データベースシステム
データ格納方式(3) B*-tree

2016年5月17日

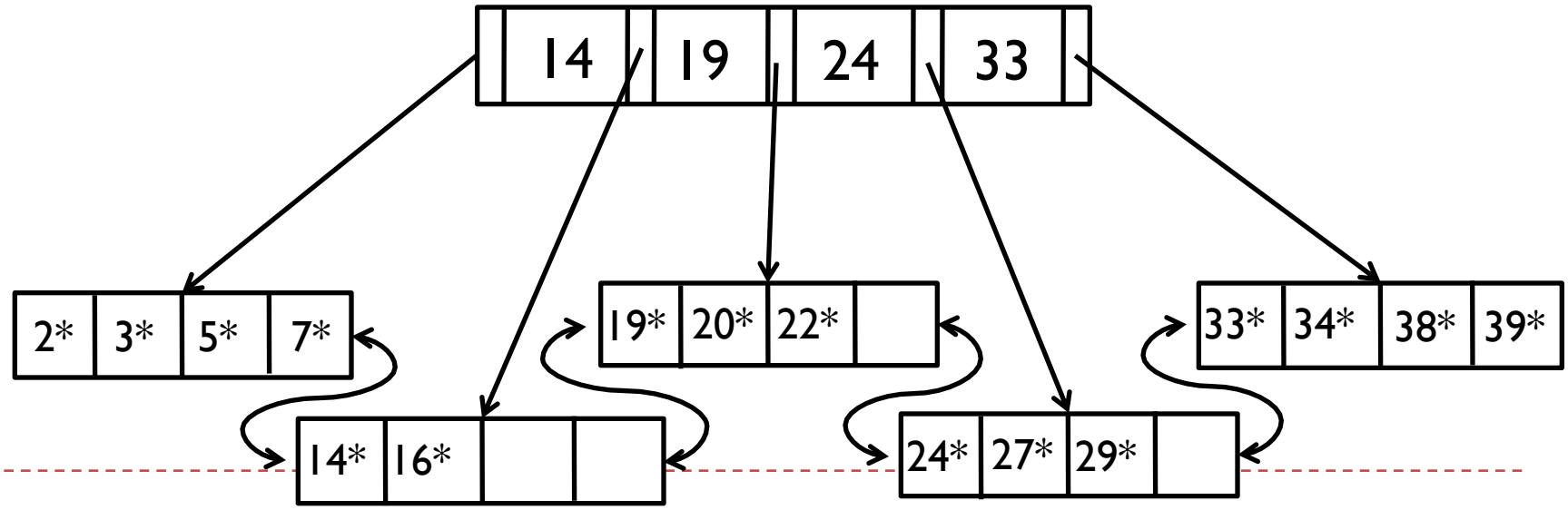
復習：各格納方式（索引）の向き不向き

	Heap file	Hash File
スキャン	N	$\frac{6}{5}N$
範囲問合せ	N	$\frac{6}{5}N$
完全一致	N	1+α
挿入	2	2+α
削除	N/2+1	2+α
更新	N/2+1	2+α

どちらも範囲問合せは得意ではない

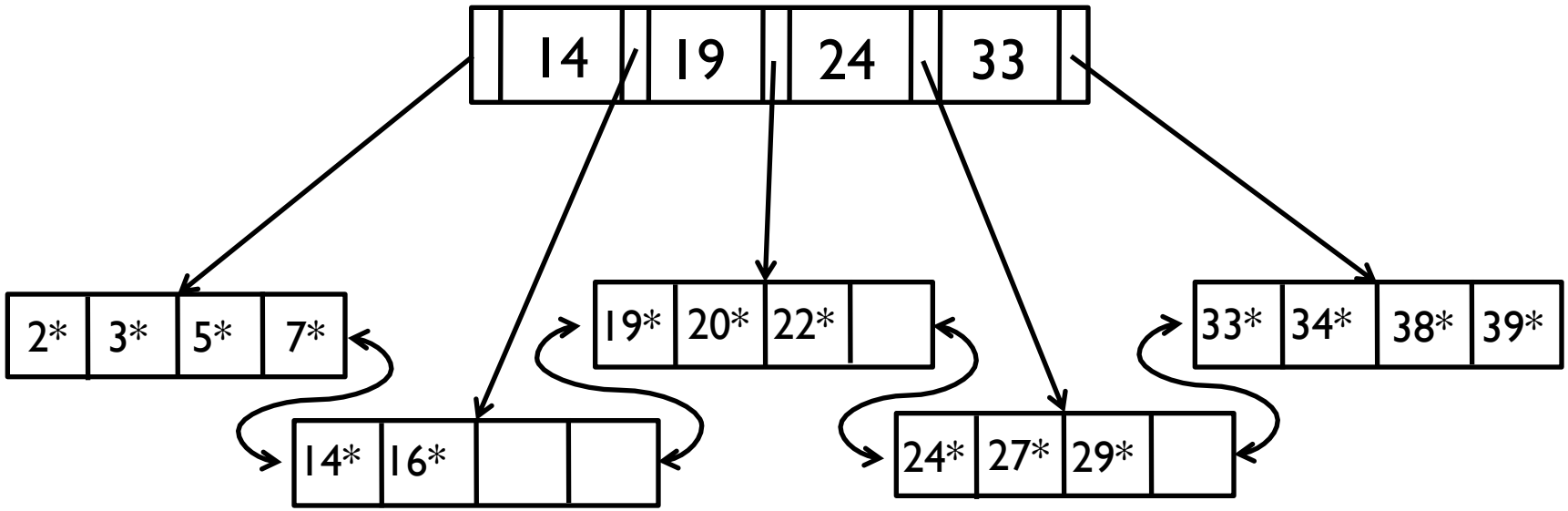
B*-TREE(Balanced Tree)

- レコードの追加や削除があるたびに動的に木構造索引を構成していく動的索引
- 木構造の高さは均一（Balanced)
- B*-treeのパラメータ：Order d
  - ノードに含まれるエントリ数m は d<=m<=2d
- 検索コスト＝木の高さ(=O(log(n))



