



情報数理科学II

Red-Black Trees

赤黒木

スライド中の数式、プログラムコード等は
<http://opendatastructures.org> から引用したものを含んでいます。

[投票]赤黒木を知っていた?

1. 名前をこの講義で初めて知った
2. 名前は聞いたことがあるが、どのようなものかは知らない
3. どのようなものは知っているが使ったことはない
4. 他の人が作ったライブラリを使ったことがある。
5. 自分で実装したことがある

[投票]次の中で赤黒木を使って実装されているものは?

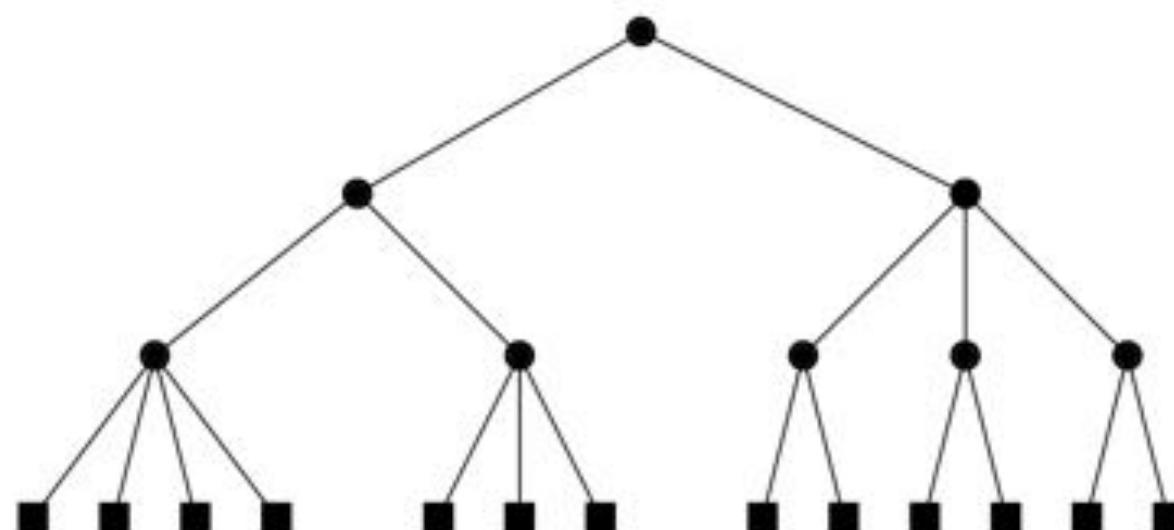
1. C++のmap
2. C++のunordered_map
3. Rubyの配列
4. Pythonのdictionary(辞書)
5. Javascriptのオブジェクト

Red-Black Trees

- 二分木の一種で広く使われる
 - C++ STL(set,map), Java Collection Framework
- 性質
 - n 個の要素を持つ赤黒木の最大の深さは $2 \log n$.
 - $\text{add}(x)$, $\text{remove}(x)$ の最悪実行時間は $O(\log n)$.
 - $\text{add}(x)$, $\text{remove}(x)$ 中に実行するrotationの回数はamortizeすると定数
- 次節では、赤黒木に先立って2-4木を導入する.

9.1 2-4 Treesの定義と性質

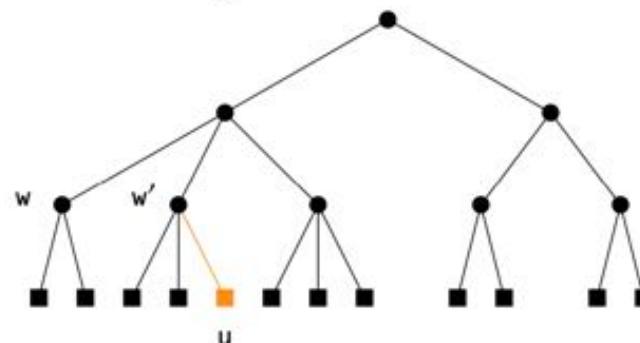
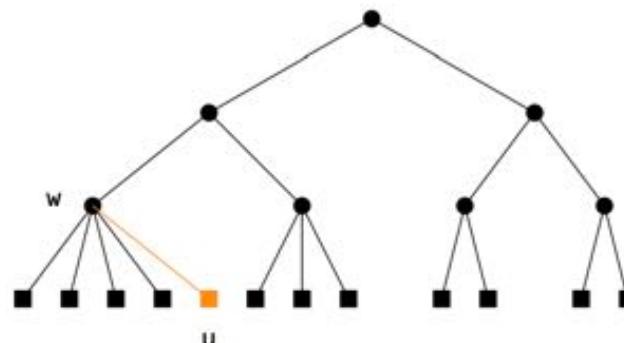
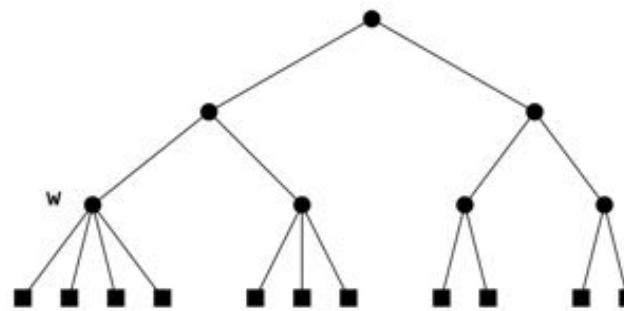
- Property 9.1 : すべてのLeaf(葉)は同じ深さ
- Property 9.2 : すべての内部ノードは2,3,4個の子供を持つ
- Lemma 9.1 : n 個の葉を持つ 2-4 Tree の高さは高々 $\log n$



9.1.1 Adding a Leaf

- leafを追加する親ノードを決める.
- 親ノードの子が2 or 3ならそのまま追加
- 親ノードの子が4の時は、親ノードを2つに分ける。これは親の親のノードに子供を1つ追加する操作になる。再帰的に繰り返して、先祖で子供が4以外のノードを探す。
- ルートに至った場合は、ルートを2つに分割して、その2つを子とするノードを作り、新たなルートとする。

