Persistent Data Structures & Finger Trees

Fun Club / Haskell User Group – Dec 13th, 2012
Frederic Kettelhoit

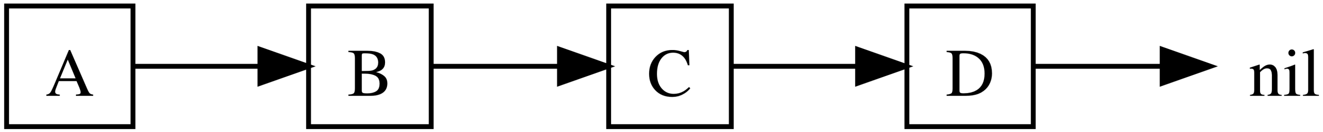
Functional Data Structures

- Functional languages emphasize **immutablity**
  - We cannot just update data structures in place
  - Return a **new version** after every change
  - We could copy the whole data structure every time
    - Very inefficient

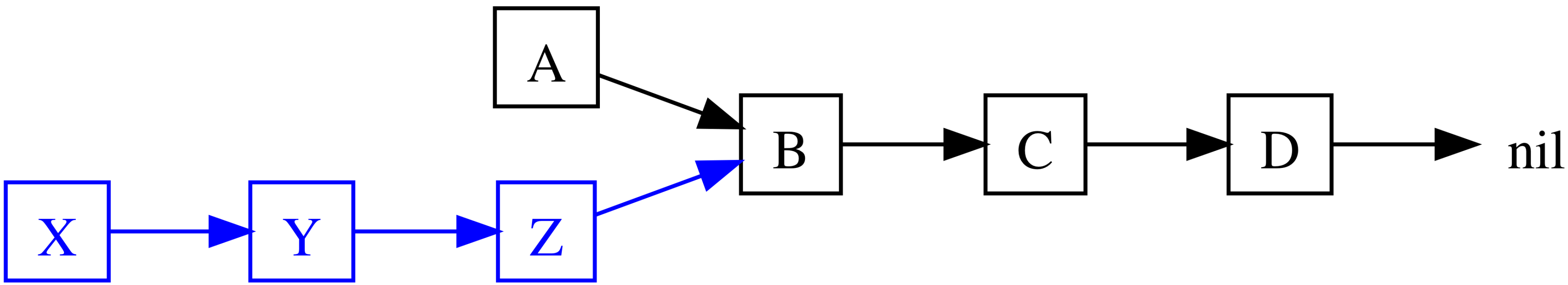
Persistent Data Structures

- When modified: preserve previous versions
  - immutable, always return **new versions**
- **Re-use** unchanged parts of the old version
- Example: (Singly) linked lists

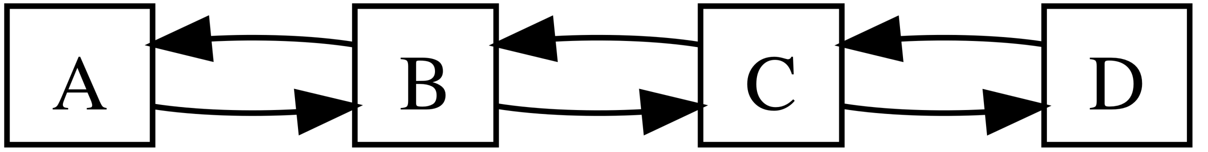
Linked Lists



Linked Lists

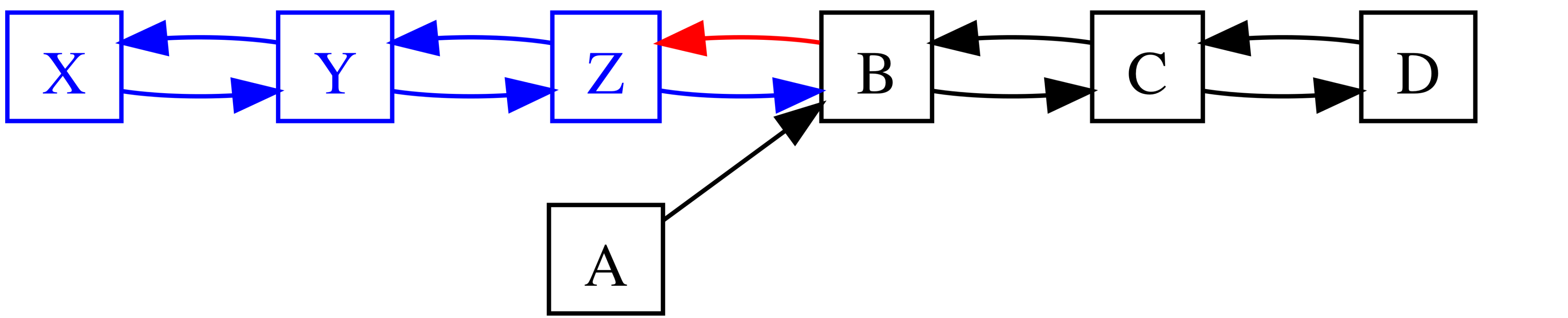


Doubly Linked List



Persistent?

Doubly Linked List



Not persistent!

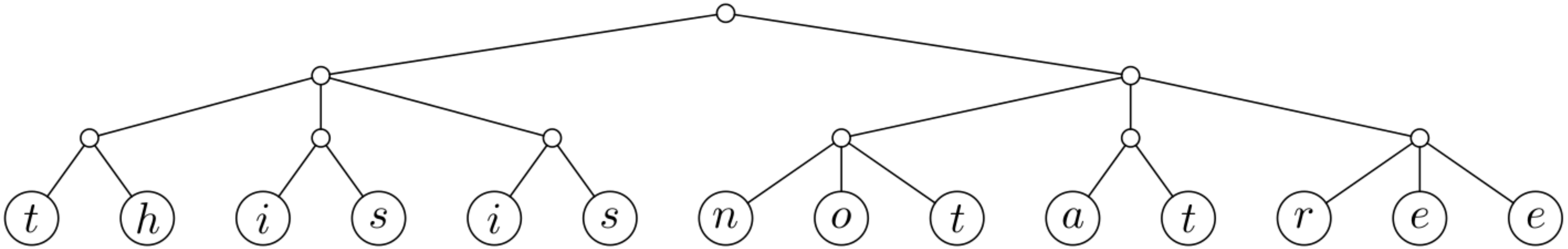
Finger Trees

- Many properties of doubly linked lists
  - cons/head/tail on both sides (deque)
- Customizable through monoids, can act as:
  - Random Access Sequence, Max-Priority Queue, Sorted Set, ...
- Good upper bounds for most operations
- Persistent

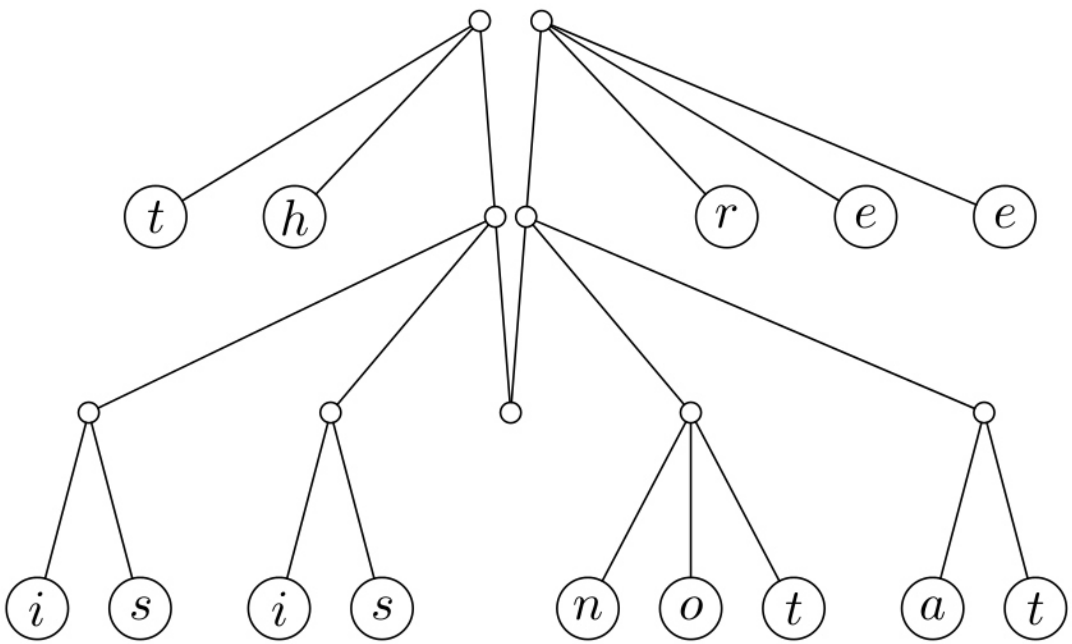
Upper Bounds

- head/tail in amortized time **O(1)**
  - left and right
- cons/snoc in amortized time **O(1)**
- concat in time **O(log (min m n))**
- split in time **O(log (min m n))**

