アルゴリズムとデータ構造

記号表と2分探索木（その２）

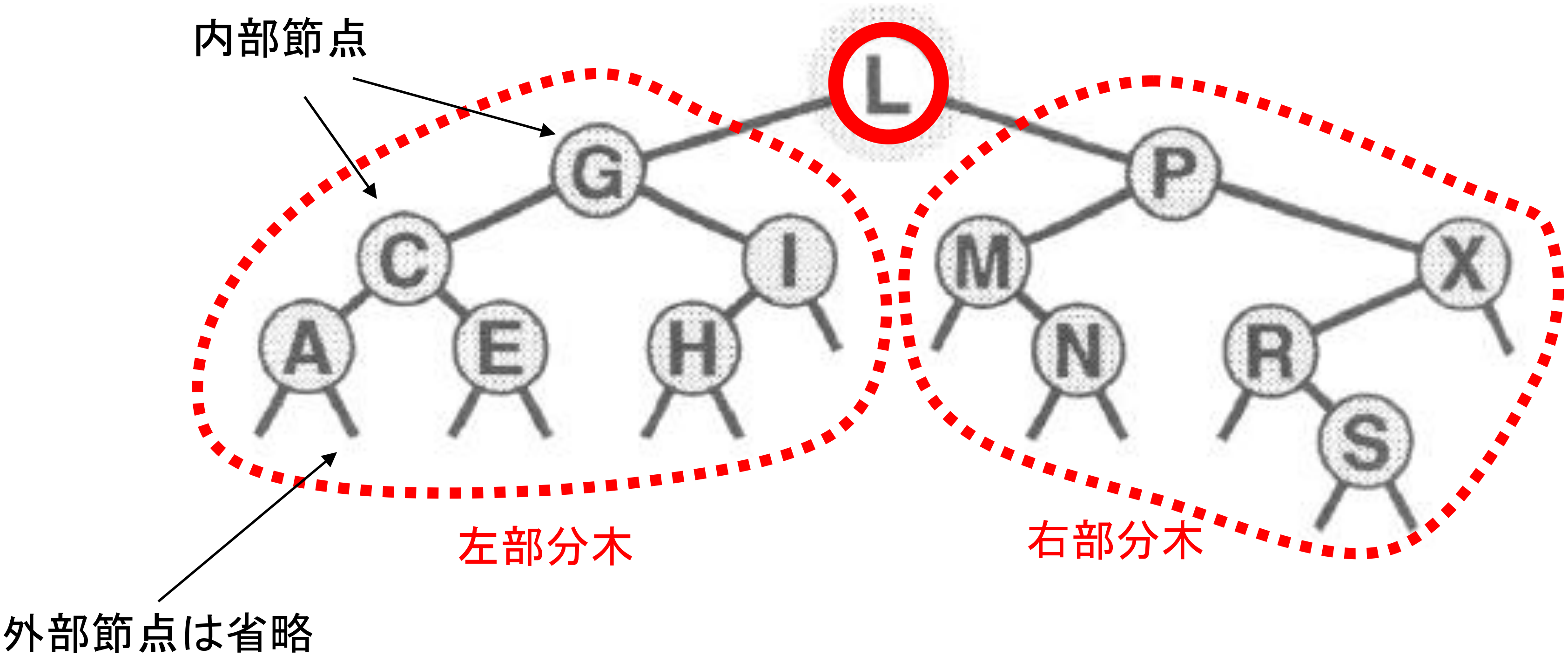
目次

- 根への挿入
  - 右回転
  - 左回転

2分探索木(復習)

- 挿入操作が高くつくという問題を解決するために，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

2分探索木(復習)

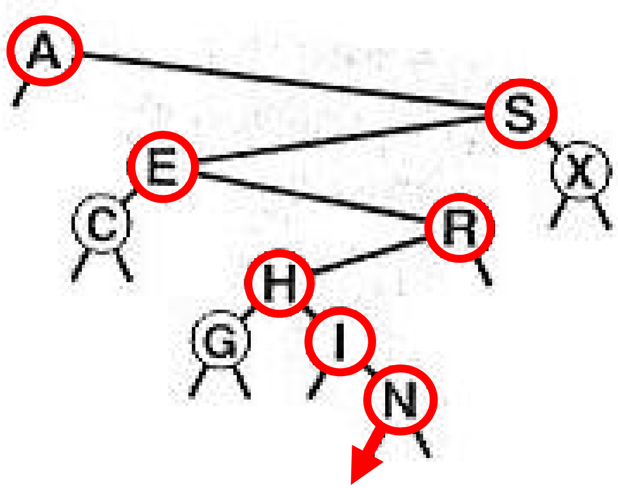


内部節点のみ表示している．外部節点は省略してある

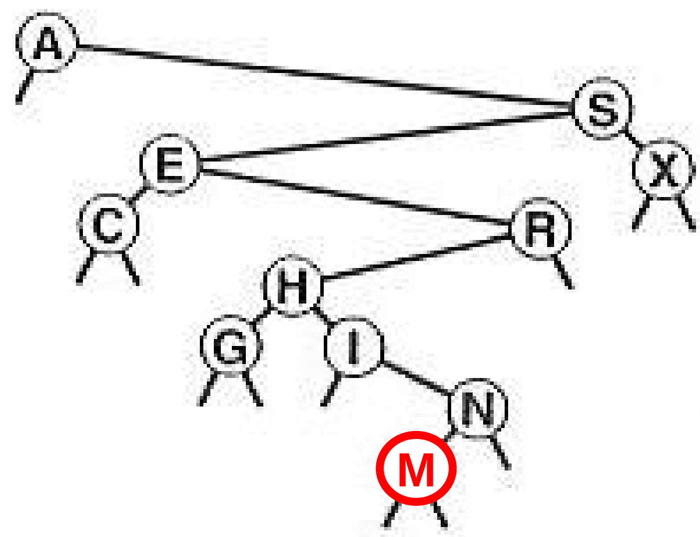
これまでの挿入(復習)

- BSTに新しい項目を挿入する手続き
  - 探索し，不成功した場所を見つける
  - その場所に節点を挿入する

Mに対する不成功探索



Mの挿入



これまでの挿入(復習)

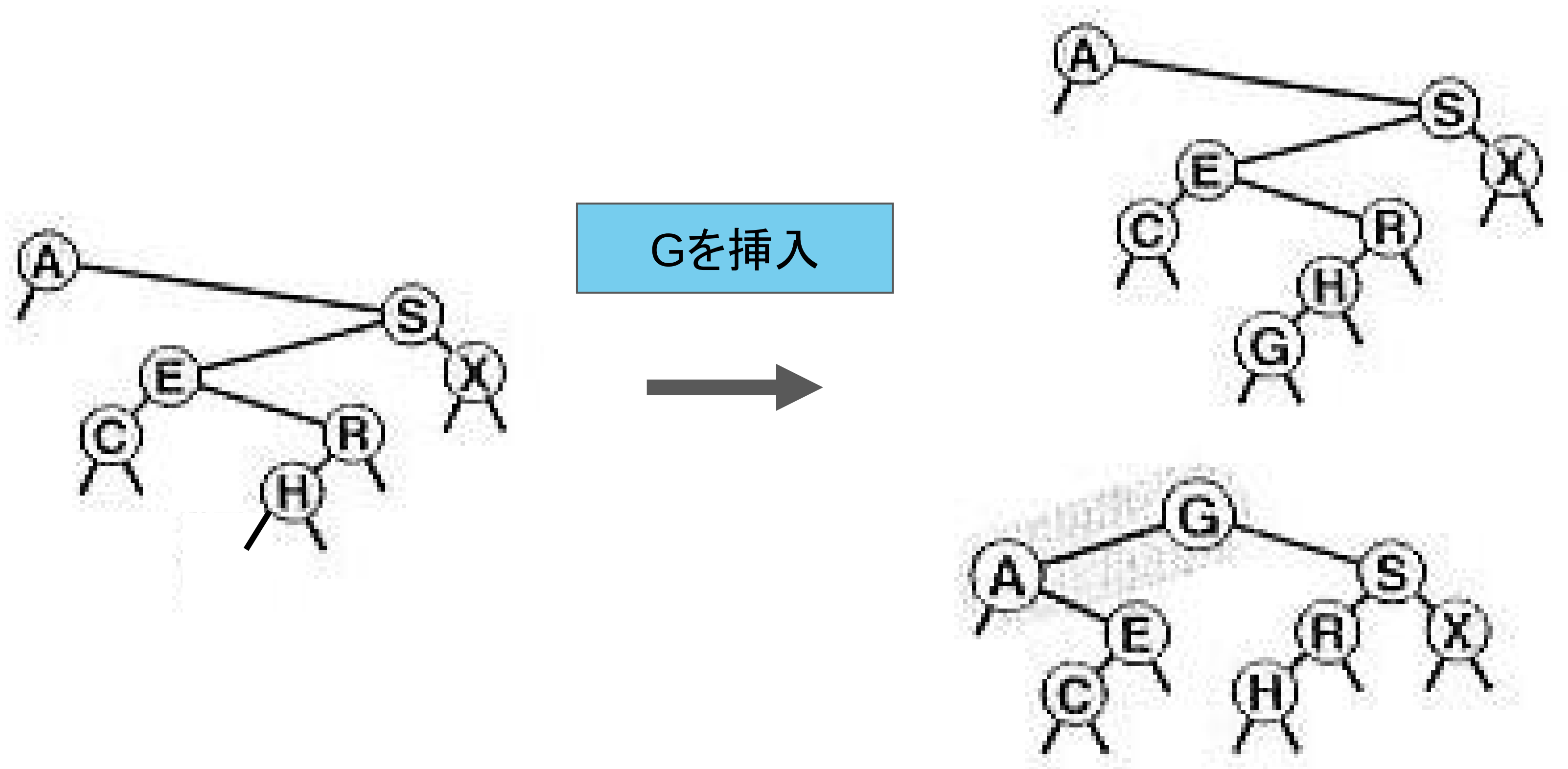
hを根とする部分木にうまく項目**item**を挿入する関数

```
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if 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); }
```

