Revision Notes

COMPSCI 220: WEEK 14

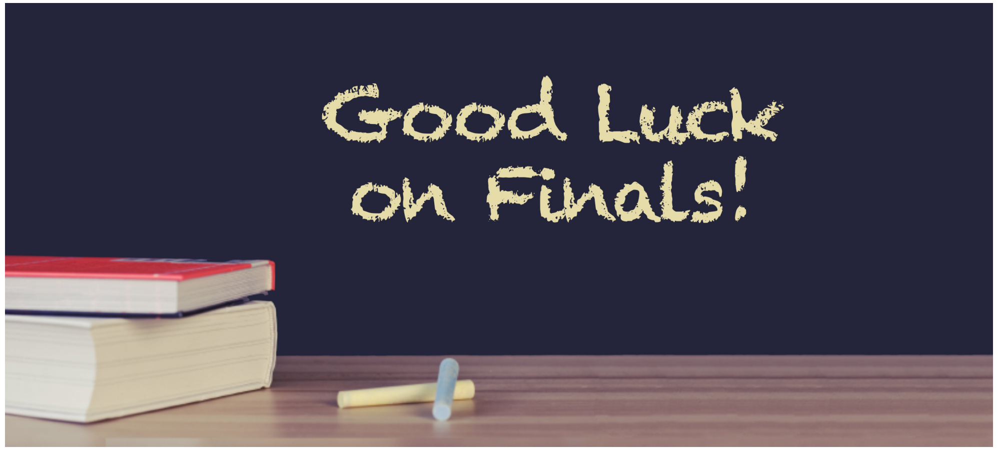
Instructor: Meng-Fen Chiang

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Final Exam

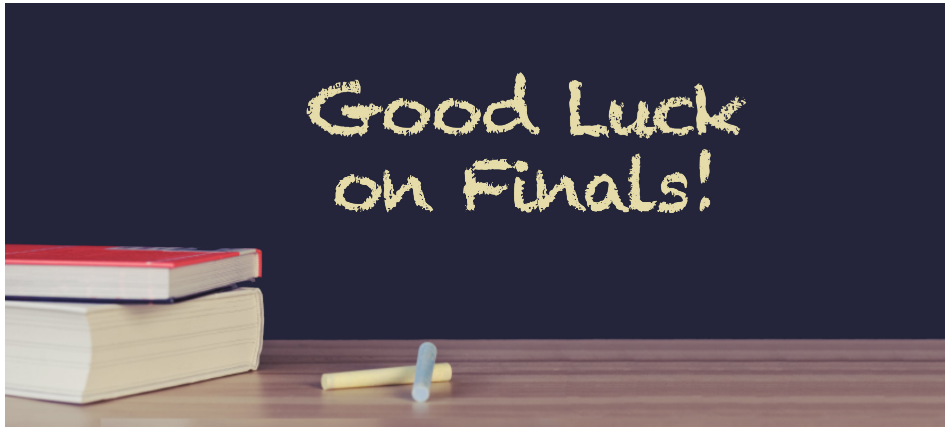
Question Types:

1. Multiple Choice Question
2. Short Answer Questions



Topics

1. Complexity [~10]
2. Sorting [~15]
3. Searching [~5]
4. Graph Traversal [~20]
5. Cycles and Girth [~10]
6. Topological Order [~5]
7. Bipartite Graphs (Coloring) [~10]
8. Shortest Path [~15]
9. Minimum Spanning Tree [~10]



Quicksort: Complexity Analysis

- What is the time complexity of PARTITION?
  - $\Theta(n)$ since we need to traverse the entire list.

- What is the exact number of comparison/swap of Hoare's partition Algorithm in the best/worst case?
  - Best Case: $\Theta(n \log n)$ ▻ when every partition half the list equally
  - Worst Case: $\Theta(n^2)$ ▻ when every partition divides the list at the end
  - Average Case: $\Theta(n \log n)$ ▻ anything between the best and the worst case

Notes on QUICKSORT

- QUICKSORT is very sensitive to input.

- Performance varies a lot between the best and worst case.

- QUICKSORT is not in-place. Unfortunately. Recursion calls require $\Theta(\log n)$ space. Not much but not constant.

- QUICKSORT is not stable. E.g., multiple elements of the same values with a pivot.

Binary Heap

- The three key operations: **Insert**, **FindMax** and **DeleteMax**

Sorted Array

1	4	7	10	12	

6

→

1	4	6	7	10	12

	FindMax	DeleteMax	Insert
Unsorted Array	Θ(n)	Θ(n)	Θ(1)
Sorted Array	Θ(1)	Θ(1)	Θ(n)
Heap (Binary)	Θ(1)	Θ(log n)	Θ(log n)

What we want:

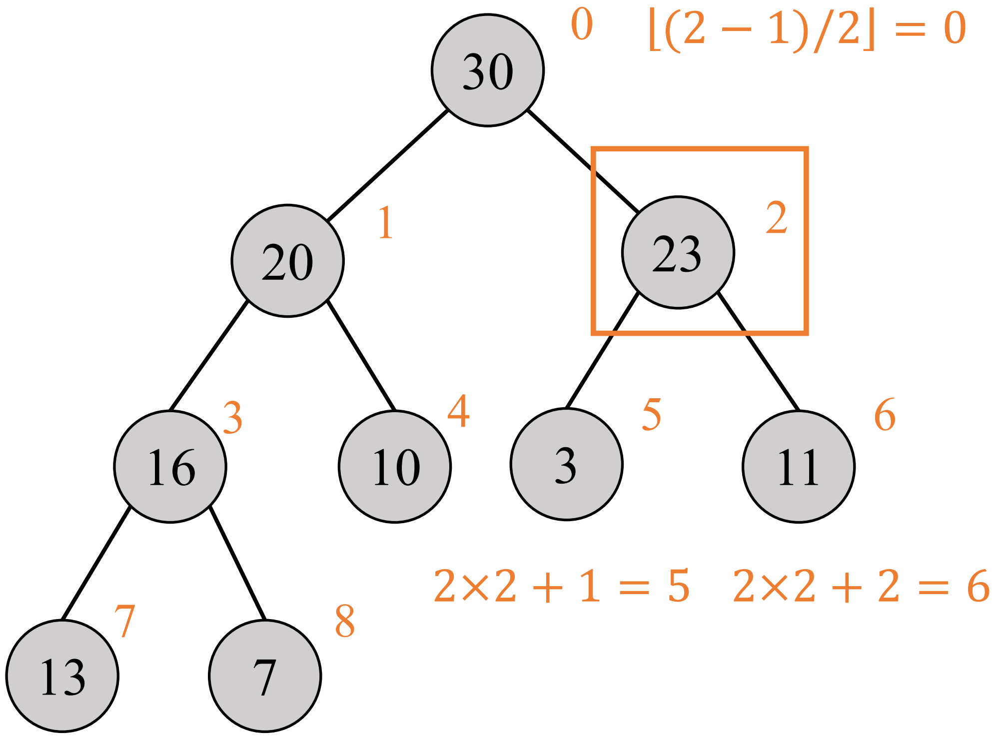
A data structure that can support dynamically organizing the items efficiently:

1. Inserting new items
2. Finding the most important one
3. Deleting the most important one and reorganizing the structure

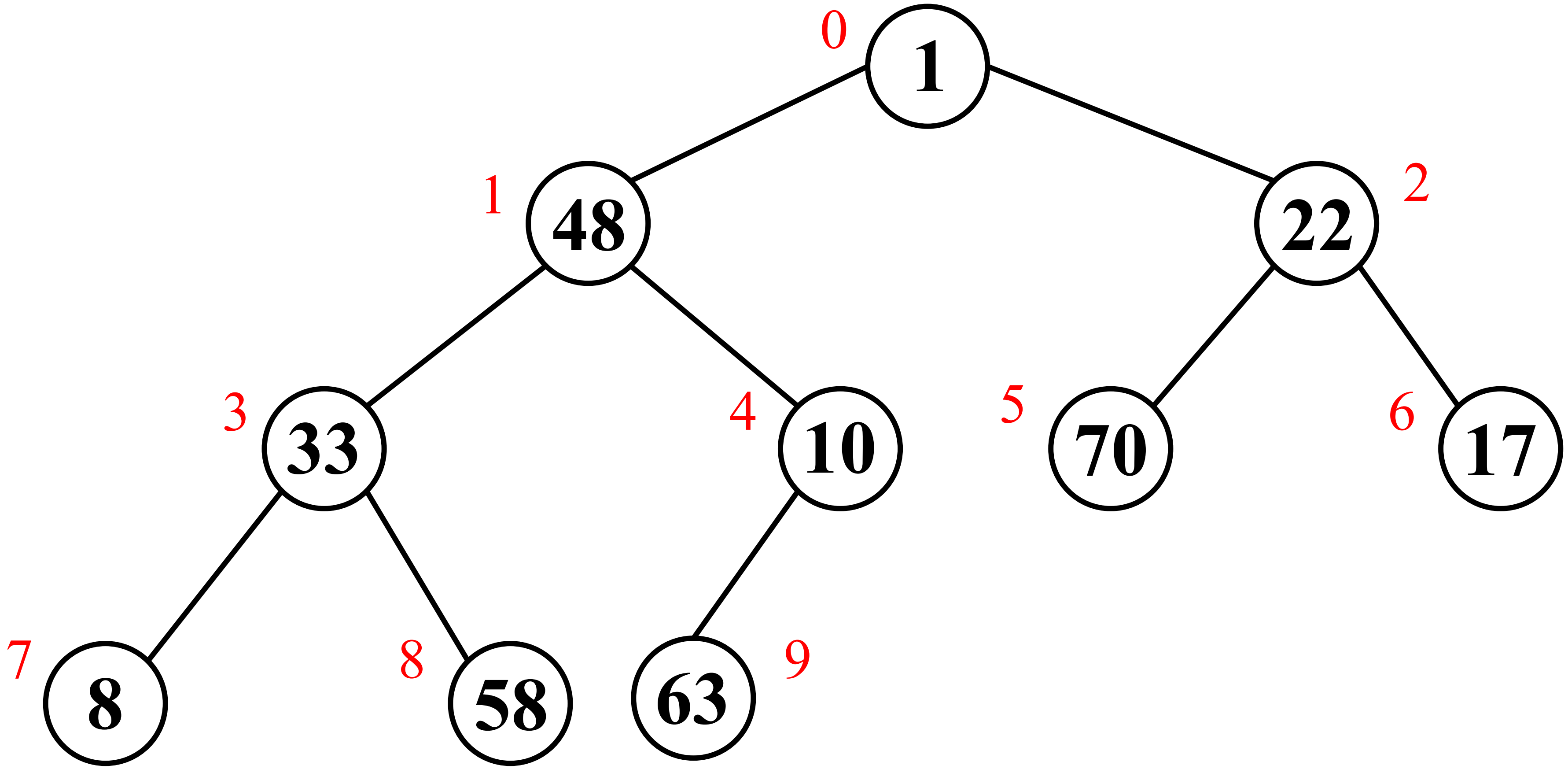
Binary Heap: Array Implementation

- Index: for the k-th element in the array
  - Left child → $2k+1$
  - Right child → $2k+2$
  - Parent → $\lfloor (k-1)/2 \rfloor$

