B - Tree

Un árbol B es una estructura de datos de árbol que mantiene datos ordenados y permite búsquedas, inserciones y eliminaciones eficientes en tiempo logarítmico en el peor de los casos. Se caracteriza por tener varios nodos internos y hojas, donde cada nodo puede tener un número variable de claves y un número variable de hijos. Los árboles B están diseñados para ser eficientes en operaciones de disco, ya que minimizan el número de accesos a disco necesarios.

Ahora, hablemos sobre el código y cómo implementa un árbol B:

1. **Clase `BTreeNode`**:
   - Representa un nodo en el árbol B.
   - Tiene tres atributos:
     - `leaf`: Un booleano que indica si el nodo es una hoja o no.
     - `keys`: Una lista que almacena las claves en el nodo.
     - `child`: Una lista que contiene referencias a los hijos del nodo.

2. **Clase `BTree`**:
   - Representa el árbol B en sí.
   - Tiene dos atributos:
     - `root`: El nodo raíz del árbol.
     - `t`: Un parámetro que define el grado mínimo del árbol. El grado mínimo se refiere al número mínimo de claves que puede contener un nodo, que es `t-1`, y al número máximo de claves que puede contener un nodo, que es `2*t-1`.
   - Tiene varios métodos para operaciones básicas en un árbol B:

     - `insert(k)`: Inserta una clave `k` en el árbol.
     - `insert_non_full(x, k)`: Inserta una clave `k` en el nodo `x` que no está lleno.
     - `split_child(x, i)`: Divide el hijo `i` del nodo `x` si está lleno.
     - `delete(k)`: Elimina una clave `k` del árbol.
     - `delete_key(x, k)`: Elimina la clave `k` del nodo `x`.
     - `delete_internal_key(x, i)`: Elimina la clave interna del nodo `x` en la posición `i`.
     - `delete_key_from_child(x, i, k)`: Elimina la clave `k` del hijo `i` del nodo `x`.
     - `get_predecessor(x, i)`: Obtiene el predecesor de la clave en la posición `i` del nodo `x`.
     - `get_successor(x, i)`: Obtiene el sucesor de la clave en la posición `i` del nodo `x`.
     - `borrow_from_previous(x, i)`: Realiza una operación de préstamo desde el nodo hermano izquierdo del hijo `i` del nodo `x`.
     - `borrow_from_next(x, i)`: Realiza una operación de préstamo desde el nodo hermano derecho del hijo `i` del nodo `x`.
     - `merge_children(x, i)`: Fusiona el hijo `i` del nodo `x` con su hermano derecho.
     - `search(k)`: Busca una clave `k` en el árbol.
     - `traverse()`: Realiza un recorrido en orden del árbol.

3. **Función `run_tests()`**:
   - Ejecuta varios casos de prueba para probar la funcionalidad del árbol B.
   - Inserta claves, elimina claves, realiza búsquedas y recorre el árbol.

Paso a paso

1. **Inicialización de la clase `BTreeNode`**:
   - `__init__(self, leaf=False)`: Este método inicializa un nodo del árbol B.
     - `leaf`: Un booleano que indica si el nodo es una hoja o no.
     - `keys`: Una lista que almacena las claves en el nodo.
     - `child`: Una lista que contiene referencias a los hijos del nodo.

2. **Inicialización de la clase `BTree`**:
   - `__init__(self, t)`: Este método inicializa el árbol B.
     - `root`: El nodo raíz del árbol.
     - `t`: Un parámetro que define el grado mínimo del árbol.

3. **Método `insert(self, k)`**:
   - Este método inserta una clave `k` en el árbol.
   - Si la raíz está llena, se crea una nueva raíz y se realiza la división.
   - Llama al método `insert_non_full(self, x, k)` para insertar la clave en un nodo no lleno.

4. **Método `insert_non_full(self, x, k)`**:
   - Este método inserta una clave `k` en un nodo `x` que no está lleno.
   - Si `x` es una hoja, se inserta la clave en la posición correcta en `x.keys`.
   - Si `x` no es una hoja, se encuentra el hijo adecuado y se llama recursivamente a `insert_non_full()` en ese hijo.

5. **Método `split_child(self, x, i)`**:
   - Este método divide el hijo `i` del nodo `x` si está lleno.
   - Crea un nuevo nodo `z` para almacenar las claves divididas.
   - Mueve las claves y los hijos apropiadamente entre los nodos `x` e `y`.

6. **Método `delete(self, k)`**:
   - Este método elimina una clave `k` del árbol.
   - Llama al método `delete_key(self, x, k)` para realizar la eliminación de la clave.

7. **Método `delete_key(self, x, k)`**:
   - Este método elimina la clave `k` del nodo `x`.
   - Si `x` es una hoja, simplemente se elimina la clave.
   - Si `x` no es una hoja, se manejan varios casos dependiendo de si la clave a eliminar está en `x` o en sus hijos.

8. **Método `delete_internal_key(self, x, i)`**:
   - Este método maneja la eliminación de una clave interna del nodo `x` en la posición `i`.
   - Se manejan varios casos para garantizar que el árbol siga siendo válido después de la eliminación.

9. **Método `delete_key_from_child(self, x, i, k)`**:
   - Este método maneja la eliminación de la clave `k` del hijo `i` del nodo `x`.
   - Se manejan varios casos, incluida la fusión de nodos y el préstamo de claves de nodos vecinos.

10. **Método `search(self, k)`**:
    - Este método busca una clave `k` en el árbol.
    - Comienza en la raíz y desciende por el árbol según sea necesario para encontrar la clave.

11. **Método `traverse(self)`**:
    - Este método realiza un recorrido en orden del árbol.
    - Comienza en la raíz y visita cada nodo y clave en orden ascendente.

12. **Función `run_tests()`**:
    - Esta función ejecuta varios casos de prueba para probar la funcionalidad del árbol B.
    - Inserta claves, elimina claves, realiza búsquedas y recorre el árbol, mostrando los resultados en cada paso.