Lec 18 Red-Black Trees

Signature design

```
Signature DICT =
  Sig
    type key = string
    type 'a entry = key * 'a
    type 'a dict
    val empty : 'a dict
    val lookup : 'a dict -> key -> 'a
    val insert : 'a dict * 'a entry -> 'a dict
  end
```

Red-Black tree (RBT) – a self-balancing tree

AVL – bruteforce
RBT – slicker, less strict,
still works good

* Rep invar

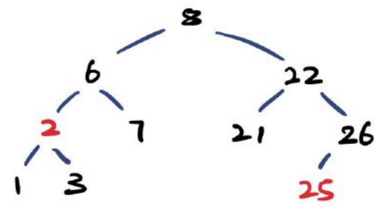
- temporarily broken -> identify place of imbalance -> reestablish balance

```
datatype 'a dict = Black of }  'a dict * 'a entry * 'a dict
                 | Red   of }
                 | Empty  <- consider this black
```

* RBT invars

1. Tree is ordered
2. No red node has a red child viz. no two red in row
3. Every node has well-defined black height
   viz. same num of black to leaves in all path downward

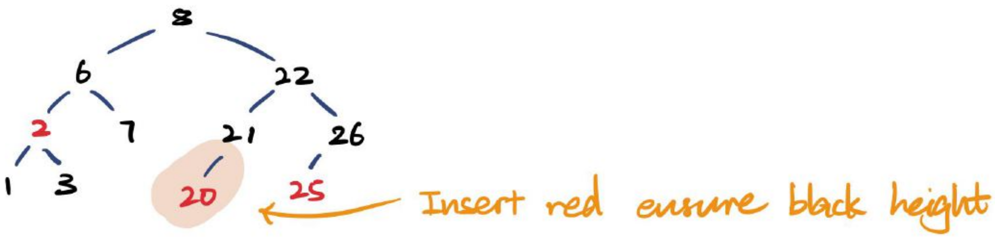
Ex.

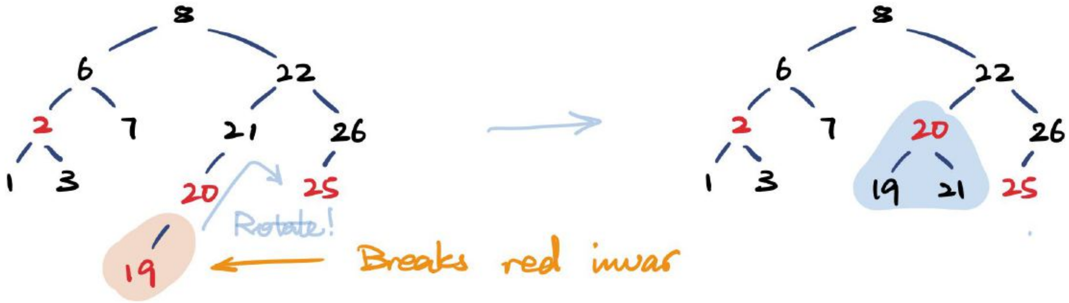


Why good invar ?

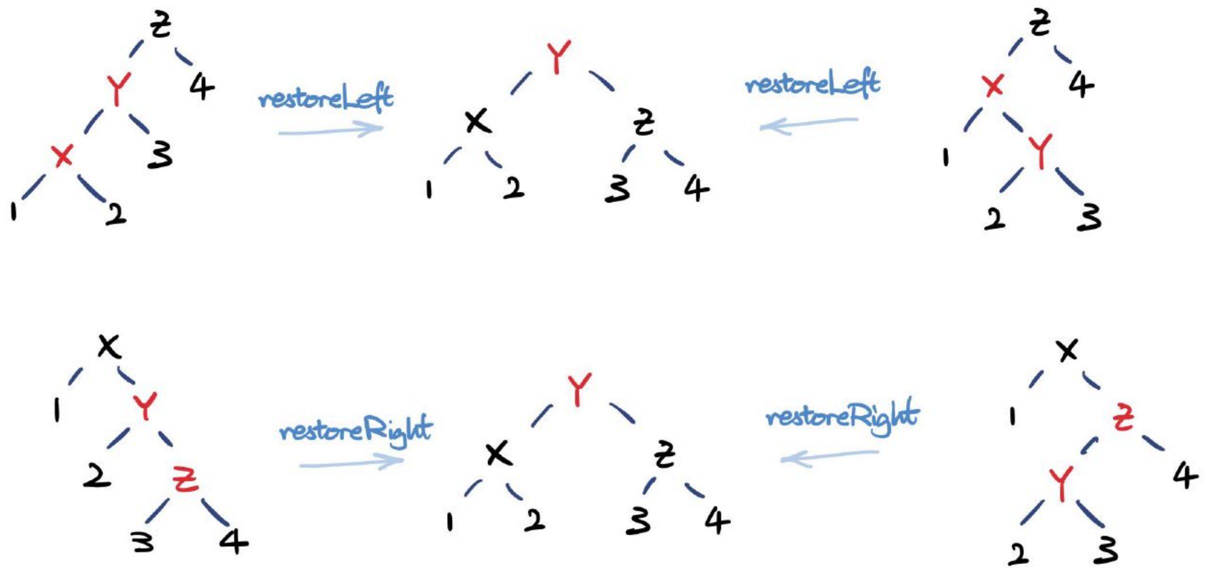
- If no red node then perfectly balanced.
- At most half red in path, at least zero red
  ⇒ longest path ≤ 2× shortest path.
  ⇒ depth ≤ $\underbrace{2 \log_2 (|\text{nodes}| + 1)}_{\text{good enough}}$

