Advanced Programming

INFO135

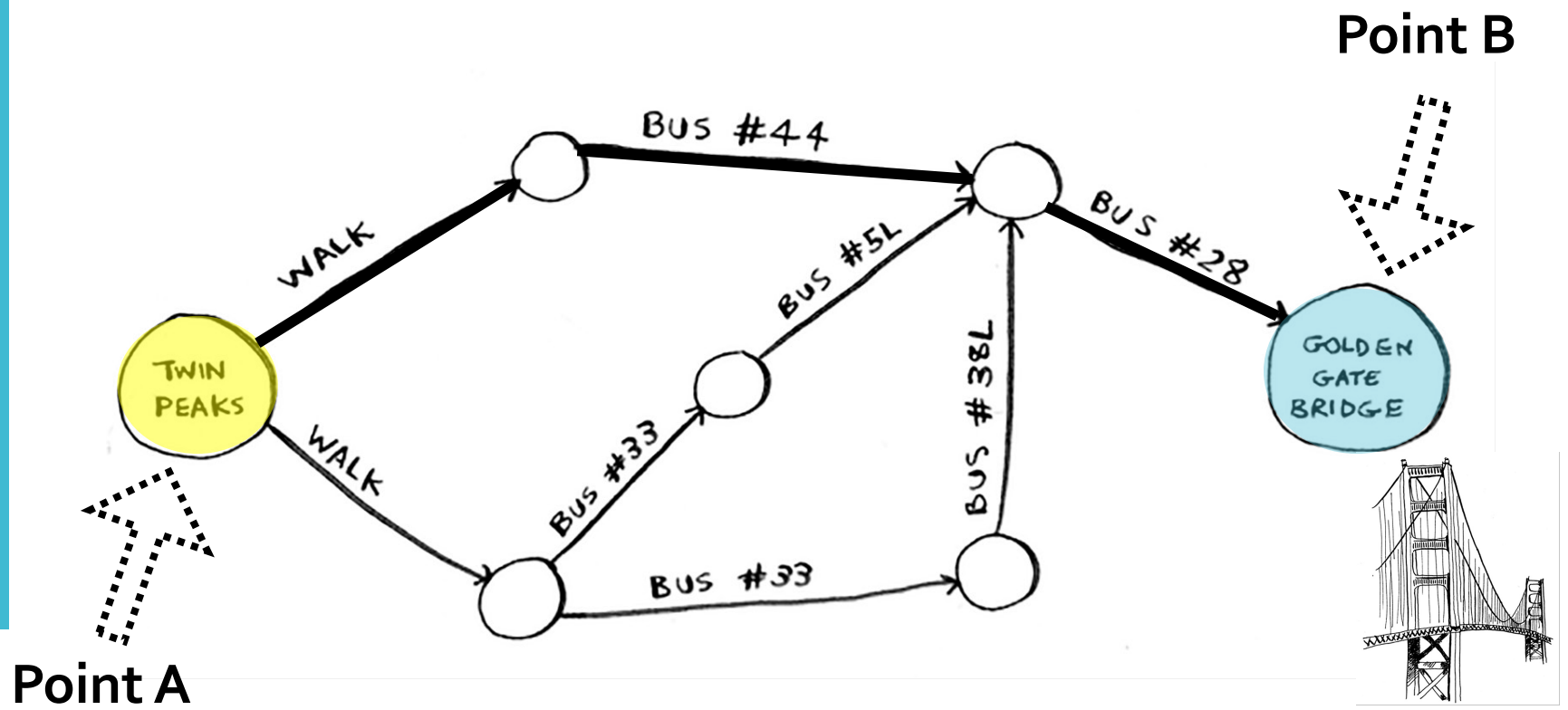
Lecture 8: Dijkstra algorithm

Mehdi Elahi

University of Bergen (UiB)

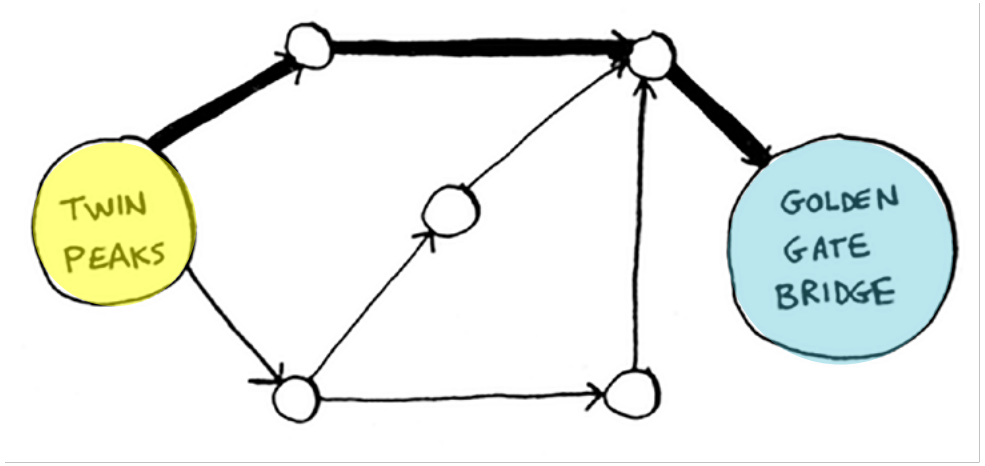
Graph

- Recall the example of Lecture 7, where we found a the **shortest path** to get from point A to point B.



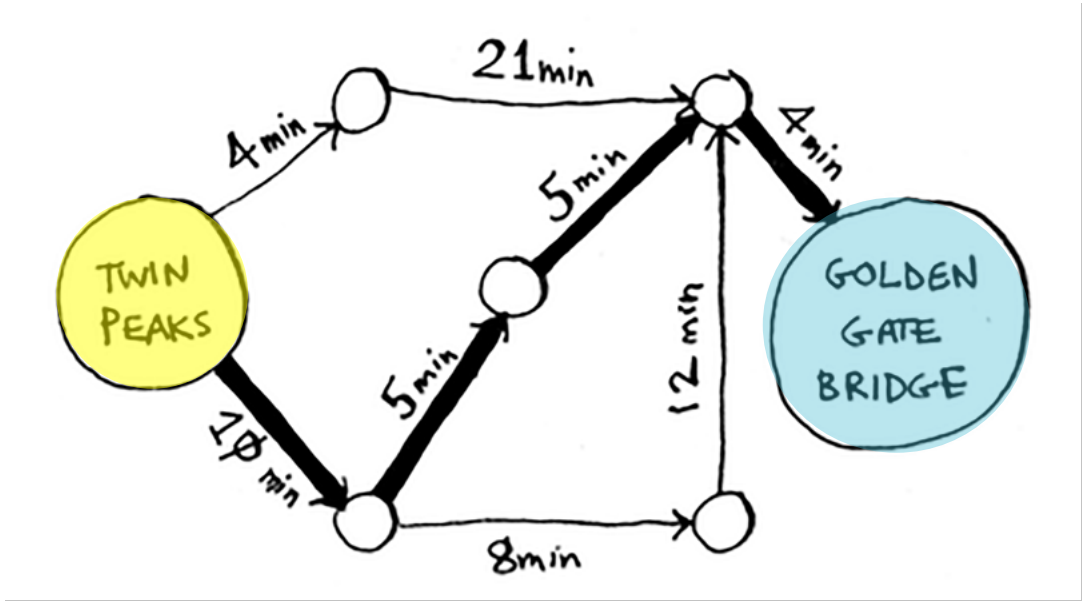
Graph

- It's the shortest path, as it passes the **least number of nodes**.
- But **is it also the fastest** path?



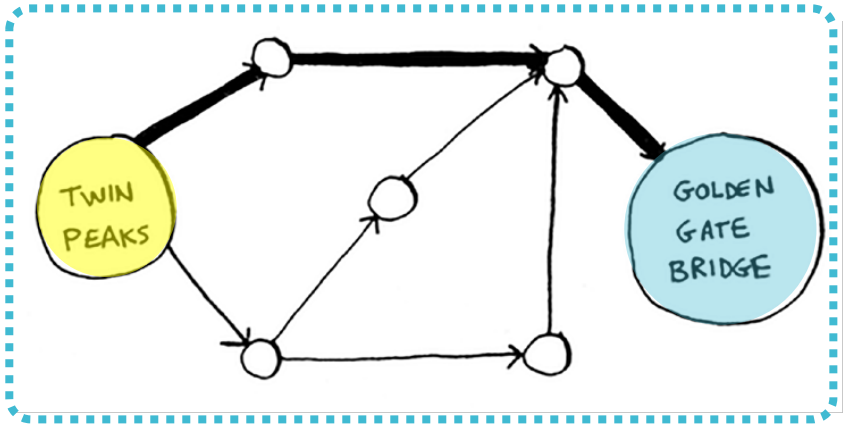
Graph

- If we measure the **travel times**, then, we can find a new path.
- The new path, includes more nodes but it is the **fastest**.

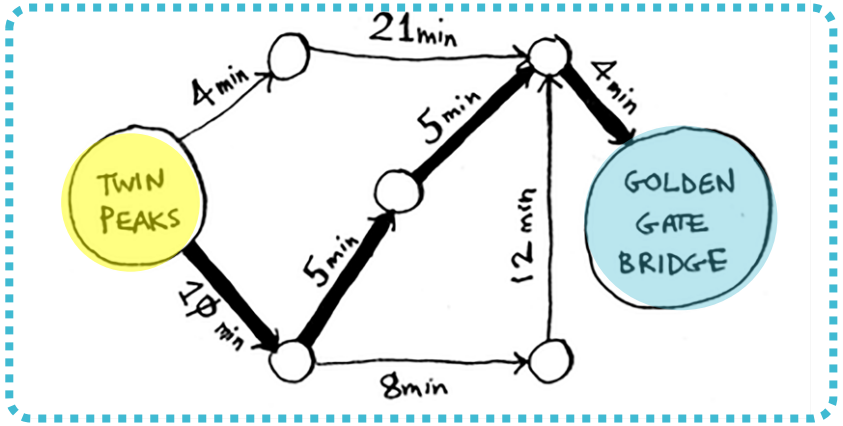


Graph

- Why this has happened?
- Because, BFS and DFS **do not take edge weights into account**.
- Here, **weights** are costs in terms time.



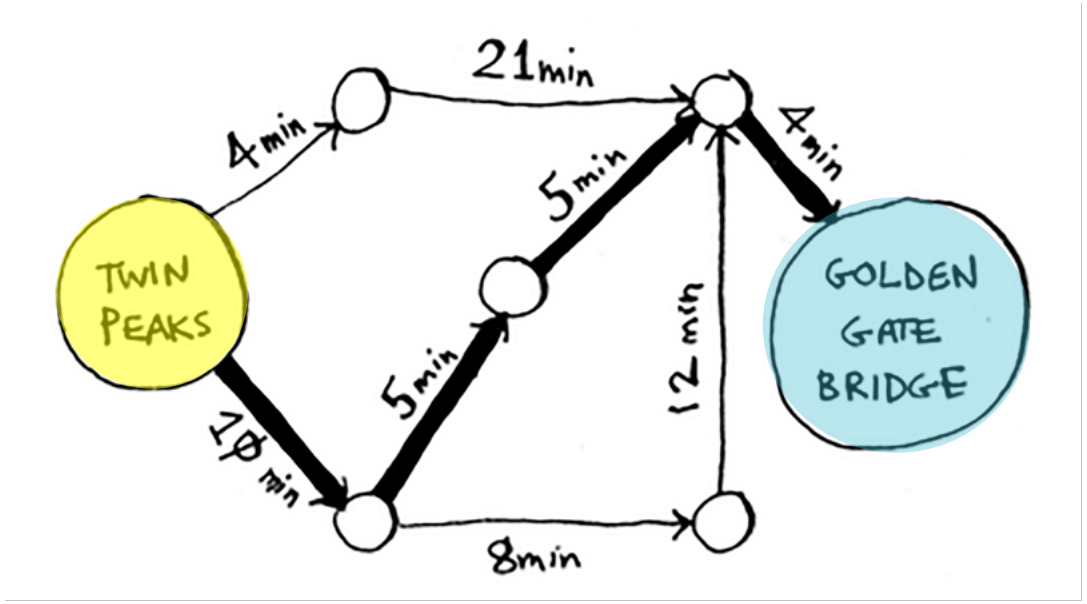
Shortest path
(less number of nodes)



Fastest path
(less time)

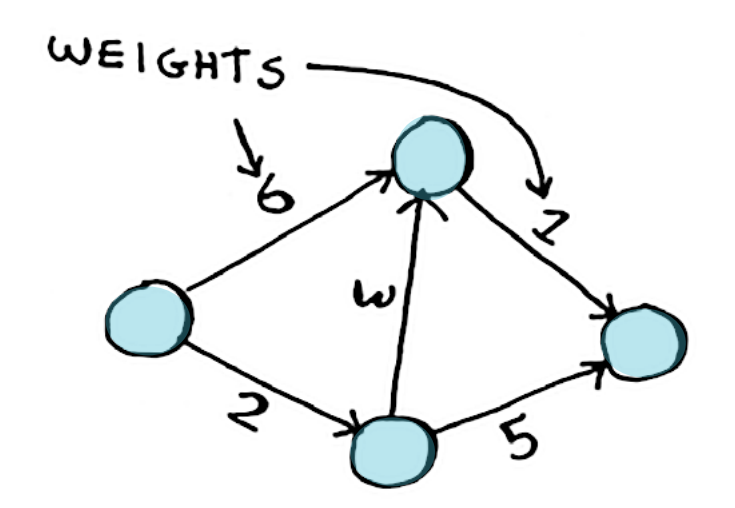
Graph

- **Dijkstra's algorithm** can find the quickest path.



Graph

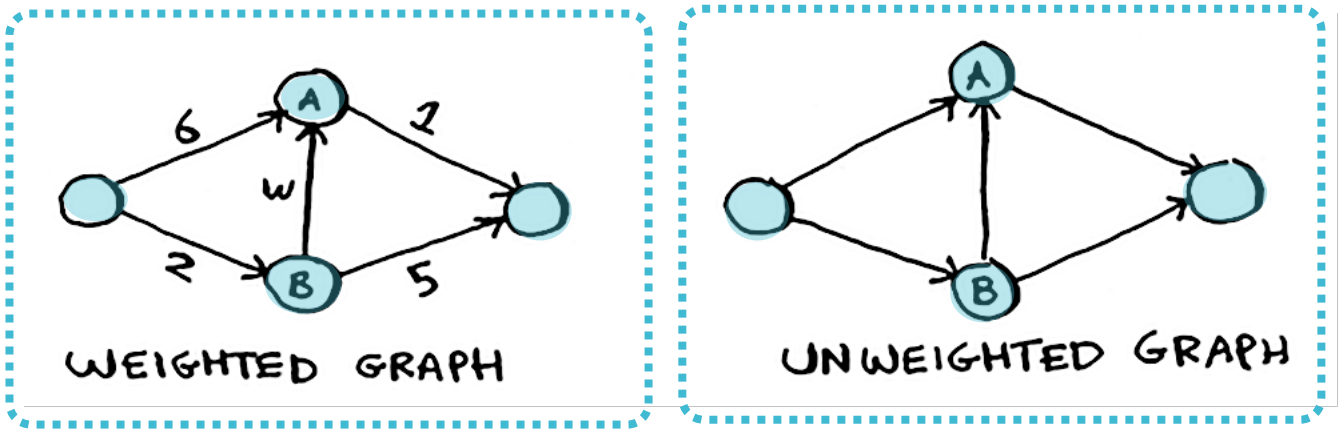
- **Weights:** numerical values associated with edges of a Graph. Weights are referred to as **cost of the edge**.



- **For examples, weights can be:**
  - the length of a route,
  - the capacity of a line,
  - the energy required to move between locations

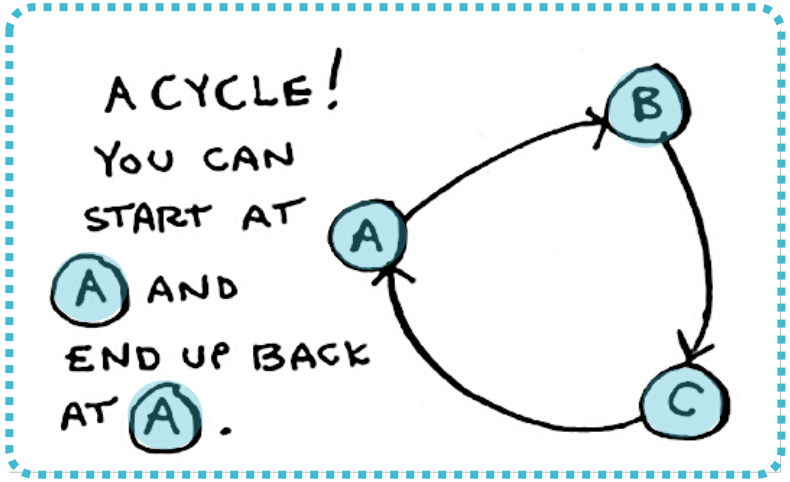
Graph

- A graph with weights is called a **Weighted Graph**.
- A graph without weights is called an **Unweighted Graph**.



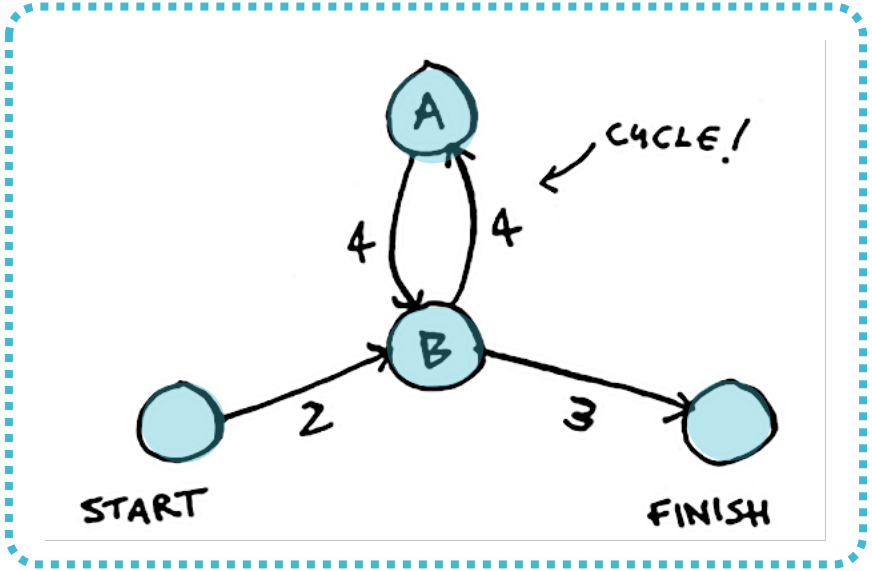
Graph

- **Cycle:** exists if we can start from a vertex (node), and traverse the Graph, & finish at the same vertex.
- **Cycle** is basically a closed path.



