Random Access Zipper

RAZ

Presented by Kyle Headley

PLDI’16 SRC

Persistent data structures offer various trade-offs programmers

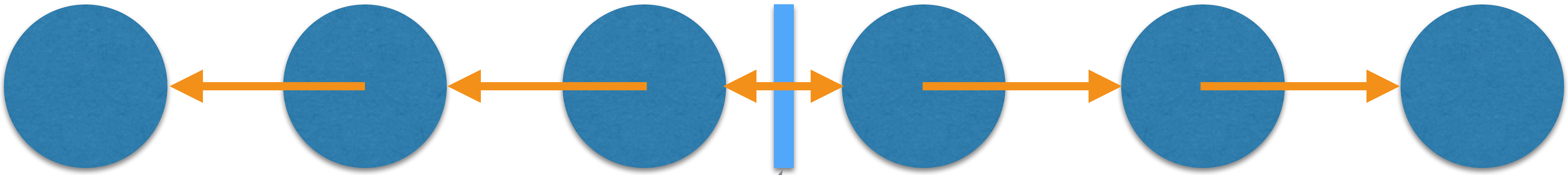
What do we have for sequences?

Zippers are great

Zippers are great

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list

type ‘a zip =
  ‘a list * ‘a list
```



zip is a Cursor

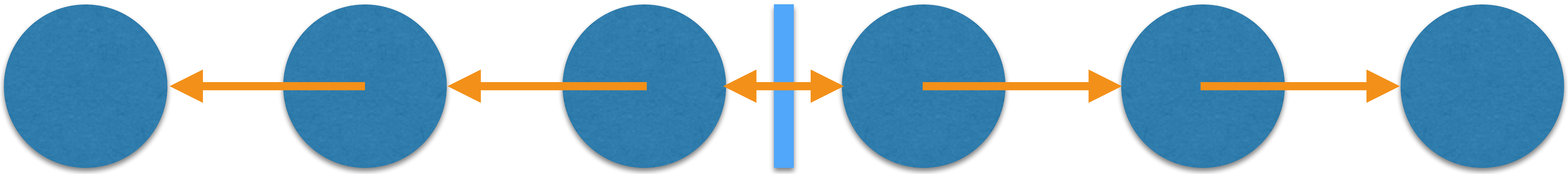
Zippers are great

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list

type ‘a zip =
  ‘a list * ‘a list
```

```
move:    dir ->
  ‘a zip -> ‘a zip
insert:  dir -> ‘a ->
  ‘a zip -> ‘a zip
remove:  dir ->
  ‘a zip -> ‘a zip
```

All O(1)!

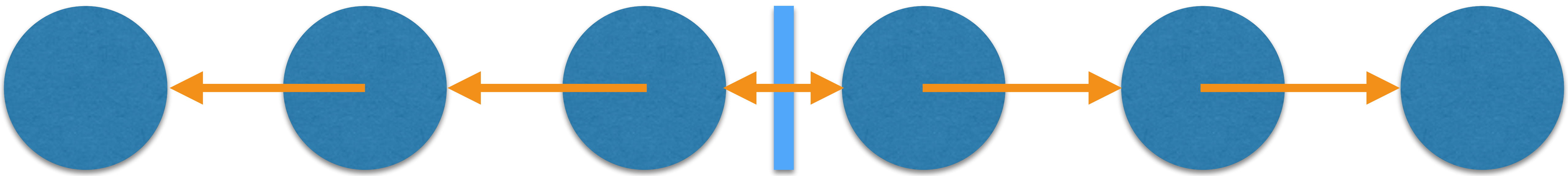


Zippers are great

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list

type ‘a zip =
  ‘a list * ‘a list
```

```
move:      dir ->
   ‘a zip -> ‘a zip
insert:    dir -> ‘a ->
   ‘a zip -> ‘a zip
remove:    dir ->
   ‘a zip -> ‘a zip
```



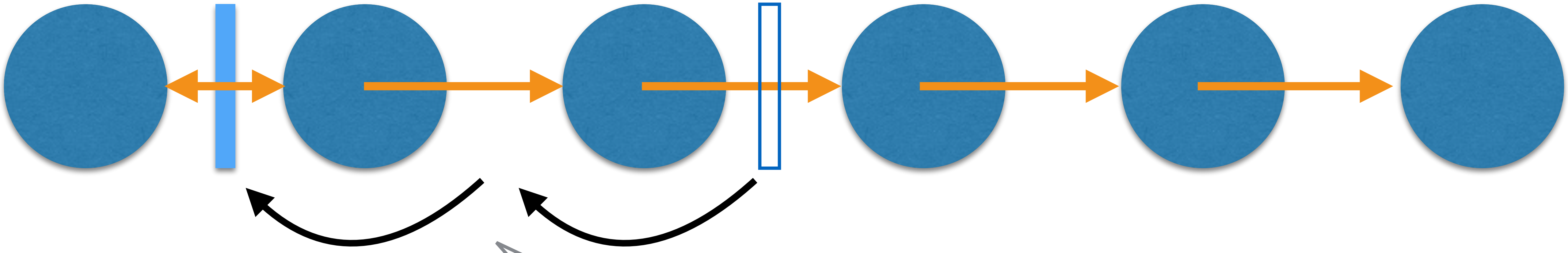
Problem: Slow random access

Zippers are great

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list

type ‘a zip =
  ‘a list * ‘a list
```

```
move:     dir ->
   ‘a zip -> ‘a zip
insert:   dir -> ‘a ->
   ‘a zip -> ‘a zip
remove:   dir ->
   ‘a zip -> ‘a zip
```



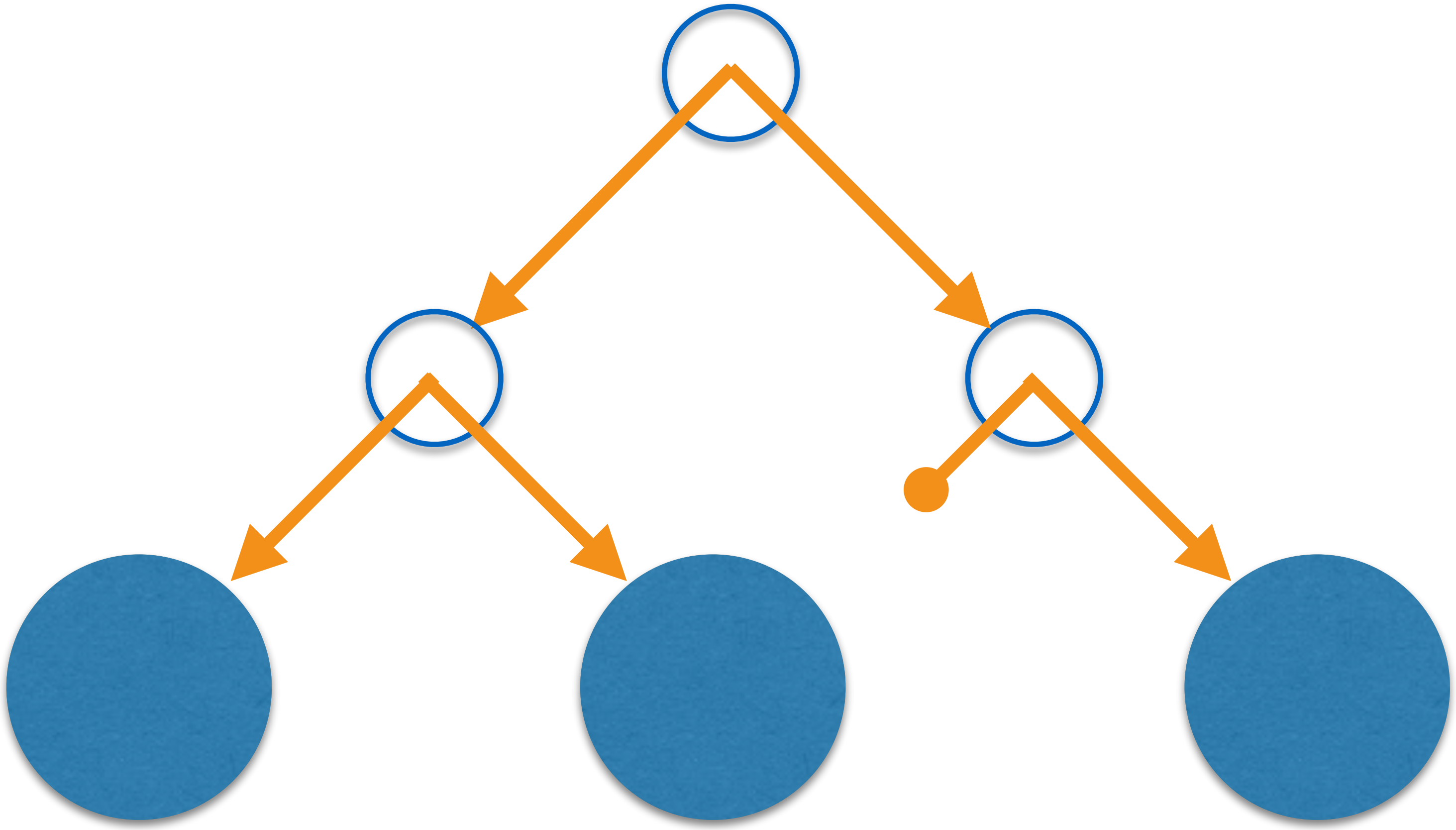
Requires n moves

Problem: Slow random access

Trees are great

Trees are great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of ‘a tree * ‘a tree
```

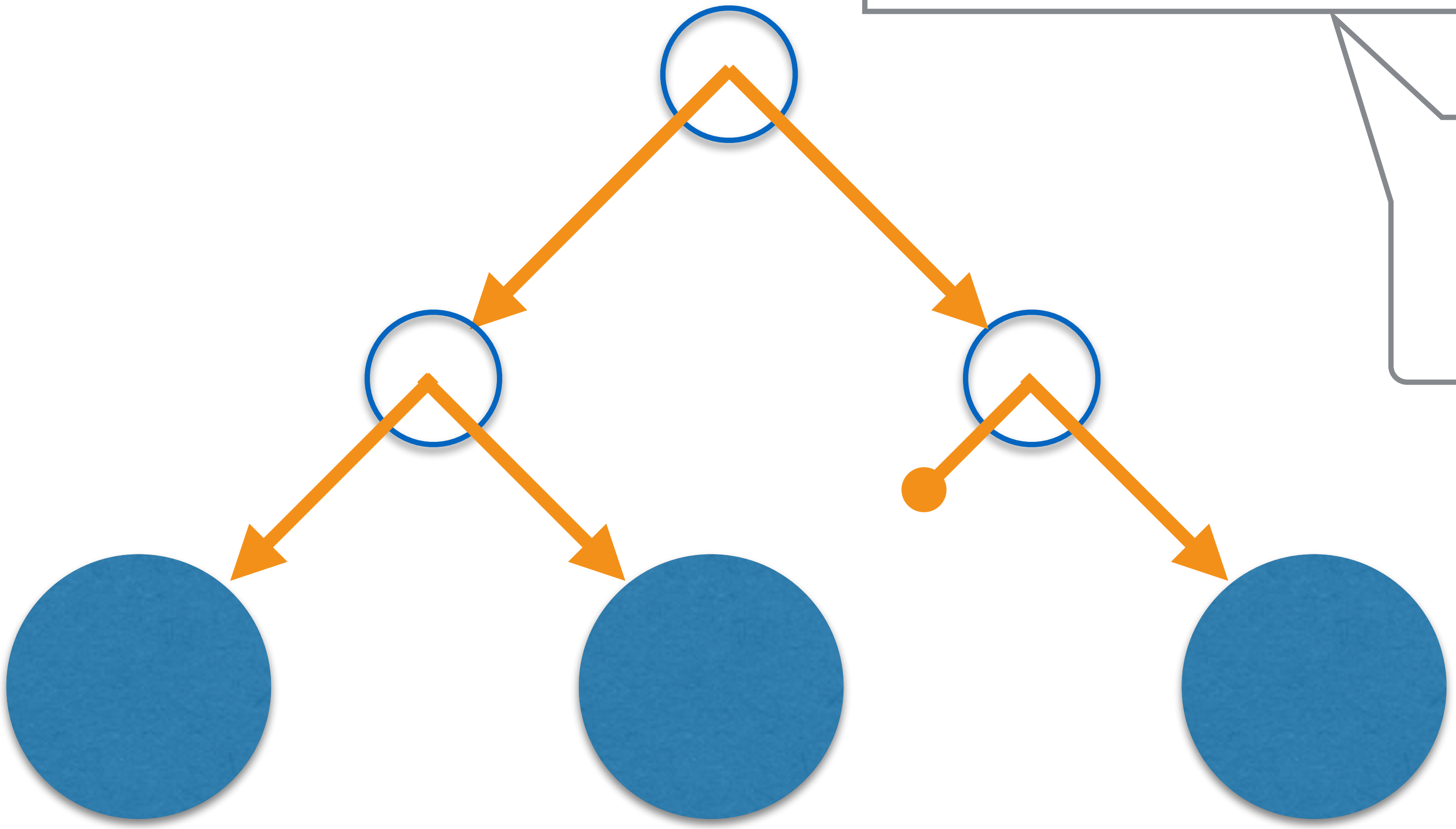


Trees are great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of ‘a tree * ‘a tree
```

```
insert: pos -> ‘a -> ‘a tree ->
‘a tree
find:   pos -> ‘a tree -> ‘a
remove: pos ->
   ‘a tree -> ‘a tree
```

