算法数理工学
第6回

定兼　邦彦

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

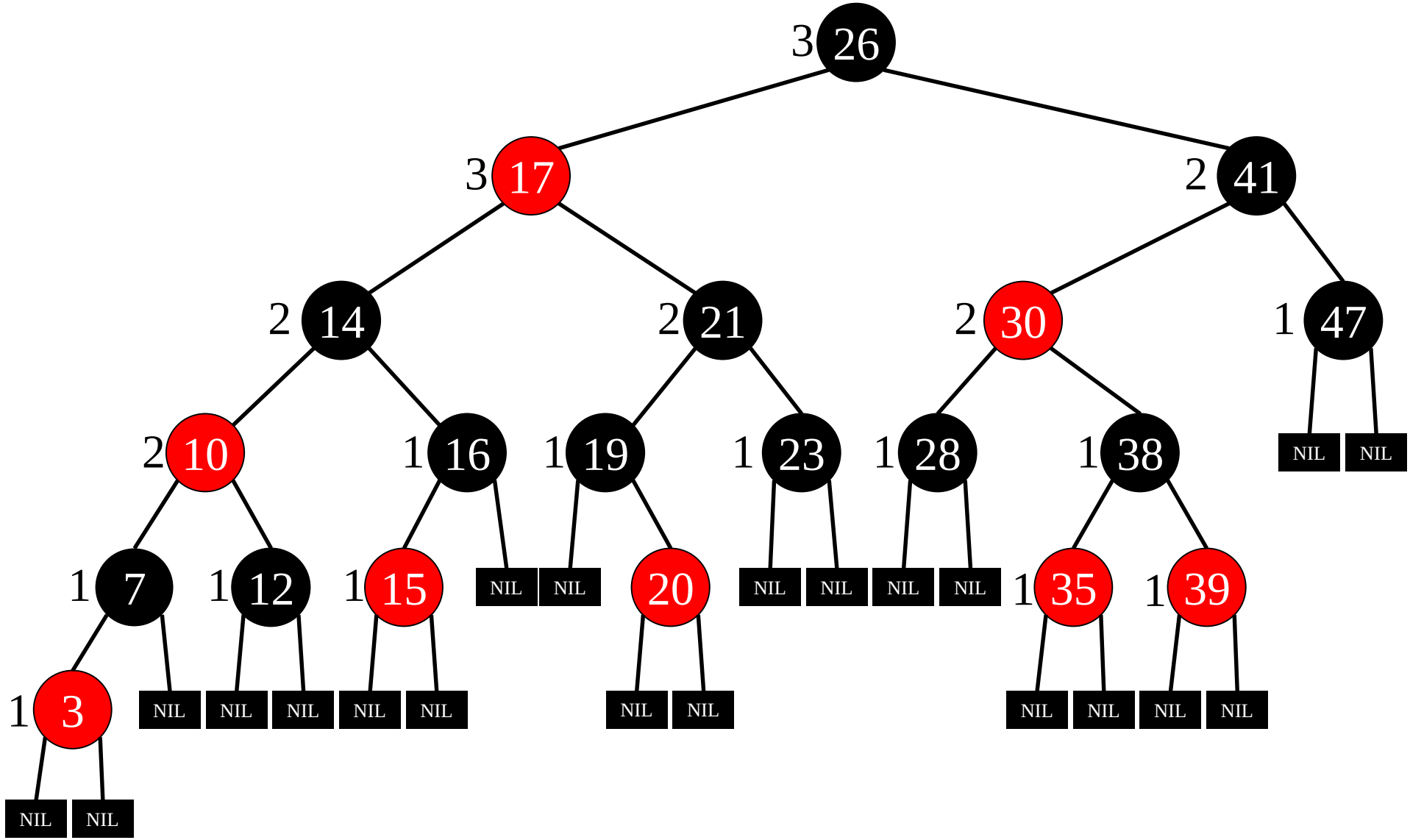
2色木条件

- 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，$\mathrm{bh}(x)$ 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 $h \leq 2 \lg (n+1)$.

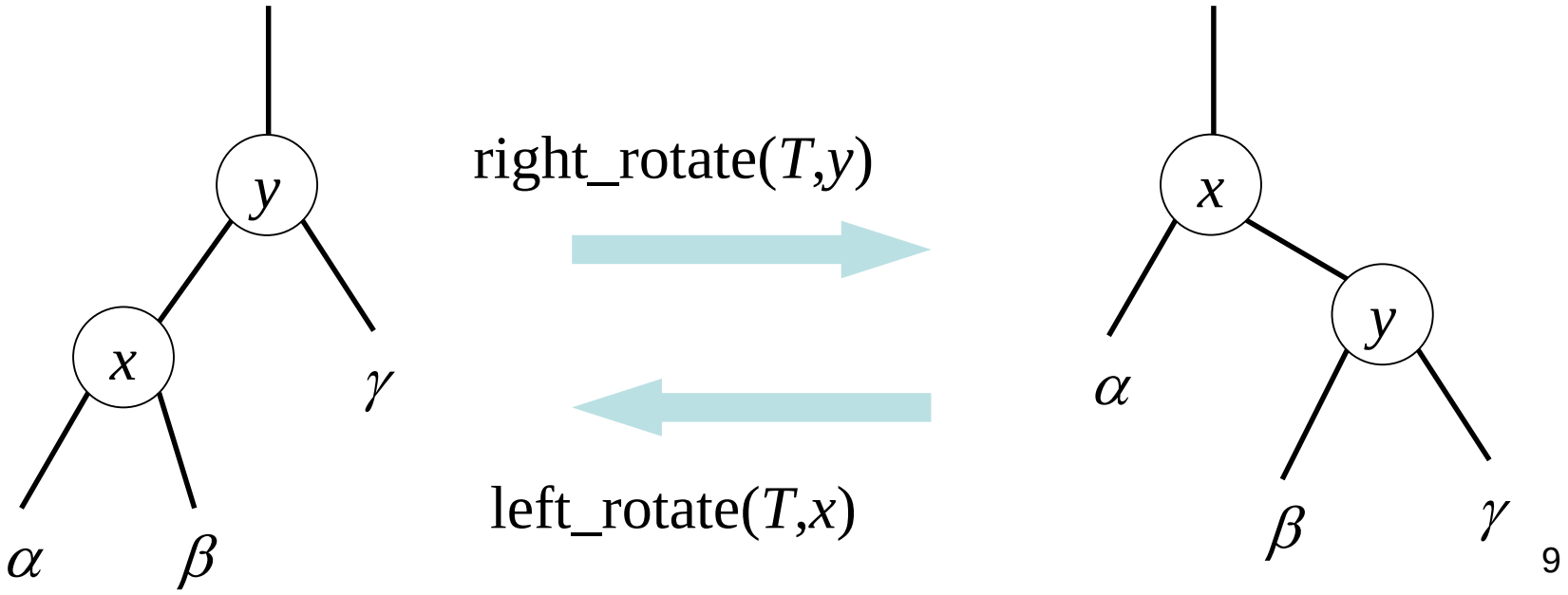
この補題より, search, minimum, maximum,
successor, predecessor は$O(h) = O(\lg n)$時間で
終わることがわかる
insert, delete は2色木条件を壊すため, アルゴリズム
を変更する必要あり

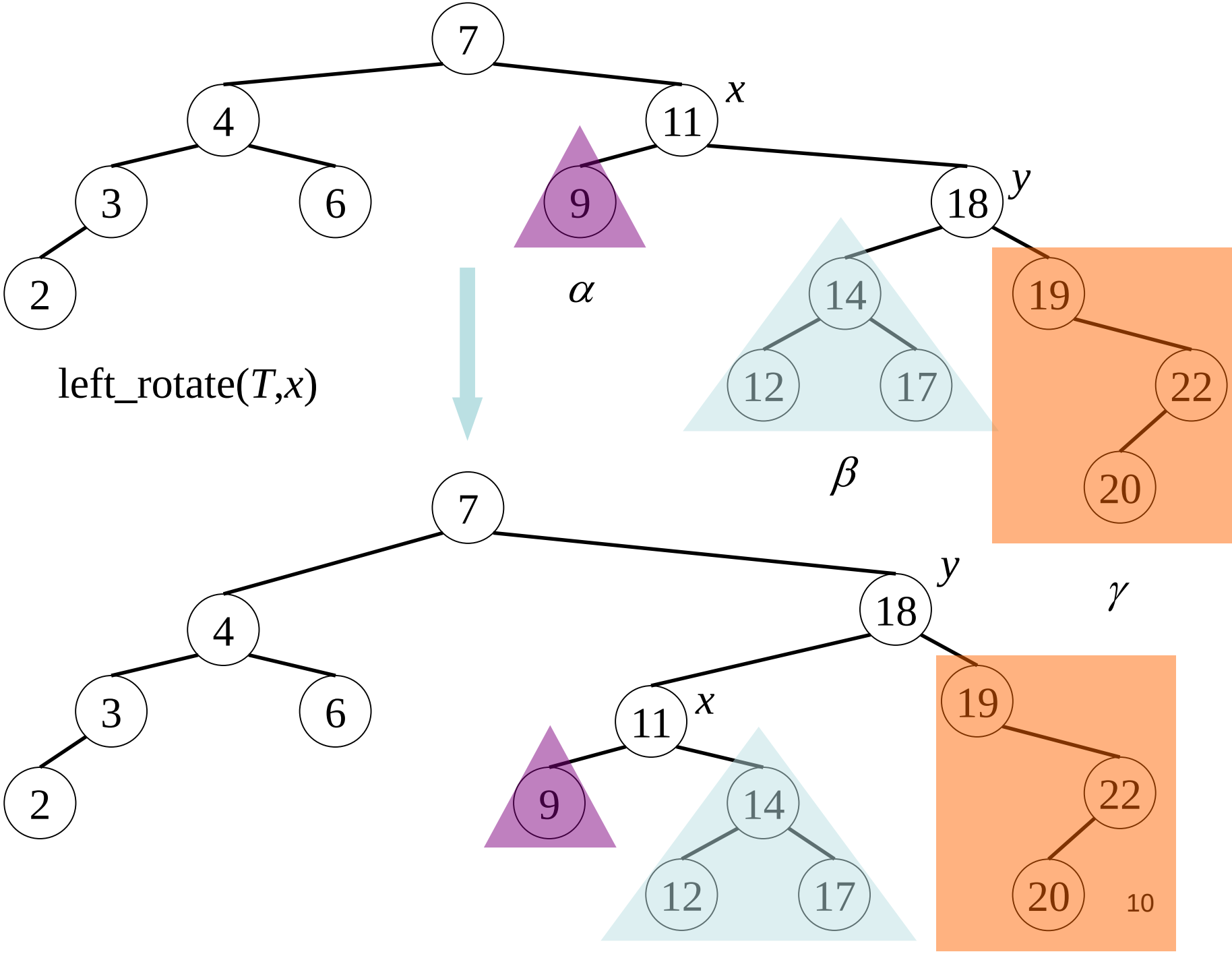
補題より，search, minimum, maximum, successor, predecessor は $O(h) = O(\lg n)$時間で終わることがわかる.

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

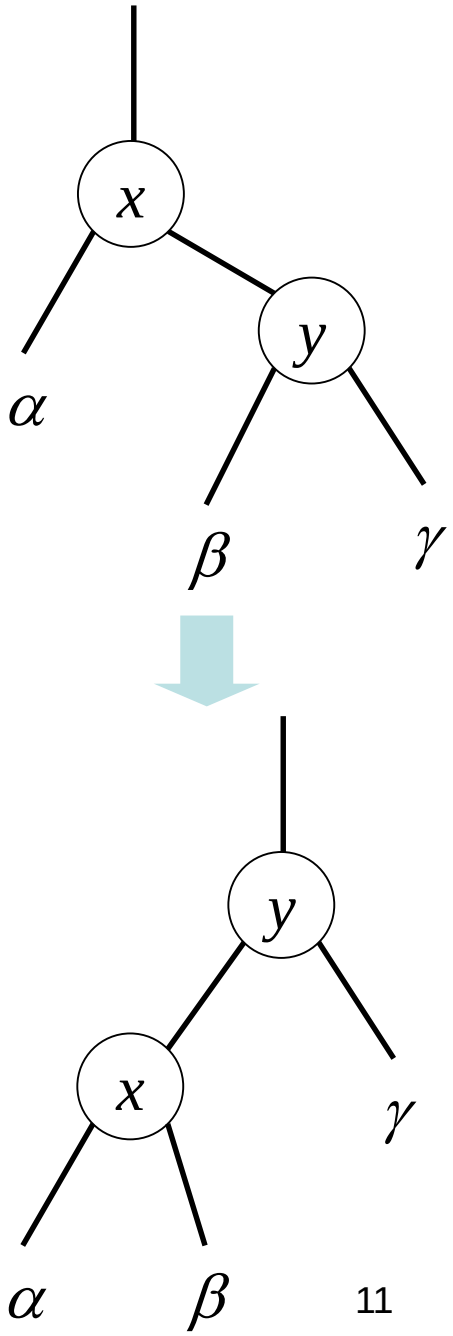
- 2色木で節点を追加/削除すると2色木の条件を満たさなくなることがある.
- 条件を満たすように木の構造を変更する
- $\alpha < x < \beta < y < \gamma$ の順を保つ




