永続スプレー木
と
永続スプレーヒープ

2012.3.4

山本和彦

おしながき

- スプレー木
- ボトムアップなスプレー木
- トップダウンなスプレー木
- スプレーヒープ

スプレー木

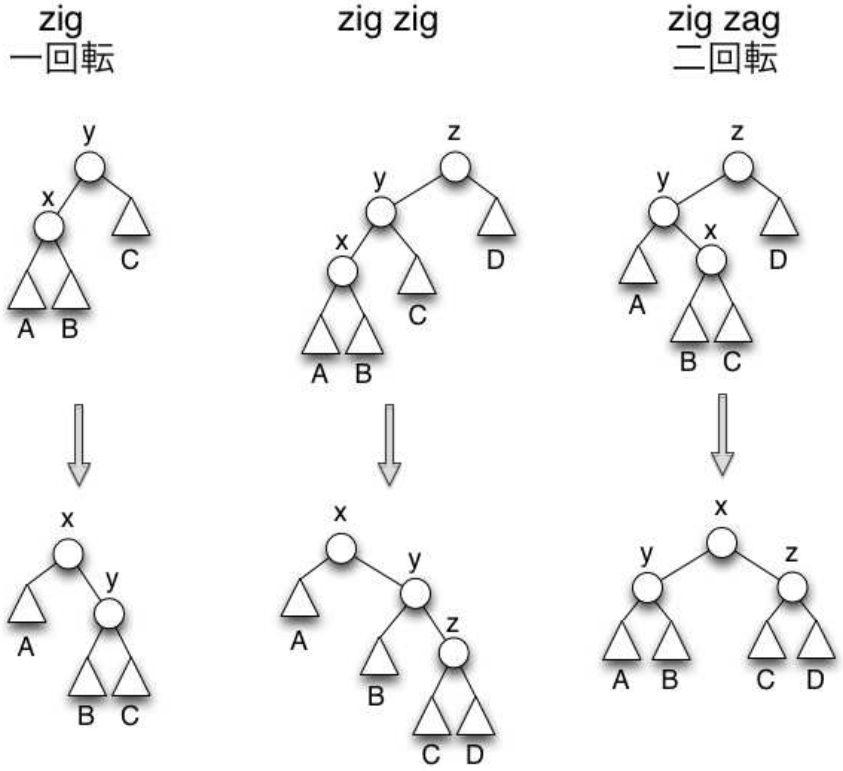
- 二分探索木
  - 平衡のための情報を持たない

```
data Splay a = Leaf | Node (Splay a) a (Splay a)
```

- 作成時
  - O(N log N)？
  - 整列済みならリスト状になるので O(N)
- 検索時も頑張る
  - アクセスした要素がルートへ
- スプレイ操作の実装には二種類ある
  - ボトムアップ
  - トップダウン

