Modules III

15-150

Lecture 19: November 14, 2024

Stephanie Balzer
Carnegie Mellon University

Recap

Recap

SML modules facilitate abstraction:

Recap

SML modules facilitate abstraction:

- Specification: **signature**.
- Implementation: **structure**.

Recap

SML modules facilitate abstraction:

- Specification: **signature**.
- Implementation: **structure**.

SML modules allow us to control the “flow of information”:

Recap

SML modules facilitate abstraction:

- Specification: **signature**.
- Implementation: **structure**.

SML modules allow us to control the “flow of information”:

- Structures can **hide** auxiliary, implementation-specific components, not specified by signature.

Recap

SML modules facilitate abstraction:

- Specification: **signature**.
- Implementation: **structure**.

SML modules allow us to control the “flow of information”:

- Structures can **hide** auxiliary, implementation-specific components, not specified by signature.
- **Transparent ascription**: for undefined type specified in signature, **representation type** chosen by structure is **revealed**.

Recap

SML modules facilitate abstraction:

- Specification: **signature**.
- Implementation: **structure**.

SML modules allow us to control the “flow of information”:

- Structures can **hide** auxiliary, implementation-specific components, not specified by signature.
- **Transparent ascription**: for undefined type specified in signature, **representation type** chosen by structure is **revealed**.
- **Opaque ascription**: for undefined type specified in signature, **representation type** chosen by structure is **hidden**.

Recap

Recap

Type classes and functors:

Recap

Type classes and functors:

Prescriptive signatures exhaustively specify a type's operations, typically using **opaque** ascription.

Recap

Type classes and functors:

Prescriptive signatures exhaustively specify a type's operations, typically using **opaque** ascription.

Descriptive signatures (aka type classes) expose a type parameter's operations, typically using **transparent** ascription.

Recap

Type classes and functors:

- **Prescriptive** signatures exhaustively specify a type's operations, typically using **opaque** ascription.
- **Descriptive** signatures (aka type classes) expose a type parameter's operations, typically using **transparent** ascription.
- A **functor** creates a structure, given a structure as an argument.

Recap

Type classes and functors:

- **Prescriptive** signatures exhaustively specify a type's operations, typically using **opaque** ascription.
- **Descriptive** signatures (aka type classes) expose a type parameter's operations, typically using **transparent** ascription.
- A **functor** creates a structure, given a structure as an argument.
    - Functor arguments are typically type classes to prevent code redundancy.

Recap

Type classes and functors:

- **Prescriptive** signatures exhaustively specify a type's operations, typically using **opaque** ascription.
- **Descriptive** signatures (aka type classes) expose a type parameter's operations, typically using **transparent** ascription.
- A **functor** creates a structure, given a structure as an argument.
  - Functor arguments are typically type classes to prevent code redundancy.

Representation invariants:

Recap

Type classes and functors:

- **Prescriptive** signatures exhaustively specify a type's operations, typically using **opaque** ascription.
- **Descriptive** signatures (aka type classes) expose a type parameter's operations, typically using **transparent** ascription.
- A **functor** creates a structure, given a structure as an argument.
  - Functor arguments are typically type classes to prevent code redundancy.

Representation invariants:

- Hidden consistency condition enforced by structure.

Today

Today

A closer look at representation invariants:

Today

A closer look at representation invariants:

Some code may necessarily violate the invariant.

Today

A closer look at representation invariants:

- Some code may necessarily violate the invariant.
- Localize violation and characterize with weaker invariant.

Today

A closer look at representation invariants:

- Some code may necessarily violate the invariant.
- Localize violation and characterize with weaker invariant.
- Complement with code that re-establishes stronger invariant, when weaker invariant holds.

Today

A closer look at representation invariants:

- Some code may necessarily violate the invariant.
- Localize violation and characterize with weaker invariant.
- Complement with code that re-establishes stronger invariant, when weaker invariant holds.

We'll explain these ideas on an example, further illustrating:

Today

A closer look at representation invariants:

- Some code may necessarily violate the invariant.
- Localize violation and characterize with weaker invariant.
- Complement with code that re-establishes stronger invariant, when weaker invariant holds.

We'll explain these ideas on an example, further illustrating:

- A functional implementation of balanced trees.

Today

A closer look at representation invariants:

- Some code may necessarily violate the invariant.
- Localize violation and characterize with weaker invariant.
- Complement with code that re-establishes stronger invariant, when weaker invariant holds.

We'll explain these ideas on an example, further illustrating:

- A functional implementation of balanced trees.
- "Picture-guided programming" thanks to pattern matching.

Let's reconsider our dictionary

Let's reconsider our dictionary

```
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
```

Let's reconsider our dictionary

```
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 time we implemented our dictionary as a binary search tree:

Let's reconsider our dictionary

```
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 time we implemented our dictionary as a binary search tree:

```
structure BST : DICT = ...
```

Let's reconsider our dictionary

```
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 time we implemented our dictionary as a binary search tree:

```
structure BST : DICT = ...
```

Representation invariant: tree is sorted on key (no duplicate keys)

Let's reconsider our dictionary

Last time we implemented our dictionary as a binary search tree:

```
structure BST : DICT = ...
```

Representation invariant: tree is sorted on key (no duplicate keys)

Let's reconsider our dictionary

Last time we implemented our dictionary as a binary search tree:

```
structure BST : DICT = ...
```

Representation invariant: tree is sorted on key (no duplicate keys)

Problem: insertion may result in an unbalanced tree and thus make lookup slow.

Let's reconsider our dictionary

Last time we implemented our dictionary as a binary search tree:

```
structure BST : DICT = ...
```

→ Representation invariant: tree is sorted on key (no duplicate keys)

Problem: insertion may result in an unbalanced tree and thus make lookup slow.

→ Implement dictionary as a red black tree!

Let's reconsider our dictionary

Last time we implemented our dictionary as a binary search tree:

```
structure BST : DICT = ...
```

Representation invariant: tree is sorted on key (no duplicate keys)

Problem: insertion may result in an unbalanced tree and thus make lookup slow.

Implement dictionary as a red black tree!

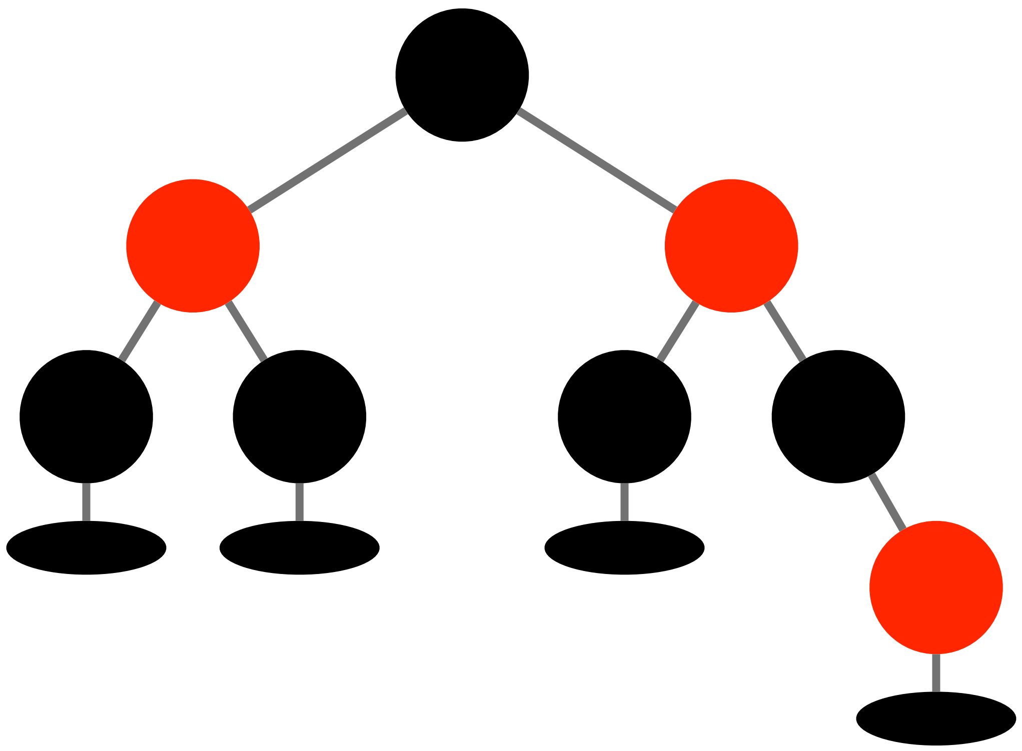
```
datatype 'a dict =
  Empty
| Red of 'a dict * 'a entry * 'a dict
| Black of 'a dict * 'a entry * 'a dict
```

Red Black Trees

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

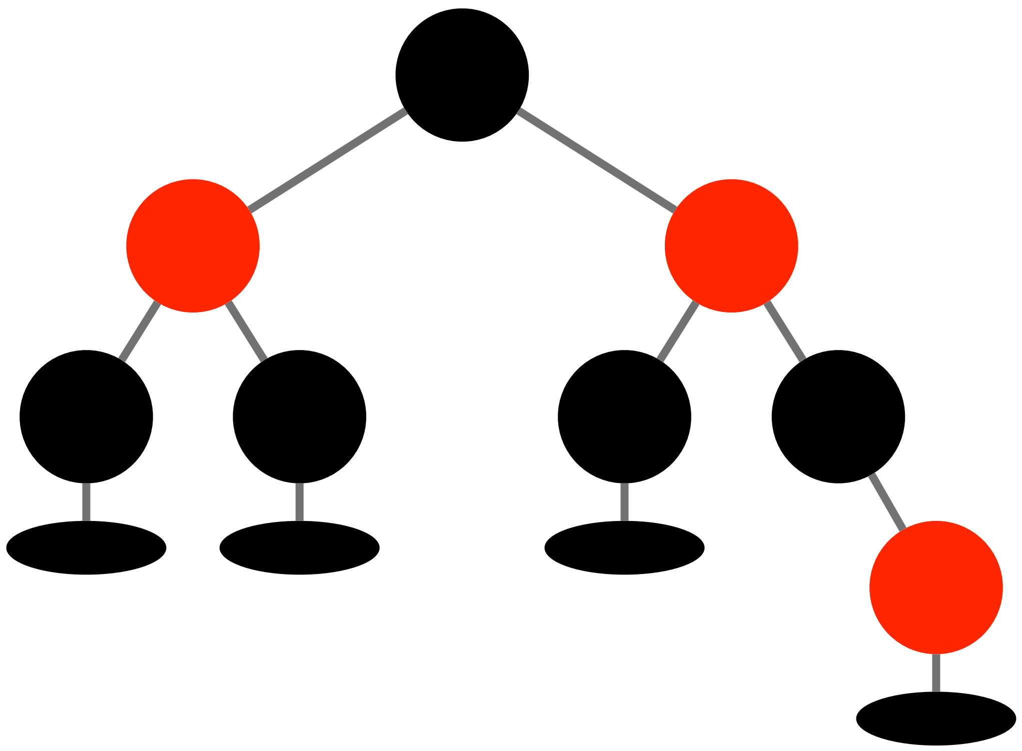
Red Black Trees

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



Red Black Trees

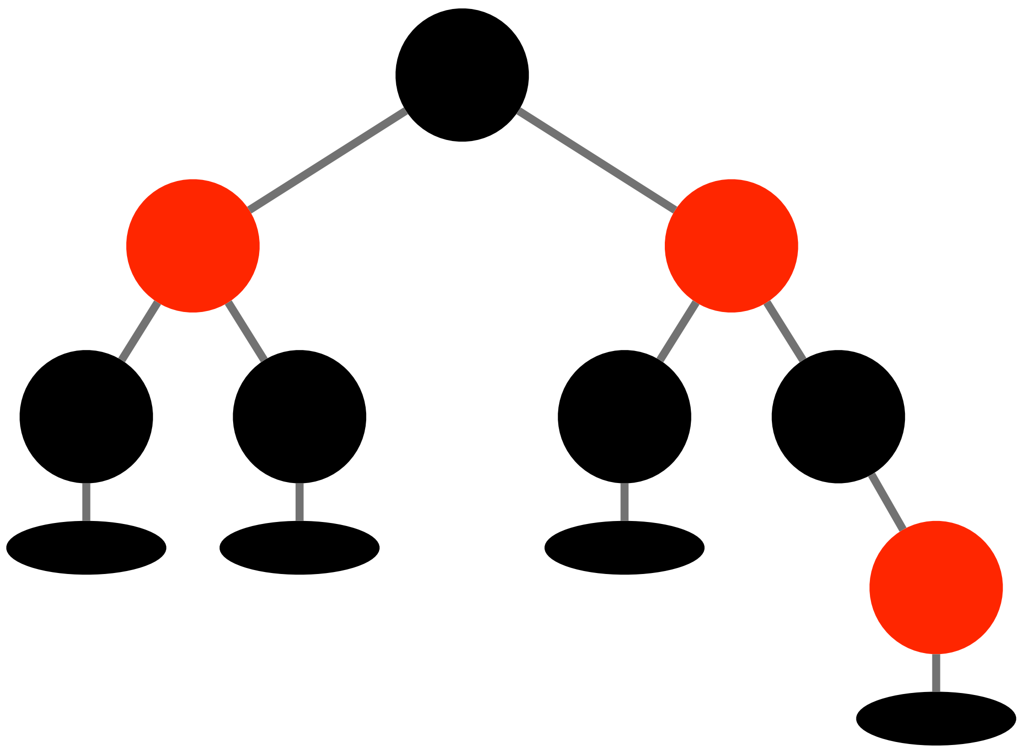
```
datatype 'a dict =
   Empty
 | Red of 'a dict * 'a entry * 'a dict
 | Black of 'a dict * 'a entry * 'a dict
```



Red Black Trees

code shown
monochromatically

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

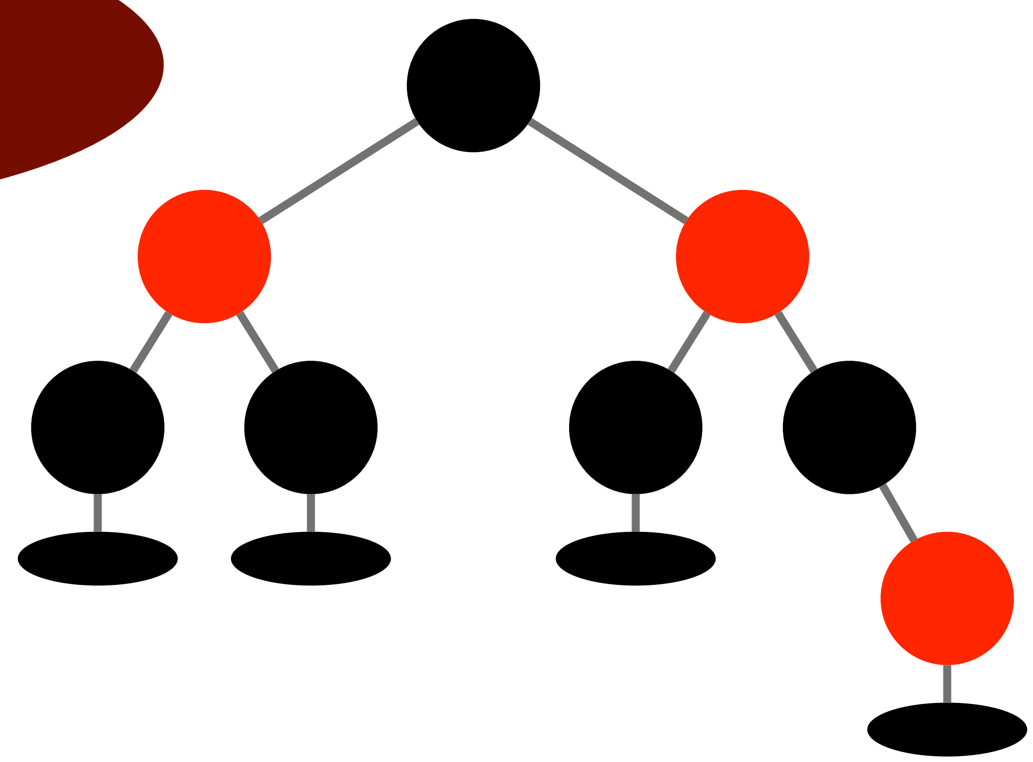


Red Black Trees

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

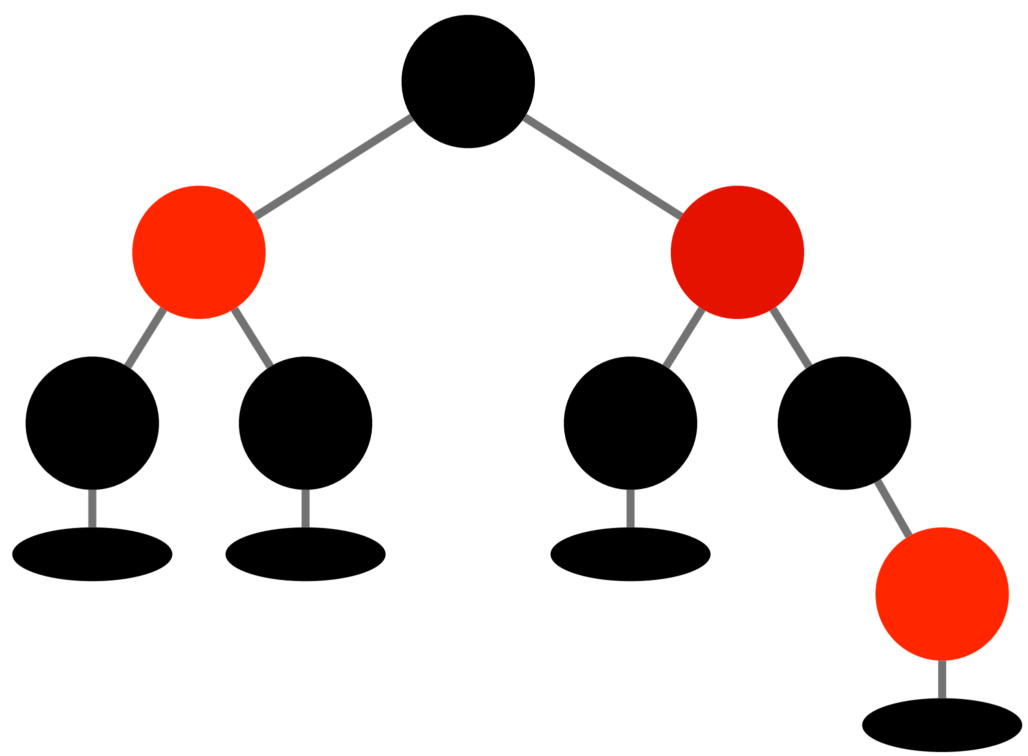
code shown monochromatically

colors used for node coloring



Red Black Trees

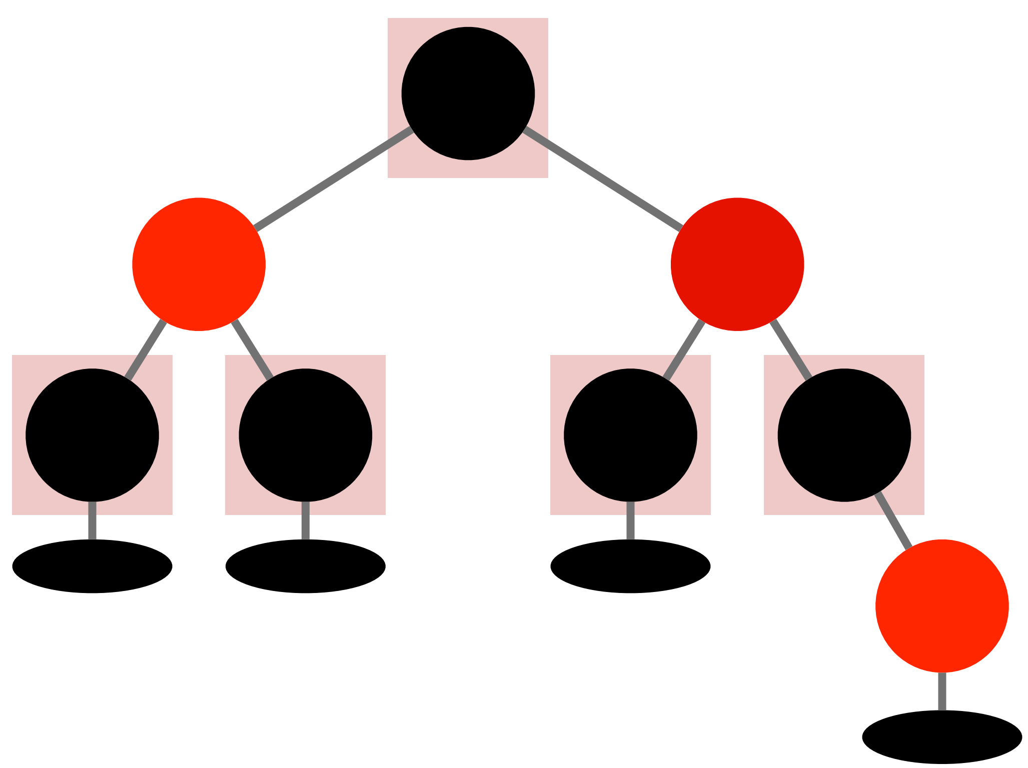
```
datatype 'a dict =
  Empty
| Red of 'a dict * 'a entry * 'a dict
| Black of 'a dict * 'a entry * 'a dict
```



Red Black Trees

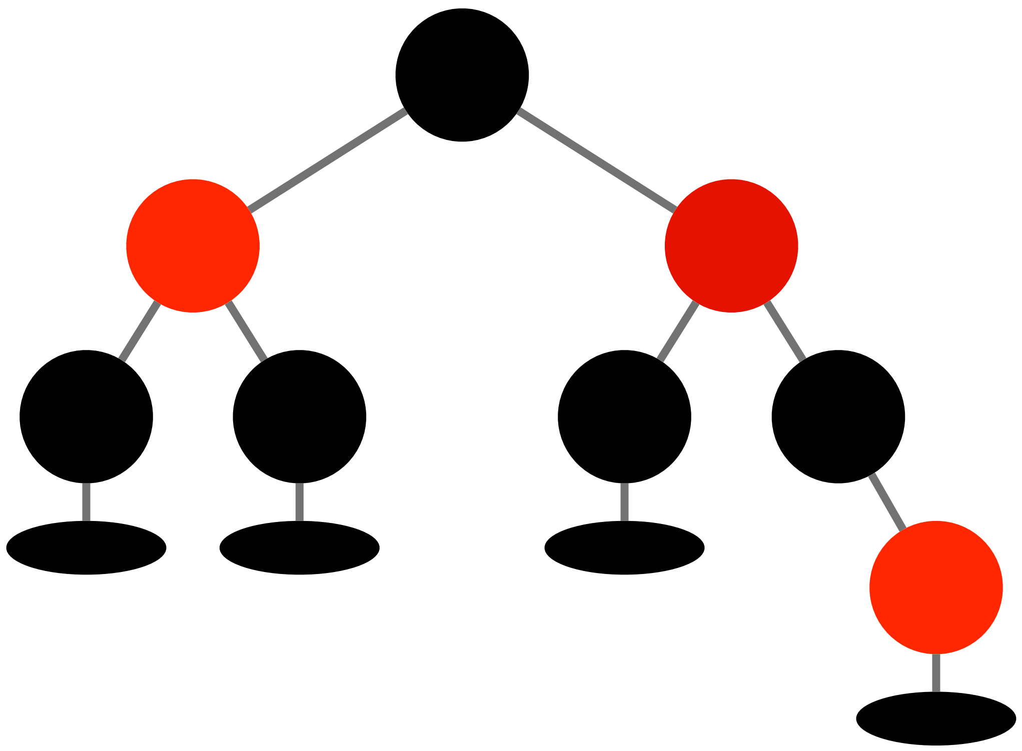
```
datatype 'a dict =
  Empty
| Red of 'a dict * 'a entry * 'a dict
| Black of 'a dict * 'a entry * 'a dict
```

black nodes



Red Black Trees

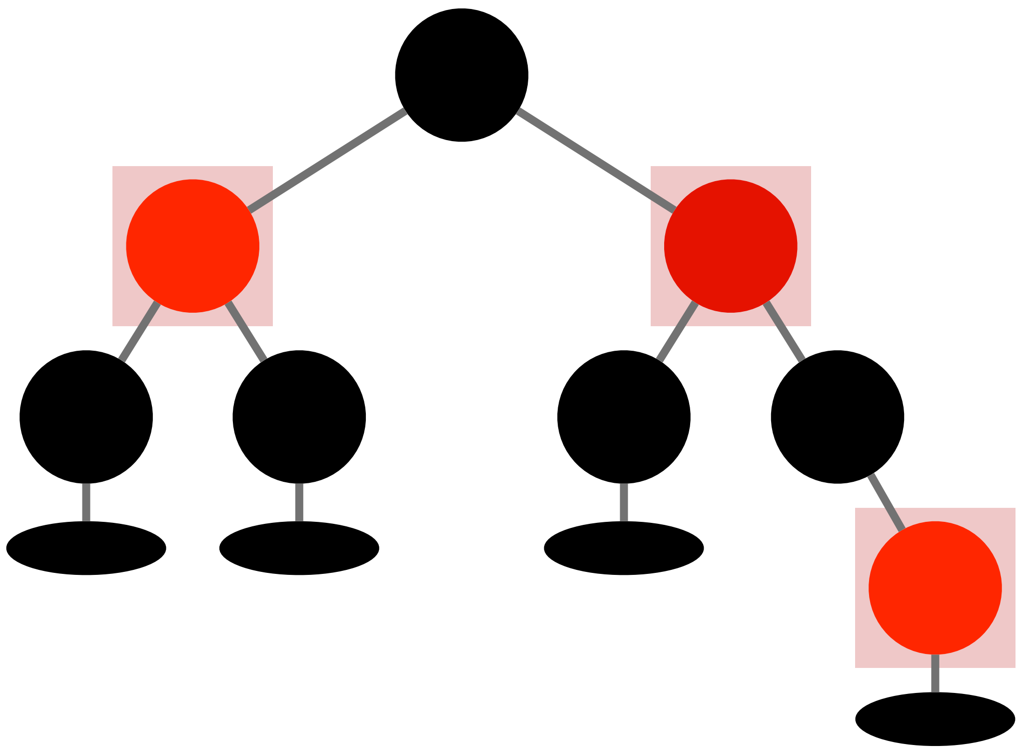
```
datatype 'a dict =
  Empty
| Red of 'a dict * 'a entry * 'a dict
| Black of 'a dict * 'a entry * 'a dict
```



Red Black Trees

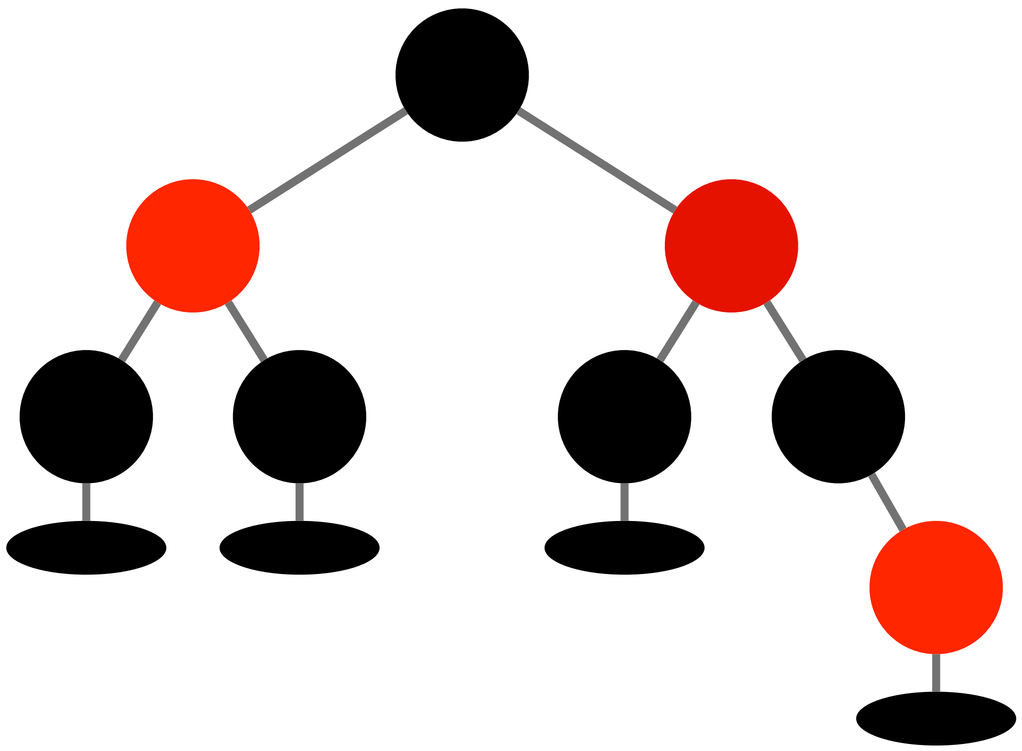
```
datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict
```

red nodes



Red Black Trees

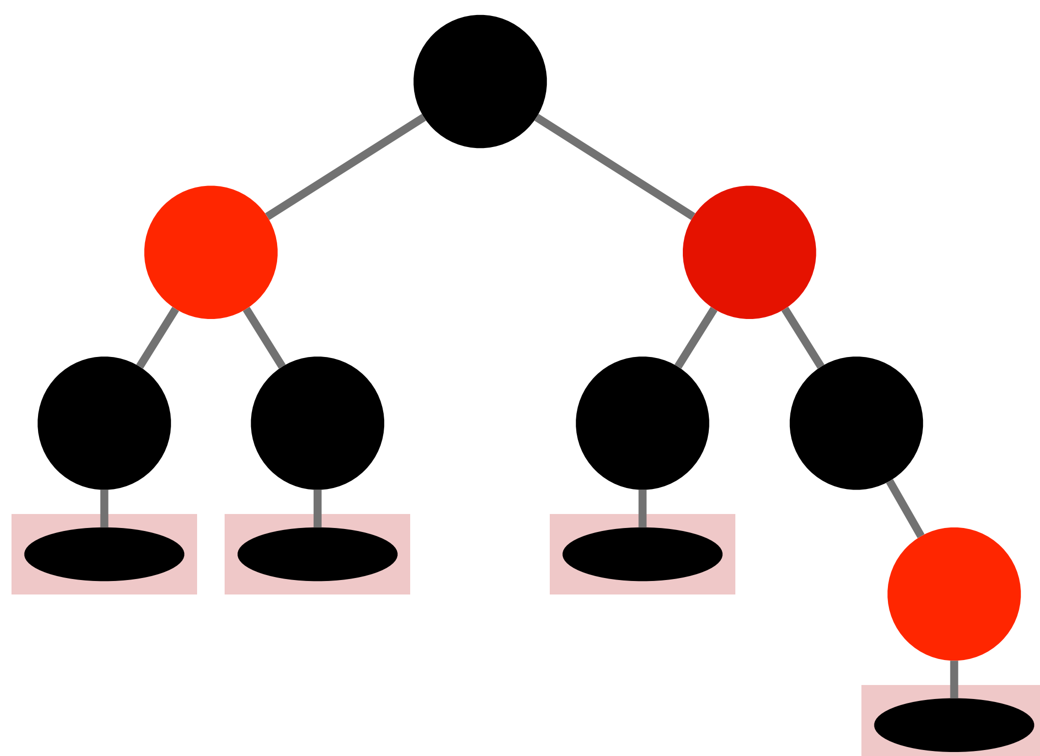
```
datatype 'a dict =
  Empty
| Red of 'a dict * 'a entry * 'a dict
| Black of 'a dict * 'a entry * 'a dict
```



Red Black Trees

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

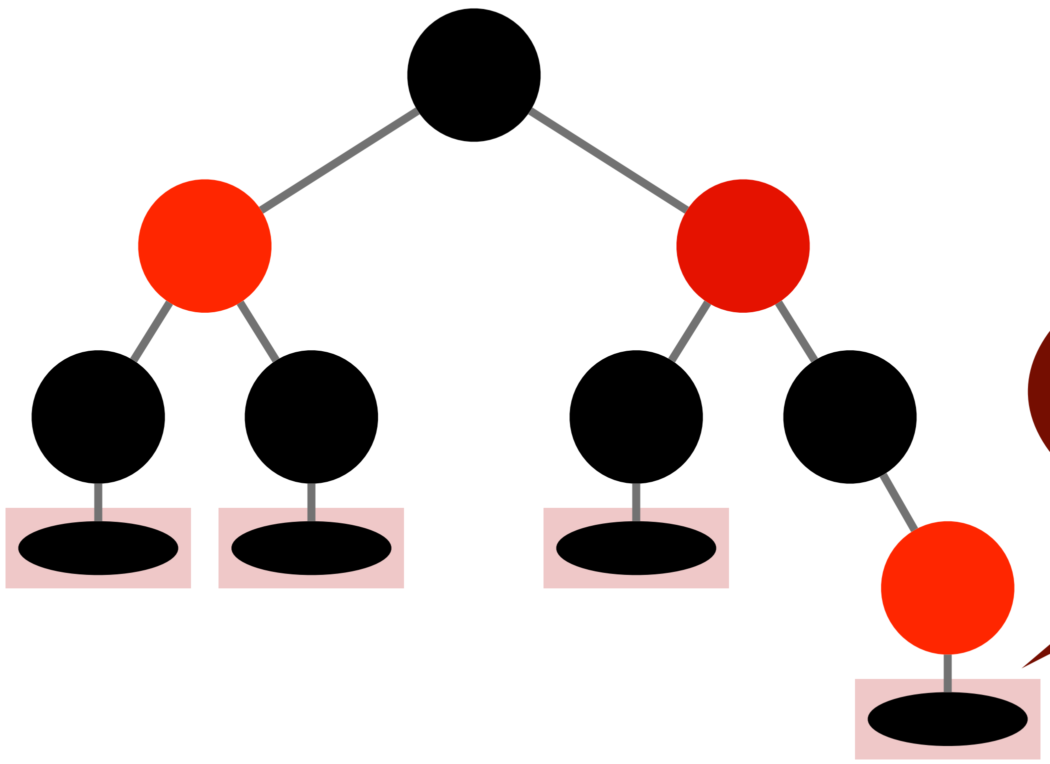
empty nodes
are black



Red Black Trees

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

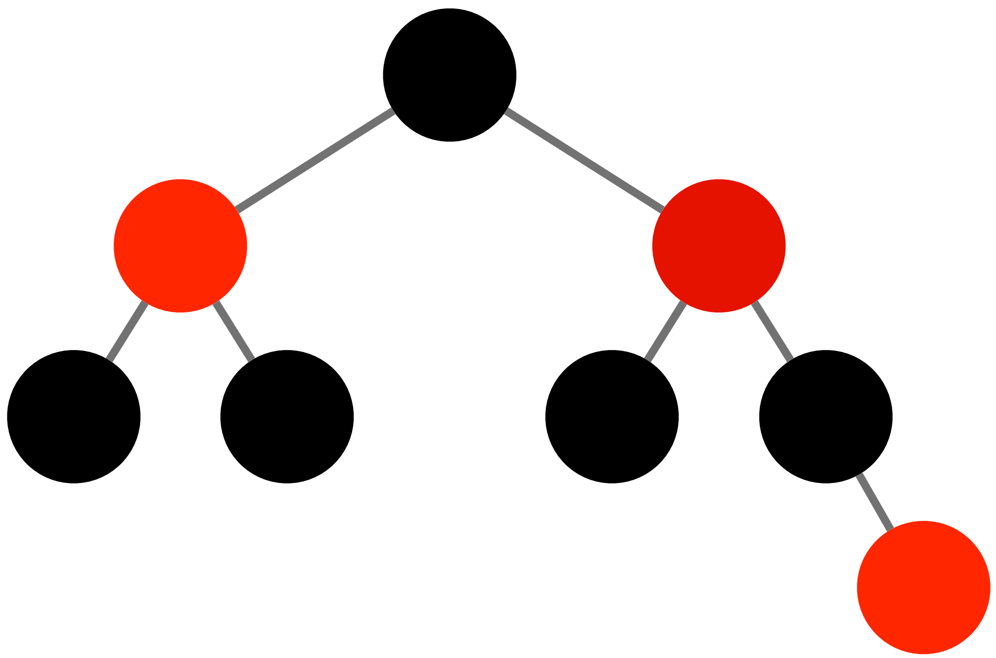
empty nodes are black



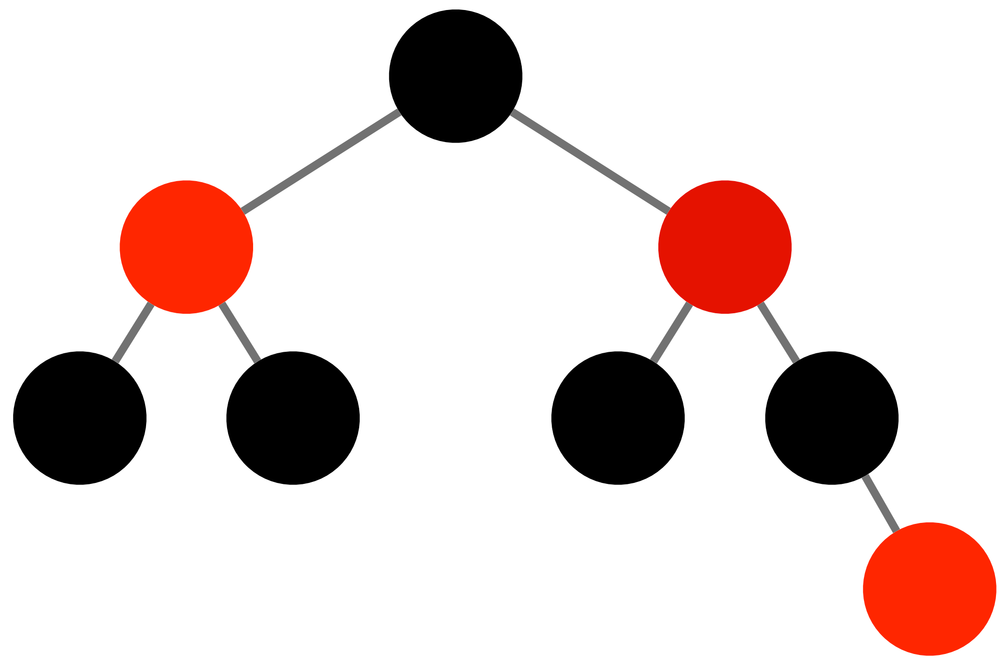
we'll suppress them, moving forward

Red Black Trees

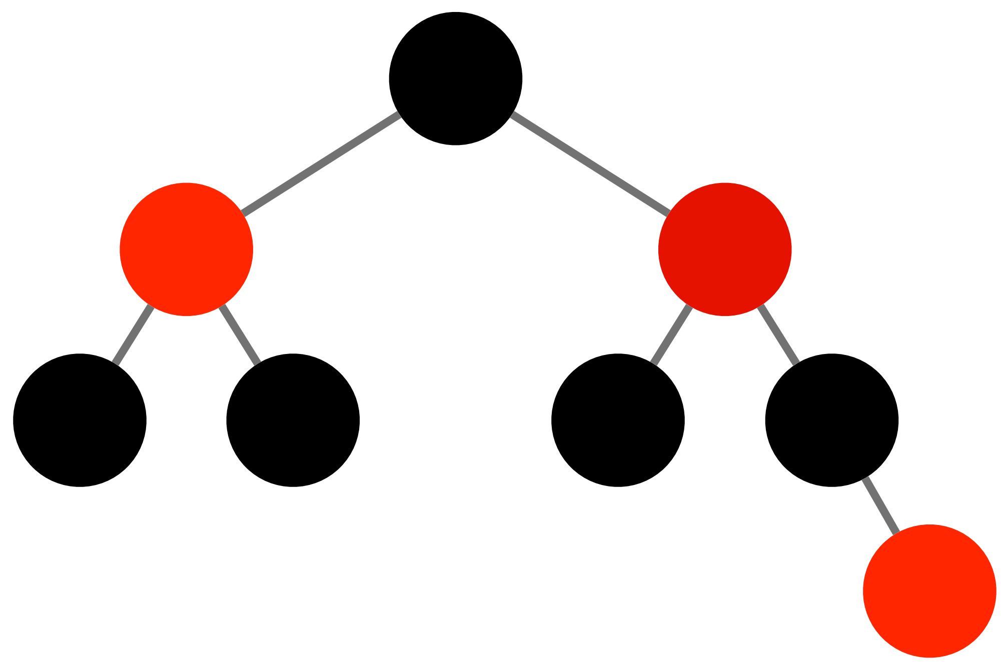
```
datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict
```



Red Black Tree (RBT) Invariant

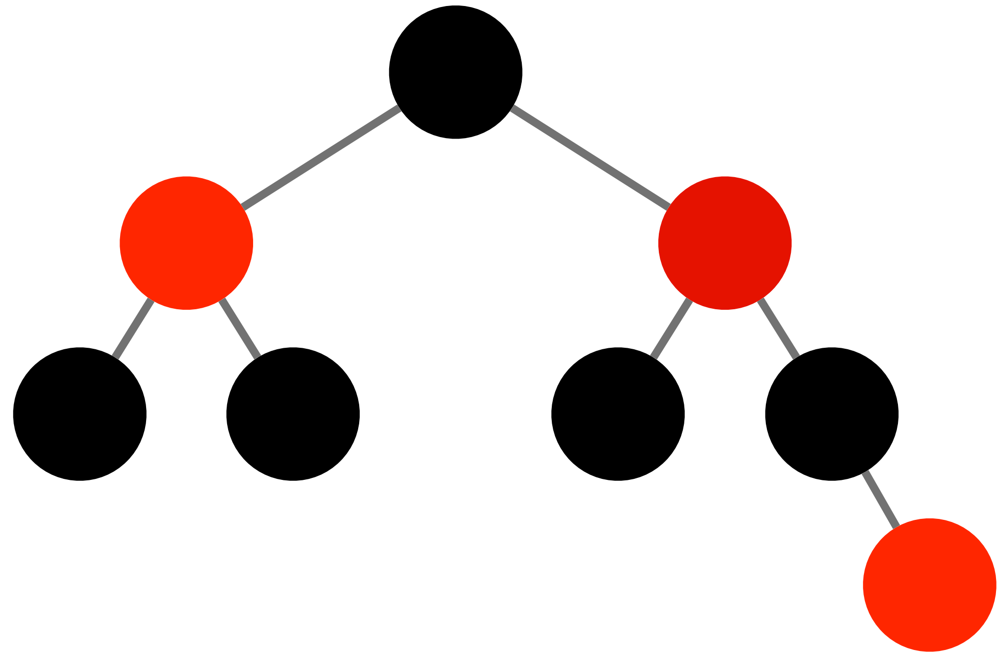


Red Black Tree (RBT) Invariant



A Tree is **sorted** according to an entry's key.

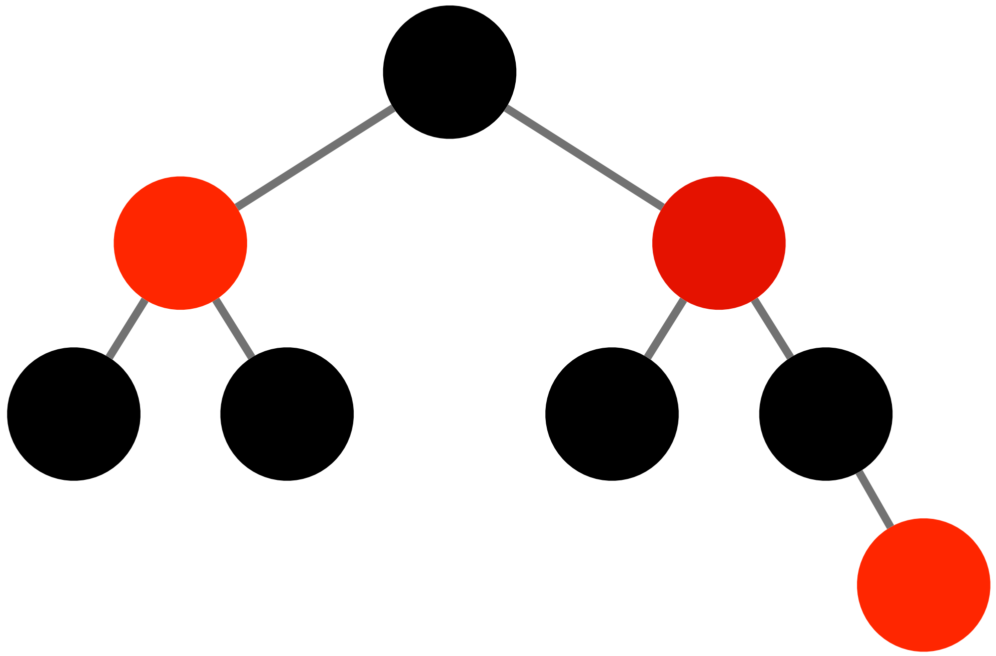
Red Black Tree (RBT) Invariant



A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

Red Black Tree (RBT) Invariant

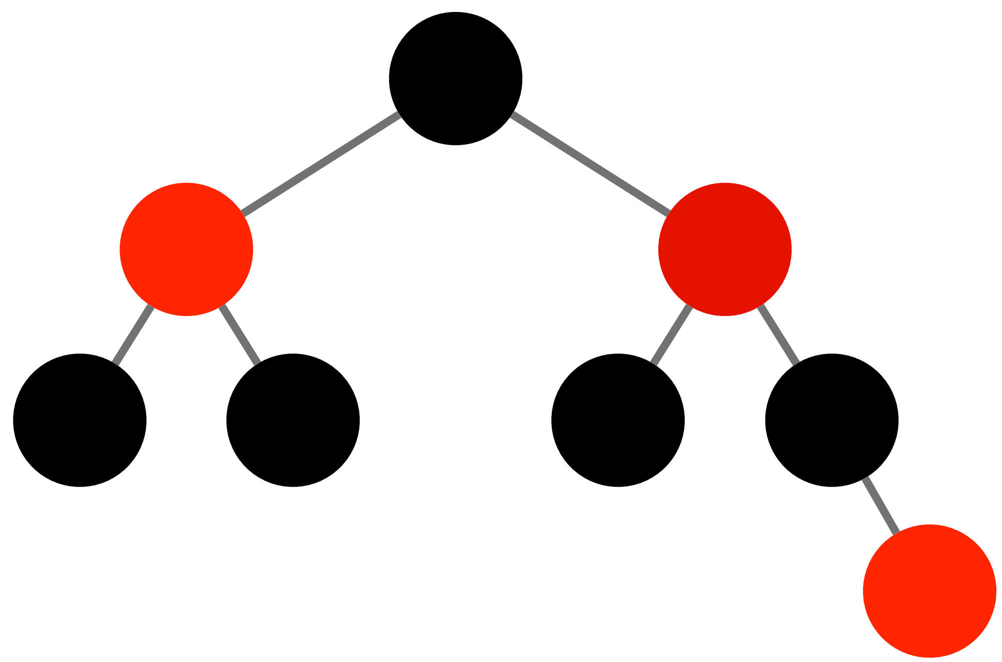


A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Red Black Tree (RBT) Invariant



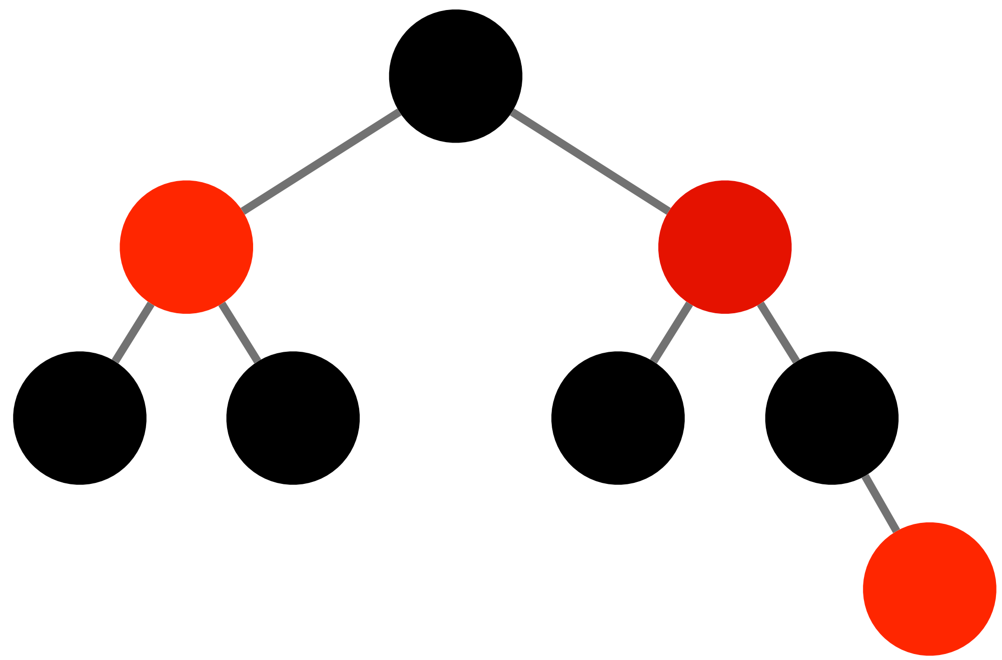