根への挿入

- 木の底ではなく，木の根に新しい節点を挿入することはできないか？



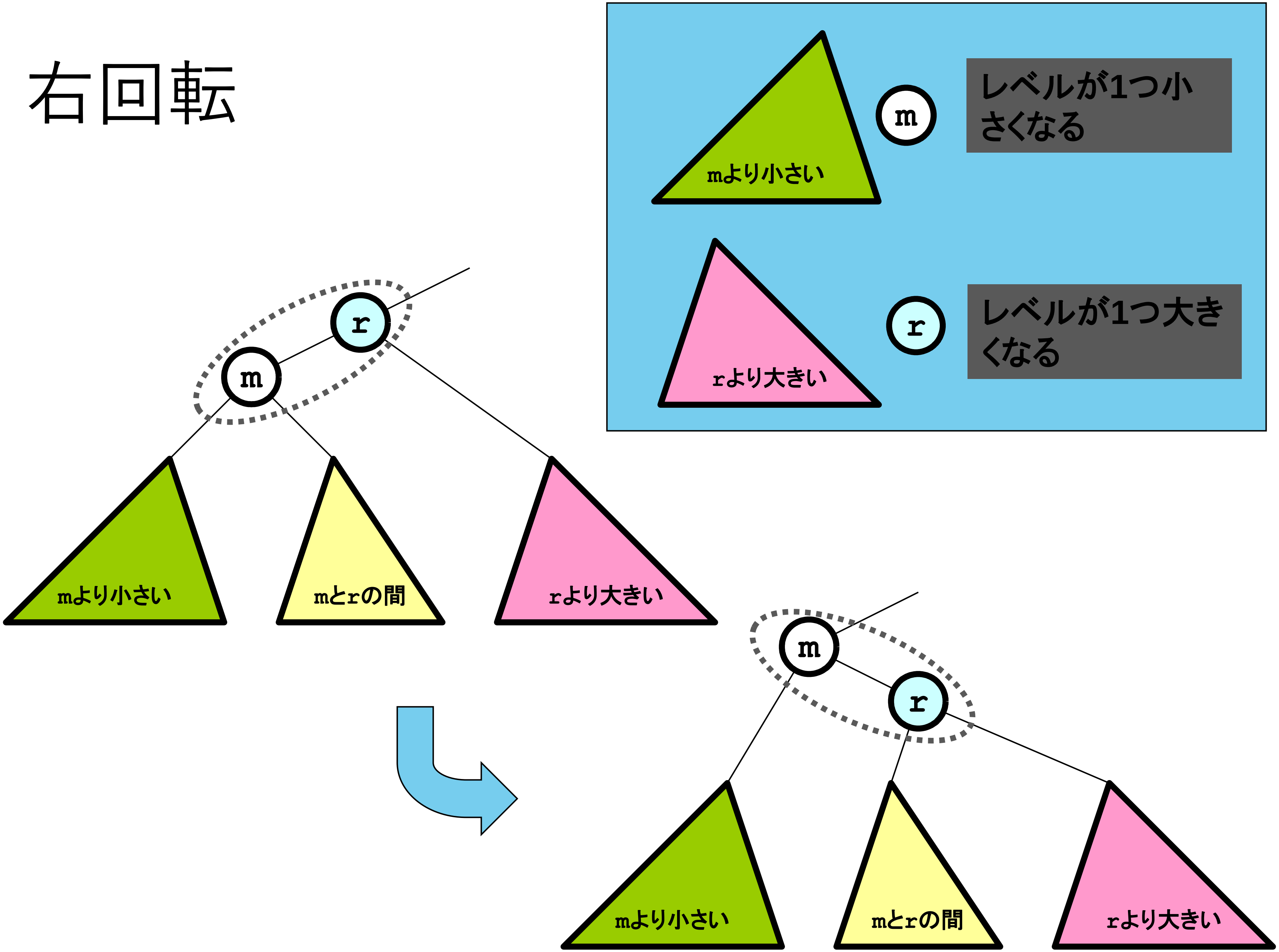
根への挿入

- 木の底ではなく，木の根に新しい節点を挿入することはできないか？
- 利点
  - 一般に，最近にinsertされた項目は，その後すぐにsearchされる可能性が高い
  - 根に近いところに，そのような項目があれば，探索の時間が短縮することが見込まれる

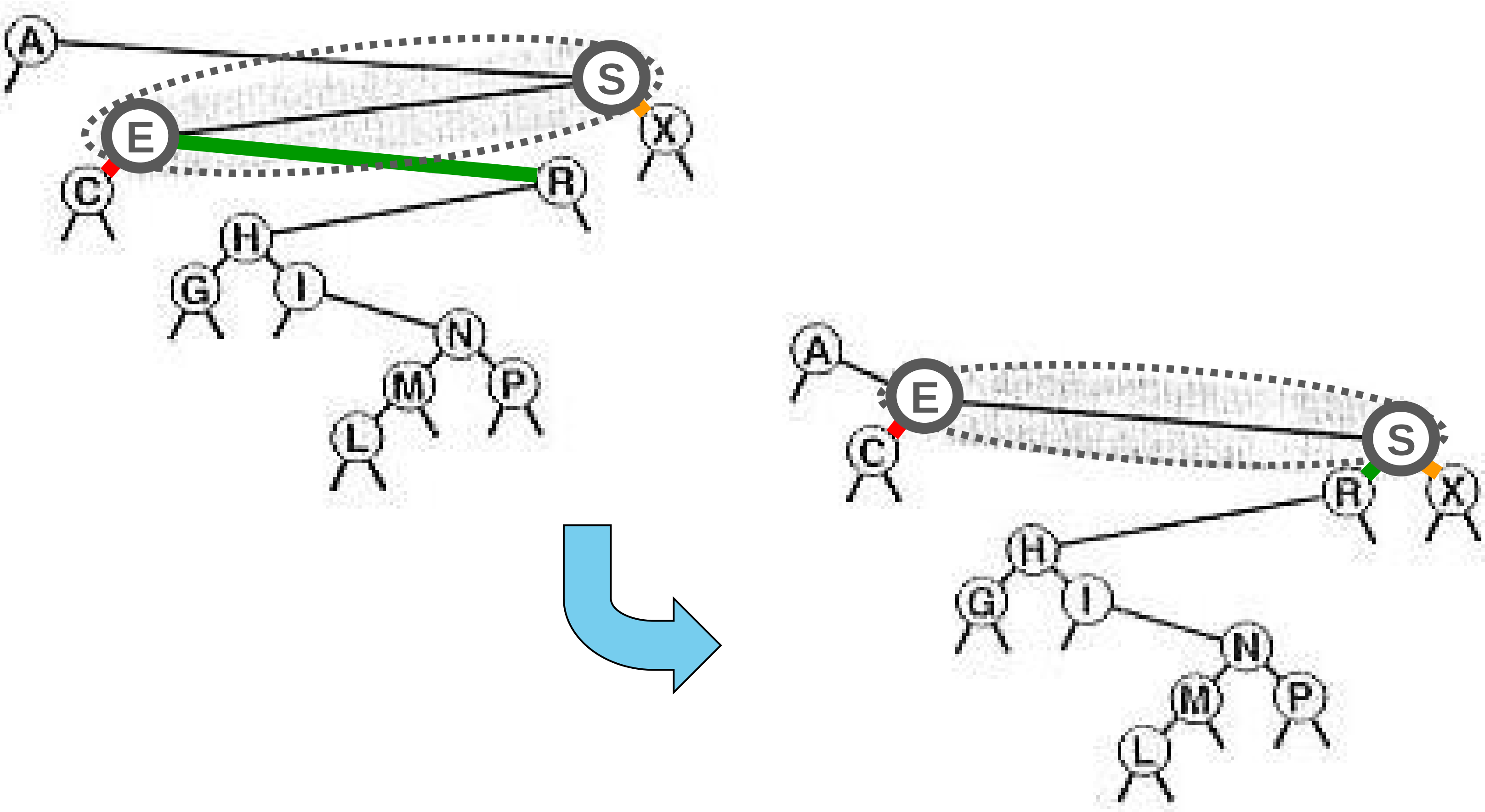
根への挿入

- 木の底ではなく，木の根に新しい節点を挿入することはできないか？
  - 根への挿入には回転を使う

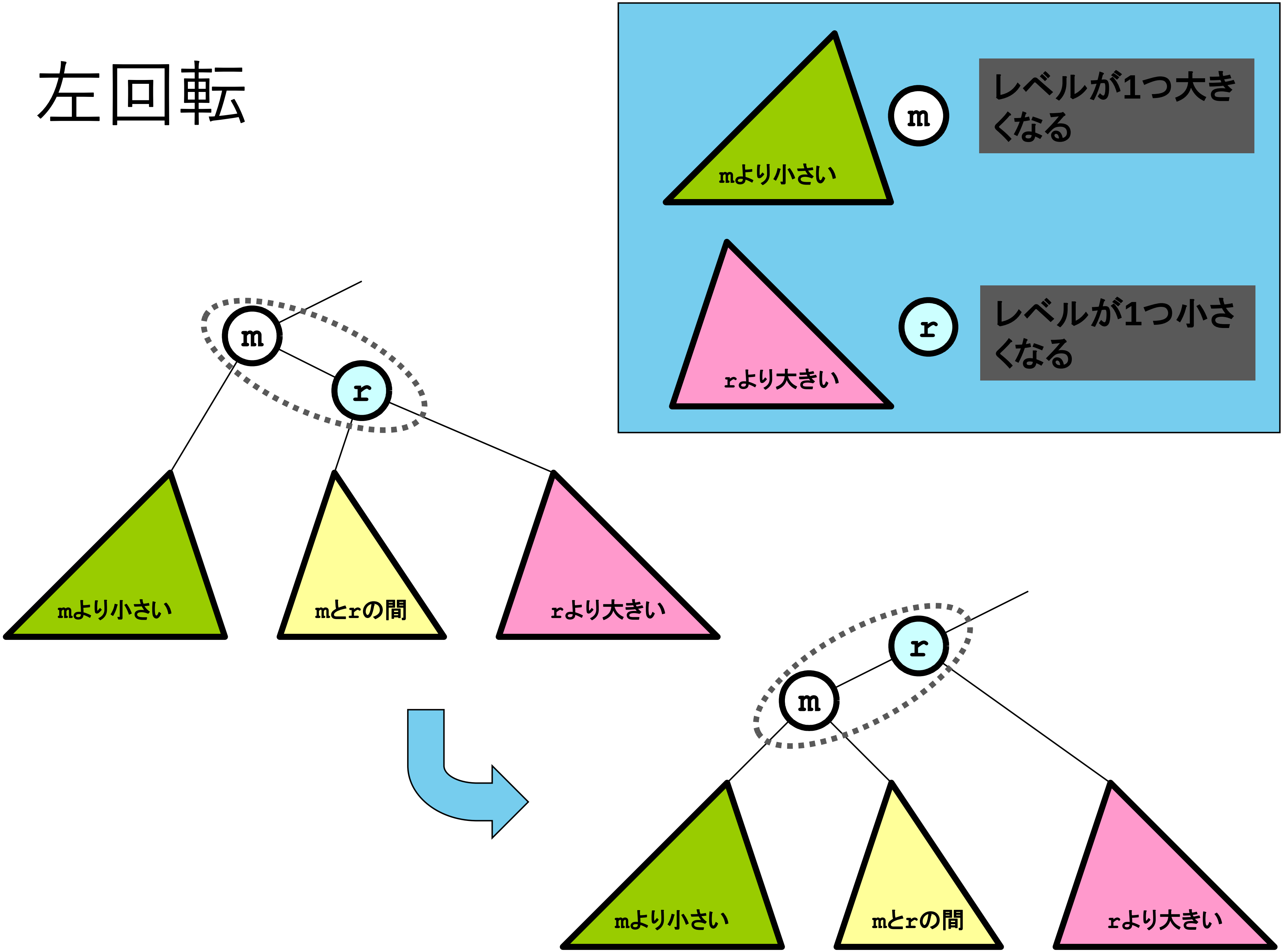
右回転


mより小さい
m
レベルが1つ小さくなる
rより大きい
r
レベルが1つ大きくなる
r
m
mより小さい
mとrの間
rより大きい
m
r
mより小さい
mとrの間
rより大きい