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

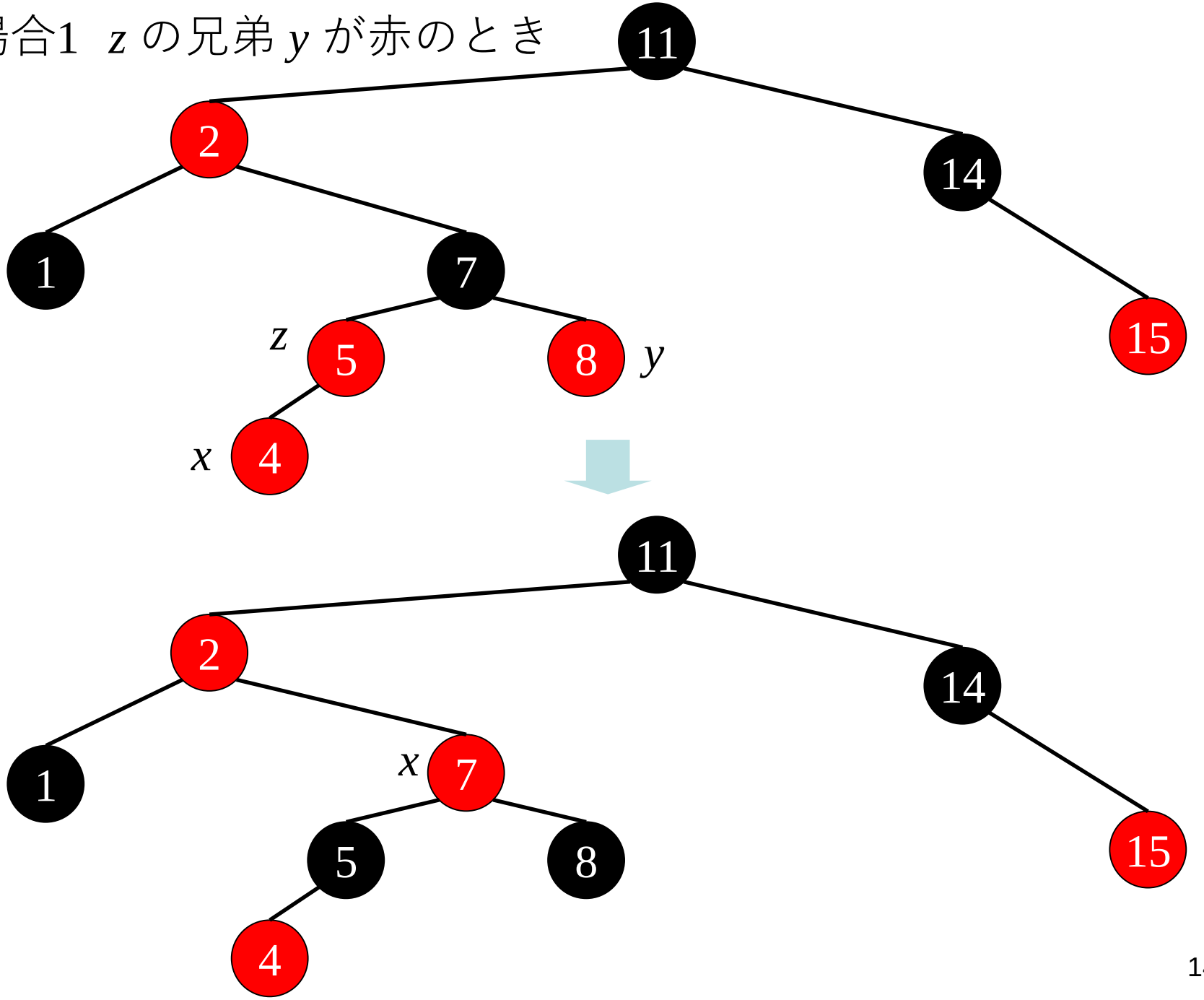
- keyに従って節点 x を葉に挿入する
- x の色を赤にする
- x を挿入後の2色木条件
  1. 各節点は赤か黒のどちらか…OK
  2. 葉 (NIL) は全て黒…OK
  3. もしある節点が赤ならば，その子供は両方黒…?
  4. 1つの節点からその子孫の葉までのどの単純な経路も，同じ数だけ黒節点を含む…OK

