Random Access Zipper

RAZ

Presented by Kyle Headley

TFP'16 College Park

Functional programmers
want simple data types

What do we have
for sequences?

Zippers are great

Zippers are great

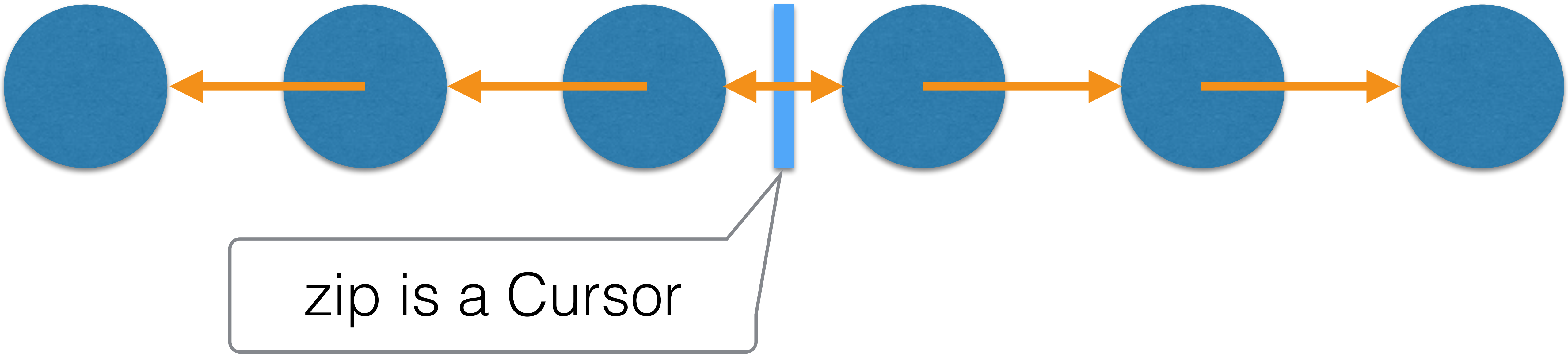
```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list

type ‘a zip =
  ‘a list * ‘a list
```

Zippers are great

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

type ‘a zip =
  ‘a list * ‘a list
```



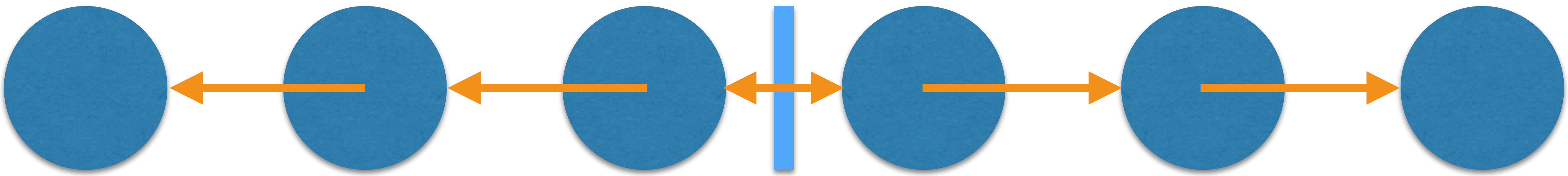
zip is a Cursor

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
```



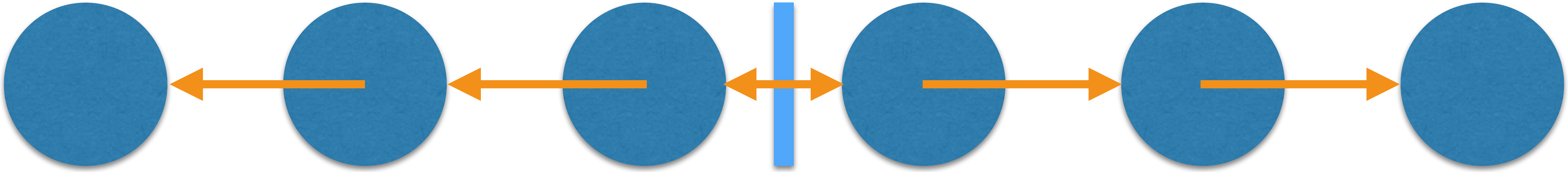
Zippers are great

```
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)!

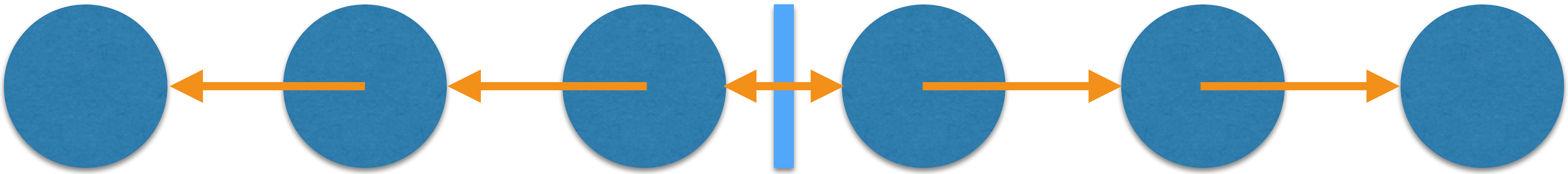


Zippers are great

```
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
```



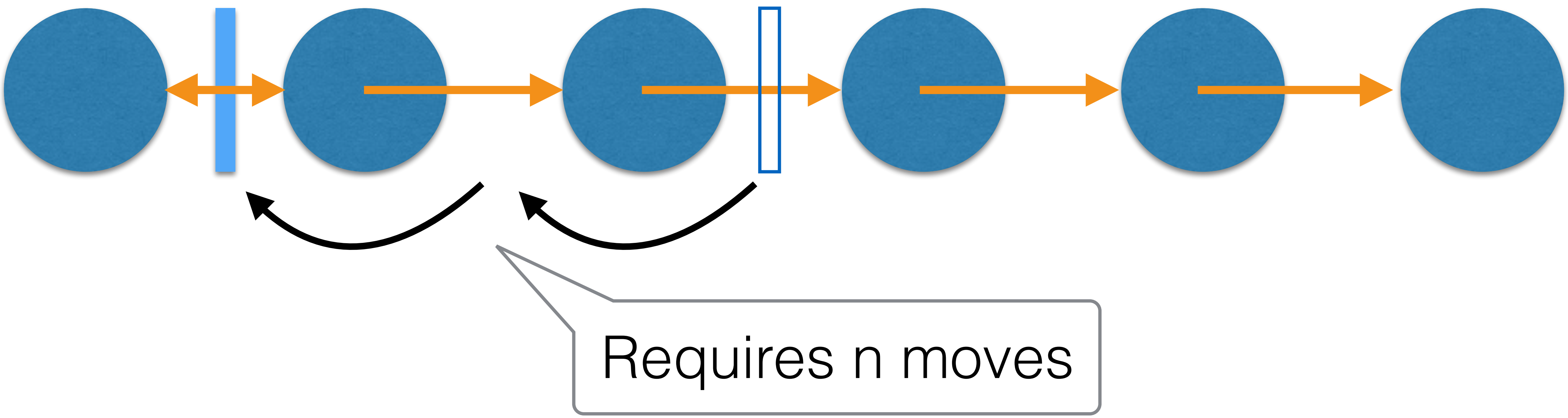
Problem: Slow random access

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

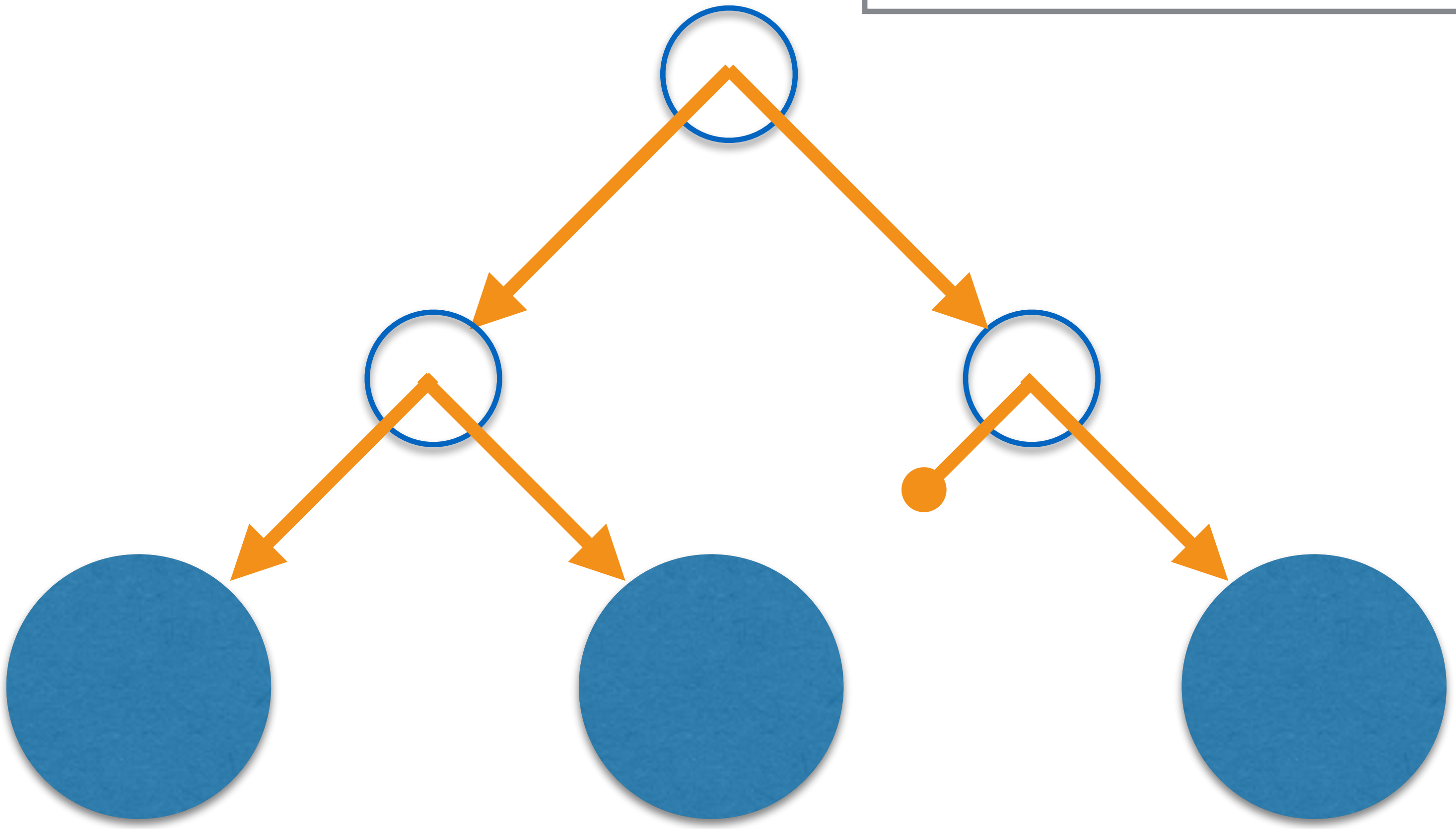
Trees are great

```
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
```

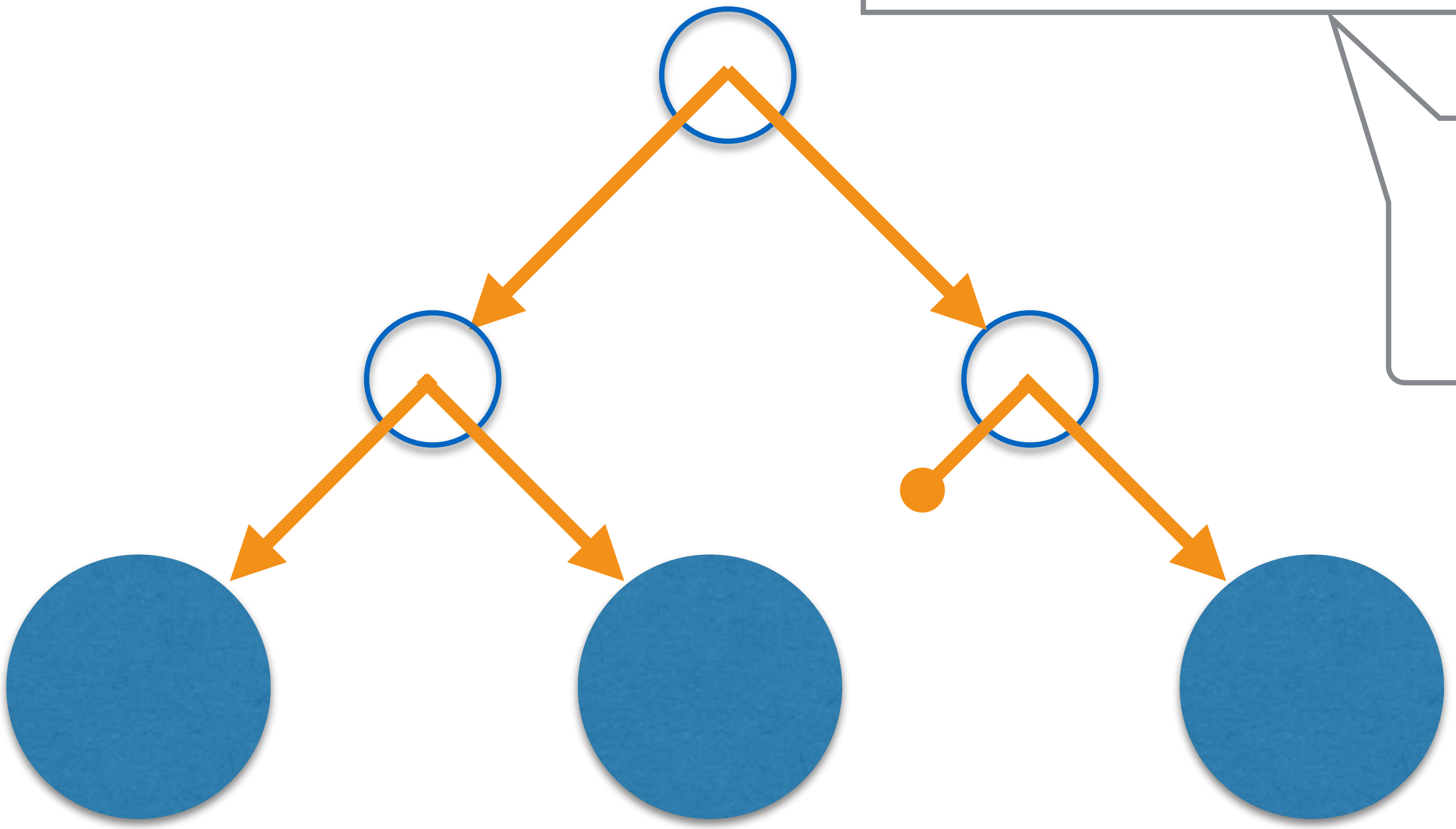


Trees are great

```
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)

