PROGRAMMING AND DATA STRUCTURES

BINARY TREES

HOURIA OUDGHIRI

FALL 2023

OUTLINE

- Binary trees
- Properties of binary trees
- Special Binary Trees
  - Heap (characteristics, operations, implementation)
  - Binary Search Tree (characteristics, operations, implementation)

STUDENT LEARNING OUTCOMES

At the end of this lecture, you should be able to:

- Describe the properties of binary trees in general and for the special trees Heap and BST
- Trace the operations on the Heap and the BST
- Implement the Heap and the BST generic data structures
- Determine the time complexity of the operations on the Heap and the BST

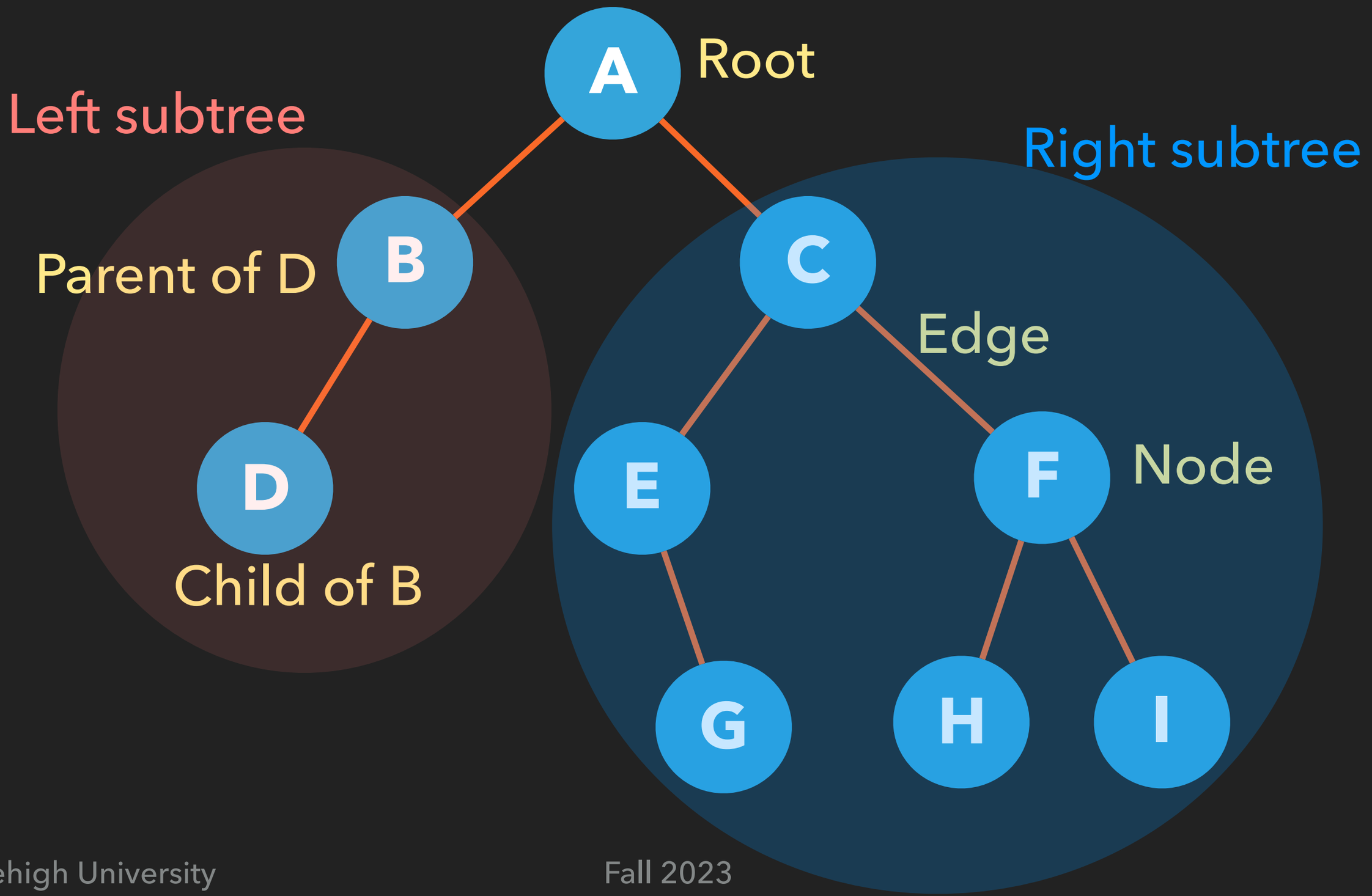
What is a binary tree?

- Data organized in a binary tree structure
- Easy and efficient access and update in large collections of data
- Used for efficient sort/search operations
- Wide range of applications: mathematical expressions, game strategies, decision trees, data compression, ...

Properties

- Set of elements called nodes (vertices) interconnected with edges (arcs)
- The first node is called the root
- The root is connected to two binary trees (left subtree and right subtree)
- Every node has a parent (except the root) and may have no child, one child or two children at most
- The root is the ancestor of all the nodes in the tree

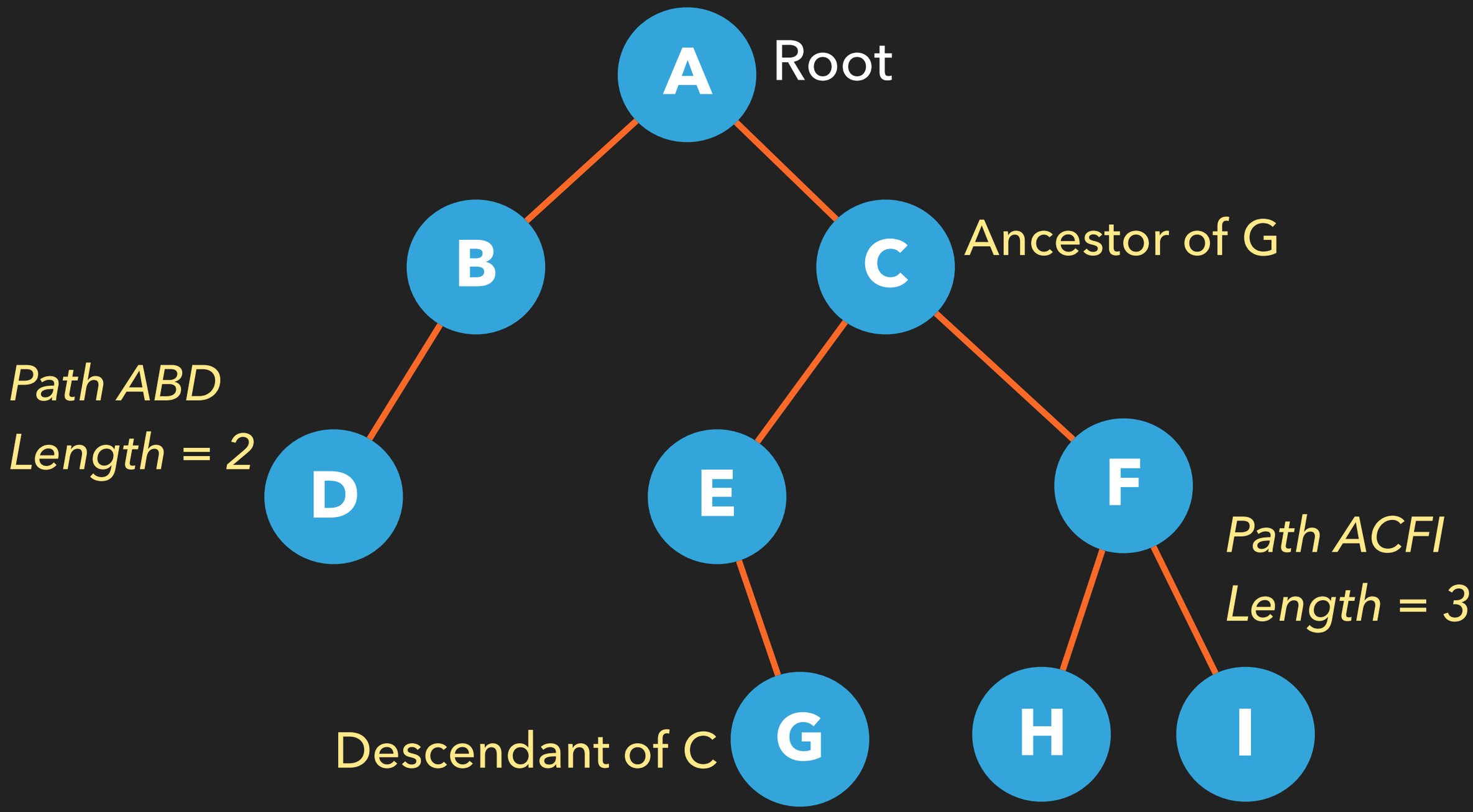
Properties


A
Root
Left subtree
Right subtree
B
C
Parent of D
Edge
D
E
F
Node
Child of B
G
H
I

Properties

- Path: Sequence of connected nodes starting at any level of the tree
- Length of a path: the number of nodes in the sequence - 1 (number of edges)
- If there is a path from node P to node Q, Q is the descendant of P and P is an ancestor of Q

