概要

探索

- 逐次探索
- 2分探索
- 探索とデータ管理
  - 2分探索木
  - 平衡木 (Balanced tree)

ここ

探索木と効率

2分探索木

探索、挿入、削除の効率は木の形状に依存する

平衡木

最悪の場合が生じないように木を構成する

バランスのとれた木構造

- 平衡2分木 (AVL木)
- 2-3木
- 2-3-4木 （トップダウン2-3-4木)
- Red-Black 木(2色木、赤黒木)

＊AVL木 : Adel'son-VelskiiとLandis が提案した木

平衡2分木、 2-3 木

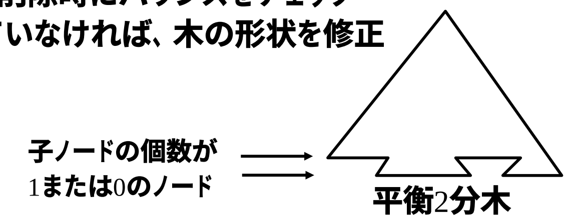
平衡2分木 (AVL 木)

バランスのほぼとれた2分木

- 子の個数が1以下のノードのレベルの差が1以下

レコードの挿入、削除時にバランスをチェック

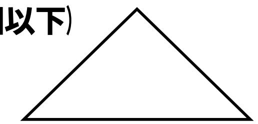
バランスがとれていなければ、木の形状を修正



2-3 木

子ノードの個数が3個以下 (2分木は2個以下)

バランスのとれた木



2-3-4木、Red-Black木

2-3-4木

子ノードの個数が4個以下

バランスがとれている (後述)

Red-Black木 (赤黒木)

2-3-4木を2分木で実現した木

2分探索木のようなシンプルなやり方で探索可能

2-3-4木(1)

2分探索木

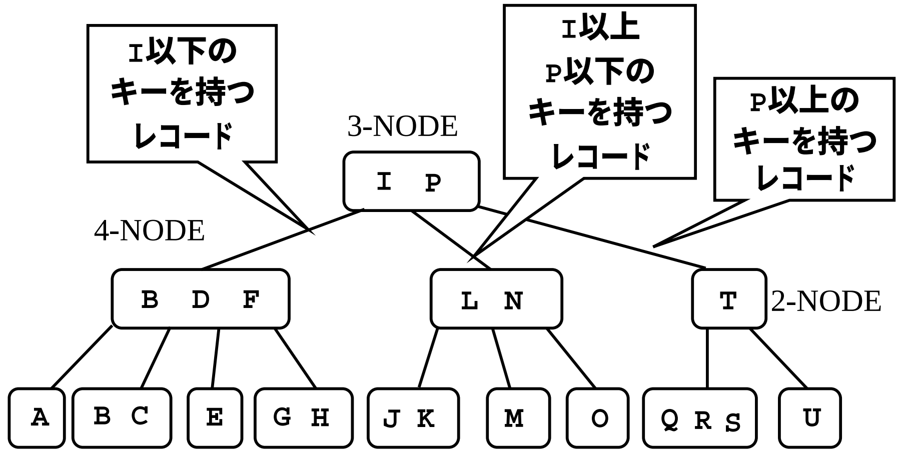
木の形状は、挿入、削除するレコードの順序に依存する
レコードの順序によっては、バランスの悪い木になり、探索、挿入、削除の効率が悪くなる

2-3-4**木**

各ノードが持つことのできるキーの個数に融通がきく

- 2-NODE　**キーを**1**つ持つ、子ノードは**2**個または**0**個**
- 3-NODE　**キーを**2**つ持つ、子ノードは**3**個または**0**個**
- 4-NODE　**キーを**3**つ持つ、子ノードは**4**個または**0**個**

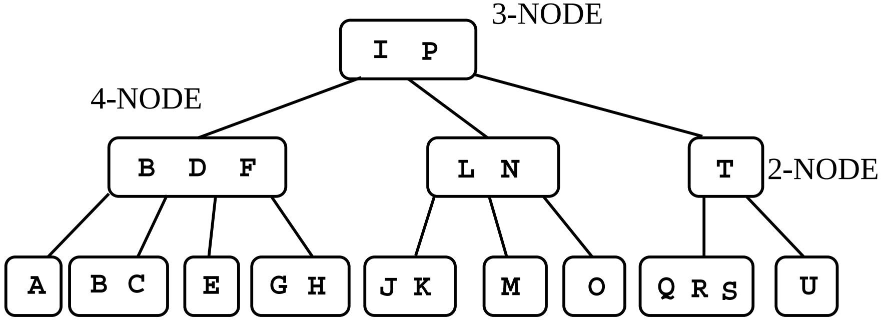
2-3-4木(2)