x の親が赤のときは条件3を満たさない⇒回転操作

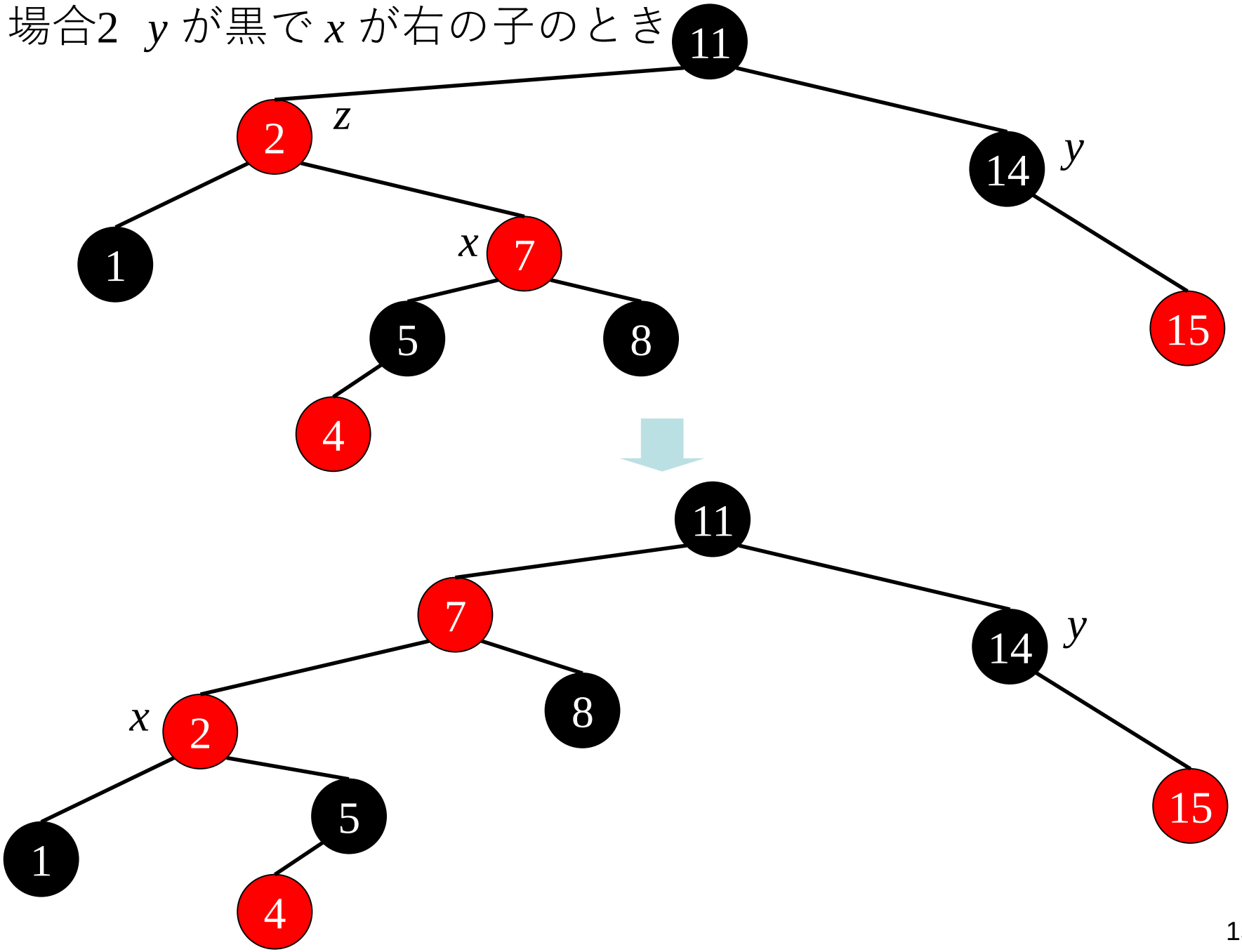
挿入後の回転操作

- 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

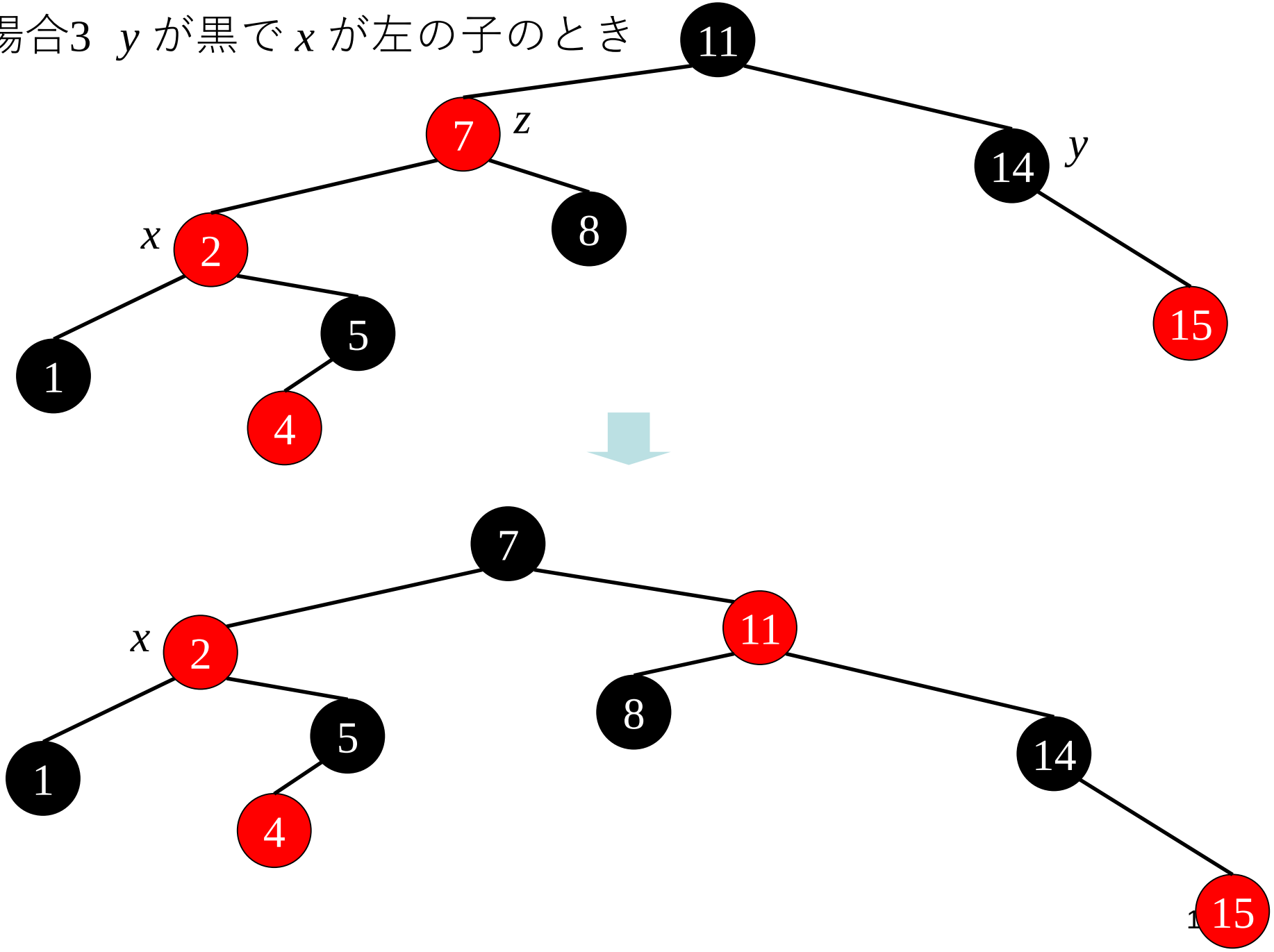
場合1 z の兄弟 y が赤のとき



場合2 y が黒で x が右の子のとき

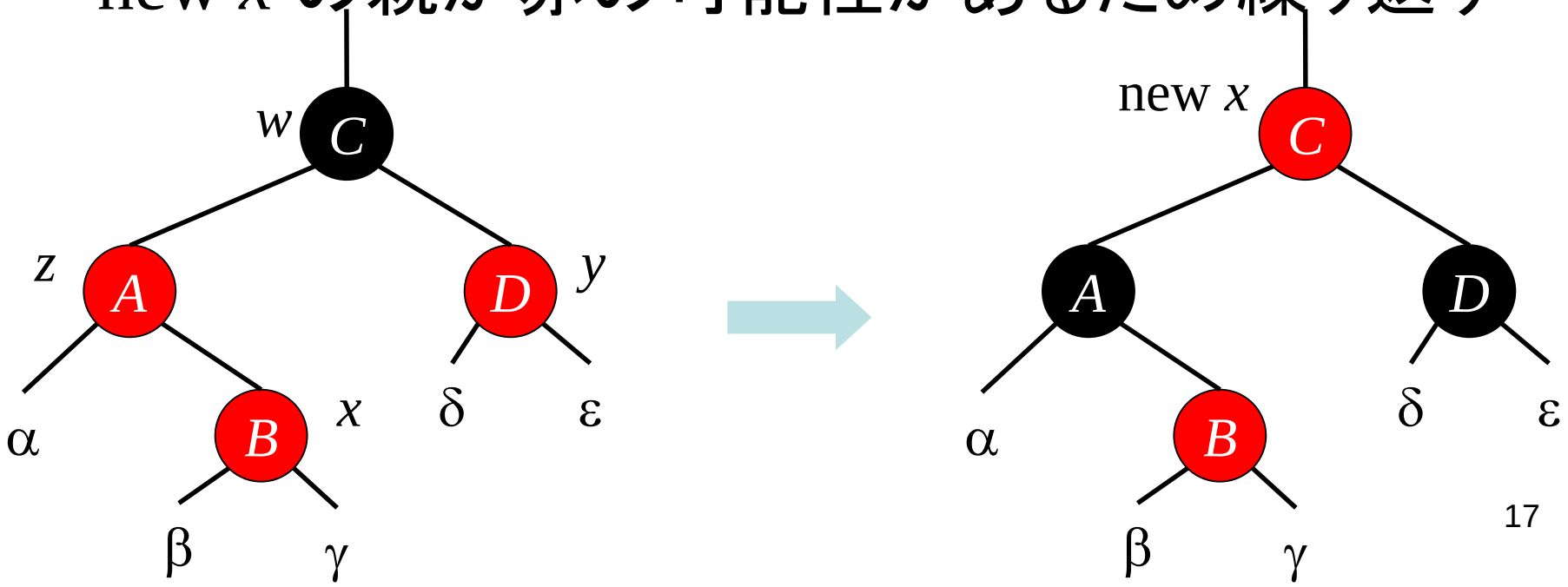


場合3 y が黒で x が左の子のとき



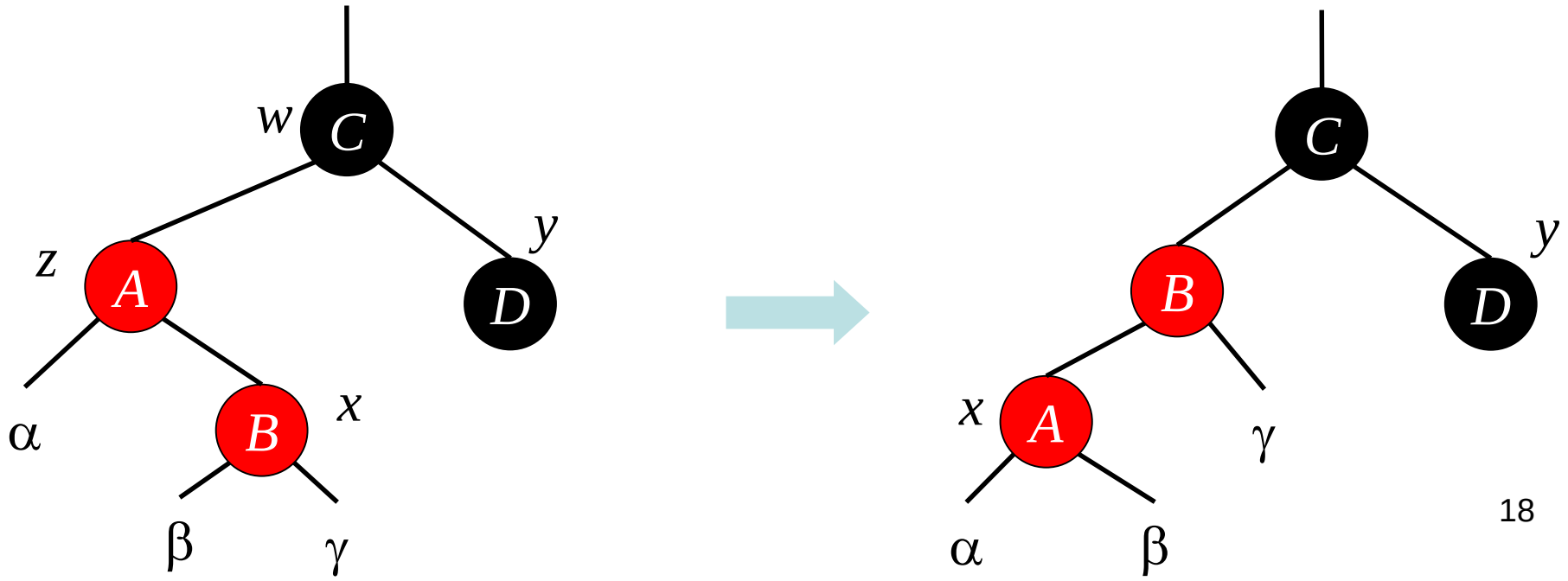
場合1: z の兄弟 y が赤のとき

- 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



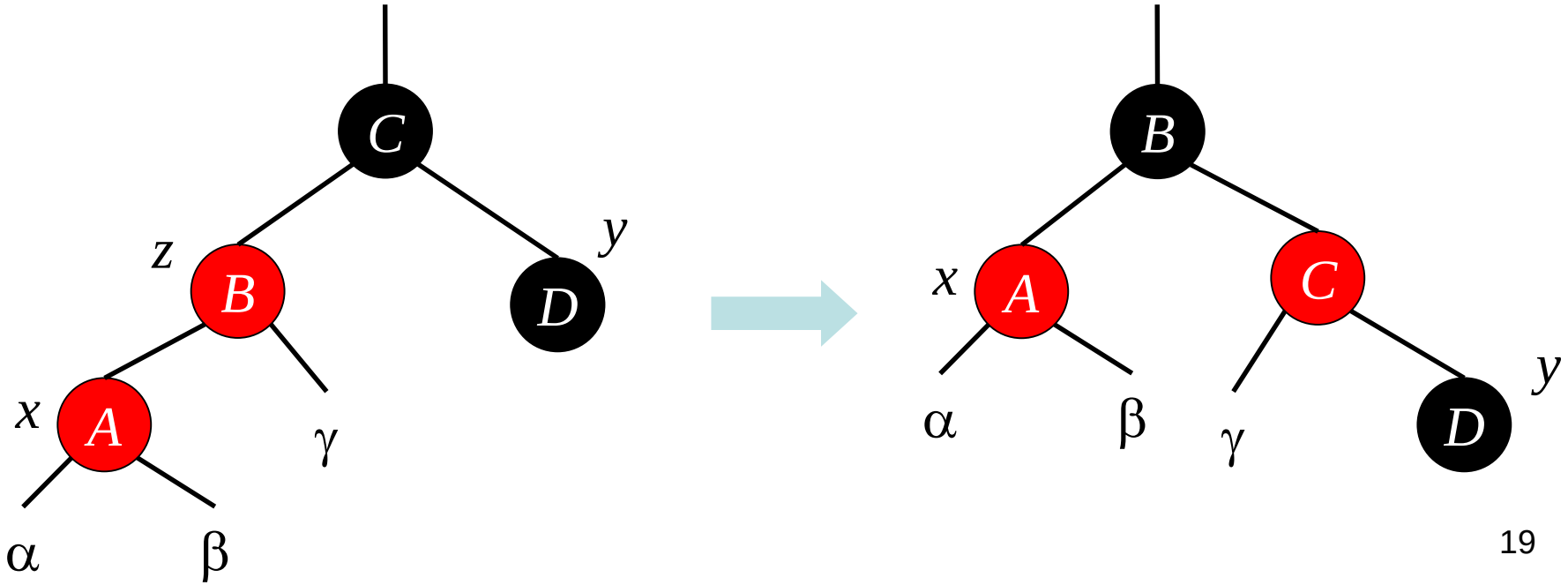
場合2: y が黒で x が右の子のとき

- 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$\Rightarrow O(\lg n)$
- 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

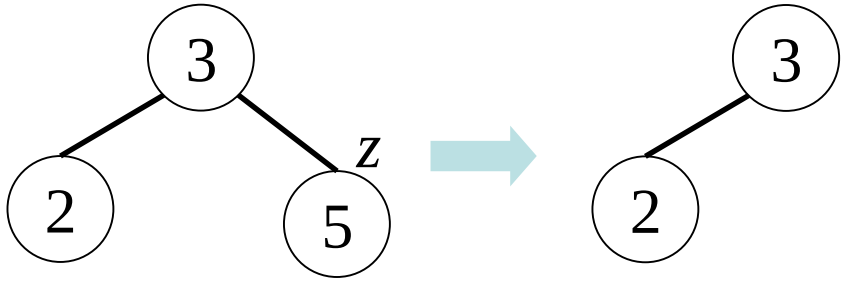
```
void rb_insert(tree *T, node *z)
{
  node *x,*y;
  y = nil(T);
  x = root(T);
  while (x != nil(T)) {// z を挿入する場所 x を決める
    y = x; // y は x の親
    if (key(z) < key(x)) x = left(x);
    else                 x = right(x);
  }
  parent(z) = y; // z の親を y にする
  if (y == nil(T)) {
    root(T) = z; // 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;
  }
  left(z) = nil(T); // z は葉なので左右の子は無い
  right(z) = nil(T);
  color(z) = RED; // 新しい節点は赤にする
  rb_insert_fixup(T, z);
}
```

```
void rb_insert_fixup(tree *T, node *x)
{
  node *y, *z;
  while (color(parent(x)) == RED) { // 2色木条件を満たしていないなら修正
    z = parent(x); // z は x の親
    if (z == left(parent(z))) { // z が左の子の場合
      y = right(parent(z)); // y は z の兄弟
      if (color(y) == RED) { // 場合1: y が赤のとき
        color(z) = BLACK;
        color(y) = BLACK;
        color(parent(z)) = RED;
        x = parent(z);
      } else {
        if (x == right(z)) { // 場合2: y が黒で x が右の子の場合
          x = z;
         left_rotate(T, x);
        } // 場合3 へ続く
        color(parent(x)) = BLACK; // 場合3: y が黒で x が左の子の場合
        color(parent(parent(x))) = RED;
        right_rotate(T, parent(parent(x)));
      }
    } else { // z が右の子の場合
```

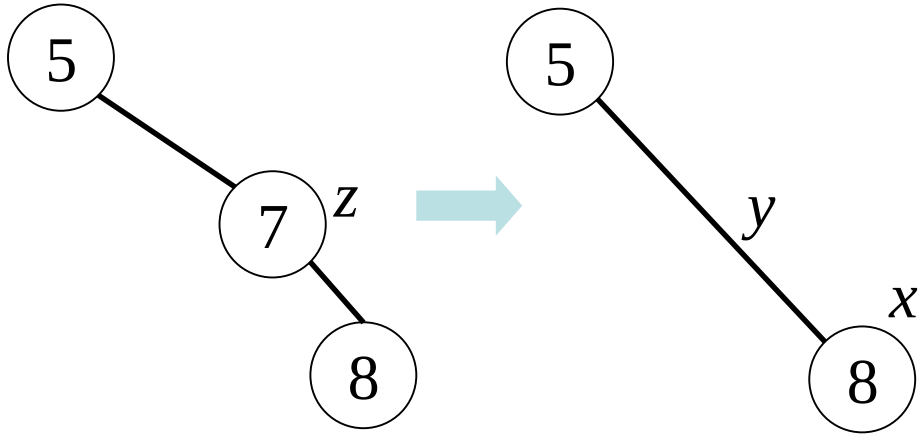
```
} else { // z が右の子の場合
      y = left(parent(z));
      if (color(y) == RED) {
        color(z) = BLACK;
        color(y) = BLACK;
        color(parent(z)) = RED;
        x = parent(z);
      } else {
        if (x == left(z)) {
          x = z;
          right_rotate(T, x);
        }
        color(parent(x)) = BLACK;
        color(parent(parent(x))) = RED;
        left_rotate(T, parent(parent(x)));
      }
    }
  }
  color(root(T)) = BLACK;
}
```

