データベースS

第12回　物理的データ格納方式

システム創成情報工学科　尾下 真樹

2018年度 Q2

今回の内容

- 今回以降の授業では、データベースシステム内部で用いられるさまざまな技術を学習する
- 物理的データ格納方式
  - リレーションがどのようにハードディスクに記録されるか？
  - どのような問題に気を付ける必要があるか？
  - アクセスを高速化するためのデータ構造

今回の内容

- 前回の復習
- 物理的データ格納方式
  - データ格納の概要
  - データ格納方法
  - データアクセス方法
  - データ格納方法の工夫

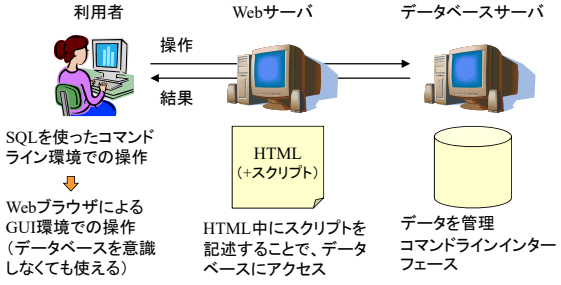
教科書

- 「リレーショナルデータベース入門［第3版］」増永良文 著、サイエンス社
  - 8章
- 「データベースシステム」北川 博之 著、昭晃堂 出版
  - 6章 102～124ページ

前回の復習

Webインターフェース

- Webページを経由してデータベースを操作



インターフェースの作成

- 作成する機能
  - 従業員データの一覧表示
  - 従業員データの追加
  - 従業員データの追加(動的生成)
  - 従業員データの削除
  - 従業員データの削除(動的生成)
  - 従業員データの更新
  - 従業員データの検索

サンプルページの構成

- メニュー(menu.html)
  - → 一覧表示(employee_list.php)
  - → 追加フォーム(exmployee_add.html)
    - → 追加処理(employee_add.php)
  - → 追加フォーム(動的生成版)(exmployee_add_form.php)
    - → 追加処理(employee_add.php)
  - → 削除フォーム(employee_delete.html)
    - → 削除処理(employee_delete.php)
  - → 削除フォーム(動的生成版)(employee_delete_form.php)
    - → 削除処理(employee_delete.php)
  - → 更新フォーム(employee_update_form1.html)
    - → 更新フォーム(employee_update_form2.php)
      - → 更新処理(employee_update.php)
  - → 検索フォーム(動的生成)(employee_search_form.php)
    - → 削除処理(employee_search.php)

追加フォームの動的生成

- 全てのデータを入力するのは大変、また、一部のデータは入力可能なデータが限られる
  - 例えば、部門番号には、外部参照整合性制約があるので、存在しない部門番号は入力不可能

- 適切な初期値や選択肢を表示することで、入力を簡便化したり、不適切なデータが入力されることを防止したりできる

演習課題(1)

前回の演習課題は終わっているものとする

1. 削除処理を行なうPHPプログラムに削除処理のためのSQLを追加し、削除が正しく動作するようにせよ(exmployee_delete.php)
2. 更新処理を行なうPHPプログラムに更新処理のためのSQLを追加し、更新が正しく動作するようにせよ(exmployee_update.php)

演習課題(2)

3. 更新機能で、更新する従業員をリストから選択できるように拡張したものを作成し、正しく動作するようにせよ(exmployee_update_form1.php)
4. 検索機能として、選択された部門の従業員の一覧を表示するSQLを追加し、検索が正しく動作するようにせよ(exmployee_search_form.php, exmployee_search.php)

レポート課題

- データベースの作成
  - 自分で決めた何らかのテーマを題材にして、データベースとWebインターフェースを作成
- Moodleから提出
  - レポート、作成したプログラム一式を提出
  - ウェブインターフェースも作成
- レポートの締め切りは後日連絡
  - 8月下旬(期末試験後)の締め切りを予定

レポート課題

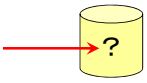
- 課題内容
  - 自分で決めた何らかのテーマを題材にして、データベースとWebインターフェースを作成
  1. スキーマの設計
     - データベースに格納するデータを決めて、思いつく属性を挙げる → 正規形を満たすように正規化
  2. テーブルの作成、データの追加
  3. Webインターフェースの作成
     - 一覧表示・追加・削除・修正
     - なるべく実用的に使えるような検索機能などを追加