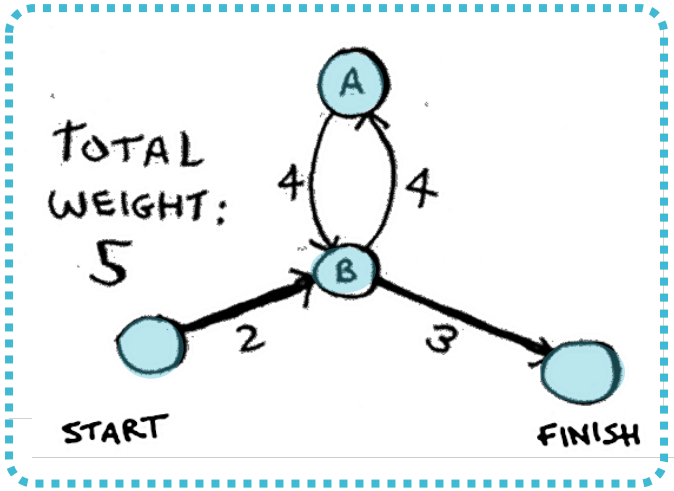
Dijkstra algorithm

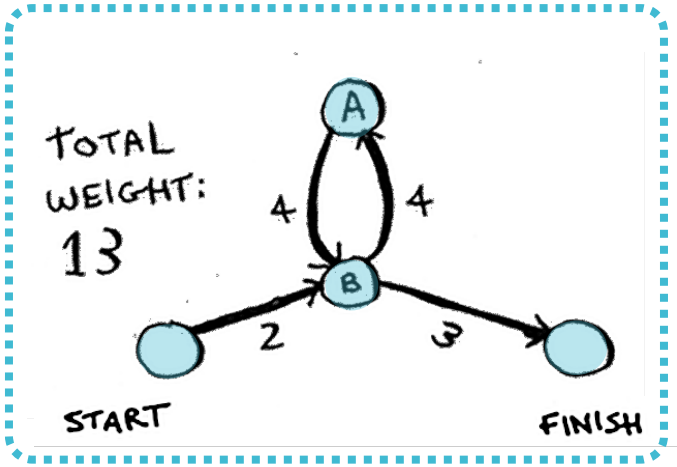
- **Dijkstra's** algorithm works with **Directed Acyclic Graphs (DAG)** with weights that are positive.
- But why?
- Look at the following graph. Would it make sense to **follow the cycle**?



Dijkstra algorithm

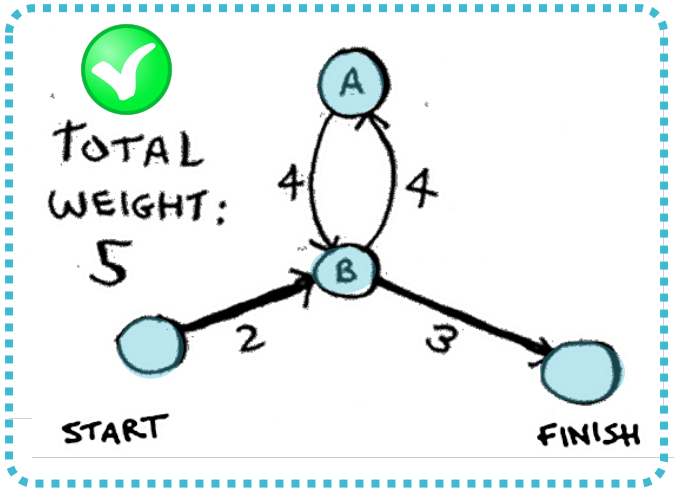
- Look at these paths. Which one would **you choose**?
  a) left path that **avoids** the **cycle**.
  b) right path that **follows** the **cycle**.

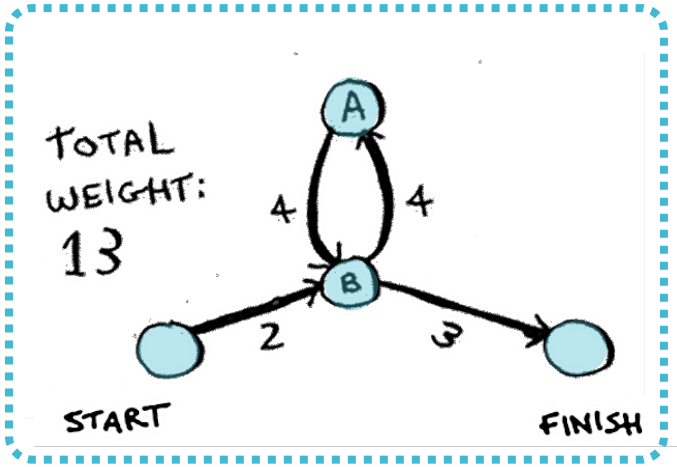




Dijkstra algorithm

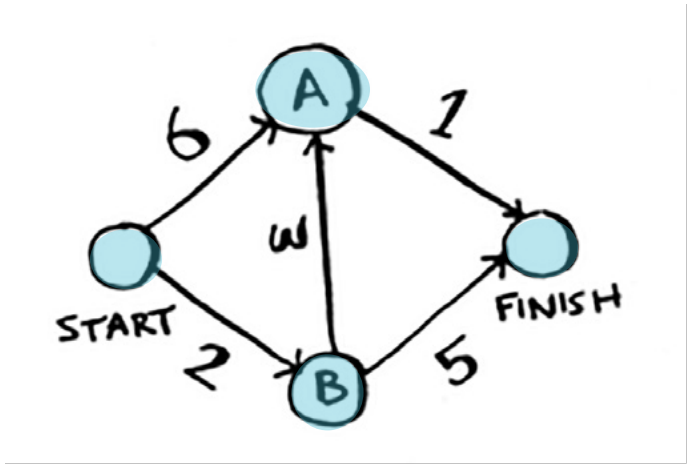
- The right path is what we choose.
- Because, if we follow the **cycle**, we just add 8 to the total weight.
- **Following cycle** will **not** find the **shortest path.**





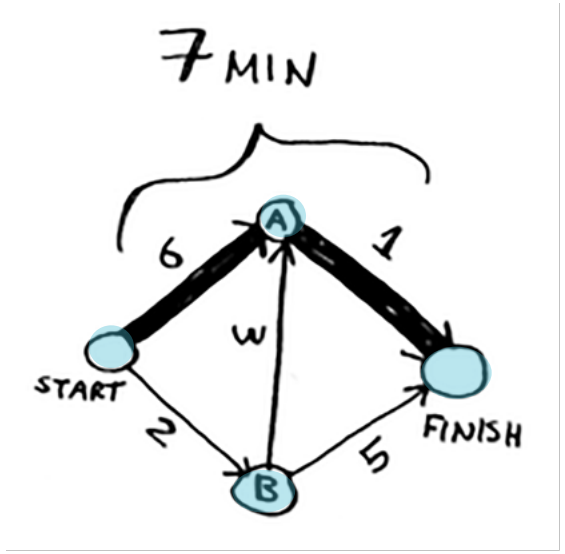
Dijkstra algorithm

- Lets see how **Dijkstra's algorithm** works.
- Check the following graph where each edge has a **travel time** in minutes.



Dijkstra algorithm

- If we find the shortest path between start and finish, **BFS** will find the following path that takes 7 minutes.



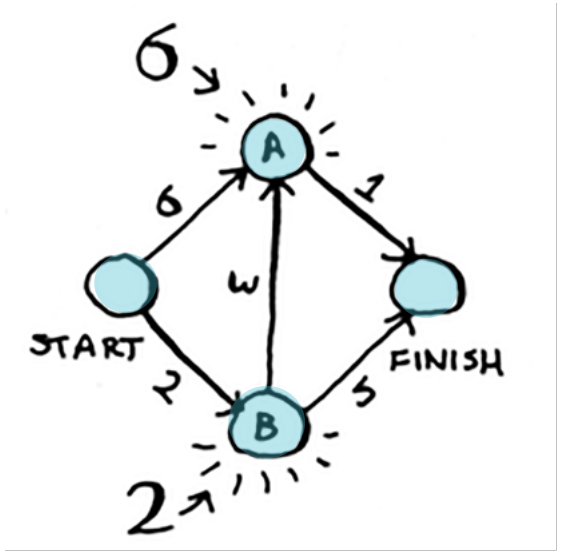
Dijkstra algorithm

- **Dijkstra's algorithm** takes the following steps:
  - **Step 1:** Find the node which we can get to in the least amount of time.
  - **Step 2:** Update the costs of neighbors of this node.
  - **Step 3:** Repeat until you've done this for every node in the graph.
  - **Step 4:** Calculate the final path.

Dijkstra algorithm

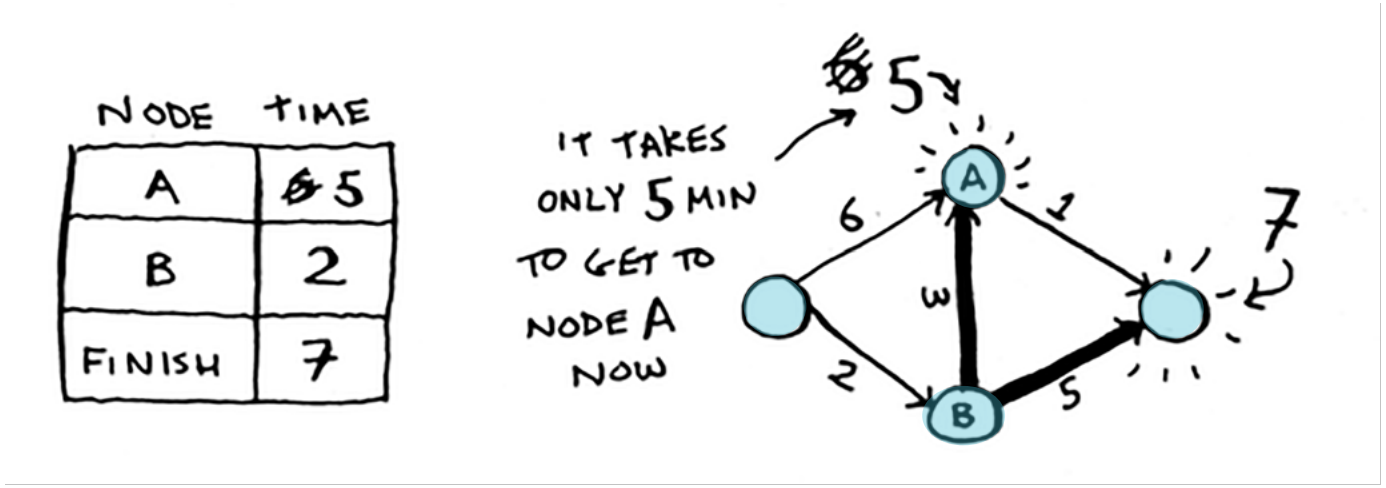
- Lets apply this algorithm:
  - **Step 1:** Find the node with least amount of time to reach (that is B). It takes 6 minutes to A and 2 minutes B. We don't know the rest and we assume infinity for them.

NODE	TIME TO NODE
A	6
B	2
FINISH	∞



Dijkstra algorithm

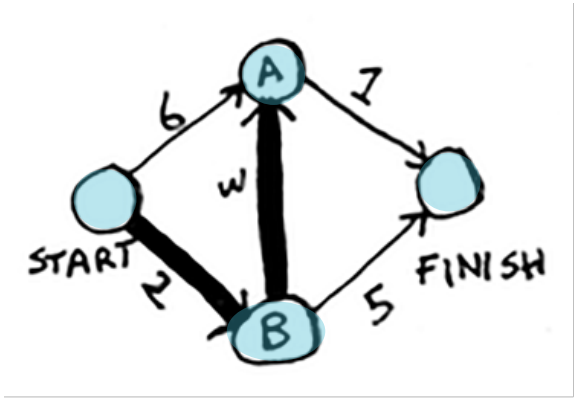
- **Step 2:** Calculate the time it takes to get to neighbors of node B, by following an edge from B.



Dijkstra algorithm

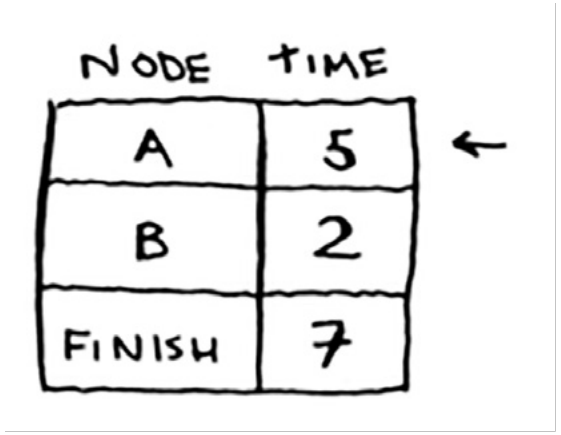
- When quicker paths found, **update costs**:
  - quicker **path to A** (from 6 minutes to 5 minutes)
  - quicker **path to finish** (from infinity to 7 minutes)

NODE	TIME
A	~~6~~ 5
B	2
FINISH	7



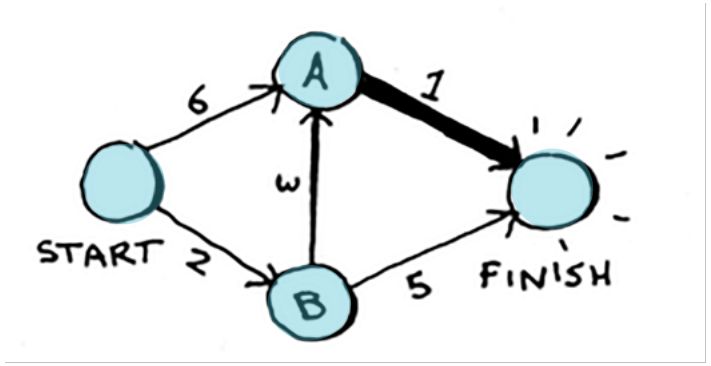
Dijkstra algorithm

- **Step 3:** Repeat!
  - **Step 1 again:** Find the node that takes the least amount of time to get to. You're done with node B, so node A has the next smallest time estimate.



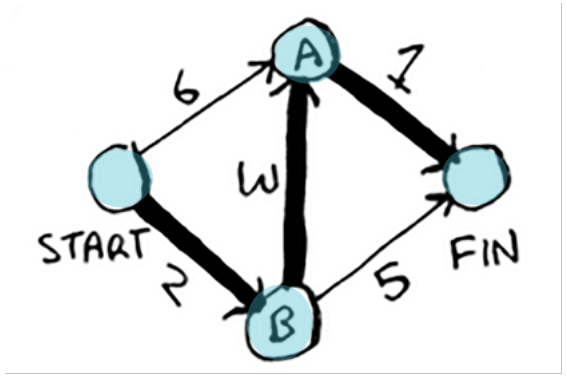
Dijkstra algorithm

- **Step 3:** Repeat!
  - **Step 2 again:** Update the costs for node A's neighbors.



Dijkstra algorithm

- **Step 3:** Repeat!
  - It takes 6 minutes to get to the finish now!



  - So far we know
    - it takes 2 minutes to get to node B.
    - it takes 5 minutes to get to node A.
    - it takes 6 minutes to get to the finish.

NODE	TIME
A	5
B	2
FINISH	6

Dijkstra algorithm

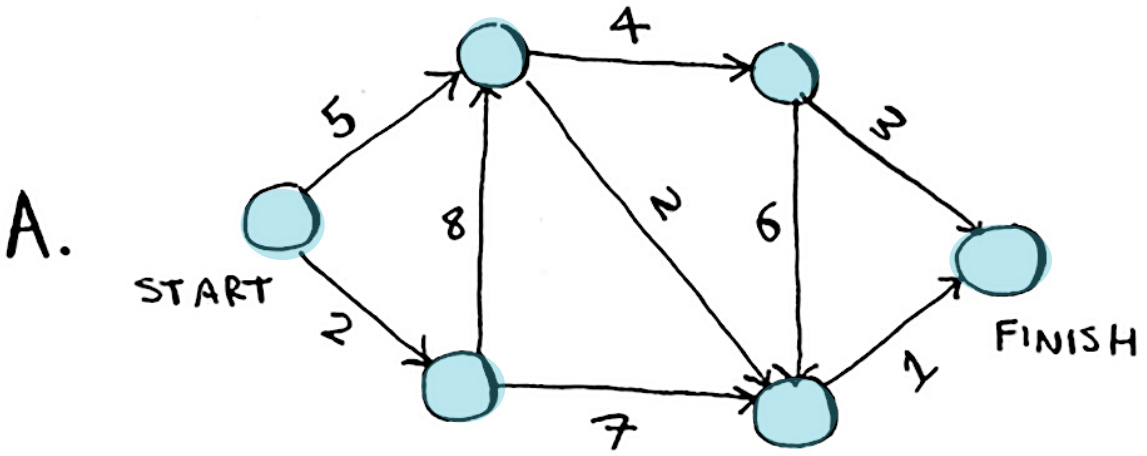
Dijkstra's Shortest Path Algorithm

ref: youtu.be/pVfj6mxhdMw

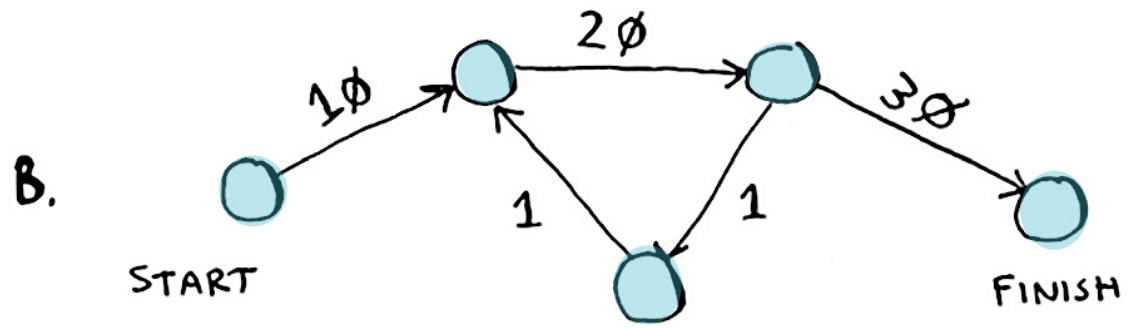
Quiz

- In each of these graphs, what is the **total weight** of the shortest path from the start node to finish node?

A.



