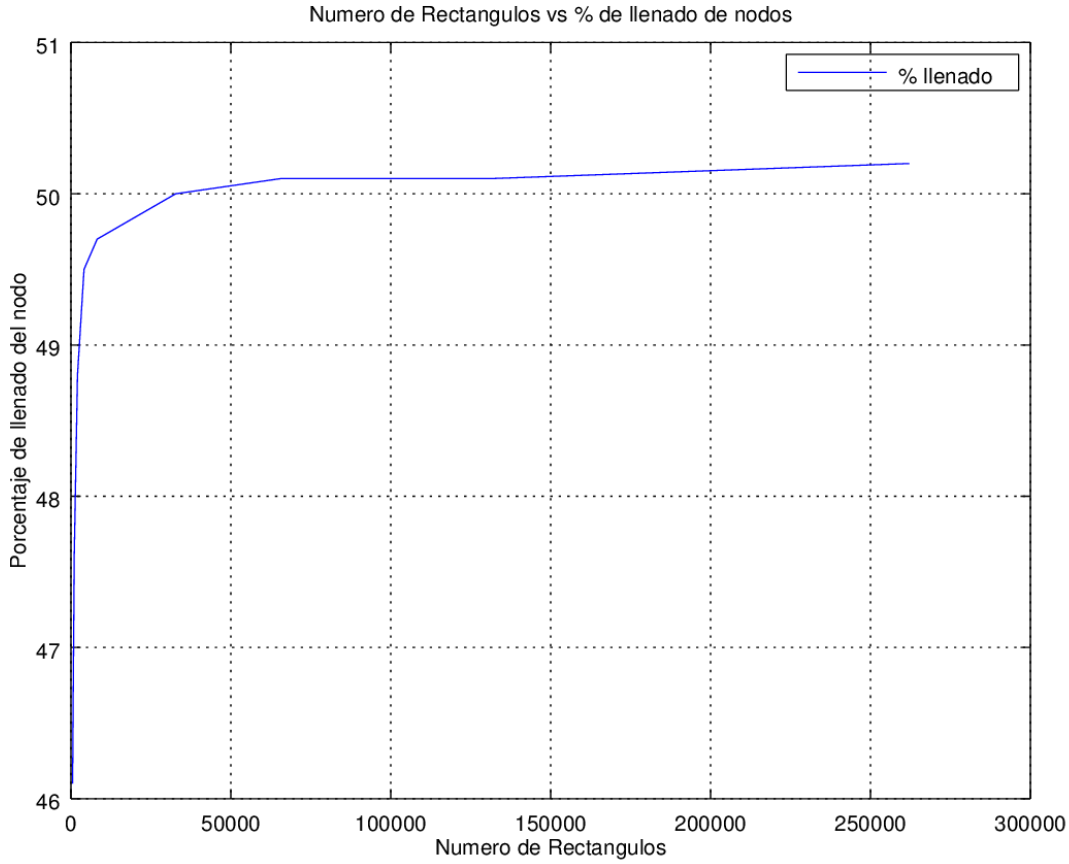
3.1. Tiempo de Inserción del R-Tree

Rectángulos	Tiempo (segs) (Linear)	Tiempo (segs) (Greene)
2^9	0.090544	0.079349
2^{10}	0.164019	0.134480
2^{11}	0.298953	0.260028
2^{12}	0.521460	0.534319
2^{13}	0.966952	0.963934
2^{14}	2.704462	2.247448
2^{15}	5.425123	4.912580
2^{16}	9.942731	9.818921
2^{17}	19.675351	19.402311
2^{18}	38.226344	38.643265



