Priority Queue & Heaps

Priority Queue ADT (PQ)

- Stores a collection of pairs (item, priority)
- Priorities are from some ordered set

  (For simplicity, we use priorities from 0, 1, 2, ... with 0 "highest priority")

- Main operations:
  - insert(item, priority)

    adds item with ~~priorty~~ priority
  - extract_min()

    removes (& returns) item with least priority
  - update(item, priority)

    changes priority of item to priority

- We want a data structure to implement efficient PQs.

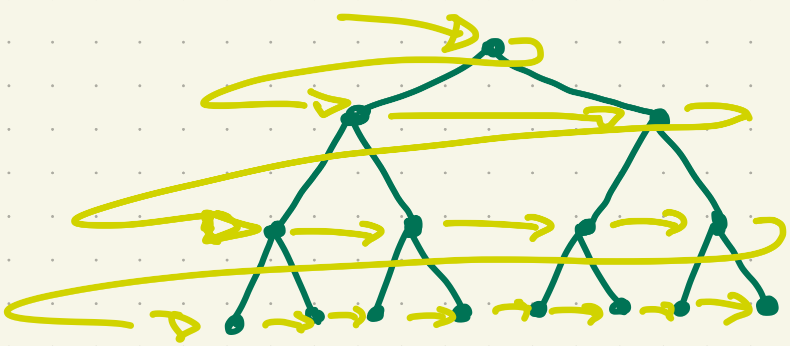
  e.g. $O(\log n)$ time for all operations.

- We (again) will use a particular kind of tree

Level-Order Traversal ...

... of ordered binary trees.

- visits each node of the tree once
- visits every node at depth i before any node at depth $i+1$ *
- visits every depth-d descendent of left(v) before any depth-d descendent of right(v).



* in some texts, it is bottom-up, not top-down.

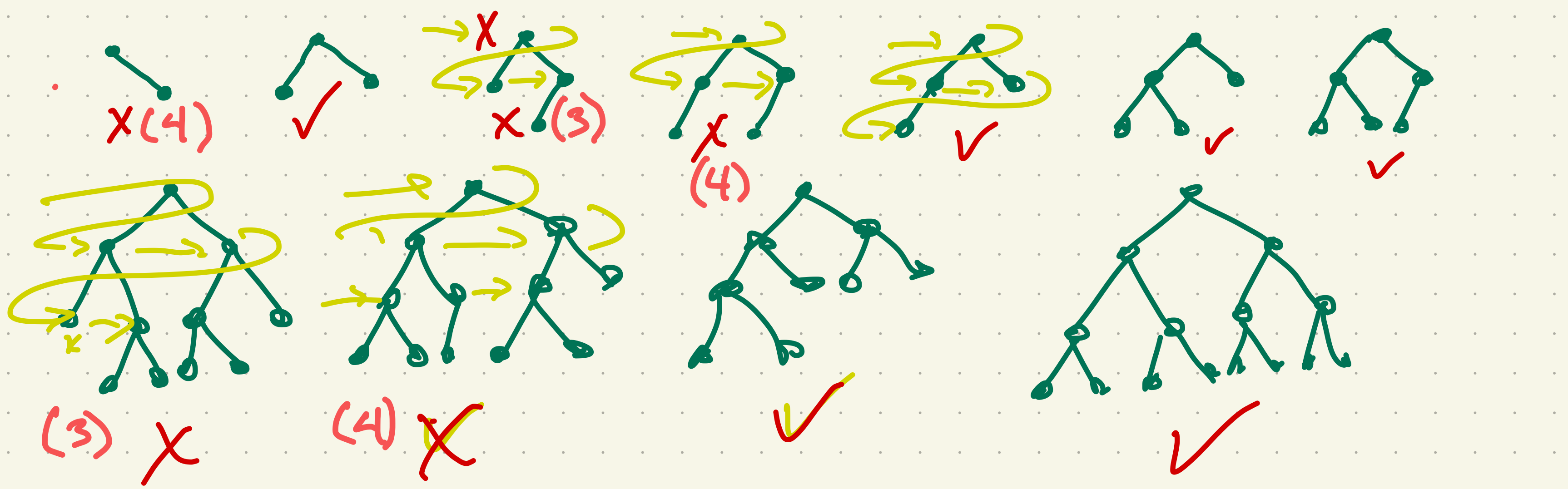
Complete Binary Tree

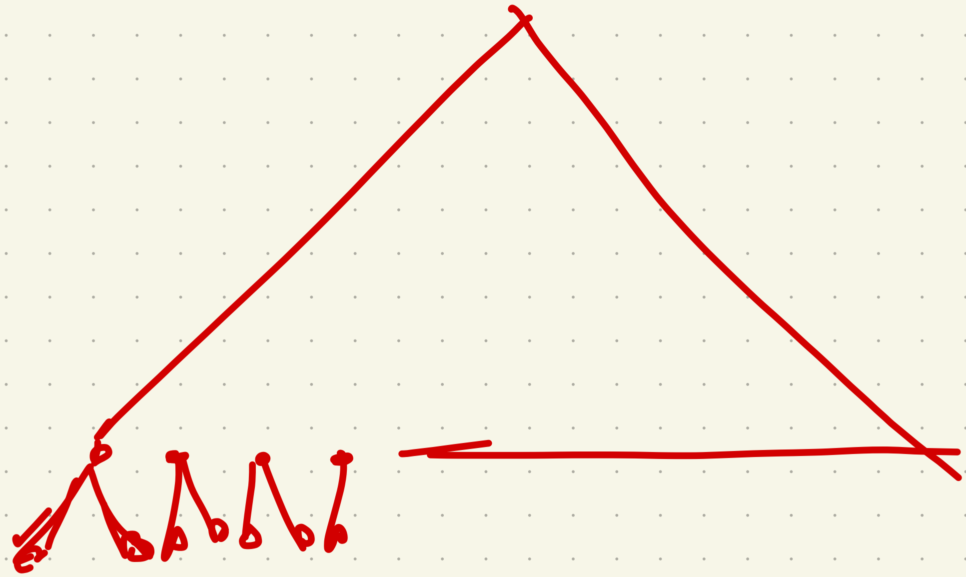
* has 2 children

** in the level-order traversal

A complete binary tree of height h is

1. a binary tree of height h;
2. with 2^d nodes at depth d, for every $0 \le d < h$
3. level order traversal visits every internal node before any leaf
4. every internal node is proper*, except perhaps the last,** which may have just a left child.





Binary Heap Data Structure

- a complete binary tree

"shape invariant"

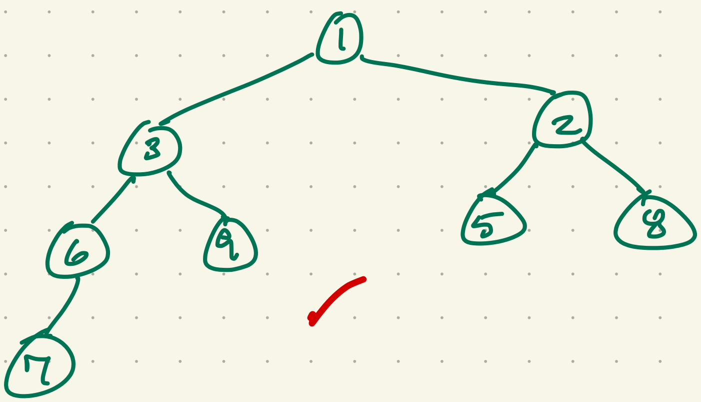
- with vertices labelled by keys from some ordered set,

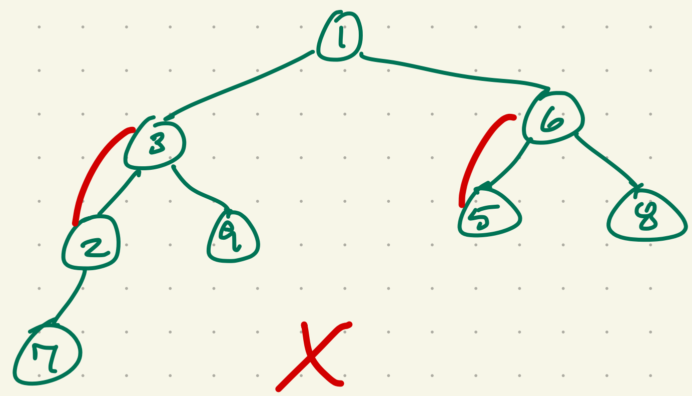
priorities

- s.t. $key(v) \geq key(parent(v))$ for every node v

"order invariant"

Eg.

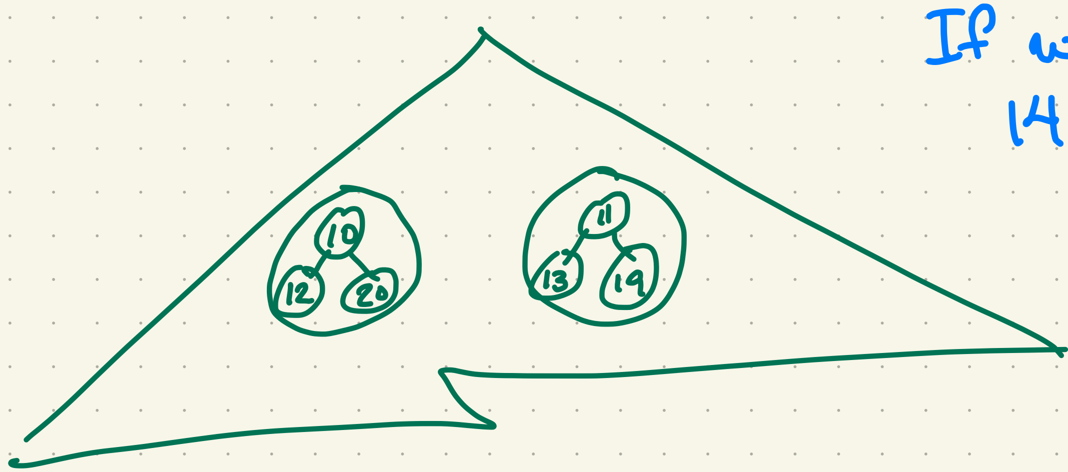




- This is the basic D.S. for implementing PQs (Binary min-heap).