2-3 Tree

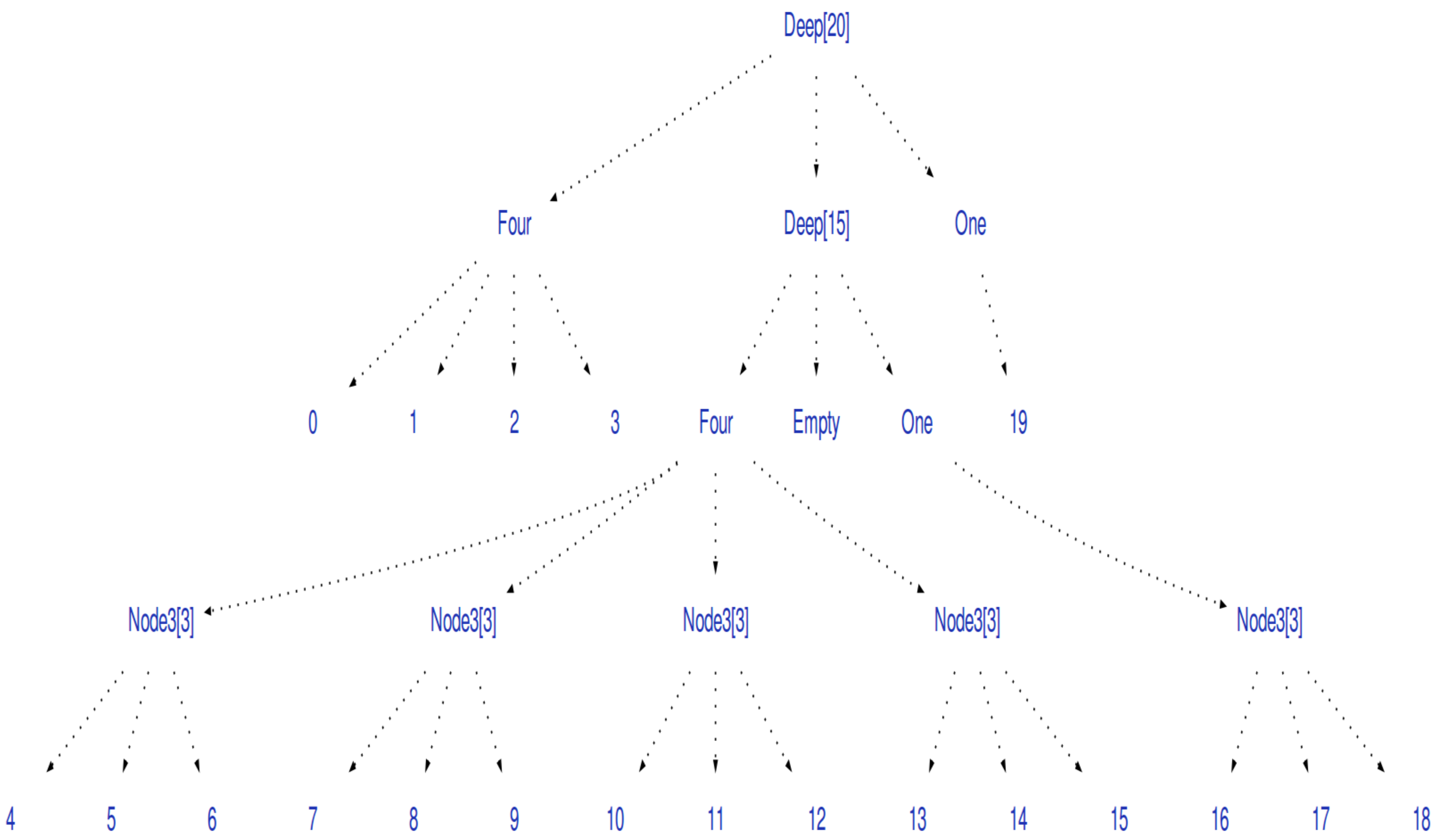


2-3 Finger Tree



Finger Trees in Haskell

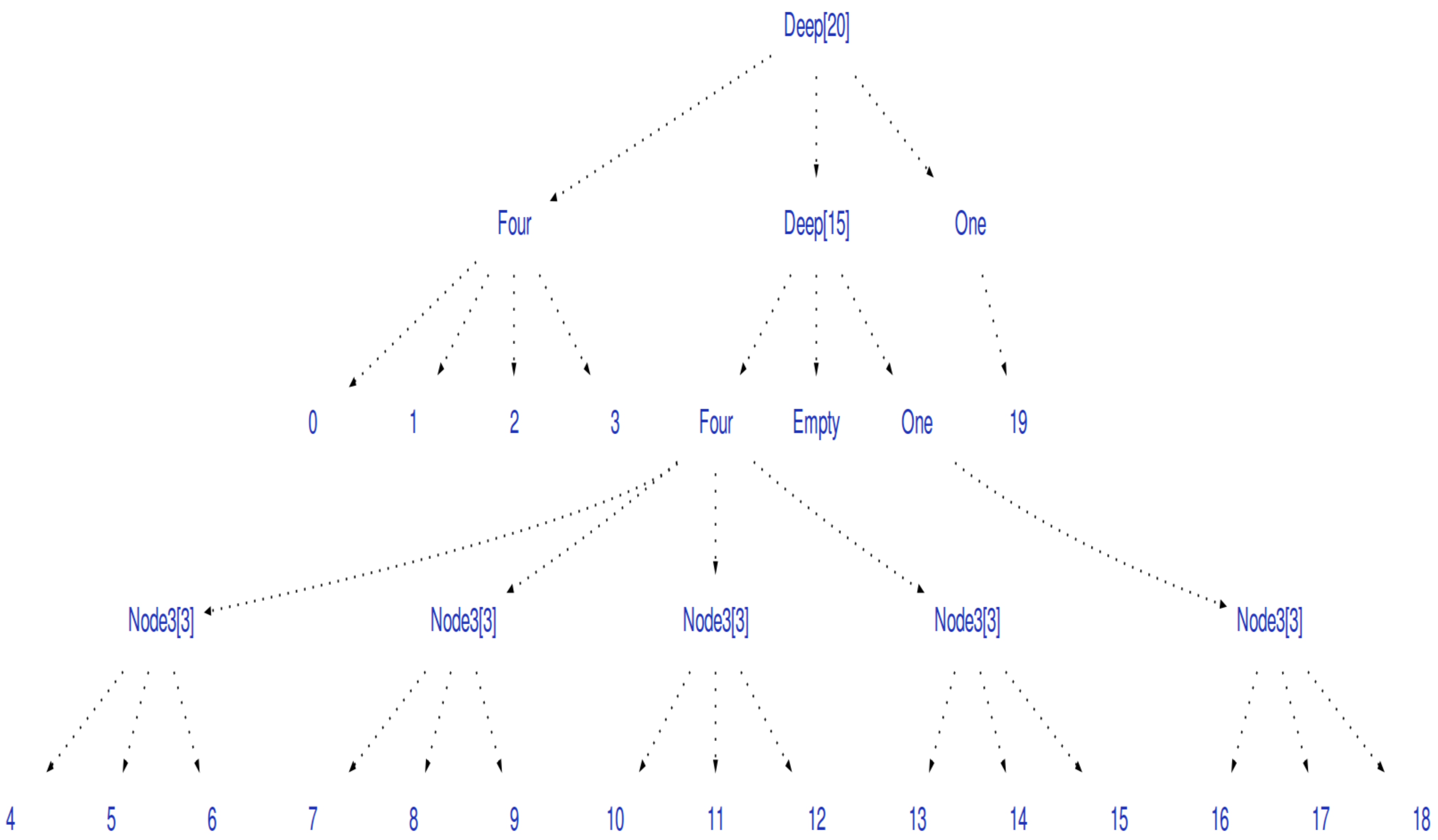
```
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)


type Digit a = One   a
             | Two   a a
             | Three a a a
             | Four  a a a a


data Node a = Node2 a a
            | Node3 a a a
```

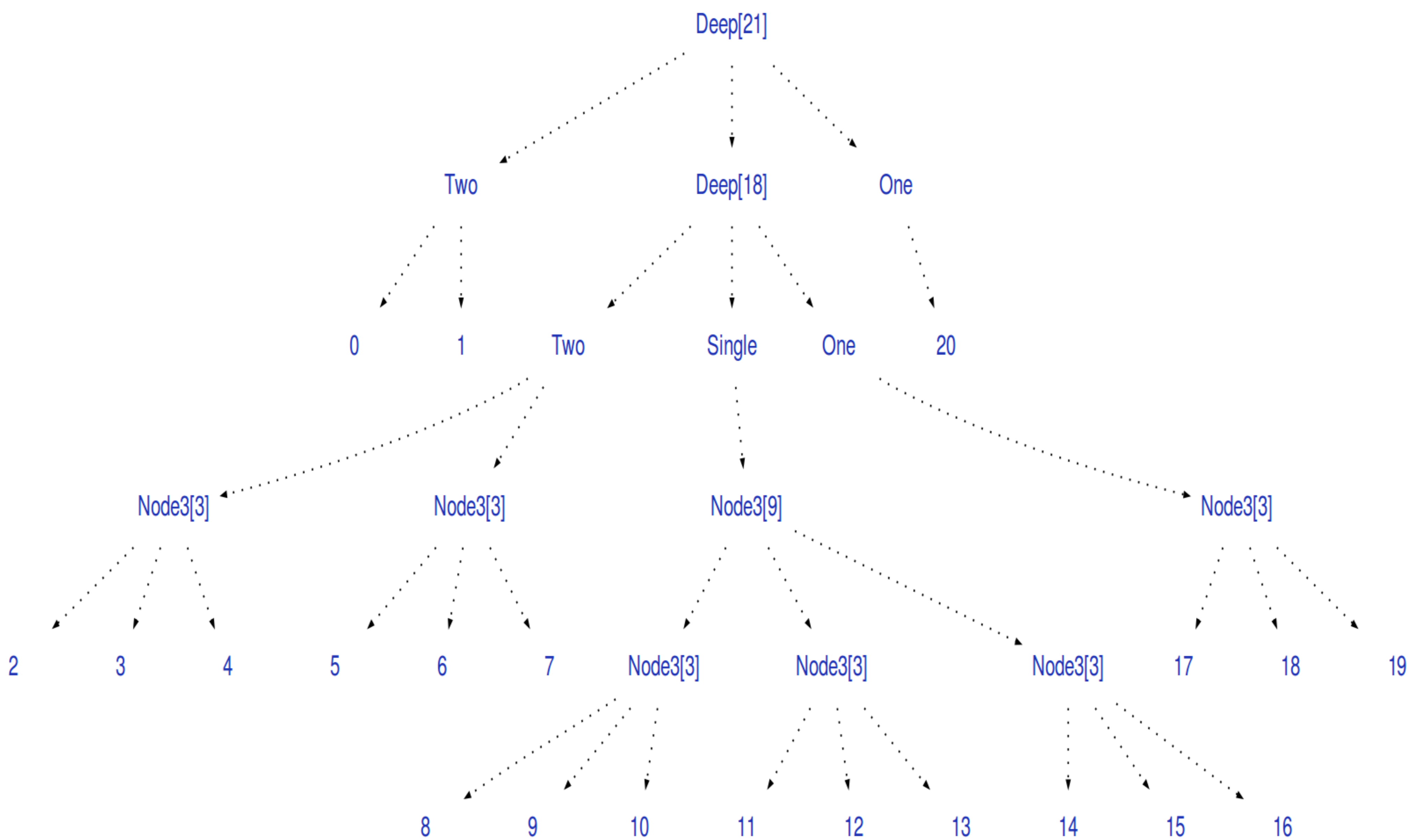

Deep[20]
Four
Deep[15]
One
0
1
2
3
Four
Empty
One
19
Node3[3]
Node3[3]
Node3[3]
Node3[3]
Node3[3]
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

cons

```
cons :: a -> FingerTree a -> FingerTree a
cons a Empty                          = Single a
cons a (Single b)                     = Deep (One a) Empty (One b)
cons a (Deep (One b) m sf)            = Deep (Two a b) m sf
cons a (Deep (Two b c) m sf)          = Deep (Three a b c) m sf
cons a (Deep (Three b c d) m sf)      = Deep (Four a b c d) m sf
cons a (Deep (Four b c d e) m sf)
  = Deep (Two a b) (cons (Node3 c d e) m) sf
```

- O(log n) worst case, O(1) amortized
- One, Two Three: Safe
- Four: Dangerous
- *(for the other side everything is simply swapped)*


Deep[20]
Four
Deep[15]
One
0
1
2
3
Four
Empty
One
19
Node3[3]
Node3[3]
Node3[3]
Node3[3]
Node3[3]
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18



Deep[21]
Two
Deep[18]
One
0
1
Two
Single
One
20
Node3[3]
Node3[3]
Node3[9]
Node3[3]
2
3
4
5
6
7
Node3[3]
Node3[3]
Node3[3]
17
18
19
8
9
10
11
12
13
14
15
16

concat

```
app3 :: FingerTree a -> [a] -> FingerTree a -> FingerTree a
app3 Empty ts xs      = cons' ts xs
app3 xs ts empty      = snoc' xs ts
app3 (Single x) ts xs = cons x (cons' ts xs)
app3 xs ts (Single x) = snoc (snoc' xs ts) x
app3 (Deep pr1 m1 sf1) ts (Deep pr2 m2 sf2)
     = Deep pr1 (app3 m1 (nodes sf1 ++ ts ++ pr2) m2) sf2

nodes :: [a] -> [Node a]
nodes [a, b] = [Node2 a b]
nodes [a, b, c] = [Node3 a b c]
nodes [a, b, c, d] = [Node2 a b, Node2 c d]
nodes (a : b : c : xs) = Node3 a b c : nodes xs

concat :: FingerTree a -> FingerTree a -> FingerTree a
concat xs ys = app3 xs [] ys
```

- concat in Zeit O(log (min m n))

Customization using Monoids

- A Monoid is a structure with 2 operations

```
mempty  :: a
mappend :: a -> a -> a
```

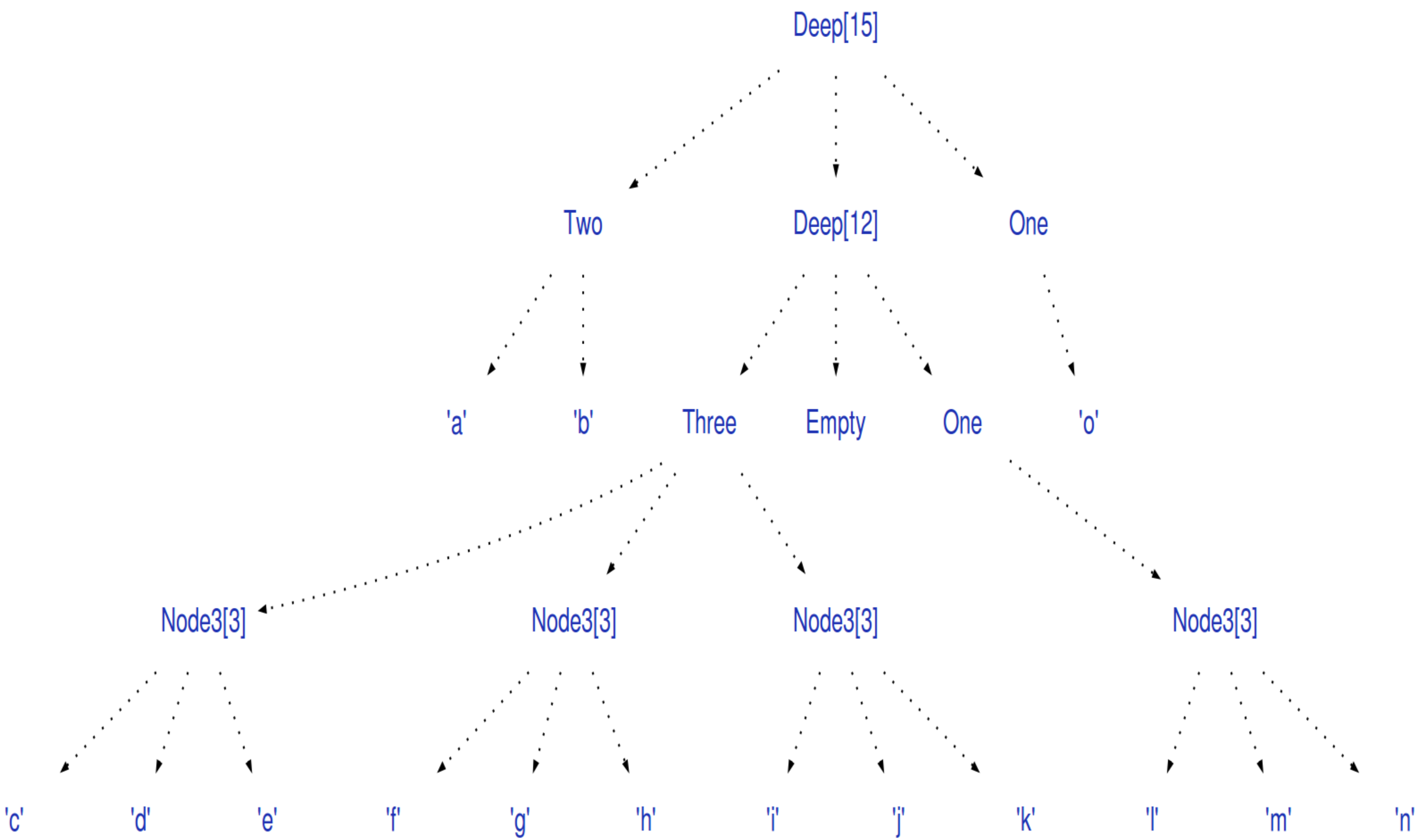
- Monoid Law:

```
mappend mempty x = x
mappend x mempty = x
mappend x (mappend y z) = mappend (mappend x y) z
```