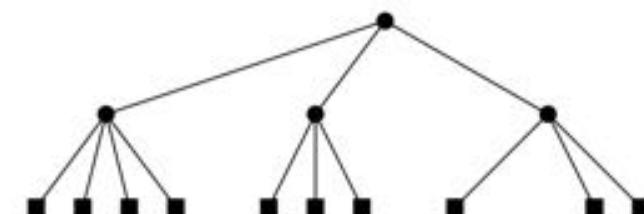
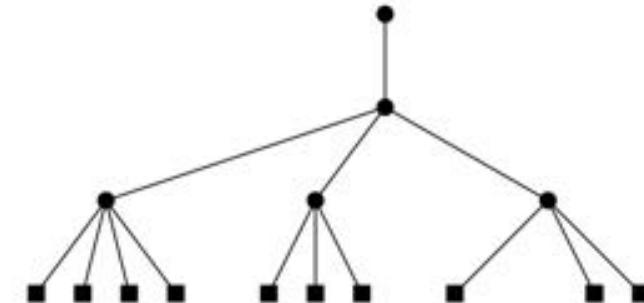
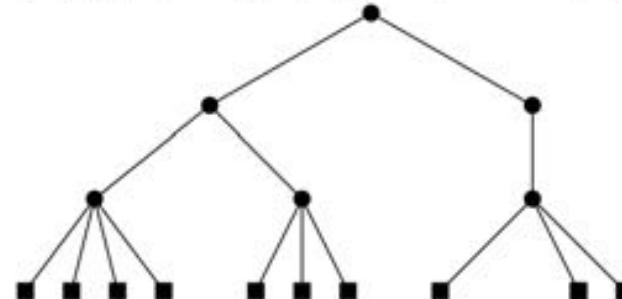
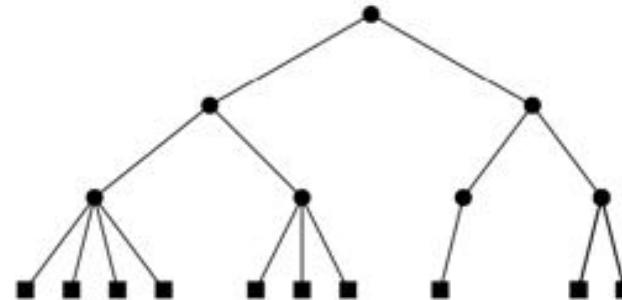
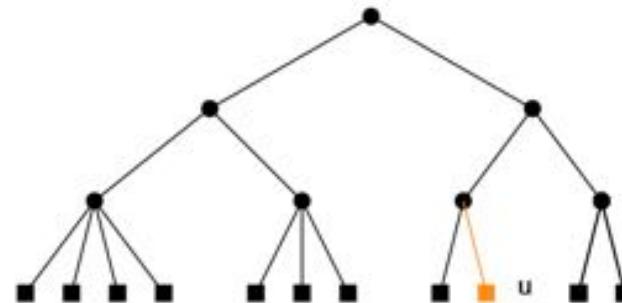
子の追加と分割



9.1.2 Removing a Leaf

- 削除するノードの親ノードの子が3 or 4ならそのまま削除
- 親ノードの子が2の時, 親ノードの前後の兄弟で子ノードが3 or 4のものがあつたら, そこから子供を一人移す.
- なかつたら, 親ノードの前後の兄弟で子ノードが2のものとmergeして一つのノードとする.
- その結果, 親ノードの親ノードの子が1になりそうになつたら, 再帰的に繰り返す.
- rootの子供が1つになつたら, rootを削除

ノードの削除



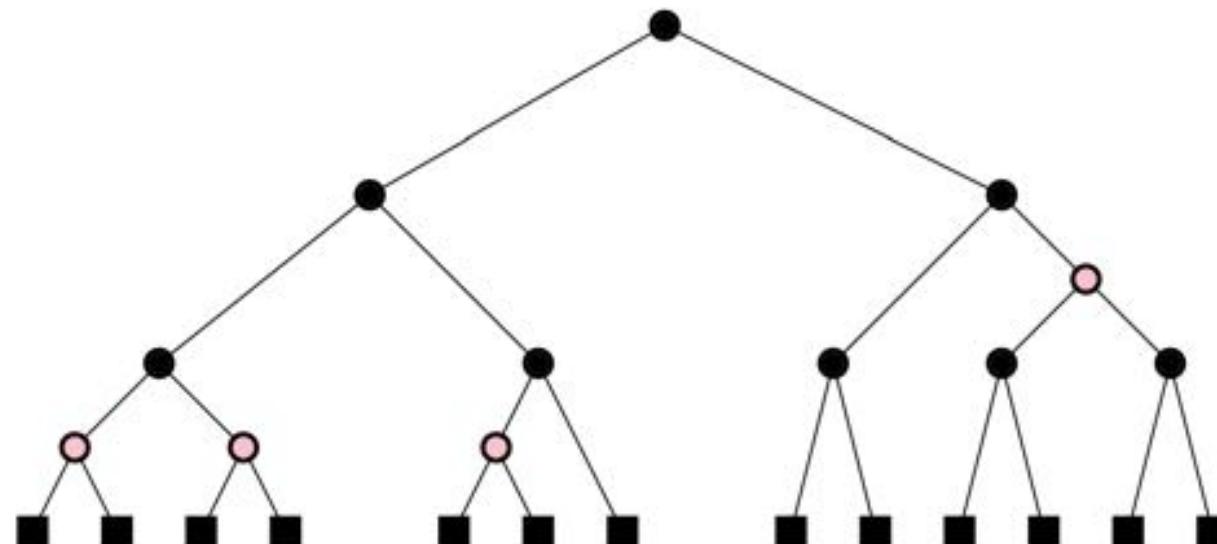
9.2 Red-Black Tree

```
class RedBlackNode : public BSTNode<Node, T> {  
    friend class RedBlackTree<Node, T>;  
    char colour;  
};  
int red = 0;  
int black = 1;
```

- Binary Search Treeの各ノードに colour(この教科書ではなぜかイギリス英語)の属性を追加
- colourは赤と黒の二種類

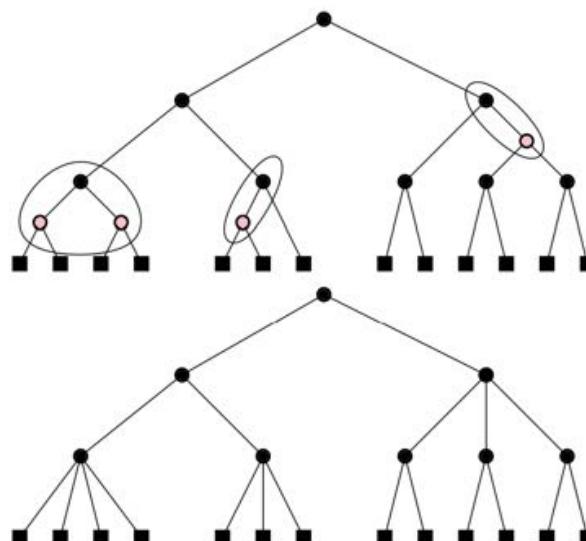
Red-Black Treeのproperty

- Property 9.3: rootからleafまでのpath中のblack nodeの数はすべてのleafで同じ
- Property 9.4 : red node同士が親子となることはない.
- とりあえず, rootは黒, leafも黒でcolouringすることにする.



