# アルゴリズムとプログラミング

## 🔷 Bツリーインデックス (B-tree Index)

- **仕組み**
  ・木構造（バランス木）を使ってデータを管理。
  ・インデックスは「キーの値」を元に並べられ、検索・挿入・削除の処理が効率的。
  ・RDB（Oracle, MySQL, PostgreSQL など）で最も一般的。
- **特徴**
  - 範囲検索（例：age BETWEEN 20 AND 30）が得意
  - 順序を保つ → ORDER BY や > < も効率的
  - 値の種類が多い（高カーディナリティな列、例：ユーザーIDや日付）に向いている
  - 更新頻度が高くてもそこそこ効率的
- **イメージ**
  辞書の索引のように、探す単語がどのページにあるかを木構造で辿っていく。

## 🔷 ビットマップインデックス (Bitmap Index)

- **仕組み**
  ・ある列の値ごとに「ビット列」を用意する。
  ・例えば gender 列が {Male, Female} の場合、Male用ビット列・Female用ビット列を作り、行ごとに該当なら1、違えば0をセット。
- **特徴**
  - 値の種類が少ない（低カーディナリティな列、例：性別、都道府県コード、フラグ値）に強い
  - 複数条件の **AND / OR** がビット演算で一瞬（高速！）
  - 更新に弱い（挿入・削除・更新が頻繁だとビット列を再構築する必要があるため遅い）
  - 典型的には **データウェアハウスや分析系クエリ（更新少なめ、参照多め）**に使われる
- **イメージ**
  テーブルの各行を「フラグの表」で管理する感じ。条件を掛け合わせるときにビット演算で一気に絞り込める。

## 🔷 使い分けまとめ

| 項目 | Bツリーインデックス | ビットマップインデックス |
|---|---|---|
| データ種類 | 値が多い（ユーザーID, 日付, 金額など） | 値が少ない（性別, フラグ, カテゴリ） |
| 得意な検索 | 範囲検索、順序付き検索 | 複数条件の組み合わせ（AND/OR） |
| 更新 | 強い（OLTP向き） | 弱い（DWH/分析向き） |
| 主な用途 | トランザクション処理 | 分析・集計処理 |

## 🔷 ハッシュインデックス (Hash Index)

- **仕組み**
  ・インデックス列の値に **ハッシュ関数** をかけて、ハッシュ値をキーにして位置を特定。
  ・ハッシュ表（ハッシュテーブル）の仕組みをそのままDBに取り入れたもの。
- **特徴**
  ✅ **等価検索に強い**
  - WHERE id = 12345 のように、ピッタリ一致する検索が最速クラス
  - O(1) に近い時間でアクセスできる
- ❌ **範囲検索に弱い**
  - id BETWEEN 1000 AND 2000 はハッシュ値の順序性がないので不可能 or 全走査になる
  - > < ORDER BY にも使えない
- ✅ 更新は比較的速い
  - 値をハッシュ化して登録するだけなので、Bツリーよりも更新コストが軽いこともある
- **用途**
  - 主に **等号検索だけを多用する場合** に有効
  - 例：ユーザーIDや商品コードなど、**キーに対して正確に1件を探すだけ** のケース
  - PostgreSQL では USING HASH で明示的に作れる（ただし制限も多い）
  - MySQL の MEMORY ストレージエンジンなどでデフォルト採用されることもある

## 🔷 再入可能 (Reentrant)

- **意味**
  あるプログラム（特に関数やサブルーチン）が、**複数の実行スレッドから同時に呼ばれても正しく動作する**こと。
- **ポイント**
  - グローバル変数や静的変数に依存しない
  - 共有データを壊さない（排他制御なしでも安全）
  - 同時呼び出しされても「誰の実行結果か」が混ざらない

## 🔷 再帰 (Recursive)

- **意味**
  関数や処理が **自分自身を呼び出すこと**。
- **ポイント**
  - 問題を小さい部分問題に分けて処理できるときに便利
  - スタックに呼び出し履歴を積むので、深すぎるとスタックオーバーフローになる

## 🔷 再配置可能 (Relocatable)

- **意味**
  プログラムやモジュールが、**メモリ上のどのアドレスにロードされても動く**こと。
- **背景**
  昔のプログラムは「0x1000番地から開始する」と決め打ちしていた。
  しかしマルチプログラム環境では、空いているメモリ領域に自由にロードできた方が便利。