アルゴリズムとデータ構造

# 探索 (Search)

情報学研究科 知能情報学専攻
音声メディア分野
吉井 和佳
yoshii@kuis.kyoto-u.ac.jp

# アルゴリズム



- 問題を解くための手順を定めたもの
  - データに対して、どのような操作を、どのような順序で行うかを、曖昧な点の残らないようにきちんと定めたもの

ユークリッドの互除法

例：２つの正整数 $m, n$ の最大公約数を求めるアルゴリズム

順序

ステップ１：$m$ を $n$ で割ったあまりを $r$ とする

ステップ２：$r = 0$ であれば終了 (このとき $n$ が最大公約数)

ステップ３：$m \leftarrow n, n \leftarrow r$ としてステップ1に戻る

操作

プログラム：アルゴリズムを計算機上で動作させるための表現

プログラムとしての表現の仕方はさまざまだが・・・

計算時間への影響は アルゴリズムの改良 ≫ プログラムの改良

# アルゴリズムの正しさ



- 正しいアルゴリズムの条件
  - どんな入力データに対しても、間違った答えを与えないこと
    - 解を与える場合にはその解が正しいこと
  - どんな入力データに対しても、有限の時間で解を与えること
    - 計算を完了すること

ユークリッドの互除法

どんな $m, n$ に対しても正しい解が得られる証明が可能 (省略)

例：２つの正整数 $m, n$ の最大公約数を求めるアルゴリズム

ステップ１：$m$ を $n$ で割ったあまりを $r$ とする

ステップ２：$r = 0$ であれば終了 (このとき $n$ が最大公約数)

ステップ３：$m \leftarrow n, n \leftarrow r$ としてステップ1に戻る

反復するたびに $n$ は減少
→ 最悪でも $n$ 回繰り返せば終了

# アルゴリズムの計算量



- アルゴリズムの良し悪しを評価するための基準
  - 時間計算量 (time complexity)
    - プログラムの中でそれぞれの文が実行される回数
  - 空間計算量 (space complexity)
    - プログラム実行時に保存しておくべき変数の最大個数

一般に「時間計算量」と「空間計算量」の間にはトレードオフが存在

プログラム実行前にあらかじめ計算結果を書きだしておけば
時間計算量は削減されるが空間計算量は増大する

多くの場合アルゴリズムの「時間計算量」が重要

入力データのサイズ $n$ に対して計算量はどうなるか → $O(n)$記法

# $O(n)$ 記法



- データサイズ $n \to \infty$ のときの漸近的な時間計算量
  - 関数 $f(n)$ に対して、ある一定の値 $n_0$ と正の定数 $c$ があり、$n \geq n_0$ を満たす全ての $n$ に対して $f(n) \geq c\ g(n)$ となるとき
    $$f(n) = O(g(n))$$
  - 計算量の和
    - $O(2n) + O(3n) = O(n)$
    - $O(1) + O(n) = O(n)$
    - $O(n^2) + O(n) = O(n^2)$
    - $O(n) + O(n \log n) = O(n \log n)$
    - $O(n^2) + O(n \log n) = O(n^2)$
    - $O(n^2) + O(1000n \log n) = O(n^2)$

係数は無視して考える

どの部分が支配的であるかに着目

| | $n = 1$ | $n = 10$ | $n = 100$ |
|---|---|---|---|
| $O(\log n)$ | 1 | 1.5 | 2 |
| $O(n)$ | 1 | 10 | 100 |
| $O(n \log n)$ | 1 | 15 | 200 |
| $O(n^2)$ | 1 | 100 | 10,000 |

# 各種の計算量



- 最悪計算量 (worst case time complexity)
  - アルゴリズムにとって最も具合の悪いデータを与えた時の計算量
  - 評価方法 (一般に簡単)
    - 最悪のデータとは何かを定める
    - その際の計算量を評価する
- 平均計算量 (average case time complexity)
  - 可能な入力データすべてに対する計算量の平均
  - 評価方法 (しばしば理論的に難しい)
    - 入力データの存在範囲とそれぞれの出現確率を定める
    - 出現確率のもとでの計算量の期待値を計算する

最悪データの出現が稀であるなら
平均計算量が小さいアルゴリズムは
実用上非常に有益

# データ構造



- プログラムを記述するうえでデータ構造の選択が重要
  - 不適切なデータ構造を用いるとアルゴリズム本来の計算量オーダを越えてしまう場合がある (オーダは同じでも実行時間がかかる)
  - 例：配列
    - 参照：番号(添え字)さえ分かれば $O(1)$
    - 探索：前から順番にたどる必要があるから $O(n)$
    - 挿入：$O(1)$　削除：データのシフトが必要なので $O(n)$
  - 例：リスト
    - 参照：前から順番にたどる必要があるから $O(n)$
    - 探索：前から順番にたどる必要があるから $O(n)$
    - 挿入・削除：枝の付け替えだけでいいので $O(1)$

# 探索のアルゴリズム



- 探索とは？
  - 問題設定：指定したキーの値をもつレコードを選び出す
    - 入力：キー (key)
    - 出力：レコード (record)
  - 評価方法：時間計算量
    - 探索・挿入・削除がどの程度高速に行えるか
  - 前提条件：レコードのキーの値は全て異なるものとする
    - キーの重複を許す場合のアルゴリズムは容易に導出可能

用語の定義
レコード：蓄えられている個々のデータ
各レコードはキーを保持するフィールドをもつものとする

# 配列上の線形探索



- 探索：$O(n)$

最初から順番に走査

最悪計算量：探すべきキーが最後尾 $O(n)$

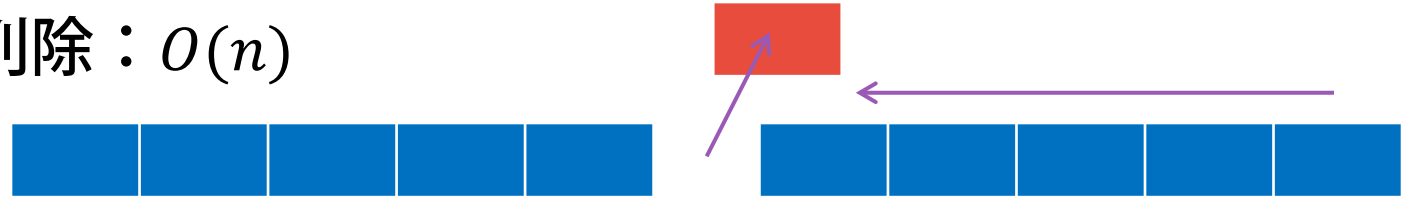
平均計算量：中央にある場合 $O\left(\frac{n}{2}\right) = O(n)$

- 挿入：$O(1)$

最後尾に新しいレコードを追加

キーの重複チェックが必要なら $O(n)$

- 削除：$O(n)$



後方のレコードを前方シフト

平均・最悪計算量ともに $O(n)$

# リスト上の線形探索



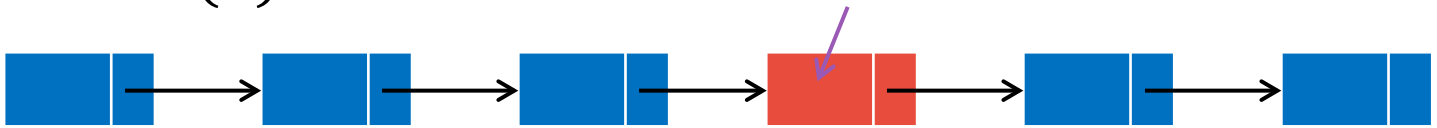
- 探索：$O(n)$

最初から順番に走査

最悪計算量：探すべきキーが最後尾 $O(n)$

平均計算量：中央にある場合 $O\left(\frac{n}{2}\right) = O(n)$

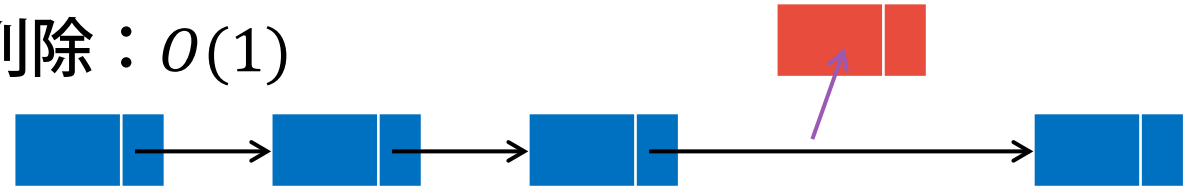
- 挿入：$O(1)$



リンクを付け替えるだけでOK

キーの重複チェックが必要なら $O(n)$

- 削除：$O(1)$



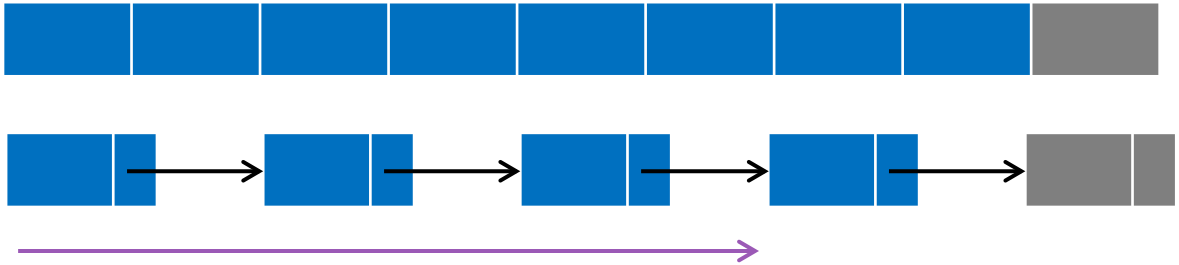
リンクを付け替えるだけでOK

キーの重複チェックが必要なら $O(n)$

# 実装上のテクニック



- 番兵 (sentinel)
  - データの最後尾に所望のキーをもつダミーレコードを付加しておく



最初から走査していくと必ずキーとマッチするレコードにぶつかる
→ 走査範囲チェック (インデクスとの比較) が不要

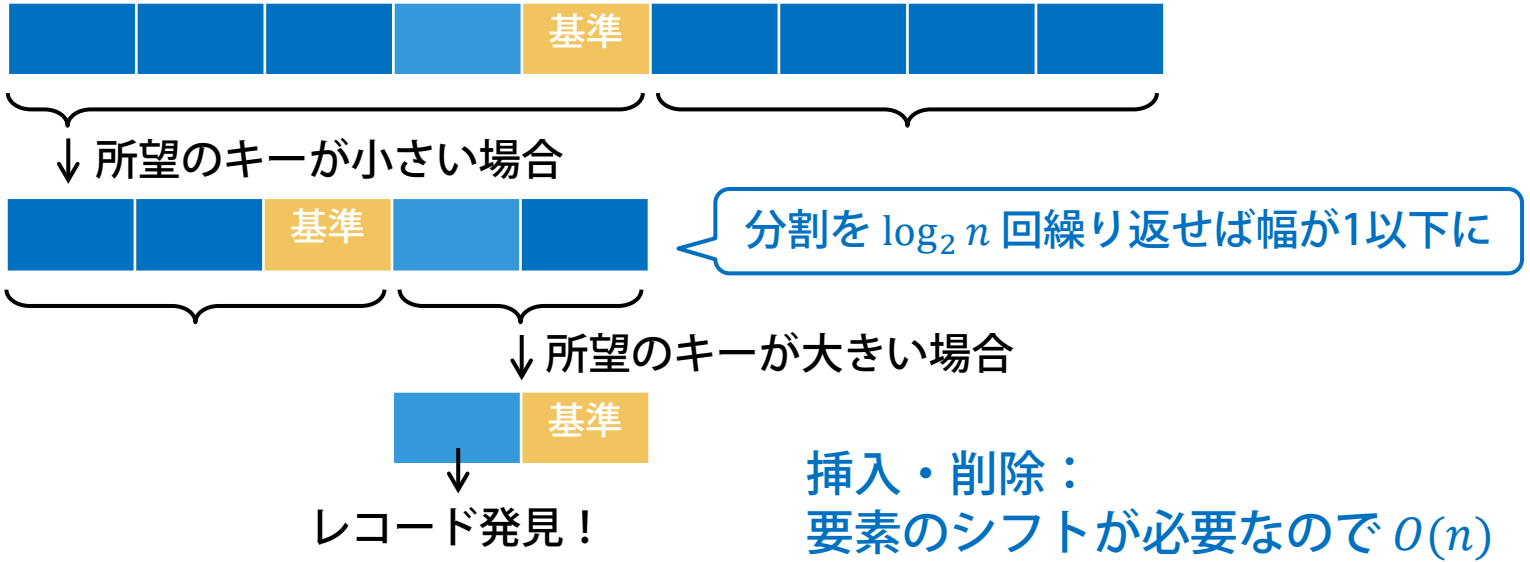
- 自己再構成リスト
  - 探索で見つけたレコードをリストの先頭に移動させる $O(n) + O(1)$
    - データアクセスの時間局所性に着目
    - 少し前にアクセスしたレコードは高速に再探索可能

# 二分探索



- 前提：キーの小さい順にレコードが並んでいる
  - 整列のアルゴリズム (次週解説) をあらかじめ適用しておく
- 探索：$O(\log n)$ 多くの問題で $O(n) \to O(\log n)$ とすることが極めて重要

配列を２分割しながら中央の要素とキーの大小を比較

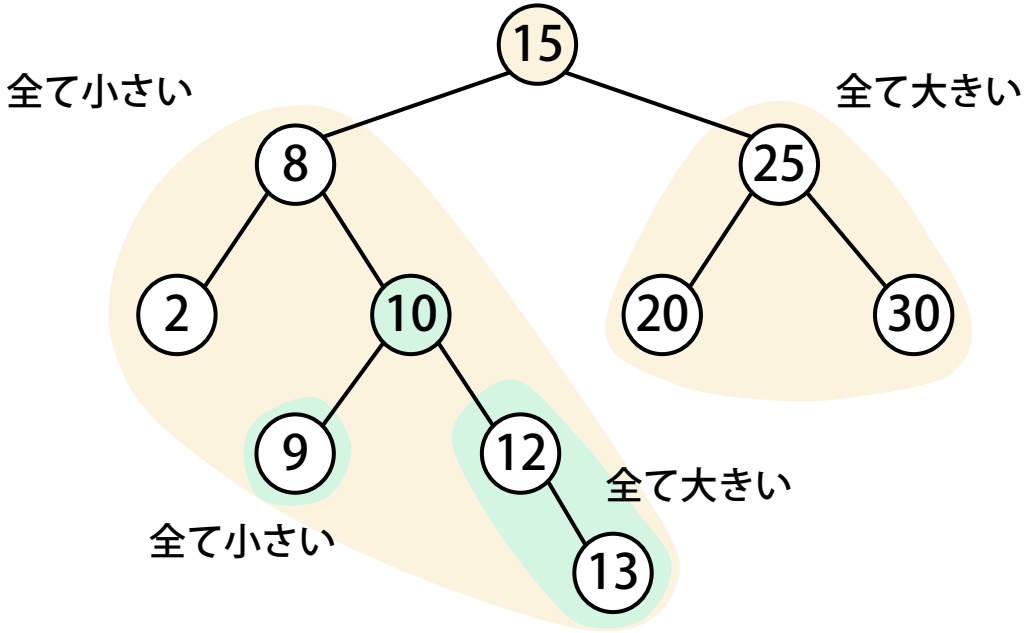



挿入・削除：
要素のシフトが必要なので $O(n)$

# 二分探索木



- 効率的な探索のために特殊な制約をもつ二分木
  - 動機：探索・挿入・削除の平均計算量を $O(\log n)$ 以下にしたい
  - 制約：ある頂点から見て、左側の子とその子孫すべてが小さい
    ある頂点から見て、右側の子とその子孫すべてが大きい




すべての頂点が制約を満たしていなければならない
(それ以外は自由)

