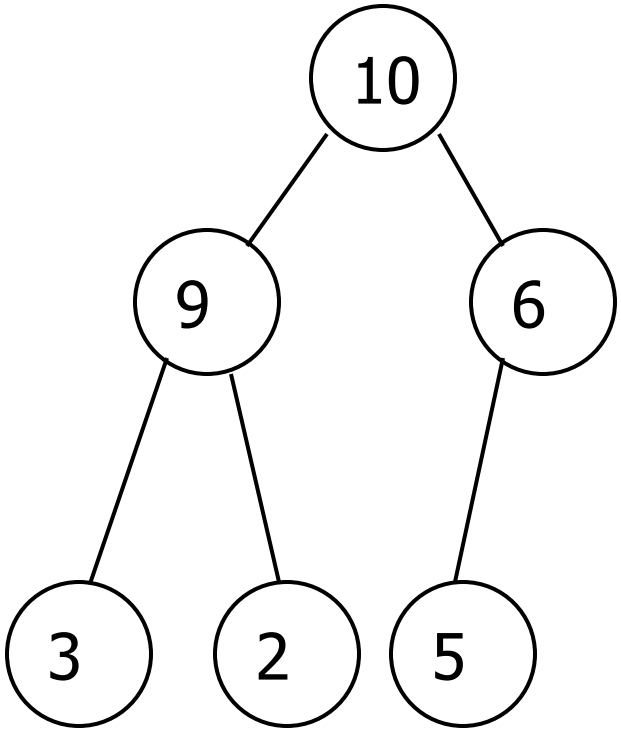
# EECS 114:
# Engineering Data Structures and Algorithms
## Lecture 6

Instructor: Ryan Rusich
E-mail: rusichr@uci.edu
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

# (Binary) Heaps

- Similar to a Binary Search Tree (BST)
- Heap is sorted in a weaker sense than BST
- Tree is a complete tree
  - Bottom level may not be filled but fills from left to right

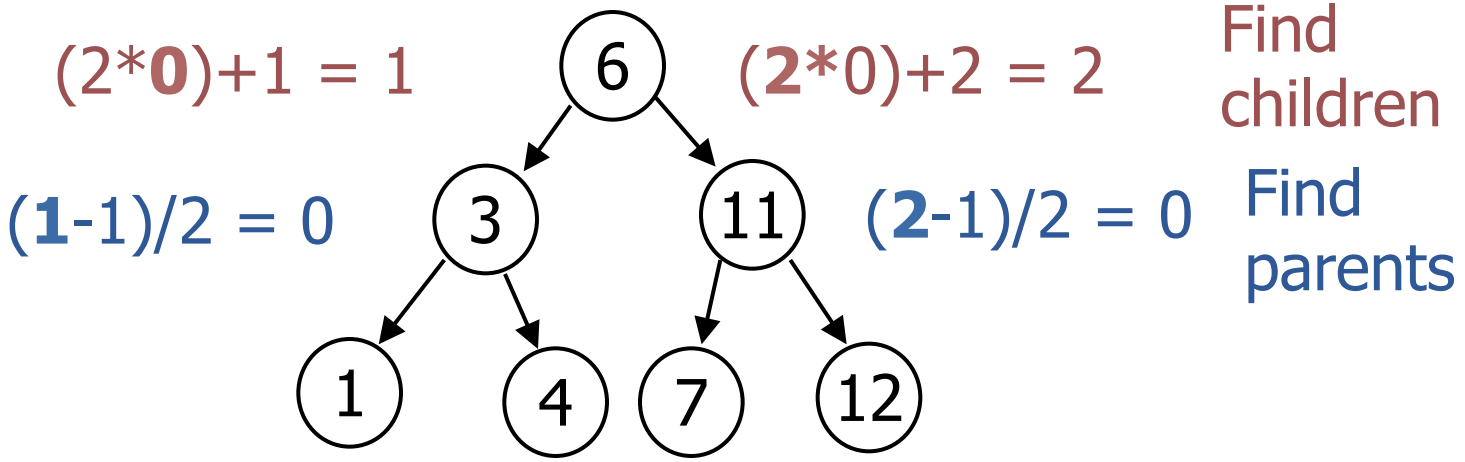


# (Binary) Heaps

- Binary trees are easily stored in an array
  - Array with index started at 0
    - Parent = (i-1)/2
    - Left Child = 2i + 1
    - Right Child = 2i + 2
  - Array with index started at 1 (from CLRS)
    - Parent = i/2
    - Left Child = 2i
    - Right Child = 2i + 1

# Binary Tree – Array

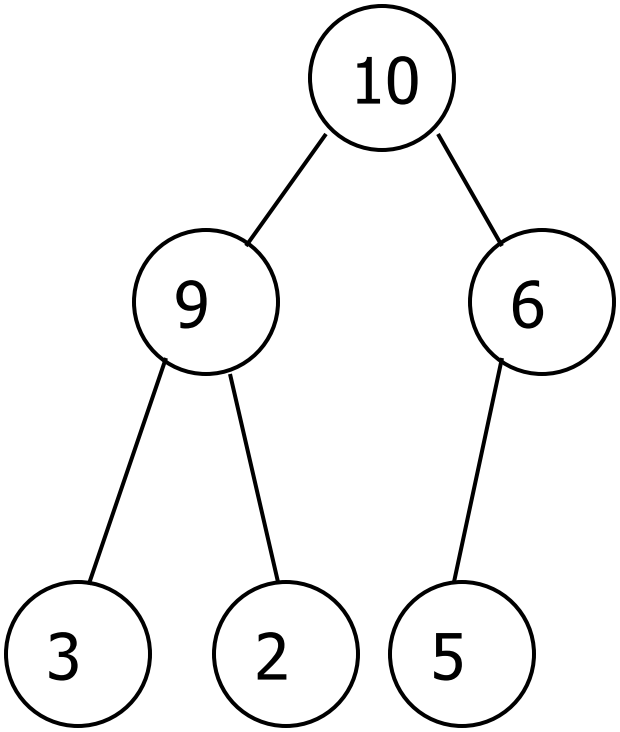
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|---|---|---|---|---|---|---|---|
| 6 | 3 | 11 | 1 | 4 | 7 | 12 | |



# Heap Order Properties

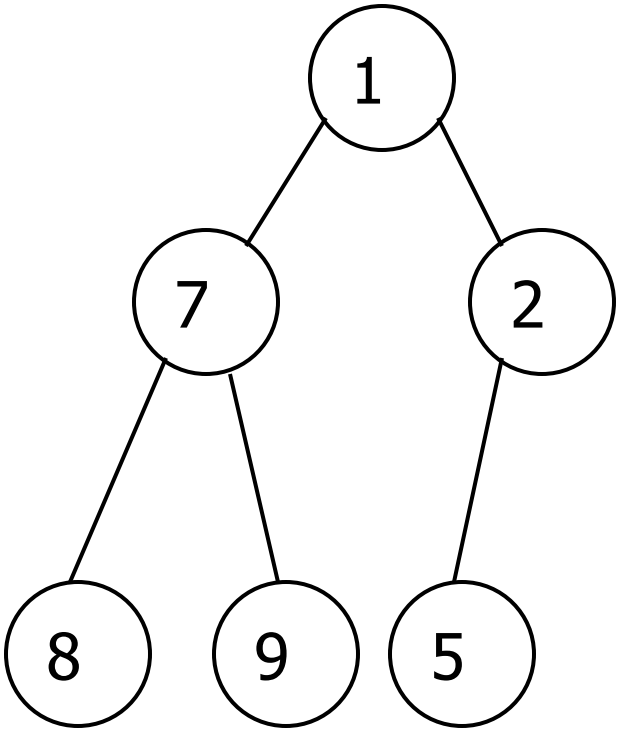
- Min Heap
  - For each node X, X.key > (X.parent).key
    - Smallest number is at the root
- Max Heap
  - For each node X, X.key < (X.parent).key
    - Largest number is at the root

# Max Heap




| |
|---|
| 10 |
| 9 |
| 6 |
| 3 |
| 2 |
| 5 |
| |
| |

# Min Heap




| |
|---|
| 1 |
| 7 |
| 2 |
| 8 |
| 9 |
| 5 |
| |
| |

# Heap - Insert

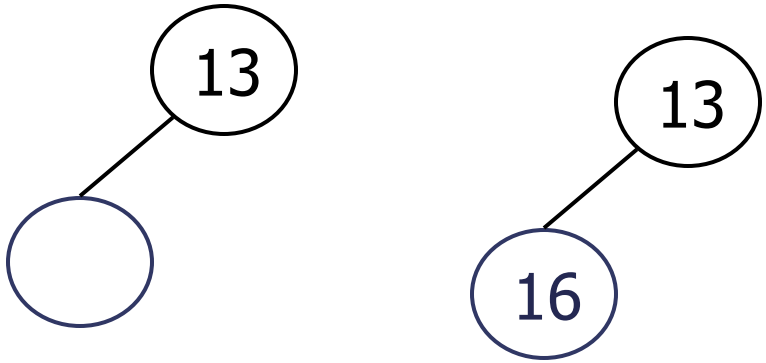
- Create a hole (empty node) in the next available complete tree location
  - Remember to fill bottom layer from left to right
- If the item can be inserted into the hole without violation of the heap property, insert item
- Otherwise, copy the hole's parent item into the hole then trickle up

# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

Insert 13

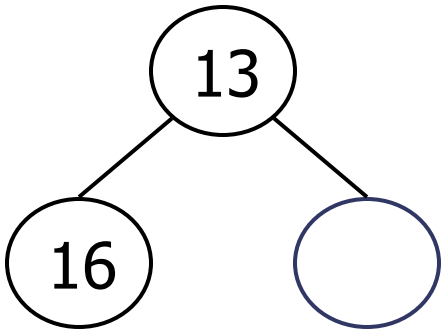

Insert 16

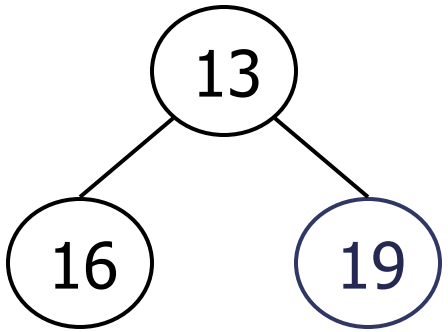


# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

Insert 19

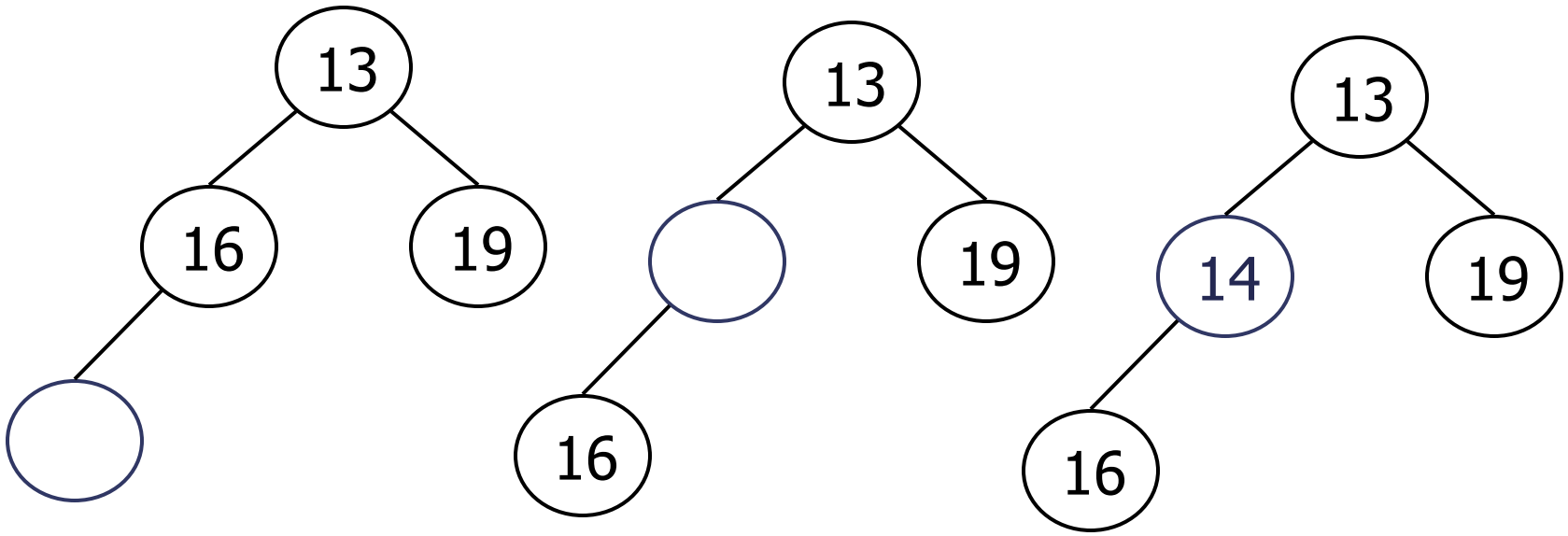


# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

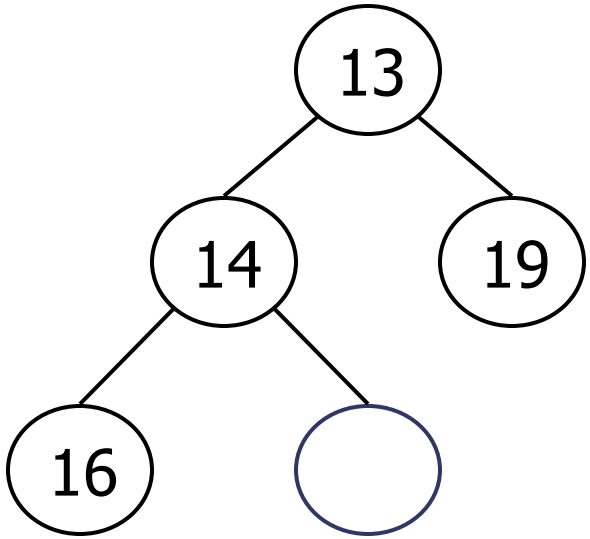
Insert 14

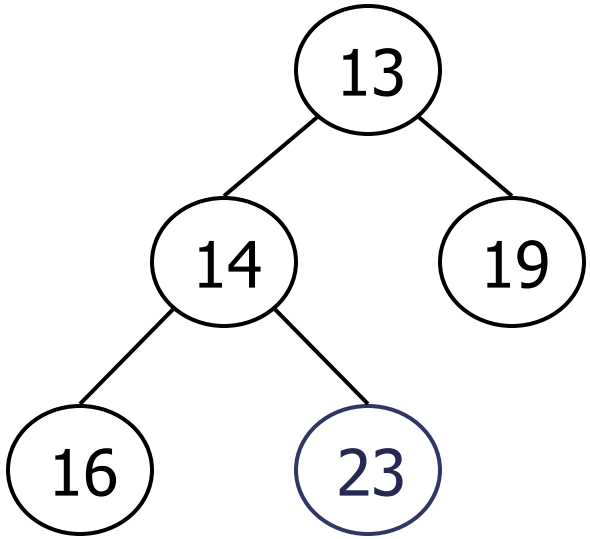


# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

Insert 23

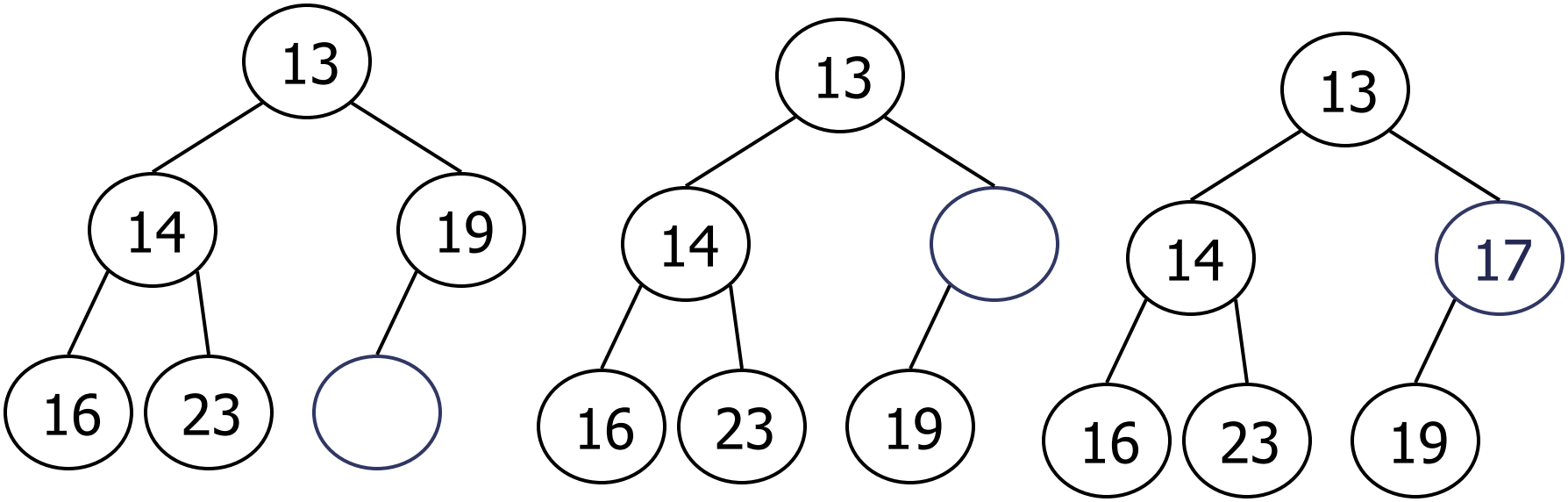


# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

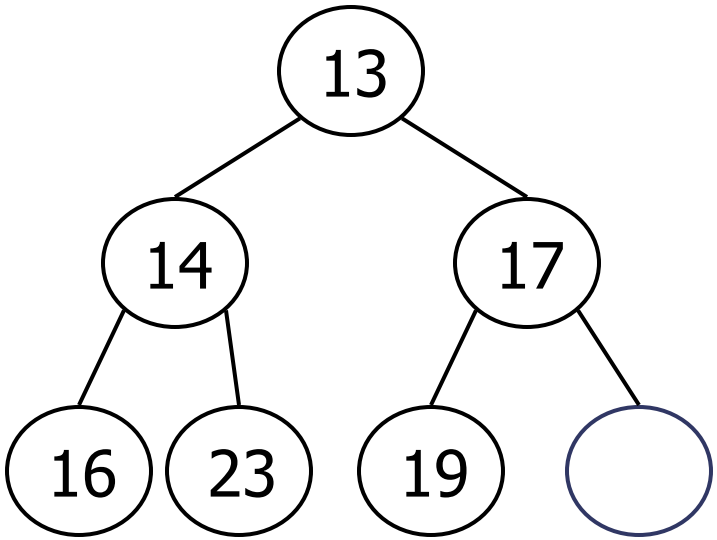
Insert 17

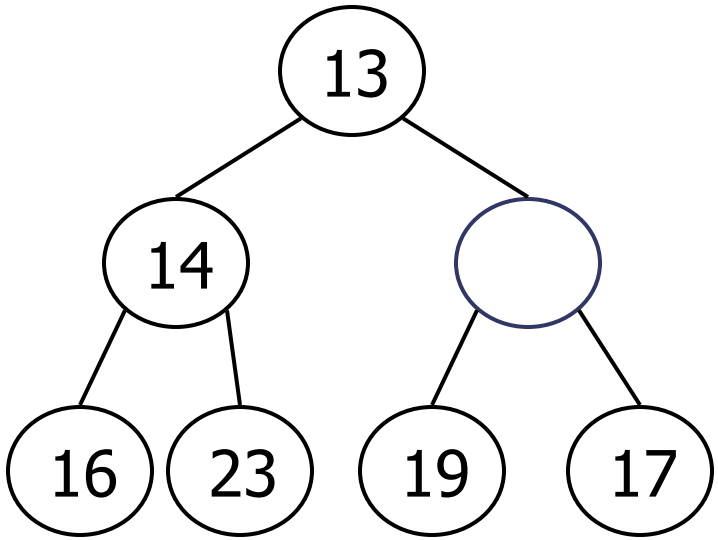


# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

Insert 6

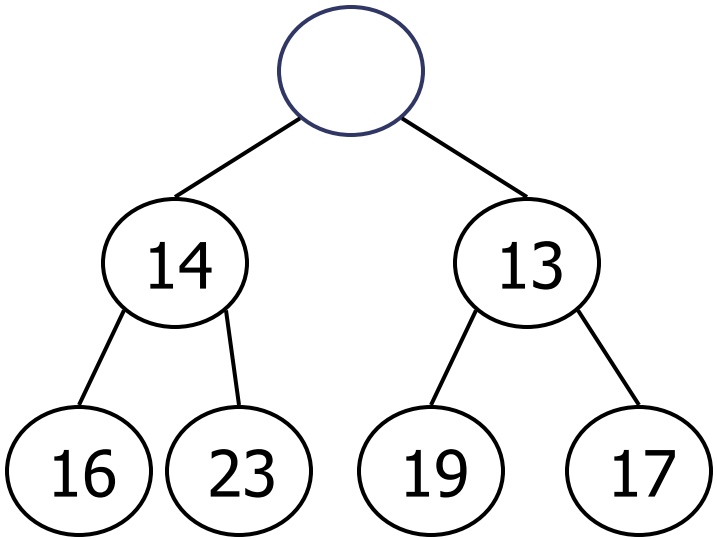


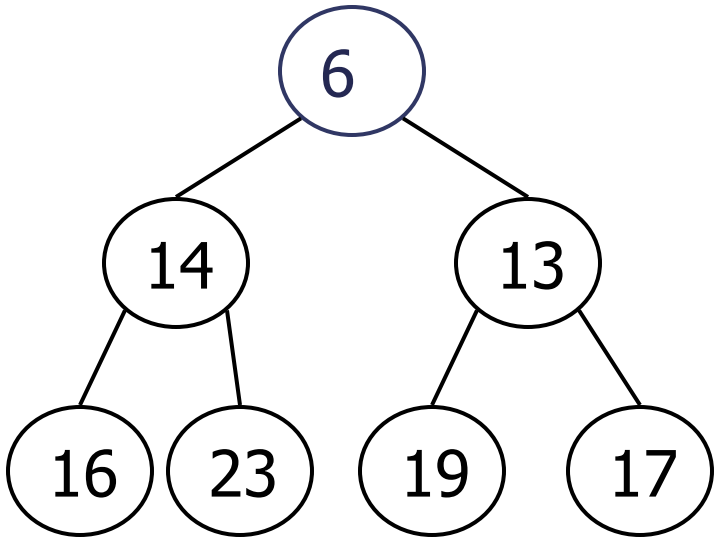



# Min Heap - Insert

13, 16, 19, 14, 23, 17, 6

Insert 6

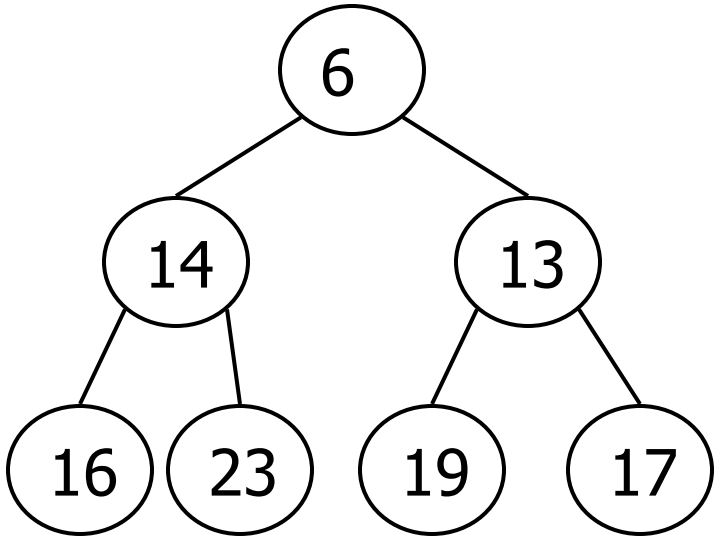


# Min Heap – Extract Min

6, 14, 13, 16, 23, 19, 17

6,14,13,16 23,19,17
13,14,17,16,23,19
14,16,17,19,23
16,19,17,23
17,19,23
19,23
23

