アルゴリズムとデータ構造

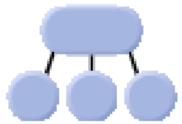
- 第１１回講義トピック：探索（続き）
  - 平衡木 (balanced tree)
    - 2-3-4木
    - 赤黒木

平衡木を使う理由

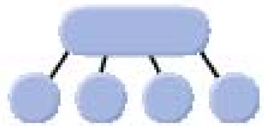
- ２分探索木における探索の効率は木のバランスに依存しる。バランスを欠いた場合（最悪の場合は連結リストに退化する）の性能は悪い。
- それを避けるため、２分探索木を作る段階で、木を「平衡化」させる必要がある。
- 平衡化技法は多数提案されているが、ここではその中の2-3-4木と赤白木について取り上げる。

2-3-4木

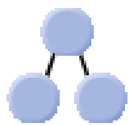
- 通常の2分木の節点は1つのキーと2つのリンクを持つ。
- 2-3-4木は木の各節点が最大3つのキーを持てるようにし、柔軟性をもたせる。
- この拡張により、以下の2種類の新しい節点を持つことが可能となる。
  - 2つのキーと3つのリンクをもつ3-節



  - 3つのキーと4つのリンクをもつ4-節



  - これに対して、通常の2分木の節点を2-節と呼ぶ。

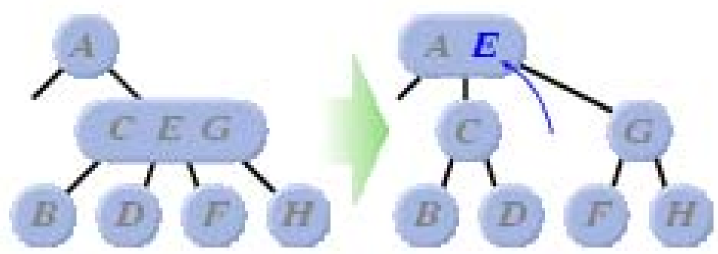


3-節と4-節の構成について

- 3-節の構成
  - 3つのリンクがそれぞれ部分木を指す。
  - 2つのキーと3つの部分木のキーとの間に以下の関係が成り立つ
    左の部分木の全てのキー　＜　左のキー　＜　中間の部分木の全てのキー　＜　右のキー　＜　右の部分木の全てのキー
- 4-節も3-節と同じように3つのキーと4つのリンクの間に大小関係が保たれる。

2-3-4木の挿入と4節の分割

- 2分木の生成と同じように挿入する際にまず探索を行い、同じキーが存在しなければ探索は外部節点まで到達し、その場所に新しいキーを挿入する。
- 挿入によって2-節と3-節は次のように変化する。
  - 2-節→3-節
  - 3-節→4-節
- 4-節の場合は節の範囲を超えるので、まず分割を行う。
  - 4-節を2つの2-節に分割し、真中のキーを親節へ渡す。
  - 分割された木に新しいキーを挿入する。



- このように挿入につれて、木が枝の方へ伸びるのではなく、逆にルートの方へ節が上がっていくので、全体としてはバランスを保つことができる。

トップダウン2-3-4木

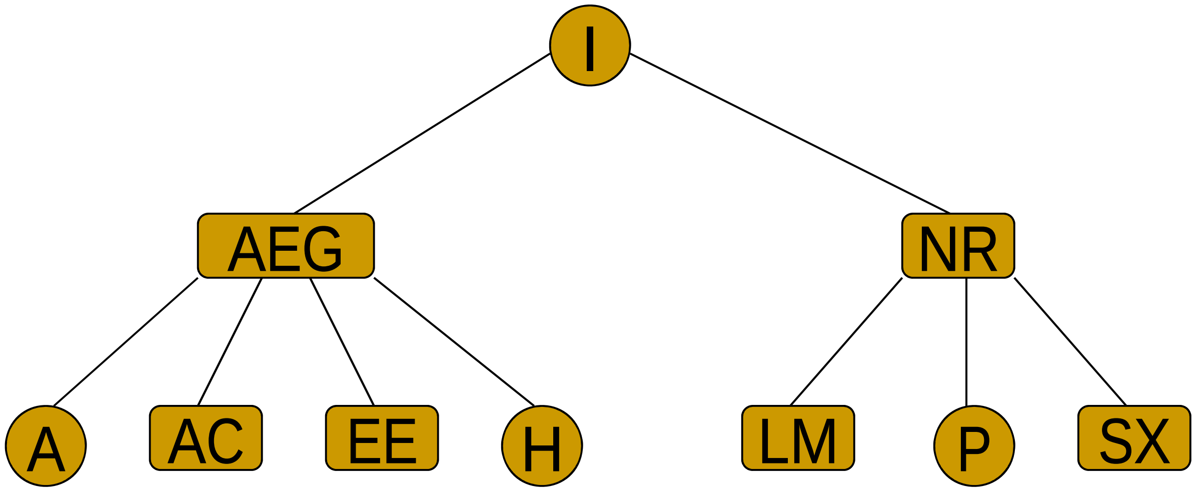
- 挿入時に4-節を分割し、中間キーを親へ渡す方法は親も4-節の場合、親の分割が必要となるので、処理が複雑となる。
- トップダウン2-3-4木は以下の新しい分割方法をとる。
  - キーを下に向かって探索していく途中で4-節に出会った場合はすぐに分割し、中間キーを親へ渡す。そうすることで、親節の分割の必要をなくす。
- 上から下へ向かっていく途中で4-節が分割されるので、<u>トップダウン2-3-4木</u>と呼ぶ。