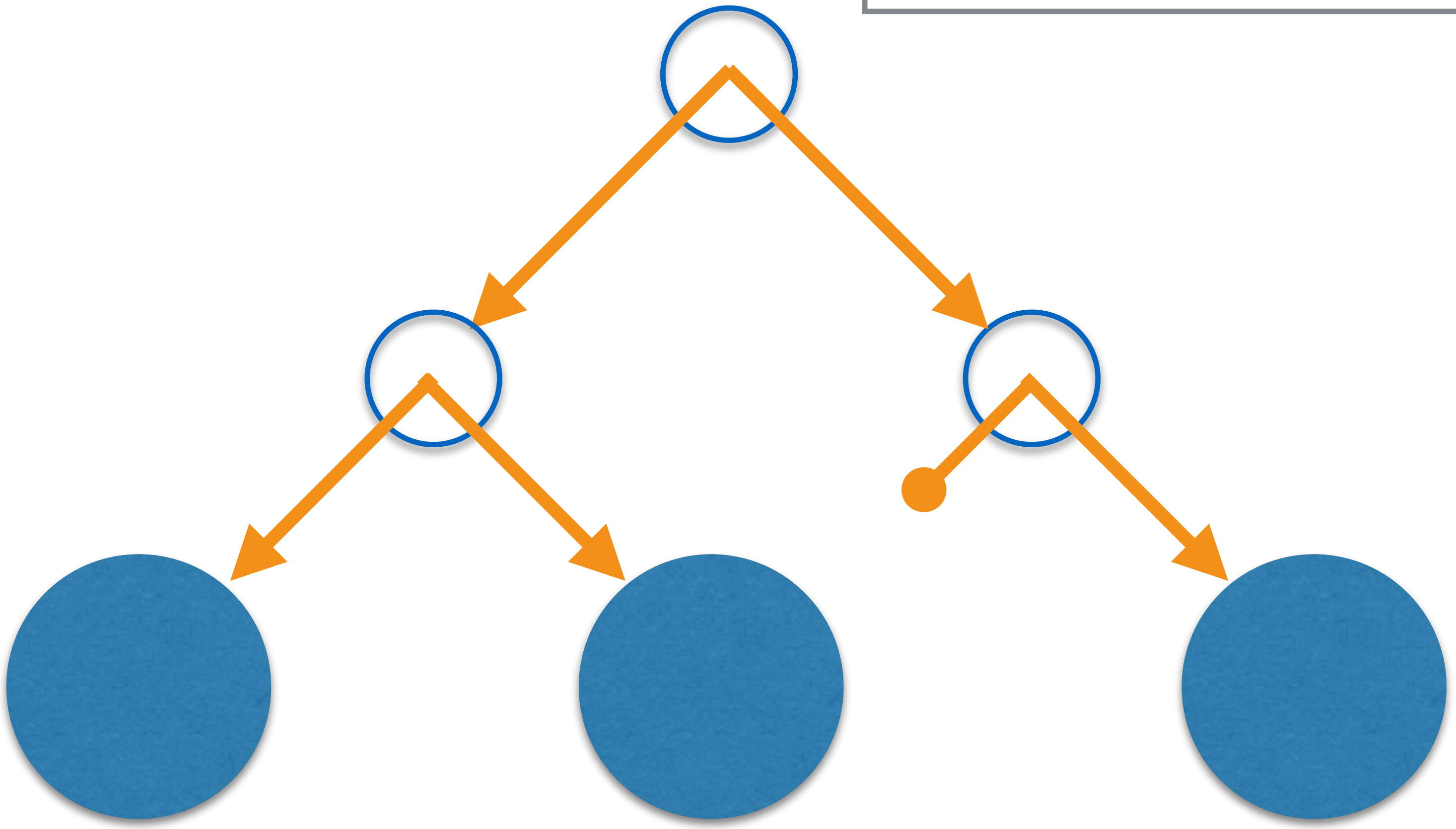
All O(log n)!
(w/meta data)



Trees are great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of ‘a tree * ‘a tree
```

```
insert: pos -> ‘a -> ‘a tree ->
‘a tree
find:   pos -> ‘a tree -> ‘a
remove: pos ->
    ‘a tree -> ‘a tree
```

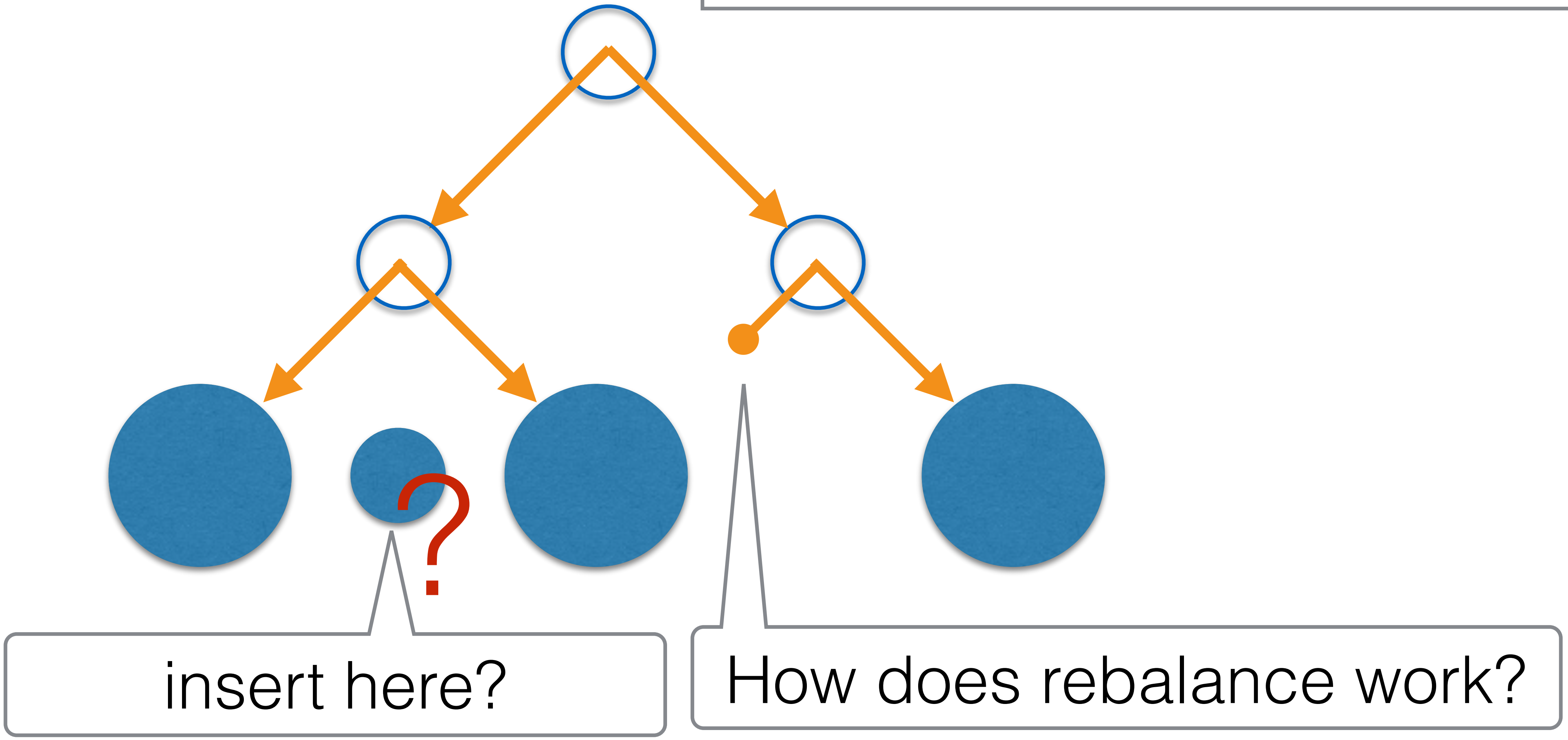


Problem: Reasoning about edits

Trees are great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of ‘a tree * ‘a tree
```

```
insert: pos -> ‘a -> ‘a tree ->
‘a tree
find:   pos -> ‘a tree -> ‘a
remove: pos ->
   ‘a tree -> ‘a tree
```



insert here?

How does rebalance work?

Problem: Reasoning about edits

Fingertrees are great

Fingertrees are great

```
first:  ‘a finger -> ‘a
last:   ‘a finger -> ‘a
cons:   ‘a ->
   ‘a finger -> ‘a finger
snoc:   ‘a ->
   ‘a finger -> ‘a finger
```

All O(1)!
(amortized)

Fingertrees are great

```
first:  ‘a finger -> ‘a
last:   ‘a finger -> ‘a
cons:   ‘a ->
   ‘a finger -> ‘a finger
snoc:   ‘a ->
   ‘a finger -> ‘a finger
```

```
split:  pos -> ‘a finger ->
  (‘a finger, ‘a finger)
append: ‘a finger -> ‘a finger
-> ‘a finger -> ‘a finger
```

Both O(log n)!

Fingertrees are great

```
first:  ‘a finger -> ‘a
last:   ‘a finger -> ‘a
cons:   ‘a ->
   ‘a finger -> ‘a finger
snoc:   ‘a ->
   ‘a finger -> ‘a finger
```

```
split:  pos -> ‘a finger ->
  (‘a finger, ‘a finger)
append: ‘a finger -> ‘a finger
-> ‘a finger -> ‘a finger
```

Problem: Not so simple

Fingertrees are great

```
type ‘a node =
| Node2 of ‘a * `a
| Node3 of ‘a * `a * `a
type ‘a digit =
| One of ‘a
| Two of ‘a * ‘a
| Three of ‘a * ‘a * ‘a
| Four of ‘a * ‘a * ‘a * ‘a
type ‘a finger =
| Nil
| Single of ‘a
| Deep of
   ‘a digit
 * (‘a node) finger
 * ’a digit
```

Nested type

```
first:  ‘a finger -> ‘a
last:   ‘a finger -> ‘a
cons:   ‘a ->
   ‘a finger -> ‘a finger
snoc:   ‘a ->
   ‘a finger -> ‘a finger
```

```
split:  pos -> ‘a finger ->
  (‘a finger, ‘a finger)
append: ‘a finger -> ‘a finger
-> ‘a finger -> ‘a finger
```

Problem: Not so simple

Alternative:
Random Access Zipper

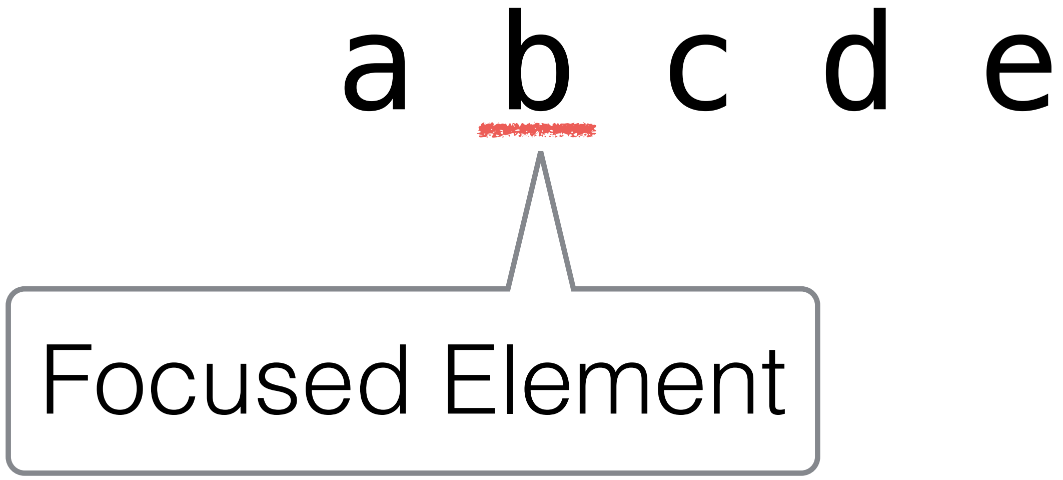
- Accessible
- Editable
- Simple

Using a RAZ

`raz` `a b c d e`

Using a RAZ

`raz`



a b c d e

Focused Element

Using a RAZ

```
raz                   a b c d e
|> insert left n      a n b c d e
```

Using a RAZ

```
raz                        a b c d e
|> insert left n           a n b c d e
|> remove left             a b c d e
```

Using a RAZ

```
raz                    a b c d e
|> insert left n       a n b c d e
|> remove left         a b c d e
|> remove right        a b d e
```

Using a RAZ

```
raz                      a b c d e
|> insert left n         a n b c d e
|> remove left           a b c d e
|> remove right          a b d e
|> unfocus               a b d e
```

Using a RAZ

```
raz                      a b c d e
|> insert left n         a n b c d e
|> remove left           a b c d e
|> remove right          a b d e
|> unfocus               a b d e
|> focus 0               a b d e
```

Refocus for random access

Using a RAZ

```
raz                   a b c d e
|> insert left n      a n b c d e
|> remove left        a b c d e
|> remove right       a b d e
|> unfocus            a b d e
|> focus 0            a b d e
|> alter right n      a n d e
```

The RAZ is great

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

A Tree

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
| Level of lev * ‘a list
| Tree of ‘a tree * ‘a list
```

In a list

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
| Level of lev * ‘a list
| Tree of ‘a tree * ‘a list
```

```
type ‘a raz =
   ‘a list * ‘a * ‘a list
```

As a zipper

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

Still get tree info

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
| Level of lev * ‘a list
| Tree of ‘a tree * ‘a list
```

```
type ‘a raz =
   ‘a list * ‘a * ‘a list
```

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
| Level of lev * ‘a list
| Tree of ‘a tree * ‘a list
```

```
type ‘a raz =
   ‘a list * ‘a * ‘a list
```

```
move:    dir ->
  ‘a zip -> ‘a zip
insert:  dir -> ‘a ->
  ‘a zip -> ‘a zip
remove:  dir ->
  ‘a zip -> ‘a zip
```

All O(1)!