position	1	2	3	4	5	6	7	8	9
keys	30	20	23	16	10	3	11	13	7
index	0	1	2	3	4	5	6	7	8



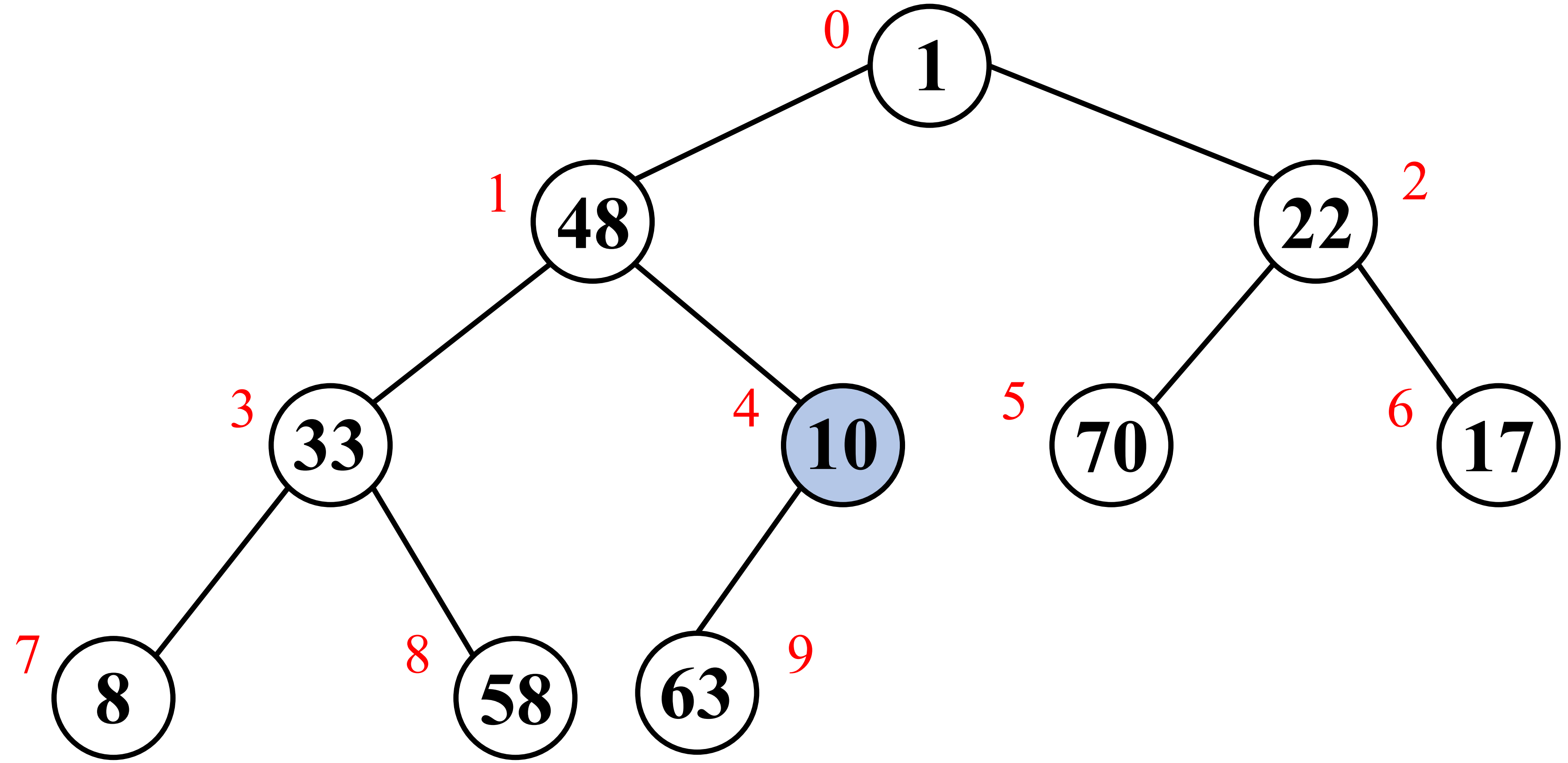
Example: Heapifying without Recursions



1	48	22	33	10	70	17	8	58	63
0	1	2	3	4	5	6	7	8	9

Example: Heapifying without Recursions

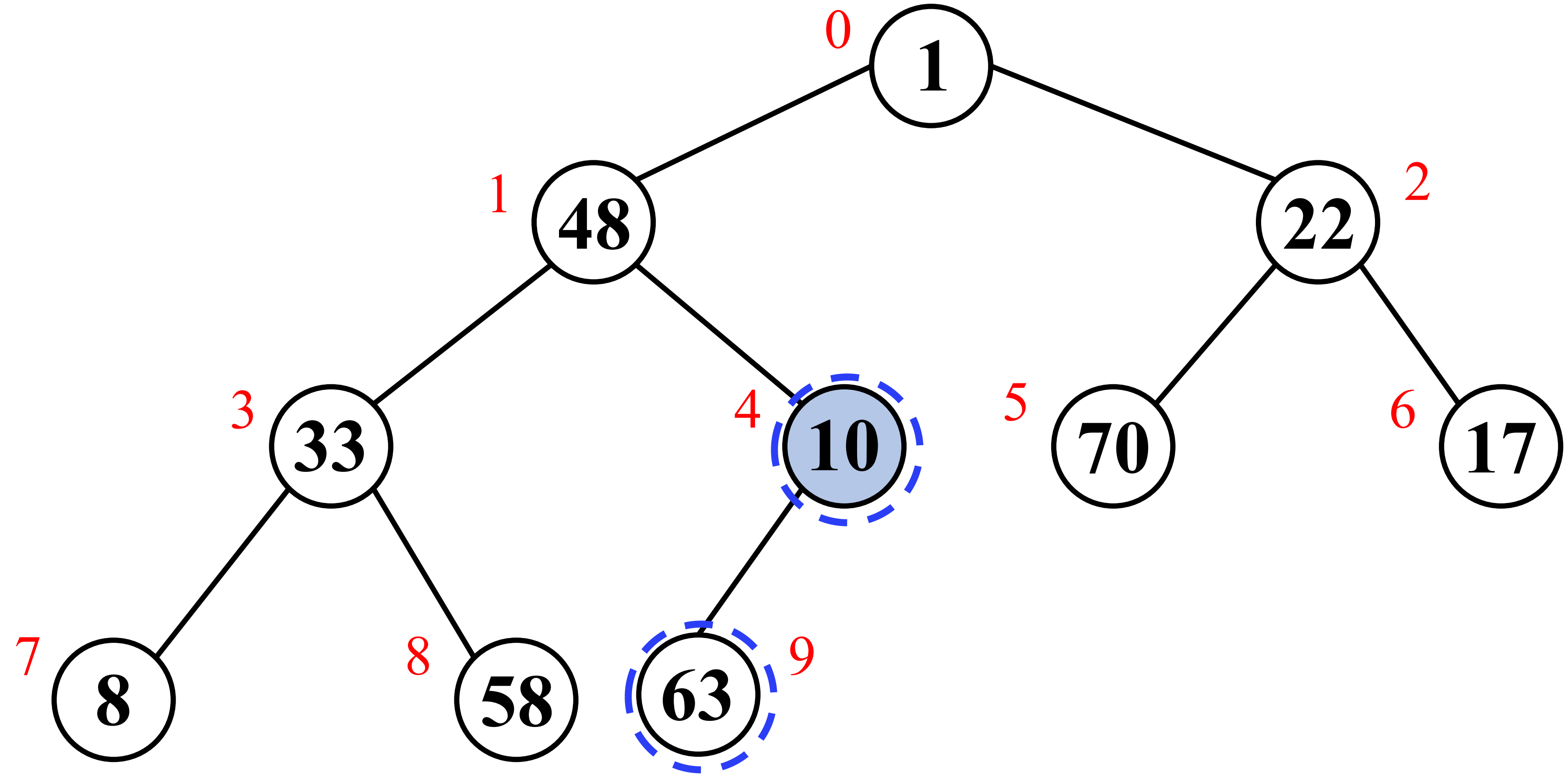
sink 10



1	**48**	**22**	**33**	**10**	**70**	**17**	**8**	**58**	**63**
0	1	2	3	4	5	6	7	8	9