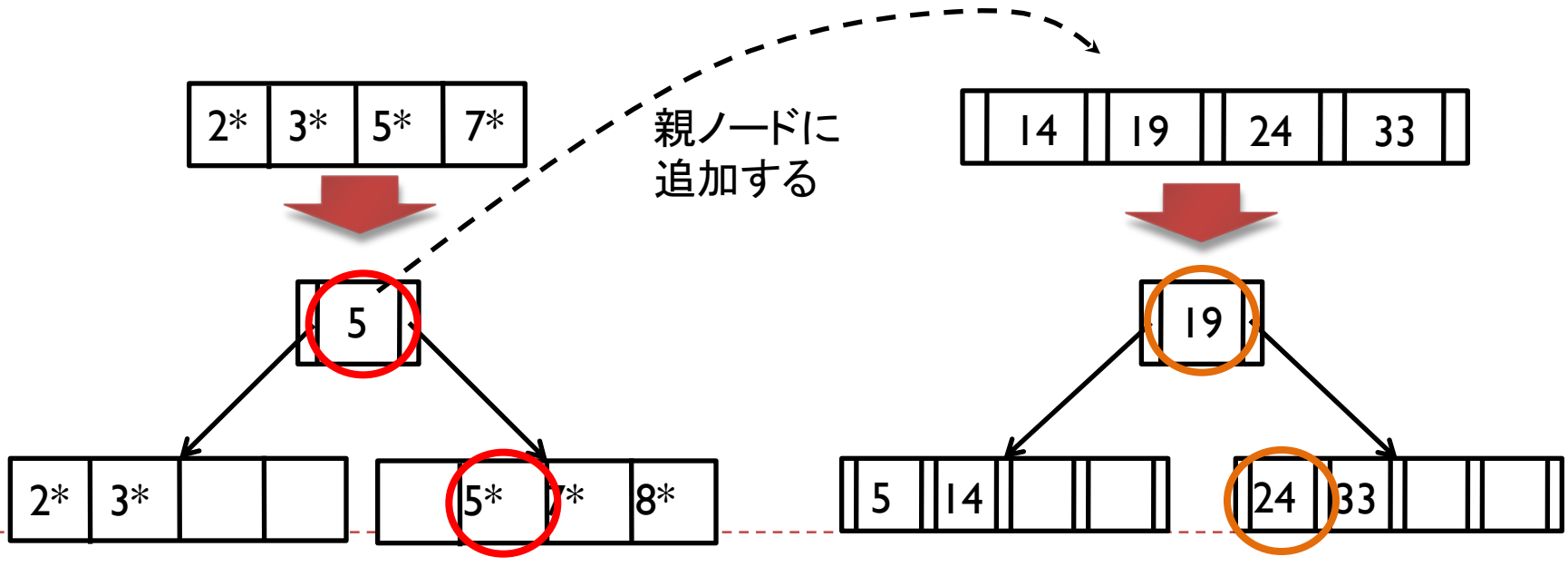
B*-TREEアルゴリズム（挿入）

- 以下の木に8*を追加することを考えよう
  - 一番左のノードは満杯な状態
  - →アルゴリズムを参照

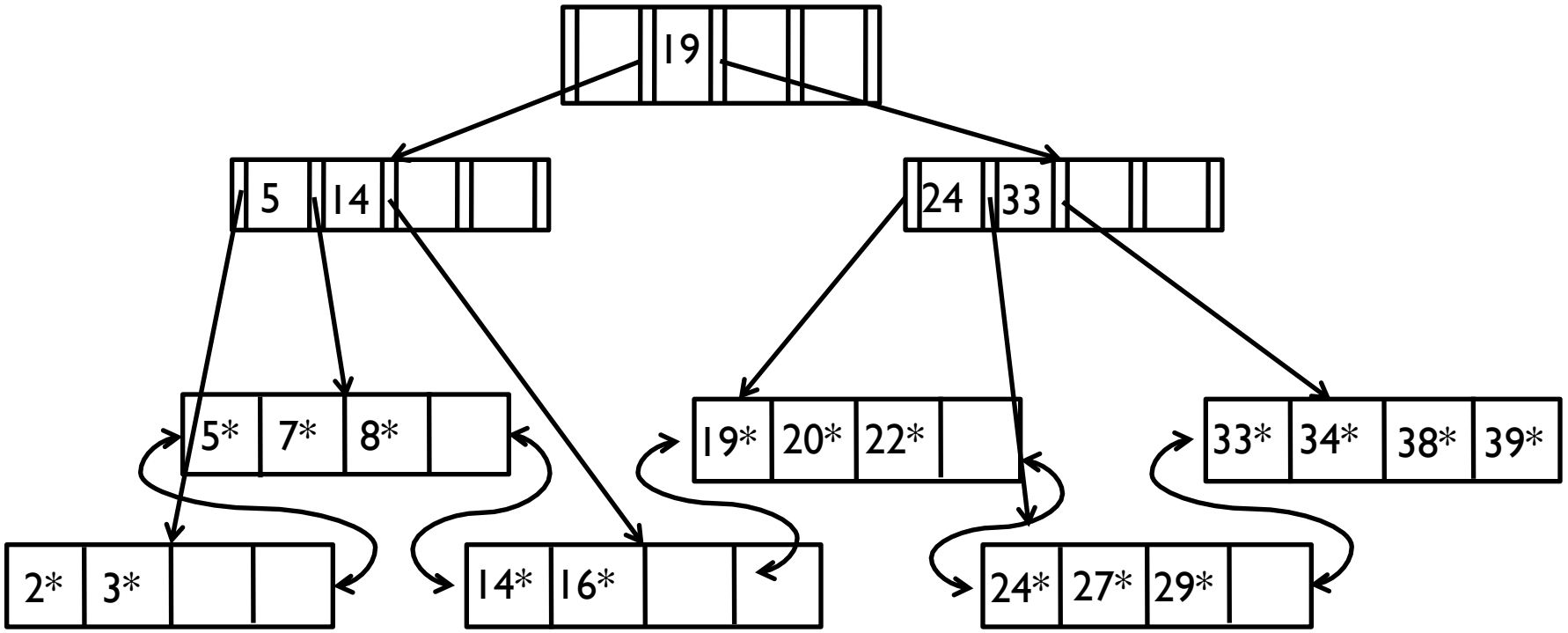


B*-TREEアルゴリズム（挿入）

- 以下の木に8*を追加することを考えよう
  - 一番左のノードは満杯な状態
- ノードを二つに分割し、初めのd個のデータエントリを左のノードに、残りを右のノードに配置する。

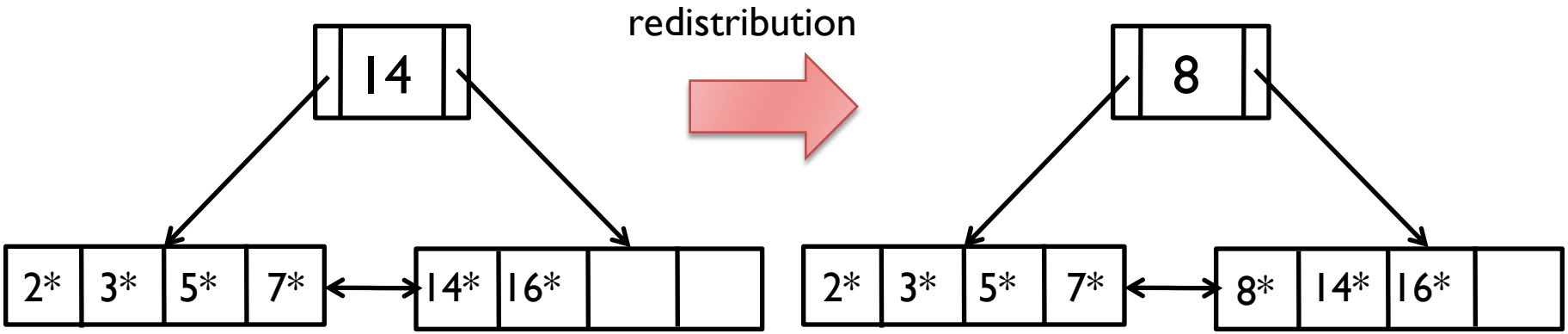


8*が挿入された後の木



B*-TREEアルゴリズム（挿入,再配分)

