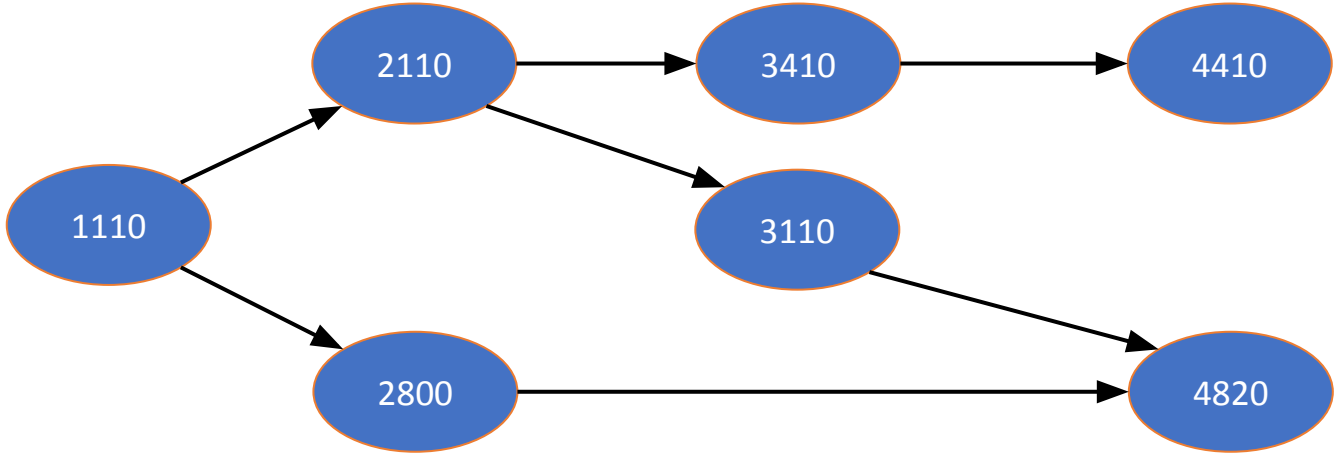
Topological sorting

CS core course prerequisites

(simplified)



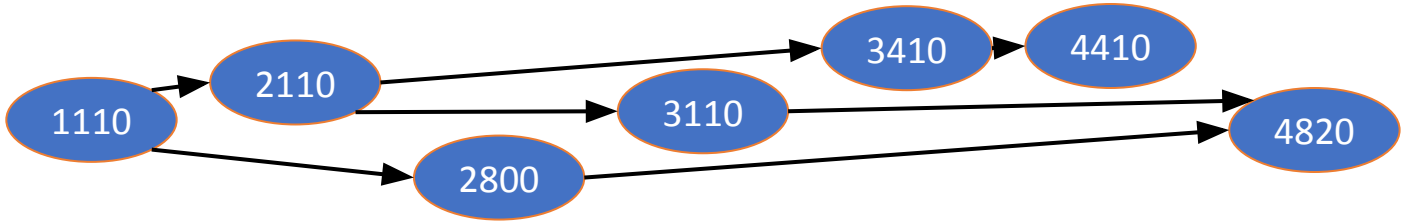
Problem: find an order in which you can take courses without violating prerequisites

e.g. 1110, 2110, 2800, 3110, 3410, 4410, 4820

Topological order

A **topological order** of directed graph G is an ordering of its nodes as v_1, v_2, …, v_n, such that for every edge (v_i, v_j), it holds that i < j.

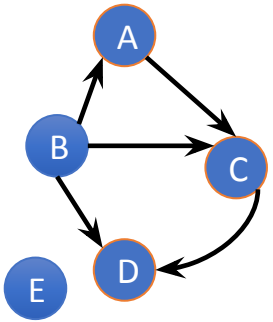
Intuition: line up the nodes with all edges pointing left to right.



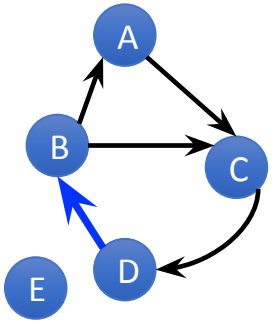
Other applications: robot planning, job scheduling, compilers

Cycles

- A directed graph can be topologically ordered if and only if it has no cycles
- A **cycle** is a path $v_0, v_1, \ldots, v_p$ such that $v_0 = v_p$
- A graph is **acyclic** if it has no cycles
- A directed acyclic graph is a **DAG**

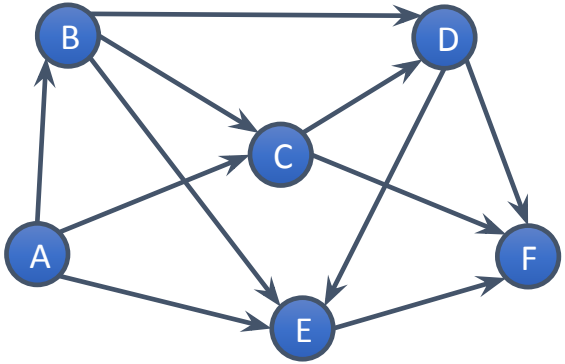


DAG



Not a DAG

Is this graph a DAG?



Yes!
It was a DAG.

- If a node is part of a cycle, it must have an incoming edge
- Deleting a node with indegree zero would not remove any cycles
- Keep deleting such nodes and see whether graph “disappears”

The order in which we removed nodes was a topological order!

Topological sort: algorithm 1

While (there is a vertex v with no incoming edges):

    Append v to result

    Remove all of v's outgoing edges from graph

If vertices with incoming edges remain, a cycle exists

- Don't want to actually mutate graph, so instead:
  - Count in-degree of all vertices (store in dictionary)
  - Add all vertices with in-degree 0 to list
  - While list is not empty:
    - Remove a vertex and add to result
    - Subtract in-degree of all neighbors by 1

CS 2110 Lecture 14b

Heaps

- Priority queues
- Heapsort

Priority queues

Common pattern: give me the “next” thing

Different choices for “next”:

- Queue (FIFO): who has been *waiting the longest*?
- Stack (LIFO): who was *added most recently*?
- **Priority queue: who is *most important*?**

Applications:

- Shortest paths
- Task deadlines (what is due soonest, regardless of when it was assigned)

Desired functionality

- `peek()`: Return the most important element
- `poll()`: Remove and return the most important element (aka “pop”)
- `add()`: Add a new element

Want these operations to be *fast* (low time complexity)

- Ideally, `peek()` should be $O(1)$ (always know what the best value is)
- `poll()` and `add()` must preserve whatever invariant makes `peek()` fast without being slow themselves

Candidate implementations (N elements)

- Unsorted list
  - `peek`: $O(N)$, `poll`: $O(N)$, `add`: $O(1)$
- Sorted list
  - `peek`: $O(1)$, `poll`: $O(1)$, `add`: $O(N)$
- Balanced binary search tree (BST)
  - `peek`: $O(\log N)$, `poll`: $O(\log N)$, `add`: $O(\log N)$
  - Balancing is complicated

Do we need/want to keep elements sorted?

Often, processing one element (poll) will cause many new elements to be added to the queue (add).

- E.g. exploring a cave: take the right fork, but at the end of that tunnel, three new tunnels open up

Keeping all these TODOs sorted is wasteful – we'll keep having to move things around when new tasks come in, and all we care about is which *one* is next

Strategy: relax invariant

Binary heaps (data structure)

Do not confuse with “heap memory” – different use of the word “heap”:

- “The stack”: Local variables live in activation records on the call stack
- “The heap”: Objects are allocated in memory apart from the call stack

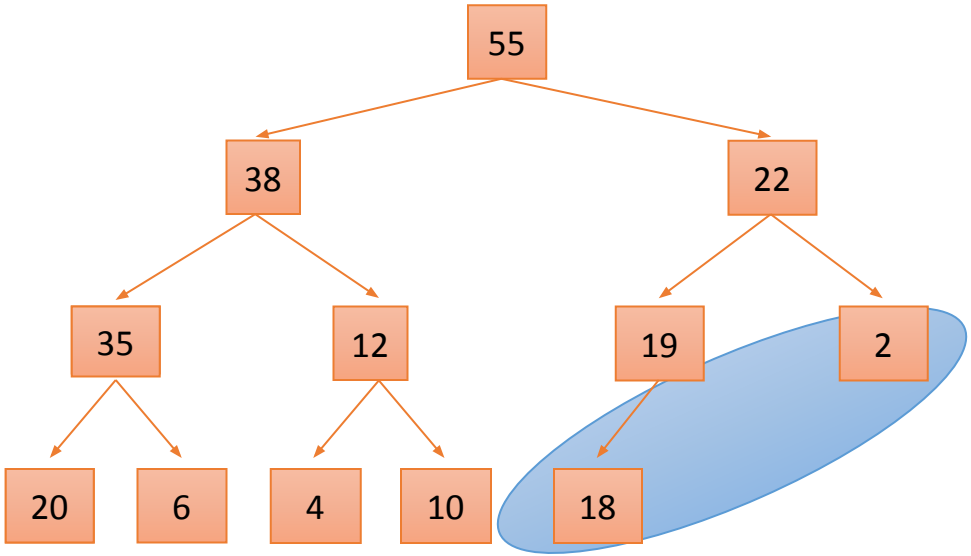
A Heap...

Is a binary tree satisfying 2 properties:

1. **Heap-ordered** (order invariant). Every node is “more important” than its children
   - Min-heap: every node is <= its children (smallest on top)
     “earliest deadline,” “shortest distance”
   - Max-heap: every node is >= its children (biggest on top)
     “largest reward”

Heap-order (max-heap)

Every element is <= its parent



Note: Bigger elements
can be deeper in the tree!

A Heap...

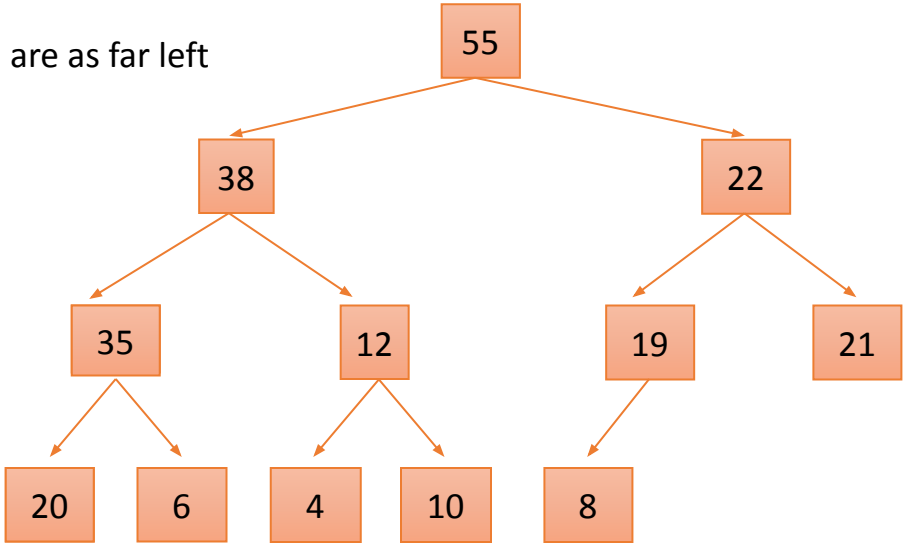
Is a binary tree satisfying 2 properties:

1. **Heap-ordered** (order invariant). Every node is “more important” than its children
2. **Completeness** (shape invariant). Every level of the tree (except last) is completely filled, and on last level nodes are as far left as possible.

Completeness

Every level (except last) completely filled.

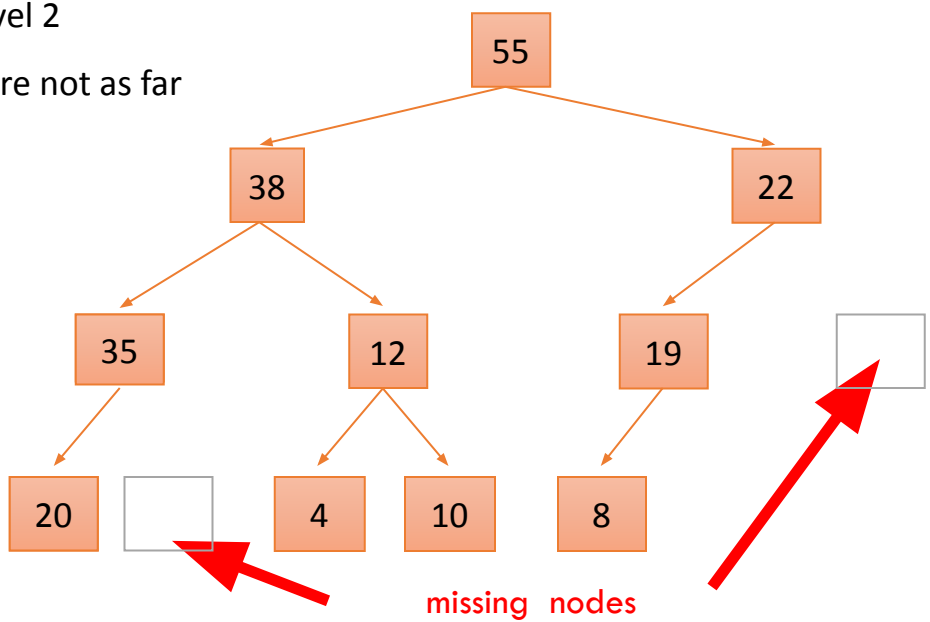
Nodes on bottom level are as far left as possible.



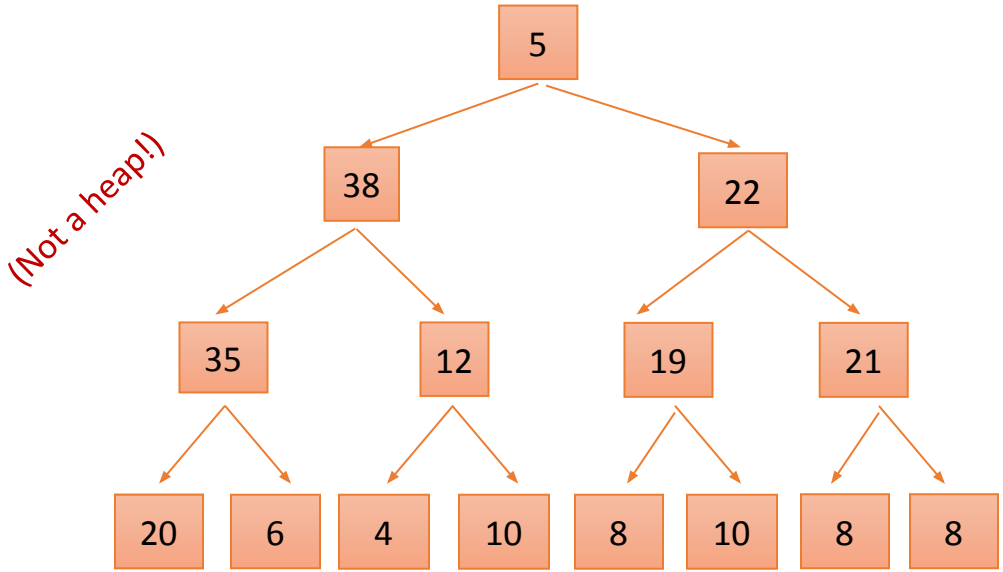
Completeness

Not a heap because:

- missing a node on level 2
- bottom level nodes are not as far left as possible



Perfect binary tree



Levels	total number of nodes
1	2^1 – 1 = 1
2	2^2 – 1 = 3
3	2^3 – 1 = 7
4	2^4 – 1 = 15

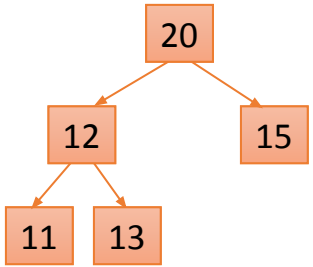
Perfect binary tree with 2^k-1 nodes has k levels

Add one more node: 2^k nodes has k+1 levels

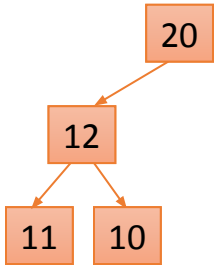
n nodes has $\lceil \lg(n+1) \rceil \subset$ O(log n) levels

Question

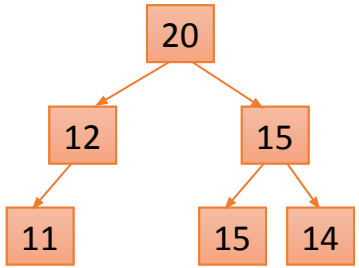
Which of the following are valid max-heaps?



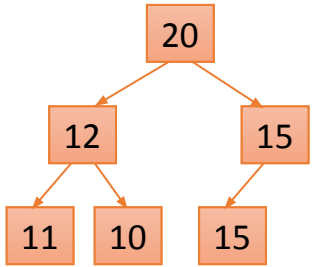
(a)



(b)



(c)

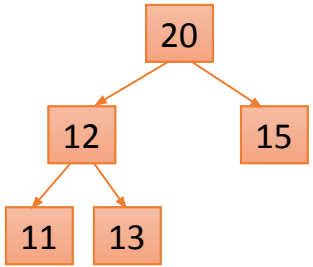


(d)

(e) none of them

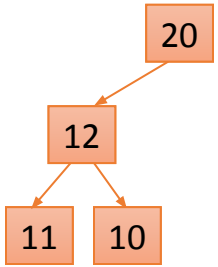
Question

Which of the following are valid heaps?



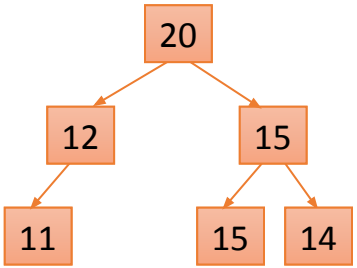
(a)

No



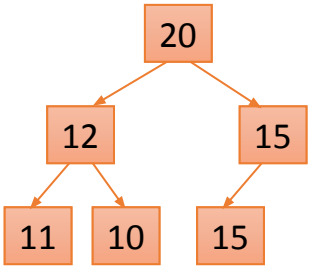
(b)

No



(c)

No



(d)

Yes

Back to priority queues

Heaps can implement priority queues

- Efficiency we will achieve (storing *N* elements):
  - add(): O(log *N*)
  - poll(): O(log *N*)
  - peek(): O(1)
- No linear time operations: better than lists
- peek() is constant time: better than balanced trees

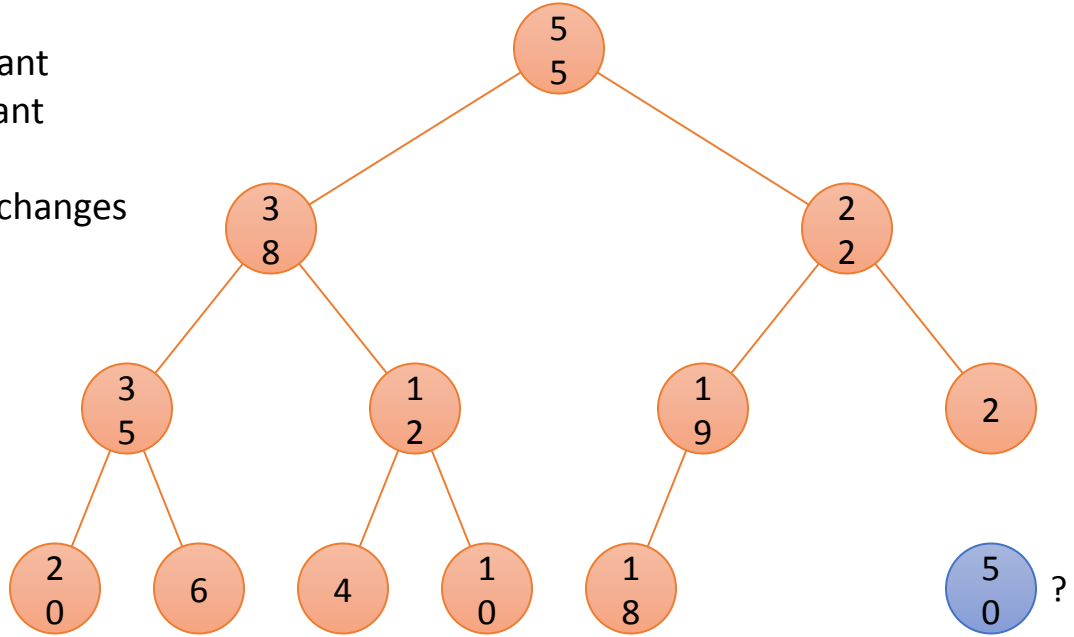
Heap algorithms

Exercise: Adding an element

Must preserve:

1. Shape invariant
2. Order invariant

Goal: minimize changes



Checkpoint: add

Where does 50 go?

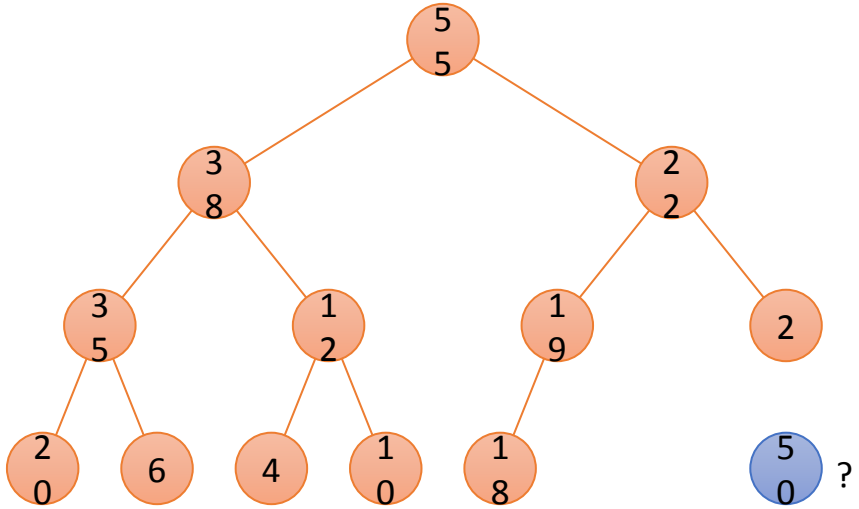
A. 2

B. 19

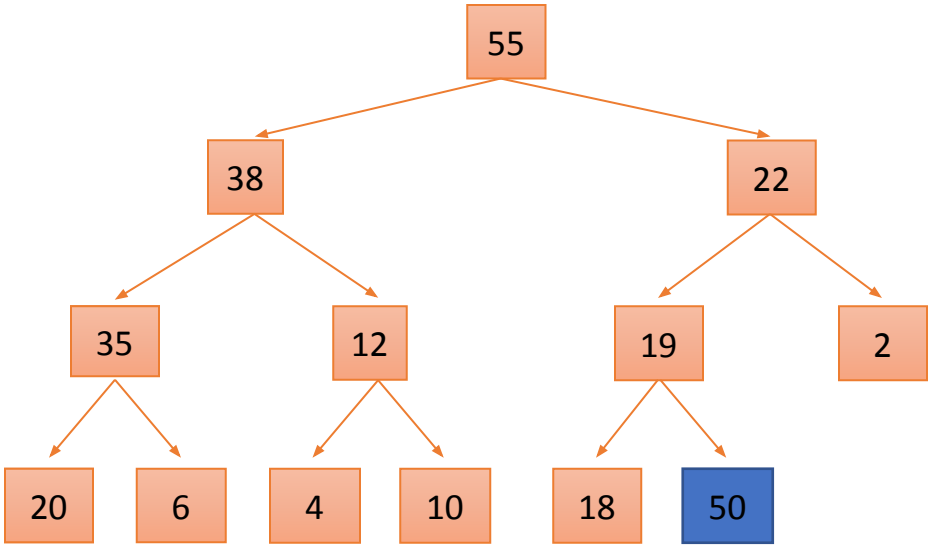
C. 22

D. 38

E. Nothing

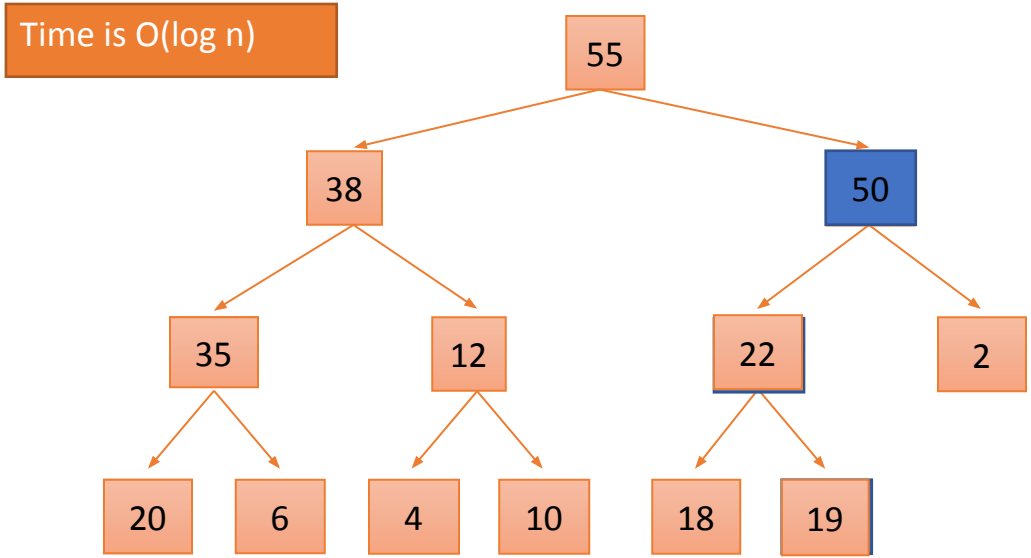


Heap: add(e)



1. Put in the new element in a new node (leftmost empty leaf)

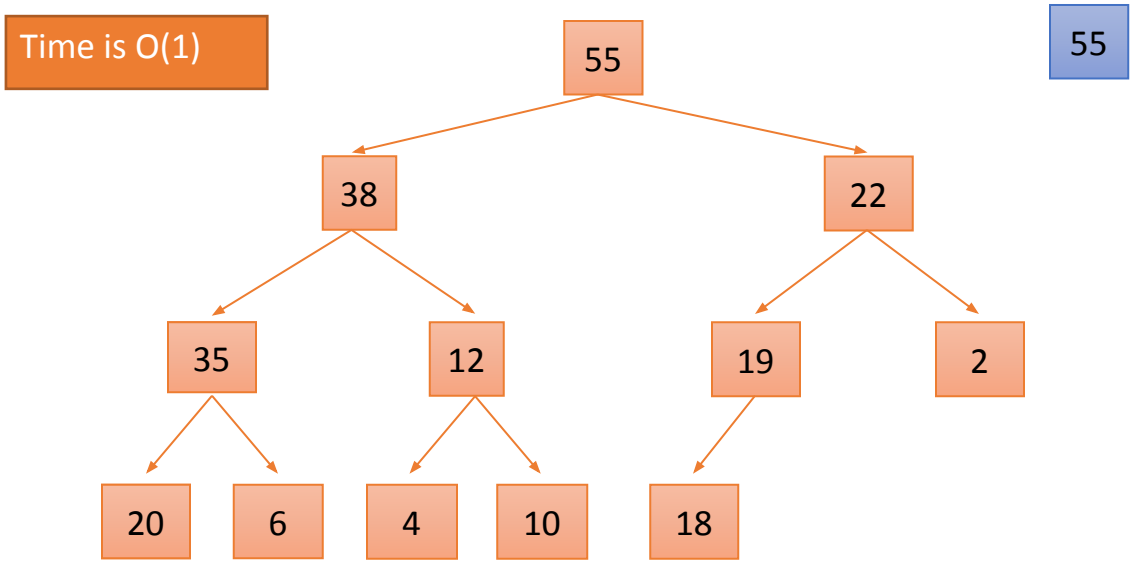
Heap: add(e)



1. Put in the new element in a new node (leftmost empty leaf)
2. Bubble new element up while greater than parent

Heap: peek( )

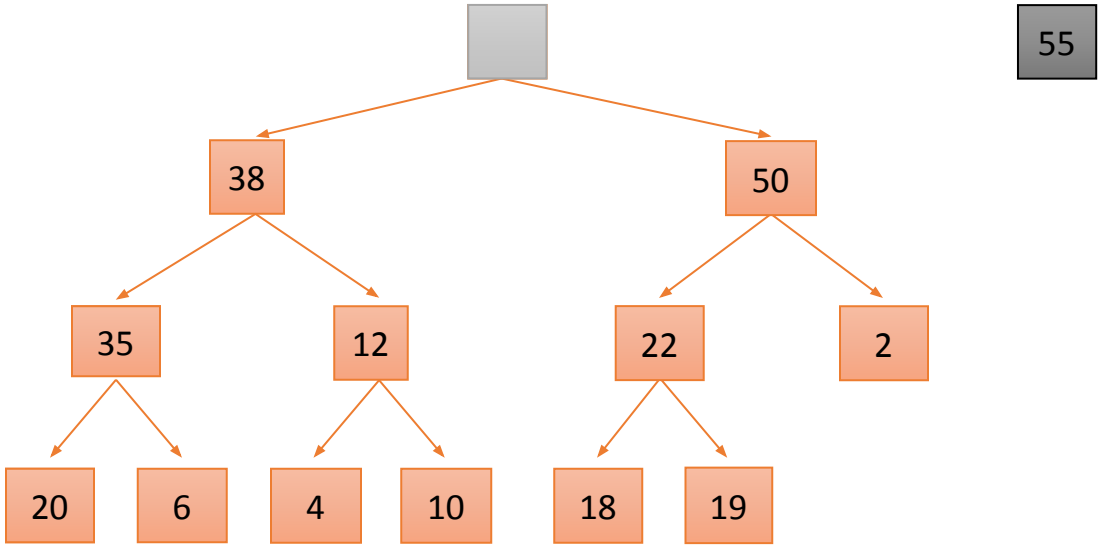
Time is O(1)



55

1. Return root value

Heap: `poll()` (aka `remove()`)



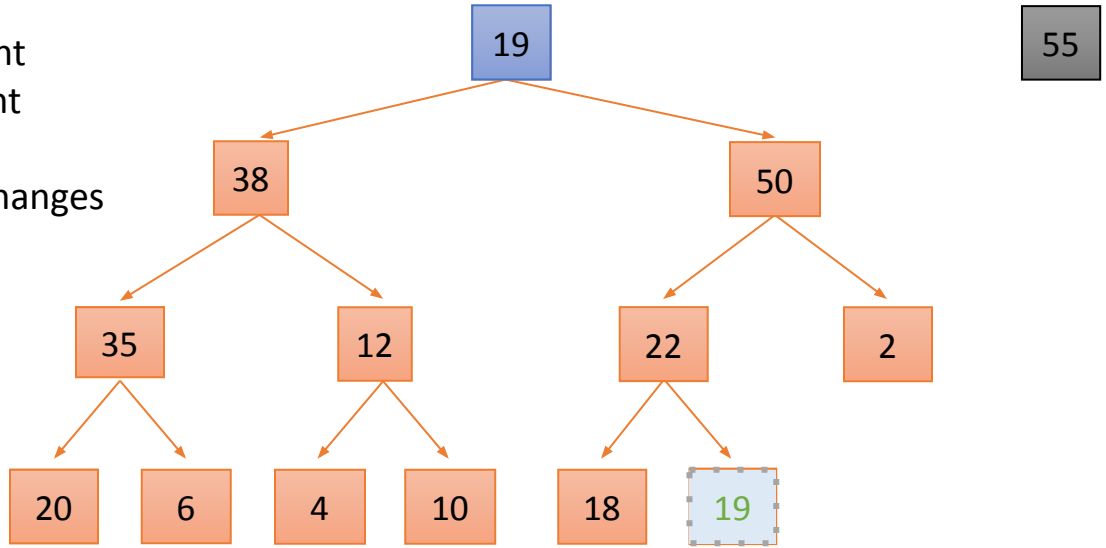
1. Save root element in a local variable

Heap: `poll()` (aka `remove()`)

Must preserve:

1. Shape invariant
2. Order invariant

Goal: minimize changes



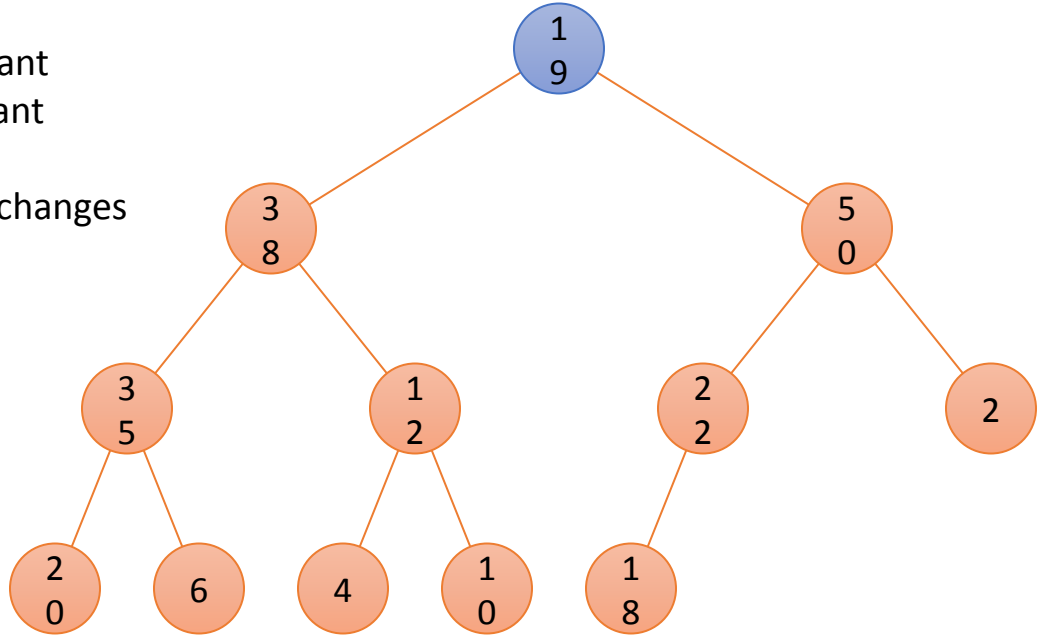
1. Save root element in a local variable
2. Assign last value to root, delete last node.

Exercise: restore the order invariant

Must preserve:

1. Shape invariant
2. Order invariant

Goal: minimize changes

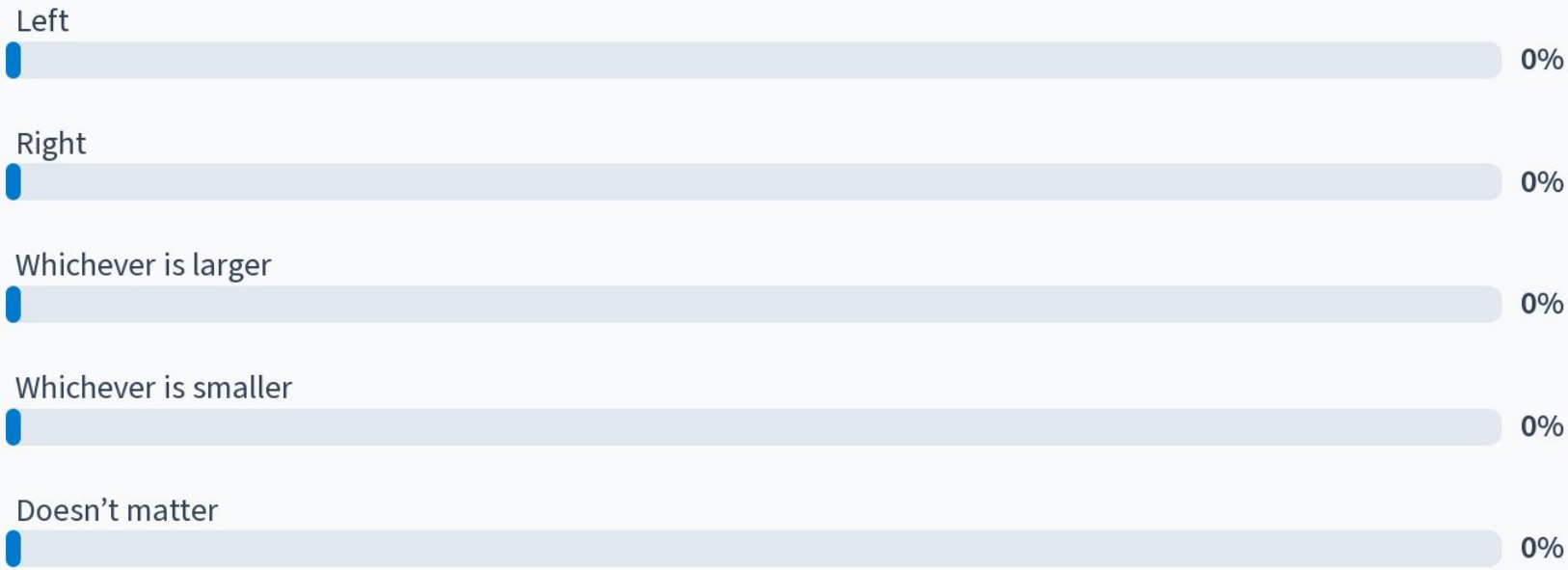


Checkpoint: Bubble down

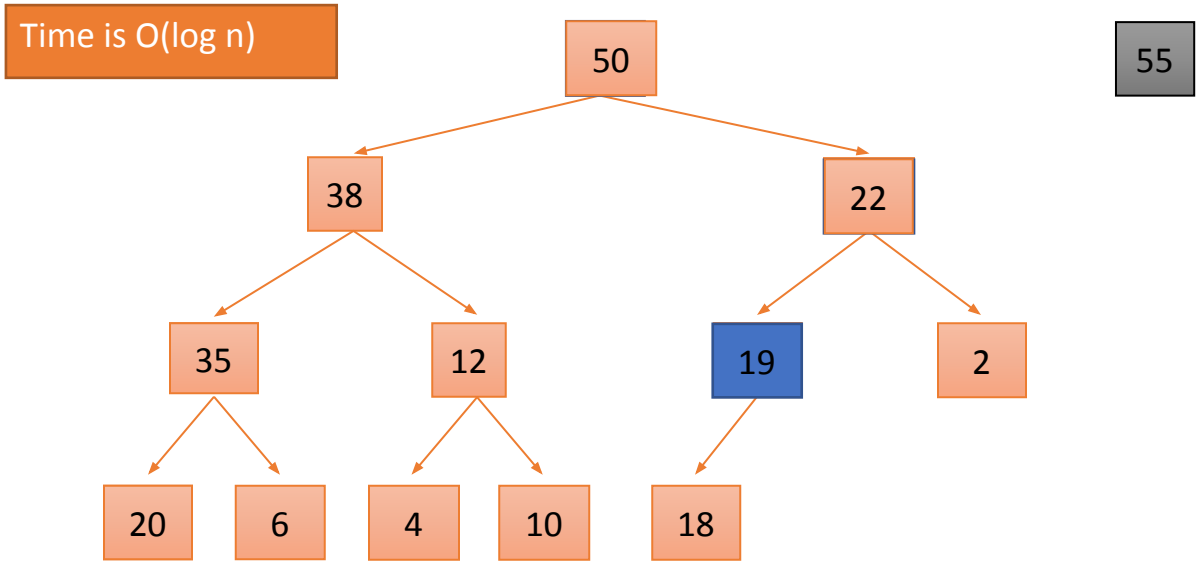
For a max-heap, when “bubbling down”, which child should you swap with?

A. Left
B. Right
C. Whichever is larger
D. Whichever is smaller
E. Doesn’t matter

For a max-heap, when "bubbling down", which child should you swap with?



Heap: `poll()` (aka `remove()`)



1. Save root element in a local variable
2. Assign last value to root, delete last node.
3. While less than a child, switch with bigger child (bubble down)

Heap implementation

Max heap

Tree implementation

```
public class HeapNode<E> {
  private E value;
  private HeapNode left;
  private HeapNode right;
  ...
}
```

But since tree is complete, even more space-efficient implementation is possible…

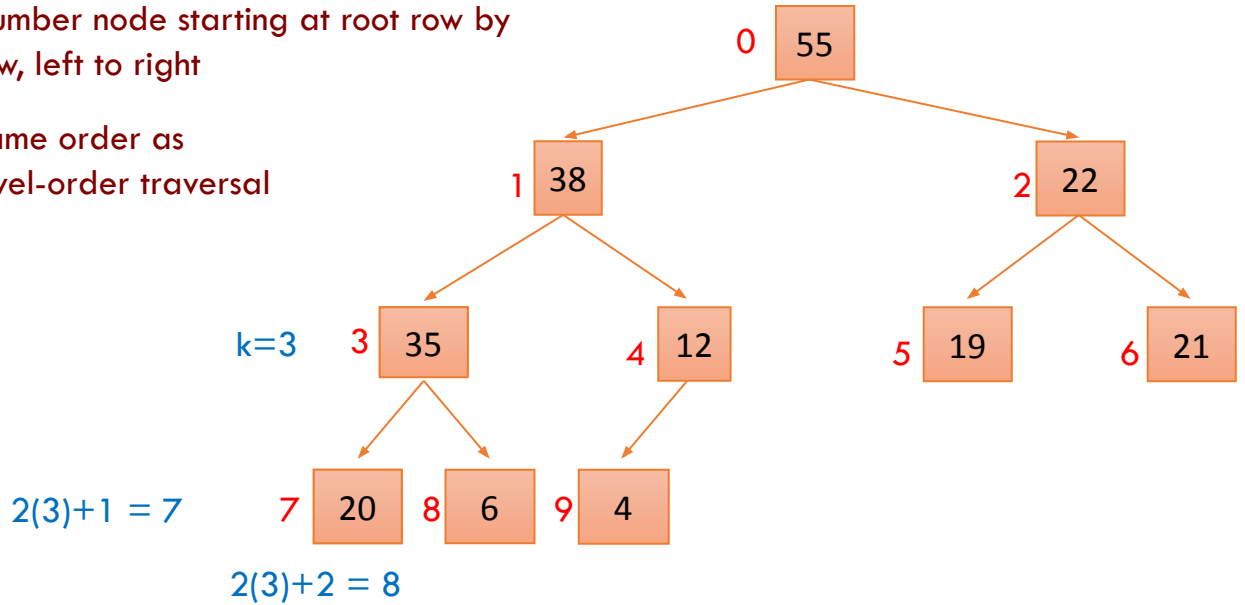
Array implementation

```
public class Heap<E> {
  /** represents a complete binary tree in `heap[0..size)` */
  private E[] heap;
  private int size;
  ...
}
```

Numbering tree nodes

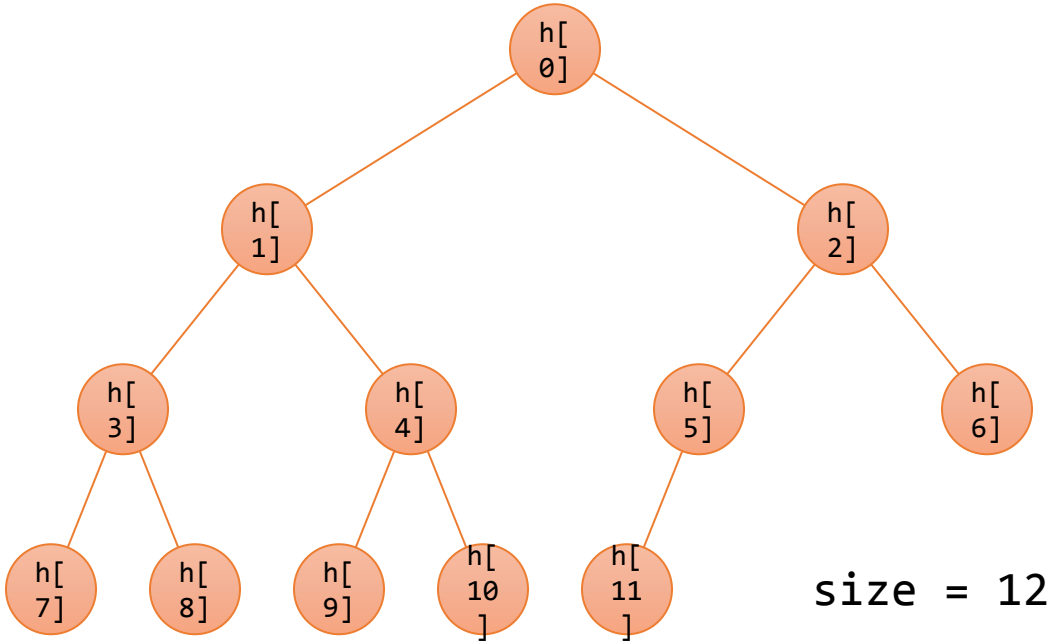
Number node starting at root row by row, left to right

Same order as level-order traversal



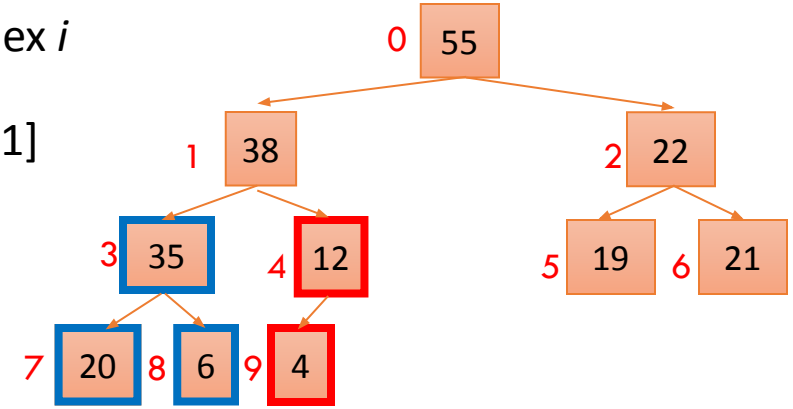
Children of node k are nodes 2k+1 and 2k+2
Parent of node k is node (k-1)/2

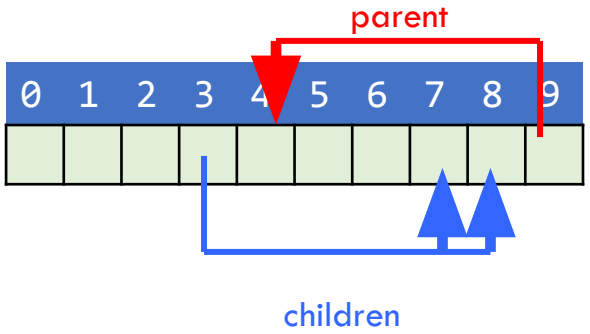
Tree nodes as array elements

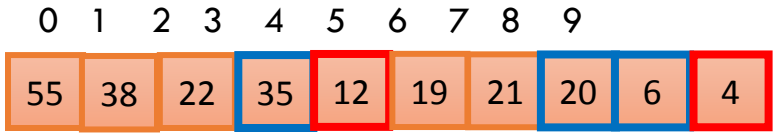


Represent tree with array

- Store node number *i* in index *i* of array b
- Children of b[*k*] are b[2*k* + 1] and b[2*k* + 2]
- Parent of b[*k*] is b[(*k*-1)/2]

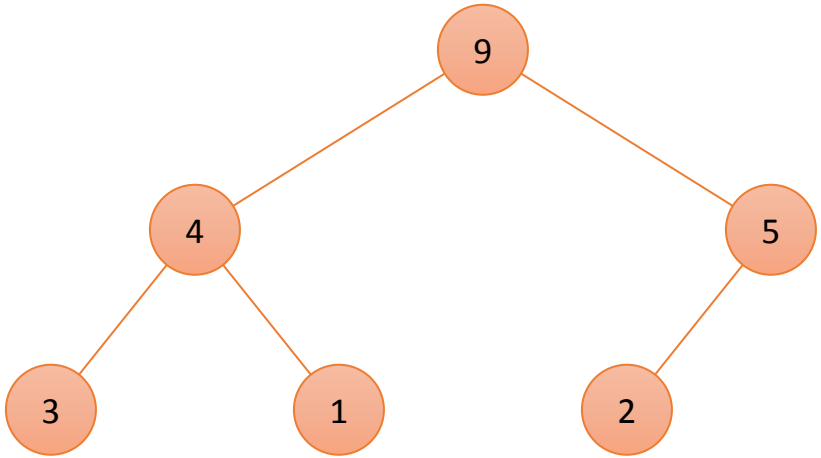






Exercise: map tree to array

What is the array representation of this tree?



A. {1, 2, 3, 4, 5, 9}

B. {3, 1, 2, 4, 5, 9}

C. {9, 4, 3, 1, 5, 2}

D. {9, 4, 5, 3, 1, 2}

Heap Representation

A: {1, 2, 3, 4, 5, 9}

0%

B: {3, 1, 2, 4, 5, 9}

0%

C: {9, 4, 3, 1, 5, 2}

0%

D: {9, 4, 5, 3, 1, 2}

0%

Demo code

Poll 2

Here's a heap, stored in an array:

[9 5 2 1 2 2]

What is the state of the array after execution of add(6)? Assume the existing array is large enough to store the additional element.

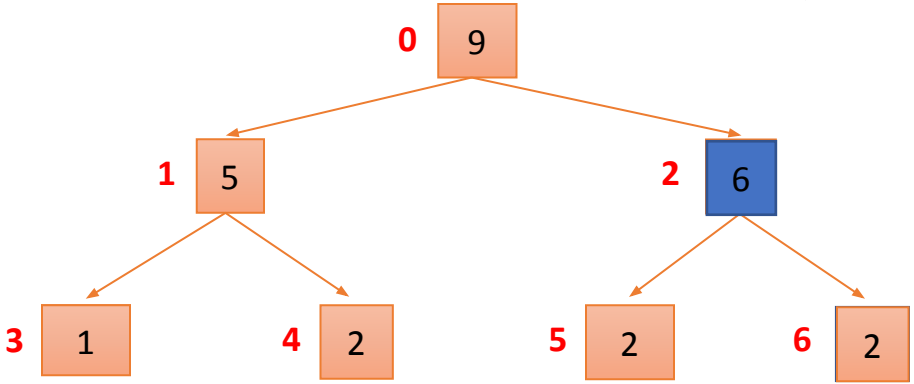
A. [9 5 2 1 2 2 6]
B. [9 5 6 1 2 2 2]
C. [9 6 5 1 2 2 2]
D. [9 6 5 2 1 2 2]

Poll 2

Here's a heap, stored in an array:

[9 5 2 1 2 2] ⇒ [9 5 6 1 2 2 2]

Write the array after execution of add(6)



Specifying priorities

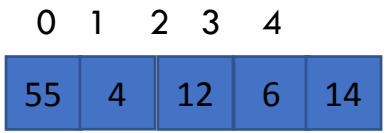
- Use element ordering: `Comparable` or `Comparator`
  - Example: Assignments ordered by their due date
  - Used by `java.util.PriorityQueue<E>` (min-heap)
- Separate priority values: heap stores (element, priority) pairs

Removing values, changing priorities

- *O*(*N*) to search whole tree for value
- Optimization: maintain an "index": for each value, remember which array index it is stored at
  - `Map<V, Integer>`
  - If Map's `get()`, `put()` are *O*(1), will not change heap's complexity analysis
- With index, `remove()`, `update()` are *O*(log *N*)
  - Remove: replace with rightmost leaf, bubble down (like `poll()`)
  - Update: bubble up or down if new priority is bigger/smaller

Heapsort

Heapsort



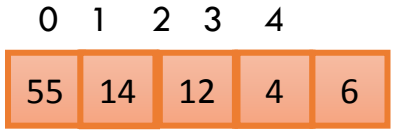
Goal: sort array in place

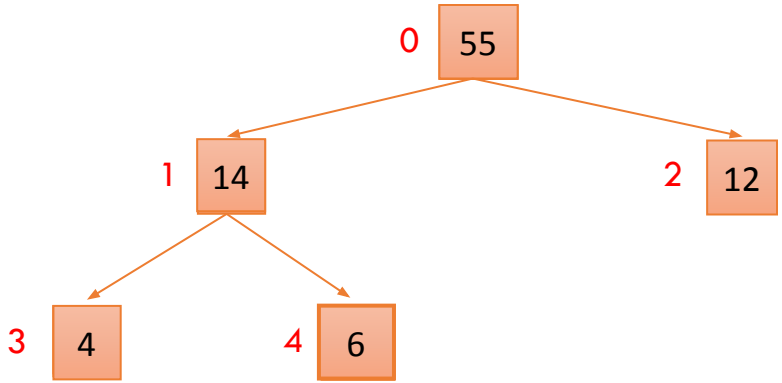
Approach:

1. turn array into a heap
2. poll repeatedly

Heapsort

// Make b[0..n-1] into a max-heap (in place)





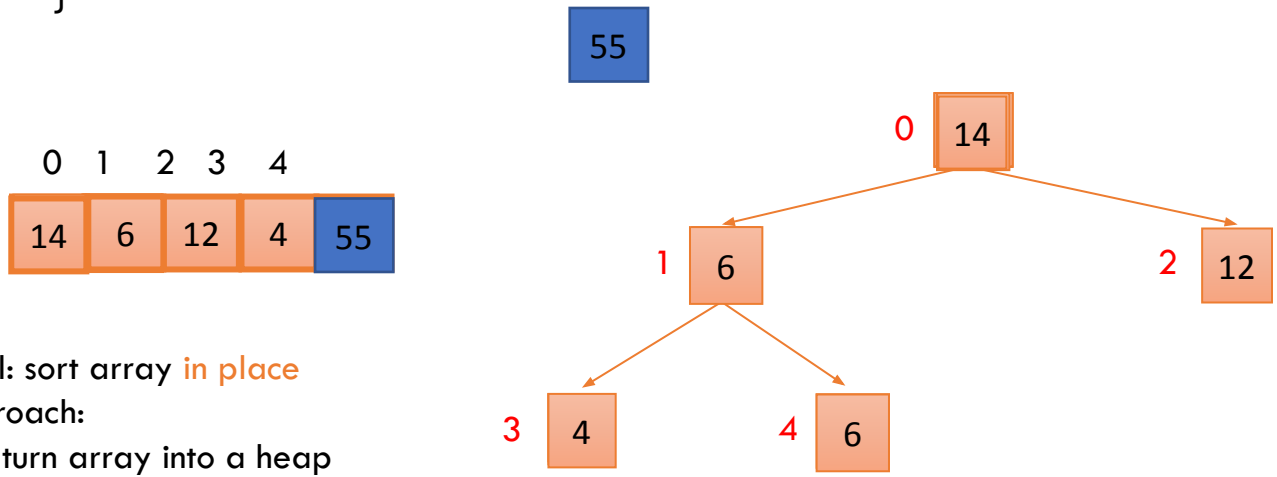
Goal: sort array in place

Approach:

1. turn array into a heap
2. poll repeatedly

Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is
sorted
for (k = size-1; k > 0; k--) {
    b[k] = poll();  // i.e., take max element out of heap.
}
```



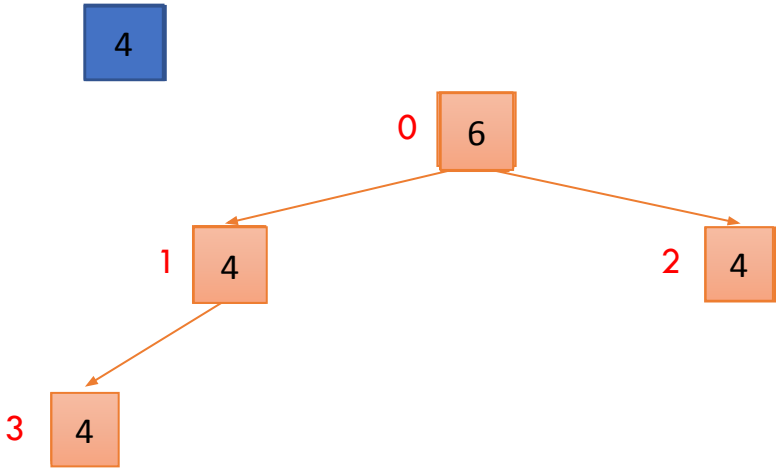
Goal: sort array in place

Approach:

1. turn array into a heap
2. poll repeatedly

Heapsort

```
// Make b[0..n-1] into a max-heap (in place)
// inv:   b[0..k] is a heap, b[0..k] <= b[k+1..], b[k+1..] is
sorted
for (k = size-1; k > 0; k--) {
    b[k] = poll();  // i.e., take max element out of heap.
}
```



Goal: sort array in place

Approach:

1. turn array into a heap
2. poll repeatedly