2-3-4木の生成例

- 以下の文字列に対する2-3-4木の生成過程

  A S E A R C H I N G E X A M P L E

生成された2-3-4木



2-3-4木における探索と挿入の計算量

- 2-3-4木は常にバランスがとれる状態を保つので、木の高さが抑えられ、探索を速く行うことができる。
- 探索
  - lgN+1個以下の節点を訪問する。
- 分割
  - 最悪の場合でも、lgN+1回より少ない
  - 平均1回未満

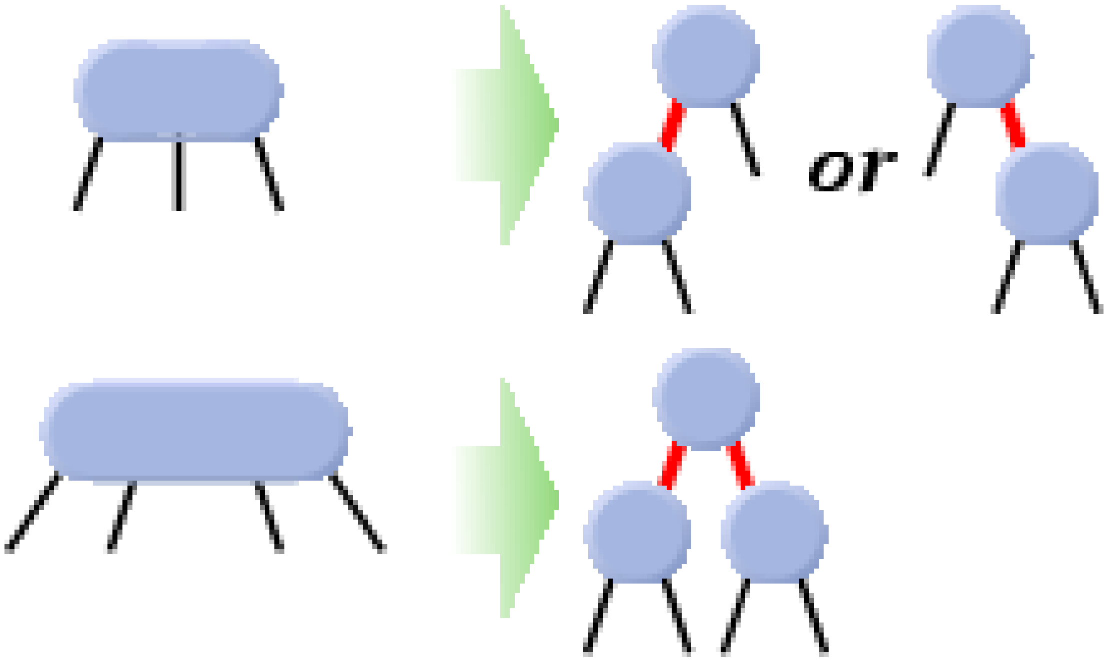
2-3-4木の問題点

- 節点のキーの数が増えるので、データ構造が複雑で、アルゴリズムの実装も難しくなる。
- バランスの面以外では、2分探索木よりも実行速度が遅くなる可能性がある。

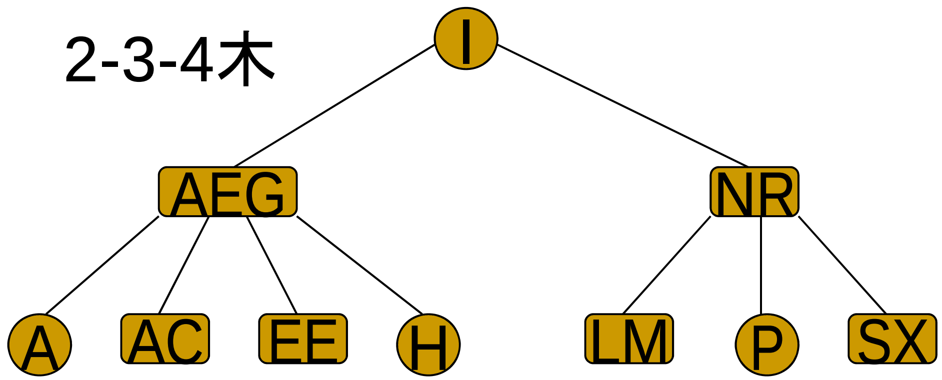
赤黒木 (red-black tree: RBT)