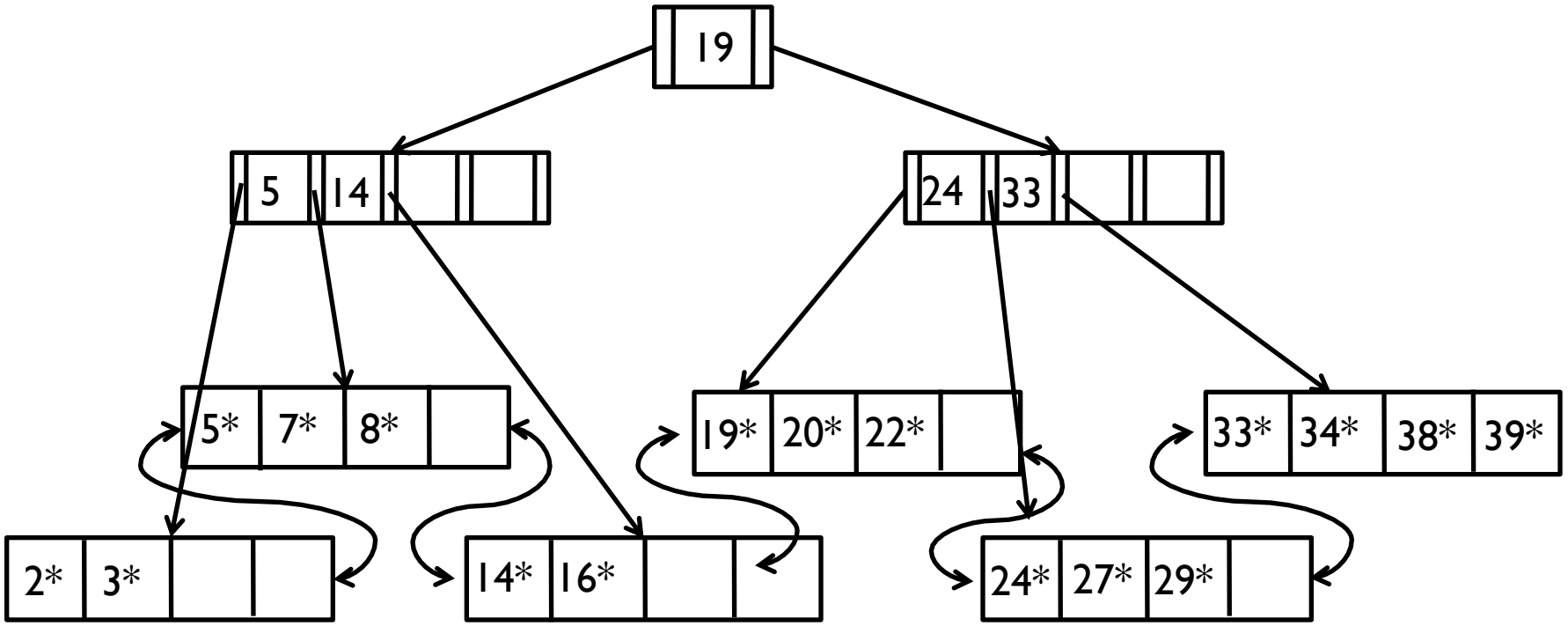
- 各ノードに半分はエントリが埋められている→索引ファイル内のデータ占有率：67%（効率が良い）
- ただし、兄弟ノードとのエントリ数に偏りがある場合は、エントリを馴らす(再配分)ことで効率よくデータを追加することができる



