Resumen Teoria

Árboles en Bases de Datos

Los árboles son estructuras jerárquicas utilizadas para organizar datos de manera eficiente, permitiendo operaciones rápidas de inserción, búsqueda y eliminación. A continuación, se detallan los principales tipos:

Árboles Binarios

- **Definición**: Árboles donde cada nodo tiene máximo dos hijos (izquierdo y derecho).
- **Usos comunes**:
  - Representación de expresiones aritméticas.
  - Árboles de decisión en algoritmos.
- **Ventajas**:
  - Simplicidad en la implementación.
  - Búsqueda eficiente si está balanceado.
- **Desventajas**:
  - Desperdicio de espacio en nodos vacíos.
  - Baja eficiencia si no está balanceado (altura desproporcionada).

Árboles AVL

- **Definición**: Árbol binario de búsqueda auto-balanceado.
- **Características clave**:
  - La altura de los subárboles de cualquier nodo difiere en máximo 1.
  - Usa rotaciones (simple o doble) para mantener el equilibrio.
- **Ventajas**:
  - Búsqueda en tiempo O(logn)$O(\log n)$.
  - Ideal para aplicaciones con inserciones/eliminaciones frecuentes.
- **Desventajas**:
  - Complejidad en la implementación de rotaciones.

Característica	Árbol Binario	Árbol AVL
Balanceo	No garantizado	Auto-balanceado
Complejidad de búsqueda	$O(n)$ en peor caso	$O(\log n)$ siempre
Uso de memoria	Menor	Mayor (almacena alturas)
Caso de uso	Datos estáticos	Datos dinámicos

Árboles Multicamino (B y B+)

- **Definición**: Árboles donde cada nodo puede tener múltiples hijos (N>2*N*>2).
- **Propósito**: Reducir la altura del árbol para minimizar accesos a disco.

Árbol B

- **Características**:
  - Todos los nodos (hojas y no hojas) almacenan claves.
  - Cada nodo tiene entre ⌈m/2⌉⌈*m*/2⌉ y m*m* hijos (donde m*m* es el orden del árbol).
- **Ventajas**:
  - Eficiente para sistemas de almacenamiento en disco.
  - Inserción y eliminación sin necesidad de rebalanceo frecuente.

Árbol B+

- **Características**:
  - Las claves se duplican en nodos hoja.
  - Nodos internos actúan como índices.
- **Ventajas**:
  - Recorrido secuencial eficiente gracias a punteros en nodos hoja.
  - Mayor densidad de almacenamiento.

Árboles AVL

Política de Inserción Mejorada

- **Redistribución antes de la división**:

  Cuando un nodo alcanza su capacidad máxima, en lugar de dividirse inmediatamente, intenta redistribuir sus claves con nodos hermanos adyacentes.

  - **Ejemplo**: Si un nodo está lleno y su hermano derecho tiene espacio, algunas claves se mueven al hermano para evitar la división.

- Solo si **todos los hermanos están llenos** se procede a dividir el nodo.
- **Mayor Densidad de Almacenamiento**

- **Ocupación mínima garantizada**:
  - En árboles B estándar: Cada nodo debe tener al menos ⌈m/2⌉⌈m/2⌉ claves.
  - En árboles B*: Cada nodo debe tener al menos ⌈2m/3⌉⌈2m/3⌉ claves.
  - **Resultado**: Menos nodos y mayor aprovechamiento del espacio (hasta un 66% de ocupación mínima).

Menos Divisiones

- Al redistribuir claves entre nodos hermanos, se reduce la necesidad de crear nuevos nodos, lo que minimiza operaciones de escritura en disco.