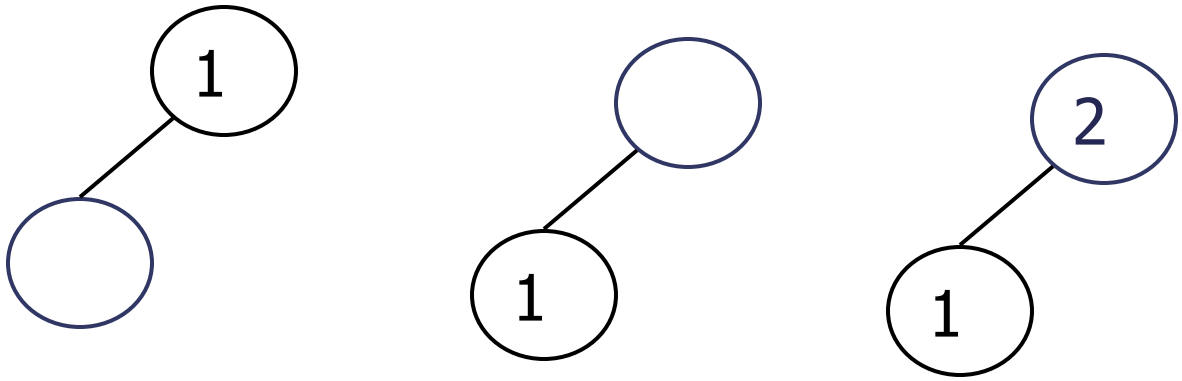


# Max Heap - Insert

1, 2, 3, 4, 5, 6

Insert 1

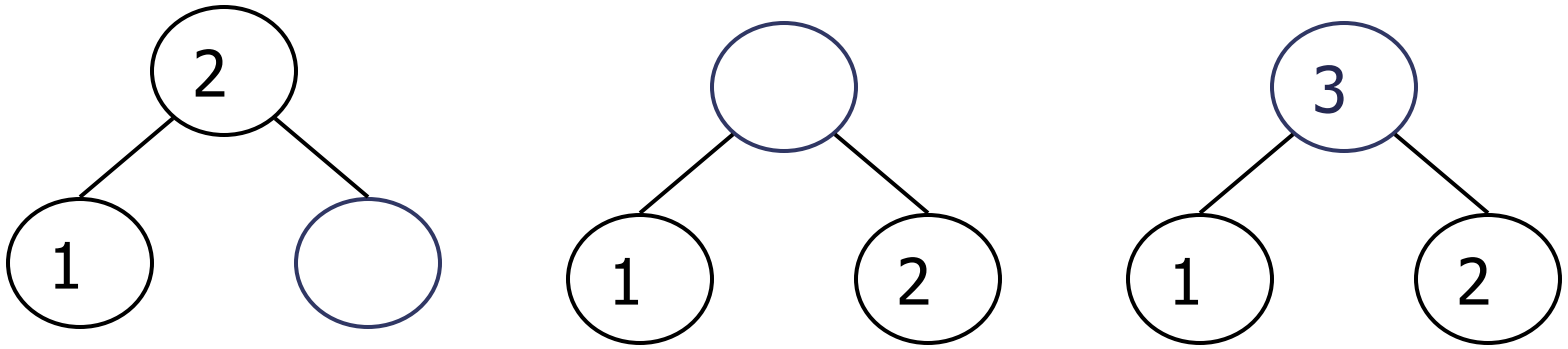

Insert 2



# Max Heap - Insert

1, 2, 3, 4, 5, 6

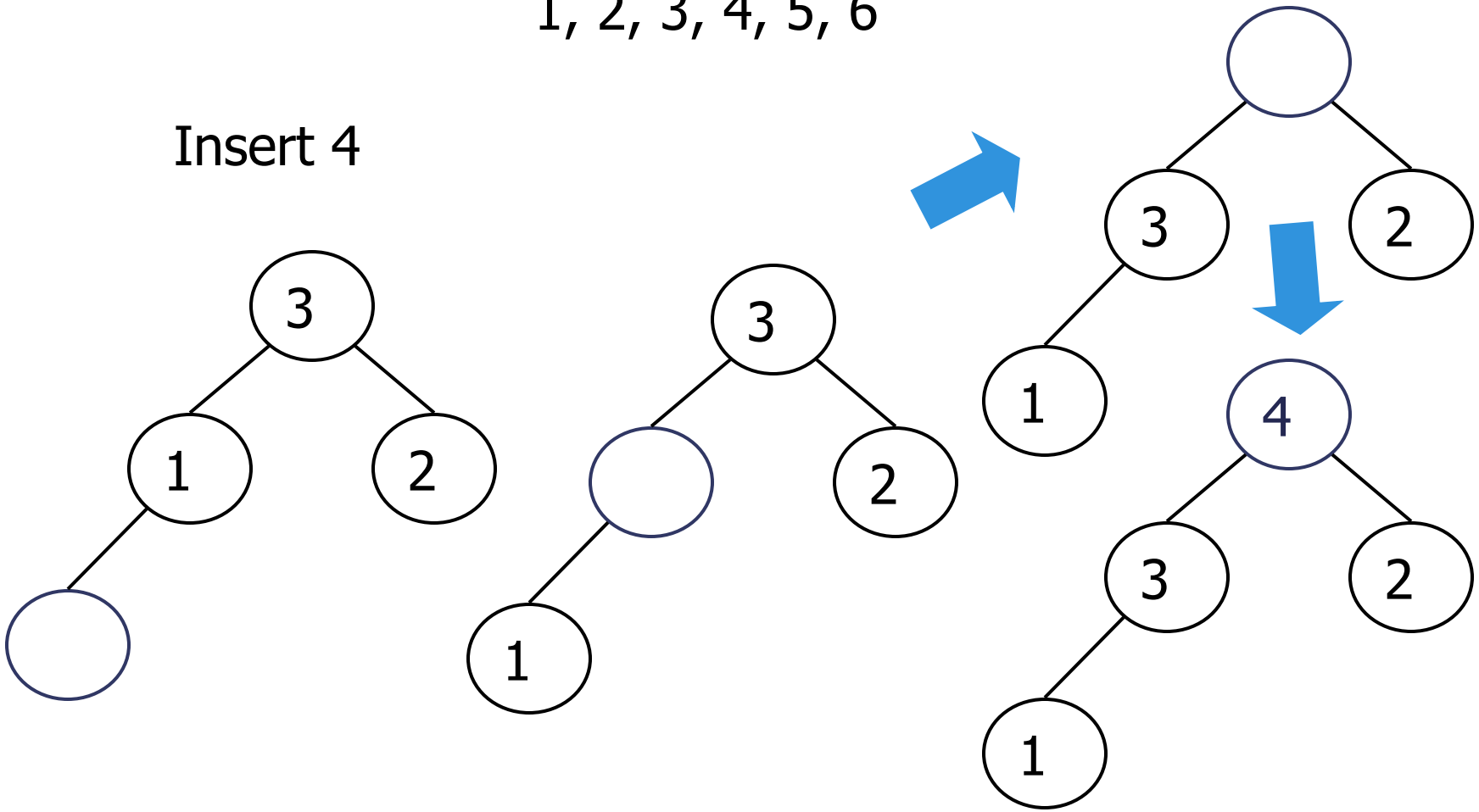
Insert 3



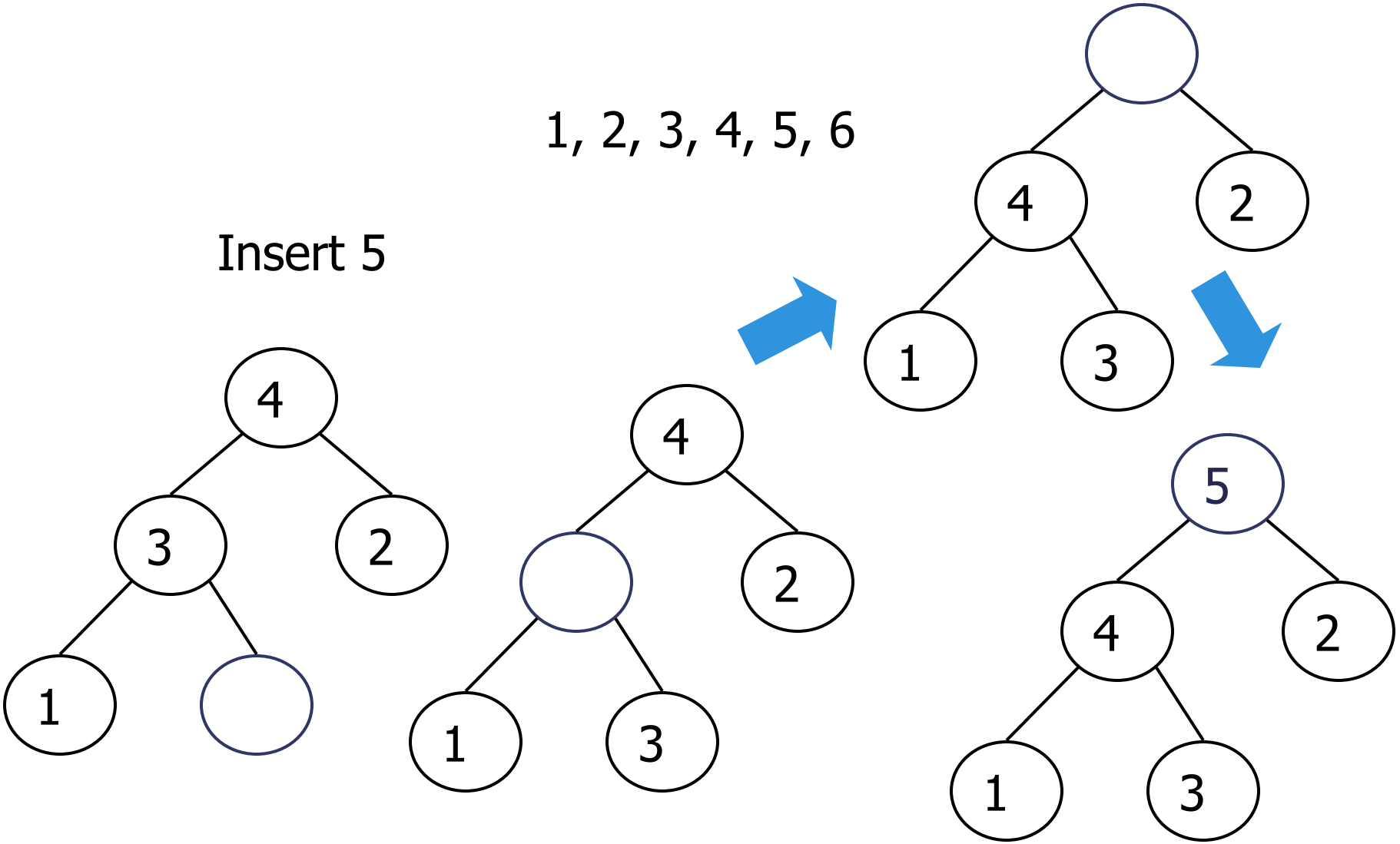
# Max Heap - Insert

1, 2, 3, 4, 5, 6

Insert 4



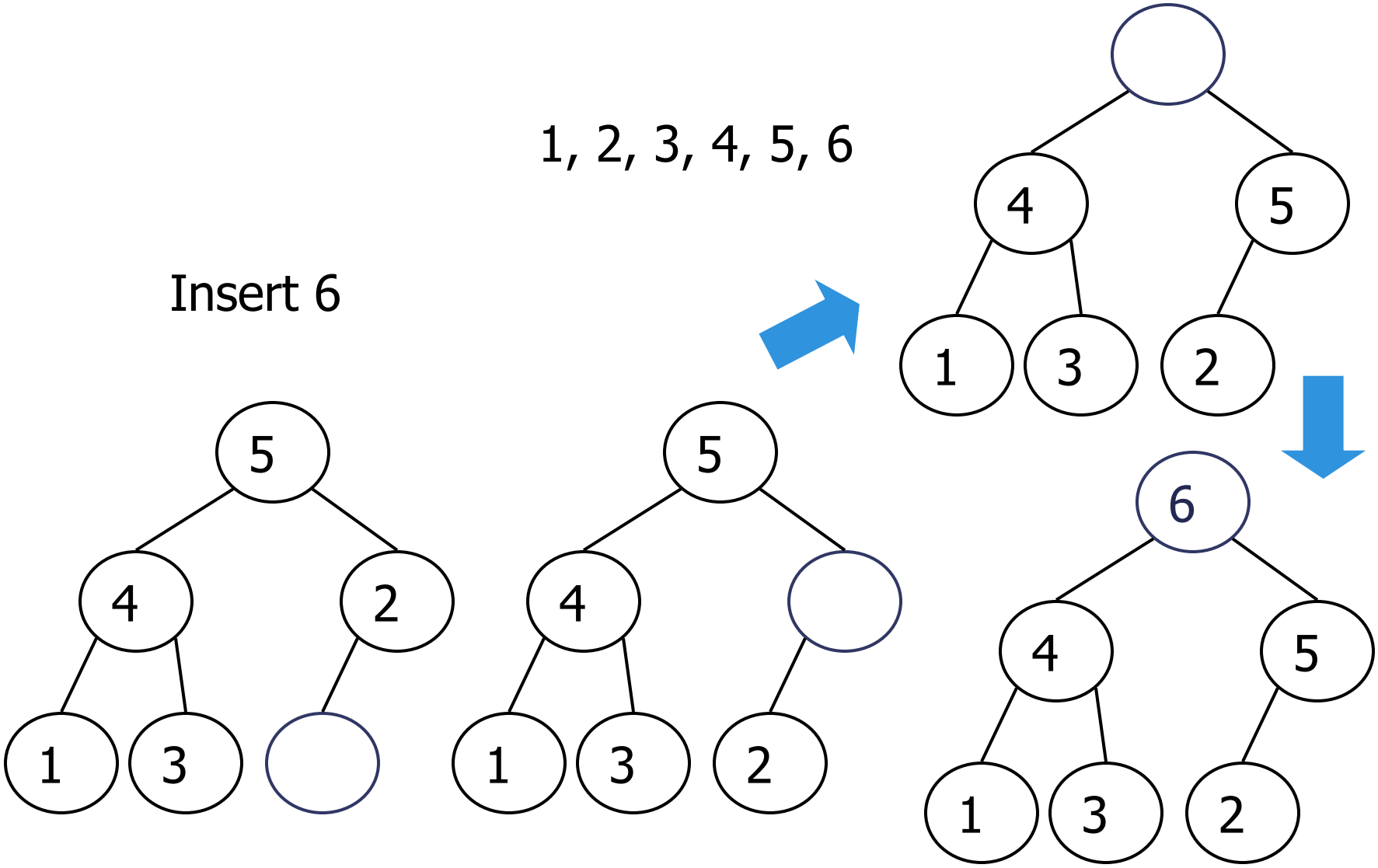
# Max Heap - Insert



# Max Heap - Insert

1, 2, 3, 4, 5, 6

Insert 6



# Min Heap Insert

Running time = O(logn)

```
void Min-Heap-Insert ( itemtype item )
        if  heap full
              throw heap full exception
        // Trickle up
        x = numNodes
        while (x > 0 && arr[(x-1)/2] > item)
            arr[x] = arr[(x-1)/2]
            x = (x-1)/2
        arr[x] = item
        numNodes++
```

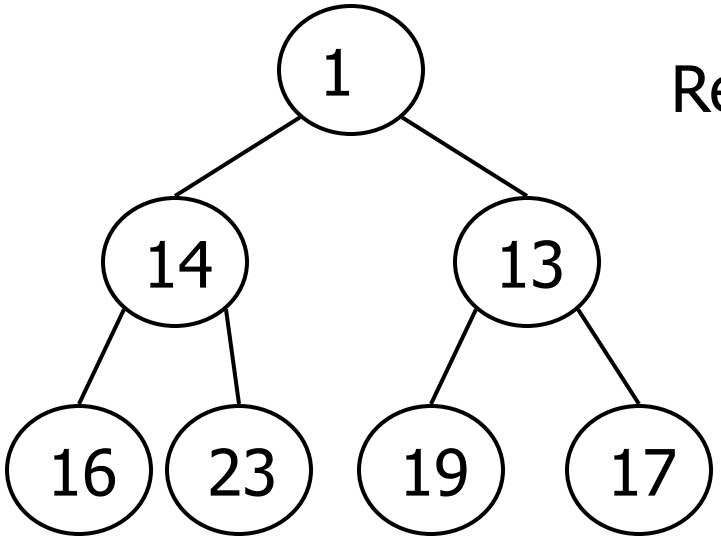
# Heap Find Min/Max

- Finding the node with the highest priority is easy - it is just at the root
  - Running time = O(1)

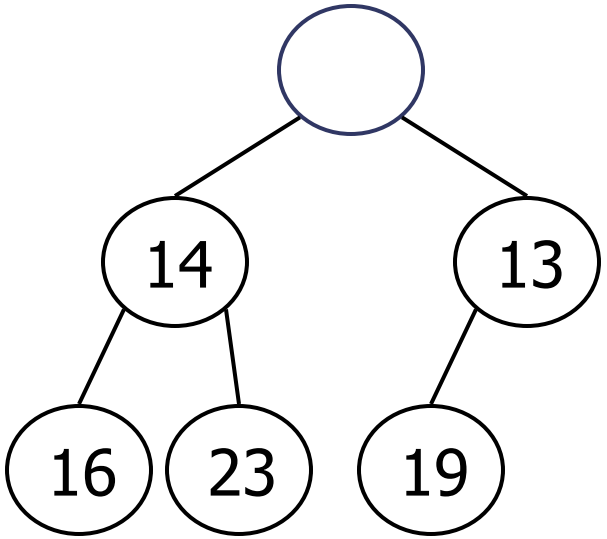
# Heap – Remove (Extract Min or Max)

- Remove highest priority item
  - Makes a hole at the root
  - Want to remain a complete tree, so attempt to place last item in the heap into the hole
    - If item can be placed in hole without violation of the heap property, then done
    - Otherwise, trickle down
    - Pick the child with the highest priority