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

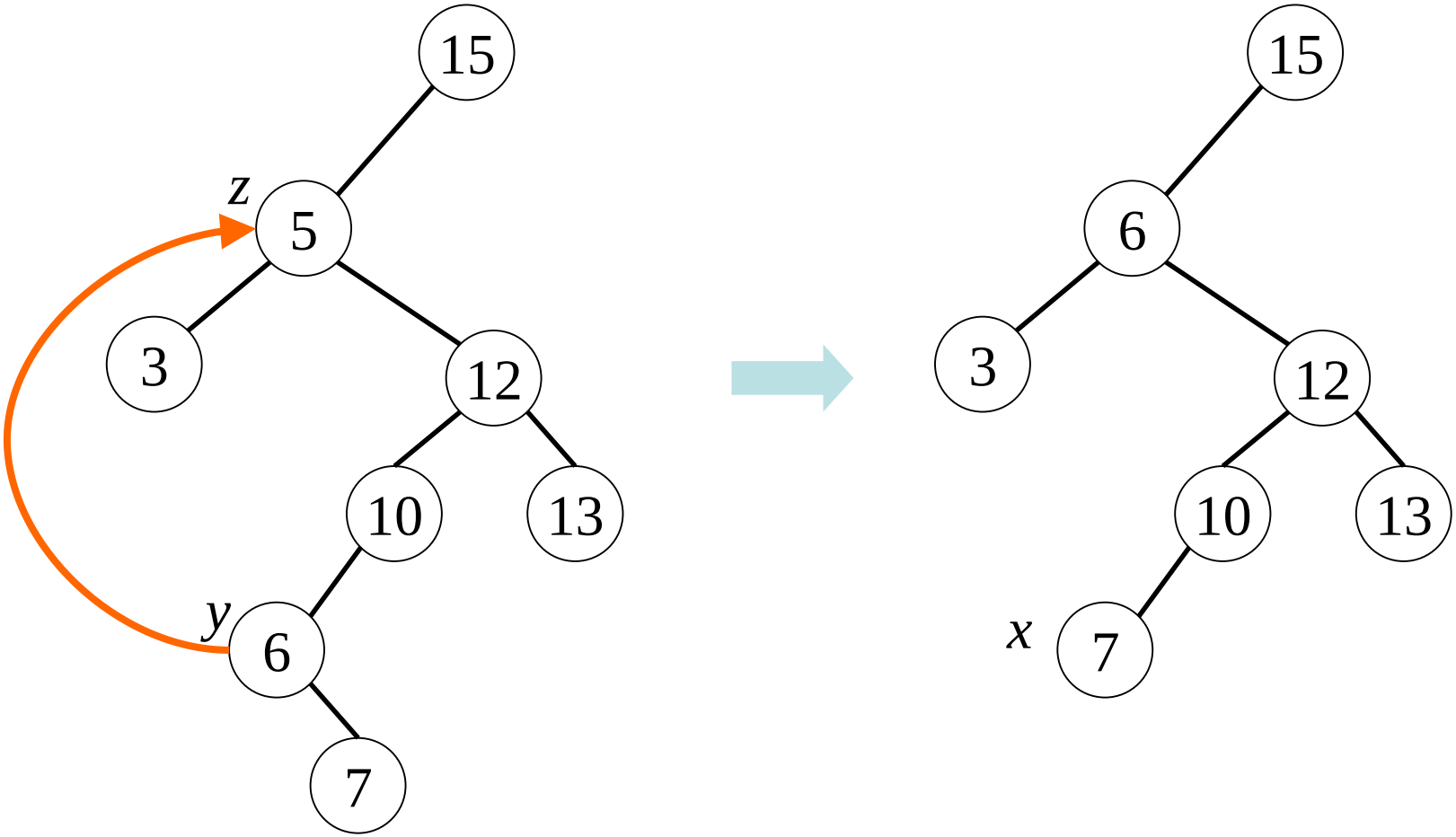


- z が子を1つ持つ場合
  - z の親と z の子を結ぶ



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

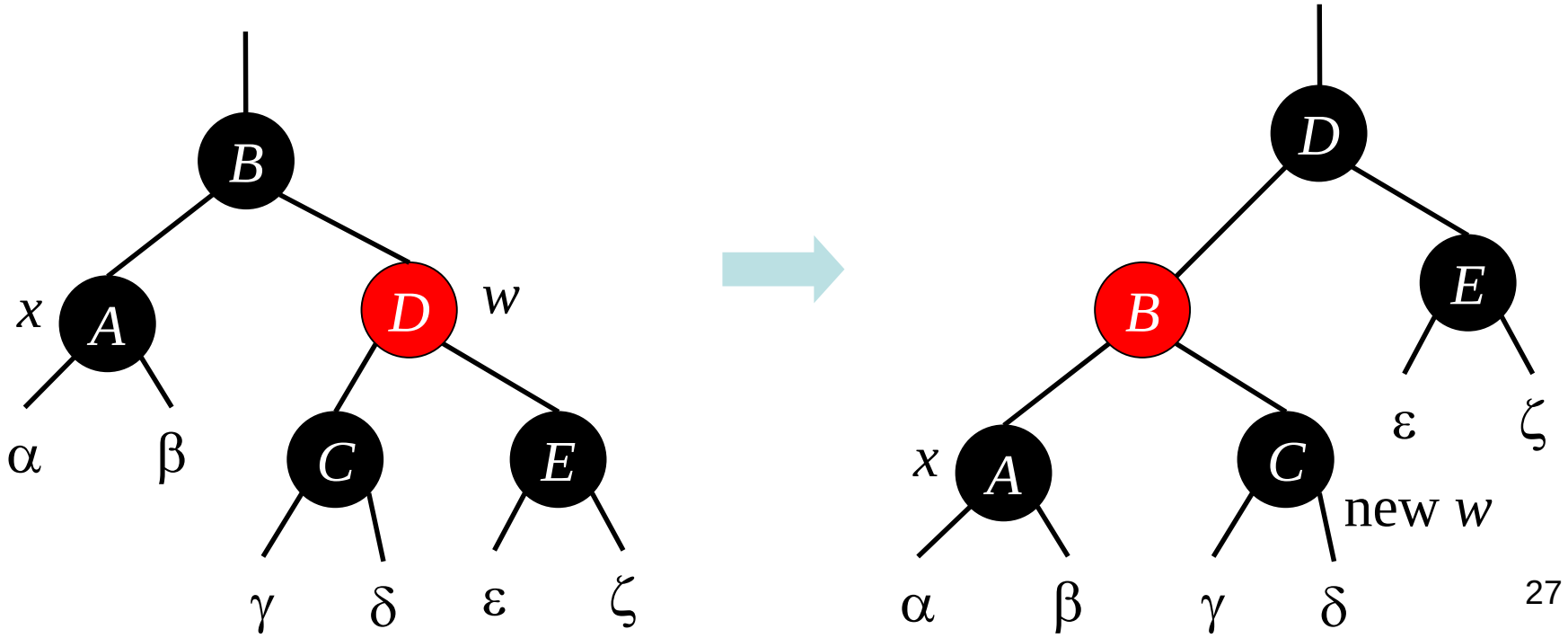


2色木の修正

- 削除される節点を y, その子の1つを x とする
- y が黒のとき，削除すると y を含んでいたどの経路も黒高さが1減る
- x を含む全ての経路で黒高さが1増えればいい

