CS2023 - Aula de Ejercicios Nº 10

Brenner H. Ojeda Rios

Semestre 2024-0

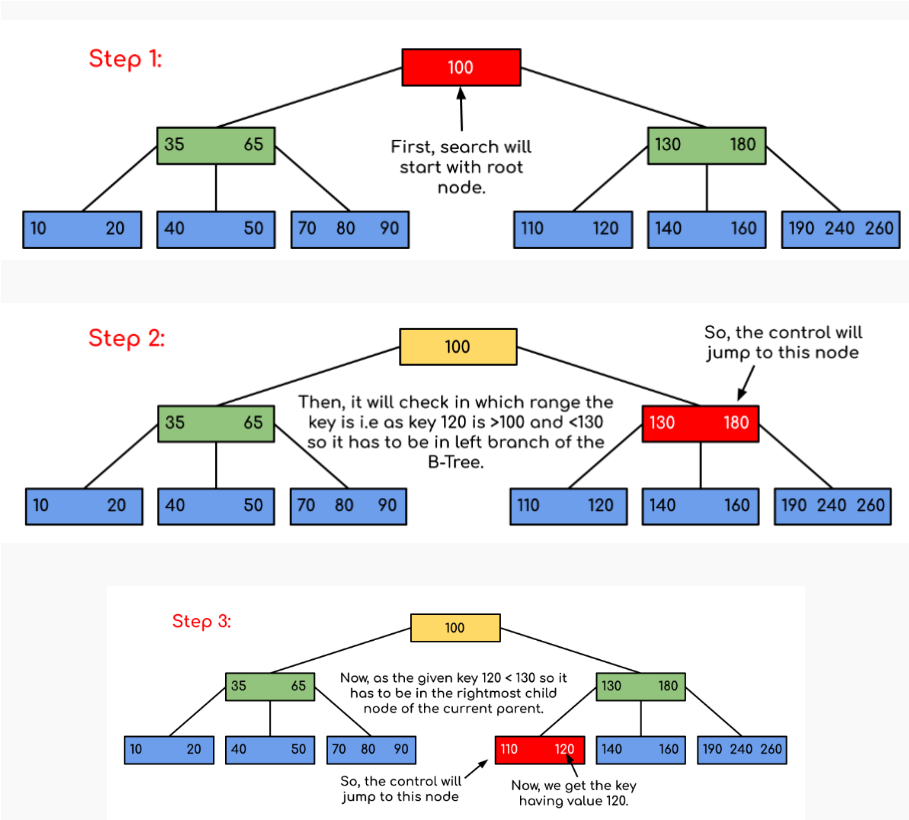
Se sugiere que cada estudiante trate de resolver los ejercicios de forma **individual** y luego los discuta en grupo.

Ejercicios

1. (5 pts) **Recorrido y búsqueda en B-Tree**

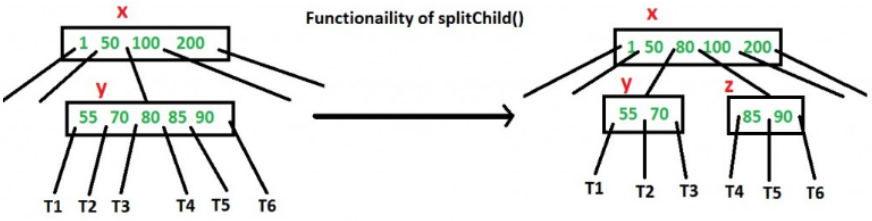
   - (2.5 pts) **Recorrido en árbol B**: El recorrido en un `B-Tree` es similar al recorrido en orden del árbol binario. Comenzamos desde el hijo más a la izquierda, imprimimos recursivamente el hijo más a la izquierda y luego repetimos el mismo proceso para los hijos y claves restantes. Al final, imprimimos recursivamente el hijo más a la derecha.
   - (2.5 pts) **Operación de búsqueda en B-Tree**: Buscar en un árbol B es también es similar a buscar en un árbol binario. El algoritmo es similar y va con recursividad. En cada nivel, la búsqueda se optimiza como si el valor de la clave no estuviera presente en el rango del padre, entonces la clave estaría presente en otra rama. Como estos valores limitan la búsqueda, también se les conoce como valores límite o valores de separación. Si llegamos a un nodo hoja y no encontramos la clave deseada, se mostrará NULL.

