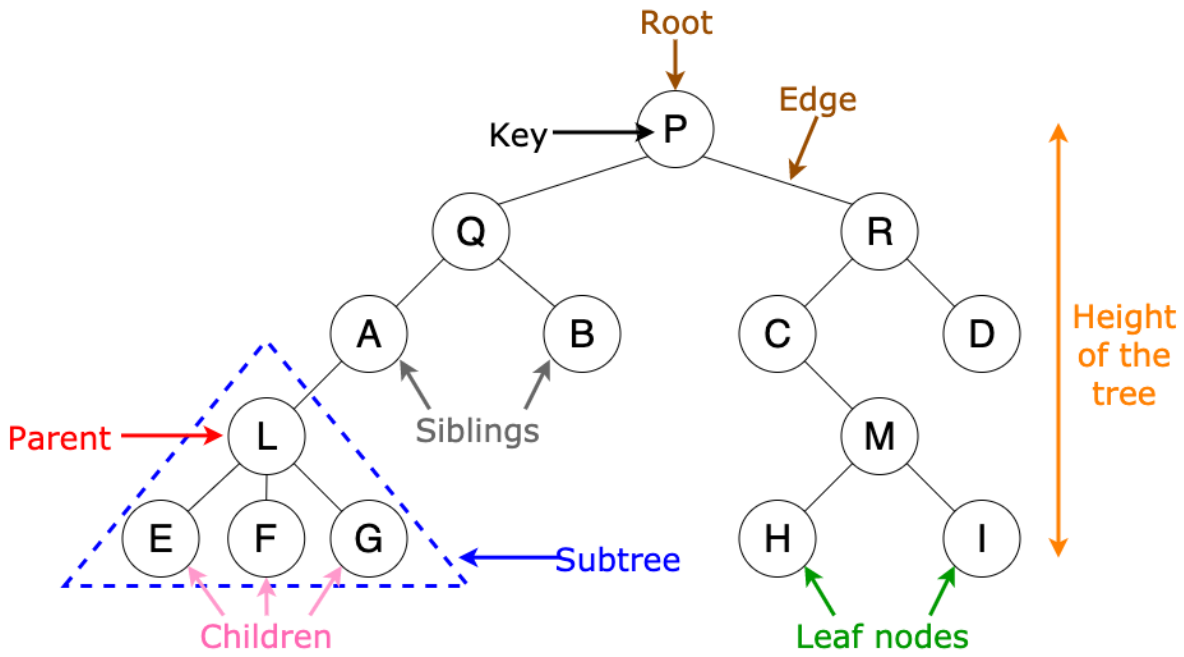
AI VIET NAM
@aivietnam.edu.vn

Data Structure

Tree and Its Applications



Data & Code

Vinh Dinh Nguyen
PhD in Computer Science

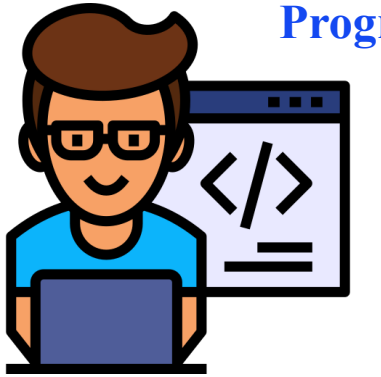
Outline

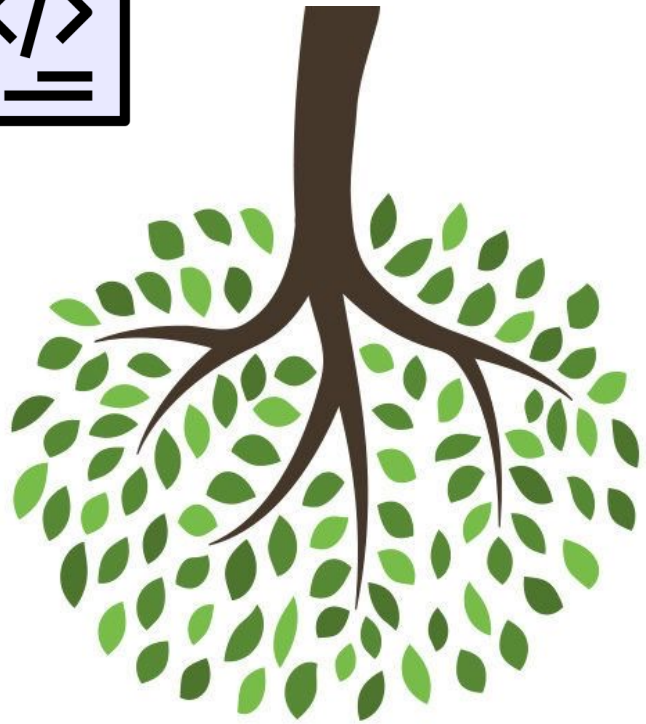
- What is a Tree Data Structure
- Algorithms on Trees
- Stack
- Queue
- What is a Binary Tree
- What is a Binary Search Tree
- Summary

What is a Tree?

Human view

Programmer View





Why Tree?

Other data structures such as arrays, linked list are linear data structures that store data sequentially

Check if 8 exists in the list. How?

5	4	7	1	3	6	8

Need a New Data Structure that Efficient Searching

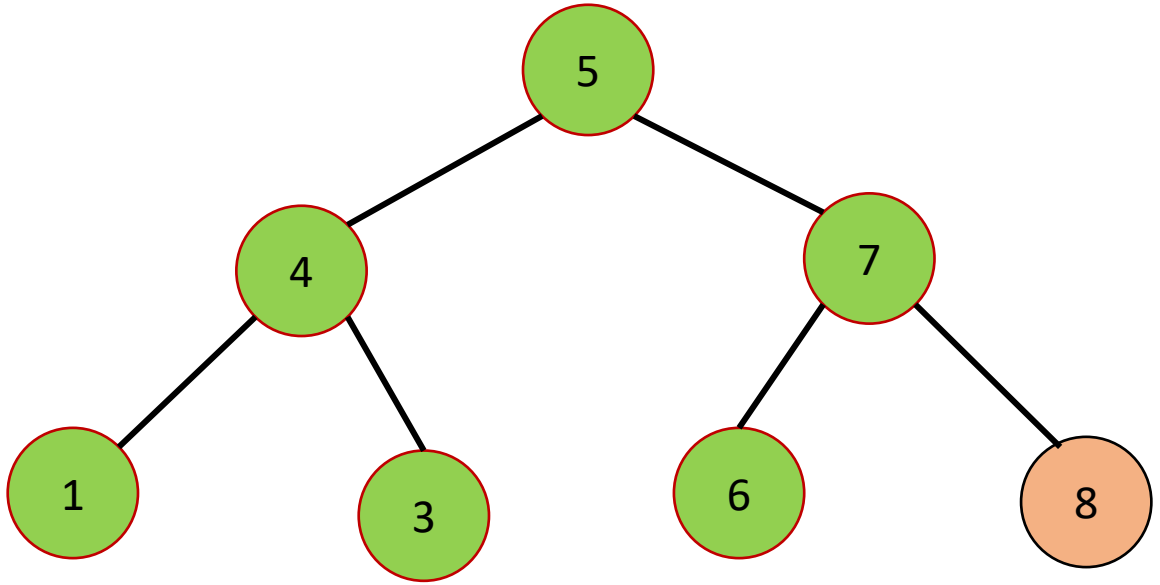
Why Tree?

List

5	4	7	1	3	6	8

Check if 8 exists. How?

Tree

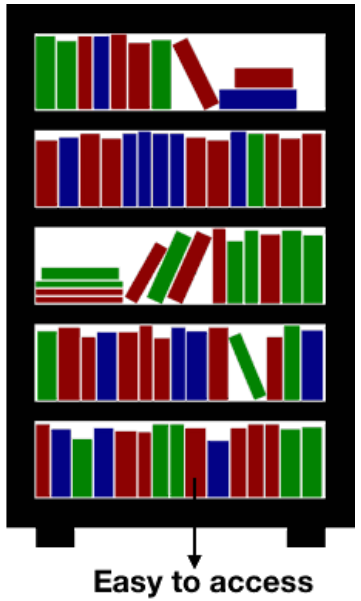


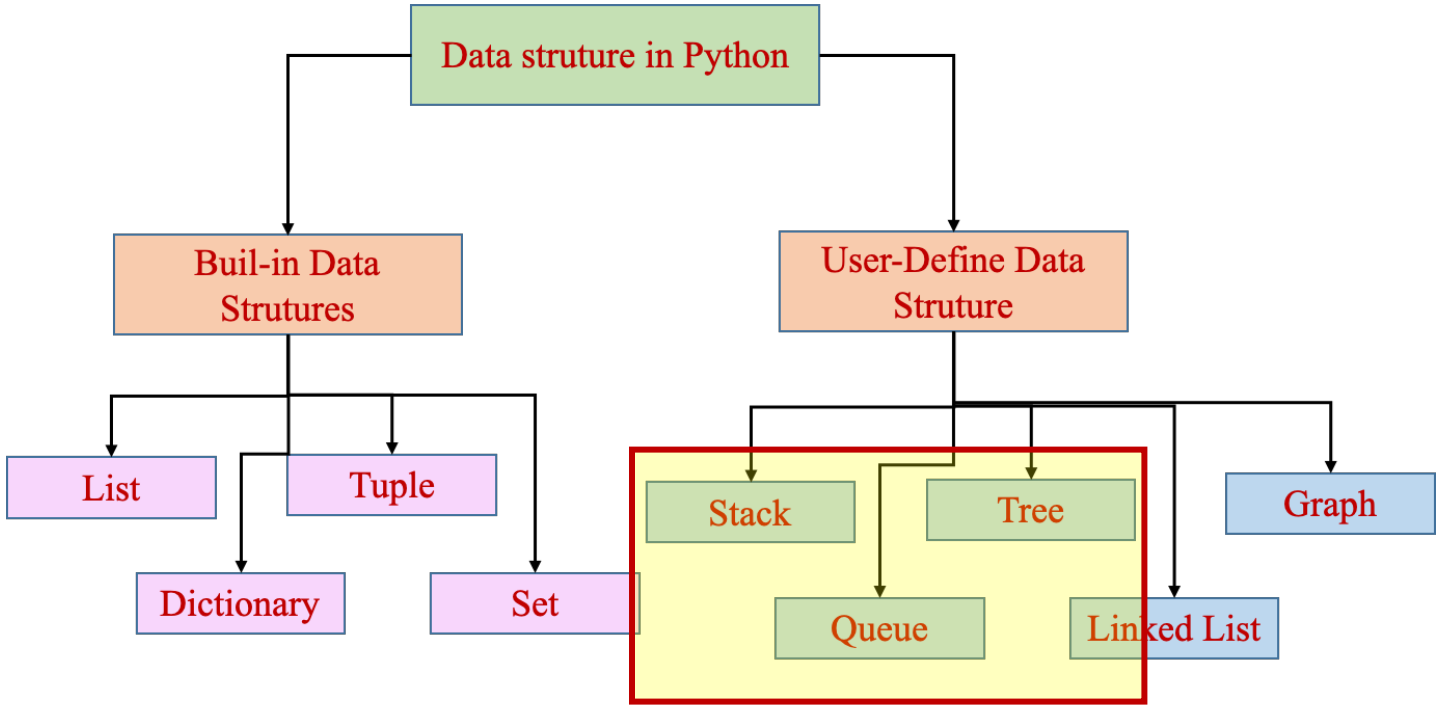
Left Value > Right Value

AI VIET NAM
@aivietnam.edu.vn

Data Structure?

