算法数理工学
第5回

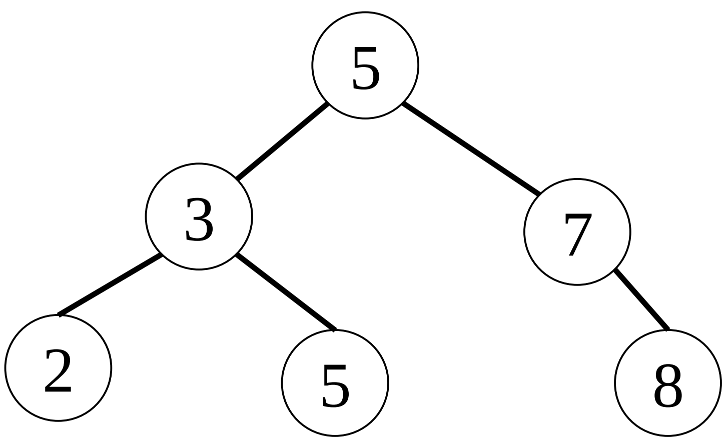
定兼　邦彦

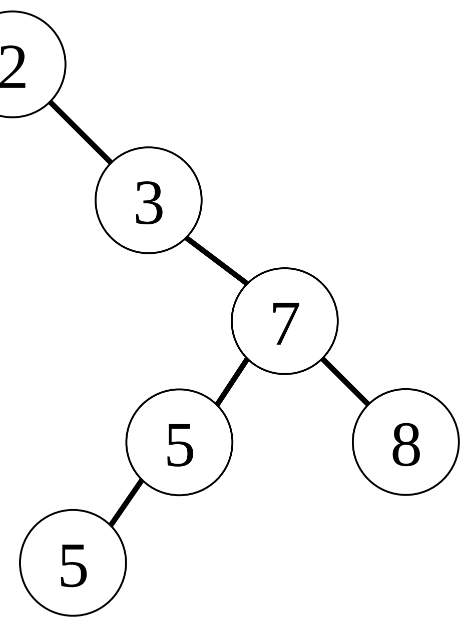
2分探索木

- 探索木: search, minimum, maximum, predecessor, successor, insert, delete等が利用できる動的集合用データ構造
- 辞書やプライオリティーキューとして利用できる
- 基本操作は木の高さに比例した時間がかかる
  - ランダムに構成された2分探索木の高さ: O(lg n)
  - 最悪時: O(n)
- 最悪時でもO(lg n) に改良できる (2色木)

2分探索木とは何か?

- 各節点は key, left, right, p フィールドを持つ
- 2分探索木条件 (binary-search-tree property)
  - 節点 y が x の左部分木に属する$\Rightarrow$ key(y)≤key(x)
  - 節点 y が x の右部分木に属する$\Rightarrow$ key(x)≤key(y)

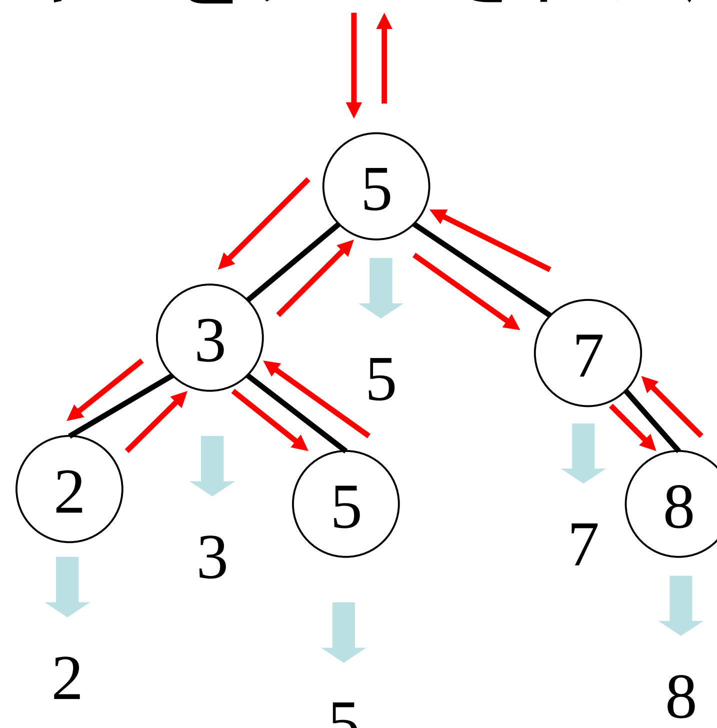




木の中間順巡回

- 木の中間順巡回 (通りがけ順, inorder tree walk)
  - 根の左部分木に出現するキー集合
  - 根のキー
  - 右部分木に出現するキー集合の順にキーを出力
- 木の根から辿ると，全てのキーをソートされた順序で出力できる
- $\Theta(n)$ 時間

```
inorder_tree_walk(node x)
{
    if (x != NIL) {
        inorder_tree_walk(left(x));
        print(key(x));
        inorder_tree_walk(right(x));
    }
}
```



その他の巡回法

- 先行順巡回 (行きがけ順, preorder tree walk):
根節点を先に出力し，次に左右の部分木を出力
- 後行順巡回 (帰りがけ順, postorder tree walk):
先に左右の部分木を出力し，最後に根節点を出力

```
preorder_tree_walk(node x)
{
    if (x != NIL) {
        print(key(x));
        preorder_tree_walk(left(x));
        preorder_tree_walk(right(x));
    }
}
```

```
postorder_tree_walk(node x)
{
    if (x != NIL) {
        postorder_tree_walk(left(x));
        postorder_tree_walk(right(x));
        print(key(x));
    }
}
```

2分探索木に対する質問操作

- 質問操作は高さに比例した時間で終了する
- 探索: 2分探索木の中から，ある与えられたキーを持つ節点のポインタを求める
  - 存在しなければNIL
  - 複数ある時はどれか一つ

```
tree_search(node x, data k)
{
    if (x == NIL || k == key(x)) return x;
    if (k < key(x)) return tree_search(left(x),k);
    else  return tree_search(right(x),k);
}
```

- 再帰はwhileループにすることができる

```
iterative_tree_search(node x, data k)
{
    while (x != NIL && k != key(x)) {
        if (k < key(x)) x = left(x);
        else  x = right(x);
    }
    return x;
}
```

探索の正当性

- キー k が見つかったら探索を終了する
- k が key(x) より小さい場合
  - 2分探索木条件より, k は x の右部分木にはない
  - 左部分木に対して探索を続行する
- k が key(x) より大きい場合
  - 右部分木に対して探索を続行する
- 探索する節点は根からのパスになる
  - 実行時間は O(h) (h: 木の高さ)

最小値と最大値

- 最小/最大のキーを持つ要素のポインタを返す
- O(h) 時間

```
tree_minimum(node x)
{
    while (left(x) != NIL) {
        x = left(x);
    }
    return x;
}
```

```
tree_maximum(node x)
{
    while (right(x) != NIL) {
        x = right(x);
    }
    return x;
}
```