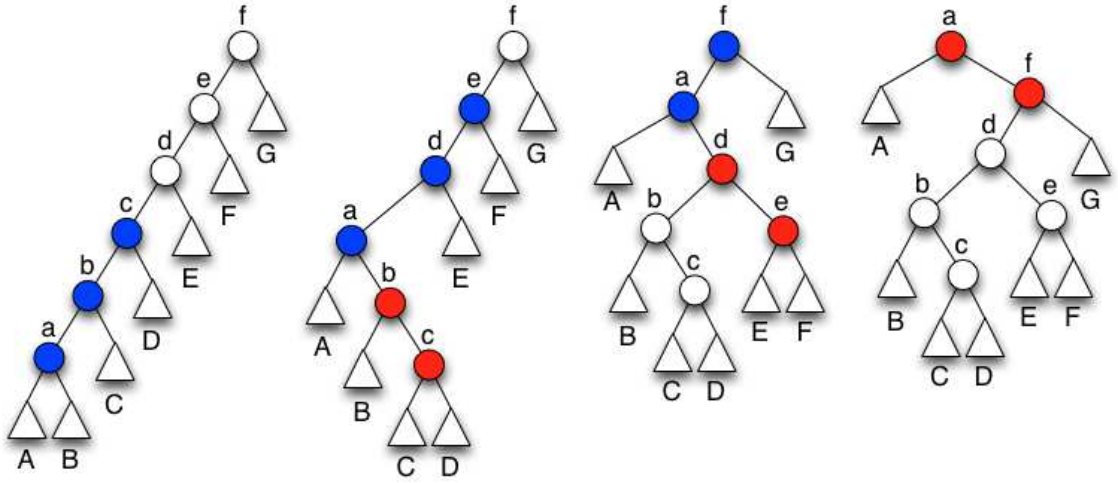
ボトムアップなスプレー木

- スプレー操作
  - x にアクセス → x がルートになる



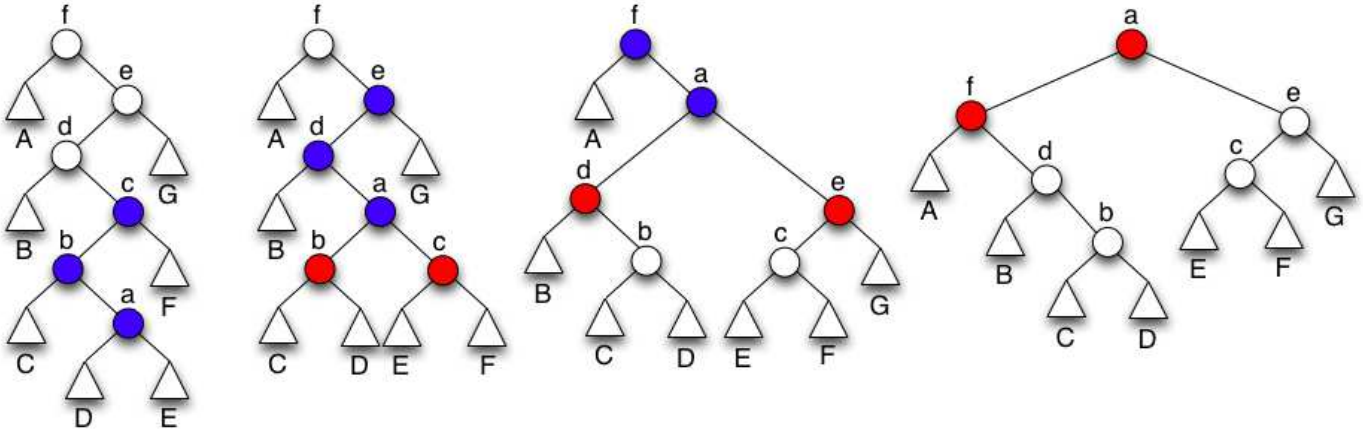
ボトムアップな zig zig

- a にアクセス



ボトムアップな zig zag

- a にアクセス



ボトムアップの問題点

- 下に一旦降りてから上に登るので、
  パスの二倍の手間がかかる
- 永続二分木には親へのリンクがない！

永続二分木では
ボトムアップなスプレー木は
実装できないの？

できるんです。
そう、Zipper ならね。

Zipper

- パスの情報をすべて格納したスタック

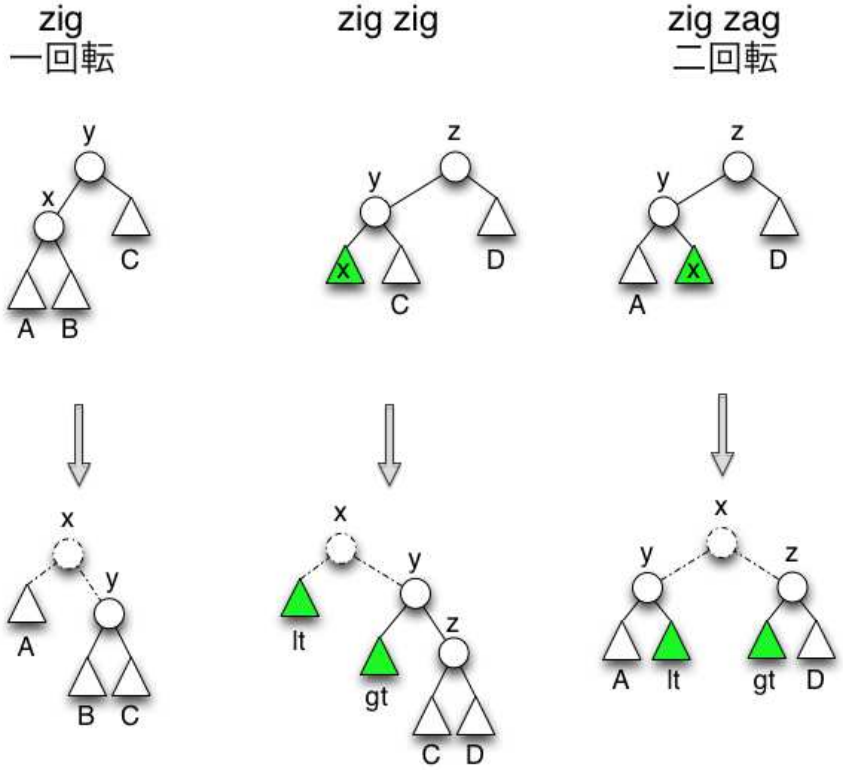
```
data Direction a = L a (Splay a) | R (Splay a) a

type Path a = [Direction a]
```