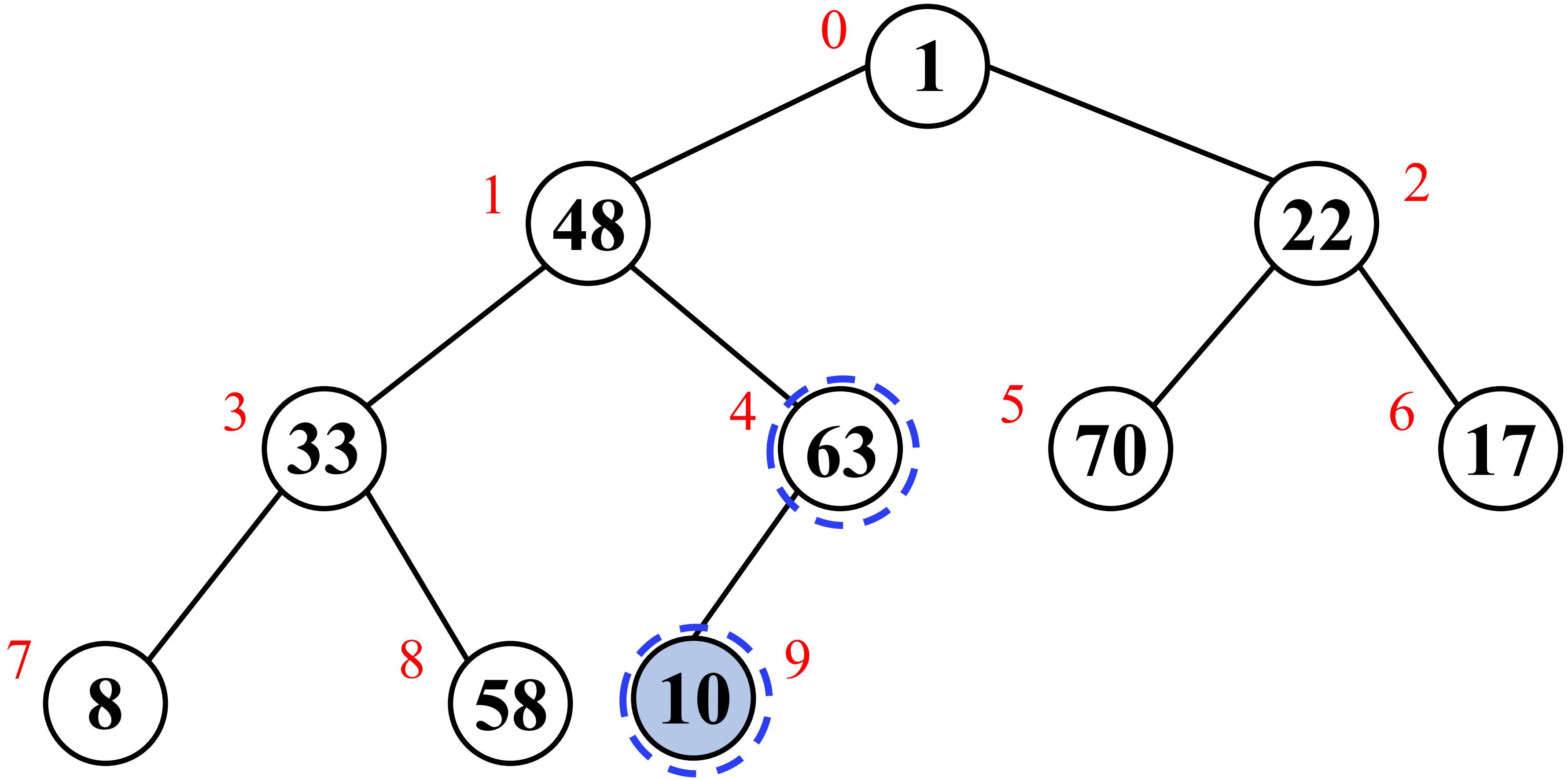
Example: Heapifying without Recursions

sink 10



1	48	22	33	10	70	17	8	58	63
0	1	2	3	4	5	6	7	8	9

Example: Heapifying without Recursions

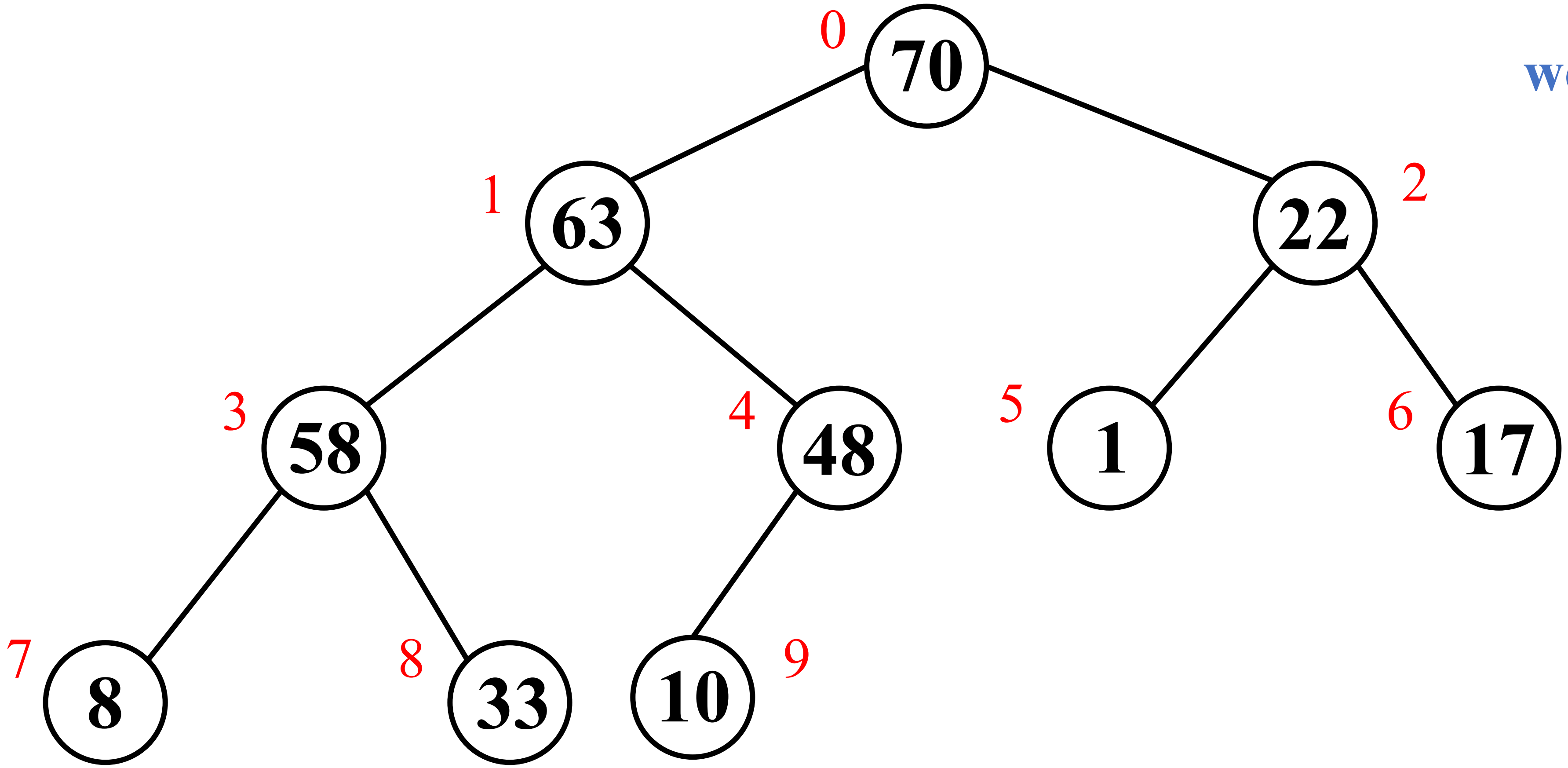


sink 10

1	48	22	33	63	70	17	8	58	10
0	1	2	3	4	5	6	7	8	9

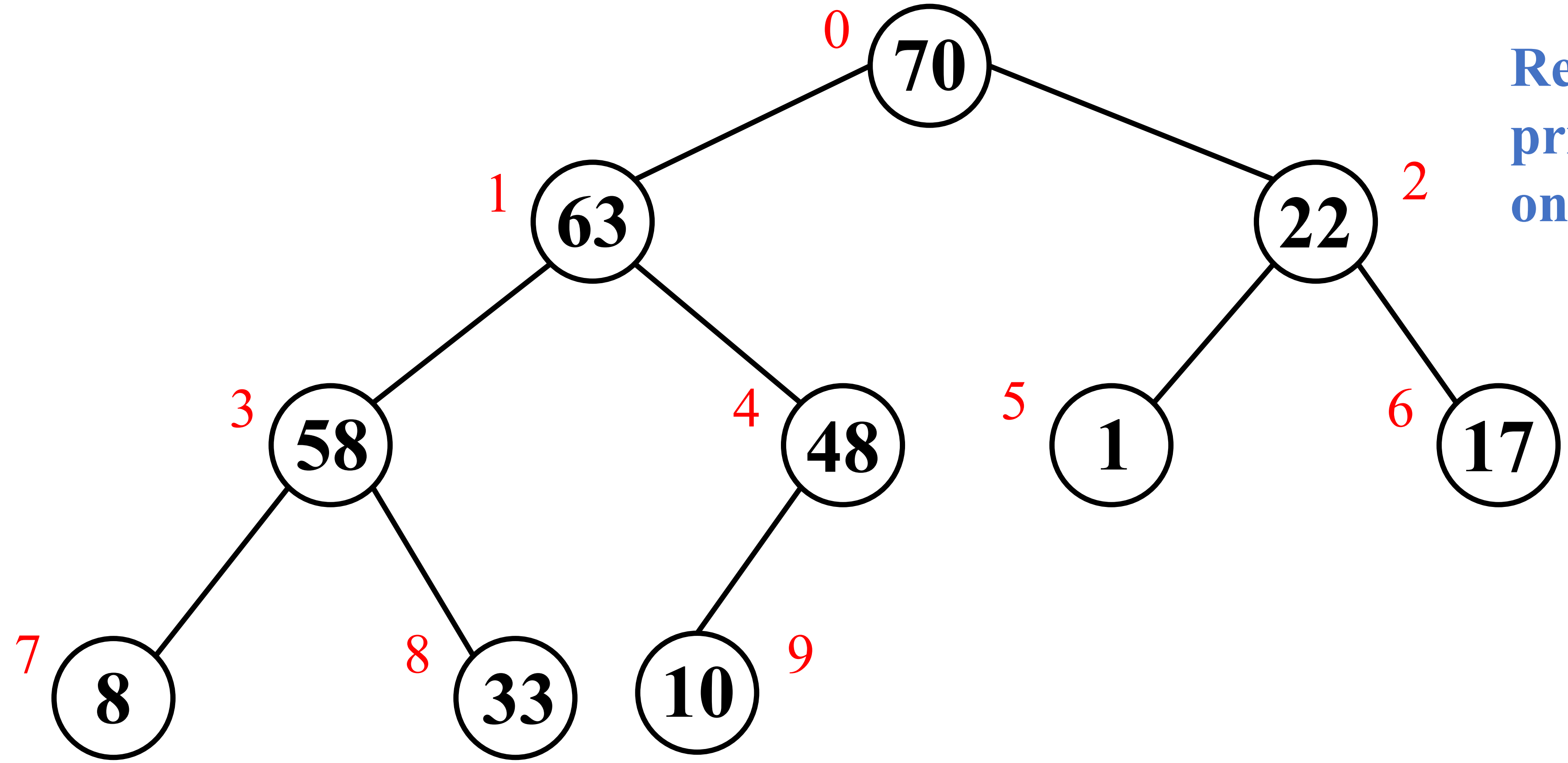
Example: Sorting

we have built the heap



70	63	22	58	48	1	17	8	33	10
0	1	2	3	4	5	6	7	8	9

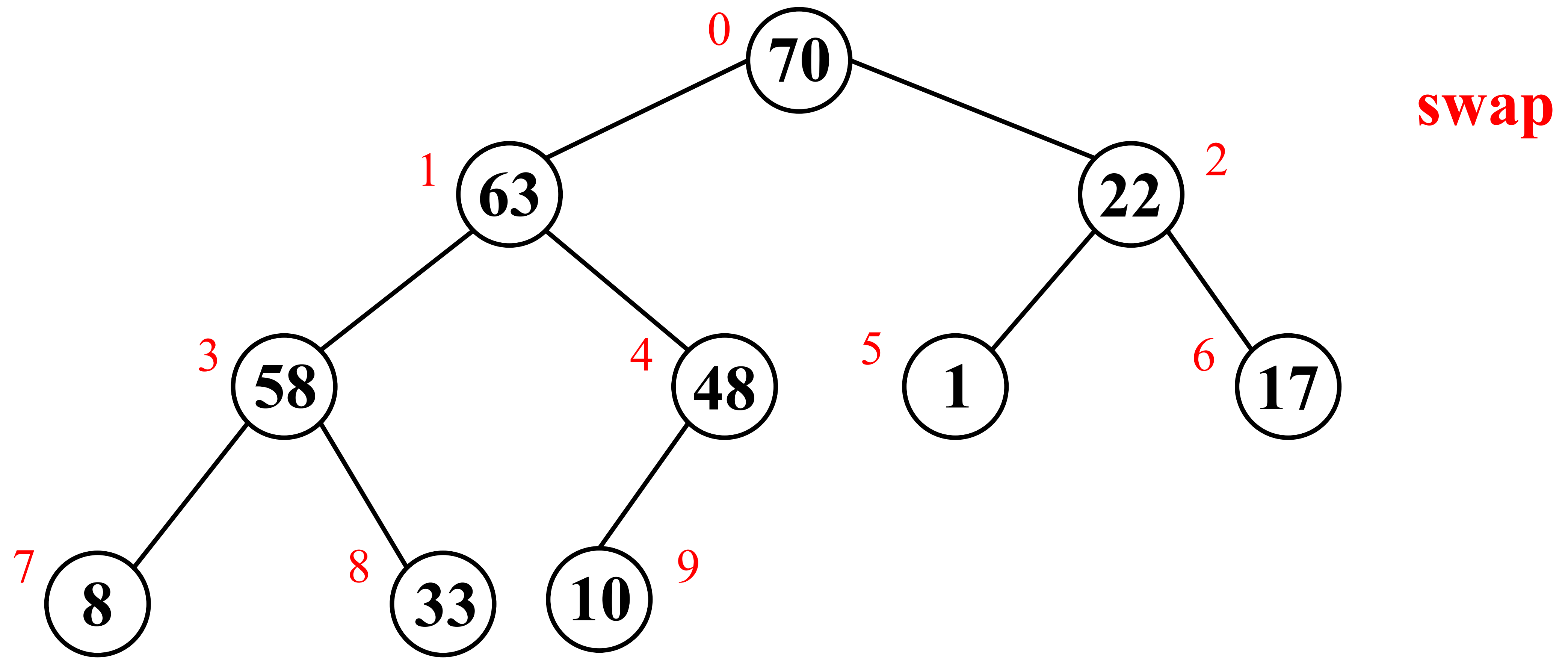
Example: Sorting



70	63	22	58	48	1	17	8	33	10
0	1	2	3	4	5	6	7	8	9

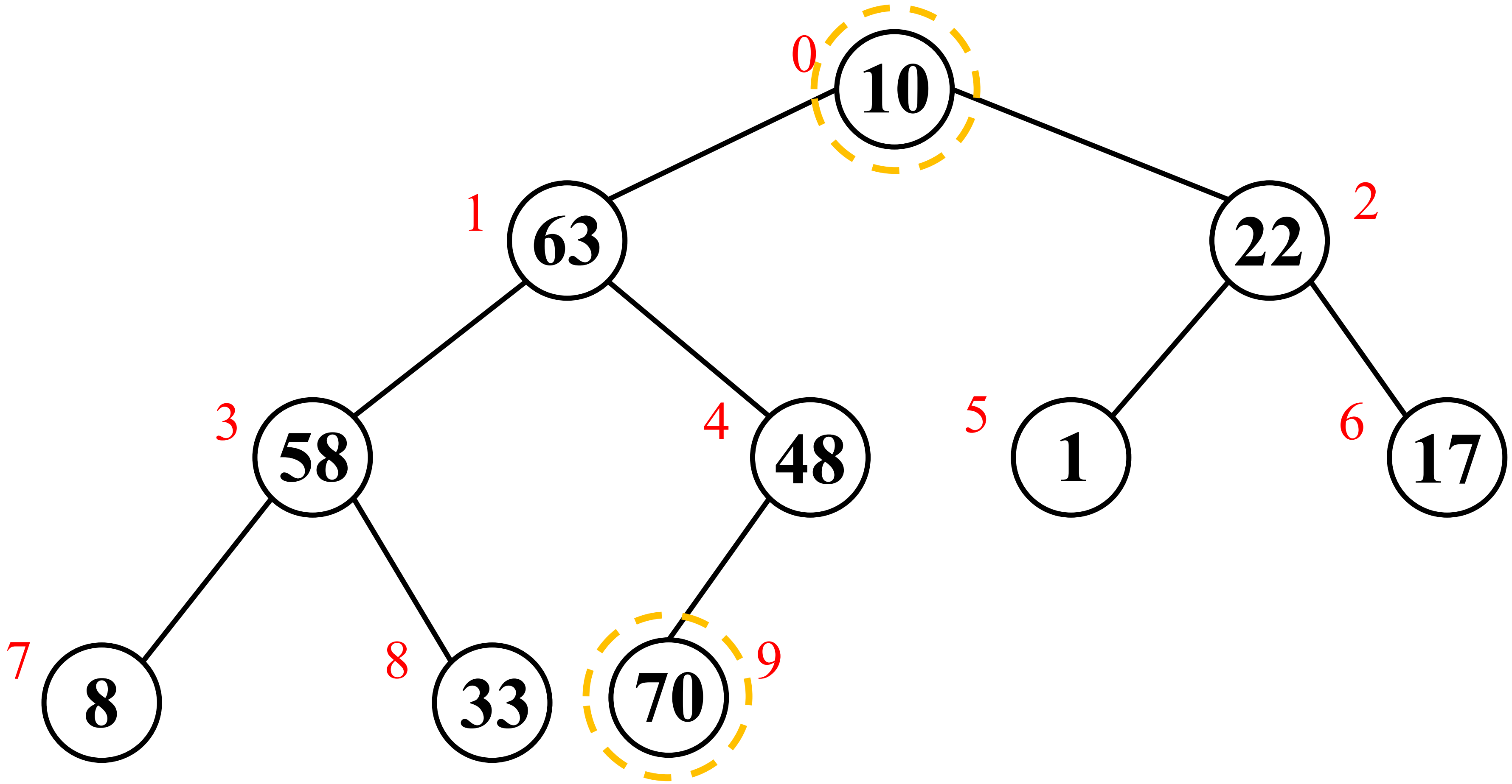
Remove the highest priority element one by one.

