

アルゴリズムとデータ構造

平衡木（その2）

目次

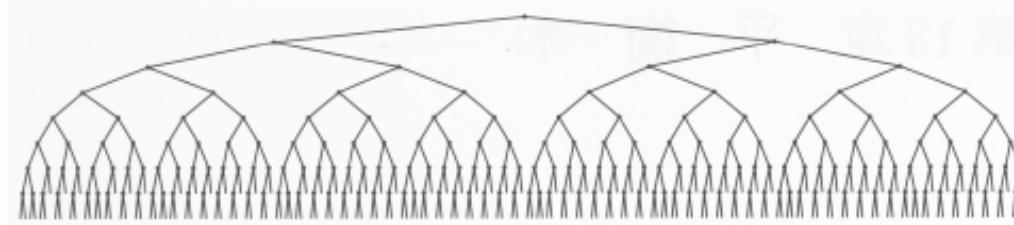
- 2-3-4木
- 赤黒木

2分探索木(復習)

- 挿入操作が高くなつてという問題を解決するためには、記号表実現の基礎として**木構造**を用いる
- 2分探索木(**binary search tree, BST**):
 - 内部節点にキーが置かれた2分木で、次の性質を満たすものである
 - 節点が置かれたキーより小さい(あるいは等しい)キーを持つ項目はすべてその節点の左部分木の中にある
 - 節点が置かれたキーより大きい(あるいは等しい)キーを持つ項目はすべてその節点の右部分木の中にある

完全に平衡がとれた2分探索木（復習）

- 2分探索木
 - 最悪の場合、生成の手間が $2^{\text{乗時間}}$ 、探索の手間が線形時間になる
 - すでに整列しているファイルを挿入するとき
 - キーが大小交互になっていて、これを挿入するとき
- 完全に平衡がとれた2分探索木
 - 木の高さが $\lg(N+1)$ (N : 内部節点の数、性質5.8)



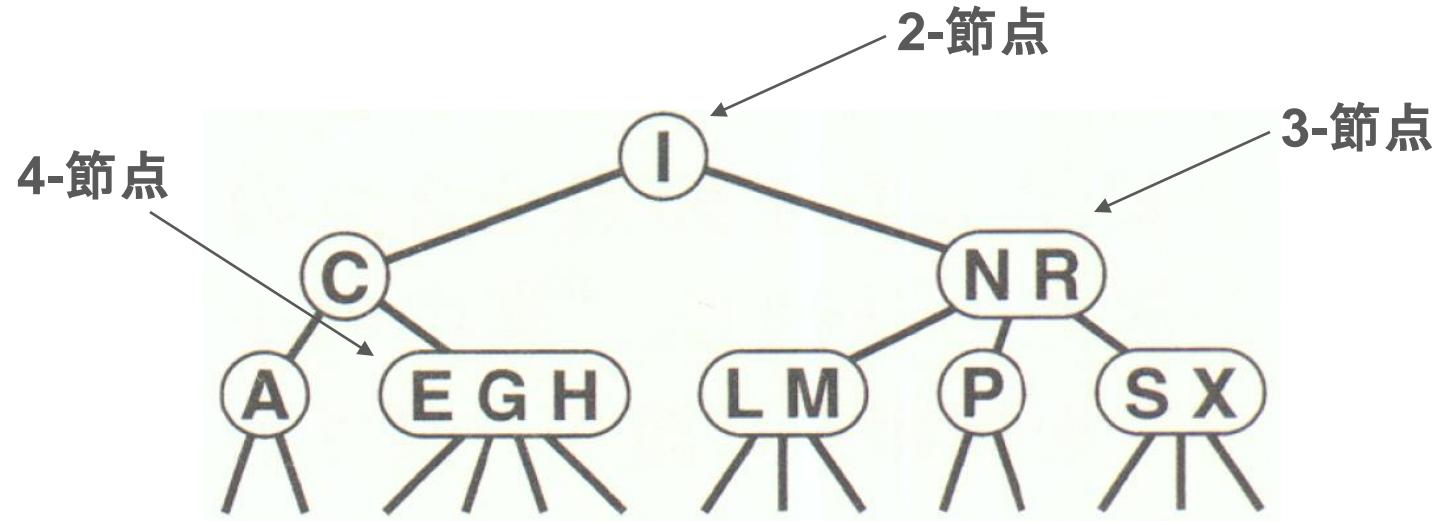
平衡木（復習）

- 平衡木
 - 木の高さが、高々 $O(\lg N)$ の 2 分探索木 (N : 内部節点数)
- 節点を挿入/削除した際に、常に木の高さが $O(\lg N)$ になるように、2 分探索木を作る
 - ランダム化 \rightarrow ランダム化 2 分探索木 (ランダム化 BST)
 - 均し \rightarrow スプレイ 2 分探索木 (スプレイ BST)
 - 最適化 \rightarrow **2-3-4木, 赤黒木**

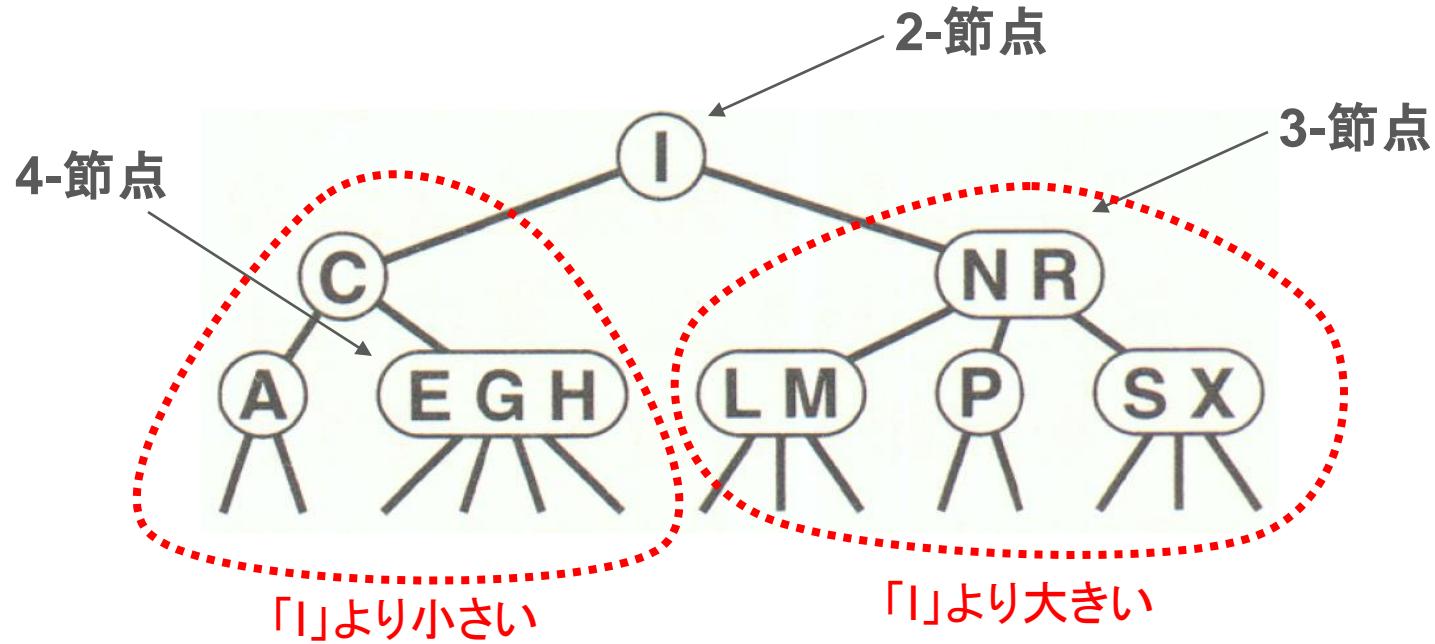
2-3-4木

- 2分探索木を少し改変した木を考えよう
 - 3種類の節点を考える
 - 2-節点:
 - 通常の2分探索木の節点. キーを1つだけ保持できる.
2つのリンクを持つ
 - 3-節点
 - キーを2つ保持し, 3つのリンクを持つ節点
 - 4-節点
 - キーを3つ保持し, 4つのリンクを持つ節点

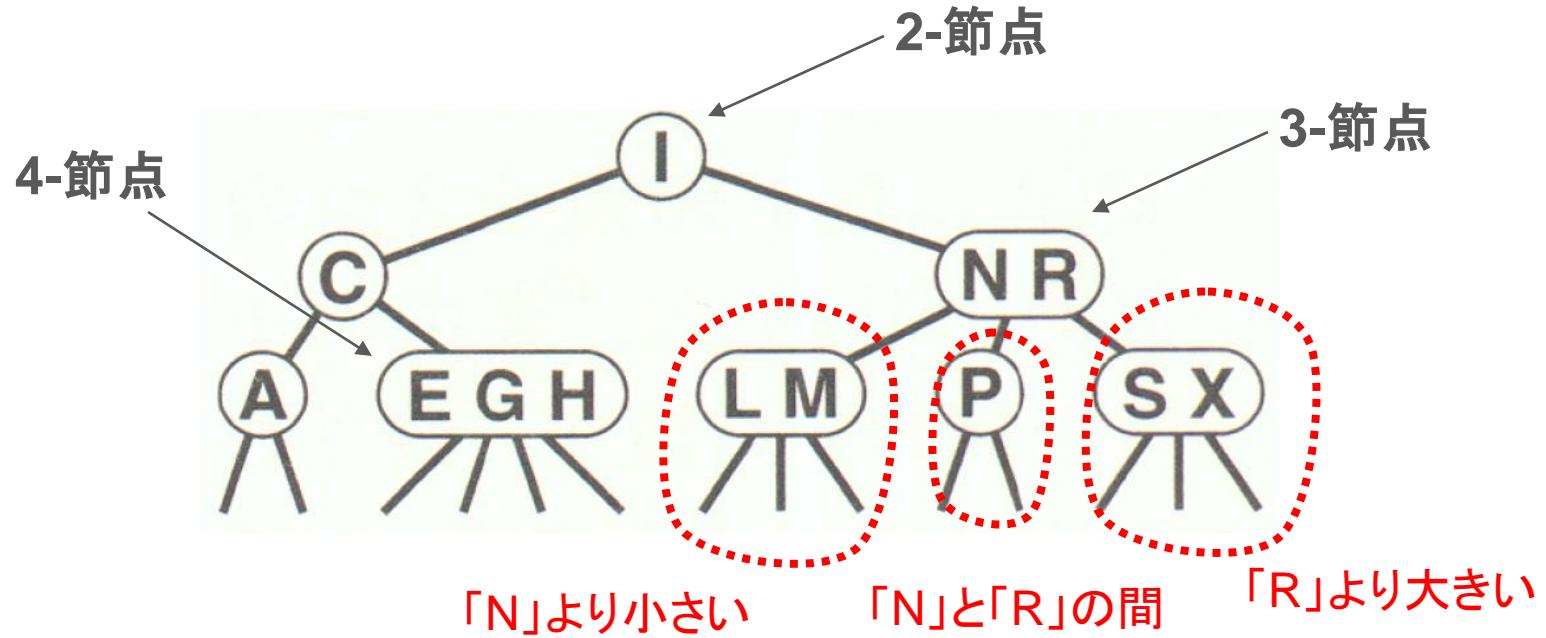
2-3-4木



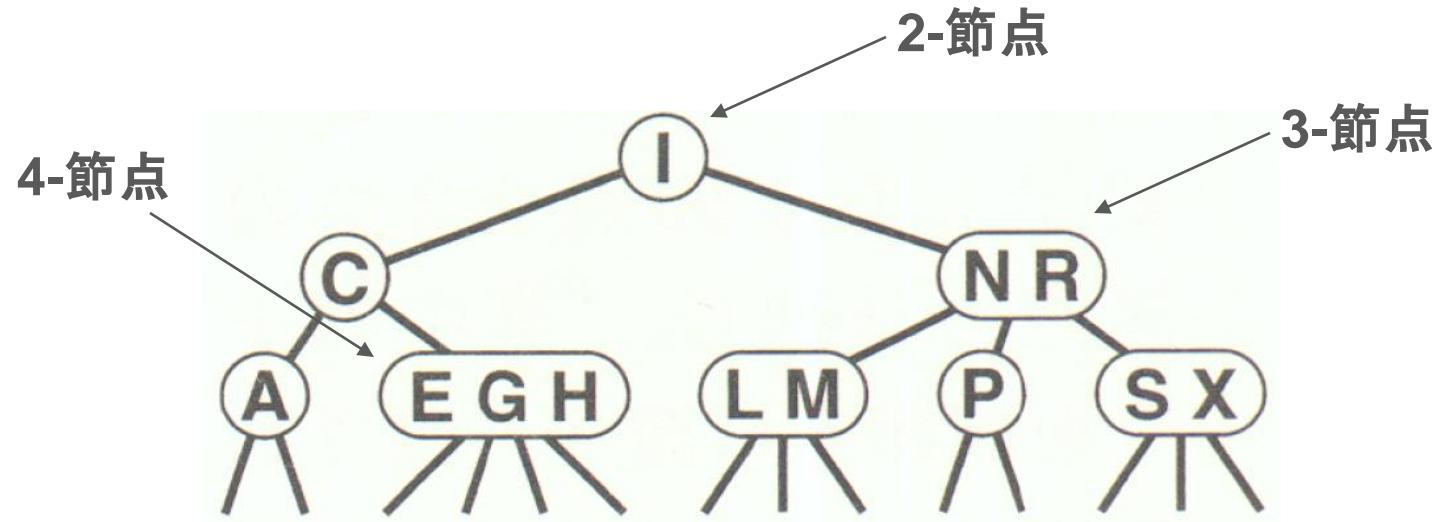
2-3-4木



2-3-4木



2-3-4木

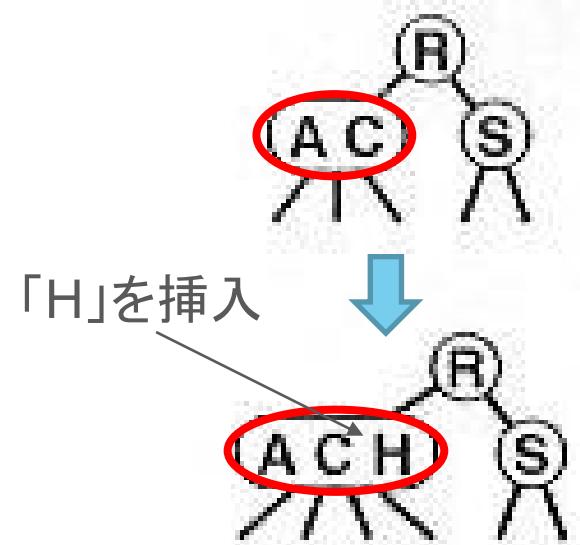
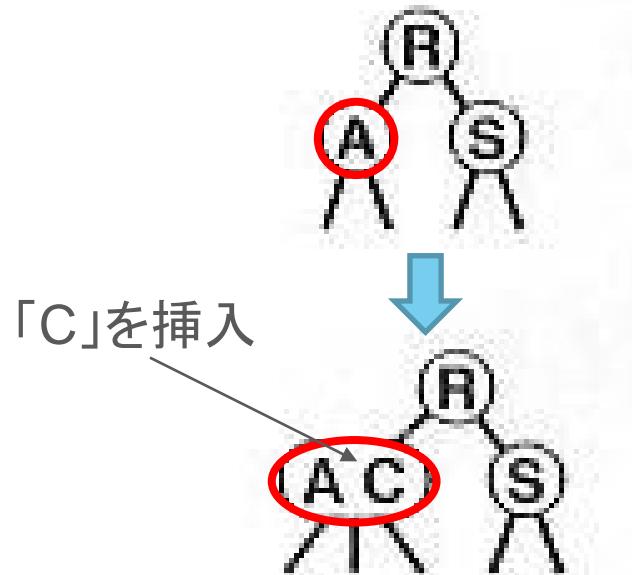


