CC4102/CC53A - Diseño y Análisis de Algoritmos
Auxiliar 2: B-Tree y R-Tree

Prof. Gonzalo Navarro
Aux. Teresa Bracamonte

10 de Septiembre, 2013

1 B-Tree[1]

Un *B-Tree* de orden m es un árbol que satisface las siguientes propiedades:

i) Cada nodo tiene $\leq m$ hijos.

ii) Cada nodo, excepto el nodo raíz y las hojas, tiene $\geq m/2$ hijos.

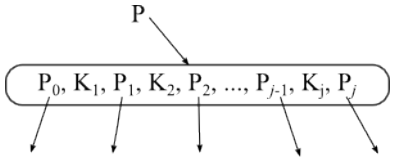
iii) Todas las hojas se encuentran en el mismo nivel y no contienen información.

iv) Un nodo que no es hoja y tiene k hijos contiene $k-1$ llaves.

Un **nodo hoja** es un nodo terminal, un nodo sin hijos. Como las hojas no contienen información, podemos verlos como nodos externos los cuales no se encuentran realmente en el árbol, es decir Λ es un puntero a una hoja.

La Fig. 1, muestra un $B-Tree$ de orden 7. Cada nodo intermedio tiene entre $\lceil 7/2 \rceil$ y 7 nodos hijos, es decir, contiene entre 3 y 6 llaves. El nodo raíz, puede contener entre 1 y 6 llaves; en el ejemplo, tiene 2. Todas las hojas se encuentran en el nivel 3. Note que (a) las llaves aparecen en orden ascendente; y (b) el número de hojas es mayor que el número de llaves en una unidad.

Un **nodo no hoja** que contiene j llaves y $j+1$ punteros puede representarse como:



(1)

donde $K_1 < K_2 < \ldots < K_j$ y P_i punteros a los sub-árboles para las llaves entre K_i y K_{i+1}.

Por lo tanto, **buscar un elemento** en un $B-Tree$ es un proceso directo: Luego que el nodo (1) ha sido llevado a memoria, se busca el argumento entre las claves $K_1, K_2, \ldots, K_j$ (cuando j es grande, se recomienda realizar búsqueda binaria; caso contrario, la búsqueda secuencial es suficiente). Si la búsqueda es exitosa, se encuentra la llave deseada; pero si la búsqueda fracasa porque el argumento está entre K_i y K_{i+1}, se lleva a memoria el nodo indicado por P_i y se continua el proceso. El

[1] D. Knuth, *The Art of Computer Programming, Vol. 3*

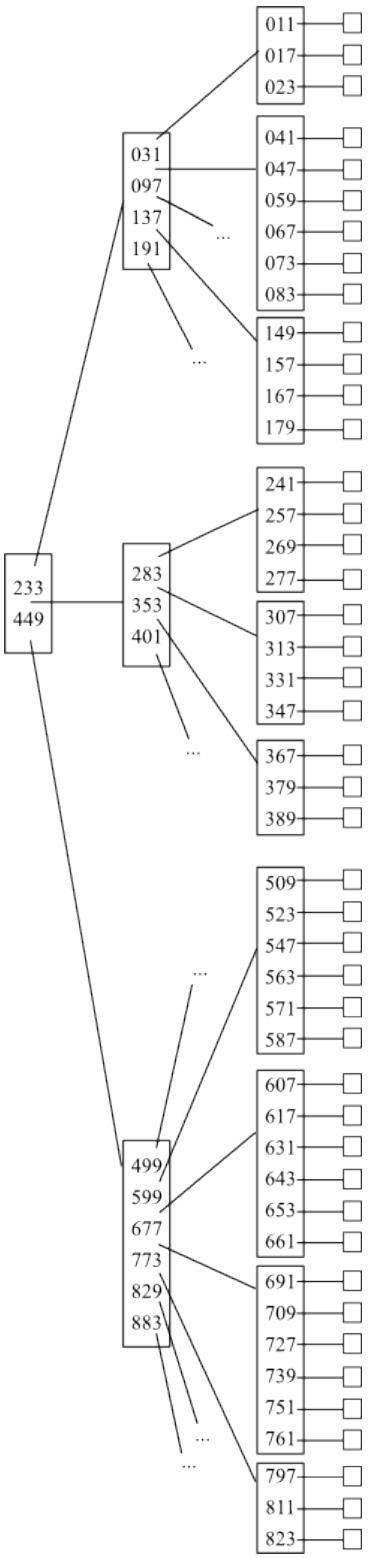
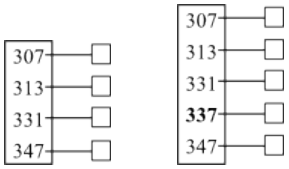


Figure 1: Ejemplo: $B - Tree$ de orden 7.