次節点と前節点

- 2分探索木のある節点が与えられたとき，木の中間順 (inorder) で次/前の節点を求める
- O(*h*) 時間

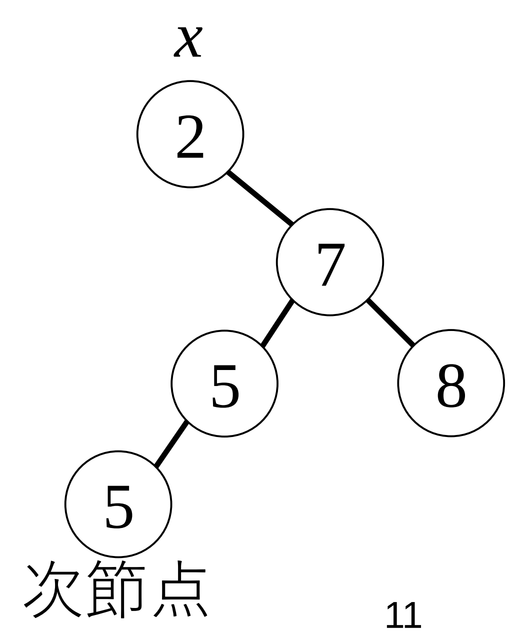
```
tree_successor(node x)
{
    node y;
    if (right(x) != NIL) return tree_minimum(right(x));
    y = p(x);
    while (y != NIL && x == right(y)) {
        x = y;
        y = p(y);
    }
    return y;
}
```

x が右部分木を持つ場合

- x の次節点は，x 以上の要素で最小

$\Rightarrow$ x の次節点は，x の右部分木の最小要素

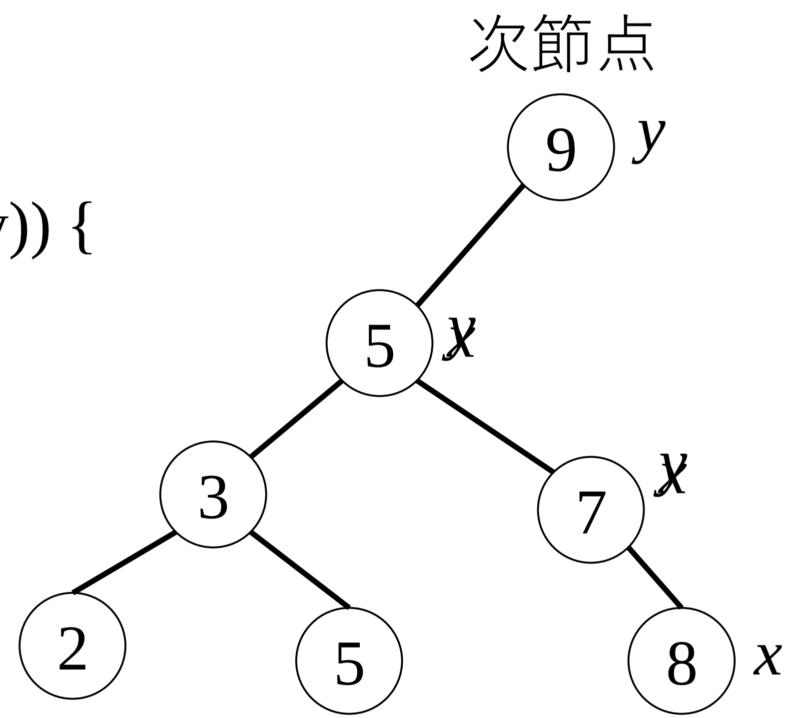
= tree_minimum(right(x))



x が右部分木を持たない場合

- x の親を y とする
- x が, x の親の右の子ならば, 親は x 以下
- y は, x を左部分木に持つ x の祖先で最も x に近いもの

```
y = p(x);
while (y != NIL && x == right(y)) {
    x = y;
    y = p(y);
}
return y;
```

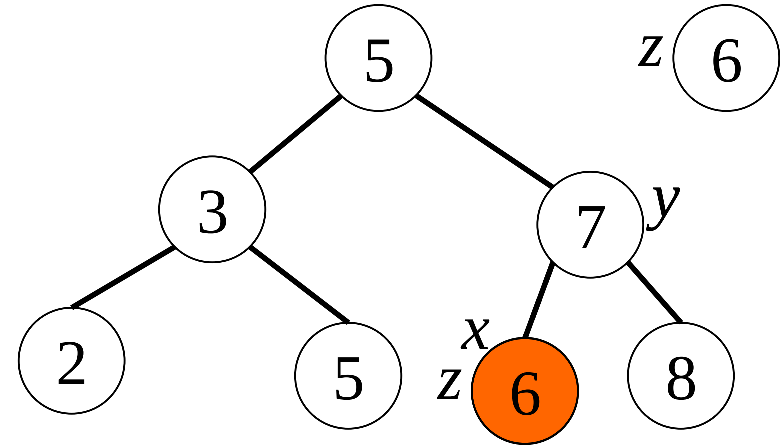


定理 高さ h の2分探索木上の動的集合演算 search, maximum, minimum, successor, predecessor は $O(h)$ 時間で実行できる.

挿入と削除

- 要素を挿入/削除したあとも2分探索木条件が満たされる必要がある
- 挿入は比較的簡単
- 削除は複雑
- どちらも O(h) 時間

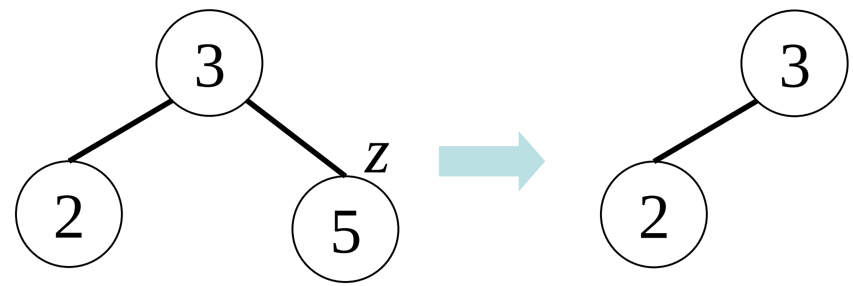
挿入



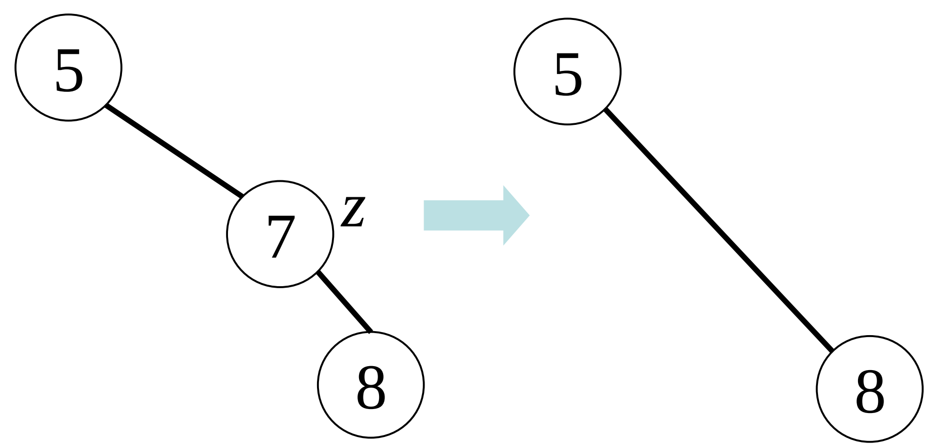

```
tree_insert(tree T, node z)
{
    node x,y;
    y = NIL;
    x = root(T);
    while (x != NIL) {                    // z を挿入する場所 x を決める
        y = x;
        if (key(z) < key(x)) x = left(x);        挿入場所は必ず葉
        else x = right(x);
    }                                     // y は x の親
    p(z) = y;                             // z の親を y にする
    if (y == NIL) root(T) = z;            // T が空なら z が根節点
    else if (key(z) < key(y)) left(y) = z; // y の子を z にする
        else right(y) = z;
}
```