- 兄弟ノードに1以上の空きがある

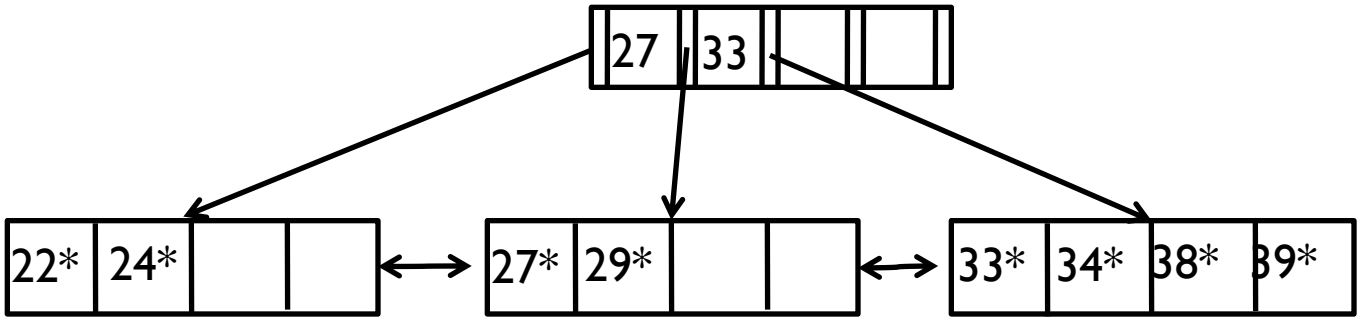
B*-TREEアルゴリズム（削除）

- 19*,20*を削除することを考えよう
- →ノード内のエントリ数がd=2を下回る

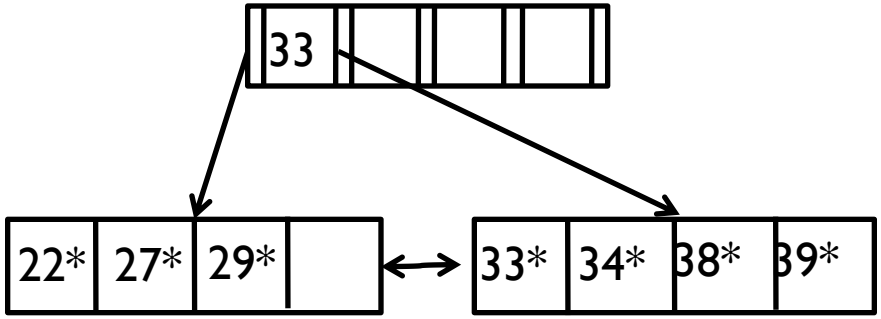


B*-TREEアルゴリズム（削除）

- 兄弟ノードとの間で再配分を試みる



- さらに24*を削除することを考える
  - →再配分も出来ない（エントリの合計数が2dを下回る）
  - →兄弟ノードを結合してしまう

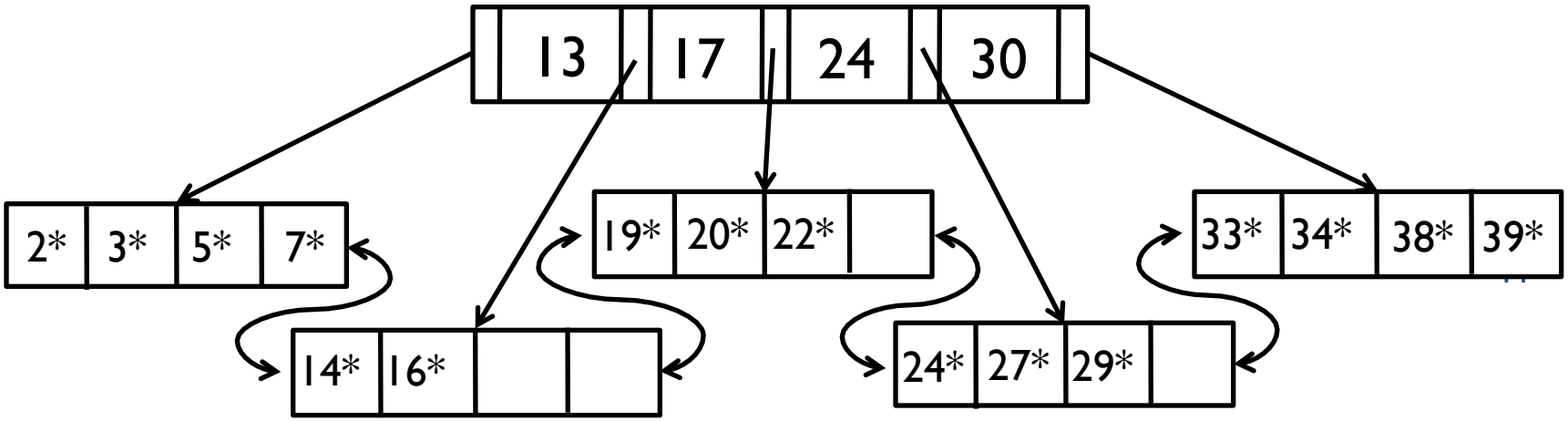


演習

- 1.以下のデータエントリが順番に挿入された時のB+-tree(d=1)を書きましょう
  - 20*,7*,13*,2*,10*,8*
- 9ページのB*-treeから3*, 5*,7*を削除した時のB+-treeを書きましょう

索引におけるノード内のエントリ数

- 例では1ノード内のエントリ数を4とした
  - 実際にはページに入りきるだけの数を用意
  - B+-treeではノード内の2/3はデータが詰まっている
  - 例）1ページ=4KB, 1レコード=100byte, 索引を作る属性値=10byte

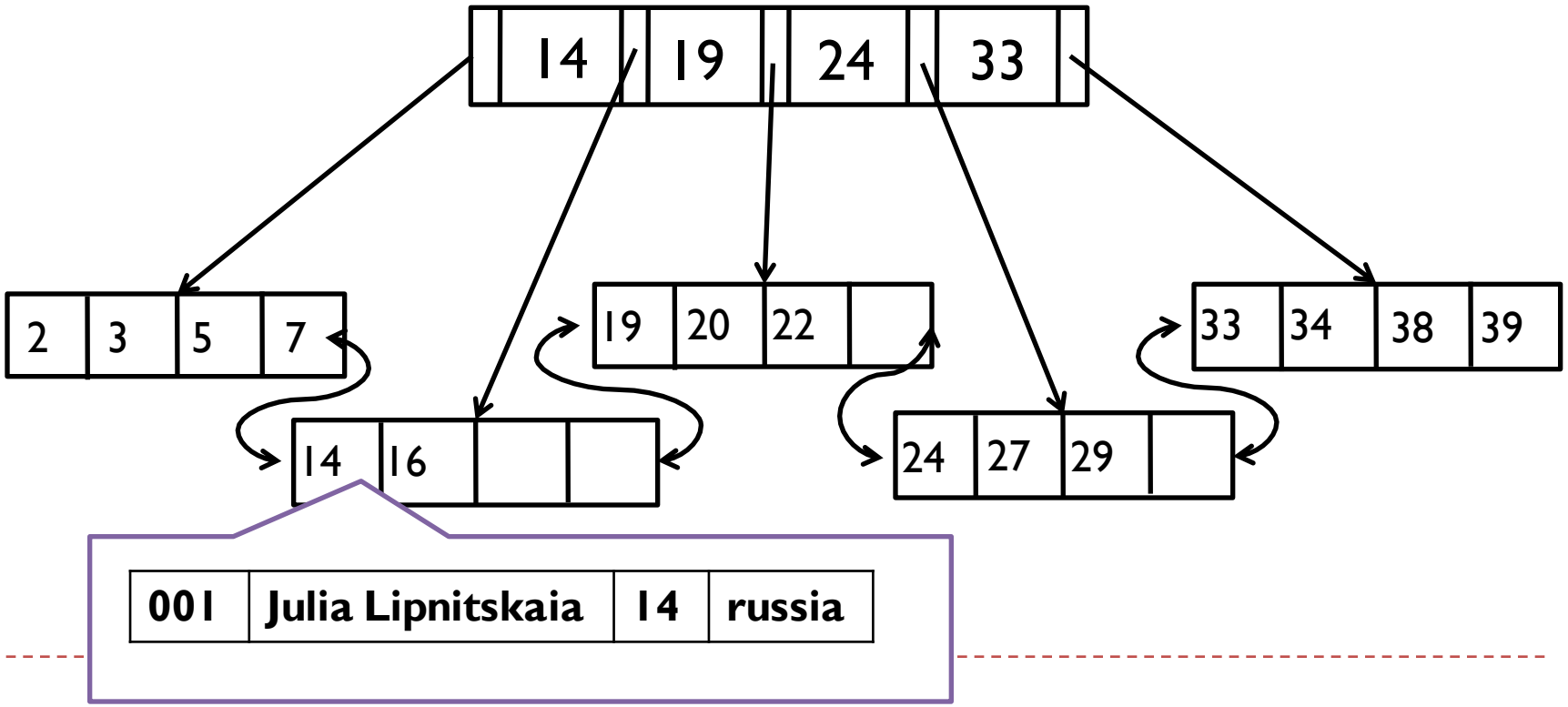


コストを計算しよう

	B*-tree (一次)	B*-tree (二次)
スキャン		
範囲問合せ		
完全一致		
挿入		
削除		
更新		

B*-tree (一次索引)

- リーフノードにはレコードが格納されている
  - 例）以下のテーブルに対してageで一次索引を作った場合
    skaters(id,name, age, country)