右回転

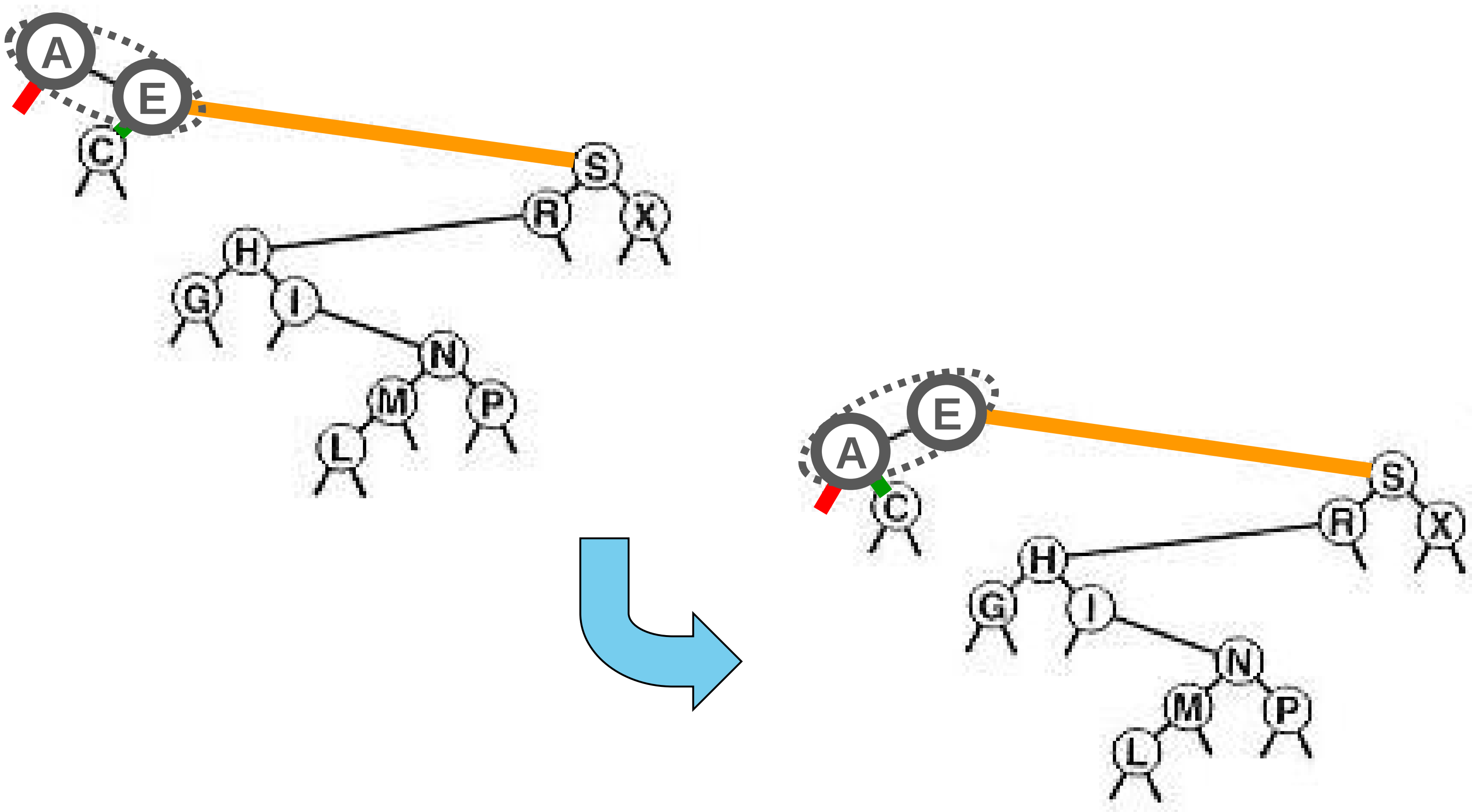


左回転



mより小さい
m
レベルが1つ大きくなる
rより大きい
r
レベルが1つ小さくなる
m
r
mより小さい
mとrの間
rより大きい
r
m
mより小さい
mとrの間
rより大きい

左回転



右回転

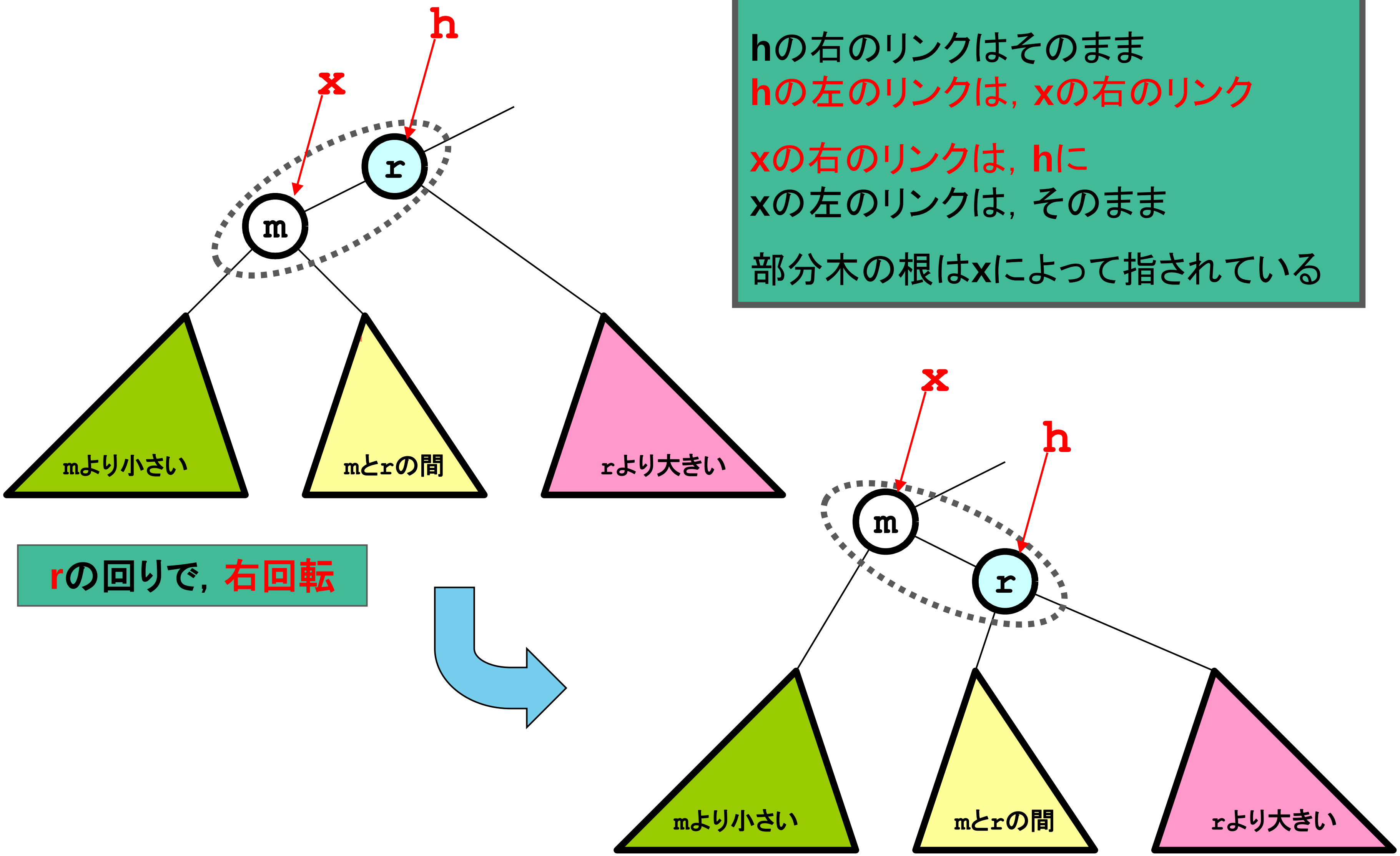
hは与えられるとする

xはhの左のリンク

hの右のリンクはそのまま
hの左のリンクは，xの右のリンク

xの右のリンクは，hに
xの左のリンクは，そのまま

部分木の根はxによっている



rの回りで，右回転

右回転

```
link rotR(link h)
  { link x = h->l; h->l = x->r; x->r = h;
    return x; }
```