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

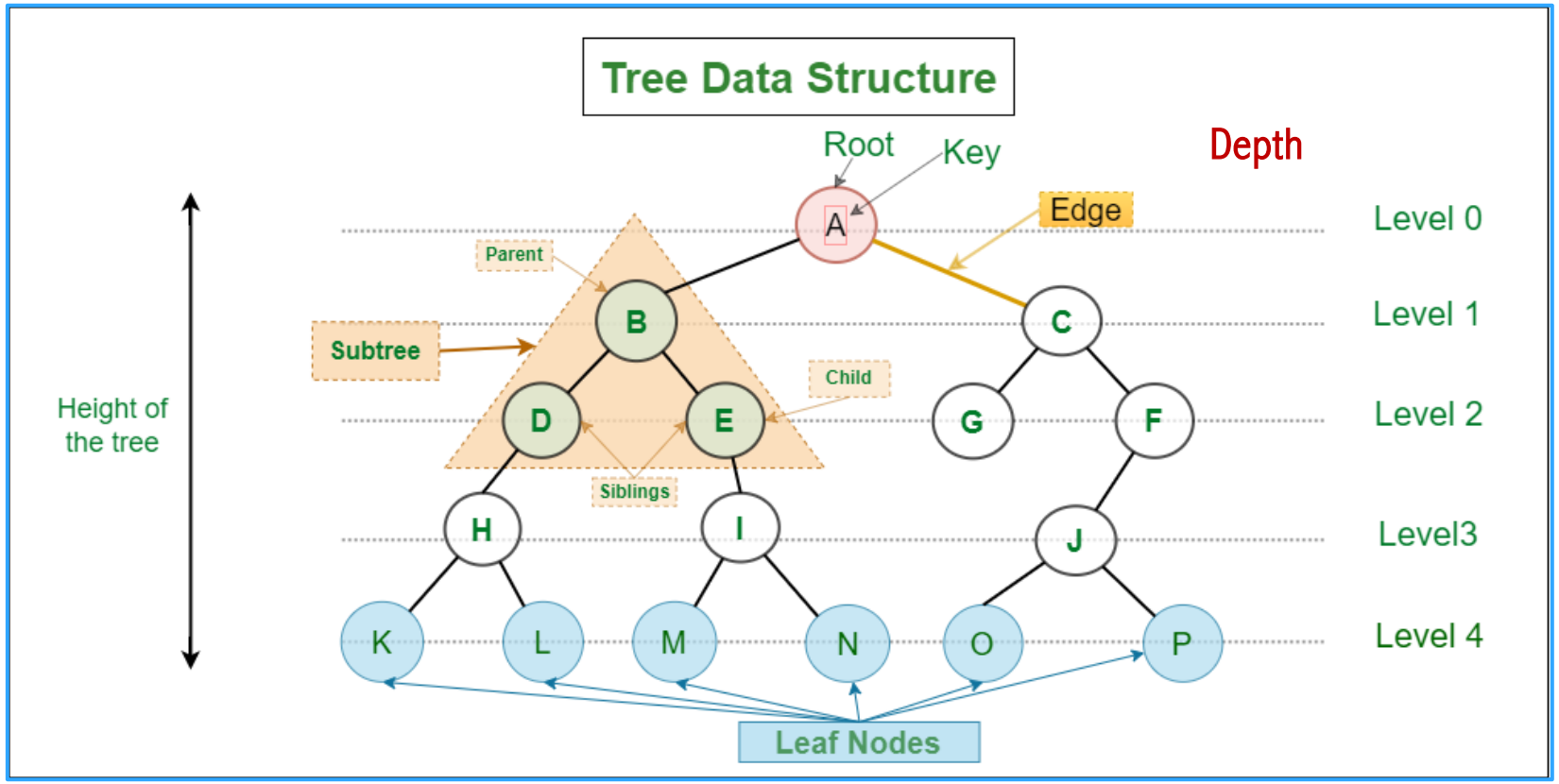




AI VIET NAM
@aivietnam.edu.vn

What is a Tree?

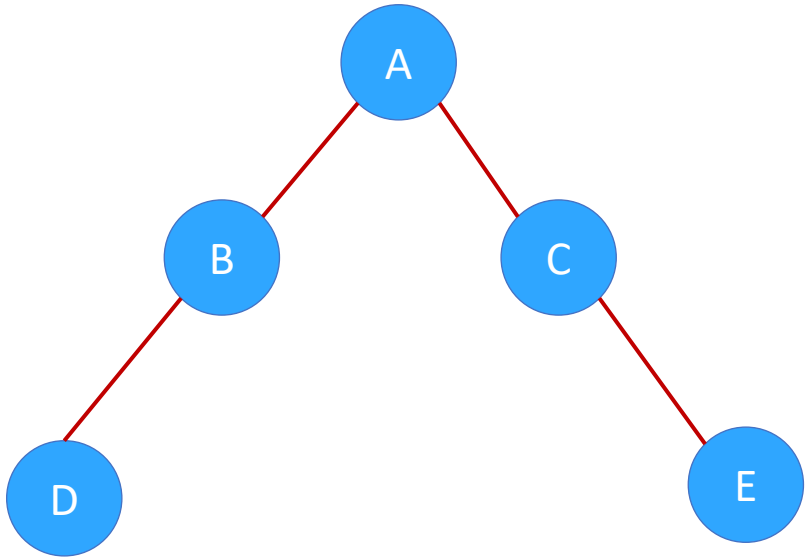
A non-linear data structure where nodes are organized in a hierarchy



How to store **Tree** data in Python?

In the above figure, where do you think the root of the tree is?

What is a Tree?



Which data structure is using ?

```
Tree = {
        'A': ['B', 'C'],
        'B': ['D'],
        'C': ['E'],
        'D': [],
        'E': []
}
```

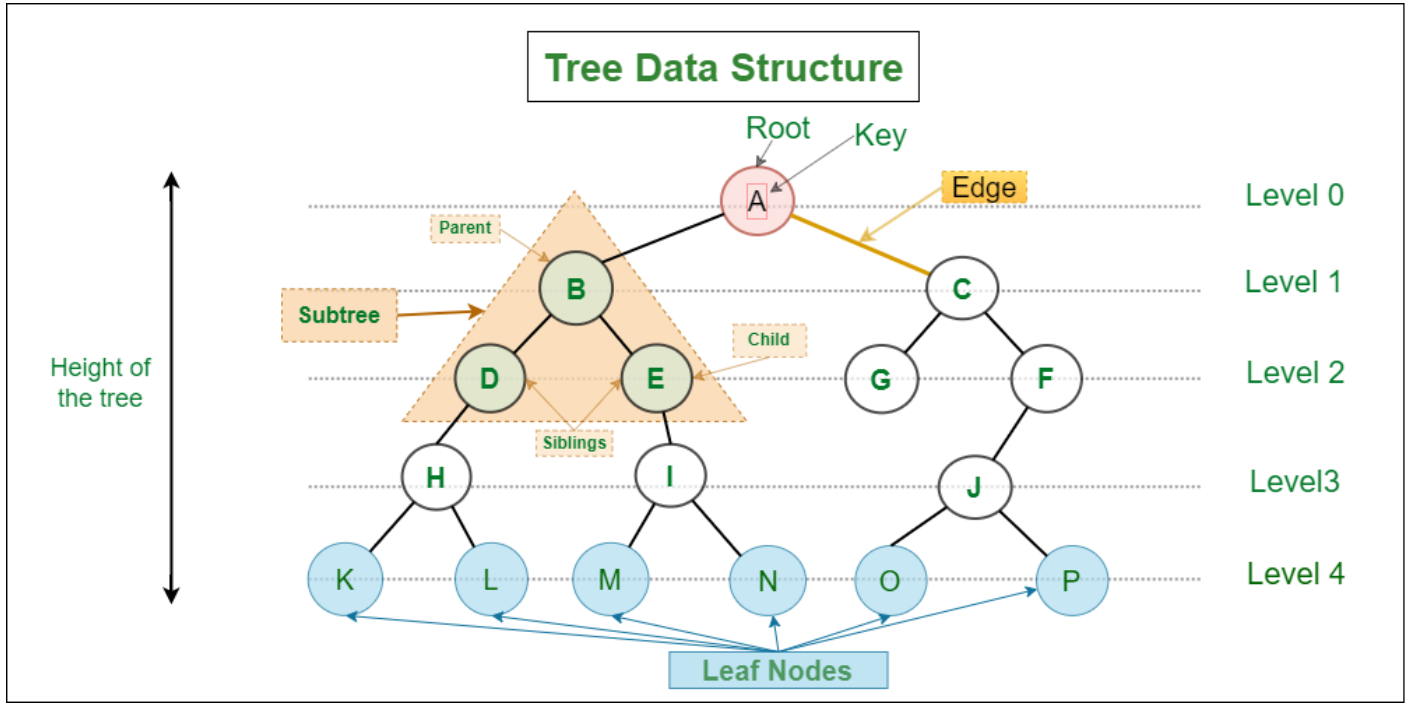
What do you think about the current implementation?

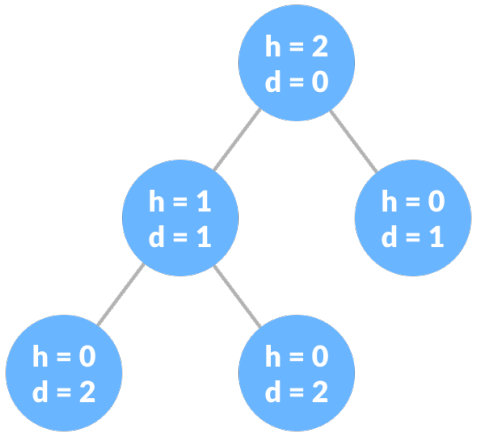
does it satisfy OOP priciples?

How to implement in Python?

AI VIET NAM
@aivietnam.edu.vn

Basic Terminologies





Height: h
Depth: d

Parent Node: {B} is the parent node of **{D, E}**.

Child Node: {D, E} are the child nodes of **{B}.**

Root Node: {A} is the root node of the tree

Leaf Node: {K, L, M, N, O, P, G} are the leaf nodes of the tree

Ancestor of a Node: {A,B} are the ancestor nodes of the node **{E}**

Sibling: {D,E} are called siblings.

Level of a node(depth): The count of edges on the path from the root node to that node.

Subtree: Any node of the tree along with its descendant.

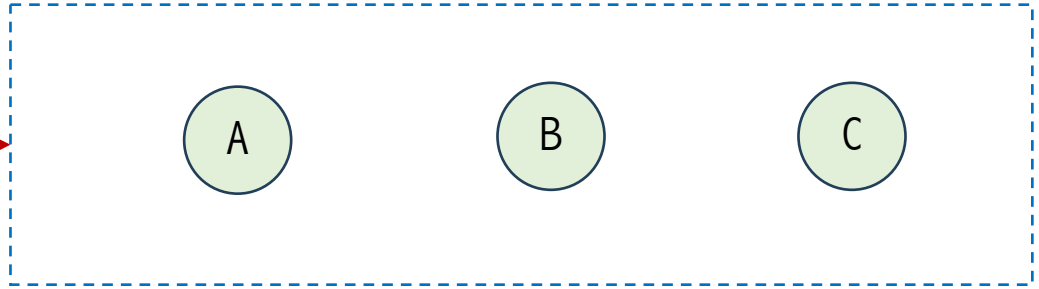
Height: Indicates the longest path from a node to a leaf.

Tree Implementation

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None
```

Node of a Tree

```
a_node = Node("A")
b_node = Node("B")
c_node = Node("C")
```



Create nodes, A, B, and C

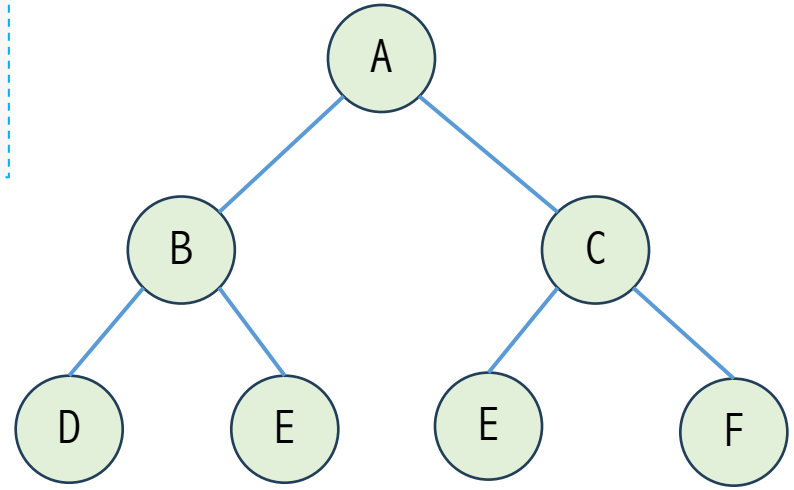
Tree Implementation

```python
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None

    def add_child(self, child):
        child.parent = self
        self.children.append(child)

    def get_level(self): #to get level of each node
        level =0
        p = self.parent
        while p:
            level += 1
            p = p.parent

        return level
    def print_tree(self):
        space = ' ' * self.get_level() * 3
        prefix = space + '|__' if self.parent else ''
        print(prefix + self.data)#add prefix
        if self.children:
            for child in self.children:
                child.print_tree()
```