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
| Level of lev * ‘a list
| Tree of ‘a tree * ‘a list
```

```
type ‘a raz =
   ‘a list * ‘a * ‘a list
```

```
move:    dir ->
  ‘a zip -> ‘a zip
insert:  dir -> ‘a ->
  ‘a zip -> ‘a zip
remove:  dir ->
  ‘a zip -> ‘a zip
```

```
focus:   val ->
  ‘a tree -> ‘a raz
unfocus: ‘a raz -> ‘a tree
```

Both O(log n)!
(plus net insertions)

The RAZ is great

```
type ‘a tree =
| Nil
| Leaf of ‘a
| Bin of lev * item_c
   * ’a tree * ‘a tree
```

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
| Level of lev * ‘a list
| Tree of ‘a tree * ‘a list
```

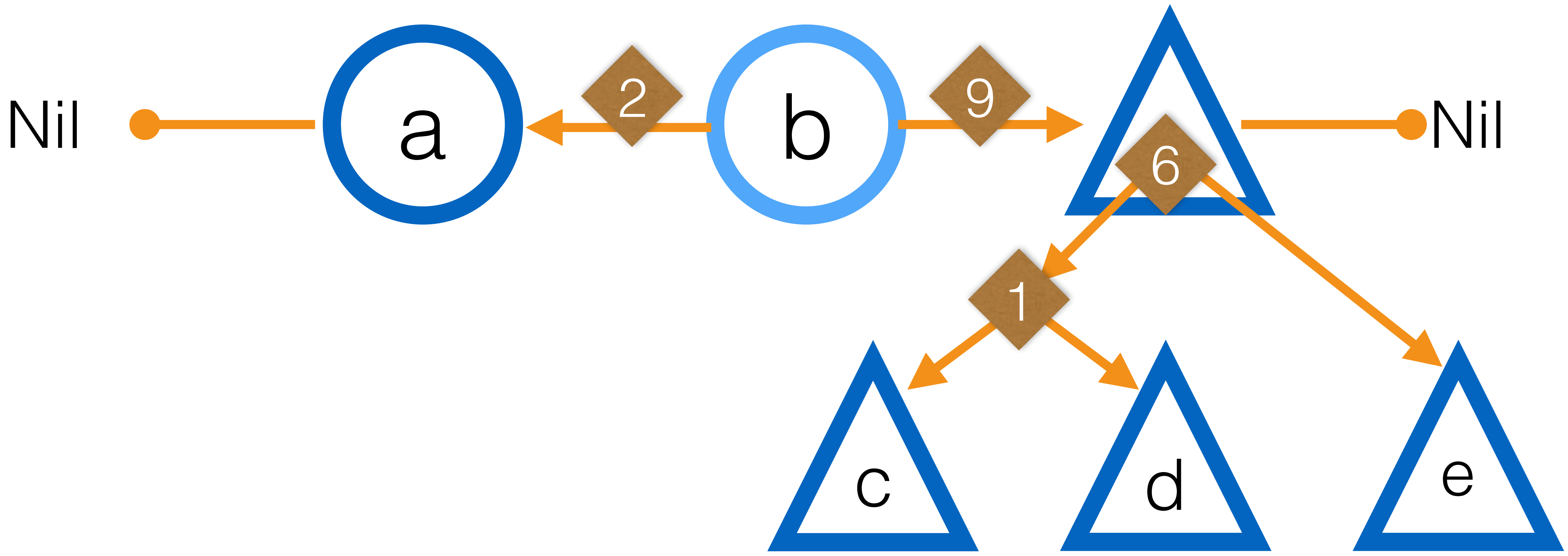
```
type ‘a raz =
   ‘a list * ‘a * ‘a list
```

```
move:    dir ->
  ‘a zip -> ‘a zip
insert:  dir -> ‘a ->
  ‘a zip -> ‘a zip
remove:  dir ->
  ‘a zip -> ‘a zip
```

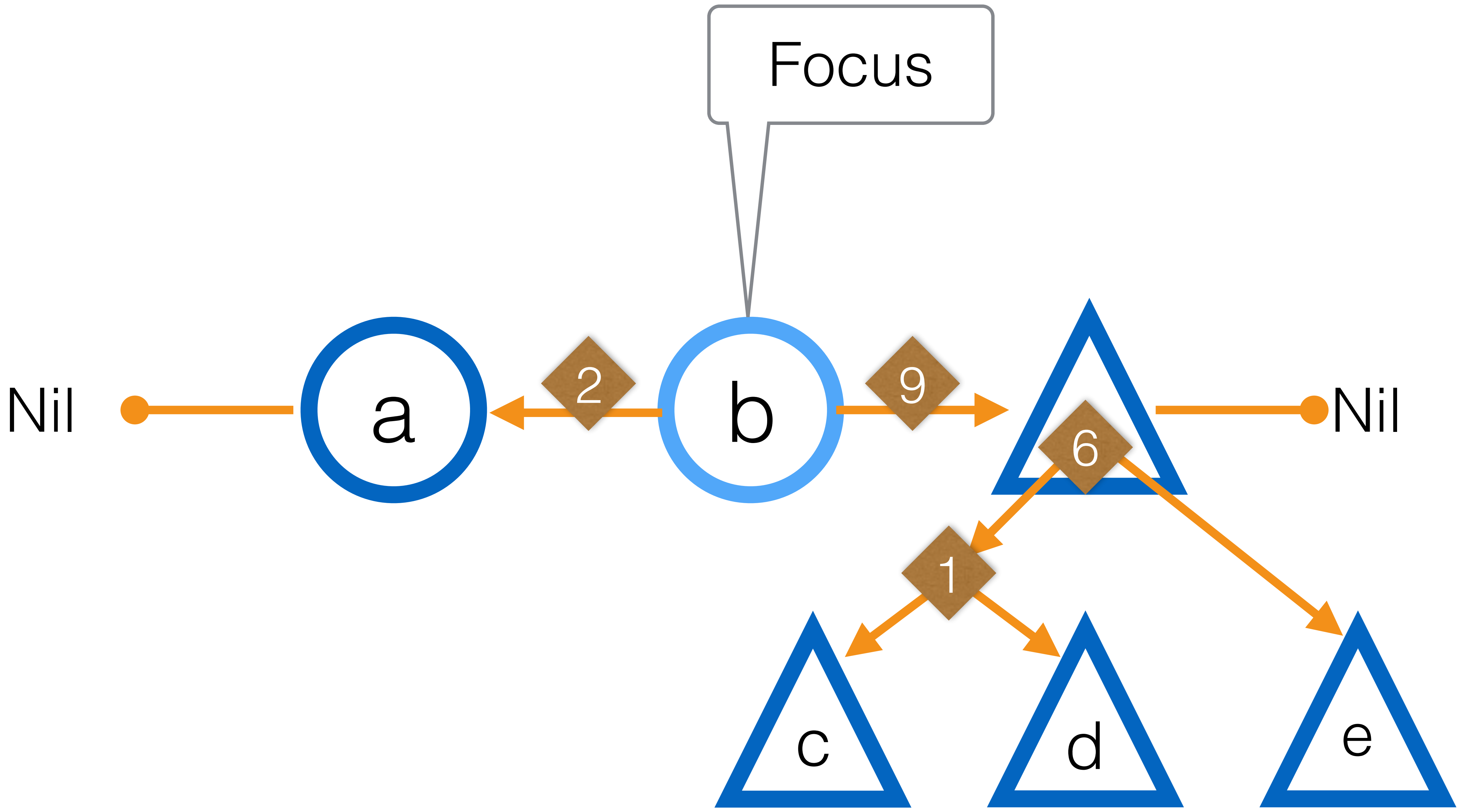
```
focus:   val ->
  ‘a tree -> ‘a raz
unfocus: ‘a raz -> ‘a tree
```