2-3-4木

- 2-3-4木(2-3-4 search tree)は、空か3種類の節点からなる木である
 - 2-節点は1個のキーとそれより小さいキーを持つ木への左リンクとそれより大きいキーを持つ木への右リンクを持つ
 - 3-節点は2つのキーと、2個のキーより小さい全てのキーを持つ木への左リンク、2個のキーの間にある全てのキーへの中央リンク、2個のキーより大きい全てのキーを持つ木への右リンクを持つ
 - 4-節点は3個のキーと4つのリンクを持つ。それぞれ3個のキーで決まる4つの区間へのリンクである
- 平衡2-3-4木は、根から外部節点への距離が全て等しい2-3-4木である

挿入

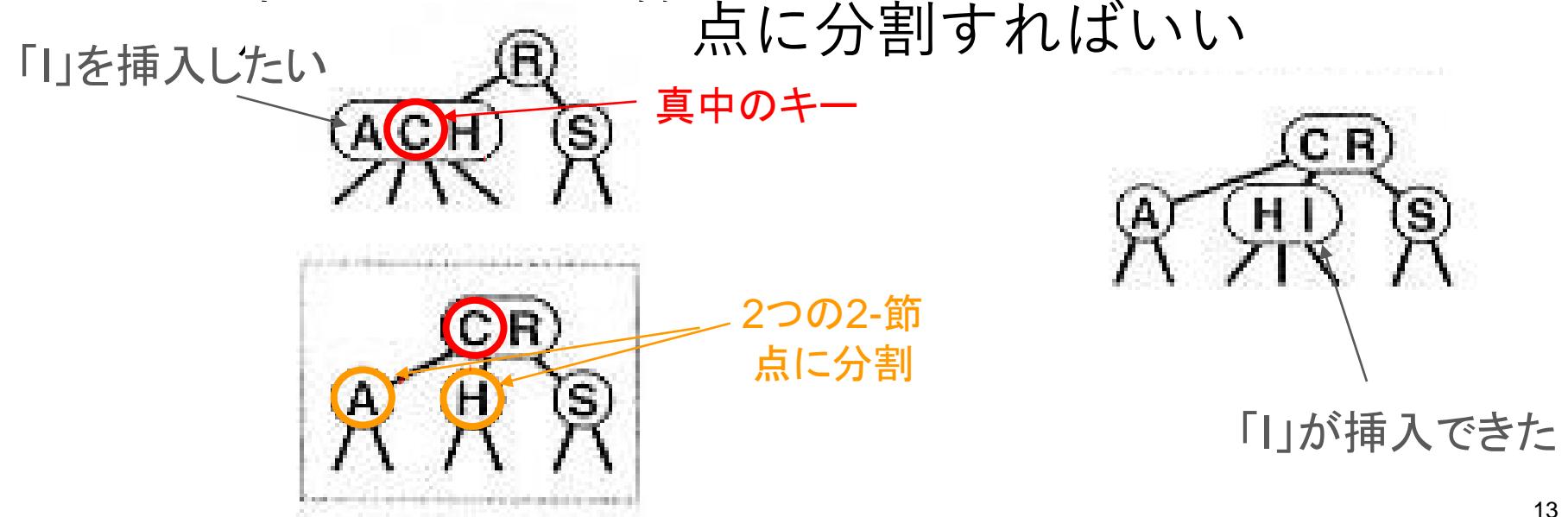
- 2-3-4木の底に、新たな節点を挿入することを考えよう
 - 挿入する位置が2-節点だったら→3-節点に変えればいい
 - 挿入する位置が3-節点だったら→4-節点に変えればいい



他の部分を変化させることなく、節点の種類を変えて挿入すればよい

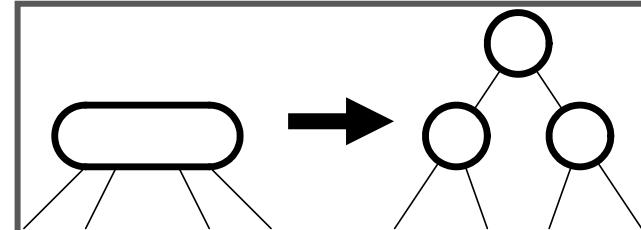
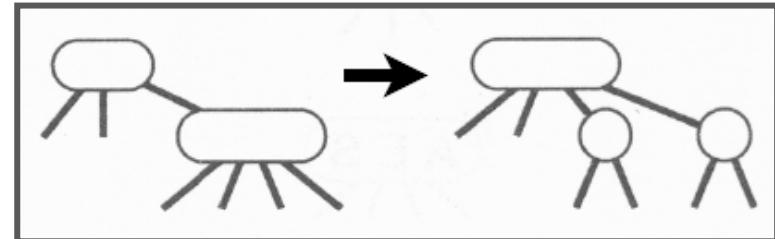
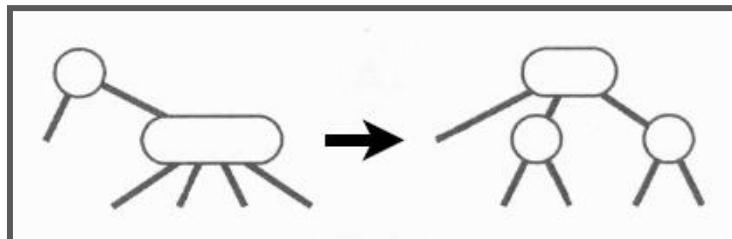
挿入

- 挿入する位置が4-節点だったら？
 - これ以上、その節点に新たなキーを挿入できない
 - 4-節点の真中のキーを上の節点に上げて、2-節点に分割すればいい



挿入

- でも、もし4-節点の上が、さらに4-節点だったら？
 - 上にある4-節点から、さらにその上に、予め中央のキーを上げてあればいい
 - そこで、ある項目を挿入するとき、探索経路中で、4-節点に出合ったら、予め中央のキーを上に上げ、2つの2-節点に分割しておけばいい
 - 根が4-節点のときは、3個の2-節点とする
 - そうすれば、必ず、キーを挿入すべき節点は、2-節点か3-節点になっている！
 - トップダウン2-3-4木

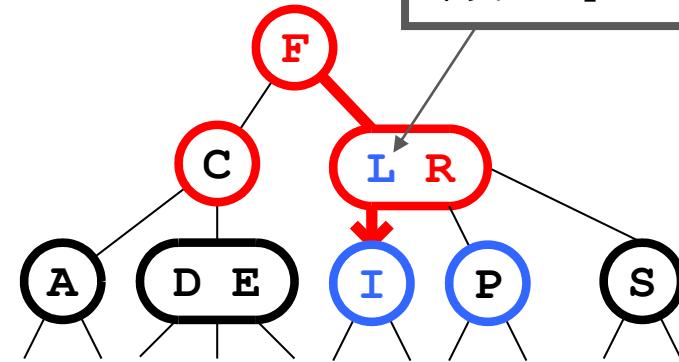
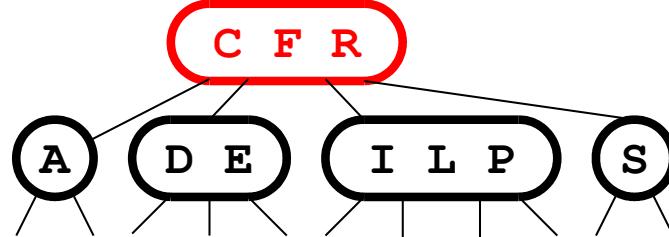


トップダウン2-3-4木への挿入

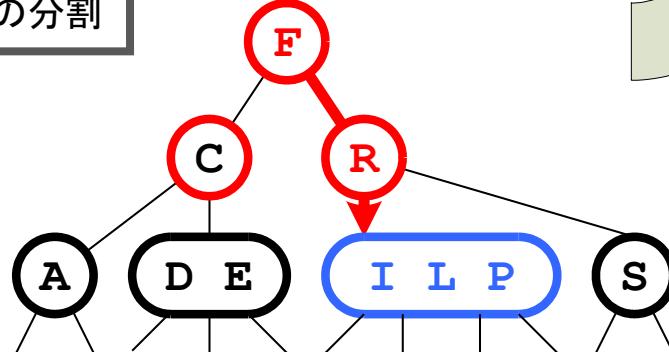
「K」の挿入

探索途中で、4-節点に出合ったら、その度に、
4-節点の分割を行う

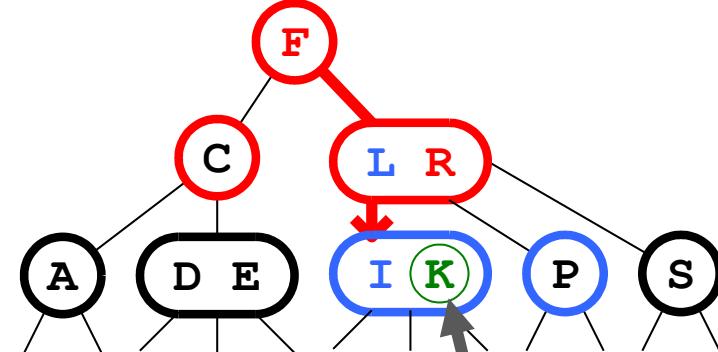
根から順に、「K」の挿入位置を探す



4-節点「CFR」に
出合った
→ 節点の分割



4-節点「ILP」に出合った → 節点の分割

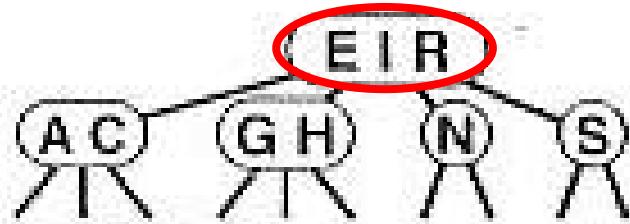


葉に到達、「K」を挿入する

トップダウン2-3-4木への挿入

「x」の挿入

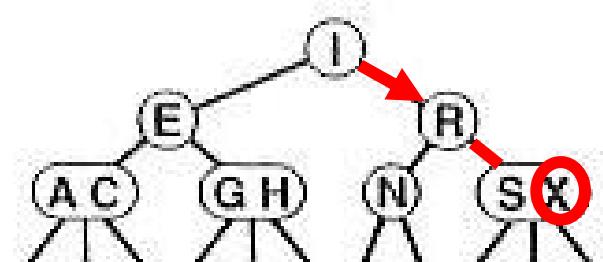
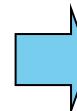
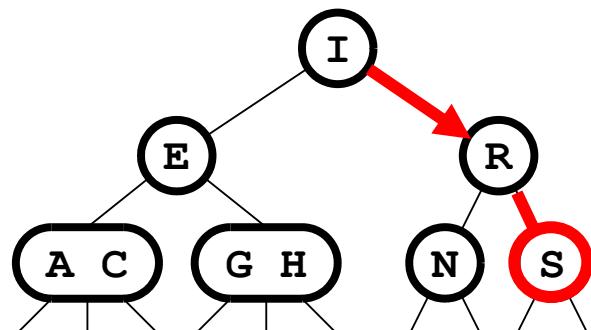
探索途中で、4-節点に出合ったら、その度に、
4-節点の分割を行う



根から順に、「x」の挿入位置を探す



4-節点に出合った→ 節点の分割

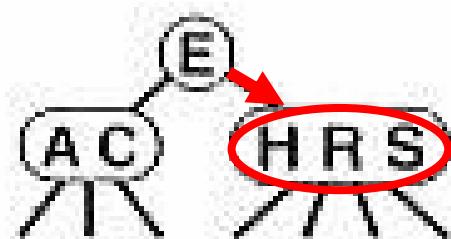


葉に到達、「x」を挿入する

トップダウン2-3-4木への挿入

「E」の挿入

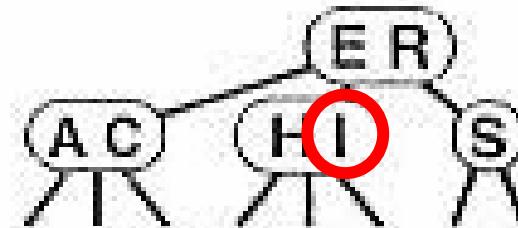
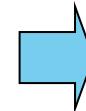
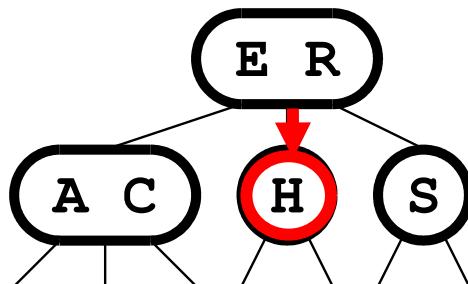
探索途中で、4-節点に出合つたら、その度に、
4-節点の分割を行う



