アルゴリズムとデータ構造

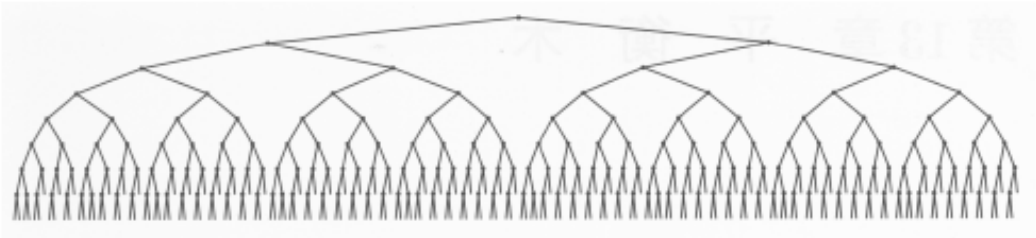
平衡木（その１）

目次

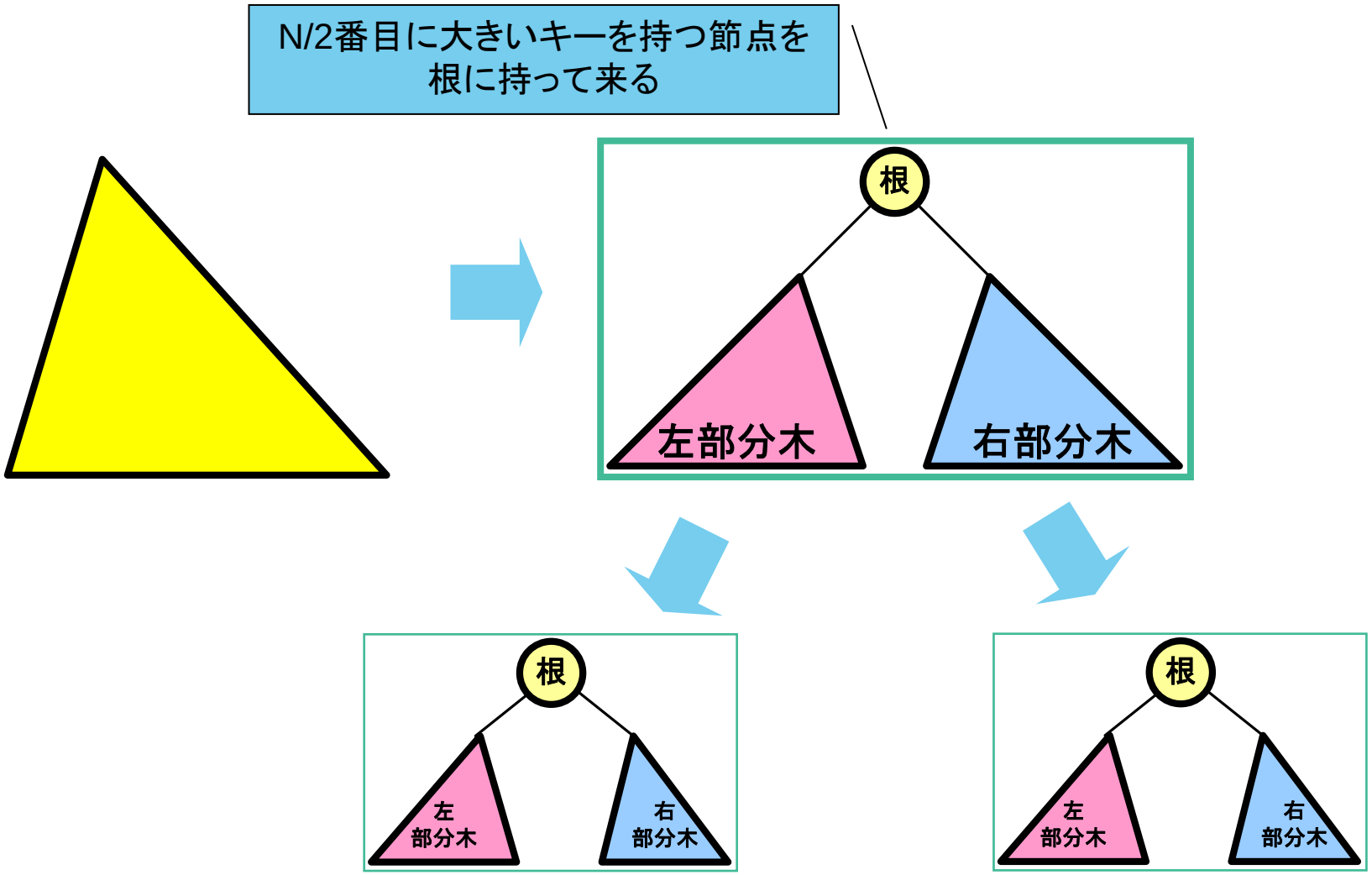
- 平衡木
- ランダム木
- スプレイ木

完全に平衡がとれた2分探索木

- 2分探索木
  - 最悪の場合，生成の手間が2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)