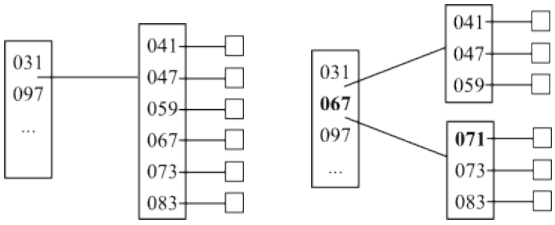
puntero P_0 se usa su el argumento es menor a K_1, y P_j s el argumento es mayor que K_j. Si $P_i = \Lambda$, la búsqueda fracasa.

Insertar un elemento en un $B-Tree$ también es un proceso sencillo. Considerando el árbol de la Fig. 1, cada hoja corresponde a un lugar donde puede insertarse un nuevo elemento. Si se desea insertar la llave 337, simplemente se cambia el nodo correspondiente de la siguiente manera:



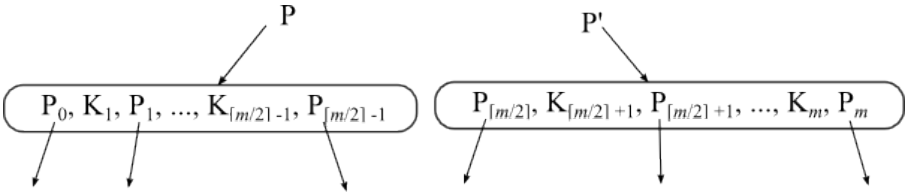
(2)

Por otro lado, si se quiere insertar la llave 071, no hay espacio disponible porque el nos correspondiente en el nivel 2 está lleno. Para este caso, el nodo se debe dividir en dos partes, con tres llaves en cada parte, y pasando la clave del medio al nivel 1:



(3)

En general, para insertar un nuevo elemento en un $B-Tree$ de orden m, cuando las hojas están en el nivel l, se inserta la nueva clave en el nodo apropiado en el nivel $l-1$. Si dicho nodo contiene m llaves, es decir tiene la forma de (1) con $j = m$, el nodo se divide en dos



(4)

y se inserta la llave $K_{\lceil m/2 \rceil}$ en el padre del nodo original. El puntero P en el nodo padre es reemplazado por la secuencia $P, K_{\lceil m/2 \rceil}, P'$. Esta inserción pude causar que el nodo padre tenga m llaves, en ese caso, se debe dividir el nodo de la misma forma. Si se necesita dividir el nodo raíz, se crea un nuevo nodo raíz con la llave $K_{\lceil m/2 \rceil}$. El árbol se vuelve un nivel más grande en este caso.

Cota superior de desempeño: Veamos cuantos nodos se deben acceder en el peor caso, mientras se busca en un $B-Tree$ de orden m. Suponga que hay N llaves y que hay $N+1$ hojas en el nivel l. Entonces, el número de nodos en los niveles $1, 2, 3, \ldots$ es de al menos $2, 2\lceil m/2 \rceil, \lceil m/2 \rceil^2, \ldots$

$$N + 1 \geq 2\lceil m/2 \rceil^{l-1} \tag{5}$$

Equivalente a,

$$l \leq 1 + \log_{\lceil m/2 \rceil} \left(\frac{N+1}{2} \right) \tag{6}$$

Por ejemplo, si $N = 1,999,998$ y $m = 199$, entonces $l \leq 3$. Dado que es necesario acceder a lo más l nodos durante la búsqueda, se garantiza que los tiempos de ejecución son pequeños.

Cuando se inserta un nodo, se podría necesitar dividir hasta l nodos. Sin embargo, el número de nodos promedios que se necesita dividir es mucho menor, dado que el número de divisiones mientras se construye el árbol es una unidad menor que el número de nodos en el árbol. Si hay p nodos, y al menos $1 + (\lceil m/2 \rceil - 1)(p - 1)$ llaves, entonces

$$p \leq 1 + \frac{N - 1}{\lceil m/2 \rceil - 1} \tag{7}$$

El número promedio de veces que un nodo se divide es menor que $1/(\lceil m/2 \rceil - 1)$ por inserción.

