ÁRBOLES B

TRABAJO PRIMER CUATRIMESTRE MARP

ÓSCAR DÍAZ RIBAGORDA

ÍNDICE

INTRODUCCIÓN

Para este trabajo, se ha realizado la implementación del tipo de datos Árbol-B. En este documento se explicará el cómo y el porqué de esta implementación.

La implementación se adjunta con el nombre de “ArbolB.h” además del archivo auxiliar “NodoB.h” y un archivo .cpp con un código de prueba.

ÁRBOL-B

El árbol-B consiste en un tipo de datos arborescente cuyos nodos pueden tener cualquier número de hijos cumpliendo las siguientes reglas:

- Para cada t, el número de claves por nodo posible de un árbol-B, está entre t-1 y 2*t – 1. Por tanto el número de claves de cada nodo está acotado para cada árbol. El valor t representa el número mínimo de hijos de cada nodo.
- Todos los nodos hoja se encuentran a la misma altura.

Con estas condiciones se consiguen árboles balanceados cuya altura se reduce mucho si se toma una t muy grande. Esto será muy útil cuando la memoria se guarda, por ejemplo, en discos rígidos, ya que se puede reducir la altura para que quepa en los bloques de la memoria.

Las funciones más relevantes en los árboles-B son la búsqueda, la inserción y el borrado.

Estas funciones son las que han sido implementadas en este trabajo.

BÚSQUEDA

La única diferencia de la búsqueda en un árbol-B respecto de un árbol binario es que, en vez de elegir cuál de las 2 ramas se escoge para buscar, dado un nodo, se busca el hijo del nodo-B en el que debería estar el valor. Por tanto, en cada etapa de la recursión se tomará una decisión en función del número de claves del nodo.

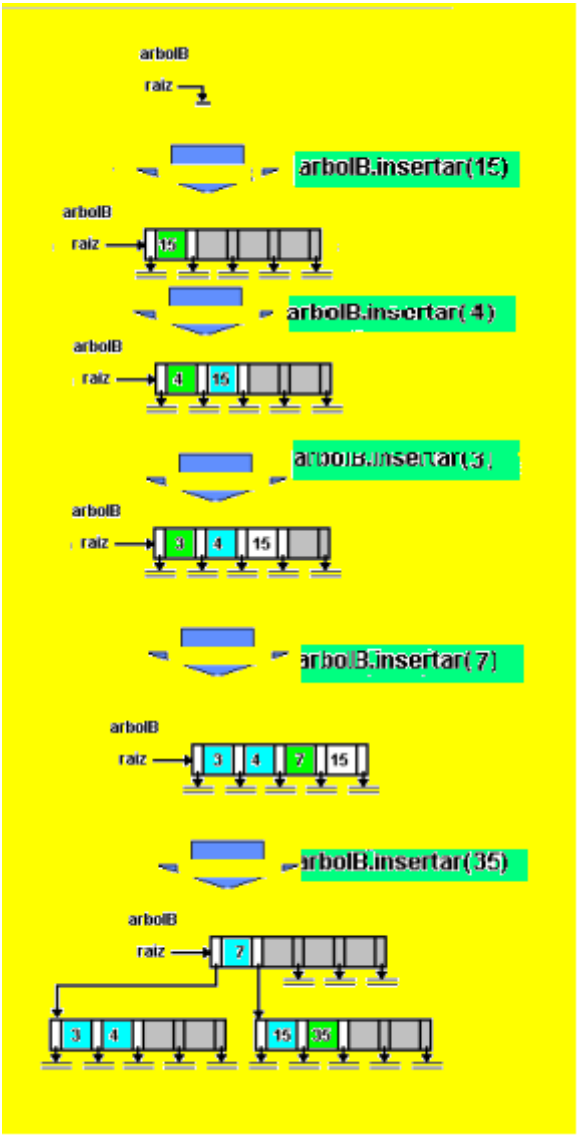
Se busca recursivamente en el hijo correspondiente del nodo y en caso de que el nodo sea un nodo nulo, quiere decir que el elemento no se ha encontrado y se para la recursión (en este programa devuelve un puntero nulo). En caso contrario, este programa devuelve un puntero al elemento que se estaba buscando.

Para saber si un elemento está o no en el árbol, basta con buscar en el árbol y comprobar si el valor devuelto es nulo o no.

Para ver el coste de la búsqueda, observamos que el número de llamadas recursivas es O(log n) y se puede comprobar que este logaritmo, tiene base t. Además por cada llamada recursiva, se busca el hijo correspondiente con el cual continuar la recursión, que tiene coste O(t). Por tanto, el coste total del algoritmo será **O(t * log n)** siendo el logaritmo, un logaritmo de base t.

INSERCIÓN

La inserción en un árbol-B sí que es más complicada que en un árbol binario. Hay varias maneras de hacerla, pero en esta implementación se ha decidido insertar preparando el árbol antes de insertar de manera que no halla que reequilibrar tras la inserción.



