

15-150

Principles of Functional Programming

Slides for Lecture 18

Red Black Trees

March 25, 2025

Michael Erdmann

Main Lesson:

- How to maintain Representation Invariants within a structure when some code necessarily violates the invariants:
 - Localize the violation
 - Characterize the violation
 - Determine weakened invariants
 - Write code that re-establishes the original invariants from data satisfying the weakened invariants

Ancillary Lessons:

- Functional implementation of balanced trees
- Pattern-matching as code-by-picture ;-)

Dictionary Signature

```
signature DICT =  
sig  
  type key = string          (* concrete *)  
  type 'a entry = key * 'a   (* concrete *)  
  
  type 'a dict                (* abstract *)  
  
  val empty : 'a dict  
  
  val lookup : 'a dict -> key -> 'a option  
  val insert : 'a dict * 'a entry -> 'a dict  
end
```

Dictionary Implementation

```
signature DICT =  
sig  
  type key = string          (* concrete *)  
  type 'a entry = key * 'a   (* concrete *)  
  
  type 'a dict                (* abstract *)  
  
  val empty : 'a dict  
  
  val lookup : 'a dict -> key -> 'a option  
  val insert : 'a dict * 'a entry -> 'a dict  
end
```

Last week we implemented

```
structure BinarySearchTree : DICT = ...
```

using a tree to represent a dictionary,
with the Representation Invariant
that the tree is **sorted** on **key**
(and there are no duplicate keys).

Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

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

(Empty considered black.)

Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

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

(Empty considered black.)

Red Black Tree (RBT) Invariants:

- (1) The tree is **sorted** on the **key** part of the entries.
- (2) The **children** of a **Red** node are **Black**.
- (3) Each node has a well-defined *black height*:
The number of **Black** nodes on *any* path
from the node down to an **Empty** is the same.

Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

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

(Empty considered black.)

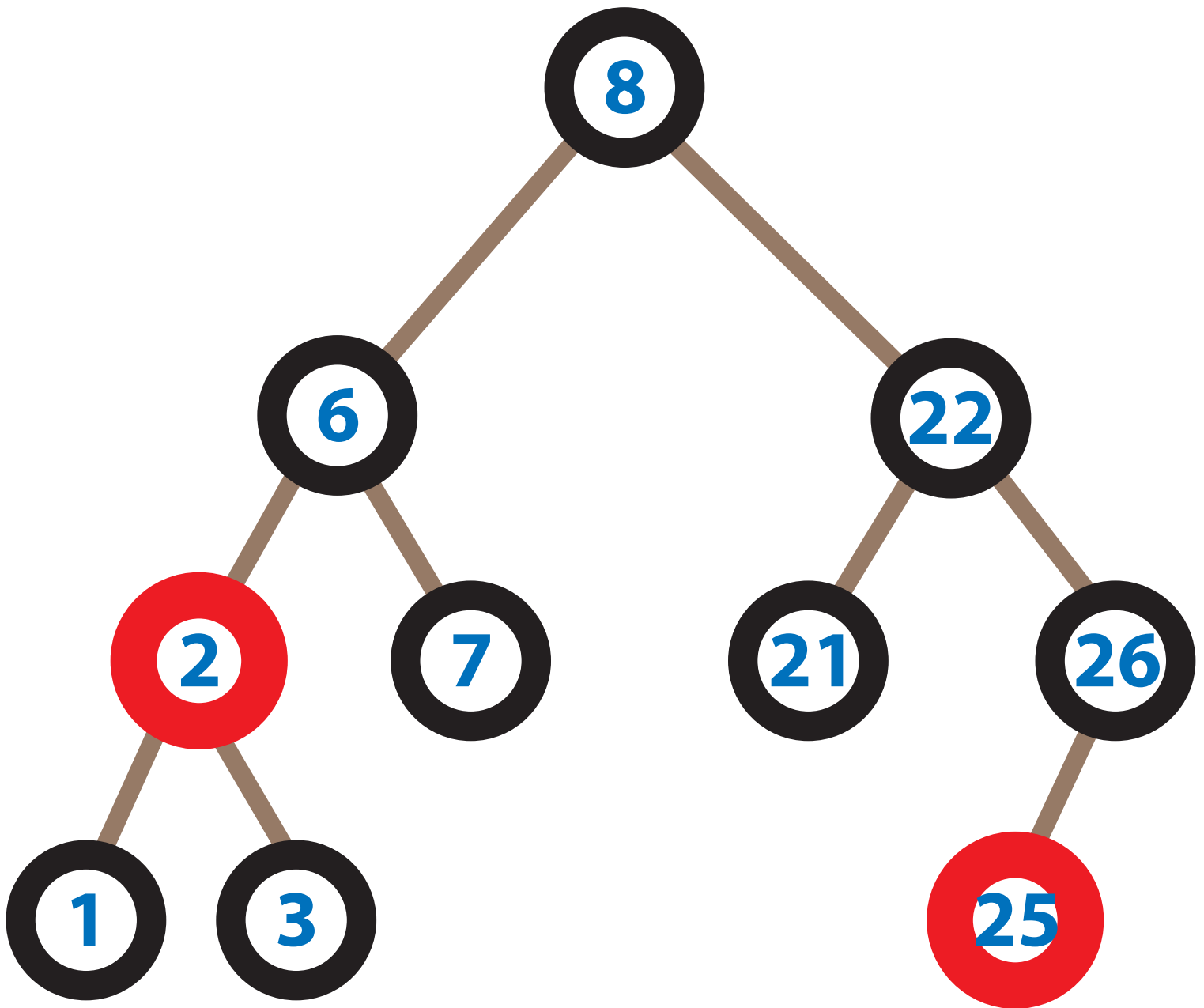
Red Black Tree (RBT) Invariants:

- (1) The tree is **sorted** on the **key** part of the entries.
- (2) The **children** of a **Red** node are **Black**.
- (3) Each node has a well-defined *black height*:
The number of **Black** nodes on *any* path
from the node down to an **Empty** is the same.

Invariants imply the tree is roughly balanced:

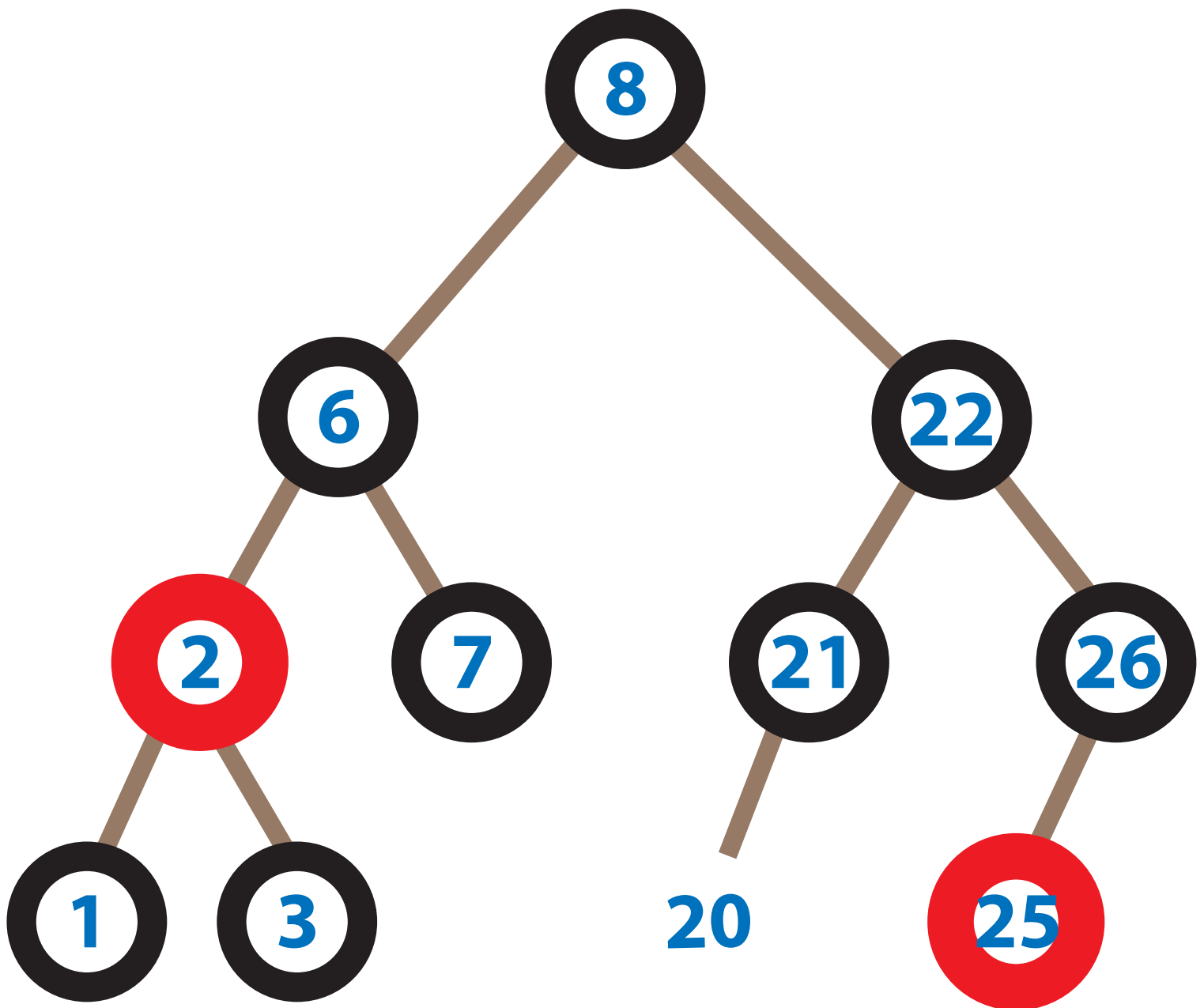
$$\text{depth} \leq 2 \lceil \log_2 (|\text{nodes}| + 1) \rceil$$

A given Red Black Tree:

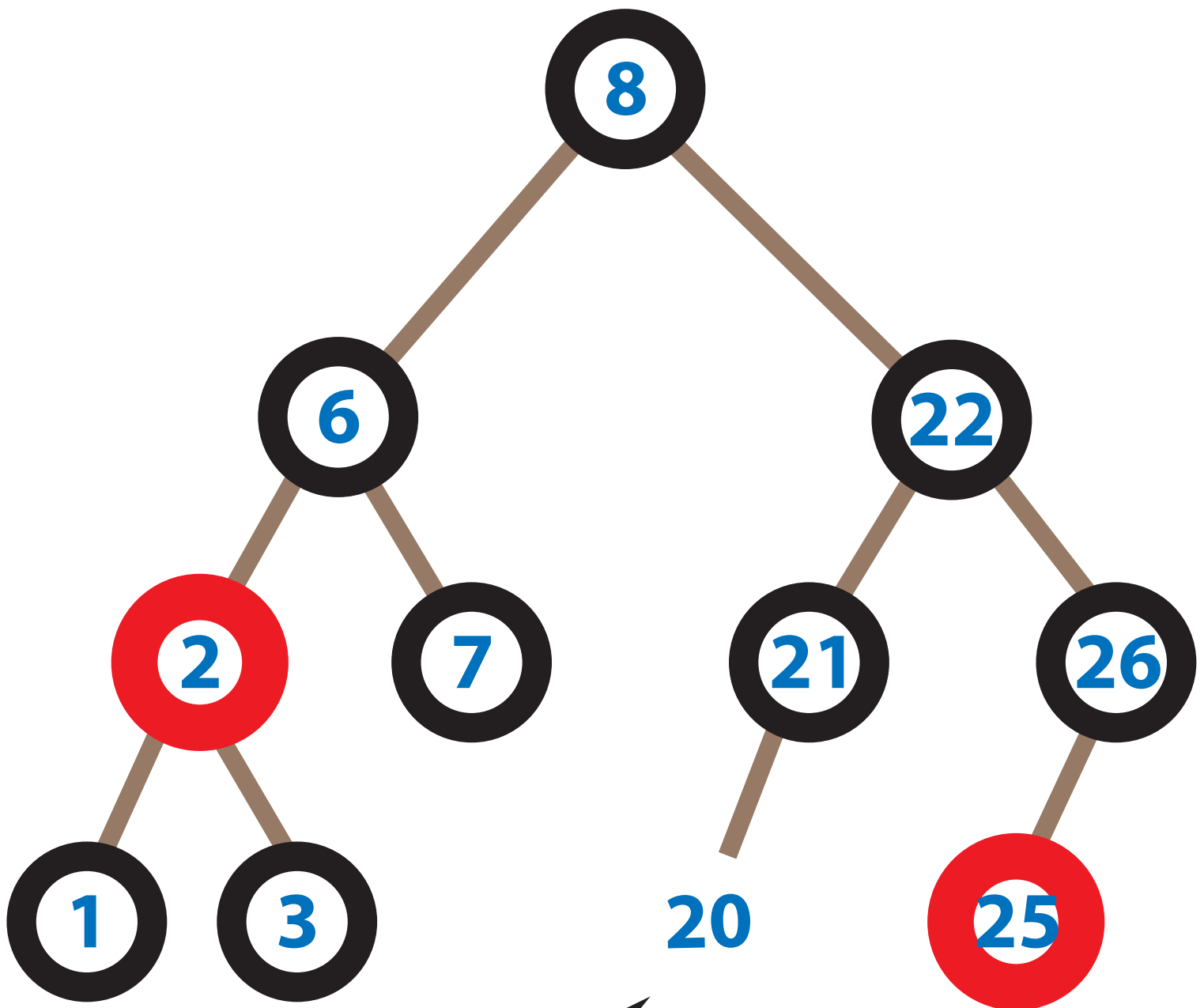


(For presentational simplicity, only showing keys, and using integer keys not strings.)