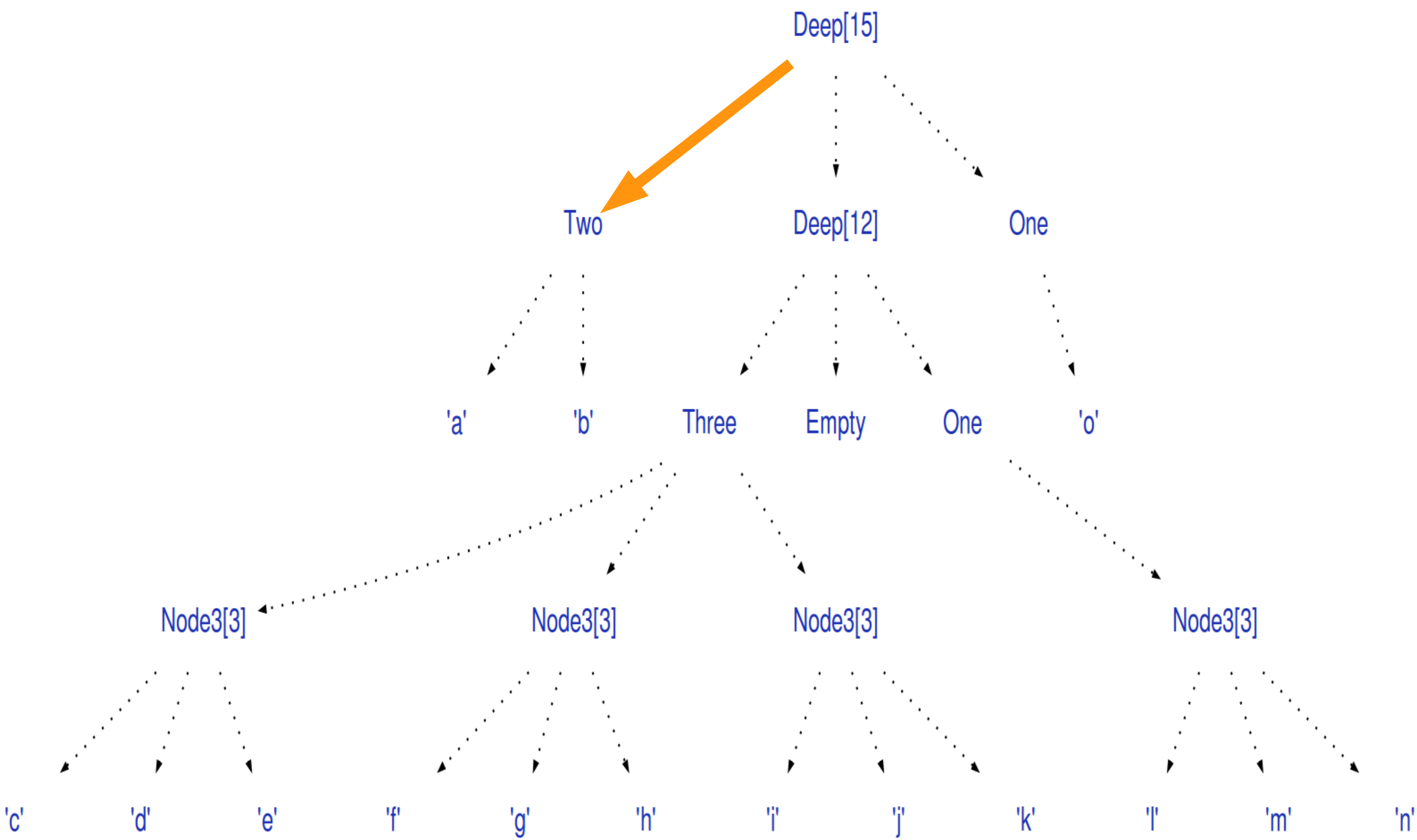
- And for the non-Category Theorists:
  - (Monoid = binary associative operation + identity)
  - For example: + and 0
- Finger Trees can be customized using Monoids

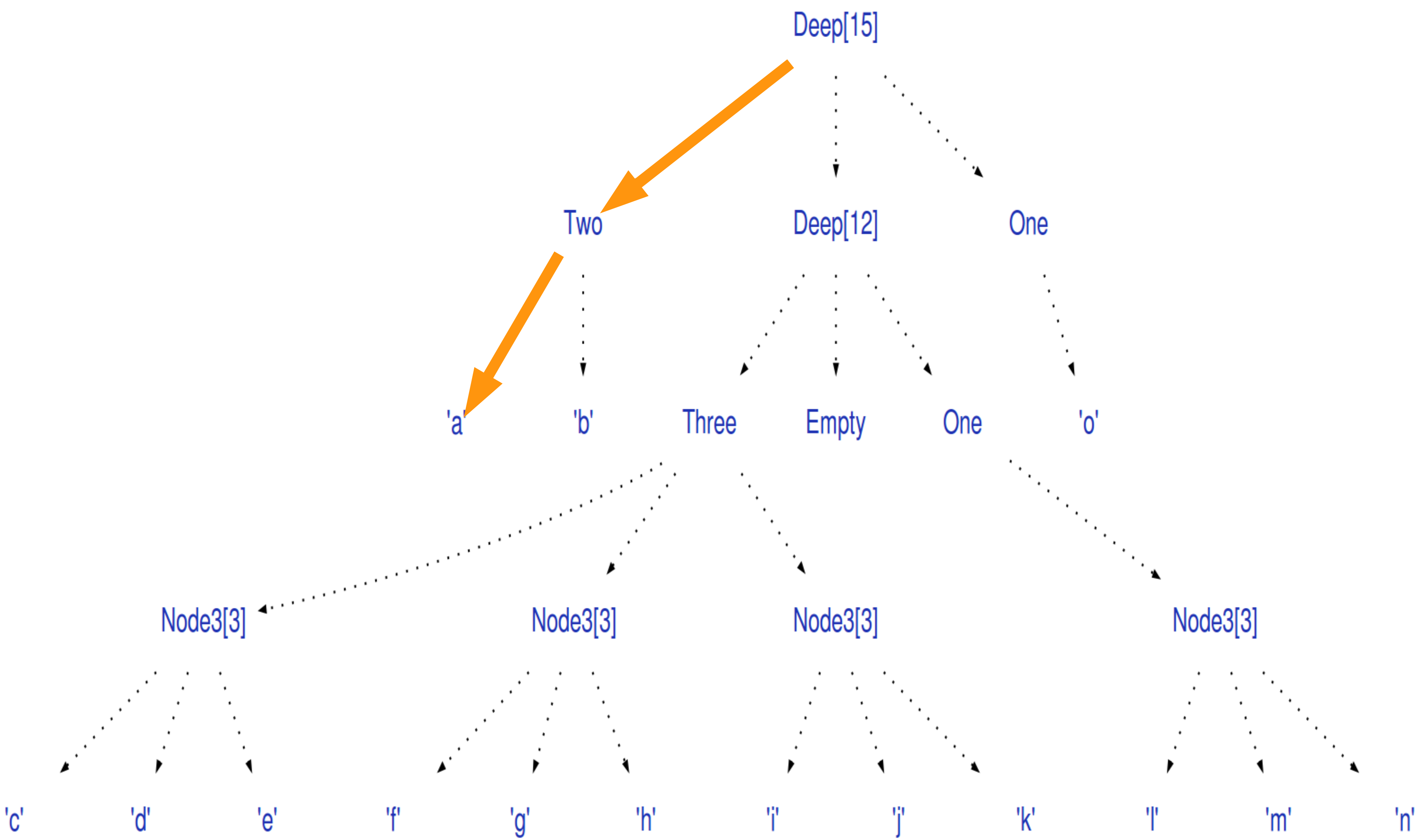
Customization using Monoids

- Every node contains an additional Monoid annotation with the value:
  - Empty = Identity of Monoid
  - Single a = ||a||
  - Deep pr m sf = mappend (mappend ||pr|| ||m||) ||sf||
- Example: Size Monoid
  - Empty = 0
  - Single a = ||a|| = (1 for primitive elements)
  - Deep pr m sf = ||pr|| + ||m|| + ||sf|| (number of elements in the tree)

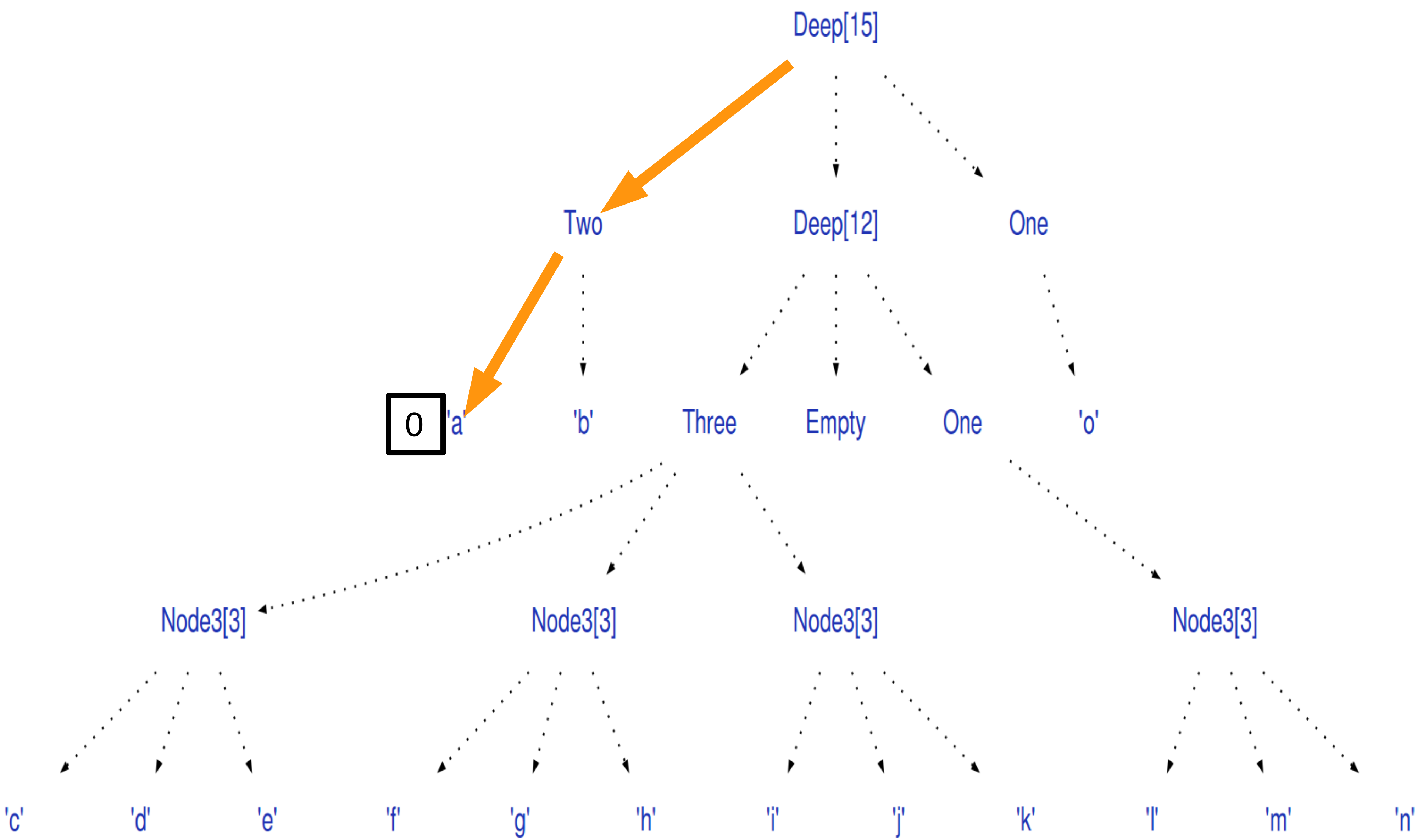


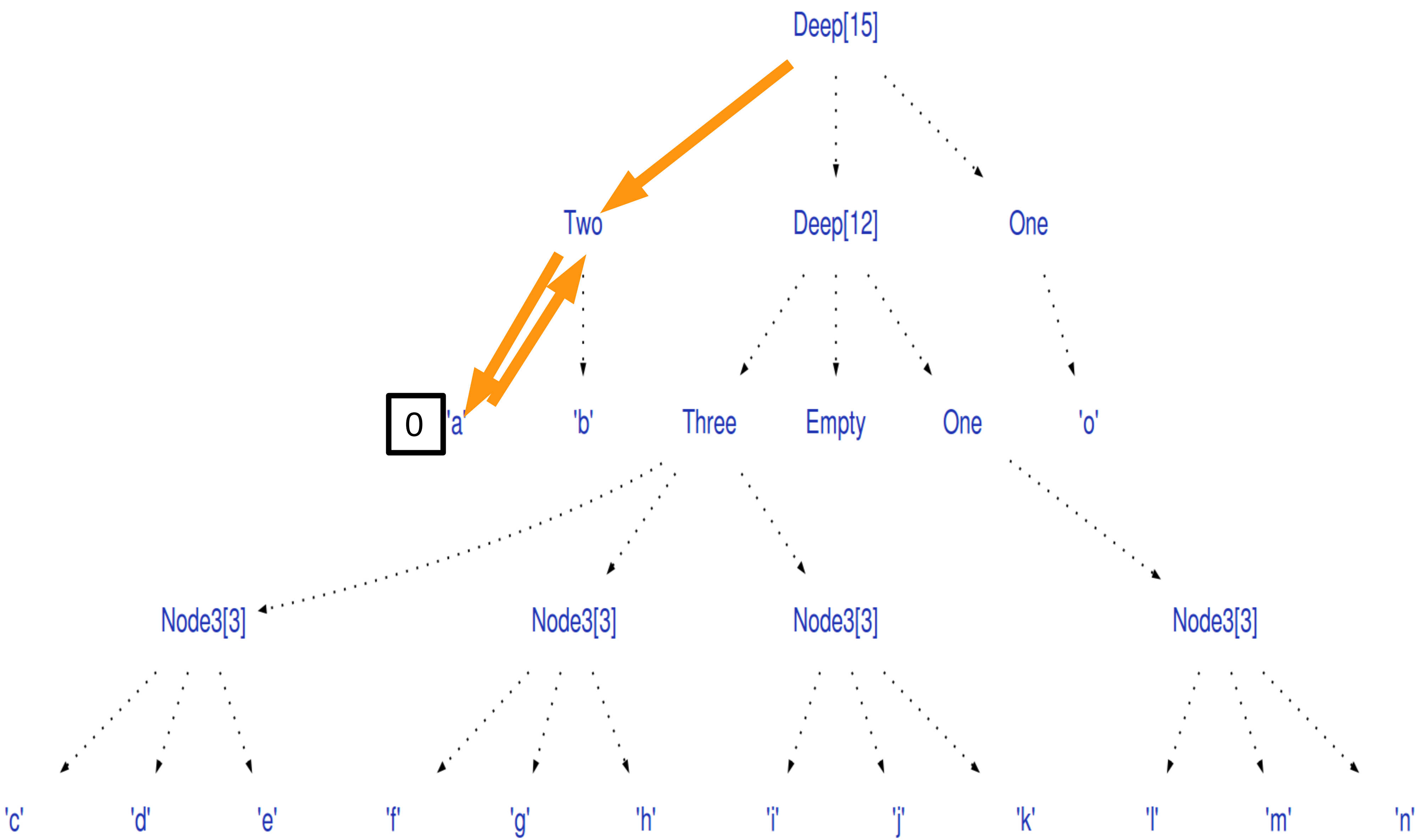
Deep[15]
Two
Deep[12]
One
'a'
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