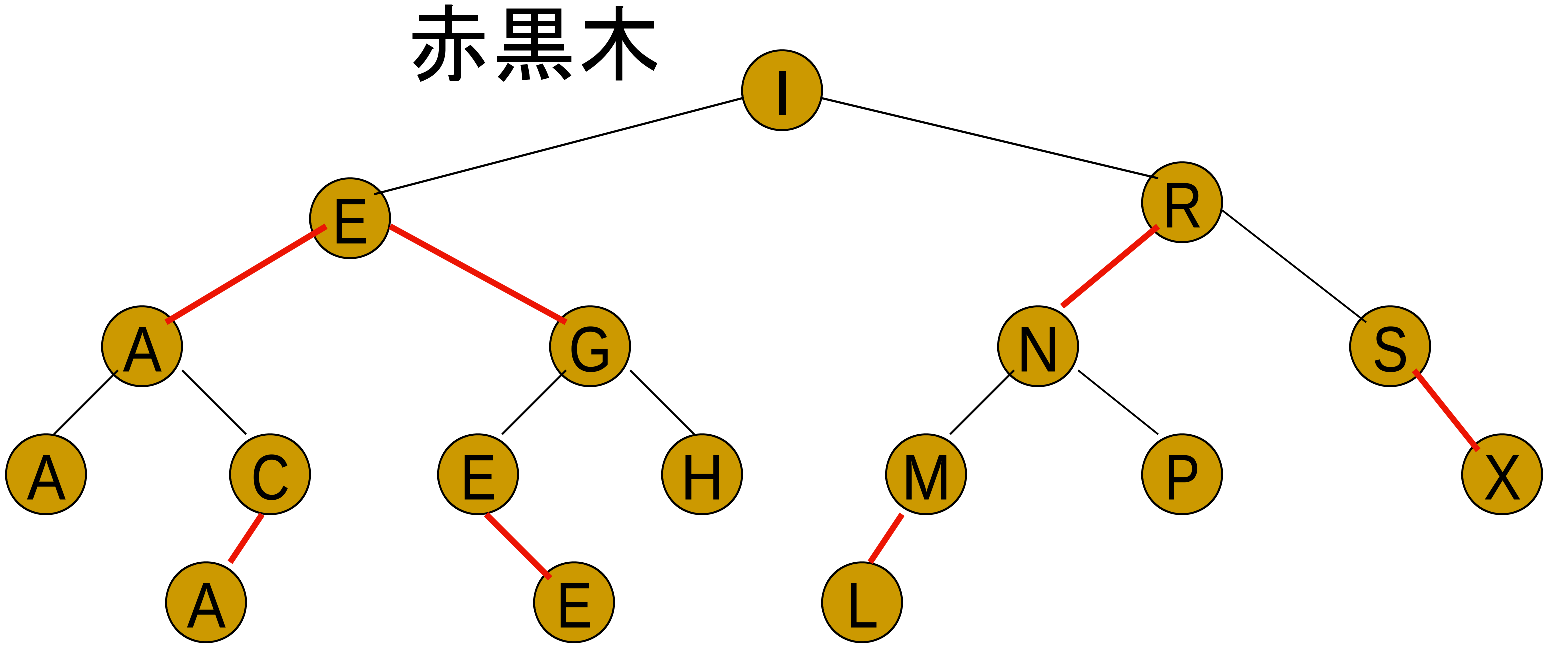
- 2分木の構造を保ちながら、2-3-4木の特徴を実現できれば、2分木探索のアルゴリズムをそのまま利用でき、効率のよいバランスの取れた木が構築できる。
- 赤黒木は2分木におけるリンクを赤と黒の2つに分ける。
  - 3-節と4-節を「赤い」リンクで結合された2つの小さな2分木として表現する。
  - 「黒い」リンクは2-3-4木の節同士を結合する。
- 赤と黒のリンクを区別するために各節点に1ビットのフラグを追加し、通常の2分木（2-節のみ）で2-3-4木を表現する。

3-節と4-節の赤黒リンクによる表現



赤黒木の例





赤黒木のデータ構造

```
struct node {
  int key, info, red;
  struct node *l, *r;
}
```

ここで、変数redは親からのリンクの色が赤か黒かを区別する。

- red = 1: red
- red = 0: black

赤黒木の特徴

- １つの2-3-4木に対応する赤黒木はいくつもある（3-節の非対称性によるもの）
＊要素の挿入により回転が必要となる場合がある。
- ルートから各外部節点へのパスにおいて、赤リンクが２つ続くことはない。
- ルートから各外部節点へのパスはどれも同じ個数の黒リンクをもつ。
- 通常の２分探索木のプログラム (treesearchなど)が修正なしでそのまま赤黒木にも使える。
＊要素挿入の場合は4節の分割と3節の回転プログラムが必要となる。