- Zipper を使ったボトムアップ・スプレー

```
splay :: Splay a -> Path a -> Splay a
splay t []                  = t
splay Leaf (L x r : bs)     = splay (Node Leaf x r) bs
splay Leaf (R l x : bs)     = splay (Node l x Leaf) bs
splay (Node a x b) [L y c] = Node a x (Node b y c) -- zig
splay (Node b y c) [R a x] = Node (Node a x b) y c -- zig
splay (Node a x b) (L y c : L z d : bs)
    = splay (Node a x (Node b y (Node c z d))) bs  -- zig zig
splay (Node b x c) (R a y : L z d : bs)
    = splay (Node (Node a y b) x (Node c z d)) bs  -- zig zag
splay (Node c z d) (R b y : R a x : bs)
    = splay (Node (Node (Node a x b) y c) z d) bs  -- zig zig
splay (Node b x c) (L y d : R a z : bs)
    = splay (Node (Node a z b) x (Node c y d)) bs  -- zig zag
```

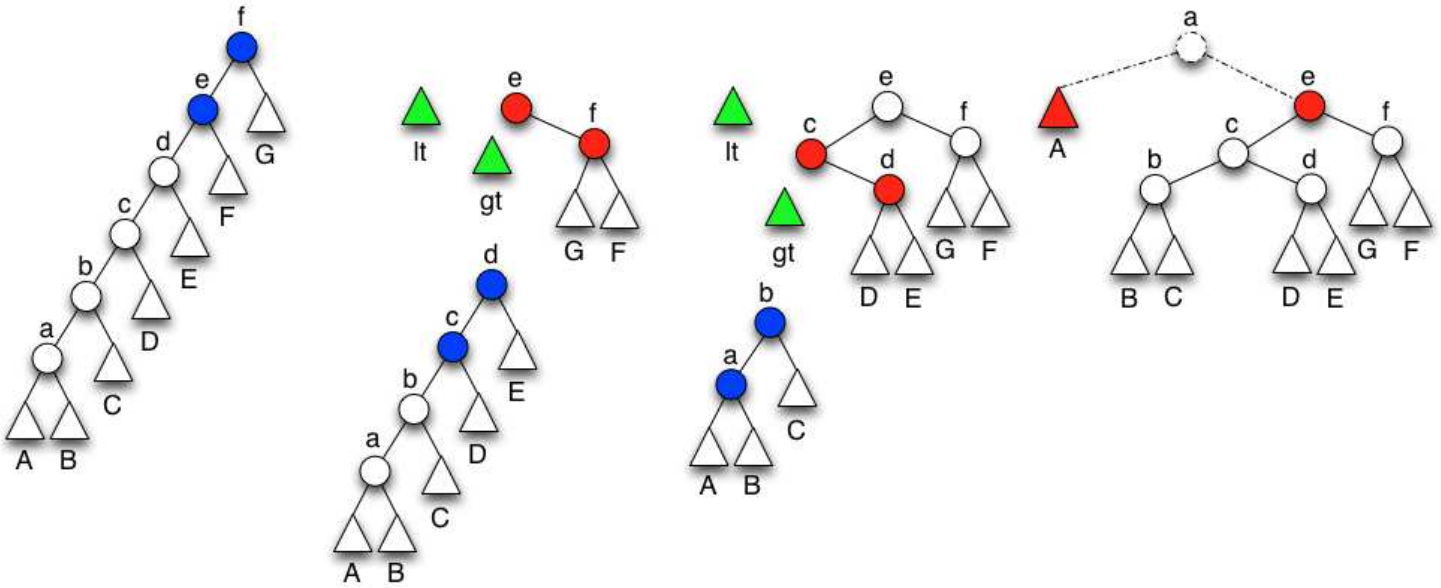
トップダウンなスプレー木

- スプレー操作
  - x にアクセス



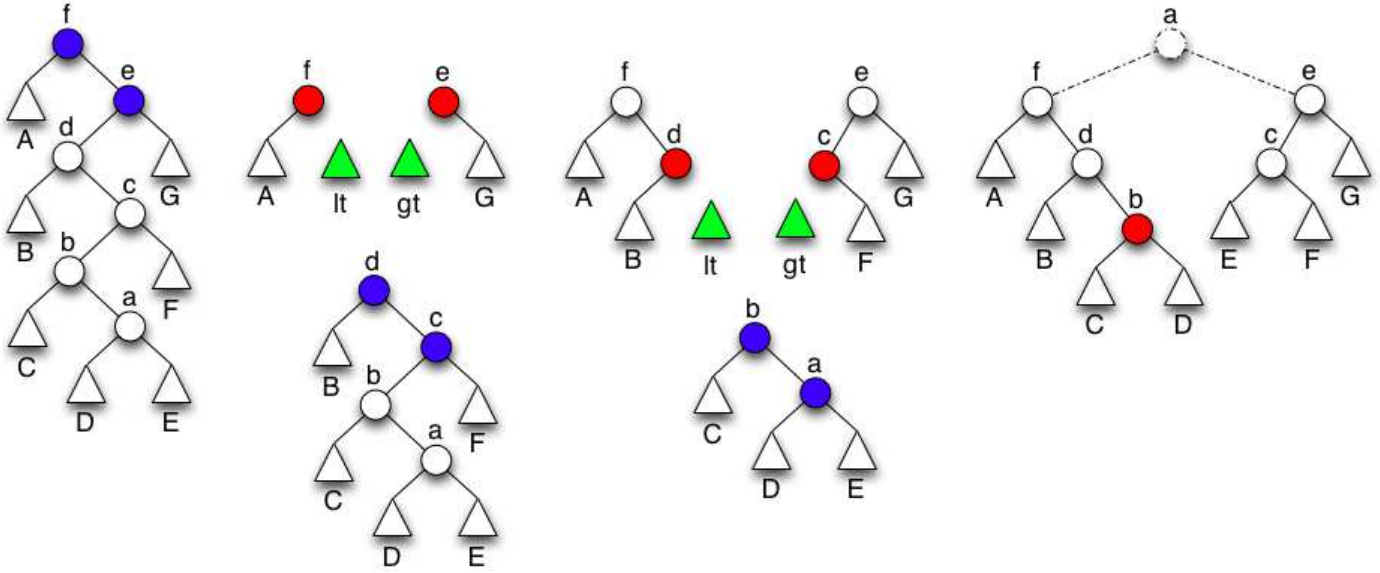
トップダウンな zig zig

- a にアクセス



トップダウンな zig zag

- a にアクセス



スプレー木の partition

```
split :: Ord a => a -> Splay a -> (Splay a, Bool, Splay a)
split _ Leaf = (Leaf,False,Leaf)
split k x@(Node xl xk xr) = case compare k xk of
  EQ -> (xl, True, xr)
  GT -> case xr of
    Leaf -> (x, False, Leaf)
    Node yl yk yr -> case compare k yk of
      EQ ->     (Node xl xk yl, True, yr)       -- R  :zig
      GT -> let (lt, b, gt) = split k yr        -- RR :zig zag
            in  (Node (Node xl xk yl) yk lt, b, gt)
      LT -> let (lt, b, gt) = split k yl        -- RL :zig zig
            in  (Node xl xk lt, b, Node gt yk yr)
  LT -> case xl of
    Leaf          -> (Leaf, False, x)
    Node yl yk yr -> case compare k yk of
