# データベース
# 【10:物理的データ格納方式】

石川　佳治

# 記憶媒体

# 記憶媒体

- 主記憶（primary storage）
  - 半導体メモリ：揮発性
  - CPUから高速にアクセス可能
  - ビット当たり価格は高価
  - 実際にはCPUと主記憶の間にレジスタ，キャッシュが存在
- 二次記憶（secondary storage）
  - 磁気ディスク，磁気テープ，光ディスクなど：不揮発性
  - CPUから直接はアクセスできない
  - 低速だが大容量
  - ビット当たり価格は低価格
  - 磁気テープは磁気ディスクに対するバックアップ（三次記憶）として用いられることもあり

# データベースの格納

- 一般に磁気ディスクに格納
  - データ量が多いため，主記憶上に保持することはコスト的に困難
  - 障害に対して安定的にデータを保持する必要性
- 磁気ディスクの特性
  - 二次記憶の中では，速度・機能とビット当たり価格のバランスに優れる
  - 磁気テープの方がビット当たり価格では有利だが，速度や使い勝手に劣る

# 記憶容量とアクセス速度

- 大まかな目安
  - 1次キャッシュ：64KB，1～5ns
  - 2次キャッシュ：2MB，10～30ns
  - 主記憶：数GB，30～100ns
  - 磁気ディスク：数百GB～数TB，10ms～20ms
- 実世界に例えると．．．

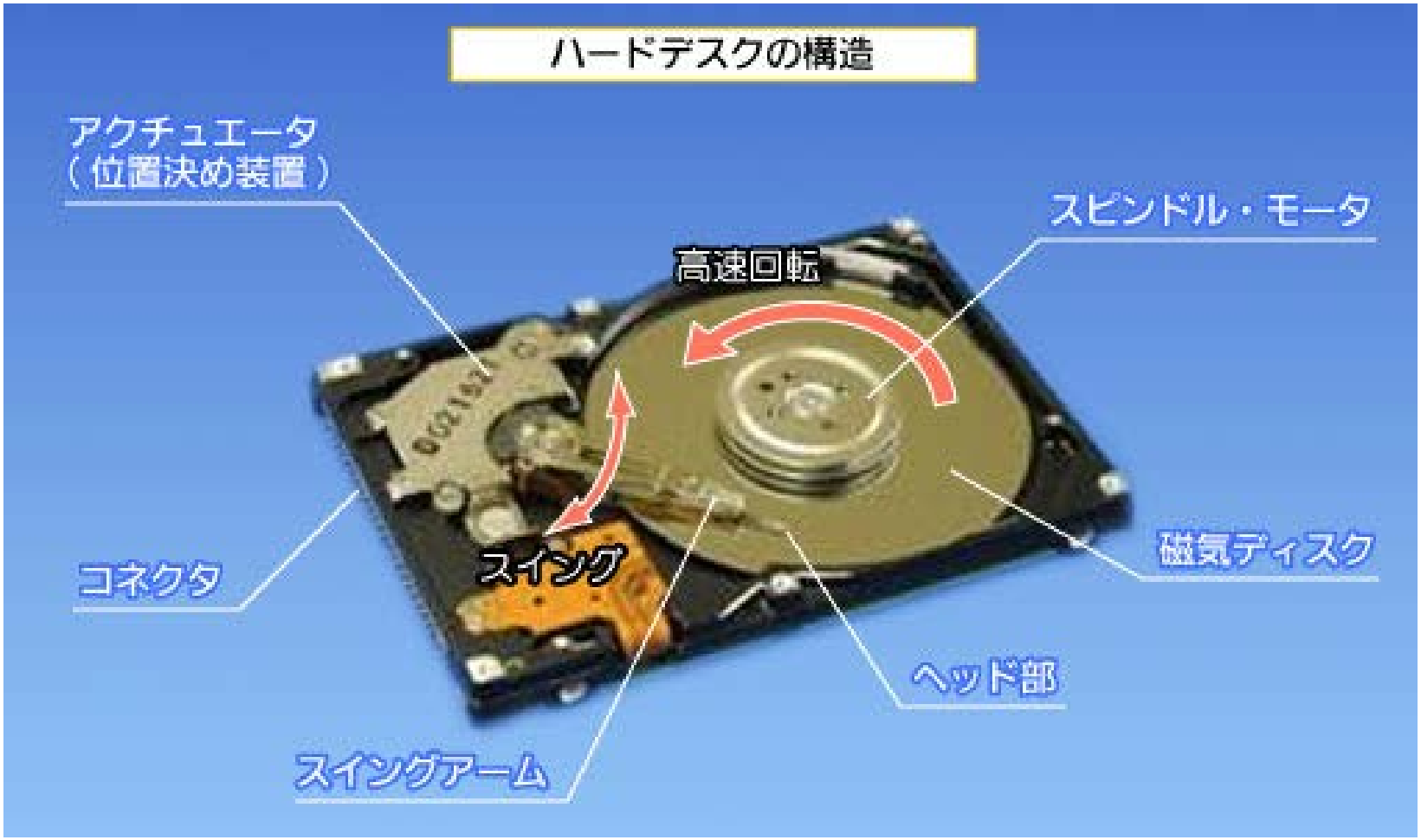
| | |
|---|---|
| キャッシュ（5ns） | 机の上の本（1秒） |
| 主記憶（50ns） | 本棚の本（10秒） |
| 磁気ディスク（10ms） | 名古屋駅の書店（2000秒=33分） |

  - 磁気ディスクは遅い!

# 磁気ディスクの構造

- 教科書 図6.1を参照
- 円盤状の磁性体(プラッタ)上に記録
- プラッタは高速に回転
- ヘッドを用いて読み書き:動作時にはわずかな間隔をあけて浮いている
- データ記録
  - トラック:円盤上の同心円
  - セクタ:トラック上の区画で記録の単位
  - ページ(ブロック):複数の連続するセクタで読み書きの単位(4KB～8KB程度)
- 読み書きの処理
  - シーク時間，回転待ち時間，転送時間が必要

# 磁気ディスクの図


ハードデスクの構造
アクチュエータ
（位置決め装置）
スピンドル・モータ
高速回転
コネクタ
スイング
磁気ディスク
ヘッド部
スイングアーム

# 磁気ディスクへのアクセス

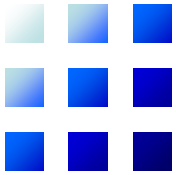
- 磁気ディスクの特性を考慮
- バッファリング（buffering）
  - 主記憶中に最近アクセスしたページを一定数保持
  - 同じページへのアクセス要求が来たときには，主記憶上のページを利用
- 同時にアクセスされる可能性が高いデータは同一ページに：アクセス数を減らす効果
- 連続したデータ読み込みの場合は，先のページの転送の間に次のページを先読み

# フラッシュメモリ（Flash Memory）

- 不揮発性のメモリ
  - 電源を切っても情報が失われない
  - 携帯機器の普及にともない大容量化・低価格化
  - 記憶素子には書込み・消去数の上限あり
  - 通常のメモリに比べると遅い：1桁程度
- NAND型フラッシュメモリ
  - データストレージに向いたフラッシュメモリ
  - ノートPCのSSDドライブでも利用
  - 細線化には限界あるので多層化で対処
    - 現在48層構造で，今後64層構造のフラッシュメモリが製品化

# フラッシュメモリと磁気ディスク

- フラッシュメモリの利点
  - 高速：磁気ディスクと1桁違う
  - 発熱が少なく，消費電力が小さい
  - 衝撃に強く，故障が少ない
- フラッシュメモリの欠点
  - 高価：ビット当たり単価では，磁気ディスクの数倍
- 現状は過渡的
  - データセンターになどにおける磁気ディスクがNAND型フラッシュメモリのストレージに置き換わっていく



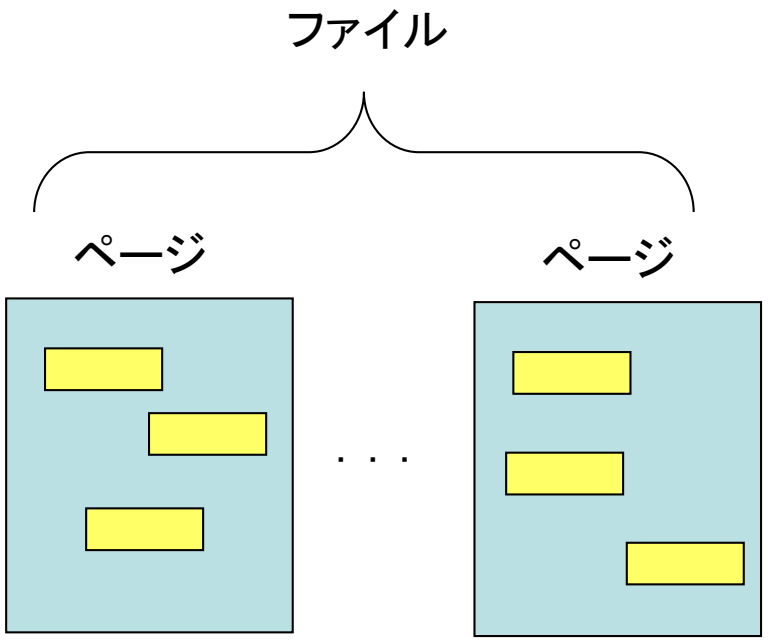
# レコードとファイル

# レコードとファイル（1）

- レコード
  - リレーションのタプルの内部表現
  - フィールド：タプルの属性値に応じて複数存在
  - さまざまな情報を管理
    - レコードフォーマット，レコードサイズ，フィールドサイズなど
- ファイル
  - 関連するレコード集合を組織化したもの
  - OSの「ファイル」とは必ずしも一致しない

# レコードとファイル（2）

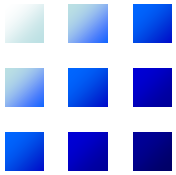
- レコードの管理方式
  - ディスクのあるページ中の連続領域に割り当て
- レコードの分類
  - 固定長レコード
    - サイズが一定
    - ページ内に密に格納可能
  - 可変長レコード
    - サイズが可変
    - 固定長レコードに比べ管理が複雑



# ファイル編成

- 一般に，レコードをページに割り当てる方式を指す
- 代表的なファイル編成について後述
  - ヒープファイル
  - ハッシュファイル
  - 索引付きファイル
  - B木，$B^+$木
  - 二次索引