# Min Heap – Remove

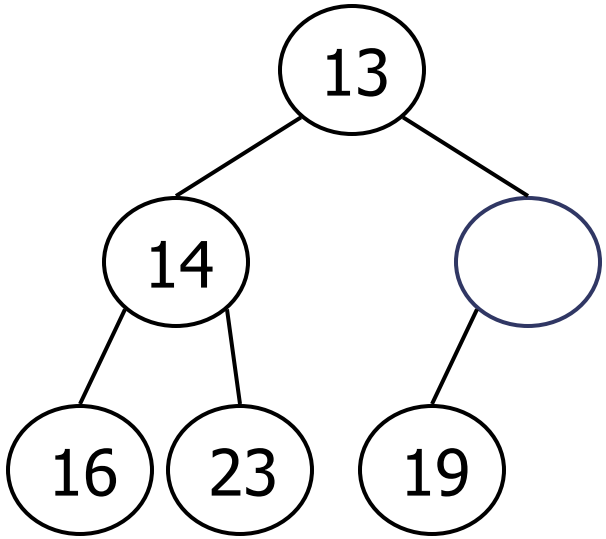



Remove 1

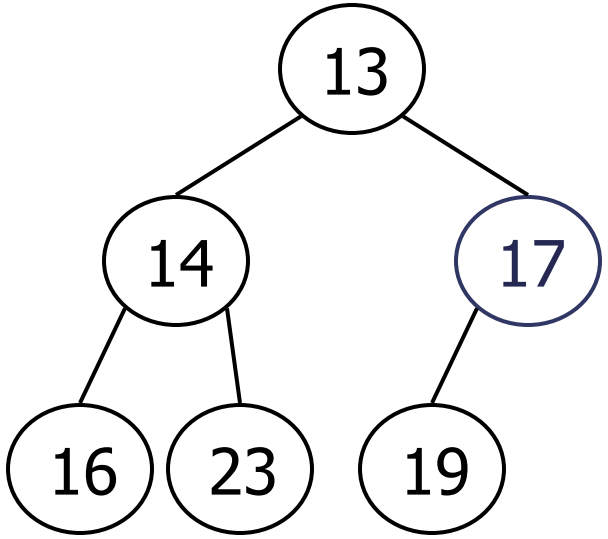



17

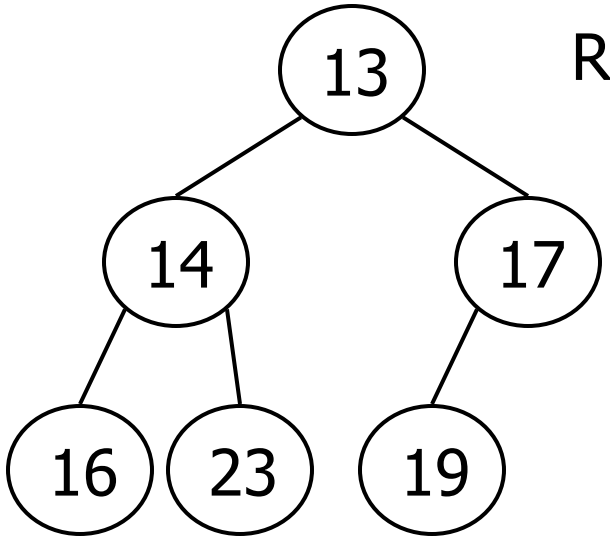
# Min Heap - Remove



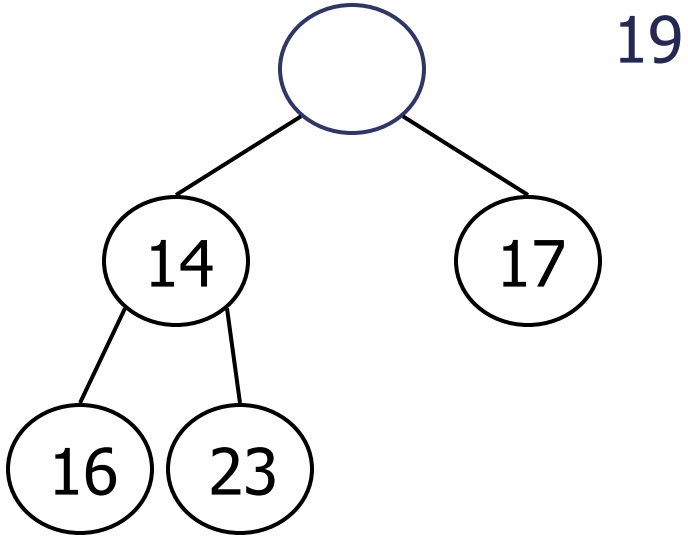

17



# Min Heap - Remove

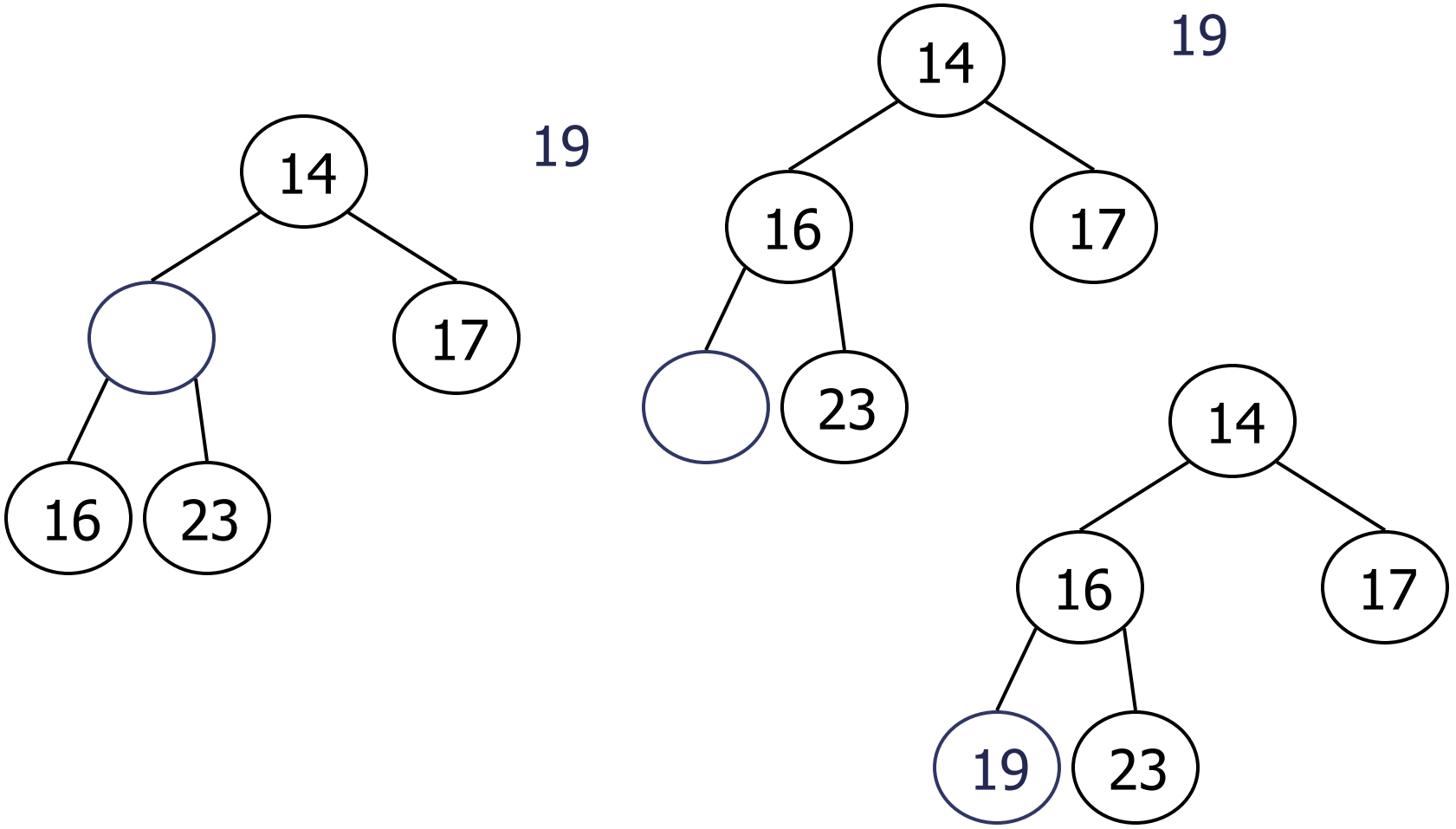



Remove 13

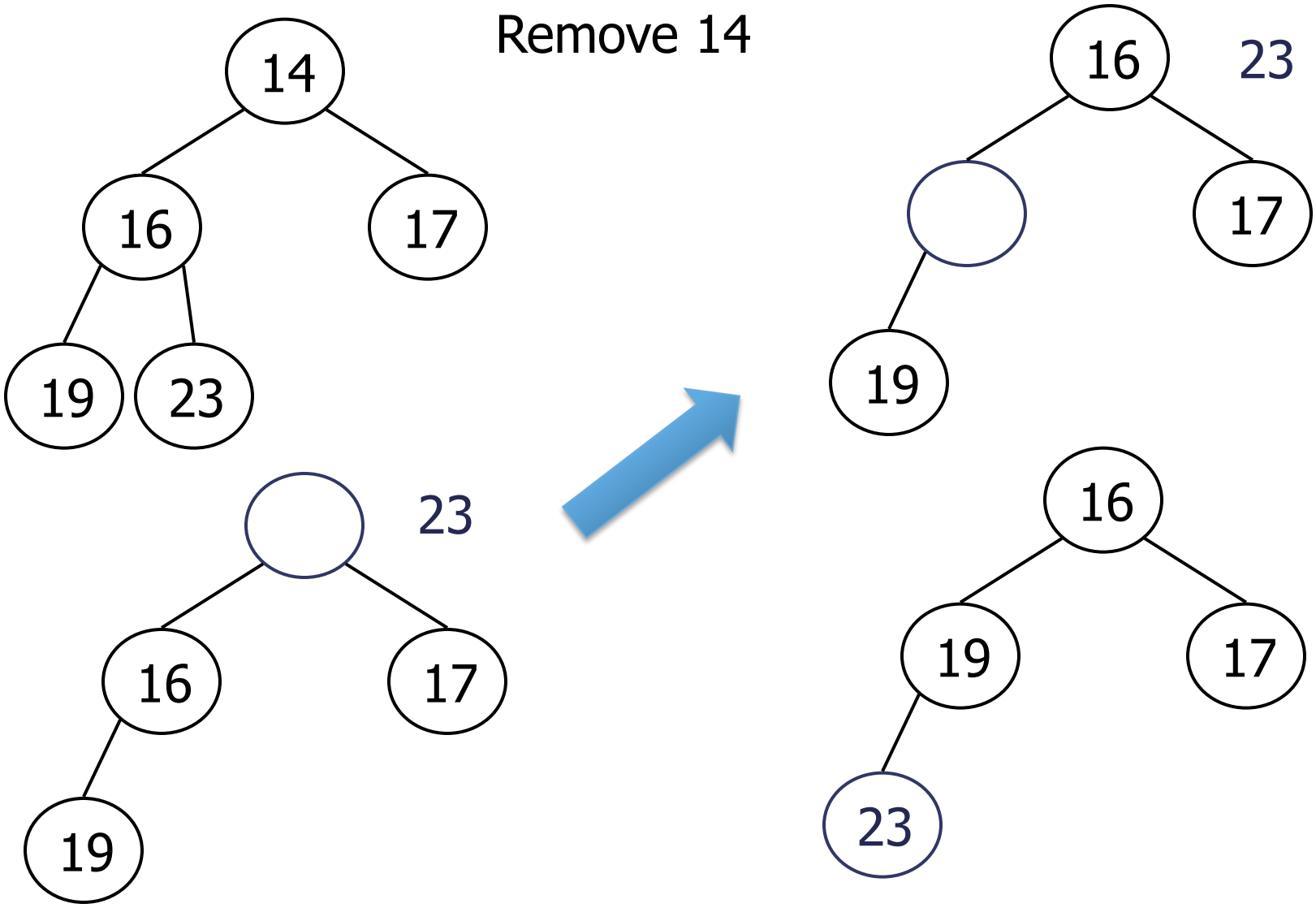



19

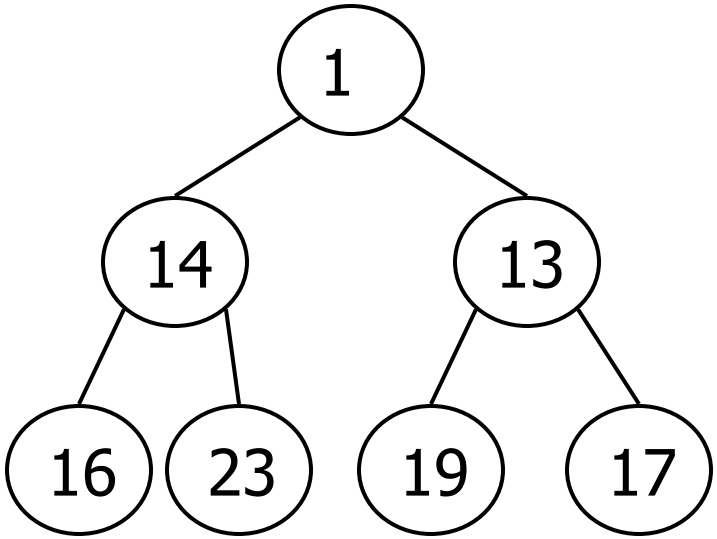
# Min Heap - Remove



# Min Heap - Remove



# Min Heap - Remove




1,14,13,16,23,19,17
13,14,17,16,23,19
14,16,17,19,23
16,19,17,23
17,19,23
19,23
23