hは与えられるとする

xは**h**の左のリンク

hの右のリンクはそのまま
hの左のリンクは，**x**の右のリンク

xの右のリンクは，**h**に
xの左のリンクは，そのまま

部分木の根は**x**によって指されている

左回転

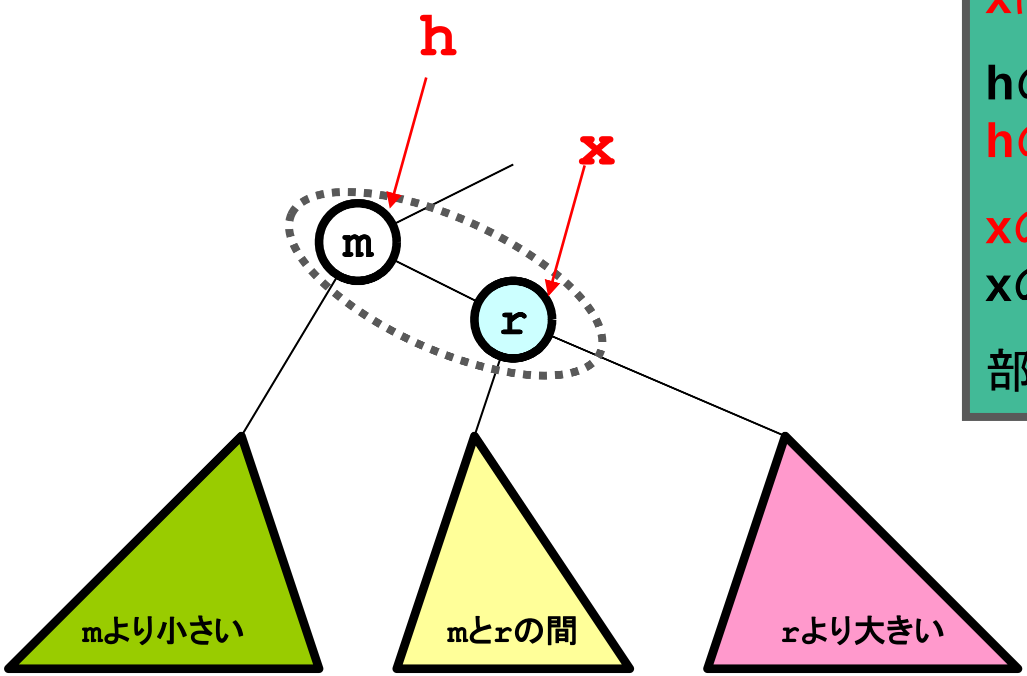
hは与えられるとする

xはhの右のリンク

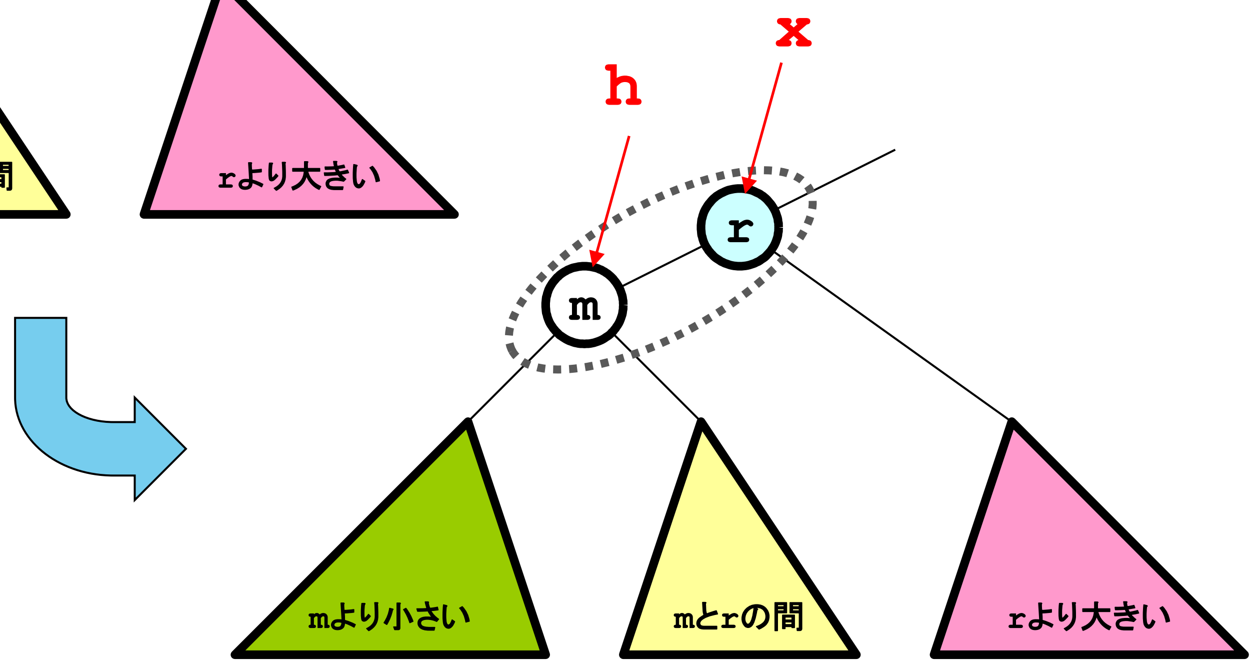
hの左のリンクはそのまま
hの右のリンクは，xの左のリンク

xの左のリンクは，hに
xの右のリンクは，そのまま

部分木の根はxによって指されている



mの回りで，左回転



左回転

```
link rotL(link h)
  { link x = h->r; h->r = x->l; x->l = h;
    return x; }
```

hは与えられるとする

xは**h**の右のリンク

hの左のリンクはそのまま

hの右のリンクは，**x**の左のリンク

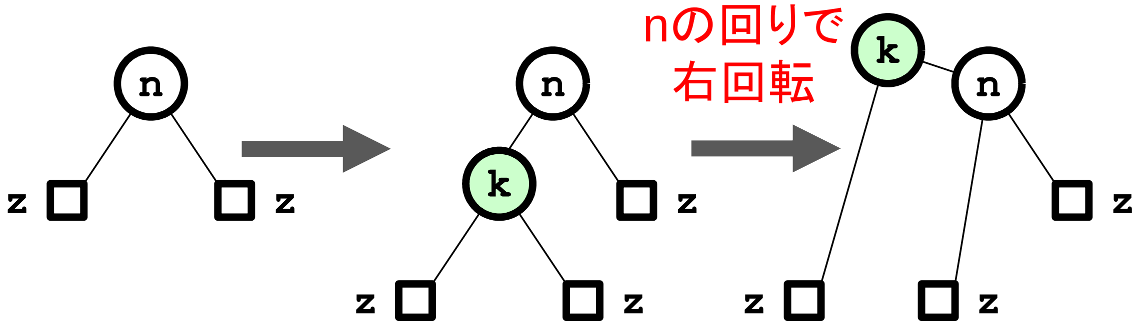
xの左のリンクは，**h**に

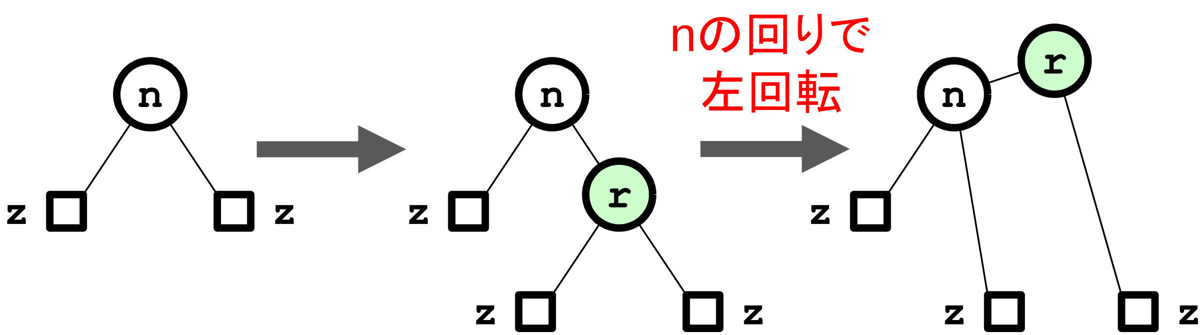
xの右のリンクは，そのまま

部分木の根**x**によって指されている

根への挿入

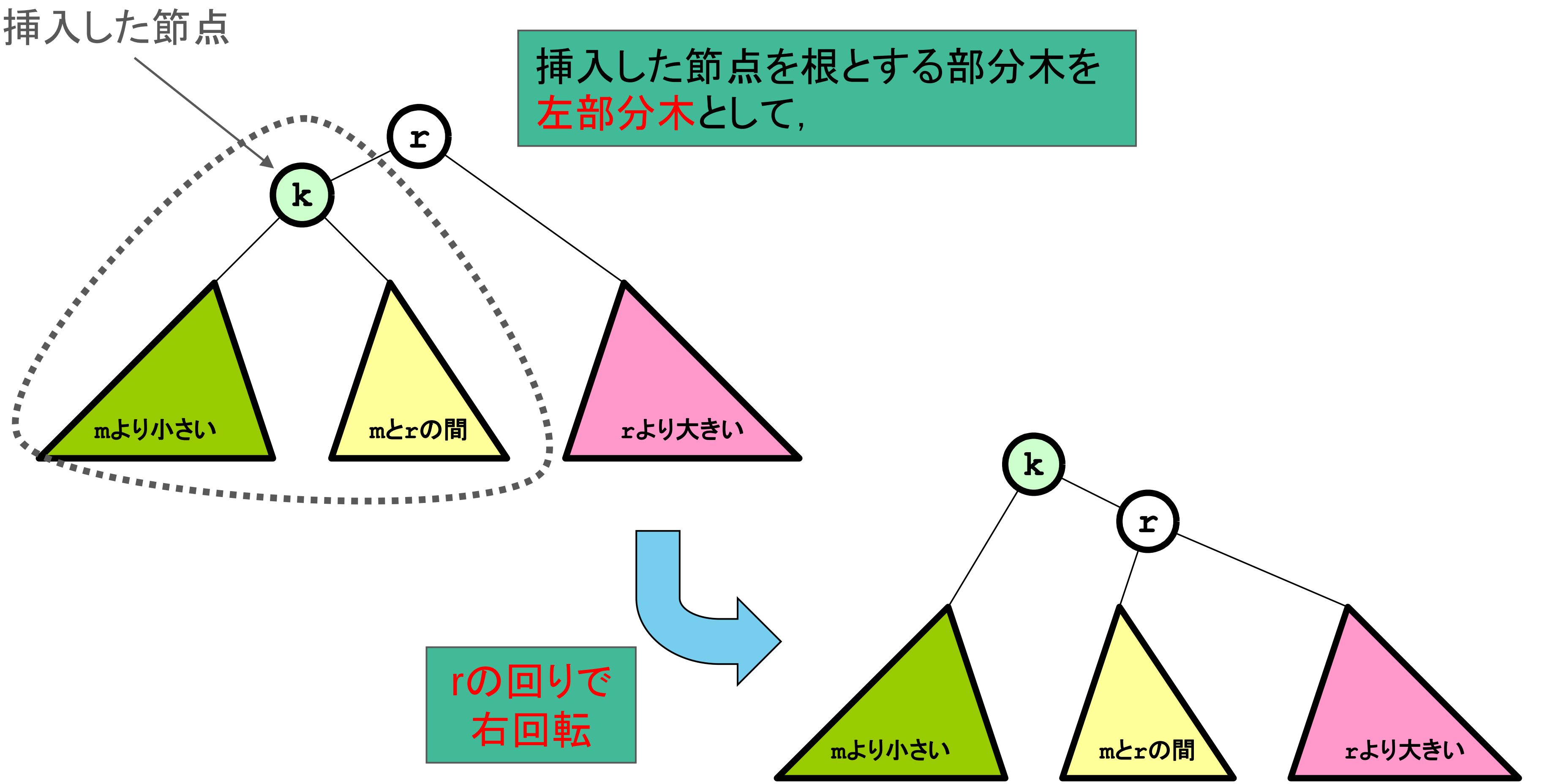
- 回転を使って根への挿入を考えよう
  - 木が1個の節点を持っていて，その底に新たな節点を挿入するとき