```python
def create_tree():
  a_node = TreeNode("A")
  b_node = TreeNode("B")
  c_node = TreeNode("C")
  d_node = TreeNode("D")
  e_node = TreeNode("E")
  f_node = TreeNode("F")
  g_node = TreeNode("G")

  a_node.add_child(b_node)
  a_node.add_child(c_node)

  b_node.add_child(d_node)
  b_node.add_child(e_node)

  c_node.add_child(f_node)
  c_node.add_child(g_node)
  return a_node
```

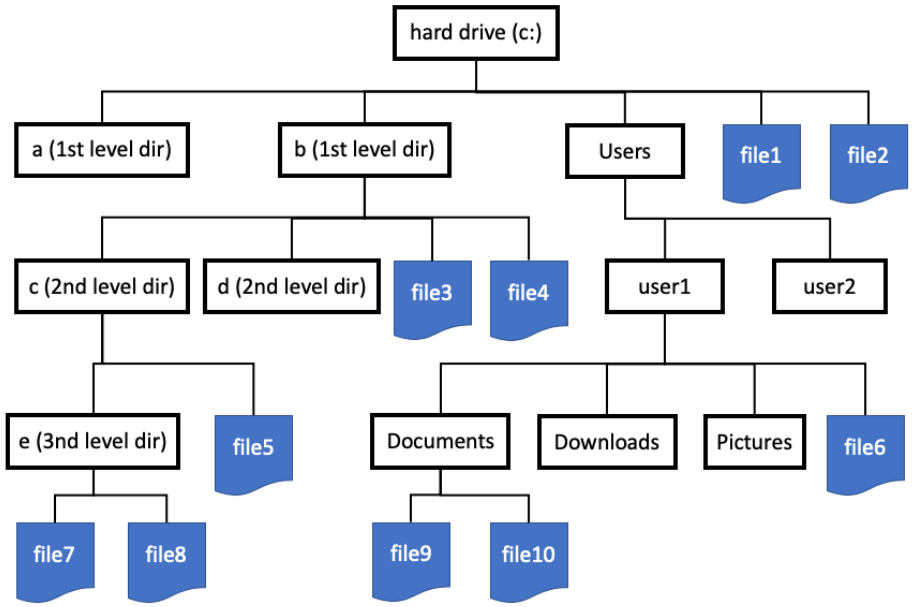
```python
tree = create_tree()
tree.print_tree()
```