  ※ 固定長レコードについて議論
- キー：フィールドの組合せのうち，レコードの位置決めに用いるもの



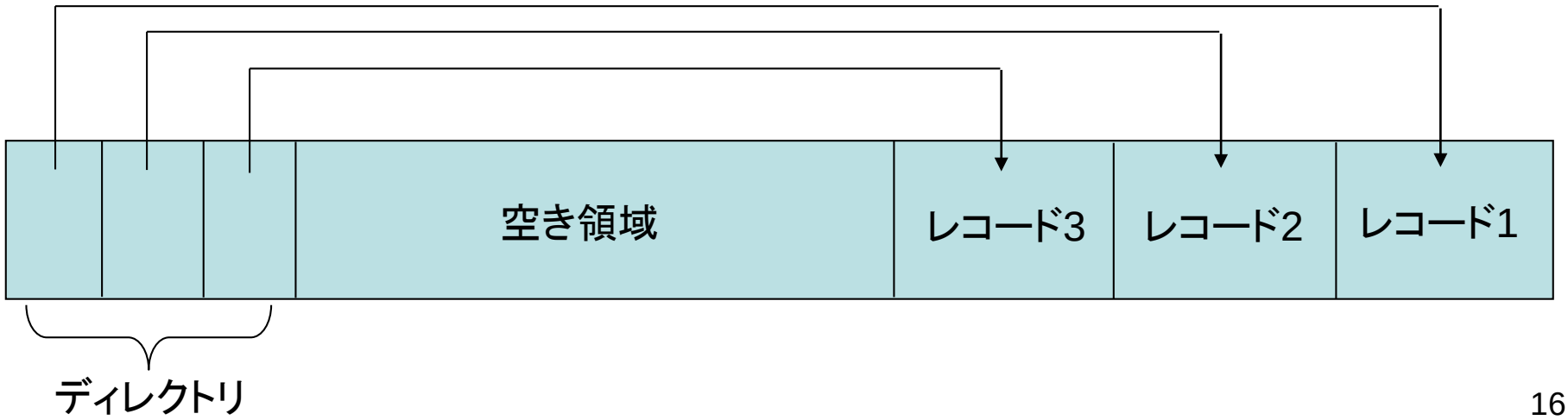
# ヒープファイル

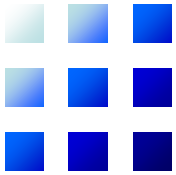
# ヒープファイル（heap file）

- ページ内にレコードを順次格納したもの
- レコードの挿入
  - あるページに空き領域があればそこに追加
  - なければ新しいページを割り当て
- レコードの削除
  - ページ内で該当レコードを削除
- 特徴
  - 格納効率が良い
  - キー値を用いたレコード検索ができない

# ヒープファイルのレコード配置

- オフセット情報を格納したスロットからなるディレクトリを利用
  - レコードは(ページ番号，スロット番号)の組で識別：レコード識別子と呼ぶ
- ページ内でのレコードの位置が変わっても，レコードを一意に識別可能



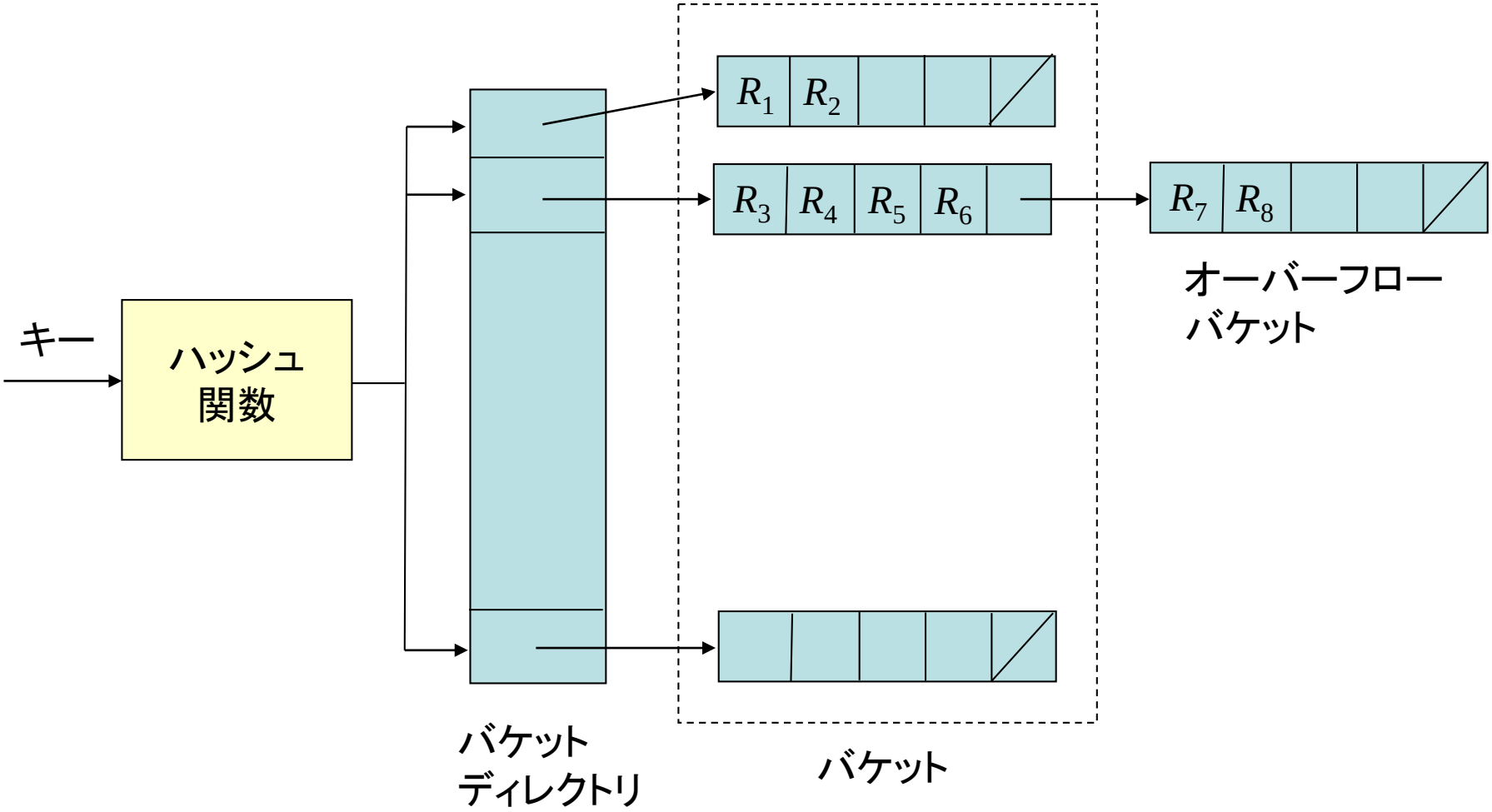


# ハッシュファイル

# ハッシュファイル（hash file）

- レコード $R_i$ のキー値をハッシュ関数でハッシュし，格納すべきバケットを決定
- バケット：一つまたは連続した複数のページから構成
- バケットディレクトリ：バケットのアドレスを格納
- 検索処理：キー値をハッシングしてバケットを決定し，該当するページのみを呼び出す
- 挿入処理
  - ハッシングで格納するバケットを決定
  - 該当バケットに空きがないならオーバーフロー処理を実行
    - いくつかの手法が存在
- 範囲検索，キー値の順でのレコード読み出しには対応できない

# ハッシュファイルの構造

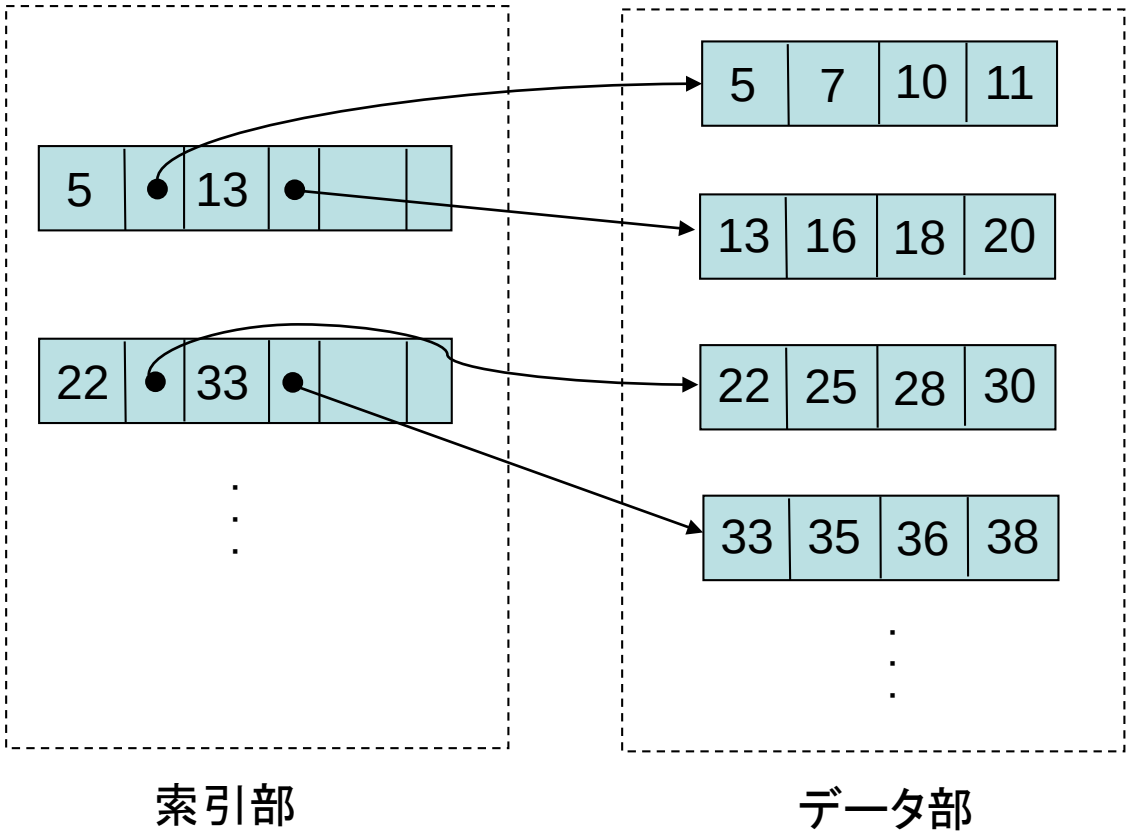


# 索引付ファイル

# 索引付ファイル（indexed file）

- 構成
  - データ部：一連のデータページに，レコードをキー値の順に格納
  - 索引部：一連の索引ページに索引レコードを順に格納
    - 索引レコードは，（データページの先頭レコードのキー値，そのデータページへのポインタ）というペア
- 検索処理
  - 索引ページを探索し，読むべきデータページを特定して，該当データページを読み出す
  - キー値の順に全レコードを読み出すことも可能
  - 範囲検索も可能
- 問題点：新たなレコードの挿入
  - 索引の途中にキー値を挿入する処理のコストが高い
  - オーバーフロー用データページなどで対応