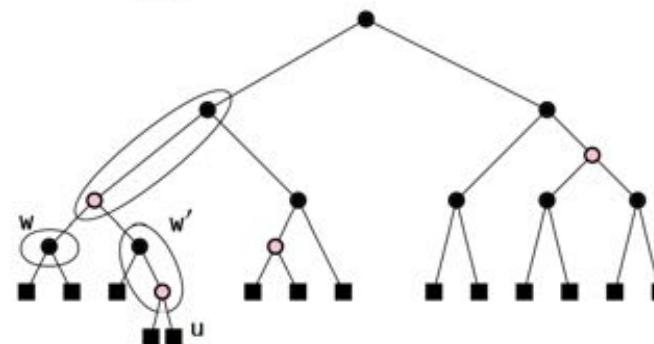
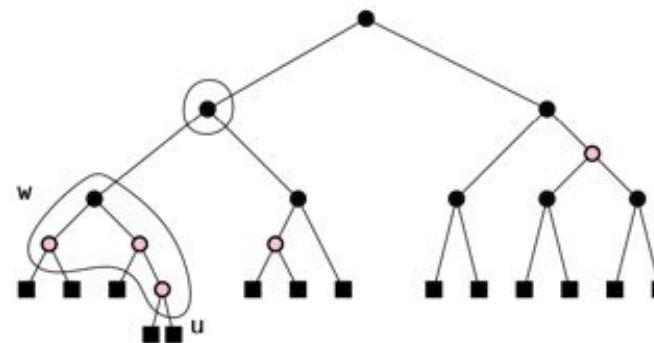
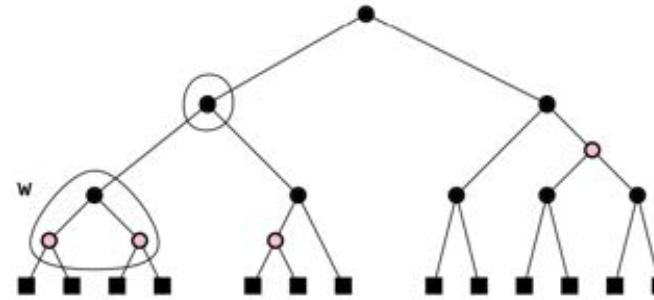
9.2.1 Red-Black Trees and 2-4 Trees

- Red-Black Treeは以下の操作で2-4 Treeに対応する
 - red nodeはすべて削除する.
 - red nodeの子はすべて直接red nodeの親のblack nodeの子に付け替える.
 - この操作で残るblack nodeの子の数は2-4



Lemma 9.2
 n 個のノードからなるred-black tree の高さは高々 $2 \log n$

2-4木のadd, removeのsimulate

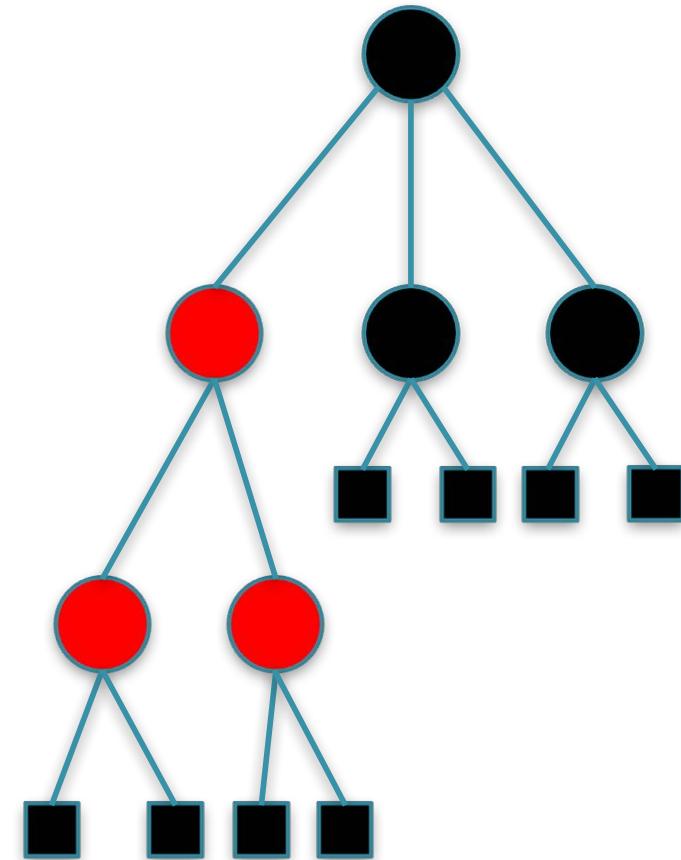


9.2.2 Left-Leaning Red-Black Trees

- Red-Black Treeの定義だけでは場合分けが多くなりすぎる.
- 以下の制約(left-leaning)を加えると操作がシンプルになる.
- Property 9.5 : すべてのノードuでu.leftがblackならu.rightもblack
- この制約で2-4木のノードに対応するred-black木の構造が一意に決定

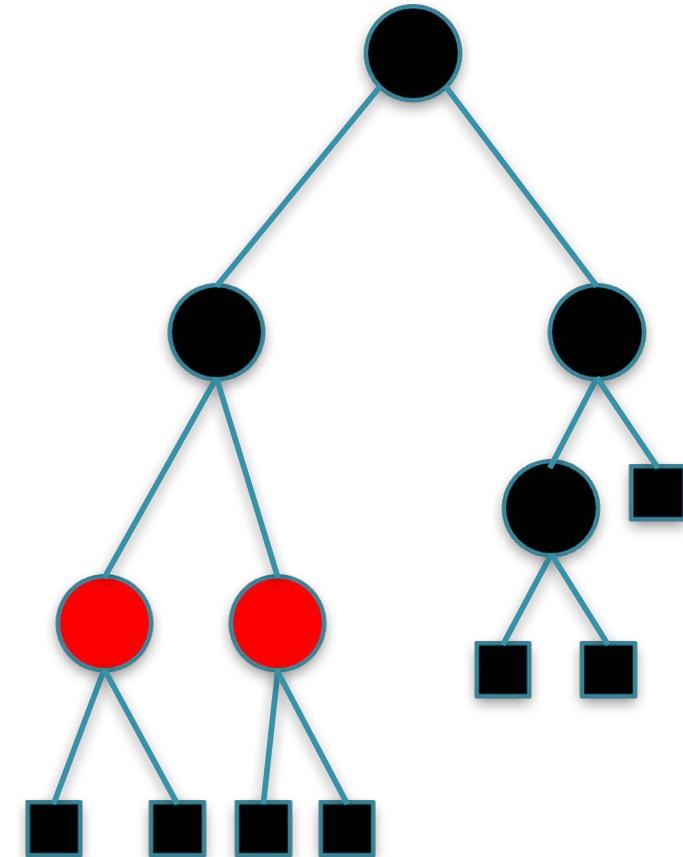
[投票]右図はLeft-Leaning Red-Black Tree?