削除

- 探索木から節点 z を削除する
- z が子を持たない場合
  - z の親 $p(z)$ を変更する

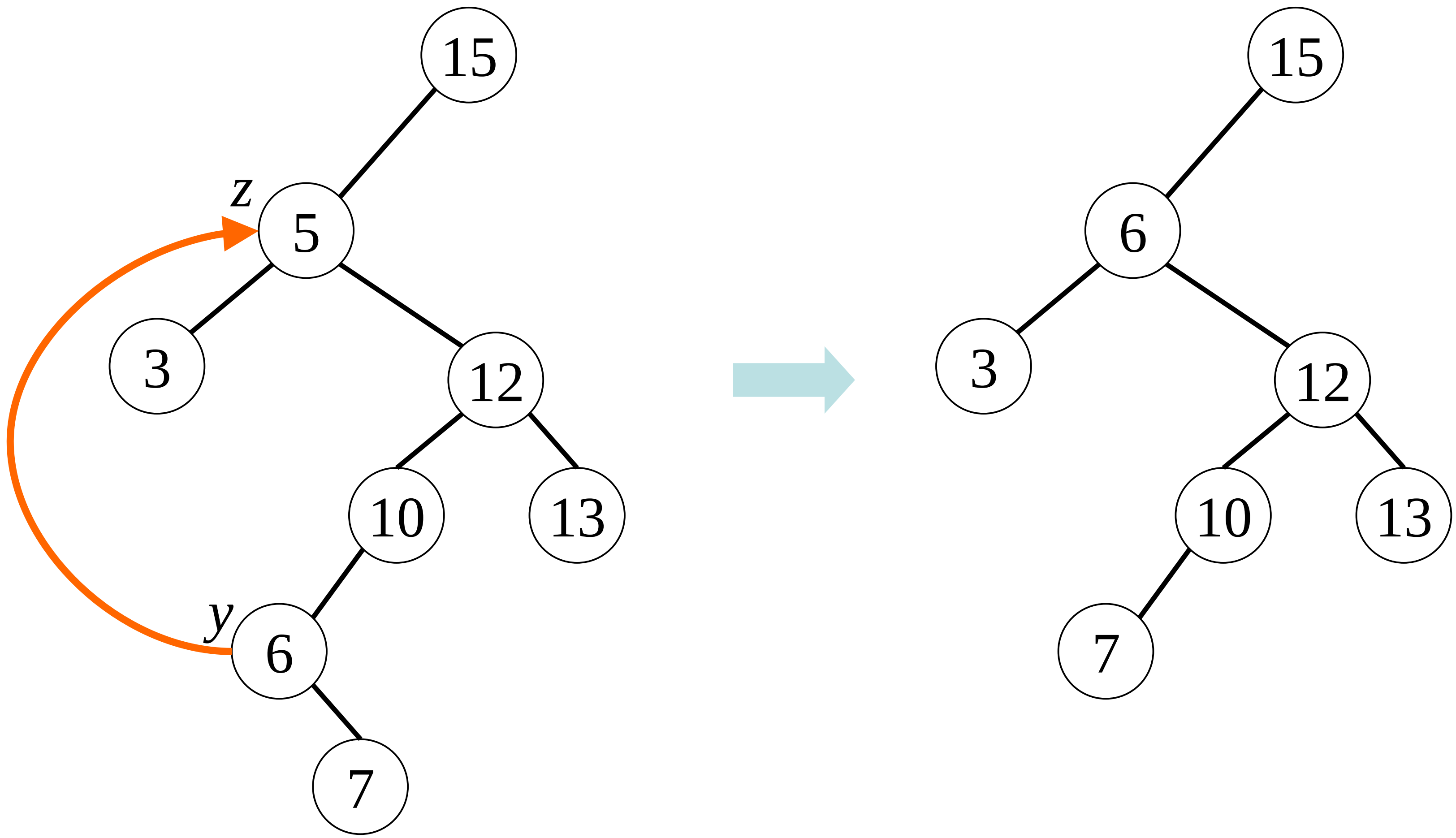


- z が子を1つ持つ場合
  - z の親と z の子を結ぶ



削除: z が子を2つ持つ場合

- z の次節点は左の子を持たない
- z の場所に y を入れ，元の y を削除する



```
tree_delete(tree T, node z)
{
   node x, y;
   if (left(z) == NIL || right(z) == NIL) y = z;  // z の子の数が1以下
   else y = tree_successor(z);                 // z は2つの子を持つ
   if (left(y) != NIL) x = left(y);  else x = right(y); //  x は y の子
   if (x != NIL) p(x) = p(y);                          // y を削除する
   if (p(y) == NIL) root(T) = x;                      // y が根なら x を根に
   else if (y == left(p(y))) left(p(y)) = x;           // y の親と子をつなぐ
        else right(p(y)) = x;
   if (y != z) {
      key(z) = key(y);                                // y の内容を z に移動
      // y の付属データを z にコピー
   }
   return y;                                         // 不必要な y を回収
}
```

ランダムに構成された2分探索木

- 2分探索木上の基本操作は O(h) 時間で実行可
- 要素の挿入削除を繰り返すと探索木の高さ h は変化する
- 2分探索木の高さは最悪 n
  - キーを小さい順/大きい順に挿入した場合
- n 個の相異なるキーをランダムな順序で挿入した2分探索木の高さを解析する
- 高さの期待値は O(lg n)

- 仮定
  - 入力キーの $n!$ 種類の順列がどれも等確率で出現する
  - 全てのキーは異なる
- 確率変数の定義
  - X_n: n 個のキー上のランダムに構成された2分探索木の高さ
  - $Y_n = 2^{X_n}$ 指数高さ
  - R_n: n 個のキーの中での根のキーの順位 (rank)
    $R_n = i$ なら, 根の左部分木は $i-1$ 個のキー上のランダムに構成された2分探索木, 右部分木は $n-i$ 個のキー上のランダムに構成された2分探索木

- 2分探索木の高さは根の左右の部分木の高さの大きいほう+1. $R_n = i$ のとき

$$X_n = \max\{X_{i-1}, X_{n-i}\} + 1$$
$$Y_n = 2\max\{Y_{i-1}, Y_{n-i}\}$$

- $Y_1 = 1$. 便宜上 $Y_0 = 0$ と定義する.
- 指標確率変数 $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$ を次のように定義

$$Z_{n,i} = \mathrm{I}\{R_n = i\} = \begin{cases} 1 & (R_n = i) \\ 0 & (\text{それ以外}) \end{cases}$$

- R_n の値は $\{1,2,\ldots,n\}$ の任意の要素を等確率で取るから, $\Pr\{R_n = i\} = 1/n$ $(i=1,2,\ldots,n)$ よって

$$\mathrm{E}[Z_{n,i}] = \frac{1}{n}$$

- 根の順位が i のときだけ $Z_{n,i} = 1$ になるから

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \left(2 \max\{Y_{i-1}, Y_{n-i}\}\right)$$

- $\mathrm{E}[Y_n]$ が多項式であることが示せれば, $\mathrm{E}[X_n]$ が $\mathrm{O}(\lg n)$ であることが示せる.
- $Z_{n,i}$ は Y_{i-1} とも Y_{n-i} とも独立
  – 根の左部分木内の各要素は根よりも小さい, つまり順位が i より小さい $i-1$ 個のキー上のランダムに構成された木である.
  – この部分木は順位の制約がない$i-1$ 個のキー上のランダムな木と同じ
  – 部分木の高さ Y_{i-1} と根の順位 $Z_{n,i}$ は独立

$$
\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\left(2\max\{Y_{i-1}, Y_{n-i}\}\right)\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[Z_{n,i}\left(2\max\{Y_{i-1}, Y_{n-i}\}\right)\right] && \text{(期待値の線形性)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}\left[2\max\{Y_{i-1}, Y_{n-i}\}\right] && \text{(独立性)} \\
&= \frac{2}{n}\sum_{i=1}^{n} \mathrm{E}\left[\max\{Y_{i-1}, Y_{n-i}\}\right] \\
&\le \frac{2}{n}\sum_{i=1}^{n} \left(\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]\right) \\
&= \frac{4}{n}\sum_{i=0}^{n-1} \mathrm{E}[Y_i]
\end{aligned}
$$

- $\mathrm{E}\left[Y_n\right] \le \frac{1}{4}\binom{n+3}{3}$ を示す

$$0 = Y_0 = \mathrm{E}\left[Y_0\right] \le \frac{1}{4}\binom{3}{3} = \frac{1}{4} \qquad 1 = Y_1 = \mathrm{E}\left[Y_1\right] \le \frac{1}{4}\binom{1+3}{3} = 1$$

$$\mathrm{E}\left[Y_n\right] \le \frac{4}{n}\sum_{u=0}^{n-1}\mathrm{E}\left[Y_i\right]$$

$$\le \frac{4}{n}\sum_{u=0}^{n-1}\frac{1}{4}\binom{i+3}{3}$$

$$= \frac{1}{n}\binom{n+3}{4}$$

$$= \frac{1}{4}\binom{n+3}{3}$$

- 関数 $f(x) = 2^x$ は下に凸であるから

$$2^{\mathrm{E}[X_n]} \le \mathrm{E}[2^{X_n}] = \mathrm{E}[Y_n]$$

- 以上より

$$\mathrm{E}[X_n] \le \log \mathrm{E}[Y_n] = \log \frac{1}{4}\binom{n+3}{3} = \mathrm{O}(\log n)$$

- 定理 n 個のキー上のランダムに構成された2分探索木の高さの期待値は $\mathrm{O}(\lg n)$.

