UNIVERSITY OF CONNECTICUT

Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

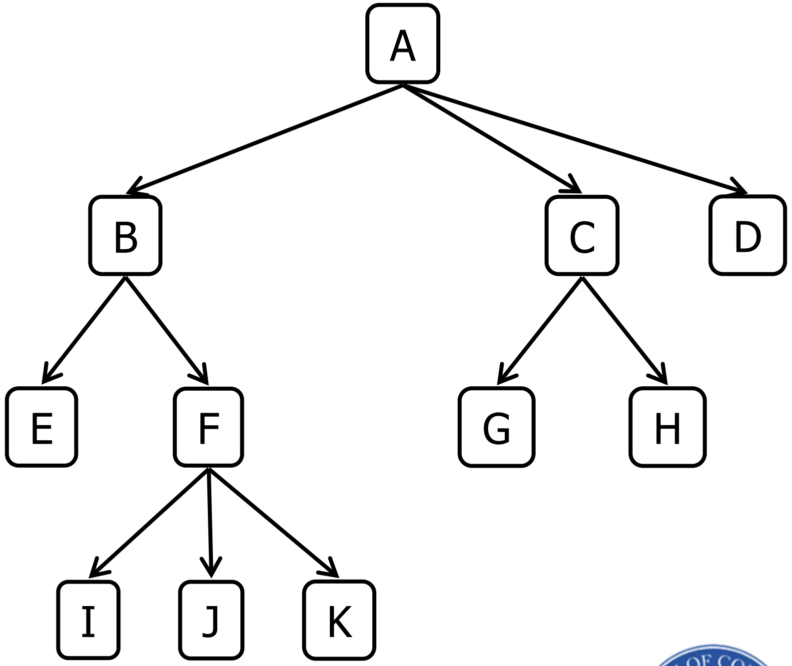
(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

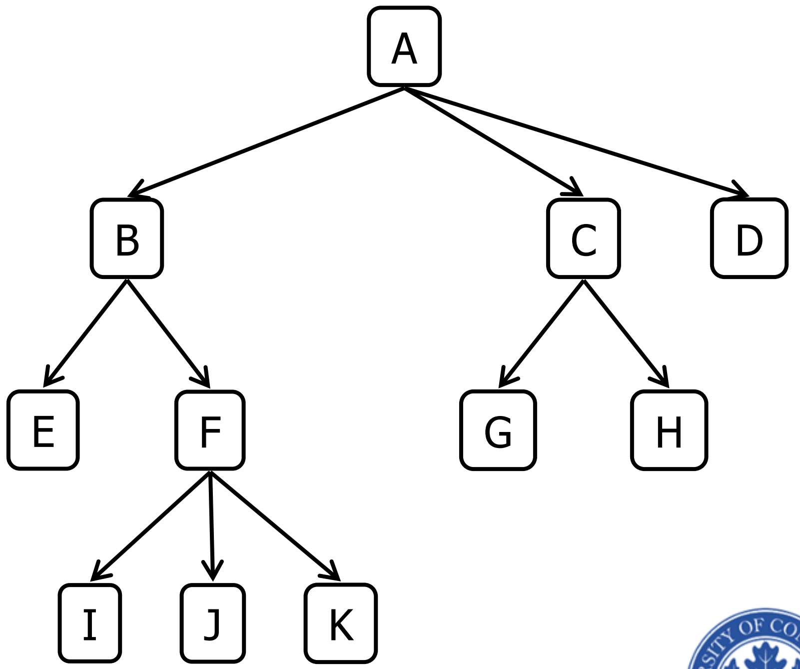
Module 9 – Trees

- **Root**: Highest-most node without parent (A)
- **Edge:** Connection between two nodes to show a relationship between them
- **Path:** A path is an ordered list of nodes that are connected by edges (from top to bottom)
- **Parent:** A node is a parent of all nodes it connects to with outgoing edges
- **Children:** The set of nodes which have incoming edges from a parent node
- **Sibling:** Nodes that are children of the same parent



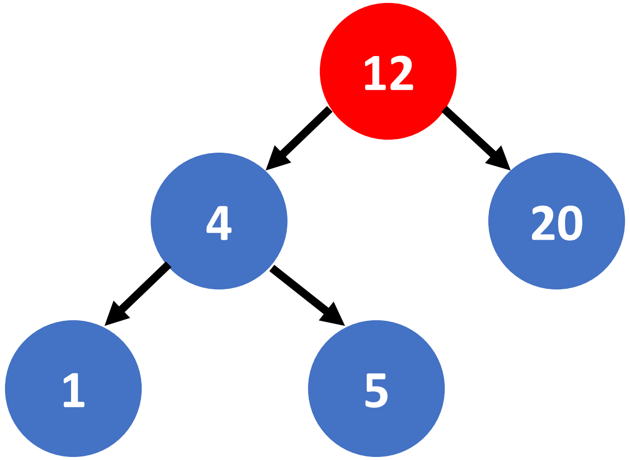
Module 9 – Trees

- **Descendant** of node (x)**:** all nodes for which there is path from x. (child, grandchild, grand-grandchild)
- **Ancestors** of node (x)**:** all nodes which x is a descendant of (parent, grandparent, grand-grandparent)
- **Leaf Node:** Nodes which have no children (J, K, etc)
- **Subtree:** Set of nodes and edges comprised of a parent and all descendants of that parent (C-G-H)
- **Degree** of a node**:** The number of its children
- **Degree** of a tree**:** Total number of nodes in it



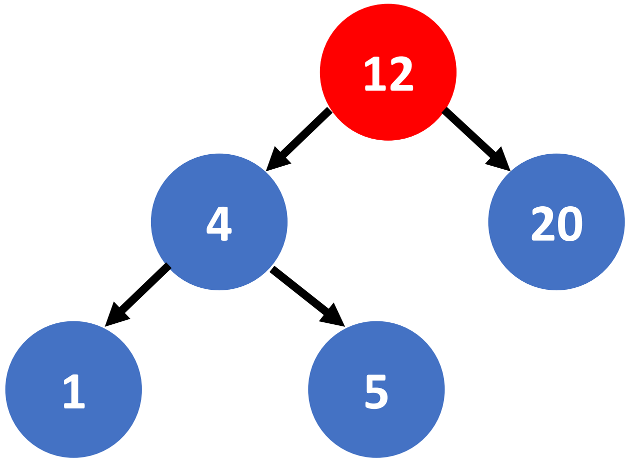
Module 9 – Trees

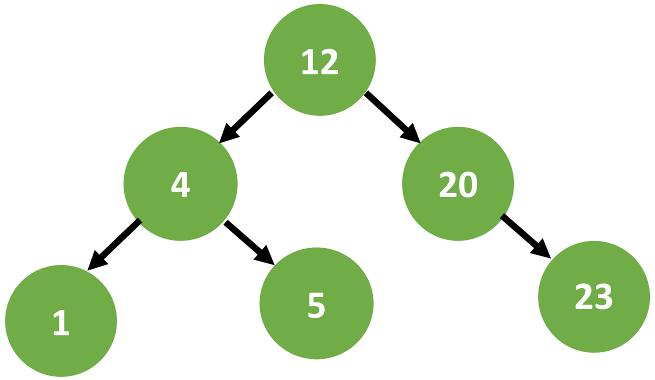
- Binary search trees
- Every node in the tree can have at most 2 children (left child and right child)
  - left child is smaller than the parent node
  - right child is greater than the parent node



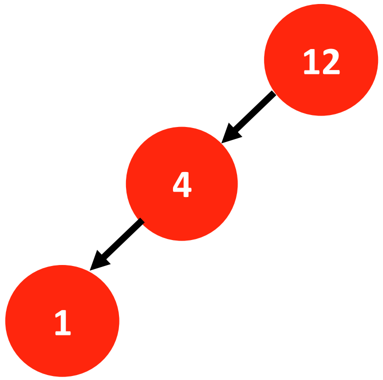
- The new data is placed in sorted order so that the search and other operations can use the principle of binary search with O(logn) running time

Module 9 – Trees





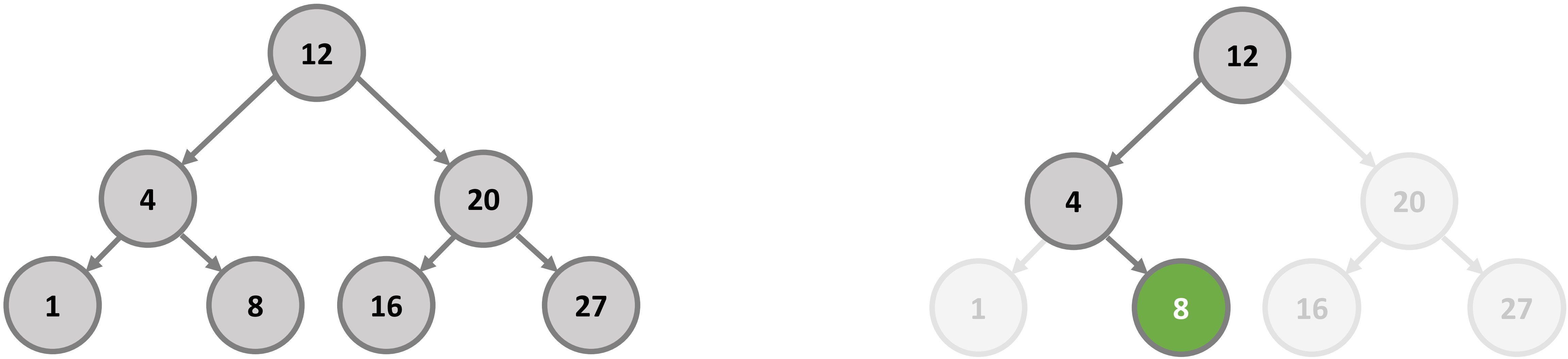
BALANCED TREE



IMBALANCED TREE

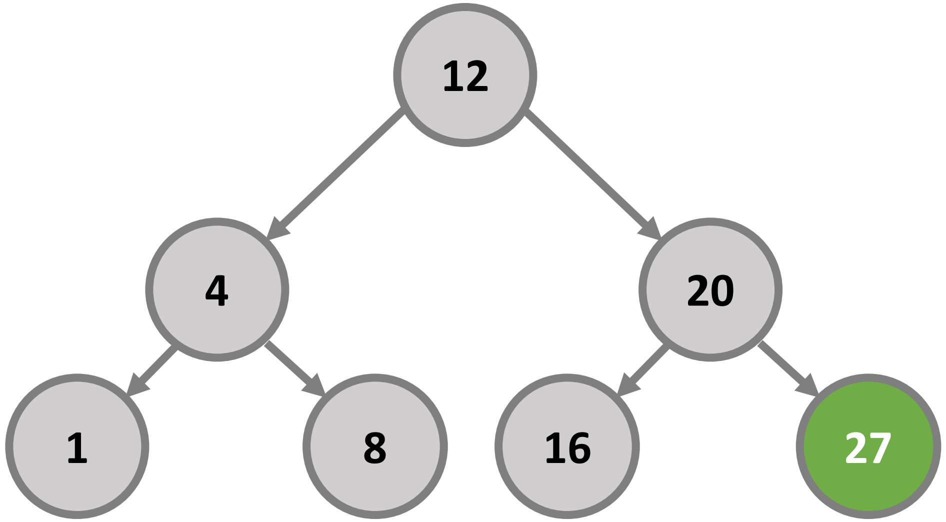
Module 9 – Trees

Search(8)



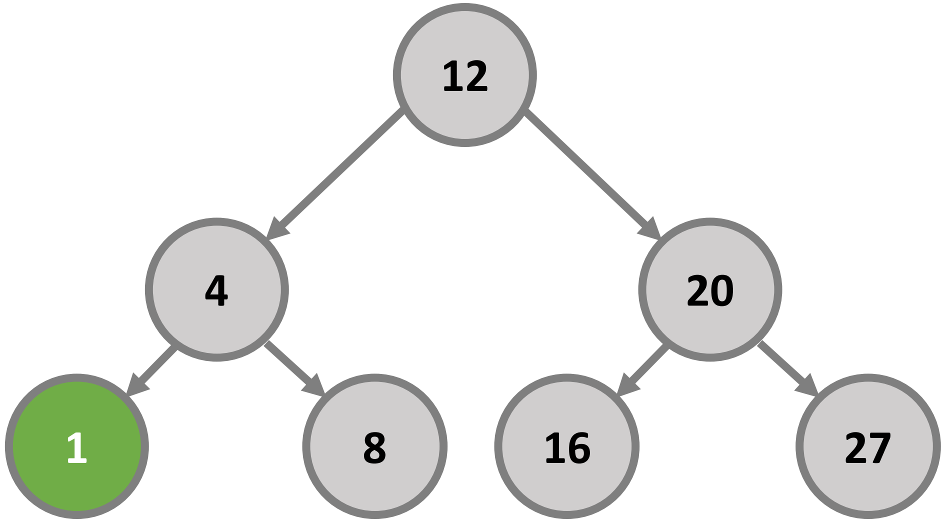
Module 9 – Trees

Search max()



the **maximum** item in the binary search tree is the **rightmost** item in the tree

Search min()

