# Lecture 7: Binary Heaps, Heapsort, Union-Find

Instructor: Saravanan Thirumuruganathan

# Outline

1. Data Structures to speed up algorithms
   - Binary Heap
     - Heapsort
   - Union Find

# In-Class Quizzes

- **URL:** `http://m.socrative.com/`
- **Room Name:** **4f2bb99e**

## Key Things to Know for Data Structures

- Motivation
- Distinctive Property
- Major operations
- Key Helper Routines
- Representation
- Algorithms for major operations
- Applications

# Data Structures for Algorithmic Speedup

- BST and RBT are two examples of data structures to represent dynamic set
- Today's topic, Heap and Union-Find, can also be used to represent dynamic set
- However, these are used more often to speed up algorithms

# Binary Heap

# Motivation

- Heap Sort (CLRS is organized that way!)
- Priority Queue
- Most space efficient data structure

# Priority Queue

- "Queue" data structure has a FIFO property
- Some times it is useful to consider **priority**
- Output element with highest priority first

# Priority Queue - Major Operations

- Insert
- FindMin (resp. FindMax)
- DeleteMin (resp. DeleteMax)
- DecreaseKey (resp. IncreaseKey)

# Priority Queue - Applications[1]

- Dijkstra's shortest path algorithm
- Prim's MST algorithm
- Heapsort
- Online median
- Huffman Encoding
- A* Search (or any Best first search)
- Discrete event simulation
- CPU Scheduling
- ...
- See Wikipedia entry for priority for details

[1] Kleinberg-Tardos Book and Wikipedia

# Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert:

# Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin:

# Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin:

# Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin: $O(1)$
  - DecreaseKey:

# Priority Queue - Candidate Implementations

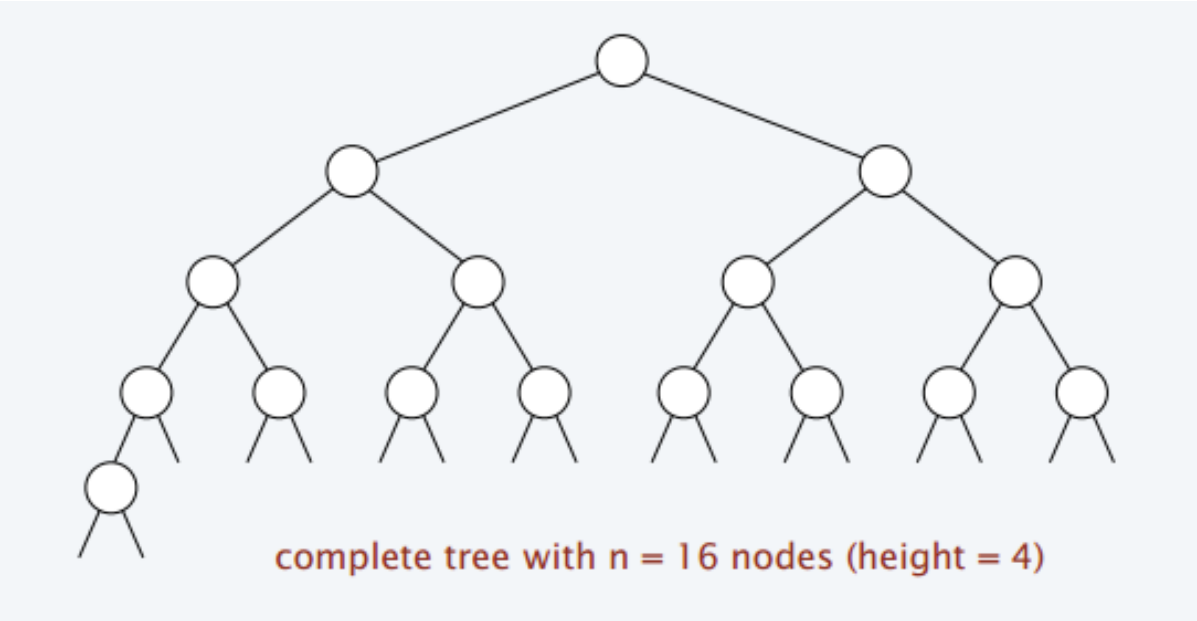
- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin: $O(1)$
  - DecreaseKey: $O(1)$
- Binary Heap
  - Insert: $O(\lg n)$
  - FindMin: $O(1)$
  - DeleteMin: $O(\lg n)$
  - DecreaseKey: $O(\lg n)$
- Binomial Heaps, Fibonacci Heaps etc.

# Binary Heaps

- Perfect data structure for implementing Priority Queue
- MaxHeap and MinHeap
- We will focus on MaxHeaps in this lecture

# Complete Tree[2]

- Perfectly balanced, except for bottom level
- Elements were inserted top-to-bottom and left-to-right

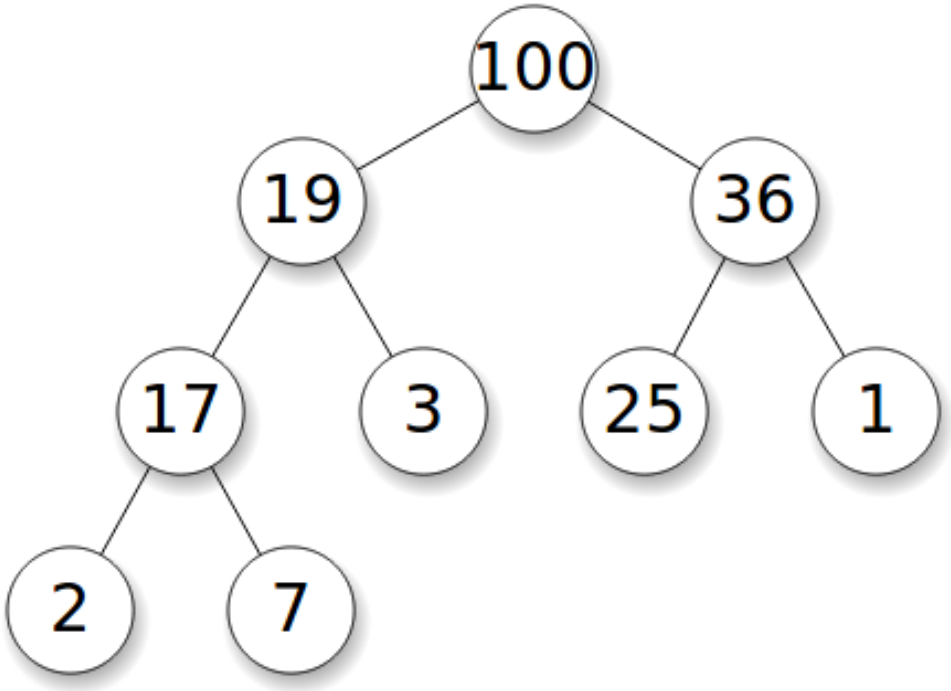



[2] http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/BinomialHeaps.pdf

# Heap Property

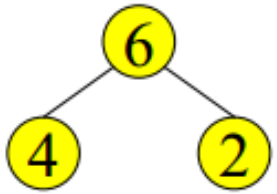
- Heap is a binary tree (**NOT BST**)
- Heap:
  - **Completeness** Property: Heap has restricted structure. It must be a complete binary tree .
  - **Ordering** Property: Relates parent value with that of its children
- MaxHeap property: Value of parent must be greater than **both** its children
- MinHeap property: Value of parent must be less than **both** its children
- Heap with $n$ elements has height $O(\lg n)$

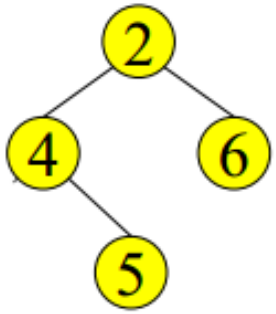
# Max Heap Example[3]

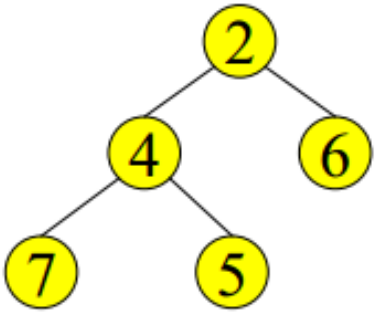


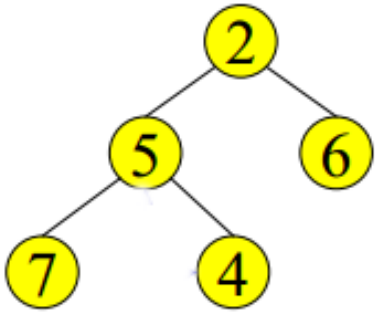

[3]Wikipedia page for Heap

# Heap Property[4]




[4] http://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture10.pdf

# Major Operations

- Insert
- FindMax
- DeleteMax (aka ExtractMax)
- IncreaseKey

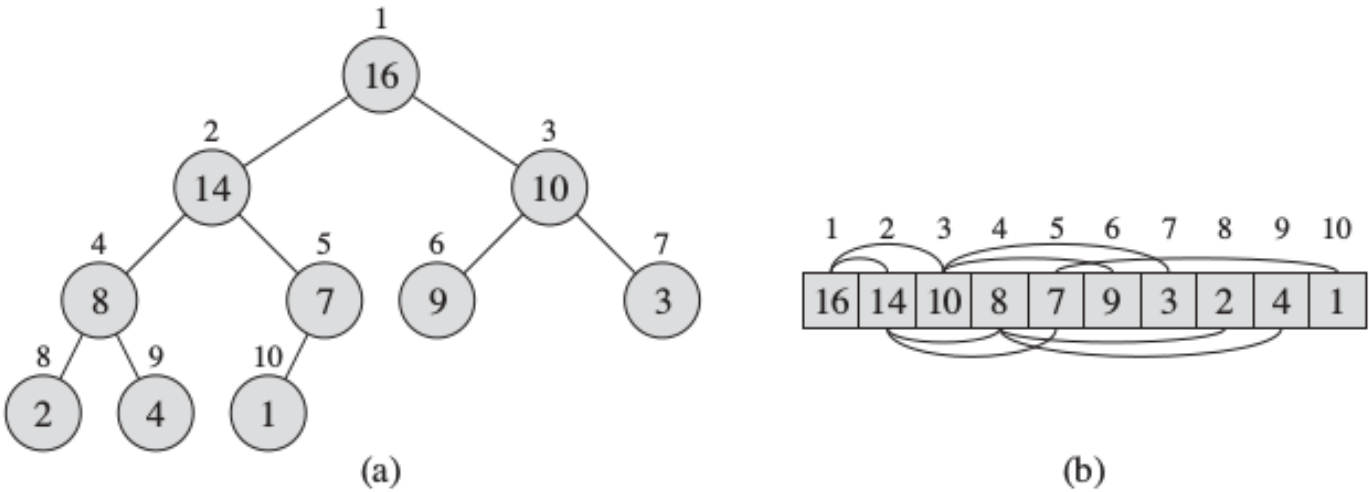
# Key Helper Routines

- Max-Heapify (or Min-Heapify)
- Bubble-Up
- Bubble-Down
- Heapify

# Representation: Arrays

- Very efficient implementation using arrays
- Possible due to completeness property
- Parent($i$): return $\lfloor i/2 \rfloor$
- LeftChild($i$): return $2i$
- RightChild($i$): return $2i + 1$

# Representation: Arrays[5]

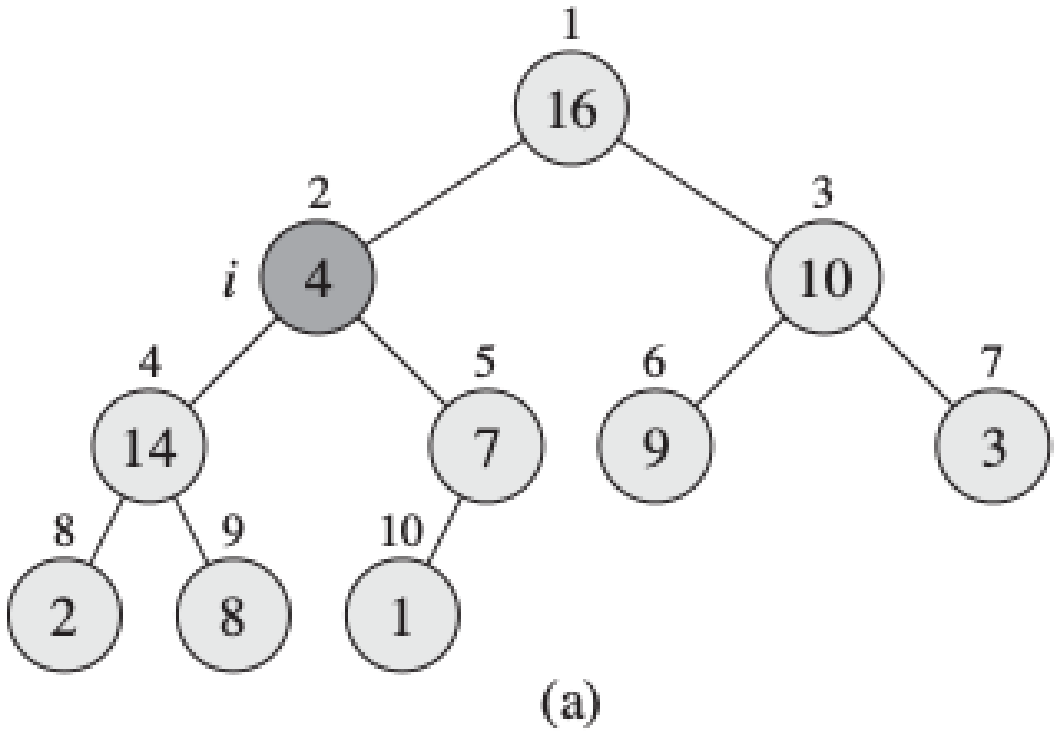



[5]CLRS Fig 6.1

# Max-Heapify

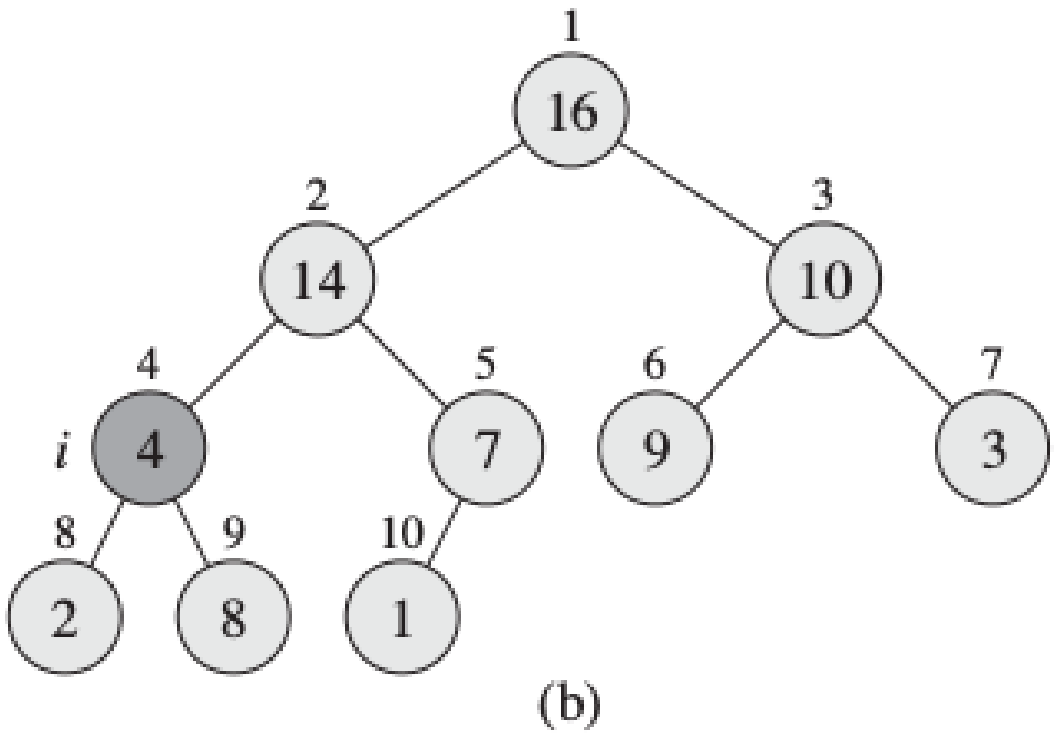
- Objective: Maintain heap property
- Invocation: Max-Heapify($A, i$)
- Assume: Left($i$) and Right($i$) are valid max-heaps
- $A[i]$ might violate max-heap property
- Bubble-Down the violation
- **Analysis:** $O(\lg n)$