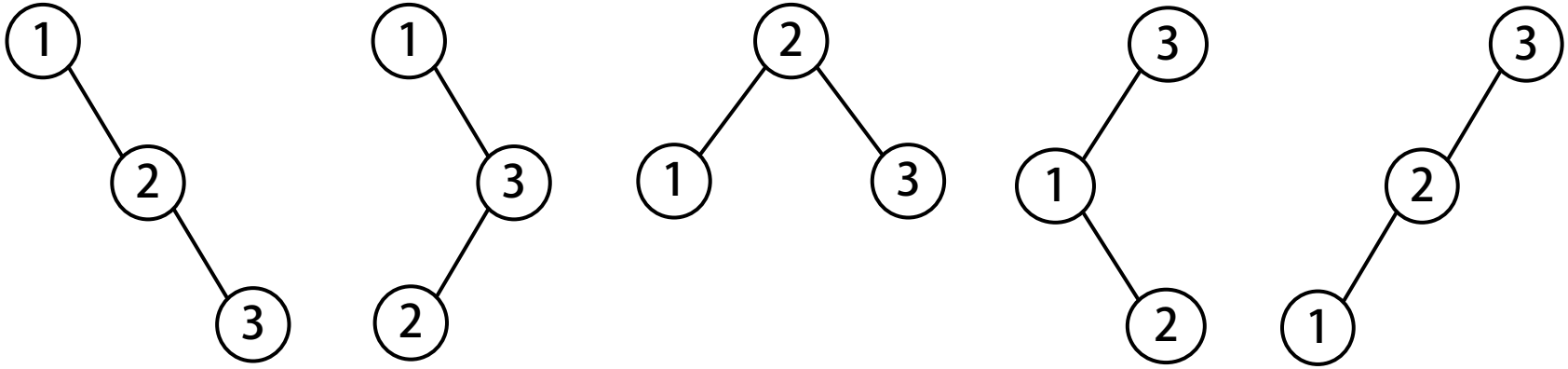
↓

同じ数の組み合わせでも
さまざまな木の構成法が存在

# 探索アルゴリズム



- 根から開始して、所望のキーと頂点とを比較して
  - 所望のキーの方が大きければ右側の子供を選択
  - 所望のキーの方が小さければ左側の子供を選択
  - 上記ステップを葉に下るまで繰り返し



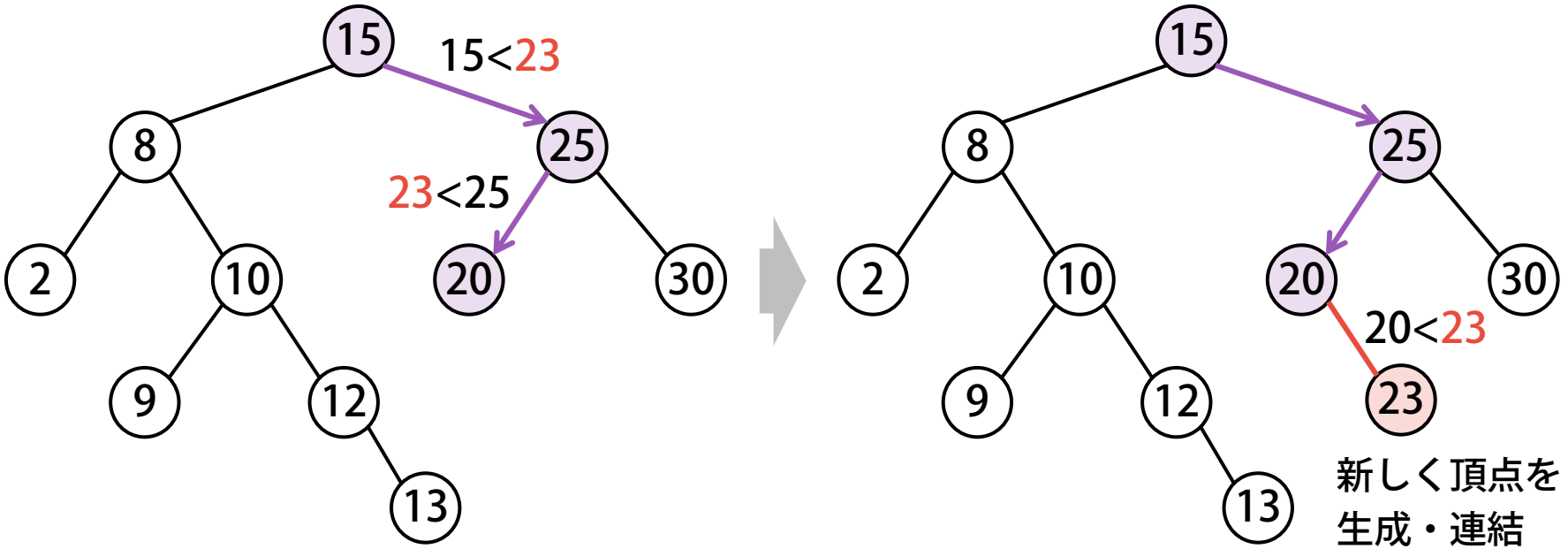

とりうる木の形によって計算量に違いが生じることに注意

根から葉まで近い木 (浅い木) の方が探索効率がいい

# 挿入アルゴリズム



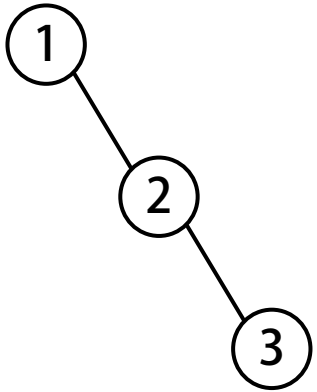
- 探索アルゴリズムを用いて挿入すべき位置を探す
  - 木を根から降りられるだけ降りる
  - 位置が決まったら新しく頂点を作成する



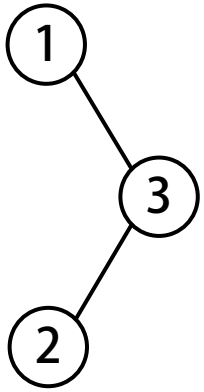
# データの挿入順序



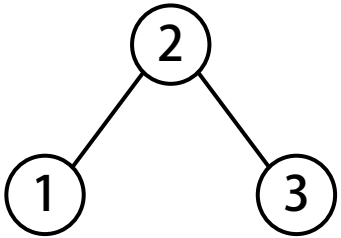
- データを挿入する順序を変えると得られる木の形も変化
  - ひとつずつデータを挿入していくとどうなるかに着目




1 → 2 → 3

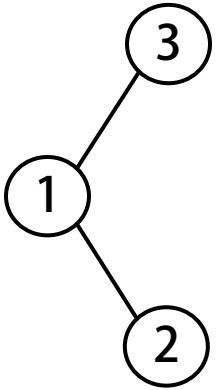



1 → 3 → 2

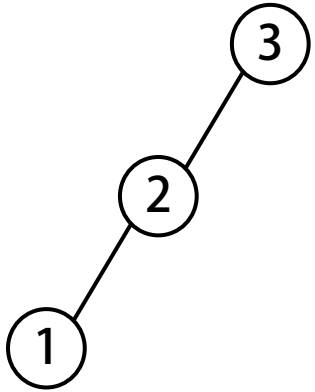



2 → 1 → 3

2 → 3 → 1




3 → 1 → 2




3 → 2 → 1

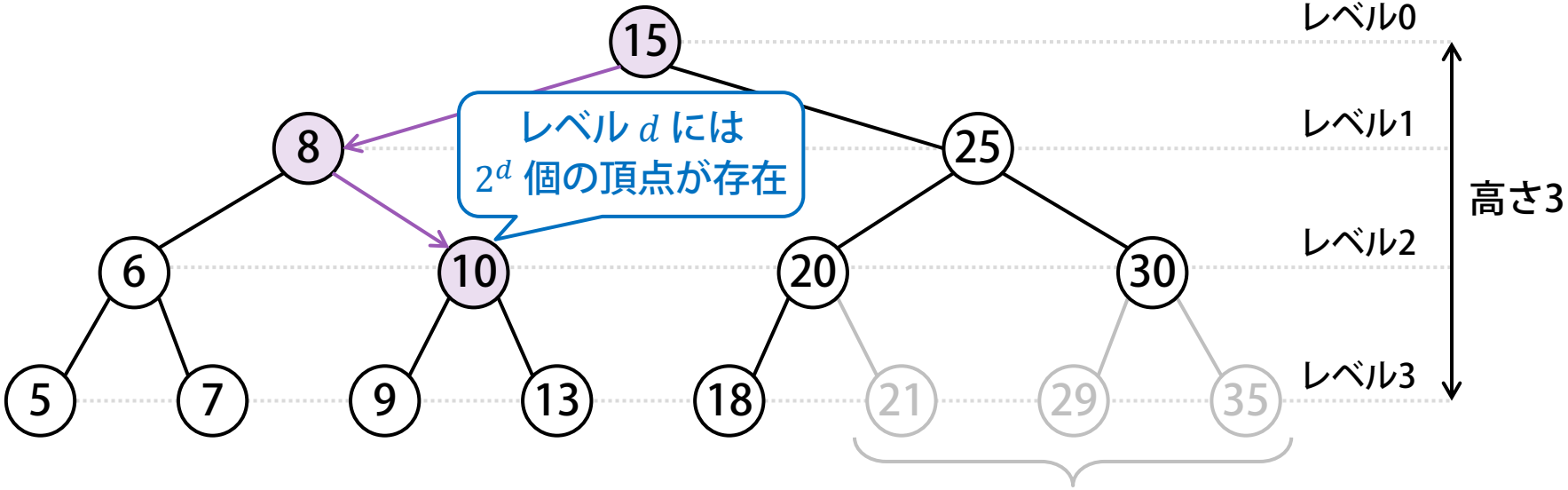
データ数を $n$ とすると考えられる順序は $n!$ 通り存在

探索に都合がいい木もあれば都合が悪い木もできる

# 最善の場合



- 完全二分木のとき探索の計算量が最小
  - 根から葉までの距離がどこでも等しくなっている木




木の深さ$d$に対して、要素数が$2^{d+1}-1$に不足するなら適当な数だけ葉を除く

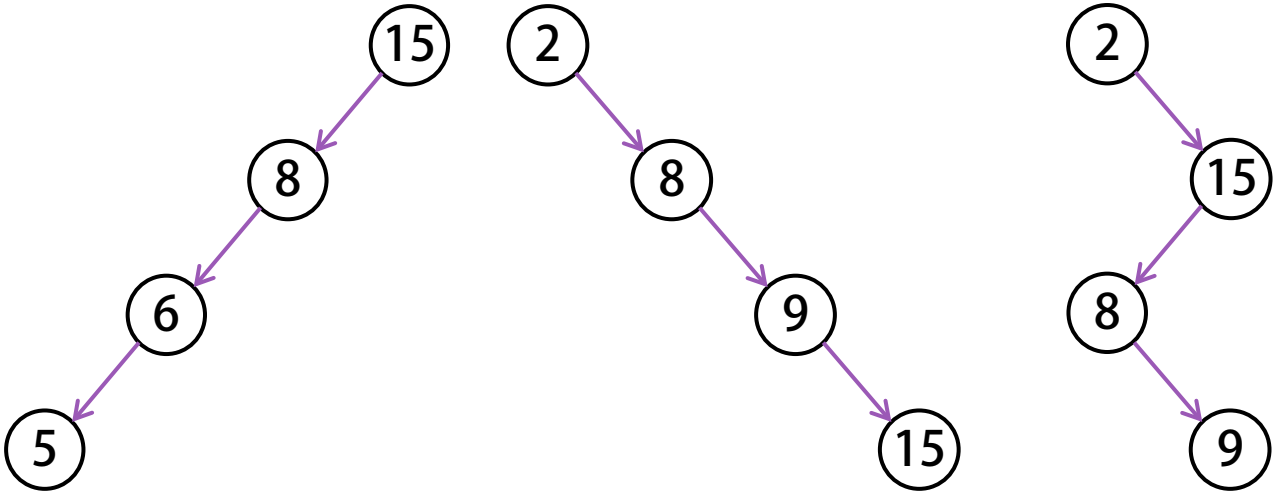
$$\text{平均比較回数} = \frac{1}{2^{d+1}-1}\left(1\times 1+2\times 2+3\times 4+\cdots+(d+1)\times 2^d\right) = \frac{1}{2^{d+1}-1}\left(d2^{d+1}+1\right)$$

$$\approx d = O(\log n)$$

# 最悪の場合



- どの頂点にも子供がひとつであるとき探索の計算量が最大
  - 単純な線形リストとしての働きしかもたない




探索に要する計算量：$O(n)$

実際の木の形は、最善の場合ばかりでも、最悪の場合ばかりでもない
→ 平均的な場合の計算量が知りたい

# 平均的な計算量



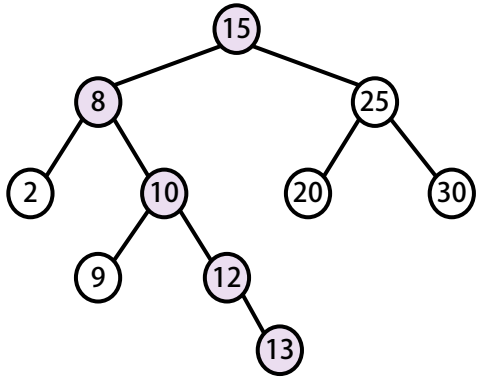
- $n$個の頂点からなる木の探索の平均的な計算量を考えたい
  - それぞれの木の形に対する出現確率が必要
  - 木の形は$n$個のデータの挿入順序で決定
  - データの挿入順序はランダムであると仮定 ($n$!通り・等確率)

$n$!通りの順序に関して平均をとった値を考える

$T_n$：$n$個の頂点の深さの総和

$P_n$：$n$個の頂点からなる木の探索時に調べる頂点の個数

$$P_n = \frac{1}{n} T_n + 1$$




深さ4 → 調べる頂点の個数5

# 漸化式の導出



- ある操作の前後での計算量に着目
  - $n$ 個のデータのうち $k$ 番目の大きさのものを最初に挿入したと仮定
    - 根の左側に $k-1$ 個のデータ
    - 根の右側に $n-k$ 個のデータ

各頂点の根からの深さの総和

$$(k-1+T_{k-1})+0+(n-k+T_{n-k})=n-1+T_{k-1}+T_{n-k}$$

$k$ は 1 から $n$ までを等確率でとるから平均を計算すると

$$T_n=\frac{1}{n}\sum_{k=1}^{n}(n-1+T_{k-1}+T_{n-k})$$

同じ範囲の和

$$=n-1+\frac{1}{n}\sum_{k=1}^{n}T_{k-1}+\frac{1}{n}\sum_{k=1}^{n}T_{n-k}=n-1+\frac{2}{n}\sum_{k=0}^{n-1}T_k$$

# 漸化式の解



- 変数変換を行いつつ計算を進める

$$T_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} T_k \xrightarrow{S_n = T_n - 2} S_n = n + 1 + \frac{2}{n}\sum_{k=0}^{n-1} S_k$$

$n$ 倍

$$nS_n = n(n+1) + 2\sum_{k=0}^{n-1} S_k \xrightarrow{n \to n-1} (n-1)S_{n-1} = n(n-1) + 2\sum_{k=1}^{n-2} S_k$$

差

$$nS_n - (n-1)S_{n-1} = 2n + 2S_{n-1}$$

$$nS_n = (n+1)S_{n-1} + 2n$$

↓

$$\frac{S_n}{n+1} = \frac{S_{n-1}}{n} + \frac{2}{n+1} = \frac{S_{n-2}}{n-1} + \cdots + \frac{2}{n} + \frac{2}{n+1} = \frac{S_1}{2} + \frac{2}{3} + \cdots + \frac{2}{n+1}$$

# 漸化式の解



- 初期値を定めて展開する

$$\frac{S_n}{n+1} = \frac{S_{n-1}}{n} + \frac{2}{n+1} = \frac{S_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \frac{S_1}{2} + \frac{2}{3} + \cdots + \frac{2}{n+1}$$