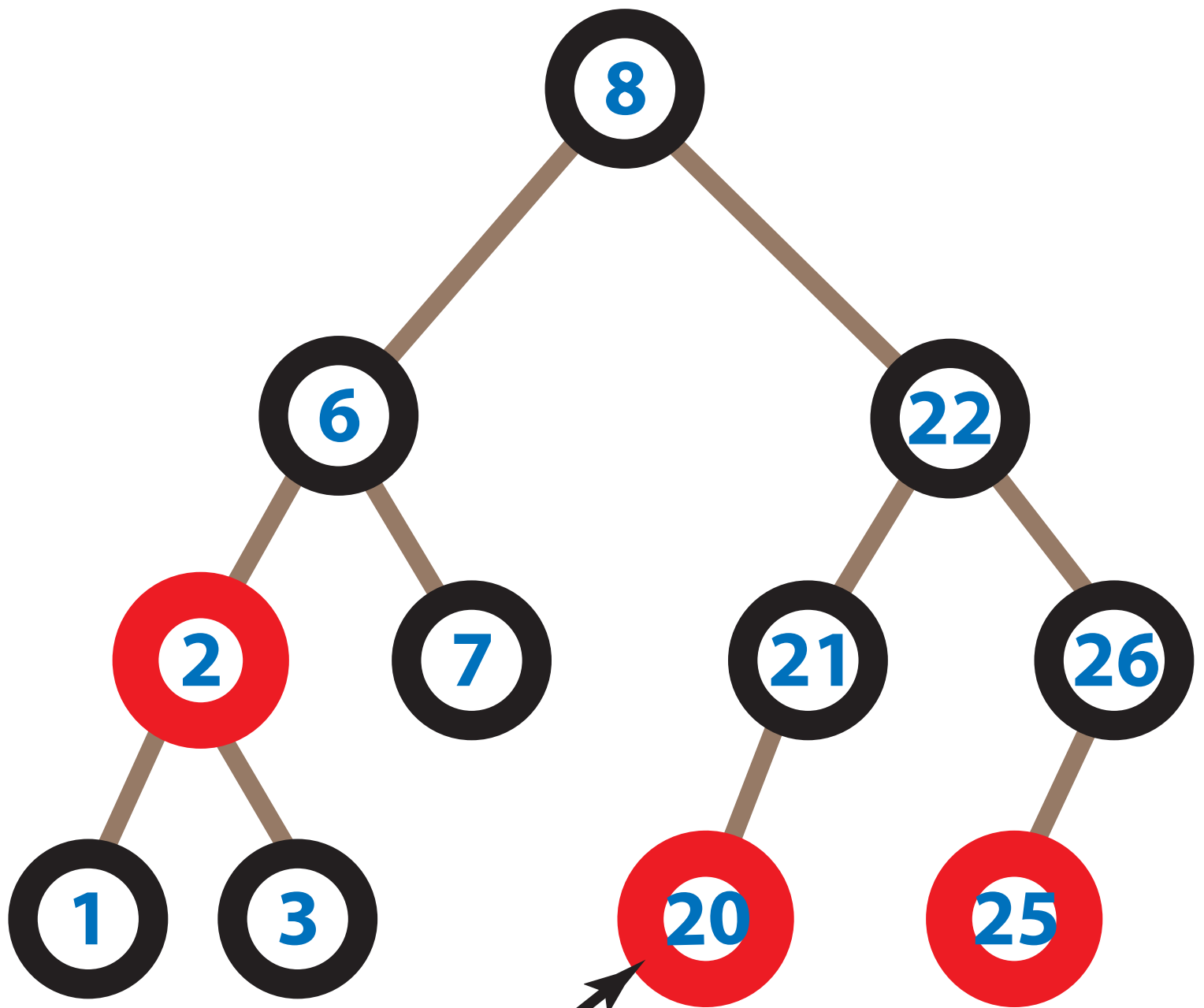
Now insert 20:



Now insert 20:

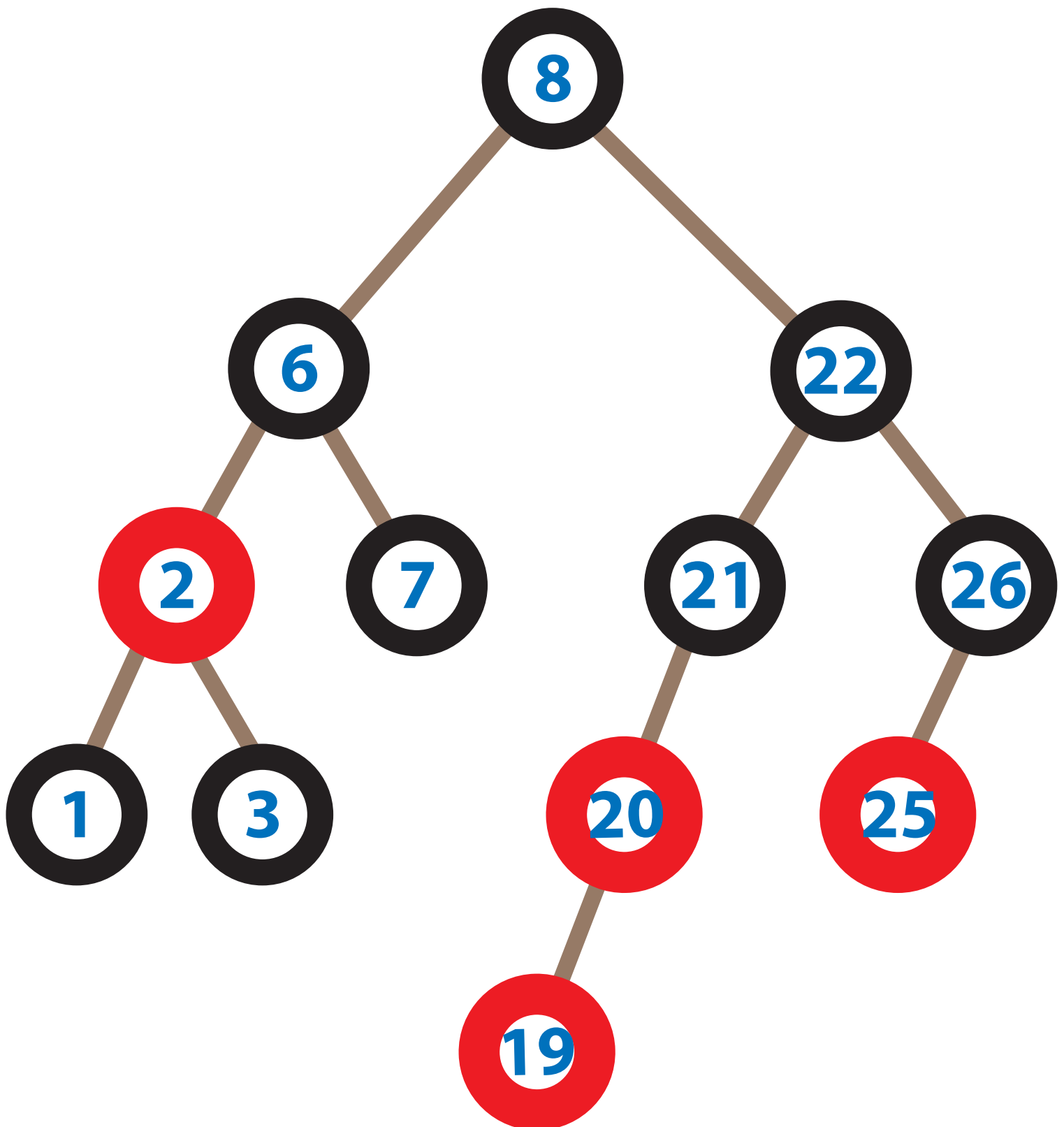


**What should we color
this node?**

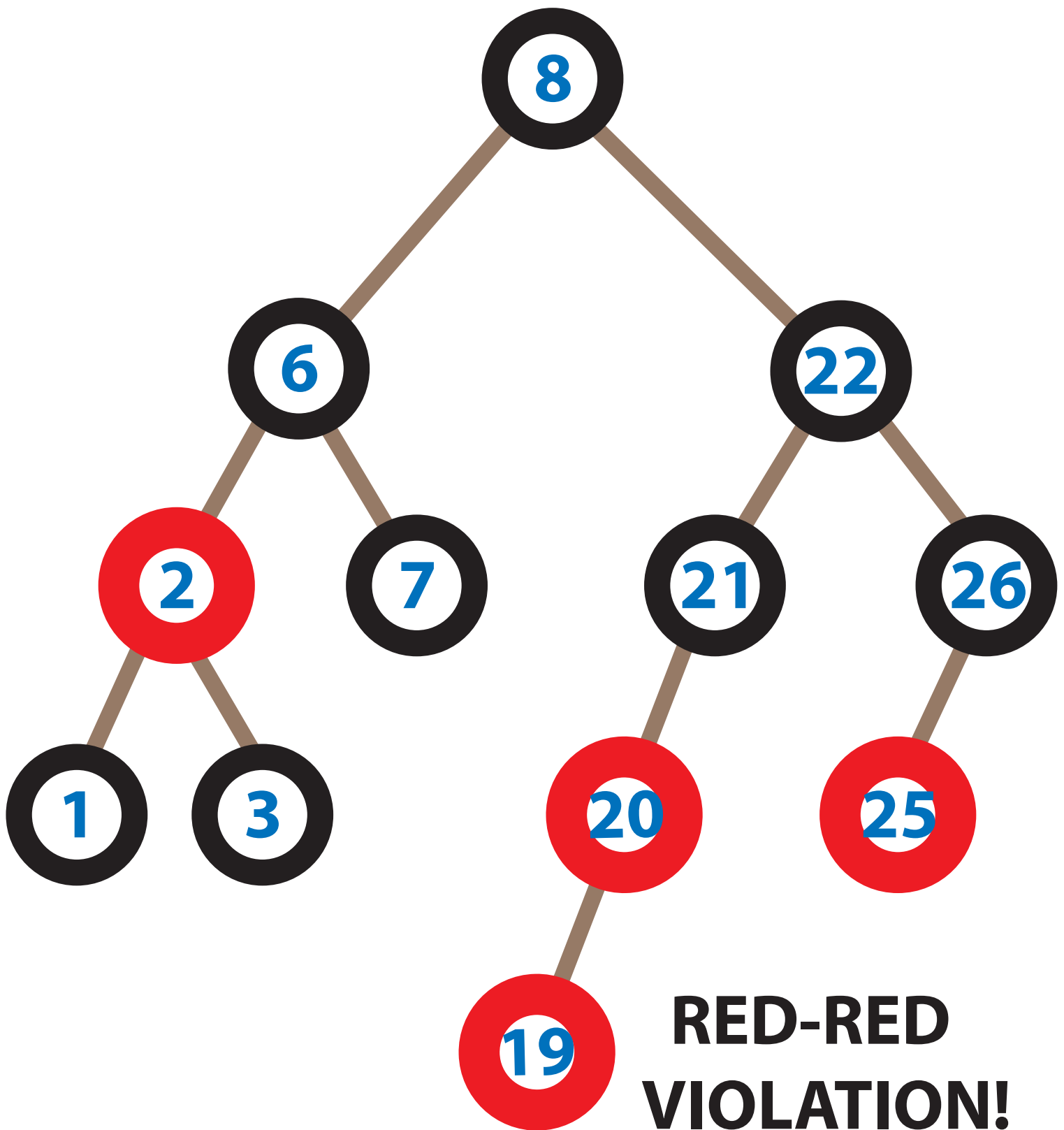


**Let's color it red,
to preserve black height.**

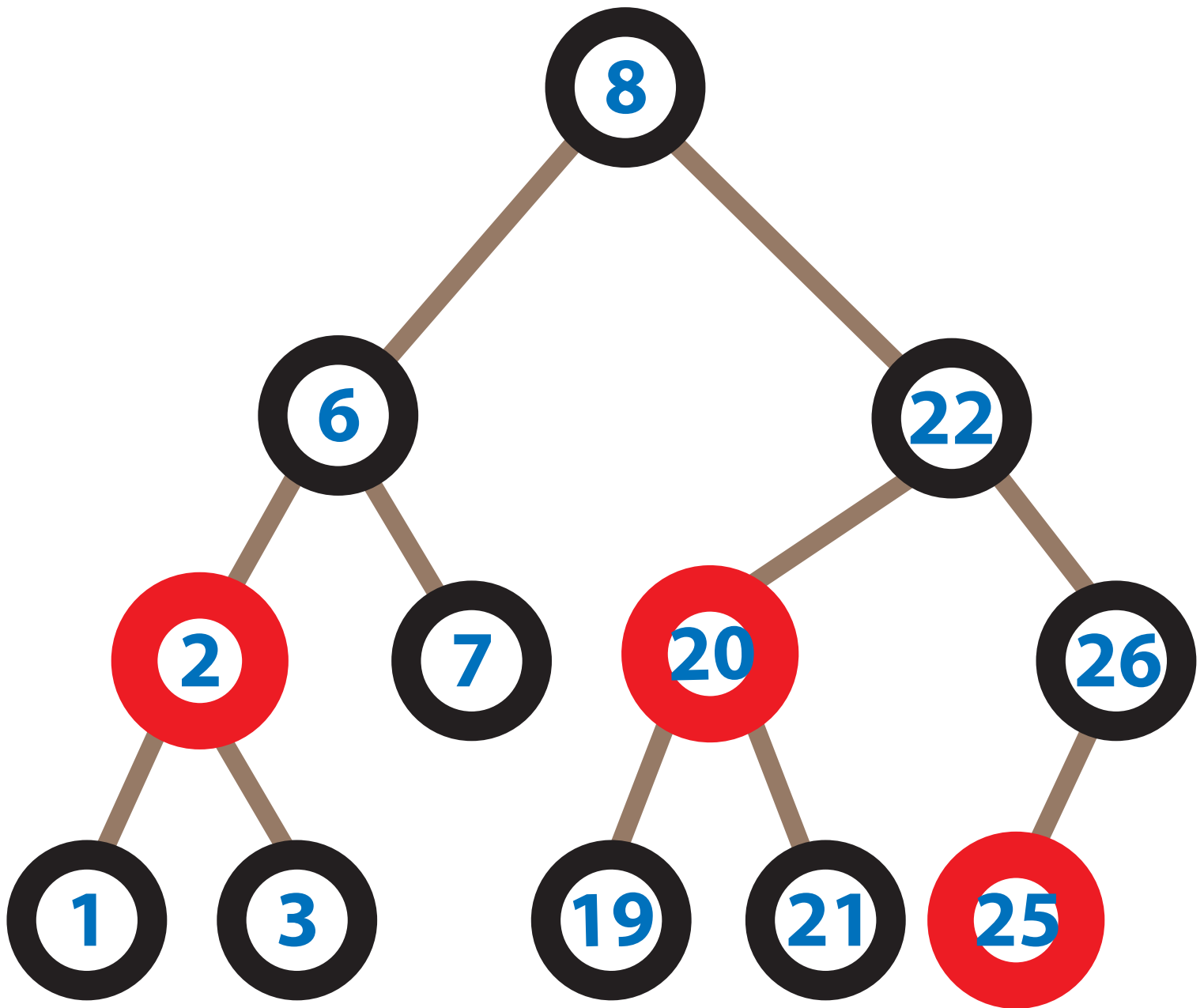
Now insert 19:



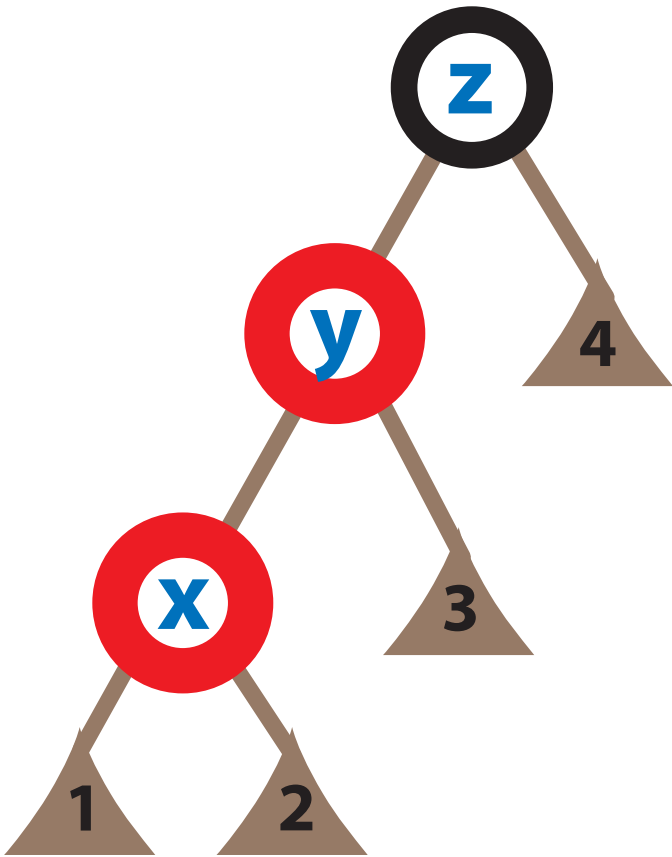
Now insert 19:



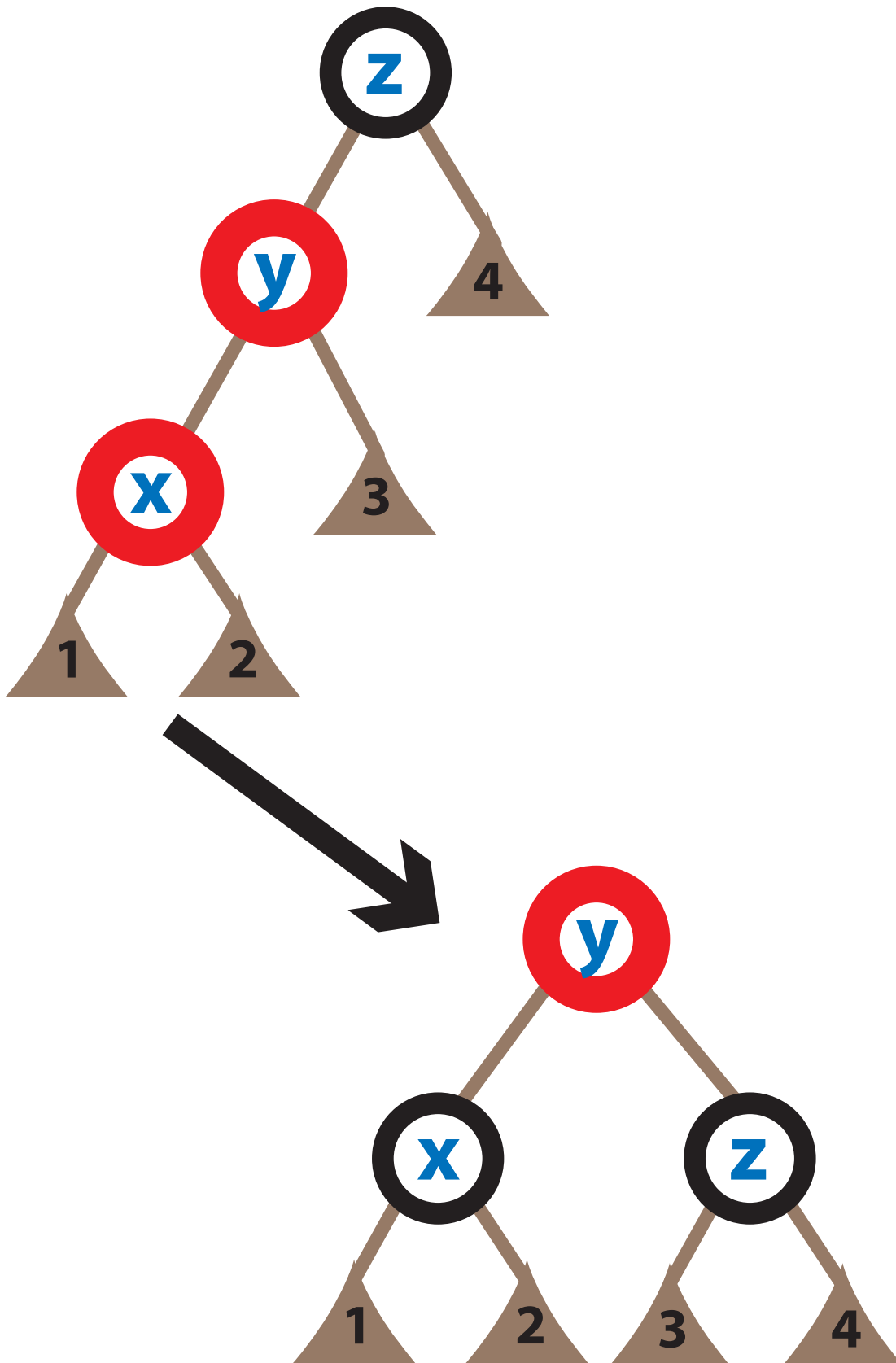
Fix with a rotation and recoloring:



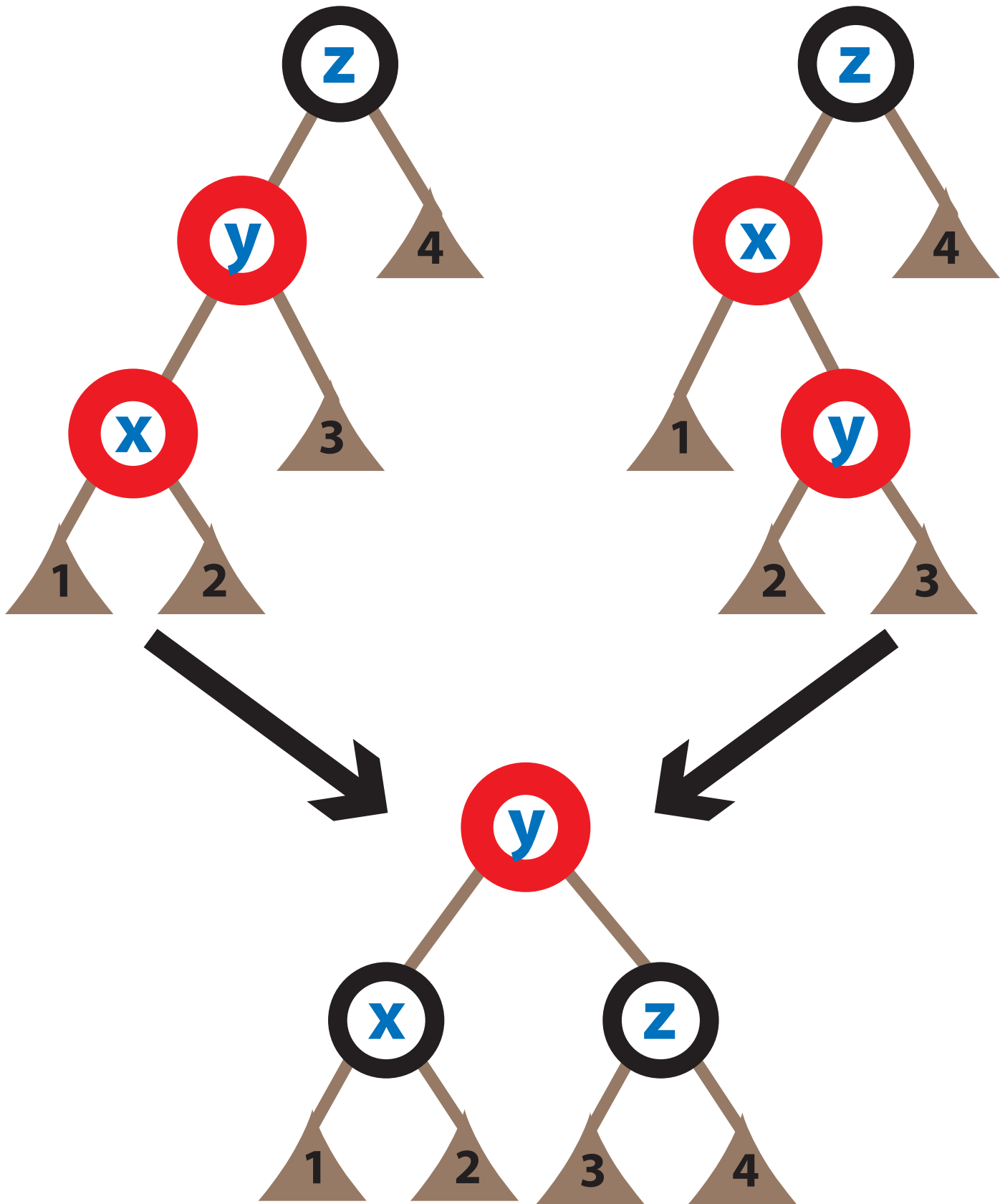
2 of the 4 possible kinds of rotations:



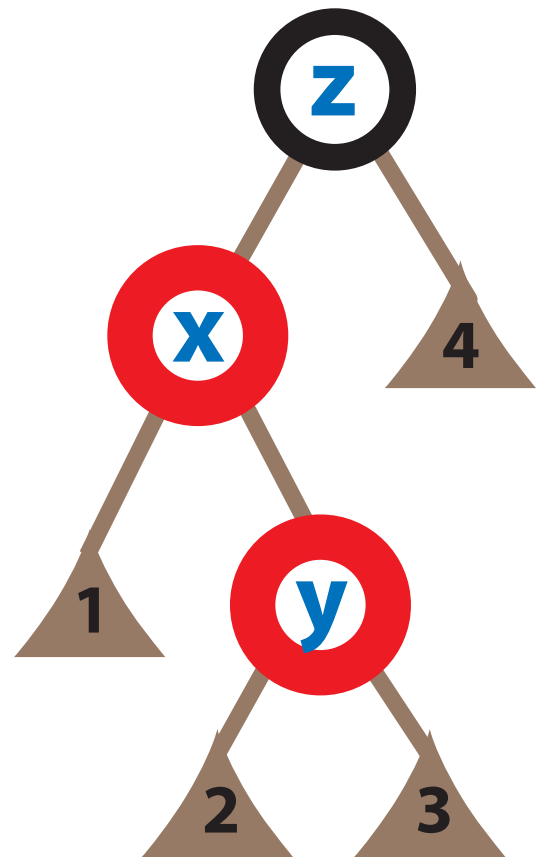
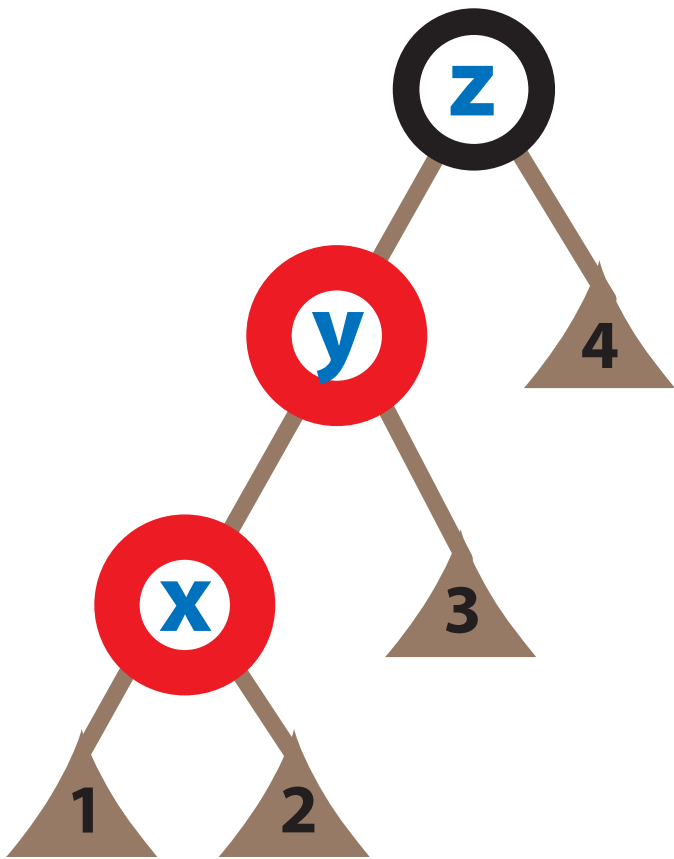
2 of the 4 possible kinds of rotations:



2 of the 4 possible kinds of rotations:

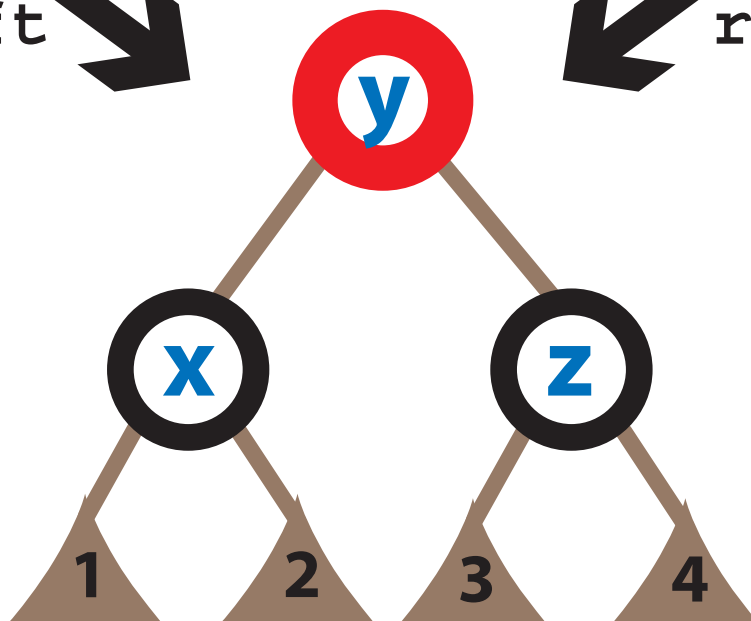


2 of the 4 possible kinds of rotations:

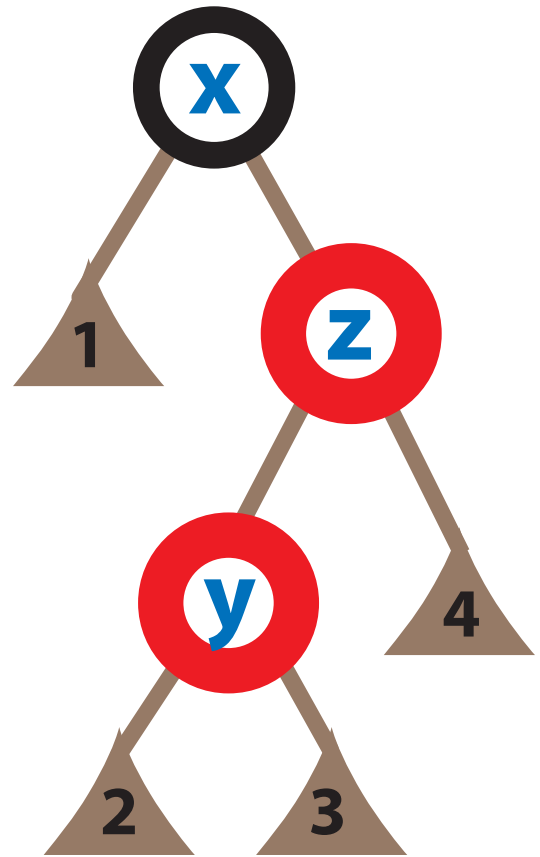
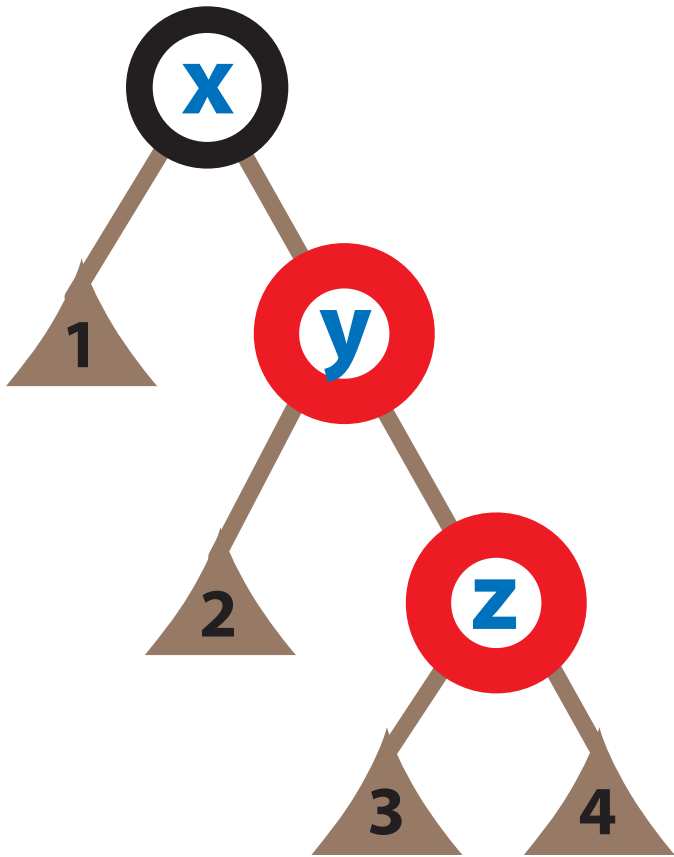


restoreLeft
(1st clause)

restoreLeft
(2nd clause)

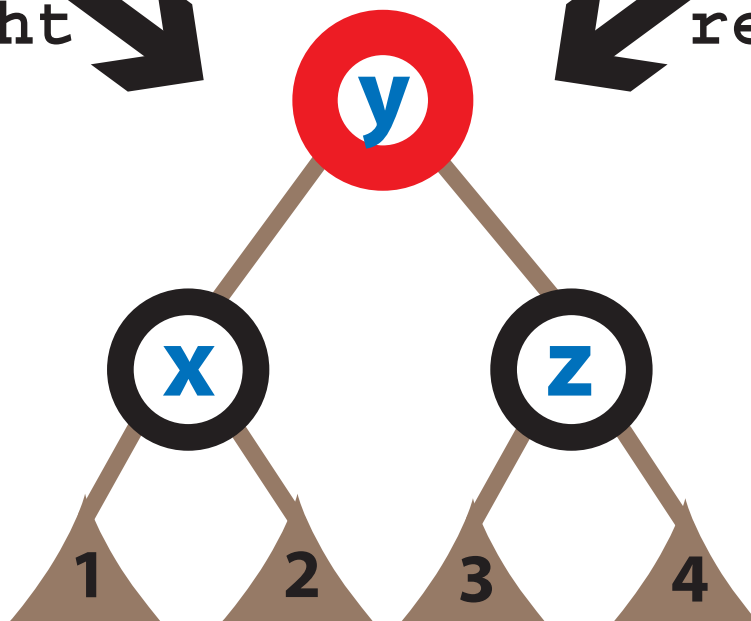


The other 2 kinds of rotations:

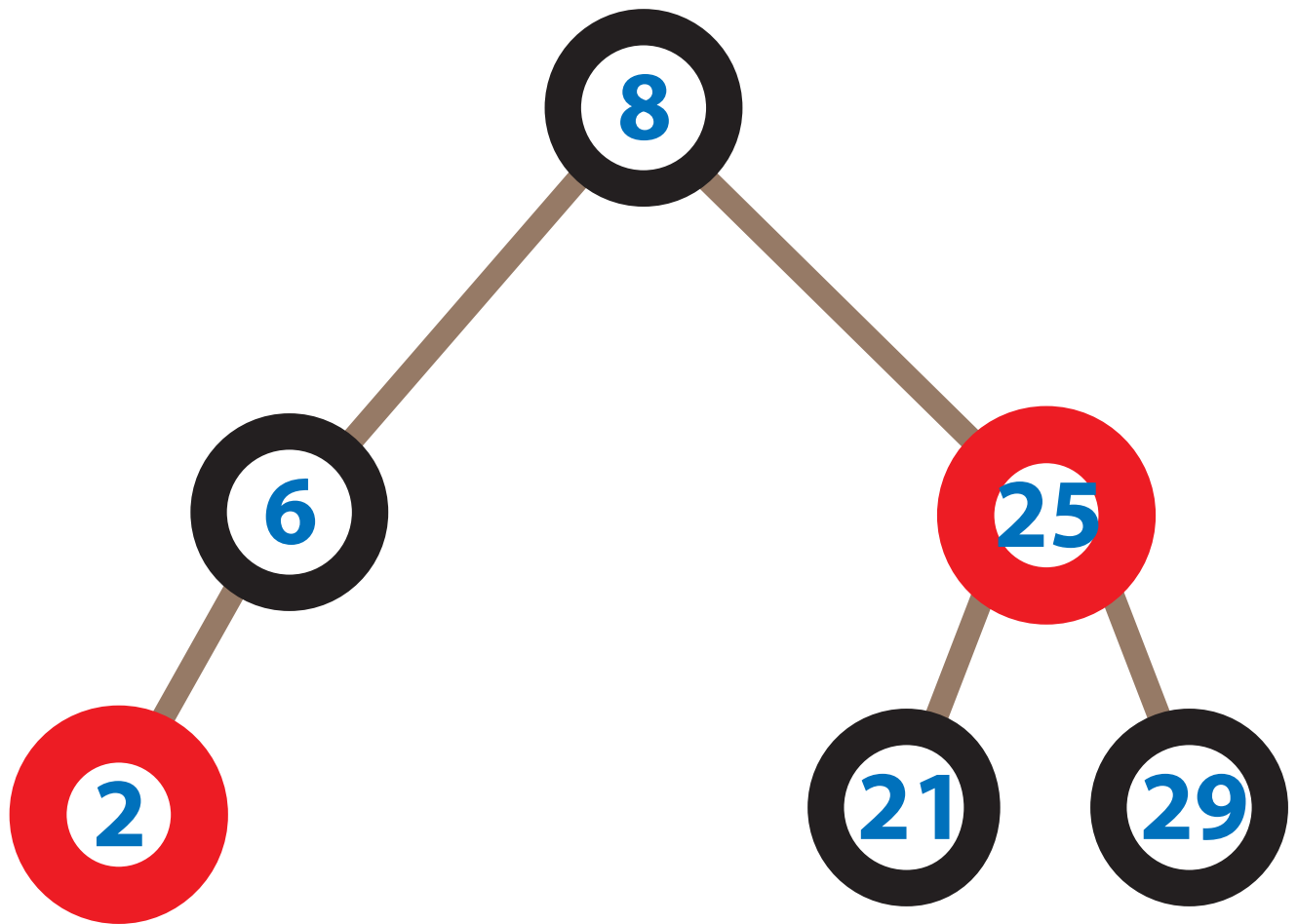


`restoreRight`
(1st clause)

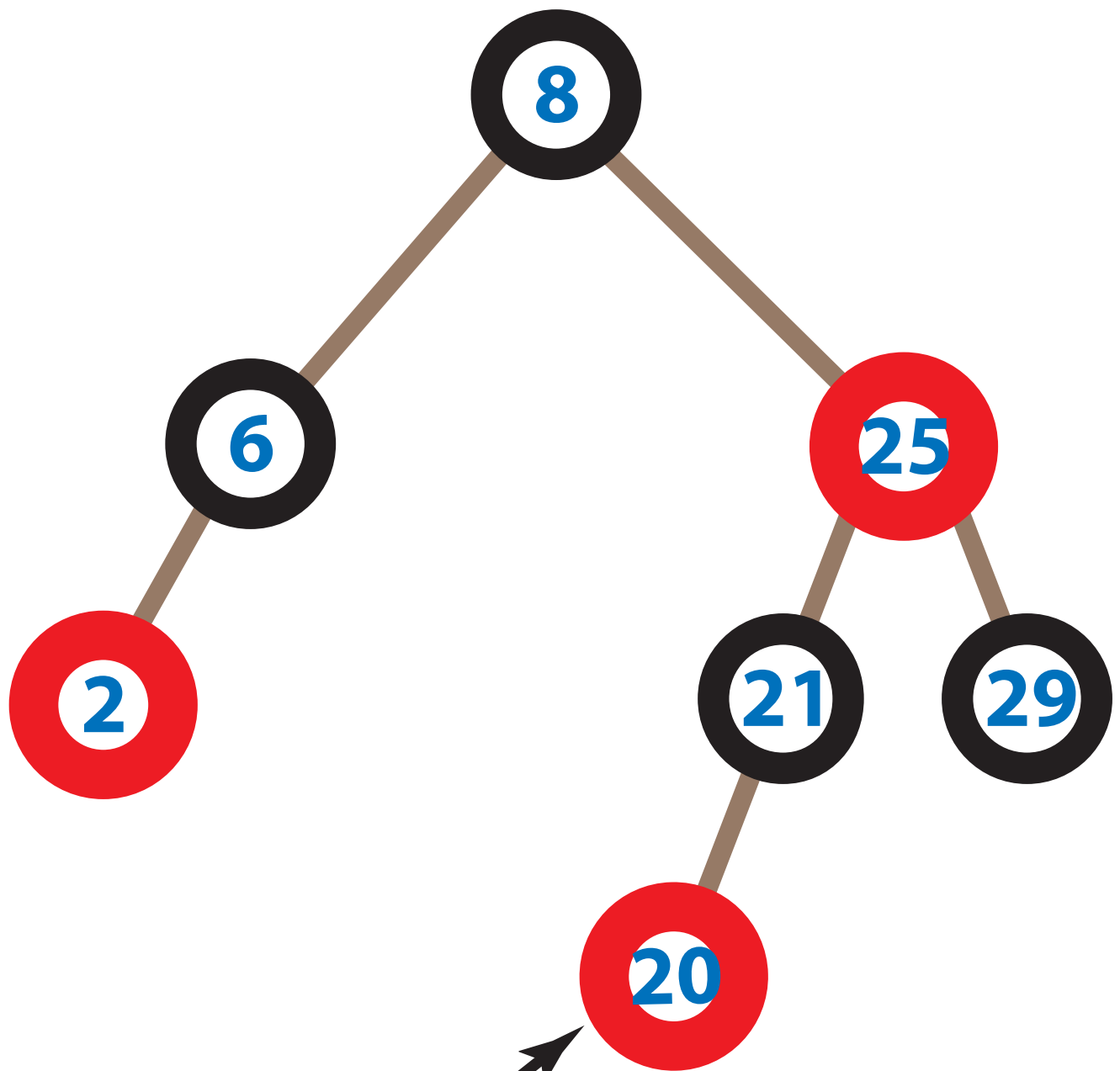
`restoreRight`
(2nd clause)



Here is another example:

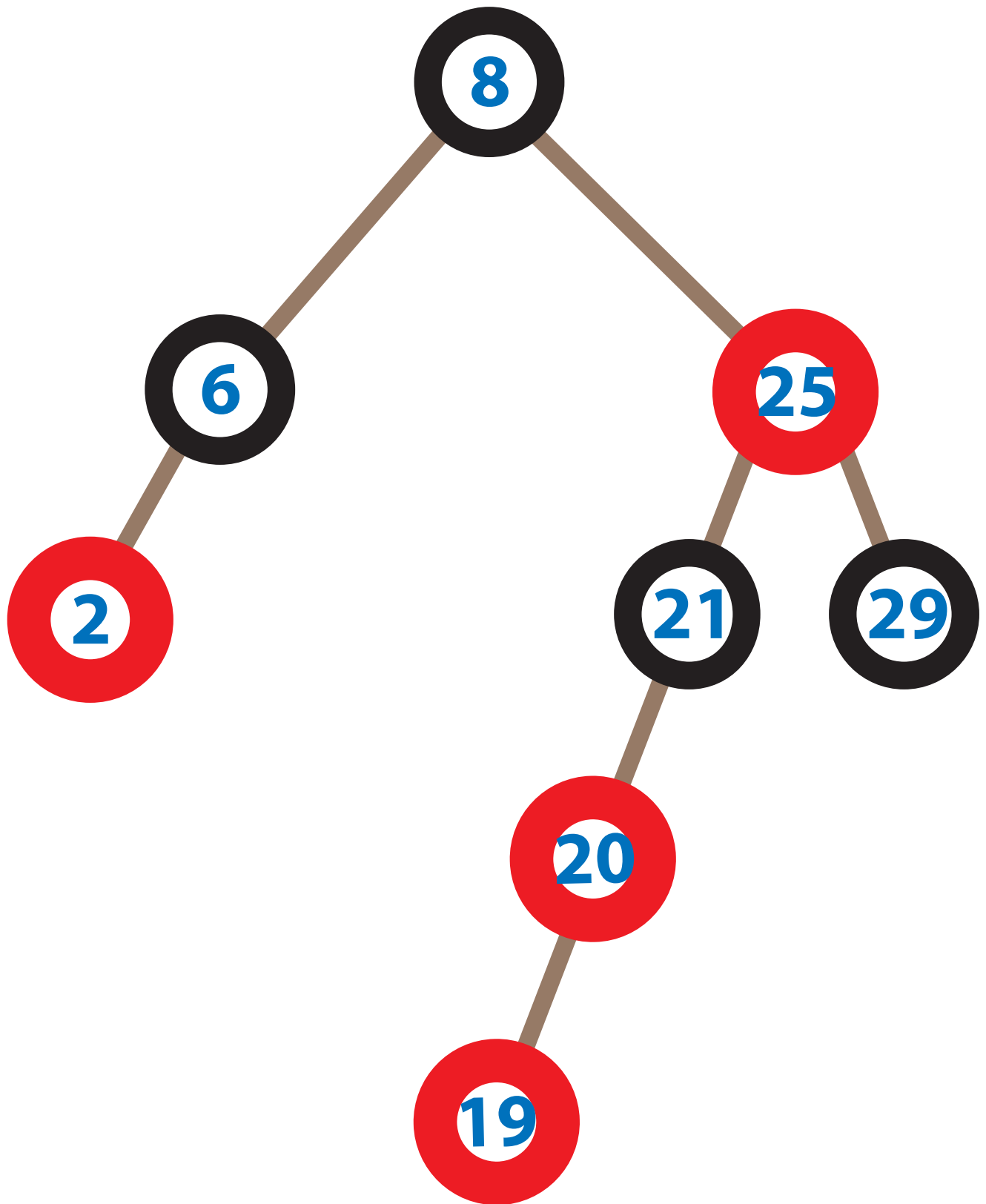


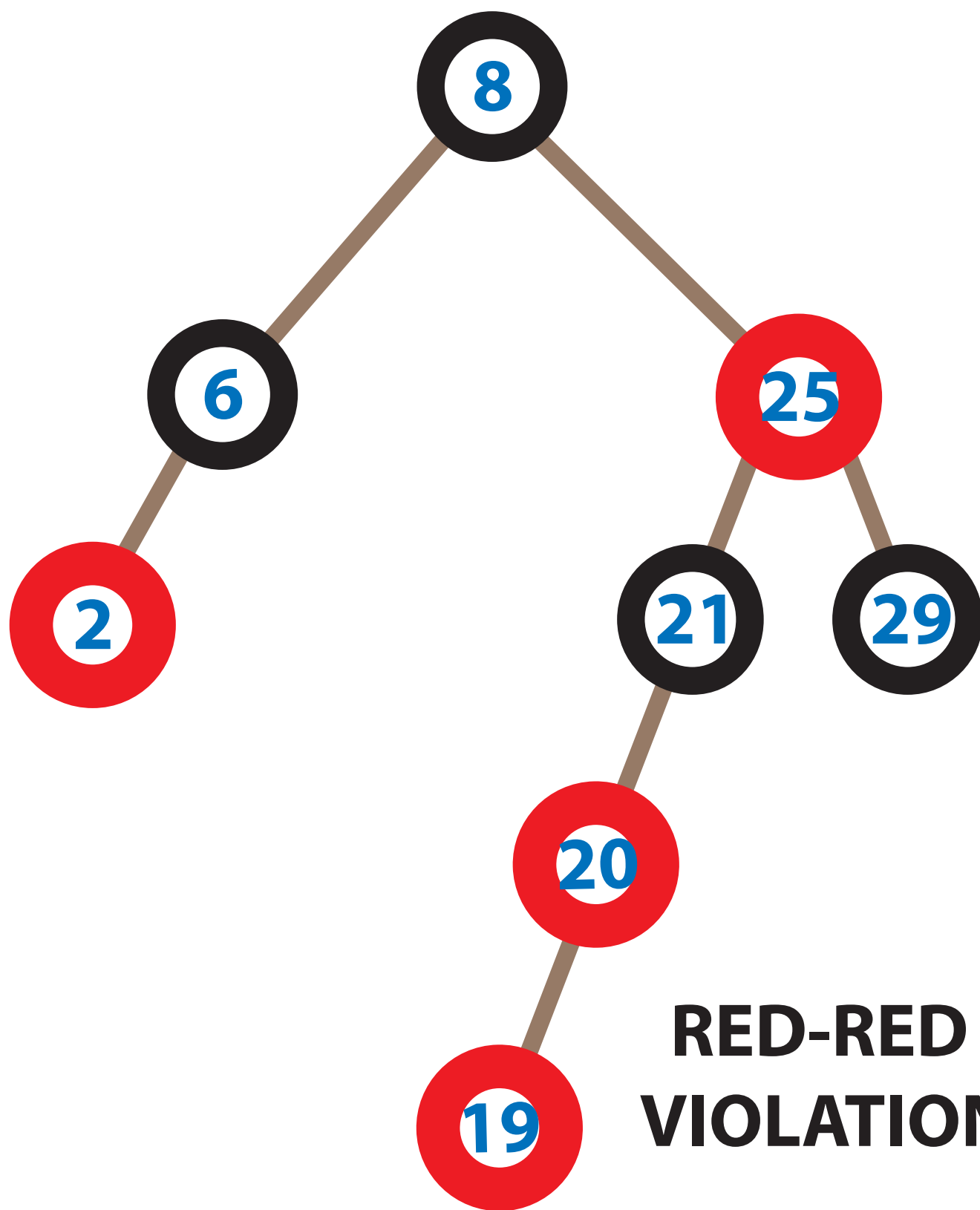
Again, let's insert 20:



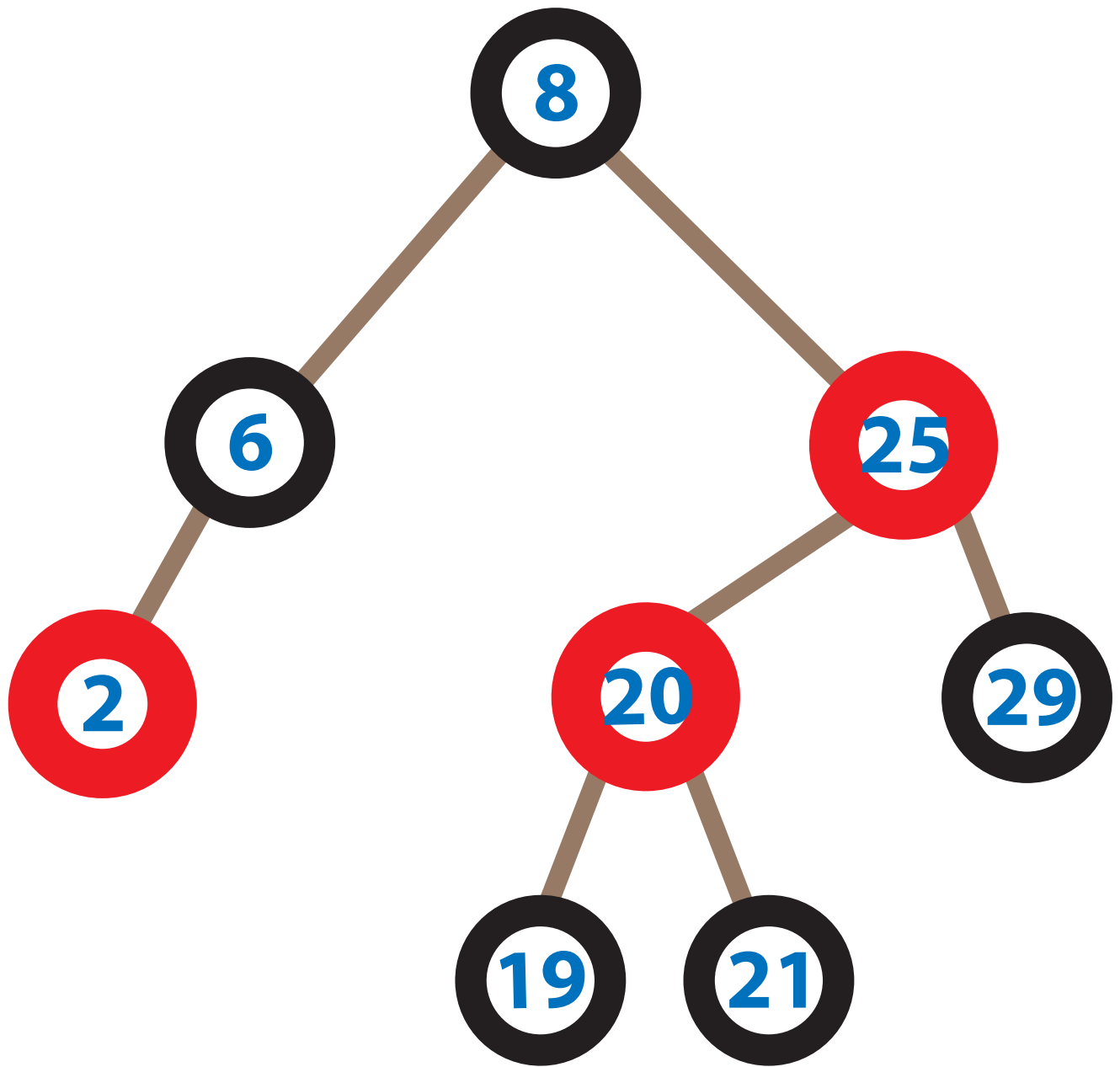
**Insert 20 and color red
(as before)**

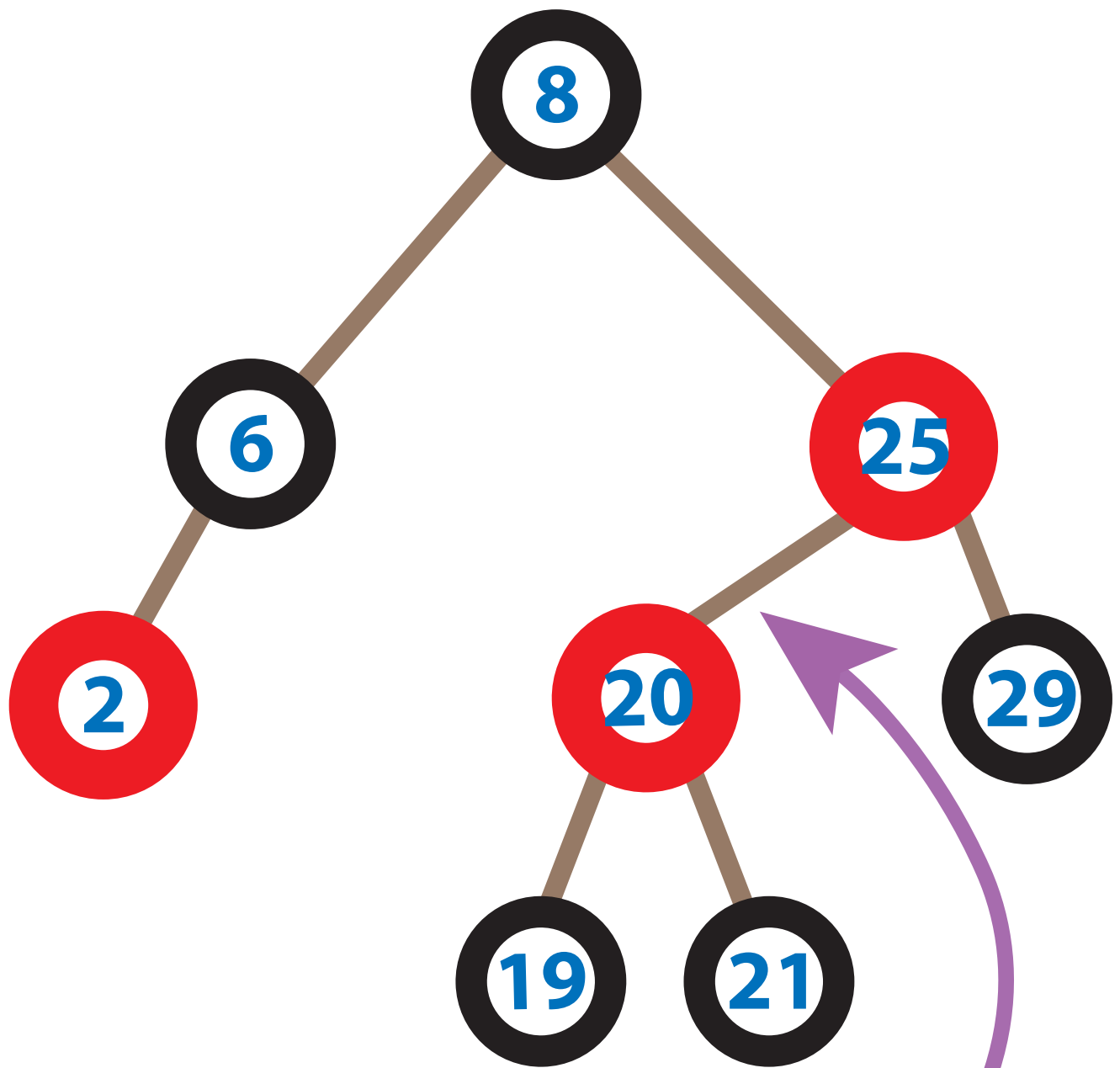
Once again, let's insert 19:





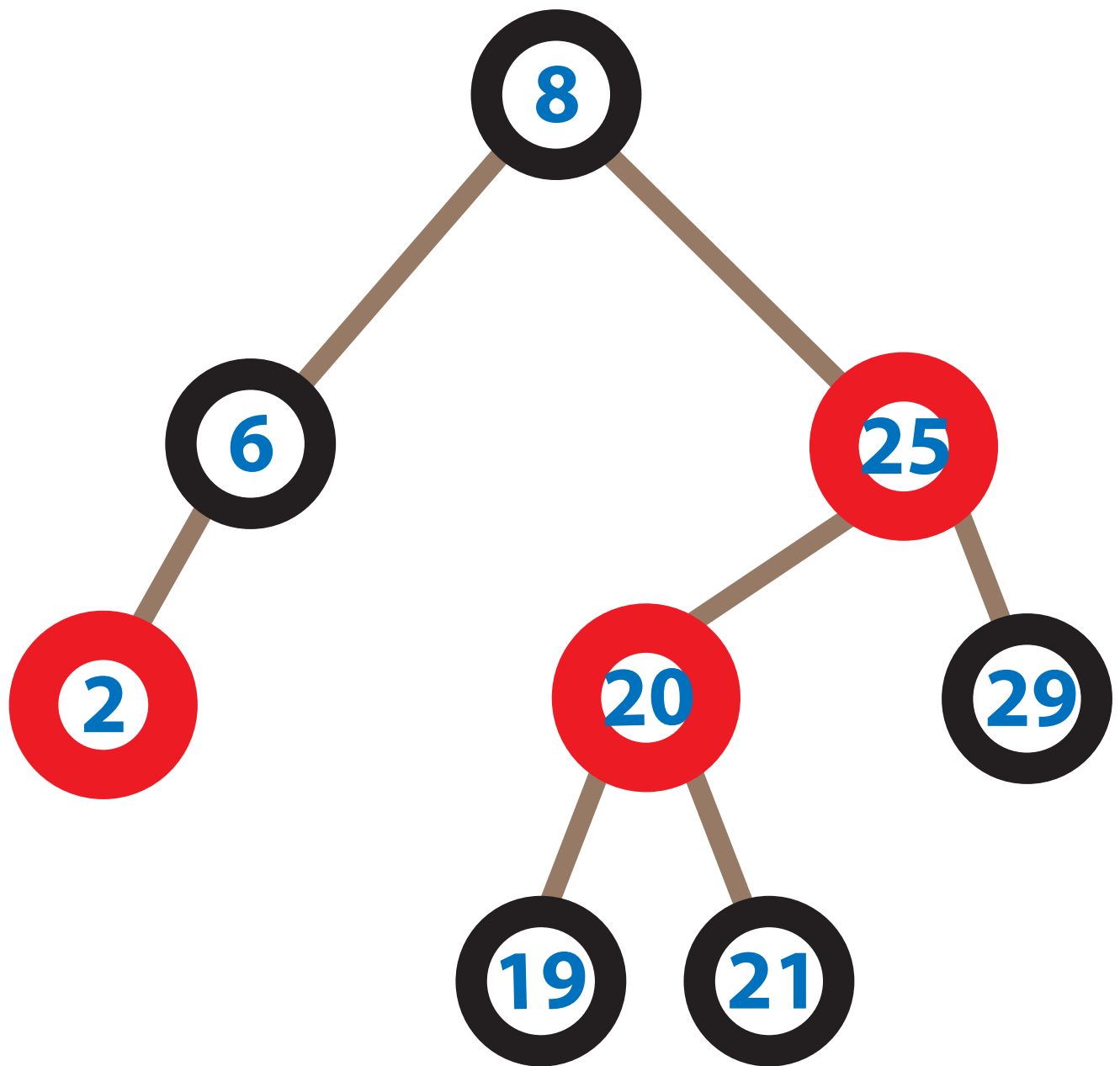
Again, fix with rotation & recoloring:



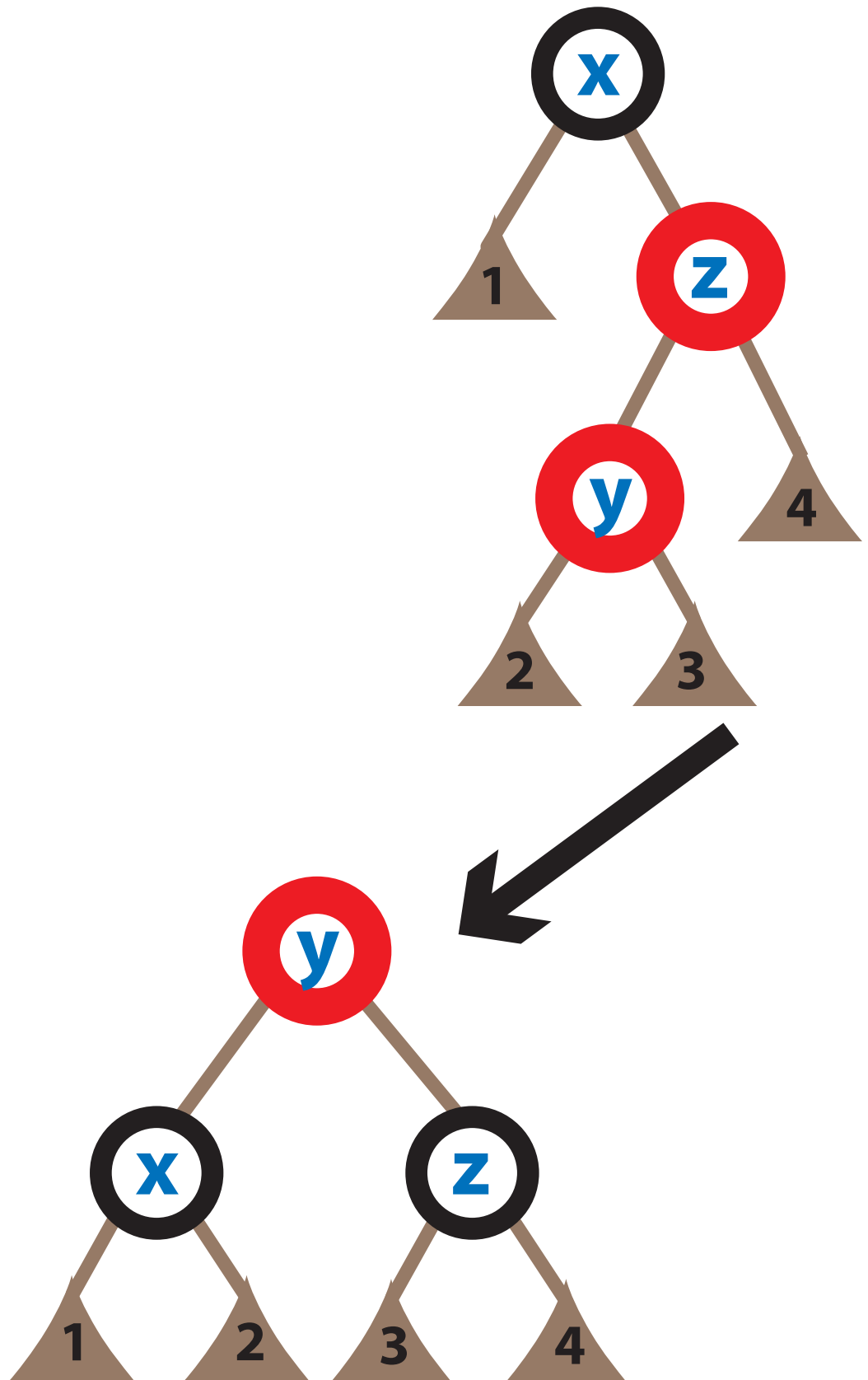


**OH NO! There is a new
RED-RED VIOLATION!**

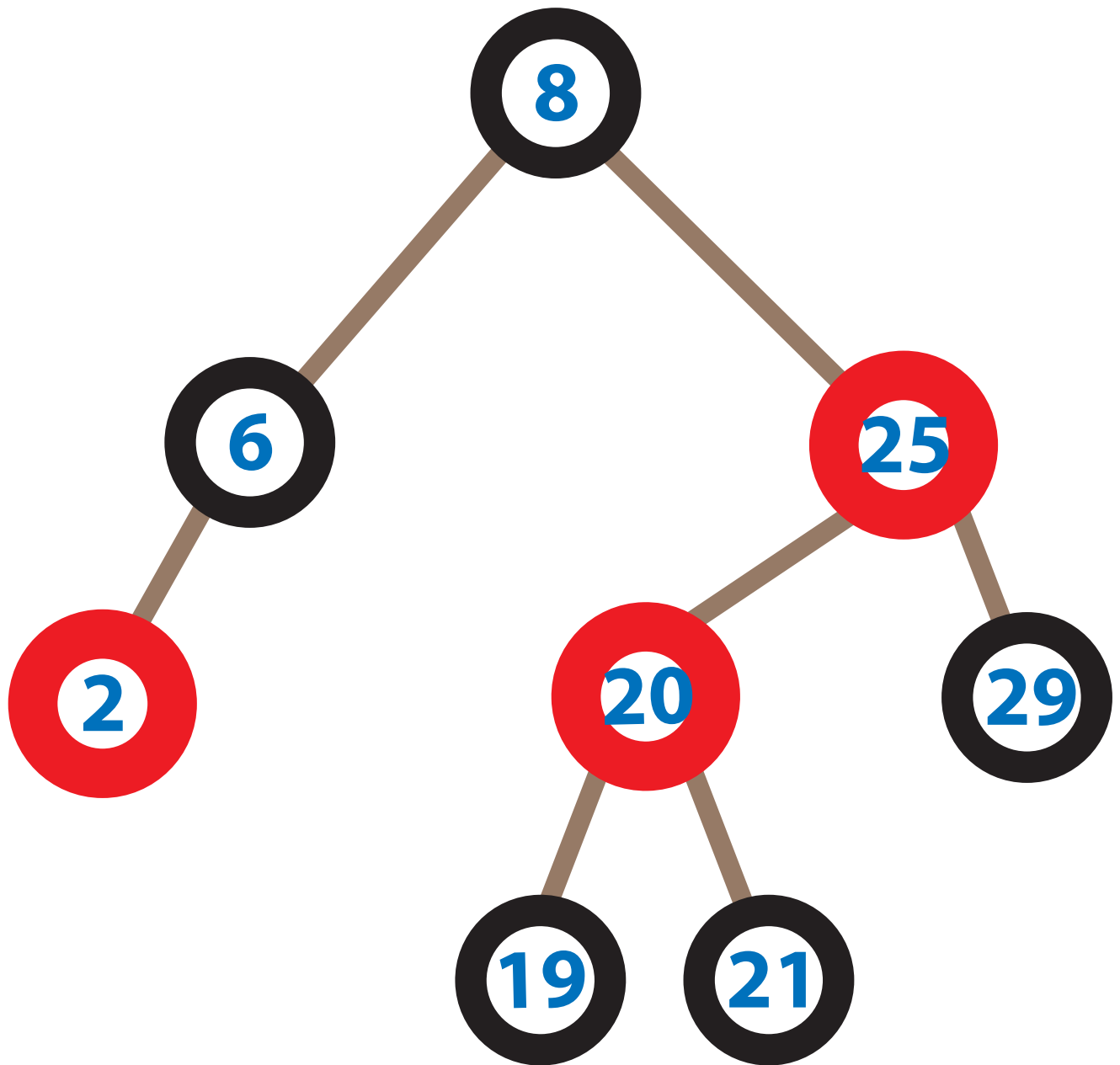
That's OK. We can rotate again ...



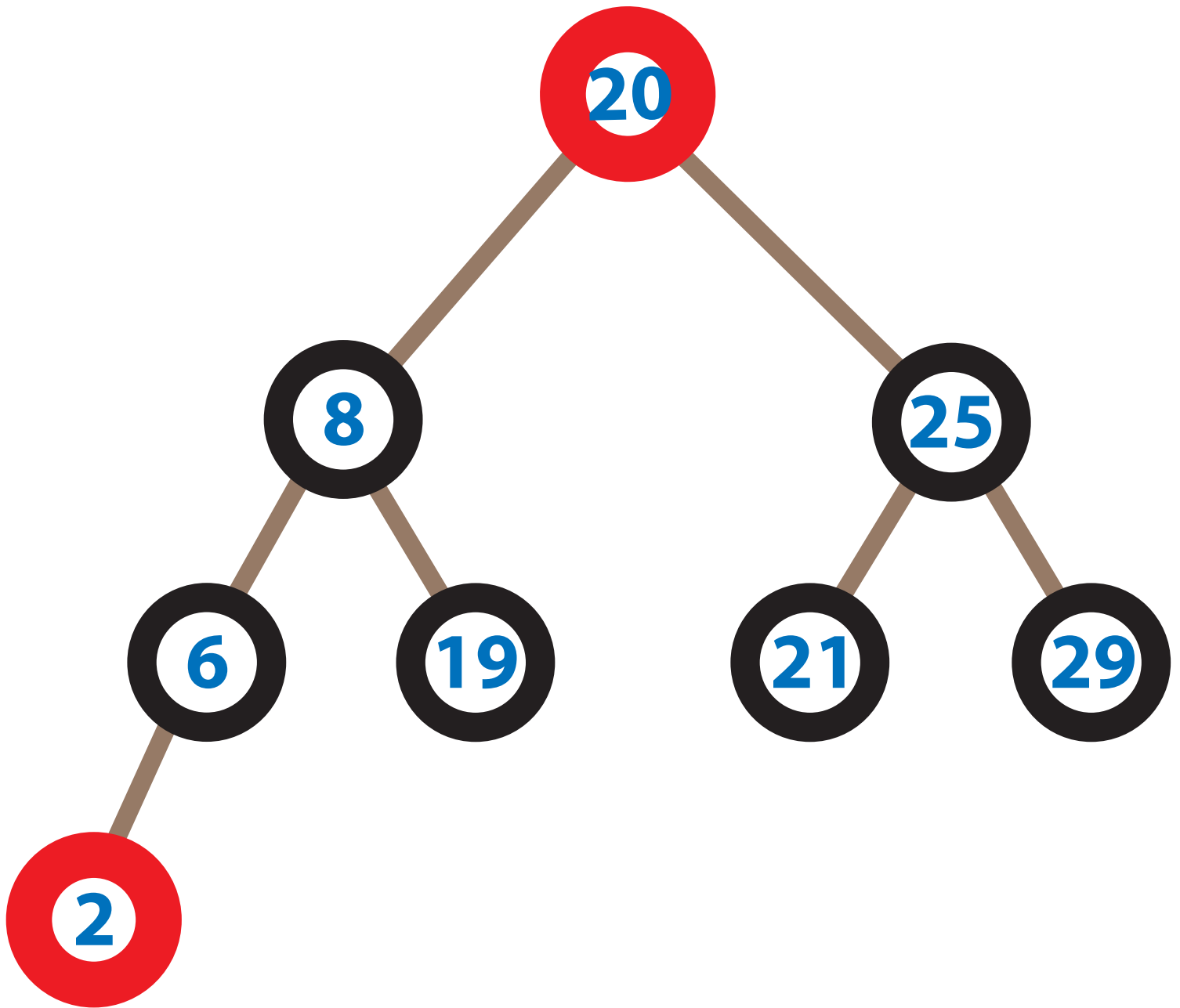
Use this kind of rotation:

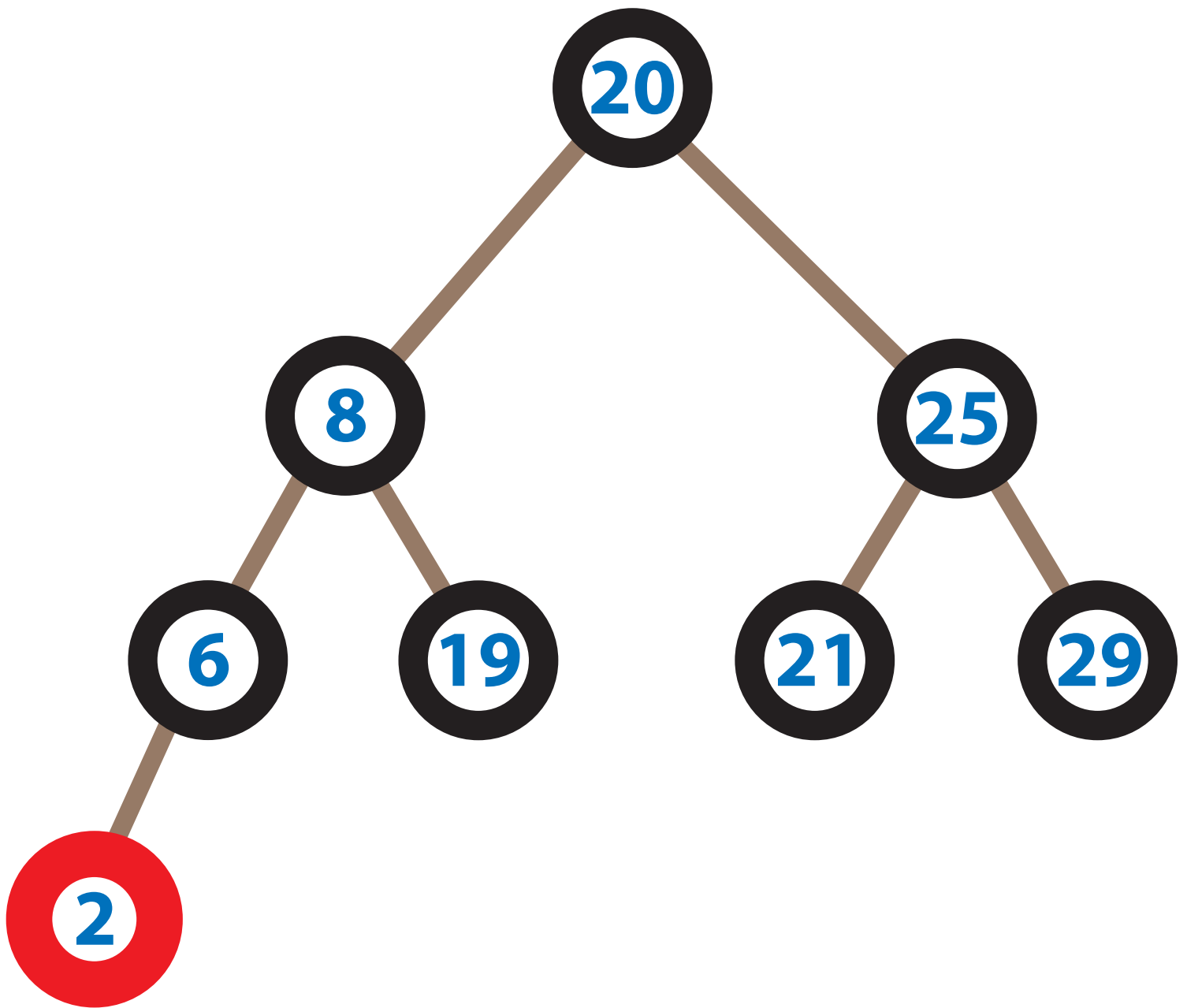


Here's the tree again before rotation:



...giving us this after the rotation:





(It's not necessary, but we can also safely recolor the root black.)

Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

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

(Empty considered black.)

Red Black Tree (RBT) Invariants:

- (1) The tree is **sorted** on the **key** part of the entries.
- (2) The **children** of a **Red** node are **Black**.
- (3) Each node has a well-defined *black height*:
The number of **Black** nodes on *any* path
from the node down to an **Empty** is the same.

Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

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

(Empty considered black.)

Red Black Tree (RBT) Invariants:

- (1) The tree is **sorted** on the **key** part of the entries.
- (2) The **children** of a **Red** node are **Black**.
- (3) Each node has a well-defined *black height*:
The number of **Black** nodes on *any* path
from the node down to an **Empty** is the same.

Almost RBT (ARBT) Invariants:

- (1) and (3) as above.
- (2') Like (2), but: **Red root** may have **one Red** child.

Specs for restoreLeft

(*

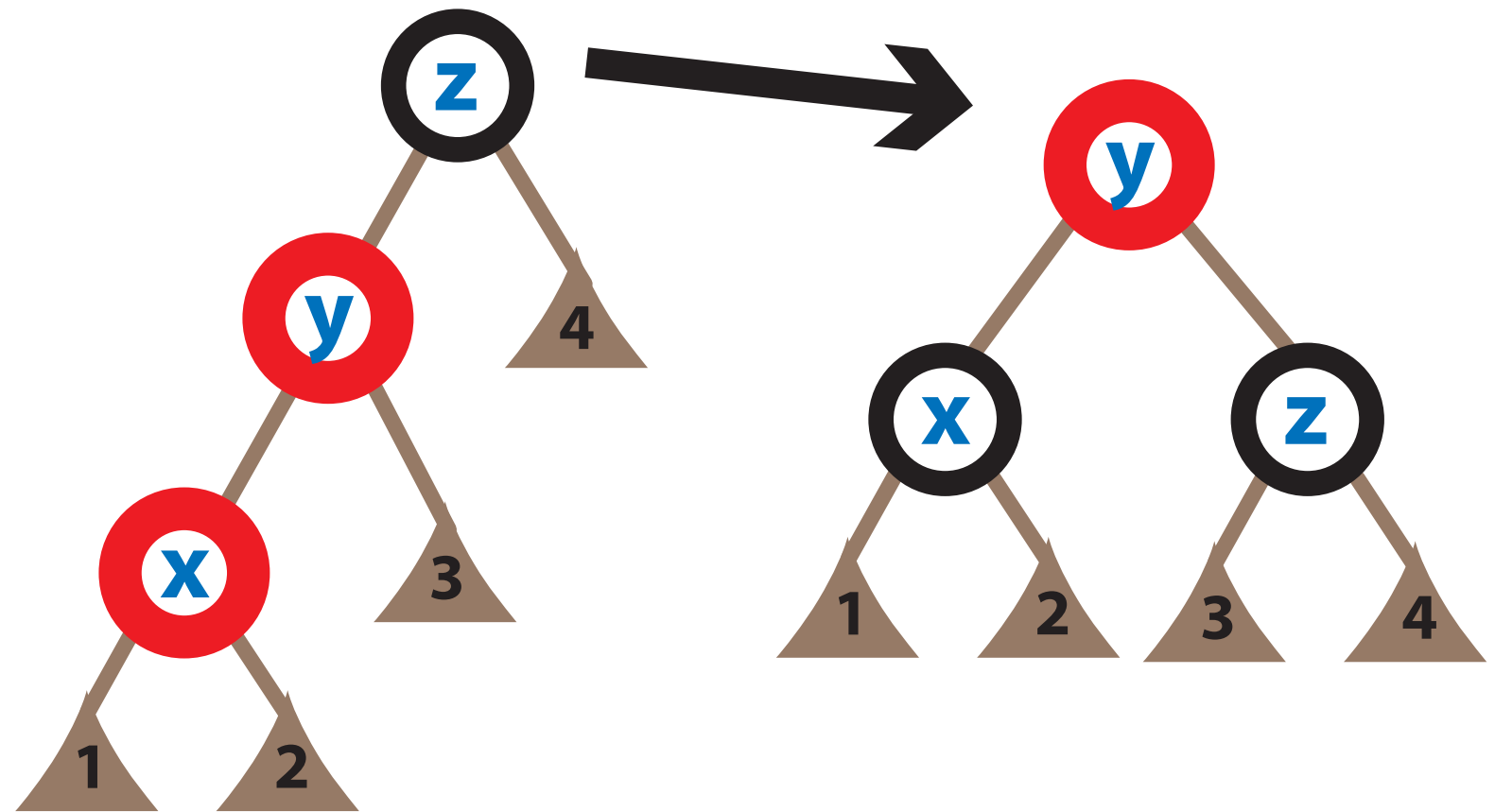
restoreLeft : 'a dict -> 'a dict

REQUIRES: Either **d** is a RBT
or **d's** root is Black,
its left child is an ARBT,
and its right child a RBT.

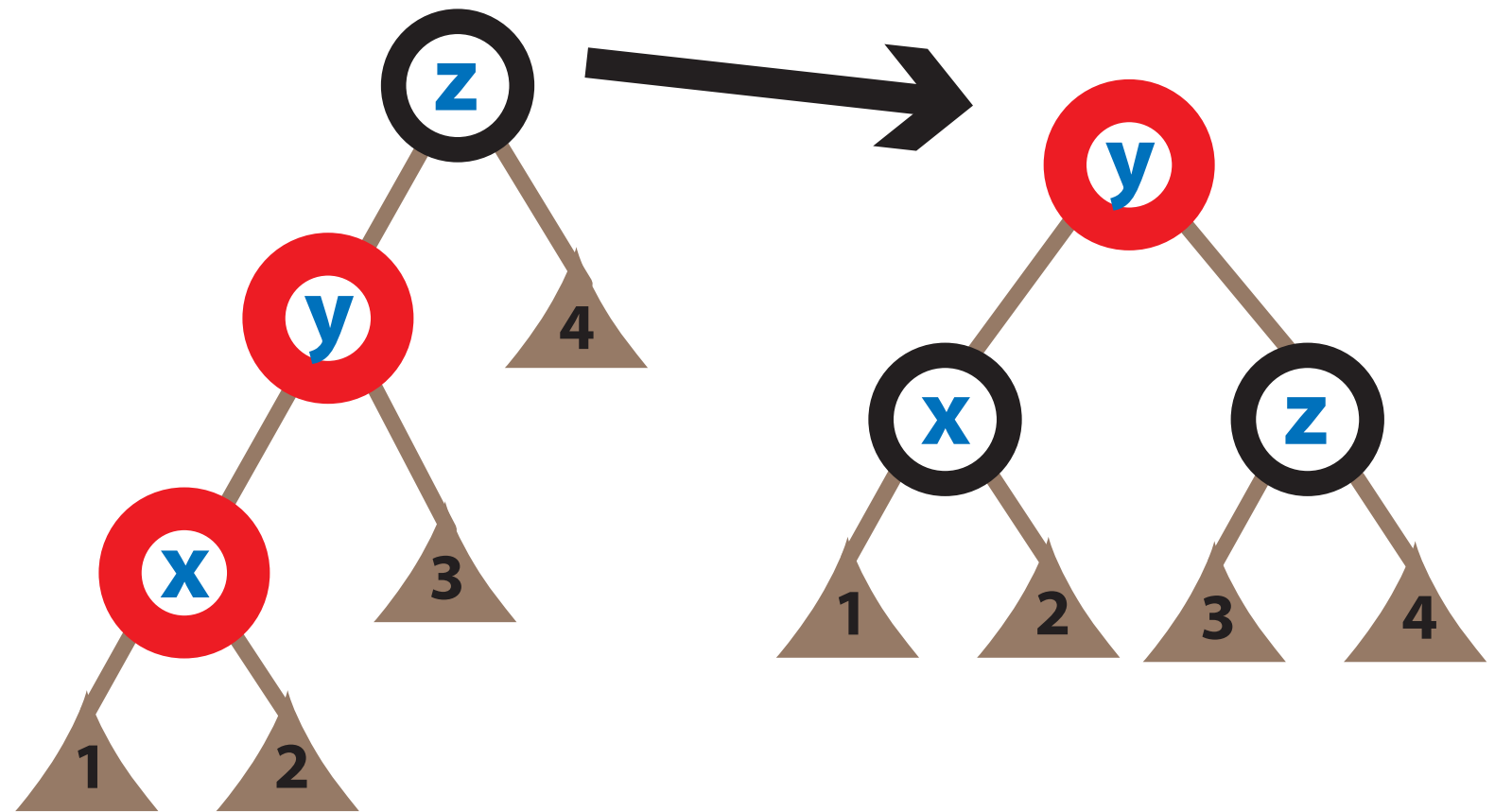
ENSURES: restoreLeft(**d**) is a RBT,
containing exactly the same
entries as **d**, and with the
same black height as **d**.