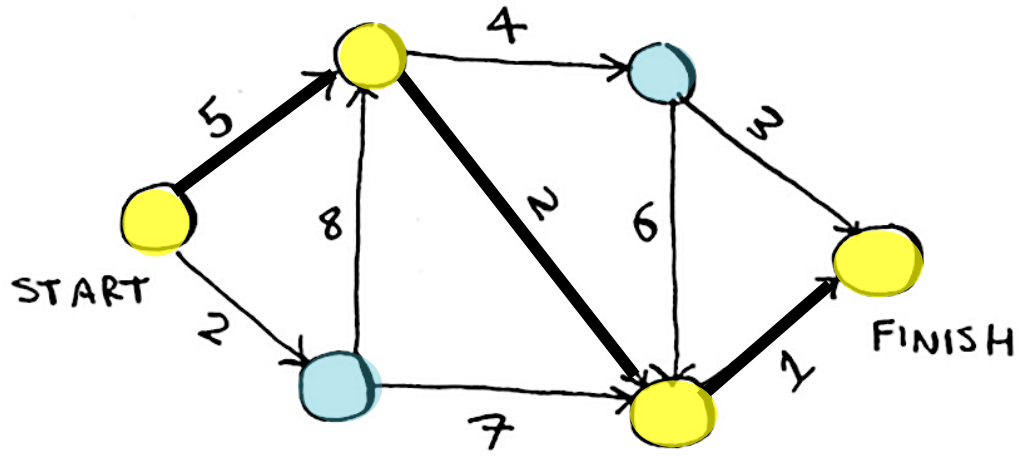
B.



Answer

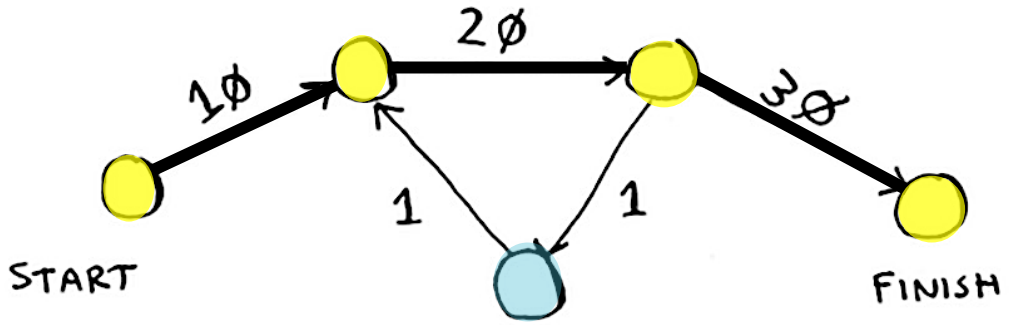
- In each of these graphs, what is the **total weight** of the shortest path from the start node to finish node?

A.



5+2+1=**8**

B.



10+20+30=**6o**

Dijkstra algorithm

Implementation of **Dijkstra algorithm (part 1)**

```
class Graph2:
    graph = dict()

    def dijkstra(self):
        visited = []
        node = self.find_lowest_cost(visited)

        while node is not None:
            print("[", node, end=" ] ")
            cost = self.costs[node]
            neighbors = self.graph[node]

            for n in neighbors.keys():
                new_cost = cost + neighbors[n]
                if self.costs[n] > new_cost:
                    self.costs[n] = new_cost
            visited.append(node)
            node = self.find_lowest_cost(visited)
```

Dijkstra algorithm

Implementation of **Dijkstra algorithm (part 1)**

```
class Graph2:
    graph = dict()

    def dijkstra(self):
        visited = []
        node = self.find_lowest_cost(visited)

        while node is not None:
            print("[", node, end=" ] ")
            cost = self.costs[node]
            neighbors = self.graph[node]

            for n in neighbors.keys():
                new_cost = cost + neighbors[n]
                if self.costs[n] > new_cost:
                    self.costs[n] = new_cost
            visited.append(node)
            node = self.find_lowest_cost(visited)
```

Dijkstra algorithm

❖ Implementation of **Dijkstra algorithm (part 1)**

```
class Graph2:
    graph = dict()

    def dijkstra(self):
        visited = []
        node = self.find_lowest_cost(visited)

        while node is not None:
            print("[", node, end=" ] ")
            cost = self.costs[node]
            neighbors = self.graph[node]

            for n in neighbors.keys():
                new_cost = cost + neighbors[n]
                if self.costs[n] > new_cost:
                    self.costs[n] = new_cost
            visited.append(node)
            node = self.find_lowest_cost(visited)
```

Dijkstra algorithm

Implementation of **Dijkstra algorithm (part 2)**

```
    def find_lowest_cost(self, visited):
        lowest_cost = float('inf')
        lowest_cost_node = None

        for node in self.costs:
            cost = self.costs[node]
            if cost < lowest_cost \
                    and node not in visited:
                lowest_cost = cost
                lowest_cost_node = node
        return lowest_cost_node

    def set_costs(self, costs):
        self.costs = costs
```

Dijkstra algorithm

Implementation of **Dijkstra algorithm (part 2)**

```
    def find_lowest_cost(self, visited):
        lowest_cost = float('inf')
        lowest_cost_node = None

        for node in self.costs:
            cost = self.costs[node]
            if cost < lowest_cost \
                    and node not in visited:
                lowest_cost = cost
                lowest_cost_node = node
        return lowest_cost_node

    def set_costs(self, costs):
        self.costs = costs
```

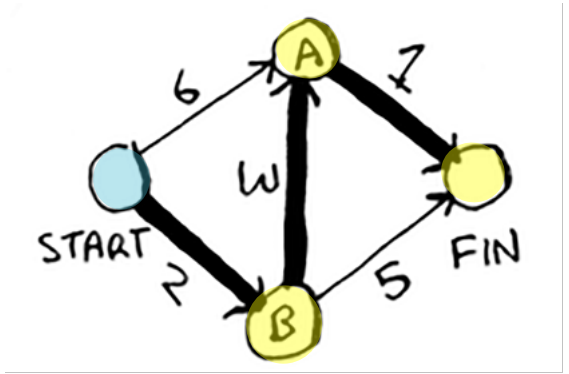
Dijkstra algorithm

Implementation of **Dijkstra algorithm (part 3)**

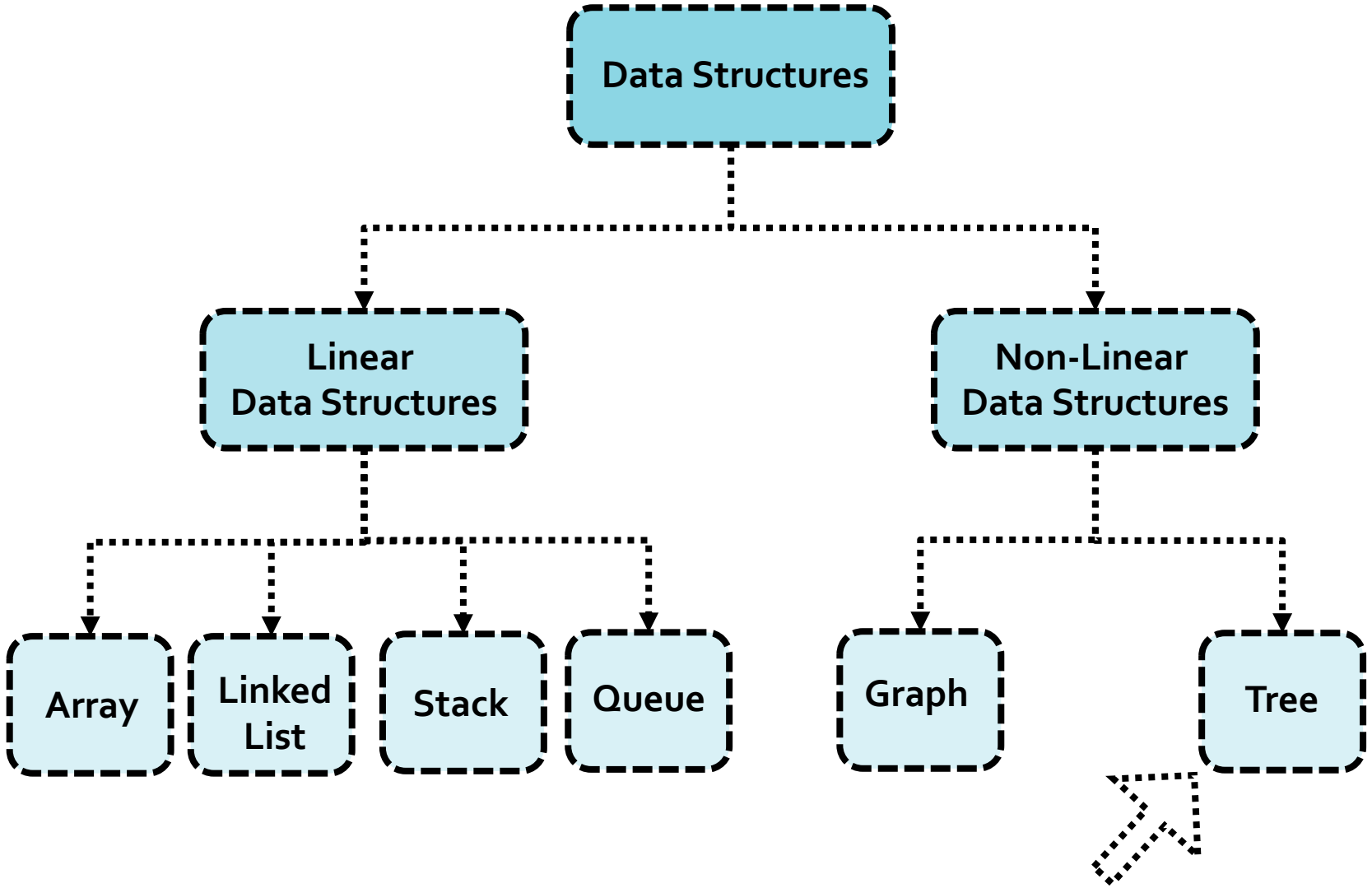
```
g2 = Graph2()

my_graph = g2.graph
my_graph['start'] = {'A': 6, 'B': 2}
my_graph['A'] = {'finish': 1}
my_graph['B'] = {'A': 3, 'finish': 5}
my_graph['finish'] = {}

costs = {'A': 6, 'B': 2, 'finish': float('inf')}
g2.set_costs(costs)

g2.dijkstra()
```

[Output:]

```
[ B ] [ A ] [ finish ]
```



Data Structures



Tree

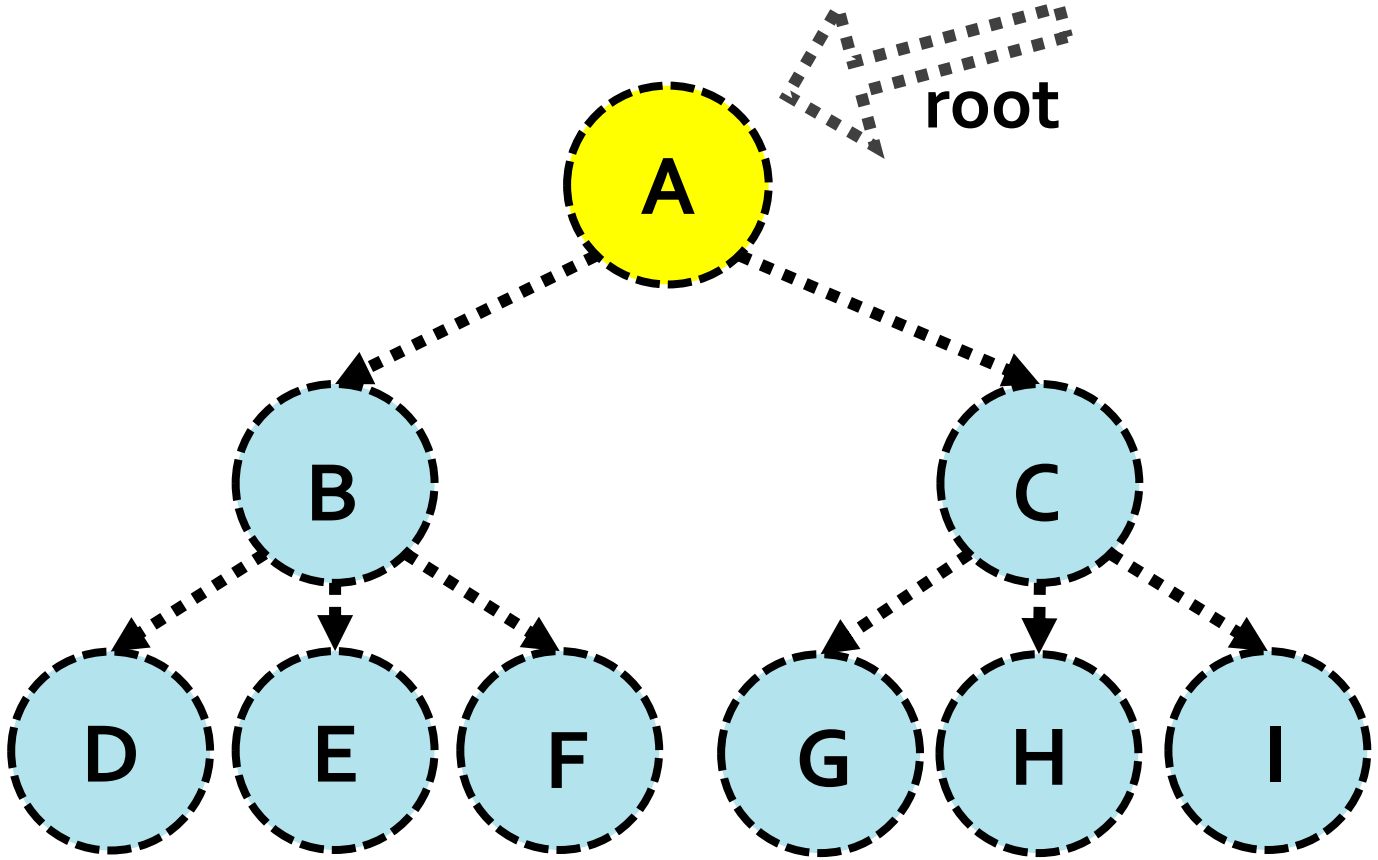
- **Tree:** a hierarchical data structure with a special node (**called root**) and a set of other nodes that are connected by some edges.

- Tree is indeed an **undirected Graph** in which any two vertices (nodes) are connected by exactly one path.

- Nodes of the Tree have a **parent-child relationship**.
- A **parent node** can have multiple child nodes, and there can only be one parent node for each child node.

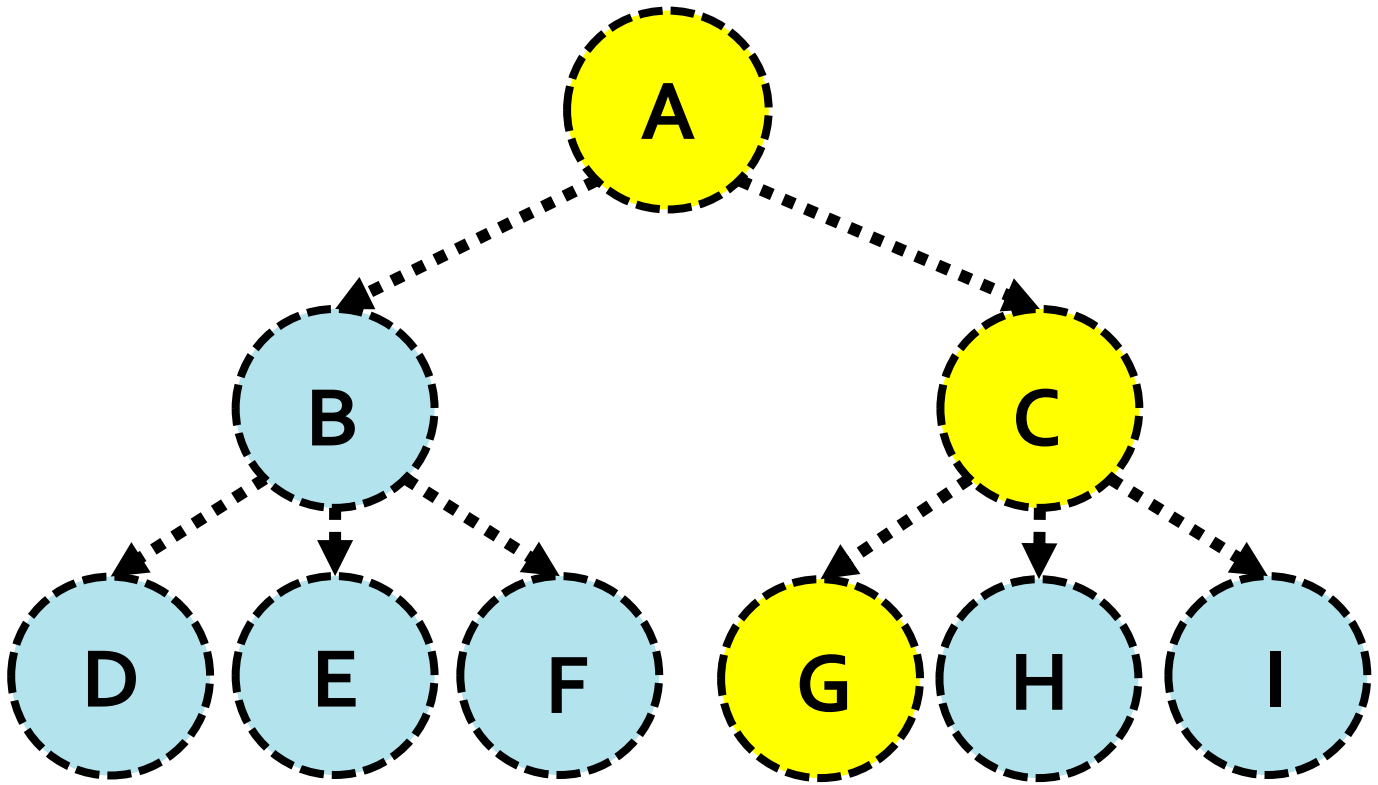
Tree

- **Root:** topmost node of the tree is the root node, the only node in tree that has no incoming edges.
- For example A is the root of this tree.