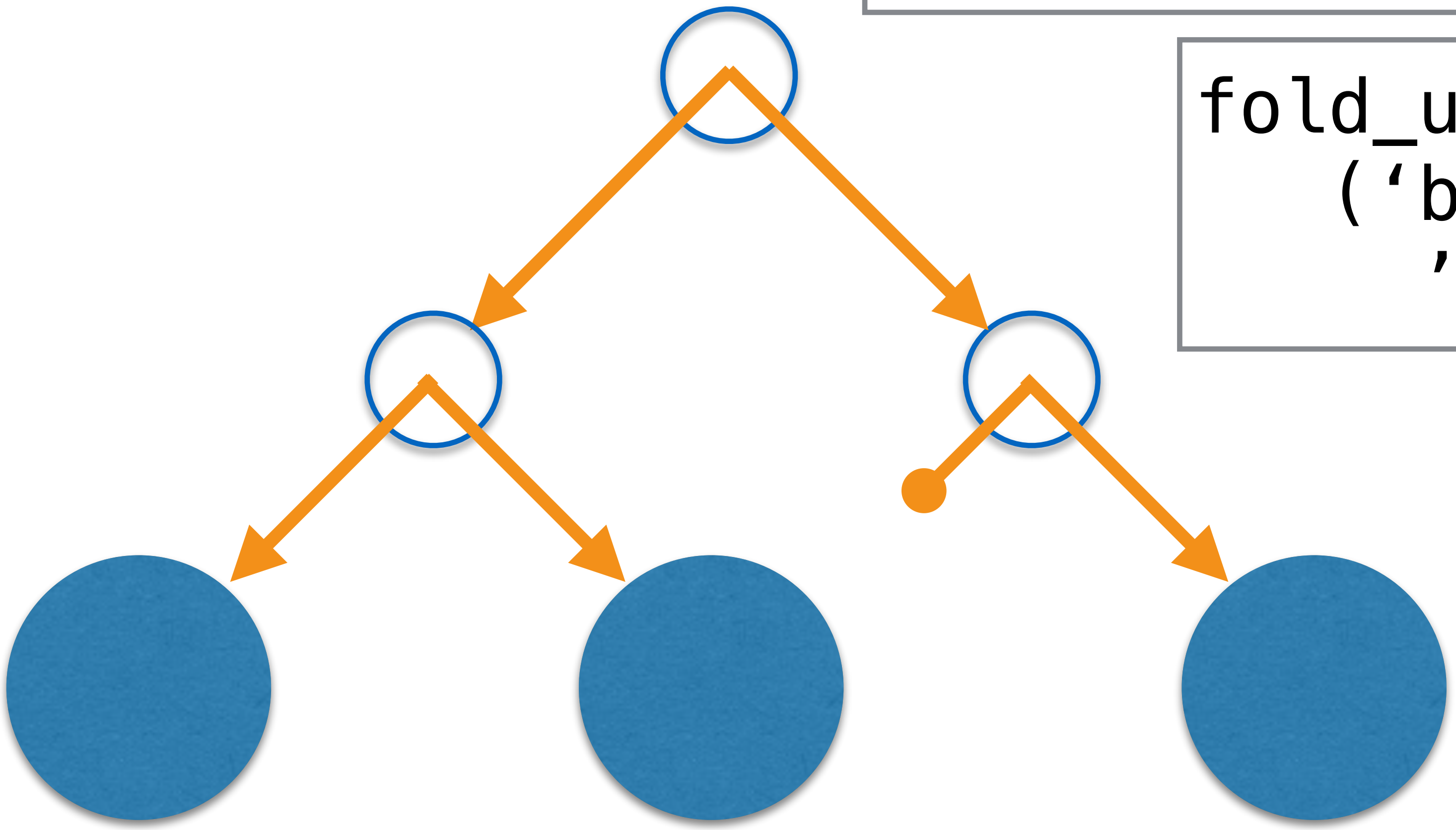


Trees are great

```
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
```

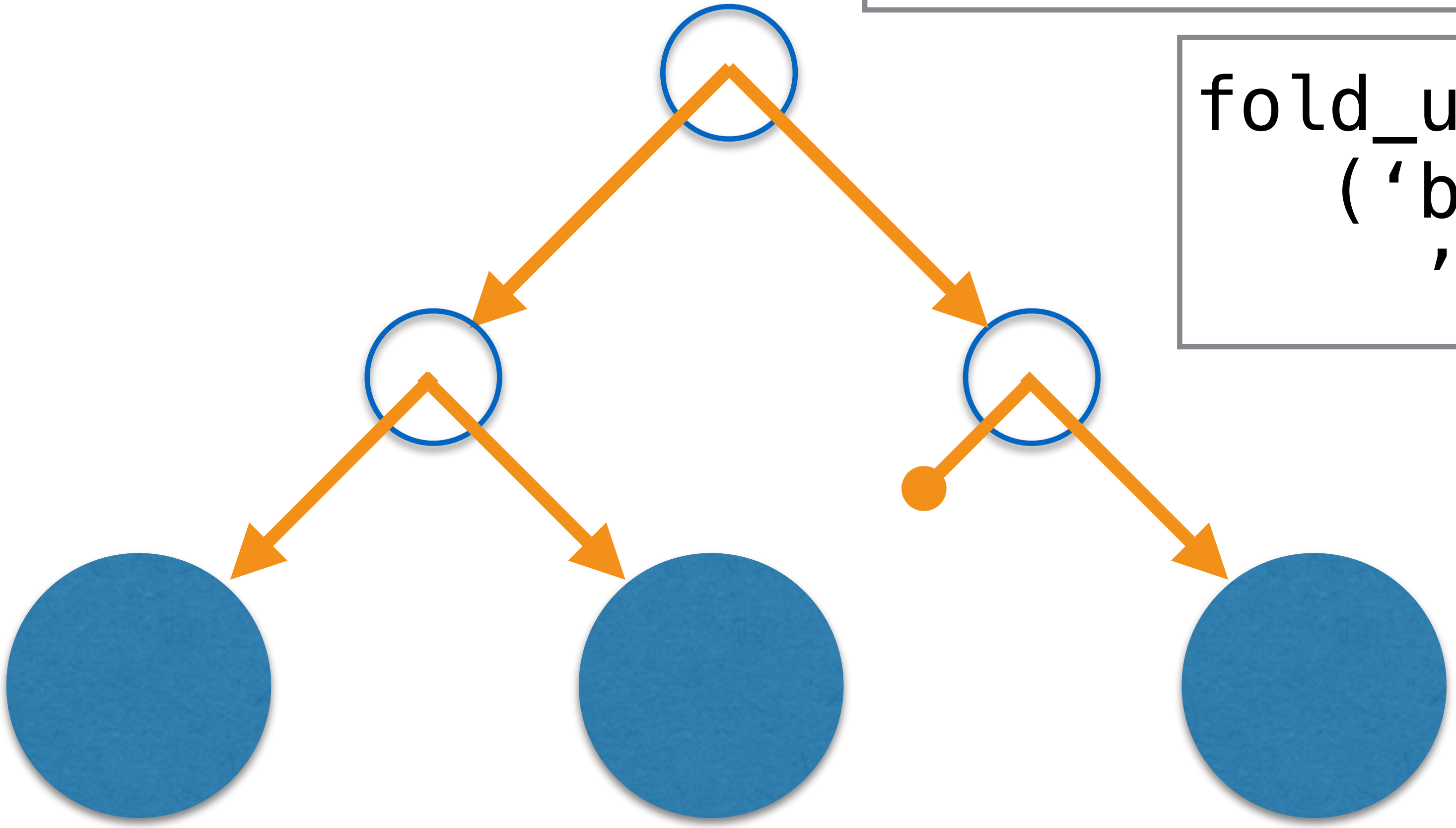
```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```



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
```



```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```

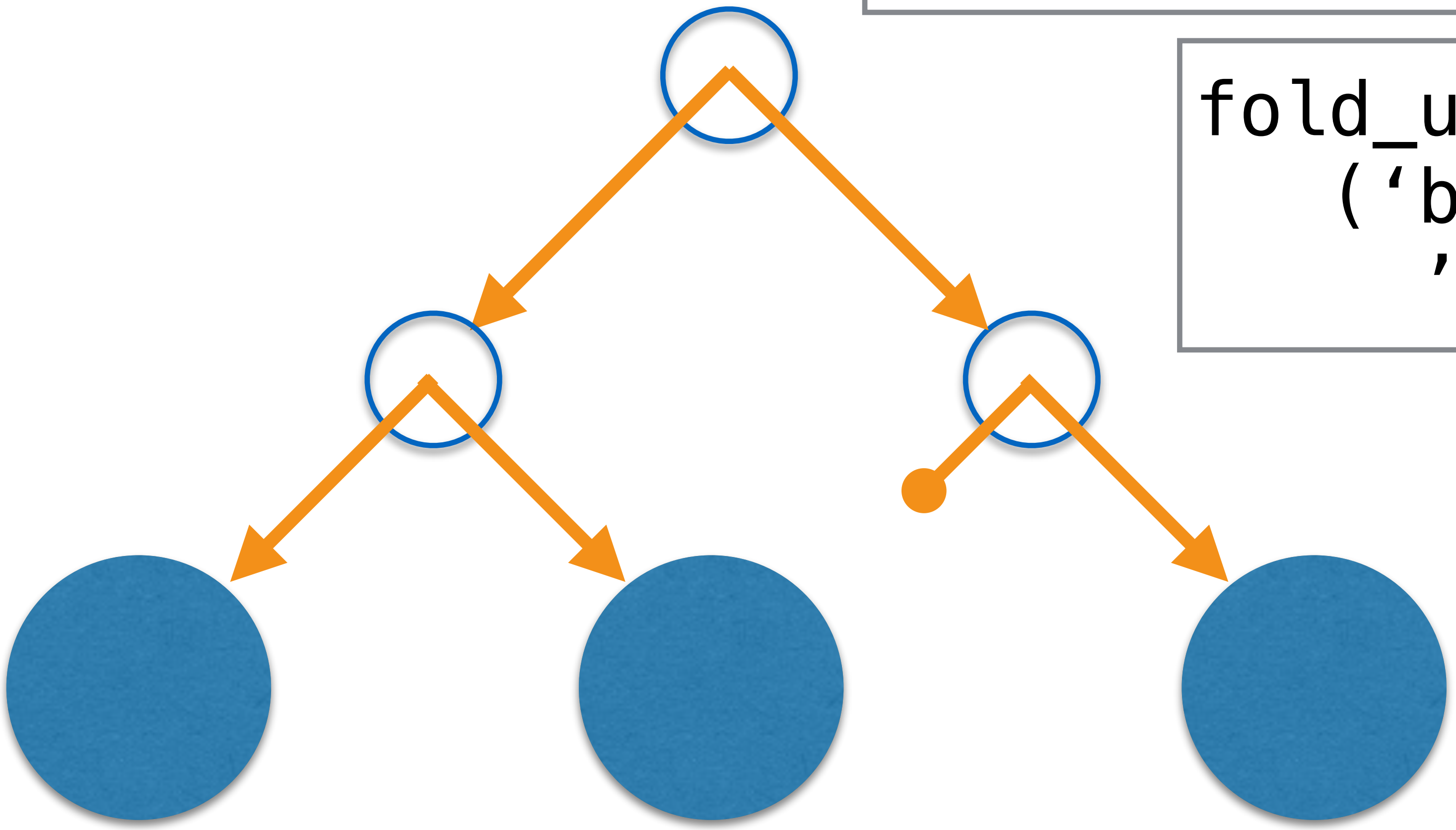
Nice incremental and parallel properties

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
```

```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```



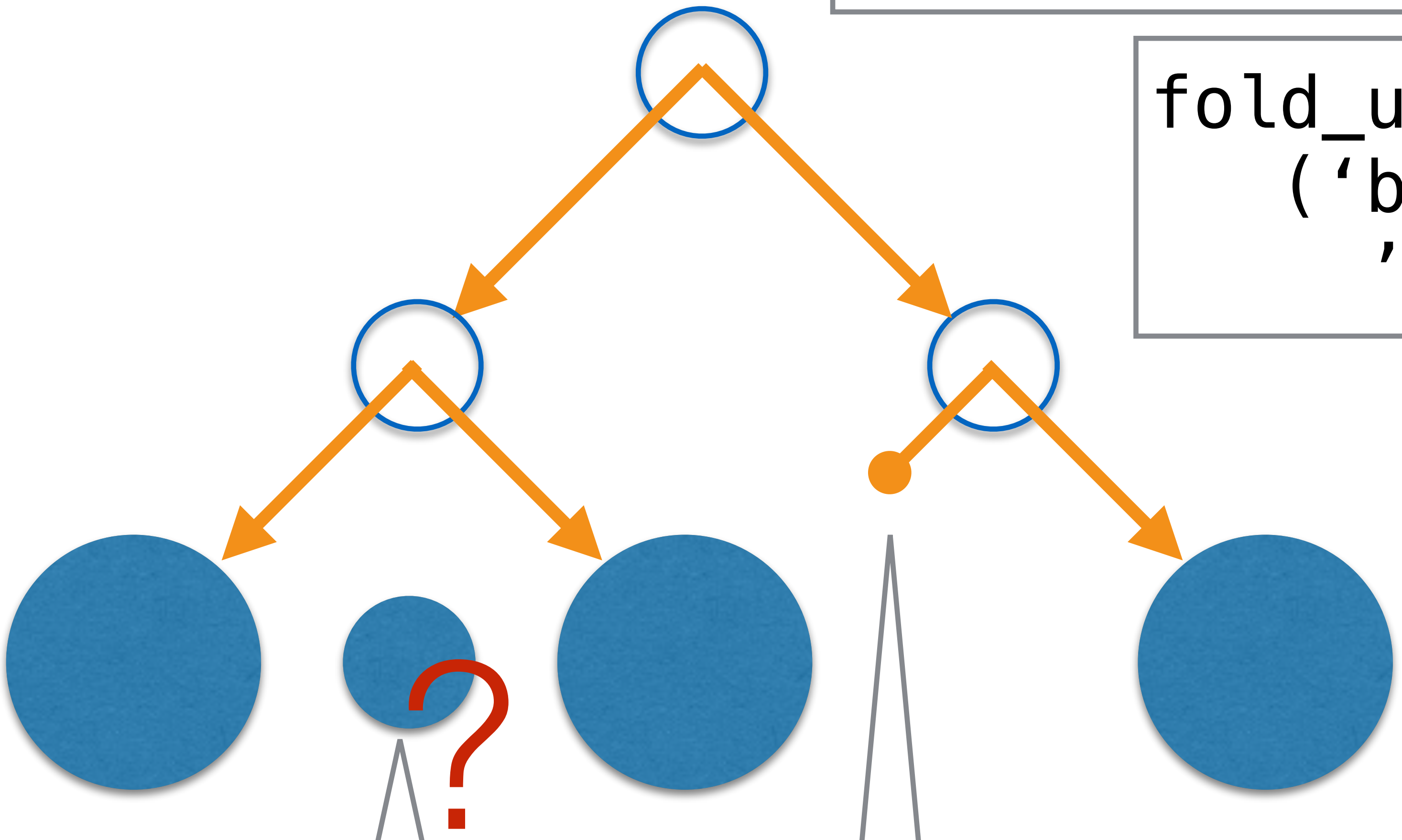
Problem: Reasoning about edits

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
```

```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```



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
```

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
```

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
```

```
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

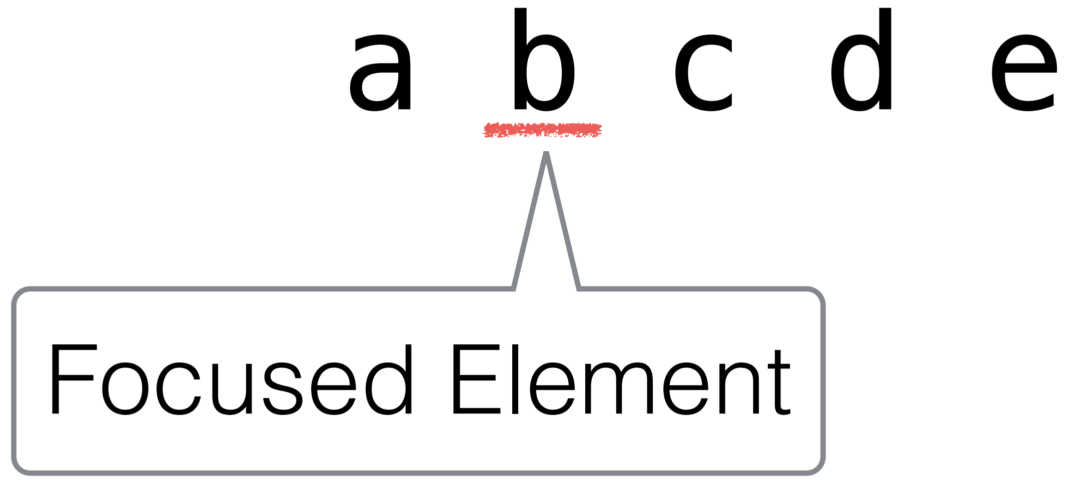
Using a RAZ

`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
```

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
```

```
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
```

```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```

Still get tree info

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
```

```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```

```
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
```

```
fold_up: (‘a -> ‘b) ->
   (‘b -> ‘b -> ‘b) ->
     ’a tree -> ‘b -> ‘b
```

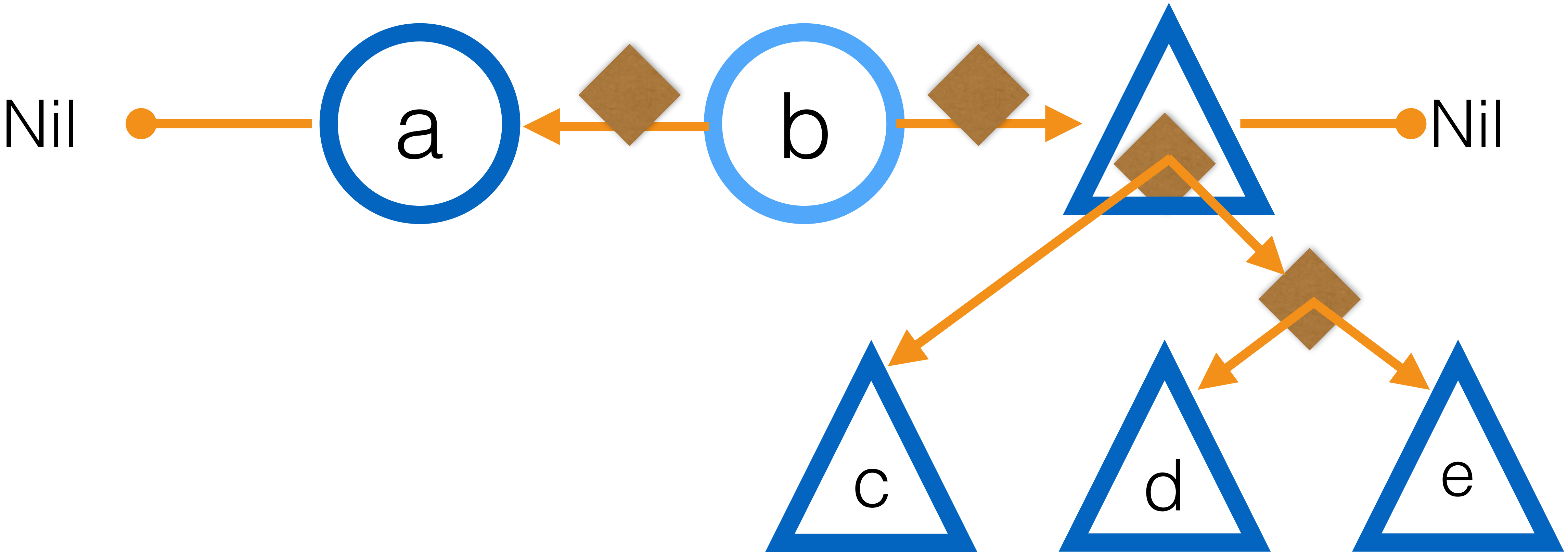
```
move:    dir ->
  ‘a zip -> ‘a zip
insert:  dir -> ‘a ->
  ‘a zip -> ‘a zip
remove:  dir ->
  ‘a zip -> ‘a zip
```