B^+ -Tree

Hay muchas formas de mejorar la estructura básica de los $B - Tree$, rompiendo algunas reglas. En primer lugar, vemos que todos los punteros de los nodos del nivel $l - 1$ son Λ y ninguno de los punteros en los otros niveles es Λ. Esto representa una gran cantidad de espacio desperdiciado, así que podemos ahorrar tiempo y espacio eliminando todos los Λ y usando un valor diferente de m para todos los nodos del "fondo".

Usar dos m diferente no arruina el algoritmo reinserción, ya que ambas mitades de un nodo que esta siendo dividido permanecen en el mismo nivel que el nodo original. De hecho, podemos definir un $B-Tree$ generalizado de ordenes $m_1, m_2, \ldots$ haciendo que todos los nodos no raíz del nivel $l-i$ tengan entre $m_i/2$ y m_i hijos; dicho $B-Tree$ tiene diferentes m en cada nivel pero el algoritmo de inserción funciona esencialmente igual que antes.

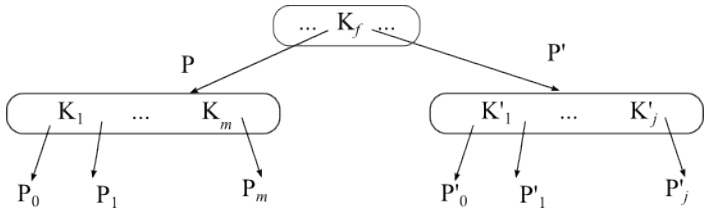
Llevando la idea anterior más allá, podemos usar un formato de nodo completamente distinto en cada nivel del árbol, y podríamos guardar información en las hojas. A veces las llaves son sólo una pequeña parte de los registros en un archivo, y en tales casos es un error guardar los registros enteros en los nodos intermedio cerca de la raíz del árbol. Esto haría m muy pequeño para las ramificaciones.

Podemos reconsiderar la figura del ejemplo, imaginando que todos los registros del archivo están ahora guardados en las hojas y que solo unas cuantas de las llaves han sido duplicadas en los nodos rama. Bajo esta interpretación, la hoja de más a la izquierda contiene todos los registros cuya clave es ≤ 011; la hoja marcada contiene todos los registros cuya clave satisface $439 < K \leq 449$ y así en adelante. Bajo esta interpretación, los nodos hoja crecen y se parten igual que los nodos rama, excepto que un registro nunca pasa de una hoja al siguiente nivel. Las hojas siempre tienen llena al menos la mitad de su capacidad. Una nueva llave se copia en un nodo intermedio siempre que una hoja se parte. Si cada hoja está unida a su sucesor en orden simétrico (ascendente) podemos recorrer el archivo secuencial y aleatoriamente de manera eficiente.

B^* -Tree

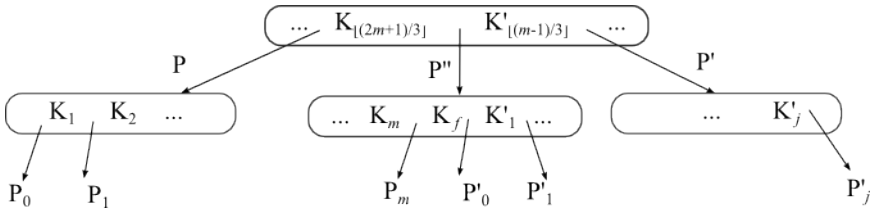
Otra modificación importante al esquema básico del $B - Tree$ es la idea del *overflow* presentada por Bayer y McCreight. La idea es mejorar el algoritmo de inserción resistiendo la tentación e partir nodos muy seguido; en vez de ello se usa una rotación local. Supongamos que tenemos un nodo que está lleno porque tiene m llaves y $m + 1$ punteros; en vez de partirlo, miramos a su nodo hermano a la derecha que tiene j llaves y $j + 1$ punteros. En el nodo padre hay una llave K_f que separa las

llaves de los dos hermanos esquemáticamente.