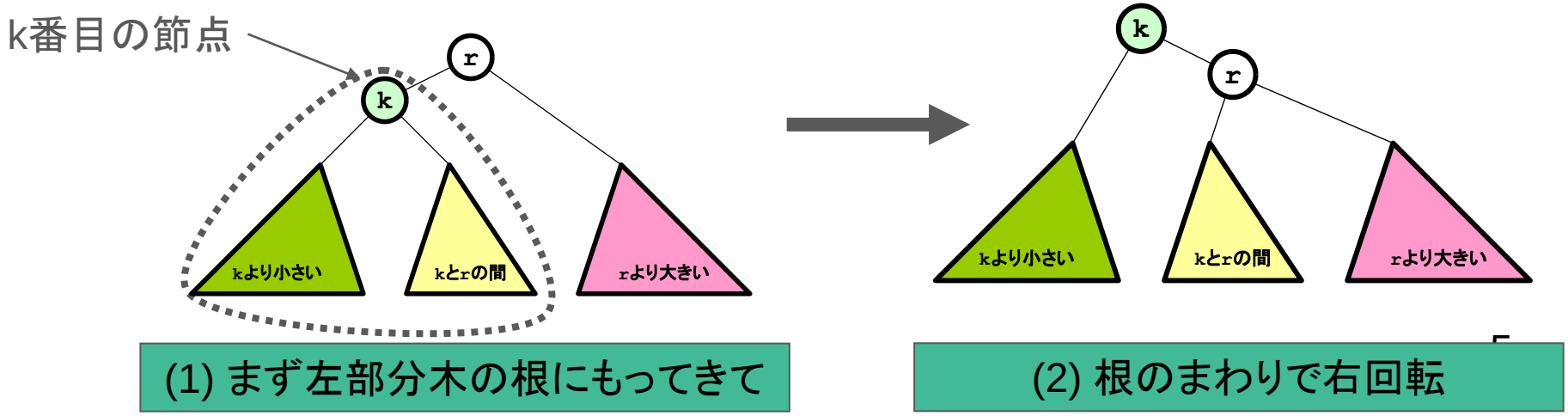
完全に平衡がとれた2分探索木



分割（復習）

- 選択操作を使って，「分割」操作が実現できる
  - 分割: **k番目の項目が根に来る**ように，2分探索木を変形すること．部分木の根に望む節点をおき，回転すれば全体の根とすることができる

k番目の節点が，左部分木にあるとき



分割（復習）

```
link partR(link h, int k)
  { int t = h->l->N;
    if (t > k)
      { h->l = partR(h->l, k); h = rotR(h); }
    if (t < k )
      { h->r = partR(h->r, k-t-1); h = rotL(h); }
    return h;
  }
```

左部分木の根に持って来て

右回転

ちょうど，k番目だったら，
自分自身を返す

完全に平衡がとれた2分探索木

- 完全に平衡がとれた2分探索木を作るには

```
link balanceR(link h)
   {
     if (h->N < 2) return h;
     h = partR(h, h->N/2);
     h->l = balanceR(h->l);
     h->r = balanceR(h->r);
     return h;
   }
```

部分木の中の節点数が1ならば，そのままでバランスしている

N/2番目の節点を根に持って来て

左右の部分木をバランスさせる

再平衡化

- 再平衡化
  - 完全に平衡がとれた2分探索木を作る(**再平衡化**)には，少なくとも節点数に比例した時間がかかる
  - 再平衡化を行うまでは，最悪，探索に線形時間かかる
  - 動的な2分探索木において，挿入/削除の都度，再平衡化を実行するのはコストが高くつく

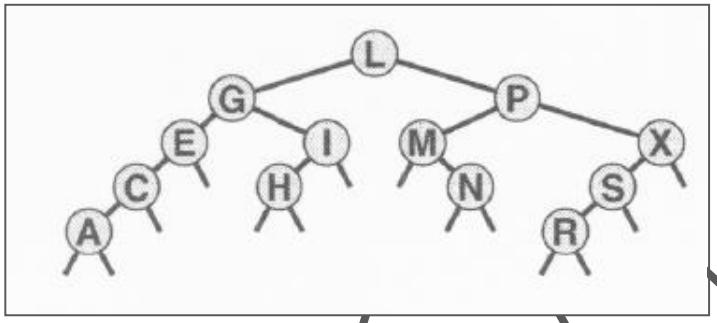
平衡木

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

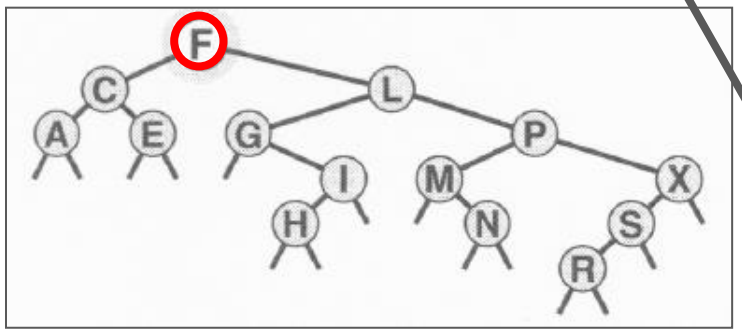
ランダム化BST

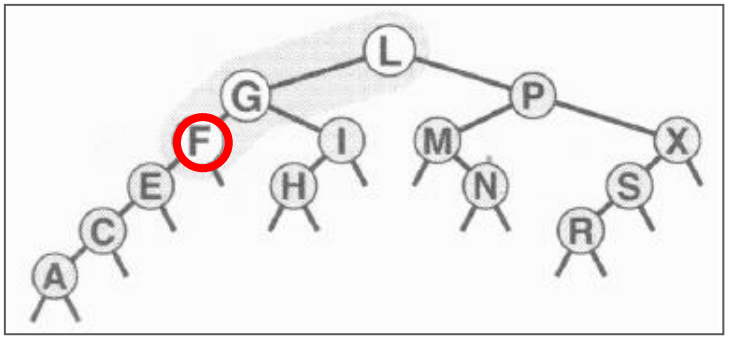
- 2分探索木の平均コスト解析の仮定
  - レコードはランダムに挿入される
    - 木のどの節点も根となる確率は等しい．部分木に関しても同様である
  - しかし，実際には，節点がある程度，整列された挿入されることが大いにあり得る
- 強いて，ランダム性を導入することができないか
  - **N**節点の木に新しい節点を挿入するときに，新しい節点が根となる確率を，**1/(N+1)**，となるようにする