```
A
   |__B
      |__D
      |__E
   |__C
      |__F
      |__G
```

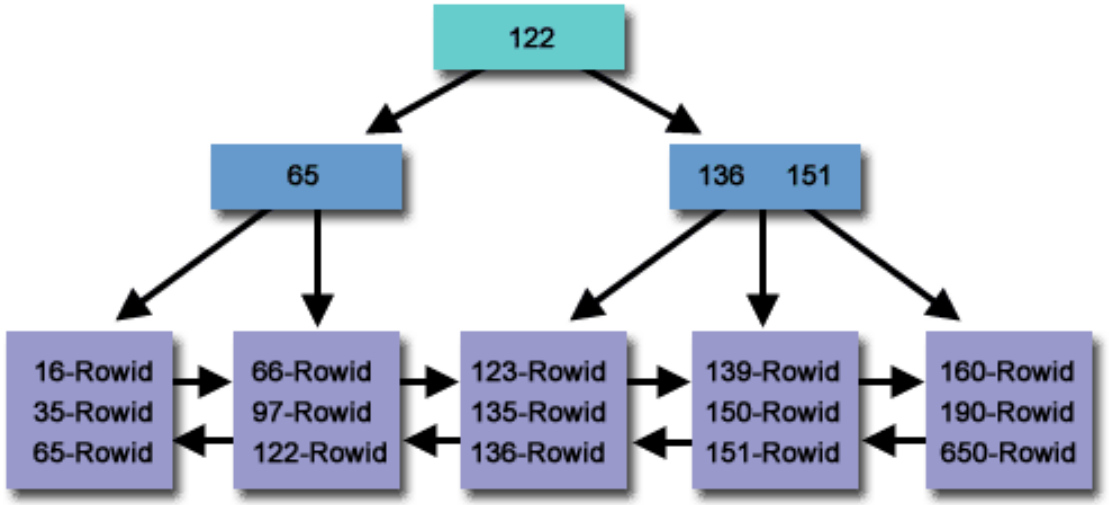


AI VIET NAM
@aivietnam.edu.vn

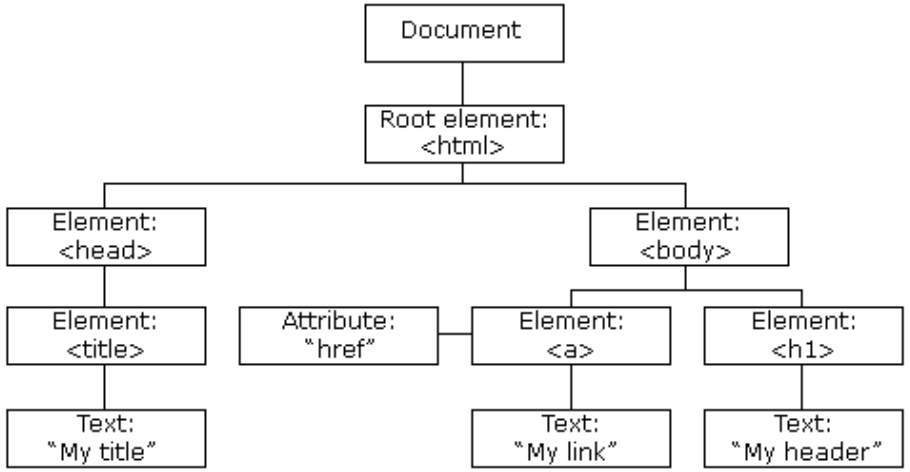
Tree Applications



File explorer



Database

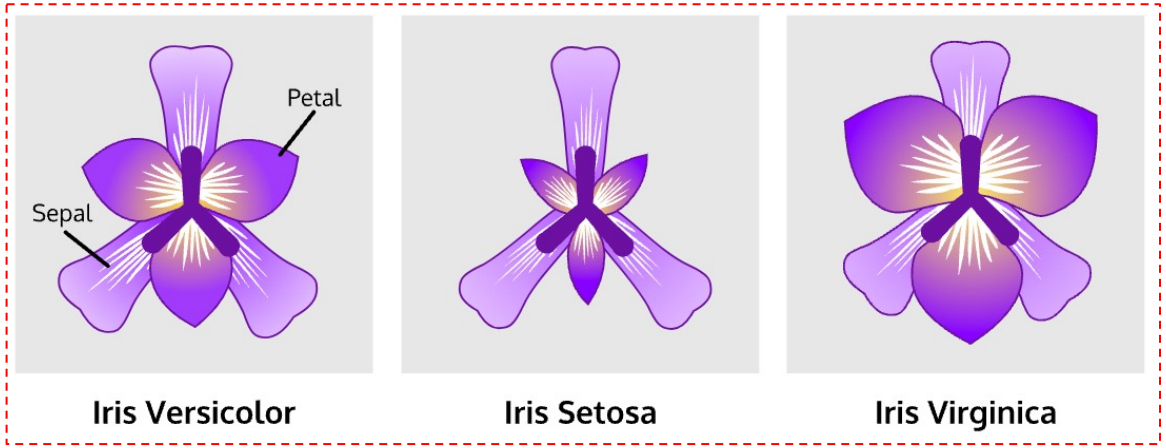


HTML DOM

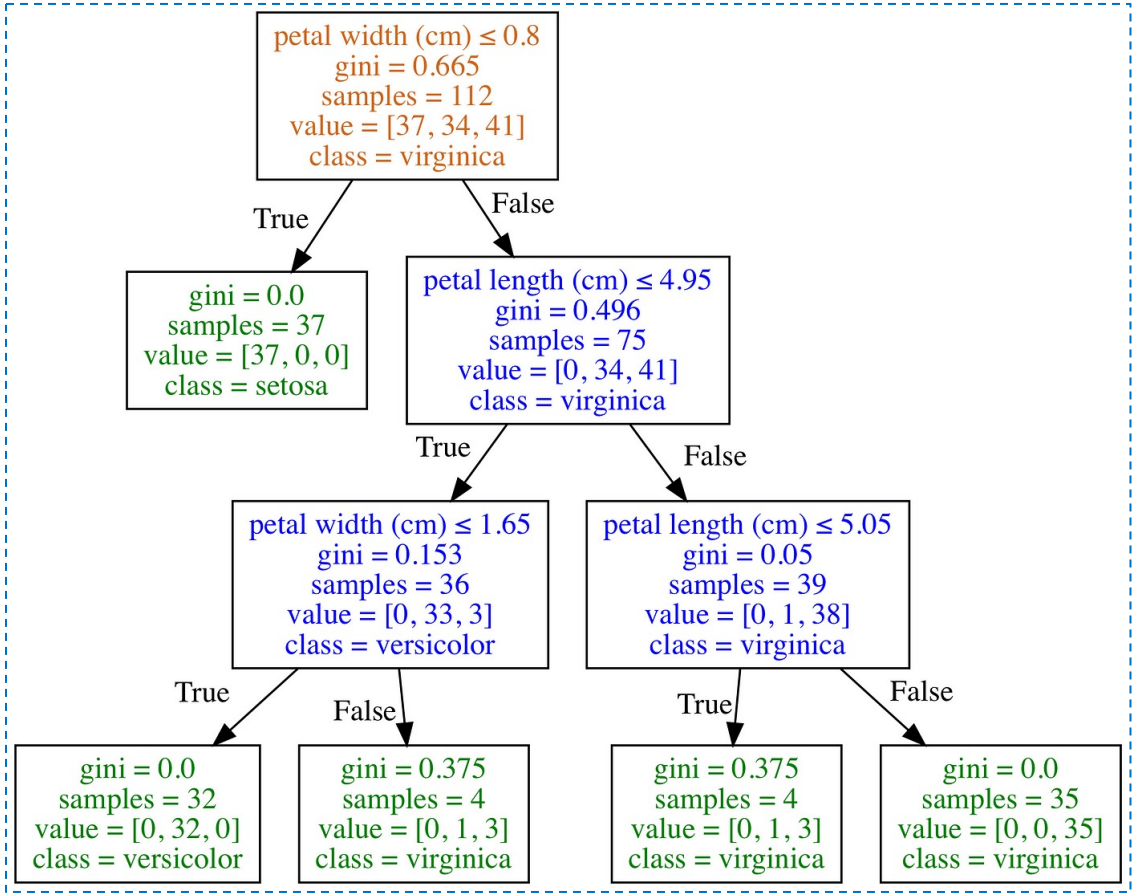
AI VIET NAM
@aivietnam.edu.vn

Tree Applications

ML & AI : Decision Tree



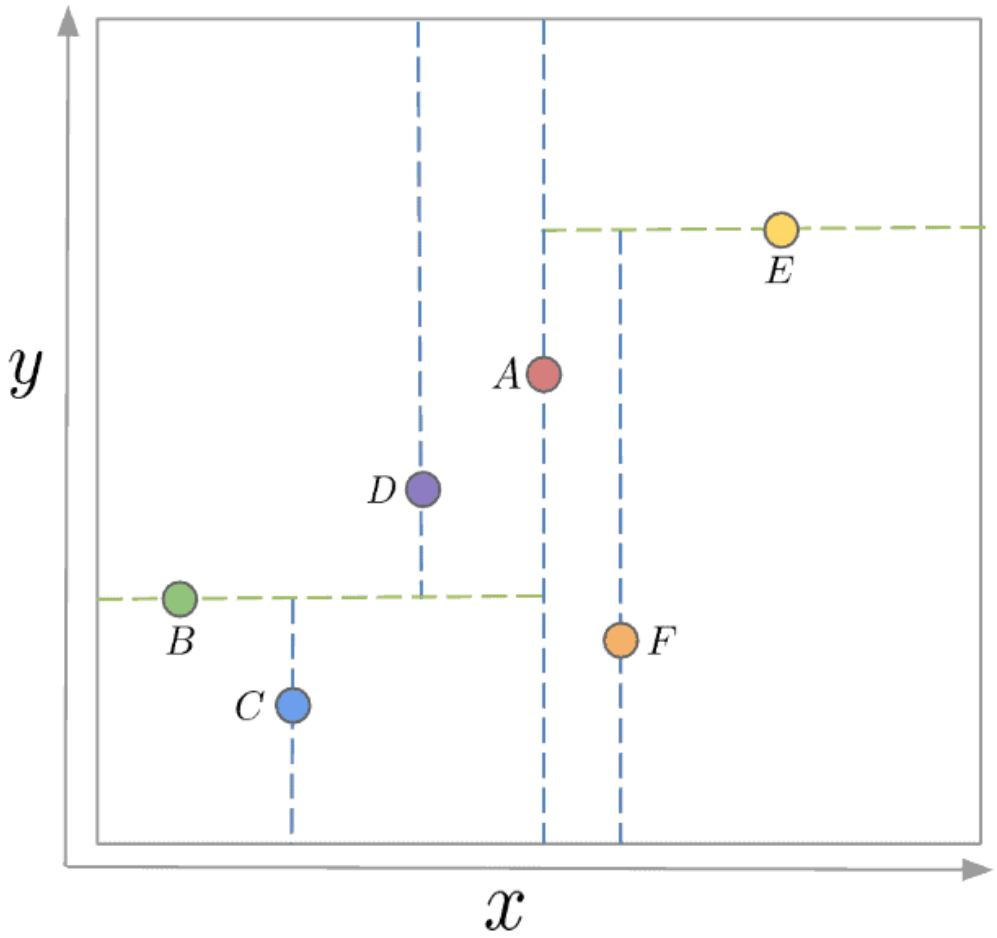
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

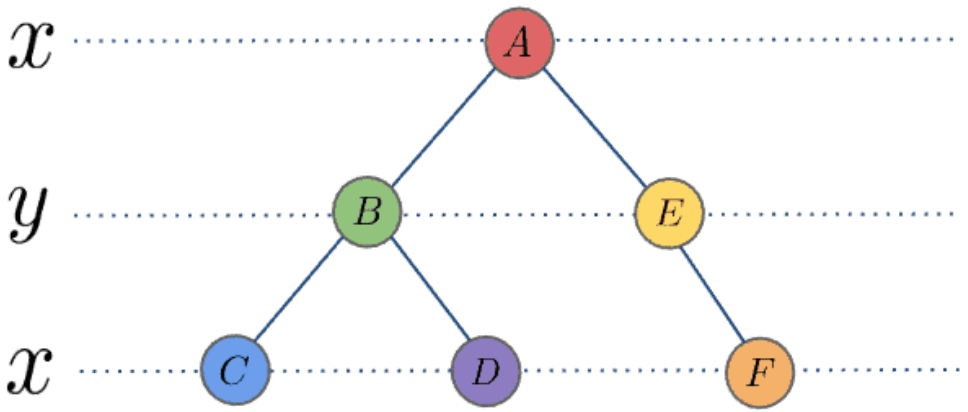


AI VIET NAM
@aivietnam.edu.vn

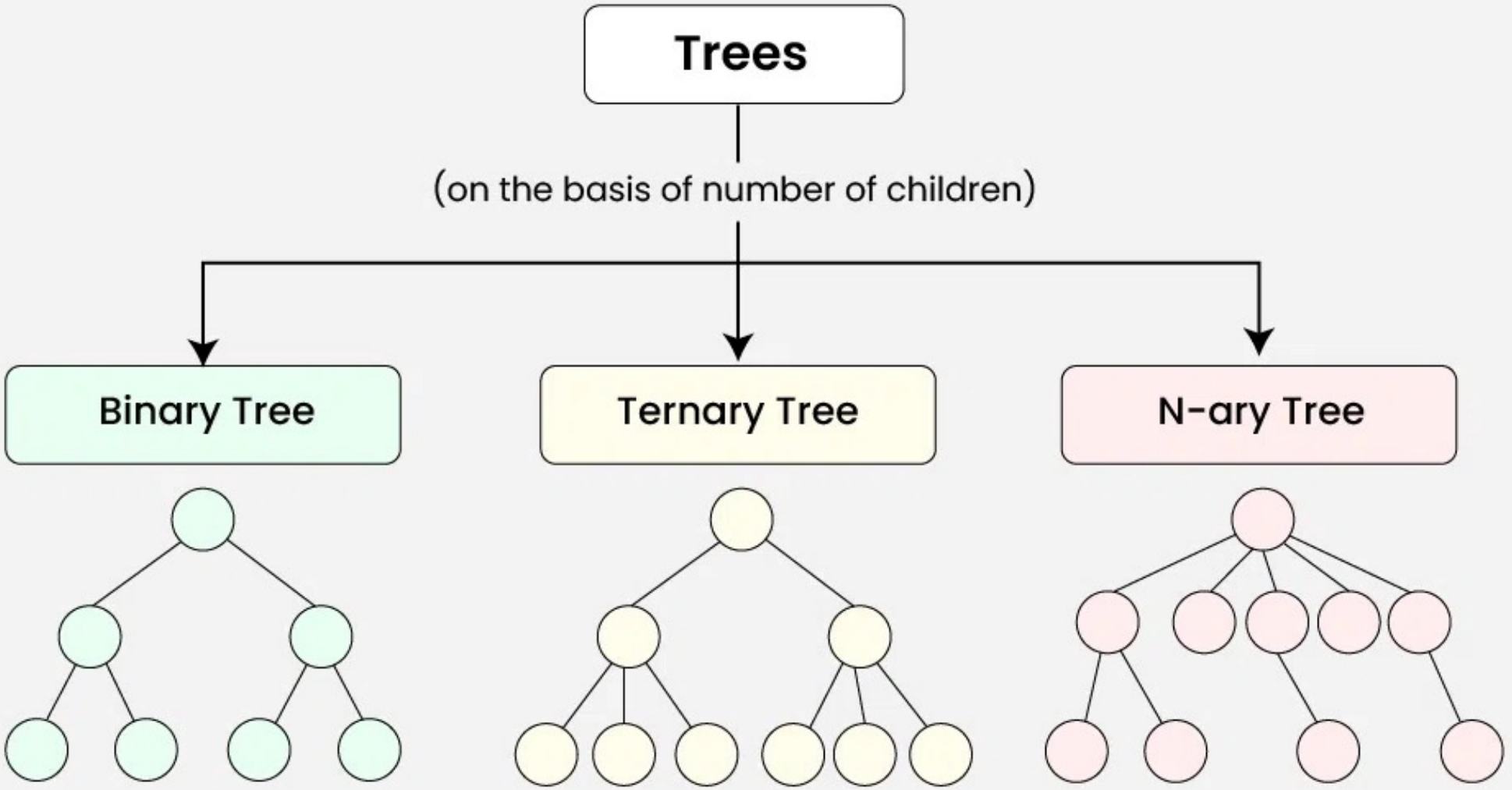
Tree Applications

ML & AI : K-D Tree (K-Dimensional Tree)





Types of Trees

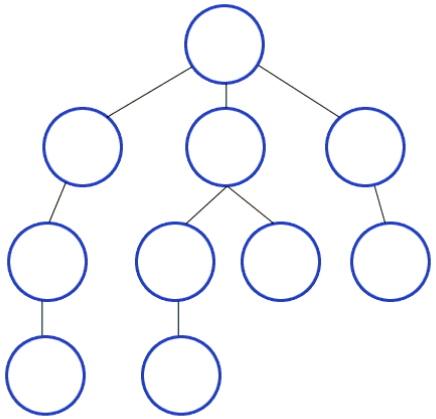


Outline

- What is a Tree Data Structure
- Algorithms on Trees
- Stack
- Queue
- What is a Binary Tree
- What is a Binary Search Tree
- Summary

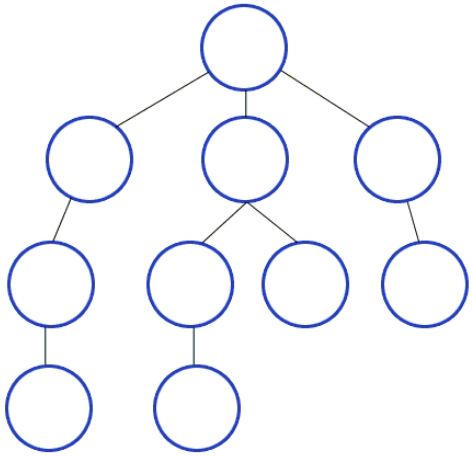
Traversal of Binary Tree

Traversal on a tree refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once, in a systematic way