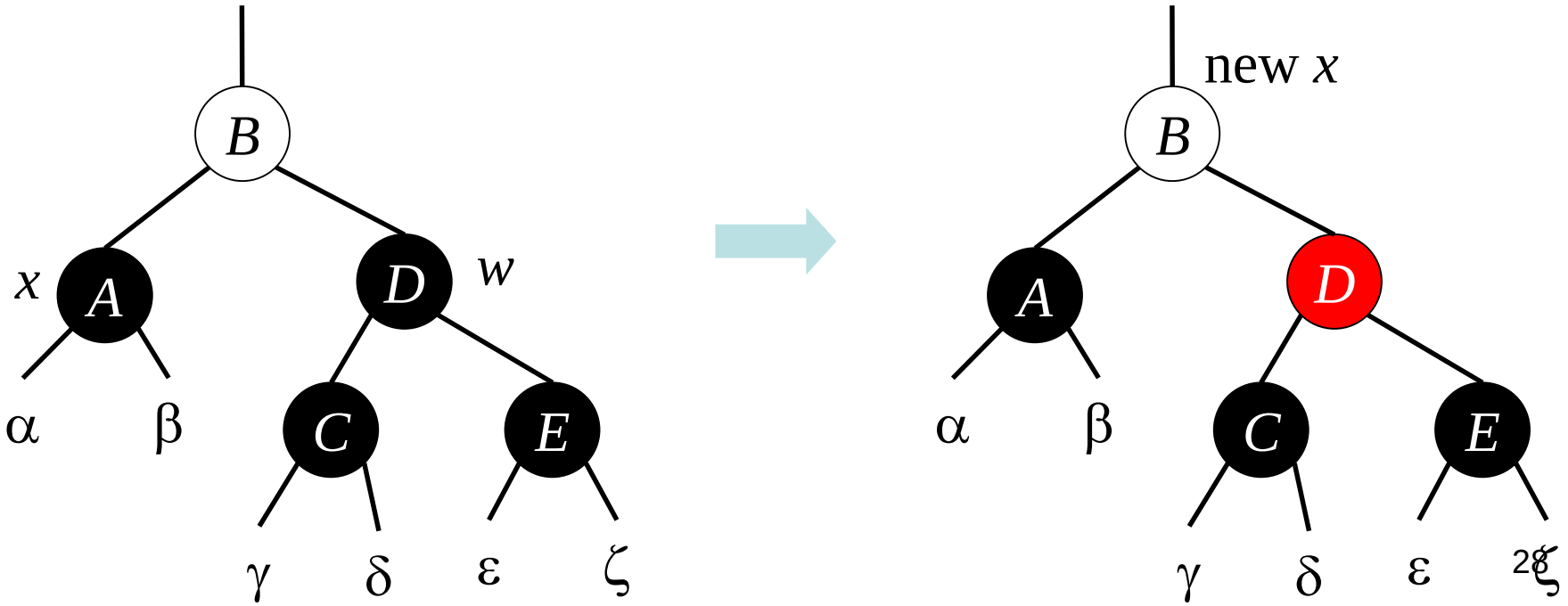
場合1: x の兄弟 w が赤

- B で左回転
- B と D の色を変える
- x の兄弟が黒になる⇒場合2, 3, 4



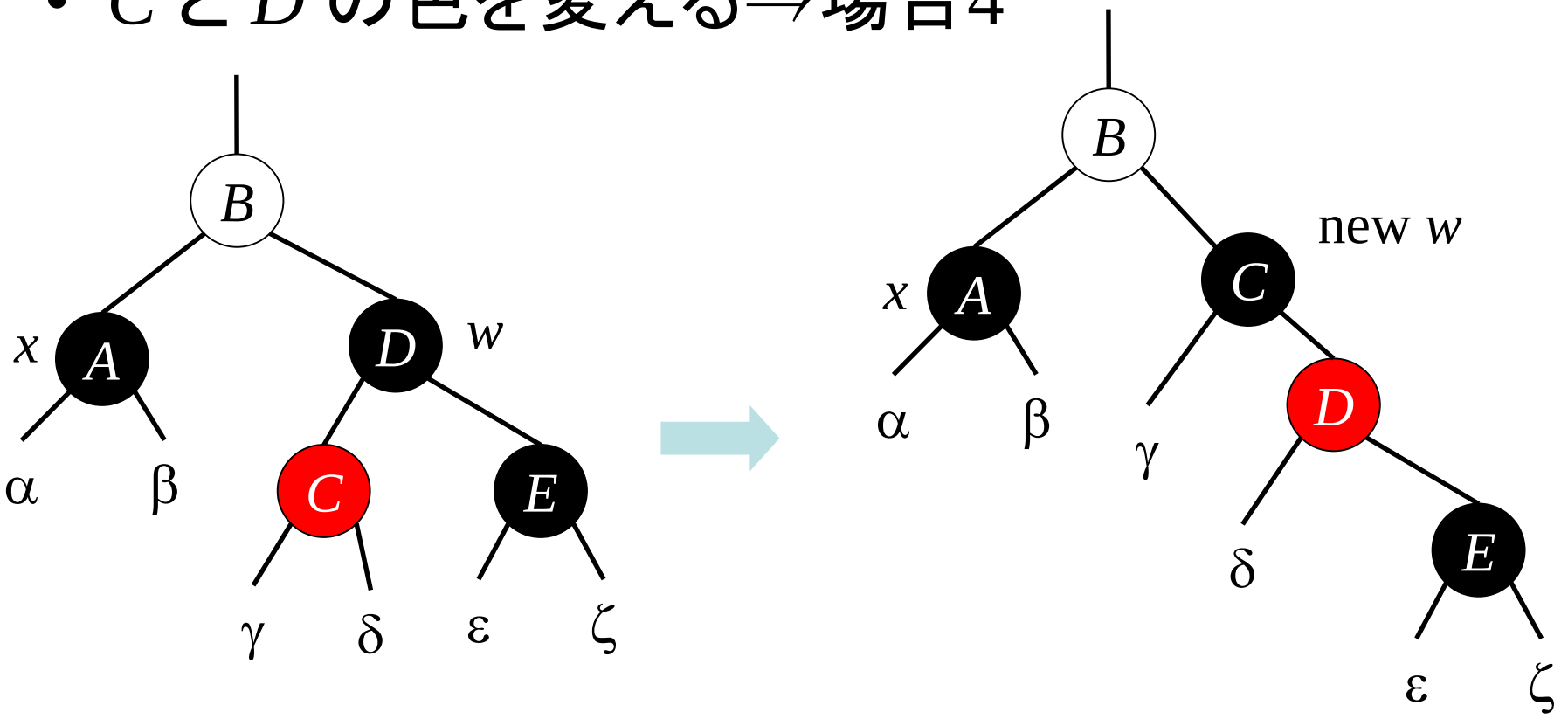
場合2: w の左右の子が黒

- D の色を変える
- B が赤ならそれを黒にして終了
- B が黒なら B を新しい x として繰り返す



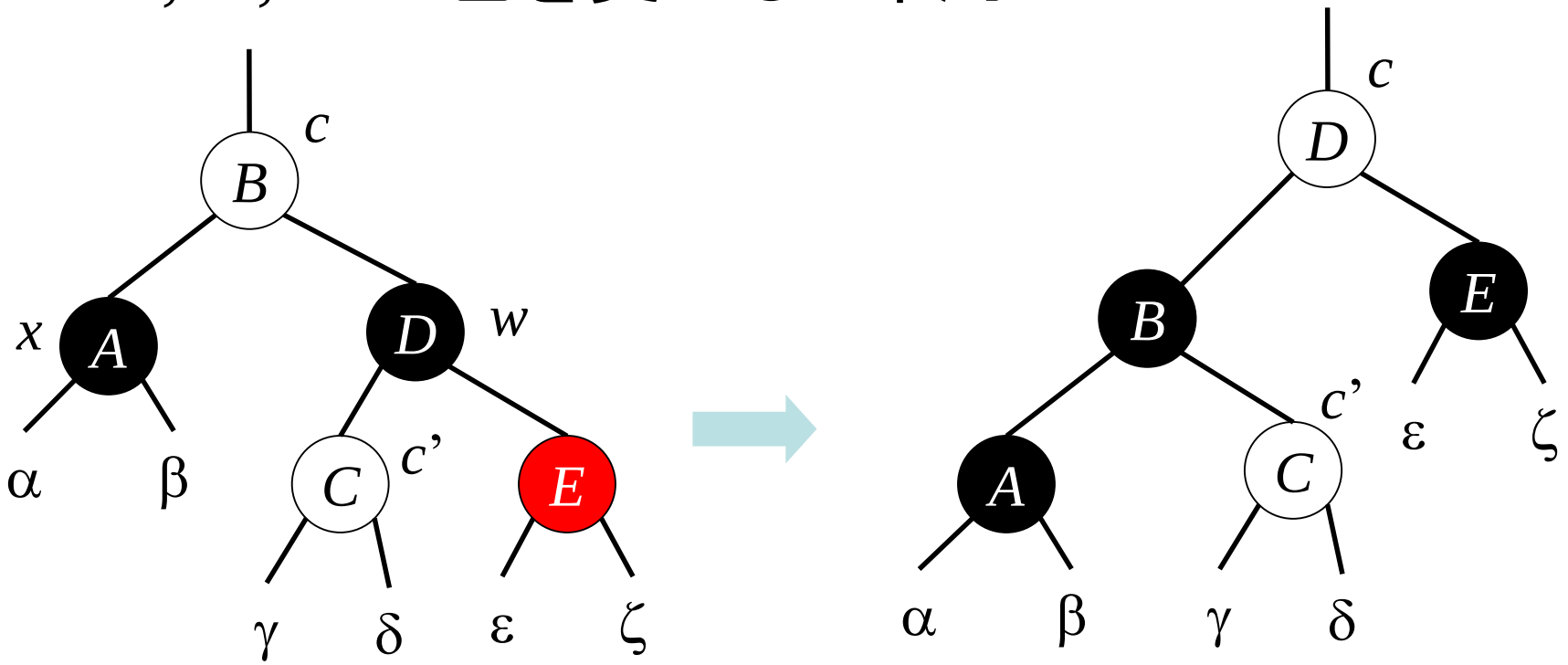
場合3: w の右の子が黒

- D で右回転
- C と D の色を変える⇒場合4



場合4: w の右の子が赤

- B で左回転
- B, D, E の色を変える⇒終了



new x = root(T)

rb_deleteの計算時間

- tree_delete は $O(\lg n)$ 時間
- 場合1, 3, 4は定数時間，最大3回の回転
  - 場合3,4に行くとそこで終了
  - 場合1から2へ移った場合，新しい x は赤なので終了
- 場合2では回転は行わず，$O(\lg n)$ 回木を登る
- 全体で $O(\lg n)$ 時間，高々3回の回転

```
node *rb_delete(tree *T, node *z)
{
  node *x, *y;
  if (left(z) == nil(T) || right(z) == nil(T)) { // z の子の数が 0 か 1
    y = z;
  } else { // z の子の数が 2
    y = rbtree_successor(T, z);
  }
  if (left(y) != nil(T)) x = left(y); else x = right(y);
  parent(x) = parent(y);
  if (parent(y) == nil(T)) {
    root(T) = x;
  } else {
    if (y == left(parent(y))) {
      left(parent(y)) = x;
    } else {
      right(parent(y)) = x;
    }
  }
  if (y != z) key(z) = key(y);
  if (color(y) == BLACK) rb_delete_fixup(T, x); // 黒節点を削除した場合は木を修正
  return y;
}
```

```
void rb_delete_fixup(tree *T, node *x)
{
  node *w;
  while (x != root(T) && color(x) == BLACK) {
    if (x == left(parent(x))) { // x が左の子の場合
      w = right(parent(x)); // w は x の兄弟
      if (color(w) == RED) { // 場合1: w が赤
        color(w) = BLACK;
        color(parent(x)) = RED;
        left_rotate(T, parent(x));
        w = right(parent(x));
      } // 場合2 へ続く
      if (color(left(w)) == BLACK && color(right(w)) == BLACK) { // 場合2: w の左右の子が黒
        color(w) = RED;
        x = parent(x); // 1つ上に移動して再度繰り返す
      } else {
        if (color(right(w)) == BLACK) { // 場合3: w の右の子が黒
          color(left(w)) = BLACK;
          color(w) = RED;
          right_rotate(T, w);
          w = right(parent(x));
        } // 場合4 へ続く
        color(w) = color(parent(x));  // 場合4: w の右の子が赤
        color(parent(x)) = BLACK;
        color(right(w)) = BLACK;
        left_rotate(T, parent(x));
        x = root(T); // 終了
      }
    } else {
```

```
 } else {
      w = left(parent(x));
      if (color(w) == RED) {
        color(w) = BLACK;
        color(parent(x)) = RED;
        right_rotate(T, parent(x));
        w = left(parent(x));
      }
      if (color(right(w)) == BLACK && color(left(w)) == BLACK) {
        color(w) = RED;
        x = parent(x);
      } else {
        if (color(left(w)) == BLACK) {
          color(right(w)) = BLACK;
          color(w) = RED;
          left_rotate(T, w);
          w = left(parent(x));
        }
        color(w) = color(parent(x));
        color(parent(x)) = BLACK;
        color(left(w)) = BLACK;
        right_rotate(T, parent(x));
        x = root(T);
      }
    }
  }
  color(x) = BLACK;
}
```

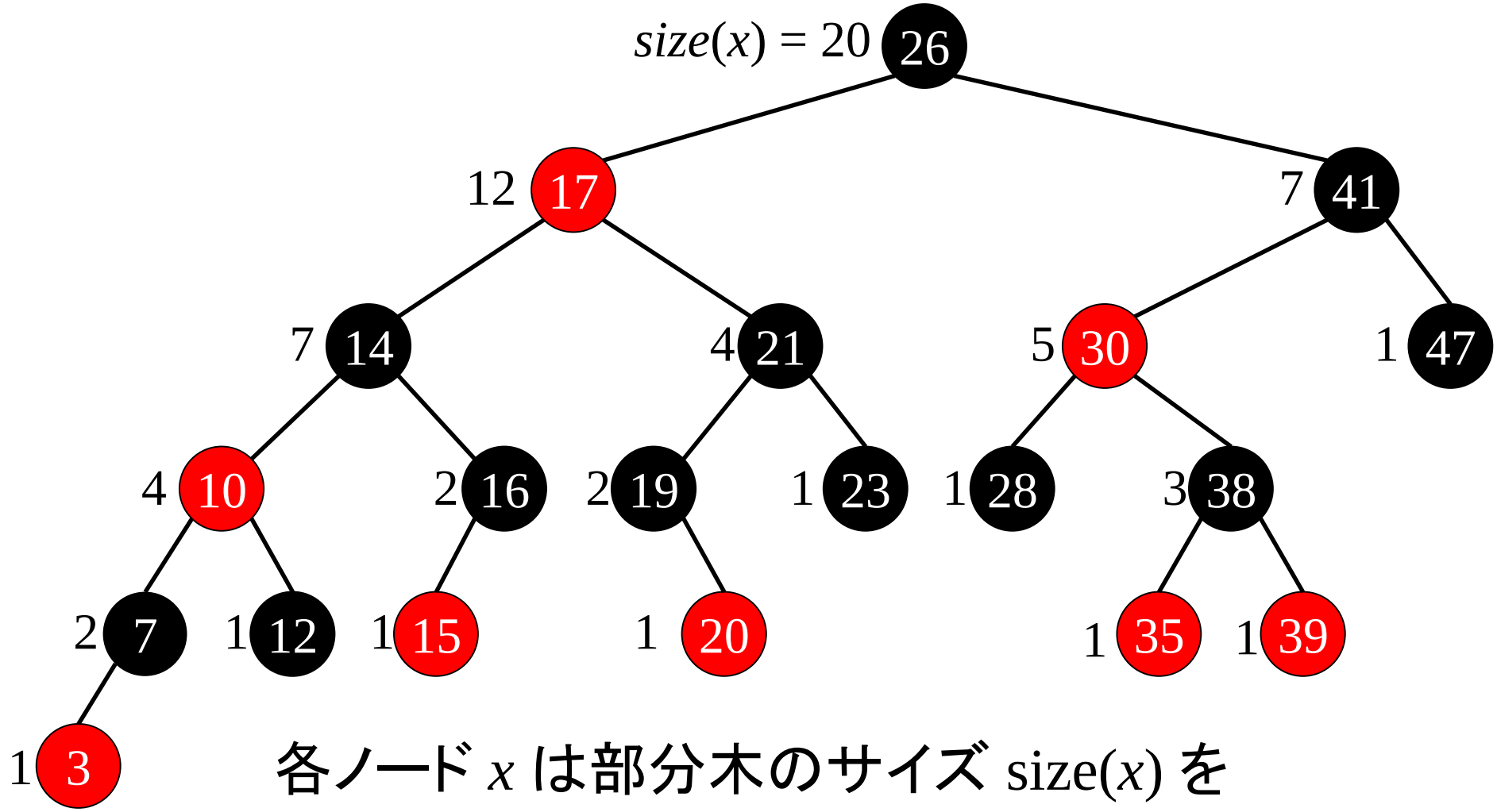
データ構造の補強

- 新しいデータ構造が必要なときでも，標準的なデータ構造に情報を追加することで十分な場合が多い
- 2色木を拡張する
  - 動的集合での順序統計量の計算
  - 区間の動的集合の管理

動的順序統計量

- n 個の要素からなる集合の i 番目の順序統計量: 集合の中で i 番目に小さいキーを持つ要素
- 任意の順序統計量は $O(n)$ 時間で求まる
- 2色木を補強して，任意の順序統計量を $O(\log n)$ 時間で決定できるようにする.
- 要素のランク (rank) (指定された要素が小さい方から何番目か) も $O(\log n)$ 時間で求まる

順序統計量木



各ノード x は部分木のサイズ size(x) を格納するフィールドを持つ

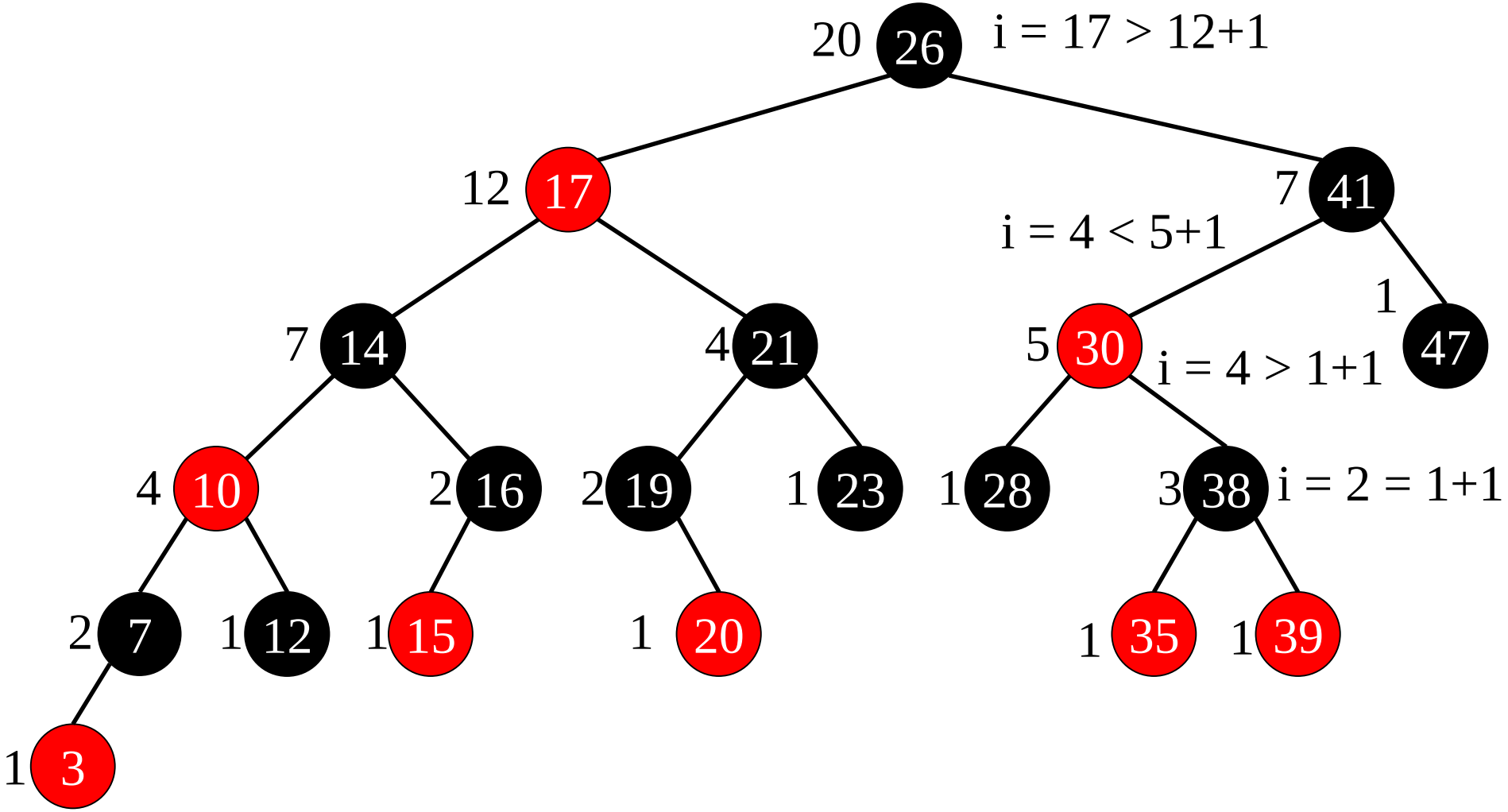
size(x) = size(left(x)) + size(right(x)) + 1