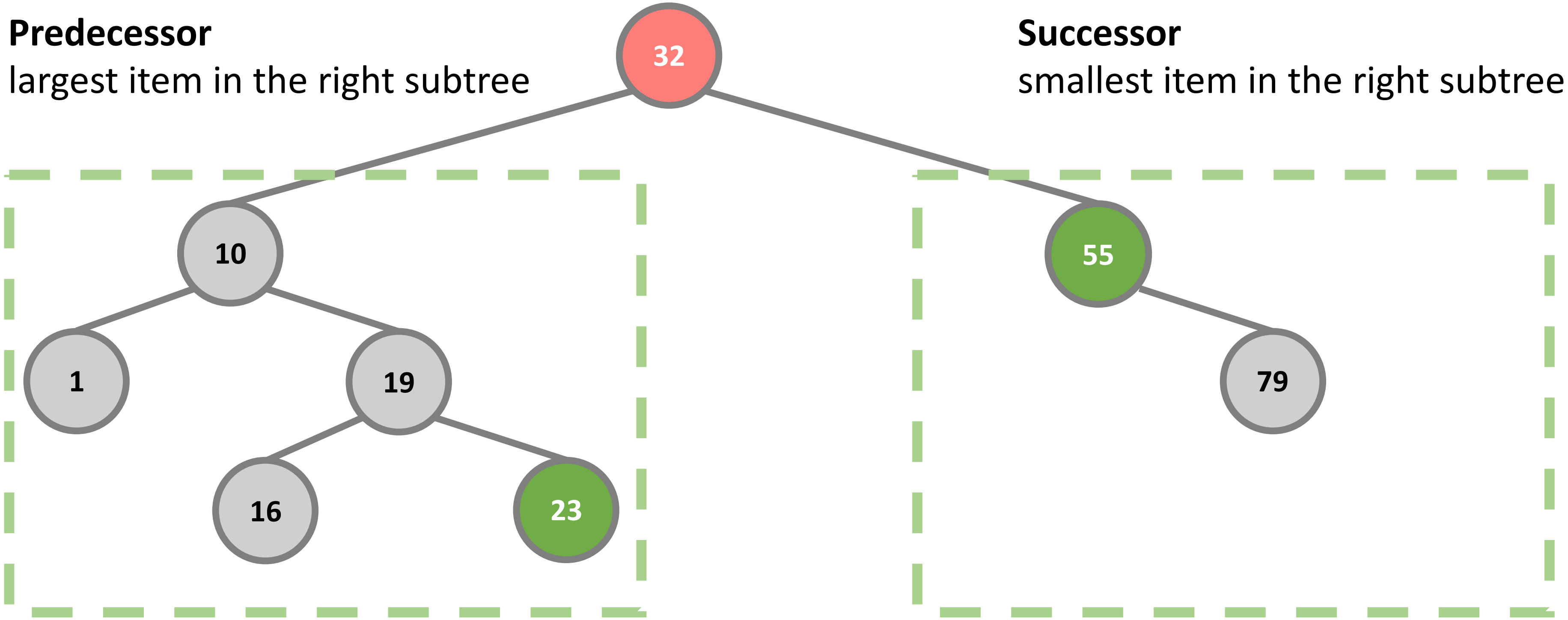


the **minimum** item in the binary search tree is the **leftmost** item in the tree

Module 9 – Trees

Predecessor
largest item in the right subtree

Successor
smallest item in the right subtree



Module 9 – Trees

Tree traversal approaches

Pre-order

- Root node → left subtree → right subtree

32, 10, 1, 19, 16, 23, 55, 79

Post-order

- Left subtree → Right subtree → Root node

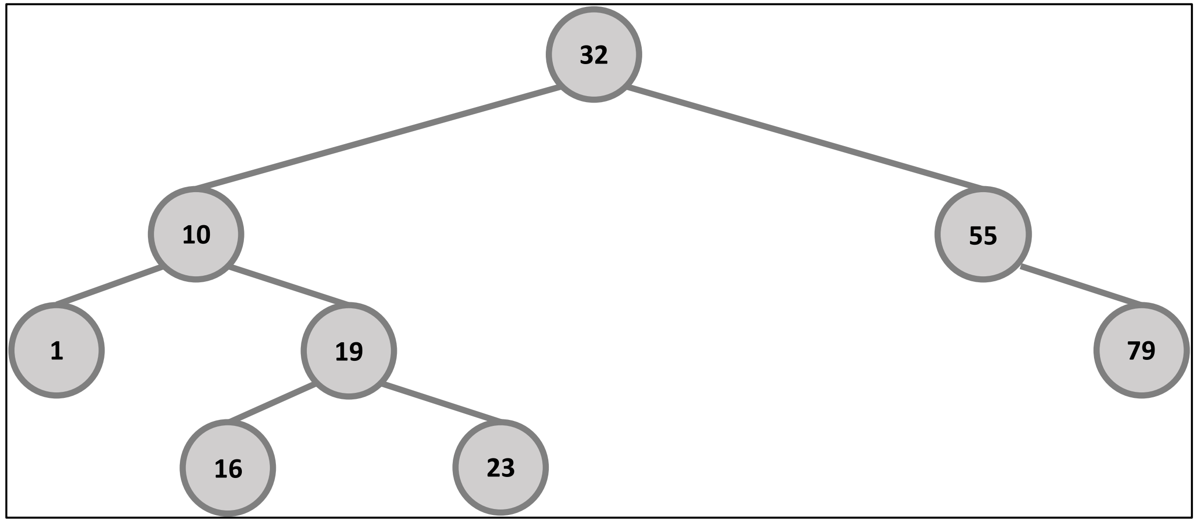
1, 16, 23, 19, 10, 79, 55, 32

In-order

- Left subtree → Root node → Right subtree

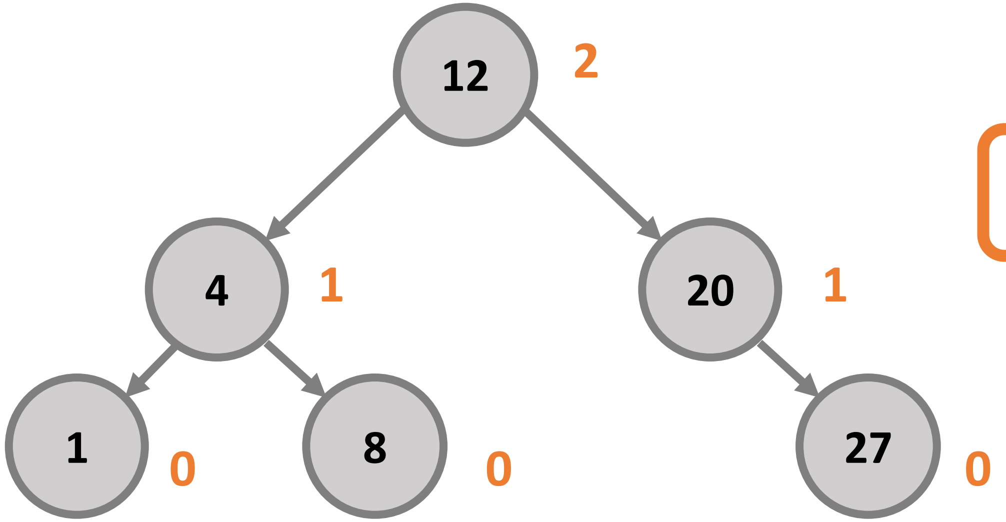
1, 10, 16, 19, 23, 32, 55, 79

- Returns elements in sorted order



Module 9 – Trees

- To determine whether the tree is balanced or not, we have to measure its height first and then calculate the balance factor
- Height of a tree (or a node) is the longest path from the root (or from the node) to a leaf node

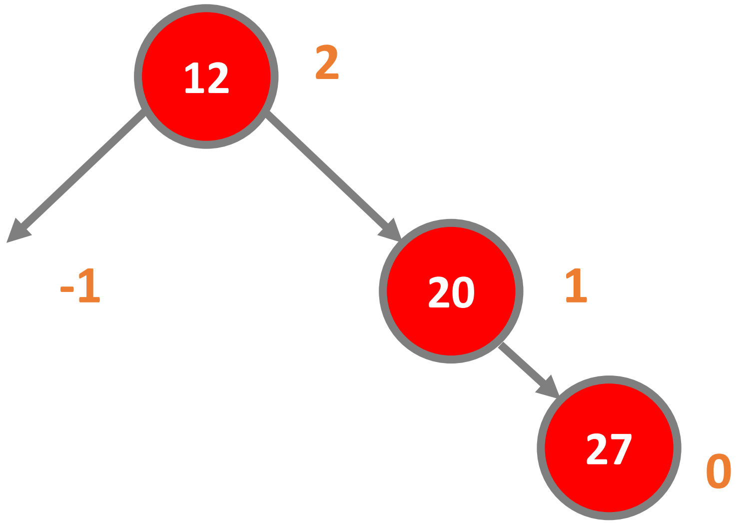


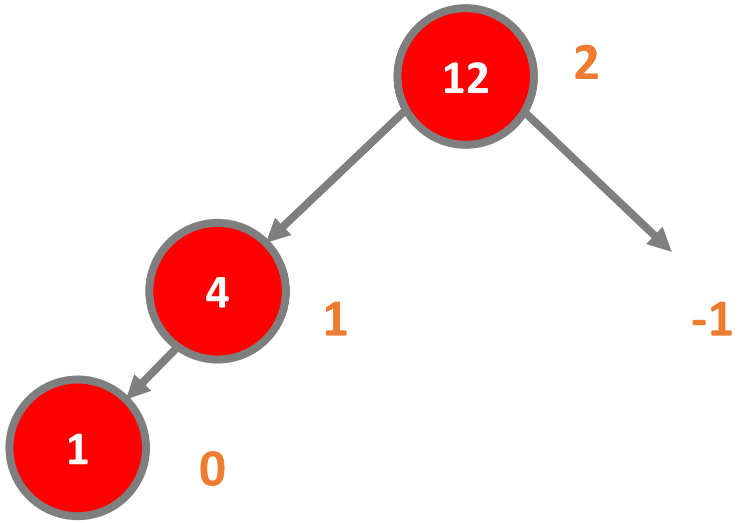
height = max (left child's height , right child's height) + 1

Balance factor: $\mathbf{h_{left} - h_{right}}$

The height of a NULL node is -1
➔ leaf nodes have height 0.