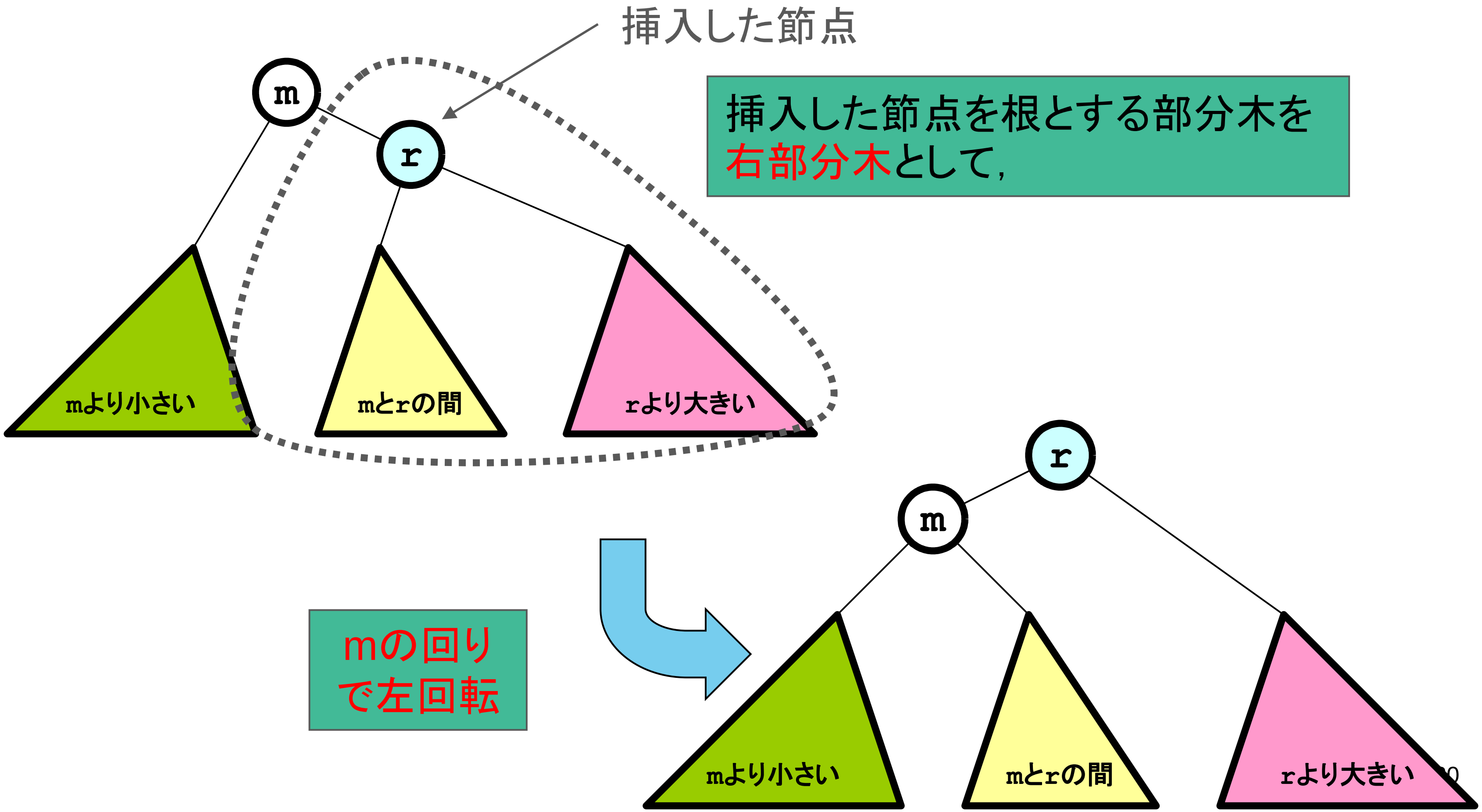
根への挿入

- 挿入した節点が根の左側にあれば・・・



根への挿入

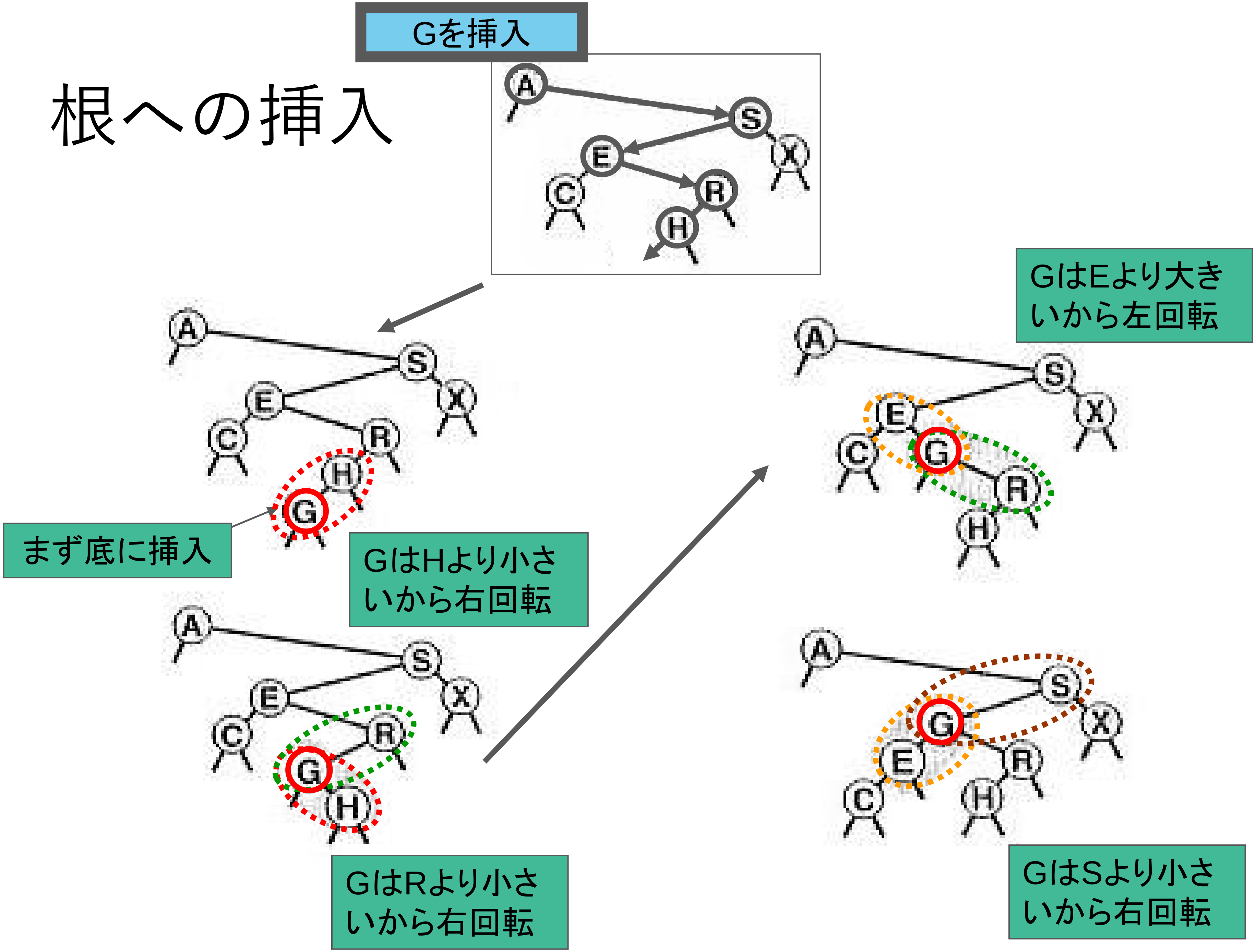
- 挿入した節点が根の右側にあれば・・・



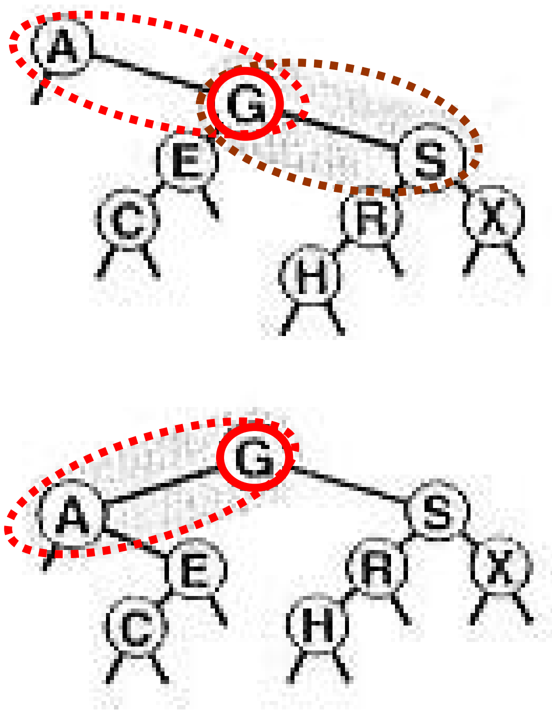
根への挿入

- つまり
  - まず，木の底に節点を挿入し，
  - 木の底の方から，順に節点**h**に対して，
    - **h**より挿入節点のキーが小さかったら(**h**の左に挿入されたら)
      - **hの左**に，「挿入された節点を根に持つ部分木」を持って来て，**hの回りに右回転(2ページ前のスライド)**
    - **h**より挿入節点のキーが大きかったら(**h**の右に挿入されたら)
      - **hの右**に，「挿入された節点を根に持つ部分木」を持って来て，**hの回りに左回転(1ページ前のスライド)**
  - これらの操作を，もとの木の根に至るまで続ければよい

根への挿入


Gを挿入
まず底に挿入
GはHより小さいから右回転
GはRより小さいから右回転
GはEより大きいから左回転
GはSより小さいから右回転

根への挿入



GはAより大き
いから左回転

根への挿入

hを根とする部分木にうまく項目**item**を挿入する関数

項目**item**は，回転により**h**の根に来る

ST_BST2.c (変更)

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

まず木の底へ挿入

hの回りで右回転

関数**insertT**により**h->l**に新たな節点を挿入した結果.

その結果，新たに挿入した節点は必ず**h->l**の根に来ている.

そのような部分木を，まず**h**の左部分木にし，その後，**h**の回りで右回転する.