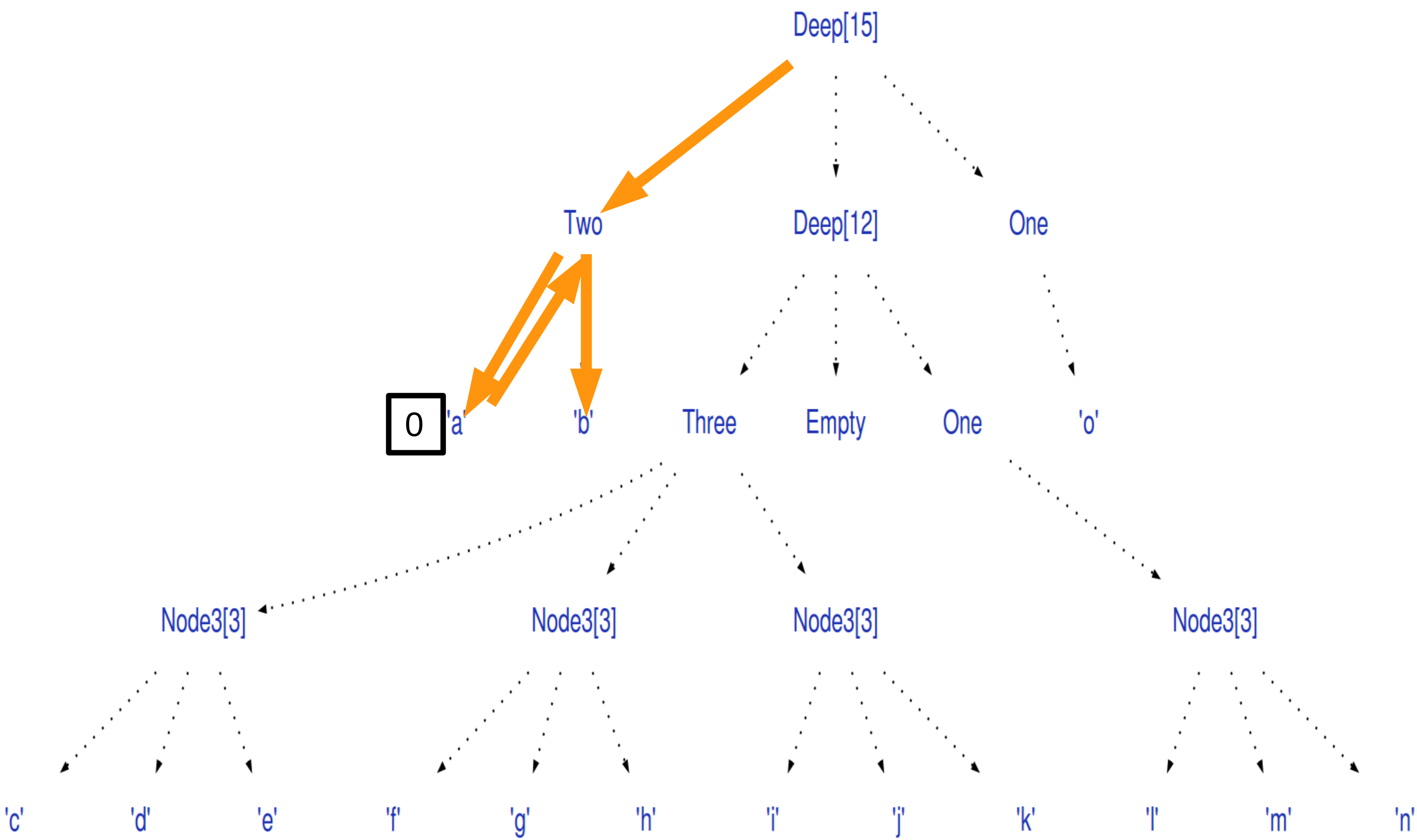

Deep[15]
Two
Deep[12]
One
'a'
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

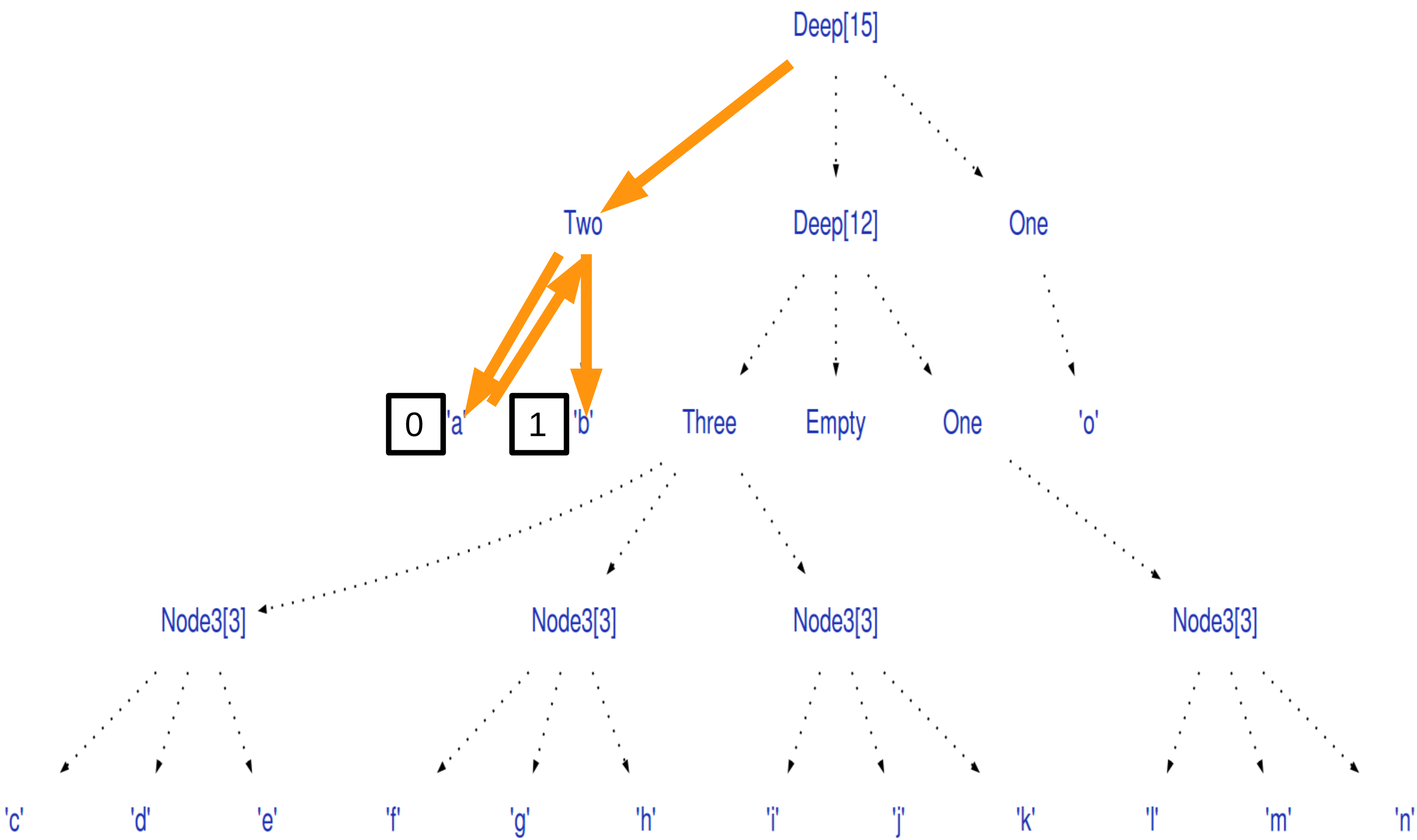


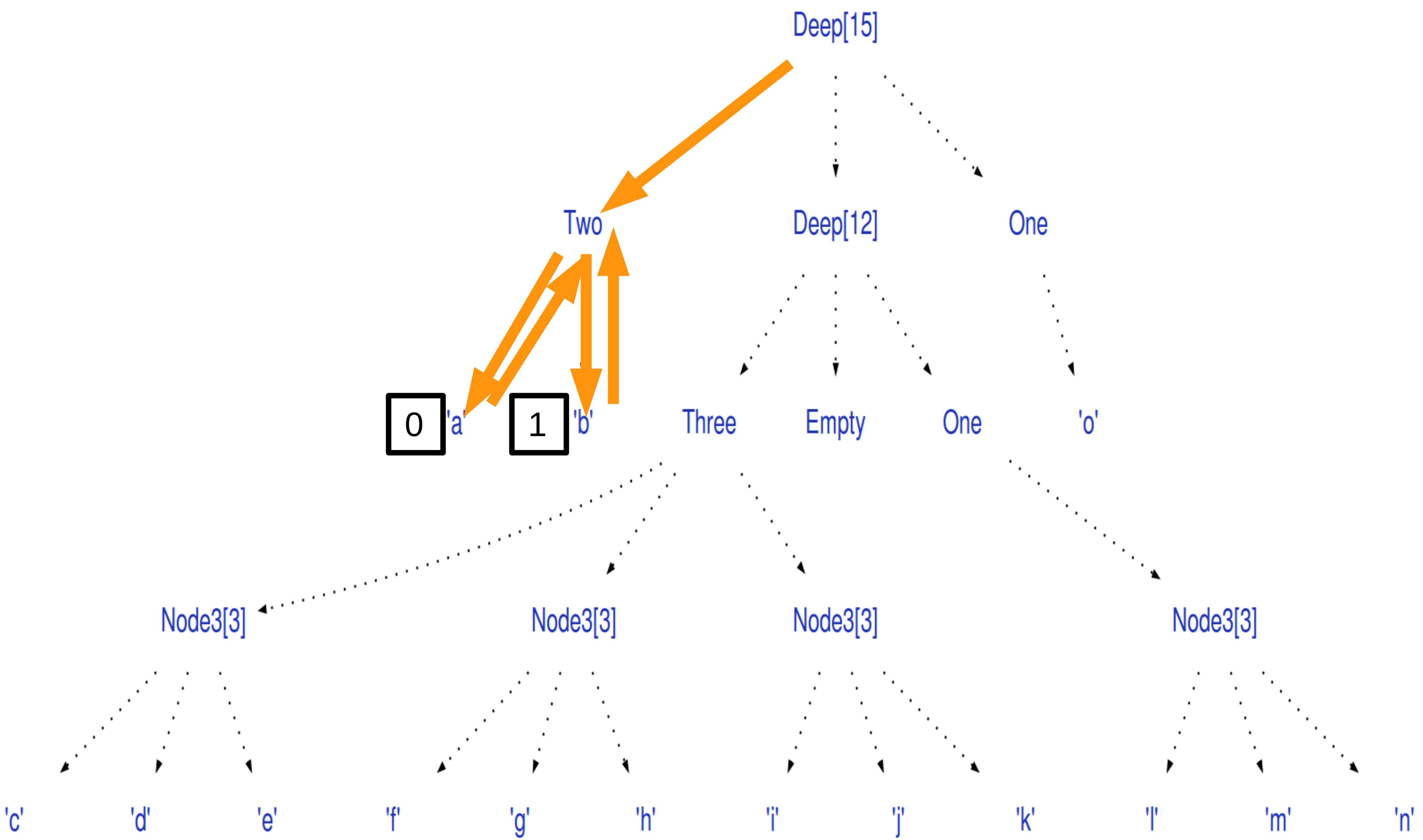
Deep[15]
Two
Deep[12]
One
'a'
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'


Deep[15]
Two
Deep[12]
One
0
'a'
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