or, the number of black nodes from the root

A — Tree is **sorted** according to an entry's key.

B — A **red** node's children must be black.

C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Red Black Tree (RBT) Invariant

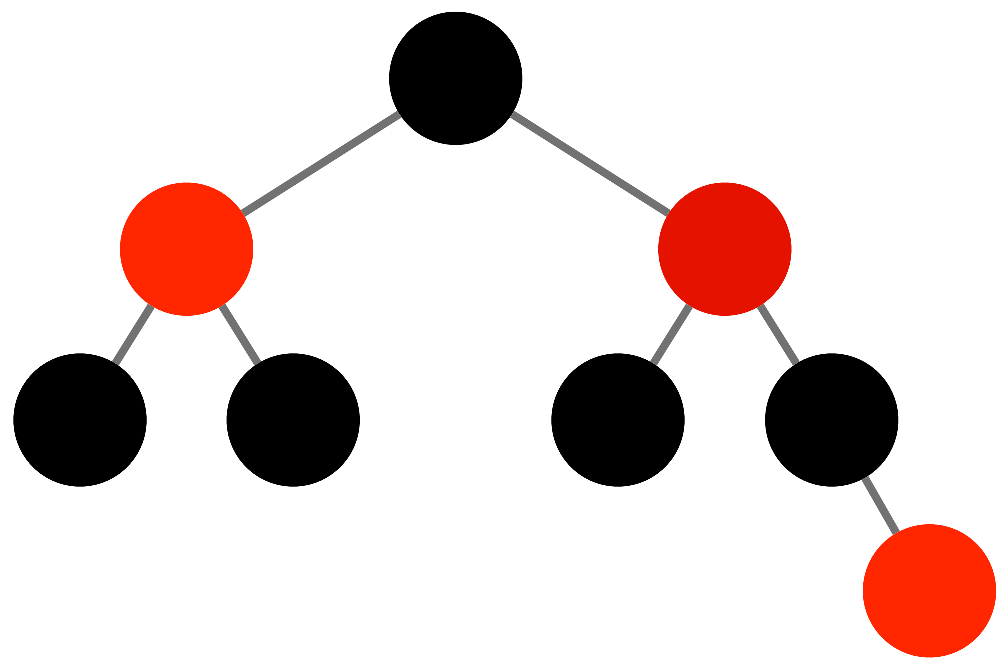


A — Tree is **sorted** according to an entry's key.

B — A **red** node's children must be black.

C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Red Black Tree (RBT) Invariant



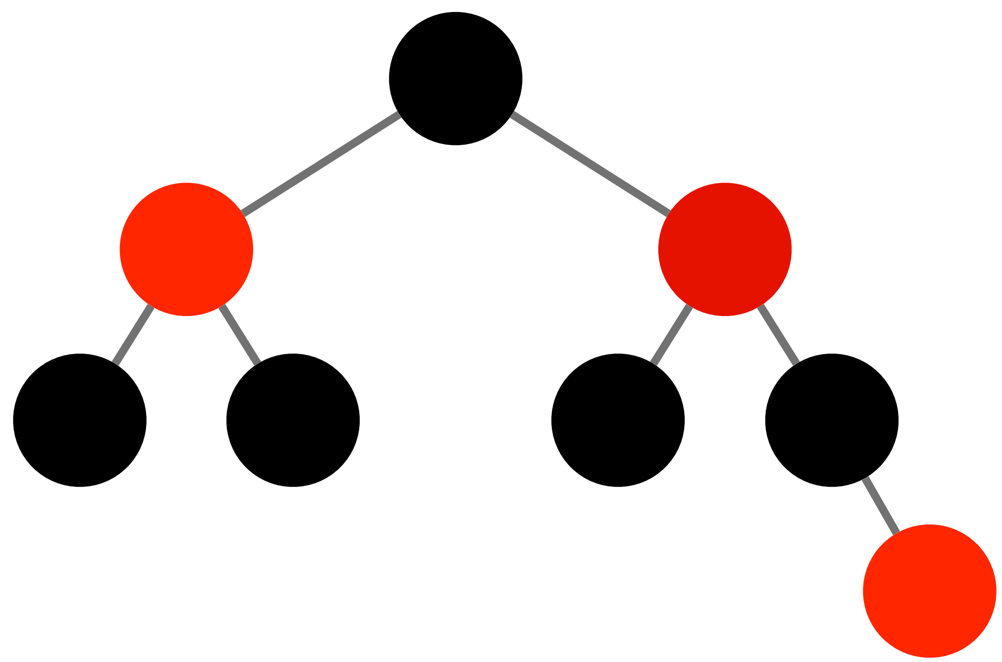
A — Tree is **sorted** according to an entry's key.

B — A **red** node's children must be black.

C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

This representation invariant ensures that tree is roughly balanced:

Red Black Tree (RBT) Invariant



A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

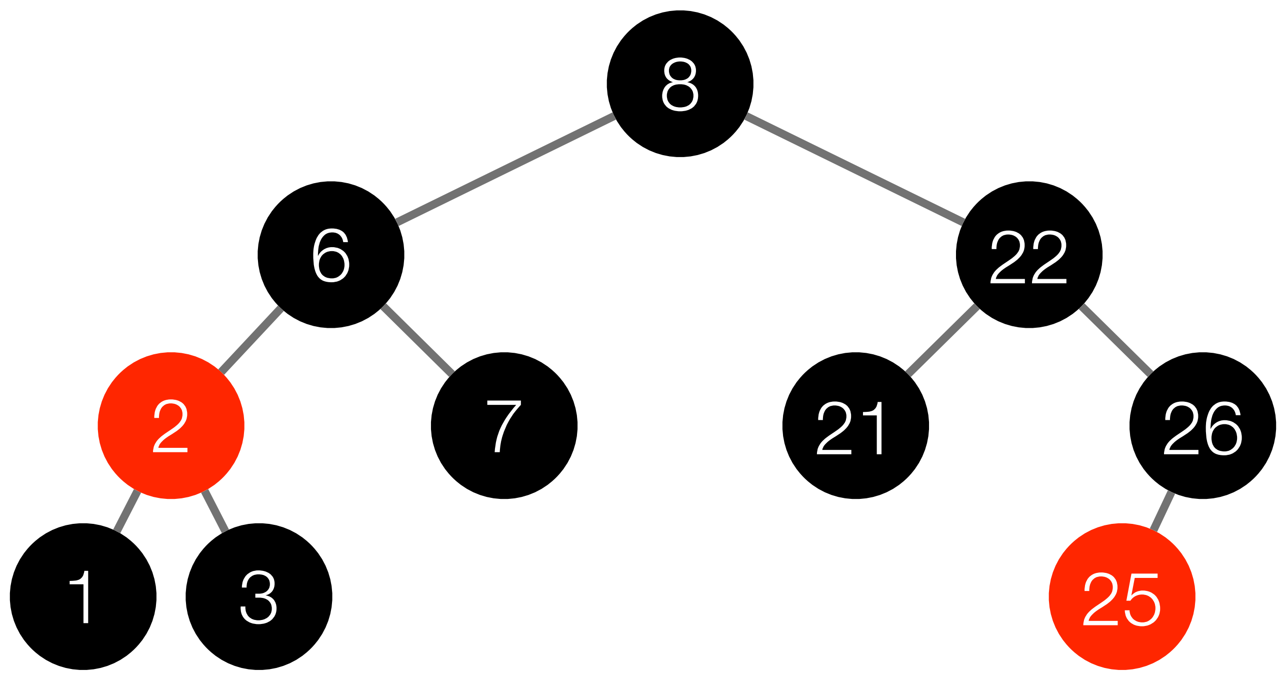
C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

This representation invariant ensures that tree is roughly balanced:

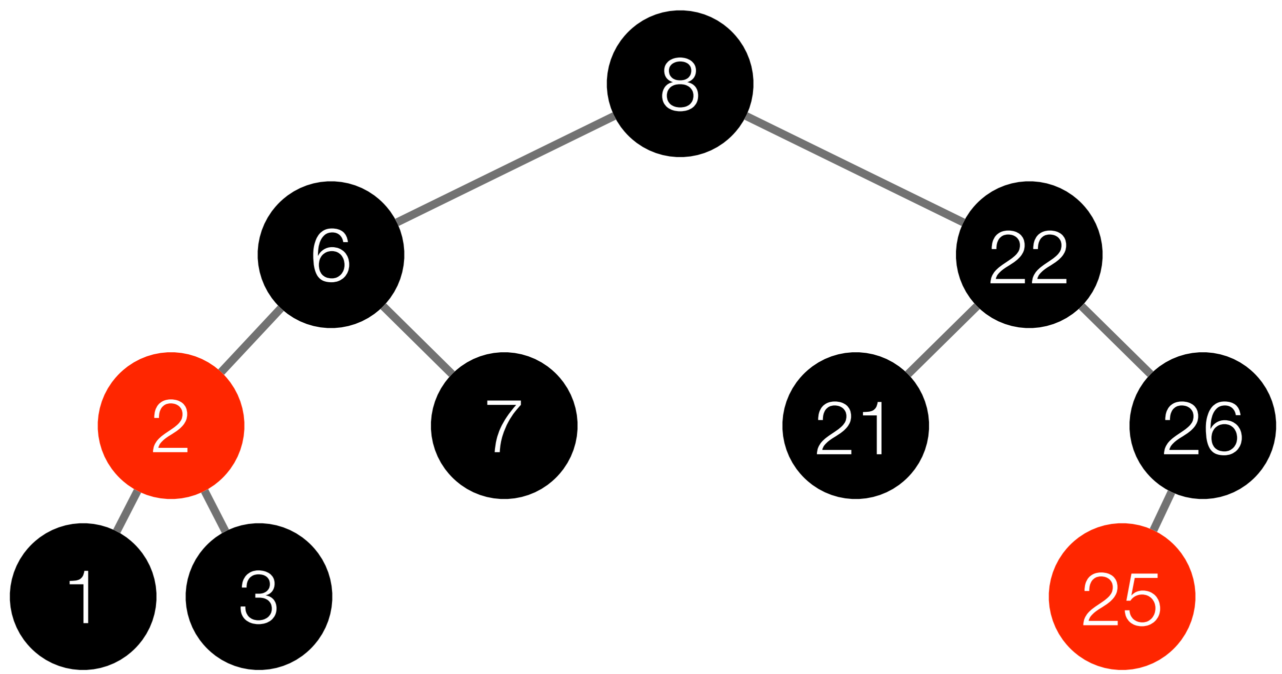
$$\text{depth} \leq 2\log_2(|\text{nodes}| + 1)$$

Let's play with a given red black tree

Let's play with a given red black tree

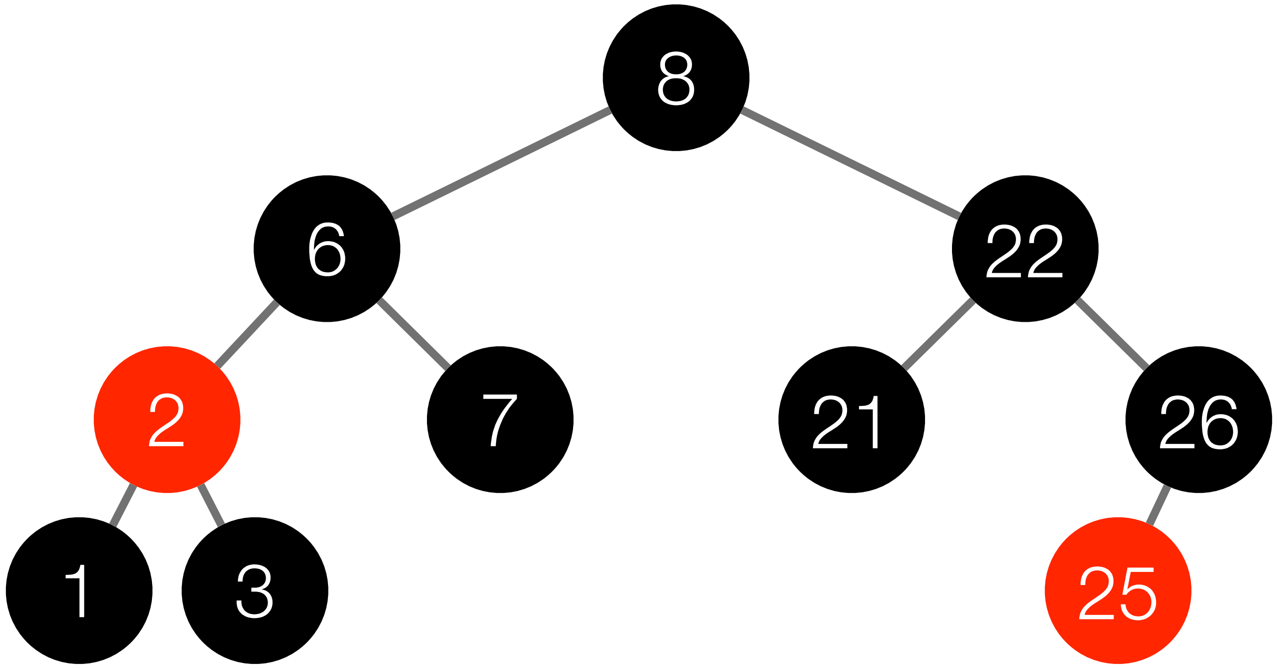


Let's play with a given red black tree



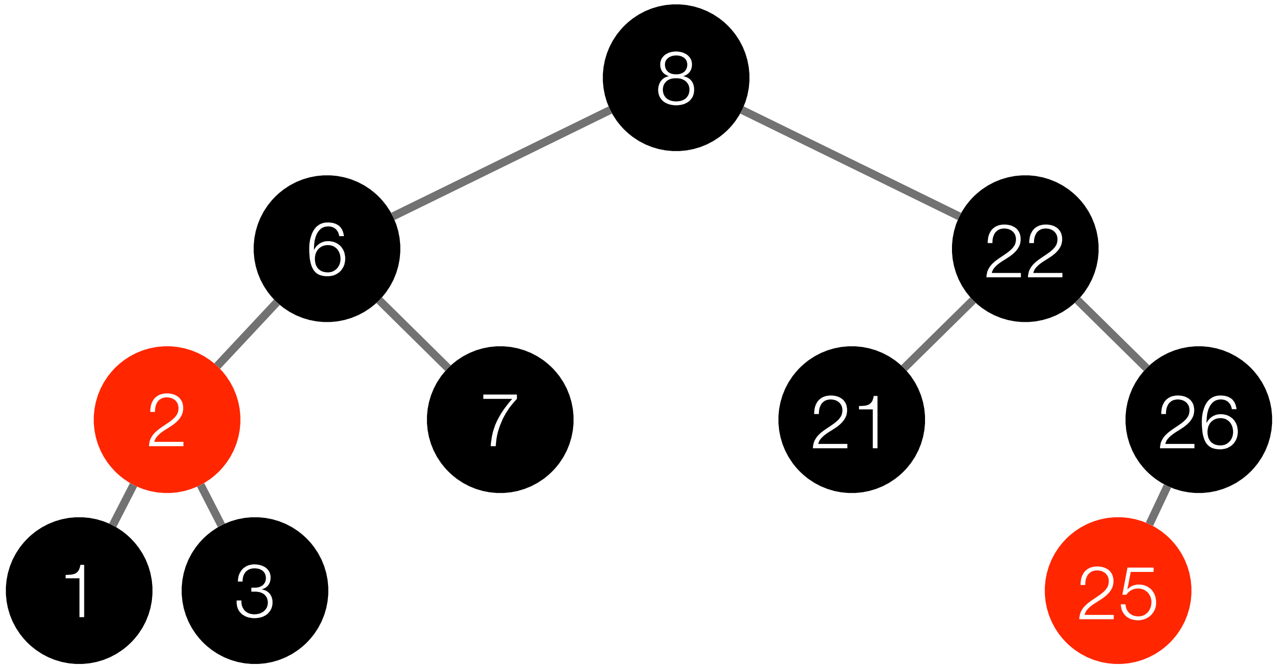
(For simplicity, we use integer keys and omit value part of an entry.)

Let's play with a given red black tree



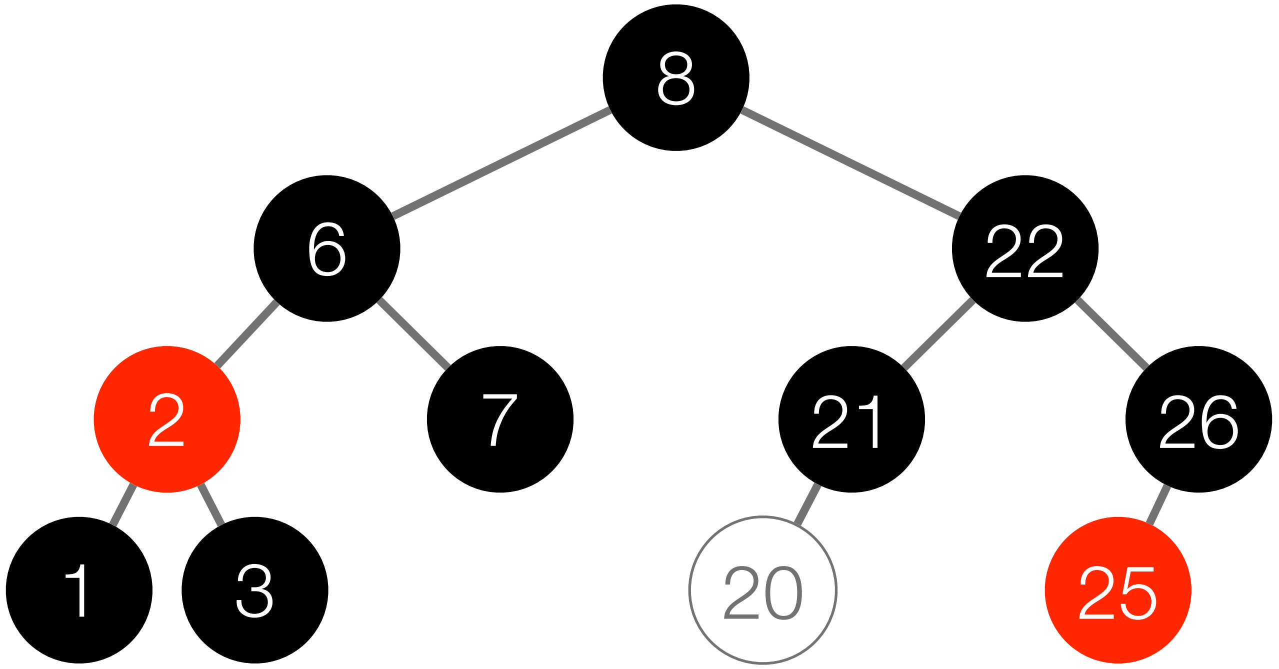
Let's play with a given red black tree

Let's insert 20:



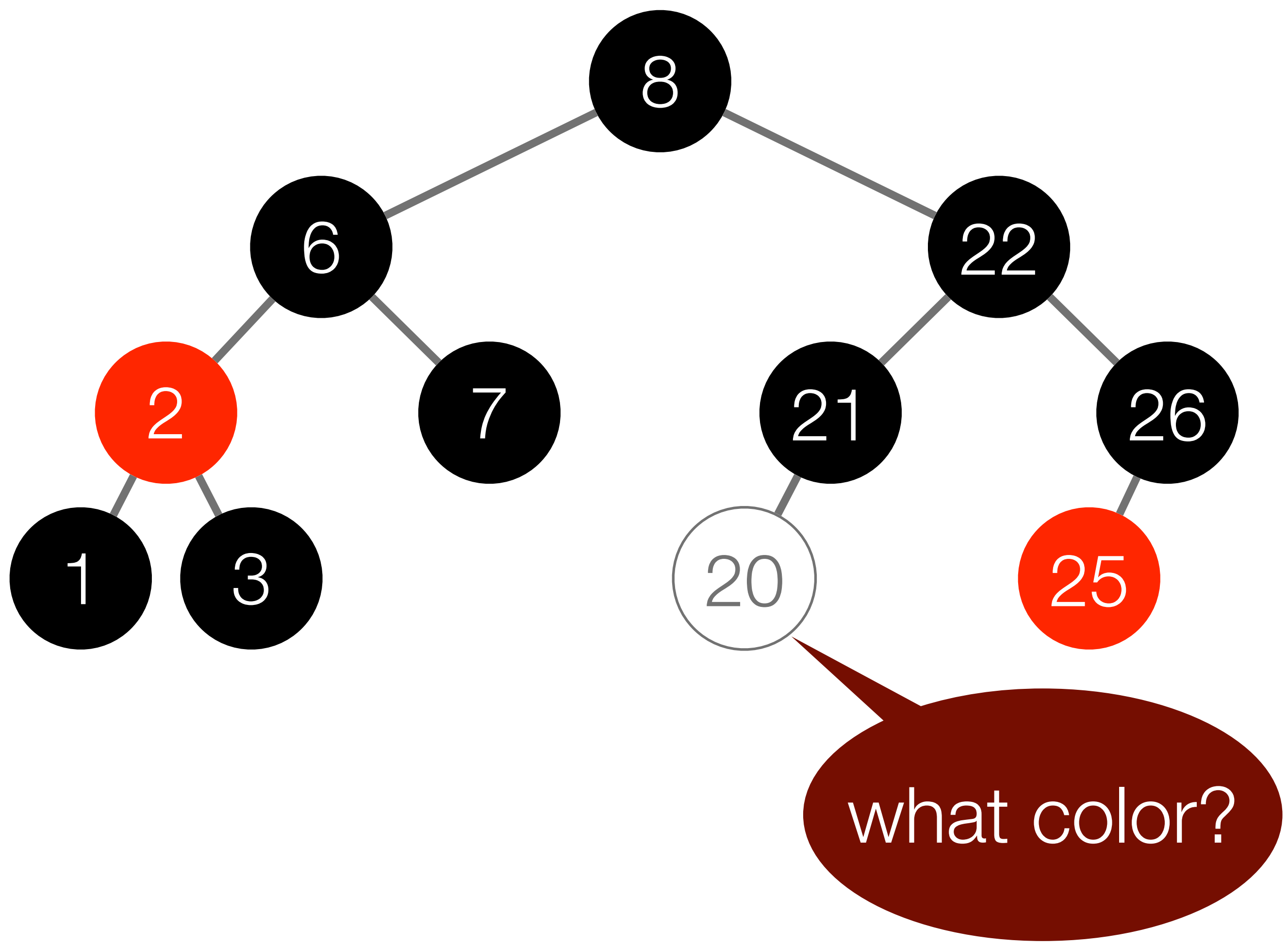
Let's play with a given red black tree

Let's insert 20:



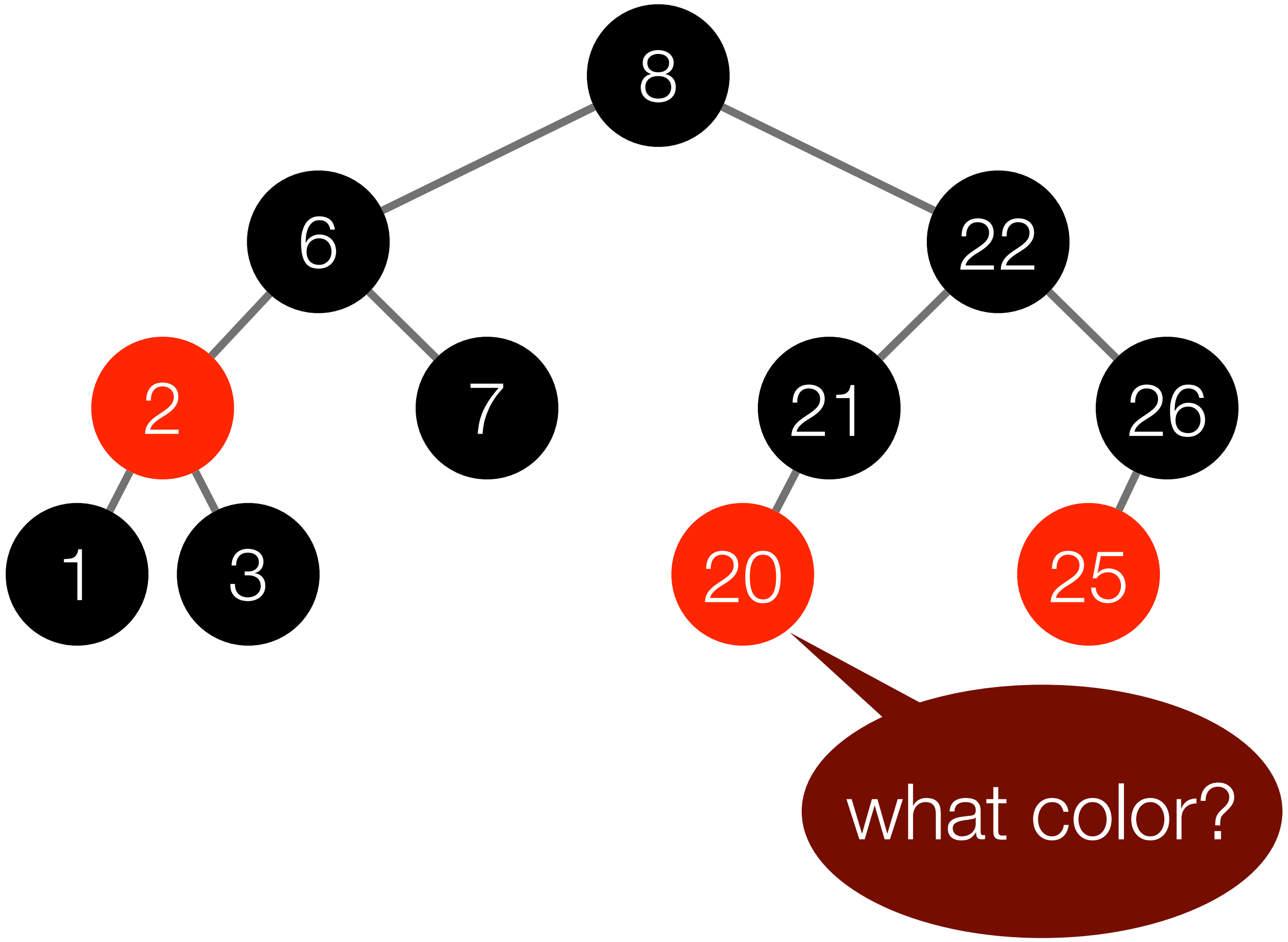
Let's play with a given red black tree

Let's insert 20:



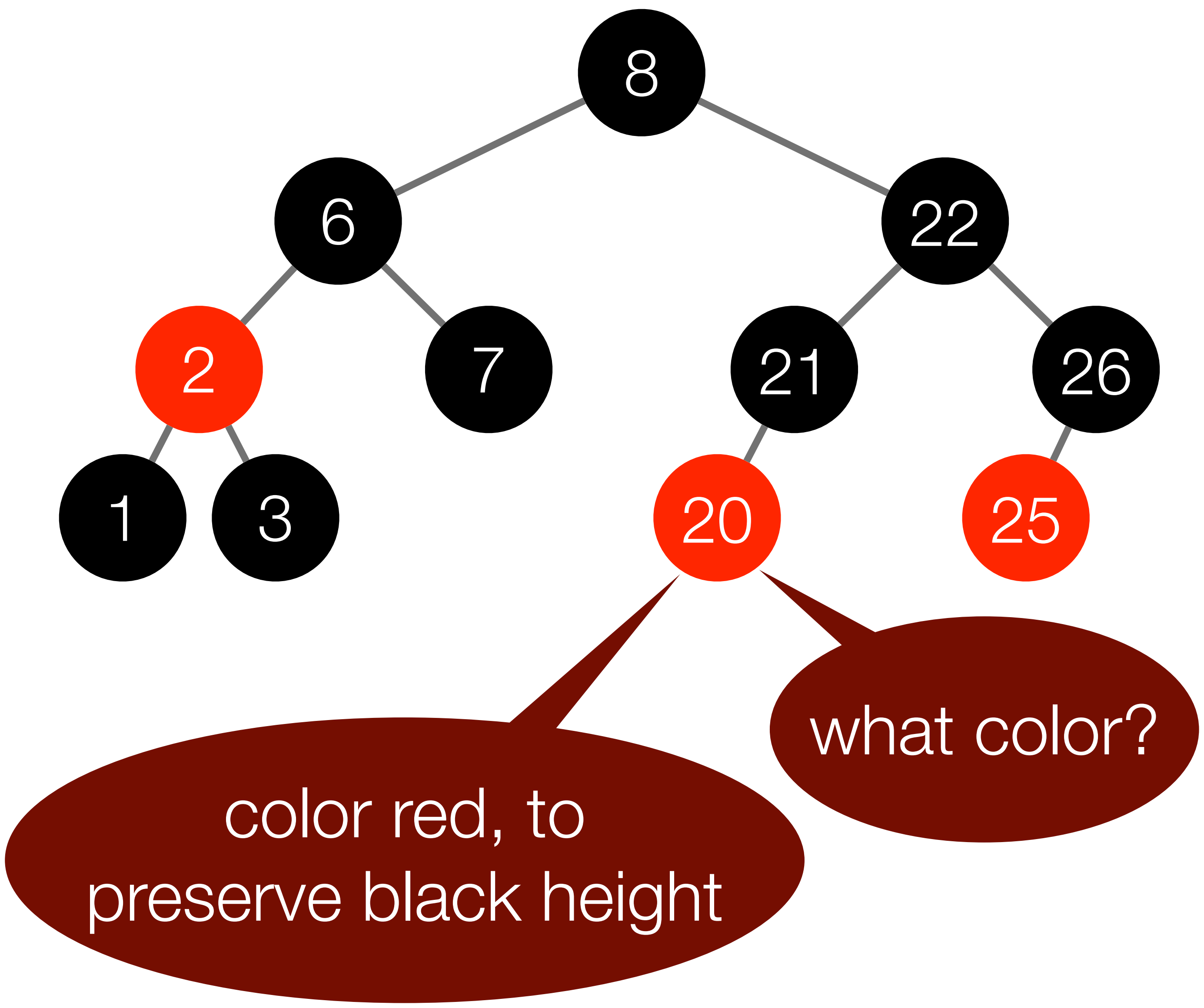
Let's play with a given red black tree

Let's insert 20:

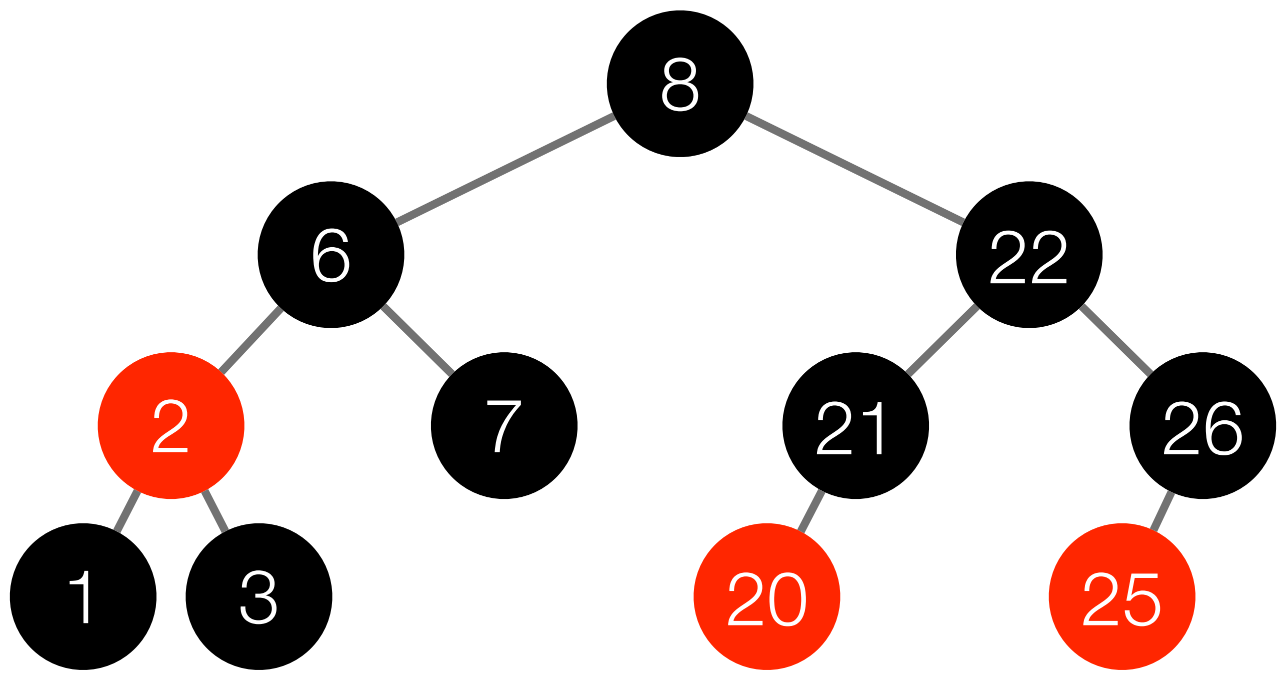


Let's play with a given red black tree

Let's insert 20:

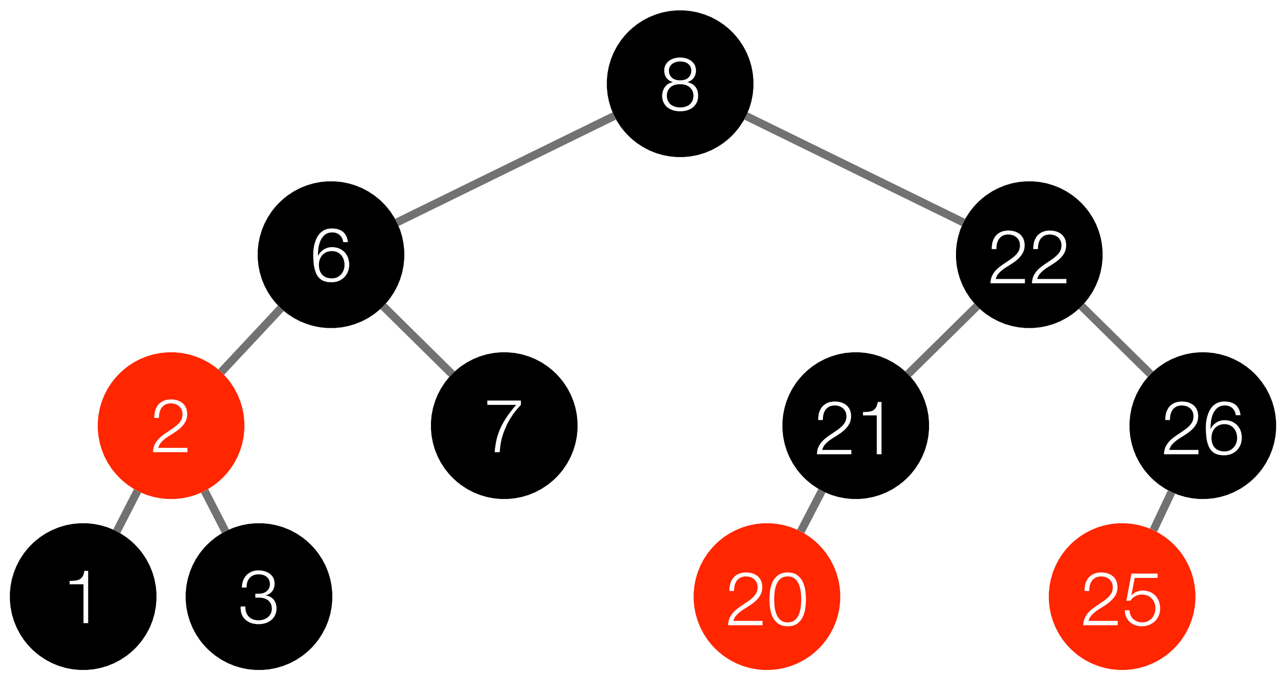


Let's play with a given red black tree



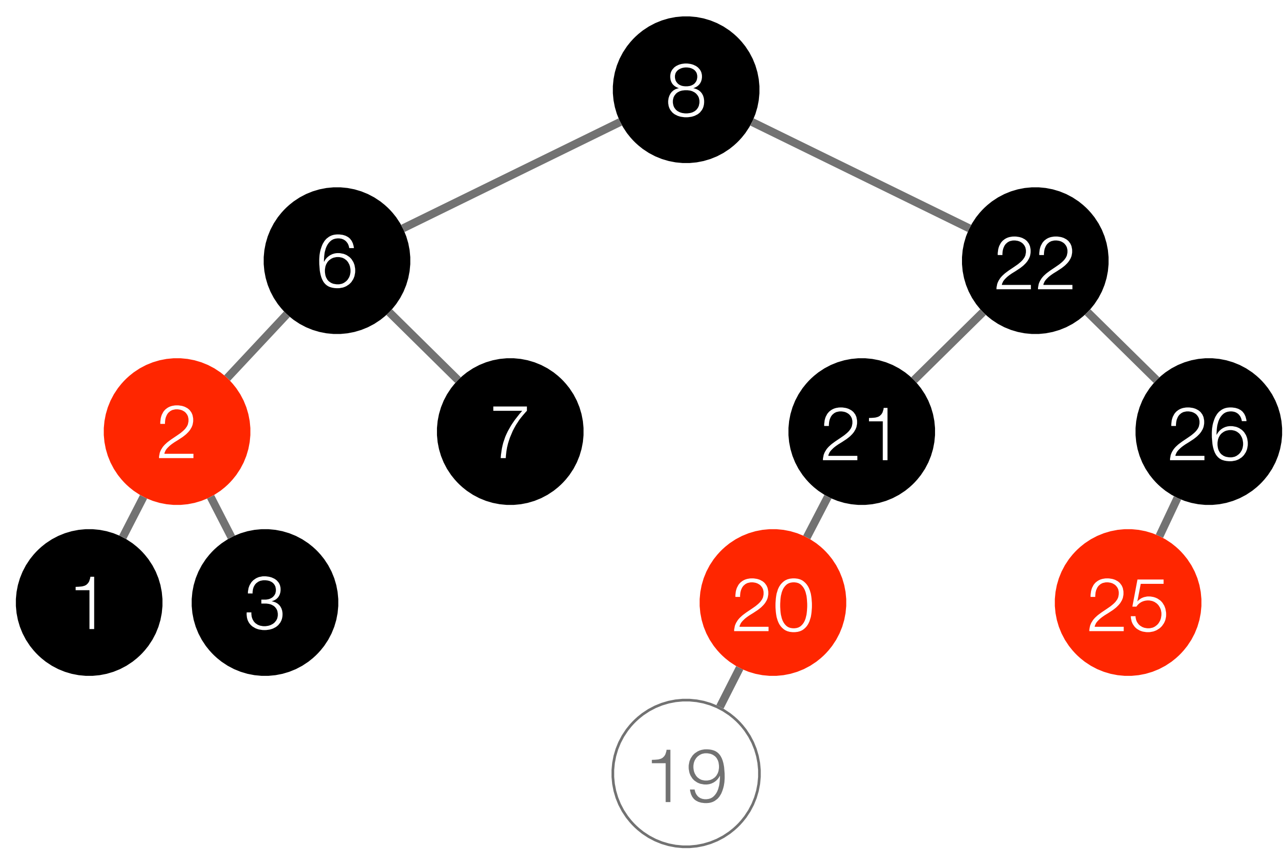
Let's play with a given red black tree

Now, let's insert 19:



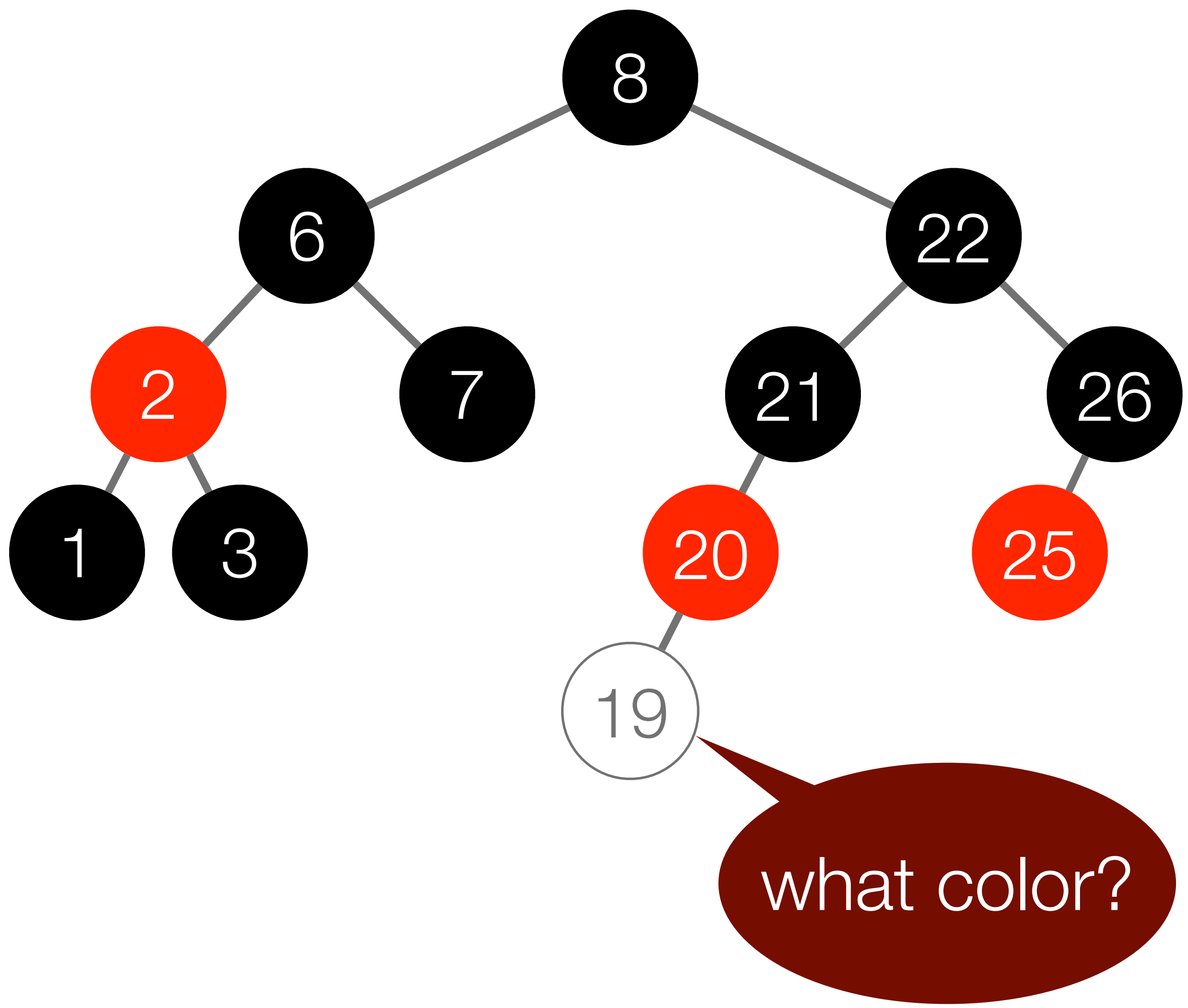
Let's play with a given red black tree

Now, let's insert 19:



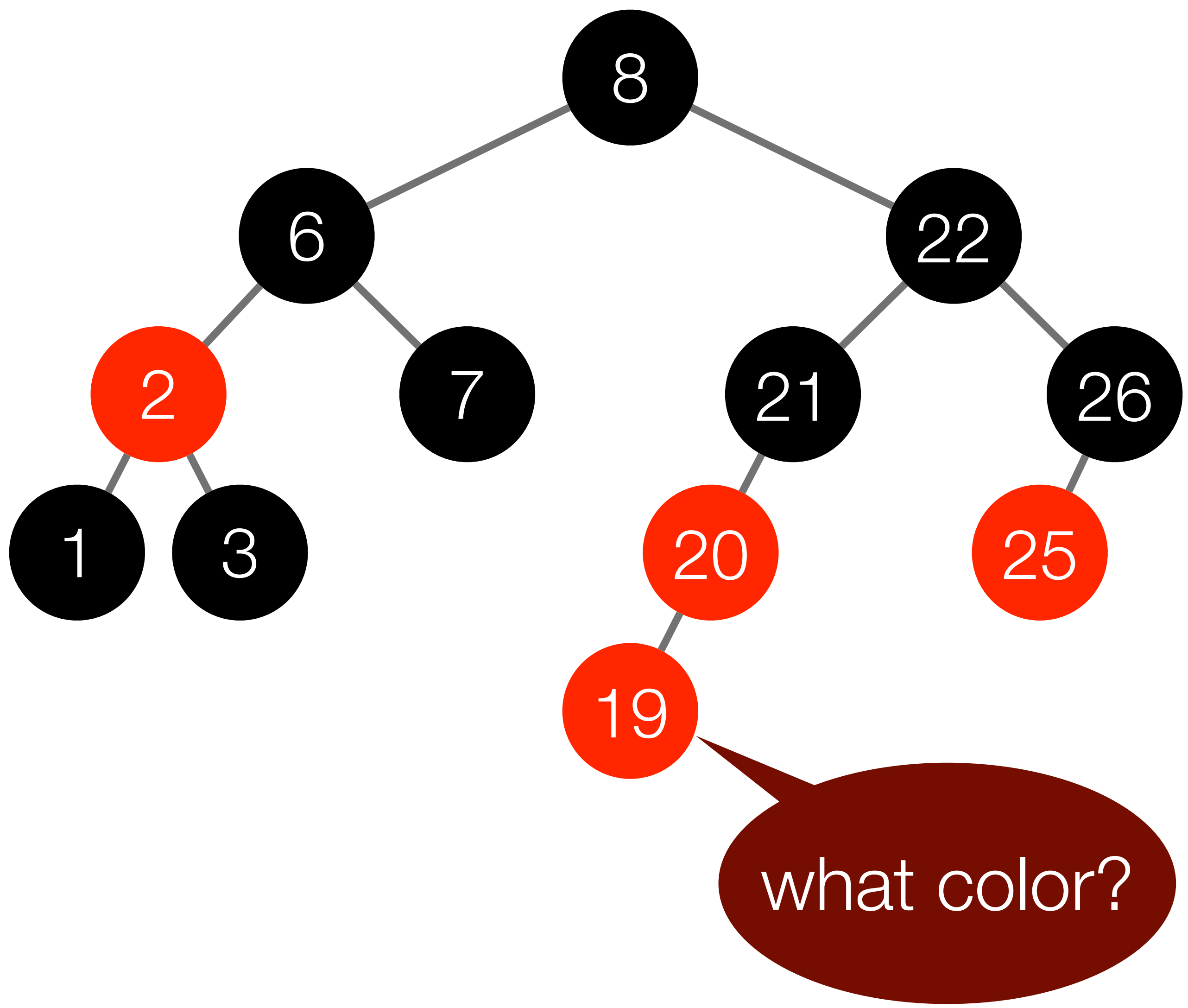
Let's play with a given red black tree

Now, let's insert 19:



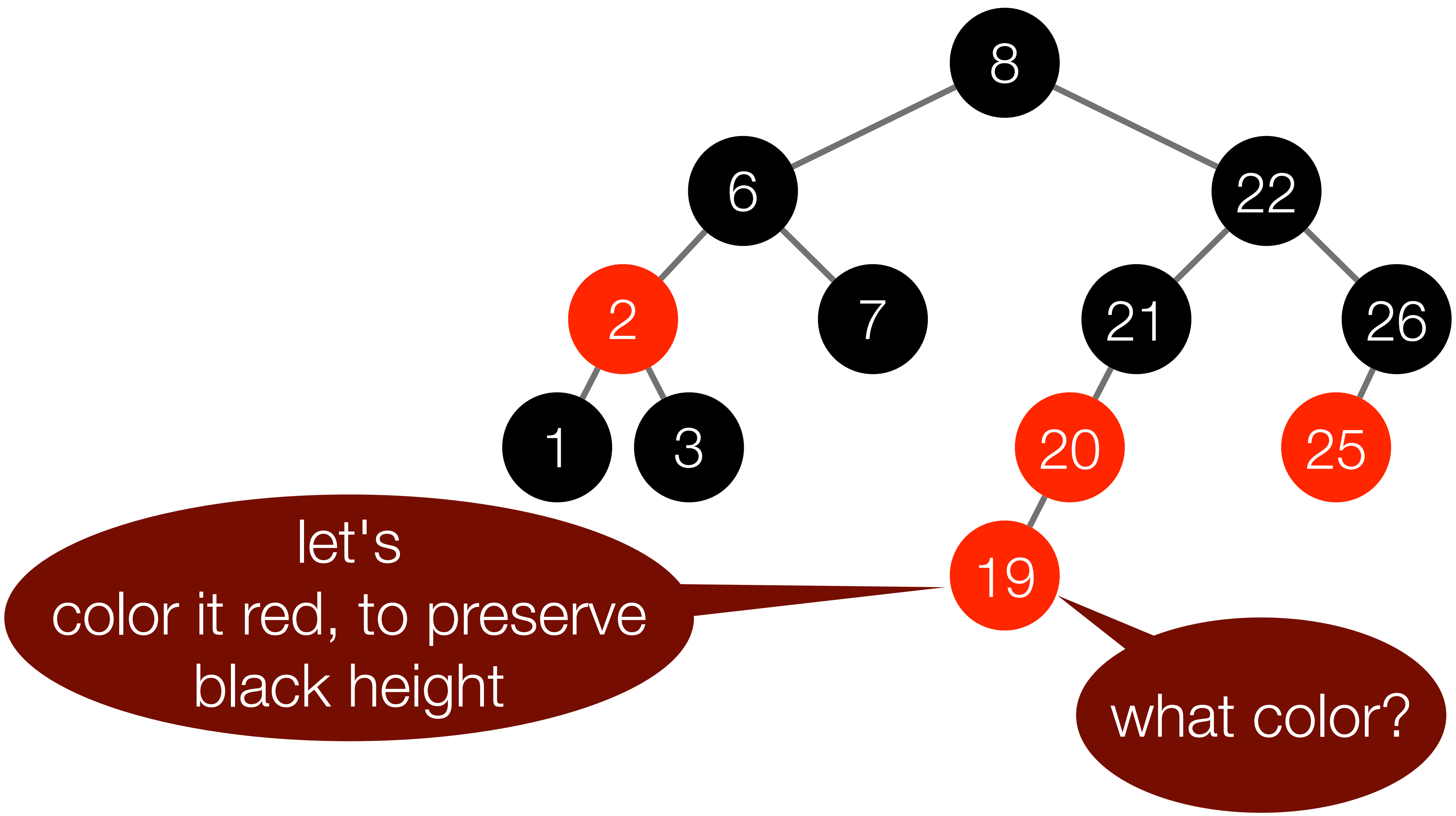
Let's play with a given red black tree

Now, let's insert 19:



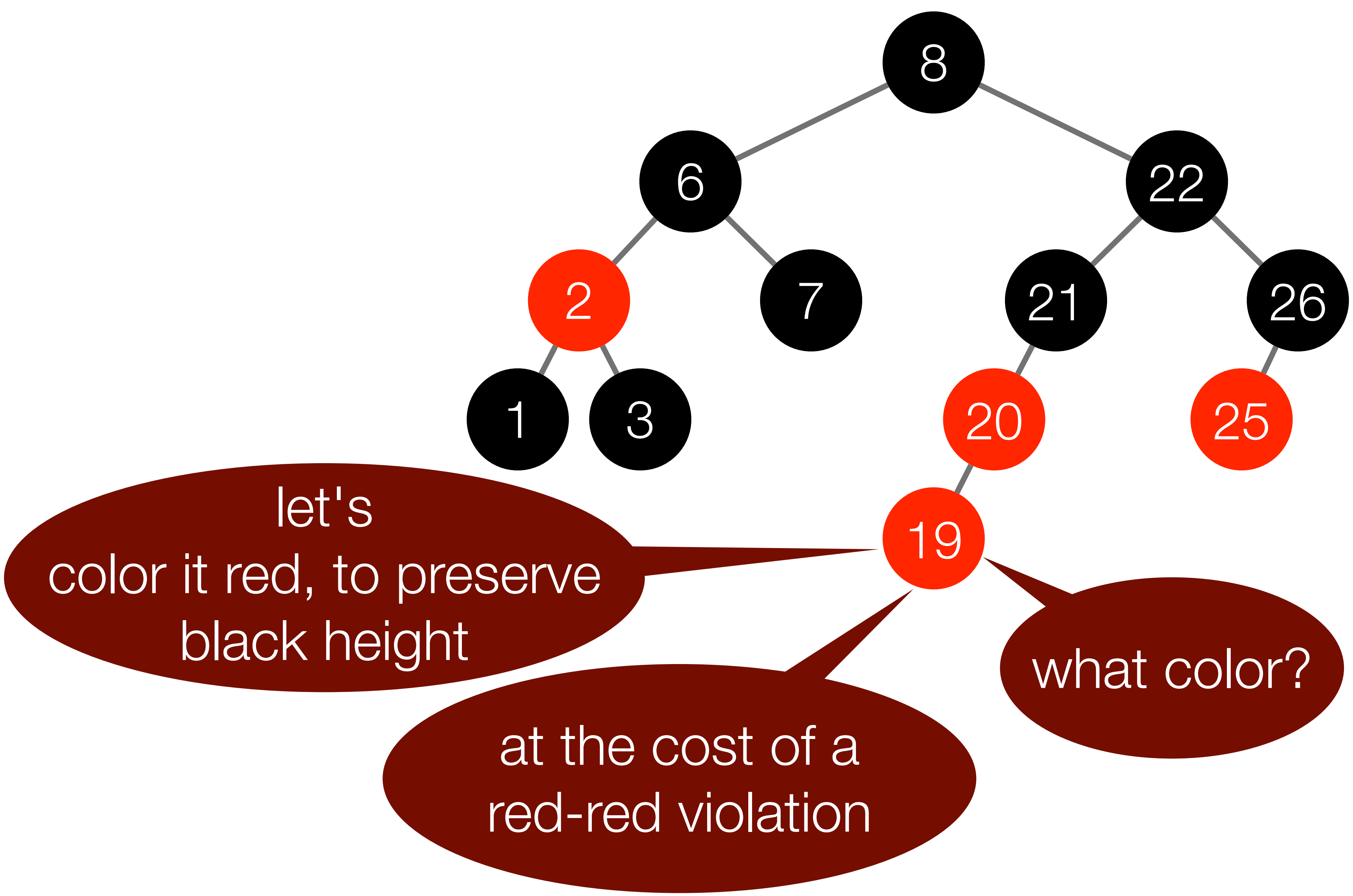
Let's play with a given red black tree

Now, let's insert 19:

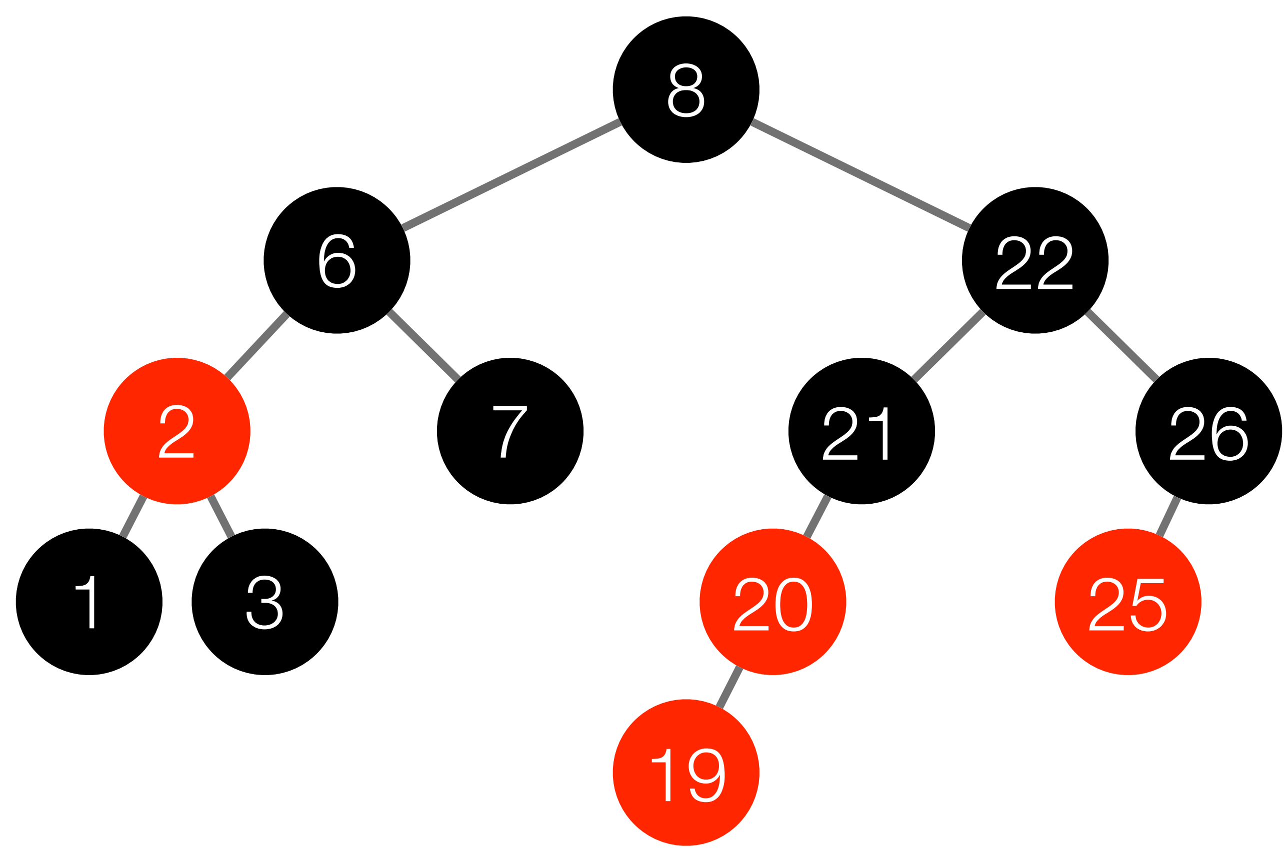


Let's play with a given red black tree

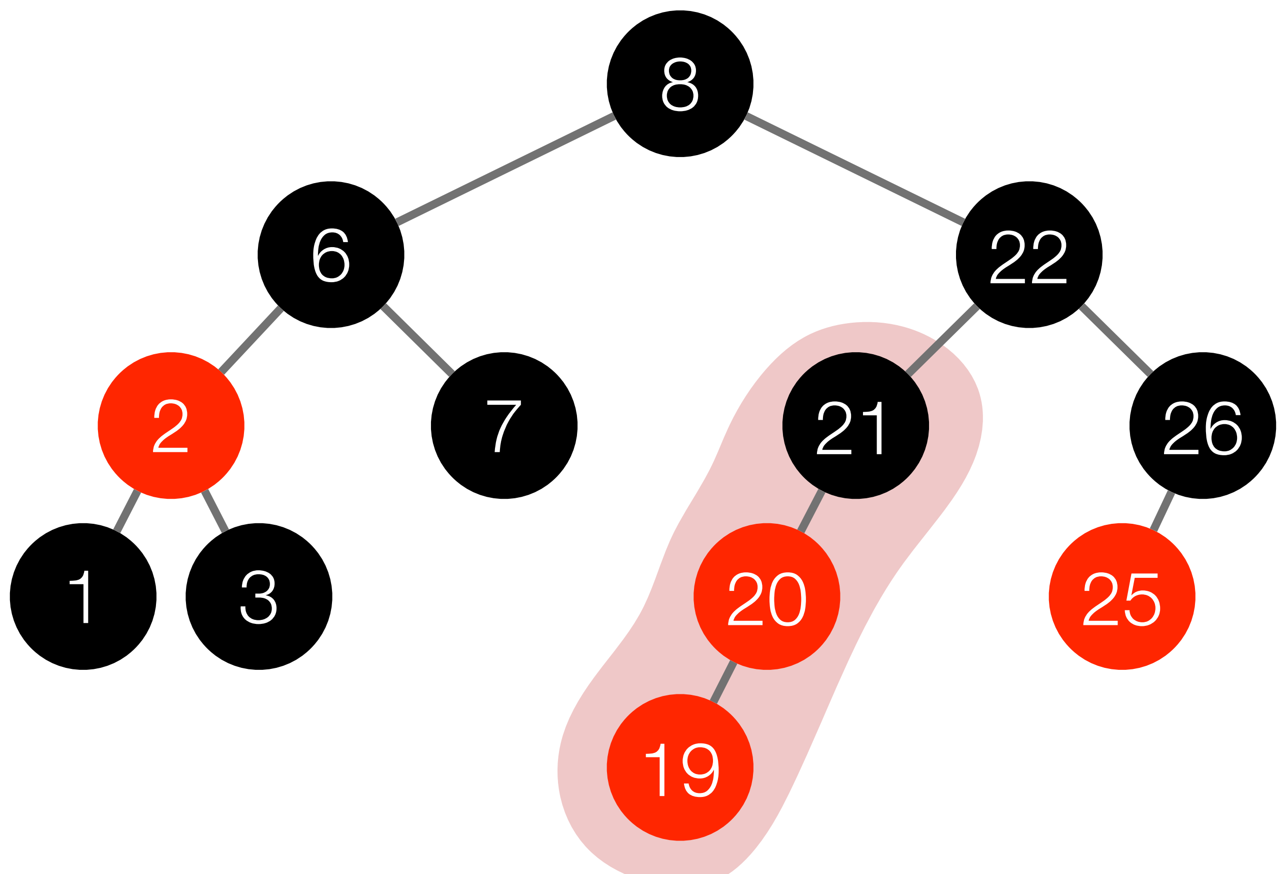
Now, let's insert 19:



Let's play with a given red black tree

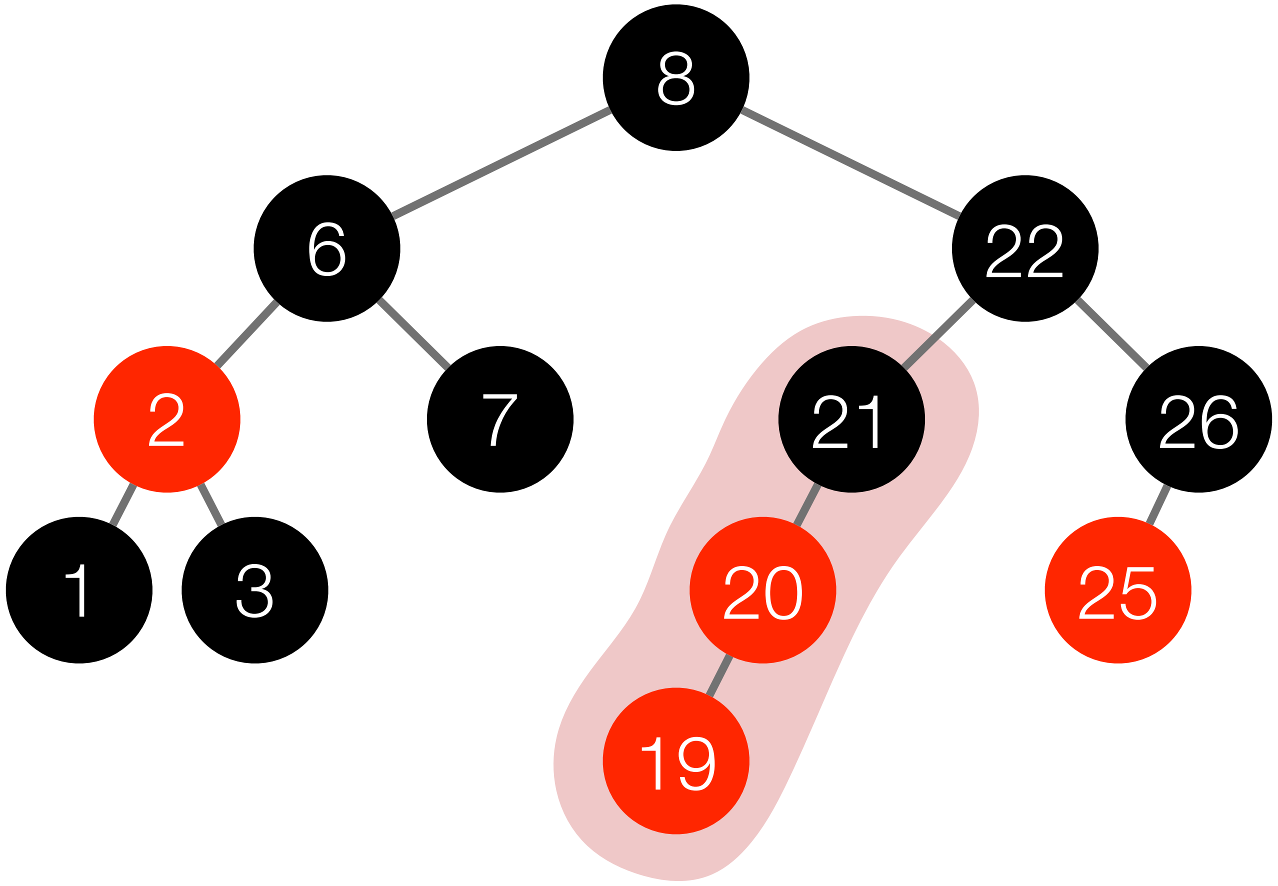


Let's play with a given red black tree



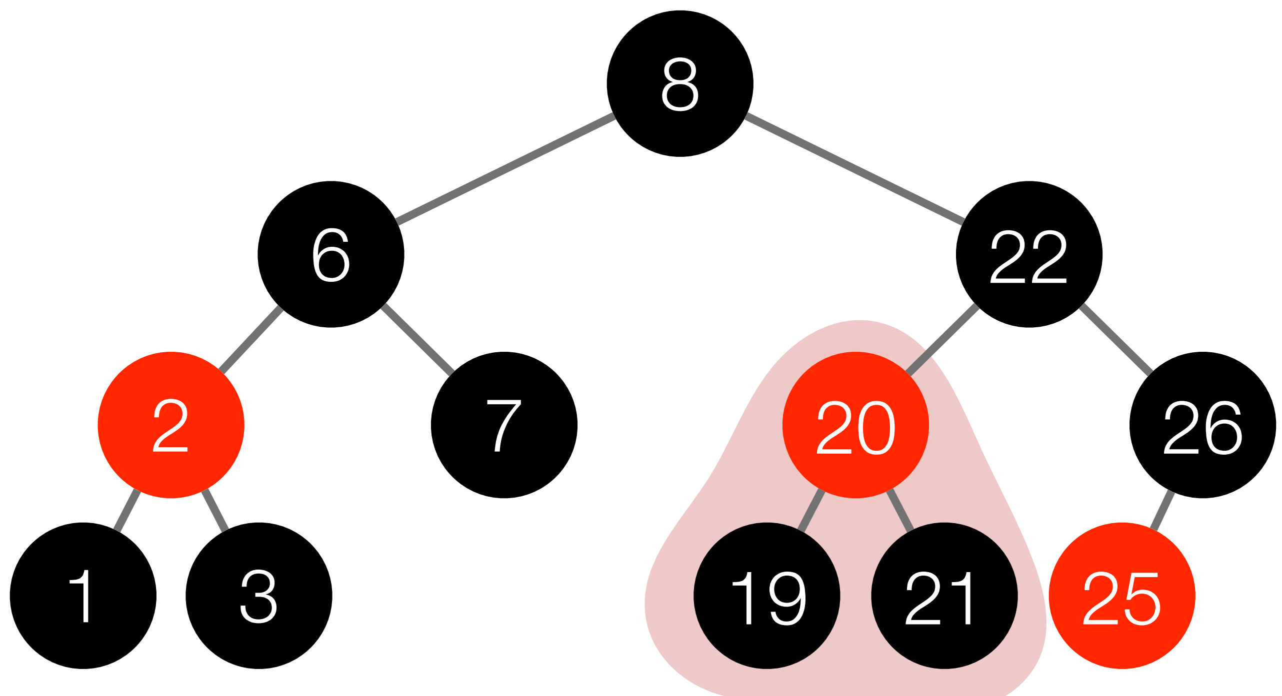
Let's play with a given red black tree

Let's fix it with a rotation and re-coloring:



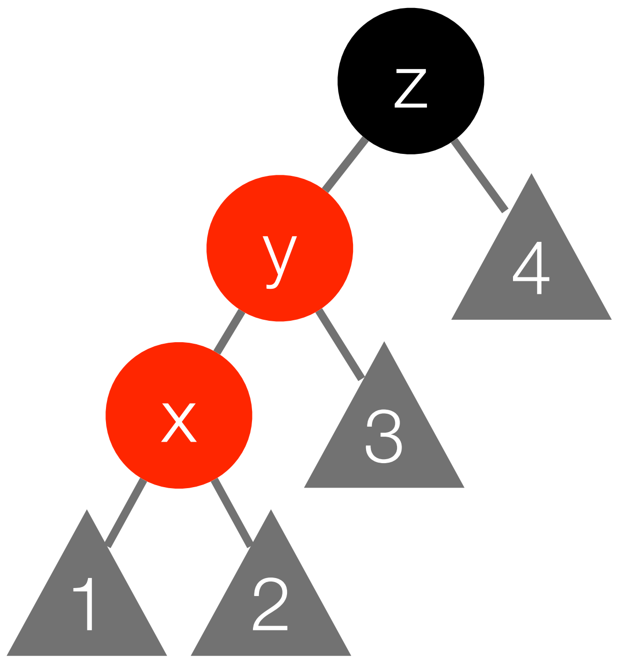
Let's play with a given red black tree

Let's fix it with a rotation and re-coloring:

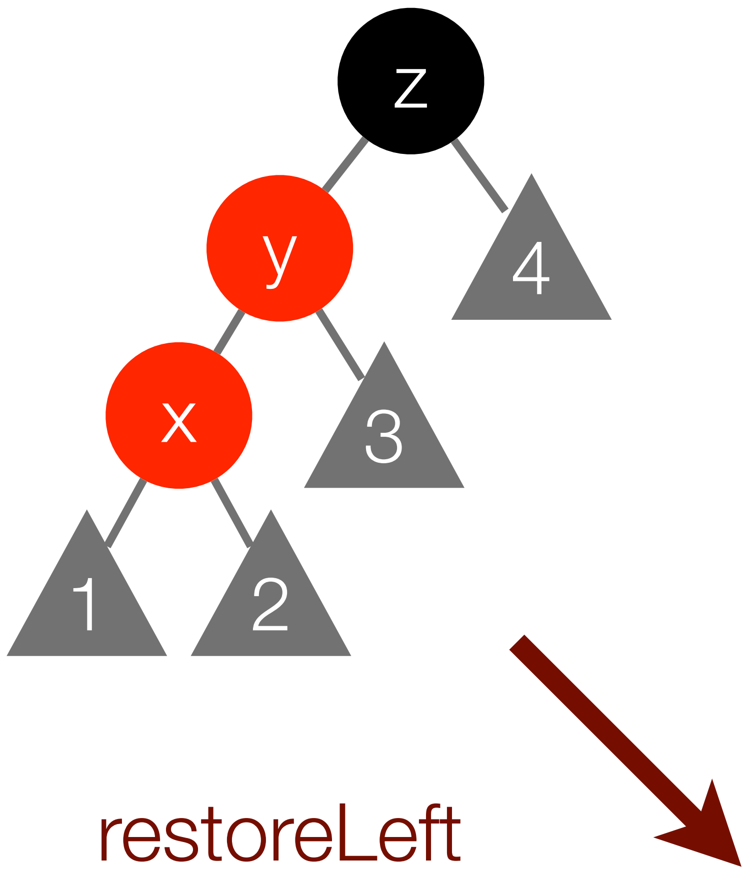


Two out of four possible rotations/re-coloring

Two out of four possible rotations/re-coloring

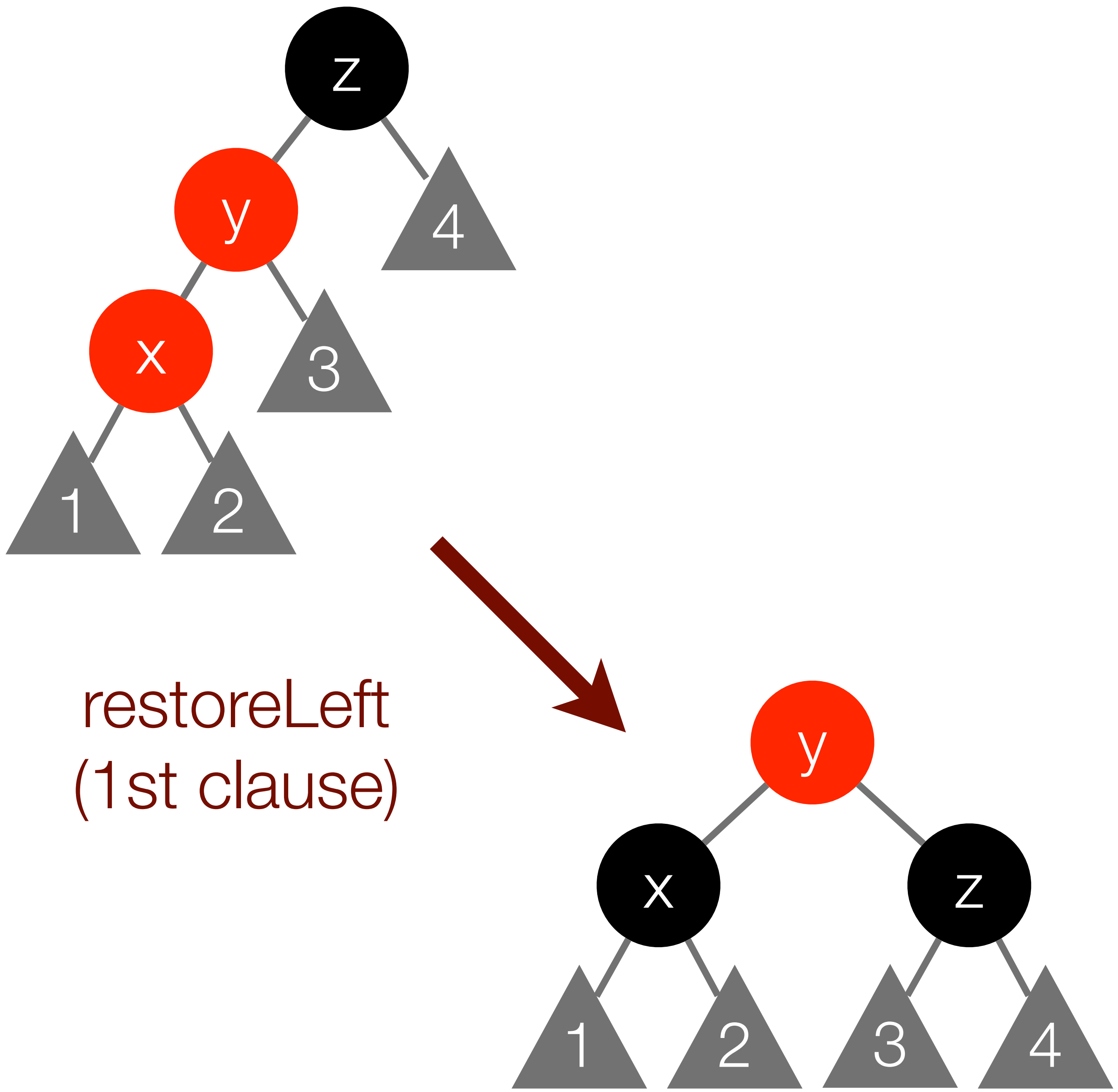


Two out of four possible rotations/re-coloring



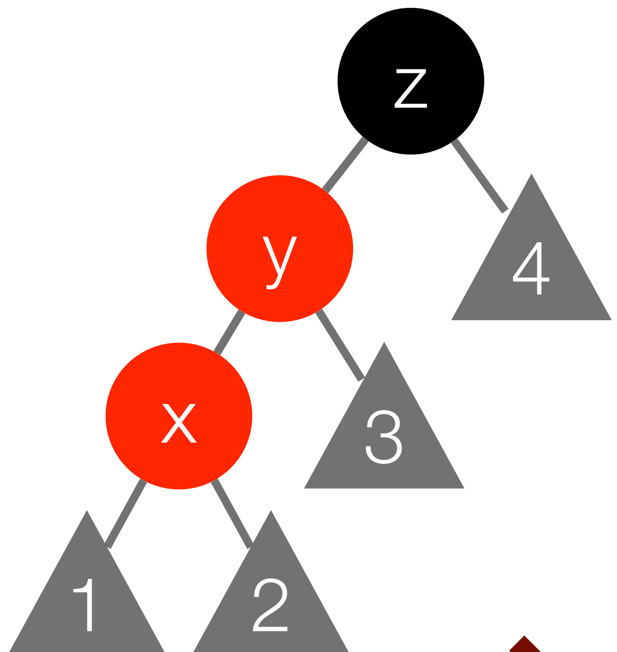
restoreLeft
(1st clause)

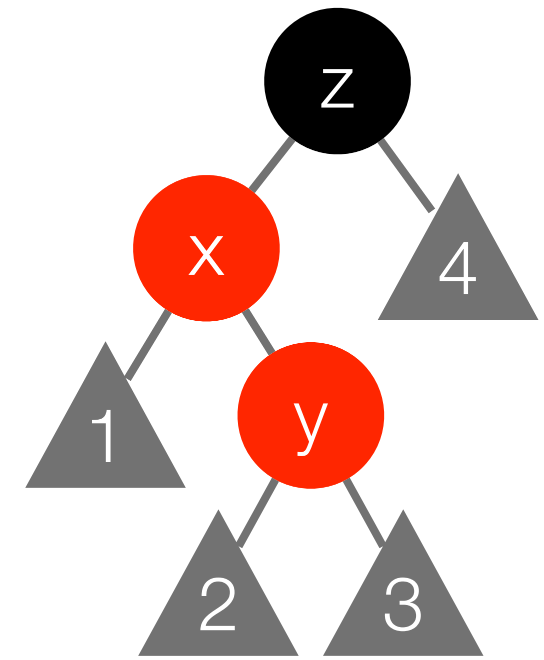
Two out of four possible rotations/re-coloring



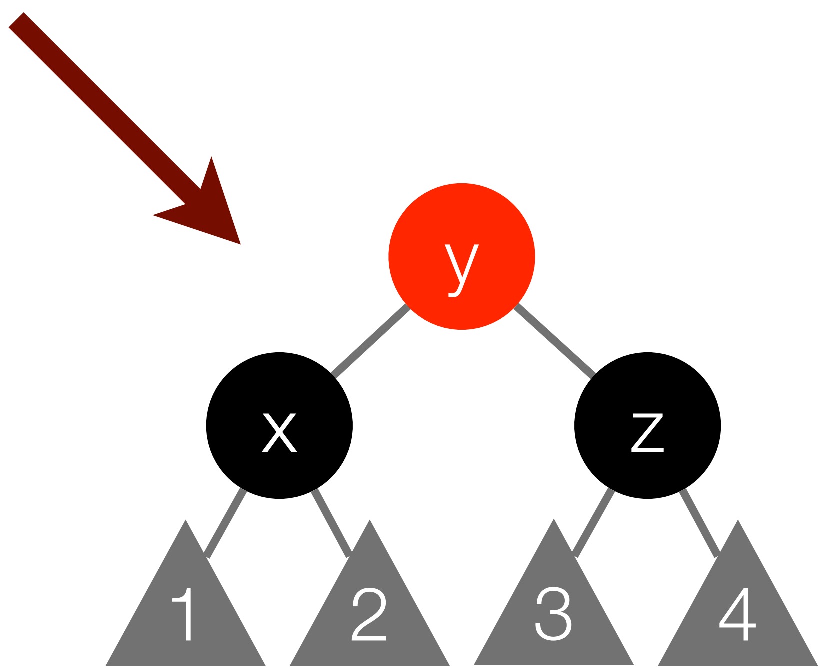
restoreLeft
(1st clause)

Two out of four possible rotations/re-coloring

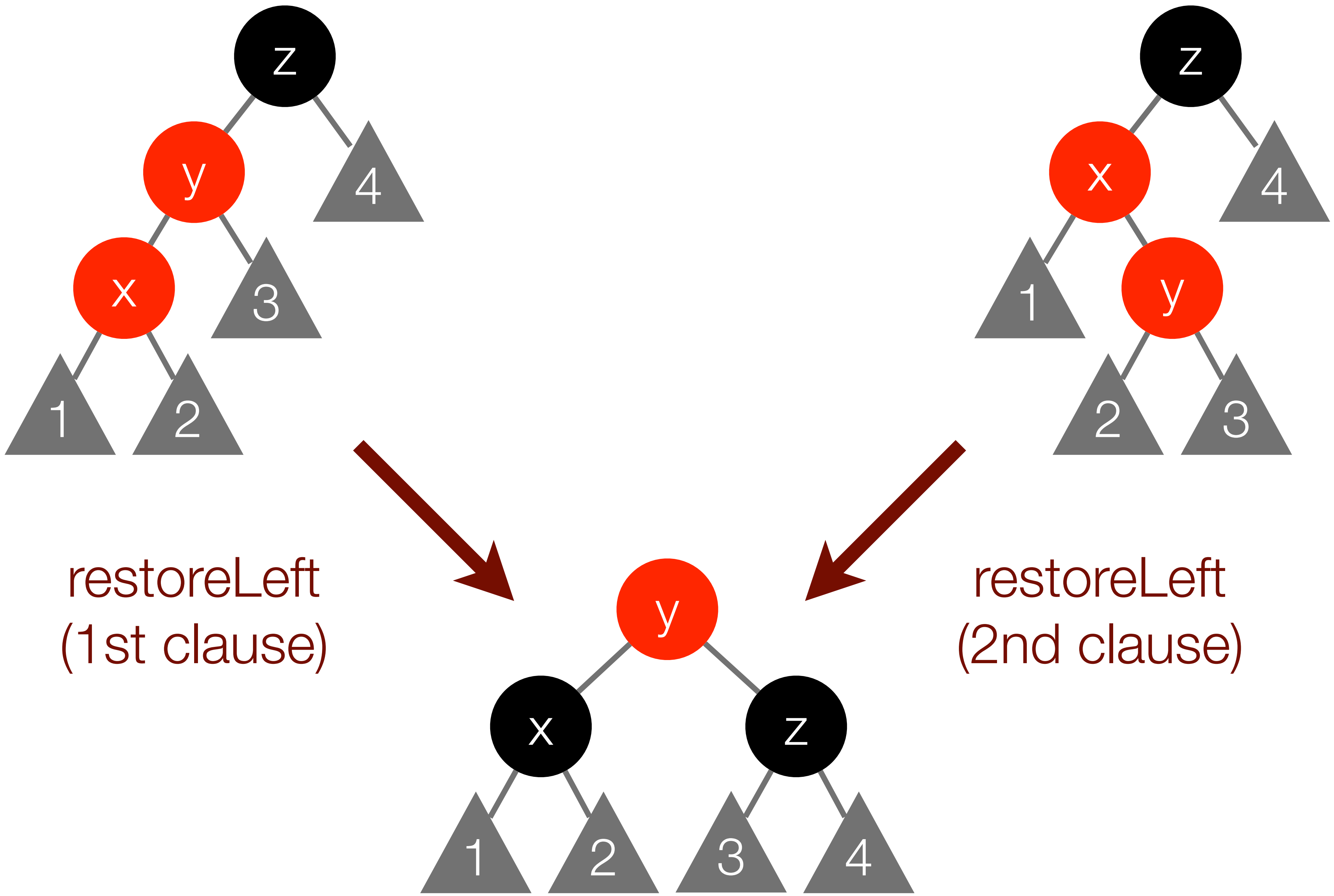




restoreLeft
(1st clause)

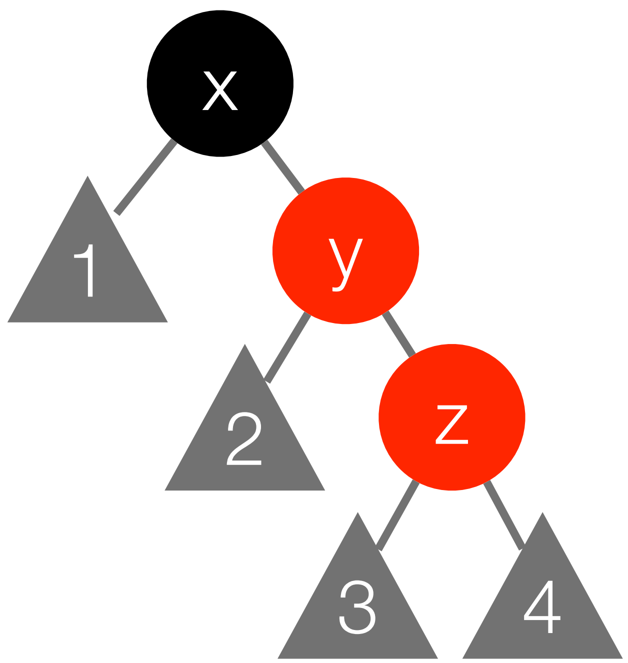


Two out of four possible rotations/re-coloring

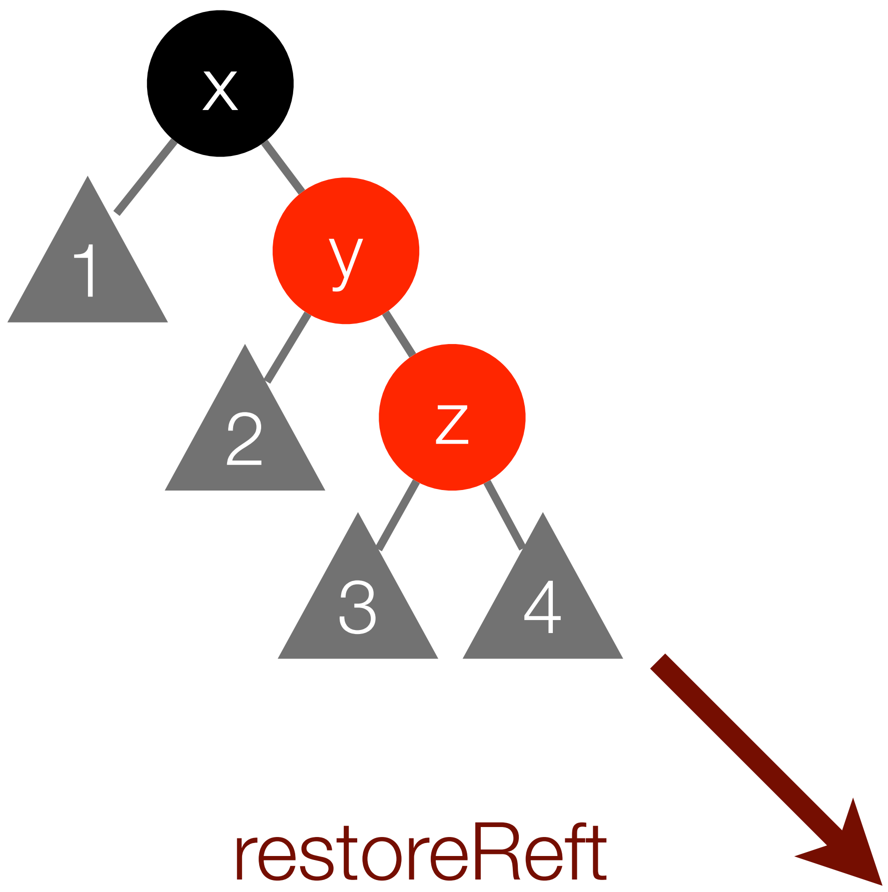


The other two possible rotations/re-coloring

The other two possible rotations/re-coloring

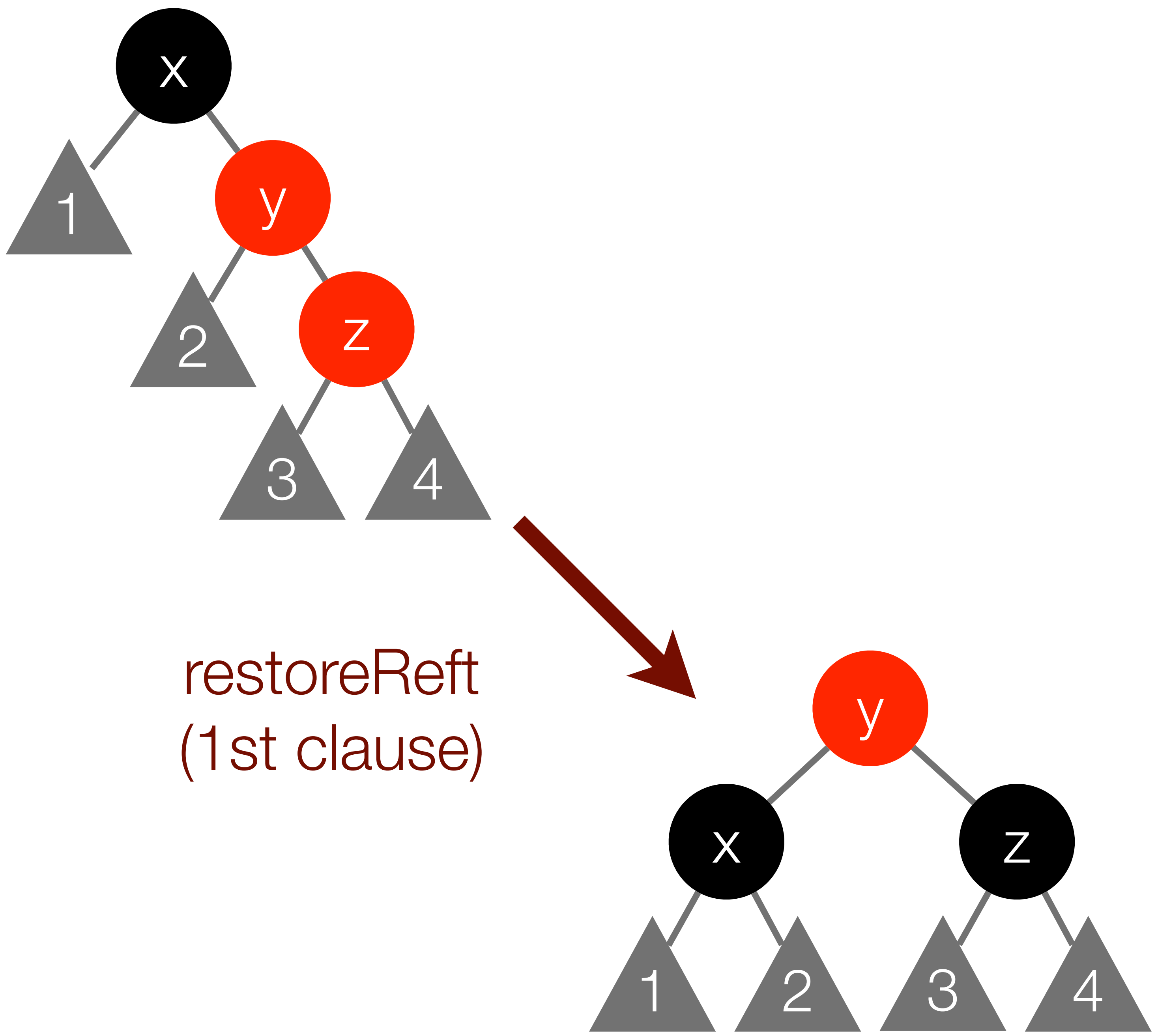


The other two possible rotations/re-coloring

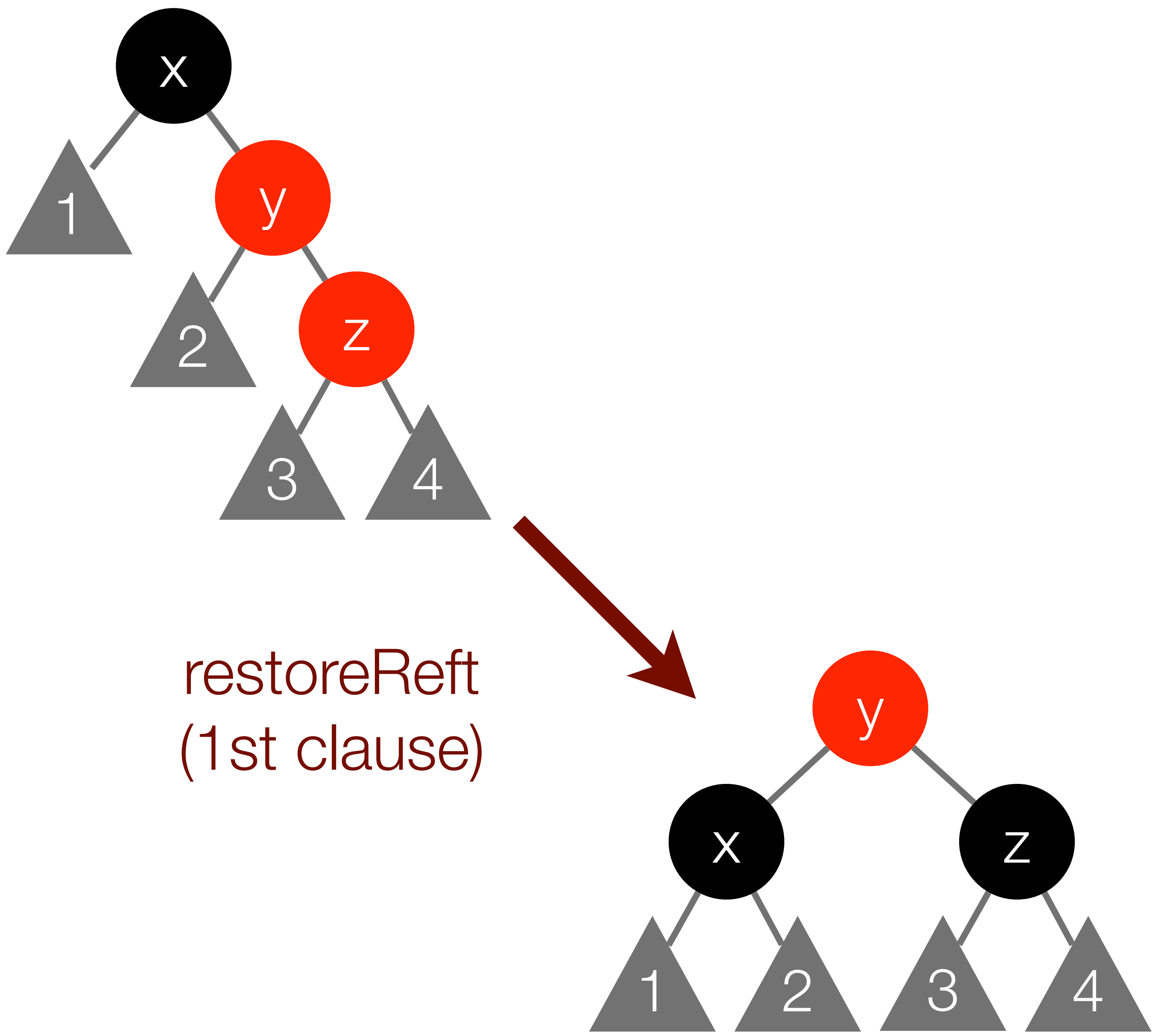


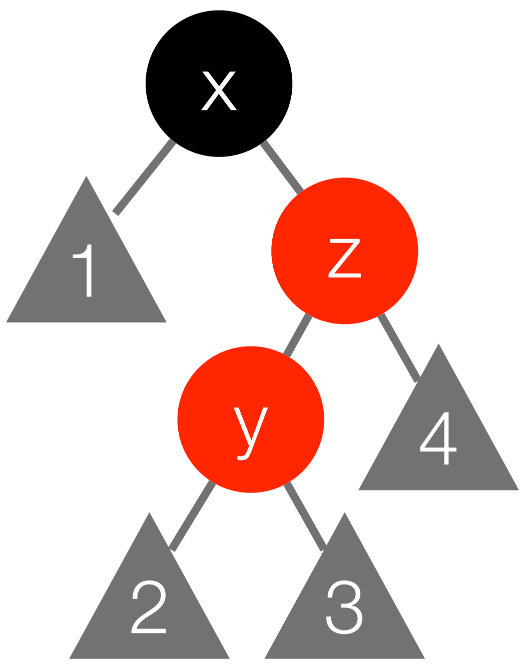
restoreReft
(1st clause)

The other two possible rotations/re-coloring

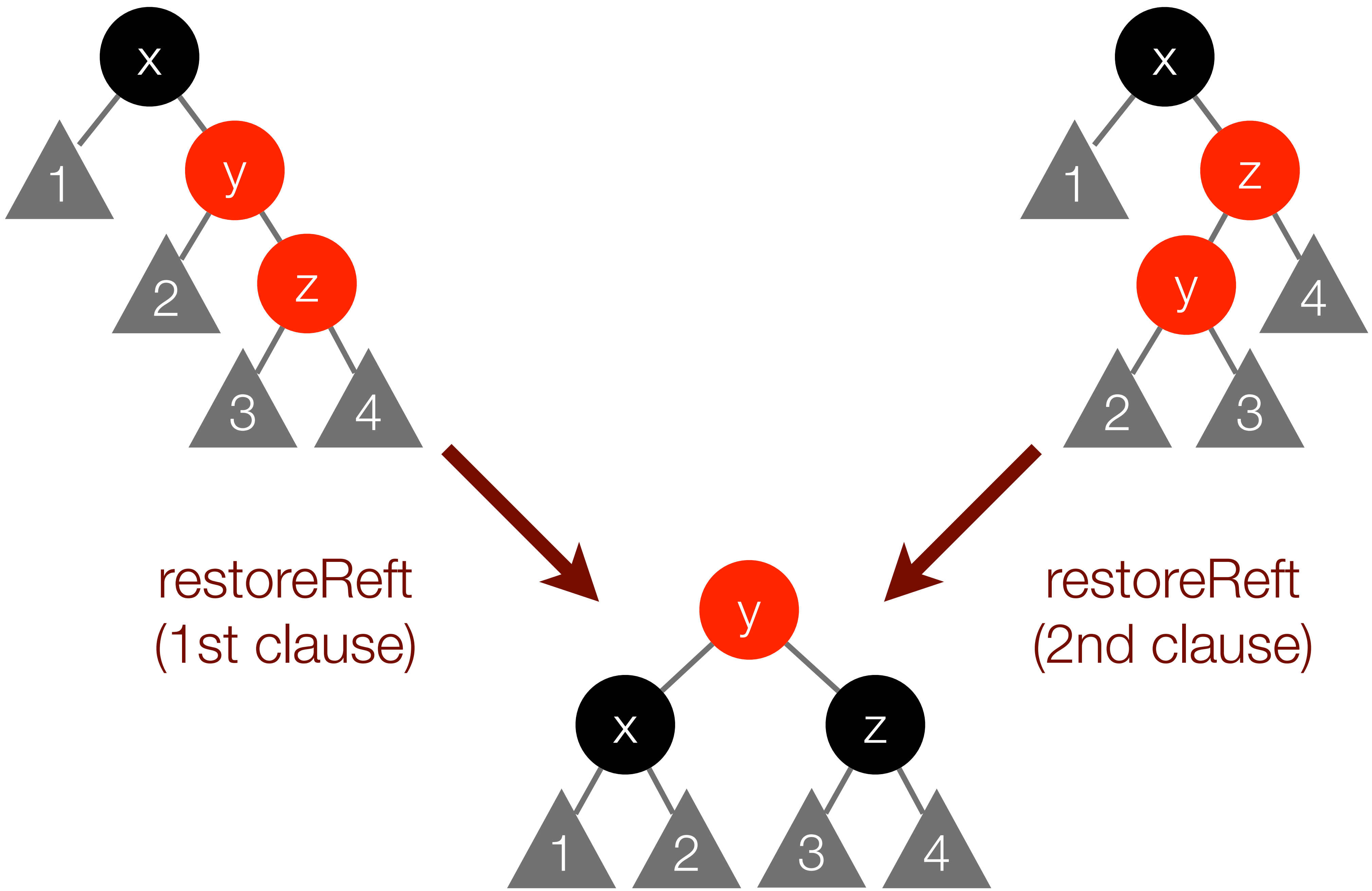


The other two possible rotations/re-coloring



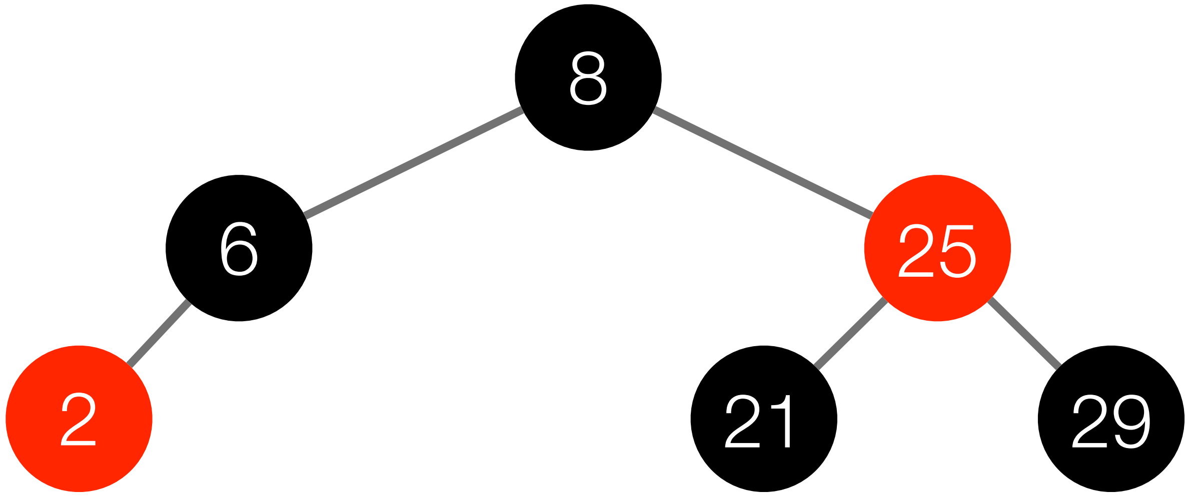


The other two possible rotations/re-coloring



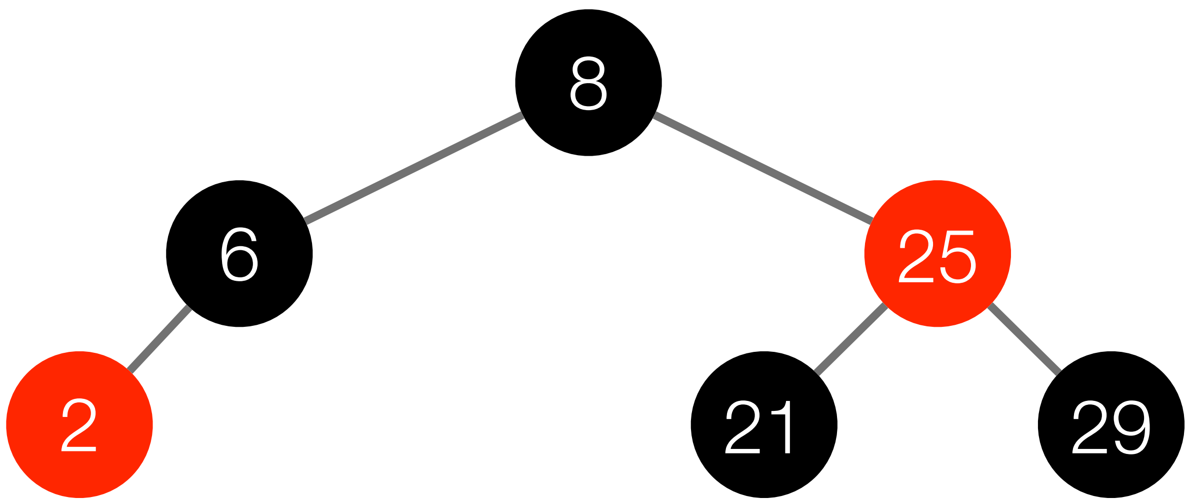
Let's look at another example

Let's look at another example



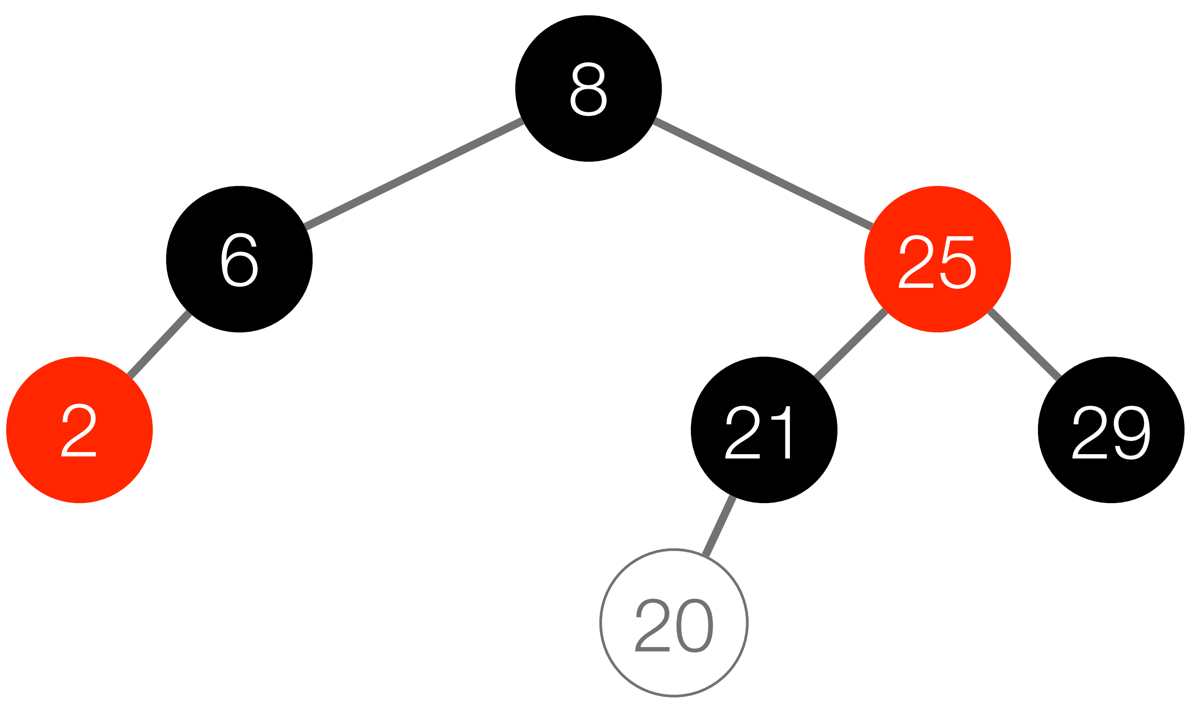
Let's look at another example

Let's insert 20:



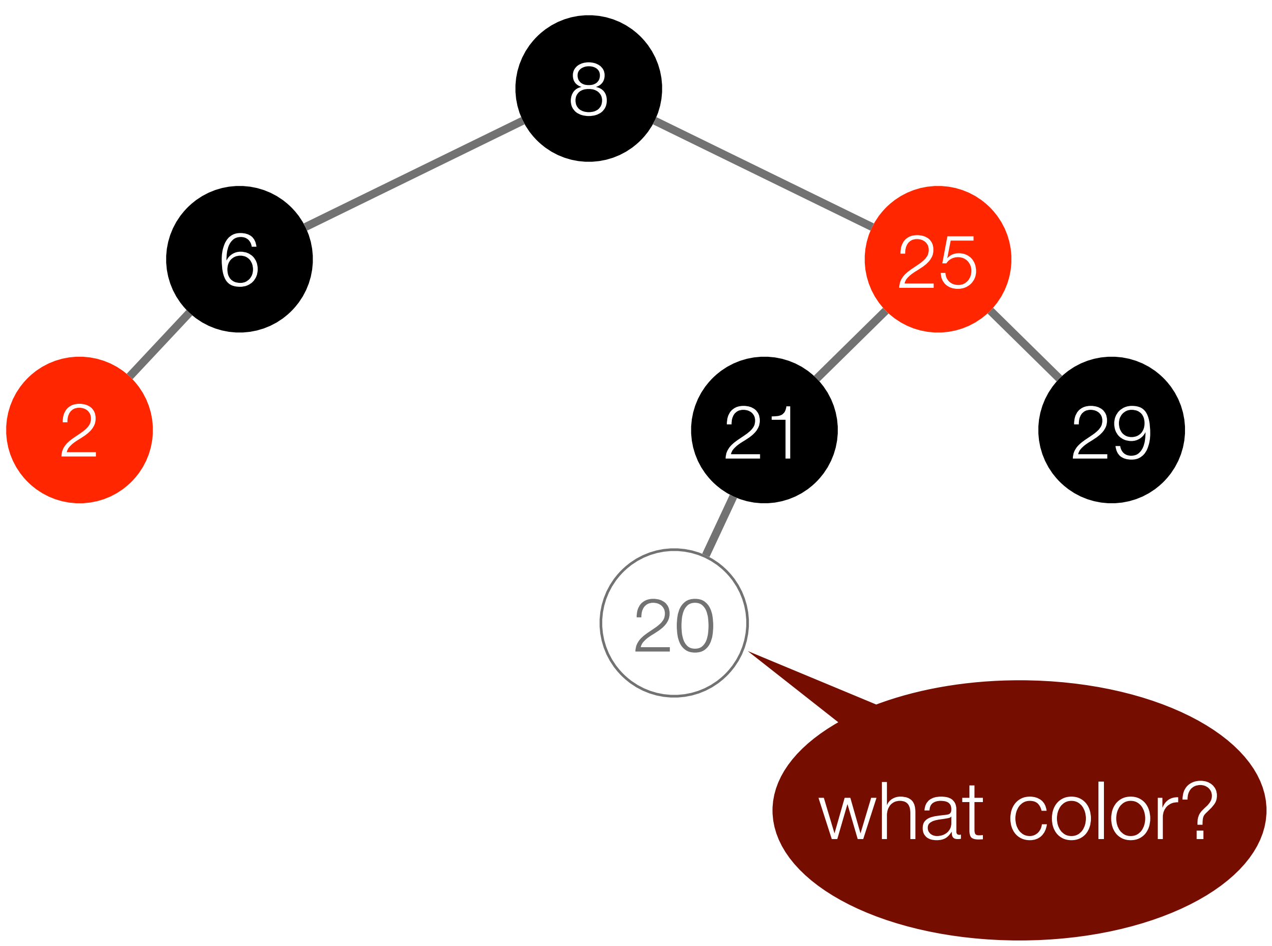
Let's look at another example

Let's insert 20:



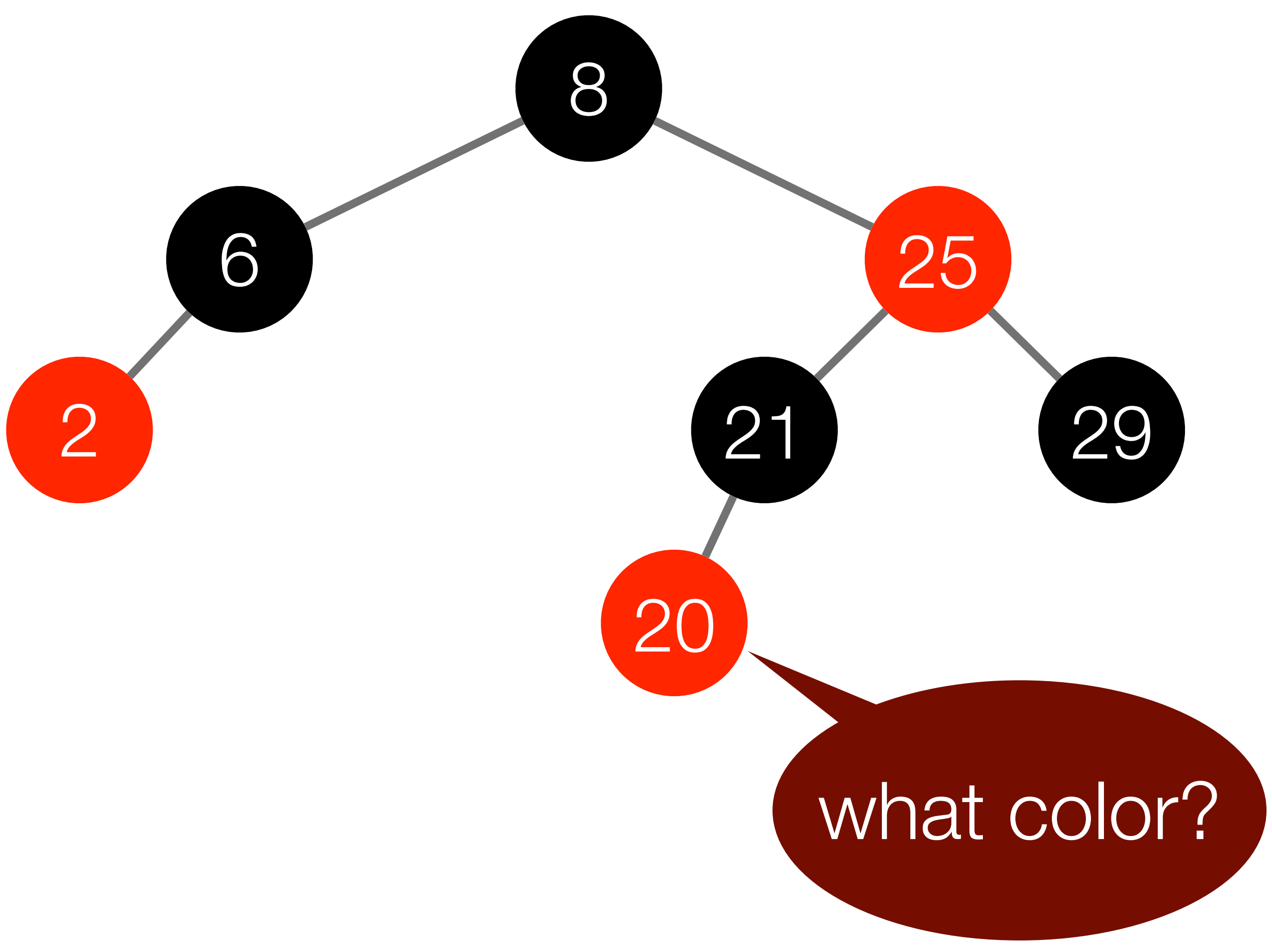
Let's look at another example

Let's insert 20:



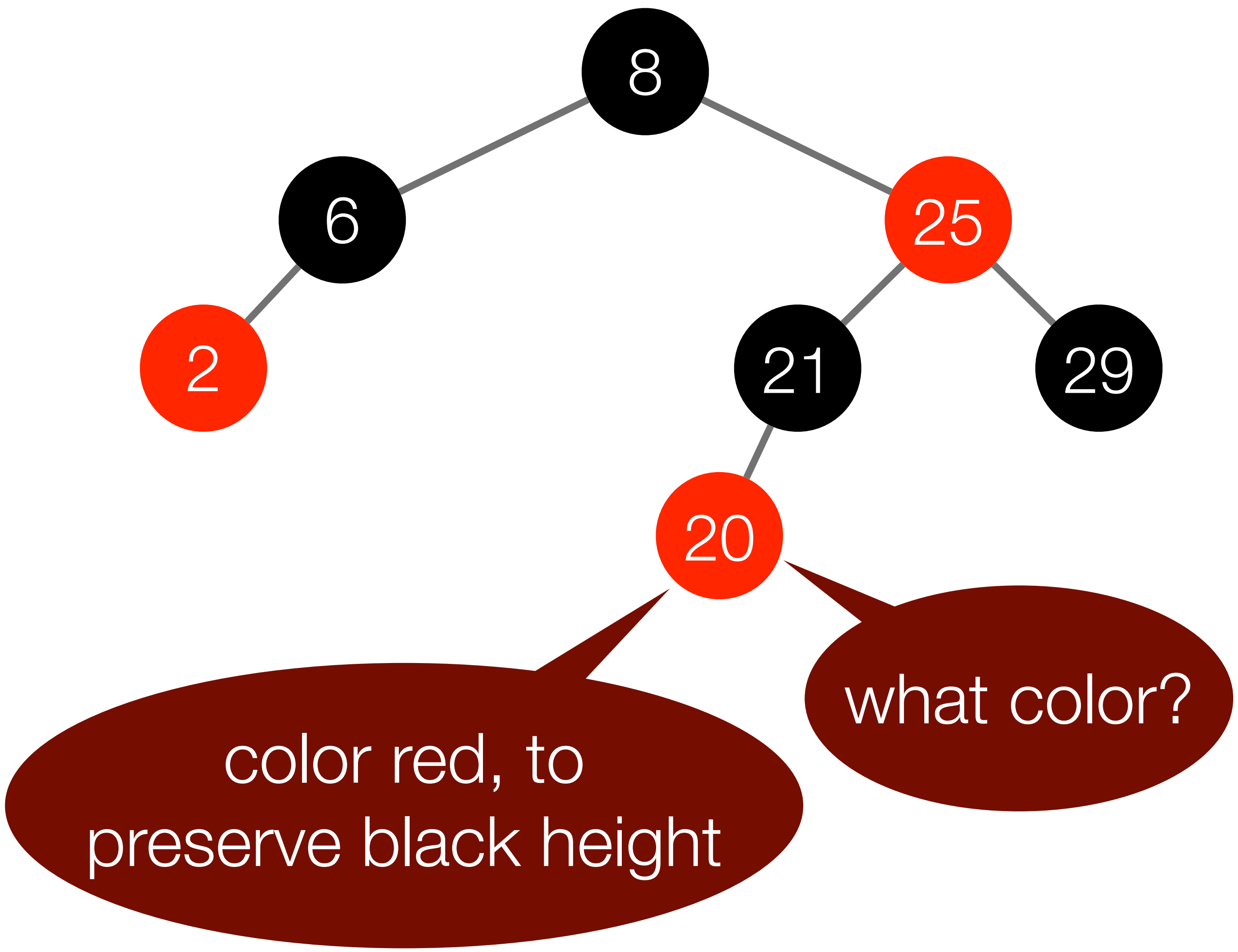
Let's look at another example

Let's insert 20:

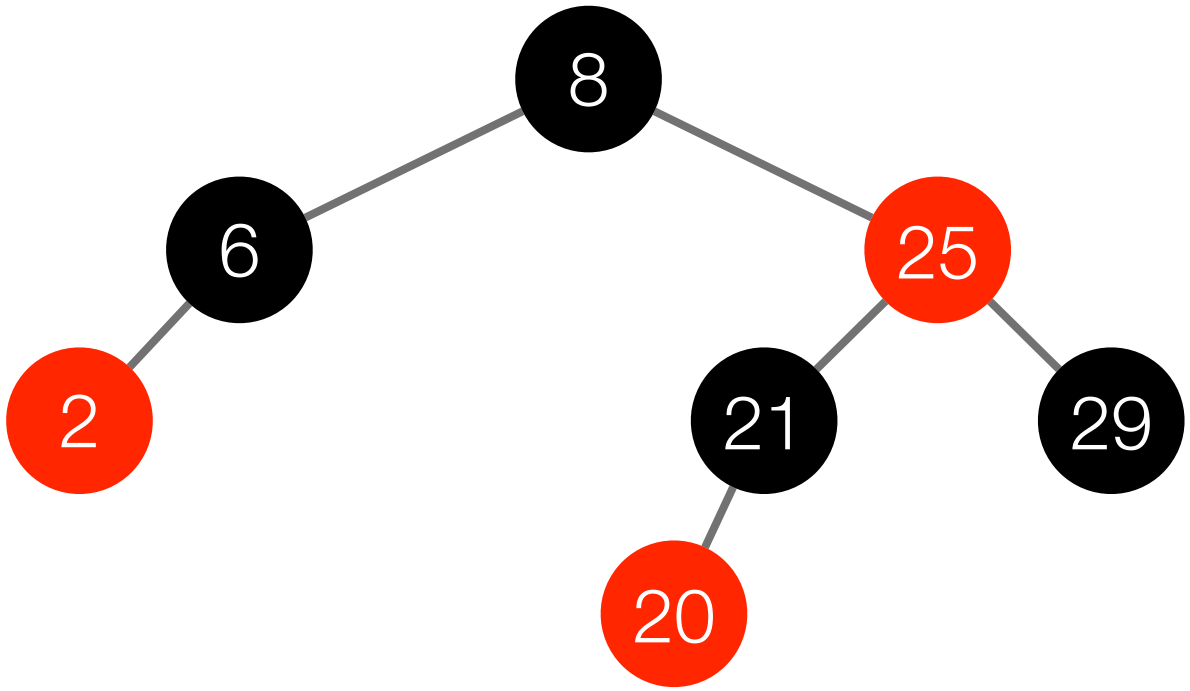


Let's look at another example

Let's insert 20:

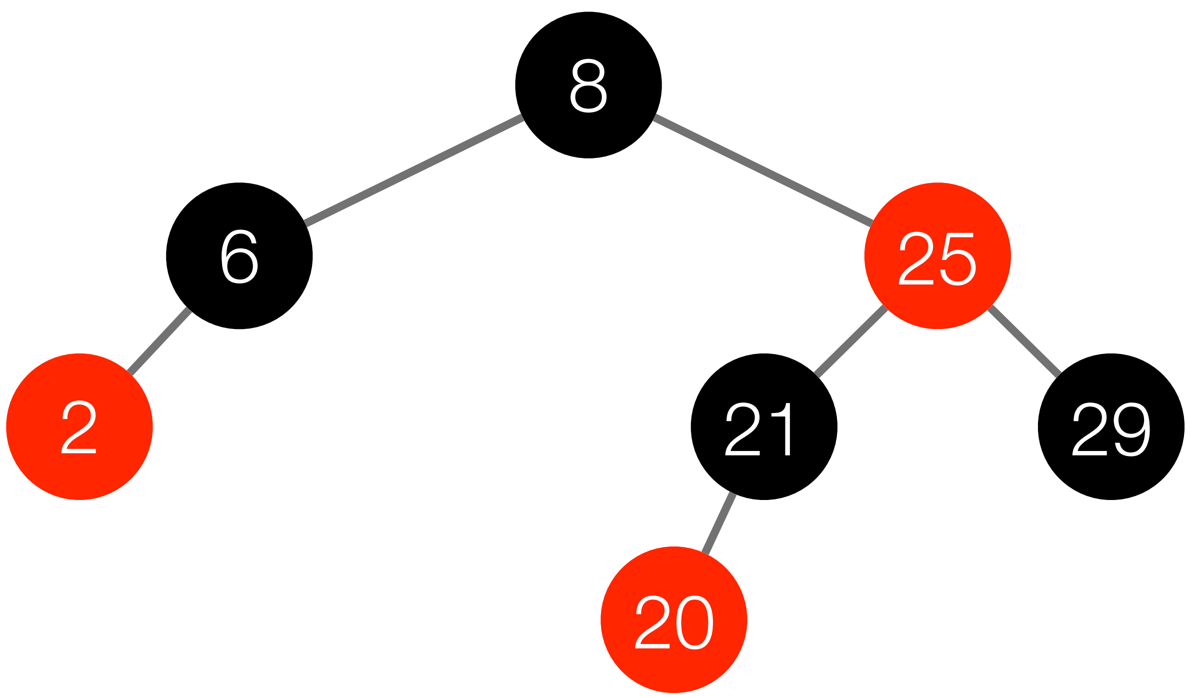


Let's look at another example



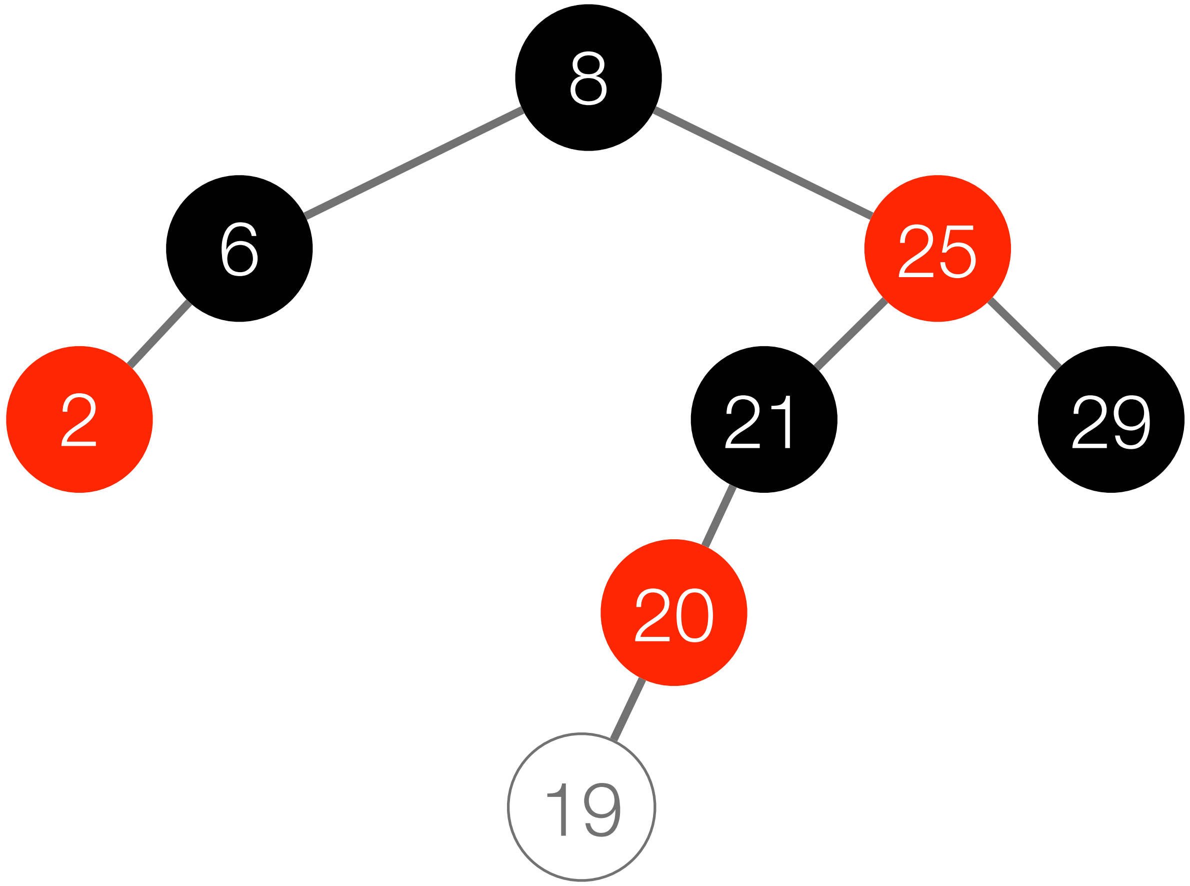
Let's look at another example

Now, let's insert 19:



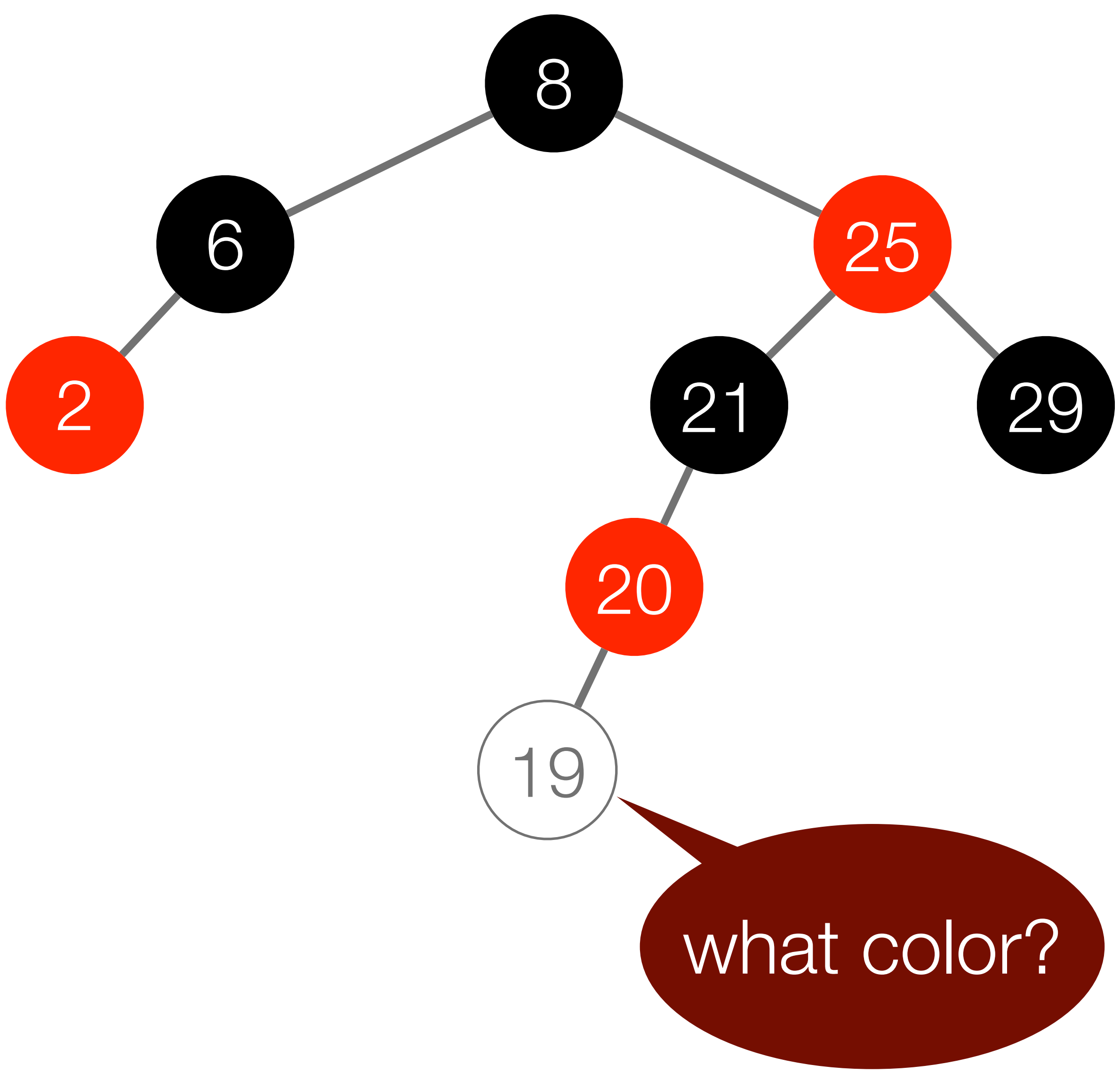
Let's look at another example

Now, let's insert 19:



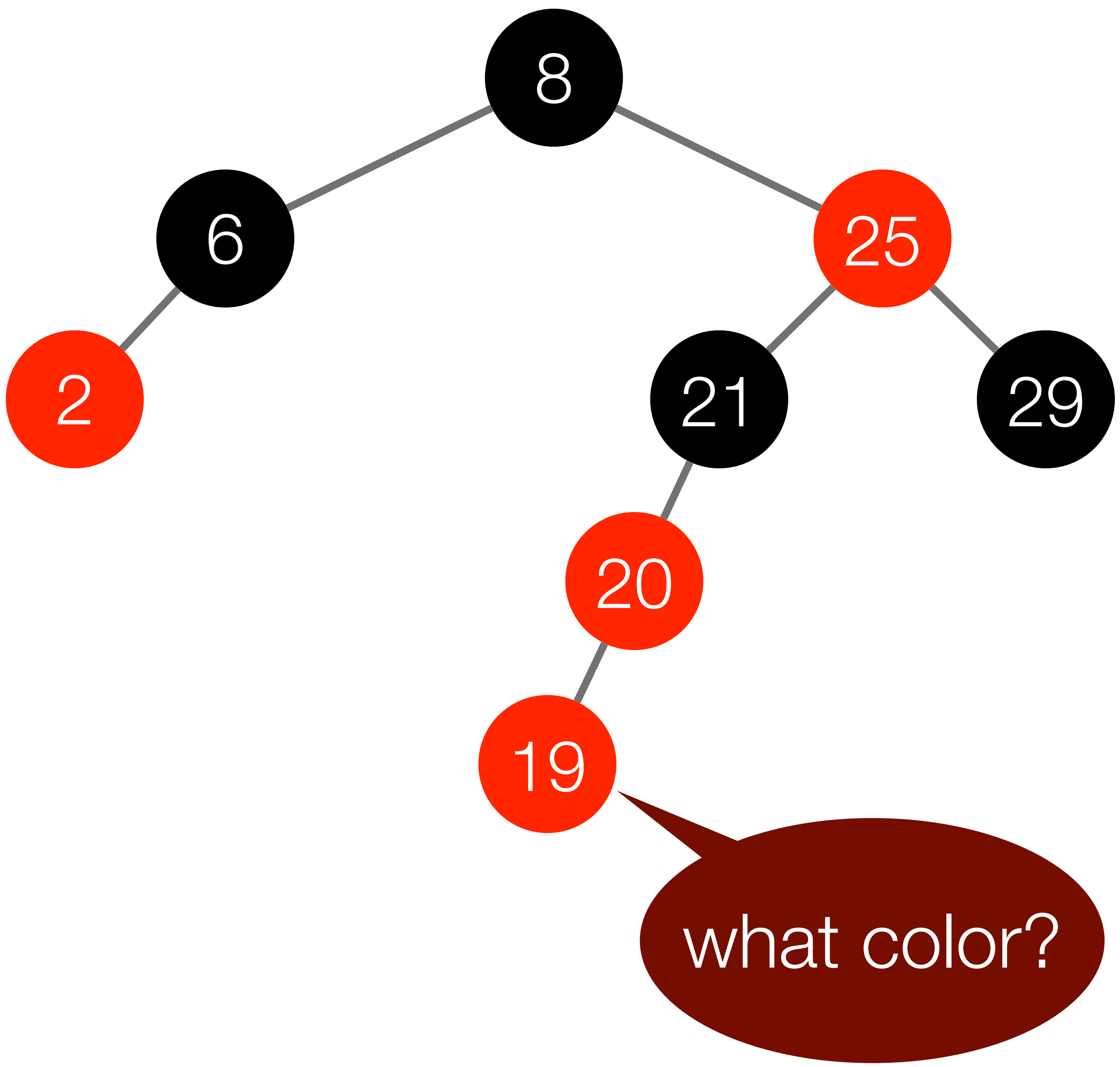
Let's look at another example

Now, let's insert 19:



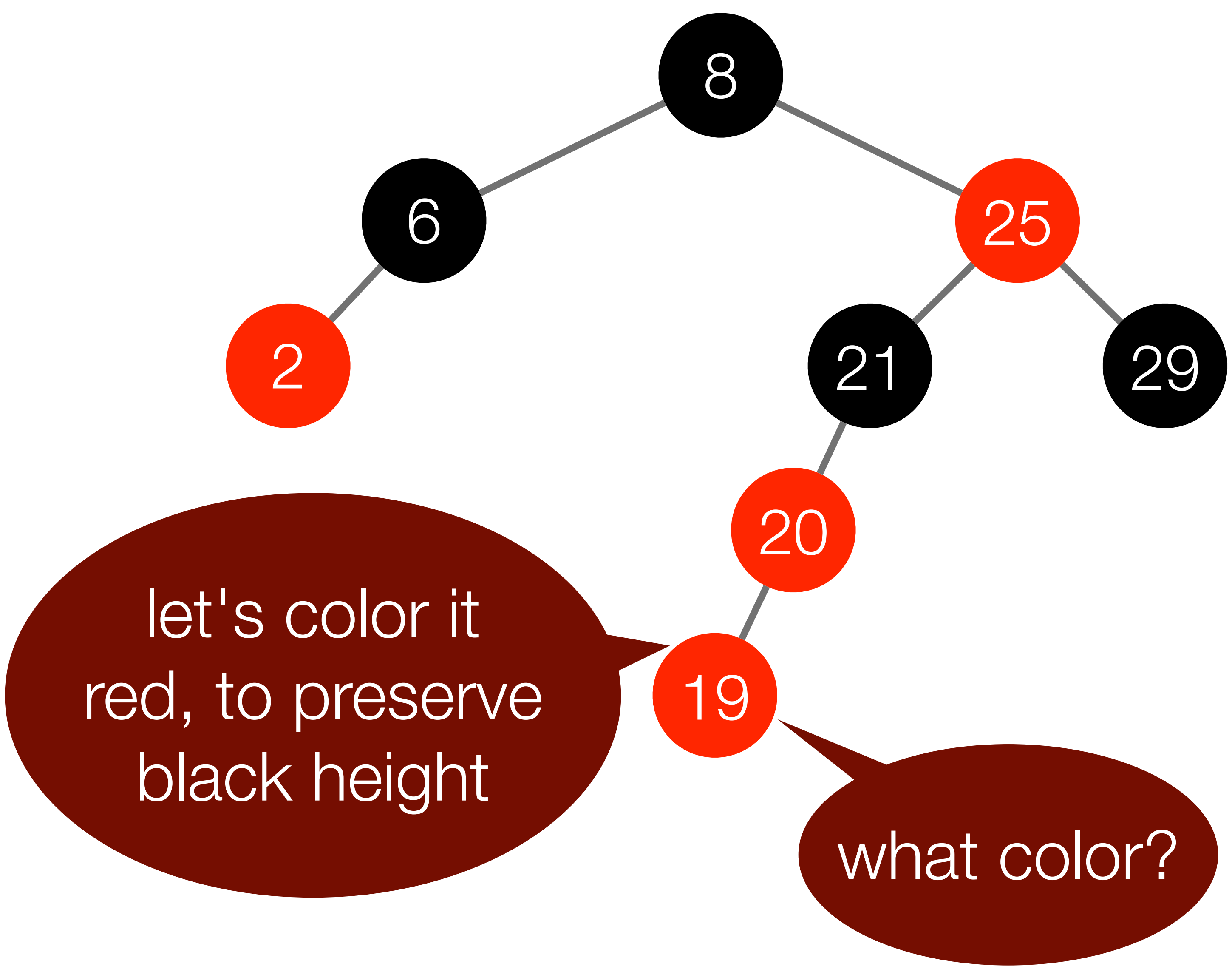
Let's look at another example

Now, let's insert 19:



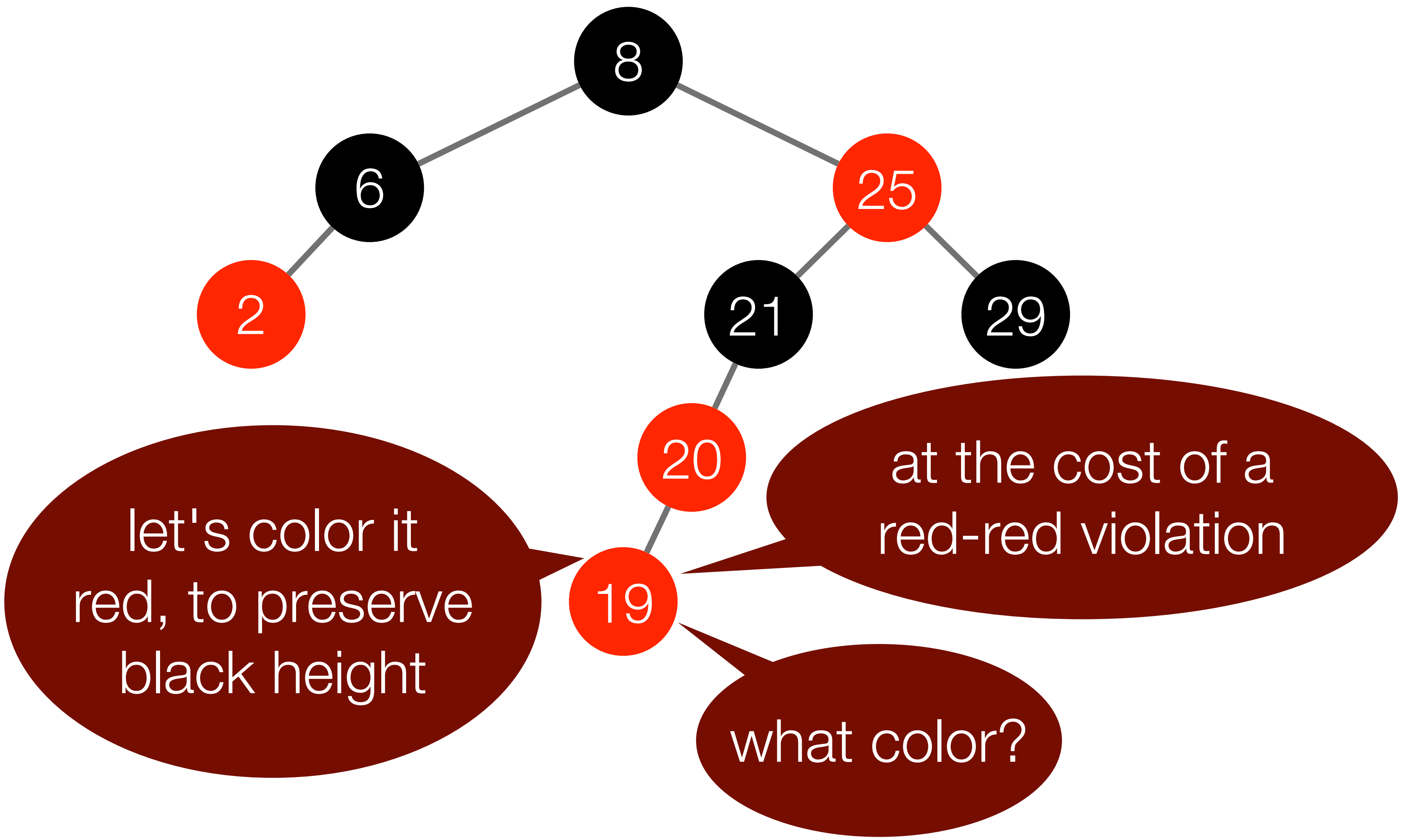
Let's look at another example

Now, let's insert 19:



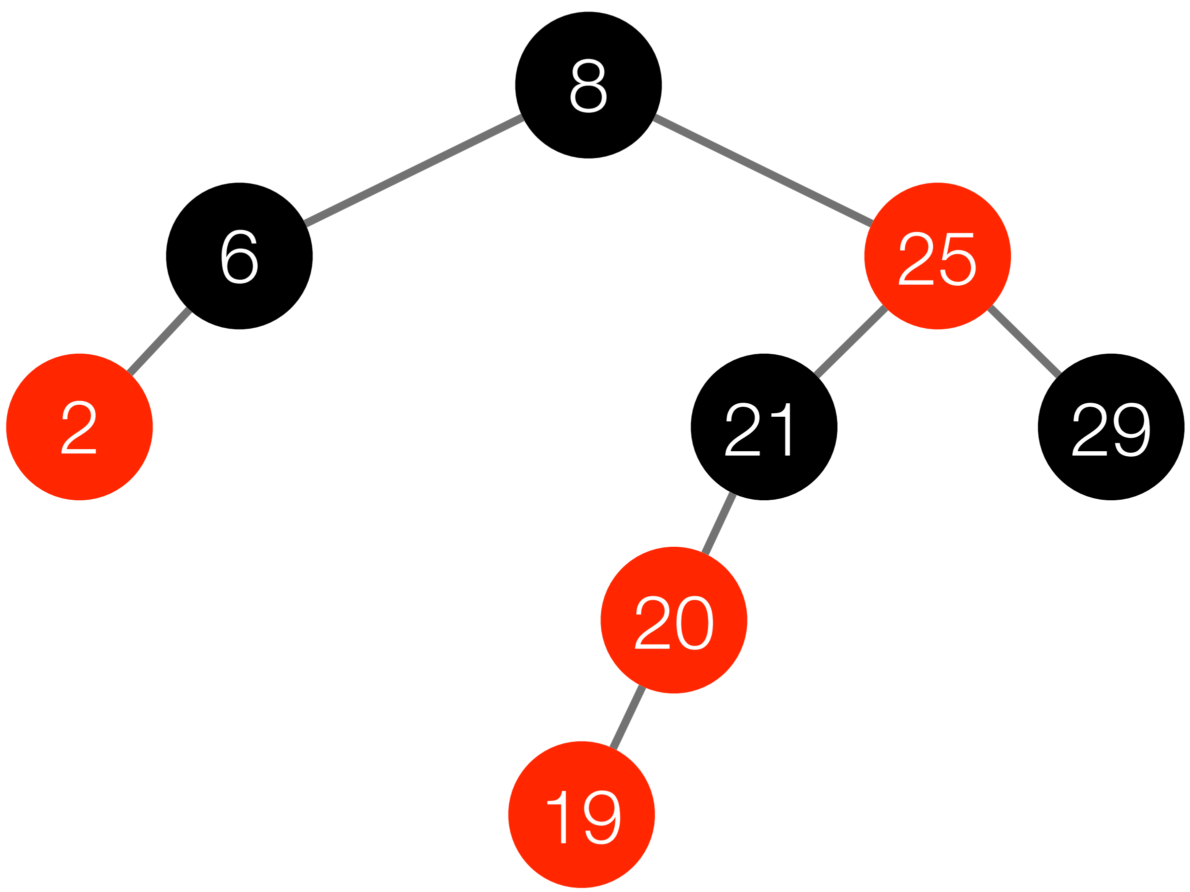
Let's look at another example

Now, let's insert 19:



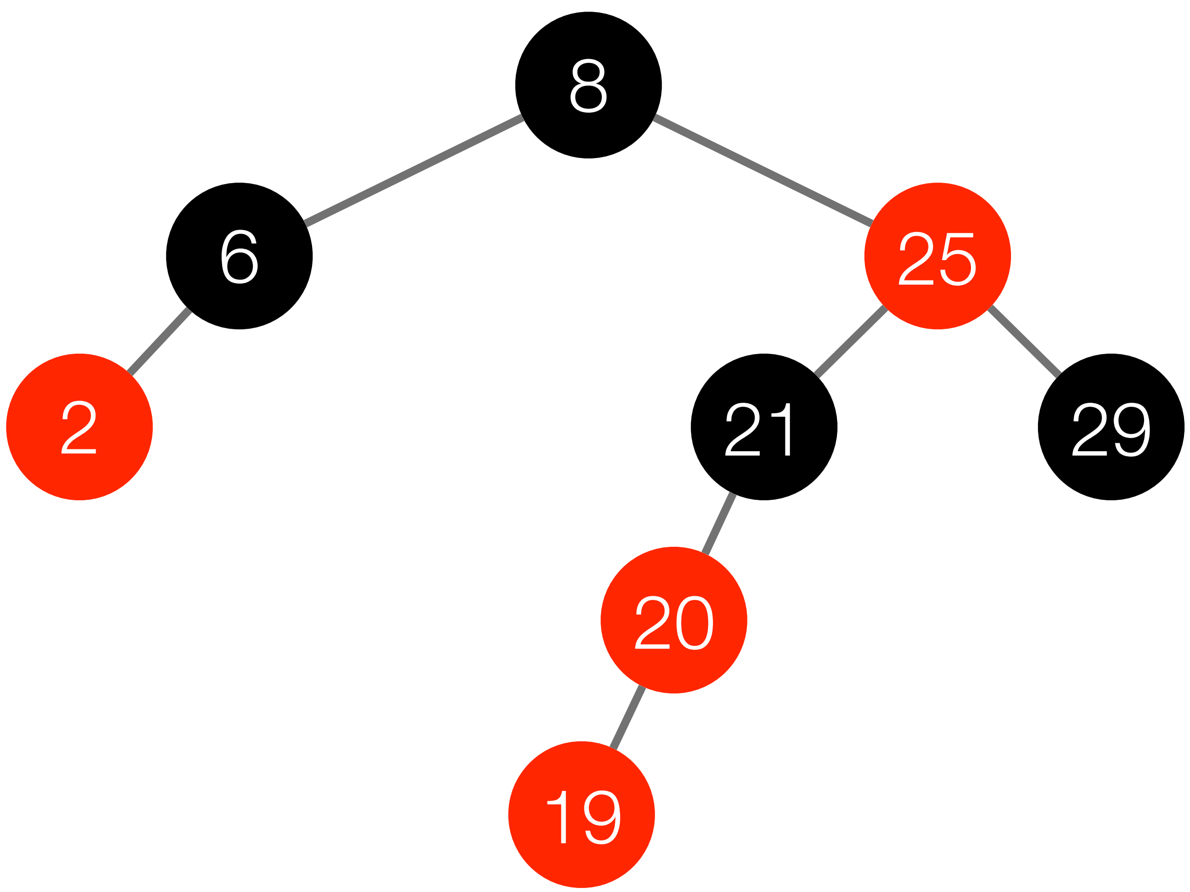
Let's look at another example

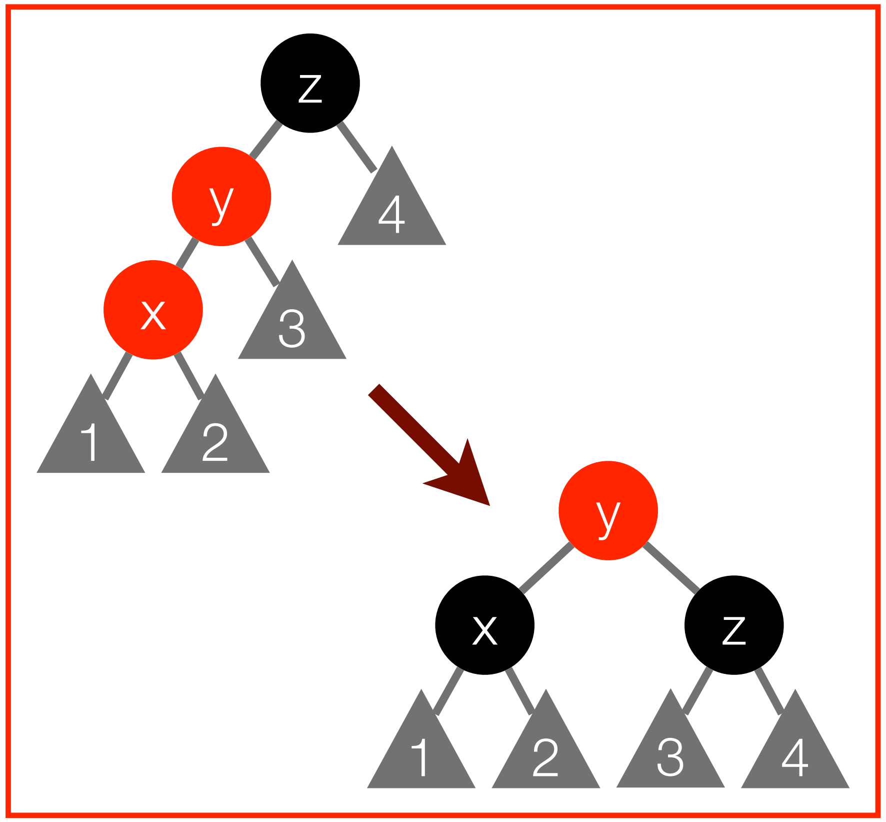
Now, let's insert 19:



Let's look at another example

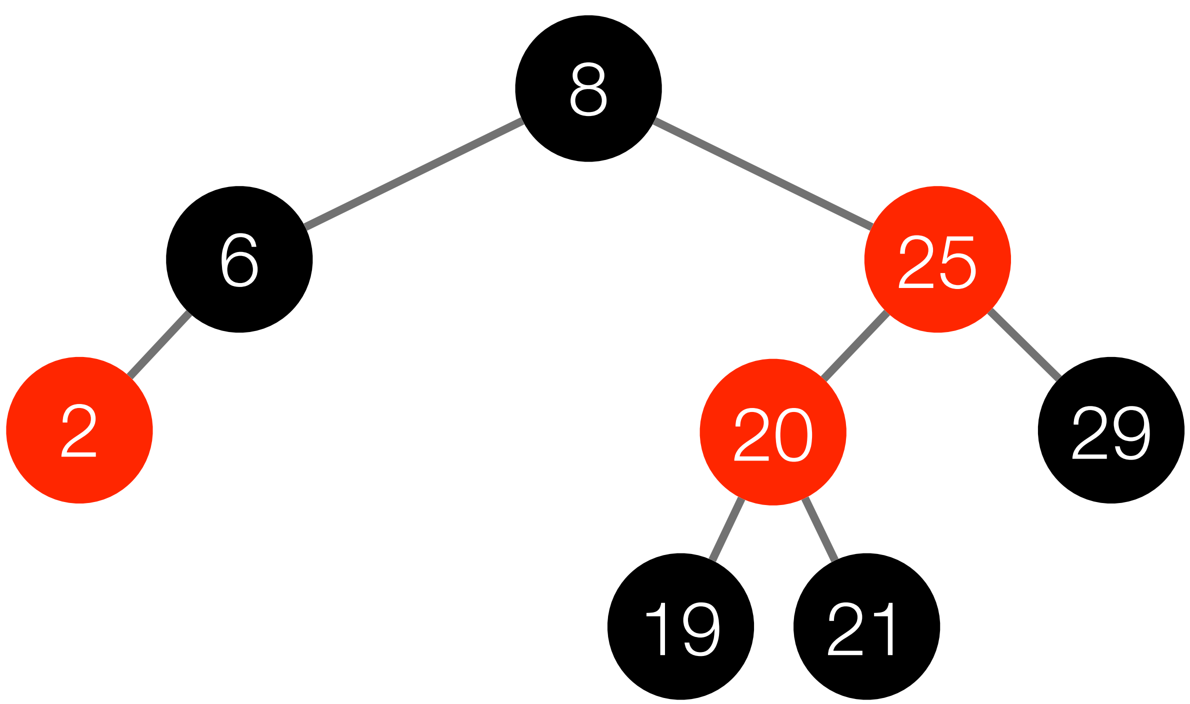
Now, let's insert 19:

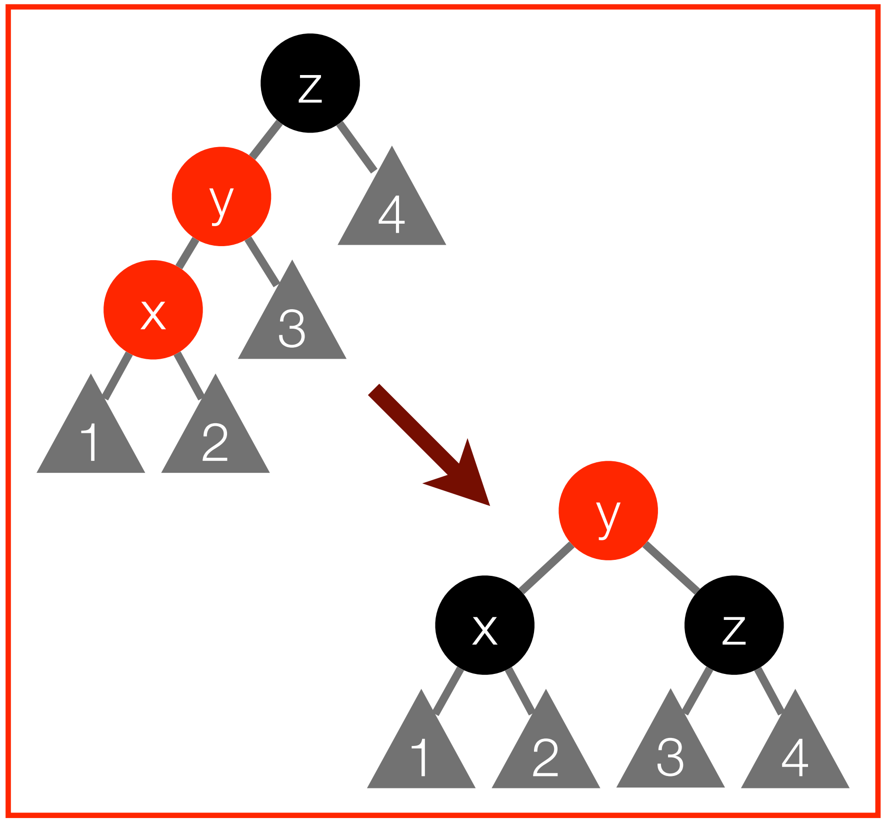




Let's look at another example