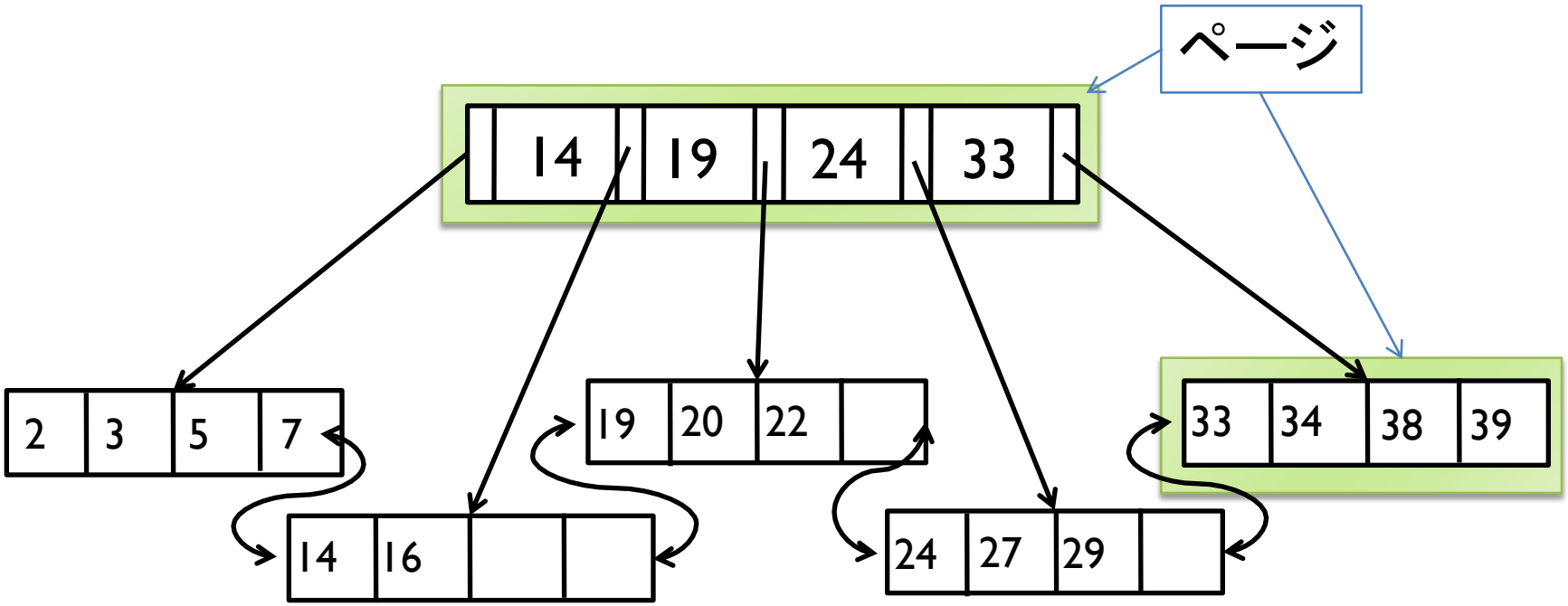


演習2：必要なノード数を計算しよう

問題：
ヒープファイルに詰め込んだときにNページ必要だったデータを、B*-treeの一次索引に入れなおしたとき、リーフノードは何ページ必要になるでしょうか？

演習2の続き：前提となる条件

- 全てのノードはページである
- リーフノードには全体の2/3の割合でデータが格納されているとする

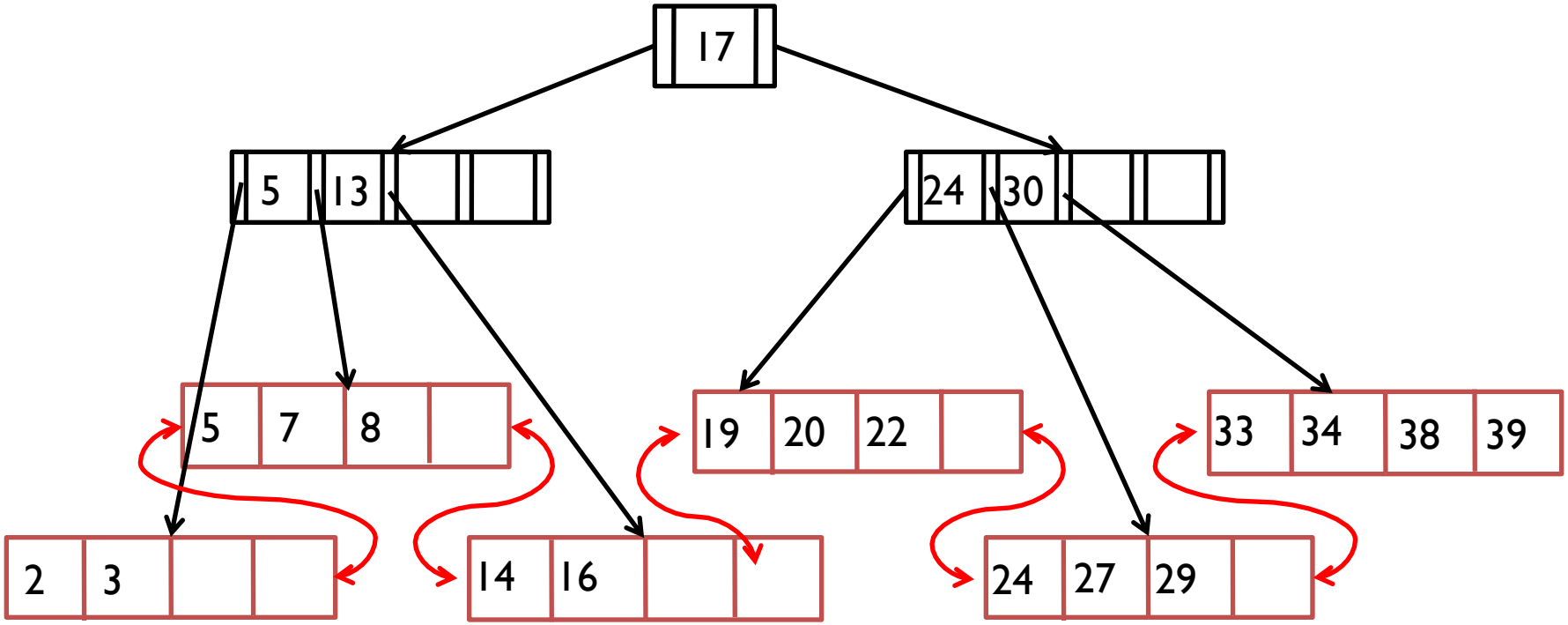


演習3：コストを計算しよう

	B*-tree (一次)	B*-tree (二次)
スキャン		
範囲問合せ		
完全一致		
挿入		
削除		
更新		

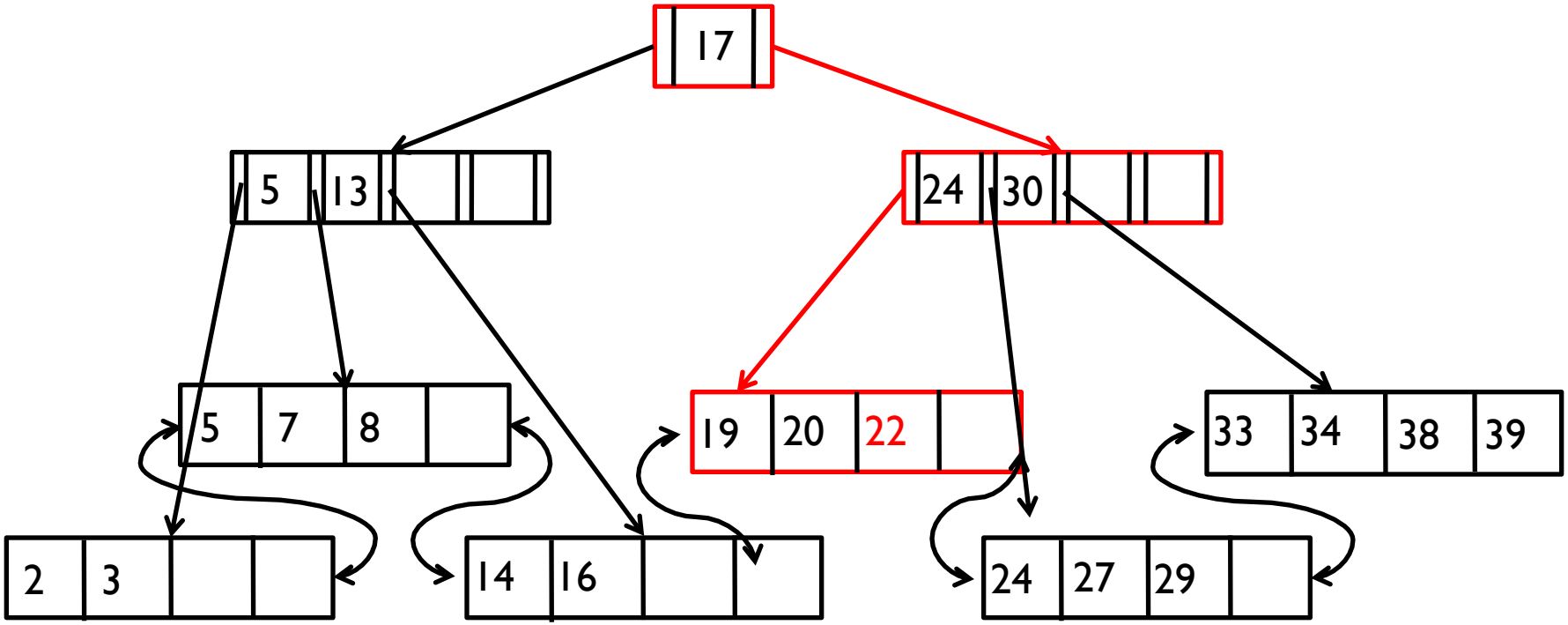