根から順に、「E」の挿入位置を探す
「E」よりも「E」の方が大きいから右へ

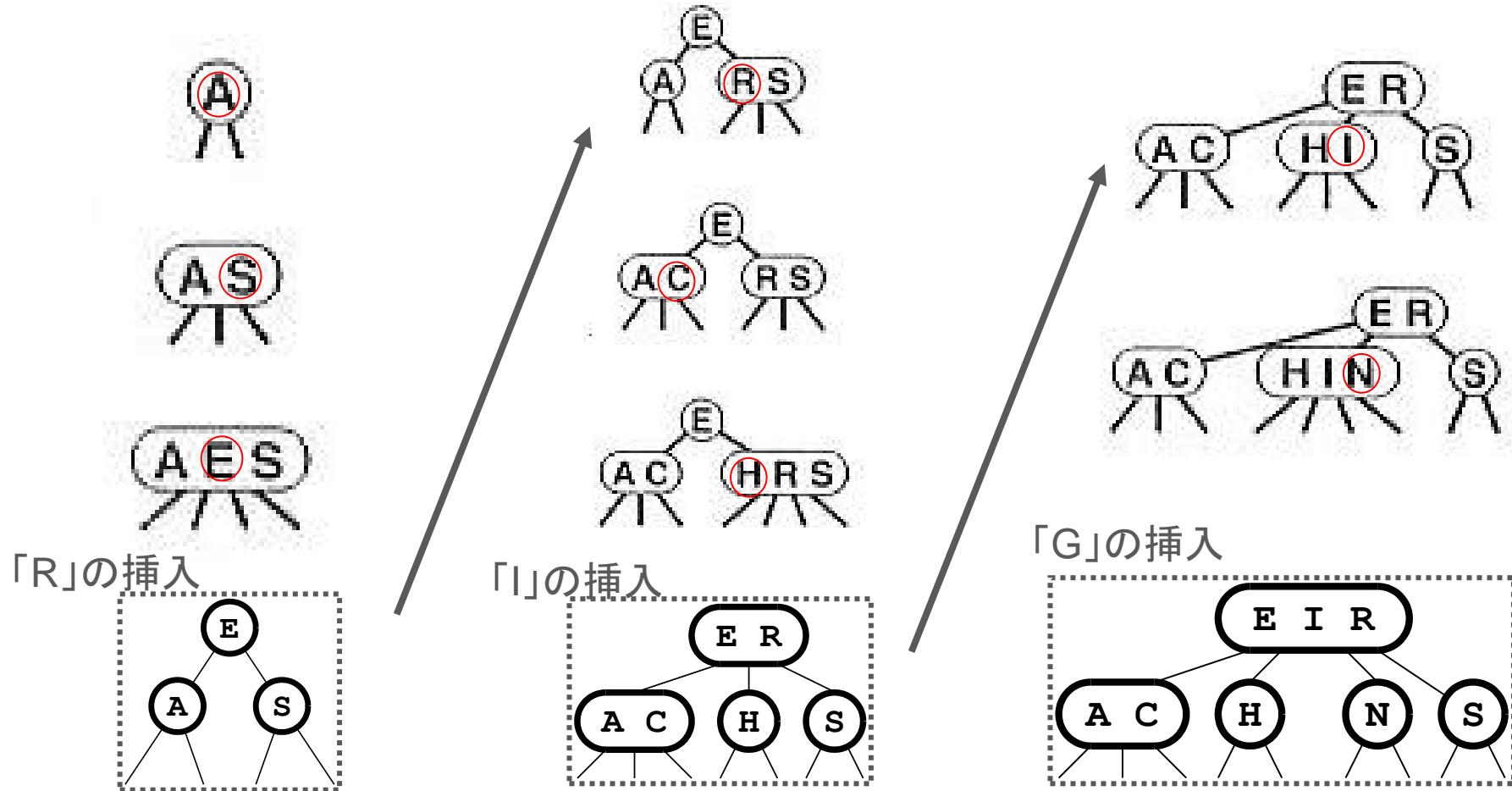


4-節点に出合つた→ 節点の分割



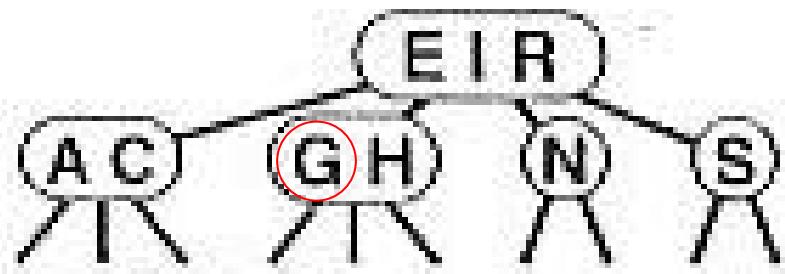
葉に到達、「E」を挿入する

トップダウン2-3-4木への挿入

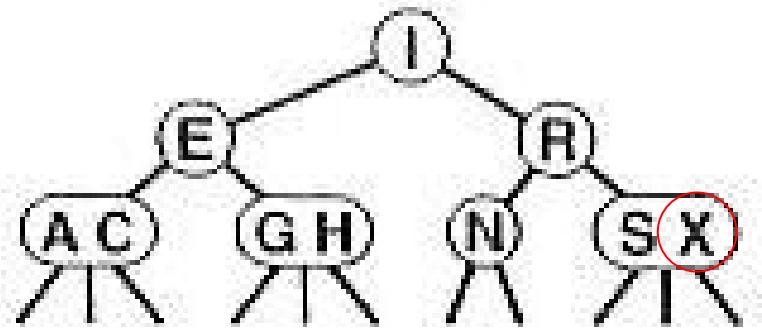
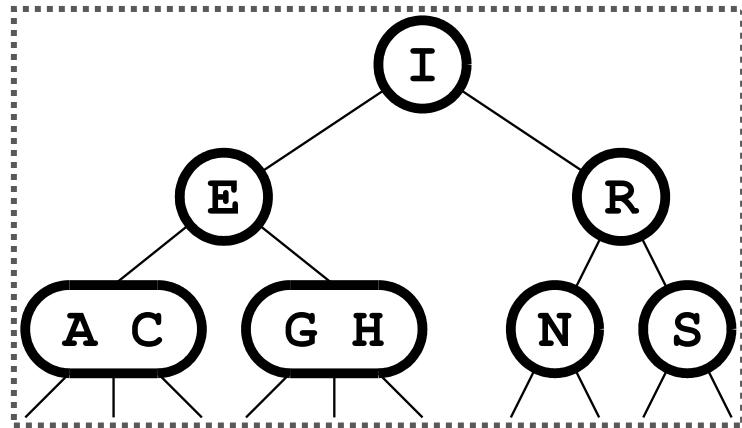


ASERCHINGXを挿入

トップダウン2-3-4木への挿入



「X」の挿入



ASERCHINGXを挿入

トップダウン2-3-4木への挿入

- 4-節点を、 2つの2-節点に分割するとき
 - 2つの2-節点のリンクの数は4つなので、 分割される節点より下に変化はない
 - もちろん、 中央のキーが1つ上に上げられる以外に、 分割される節点よりも変化がない
 - 分割は、「**局所的**」であると言える

2-3-4木

● 性質13.6

- N 個の節点を持つ2-3-4木における探索は、 $\lg N + 1$ 個以下の節点を訪問する

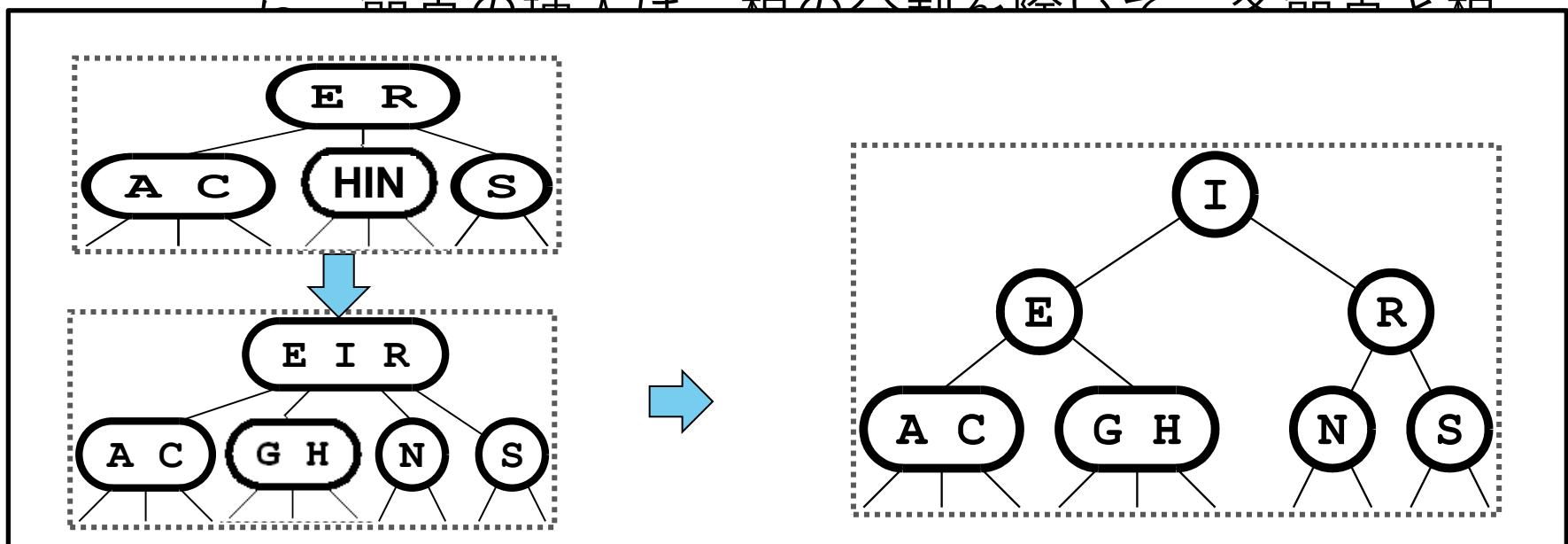
- 根から外部節点までの距離はすべて等しい。なぜなら、節点の挿入は、根の分割を除いて、各節点と根への距離を変化させないから。根の分割は、すべての節点への距離を1だけ増加させる
- 全ての節点が2-節点のときは、木はバランスしている2分木と同じで、この性質が成り立つ
- もし、3-節点や4-節点があれば、木の高さは、変化するとしても、 $\lg N + 1$ より小さくなるだけである

2-3-4木

● 性質13.6

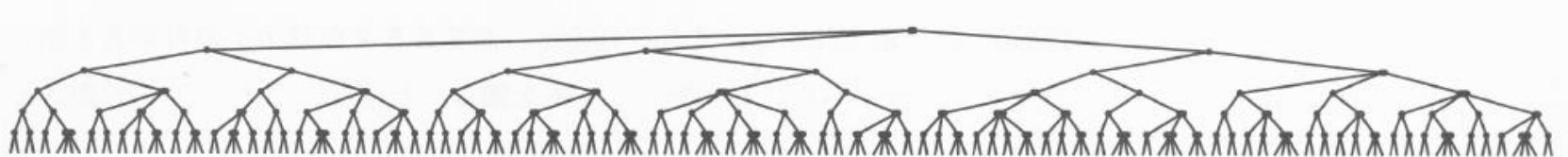
- N 個の節点を持つ2-3-4木における探索は、 $\lg N + 1$ 個以下の節点を訪問する

- 根から外部節点までの距離はすべて等しい。なぜなら
外部節点の挿入は 相の八割を除いて 外部節点と相



2-3-4木

- 性質13.7
 - N 個の節点を持つ2-3-4木への挿入において、節点の分割は、最悪の場合でも $\lg N + 1$ よりも少ない
 - 探索経路上のすべての節点が、4-節点のとき最悪で、このとき $\lg N$ 回の分割が必要である
- 実際には、あまり4-節点はできない
 - 200回の挿入を行った2-3-4木の例