Example: Sorting



70	63	22	58	48	1	17	8	33	10
0	1	2	3	4	5	6	7	8	9

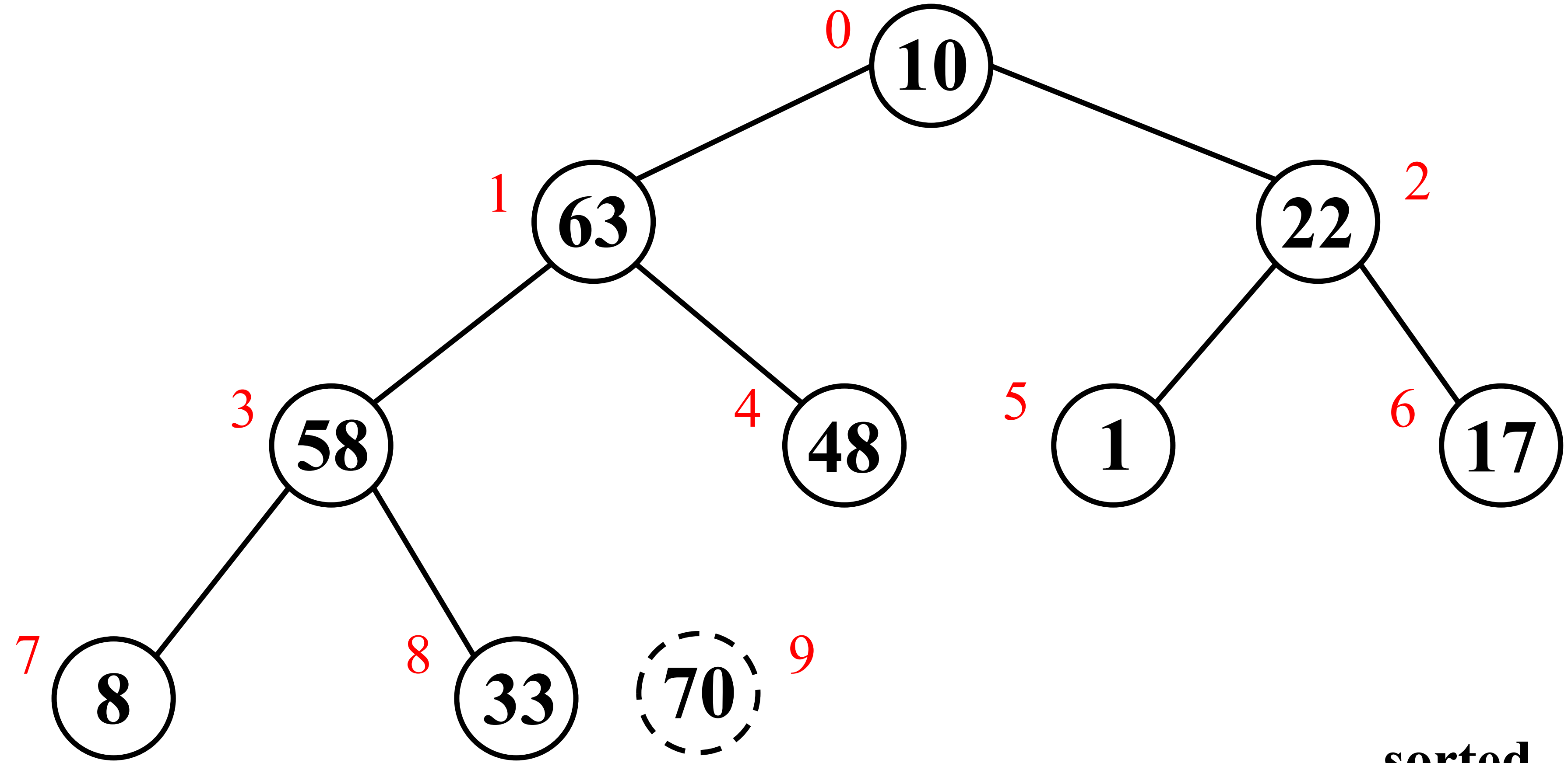
Example: Sorting



swap

10	63	22	58	48	1	17	8	33	70
0	1	2	3	4	5	6	7	8	9

Example: Sorting



delete

sorted

10	**63**	**22**	**58**	**48**	**1**	**17**	**8**	**33**	**70**
0	1	2	3	4	5	6	7	8	9

Heapsort: Complexity Analysis

- Lemma: Heapsort runs in time in $\Theta(n \log n)$ in the worst and average case.
  1. Building the heap
     - straightforward method $\Theta(n \log n)$
     - Bottom-up method with $\Theta(n)$
  2. Removing the maximum key
     - Then heapsort repeats n times the deletion of the maximum key and restoration of the heap property (each restoration is logarithmic in the worst and average case).
     - The running time is $\log(n) + \log(n-1) + \log(n-2) + \cdots + 1 = \log(n!) \in \Theta(n \log n)$.
  - The total running time is $\Theta(n \log n)$

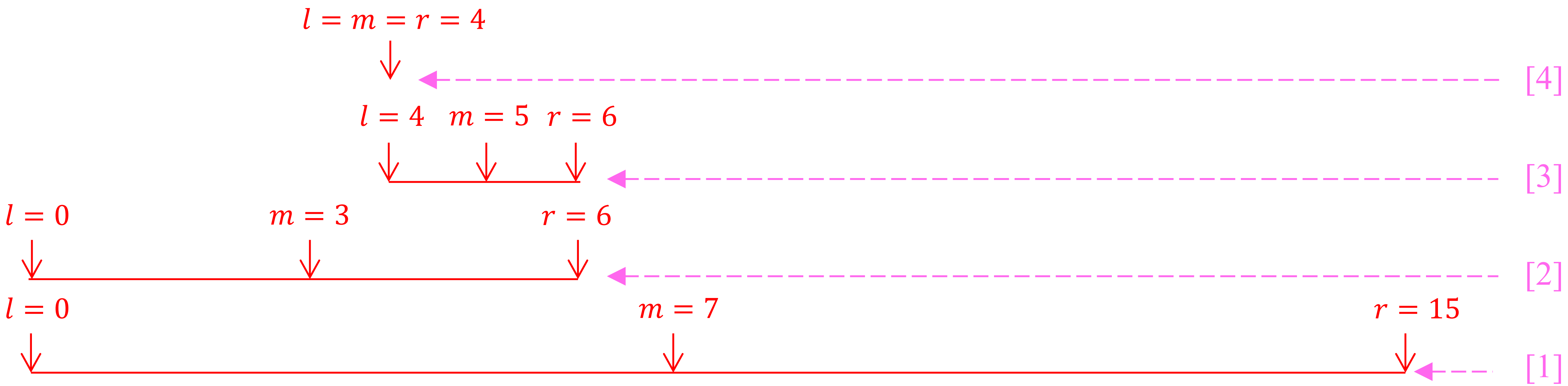
- Heapsort is not in-place. We need extra space for the heap.
- Heapsort is not stable.

Lower Bound of Comparison-based Sorting Algorithms

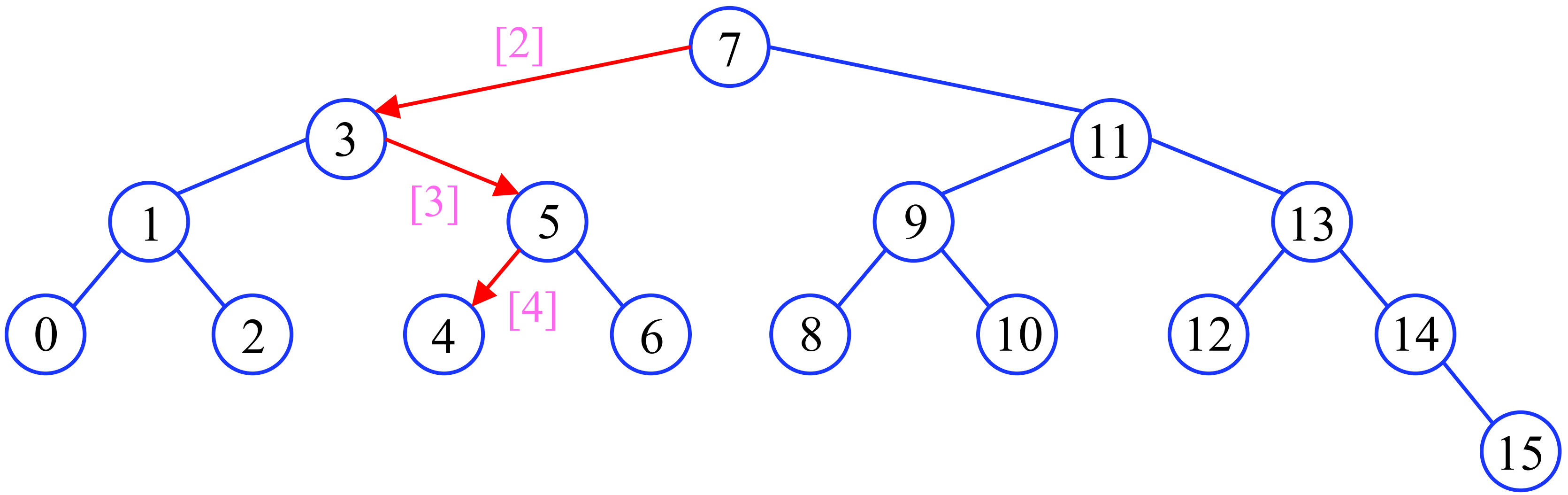
- Best worst/average case time complexity we have seen so far is n log$n$.

Algorithm	**Best**	**Worst**	**Average**
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Mergesort	$n \log n$	$n \log n$	$n \log n$
QUICKSORT	$n \log n$	n^2	$n \log n$
Heapsort	$n \log n$	$n \log n$	$n \log n$

Binary Search in Array $\{k_0 = 7, \ldots, k_{15} = 99\}$ for Key k=42



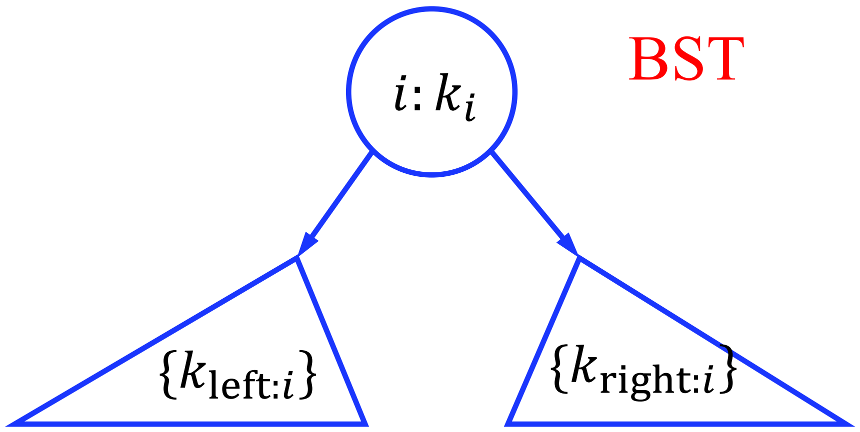
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k_i	7	14	27	33	42	49	51	53	67	70	77	81	89	94	95	99

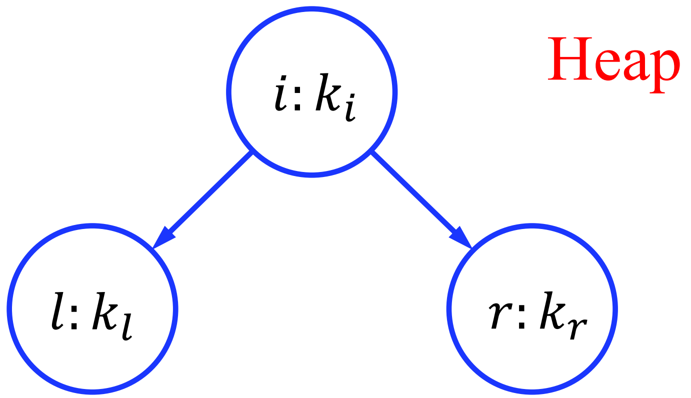


Binary Search Tree: Left-Right Ordering of Keys

- Left-to-right numerical ordering in a BST: for every node i,
  - the values of all the keys $k_{\mathrm{left}:i}$ in the left subtree are smaller than the key k_i in i and
  - the values of all the keys $k_{\mathrm{right}:i}$ in the right subtree are larger than the key k_i in i:

$$\{k_{\mathrm{left}:i}\} \ni l < k_i < r \in \{k_{\mathrm{right}:i}\}$$

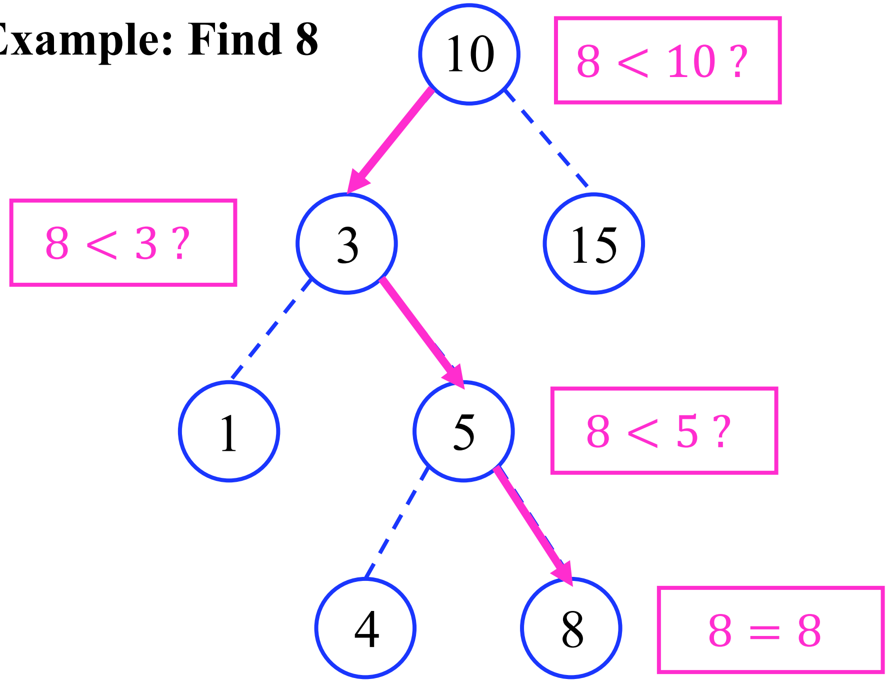




BST Operations: Find / Insert a Node

find: a successful binary search

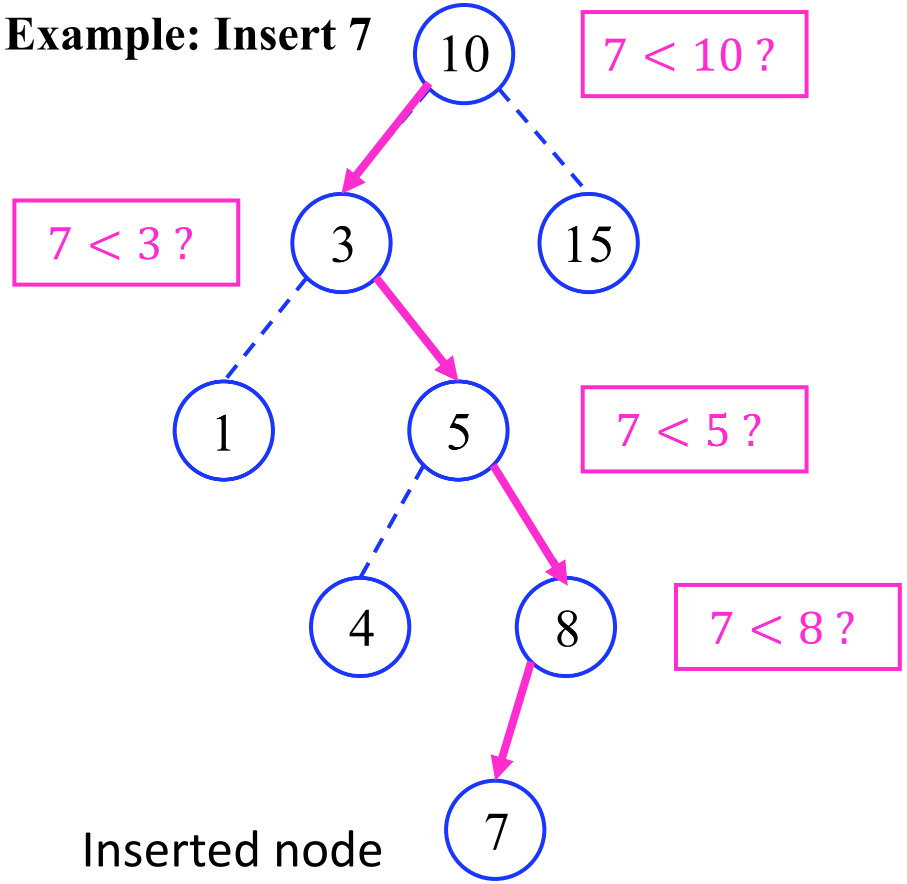
Example: Find 8



Found node

insert: creating a new node at the point where an unsuccessful search stops.

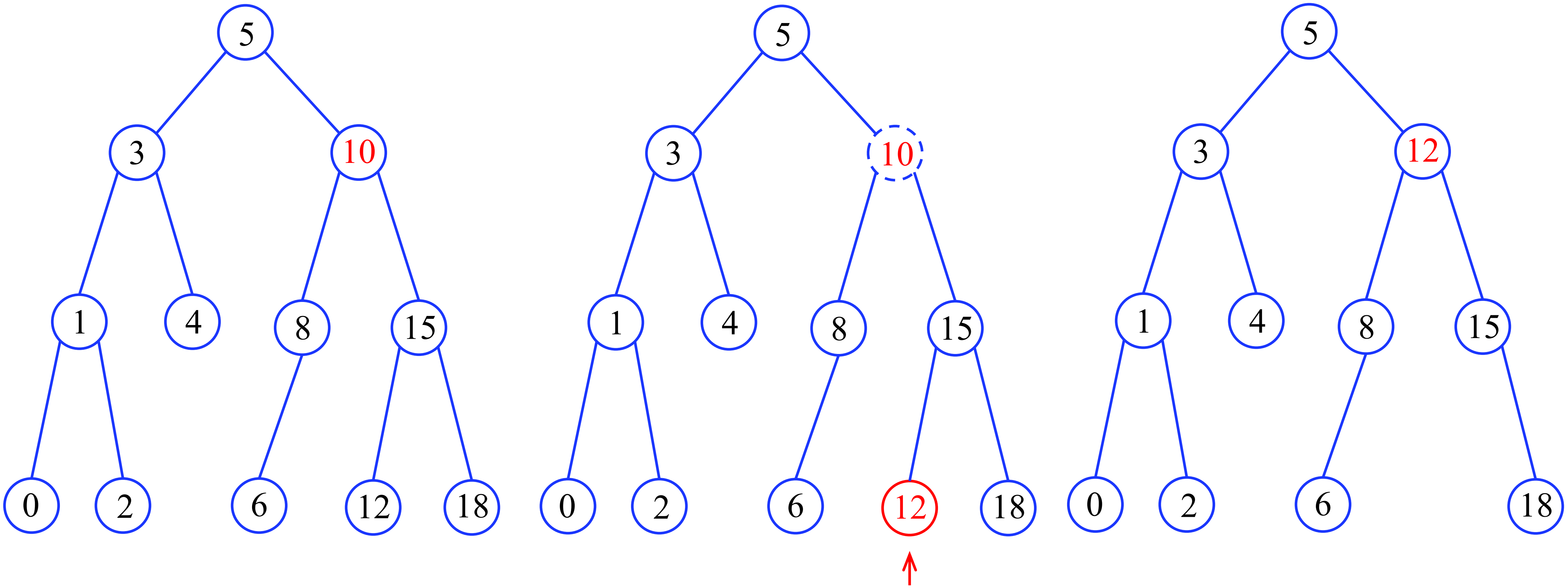
Example: Insert 7



Inserted node

BST Operation: Remove a Node

Remove 10 ⇒ Replace 10 (swap with 12 and delete)



Minimum key in the right subtree

General Graph Traversal: Visit(s)

Algorithm 1 Visit.

function VISIT(node s of digraph G)
    $color[s] \leftarrow$ Grey
    $pred[s] \leftarrow$ Null
    while there is a Grey node **do**
        choose a Grey node u
        if u has a WHITE (out-)neighbour **then**
            choose such a white (out-)neighbour v
            $color[v] \leftarrow$ Grey
            $pred[v] \leftarrow u$
        else
            $color[u] \leftarrow$ Black

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Graph Traversal

Algorithm 1 Visit.

```
function VISIT(node s of digraph G)
    color[s] ← Grey
    pred[s] ← Null
    while there is a Grey node do
        choose a Grey node u
        if u has a WHITE (out-)neighbour then
            choose such a (out-)neighbour v
            color[v] ← Grey
            pred[v] ← u
        else
            color[u] ← Black
```

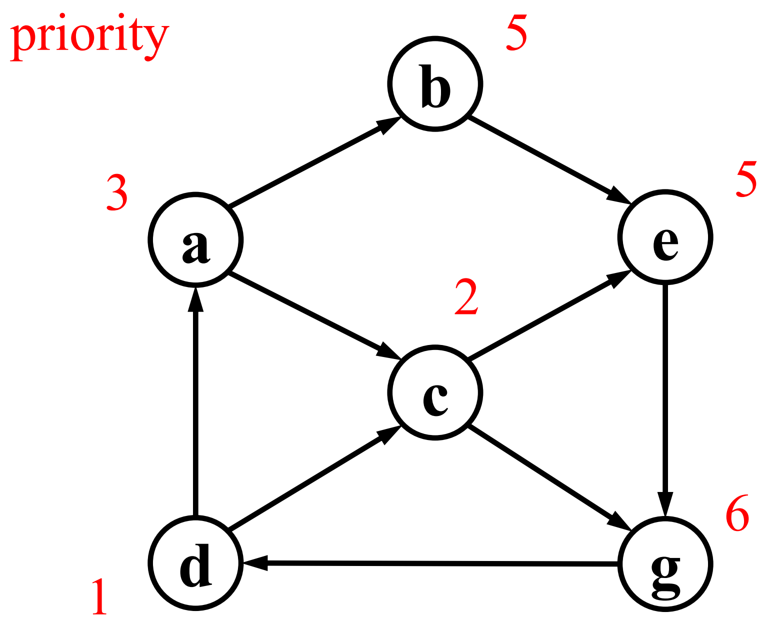
BFS and DFS are special cases of simple PFS.

- BFS, the priority values are the order in which the vertices turn grey (1, 2, 3, …).
- DFS, the priority values are the negative order in which the vertices turn grey (-1, -2, -3, …).

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

PFS Example

- Start at a



(a,3)

(a,3) (b,5)

(a,3) (b,5) (c,2)

(a,3) (b,5) (c,2) (e,5)

(a,3) (b,5) (c,2) (e,5) (g,6)

(a,3) (b,5) (e,5) (g,6)

(b,5) (e,5) (g,6)

(e,5) (g,6)

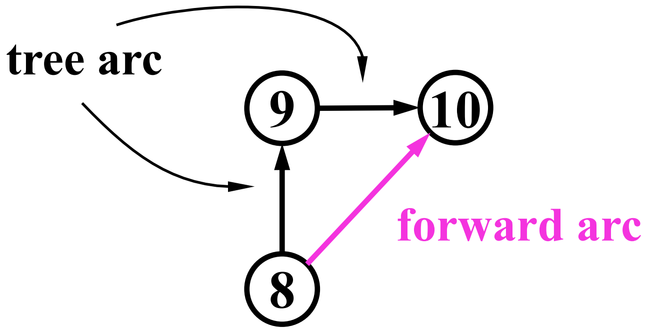
(g,6)

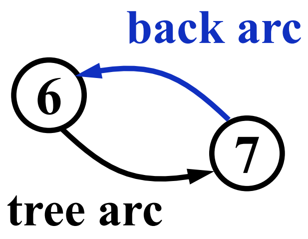
(g,6) (d,1)

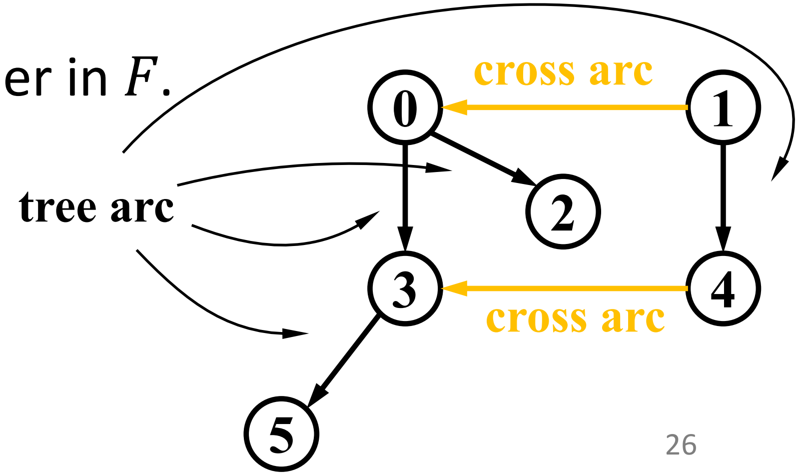
(g,6)

Traversal Arc Classifications

- Suppose we have performed a traversal of a digraph G, resulting in a search forest F. Let $(u, v) \in E(G)$ be an arc.
- The arc is called a tree arc if it belongs to one of the trees of F. If the arc is not a tree arc, there are three possibilities:
  - a forward arc if u is an ancestor of v in F,
  - a back arc if u is a descendant of v in F, and
  - a cross arc if neither u nor v is an ancestor of the other in F.