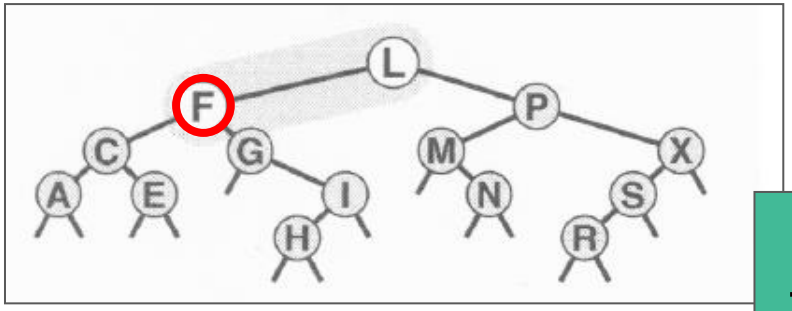
ランダム化BST



これにFを挿入する

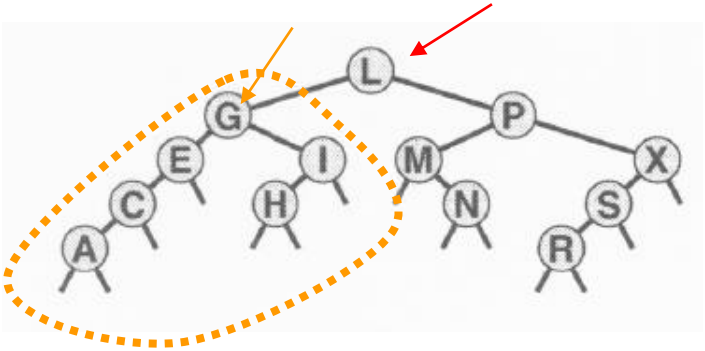






どれになるかは
ランダムに決定される

ランダム化BST

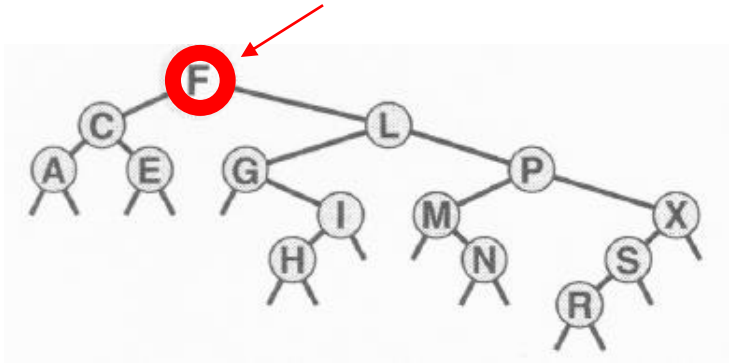


節点数: 6

これにFを挿入する

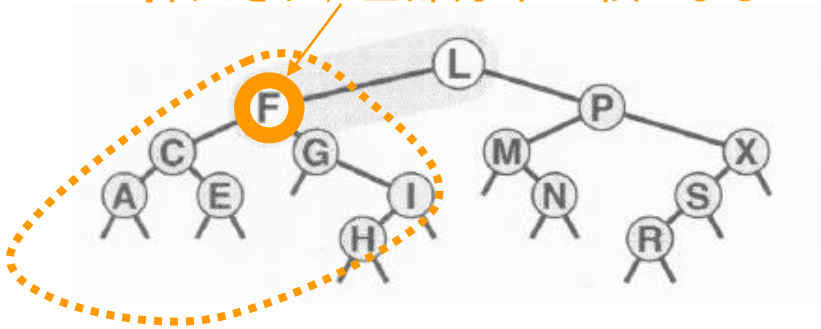
節点数: 13

P_1=1/(13+1)の確率で，Fは根に挿入される



(1-P_1)の確率で，Gを根とする部分木をチェックして，

その後，**P_2=1/(6+1)**の確率でFはここに挿入され，左部分木の根となる



ランダム化BST

itemをhの根に挿入する関数

```
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if (rand()< RAND_MAX/(h->N +1))
      return insertT(h, item);
    if less(v, t) h->l = insertR(h->l, item);
             else h->r = insertR(h->r, item);
    (h->N)++; return h;
  }
void STinsert(Item item)
  { head = insertR(head, item); }
```

ある確率で，部分木hの根にitemを挿入する

そうでなければ，通常通り，木の底の方へ，1レベル増やして，挿入を試みる

ランダム化BST

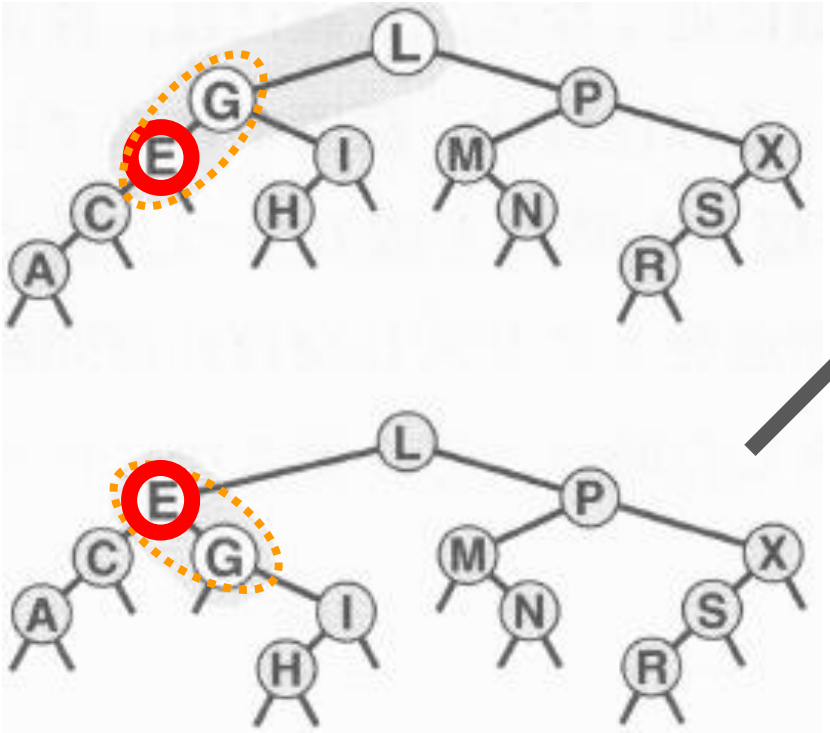
- 性質13.1
  - ランダム化BSTを生成することは，キーのランダム順列から標準のBSTを生成することと同値である
    - Nレコードのランダム化BSTを生成するには，約2NlnN回の比較を行う
    - また，探索には約2lnN回の比較を行う
      - どの要素も根になる確率は等しく，この性質はどの部分木にも成り立つから
- 性質13.2
  - ランダム化BSTの生成コストが平均のα倍より大きい確率は，$e^{-\alpha}$より小さい