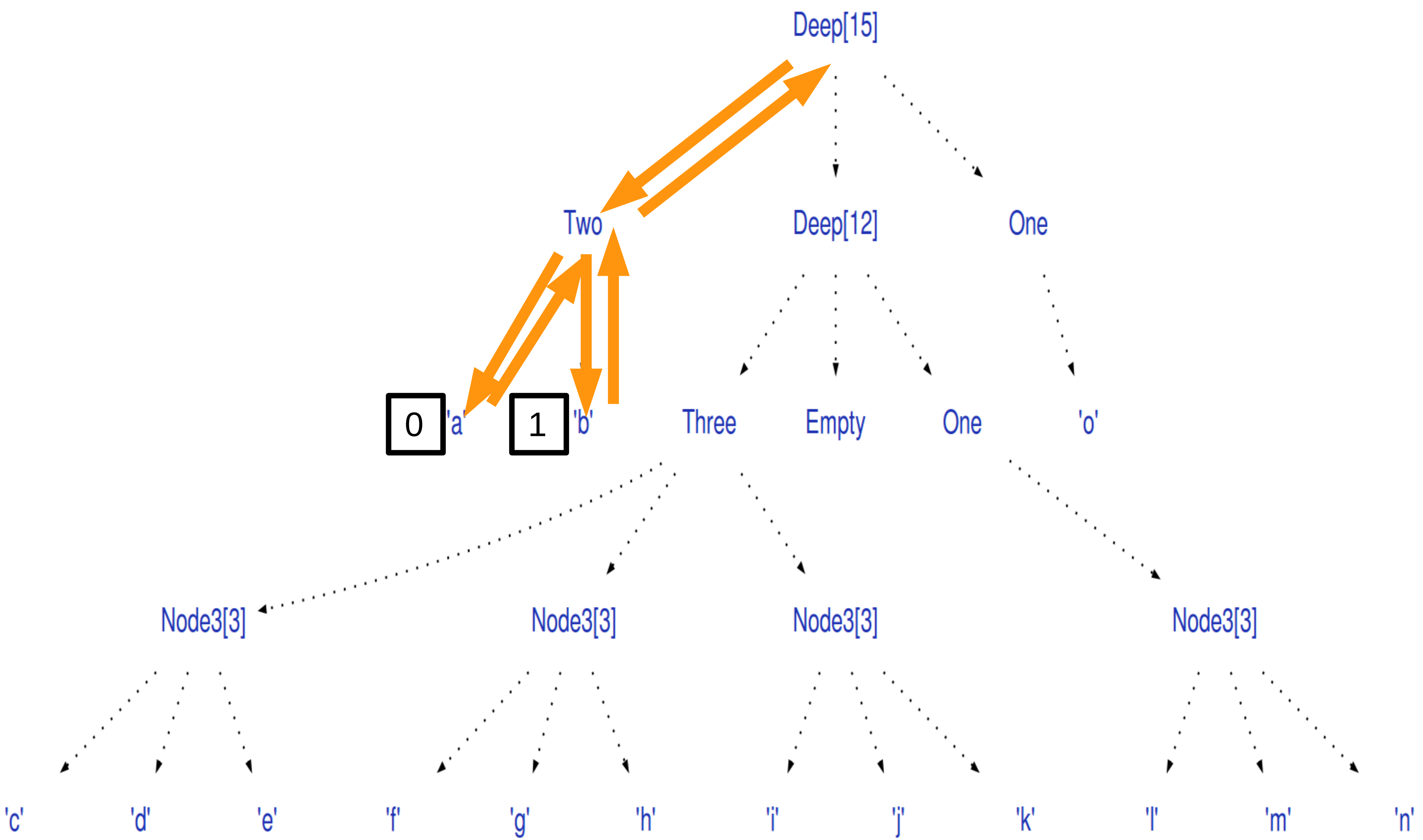

Deep[15]
Two
Deep[12]
One
0
'a'
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

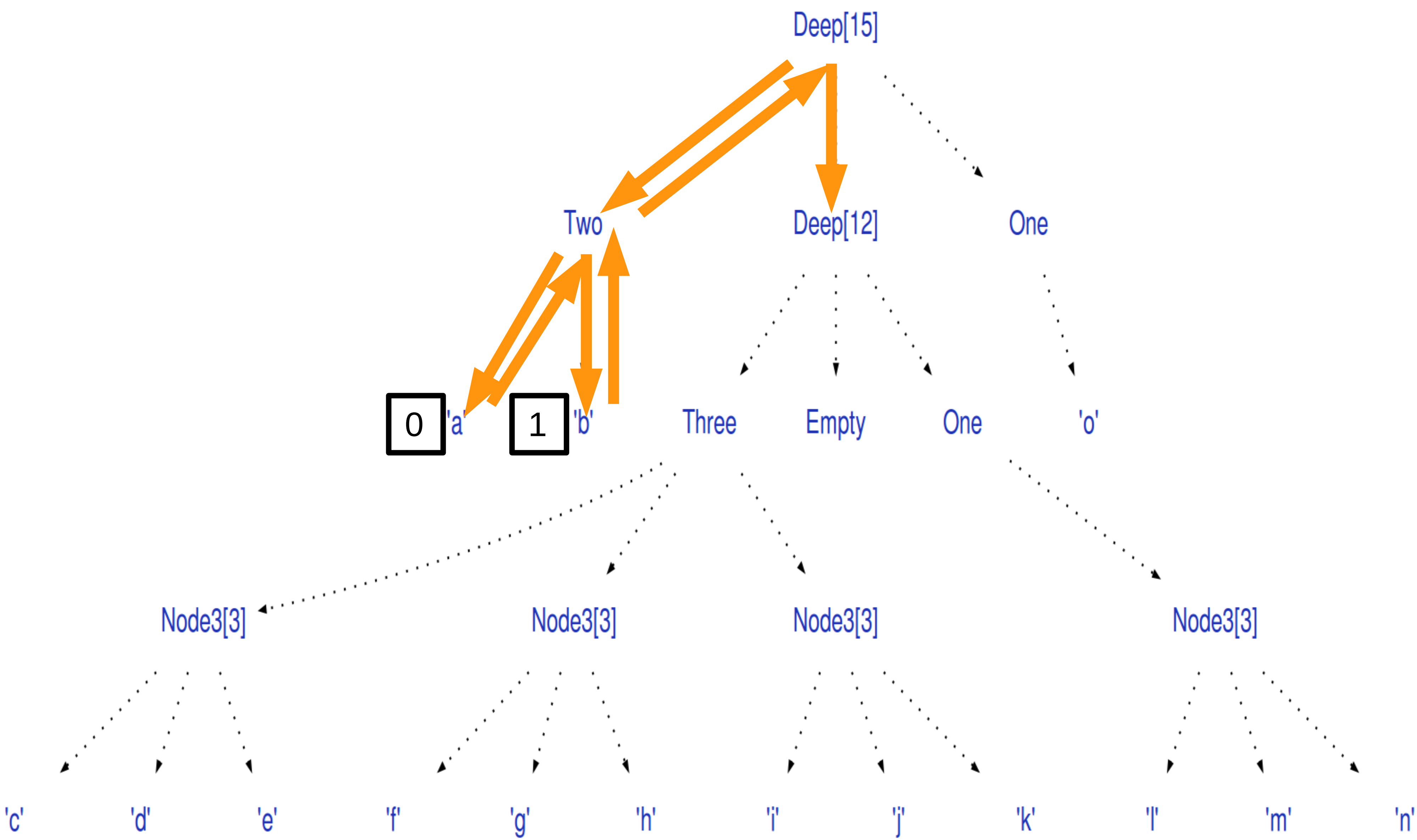

Deep[15]
Two
Deep[12]
One
0
'a'
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'