La idea en esta inserción es que un elemento siempre se podrá insertar en un nodo incompleto sin necesidad de reequilibrar tras ella. Además sabemos que la inserción siempre se realiza en los nodos hoja. Como no podemos estar seguros de que un nodo hoja vaya a estar incompleto, implementaremos un algoritmo que, a la vez que busca el nodo hoja donde hay que insertar, asegura que el nodo al que se desciende, está incompleto.

Por tanto, nuestro algoritmo, comenzará en la raíz y buscará el nodo por el que tiene que descender para la inserción. Si este nodo no está incompleto, se aplicará la función separarHijo.

La función separarHijo, coge un hijo completo y lo divide en 2 con el número mínimo de nodos en cada uno y la mediana de este hijo, pasa a ser una nueva clave de su padre. Así el padre tendrá un hijo y una clave más posicionados en el lugar correcto. Y además, el hijo donde debo buscar ya será incompleto (el hijo por el que debía bajar, ha sido dividido en 2 hijos incompletos). Además la función separarHijo no añade altura al árbol por tanto sabemos que el árbol resultante seguirá siendo un árbol-B.

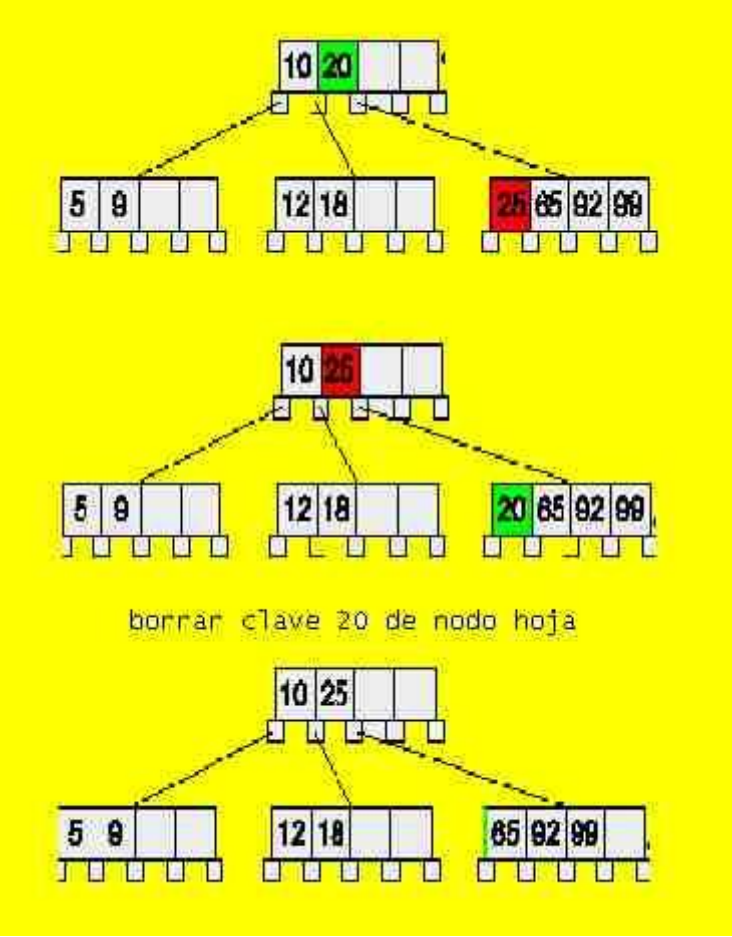
El proceso de separarHijo se aplicará primero a la raíz en caso necesario y según se va bajando, a cada nodo completo con quien se tope. El algoritmo acabará cuando se llegue a un nodo hoja, para el cual, podemos estar seguros que está incompleto y entonces podemos insertar sin problemas.

El coste de la inserción está en función de lo que cuesta separarHijo y el número de llamadas recursivas que se hacen hasta llegar a la hoja. En este caso, separarHijo tiene un coste de O(t) ya que para separar el hijo, se recorre en bucles una cantidad equivalente a t claves. Y las llamadas recursivas dependerán de la altura que, en el caso del árbol-B, es O(log n) (log base t). Por tanto, el coste total del algoritmo es **O(t * log n)** (log base t).

BORRADO

Al igual que la inserción, el borrado también es más complicado que en los árboles binarios habituales y de nuevo hay que preocuparse de que el árbol se mantenga equilibrado.

La idea que hemos tomado para el borrado en esta implementación es parecida a la de la inserción pero más complicada ya que el borrado se puede realizar de un elemento que no esté en una hoja. Entonces nos tenemos que preocupar que al borrar un elemento de un nodo intermedio, el nodo se reequilibre de manera que no se quede un hijo colgando.



Además, nos tenemos que asegurar de que al eliminar no se quede un nodo con menos elementos que el mínimo posible. Para poder asegurar esto sin tener que reequilibrar tras el borrado, el algoritmo que se implementa obliga a que el hijo al que se descienda tenga al menos uno más que el mínimo número de claves. Así cuando se borre, la única preocupación ha de ser la de no perder los hijos correspondientes al elemento eliminado.