B*-tree(一次)：スキャンのための処理

- リーフノードを左から順番にたどる
- コスト…リーフノード数



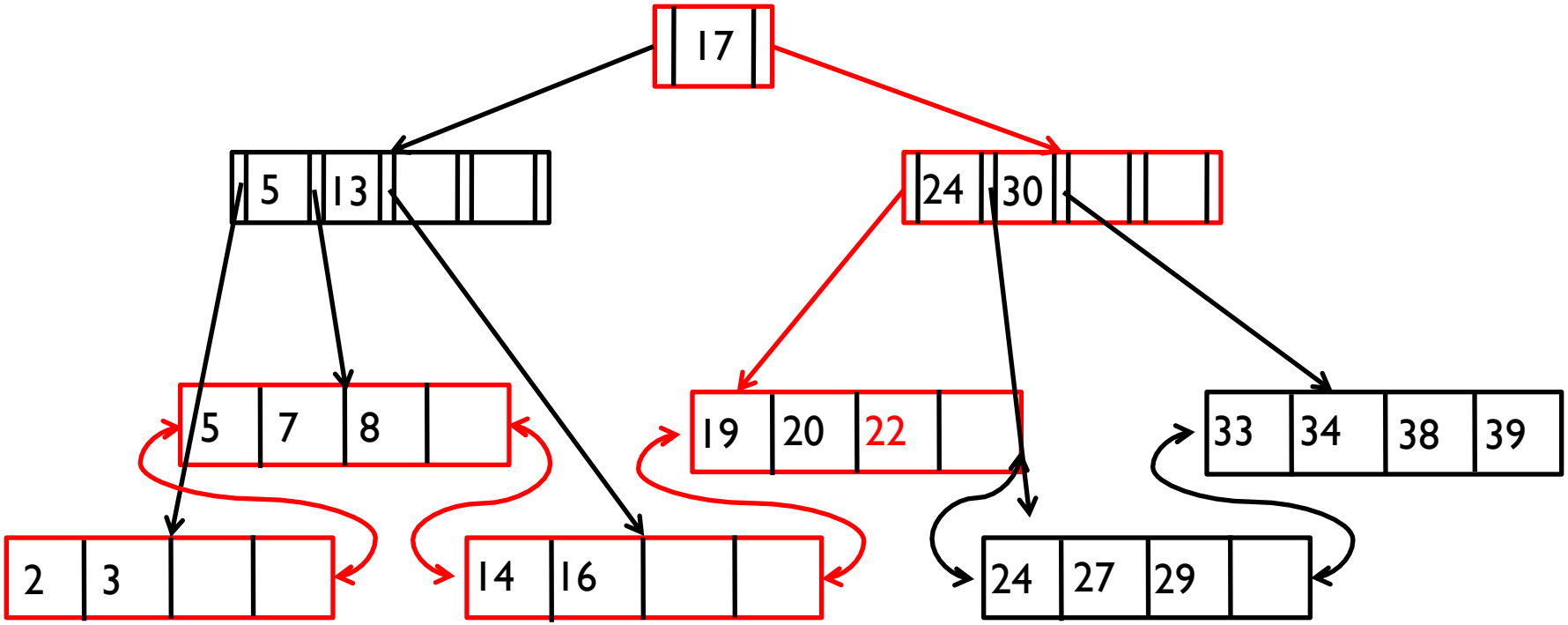
B*-tree(一次)：完全一致問合せのための処理

- ルートノードから順番にたどる
  - 例) age = 22のレコードを探す
- コスト：$\log_{\text{ノード内のエントリ数}}$リーフノード数



B*-tree(一次)：範囲問合せのための処理

- ルートノードから順番にたどり、リーフノードをたどる
  - 例) age <= 22のレコードを探す
- コスト：$\log_{\text{ノード内のエントリ数}}$リーフノード数 ＋ α