ST_BST1.cのSTinsert 関数/insertR関数(bst1.pdfの35ページ目)を，これに置き換えると根への挿入になる．改めて，この変更を行ったものをST_BST2.cとする

根への挿入

ST_BST2.c (追加)

```
link rotR(link h)
  { link x = h->l; h->l = x->r; x->r = h;
    return x; }
link rotL(link h)
  { link x = h->r; h->r = x->l; x->l = h;
    return x; }
```

前ページのinsertTの前に，さらにこれら2つの関数を追加する

アルゴリズムとデータ構造

他のADT関数のBSTによる実現

他のADT関数のBSTによる実現

- 2分探索木を使って，以下の関数を実現することを考える
  - select: 選択
  - join: 結合
  - delete: 削除

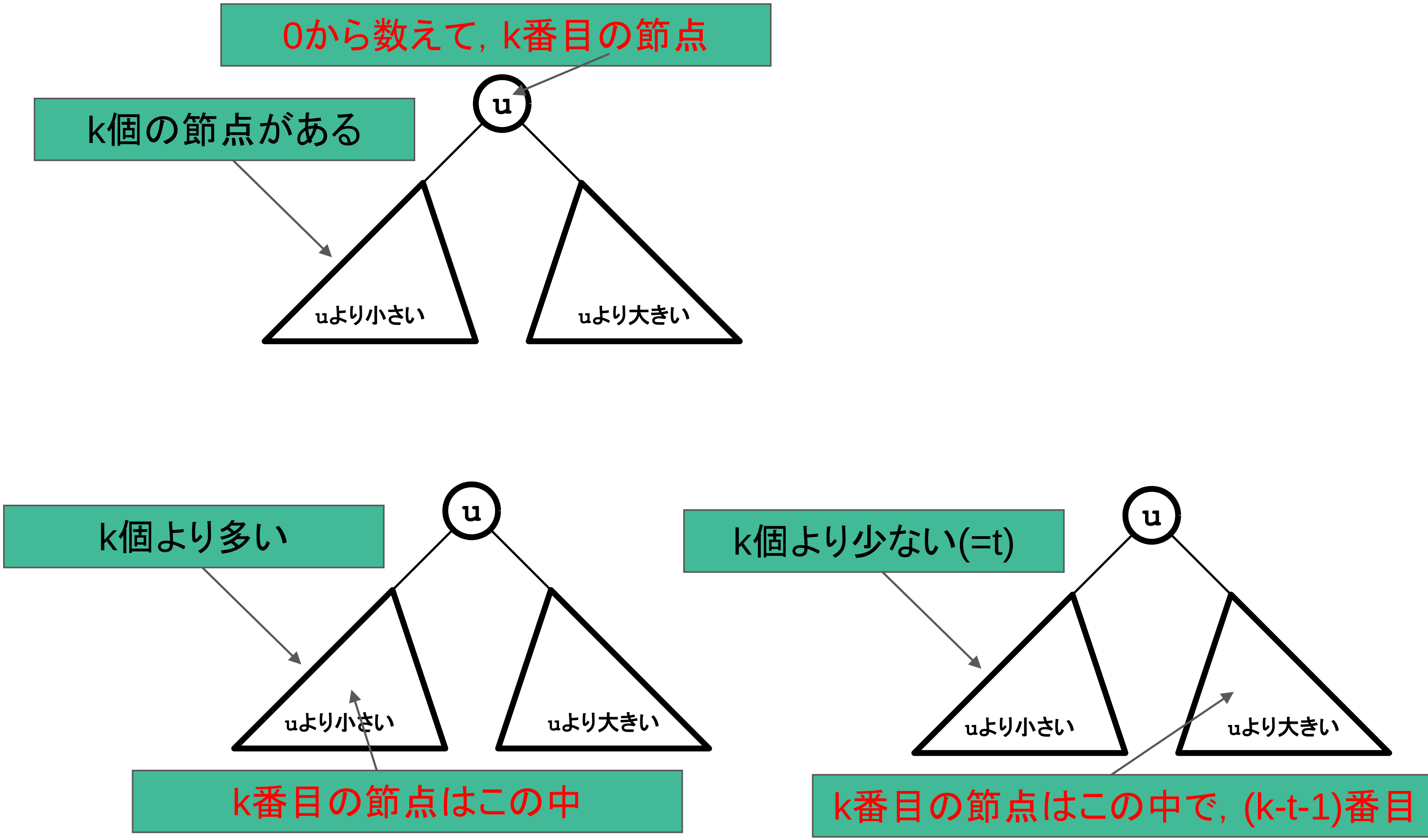
記号表抽象データ型(復習)

- 記号表抽象データ型に対する操作
  - insert: 新しい項目を挿入する
  - search: 与えられたキーを持つ項目を探索する
  - delete: 指定された項目を削除する
  - select: k番目の項目を選択する
  - sort: 記号表を整列する
  - join: 2つの記号表を合併して，1つの大きな記号表を作る
  - その他，初期設定(initialize)，空であるかの検査(test_if_empty)など

選択

- 前提
  - 添字は0から始まる
  - 従って，k=3のとき，4番目に小さいキーを得る
- 2分探索木にある項目のうち，k番目のキーを持つ項目を選択する
- 考え方
  - 今，ある部分木の根に注目する
    - 左部分木の節点の個数がk個ならば，根にある項目がk番目のキーを持つ
    - k個より多ければ，左部分木の中でk番目のキーを探せば良い
    - k個より少なければ(t個とする)，右部分木の中で(k-t-1)番目のキーを探せば良い

選択 - 考え方


0から数えて, k番目の節点
k個の節点がある
u
uより小さい
uより大きい
k個より多い
u
uより小さい
uより大きい
k番目の節点はこの中
k個より少ない(=t)
u
uより小さい
uより大きい
k番目の節点はこの中で, (k-t-1)番目

選択

```
Item selectR(link h, int k)
  { int t = h->l->N;
    if (h == z) return NULLitem;
    if (t > k) return selectR(h->l, k);
    if (t < k) return selectR(h->r, k-t-1);
    return h->item;
  }
Item STselect(int k)
  { return selectR(head, k); }
```

tに左部分木の節点数を代入

大きすぎなど，不正なkを与えた

左部分木の節点数によって，3通りの操作

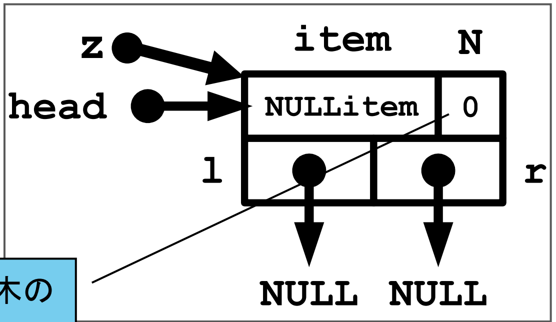
選択

- 注意
  - 2分探索木の各節点に，カウント欄を作っている
  - カウント欄を参照することで，部分木の節点数を簡単に実現している
    - カウント欄を持つことで余分なメモリを使用する
    - 選択操作は速くなる

```
int t = h->l->N;
```

選択 - BSTによる記号表

外部節点 □ z



その節点を根とする部分木の節点数の総数

ST_BST.c

ST.cに名前を変更して使用する

まず，STinitによって「**外部節点1つの空な木**」に初期化される

```
#include <stdlib.h>
#include "Item.h"
typedef struct STnode* link;
struct STnode { Item item; link l, r; int N };
static link head, z;
link NEW(Item item, link l, link r, int N)
  { link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r; x->N = N;
    return x;
  }
void STinit()
  { head = (z = NEW(NULLitem, 0, 0, 0)); }
int STcount() { return head->N; }
```