Simple: < 200 lines of code

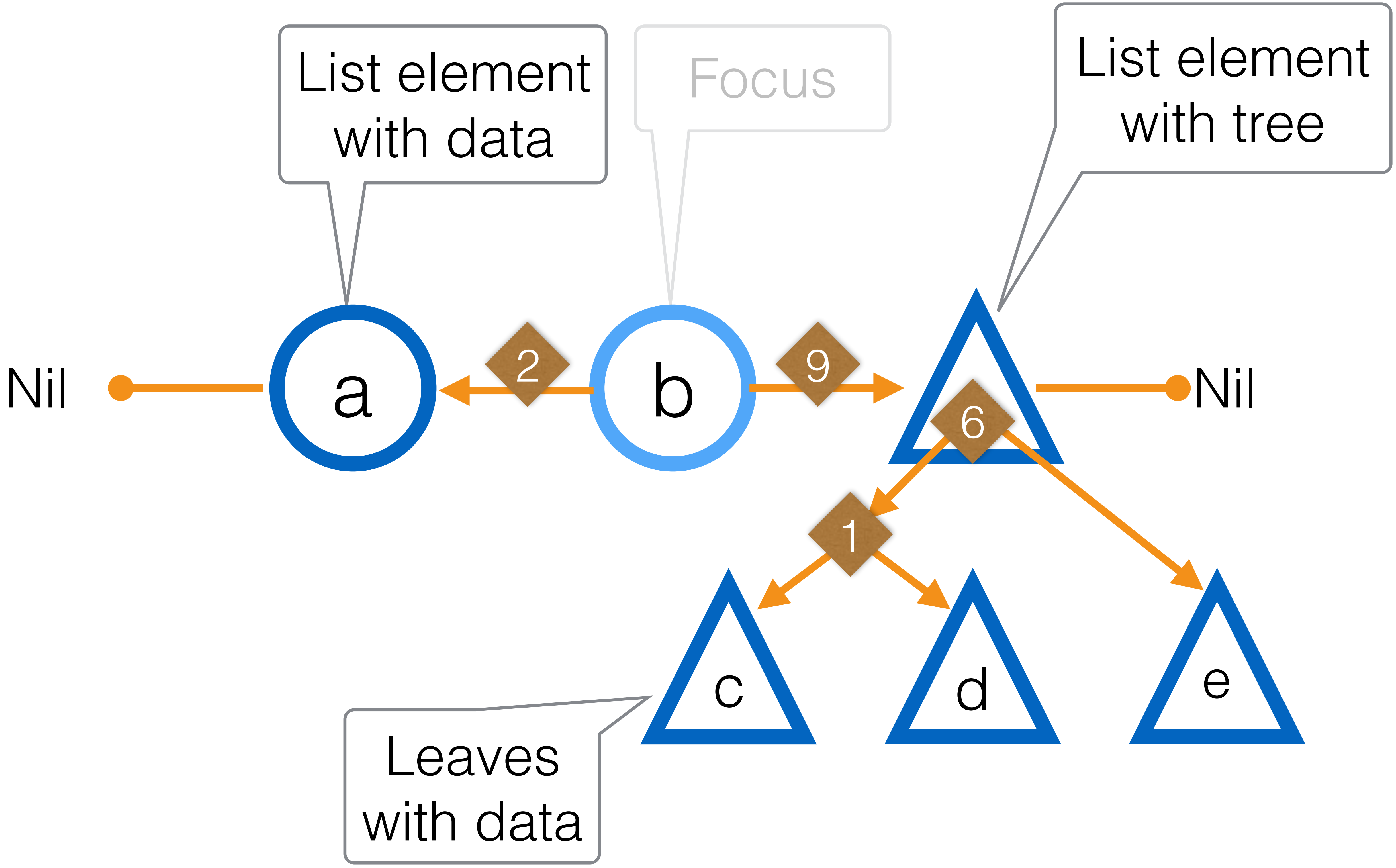
Zipper of Trees



Zipper of Trees



Zipper of Trees



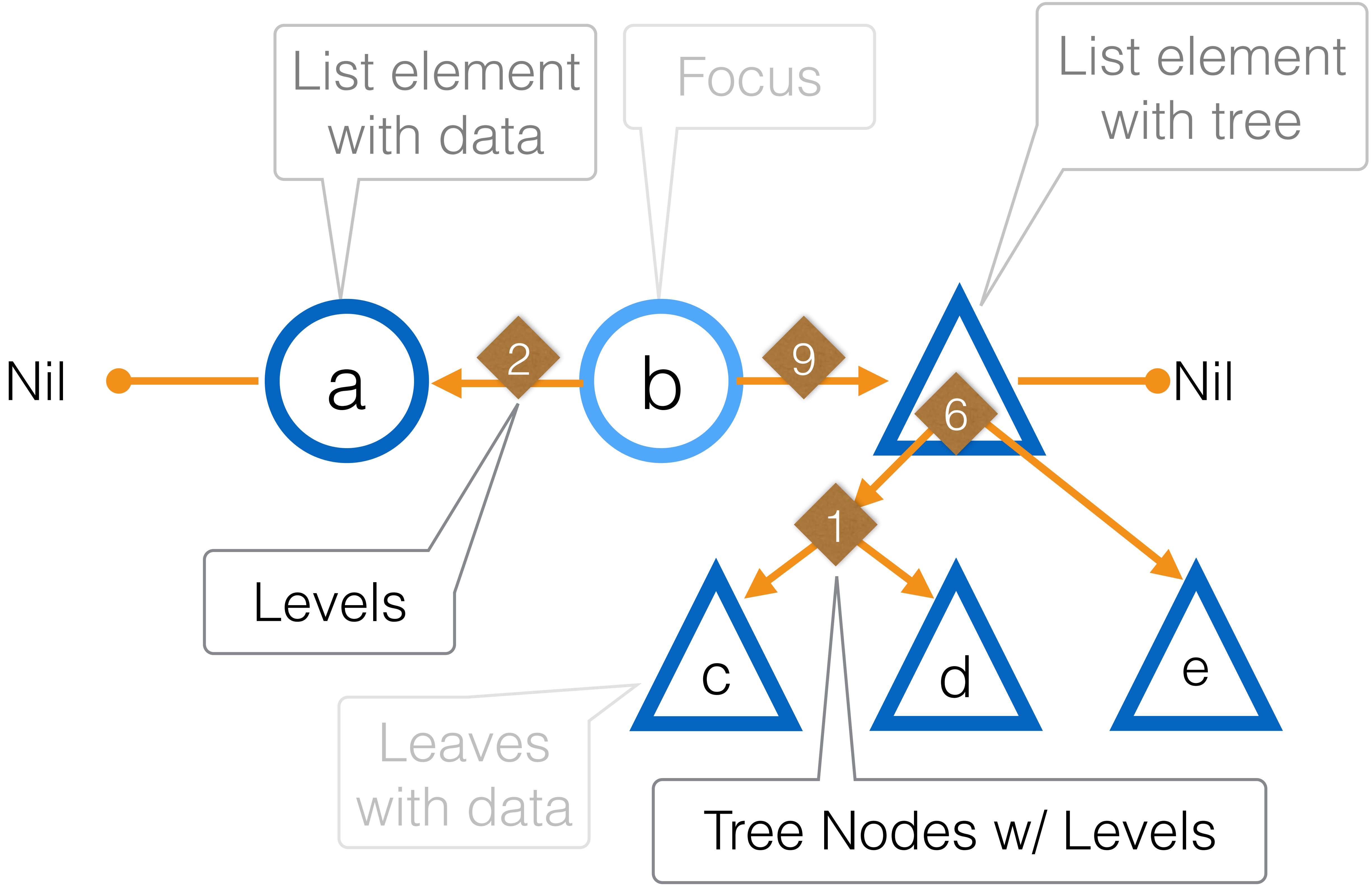
List element with data

Focus

List element with tree

Leaves with data

Zipper of Trees



List element with data

Focus

List element with tree

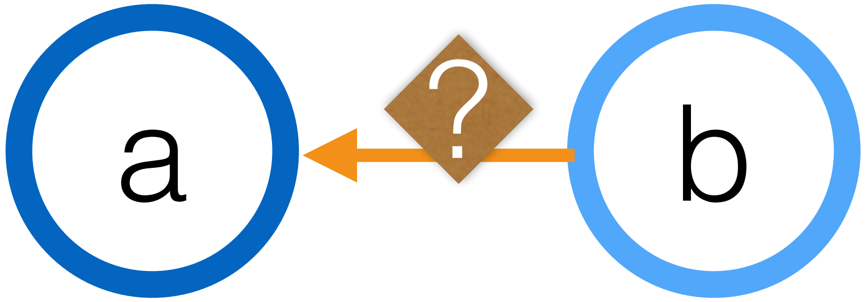
Levels

Leaves with data

Tree Nodes w/ Levels

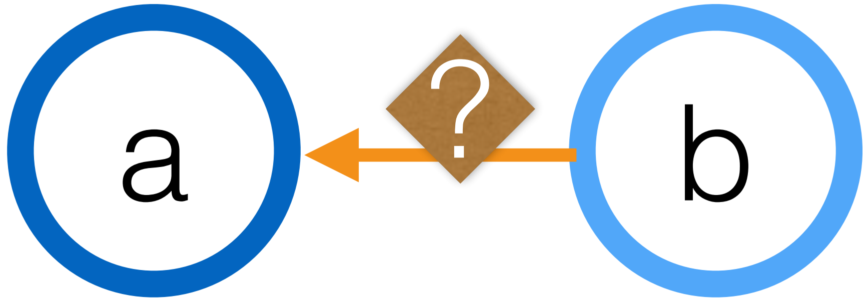
Levels for Balance

Balance



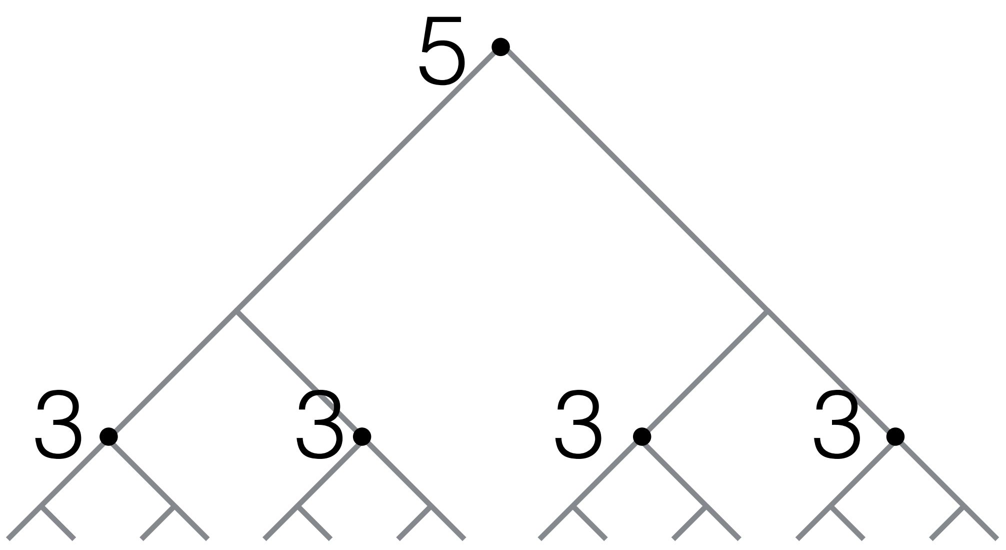
Levels provide a way to maintain the balance of the tree elements of a RAZ

Balance

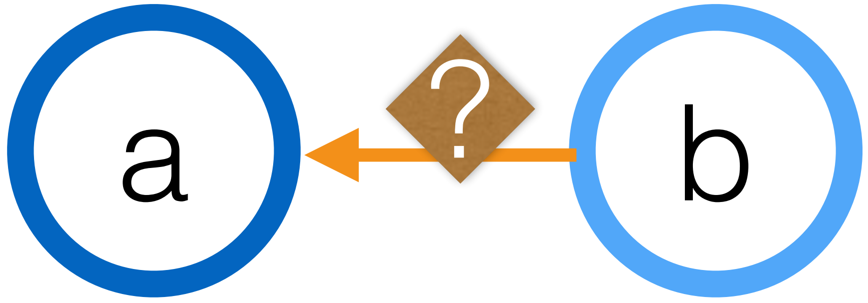


Levels provide a way to maintain the balance of the tree elements of a RAZ

Random levels drawn from distribution of levels in a (huge) binary tree

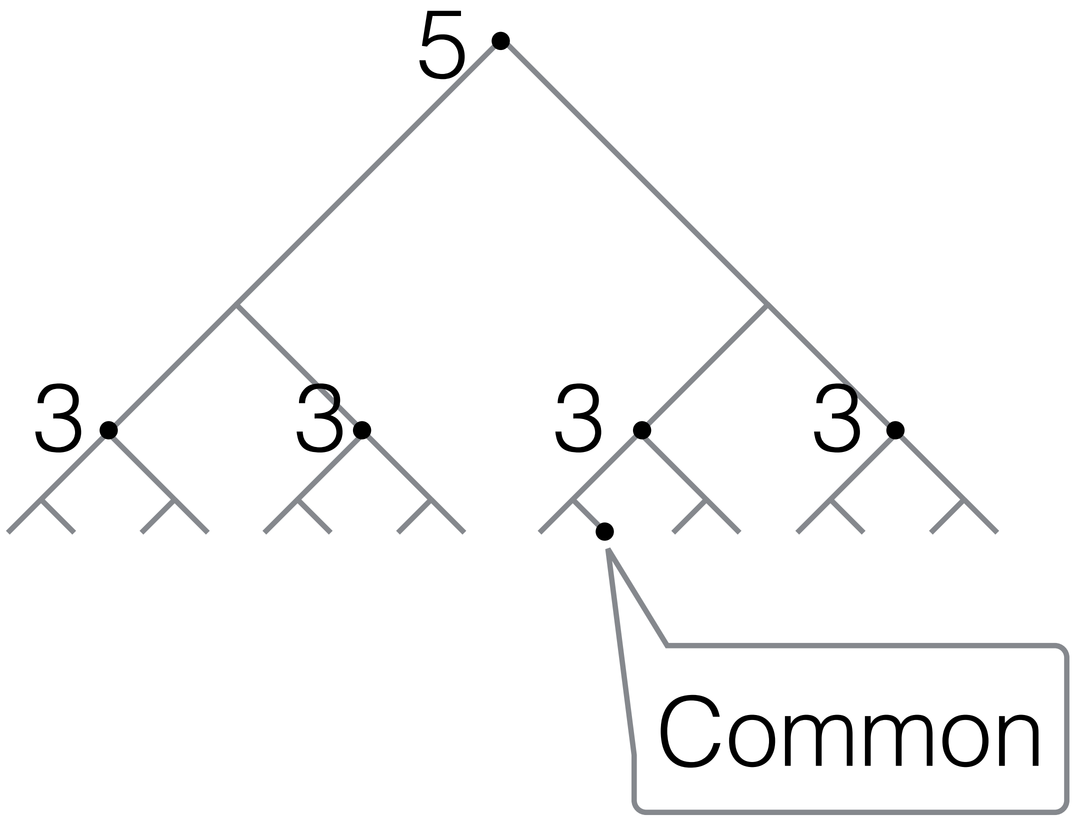


Balance

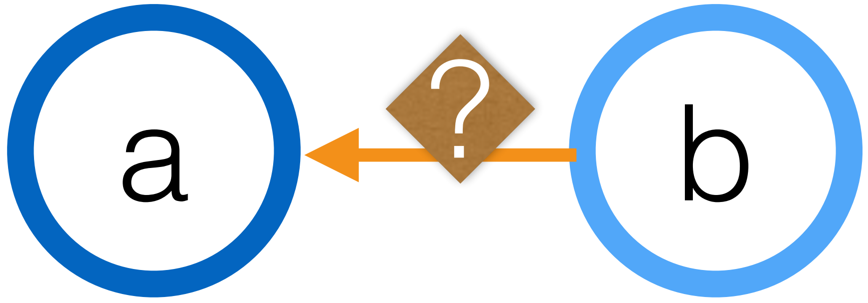


Levels provide a way to maintain the balance of the tree elements of a RAZ

Random levels drawn from distribution of levels in a (huge) binary tree

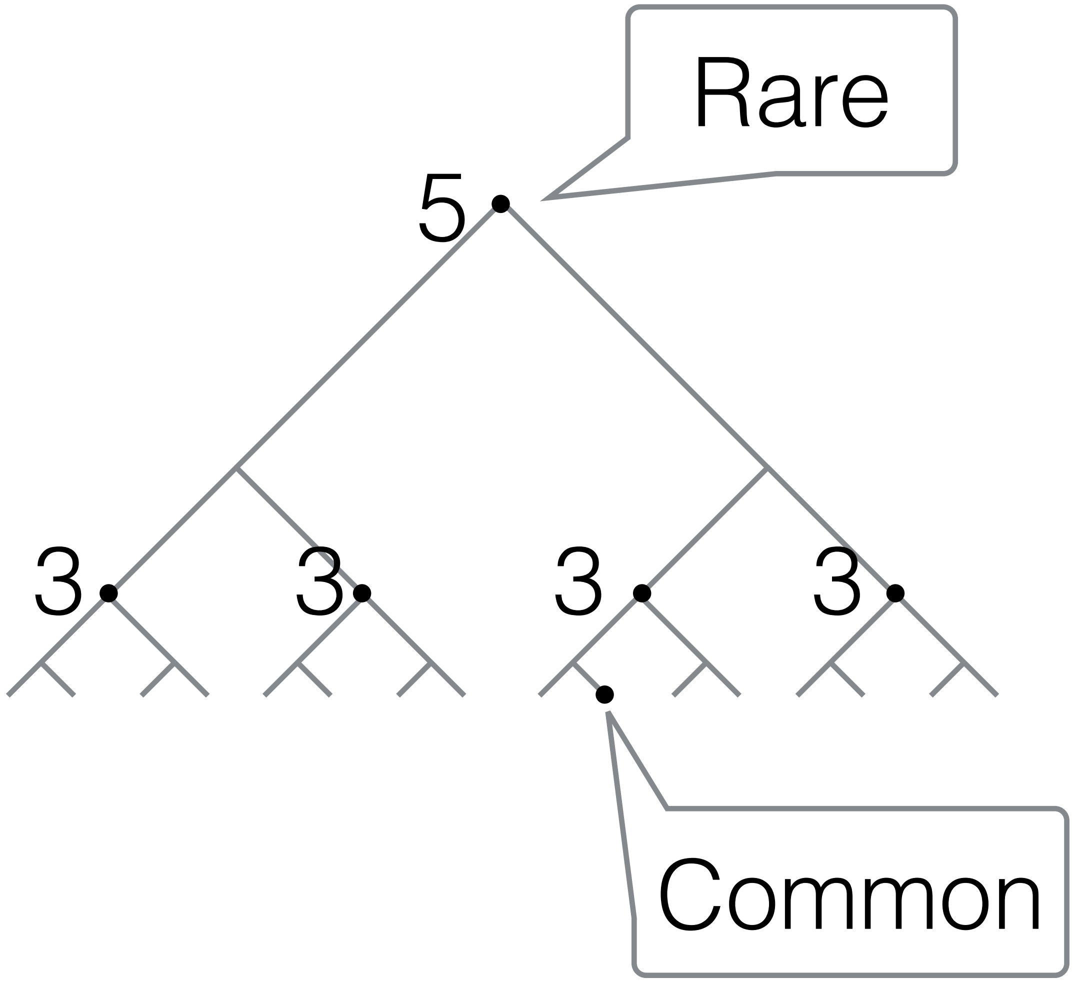


Balance



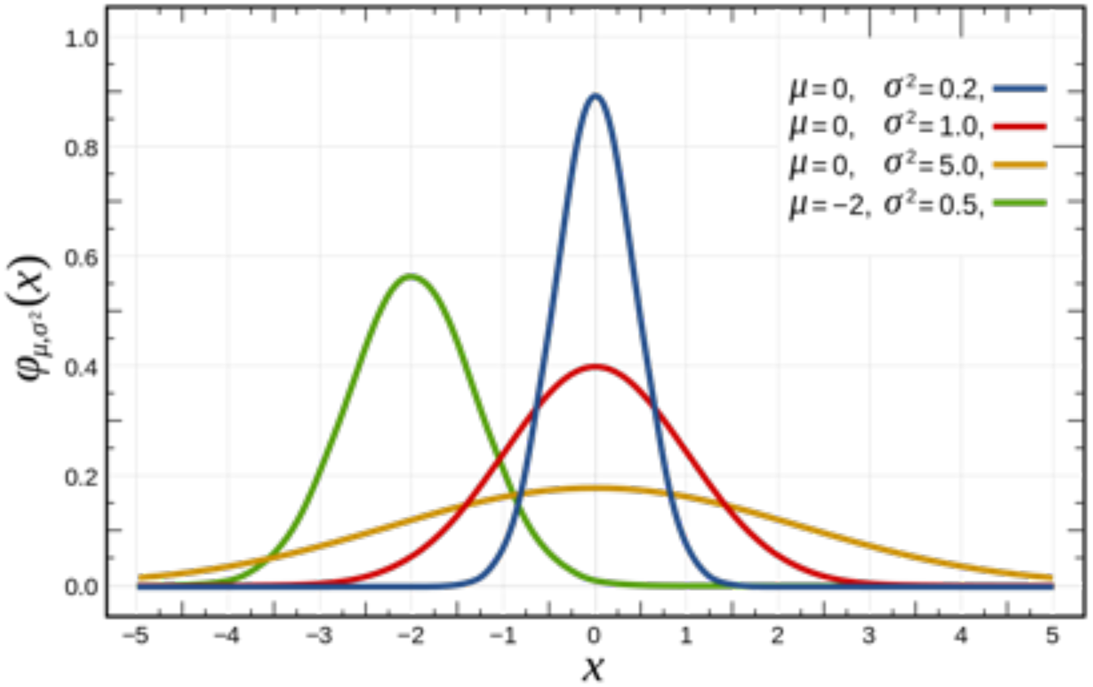
Levels provide a way to maintain the balance of the tree elements of a RAZ