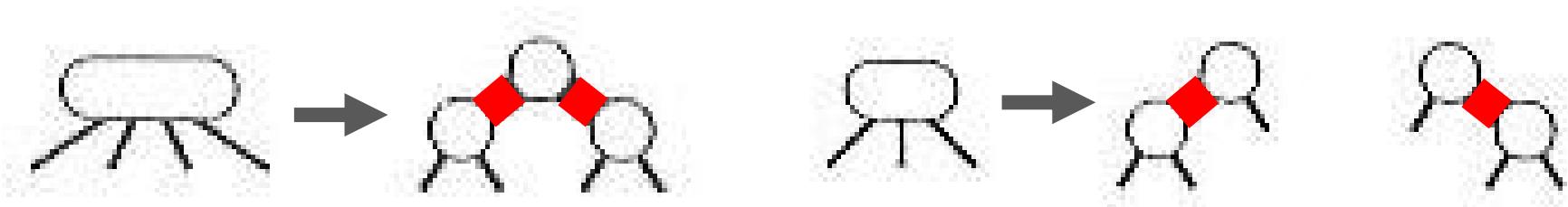


2-3-4木

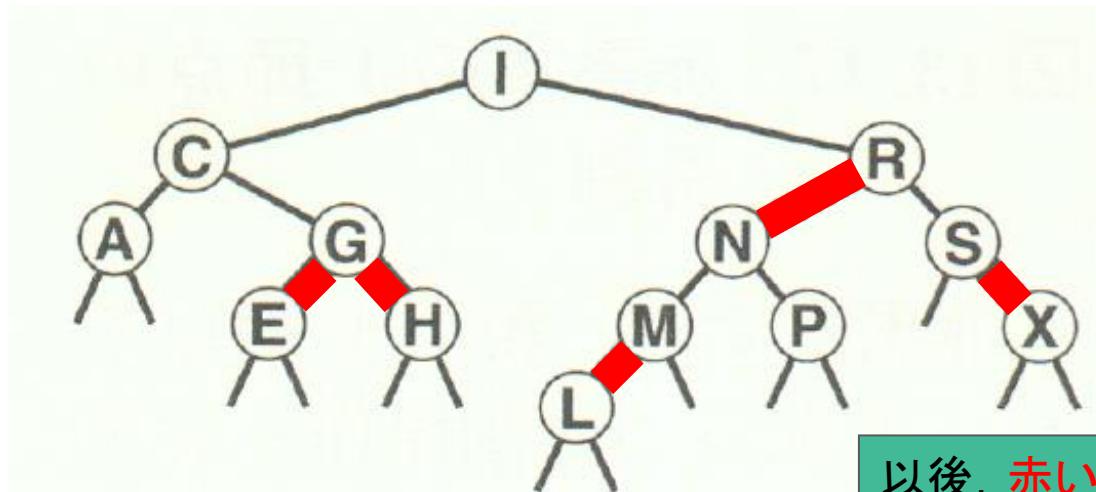
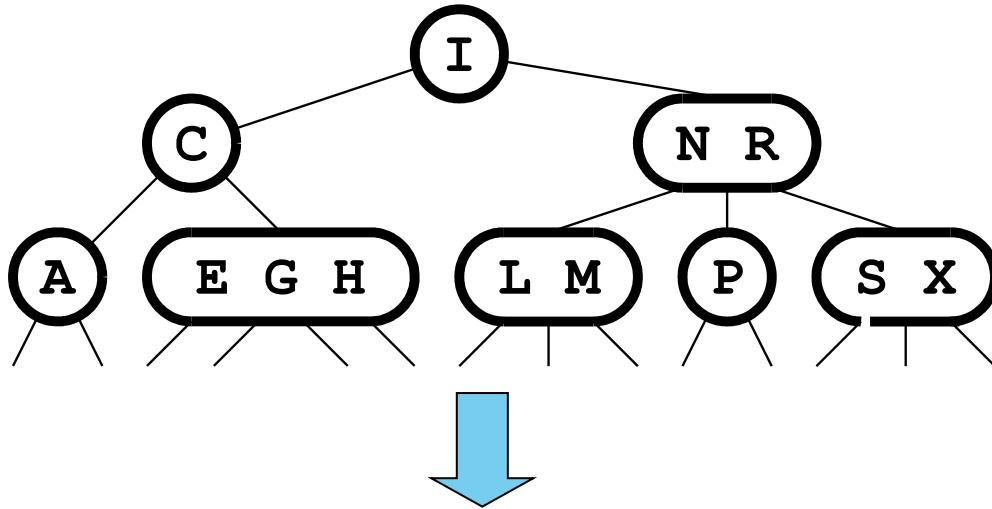
- 結論として、2-3-4木を用いると、最悪の場合でも、対数時間が保証されたアルゴリズムを作ることができる
- では、どうやって、2-3-4木をデータ構造として持つか？
 - 赤黒木

赤黒木

- 2-3-4木を実現するにはどうするか？
 - 通常の2分探索木で、2-3-4木を表現する
 - どうやって、3-節点や4-節点を表現するか？
 - 枝に「色」をつける
 - 「黒い」枝は、2-3-4木の枝である
 - 「赤い」枝は、小さな2分木を結合する。これは3-節点や4-節点を形成する節点を結ぶ枝である

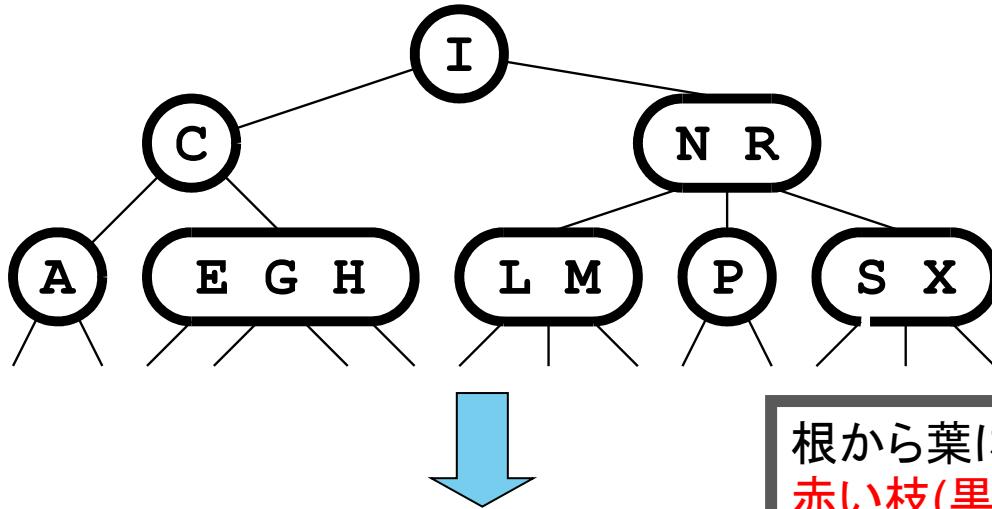


赤黒木

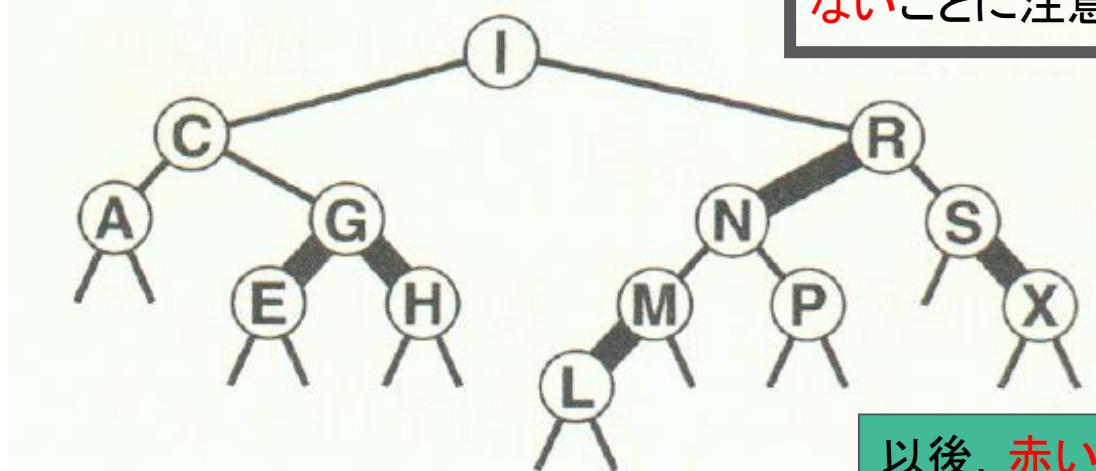


以後、赤い枝は、黒い太線
で記述

赤黒木



根から葉に至る道において、
赤い枝(黒い太枝)は、連続しないことに注意



以後、赤い枝は、黒い太線
で記述

赤黒木

節点を表すデータ型
にredという項目を追
加すれば赤黒木が
表現できる

```
struct STnode {  
    Item item;  
    link l,r;  
    int N;  
    int red;  
};
```

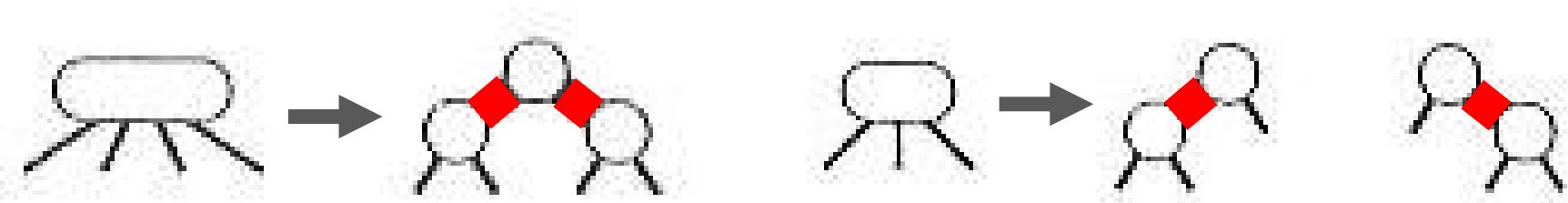
● 注意

○ 枝(リンク)に色をつけること = 各節点に色をつけること

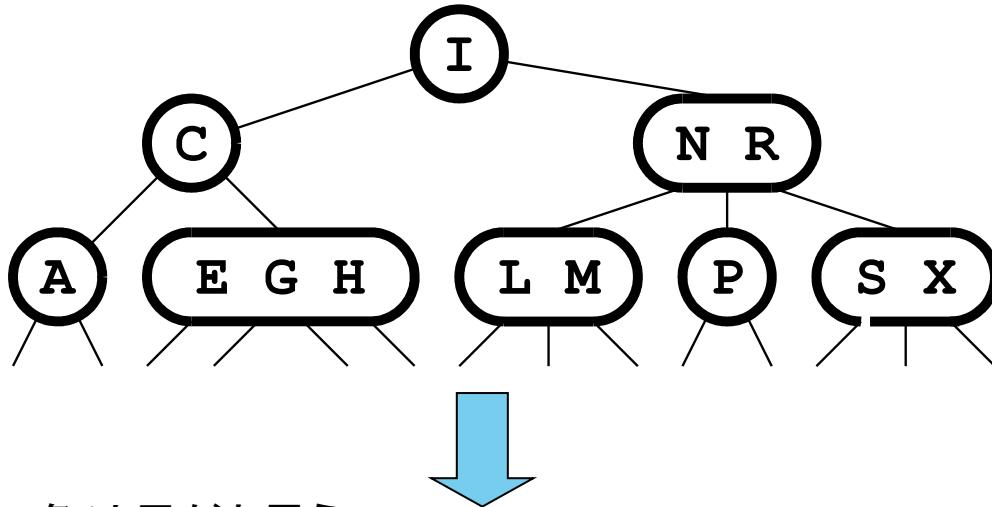
- 各節点は、ただ1つの親を持っているので、親とその節点とを結ぶ枝の色が、その節点の色だと思えばよい(次のページ)

○ 3-節点を表すのに2通りあるが、どうするか

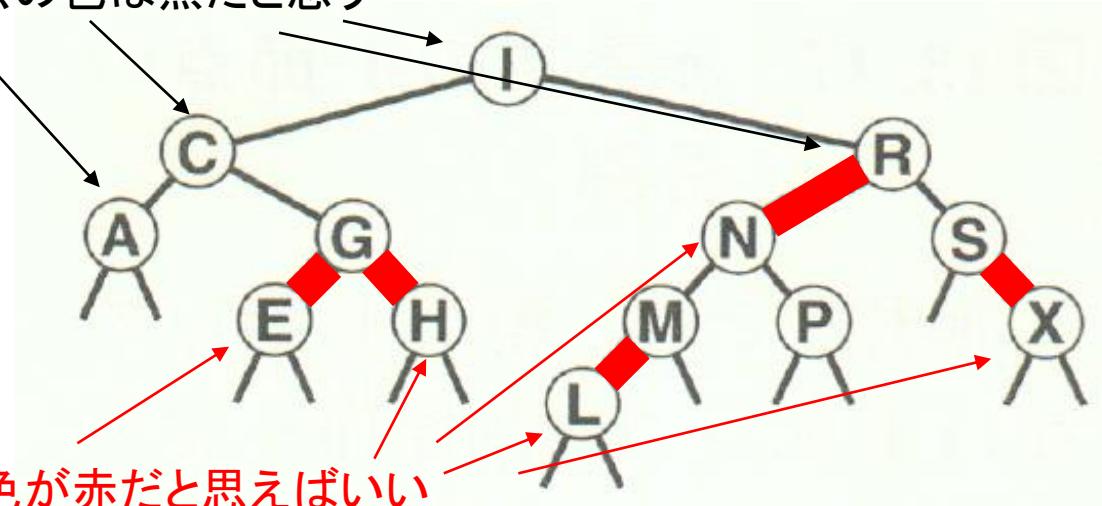
- アルゴリズムの中で、都合のいい方に自動的に決まる



赤黒木



これらの節点の色は黒だと思う



これらの節点の色が赤だと思えばいい

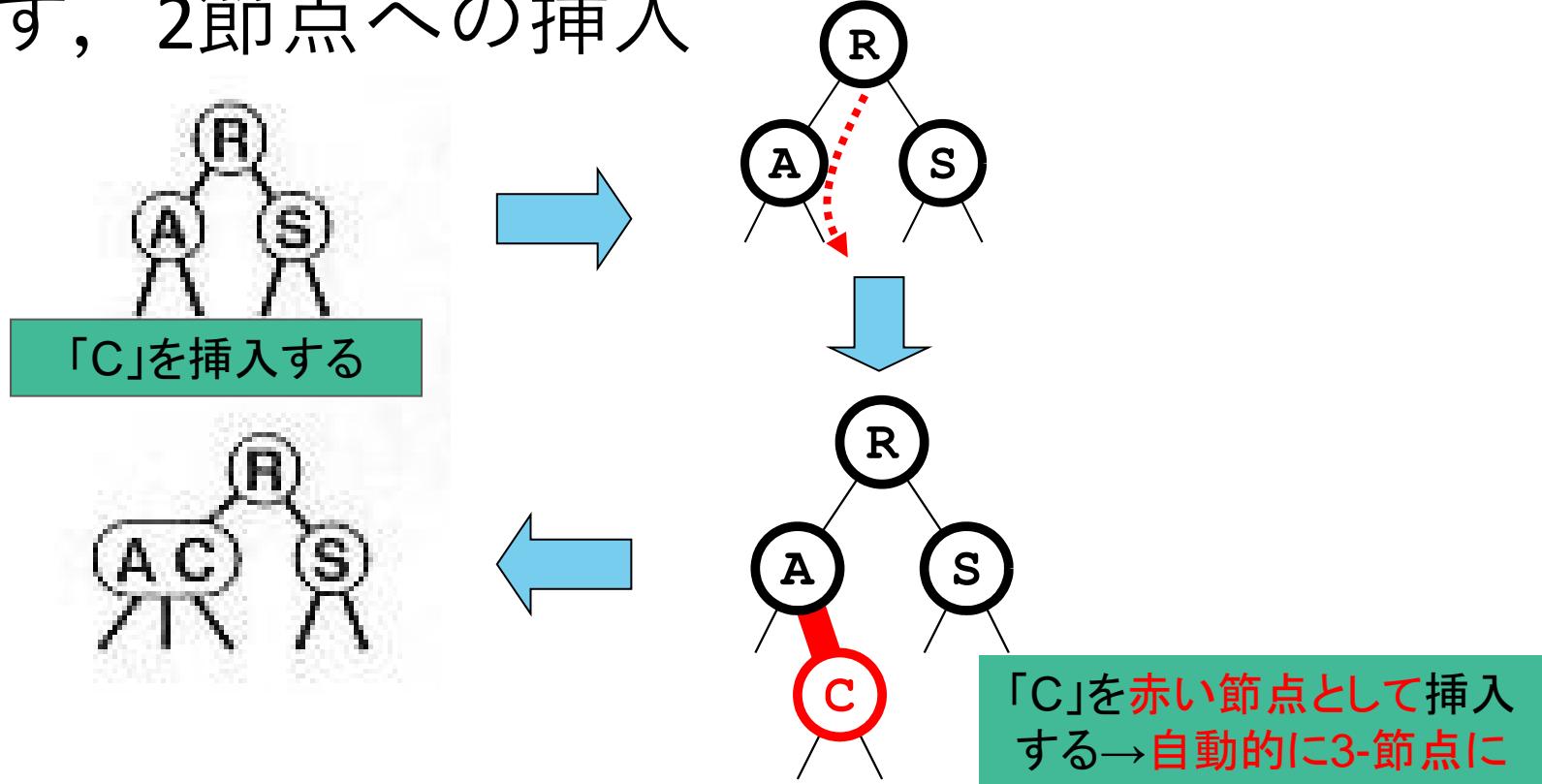
赤黒木

- 利点

- 赤黒木は、枝の色を考えないと、単に通常の2分探索木である。そのため、2分探索木における手続きsearchがそのまま使える
- 赤黒木は、直接2-3-4木と対応しているので、平衡2-3-4木のアルゴリズムを実現できる