Properties

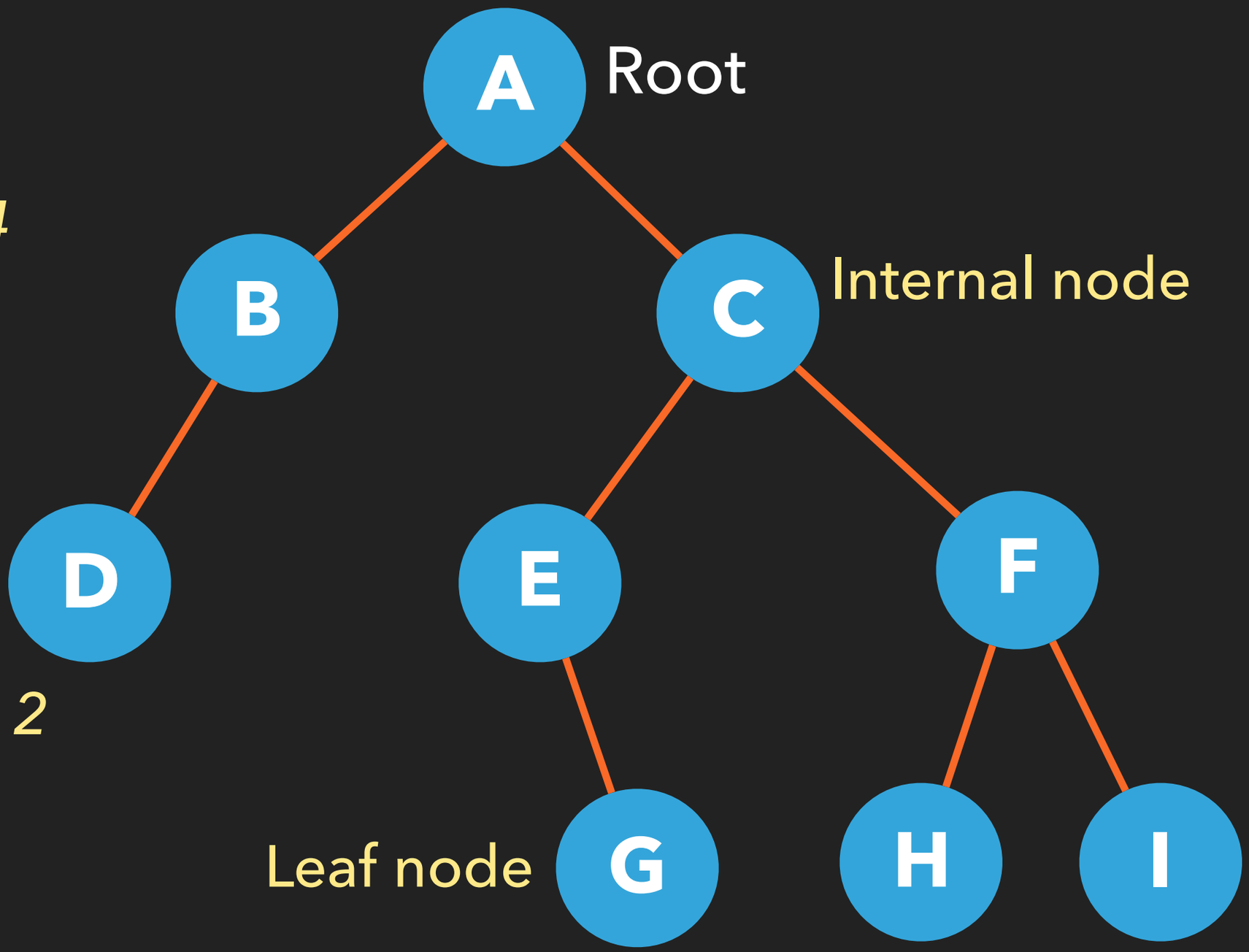

A
Root
B
C
Ancestor of G
Path ABD
Length = 2
D
E
F
Path ACFI
Length = 3
Descendant of C
G
H
I

Properties

- Depth of a node: Length of the path from the root to the node
- Height of a tree: the depth of the deepest node + 1
- Leaf node: node with no children
- Internal node: node with at least one child

Properties

Height = 4

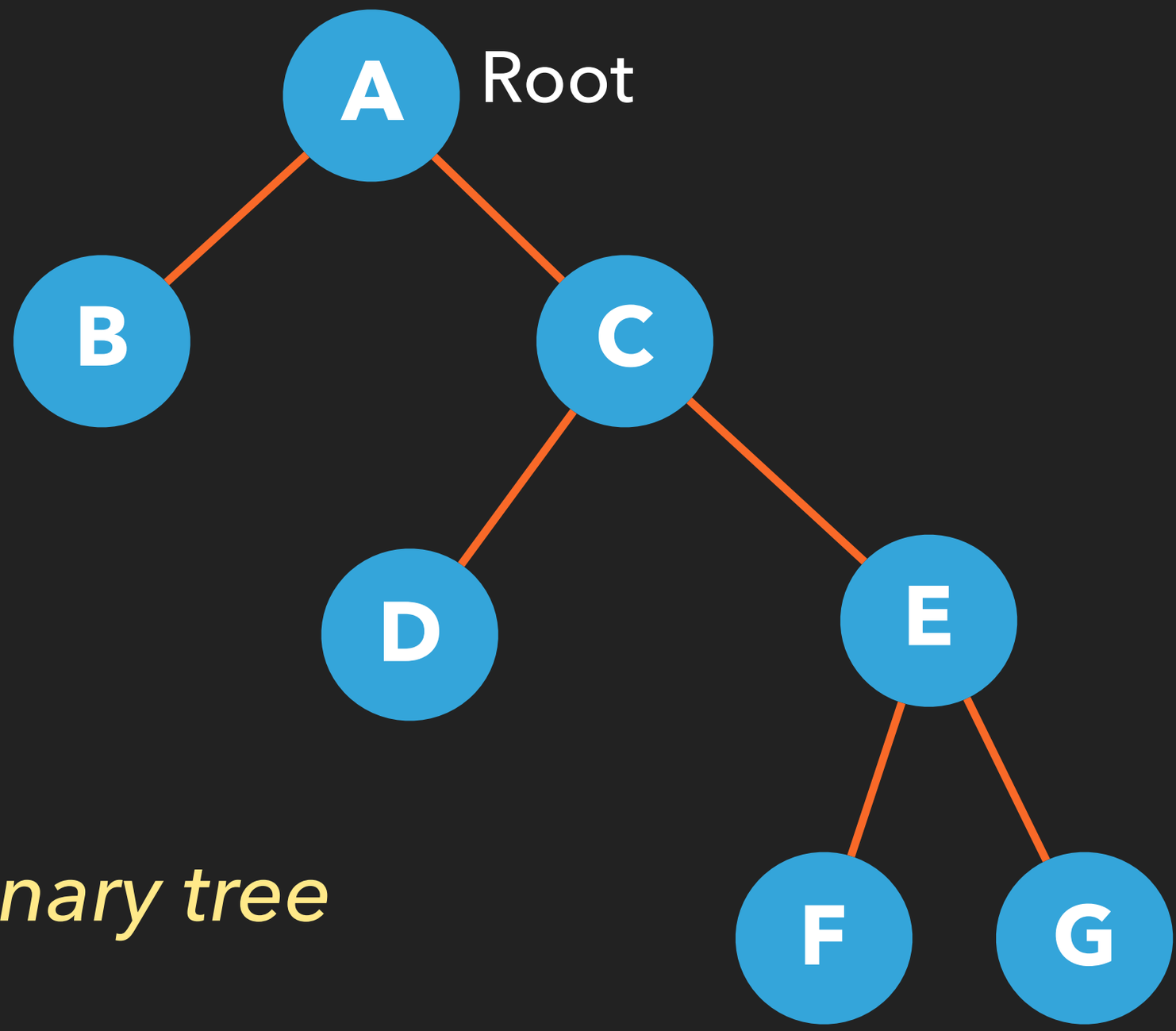


Properties

- **Full Binary Tree**: each node is a leaf or an internal node with exactly two children

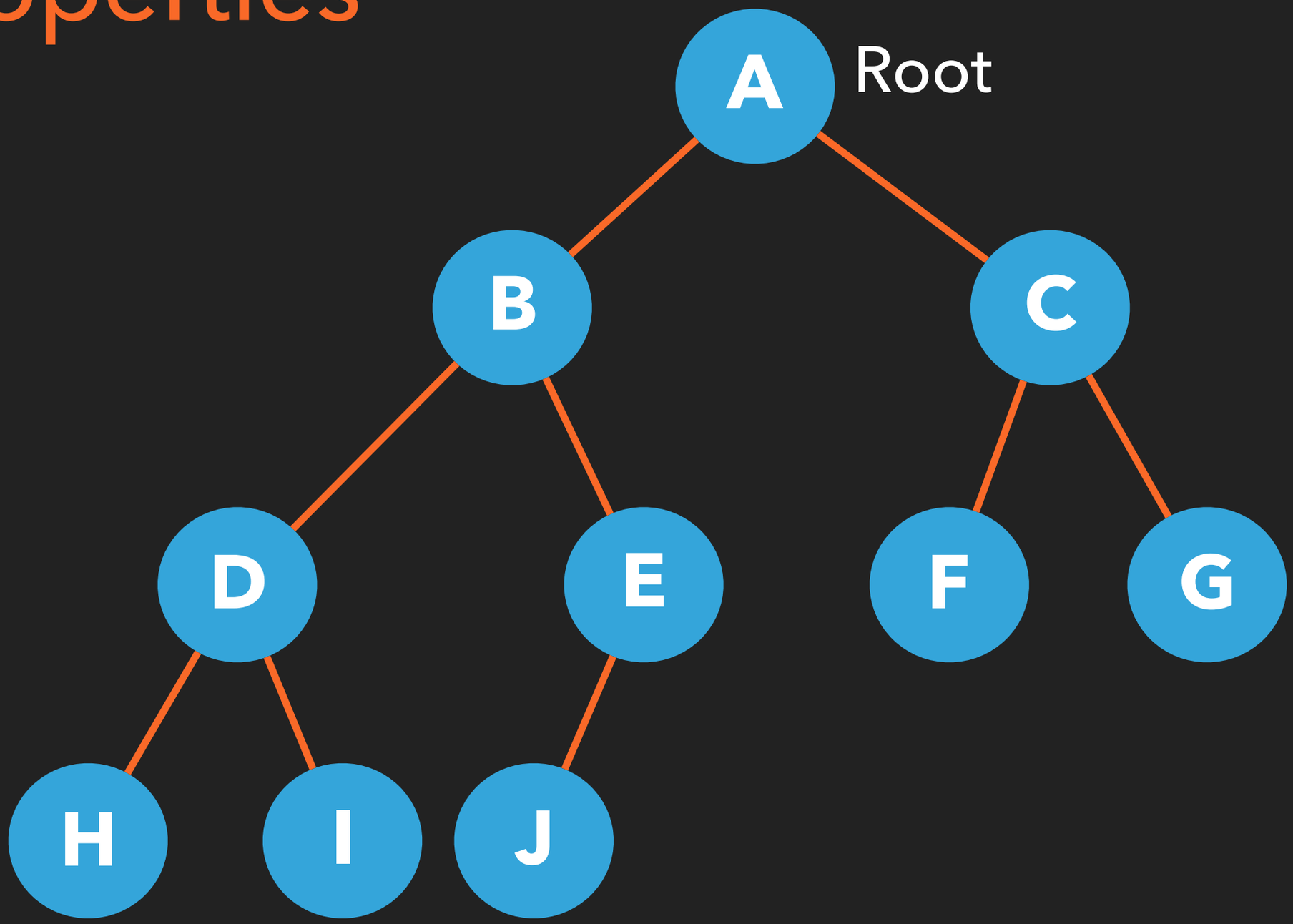
- **Complete Binary Tree**: Every level is filled (two children) except the last level, and the leaves on the last level are placed leftmost