与えられたランクを持つ要素を求める

- os_select(x, i): x を根とする部分木で i 番目に小さいキー (順序統計量) を持つ節点を返す
- size(left(x)) + 1はxを根とする部分木でのxのランク
- 計算量: O(lg n)

```
node os_select(node x, int i)
{
    int r;
    r = size(left(x)) + 1;
    if (i == r)  return x;
    else if (i < r)  return os_select(left(x),i);
          else         return os_select(right(x), i-r);
}
```

os_select(root(T), 17) の場合

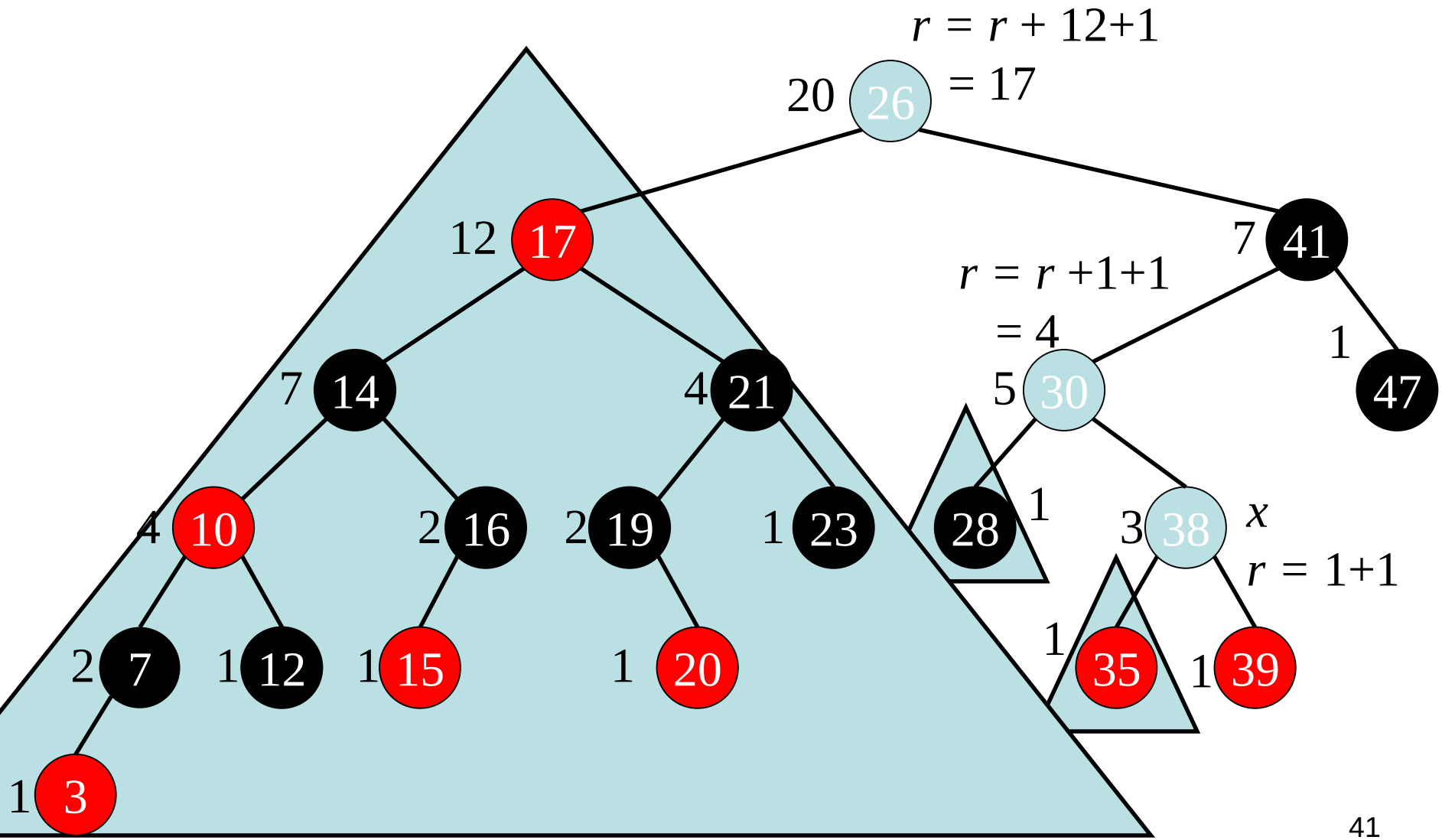


要素のランクを求める

- 節点 x のポインタが与えられたときに，その要素が小さい方から何番目かを返す
- 計算量: $O(\lg n)$

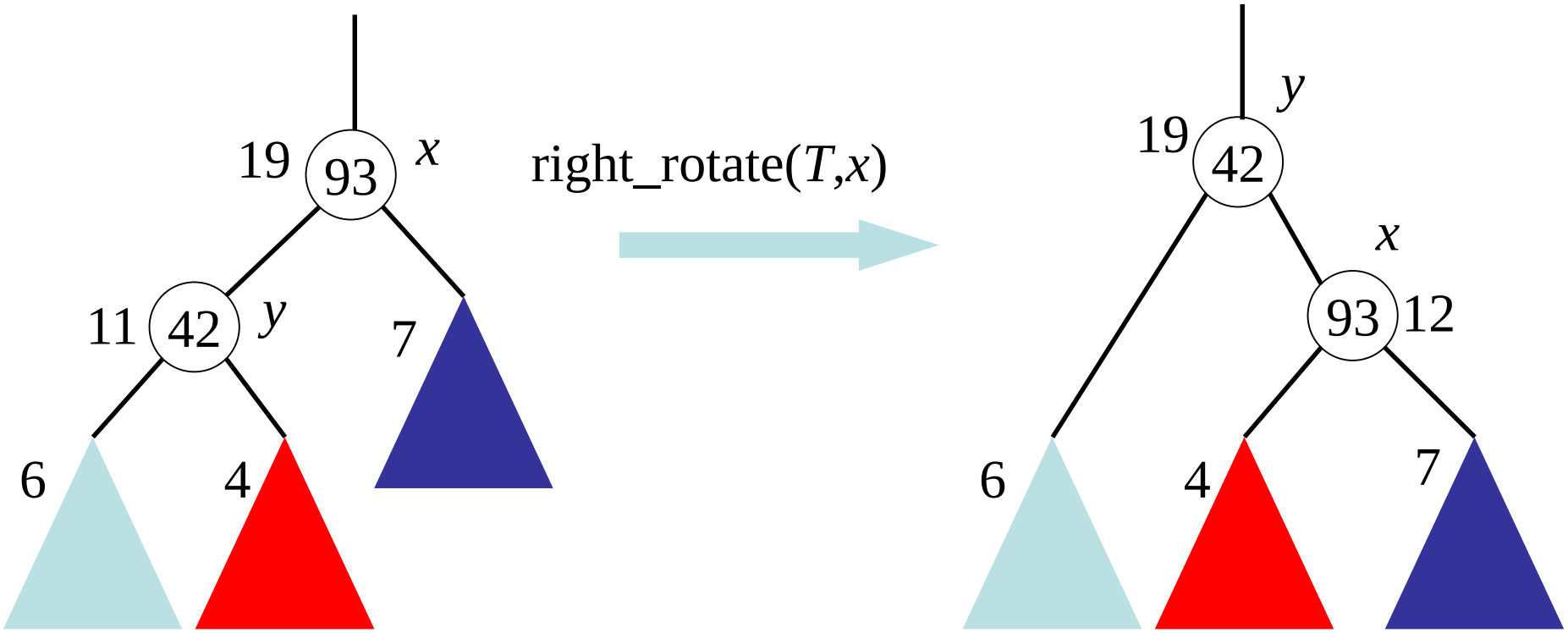
```
int os_rank(tree T, node x)
{
    int r;  node y;
    r = size(left(x)) + 1; // x を根とする部分木での x のランク
    y = x;
    while (y != root(T)) {
        if (y == right(p(y)))  r += size(left(p(y)) + 1;
        y = p(y);
    }
    return r;
}
```

os_rank(T, x) の例


r = r + 12+1
= 17
20 26
12 17
7 41
r = r +1+1
= 4
7 14
4 21
5 30
1 47
4 10
2 16
2 19
1 23
28 1
3 38
x
r = 1+1
2 7
1 12
1 15
1 20
1 35
1 39
1 3

部分木のサイズの管理

- 2色木の形状が変化すると size フィールドの値は変化する
- 挿入，削除後も size の値を維持できることを示す
- 挿入の場合
  - 木を根から辿り新しい要素を挿入する場所を見つける
  - 新しい要素の祖先の size を1ずつ増やす
  - 2色木の回転操作を行う
  - size を調整する



size(y) = size(left(y)) + size(right(y)) + 1

size(x) = size(left(x)) + size(right(x)) + 1

1回の回転は O(1) 時間

回転は高々2回

全体の計算量: O(lg n)

削除の場合

- ノード y を削除したあとに size が変化する節点は, y の祖先のみ
- y から根までの経路を辿りながら size を1ずつ減らせばいい
- 回転操作は挿入と同様
  - 高々3回の回転
- 計算量: $O(\lg n)$

データ構造の補強の方針

1. 元になるデータ構造を選ぶ
   - 2色木
2. 元になるデータ構造で追加情報として管理されるものを決定する
   - 部分木のサイズ size
3. 元になるデータ構造の基本操作に対して追加情報を維持できることを確認する
   - 挿入，削除時の size の更新が $O(\lg n)$ 時間
4. 新しい操作を開発する
   - os_select, os_rank

定理 (2色木の補強) T を n 節点からなる2色木, f を T を補強するフィールドとする. 節点 x に対する f の内容は x の情報 (left(x), right(x), f(left(x), f(right(x)) のみを用いて計算できるとする.
このとき, 挿入/削除の際に T の全ての節点の f の値を $O(\lg n)$ 時間で維持できる.

証明: (アイデア) f の内容はその子供から決まる. よってある節点 x での f の変更はその祖先にしか影響を与えない. よって挿入/削除の際の f の更新は $O(\lg n)$ 時間.

挿入の場合:

第1段階: 新しい節点xは既存の節点の子として挿入.

x の子は共にNILなので $f(x)$ は $O(1)$ 時間で計算可.

x の祖先の節点での f の更新は $O(\lg n)$ 時間.

第2段階: 木構造の変化は回転によってのみ発生.

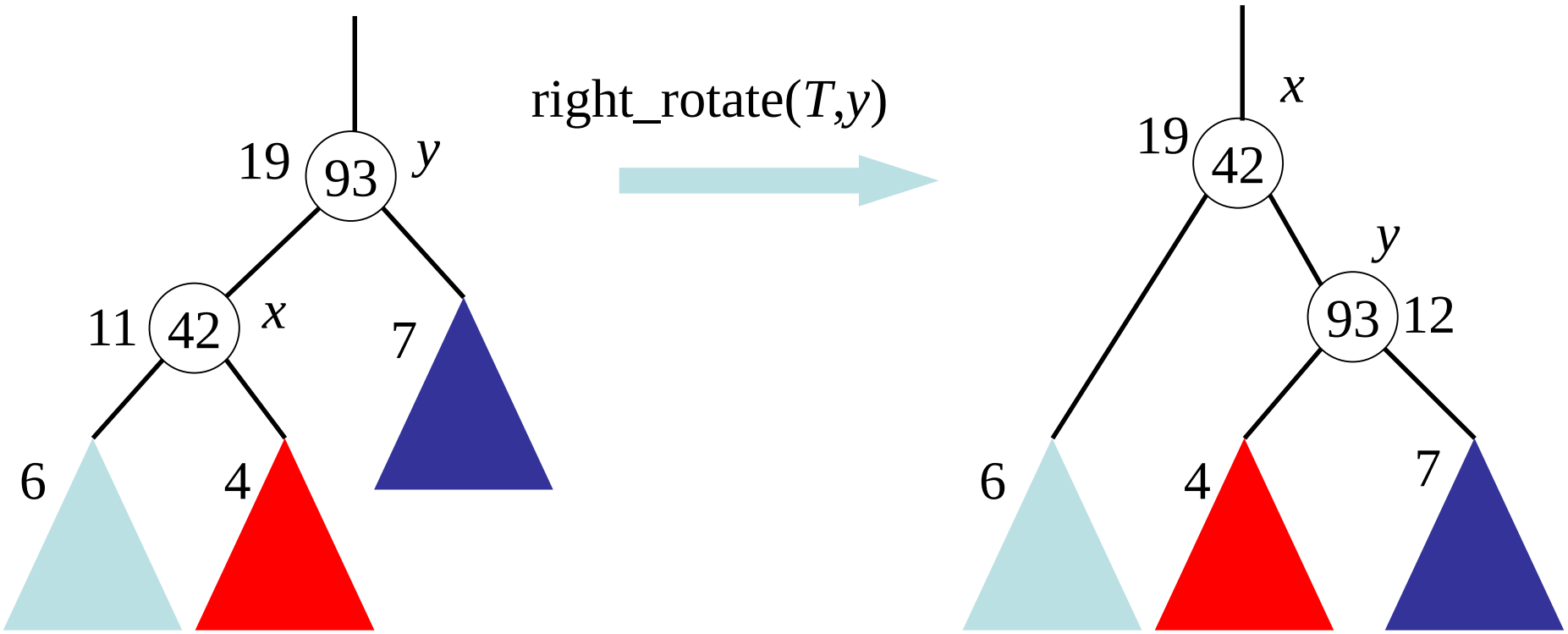
1つの回転では2つの節点のみが変化する.