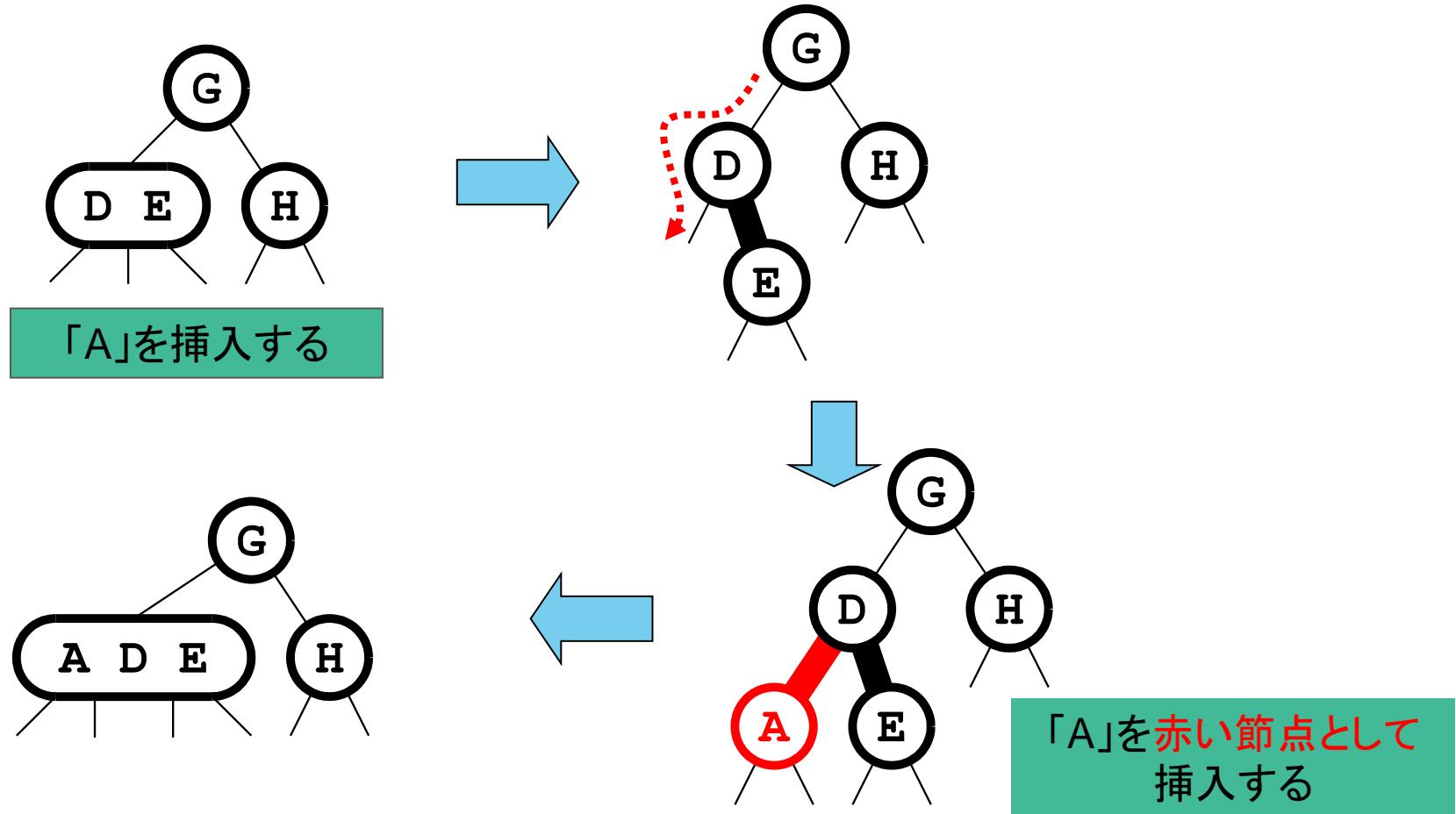
挿入

- では、赤黒木に対する挿入をどのように実現するか
- まず、2節点への挿入



挿入

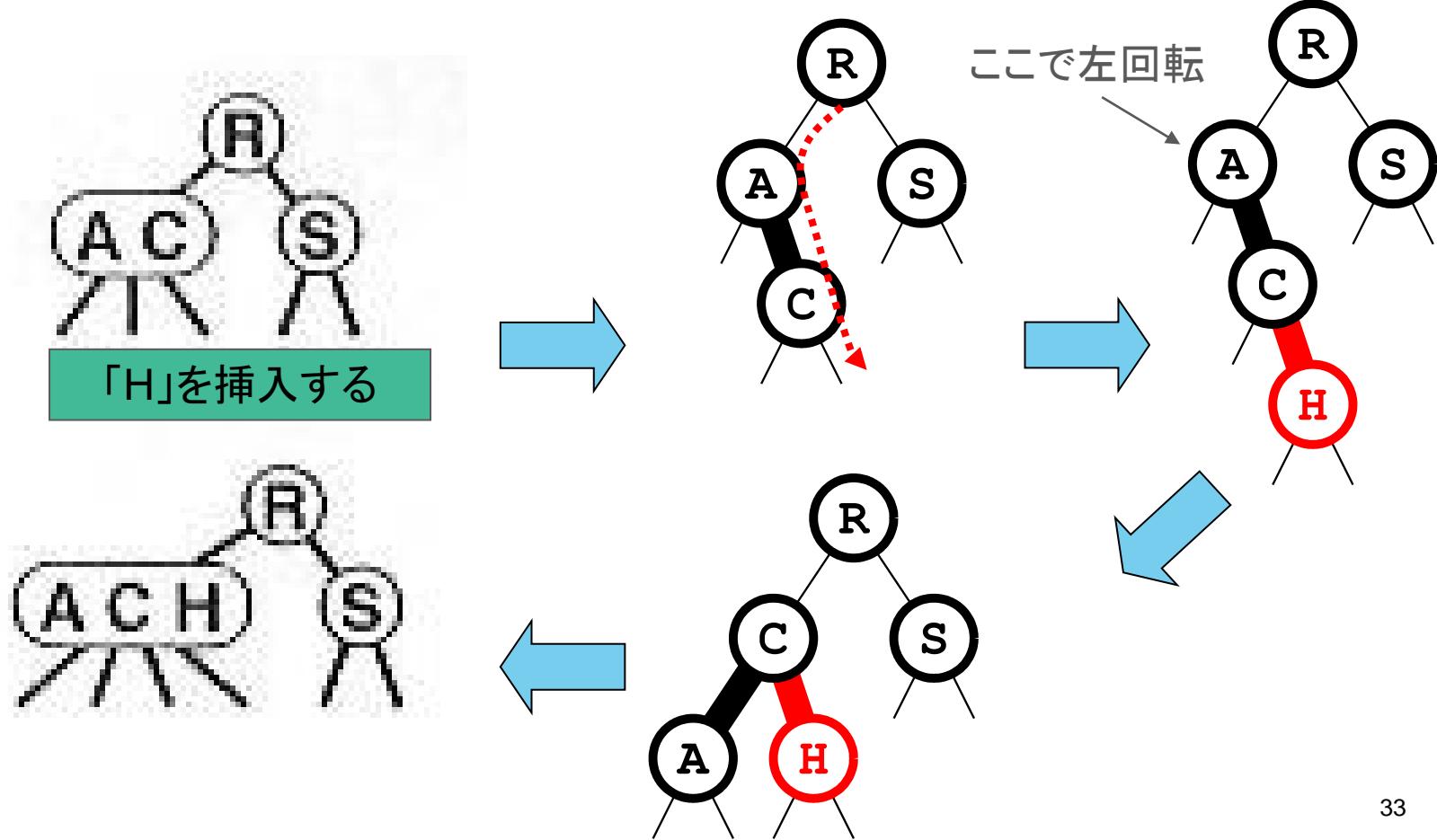
- 3節点への挿入(その1)



挿入

● 3節点への挿入(その2)

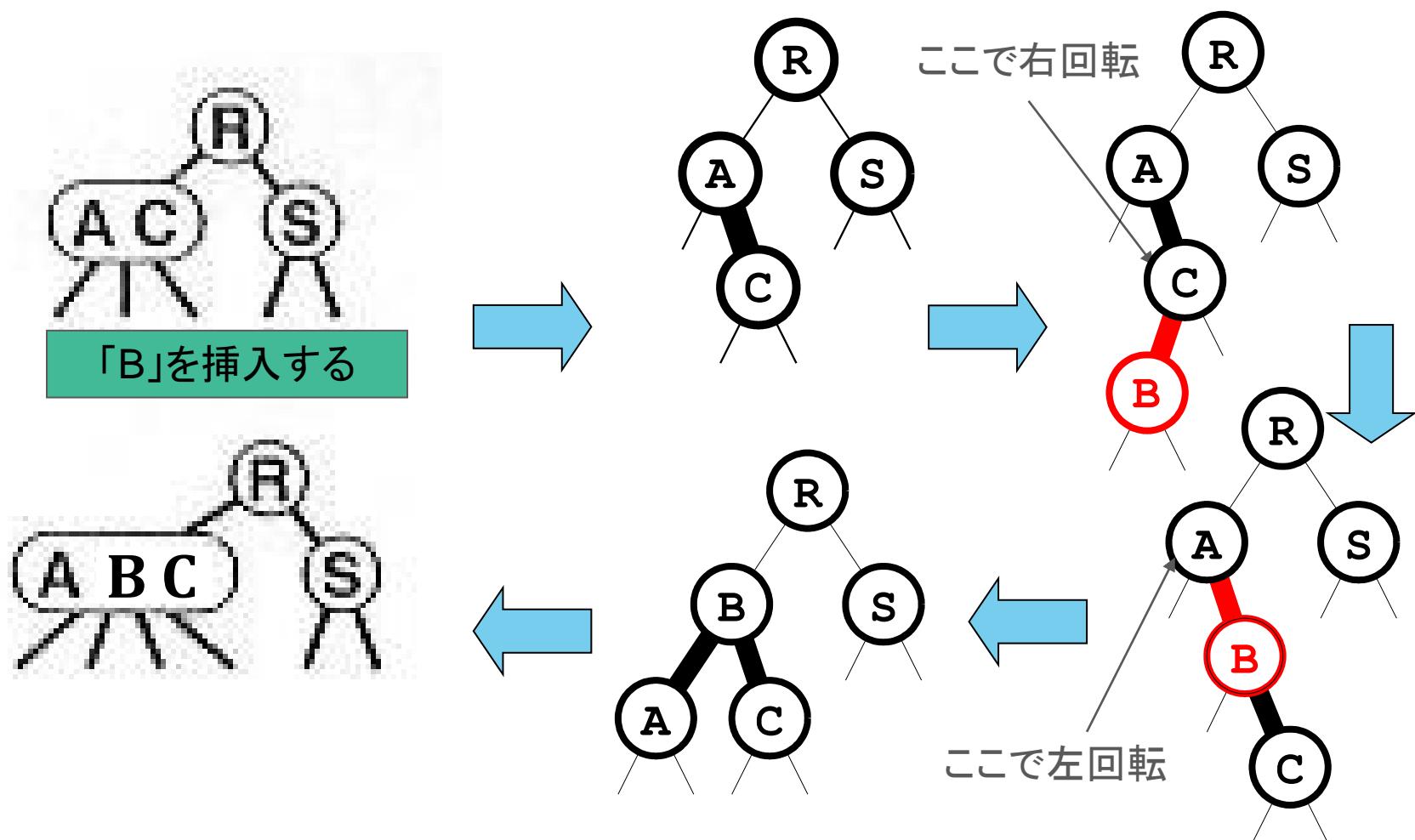
「H」を赤い節点として
挿入する



挿入

● 3節点への挿入(その3)

「B」を赤い節点として
挿入する

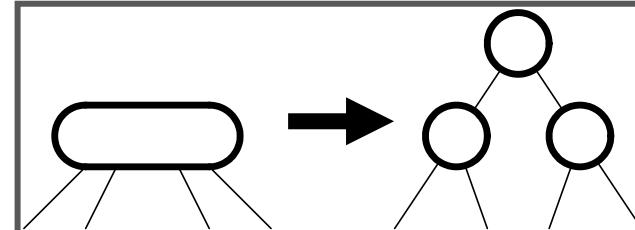
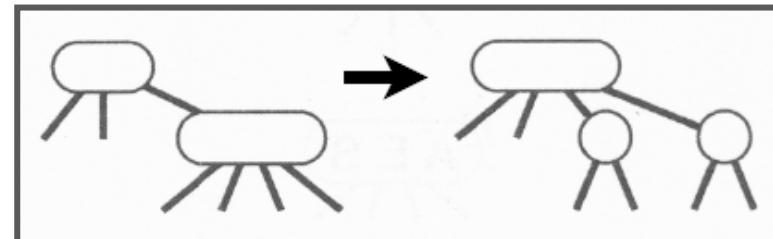
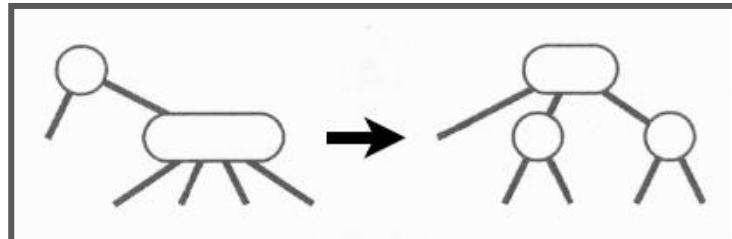