Properties



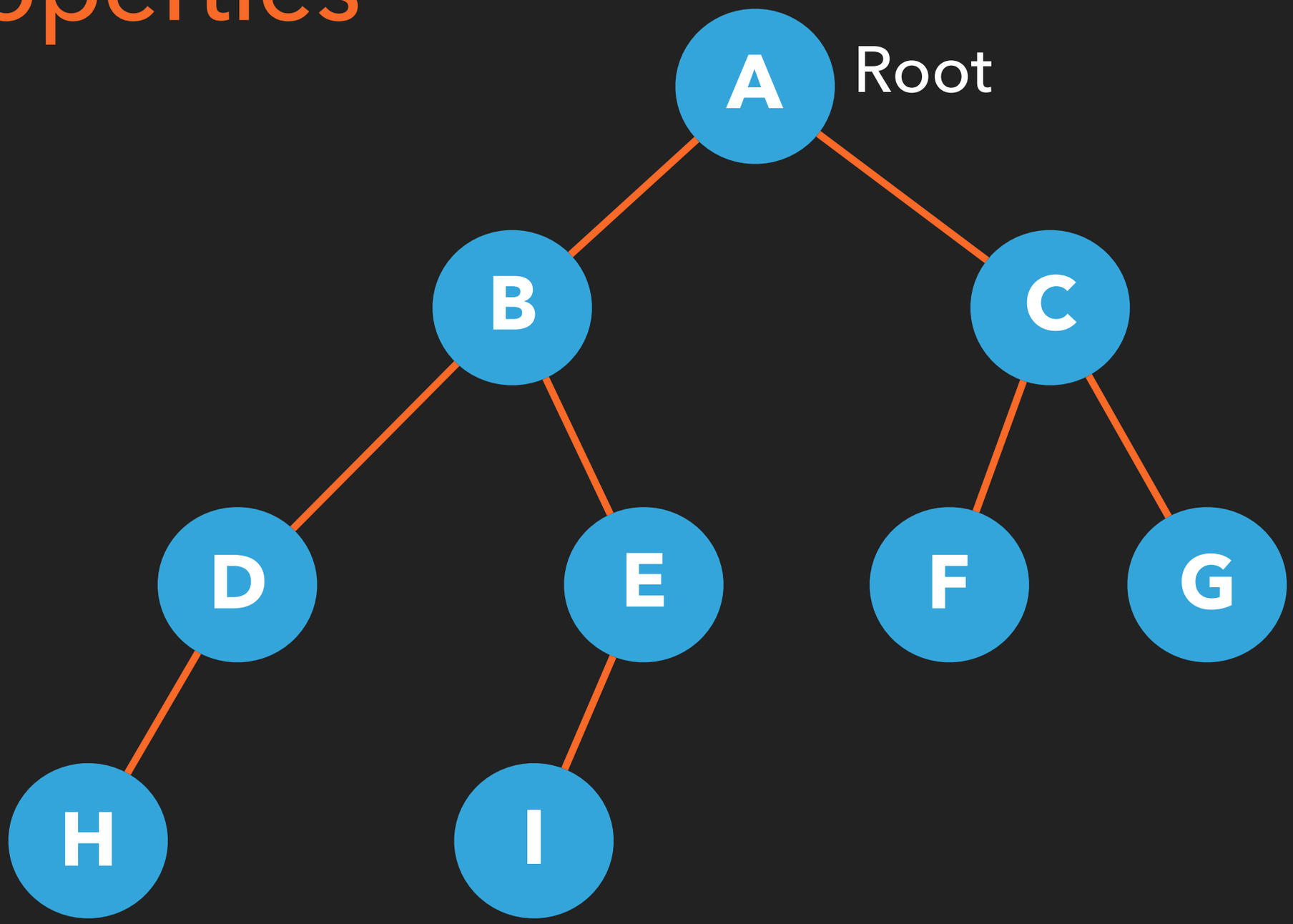
Full binary tree

Properties



Complete binary tree but not full

Properties

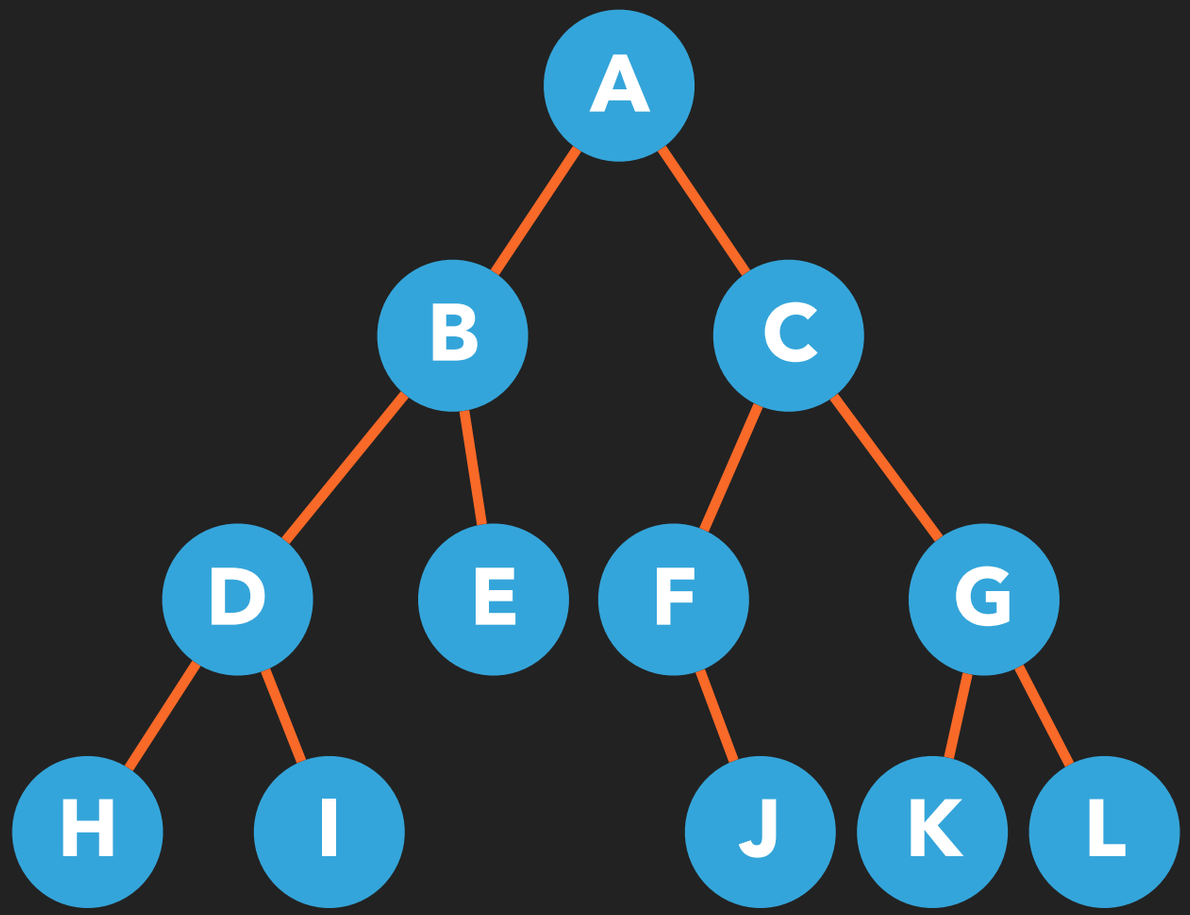


Not Complete - not full

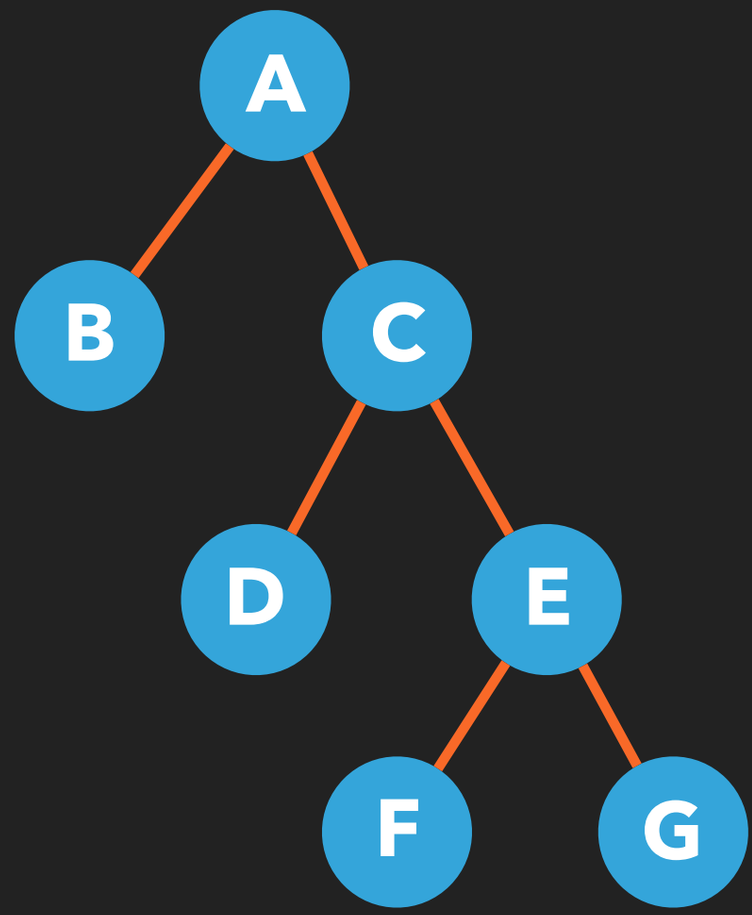
Properties

- Balanced Binary Tree: for each node, the height of the left subtree and the height of the right subtree differ by at most 1

Properties



Balanced binary tree



Unbalanced binary tree

Practice

Root?

Depth(35)?

Path from 70 to 72?

Height of the tree?

Leaf nodes?

Internal nodes?

