15-150 Fall 2013
Lecture 18

Stephen Brookes

Today

A case study in *modular programming*

- Implementing ***dictionaries***
  - binary search trees
  - red-black trees

dictionaries

- Earlier we talked about ***sorting*** with respect to a *comparison function*
- Let's revisit this using ***modular*** programming
  - A signature for *dictionaries*
    - collections of *entries*, sorted by *keys*
  - An implementation of dictionaries as *binary search trees*, parameterized by an *ordered type* of keys

signatures

```
signature ORDERED =
 sig
   type t
   val compare : t * t -> order
 end

signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : Key.t * 'a -> 'a dict -> 'a dict
    val lookup : Key.t -> 'a dict -> 'a option
    val trav : 'a dict -> (Key.t * 'a) list
 end
```

signatures can contain structures

ordered types

```
sig
  type t
  val compare : t * t -> order
end
```

```
structure Integers : ORDERED =
 struct
   type t = int
   fun compare(x, y) =
     if x<y then LESS else
     if y<x then GREATER else EQUAL
 end
```

```
structure Strings : ORDERED =
 struct
   type t = string
   val compare = String.compare
 end
```

A functor for dictionaries

```
functor BSTDict(Key : ORDERED) : DICT =
 struct
  structure Key : ORDERED = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

  val empty = Leaf

  fun lookup k Leaf = NONE
   |  lookup k (Node (l, (k', v'), r))  =
        case Key.compare (k, k') of
           EQUAL         =>  SOME v'
         | LESS          =>  lookup k l
         | GREATER       =>  lookup k r
  ......
```

structures can contain structures

```
sig
  structure Key : ORDERED
  type 'a dict
  val empty : 'a dict
  val insert : Key.t * 'a -> 'a dict -> 'a dict
  val lookup : Key.t -> 'a dict -> 'a option
  val trav : 'a dict -> (Key.t * 'a) list
end
```

Leaf, Node not visible

BSTDict(Key : ORDERED) : DICT
(continued)

.....

```
  fun insert (k, v) Leaf  = Node(Leaf, (k, v), Leaf)
   |  insert (k, v) (Node(l, (k', v'), r)) =
      case Key.compare(k, k') of
             EQUAL      => Node(l, (k,v), r)
       |     LESS       => Node(insert (k,v) l, (k',v'), r)
       |     GREATER => Node(l, (k',v'), insert (k,v) r)

  fun trav Empty = [ ]
   |   trav (Node(l, (k,v), r)) = (trav l) @ (k,v) :: (trav r)

end
```

```
functor BSTDict(Key : ORDERED) : DICT =
 struct
  structure Key : ORDERED = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree
  val empty = Leaf
  fun lookup k Leaf = NONE
   |  lookup k (Node (l, (k', v'), r))  =
        case Key.compare (k, k') of
           EQUAL         =>  SOME v'
         | LESS          =>  lookup k l
         | GREATER       =>  lookup k r
  fun insert (k, v) Leaf  = Node(Leaf, (k, v), Leaf)
   |  insert (k, v) (Node(l, (k', v'), r)) =
        case Key.compare(k, k') of
             EQUAL      => Node(l, (k,v), r)
        |    LESS       => Node(insert (k,v) l, (k',v'), r)
        |    GREATER    => Node(l, (k',v'), insert (k,v) r)

  fun trav Empty = [ ]
   |   trav (Node(l, (k,v), r)) = (trav l) @ (k,v) :: (trav r)
 end
```

Leaf, Node not visible

use of b.s.t's
is NOT apparent
from the signature

bst

- Leaf is a binary search tree
- Node(l, (k,v), r) is a binary search tree iff

  every key in l is K.compare-LESS than k
  every key in r is K.compare-GREATER than k
  and l and r are binary search trees

bst(T) = “T is a binary search tree”

properties

- Suppose K:ORDERED
  and K.compare is a *comparison function*
  for type K.t
- Let S = BSTDict(K)
- For all types entry,
  every value of type entry S.dict
  definable from S.empty and S.insert
  is a K.compare binary search tree

why?

- S.empty satisfies bst property for K.compare
- If T satisfies bst, so does S.insert (k,v) T

- Only S.empty and S.insert are visible to users
- S.Leaf and S.Node are not!

importance?

- By doing only ***local*** *reasoning* we guarantee a ***global*** *property*
- The correctness proof only examines code inside the structure ("local reasoning")
- Since users have limited access, they cannot *break the abstraction barrier*
  - *representation invariant*

results

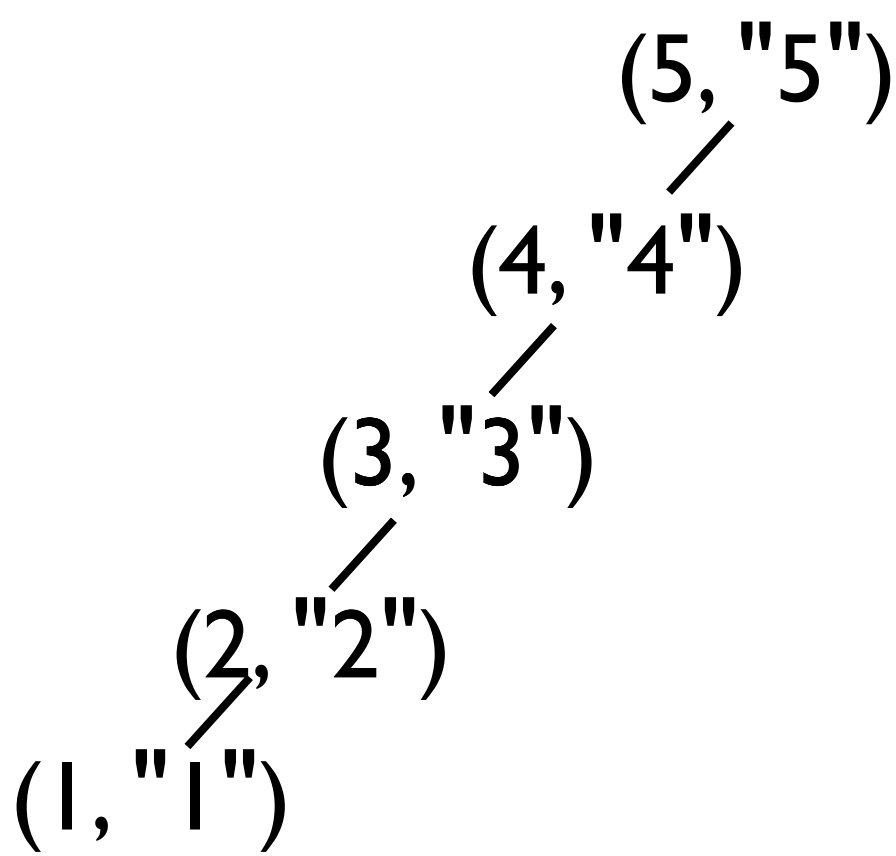
```
structure S = BSTDict(Integers);

open S;

fun build [ ] = empty
 |     build (x::L) = insert (x, Int.toString x) (build L) ;

val D = build [1,2,3,4,5];
```

Type of D?
Value of D?

results

- val D = build [1,2,3,4,5];

val D = Node (Node (Node #,(#,#),Leaf),(5,"5"),Leaf)
: string dict



a binary search tree

but badly balanced

so far

- We have a functor for building implementations of DICT
- But the work to evaluate

  lookup k T

  is O(size T) in the worst case

even though we represent a dictionary as a binary search tree

balancing trees

- Now we'll implement ***red-black trees***
- Binary search trees with *colored* nodes
- Color used to *constrain* tree structure
- Constraints ensure that the ratio of

longest path : shortest path
is no worse than
2 : 1

why is this good?

