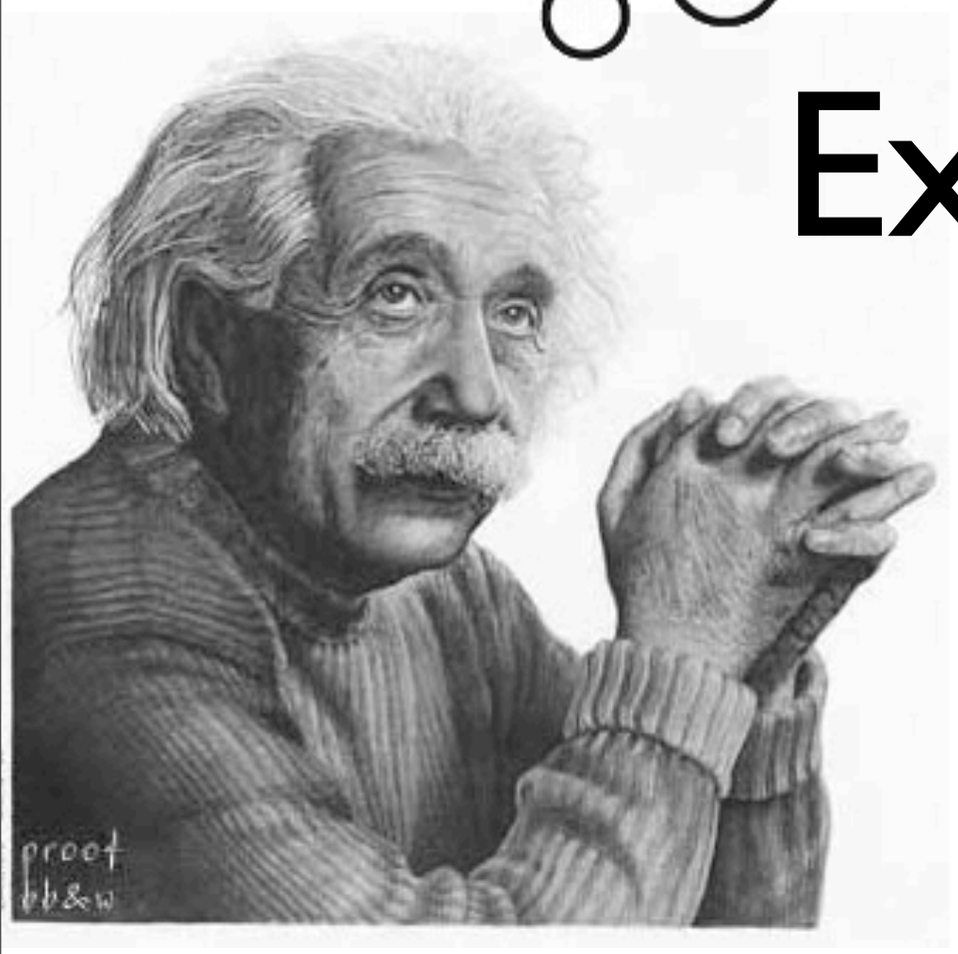


Extreme Cleverness

Functional Data Structures in Scala

Agenda

- Functional data structures
- Implementations
  - Sequential
  - Associative
- Modern computer architecture

Functional Data Struct.

- Immutable, immutable, immutable

Functional Data Struct.

- Immutable, immutable, immutable
- What we want…
  - Comparable asymptotic performance
  - Non-degraded versions
    - (full persistence)

Functional Data Struct.

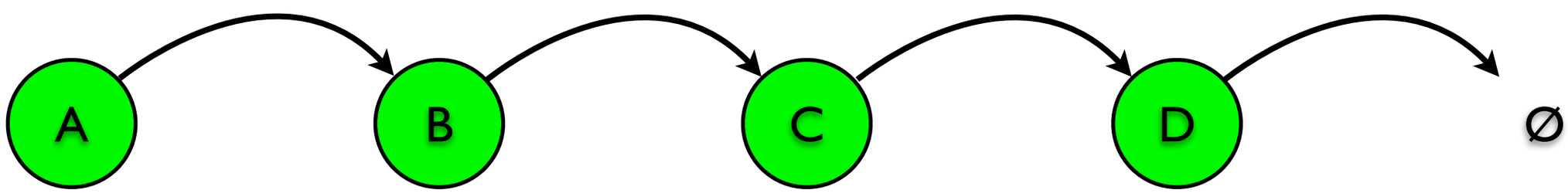
- Immutable, immutable, immutable
- What we want…
  - Comparable asymptotic performance
  - Non-degraded versions
    - (full persistence)
- Structural sharing

Sequential

- Singly-Linked List
- Banker's Queue
- 2-3 Finger Tree

Singly-Linked List

```
List(a, b, c, d)
```



Complexity

$O(1)$	$O(\log n)$	$O(n)$
`first`		`append`
`prepend`		`concat`
		`insert`
		`last`
		`nth`

Anatomy

- A list is *either*…
  - A “cons” cell with a value and a tail
  - An empty list, called “nil”
- These are the only cases!

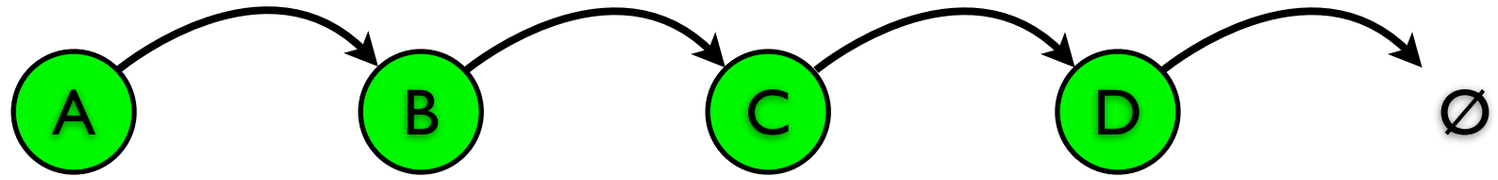
```scala
sealed trait List[+A] {
  def ::[B >: A](b: B): List[B] =
    new ::(b, this)
}

case class ::[+A](hd: A, tail: List[A]) extends List[A]
case object Nil extends List[Nothing]
```

Structural Sharing

```
val foo =  a :: b :: c :: d :: Nil
```

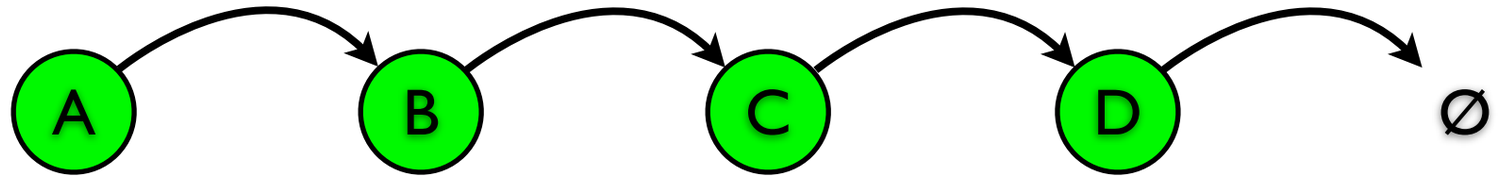
Structural Sharing

```
val foo =
```



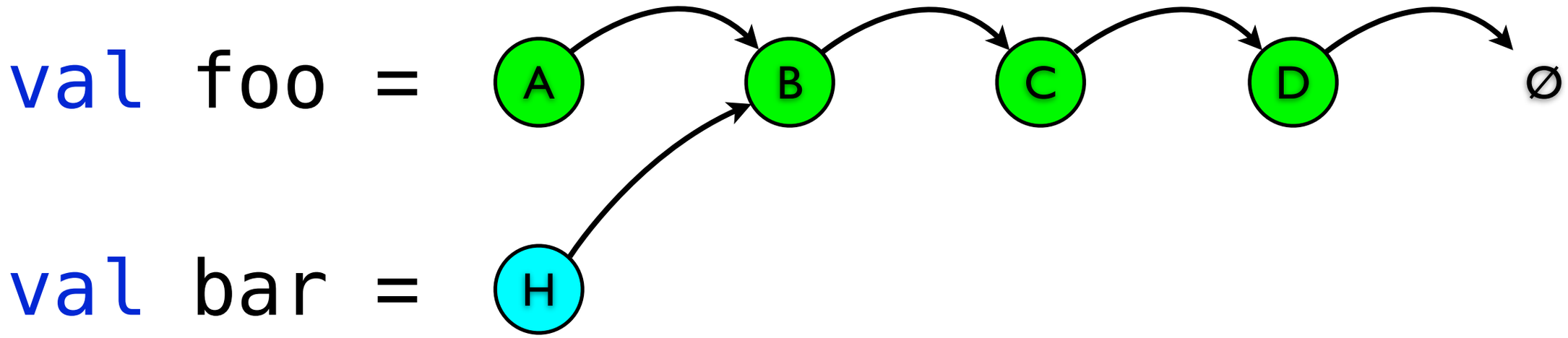
Structural Sharing

```
val foo =
```



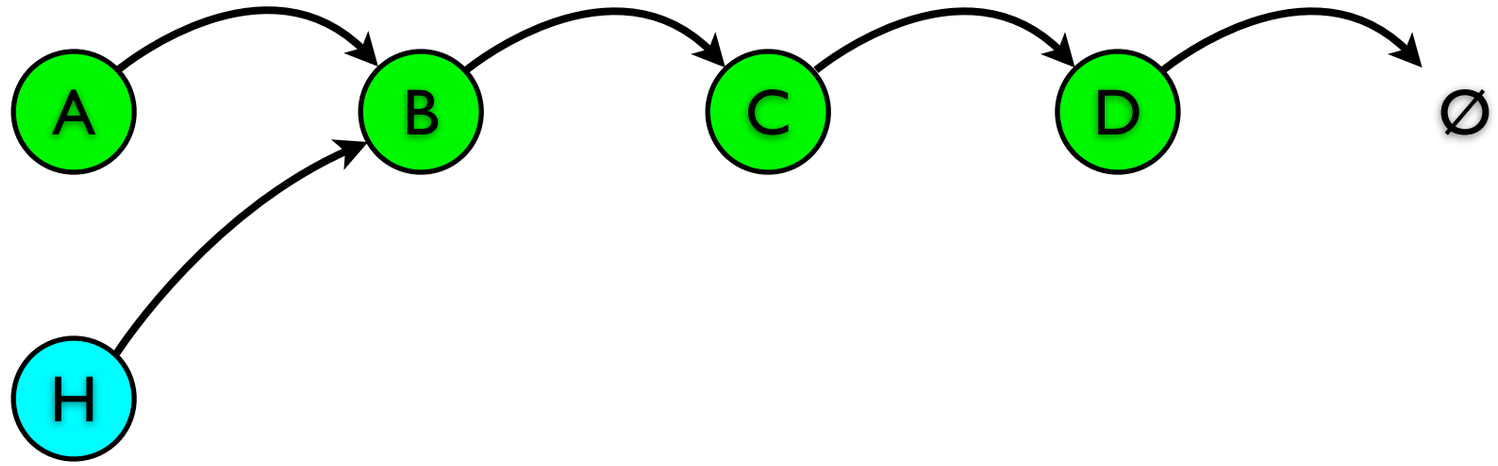

```
val bar =  h :: foo.tail
```

Structural Sharing



Structural Sharing

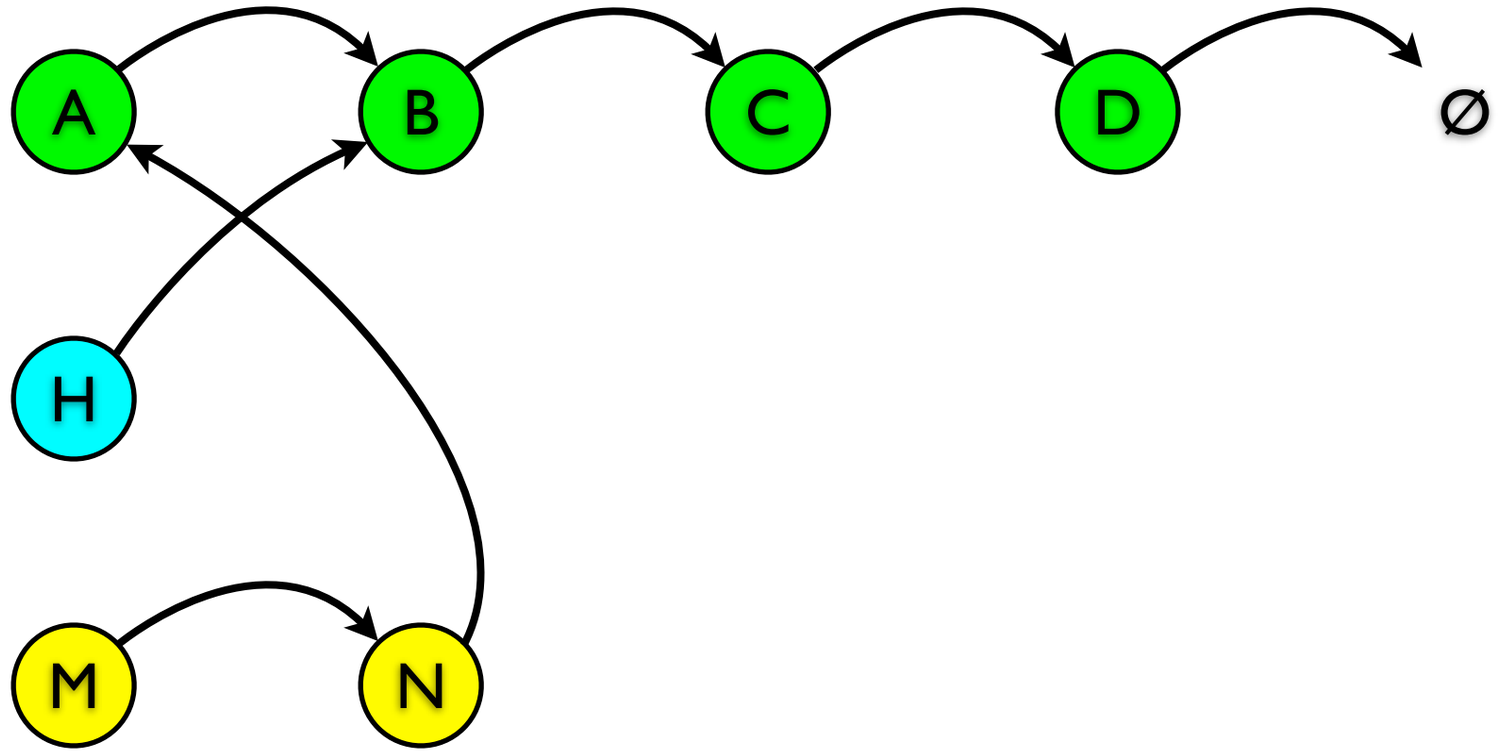
```
val foo =
```




```
val bar =

val baz =  m :: n :: foo
```