2色木

- 2分探索木は，基本的な動的集合操作を木の高さに比例する時間で実現できる
- 探索木の高さは要素の挿入順に依存し，最悪の場合は O(n) になる
- 2色木は，基本操作が最悪でも O(lg n)時間になるような探索木の1つである

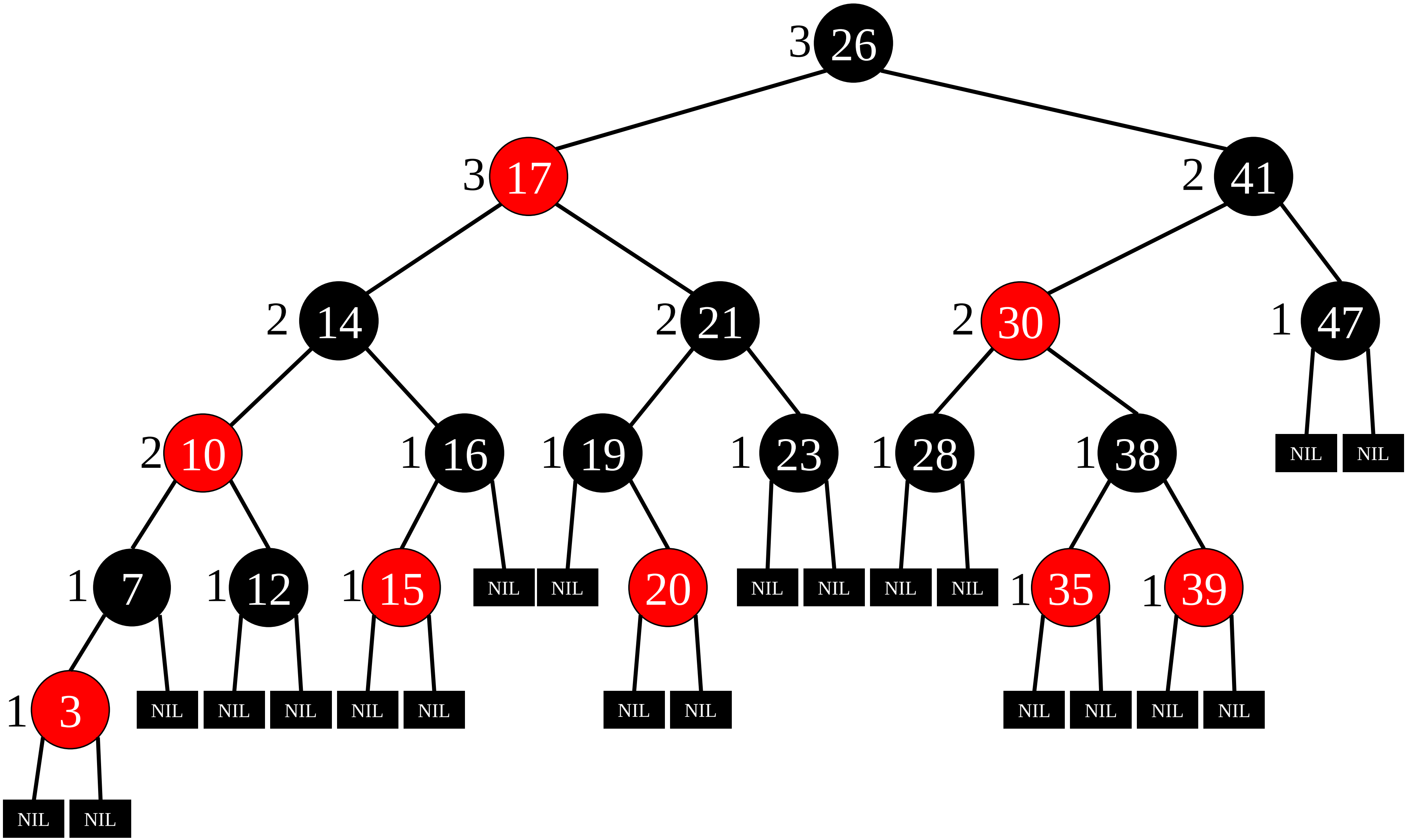
2色木条件

- 各節点に1ビットの情報(色)を加えた2分探索木
- 各節点は赤または黒の色がついている
- 各節点の要素
  - color, key, left, right, p
- 外部節点 (葉) は NIL で表される
- 内部節点のみが key を持つ

2色木条件 (Red-Black Property)

2分探索木が下記の2色木条件を満たすならば, 2色木である.

1. 各節点は赤か黒のどちらか
2. 葉 (NIL) は全て黒
3. もしある節点が赤ならば, その子供は両方黒
4. 1つの節点からその子孫の葉までのどの単純な経路も, 同じ数だけ黒節点を含む.



ある節点 x から葉までの経路上の黒節点の数

(x は含まない) を黒高さと呼び，bh(x) で表す

補題: n 個の内点をもつ2色木の高さは高々 $2\lg(n+1)$ である

証明: まず，任意の節点 x を根とする部分木は少なくとも $2^{\mathrm{bh}(x)}-1$ 個の内点を含むことを示す．

x の高さが 0 のとき，x は葉で，x を根とする部分木は少なくとも $2^{\mathrm{bh}(x)}-1=2^0-1=0$ 個の内点を含む．

高さ $h\geq 0$ 以下の全ての木で成り立つとする．高さ $h+1$ の木の根 x は2つの子供を持ち，それぞれ $\mathrm{bh}(x)$ または $\mathrm{bh}(x)-1$ の黒高さを持つ．各子供は少なくとも $2^{\mathrm{bh}(x)-1}-1$ 個の内点を持つため，x は少なくとも $(2^{\mathrm{bh}(x)-1}-1)+(2^{\mathrm{bh}(x)-1}-1)+1=2^{\mathrm{bh}(x)}-1$ 個の内点を含む．よって高さ $h+1$ の木でも成り立つ

証明の続き:

木の高さを h とする．条件3より，根から葉までのどの経路上の根以外の節点の少なくとも半分は黒．よって，根の黒高さは少なくとも $h/2$.

上の命題より $n \geq 2^{h/2}-1$，つまり $h \leq 2 \lg (n+1)$.