# 索引付ファイルの構成



# B木

# B木（B-tree）

- 動的に挿入，削除が発生する状況に適したファイル編成手法
- データベースの場合，$B^+$木（後述）が最もよく利用される：B木はその基礎となる
- B木は主記憶上のデータ構造
- 一方，$B^+$木は二次記憶に対応したデータ構造

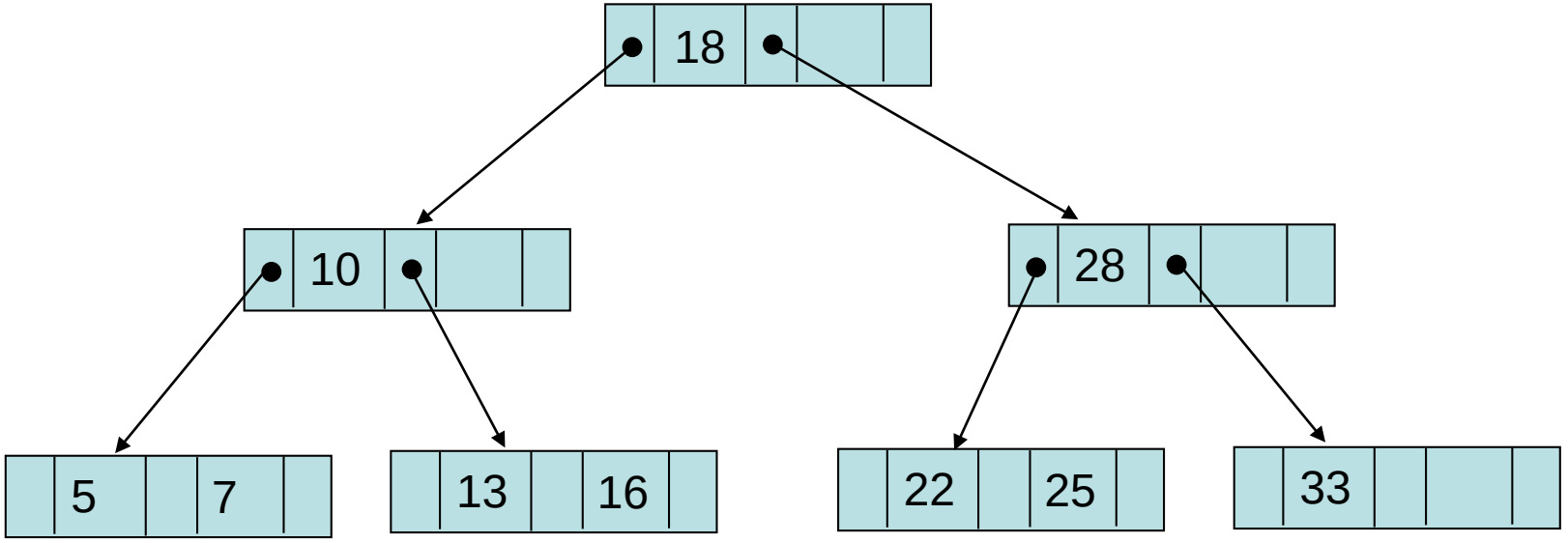
# B木の構造(1)

- $d$ 次のB木:ページをノードとし以下の条件を満たすルート付木
  - ① ルートから各リーフノードまでのパスの長さはすべて同じ
  - ② ルート以外のノードは，キー値の順に並んだ $i$ 個($d \leq i \leq 2d$)のレコード $R_1, \ldots, R_i$ をもつ
  - ③ $i$ 個のレコードを持つ非リーフノード $N$ は $i+1$ 個の子ノードポインタ $PTR_1, \ldots, PTR_{i+1}$ をもつ.

# B木の構造(1)

- (③の続き)ポインタ $PTR_j$ $(1 \leq j \leq i+1)$ の指す部分木に格納されたすべてのレコードのキー値 $K$ は次の条件を満たす.
  - $a.\ j = 1$ のとき, $K < Key(R_i)$
  - $b.\ i < j < i+1$ のとき, $Key(R_j - 1) < K < Key(R_j)$
  - $c.\ j = i+1$ のとき, $Key(R_i) < K$
- 条件①を満たす木をバランス木という
- ルートからリーフノードまでのパスの長さを木の高さと呼ぶ

# B木の例(1次の場合)




- 探索の際は，必ずしもリーフまで辿る必要はない：探しているキー値が非リーフノードに見つかれば，そこで終了

# レコード挿入処理(1)

① 挿入レコードのキー値によりB木をたどり，リーフノード $N$ を特定

② $N$ のレコード数が $2d$ 個未満なら，N にレコードを挿入して終了．それ以外は③へ

③ (オーバーフローが発生)ノードを分割

a. 新たなノード $N'$ を確保

b. 挿入レコードを含む $2d+1$ 個のレコードのうち，キー値の小さい順に $d$ 個を $N$ に，大きい順に $d$ 個を $N'$ に格納

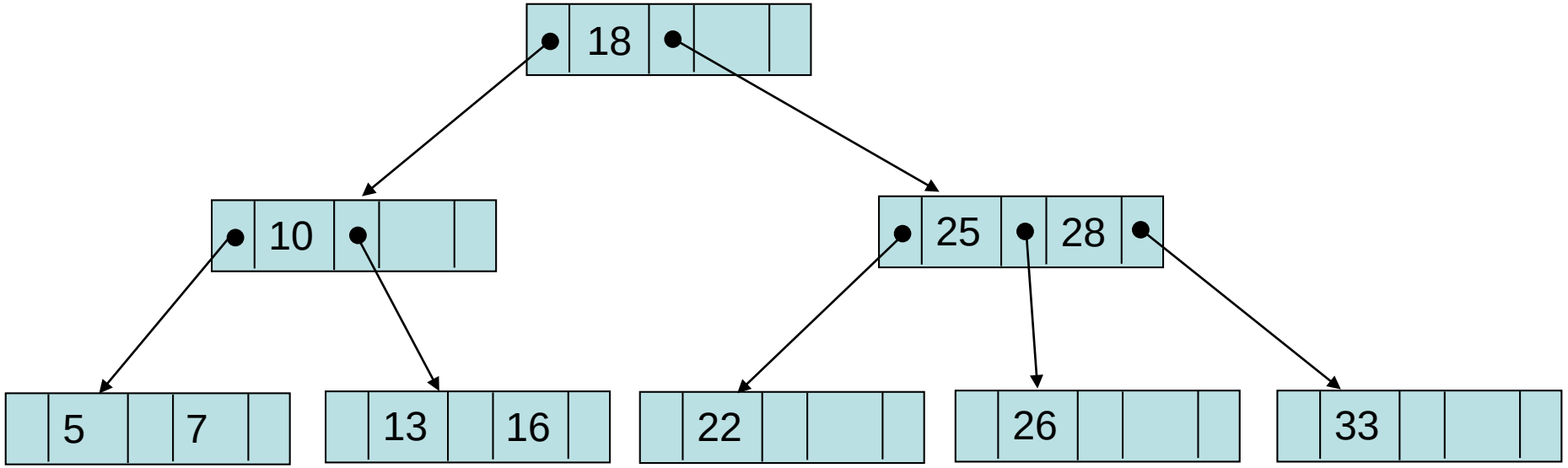
c. 中間のキー値を持つレコード $R$ と $N'$ へのポインタ $PTR(N')$ をペアにして，$N$ の親ノードに挿入

# レコード挿入処理（2）

④ 親ノードでは $R$ と $PTR(N')$ のペアを挿入レコードとみなして，②③と同様に処理．

- ノードの分割がルートまで波及し $N$ がルートになった場合：③で新たなノードをルートとして確保し，$PTR(N), R, PTR(N')$ を格納 ⇒ B木の高さが1増加