赤黒木の初期化プログラム

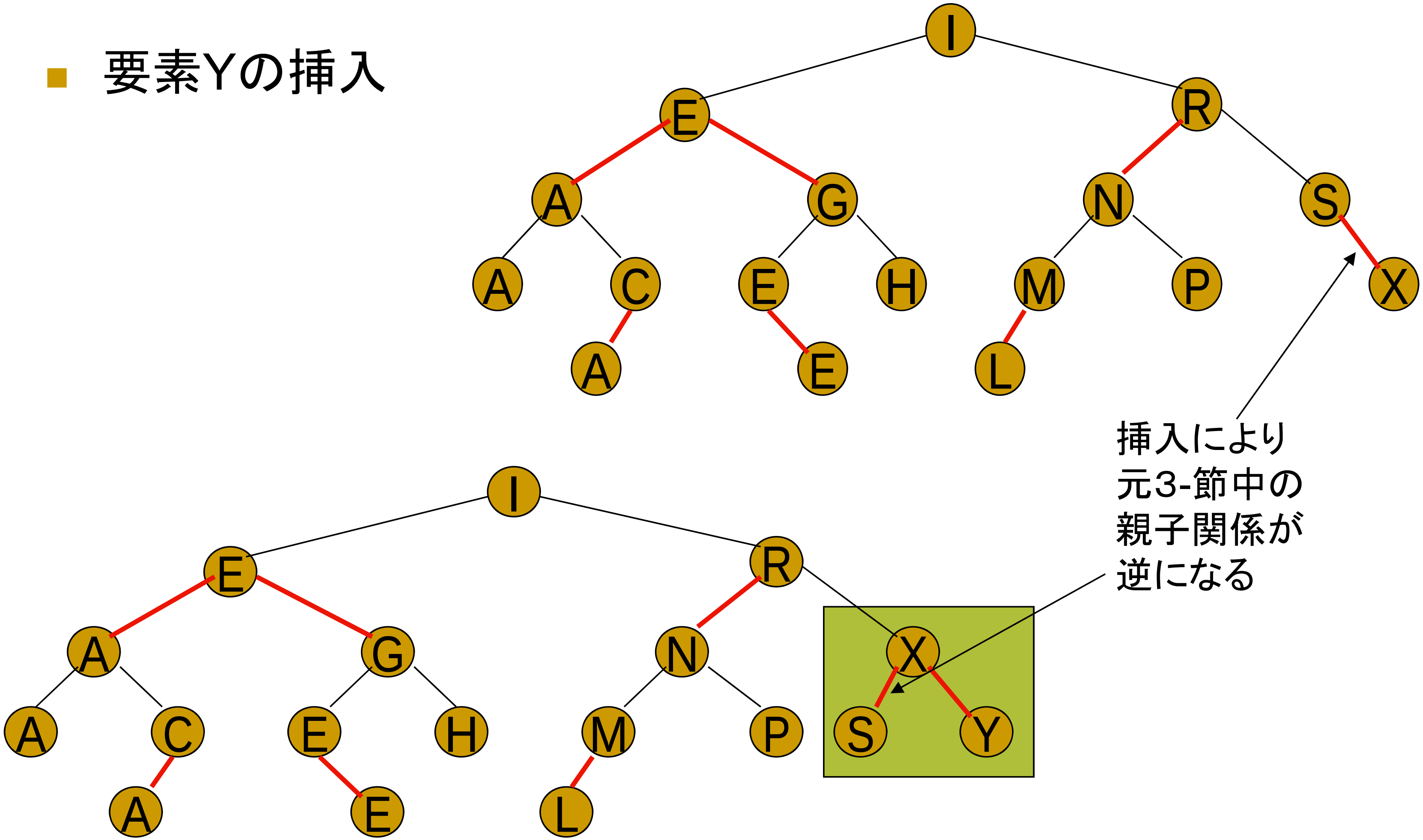
```
struct node {
  int key, info, red;
  struct node *l, *r;
};
struct node *head, *z, *gg, *g, *p, *x;
rbtreeinitialize() {
  z = (struct node *) malloc(sizeof *z);
  z->l = z; z->r = z; z->red = 0; z->info = -1;
  head = (struct node *) malloc(sizeof *head);
  head->r = z; head->key = 0; head->red = 0;
}
```