(8)

Si $j < m-1$, un reordenamiento simple hace que partir sea innecesario: dejamos $\lfloor (m+j)/2 \rfloor$ llaves en le nodo izquierdo, reemplazamos K_f por $K_{\lfloor (m+j)/2 \rfloor + 1}$ en el nodo padre, y ponemos las $\lceil (m+j)/2 \rceil$ llaves restantes (incluyendo K_f) y los punteros correspondientes en el nodo de la derecha. El nodo entero "fluye" hacia su nodo hermano. Por otro lado, si el nodo hermano ya esta lleno ($j = m-1$) podemos partir amos nodos teniendo 3 nodos 2/3 llenos conteniendo respectivamente $\lfloor (2m-2)/3 \rfloor$, $\lfloor (2m-1)/3 \rfloor$ y $\lfloor 2m/3 \rfloor$:



(9)

Si el nodo original no tiene hermano a la derecha, podemos ver al nodo izquierdo de la misma manera (si el nodo original tiene hermanos a la izquierda y la derecha, podríamos incluso evitar parto en un nuevo nodo a menos que ambos hermano están llenos). Finalmente, si el nodo original no tiene hermanos, debe ser la raíz; podemos cambiar la definición de $B-Tree$ permitiendo que la raíz contenga hasta $2\lfloor (2m-2)/3 \rfloor$ llaves, para que cuando la raíz se parta produzca dos nodos de $\lfloor (2m-2)/3 \rfloor$ llaves. El efecto de lo explicado anteriormente es producir una clase superior de árbol, $B*-Tree$ de orden m, que se define a continuación:

i) Cada nodo excepto la raíz tiene como máximo m hijos.

ii) Cada nodo, excepto la raíz y las hojas, tiene $\geq (2m-1)/3$ hijos.

iii) La raíz tiene, al menos 2 y como máximo $2\lfloor (2m-2)/3 \rfloor + 1$ hijos.

iv) Todas las hojas están en el mismo nivel.

v) Un nodo intermedio con k hijos tiene $k-1$ llaves.

El cambio importante es la condición *ii*) que indica que usamos al menos 2/3 del espacio disponible en cada nodo. Este cambio no sólo usa el espacio más eficientemente, sino que también hace que el proceso de búsqueda sea más rápido. Ya que podemos reemplazar $\lfloor m/2 \rfloor$ por $\lfloor (2m-1)/3 \rfloor$ en (6) y (7) .

La raíz de los $B-Tree$ puede tener un grado tan bajo como 2. Porque usar acceso a disco por una decisión binaria? Un esquema simple de *buffering* llamado *least-recently-used page replacement* soluciona este problema. Podemos mantener en memoria varias páginas del *buffer* para que los comandos de entrada puedan evitarse si la página solicitada ya está en memoria. En este esquema

el algoritmo de búsqueda o inserción usa comandos de "lectura virtual" que se traducen en instrucciones de entrada reales sólo cuando la página necesaria no esta en memoria. Un comando posterior de liberación es enviado cuando el *buffer* ha sido leído y posiblemente modificado por el algoritmo. Cuando una lectura real es necesaria, se escoge el buffer más antiguo en memoria (el último en la lista de accesos recientes), y se escribe en ese *buffer* si su contenido ha variado desde que se leyó. Luego leemos la página en el *buffer* escogido.

Como el número de niveles del árbol es generalmente pequeño en comparación con el número de buffer, el esquema de paginación nos asegura que la página raíz siempre estará presente en memoria; y que si la raíz tiene sólo 2 o 3 hijos, las páginas del primer nivel probablemente permanecerá ahí también. Podemos incorporar algunos mecanismo especial para asegurar que un número mínimo de páginas cerca de la raíz están siempre presentes. El esquema *least-recently-used* implica que las páginas que podrían partirse durante una inserción están automáticamente en memoria cuando se necesitan.

Ejercicios

1. Insertar la llave 613 en el $B-Tree$ de orden 7 del ejemplo.

2. Diseñe un algoritmo para eliminar un elemento en un $B-Tree$.

2 R-Tree[2]

Los $R-Tree$ son estructuras de datos jerárquicas basadas en $B^+-Trees$ que son usadas para organizar dinámicamente un conjunto de objetos geométricos de dimensión d. Cada objeto geométrico se representa usando rectángulos d-dimensionales que se ajustan al objeto (MBR, *minimum bounding rectangles*). Cada nodo del $R-Tree$ representa el MBR que delimita a sus hijos. Los nodos hoja del árbol contienen punteros a la base de datos de objetos en lugar de punteros a los nodos hijos. Normalmente, lo nodos son implementados como páginas de disco.

Nótese que los MBRs que comprenden diferentes nodos se pueden solapar entre si. Además, un MBR puede estar incluido (geométricamente) en el área de varios nodos, pero sólo se encuentra asociado a uno. Esto implica que la búsqueda espacial podría visitar varios nodos antes de confirmar la existencia de un MBR dado. Además, la representación de objetos geométricos mediante MBRs puede generar falsos positivos. Para evitar esto, los objetos candidatos deben ser examinados. La figura siguiente muestra un ejemplo donde dos polígonos no se interceptan, pero sus MBR's sí. El $R-Tree$ juega el rol de un mecanismo de filtrado para reducir el costo de examinar directamente la geométrica de los objetos.