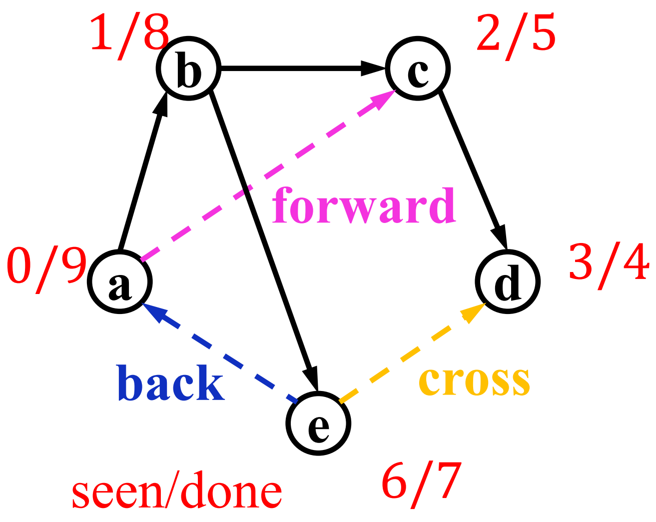






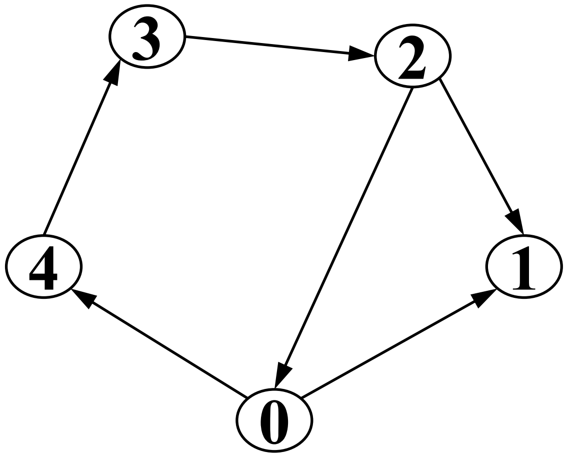
Cycle Detection

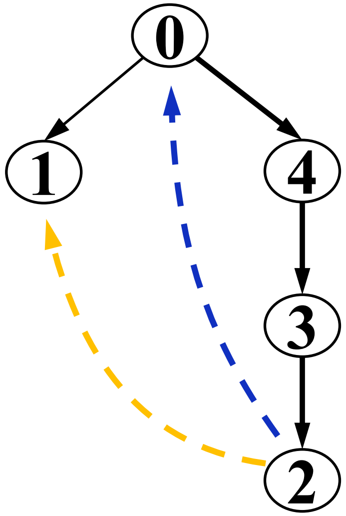
- Suppose that there is a cycle in G and let v be the node in the cycle visited first by DFS. If (u, v) is an arc in the cycle, then it must be a back arc.
- Conversely, if there is a back arc, we must have a cycle.
- Suppose that DFS is run on a digraph G. Then G is acyclic if and only if G does not contain a back arc.



Using DFS to Find Cycles in Digraphs

- Once DFS finds a cycle, the stack contains the nodes that form the cycle

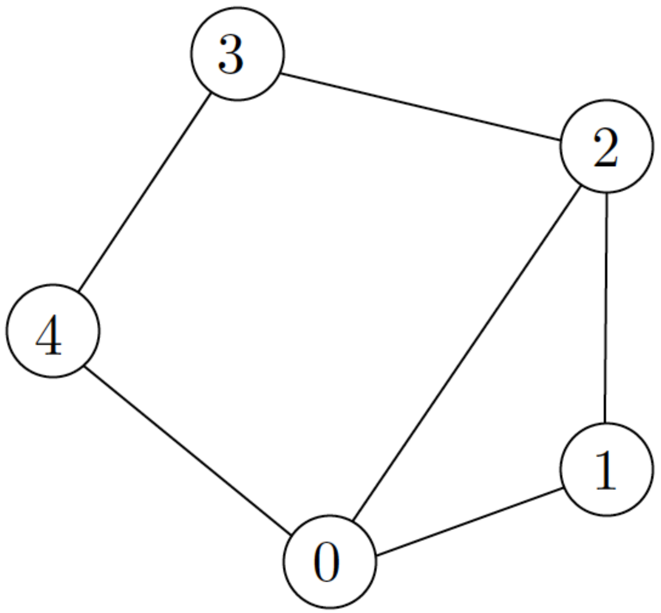


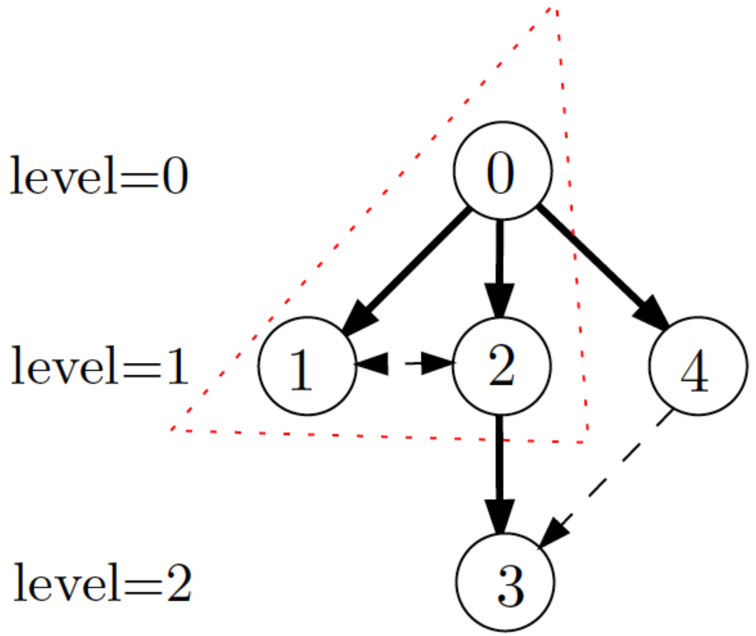


- An easy-to-implement DFS idea may not work properly to find the smallest cycle in undirected graphs.

Finding the Girth of a Graph

- **Using BFS to find cycles in graphs**. Cycles can also be easily detected in a graph using BFS. Finding a cycle of minimum length in a graph is not difficult using BFS (better than DFS).

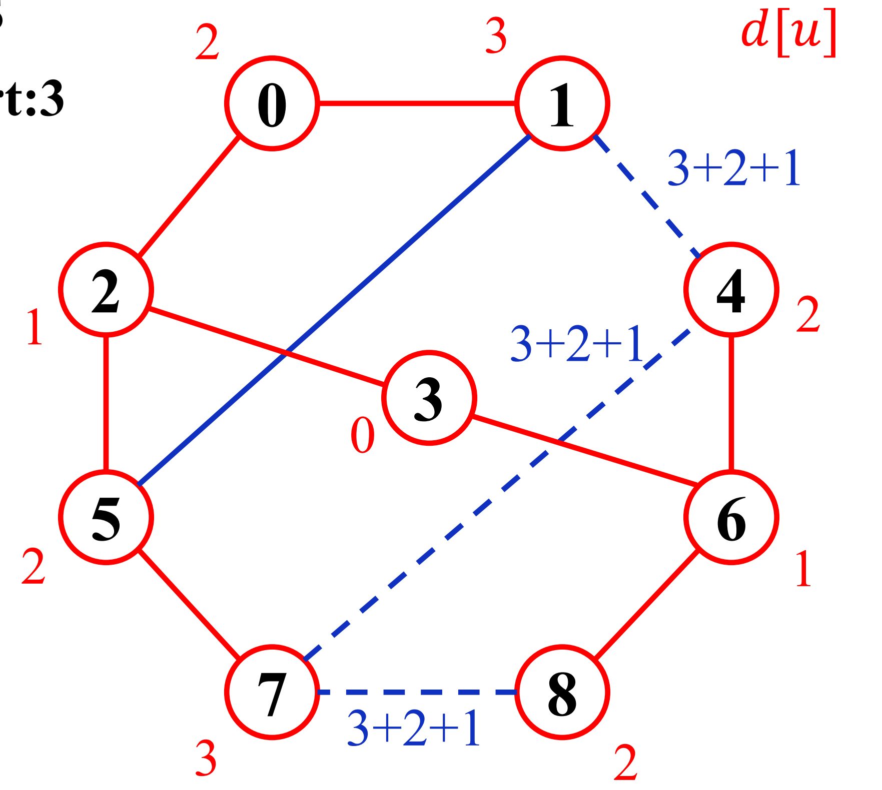




Example

BFS

Start:3

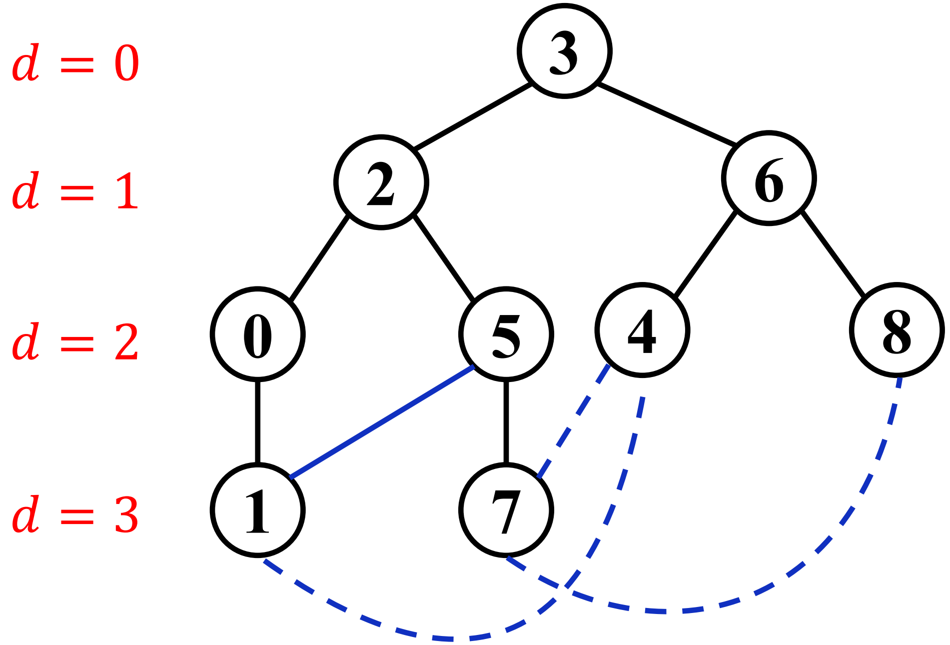


tree edges

cross edge same subtree

tree edges different subtree

(relative to root 3)



Shortest cycle containing 3 has length 6

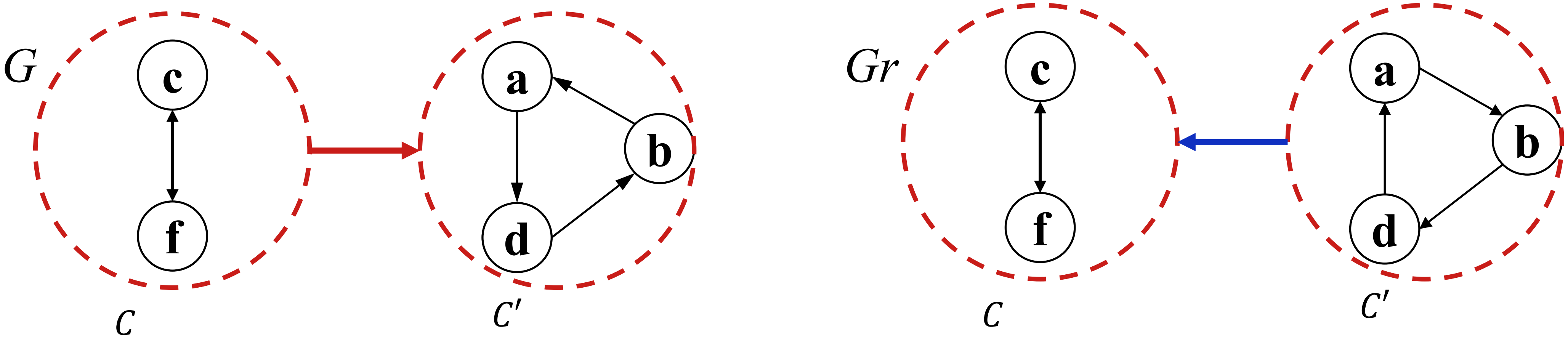
Graph and Digraph Connectivity

	(Undirected) Graph	Di-graph
Connectivity	An undirected graph G is connected if for each pair of vertices $u, v \in V(G)$, there is a path between them.	Strongly connected - for each pair of vertices are mutually reachable Weakly connected - if its underlying graph is connected.
Components	Components - the maximal induced connected subgraphs.	Strong components - the maximal sub-digraphs induced by mutually reachable nodes
Finding components	BFS / DFS	Phase1: DFS on Gr Phase2: DFS on G