赤黒木における要素の挿入プログラム

```
struct node *head, *z, *gg, *g, *p, *x;
// p: parent, g: grandparent, gg: great-grandparent
rbtreeinsert(int v, int info) {
  x = head; p = head; g = head;
  while (x!=z) {
    gg = g; g = p; p = x;
    x = (v < x->key) ? x->l : x->r;
    if (x->l->red && x->r->red)
      split(v); //4-節を分割する。必要に応じて回転も行う。
  }
  x = (struct node *) malloc(sizeof *x);
  x->key=v; x->info=info; x->l=z; x->r=z; //新しい節を生成
  if (v < p->key) p->l=x; else p->r=x; //新しい節を加える
  split(v); //xとダミー節点からなる仮想4節の分割を行う。
}
```

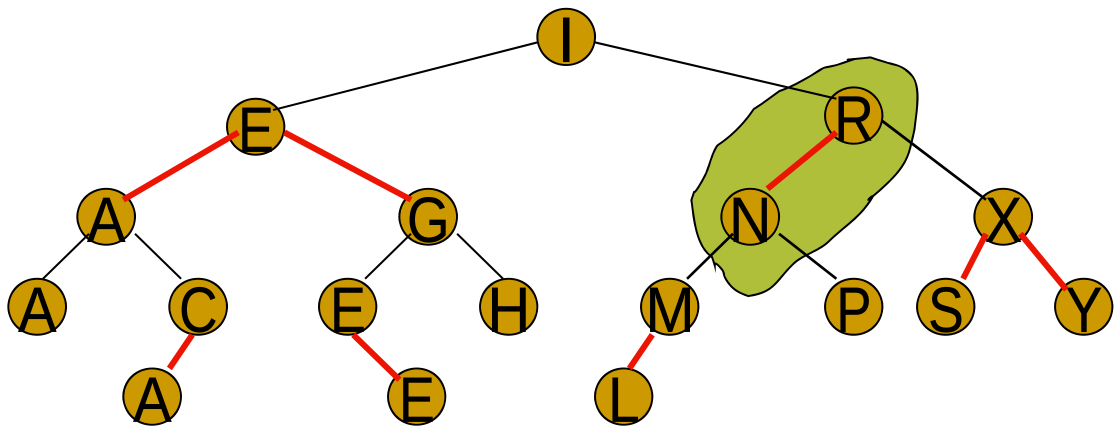
赤黒木への挿入例

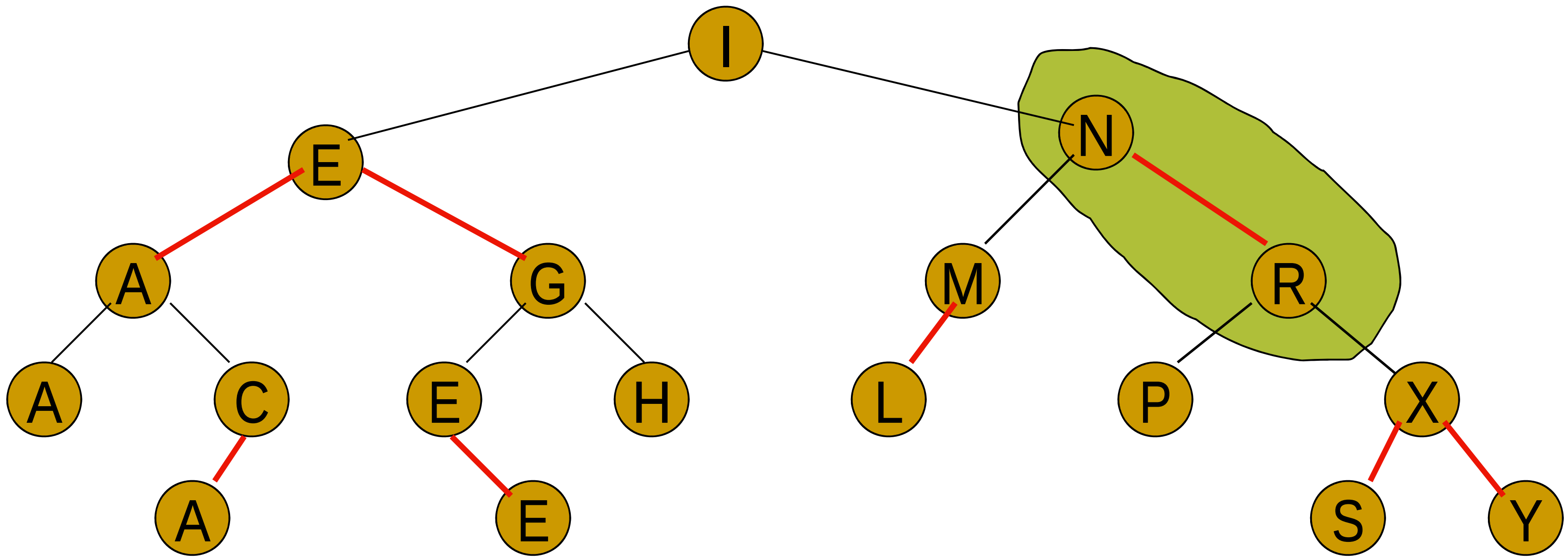
- 要素Yの挿入



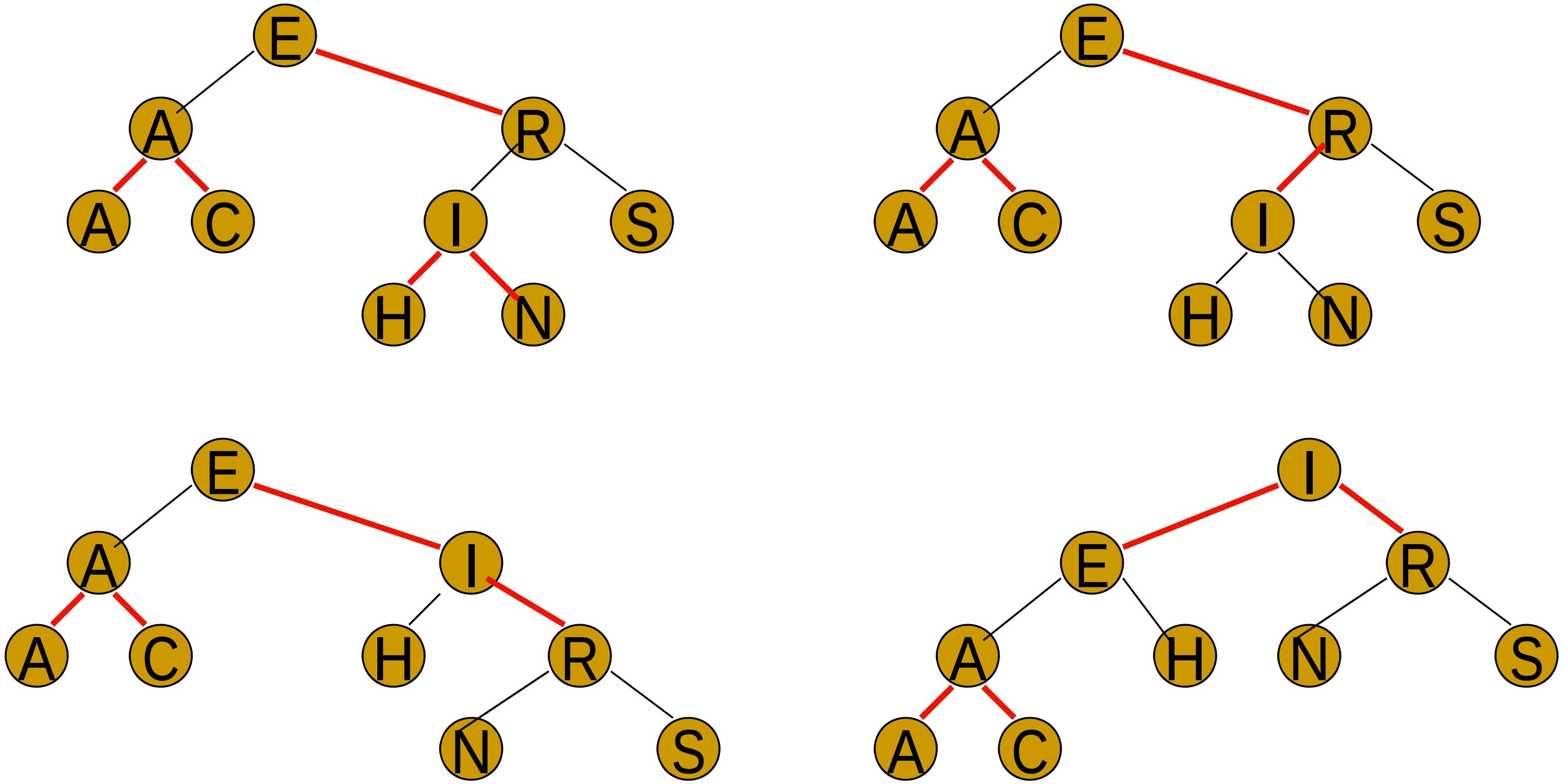
3節の回転

- もっと複雑な場合の3-節の回転
＊3-節からの3つのリンクの位置に注意





節点分割と2重回転

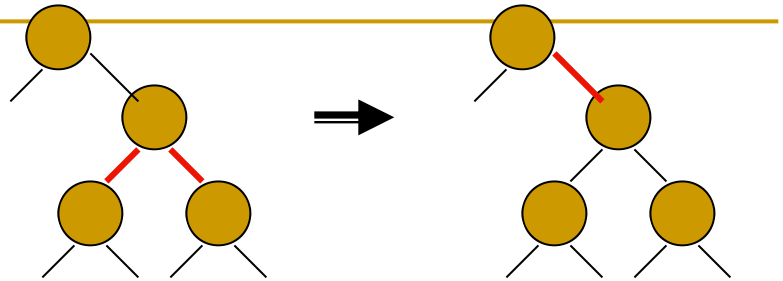


次にGが挿入される場合

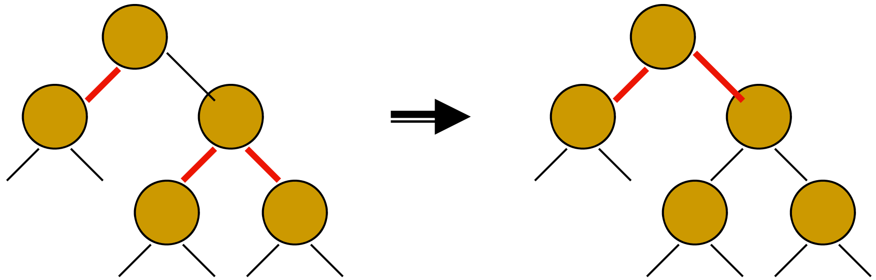
節点の分割と回転

- 4-節の分割