$T_1 = 0$ すなわち $S_1 = -2$ だから

$$\frac{S_n}{n+1} = 2(H_{n+1} - 2)$$

調和級数：$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

$S_n$ について整理すると

$$S_n = 2(n+1)(H_{n+1} - 2)$$

$$T_n = 2(n+1)(H_{n+1} - 2) + 2$$

$$P_n = \frac{1}{n}[2(n+1)(H_{n+1} - 2) + 2] + 1$$

$H_{n+1} = H_n + \frac{1}{n+1}$ であるから

$$P_n = \frac{2(n+1)}{n} H_n - 3$$

$$H_n \approx \log_e n + \gamma$$

オイラー定数

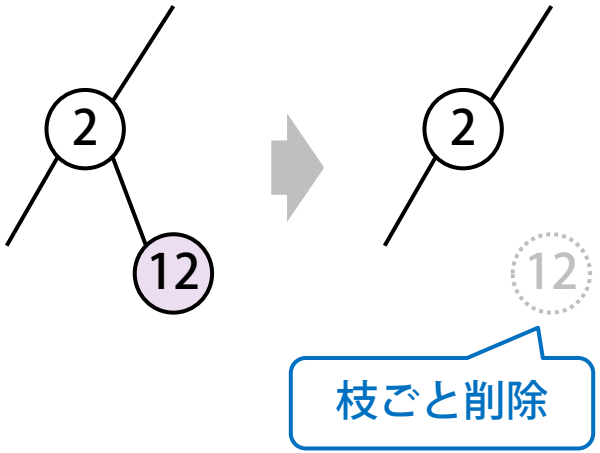
$$P_n \approx 2\log_e n \approx 1.39 \log_2 n$$

# 削除アルゴリズム

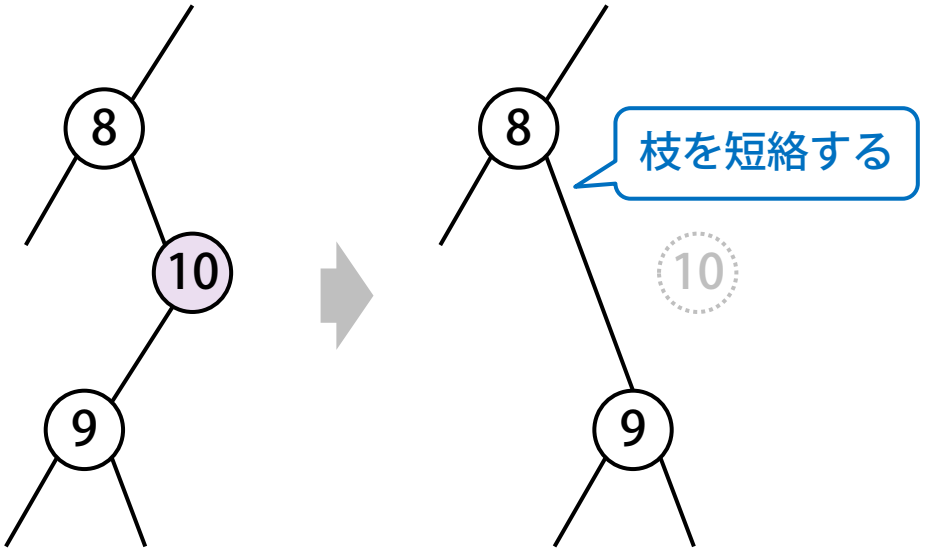


- 除きたい頂点に子が0個か1個のときは簡単
  - 関連する枝と頂点を削除するだけ

子をもたない場合



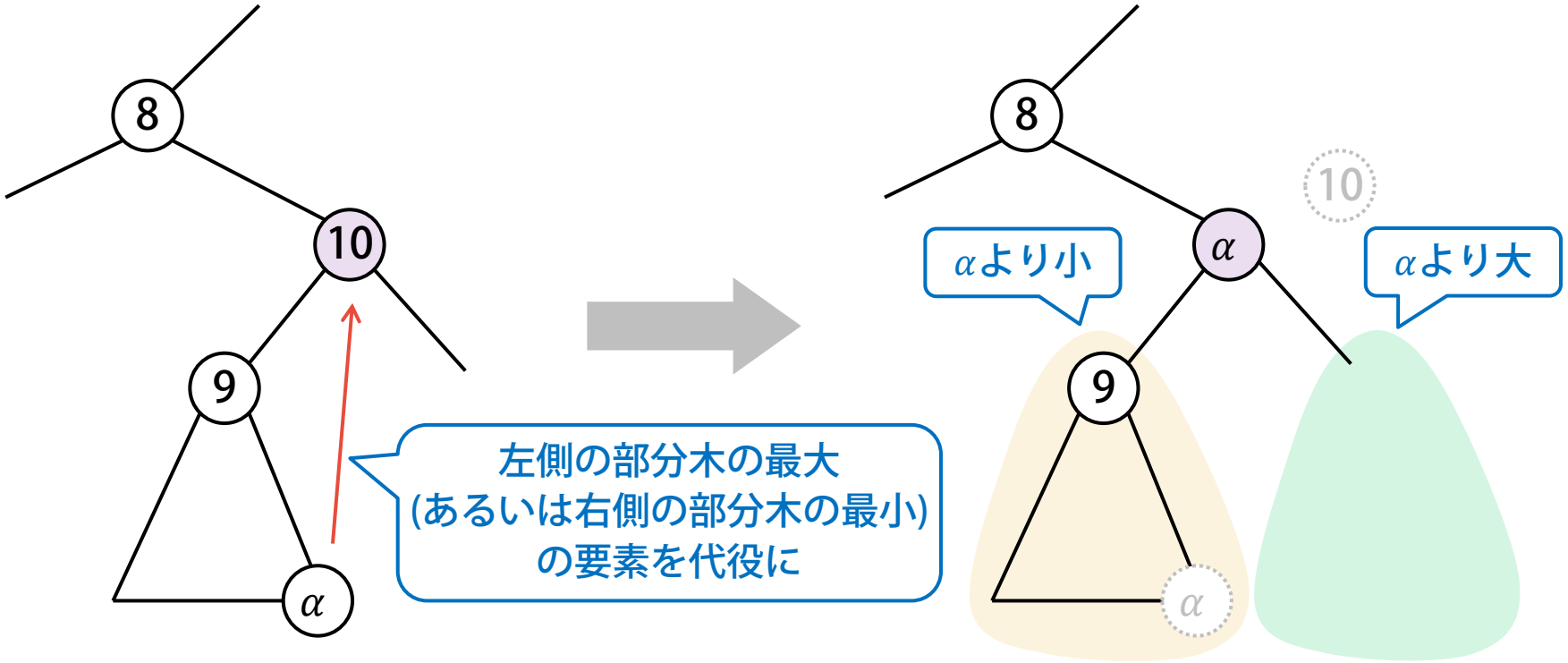

子が左右のうちいずれかの場合



# 削除アルゴリズム



- 除きたい頂点に子が2個存在する場合は木の修正が必要
  - 2分探索木の条件を満たすように頂点の付け替えが発生



# 平衡木



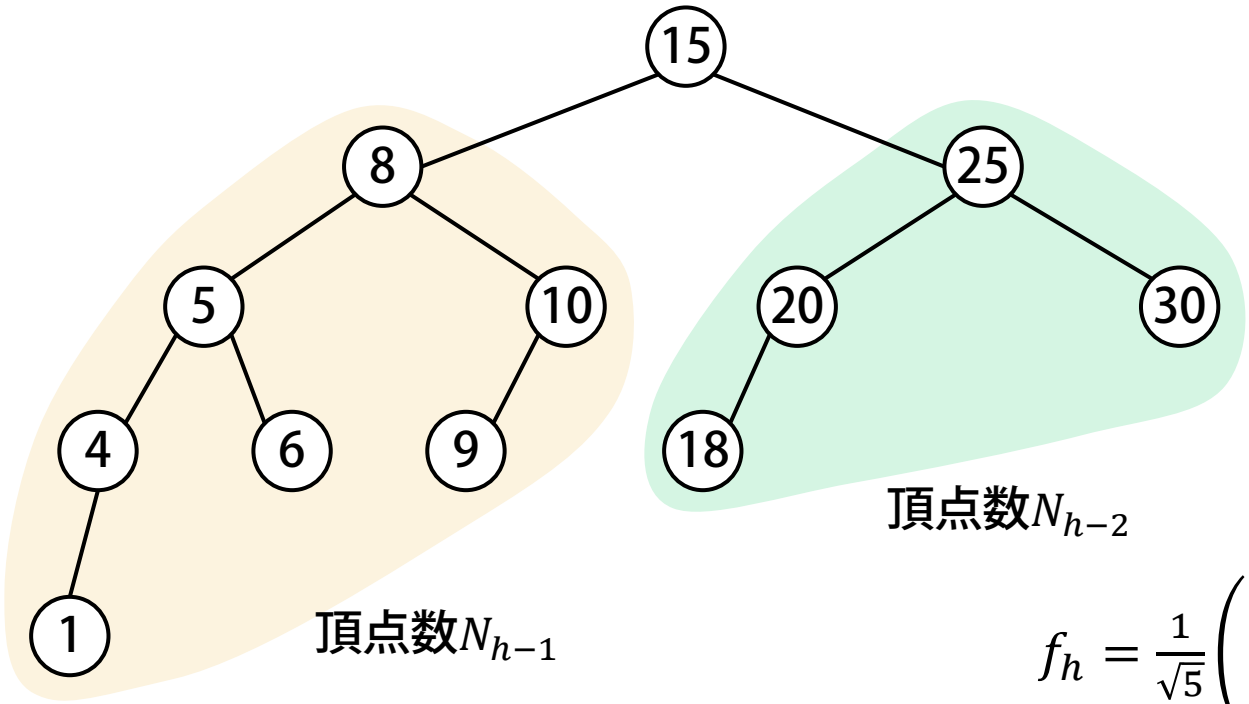
- 最悪の場合の計算量も$O(\log n)$となるデータ構造
  - 通常の二分探索木は平均$O(\log n)$ 最悪$O(n)$
    - 片寄った形のものは性能が悪い
    - バランスが取れた形のものは性能が良い
  - 探索木が偏った形になれば左右のバランスの良い形に作り替える！
    - このときの計算量は$O(\log n)$を越えないことが重要
    - 作り替えに$O(n)$かかってしまったら二分探索木で十分
  - 平衡木の例
    - AVL木 (Adelson-Velskii and Landis' tree)
    - 赤黒木 (red-black tree)
    - スプレー木 (splay tree)

# AVL木 (1962)



- どの頂点においても左部分木と右部分木の高さの差が +1, 0, -1 のいずれかである二分探索木

高さ$h = 4$の頂点数最小のAVL木の例




高さ$h$のAVL木の最小頂点数

$$N_h = N_{h-1} + N_{h-2} + 1$$

$$N_0 = 0, N_1 = 2$$

$f_h = N_h + 1$ とおく

$$f_h = f_{h-1} + f_{h-2}$$

$$f_0 = 2, f_1 = 3$$

フィボナッチ数列

$$f_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+3}\right)$$

# 最悪計算量



- 最悪の場合でも探索計算量は$O(\log n)$
  - 高さ$h$のAVL木に対して頂点数は指数的に増加
  - AVL木の頂点数$n$に対して高さは対数オーダで増加

$$N_h = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+3}\right) - 1$$

↓ $h$が大きくなると

$$N_h \approx \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}\right)$$

$$\log_2 \frac{1+\sqrt{5}}{2} = \frac{1}{1.44}$$

AVL木による探索は完全二分木と比較して
最悪でも1.44倍しか遅くはならない！

二分探索木の平均$O$(約$1.39 \log n$)
AVL木の平均$O$(約$1.0 \log n$) (証明は省略)

# 挿入アルゴリズム

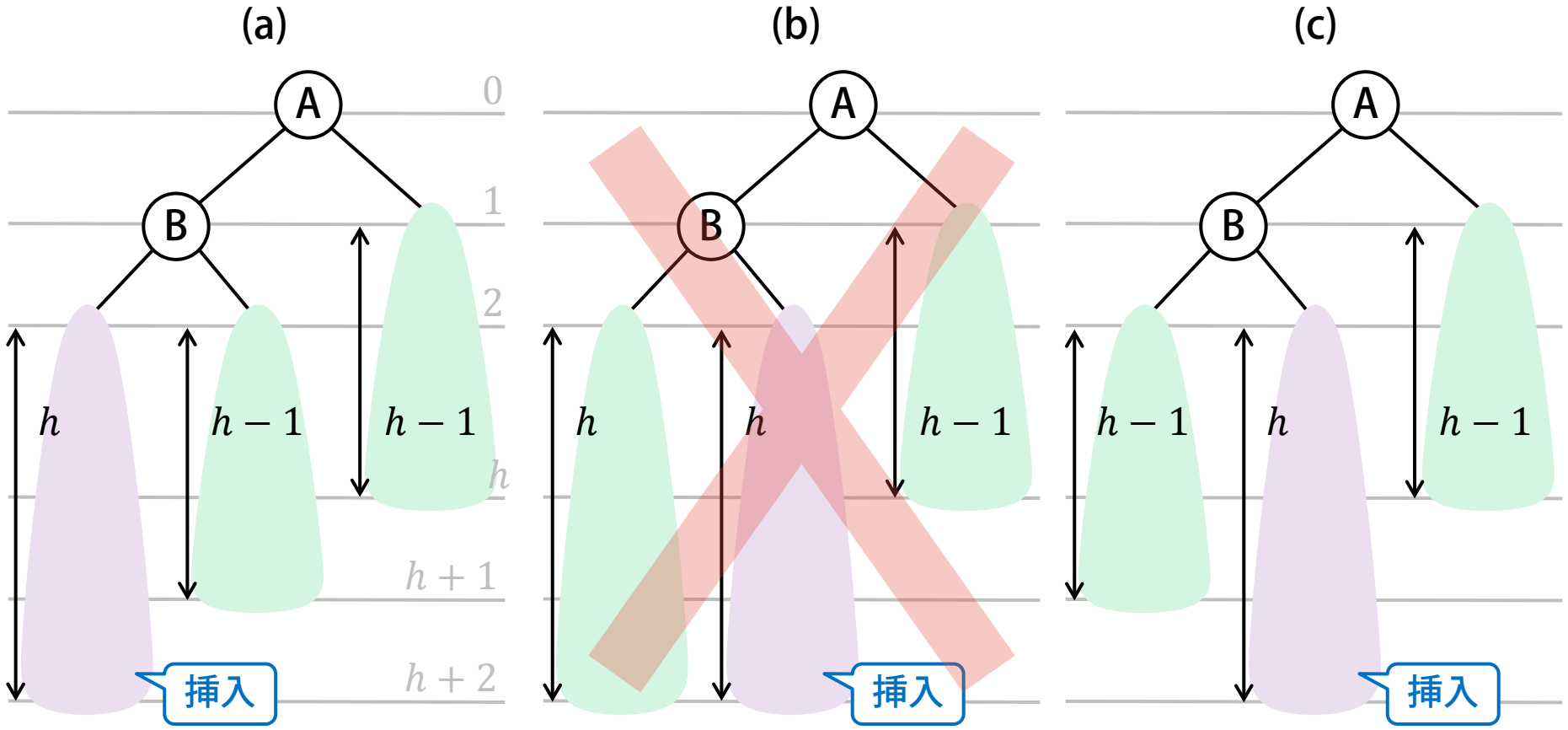


- 通常の二分探索木と同様にデータを挿入してから必要であれば木を作り替える
  - ある頂点から見て左の部分木に挿入した結果、左の部分木の高さが１だけ増える状況を考える
    - 場合１：当初、右の部分木の高さが１だけ高かった場合
      - 左右の木の高さが等しくなるので操作不要
    - 場合２：当初、左右の木の高さが等しかった場合
      - 左の高さが右の高さ１だけ大きくなるが許容範囲内
    - 場合３：当初、左の部分木の高さが１だけ高かった場合
      - 左右の高さの差が２となるので作り替えが必要