Deep[15]
Two
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

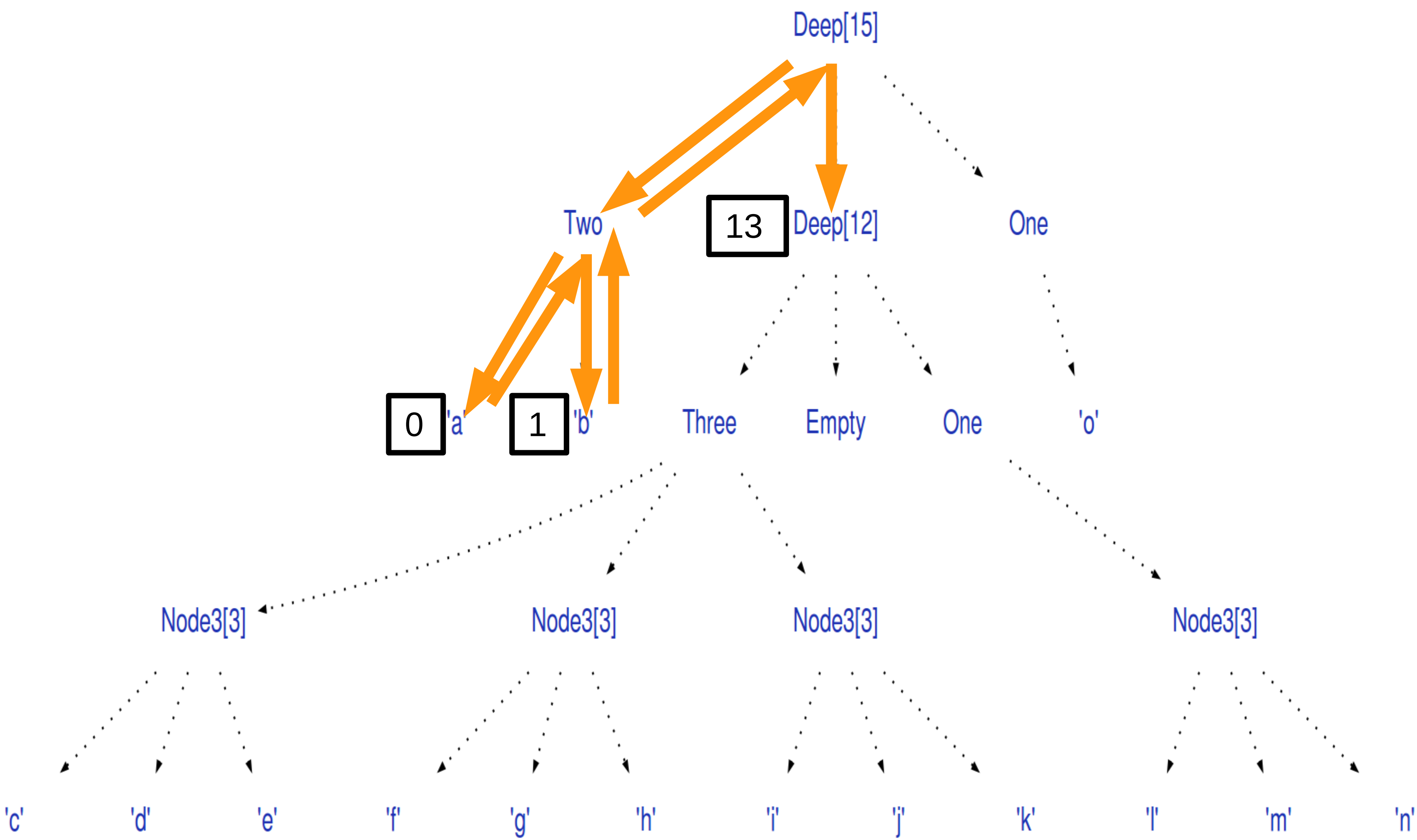


Deep[15]
Two
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

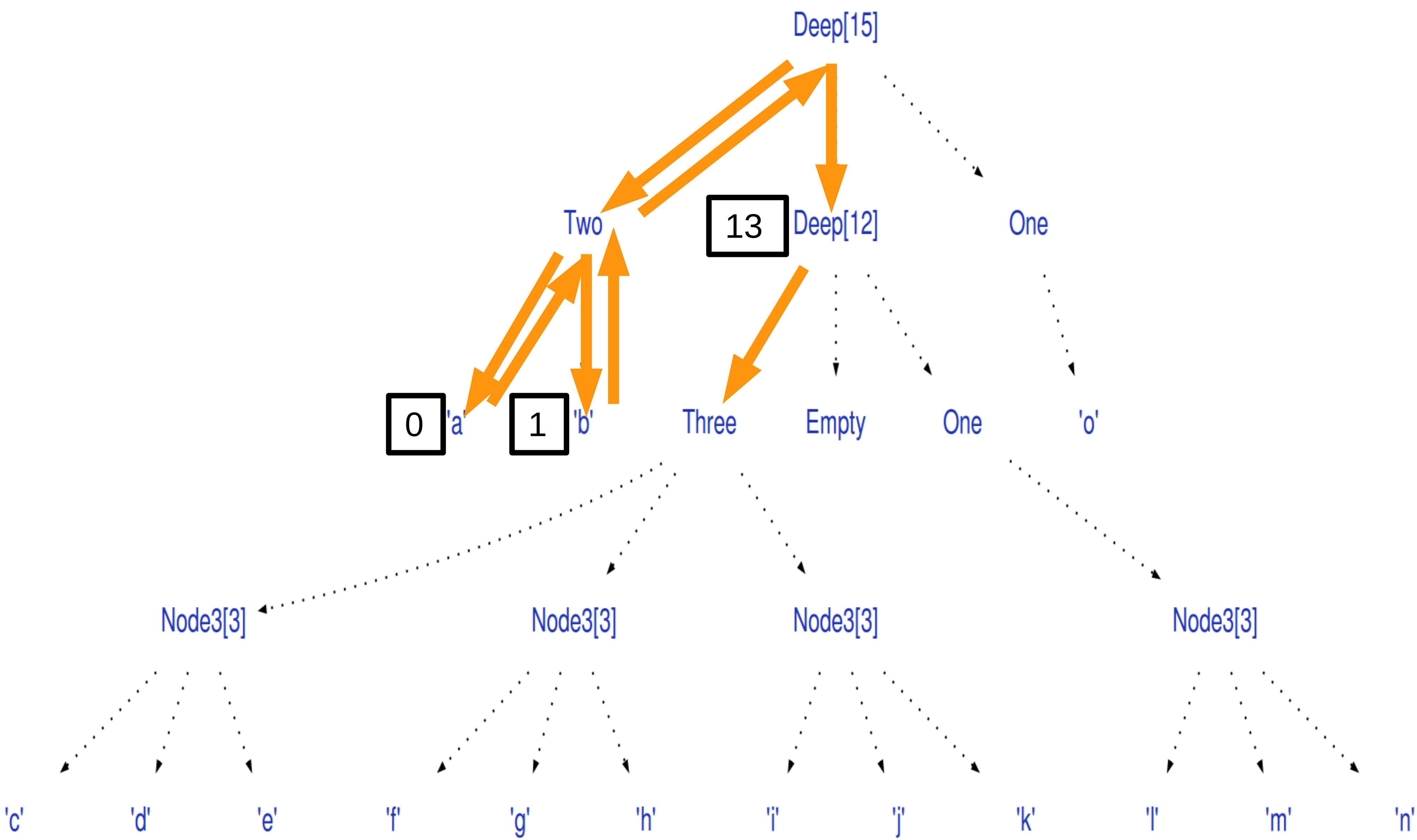

Deep[15]
Two
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

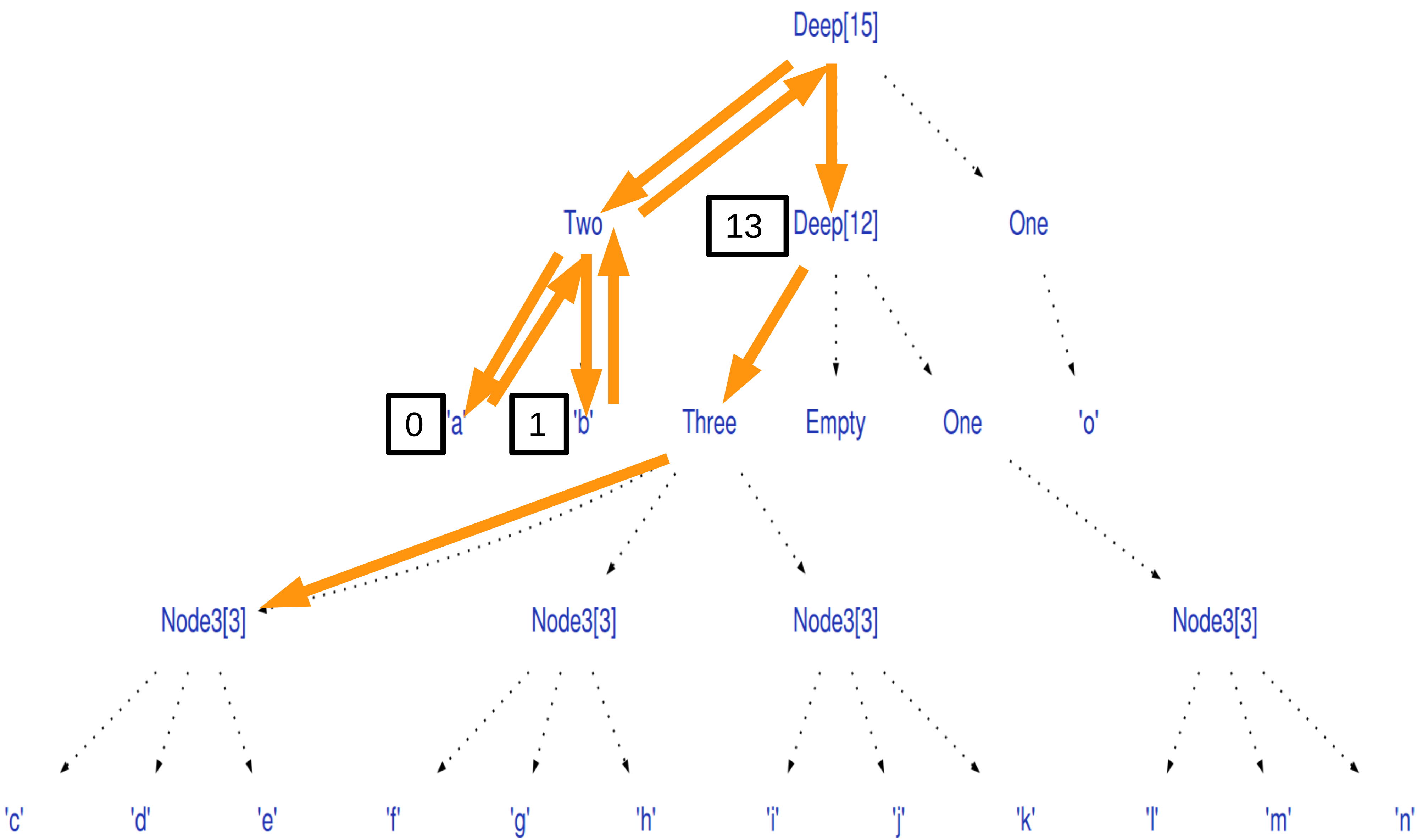


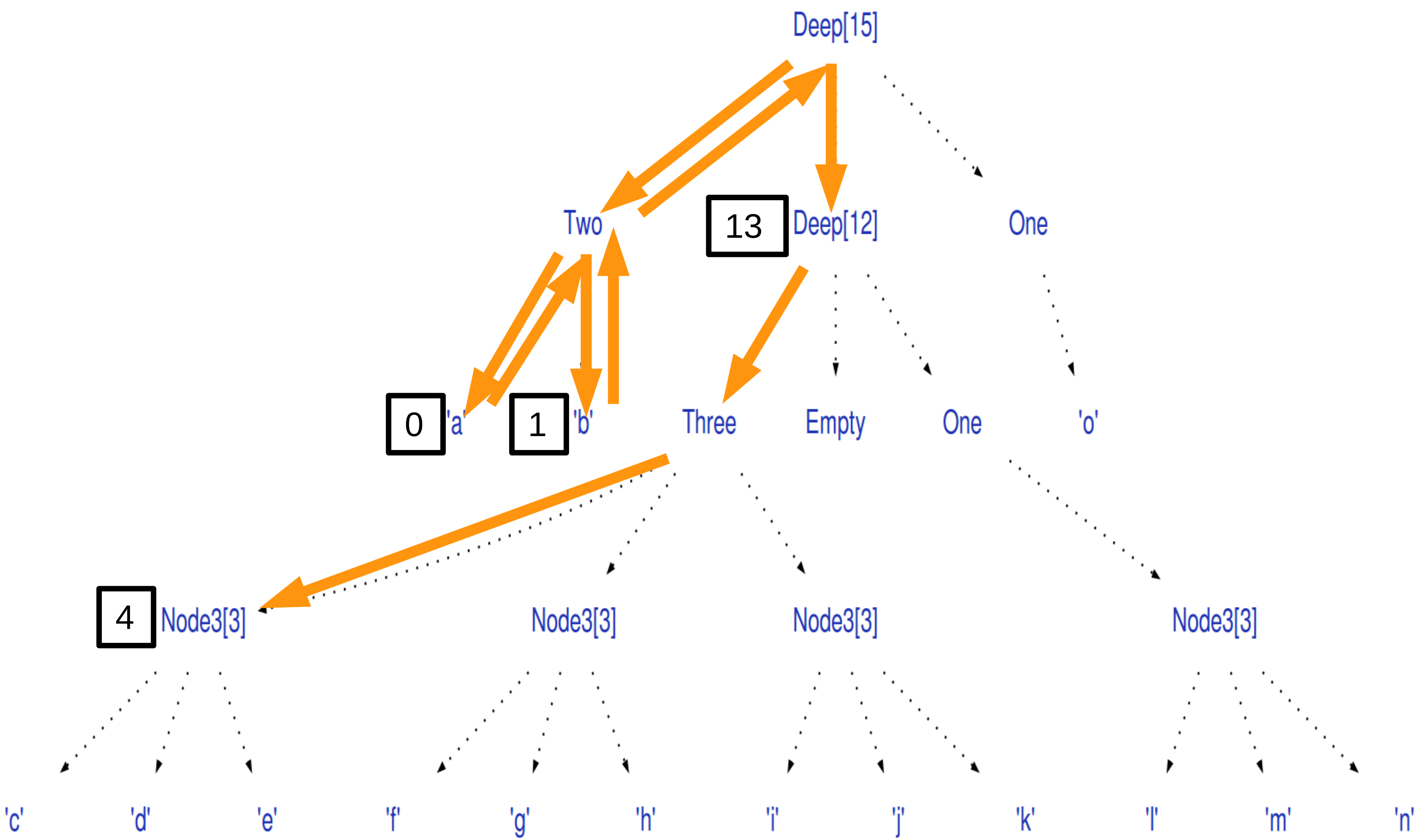
Deep[15]
Two
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'



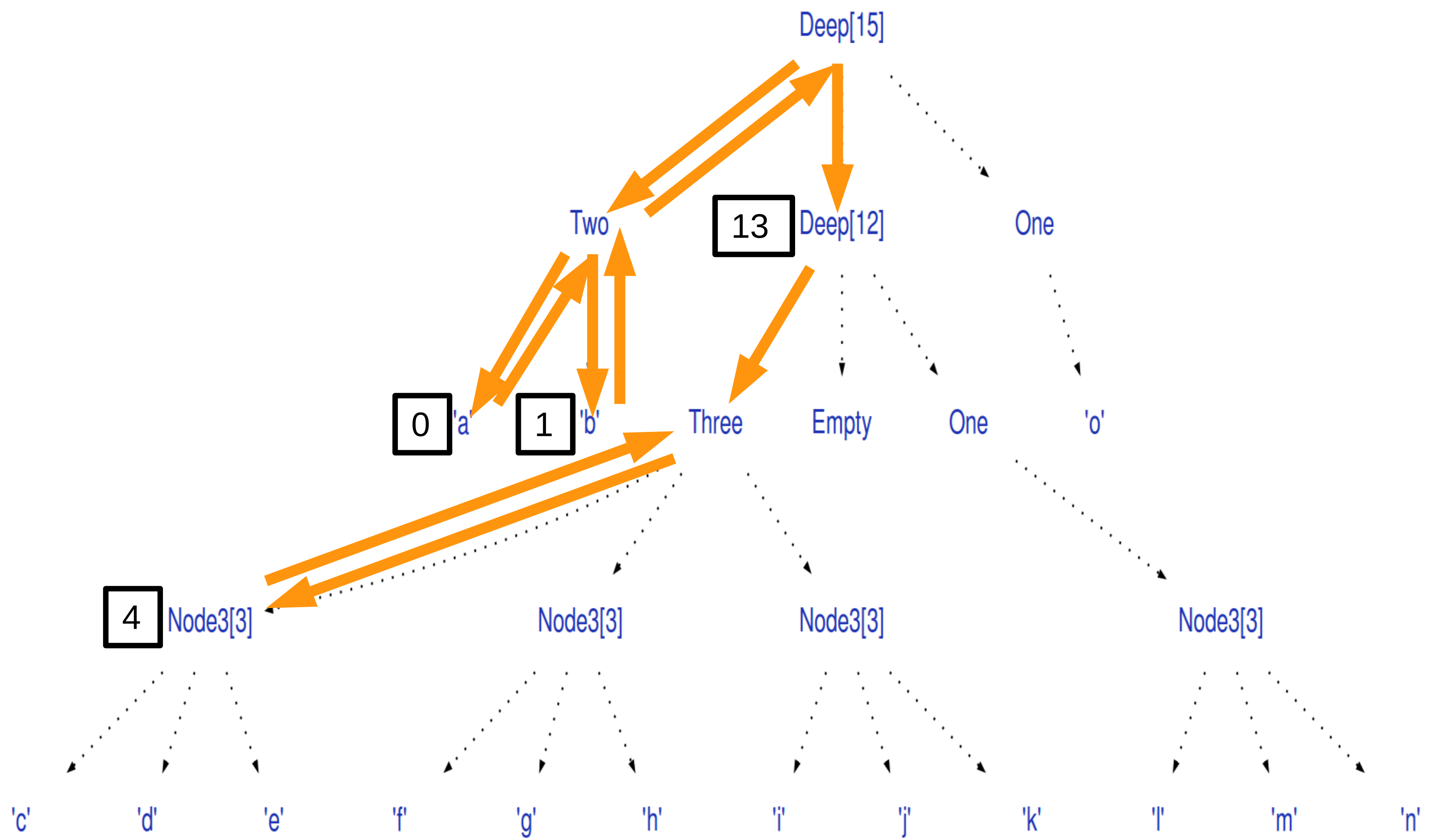
Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