- 1. はい
- 2. Red-Black TreeだがLeft-Leaningではない
- 3. Red-Black Treeではない



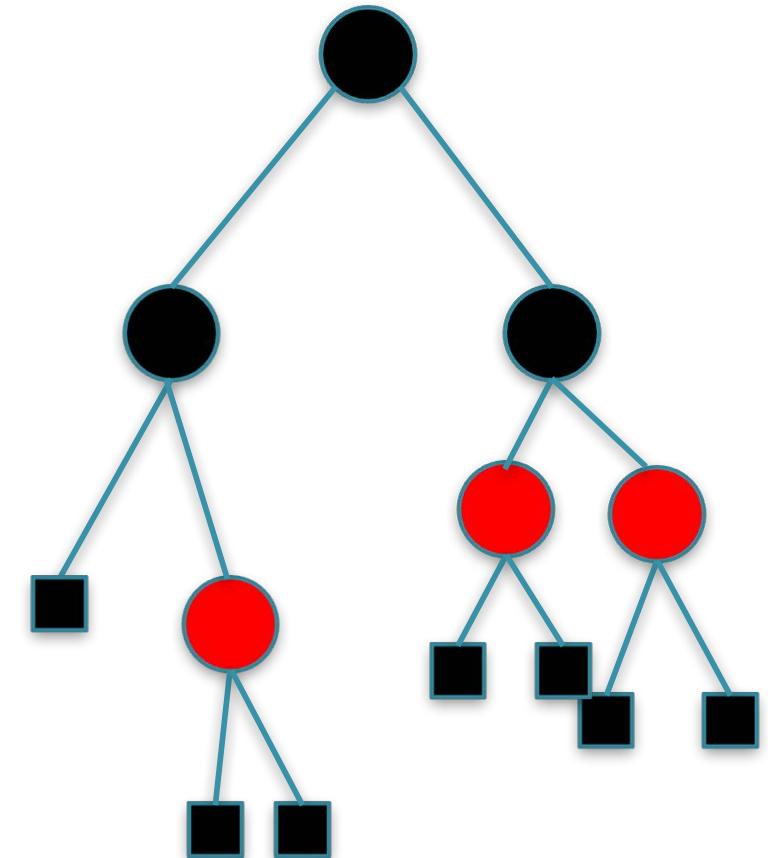
[投票]右図はLeft-Leaning Red-Black Tree?

- 1. はい
- 2. Red-Black TreeだがLeft-Leaningではない
- 3. Red-Black Treeではない



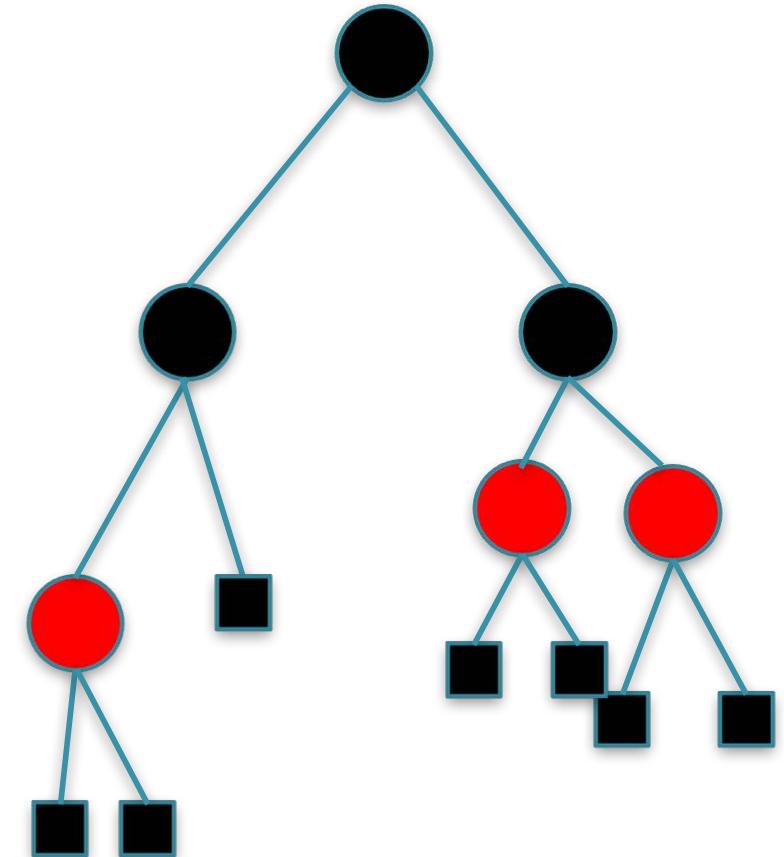
[投票]右図はLeft-Leaning Red-Black Tree?

1. はい
2. Red-Black TreeだがLeft-Leaningではない
3. Red-Black Treeではない



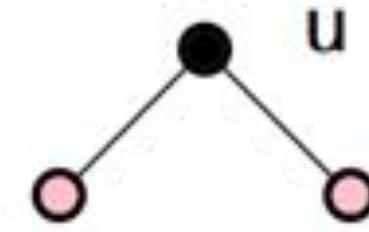
[投票]右図はLeft-Leaning Red-Black Tree?

- 1. はい
- 2. Red-Black TreeだがLeft-Leaningではない
- 3. Red-Black Treeではない

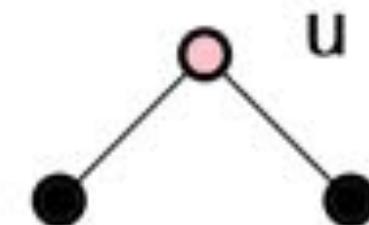


準備(pushBlack)

```
void pushBlack(Node *u) {  
    u->colour--;  
    u->left->colour++;  
    u->right->colour++;  
}
```

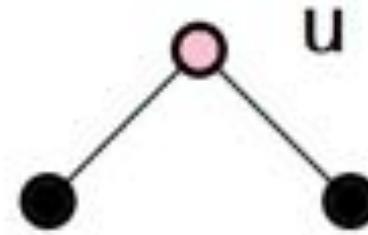


pushBlack(u)
↓

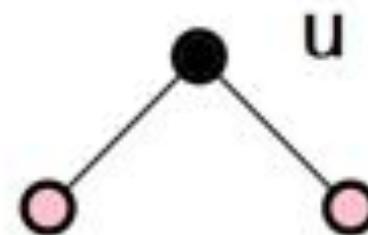


準備(pullBlack)