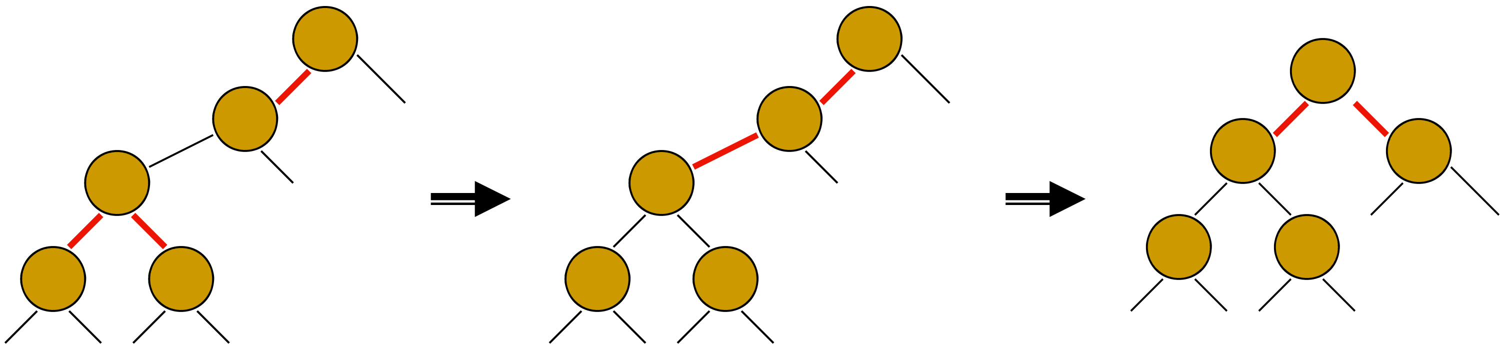
親が2-節



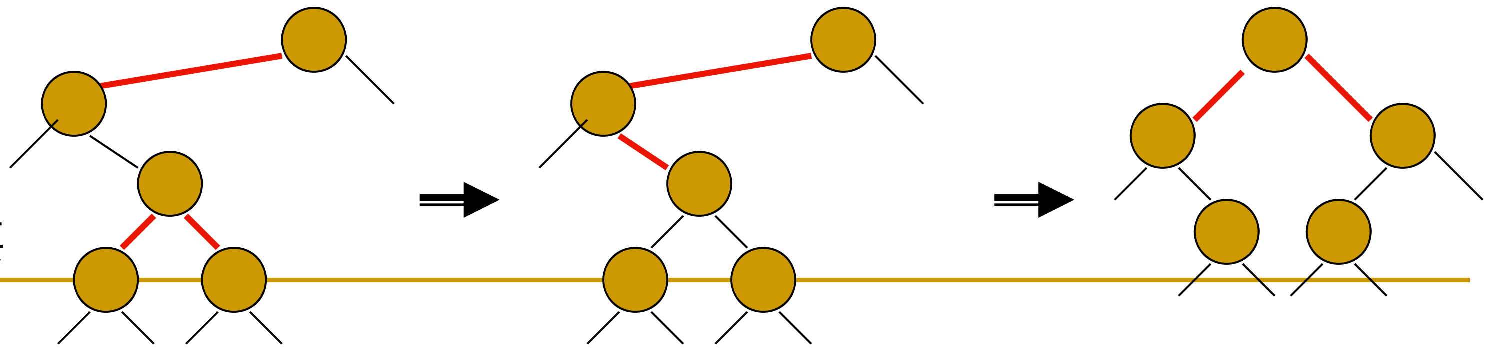
親が3-節
回転なし



親が3-節
回転必要



親が3-節
２重回転必要



節点の分割プログラム

```
split(int v) {
  // まずは赤リンクを上に上げる
  x->red = 1; x->l->red = 0; x->r->red = 0;
  if (p->red) { //親も赤の場合の回転処理
    g->red = 1;
    if (x < g->key != v < p->key)
      p = rotate(v, g); //一回目の回転
    x = rotate(v, gg); //二重回転
    x->red = 0;
  }
  head->r->red = 0; //headからのリンクが常に黒にする
}
```