Random levels drawn from distribution of levels in a (huge) binary tree



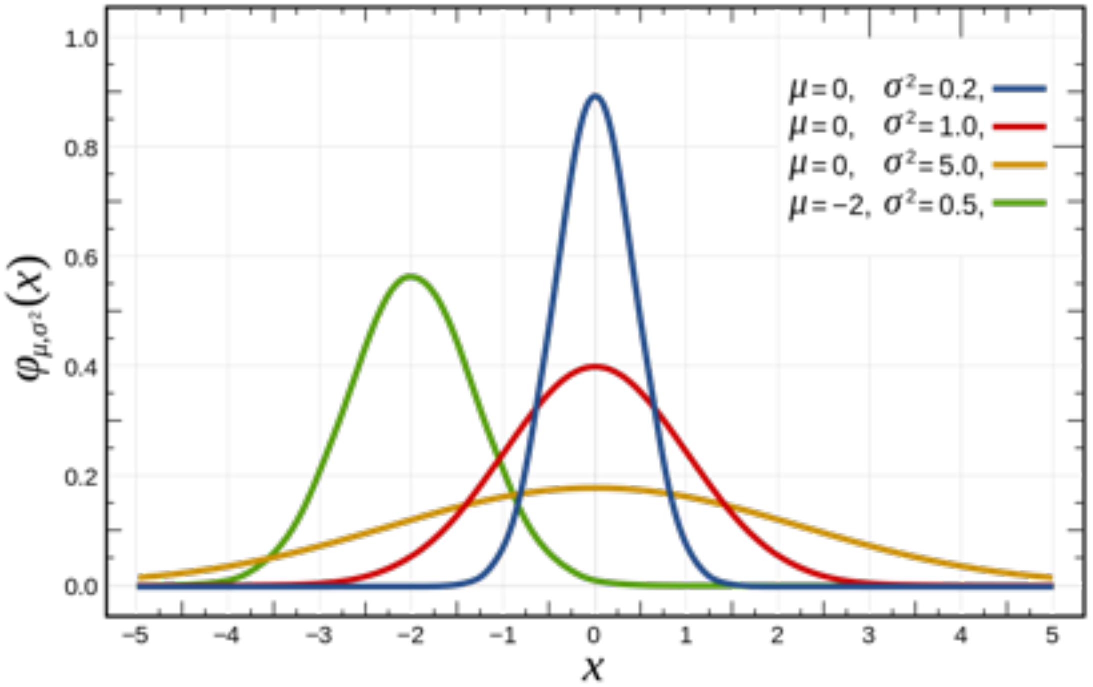
Balance

Because of the way randomness behaves, we get good balance at scale



Balance

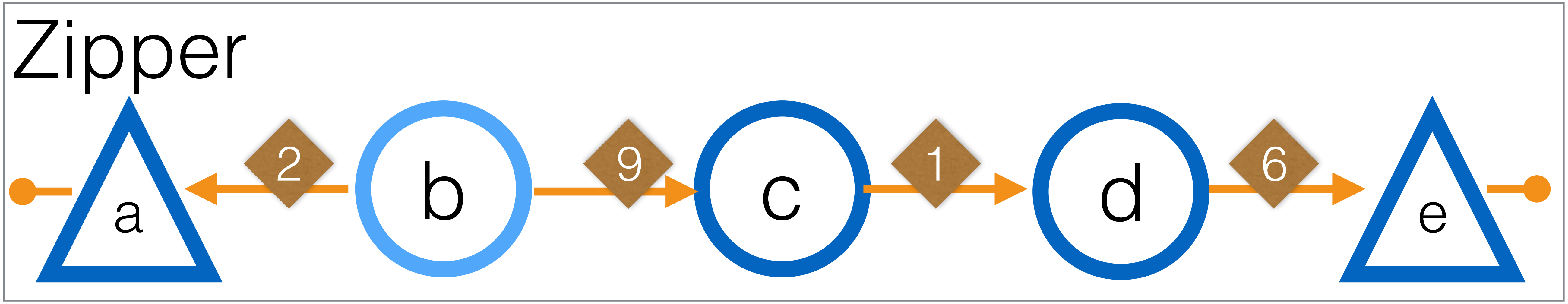
Because of the way randomness behaves, we get good balance at scale



Levels track tree balance, and we store then in list nodes so that height is not lost when deconstructed

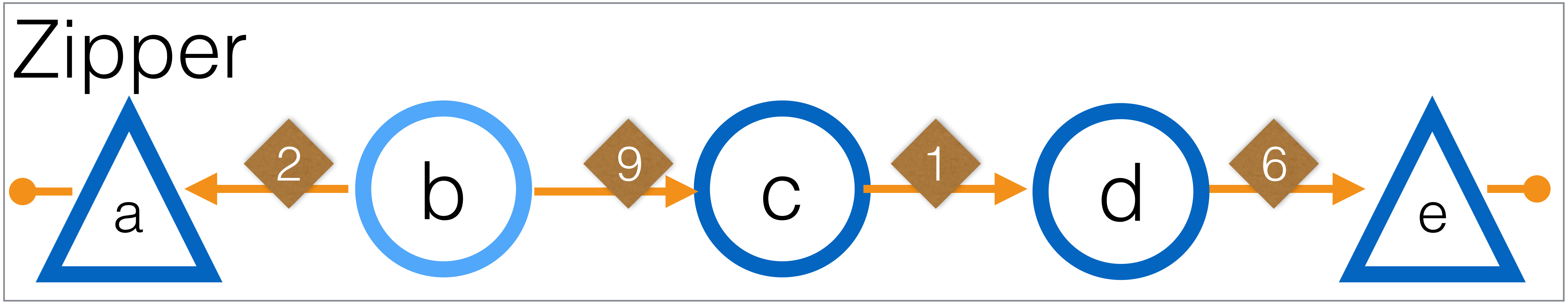
Two Forms of RAZ

Two Forms of RAZ

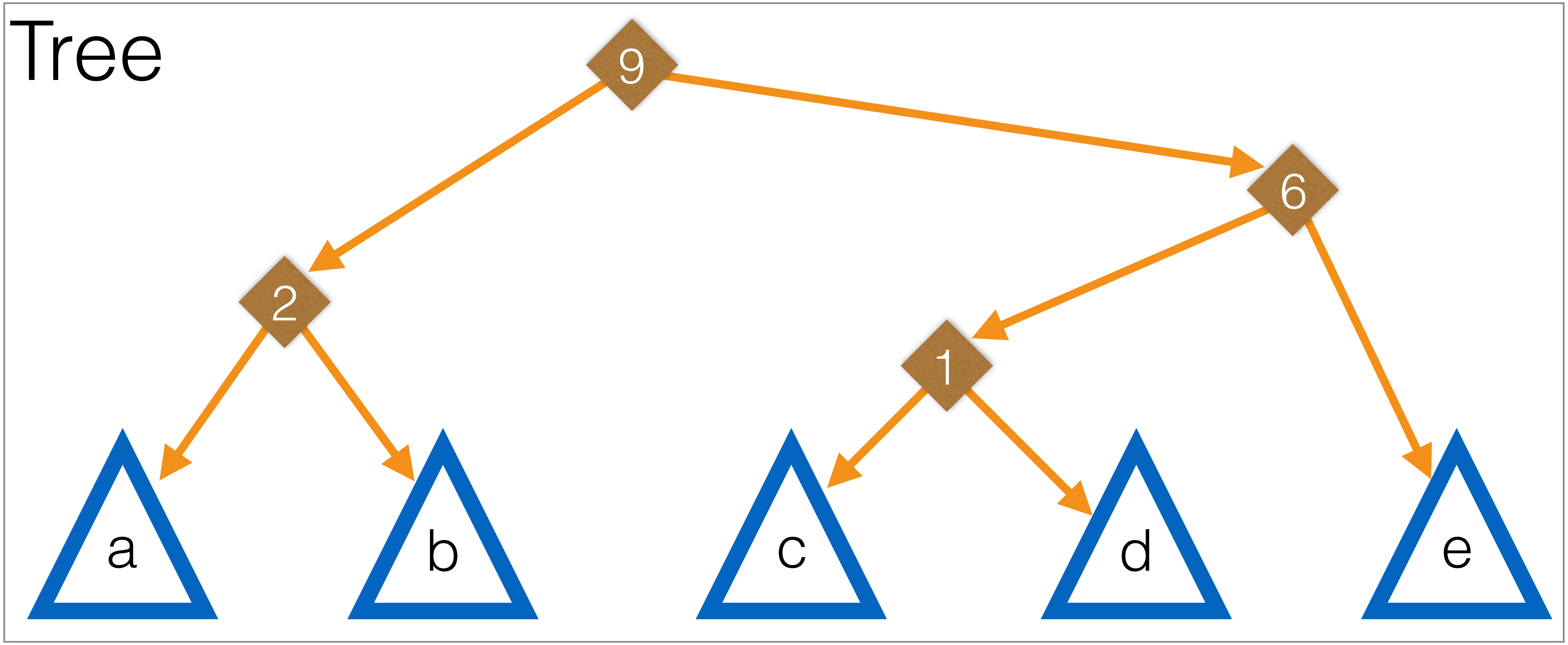


Two Forms of RAZ

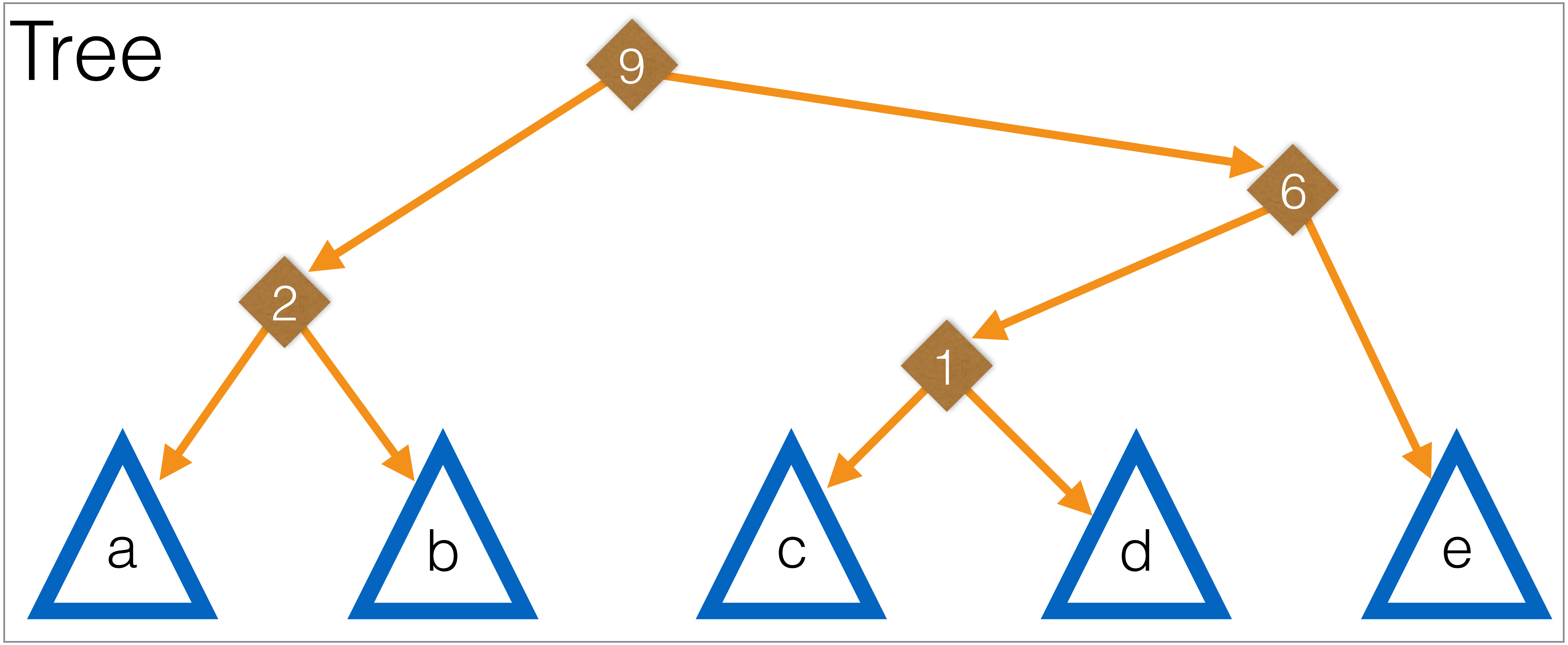
Editable: make insertions and deletions just like a common linked list