Para ver como se implementa el algoritmo, distinguimos 3 casos: que el elemento a borrar esté en el nodo actual y este sea una hoja, que el elemento a borrar esté en el nodo actual y este sea un nodo intermedio o que el elemento a borrar no esté en el nodo actual.

Para el primer caso, si el elemento a borrar está en este nodo y el nodo es hoja, el elemento se borra y ya está. No hay que preocuparse por los hijos porque no tiene.

Para el segundo caso, si el elemento a borrar está en este nodo y el nodo es intermedio, se distinguen otros 3 casos:

- Si el nodo a la izquierda del que se quiere borrar tiene al menos t claves, se coge el predecesor del elemento a borrar (que está en el hijo izquierdo) y se sube cambiándolo con el elemento a borrar. Luego recursivamente se elimina el elemento del hijo en el que lo acabamos de meter. De esta manera nos aseguramos de que el elemento a borrar acabe en una hoja y por tanto se pueda borrar sin problemas.
- Si el nodo de la izquierda no tiene suficientes elementos pero el de la derecha sí, se hace la operación análoga con el hijo de la derecha del elemento a borrar.
- Si ninguno de los hijos tiene suficientes elementos, entonces lo que se hace es unir los dos hijos en uno (que tendrá menos claves que el máximo posible) y de esta manera

podemos quitar el elemento del nodo sin perder uno de los hijos, ya que hemos metido los dos en el mismo.

En el último caso, si el elemento a borrar no está en el nodo, la idea es elegir el hijo por el que tiene que bajar la recursión pero asegurándose de que este hijo tenga al menos t claves. Si tiene al menos t claves, se baja recursivamente, y si hacemos el siguiente arreglo. Distinguimos 3 nuevos casos:

- Si el hijo de la izquierda del que se quiere bajar tiene al menos t claves, la idea es que se pasa una. Para ello, se añade una nueva clave al hijo que será el valor de la clave del padre. Y a la clave del padre se le asigna el último valor de la clave del hijo Izquierdo. Para que no se pierda, el último hijo del hijo Izquierdo, pasa a ser el primer hijo del hijo por el que queremos descender. Así, el hijo izquierdo pierde un elemento, pero no pasa nada porque tenía suficientes y el hijo por el que hay que descender gana un elemento, teniendo así al menos t claves.
- Si en vez del hijo de la izquierda, es el hijo de la derecha el que tiene al menos t claves, se hace el análogo por la derecha del anterior apartado.
- En el caso de que ambos hijos tengan el mínimo de claves, se realiza una operación de unión de nodos igual que la que habíamos hecho antes en el segundo caso cuando los dos hijos no tenían suficientes elementos. Esta unión se hará, en este caso con el hijo por el que se quiere bajar y el hijo a la izquierda de este. Así el padre tendrá un elemento menos pero el hijo por el cual se quería bajar ya tiene suficientes claves y se ejecuta recursivamente la función borrar sobre este hijo.

El coste de este algoritmo de nuevo será **O(t * log n)** (log en base t) ya que el número de llamadas recursivas es la altura del árbol (O(log n) (log en base t)) y el procedimiento aplicado en cada llamada recursiva, aunque complicado, solo tiene coste lineal en t para el caso peor.

CÓDIGO DE PRUEBA

En el código de prueba adjuntado, se prueban 4 casos (se pueden añadir más si se quiere). La salida de los casos de prueba se guarda en el fichero “salida.txt” y la entrada la recibe por “prueba.txt” con el formato: primero el número de nodos, después el valor de las claves en cualquier orden. Para elegir el tipo de Árbol-B, el número de casos de prueba que se hacen y el tipo de prueba que se hace, se tiene que meter en el archivo “PruebaArbolB.cpp” y añadir o cambiar los datos adecuadamente.

Los casos adjuntados son:

Ejemplo 1: Creación de un Árbol-B 3-4-5-6 (número mínimo de hijos es 3) con 100 claves y el borrado de la raíz que se considera como el borrado de un elemento que no es hoja.

Ejemplo 2: Se crea un Árbol-B 2-3-4 (número mínimo de hijos es 2) con 10 elementos y se borra un elemento de la raíz, en este caso, el 10.

Ejemplo 3: Se crea un Árbol-B con número mínimo de hijos 20 con 33 elementos. Por tanto, es un árbol con todos sus elementos en la raíz. De ahí, se borra un elemento cualquiera, en este caso el 13.

Ejemplo 4: Se crea un Árbol-B 2-3-4 (número mínimo de hijos es 2) con 25 nodos y se busca en el árbol el elemento 10 que, como se comprueba, está en el árbol.

REFERENCIAS

- Cormen, capítulo 18
- Sedgewick, capítulo 6.2
- E. Horowitz, S. Sahni & D. Mehta, capítulo 10.6
- Weiss, capítulo 18.7
- es.wikipedia.org/wiki/Árbol-B