データ格納の概要

物理的データ格納方式

- リレーショナルデータモデルの特徴
  - 抽象的な「リレーショナルモデル」として定義
  - リレーションがメモリやハードディスクにどのように格納されるかは規定していない
    → 論理的モデルと物理的構造の分離
  - 各データベースシステムごとに、物理的なデータ構造を工夫することができる

従業員番号	部門番号	氏名	年齢
0001	01	織田 信長	48
0002	02	豊臣 秀吉	45
0003	03	徳川 家康	39

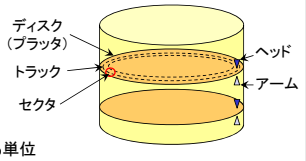


物理的データ格納方式の課題

- データはハードディスク上に格納される
  - データベースではデータ量が非常に大きくなるため、全てをメモリ上にのせることができない
  - メモリ上にあるデータは簡単に消えてしまうため、常にハードディスクにデータを記録することが望ましい
- 効率的なデータ構造・アルゴリズムが必要
  - データ量が増えても極端に遅くなったりしない方式が要求される
  - ハードディスクのアクセス速度はメモリに比べて非常に遅い（アクセス速度＝シーク速度＋読み込み速度）
    - なるべくアクセス回数を減らすような工夫が必要になる

ハードディスクへの格納

- ハードディスクの仕組み
  - 物理的構造
    - ディスク（プラッタ）
    - トラック
    - セクタ（512バイト）
    - ページ（ブロック）
      - 一度に読み書きする単位
      - セクタを数KB程度ずつにまとめたもの



  - 論理的構造
    - ファイル
      - 複数のページによって構成される
      - 物理的なページは離れていてもアプリケーションからは連続しているかのように扱える（OSの機能）

ファイルとリレーションの対応

- リレーショナルモデルと物理モデルの対応
  - リレーション → ファイル
  - タプル → レコード
  - 属性 → フィールド

従業員番号	部門番号	氏名	年齢
0001	01	織田 信長	48
0002	02	豊臣 秀吉	45
0003	03	徳川 家康	39
0004	01	柴田 勝家	60

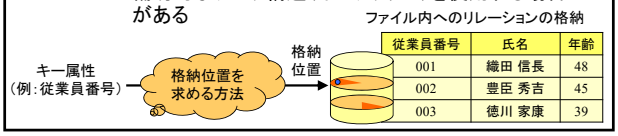
- 物理モデル
  - レコード
    - 記録の単位
    - 一般にレコードのサイズは小さいので、1ページに複数のレコードが記録される
  - フィールド
    - レコード内に含まれる各値

データの格納と検索

- データ(レコード)の格納と検索
  - データは何らかのルールに従ってファイル内に格納される
  - 検索条件が与えられた時、そのレコードがファイル内のどこに格納されているかを高速に求めることができるようになっていることが必要
- 検索の種類
  - 等号検索
    - 例:年齢が20歳の職員を検索
  - 範囲検索(レンジ・クエリー)
    - 例:年齢が20歳～25歳の職員を検索

データの格納と検索

- 下記の2つは区別する必要がある
  - データ格納方法
    - ファイル内にレコードをどのような方法で配置するか
  - アクセス方法(データの格納位置の決定方法)
    - 検索条件が与えられた時にそのデータがファイル内のどこに配置されているかを求めるための方法
    - 補助的なデータ構造(インデックス)を使用する場合がある



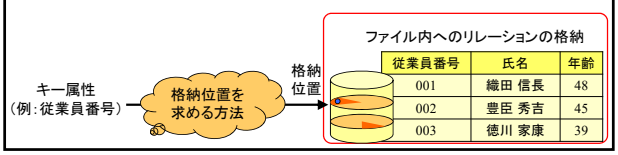
データの格納と検索

- ファイル内のデータの格納方法の種類
  - 順序なしの格納
  - 順序付きの格納
  - ハッシュを使った格納
- アクセス方法(データ格納位置の決定方法)
  - スキャン法
  - 探索法
  - インデックス法
  - ハッシュ法