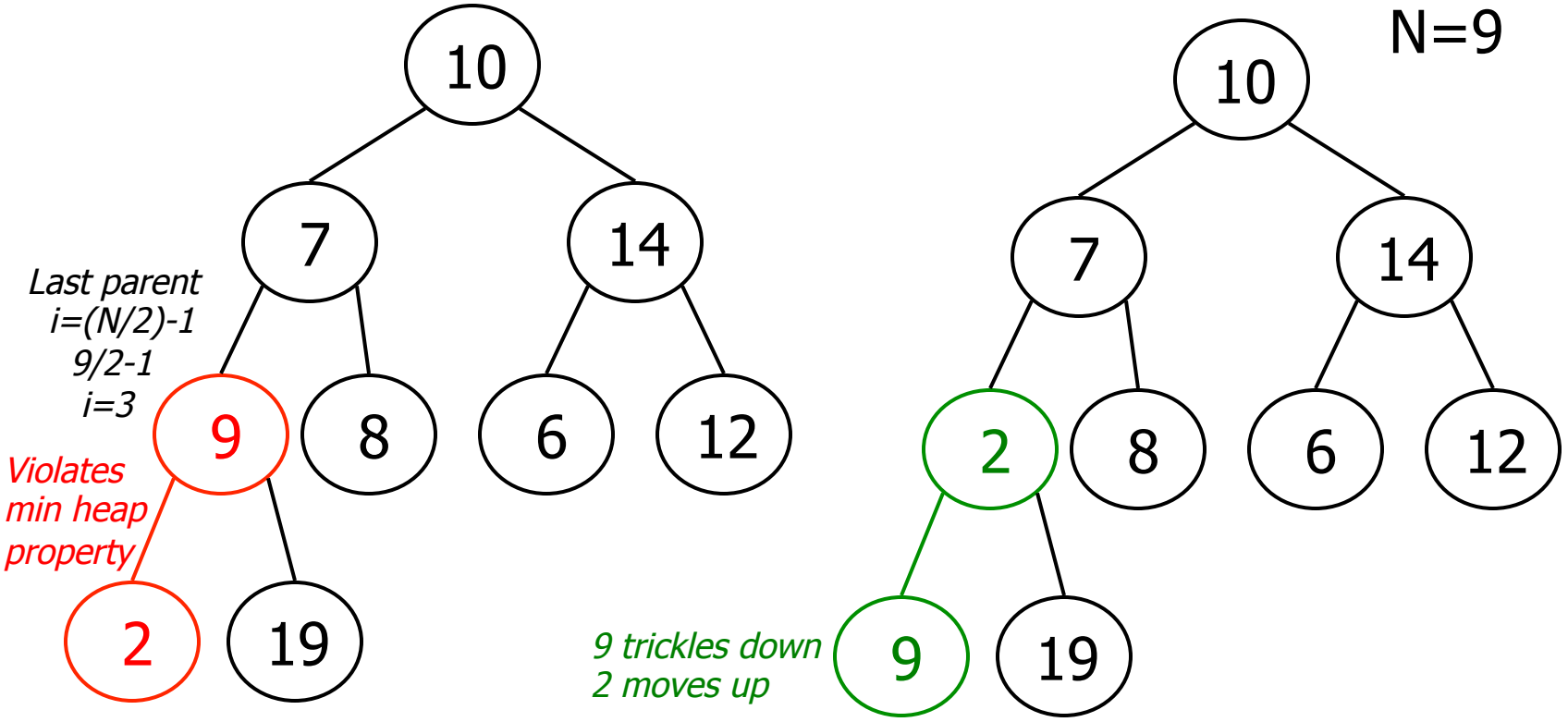
# Building a Heap

- Naïve method
  - Perform N insert operations - O(nlogn)
- Better solution
  - Given an array of unordered items:

*for (int i=(N/2)-1; i >= 0; i--)*
*Trickle down*

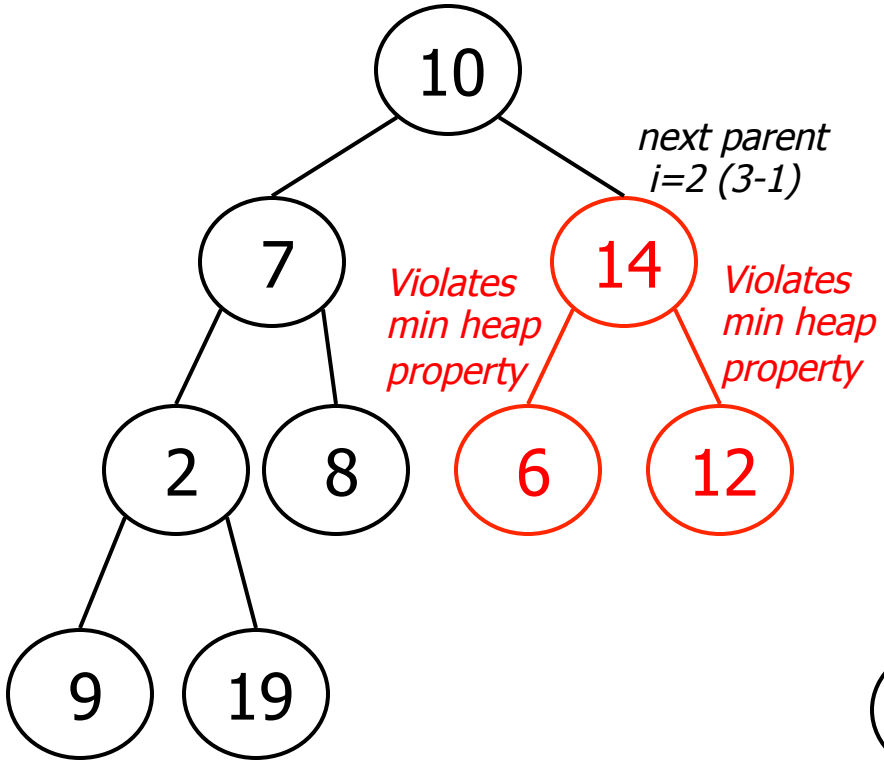
# Building a Min Heap

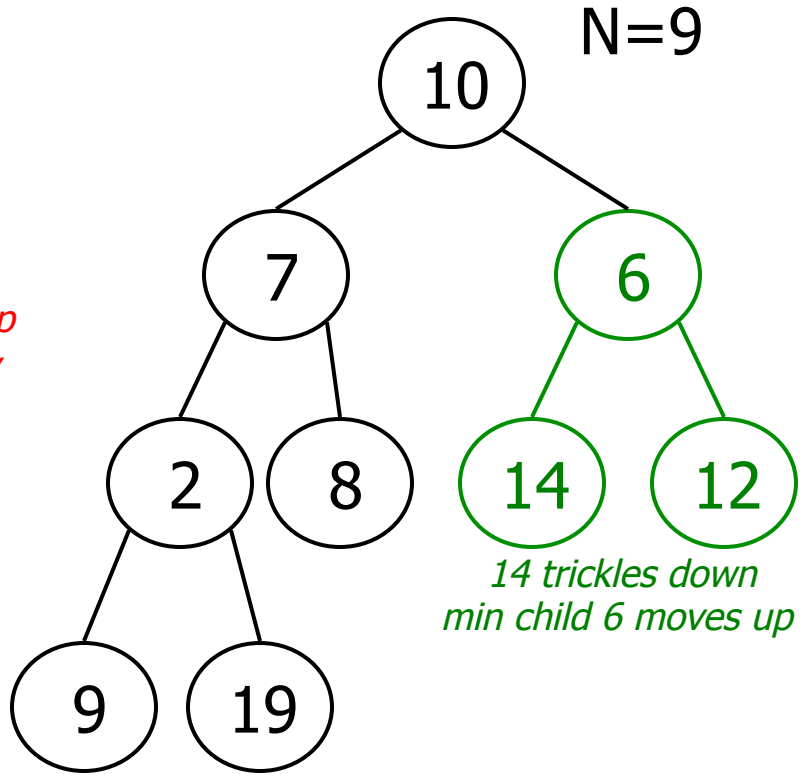
| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 7 | 14 | 9 | 8 | 6 | 12 | 2 | 19 |



# Building a Min Heap

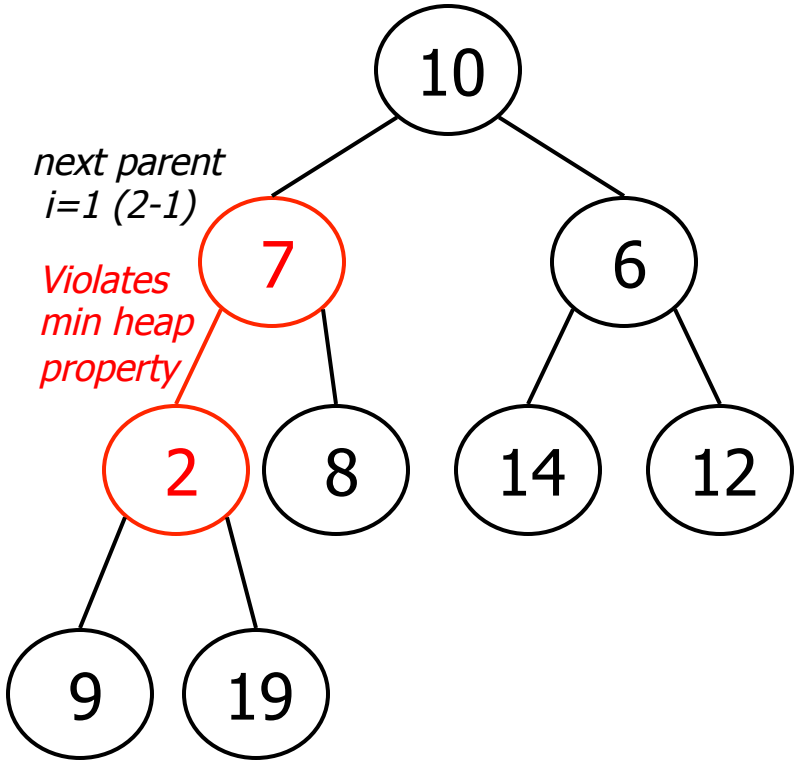
| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 7 | 14 | 2 | 8 | 6 | 12 | 9 | 19 |

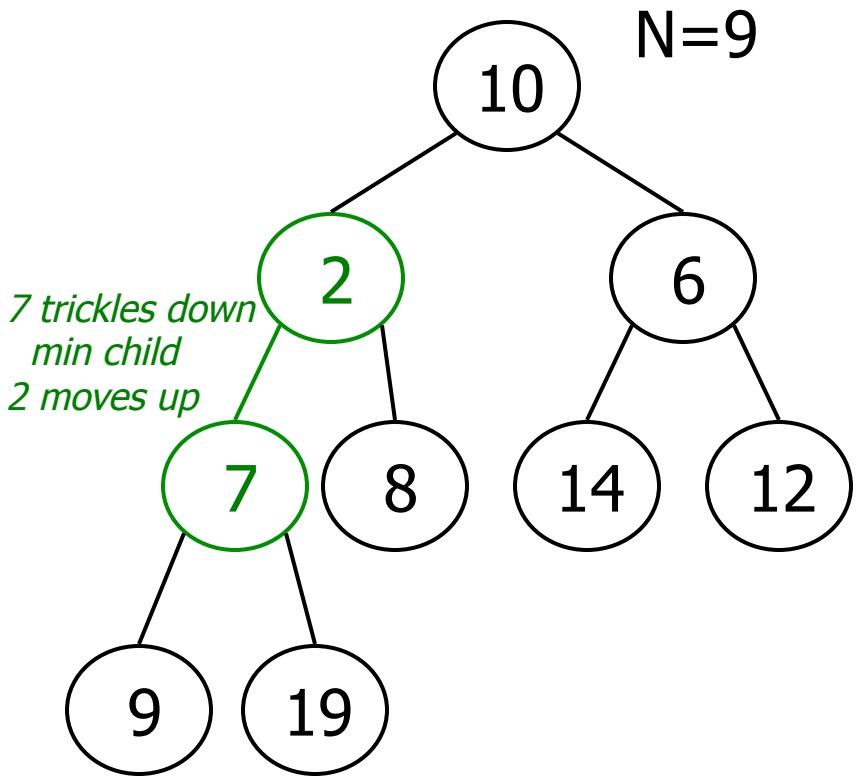


# Building a Min Heap

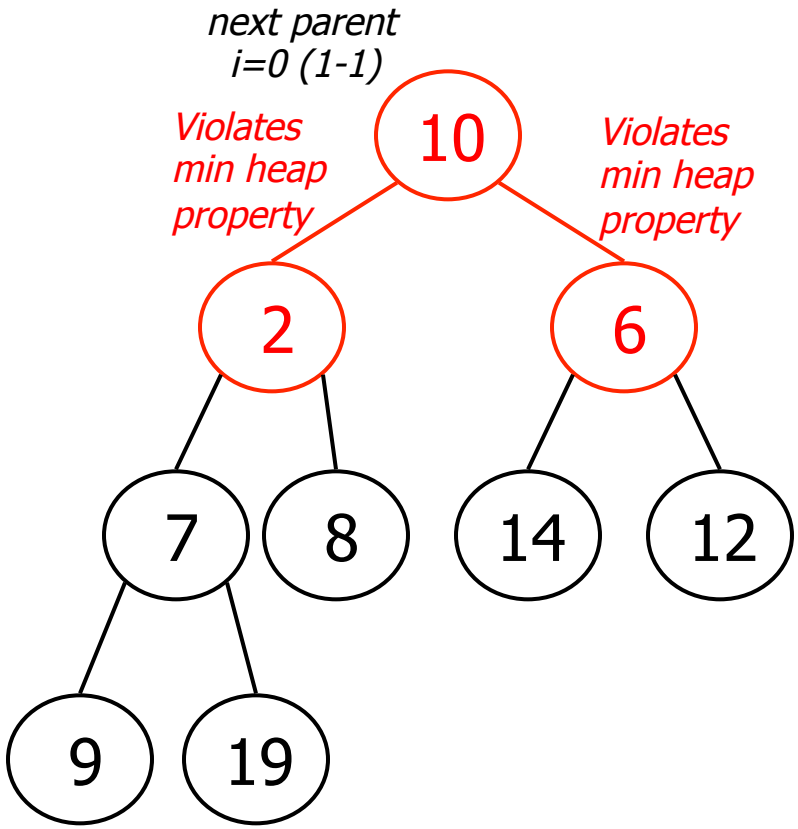
| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 7 | 6 | 2 | 8 | 14 | 12 | 9 | 19 |