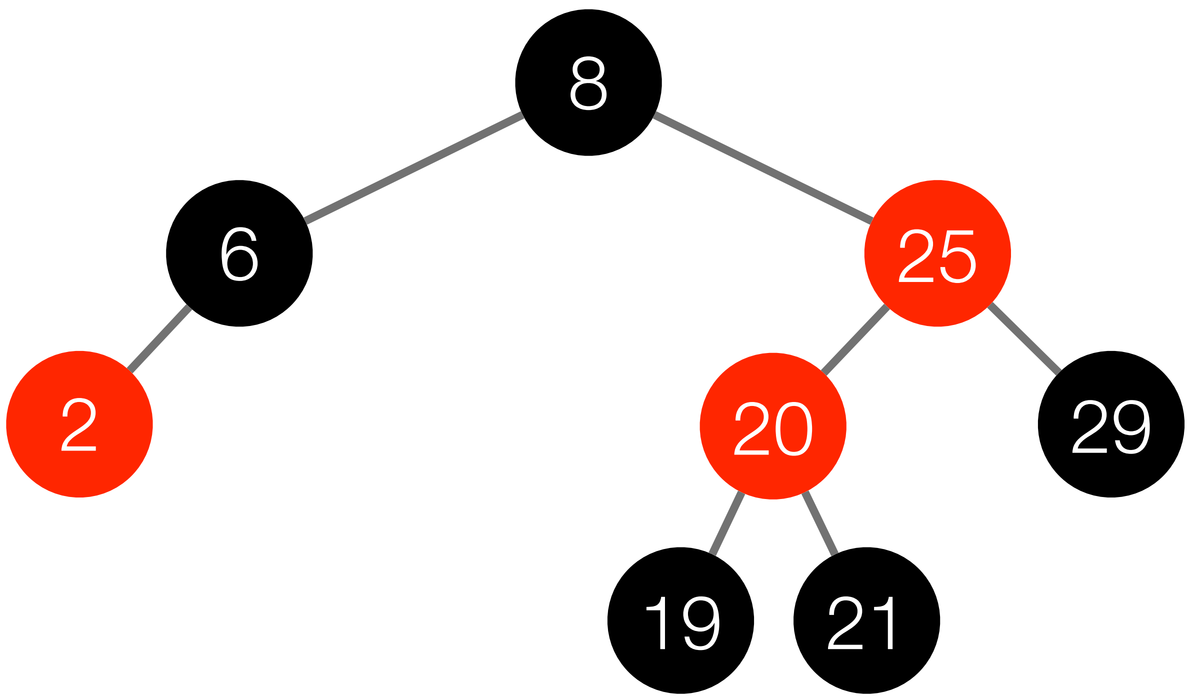
Now, let's insert 19:





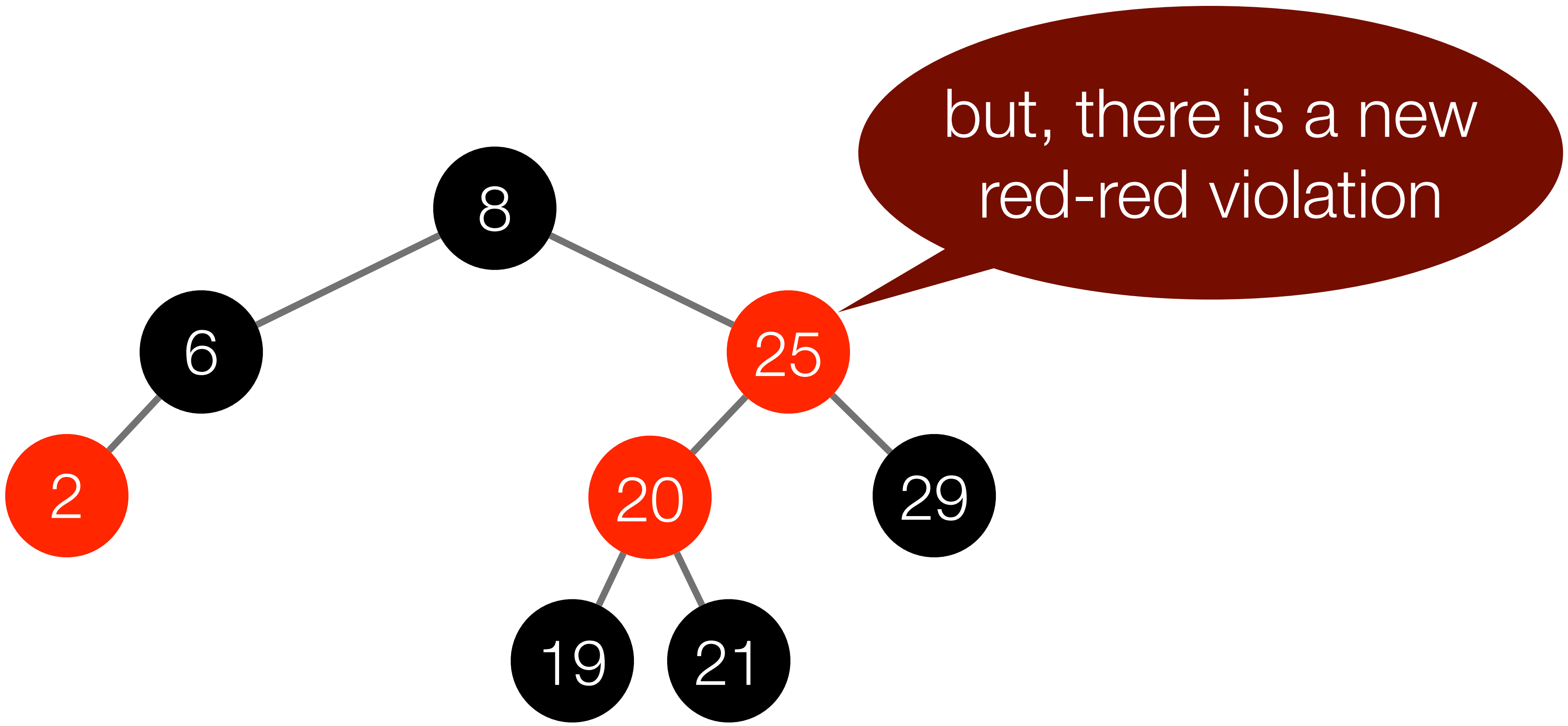
Let's look at another example

Now, let's insert 19:



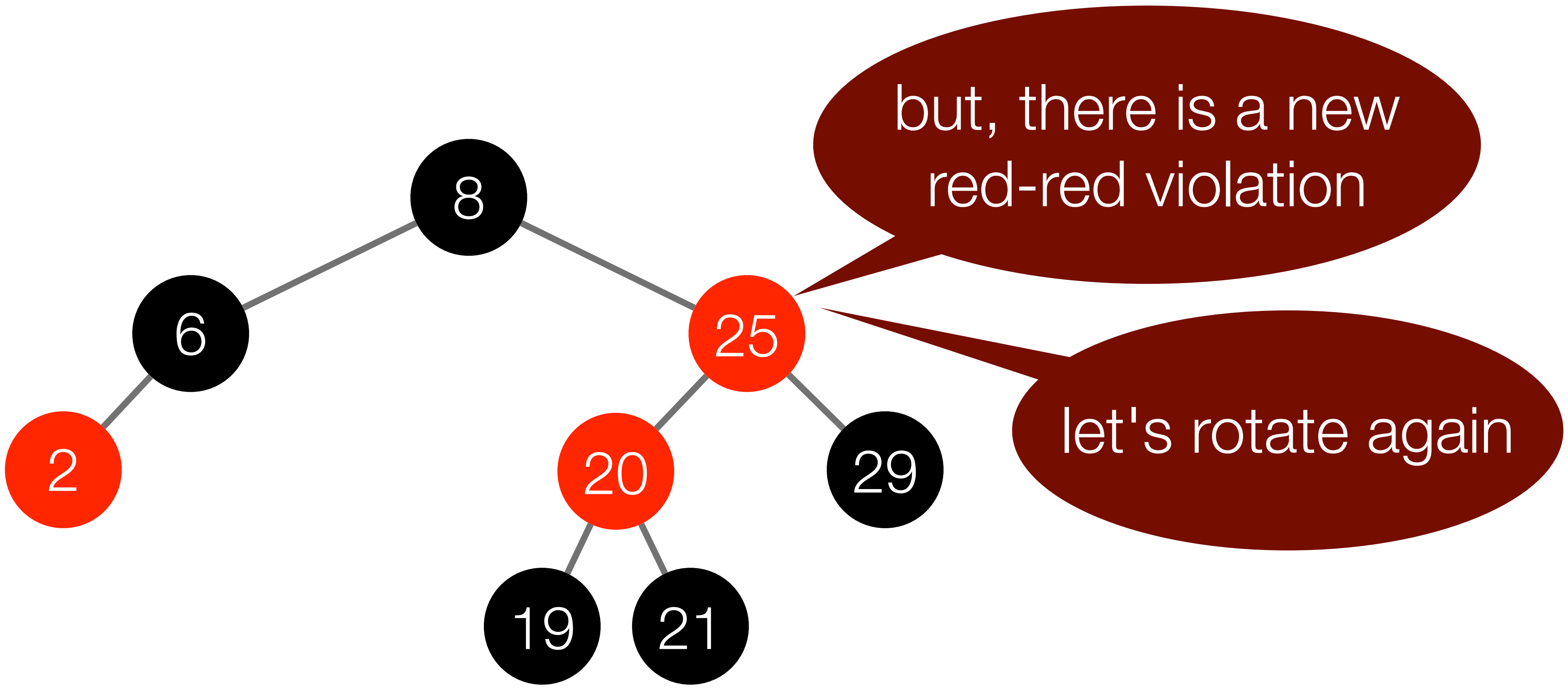
Let's look at another example

Now, let's insert 19:



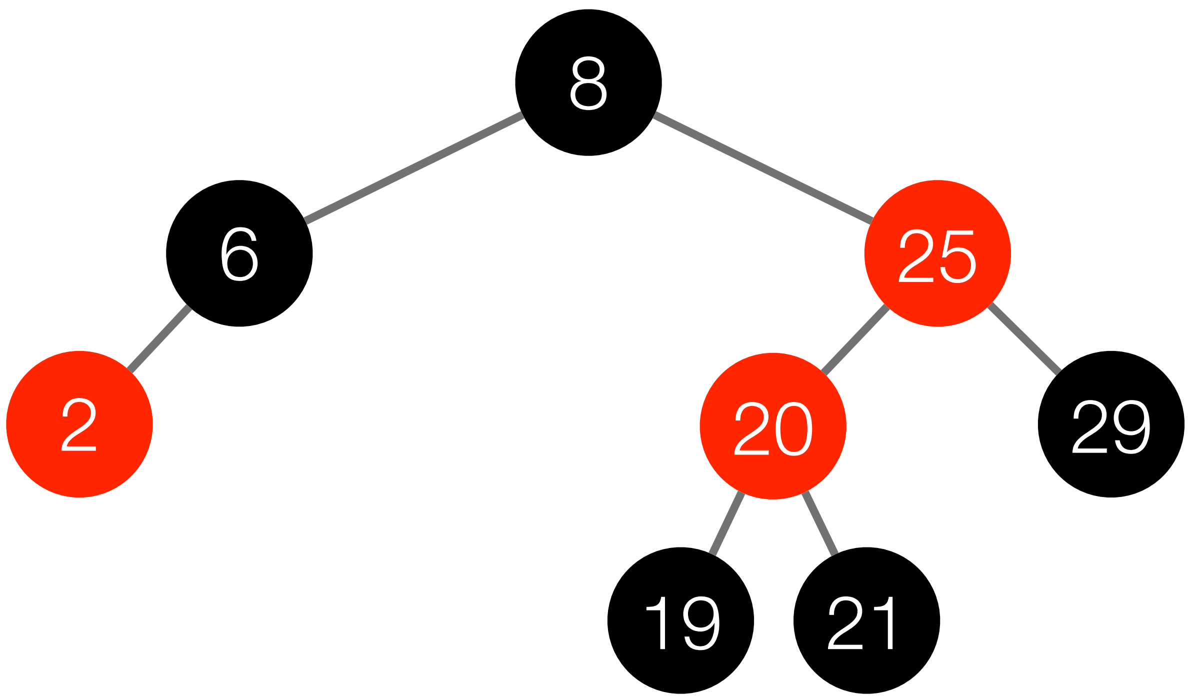
Let's look at another example

Now, let's insert 19:



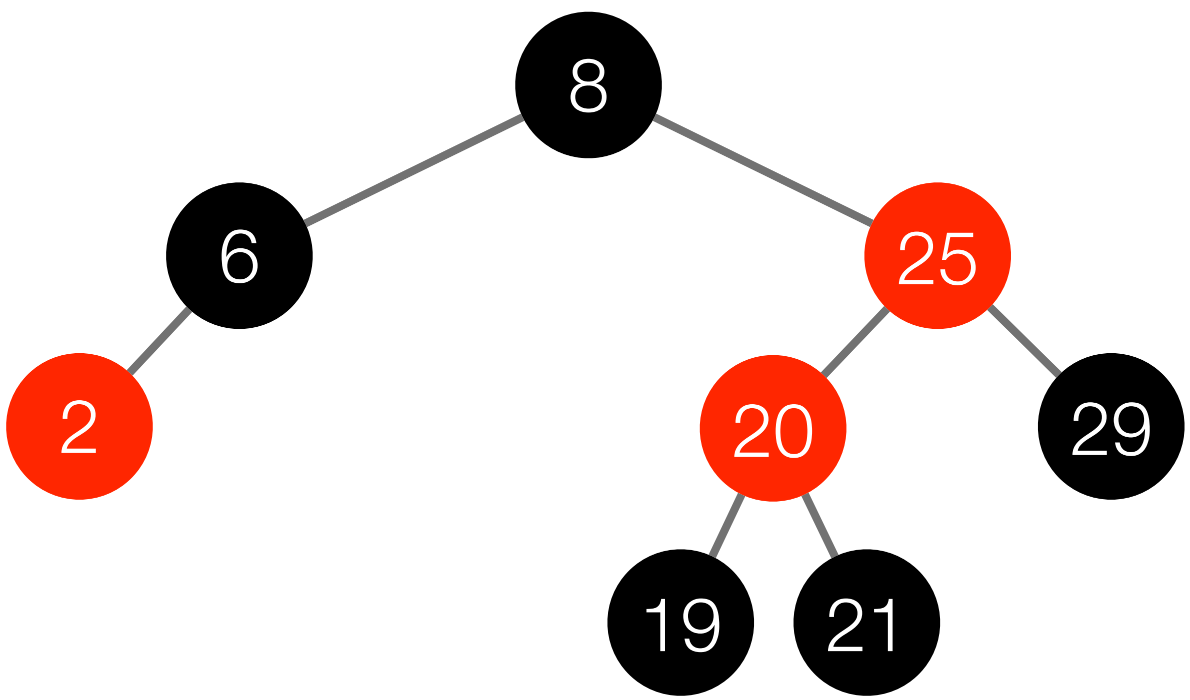
Let's look at another example

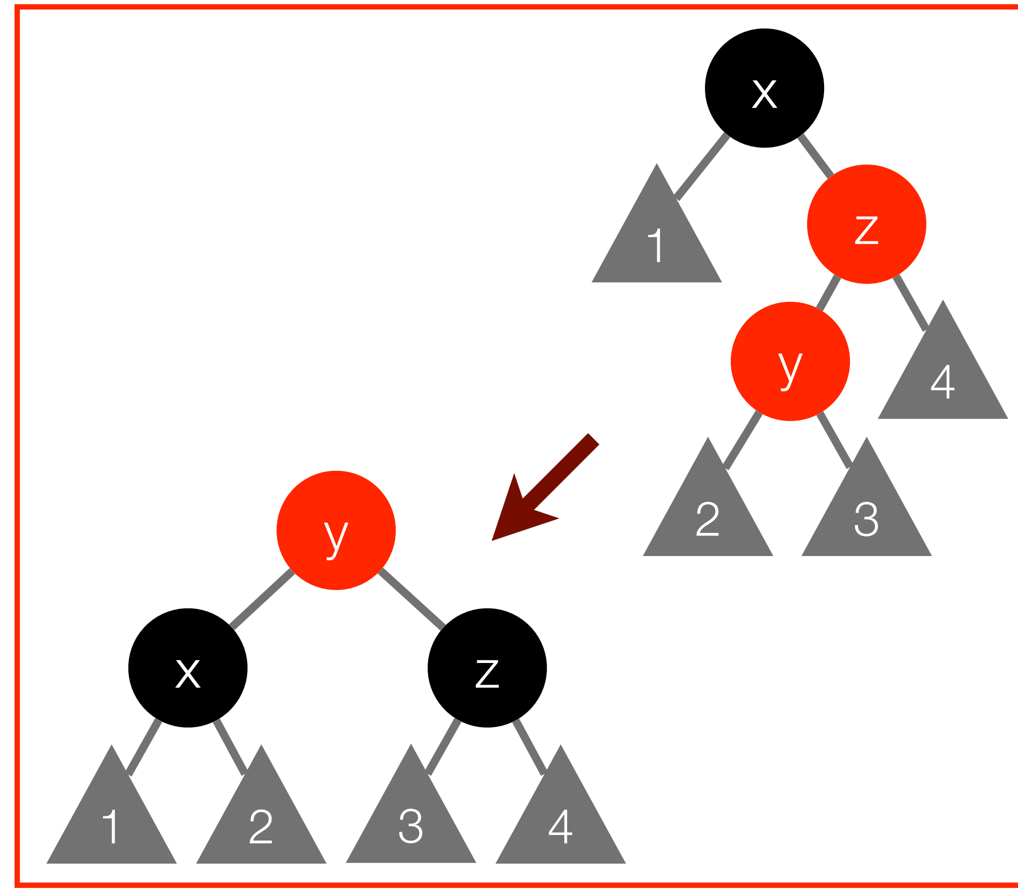
Now, let's insert 19:



Let's look at another example

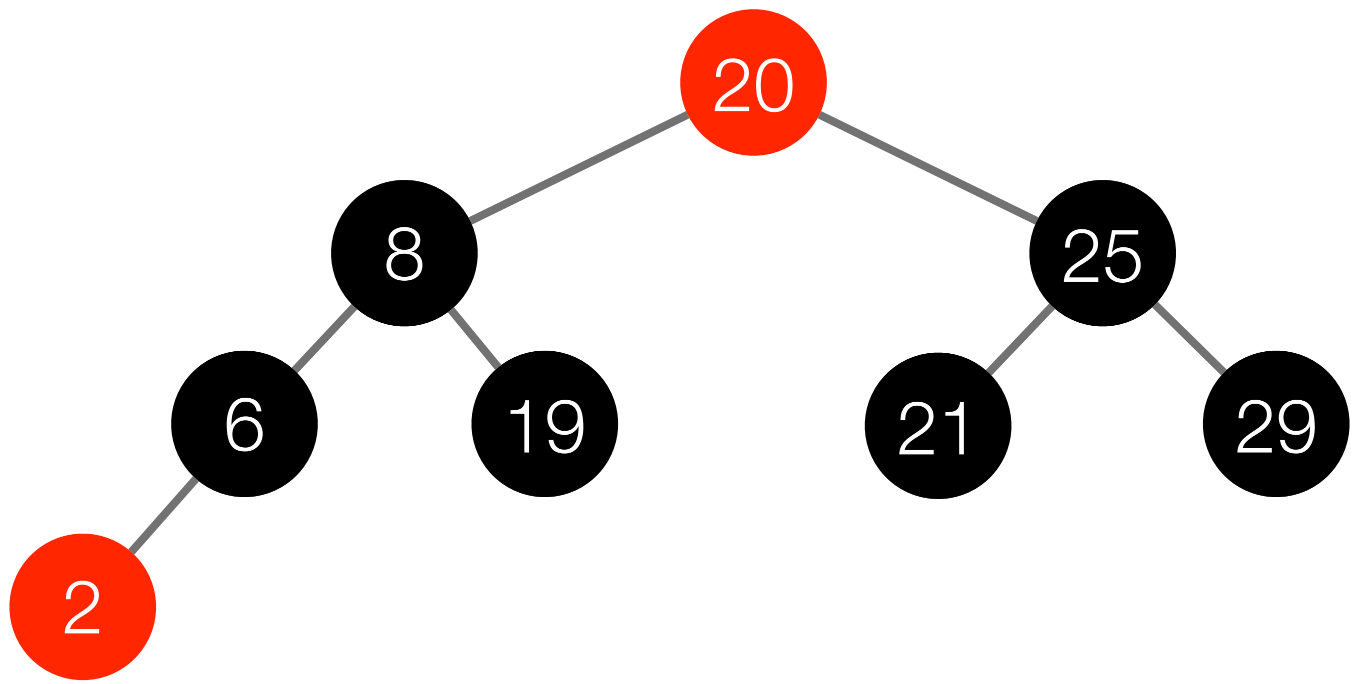
Now, let's insert 19:

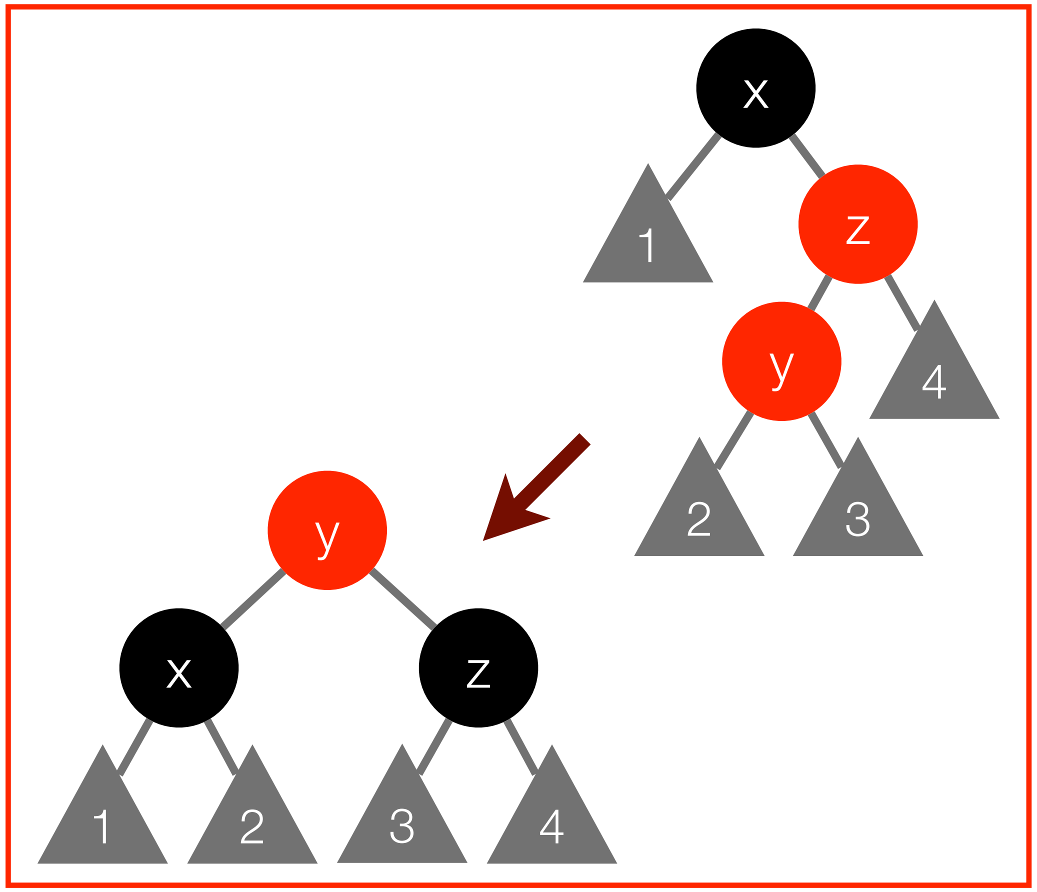




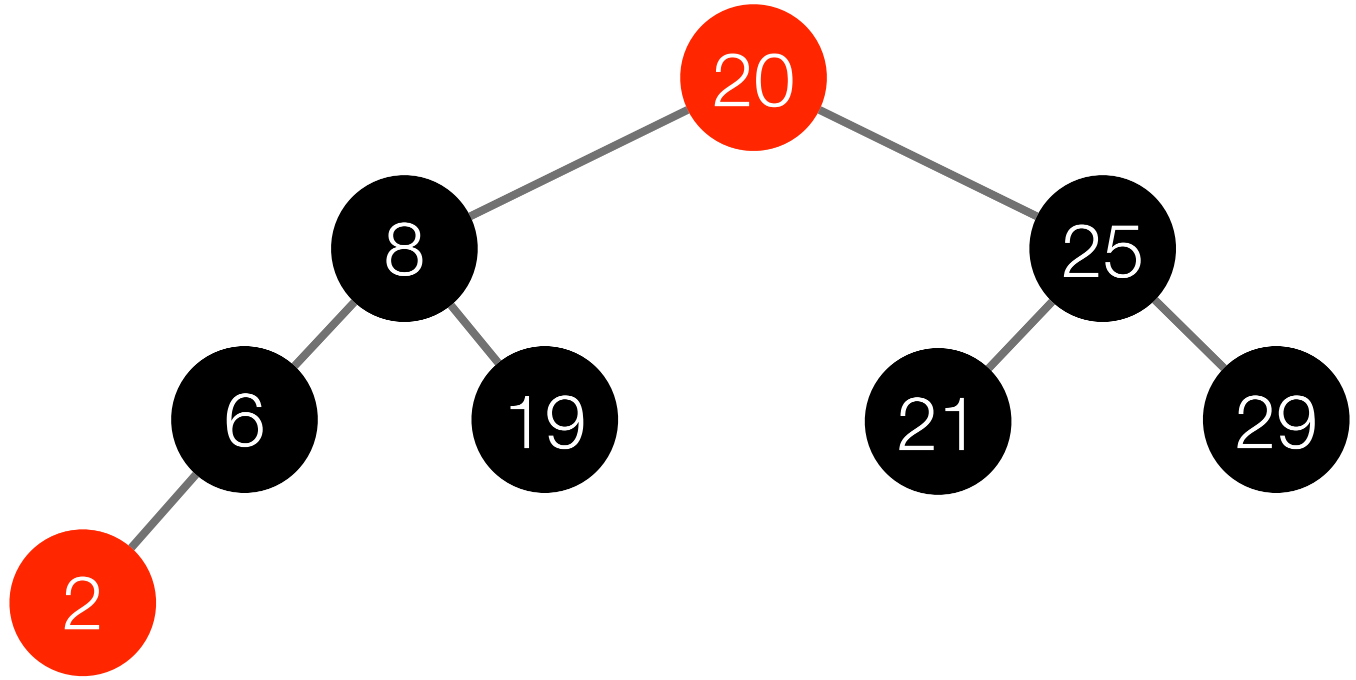
Let's look at another example

Now, let's insert 19:



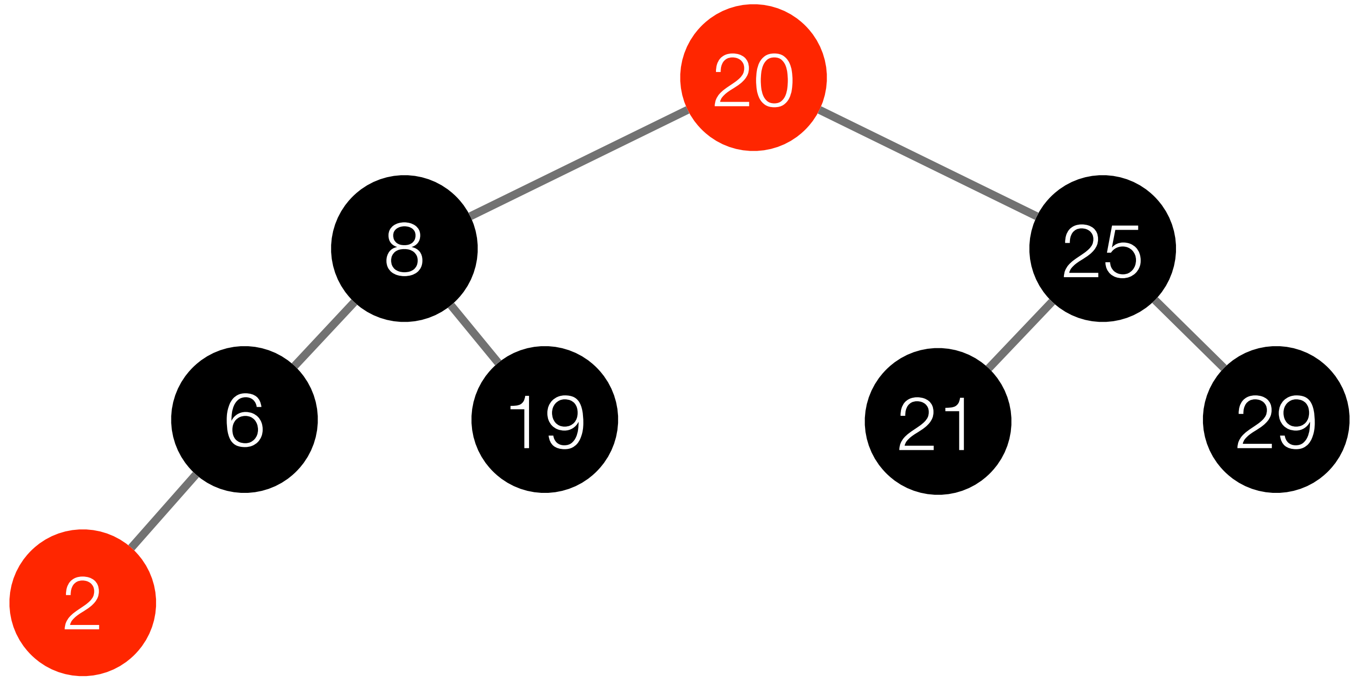


Let's look at another example



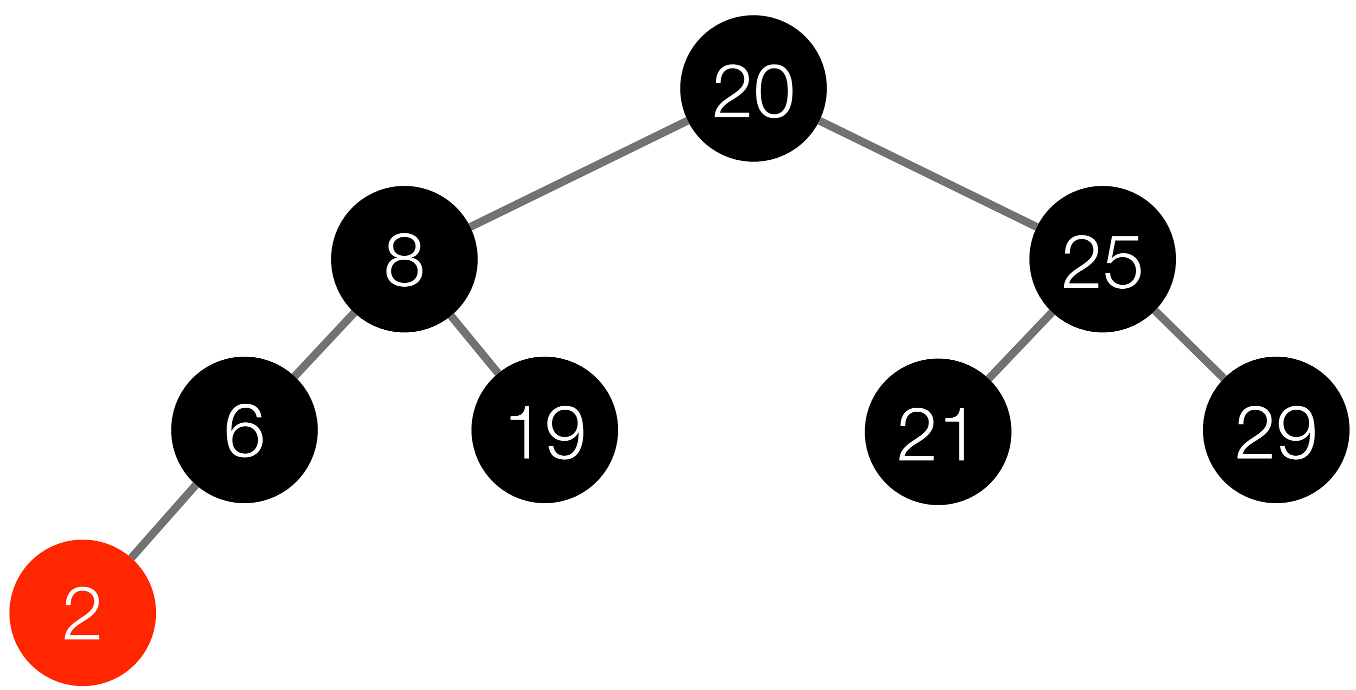
Let's look at another example

If we wanted, we could safely re-color the root:



Let's look at another example

If we wanted, we could safely re-color the root:



Now, let's implement our dictionary!

Now, let's implement our dictionary!

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

Now, let's implement our dictionary!

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

Red Black Tree (RBT) invariant:

Now, let's implement our dictionary!

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

Red Black Tree (RBT) invariant:

A Tree is **sorted** according to an entry's key.

Now, let's implement our dictionary!

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

Red Black Tree (RBT) invariant:

A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

Now, let's implement our dictionary!

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

Red Black Tree (RBT) invariant:

A. Tree is **sorted** according to an entry's key.

B. A **red** node's children must be black.

C. **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Now, let's implement our dictionary!

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

Red Black Tree (RBT) invariant:

- A: Tree is **sorted** according to an entry's key.
- B: A **red** node's children must be black.
- C: **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

→ RBT invariant will become **representation invariant** of structure.

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

RBT invariant will become **representation invariant** of structure.

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

A — Tree is **sorted** according to an entry's key.

B — A **red** node's children must be black.

C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

→ RBT invariant will become **representation invariant** of structure.

Recall, representation invariants are hidden consistency conditions, s.t.

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

→ RBT invariant will become **representation invariant** of structure.