データ格納方法

データの格納方法

- ファイル内のデータの格納方法の種類
  - 順序なしの格納
  - 順序付きの格納
  - ハッシュを使った格納

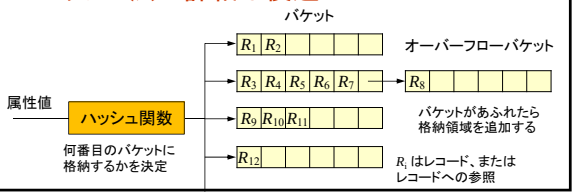


順序なし／順序付きの格納

- 順序なしの格納(ヒープ編成)
  - 挿入された順番でファイル上にレコードを格納
    - 挿入処理は容易
    - 検索に手間がかかる
- 順序付きの格納(順次編成)
  - キー属性値を基準として昇順or降順で順番に(ソートした状態で)レコードを格納
    - 挿入処理に手間がかかる
    - 検索を効率的に行える(後述)

ハッシュを使った格納

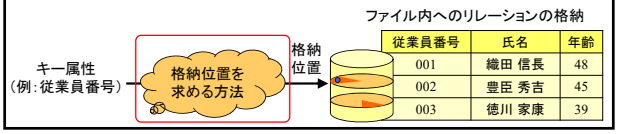
- 何らかのハッシュ関数を用いて、キー属性値からレコードを格納するバケットを決定
- 各バケットでは、順序なしでレコードを配置
- ハッシュ法の詳細は後述



データアクセス方法

アクセス・メソッド

- データの格納位置の決定方法の種類
  - スキャン法
  - 探索法
  - インデックス法
  - ハッシュ法



計算量の議論のための予備知識

- オーダー表記 O(～)
  - アルゴリズムの効率を評価するための指標
  - 処理すべきデータの量を n とした時、n と処理量の関係を表したもの
    - $O(n)$・・・処理量はデータ量に比例する
    - $O(1)$・・・処理量は常に一定
  - オーダー表記では係数は無視される
  - 一般に、n が非常に大きくなる可能性のある問題を扱う時、オーダーは非常に重要になる
    - $O(n)$より小さいオーダーが求められる($O(\log n)$や$O(1)$)

データの格納と検索(確認)

- ファイル内のデータの格納方法の種類
  - 順序なしの格納
  - 順序付きの格納
  - ハッシュを使った格納
- アクセス方法(データ格納位置の決定方法)
  - スキャン法
  - 探索法
  - インデックス法
  - ハッシュ法

格納方法と検索方法の対応

- 順序なしの格納 → スキャン法
- 順序付きの格納 → 探索法
- 順序なしの格納＋インデックス → インデックス法
- 順序付きの格納＋インデックス → インデックス法
- ハッシュを使った格納 → ハッシュ法

データアクセス方法

- スキャン法
- 探索法
- インデックス法
- ハッシュ法

スキャン法

- 走査法、順次アクセス法、などとも呼ばれる
- データが順序なしで格納されている場合に用いる方法
- 格納されている順番で全てのデータを探索
  - アクセス時は全てのデータを順番に調べる必要があるため、$O(n)$ となり、処理時間がかかる
  - 新しいデータを挿入する時は、データ領域の末尾に追加するだけなので $O(1)$

スキャン法の例

従業員番号 001 の社員を検索

従業員番号	氏名	年齢
005	伊達 政宗	15
003	徳川 家康	39
002	豊臣 秀吉	45
004	柴田 勝家	60
007	島津 家久	35
006	上杉 景勝	26
001	織田 信長	48

データアクセス方法

- スキャン法
- 探索法
- インデックス法
- ハッシュ法

探索法

- アクセスする属性値の順序に従ってレコードが格納されている場合に使用可能な方法
- 順番に並んでいることを利用して高速に探索
  - 2分探索
    - 全体の中央の属性値を調べて、求める属性値が存在する範囲を絞り込んでゆく
    - アクセス時の手間は小さくなる $O(\log n)$
      - 最悪でも $\log_2 n$ 回の判定で探索対象のデータに到達する
    - 新しいデータを挿入する時にも、挿入場所を決定するために、同じ手間がかかる $O(\log n)$
  - ブロック探索（n分探索）（原理は同じ、詳しい説明は省略）

探索法の例

従業員番号 001 の社員を検索

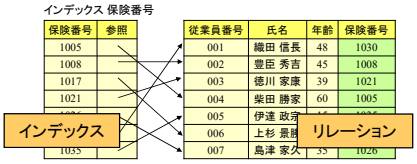
従業員番号	氏名	年齢
001	織田 信長	48
002	豊臣 秀吉	45
003	徳川 家康	39
004	柴田 勝家	60
005	伊達 政宗	15
006	上杉 景勝	26
007	島津 家久	35