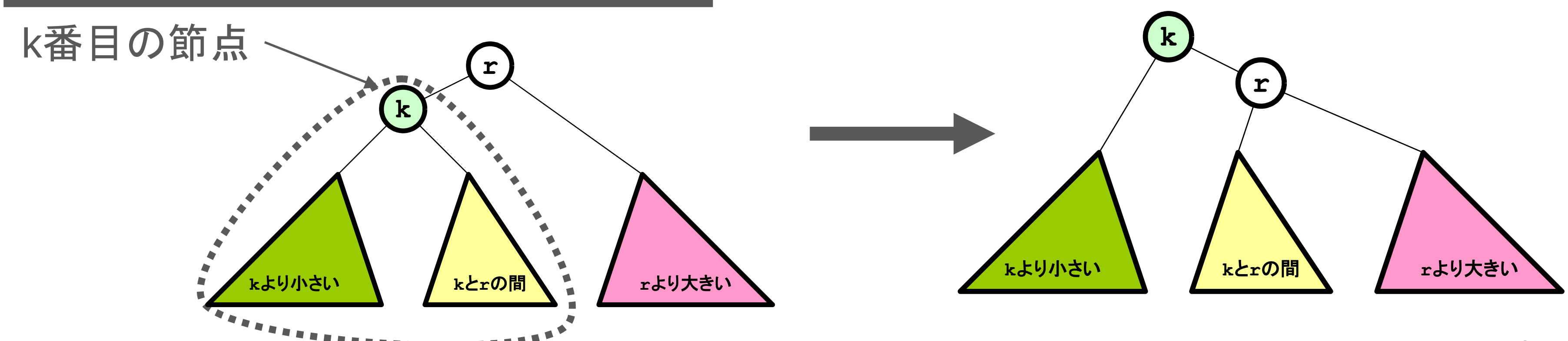
カウント欄

その節点を根とする部分木の節点数

分割

- 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

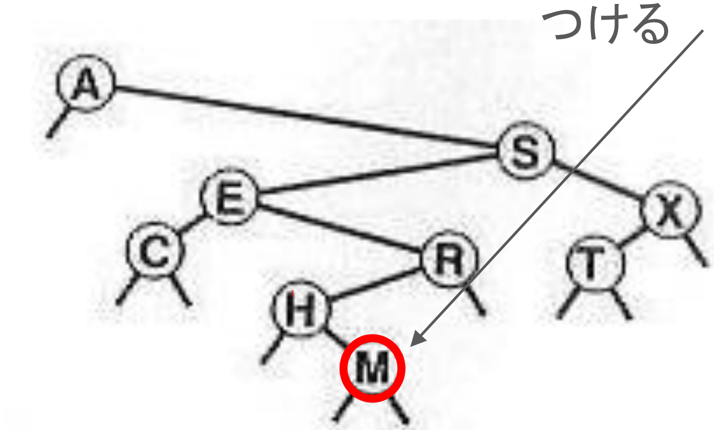


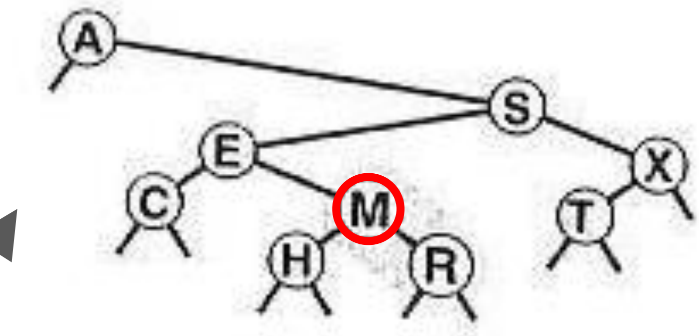
(1) まず左部分木の根にもってきて

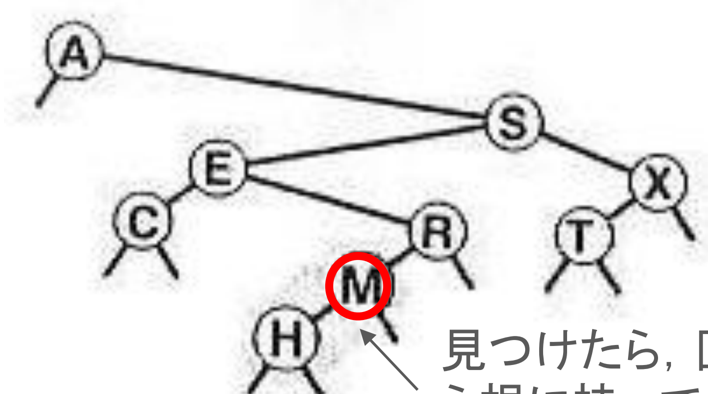
(2) 根のまわりで右回転

分割

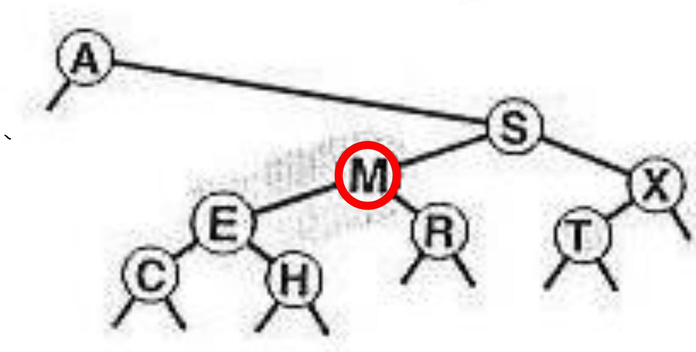
まずk=4番目の節点を見つける





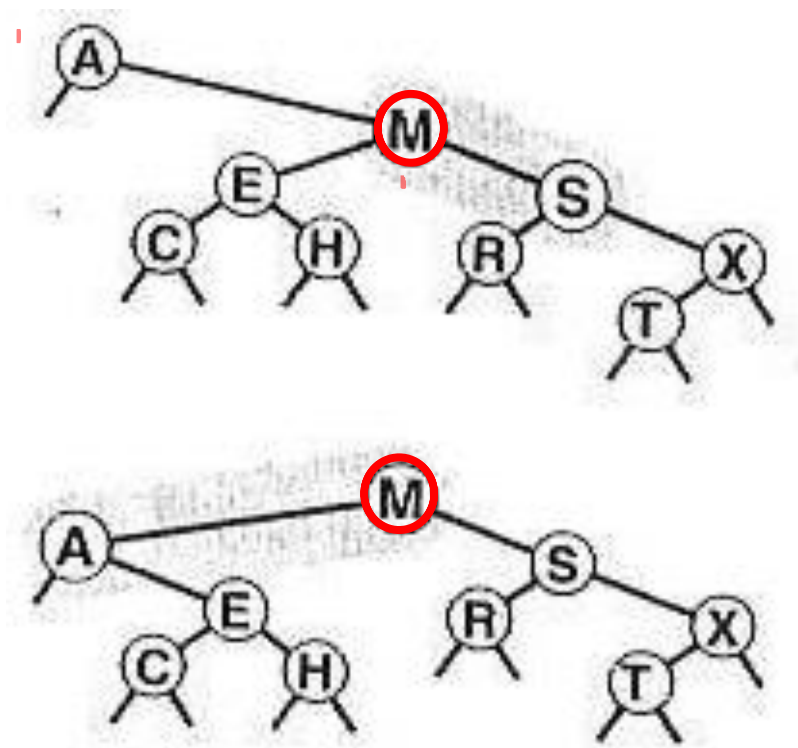


見つけたら，回転しながら根に持って来る



k=4番目(0から数えて)を見つけて，根に持って来る

分割



分割

```
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;
  }
```

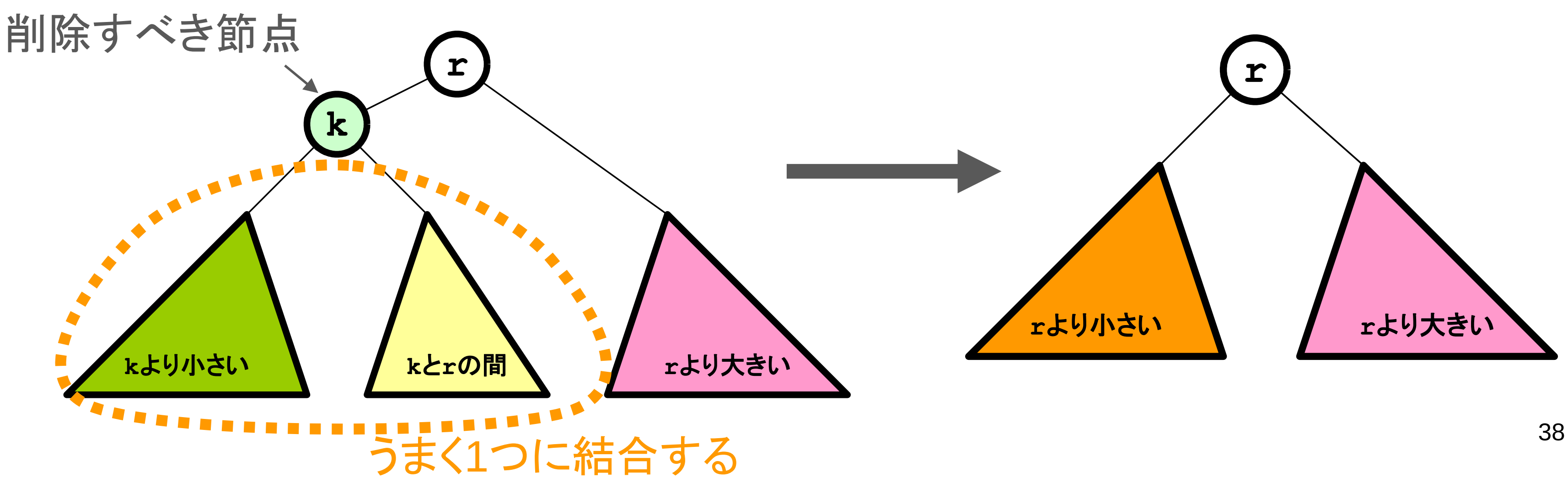
左部分木の根に持って来て

右回転

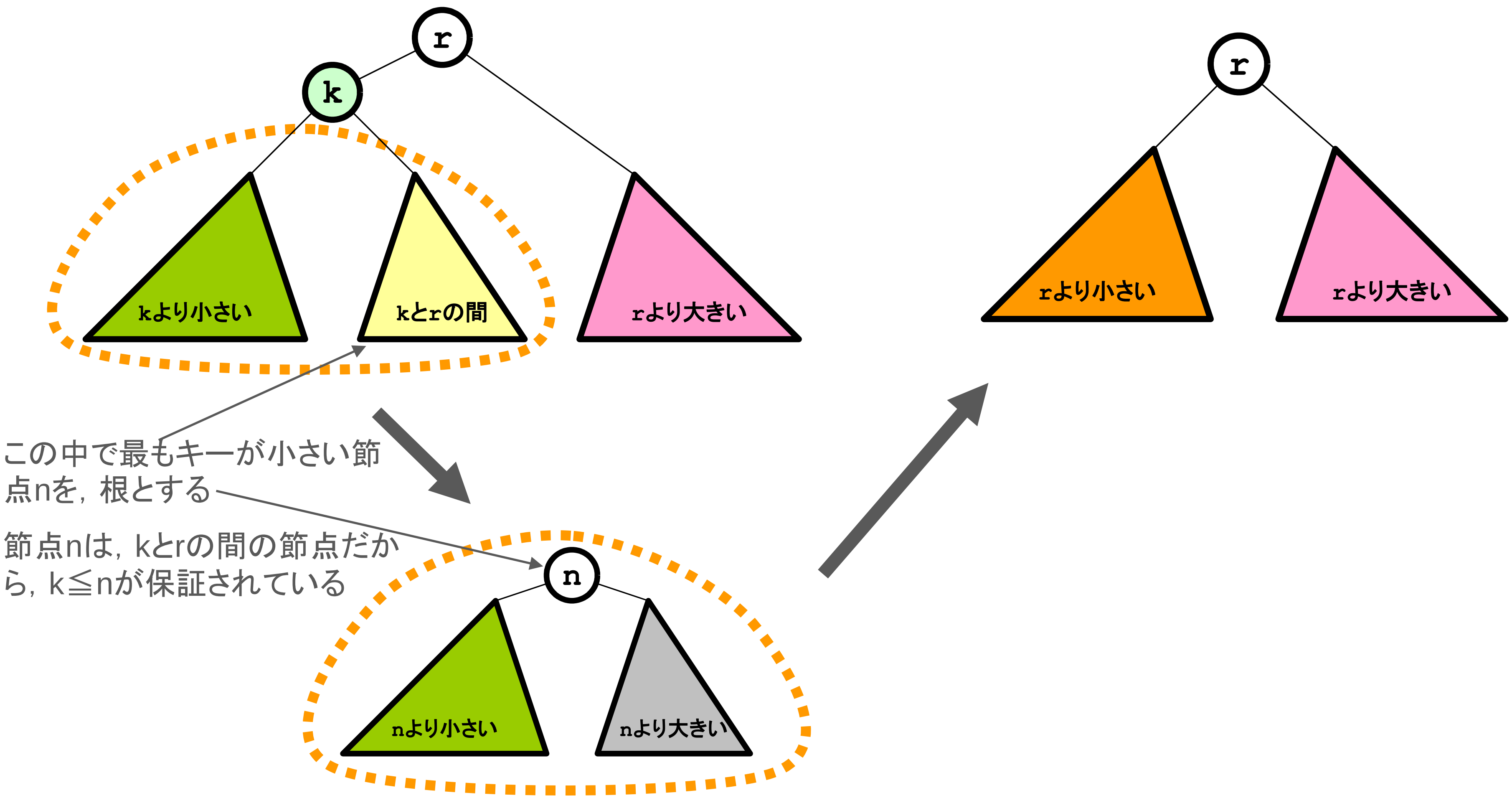
ちょうど，k番目だったら，
自分自身を返す

削除

- 与えられたキーを持つ節点を削除するには
  - まず，どの部分木にその節点が含まれているかを調べる
  - 削除される節点を根とする部分木をうまく結合して，もとの木に接続する



削除 - どのように結合するか

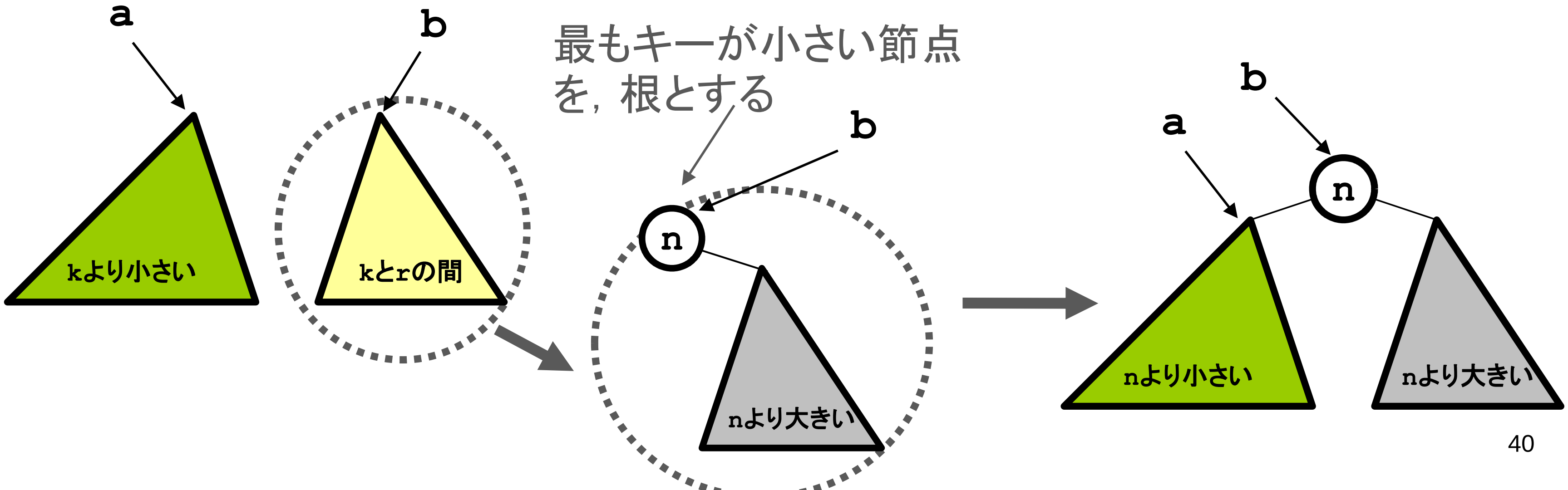


削除

```
link joinLR(link a, link b)
  {
    if (b == z) return a;
    b = partR(b, 0); b->l = a;
    return b;
  }
```