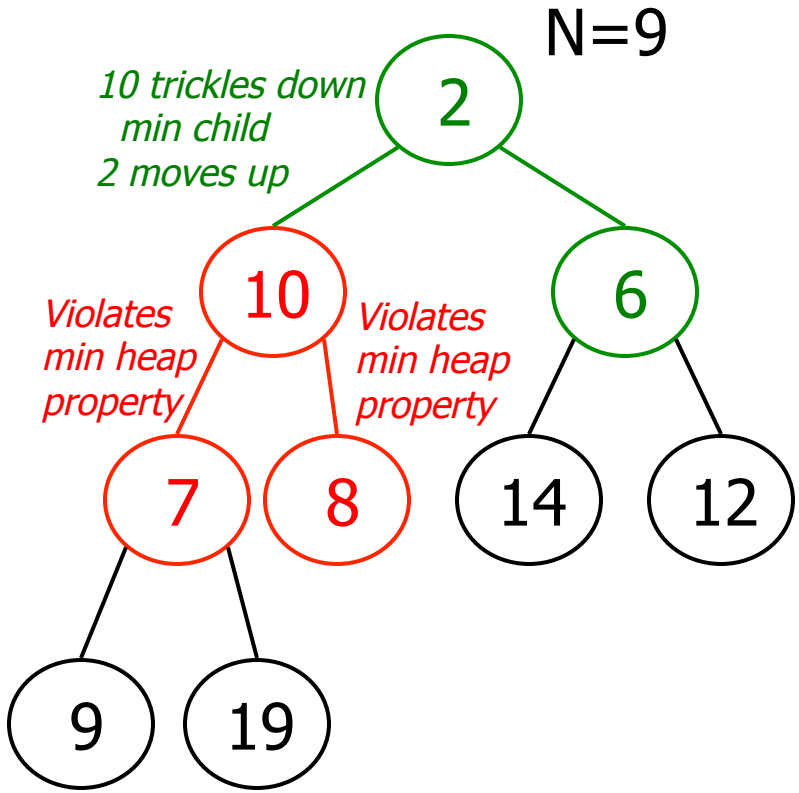


# Building a Min Heap

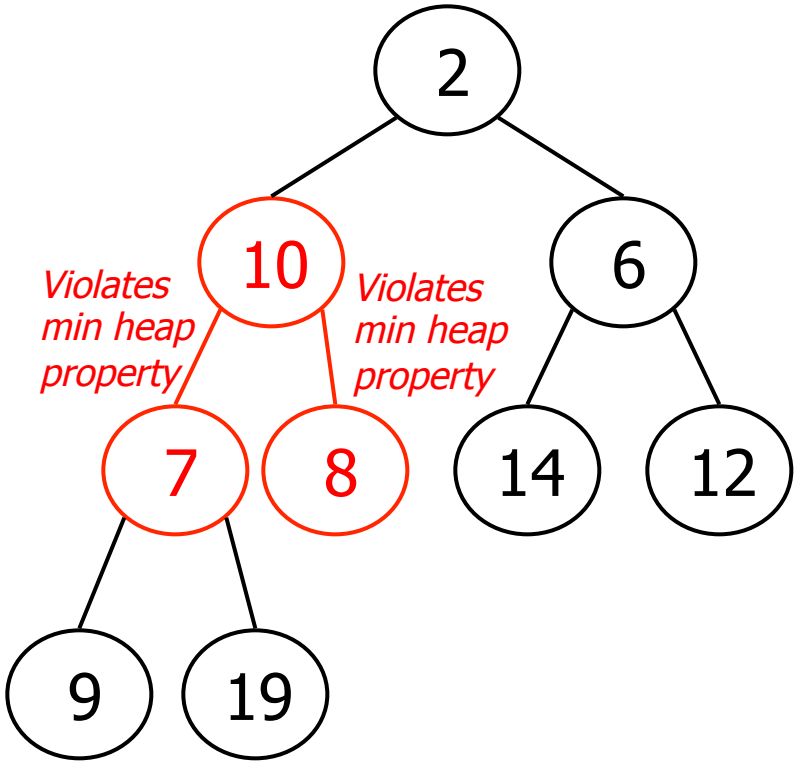
| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 2 | 6 | 7 | 8 | 14 | 12 | 9 | 19 |

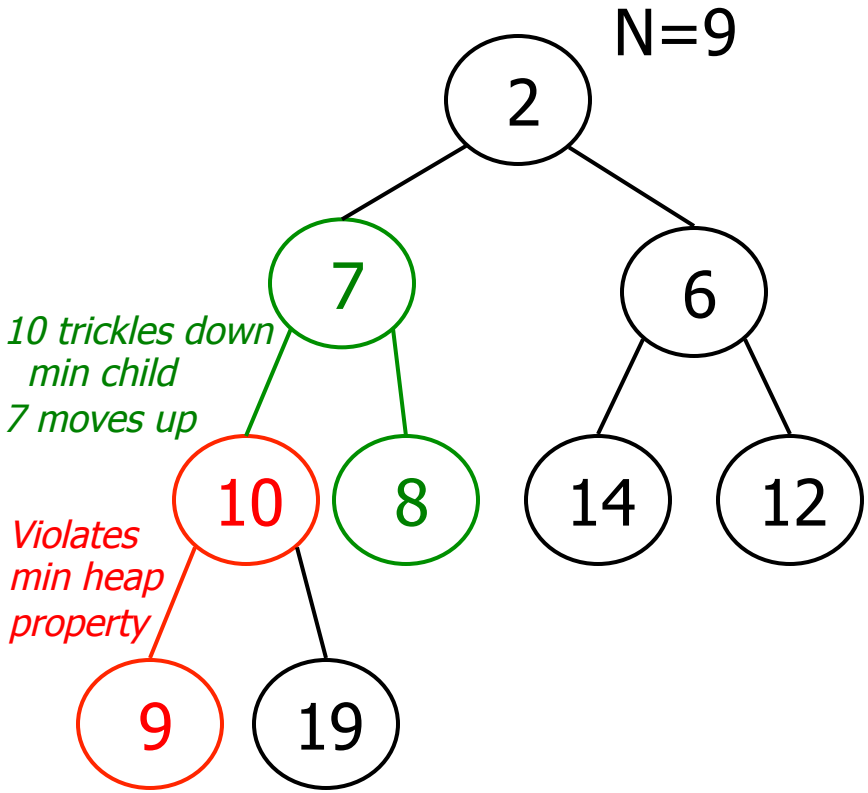


# Building a Min Heap

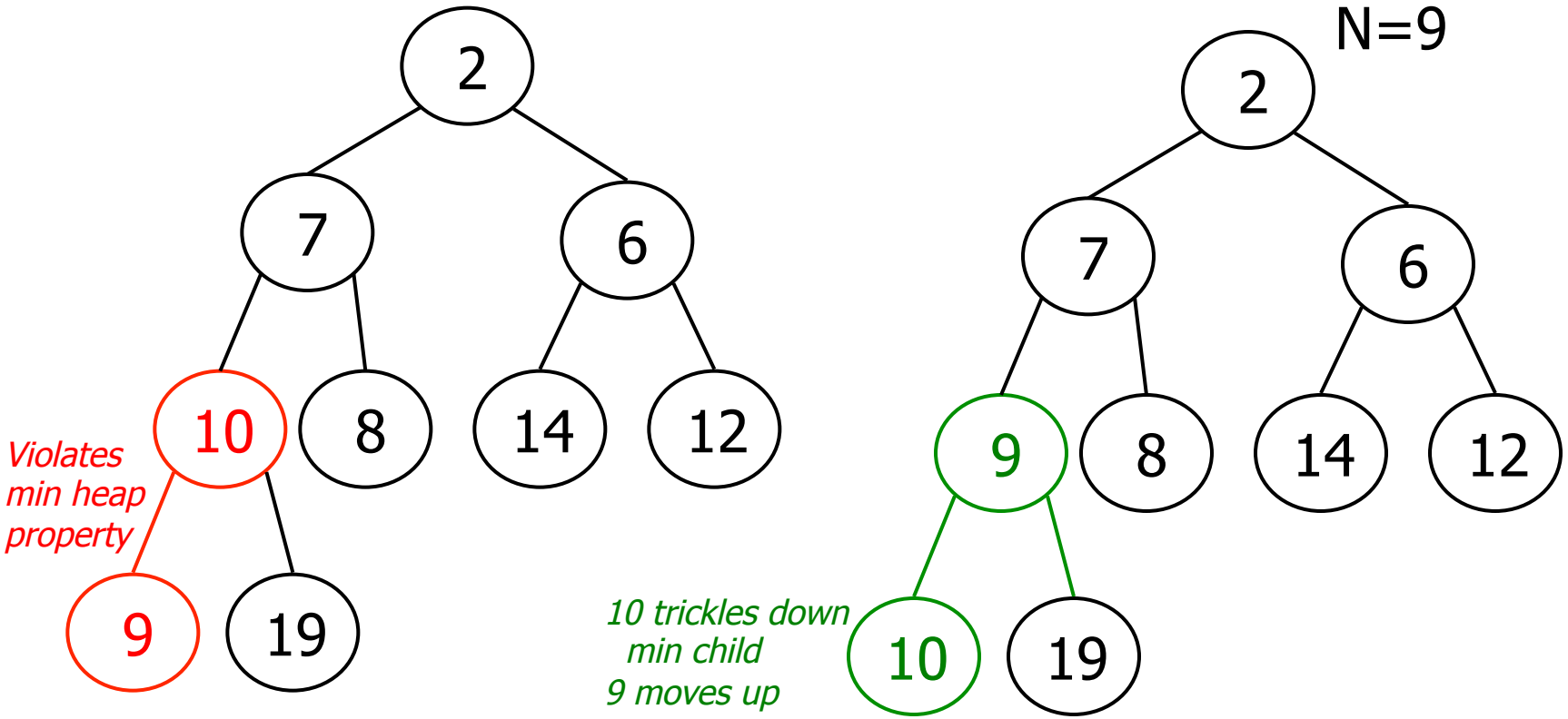
| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 2 | 10 | 6 | 7 | 8 | 14 | 12 | 9 | 19 |



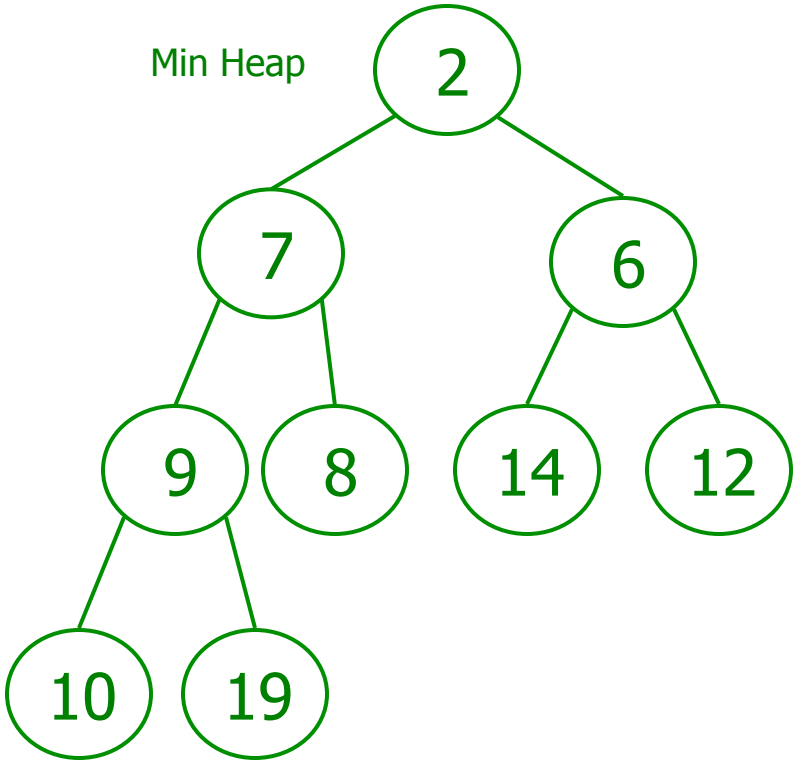
# Building a Min Heap

| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 6 | 10 | 8 | 14 | 12 | 9 | 19 |



# Building a Min Heap

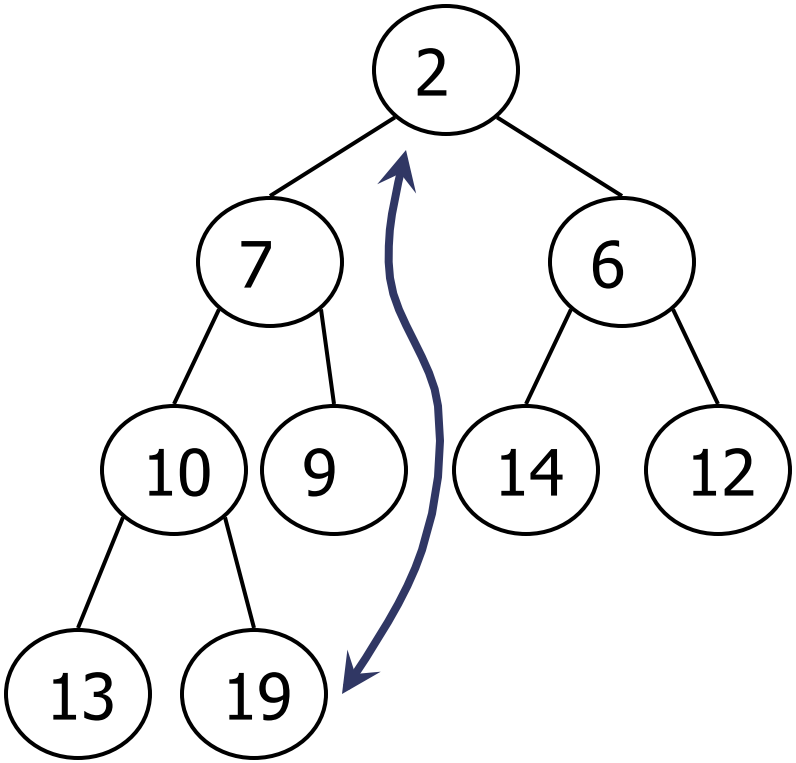
| i=0 | i=1 | i=2 | i=3 | i=4 | i=5 | i=6 | i=7 | i=8 |
|---|---|---|---|---|---|---|---|---|
| 2 | 7 | 6 | 9 | 8 | 14 | 12 | 10 | 19 |