B*-tree(一次)：変更のための処理

挿入

- 1つのレコードを挿入する
- 挿入するリーフノードを検索するコスト(完全一致)＋書き込み

削除

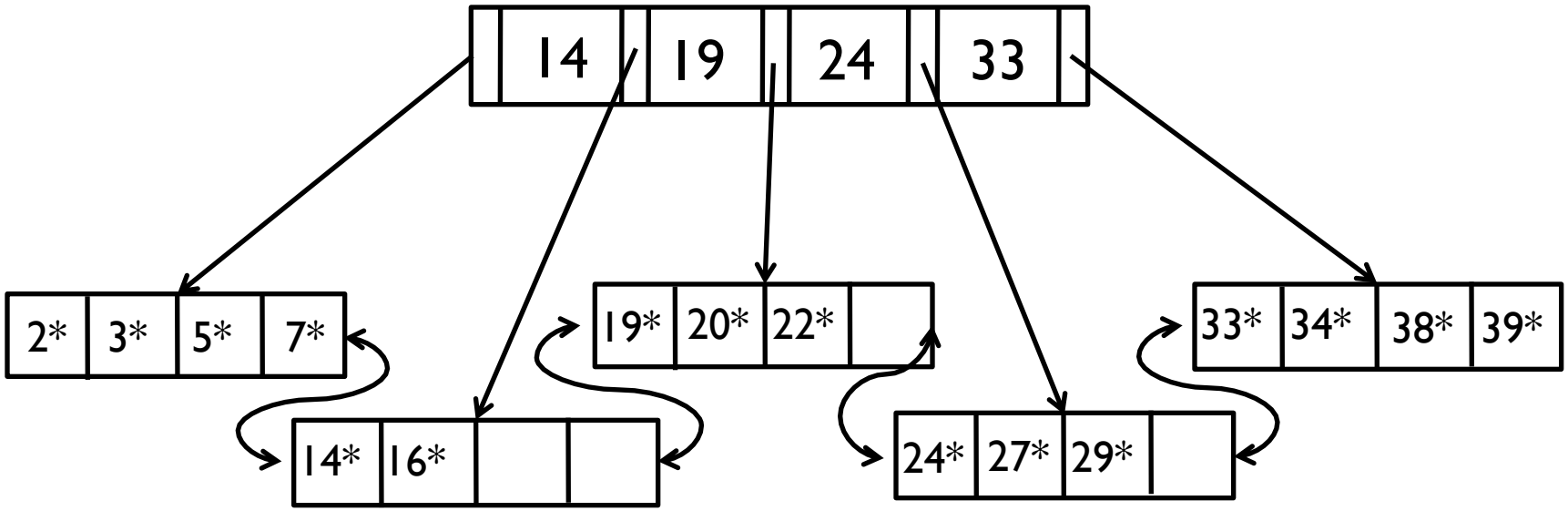
- 1つのレコードを削除する
- 削除するレコードのあるリーフノードを検索するコスト(完全一致)＋書き込み

更新

- 1つのレコードの属性値を変更する
- 更新するレコードのあるリーフノードを検索するコスト(完全一致)＋書き込み

B*-tree (二次索引)

- リーフノードにはレコードへのポインタが格納されている
- 実際のレコードは一次索引に格納されている
  - 基本的には一次索引はヒープファイルと考える
  - 例）以下のテーブルに対してageで二次索引を作った場合
    skaters(id,name, age, country)

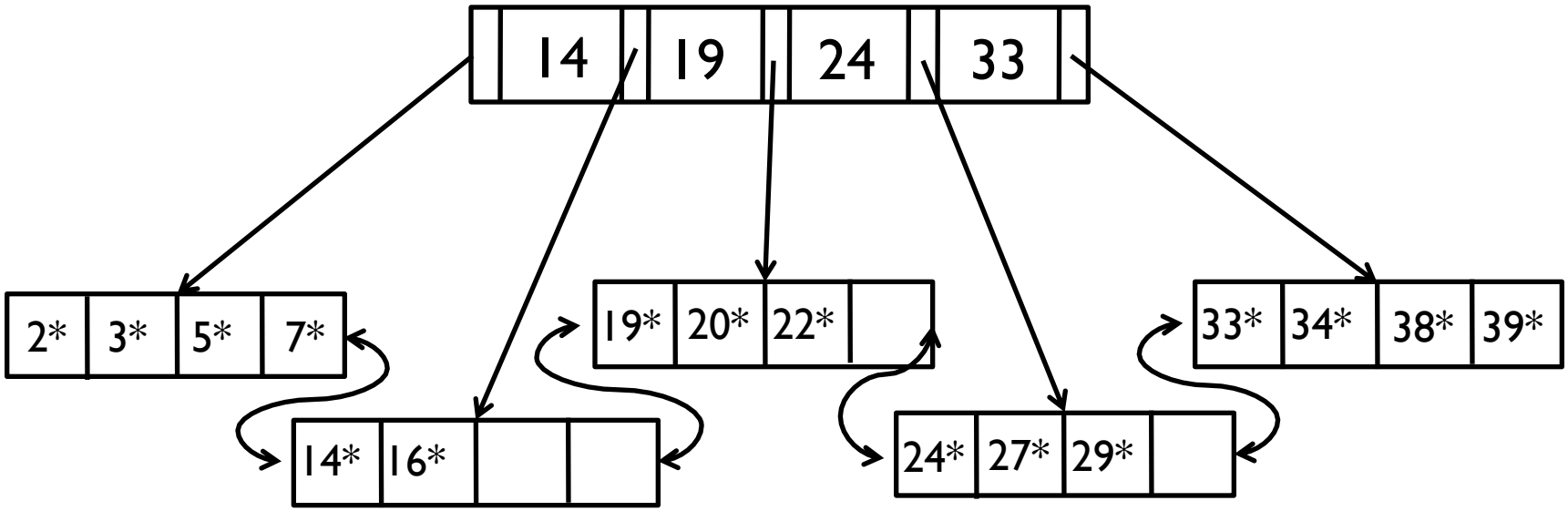


演習4：必要なノード数を計算しよう

問題：
ヒープファイルに詰め込んだときにNページ必要だったデータからB*-treeの二次索引を作成するとき、二次索引のリーフノードは何ページ必要になるでしょうか？