Module 9 – Trees





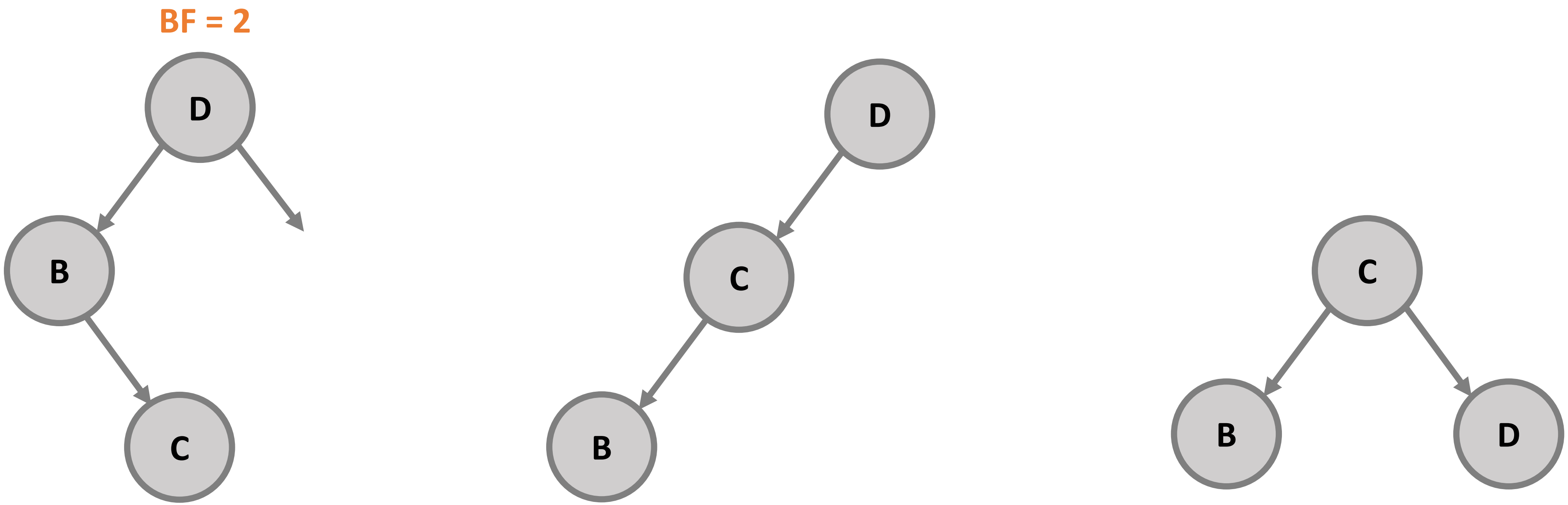
Balance factor: -**1 – 1 = -2**

Right-heavy case
→ Rotate **left** to balance

Balance factor: **1 – (–1) = 2**

Left-heavy case
→ Rotate **right** to balance

Module 9 – Trees



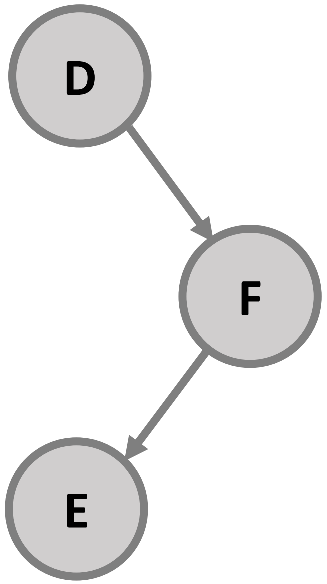
→left-right heavy

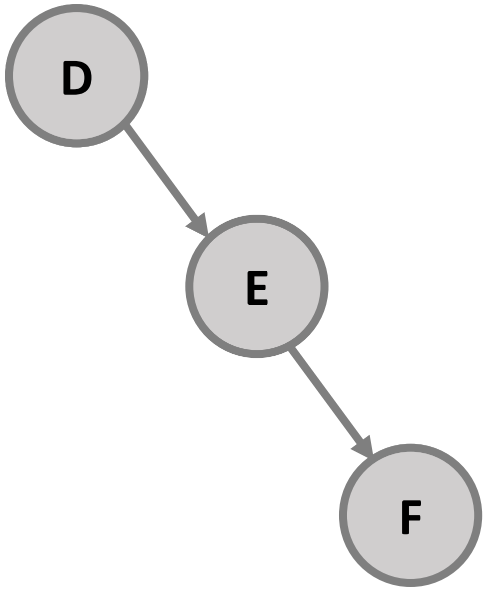
→ Rotate B left

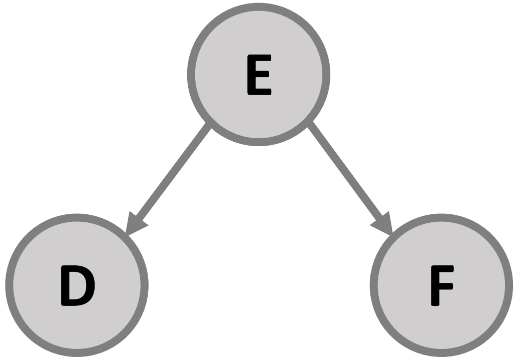
→ Rotate D right

Module 9 – Trees

BF = -2



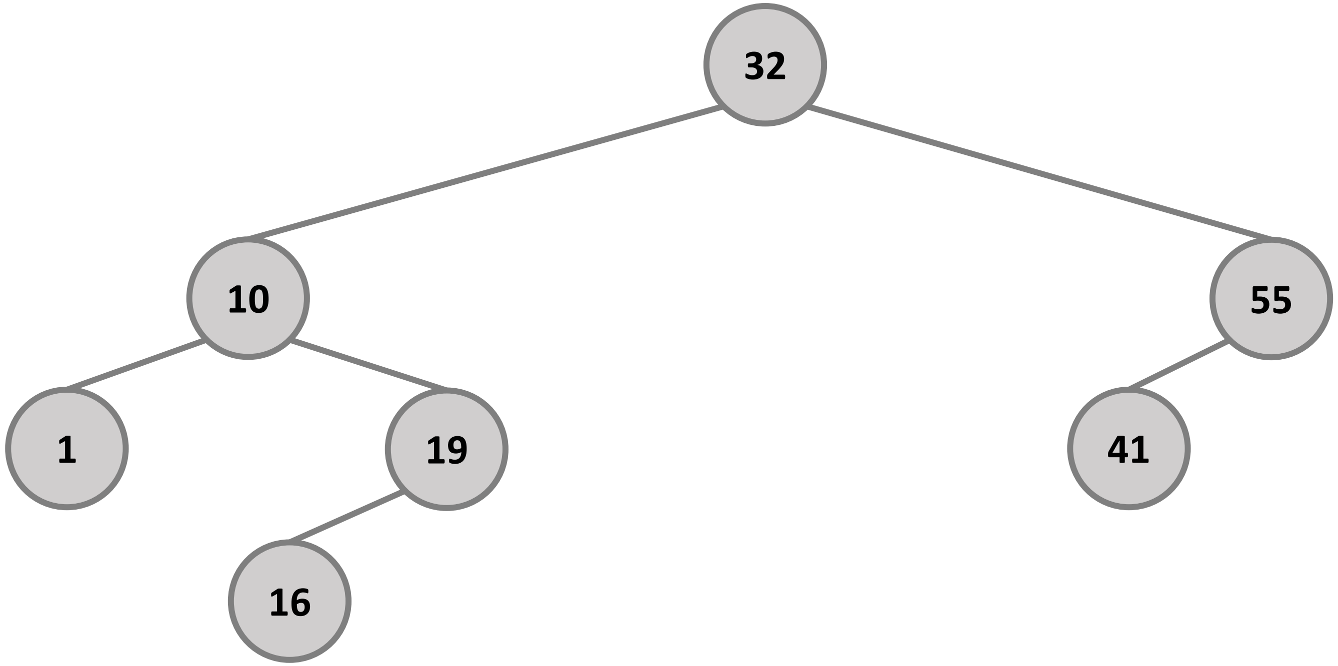




→right-left heavy
- → Rotate F right
- →Rotate D left

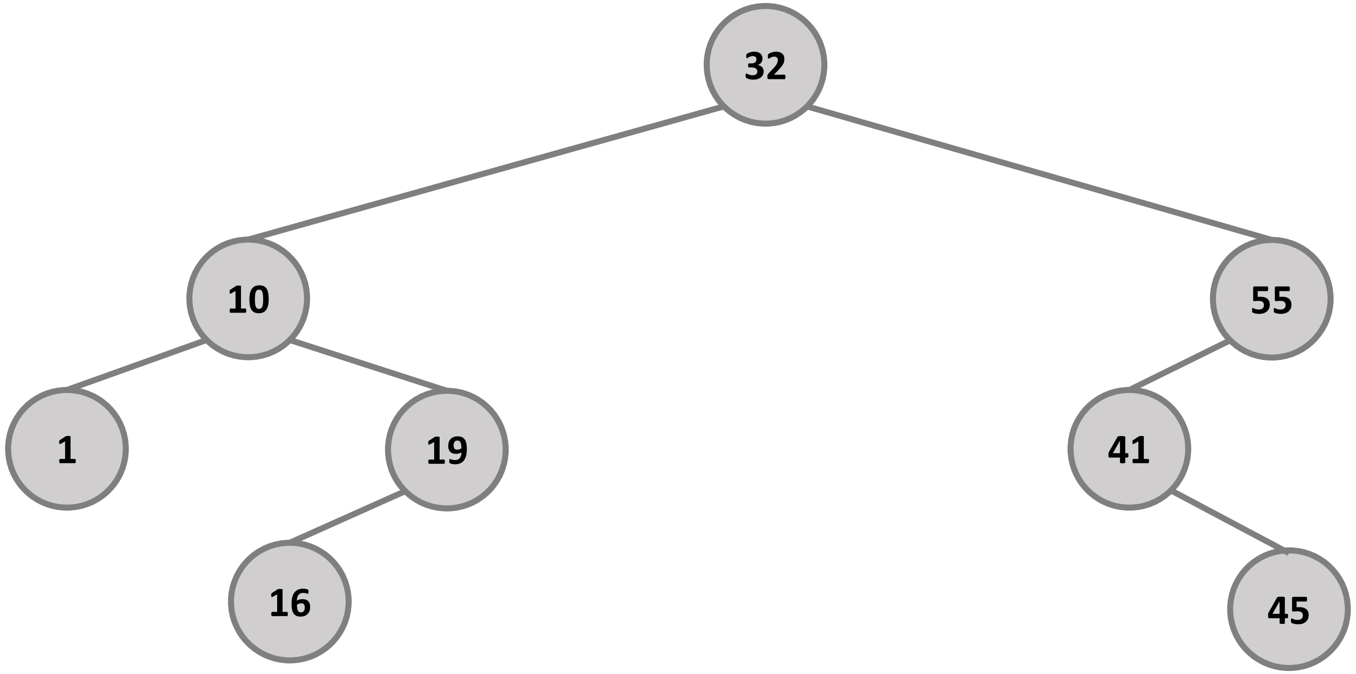
Module 9 – Trees

- Insertion in AVL trees. E.g., insert 45



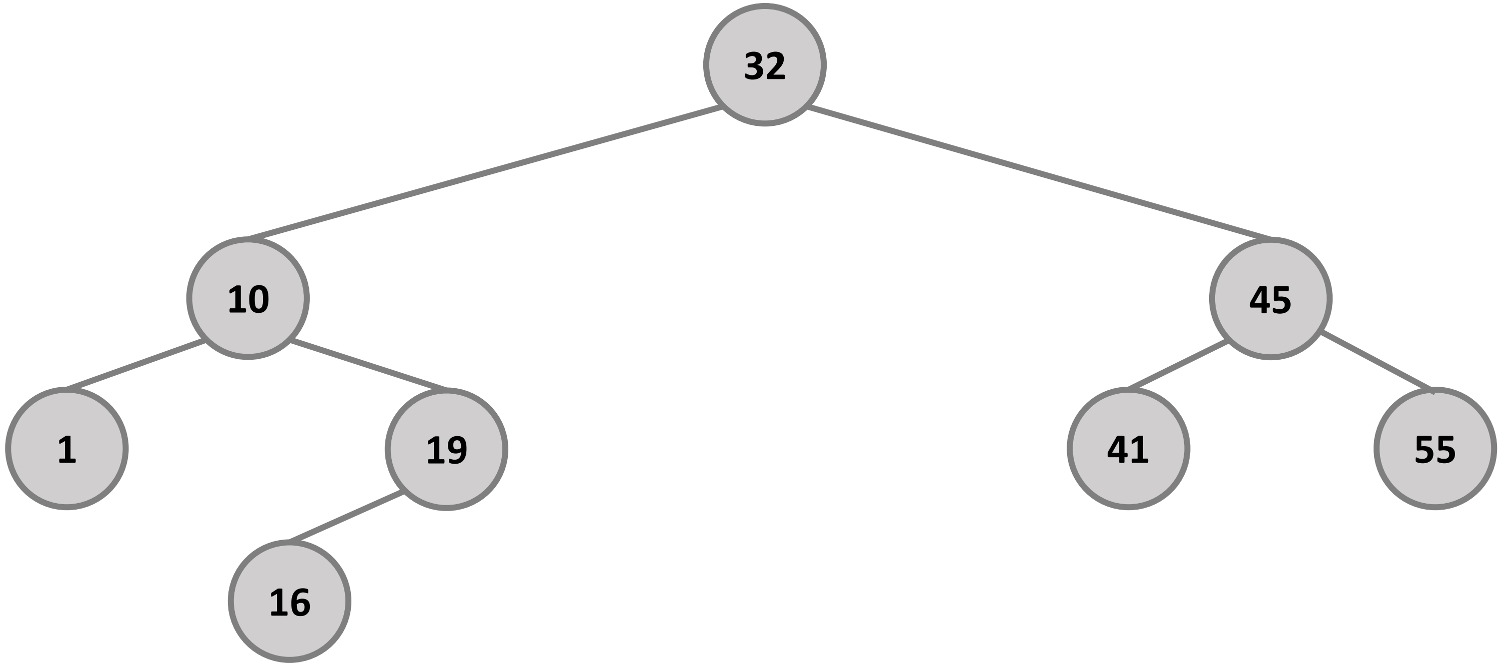
Module 9 – Trees

- Insertion in AVL trees. E.g., insert 45



Module 9 – Trees

- Insertion in AVL trees



Module 10 – Priority Queues

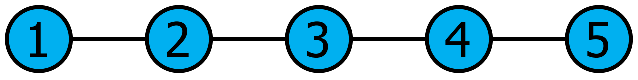
- A priority queue stores a collection of items in a (key, value) pair
  - Key → defines the priority
  - Value → the actual data

- Item with the highest priority is dequeued first

Module 10 – Priority Queues

Unsorted List

Sorted List



Unsorted List		Sorted List	
`insert(key, value)`	→ O(1)	`insert(key, value)`	→ O(n)
`findmin()`	→ O(n)	`findmin()`	→ O(1)
`removemin()`	→ O(n)	`removemin()`	→ O(n)

Module 10 – Priority Queues

Unsorted List		Sorted List	
			
`insert(key, value)`	→ O(1)	`insert(key, value)`	→ O(n)
`findmin()`	→ O(n)	`findmin()`	→ O(1)
`removemin()`	→ O(n)	`removemin()`	→ O(1)

By storing the data in reverse order

Module 10 – Priority Queues

- **Heaps** are data structures that are used to implement priority queues (ADT)

- Most common implementation of heap → binary tree

- Standard term used for *priority* is “key”
  - Unlike **maps** data structure, keys (priority) in **heaps** can be same

- Heaps are <u>**complete**</u> data structure

  → BST → each node has left and right child

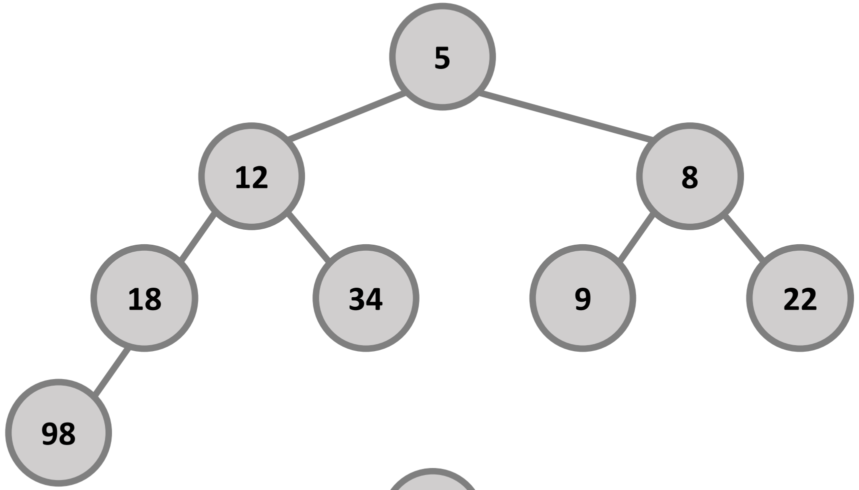
  → <u>Construct heap from left to right across each row</u>

  → Last row may not be fully completed

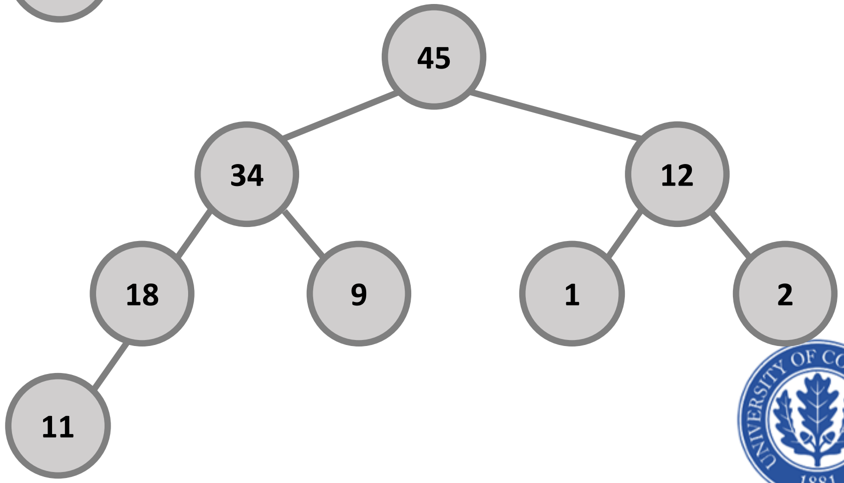
Module 10 – Priority Queues

Every node can have 2 children, so heaps are almost-complete binary trees.

min heap: the parent node is always smaller than the child nodes (left and right nodes)



max heap: the parent node is always greater than the child nodes (left and right nodes)