挿入

- 問題は4-節点に出合ったときに分割するときである
 - 4-節点の分割とは・・・(次頁のスライド)

挿入（復習，2-3-4木のとき）

- でも、もし4-節点の上が、さらに4-節点だったら？
 - 上にある4-節点から、さらにその上に、予め中央のキーを上げてあればいい
 - すなわち、ある項目を挿入するとき、探索経路中で、4-節点に出会ったら、予め中央のキーを上に上げ、2つの2-節点に分割しておけばいい
 - 根が4-節点のときは、3個の2-節点とする
 - そうすれば、必ず、キーを挿入すべき節点は、2-節点か3-節点になっている！
 - **トップダウン2-3-4木**

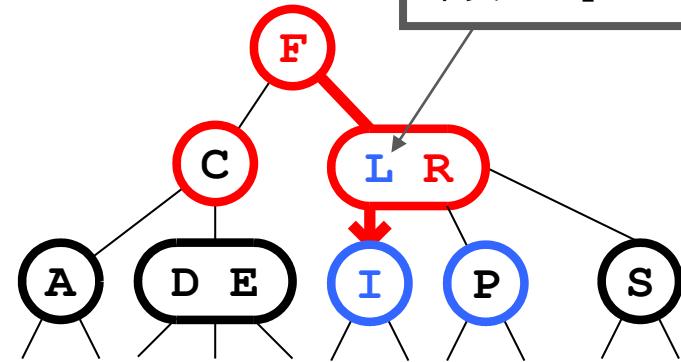
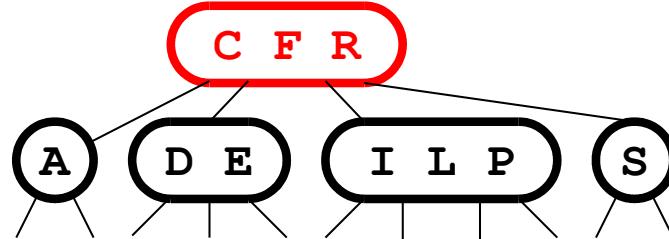