この補題より，search, minimum, maximum, successor, predecessor はO(h) = O(lg n)時間で終わることがわかる

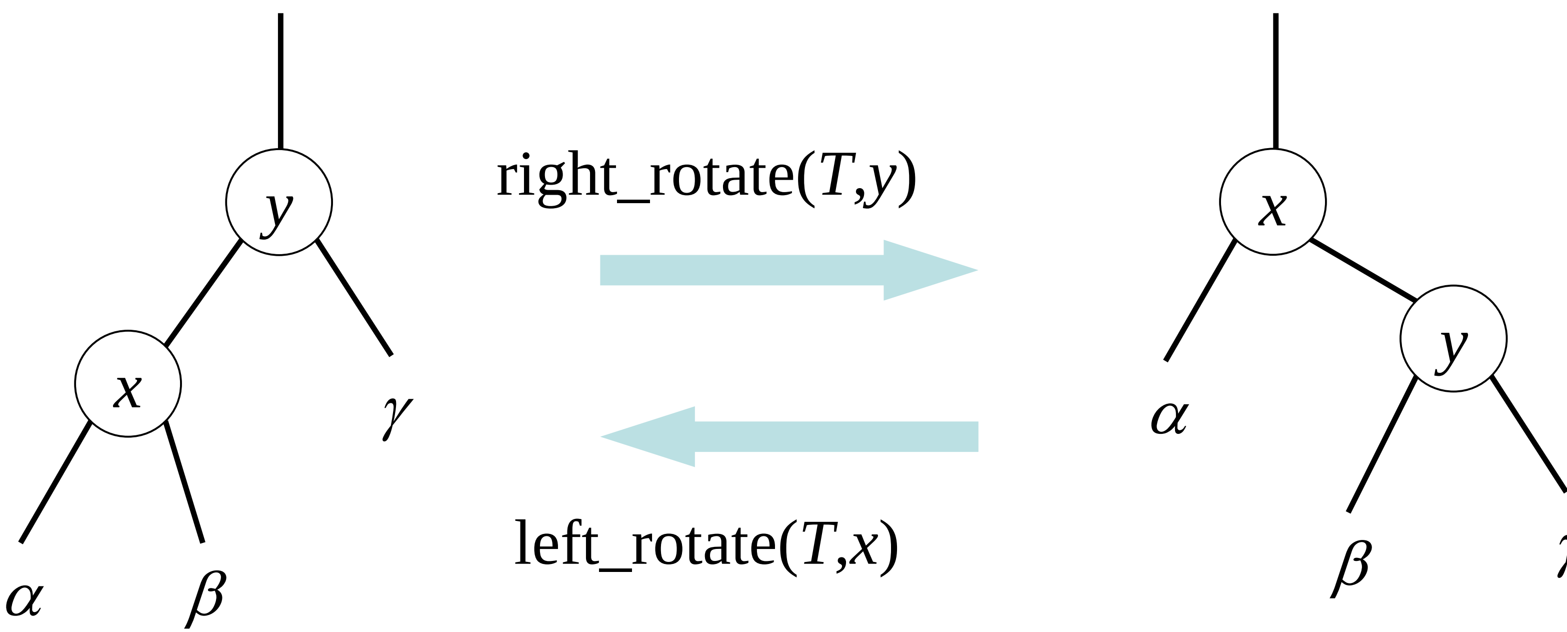
insert, delete は2色木条件を壊すため，アルゴリズムを変更する必要あり

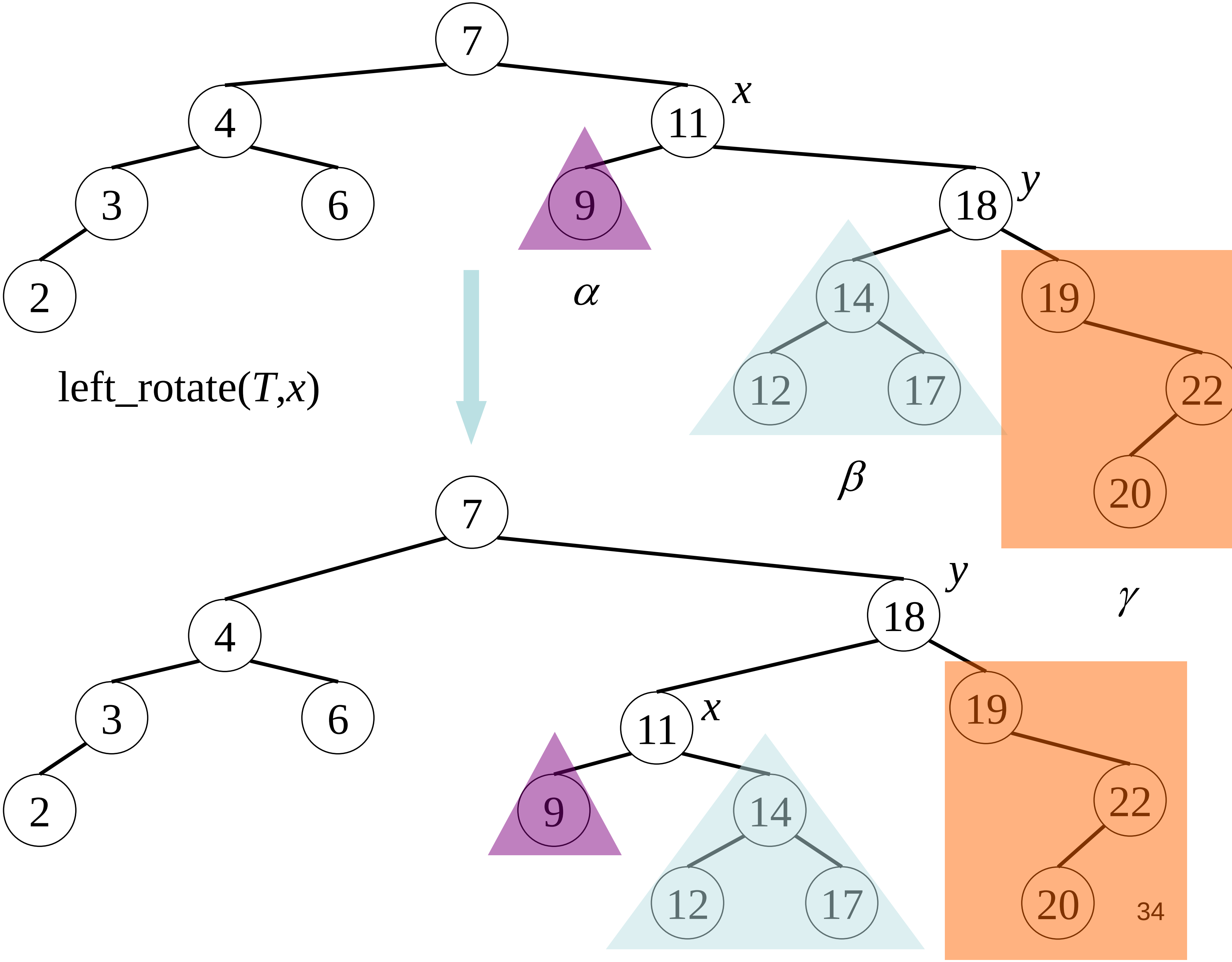
補題より，search, minimum, maximum, successor, predecessor は $O(h) = O(\lg n)$時間で終わることがわかる.

insert, deleteは2色木条件を壊すため，
アルゴリズムを変更する必要あり

回転

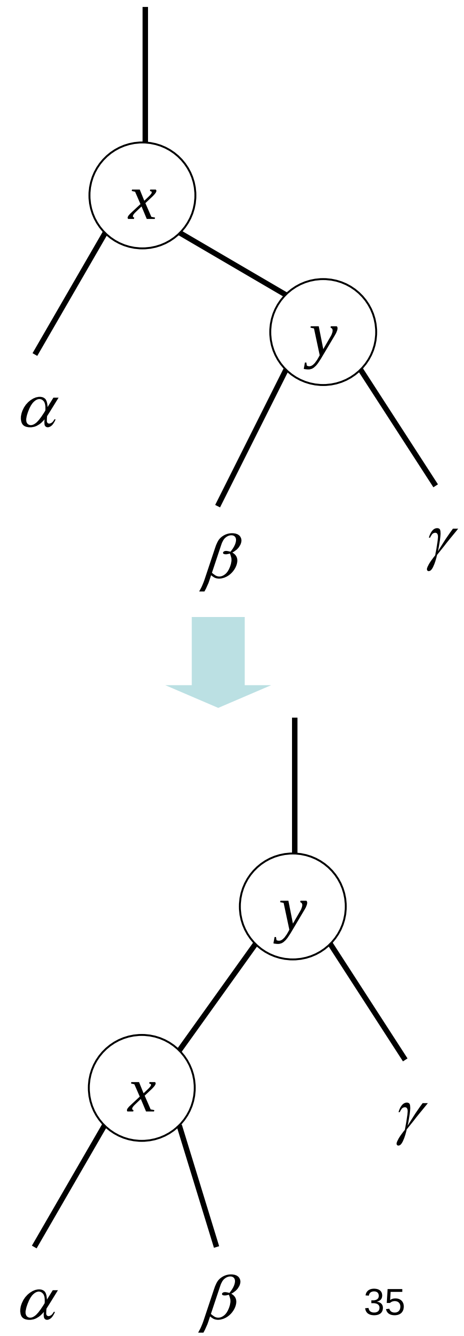
- 2色木で節点を追加/削除すると2色木の条件を満たさなくなることがある.
- 条件を満たすように木の構造を変更する
- $\alpha < x < \beta < y < \gamma$ の順を保つ