# Max-Heapify: Example[6]



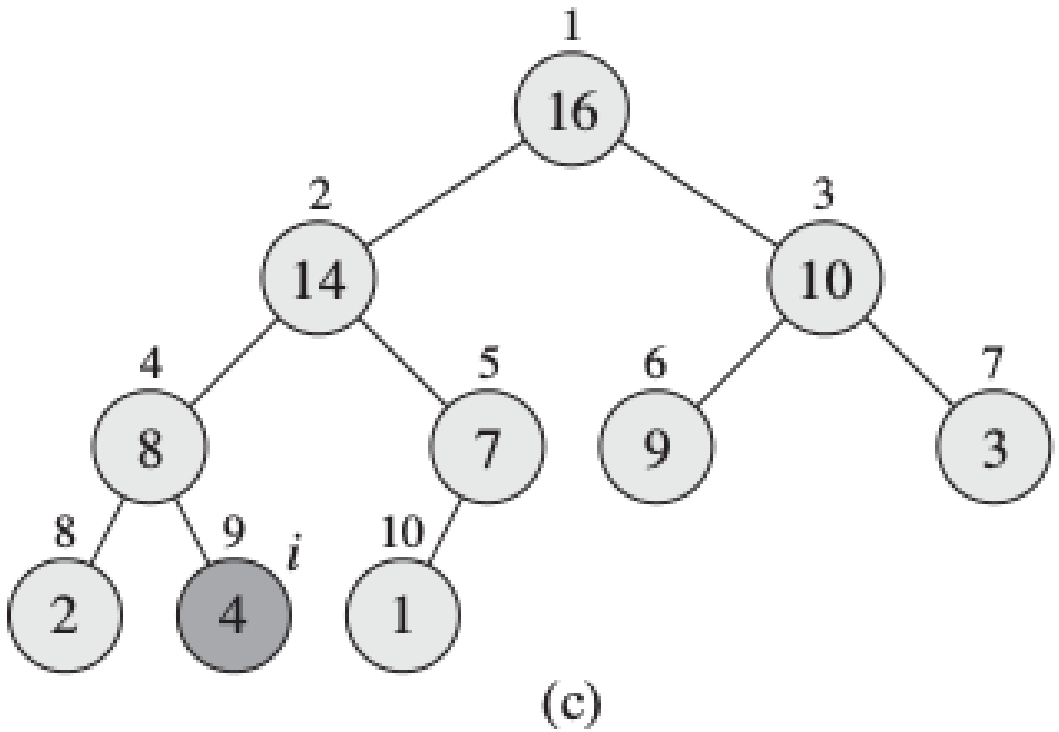

[6] CLRS Fig 6.2

# Max-Heapify: Example[7]



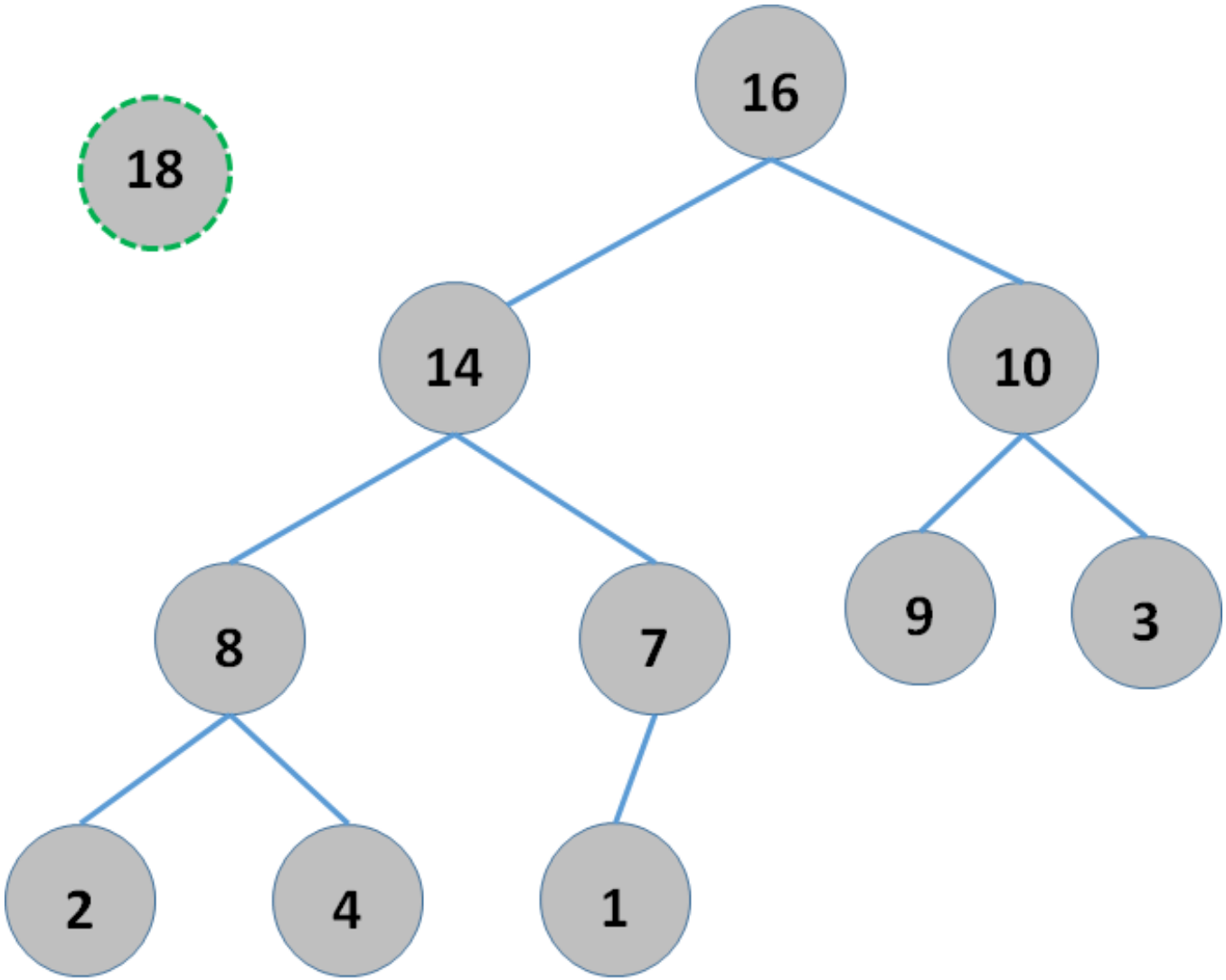

[7] CLRS Fig 6.2

# Max-Heapify: Example[8]

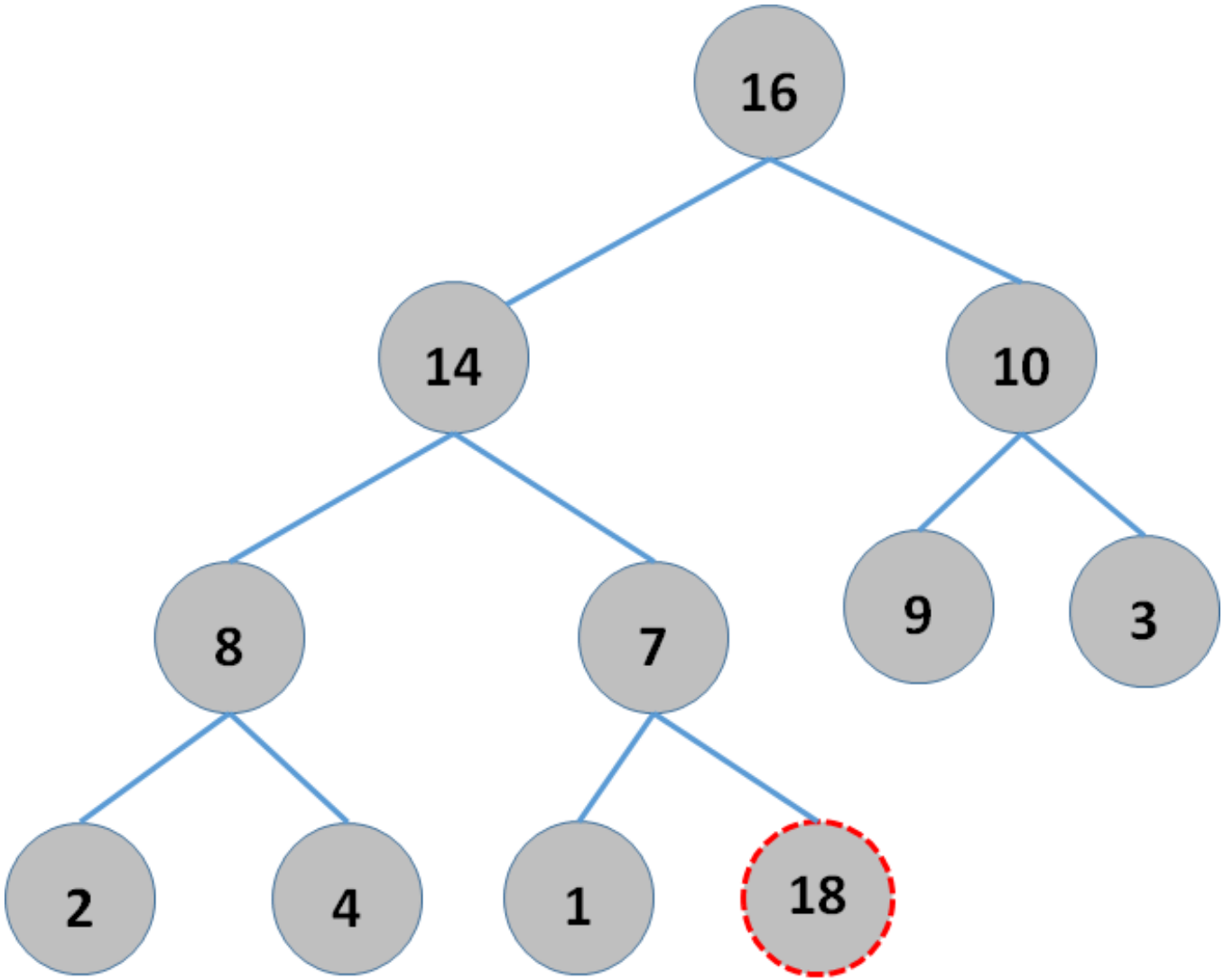



[8] CLRS Fig 6.2

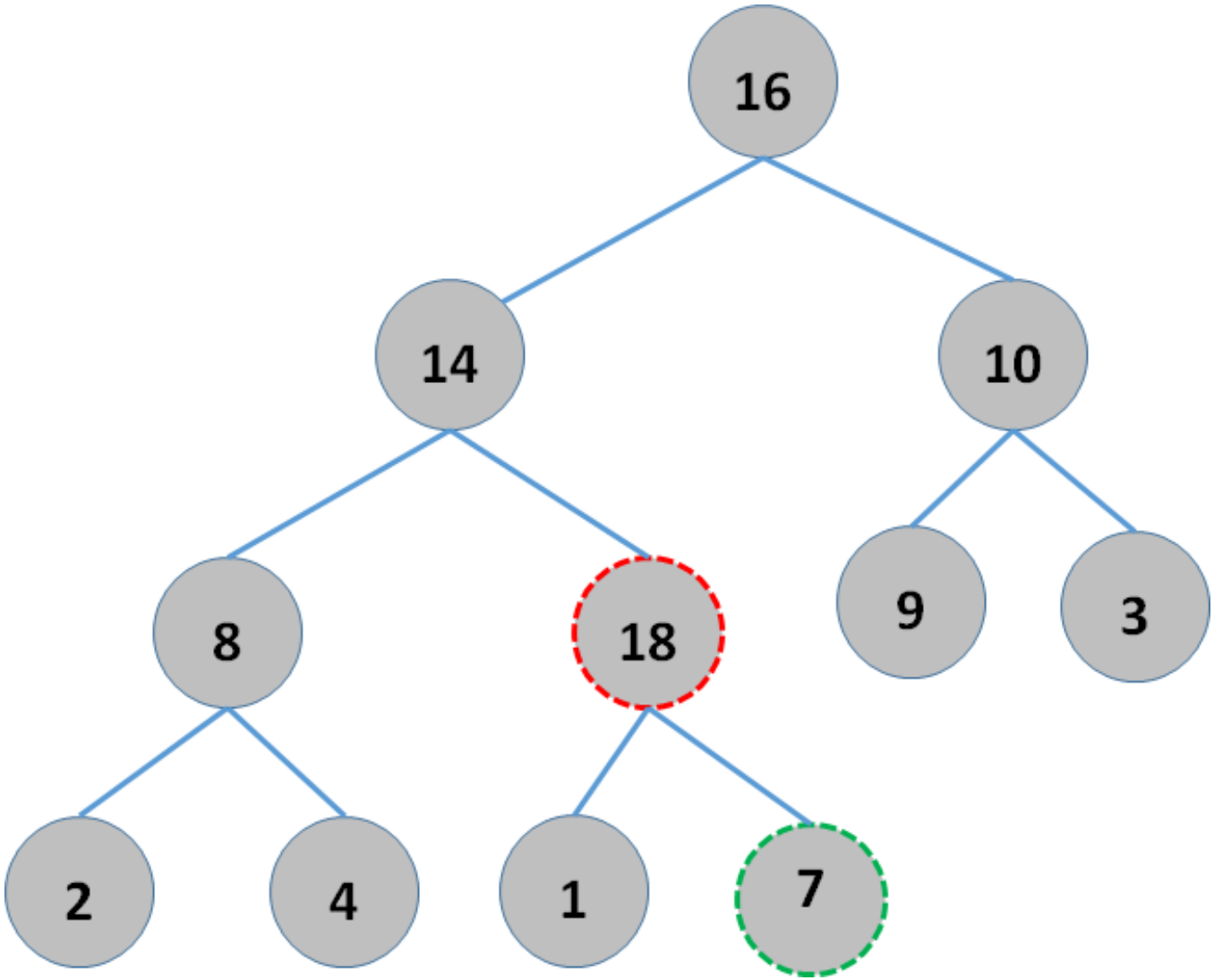
# Heap : Insert



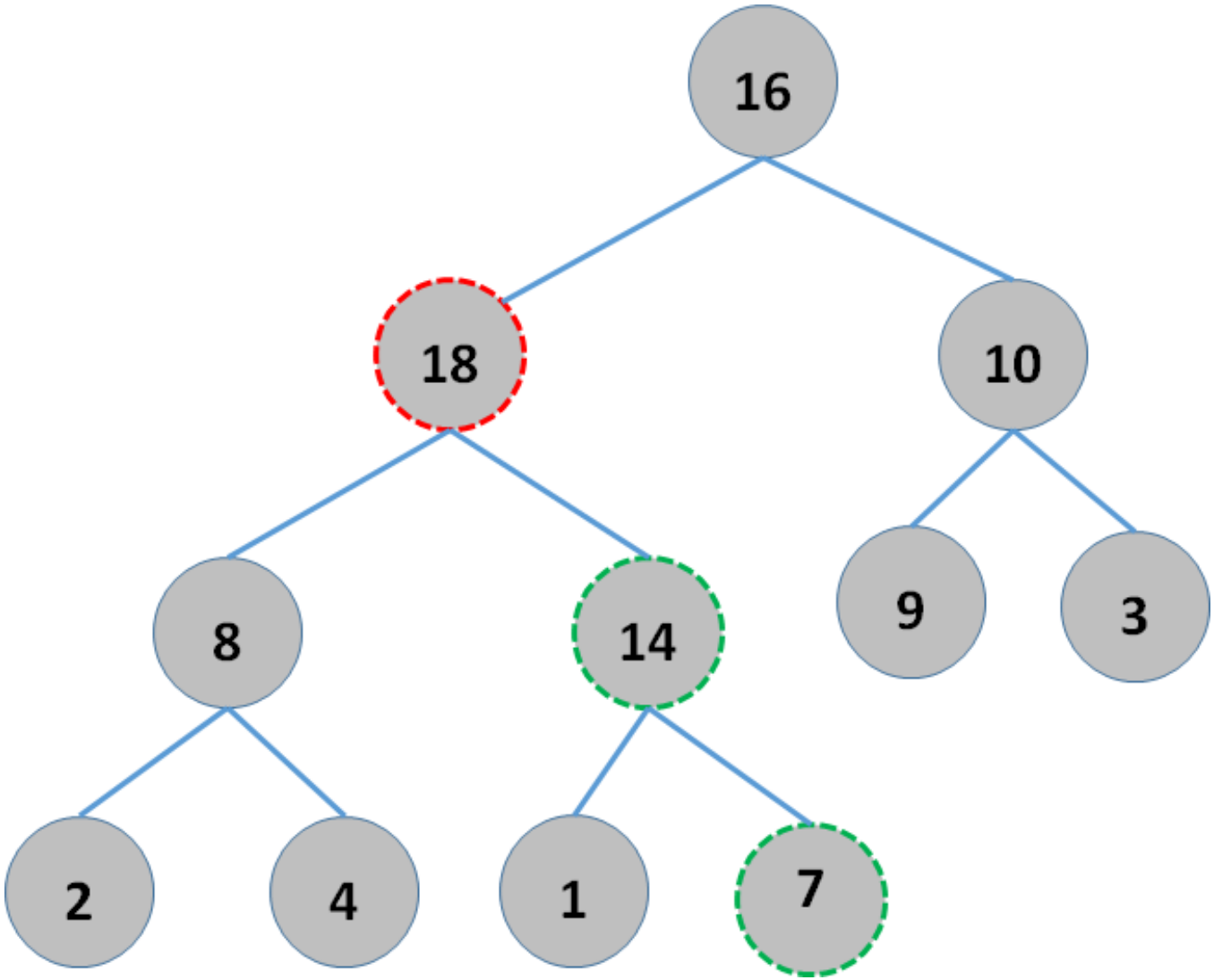
# Heap : Insert



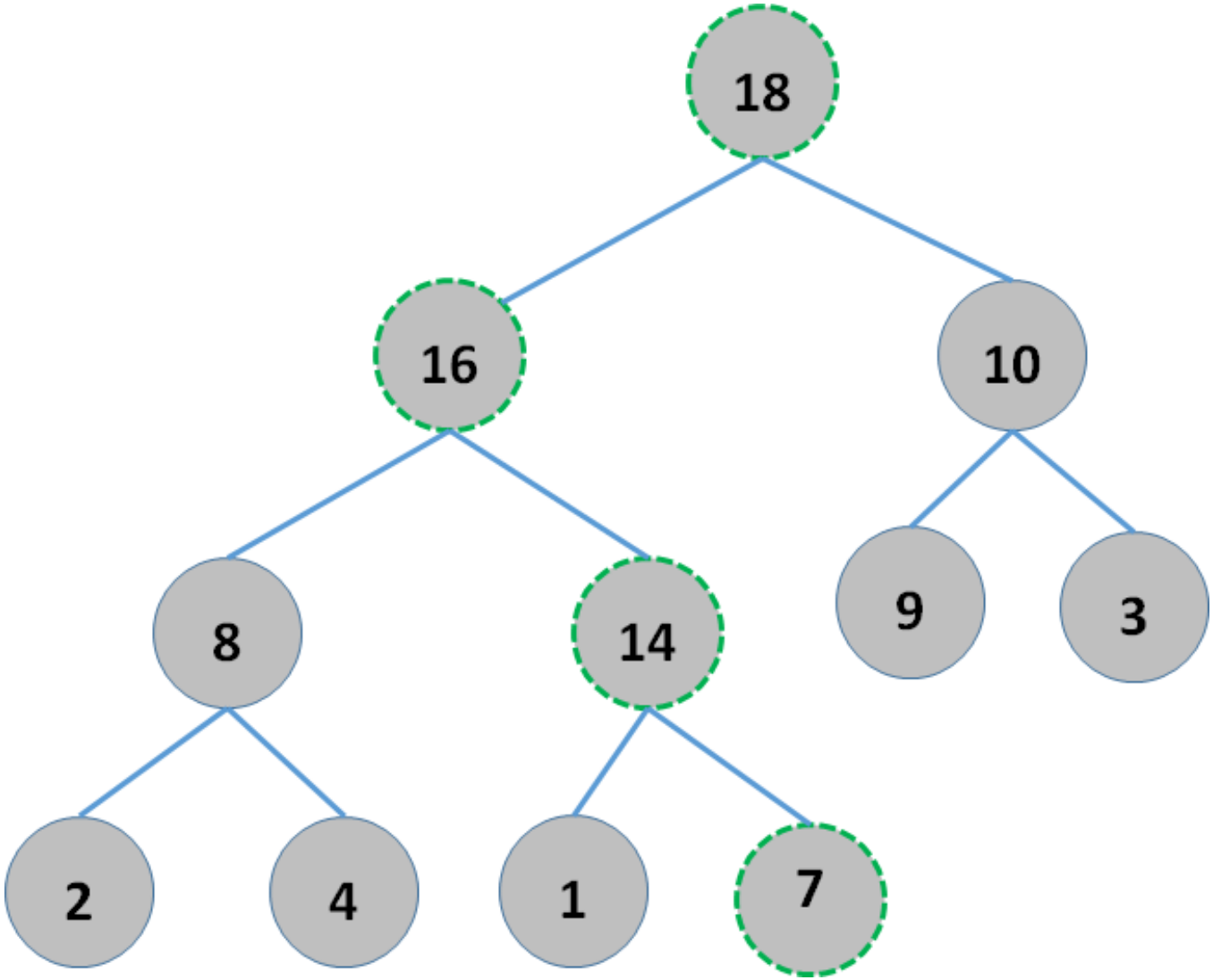
# Heap : Insert



# Heap : Insert



# Heap : Insert



# Heap : Insert

- Insert element at first available slot (no completeness property violation!)
- Fix heap property violations by bubbling up the vilolation till it is fixed
- Complexity:

# Heap : Insert

- Insert element at first available slot (no completeness property violation!)
- Fix heap property violations by bubbling up the vilolation till it is fixed
- Complexity: $O(\lg n)$