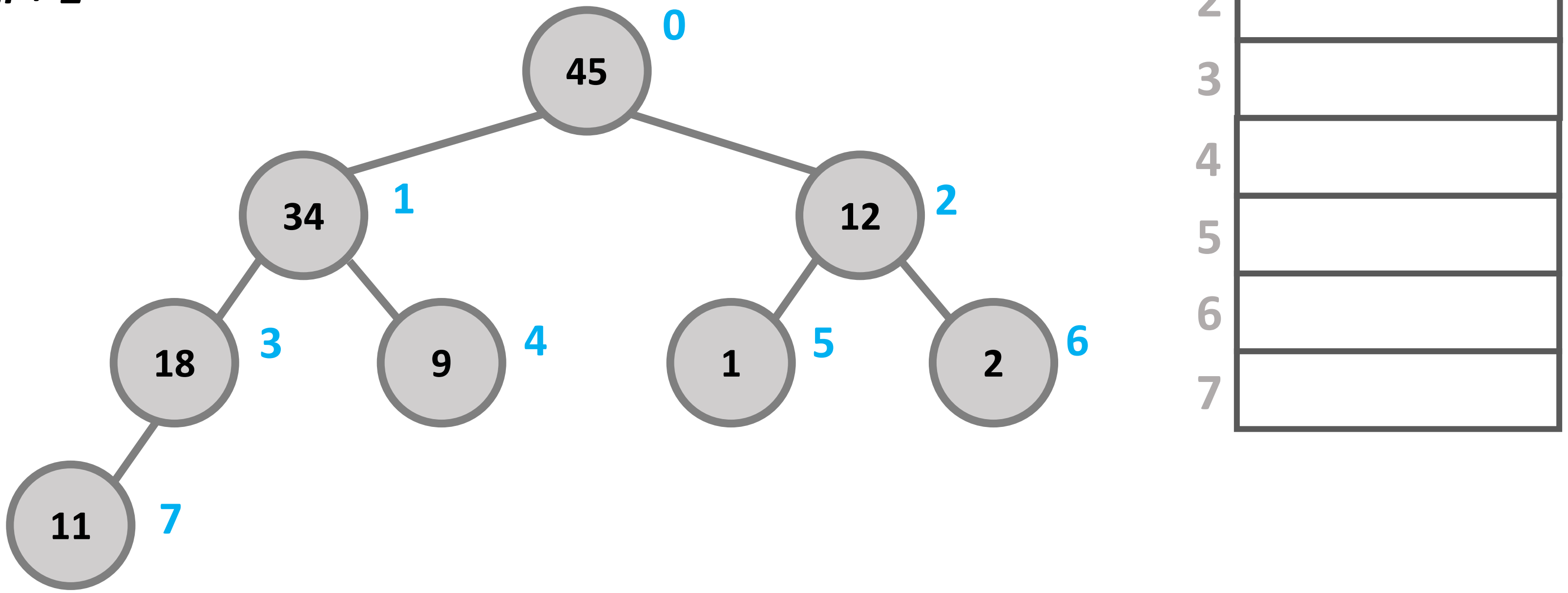
Module 10 – Priority Queues

Heaps can be represented in 1-D form

- Each element can be given an index value

Any node placed at index ***i*** has:

- **Left** child placed at index ***2i + 1***
- **Right** child placed at index ***2i + 2***



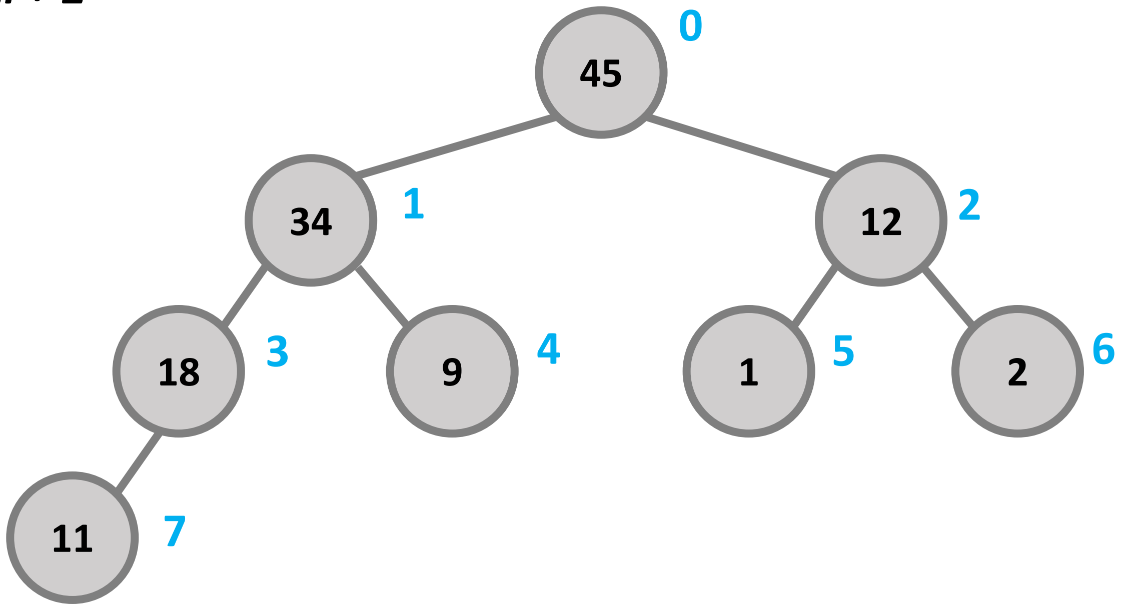
Module 10 – Priority Queues

Heaps can be represented in 1-D form

- Each element can be given an index value

Any node placed at index ***i*** has:

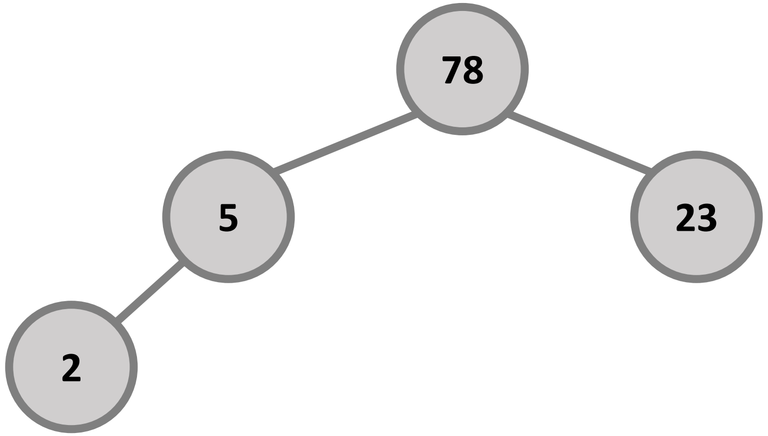
- **Left** child placed at index ***2i + 1***
- **Right** child placed at index ***2i + 2***



0	45
1	34
2	12
3	18
4	9
5	1
6	2
7	11

Module 10 – Priority Queues

Insert(92)

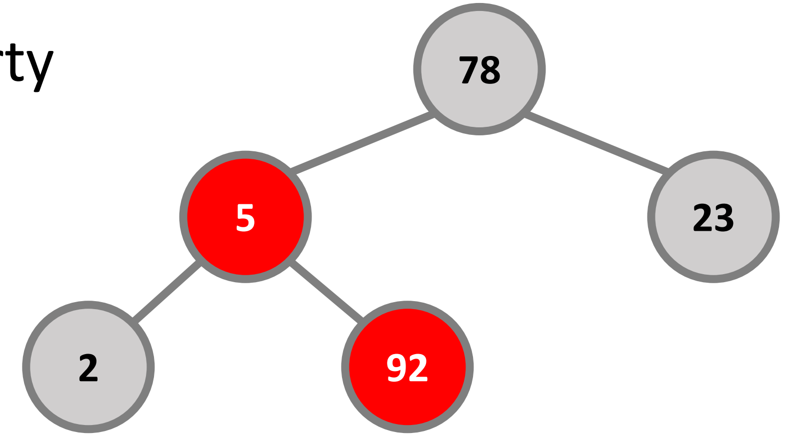


0	78
1	5
2	23
3	2
4	
5	
6	
7	

Module 10 – Priority Queues

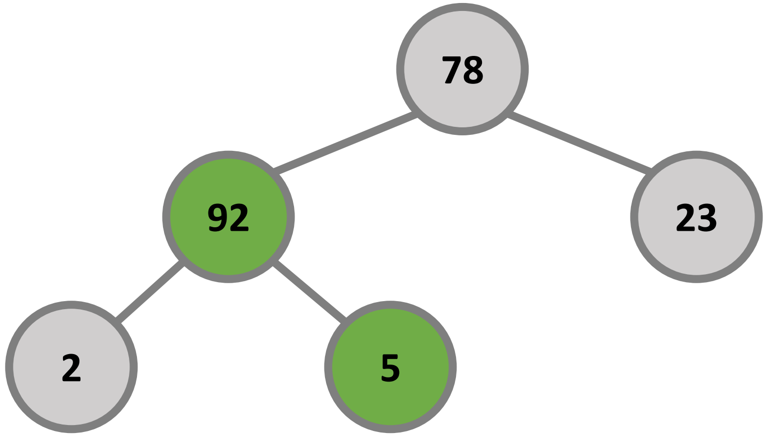
Insert(92)

Violating max heap property



0	78
1	5
2	23
3	2
4	92
5	
6	
7	

Module 10 – Priority Queues

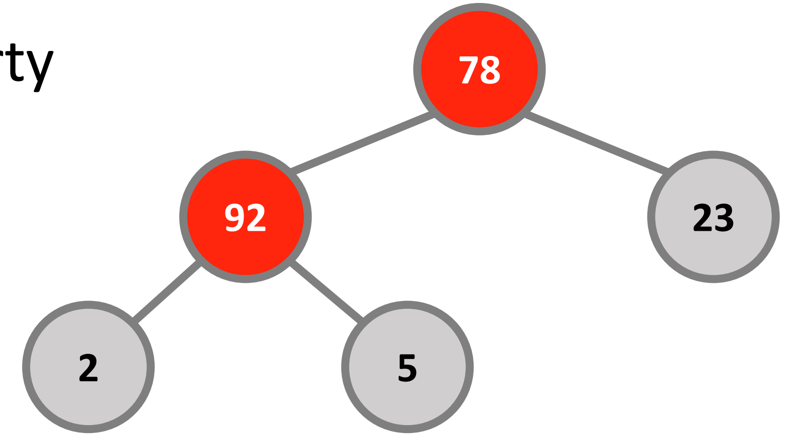


0	78
1	92
2	23
3	2
4	5
5	
6	
7	

Module 10 – Priority Queues

Insert(92)

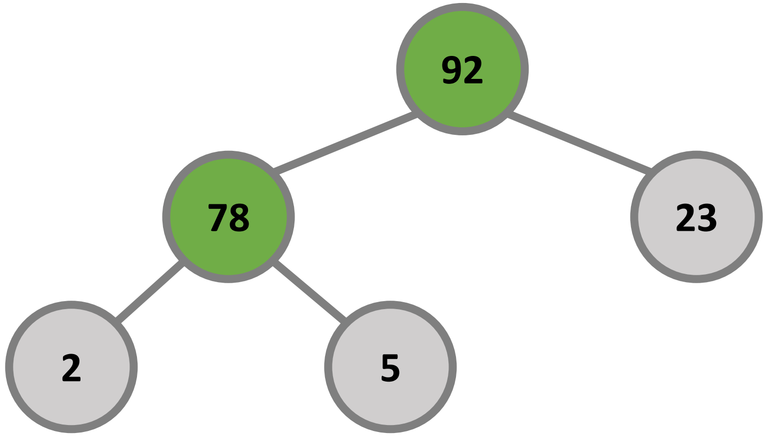
Violating max heap property



0	78
1	92
2	23
3	2
4	5
5	
6	
7	

Module 10 – Priority Queues

Insert(92)



0	92
1	78
2	23
3	2
4	5
5	
6	
7	

Module 10 – Priority Queues

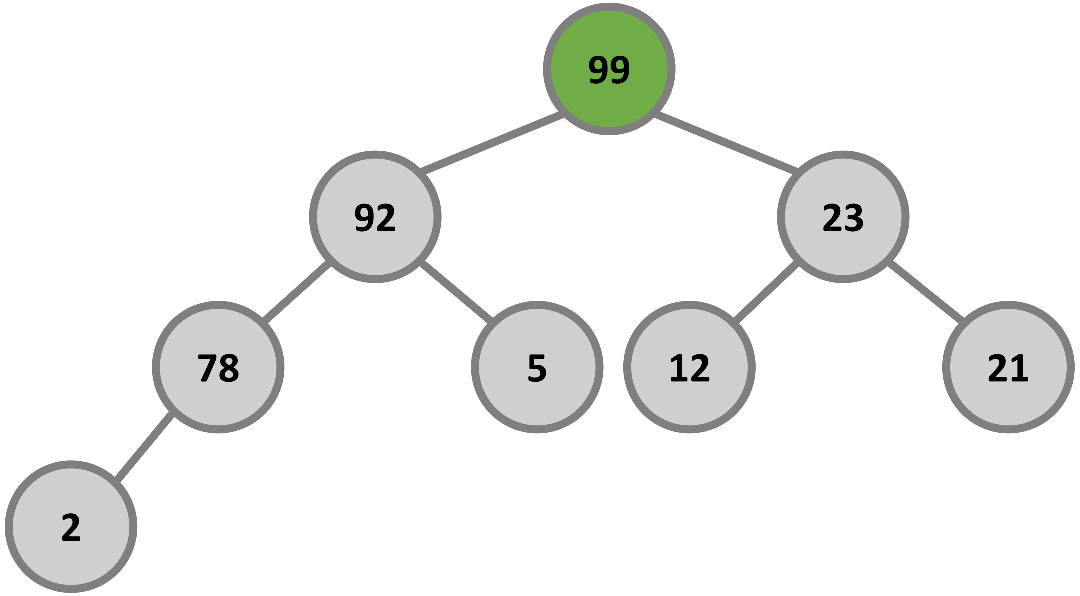
```
insert(A, 23)
insert(B, 5)
insert(C, 78)
insert(D, 2)
insert(E, 29)
insert(F, 12)
insert(G, 14)
insert(H, 1)
```

Show how the following "**Min**" heap array will be filled up when the `insert` method is executed periodically on the given data. Also indicate where heap violation occurs (X) and where it is finally fixed (✓) as shown in the example below.