節点の回転プログラム

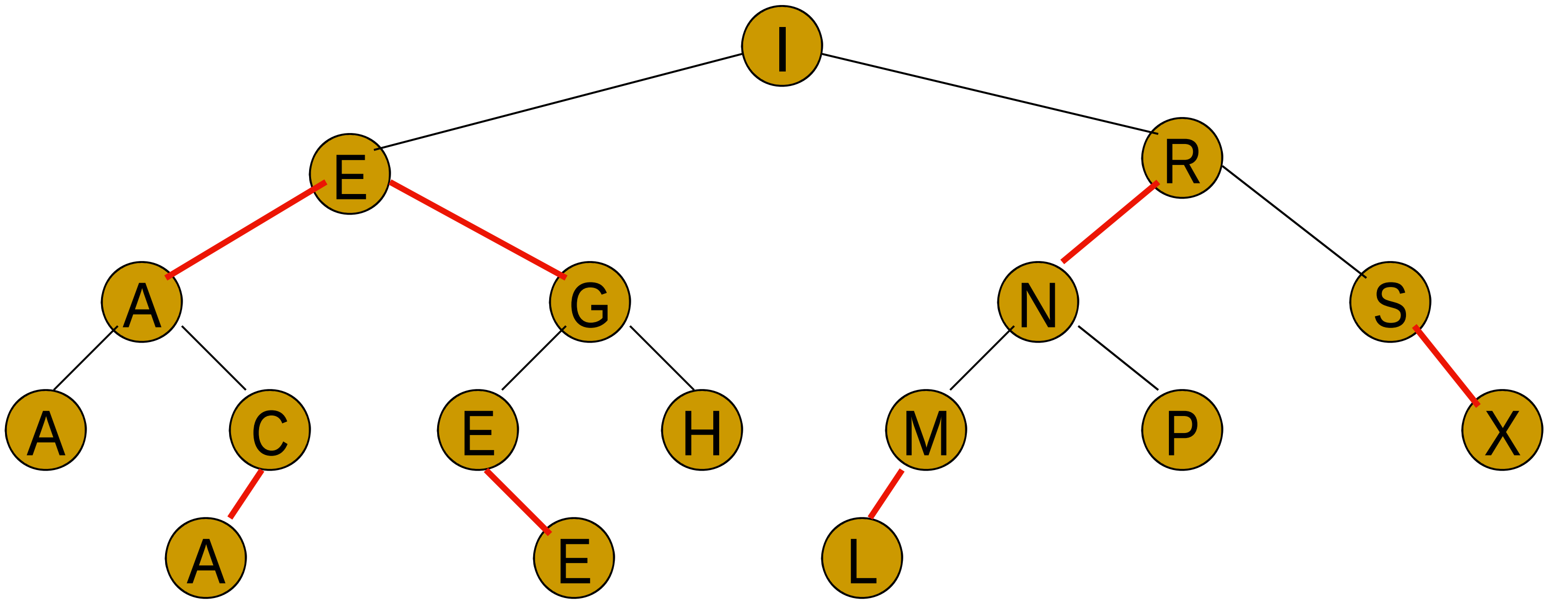
```
struct node *rotate(int v, struct node *y) {
  struct node *c, *gc;
  c = (v < y->key) ? y->l : y->r;
  // c(親)-gc(子) の枝を gc(親)-c(子) の向きに変える
  if (v < c->key) {
    gc = c->l; c->l = gc->r; gc->r = c;
  }
  else {
    gc = c->r; c->r = gc->l; gc->l = c;
  }
  // yはcの代わりにgcへリンクする
  if (v < y->key) y->l = gc; else y->r = gc;
  return gc;
}
```

赤黒木の生成例

- 以下の文字列に対する赤黒木の生成過程

  A S E A R C H I N G E X A M P L E

生成された赤黒木



赤黒木の計算量

- N個のランダムなキーから構成された赤黒木の探索は約lgN回の比較が必要となる。挿入時に必要となる回転の平均回数は1回以下である。
- 一般的にN節点の赤黒木はワーストケースでは、探索は約2lgN+2回以下の比較を必要とし、挿入で必要となる回転の回数は比較回数の１／４以下となる。