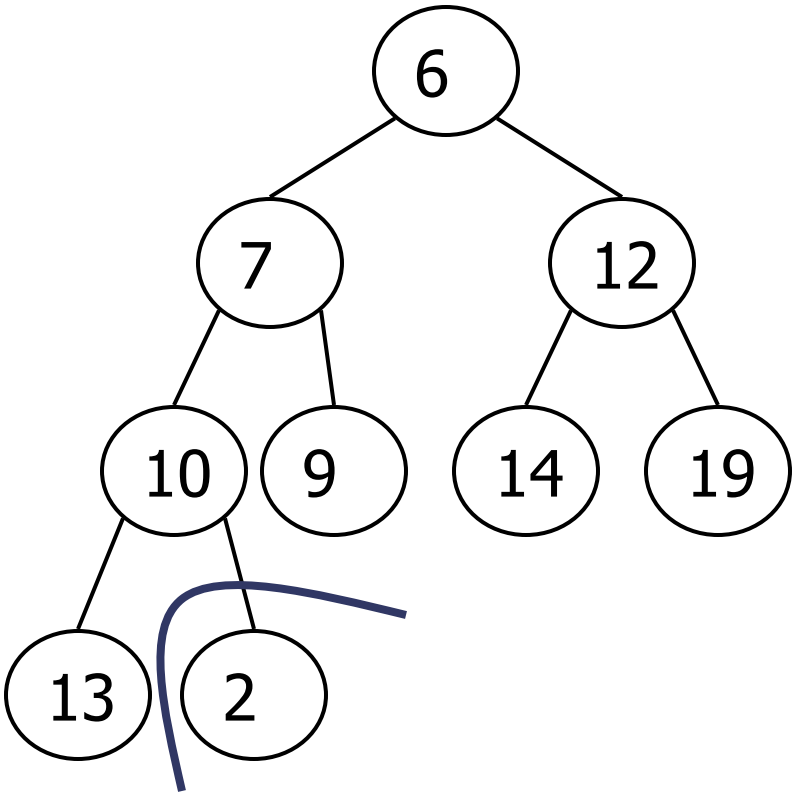


# Heapsort

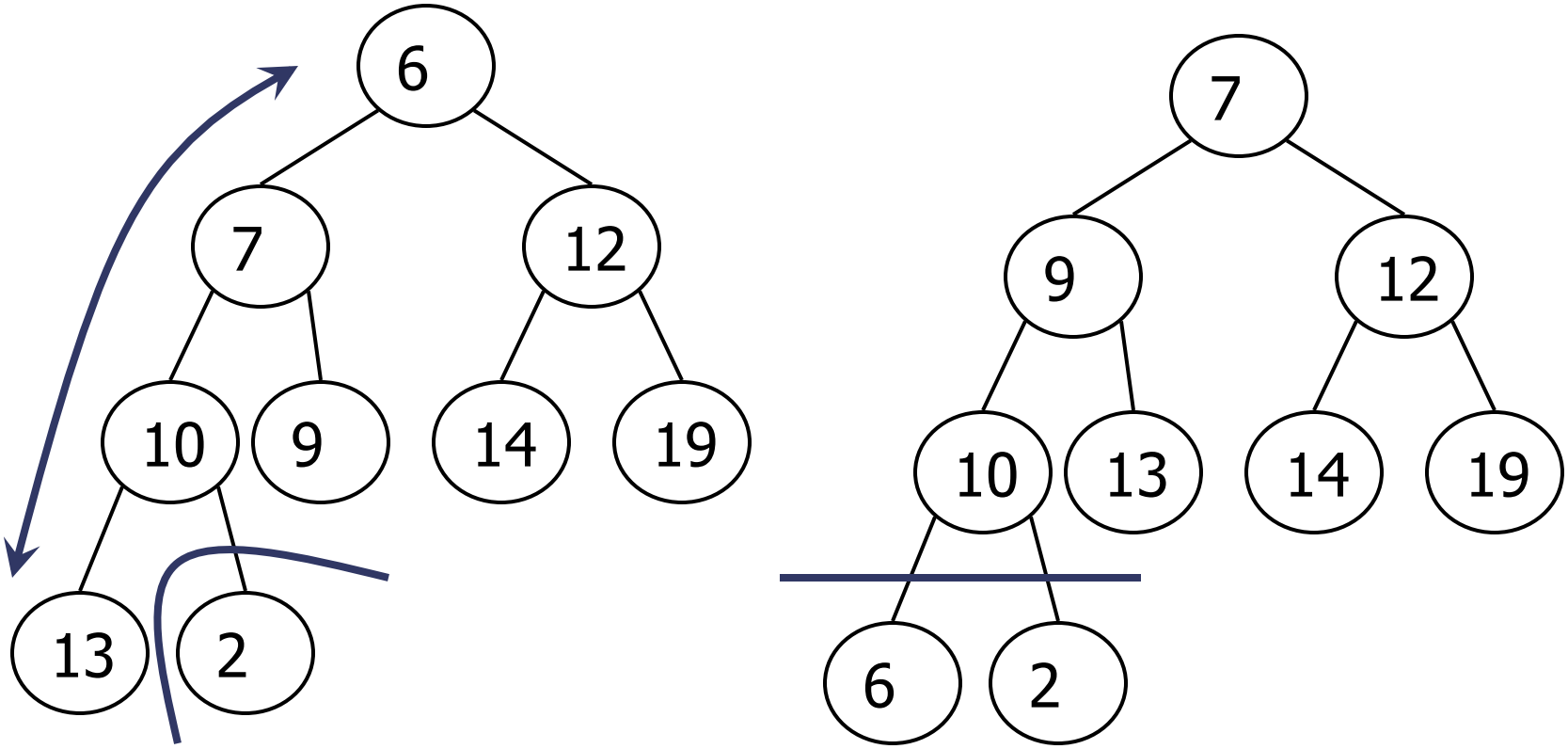
- Build heap for the opposite of what you want
  - Max heap for ascending order
  - Min heap for descending order
- Take root and place in last array position, then think of array as 1 smaller
- Trickle down from root to rebuild heap