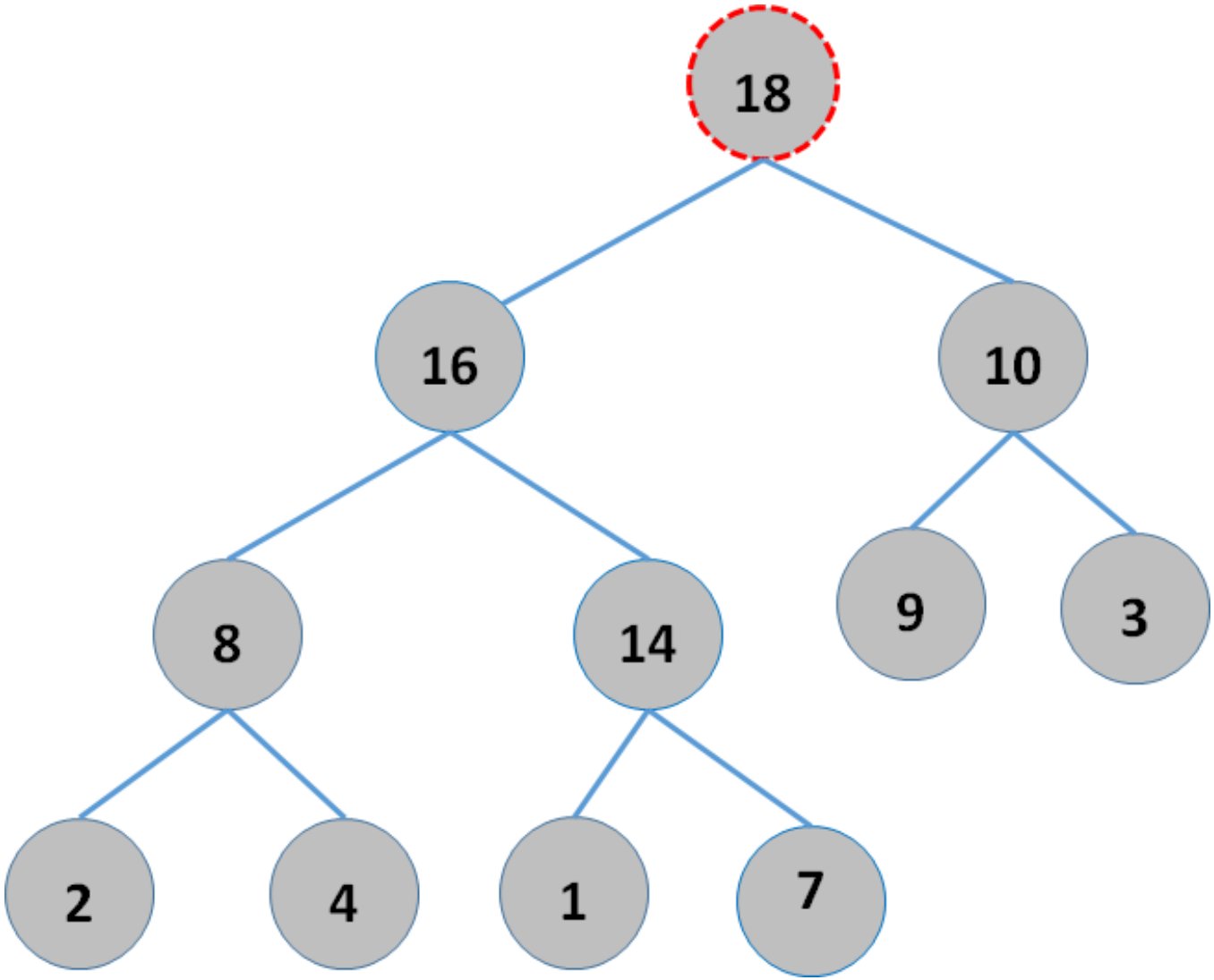
# Heap : FindMax

- Look at the root element
- Time complexity: $O(1)$

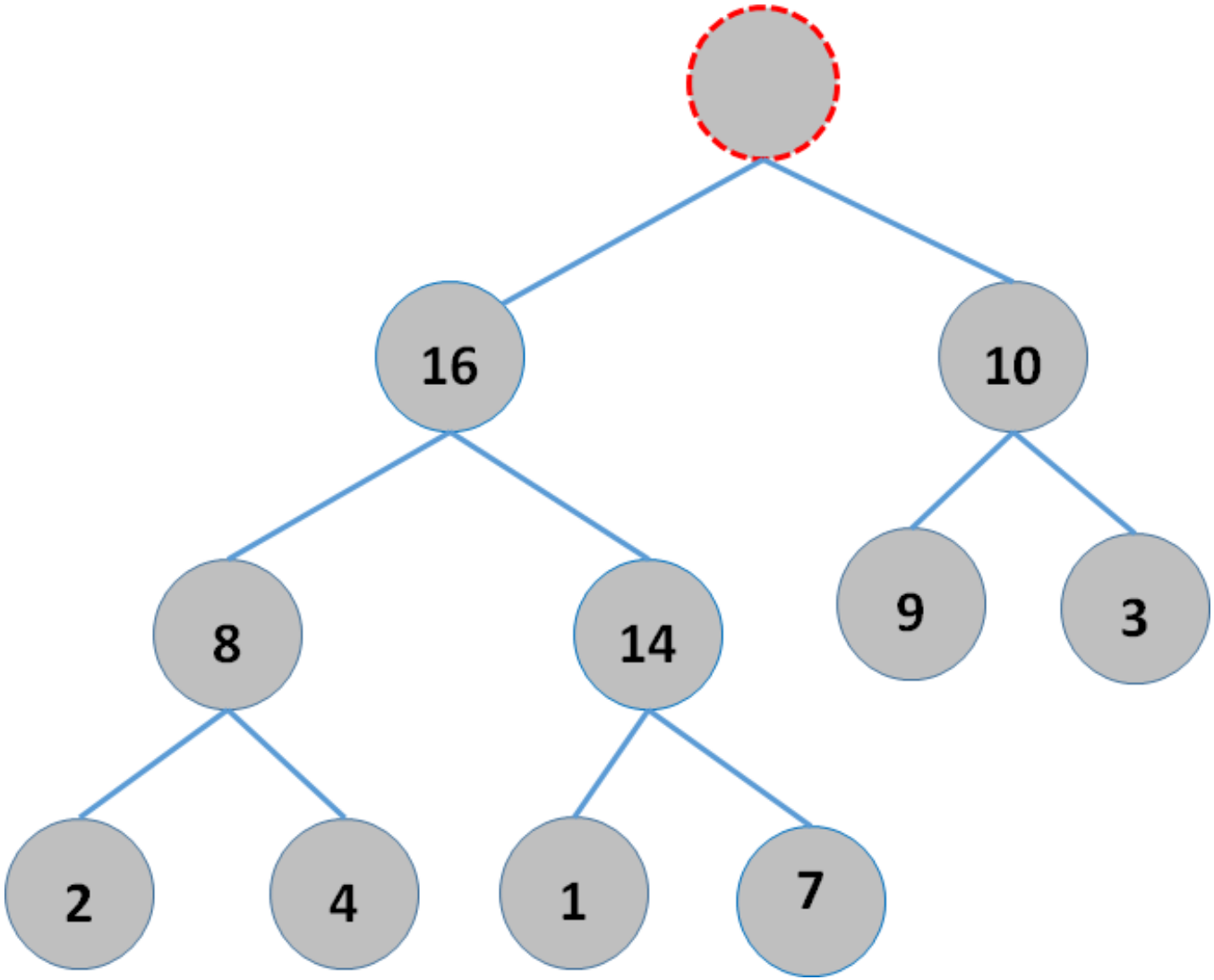
# Heap : DeleteMax

- Delete the maximum element (root)
- Fix the heap

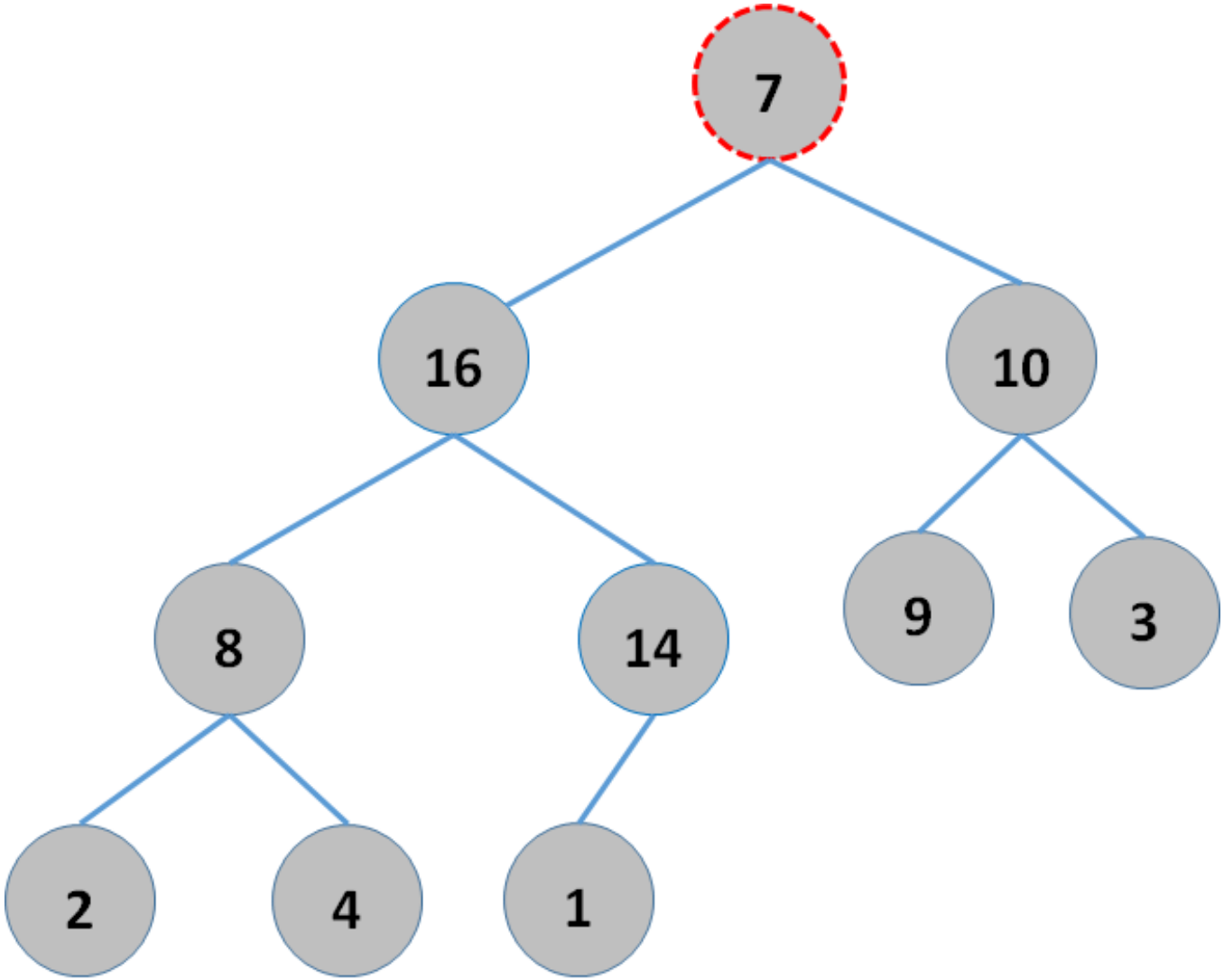
# Heap : DeleteMax



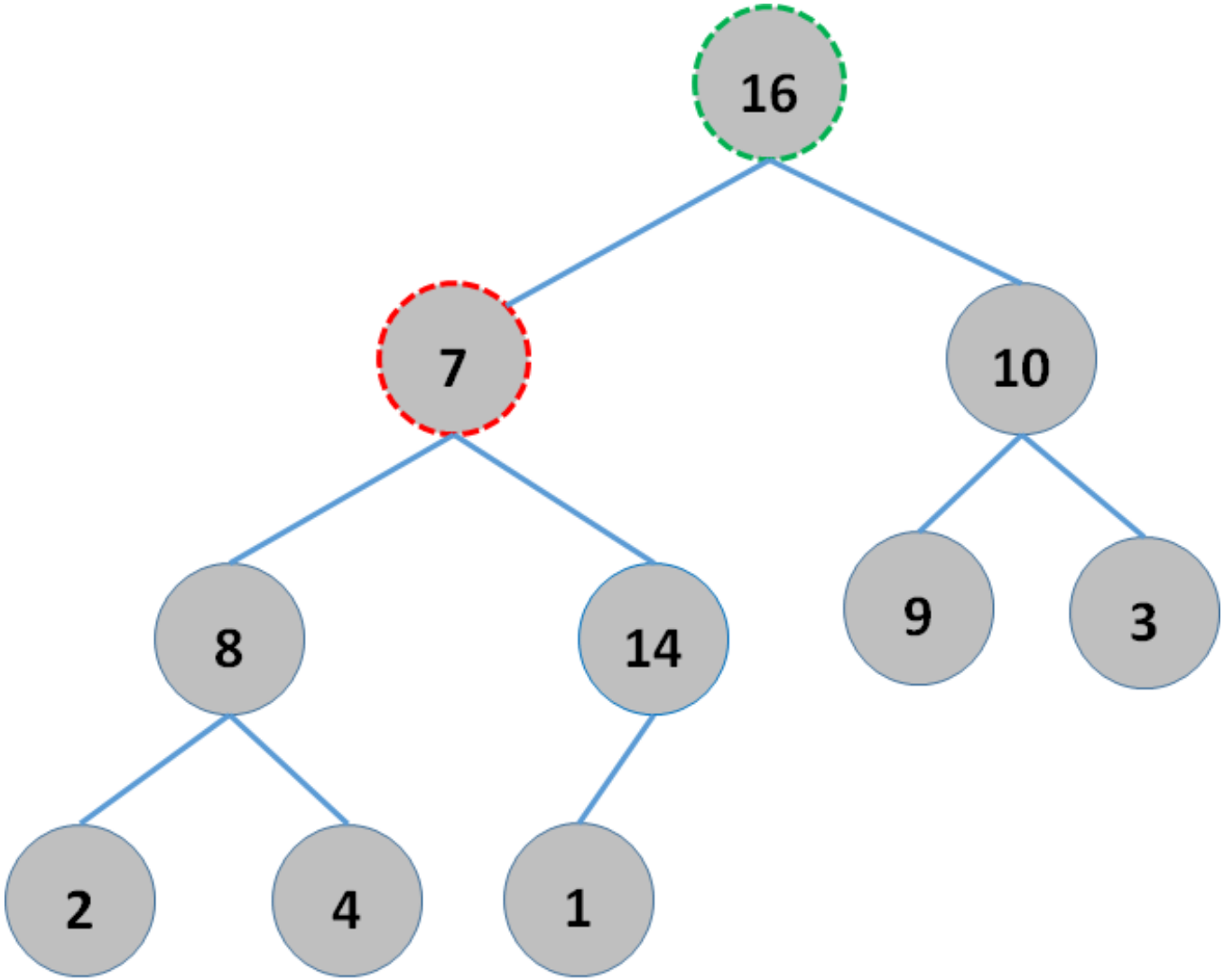
# Heap : DeleteMax



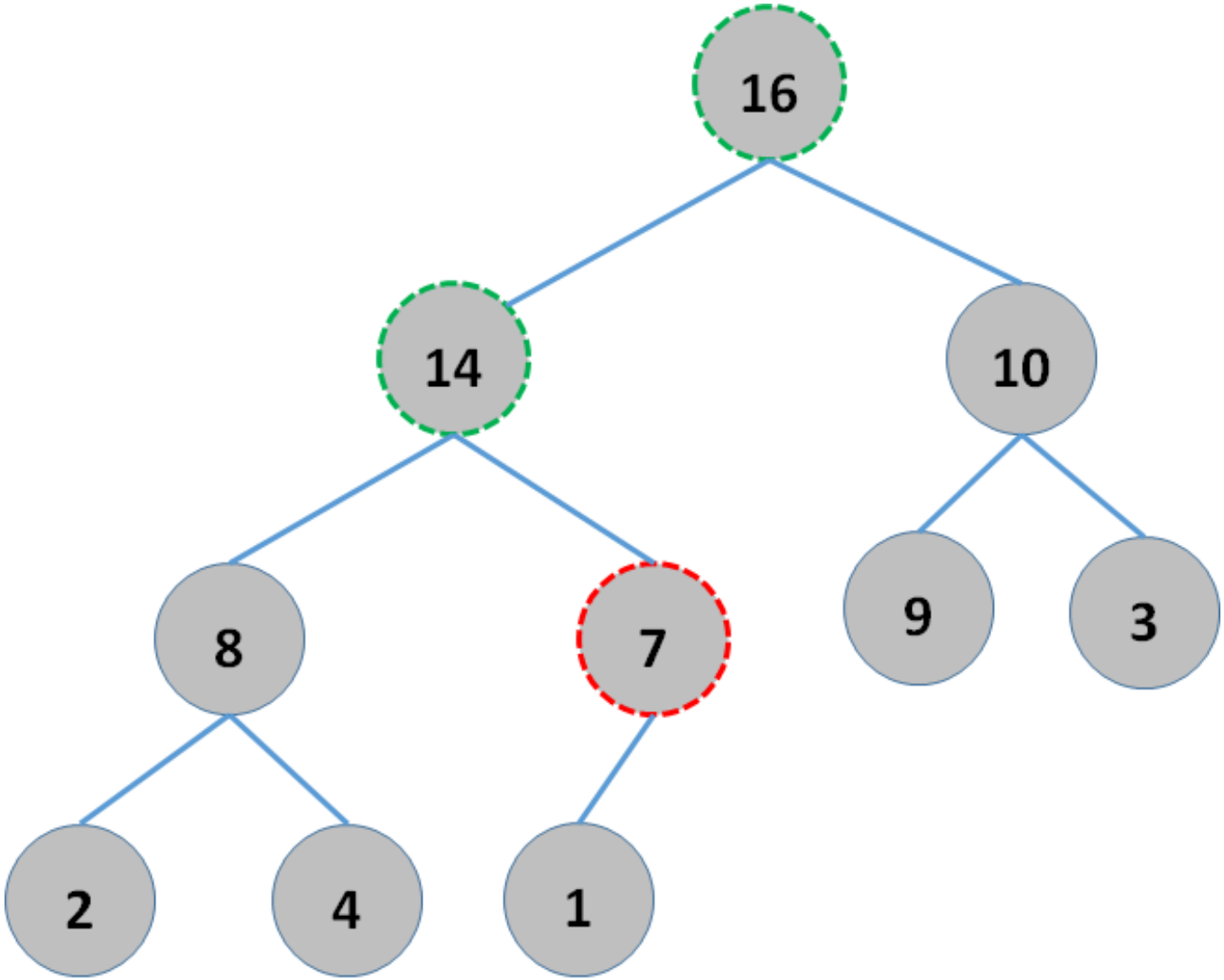
# Heap : DeleteMax



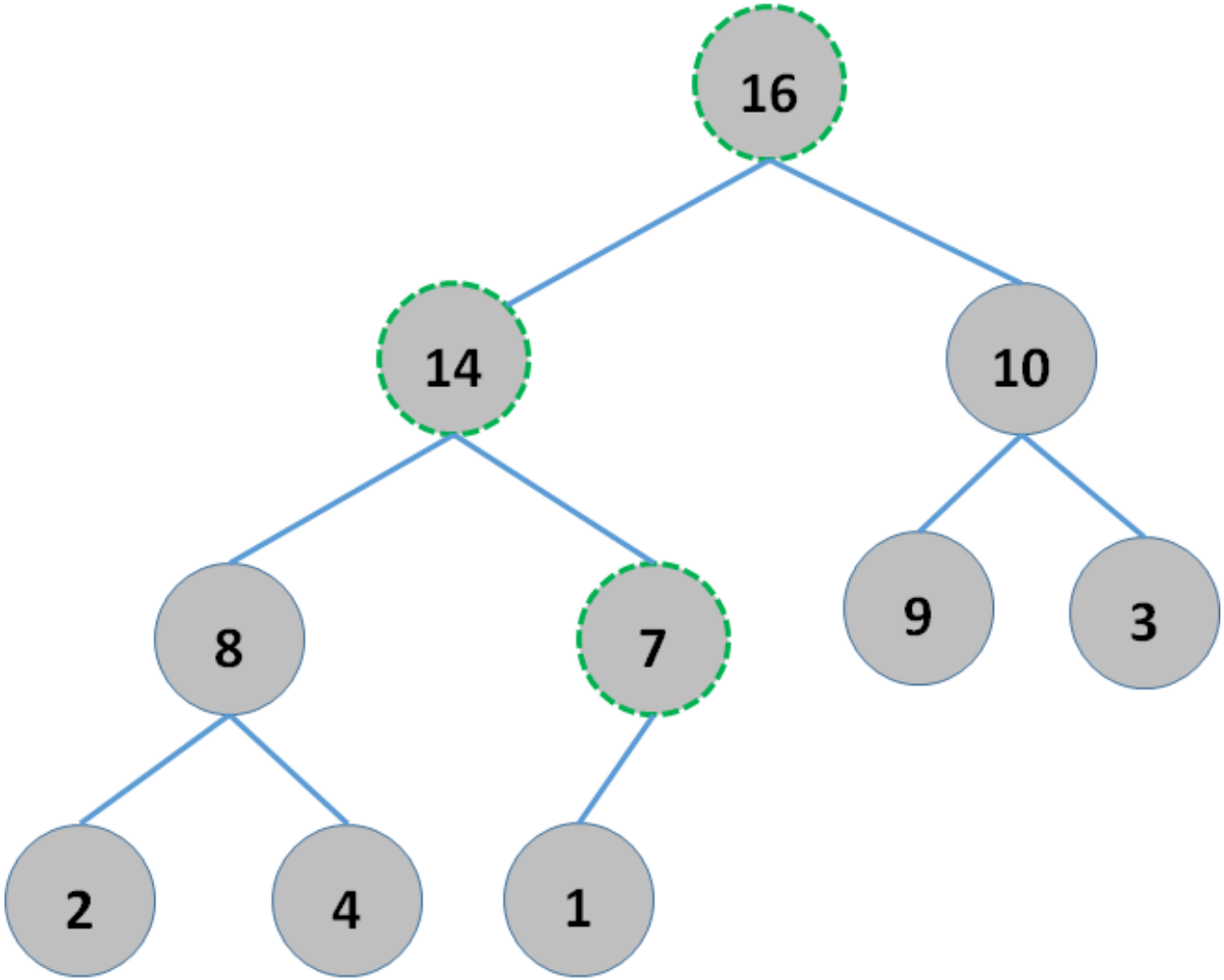
# Heap : DeleteMax



# Heap : DeleteMax



# Heap : DeleteMax



# Heap : DeleteMax

- Remove root
- Replace root with last element (does not affect Completeness property)
- Fix heap violations by bubbling it down till it is fixed
- Complexity:

# Heap : DeleteMax

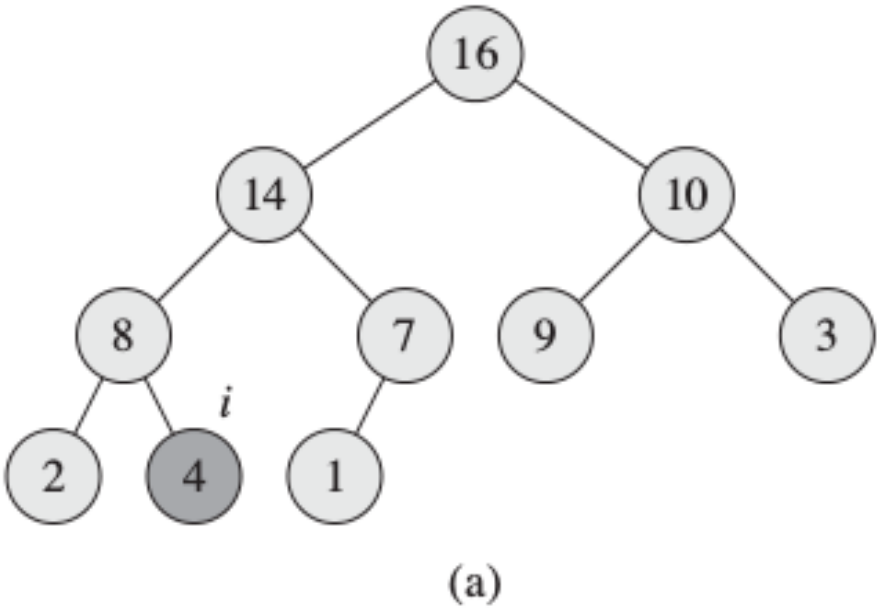