データアクセス方法

- スキャン法
- 探索法
- インデックス法
- ハッシュ法

インデックス法

- インデックス（索引）
  - データ（レコード）が格納されている場所を求めるための別のデータ構造を作成・利用する
  - インデックスを用いることで、順序づけされていない属性についても探索法が適用可能になる



インデックスの種類

- インデックスの種類
  - 1次インデックス
    - 順序キー属性についてのインデックス
      - インデックスの中に、リレーションのデータを同時に格納
  - 2次インデックス
    - 順序キー以外についてのインデックス
      - インデックスの中に、データへの参照を格納
    - ある属性に対するインデックスを作成することで、その属性にもとづいた検索を高速化できる
    - 非候補キー属性の場合、ひとつの属性値に複数のレコードが対応することがある

1次インデックスの例

- インデックスとデータを同時に格納

テーブル ＋ 従業員番号インデックス

従業員番号	部門番号	氏名	年齢	保険番号
0001	01	織田 信長	48	1030
0002	02	豊臣 秀吉	45	1008
0003	03	徳川 家康	39	1021
0004	01	柴田 勝家	60	1005
0005	01	伊達 政宗	15	1035
0006	02	上杉 景勝	26	1017
0007	03	島津 家久	35	1026

2次インデックスの例

- 一つの属性値に一つのデータが対応する例

保険番号 インデックス

保険番号	参照
1005	
1008	
1017	
1021	
1026	
1030	
1035	

従業員番号	氏名	年齢	保険番号
001	織田 信長	48	1030
002	豊臣 秀吉	45	1008
003	徳川 家康	39	1021
004	柴田 勝家	60	1005
005	伊達 政宗	15	1035
006	上杉 景勝	26	1017
007	島津 家久	35	1026

2次インデックスの例

- 一つの属性値に複数のデータが対応する例

部門番号 インデックス

部門番号	参照
01	
02	
03	

ブロック

従業員番号	部門番号	氏名	年齢
0001	01	織田 信長	48
0002	02	豊臣 秀吉	45
0003	03	徳川 家康	39
0004	01	柴田 勝家	60
0005	01	伊達 政宗	15
0006	02	上杉 景勝	26
0007	03	島津 家久	35

インデックスから直接レコードを参照するのではなく、複数のレコードへの参照を格納したブロックを参照

多段インデックス（ツリー）

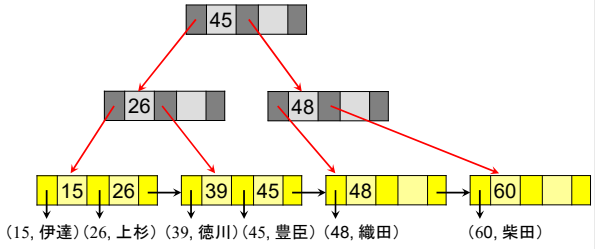
- 単純なインデックスの問題点
  - レコードが挿入・削除されると、インデックス全体を更新する必要がある（処理時間がかかる）
  - インデックス全体の探索が必要になる
- ツリー構造を用いることでこの問題を解決
- 多段インデックス（ツリー）
  - ISAMツリー
  - B-ツリー
  - B+-ツリー

B+-ツリー

- リレーショナルデータベースシステムで広く使われているインデックス
- 中間ノードと葉ノードから構成される
  - 中間ノード、葉ノードも一種のレコード
- ツリー全体で高さが同じ（平衡木）
  - レコードが一定以上挿入されたら、ツリーの高さが一段増える
  - レコードが一定以上削除されたら、一段減る
  - ツリーの高さが同じなので、どのデータも同じ速度で探索できる

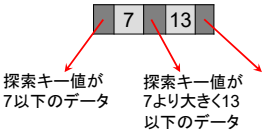
B+-ツリーの構成

- B+-ツリーの例

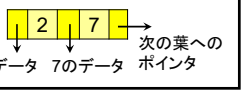


B+-ツリーのノード

- 中間ノード
  - $p/2$ ～ p 個の下位ノード（中間ノードor葉ノード）への参照を格納
    - 左のキー値より大きく、右のキー値以下のデータを含む下位ノードへの参照を持つ