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

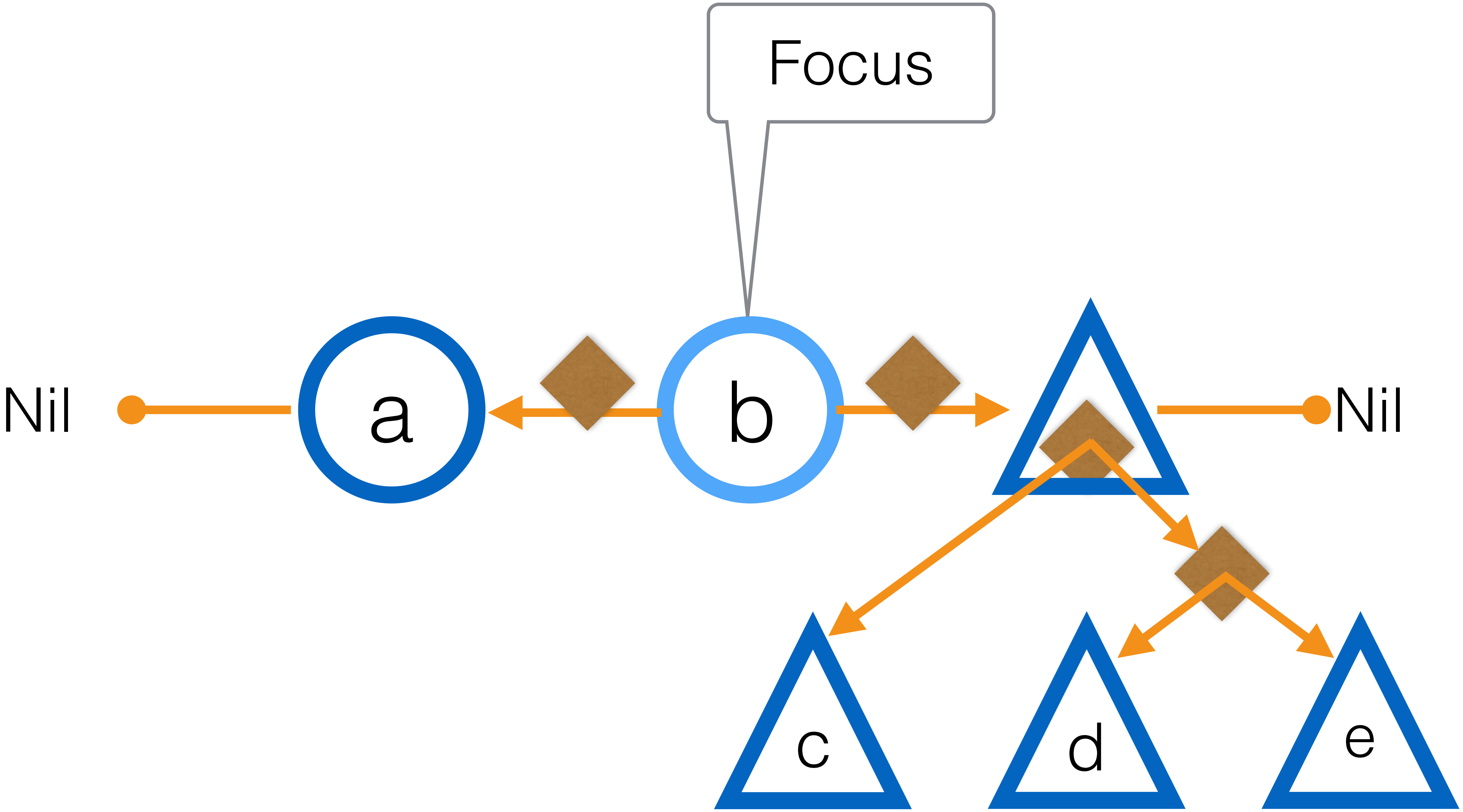
Both O(log n)!

(plus net insertions)

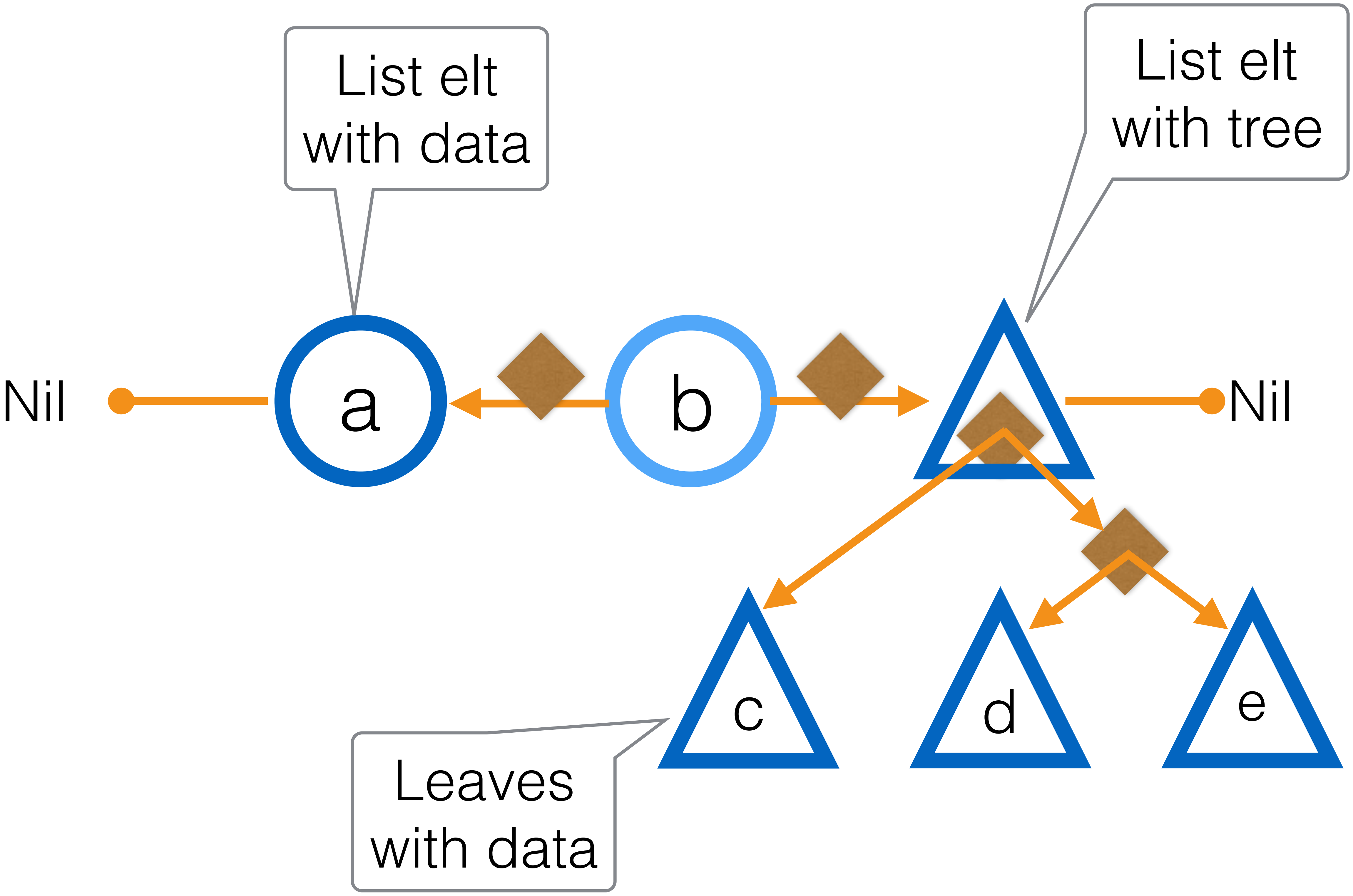
Zipper of Trees



Zipper of Trees



Zipper of Trees

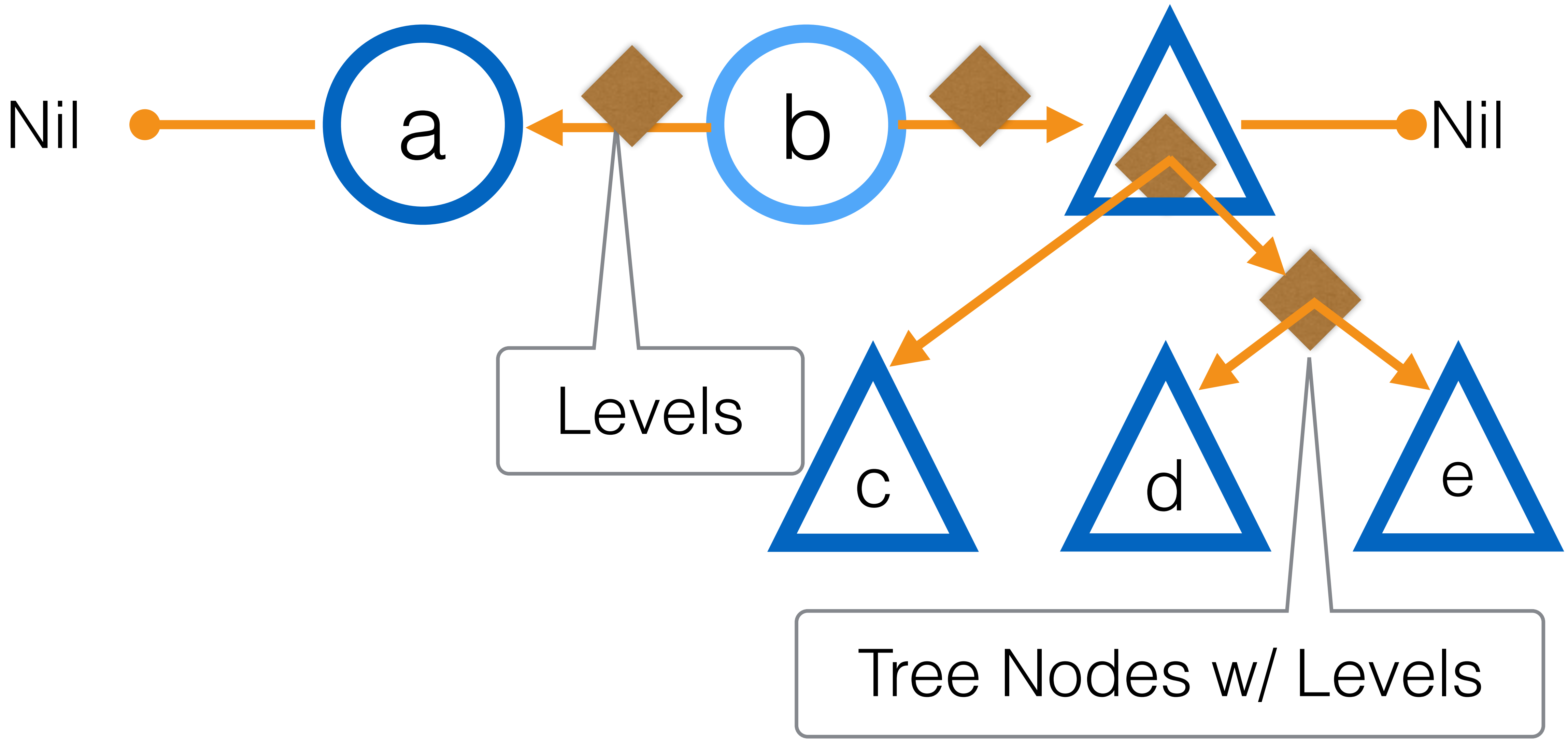


List elt with data

List elt with tree

Leaves with data

Zipper of Trees



Levels

Tree Nodes w/ Levels

Balance

Balance

We use a probabilistic balance, inserting random numbers as levels

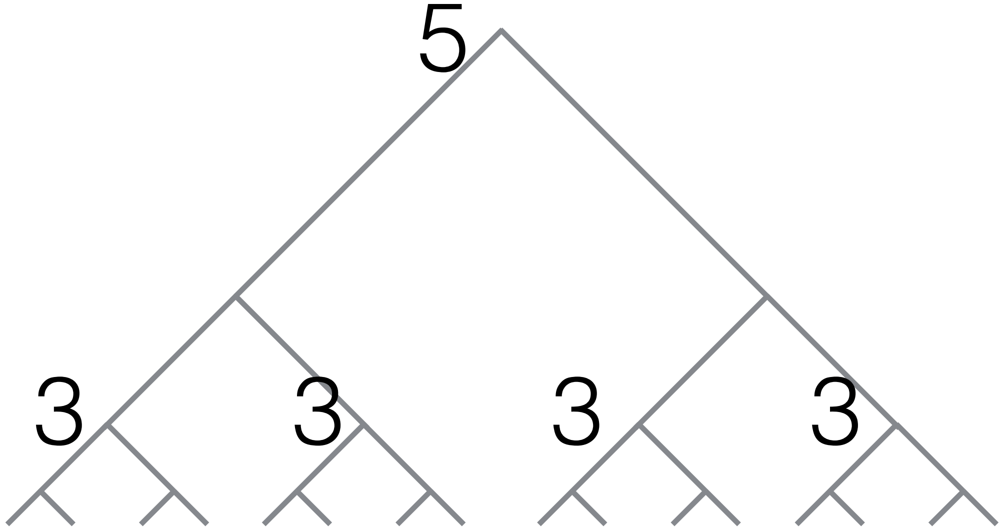
Because of the way randomness behaves, we get good balance at scale

Balance

We use a probabilistic balance, inserting random numbers as levels

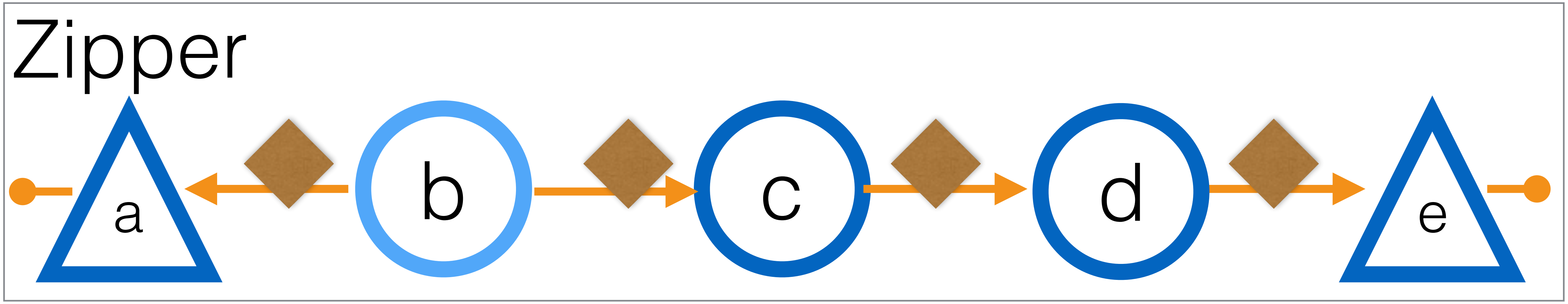
Because of the way randomness behaves, we get good balance at scale