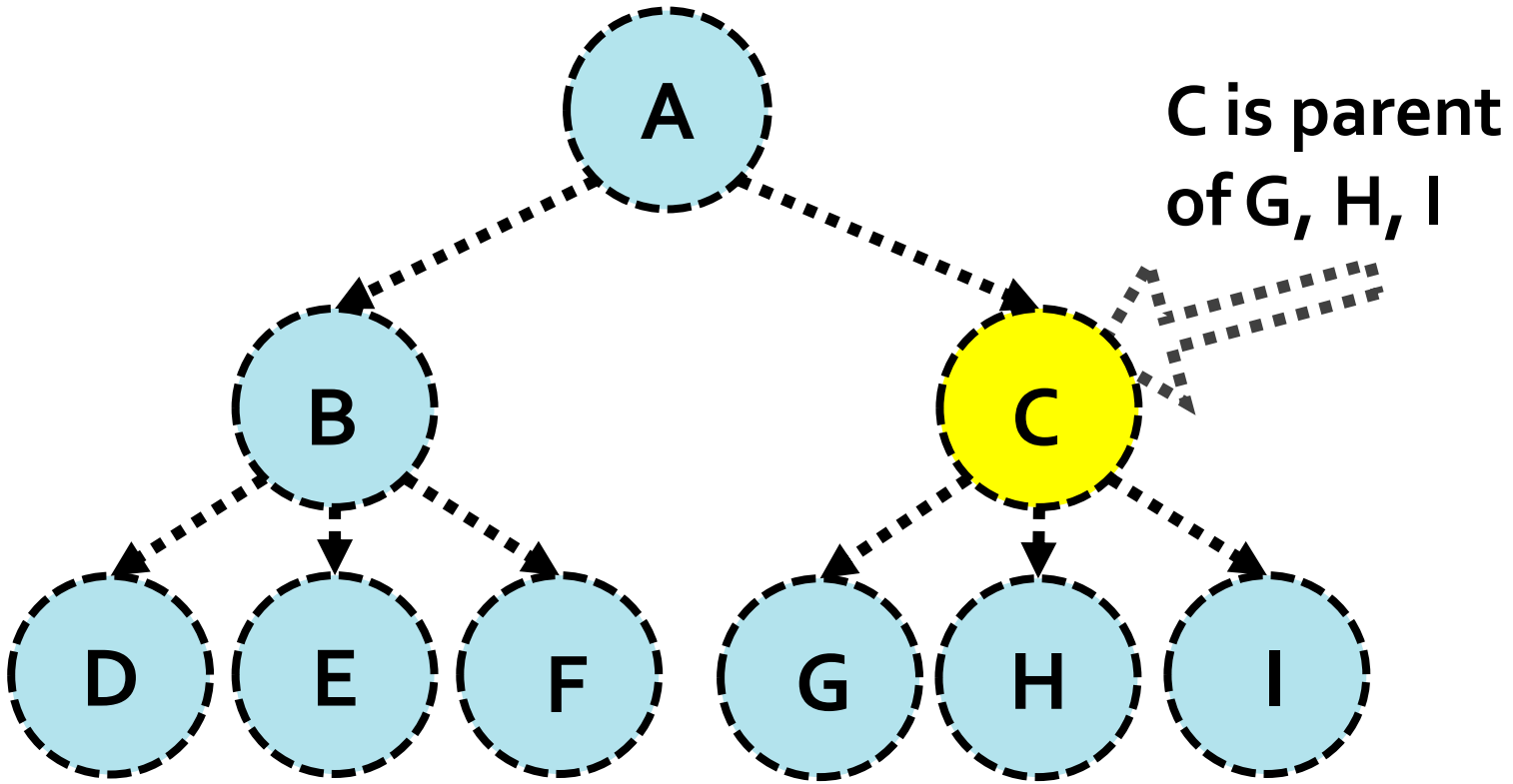
Tree

- **Path:** an ordered list of nodes that are connected by edges.
- For example, **A → C → G** is a path.



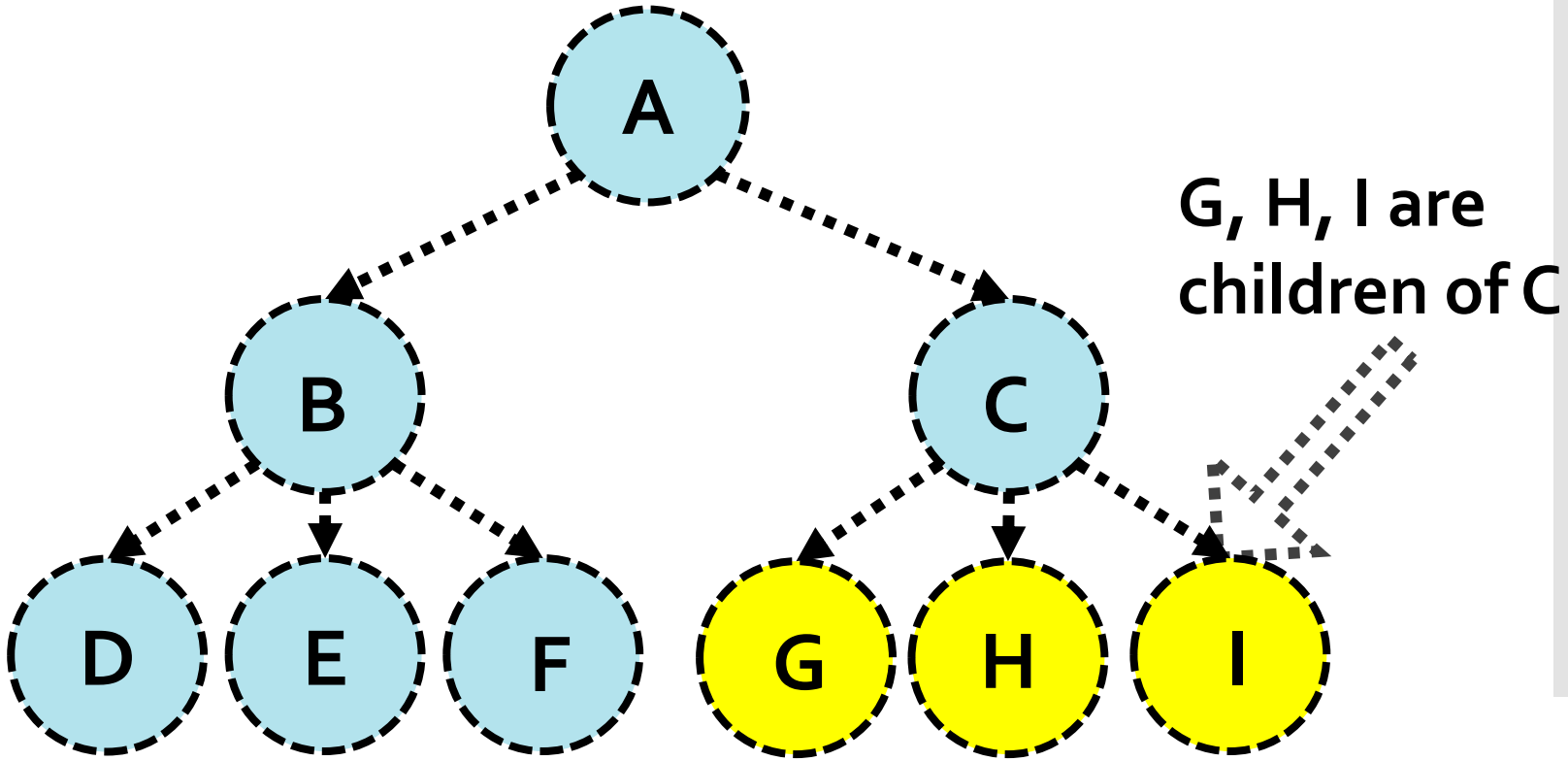
Tree

- **Parent:** each node in Tree, except the root, has a parent node, identified by the incoming edge.
- For example C is parent of G, H and I



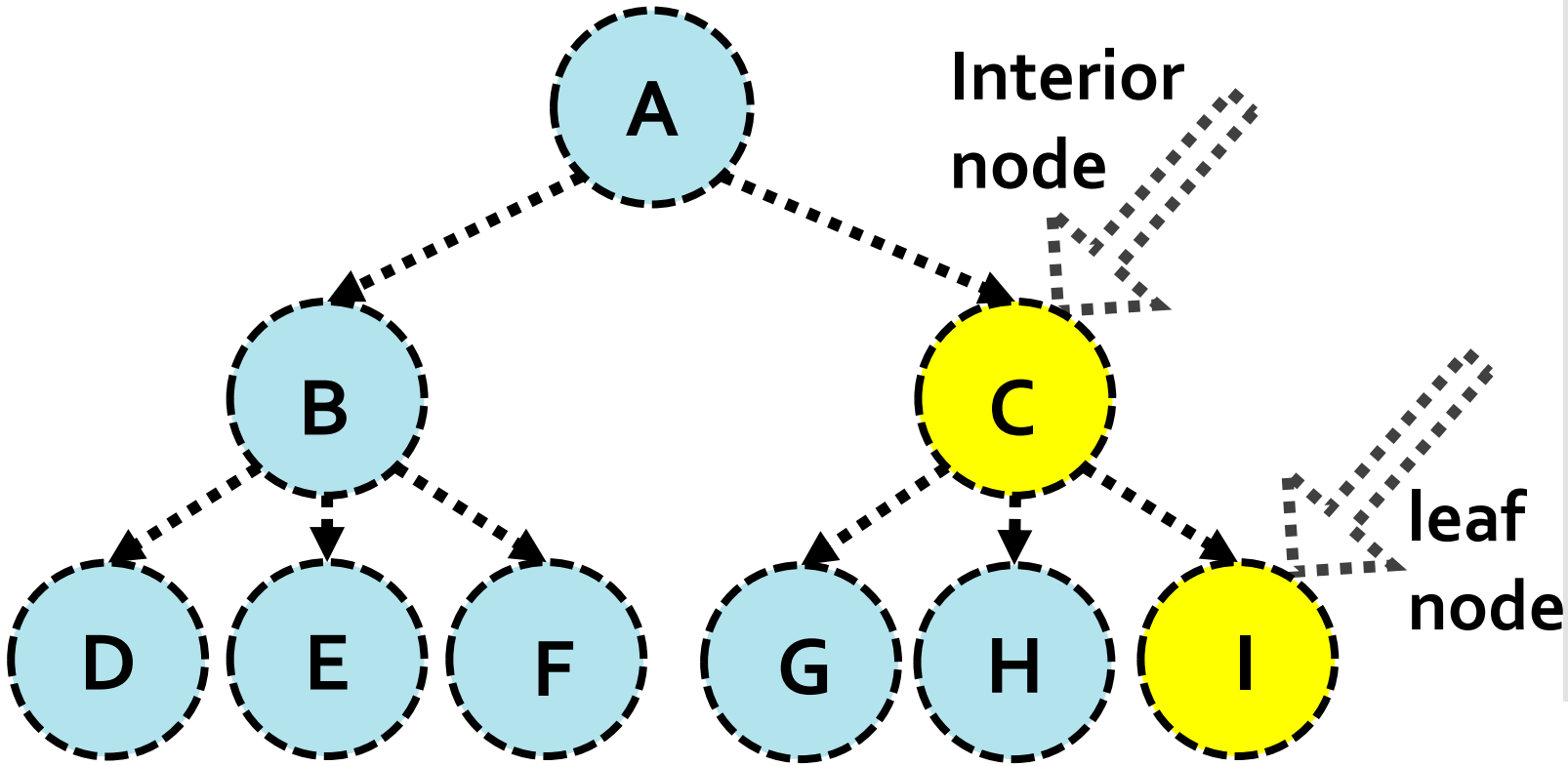
Tree

- **Children:** a set of nodes with incoming edges from the same node (Parent) are the children of that node.
- Example: G, H, and I are children of C



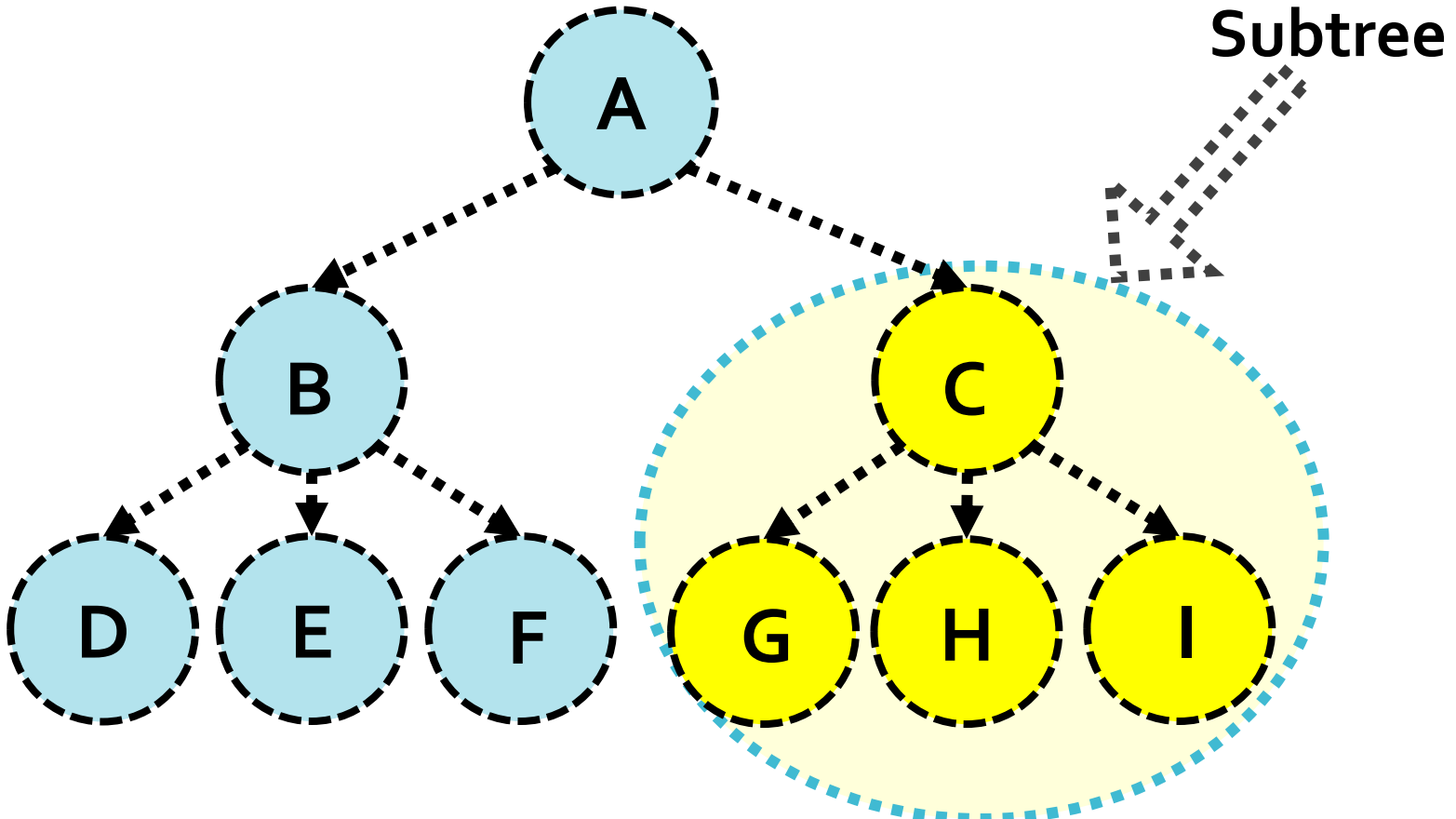
Tree

- **Leaf Node:** Nodes that have no children.
- **Interior Node:** Nodes that have at least one child are known as interior nodes.
- **Example:** C is an interior node and I is a leaf node



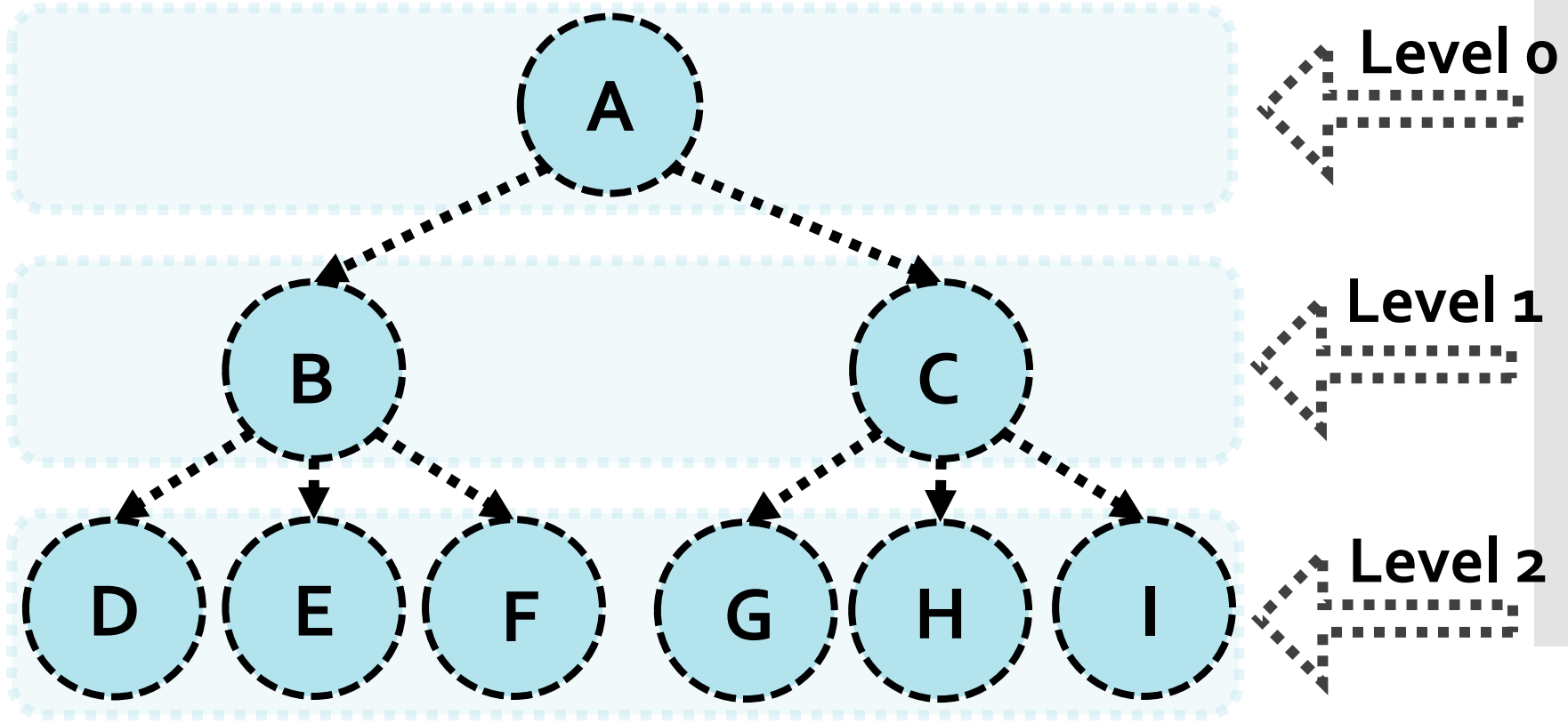
Tree

- **Subtree:** since Tree is a recursive structure, hence each node of the Tree can be the root of its own subtree.