Breadth First Search

Explores all the nodes in a tree at the current depth before moving on to the vertices at the next depth level.

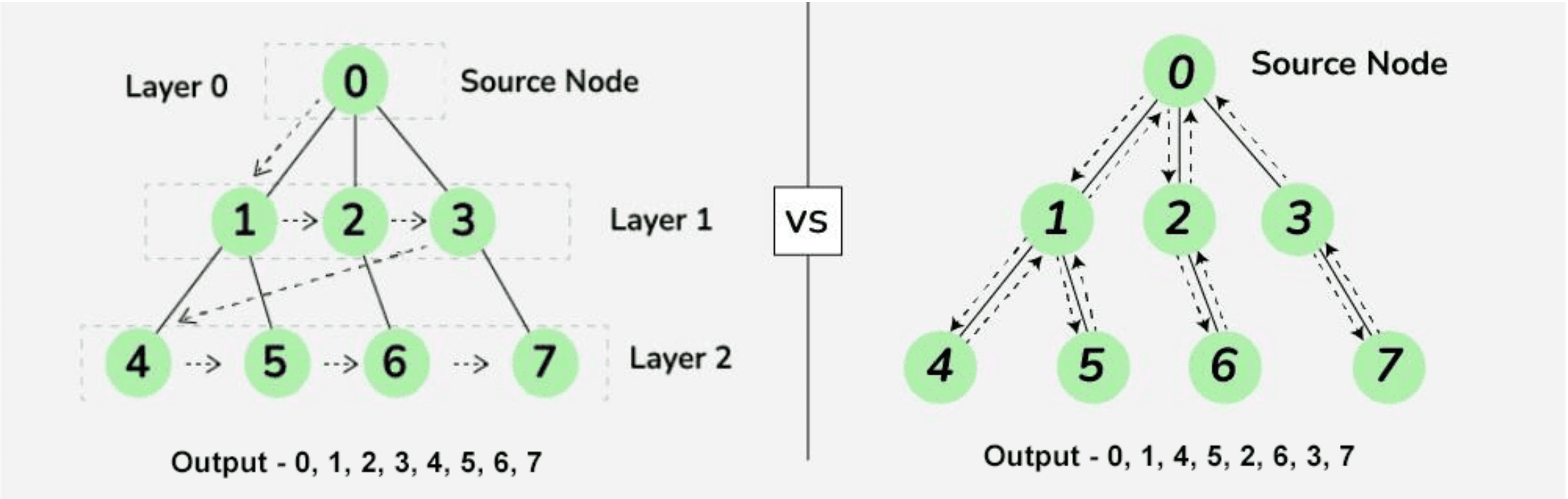


Depth First Search

The algorithm starts at the root node and explores as far as possible along each branch before backtracking

Traversal of Binary Tree

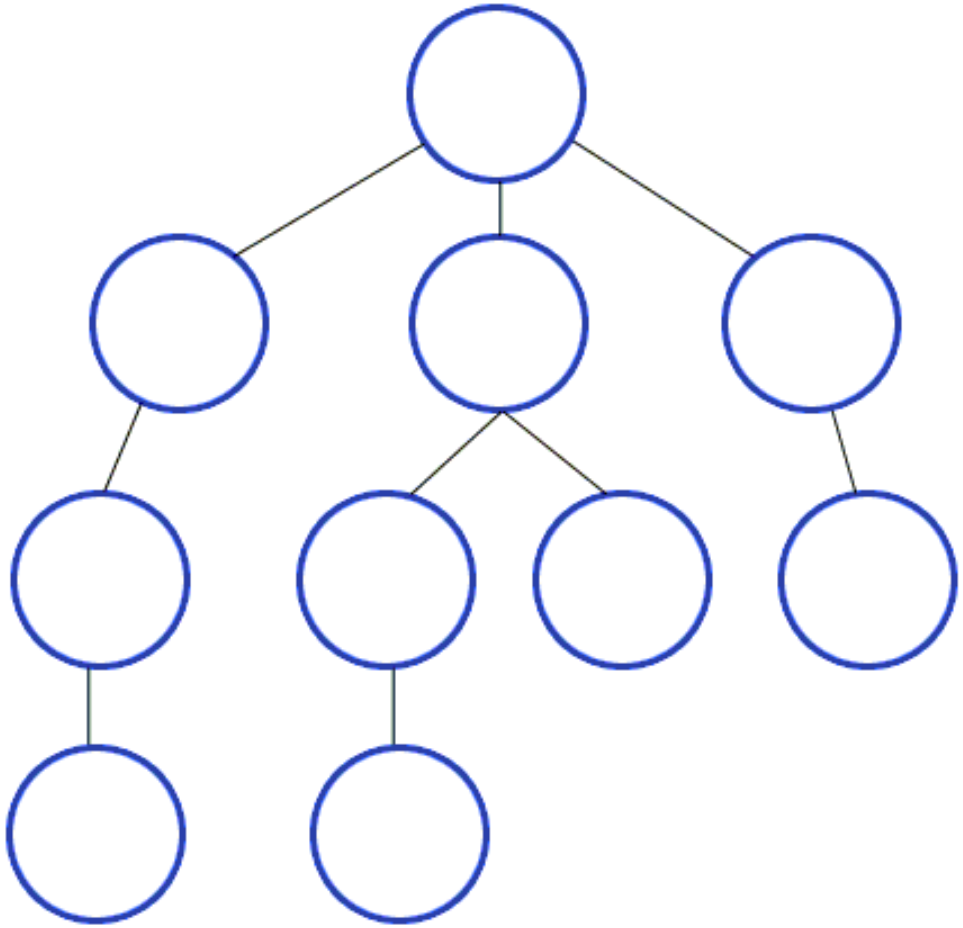
Traversal of Binary Tree involves visiting all the nodes of the binary tree. Tree Traversal algorithms can be classified broadly into two categories:



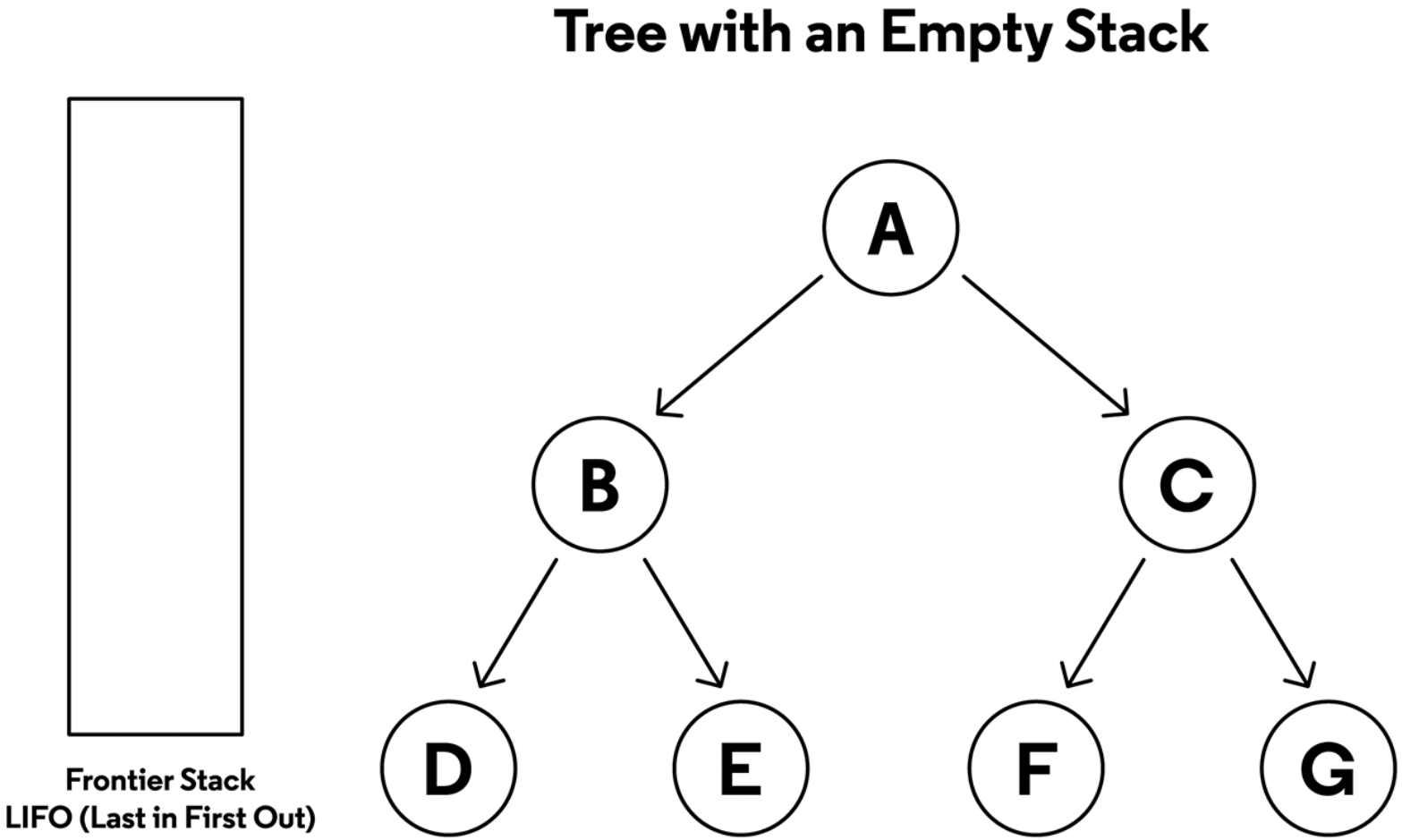
Breadth First Search

Depth First Search

Depth First Search

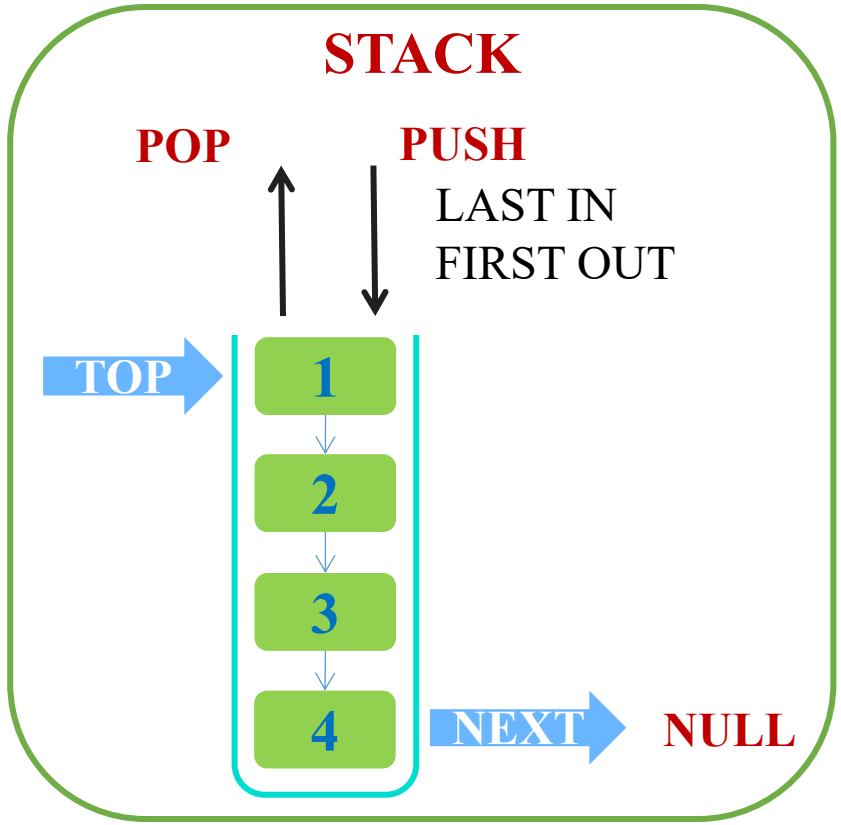


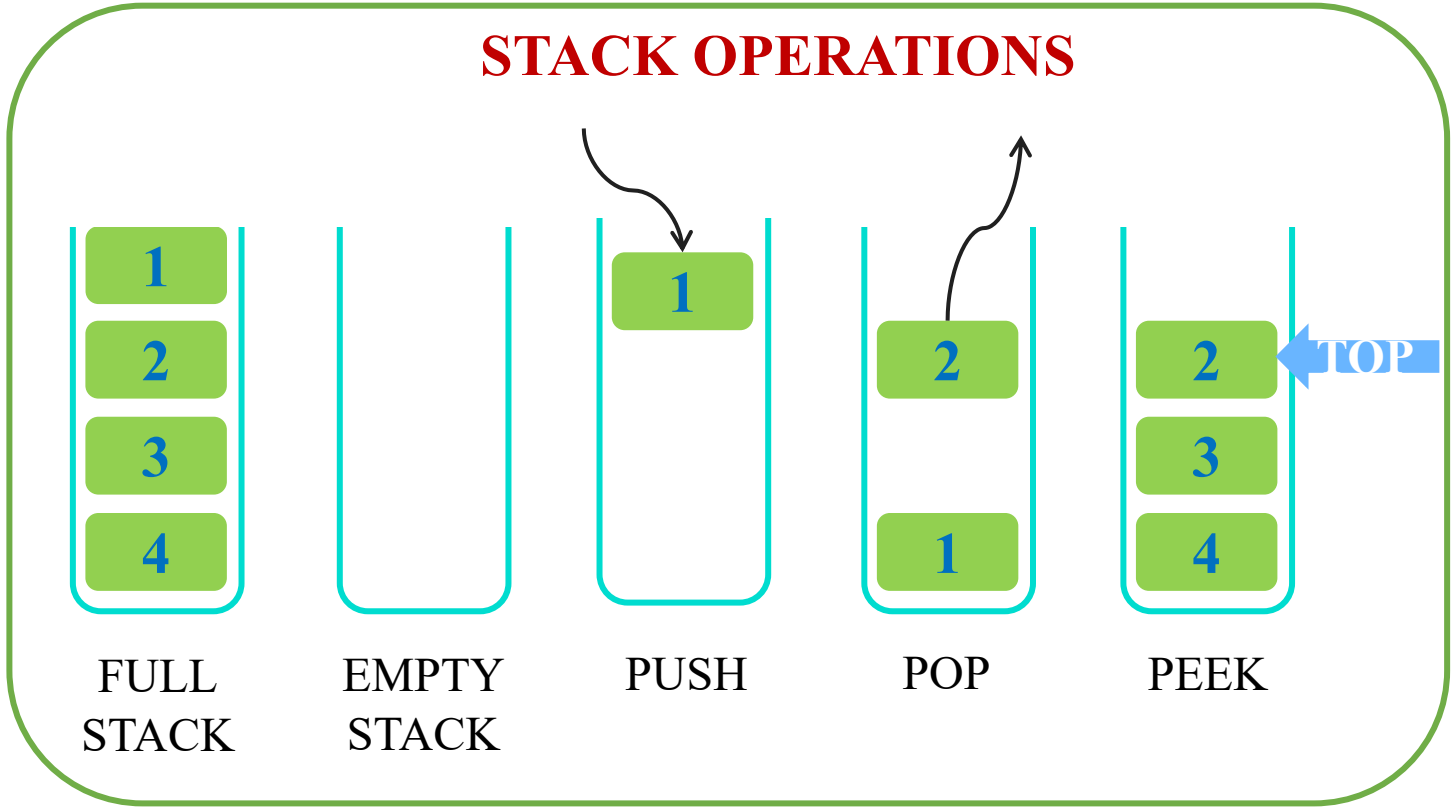
Depth First Search with Stack



Stack Data Structure Using List

- LIFO (Last in first out)
- Element are inserted and extracted only from ONE end





List Review

❖ Add an element

data = | 6 | 5 | 7 | 1 | 9 | 2 |

data.append(4) # thêm 4 vào vị trí cuối list

data = | 6 | 5 | 7 | 1 | 9 | 2 | 4 |