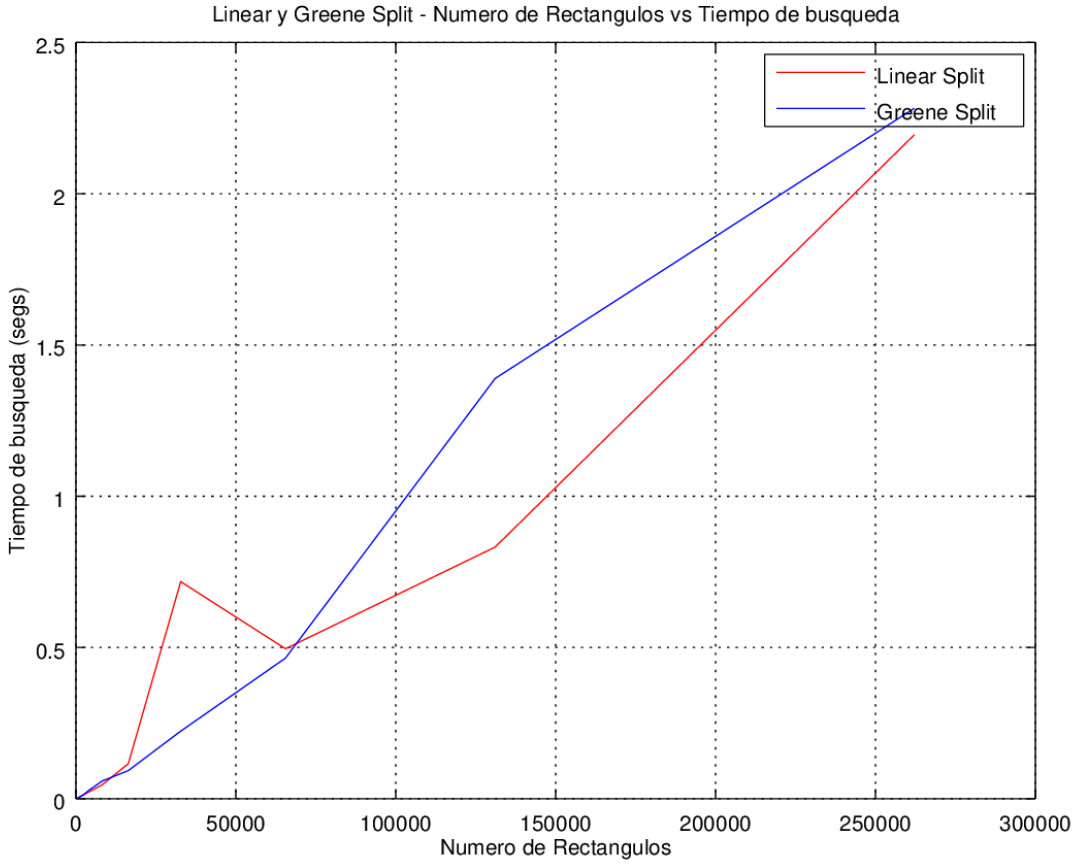
3.2. Espacio ocupado y porcentaje de llenado de páginas de disco

Rectángulos	Nodos	Tamaño Total	Tamaño Promedio Nodo	% llenado	Accesos a disco
2^9	18	16.5 KB	956 bytes	46,1 %	102
2^{10}	34	136 KB	972 bytes	47,6 %	204
2^{11}	68	272 KB	982 bytes	48,8 %	408
2^{12}	134	536 KB	987 bytes	49,5 %	818
2^{13}	266	1.1 MB	990 bytes	49,7 %	1638
2^{14}	530	2.1 MB	992 bytes	49,8 %	3276
2^{15}	1058	4.2 MB	994 bytes	50,0 %	6552
2^{16}	2116	8.4 MB	995 bytes	50,1 %	13106
2^{17}	4240	17 MB	996 bytes	50,1 %	26214
2^{18}	8458	34 MB	996 bytes	50,2 %	52428



3.3. Desempeño de operación *Buscar*

Rectángulos	Tiempo (seg) (Linear)	Tiempo (seg) (Greene)
$2^9/10$	0.002877	0.002891
$2^{10}/10$	0.005631	0.005252
$2^{11}/10$	0.011399	0.010876
$2^{12}/10$	0.022630	0.028296
$2^{13}/10$	0.045871	0.058527
$2^{14}/10$	0.116145	0.093235
$2^{15}/10$	0.717755	0.223559
$2^{16}/10$	0.495848	0.465250
$2^{17}/10$	0.832019	1.389346
$2^{18}/10$	2.194899	2.282709



4. Análisis y Conclusiones

4.1. Control de Overflow

En nuestra implementación, tanto Linear Split como Greene Split toman tiempos muy parecidos, lineal con respecto al número de rectángulos.
Esto es, para al duplicar el número de rectángulos, se duplica también el tiempo, con una ventaja despreciable de Linear Split al inicio y de Greene Split al final.

4.2. Buscar

Un arreglo con control de Overflow Linear Split permite buscar ligeramente más rápido que uno con control de Overflow Greene Split, dado que comienzan a diverger en $n = 2^{16} = 65536$ rectángulos.

4.3. Conclusiones

El trabajo de un algoritmo en memoria secundaria es considerablemente más lento que uno en memoria principal. Sin embargo, a menudo se requiere trabajar con una cantidad de datos muy alta, lo que hace que cargarlos todos en memoria principal sea inviable.
Para este caso, los experimentos realizados hacen que los rectángulos insertados y buscados ocupen tanta memoria que se requiere leer y escribir de disco.

- Según el análisis, tanto Linear Split como Greene Split muestran resultados parecidos, con una muy ligera ventaja de Linear Split.
- La búsqueda produce resultados un poco más apreciables, donde se muestra la ventaja que consiguió Linear Split.
- El uso de C facilitó el manejo preciso de memoria, pero aumentó considerablemente la complejidad para el algoritmo de uso correcto de la memoria, tanto principal como secundaria.