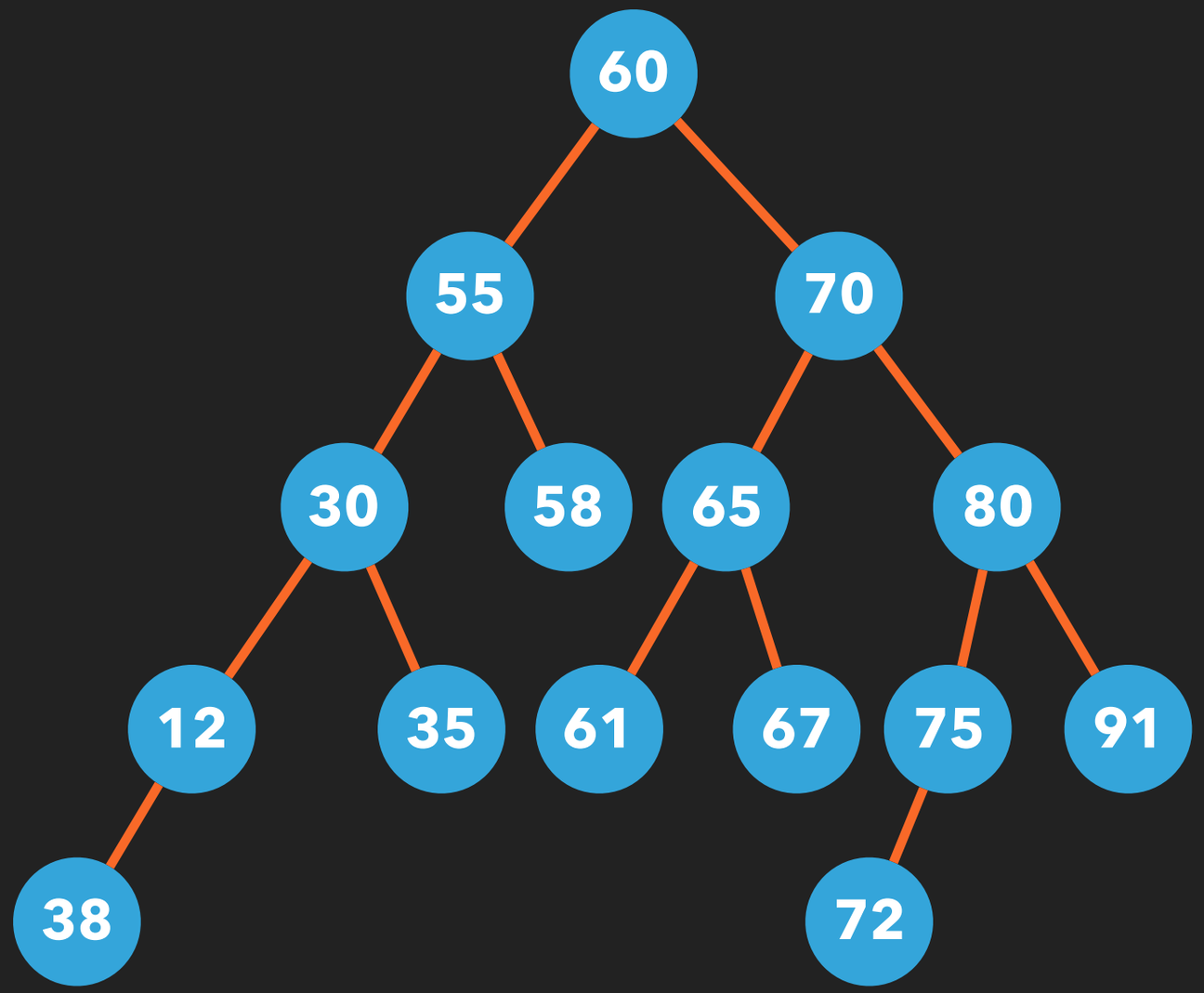
Children of 55?

Descendants of 55?

Full?

Complete?

Balanced?



Binary Tree Traversal

- Any process of visiting all the nodes in the tree is called traversal
- Three common traversals
  - Preorder
  - Inorder
  - Postorder

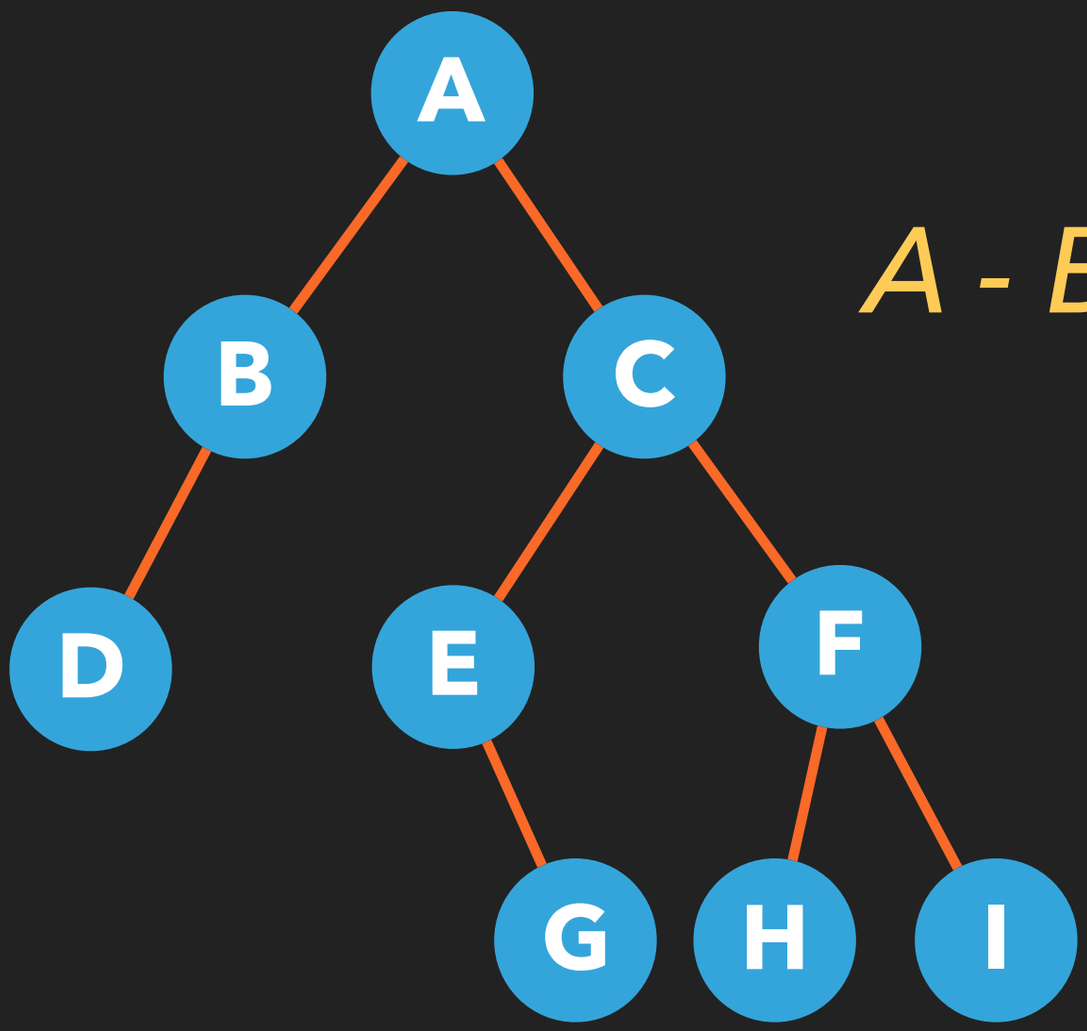
Preorder Traversal

- Any node is visited before its children
- V-L-R: Visit Node - Go Left - Go Right

- Visit the node
- Traverse the left subtree
- Traverse the right subtree

Preorder Traversal

- V-L-R: Visit Node - Go Left - Go Right



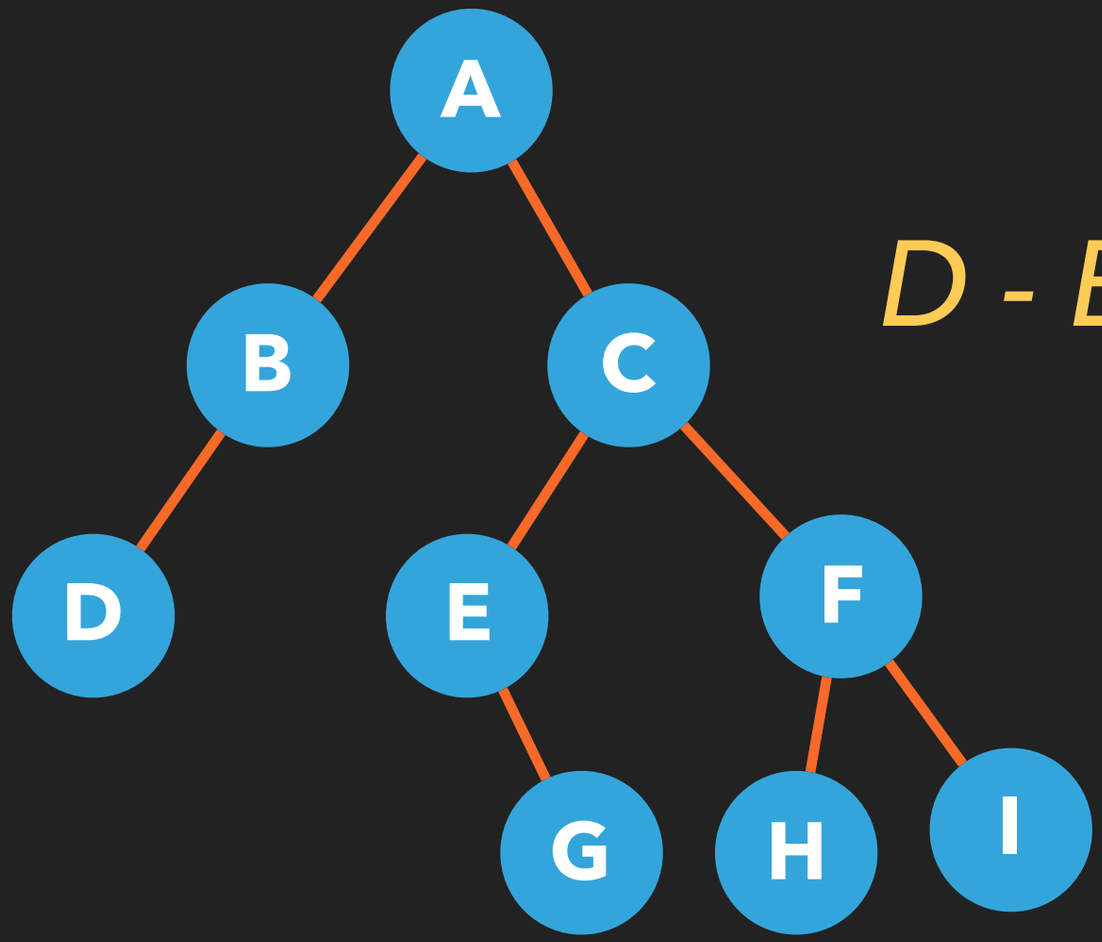
A - B - D - C - E - G - F - H - I

Inorder Traversal

- Any node is visited after its left subtree and before its right subtree
- L-V-R: Go Left - Visit Node - Go Right
  - Traverse the left subtree
  - Visit the node
  - Traverse the right subtree