```
1 data = [6, 5, 7, 1, 9, 2]
2 print(data)
3 data.append(4)
4 print(data)
```

```
[6, 5, 7, 1, 9, 2]
[6, 5, 7, 1, 9, 2, 4]
```

❖ Deleting an element

data = | 6 | 5 | 7 | 1 | 9 | 2 |

data.pop(-1) # xóa phần tử ở vị trí cuối

data = | 6 | 5 | 7 | 1 | 9 |

```
1 data = [6, 5, 7, 1, 9, 2]
2 print(data)
3 data.pop(-1)
4 print(data)
```

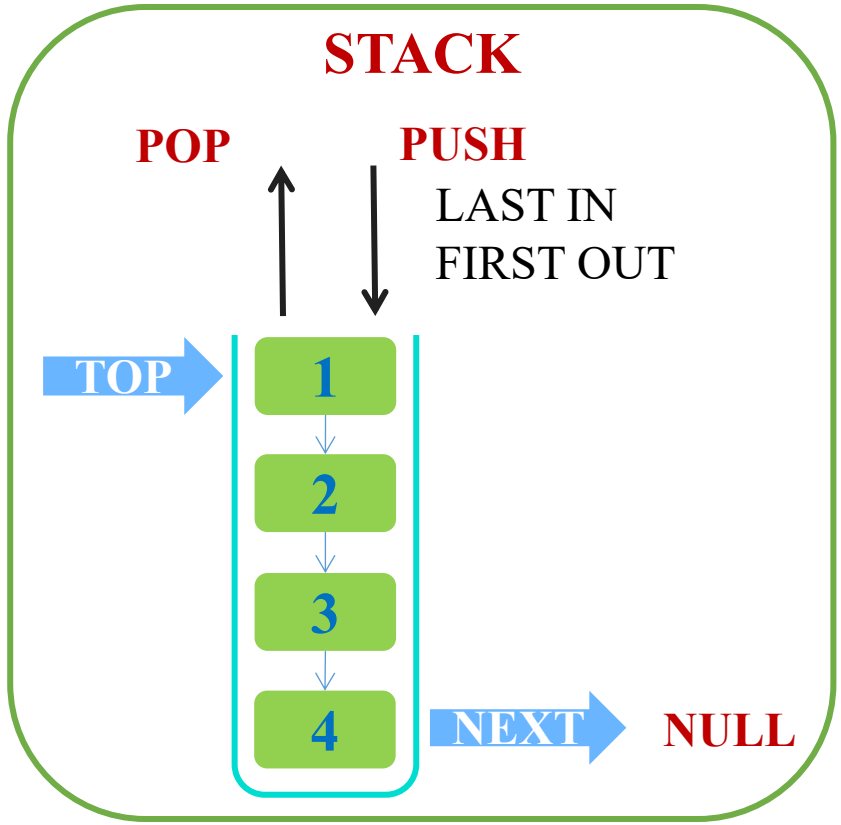
```
[6, 5, 7, 1, 9, 2]
[6, 5, 7, 1, 9]
```

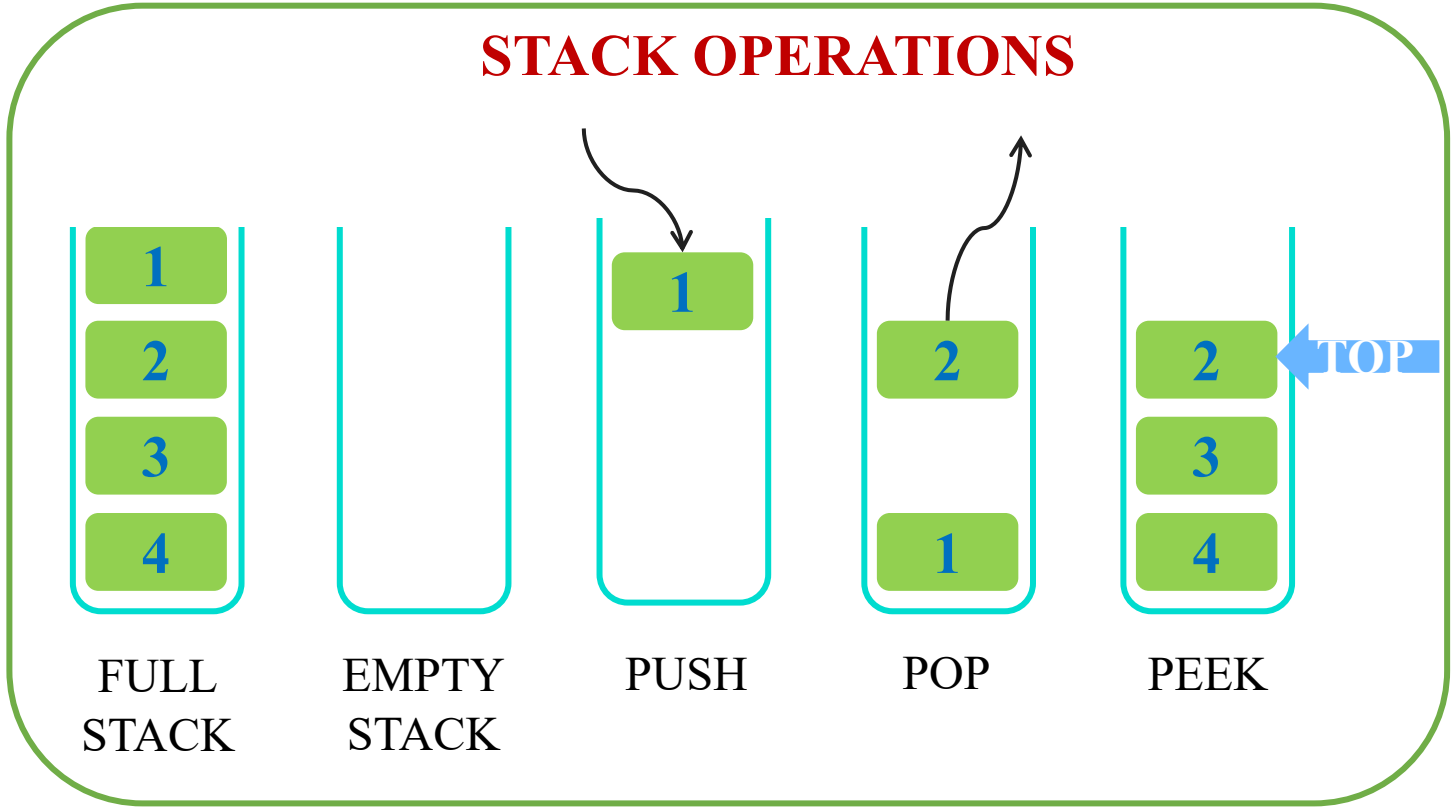
Outline

- What is a Tree Data Structure
- Algorithms on Trees
- Stack
- Queue
- What is a Binary Tree
- What is a Binary Search Tree
- Summary

Stack Data Structure Using List

- LIFO (Last in first out)
- Element are inserted and extracted only from ONE end





List Review

❖ Add an element

data = | 6 | 5 | 7 | 1 | 9 | 2 |

data.append(4) # thêm 4 vào vị trí cuối list

data = | 6 | 5 | 7 | 1 | 9 | 2 | 4 |

```
1 data = [6, 5, 7, 1, 9, 2]
2 print(data)
3 data.append(4)
4 print(data)
```

```
[6, 5, 7, 1, 9, 2]
[6, 5, 7, 1, 9, 2, 4]
```

❖ Deleting an element

data = | 6 | 5 | 7 | 1 | 9 | 2 |

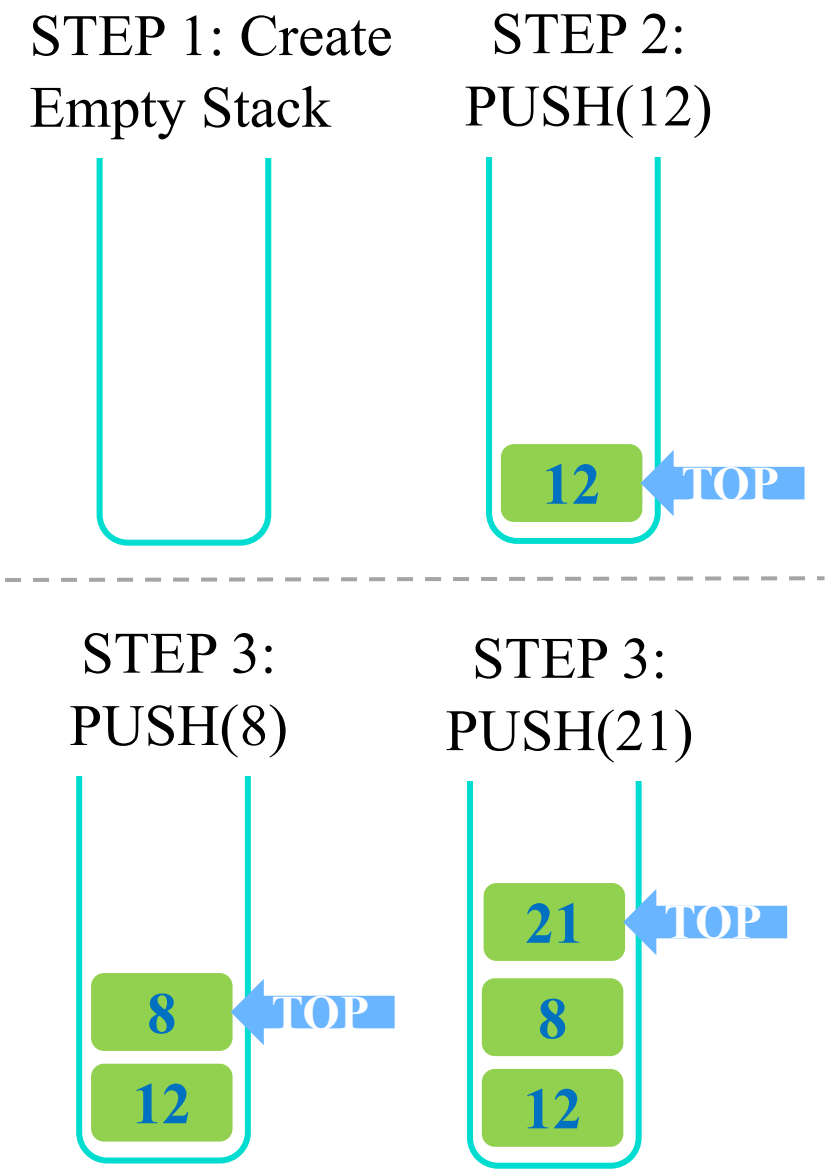
data.pop(-1) # xóa phần tử ở vị trí cuối

data = | 6 | 5 | 7 | 1 | 9 |

```
1 data = [6, 5, 7, 1, 9, 2]
2 print(data)
3 data.pop(-1)
4 print(data)
```

```
[6, 5, 7, 1, 9, 2]
[6, 5, 7, 1, 9]
```

Stack Data Structure Using List

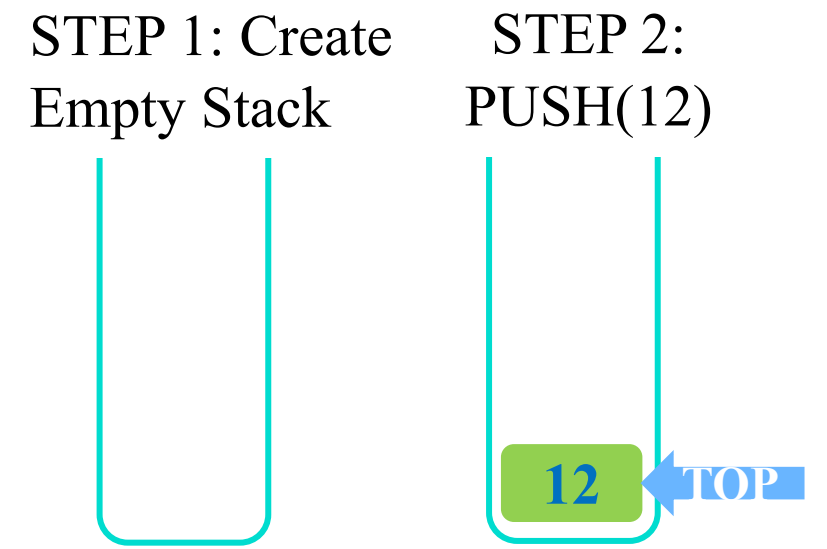


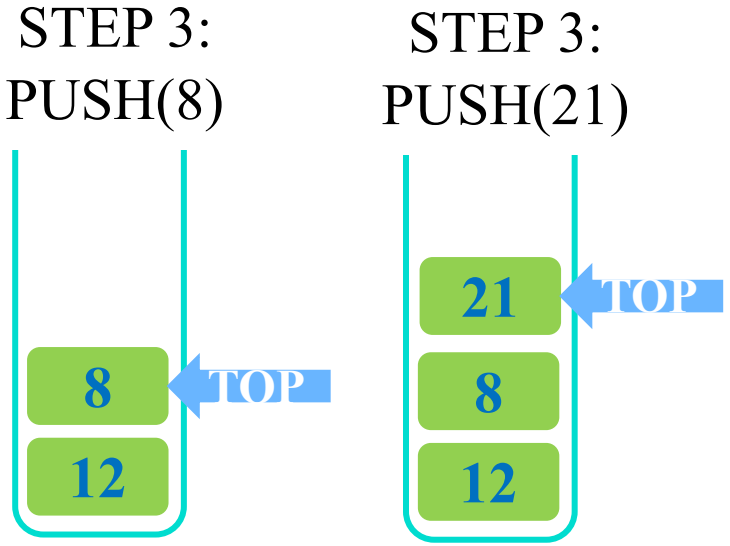

```python
class MyStack:
    def __init__(self, capacity):
        self.__capacity = capacity
        self.__stack = []

    def push(self, value):
        self.__stack.append(value)

    def print(self):
        print(self.__stack)
```

```python
stack = MyStack(5)
stack.push(12)
stack.push(8)
stack.push(21)

stack.print()
```

```
[12, 8, 21]
```

Stack Data Structure





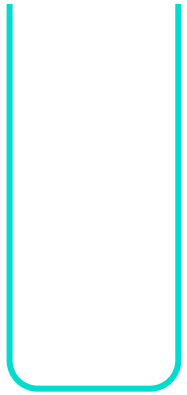

```python
class MyStack:
    def __init__(self, capacity):
        self.__capacity = capacity
        self.__stack = []
    def is_full(self):
        if len(self.__stack) == self.__capacity:
            return True
        else:
            return False
    def push(self, value):
        if self.is_full():
            print('Do nothing!')
        else:
            self.__stack.append(value)
    def print(self):
        print(self.__stack)
```