- Remove root
- Replace root with last element (does not affect Completeness property)
- Fix heap violations by bubbling it down till it is fixed
- Complexity: $O(\lg n)$

# Heap : IncreaseKey

- Given a node, increase its priority to a new, higher value
- Fix heap property violations

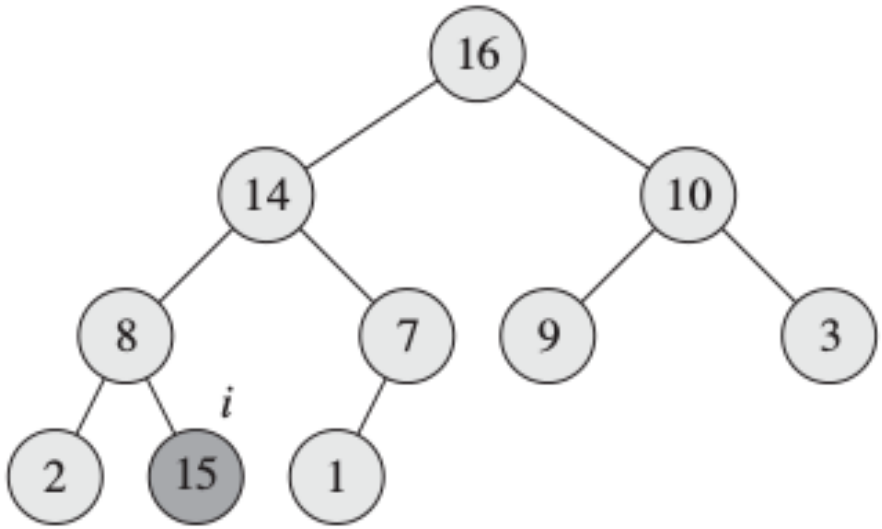
# Heap : IncreaseKey[9]

IncreaseKey: Increase value of 4 to 15




[9] CLRS Fig 6.5

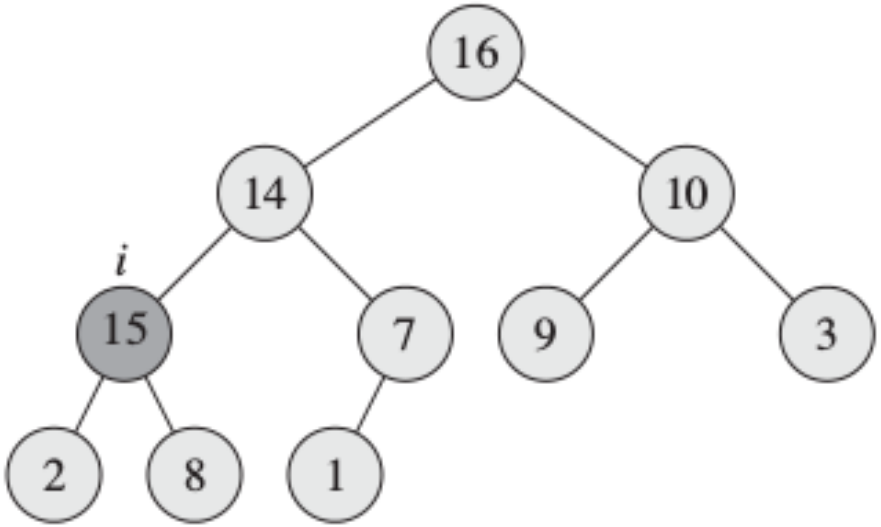
# Heap : IncreaseKey[10]




(b)

[10] CLRS Fig 6.5

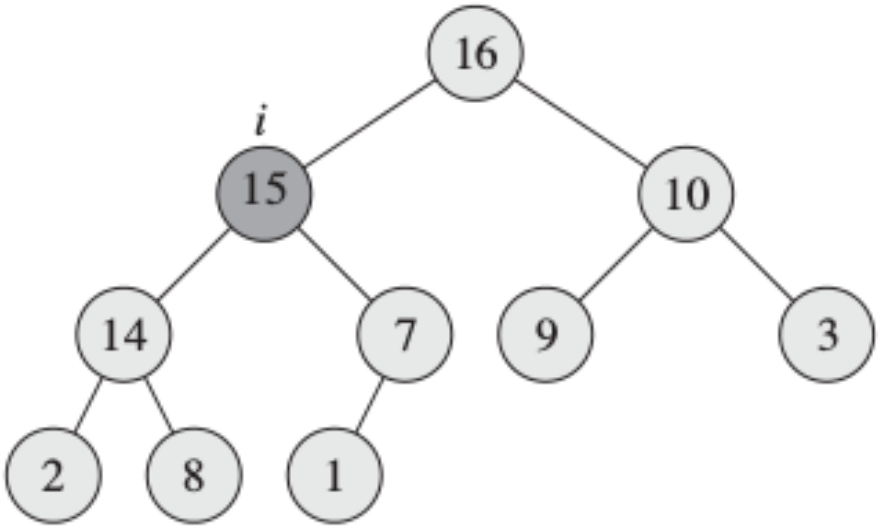
# Heap : IncreaseKey[11]




(c)

[11] CLRS Fig 6.5

# Heap : IncreaseKey[12]




(d)

[12] CLRS Fig 6.5

# Heap : IncreaseKey

- Update element
- Fix heap violations by bubbling it up till it is fixed
- Complexity:

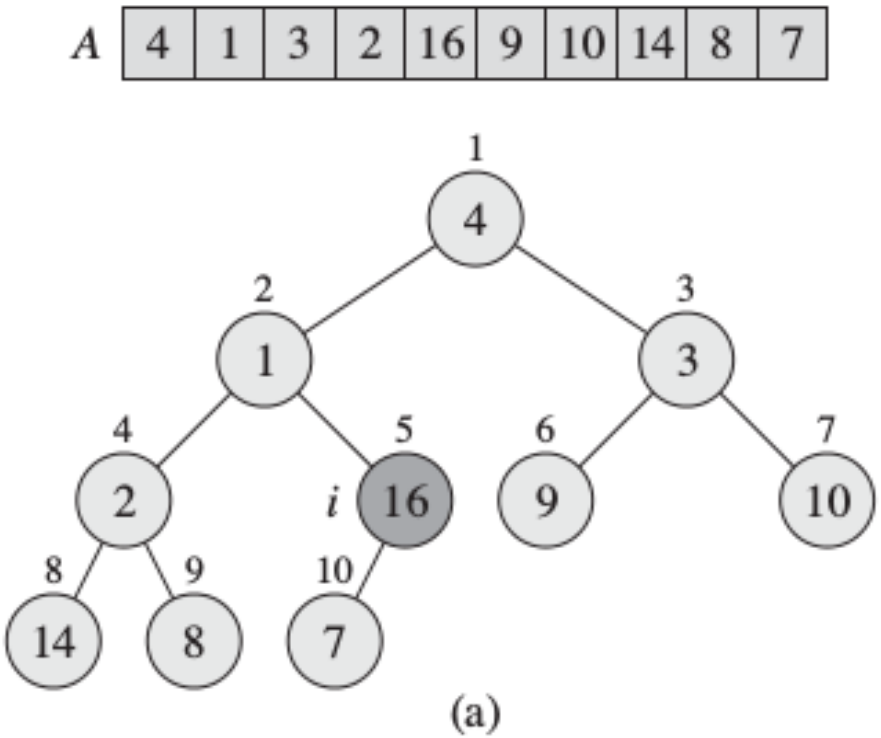
# Heap : IncreaseKey

- Update element
- Fix heap violations by bubbling it up till it is fixed
- Complexity: $O(\lg n)$

# Build-Max-Heap

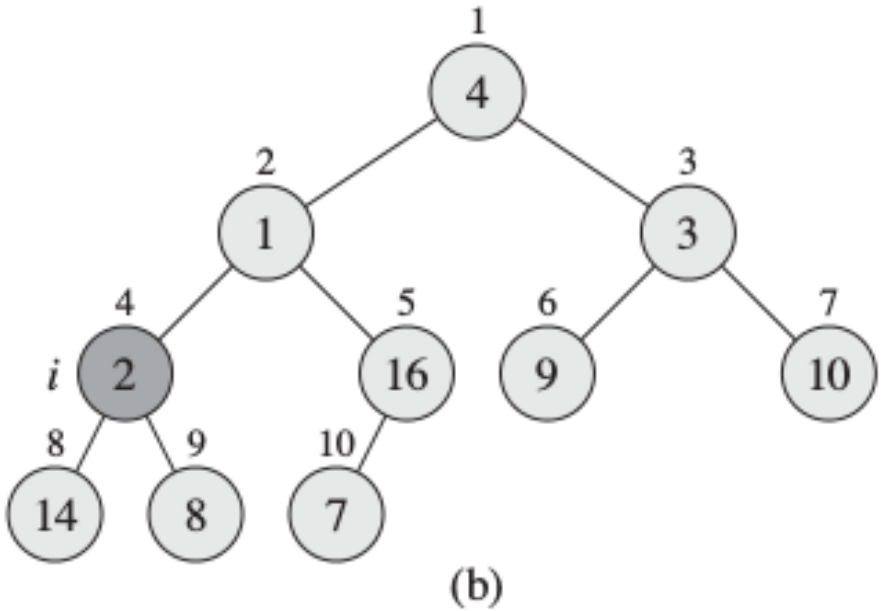
- Given an array $A$, convert it to a max-heap
- $A.length$: Length of the array
- $A.heapSize$: Elements from 1 ... A.heapSize form a heap
- Build-Max-Heap($A$):
  - A.heapSize = A.length
  - for $i = \lfloor A.length/2 \rfloor$ down to 1
    Max-Heapify($A, i$)
- **Analysis:** $O(n)$ (See book for details)

# Build-Max-Heap : Example [13]



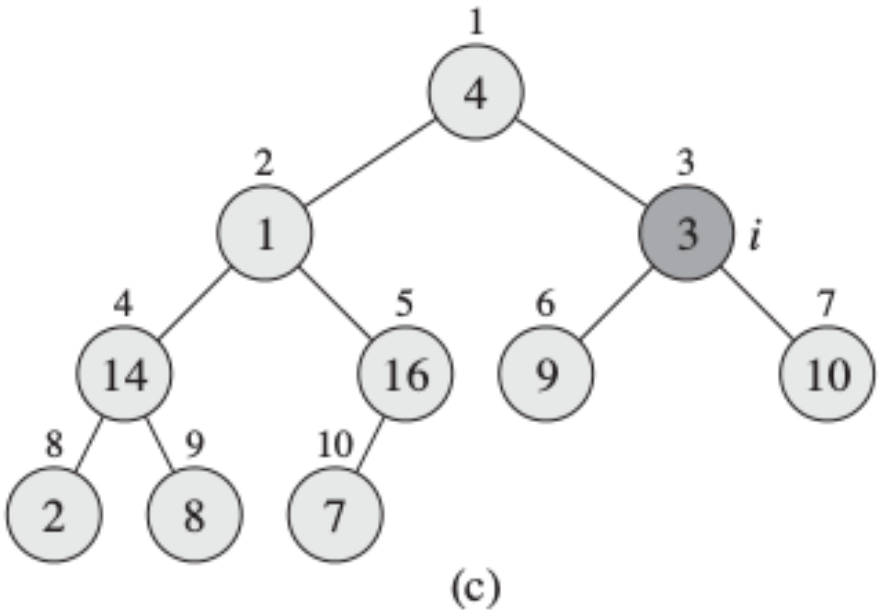

[13] CLRS Fig 6.3

# Build-Max-Heap : Example [14]



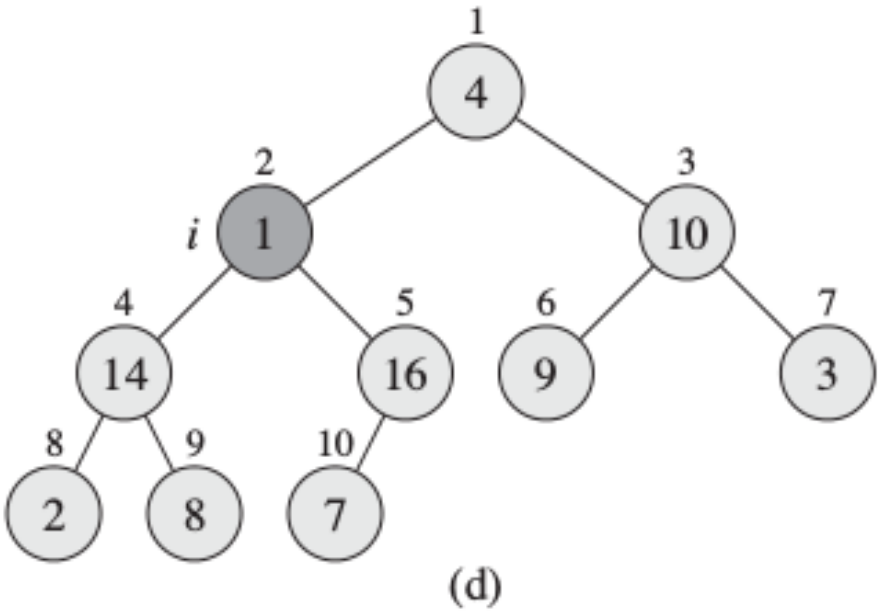

[14] CLRS Fig 6.3

# Build-Max-Heap : Example [15]



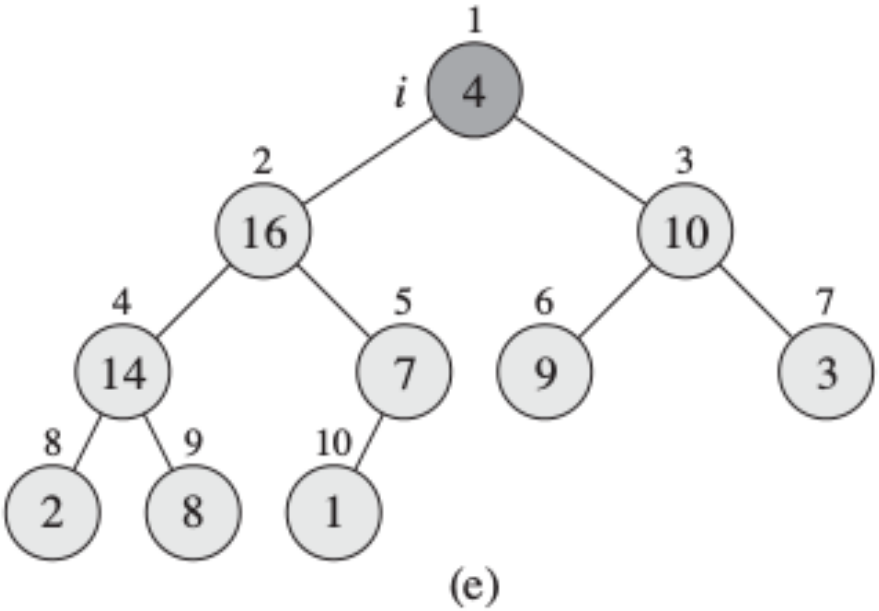

[15] CLRS Fig 6.3

# Build-Max-Heap : Example [16]



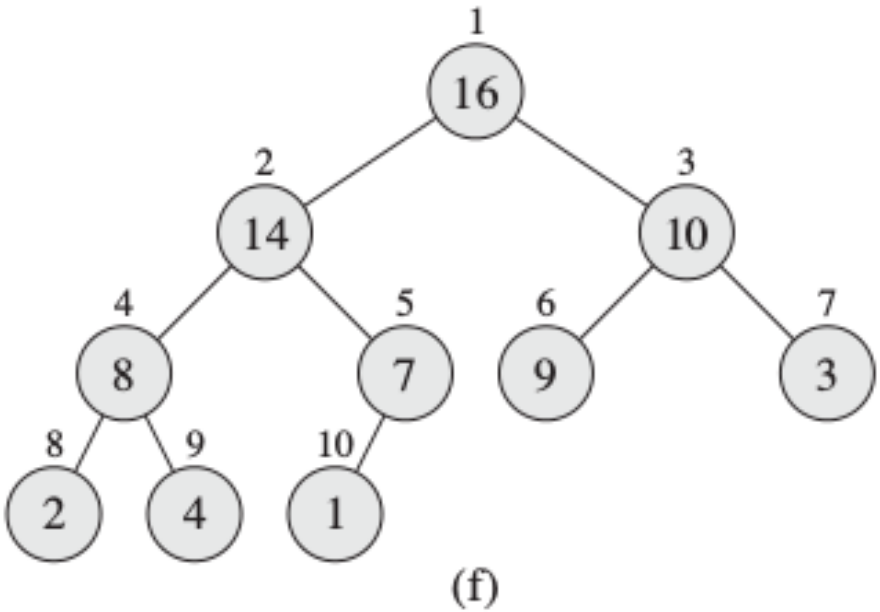

[16] CLRS Fig 6.3

# Build-Max-Heap : Example [17]




[17] CLRS Fig 6.3

# Build-Max-Heap : Example [18]



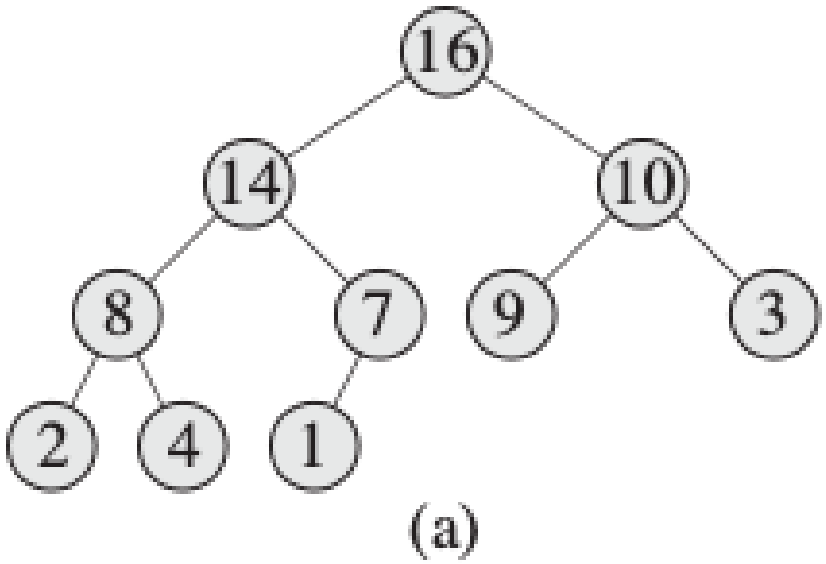
(f)

[18] CLRS Fig 6.3

# HeapSort

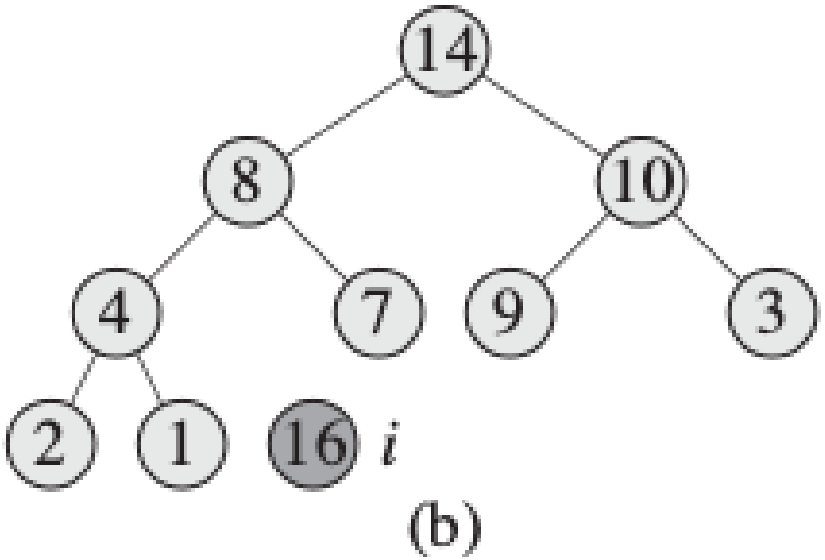
```
HeapSort(A):
    Build-Max-Heap(A)
    for i = A.length down to 2
        Exchange A[1] with A[i]
        A.heapSize = A.heapSize - 1
        Max-Heapify(A, 1)
```