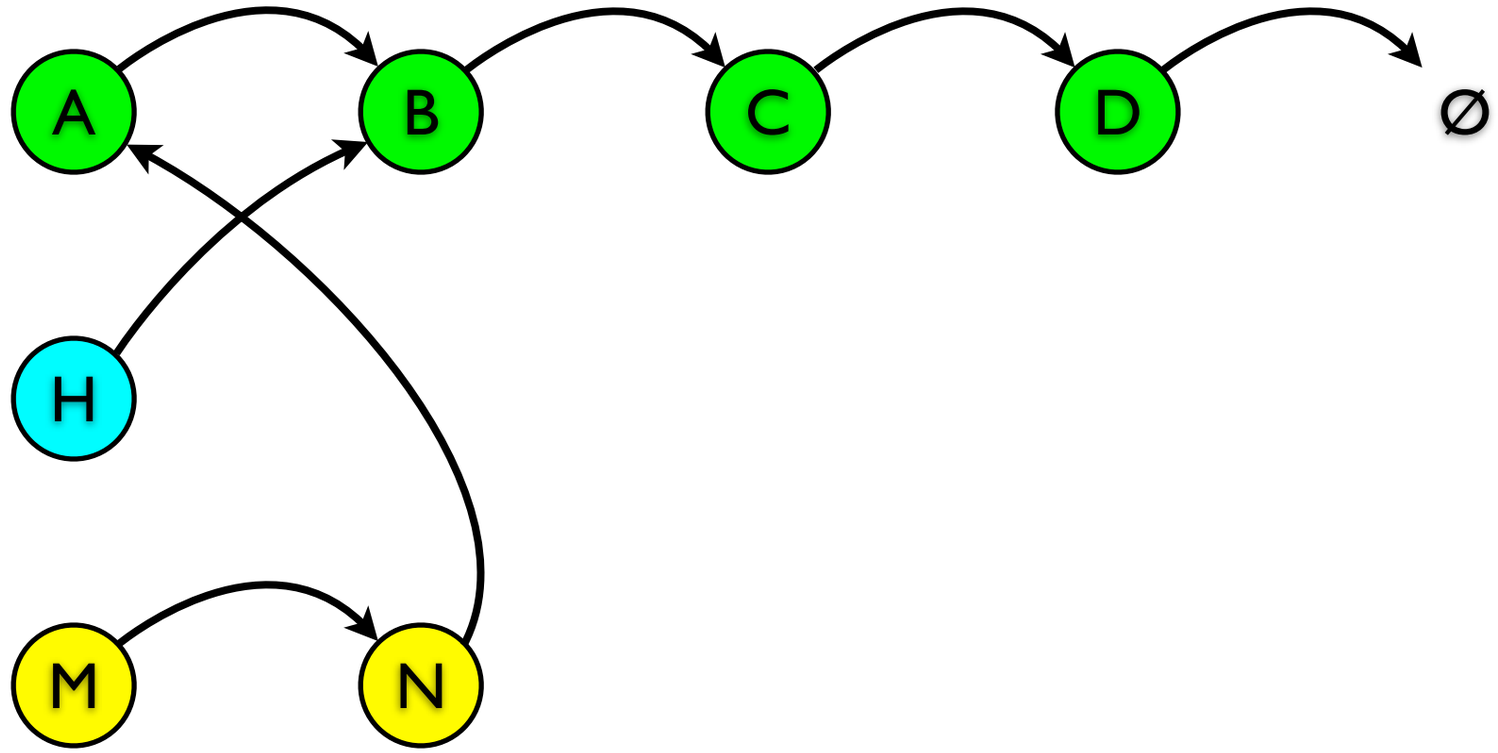
Structural Sharing

```
val foo =
val bar =
val baz =
```



Structural Sharing

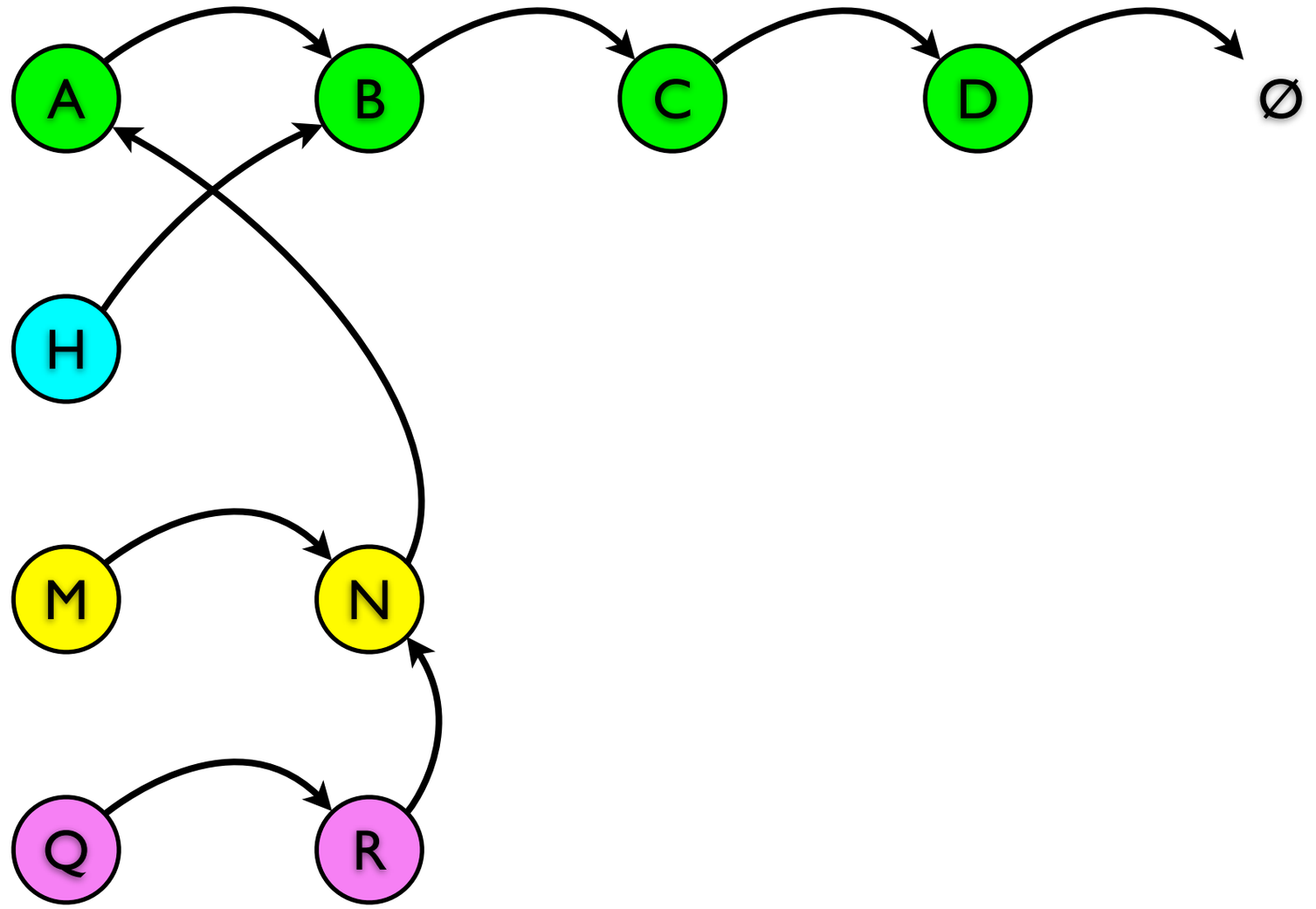
```
val foo =
```




```
val bar =

val baz =

val raz =  q :: r :: baz.tail
```

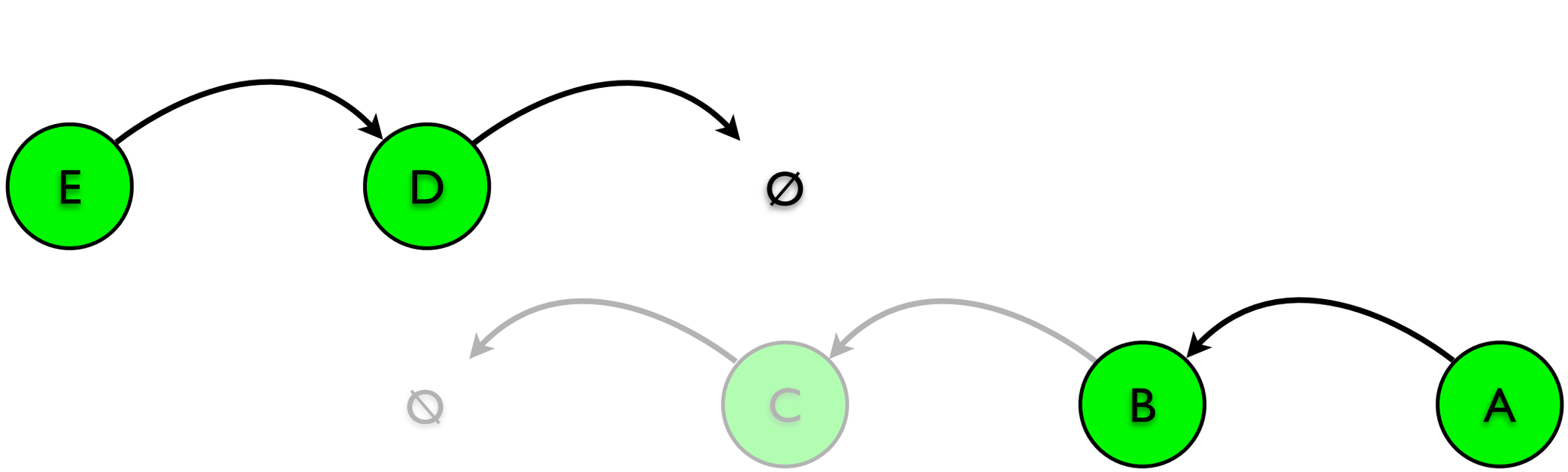
Structural Sharing

```
val foo =
val bar =
val baz =
val raz =
```



Banker's Queue [1]

`Queue(a, b, c, d, e)`



Motivation

- We want a functional queue
- Linked list is obvious
  - `prepend` and `last` are opposing
  - One will be $O(1)$, the other $O(n)$
- Can we have our cake and eat it too?

Complexity

$O(1)$	$O(\log n)$	$O(n)$
`append`		`concat`
`last`		`first`
`prepend`		`insert`
		`nth`

Complexity

amortized

$O(1)$	$O(\log n)$	$O(n)$
`append`		`concat`
`last`		`first`
`prepend`		`insert`
		`nth`

Anatomy

- Naïve persistent queue
- Two lazy singly-linked lists
  - Front list (for dequeue)
  - Rear list (for enqueue)
- Periodically reverse rear into the front
- Lazy amortization

Amortization

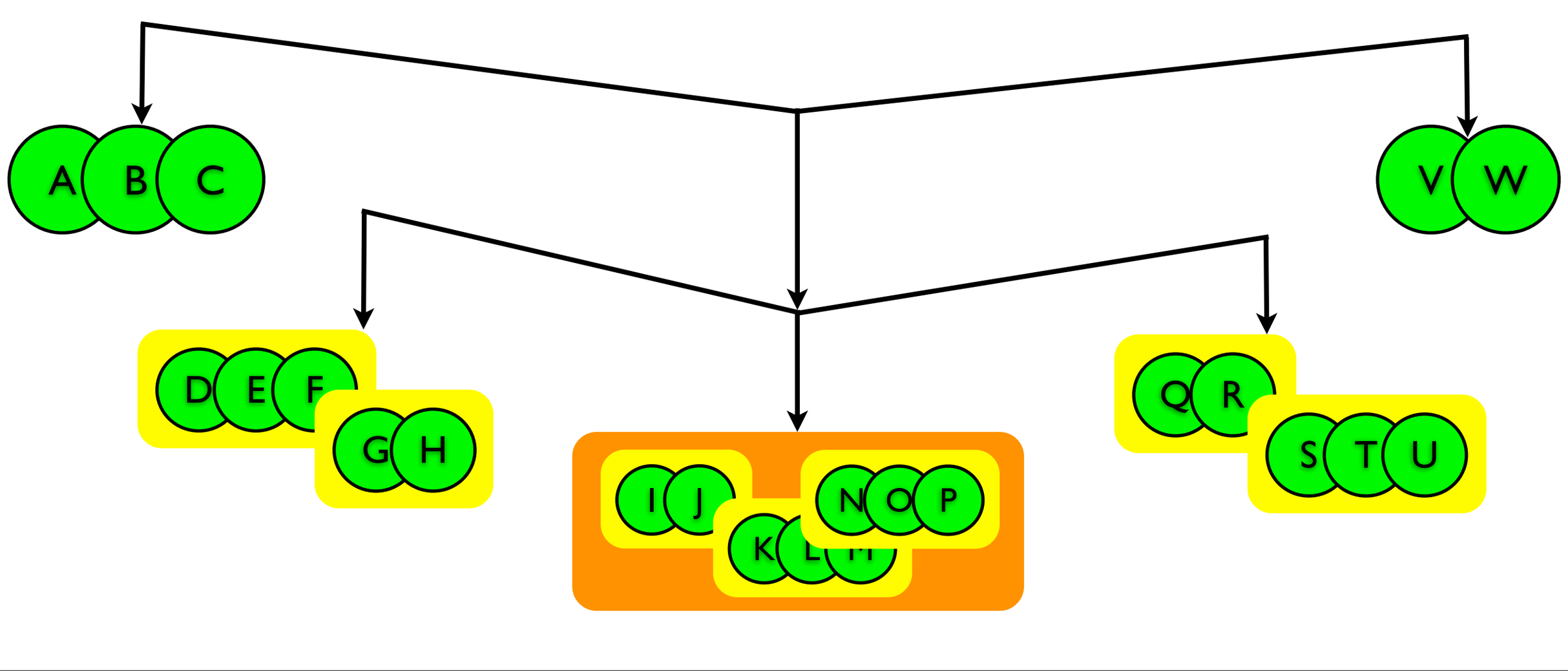
- *Most* operations are legitimately fast
  - *Few* operations are very slow
- Laziness distributes the work
- Net result: constant factor degradation
  - Translation: the net average is fast
- Also works without laziness!

```scala
class BankersQueue[+A](fsize: Int, front: Stream[A],
                       rsize: Int, rear: Stream[A]) {
  …
}

object BankersQueue {
  def check[A](q: BankersQueue[A]) = {
    if (q.rsize <= q.fsize) {
      q                    // already valid
    } else {
      val fsize2 = q.fsize + q.rsize
      val front2 = q.front ++ q.rear.reverse

      new BankersQueue(fsize2, front2, 0, Stream())
    }
  }
}
```

```
class BankersQueue[+A](fsize: Int, front: Stream[A],
                       rsize: Int, rear: Stream[A]) {

  def enqueue[B >: A](b: B) =
    check(new BankersQueue(fsize, front,
                           rsize + 1, b #:: rear))

  def dequeue = front match {
    case hd #:: tail => {
      val rem = new BankersQueue(fsize - 1, tail,
                                 rsize, rear)
      (hd, check(rem))
    }

    case _ => throw new NoSuchElementException
  }
}
```

2-3 Finger Tree [3]

```
FingerTree('A' to 'W': _*)
```

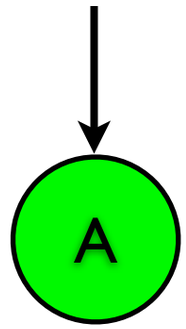


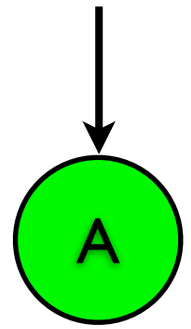
Complexity

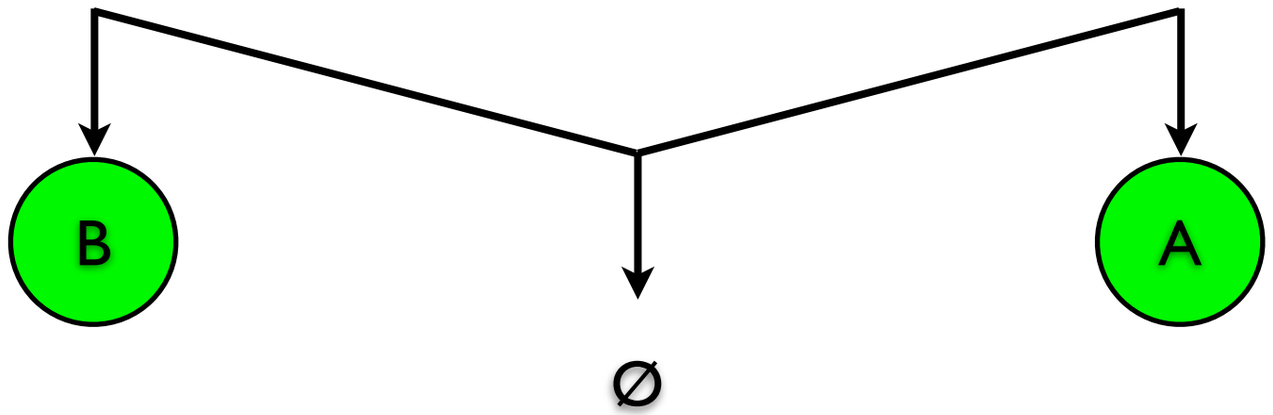
$O(1)$	$O(\log n)$	$O(n)$
`append`	`insert`	`concat`
`first`	`nth`	
`last`		
`prepend`		

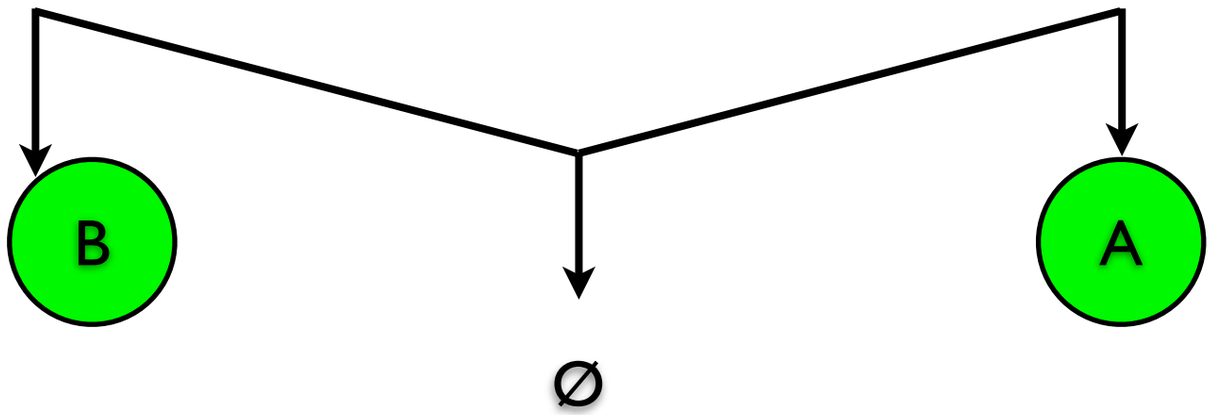
Anatomy

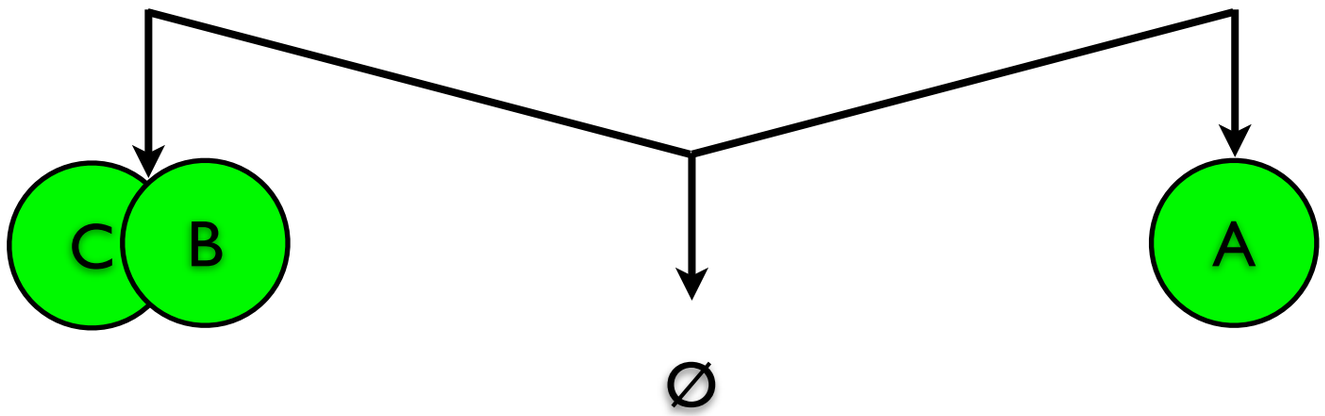
- Ideal persistent deque
- Digits of length 1, 2, 3 or 4
    - Head and tail
- Branching factor of 2 or 3
- Recursive tree body

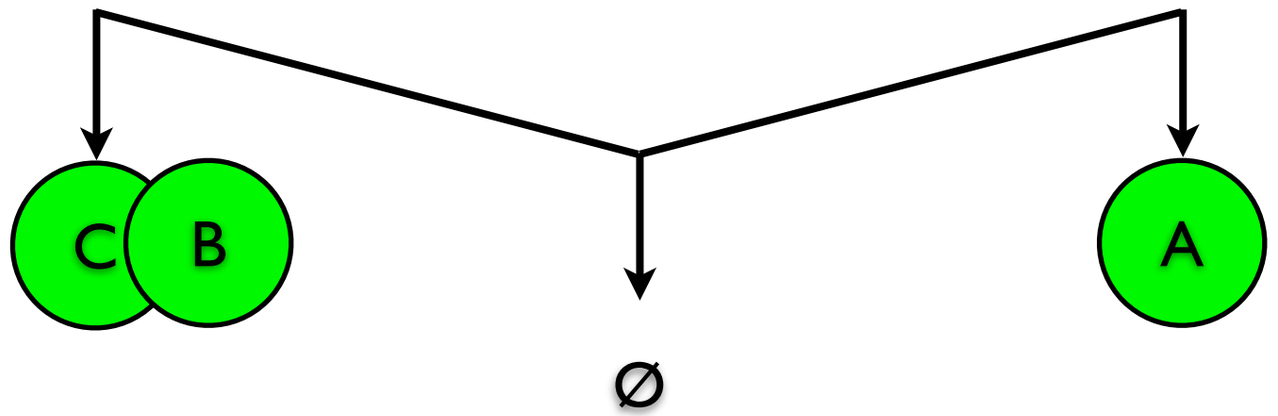

A

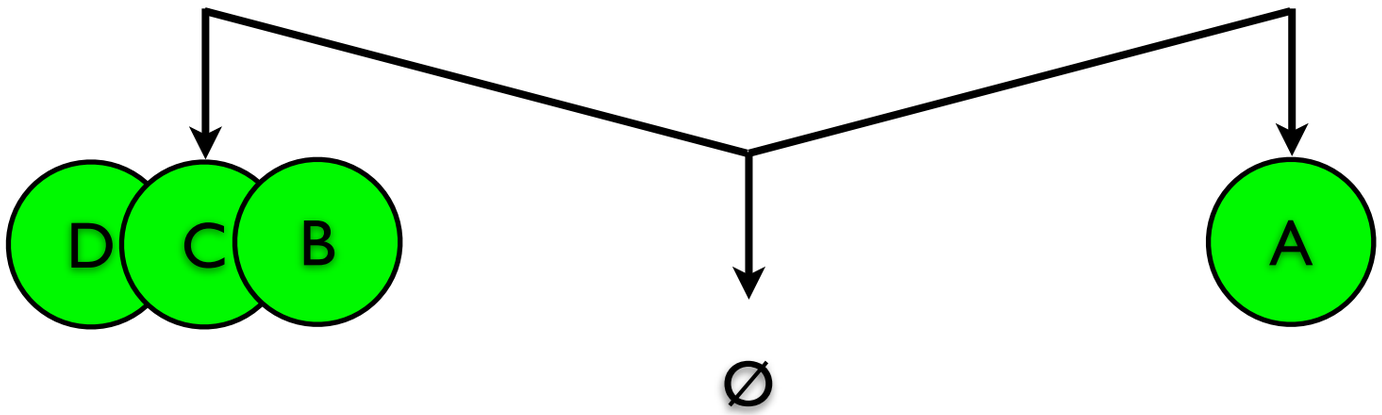

A

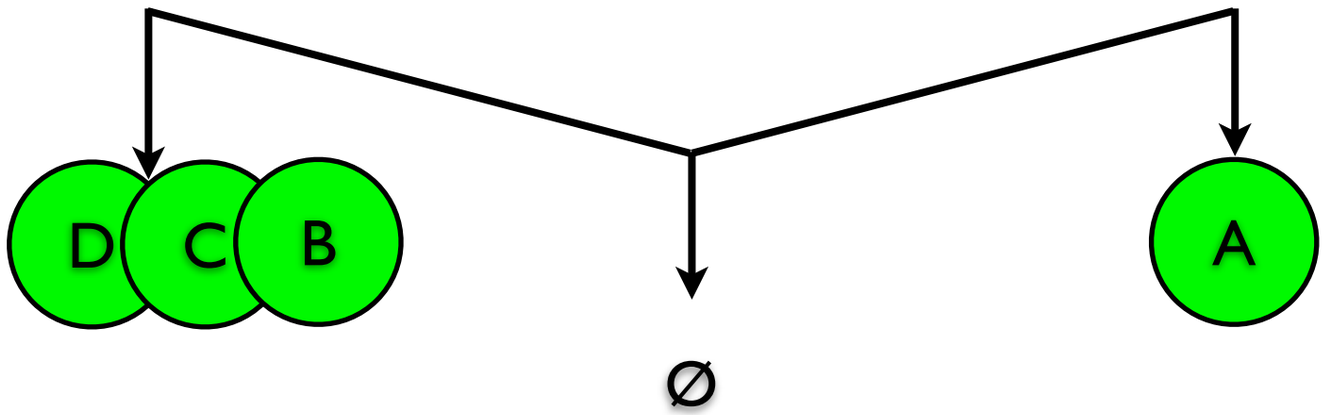

B
A
∅


B
A
∅

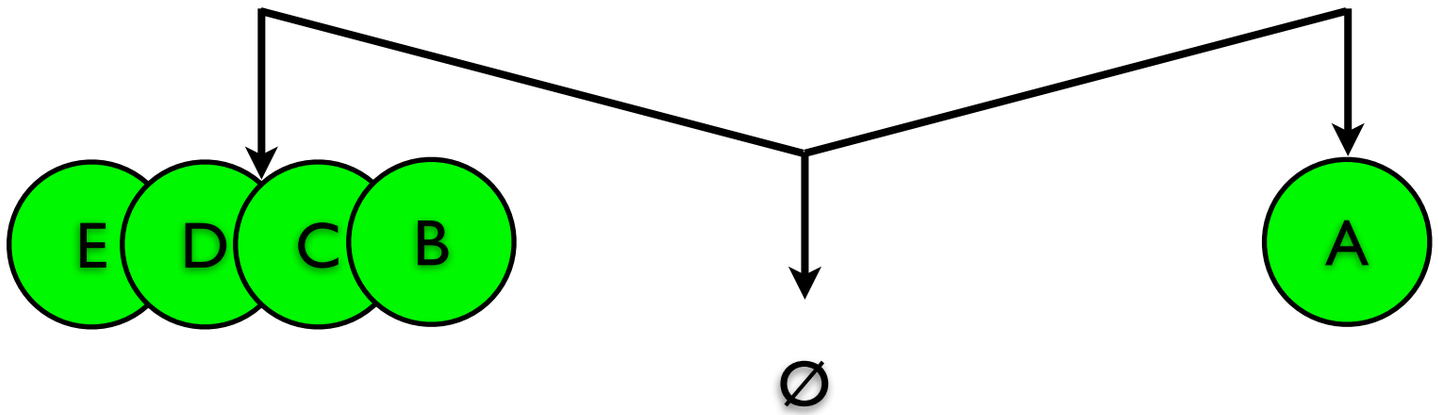

C
B
A
∅


C
B
A
∅

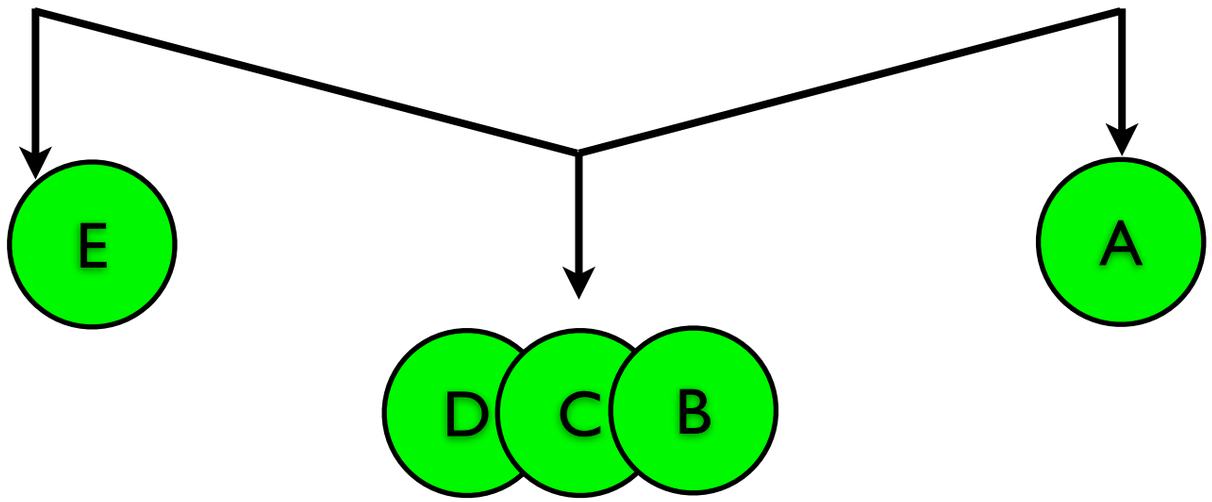

D
C
B
A
∅

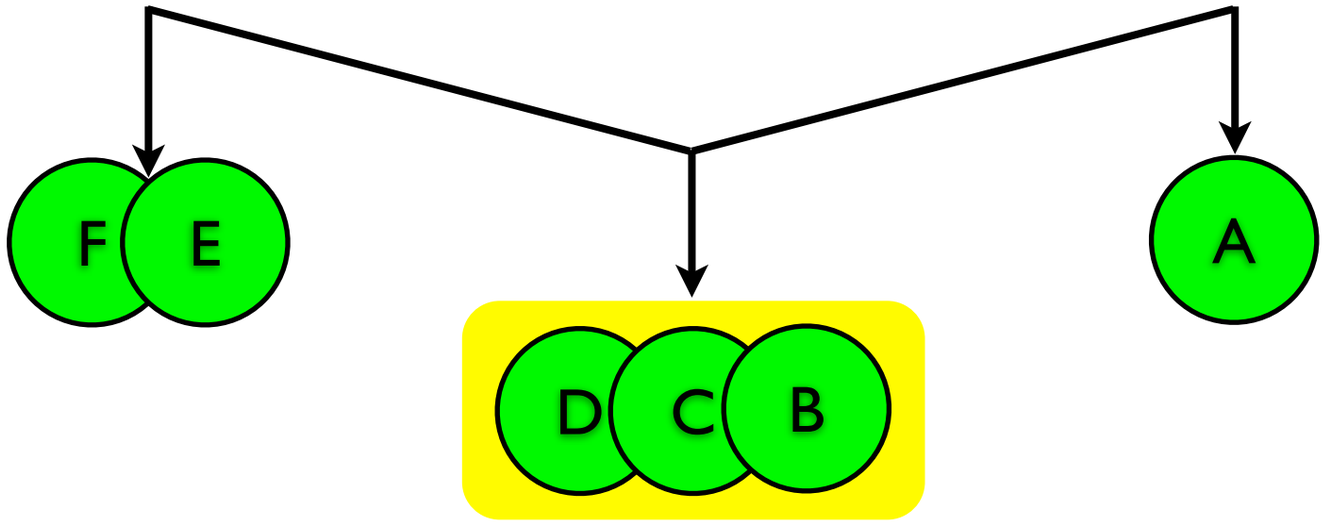

D
C
B
A

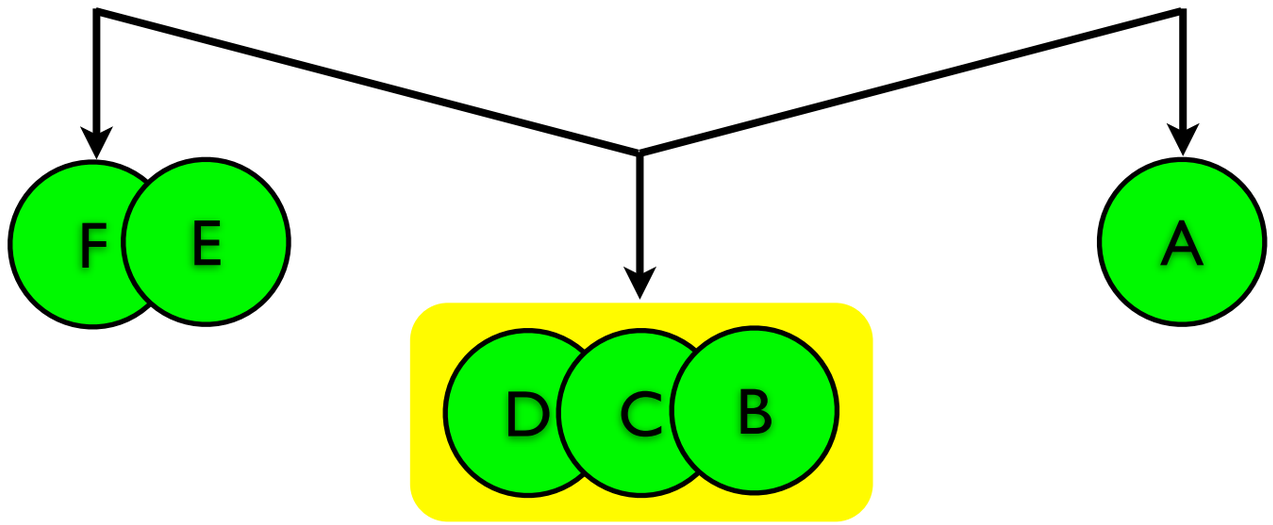
∅

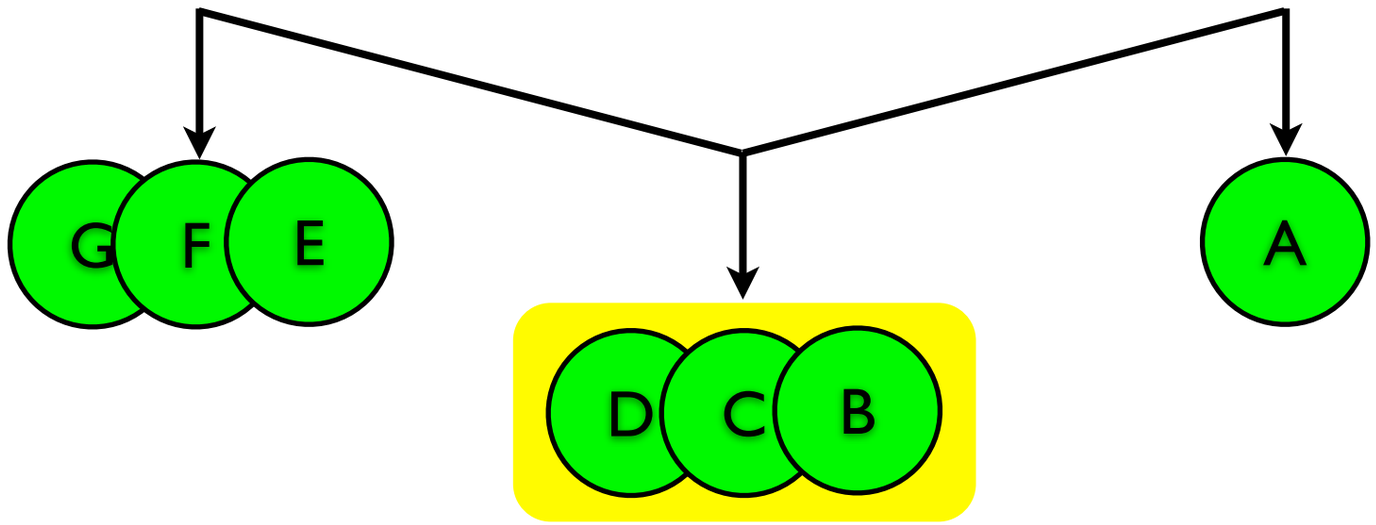

E
D
C
B
A

∅


E
A
D
C
B


F
E
A
D
C
B


F
E
A
D
C
B


G
F
E
D
C
B
A

```
sealed trait FingerTree[+A] {
  …
}

case class Single[+A](a: A) extends FingerTree[A] {
  …
}

case class Deep[+A](prefix: Digit[A],
                    tree: FingerTree[Node[A]],
                    suffix: Digit[A])
    extends FingerTree[A] {
  …
}

case object Empty extends FingerTree[Nothing] {
  …
}
```

```scala
sealed trait Digit[+A]

case class One[+A](a1: A) extends Digit[A]
case class Two[+A](a1: A, a2: A) extends Digit[A]
case class Three[+A](a1: A, a2: A, a3: A) extends Digit[A]
case class Four[+A](a1: A, a2: A, a3: A, a4: A)
    extends Digit[A]


sealed trait Node[+A]

case class Node2[+A](a1: A, a2: A) extends Node[A]
case class Node3[+A](a1: A, a2: A, a3: A) extends Node[A]
```

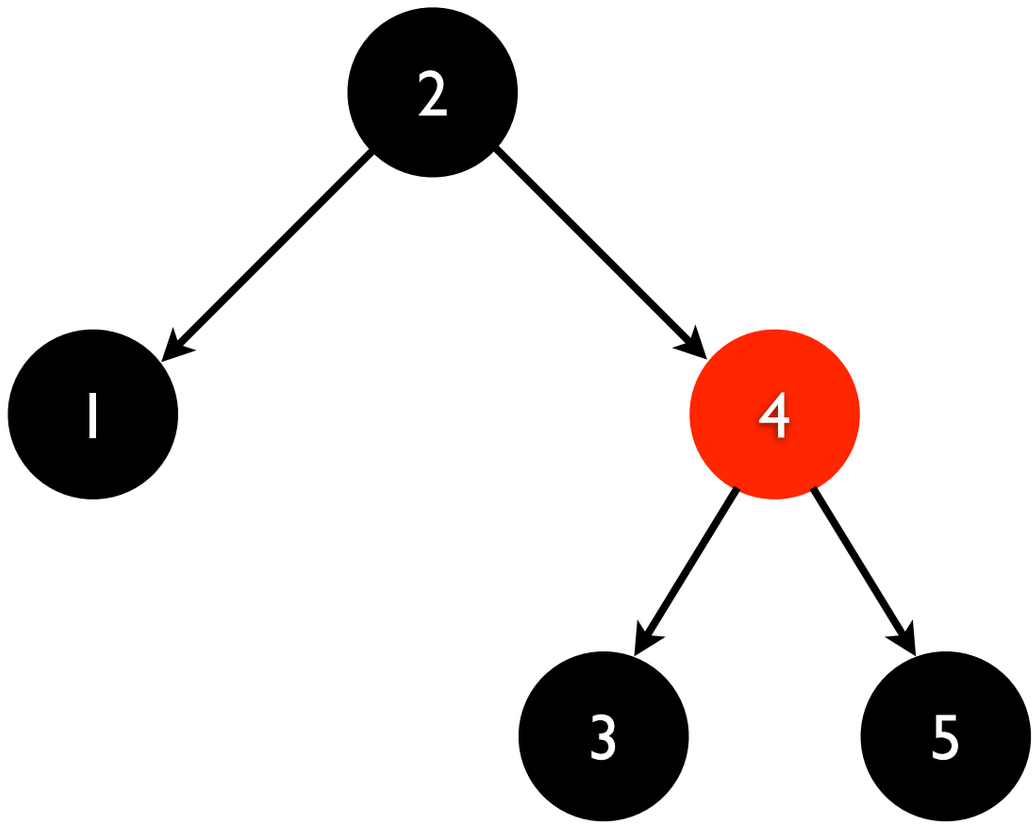
```scala
case class Deep[+A](...) extends FingerTree[A] {
  …
  def +:[B >: A](b: B) = prefix match {
    case Four(d, e, f, g) =>
      Deep(Two(b, d), Node3(e, f, g) +: tree, suffix)

    case _ => Deep(b +: prefix, tree, suffix)
  }

  def :+[B >: A](b: B) = suffix match {
    case Four(g, f, e, d) =>
      Deep(prefix, tree :+ Node3(g, f, e), Two(d, b))

    case _ => Deep(prefix, tree, suffix :+ b)
  }
  …
}
```

Associative

- Red-Black Tree

Red-Black Tree [2]

```
RedBlack(1, 2, 3, 4, 5)
```



Complexity

$O(1)$	$O(\log n)$	$O(n)$
	`get`	`intersect`
	`insert`	`union`
	`update`	

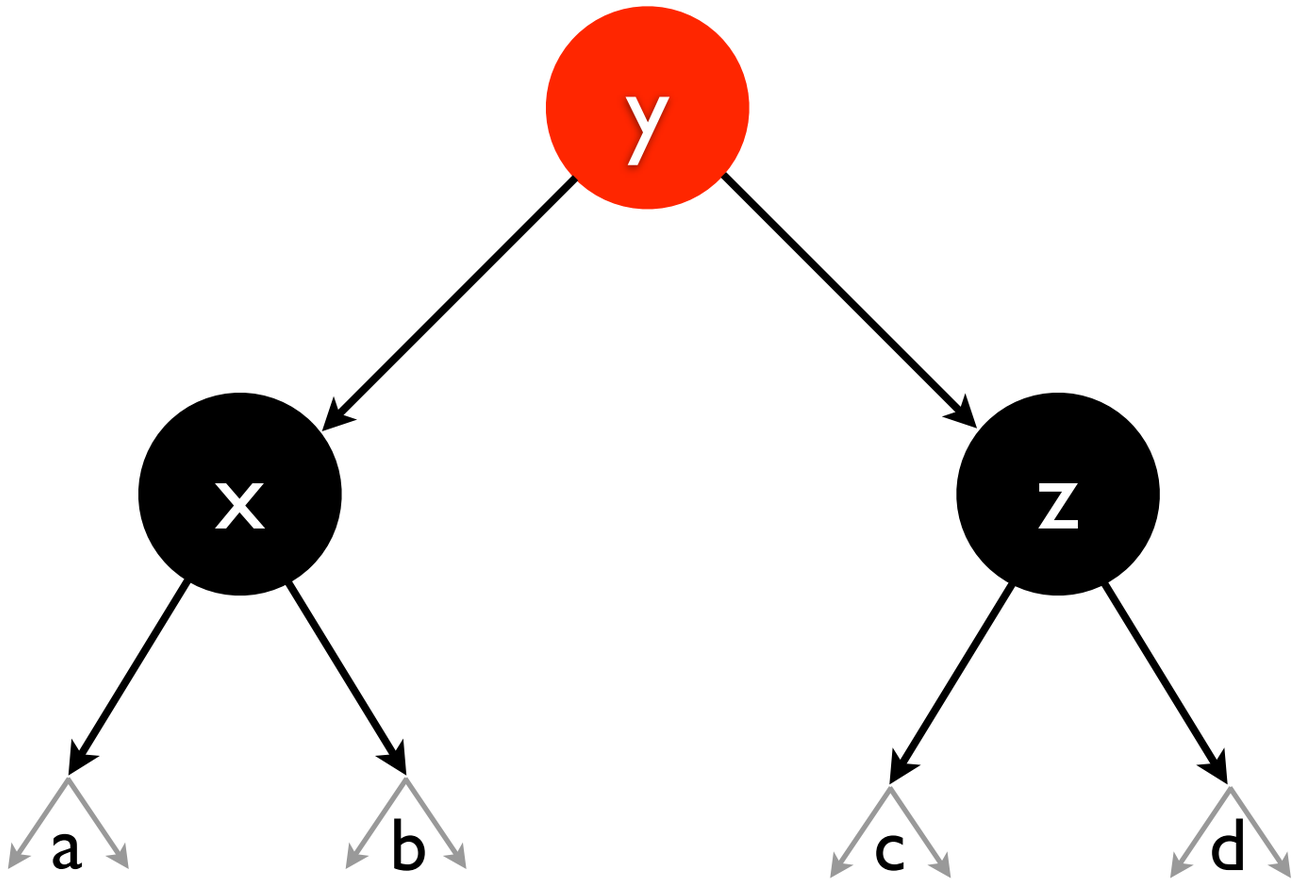
Anatomy

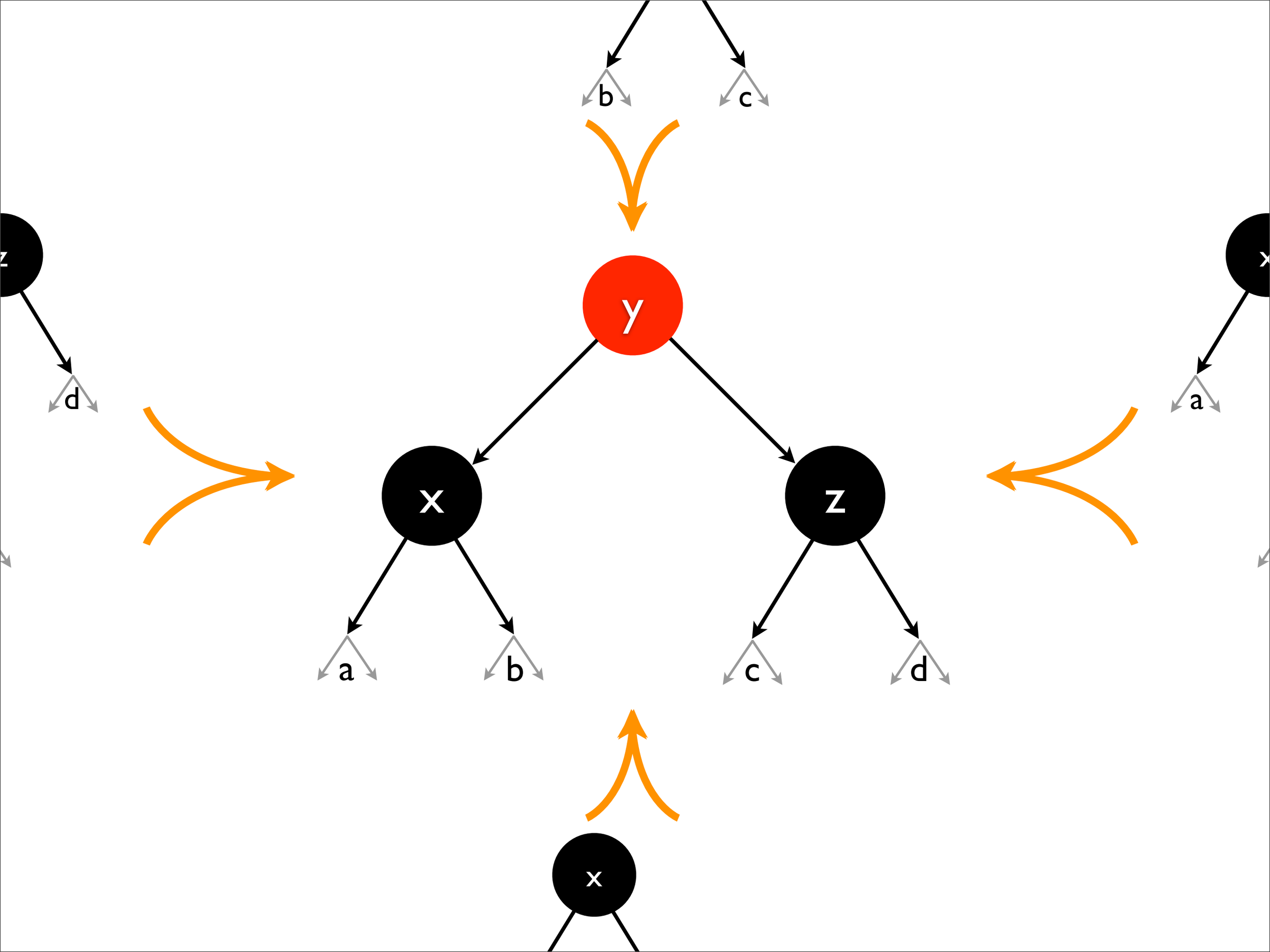
- Balanced binary search tree
- Invariants…
  - Every path from root to a leaf contains the *same* number of **black** nodes
  - No **red** node has a **red** parent
- Need to rebalance after any "update"

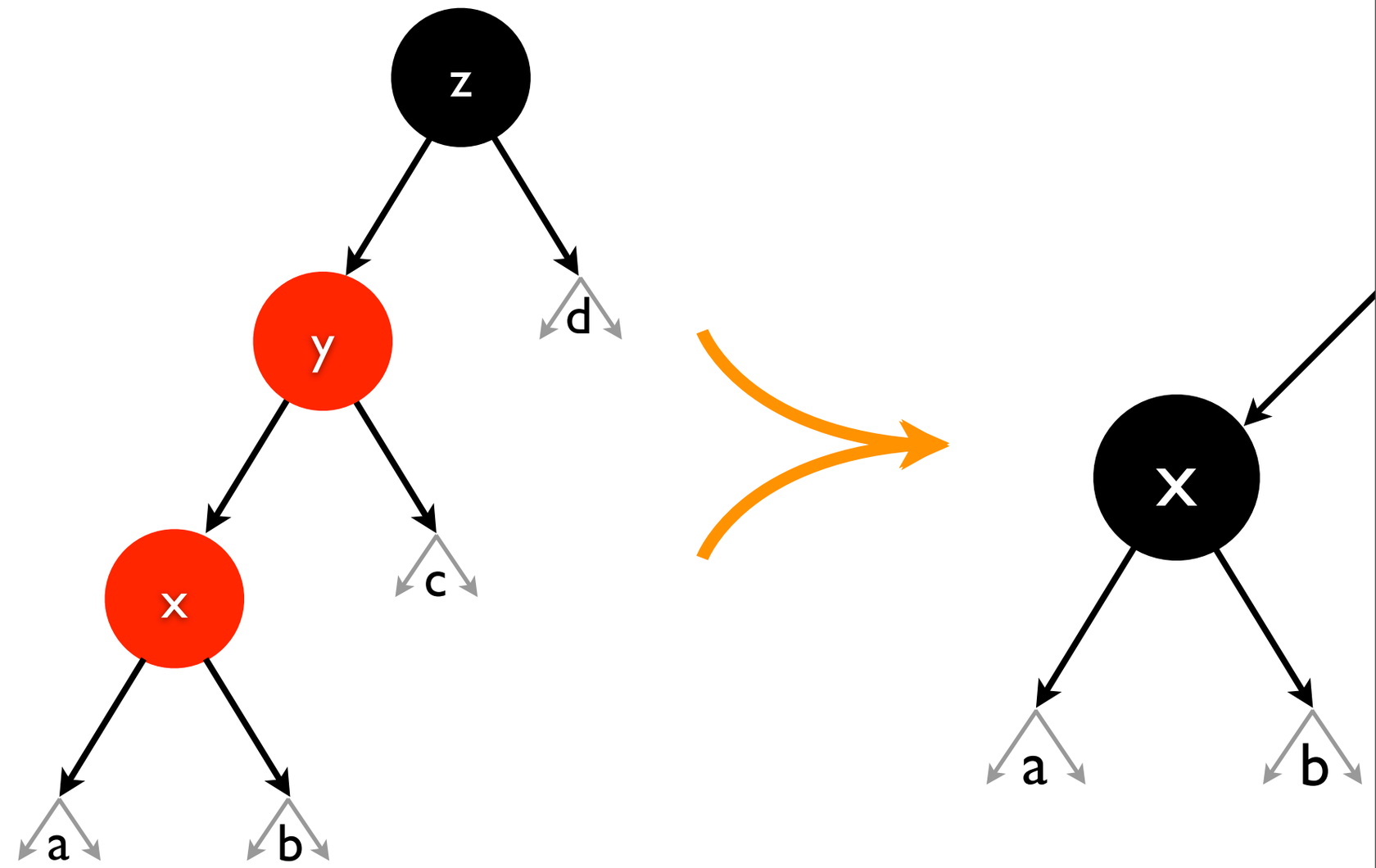
```scala
sealed abstract class Tree[K : Ordering, +V] {
  val isBlack: Boolean

  def left: Tree[K, V]

  def key: K
  def value: V

  def right: Tree[K, V]
}
```

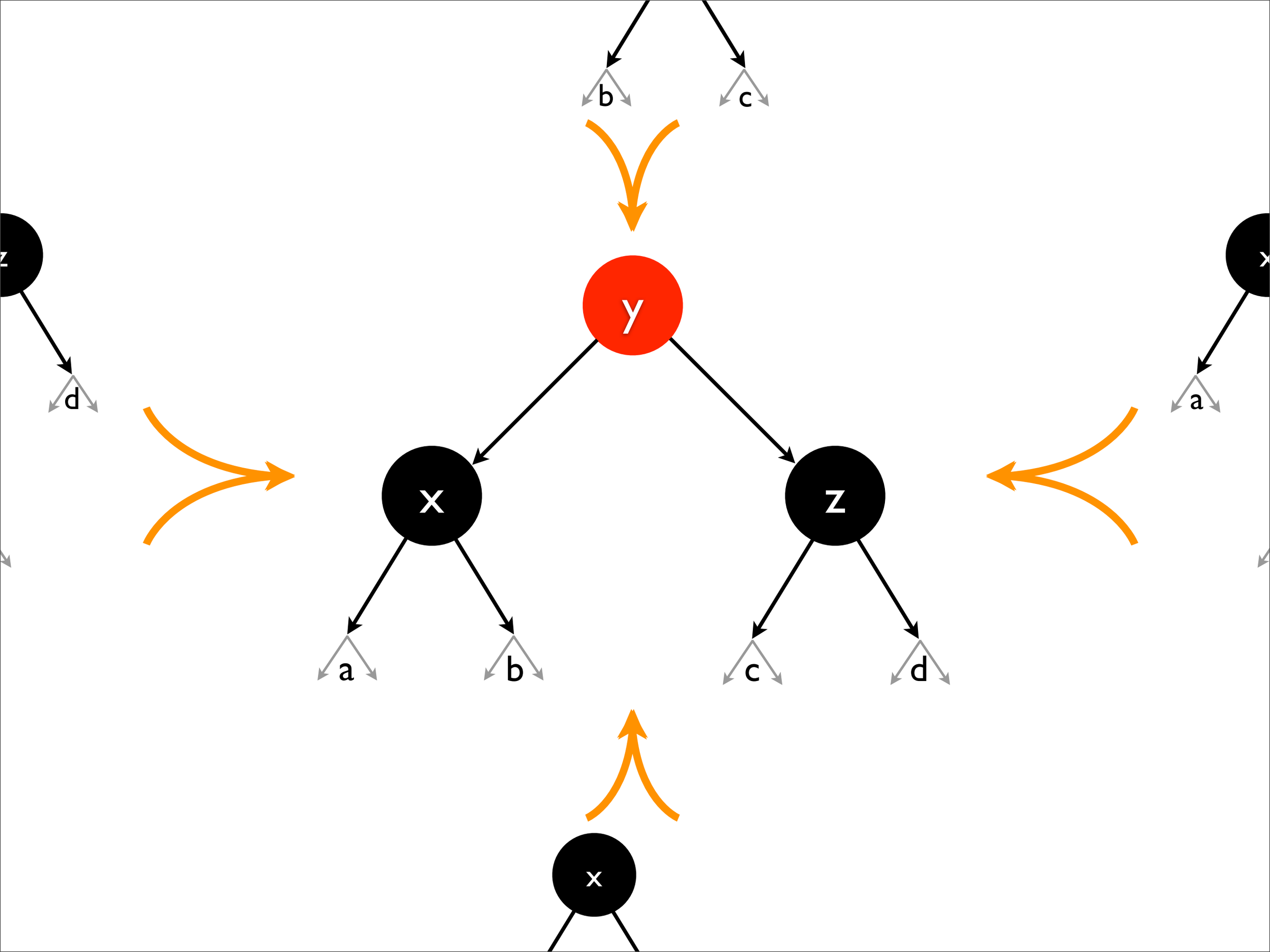
```
case class Node[K : Ordering, +V](isBlack: Boolean,
                                  left: Tree[K, V],
                                  key: K, value: V,
                                  right: Tree[K, V])
    extends Tree[K, V] {
  …
}
```

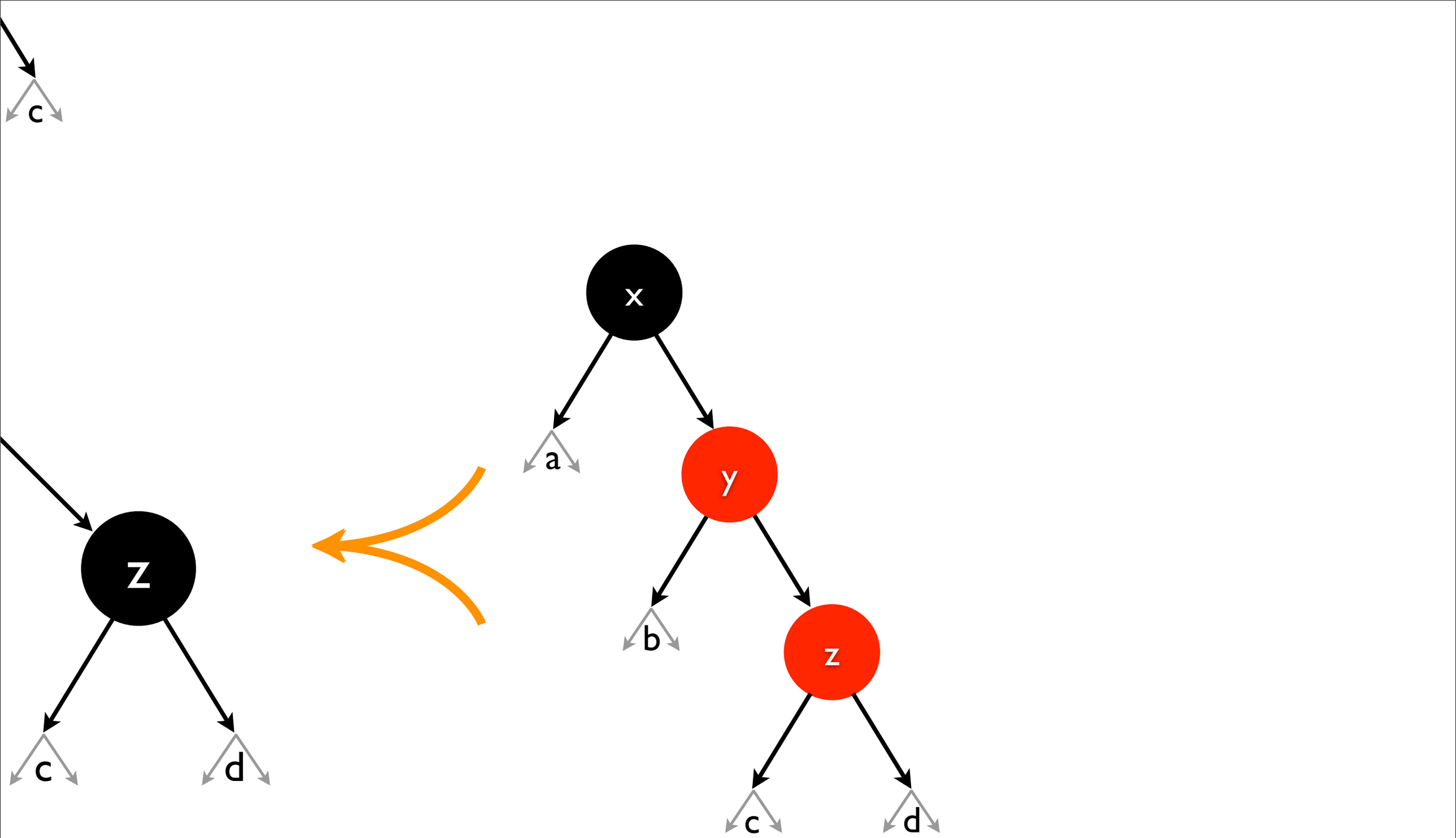
```
case class Leaf[K : Ordering]()
    extends Tree[K, Nothing] {

  val isBlack = true
  …
}
```

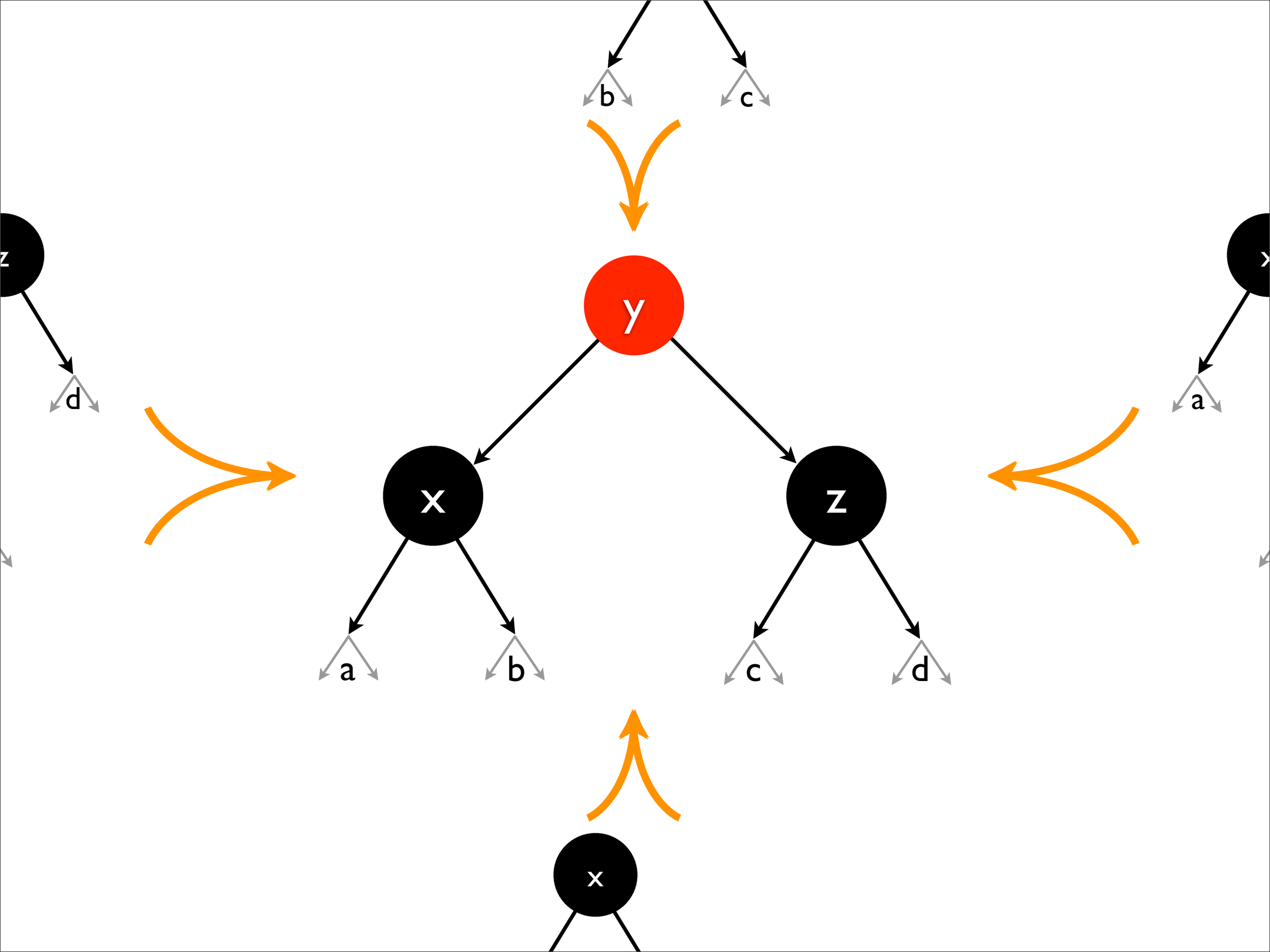

y
x
z
a
b
c
d

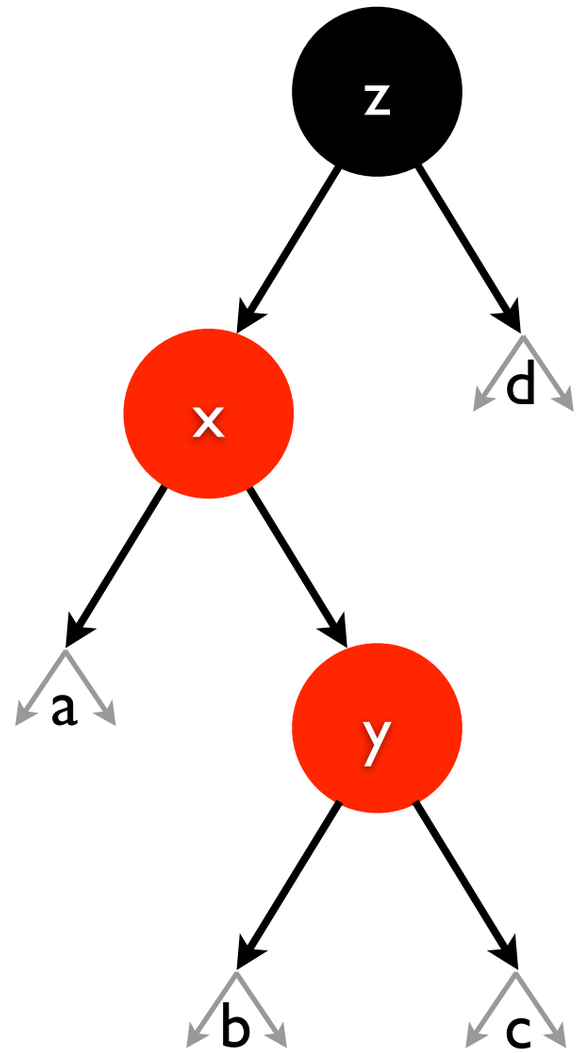

b
c
y
d
a
x
z
a
b
c
d
x


z
y
d
x
c
a
b
x
a
b

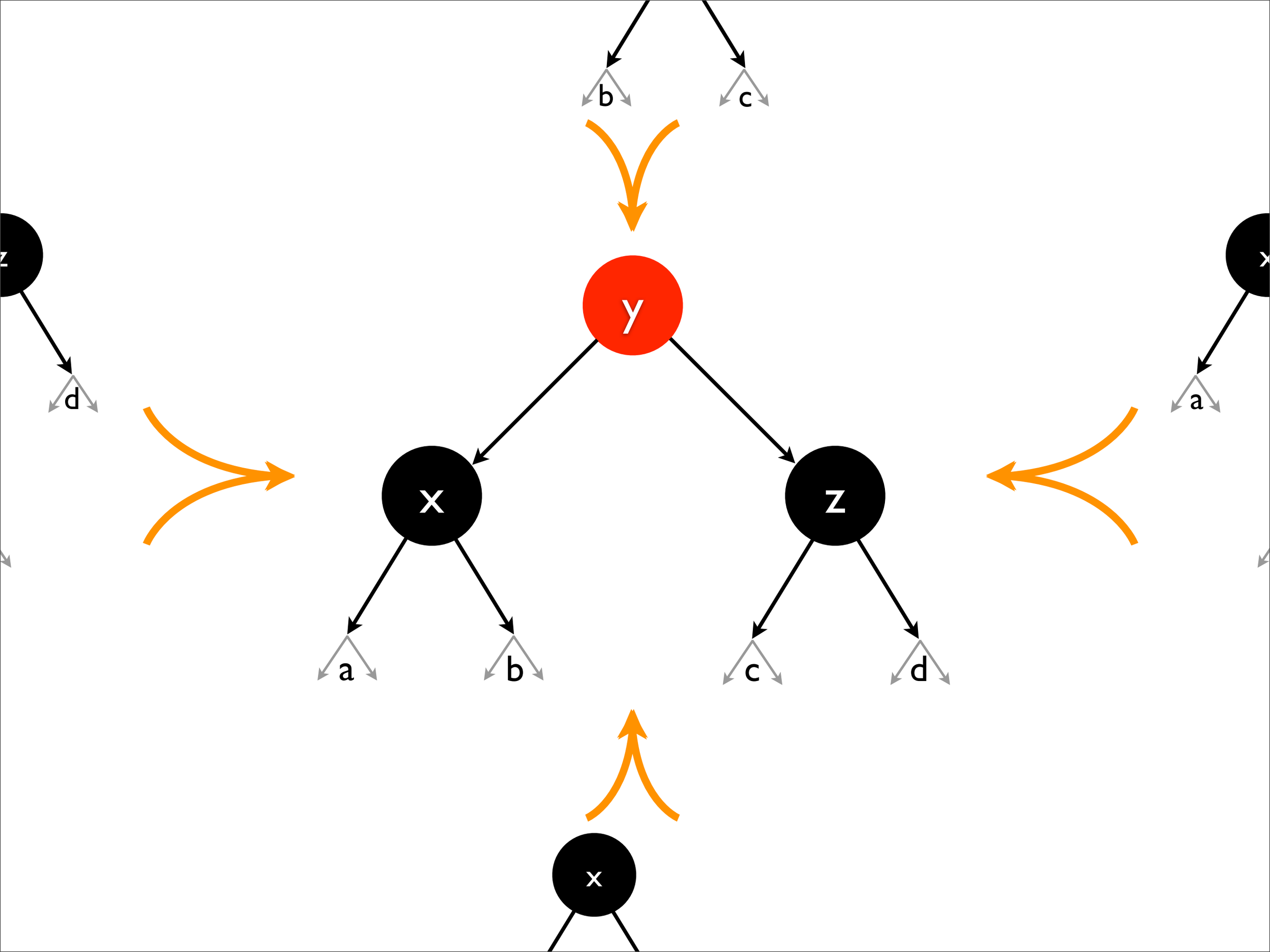

b
c
y
d
a
x
z
a
b
c
d
x

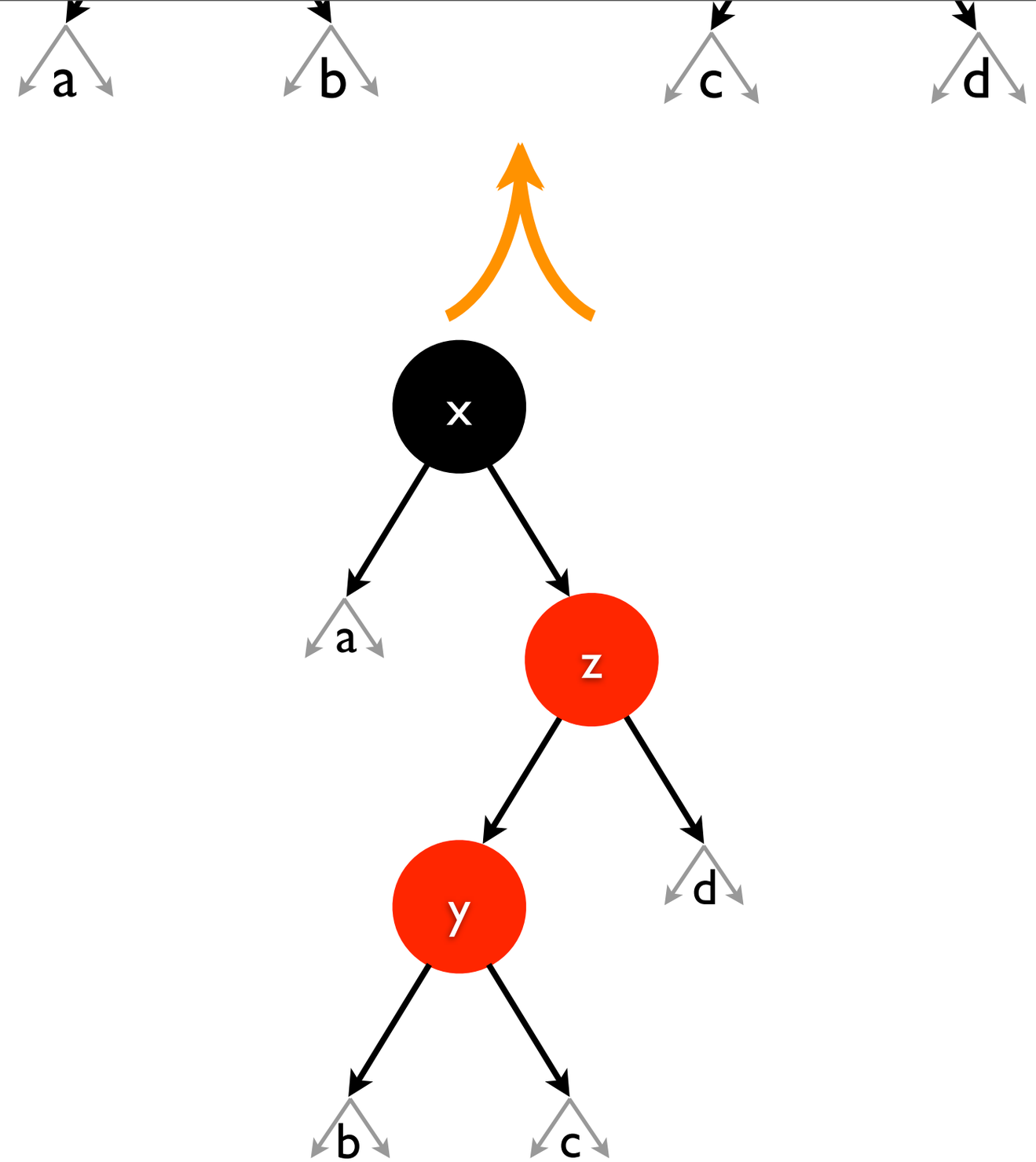

c
x
a
y
z
b
z
c
d
c
d

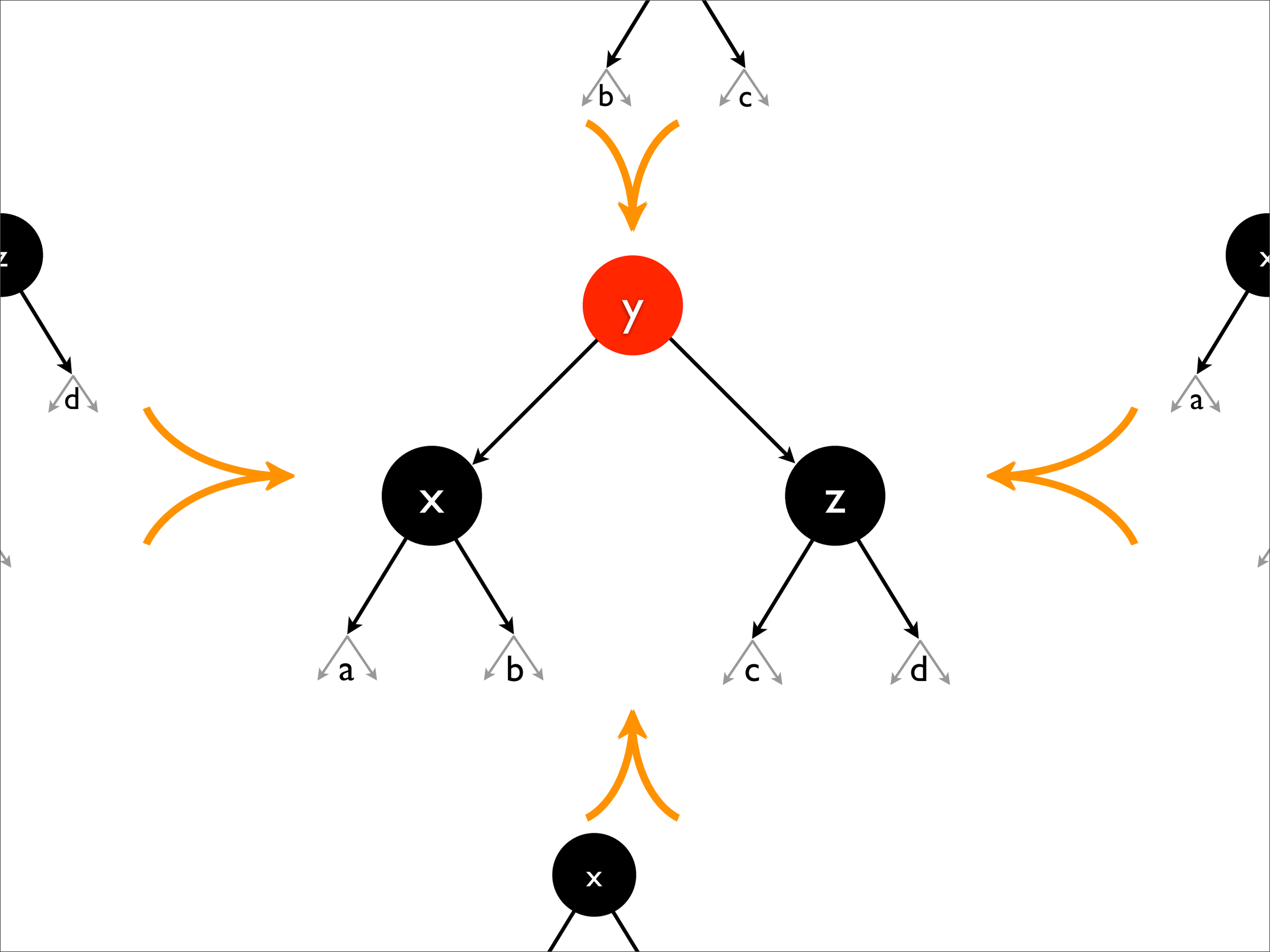

b
c
y
d
a
x
z
a
b
c
d
x


z
x
d
a
y
b
c


y


b
c
y
d
a
x
z
a
b
c
d
x


a
b
c
d
x
a
z
y
d
b
c


b
c
y
d
a
x
z
a
b
c
d
x

```
def balance[K : Ordering, V](isBlack: Boolean, left: Tree[K, V], key: K,
                             value: V, right: Tree[K, V]) = {
  (isBlack, left, key, value, right) match {
    case (true, Node(false, Node(false, a, xk, xv, b), yk, yv, c), zk, zv, d) =>
      Node(false, Node(true, a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

    case (true, Node(false, a, xk, xv, Node(false, b, yk, yv, c)), zk, zv, d) =>
      Node(false, Node(true, a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

    case (true, a, xk, xv, Node(false, Node(false, b, yk, yv, c), zk, zv, d)) =>
      Node(false, Node(true, a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

    case (true, a, xk, xv, Node(false, b, yk, yv, Node(false, c, zk, zv, d))) =>
      Node(false, Node(true, a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

    case (isBlack, a, xk, xv, b) => Node(isBlack, a, xk, xv, b)
  }
}
```

```
isBlack: Boolean, left: Tree[K, V], key: K,
                  value: V, right: Tree[K, V]) = {
 right) match {
ode(false, a, xk, xv, b), yk, yv, c), zk, zv, d) =>
a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

, xk, xv, Node(false, b, yk, yv, c)), zk, zv, d) =>
a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

de(false, Node(false, b, yk, yv, c), zk, zv, d)) =>
a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

de(false, b, yk, yv, Node(false, c, zk, zv, d))) =>
a, xk, xv, b), yk, yv, Node(true, c, zk, zv, d))

 b) => Node(isBlack, a, xk, xv, b)
```

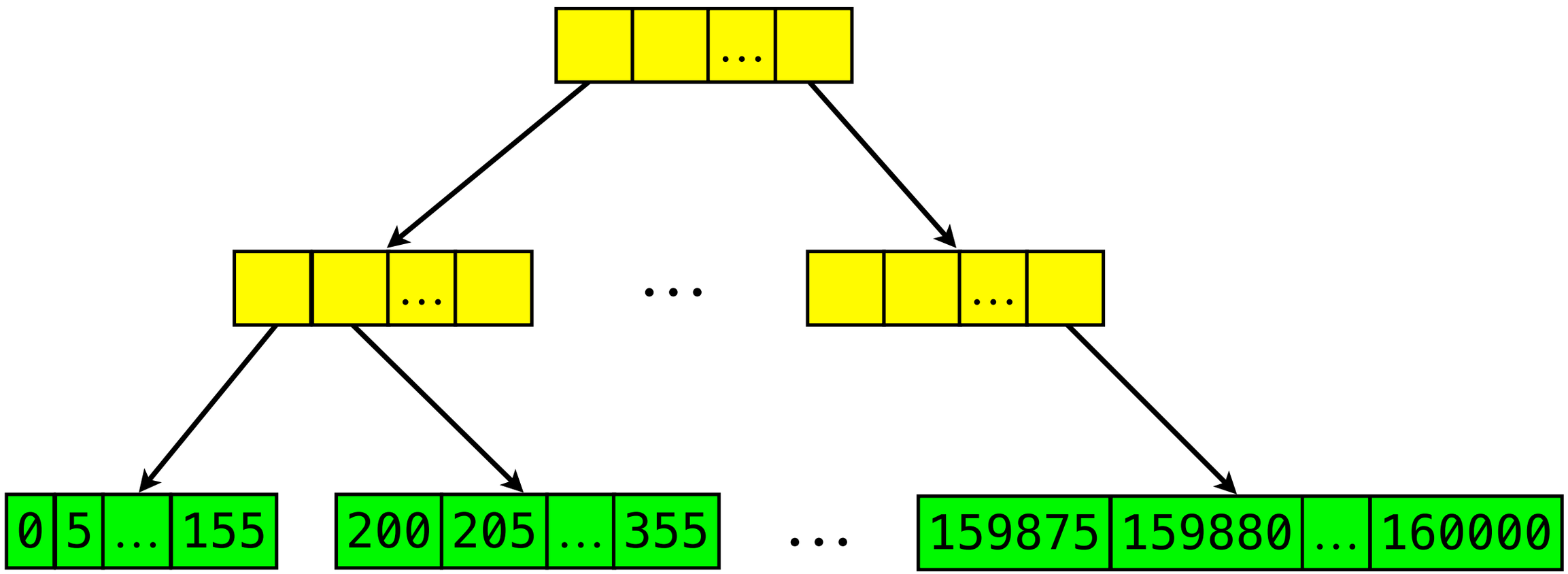
```scala
case class Node[K : Ordering, +V](...) extends … {
  def +[A >: V](pair: (K, A)): Tree[K, V] = {
    val (k2, v2) = pair

    if (key > k2)
      balance(isBlack, left + pair, key, value, right)
    else if (key == k2)
      Node(isBlack, left, k2, v2, right)
    else
      balance(isBlack, left, key, value, right + pair)
  }
}
```

```
case class Node[K : Ordering, +V](...) extends … {
  def +[A >: V](pair: (K, A)): Tree[K, V] = {
    val (k2, v2) = pair

    if (key > k2)
      balance(isBlack, left + pair, key, value, right)
    else if (key == k2)
      Node(isBlack, left, k2, v2, right)
    else
      balance(isBlack, left, key, value, right + pair)
  }
}
```

A little of both…

Bitmapped Vector Trie

```
Vector(0 to 160000 by 5: _*)
```



Complexity

$O(1)$	$O(\log n)$	$O(n)$
`append`		`concat`
`first`		`insert`
`last`		`prepend`
`nth`		
`update`		

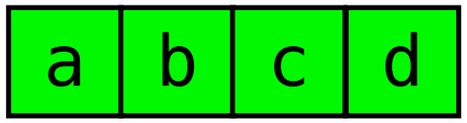
$$O(\log_{32} n)$$

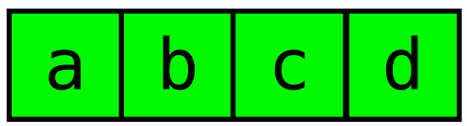
$$O(\log_{32} n) \approx O(1)$$

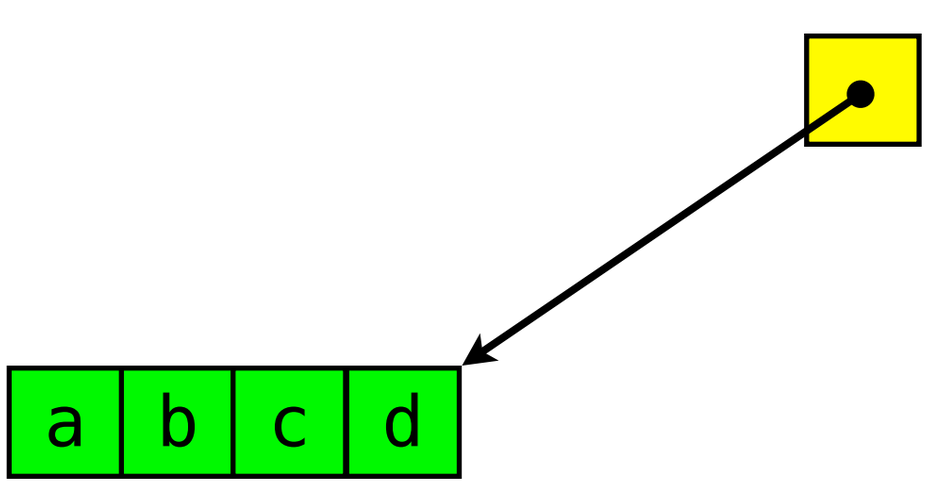
Anatomy

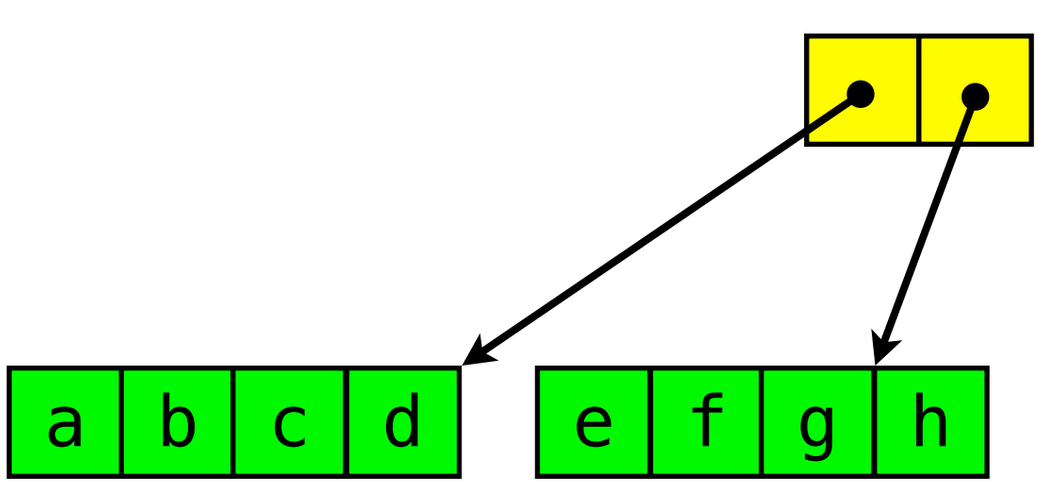
- Start with an array with max length 32
  - Copy on write
- Array of arrays, max length 32
  - Array of array of arrays, max length 32
    - …
- Maximum depth is 7!

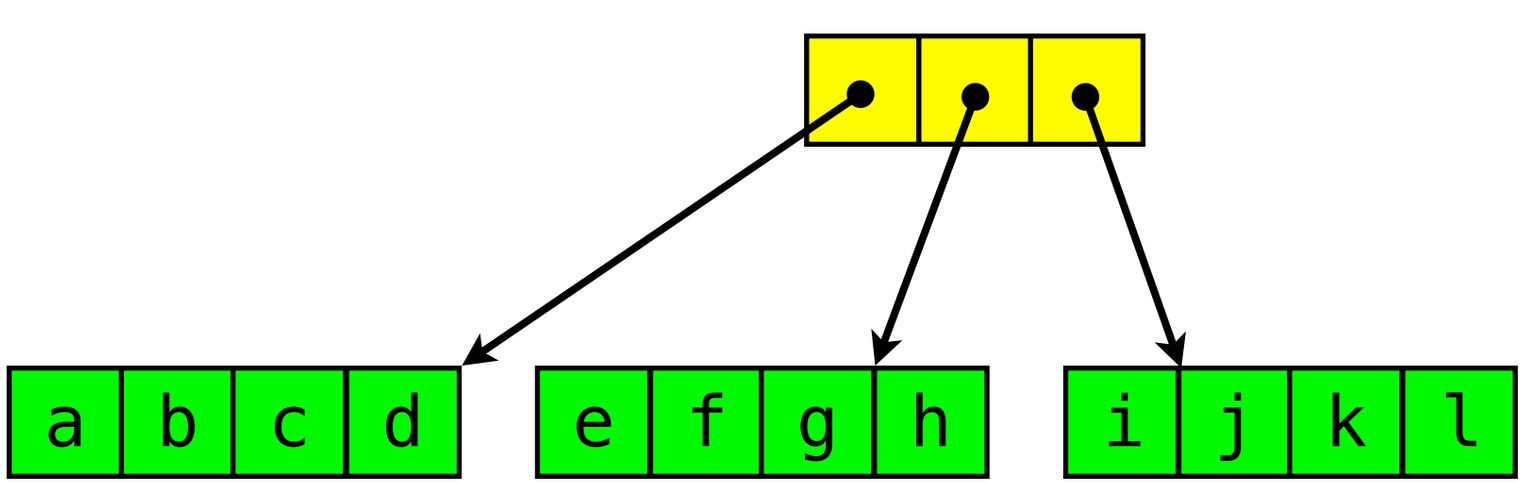
a

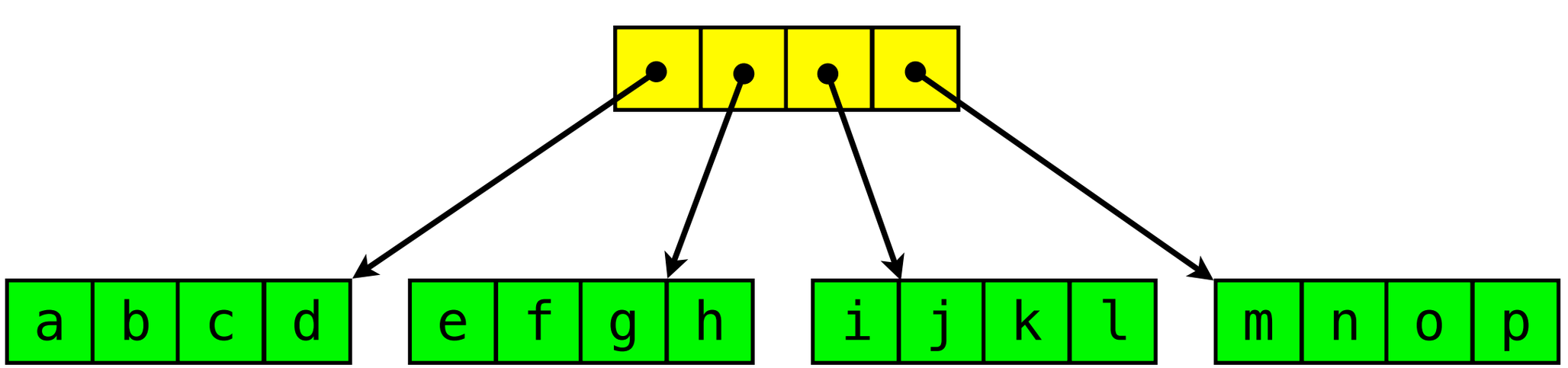

a
b
c
d

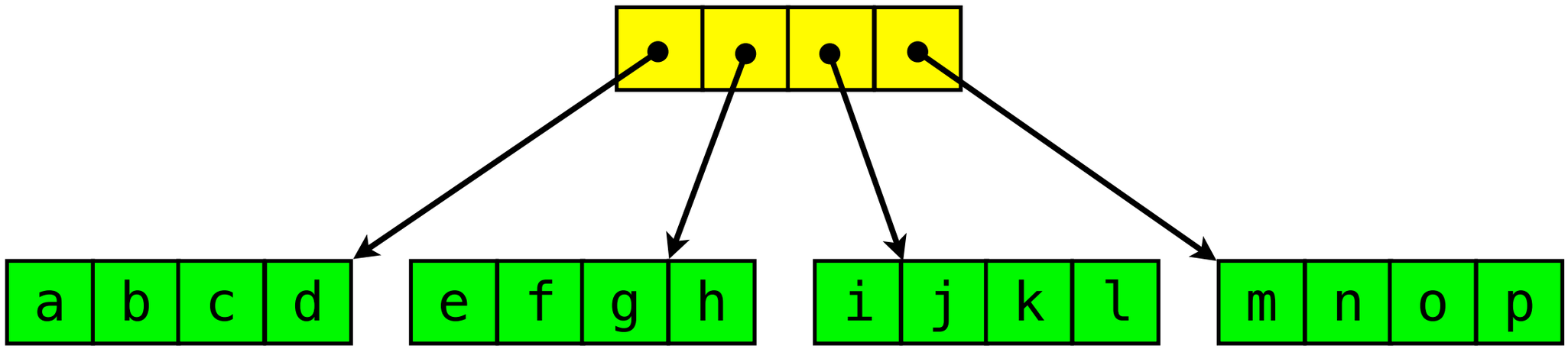

a
b
c
d

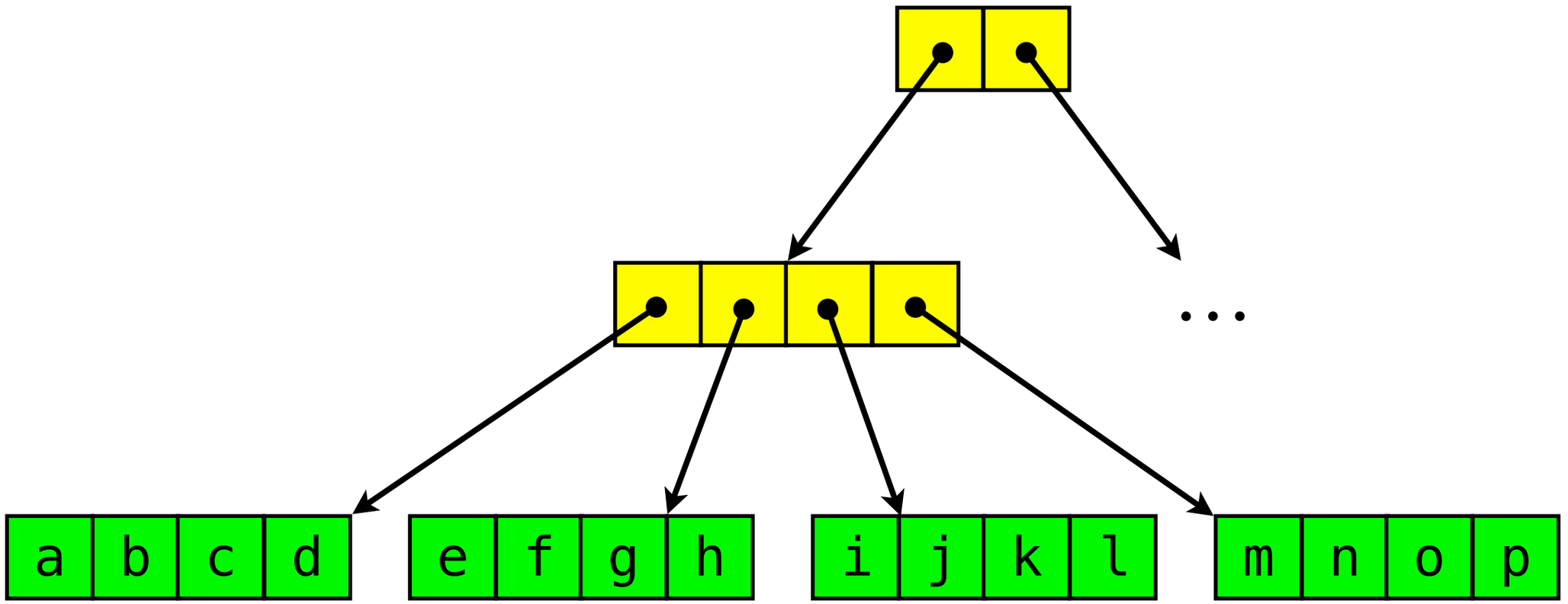

a
b
c
d


a
b
c
d
e
f
g
h


a
b
c
d
e
f
g
h
i
j
k
l


a b c d
e f g h
i j k l
m n o p

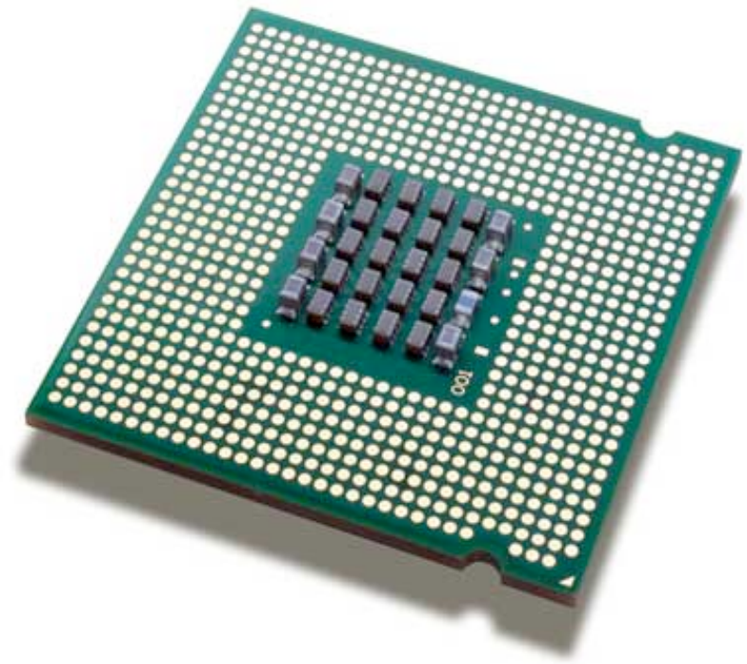

a b c d
e f g h
i j k l
m n o p


a b c d
e f g h
i j k l
m n o p
...

```
sealed trait Case {
  type Self <: Case

  val shift: Int

  def apply(i: Int): Array[AnyRef]
  def updated(i: Int, obj: AnyRef): Self

  def :+(node: Array[AnyRef]): Case
  def pop: (Case, Array[AnyRef])
}
```

```
case class One(trie: Array[AnyRef]) extends Case {
  type Self = One

  val shift = 0

  def apply(i: Int) = trie

  def updated(i: Int, obj: AnyRef) = {
    val trie2 = copy1(trie, new Array(trie.length))
    trie2(i & 0x01f) = obj
    One(trie2)
  }

  def :+(tail: Array[AnyRef]) = Two(Array(trie, tail))

  def pop = (Zero, trie)
}
```

Modern Architectures



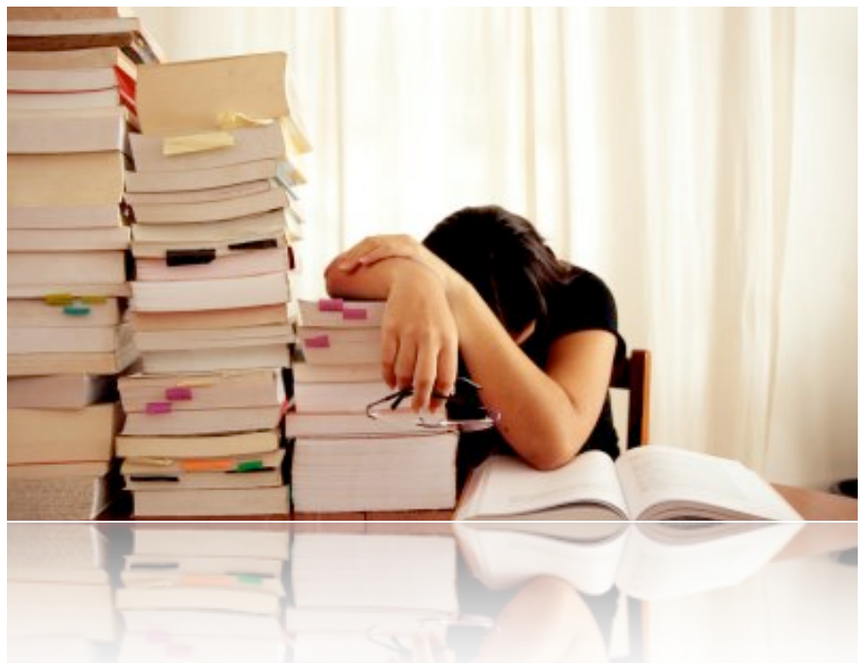
- Locality of reference
- Caching
  - Bite-sized data chunks
- JVM considerations
  - Heap locality

References

[1] Okasaki; *Purely Functional Data Structures*

[2] Okasaki; *Red-Black Trees in a Functional Setting*

[3] Hinz & Paterson; *Finger trees, a simple general-purpose data structure*



http://github.com/djspiewak/extreme-cleverness