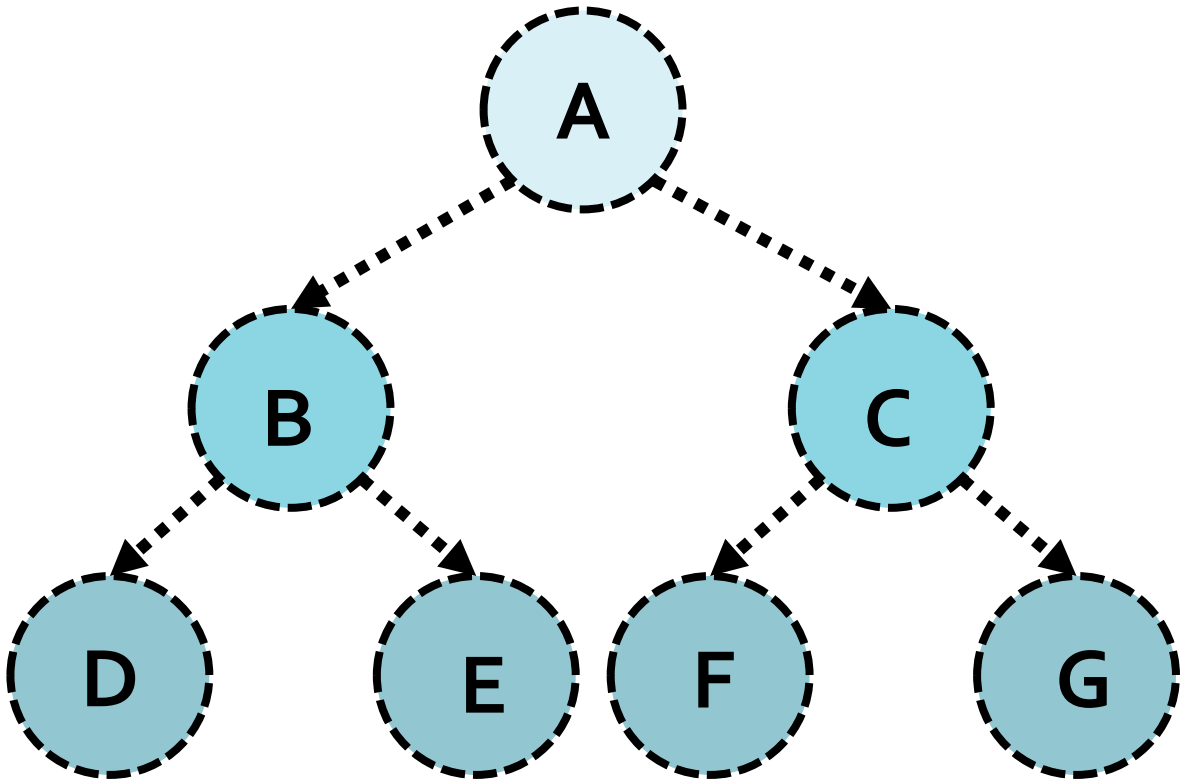
Tree

- **Level:** the number of edges on the path from the root node to leaf node.
- **For example**, the level of the root node is 0, the level of node C is 1, and the level of node I is 2.



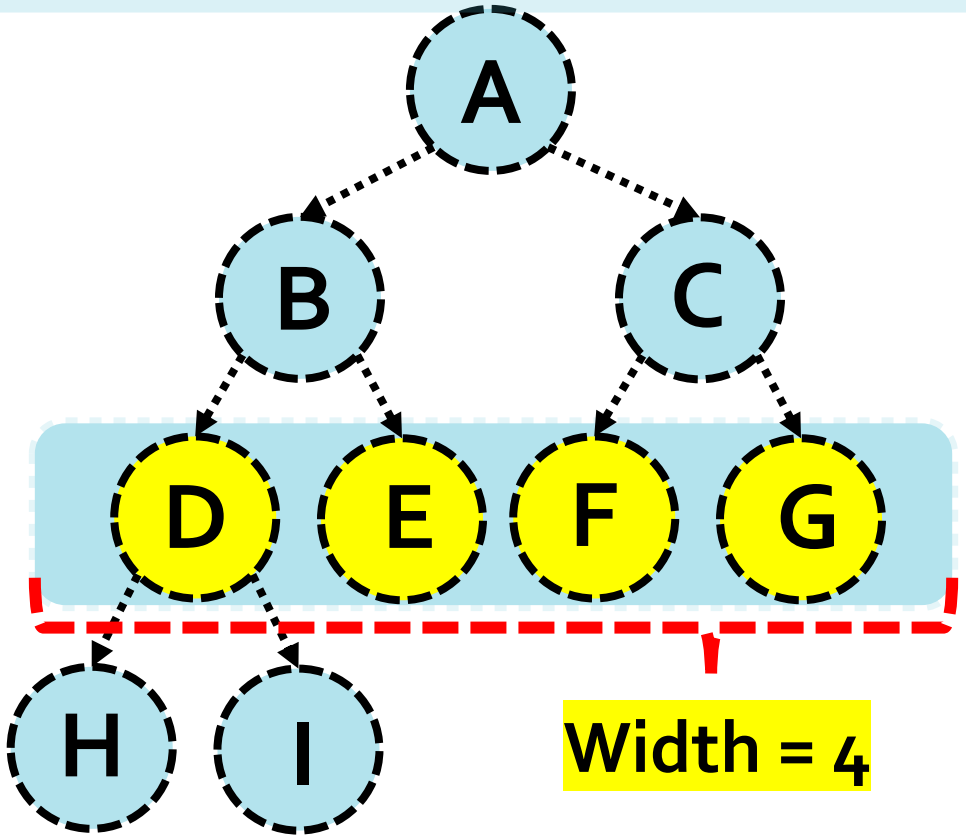
Binary Tree

- **Binary Tree:** a particular type of tree where each node could have maximum 2 children.



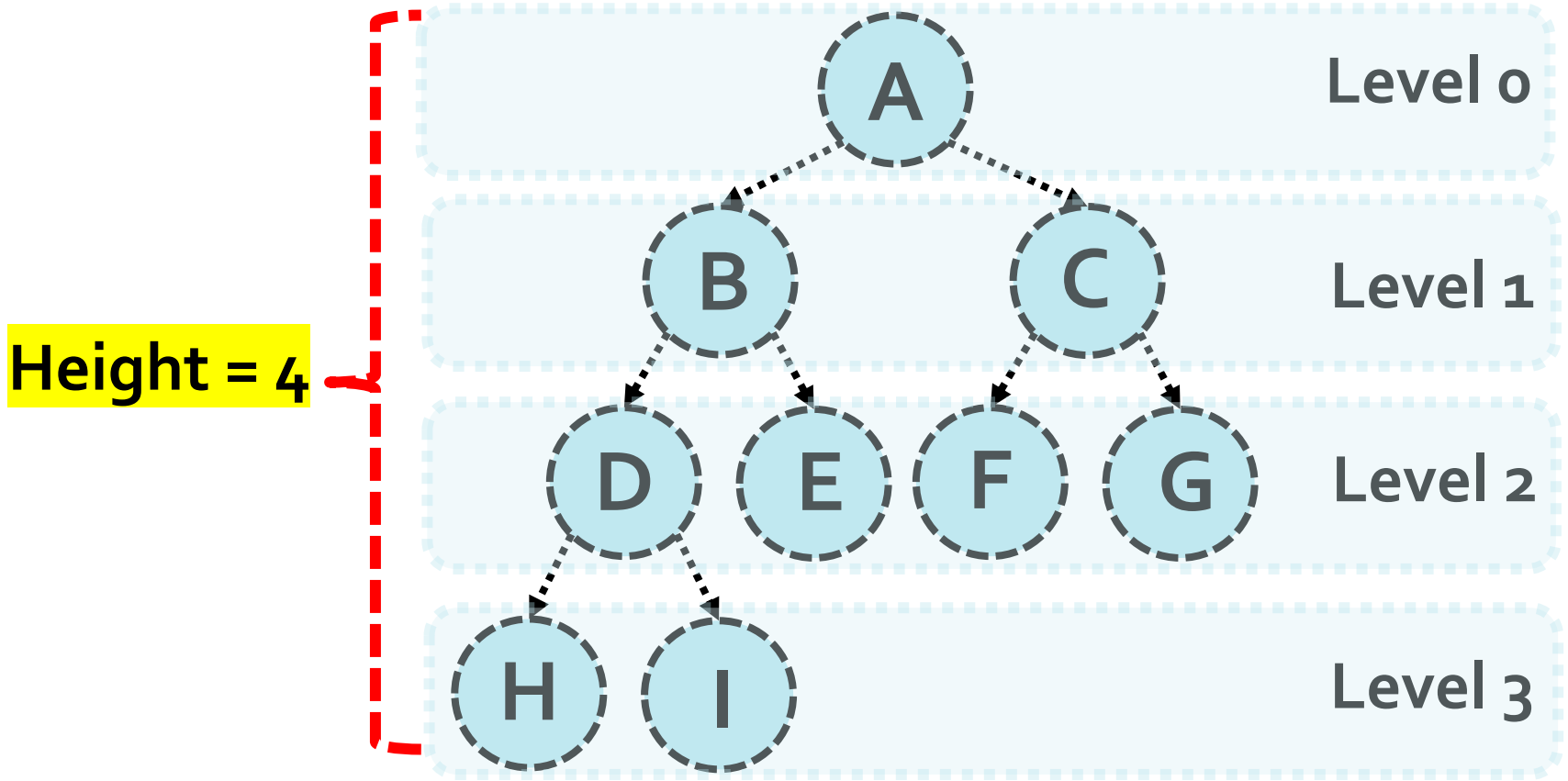
Binary Tree

- **Width of Binary Tree:** the number of nodes on the level containing the most nodes.
- **Example:** following binary trees has width of 4



Binary Tree

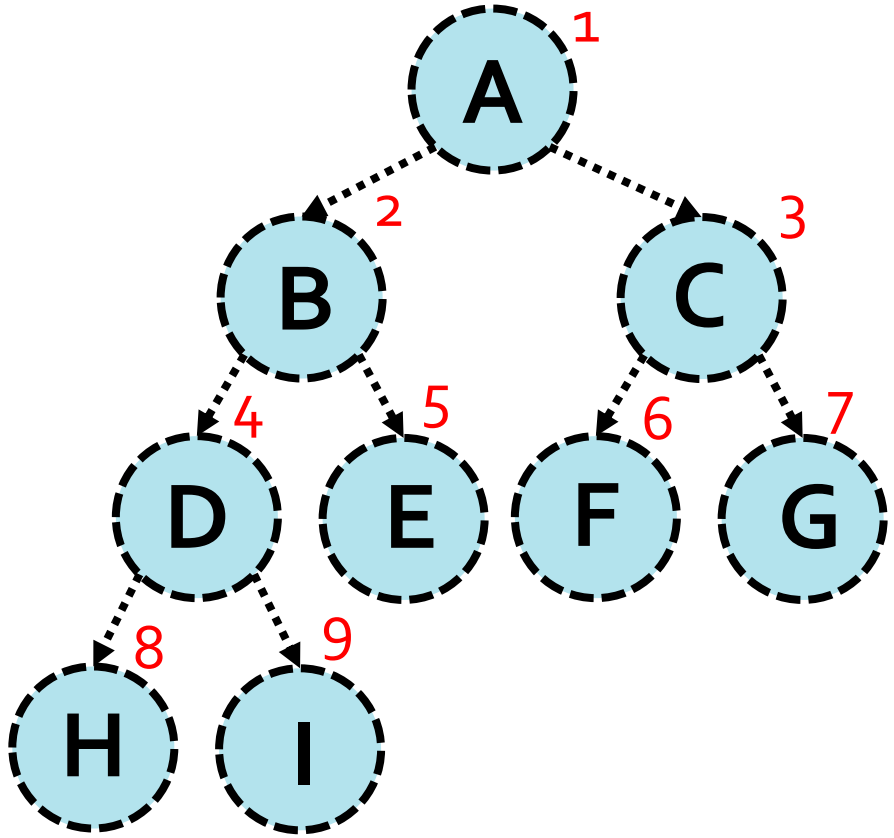
- **Height of Binary Tree:** the number of levels in the tree.
- **Example:** following binary trees has height of 4



Binary Tree

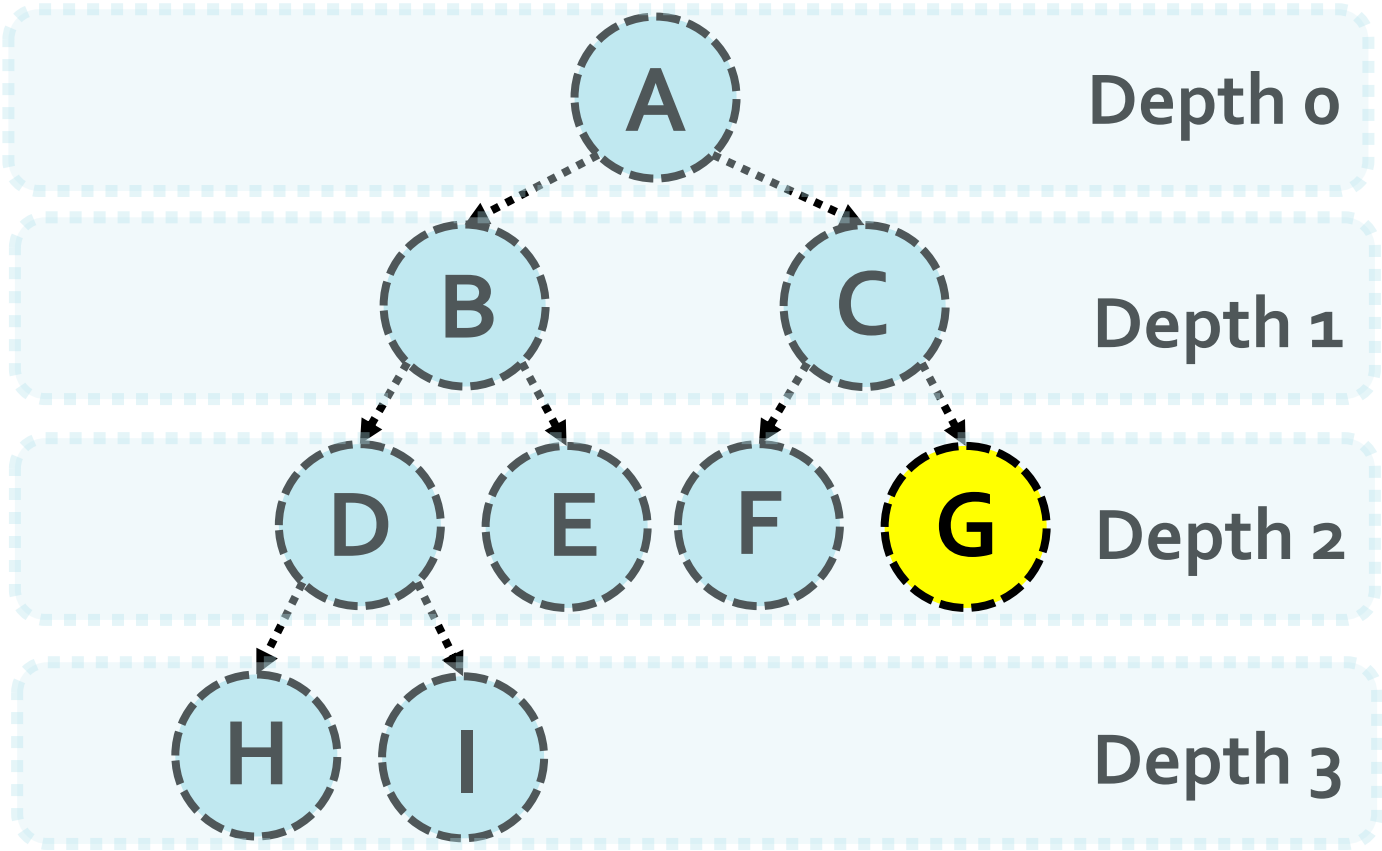
- **Size of a Binary Tree:** the number of nodes in the binary tree.
- **Example:** following binary trees has size of 9

Size = 9



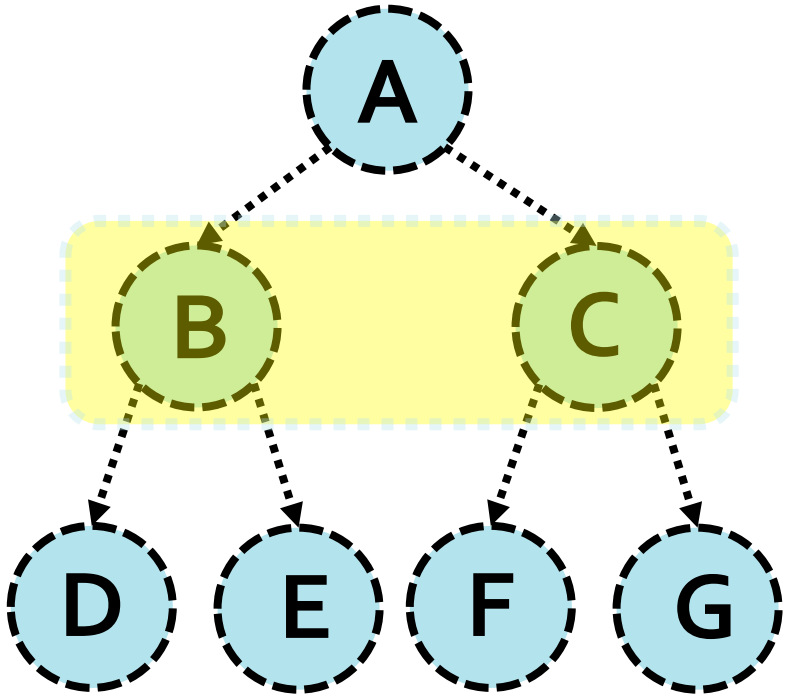
Binary Tree

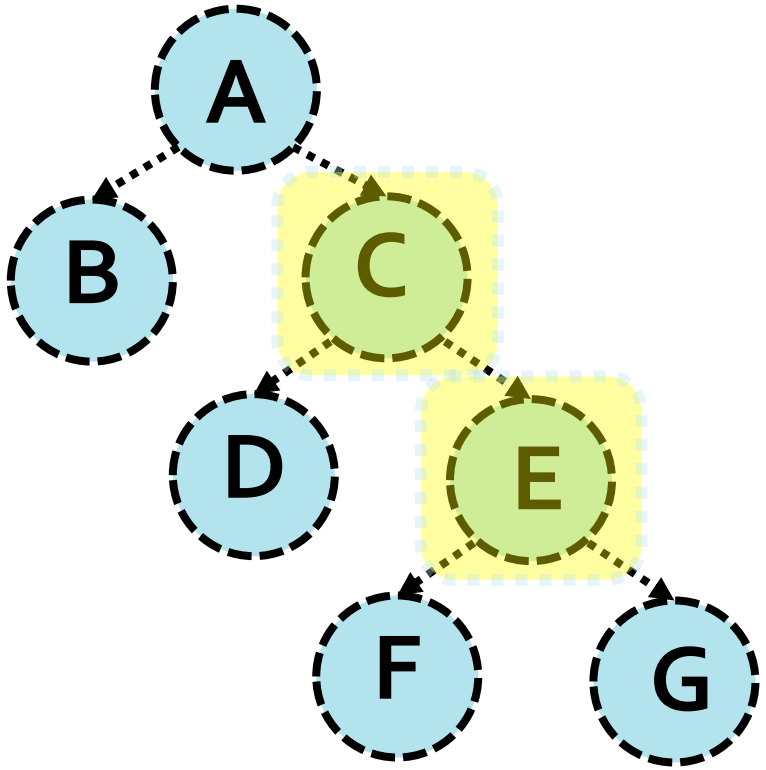
- **Depth of a Node:** the number of edges from the root to that node.
- **Example:** depth of node G is 2.