Recall, representation invariants are hidden consistency conditions, s.t.

→ All functions declared by structure

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

→ RBT invariant will become **representation invariant** of structure.

Recall, representation invariants are hidden consistency conditions, s.t.

→ All functions declared by structure

→ may **assume** representation invariant for input,

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

- A: Tree is **sorted** according to an entry's key.
- B: A **red** node's children must be black.
- C: **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

→ RBT invariant will become **representation invariant** of structure.

Recall, representation invariants are hidden consistency conditions, s.t.

→ All functions declared by structure
  → may **assume** representation invariant for input,
  → and must **assert** representation invariant for output.

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

- A — Tree is **sorted** according to an entry's key.
- B — A **red** node's children must be black.
- C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

- A — Tree is **sorted** according to an entry's key.
- B — A **red** node's children must be black.
- C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Our implementation will even make use of a weaker invariant, which can be locally and temporarily violated, but is restored in the end.

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

- A — Tree is **sorted** according to an entry's key.
- B — A **red** node's children must be black.
- C — **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Our implementation will even make use of a weaker invariant, which can be locally and temporarily violated, but is restored in the end.

Almost RBT (ARBT) invariant:

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

- A Tree is **sorted** according to an entry's key.
- B A **red** node's children must be black.
- C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Our implementation will even make use of a weaker invariant, which can be locally and temporarily violated, but is restored in the end.

Almost RBT (ARBT) invariant:

A and C as above,

Now, let's implement our dictionary!

Red Black Tree (RBT) invariant:

A Tree is **sorted** according to an entry's key.

B A **red** node's children must be black.

C **Black height**: for any node, the number of black nodes along any path from the node to a leaf (empty) is the same.

Our implementation will even make use of a weaker invariant, which can be locally and temporarily violated, but is restored in the end.

Almost RBT (ARBT) invariant:

A and C as above,

B' A **red** node's children must be black, unless for a **red root** node, who may have **one** red child.

Specification for restoreLeft

Specification 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.
 *)
```