# Heapsort - Sort in descending order

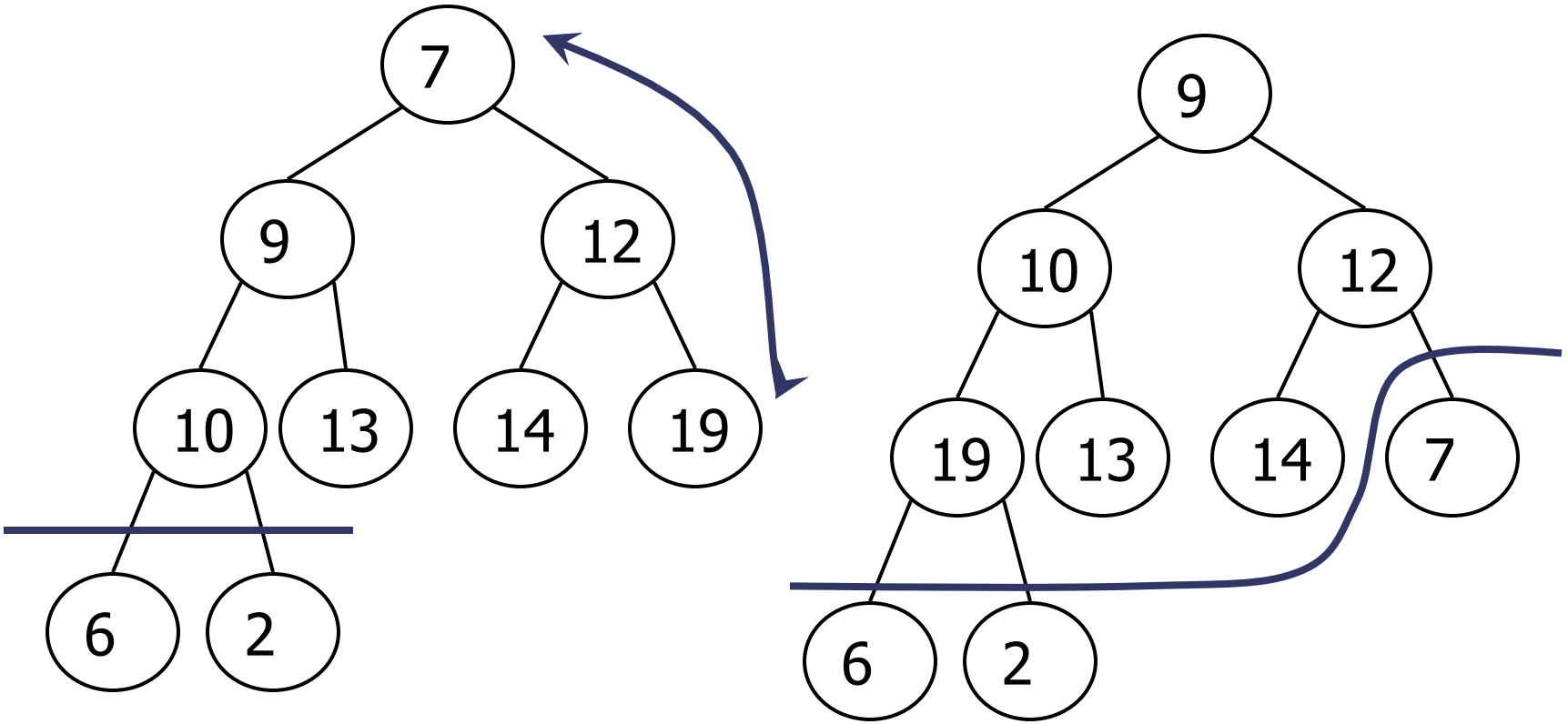


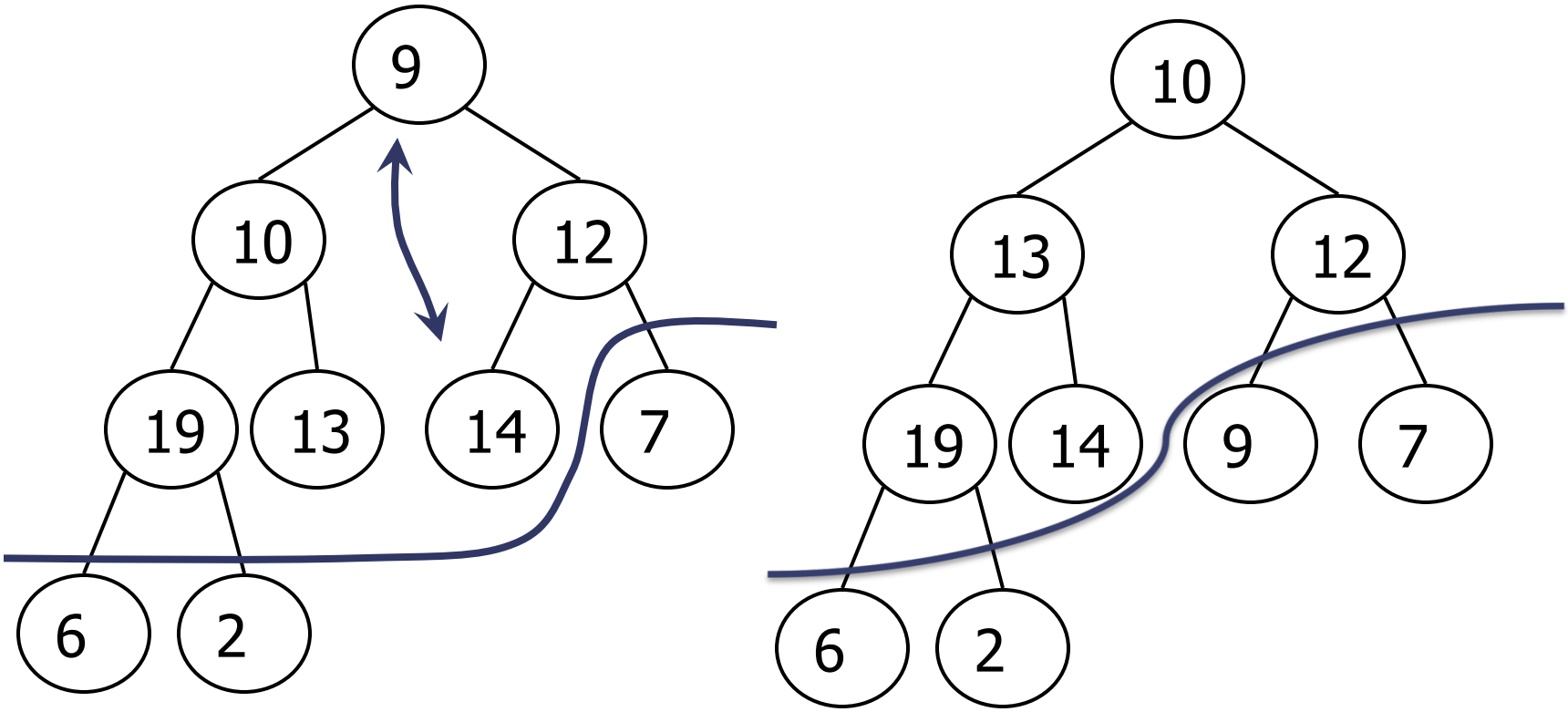
# Heapsort - Sort in descending order



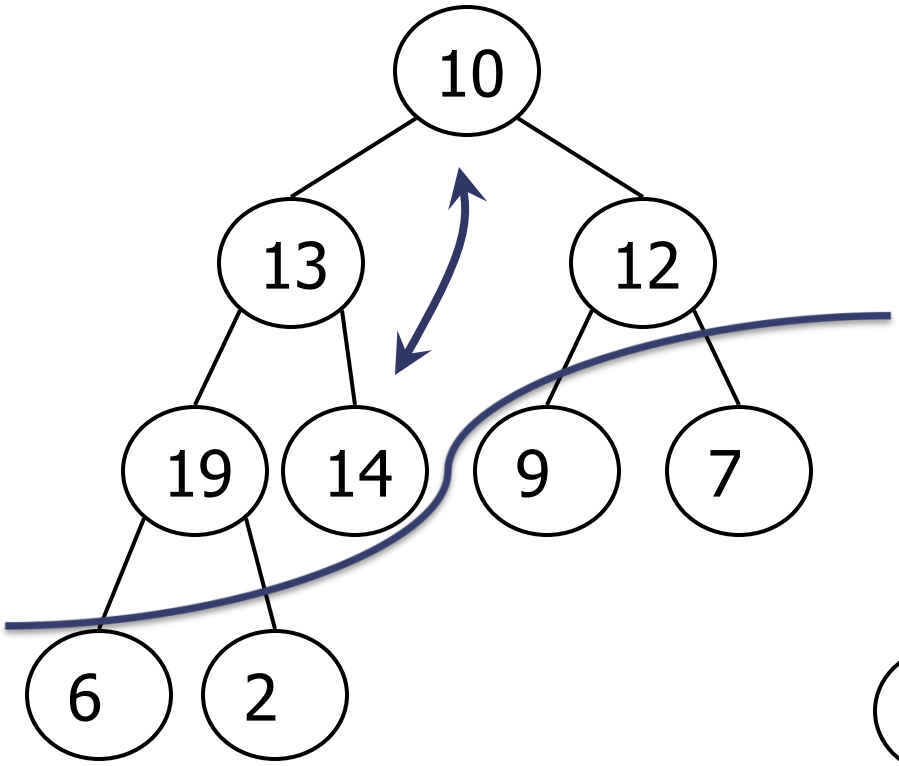
# Heapsort - Sort in descending order

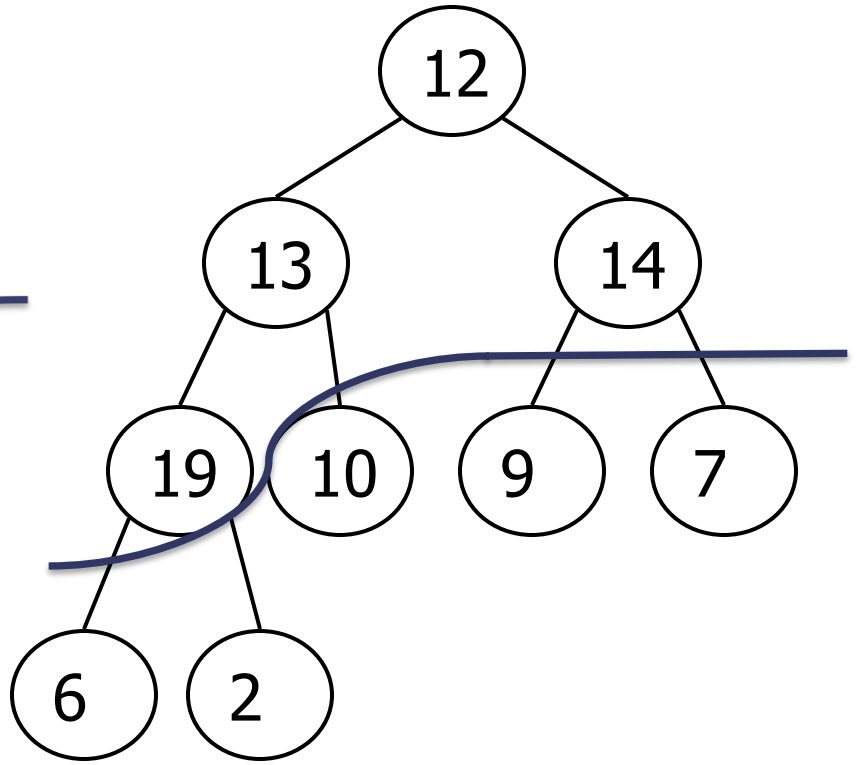


# Heapsort - Sort in descending order

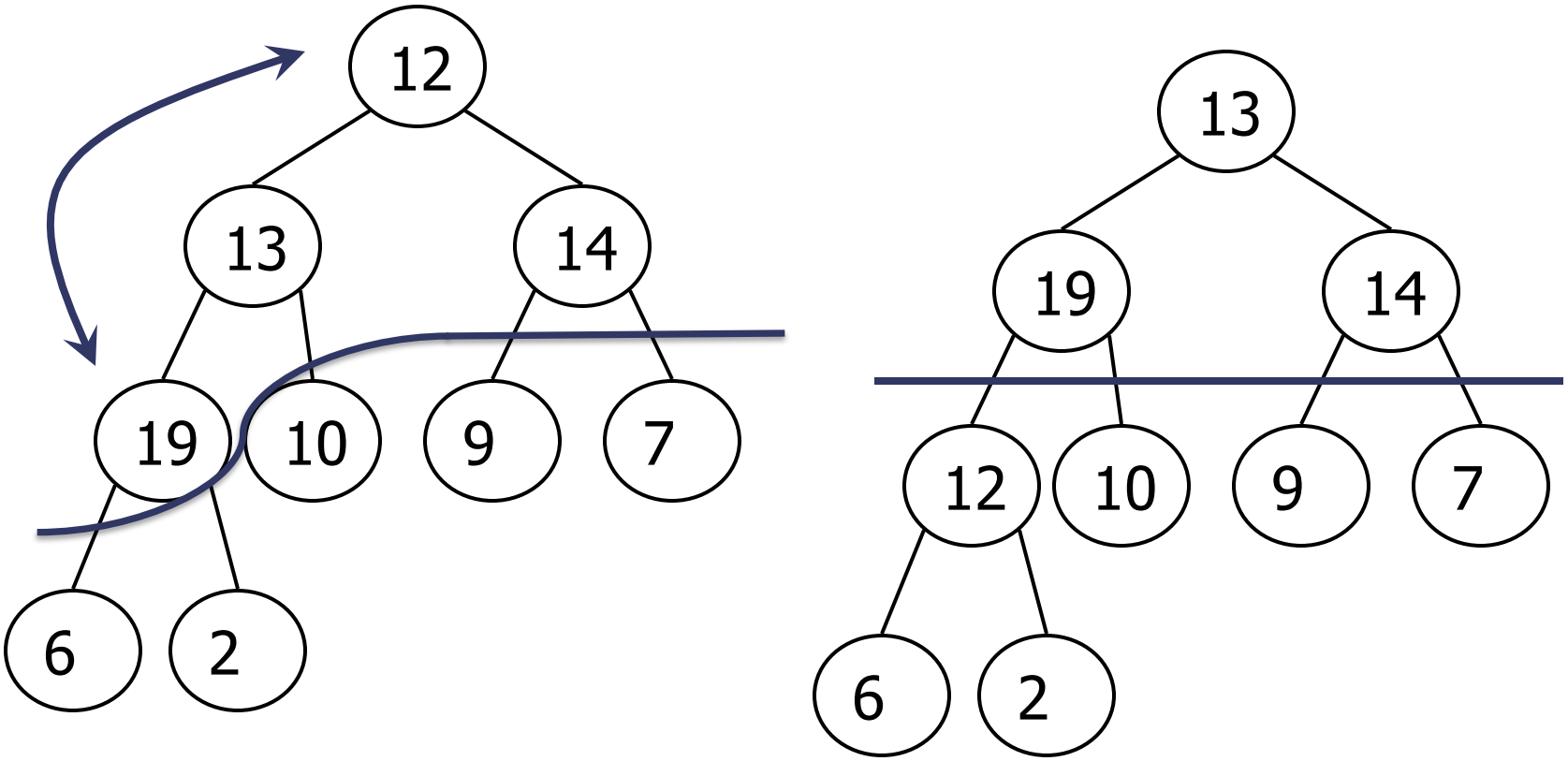


# Heapsort - Sort in descending order

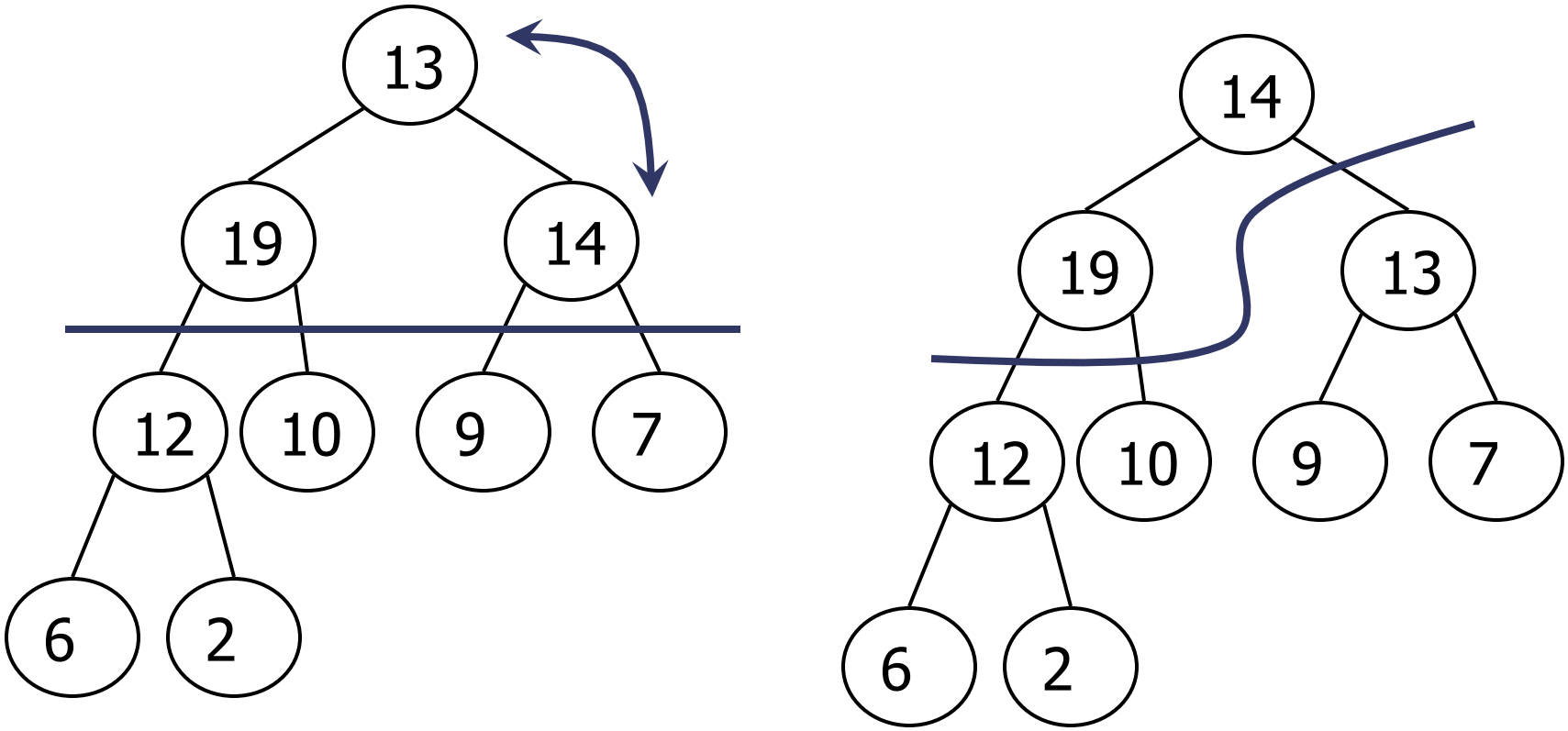


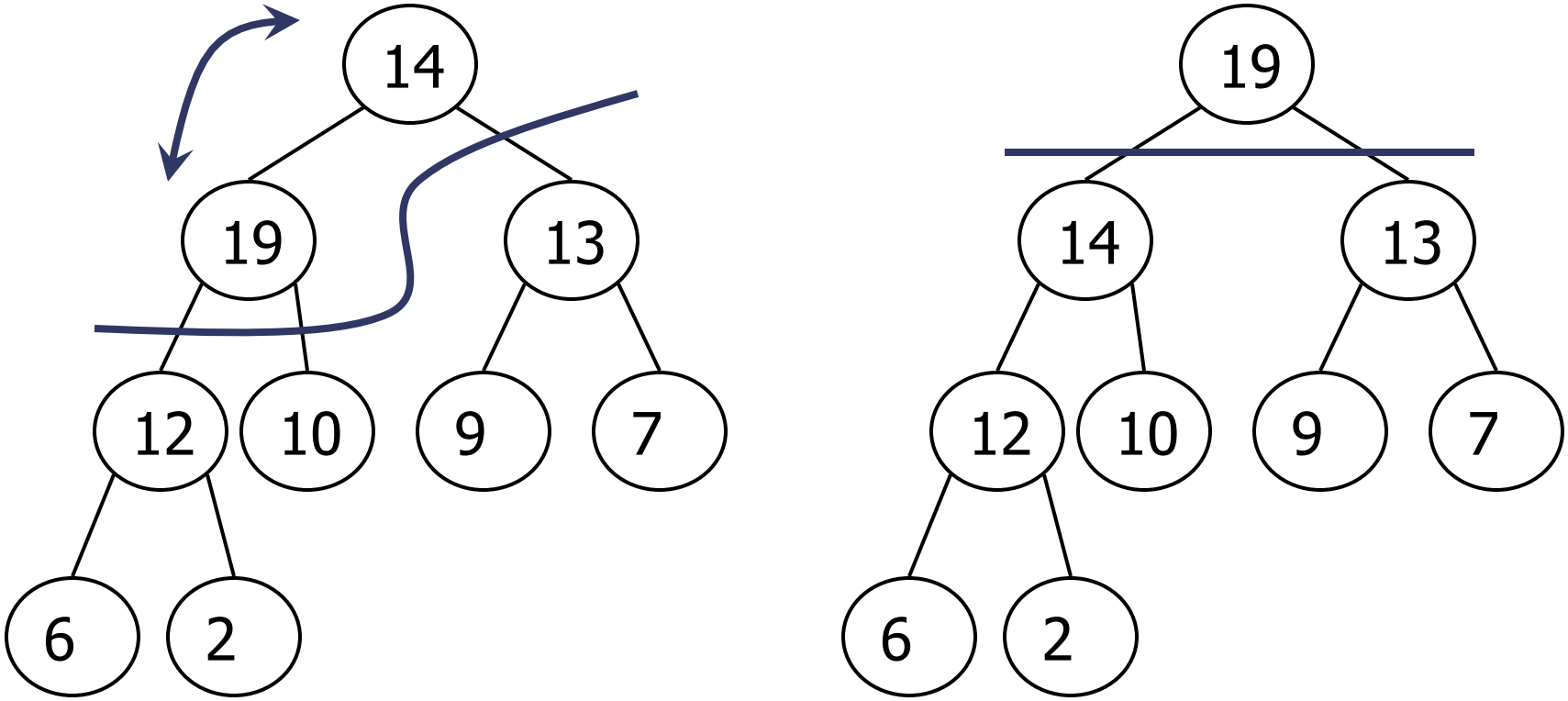
# Heapsort - Sort in descending order



# Heapsort - Sort in descending order



# Heapsort - Sort in descending order



# Heapsort Run Time

- Build step - O(n)
- Sorting step - O(nlogn)
- Heapsort - O(n) + O(nlogn) = O(nlogn)