Binary Tree

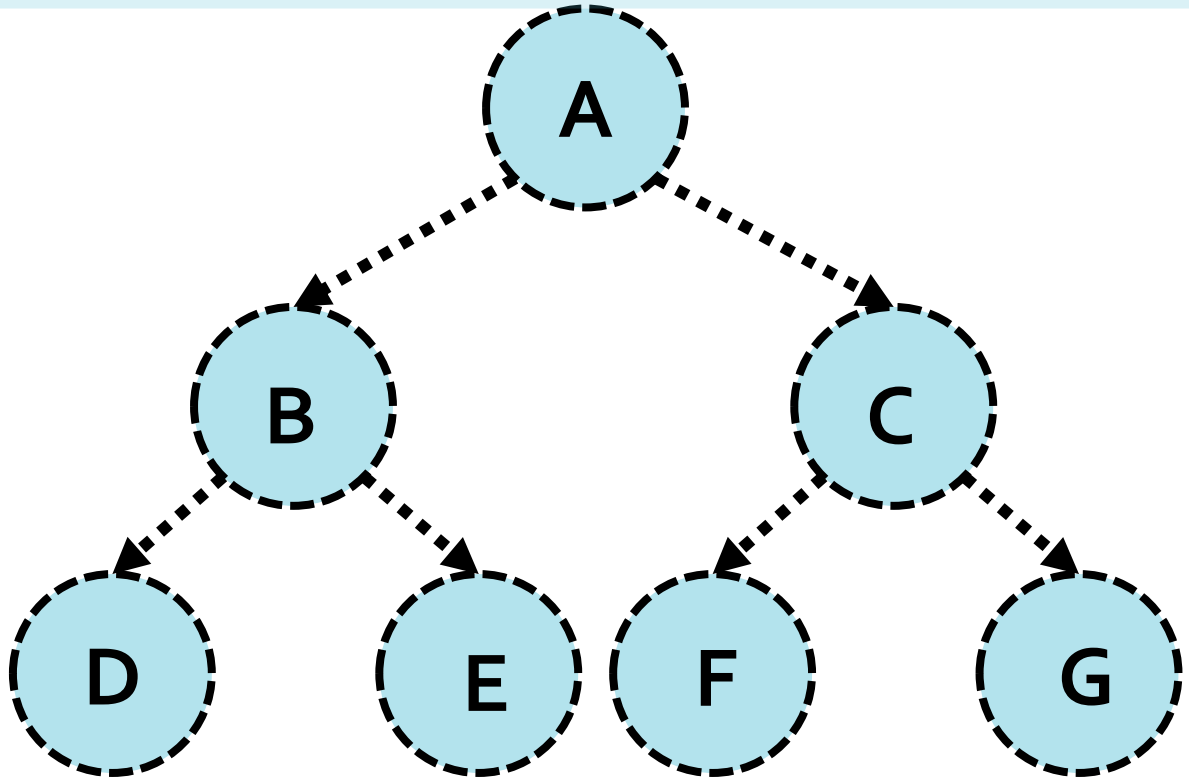
- **Full Binary Tree:** a binary tree in which each interior node contains two children.
- **Examples:** followings are both Full Binary Trees:





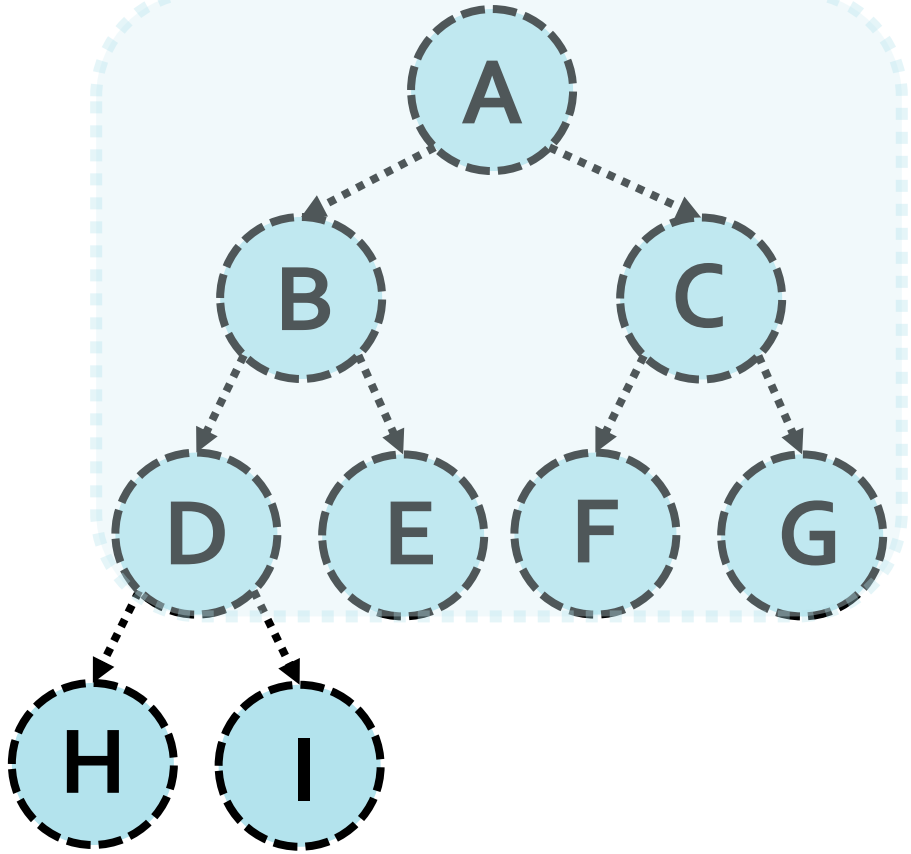
Binary Tree

- **Perfect Binary Tree:** full binary tree where all leaf nodes are at the same level.
- Perfect Binary Tree has all nodes slots filled from top to bottom with **no gaps**.
- **Example:**



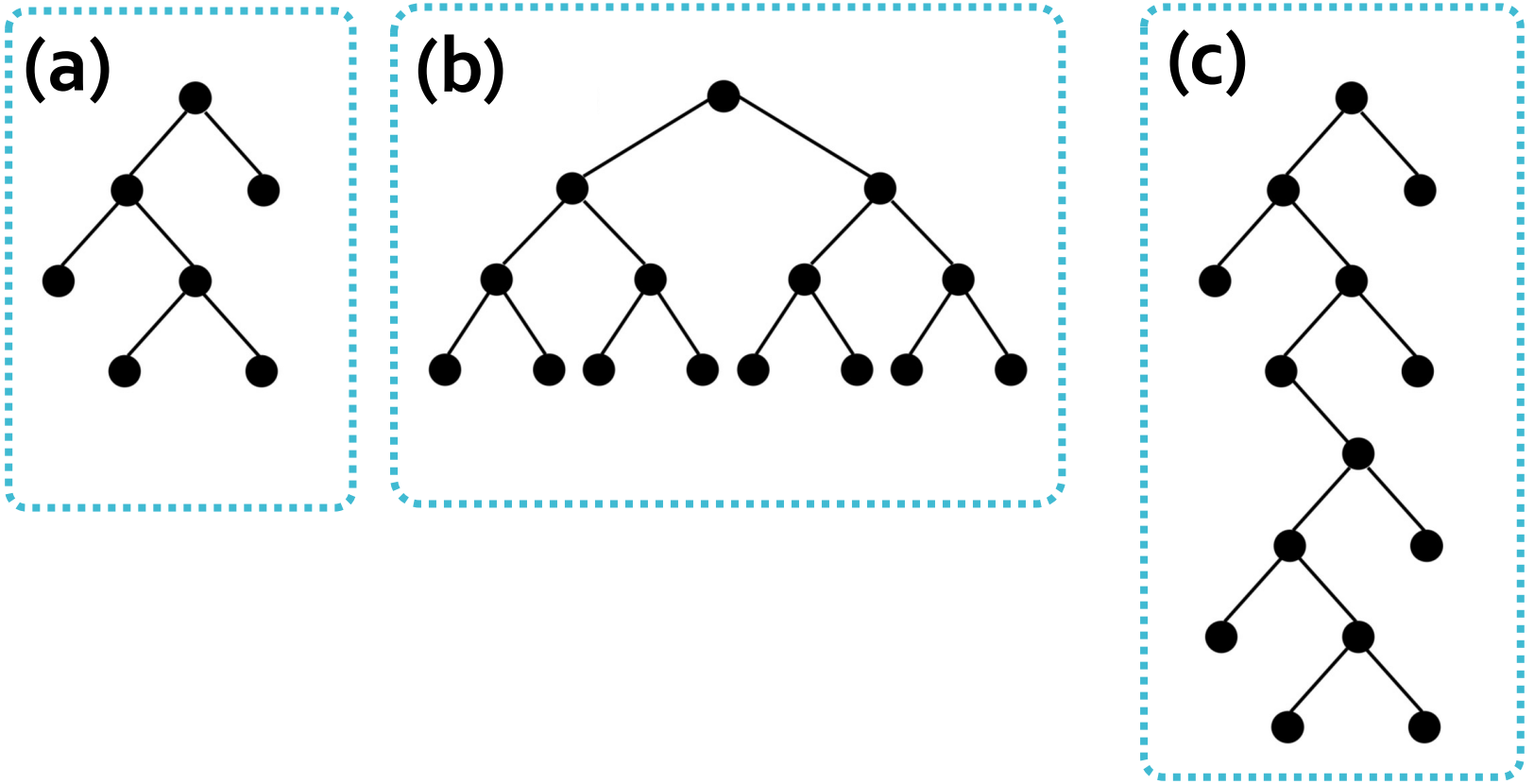
Binary Tree

- **Complete Binary Tree:** a binary tree of *height h* where it is a perfect binary tree down to the *height* *h – 1* and the nodes on the lowest level fill the available slots from left to right leaving no gaps.



Quiz

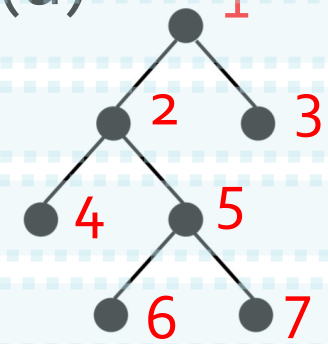
- Given the following binary trees:
  - what is the **size**, **height** & **width** of each tree?
  - which binary tree is **full**, **perfect**, or **complete**?



Answer

Size = 7
Width = 2
Height = 4

(a)



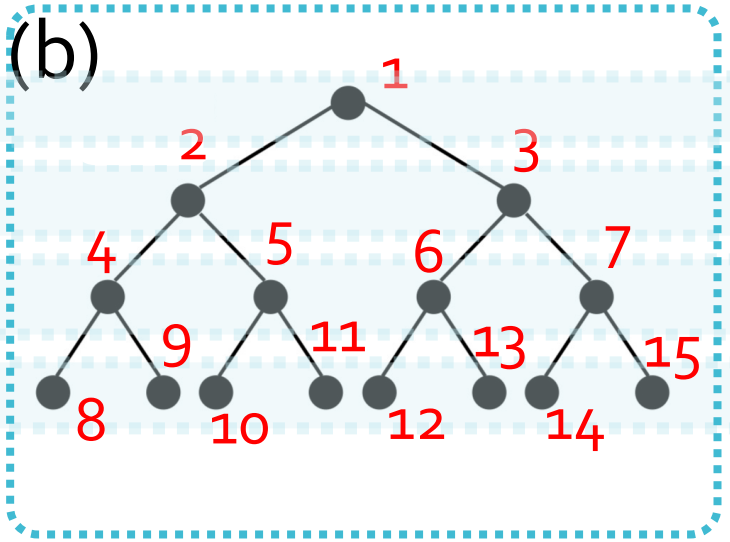
Level 0
Level 1
Level 2
Level 3

Answer

Size = 15

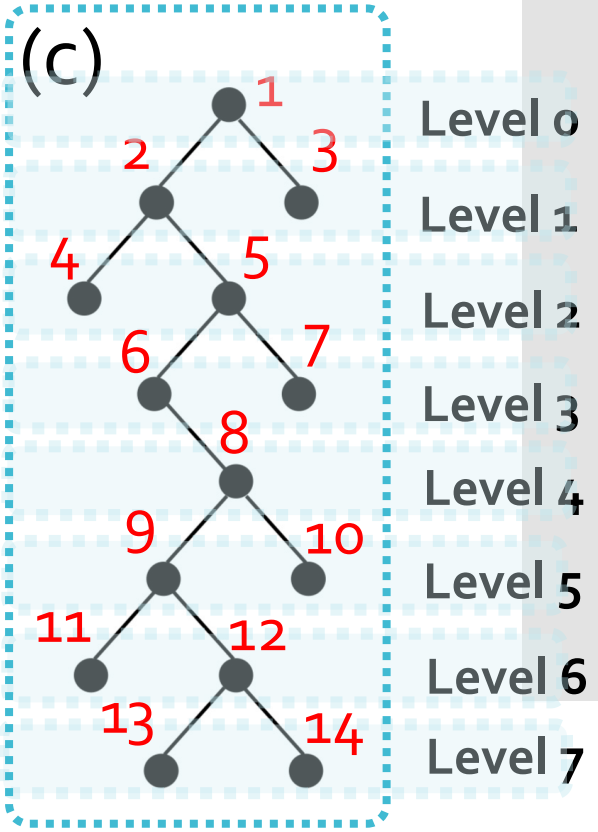
Width = 8

Height = 4



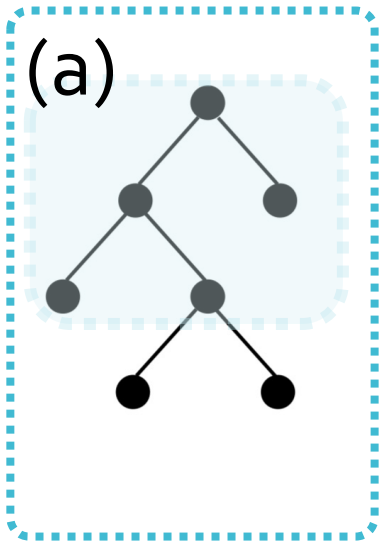
Answer

Size = 14
Width = 2
Height = 8

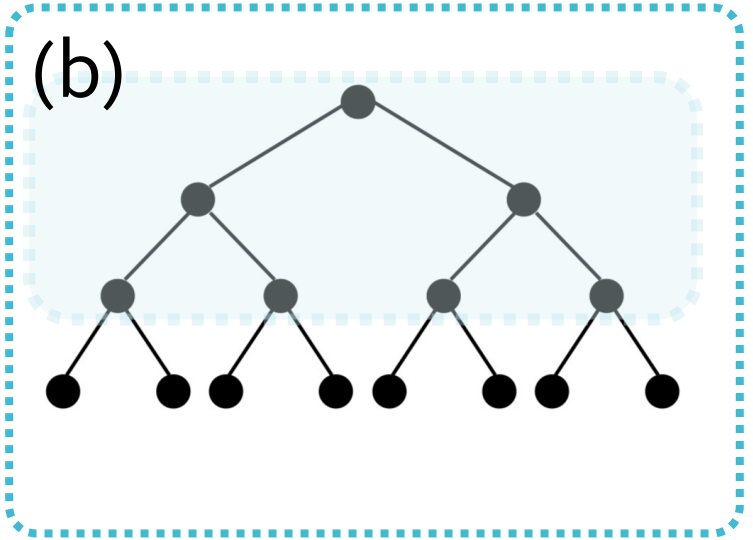


Answer

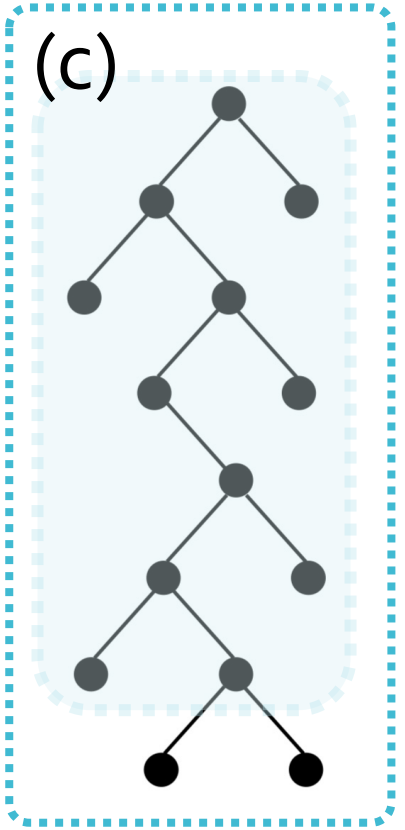
full
not perfect
not complete



full
perfect
complete



not full
not perfect
not complete



Binary Tree

- We can implement **Binary Trees** in different ways.