Specification 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.
 *)
```

Specification 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.
 *)
```

input may
only satisfy weaker
invariant

Specification 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.
 *)
```

input may only satisfy weaker invariant

Specification for restoreLeft

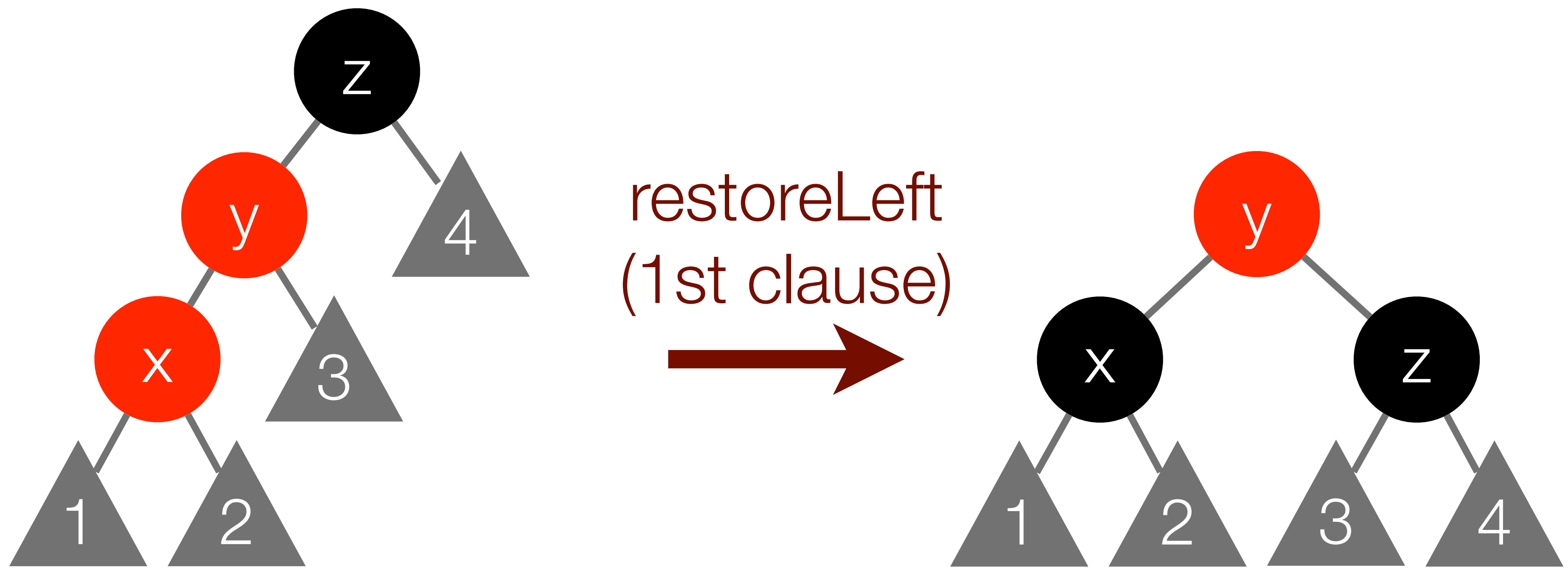
```
(*
  restoreLeft : 'a dict -> 'a dict

  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.
 *)
```

input may only satisfy weaker invariant

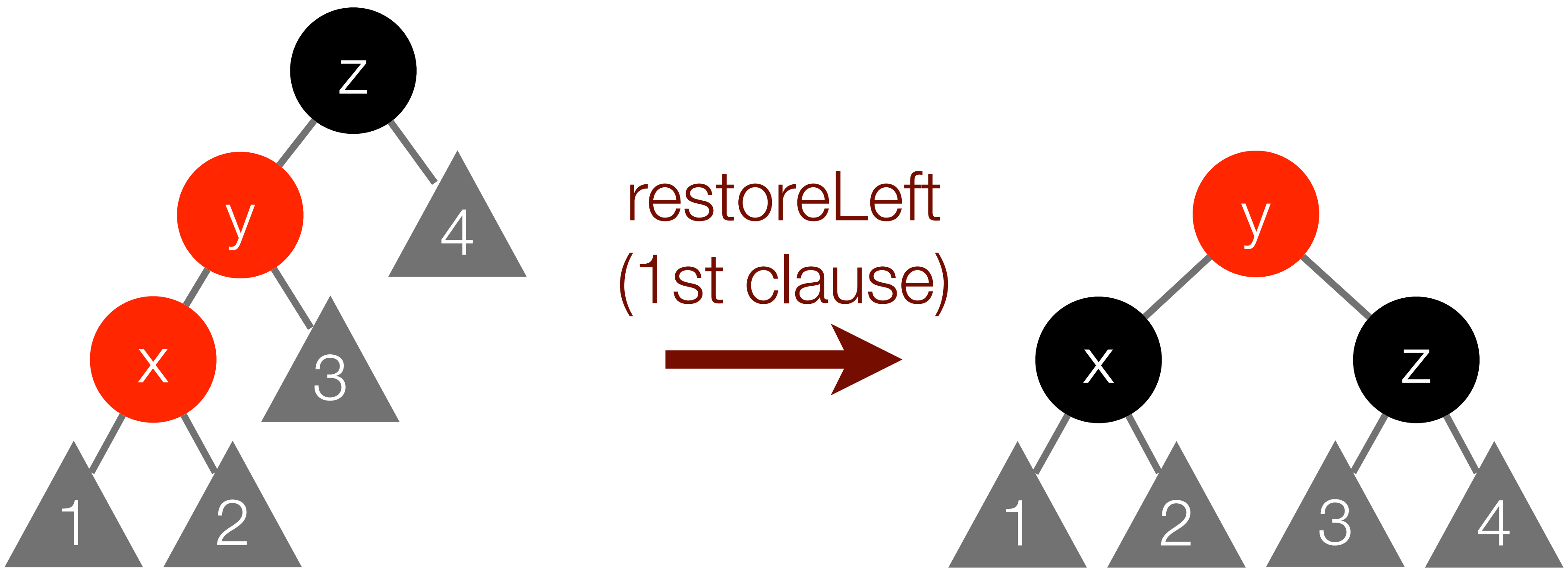
representation invariant established for output

Picture-guided implementation of restoreLeft

Picture-guided implementation of restoreLeft

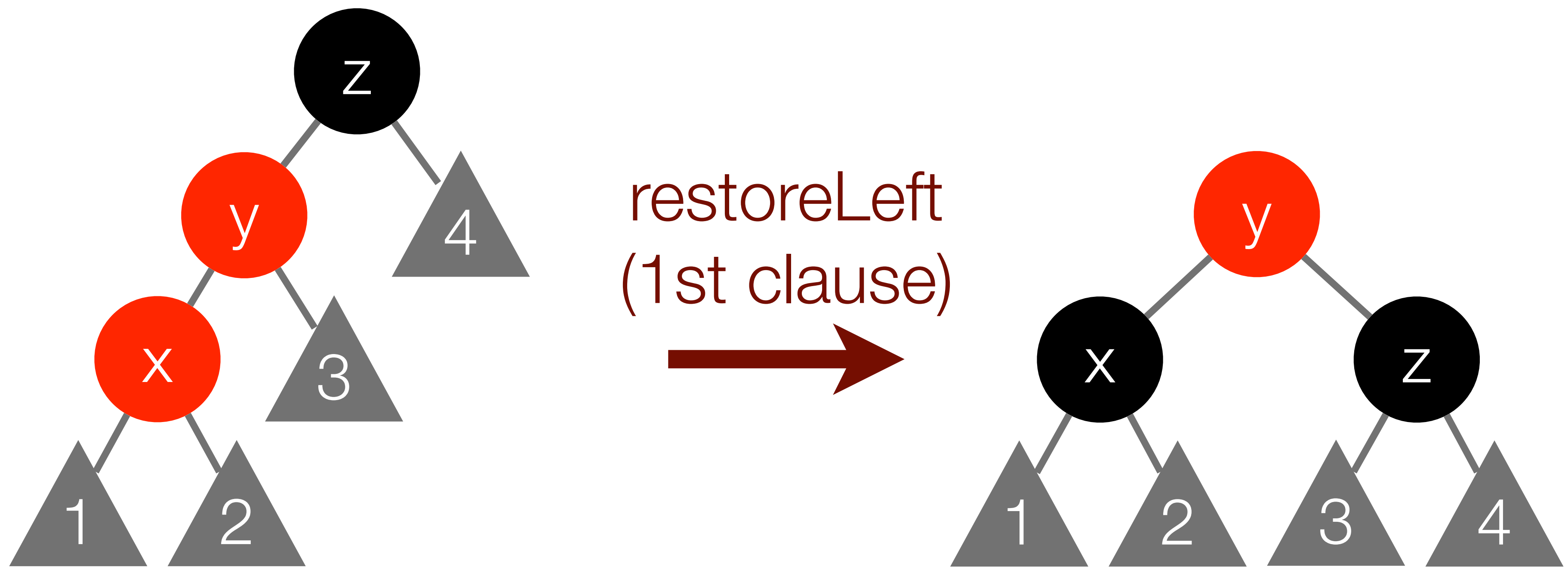


Picture-guided implementation of restoreLeft




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

Picture-guided implementation of restoreLeft

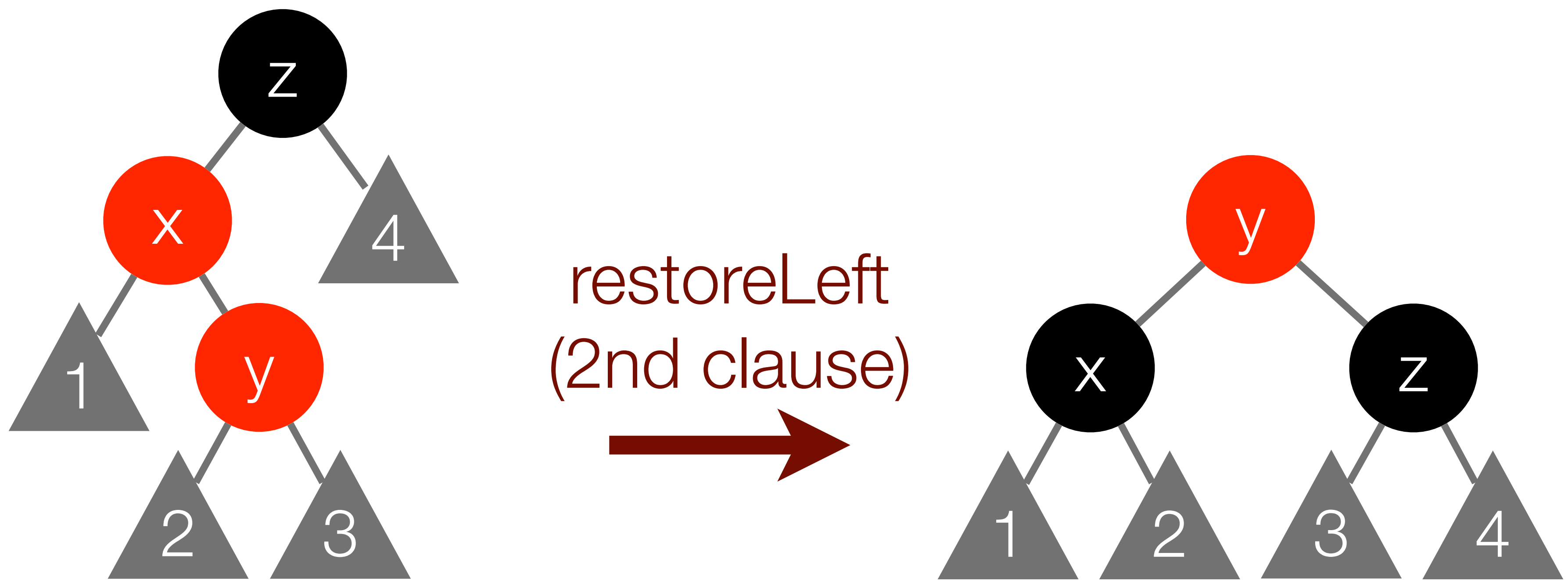