# Heap Sort: Example[19]



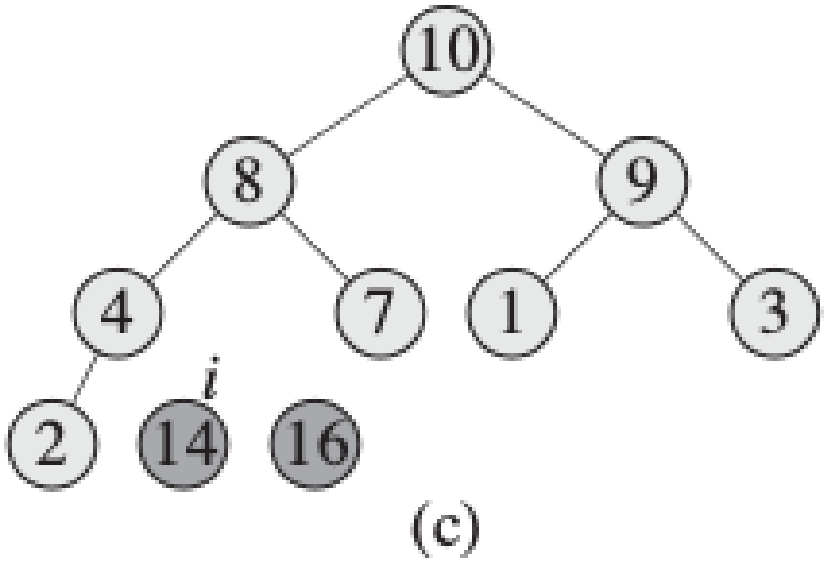

[19] CLRS Fig 6.4

# Heap Sort: Example[20]



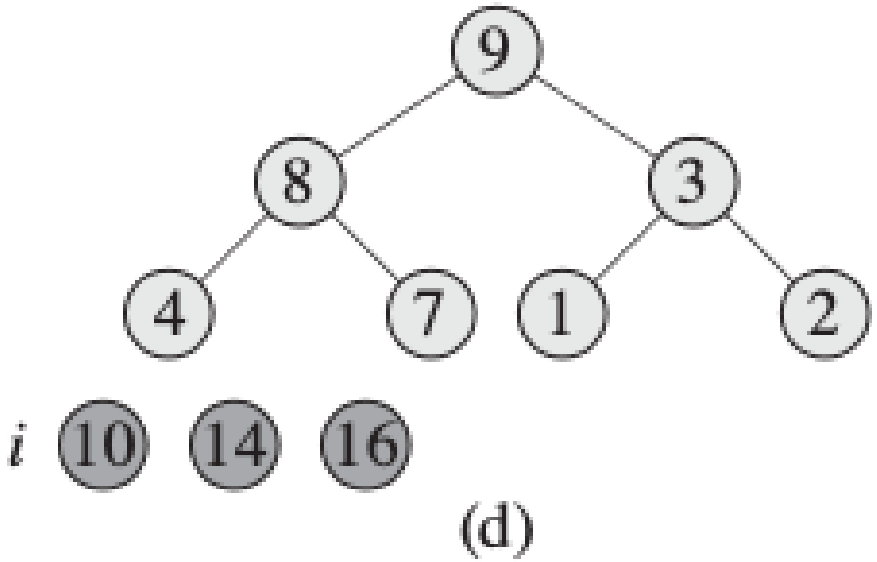

[20] CLRS Fig 6.4

# Heap Sort: Example[21]



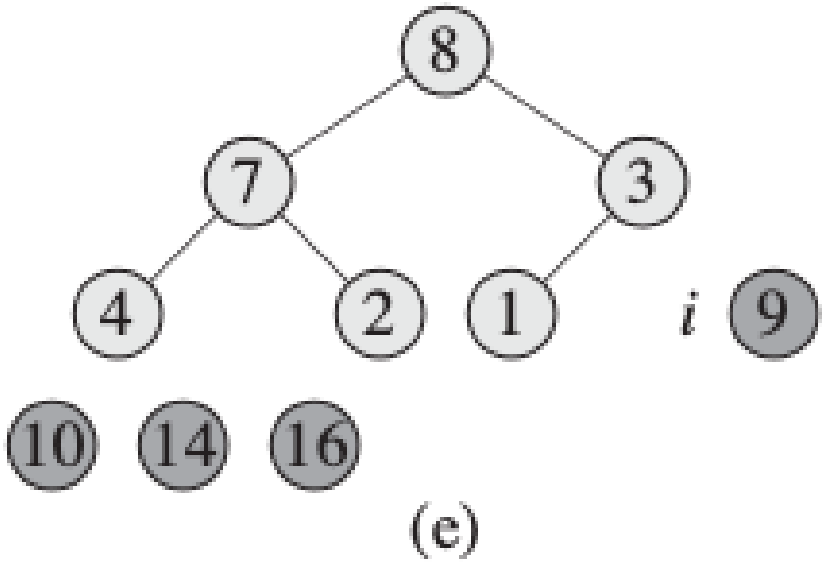

[21] CLRS Fig 6.4

# Heap Sort: Example[22]



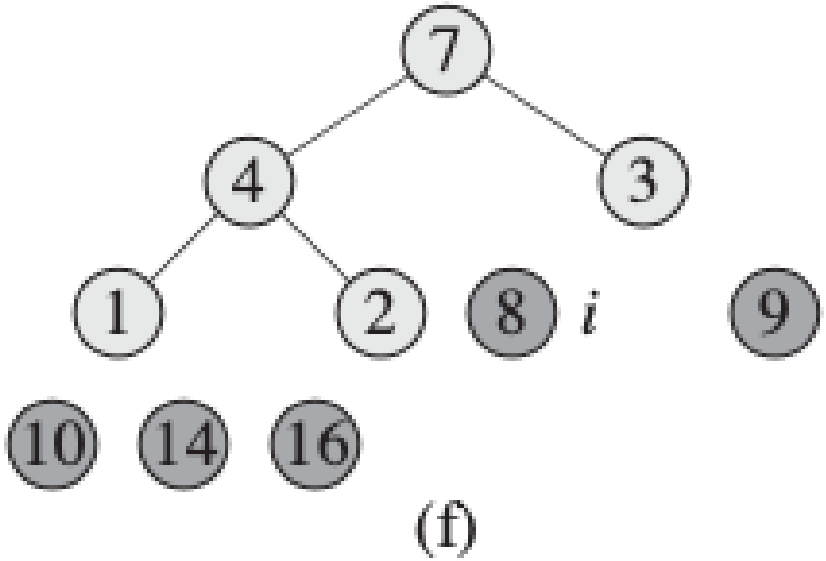

[22] CLRS Fig 6.4

# Heap Sort: Example[23]



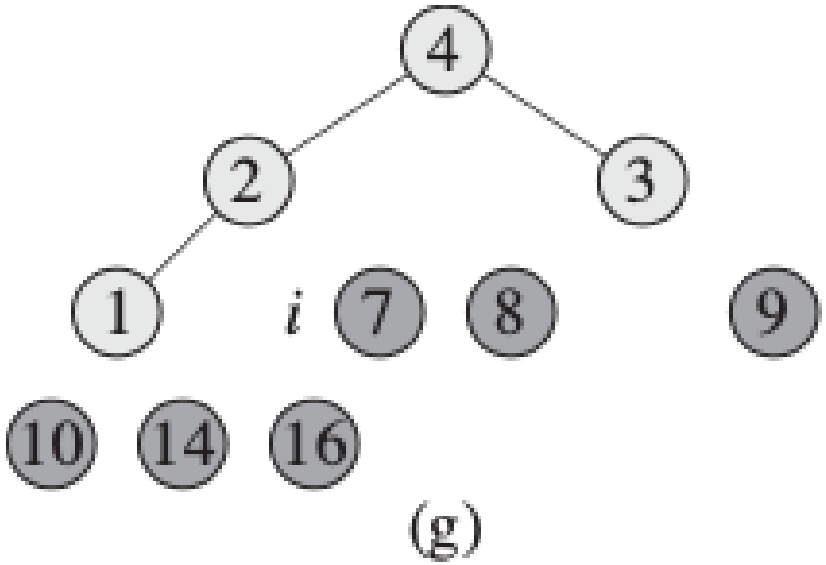

[23] CLRS Fig 6.4

# Heap Sort: Example[24]



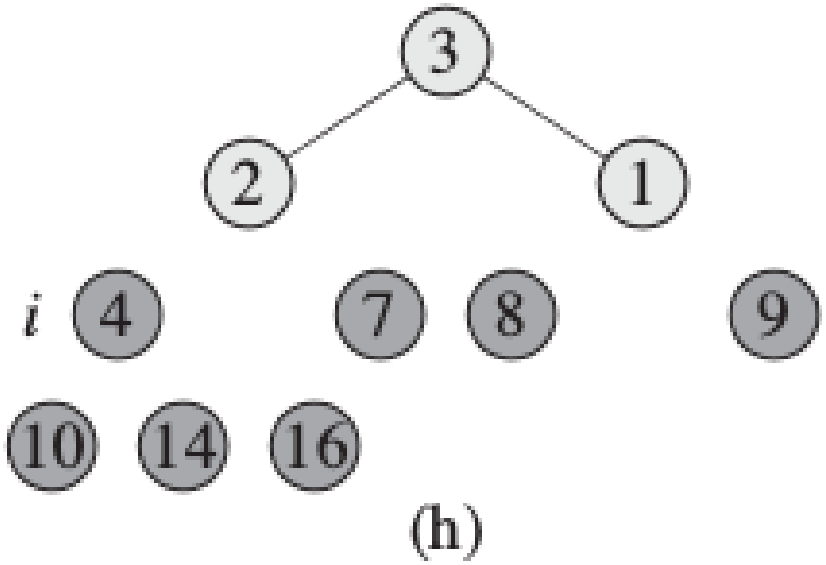

[24] CLRS Fig 6.4

# Heap Sort: Example[25]



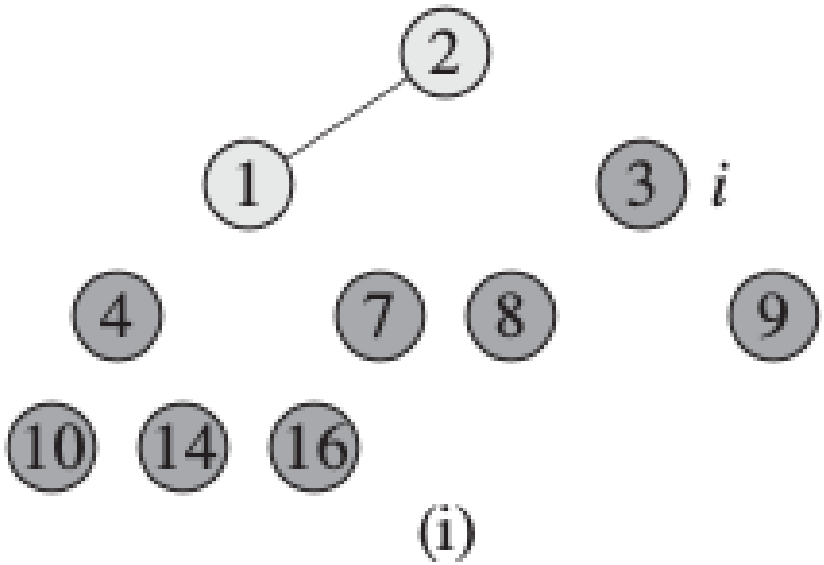

[25] CLRS Fig 6.4

# Heap Sort: Example[26]



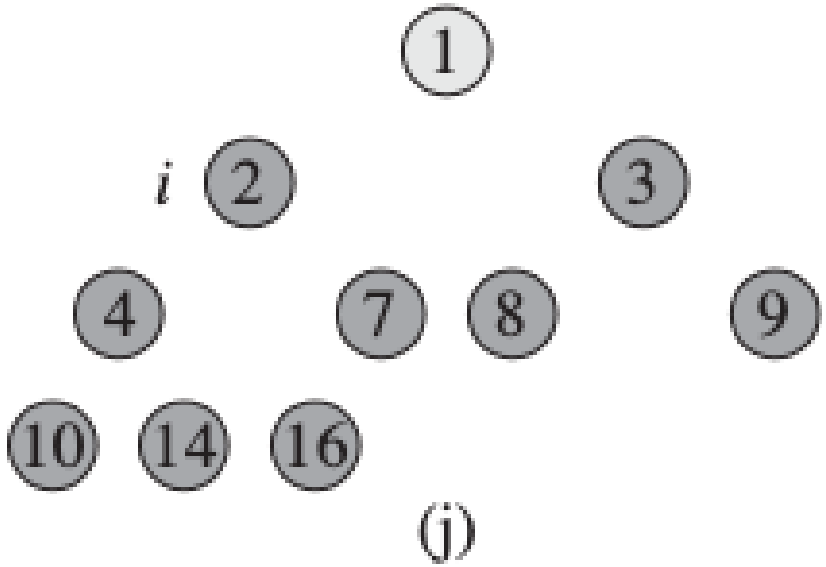

[26] CLRS Fig 6.4

# Heap Sort: Example[27]




[27] CLRS Fig 6.4

# Heap Sort: Example[28]




[28] CLRS Fig 6.4

# Heap Sort: Example[29]


[29] CLRS Fig 6.4

# HeapSort: Analysis

- Operations:
  - Build-Max-Heap:

# HeapSort: Analysis

- Operations:
  - Build-Max-Heap: $O(n)$
  - $n$ Max-Heapify:

# HeapSort: Analysis

- Operations:
  - Build-Max-Heap: $O(n)$
  - $n$ Max-Heapify: $n \times \lg n = O(n \lg n)$
  - Complexity: $O(n) + O(n \lg n) = O(n \lg n)$

# HeapSort

- Very efficient in practice - often competitive with QuickSort
- In-Place but not stable (why?)
- Requires constant extra space
- Best, average and worst case complexity is $O(n \lg n)$ (unlike Quicksort)

# Data Structures for Disjoint Sets

# Disjoint Sets ADT

- Objective: Represent and manipulate disjoint sets (sets that do not overlap)
- Required Operations
  - MakeSet($x$): Create a new set $\{x\}$ with single element $x$
  - Find($x$): Find the set containing $x$
  - Union($x, y$): Merge sets containing $x$ and $y$

# Disjoint Sets: Example[30]

- Objects.

```
0  1  2  3  4  5  6  7  8  9
```

- Disjoint sets of objects.

```
0  1  { 2 3 9 }  { 5 6 }  7  { 4 8 }
```

- Find query: are objects 2 and 9 in the same set?

```
0  1  { 2 3 9 }  { 5 6 }  7  { 4 8 }
```

- Union command: merge sets containing 3 and 8.

```
0  1  { 2 3 4 8 9 }  { 5 6 }  7
```

[30] https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf

# Disjoint Sets: Applications[31]

- Network connectivity: are two computers connected?
- Compilers: are two variables aliases?
- Image segmentation: are both pixels in same segment?
- Chip design: are two transistors connected to each other?
- Maze design
- Speeding up Kruskal's MST algorithm
- Many many more

[31] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/18-disjoint-union-find.pdf

# Disjoint Sets: Naming

- Represent each disjoint set by a unique name
- For convenience, the name is one of its elements
- This element is called the **leader** of the set
- Find($x$) returns the leader of set containing $x$
- Typically, Union takes leaders as input. For eg, Union($a, b$). If not easily fixable by Union(Find($a$), Find($b$))

# Data Structures for Disjoint Sets

**Objective:**

- Design an efficient data structure to represent DS ADT
- Assume that there are $N$ elements - represented by $1, \ldots, N$
- $M$ operations (any mixture of union and find)

**Candidate Representations:**

- Array based
- Linked List based
- Tree based

# Disjoint Sets Implementation: Arrays

**Idea:**

- Maintain an array $A$ with $N$ elements
- $A[i]$ stores the leader for set containing element $i$

**Find:** Find(3)=4, Find(6)=8

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A[i] | 1 | 2 | 4 | 4 | 5 | 8 | 5 | 8 | 8 | 5 |

**Union:** Union$(4, 8)$

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A[i] | 1 | 2 | 8 | 8 | 5 | 8 | 5 | 8 | 8 | 5 |

# Disjoint Sets Implementation: Arrays

**Analysis:**

- Find:

# Disjoint Sets Implementation: Arrays

**Analysis:**

- Find: $O(1)$
- Union:

# Disjoint Sets Implementation: Arrays

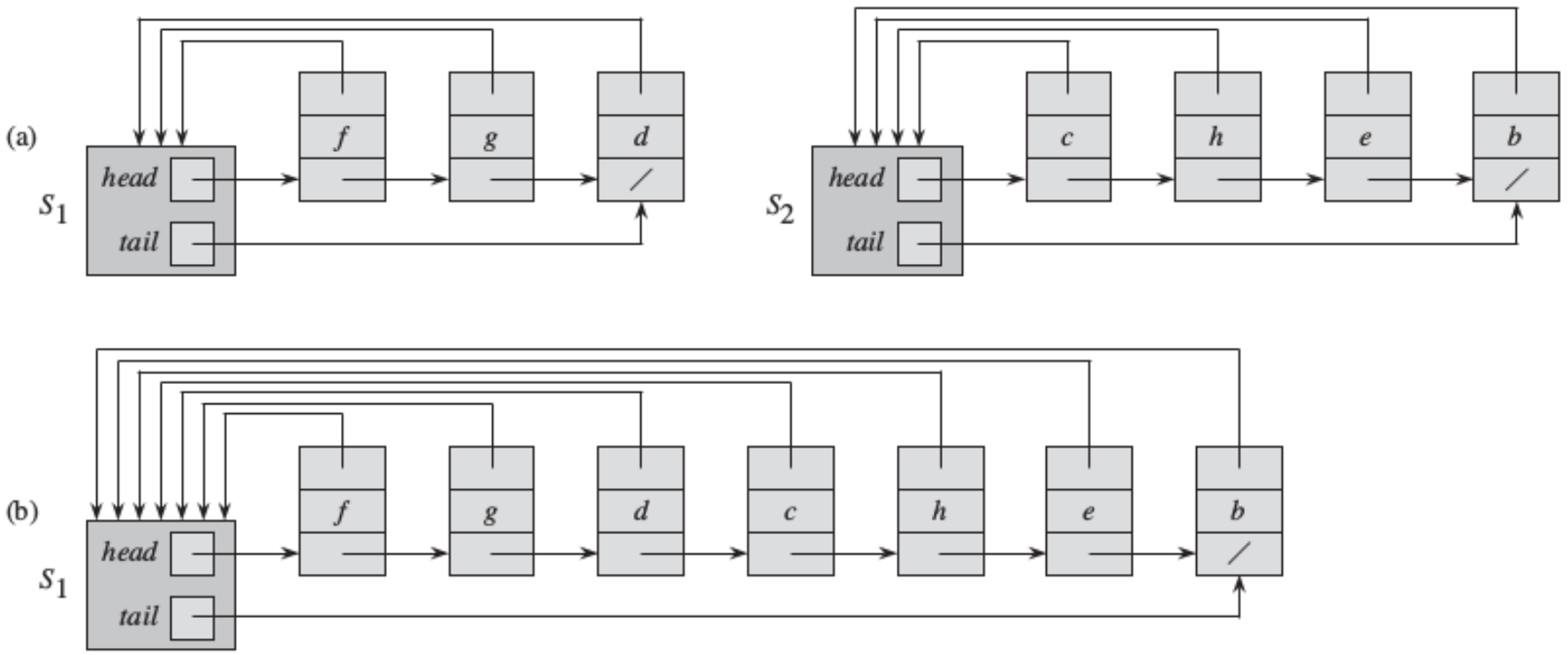
**Analysis:**

- Find: $O(1)$
- Union: $O(N)$
- Complexity for $M$ operations: $O(MN)$

# Disjoint Sets Implementation: Linked List[32]

**Idea:**

- Represent each set as a linked list
- Set first element of each linked list as the leader