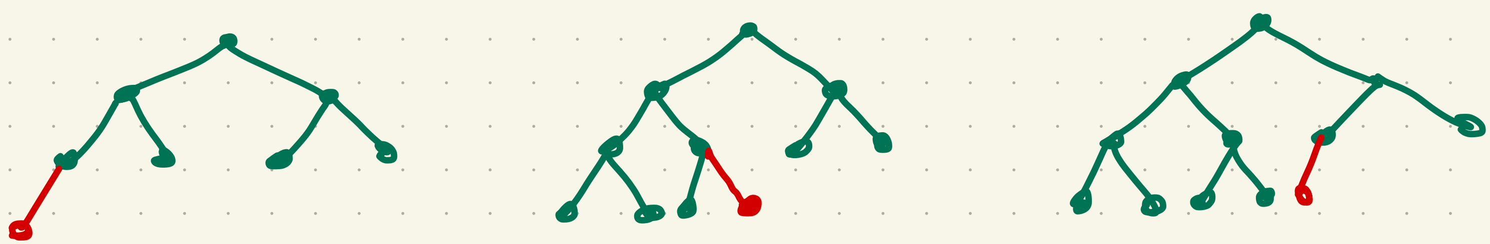
- How do we implement the operations so that the invariants are maintained?

- Consider Insertion:



If we want to insert 14 to the heap, where should it go?

Notice: there no choice about how the shape changes:



Heap Insert

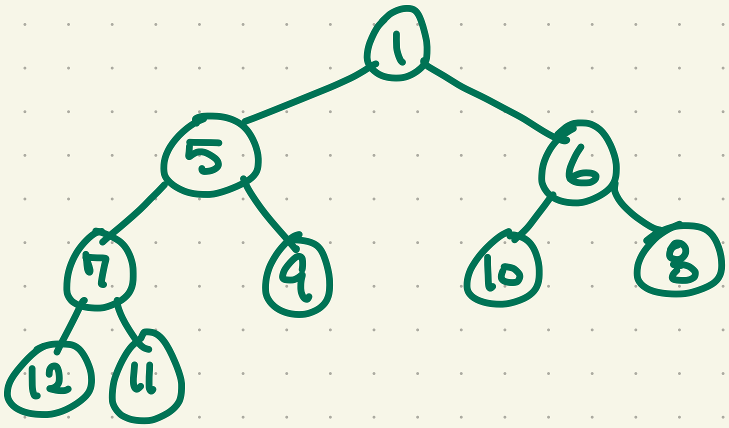
To insert an item with key k:

1. add a new leaf v with key(v) = k, so as to maintain the shape invariant

2. re-establish the order invariant by executing percolate_up(v).

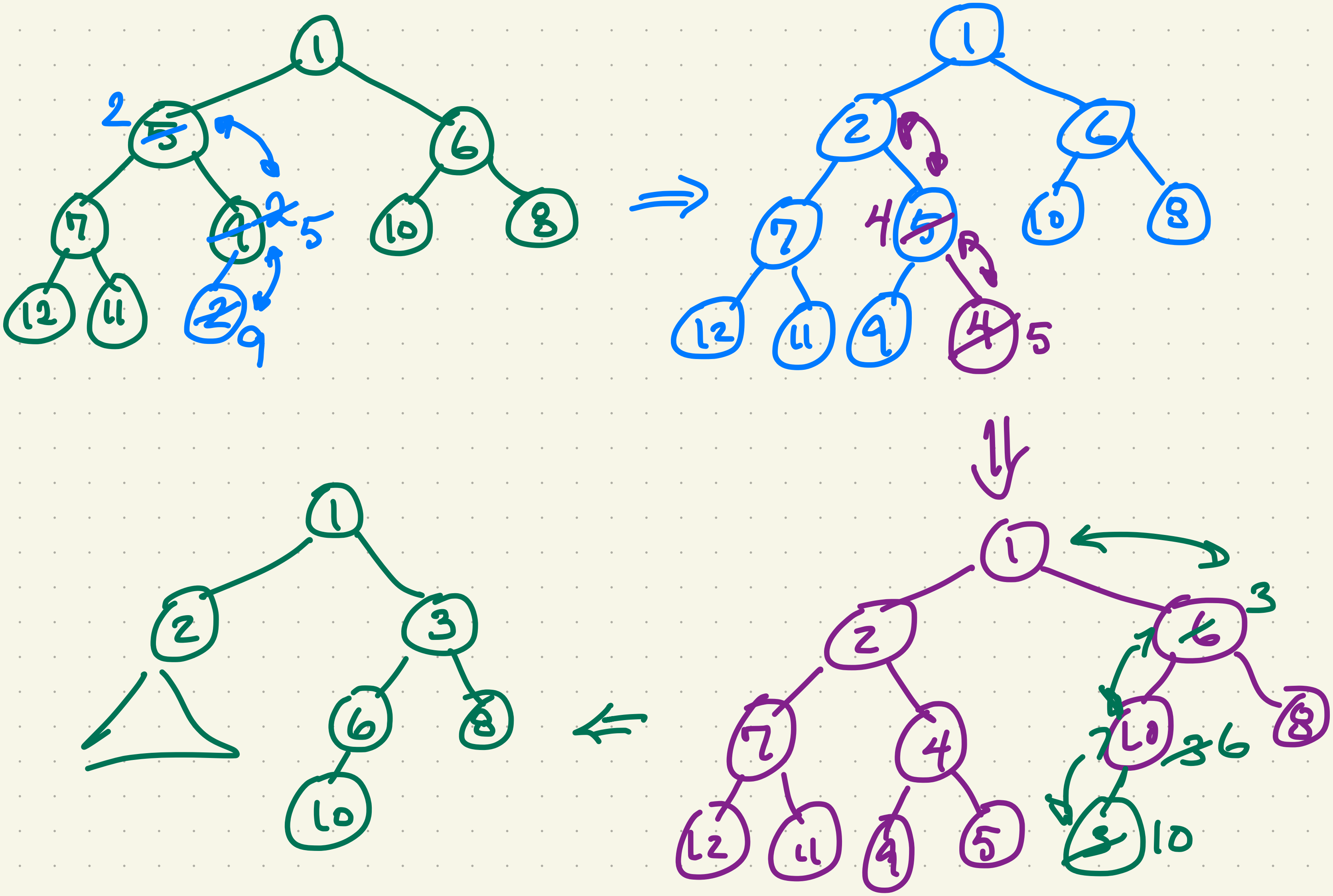
```
percolate_up(v){
    while(v is not root and key(v) < key(parent(v)) ){
        swap positions of v and parent(v) in the tree
    }
}
```

Insert 2, then 4, then 3 into:



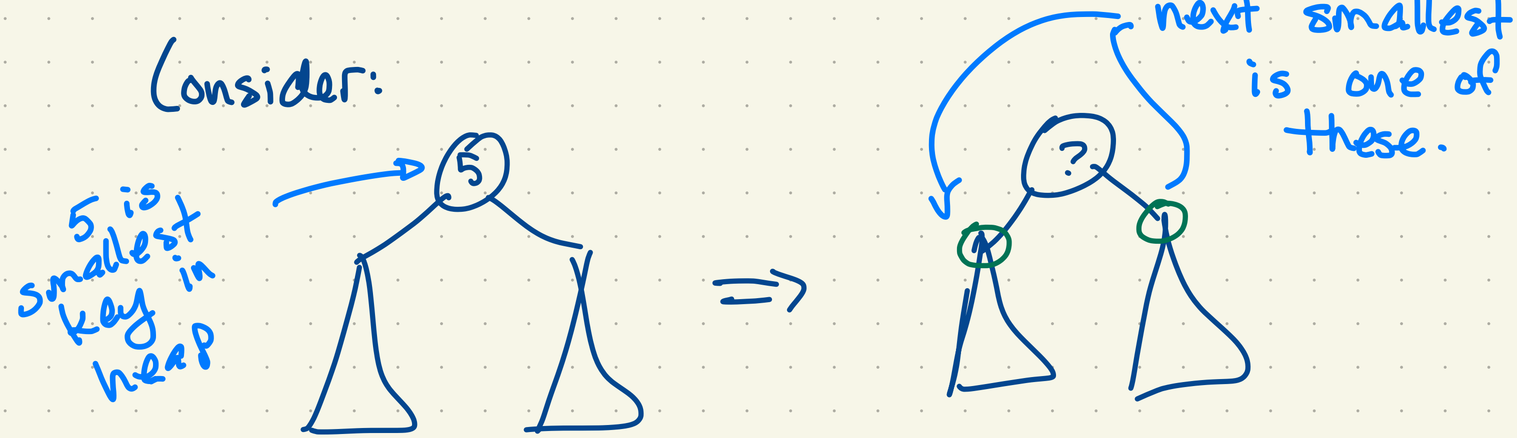
$\Rightarrow$

Insert 2, then 4, then 3 into:

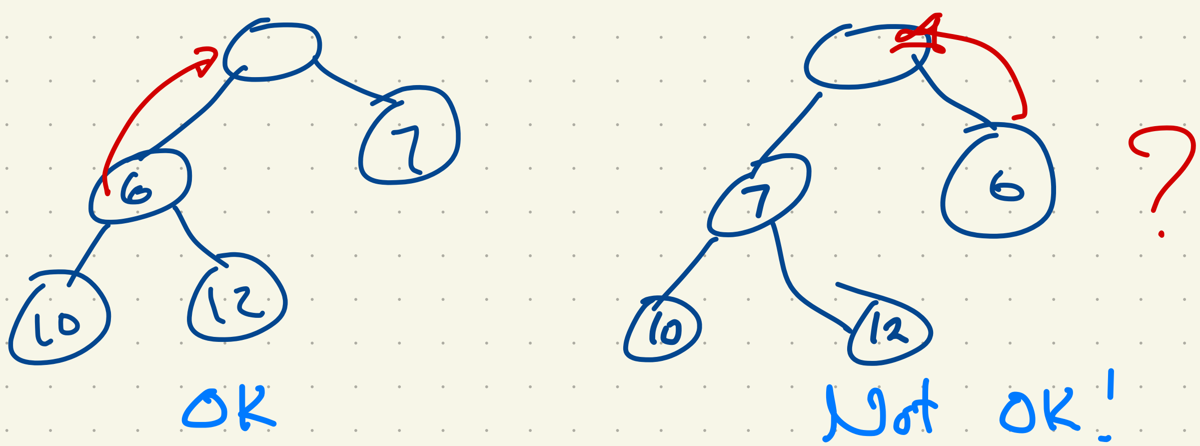


Heap Extract-Min.

Consider:



We must replace the root with the smaller of its children:



Heap Extract-Min.

To remove the (item with the) smallest key from the heap:

1. remove the root
2. replace the root with the "last leaf", so as to maintain the shape invariant.
4. restore the order invariant by calling

   percolate_down(root)

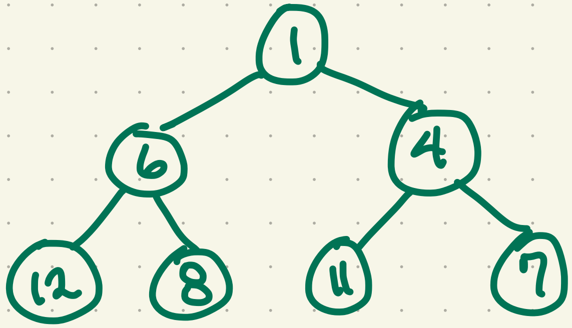
Percolate_down is more work than percolate_up, because it must look at <u>both</u> children to see what to do (and the children may or may not exist)

```
percolate_down(v){
    while (v has a child c with key(c) < key(v)){
        c ← child of v with the smallest key
            among the children of v.
        swap v and c in the tree
    }
}
```

Notice that:

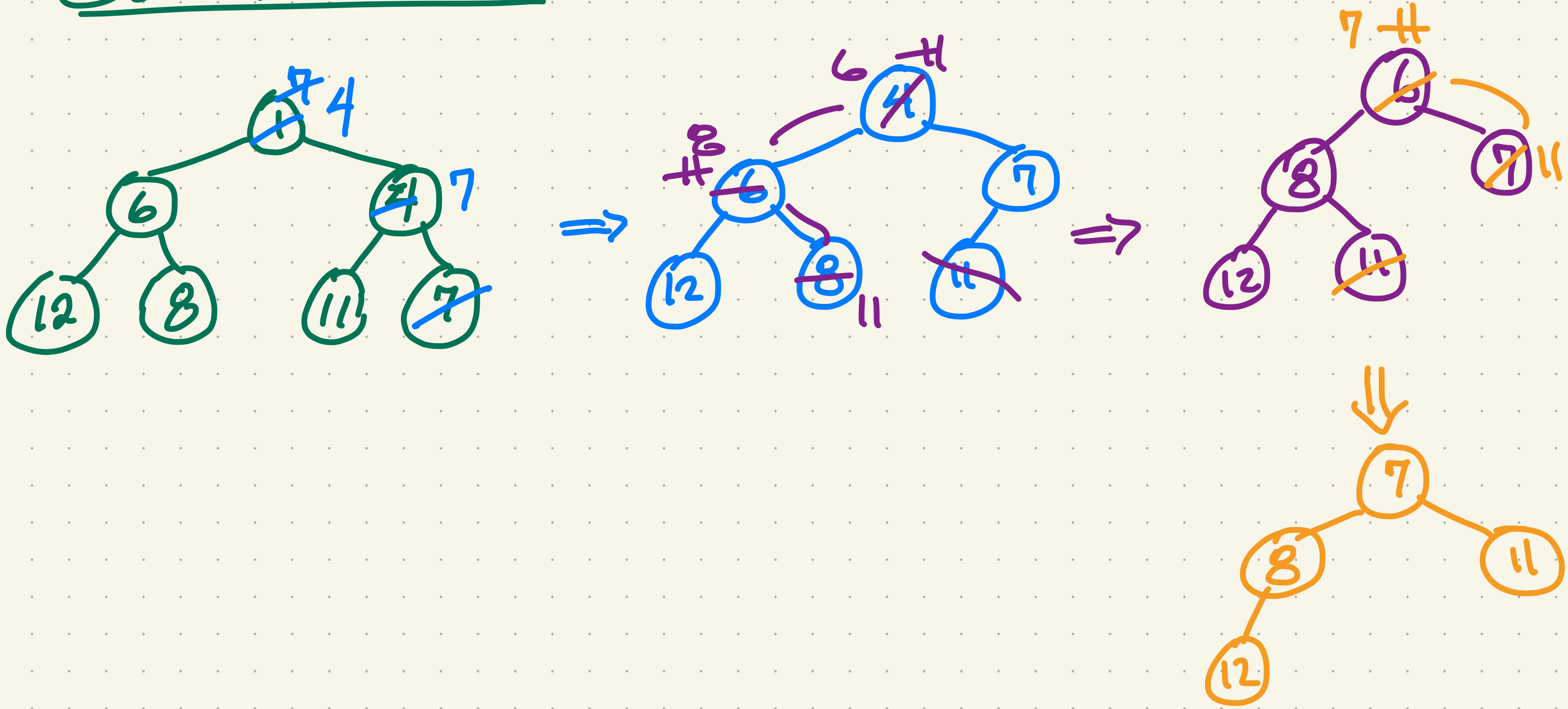
- v may have 0, 1, or 2 children
- if v has 2 children, we care about the one with the smallest key.

Do extract-min 3 times:



$\Rightarrow$

Do extract-min:



Complexity of Heap Insert & Extract-min

Claim: Insert & Extract-min take time $O(\log n)$ for heaps of size n.

Recall: A perfect binary tree of height h has $2^{h+1}-1$ nodes.

Pf: By induction on h (or "the structure of the tree").

Basis: If $h=0$ then we have $2^{0+1}-1 = 1$ nodes. ✓

I.H.: Consider some $h \geq 0$ and assume the perfect binary tree of height h has $2^{h+1}-1$ nodes.

I.S.: show the p.b.t. of height $h+1$ has $2^{(h+1)+1}-1$ nodes

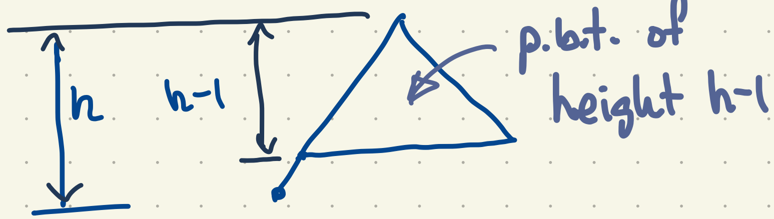
The tree is:

So it has $2^{h+1}-1+2^{h+1}-1+1 = 2\cdot 2^{h+1}-1 = 2^{(h+1)+1}-1$ nodes

Size bounds on complete binary trees

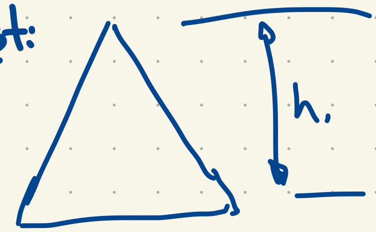
- Every complete binary tree with height h and n nodes satisfies: $$2^h \le n \le 2^{h+1} - 1$$

Smallest:



$$\#\text{nodes} = 2^{(h-1)+1} - 1 + 1 = 2^h$$

Largest:



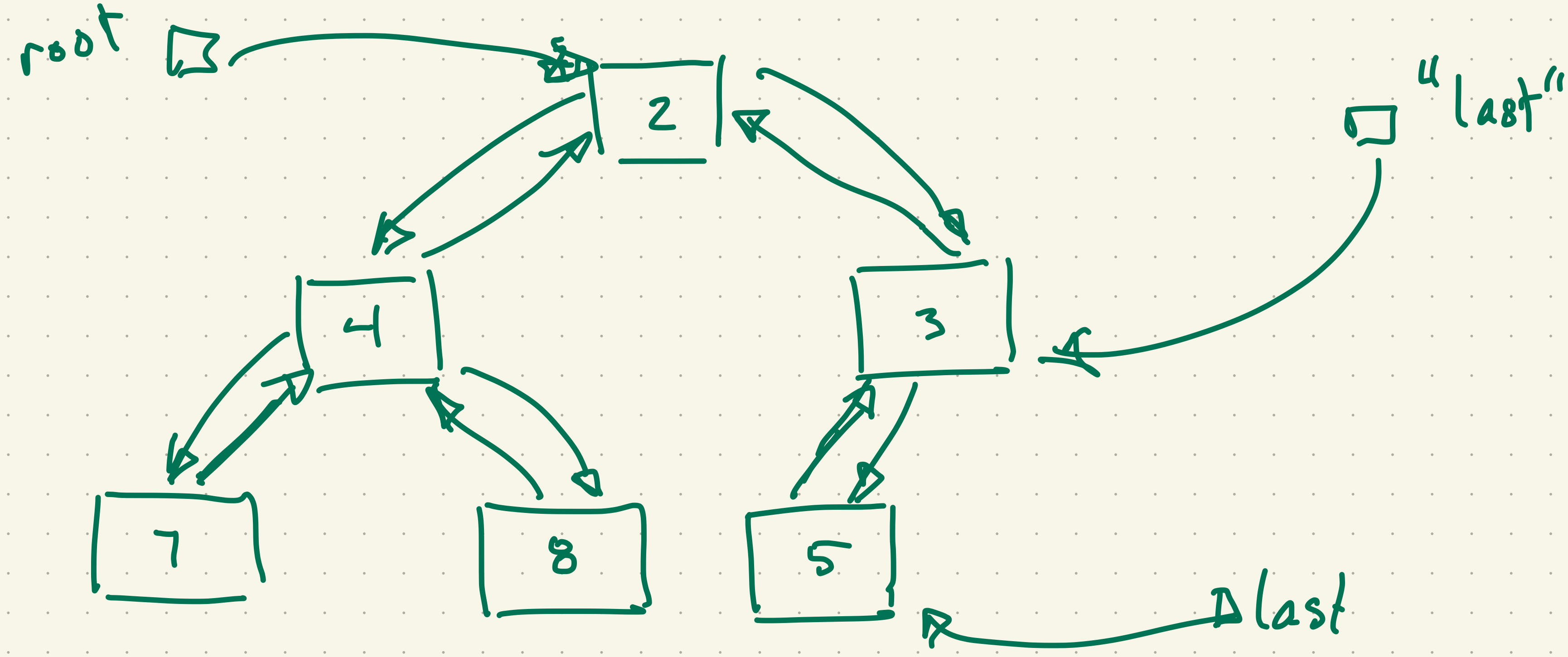
p.b.t. of height h.

So, we have:

$$2^h \le n$$
$$\log_2 2^h \le \log_2 n$$
$$h \le \log_2 n$$
$$h = O(\log n)$$

$\Rightarrow$ Heap insert & extract min take time $O(\log n)$

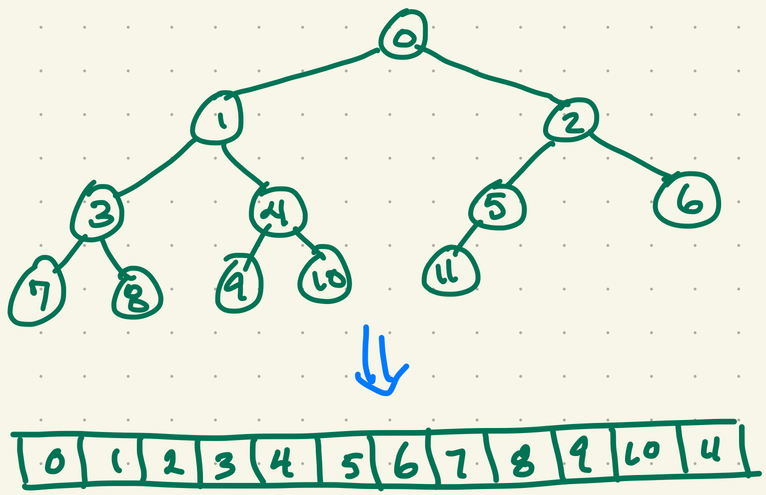
Linked Implementation of Heap



Node:

Array-Based Binary Heap Implementation

Uses this embedding of a complete binary tree of size n in a size-n array:

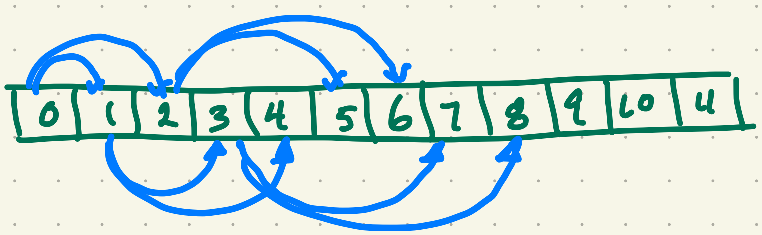


i^{th} node in level-order traversal

⇓

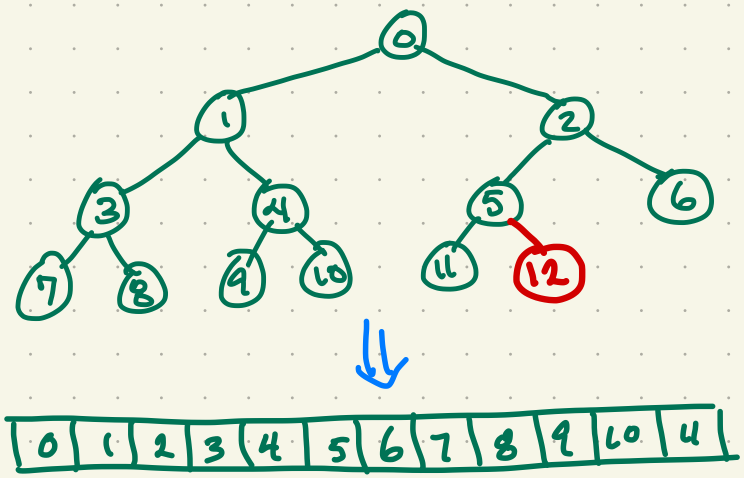
i^{th} array element

- Children of node i are nodes $2i+1$ & $2i+2$
- Parent of node i is node $\lfloor (i-1)/2 \rfloor$



Array-Based Binary Heap Implementation

Uses this embedding of a complete binary tree of site n in a size-n array:

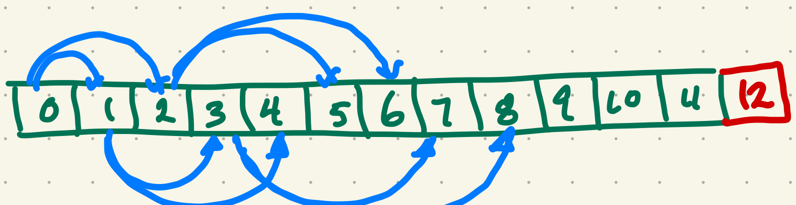


i^{th} node in level-order traversal

⇓

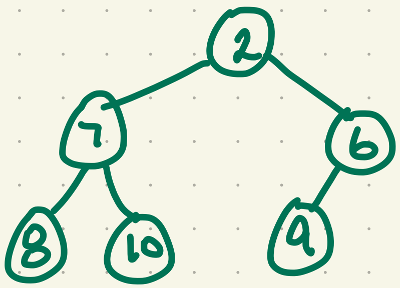
i^{th} array element

- Children of node i are nodes $2i+1$ & $2i+2$
- Parent of node i is node $\lfloor (i-1)/2 \rfloor$

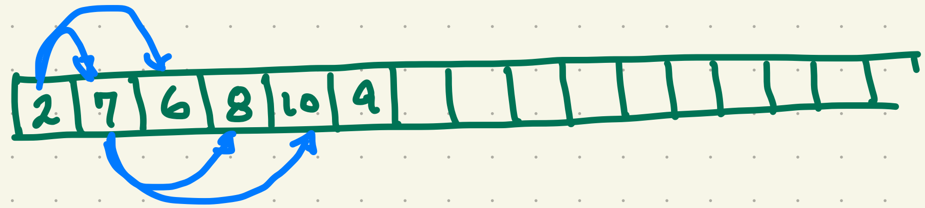


* Growing & Shrinking the tree is easy in the array embedding

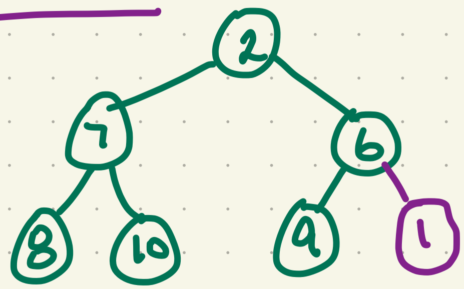
Partially-filled Array Implementation of Binary Heap: Insert