In order Traversal

- L-V-R: Go Left - Visit Node - Go Right



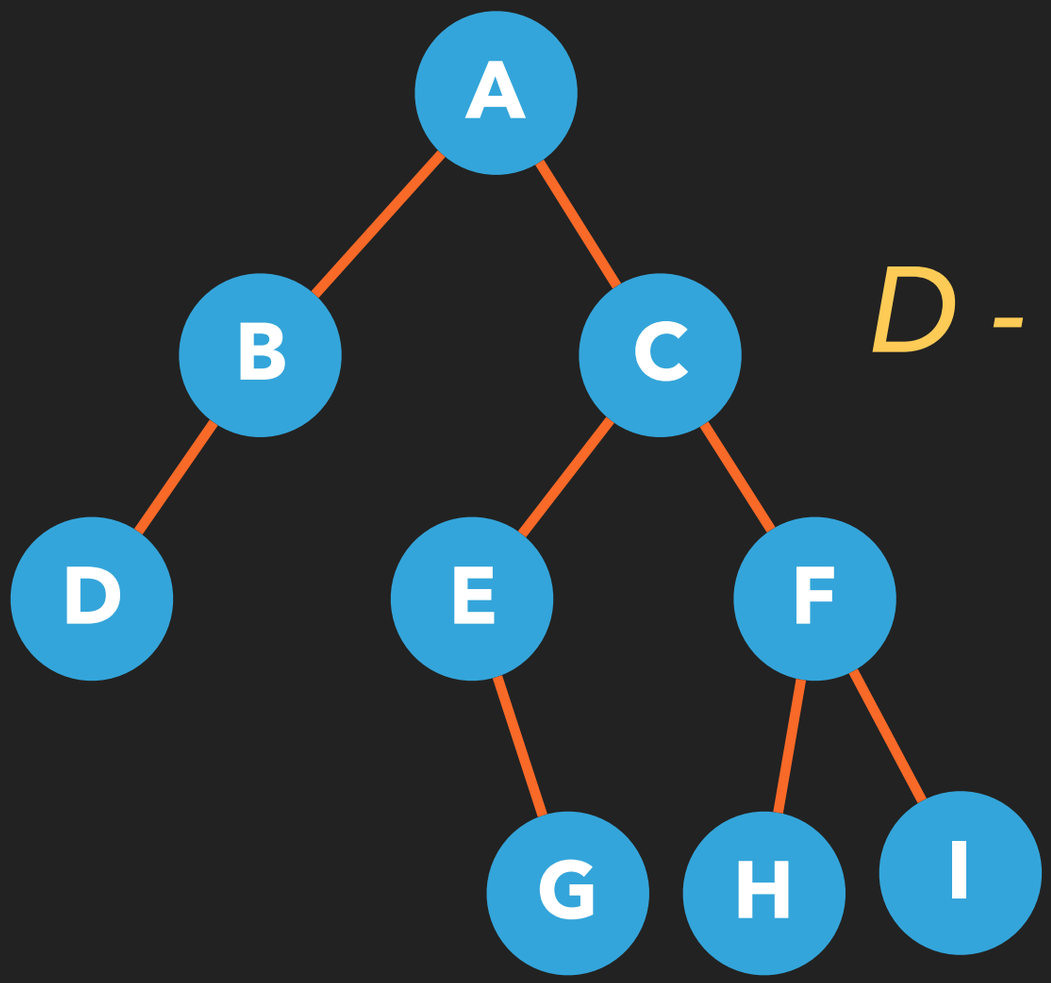
D - B - A - E - G - C - H - F - I

Postorder Traversal

- Any node is visited after its left subtree and right subtree
- L-R-V: Go Left - Go Right - Visit Node
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the node

Postorder Traversal

- L-R-V: Go Left - Go Right - Visit Node



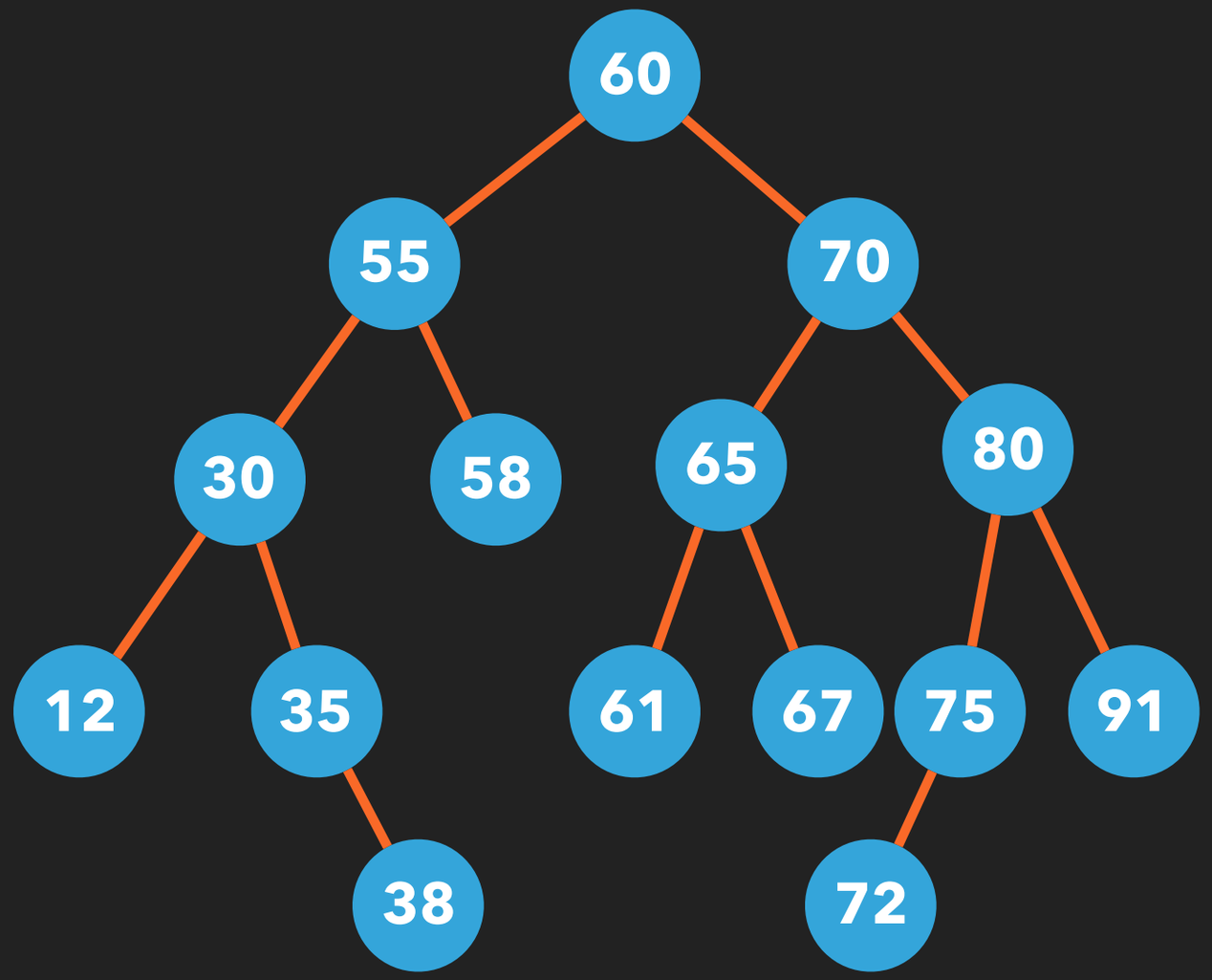
D - B - G - E - H - I - F - C - A

Practice

Preorder: ?

Inorder: ?

Postorder: ?



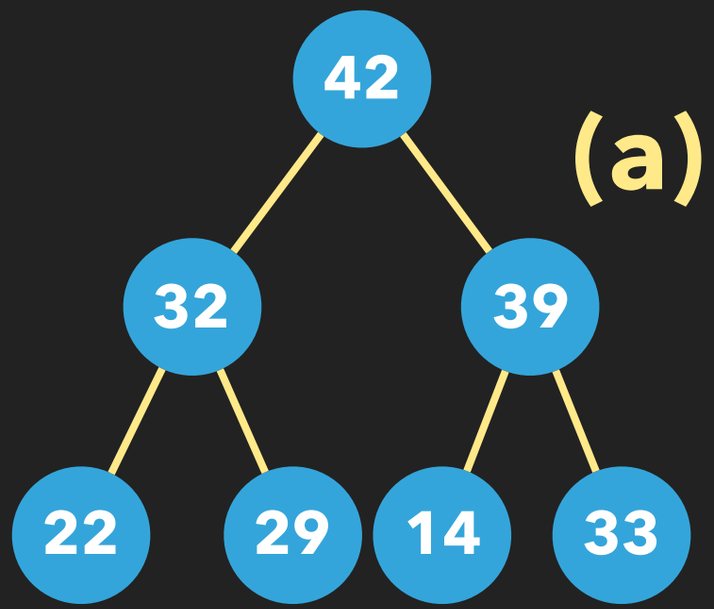
Heap

- Special binary tree used to order data (not to retrieve data)
- Used for efficient sorting (heap sort)
- The priority queue is implemented as a heap

