```
          _____                    _____                    _____                    _____                    _____                    _____            _____
         /\    \                  /\    \                  /\    \                  /\    \                  /\    \                  /\    \          /\    \
        /::\____\                /::\    \                /::\    \                /::\____\                /::\    \                /::\____\        /::\____\
       /:::/    /               /::::\    \              /::::\    \              /:::/    /               /::::\    \              /:::/    /       /:::/    /
      /:::/    /               /::::::\    \            /::::::\    \            /:::/    /               /::::::\    \            /:::/    /       /:::/    /
     /:::/    /               /:::/\:::\    \          /:::/\:::\    \          /:::/    /               /:::/\:::\    \          /:::/    /       /:::/    /
    /:::/____/               /:::/__\:::\    \        /:::/__\:::\    \        /:::/____/               /:::/__\:::\    \        /:::/    /       /:::/    /
   /::::\    \    _____     /::::\   \:::\    \       \:::\   \:::\    \      /::::\    \              /::::\   \:::\    \      /:::/    /       /:::/    /
  /::::::\    \            /::::::\   \:::\    \    ___\:::\   \:::\    \    /::::::\____\________    /::::::\   \:::\    \    /:::/    /       /:::/    /
 /:::/\:::\    \ /\    \  /:::/\:::\   \:::\    \  /\   \:::\   \:::\    \  /:::/\:::::::::::\    \  /:::/\:::\   \:::\    \  /:::/    /       /:::/    /
/:::/  \:::\    /::\____\/:::/  \:::\   \:::\____\/::\   \:::\   \:::\____\/:::/  |:::::::::::\____\/:::/__\:::\   \:::\____\/:::/____/       /:::/____/
\::/    \:::\  /:::/    /\::/    \:::\  /:::/    /\:::\   \:::\    \::/   /\::/   |::|~~~|~~~~~~    \:::\   \:::\   \::/    /\:::\    \       \:::\    \
 \/____/ \:::\/:::/    /  \/____/ \:::\/:::/    /  \:::\   \:::\    \/____/ \/____|::|   |           \:::\   \:::\   \/____/  \:::\    \       \:::\    \
          \::::::/    /            \::::::/    /    \:::\   \:::\    \            |::|   |            \:::\   \:::\    \       \:::\    \       \:::\    \
           \::::/    /              \::::/    /      \:::\   \:::\____\           |::|   |             \:::\   \:::\____\       \:::\    \       \:::\    \
           /:::/    /               /:::/    /        \:::\  /:::/    /           |::|   |              \:::\   \::/    /        \:::\    \       \:::\    \
          /:::/    /               /:::/    /          \:::\/:::/    /            |::|   |               \:::\   \/____/          \:::\    \       \:::\    \
         /:::/    /               /:::/    /            \::::::/    /             |::|   |                \:::\    \               \:::\    \       \:::\    \
        /:::/    /               /:::/    /              \::::/    /              \::|   |                 \:::\____\               \:::\____\       \:::\____\
        \::/    /                \::/    /                \::/    /                \:|   |                  \::/    /                \::/    /        \::/    /
         \/____/                  \/____/                  \/____/                  \|___|                   \/____/                  \/____/          \/____/
```

Haskell Done Quick (03)

Lars Pfrenger | 07. Februar 2025

1. Stacks

Funktionsweise

Push

Etwas auf den Stack “legen”

Pop

Etwas vom Stack “nehmen”

$\rightarrow$ Man nehmet/leget nur von “oben”

$\rightarrow$ Last In First Out (**LIFO**)

Funktionsweise

Push

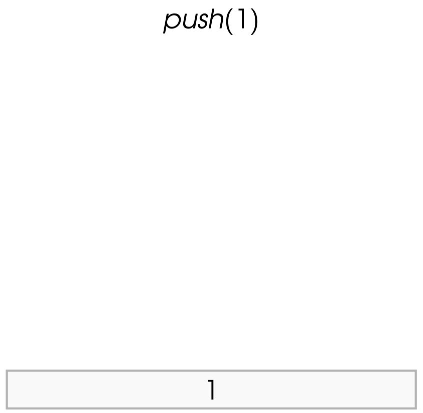
Etwas auf den Stack “legen”