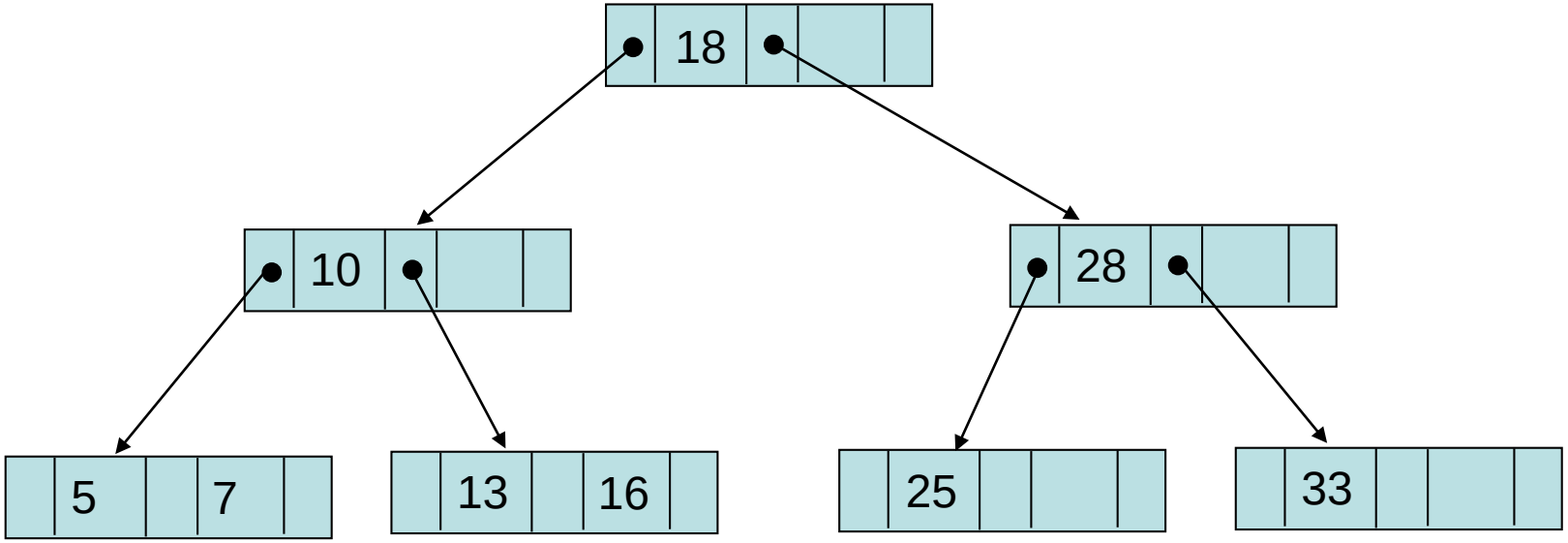
# B木の例:挿入後




- キー値26のレコードを挿入した結果

# 削除処理(1)

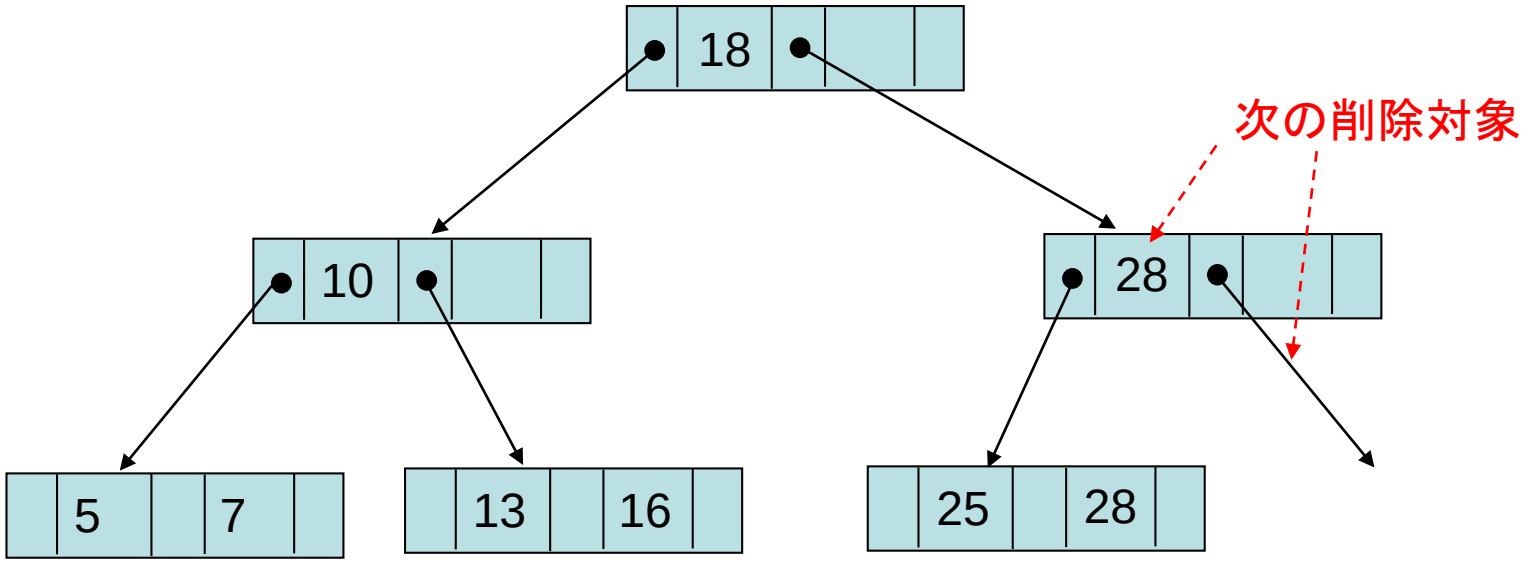
- 省略:教科書を参照




- 図6.5から22を削除した例

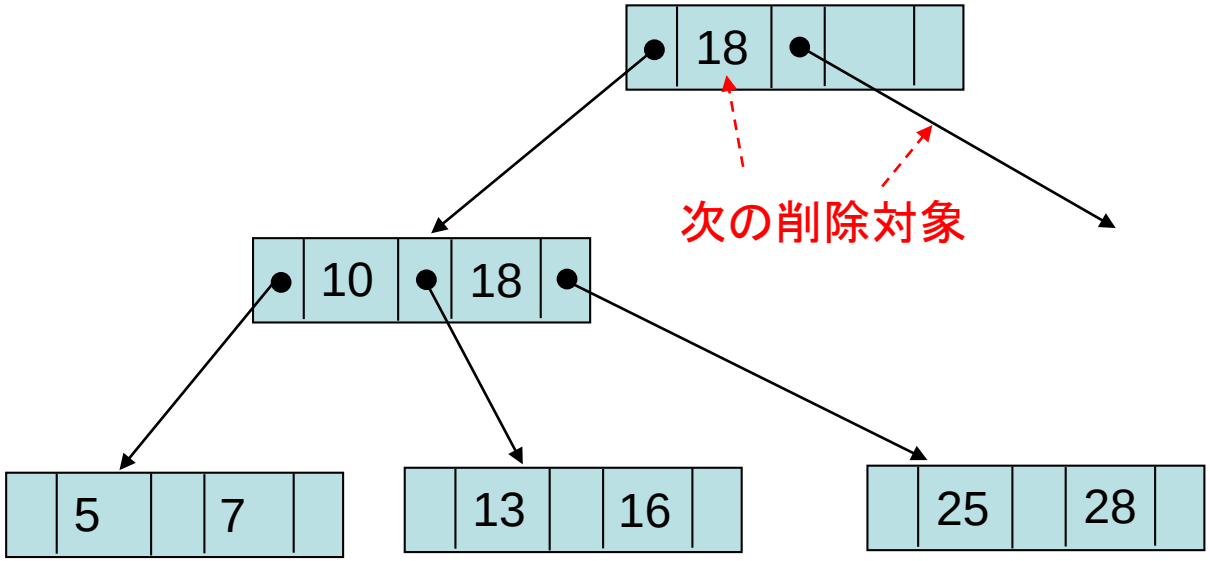
# 削除処理(2)

- 次いで33を削除:「$N$ がリーフノードの場合」のステップ④を実行した時点のB木
  - 33を削除，対応するノードも削除
  - 28を左の子ノードに挿入



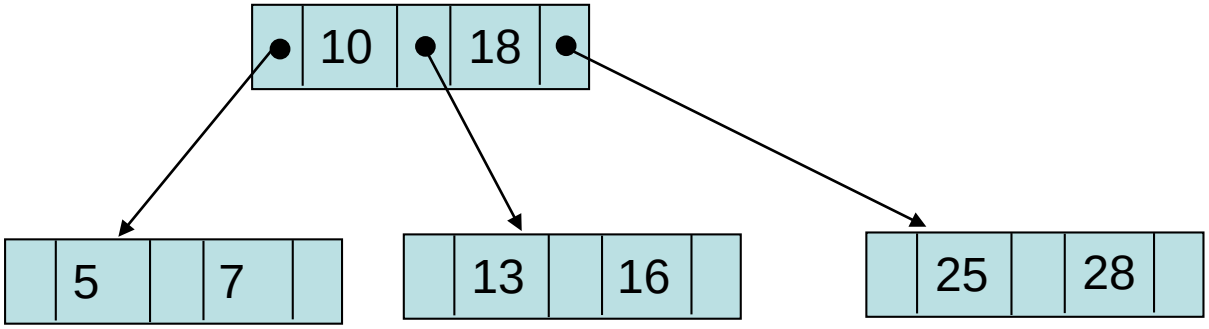
# 削除処理(3)

- 28を中間のノードから削除するとアンダーフローが発生:同様にノードを統合する
  - 28を削除，対応するノードも削除
  - 18を10のノードに挿入
  - 中間ノードの18の右のポインタを修正



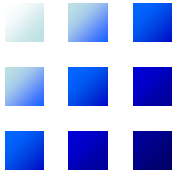
# 削除処理(4)

- ルートから18を削除すると，ルート中のレコード数に関する制約条件を満たさなくなる
  - ルートノードを削除



# B木に関するまとめ

- B木の利点
  - レコードの挿入・削除に対して動的に再構成可能
  - 高速：レコード検索時に読み込むページ数が少ない（ディスクアクセス回数が少ない）
    - $n$ 個のレコードを格納したB木の高さは $O(\log(n))$
  - 空間使用効率が比較的良い
    - ルートを除いて，最低50%の空間使用効率を保証
    - 平均的な空間使用効率は69%
- データ構造の定義，挿入・削除アルゴリズムにはさまざまなバリエーションが存在
  - 削除処理（アンダーフローの際）で工夫する理由
    - 大幅な構造の変化は避けたい
    - すぐ後にまた挿入処理が起きることが多い



# $B^+$木

# $B^+$木（$B^+$-tree）

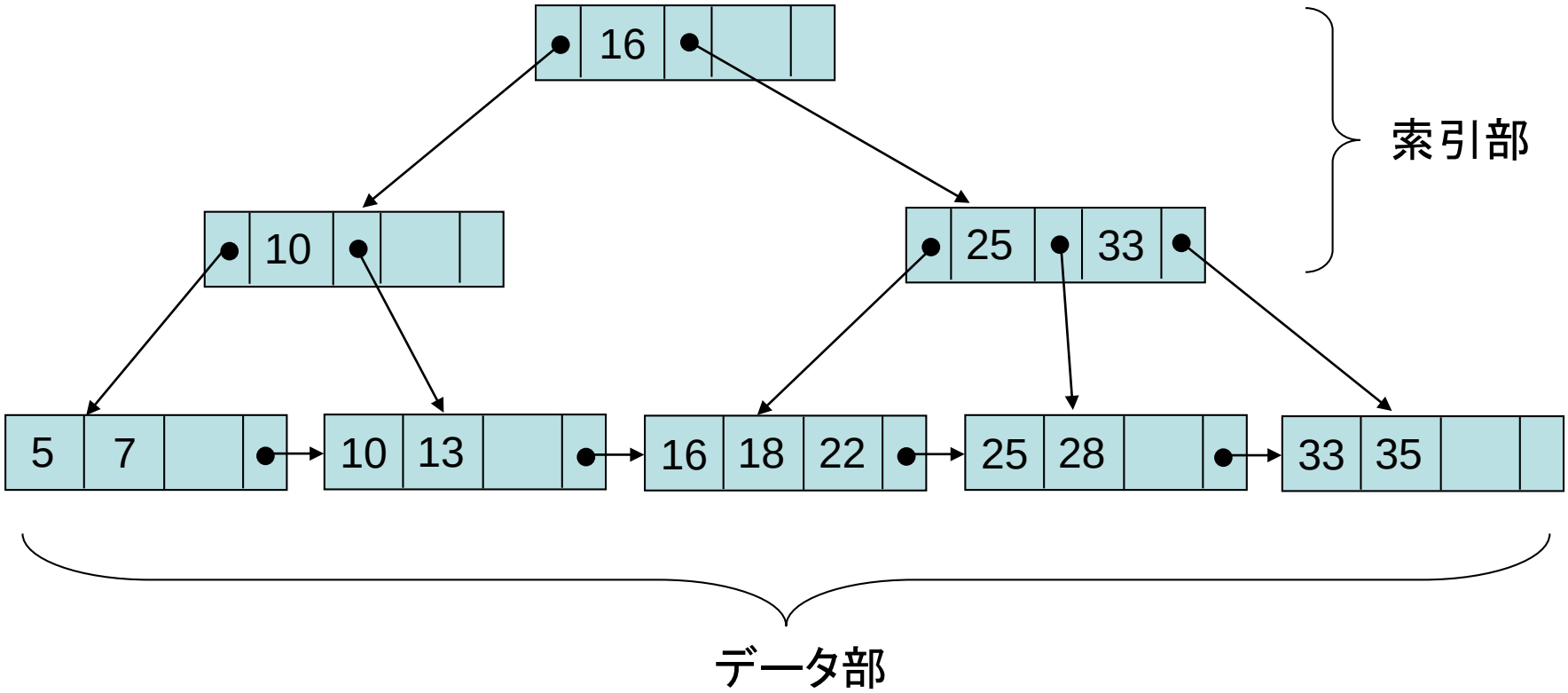
- B木を変形したデータ構造で，DBMSで最もよく用いられる
- B木との相違点
  - レコードはリーフノードにのみ格納：データ部に相当
  - 非リーフノードにはキー値のみを格納：索引部に相当