2-3-4木の特徴

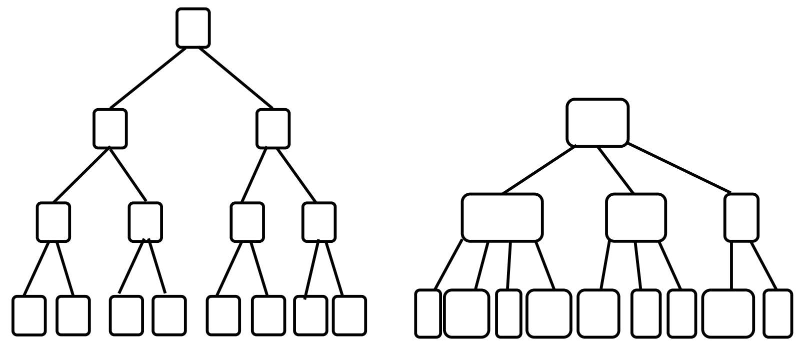
すべての葉のレベルが等しい

レコードの挿入、削除時にはすべての葉のレベルが等しくなるよう調整する



木の高さ

2-3-4**木の高さは**　$\log n$　**以下**　(n : **レコード数**)



すべてのノードが2-NODE**なら高さは** $\lfloor \log n \rfloor$

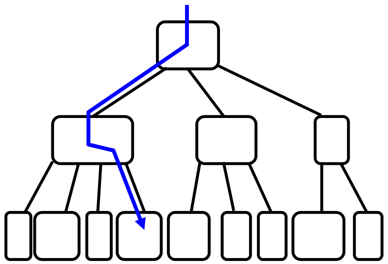
3-NODE, 4-NODE**があれば、高さは** $\lfloor \log n \rfloor$ **以下**

トップダウン 2-3-4 木 : 探索

探索 (基本方針)

2分探索木と同様の操作

計算量　O(木の高さ　) = $O(\log\ n)$



＊挿入時の効率のために、後で少し変更

トップダウン2-3-4木 : 挿入 (1)

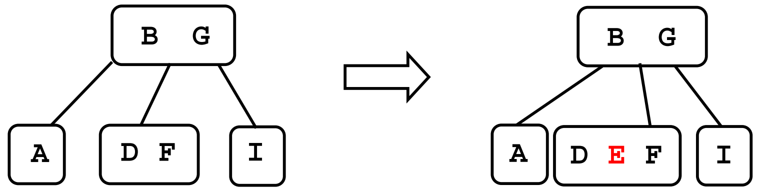
キーvのレコードの挿入 (基本方針)

vを挿入すべき場所を探索 (葉に到達するまで探索)

葉が持つキーの個数に余裕があれば、そこに挿入

＊1ノードにキーを3個まで挿入可能

Eを挿入



トップダウン2-3-4木 : 挿入 (2)

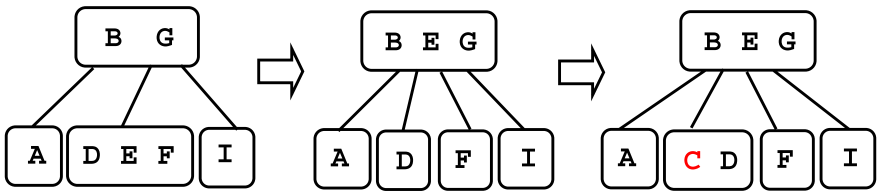
キーvのレコードの挿入 (基本方針)