演習３の続き：前提となる条件

- ポインタの大きさはレコードの大きさの10分の1とする
- リーフノードには全体の2/3の割合でデータが格納されているとする

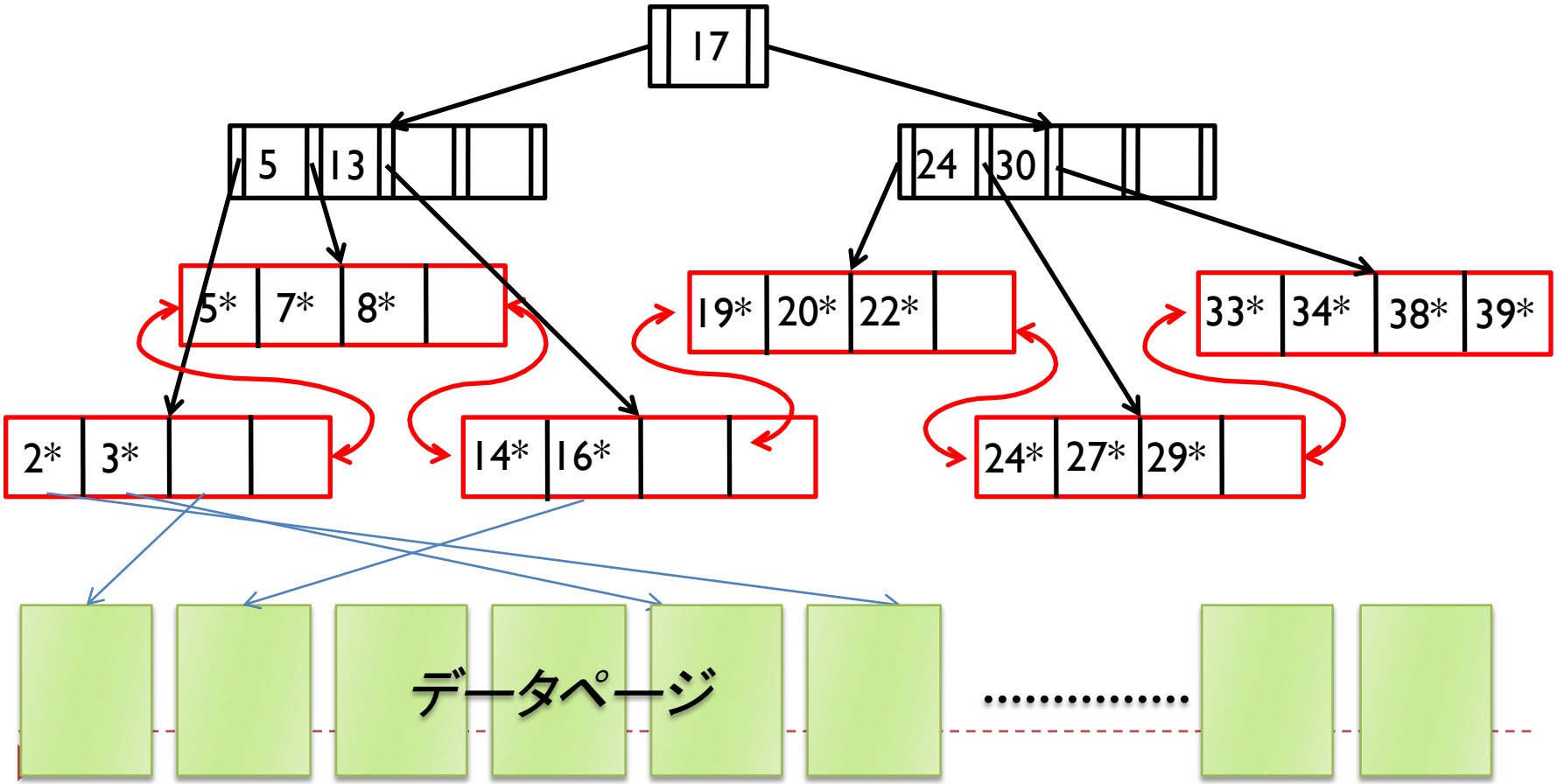


演習5：コストを計算しよう

	B*-tree (一次)	B*-tree (二次)
スキャン		
範囲問合せ		
完全一致		
挿入		
削除		
更新		

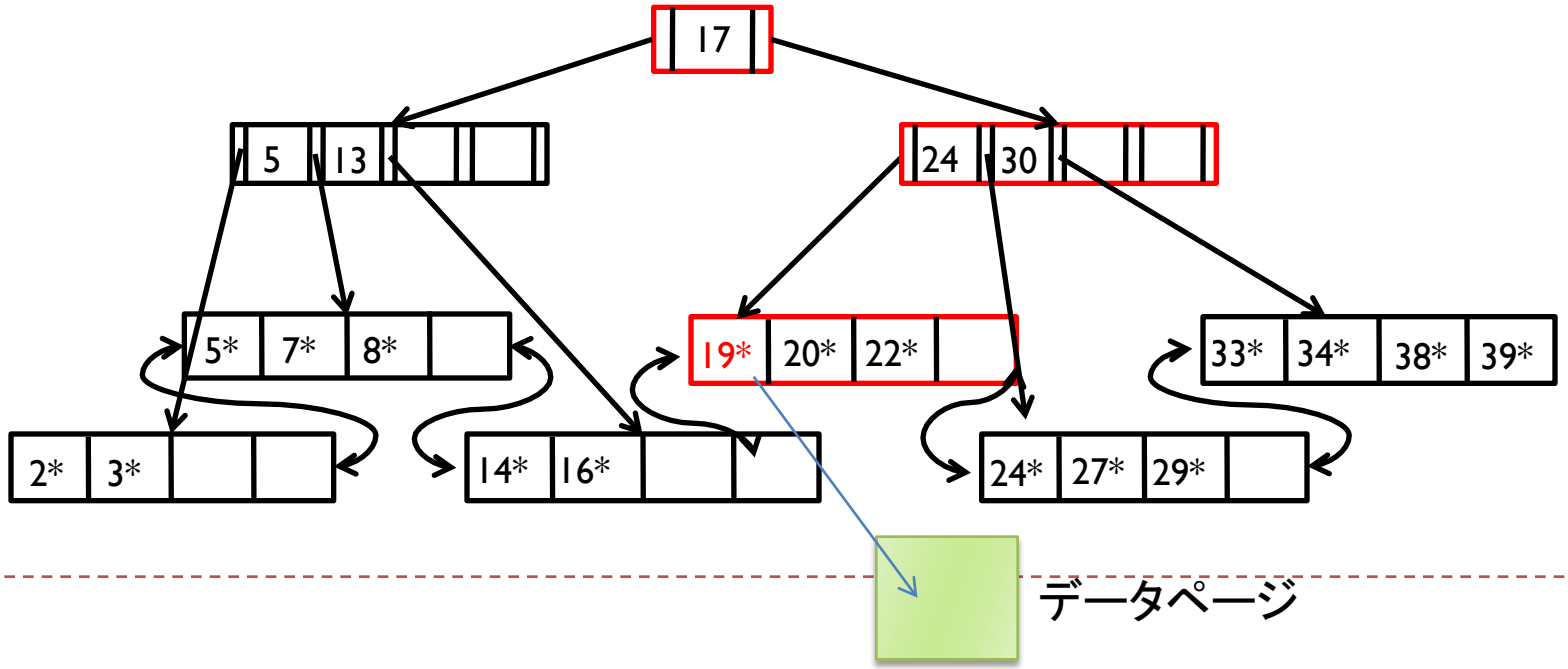
B*-tree(二次)：スキャンのための処理

- リーフノードをたどりながらポインタ先のページを読み込む
  - 最悪の場合（全部のレコードが別のページに入ってる）を考えよう
  - ヒープファイルの1ページに入っているレコード数をRとする



B*-tree(二次):完全一致問合せ

- ルートノードからたどり，リーフノードで該当するレコードにアクセスする
- 該当するレコード数をrとする
- コスト： $\log_{\text{ノード内のエントリ数}}$リーフノード数+r



B*-tree(二次):範囲問合せ

- リーフノード内のデータエントリはすべて異なるデータページを指しているとする
- 横にたどるリーフノード数をαとする
  - 下の図の場合、α=3