# B+木のデータ構造

- 索引部（非リーフノード）
  - レコードでなくキー値が入ること以外はB木と同じ
- データ部（リーフノード）
  - レコードが入る
  - キー値の順に並んだ $i$ 個のレコード $R_1, \ldots, R_i$ をもつ．ただし，$i$ は与えられたパラメータ $e$ に対して $\left\lceil \frac{e}{2} \right\rceil \leq i \leq e$ を満たす
  - 通常，リーフノード同士を結ぶ横方向のポインタを設ける（一方向 or 両方向）

# B+木の例

$d = 1, e = 3$



# 検索処理

- キー値による検索
  - ルートを最初のノードとし，キー値に応じて子ノードポインタをリーフノードまでたどる（B木との相違点）
- キー値の順での読み出し
  - データ部のリーフノードを順次読み出す
- 範囲検索
  - 索引部を用いて先頭リーフノードを特定
  - そこから横方向のポインタをたどって範囲を超えるまでリーフノードを読む

# レコード挿入（1）

① 挿入レコードのキー値を用いてB$^+$木をたどり，リーフノード $N$ を特定

② $N$ に入っているレコード数が $e$ 個未満なら，$N$ の適当な位置に挿入レコードを格納して終了．それ以外は③へ

③ （オーバーフロー処理）新たなノード $N'$ を確保し，挿入レコードも含めた $e+1$ 個のレコードをキーの小さい順に $\left\lceil \frac{e+1}{2} \right\rceil$ 個を $N$ に，残りを $N'$ に

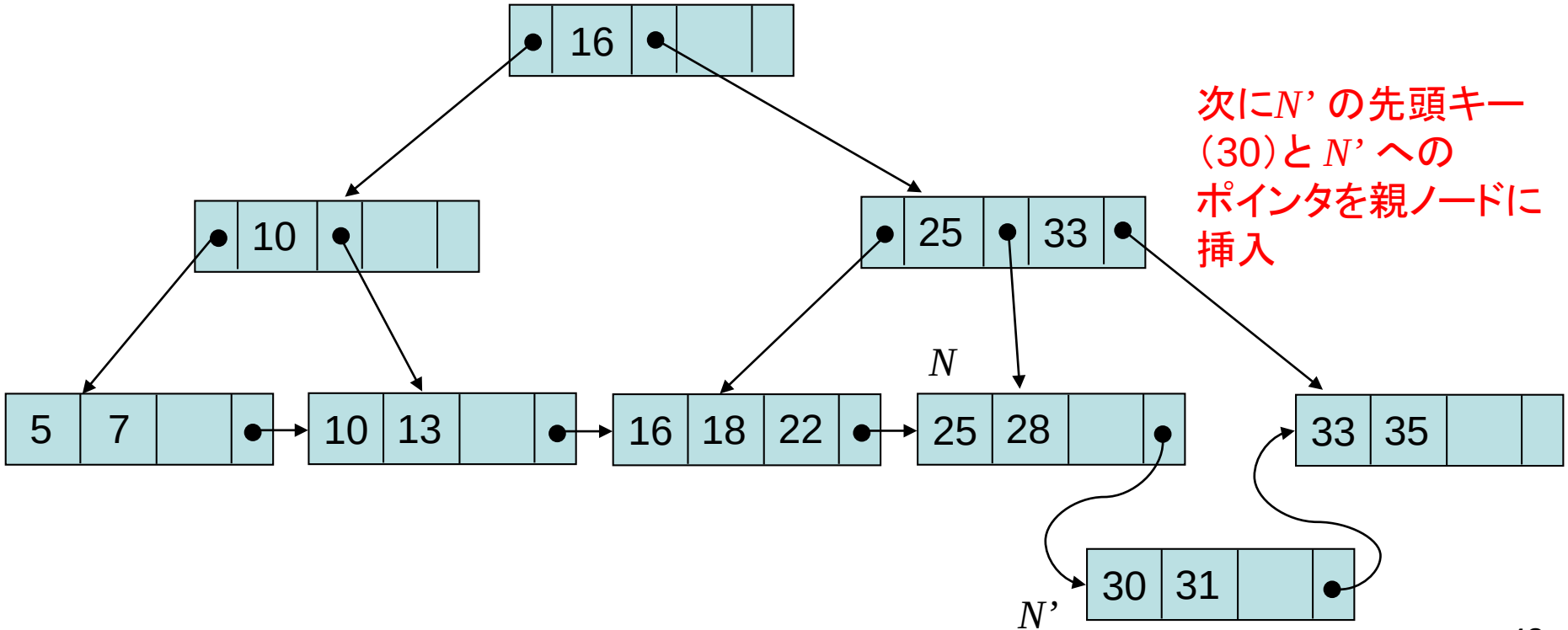
# レコード挿入（2）

③ （続き）$N'$ の先頭レコードのキー値 $K$ と $N'$ へのポインタ $PTR(N')$ をペアにして，$N$ を指す索引部の親ノードへ挿入

④ 親ノードでは $K$ と $PTR(N')$ のペアを挿入レコードとして挿入処理を行う：B木の手順に従う

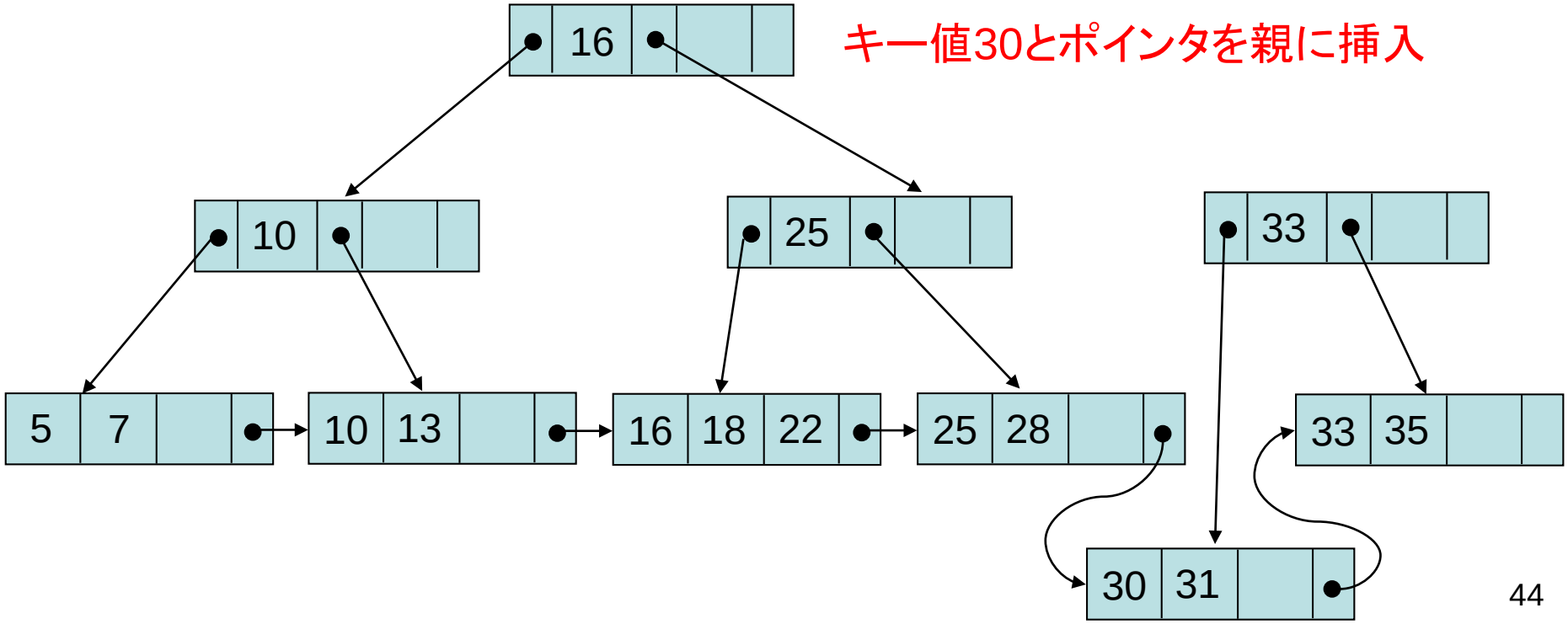
# 挿入の例(1)