Finding Strong Components in a Reverse Digraph

- **Observation**. If we run the **DFS** on the **reverse digraph** Gr, there are two cases in the resulting search forests:

1. Start DFSVisit on some node x in C': The starting node x in C' will be the last one finished (with the greatest $done[x]$) among all nodes in both C and C'.
2. Start DFSVisit on some node y in C: All nodes in C will be finished before the second run of DFSVisit on some node x in C'. Thus, x still has the greatest $done[x]$

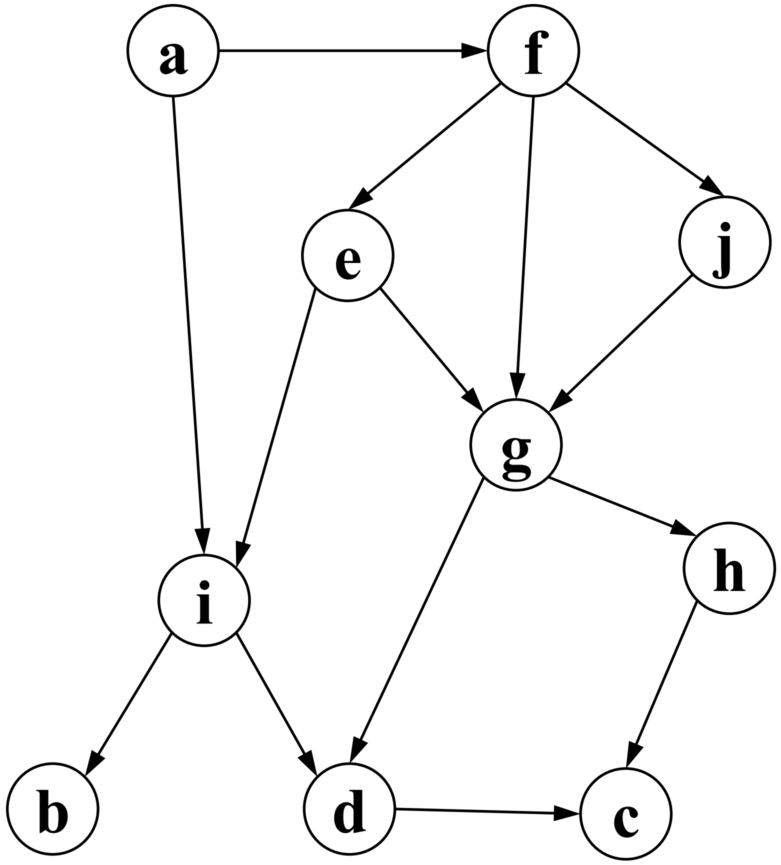


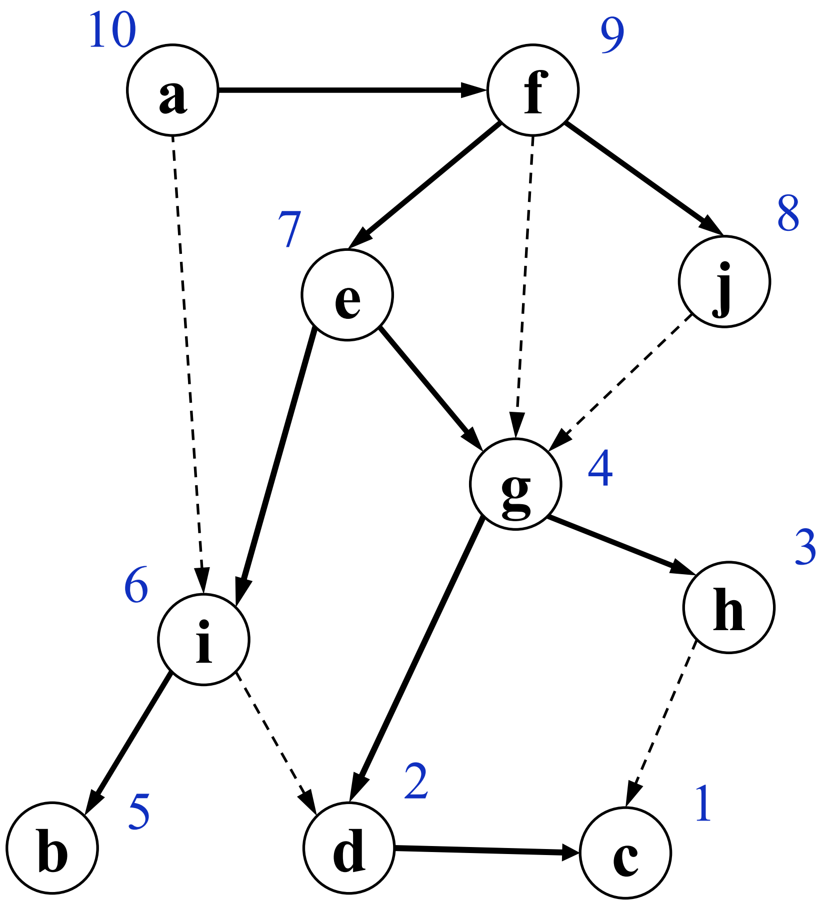
Topological Sorting

- Two solutions:

1. List of finishing times by DFS, in reverse order (since there are no back arcs, each node finishes before anything pointing to it).
2. Zero in-degree sorting – Find a node of in-degree zero, delete it and repeat until all nodes listed.

Example: Topological Order

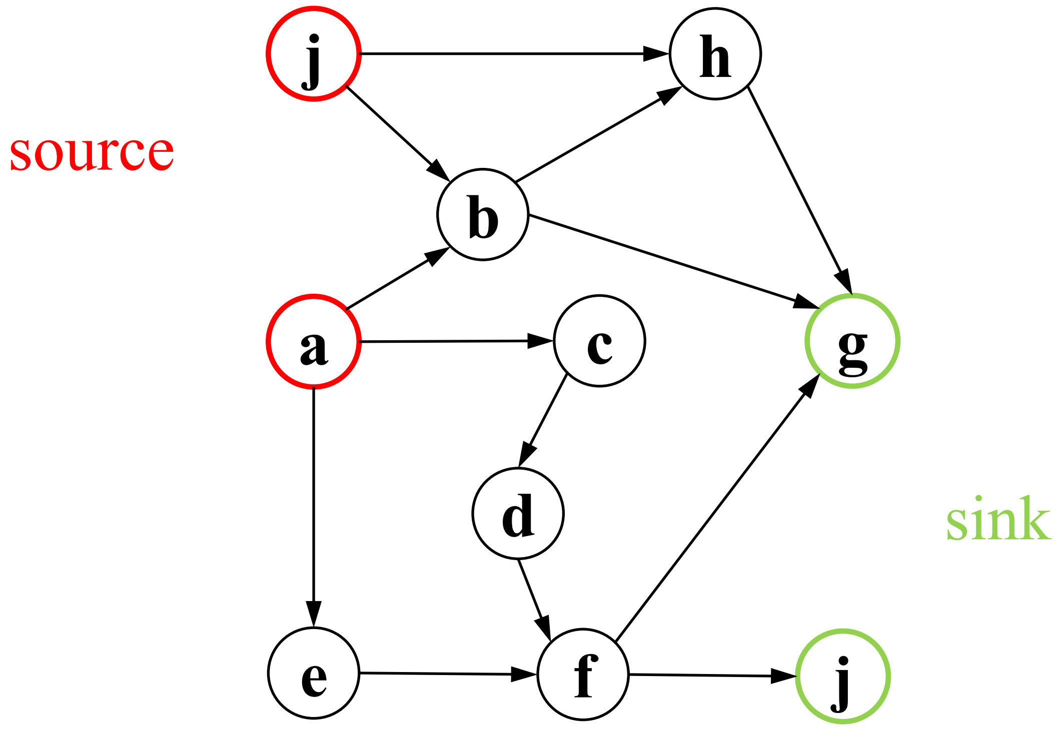




Properties on Topological Sorting

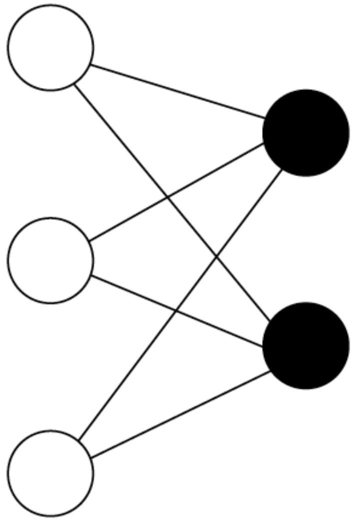
For each arc (u, v), u appears before v in a topological sorting.

- a e j b c d f i h g.
- Usually not unique.



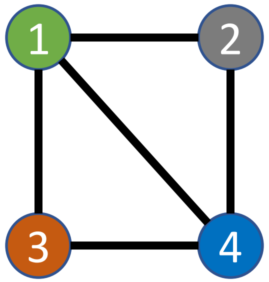
Bipartite Graphs

- **Theorem**. The following conditions on a graph G are equivalent.
    1. G has a 2-coloring;
    2. G is bipartite;
    3. G does not contain an odd length cycle.

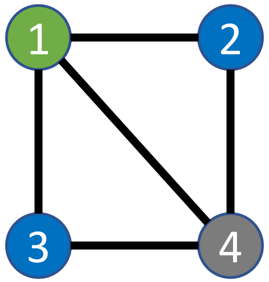


Properties on K-Colourings

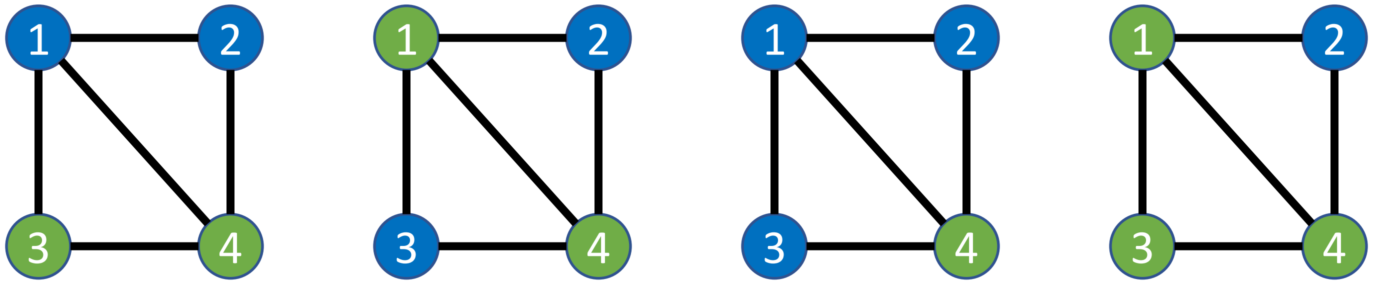
- If a graph has a k-colouring, then it also has a (k+1)-colouring. The reverse does not apply!



This graph has a 4-colouring



… and a 3-colouring…



… but no 2-colouring!

Shortest Path Algorithms

- **Dijkstra** provides the shortest path from **o**[illegible]o any other nodes in a graph
- **Bellman-Ford** is similar to Dijkstra b[illegible]andle **negative costs**
- **Floyd-Warshall** gives short[illegible] between **all** pairs of nodes and can handle **negative costs**

NO NEGATIVE CYCLE ALLOWED

Comparison

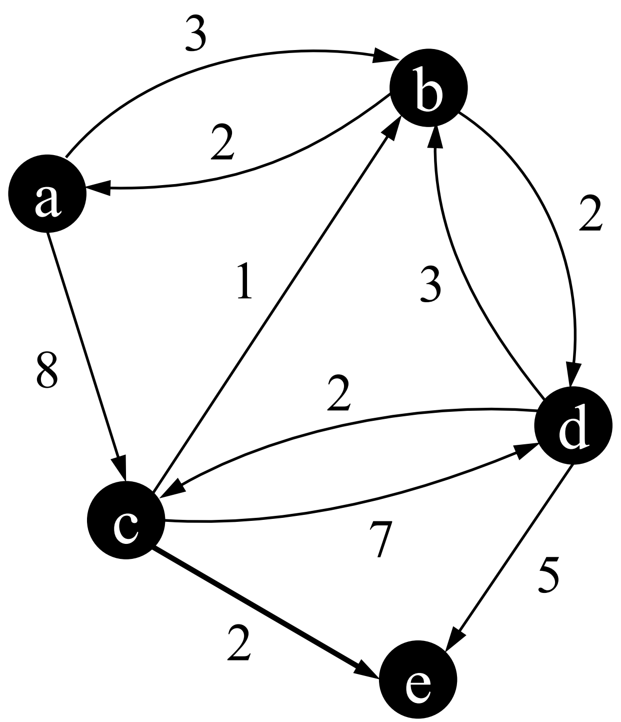
- Summary of how BFS, Djikstra, Bellman-Ford and Floyd can be used to solve the SSSP and APSP problems for weighted and unweighted graphs and digraphs with or without negative arcs.