		(X)	(✓)		(X)		(✓)		(X)	(✓)		(X)			(✓)
0	A, 23	A, 23	B, 5	B, 5	B, 5	B, 5	D, 2	D, 2	D, 2	D, 2	D, 2	D, 2	D, 2	D, 2	H, 1
1		B, 5	A, 23	A, 23	A, 23	D, 2	B, 5	B, 5	B, 5	B, 5	B, 5	B, 5	B, 5	H, 1	D, 2
2				C, 78	C, 78	C, 78	C, 78	C, 78	C, 78	F, 12	F, 12	F, 12	F, 12	F, 12	F, 12
3					D, 2	A, 23	A, 23	A, 23	A, 23	A, 23	A, 23	A, 23	H, 1	B, 5	B, 5
4								E, 29	E, 29	E, 29	E, 29	E, 29	E, 29	E, 29	E, 29
5									F, 12	C, 78	C, 78	C, 78	C, 78	C, 78	C, 78
6											G, 14	G, 14	G, 14	G, 14	G, 14
7												H, 1	A, 23	A, 23	A, 23

Module 10 – Priority Queues

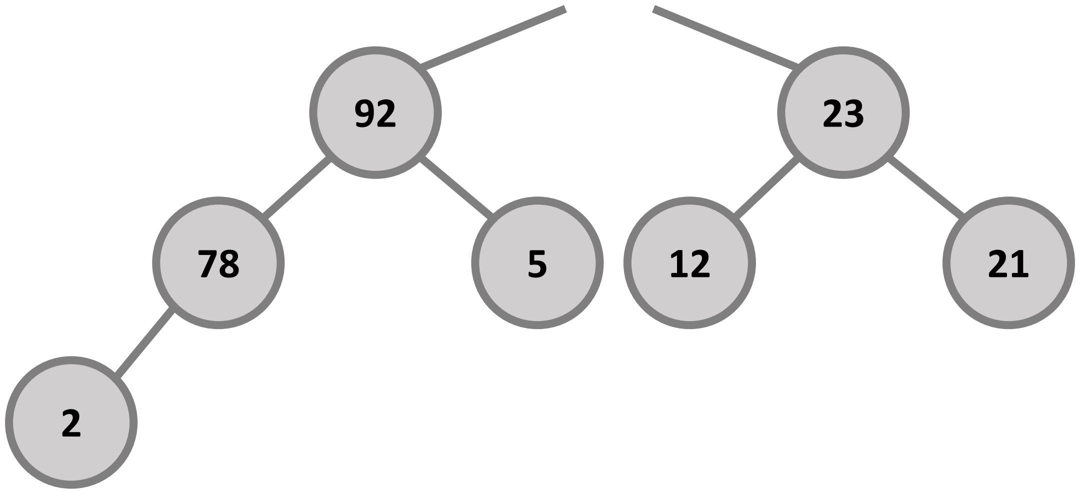
RemoveMax()



0	99
1	92
2	23
3	78
4	5
5	12
6	21
7	2

Module 10 – Priority Queues

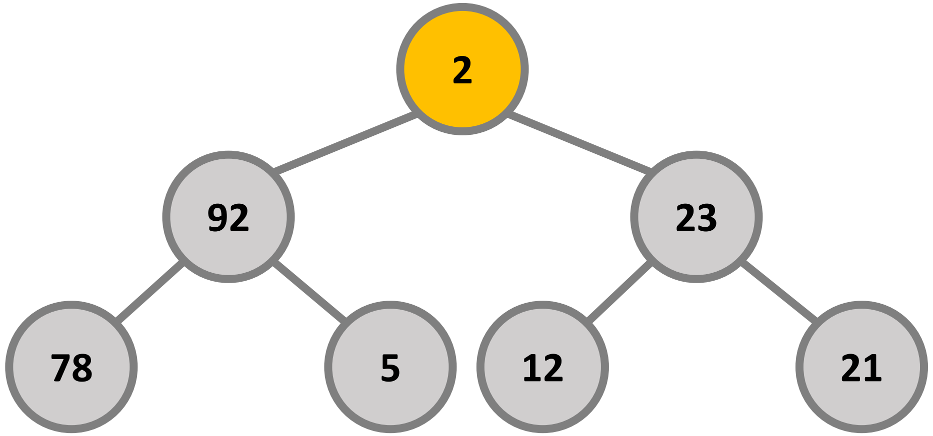
RemoveMax()



0	
1	92
2	23
3	78
4	5
5	12
6	21
7	2

Module 10 – Priority Queues

RemoveMax()

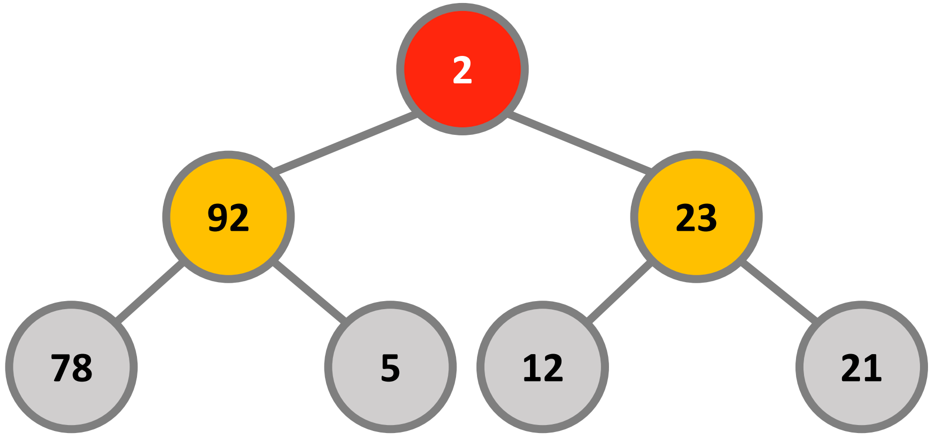


0	2
1	92
2	23
3	78
4	5
5	12
6	21
7	

- Replaced the empty element with the last element

Module 10 – Priority Queues

RemoveMax()



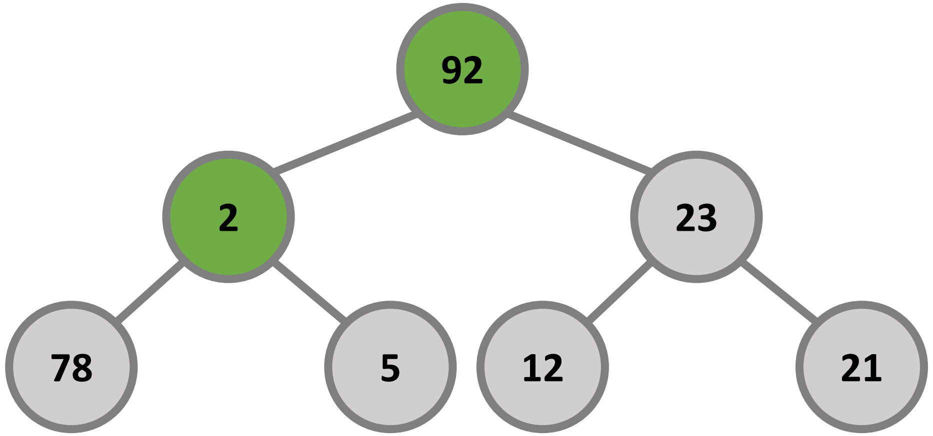
0	2
1	92
2	23
3	78
4	5
5	12
6	21
7	

- Now check for property violation starting from the root node to the lead node
- → **Heapify operation**

- Compare both children and swap with the greatest one

Module 10 – Priority Queues

RemoveMax()

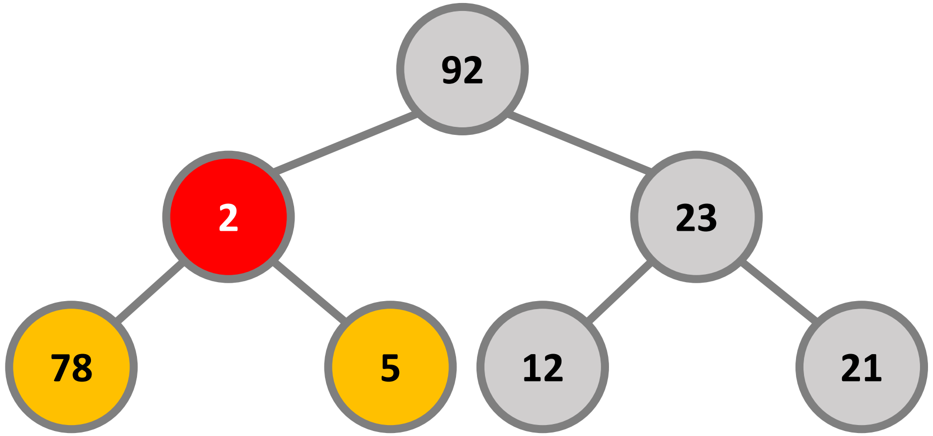


0	92
1	2
2	23
3	78
4	5
5	12
6	21
7	

- Now check for property violation starting from the root node to the lead node
- → **Heapify operation**

Module 10 – Priority Queues

RemoveMax()



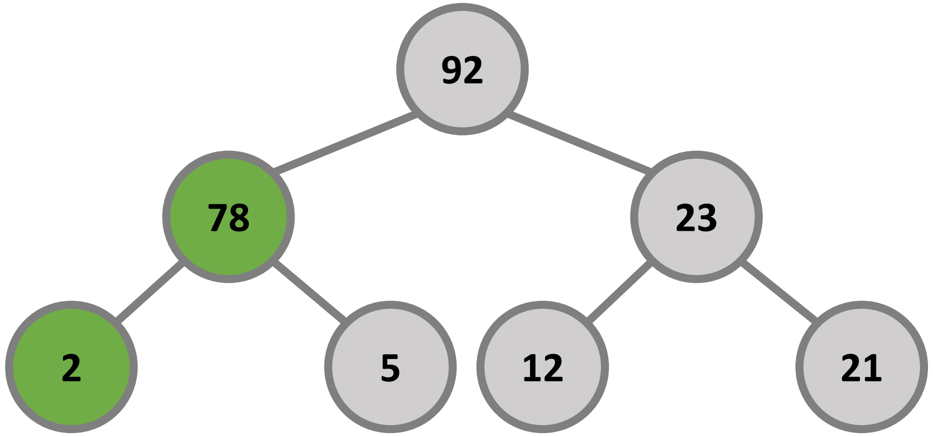
0	92
1	2
2	23
3	78
4	5
5	12
6	21
7	

- Now check for property violation starting from the root node to the lead node
- → **Heapify operation**

- Compare both children and swap with the greatest one

Module 10 – Priority Queues

RemoveMax()



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

- Now check for property violation starting from the root node to the lead node
- → **Heapify operation**

- Compare both children and swap with the greatest one

Module 10 – Priority Queues

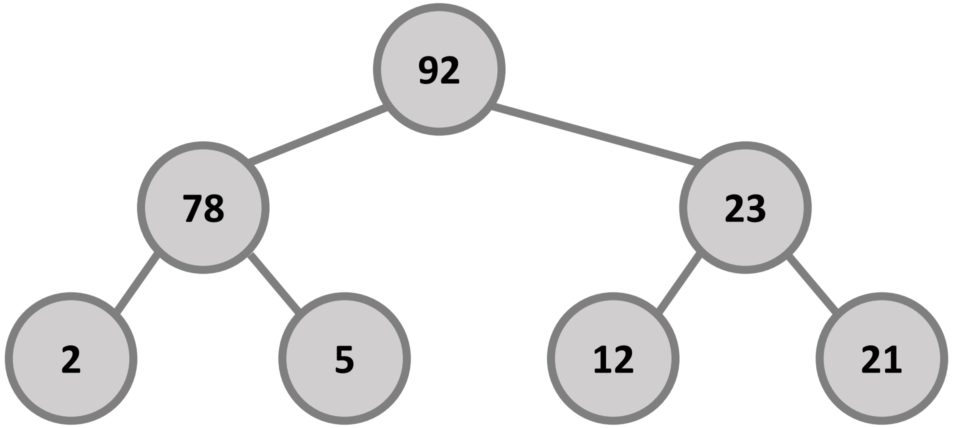
- Removing the root node (and usually this is the case) can be done in O(logn) running time

- Remove an arbitrary item
  - Finding it in the array requires O(n) and then we can remove it in O(logn)

➔ Removing arbitrary item requires O(n) time complexity

Module 10 – Priority Queues

REMOVE(12)

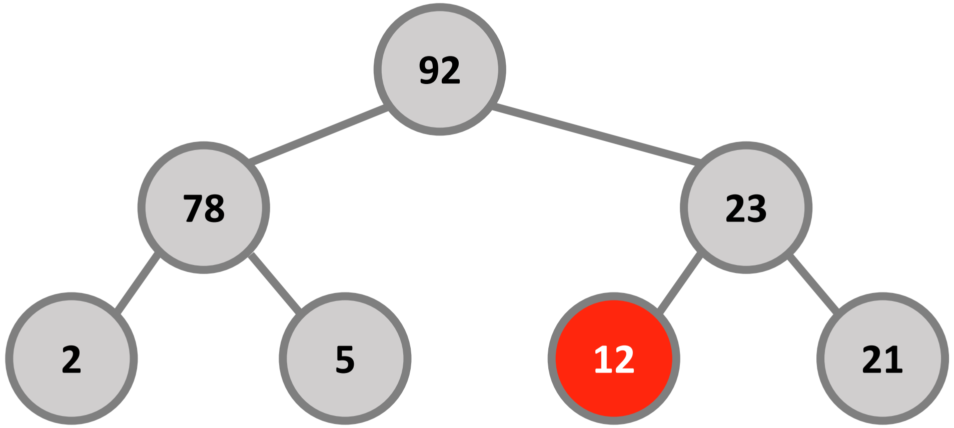


0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Module 10 – Priority Queues

REMOVE(12)

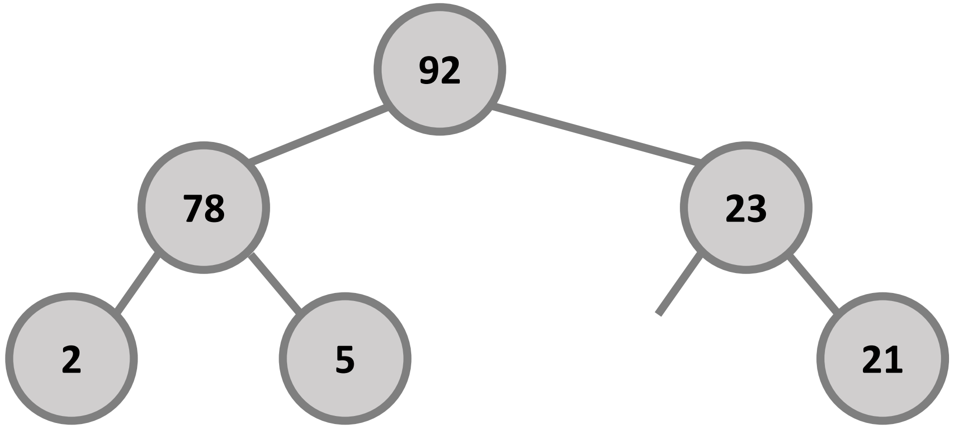
Finding it in the array requires O(n)



0	92
1	78
2	23
3	2
4	5
5	12
6	21
7	

Module 10 – Priority Queues

REMOVE(12)

