# Foundations of Computer Science

## Functional arrays

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

**Arrays** are . . .

. . . an **indexed storage area** for values

. . . a very **common data structure** alongside lists and trees in most languages.

. . . **usually updated in-place** and are imperative or mutable data structures.

. . . **used in many classic algorithms** such as the original Hoare in-place partition-sort.

Arrays are an indexed storage area for values

| | |
|---|---|
| **list** | elements reached by counting from the head of the list |
| **tree** | elements reached by following a path from the root |
| **array** | elements uniformly designated by number (the "subscript") |

Arrays are an indexed storage area for values

Let's first consider **immutable arrays**

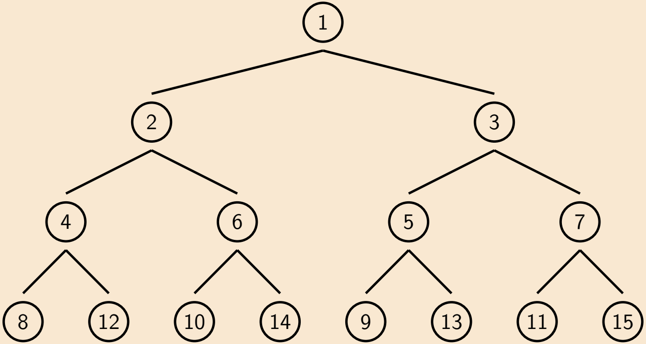
Immutable arrays are also known as **functional arrays**; they map integers to data.

$$\begin{array}{lcl} 1 & \mapsto & \text{"Orange"} \\ 2 & \mapsto & \text{"Apple"} \\ 3 & \mapsto & \text{"Banana"} \end{array}$$

**Updating implies copying** the array to return a new version, (but pointers to old copies remain).

Can updates be **efficient**?

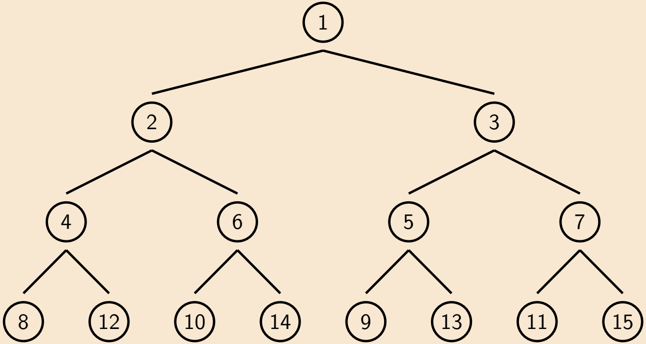
The path to element i follows the **binary code** for i (the "subscript")




(The numbers above are not the values, but the positions of array elements.)

**Complexity** of access to this is always $O(\log n)$ as the tree is always **balanced**.

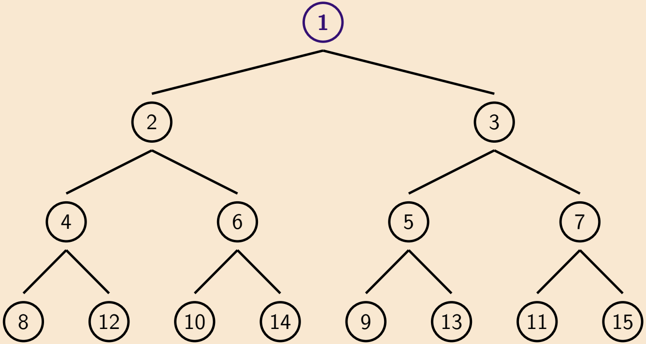
The path to element i follows the **binary code** for i (the "subscript")




Example: `sub t 5`

```
let rec sub = function
| Lf, _ -> raise Subscript
| Br (v, t1, t2), 1 -> v
| Br (v, t1, t2), k when k mod 2 = 0 -> sub (t1, k / 2)
| Br (v, t1, t2), k -> sub (t2, k / 2)
```

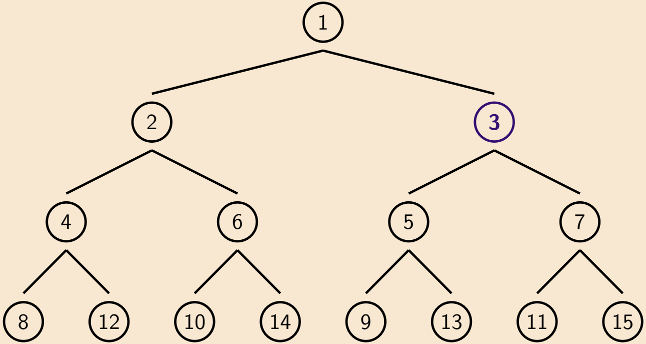
The path to element i follows the **binary code** for i (the "subscript")




Example: sub t 5

```
let rec sub = function
| Lf, _ -> raise Subscript
| Br (v, t1, t2), 1 -> v
| Br (v, t1, t2), k when k mod 2 = 0 -> sub (t1, k / 2)
| Br (v, t1, t2), k -> sub (t2, k / 2)
```

The path to element i follows the **binary code** for i (the "subscript")