Two Forms of RAZ

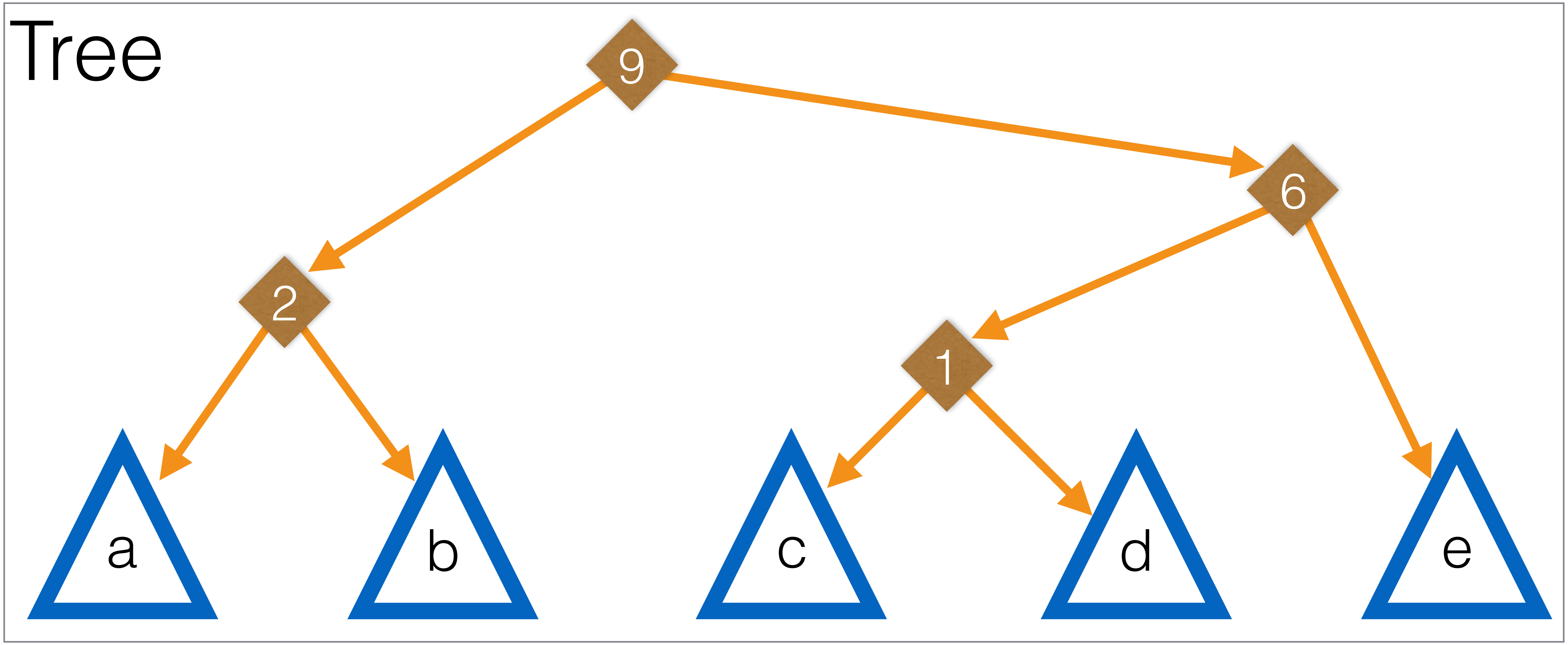


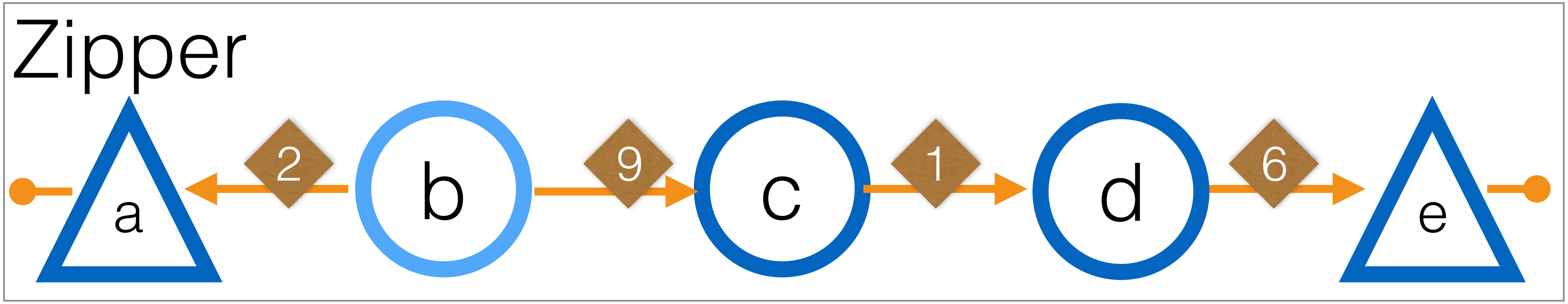
Two Forms of RAZ



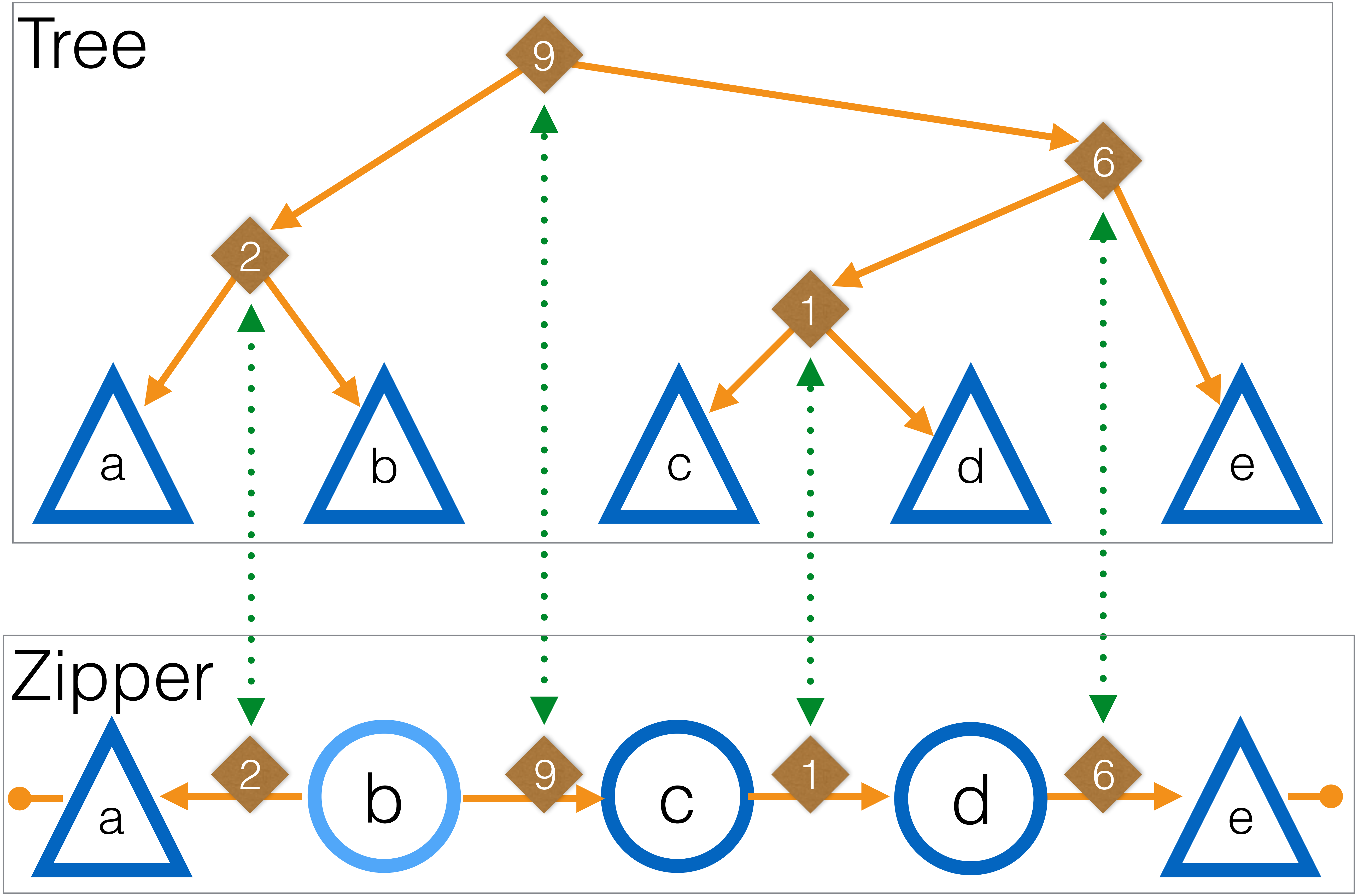
A single balanced binary tree: efficient searching algorithms