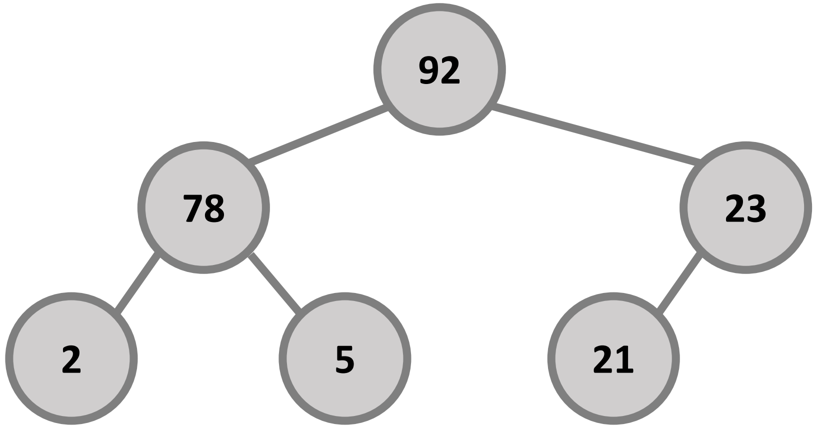


0	92
1	78
2	23
3	2
4	5
5	
6	21
7	

- There can not be a **hole** in the data structure
- Replace it with the last item in the heap

Module 10 – Priority Queues

REMOVE(12)



0	92
1	78
2	23
3	2
4	5
5	21
6	
7	

- There can not be a **hole** in the data structure
- Replace it with the last item in the heap

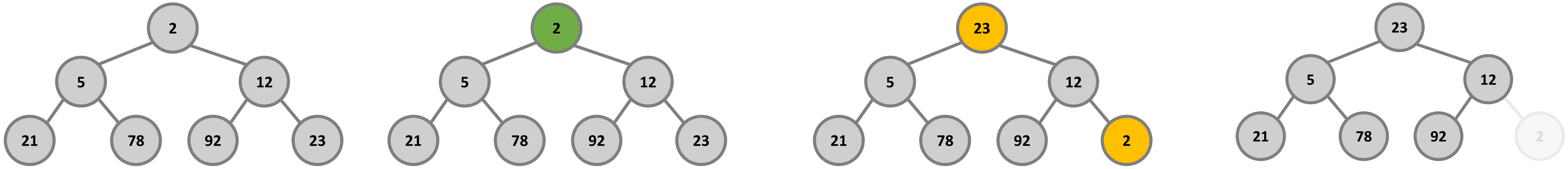
- After swapping, check for property violation all the way up till the root node → O(logn)

Removing arbitrary item: O(n) + O(logn) = O(n)

Module 10 – Priority Queues

Heapsort

1. Read the root node
2. Swap the root node with the last item
3. Heapify to abide by Heap properties
4. Repeat steps 1 and 2 for all nodes (except the last items which have already been read)

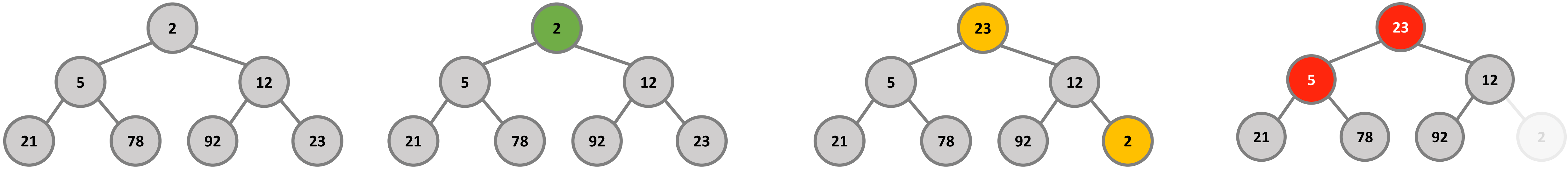


[2]

Module 10 – Priority Queues

Heapsort

1. Read the root node
2. Swap the root node with the last item
3. Heapify to abide by Heap properties
4. Repeat steps 1 and 2 for all nodes (except the last items which have already been read)

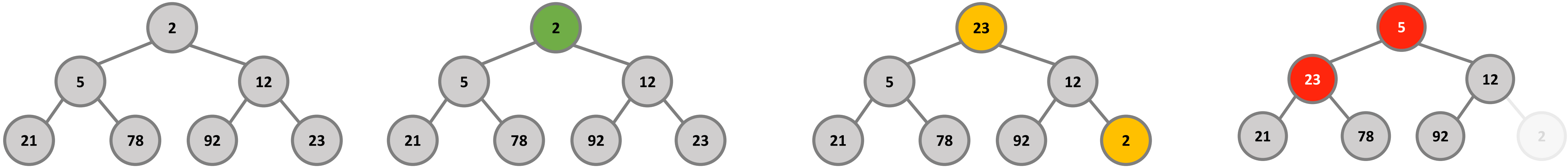


[2]

Module 10 – Priority Queues

Heapsort

1. Read the root node
2. Swap the root node with the last item
3. Heapify to abide by Heap properties
4. Repeat steps 1 and 2 for all nodes (except the last items which have already been read)

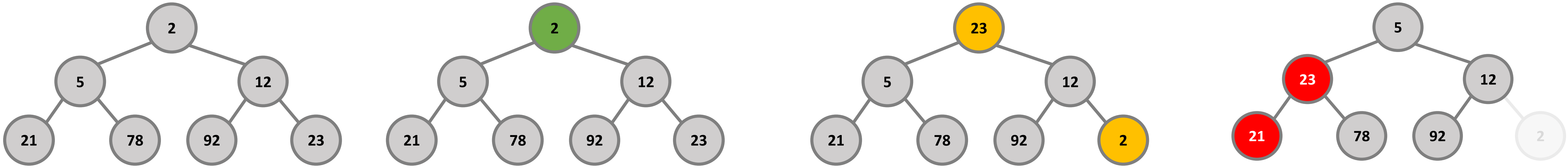


[2]

Module 10 – Priority Queues

Heapsort

1. Read the root node
2. Swap the root node with the last item
3. Heapify to abide by Heap properties
4. Repeat steps 1 and 2 for all nodes (except the last items which have already been read)

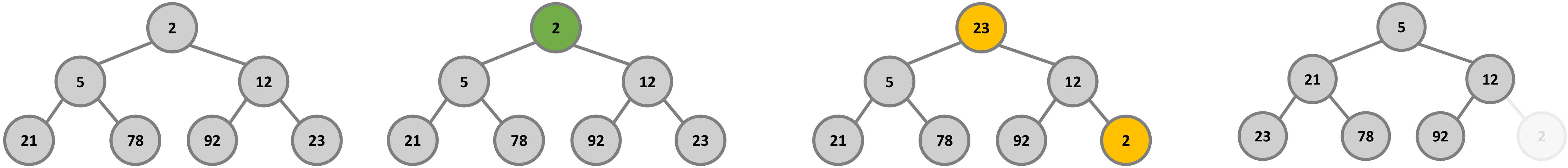


[2]

Module 10 – Priority Queues

Heapsort

1. Read the root node
2. Swap the root node with the last item
3. Heapify to abide by Heap properties
4. Repeat steps 1 and 2 for all nodes (except the last items which have already been read)

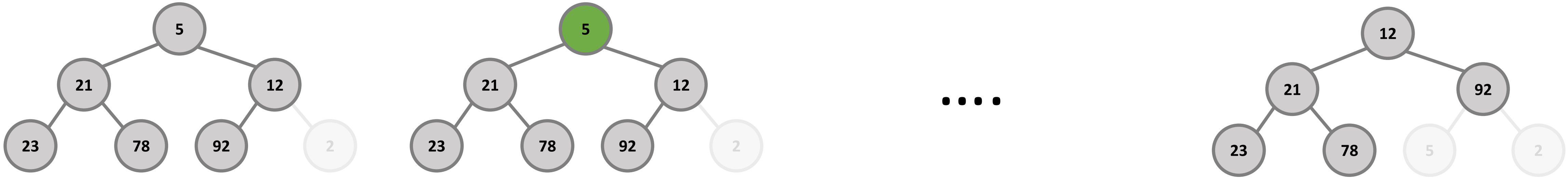


[2]

Module 10 – Priority Queues

Heapsort

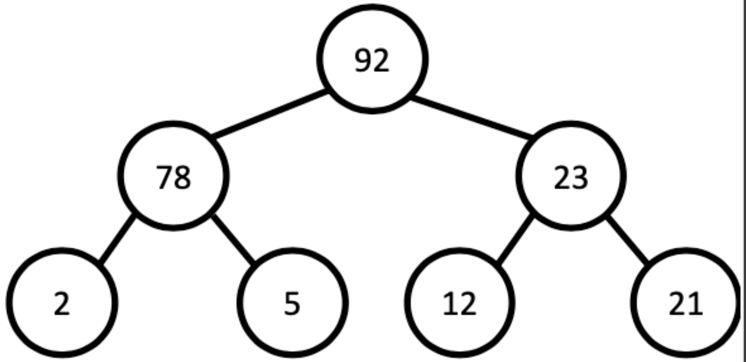
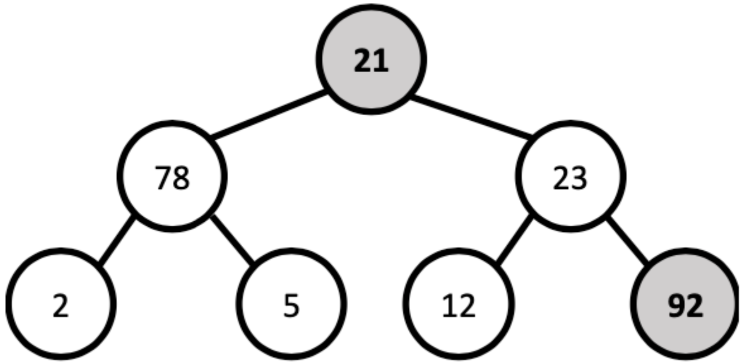
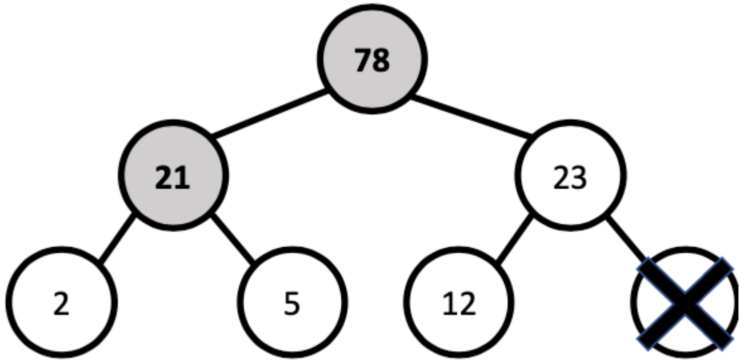
1. Read the root node
2. Swap the root node with the last item
3. Heapify to abide by Heap properties
4. Repeat steps 1 and 2 for all nodes (except the last items which have already been read)



[2, 5]

Module 10 – Priority Queues

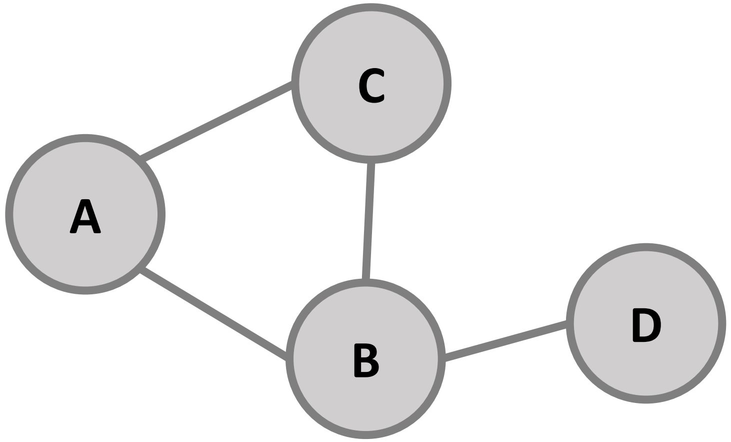
Go through this activity

	Read	Swap	After final Heapify
1.	 [92]		