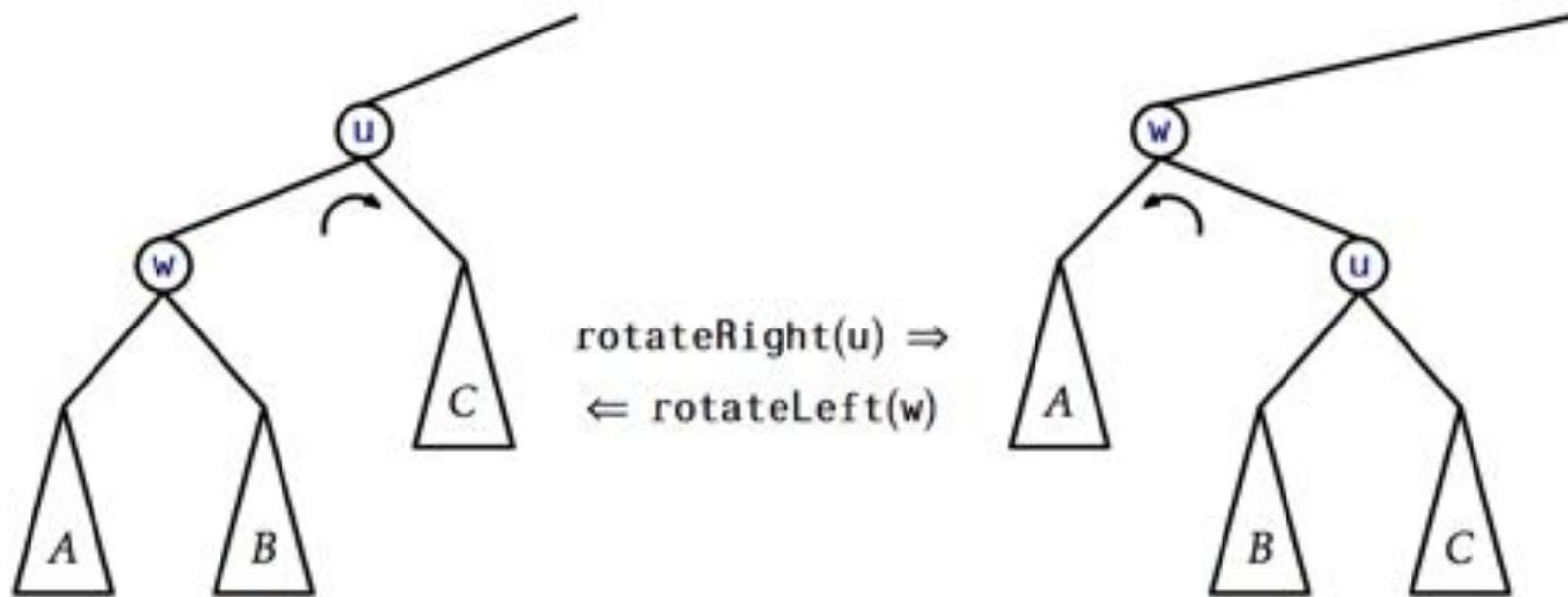
```
void pullBlack(Node *u) {  
    u->colour++;  
    u->left->colour--;  
    u->right->colour--;  
}
```



pullBlack(u)
↓

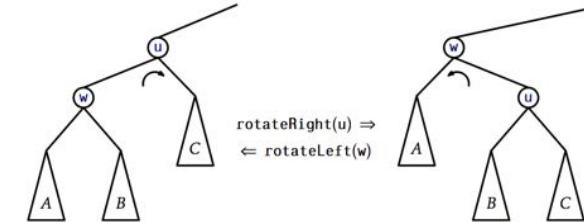


準備(rotateLeft, rotateRight)



準備(rotateRight)

```
void rotateRight(Node *u) {  
    Node *w = u->left;  
    w->parent = u->parent;  
    if (w->parent != nil) {  
        if (w->parent->left == u) {  
            w->parent->left = w;  
        } else {  
            w->parent->right = w;  
        }  
    }  
    u->left = w->right;  
    if (u->left != nil) {  
        u->left->parent = u;  
    }  
    u->parent = w;  
    w->right = u;  
    if (u == r) { r = w; r->parent = nil; }  
}
```



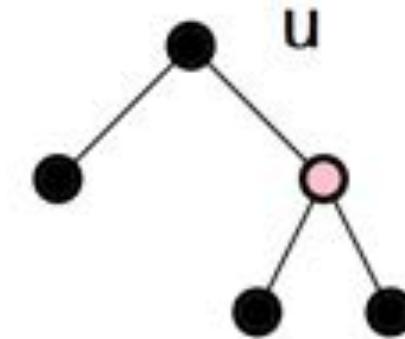
rotateRight(*u*) \Rightarrow
 \Leftarrow rotateLeft(*w*)

準備(rotateLeft)

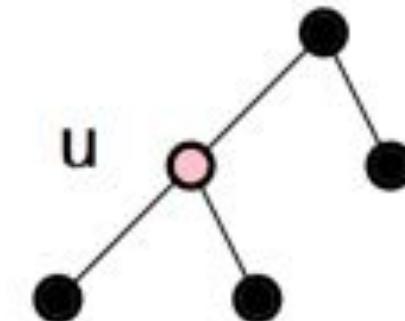
```
void rotateLeft(Node *u) {  
    Node *w = u->right;  
    w->parent = u->parent;  
    if (w->parent != nil) {  
        if (w->parent->left == u) {  
            w->parent->left = w;  
        } else {  
            w->parent->right = w;  
        }  
    }  
    u->right = w->left;  
    if (u->right != nil) {  
        u->right->parent = u;  
    }  
    u->parent = w;  
    w->left = u;  
    if (u == r) { r = w; r->parent = nil; }  
}
```

準備(flipLeft)

```
void flipLeft(Node *u) {  
    swapcolours(u, u->right);  
    rotateLeft(u);  
}
```

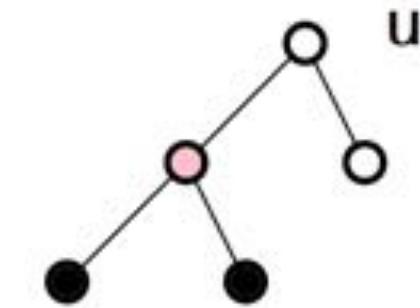


flipLeft(**u**)

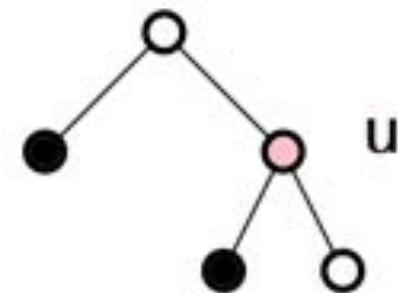


準備(flipRight)

```
void flipRight(Node *u) {  
    swapcolours(u, u->left);  
    rotateRight(u);  
}
```



