

Informe de Laboratorio: Abstracción e Implementación de Árboles B

Materia: Ciencias de la Computación II

1. El Problema de la Abstracción: ¿Por qué Árboles B?

En Ciencias de la Computación, una **abstracción** es un modelo simplificado de una realidad compleja. Hasta ahora, hemos trabajado con Árboles Binarios (BST, AVL) bajo la suposición de que "acceder a un nodo es barato". Esta suposición es cierta en la memoria RAM, pero falsa en el disco duro.

El **Árbol B** nace de una necesidad física: los discos duros leen datos en bloques (páginas) grandes, no byte por byte.

- **La Abstracción del Árbol Binario:** Un nodo = Un dato. (Ineficiente en disco: requiere muchas lecturas para encontrar algo).
- **La Abstracción del Árbol B:** Un nodo = Una página de disco llena de datos.

El Árbol B generaliza el árbol binario permitiendo que cada nodo tenga **múltiples claves** y **múltiples hijos**. Esto reduce drásticamente la altura del árbol, minimizando las costosas operaciones de entrada/salida (I/O).

2. Definición Formal de la Estructura

Para implementar esta abstracción en Java, primero debemos respetar sus reglas matemáticas (invariantes). Un Árbol B de grado mínimo $t \geq 2$ satisface:

1. **Nodos:** Cada nodo contiene n claves ordenadas tal que $k_1 \leq k_2 \leq \dots \leq k_n$
2. **Rango de Claves:**
 - Todo nodo (excepto la raíz) debe tener al menos $t-1$ claves.
 - Todo nodo puede tener como máximo $2t-1$ claves.
3. **Hijos:** Si un nodo interno tiene n claves, entonces tiene obligatoriamente $n+1$ hijos.
4. **Profundidad:** Todas las hojas aparecen al mismo nivel (el árbol está perfectamente balanceado).

3. Implementación: De la Teoría a la Clase Java

Nuestra implementación orientada a objetos ("POO") busca mapear estos conceptos teóricos directamente a clases de software.

3.1 La Clase `BTreeNode`: La Unidad Fundamental

En lugar de crear una clase para "Nodo Hoja" y otra para "Nodo Interno", abstraemos ambos conceptos en una sola clase `BTreeNode` utilizando una bandera booleana.

```
class BTreeNode {  
  
    int[] keys;           // El bloque de datos (página del disco)  
  
    BTreeNode[] children; // Punteros a otras páginas  
  
    boolean leaf;        // ¿Es el final del camino?  
  
    int n;               // Ocupación actual de la página  
  
    int t;               // El factor de grado (definido por el sistema)  
  
}
```

Justificación del Diseño:

- **Arrays estáticos (`keys` `children`):** Usamos arreglos de tamaño fijo ($2t-1$) porque representan bloques de memoria física de tamaño fijo. No usamos `ArrayList` dinámicos porque violarían la simulación de una página de disco real.
- **Encapsulamiento:** El nodo gestiona su propia información local. Por ejemplo, el método `search(k)` dentro del nodo sabe buscar en su propio arreglo antes de delegar a un hijo.

3.2 La Clase `BTREE`: El Gestor de Invariantes

Mientras que el nodo es un contenedor pasivo, la clase `BTREE` representa la inteligencia del sistema. Es la responsable de mantener los invariantes definidos en la sección 2.

- **Abstracción de Acceso:** El usuario final solo ve `insert(k)` y

- `search(k)`. No sabe (ni debe saber) que internamente los nodos se están dividiendo o fusionando.
- **Manejo de la Raíz:** La raíz es el único nodo que puede violar la regla del mínimo de claves (puede tener solo 1). La clase `BTree` maneja esta excepción.

4. Dinámica de la Estructura: División y Crecimiento

La operación más crítica en la abstracción del Árbol B es el **Split** (División).

En un árbol binario, crecemos hacia abajo añadiendo hojas. En un Árbol B, crecemos **hacia arriba**. Cuando un nodo se llena (alcanza **$2t-1$** claves):

1. Se divide en dos nodos de **$t-1$** claves cada uno.
2. La clave mediana sube al padre.

Estrategia "Pre-emptive Split" (División Proactiva): Nuestra implementación en Java utiliza una técnica avanzada: dividimos los nodos llenos **mientras descendemos** buscando dónde insertar. Esto garantiza que, al intentar insertar en una hoja, el padre de esa hoja nunca estará lleno, permitiendo subir la mediana sin problemas. Simplifica el código al eliminar la necesidad de volver atrás (backtracking).

Política de Duplicados: Nuestra lógica (`keys[i] > k`) dicta que si insertamos un valor que ya existe, este se colocará a la **derecha** de su igual. Esto mantiene la consistencia semántica del ordenamiento.

5. Verificación de la Abstracción: Visualización Web

Entender una estructura tan compleja solo con código es difícil. Para verificar que nuestra abstracción Java (`BTreeNode` `BTree`) se comporta realmente como la teoría dicta, implementamos un módulo de visualización.

El Método `exportarHTML`

Este método actúa como un puente entre la memoria del programa y la representación visual humana.

1. **Serialización:** Convierte el grafo de objetos Java en una estructura JSON jerárquica.

2. **Representación:** Genera un archivo HTML donde cada objeto `BTreeNode` se dibuja como una caja (Página) y las referencias `children` se dibujan como curvas (Punteros).

Esto nos permite confirmar visualmente que:

- Las hojas están todas al mismo nivel.
- Los nodos no exceden el límite de claves.
- La estructura crece correctamente al dividirse la raíz.

6. Conclusión

La implementación del Árbol B en Java es un excelente ejercicio de abstracción. Hemos tomado restricciones físicas (lectura de bloques en disco) y las hemos convertido en reglas lógicas de software.

A través de las clases `BTreeNode` y `BTtree`, encapsulamos la complejidad del balanceo automático, ofreciendo una estructura eficiente $O(\log_t n)$ ideal para sistemas de almacenamiento masivo.