Module 11 – Graphs

Graphs

- Edges can connect any vertex
- Any vertex can be accessed through any path



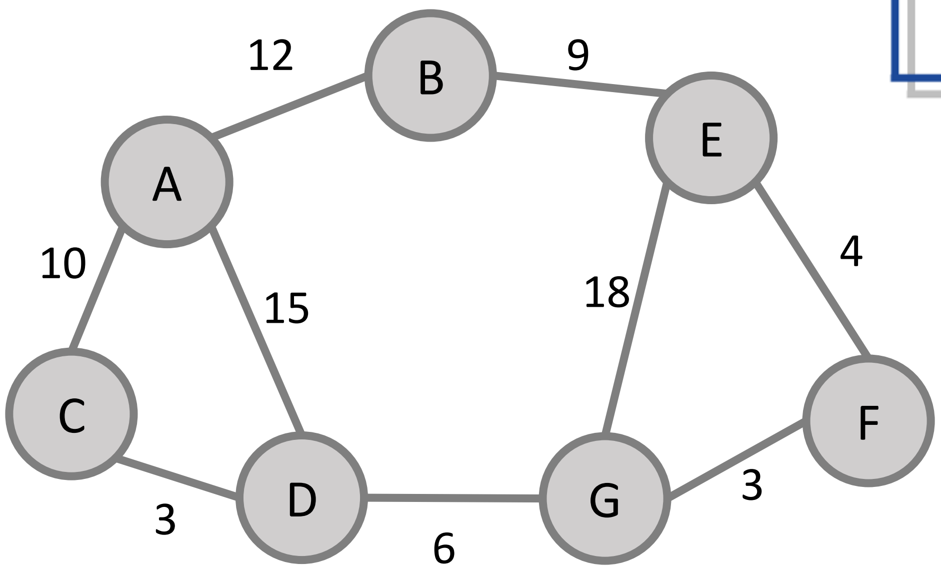
- Edges can be directed or undirected

- G = (V, E)
  - V = {A, B, C, D}
  - E = { {A, C}, {A, B}, {B, C}, {B,D} }

Module 11 – Graphs

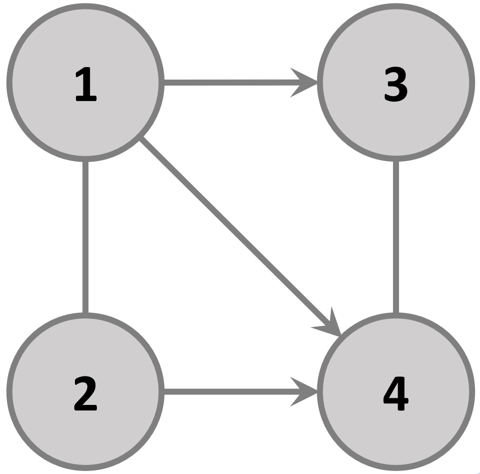
Graphs Terms and Types

- Cycle, Path,
- Directed, undirected, mixed, complete, weighted, in/out-degree



- Representation
  - Edge Set Representation

V = {1, 2, 3, 4}
E = { (1, 2), (1, 3), (1, 4),
(2, 1), (2, 4),
(3,4), (4,3) }



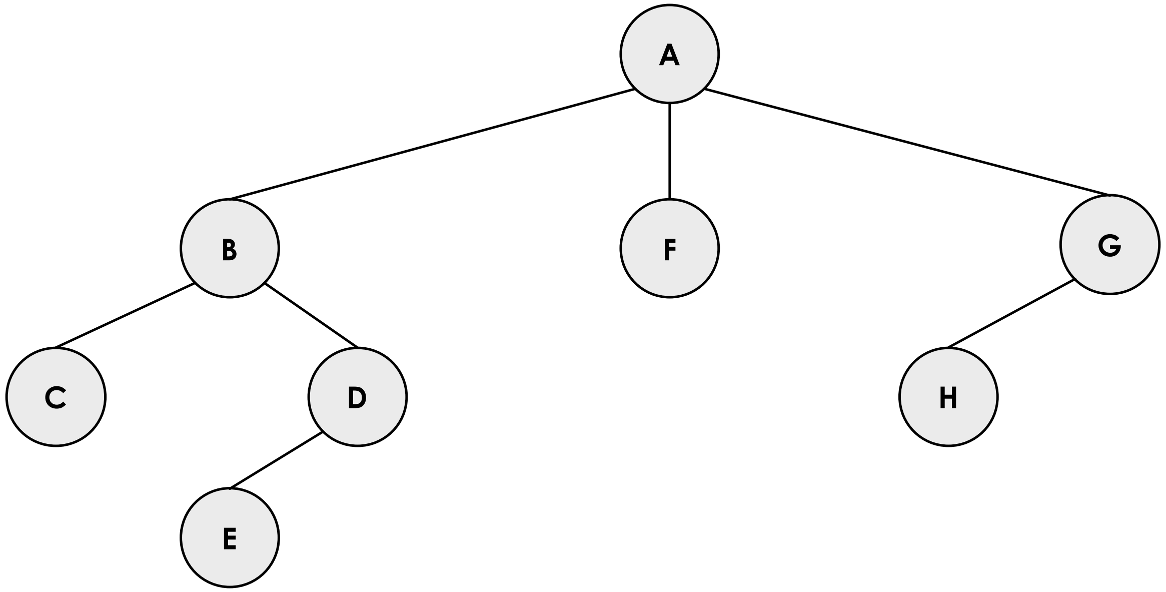
  - Adjacency Set Representation

V = {1, 2, 3, 4}
nbrs = { 1: {2, 3, 4},
2: {1, 4},
3: {4},
4: {3}, }

Module 11 – Graphs

Purpose of graph traversal is to visit all the nodes/vertices of a graph

- Breadth-First Search (BFS)
  → Visit order → A, B, F, G, C, D, H, E
  → Based on queue implementation

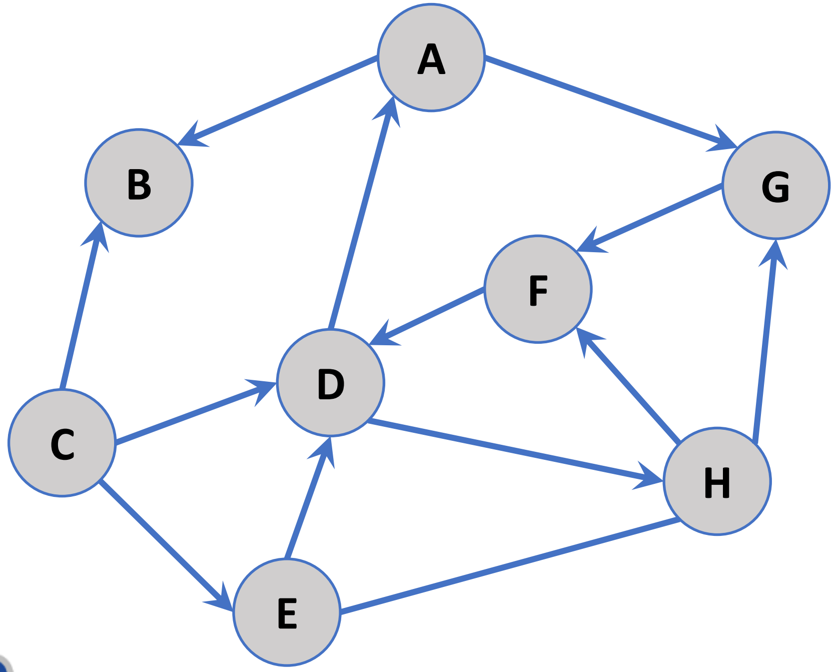


- Depth-First Search (DFS)
  → Visit order → A, B, C, D, E, F, G, H
  → Based on stack implementation

Module 11 – Graphs

Using BFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is the front of Q.
After dequeuing a vertex, enqueue (if not already in V) all its neighbors in Q in alphabetical order. Add them in V.

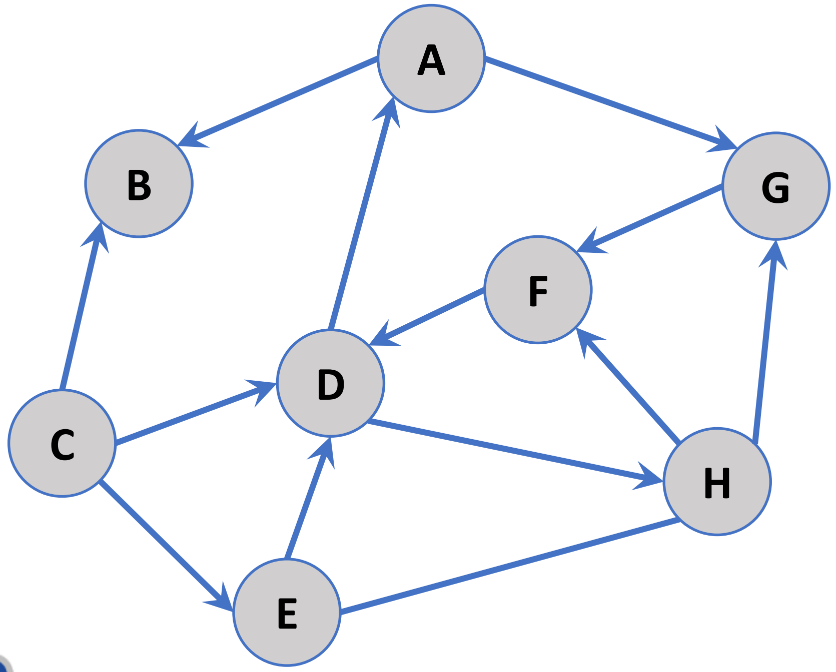


	Q	V
0	[C]	[C]
1	[B, D, E]	[C, B, D, E]
2		
3		
4		
5		
6		
7		
8		
9		
10		

Module 11 – Graphs

Using BFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is the front of Q.
After dequeuing a vertex, enqueue (if not already in V) all its neighbors in Q in alphabetical order. Add them in V.



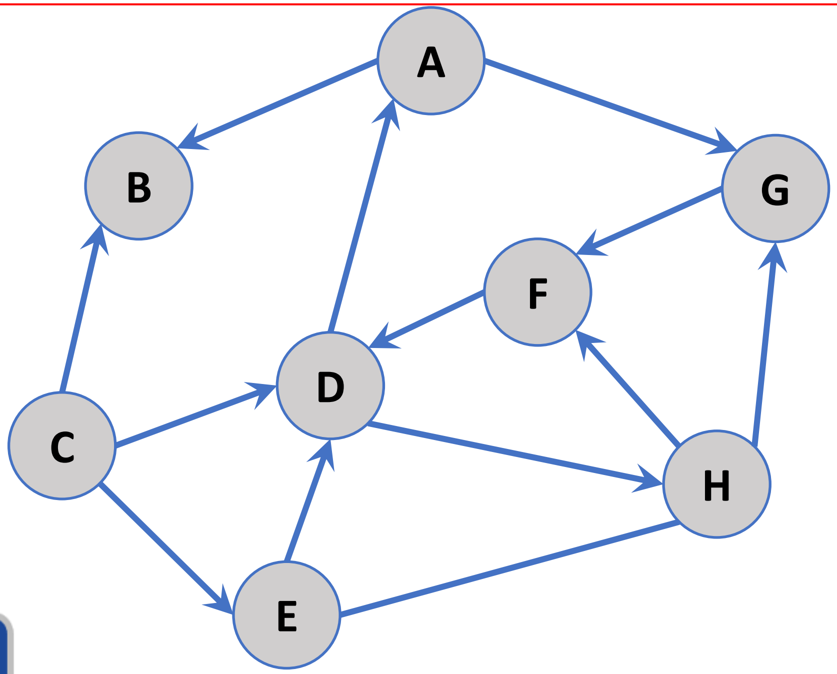
	Q	V
0	[C]	[C]
1	[B, D, E]	[C, B, D, E]
2	[D, E]	[C, B, D, E]
3	[E, A, H]	[C, B, D, E, A, H]
4	[A, H]	[C, B, D, E, A, H]
5	[H, G]	[C, B, D, E, A, H, G]
6	[G, F]	[C, B, D, E, A, H, G, F]
7	[F]	[C, B, D, E, A, H, G, F]
8	[]	[C, B, D, E, A, H, G, F]
9		
10		

Module 11 – Graphs

Using DFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is Top of the stack

After popping a vertex out, push its neighbors (if not already in V) in S in alphabetical order. Also, add popped element in V.



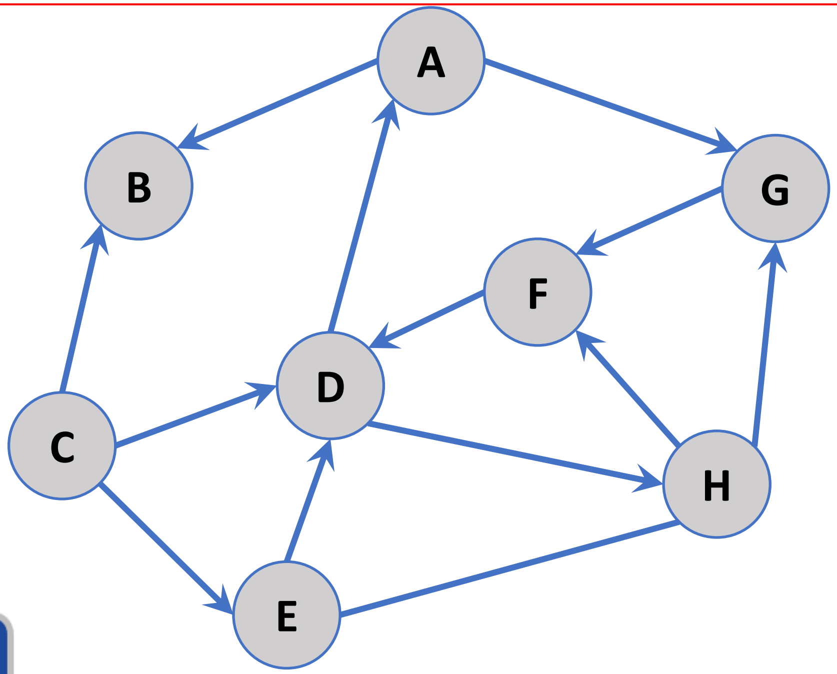
	S	V
0	[C]	[ ]
1	[B, D, E]	[C]
2		
3		
4		
5		
6		
7		
8		
9		
10		

Module 11 – Graphs

Using DFS, Starting from vertex C, visit all the vertices in alphabetical order and fill the following table.

First element is Top of the stack

After popping a vertex out, push its neighbors (if not already in V) in S in alphabetical order. Also, add popped element in V.



	S	V
0	[C]	[ ]
1	[B, D, E]	[C]
2	[D, E]	[C, B]
3	[A, H, E]	[C, B, D]
4	[G, H, E]	[C, B, D, A]
5	[F, H, E]	[C, B, D, A, G]
6	[H, E]	[C, B, D, A, G, F]
7	[E]	[C, B, D, A, G, F, H]
8	[]	[C, B, D, A, G, F, H, E]
9		
10		

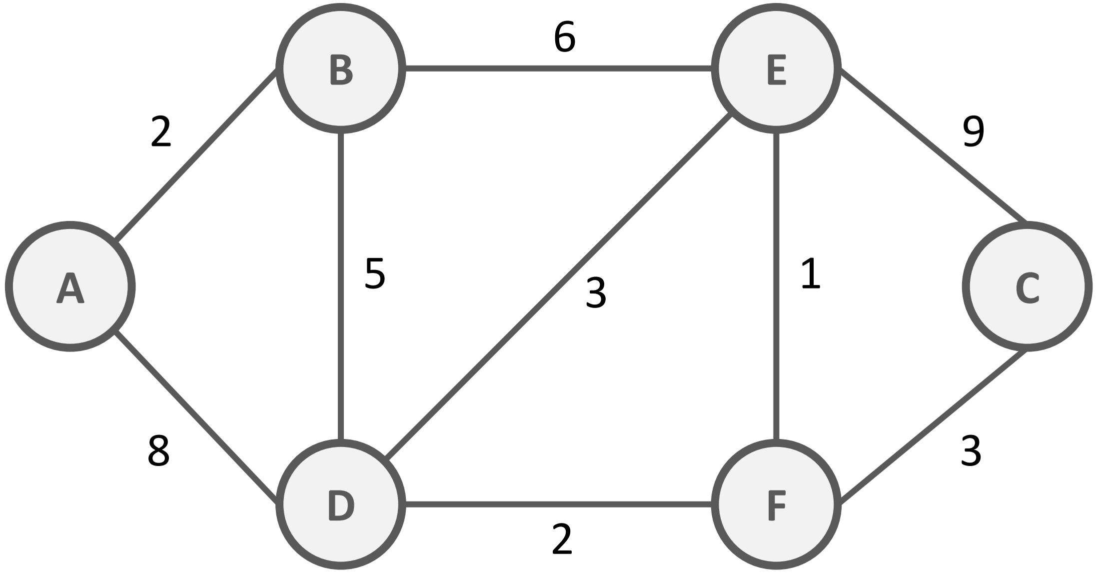
Module 11 – Graphs

Dijkstra's Algorithm

- Is used to find the shortest path in a G(V, E) graph from vertex u to v, alongside constructing a shortest path tree as well
- It can handle positive edge weights
- During every iteration, it searches for the minimum distance to the next vertex
- The appropriate data structure is a Heap (Priority Queue)