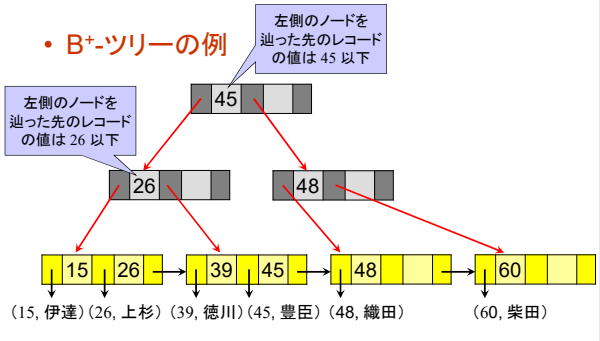
- 葉ノード
  - $(q\text{-}1)/2$ ～ q 個のレコード群への参照を格納
    - 右のキー値のデータ（データブロック）を持つ



※ $p/2$, $(q\text{-}1)/2$ はともに切り上げ

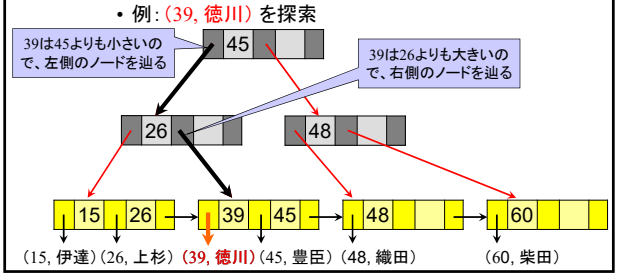
B+-ツリーの構成

- B+-ツリーの例



レコード探索

- B+-ツリーでのレコード探索
  - ルートノードから順番に下にノードをたどる
    - 例：(39, 徳川) を探索



レコード挿入（ツリーの構築）

- B+-ツリーでのレコード挿入の手順
  - 探索アルゴリズムを用いて、レコードを格納する葉ノードを決定
  - 葉ノードに余裕があれば、そこにレコードを追加して終了
  - 葉ノードに余裕がなければ、葉ノードを2つに分割して、ひとつ上の中間ノードに追加
    - 隣の葉ノードに余裕があっても、レコードを移動したりはせず、葉ノードを分割することで、レコードを格納する
  - 中間ノードもあふれてしまった場合は、その中間ノードも分割して、さらの上の枝ノードに追加を繰り返す
  - ルートノードが分割されたら、新たなルートノードを追加
    - ツリー全体の高さが1段高くなる

レコード挿入の例（0）

- B+-ツリーの例
  - 中間ノードに格納できる参照の数を 3個（p=3）
  - 葉ノードに格納できるデータの数を 2個（q =2）として、レコード挿入の例を考える
    - 葉ノードが溢れた場合は、2つに分割し、左側のノードに2つ、右側のノードに1つのデータを格納する

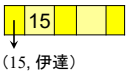
※ 実際には、ノードに格納可能なデータ数はもっと大きいが、分かりやすいように、少ない例で考える

レコード挿入の例（1）

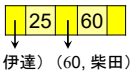
- 従業員（従業員番号, 部門番号, 氏名, 年齢）のリレーションに対して、「年齢」属性でのインデックス（B+-ツリー）を作成する
- リレーションが空の状態から、順番に複数のデータを挿入していったときに、どのようにツリーが更新されるかを考える
  - 最終的には同じ複数のデータを挿入するとしても、挿入する順番によってツリーの状態は変わることに注意する

レコード挿入の例（3）

- （15, 伊達）を挿入

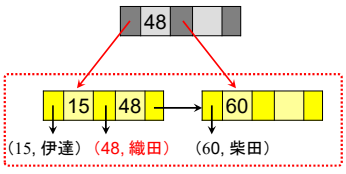


- （60, 柴田）を挿入



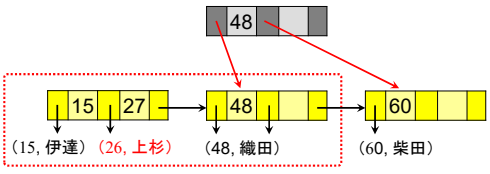
レコード挿入の例（4）

- （48, 織田）を挿入
  - 葉ノードを分割、中間ノードを追加



レコード挿入の例（5）

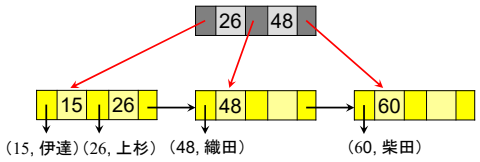
- （26, 上杉）を挿入
  - 葉ノードを分割



注意：ここで、（48, 織田）を右の葉ノードに移せば、葉ノードを分割しなくとも格納できるが、そのような処理は B+ツリーでは行わず、機械的に分割する（アルゴリズムの簡素化のため）