7
4
11
x
3
6
9
18
y
2
α
14
19
12
17
22
20
β
left_rotate(T,x)
7
4
18
y
γ
3
6
11
x
19
2
9
14
22
12
17
20

```
left_rotate(tree T, node x)
{
    node y;
    y = right(x);
    right(x) = left(y);
    if (left(y) != NIL) p(left(y)) = x;
    p(y) = p(x);
    if (p(x) == NIL) {
        root(T) = y;
    } else {
        if (x == left(p(x)) left(p(x)) = y;
        else right(p(x)) = y;
    }
    left(y) = x;
    p(x) = y;
}
```

O(1) 時間



新しい節点の挿入

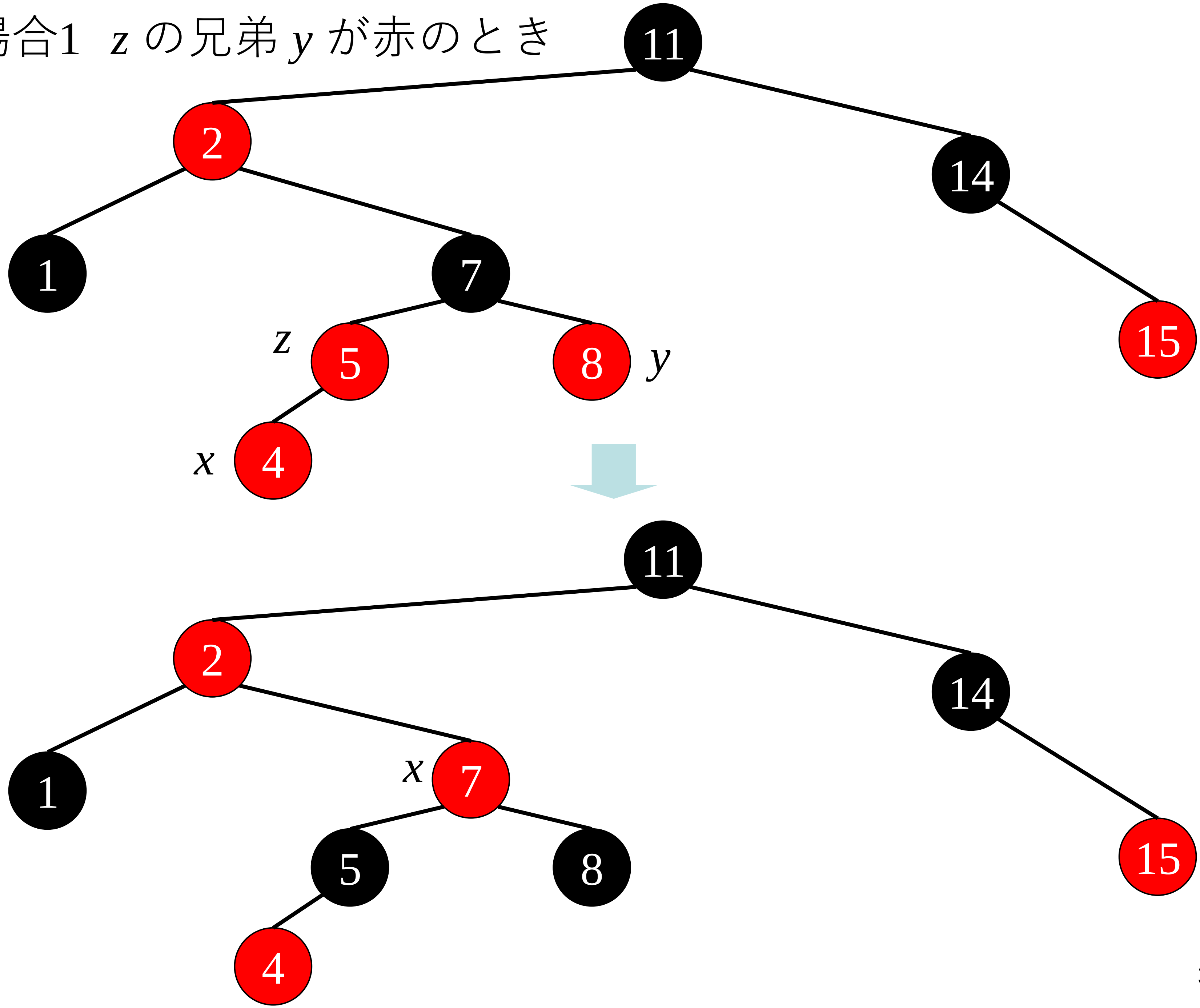
- keyに従って節点 x を葉に挿入する
- x の色を赤にする
- x を挿入後の2色木条件
  1. 各節点は赤か黒のどちらか…OK
  2. 葉 (NIL) は全て黒…OK
  3. もしある節点が赤ならば，その子供は両方黒…?
  4. 1つの節点からその子孫の葉までのどの単純な経路も，同じ数だけ黒節点を含む…OK