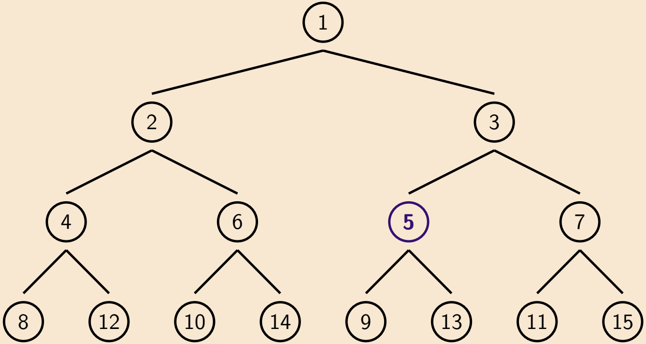



Example: sub t 5

$5 \,/\, 2 \leadsto 2$

```
let rec sub = function
| Lf, _ -> raise Subscript
| Br (v, t1, t2), 1 -> v
| Br (v, t1, t2), k when k mod 2 = 0 -> sub (t1, k / 2)
| Br (v, t1, t2), k -> sub (t2, k / 2)
```

The path to element i follows the **binary code** for i (the "subscript")




Example: `sub t 5`

$5 \;/\; 2 \rightsquigarrow 2$

$2 \;/\; 2 \rightsquigarrow 1$

```
let rec sub = function
| Lf, _ -> raise Subscript
| Br (v, t1, t2), 1 -> v
| Br (v, t1, t2), k when k mod 2 = 0 -> sub (t1, k / 2)
| Br (v, t1, t2), k -> sub (t2, k / 2)
```

The path to element i follows the **binary code** for i (the "subscript")

$O(\log n)$ if the tree is balanced:

**In[1]:**
```
let rec update = function
    | Lf, 1, w -> Br (w, Lf, Lf)
    | Lf, k, w -> raise Subscript  (* Gap in tree *)
    | Br (v, t1, t2), 1, w -> Br (w, t1, t2)
    | Br (v, t1, t2), k, w when k mod 2 = 0 ->
                                Br (v, update (t1, k / 2, w), t2)
    | Br (v, t1, t2), k, w -> Br (v, t1, update (t2, k / 2, w))
```

**Out[1]:**
```
val update : 'a tree * int * 'a -> 'a tree = <fun>
```

# Functional Trees

The path to element i follows the **binary code** for i (the "subscript")

$O(\log n)$ if the tree is balanced:

```
In[1]:  let rec update = function
          | Lf, 1, w -> Br (w, Lf, Lf)
          | Lf, k, w -> raise Subscript  (* Gap in tree *)
          | Br (v, t1, t2), 1, w -> Br (w, t1, t2)
          | Br (v, t1, t2), k, w when k mod 2 = 0 ->
                                  Br (v, update (t1, k / 2, w), t2)
          | Br (v, t1, t2), k, w -> Br (v, t1, update (t2, k / 2, w))

Out[1]: val update : 'a tree * int * 'a -> 'a tree = <fun>
```

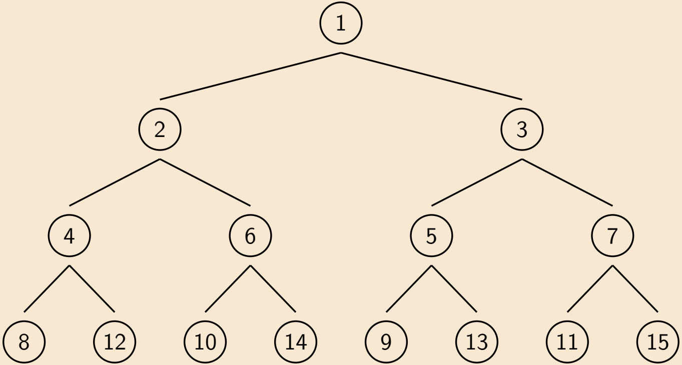
The path to element i follows the **binary code** for i (the "subscript")

$O(\log n)$ if the tree is balanced:

```
In[1]:  let rec update = function
          | Lf, 1, w -> Br (w, Lf, Lf)
          | Lf, k, w -> raise Subscript  (* Gap in tree *)
          | Br (v, t1, t2), 1, w -> Br (w, t1, t2)
          | Br (v, t1, t2), k, w when k mod 2 = 0 ->
                                  Br (v, update (t1, k / 2, w), t2)
          | Br (v, t1, t2), k, w -> Br (v, t1, update (t2, k / 2, w))

Out[1]: val update : 'a tree * int * 'a -> 'a tree = <fun>
```

The path to element i follows the **binary code** for i (the "subscript")




$$
\begin{aligned}
15 &= \text{0b1111} \quad \text{(right, right, right, here)} \\
12 &= \text{0b1100} \quad \text{(left, left, right, here)} \\
11 &= \text{0b1011} \quad \text{(right, right, left, here)}
\end{aligned}
$$

# Complexity of Dictionary Data Structures

**Linear search** Most general, needing only **equality** on keys, but inefficient (linear time)

**Binary search** Needs an **ordering** on keys.
$O(\log n)$ in the average case,
binary search trees are $O(n)$ in the worst case.

**Array subscripting** Least general, requiring keys to be **integers**, but even worst-case time is $O(\log n)$.