# データ挿入直後の状態



- 左右の高さが２となる場合にはいくつかパターンが存在

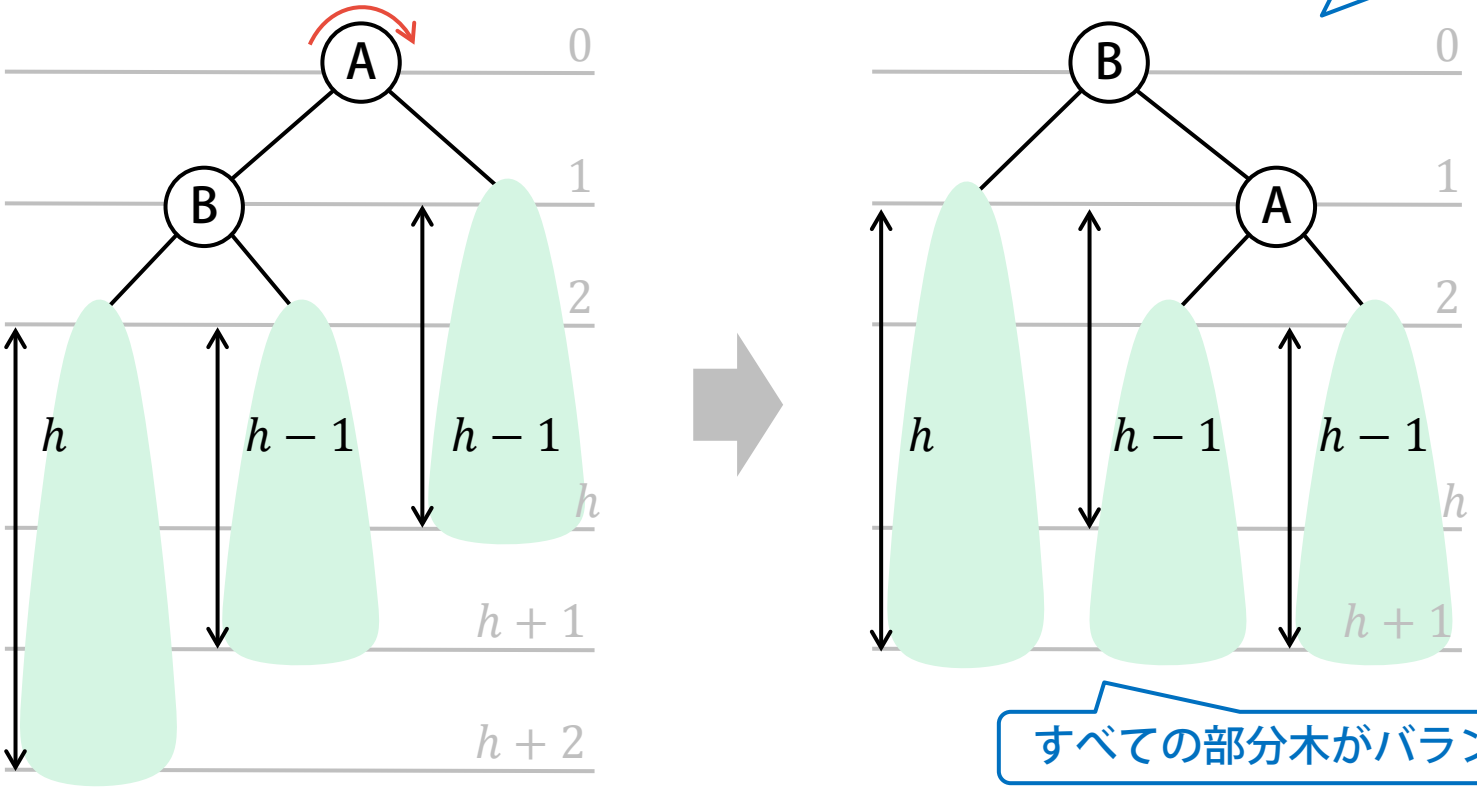


# 一重回転



- (a)の場合に木をバランスさせる方法
  - 頂点AとBの役割を入れ替える：Bを親に・Aを子に

回転は$O(1)$

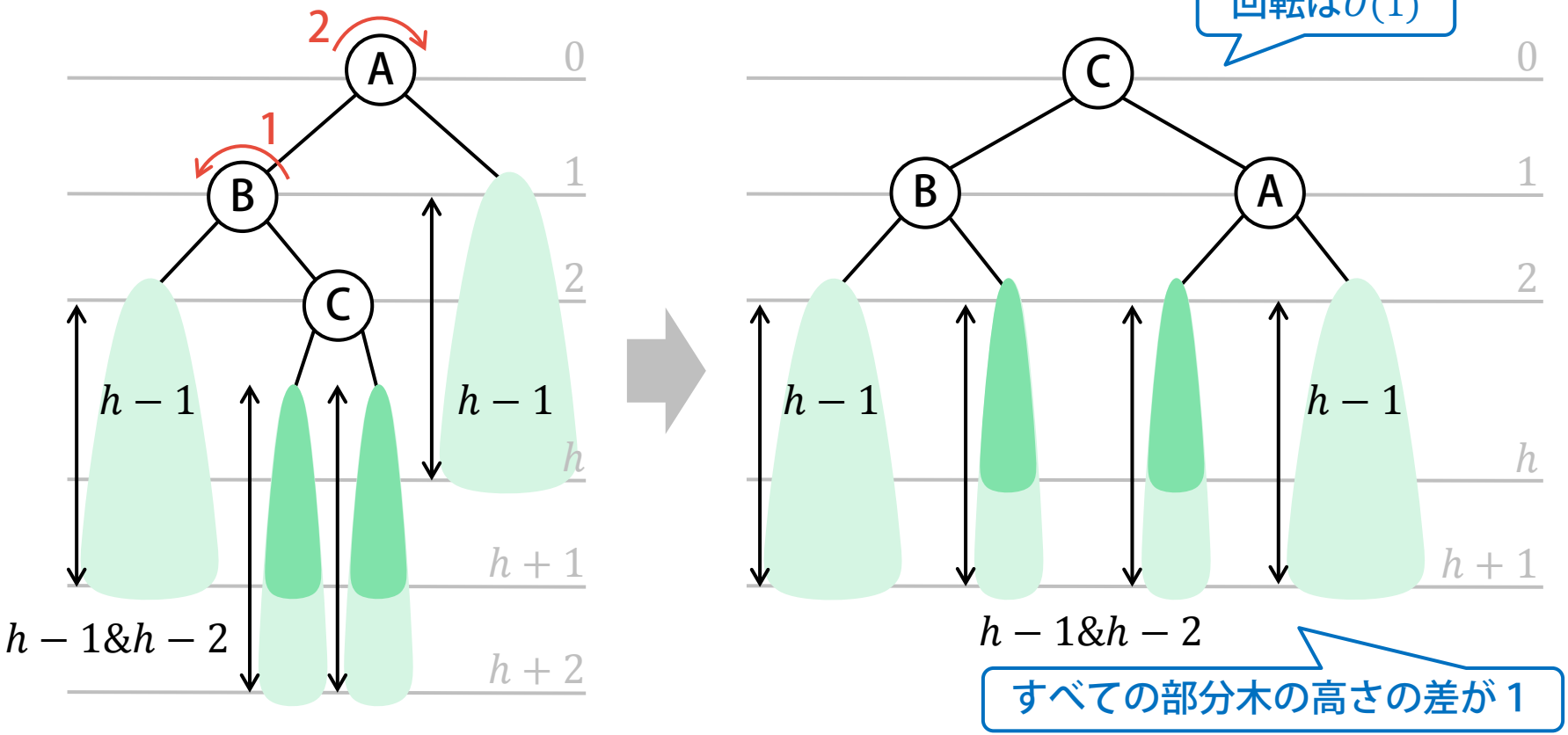



すべての部分木がバランス

# 二重回転



- (c)の場合に木をバランスさせる方法
  - 部分木を２つに分割してから頂点を入れ替える



# 削除アルゴリズム

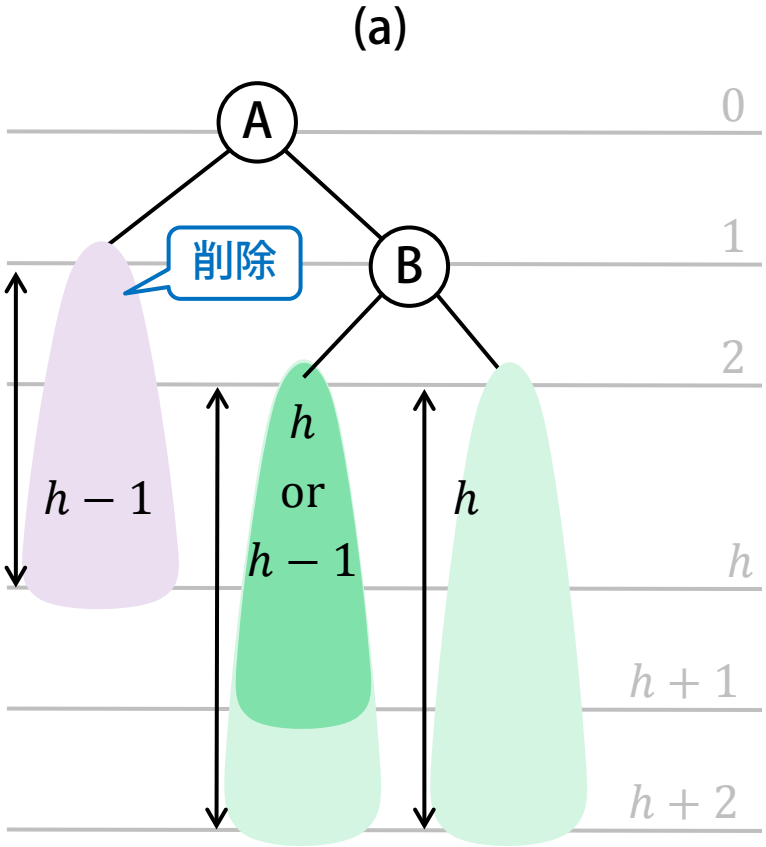


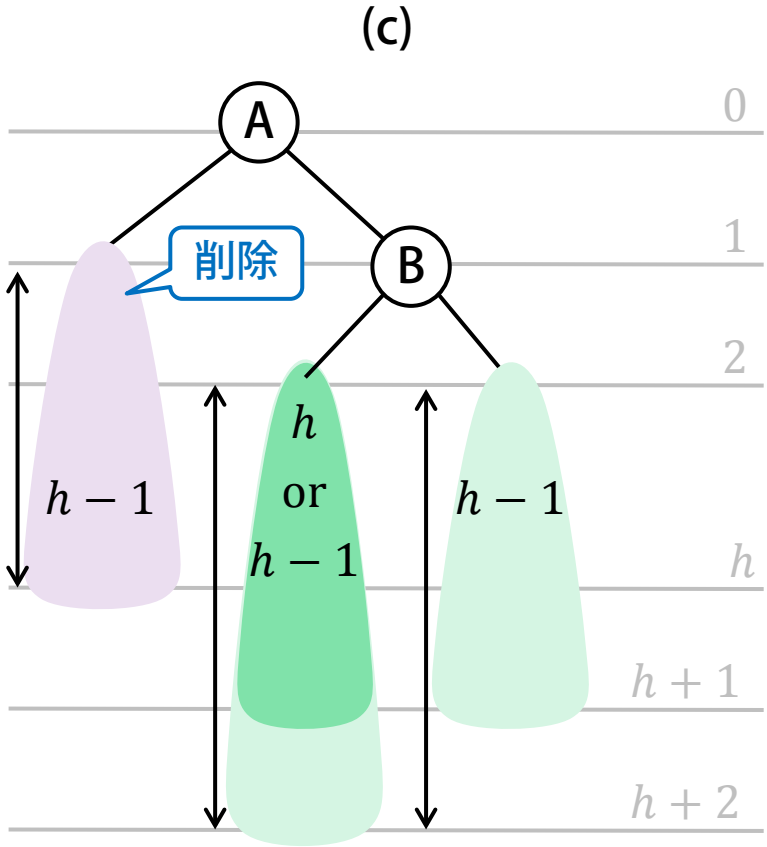
- 通常の二分探索木と同様にデータを削除してから必要であれば木を作り替える
  - ある頂点から見て左の部分木から削除した結果、左の部分木の高さが１だけ減る状況を考える
    - 場合１：当初、左の部分木の高さが１だけ高かった場合
      - 左右の木の高さが等しくなるので操作不要
    - 場合２：当初、左右の木の高さが等しかった場合
      - 左の高さが右の高さ１だけ小さくなるが許容範囲内
    - 場合３：当初、左の部分木の高さが１だけ低かった場合
      - 左右の高さの差が２となるので作り替えが必要