Choose a random level based on balanced tree height distribution

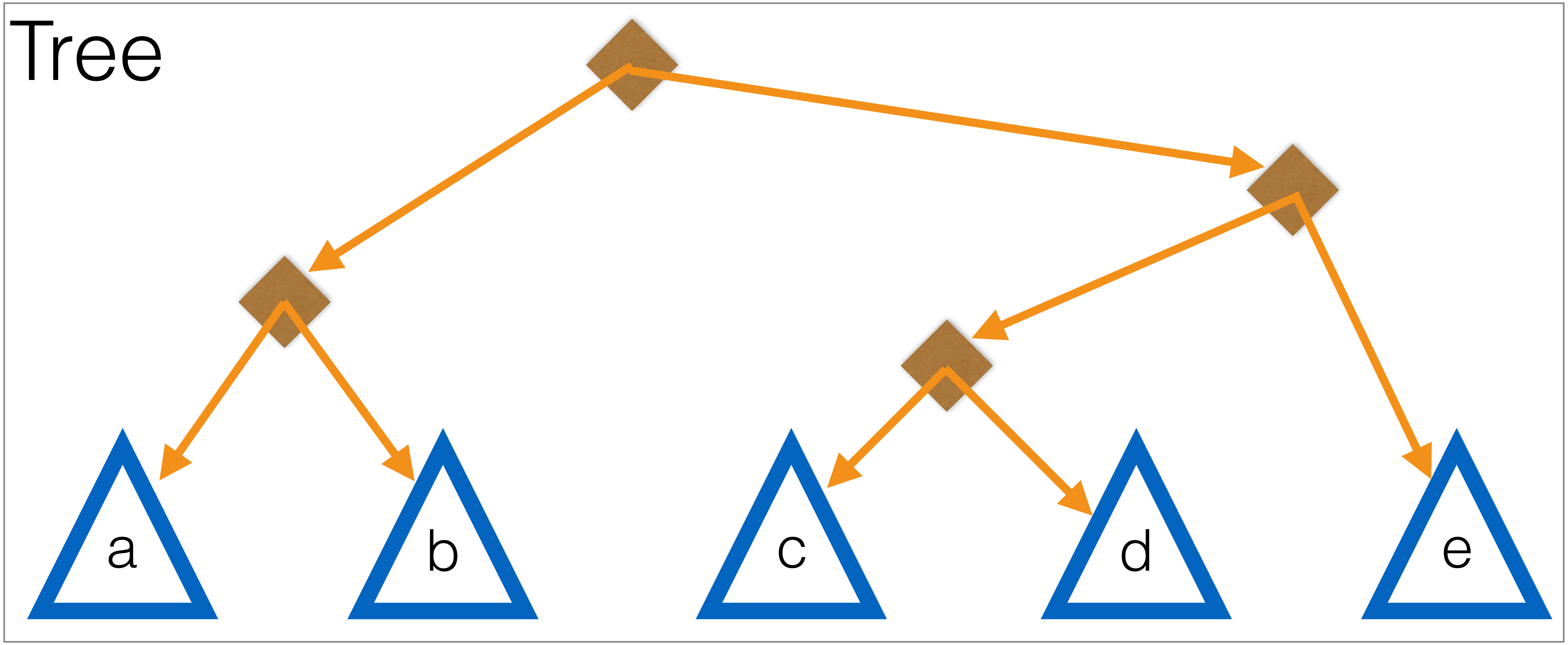


Two Forms of RAZ

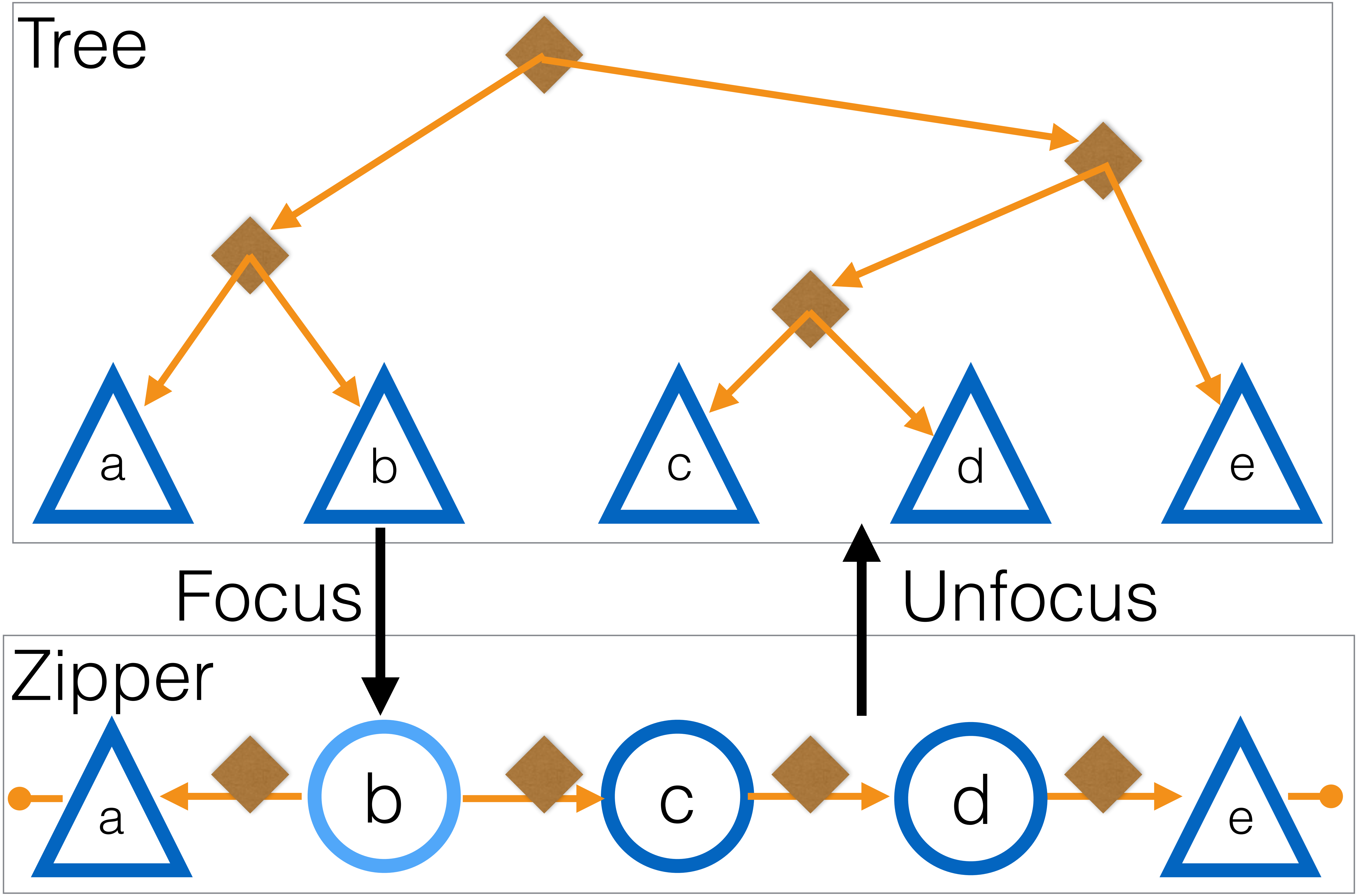
Two Forms of RAZ



Two Forms of RAZ



Two Forms of RAZ



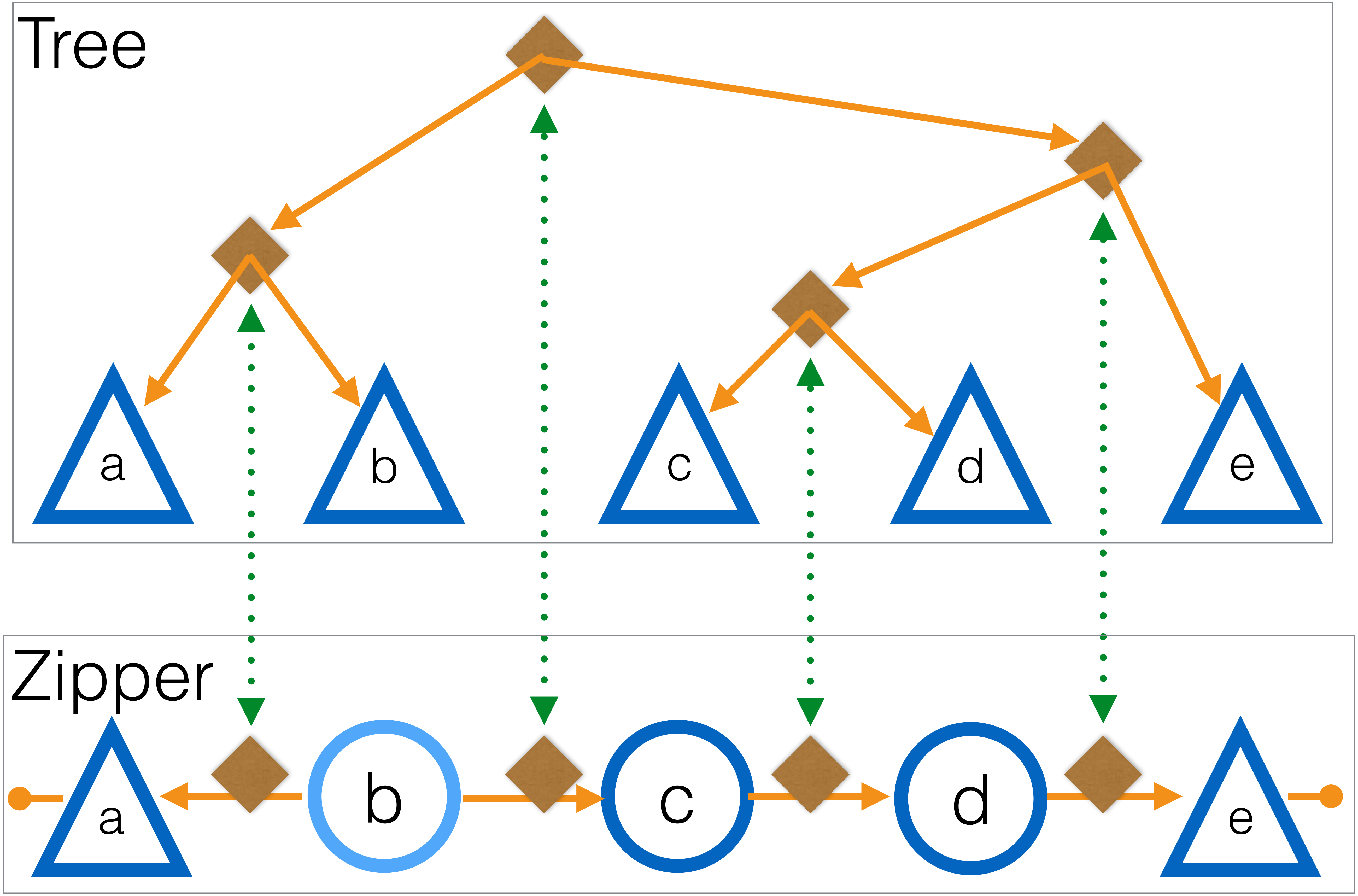
Tree

Focus

Unfocus

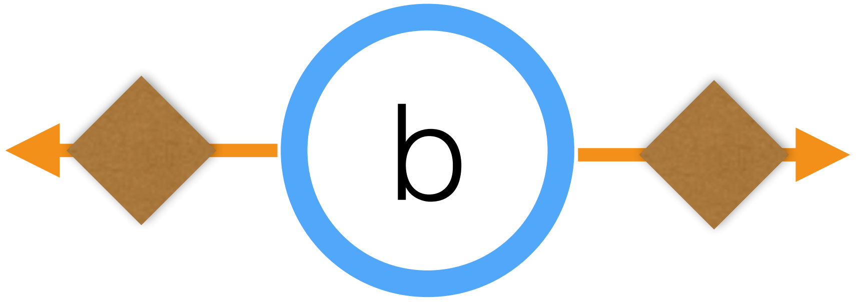
Zipper

Two Forms of RAZ



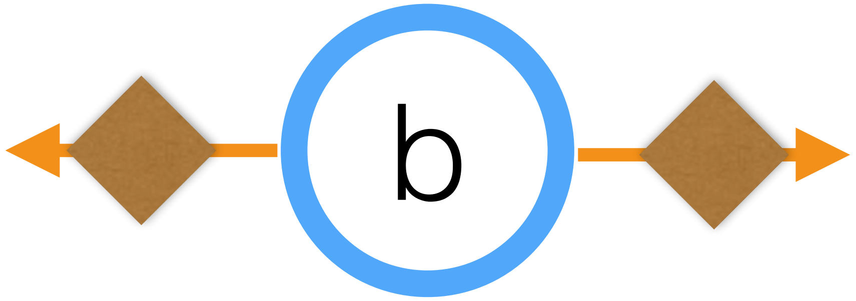
Invariants

Levels on each side of the focused element

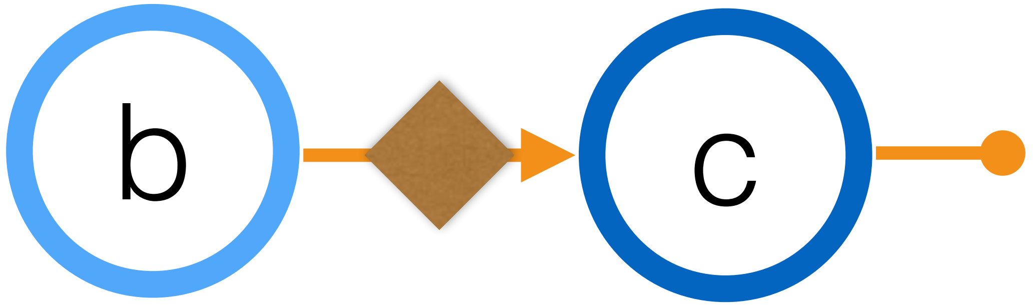


Invariants

Levels on each side of the focused element

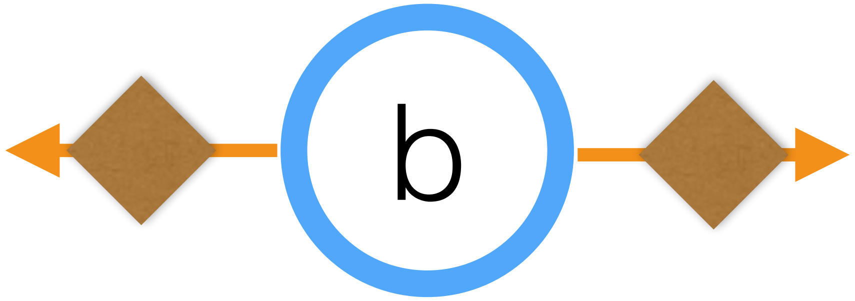


Levels between each element except Nil

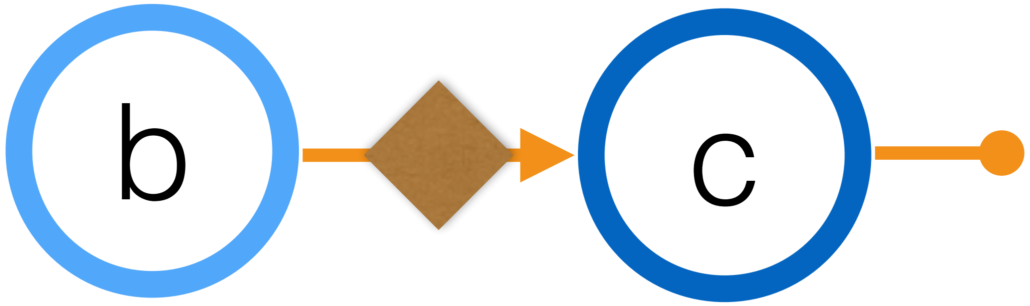


Invariants

Levels on each side of the focused element



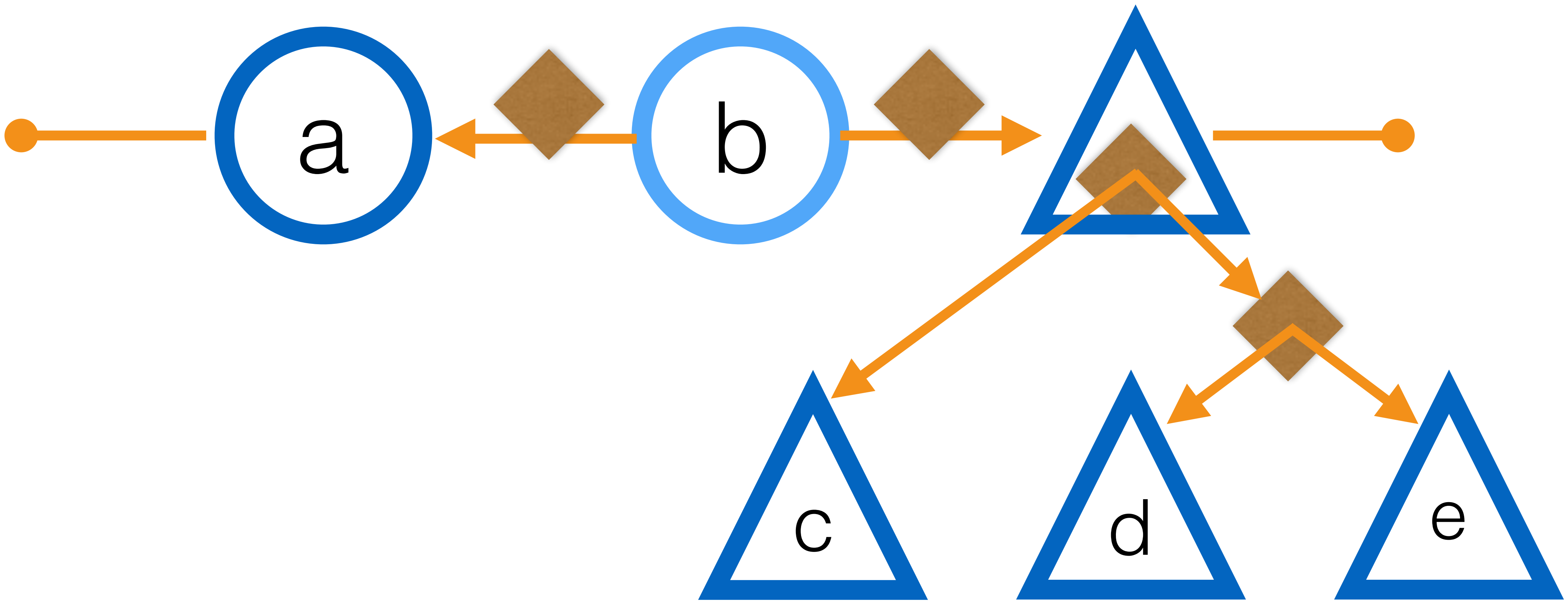
Levels between each element except Nil



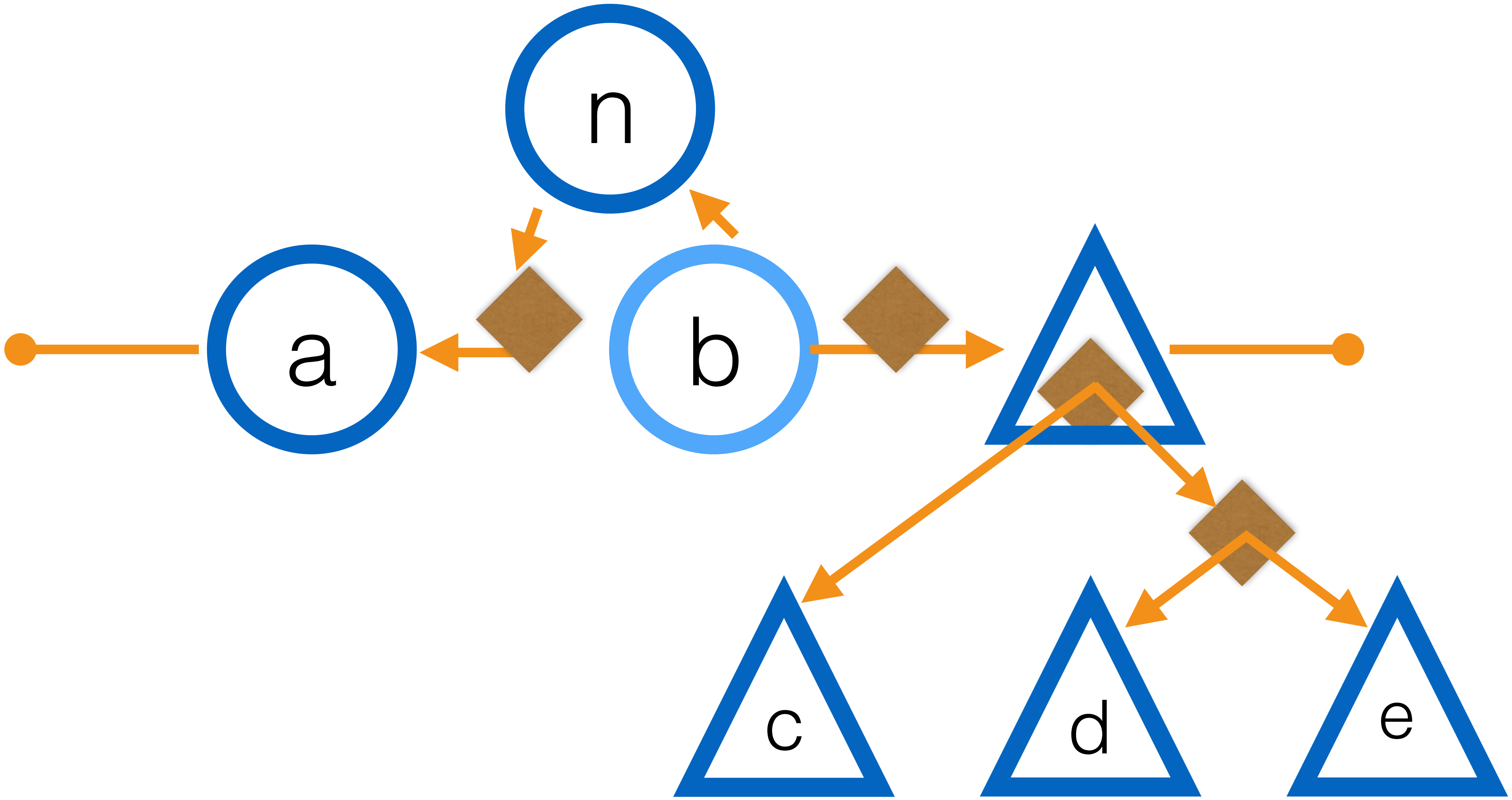
No Nil values in an unfocused RAZ

Closer look at our example

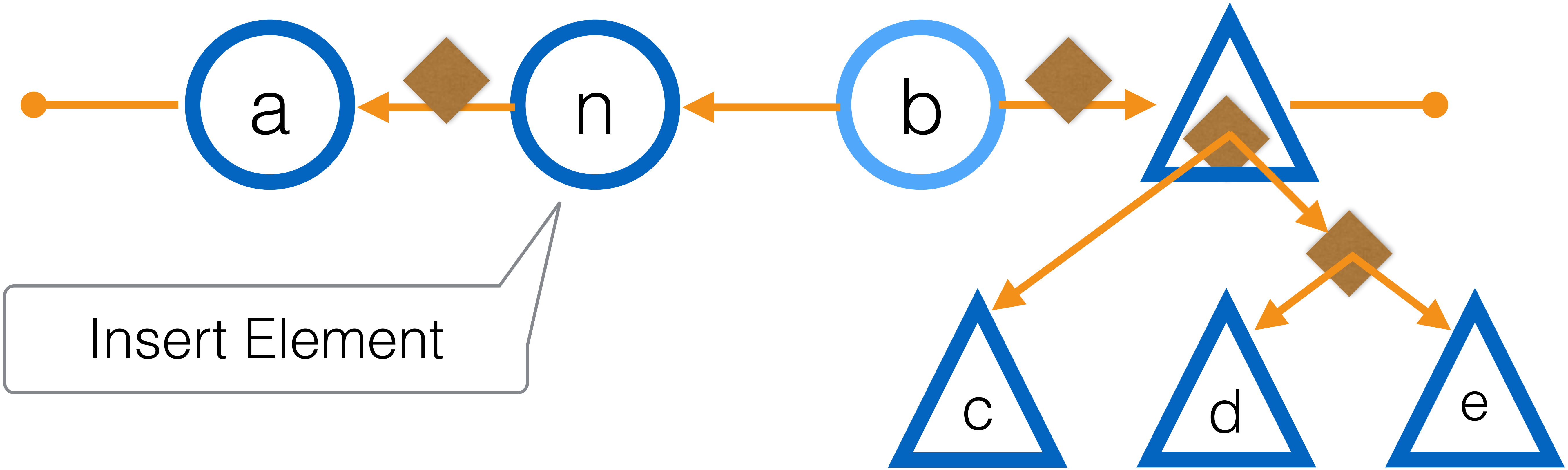
List-like Insertion



List-like Insertion

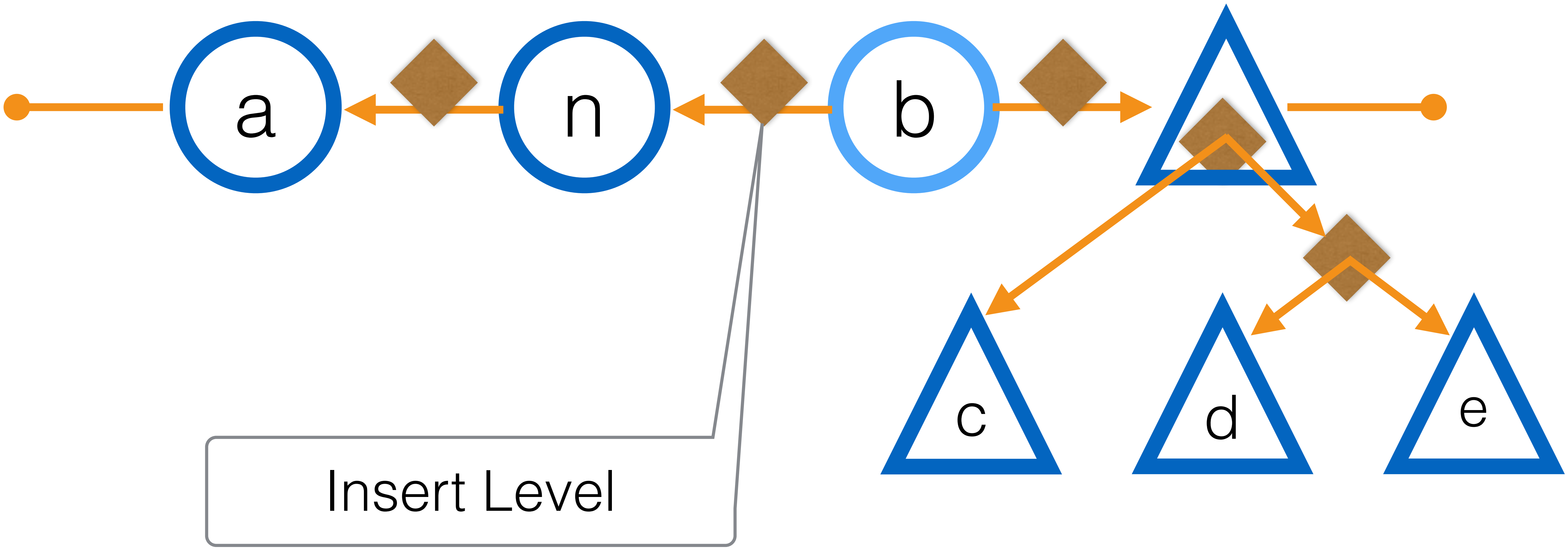


List-like Insertion

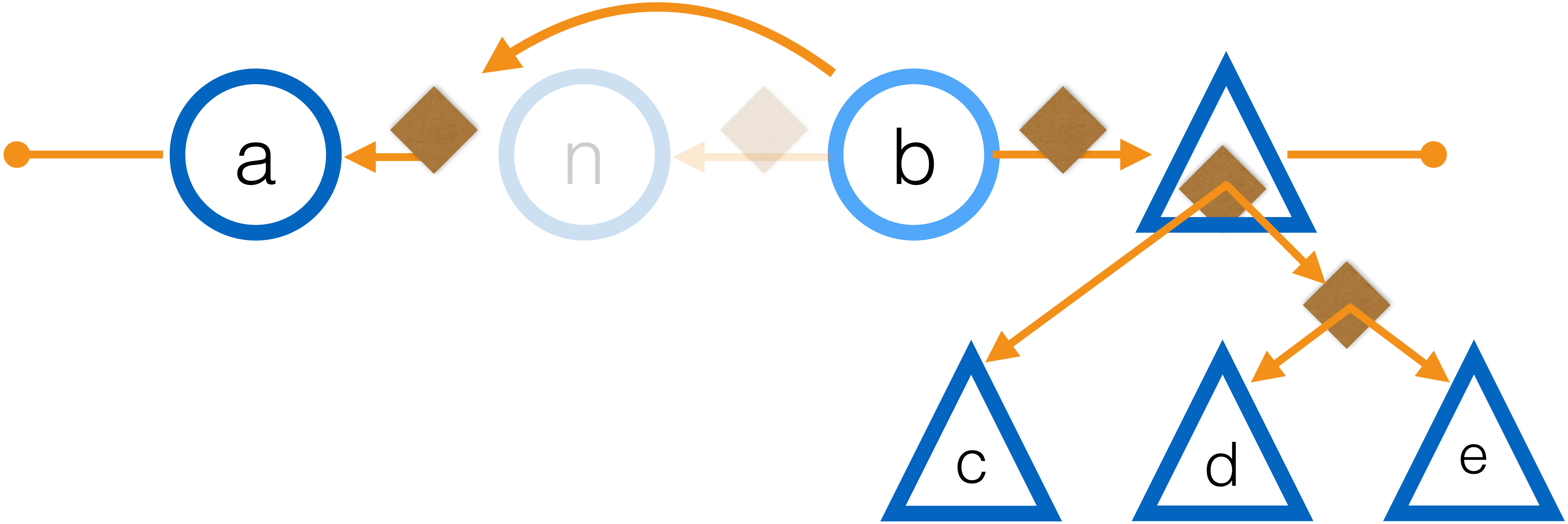


Insert Element

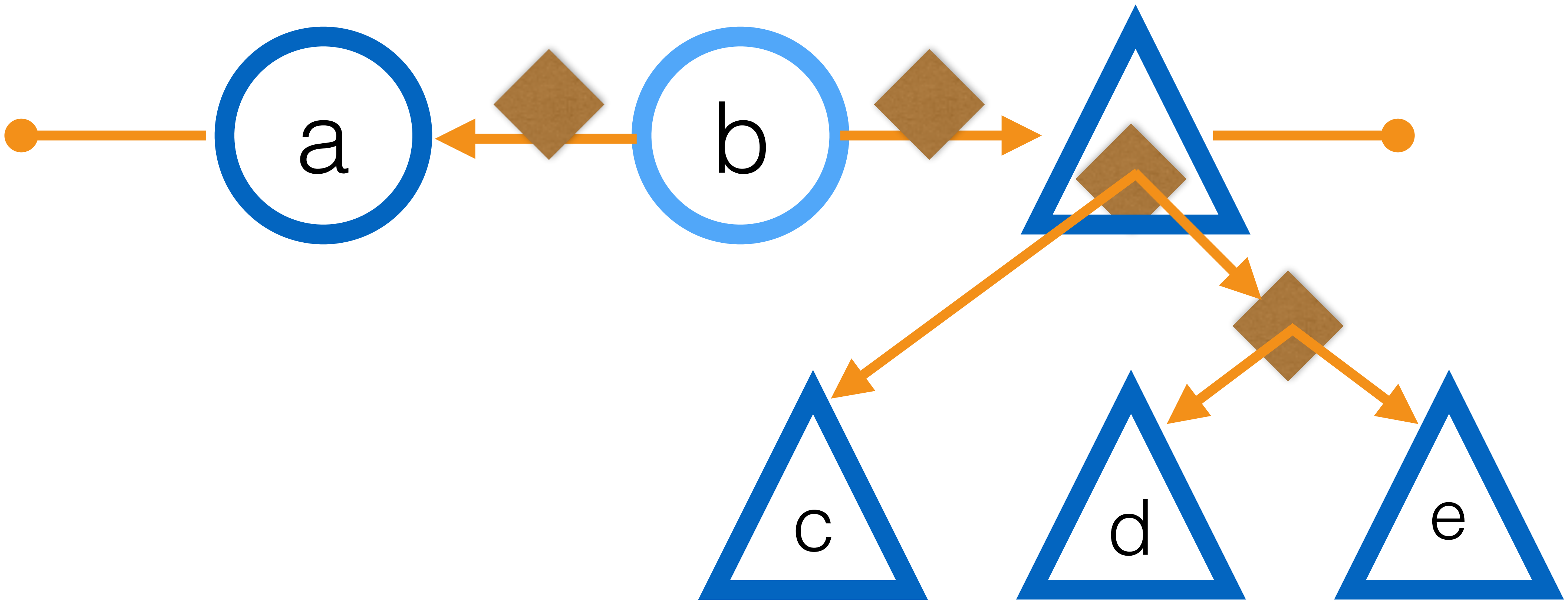
List-like Insertion



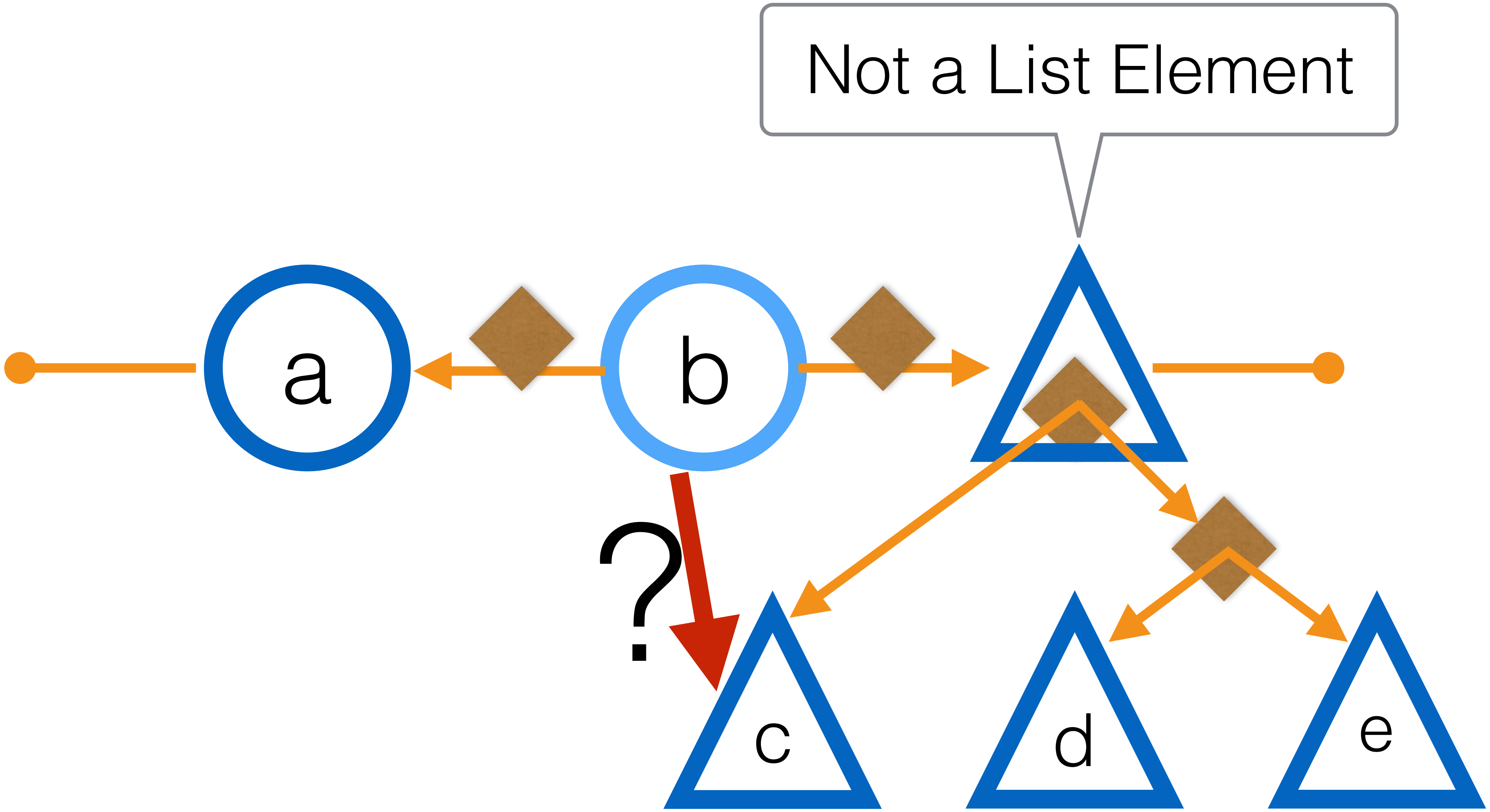
List-like Removal



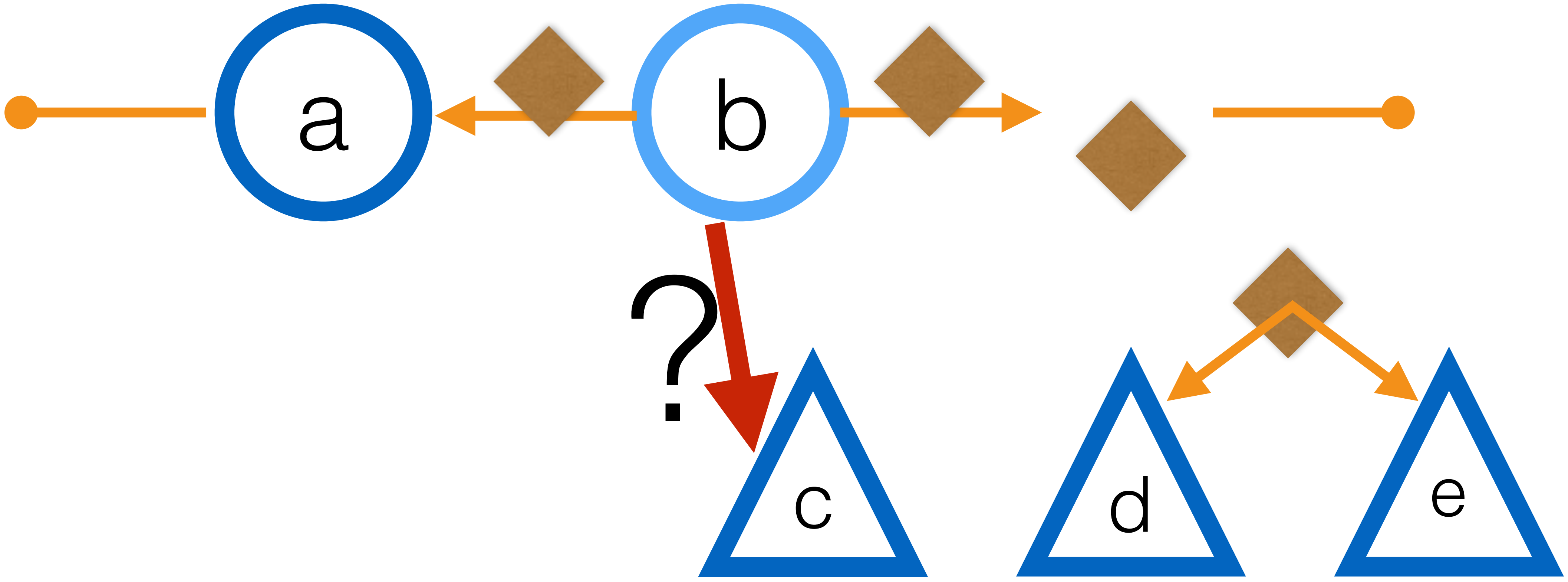
List-like Removal



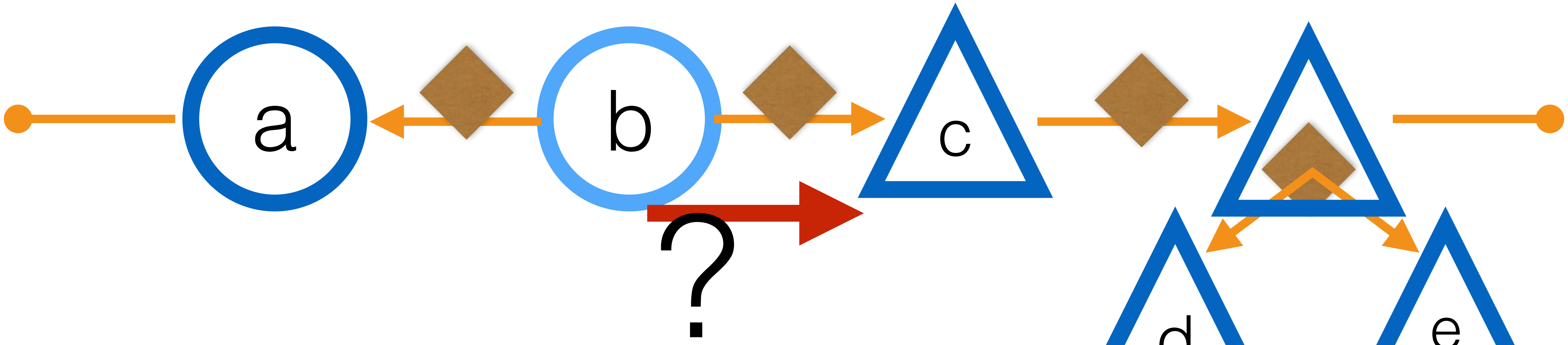
Trimmimg Trees


Not a List Element