x の親が赤のときは条件3を満たさない⇒回転操作

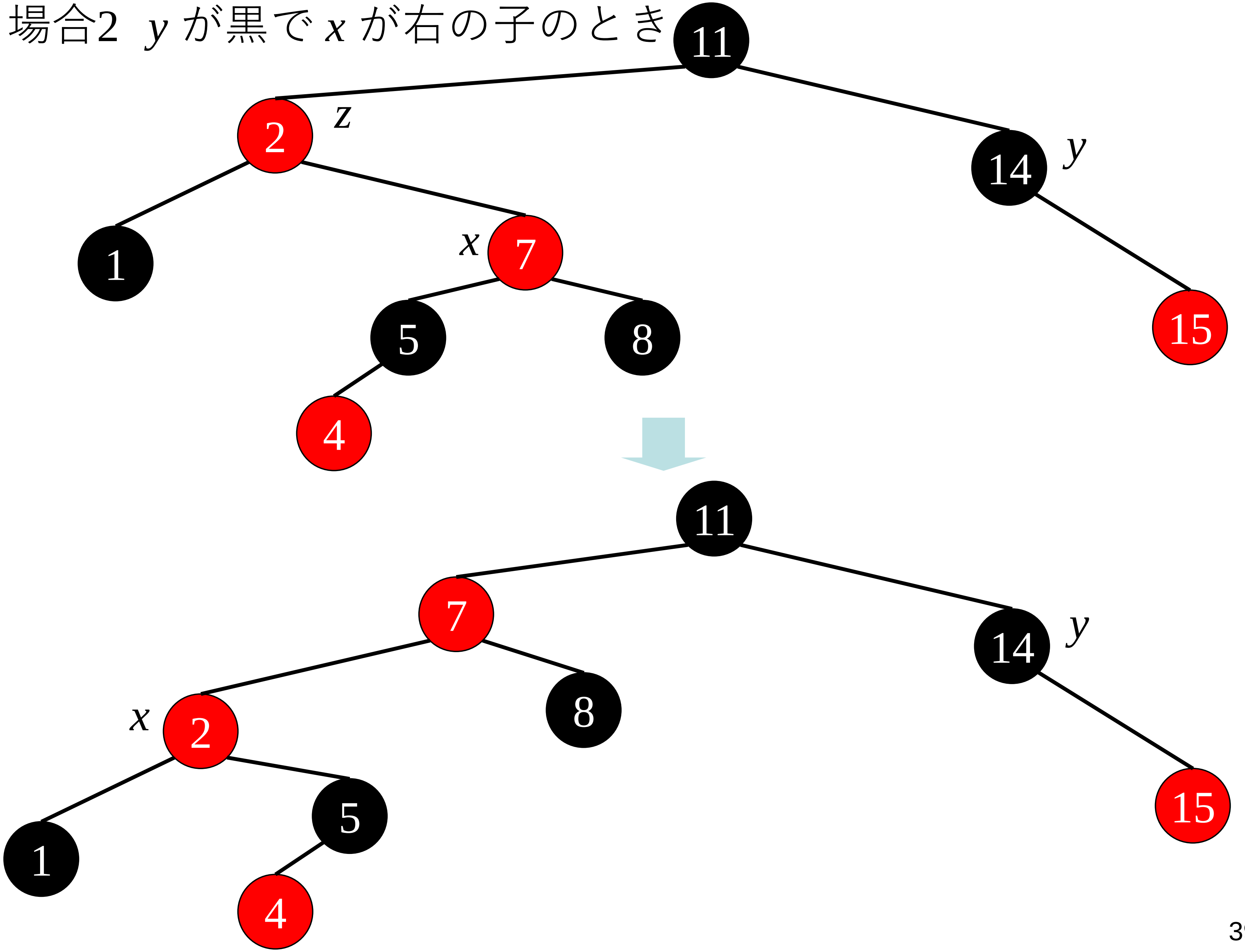
挿入後の回転操作

- x の親 z が赤である間以下を繰り返す
- 場合1: z の兄弟 y が赤のとき
  - z と y を黒, x をそれらの親とし, 赤にする
- 場合2: y が黒で x が右の子のとき
  - 左回転→場合3
- 場合3: y が黒で x が左の子のとき
  - x の親を黒, その親を赤にして右回転