ランダム化BST

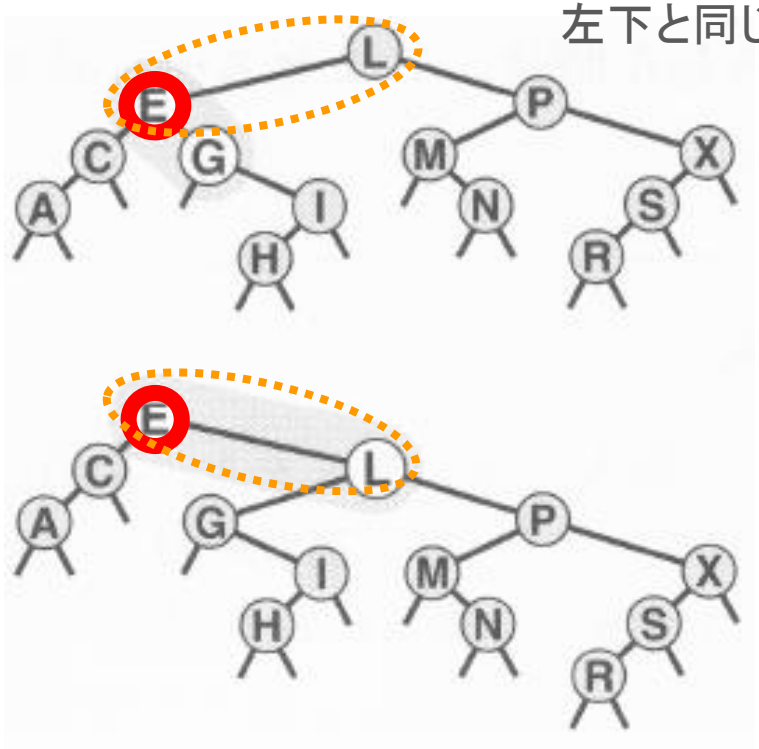
- 利点
  - 節点の挿入順序によらず，ランダム性を持ったBSTを生成することができる(性質13.1)
- 欠点
  - 乱数の発生
  - 各節点が，その節点を根とする部分木の節点数を保持する必要がある(**h->N**)
    - **h->N**の保持は，select操作などで必要になることもあるので，欠点にならない場合もある