*)

Picture-based Programming



Picture-based Programming

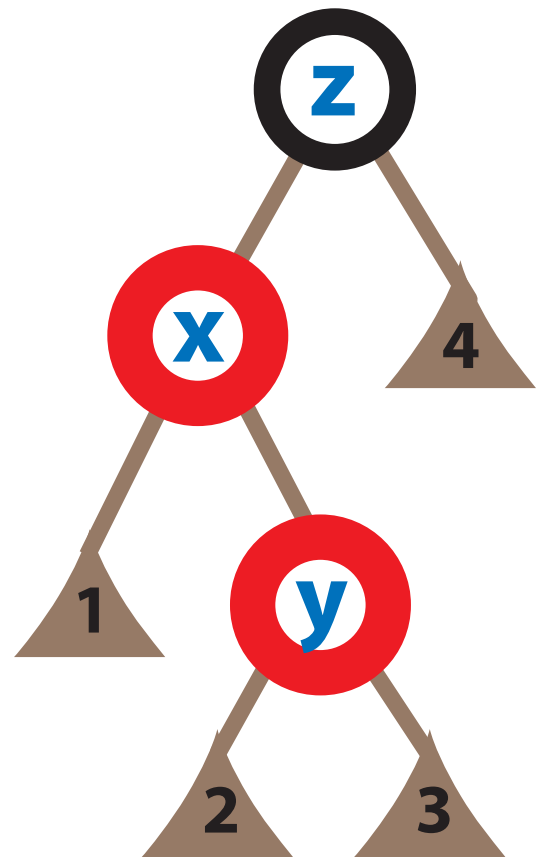
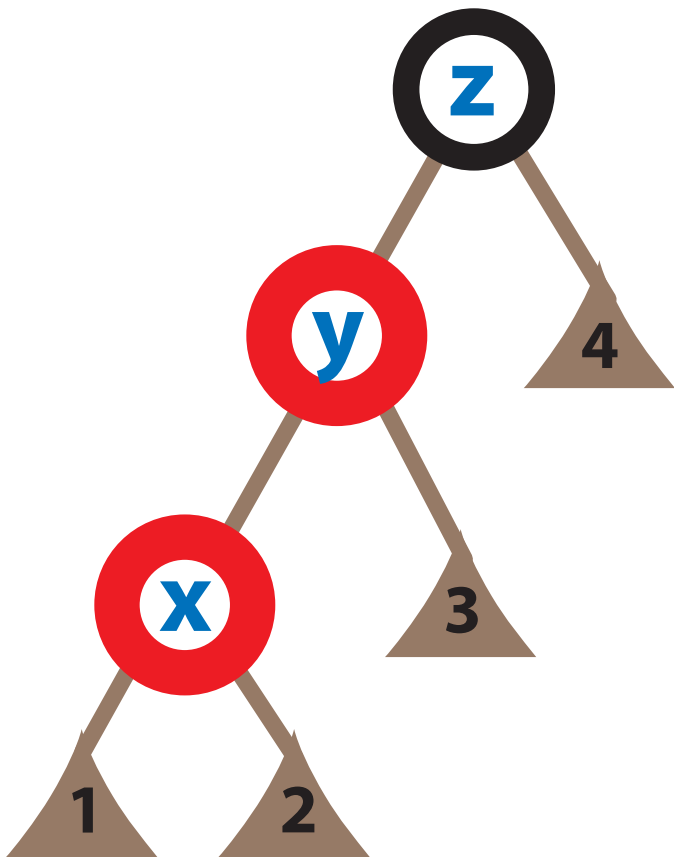


fun

restoreLeft

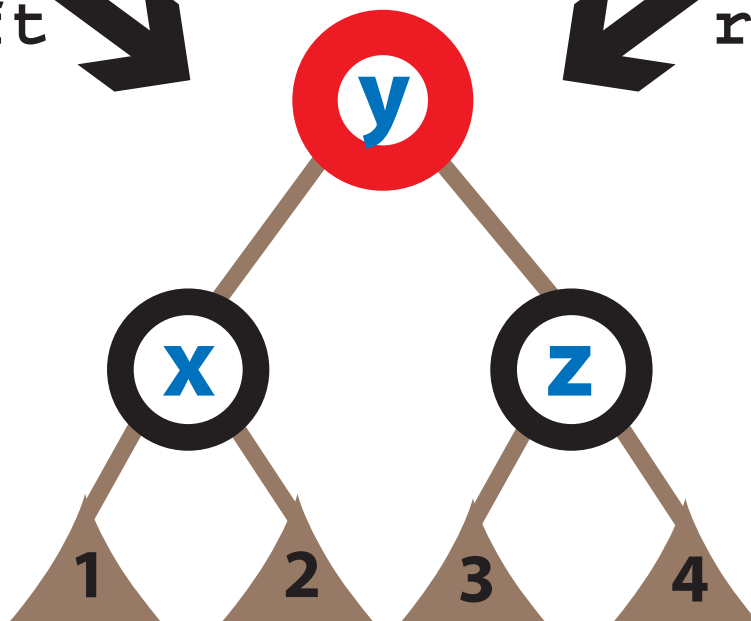
```
(Black (Red (Red (d1, x, d2), y, d3), z, d4)) =  
  Red (Black (d1, x, d2), y, Black (d3, z, d4))
```

2 of the 4 possible kinds of rotations:



restoreLeft
(1st clause)

restoreLeft
(2nd clause)



Code for restoreLeft

```
(*
  restoreLeft : 'a dict -> 'a dict

  REQUIRES: Either d is a RBT
             or d's root is Black,
             its left child is an ARBT,
             and its right child a RBT.

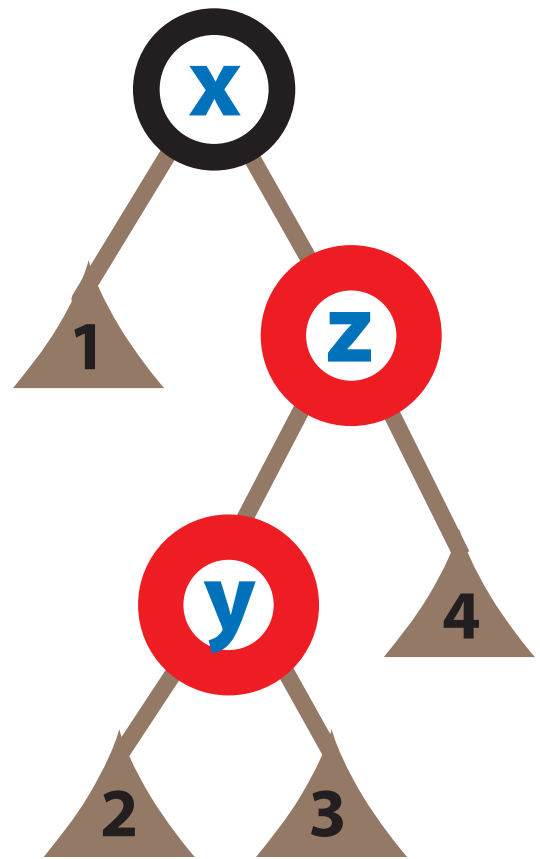
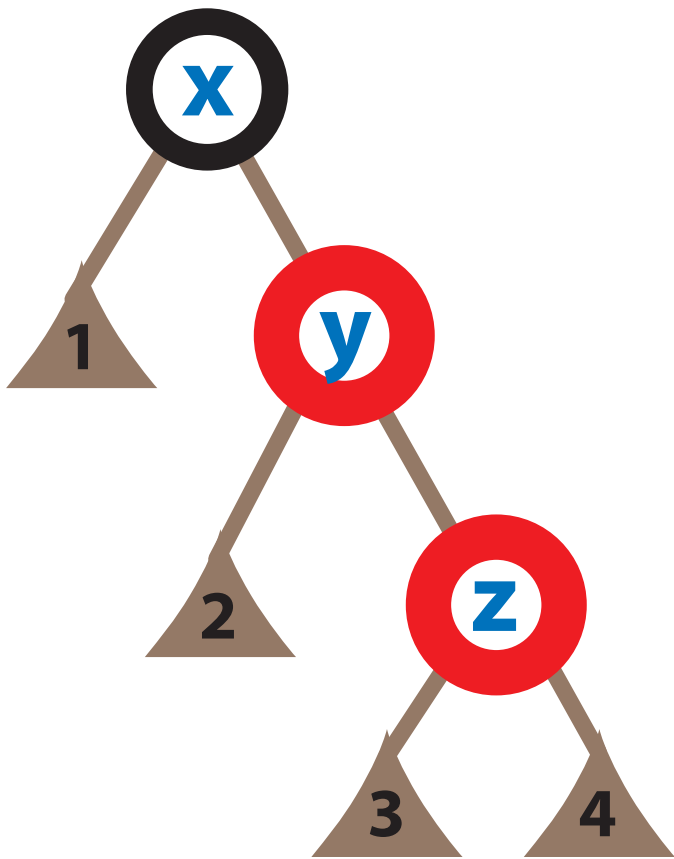
  ENSURES: restoreLeft(d) is a RBT,
            containing exactly the same
            entries as d, and with the
            same black height as d.
*)

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))

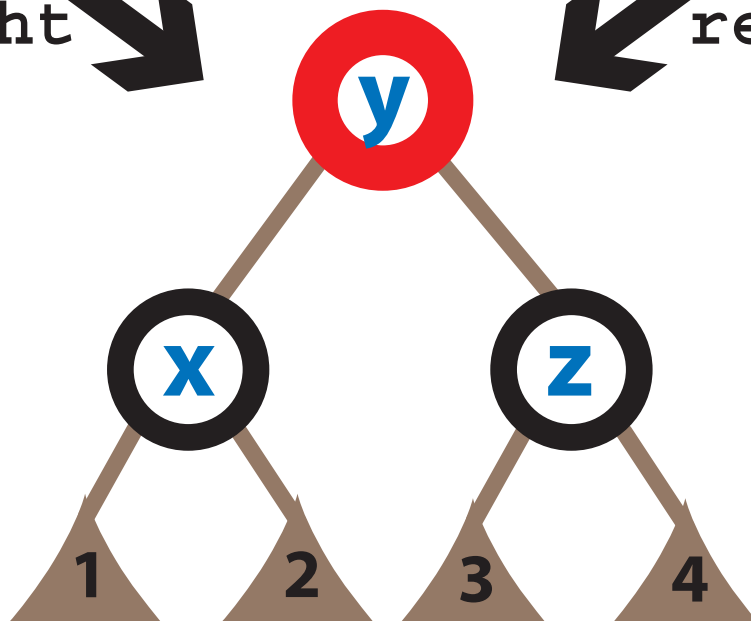
| restoreLeft d = d
```

The other 2 kinds of rotations:



restoreRight
(1st clause)

restoreRight
(2nd clause)



Code for restoreRight

```
(*
  restoreRight : 'a dict -> 'a dict

  REQUIRES: Either d is a RBT
             or d's root is Black,
             its right child is an ARBT,
             and its left child a RBT.

  ENSURES: restoreRight(d) is a RBT,
            containing exactly the same
            entries as d, and with the
            same black height as d.

*)

fun
  restoreRight (Black(d1,x,Red(d2,y,Red(d3,z,d4)))) =
    Red(Black(d1,x,d2), y, Black(d3,z,d4))

| restoreRight (Black(d1,x,Red(Red(d2,y,d3),z,d4))) =
    Red(Black(d1,x,d2), y, Black(d3,z,d4))

| restoreRight d = d
```

Dictionary Signature

```
signature DICT =  
sig  
  type key = string          (* concrete *)  
  type 'a entry = key * 'a   (* concrete *)  
  
  type 'a dict                (* abstract *)  
  
  val empty : 'a dict  
  
  val lookup : 'a dict -> key -> 'a option  
  val insert : 'a dict * 'a entry -> 'a dict  
end
```

Comment:

Our code for
`restoreLeft` and `restoreRight`
will be visible *only inside* our structure,
not to a client.

Specs for `insert` and `ins`

(*

`insert` : 'a dict * 'a entry -> 'a dict

REQUIRES: `d` is a RBT.

ENSURES: `insert(d,e)` is a RBT containing exactly all the entries of `d` plus `e`, with `e` replacing an entry of `d` if the keys are EQUAL.

Locally defined helper function `ins`:

`ins` : 'a dict -> 'a dict

REQUIRES: `d` is a RBT.

ENSURES: `ins(d)` is a tree containing exactly all the entries of `d` plus `e`, with `e` replacing an entry of `d` if the keys are EQUAL.

`ins(d)` has the same black height as `d`.

Moreover, `ins(Black(t))` is a RBT
and `ins(Red(t))` is an ARBT.

*)

Code for insert

```
(* insert : 'a dict * 'a entry -> 'a dict
    REQUIRES and ENSURES RBT. *)
```

```
fun insert (d , e as (k, v)) =
  let
    fun ins ... (will write shortly)
  in
    (case ins(d) of
      Red(t as (Red(_),_,_)) => Black(t)
    | Red(t as (_,_,Red(_))) => Black(t)
    | d' => d')
  end
```

Code for insert

```
(* insert : 'a dict * 'a entry -> 'a dict
   REQUIRES and ENSURES RBT. *)
```

```
fun insert (d , e as (k, v)) =
  let
    fun ins ... (will write shortly)
  in
    (case ins(d) of
      Red(t as (Red(_,_,_))) => Black(t)
    | Red(t as (_,_,Red(_))) => Black(t)
    | d' => d')
  end
```

recall the keyword **as** means
layered pattern matching

Code for insert

```
(* insert : 'a dict * 'a entry -> 'a dict
    REQUIRES and ENSURES RBT. *)
```

```
fun insert (d , e as (k, v)) =
  let
    fun ins ... (will write shortly)
  in
    (case ins(d) of
      Red(t as (Red(_,_,_))) => Black(t)
    | Red(t as (_,_,Red(_))) => Black(t)
    | d' => d')
  end
```

Here is an acceptable alternate for the `case` :

```
(case ins(d) of
  Red(t) => Black(t)
| d' => d')
```

Code for `ins`

```
(* ins : 'a dict -> 'a dict
```

```
REQUIRES: d is RBT.
```

```
ENSURES: ins(Black(t)) is RBT,
```

```
         ins(Red(t)) is ARBT.
```

```
Recall:   e as (k,v) is in scope. *)
```

```
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)    (* 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)  
  | GREATER => Red(ℓ,e',ins(r)))
```

Code for ins

```
(* ins : 'a dict -> 'a dict
   REQUIRES: d is RBT.
   ENSURES: ins(Black(t)) is RBT,
            ins(Red(t)) is ARBT.
   Recall:  e as (k,v) is in scope. *)

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)    (* 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)
  | GREATER => Red(l,e',ins(r)))
```

Why do we not call `restoreLeft`
or `restoreRight` here ?

Code for lookup

```
(* 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  
  end
```

Code for lookup

```
(* 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  
  end
```

mutual
recursion

Sample Usage

Suppose we have implemented the previous code as:

```
structure RBT :> DICT = struct ... end
```

Now consider:

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

Then ML will print:

```
val r1 = - : int RBT.dict
```

 because of opaque ascription

 because we put in an integer value

Now create the following:

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

```
val look2 = RBT.lookup r2
```

Then `look2 : RBT.key -> int option`

```
look2 "a" ==> SOME 1
```

```
look2 "c" ==> NONE
```

That is all.

Midterm on Thursday.

Next week we will discuss cost graphs and an abstract datatype designed for writing parallel code (SEQUENCES).