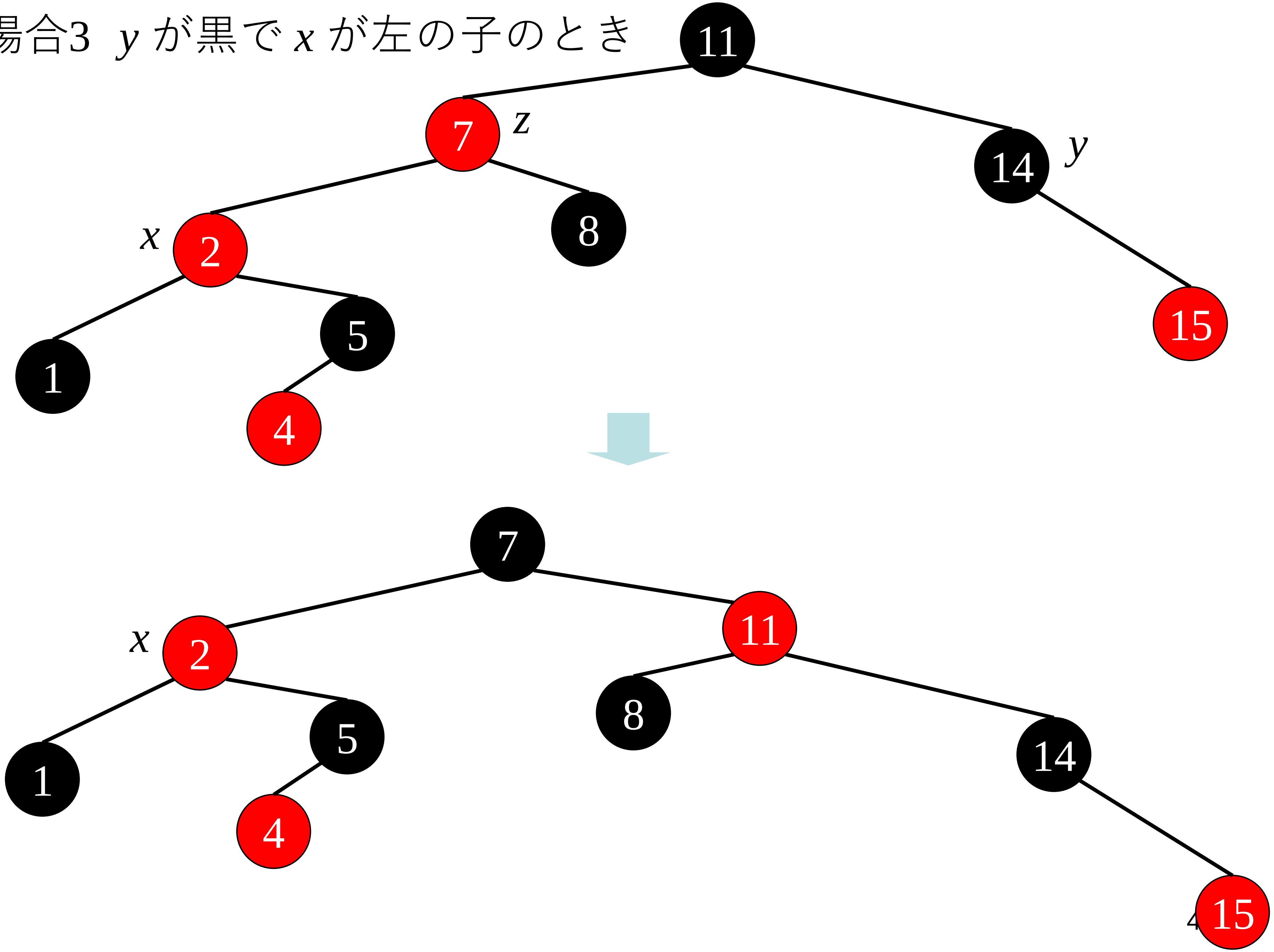
場合1 z の兄弟 y が赤のとき



場合2 y が黒で x が右の子のとき

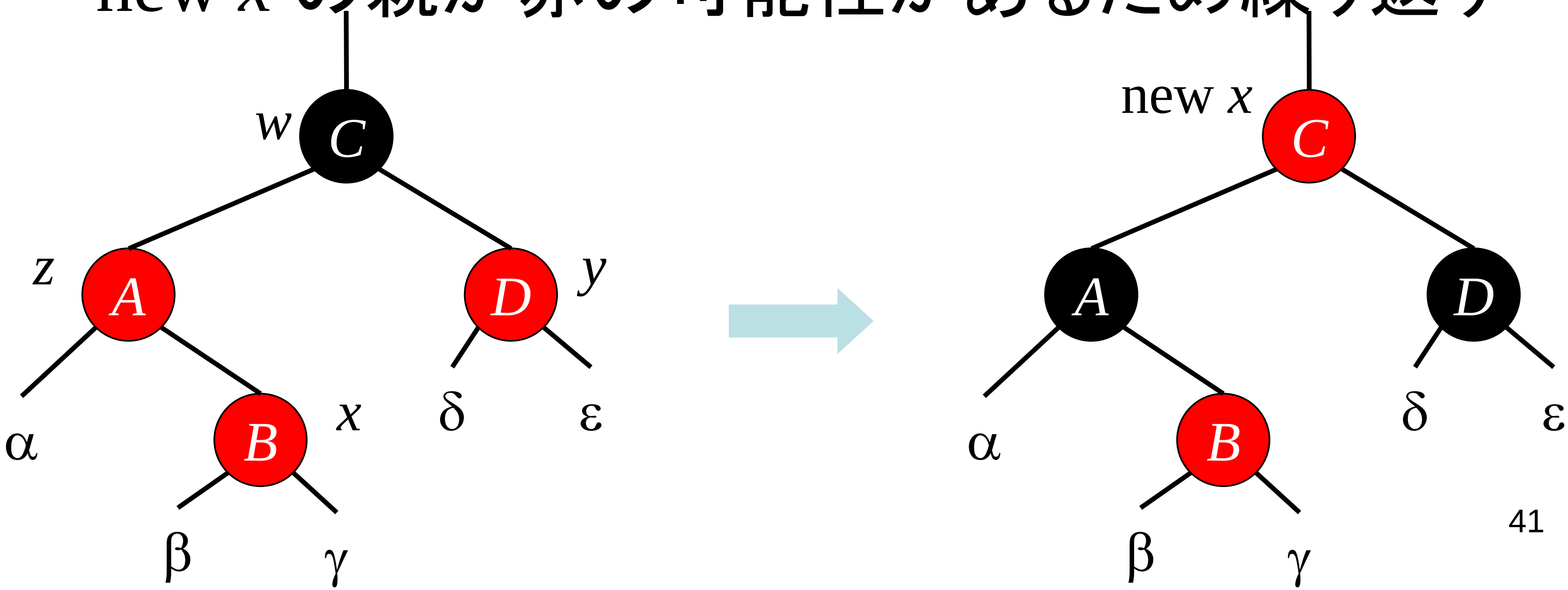


場合3　y が黒で x が左の子のとき



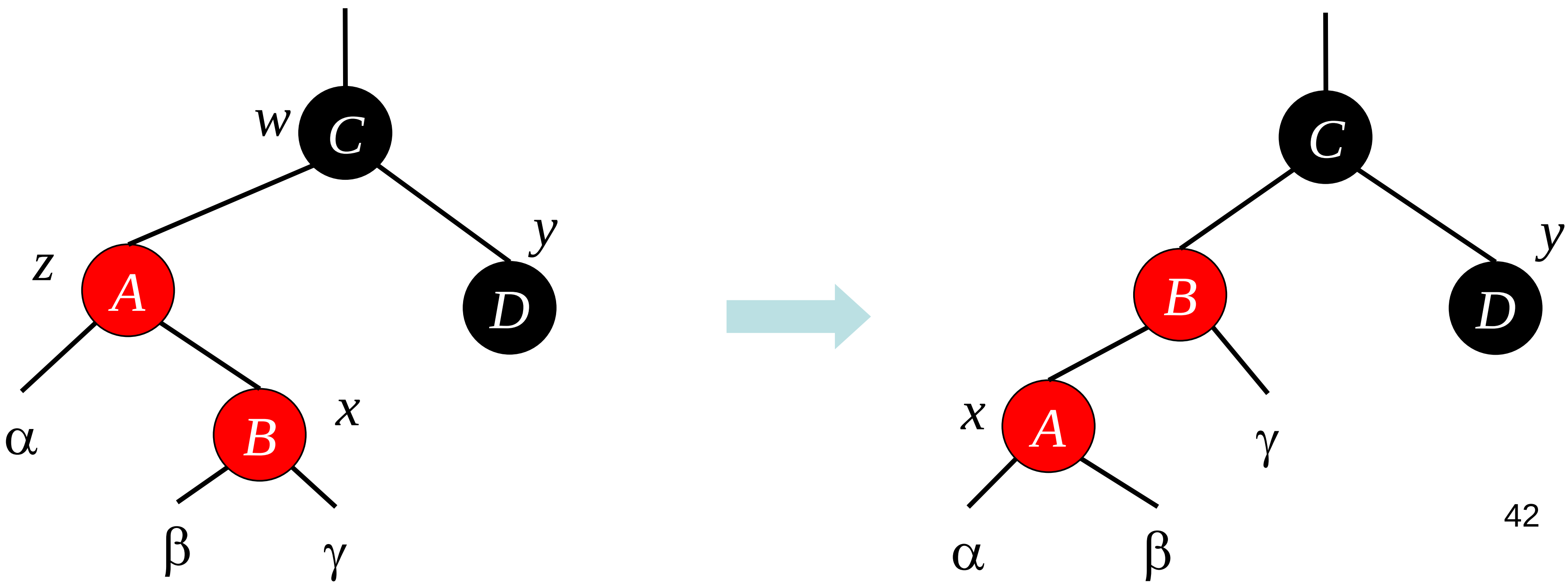
場合1: z の兄弟 y が赤のとき

- z の親 w は黒 (元の木では赤は連続しない)
- y, z の子孫の節点の黒深さは変化しない
- w の黒深さは1増える
- w の祖先の黒深さは変化しない
- new x の親が赤の可能性があるため繰り返す



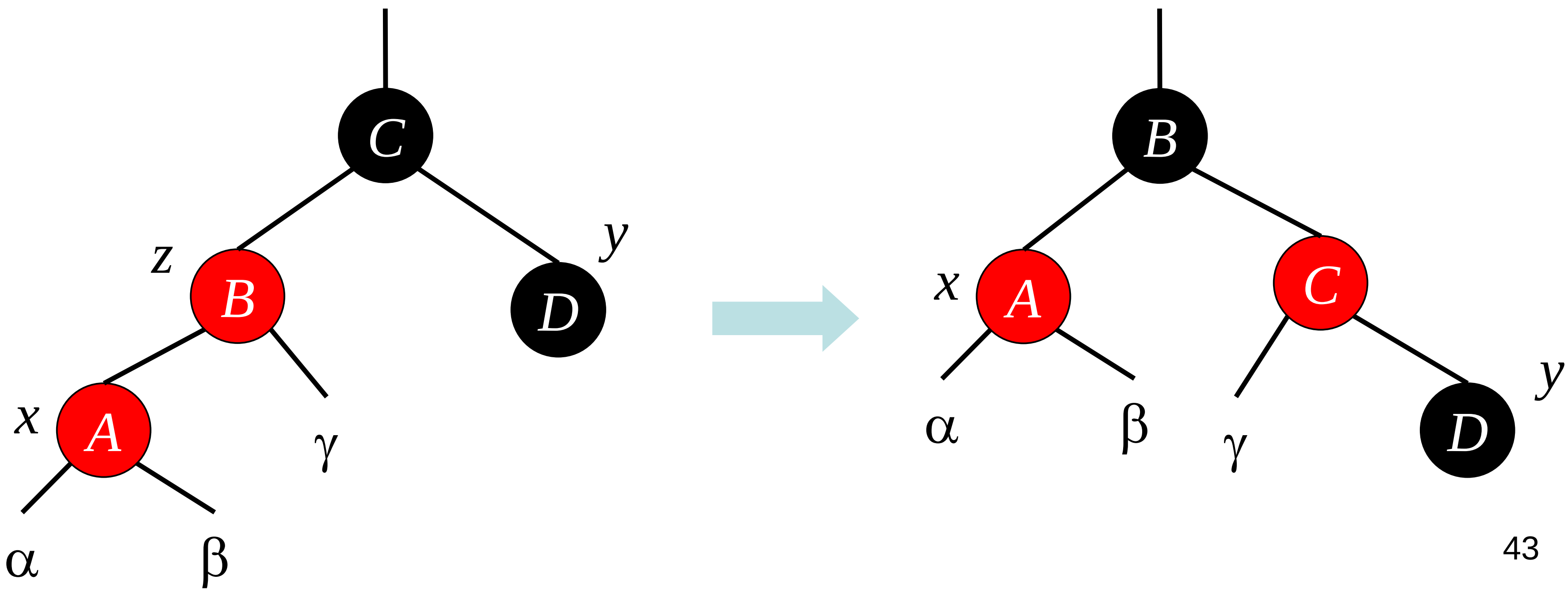
場合2: y が黒で x が右の子のとき

- z で左回転を行う⇒ x は左の子になる
- x, z ともに赤であるため，条件 3, 4 は満たされる
- x, z の子孫の黒高さも変化しない
- 場合3に移る



場合3: y が黒で x が左の子のとき

- p(p(x)) で右回転を行う
- 各部分木で黒高さは保存される
- 赤節点が連続することはない⇒終了



計算量

- 2色木の高さは O(lg n)
- tree_insertは O(lg n) 時間
- rb_insertでのwhileループでは x のポインタは木を登っていく
- ループの実行回数は木の高さ以下⇒ O(lg n)
- ループ内の処理は定数時間
- 全体でも O(lg n) 時間，高々2回の回転

注: 次回は 11月8日(木) 8:30