vを挿入すべき場所を探索

探索した葉が4-NODEならば、

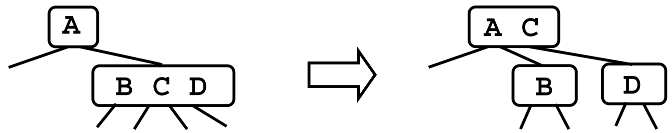
4-NODEを2つの2-NODEに分割してからvを挿入

Cを挿入

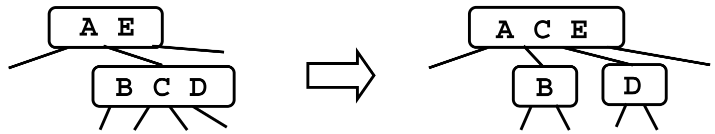


4-NODEの分割

分割するノードの親が2-NODEの場合



分割するノードの親が3-NODEの場合



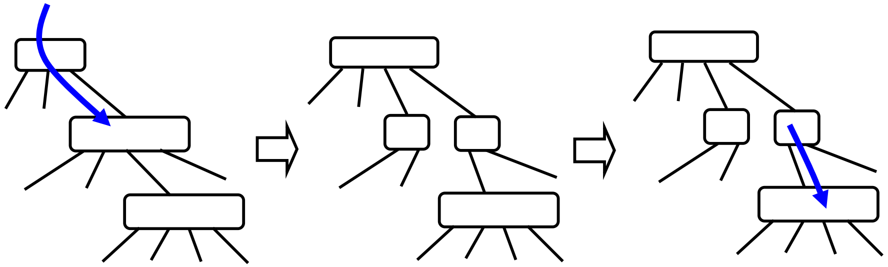
分割するノードの親が4-NODEの場合

この場合が生じないよう工夫する

→ トップダウン2-3-4木

4-NODEの連続を回避

探索、挿入時に根から葉に向けて順にノードを訪問
このとき、4-NODEに出会えば、2-NODEに分割する

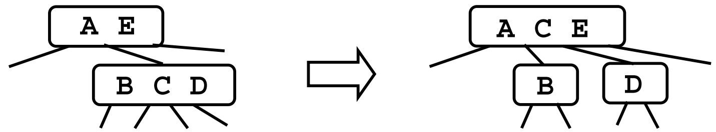


4-NODEを訪問　　2-NODEに分割　　4-NODEを訪問
親は4-NODEでない

探索は根 (トップ) から下向きに行うのでトップダウン2-3-4木

4-NODEの分割

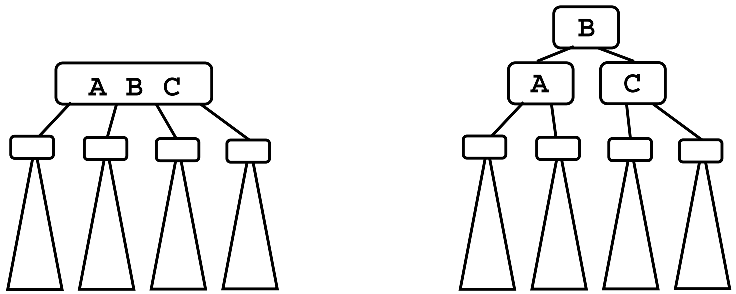
4-NODE (子ノードが4個) を
2つの 2-NODE (子ノードが2個) に分割するので
分割は容易 (子ノードの総数が同じ)



親ノードが3-NODEのとき、
分割によって、親ノードが 4-NODE になる
次の探索時に分割されるので、
とりあえず4-NODEのままにしておく
＊4-NODEになった親が根のときは、例外的に分割しておく

根の分割と木の高さ

根が4-NODEになれば2つの2-NODEに分割

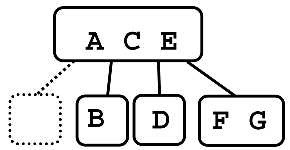


木の高さが変わるのは、根の分割のときだけ
すべての葉のレベルが 1 上がる
→分割後も、すべての葉のレベルは等しい

トップダウン2-3-4木 : 削除 (1)

探索、挿入ではすべての葉のレベルを等しく保つ

削除時は、
2-NODEの葉を削除することがある
2-3-4木でなくなるので、
形状を調整する

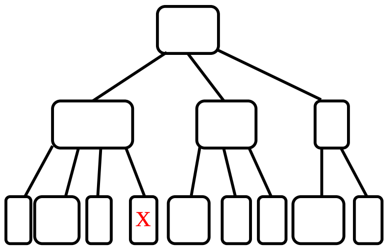


キーが3つの4-NODEは、
子が4個または0個
でなければならない

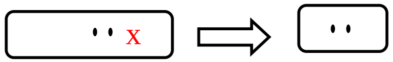
トップダウン2-3-4木 : 削除 (2)