- 図6.8に，まず30を挿入
- 次いで31を挿入したいがオーバーフローするので，手順③に従う



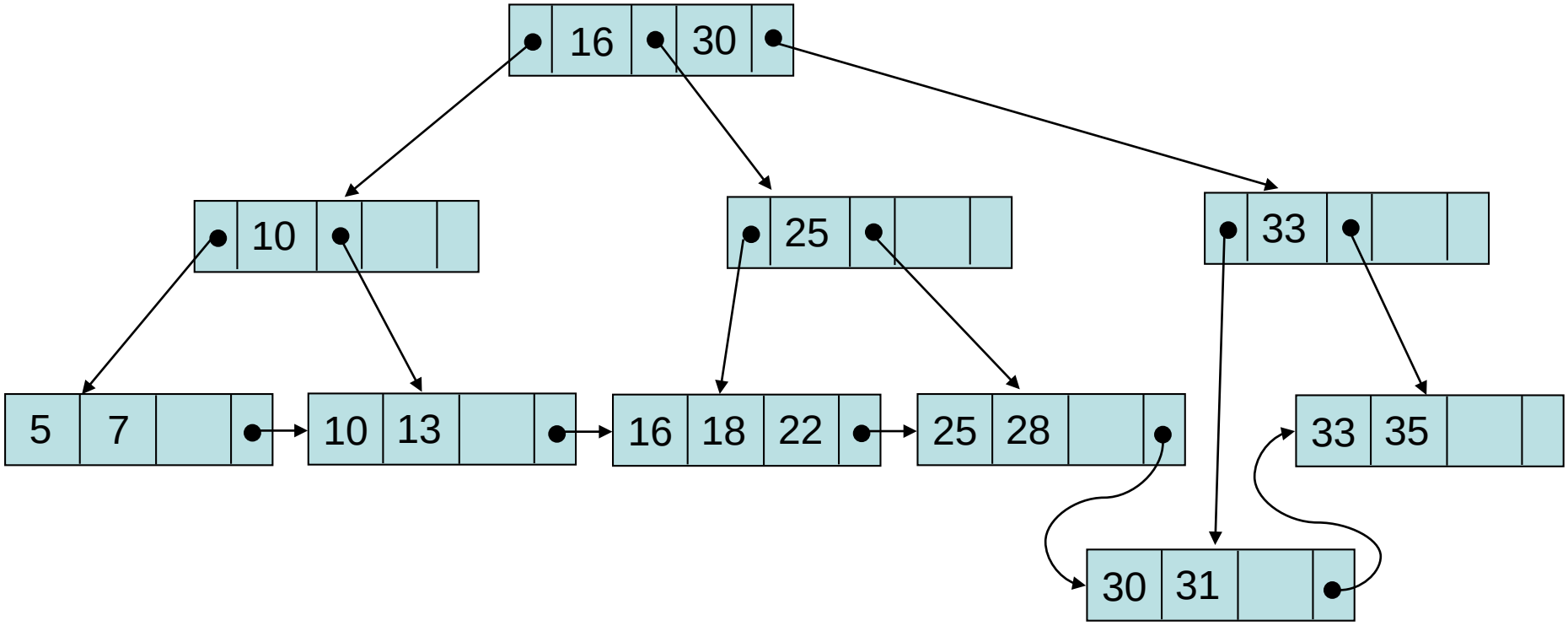
# 挿入の例(2)

- 親ノードもオーバーフローした
  - 2つのキーしか入らないのに，25, 30, 33が存在
  - B木の手順に従い処理：ノードを分割．左ノードへ25, 右ノードへ33を振り分け，30はさらに親へ



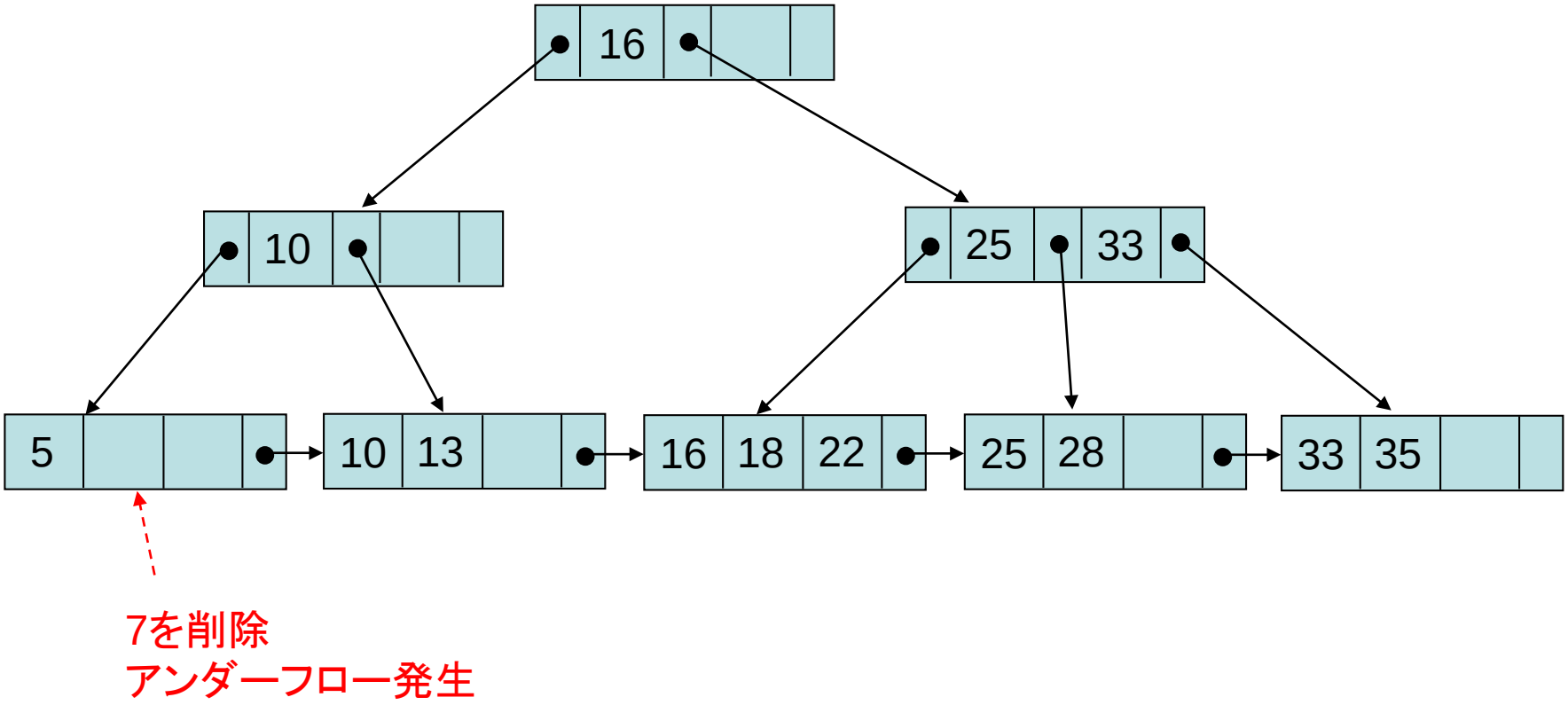
# 挿入の例(3)

- 完成版



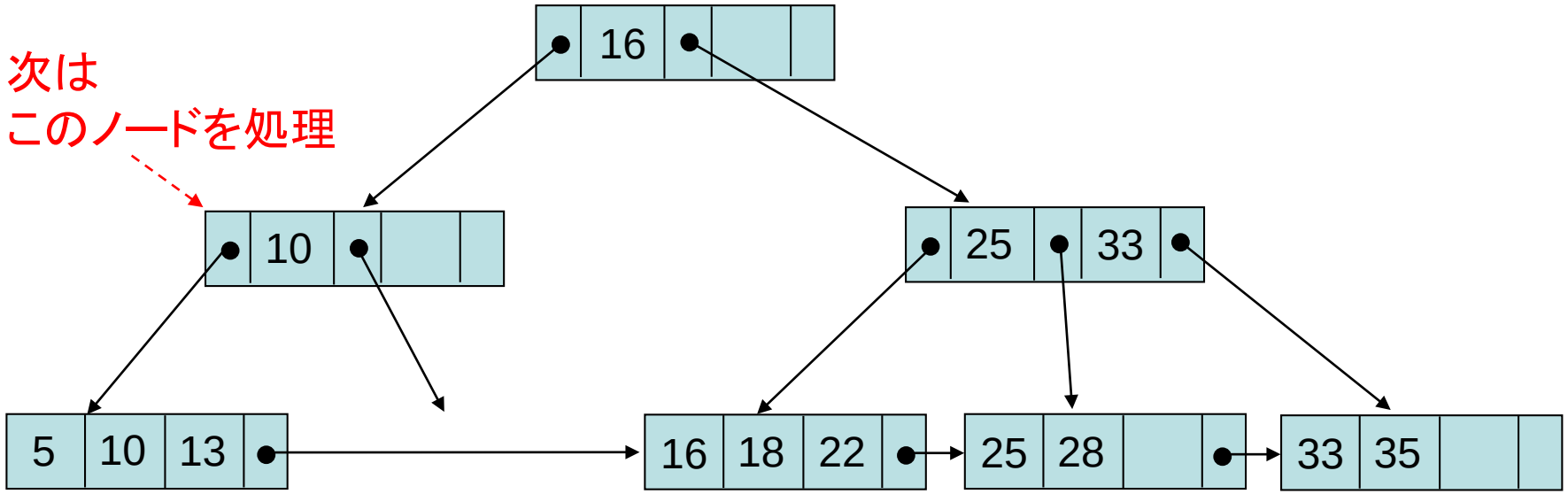
# 削除処理(1)

- 省略．教科書を参照
- 図6.8からキー値7のレコードを削除する例



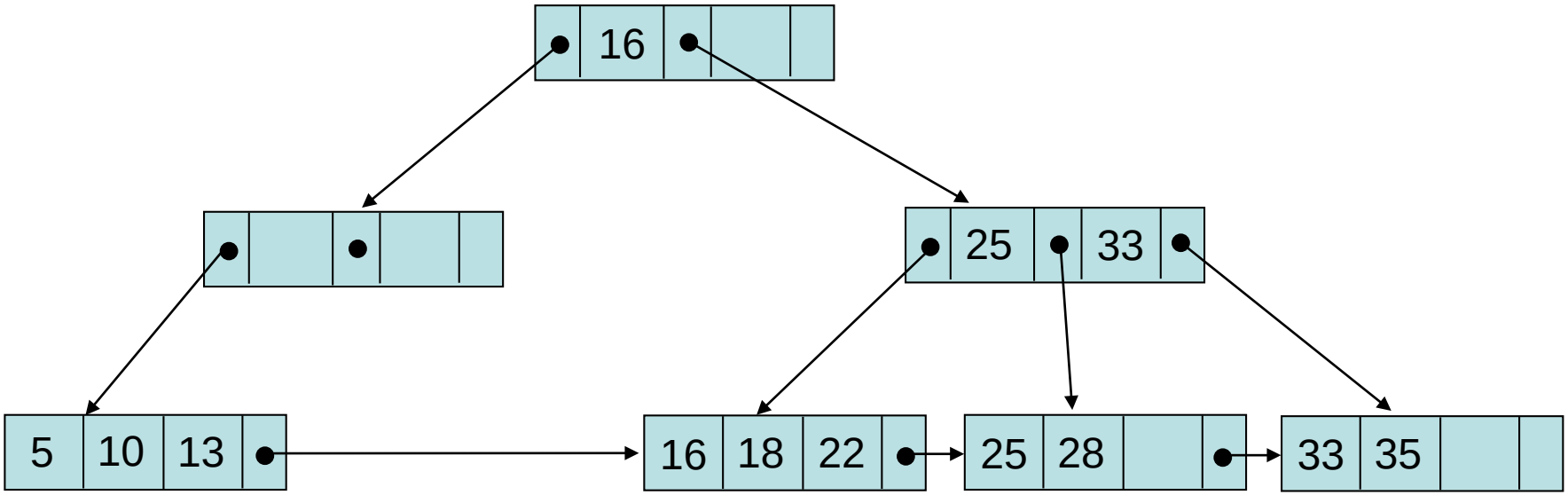
# 削除処理(2)