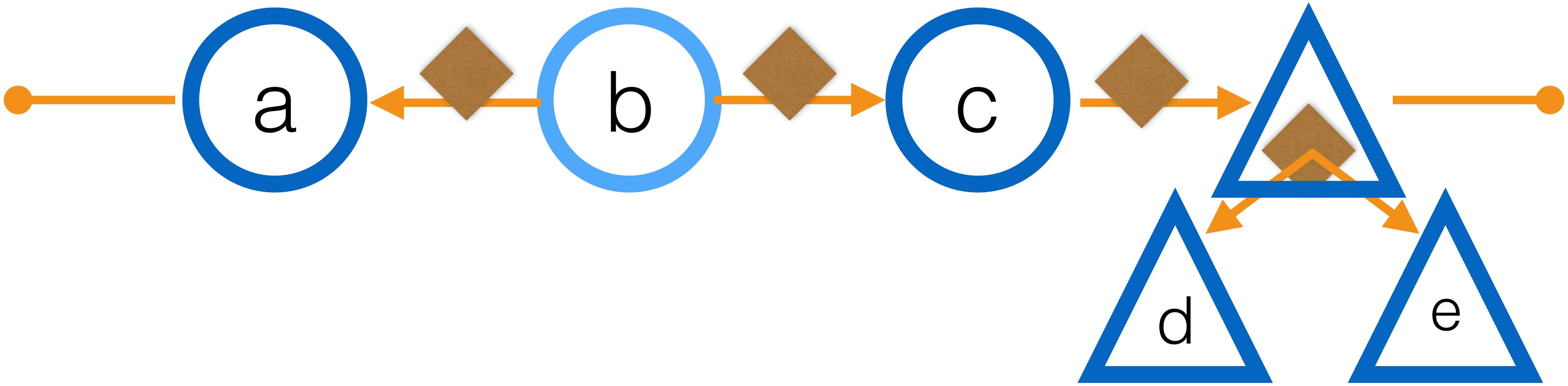
Trimmimg Trees



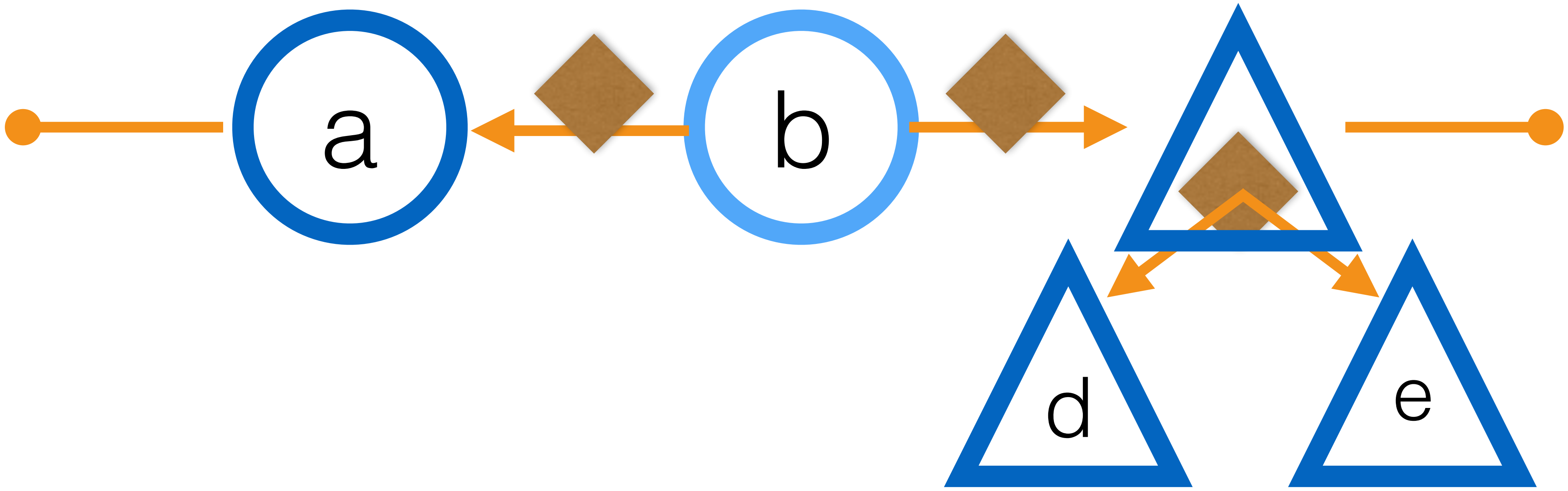
Trimmimg Trees



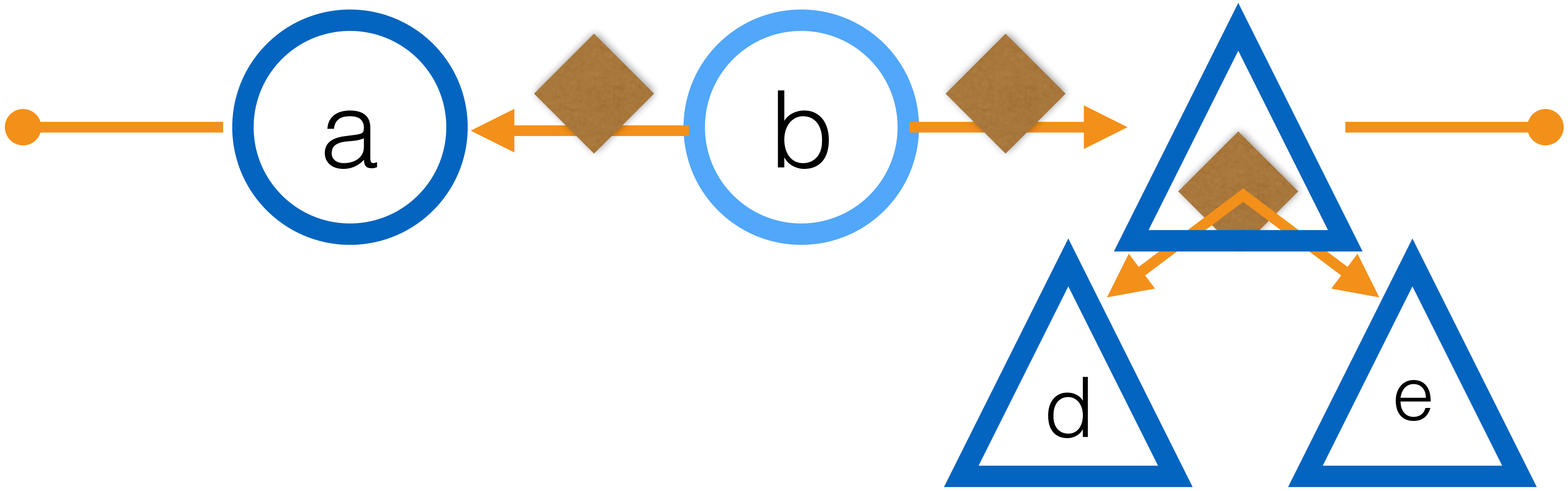
Trimmimg Trees



Trimmimg Trees

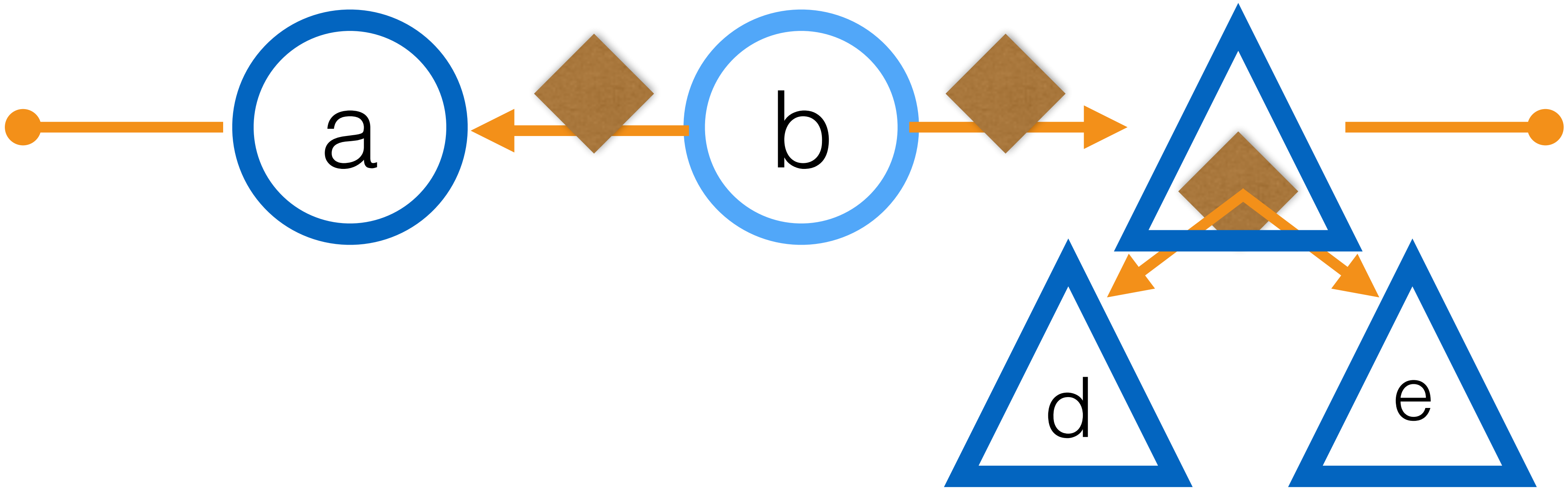


Unfocusing



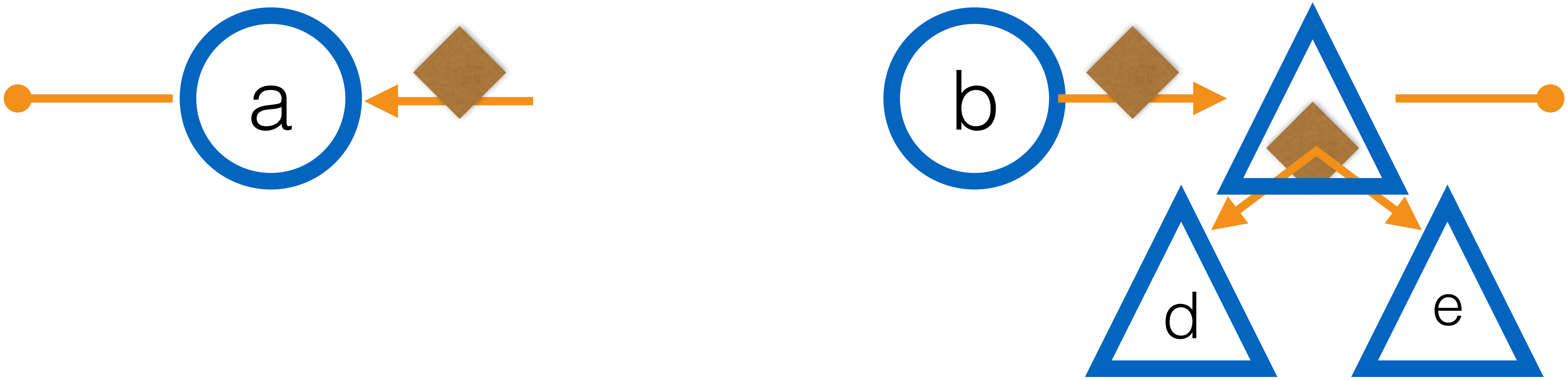
Unfocusing

In brief



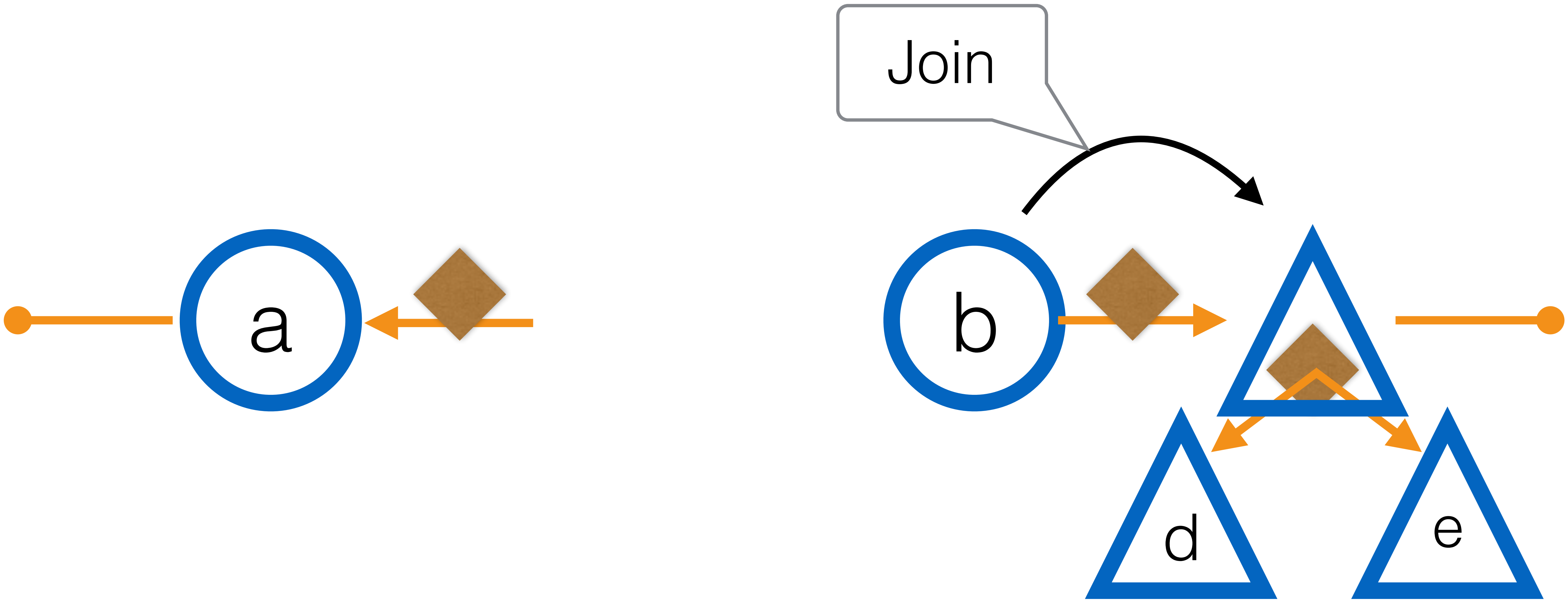
Unfocusing

In brief



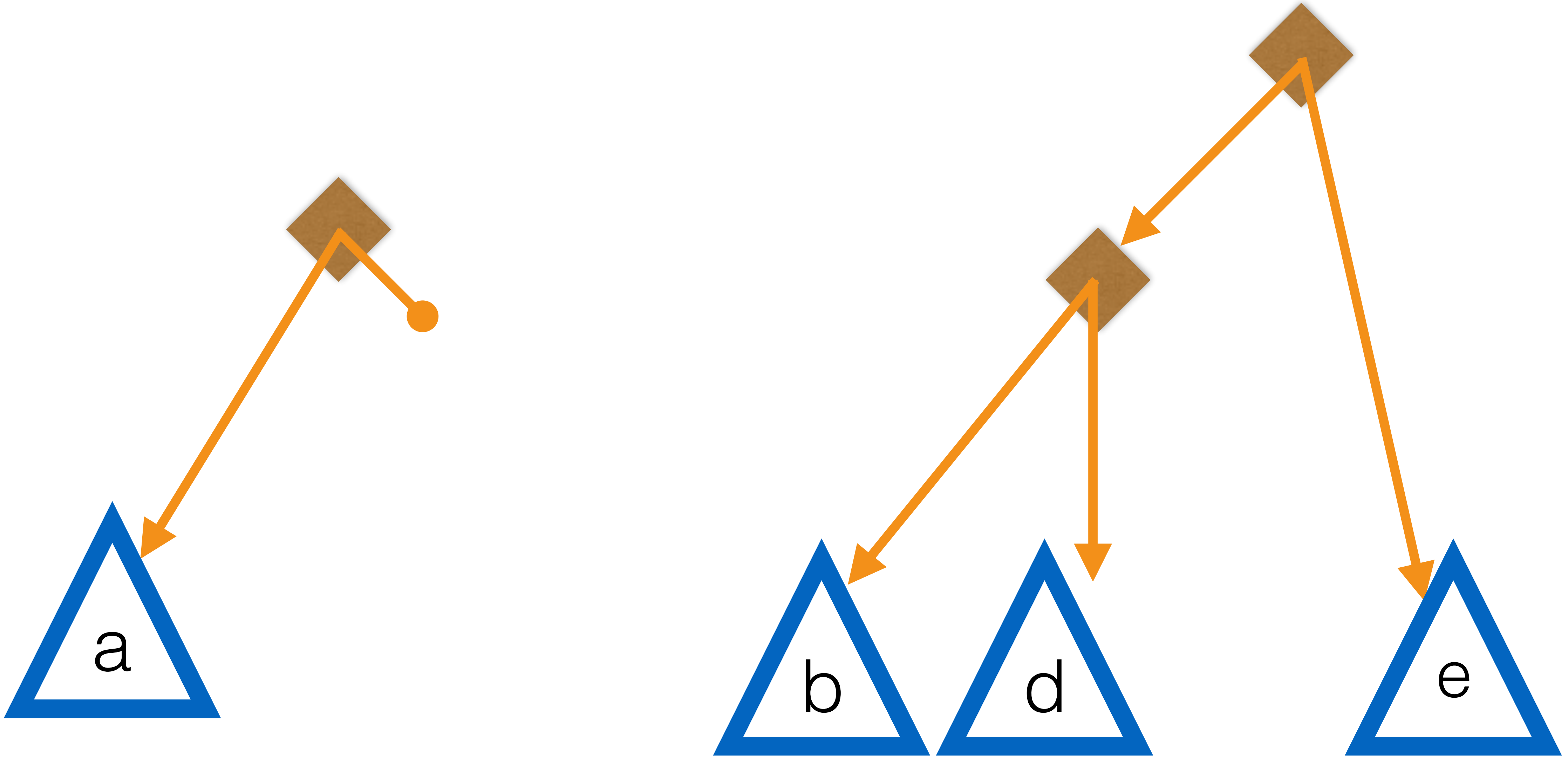
Unfocusing

In brief



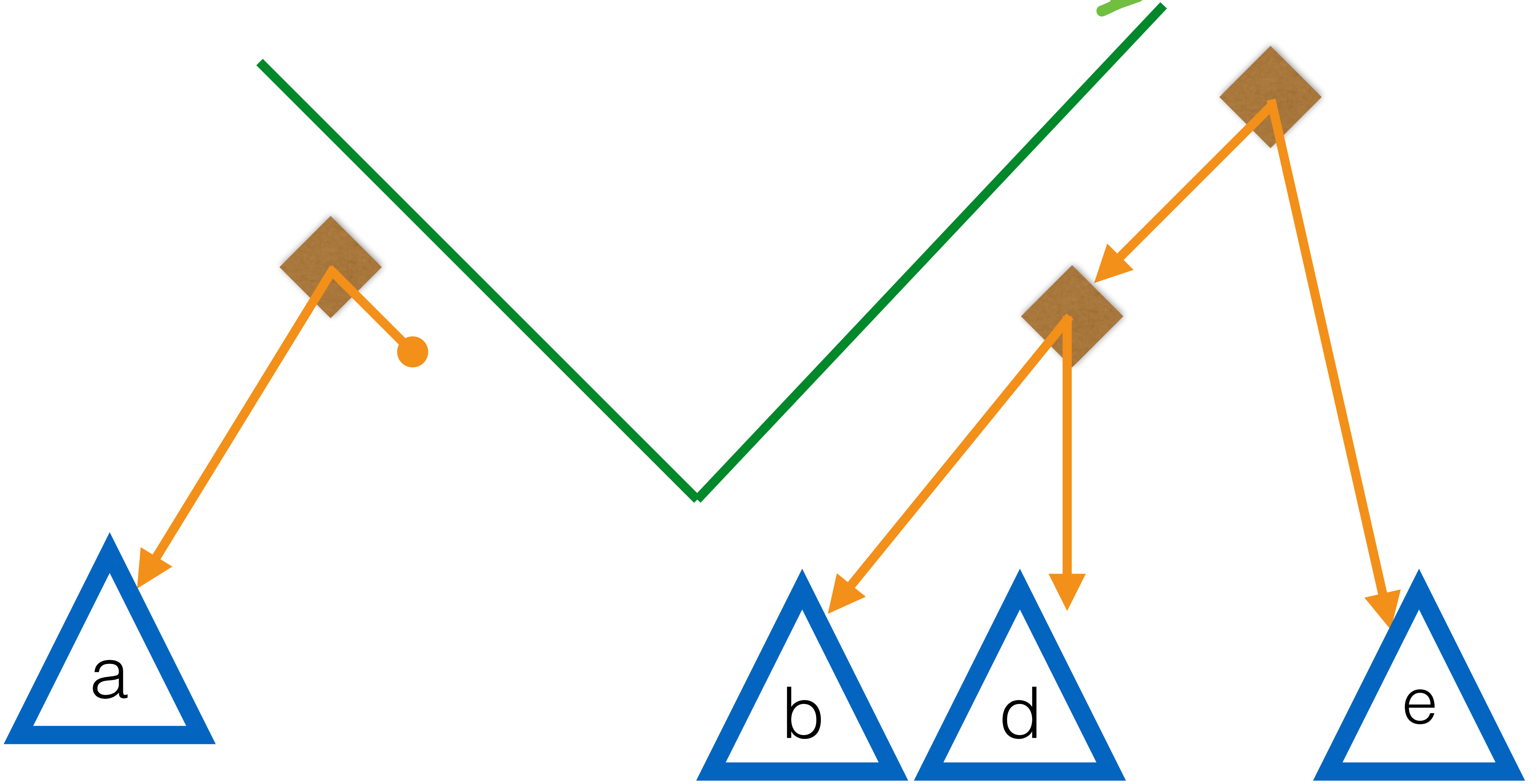
Unfocusing

In brief



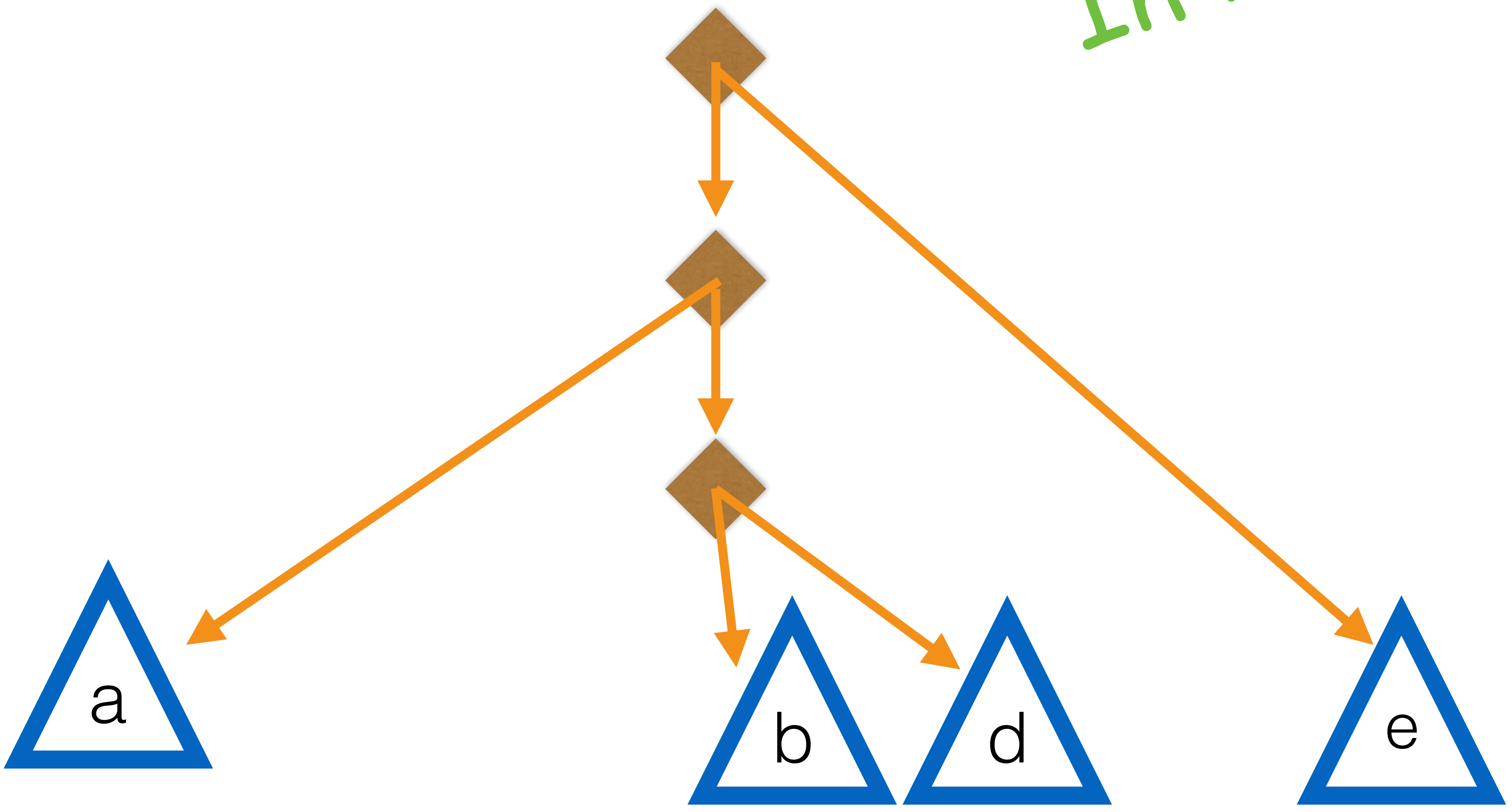
Unfocusing

In brief



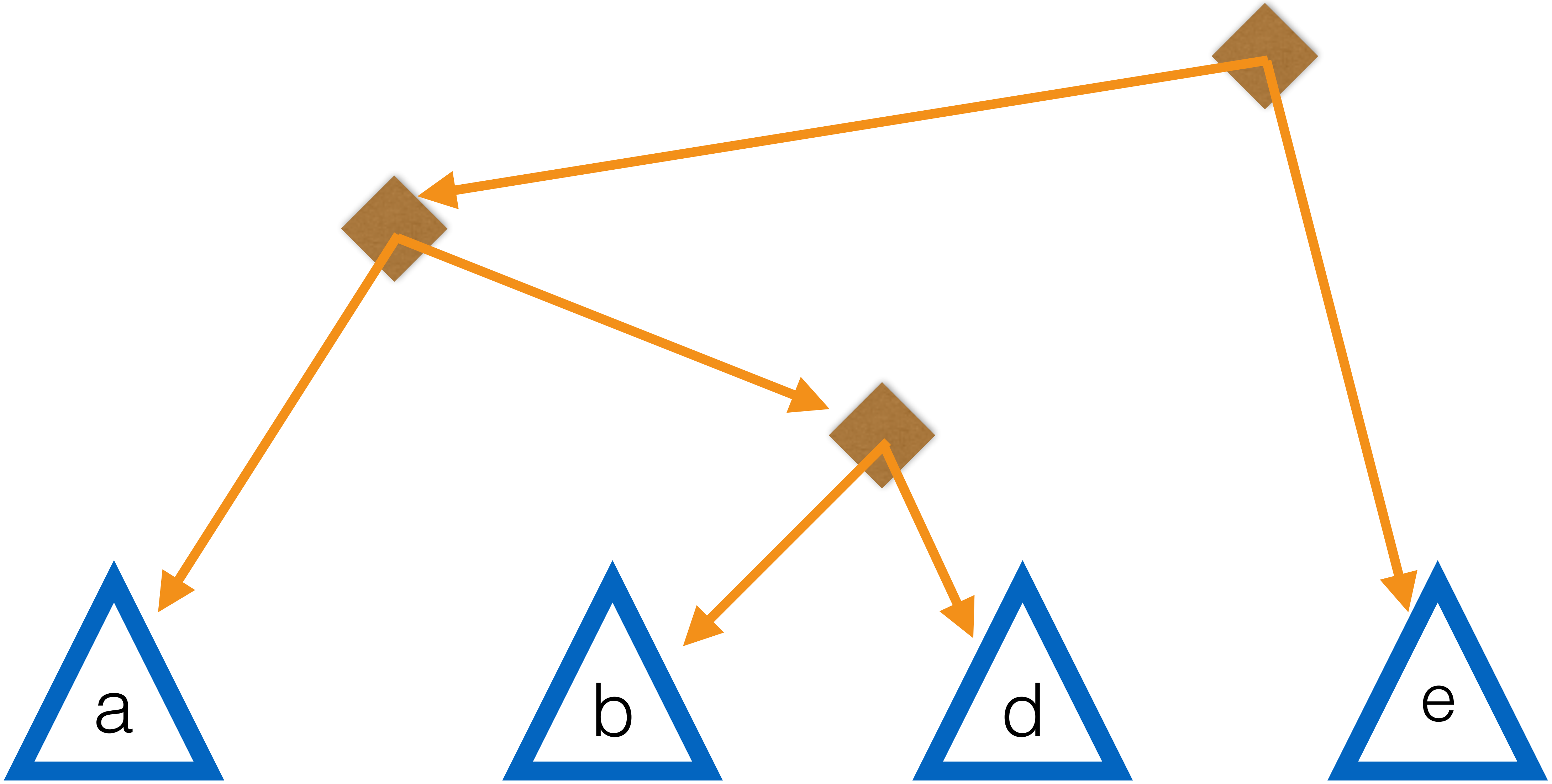
Unfocusing

In brief



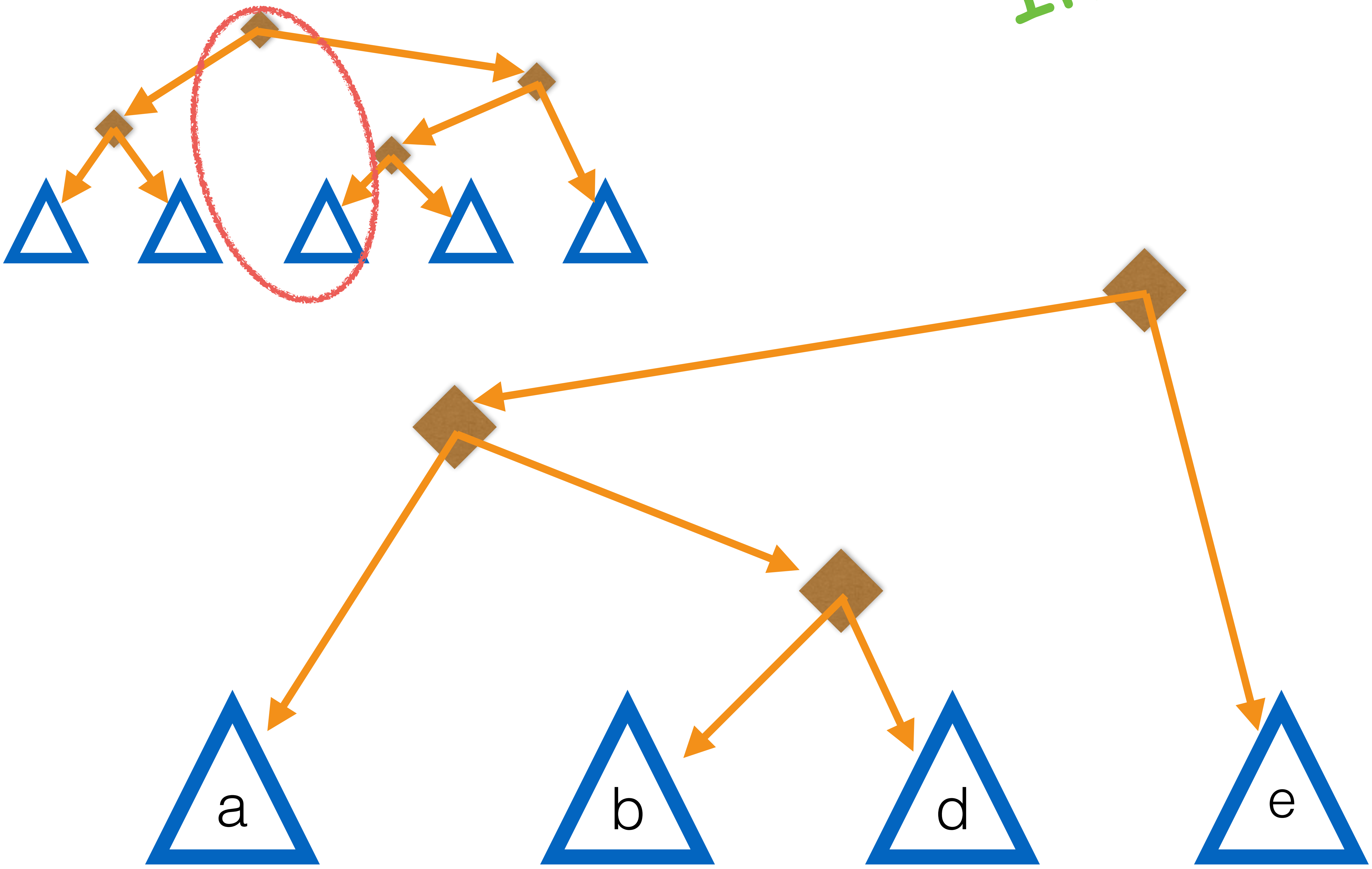
Unfocusing

In brief



Unfocusing

In brief



Switch to code

Focus on Focus

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith “focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
         (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
         (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

Focus on Focus

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith “focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
         (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
         (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

Recursive function
with accumulator

Focus on Focus

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith “focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