Pop

Etwas vom Stack “nehmen”

$\rightarrow$ Man nehmet/leget nur von “oben”

$\rightarrow$ Last In First Out (**LIFO**)



Funktionsweise

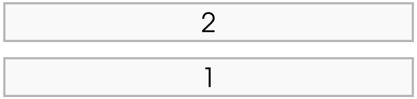
Push

Etwas auf den Stack “legen”

Pop

Etwas vom Stack “nehmen”

push(2)



→ Man nehmet/leget nur von “oben”

→ Last In First Out (**LIFO**)

Funktionsweise

Push

Etwas auf den Stack “legen”

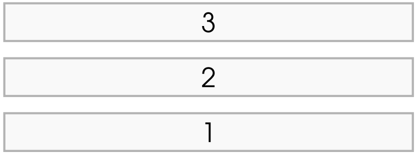
Pop

Etwas vom Stack “nehmen”

$\rightarrow$ Man nehmet/leget nur von “oben”

$\rightarrow$ Last In First Out (**LIFO**)

push(3)



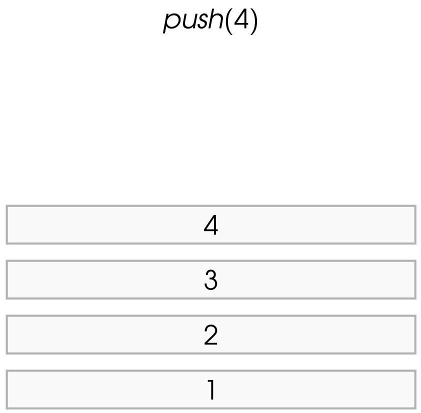
Funktionsweise

Push

Etwas auf den Stack “legen”

Pop

Etwas vom Stack “nehmen”



→ Man nehmet/leget nur von “oben”

→ Last In First Out (**LIFO**)

Funktionsweise

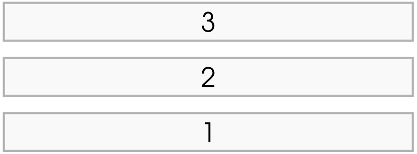
Push

Etwas auf den Stack “legen”

Pop

Etwas vom Stack “nehmen”

$pop() \rightarrow 4$



$\rightarrow$ Man nehmet/leget nur von “oben”

$\rightarrow$ Last In First Out (**LIFO**)

Funktionsweise

Push

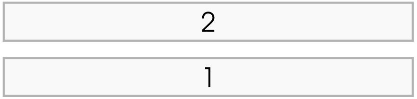
Etwas auf den Stack "legen"

Pop

Etwas vom Stack "nehmen"

→ Man nehmet/leget nur von "oben"
→ Last In First Out (**LIFO**)

$pop() \rightarrow 3$



Funktionsweise

Push

Etwas auf den Stack "legen"

Pop

Etwas vom Stack "nehmen"

$pop() \rightarrow 2$

$\rightarrow$ Man nehmet/leget nur von "oben"

$\rightarrow$ Last In First Out (**LIFO**)

Funktionsweise

Push

Etwas auf den Stack “legen”

Pop

Etwas vom Stack “nehmen”

$pop() \rightarrow 1$

→ Man nehmet/leget nur von “oben”

→ Last In First Out (**LIFO**)

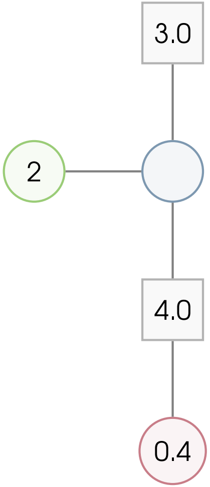
2. Baum Serialisieren

Baum (Darstellung)

```
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```

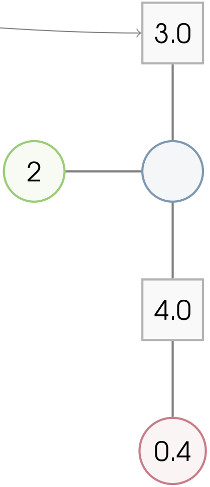
Baum (Darstellung)