スプレイBST

- 根への節点の挿入を思い出そう
  - 挿入しながら右回転あるいは左回転を繰り返す

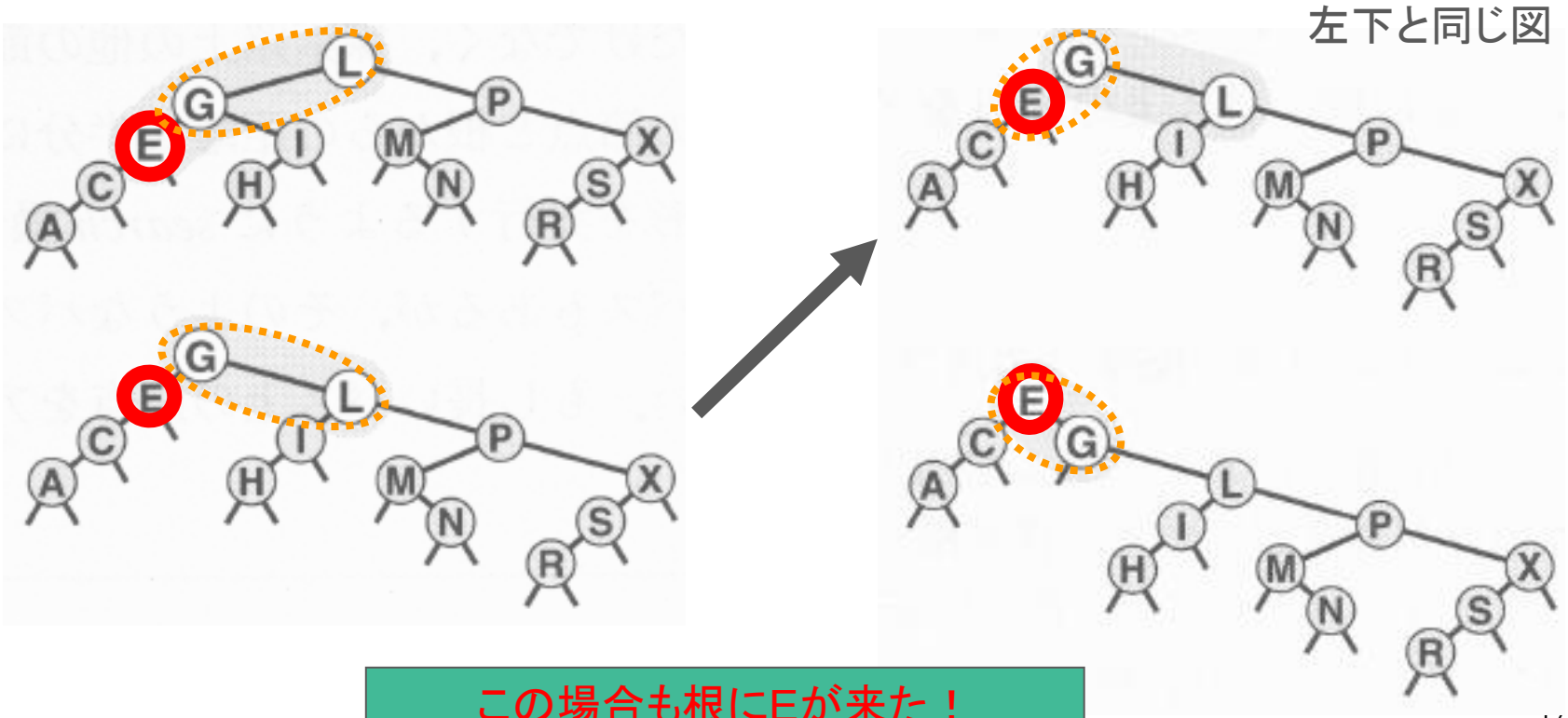


左下と同じ図

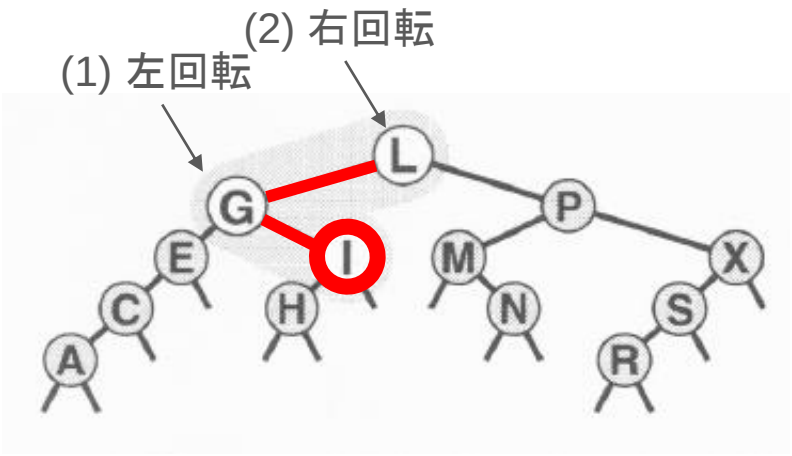


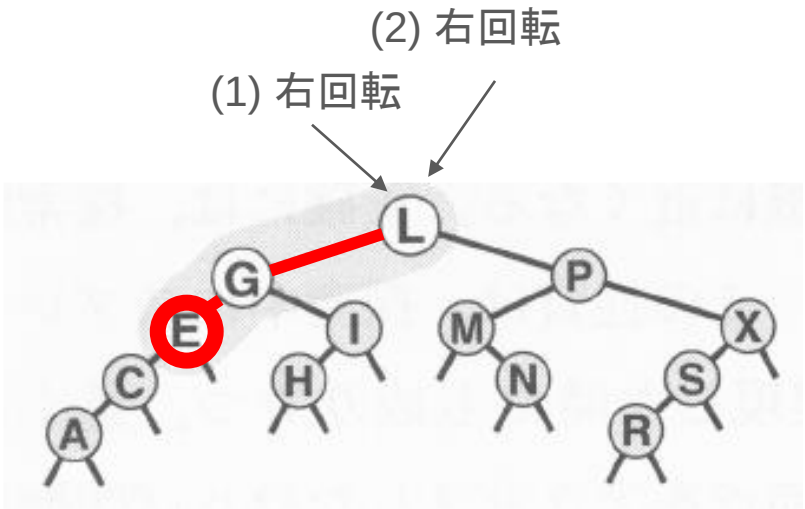
スプレイBST

- 根のまわりで2回右回転したらどうなるか



スプレイBST





部分木の根から見て，向きが異なるところに，挿入した節点があるとき

→ 根への挿入と同じで，下から上に左回転，右回転

部分木の根から見て，向きが同じところに，挿入した節点があるとき

→ 根の位置で右回転，右回転

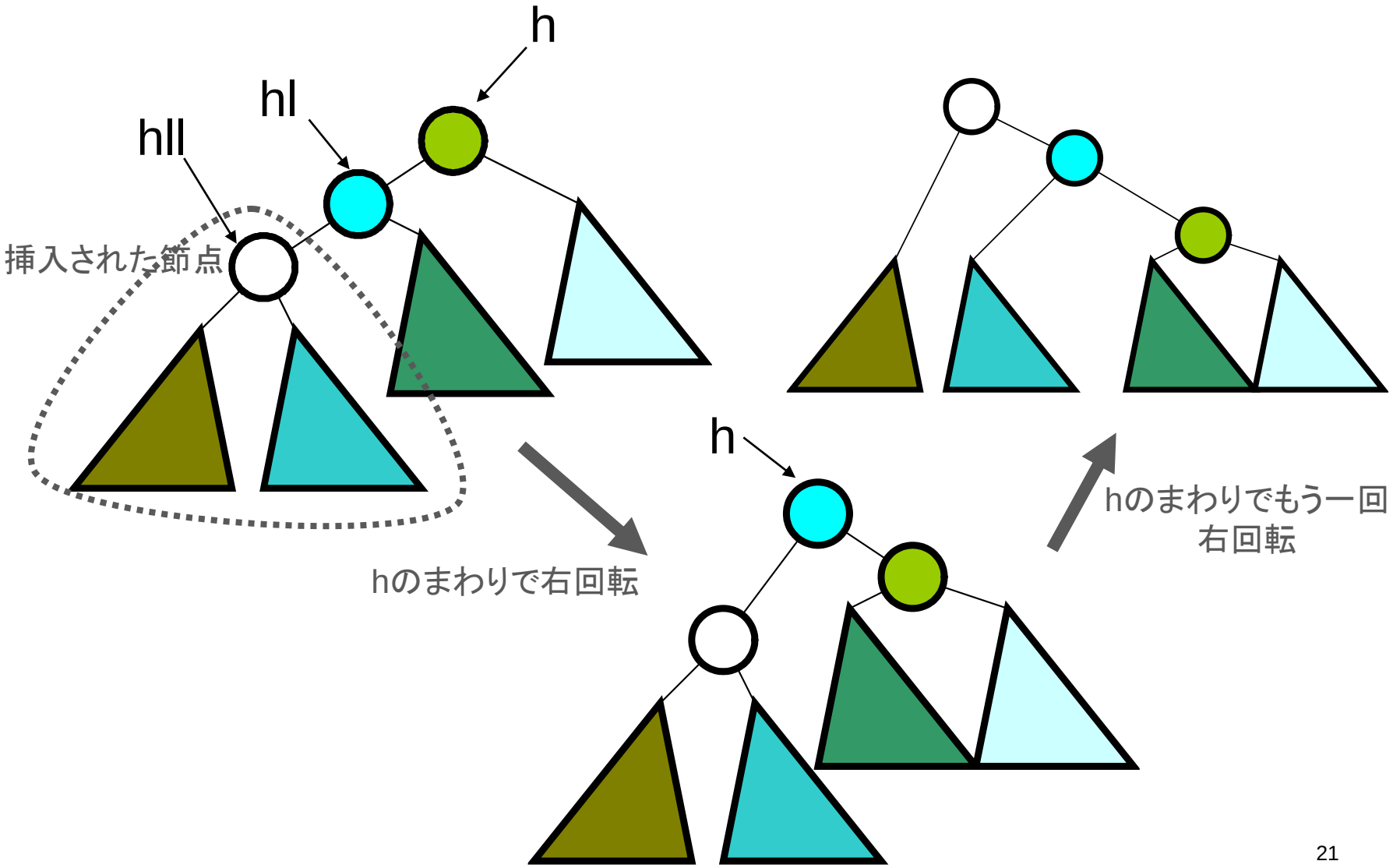
スプレイBST

- すなわち,
  - 根から順に新たな節点の挿入する位置を探索していき
  - 2回連続して左向き(右向き)に分岐したら，その部分木の根において，2回右回転(2回左回転)
  - 左向き，右向き(右向き，左向き)に分岐したら，根への挿入と同じで，下から上に向かって，左回転，右回転(右回転，左回転)

スプレイBST

```
link splay(link h, Item item)
  { Key v = key(item);
    if (h == z) return NEW(item, z, z, 1);
    if (less(v, key(h->item)))
      {
        if (h->l == z) return NEW(item, z, h, h->N+1);
        if (less(v, key(h->l->item)))
          { h->l->l = splay(h->l->l, item); h = rotR(h);}
        else
          { h->l->r = splay(h->l->r, item); h->l = rotL(h->l);}
        return rotR(h);
      }
    else
      {
        if (h->r == z) return NEW(item, h, z, h->N+1);
        if (less(key(h->r->item), v))