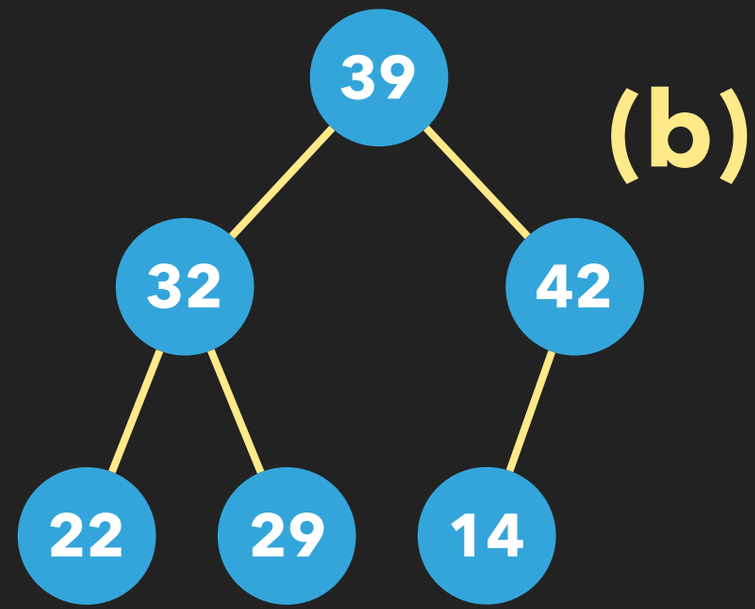
Heap

- Properties of the heap
  - Property 1: Complete binary tree (All the levels are filled except the last level All leaves on the last level are placed leftmost)
  - Property 2: every node is greater than or equal to any of its children (Max Heap) or less than or equal to any of its children (Min Heap)

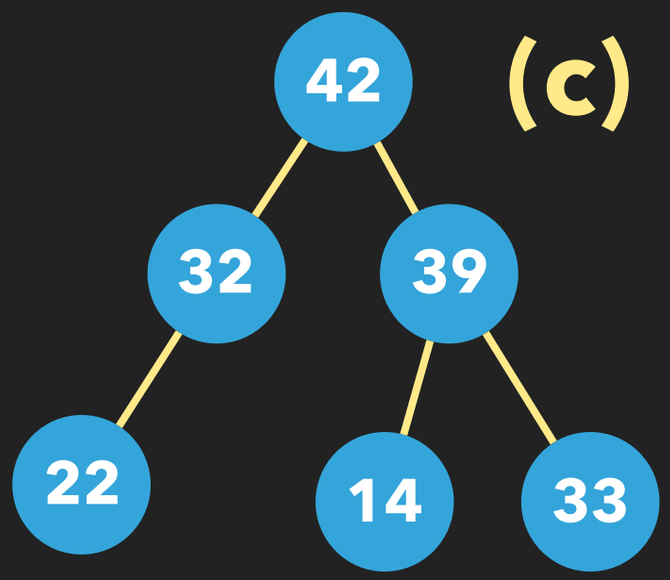
Heap



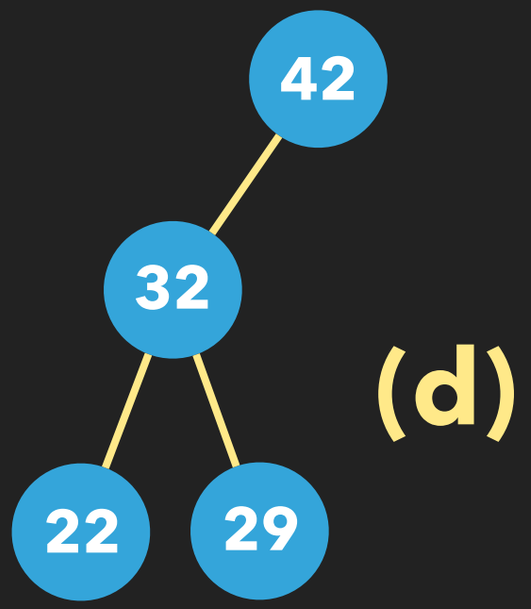
(a)



(b)



(c)



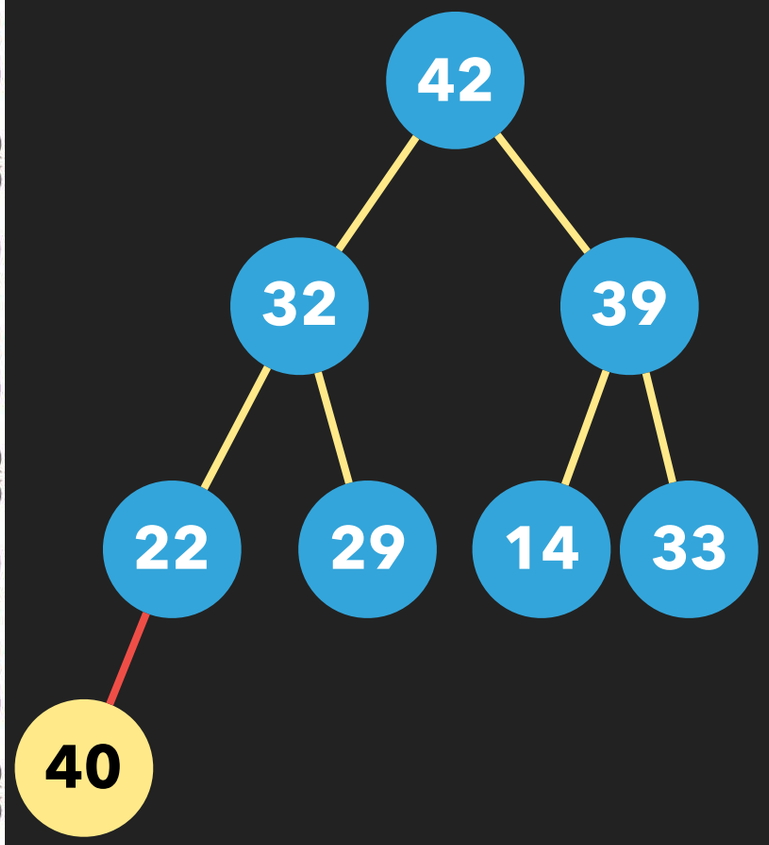
(d)

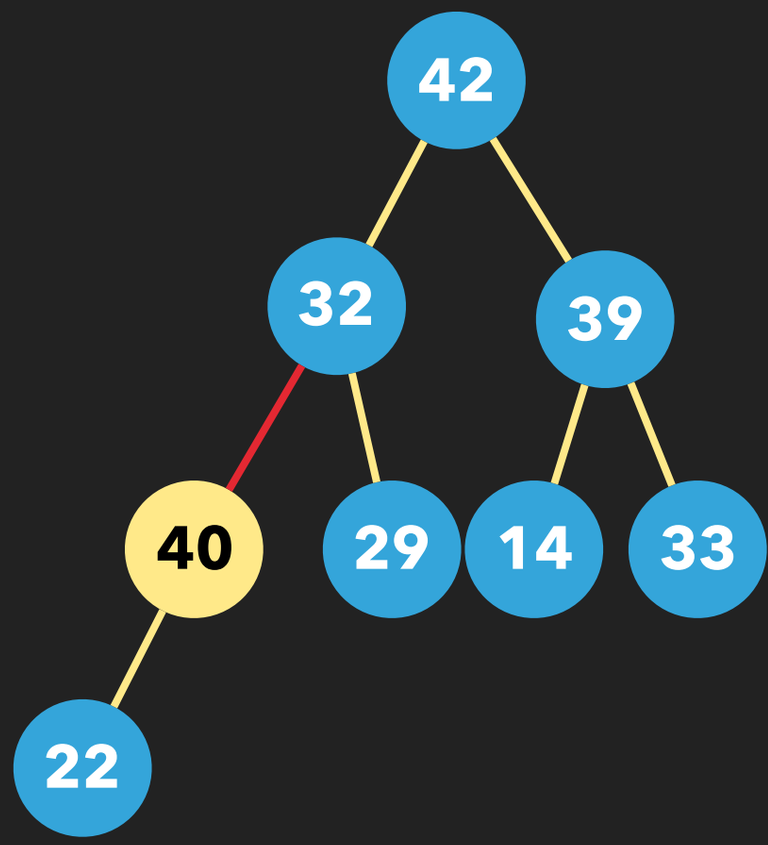
Heap

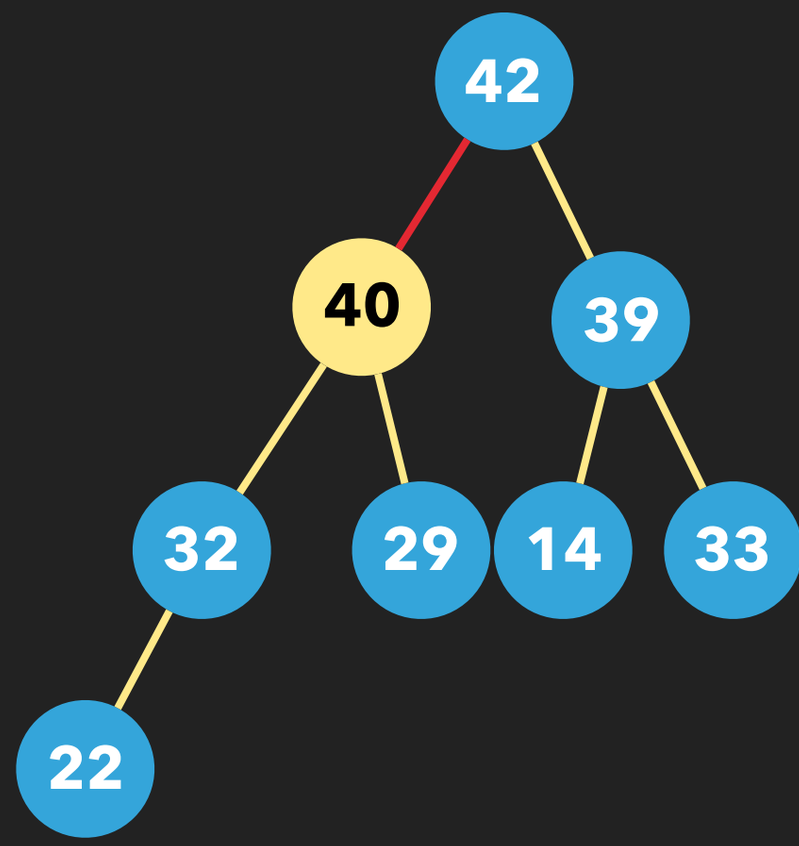
- Two main operations on the Heap
  - Adding a new node while keeping the heap properties
  - Removing a node while keeping the heap properties