      EQ ->     (yl, True, Node yr xk xr)       -- L  :zig
      GT -> let (lt, b, gt) = split k yr        -- LR :zig zag
            in  (Node yl yk lt, b, Node gt xk xr)
      LT -> let (lt, b, gt) = split k yl        -- LL :zig zig
            in  (lt, b, Node gt yk (Node yr xk xr))
```

スプレーヒープ

- 探索木をヒープとして使う
  - 順序付けられているので最小値は分かる
- 要素の重複を許す
  - partition では左の部分木へ
- minimum (findMin)
  - 最小値を返す
  - 木は再構築しない
- deleteMin
  - 最小値を削り、木を再構築して返す

スプレーヒープの partition

```
partition :: Ord a => a -> Splay a -> (Splay a, Splay a)
partition _ Leaf = (Leaf, Leaf)
partition k x@(Node xl xk xr) = case compare k xk of
  LT -> case xl of
    Leaf          -> (Leaf, x)
    Node yl yk yr -> case compare k yk of
      LT -> let (lt, gt) = partition k yl        -- LL :zig zig
            in  (lt, Node gt yk (Node yr xk xr))
      _  -> let (lt, gt) = partition k yr        -- LR :zig zag
            in  (Node yl yk lt, Node gt xk xr)
  _ -> case xr of
    Leaf          -> (x, Leaf)
    Node yl yk yr -> case compare k yk of
      LT -> let (lt, gt) = partition k yl
            in  (Node xl xk lt, Node gt yk yr)   -- RL :zig zig
      _  -> let (lt, gt) = partition k yr        -- RR :zig zag
            in  (Node (Node xl xk yl) yk lt, gt)
```

minimum

- minimum を O(1) へ

```
data Heap a = None | Some a (Splay a)

minimum :: Heap a -> Maybe a
minimum None       = Nothing
minimum (Some m _) = Just m

deleteMin :: Heap a -> Heap a
deleteMin None       = None
deleteMin (Some _ t) = fromMaybe None $ do
    t' <- deleteMin' t
    m  <- findMin' t'
    return $ Some m t'

deleteMin' :: Splay a -> Maybe (Splay a)
deleteMin' = 本の通り
```