Insertion

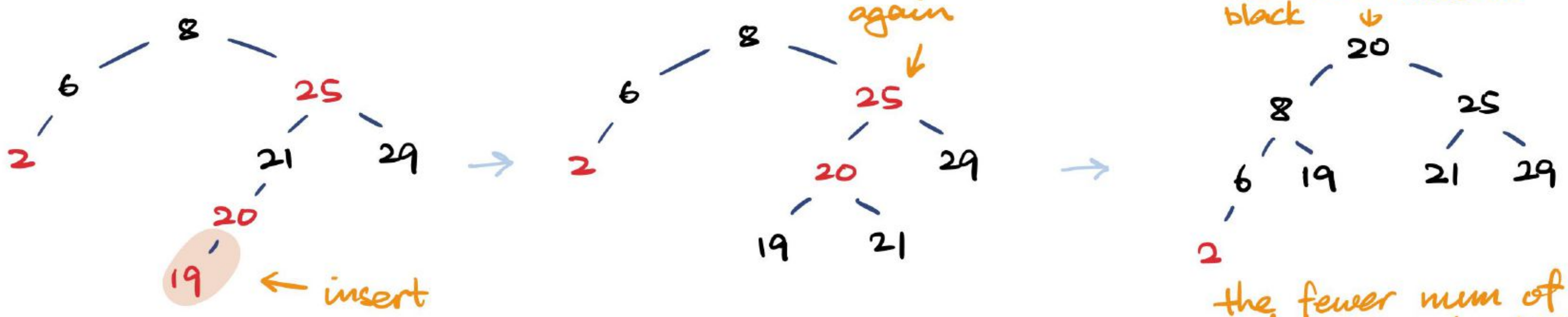




Rotation in general cases



# One more problem



more violation. Need to rotate again

root color doesn't break invar but we can make it black

insert

the fewer num of red maybe the less rebalance to do

ARBT – Almost Red-Black Tree

What do we know in case of red-red violation

1. As before
2'. As (2) but a red root can have one child
3. As before

# Code

```
(* restoreLeft : 'a dict -> 'a dict
   REQ  D is RBT or (D's root is black and left child is ARBT and right is RBT)
   ENS  restoreLeft D is RBT with same elems
*)
fun  restoreLeft (Black(Red (Red(d1, x, d2), Y, d3), Z, d4)
         = Red( Black(d1, x, d2), Y, Black(d3, Z, d4))
  |  restoreLeft (Black (Red (d1, x, Red(d2, Y, d3), Z, d4)
         = Red( Black(d1, x, d2), Y, Black(d3, Z, d4))
  |  restoreLeft D
         = D
```

[restoreRight just mirror]

```
(* ins : 'a dict -> 'a dict
   REQ  D is RBT
   ENS  ins D have same black height as D
        ins D is :  RBT if D black
                    ARBT if D red
        ins D has right elems
```

```
(* insert : 'a dict * 'a entry -> 'a dict
   REQ  D is RBT
   ENS  insert(D, e)  is RBT with  right elems
*)
fun insert (D , e as (k, _)) =
  let
    fun ins Empty = Red (Empty, e, Empty)
      | ins (Black (L, e' as (k', _) , R)) =
          (case String.compare (k, k') of
             EQUAL => Black (L, e, R) |
             LESS    => restoreLeft ( Black (ins L, e', R)) |
             GREATER => restoreRight ( Black (L, e', ins R))
          )
      | ins (Red (L, e' as (k', _) , R)) =
          (case String.compare (k, k') of
             EQUAL => Red (L, e, R) |          L must be Black,
             LESS    => Red (ins L, e', R)     so ins L must be RBT
             GREATER => Red (L, e', ins R)
          )                                    simili
  in
    ( case ins D of
        Red t => Black t |
        D'    => D'
    )
  end

[ lookup omitted ]
```