	SSSP			APSP		
	weighted	unweighted	Complexity	weighted	unweighted	Complexity
BFS	no	yes	$O(m+n)$	no	(yes)	$O(mn+n^2)$
Dijkstra	yes	yes	$O\big((m+n)\log n\big)$	(yes)	(yes)	$O\big((mn+n^2)\log n\big)$
Bellman-Ford	yes	yes	$O(mn)$	(yes)	(yes)	$O(mn^2)$
Floyd	yes	yes	$O(n^3)$	yes	yes	$O(n^3)$

(yes) – need to run for n times

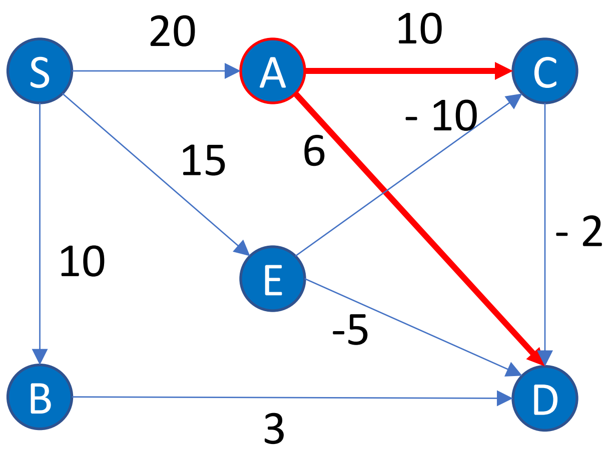
Floyd and Bellman-Ford can detect negative weighted cycles.

Illustrating Dijkstra's Algorithm



BLACK	$dist[x]$
	a, b, c, d, e
a	0, 3, 8, ∞, ∞
a, b	0, 3, 8, 3 + 2 = 5, ∞
a, b, d	0, 3, 3 + 2 + 2 = 7, 5, 10
a, b, c, d	0, 3, 7, 5, 7 + 2 = 9
$V(G)$	

Illustrating Bellman-Ford Algorithm



S	0, S
A	20, S
B	10, S
C	30, A
D	∞
E	15, S

1st Iteration

From S we can get to A with a cost of 20
From A we can get to C with a cost of 10
So we can get from A to C with a total cost of 30

Illustrating Floyd's Algorithm

$d[u,v] = \min(d[u,v], d[u,x] + d[x,v])$

$x = 3$

	0	1	2	3	4
0	0	0	-1	2	5
1	2	0	1	2	7
2	3	1	0	3	6
3	0	-2	-1	0	-3
4	∞	∞	∞	∞	0

resulting matrix for $x = 2$

$$\begin{aligned} d[0,4] &= \min(d[0,4], d[0,3] + d[3,4]) \\ &= \min(5, 2 + (-3)) = -1 \end{aligned}$$

$$\begin{aligned} d[1,4] &= \min(d[1,4], d[1,3] + d[3,4]) \\ &= \min(7, 2 + (-3)) = -1 \end{aligned}$$

$$\begin{aligned} d[2,4] &= \min(d[2,4], d[2,3] + d[3,4]) \\ &= \min(6, 3 + (-3)) = 0 \end{aligned}$$

Minimum Spanning Tree Algorithms

Both algorithms choose and add at each step a min-weight edge from the remaining edges, subject to constraints

Prim's MST algorithm:

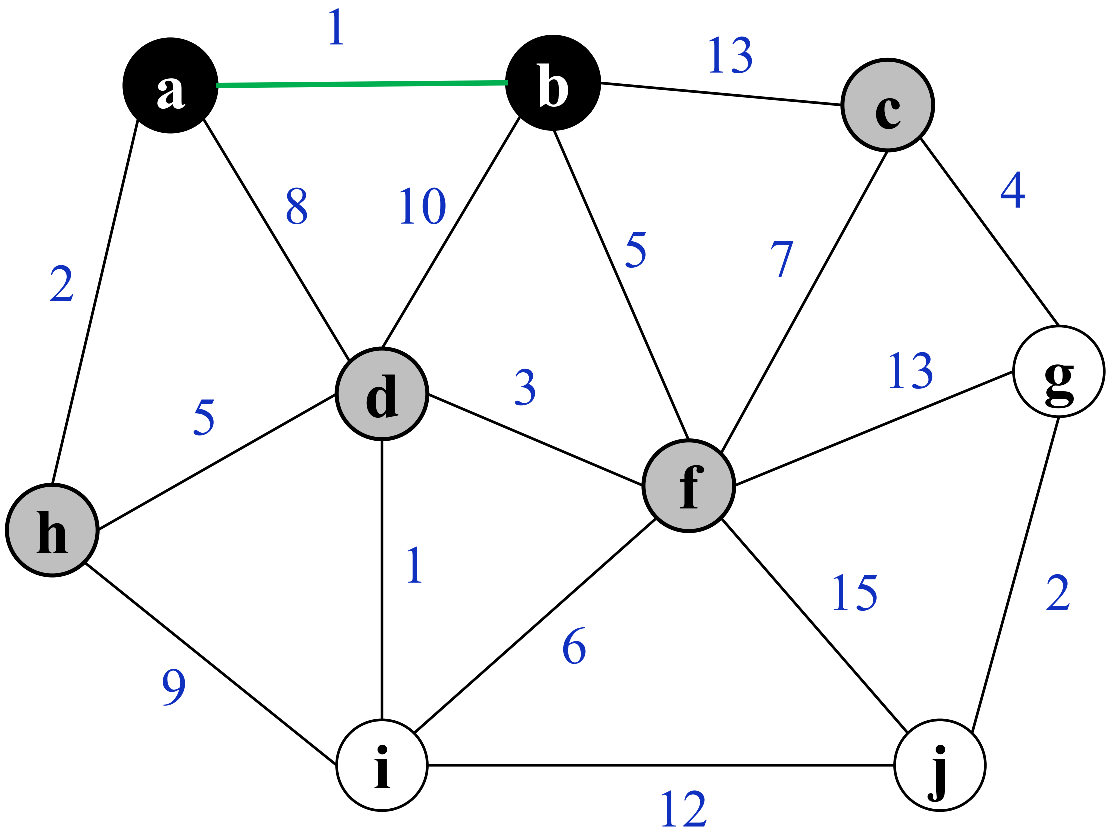
- Start at a root vertex.
- Two rules for a new edge:
    1. No cycle in the subgraph built so far.
    2. Connect the subgraph built so far.
- Terminate if no more edges to add can be found.
- At each step: an acyclic connected subgraph being a tree.

Kruskal's MST algorithm:

- Start at a min-weight edge.
- One rule for a new edge:
    - No cycle in a forest of trees built so far.
- Terminate if no more edges to add can be found.
- At each step: a forest of trees merging as the algorithm progresses (can find a spanning forest for a disconnected graph).

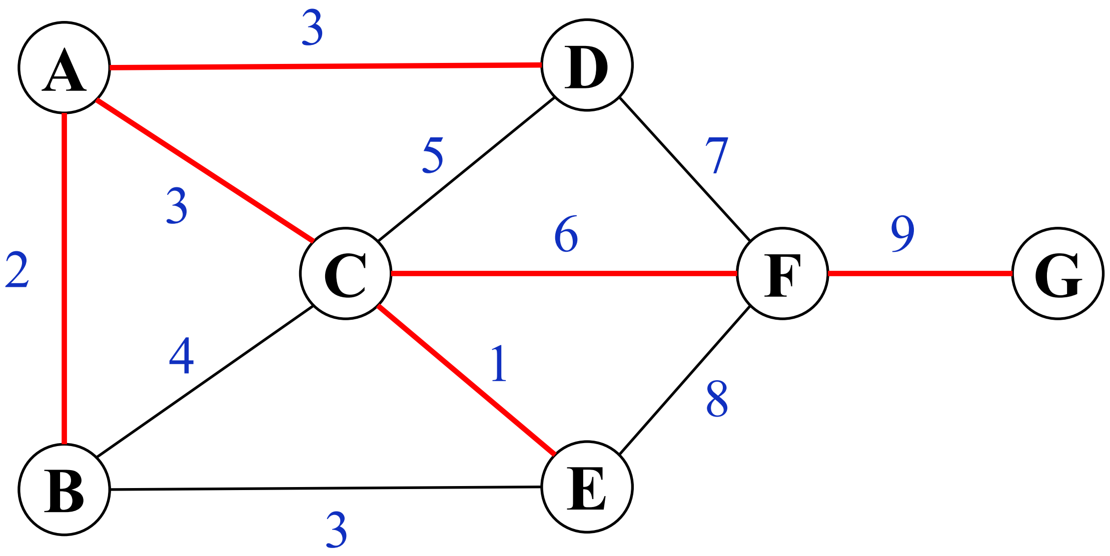
Illustrating Prim's Algorithm

Priority Queue Q:
(d, 8)
(h, 2)
(c, 13)
(f, 5)



Pred:
a: null
b: a
c: b
d: a
f: b
g: null
h: a
i: null
j: null

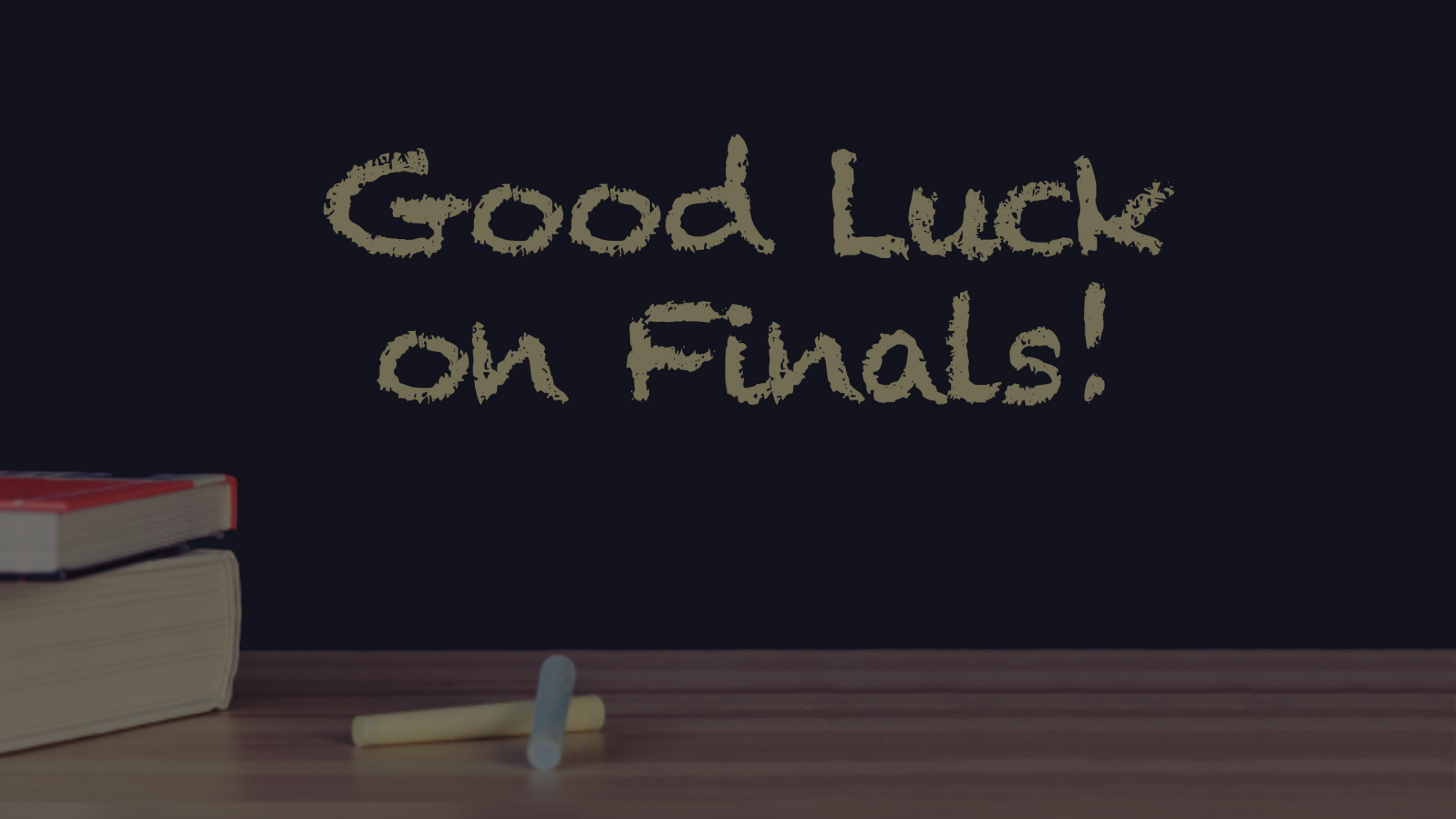
Illustrating Kruskal's Algorithm



$\{C, E\}, \{A, B\}, \{A, C\}, \{A, D\}, \{C, F\}, \{F, G\}$

1 2 3 3 6 9

MST of weight: 24


Good Luck on Finals!