キー x の削除

1. xが葉ノードにあるとき



3-NODE または 4-NODEのとき
葉から x を削除



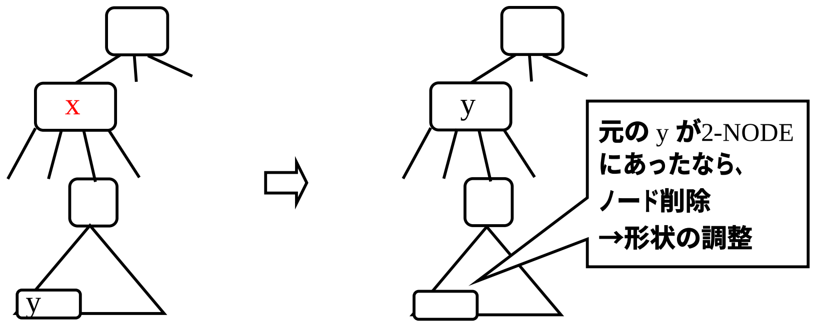
2-NODEのとき
ノードを削除→形状の調整

調整

トップダウン2-3-4木 : 削除 (3)

キー x の削除

2. xが内部ノードにあるとき



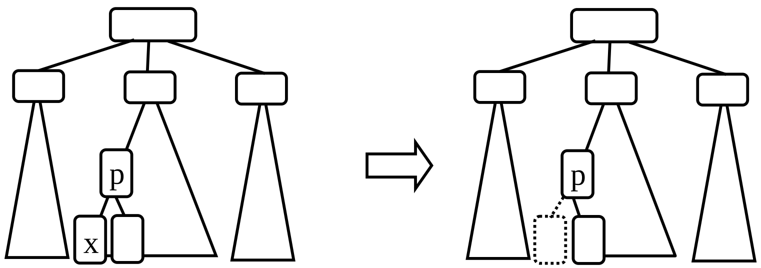
x を含むノードを根とする部分木で
xの次のキーを y とする

y を x の場所へコピーして、
元のy を削除すると考える

トップダウン2-3-4木 : 削除 (4)

形状の調整が必要なとき

- ノードの削除が行われるときだけ



削除前　　　削除後

トップダウン2-3-4木 : 削除 (5)

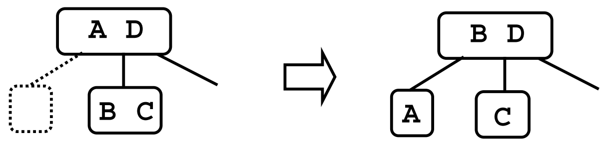
調整の方針

なるべく局所的 (ローカル) に調整する
だめなら、根の方向に調整箇所を移動

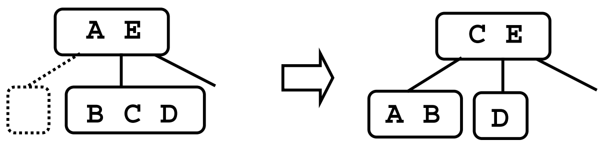
1. 隣の兄弟が余分にキーを持っている
   1. 隣の兄弟が3-NODE
   2. 隣の兄弟が4-NODE
2. 親が余分にキーを持っている
   1. 親が3-NODE
   2. 親が4-NODE
3. 親が2-NODE、かつ隣の兄弟も2-NODE
   根の方向に調整箇所を移動

トップダウン2-3-4木 : 削除 (6)

1. 隣の兄弟が余分にキーを持っている
   1. 隣の兄弟が3-NODE

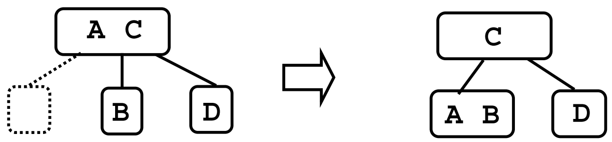


   2. 隣の兄弟が4-NODE

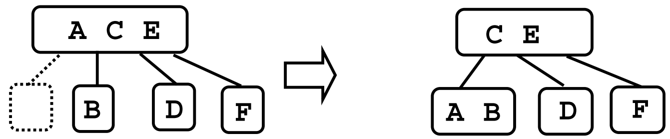


トップダウン2-3-4木 : 削除 (7)