flipRight(u)
↓



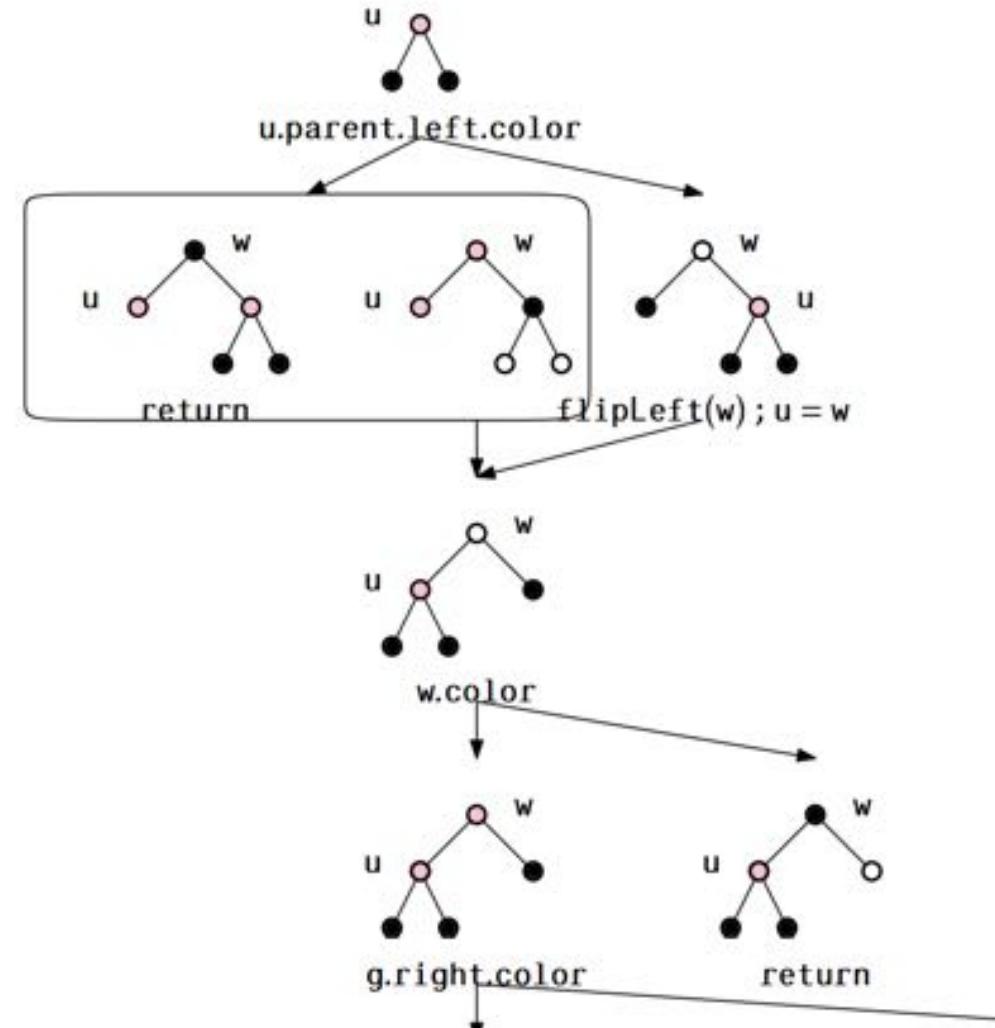
9.2.3 Addition

```
bool add(T x) {
    Node *u = new Node();
    u->left = u->right = u->parent = nil;
    u->x = x;
    u->colour = red;
    bool added = BinarySearchTree<Node, T>::add(u);
    if (added)
        addFixup(u);
    return added;
}
```

- red-red edgeができるかもしれない.
- Left-leaning propertyを満たさなくなるかもしれない.
- addFixupで修正する. uがredという前提

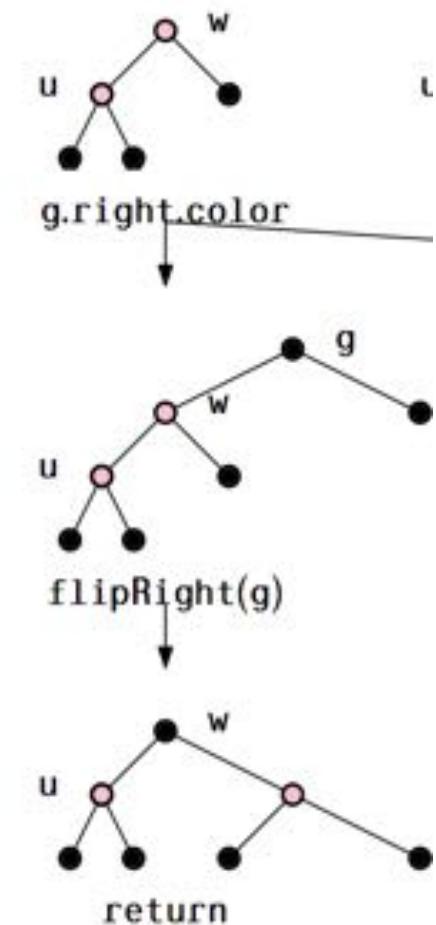
addFixupの動作(I)

- uがrootなら
redからblack
に
- uにはparent
がいる.wとする.
- uがwの左右
どちらの子
供か, wの
色で条件分
岐



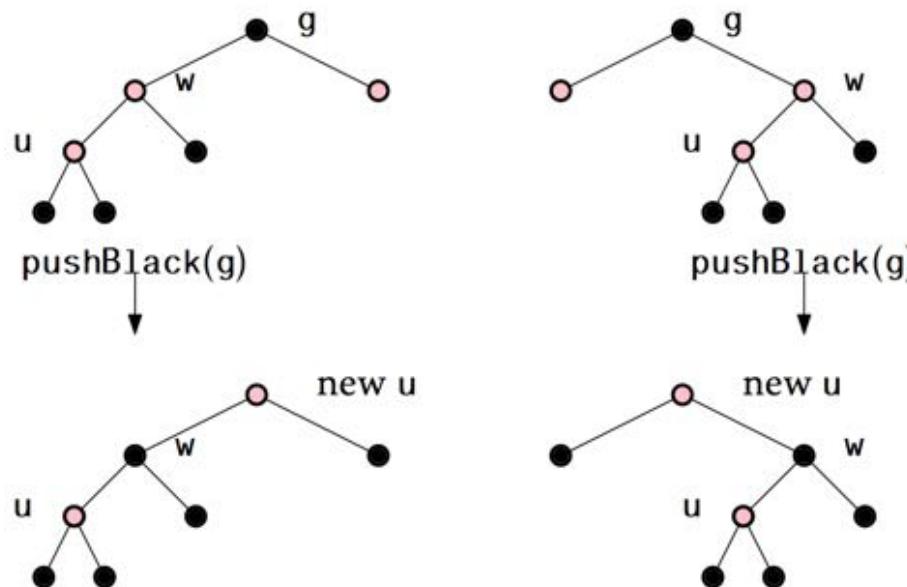
addFixupの動作(2)

- uがred, wがredでuがwのleft childとする.
- wの親をgとする.
- もともと red-red edgeはなかったので, gの色は黒のはず
- gのrightが黒の時はwはleft child
- flipRight(g)で解決



addFixupの動作(3)

- gの子供が両方共redの時, wがleft childか right childかで分岐



操作後にgが
blackから
redに変わ
るので,
更に
addFixup

addFixupのコード

```
void addFixup(Node *u) {
    while (u->colour == red) {
        if (u == r) { // u is the root - done
            u->colour = black;
            return;
        }
        Node *w = u->parent;
        if (w->left->colour == black) { // ensure left-leaning
            flipLeft(w);
            u = w;
            w = u->parent;
        }
        if (w->colour == black)
            return; // no red-red edge = done
        Node *g = w->parent; // grandparent of u
        if (g->right->colour == black) {
            flipRight(g);
            return;
        } else {
            pushBlack(g);
            u = g;
        }
    }
}
```

addFixupの実行時間は $O(\log n)$.