Un $R-Tree$ de orden (m, M) tiene las siguientes características:

i) Cada nodo hoja (a menos que sea la raíz) puede tener hasta M elementos, siendo el mínimo número de entradas $m \geq M/2$. Cada entrada es de la forma (mbr, oid), tal que mbr es el MBR que contiene espacialmente el objeto y oid es el identificador del objeto.

ii) El número de entradas que cada nodo interno puede almacenar está entre $m \geq M/2$ y M. Cada entrada es de la forma (mbr, p), donde p es un puntero al nodo hijo y mbr es el MBR que contiene los MBR's de los hijos.

[2] A. Guttman, *R-Trees: A Dynamic Index Structure for Spatial Searching, 1984*

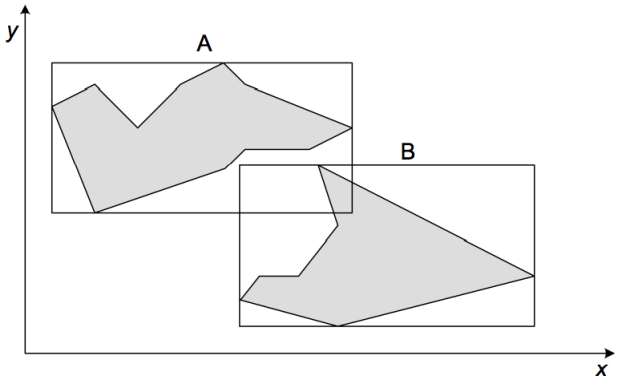


Figure 2: Ejemplo: Objetos geométricos y sus respectivos MBRs.

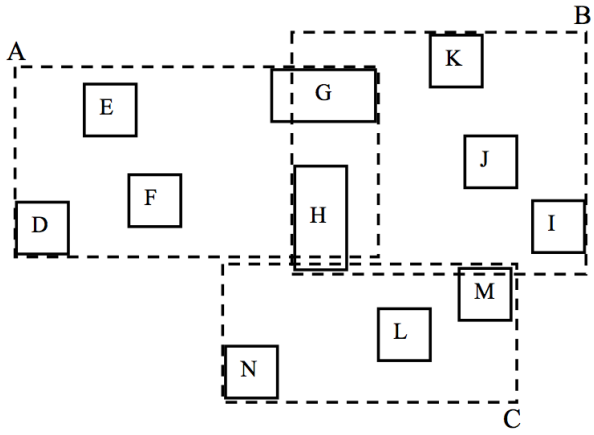


Figure 3: Ejemplo: MBRs organizados jerárquicamente.

iii) El número mínimo de entradas en el nodo raíz es 2, a menos que sea un hoja.

iv) Todas las hojas del $R-Tree$ está en el mismo nivel.

La siguiente figura muestra un conjunto de MBR's de objetos geométricos. Los MBR's D, E, F, G, H, I, J, K, L, M y N se almacenan en los nodos hojas del $R-Tree$. Y los MBRs A, B, y C son nodos internos que organizan los MBR's de las hojas.

Asumiendo que $M = 4$ y $m = 2$, la siguiente figura muestra los correspondientes MBR's. Dado que varios $R-Tree$ pueden representar el mismo conjunto de rectángulos, el $R-Tree$ resultante depende del orden en que se insertan/eliminan las entradas.

Sea un $R-Tree$ que almacena N rectángulos, su altura máxima h es:

$$h_{max} = \lceil \log_m N \rceil - 1 \tag{10}$$

El máximo número de nodos puede ser derivado sumando el máximo número de nodos por nivel. Este numero se obtiene cuando todos los nodos contienen el mínimo número de elementos

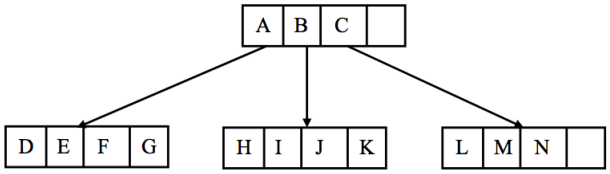


Figure 4: Ejemplo: MBRs organizados en un $R-Tree$.