2. 親が余分にキーを持っている
   1. 親が3-NODE



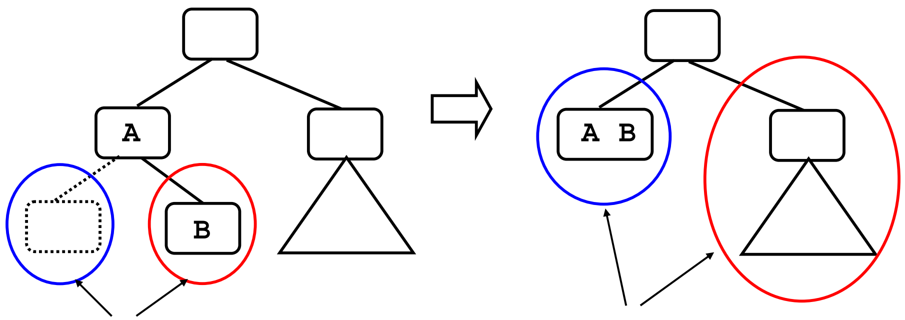
   2. 親が4-NODE



トップダウン2-3-4木 : 削除 (8)

3. 親が2-NODE、かつ隣の兄弟も2-NODE

   根の方向に調整箇所を移動



高さが異なっている
調整必要

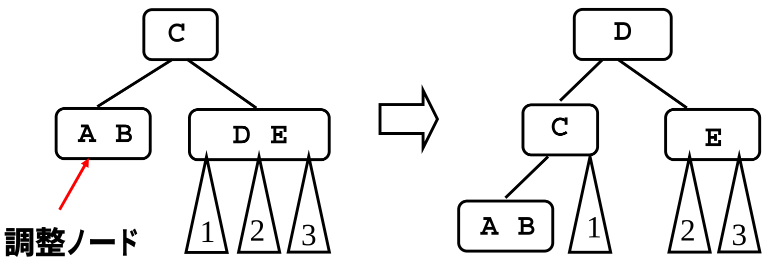
高さが異なっている
調整必要

トップダウン2-3-4木 : 削除 (9)

3. 親が2-NODE、かつ隣の兄弟も2-NODE

   調整ノードの隣の兄弟、親が余分にキーを持っていれば、調整可能

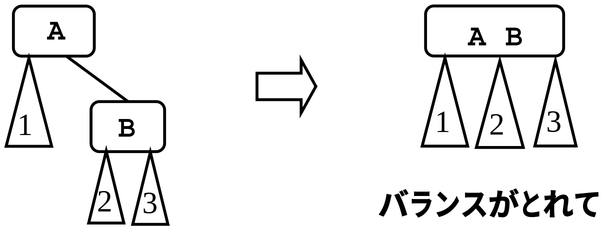
   (1, 2 と同様の調整を行う)



隣の兄弟で調整する場合

トップダウン2-3-4木 : 削除 (10)

3. 調整箇所がこれ以上移動できないとき



バランスがとれているので、
調整完了している
→結果的に、木の高さが
1つ減少している

2-3-4木 : 操作の効率

探索

根から葉に向かう1つの経路上のノードだけ訪れる
各ノードでは、探索キーとノードキーの比較、
(必要なら)ノードの分割を行う
→ 計算量 O(log n)

挿入

根から葉に向かって、挿入箇所を探索する
計算量は探索と等しい → 計算量 O(log n)

削除

削除レコードの次の要素の探索(葉への経路上)、
調整箇所の根方向への移動 → 計算量 O(log n)

Red-Black木 (赤黒木)(1)

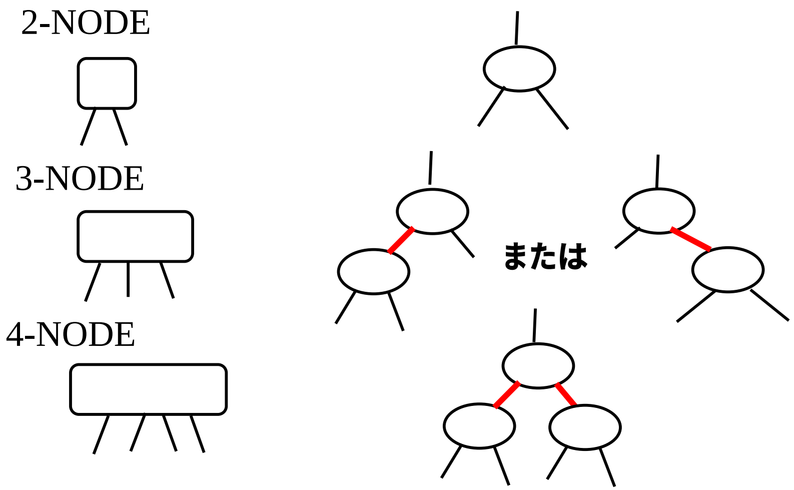
2-3-4木は、3種類のノードを用いる
よって、探索アルゴリズムは2分探索木より
複雑になる。