         (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
         (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

A single pattern match!

Focus on Focus

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith “focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
         (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
         (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

Return the accumulator as a RAZ focused on this element

Focus on Focus

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith “focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
         (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
         (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

Branch based on which side has the focus element

Focus on Focus

```
let focus pos tree =
  let rec focus = fun pos tree (l, r) ->
  match tree with
  | Nil -> failwith “focus: internal Nil"
  | Leaf(elm) ->
    assert (pos == 0);
    (l,elm,r)
  | Bin(lv, _, branch_l, branch_r) ->
    let cnt = item_count branch_l in
    if pos < cnt
    then
      focus pos branch_l
         (l, Level(lv, Tree(branch_r, r)))
    else
      let new_pos = (pos - cnt) in
      focus new_pos branch_r
         (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

On left, recurse on left branch with right branch in right accumulator

Focus on Focus

```
let focus pos tree =
 let rec focus = fun pos tree (l, r) ->
 match tree with
 | Nil -> failwith “focus: internal Nil"
 | Leaf(elm) ->
  assert (pos == 0);
  (l,elm,r)
 | Bin(lv, _, branch_l, branch_r) ->
  let cnt = item_count branch_l in
  if pos < cnt
  then
   focus pos branch_l
     (l, Level(lv, Tree(branch_r, r)))
  else
   let new_pos = (pos - cnt) in
   focus new_pos branch_r
     (Level(lv, Tree(branch_l, l)), r)
in focus pos tree (Nil, Nil)
```

On right, recurse on right branch with left branch in left accumulator

Focus on Local Edits

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith “alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Focus on Local Edits

Local edits take directions

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith “alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Focus on Local Edits

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith “alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Two pattern matches

Focus on Local Edits

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith “alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Edit is similar for each side

Focus on Local Edits

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
 let rec alter new side zip = match zip with
 | Nil -> failwith “alter: past end of seq"
 | Cons(_,rest) -> Cons(new,rest)
 | Level(lv,rest) -> Level(lv,alter new side rest)