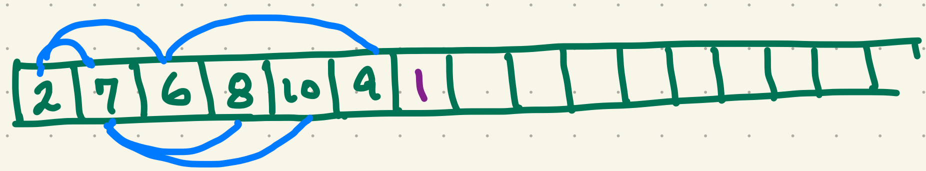


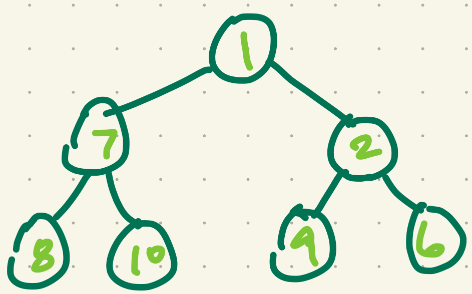
≈

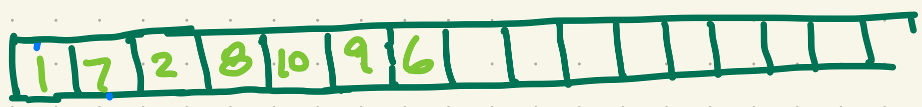


Insert 1

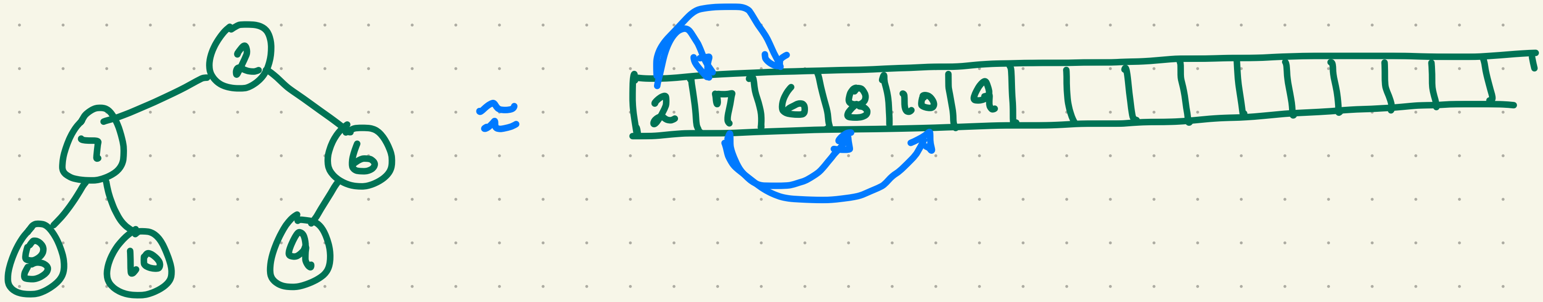




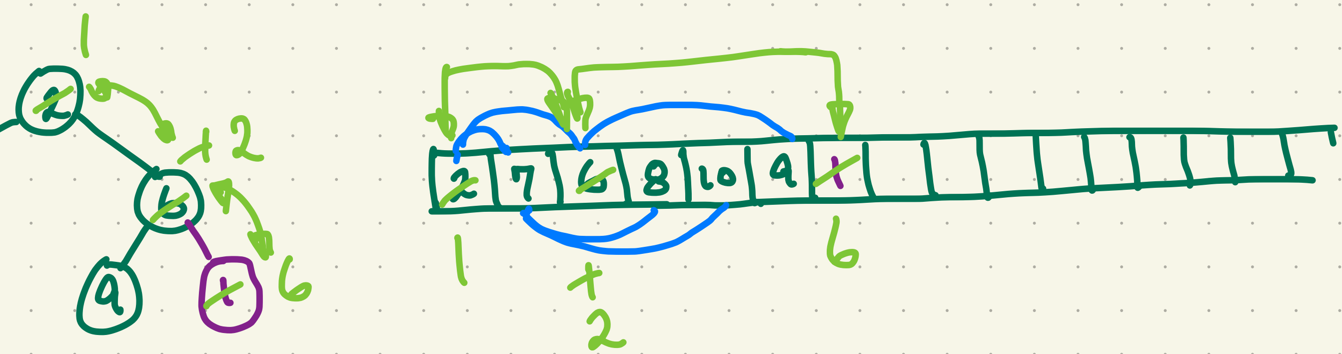


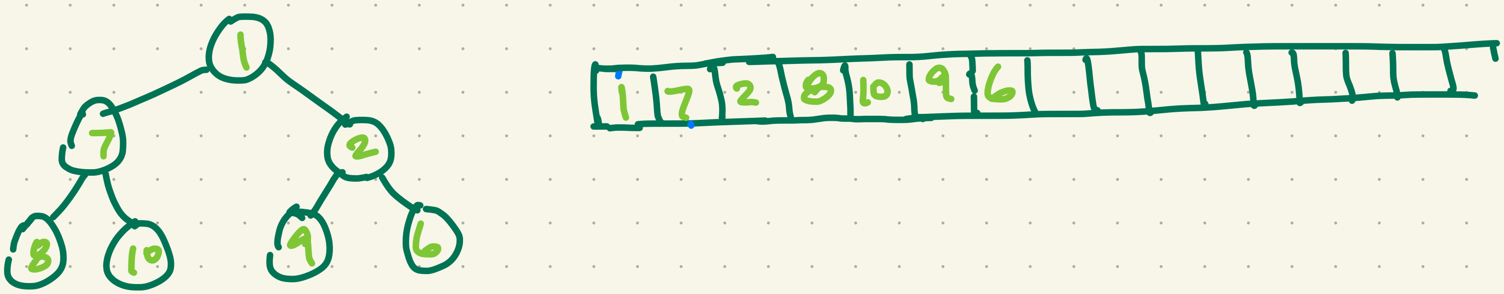


Partially-filled Array Implementation of Binary Heap: Insert



Insert 1





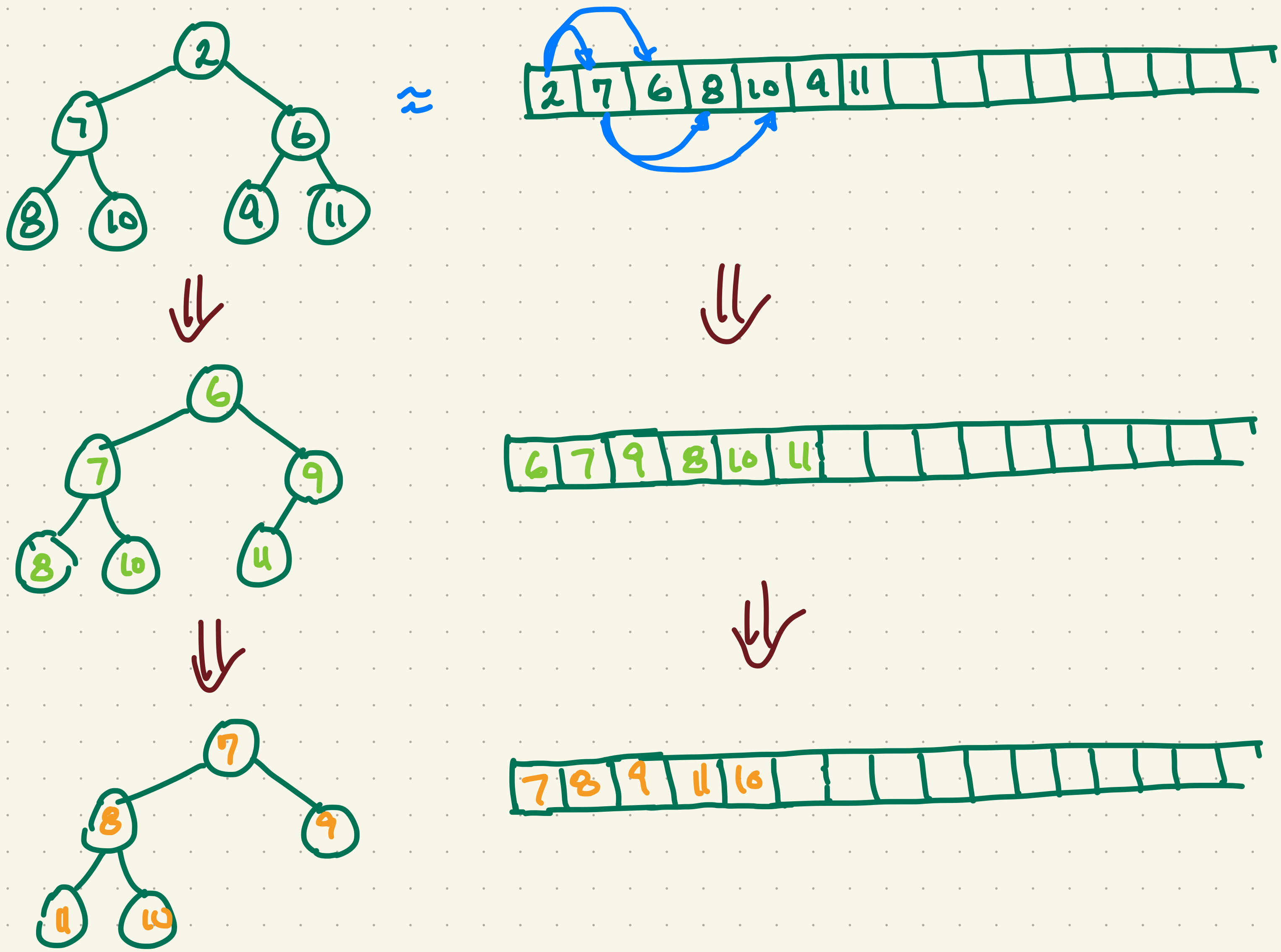
Insert for Array-based Heap

- Variables: array A, size
- Heap elements are in A[0] .. A[size - 1]

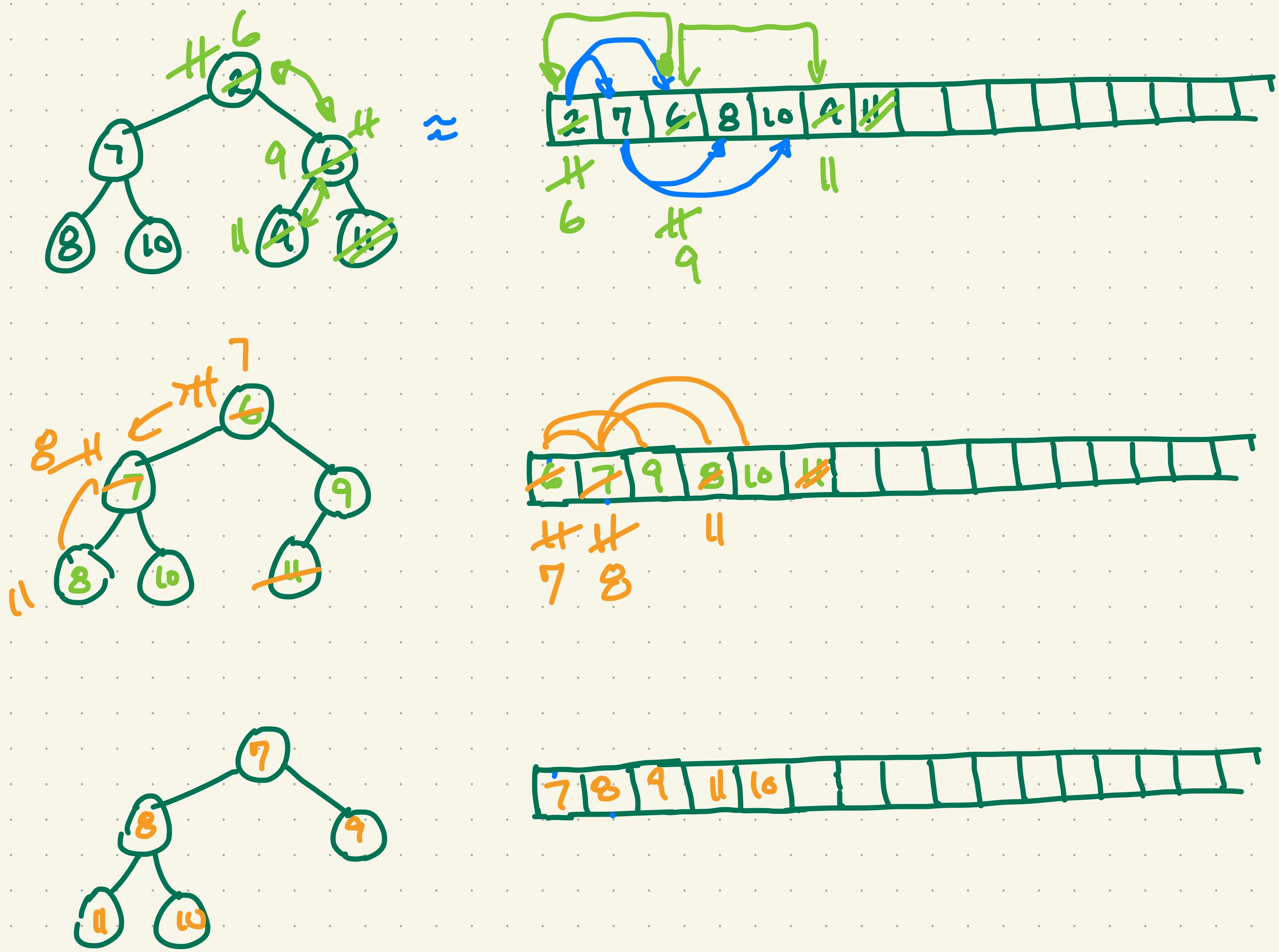
```
insert(k){
    A[size] ← k; // Add k as the new 'last leaf'
    v ← size
    p ← ⌊(v-1)/2⌋ // p ← parent(v)
    while ( v > 0 and A[v] < A[p]){
        swap A[v] and A[p]
        v ← p
        p ← ⌊(v-1)/2⌋
    }
    Size ← Size + 1;
}
```

percolate_up

Partially-filled Array Implementation of Binary Heap: Extract-min



Partially-filled Array Implementation of Binary Heap: Extract-min



Extract_min for Array-based Heap

```
extract_min(){
    temp ← A[0]  // record value to return
    size ← size - 1
    A[0] ← A[size] // move old last leaf to root
    i ← 0
    while( 2i+1 < size ){ // while i not a a leaf
        child ← 2i+1  // the left child of i
        if( 2i+2 < size  AND  A[2i+2] < A[2i+1]){
            child ← 2i+2 // use the right child if it exists
        }                // and has smaller key
        if (A[child] < A[i]){ //if order violated,
            swap A[child] and A[i]   // swap parent + child.
            i ← child
        }else {
            return temp
        }
    }
    return temp.
}
```

percolate down

A small space-for-time trade-off in Extract-min

- Extract-min does many comparisons, eg $(2i < size)$ to check if i is a leaf.
- Suppose we ensure the array has size $\geq 2 \cdot size$ and there is a big value, denoted ∞, that can be stored in the array but will never be a key. and every array entry that is not a key is ∞.
- Then, we can skip the explicit checks for being a leaf.

Extract-min variant

i has a child that is out of order

```
extract_min(){
    temp ← A[0]  // record value to return
    size ← size - 1
    A[0] ← A[size] // move old last leaf to root
    A[size] ← ∞ // **
    i ← 0
    while(A[2i+1] < A[i] or A[2i+2] < A[i]){
        if (A[2i+1] < A[2i+2]){
            swap A[2i+1] and A[i]
            i ← 2i+1
        }else{
            swap A[2i+2] and A[i]
            i ← 2i+2
        }
    }
    return temp
}
```

percolate down

it is the left child

it is the right child

Making a Heap from a Set

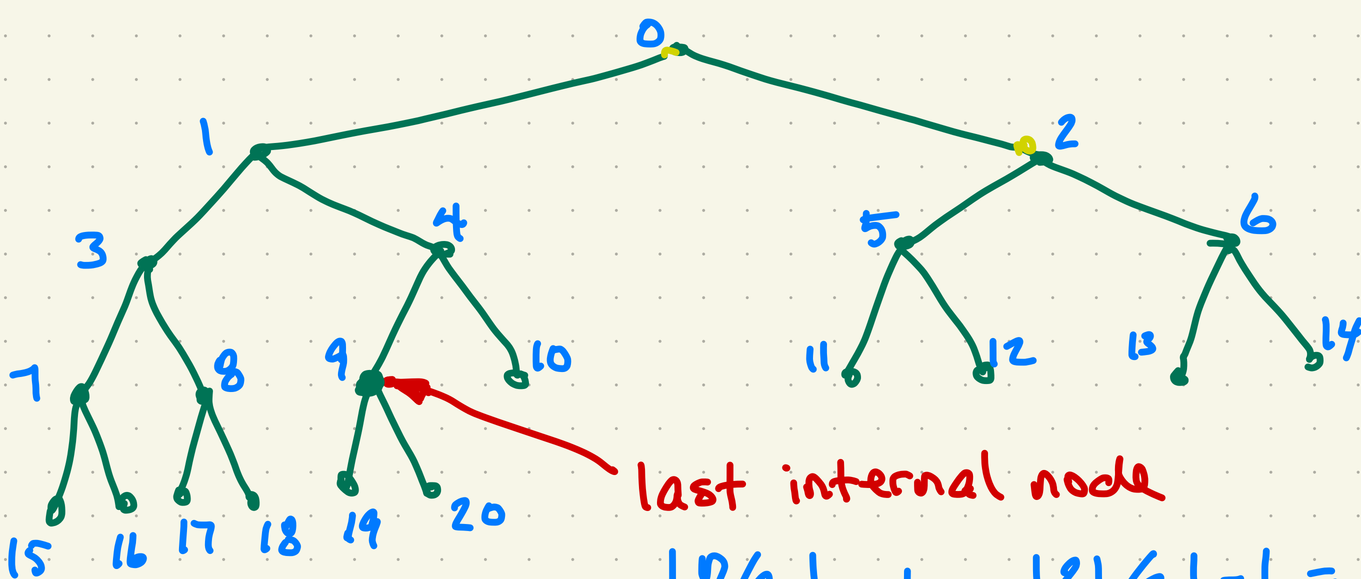
- Suppose you have n keys and want to make a heap with them.
- Clearly can be done in time $O(n \log n)$, with n inserts.
- Claim: the following alg. does it in time $O(n)$.

```
make_heap(T) {
    //T is a complete b.t. with n keys.
    for( i = ⌊n/2⌋-1 down to 0){
        call percolate-down on node i
    }
}
```

How does make-heap work?

- $\lfloor n/2 \rfloor - 1$ is the last internal node
- the algorithm does a percolate-down at each internal node, working bottom-up.

  (percolate-down makes a tree into a heap if the only node violating the order property is the root)



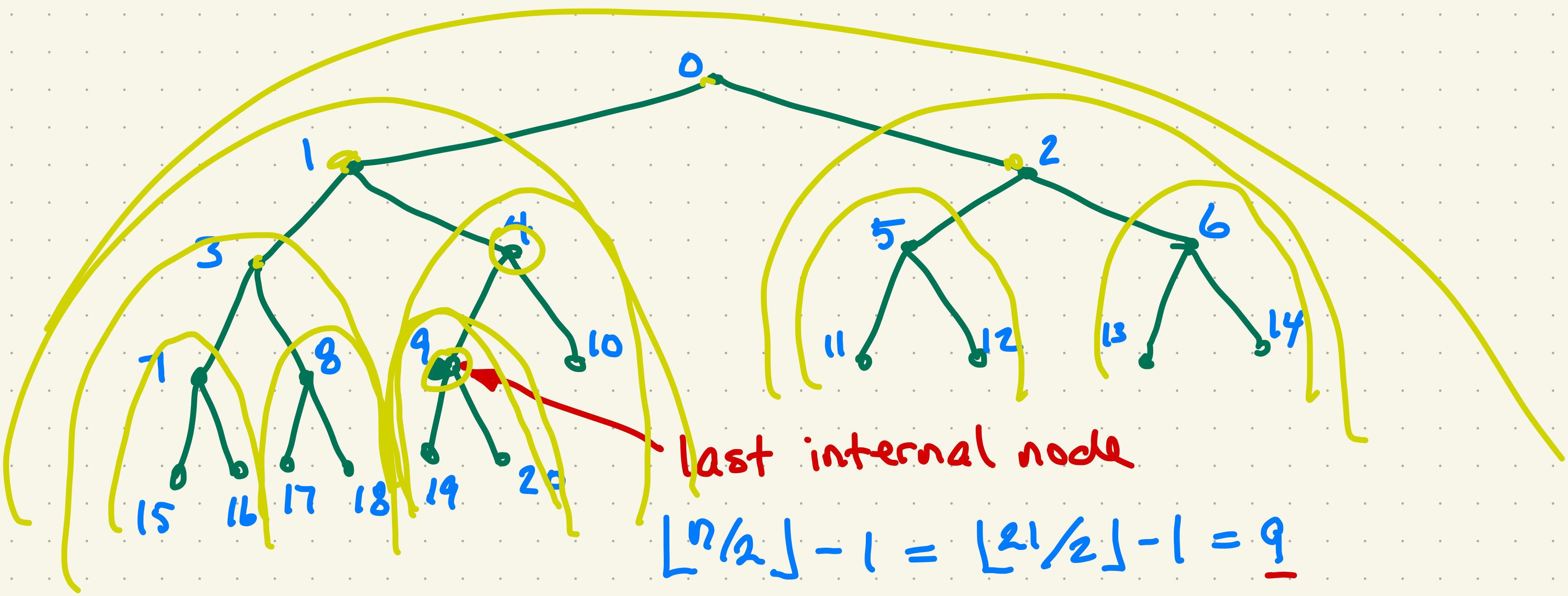
last internal node

$$\lfloor n/2 \rfloor - 1 = \lfloor 21/2 \rfloor - 1 = \underline{9}$$

How does make-heap work?

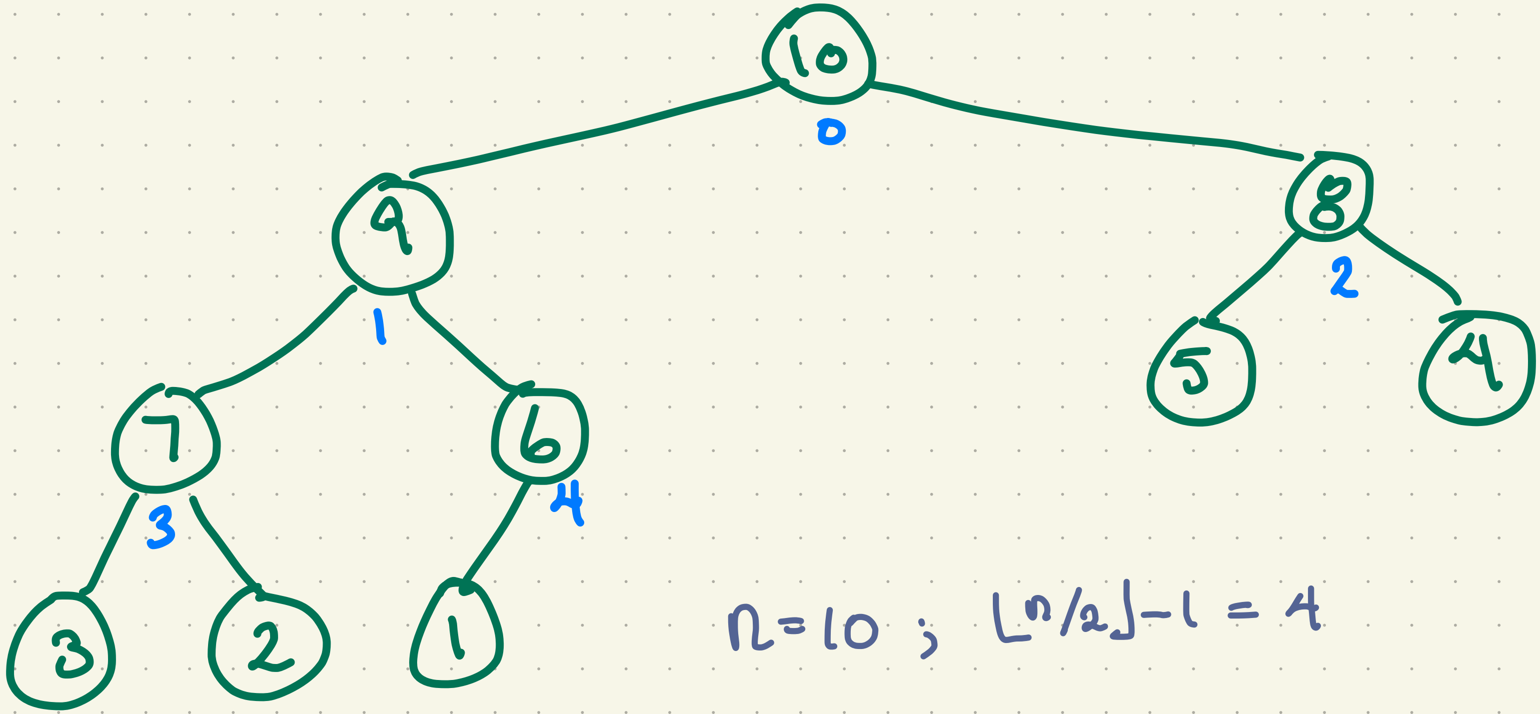
- $\lfloor n/2 \rfloor - 1$ is the last internal node
- the algorithm does a percolate-down at each internal node, working bottom-up.

  (percolate-down makes a tree into a heap if the only node violating the order property is the root)



$$\lfloor n/2 \rfloor - 1 = \lfloor 21/2 \rfloor - 1 = \underline{9}$$

Make heap Example

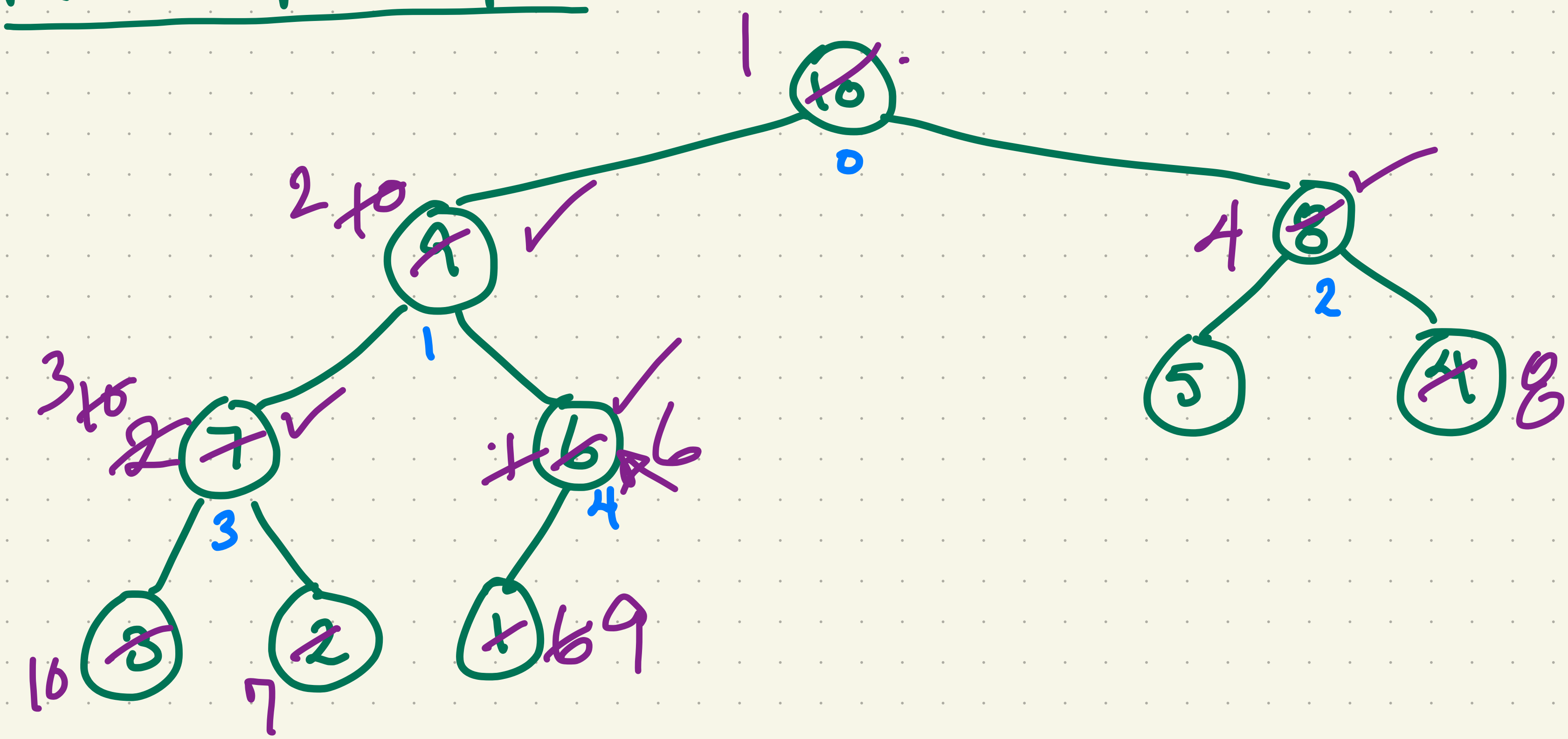


$n = 10$; $\lfloor n/2 \rfloor - 1 = 4$

<u>Notice</u>: The exact order of visiting nodes does not matter – as long as we visit children before parents.

[It follows that it is easy to do a recursive make-heap]

Make heap Example



$n = 10$; $\lfloor n/2 \rfloor - 1 = 4$

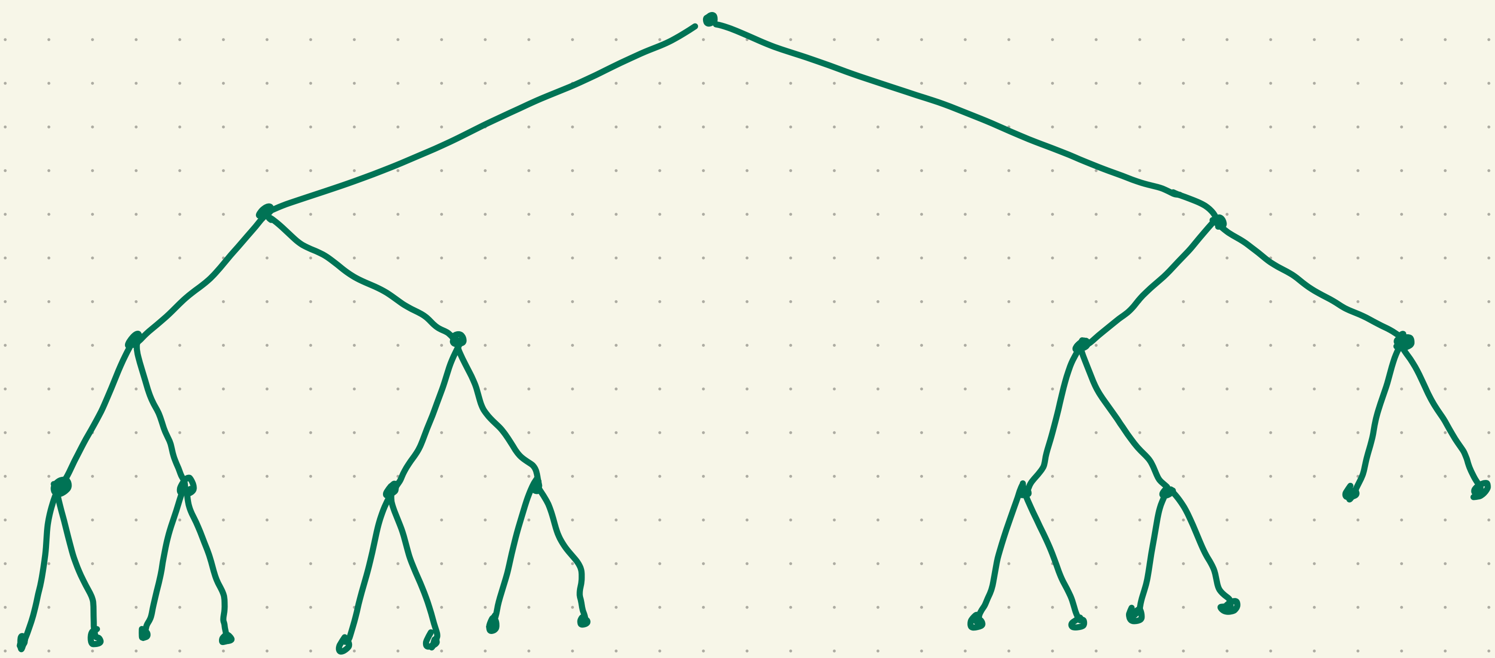
Notice: The exact order of visiting nodes does not matter – as long as we visit children before parents.

[It follows that it is easy to do a recursive make-heap]

Make-heap Complexity

- Clearly $O(n \log n)$: n percolate-down calls, each $O(\log n)$.
- How can we see it is actually $O(n)$?
- Intuition: mark a distinct edge for <u>every possible swap</u>.

  (Time taken is bounded by max. # of swaps possible.)



Time Complexity of Make-heap

Let $s(n)$ be the max number of swaps carried out by make-heap on a set of size n.

We can bound $s(n)$ by:

$$S(n) \leq \sum_{d=0}^{h-1} 2^d (h-d)$$

The max # of swaps for a call to percolate-down on a node at depth d is $h-d$

there are 2^d nodes at depth d

percolate-down is called, at most, on each node at each depth d from 0 to $h-1$

$$s(n) \leq \sum_{d=0}^{h-1} 2^d (h-d)$$

$$= 2^0(h-0) + 2^1(h-1) + \cdots + 2^{h-2}(\underbrace{h-(h-2)}_{=2}) + 2^{h-1}(\underbrace{h-(h-1)}_{=1})$$

Set $i = h-d$, so $d = h-i$ and while d ranges over $0, 1, \ldots, h-1$, i will range over $\underbrace{h-0}_{=h}, h-1, \ldots \underbrace{h-(h-1)}_{=1}$

Now

$$s(n) \leq \sum_{i=1}^{h} 2^{h-i}(i) = \sum_{i=1}^{h} \frac{2^h}{2^i} \cdot i \leq \sum_{i=1}^{h} \frac{n}{2^i} \cdot i \quad \left(\begin{array}{c}\text{because} \\ n \geq 2^h\end{array}\right)$$

$$= n \sum_{i=1}^{h} \frac{i}{2^i} \leq n \sum_{i=0}^{h} \frac{i}{2^i} \leq \underline{2n}$$

$$\left(\sum_{i=0}^{h} \frac{i}{2^i} = \frac{0}{2^0} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots = \frac{1}{2} + \frac{1}{2} + \underbrace{\frac{3}{8} + \frac{1}{4} + \frac{5}{32} + \cdots}_{\leq 1} \right)$$

Complexity of Make-heap

Work done by make-heap is bounded by a constant times the number of swaps so is $O(n)$.

Updating Priorities

- Suppose a heap contains an item with priority k, and we execute update_priority(item, j).
- We replace k with j in the heap, and then restore the order invariant:

  if $j < k$, do percolate_up from the modified node

  if $k < j$, do percolate_down from the modified node.

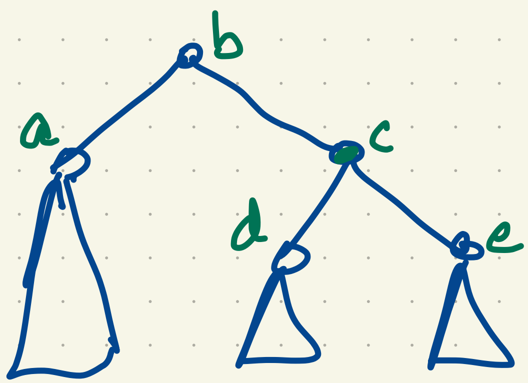
- This takes $O(\log n)$ time — but how do we find the right node to change??

Restoring order

- To do this we need an auxilliary data structure.

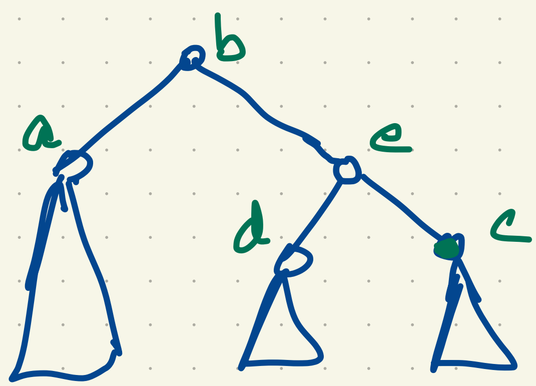
End

Correctness of swapping in percolate down



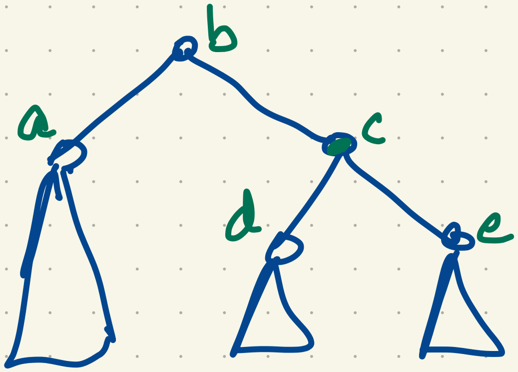
- Suppose we are percolating down c
- Then c and b were previously swapped, so we know $b \leq e$, $b \leq d$, and $b < c$.
- If $c > e$ and $e \leq d$, we swap c, e

Now:



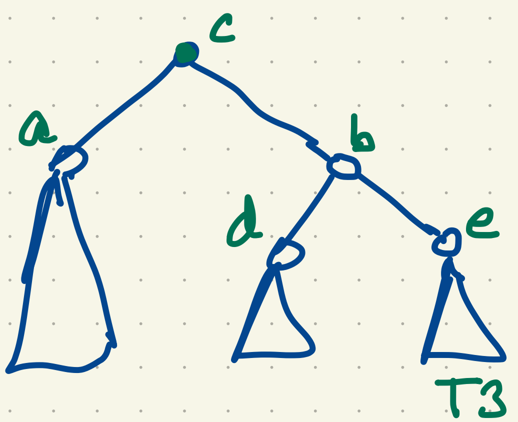
- we know $b \leq e \leq c$ and $b \leq e \leq d$
- so order is OK, except possibly below c — which we still have to look at.

Correctness of swapping in percolate-up



- suppose we are percolating up c
- we know $c \leq d$, $c \leq e$, because we previously swapped c with d or e
- we know that $b \leq a$
- if $c < b$, we swap c, b

Now:



- we now know that

$$c < b \leq e$$

and $c < b \leq d$

and $c < b \leq a$

- So order is OK, except possibly with ancestors of c, which we still must check.