- 削除処理手順の④に従い隣のノードと融合する
- 隣接ノードを削除



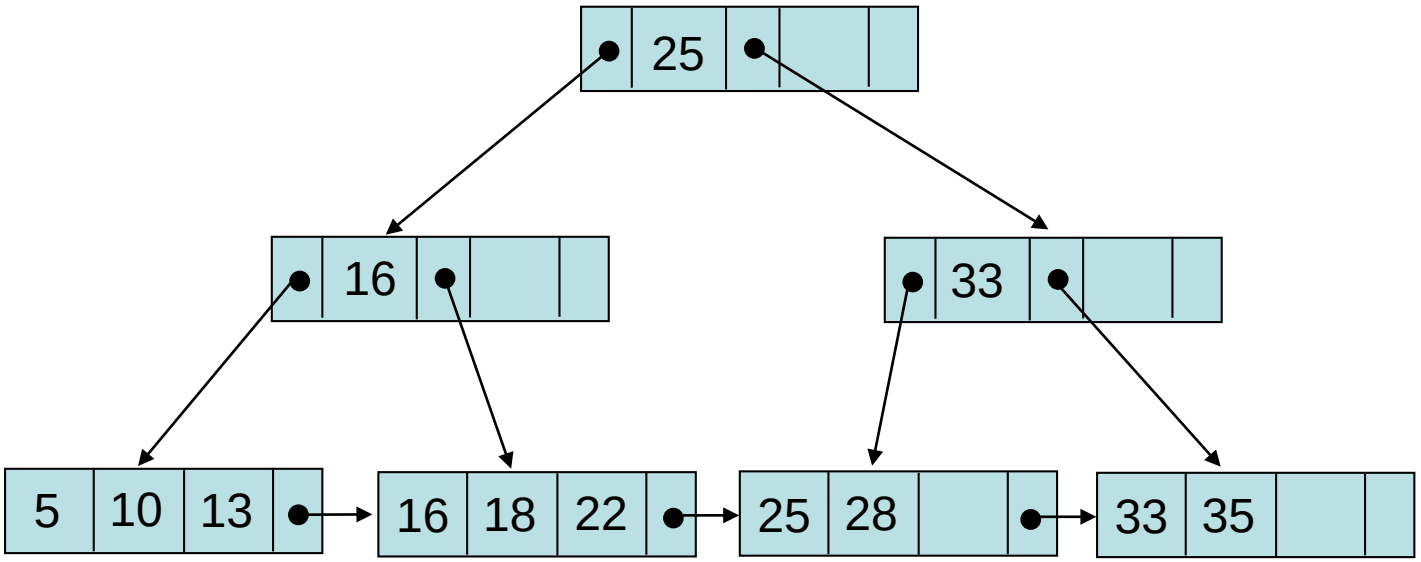
# 削除処理(3)

- 削除処理手順の④に従い，対象ノードからキー値10と削除したノードへのポインタを削除する
  - アンダーフローが発生



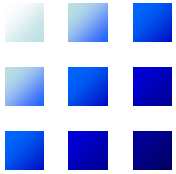
# 削除処理(4)

- 右の兄弟ノードに余裕があるので，B木のアンダーフロー手続き(p. 113の③)を利用



# $B^+$木の特徴

- B木の利点を受け継ぐ
- B木と比較した特徴
  - ① 全レコードをキー値順に読み出す処理，範囲検索処理などを効率よく処理可能
  - ② 索引部のノードにはキー値のみが格納される：レコードを格納する場合より（キー値がコンパクトであるため），より多くのキー値をノードに収容可能
    ⇒多くの子ノードポインタを出すことができる
    ⇒横広がりで高さの低い木構造にできる
    ⇒検索処理などにおけるページアクセス数の減少
- DBMSではB木よりも$B^+$木が用いられる
  - $B^+$木のことをB木と呼ぶことも多い



# 二次索引

# 主索引（primary index）

- キー値とレコードの格納場所を直接結びつける索引構造を伴うファイル編成法
- ハッシュファイル，索引付ファイル，$B^+$木
- 一般にレコードの主キーをファイル編成上のキーとして利用
- 主キーが与えられた際のレコード検索は，主索引を用いて効率的に実行可能

# 二次索引(secondary index)

- 主キー以外のフィールドによる検索のためのファイル編成法
- 索引フィールド(indexing field):索引を構成するためのフィールドを総称
- 主索引との違い
  ① データレコードはデータファイルに格納済みなので,データレコードへのポインタのみが得られればよい
  ② 指定した索引フィールド値を持つデータレコードは,ファイル中に複数存在しうる(主キーでないので)

# 二次索引の構成

- 索引フィールド値 $V$ と該当データレコードへのポインタ列 $\{P_1, \ldots, P_n\}$ の二つのフィールドを持つ索引レコード $(V, \{P_1, \ldots, P_n\})$ からなるファイル
- ファイル編成方式：B$^+$木が一般的
- データレコードへのポインタ：さまざまな手法あり
- 問題点
  - 索引レコード $V$ ごとにポインタ列の長さ $n$ が異なり，更新に伴い長さが変化する：ファイル構成に工夫が必要

# 主索引と二次索引の違い(1)

- 例:リレーション 学生(学生番号, 氏名, 専攻, 年齢)のファイル編成
- 想定:
  - 学生レコードを主索引で管理:学生番号がキー
  - 年齢について二次索引を作成

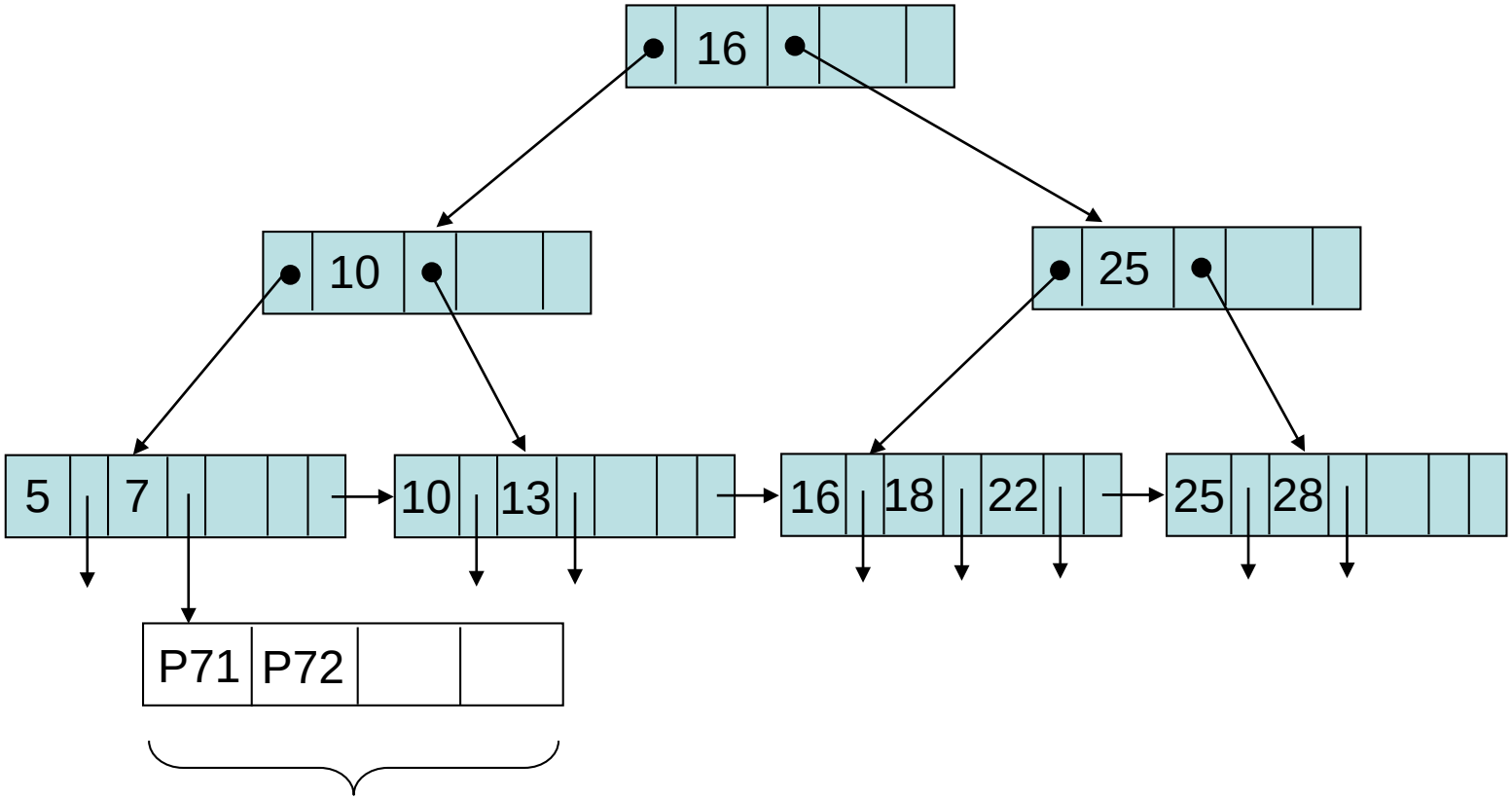
| 学生番号 | 氏名 | 専攻 | 年齢 |
|---|---|---|---|
| 00001 | 山田一郎 | 情報工学 | 20 |
| 00002 | 鈴木明 | 情報工学 | 19 |
| 00003 | 佐藤花子 | 知識工学 | 20 |
| … | … | … | … |