節点が1つ変化するたびにその節点とその先祖の f を更新する. その時間は $O(\lg n)$.

挿入の際に発生する回転は高々2回であるため,

挿入に要する総時間は $O(\lg n)$.　　削除も同様.

注: 多くの場合，回転後の更新のコストは O(lg n) ではなく O(1).



size(y) = size(left(y)) + size(right(y)) + 1

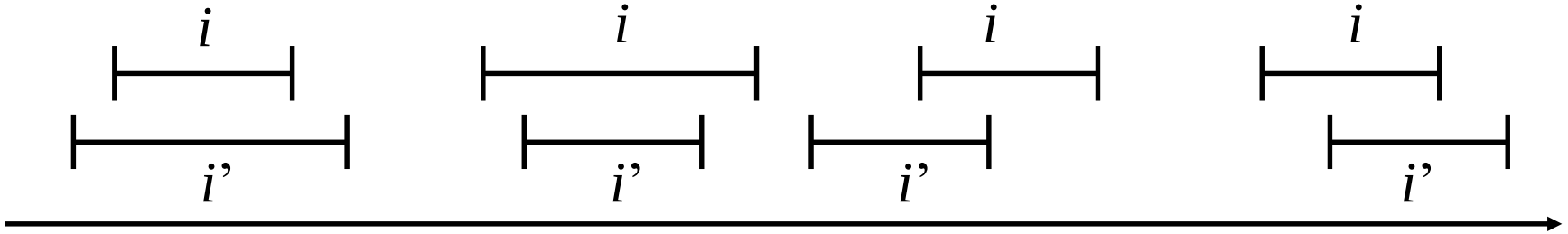
size(x) = size(left(x)) + size(right(x)) + 1

区間木

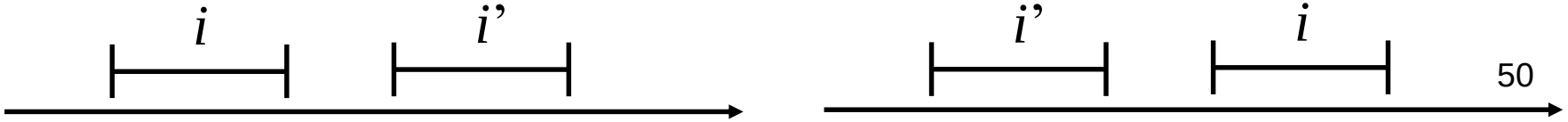
- 定義: 閉区間 (closed interval) とは，実数の順序対 $[t_1, t_2]$ $(t_1 \leq t_2)$
- 開区間(open interval),半開区間(half-open interval)は区間の短点の両方または片方を省いたもの
- ここでは閉区間のみを扱う
- 区間木は，ある与えられた期間に起こった出来事を見つける質問などに用いられる

区間の間の関係

- 区間 $[t_1, t_2]$ をオブジェクト i とする.
- i はフィールド $\mathrm{low}[i] = t_1$ (下端点), $\mathrm{high}[i] = t_2$ (上端点) で表される
- 区間 i と i' が重なる $\leftrightarrow i \cap i' \neq \emptyset$ つまり
  1. $\mathrm{low}[i] \leq \mathrm{high}[i']$ かつ $\mathrm{low}[i'] \leq \mathrm{high}[i]$



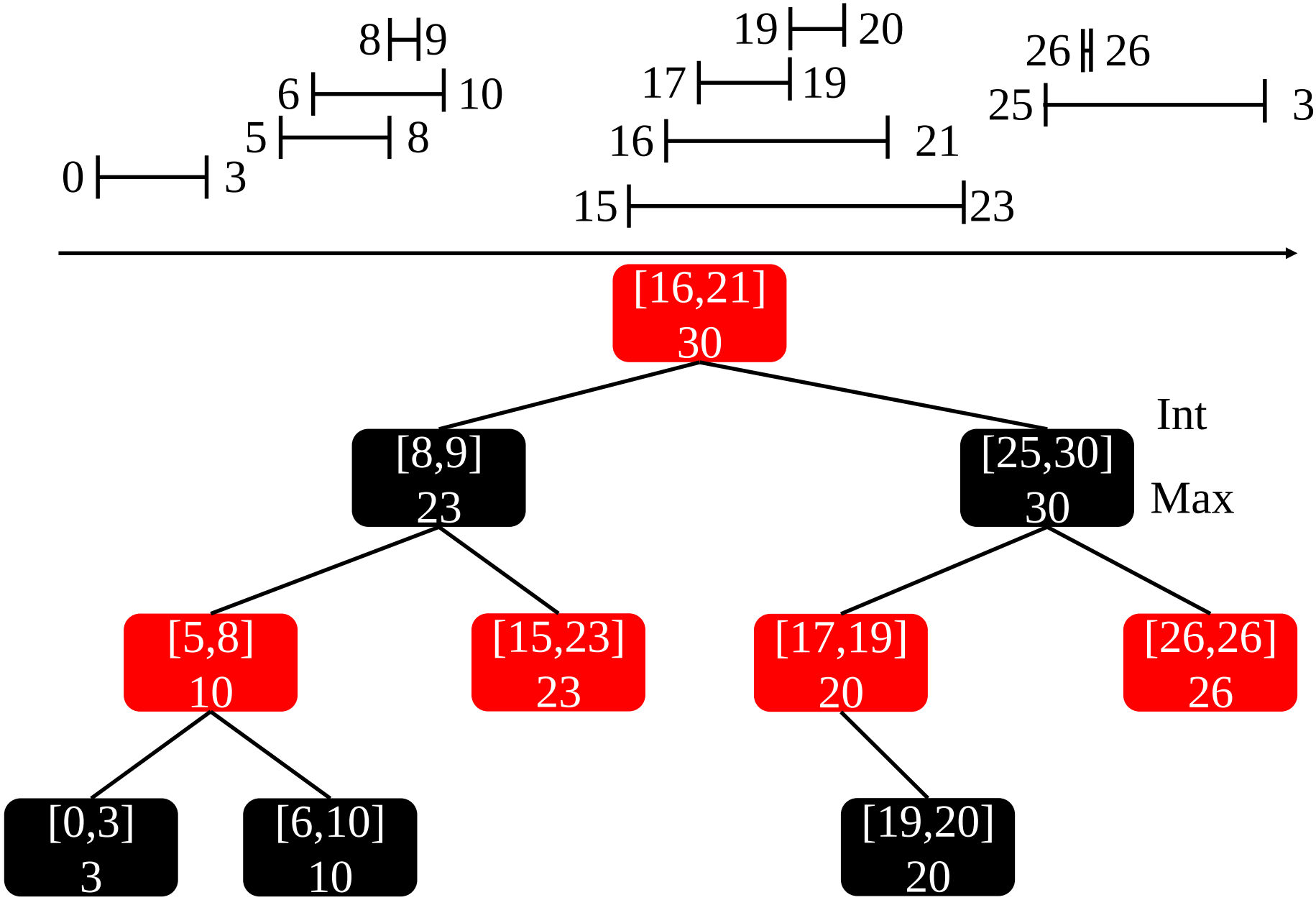
- 区間 i と i' が重ならない場合
  - 2. $\mathrm{high}[i] < \mathrm{low}[i']$ または 3. $\mathrm{high}[i'] < \mathrm{low}[i]$



区間木での操作

- 木の各要素は区間 Int(x) を持つ
- INTERVAL_INSERT(tree T, node x): x を挿入
- INTERVAL_DELETE(tree T, node x): x を削除
- INTERVAL_SEARCH(tree T, interval i): Int(x) が区間 i と重なっている要素 x のポインタを返す

1. 元になるデータ構造: 2色木
   - 各節点 x は区間 Int(x) を含む
   - x のキーは区間の下端点 low(Int(x))



Max(x): x を根とする部分木に蓄えられている区間の端点の最大値 (つまり上端点の最大値)

2. 追加情報
   - Int(x): 区間
   - Max(x): x を根とする部分木に蓄えられている区間の端点の最大値 (つまり上端点の最大値)
3. 情報の更新
   - 挿入/削除を行なったときに Max が $O(\lg n)$ 時間で更新できることを示す
   - $\mathrm{Max}(x) = \max\{\mathrm{high}(\mathrm{Int}(x)), \mathrm{Max}(\mathrm{left}(x)), \mathrm{Max}(\mathrm{right}(x))\}$ より，$O(\lg n)$ 時間で更新できる