Característica	Árbol B	Árbol B+	Árbol B*
Claves en nodos hoja	Sí (todas las claves)	Sí (duplicadas en hojas)	Sí (similar a B+)
Claves en nodos internos	Sí	Solo punteros a hojas	Sí (pero con redistribución)
Ocupación mínima	$\lceil m/2 \rceil$	$\lceil m/2 \rceil$	$\lceil 2m/3 \rceil$
Manejo de overflow	División inmediata	División inmediata	Redistribución primero
Acceso secuencial	Ineficiente	Eficiente (punteros en hojas)	Similar a B+
Uso en bases de datos	Índices secundarios	Índices primarios/clustering	Sistemas con alta inserción

Requisito	Árbol B	Árbol B+	Árbol B*
Acceso secuencial	No recomendado	✅ Ideal	✅ Similar a B+
Altas inserciones	❌ Poco eficiente	❌ Moderado	✅ Óptimo
Espacio en disco limitado	❌ (50% ocupación)	❌ (50% ocupación)	✅ (66% ocupación)
Implementación sencilla	✅ Más simple	❌ Moderada	❌ Compleja
Claves duplicadas	✅ Permitidas	❌ No recomendado	❌ No recomendado

Dispersión (Hashing)

Técnica que convierte una clave en una dirección única para almacenar/recuperar datos rápidamente.

Métodos de Dispersión

1. **División**:
   - **Fórmula**: h(k)=kmod mh(k)=kmodm, donde mm es el tamaño de la tabla.
   - **Ejemplo**: Para k=1234k=1234 y m=100m=100, h(k)=34h(k)=34.
   - **Ventaja**: Simple y rápido.
2. **Plegado**:
   - **Pasos**: Dividir la clave en partes, sumarlas.
   - **Ejemplo**: k=123456k=123456 → 12+34+56=10212+34+56=102.
3. **Cuadrados Centrales**:
   - **Pasos**: Elevar al cuadrado la clave y tomar dígitos centrales.
   - **Ejemplo**: k=1234k=1234 → 12342=152275612342=1522756 → Dígitos centrales: 2275.

Manejo de Colisiones

- **Cubetas (Buckets)**: Espacios que almacenan múltiples registros en una misma dirección.
- **Parámetros clave**:
  - **Densidad de empaquetamiento (DEDE)**: DE=Nº de registrosNº de direccionesDE=Nº de direccionesNº de registros.
  - **Tamaño de cubeta**: Afecta la fragmentación y velocidad de acceso.

Método	Ventajas	Desventajas
División	Simple, rápido	Sensible a patrones de claves
Plegado	Bueno para claves largas	Puede generar colisiones frecuentes
Cuadrados Centrales	Distribución uniforme	Costoso computacionalmente

Dispersión (Hashing)

La **dispersión extensible** es un método de hashing dinámico diseñado para manejar el crecimiento de una tabla de hash de manera eficiente, evitando colisiones excesivas y manteniendo un rendimiento óptimo. A diferencia del hashing estático, donde el tamaño de la tabla es fijo, este método ajusta automáticamente el tamaño de la tabla utilizando un directorio de punteros y buckets (cubetas) que pueden dividirse según sea necesario.

Estructura Básica

- **Directorios**:
  - Tabla de punteros que indexa buckets.
  - Su tamaño crece exponencialmente (en potencias de 2) según la necesidad.
- **Buckets**:
  - Contenedores que almacenan registros.
  - Cada bucket tiene una **profundidad local** (dl*dl*), que indica el número de bits utilizados para direccionarlo.
- **Profundidad global** (dg*dg*):
  - Número de bits utilizados para indexar el directorio.
  - Relacionado con el tamaño del directorio: Taman˜o del directorio=2dgTaman˜o del directorio=2*dg*.

Funcionamiento

1. **Inserción**:
   - Se calcula el hash de la clave y se toman los primeros dg*dg* bits para ubicar el bucket correspondiente en el directorio.
   - Si el bucket se llena, se divide en dos nuevos buckets, incrementando su profundidad local (dl+1*dl*+1).
   - Si dl=dg*dl*=*dg*, el directorio duplica su tamaño (dg+1*dg*+1) para acomodar más buckets.
2. **Búsqueda**:
   - Usa los primeros dg*dg* bits del hash para acceder directamente al bucket.