9.2.4 Removal

- `remove(x)`はどのred-black treeでも難しい.
- `BinarySearchTree`の`remove`の時, u の親 w の子供が u だけになって, w を外して w の親が直接 u を子供に持つようにする.
- w が黒の時に, black-height propertyの侵害
 - u の色が赤の時に黒になると, left-leaning propertyを侵害することもある.
 - u が黒の時には u を一旦 double-blackにするが, これは後で解決する必要がある.
- `removeFixup(u)`で解決

remove のコード

```
bool remove(T x) {
    Node *u = findLast(x);
    if (u == nil || compare(u->x, x) != 0)
        return false;
    Node *w = u->right;
    if (w == nil) {
        w = u;
        u = w->left;
    } else {
        while (w->left != nil)
            w = w->left;
        u->x = w->x;
        u = w->right;
    }
    splice(w);
    u->colour += w->colour;
    u->parent = w->parent;
    delete w;
    removeFixup(u);
    return true;
}
```

removeFixup のコード

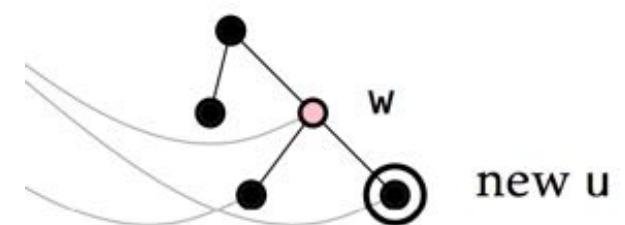
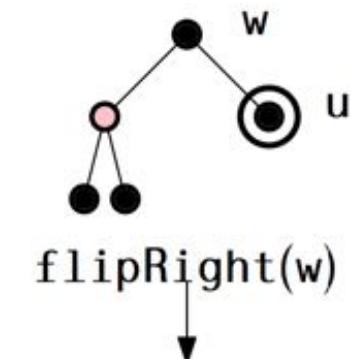
```
void removeFixup(Node *u) {
    while (u->colour > black) {
        if (u == r) {
            u->colour = black;
        } else if (u->parent->left->colour == red) {
            u = removeFixupCase1(u);
        } else if (u == u->parent->left) {
            u = removeFixupCase2(u);
        } else {
            u = removeFixupCase3(u);
        }
        if (u != r) { // restore left-leaning property, if
necessary
            Node *w = u->parent;
            if (w->right->colour == red && w->left->colour ==
black) {
                flipLeft(w);
            }
        }
    }
}
```

removeFixupCase1(u)

```
Node* removeFixupCase1(Node *u) {  
    flipRight(u->parent);  
    return u;  
}
```

- uはblackあるいはdouble-blackで、parentのright childの時
- flipRighthして続行

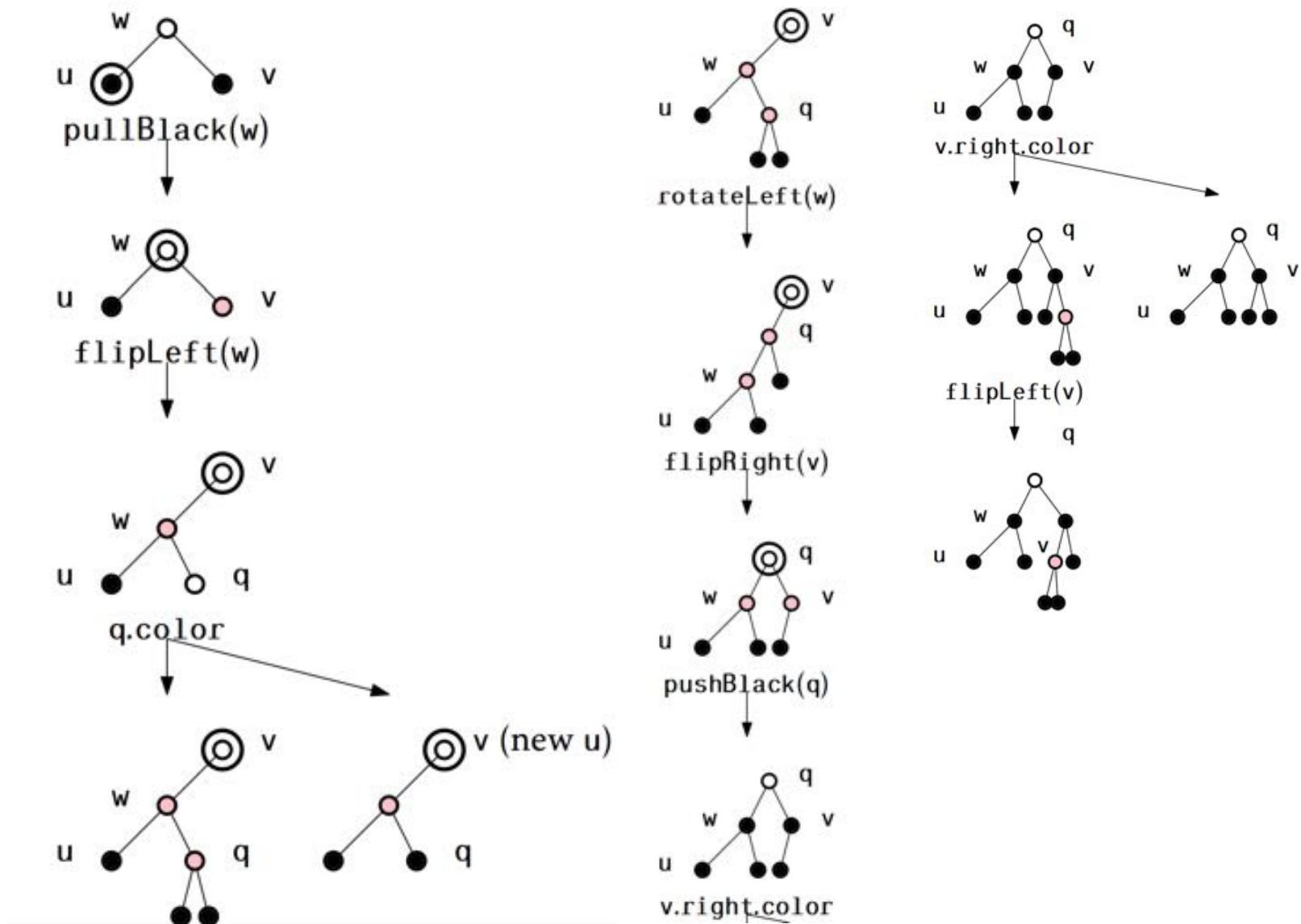
removeFixupCase1(u)



removeFixupCase2(u)

```
Node* removeFixupCase2(Node *u) {
    Node *w = u->parent;
    Node *v = w->right;
    pullBlack(w); // w->left
    flipLeft(w); // w is now red
    Node *q = w->right;
    if (q->colour == red) { // q-w is red-red
        rotateLeft(w);
        flipRight(v);
        pushBlack(q);
        if (v->right->colour == red)
            flipLeft(v);
        return q;
    } else {
        return v;
    }
}
```

