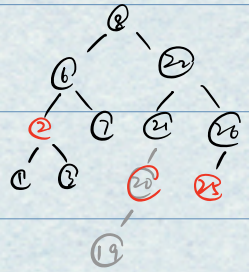
Red Black Tree

```
datatype 'a dict = Empty                          (* considered Black *)
                 | Red   of 'a dict * 'a entry * 'a dict
                 | Black of 'a dict * 'a entry * 'a dict
```

RBT invariants

1) sorted on the key
2) children of Red are Black
3) # Black on any path from the node down to an Empty is the same — "well-balanced black height"

depth $\leq 2 \log_2(|\text{nodes}| + 1)$



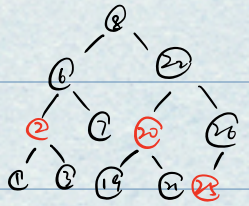
valid RBT!

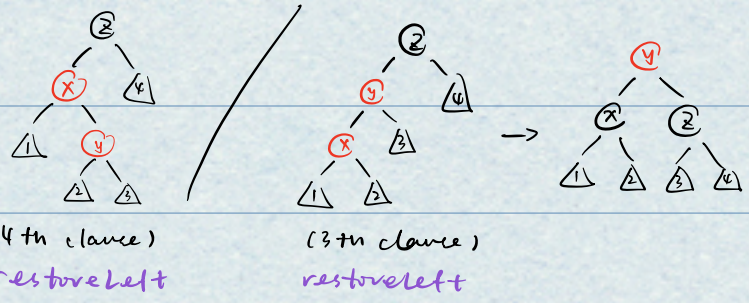
insert 25 => must be red to keep (iii) invariant

insert 19 => red? (i) violation!

=> black? (iii) violation!

We need a rotation!





(4th clause) restoreLeft

(3th clause) restoreLeft

Almost RBT (ARBT) invariants:

(ii) Red root may have one Red child

(i) & (iii) the same

```
(* restoreLeft : 'a dict -> 'a dict
   REQUIRES: d is RBT / d's root is Black, its left child is ARBT, right child is RBT
   ENSURES: restoreLeft(d) is RBT, same black height as d
*)
fun restoreLeft (Black(Red(Red(d1, x, d2), y, d3), z, d4)) =
```

Red (Black (a1, x, a2), y, Black (a3, z, a4))

Insert

```
(* insert : 'a dict * 'a entry -> 'a dict
   RE & EN: RBT
*)
fun insert (d, e as (k, v)) =
    let
      fun ins = ---
    in
      (case ins(d) of
         Red (t as (Red(_), _, _)) => Black (t)
         Red (t as (_, _, Red(_))) => Black (t)
       | d' => d')

(* ins : 'a dict -> 'a dict
   RE: d is RBT
   EN: ins(Black(t)) is RBT, ins(Red(t)) is ARBT
*)
fun ins (Empty) = Red (Empty, e, Empty)   => can have Red - Red violation
  | ins (Black (l, e' as (k', _), r)) =
    case String.compare (k, k') of
      EQUAL => Black (l, e, r)            => replace
    | LESS  => restoreLeft (Black (ins (l), e', r))
    | _     => restoreRight (Black (l, e', ins(r)))
  | ins (Red (l, e' as (k', _), r)) =
    case String.compare (k, k') of
      EQUAL => Red (l, e, r)              => replace
    | LESS  => Red (ins (l), e', r)       } why don't call restore ? No need to !
    | _     => Red (l, e', ins(r))

(* lookup : 'a dict -> key -> 'a option *)
fun lookup d k =
    let
      fun lk (Empty) = NONE
        | lk (Red t) = lk' t
        | lk (Black t) = lk' t
      and lk' (l, (k', v), r) =
          (case String.compare (k, k') of
             EQUAL => SOME (v)
           | LESS => lk (l)
           | GREATER => lk (r))
    in
      lk d
```

mutual recursion

end

structure RBT :> DICT = struct ... end

val r1 = RBT.insert (RBT.empty, ("a", 1))

int RBT.dict

val r2 = RBT.insert (r1, ("b", 2))

val look2 = RBT.lookup r2

RBT.key -> int option

look2 "b" => SOME 2

look2 "c" => NONE