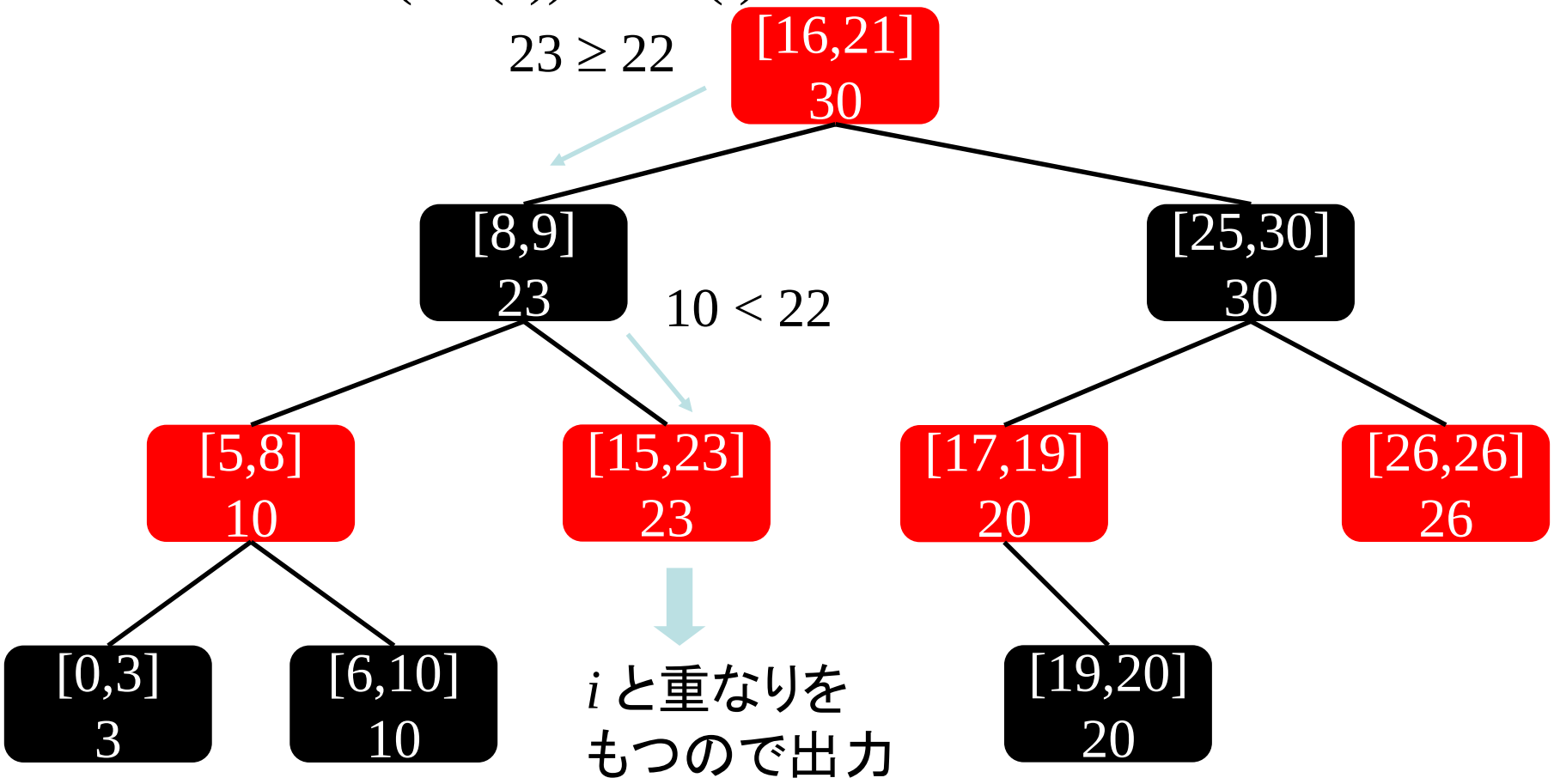
4. 新しい操作の実装

区間 *i* と重なっている区間を見つける
O(lg *n*) 時間

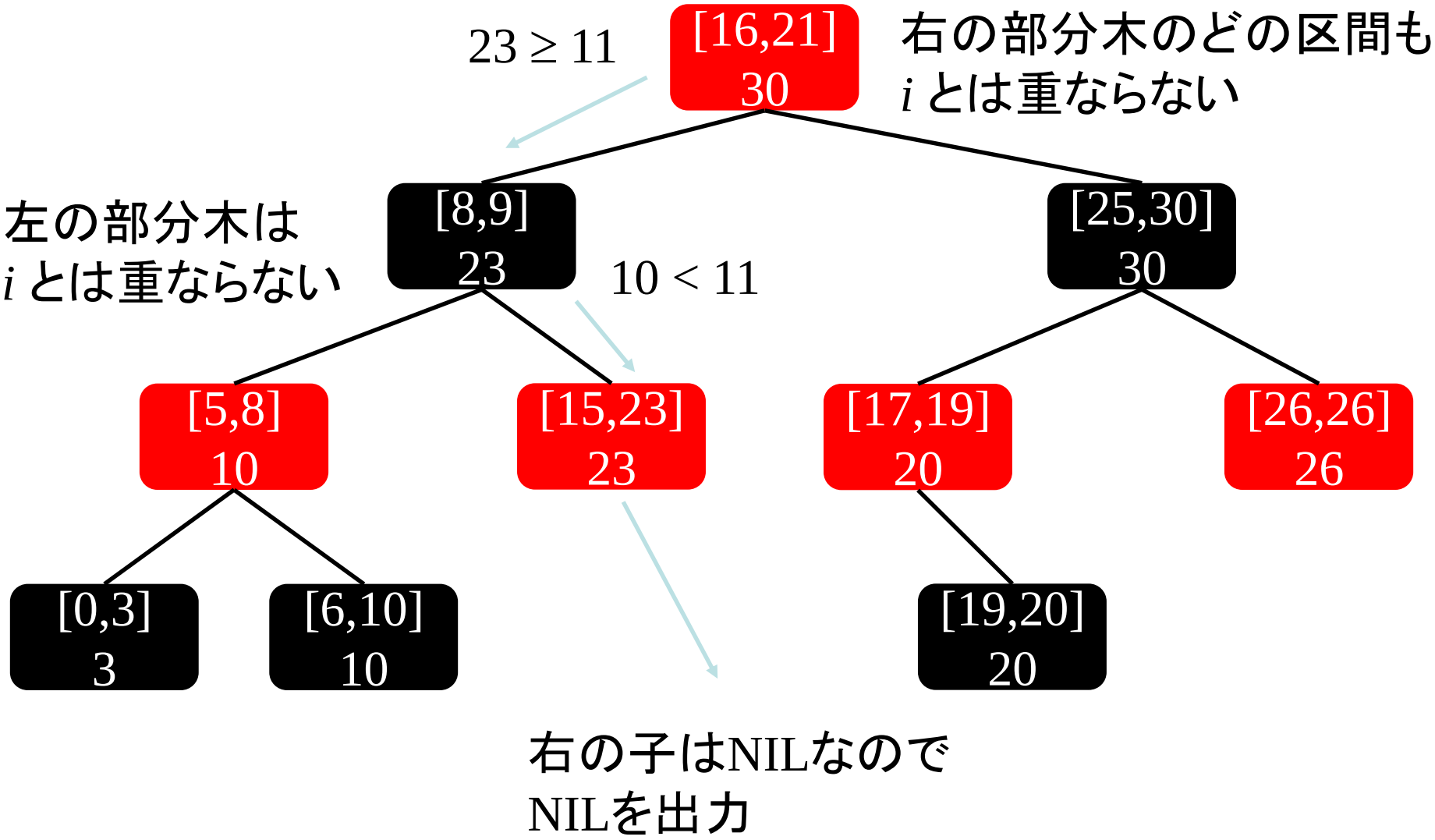
```
node INTERVAL_SEARCH(tree T, interval i)
{
    node x;
    x = root(T);
    while ((x != NIL)
            && (high(i) < low(int(x)) || high(int(x)) < low(i))) {
                //          i と int(x) が重ならない
        if (left(x) != NIL && Max(left(x)) >= low(i))  x = left(x);
        else  x = right(x);
    }
    return x;
}
```

i = [22,25]

Max(left(x)) ≥ low(i)



i = [11,14]


23 ≥ 11
[16,21]
30
右の部分木のどの区間も
i とは重ならない
左の部分木は
i とは重ならない
[8,9]
23
10 < 11
[25,30]
30
[5,8]
10
[15,23]
23
[17,19]
20
[26,26]
26
[0,3]
3
[6,10]
10
[19,20]
20
右の子はNILなので
NILを出力

定理 15.2 INTERVAL_SEARCH(T, i) の実行中の whileループにて,

1. 探索が左に行ったとき: x の左部分木は i と重なり合う区間を含むか, x の右部分木のどの区間も i とは重ならない
2. 探索が右に行ったとき: x の左部分木のどの区間も i とは重ならない

証明: 2の場合: 右に行く場合は $\mathrm{left}(x) = \mathrm{NIL}$ または $\mathrm{Max}(\mathrm{left}(x)) < \mathrm{low}(i)$ が成り立つ.

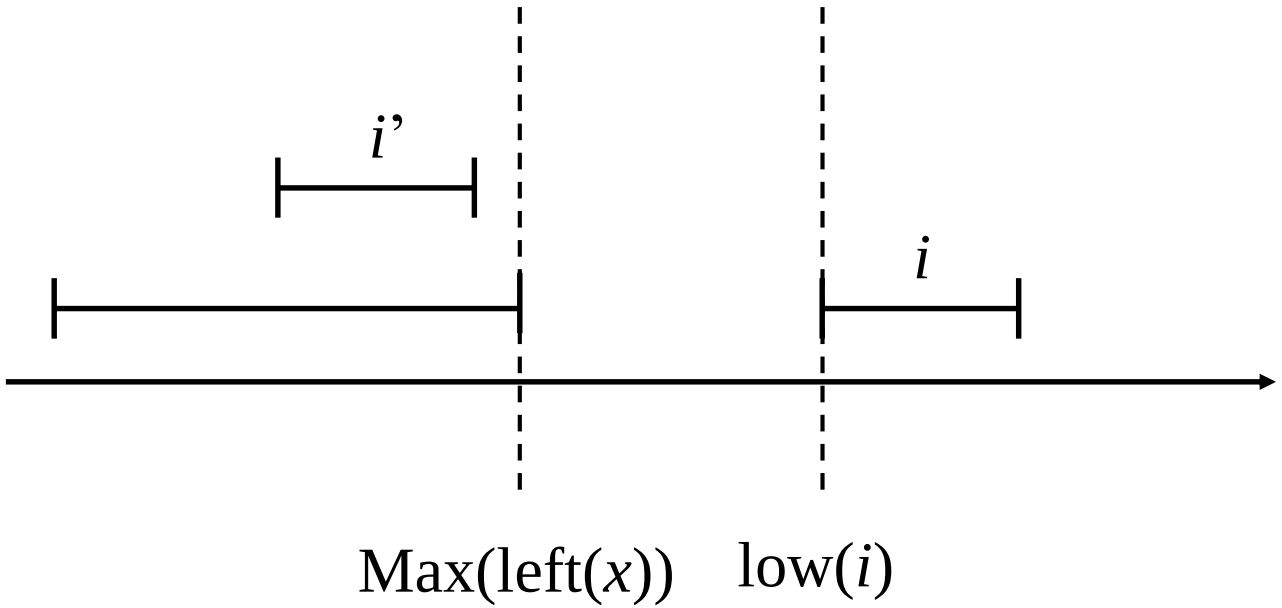
前者の場合は x の左部分木は空であるので成立.

後者の場合，i’ を x の左部分木内の区間とする.

Max(left(x))は x の左部分木中の最大の端点なので

$$\begin{aligned} \mathrm{high}(i') &\leq \mathrm{Max}(\mathrm{left}(x)) \\ &< \mathrm{low}(i) \end{aligned}$$

となり，どの区間も i と重ならない.



1の場合: x の左部分木中の区間で i と重なり合うものが見つかれば探索は終了する.

見つからない場合に, x の右部分木中のどの区間も i と重ならないことを示す.

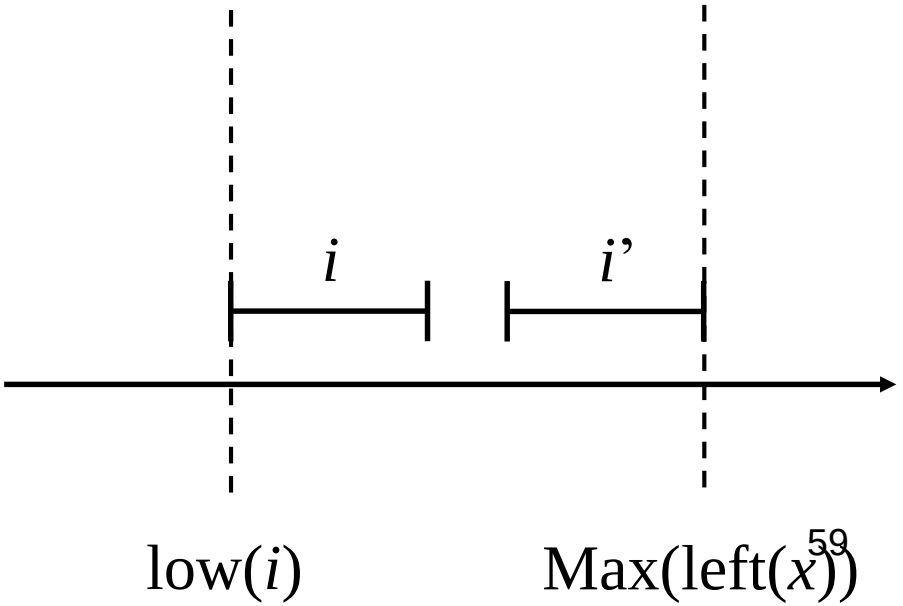
分岐の条件より, $\mathrm{Max}(\mathrm{left}(x)) \geq \mathrm{low}(i)$.

Maxの定義より, x の左部分木中にある区間 i’ が存在して

$$\mathrm{high}(i') = \mathrm{Max}(\mathrm{left}(x)) \geq \mathrm{low}(i)$$

i と i’ は重ならないため

$$\mathrm{high}(i) < \mathrm{low}(i')$$



キーは区間の下端点であり，探索木の性質から，

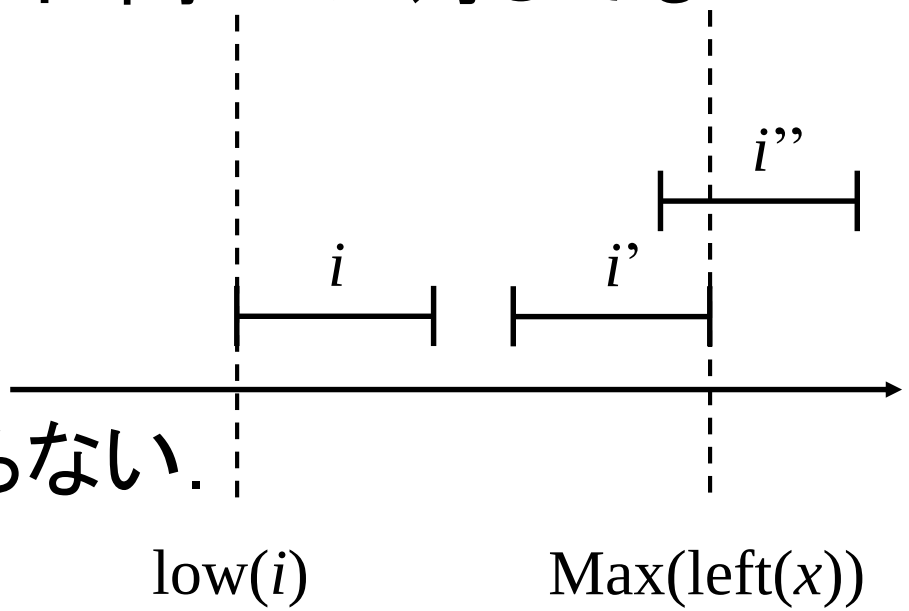
x の右部分木中のどの区間 i'' に対しても

$$\mathrm{low}(i') \leq \mathrm{low}(i'')$$

が成り立つ．つまり

$$\mathrm{high}(i) < \mathrm{low}(i'')$$

となり，i と i'' は重ならない．



この定理から，ある方向に探索して区間が見つからなかった場合は反対側にも解がないことが保障される．