          { h->r->r = splay(h->r->r, item); h = rotL(h); }
        else
          { h->r->l = splay(h->r->l, item); hr = rotR(h->r); }
        return rotL(h);
      }
  }
void STinsert(Item item)
  { head = splay(head, item); }
```

スプレイBST



スプレイBST

hl: h->l, hll: h->l->l, hlr: h->l->r

```
if (less(v, key(h->item)))
      {
        if (hl == z) return NEW(item, z, h, h->N+1);
        if (less(v, key(hl->item)))
          { hll = splay(hll, item); h = rotR(h); }
        else
          { hlr = splay(hlr, item); hl = rotL(hl); }
        return rotR(h);
      }
```

ある部分木の根hの左に，挿入されるとき

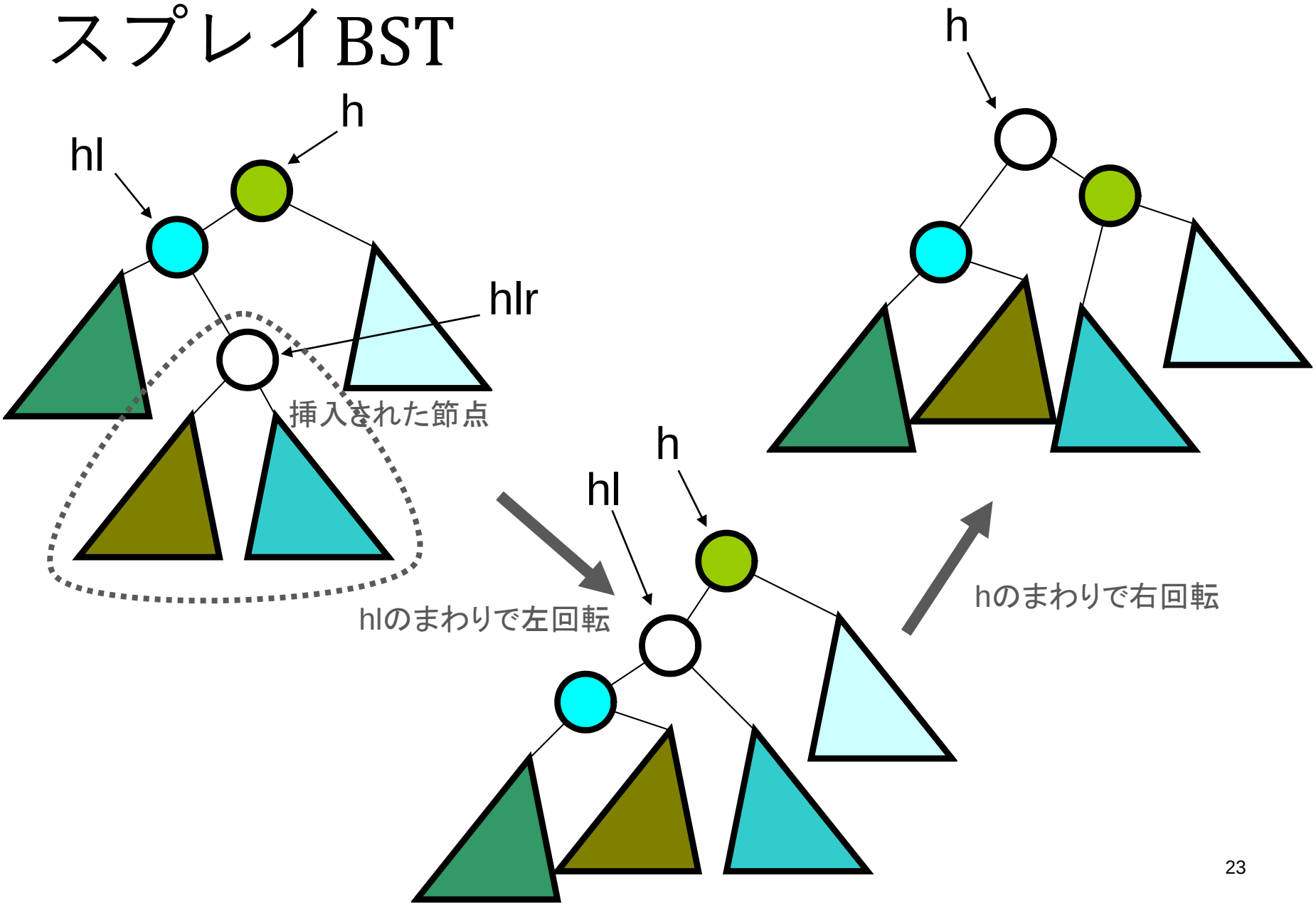
さらに左に挿入されるとき

根の左の左に，挿入する節点を根とした部分木を持って来て

根で右回転

根でもう一回右回転

スプレイBST



スプレイBST

hl: h->l, hll: h->l->l, hlr: h->l->r

```
if (less(v, key(h->item)))
     {
       if (hl == z) return NEW(item, z, h, h->N+1);
       if (less(v, key(hl->item)))
         { hll = splay(hll, item); h = rotR(h); }
       else
         { hlr = splay(hlr, item); hl = rotL(hl); }
       return rotR(h);
     }
```