```
fun
  restoreLeft(Black(Red(Red(d1, x, d2), y, d3), z, d4)) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
```

Picture-guided implementation of restoreLeft

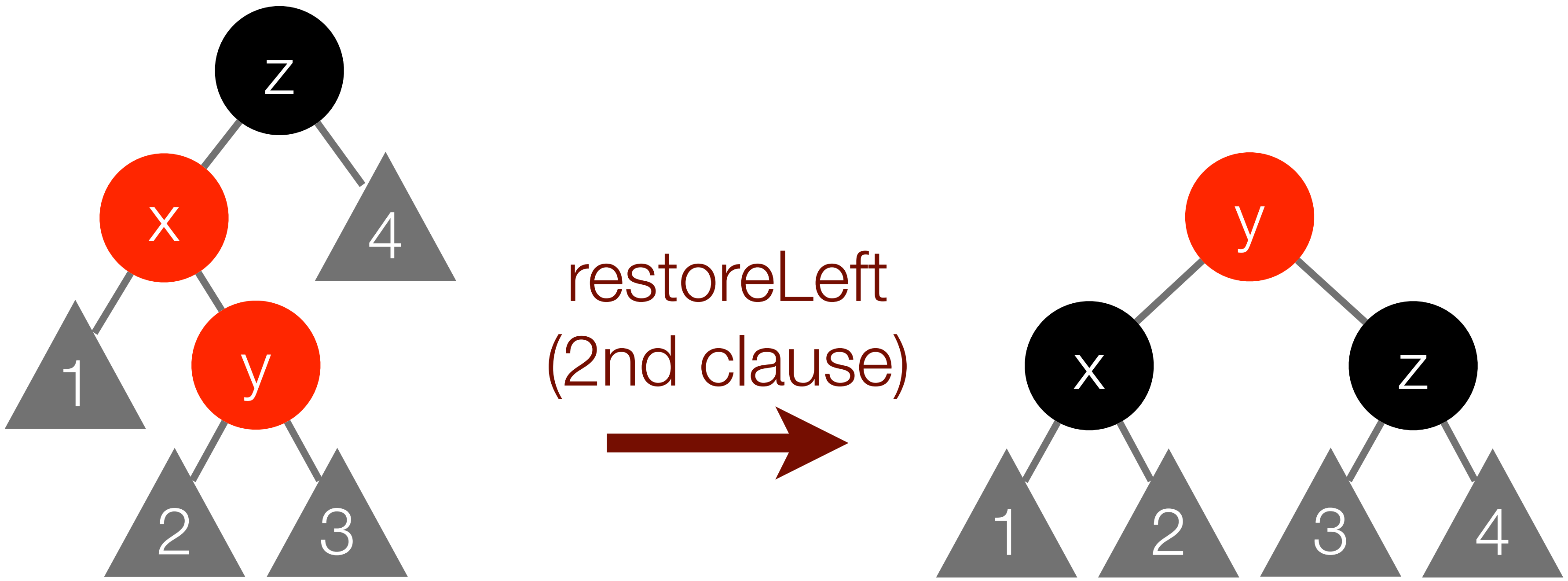
```
fun
  restoreLeft(Black(Red(Red(d1, x, d2), y, d3), z, d4)) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
```

Picture-guided implementation of restoreLeft



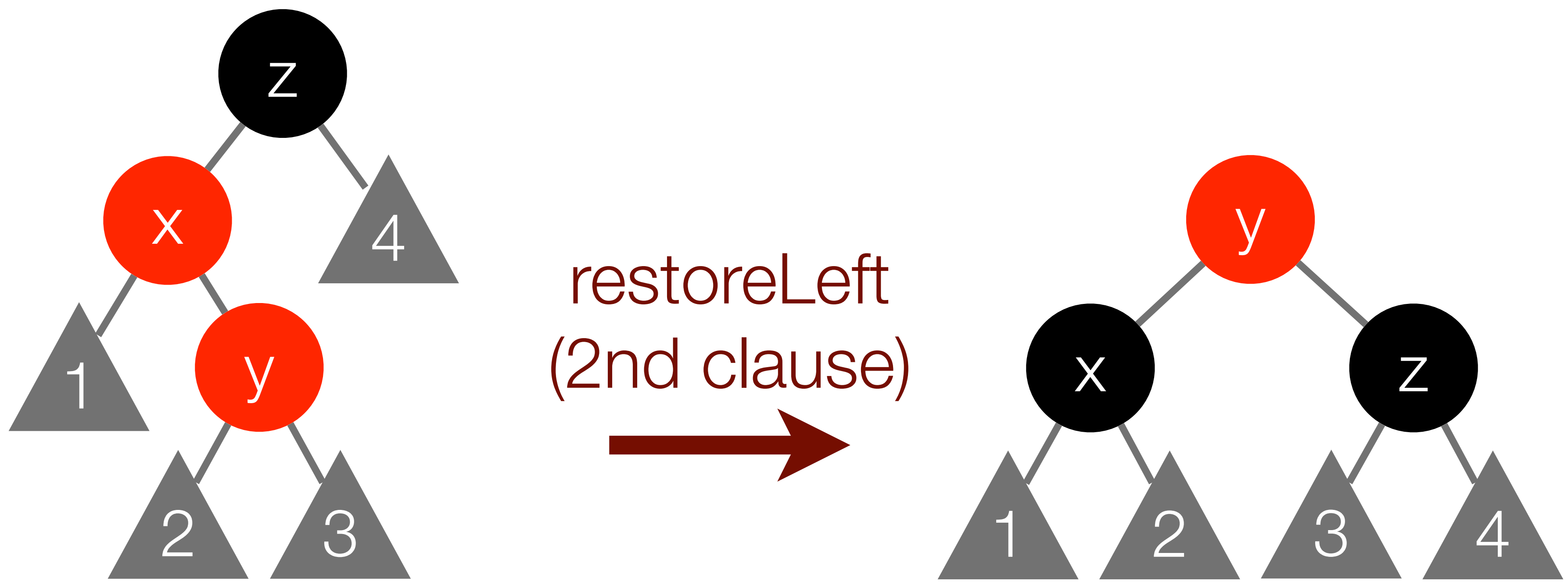

```
fun
  restoreLeft(Black(Red(Red(d1, x, d2), y, d3), z, d4)) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
```

Picture-guided implementation of restoreLeft



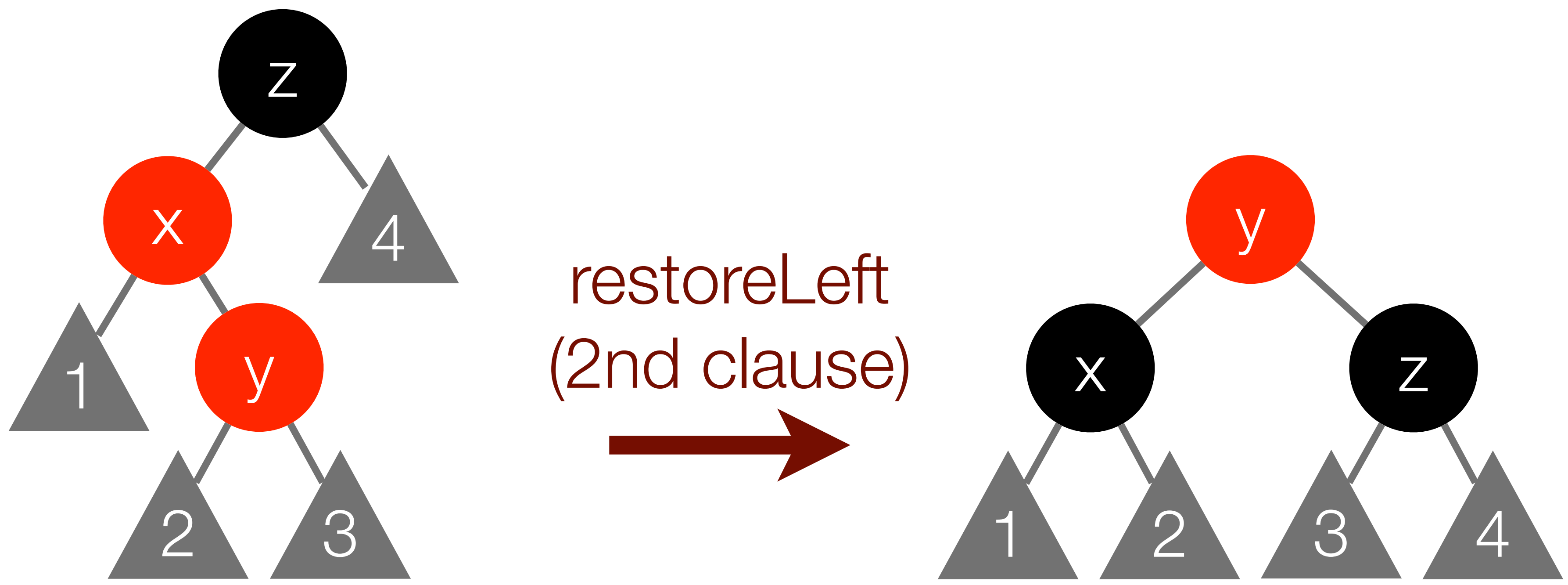

```
fun
  restoreLeft(Black(Red(Red(d1, x, d2), y, d3), z, d4)) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
 |restoreLeft(Black(Red(d1, x, Red(d2, y, d3)), z, d4)) =
```

Picture-guided implementation of restoreLeft




```
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))
```

Picture-guided implementation of restoreLeft




```
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
```

Specification for restoreRight

Specification 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.
 *)
```

Specification 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.
 *)
```

Specification 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.
 *)
```

input may only satisfy weaker invariant

Specification 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.
 *)
```

input may only satisfy weaker invariant

Specification 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.
*)
```

input may only satisfy weaker invariant

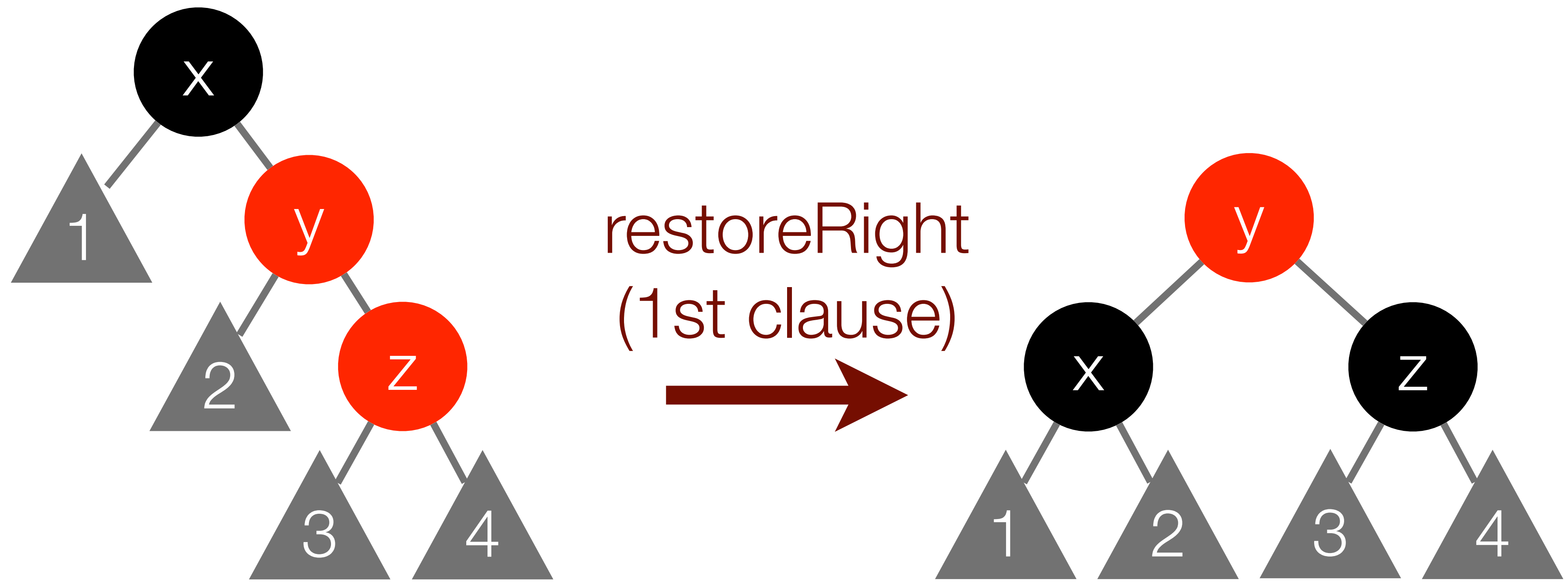
representation invariant established for output

Picture-guided implementation of restoreR

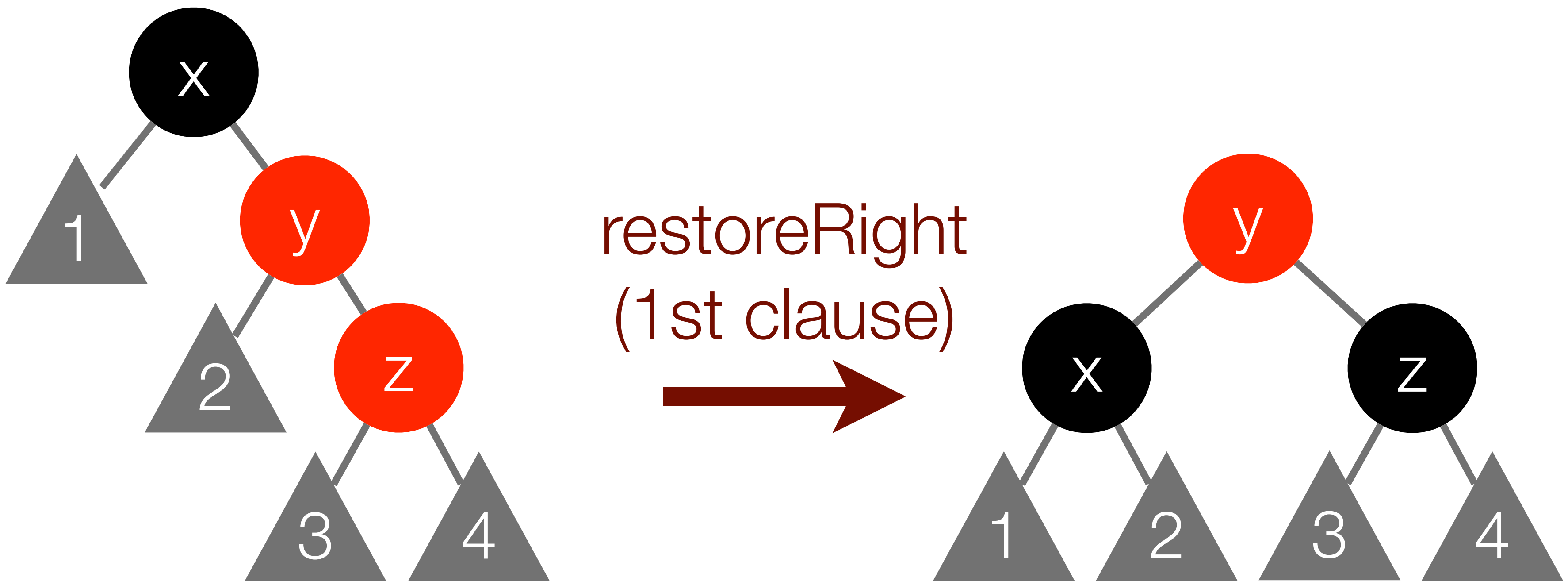
:

:

Picture-guided implementation of restoreR

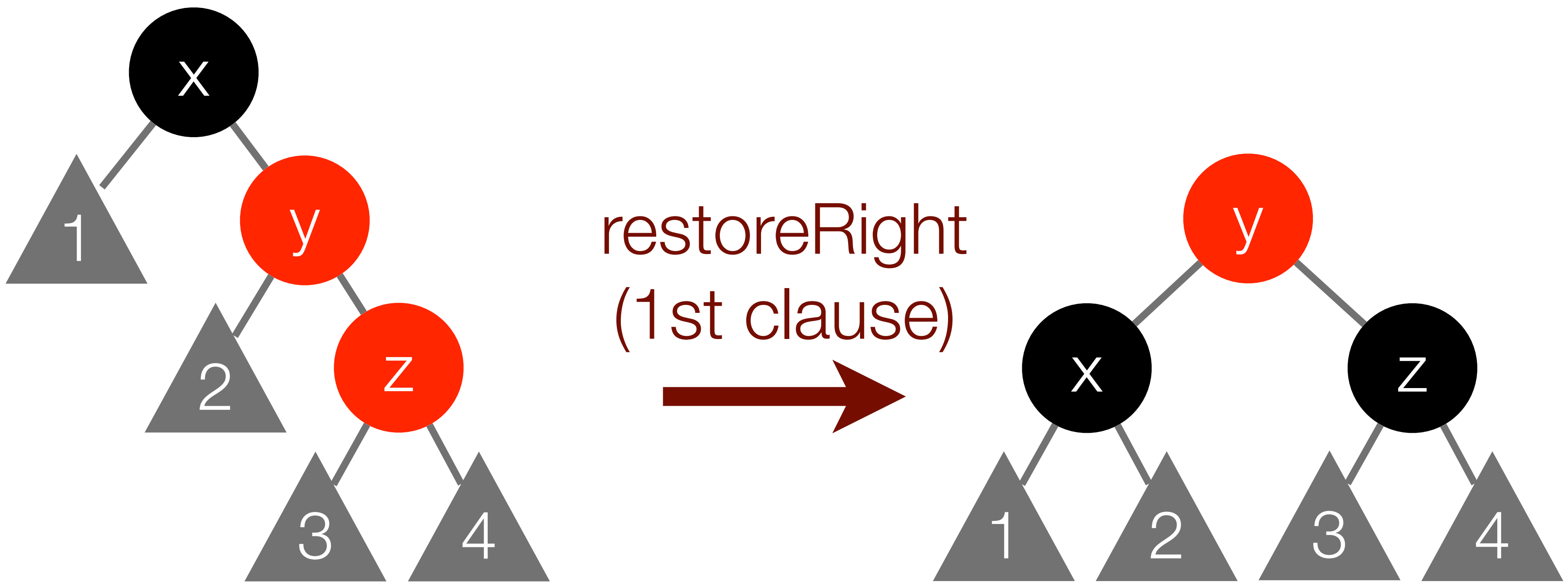


Picture-guided implementation of restoreR




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




                                                              :
```

Picture-guided implementation of restoreR

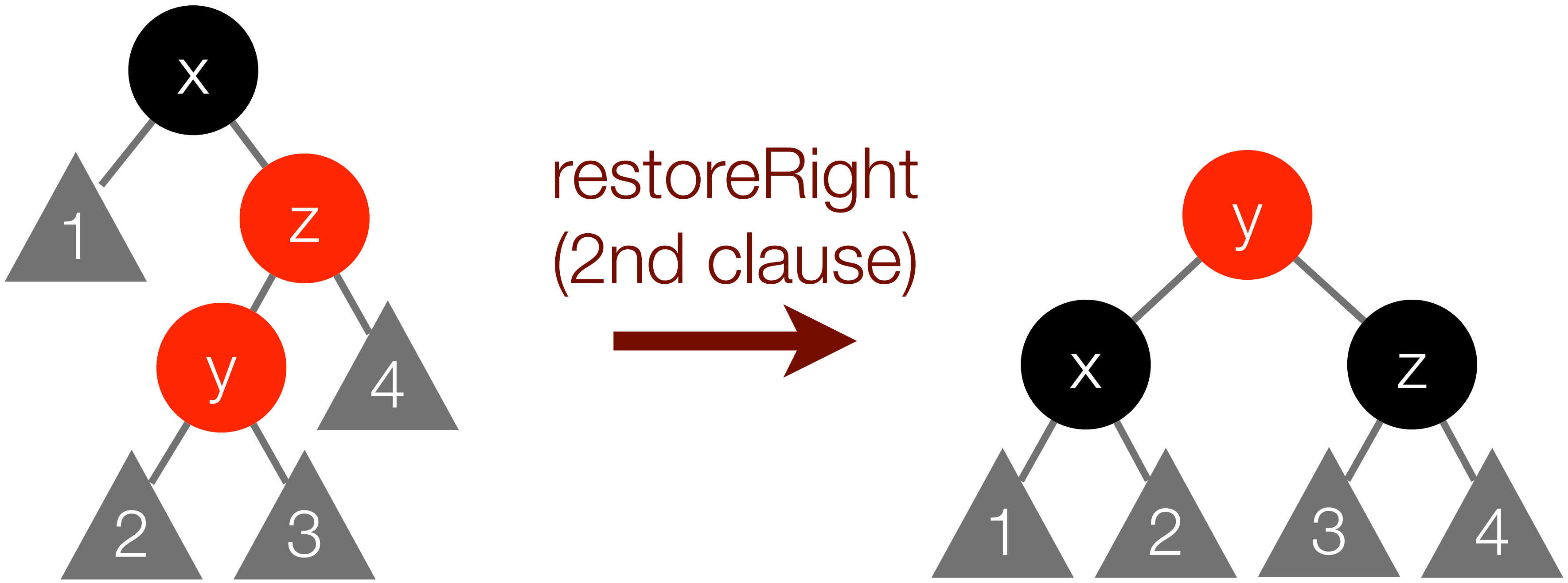



```
fun
  restoreRight(Black(d1, x, Red(d2, y, Red(d3, z, d4)))) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
                                                           :
```

Picture-guided implementation of restoreR

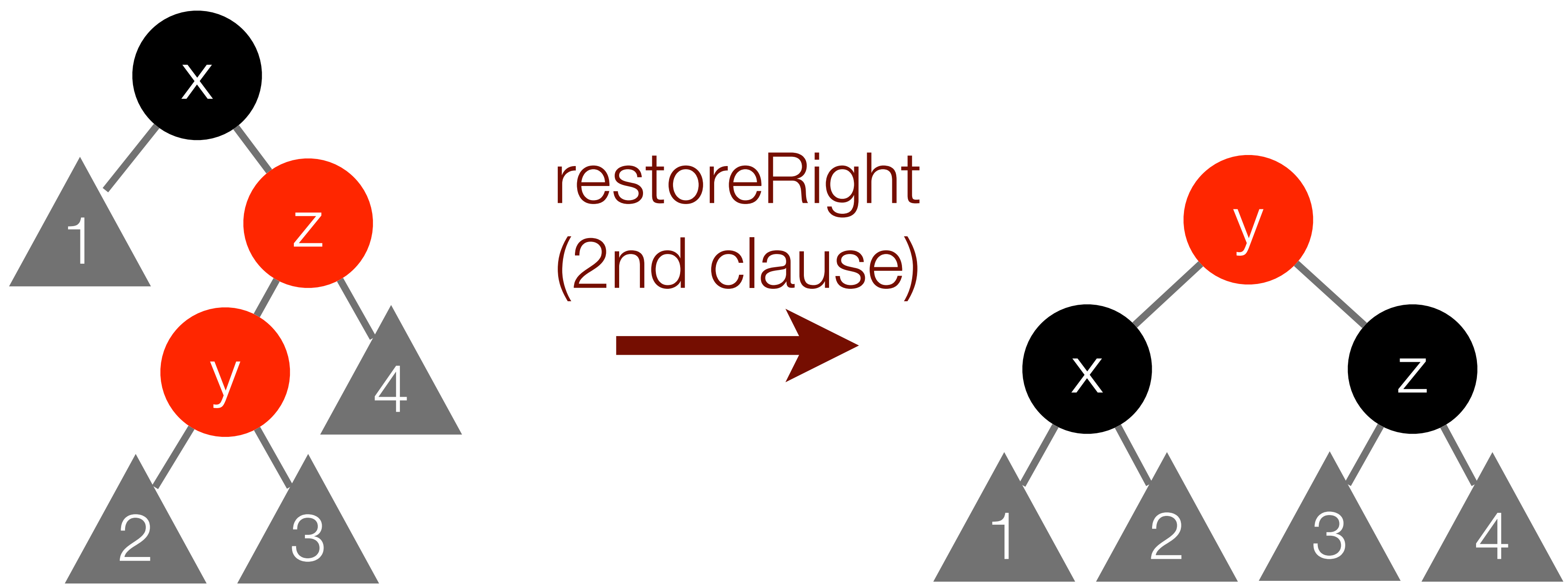
```
fun
  restoreRight(Black(d1, x, Red(d2, y, Red(d3, z, d4)))) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
                                                          :
```

Picture-guided implementation of restoreR



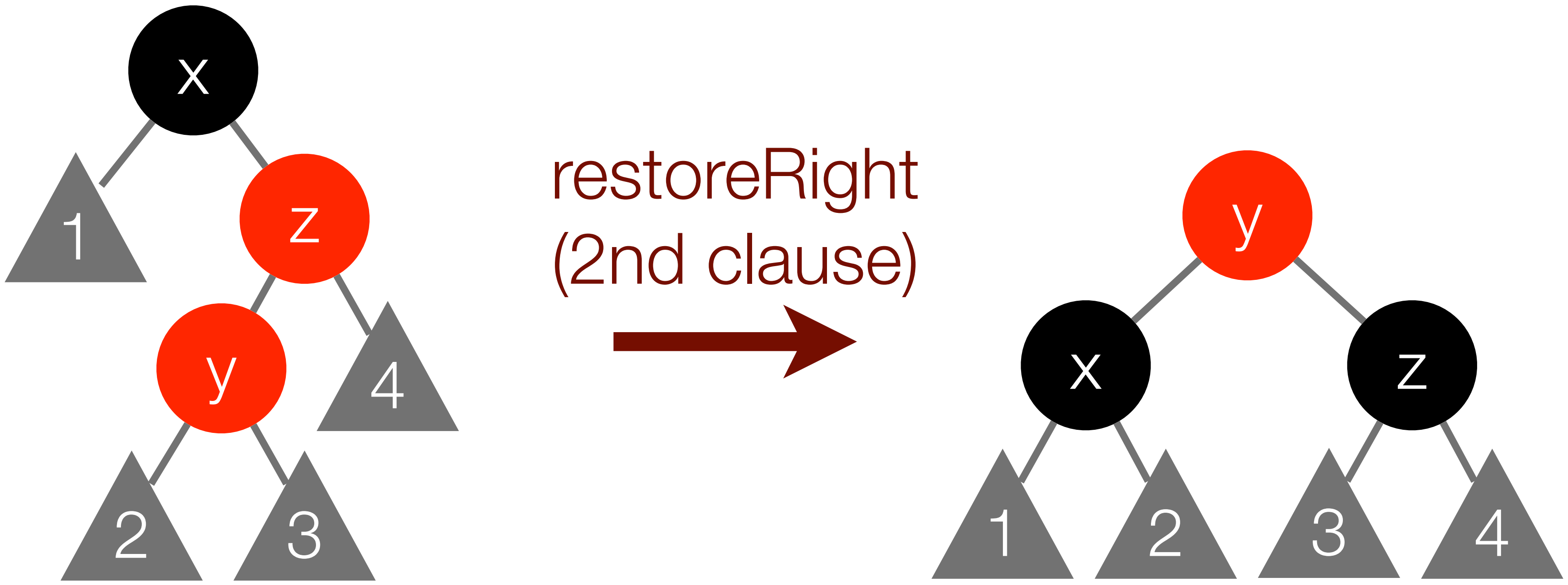

```
fun
  restoreRight(Black(d1, x, Red(d2, y, Red(d3, z, d4)))) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
                                                        :
```

Picture-guided implementation of restoreR



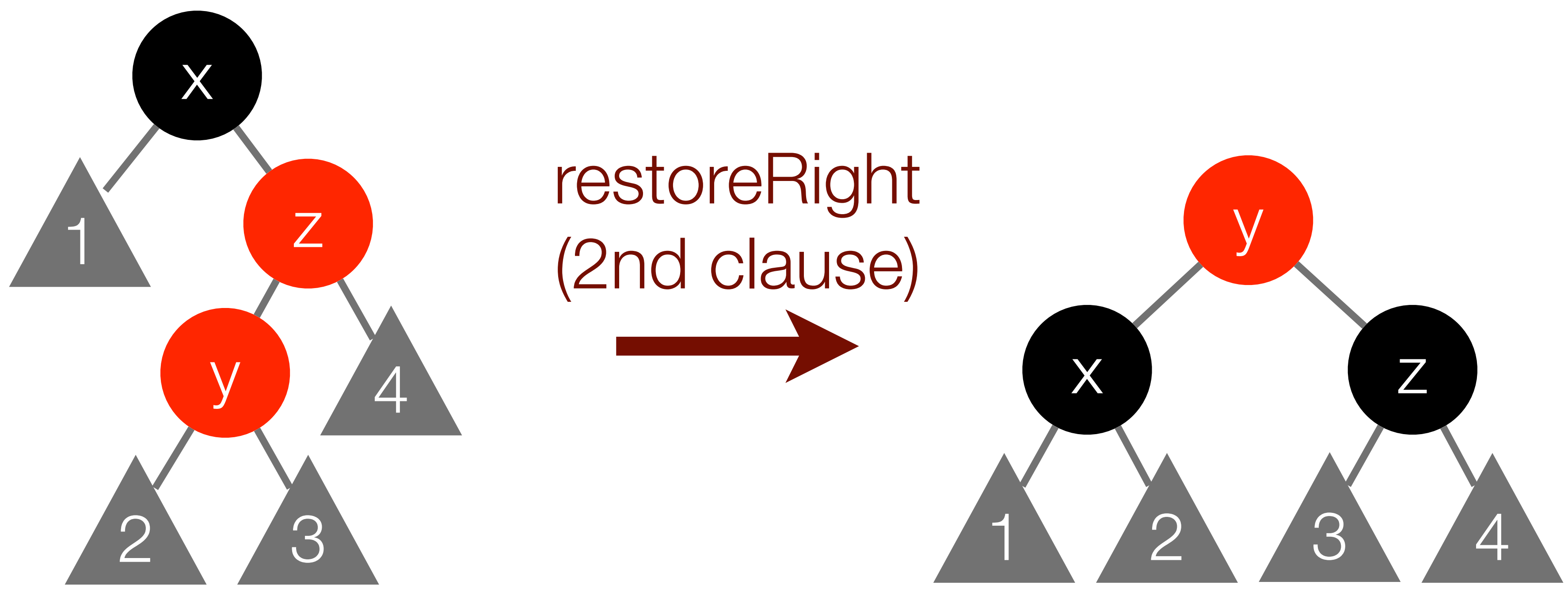

```
fun
  restoreRight(Black(d1, x, Red(d2, y, Red(d3, z, d4)))) =
    Red(Black(d1, x, d2), y, Black(d3, z, d4))
 |restoreRight(Black(d1, x, Red(Red(d2, y, d3), z, d4))) =
```

Picture-guided implementation of restoreR




```
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))
```

Picture-guided implementation of restoreR




```
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
```

What else?

What else?

```
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
```

What else?

```
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
```

Note: restoreLeft and restoreRight are not externally visible!

What else?

```
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
```

Note: restoreLeft and restoreRight are not externally visible!

Let's implement insert next.

Specification for insert

Specification for insert

```
(* 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.
```

Specification for insert

```
(* 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.
```

Specification for insert

```
(* 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.
```

expects
representation
invariant

Specification for insert

```
(* 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.
```

expects representation invariant

Specification for insert

```
(* 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.
```

expects representation invariant

establishes representation invariant

Specification for insert

```
(* 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.
```

Specification for insert

```
(* 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.

   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.
```

Specification for insert

```
(* 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.
```

```
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.
```

Specification for insert

```
(* insert: 'a dict * 'a entry -> 'a dict
   REQUIRES: d is a RBT.
   ENSURES:  insert(d,e) is a RBT contai
             all the entries of d plu
             with e replacing an ent
             if the keys are EQUAL.
```

insert makes use of a locally defined helper function

```
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.
```

Specification for insert

```
(* 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.

   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.
```

Specification for insert

```
(* 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.

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

Specification for insert

```
(* 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.

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

Specification for insert

```
(* 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.

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

may
temporarily violate
representation invariant

Let's implement insert

Let's implement insert

```
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
```

Let's implement insert

```
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
```

Let's implement insert

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

re-color in case of a red-red violation at the root

Let's implement insert

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

re-color in
case of a red-red violation
at the root

RBT representation invariant preserved.

Let's implement insert

```
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
```

Let's implement insert

```
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
```

Let's implement insert

```
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 layered pattern matching!

Let's implement ins

Let's implement ins

```
fun ins (Empty) = Red(Empty, e, Empty)
  | ins (Black(l, e' as (k',_), r)) =
    (case String.compare (k, k') of
      EQUAL =>
    | LESS =>
    | GREATER =>
```

Let's implement ins

```
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 =>
    | GREATER =>
```

Let's implement ins

```
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 =>
    | GREATER =>
```

Let's implement ins

```
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 =>
    | GREATER =>
```

only choice,
otherwise we destroy
black height

Let's implement ins

```
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 =>
    | GREATER =>
```

only choice, otherwise we destroy black height

recall: weaker invariant still guarantees black height

Let's implement ins

```
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 =>
    | GREATER =>
```

Let's implement ins

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

Let's implement ins

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

Let's implement ins

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

➡ Is that really it?

Let's implement ins

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

Is that really it?

No, we have to invoke restore functions because `ins` may return a tree that only satisfies ARBT!

Let's implement ins

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

Is that really it?

No, we have to invoke restore functions because `ins` may return a tree that only satisfies ARBT!

Let's implement ins

```
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)))
```

Is that really it?

No, we have to invoke restore functions because `ins` may return a tree that only satisfies ARBT!

Let's implement ins

```
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)))
```

Let's implement ins

```
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 =>
    | LESS =>
    | GREATER =>
```

Let's implement ins

```
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)
    | LESS =>
    | GREATER =>
```

Let's implement ins

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

Let's implement ins

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

Let's implement ins

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

Let's implement ins

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

Should we call the restore functions here too?

Let's implement ins

```
| ins (Red(l, e' as (k',_), r)) =
  (case String.compare (k, k') of
    EQUAL => Red(l, e, r)
  | LESS => Red(ins l, e', r)
  | GREATER => Red(l, e', ins r))
```

Should we call the restore functions here too?

Let's implement ins

```
| ins (Red(l, e' as (k',_), r)) =
  (case String.compare (k, k') of
    EQUAL => Red(l, e, r)
  | LESS => Red(ins l, e', r)
  | GREATER => Red(l, e', ins r))
```

Should we call the restore functions here too?

No, restore functions require black roots.

Let's implement ins

```
| ins (Red(l, e' as (k',_), r)) =
  (case String.compare (k, k') of
    EQUAL => Red(l, e, r)
  | LESS => Red(ins l, e', r)
  | GREATER => Red(l, e', ins r))
```

Should we call the restore functions here too?

No, restore functions require black roots.

Moreover, `l` and `r` must have black roots by the pre-condition.

Let's implement ins

```
| ins (Red(l, e' as (k',_), r)) =
  (case String.compare (k, k') of
     EQUAL => Red(l, e, r)
   | LESS => Red(ins l, e', r)
   | GREATER => Red(l, e', ins r))
```

- Should we call the restore functions here too?
- No, restore functions require black roots.
- Moreover, `l` and `r` must have black roots by the pre-condition.
- And, we get back an RBT by the post-condition.

Finishing up

Finishing up

Look at lecture code for function `lookup`.

Finishing up

- Look at lecture code for function `lookup`.
  - Uses SML's `and` construct for mutually recursive functions.

Finishing up

- Look at lecture code for function `lookup`.
  - Uses SML's **`and`** construct for mutually recursive functions.
- We use opaque ascription for our RBT structure.

Finishing up

- Look at lecture code for function `lookup`.
  - Uses SML's `and` construct for mutually recursive functions.
- We use opaque ascription for our RBT structure.
  - Encapsulates and protects representation invariant.

Finishing up

- Look at lecture code for function `lookup`.
  - Uses SML's `and` construct for mutually recursive functions.
- We use opaque ascription for our RBT structure.
  - Encapsulates and protects representation invariant.
  - Experiment with the code to see hiding at play!

That's all for today.