トップダウン2-3-4木への挿入（復習）

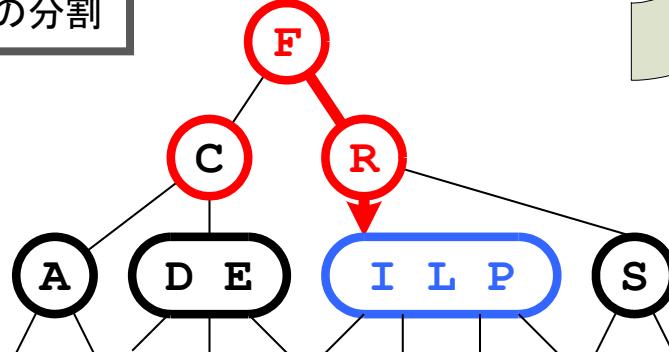
「K」の挿入

探索途中で、4-節点に出合ったら、その度に、
4-節点の分割を行う

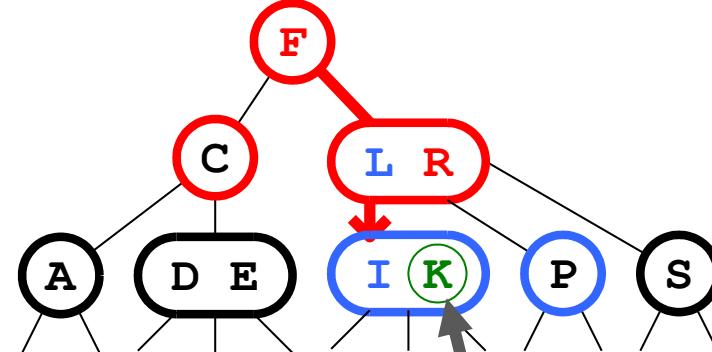
根から順に、「K」の挿入位置を探す



4-節点「CFR」に
出合った
→ 節点の分割



4-節点「ILP」に出合った → 節点の分割

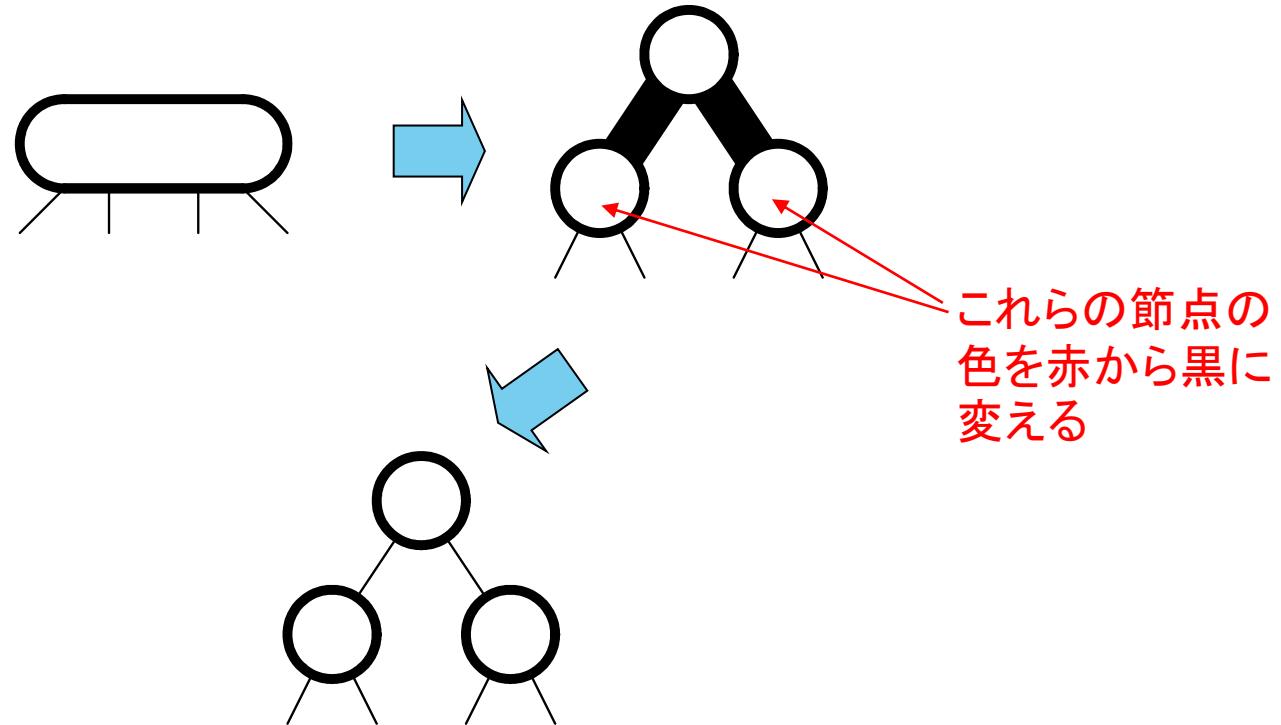


葉に到達、「K」を挿入する

中央の「L」が上に上がった

挿入

- 根が4節点のとき

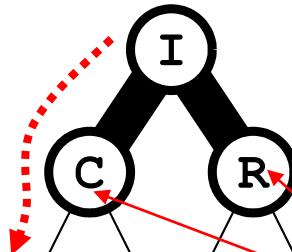


挿入

● 根が4節点のとき

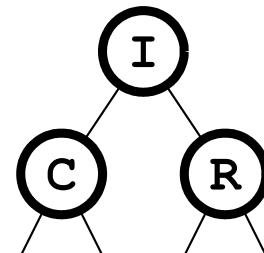


「A」を挿入する

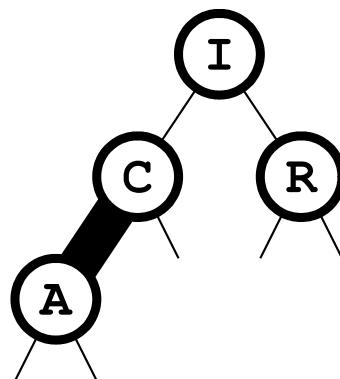


「A」を挿入する位置を探して行く

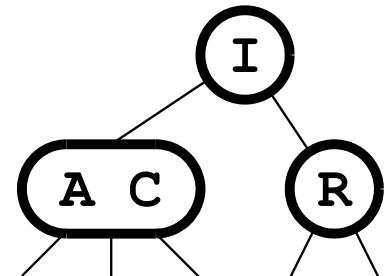
「I」の右も左が赤い枝なので、「色替え」を行う



これらの節点の色を赤から黒に変える

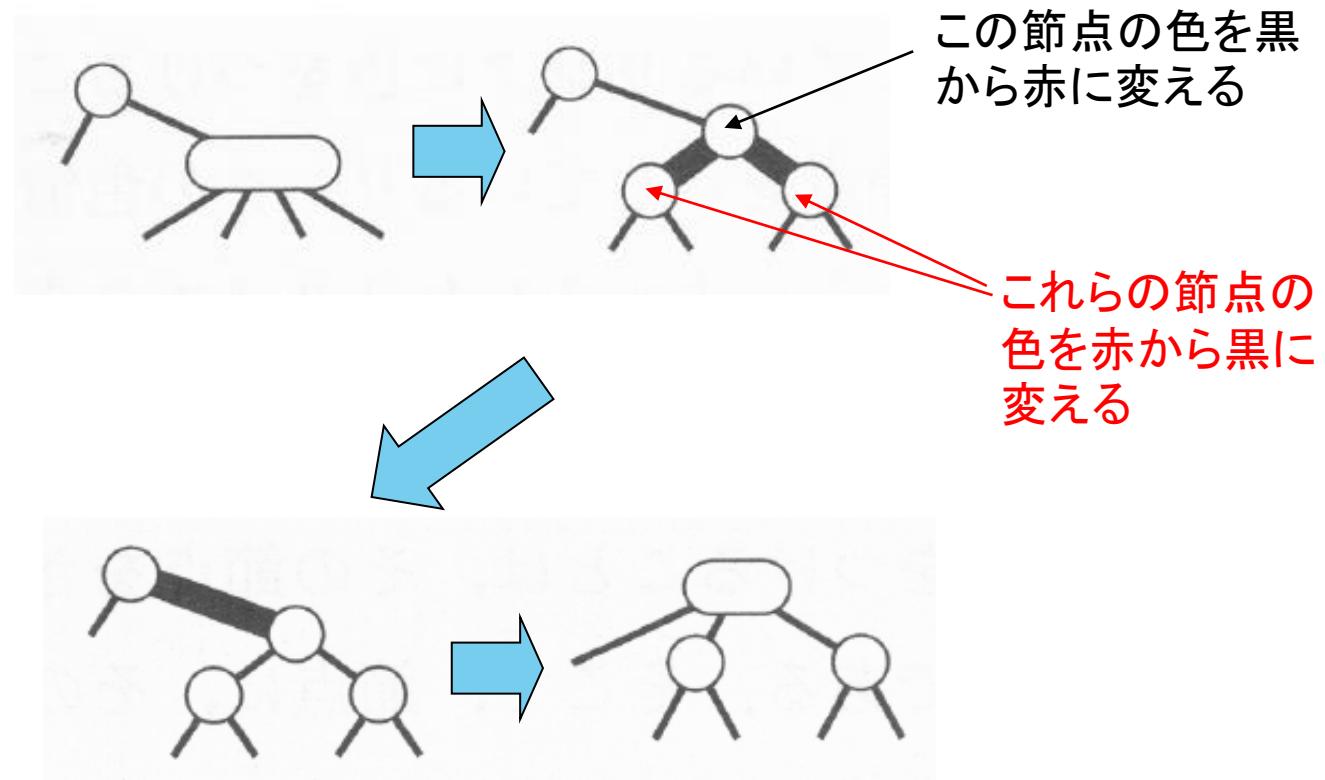


「A」を赤い節点として挿入する→自動的に3-節点に



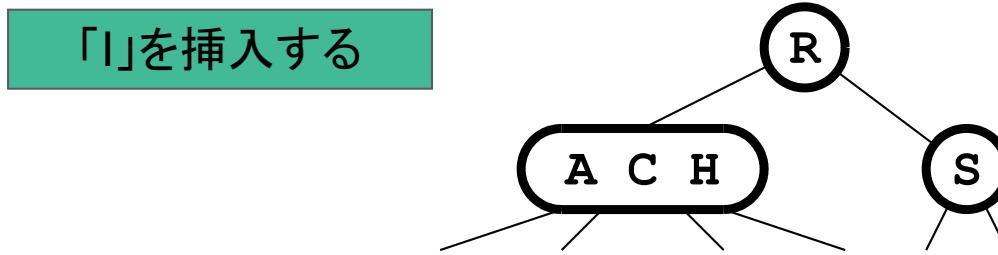
挿入

- 4-節点が2-節点の子節点である場合



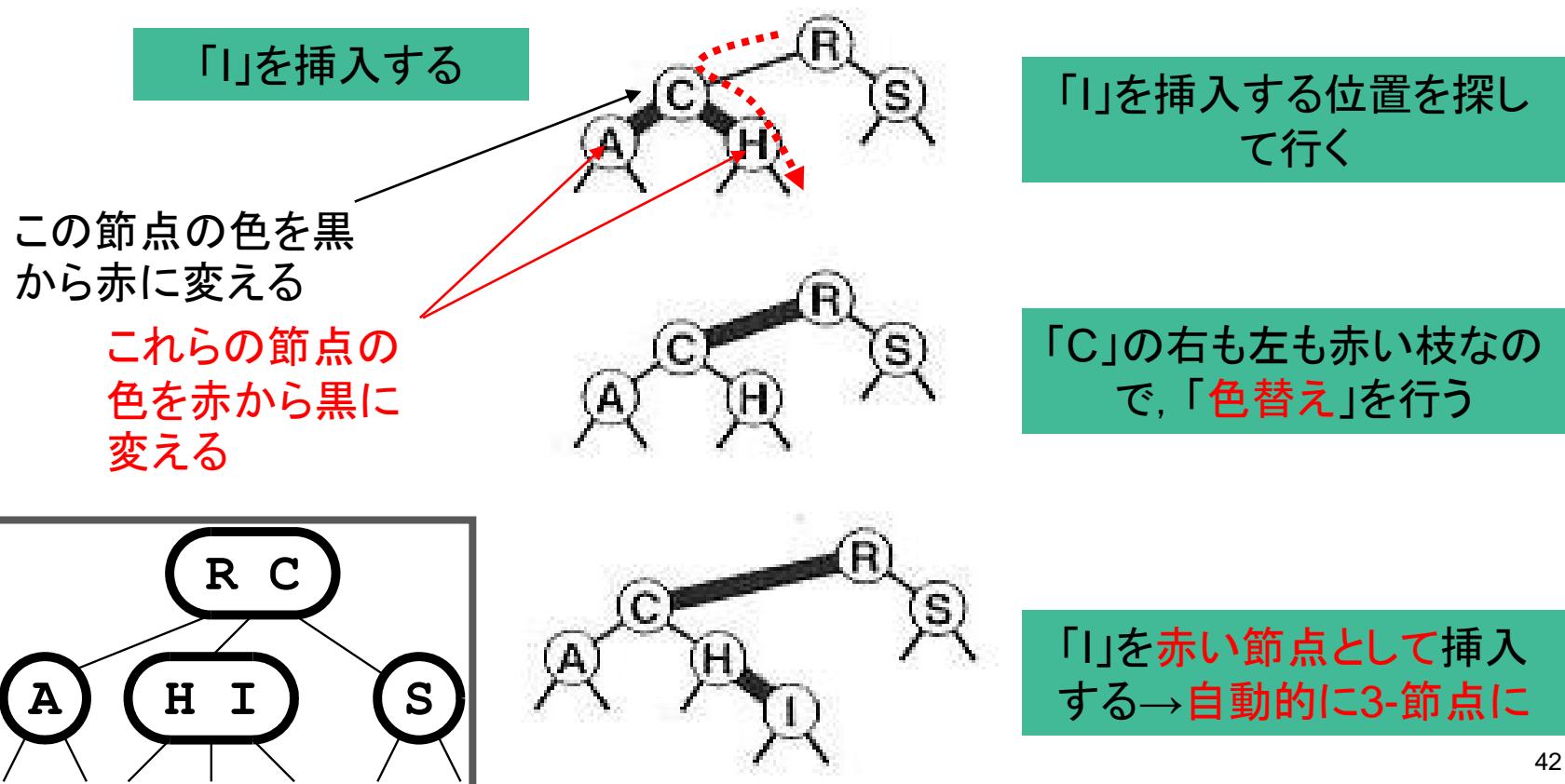
挿入

- 4-節点が2-節点の子節点である場合



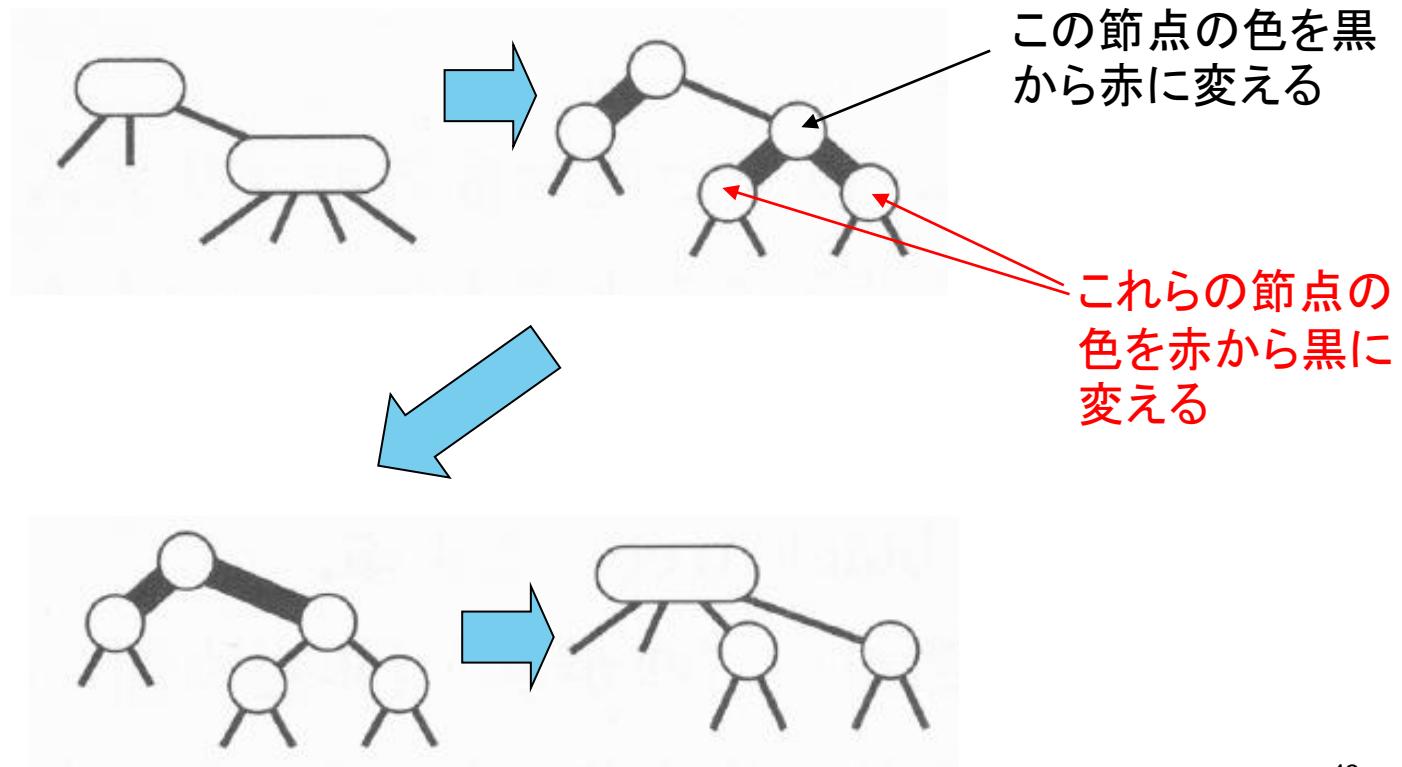
挿入

● 4-節点が2-節点の子節点である場合



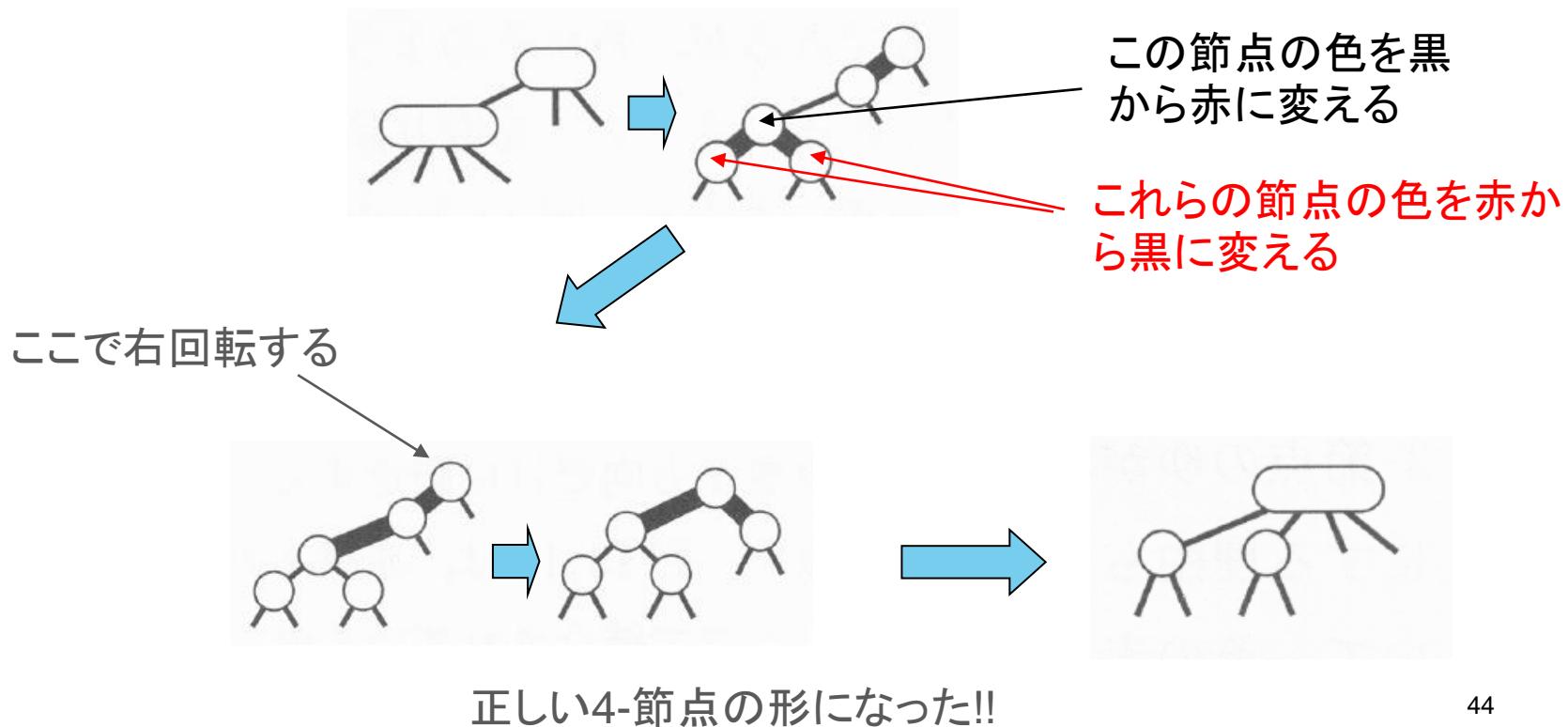
挿入

- 4-節点が3-節点の子節点である場合で、「正しい向き」にある場合



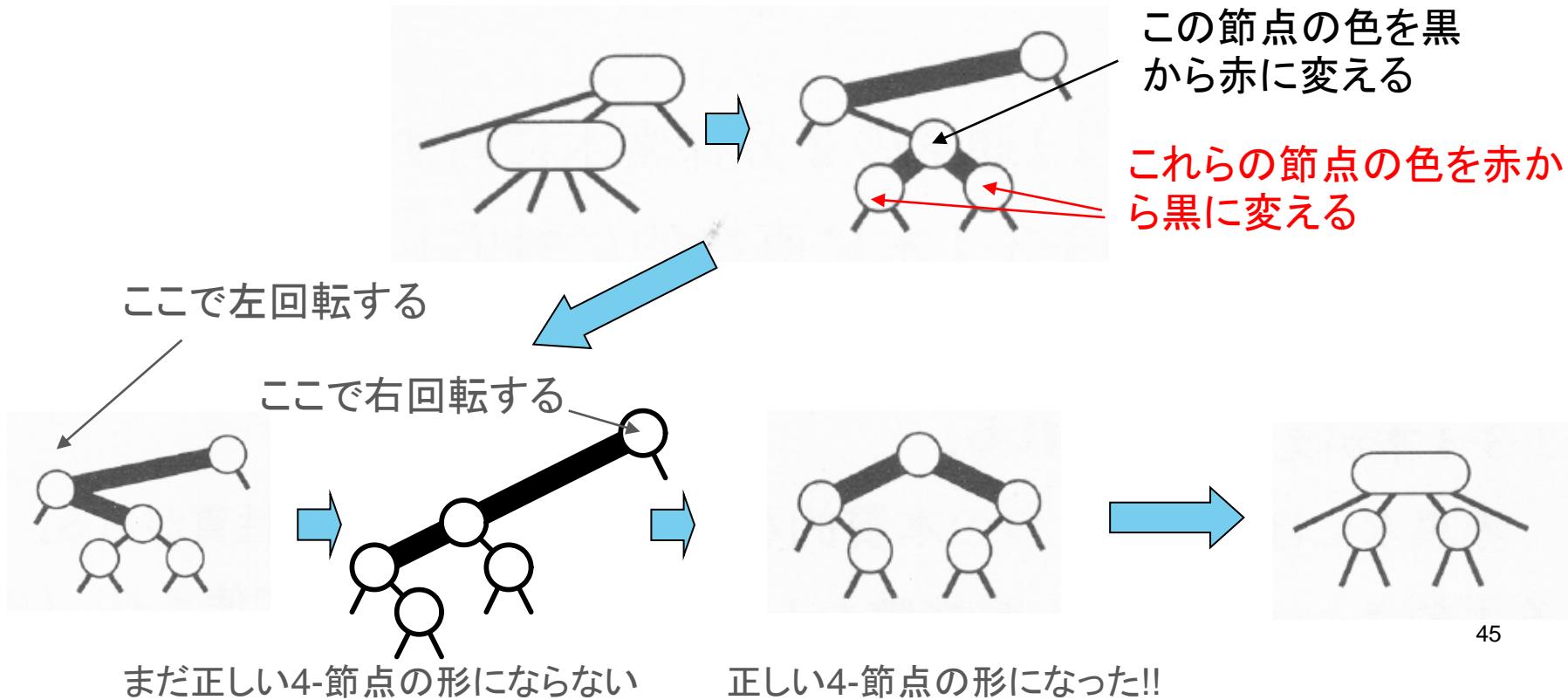
挿入

- 4-節点が3-節点の子節点である場合で、「正しい向き」にない場合(その1)



挿入

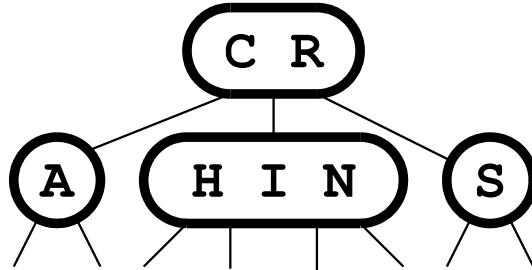
- 4-節点が3-節点の子節点である場合で、「正しい向き」にない場合(その2)