Two Forms of RAZ

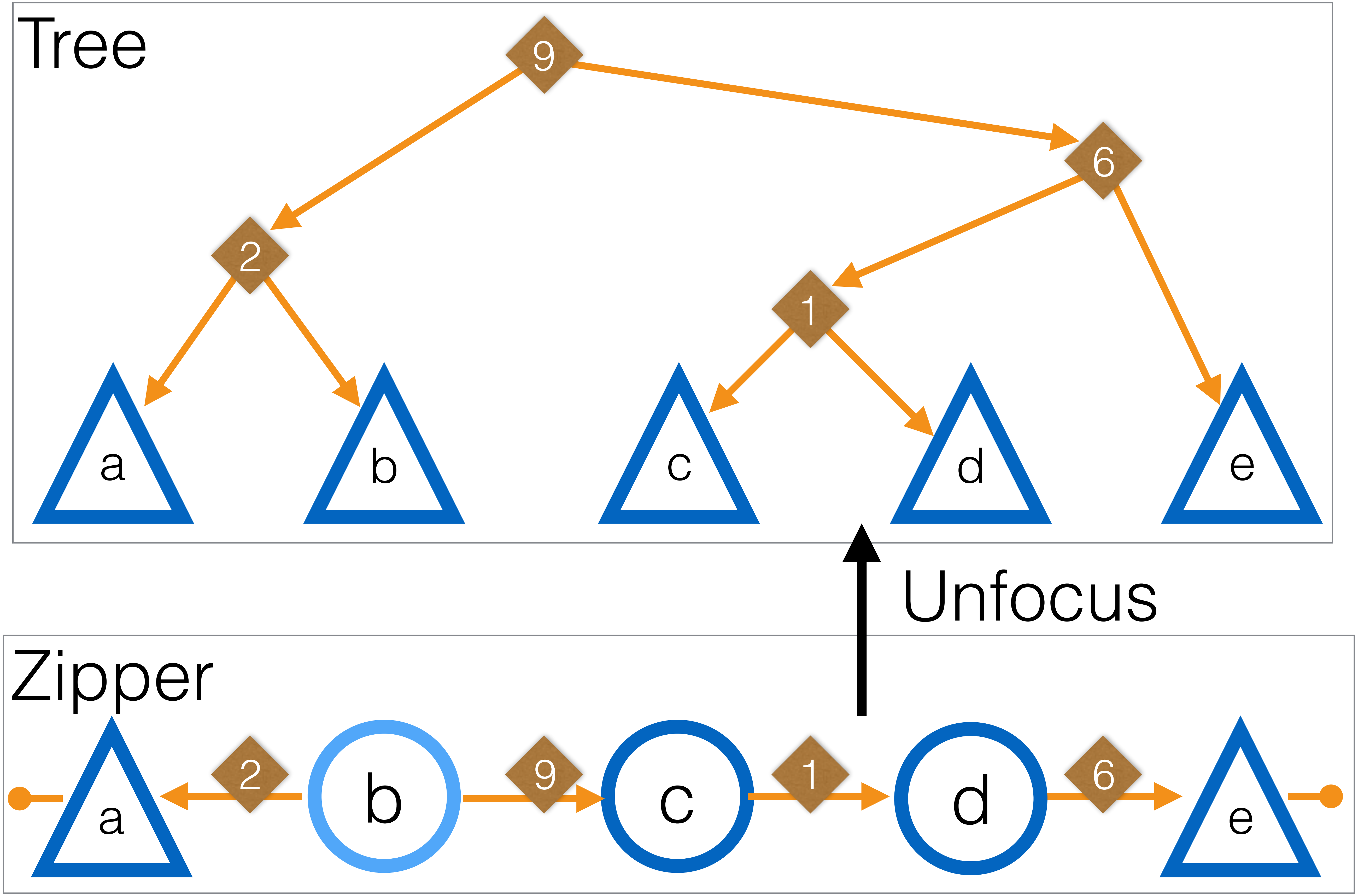




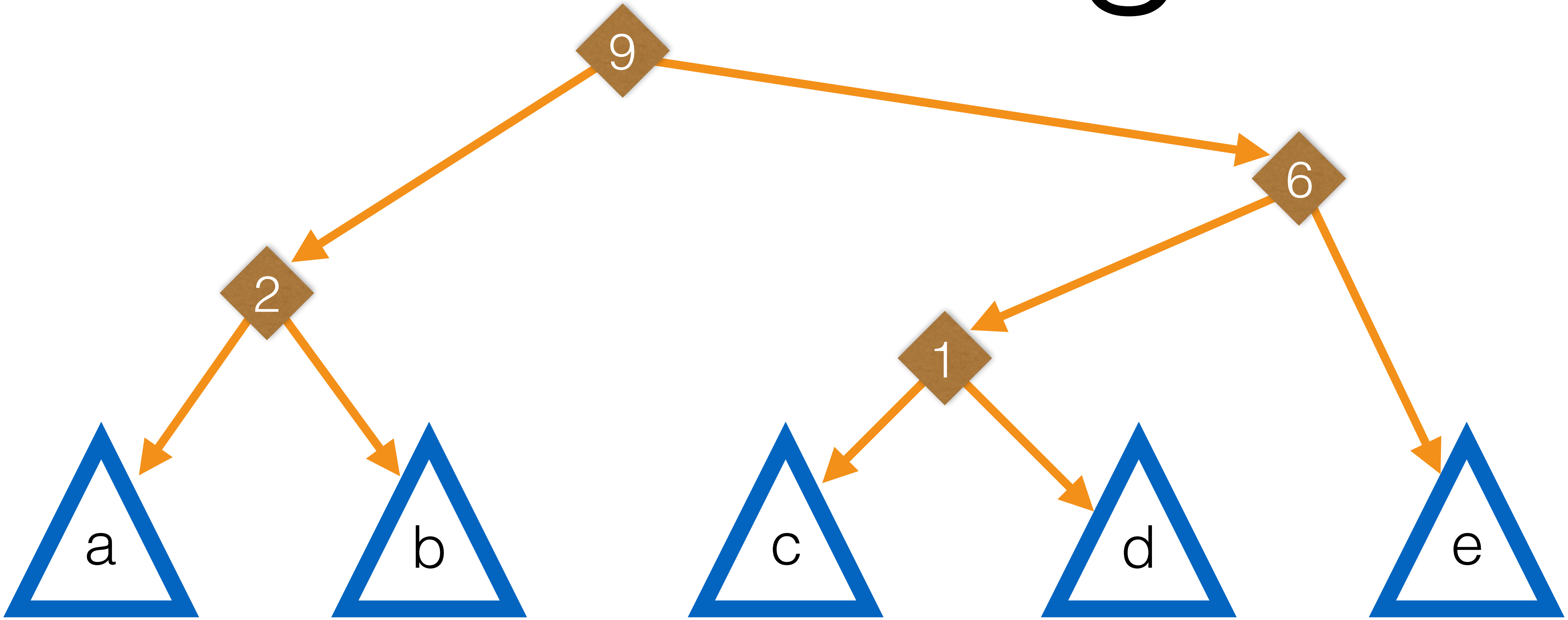
Two Forms of RAZ



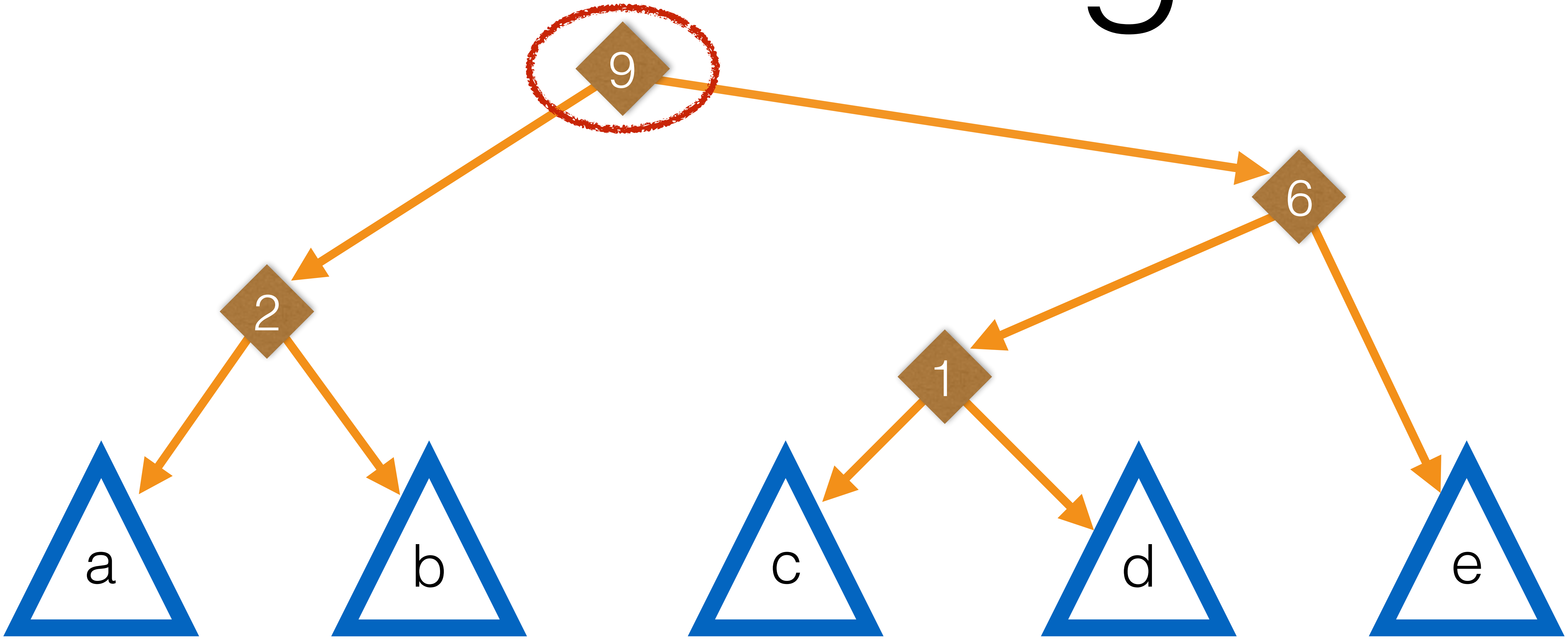
Two Forms of RAZ



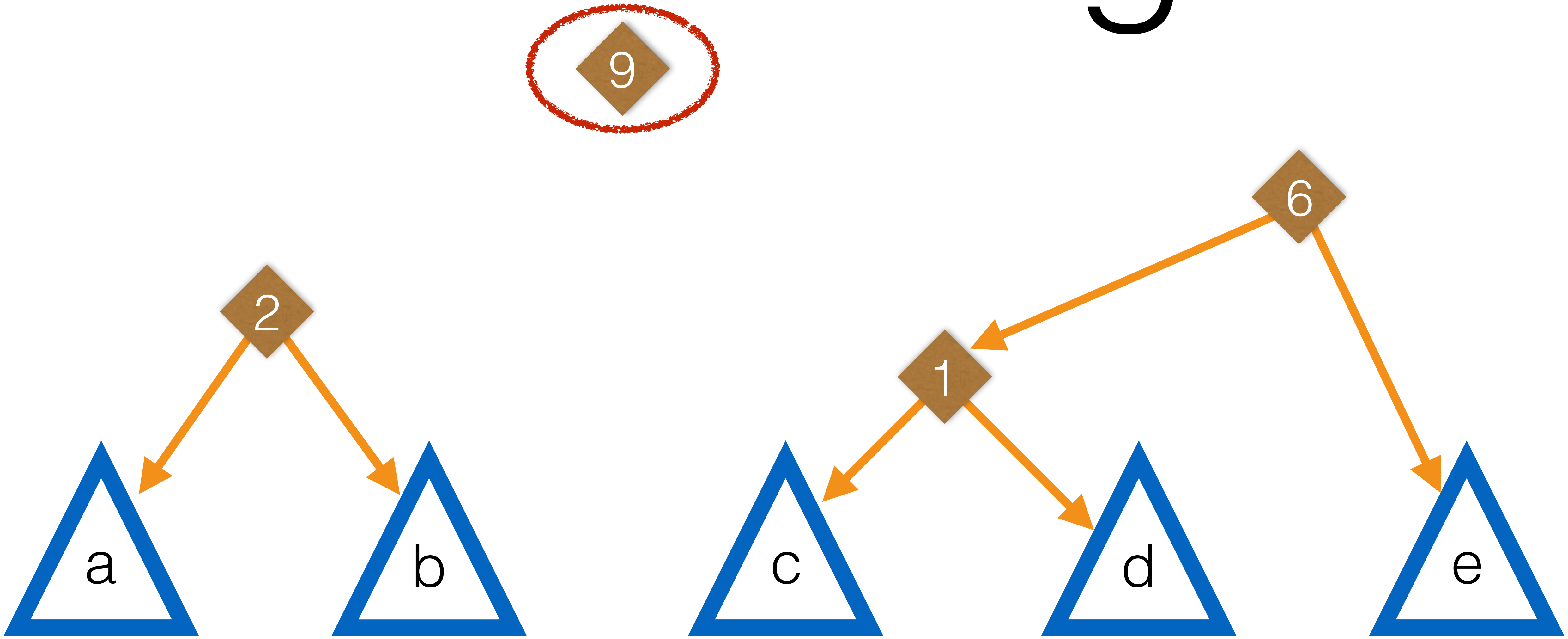
Focusing



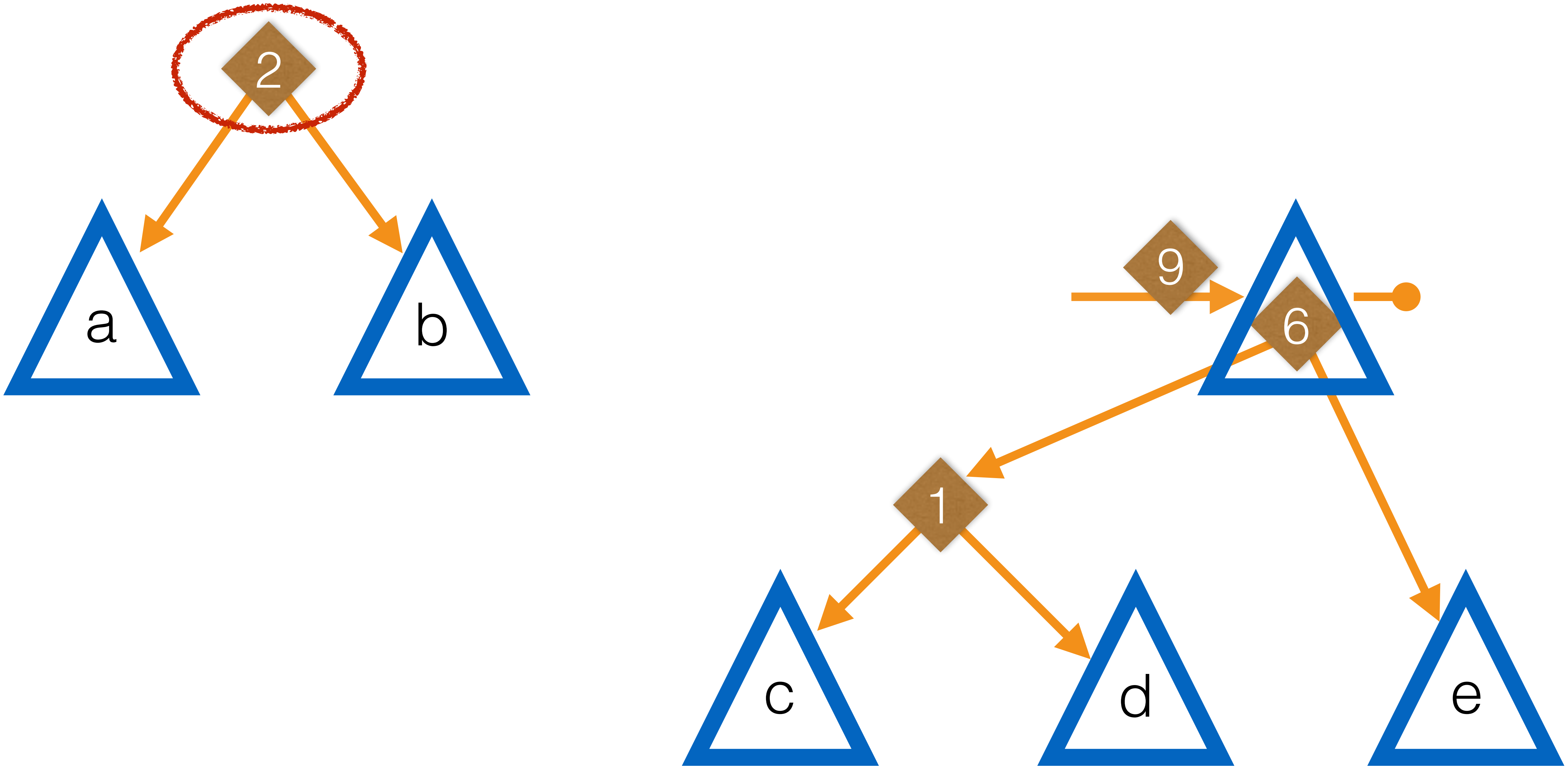
Focusing



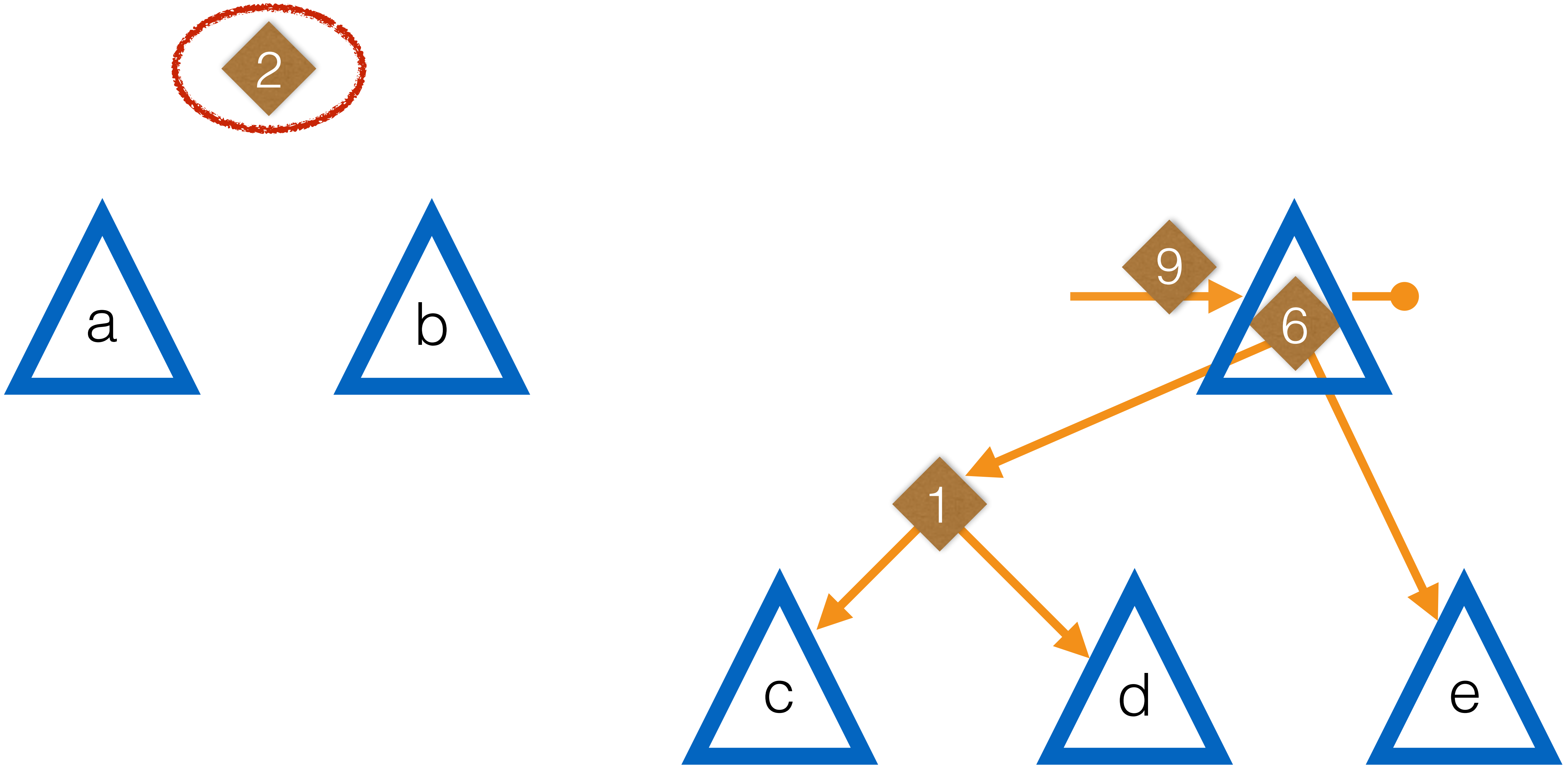
Focusing



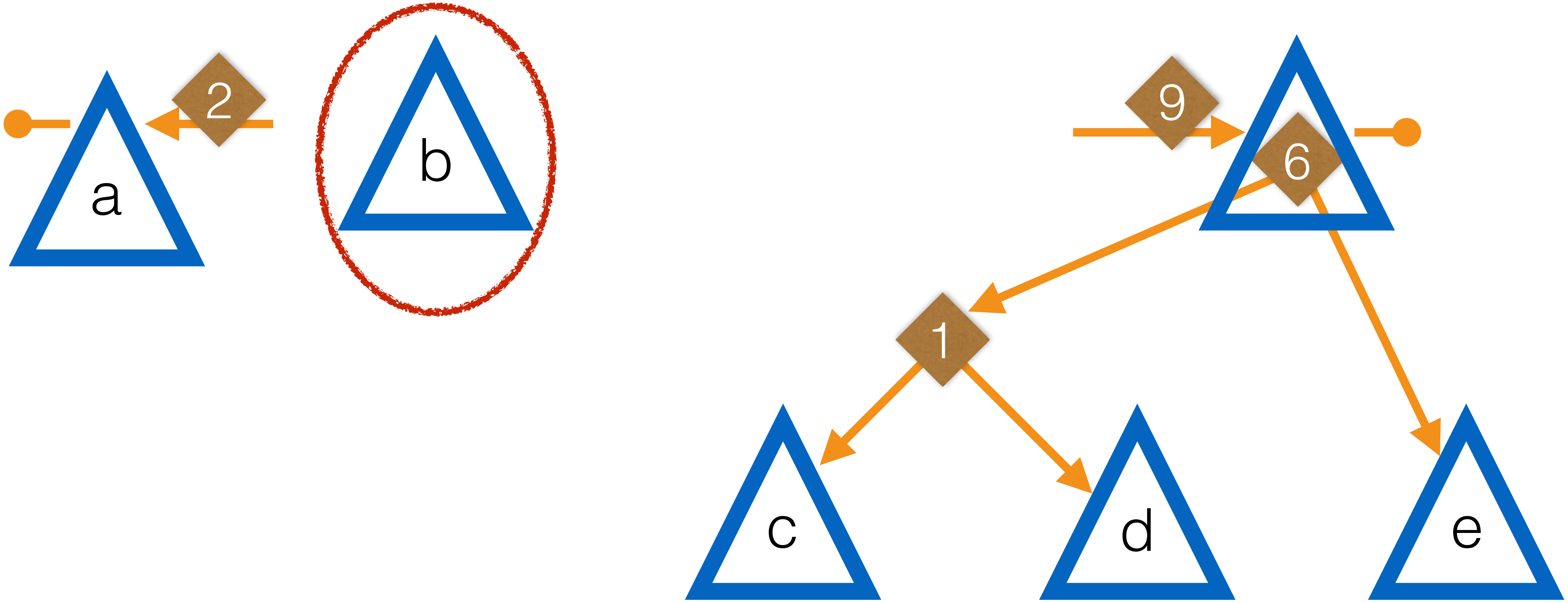
Focusing



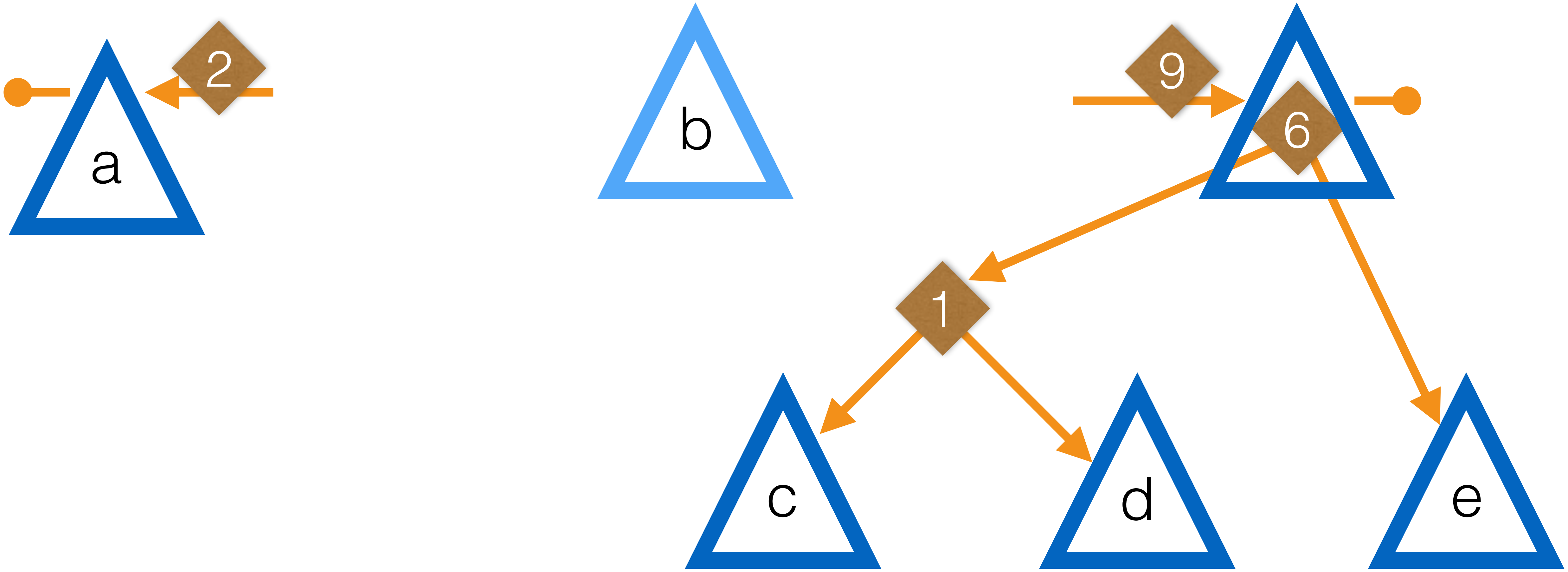
Focusing



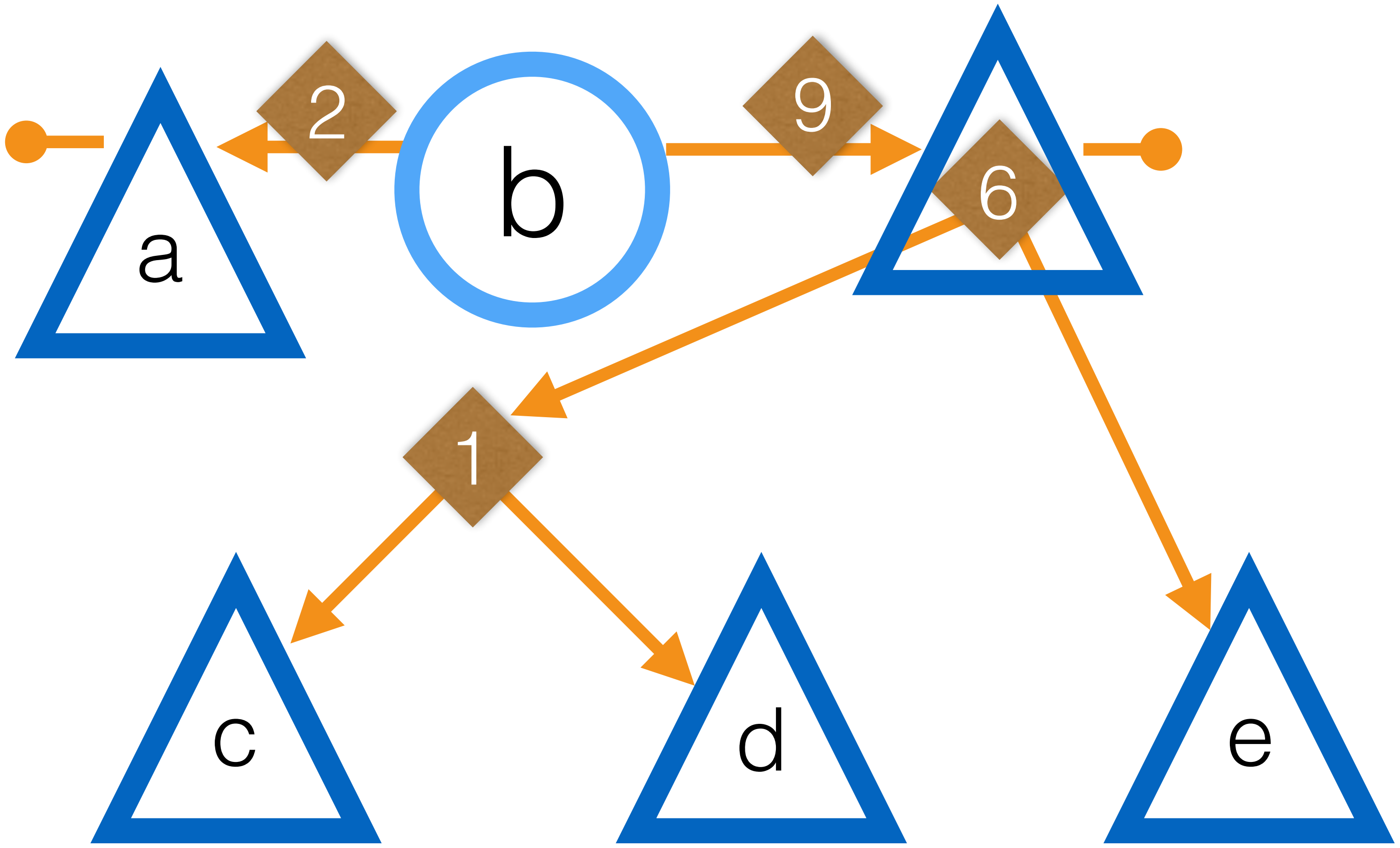
Focusing