Heap (add operation)

- Adding a new node to the heap (40)







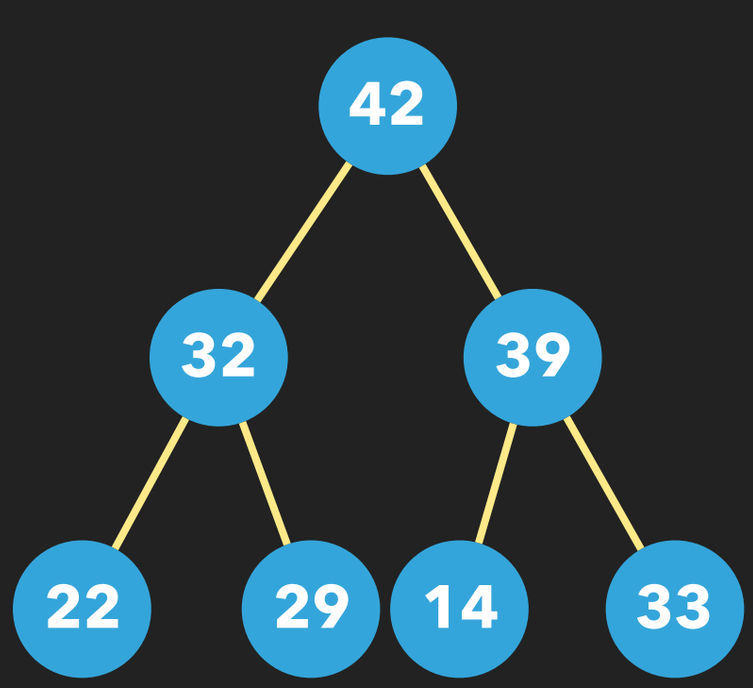
Heap (add)

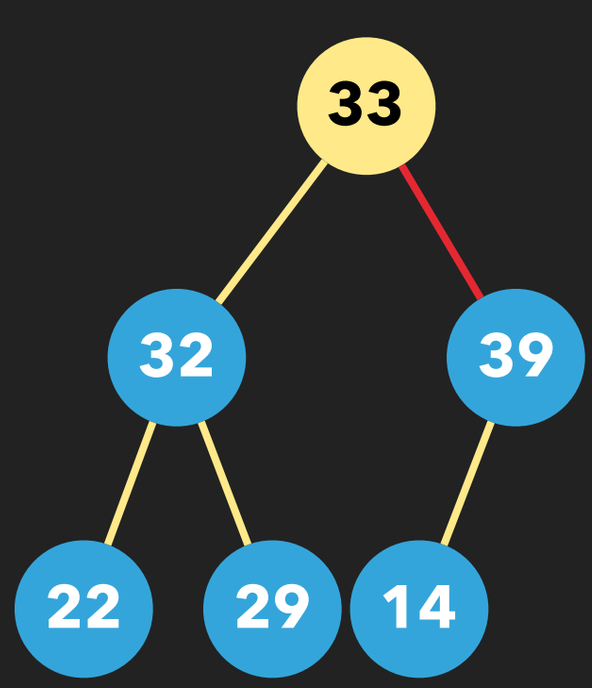
- Adding a new node to the heap

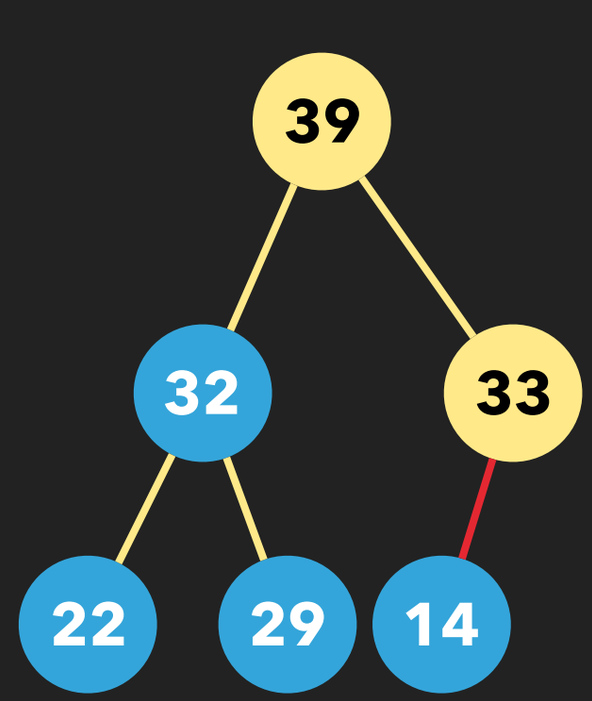
```
Algorithm add

  Add the new node at the end of the heap

  Current node = added node

  While (current node > its parent)

    Swap current node with its parent

    Current node becomes the parent

End
```

Heap (remove)

- Removing a node from the heap (42)
- Always the root







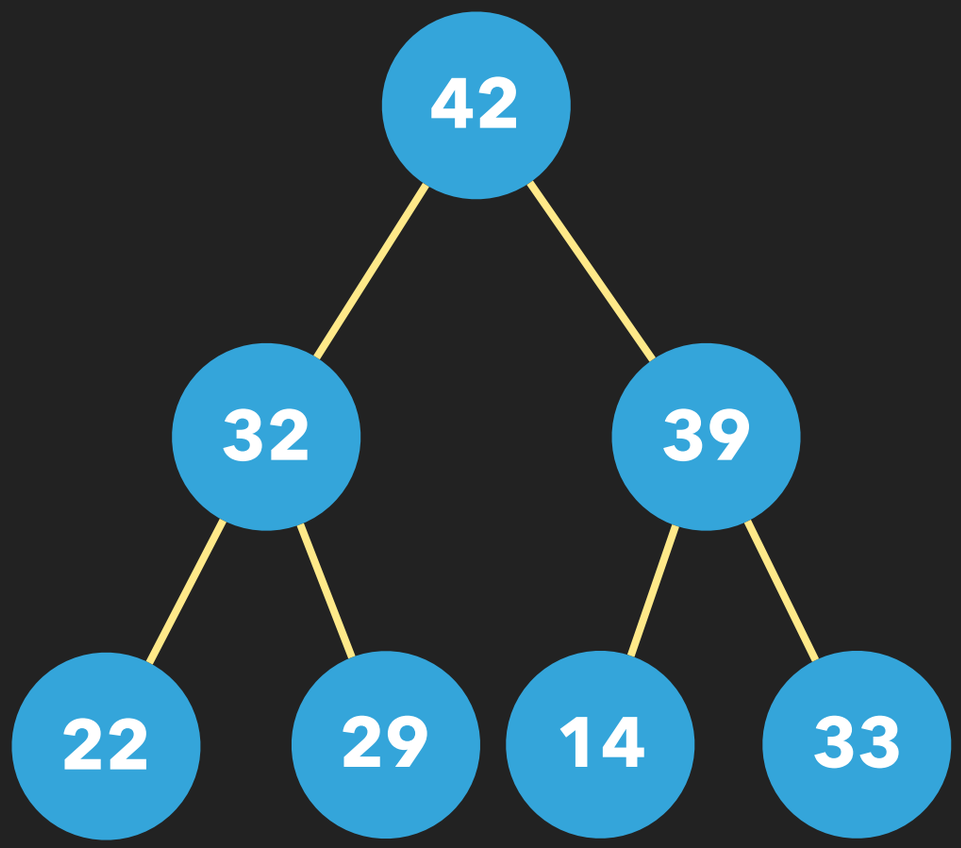
Heap (remove)

- Removing a node from the heap (root)

```
Algorithm remove

  Copy the value of the last node to the root

  Current node = root

  While (current node < its children)

    Swap current node with the largest
    of its children

    Current node becomes the largest child

End
```

Heap implementation

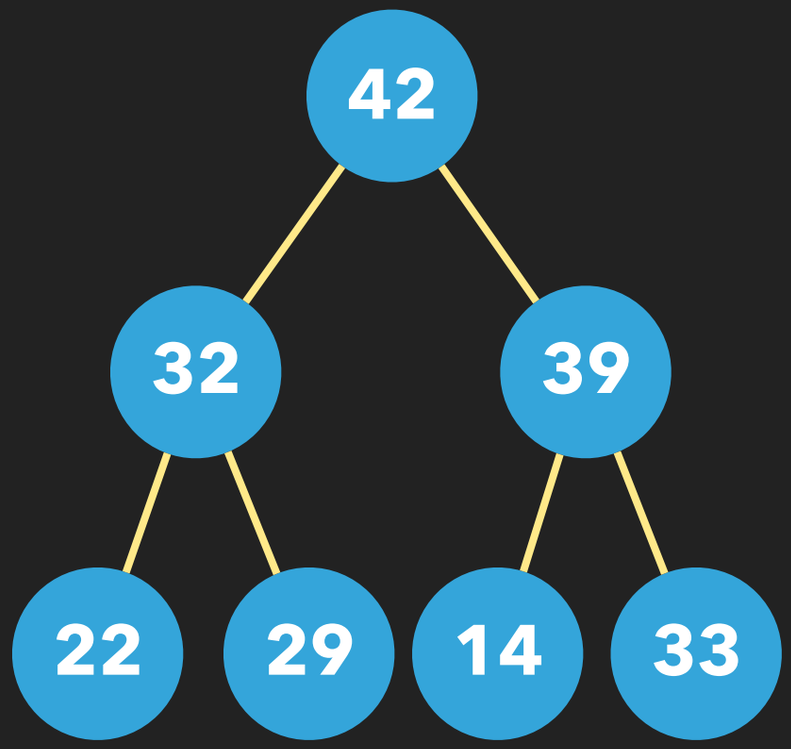
- The nodes of the heap are stored in an array list with easy access to the children and parent