右部分木の0番目の節点を根に持って来て

その左部分木をaとする



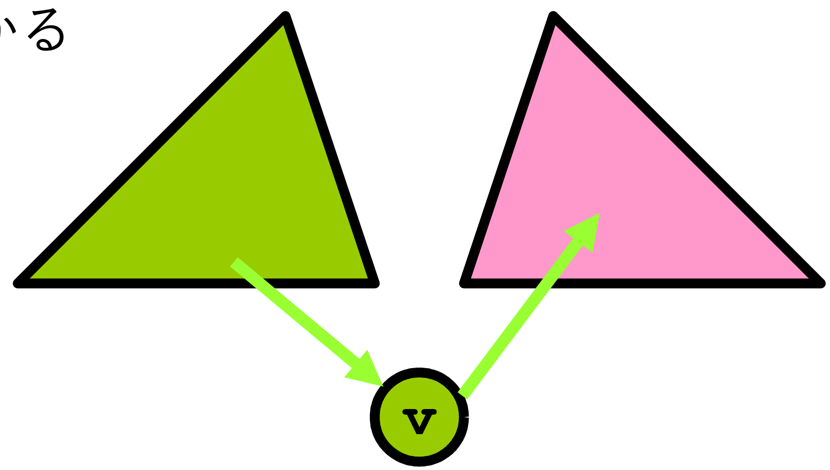
削除

```
link deleteR(link h, Key v)
  { link x; Key t = key(h->item);
    if (h == z) return z;
    if (less(v, t)) h->l = deleteR(h->l, v);
    if (less(t, v)) h->r = deleteR(h->r, v);
    if (eq(v, t))
      { x = h; h = joinLR(h->l, h->r); free(x); }
    return h;
  }
void STdelete(Key v)
  { head = deleteR(head, v); }
```

削除する節点が見つかったら，joinLRを呼ぶ

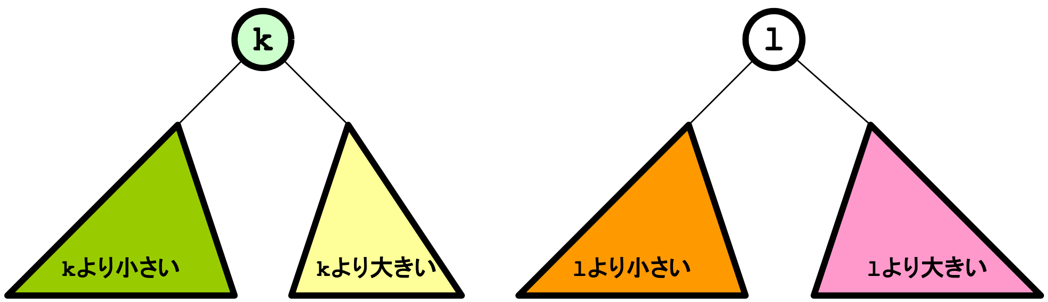
結合

- 2つの2分探索木を結合することを考えよう
  - 簡単な方法
    - 1つの2分探索木の節点を1つずつ探索し，もう1つの2分探索木に1つずつ節点を挿入していけばよい
      - 1回の挿入につき，最悪で2分探索木全体の大きさの時間がかかる

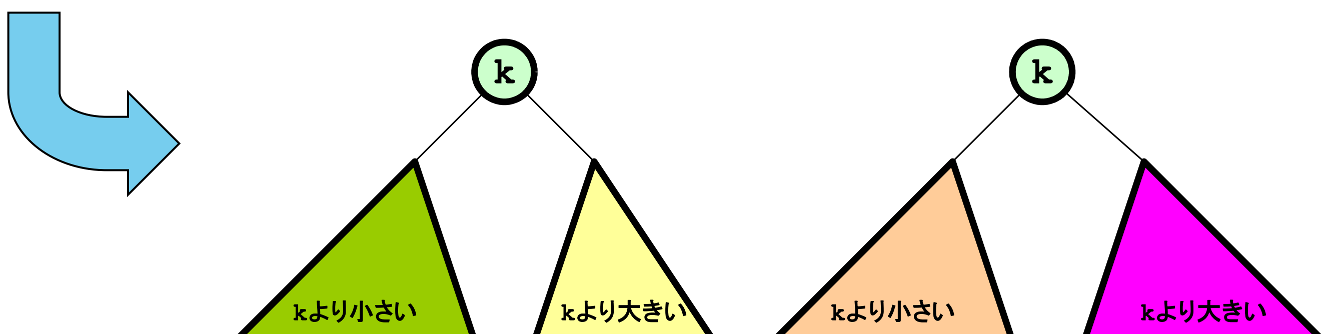


結合

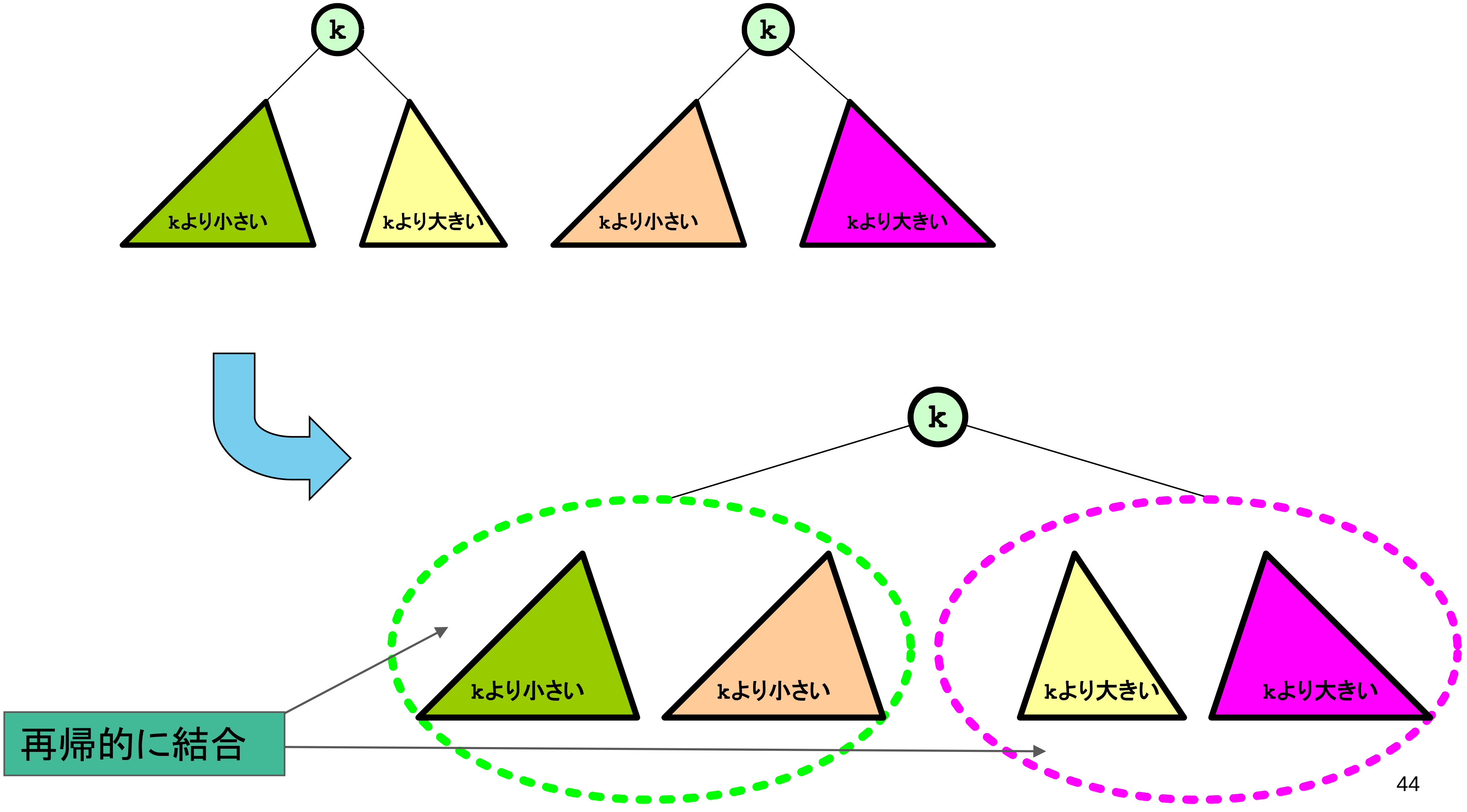
- さらに良い方法



節点kを，もう一方の木に挿入．このときの挿入は，根になるように挿入



結合


k
k
kより小さい
kより大きい
kより小さい
kより大きい
k
kより小さい
kより小さい
kより大きい
kより大きい
再帰的に結合

結合

```
link STjoin(link a, link b)
  {
    if (b == z) return a;
    if (a == z) return b;
    b = STinsert(b, a->item);
    b->l = STjoin(a->l, b->l);
    b->r = STjoin(a->r, b->r);
    free(a);
    return b;
  }
```

bの根に，**a**の根の項目を持って来て

再帰的に，右の部分木どうし，左の部分木どうしを結合

まとめ

- 根への挿入
  - 右回転
  - 左回転
- 色々な操作
  - 選択
  - 削除
  - 結合

注意点

- ST_BST2.cでは，各接点を根とする部分木の内部接点の総数（以下，カウント）を計算していない．
- カウントを正しく計算するには，以下の関数を修正する必要がある．
  - InsertT
  - rotL
  - rotR

注意点

- insertTの修正には，insertRが参考になる．ただし，insertTには木の回転操作（rotR, rotL）が含まれることに注意すること．
- 回転の際には，カウントを計算しなおす必要がある．

右回転の場合: