CPSC 259: Data Structures and Algorithms for Electrical Engineers

Binary Heaps and Heapsort

Textbook Reference:
Thareja first edition: Chapter 12, pages 501-506
Thareja second edition: Chapter 12, pages 361-365

Hassan Khosravi

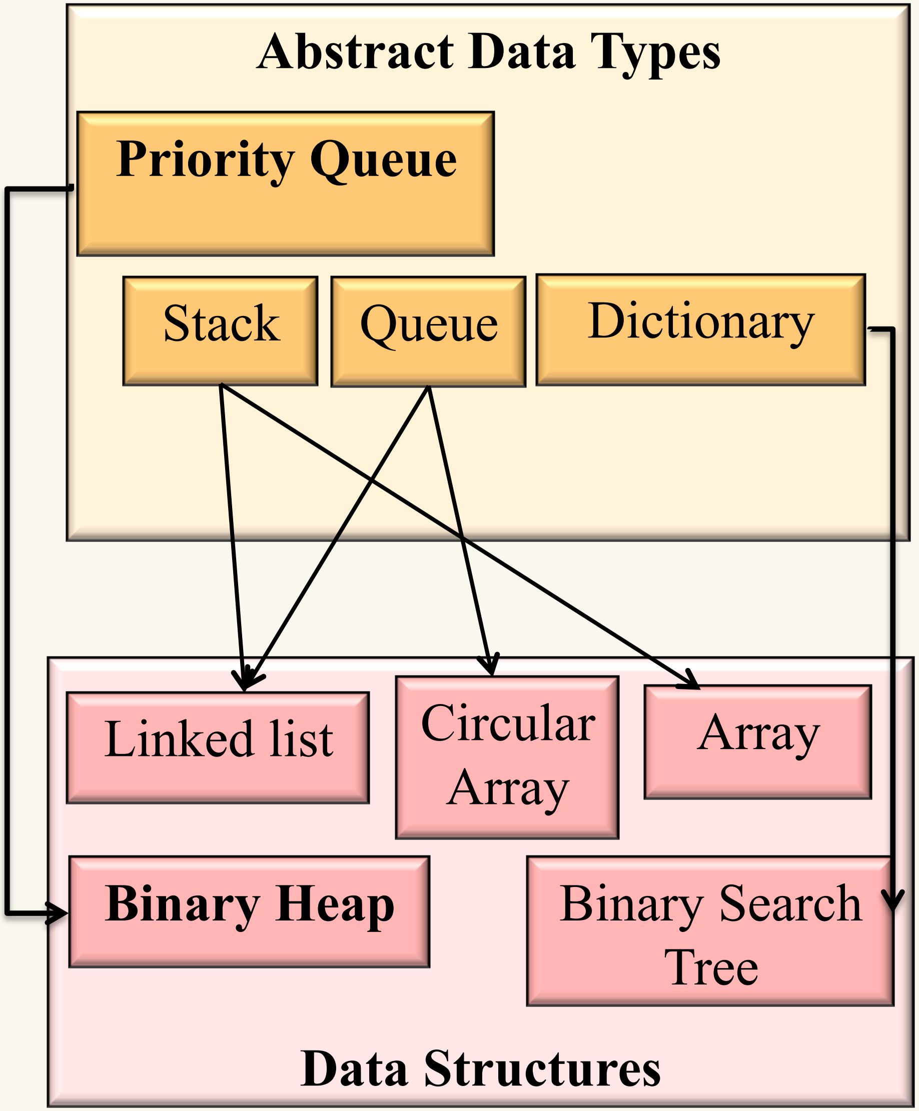
Learning Goals

- Provide examples of appropriate applications for priority queues and heaps.
- Implement and manipulate a heap using an array as the underlying data structure.
- Describe and use the Heapify (“Build Heap”) and Heapsort algorithms.
- Analyze the complexity of operations on a heap.

CPSC 259 Administrative Notes

- MT2 on Friday
  - See course website for details
  - covers all of the course material up to and including what we cover today
- No Lecture on Wednesday
- No labs this week.
- Quiz 3 Marks are released
- PeerWise grades for second call are posted
- PeerWise: Third and final call ends Dec 4th.

CPSC 259 Journey



Abstract Data Types

Priority Queue

Stack

Queue

Dictionary

Linked list

Circular Array

Array

Binary Heap

Binary Search Tree

Data Structures

Tools

Pointers

Dynamic Memory Allocation

Asymptotic Analysis

Recursion

Heapsort

Algorithms

Priority Queues

- Let’s say we have the following tasks.

  2 - Water plants
  5 - Order cleaning supplies
  1 - Clean coffee maker
  3 - Empty trash
  9 - Fix overflowing sink
  2 - Shampoo carpets
  4 - Replace light bulb
  1 - Remove pencil sharpener shavings

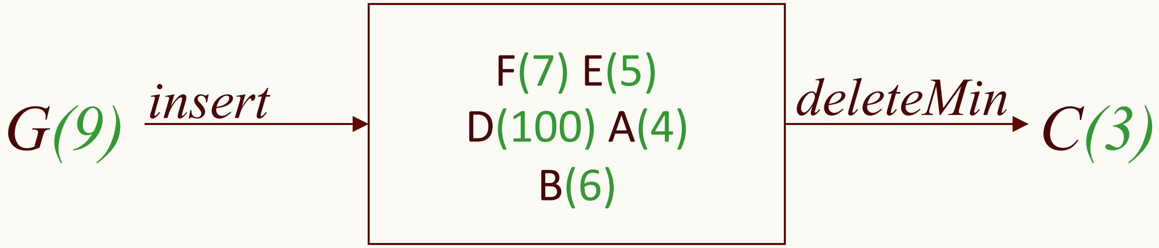
  We are interested in finding the task with the highest priority quickly.

Priority Queues

- A collection organized so as to permit fast access to and removal of the largest/smallest element
  - *Prioritization* is a weaker condition than ordering.
  - Order of insertion is irrelevant.
  - Element with the highest priority (whatever that means) comes out next.
    - Not really a *queue*: not a FIFO

Priority Queue ADT

- Priority Queue operations
  - create
  - destroy
  - insert
  - deleteMin
  - isEmpty



- Priority Queue property: for two elements in the queue, *x* and *y*, if *x* has a lower priority value than *y*, *x* will be deleted before *y*.

Heaps and Priority Queues

- A **priority queue** is an abstract data type (ADT) that maintains a *bag* of items.
  - What is the difference between a set and a bag?
    - A set does not contain duplicates.
- Two or more distinct items in a priority queue can have the same priority. If all items have the same priority, you might think a priority queue should behave like a queue, but it may not. We' ll see why, shortly.
- A binary heap can implement a priority queue, efficiently.

Applications of the Priority Q

- Hold jobs for a printer in order of length
- Manage limited resources such as bandwidth on a transmission line from a network router
- Simulate events (simulation time used as the priority)
- Sort numbers
- Anything *greedy*: an algorithm that makes the “locally best choice” at each step

Naïve Priority Q Data Structures

- Let's use an unsorted list (could be implemented with either an Array or Linked List)
- Running time of insert ?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use an unsorted list (could be implemented with either an Array or Linked List)
- Running time of insert ?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use an unsorted list (could be implemented with either an Array or Linked List)
- Running time of deleteMin?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use an unsorted list (could be implemented with either an Array or Linked List)
- Running time of deleteMin?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let’s use a sorted list (could be implemented with either an Array or Linked List)
- Running time of insert ?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use a sorted list (could be implemented with either an Array or Linked List)
- Running time of insert ?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use a sorted list (could be implemented with either an Array or Linked List)
- Running time of deleteMin?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use a sorted list (could be implemented with either an Array or Linked List)
- Running time of deleteMin?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let’s use a Binary Search Tree (could be implemented with either an Array or Linked List)
- Worst case running time of insert ?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let’s use a Binary Search Tree (could be implemented with either an Array or Linked List)
- Worst case running time of insert ?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use a Binary Search Tree (could be implemented with either an Array or Linked List)
- Worst case running time of deleteMin?

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures

- Let's use a Binary Search Tree (could be implemented with either an Array or Linked List)
- Worst case running time of deleteMin?

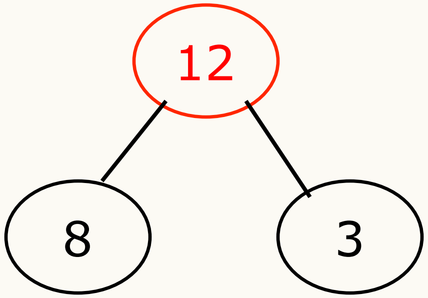
a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. Something else

Naïve Priority Q Data Structures (Summary)

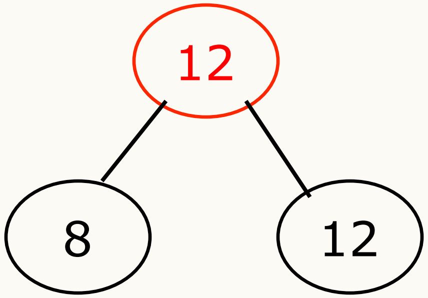
Data Structure	Insert (worst case)	deleteMin (worst case)
Unsorted lists	O(1)	O(n)
Sorted lists	O(n)	O(1)
Binary Search Trees	O(n)	O(n)
Binary Heaps	**O(lg n)**	**O(lg n)**

The heap property

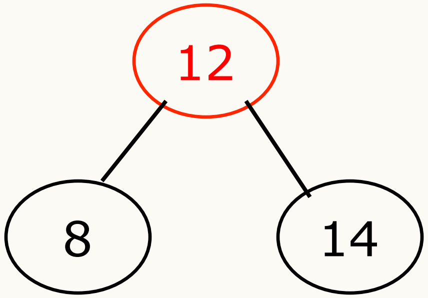
- A node has the heap property if the priority of the node is as high as or higher than the priority of its children.



Red node has heap property



Red node has heap property

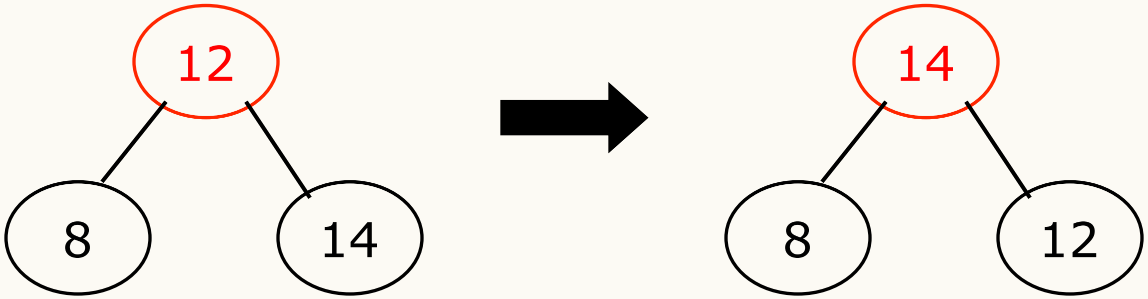


Red node does not have heap property

- All leaf nodes automatically have the heap property.
- A binary tree is a heap if *all* nodes in it have the heap property.

Percolate-up

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the child with the higher priority.



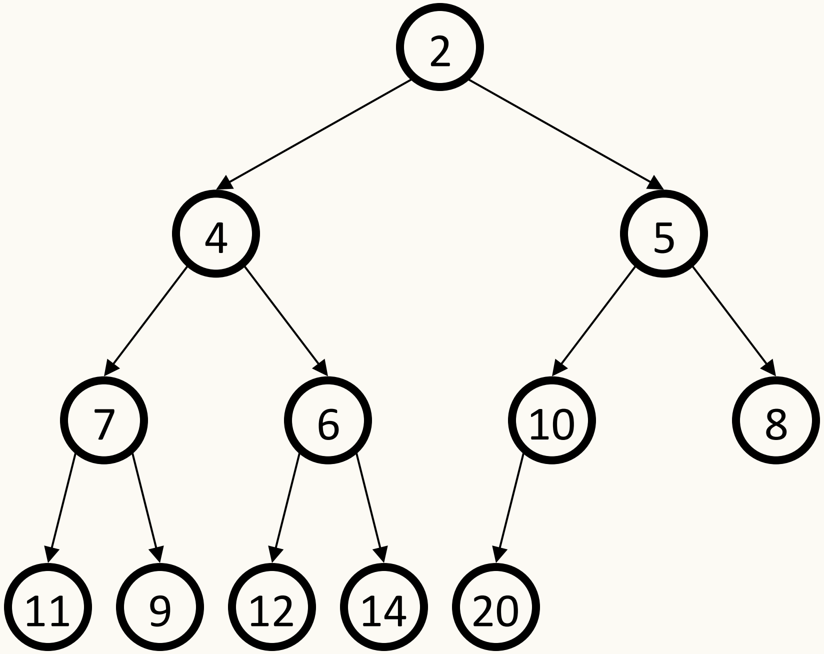
Red node does not have heap property

Red node has heap property

- This is sometimes called Percolate-up (sifting up).
- Notice that the child may have *lost* the heap property.

Binary Heap Priority Q Data Structure

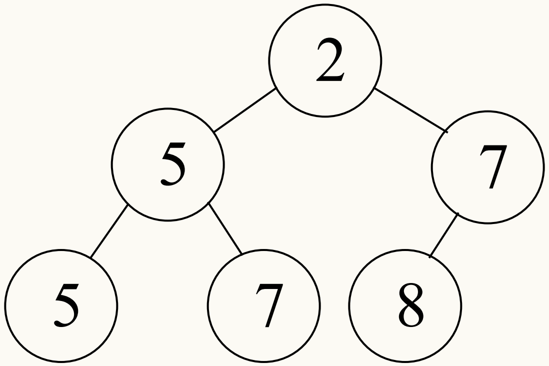
- Heap-order property
  - parent's key is less than or equal to children's keys
  - result: minimum is always at the top
- Structure property
  - "nearly complete tree"

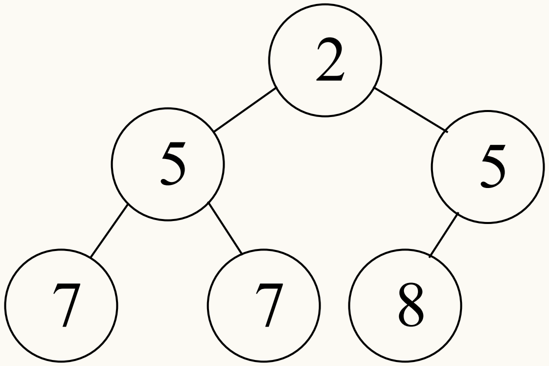


depth is always O(log n);
next open location always known

***WARNING**: this has NO SIMILARITY to the "heap" you hear about when people say "objects you create with **`new`** go on the heap".* 25

It is important to realize that two binary heaps can contain the same data but the data may appear in different positions in the heap:



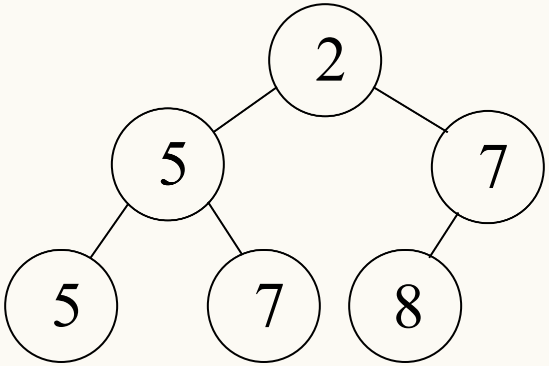


Both of the minimum binary heaps above contain the same data: 2, 5, 5, 7, 7, and 8.

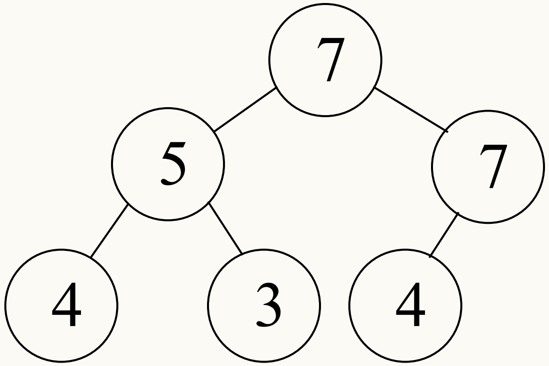
Even though both heaps satisfy all the properties necessary of a minimum binary heap, the data is stored in different positions in the tree.

Min-heap and Max-heap

(There’ s also a **maximum binary heap**, where “parent ≤ each child” simply changes to “parent ≥ each child”.)



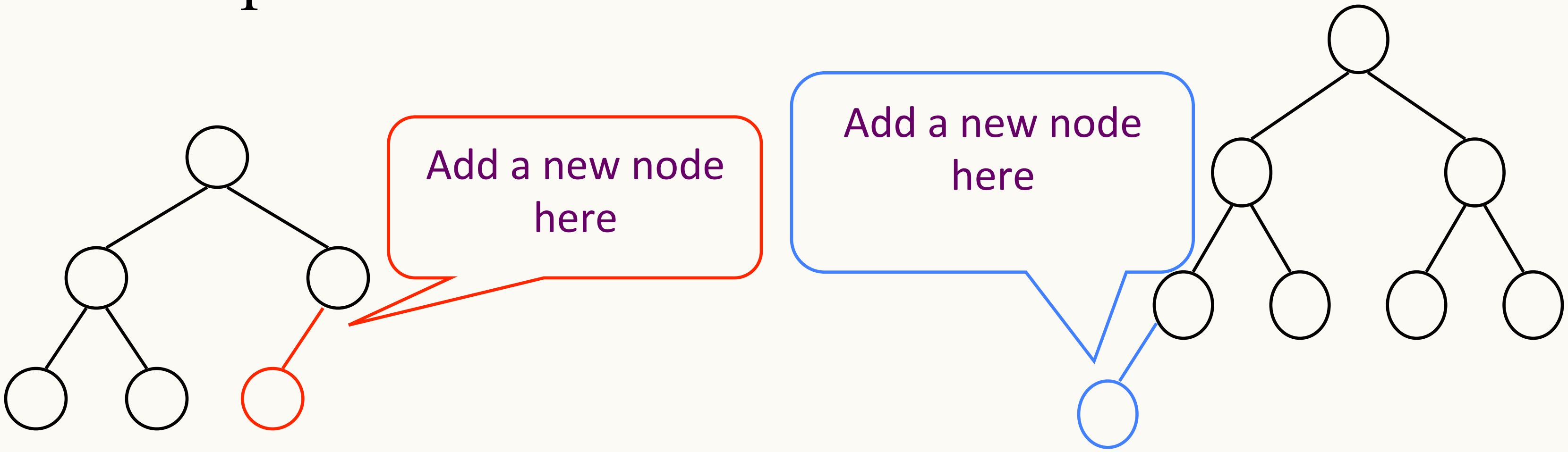
Min-heap



Max-heap

Constructing a heap I (Naïve approach)

- A tree consisting of a single node is automatically a heap.
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level.
  - If the deepest level is full, start a new level with the leftmost position.
- Examples:



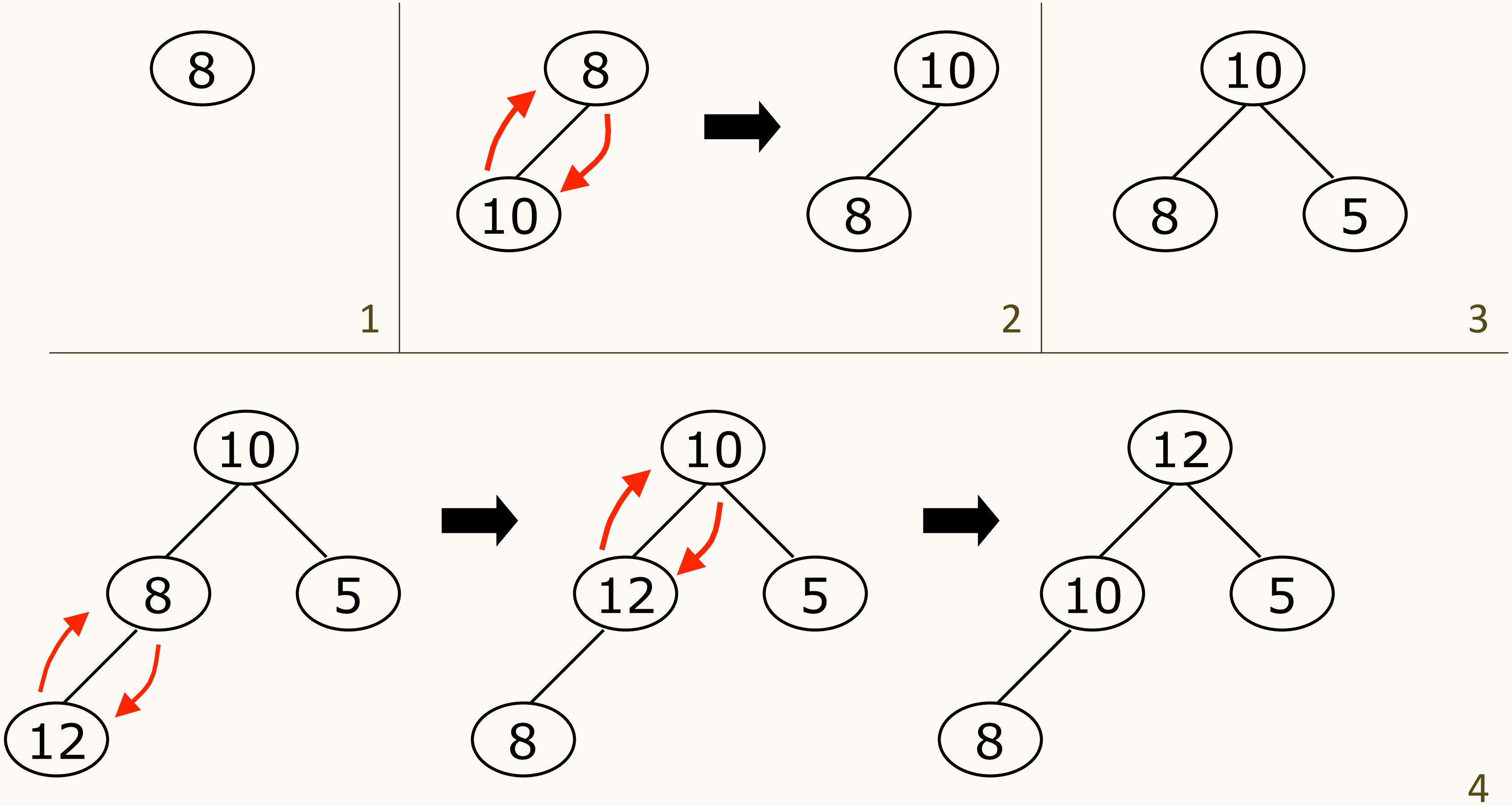
Add a new node here

Add a new node here

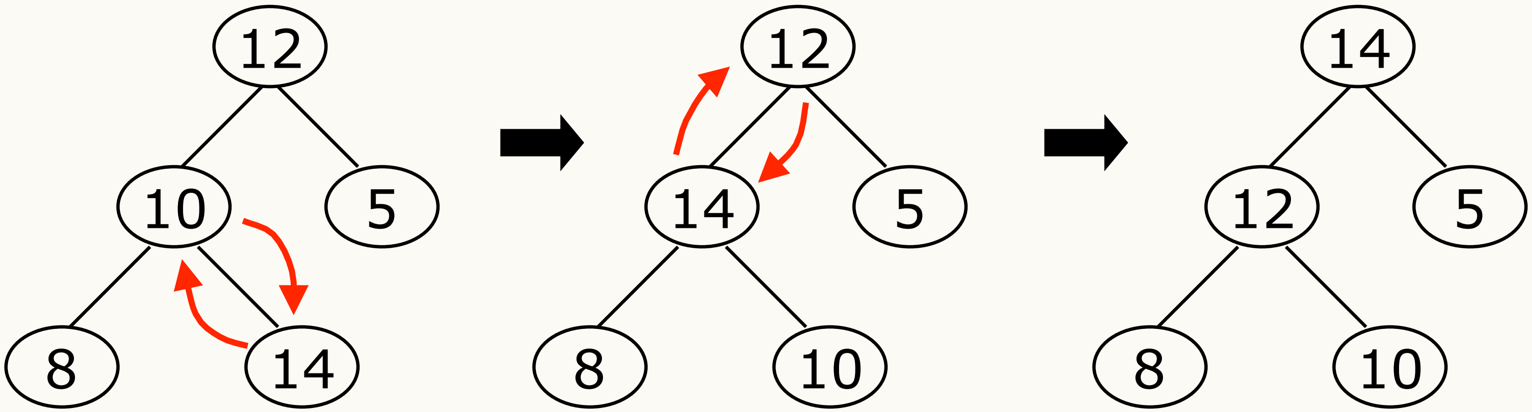
Constructing a heap II (Naïve approach)

- Each time we add a node, we may destroy the heap property of its parent node. To fix this, we percolate-up.
- But each time we percolate-up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node.
- We repeat the percolate-up process, moving up in the tree, until either:
  - we reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - we reach the root

Constructing a heap III (Naïve approach)



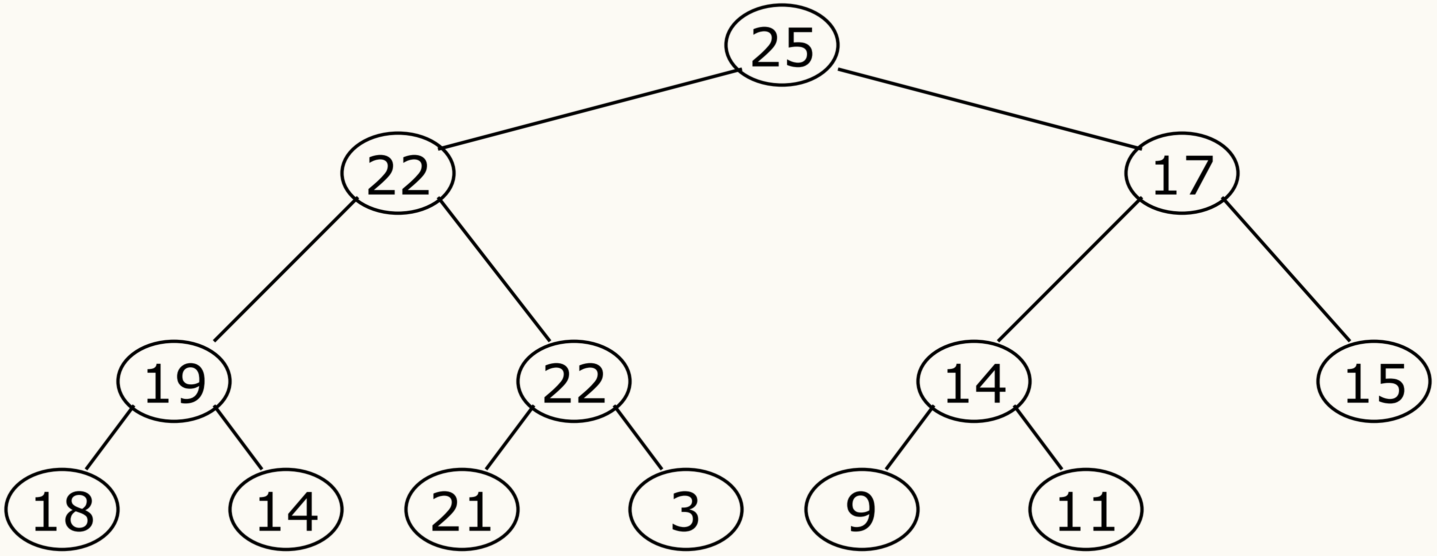
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller.
- The node containing 5 is not affected because its parent gets larger, not smaller.
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally.

A sample heap

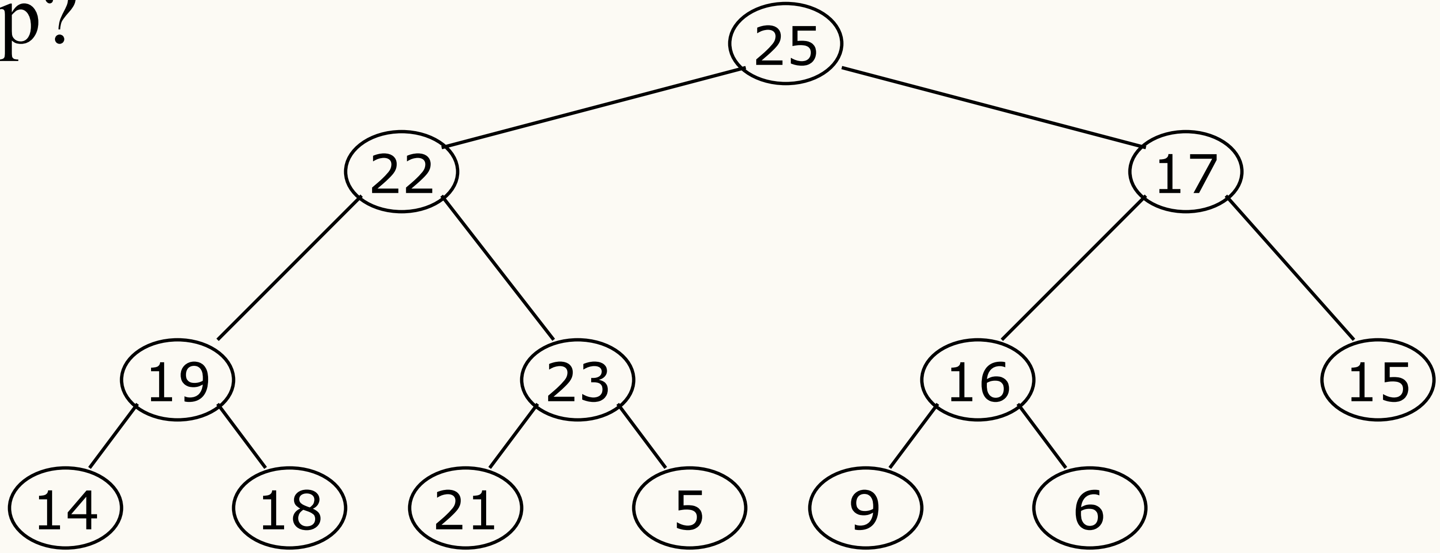
- Here's a sample binary tree after it has been heapified.



- Notice that heapified does *not* mean sorted.
- Heapifying does *not* change the shape of the binary tree; this binary tree is still a nearly complete binary tree.

Clicker Question

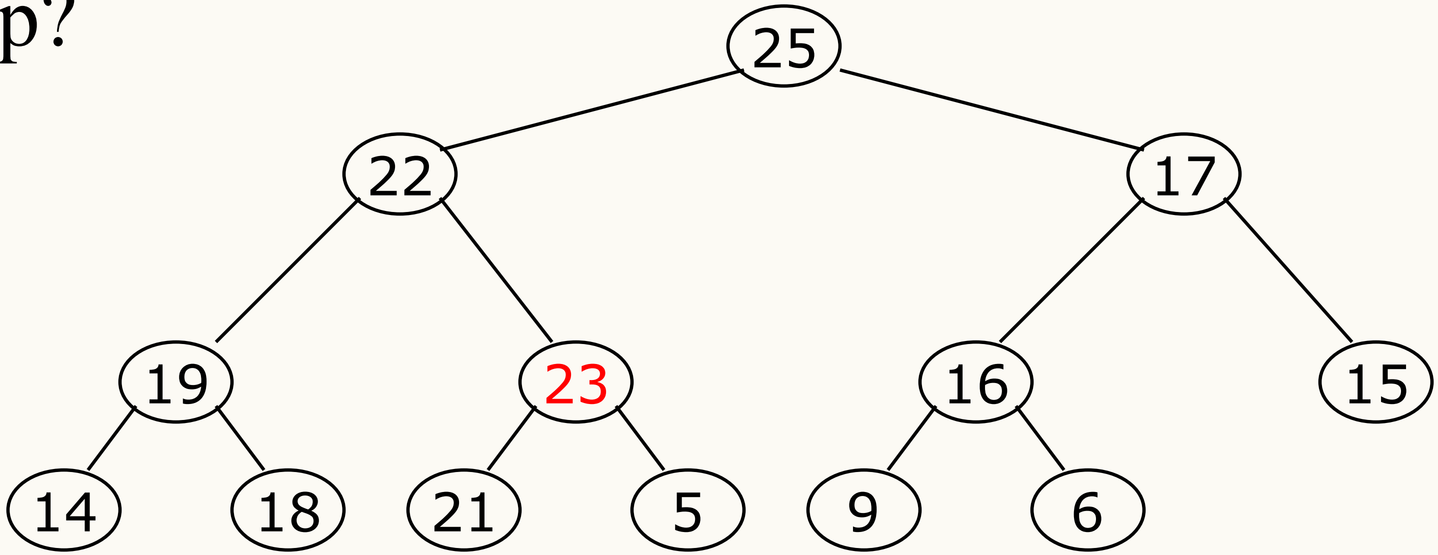
- Is the following binary tree a maximum binary heap?



  - A: It is a maximum binary heap
  - B: It is not a maximum binary heap
  - C: I don’t know

Clicker Question (answer)

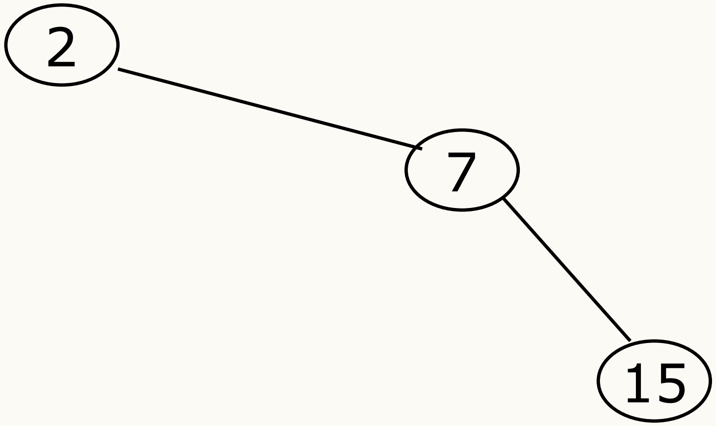
- Is the following binary tree a maximum binary heap?



  - A: It is a maximum binary heap
  - B: It is not a maximum binary heap
  - C: I don’t know

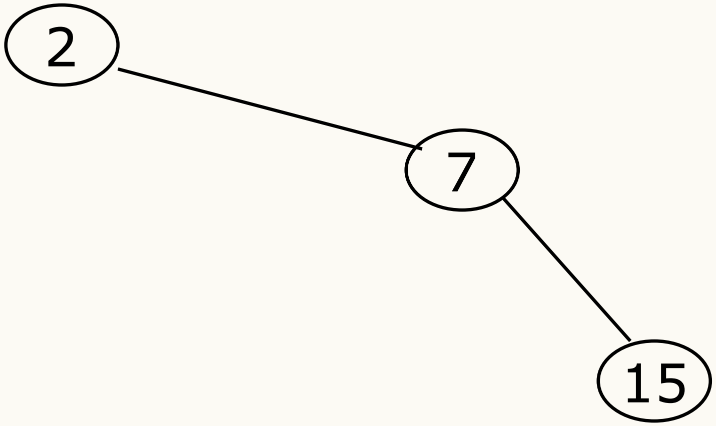
Clicker question

- Which of the following statement(s) are true of the tree shown below?
  - a. It is a binary search tree.
  - b. It is a complete tree.
  - c. It is a Max-heap.
  - d. It is a Min-heap.
  - e. More than one of the above statements is true.



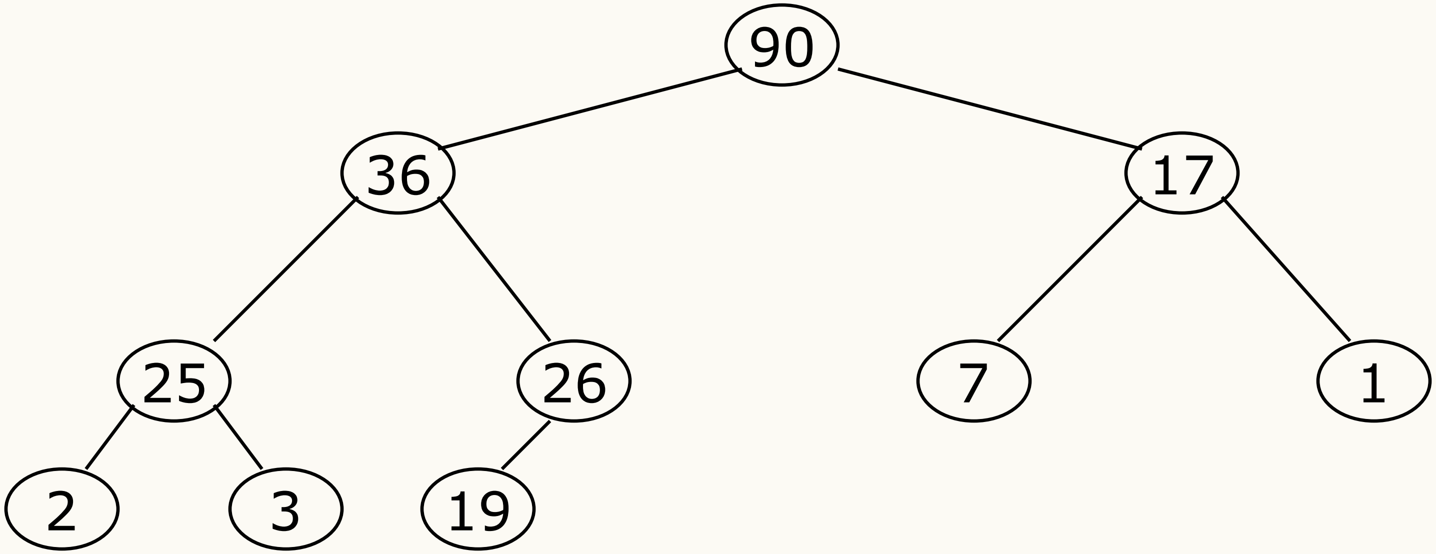
Clicker question

- Which of the following statement(s) are true of the tree shown below?
  - a. It is a binary search tree.
  - b. It is a complete tree.
  - c. It is a Max-heap.
  - d. It is a Min-heap.
  - e. More than one of the above statements is true.



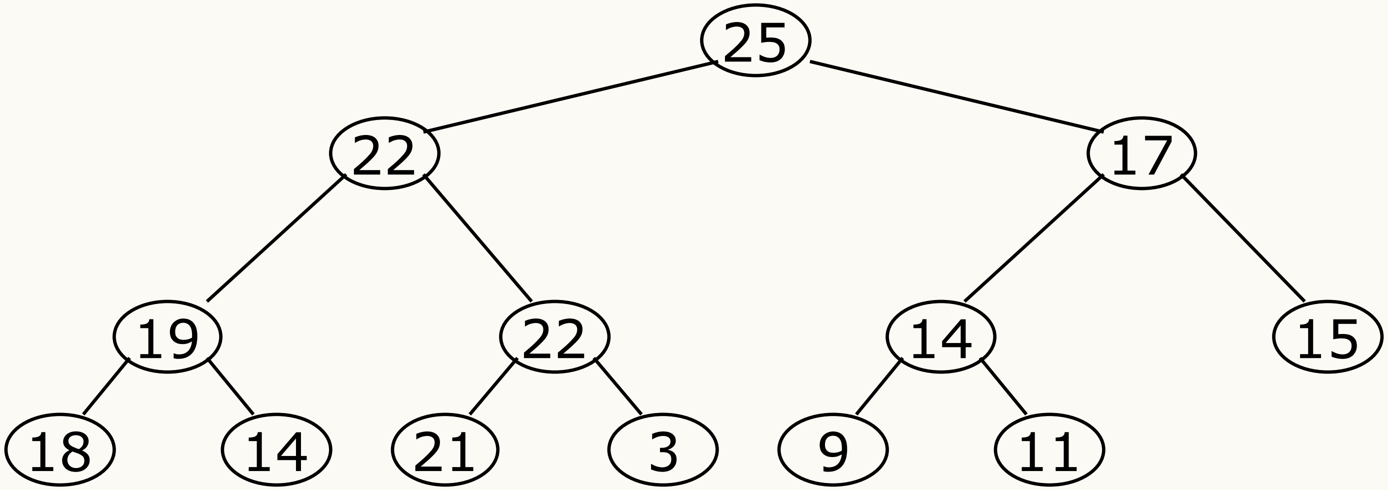
In-class exercise

- Build a binary Max-heap using the following numbers, assuming numbers arrive one at a time, so you don’t have the whole list to start with).
  - 2,7,26,25,19,17,1,90,3,36



See http://visualgo.net/heap.html

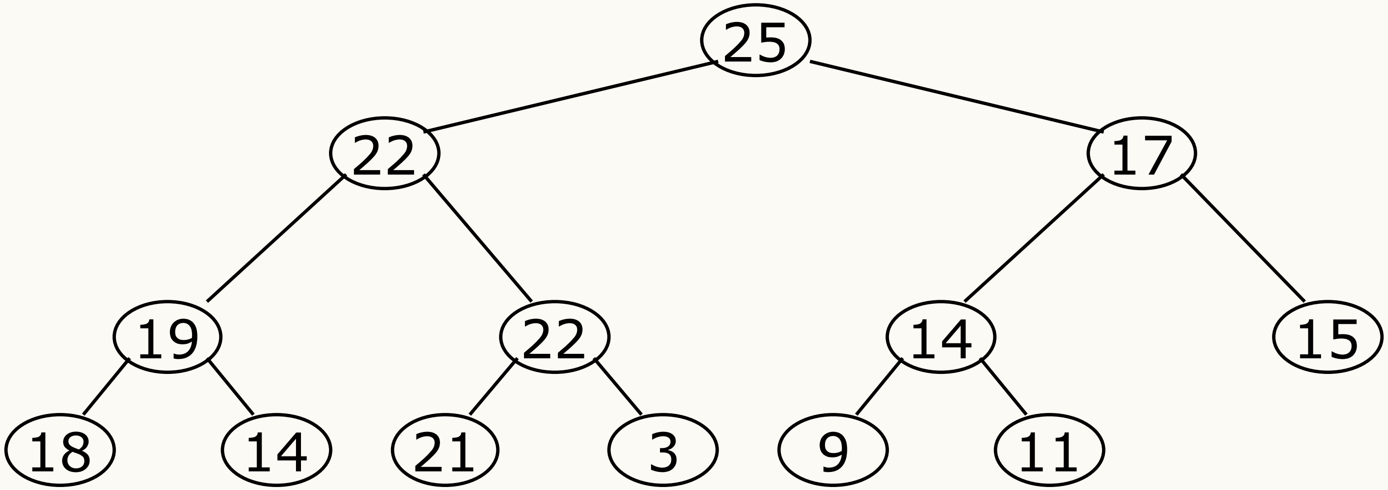
Mapping into an array



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Because of the heap's shape, node values can be stored in an array.

Mapping into an array



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Left child = 2*node + 1

right child= 2*node + 2

parent = floor((node-1)/2)

nextfree = length

root = 0

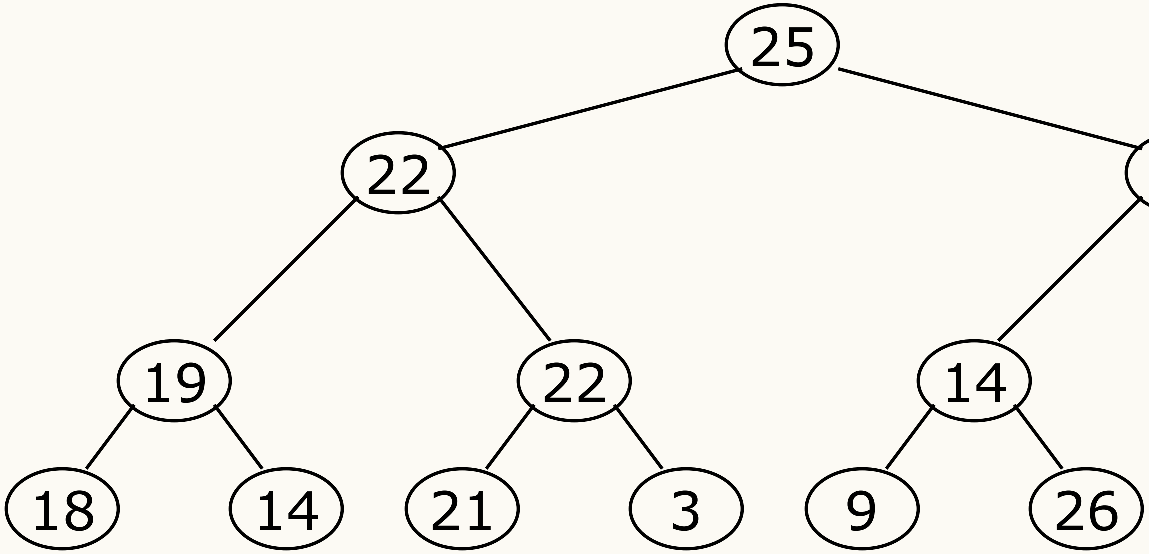
Adding an item to a heap

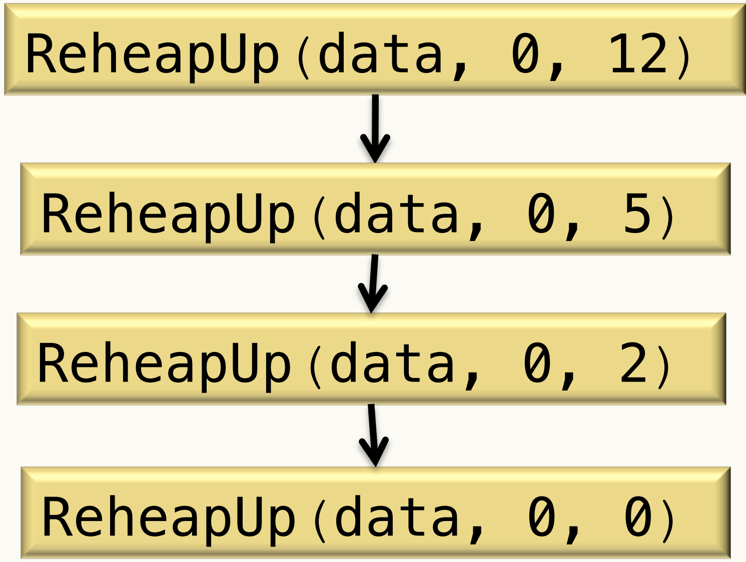
- If a new item is added to the heap, use ReheapUp (percolate-up) to maintain a heap.

```
/* This function performs the Reheapup operation on an array,
to establish heap properties (for a subtree).

 PARAM:   data   – integer array containing the heap
             top    – position of the root
             bottom – position of the added element
 */
void ReheapUp( int * data, int top, int bottom ){
    if (bottom > top) {
        int parent = getparent(bottom);
        if (data[parent] < data[bottom]) {
            swap( &data[parent], &data[bottom]);
            ReheapUp(data, top, parent);
        }
    }
}
```

In-class exercise

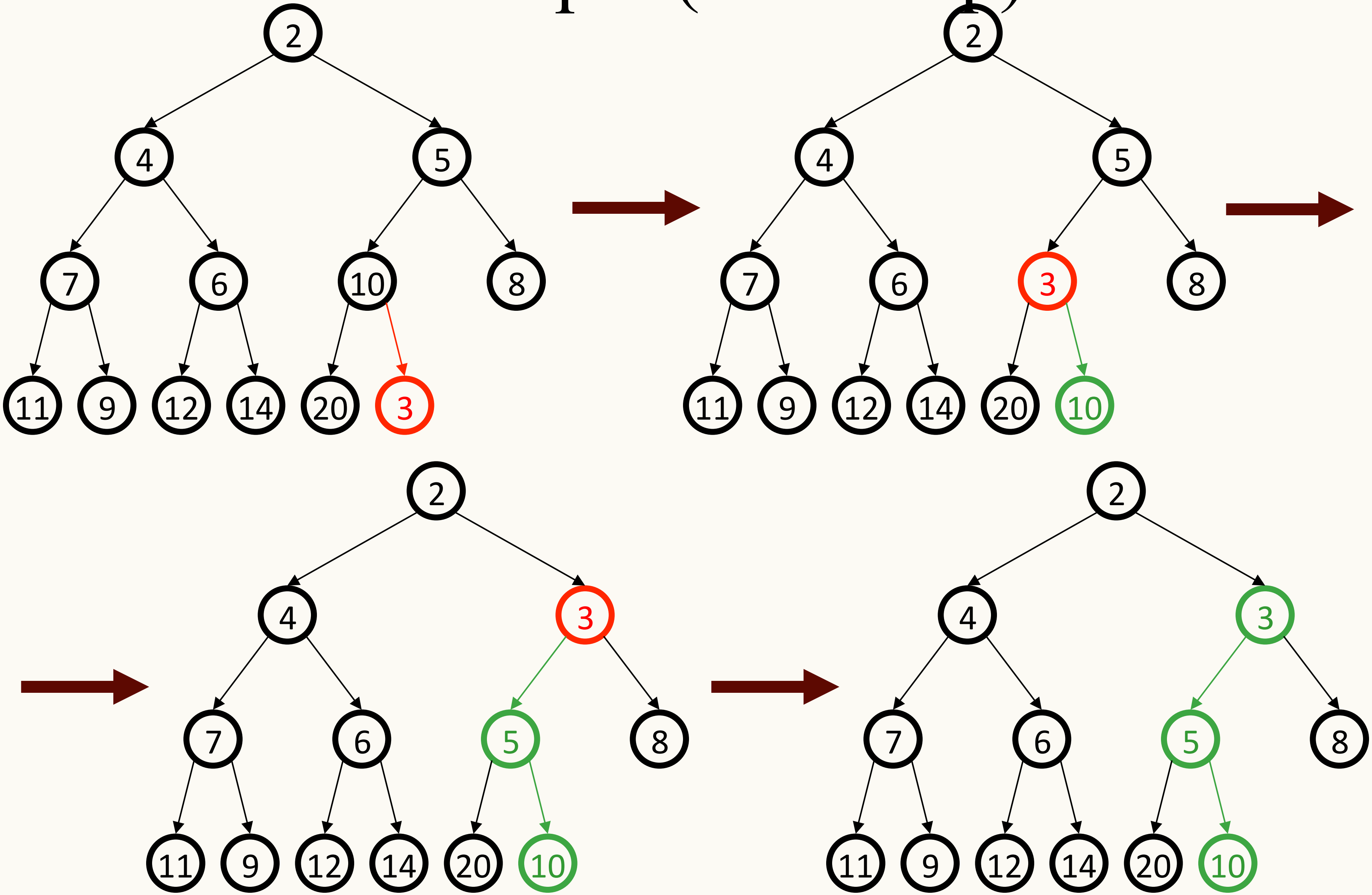
- For the max-heap below draw the recursion tree of `ReheapUp(data, 0, 12)`, where 25 is stored in `data[0]`





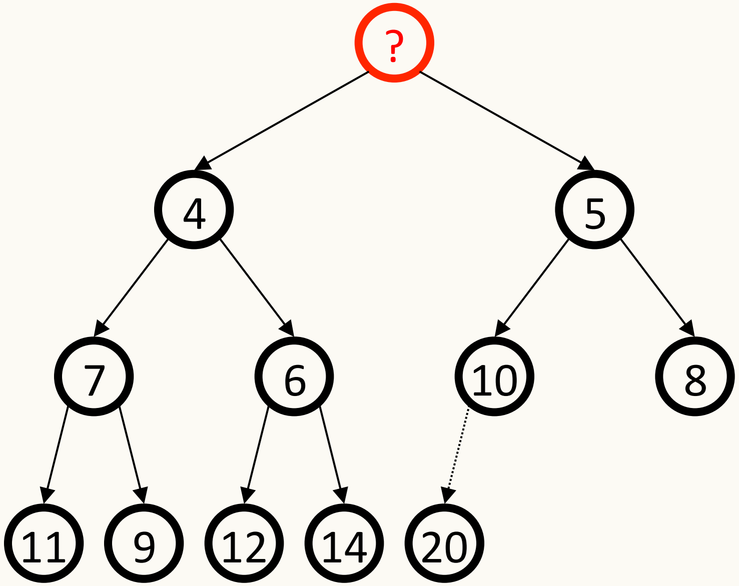
```
ReheapUp(data, 0, 12)
ReheapUp(data, 0, 5)
ReheapUp(data, 0, 2)
ReheapUp(data, 0, 0)
```

Example (min-heap)

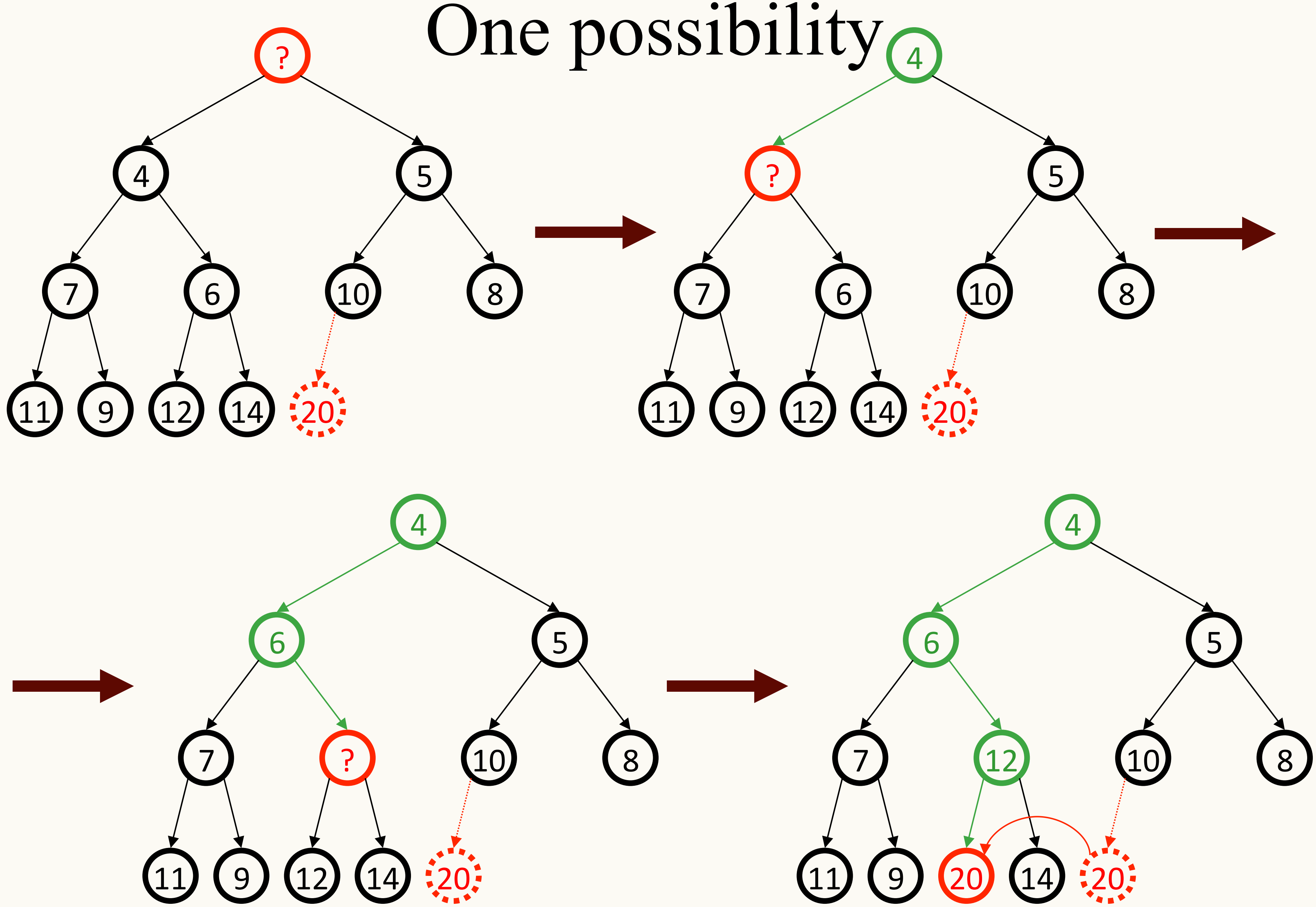


Removing the root (Min-heap example)

- So we know how to add new elements to our heap.
- We also know that root is the element with the highest priority.
- But what should we do once root is removed?
  - Which element should replace root?

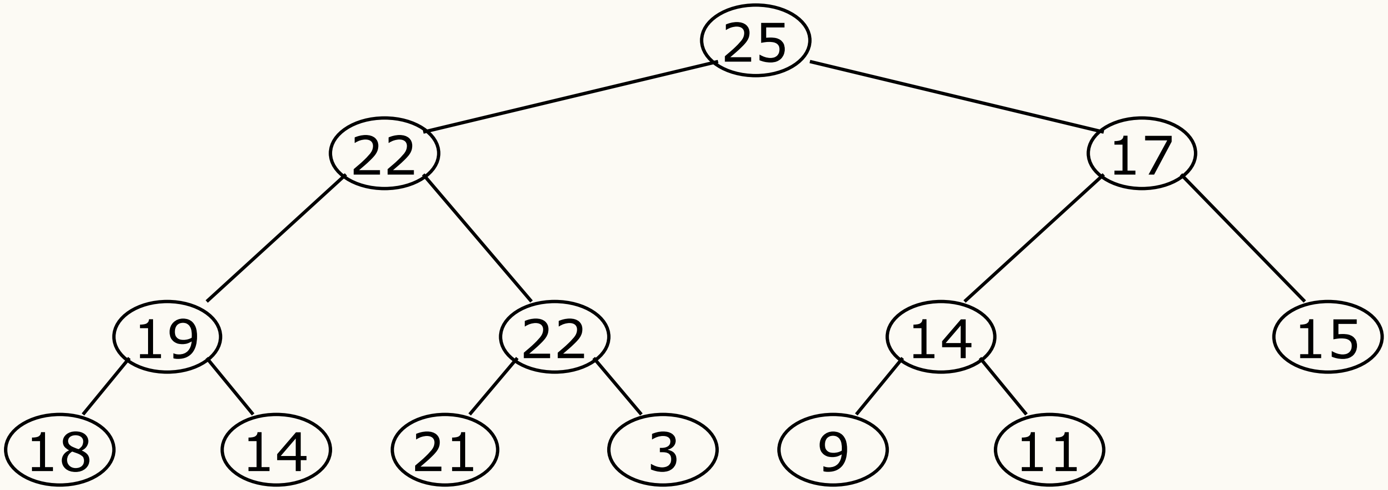


One possibility



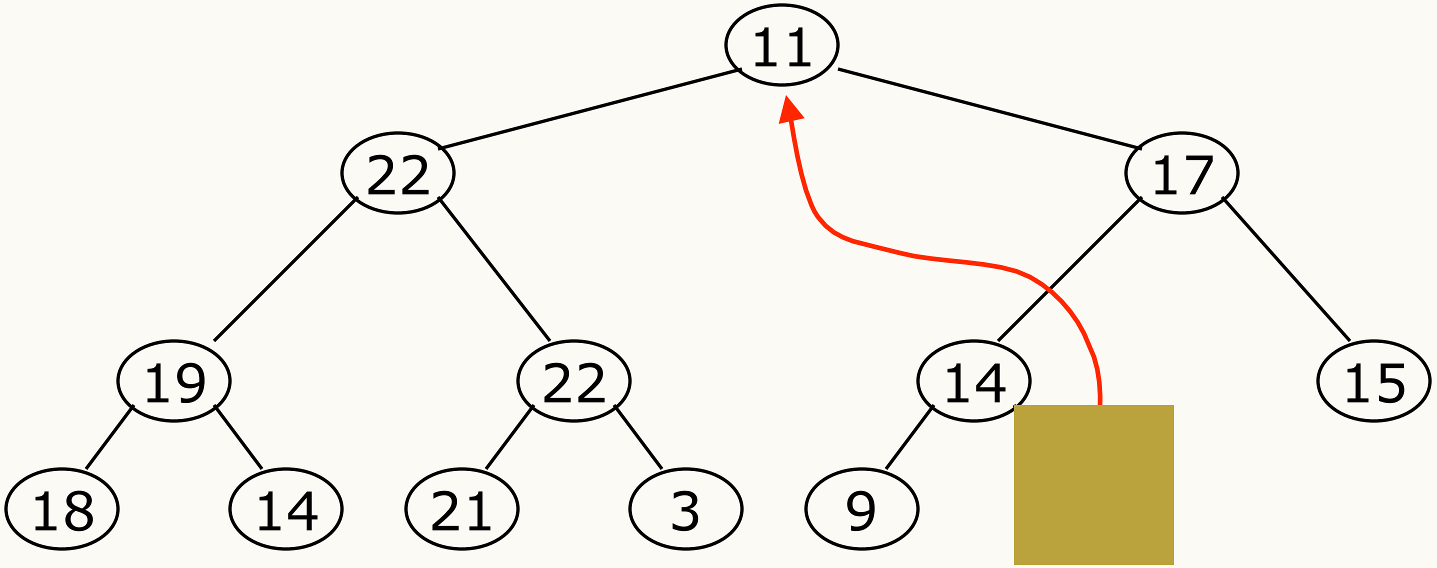
Removing the root (max-heap example)

- So we know how to add new elements to our heap.
- We also know that root is the element with the highest priority.
- But what should we do once root is removed?
  - Which element should replace root?



Removing the root

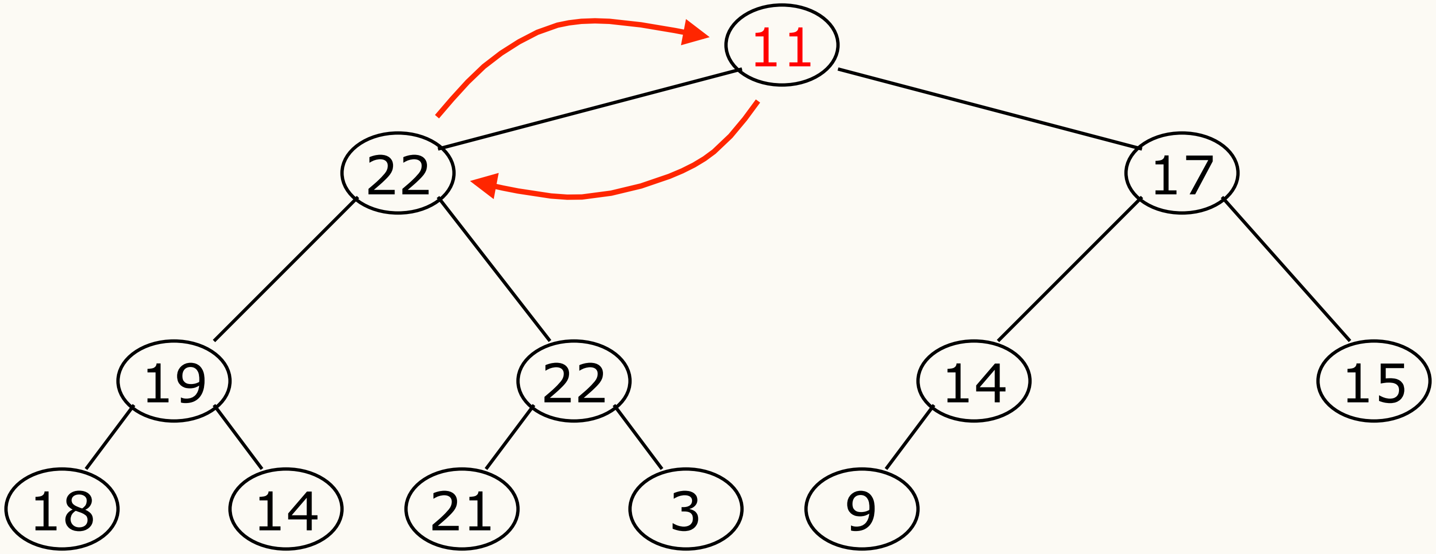
- Suppose we *remove* the root:



- How can we fix the binary tree so it is once again *a nearly complete tree?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root.

The percolate-down method

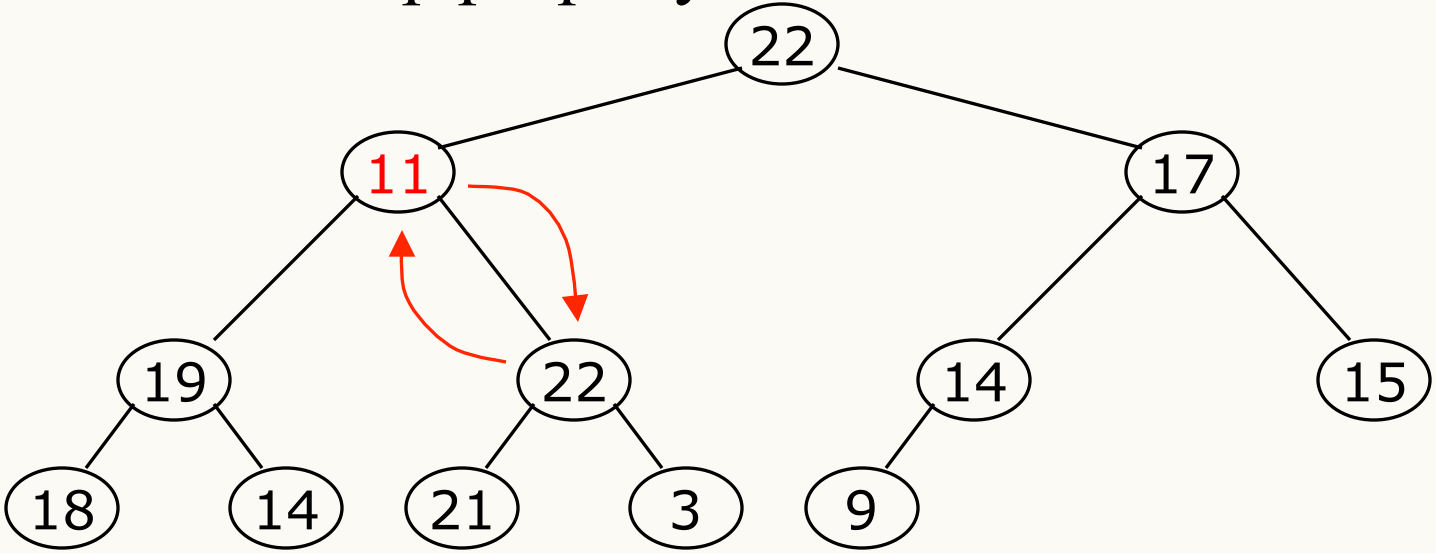
- Our tree is now a nearly complete binary tree, but no longer a heap.
- However, *only the root* lacks the heap property.



- We can percolate-down the root.
- After doing this, one and only one of its children may have lost the heap property.

The percolate-down method

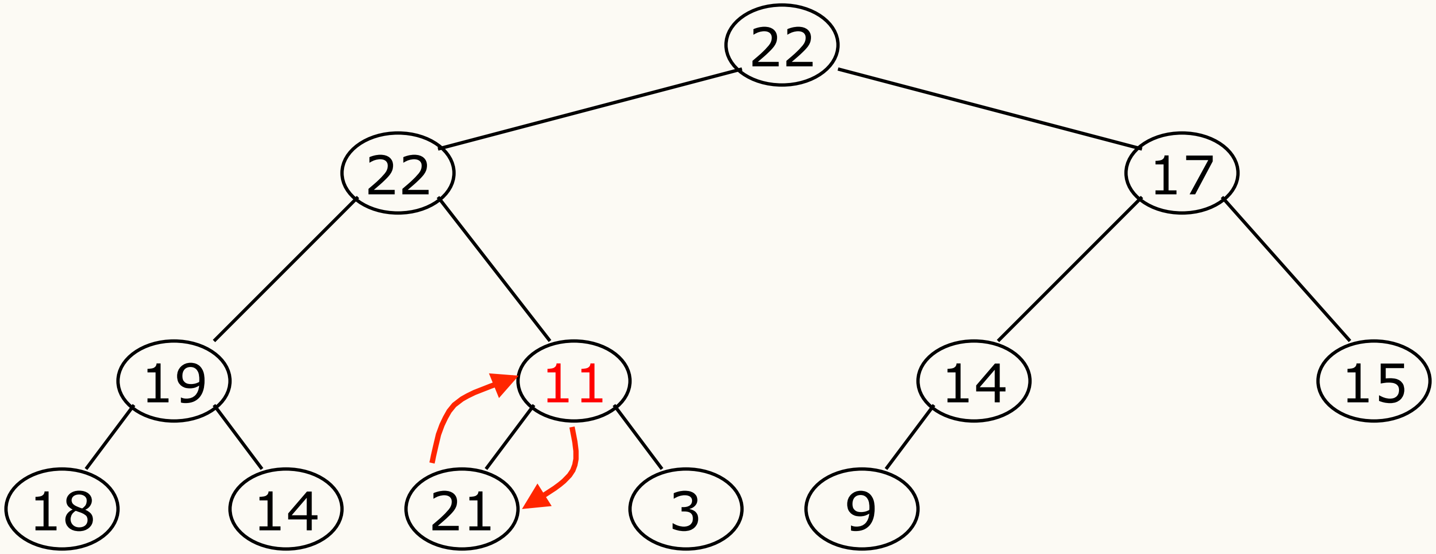
- Now the left child of the root (still the number **11**) lacks the heap property.



- We can percolate-down this node.
- After doing this, one and only one of its children may have lost the heap property.

The percolate-down method

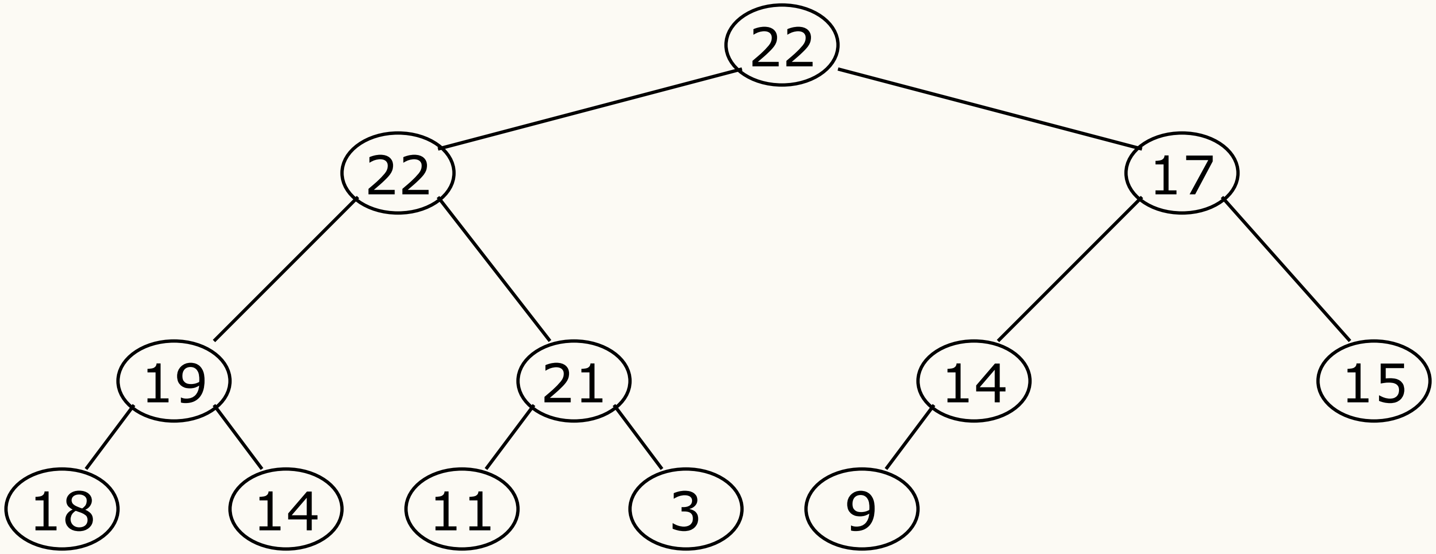
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can percolate-down this node.
- After doing this, one and only one of its children may have lost the heap property.

The percolate-down method

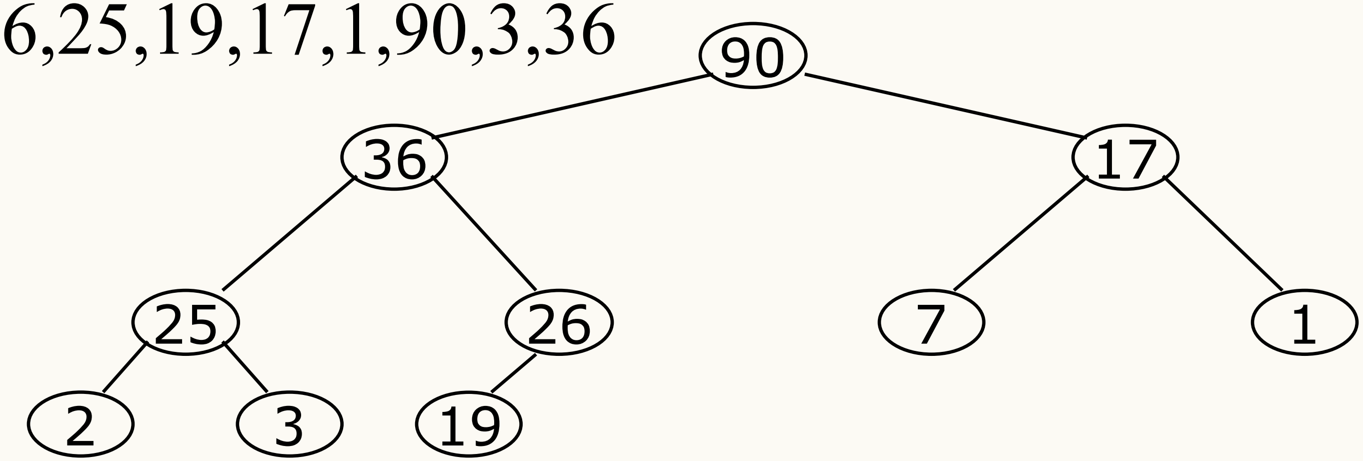
- Our tree is once again a heap, because every node in it has the heap property



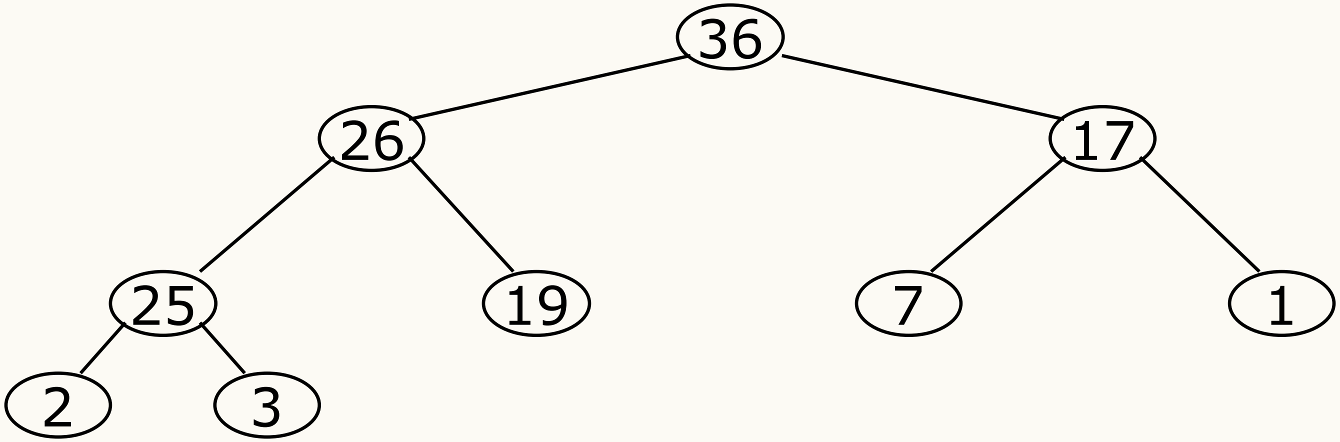
  – Once again, the largest (or *a* largest) value is in the root

In-class exercise

- Build a binary Max-heap using the following numbers
  - 2,7,26,25,19,17,1,90,3,36



  - Now remove max and reheap.



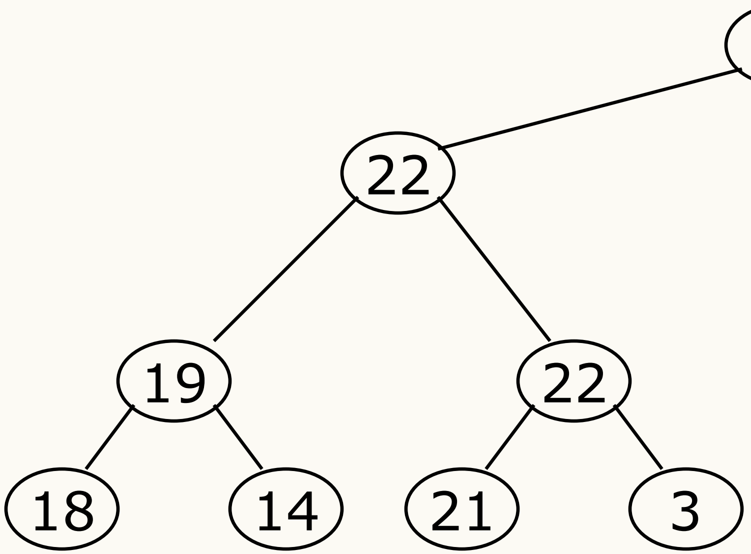
Removing an item from a heap

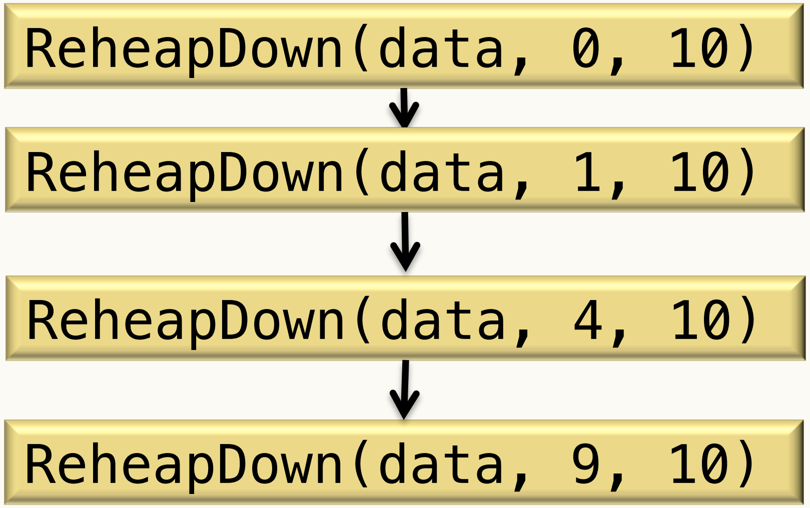
- Use ReheapDown (percolate-down) to remove an item from a max heap

```
/* This function performs the ReheapDown operation on an
array, to establish heap properties (for a subtree).

 PARAM:     data   – integer array containing the heap
            top    – position of the root
            bottom – position of the final elements in heap
*/
void ReheapDown( int * data, int top, int bottom){
    if (!isLeaf(top, bottom)){ /* top is not a leaf */
        int maxChild = getMaxChild(top) /* position of the
        child having largest data value */

        if ( data[top] < data[maxChild] ){
            swap( &data[top], &data[maxChild])
            ReheapDown( data, maxChild, bottom);
        }
    }
}
```

In-class exercise

- For the max heap below draw the recursion tree of `ReheapDown(data, 0, 10)`.





ReheapDown(data, 0, 10)

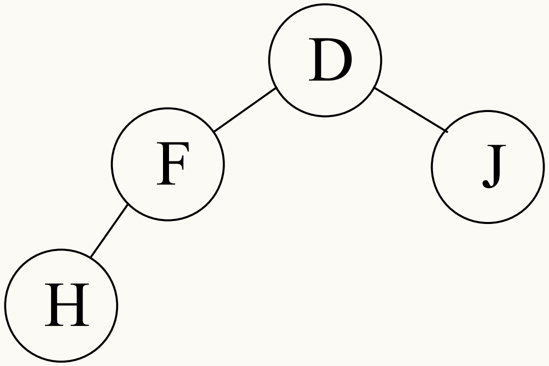
ReheapDown(data, 1, 10)

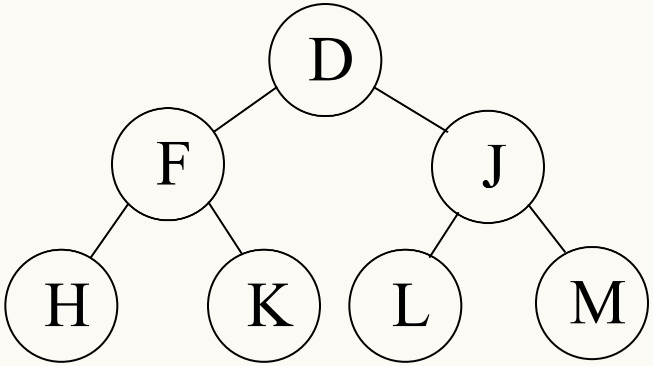
ReheapDown(data, 4, 10)

ReheapDown(data, 9, 10)

Time Complexity

When performing either a `ReheapUp` or `ReheapDown` operation, the number of operations depends on the depth of the tree. Notice that we traverse only one path or branch of the tree. Recall that a nearly complete binary tree of height h has between 2^h and $2^{h+1}-1$ nodes:





We can now determine the height of a heap in terms of the number of nodes n in the heap. The height is lg n.

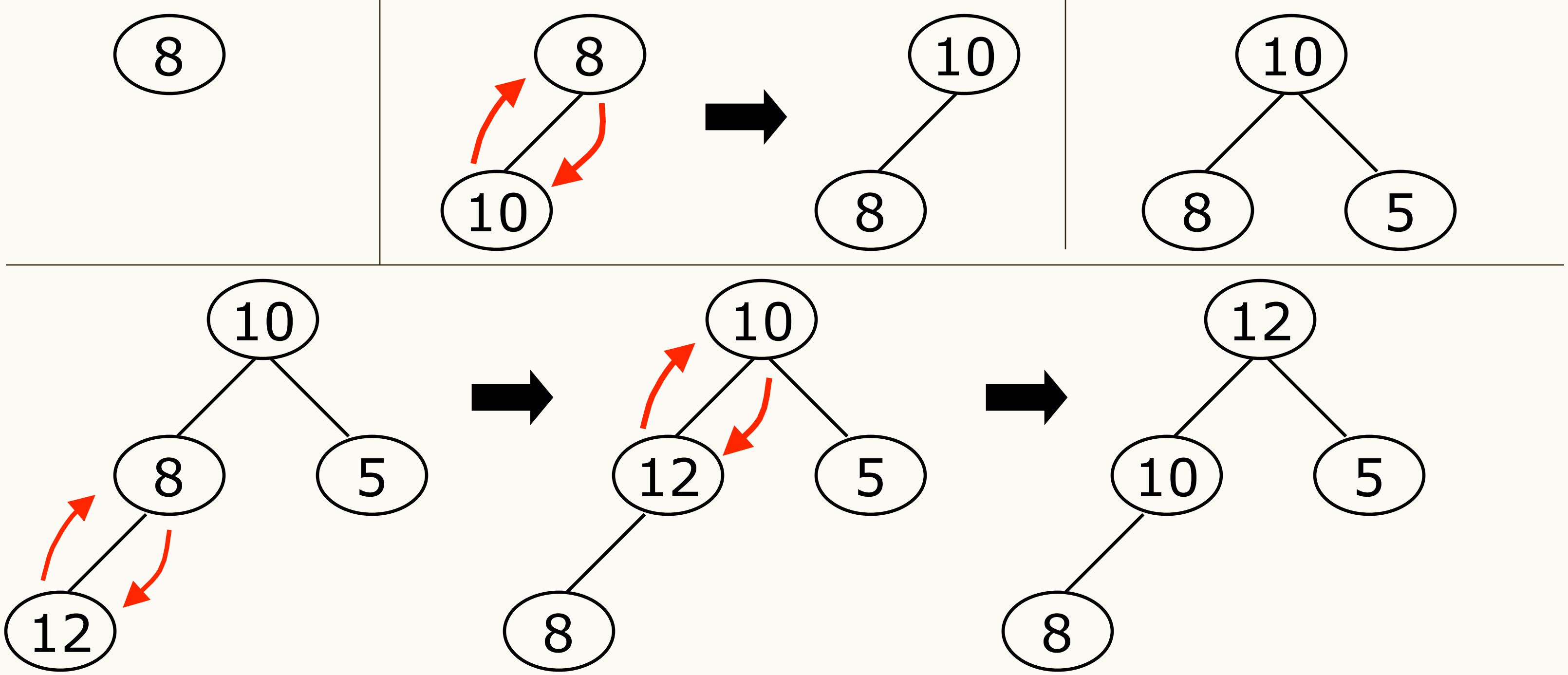
The time complexity of the `ReheapUp` and `ReheapDown` operations is therefore O(lg n).

Building a Heap (Naïve)

– Adding the elements one add a time with a reheapUp

- See http://visualgo.net/heap.html

```
/* This function builds a heap from an array.

 PARAM:     data   – integer array (no order is assumed)
            top    - position of the root
            bottom – position of the final elements in heap
  */
void Build_heap( int * data, int top, int bottom )
{
    int index = 0;
     while (index <= bottom){
        ReheapUp(data, top, index);
        index ++;
    }
}
```



- Complexity analysis:
  - we add each of n nodes and each node has to be sifted up, possibly as far as the root
  - Since the binary tree is a nearly complete binary tree, sifting up a single node takes *O(lg n)* time
  - Since we do this *N* times, `Build_heap` takes *N*O(lg n)* time, that is, *O(n lg n)* time

Heapify method

- See http://visualgo.net/heap.html

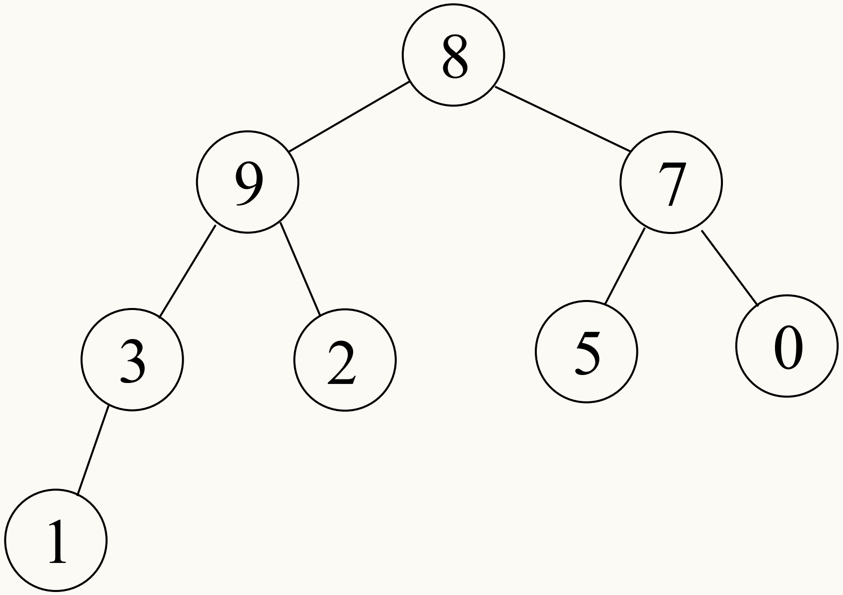
```
/* This function builds a heap from an array.

 PARAM:   data  - integer array (no order is assumed)
            top    - position of the root
            bottom – position of the final elements in heap


 */
void Heapify( int * data, int top, int bottom )
{
    int  index = position of last parent node in entire tree;
    while (index => top){
    /* go backwards from the last parent */
        ReheapDown( data, index, bottom );
        index --;
    }
}
```

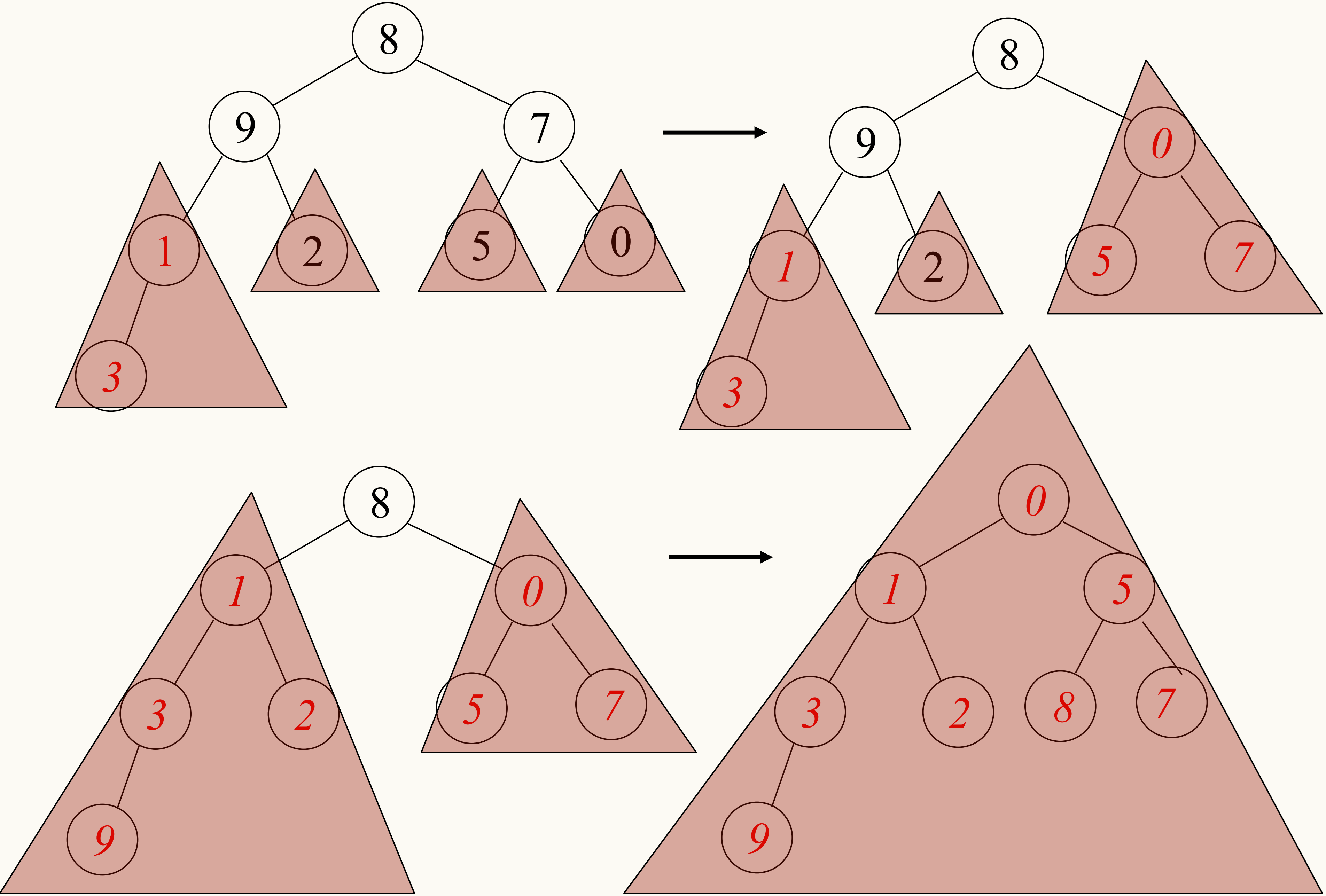
In-class exercise

Example: Convert the following array to a min-heap:

8	9	7	3	2	5	0	1

To do so, picture the array as a nearly complete binary tree:

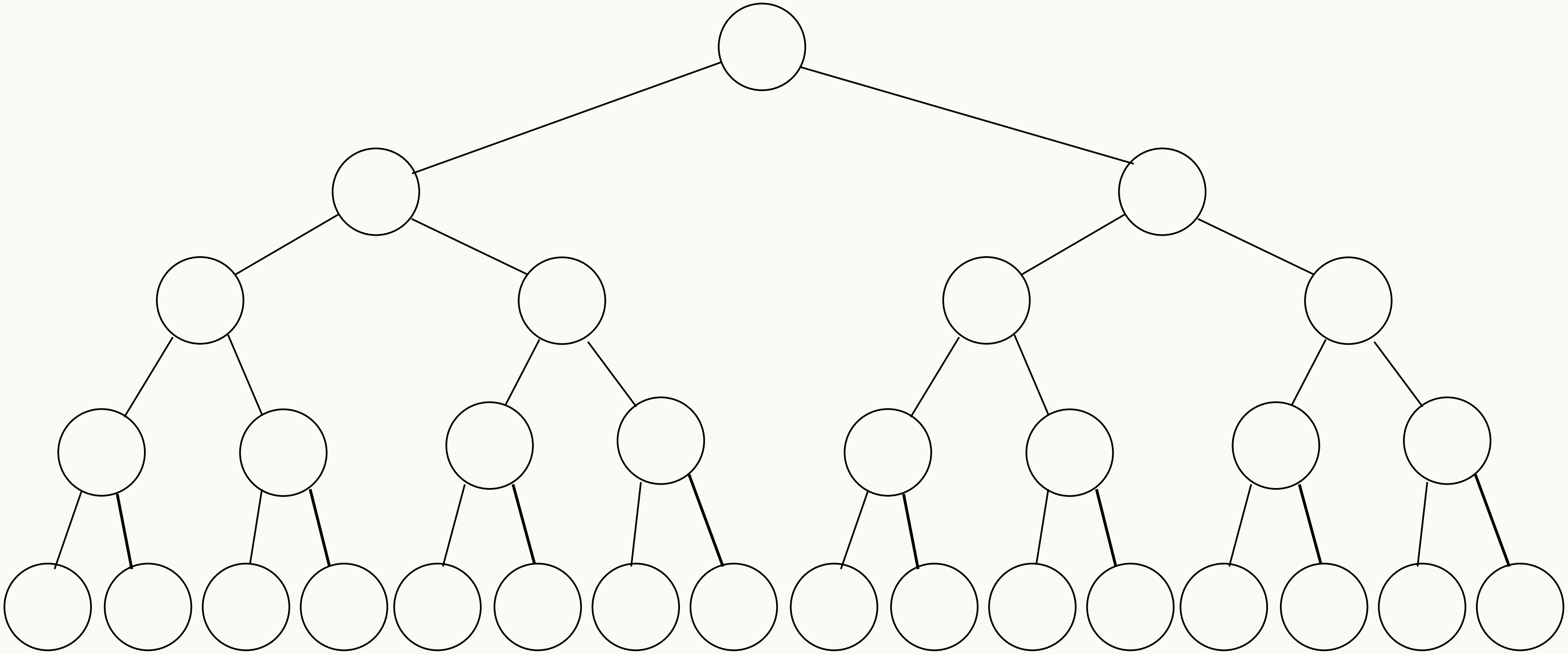



8
9
7
1
2
5
0
3
8
9
0
1
2
5
7
3
8
1
0
3
2
5
7
9
0
1
5
3
2
8
7
9

Time complexity of Heapify

- We can determine the time complexity of the `Heapify` function by looking at the total number of times the comparison and swap operations occur while building the heap. Let us consider the worst case, which is
  - when the last level in the heap is full, and all of the nodes with high priorities are in the leafs.
- We will colour all the paths from each node, starting with the lowest parent and working up to the root, each going down to a leaf node. The number of edges on the path from each node to a leaf node represents an upper bound on the number of comparison and swap operations that will occur while applying the `ReheapDown` operation to that node. By summing the total length of these paths, we will determine the time complexity of the `Heapify` function.

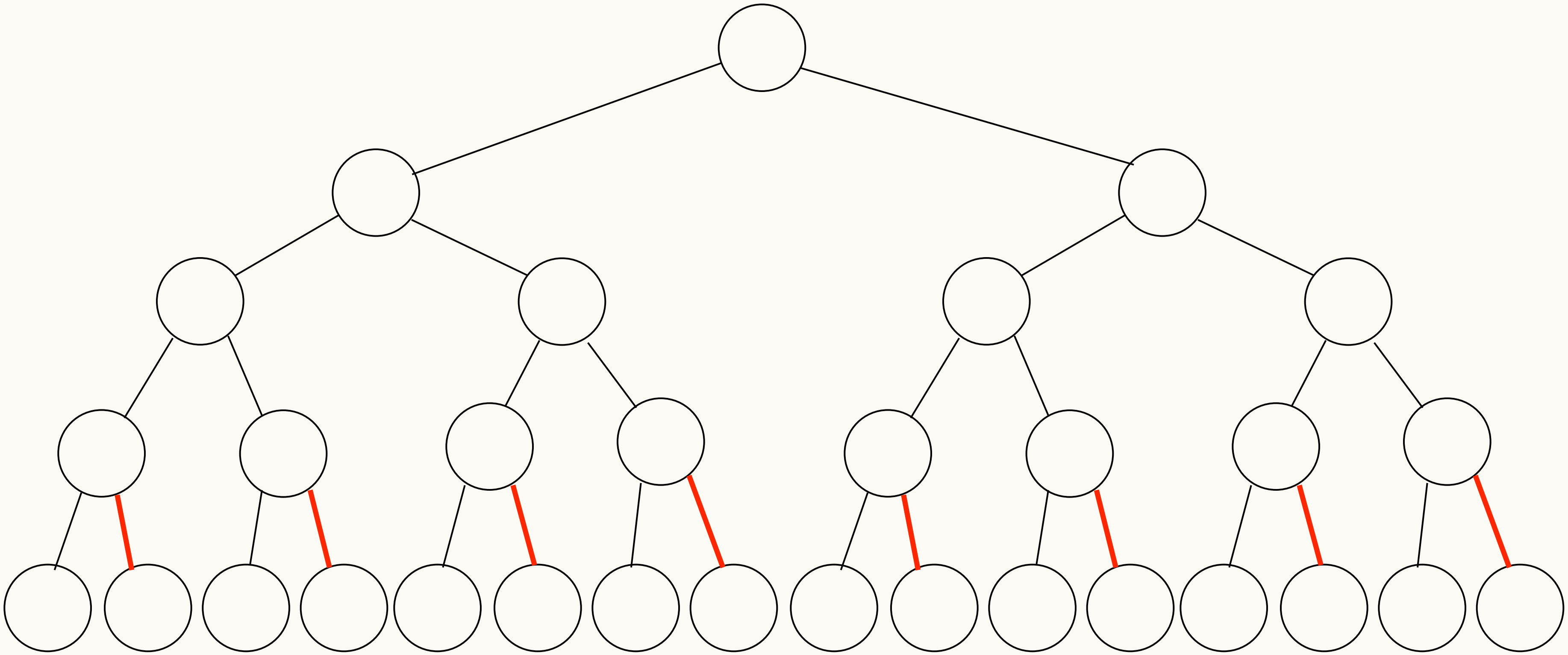
Time complexity of Heapify



In the worse possible case how many swaps are going to take place?

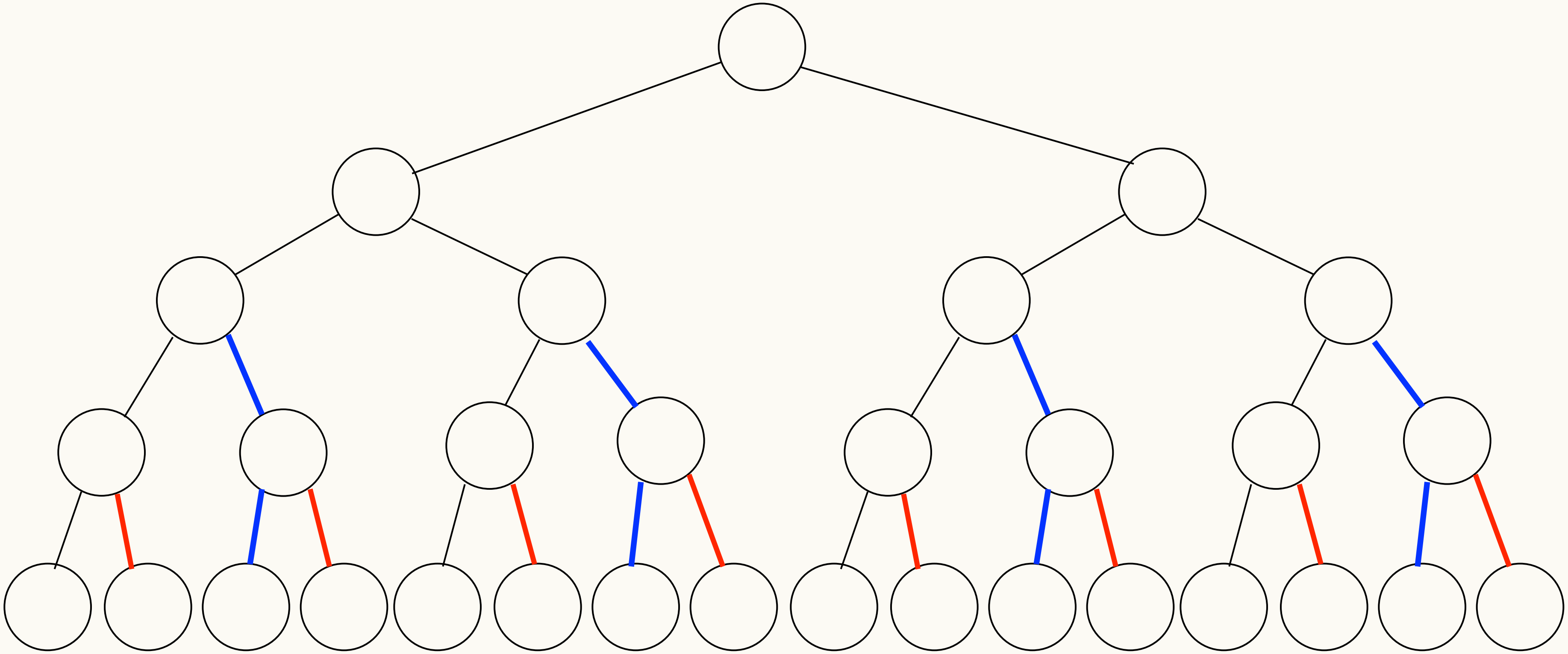
Relate the number of swaps first to the number of edges and then nodes.

Time complexity of Heapify



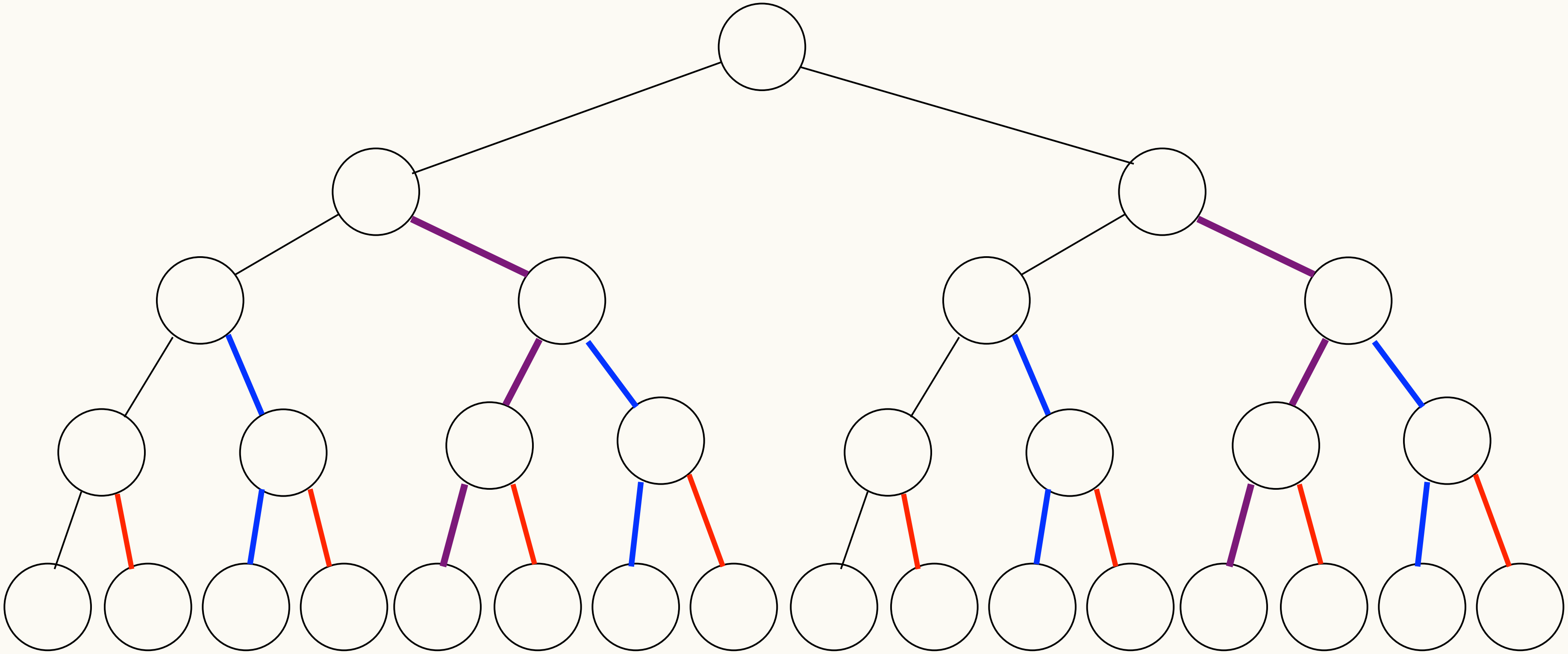
Note that no edge is coloured more than once. Hence the work done by the `Heapify` function to build the heap can be measured in terms of the number of coloured edges.

Time complexity of Heapify



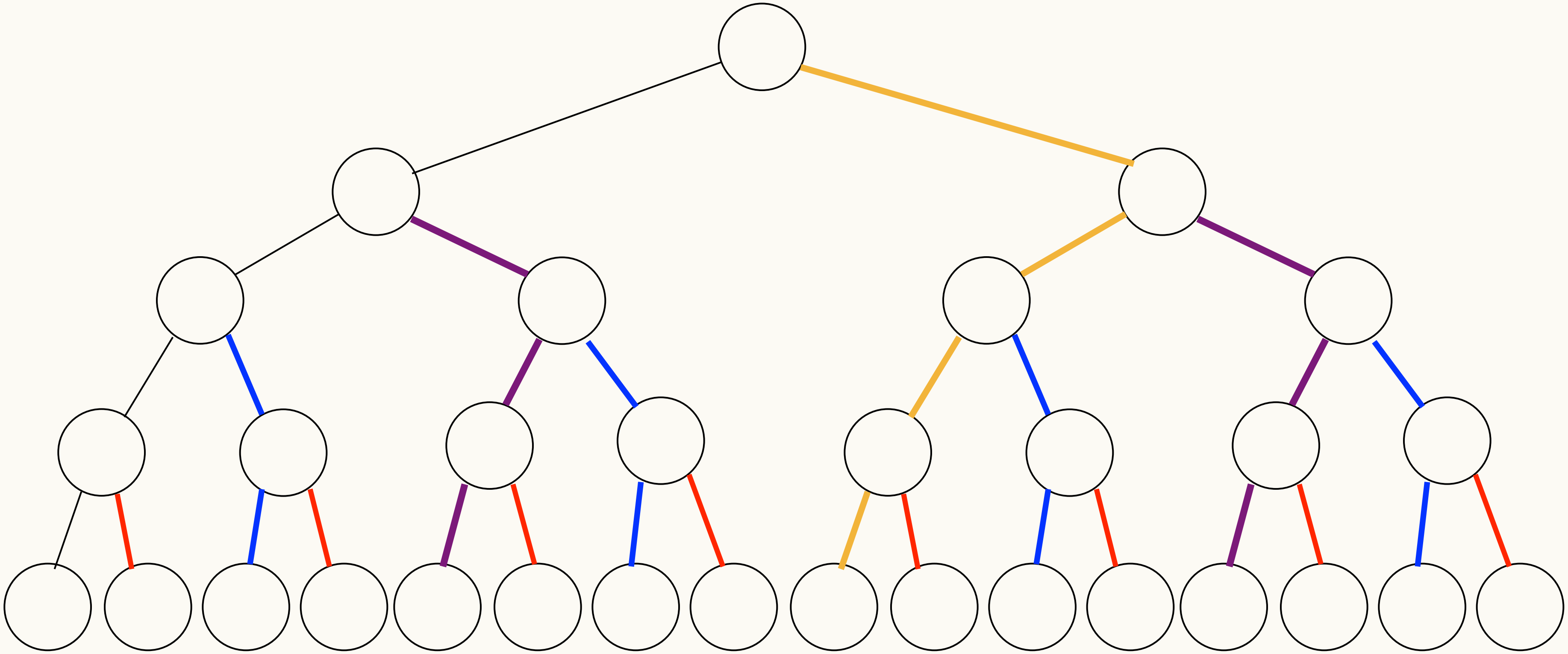
Note that no edge is coloured more than once. Hence the work done by the `Heapify` function to build the heap can be measured in terms of the number of coloured edges.

Time complexity of Heapify



Note that no edge is coloured more than once. Hence the work done by the `Heapify` function to build the heap can be measured in terms of the number of coloured edges.

Time complexity of Heapify



Note that no edge is coloured more than once. Hence the work done by the `Heapify` function to build the heap can be measured in terms of the number of coloured edges.

Time complexity of Heapify (sketch)

Suppose *H* is the height of the tree, *N* is the number of elements in the tree, and *E* is the number of edges in the tree.

- How many edges are there in a nearly complete tree with N elements?

  N-1

- Total number of coloured edges or swaps =

  $E – H = N - 1 - H = N – 1 – \lg N$

  $T(n) \in O(n)$

Hence, in the worst case, the overall time complexity of the `Heapify` algorithm is:

$O(n)$

The Heapsort Algorithm

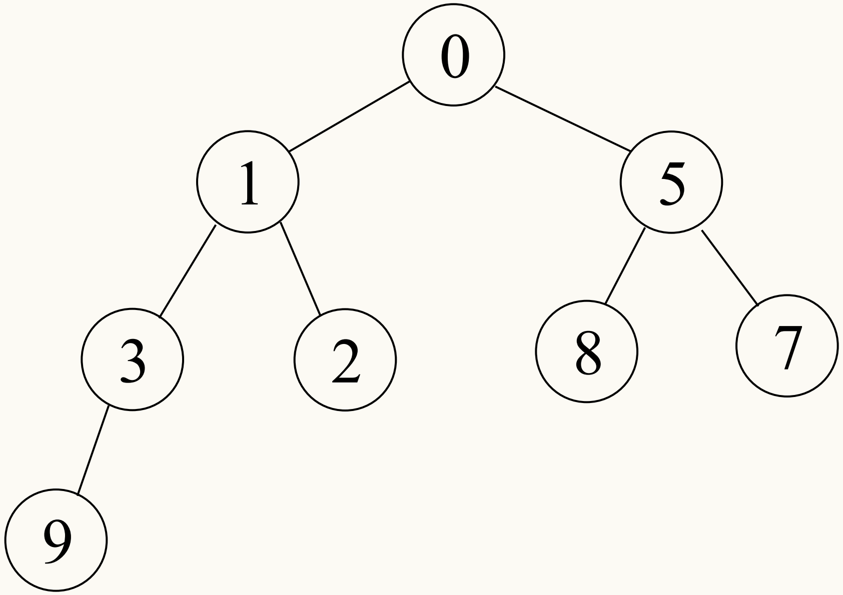
Motivation: In this section, we examine a sorting algorithm that guarantees worst case O(*n* lg *n*) time.

We will use a binary heap to sort an array of data

The **Heapsort** algorithm consists of 2 phases:

1. [Heapify] Build a heap using the elements to be sorted.
2. [Sort] Use the heap to sort the data.

Having built the heap, we now sort the array:



0	*1*	*5*	*3*	*2*	*8*	*7*	*9*

Note: In this section, we represent the data in both binary tree and array formats. It is important to understand that in practice the data is stored only as an array.

More about this later when we cover sorting!!!

Time Complexity of Heapsort

We need to determine the time complexity of the `Heapify` *O(n)* operation, and the time complexity of the subsequent sorting operation.

The time complexity of the sorting operation once the heap has been built is fairly easy to determine. For each element in the heap, we perform a single swap and a `ReheapDown`. If there are *N* elements in the heap, the `ReheapDown` operation is *O(lg n)*, and hence the sorting operation is *O(n lg n)*.

Hence, in the worst case, the overall time complexity of the `Heapsort` algorithm is:

$$\underbrace{O(n)}_{\substack{\text{build heap from}\\ \text{unsorted array}}} + \underbrace{O(n \lg n)}_{\substack{\text{essentially}\\ \text{perform } N\\ \texttt{RemoveMin's}}} = O(n \lg n)$$

Learning Goals revisited

- Provide examples of appropriate applications for priority queues and heaps.
- Implement and manipulate a heap using an array as the underlying data structure.
- Describe and use the Heapify (“Build Heap”) and Heapsort algorithms.
- Analyze the complexity of operations on a heap.