# データ削除直後の状態



- 左右の高さが２となる場合にはいくつかパターンが存在

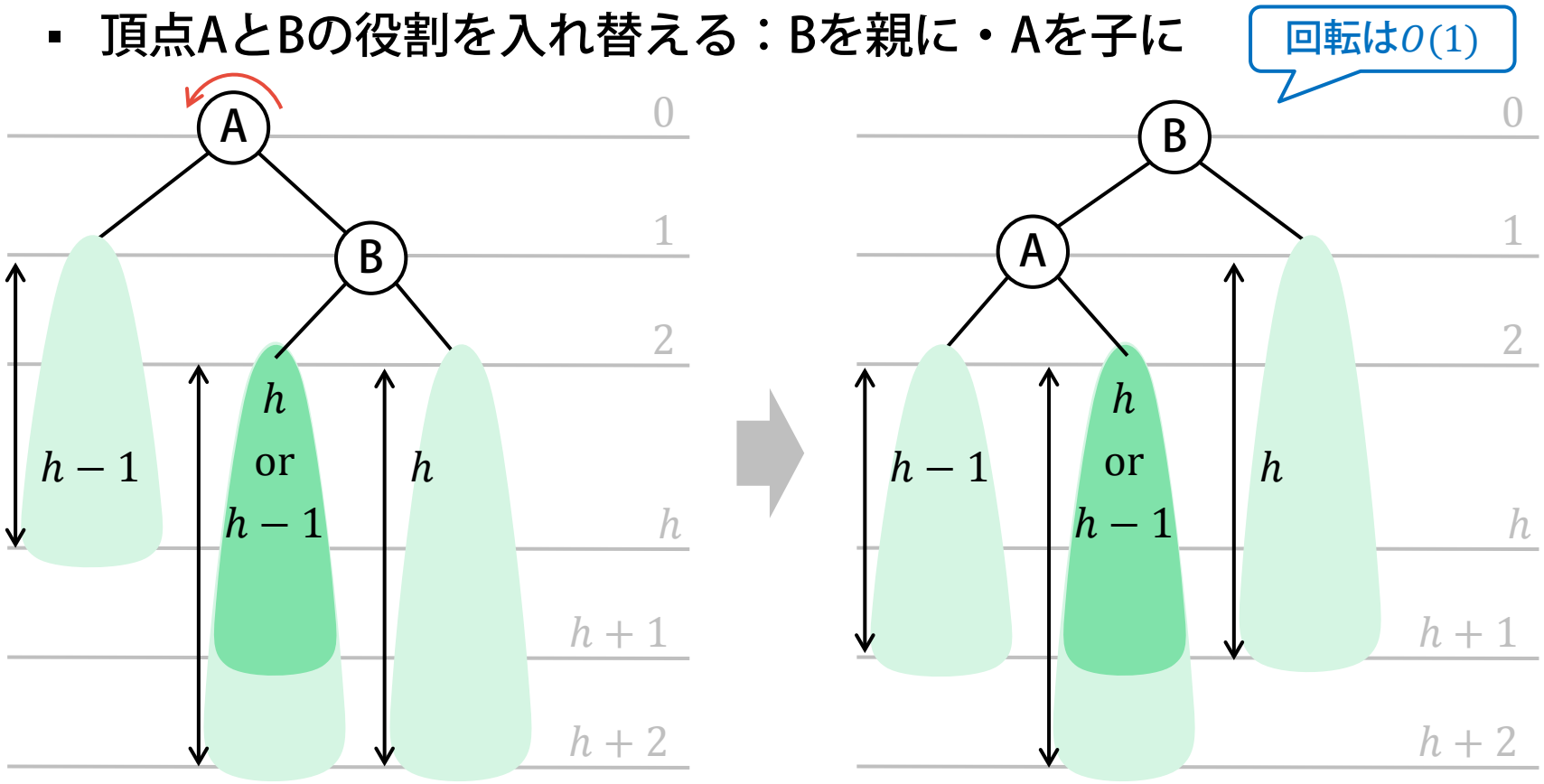


# 一重回転



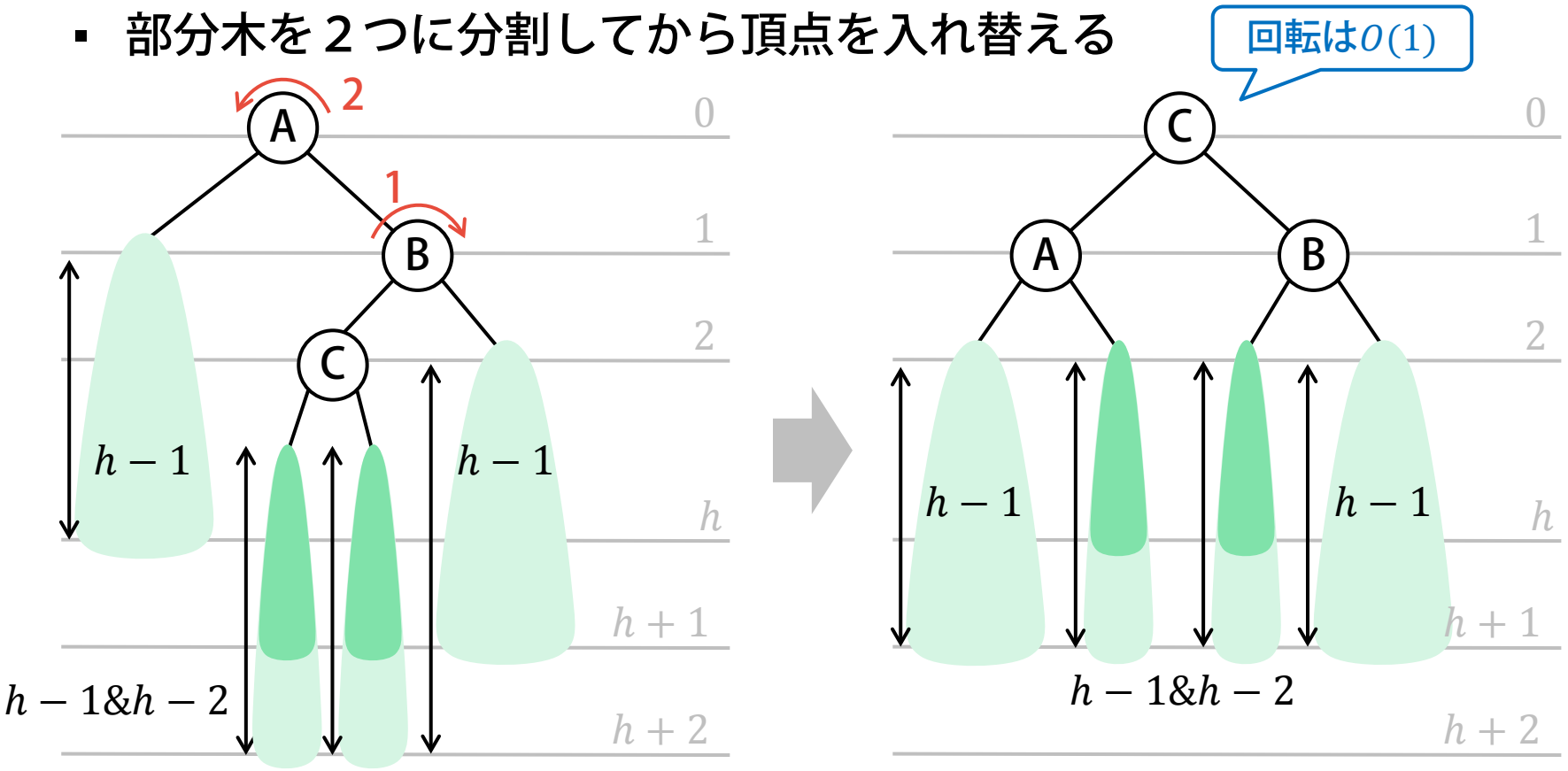
- (a)の場合に木をバランスさせる方法
  - 頂点AとBの役割を入れ替える：Bを親に・Aを子に



# 二重回転



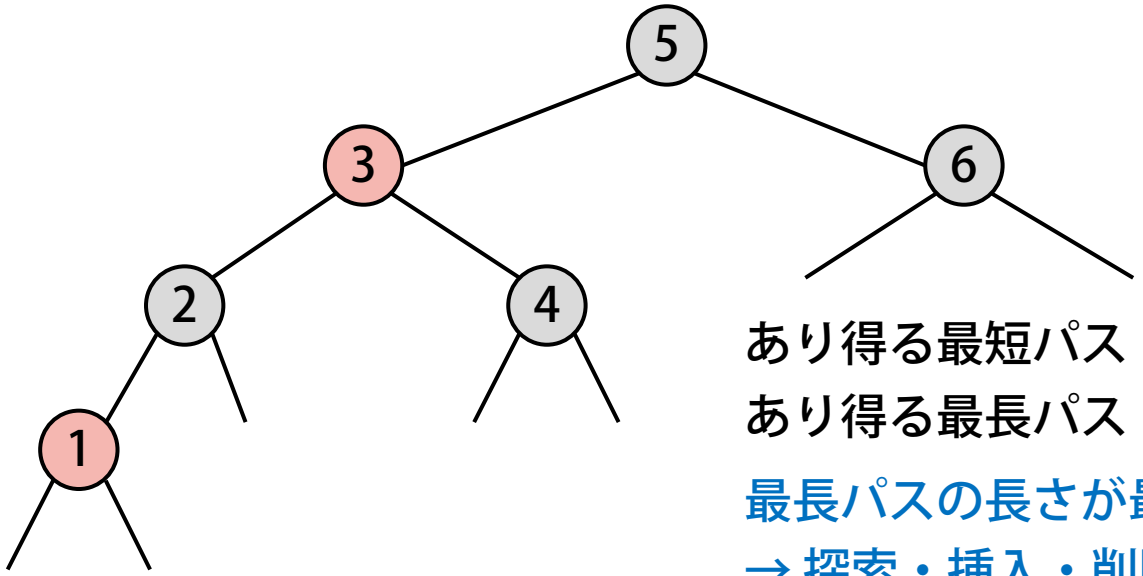
- (c)の場合に木をバランスさせる方法
  - 部分木を２つに分割してから頂点を入れ替える



# 赤黒木 (1972)



- 以下の制約を満たす二分探索木
  - 条件1：各頂点は赤か黒 (ただし根は黒)
  - 条件2：赤い頂点の子は黒
  - 条件3：全ての葉から根へのパス上には同じ個数の黒の頂点が存在




あり得る最短パス：黒のみを含む

あり得る最長パス：赤と黒が交互に出現

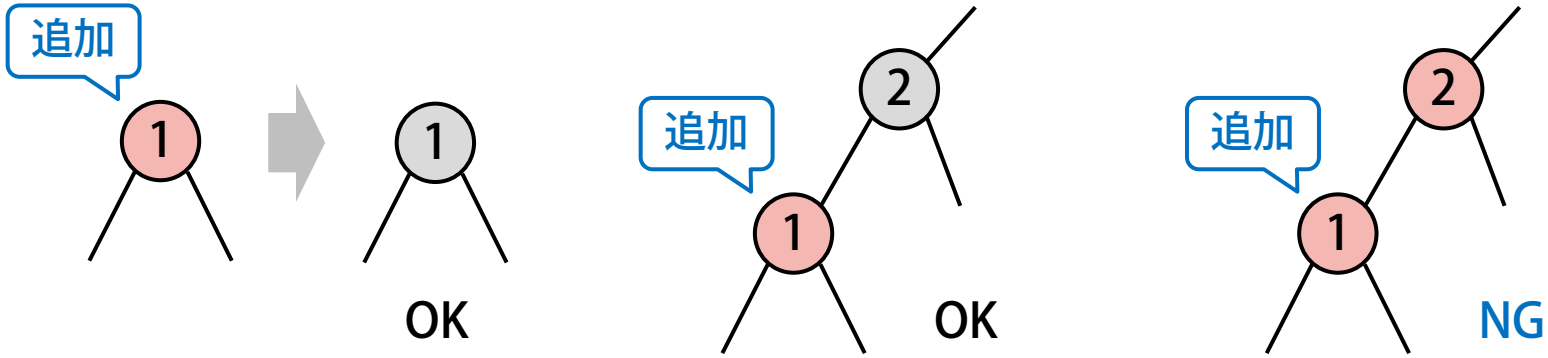
最長パスの長さが最短パスの２倍以内に収まる

→ 探索・挿入・削除が$O(\log n)$で可能

# 挿入アルゴリズム



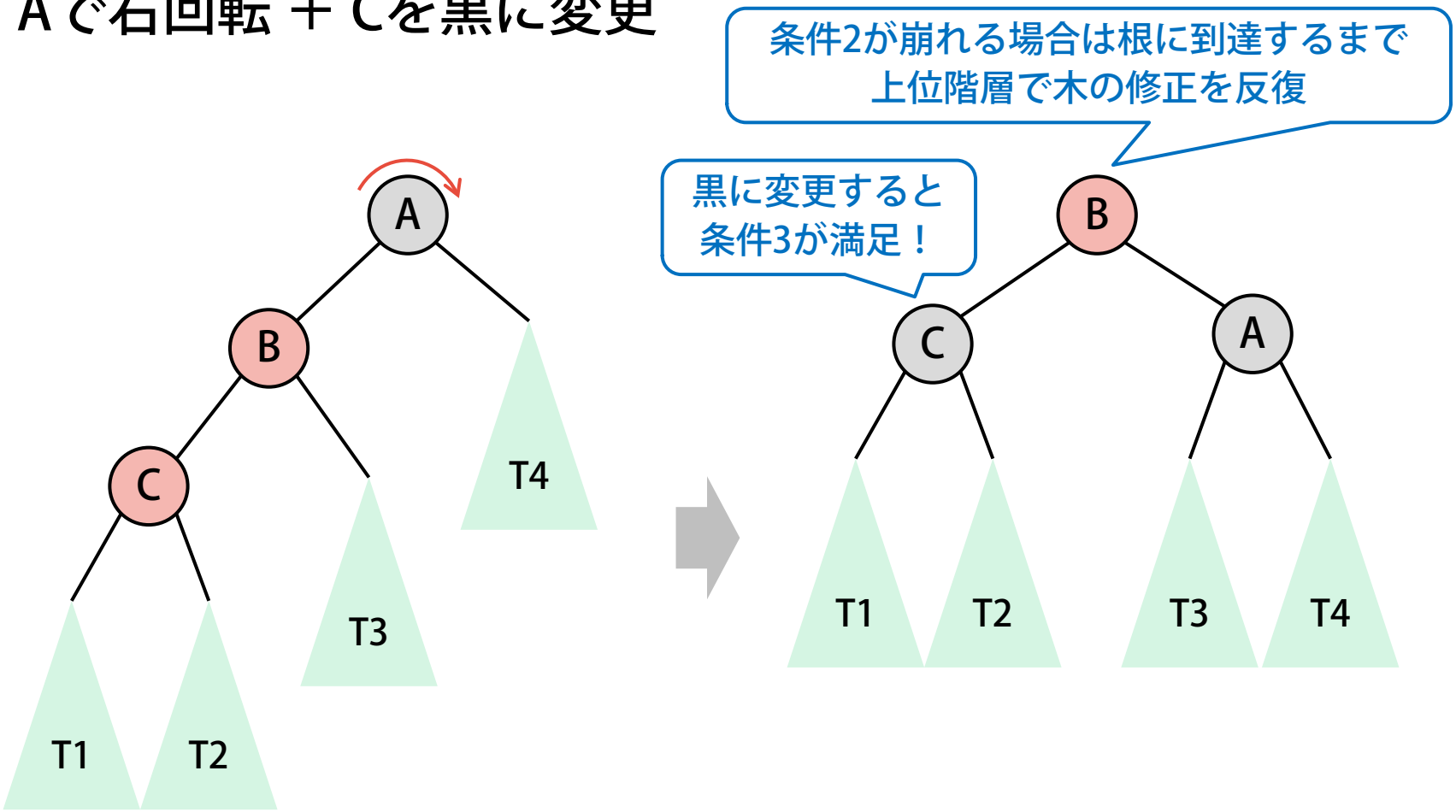
- 通常の二分探索木と同様にデータを挿入してから必要であれば木を作り替える
  - 新しい頂点は常に赤で追加
    - 新しい頂点が根なら黒に変更
    - 新しい頂点が黒の頂点の子なら何もしない
    - 新しい頂点が赤の頂点の子なら修正が必要
      - 修正後、条件3が守られるように注意