Module 11 – Graphs

Dijkstra's Algorithm



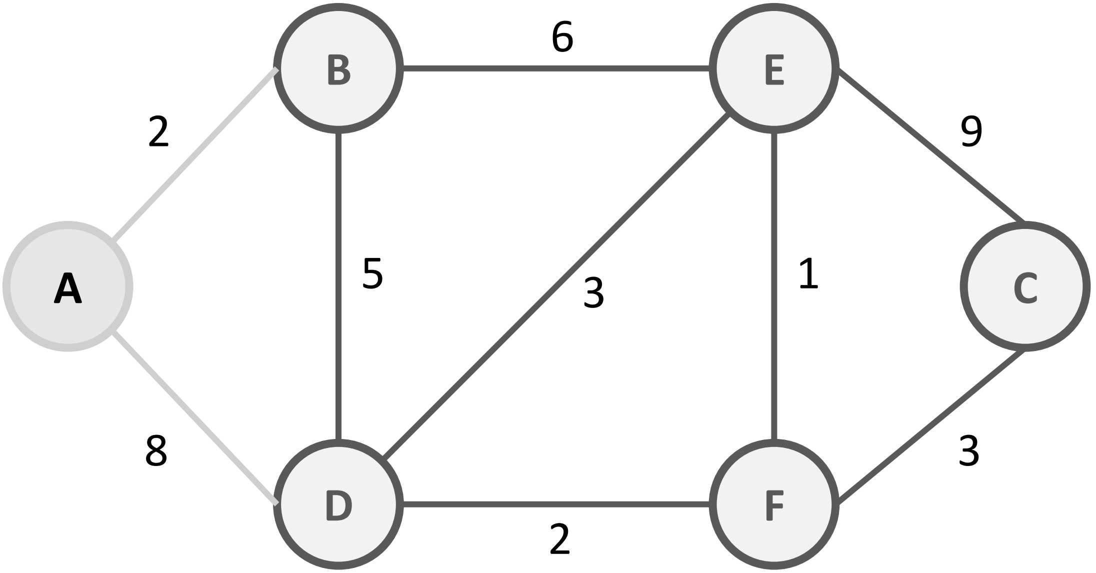
PQ = [A:0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞]

V = []

Vertex	Shortest Distance	Predecessor Vertex
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Module 11 – Graphs

Dijkstra's Algorithm



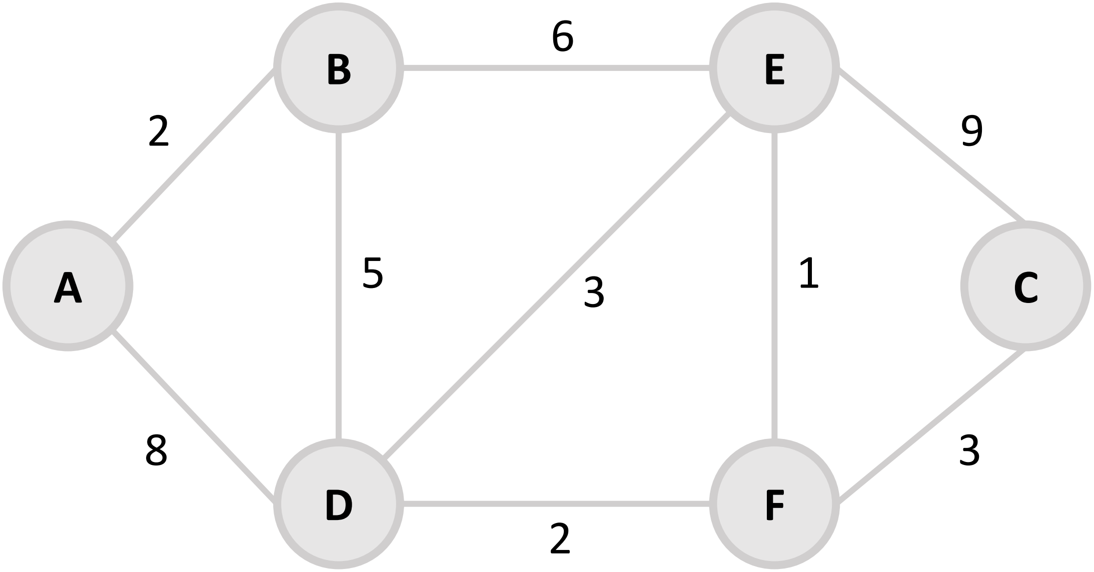
PQ = [B: 2, C: ∞, D: 8, E: ∞, F: ∞]

V = [A]

Vertex	Shortest Distance	Predecessor Vertex
A	0	
B	2	A
C	∞	
D	8	A
E	∞	
F	∞	

Module 11 – Graphs

Dijkstra's Algorithm

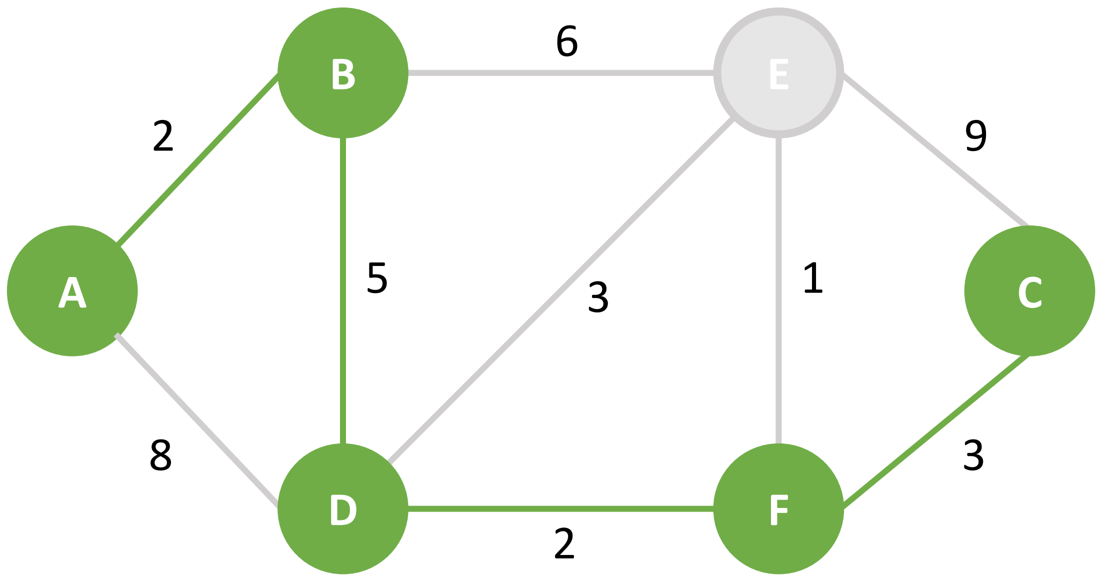


PQ = []
V = [A, B, D, E, F, C]

Vertex	Shortest Distance	Predecessor Vertex
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

Module 11 – Graphs

Dijkstra's Algorithm



PQ = []
V = [A, B, D, E, F, C]

Vertex	Shortest Distance	Predecessor Vertex
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

Shortest path from A to C:
A-B-D-F-C = 12

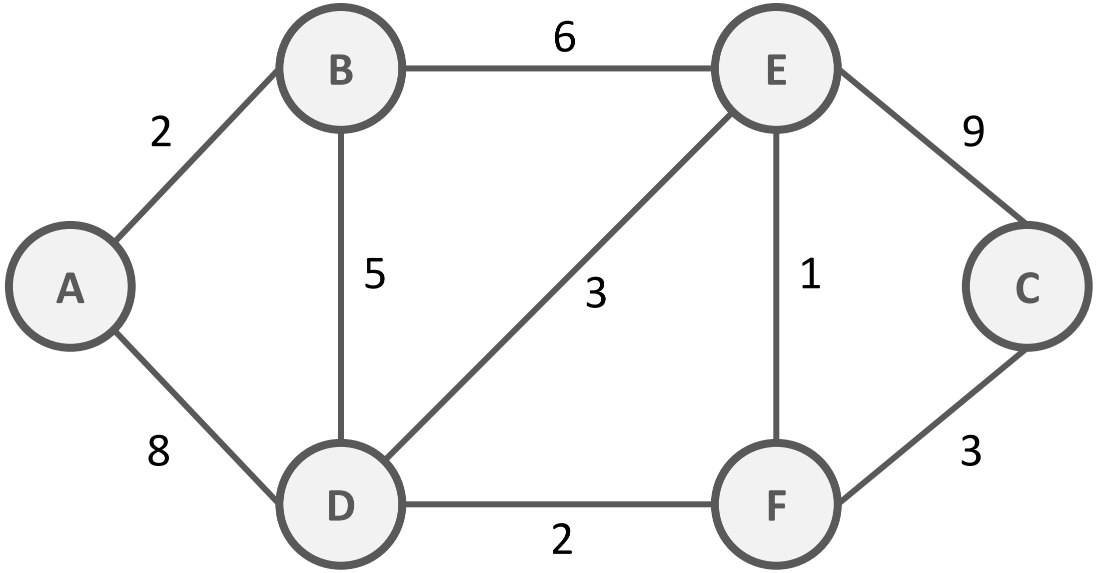
Module 11 – Graphs

Minimum Spanning Tree

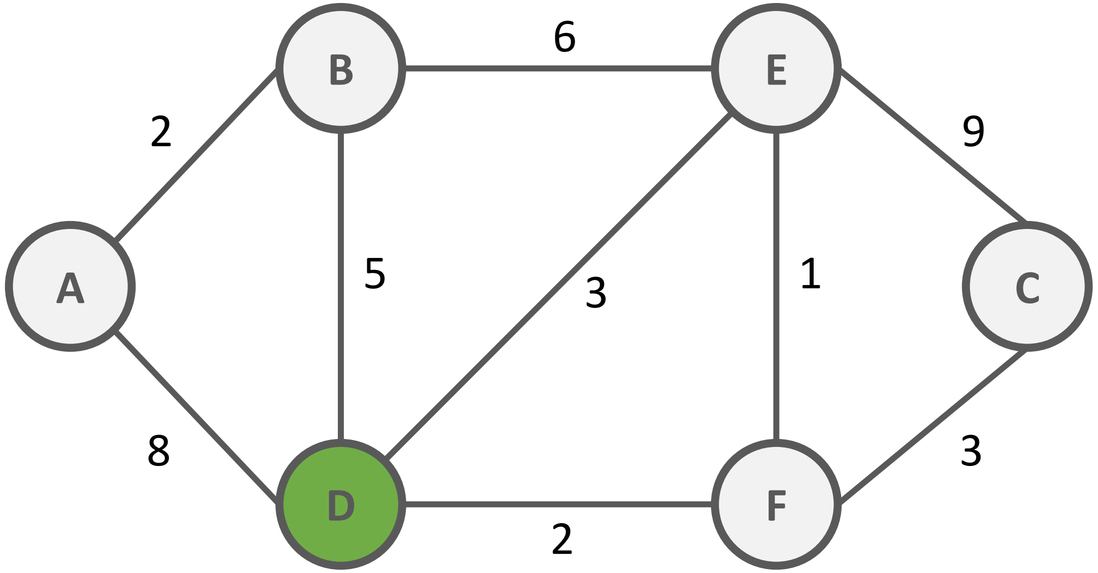
- Given an undirected graph G with weighted edges, a minimum spanning tree (MST) is a subset of the edges in the graph which:
  - connects all vertices together
  - have no cycles
  - Include edges with minimum weight only
  - Maintain a Priority Queue (PQ) of edges

Module 11 – Graphs

Minimum Spanning Tree



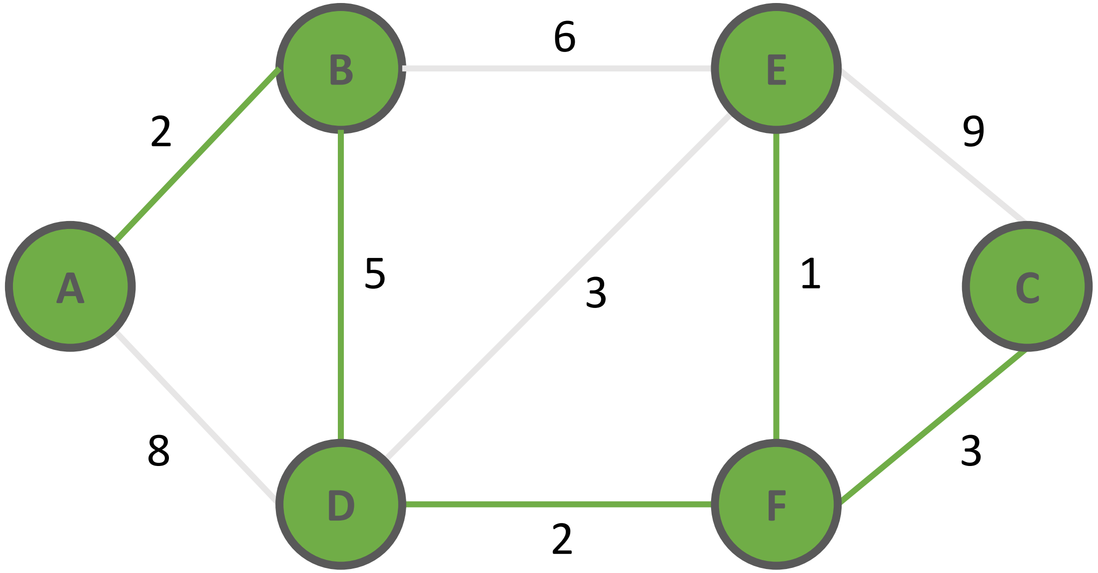
Module 11 – Graphs



Visited
D

PQ

Module 11 – Graphs



MST weight = 2 + 5 + 2 + 1 + 3 = 13

Visited
D
F
E
C
B
A

PQ
D-A, 8
E-B, 6
E-C, 9

MST Edges = 5