赤黒木

2分木を用いて、2-3-4木を実現する
アイディア
辺に赤または黒の色を割り当てる
赤い辺は、両端のノードが2-3-4木の同じノード
であることを示す

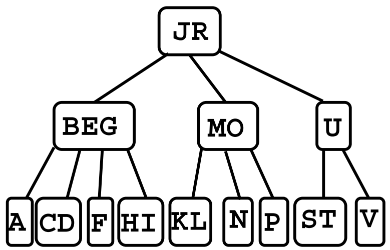
赤黒木(2)

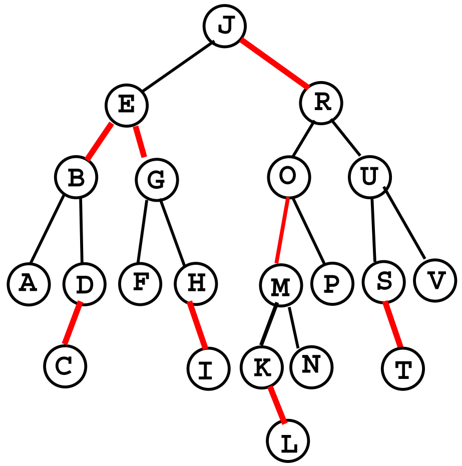
2-3-4木のノード　　赤黒木のノード



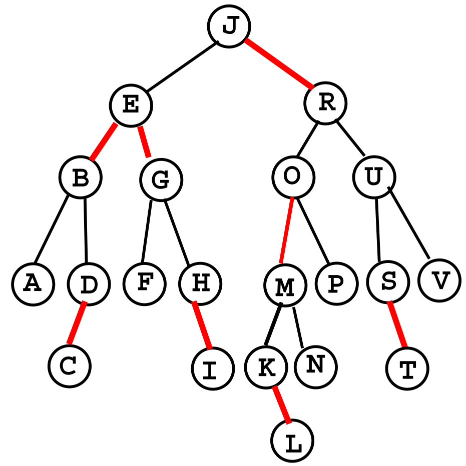
赤黒木(3)

2-3-4木　　　　赤黒木





赤黒木(4)



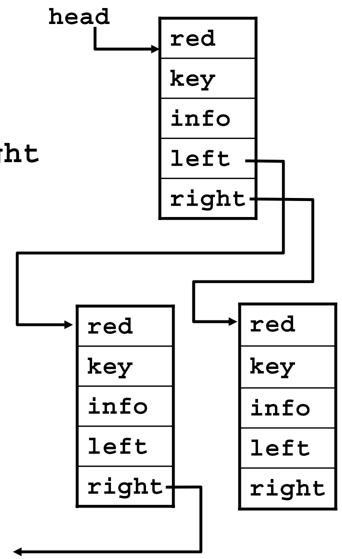
根から葉への経路で、
赤い辺は連続して表れない

根から葉への経路に現れる
黒い辺の個数は等しい
→対応する2-3-4木の高さ

赤黒木の高さO(log *n*)

C言語で実現すると

```
static struct node {
  int red;
  int key,  info;
  struct node *left, *right
}
struct node *head;
```



2分探索木のノードにredを
追加しただけ
red　は、1ビットで十分
親への辺が赤のとき　red=1
　　　　　黒のとき　red=0

赤黒木での操作

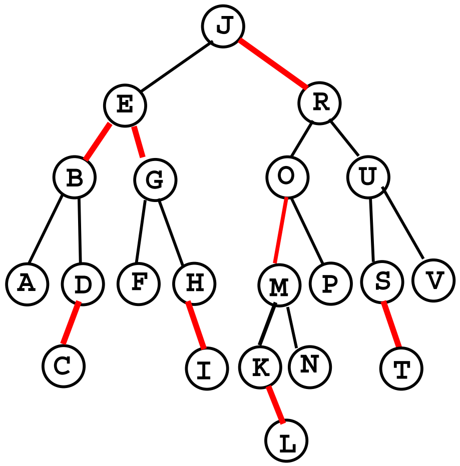
探索、挿入、削除
　キーの探索、挿入箇所の探索、
　次に小さいキーの探索など2分探索木と同様
　＊辺の色を気にしないで探索できる