# パターンLL：一重回転



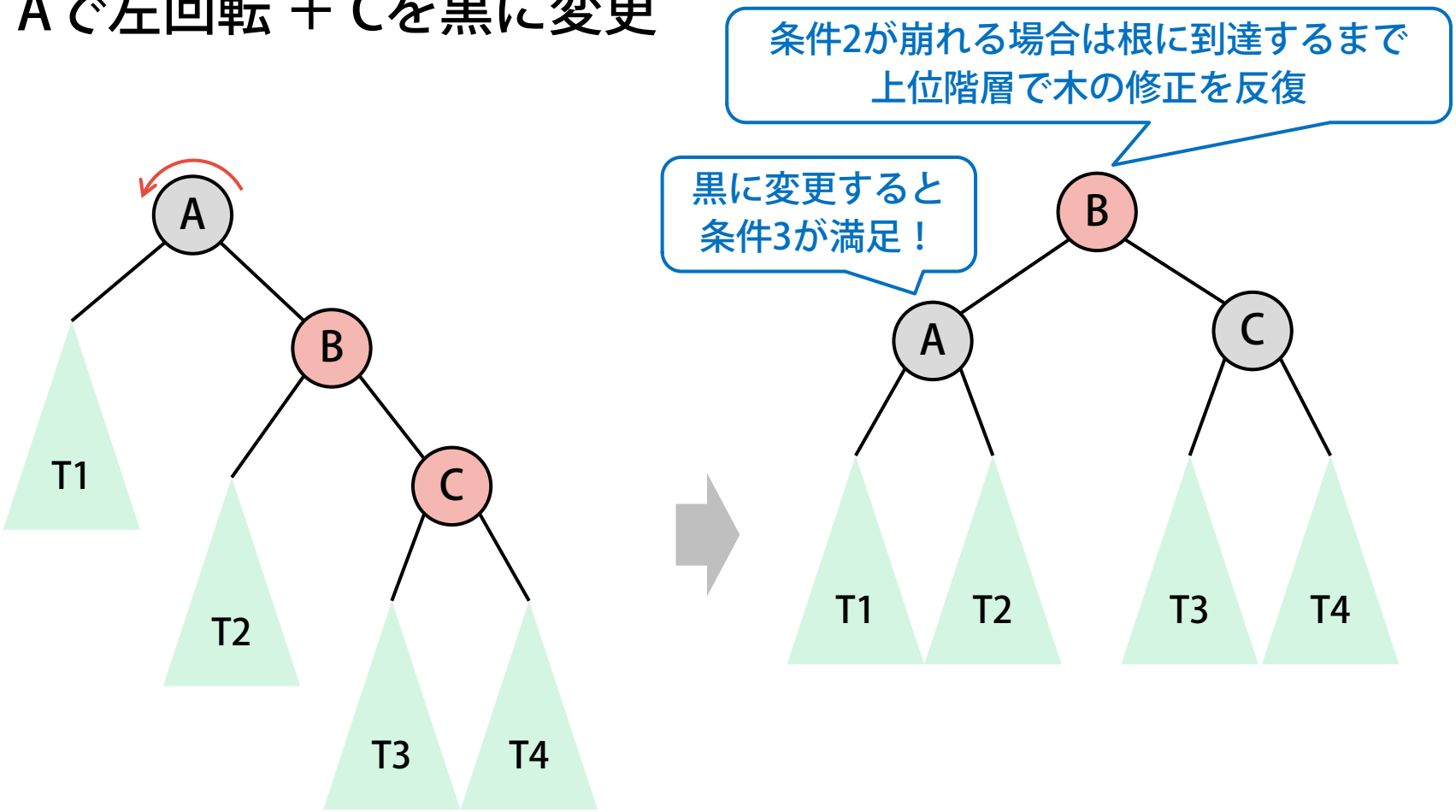
- Aで右回転 ＋ Cを黒に変更



# パターンRR：一重回転



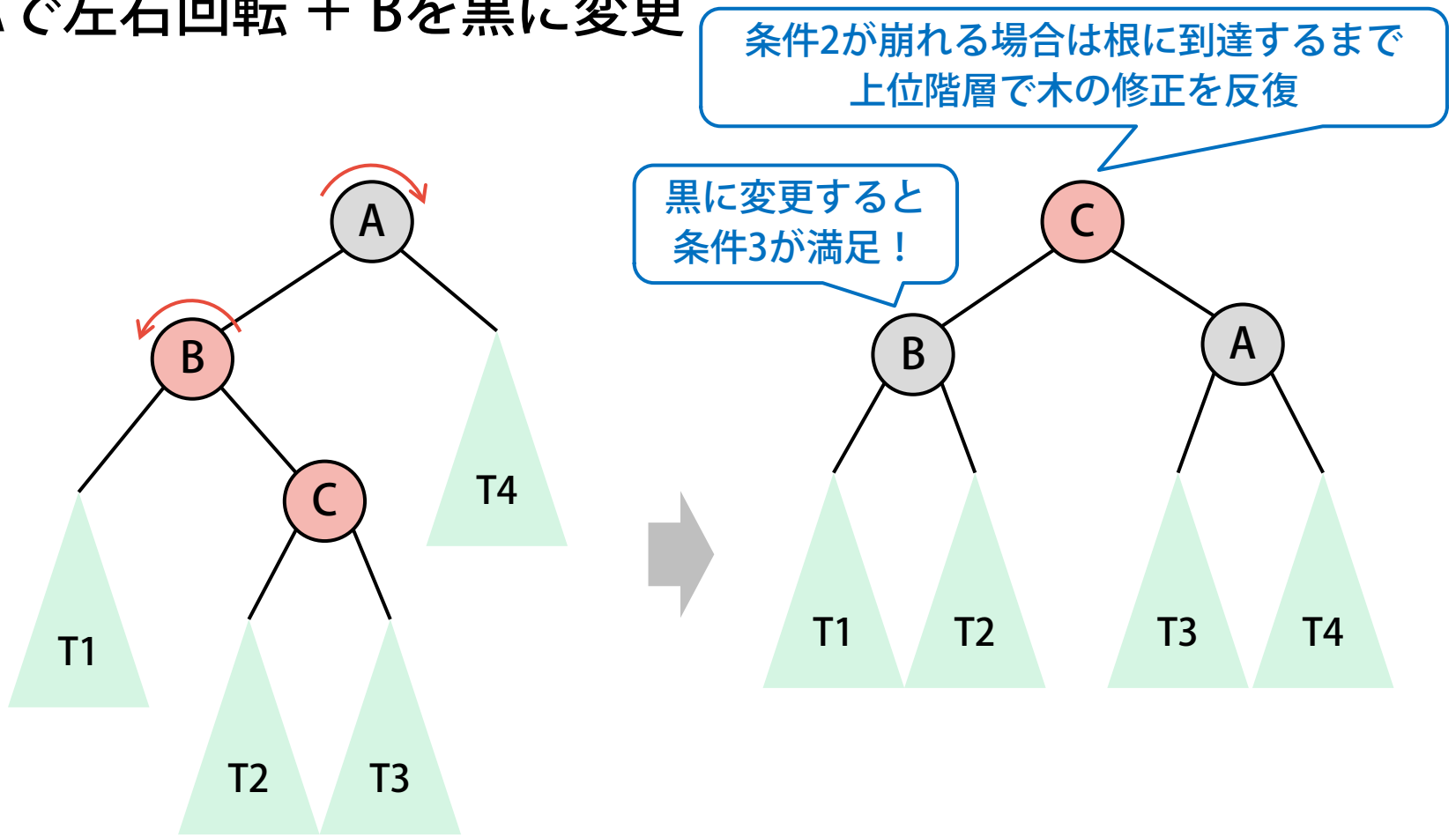
- Aで左回転 ＋ Cを黒に変更



# パターンLR：二重回転



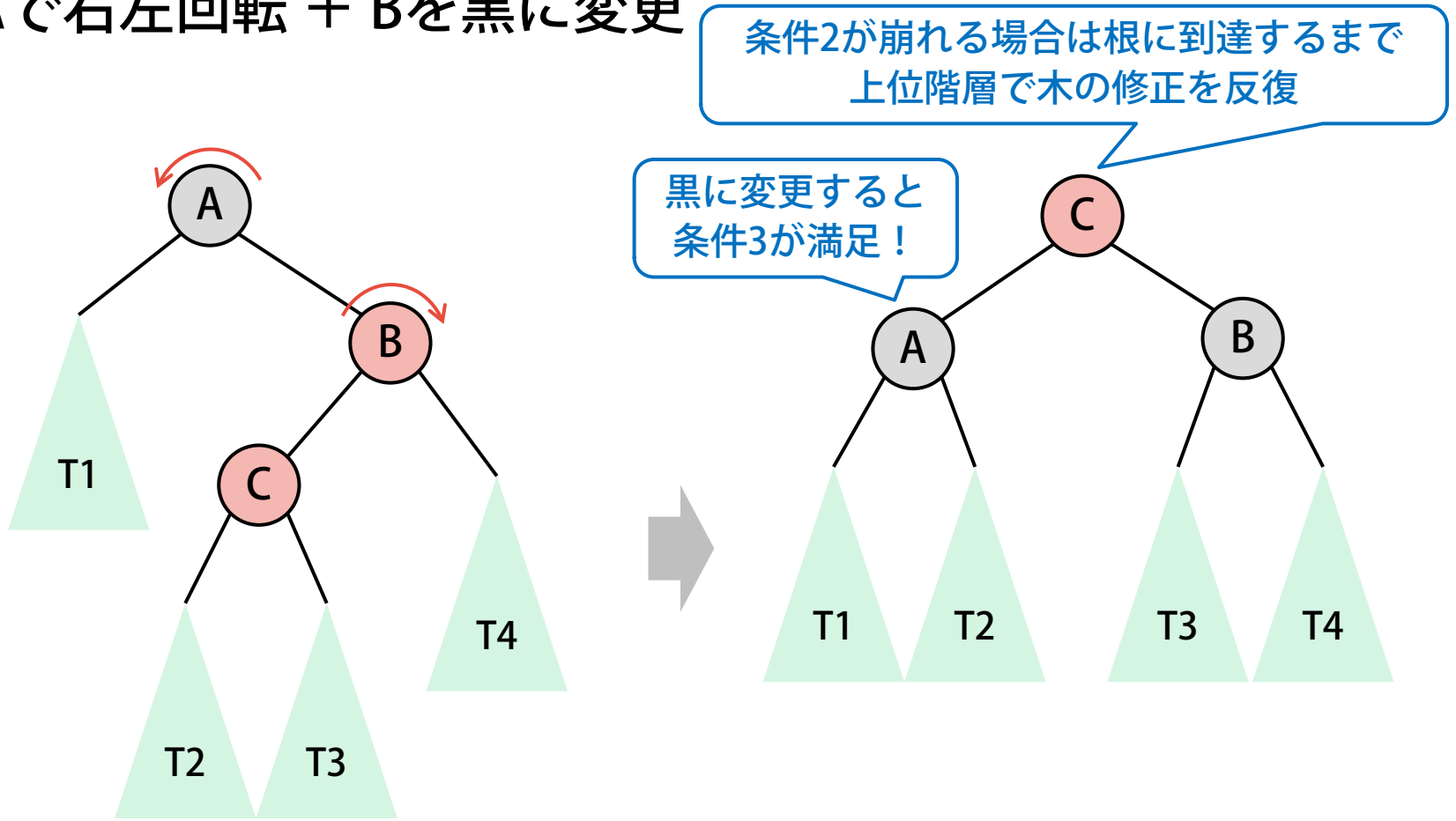
- Aで左右回転 ＋ Bを黒に変更



# パターンRL：二重回転



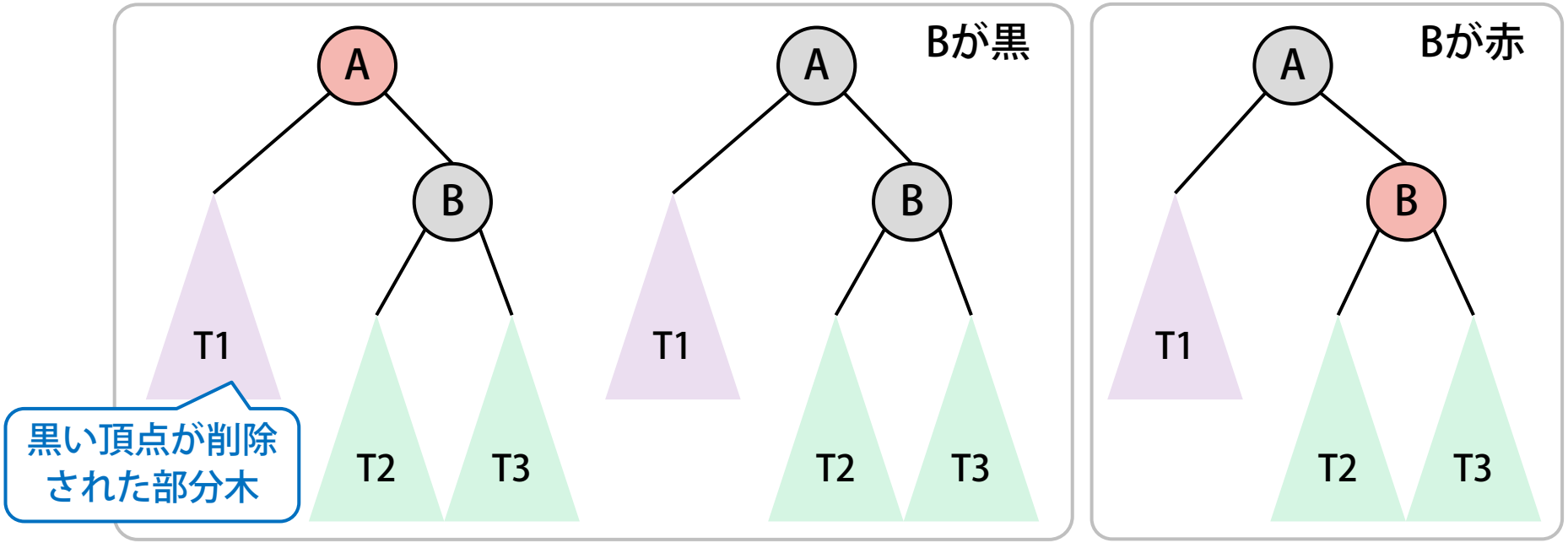
- Aで右左回転 ＋ Bを黒に変更



# 削除アルゴリズム



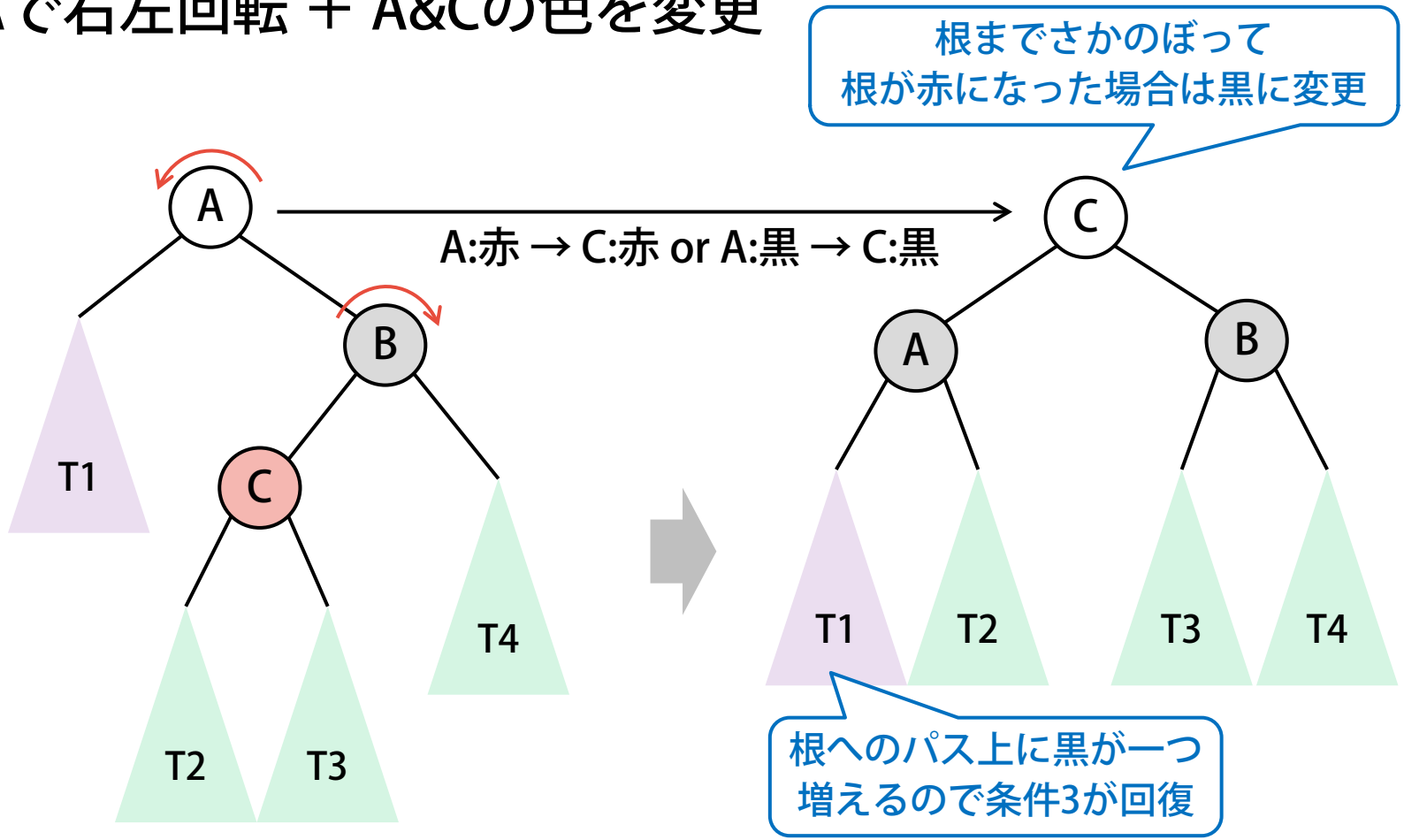
- 通常の二分探索木と同様にデータを削除してから必要であれば(再帰的に)木を作り替える
  - 赤い頂点が削除されたら何もしない
  - 黒い頂点が削除された場合は条件3を満たすように修正が必要