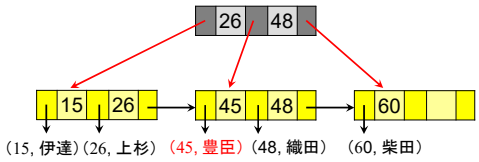
レコード挿入の例（6）

- (26, 上杉)を挿入
  - 分割した葉ノードをひとつ上の中間ノードに追加



レコード挿入の例（7）

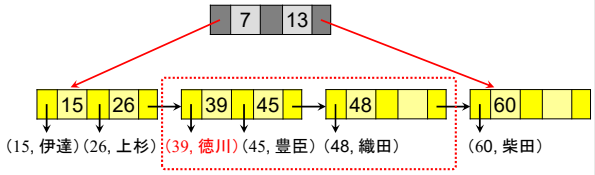
- (45, 豊臣)を挿入
  - 空いている葉ノードに追加



レコード挿入の例（8）

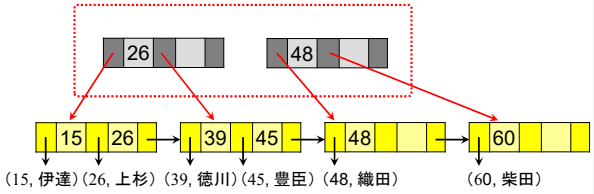
- (39, 徳川)を挿入
  - 葉ノードを分割

注意：前の追加と同じく、(48, 織田) を右の葉ノードに移せば、ノードを分割しなくとも格納できるが、B⁺ツリーではそのような処理は行わず、機械的に分割する



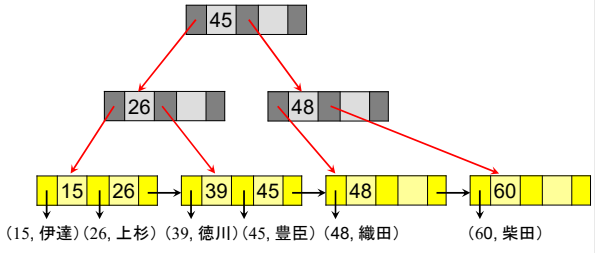
レコード挿入の例（9）

- (39, 徳川)を挿入
  - ひとつ上の中間ノードも分割
  - ツリーの高さを1段増やして、中間ノードを追加



レコード挿入の例（10）

- (39, 徳川)を挿入

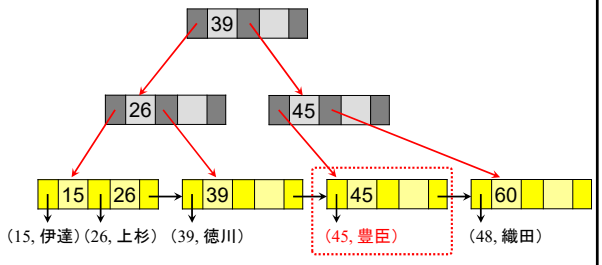


レコード削除

- B^+-ツリーでのレコード削除の手順
  - 挿入の場合と考え方は同じ
  - 削除すべきレコードのある葉ノードを探索
  - 削除の結果、ノード内のデータ数が $p/2$ 以下になったら、以下のどちらかの処理を行なう
    - 隣接するノードからデータを分けてもらう（隣接するノードに十分な数のデータがある場合）
    - 隣接するノードと合体（隣接するノードに十分な数のデータがない場合）
  - 上のノードに向かって順番に繰り返す