赤黒木特有の操作
　4-NODEの分割、木の形状の調整
　辺の色の付け替え、回転操作を用いて行う
　＊2-3-4木では、分割、形状調整はあまり起こらない
　　よって、ほとんどの操作は2分探索木と同様
　　しかし、木のバランスは保たれる
　赤黒木特有の操作を加えても各操作の計算量は*O*(log *n*)

赤黒木：探索

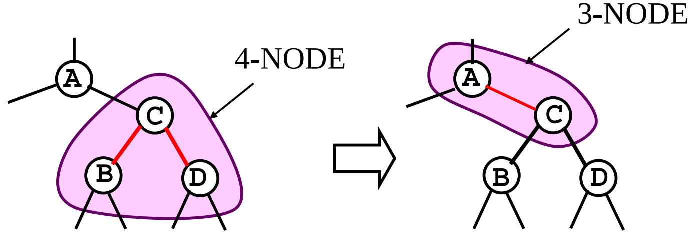
探索方法は、
分割操作を除いて
2分探索木と同じ

辺の色（赤か黒）は
意識しない



赤黒木：4-NODEの分割(1)

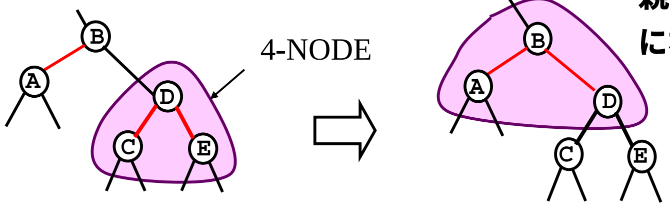
親が2-NODE



親ノードが3-NODEになる
4-NODEは2つの2-NODEになる
辺の色を付け替えるだけ

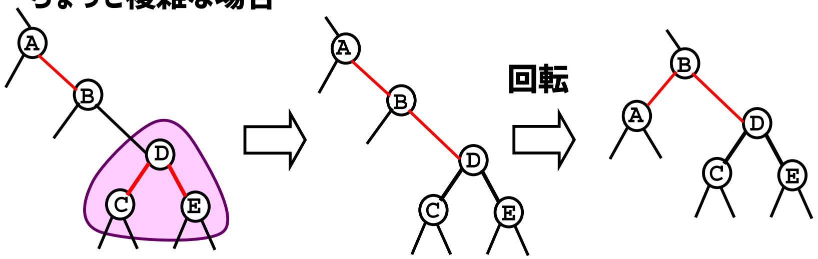
赤黒木：4-NODEの分割(2)

親が3-NODE



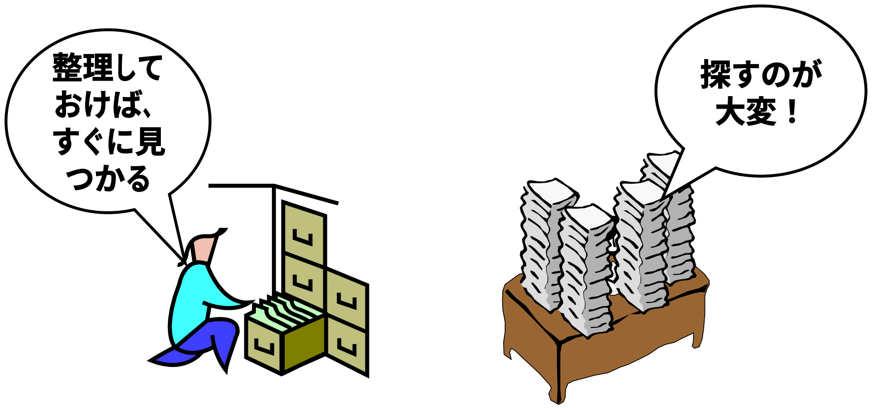
親が4-NODE
になる

ちょっと複雑な場合



探索のまとめ

探索の効率とデータの管理は密接な関係にある



赤黒木：回転操作(参考)