# パターンL1：二重回転



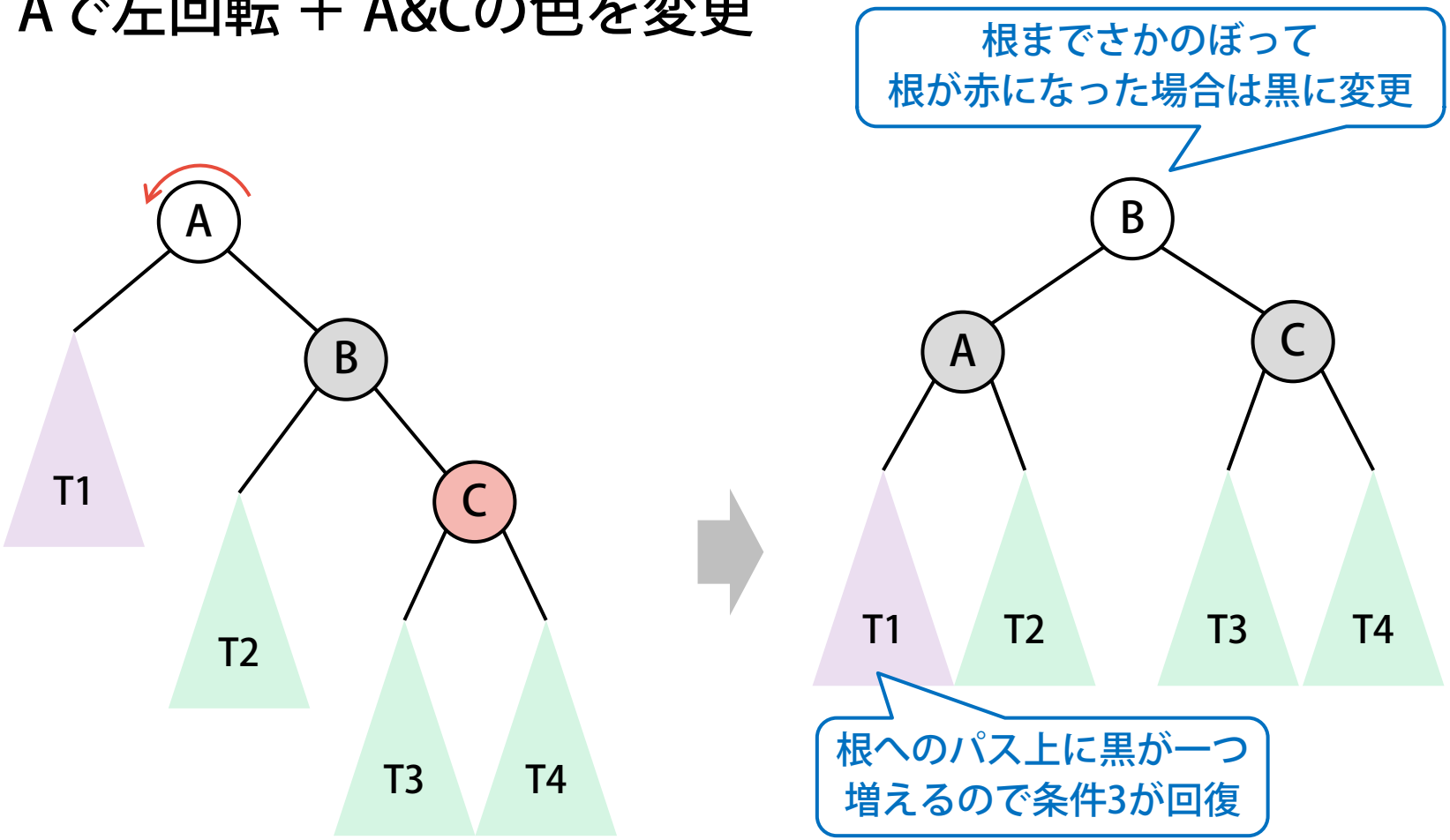
- Aで右左回転 ＋ A&Cの色を変更



# パターンL2：一重回転



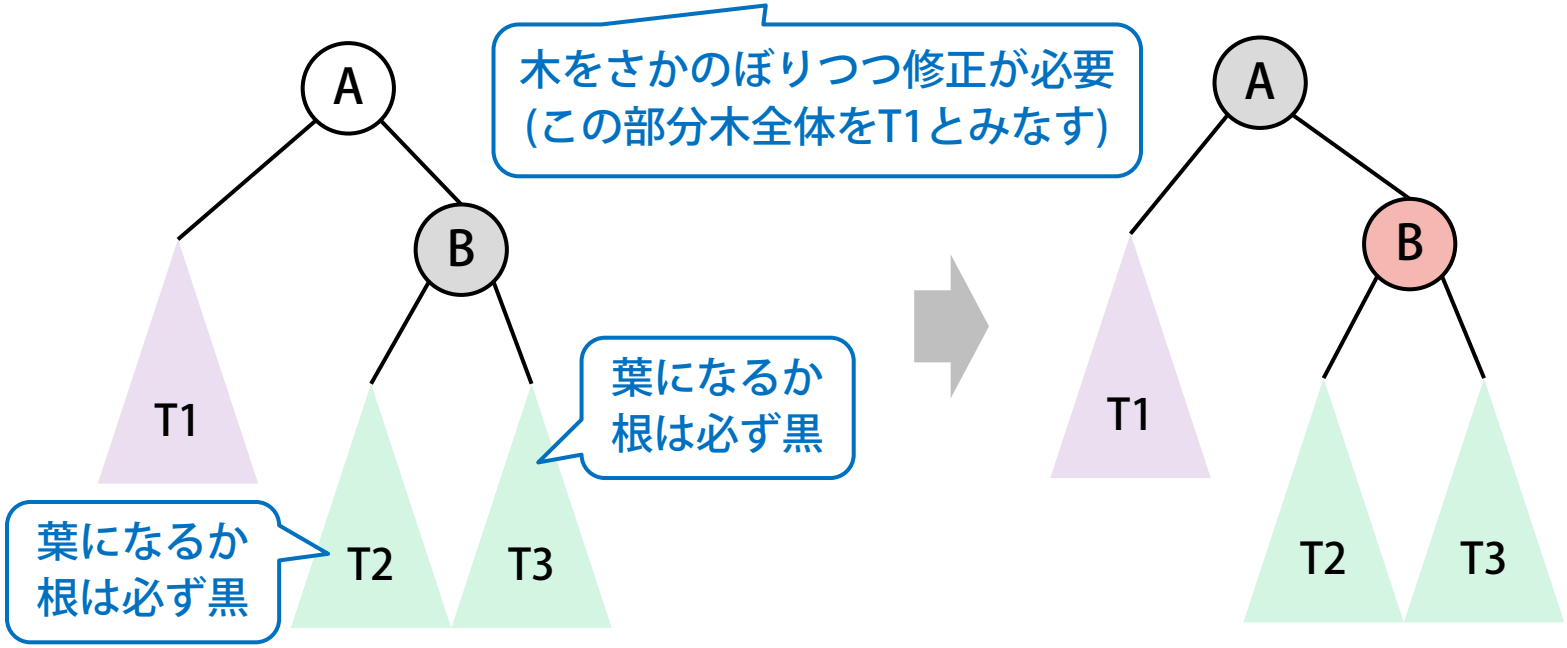
- Aで左回転 ＋ A&Cの色を変更



# パターンL3 (パターンL1&L2以外)



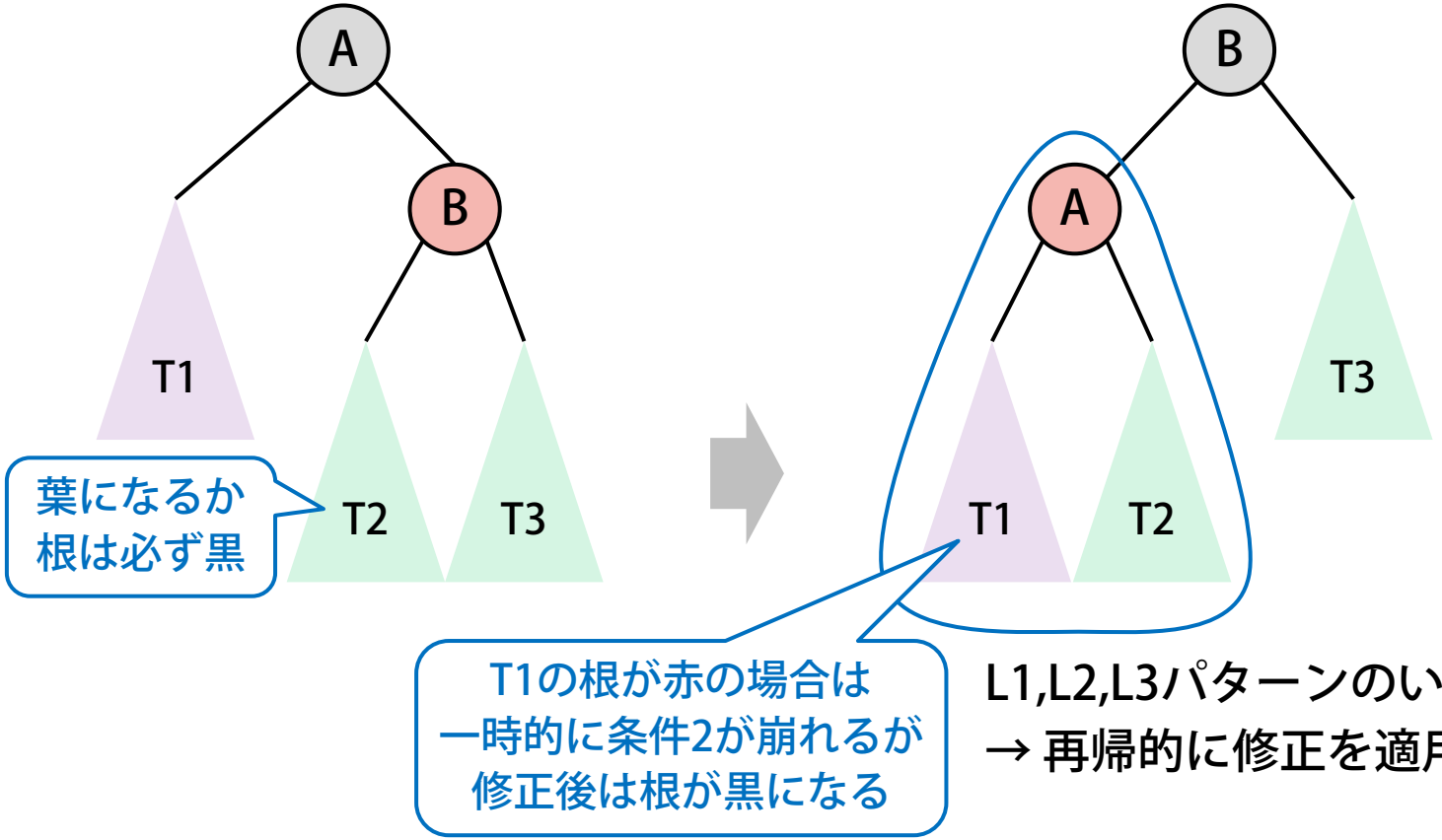
- Aを黒に変更 ＋ Bを赤に変更
  - この修正を行っても条件2は保持
  - Aが赤であった場合は条件3は保持 (T1から根へのパス上に黒が増加)
  - Aが黒であった場合は部分木全体で根へのパス上の黒が減少



# パターンL4



- Aで左回転＋A&Bで色を交換



# スプレー木 (1985)

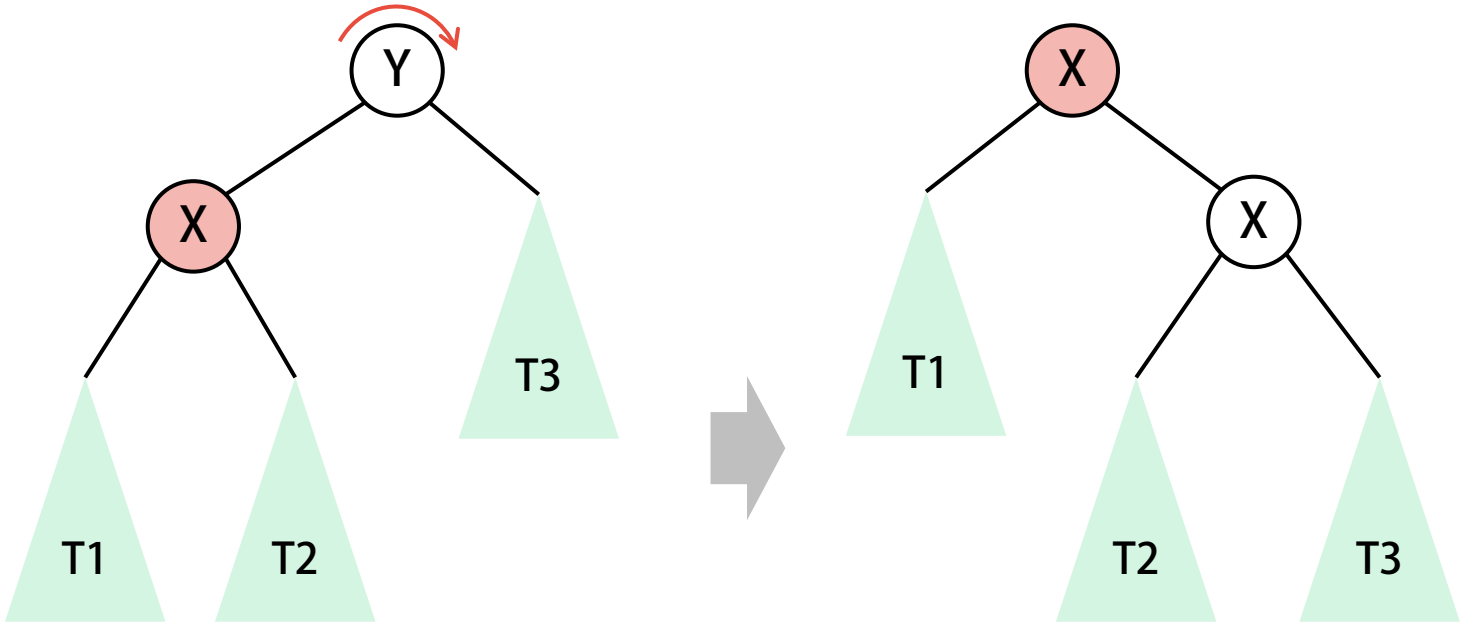


- アクセスした頂点を根に移動させる平衡二分探索木
  - データアクセスの時間局所性を利用
    - 頻繁にアクセスされるデータほど浅いレベルに集める
    - キャッシュやガベージコレクションの実装に便利
  - 他の平衡二分探索木と似た性能を示す
    - 最悪の場合には計算量は$O(n)$だが平均的には$O(\log n)$
  - スプレー操作
    - zipステップ
    - zig-zigステップ
    - zig-zagステップ

# zigステップ



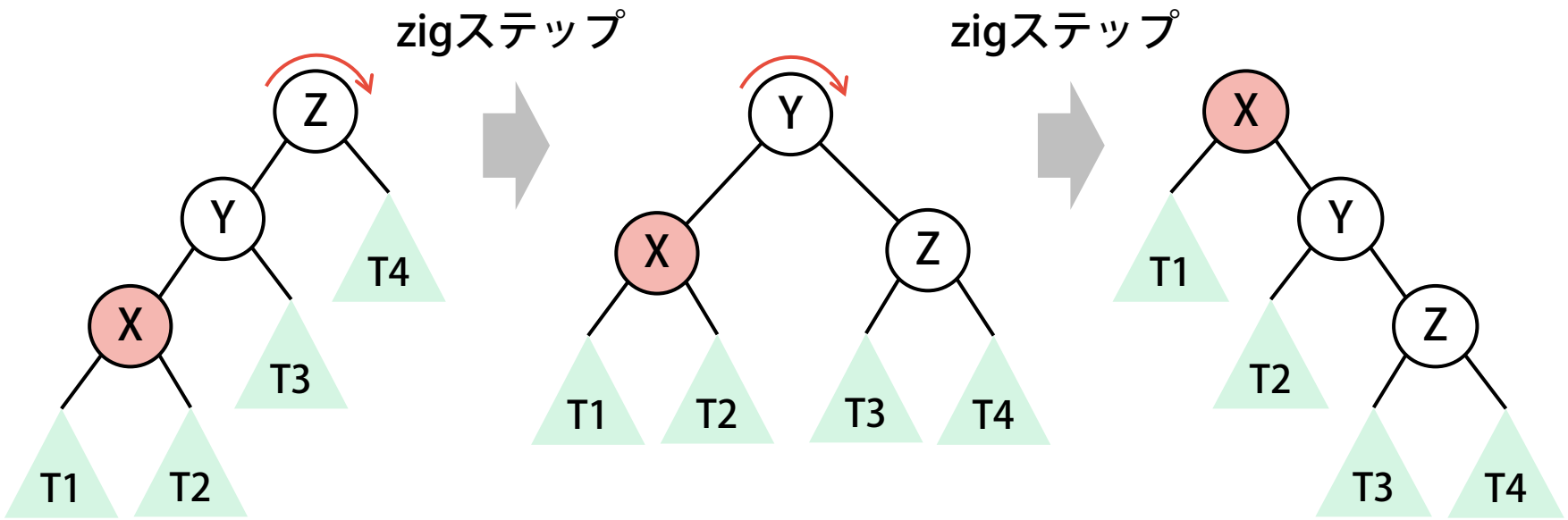
- Yが根のときに適用
  - 通常の回転操作と同様：Yを回転



# zig-zigステップ



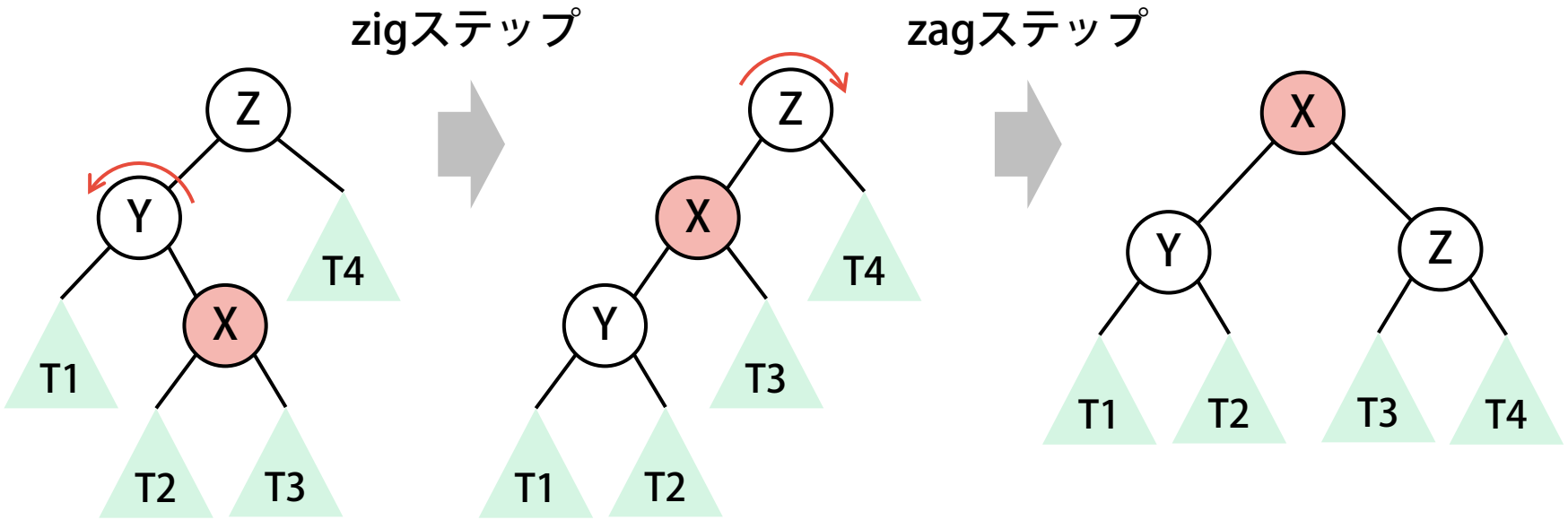
- Yが根ではなくX<Y<Zのときに適用
  - 回転の順番が重要：Zを回転 → Yを回転