レコード削除の例（1）

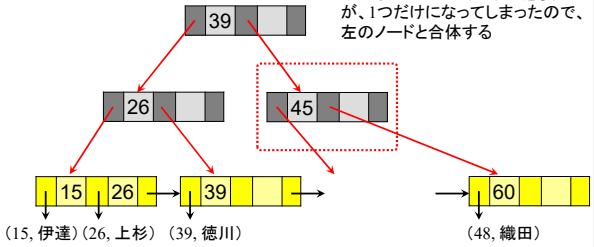
- （45, 豊臣）を削除



レコード削除の例（1）

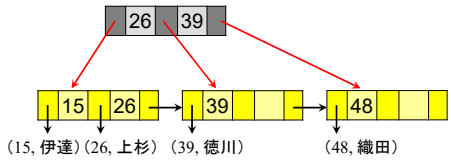
- （45, 豊臣）を削除

この例では、中間ノードは、最低2つの子ノードへのリンクが必要だが、1つだけになってしまったので、左のノードと合体する



レコード削除の例（1）

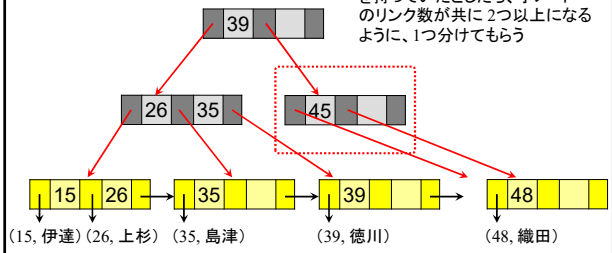
- （45, 豊臣）を削除



レコード削除の例（2）

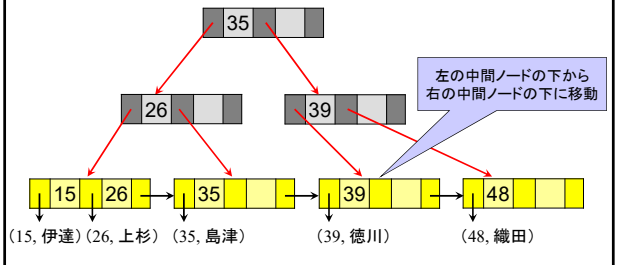
- （45, 豊臣）を削除

もし、左のノードが3つの子ノードを持っていたとしたら、子ノードへのリンク数が共に 2つ以上になるように、1つ分けてもらう



レコード削除の例（2）

- （45, 豊臣）を削除

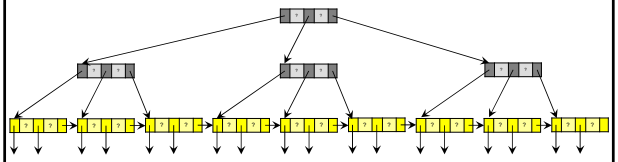


B⁺-ツリーに格納可能なレコード数

- 原理的には、ツリーの深さを深くしていくことで、いくらでも多くのレコードを格納できる
- 例えば、前の例の B⁺-ツリー（p=3、q=2）では
  - 中間ノードの深さが2段のときには、最大、3×3×2=18個までレコードが格納可能
    - この状態でレコードが追加されると、中間ノードの深さが増える
  - 実際には、レコードが均一に葉ノードに格納されるとは限らないので、最小で 3×2×1+3 = 9 個のレコードが格納された時点で、中間ノードの深さが増える
    - 6個の葉ノードにレコードが1つずつ、残り1個の葉ノードに3つ

格納可能な最大レコード数

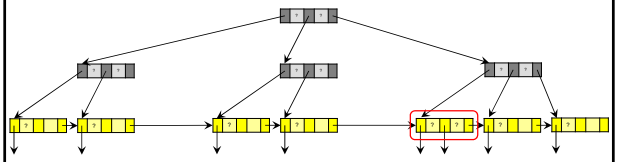
- 格納可能な最大レコード数
  - 中間ノードの深さが2段のときの例
  - 最大 3×3×2=18 個のデータを格納可能



全ての中間・葉ノードが一杯になった状態

格納可能な最小レコード数

- 格納可能な最小レコード数
  - 中間ノードの深さが2段のときの例
  - 最小 3×2×1+3 = 9 個のレコードが格納された時点で、中間ノードの深さが増える



中間・葉ノードに最小限のレコードが格納された状態で、たまたま一杯になっている中間・葉ノードにレコードが追加されると、深さが増える

B^+-ツリー と B-ツリー の違い

- B-ツリー
  - B^+-ツリーのもとになったもの
  - 中間ノードにもレコードへの参照を格納
- 主な違い
  - データ構造が単純
  - 範囲検索時の連続データの取り出しが容易
    - 葉ノードを横にトラバースするだけで良い
  - 中間ノードのデータ量が多くなるので木の高さが低くて済む