# 主索引と二次索引の違い(2)

- 主索引のレコード
  - (00001, 山田一郎, 情報工学, 20)
  - (00002, 鈴木明, 情報工学, 19)
  - (00003, 佐藤花子, 知識工学, 20)
  - …
- 二次索引のレコード
  - (20, {$RID_1$, $RID_3$, …})
  - (19, {$RID_2$, …})
  - …
- $RID_i$ は，主索引のi番目のレコードのレコード識別子(教科書p.107)
  - ページ番号，スロット番号の組などで表される

# 二次索引の構成例

- $B^+$木を用いた例



# 索引の分類

- 疎な索引・密な索引
  - 疎な索引（sparse index）：データレコードのもつすべての索引フィールド値（キー値）が索引に現れない索引
    - 索引付ファイル，$B^+$木（主索引の場合：6.7節）
    - 索引部だけ見ると，すべてのキー値は現れない
  - 密な索引（dense index）：すべての索引フィールド値が出現
- クラスタリング索引（clustering index）
  - 索引に用いられる索引フィールドの順にデータレコードがまとめられている索引
    - 索引付ファイル，$B^+$木（主索引の場合）などが相当
  - 索引フィールド値による範囲検索の際に読み出すデータファイルのページ数は最小

# 実際のファイル編成

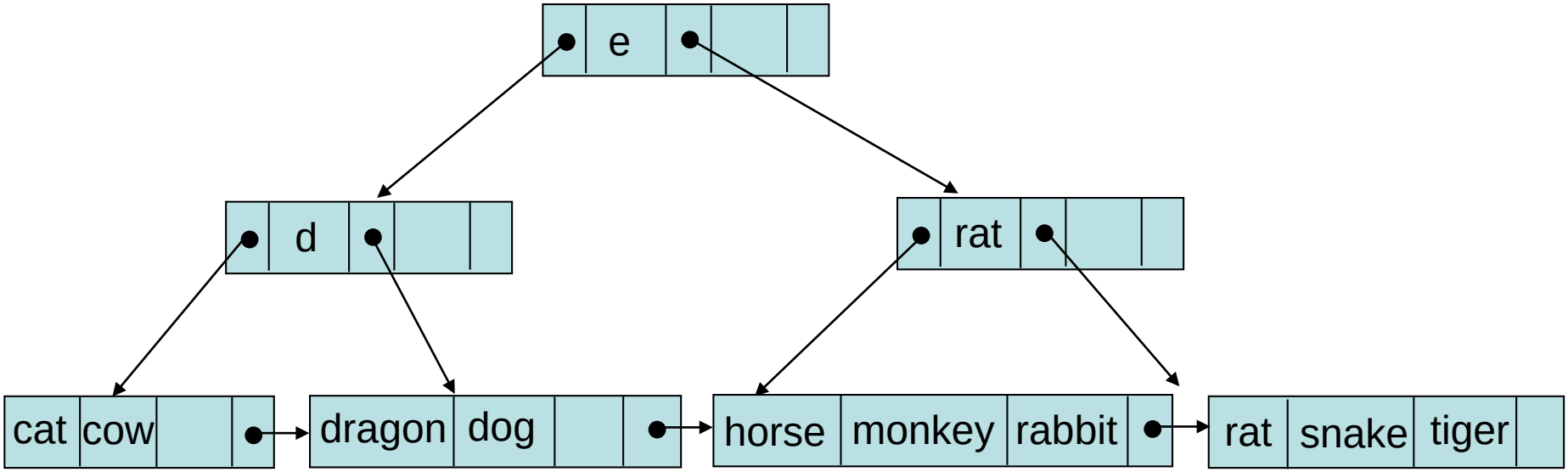
- ファイル編成方式を組み合わせる
- 例
  - 対象のリレーション（例：学生）を，主キー（学生番号）に基づきB$^+$木で管理
  - 必要に応じて属性（例：年齢）に二次索引を付与
- 注意点
  - 問合せの条件指定（例：WHERE 年齢 = 20）で頻繁に用いられる属性には二次索引を付与
    - 大幅な性能向上：詳しくは7章
  - 索引の管理にはオーバヘッドが伴う
  - 最近の商用DBMSにはチューニング機構が存在

# 補足1：接頭辞B$^+$木（prefix B$^+$-tree）（1）

- 文字列をキーとする場合に利用
- 非リーフノードのキーを接頭辞を用いて圧縮：効率を高める
- 非リーフノードのキー値 $v$ は，「左部分木における最大のキー値 $< v \leq$ 右部分木における最小のキー値」を満たす文字列
- 文字列がキー値の場合，大小関係は辞書式順序に従う

# 補足1：接頭辞B⁺木（prefix B⁺-tree）（2）

- 接頭辞B⁺木の例



# 補足2:ビットマップ索引(1)

- カテゴリ属性の存在
  - 専攻，性別，学年などの属性はとりうる値が少ない

| 学生番号 | 氏名 | 専攻 | 性別 | 学年 |
|---|---|---|---|---|
| 00001 | 山田一郎 | 情報工学 | 男 | 3 |
| 00002 | 鈴木明 | 知識工学 | 男 | 2 |
| 00003 | 佐藤花子 | 知識工学 | 女 | 3 |
| … | … | … | | … |

  - 問合せ例

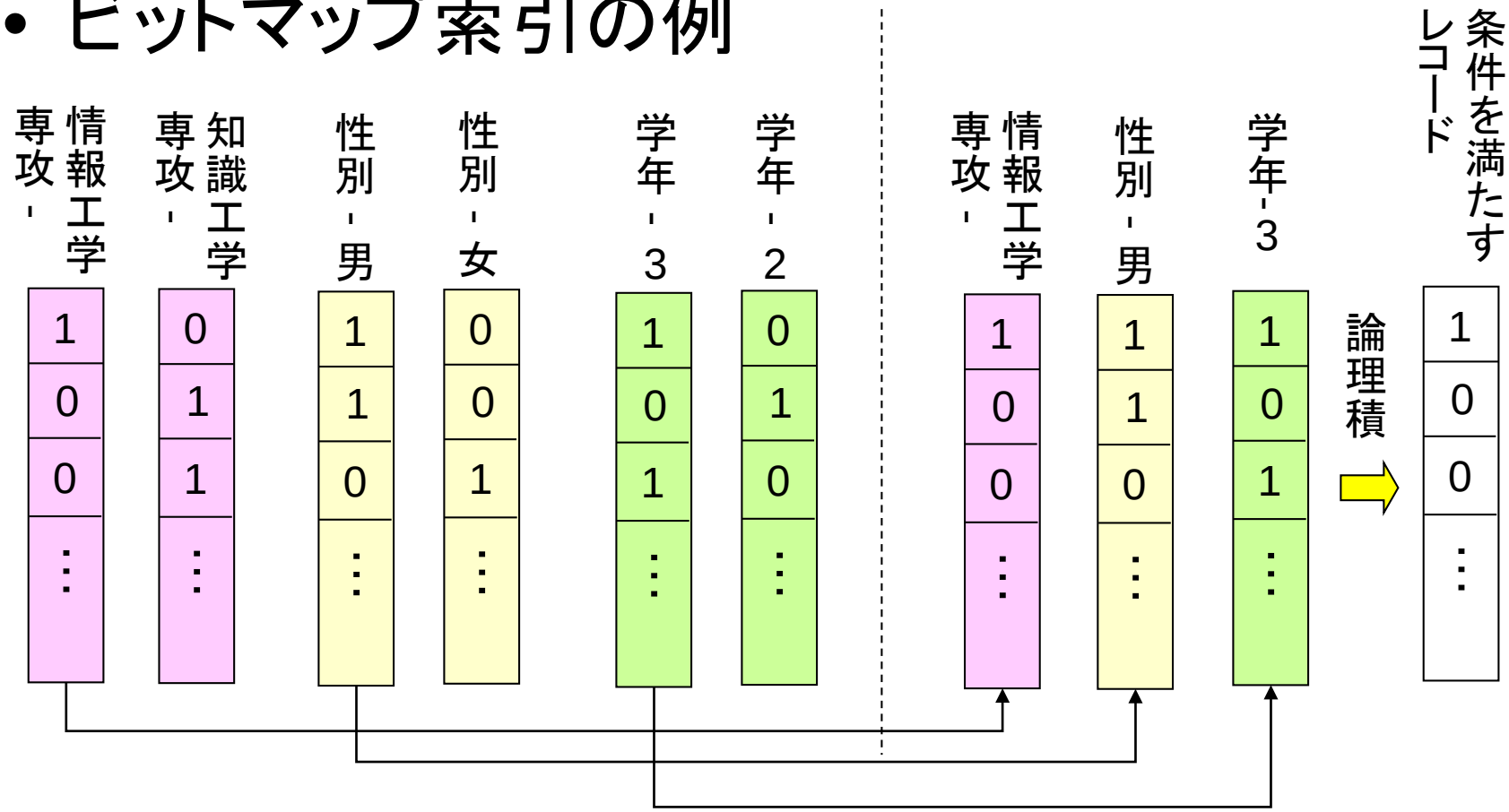
```
SELECT 氏名
FROM　学生
WHERE　専攻 = ’情報工学’ AND 性別 = ‘男’ AND 学年 = 3
```

# 補足2:ビットマップ索引(2)

- 問題点:$B^+$木は有効でない
  - 「性別」に二次索引を付与しても,「性別 = '男'」という条件で絞込みが利かない
- ビットマップ索引(bitmap index)
  - 一定の順番(例:レコード識別子の順番)でビットを配置
  - 各属性値ごとに作成:その属性値をとるとき値を1とする
    - ある属性値をとるレコードを効率よく列挙できる
  - コンパクトな表現:圧縮がしばしば利用される
  - 問合せ処理では論理演算(AND, OR, NOT)を利用
  - 最近のDBMSの多くが実装

# 補足2:ビットマップ索引(3)

- ビットマップ索引の例




条件:「専攻 = '情報工学' AND 性別 = '男' AND 学年 = 3」

# 補足3：現在のストレージ技術

- RAID（Redundant Arrays of Inexpensive Disks）
  - 複数のディスクを組み合わせ，冗長性を向上させ，信頼性・可用性を向上させる
  - どのような冗長化をするかで，さまざまな種類がある（RAID 0, RAID 1，…など）
- SAN (Storage Area Network)
  - ストレージ（記憶装置）どうしをつなぐ，ストレージ専用のネットワークを構成
  - サーバ機器をSANに接続することで，大規模データに対する高速アクセスが可能
- その他に，NAS (Network Attached Storage) などの技術もある