- In a tree with 2^n nodes
  and a path ratio bounded by 2,
  the cost of a lookup is $O(n)$.
- For ordinary binary search trees,
  this cost is $O(2^n)$ in worst case.

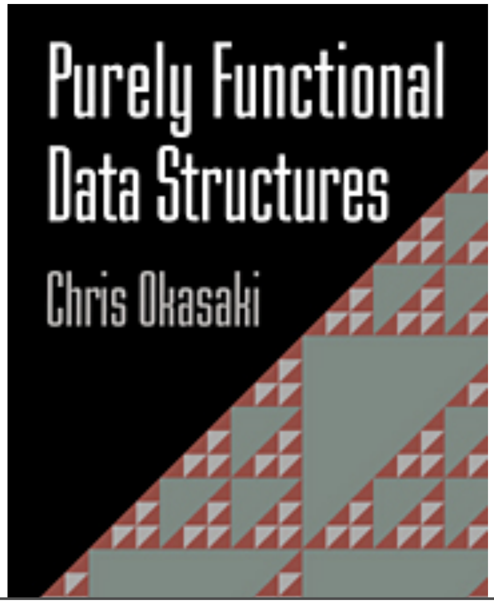
balanced *implies* fast lookup and insert

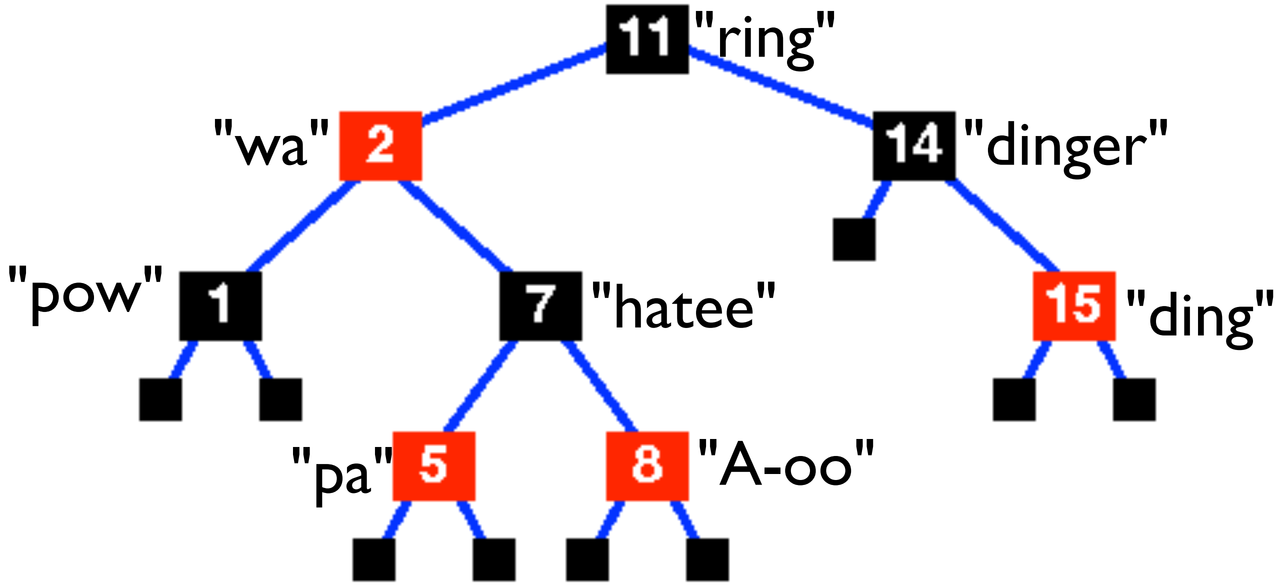
colored trees

datatype color = Red | Black

datatype 'a tree = Leaf
  | Node of 'a tree * (color * (Key.t * 'a)) * 'a tree

type 'a dict = 'a tree





red-black trees

nodes are red or black
leaves are (implicitly) black

- **Sorted** as with binary search trees.
- **Well-red**: no red node has a red child.
- **Well-black**: every path from root to a leaf has the *same number of black nodes*
    - this is the *black height* of the tree

empty

val empty = Leaf

(This is a red-black tree!)

lookup

lookup : Key.t -> 'a dict -> 'a option

```
fun lookup k Leaf = NONE
 |    lookup k (Node (l, (_ , (k', v)), r)) =
        case Key.compare (k,k') of
           EQUAL        =>  SOME v
         | LESS         =>  lookup k l
         | GREATER      =>  lookup k r
```

(colors are ignored by **lookup**)

insertion

Two kinds of inserts:

- insert (k,v) D is an ***update*** when k is EQUAL to a key already in D
  - doesn't change the tree structure
- insert (k,v) D is a ***new insertion*** when k is not EQUAL to any key in D
  - (k,v) goes in at a leaf node of D

What should we do about colors?

updates

- Inserting with an existing key doesn't change tree structure, so *copy* the color

  We'll define

  insert (k, v) (Node(l, (c,(k',v')),r))
  = Node(l, (c,(k, v)), r)

  when Key.compare(k, k') = EQUAL

updates

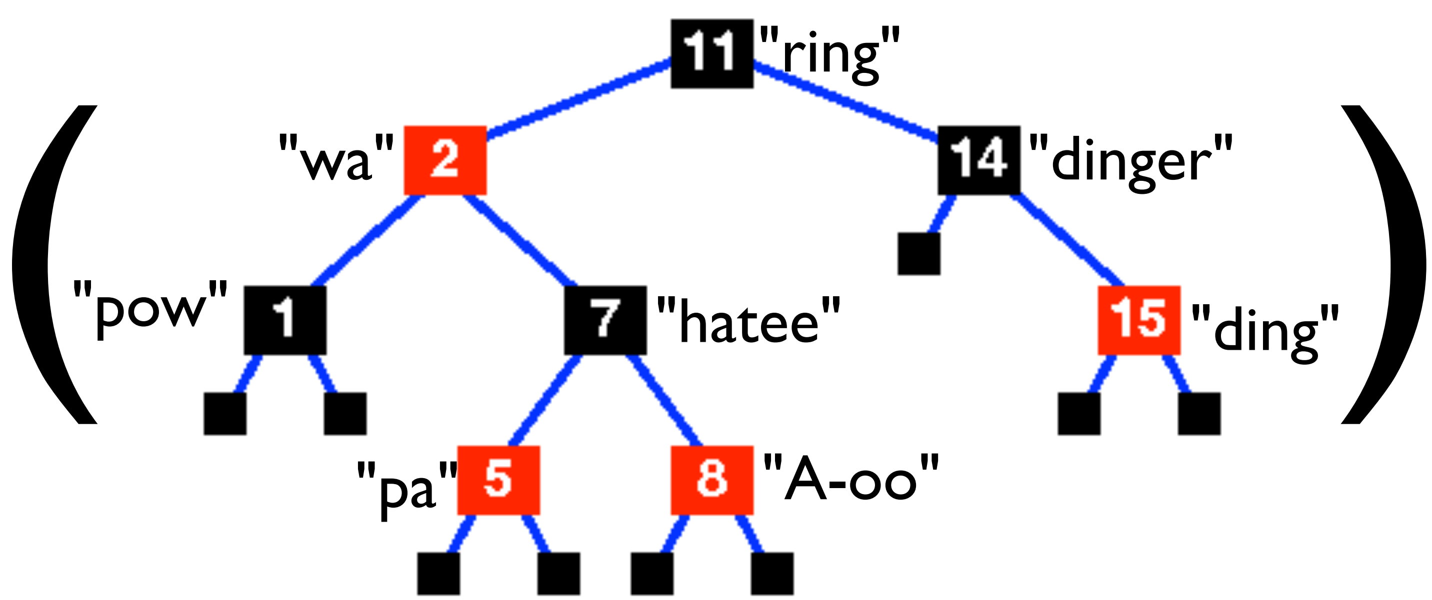
- Inserting with an existing key doesn't change tree structure, so *copy* the color

We'll define

insert (k, v) (Node(l, (c,(k',v')),r))
= Node(l, (c,(k, v)), r)

when Key.compare(k, k') = EQUAL

insert (2,"Abay-ba")



updates

- Inserting with an existing key doesn't change tree structure, so *copy* the color

  We'll define

  insert (k, v) (Node(l, (c,(k',v')),r))
  = Node(l, (c,(k, v)), r)

  when Key.compare(k, k') = EQUAL

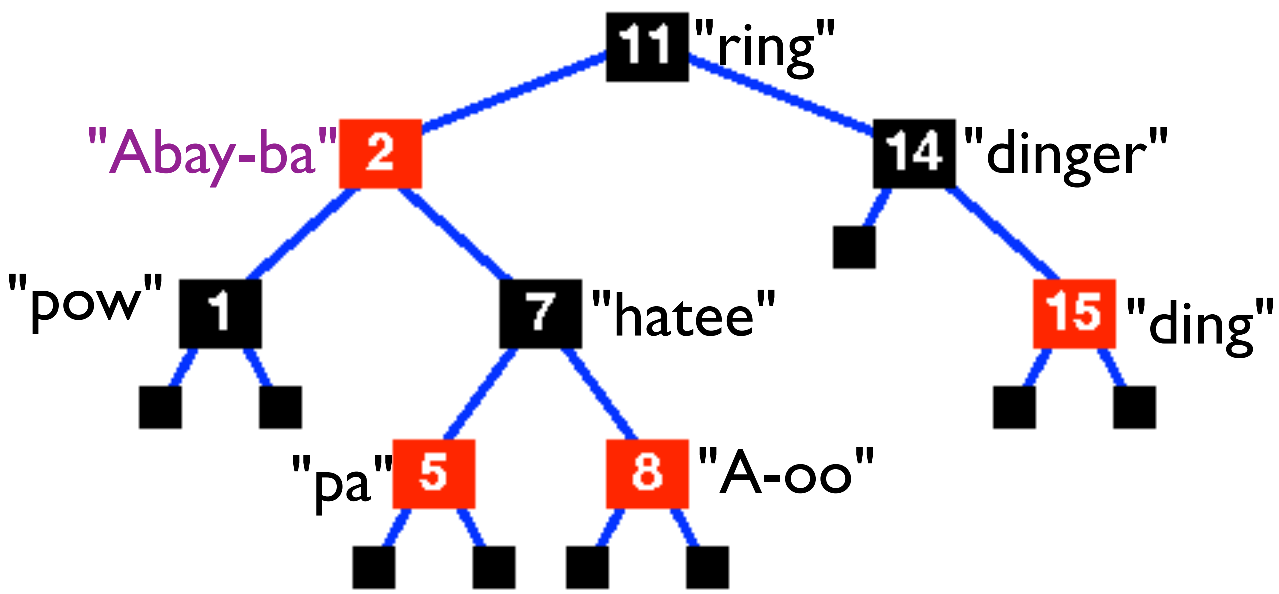
updates

- Inserting with an existing key doesn't change tree structure, so *copy* the color

  We'll define

  insert (k, v) (Node(l, (c,(k',v')),r))
  = Node(l, (c,(k, v)), r)

  when Key.compare(k, k') = EQUAL



new inserts

Let's define

insert (k, v) Leaf = Node(Leaf, (**Red**, (k, v)), Leaf)

Reason:

new insertions happen at leaf nodes
and choosing **Black** would mess up
the black height

new inserts

insert (k, v) (Node(l, (k’, v’), r))

When Key.compare(k, k’) is LESS

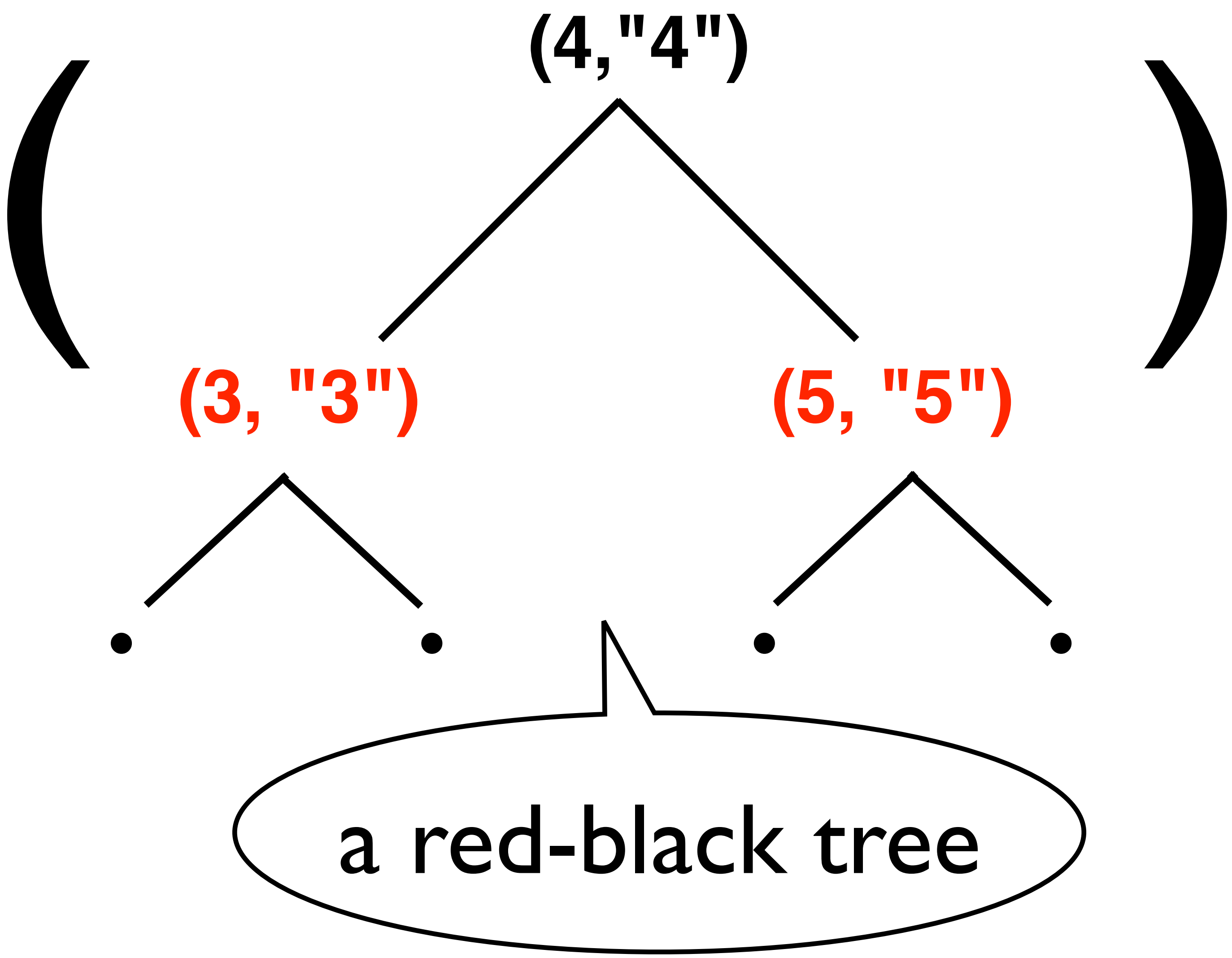
we need to put (k,v) into the left subtree l

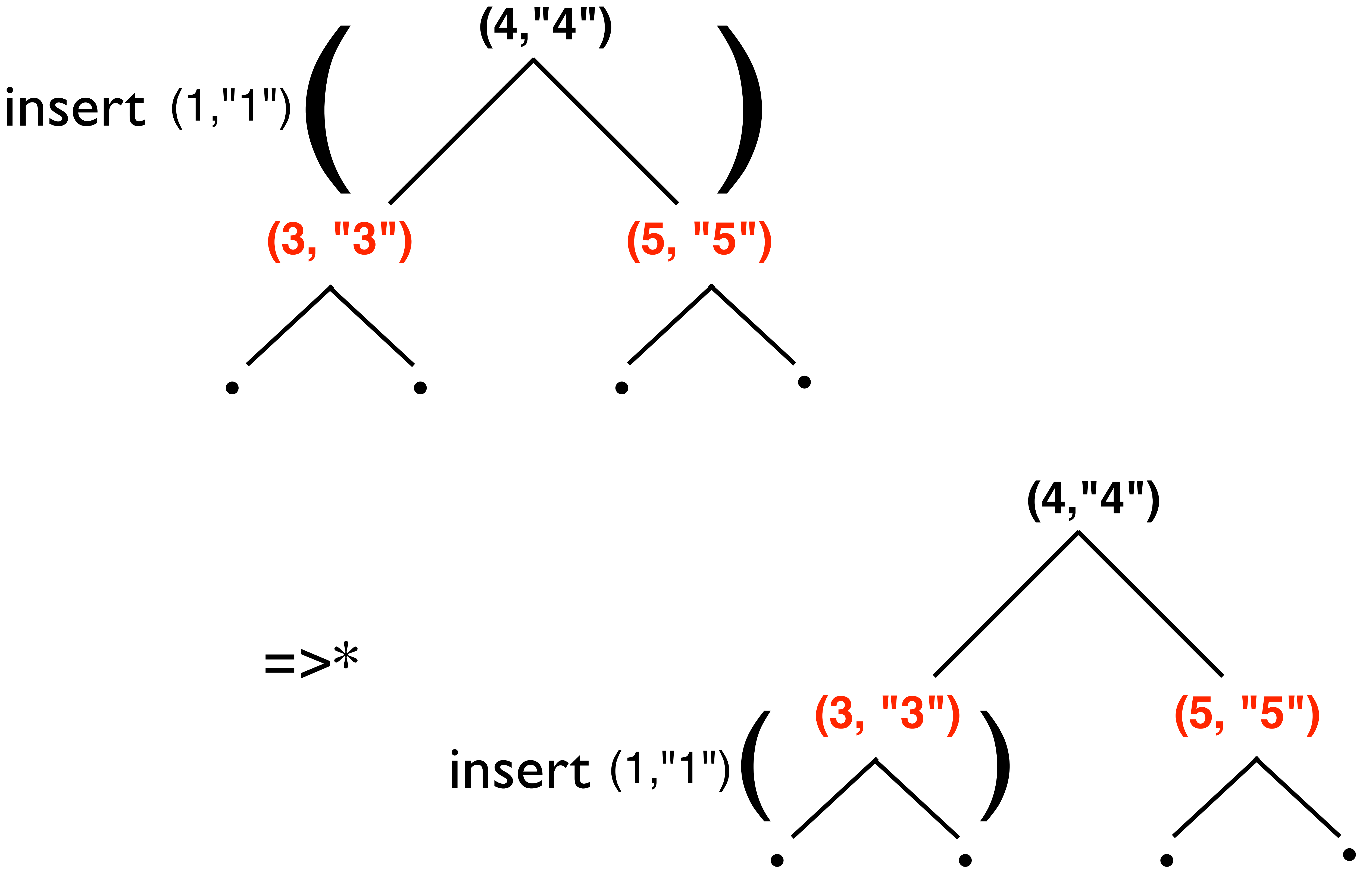
When Key.compare(k, k’) is GREATER

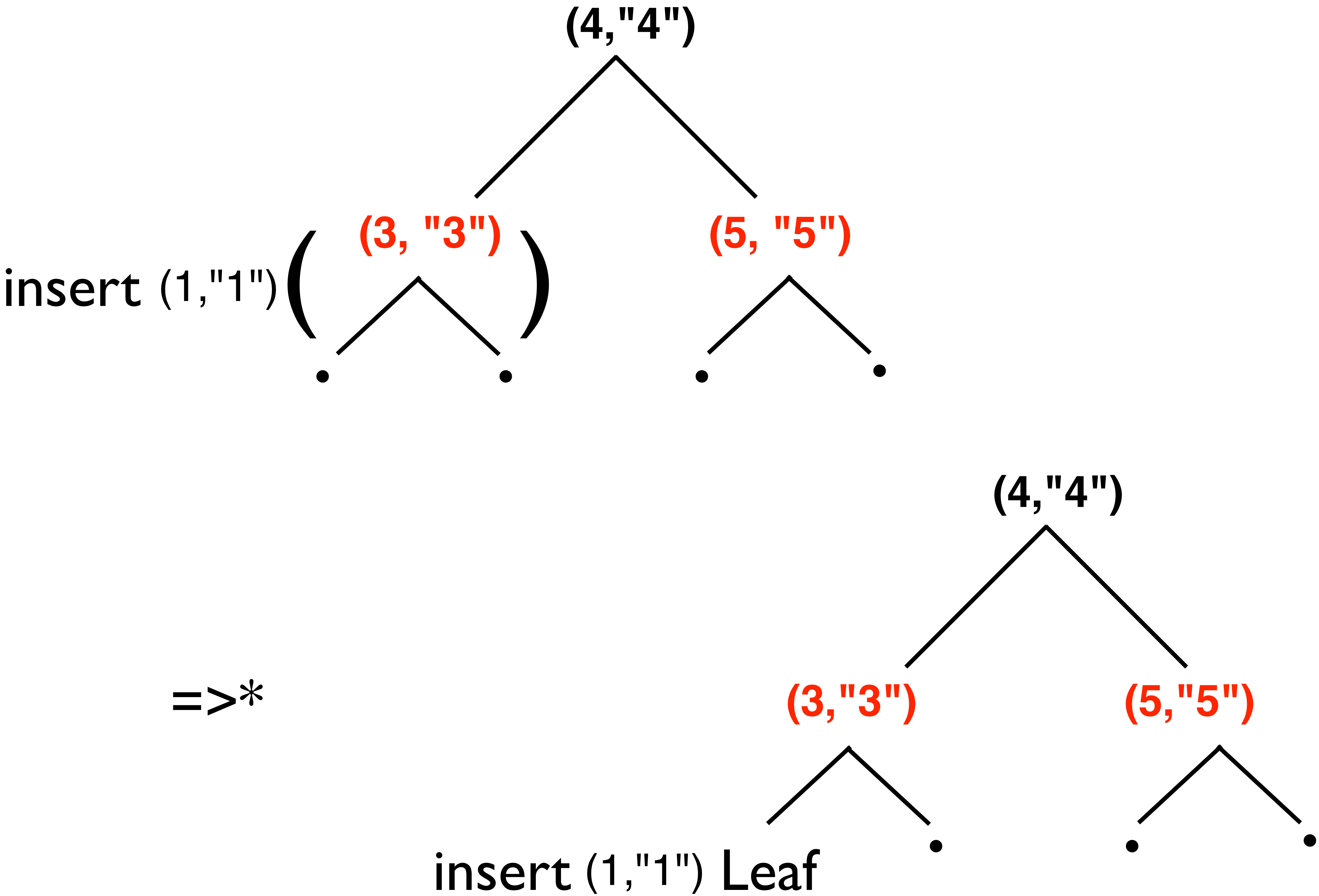
we need to put (k,v) into the right subtree r

inserting a new key

insert (1,"1")

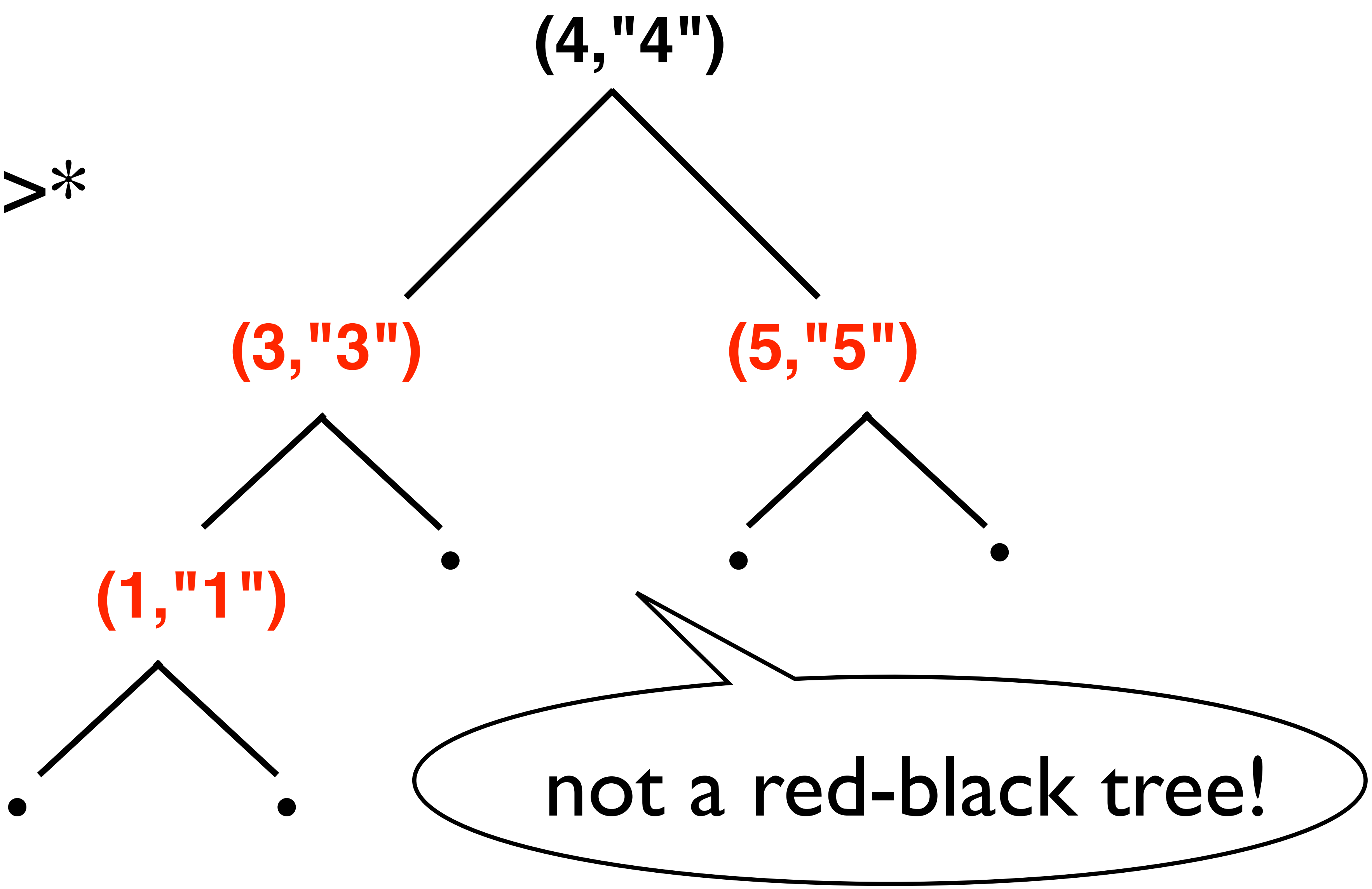



insert (1,"1")
(4,"4")
(3, "3")
(5, "5")
=>*
insert (1,"1")
(4,"4")
(3, "3")
(5, "5")


(4,"4")
(3, "3")
(5, "5")
insert (1,"1")
=>*
(4,"4")
(3,"3")
(5,"5")
insert (1,"1") Leaf

inserting a new key

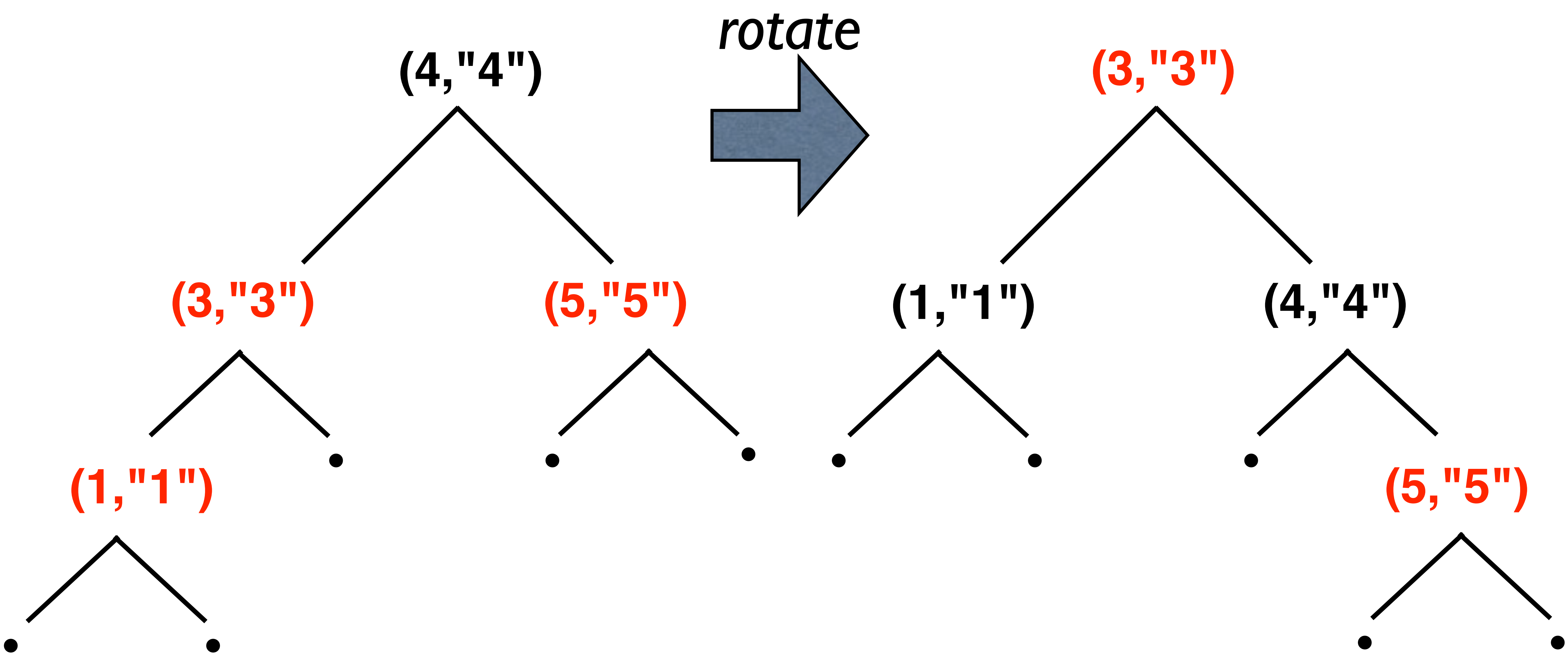
=>*



re-balance

- The problem: we're trying to combine an *almost* red-black tree and a red-black tree
  - The left child is ***almost*** red-black
    - Has a red root and its left child is red
    - That's the only *well-red violation*
- We can *re-balance*, by *rotating the tree* at the root and *re-coloring*, to obtain a red-black tree...

balancing



(not red-black) (red-black)

same black height
same traversal list
same lookup behavior

results

- The original tree had an *almost* red-black child, and the rebalanced tree *is* red-black
- Same traversal lists!
- Same lookup behavior!

rebalancing ***restores*** the rbt property
and ***respects*** the representation

more generally

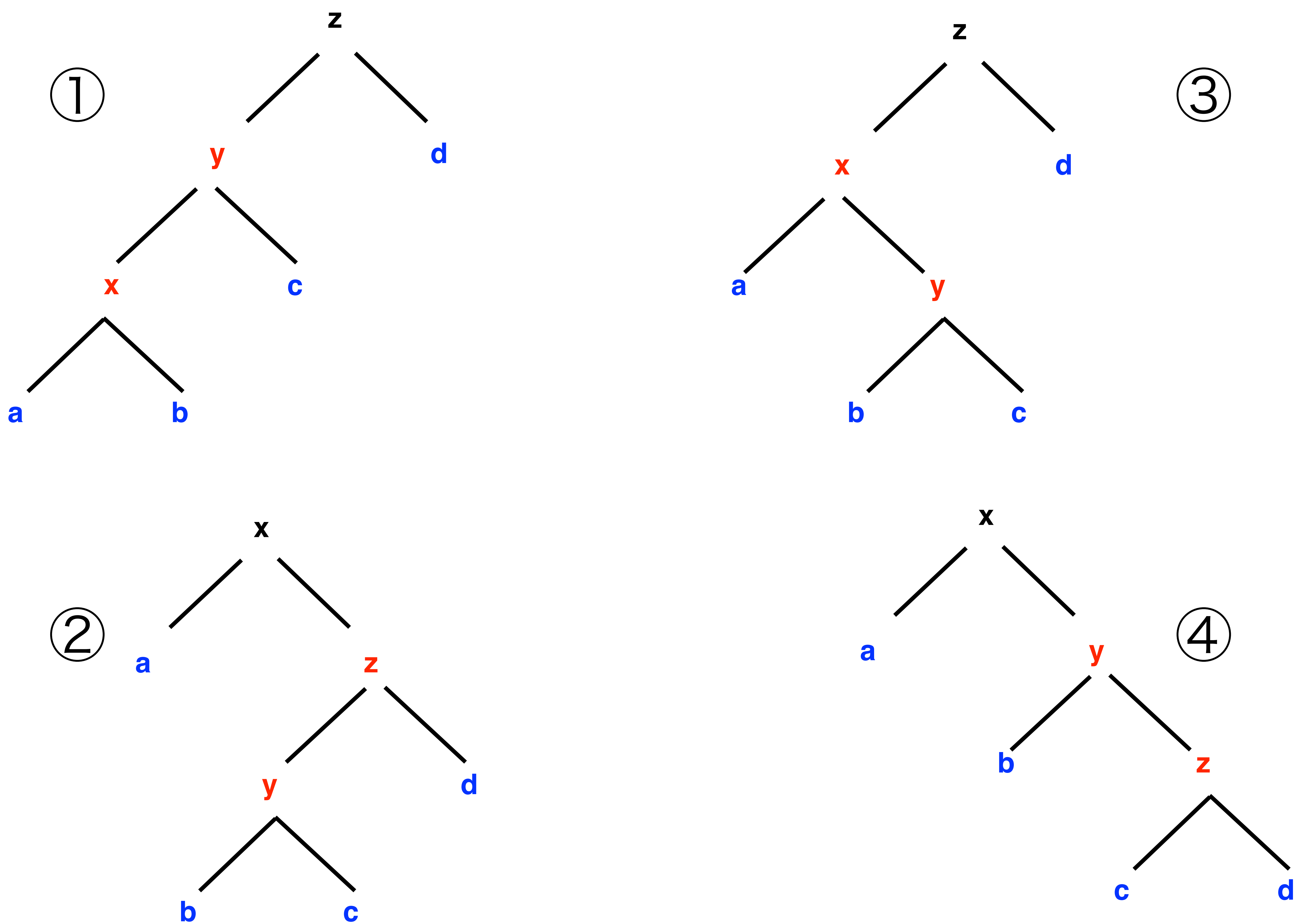
- When doing a new insert into a red-black tree there are 4 awkward cases

① ***left*** child *red*
left-left grandchild *red*

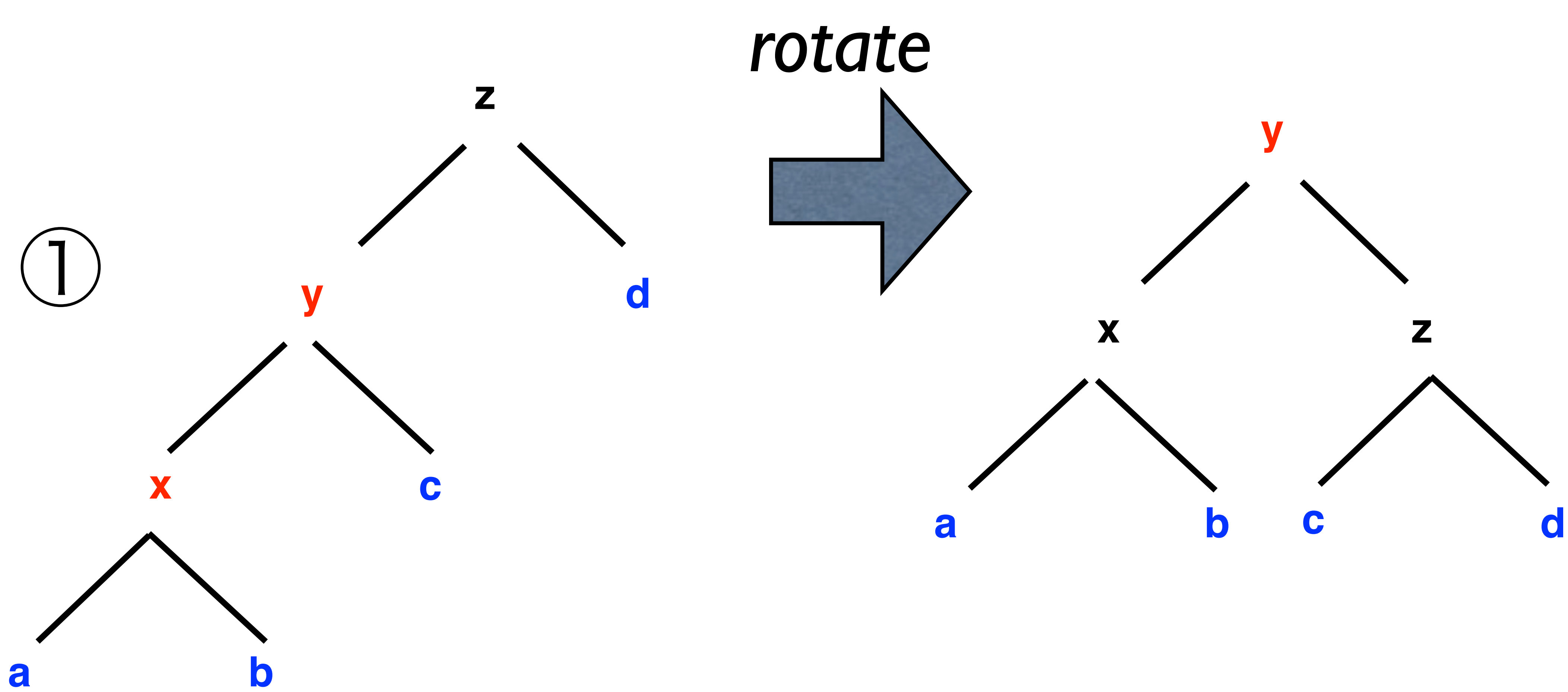
right child *red* ②
right-*left* grandchild *red*

③ ***left*** child *red*
left-*right* grandchild *red*

right child *red* ④
right-right grandchild *red*

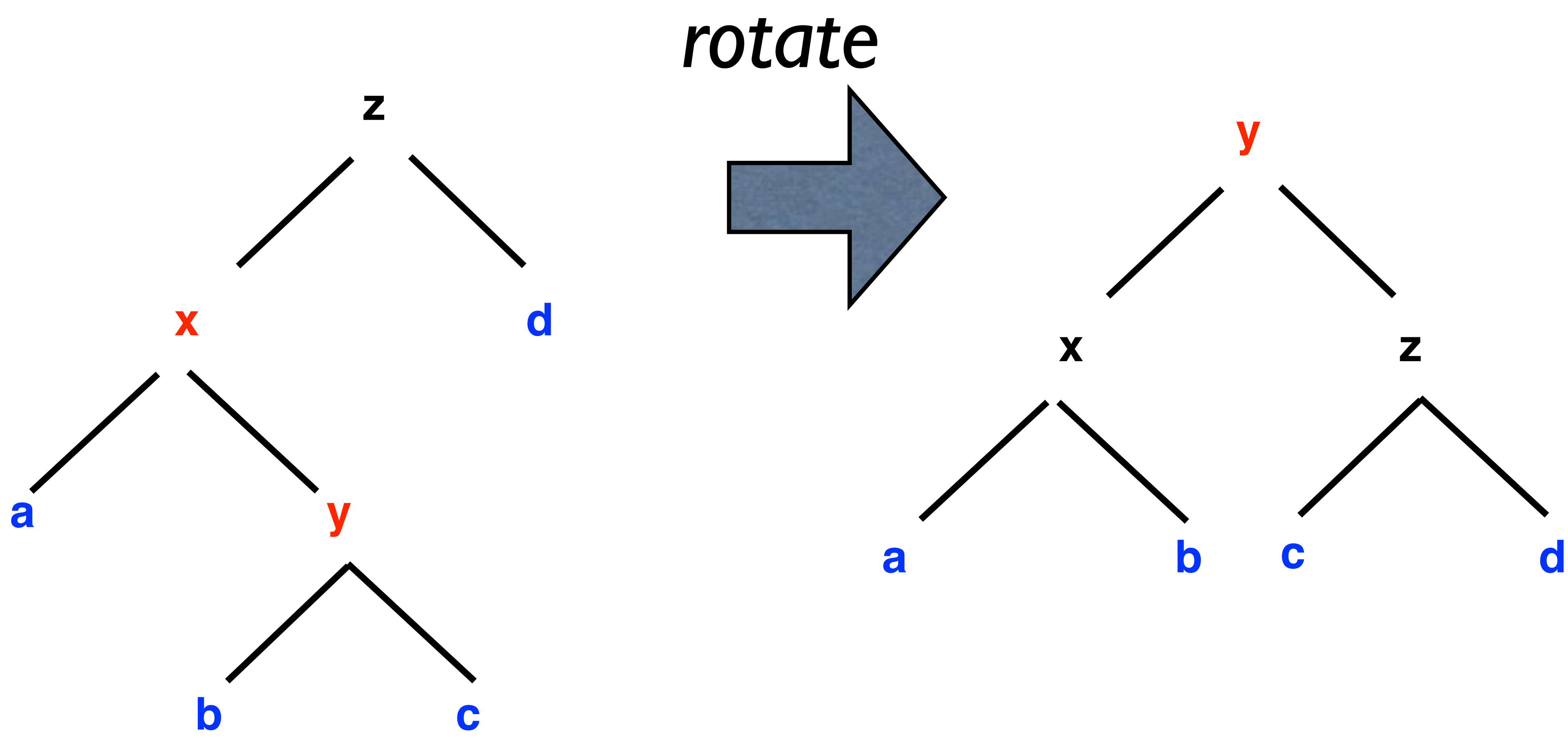

①
z
y
d
x
c
a
b
②
x
a
z
y
d
b
c
③
z
x
d
a
y
b
c
④
x
a
y
b
z
c
d

balancing



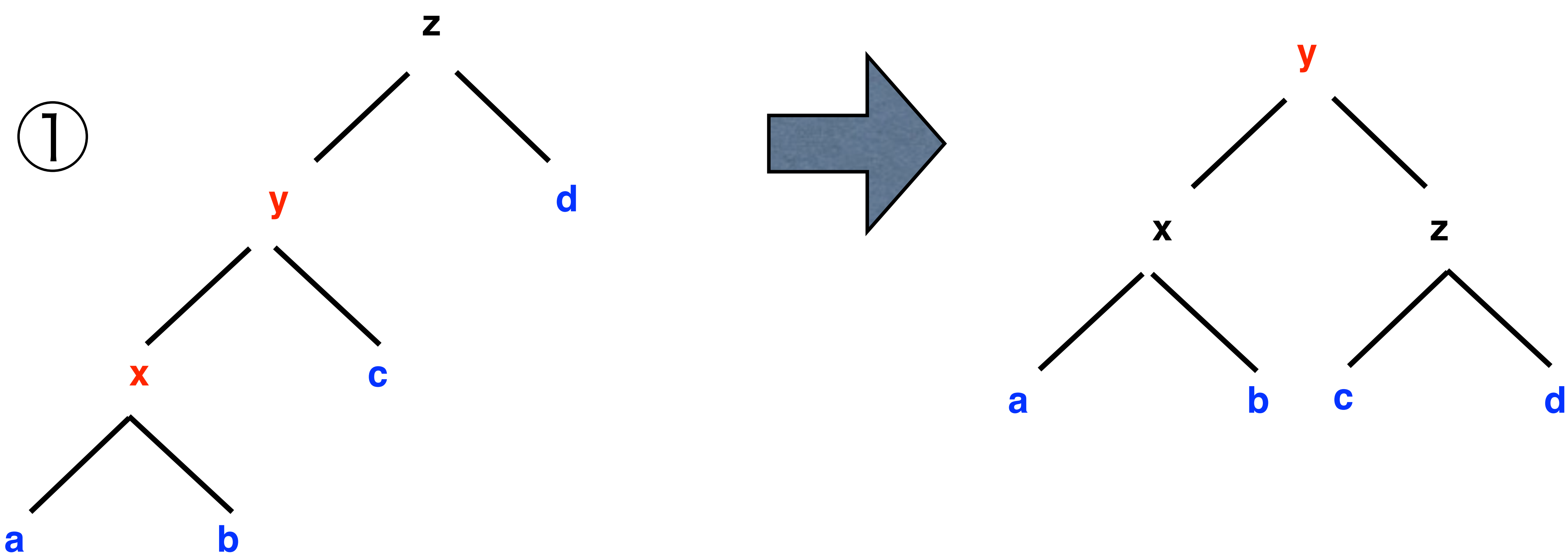
balance

③



and so on
(all four cases)

using patterns



Node(Node(Node(a, (Red, x), b), (Red, y), c), (Black, z), d)

⇒ Node(Node(a, (Black, x), b), (Red, y), Node(c, (Black, z), d))

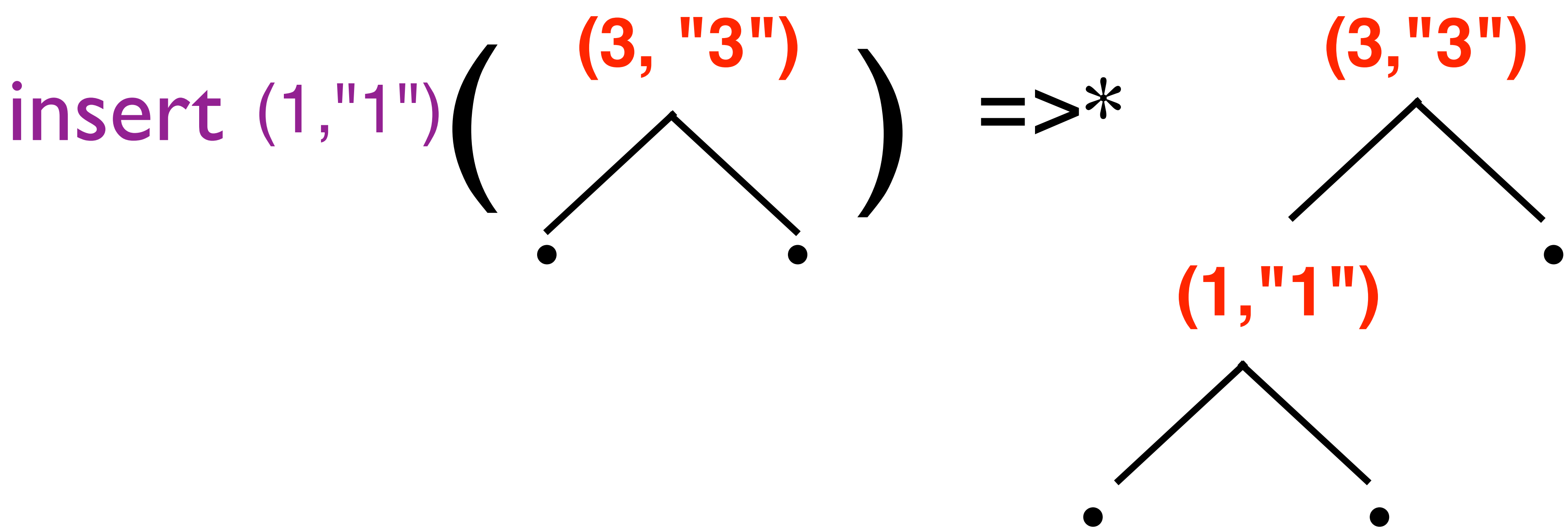
(similarly for all four cases)

balance

balance : 'a tree * (color * (Key.t * 'a)) * 'a tree -> 'a tree * (color * (Key.t * 'a)) * 'a tree

```
fun balance (Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d)                ①
        = (Node(a,(Black,x),b),  (Red,y), Node(c,(Black,z),d))
 |   balance (a,(Black,x),  Node(Node(b,(Red,y),c), (Red, z), d))               ②
        = (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z), d))
 |   balance (Node(a, (Red,x), Node(b,(Red,y),c)),  (Black,z), d)               ③
        = (Node(a,(Black,x),b), (Red, y), Node(c,(Black,z),d))
 |   balance (a, (Black,x), Node(b, (Red,y), Node(c,(Red,z), d)))               ④
        = (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z), d))
 |   balance p = p
```

not done yet



an *almost* red-black tree

(the only *wellred* failure is at the root node)

Not handled by **balance**

almost there

- It’s easy to turn an *almost* rbt with just a root violation into a truly red-black tree...
  - ***blacken*** its root!

```
(*  blackenroot : 'a dict -> 'a dict  *)
fun blackenroot Leaf = Leaf
  |    blackenroot (Node(l, (_, (k,v)), r))
                      = Node(l, (Black, (k,v)), r);
```

insert

```
fun ins (k, v) Leaf  = Node(Leaf, (Red, (k, v)), Leaf)
  |  ins (k, v) (Node(l, (c, (k', v')), r)) =
     case Key.compare(k, k') of
         EQUAL      => Node (l, (c, (k,v)), r)
       | LESS       => Node (balance(ins (k,v) l, (c, (k',v')), r))
       | GREATER => Node (balance(l, (c, (k',v')), ins (k,v) r))

fun insert (k, v) D = blackenroot (ins (k, v) D)
```

We renamed the *recursive inserting* function ins...

insert calls ins first, which balances at every recursive call, and finishes with blackenroot

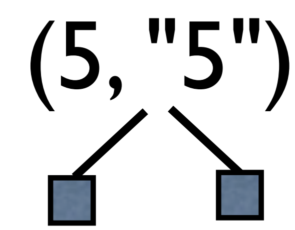
trav

- Traversal ignores color

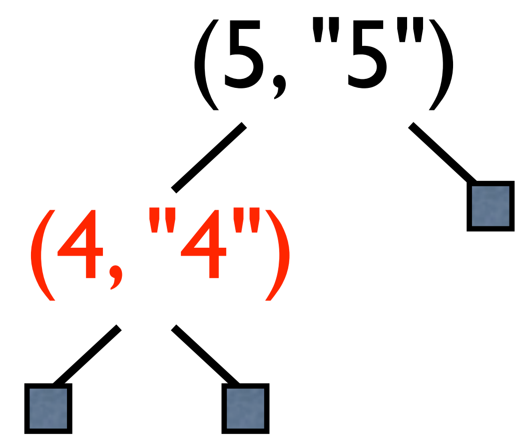
```
fun trav Leaf = [ ]
 |     trav (Node(l, (_, (k,v)), r))
                = (trav l) @ (k,v) :: (trav r)
```

```
functor RBTDict(Key : ORDERED) : DICT =
 struct
   structure Key : ORDERED = Key
   datatype color = Red | Black
   datatype 'a tree = Leaf | Node of 'a tree * (color * (Key.t * 'a)) * 'a tree
   type 'a dict = 'a tree
   val empty = Leaf
   fun lookup Leaf k = NONE
    |   lookup (Node (l, (_,(k', v')), r)) k  =
          case Key.compare (k, k') of
             EQUAL          =>  SOME v'
           | LESS           =>  lookup l k
           | GREATER        =>  lookup r k

    fun balance ...
    fun blackenroot ...
    fun ins ...
    fun insert (k,v) D = blackenroot(ins (k,v) D)

    fun trav Empty = [ ]
     |     trav (Node(l, (_,(k,v)), r)) = (trav l) @ (k,v) :: (trav r)
  end
```

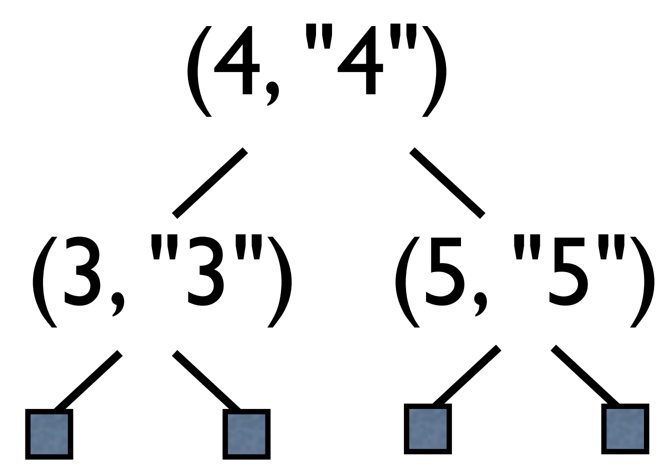
val D_1 = insert (5, "5") empty



val D_2 = insert (4, "4") D_1



val D_3 = insert (3, "3") D_2



■ is Leaf

representation independence

- Every value built from *empty* and *insert* is a ***red-black tree***

- Ignoring colors, also a ***binary search tree***

Bst.insert (k_n,v_n) (...(Bst.insert (k_1,v_1) Bst.empty)...)

and

Rbt.insert (k_n,v_n) (...(Rbt.insert (k_1,v_1) Rbt.empty)...)

represent the *same dictionary*