 | Tree _ -> alter new side (trim side zip)
 in fun side elm (l,e,r) -> match side with
 | L -> (alter elm L l,e,r)
 | R -> (l,e,alter elm R r)
```

Two common cases

Focus on Local Edits

We always reach a level first

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith “alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Focus on Local Edits

```
let alter : dir -> 'a -> 'a raz -> 'a raz =
  let rec alter new side zip = match zip with
  | Nil -> failwith “alter: past end of seq"
  | Cons(_,rest) -> Cons(new,rest)
  | Level(lv,rest) -> Level(lv,alter new side rest)
  | Tree _ -> alter new side (trim side zip)
  in fun side elm (l,e,r) -> match side with
  | L -> (alter elm L l,e,r)
  | R -> (l,e,alter elm R r)
```

Return new element

Experiments

Experiments

RAZ in OCaml

Experiments

RAZ in OCaml

Fingertree in OCaml

Experiments

RAZ in OCaml

Insertion and removal at random point

Fingertree in OCaml

Experiments

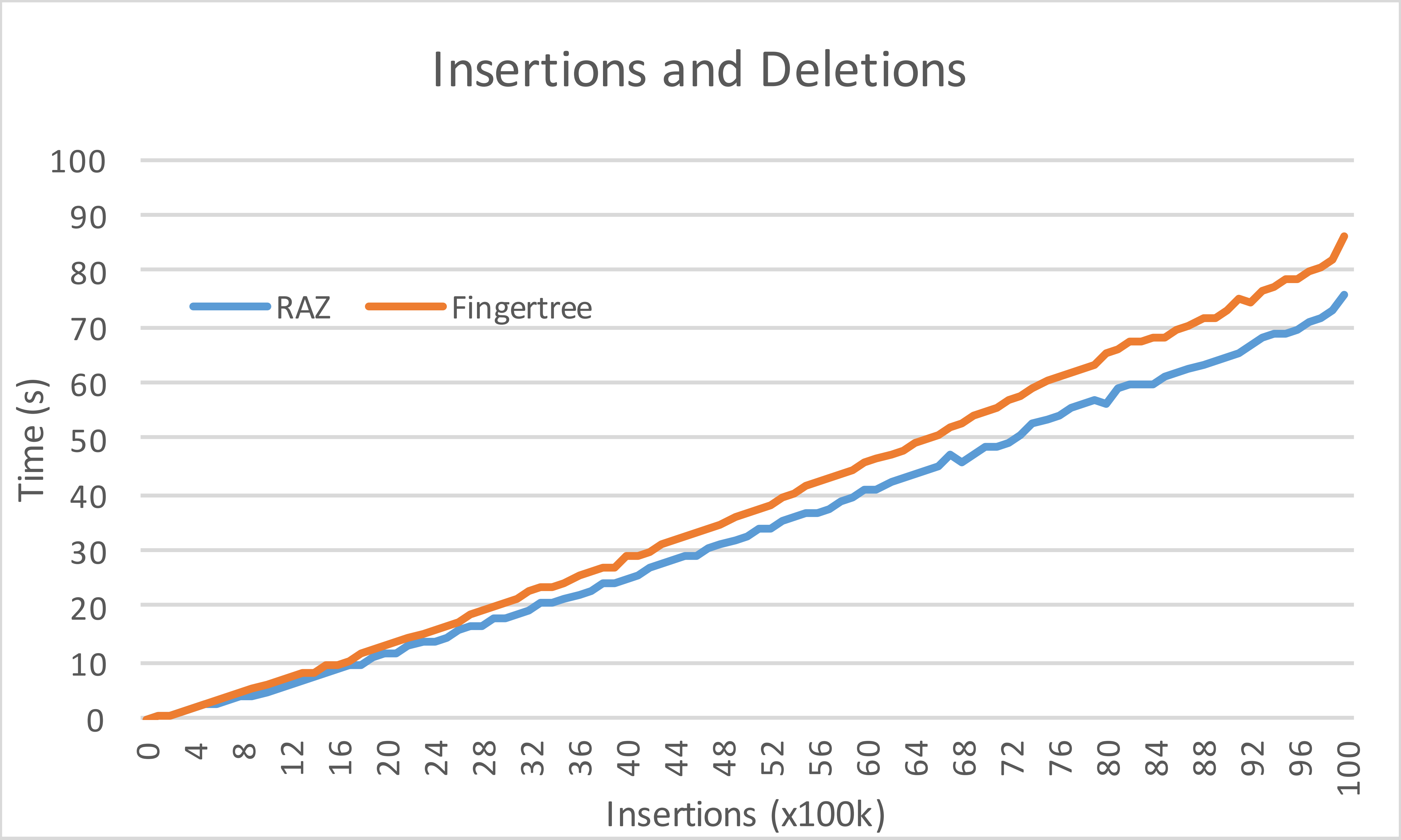
RAZ in OCaml

Fingertree in OCaml

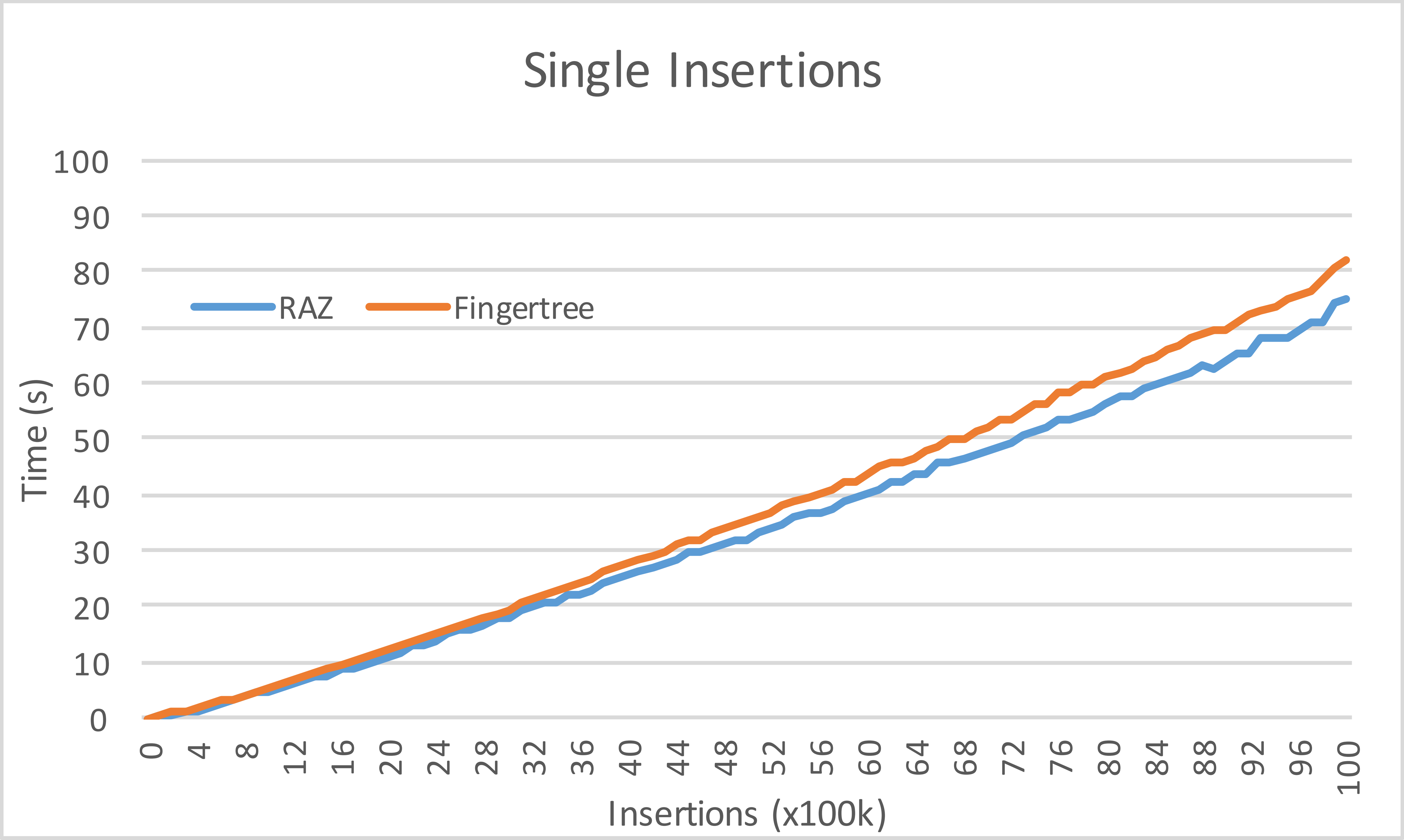
Insertion and removal at random point

Insertion at random point

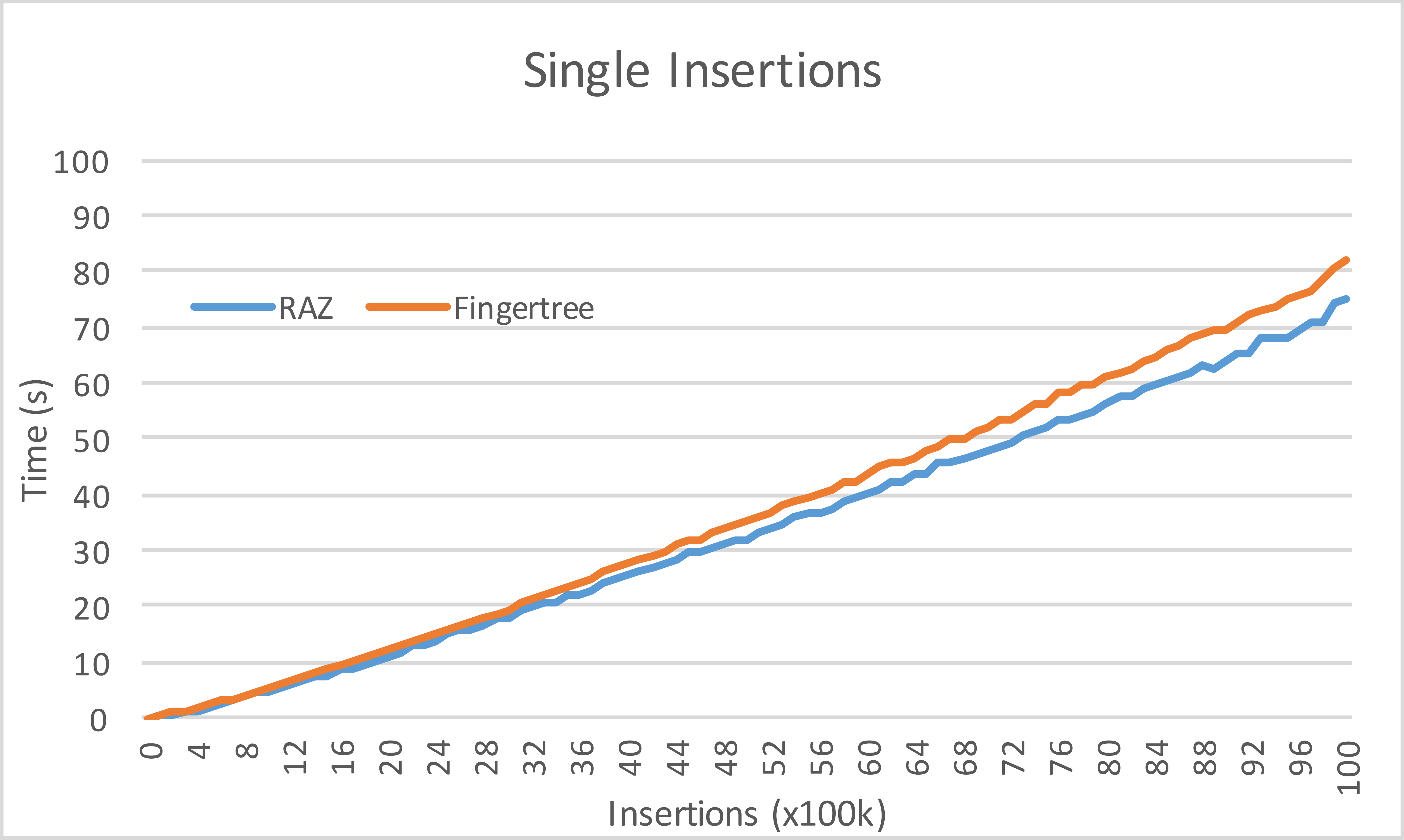
Insertion, Removal



Insertion at random



Insertion at random



Simplicity as performance?

Random Access Zipper

- Accessible
- Editable
- Simple
- Fast

Random Access Zipper

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