インデックス(ツリー)の特徴

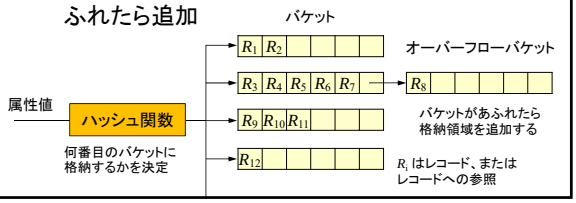
- 任意の格納方法に適用可能
  - 探索法のように、その属性値で順序付きで格納されている必要はない
- 等号検索・範囲検索の両方に対応可能
- データ量が増えても、高速に処理できる
  - 探索・追加のコスト $O(\log n)$
  - オーダーとしては探索法と同じだが、全てのノードをメモリに読み込む必要がないため、より効率的に実行できる

データアクセス方法

- スキャン法
- 探索法
- インデックス法
- ハッシュ法

ハッシュ法

- 何らかのハッシュ関数を用いて、キー属性値からレコードを格納するバケットを決定
- 各バケットでは、順序なしでレコードを配置
  - 各バケットには一定の領域を確保しておき、あふれたら追加



ハッシュ関数

- ハッシュ関数
  - 属性値を引数として、そのレコードを格納するバケットの番号（ハッシュ値）を返す関数
    - f（属性値）→ バケット番号
    - f は任意の関数
      - 例、剰余（バケットが k 個ある場合、属性値を k で割った余りをハッシュ値とする）
  - あらかじめハッシュ関数が決められていれば、多段ツリーのように複数のノードを辿らなくとも、ある属性のレコードがどの領域（バケット）に格納されているかが分る

ハッシュ法の特徴

- ハッシュ法の特徴
  - インデックスのための領域が不要
  - データが正しく分散していれば、アクセス時やデータ挿入時の手間は $O(1)$ となり、高速
    - ただし、レコードが一部のバケットに偏っていると、最悪 $O(n)$ になる
  - うまくデータが分散するようなハッシュ関数を選ぶ必要がある
  - バケットがあふれた場合の処理が必要
  - 範囲検索には使えない（等号検索には有効）

ハッシュ関数の種類

- 静的ハッシング
  - 固定のハッシュ関数を使用する
  - 例：剰余（バケットが k 個ある場合、属性値を k で割った余りをハッシュ値とする）
- 動的ハッシング
  - ハッシュ関数を動的に変更する
  - 例：拡張ハッシュ法
    - キー値の上位 d ビットによりバケットを決定
    - バケットサイズが大きくなったら部分的に d を縮小する

データ格納方法の工夫

データの格納と検索（まとめ）

- ファイル内のデータの格納方法の種類
  - 順序なしの格納（ヒープ）
  - 順序付きの格納（リスト）
  - ハッシュを使った格納
- アクセス方法（データ格納位置の決定方法）
  - スキャン法
  - 探索法
  - インデックス
  - ハッシュ

格納方法と検索方法の対応（まとめ）

- 順序なしの格納 → スキャン法
- 順序付きの格納 → 探索法
- 順序なしの格納＋インデックス → インデックス法
- 順序付きの格納＋インデックス → インデックス法
- ハッシュを使った格納 → ハッシュ法

データの格納方法の組み合わせ

- 順序なし格納 ＋ 各属性に適宜インデックス
  - 一般的な構成
  - 属性の種類や更新頻度に応じて、ツリーやハッシュを使用
- キー属性で順序付き配列 ＋ 他の属性には適宜インデックス
  - こちらも一般的な構成
- ハッシュ
  - 値での検索（等号検索）が頻繁に起こる場合

データベース設計時の注意点

- データベースシステムでは、リレーションを作成する時に、格納方法、各属性にインデックスを付けるか付けないか（あるいはインデックスの種類）、を指定できるようになっているものが多い
- データベースの用途（主に実行される問い合わせの種類など）を考えて、なるべく処理効率が良くなるように設定を行う必要がある

データ配置の最適化

- より効率的に処理を行うためにデータの配置位置を最適化することもできる
  - 連続するデータがディスク内部でも連続して配置されるように工夫
  - 異なるリレーションの関連するデータが同じページ上に配置されるように工夫
    - 例：リレーション部門の部門番号とリレーション職員の部門番号が同じデータを同じページに配置
      → 部門と職員を自然結合した時、あるページを見れば両方のデータがあるので、アクセス回数を減らせる

まとめ

- 物理的データ格納方式
  - データ格納の概要
  - データ格納方法
  - データアクセス方法
  - データ格納方法の工夫

次回予告

- 同時実行制御
  - トランザクション
  - トランザクションの同時実行制御
  - 計画的な並行処理
  - 逐次的な並行処理