   Su tarea es implementar la función recorrido y la función búsqueda en un árbol `B-Tree` **usando la plantilla presentada en este laboratorio**. Vea el siguiente ejemplo de búsqueda de la llave 120.



2. (6 pts) En ejercicio anterior implementamos las funciones de `búsqueda()` y `recorrido()`. En este ejercicio, se implementará la operación `insert()`. Siempre se inserta una nueva clave en el nodo hoja. Deje que la llave a insertar sea k. Al igual que BST, comenzamos desde la raíz y recorremos hacia abajo hasta llegar a un nodo hoja. Una vez que llegamos a un nodo hoja, insertamos la clave en ese nodo hoja. A diferencia de los BST, tenemos un rango predefinido en la cantidad de claves que puede contener un nodo. Entonces, antes de insertar una clave en el nodo, nos aseguramos de que el nodo tenga espacio adicional.

   ¿Cómo asegurarse de que un nodo tenga espacio disponible para una clave antes de insertarla? Usamos una operación llamada `splitChild()` que se usa para dividir un hijo de un nodo. Consulte el siguiente diagrama para comprender la división. En el siguiente diagrama, el hijo y de x se divide en dos nodos y y z. Tenga en cuenta que la operación `splitChild` mueve una clave hacia arriba y esta es la razón por la que los `B-Trees` crecen hacia arriba, a diferencia de los BST que crecen hacia abajo.



   Su tarea es implementar la función `insert()` en un árbol `B-Tree` **usando la plantilla presentada en este laboratorio**.

3. (9 pts) La eliminación de un árbol B es más complicada que la inserción porque podemos eliminar una clave de cualquier nodo (no sólo una hoja) y cuando eliminamos una clave de un nodo interno, tendremos que reorganizar los hijos del nodo.

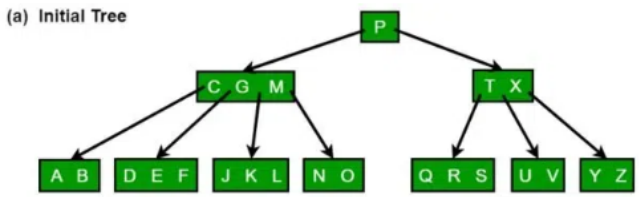
   Al igual que en la inserción, debemos asegurarnos de que la eliminación no viole las propiedades del árbol B. Así como teníamos que asegurarnos de que un nodo no creciera demasiado debido a la inserción, debemos asegurarnos de que un nodo no se vuelva demasiado pequeño durante la eliminación (excepto que a la raíz se le permite tener menos del número mínimo de $t-1$ llaves).

   El procedimiento de eliminación elimina la clave k del subárbol con raíz en x. Este procedimiento garantiza que siempre que se llame a sí mismo de forma recursiva en un nodo x, el número de claves en x sea al menos el grado mínimo t.

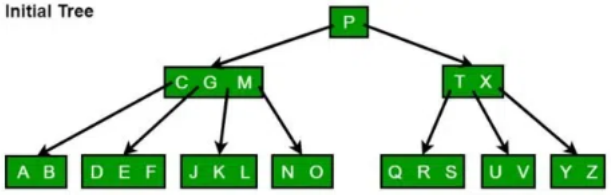
   Casos de eliminación

   - Caso 1: si la clave k está en el nodo x y x es una hoja, elimine la clave k de x.
   - Caso 2: Si la clave k está en el nodo x y x es un nodo interno, haga lo siguiente.
     a. Si el hijo y que precede a k en el nodo x tiene al menos t claves, entonces encuentre el predecesor k_0 de k en el subárbol con raíz en y. Elimine recursivamente k_0 y reemplace k con k_0 en x.
     b. Si y tiene menos de t claves, entonces, simétricamente, examine el hijo z que sigue a k en el nodo x. Si z tiene al menos t claves, entonces encuentre el sucesor k_0 de k en el subárbol con raíz en z. Elimine recursivamente k_0 y reemplace k con k_0 en x.
     c. De lo contrario, si tanto y como z tienen solo $t-1$ claves, fusione k y todo z en y, de modo que x pierda tanto k como el puntero a z, y y ahora contenga $2t-1$ claves. Luego libere z y elimine recursivamente k de y.
   - Caso 3: Sea $x.c(i)$ el i-ésimo hijo de x. Si la clave k no está presente en el nodo interno x, determine la raíz $x.c(i)$ del subárbol apropiado que debe contener k, si es que k está en el árbol. Si $x.c(i)$ tiene solo $t-1$ claves, ejecute los pasos $3.a$ o $3.b$ según sea necesario para garantizar que descendemos a un nodo que contenga al menos t claves. Luego termine recurriendo al hijo apropiado de x.
     a. Si $x.c(i)$ tiene solo $t-1$ claves pero tiene un hermano inmediato con al menos t claves, déle a $x.c(i)$ una clave adicional moviendo una clave de x hacia abajo a $x.c(i)$, moviendo una clave de $x.c(i)$ el hermano inmediato izquierdo o derecho de x, y moviendo el puntero secundario apropiado del hermano a $x.c(i)$.
     b. Si $x.c(i)$ y ambos hermanos inmediatos de $x.c(i)$ tienen $t-1$ claves, fusione $x.c(i)$ con un hermano, lo que implica mover una clave de x hacia abajo al nuevo nodo fusionado para convertirse en la clave mediana para ese nodo.

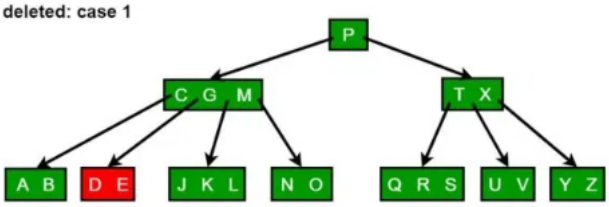
   Dado que la mayoría de las claves de un árbol B están en las hojas, las operaciones de eliminación se utilizan con mayor frecuencia para eliminar claves de las hojas. Vea un ejemplo en la siguiente pagina. Su tarea es implementar la función `remove()` **usando la plantilla presentada en este laboratorio**.



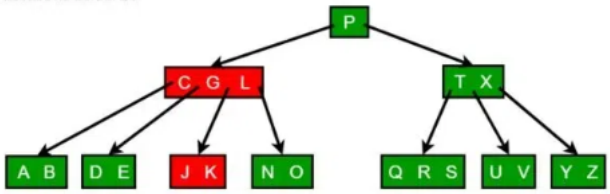
(a) Initial Tree



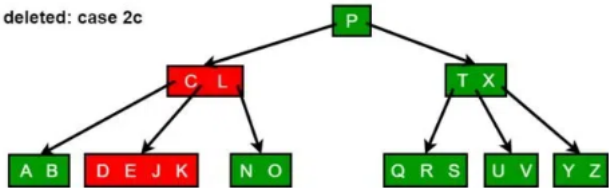
(b) F deleted: case 1



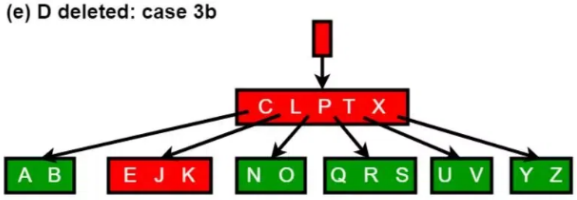
(c) M deleted: case 2a



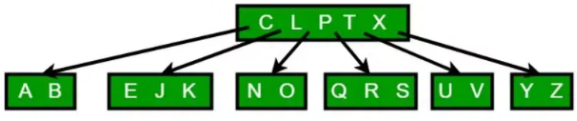
(d) G deleted: case 2c



(e) D deleted: case 3b



(e') tree shrinks in height



(f) B deleted: case 3a