Algorithm RangeSearch(TypeNode RN**, TypeRegion** Q**)**
/* Finds all rectangles that are stored in an R-tree with root node RN, which are intersected by a query rectangle Q. Answers are stored in the set $\mathcal{A}$ */

1. **if** RN is not a leaf node
2. examine each entry e of RN to find those $e.mbr$ that intersect Q
3. **foreach** such entry e call RangeSearch($e.ptr$,Q)
4. **else** // RN is a leaf node
5. examine all entries e and find those for which $e.mbr$ intersects Q
6. add these entries to the answer set $\mathcal{A}$
7. **endif**

Figure 5: Algoritmo: Búsqueda en $R-Tree$.

por entrada, i.e. m.

$$\sum^{h_{max}} i = 1\lceil N/m^i \rceil = \lceil N/m \rceil + \lceil N/m^2 \rceil + \ldots + 1 \tag{11}$$

La estructura de los $R-Tree$ permite responder consultas de rango. Es decir, dado un rectángulo Q, encontrar todos los rectángulos que se intersectan con Q. En el algoritmo de búsqueda que se muestra en la figura siguiente, los rectángulos que se encuentran mediante la búsqueda por rango (en nodos internos) corresponden al conjunto de candidatos del proceso de filtrado. El conjunto real de objetos interceptados por la consulta se encuentran en el paso de refinamiento (para nodos hoja).

La inserción en un $R-Tree$ es similar a la inserción en un B^+-Tree. Se recorre el $R-Tree$ para encontrar un nodo hoja en el que se pueda agregar el nuevo elemento. El elemento se inserta, y todos los nodos en el camino de la raíz a la hoja se actualizan. En caso la hoja se encuentre llena, se divide en dos nodos. Guttman propone tres alternativas para dividir los nodos:

- Particionamiento Lineal: Seleccionar dos MBR como semilla para los nodos, tal que estos se encuentren lo más alejado posible. Luego, tomar cada elemento restante en orden aleatorio, y agregarlo al nodo cuyo MBR requiera el menor crecimiento.

- Particionamiento Cuadrático: Seleccionar dos MBR como semilla para los nodos, tal que si estos objetos se juntan generan tanto "espacio muerto" como es posible (espacio muerto es el espacio que queda del MBR si el área de estos objetos es ignorada). Luego, para cada uno de los objetos restantes, insertar el objeto para el que la diferencia de espacio muerto asignada a cada uno de los nodos se maximiza en el nodo que requiere el menor crecimiento de su MBR.

Algorithm Insert(TypeEntry E, TypeNode RN)
/* Inserts a new entry E in an R-tree with root node RN */

1. Traverse the tree from root RN to the appropriate leaf. At each level, select the node, L, whose MBR will require the minimum area enlargement to cover $E.mbr$
2. In case of ties, select the node whose MBR has the minimum area
3. **if** the selected leaf L can accommodate E
4. Insert E into L
5. Update all MBRs in the path from the root to L, so that all of them cover $E.mbr$
6. **else** // L is already full
7. Let $\mathcal{E}$ be the set consisting of all L's entries and the new entry E Select as seeds two entries $e_1, e_2 \in \mathcal{E}$, where the distance between e_1 and e_2 is the maximum among all other pairs of entries from $\mathcal{E}$ Form two nodes, L_1 and L_2, where the first contains e_1 and the second e_2
8. Examine the remaining members of $\mathcal{E}$ one by one and assign them to L_1 or L_2, depending on which of the MBRs of these nodes will require the minimum area enlargement so as to cover this entry
9. **if** a tie occurs
10. Assign the entry to the node whose MBR has the smaller area
11. **endif**
12. **if** a tie occurs again
13. Assign the entry to the node that contains the smaller number of entries
14. **endif**
15. **if** during the assignment of entries, there remain λ entries to be assigned and the one node contains $m - \lambda$ entries
16. Assign all the remaining entries to this node without considering the aforementioned criteria /* so that the node will contain at least m entries */
17. **endif**
18. Update the MBRs of nodes that are in the path from root to L, so as to cover L_1 and accommodate L_2
19. Perform splits at the upper levels if necessary
20. In case the root has to be split, create a new root
21. Increase the height of the tree by one
22. **endif**

Figure 6: Algoritmo: Inserción en $R - Tree$.

- Particionamiento Exponencial: Todos los agrupamientos posibles son evaluados y se escoge el que minimice el creciendo de los MBR.

Ejercicios

1. Diseñe un algoritmo para eliminar un elemento de un $R - Tree$.