ある部分木の根hの左に，挿入されるとき

根の左の右に挿入されるとき

根の左の節点で左回転

根でもう一回右回転

根の左の右に，挿入する節点を根とした部分木を持って来て

スプレイBST

- 性質13.4
  - はじめ空であった木にN回挿入してスプレイBSTを作るときはO(NlgN)の比較を行う

スプレイBST

- 性質13.4は(教科書の性質13.5も)，均し性能の1つである
  - N個の項目を挿入したとき，平均的に挿入の手間はO(lgN)である
  - 総実行時間が，O(NlgN)で抑えられていることから導かれる
    - これは，確率的な平均ではない
    - 必ずしも，各回の挿入に対して，O(lgN)回の比較を行うのみということを保証しているわけでない
    - ただ，N個の節点を挿入するときの総実行時間はO(NlgN)で抑えられている

均し性能の解析

- 集計法
  - n回の操作の最悪計算量を算出し，その平均を求める方法.
- ポテンシャル法
  - 後述

（例）スタック操作

- スタック操作
  - プッシュ：末尾に値を一つ追加する.
  - ポップ：末尾から値を一つ取り出す.
  - kポップ：末尾から値を k 個取り出す．スタックに含まれる値の数sがkより小さい場合は，s個の値を取り出す.
- 雑な解析
  - N回の操作でスタックサイズは最大Nとなりうるが，この場合，kポップの計算量はO(N)となる.
  - kポップの最悪計算量はO(N)のため，N回kポップを実施するとO(N^2)の計算量が必要．よって，１回あたりの最悪計算量はO(N)

（例）スタック操作

- kポップの計算量はO(N)となりうる.
- 空のスタックに対してN回のスタック操作をした場合，プッシュされる値の総数は高々N.
- これらすべてを取り出すのに呼び出されるポップの回数は，kポップ内のポップも含めて高々N.
- 合計N回の操作のうちプッシュ，ポップ，kポップをどのような割合で用いたとしても，全体の計算量はO(N)で抑えられ，1回あたりの計算量はO(1)となる.

ポテンシャル法

- ポテンシャル法では，一回の操作に必要な均しコストと実際に各操作で必要となるコストの差分を，**ポテンシャル**と呼ぶスコアによって記述して分析.
- ポテンシャルは，ある時点のデータ構造全体に与えられ．ある操作を行う前後のポテンシャルの差分により，均しコストと実コストの差分を吸収.

ポテンシャル法

- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - $\hat{c_i}$：均しコスト，Φ：ポテンシャル関数
  - ある操作を行った後のデータ構造D_iと前のデータ構造D_{i-1}
  - c_i：ある操作に必要な実コスト
- $\Sigma_{i=1}^{n} \hat{c_i} = \Sigma_{i=1}^{n} \{c_i + \Phi(D_i) - \Phi(D_{i-1})\}$
  $= \Sigma_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$
- 任意のnについて，$\Phi(D_n) \geq \Phi(D_0)$であれば均しコストは実コストの上界.

（例）スタック操作

- $\Phi(D_i)$は，スタックDiに含まれる値の総数を与える関数と定義.
  - 例：S = (1, 4, 10, 5, 8) の時ϕ(S)=5
- ポテンシャル法による解析
  - スタックサイズは非負なので$\Phi(D_n) \geq 0$
  - 初期状態は空とすれば$\Phi(D_0) = 0$，よって$\Phi(D_n) \geq \Phi(D_0)$

（例）スタック操作

- プッシュ
  - $\Phi(D_i) - \Phi(D_{i-1}) = 1$
  - $\hat{c}_i = 1 + 1 = 2$
- ポップ
  - $\Phi(D_i) - \Phi(D_{i-1}) = -1$
  - $\hat{c}_i = 1 - 1 = 0$
- kポップ
  - $\Phi(D_i) - \Phi(D_{i-1}) = -k$
  - $\hat{c}_i = k - k = 0$

プッシュすると，スタックサイズが1増加する.

実コストは1

ポップすると，スタックサイズが1減少する.

実コストは1

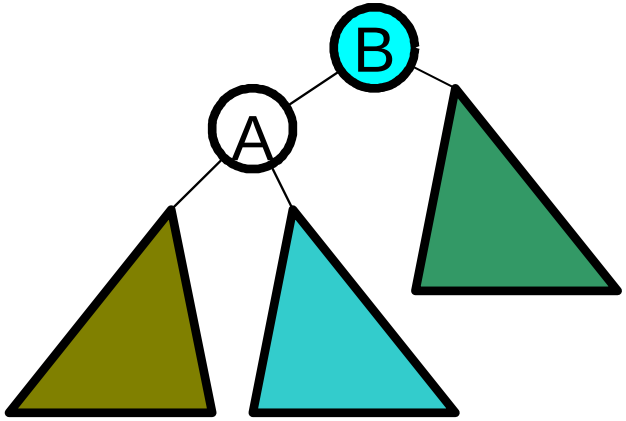
ポップすると，スタックサイズがk減少する.