```python
stack = MyStack(5)
stack.push(12)
stack.push(8)
stack.push(21)
stack.print()
```

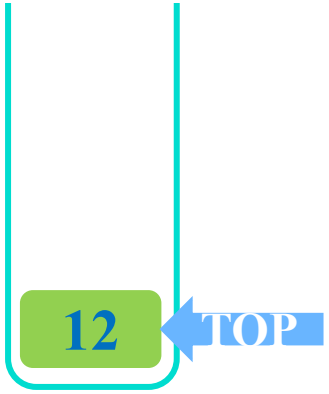
```
[12, 8, 21]
```

Stack Data Structure

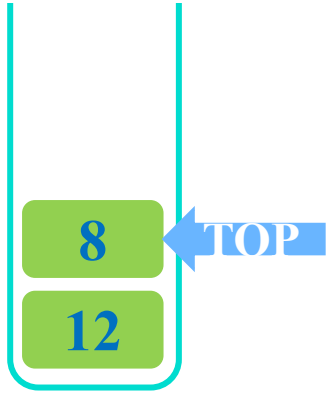
STEP 1: Create Empty Stack



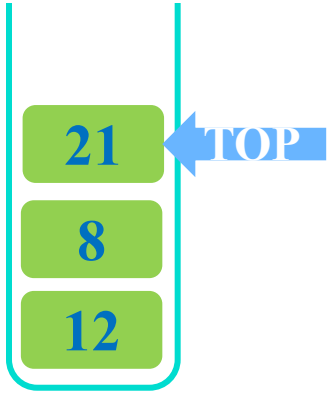
STEP 2: PUSH(12)



STEP 3: PUSH(8)



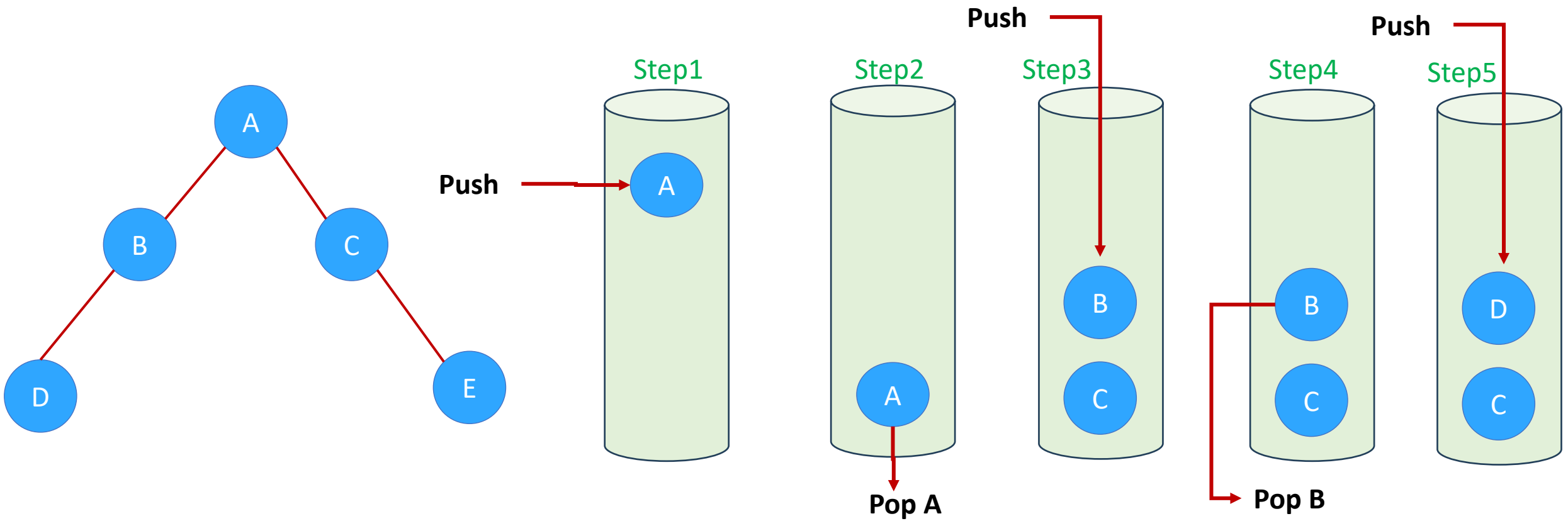
STEP 3: PUSH(21)



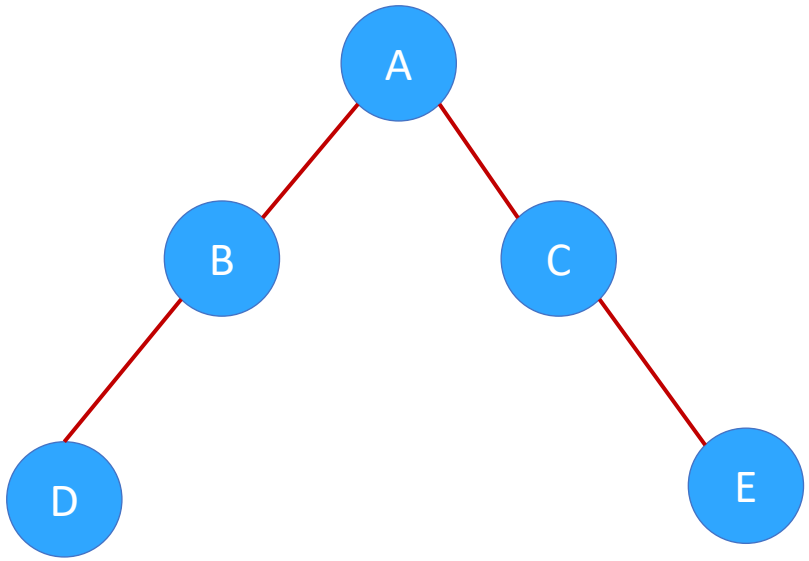

```python
class MyStack:
    def __init__(self, capacity):
        self.__capacity = capacity
        self.__stack = []
    def is_full(self):
        return len(self.__stack) == self.__capacity
    def push(self, value):
        if self.is_full():
            print('Do nothing!')
        else:
            self.__stack.append(value)
    def print(self):
        print(self.__stack)
```

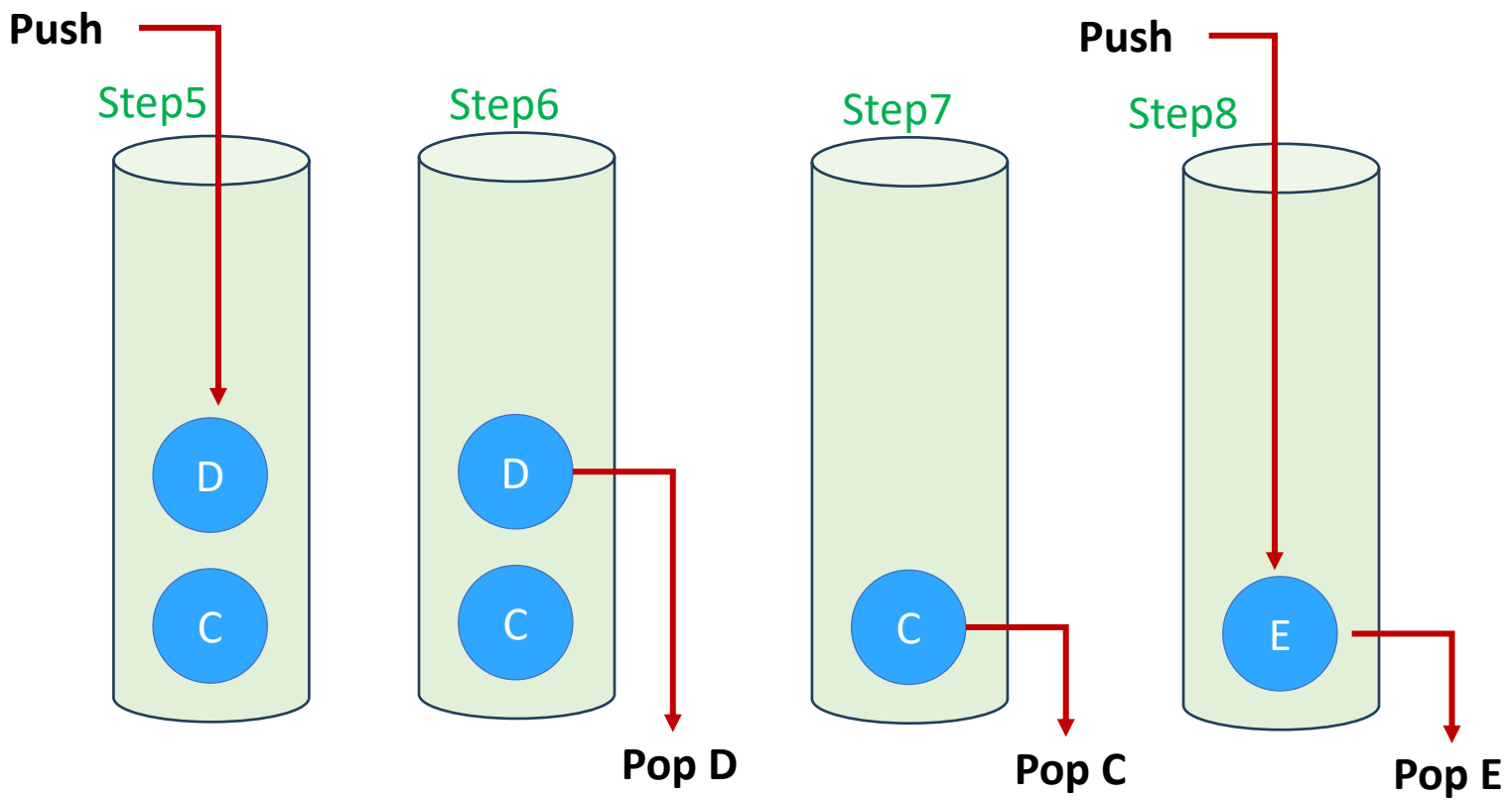
```python
stack = MyStack(5)
stack.push(12)
stack.push(8)
stack.push(21)
stack.push(33)
stack.push(34)
stack.push(35)
stack.print()
```

```
Do nothing!
[12, 8, 21, 33, 34]
```


A
B
C
D
E
Step1
Step2
Step3
Step4
Step5
Push
Push
Push
A
A
B
C
B
C
D
C
Pop A
Pop B

Visited : A, B





Visited : A, B, D, C, E

What is a Tree?

define Stack class

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed: {item}")

    def pop(self):
        if not self.is_empty():
            item = self.items.pop()
            print(f"Popped: {item}")
            return item

    def peek(self):
        if not self.is_empty():
            print(f"Peeked: {self.items[-1]}")
            return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)
```

Example

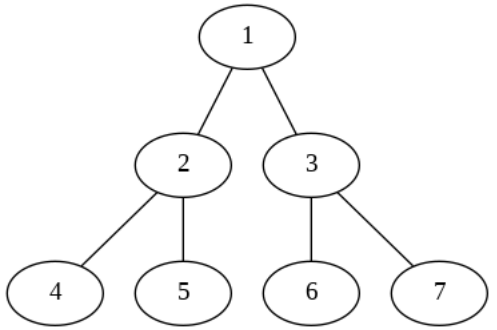
```
# Example usage
stack = Stack()

# Push elements
stack.push("Apple")
stack.push("Banana")
stack.push("Cherry")

# Peek at the top element
stack.peek()  # Should be "Cherry"

# Pop elements
stack.pop()   # Removes "Cherry"
stack.pop()   # Removes "Banana"

# Peek again
stack.peek()  # Should now be "Apple"

#Check size
print("Current size:", stack.size())

# Final pop
stack.pop()   # Removes "Apple"

#Is the stack empty now?
print("Is empty?", stack.is_empty())
```

define DFS function

What is a Tree?

```
class BinaryTree:
    def __init__(self, root=None):
        self.root = root

    def dfs(self):
        if not self.root:
            print("Tree is empty.")
            return

        visited = set()
        stack = Stack()
        stack.push(self.root)

        print("DFS Traversal (Pre-order):")

        while not stack.is_empty():
            current = stack.pop()
            if current.val not in visited:
                print(current.val)
                visited.add(current.val)

            # Push right first so left is processed first
            if current.right:
                stack.push(current.right)
            if current.left:
                stack.push(current.left)
```

```
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```



define DFS function

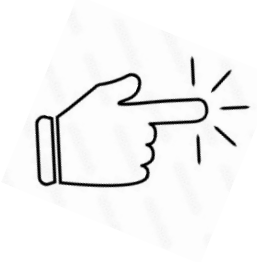
What is a Tree?