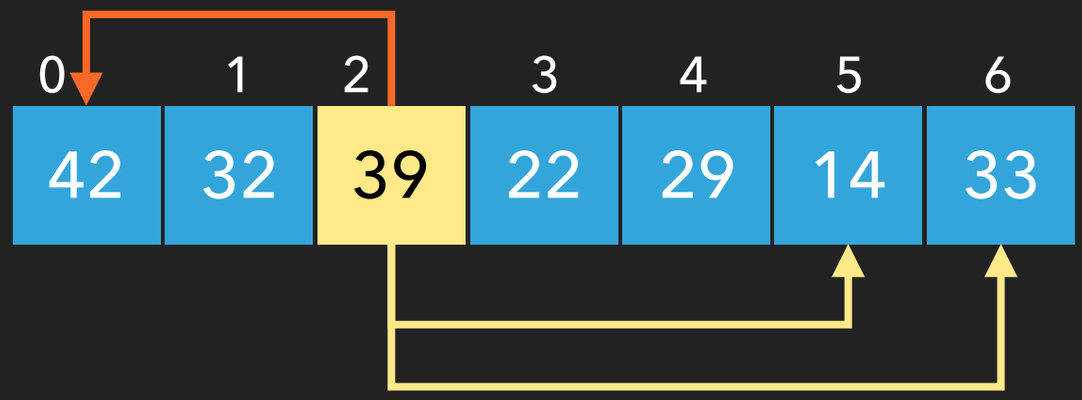


ArrayList

0	1	2	3	4	5	6
42	32	39	22	29	14	33

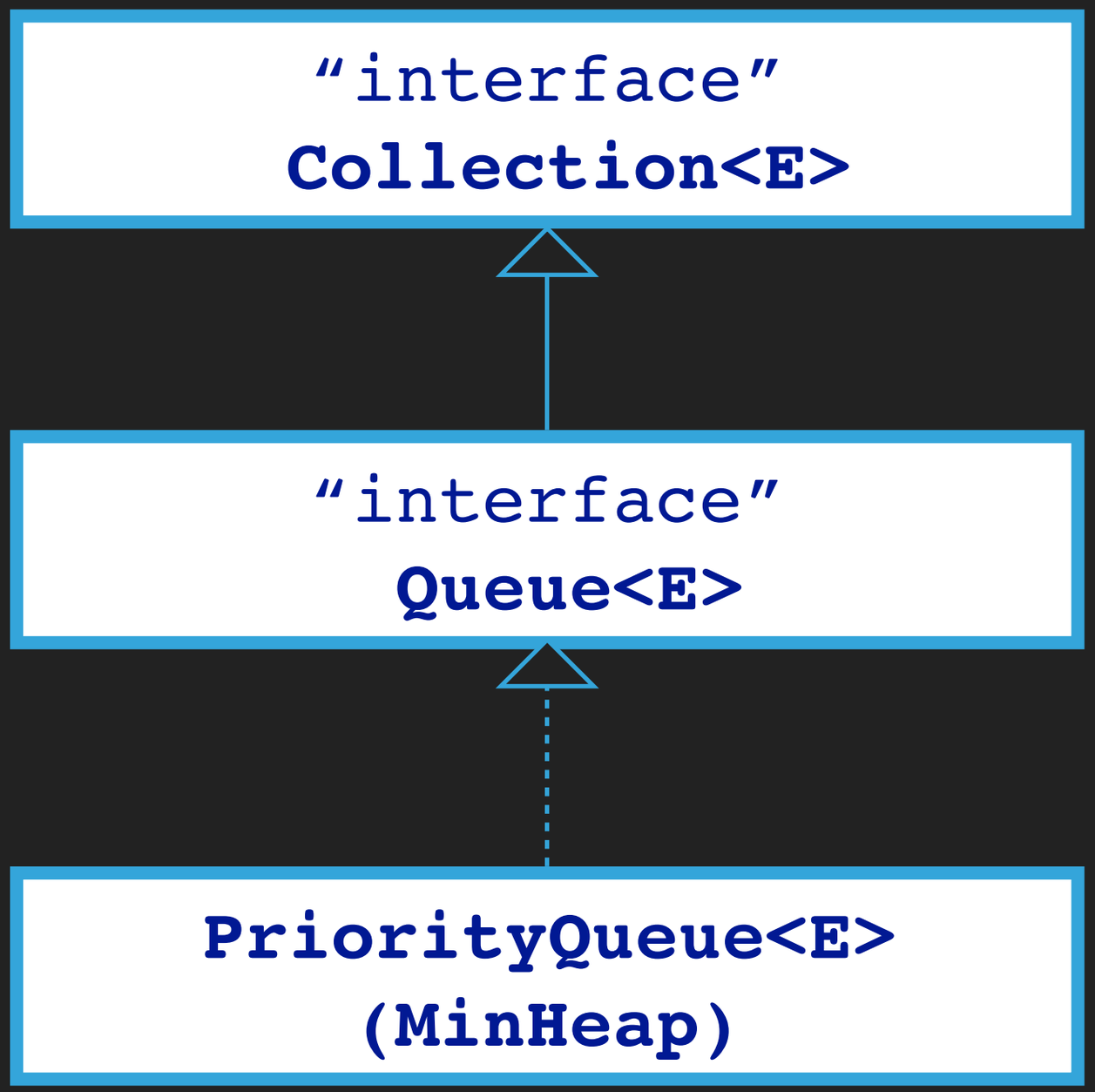
Heap implementation



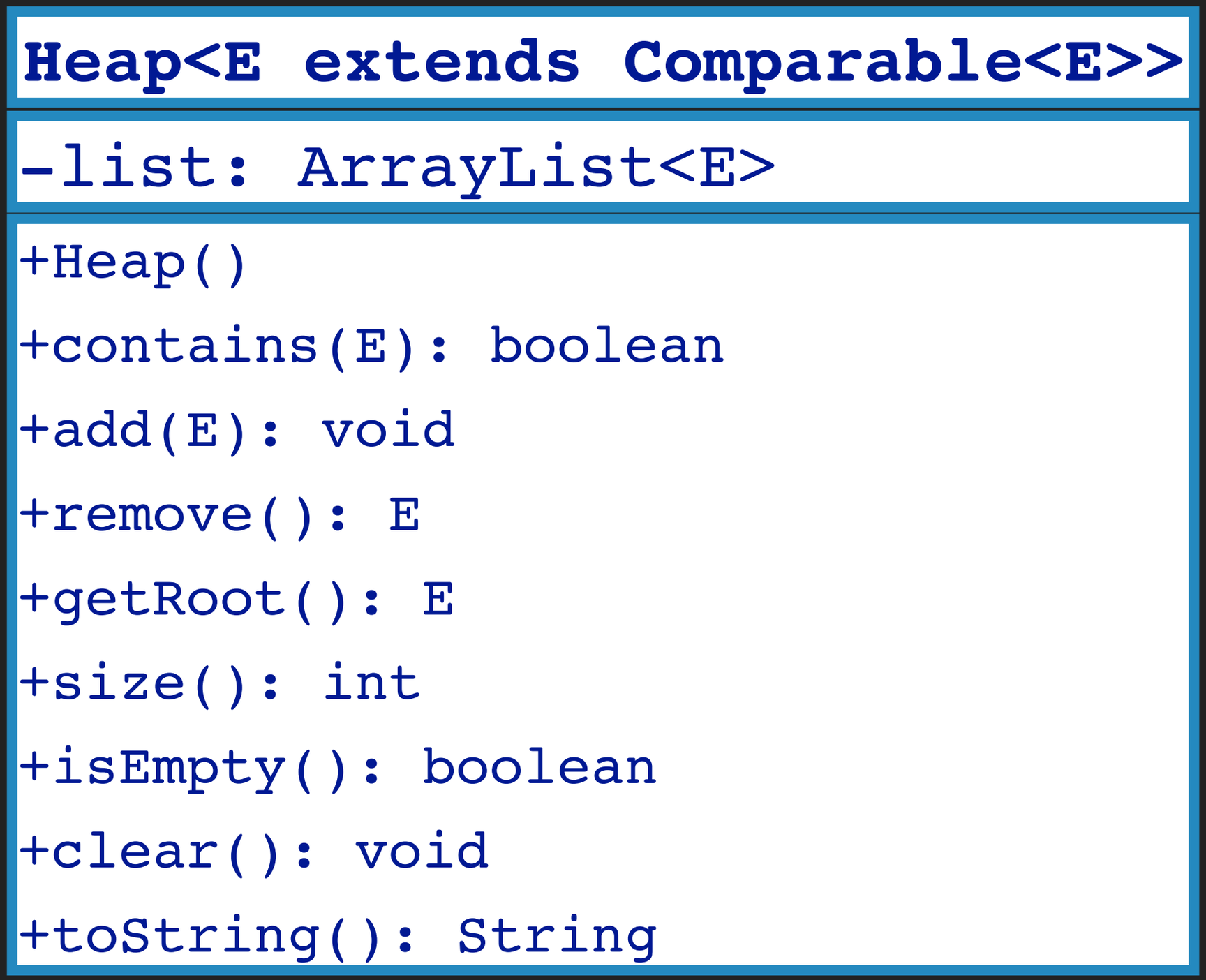



```
IndexOf(Parent) = (IndexOf(current) - 1) / 2

IndexOf(Left child) =  2 * IndexOf(current) + 1
IndexOf(Right child) = 2 * IndexOf(current) + 2
```

Heap implementation



Heap implementation



Heap implementation

Test.java

```
public class Test {
    public static void main(String[] args) {
        Heap<String> heap = new Heap<>();
        heap.add("Kiwi");
        heap.add("Strawberry");
        heap.add("Apple");
        heap.add("Banana");
        heap.add("Orange");
        heap.add("Lemon");
        heap.add("Watermelon");
        System.out.println("\nHeap: " + heap.toString());
        System.out.println("Root: " + heap.getRoot());
        System.out.println("Removed: " + heap.remove());
        System.out.println("Heap: " + heap.toString());
        System.out.println("Heap contains Pear?: " +
                           heap.contains("Pear"));
    }
}
```

Heap implementation

- Complexity of the Heap operations

Method	Complexity
`Heap()`	**O(1)**
`size()`	**O(1)**
`clear()`	**O(1)**
`isEmpty()`	**O(1)**
`getRoot()`	**O(1)**
`add(E)`	**O(log n)**
`remove(E)`	**O(log n)**
`contains(E)`	**O(n)**
`toString()`	**O(n)**

Summary

- Heap - Special binary tree
- Operations: Add and Remove mainly
- Implementation - Using an ArrayList
- Performance of the operations (logarithmic for add and remove)
- Heap is a balanced binary tree always (`height = log(number of nodes)`)