```
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```



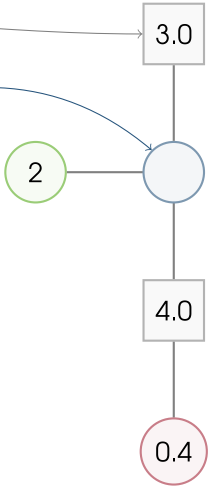
Baum (Darstellung)

```
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```



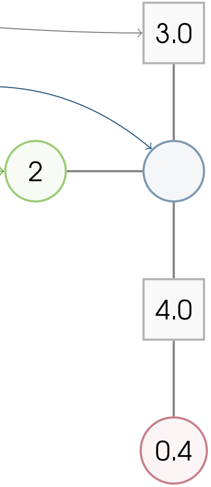
Baum (Darstellung)

```
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```



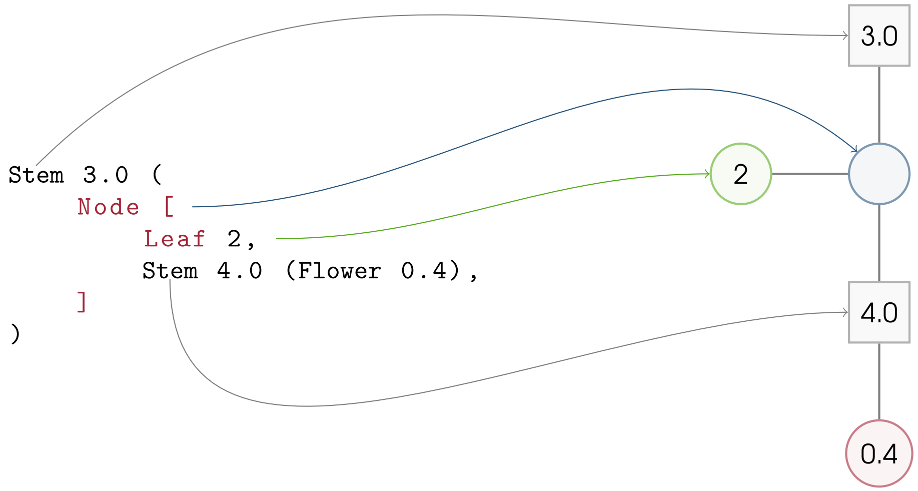
Baum (Darstellung)

```
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```

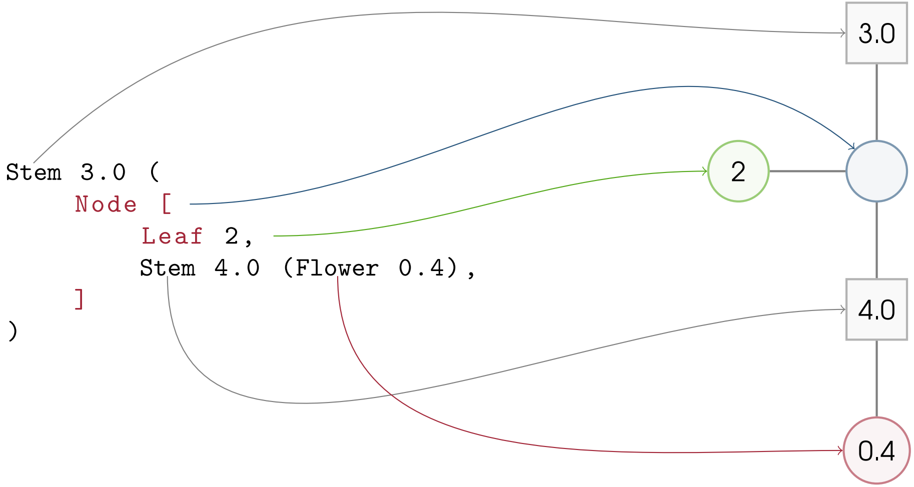


Baum (Darstellung)

```haskell
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```

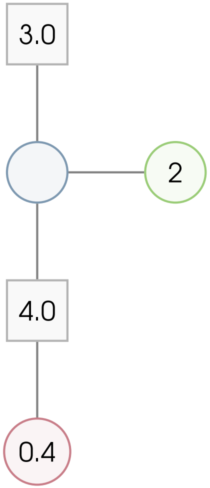


Baum (Darstellung)

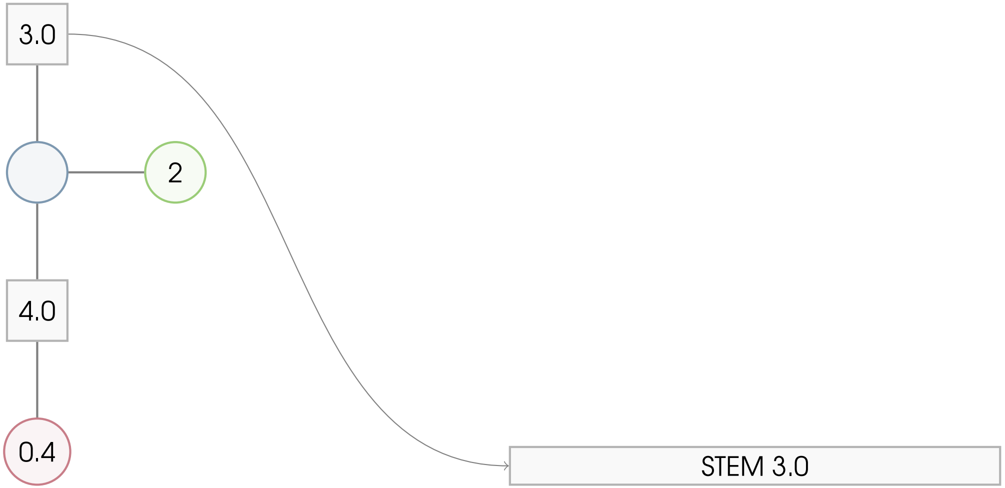


```
Stem 3.0 (
    Node [
        Leaf 2,
        Stem 4.0 (Flower 0.4),
    ]
)
```

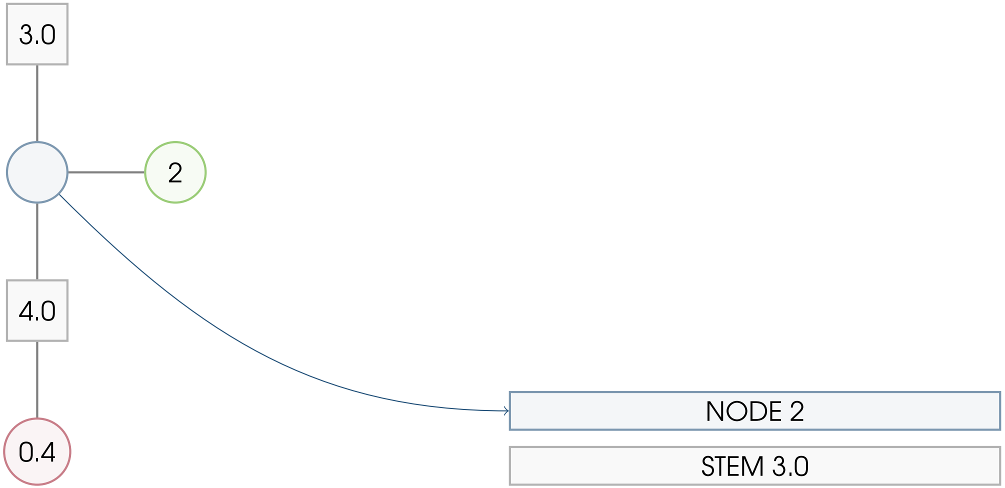
Baum zu Stack (Idee)



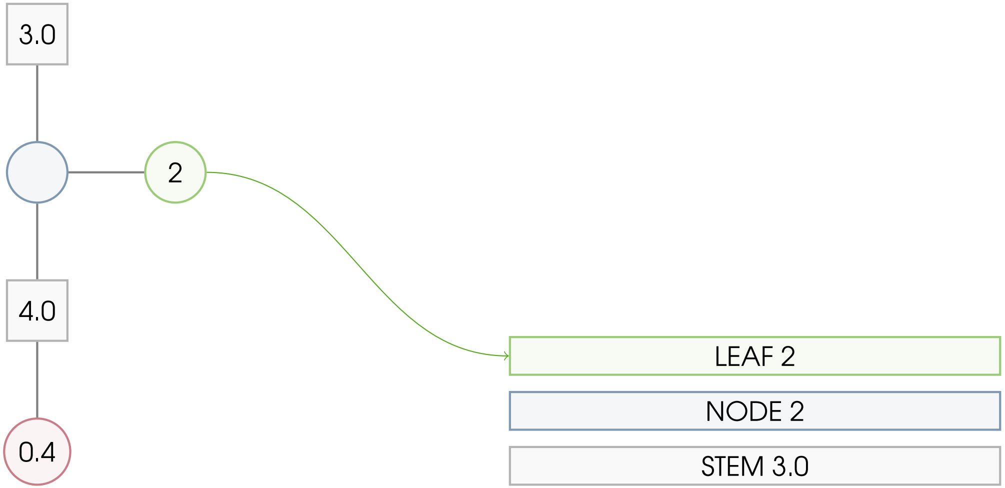
Baum zu Stack (Idee)



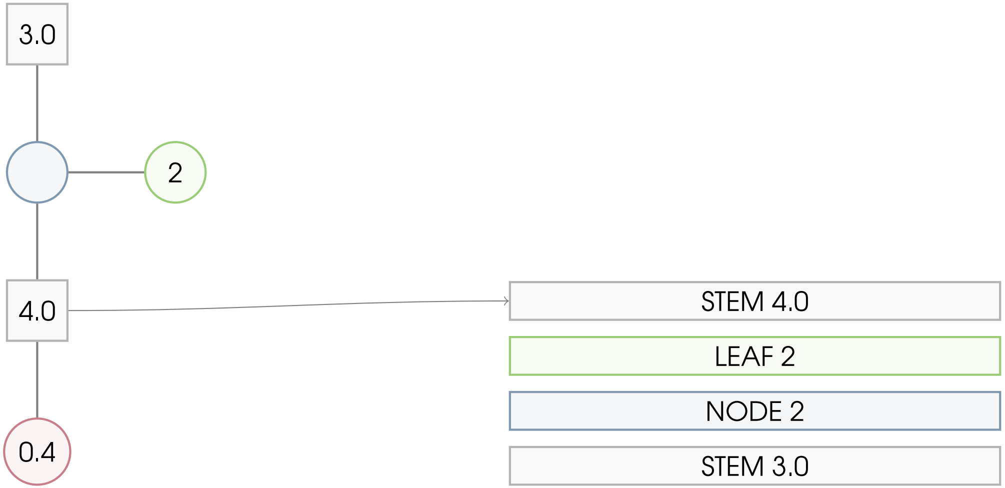
Baum zu Stack (Idee)



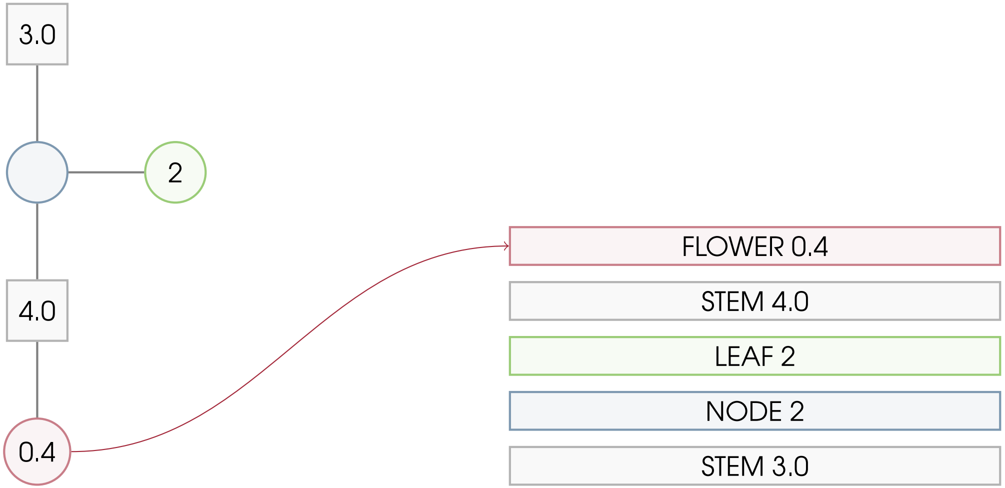
Baum zu Stack (Idee)



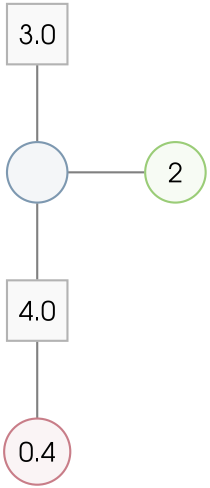
Baum zu Stack (Idee)



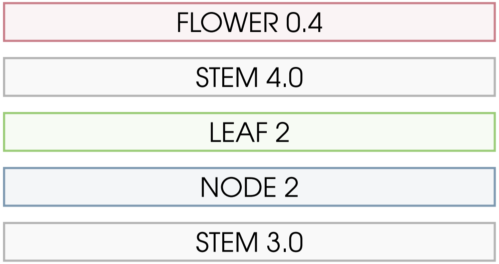
Baum zu Stack (Idee)



Baum zu Stack (Idee)



```
[FLOWER 0.4, STEM 4.0, LEAF 2, NODE 2, STEM 3.0]
```



Baum zu Stack (Code)

```haskell
plantToStack :: Plant leaf flower fruit stem
    -> [StackPlant leaf flower fruit stem]

plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p ++ [STEM s])
    (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

Baum zu Stack (Code)

```
plantToStack :: Plant leaf flower fruit stem
    -> [StackPlant leaf flower fruit stem]

plantToStack = foldPlant
 fLeaf (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p ++ [STEM s])
    (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

Baum zu Stack (Code)

```
plantToStack :: Plant leaf flower fruit stem
    -> [StackPlant leaf flower fruit stem]

plantToStack = foldPlant
    (\a -> [LEAF a])
fFlower (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p ++ [STEM s])
    (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

Baum zu Stack (Code)

```
plantToStack :: Plant leaf flower fruit stem
    -> [StackPlant leaf flower fruit stem]

plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
 fFruit (\a -> [FRUIT a])
    (\s p -> p ++ [STEM s])
    (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

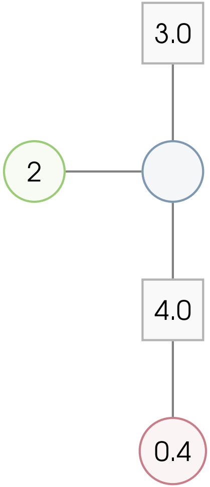
Baum zu Stack (Code)

```
plantToStack :: Plant leaf flower fruit stem
    -> [StackPlant leaf flower fruit stem]

plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
fStem (\s p -> p ++ [STEM s])
    (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

Baum zu Stack (Code)

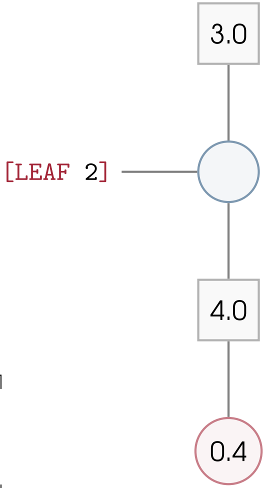
```
plantToStack :: Plant leaf flower fruit stem
    -> [StackPlant leaf flower fruit stem]

plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p ++ [STEM s])
fNode (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

What the fold doin? (post-order-traversal)



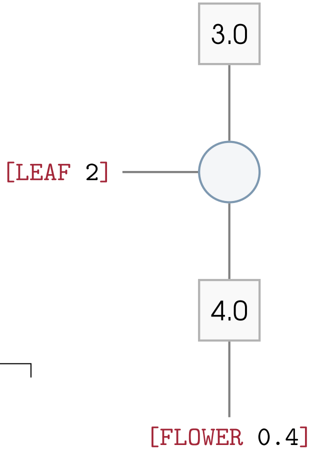
```
plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p++[STEM s])
    (\ps -> concat (reverse ps)++[NODE (length ps)])
```

What the fold doin? (post-order-traversal)



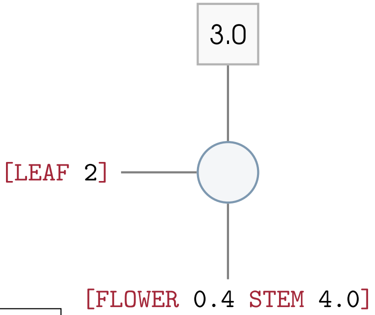
```
plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p++[STEM s])
    (\ps -> concat (reverse ps)++[NODE (length ps)])
```

What the fold doin? (post-order-traversal)



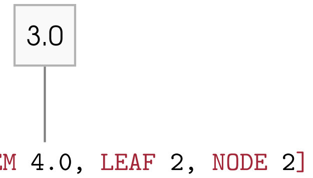
```
plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p++[STEM s])
    (\ps -> concat (reverse ps)++[NODE (length ps)])
```

What the fold doin? (post-order-traversal)



```
plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p++[STEM s])
    (\ps -> concat (reverse ps)++[NODE (length ps)])
```

What the fold doin? (post-order-traversal)



```
plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p++[STEM s])
    (\ps -> concat (reverse ps)++[NODE (length ps)])
```

What the fold doin? (post-order-traversal)

```
[FLOWER 0.4, STEM 4.0, LEAF 2, NODE 2, STEM 3.0]
```

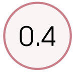
```haskell
plantToStack = foldPlant
    (\a -> [LEAF a])
    (\a -> [FLOWER a])
    (\a -> [FRUIT a])
    (\s p -> p++[STEM s])
    (\ps -> concat (reverse ps) ++ [NODE (length ps)])
```

Stack zu Baum (Idee)

```
[FLOWER 0.4, STEM 4.0, LEAF 2, NODE 2, STEM 3.0]
```

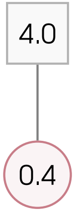
Stack zu Baum (Idee)

```
[STEM 4.0, LEAF 2, NODE 2, STEM 3.0]
```



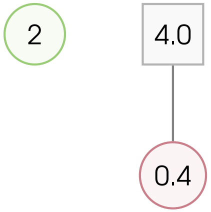
Stack zu Baum (Idee)

```
[LEAF 2, NODE 2, STEM 3.0]
```



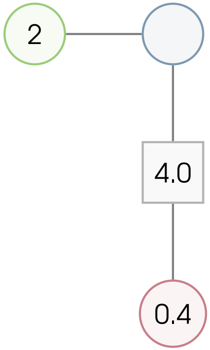
Stack zu Baum (Idee)

```
[NODE 2, STEM 3.0]
```



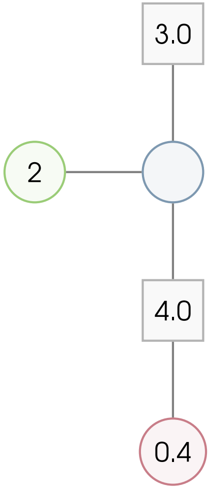
Stack zu Baum (Idee)

```
[STEM 3.0]
```



Stack zu Baum (Idee)

`[]`



Stack zu Baum (Code)

```
stackToPlants :: [StackPlant leaf flower fruit stem]
    -> [Plant leaf flower fruit stem]

stackToPlants = foldl
    (\ps c -> case c of
        LEAF l -> Leaf l : ps
        FLOWER f -> Flower f : ps
        FRUIT f -> Fruit f : ps
        STEM s -> Stem s (head ps) : tail ps
        NODE n -> Node (take n ps) : drop n ps)
    []
```

What the fold doin (the sequel)

```
[FLOWER 0.4, STEM 4.0,                 []
 LEAF 2, NODE 2, STEM 3.0]
```

What the fold doin (the sequel)

```
[STEM 4.0,
 LEAF 2, NODE 2, STEM 3.0]
```

```
[Flower 0.4]
```

What the fold doin (the sequel)

```
[LEAF 2, NODE 2, STEM 3.0]        [(Stem 4.0 (Flower 0.4))]
```

What the fold doin (the sequel)

```
[NODE 2, STEM 3.0]          [Leaf 2, (Stem 4.0 (Flower 0.4))]
```

What the fold doin (the sequel)

```
[STEM 3.0]          [(Node [Leaf 2, (Stem 4.0 (Flower 0.4))])]
```

What the fold doin (the sequel)

```
[]      [(Stem 3.0 (Node [Leaf 2, (Stem 4.0 (Flower 0.4))]))]
```