# zig-zagステップ



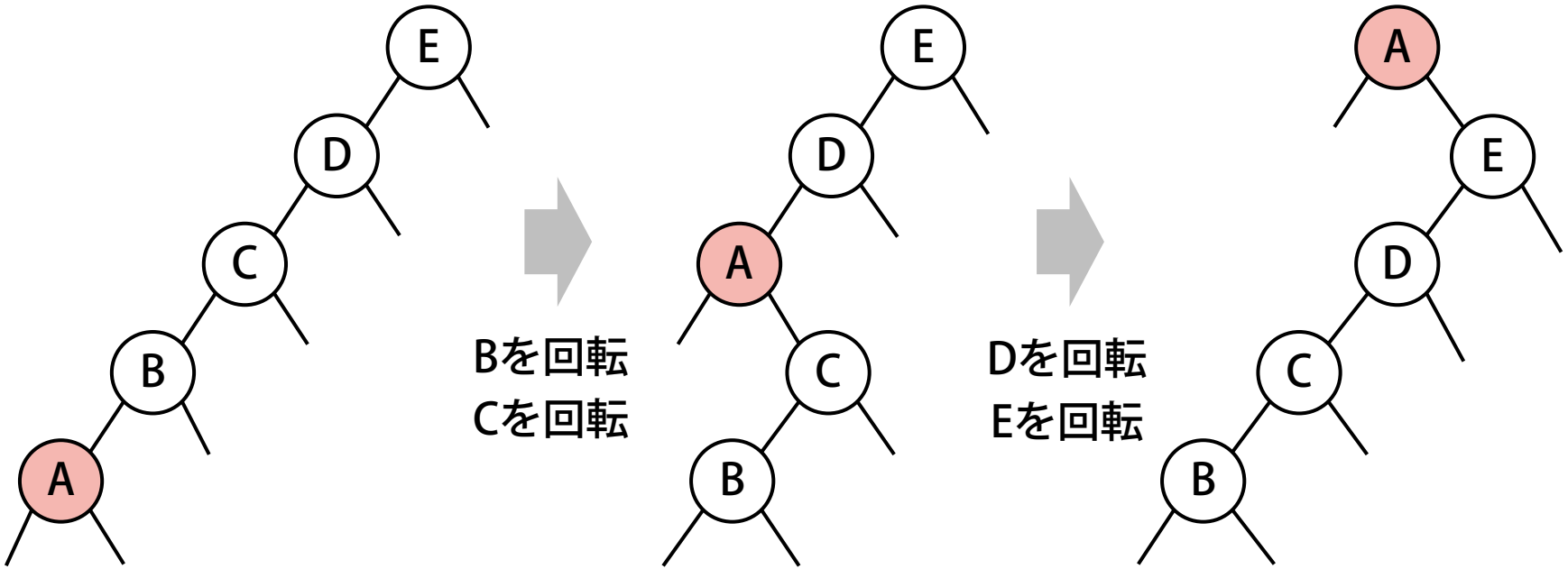
- Yが根ではなくY<X<Zのときに適用
  - 回転の順番が重要：Yを回転 → Zを回転



# 回転操作によるMove to Root



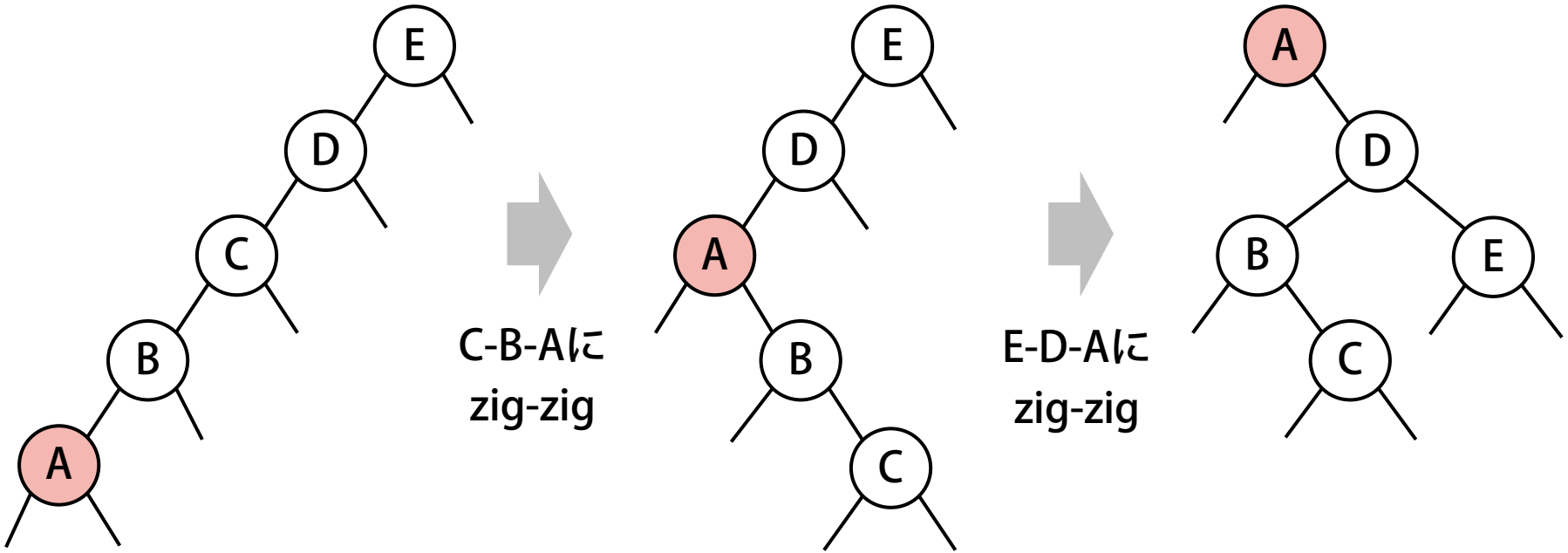
- 木の高さは変化しない
  - 深い木は深い木のままなので探索効率が向上しない



# スプレー操作によるMove to Root



- 木の高さがひとつ減少する
  - 平衡二分探索木としての機能を発揮：探索効率が向上



# ハッシュ



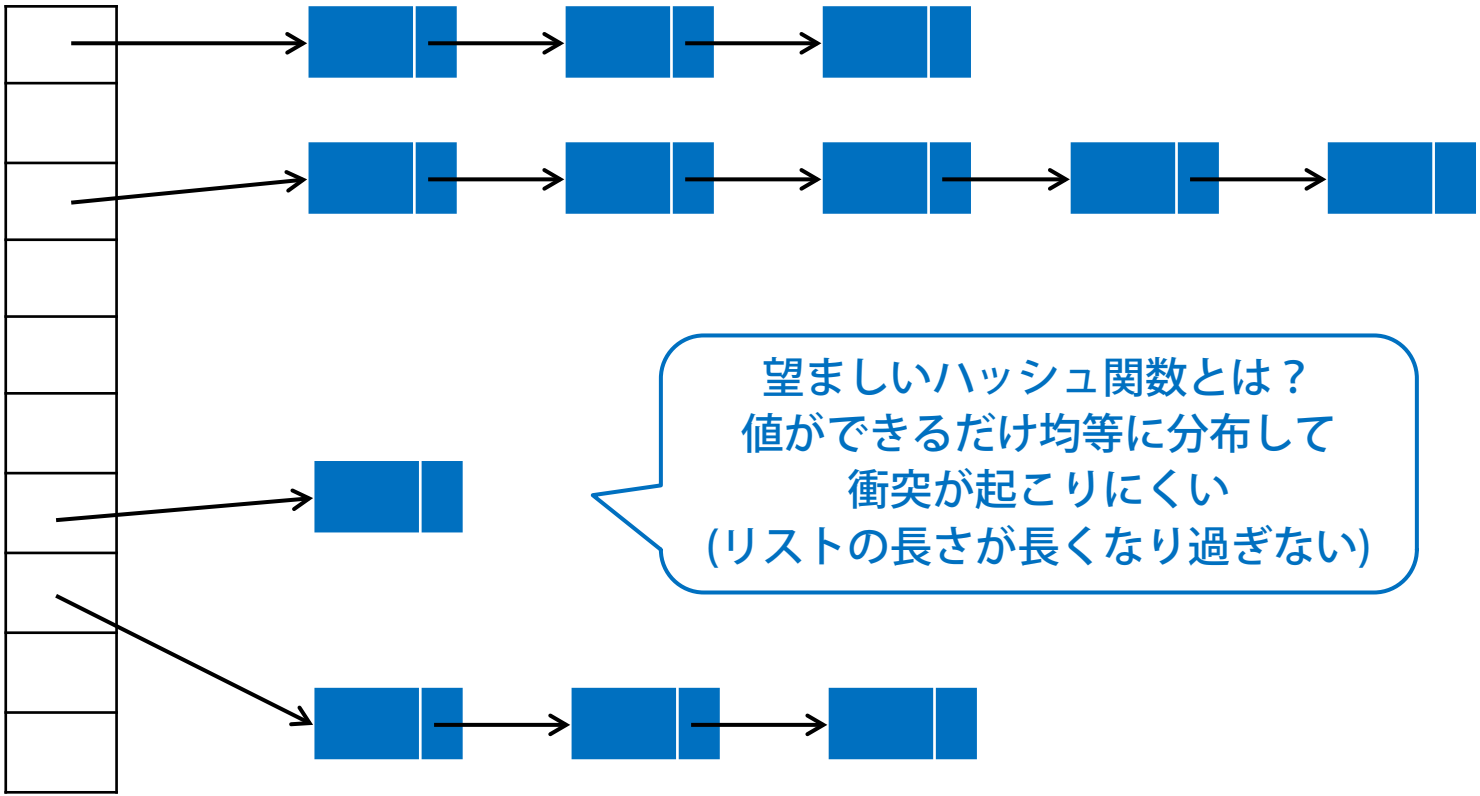
- 平均して$O(1)$の計算量で探索・挿入・削除を行える
  - 二分探索木のように比較に基づく探索では$O(\log n)$が限界
  - 例：キーの取りうる範囲が1~100までなら
    - 1~100の整列を添え字とする配列を準備
    - 探索：与えられた値を添え字として配列の要素を参照
  - キーの値が限定された範囲の整数でない (例：文字列) ときは？
    - キーの値を引数として”ハッシュ関数”の値を計算
    - この数値を添え字として配列の要素を参照

メモリ容量の制約のため配列の添え字として使える範囲は限られている
→ 異なるキーでも同じハッシュ関数の値を返すことがある (衝突)
→ ハッシュ法は衝突の回避方法によって分類できる

# チェイン法



- ハッシュ関数の値が同じになるレコードをリストで連結
  - ハッシュ表の各要素にはリストの先頭を指すポインタを格納



# チェイン法の計算量



- 占有率$\alpha = \frac{n}{m}$に依存 (データ数$n$・ハッシュ表のサイズ$m$)
  - 平均比較回数 (証明省略)
    - 探索が成功する場合：$1 + \frac{\alpha}{2}$
    - 探索が失敗する場合：$\alpha + e^{-\alpha}$
  - 直感的な理解
    - $n$個のデータを$m$個の箱に振り分けるからリストの長さの平均は$\alpha$
    - リスト探索の計算量は$\alpha$の程度
  - 実際上の注意
    - $n$に対して$m$を十分に大きくとっておけば計算量は$O(1)$

# 開番地法



- ハッシュ表に直接レコードを記入
  - ハッシュ値が示す番地を調べて
    - 未使用であればその場所に書き込む
    - すでに使用済みであれば別の番地を探す
    - 未使用の番地が見つかるまで繰り返す
  - どこの番地を調べるかで変種が存在
    - 線形走査法：添え字をひとつづつ増やして調べる
      - “クラスタ”問題が起きやすく実用上は性能が悪い
    - 均一ハッシュ法：ハッシュ関数を$m$個使用
    - 二重ハッシュ法：ハッシュ関数を$h, g$を準備
      - $h(x), h(x)+g(x), h(x)+2g(x), \cdots$の順番に調べる

隣り合う番地にレコードが
密集しやすい
→ 探索効率が悪い

# 開番地法の計算量



- 均一ハッシュ法の挿入の効率について解析
  - $n$個のデータを順次挿入していく過程を考える
  - $k$個の挿入が終わった段階で$k+1$個目を挿入したい
  - 未使用の場所を見つけるまでに調べる場所の数とその確率は
    - 1　$p_1 = \frac{m-k}{m}$
    - 2　$p_2 = \frac{k}{m} \times \frac{m-k}{m-1}$
    - 3　$p_3 = \frac{k}{m} \times \frac{k-1}{m-1} \times \frac{m-k}{m-2}$
    - $k+1$　$p_{k+1} = \frac{k}{m} \times \frac{k-1}{m-1} \times \cdots \times \frac{1}{m-k+1} \times \frac{m-k}{m-k}$

$$p_i = \frac{{}_{m-i}C_{k-i+1}}{{}_{m}C_{k}}$$

調べる場所の数の平均

$$\sum_{i=1}^{k+1} i\, p_i = \frac{m+1}{m-k+1} \approx \frac{1}{1-\alpha}$$

$m$と$k$がともに大きい場合

# 開番地法の計算量



- 均一ハッシュ法の探索の効率について解析
  - 探索が失敗する場合：$\frac{1}{1-\alpha}$ (データが表にないので挿入と同様)
  - 探索が成功する場合
    - $n$個のデータそれぞれを挿入するときの手間の平均と等しい
    - $k=0, k=1, \cdots, k=n-1$の時の挿入操作の平均

$$\frac{1}{n}\left(\frac{m+1}{m+1}+\frac{m+1}{m}+\cdots+\frac{m+1}{m-n+2}\right)=\frac{m+1}{n}(H_{m+1}-H_{m-n+1})$$

$m \to \infty$ のとき $H_m = \log_e m + \gamma$ で近似できるから $\approx \frac{1}{\alpha}\log_e \frac{1}{1-\alpha}$

線形走査法では探索成功：$\frac{1-\frac{\alpha}{2}}{1-\alpha}$　探索失敗：$\frac{1}{2}+\frac{1}{2}(1-\alpha)^2$

$\alpha$が1に近づくと急激に悪化

二重ハッシュ法は
わずか劣るが
(実際上問題なし)