- Here we will try two:
  - as a **class**
  - as a **function**

Binary Tree

Implementation of **Binary Tree class (Part 1)**

```
class BinaryTree:
    def __init__(self, value):
        self.value = value
        self.left_child = None
        self.right_child = None
```

Implementation of **Binary Tree class (Part 2)**

Binary Tree

```
    def insert_left(self, value):
        if self.left_child is None:
            self.left_child = BinaryTree(value)
        else:
            new_node = BinaryTree(value)
            new_node.left_child = self.left_child
            self.left_child = new_node
```

Implementation of **Binary Tree class (Part 3)**

Binary Tree

```
    def insert_right(self, value):
        if self.right_child is None:
            self.right_child = BinaryTree(value)
        else:
            new_node = BinaryTree(value)
            new_node.right_child = self.right_child
            self.right_child = new_node
```

Binary Tree

- Implementation of **Binary Tree class (All Parts)**

```
class BinaryTree:
    def __init__(self, value):
        self.value = value
        self.left_child = None
        self.right_child = None
```

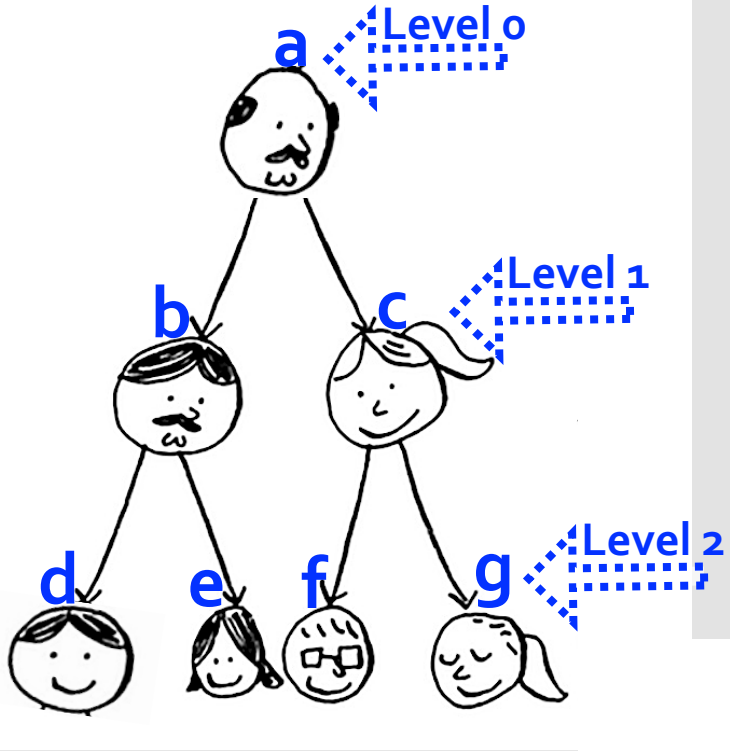
```
    def insert_left(self, value):
        if self.left_child is None:
            self.left_child = BinaryTree(value)
        else:
            new_node = BinaryTree(value)
            new_node.left_child = self.left_child
            self.left_child = new_node
```

```
    def insert_right(self, value):
        if self.right_child is None:
            self.right_child = BinaryTree(value)
        else:
            new_node = BinaryTree(value)
            new_node.right_child = self.right_child
            self.right_child = new_node
```

Binary Tree

Implementation of **Binary Tree class (testing)**

```
node_a = BinaryTree('a')

node_a.insert_left('b')
node_a.insert_right('c')

node_b = node_a.left_child
node_b.insert_left('d')
node_b.insert_right('e')

node_c = node_a.right_child
node_c.insert_left('f')
node_c.insert_right('g')

node_d = node_b.left_child
node_e = node_b.right_child
node_f = node_c.left_child
node_g = node_c.right_child
```



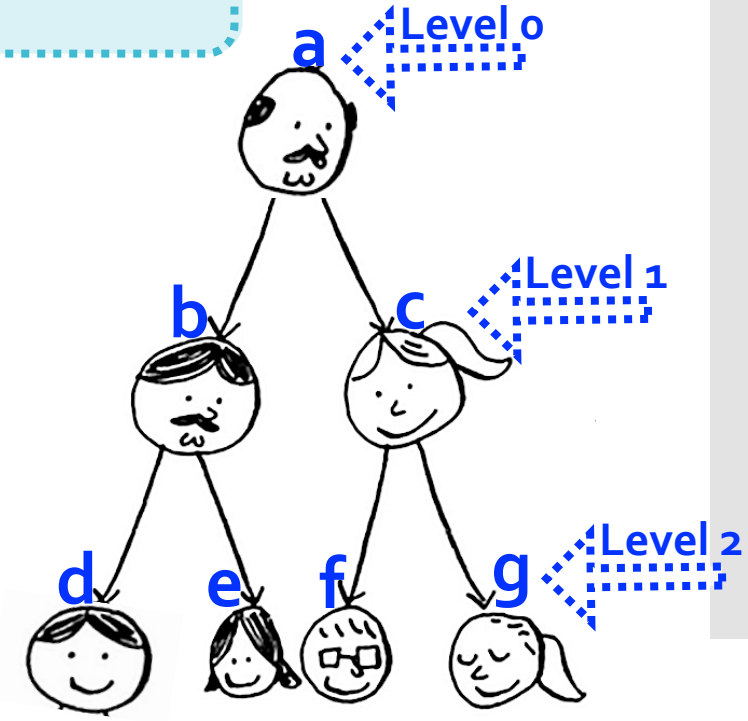
Binary Tree

❖ Implementation of **Binary Tree class (testing)**

```
print("level 0 (root):", node_a.value)
print("level 1:", node_b.value)
print("level 1:", node_c.value)
print("level 2:", node_d.value)
print("level 2:", node_e.value)
print("level 2:", node_f.value)
print("level 2:", node_g.value)
```

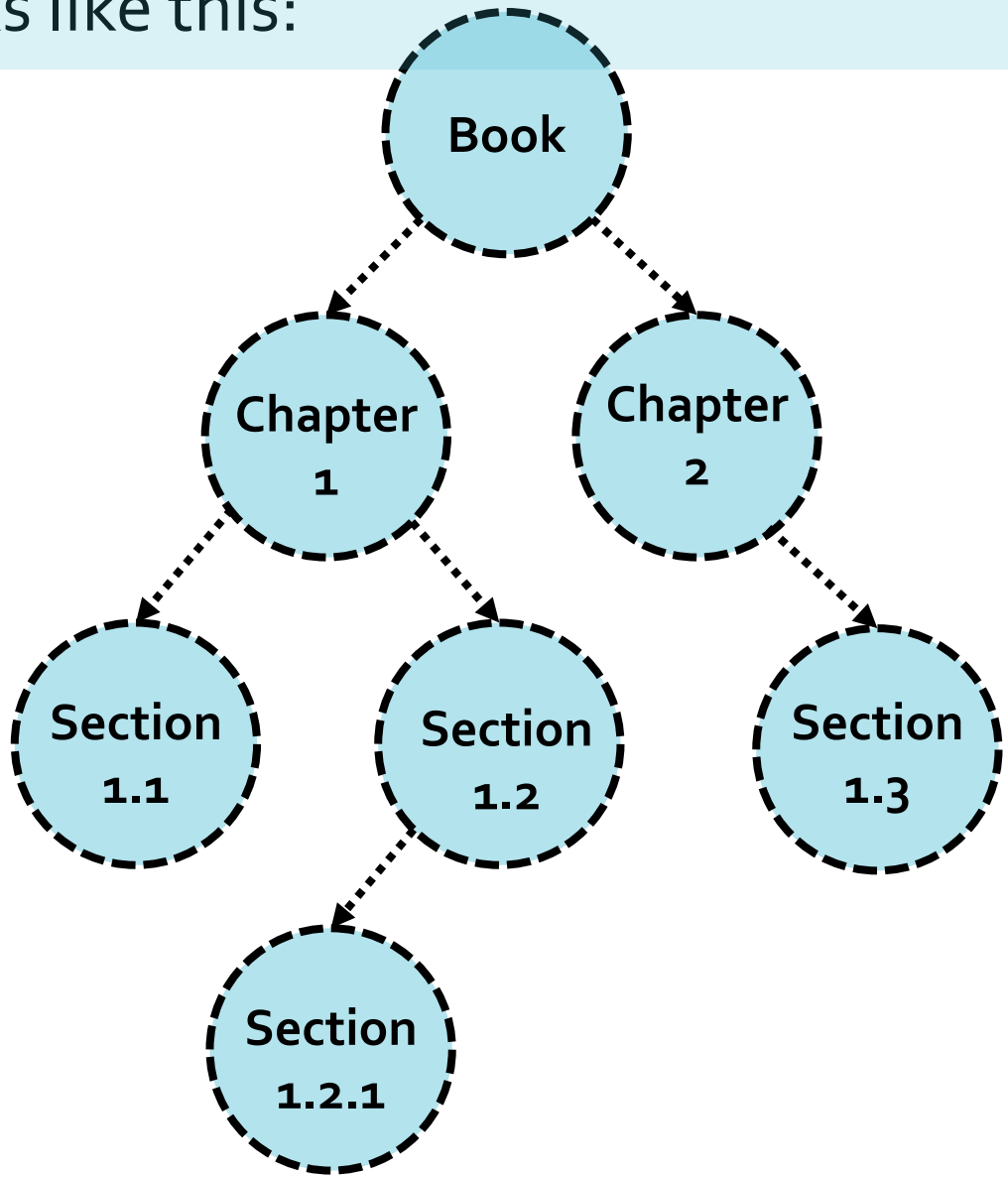
[Output:]

```
level 0 (root): a
level 1: b
level 1: c
level 2: d
level 2: e
level 2: f
level 2: g
```



Quiz

- Write a function **`build_book_tree()`** that uses the Binary Tree class and returns a binary tree that looks like this:



Answer

```
def build_book_tree():
    node_book = BinaryTree('Book')

    node_book.insert_left('Chapter 1')
    node_book.insert_right('Chapter 2')

    node_chap1 = node_book.left_child
    node_chap1.insert_left('Section 1.1')
    node_chap1.insert_right('Section 1.2')
    node_chap2 = node_book.right_child
    node_chap2.insert_right('Section 1.3')

    node_sec1_1 = node_chap1.left_child
    node_sec1_2 = node_chap1.right_child
    node_sec1_3 = node_chap2.right_child
    node_sec1_2.insert_left('Section 1.2.1')
    node_sec1_2_1 = node_sec1_2.left_child
```

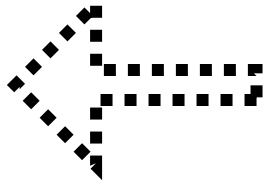
Binary Tree

- An alternative way to implementing the binary tree as a **function** based on creating List of Lists.
- Lets check it out.

Binary Tree

- Implementing **Binary Tree** with **List of Lists** (part 1)

```
def binary_tree(r):
    return [r, [], []]
```



```
def get_left_child(root):
    return root[1]
```

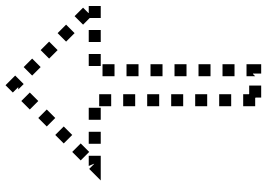
```
def get_right_child(root):
    return root[2]
```

Binary Tree

❖ Implementing **Binary Tree** with **List of Lists** (part 1)

```
def binary_tree(r):
    return [r, [], []]
```

```
def get_left_child(root):
    return root[1]
```



```
def get_right_child(root):
    return root[2]
```

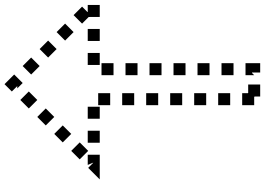
Binary Tree

❖ Implementing **Binary Tree** with **List of Lists** (part 1)

```
def binary_tree(r):
    return [r, [], []]
```

```
def get_left_child(root):
    return root[1]
```

```
def get_right_child(root):
    return root[2]
```



Binary Tree

Implementing **Binary Tree** with **List of Lists** (part 2)

```
def insert_left_child(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root
```

Binary Tree

Implementing **Binary Tree** with **List of Lists** (part 2)

```
def insert_left_child(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root


def insert_right_child(root, new_branch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2, [new_branch, [], t])
    else:
        root.insert(2, [new_branch, [], []])
    return root
```

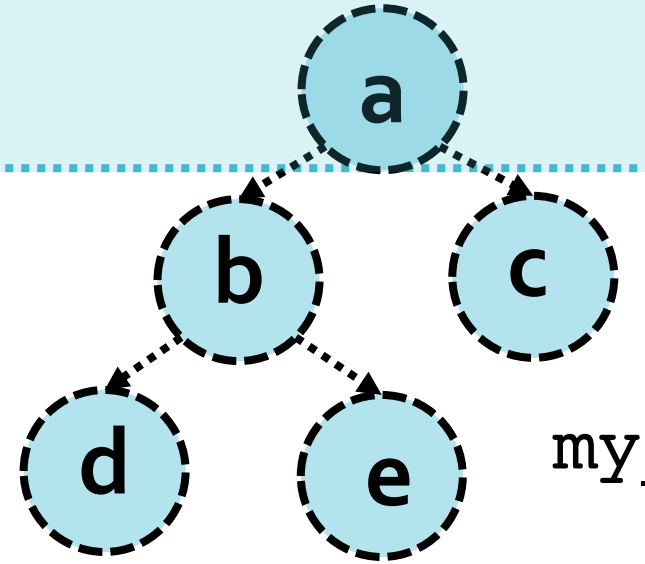
Quiz

- What is the printed output of the following code snippet?

```
my_tree = binary_tree('a')

insert_left_child(my_tree, 'b')
insert_right_child(my_tree, 'c')

insert_left_child(get_left_child(my_tree), 'd')
insert_right_child(get_left_child(my_tree), 'e')

print(my_tree)
```

Answer

```
my_tree = binary_tree('a')

insert_left_child(my_tree, 'b')
insert_right_child(my_tree, 'c')

insert_left_child(get_left_child(my_tree), 'd')
insert_right_child(get_left_child(my_tree), 'e')

print(my_tree)
```



my_tree

[Output:]

```
['a', ['b', ['d', [], []], ['e', [], []]], ['c', [], []]]
```

Heap

- **Heap:** a complete binary tree in which the nodes are organized based on their data entry values.

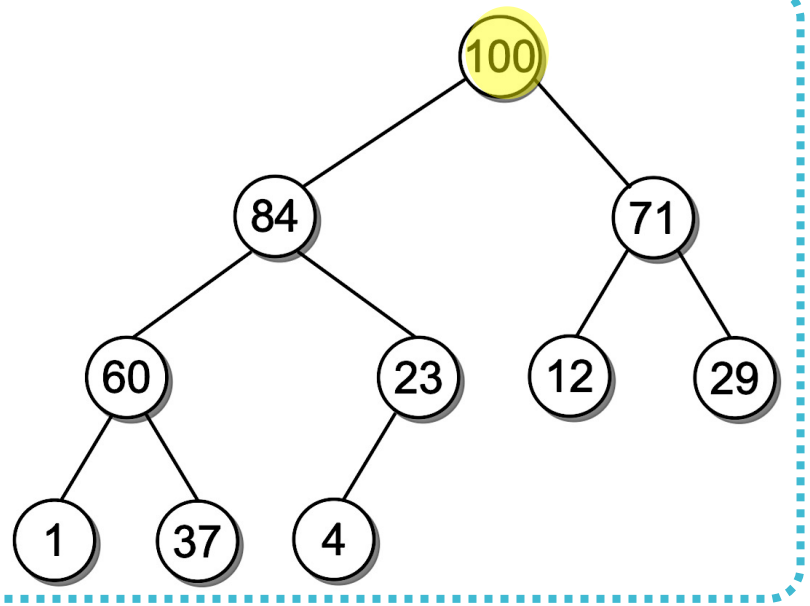
- Types of of heap structure:

  1) **Max-heap:** for each non-leaf node, the value is greater than the value of its children (heap order property).

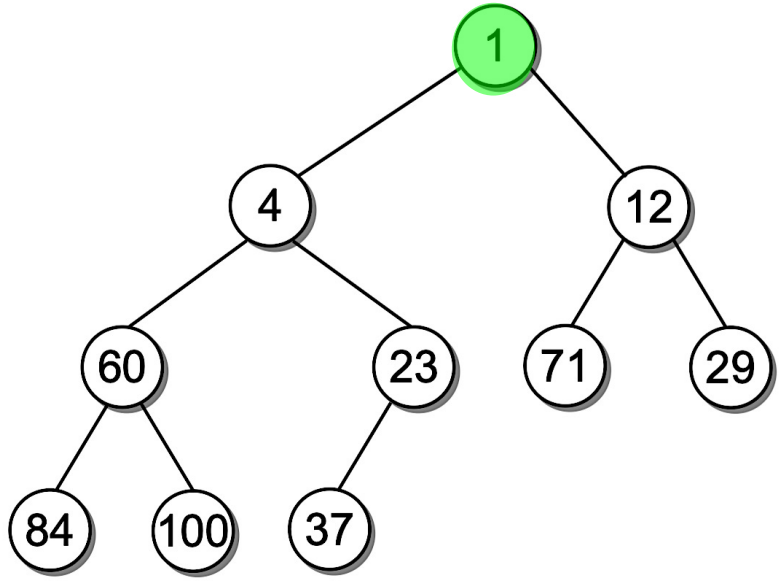
  2) **Min-heap:** for each non-leaf node, the value is smaller than the value of its children.

Heap

- **Max-heap:**
  - root -> largest
  - leaf nodes -> smallest

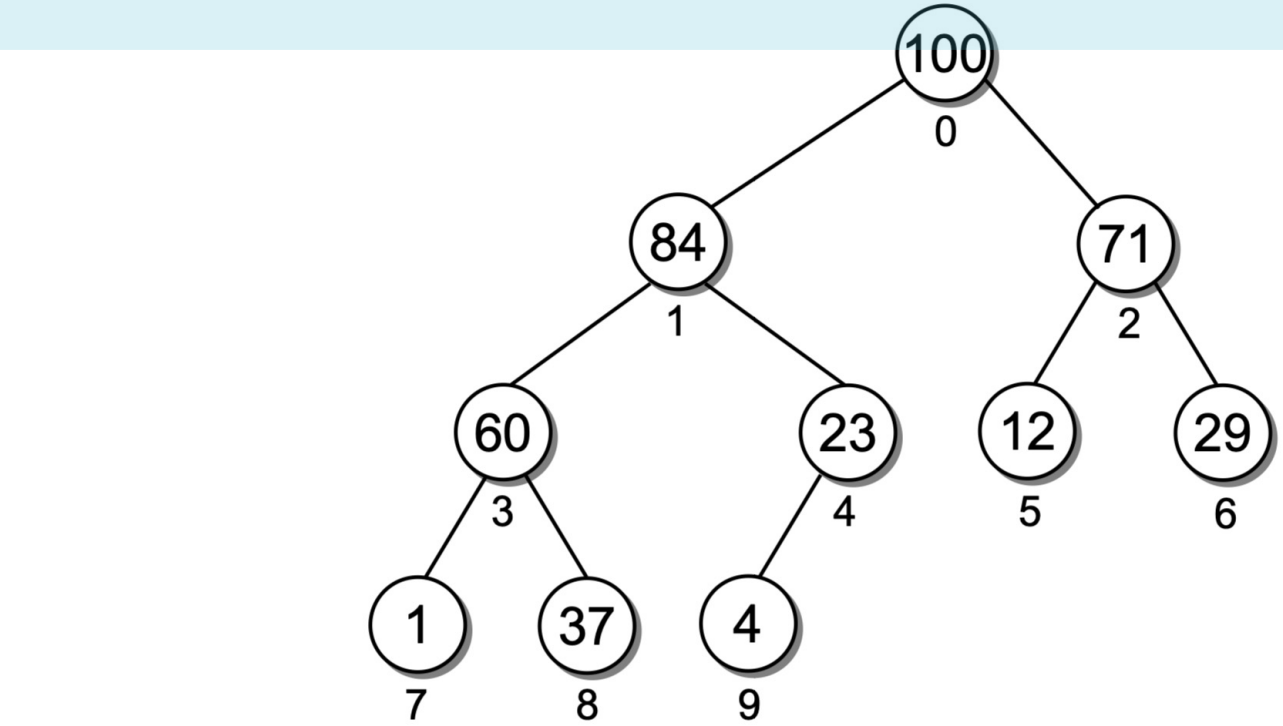


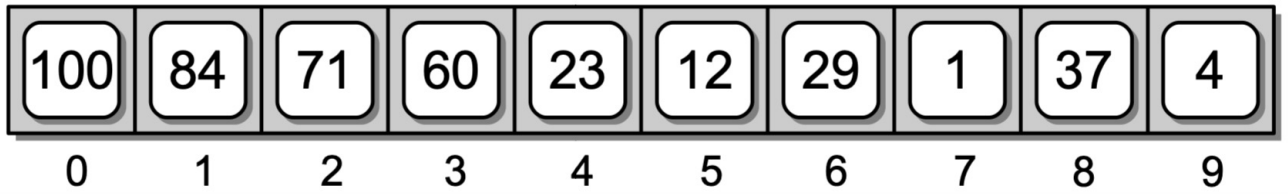
- **Min-heap:**
  - root -> smallest
  - leaf nodes -> largest



Heap Sort

- **Heap** can be implemented using an array or list.