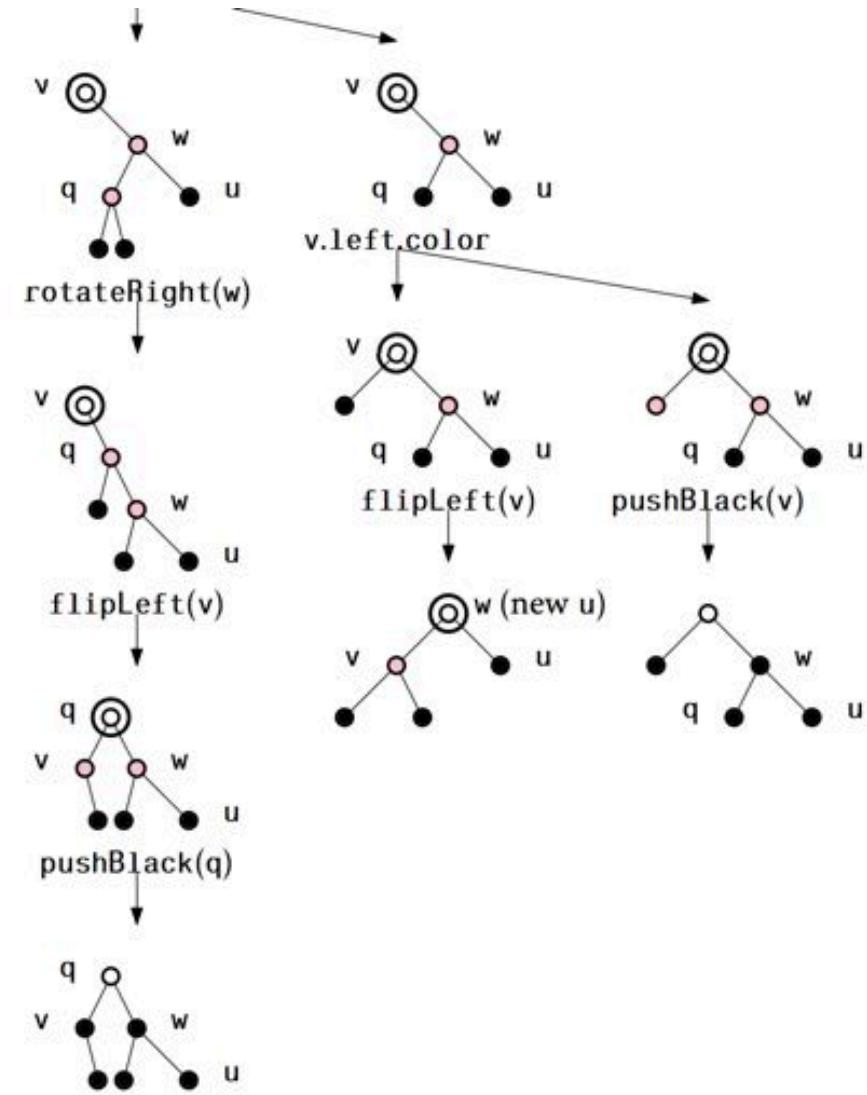
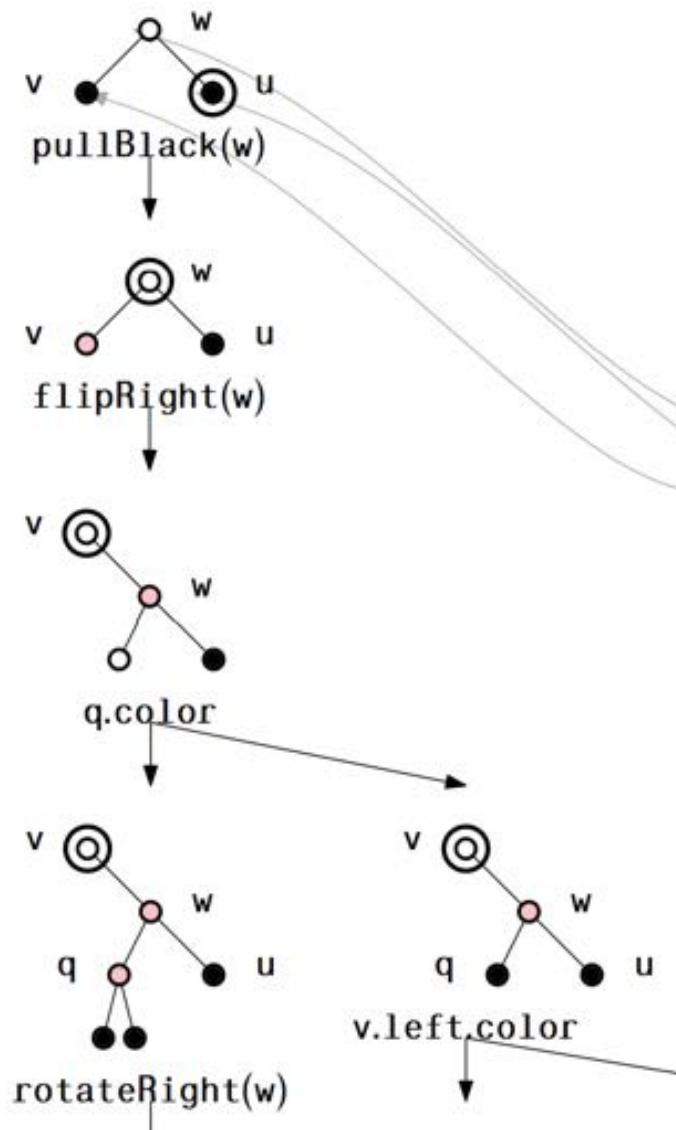
removeFixupCase2



removeFixupCase3(u)

```
Node* removeFixupCase3(Node *u) {
    Node *w = u->parent;
    Node *v = w->left;
    pullBlack(w);
    flipRight(w);           // w is now red
    Node *q = w->left;
    if (q->colour == red) { // q-w is red-red
        rotateRight(w);
        flipLeft(v);
        pushBlack(q);
        return q;
    } else {
        if (v->left->colour == red) {
            pushBlack(v); // both v's children are red
            return v;
        } else {          // ensure left-leaning
            flipLeft(v);
            return w;
        }
    }
}
```

removeFixupCase3



9.3 Summary

- Theorem 9.1 : RedBlackTreeはSSetインターフェースを実現
 - $\text{add}(x)$, $\text{remove}(x)$, $\text{find}(x)$ は $O(\log n)$ 時間(最悪)で実現
- Theorem 9.2 : 空の状態から $\text{add}(x)$ と $\text{remove}(x)$ を m 回実行した時の $\text{addFixup}()$, $\text{removeFixup}(u)$ のコストは $O(m)$

9.4 Discussion and Exercises

- 歴史
- 赤黒木のいろいろなバリエーション
- AVL木との関係