[32] CLRS Fig 21.2

# Disjoint Sets Implementation: Linked List

**Analysis:**

- Find:

# Disjoint Sets Implementation: Linked List

**Analysis:**

- Find: $O(1)$
- Union:

# Disjoint Sets Implementation: Linked List

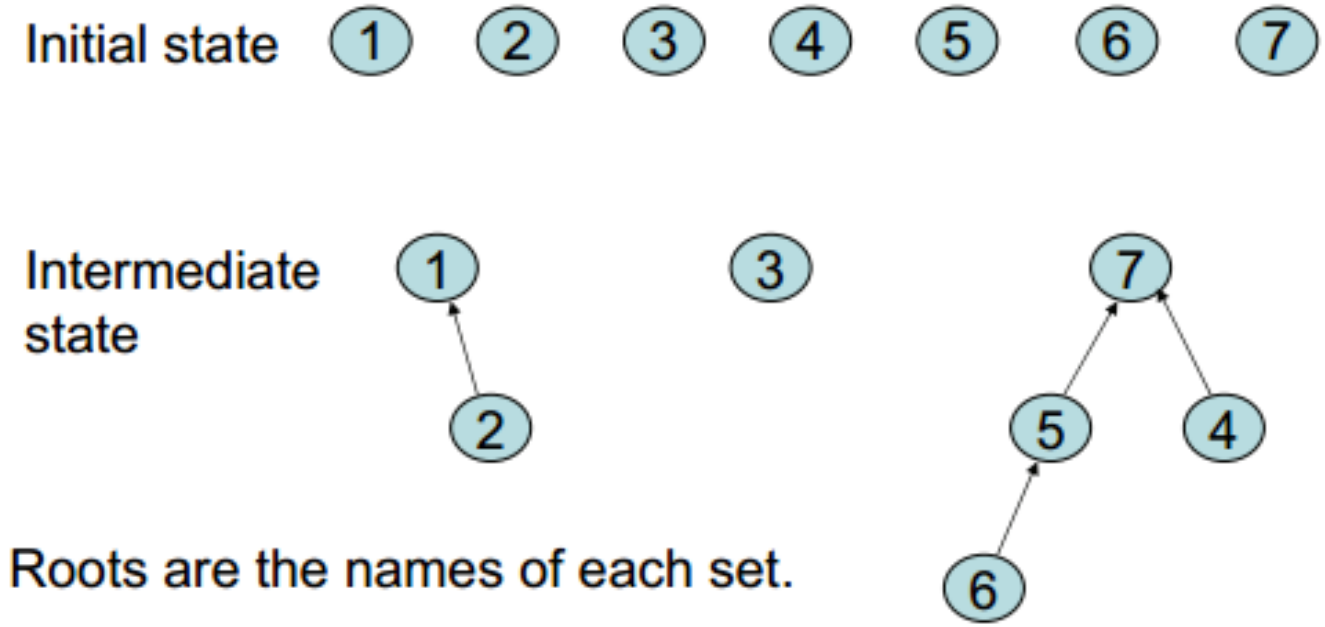
**Analysis:**

- Find: $O(1)$
- Union: $O(N)$
- Complexity for $M$ operations: $O(MN)$

# Disjoint Sets Implementation: Tree

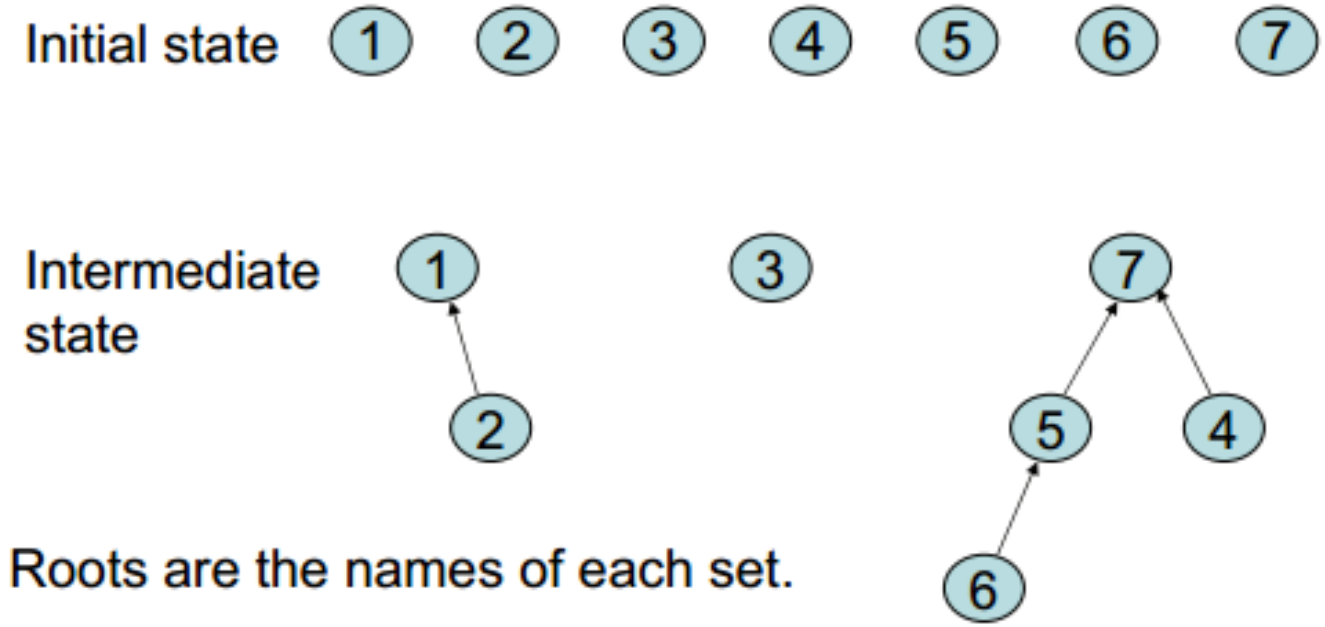
- Also called as Union-Find data structure
- Idea: Represent each set as a tree
- Store all sets as a forest (collection of disconnected trees)
- Allow each node to have arbitrary number of children
- Root of each tree is the leader

# Union-Find: Up-Tree Representation[33]



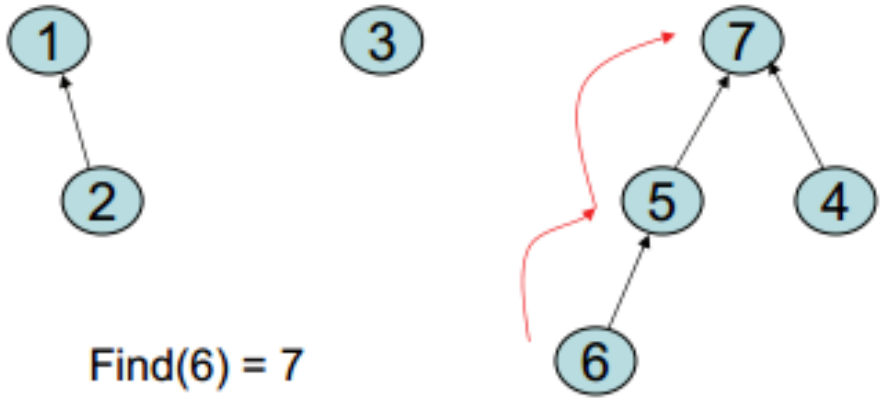

[33] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Up-Tree Representation [34]



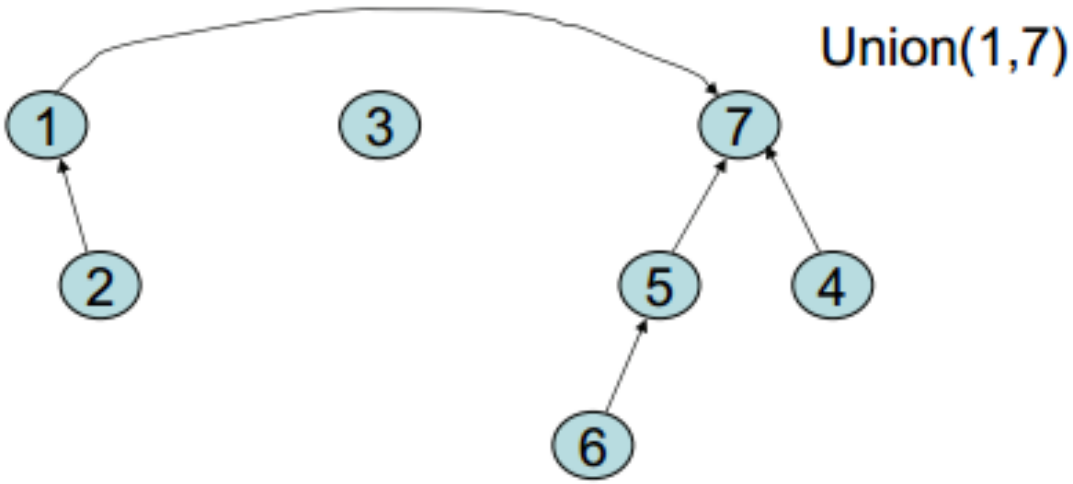

[34] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Find Operation [35]




[35] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Union Operation [36]



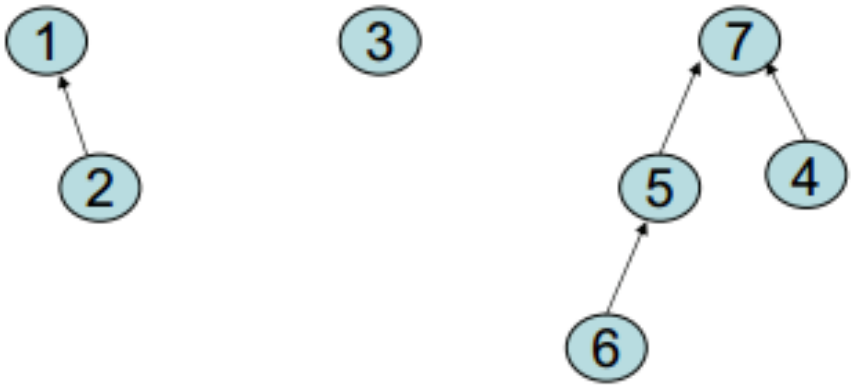

11

[36] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Up-Tree Implementation [37]

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| up | -1 | 1 | -1 | 7 | 7 | 5 | -1 |

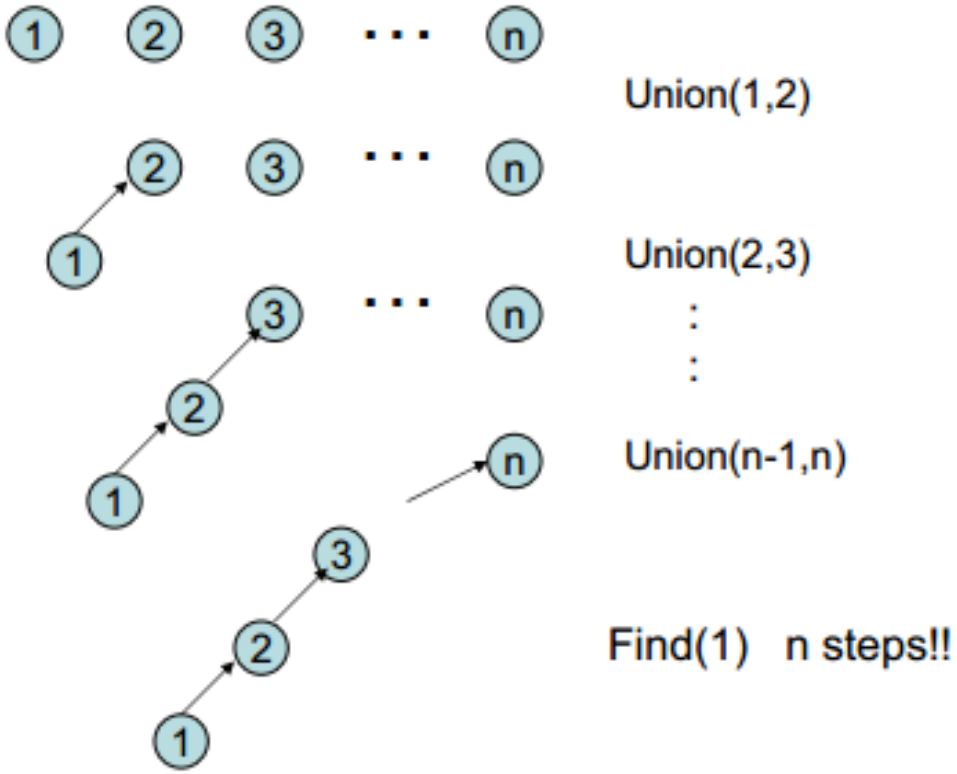
Up[x] = -1 means
x is a root.




12

[37] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Worst Case[38]




[38] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

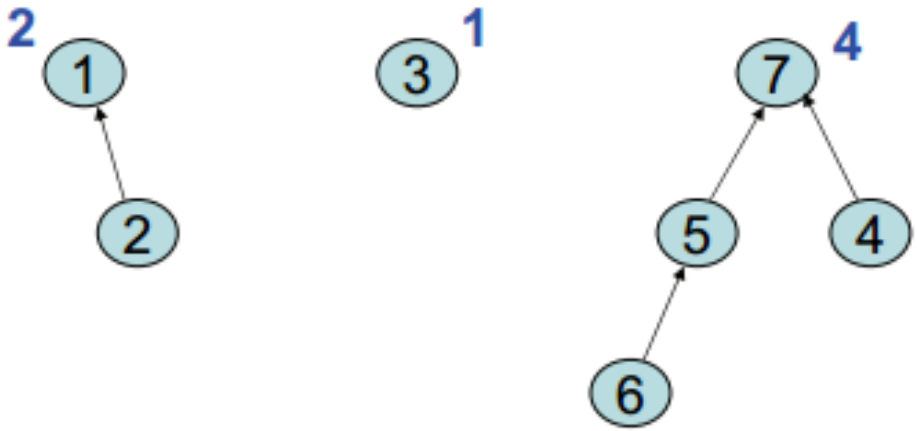
# Union-Find: Candidate Improvements

# Union-Find: Candidate Improvements

- Improve Union
  - Union by Size
  - Union by Rank (depth)
- Improve Find

# Union-Find: Improved Union[39]

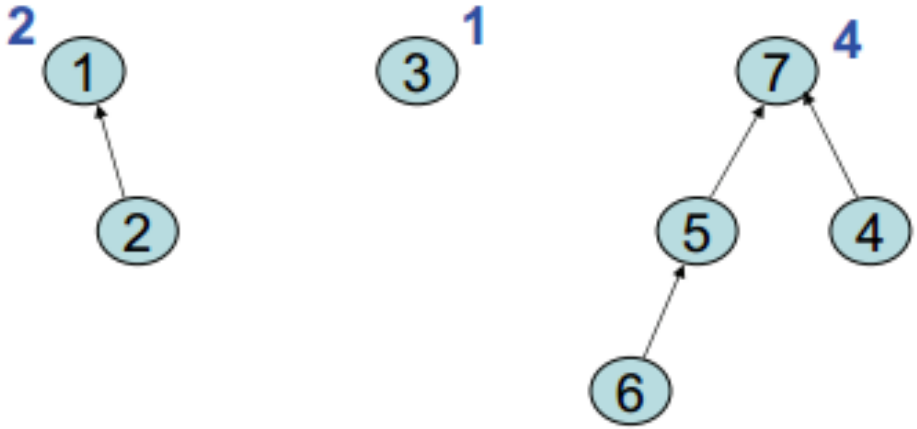
**Union by Size (Weighted Union):** Always point smaller tree to root of larger tree. Break ties arbitrarily.