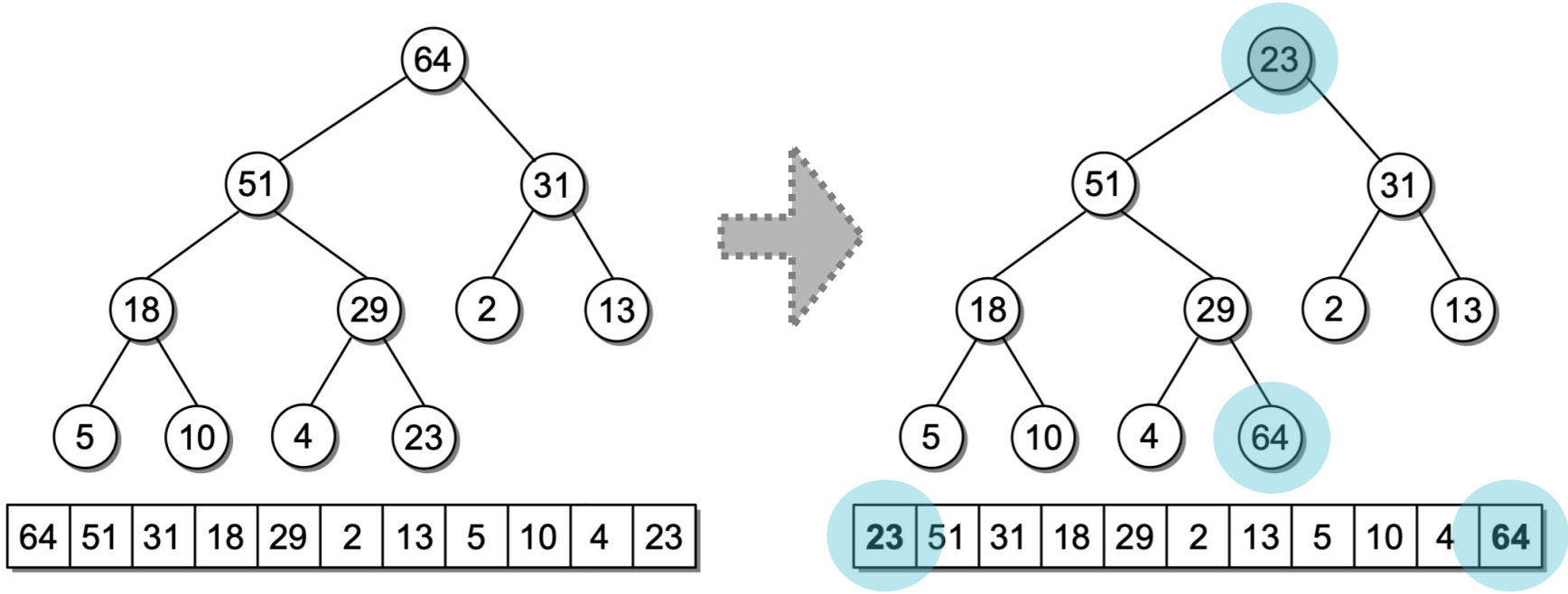
Heap Sort

- Heap data structure can be used for **sorting**.
- First we build a **Max-heap** from unsorted array.
- This can be done by **`heapify()`** function.
- This function assumes part of the array is sorted.

- Lets check an example.

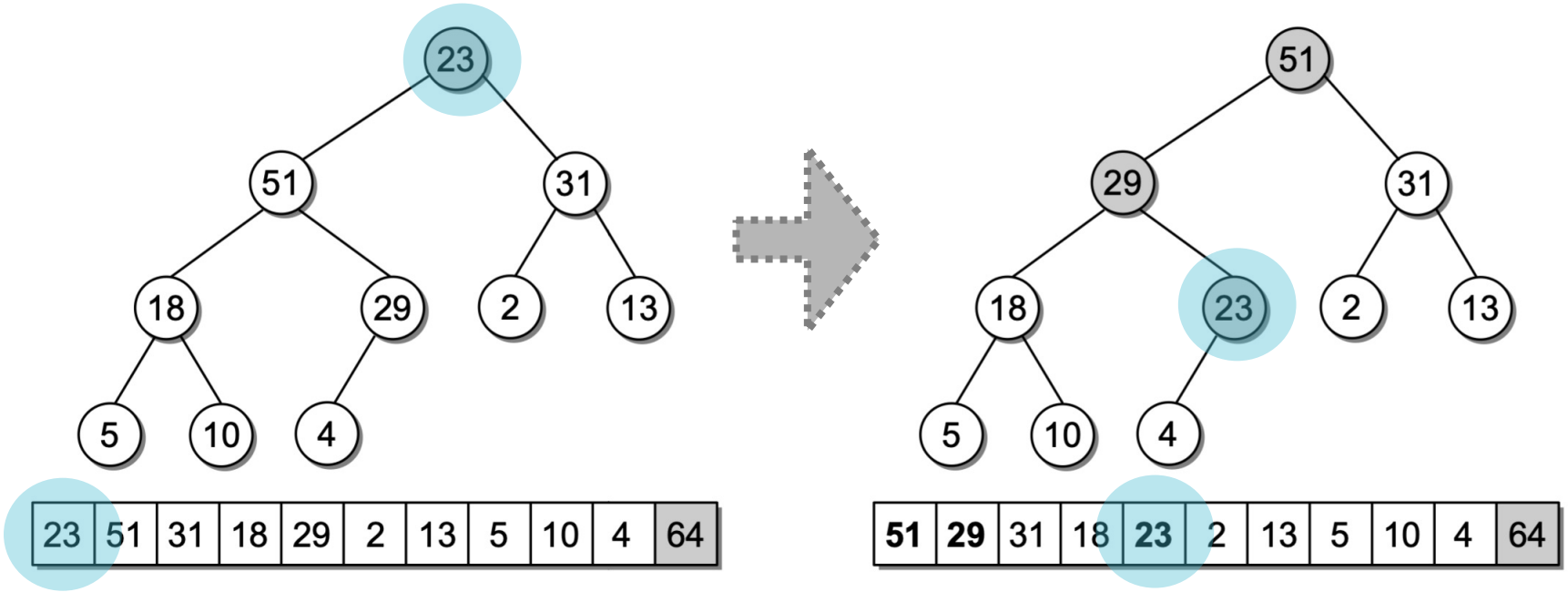
Heap Sort

- First, **swap** the **first** and **last** items in the heap.



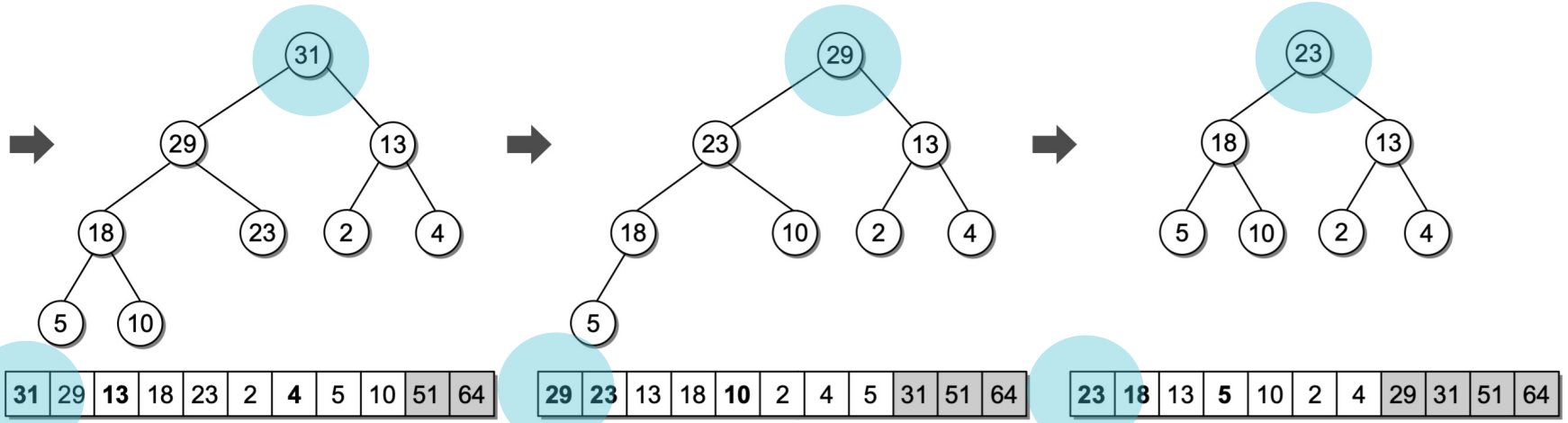
Heap Sort

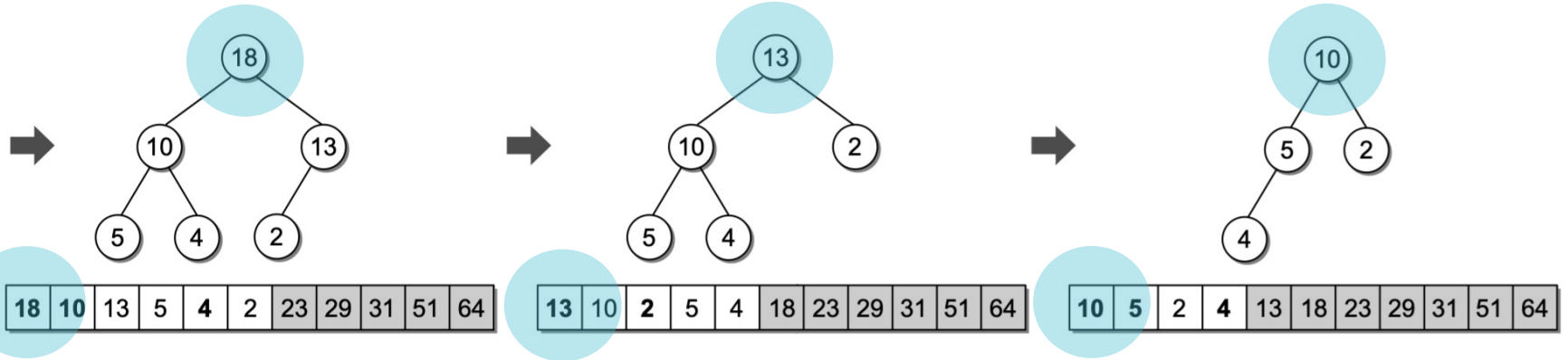
- Then, **remove the last item** from the heap and shift the root value down the tree.



Heap Sort

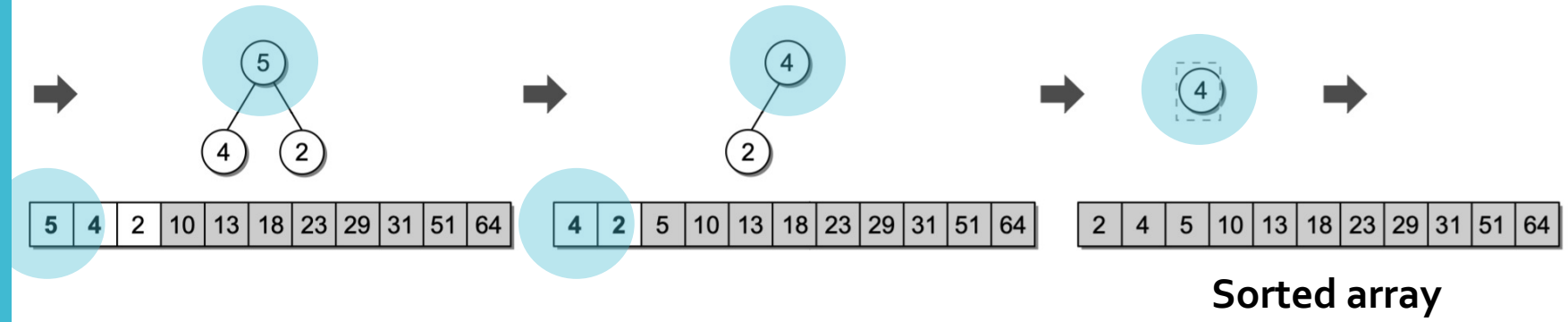
- **Repeat** this process again:
  - **swapping** the **root** with the last item in the subarray (heap)



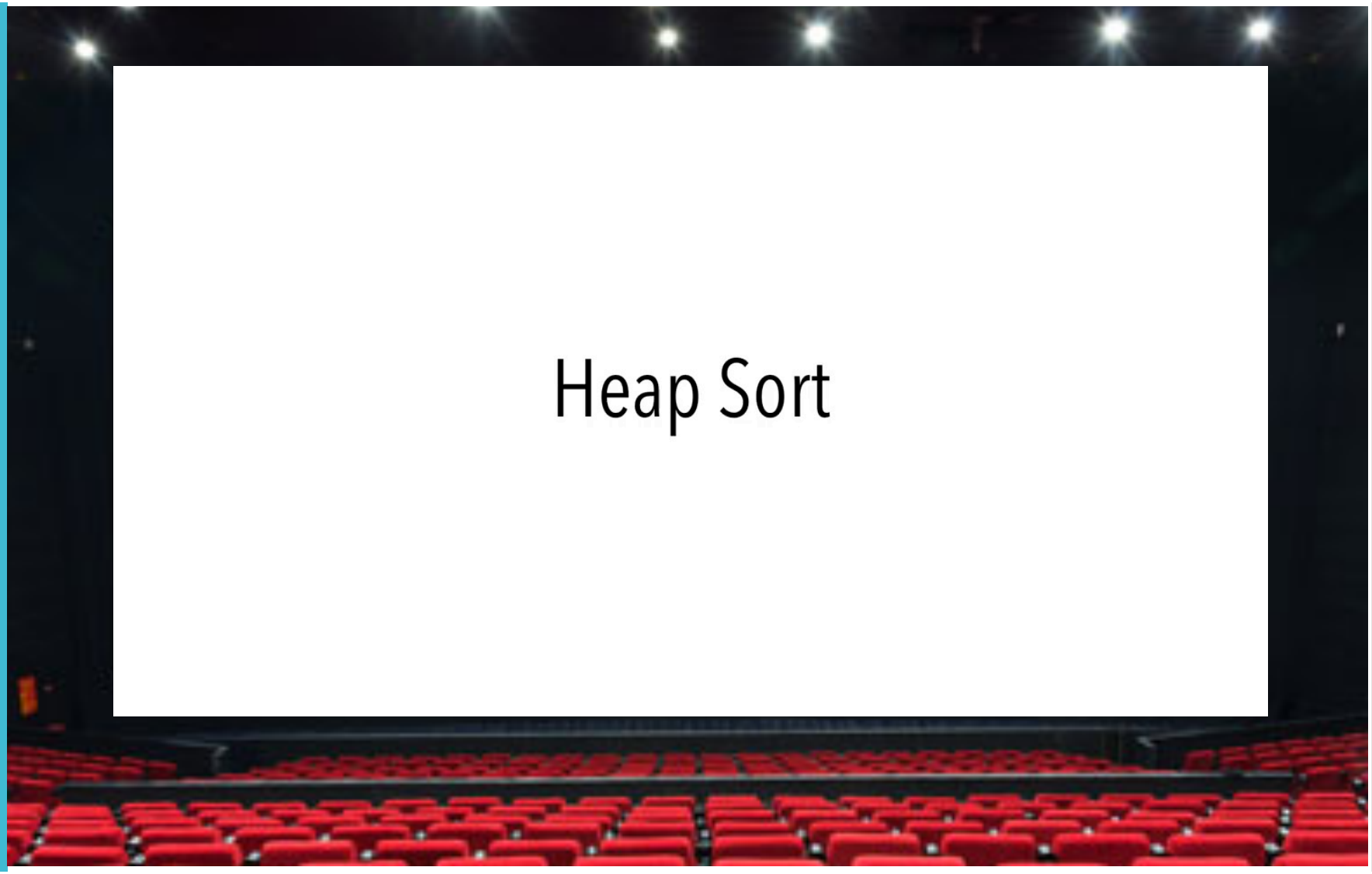


Heap Sort

- At the end, we end up with a **sorted array** of values in ascending order.



Heap sort



ref : youtu.be/2DmK_H7IdTo

Heap sort

❖ Implementation of **Heap Sort (part 1)**

```
def heap_sort(arr):
    size = len(arr)

    for i in range(size, 0, -1):
        heapify(arr, size, i)

    for i in range(size - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Heap sort

Implementation of **Heap Sort (part 2)**

```
def heapify(array, n, i):
    maximum = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and array[i] < array[left]:
        maximum = left

    if right < n and array[maximum] < array[right]:
        maximum = right

    if maximum != i:
        array[i], array[maximum] = \
            array[maximum], array[i]

        heapify(array, n, maximum)
```

Heap sort

Implementation of **Heap Sort (part 2)**

```
def heapify(array, n, i):
    maximum = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and array[i] < array[left]:
        maximum = left

    if right < n and array[maximum] < array[right]:
        maximum = right

    if maximum != i:
        array[i], array[maximum] = \
            array[maximum], array[i]

        heapify(array, n, maximum)
```

Heap sort

Implementation of **Heap Sort (part 2)**

```
def heapify(array, n, i):
    maximum = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and array[i] < array[left]:
        maximum = left

    if right < n and array[maximum] < array[right]:
        maximum = right

    if maximum != i:
        array[i], array[maximum] = \
            array[maximum], array[i]

        heapify(array, n, maximum)
```

Heap sort

❖ Implementation of **Heap Sort (part 3)**

```
my_array = [100, 84, 71, 60, 23, 12, 29, 1, 37, 4]

heap_sort(my_array)
print('Sorted array:', my_array)
```

[Output:]

```
Sorted array: [1, 4, 12, 23, 29, 37, 60, 71, 84, 100]
```

Heap Sort

	Best case	Average case	Worse case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)

Next Lesson

- Greedy Algorithms