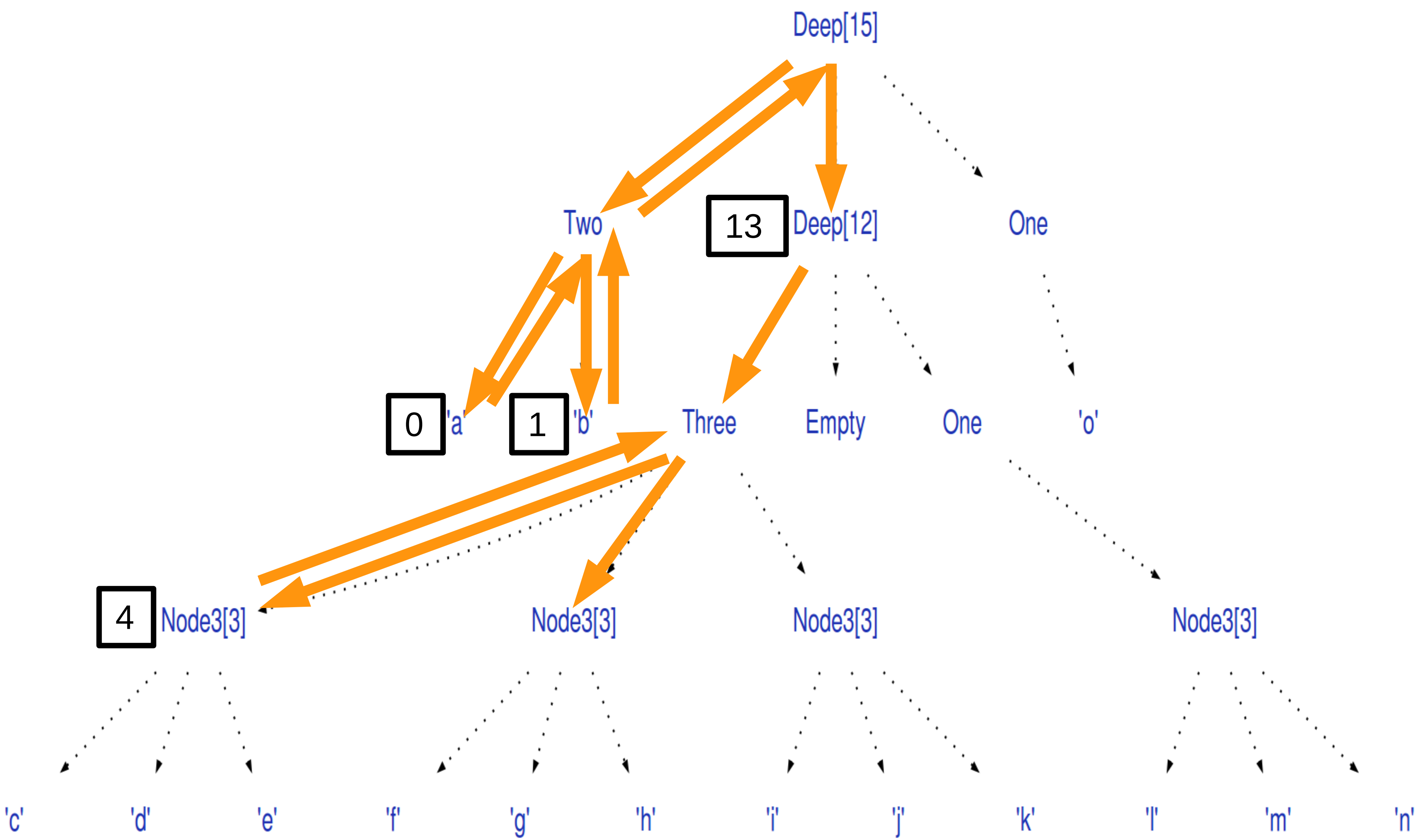

Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'


Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

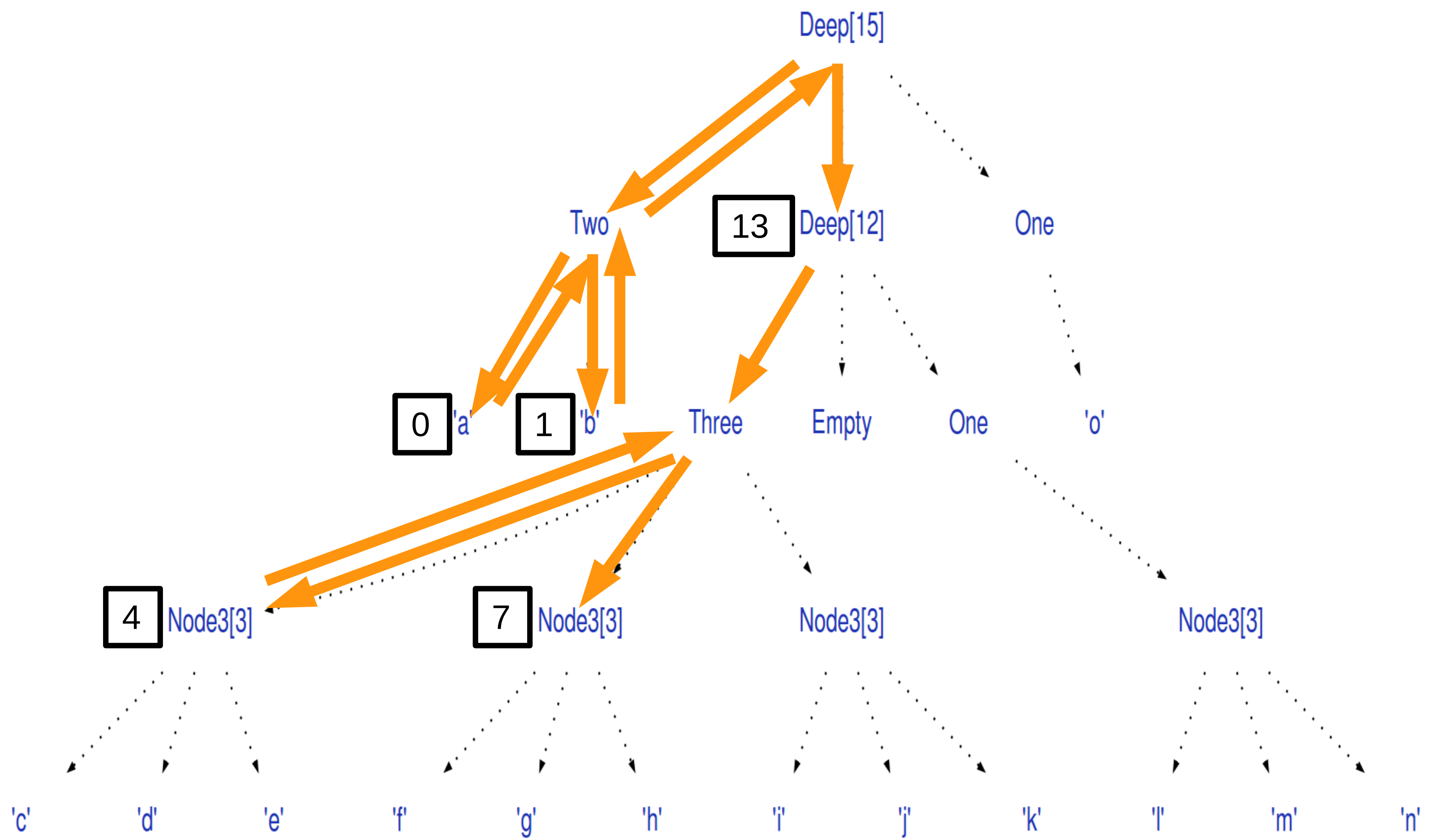


Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

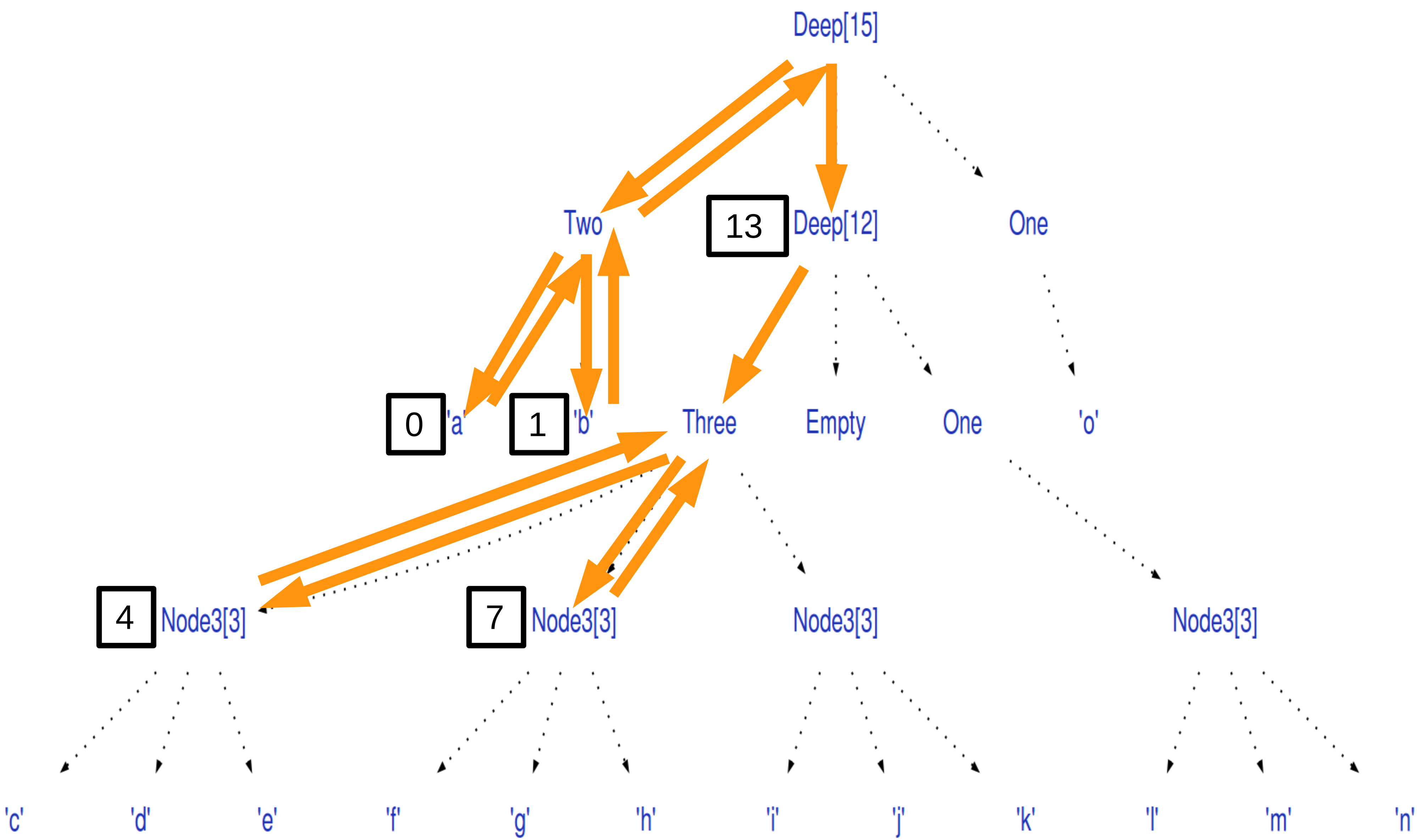

Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'



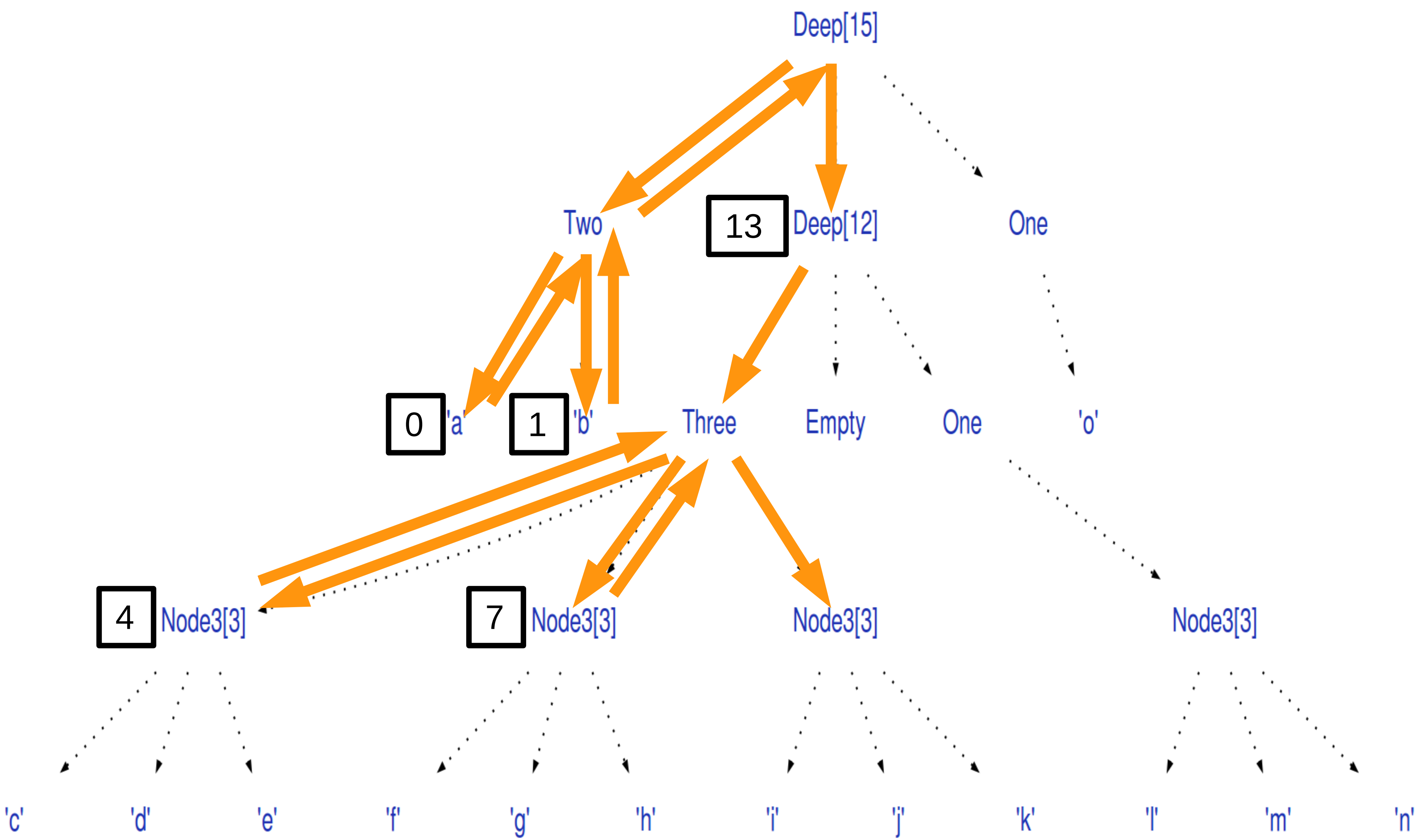
Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