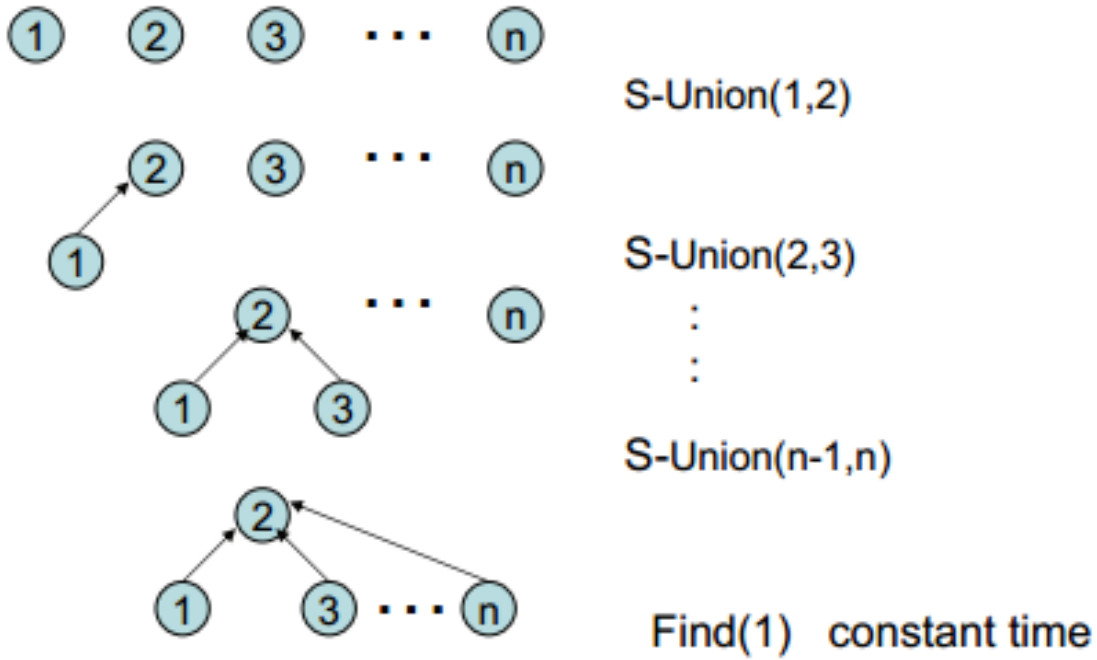

[39] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Improved Union

**Union by Rank:** Always point tree with smaller rank (depth) to root of larger tree. Break ties arbitrarily.



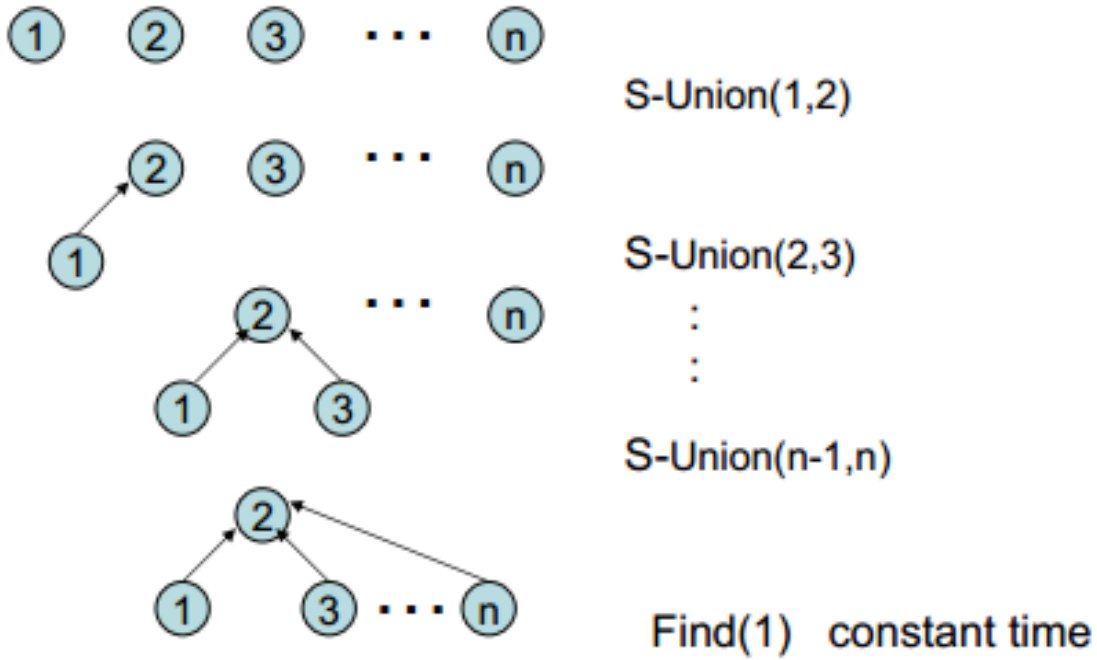
# Union by Size: Best Case[40]




**Complexity:**

[40] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

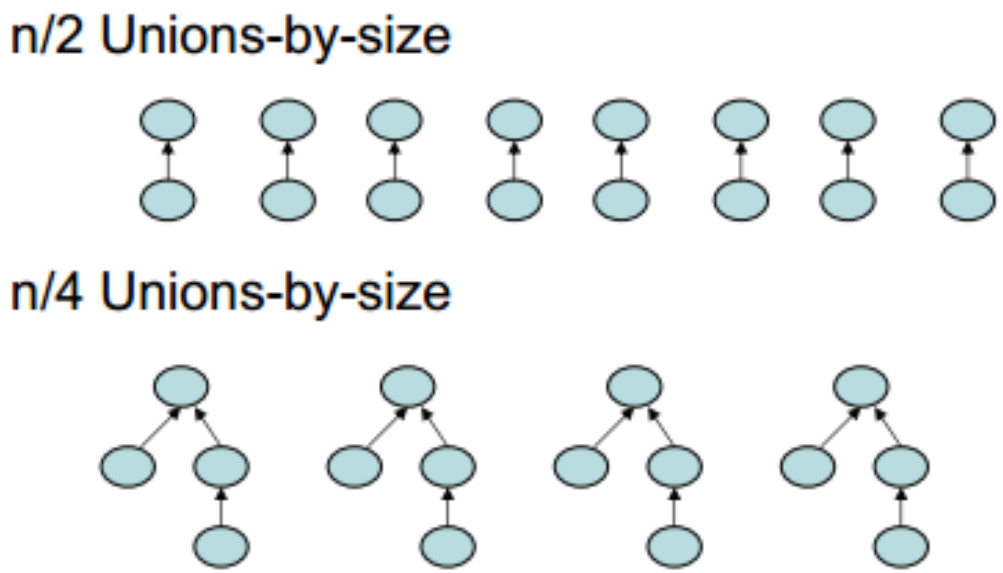
# Union by Size: Best Case[40]




**Complexity:** Union and Find take $O(1)$ time

[40] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

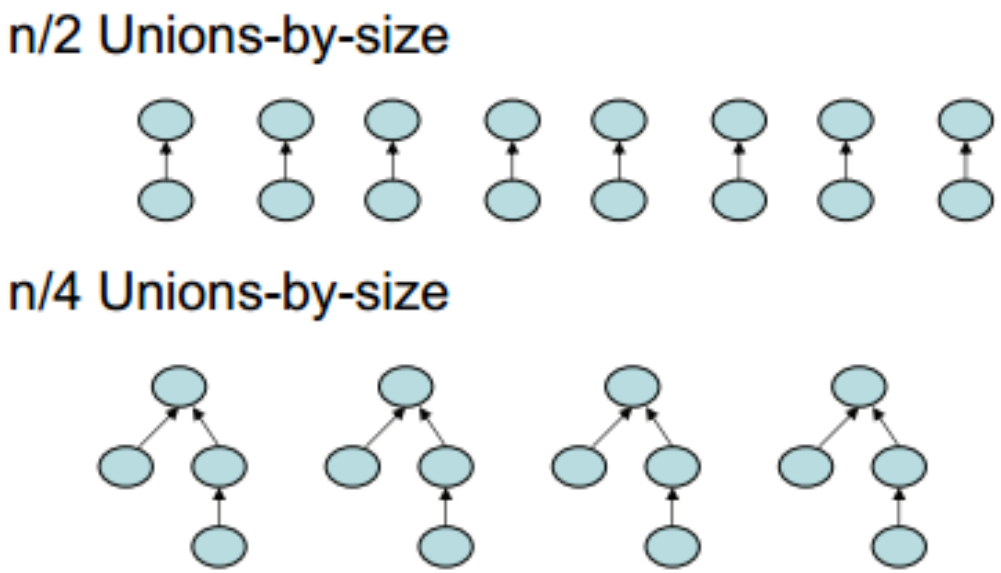
# Union by Size: Worst Case[41]




**Complexity:**

[41] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

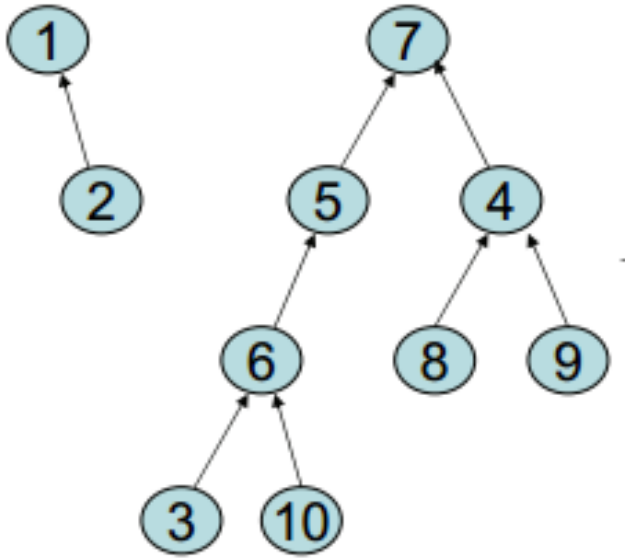
# Union by Size: Worst Case[41]




**Complexity:** Union takes $O(1)$ and Find take $O(\lg n)$ time

[41] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

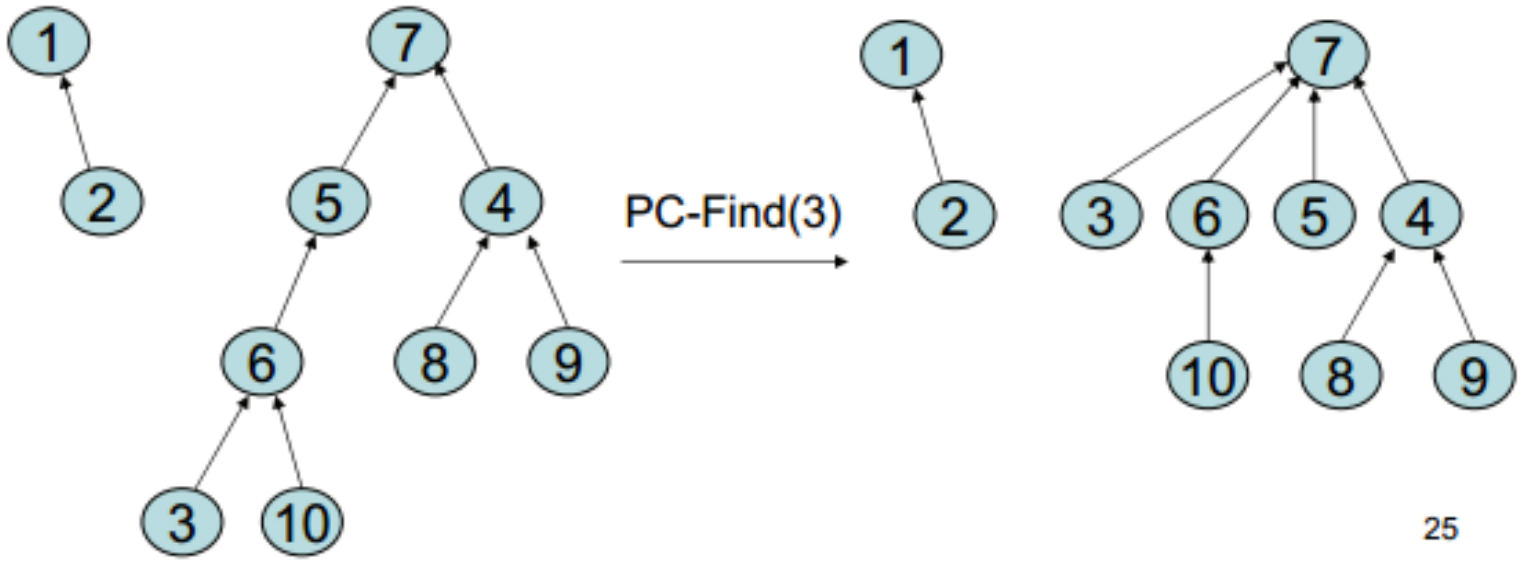
# Union-Find: Improved Find[42]




[42] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

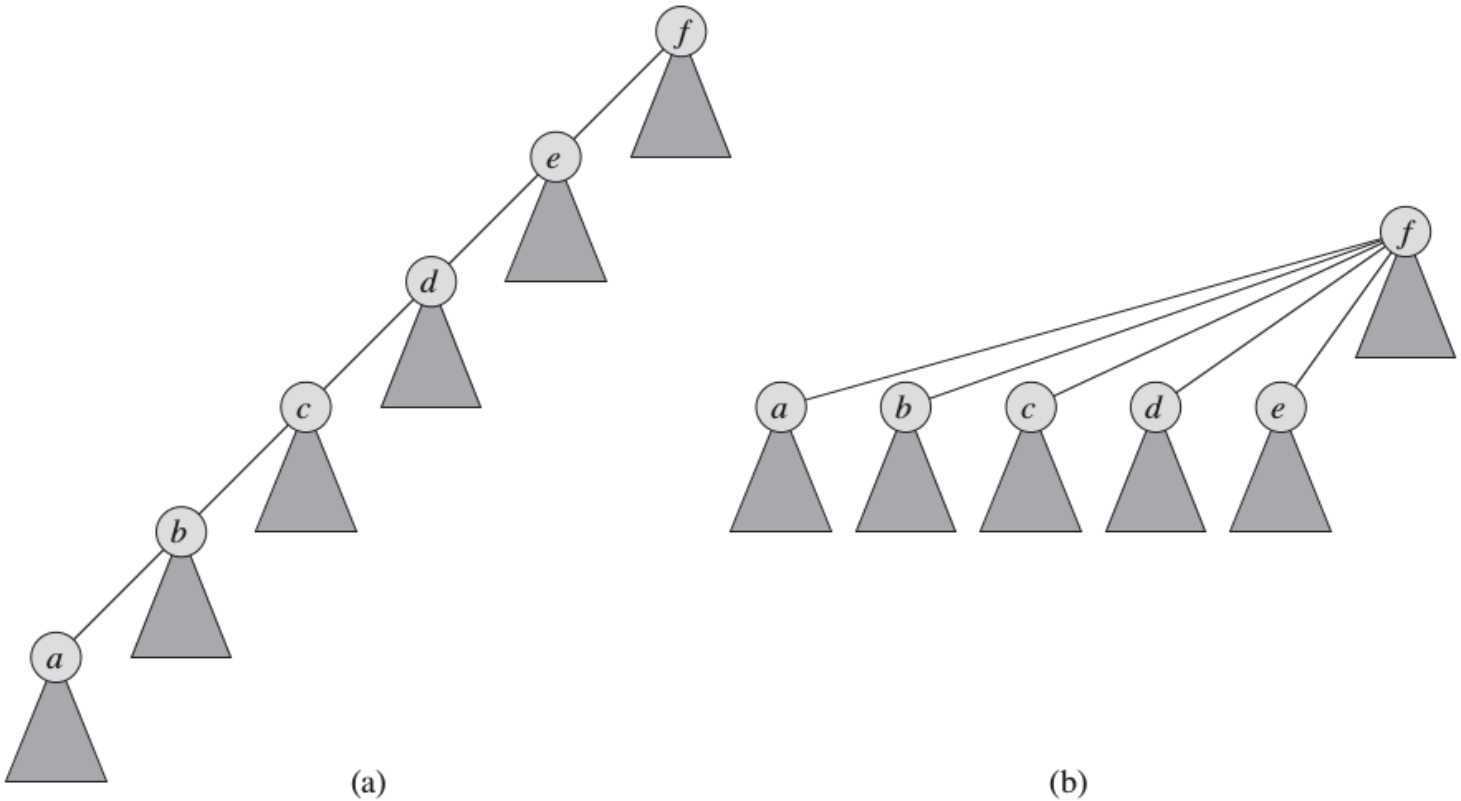
# Union-Find: Improved Find[43]

**Path Compression:** When doing find, point all nodes on search path to root.




[43] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Union-Find: Improved Find[44]




[44] CLRS Fig 21.5

# Union-Find

- Amortized analysis
- Similarity with Red-Black trees
- Worst Case Analysis of Union-by Size + Path Compression
  - Single Union-by-Size:
  - Single Find with Path Compression:
  - Amortized Complexity for $M \geq N$ operations: $O(m \log^* n)$

# $Log^*$ Function[45]

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 =$

[45] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# $Log^*$ Function[45]

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 = \log^* 2^{2^2} = 3$ (i.e. $\log\log\log 16 = 1$)
- $\log^* 65536 =$

[45] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# $Log^*$ Function[45]

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 = \log^* 2^{2^2} = 3$ (i.e. $\log\log\log 16 = 1$)
- $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$ (i.e. $\log\log\log\log 65536 = 1$)
- $\log^* 2^{65536} =$

[45] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# $Log^*$ Function[45]

- $\log^* 2 = 1$
- $\log^* 4 = \log^* 2^2 = 2$
- $\log^* 16 = \log^* 2^{2^2} = 3$ (i.e. $\log\log\log 16 = 1$)
- $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$ (i.e. $\log\log\log\log 65536 = 1$)
- $\log^* 2^{65536} = \ldots = 5$ (i.e. $\log\log\log\log\log 2^65536 = 1$)
- In summary for all reasonable $n$ , $\log^* n \leq 5$

[45] http://courses.cs.washington.edu/courses/cse326/08sp/lectures/19-disjoint-union-find-part-2.pdf

# Tighter Bound

- Tarjan's tighter bound when $M \geq N$, $\Theta(M\alpha(M, N))$
- $\alpha(a, b)$ is the inverse Ackerman function
- It grows even slower than $\log^* n$!

# Summary

## Major Concepts:

- Binary Heap
- Heapsort
- Disjoint set data structures
- Union-Find