実コストはk

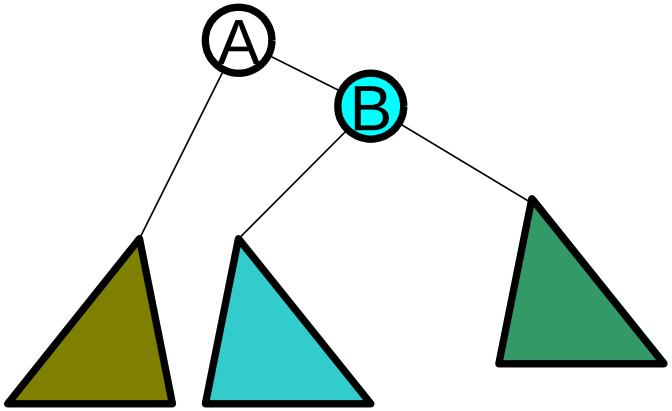
スプレイ木における回転操作の解析

- 節点xを根とする木の節点数： $s(x)$
- $r(x) = \lg\big(s(x)\big)$： xのランクと呼ぶ
- $\Phi(T) = \Sigma_{x \in T}\, r(x)$：Tのポテンシャル

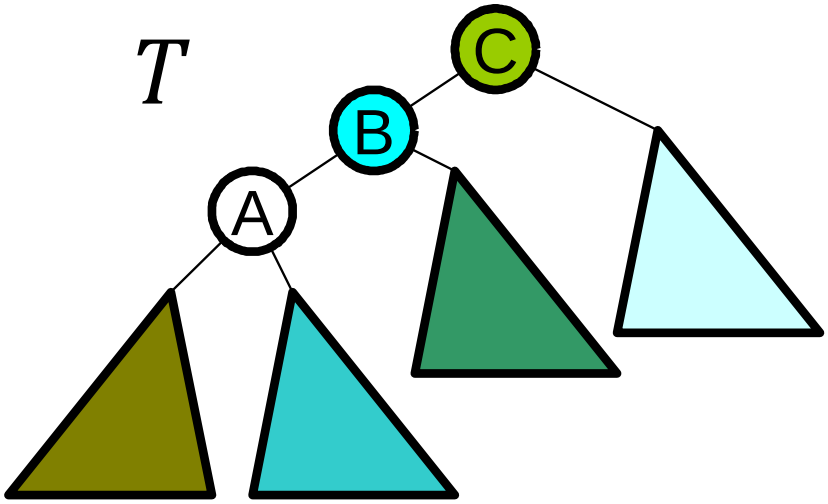
T

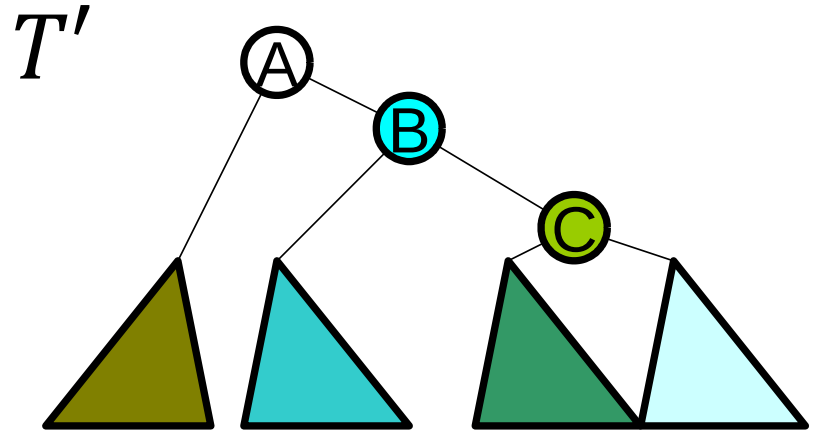


T'



$$\hat{c} = 1 + \Phi(T') - \Phi(T)$$
$$= 1 + r'(A) + r'(B) - r(A) - r(B)$$
$$= 1 + r'(B) - r(A) \leq 1 + r'(A) - r(A)$$





$$\hat{c} = 2 + \Phi(T') - \Phi(T)$$
$$= 2 + r'(A) + r'(B) + r'(C) - r(A) - r(B) - r(C)$$
$$= 2 + r'(B) + r'(C) - r(A) - r(B)$$
$$\leq 2 + r'(A) + r'(C) - 2r(A)$$

(lg x+lg y)/2 ≤ lg(x+y)/2 より

$$\frac{r(A) + r'(C)}{2} \leq \lg \frac{s(A) + s'(C)}{2} \leq \lg \frac{s'(A)}{2} = r'(A) - 1$$
$$r'(C) \leq 2r'(A) - r(A) - 2$$

以上より，　$\hat{c} \leq 3(r'(A) - r(A))$

スプレイ操作の性能

- 定義

  スプレイ木の節点xを回転により根に移動することをxに対するスプレイ操作と呼ぶ.

- 補助定理

  節点xに対するスプレイ操作の均し計算量はO(lgN)である.

- 証明

  スプレイ操作の均しコストは，xが根tに移動するまでの回転操作の総コストのため，以下で抑えられる.

$$1 + \Sigma\, 3\left(r^{i+1}(x) - r^{i}(x)\right) = 1 + 3\big(r(t) - r(x)\big) \leq 3\lg N + 1$$

  よってO(lgN)となる.

スプレイ木への挿入

- 定理

  スプレイ木に対する挿入の均し計算量はO(lgN)である．

- 証明

  補助定理より，挿入節点に対するスプレイ操作の計算量はO(lgN)．挿入した節点の全先祖を根に向かう順に$x_1, x_2, \ldots, x_k$と記述すると，挿入によるポテンシャルの増分は$\Sigma \lg(s(x_i)+1) - \lg\big(s(x_i)\big) = \lg \prod \frac{s(x_i)+1}{s(x_i)}$となるが，$s(x_1)+1 \leq s(x_2)$のため$\lg \frac{s(x_2)}{s(x_1)} \cdot \ldots \cdot \frac{s(x_k)}{s(x_{k-1})} \cdot \frac{s(x_k)+1}{s(x_k)} = \lg \frac{s(x_k)+1}{s(x_1)}$で抑えることができる．よって増分はO(lgN)となり，節点挿入に必要な均し計算量はO(lgN)となる．

目次

- 平衡木
- ランダム木
- スプレイ木