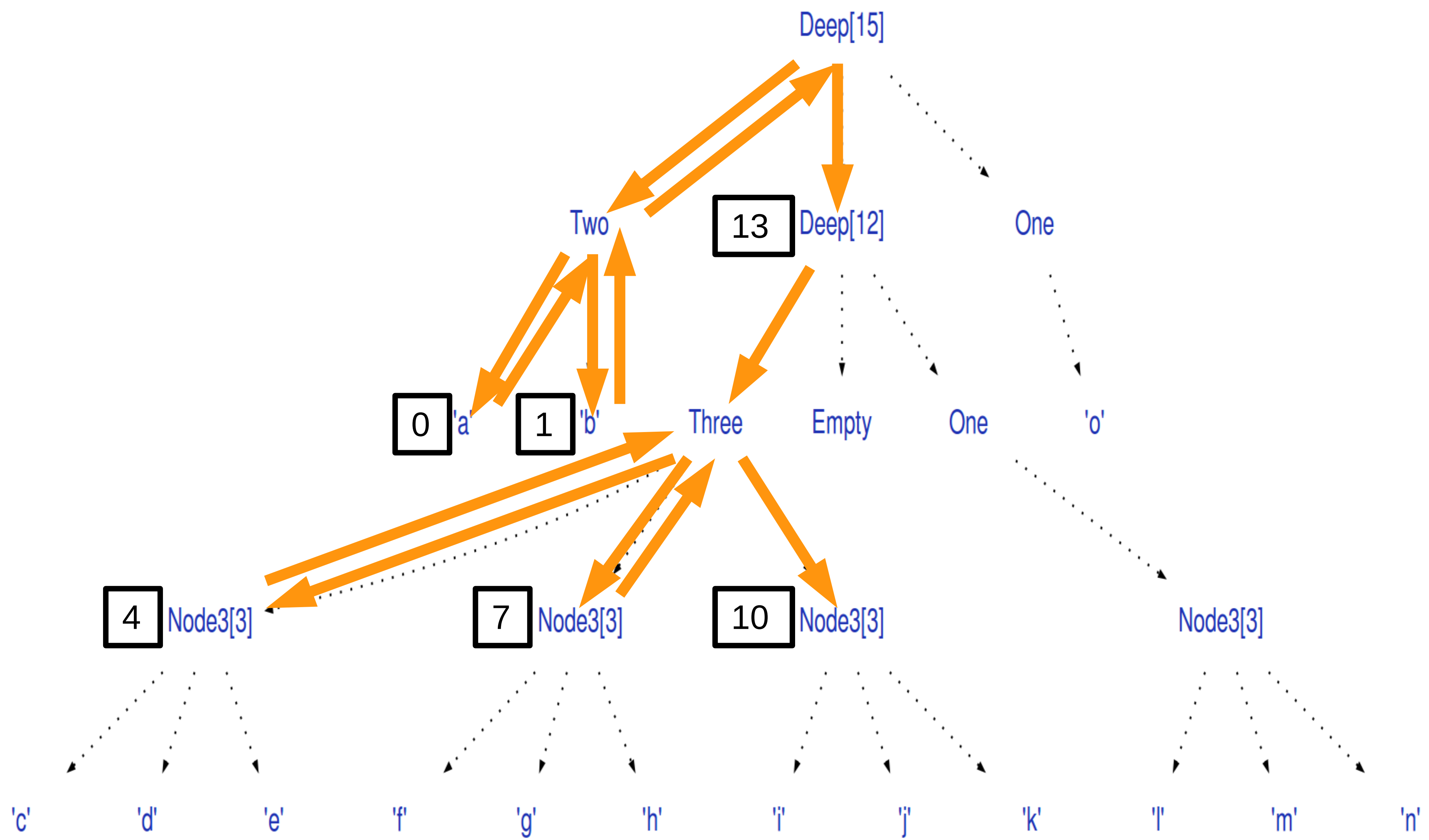


Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
7
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'

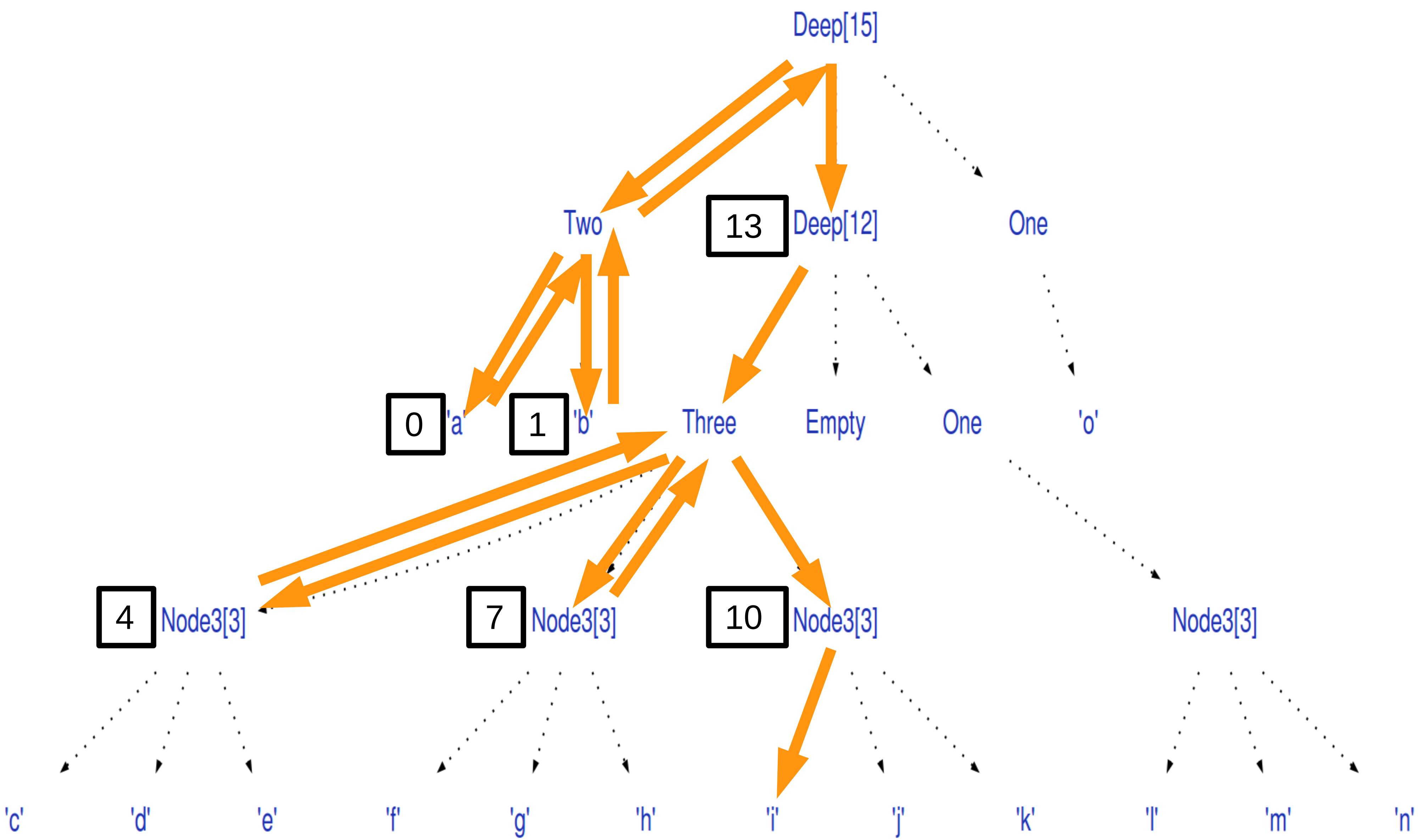


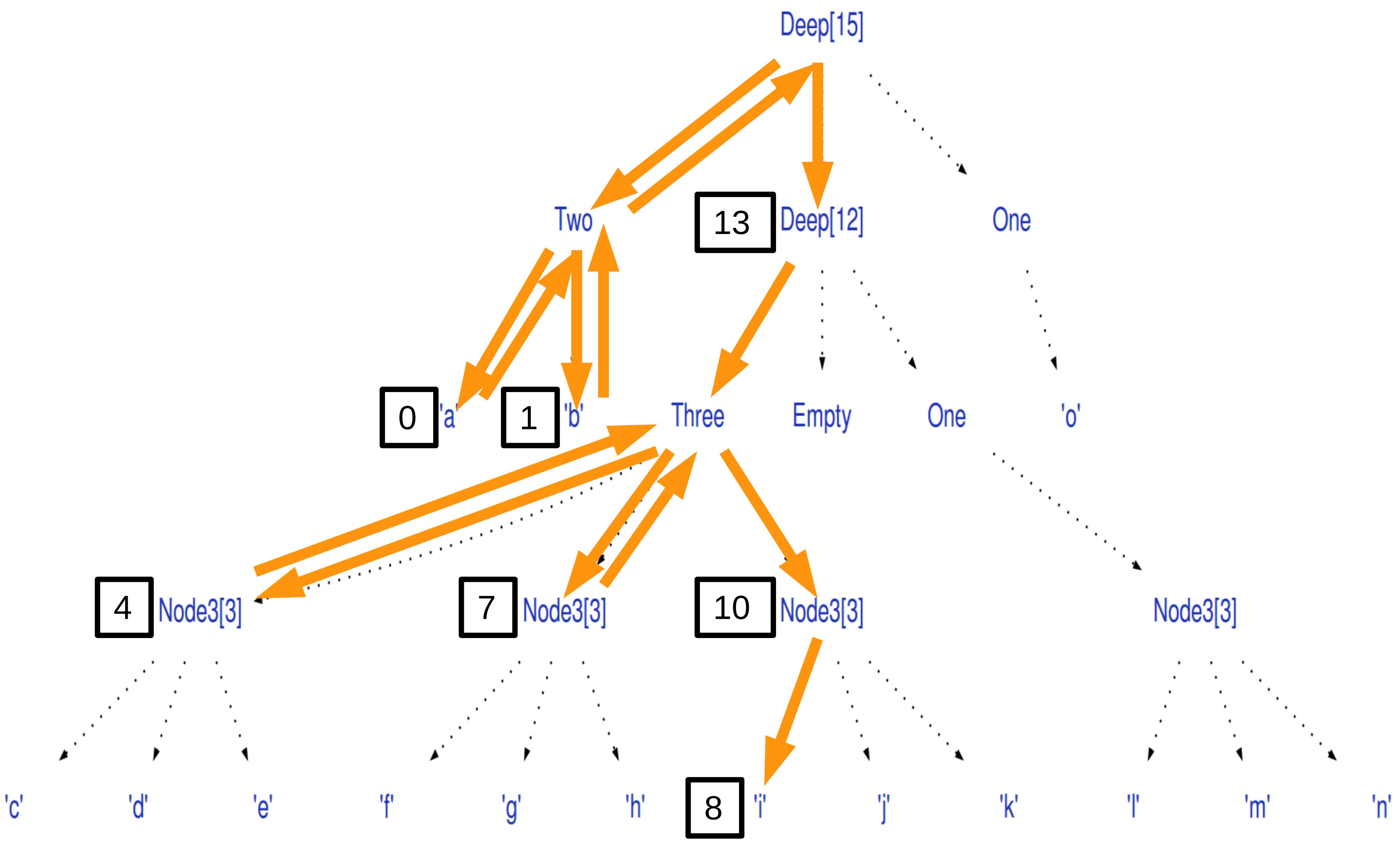
Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
7
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'


Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
7
Node3[3]
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'



Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
7
Node3[3]
10
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'


Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
7
Node3[3]
10
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
'i'
'j'
'k'
'l'
'm'
'n'


Deep[15]
Two
13
Deep[12]
One
0
'a'
1
'b'
Three
Empty
One
'o'
4
Node3[3]
7
Node3[3]
10
Node3[3]
Node3[3]
'c'
'd'
'e'
'f'
'g'
'h'
8
'i'
'j'
'k'
'l'
'm'
'n'

Finger Trees in Comparison

	Time (in ns) per operation randomly selected from			
	$\triangleleft$, $view_L$ (stack)	$\triangleright$, $view_L$ (queue)	$\triangleleft$, $view_L$, $\triangleright$, $view_R$ (deque)	*index*
Bankers queue	51	147	—	—
Bankers deque	56	229	75	—
Catenable deque	78	215	100	—
Skew binary random access list	44	—	—	295
Finger tree	67	106	89	—
Finger tree with sizes	74	128	94	482

Table 1. *Comparing persistent sequence implementations*

Resources

- The original paper:
  - http://www.soi.city.ac.uk/~ross/papers/FingerTree.html
- Finger Trees in Haskell:
  - http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Sequence.html
- Finger Trees in Clojure:
  - https://github.com/clojure/data.finger-tree