挿入

- 4-節点が3-節点の子節点である場合で、「正しい向き」にない場合(その2)

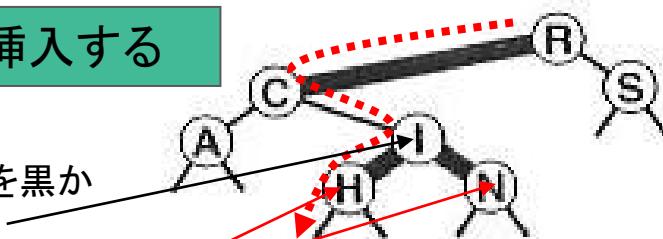
「G」を挿入する



挿入

- 4-節点が3-節点の子節点である場合で、「正しい向き」 にない場合(その2)

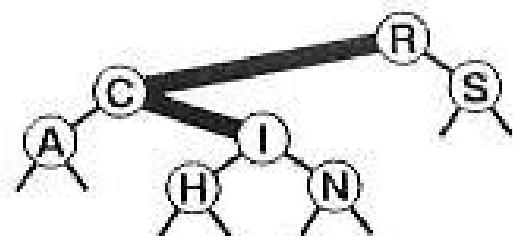
「G」を挿入する



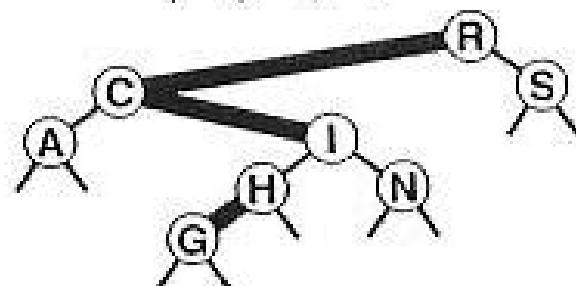
「G」を挿入する位置を探して行く

「I」の右も左も赤い枝なので、「色替え」を行う。向きが正しくないがとりあえずそのままにしておく

これらの節点の色を赤から黒に変える



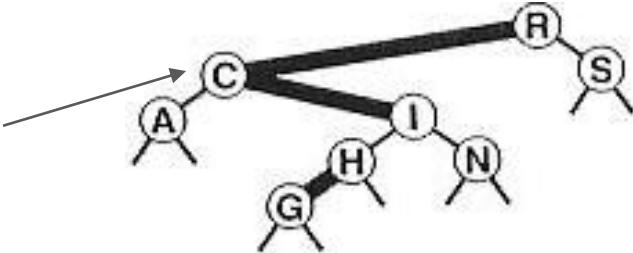
「G」を赤い節点として挿入



挿入

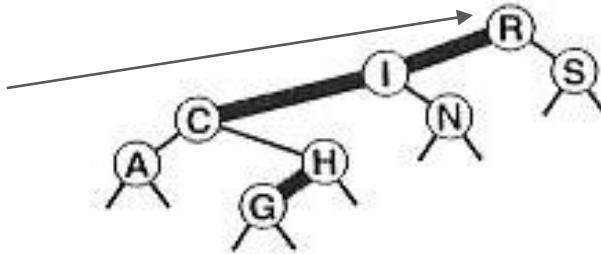
- 4-節点が3-節点の子節点である場合で、「正しい向き」にない場合(その2)

ここで左回転する

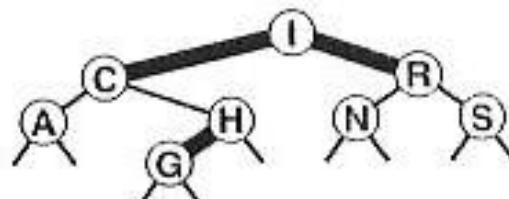
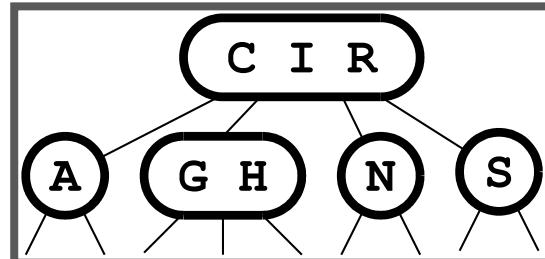


底から根にさかのぼりながら、「C」で左回転

ここで右回転する



さらにさかのぼりながら、「R」で右回転



正しく赤黒木ができた!!

插入

このプログラムは、エレガントに書かれているが、理解するには少し難しい。以下のブロックごとに分けて考えよう

```
link RBinsert(link h, Item item, int sw)
{ Key v = key(item);
  if (h == z) return NEW(item, z, z, 1, 1);
  if ((hl->red) && (hr->red))
    { h->red = 1; hl->red = 0; hr->red = 0; }
  if (less(v, key(h->item)))
  {
    hl = RBinsert(hl, item, 0);
    if (h->red && hl->red && sw) h = rotR(h);
    if (hl->red && hll->red)
      { h = rotR(h); h->red = 0; hr->red = 1; }
  }
else
{
  hr = RBinsert(hr, item, 1);
  if (h->red && hr->red && !sw) h = rotL(h);
  if (hr->red && hrr->red)
    { h = rotL(h); h->red = 0; hl->red = 1; }
}
fixN(h); return h;
}

void STinsert(Item item)
{ head = RBinsert(head, item, 0); head->red = 0; }
```

部分木に含まれる節点数の修正

節点のデータ構造

もし、空の木が与えられたら、赤い節点を新たに作成する

```
if (h == z) return NEW(item, z, z, 1, 1);
```

```
typedef struct STnode *link;  
struct STnode {  
    Item item;  
    link l, r;  
    int N;  
    int red;  
};
```

red = 1(赤)として、新たな節点を作成する

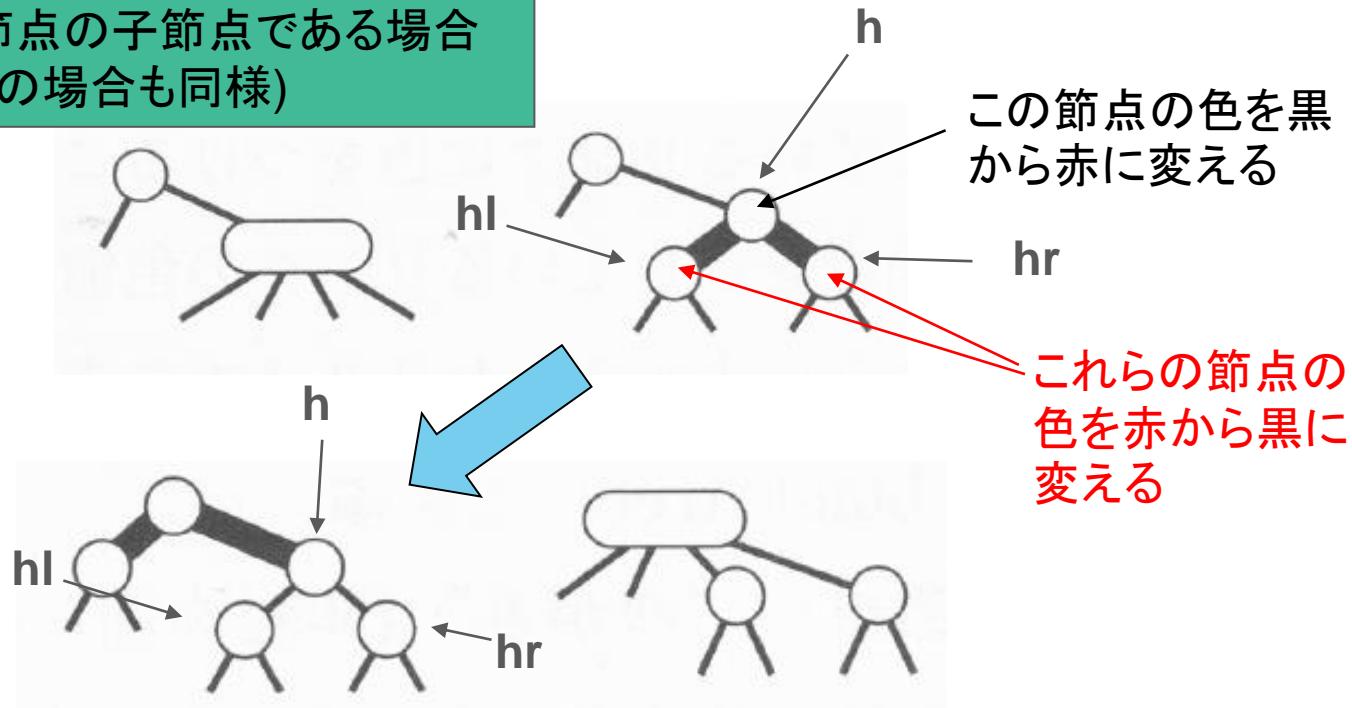
ST.cにおいて、STnodeをこのように変更する。
ここでredは、その節点(のすぐ上の枝)が赤か黒か
赤ならばred = 1. 黒ならばred = 0

節点の色替え

節点の挿入位置を探索している最中、再帰を深くしていく最中に、右の子も左の子も赤い節点であれば、色替えを行う

```
if ((hl->red) && (hr->red))  
{ h->red = 1; hl->red = 0; hr->red = 0; }
```

4-節点が2-節点の子節点である場合
(他の場合も同様)



回転

再帰から戻って来たら、回転を行う

```
if (less(v, key(h->item)) )
```

ある部分木の根hの左に、挿入されるとき

```

    {
        hl = RBinsert(hl, item, 0);
        if (h->red && hl->red && sw) h = rotR(h);
        if (hl->red && hll->red)
            { h = rotR(h); h->red = 0; hr->red = 1; }
    }

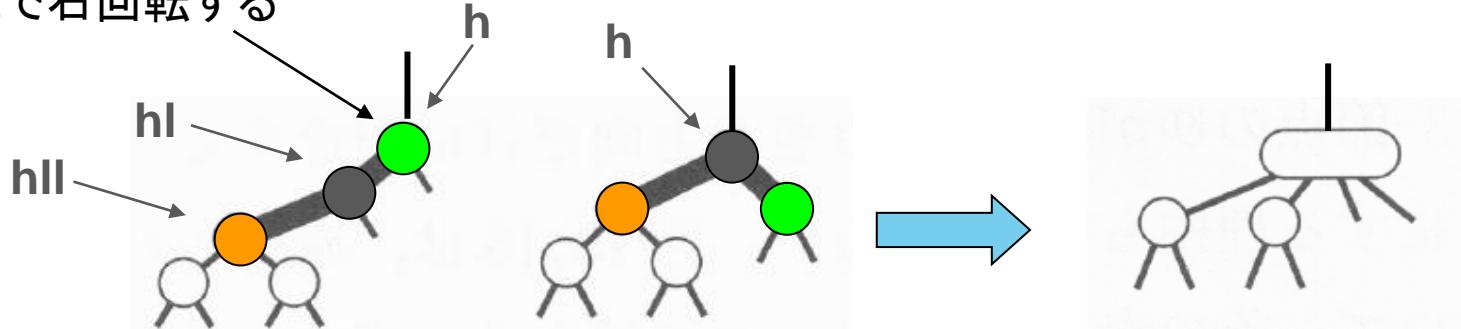
```

不成立

hの左の部分木まで、正しく挿入する
このとき、「左」を表す0を引数とする

4-節点が3-節点の子節点である場合で、
「正しい向き」にない場合(その1)

ここで右回転する



正しい4-節点の形になった!!

回転

再帰から戻って来たら、回転を行う

else

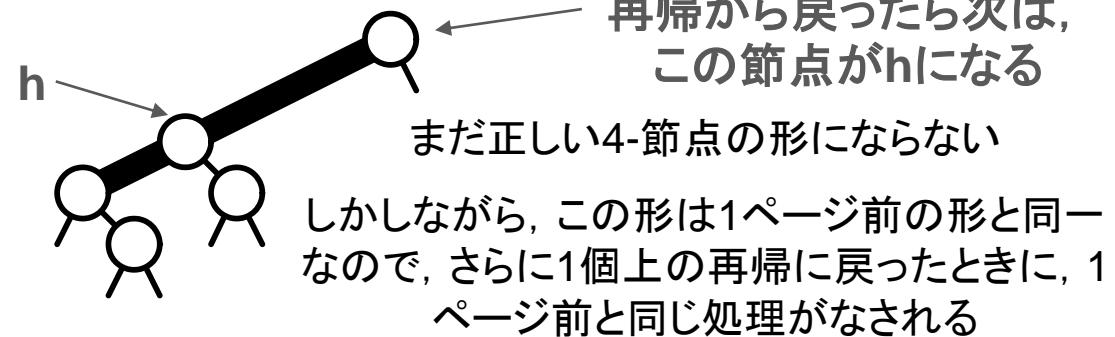
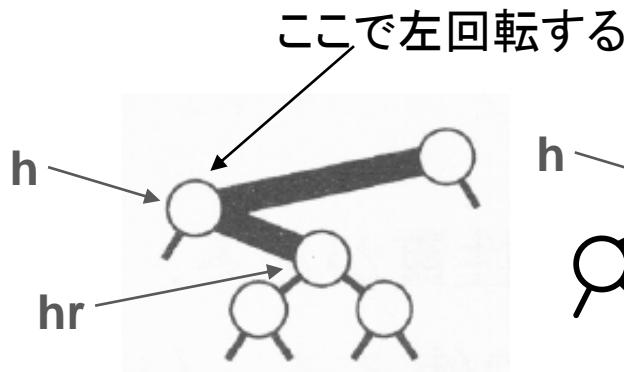
ある部分木の根hの右に、挿入されるとき

```
{  
    hr = RBinsert(hr, item, 1);  
    if (h->red && hr->red && !sw) h = rotL(h);  
    if (hr->red && hrr->red)  
    { h = rotL(h); h->red = 0; hl->red = 1; }  
}
```

hの右の部分木まで、正しく挿入する
このとき、「右」を表す1を引数とする

4-節点が3-節点の子節点である場合で、
「正しい向き」にない場合(その2)

挿入位置を探索する経路において、
hは、親に対して左の部分木になっ
ている。すなわちsw==0



赤黒木の性質

- 性質13.8
 - N 個のランダムなキーから作られた赤黒木における探索で実行される比較の回数は、 $2\lg N + 2$ より少ない
 - 性質13.7から、性質13.8を導くことができる