Deriving work-efficient, in-place parallel scan

Conal Elliott

March 7, 2011

IFIP working group 2.8

Modern GPU hardware is

- inexpensive to buy, but
- expensive to program.

```
scanl :: (b → a → b) → b → [a] → [b]


scanl f z [x1, x2,  ⋯ ] ≡
  [z, z `f` x1, (z `f` x1) `f` x2, ⋯]


scanl :: (b → a → b) → b → [a] → [b]
scanl f z ls =
  z : (case ls of
         []   → []
         x:xs → scanl f (z `f` x) xs)
```

sequential dependencies

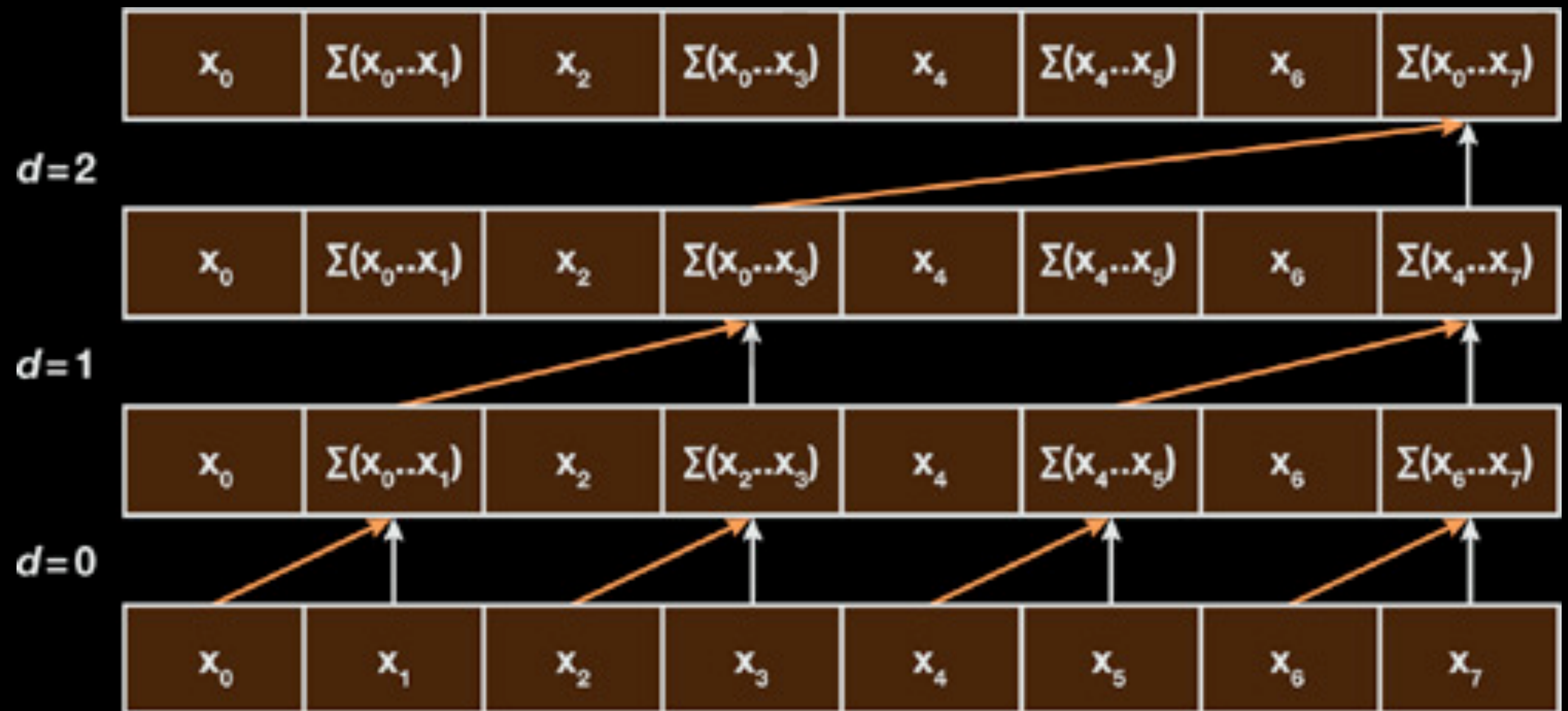
```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

CUDA C code

What is going on here?

source

Phase 1: upsweep (reduce)



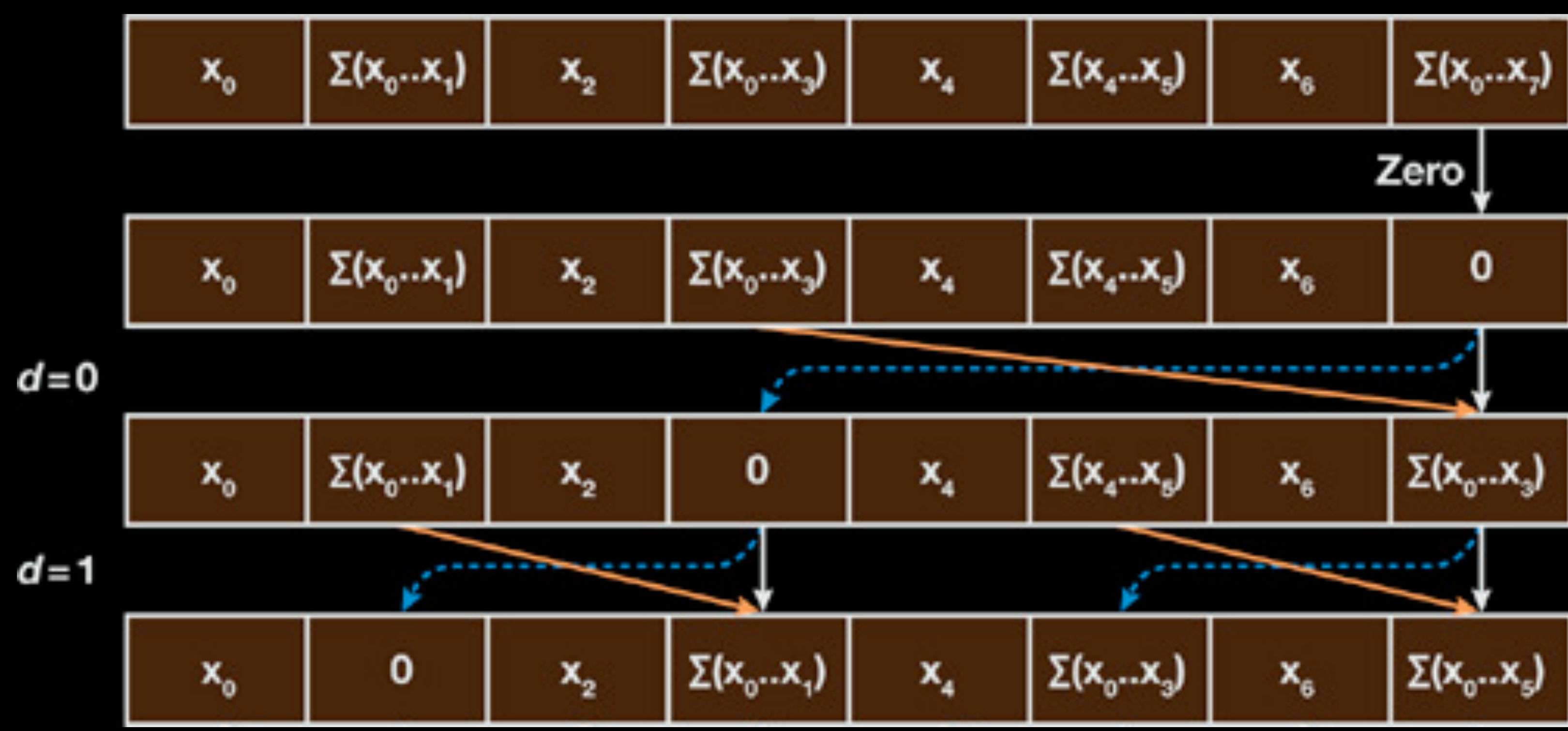
for $d = 0$ to $\log_2 n - 1$ **do**

  for all $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**

    $x[k + 2^{d+1} - 1] = x[k + 2^{d} - 1] + x[k + 2^{d+1} - 1]$

source

Phase 2: downsweep



$x[n-1] := 0$

for $d = \log_2 n - 1$ down to 0 **do**

  for all $k = 0$ to $n - 1$ by 2^{d+1} <u>in parallel</u> **do**

    $t = x[k + 2^{d-1}]$

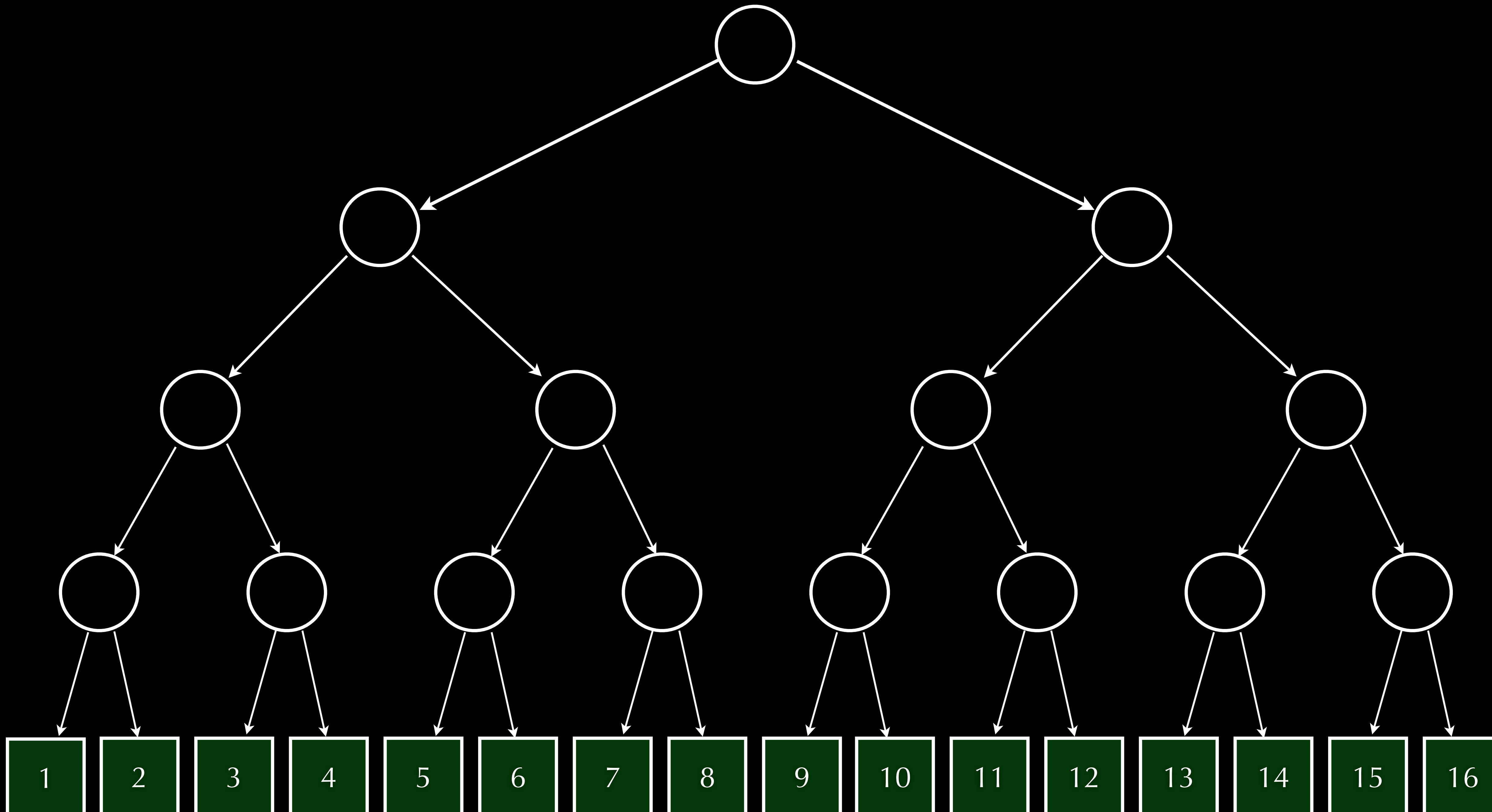
    $x[k + 2^{d-1}] = x[k + 2^{d+1} - 1]$

    $x[k + 2^{d+1} - 1] = t + x[k + 2^{d+1} - 1]$

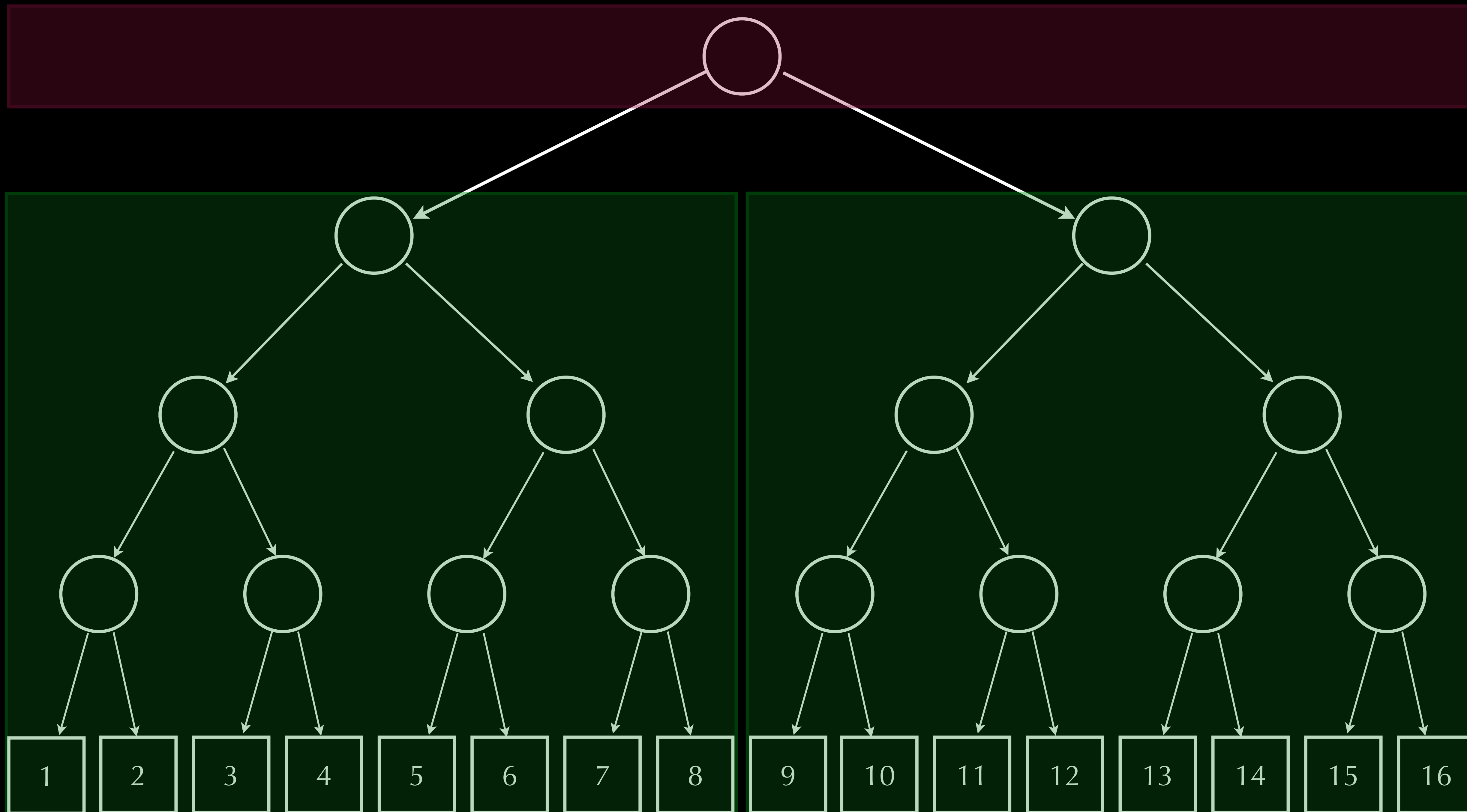
source

How would you parallelize scan?

(Assuming associativity.)

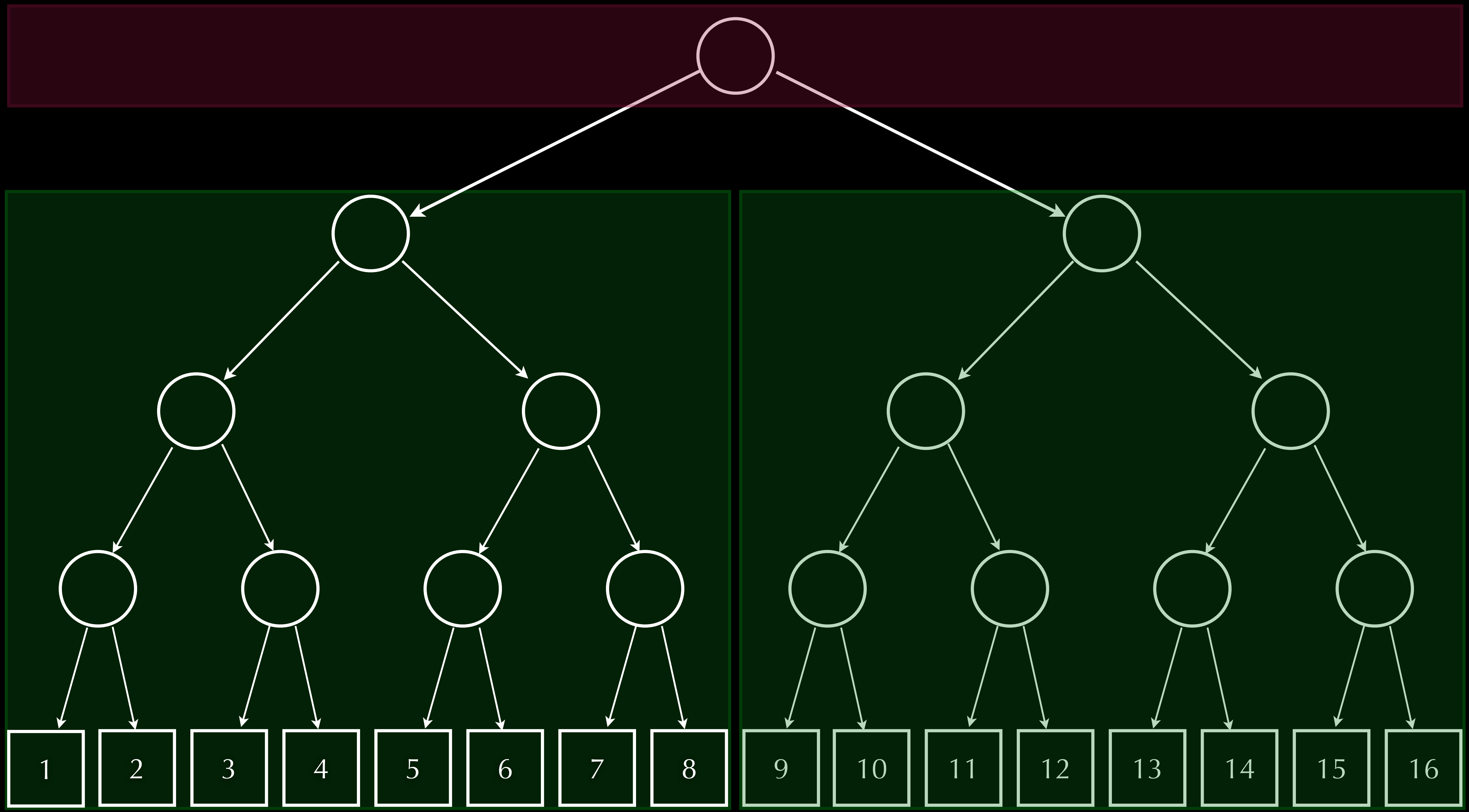


Divide and conquer ...



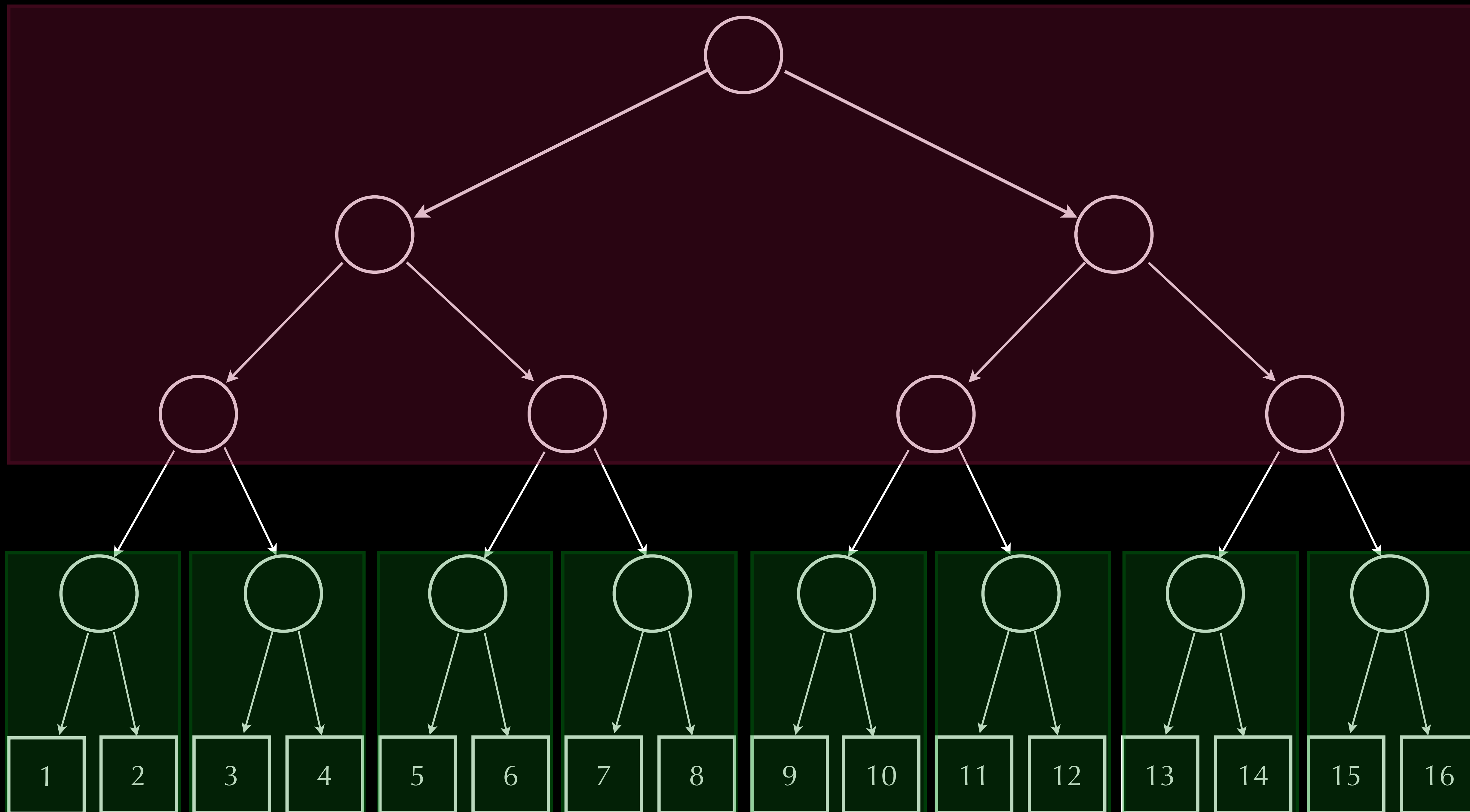
scan left ; scan right ; adjust right

Depth: log *n*; Work: *n* log *n*



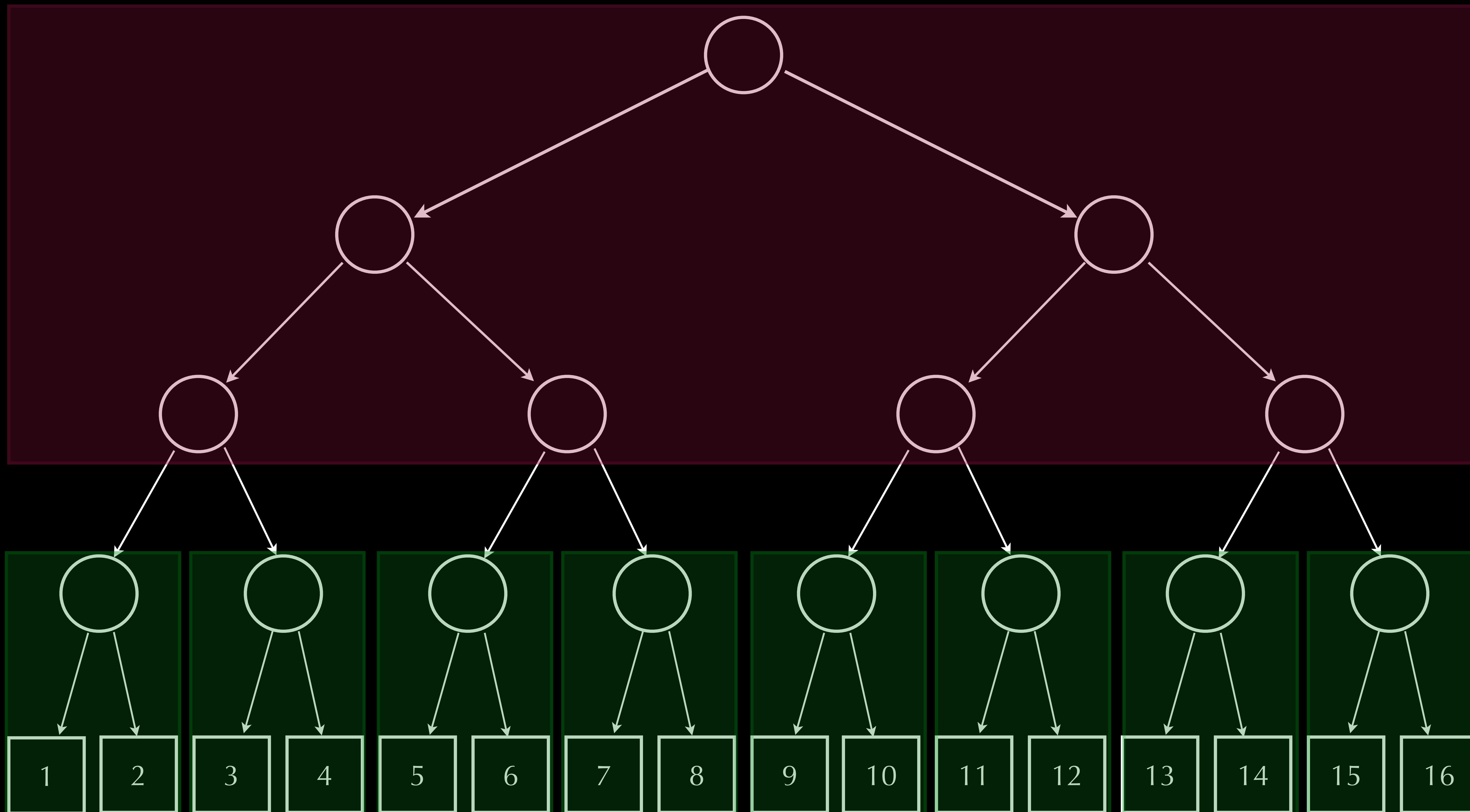

```
type Tree = Id + Pair ∘ Tree
```

A pair of trees, or … ?




```
type Tree = Id + Tree ∘ Pair
```

… a tree of pairs.



sum pairs ; scan ; adjust

Depth: log *n;* Work: *n*

```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

CUDA C code

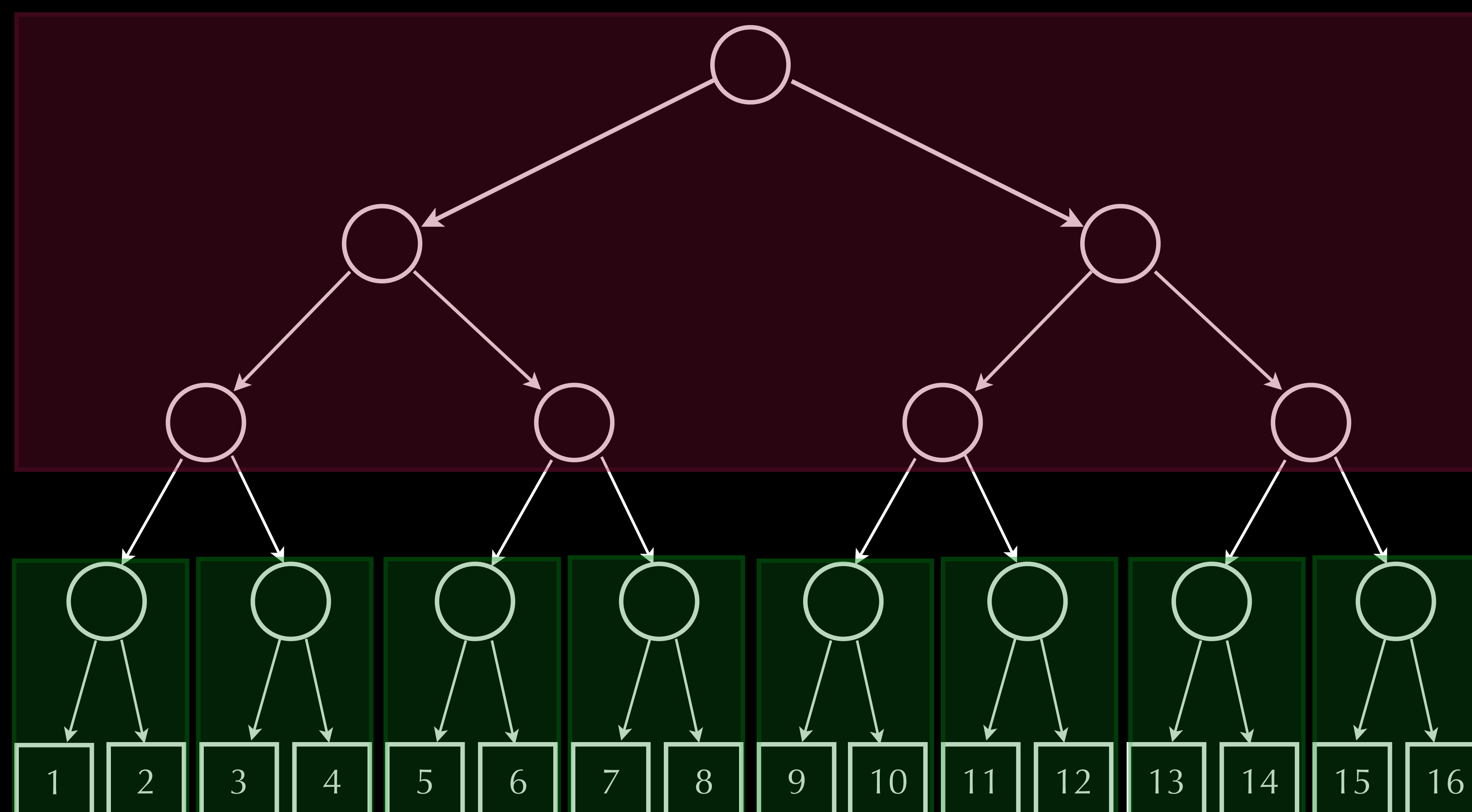
What is going on here?

[source]

```
__global__ void prescan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[];  // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    // load input into shared memory
    temp[2*thid] = g_idata[2*thid];
    temp[2*thid+1] = g_idata[2*thid+1];
    // build sum in place up the tree
    for (int d = n>>1; d > 0; d >>= 1) {
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset *= 2;
    }
    // clear the last element
    if (thid == 0) { temp[n - 1] = 0; }
    // traverse down tree & build scan
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (thid < d) {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }
    __syncthreads();
    // write results to device memory
    g_odata[2*thid] = temp[2*thid];
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

sum pairs ; scan ; adjust

- tail recursion
- in-place update

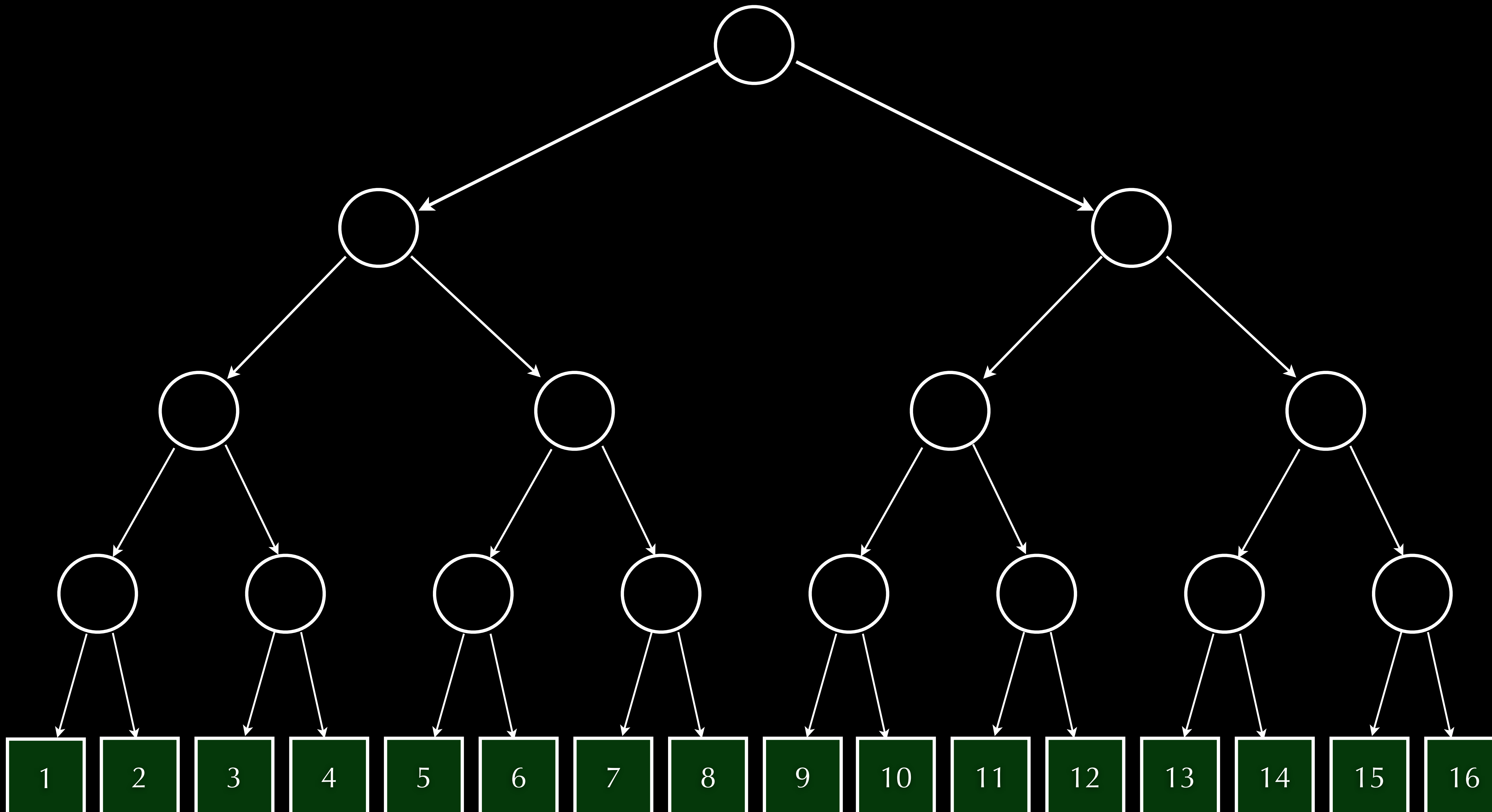


function scan(a) =
if #a == 1 then [0]
else
let e = even_elts(a);
o = odd_elts(a);
s = scan({e + o: e in e; o in o})
in interleave(s,{s + e: s in s; e in e});
	Work $= O(n)$
	Depth $= O(\log n)$

parallel prefix scan in NESL

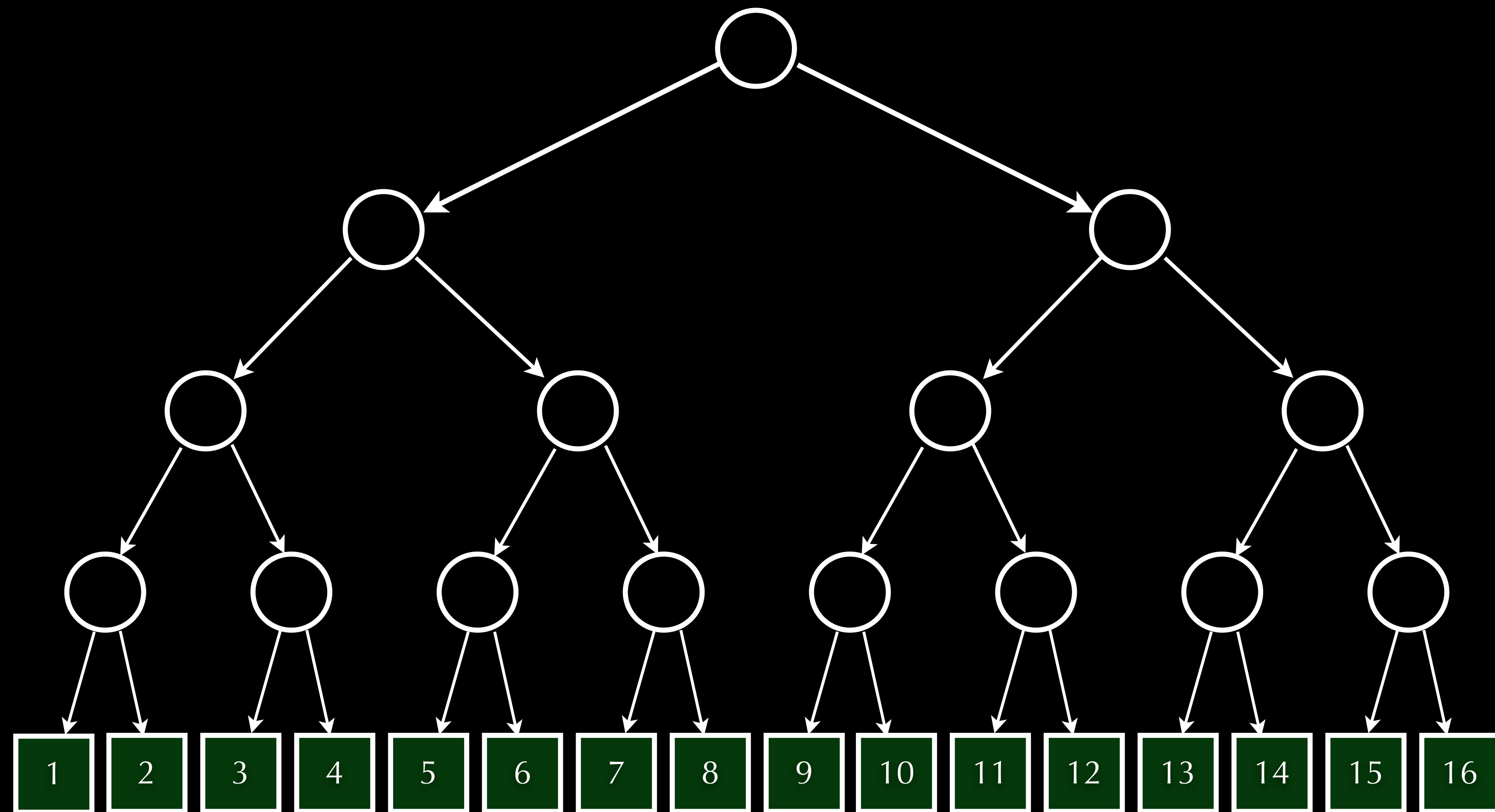
Source: *Programming parallel algorithms* (Blelloch 1990)

- Trees via functor composition
- Relate arrays to complete binary leaf trees, via vectors & tries
- Right- vs left-folding
- Static typing for size & depth
- From non-destructive "update" to in-place update via semantic editor combinators



What is the type of this tree?

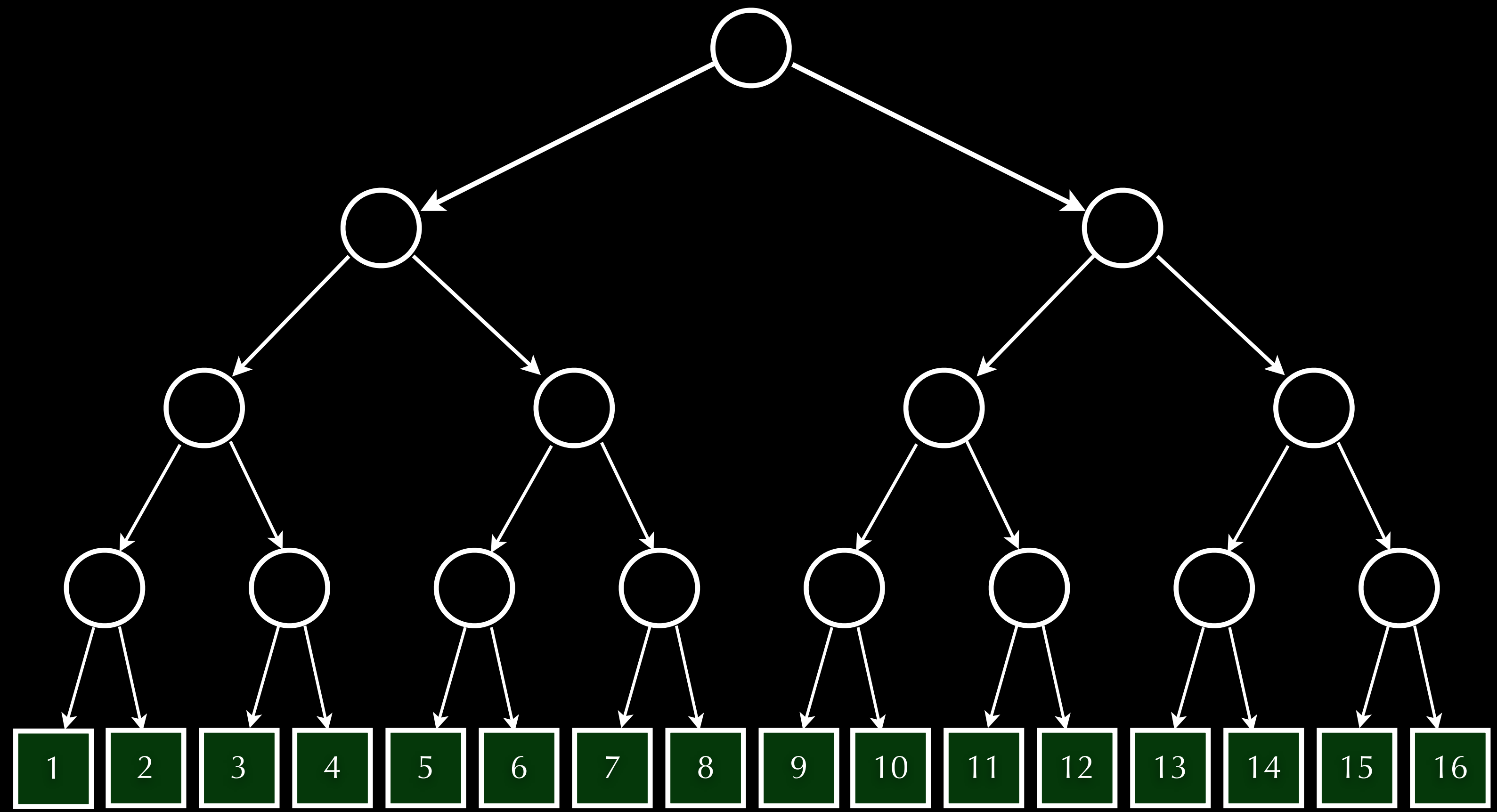
Enforce complete & depth 4



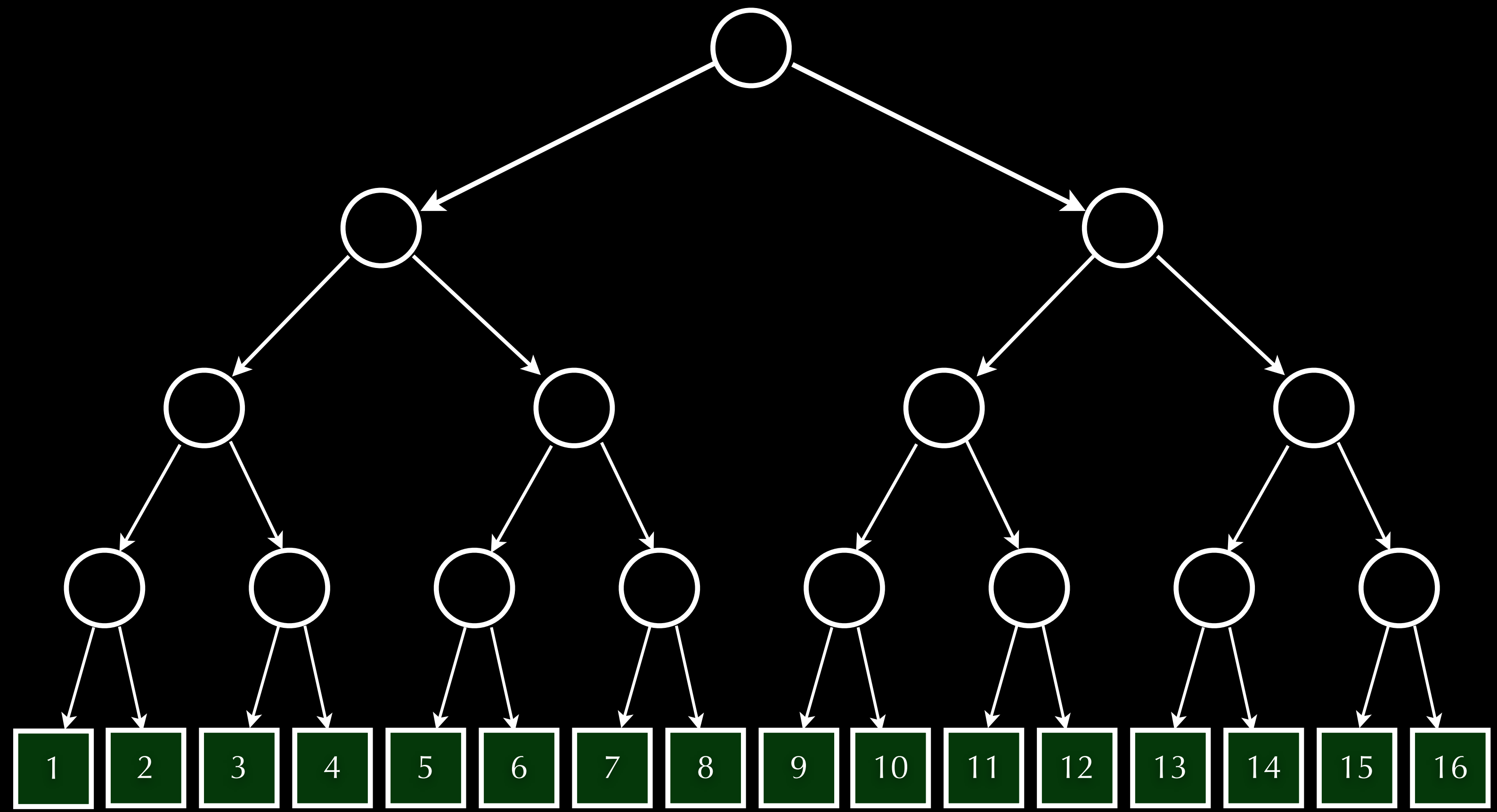
What is the type of this tree?

Enforce complete & depth 4

```
t :: Tree Four Integer
```

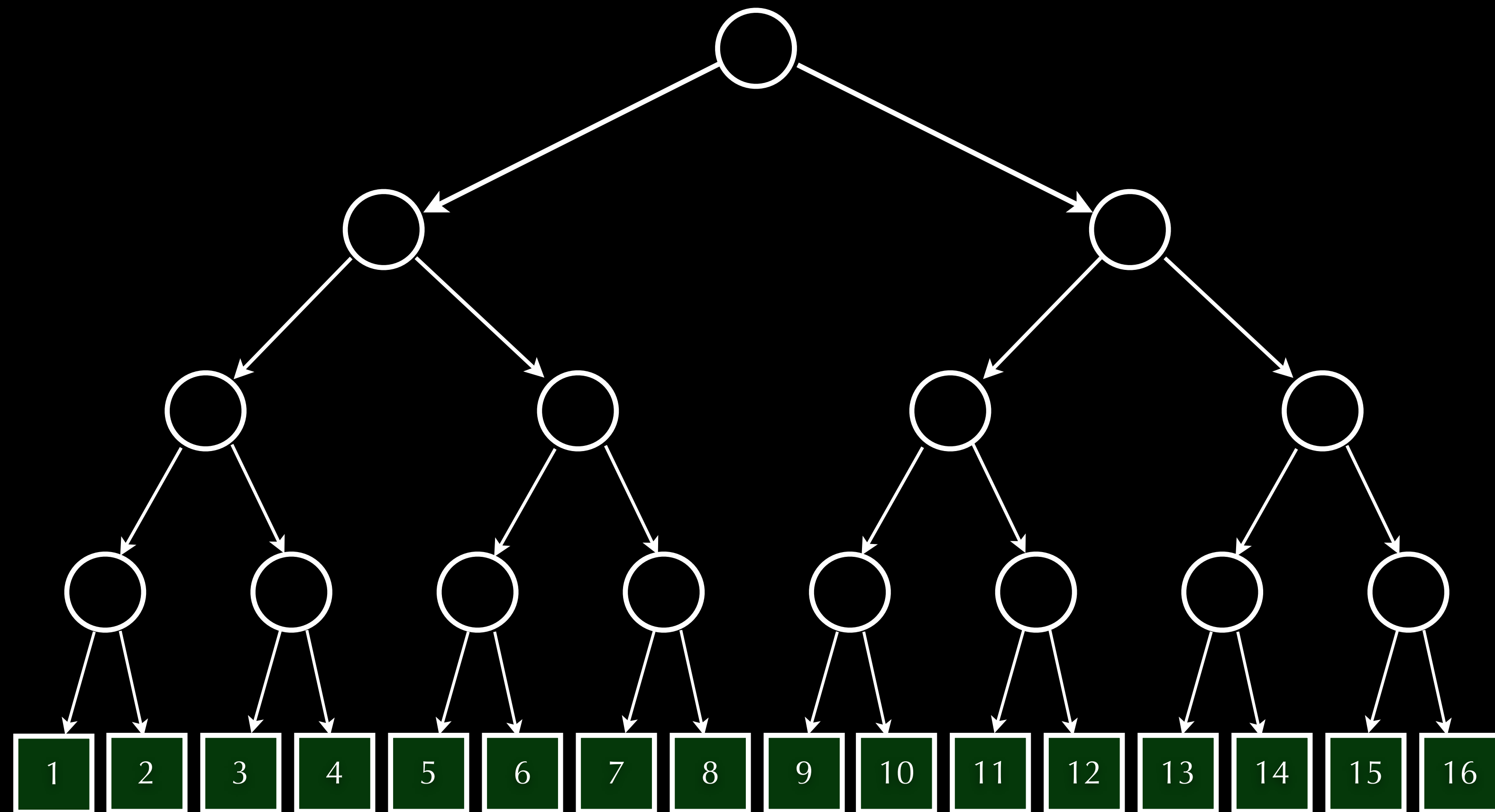



```
t :: Tree Four Integer
```

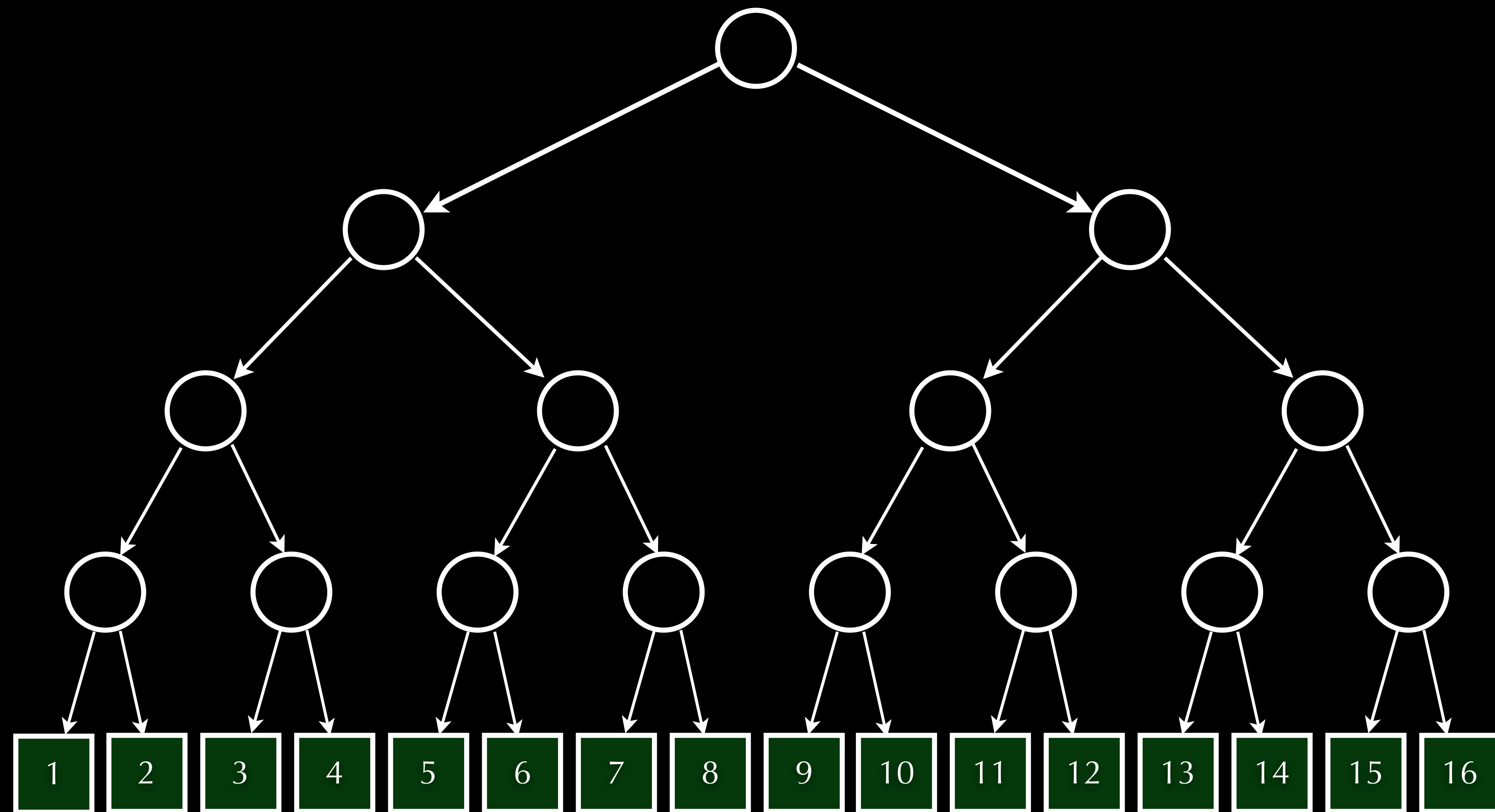



```
t :: Tree Four Integer

  :: (P ∘ P ∘ P ∘ P) Integer

data P a = a :# a
```

pair functor




```
t :: Tree Four Integer

  :: (P ∘ P ∘ P ∘ P) Integer

  :: (P ^ Four) Integer

Tree n = P ^ n

data P a = a :# a
```




```
t :: Tree Four Integer

Tree n = P ^ n

f ^ n ≅ f ∘ ⋯ ∘ f    -- (n times)

data P a = a :# a
```

`f ^ n ≅ f ∘ ⋯ ∘ f  --` (*n* times)

`f ^ n ≅ f ∘ ⋯ ∘ f   --` (*n* times)

Which way?

`f ↱ n = f ∘ (⋯ ∘ f)`   *right*-folded

`f ↰ n = (f ∘ ⋯) ∘ f`   *left*-folded

```
f ↱ n = f ∘ (⋯ ∘ f)
```

right-folded

```
f ↱ Z   ≅ Id
f ↱ S n ≅ f ∘ (f ↱ n)

data (↱) :: (* → *) → * → (* → *) where
  RL :: a  → (f ↱ Z) a
  RB :: IsNat n ⇒
    f ((f ↱ n) a) → (f ↱ (S n)) a
```

```
f ↰ n ≅ (f ∘ ⋯) ∘ f
```

left-folded

```
f ↰ Z   ≅ Id
f ↰ S n ≅ (f ↰ n) ∘ f
```

```
data (↰) :: (* → *) → * → (* → *) where
  LL :: a  → (f ↰ Z) a
  LB :: IsNat n ⇒
     (f ↰ n) (f a) → (f ↰ (S n)) a
```

```
function scan(a) =
if #a == 1 then [0]
else
   let e = even_elts(a);
       o = odd_elts(a);
       s = scan({e + o: e in e; o in o})
   in interleave(s,{s + e: s in s; e in e});
```

parallel prefix scan in NESL

Source: *Programming parallel algorithms* (Blelloch 1990)

```
scanL [_] = [0]
scanL xs = interleave s (s +* e)
 where
   (e,o) = uninterleave xs
   s     = scanL (e +* o)

(+*) = zipWith (+)
```

```
scan (L _ ) = L 0
scan (B as) = B (invert (ss :# ss + es))
 where
   (es :# os) = invert as
   ss         = scan (es + os)

invert :: (Traversable f, Applicative g)
       ⇒ f (g a) → g (f a)

invert = sequenceA
```

```
scan = inT (const 0) (inInvert h)
 where
   h (es :# os) = (ss :# ss + es)
    where ss = scan (es + os)
```

```
scan = inT (const 0) (inInvert h)
 where
   h (es :# os) = (ss :# ss + es)
    where ss = scan (es + os)
```

Note final uses. Can overwrite.

Wasteful zipping & unzipping.

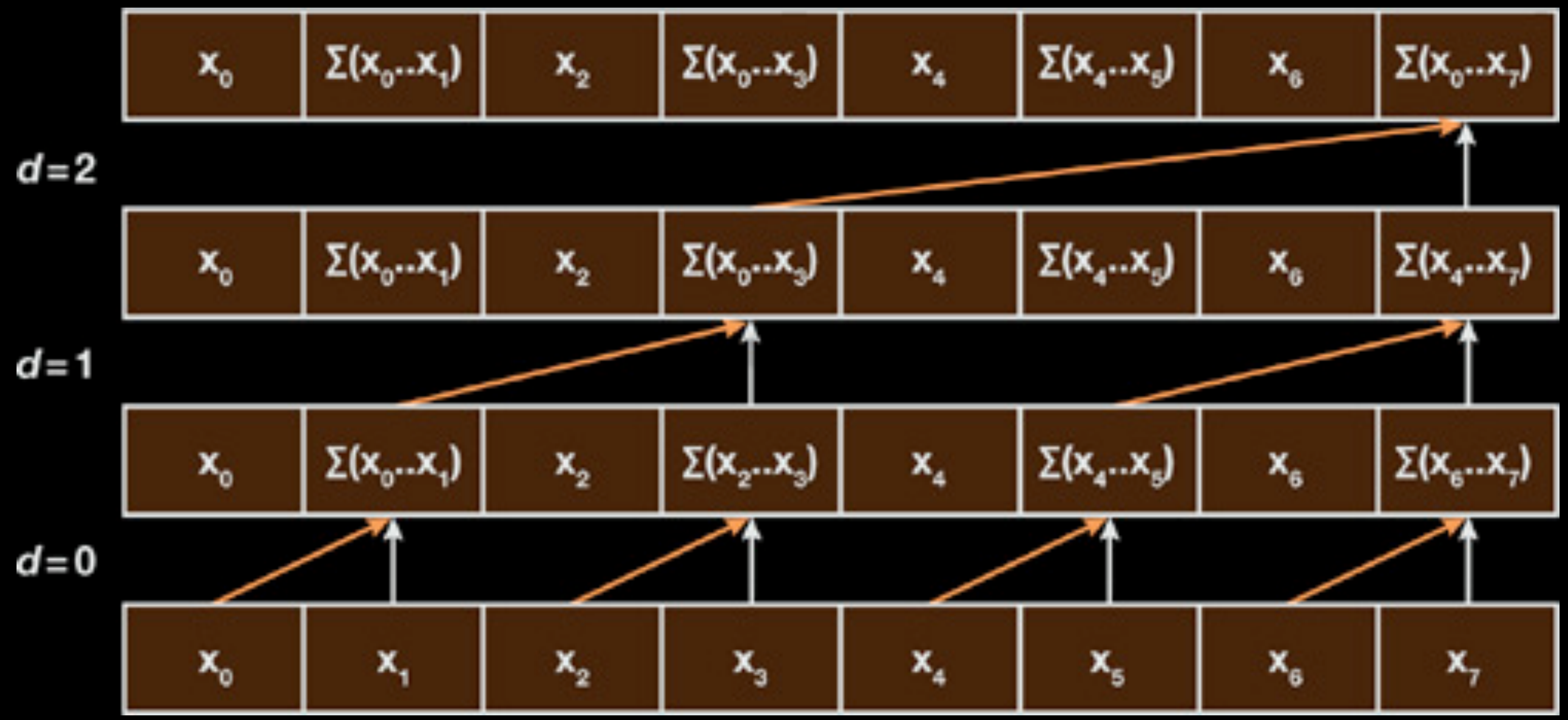
```
scan = inT (const 0)
           ( fmap after
           ∘ seconds scan
           ∘ fmap before
           )

before (e :# o) = (e :# e+o)
after  (e :# s) = (s :# s+e)

seconds = inInvert ∘ second
```

downsweep

spread

upsweep

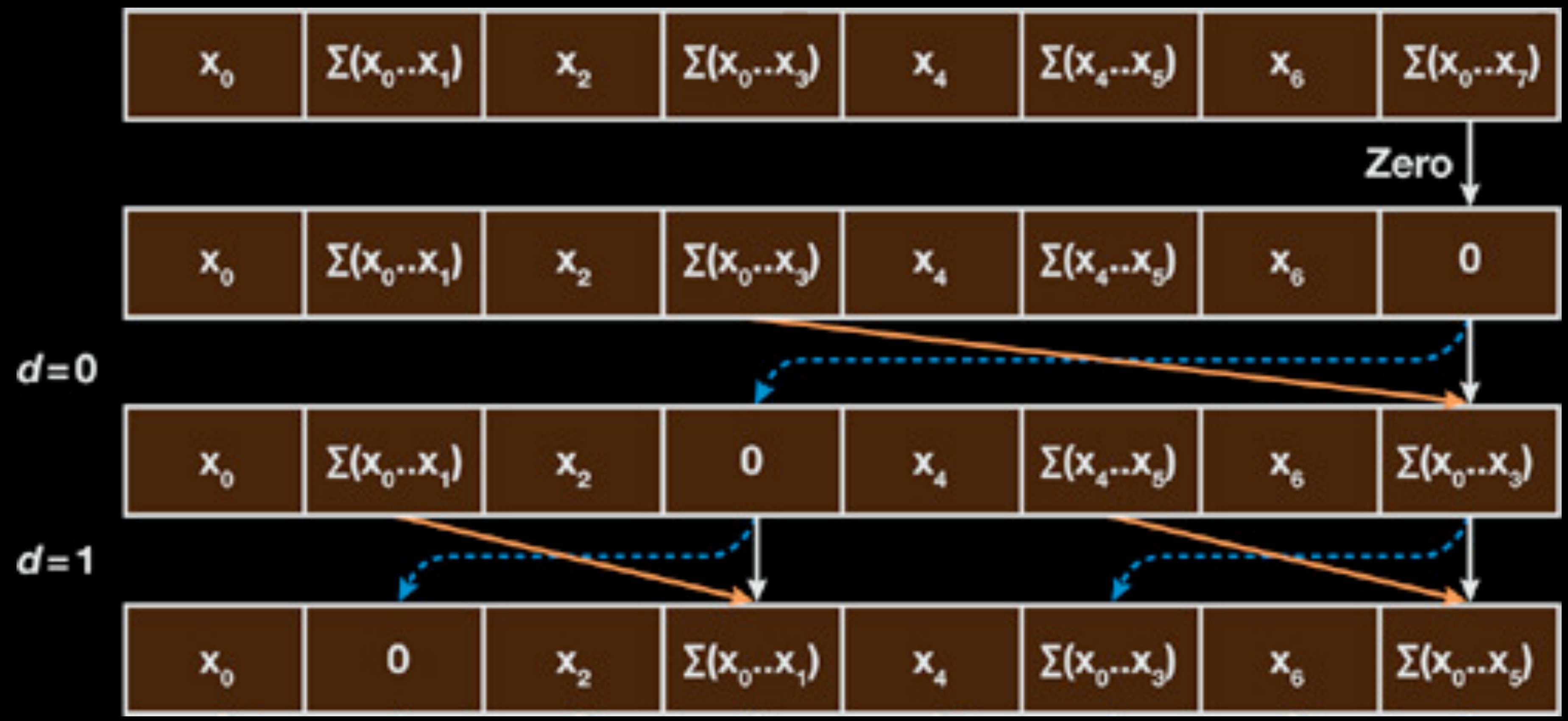
Phase 1: upsweep (reduce)




```
before (e :# o) = (e :# e+o)
```

source

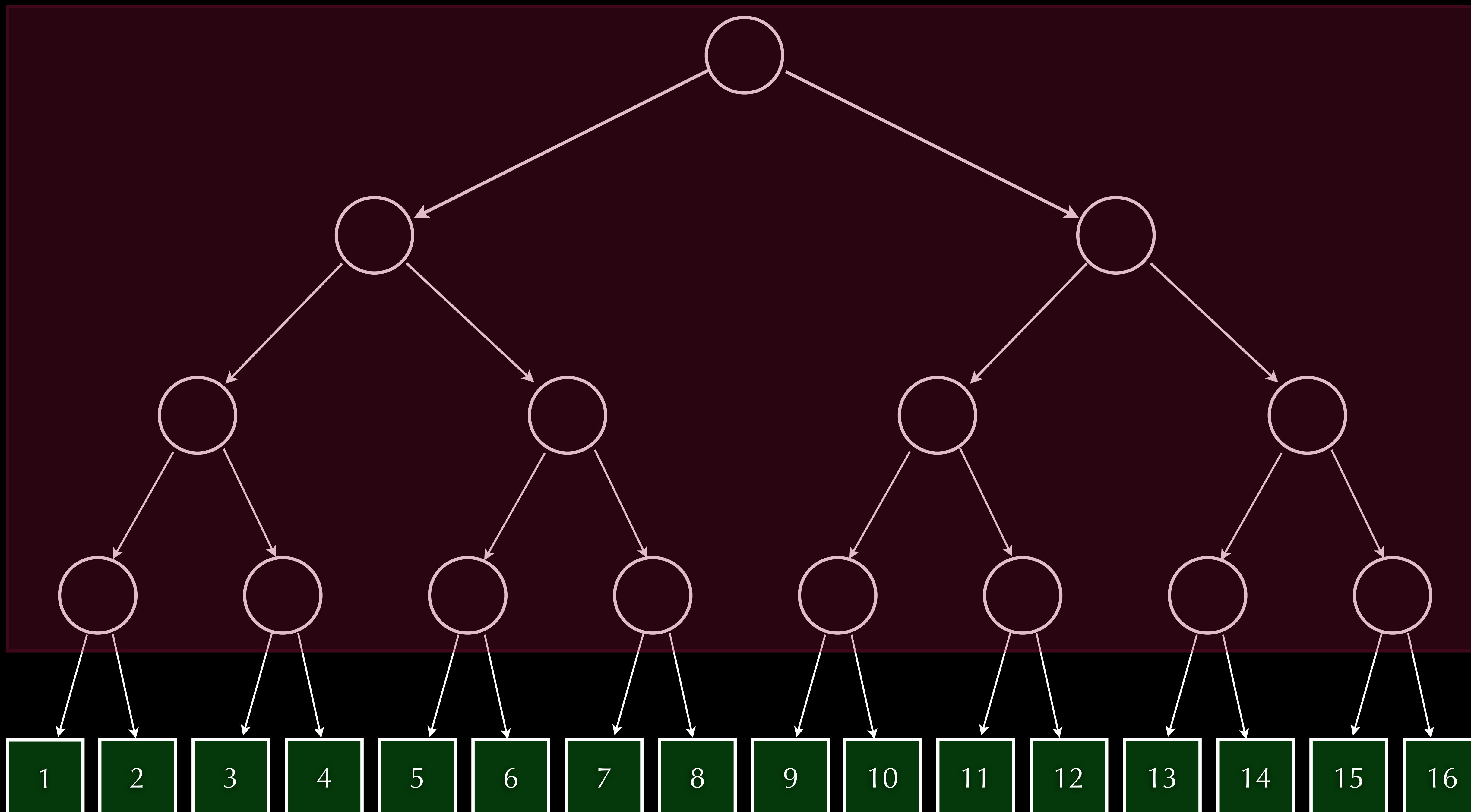
Phase 2: downsweep

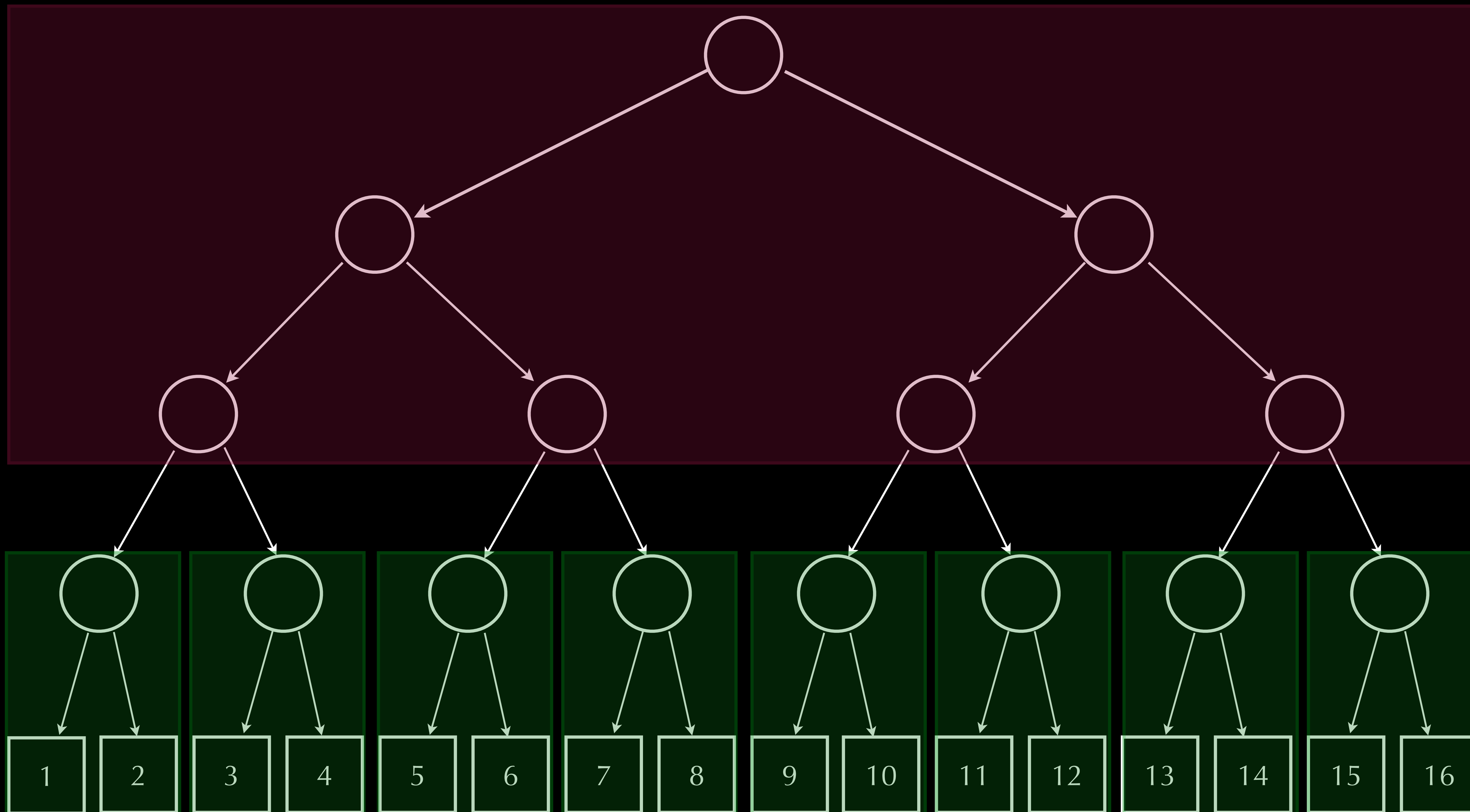


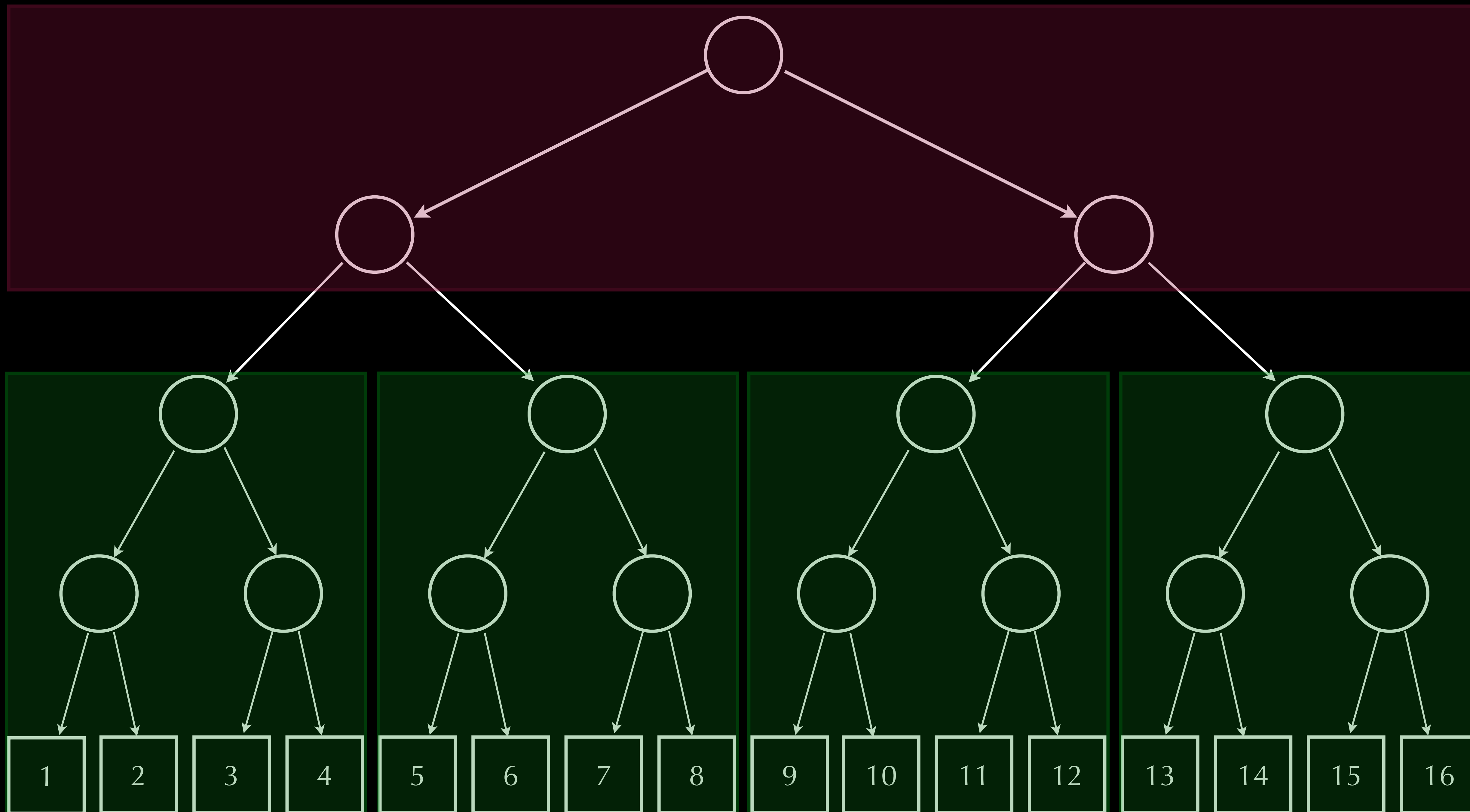

```
after  (e :# s) = (s :# s+e)
```

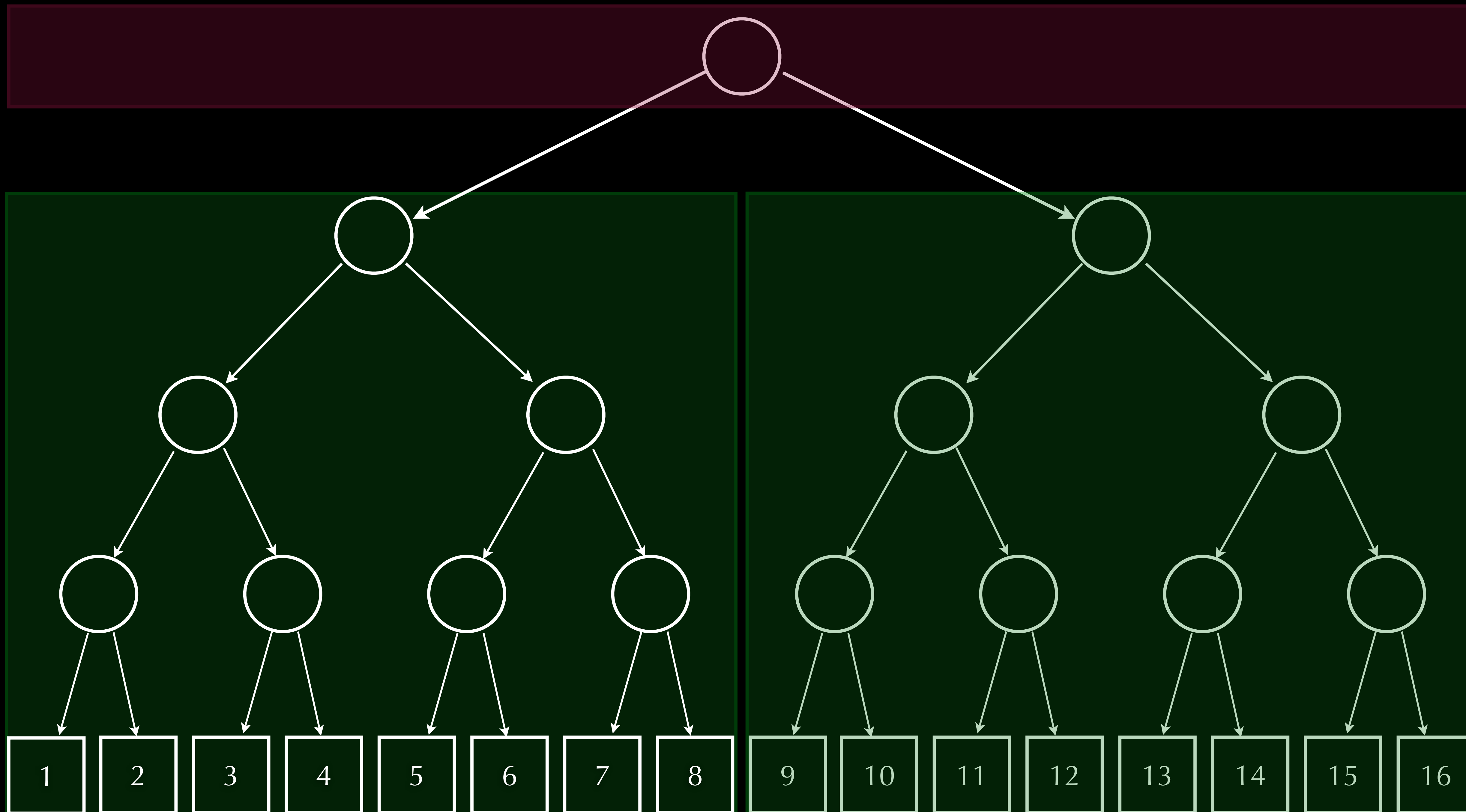
source

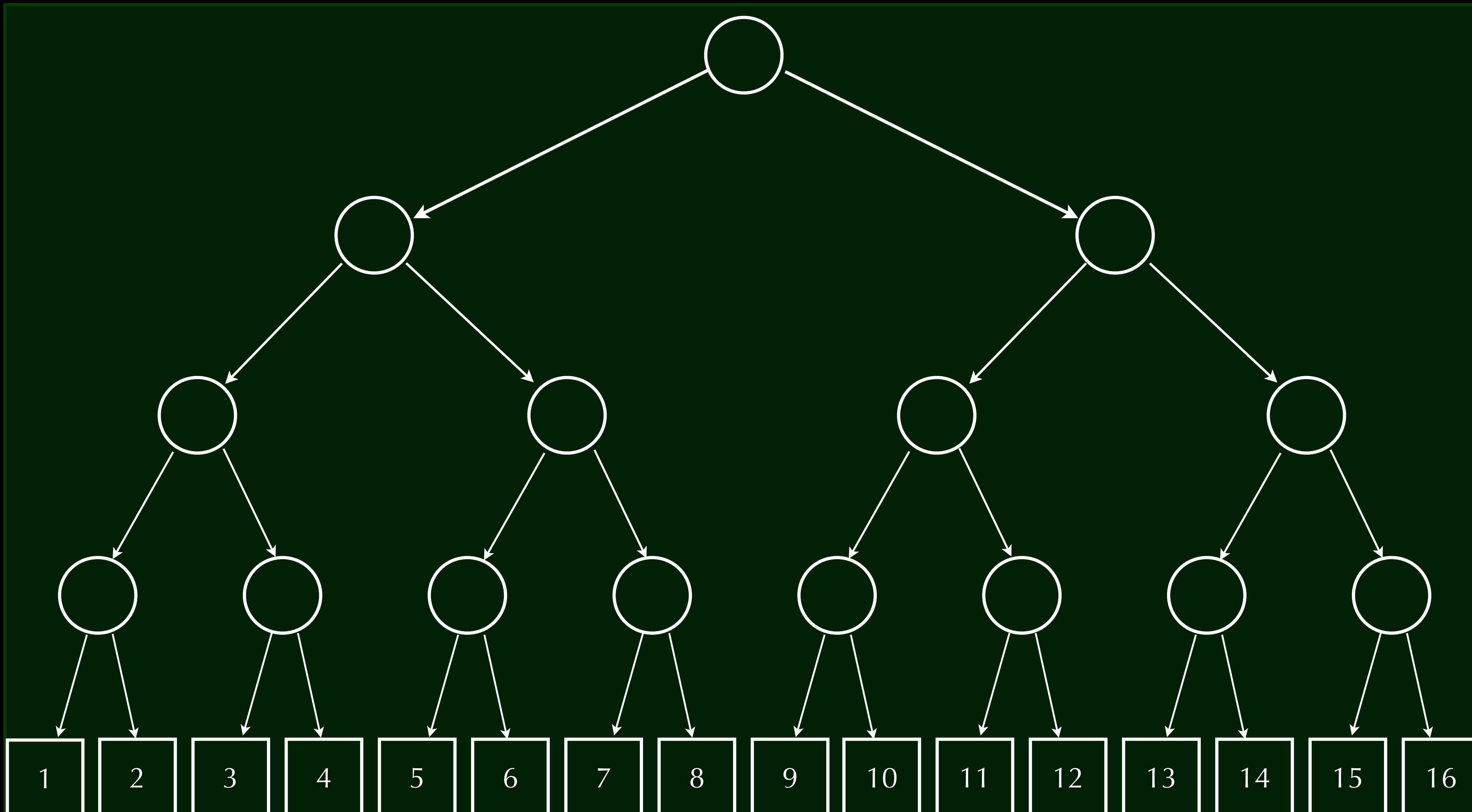
Next: flatten into a fmap chain

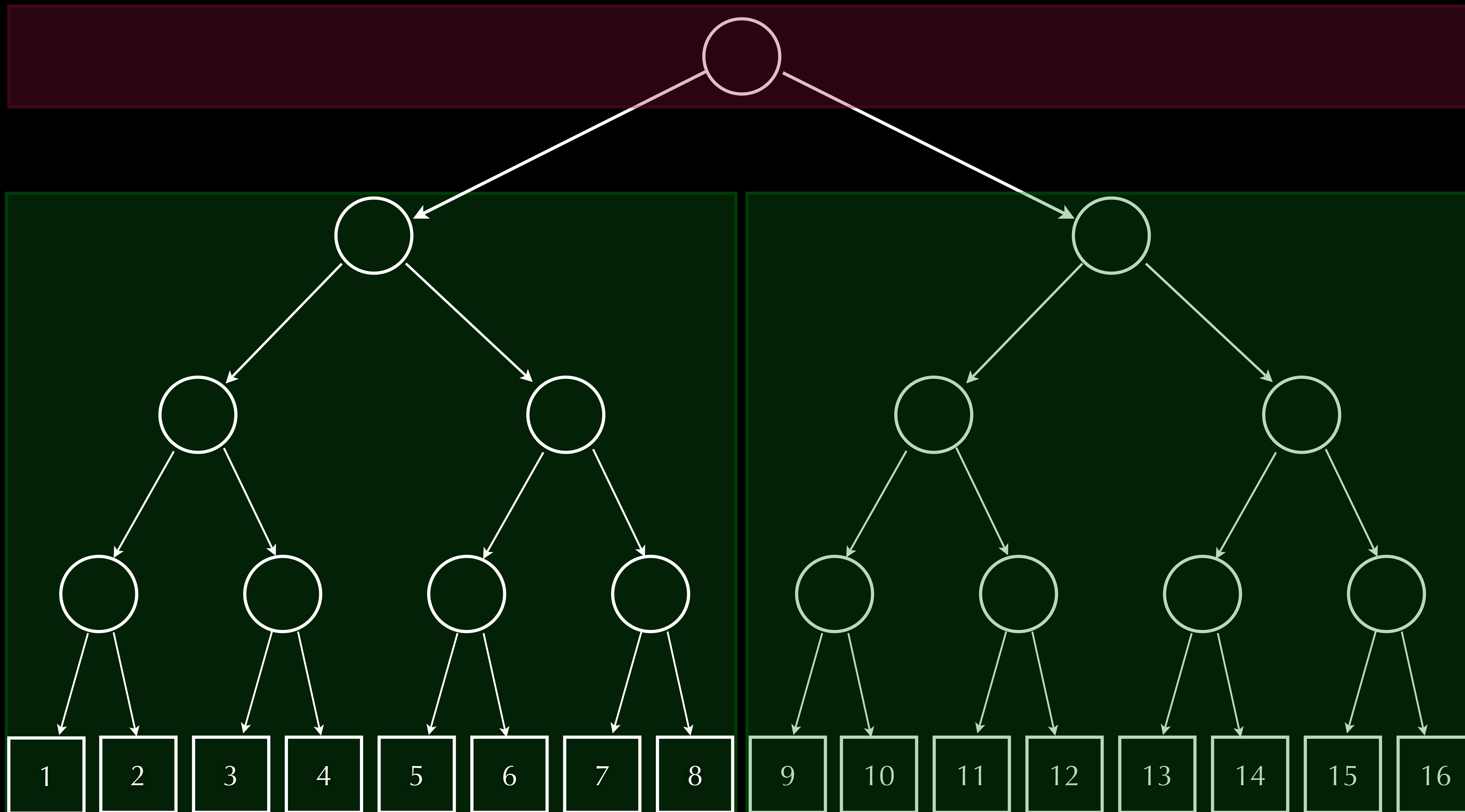

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

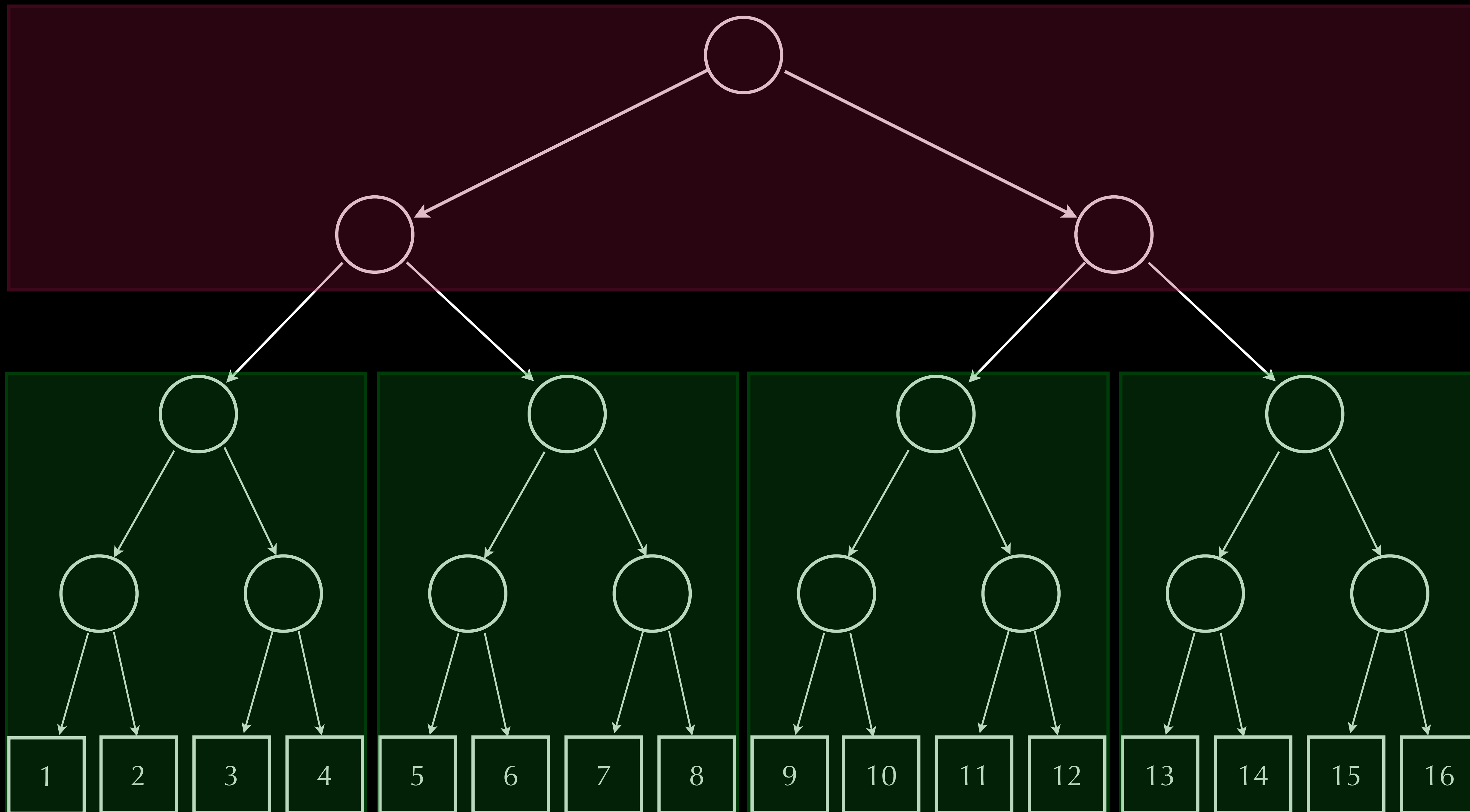

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

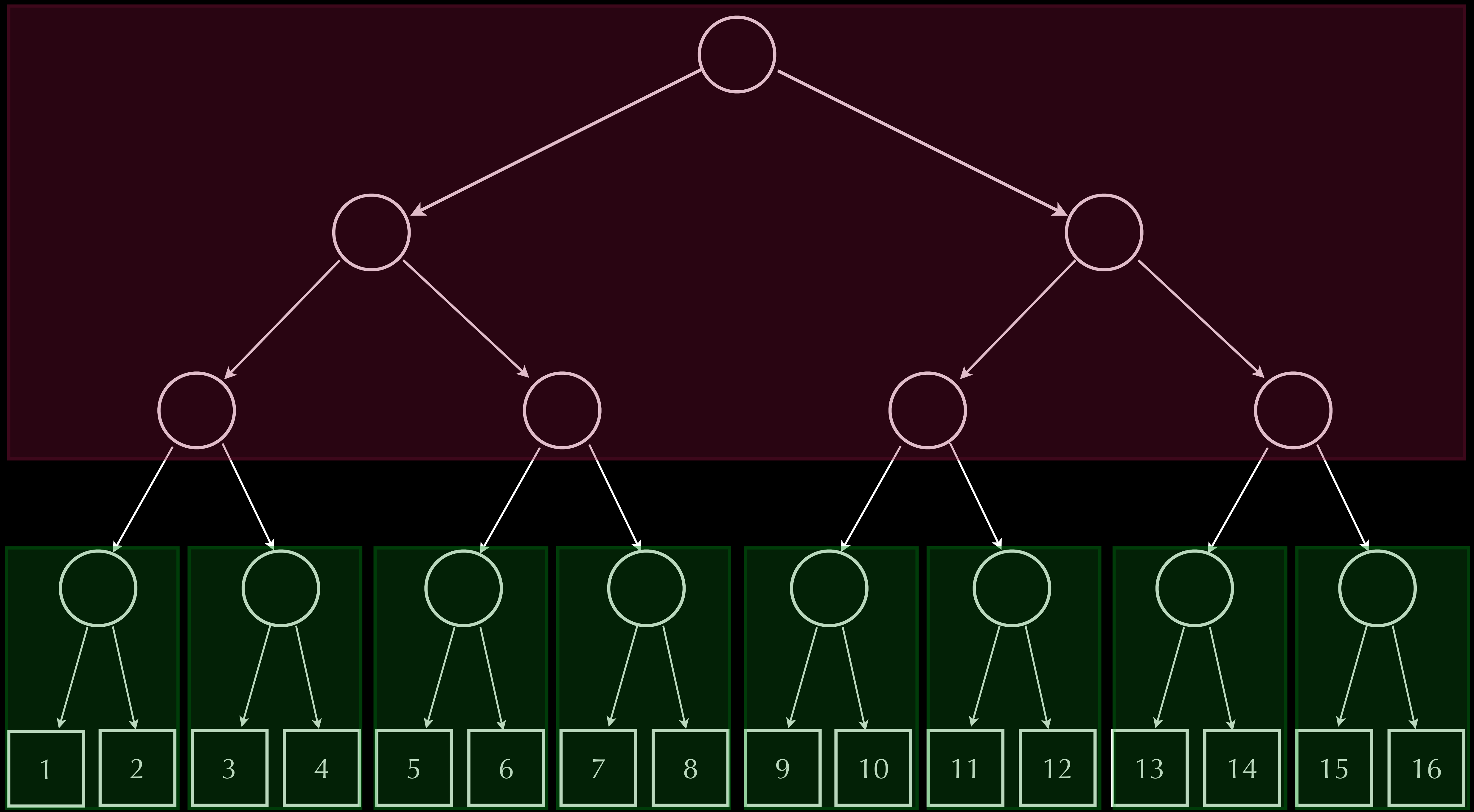

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

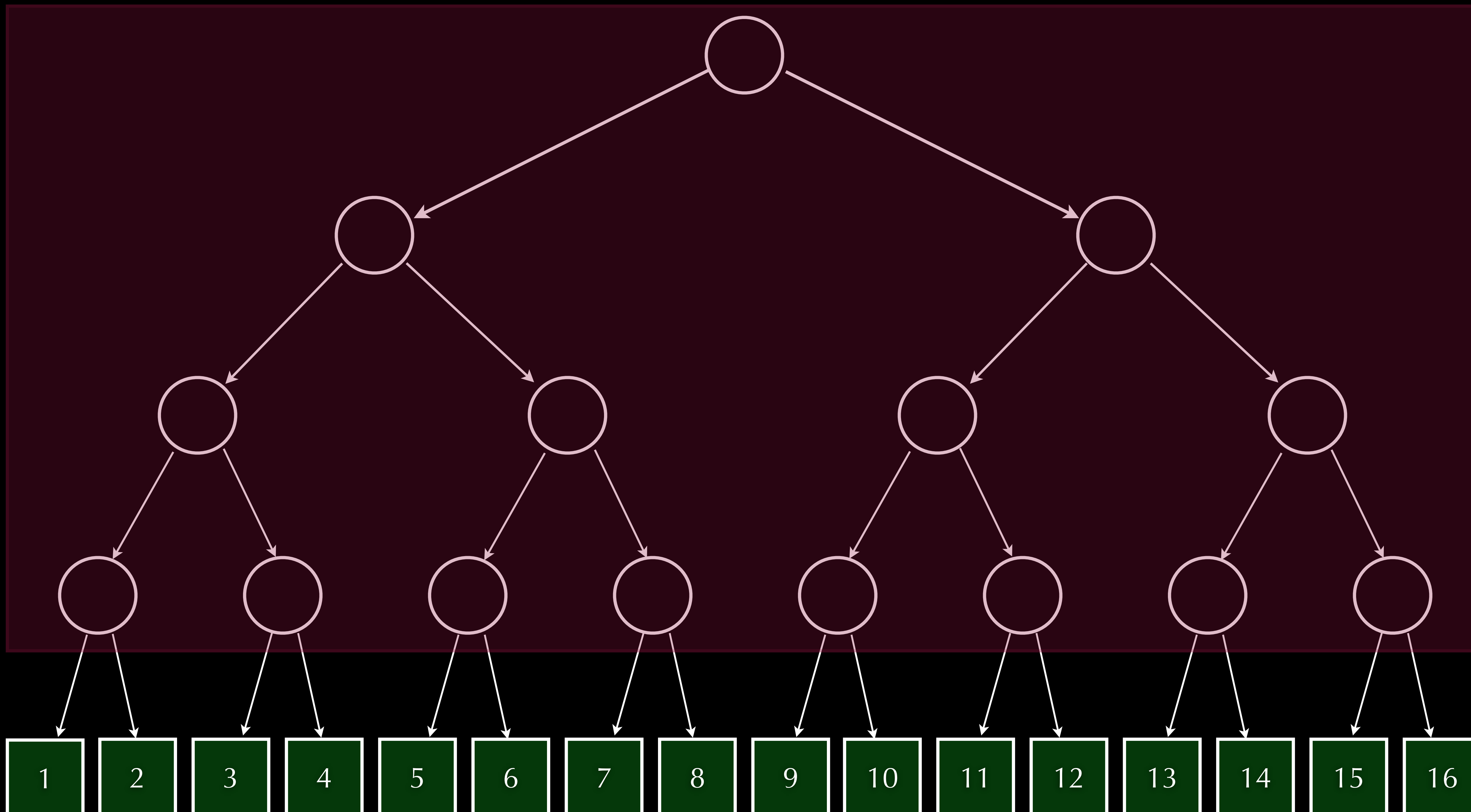

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16



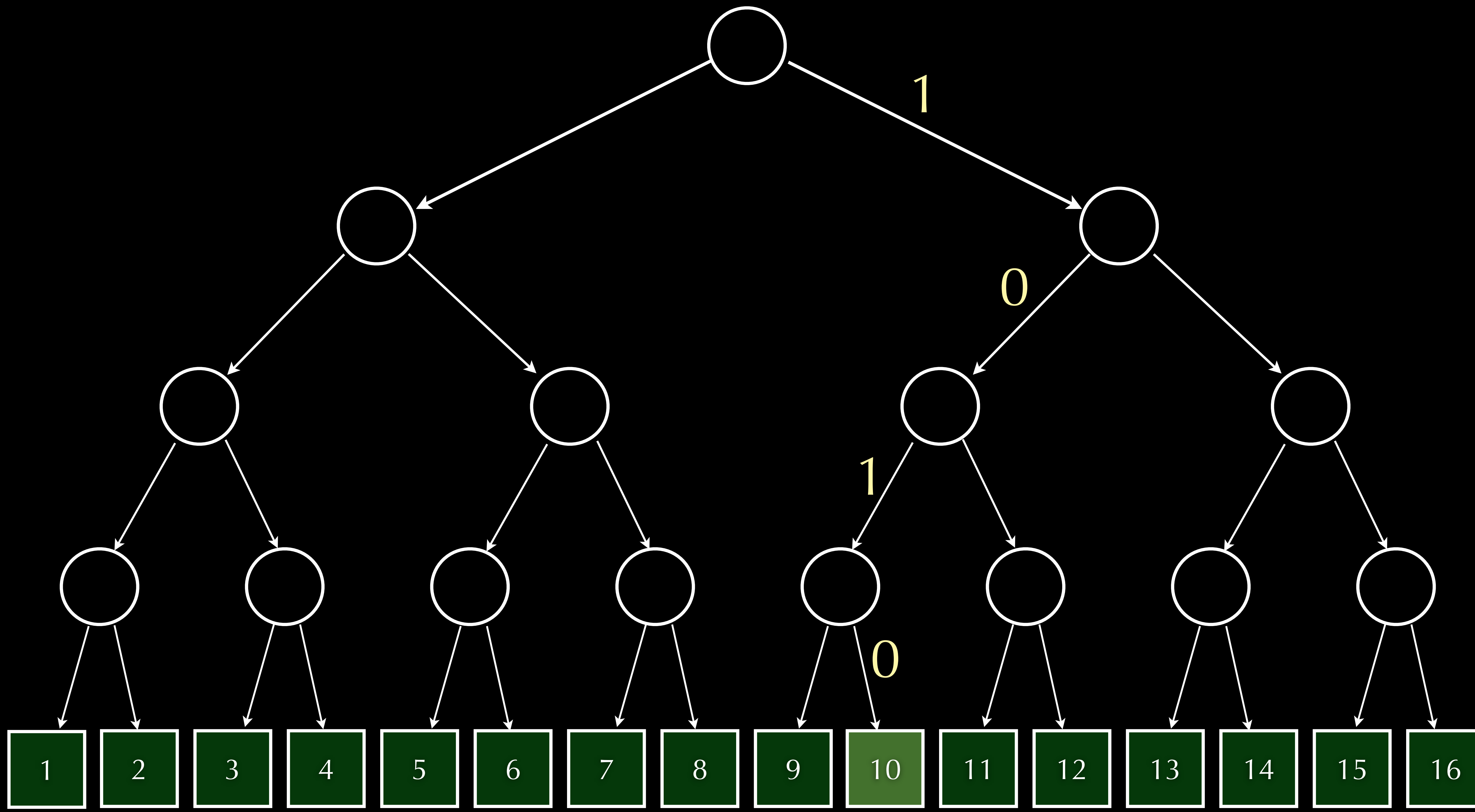
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

```
up   :: (Functor f, IsNat m)
      ⇒ (f ↰ S n) ((f ↱ m) a)
      → (f ↰ n) ((f ↱ (S m)) a)
up   = fmap RB ∘ unLB

down :: (Functor f, IsNat n)
      ⇒ (f ↰ n) ((f ↱ (S m)) a)
      → (f ↰ S n) ((f ↱ m) a)
down = LB ∘ fmap unRB
```

Next, relate these trees to arrays.



Binary trees have binary indices.

Are binary *trees* binary *tries*?

Are binary *trees* binary *tries*?

```
type Binary n = Vec n Bool

Binary n → b
    ≡ Vec n Bool → b
     ≡ Vec n Bool → b
     ≅ (Bool × ⋯ × Bool) → b
     ≅ Bool → ⋯ → Bool → b
     ≡ Trie Bool (⋯ (Trie Bool b)⋯)
     ≅ (Trie Bool ∘ ⋯ ∘ Trie Bool) b
     = (Trie Bool ^ n) b
```

Are binary *trees* binary *tries*?

```
Binary n → b ≅ (Trie Bool ^ n) b

Trie Bool ≡ P

Binary n → b ≅ (P ^ n) b
             ≅ Tree n


Tree n ≅ Trie (Binary n)
```

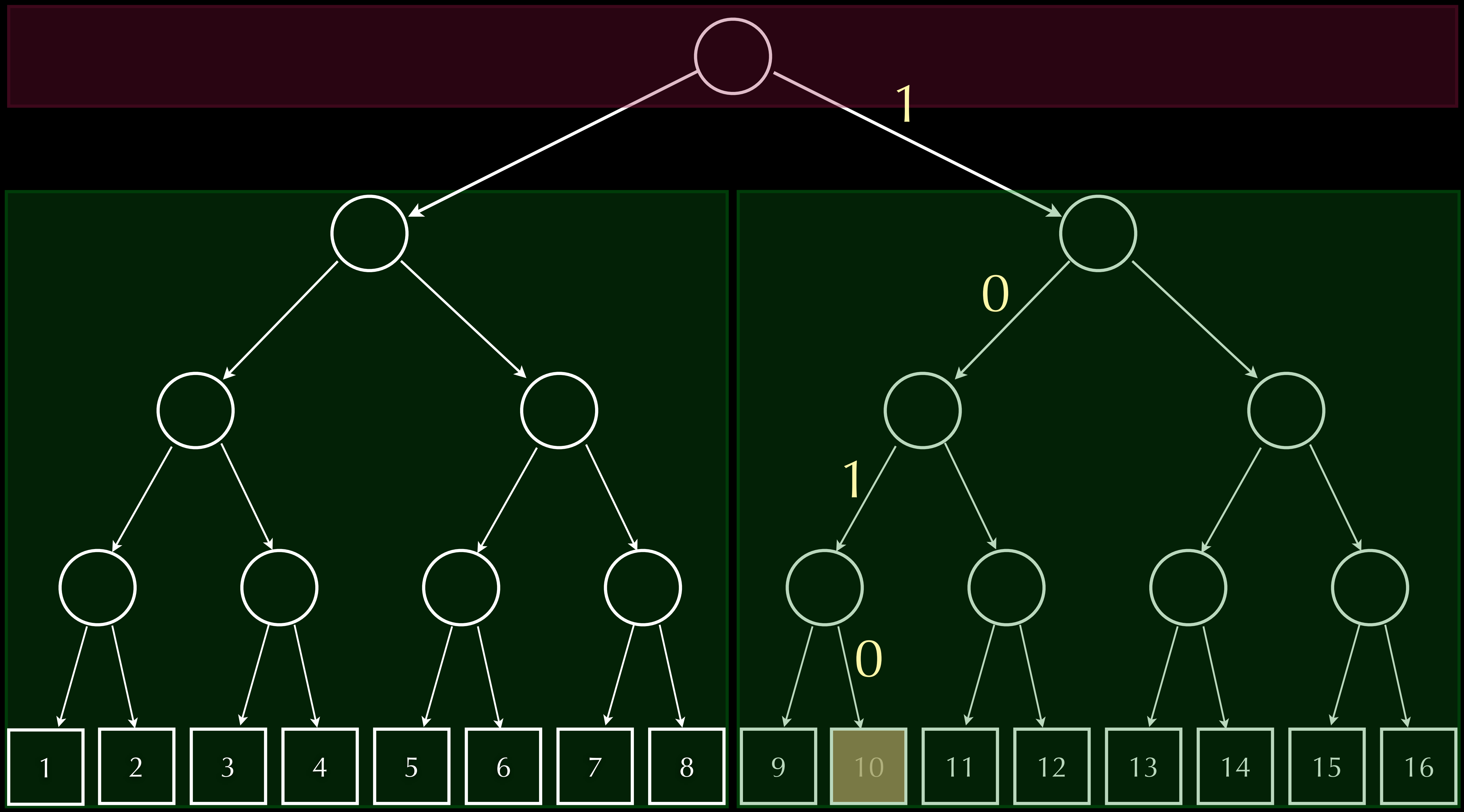
Yes!

Are binary *trees* binary *tries*?

```
Tree n ≅ Trie (Binary n)
```

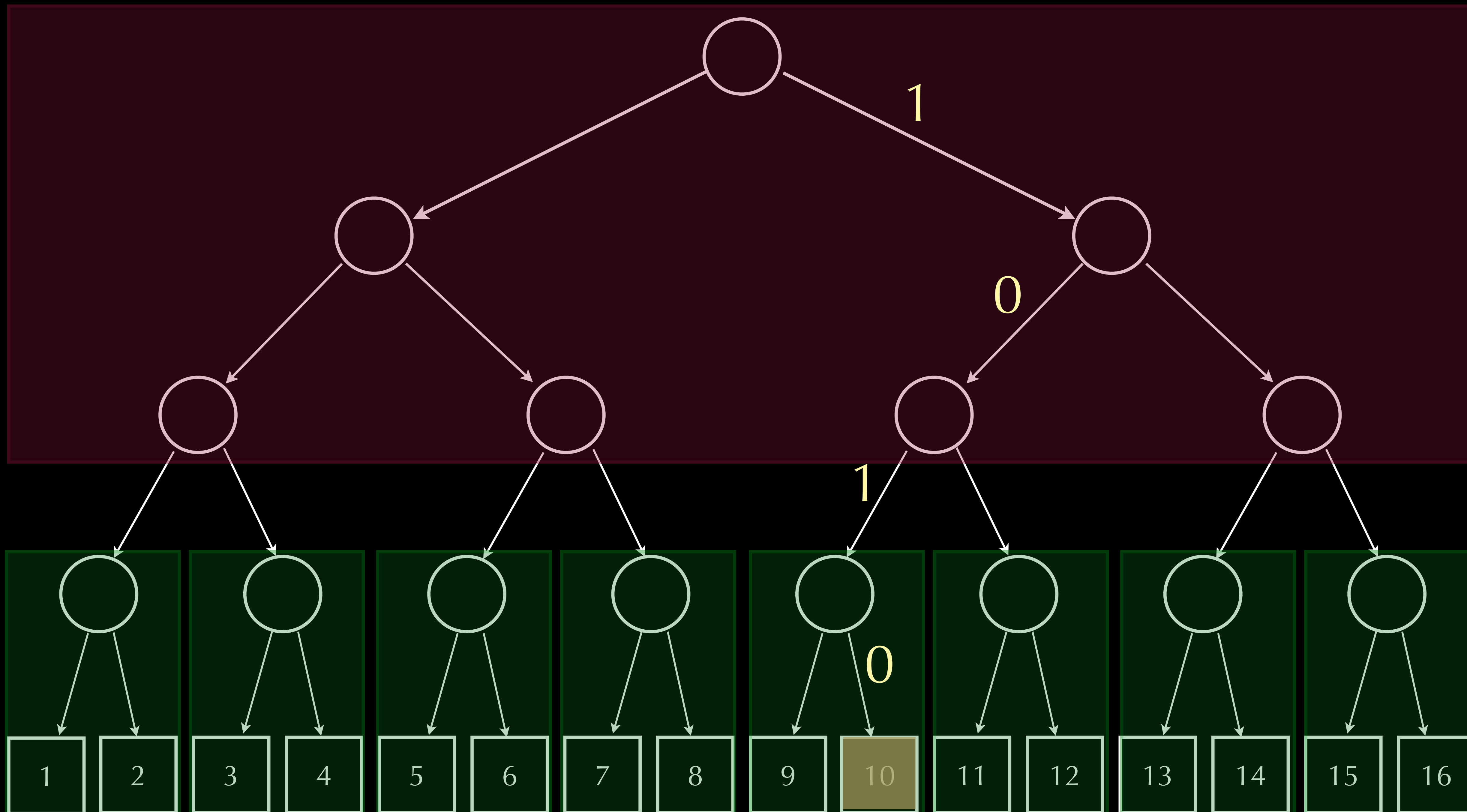
Generalizing:

```
Trie d ^ n = Trie (Vec n d)
```



Right-folded: pair of trees

Big endian indices



Left-folded: pair of trees

Little endian indices