Focusing



Focusing



Experiments

Experiments

RAZ in OCaml

Experiments

RAZ in OCaml

Fingertree in OCaml

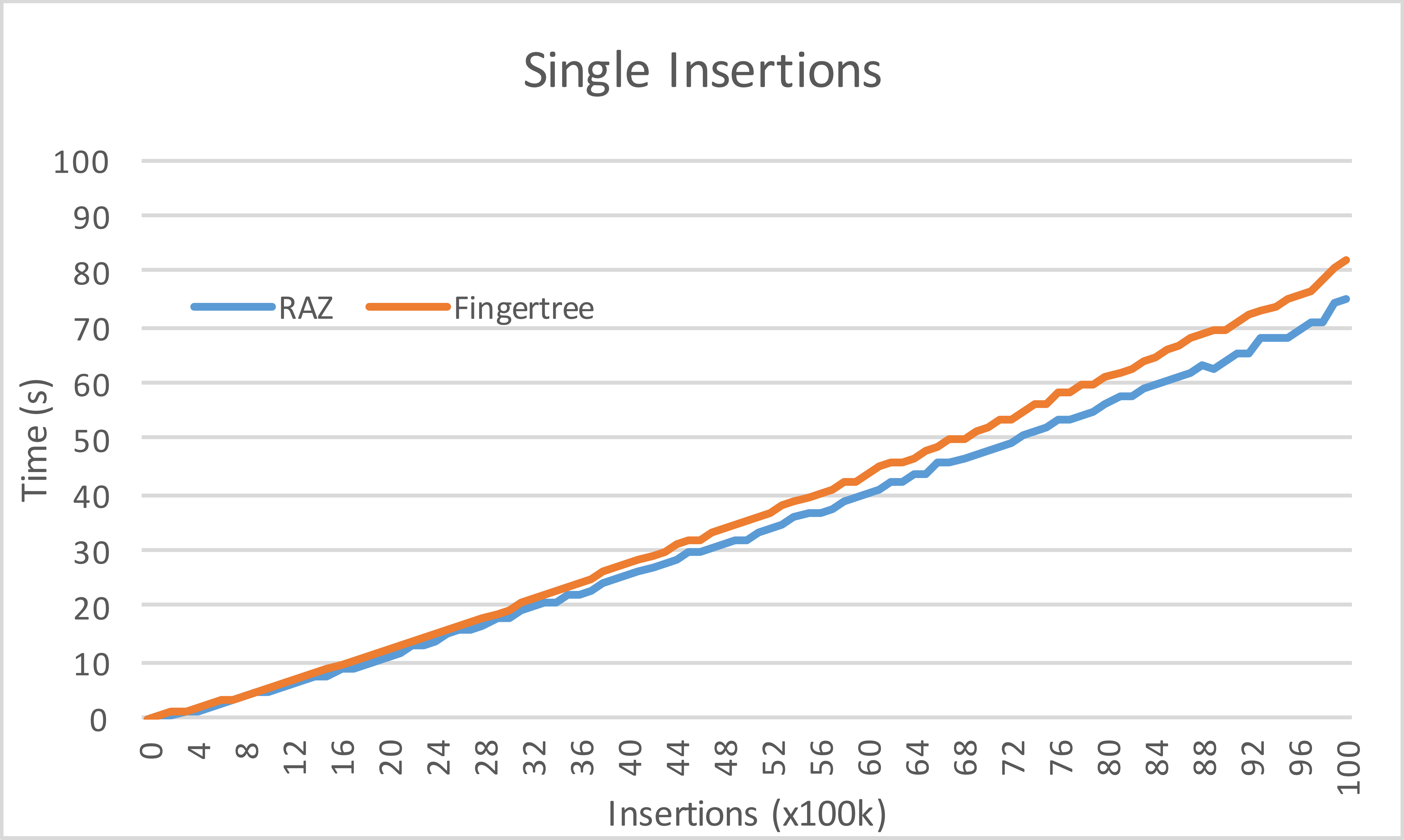
Experiments

RAZ in OCaml

Fingertree in OCaml

Build a sequence by insertions at random points

Insertion at random



Random Access Zipper

Accessible	Focus/Unfocus
Editable	No edit rebalance
Simple	< 200 LoC
Fast	Beats Fingertree

Random Access Zipper

Simple enough to include these principles in your own data types!

More info at kyleheadley.github.io