```
class BinaryTree:
    def __init__(self, root=None):
        self.root = root

    def dfs(self):
        if not self.root:
            print("Tree is empty.")
            return

        visited = set()
        stack = Stack()
        stack.push(self.root)

        print("DFS Traversal (Pre-order):")

        while not stack.is_empty():
            current = stack.pop()
            if current.val not in visited:
                print(current.val)
                visited.add(current.val)

            # Push right first so left is processed first
            if current.right:
                stack.push(current.right)
            if current.left:
                stack.push(current.left)
```

Example

```
# Create binary tree nodes
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Create the tree and perform DFS
binary_tree = BinaryTree(root)
binary_tree.dfs()
```

Outline

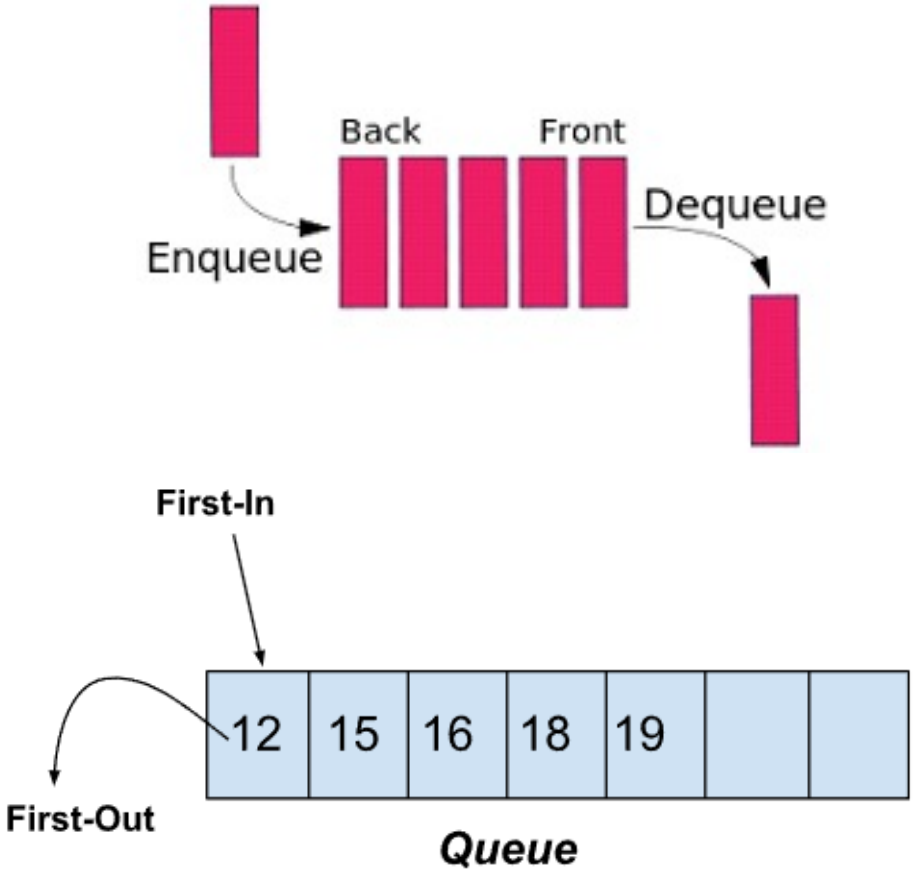


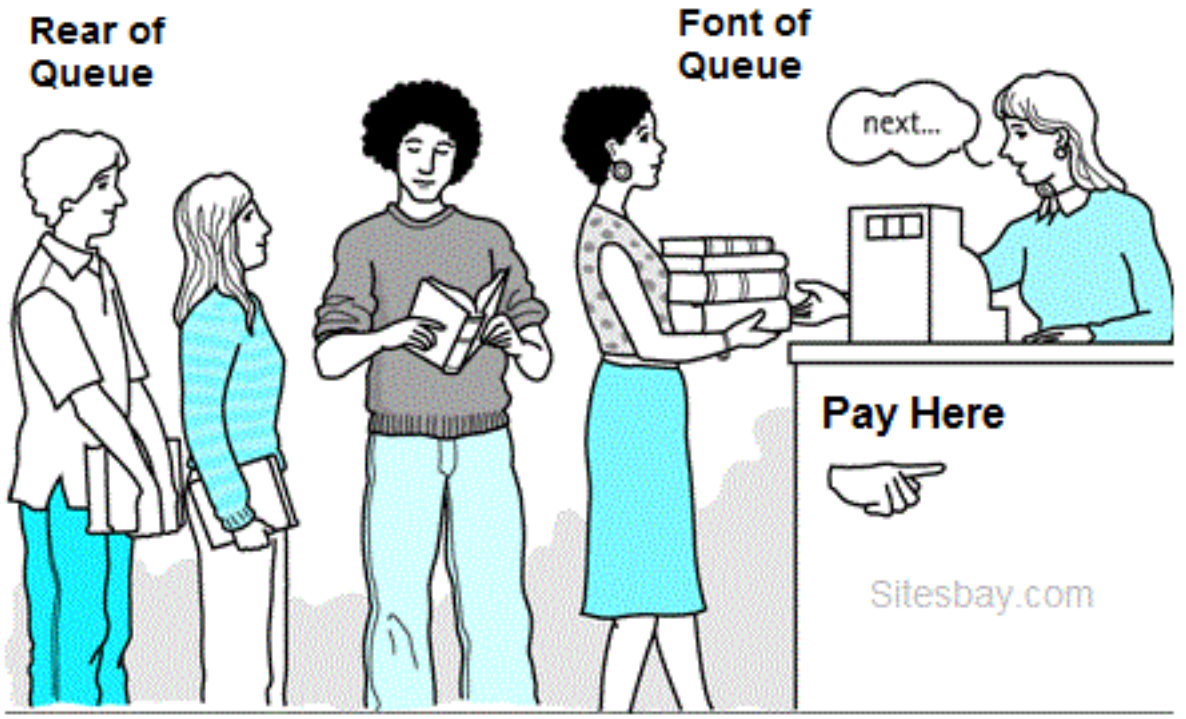
- What is a Tree Data Structure
- Algorithms on Trees
- Stack
- Queue
- What is a Binary Tree
- What is a Binary Search Tree
- Summary

AI VIET NAM
@aivietnam.edu.vn

Queue

A queue is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. It operates like a line where elements are added at one end (**rear**) and removed from the other end (**front**).





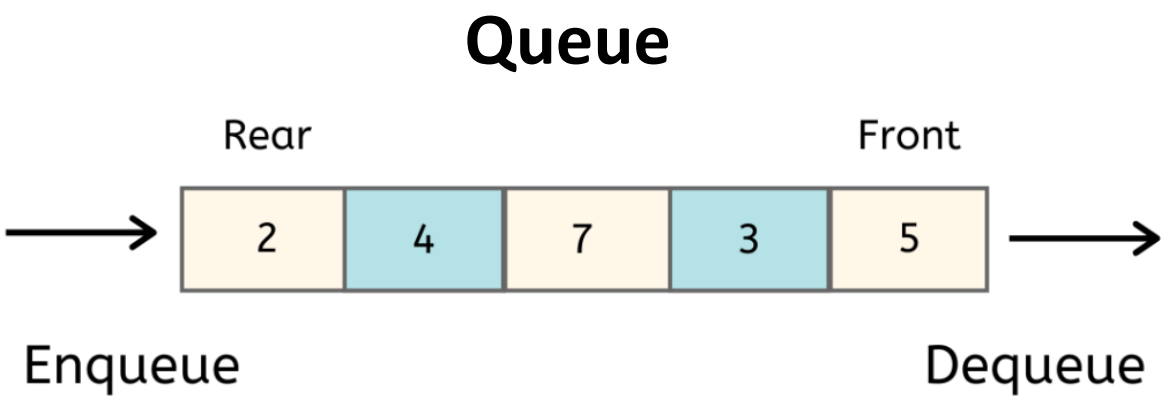
Real Life Example of Queue : Library Counter

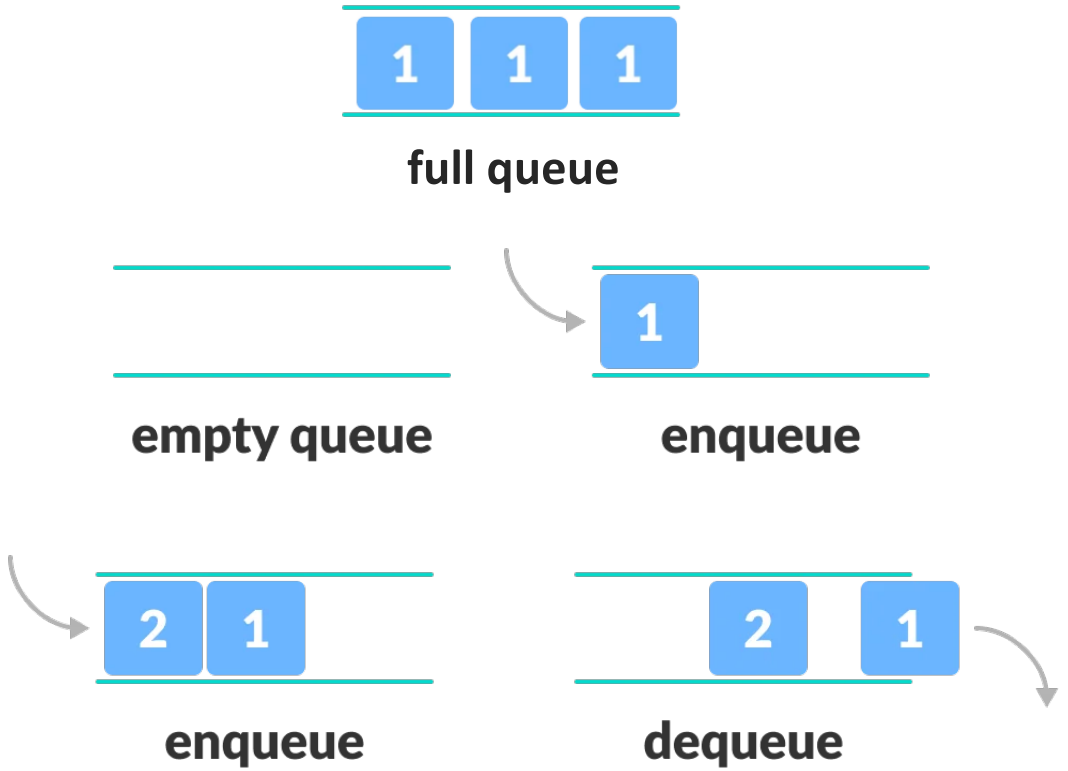
Queue Data Structure

❖ Introduction

- Operate in a **FIFO (First in First out)** context
- **Element** are **inserted** (enqueue) and **extracted** (dequeue) happens at **OPPOSITE** ends

Basic Operations of Queue
- **enqueue**: Add an element to the end of the queue
- **dequeue**: Remove an element from the front of the queue
- **is_empty**: Check if the queue is empty
- **is_full**: Check if the queue is full
- **peek**: Get value of the front of the queue without removing it





List Review

❖ Add an element

data = | 6 | 5 | 7 | 1 | 9 | 2 |

data.append(4) # thêm 4 vào vị trí cuối list

data = | 6 | 5 | 7 | 1 | 9 | 2 | 4 |

```
1 data = [6, 5, 7, 1, 9, 2]
2 print(data)
3 data.append(4)
4 print(data)
```

```
[6, 5, 7, 1, 9, 2]
[6, 5, 7, 1, 9, 2, 4]
```

❖ Deleting an element

data = | 6 | 5 | 7 | 1 | 9 | 2 |

data.pop(0) # xóa phần tử ở vị trí đầu tiên

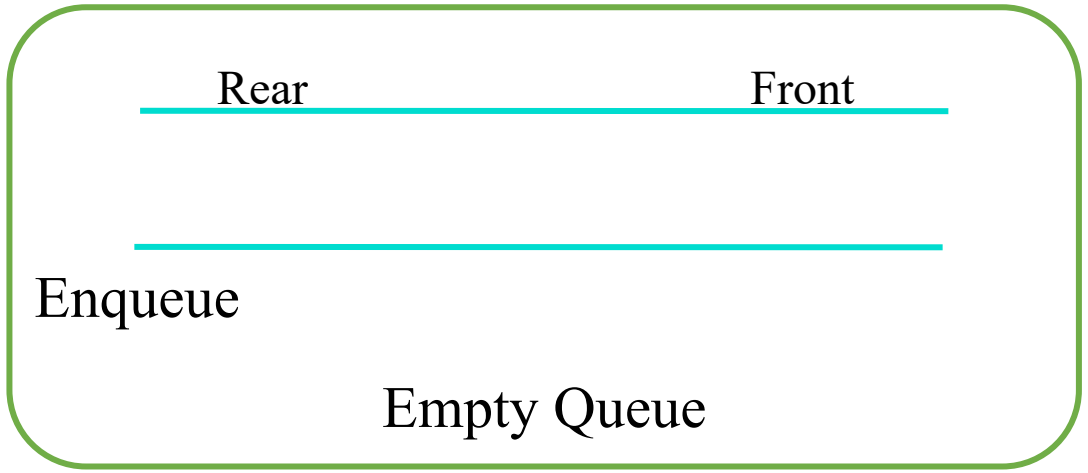
data = | 5 | 7 | 1 | 9 | 2 |

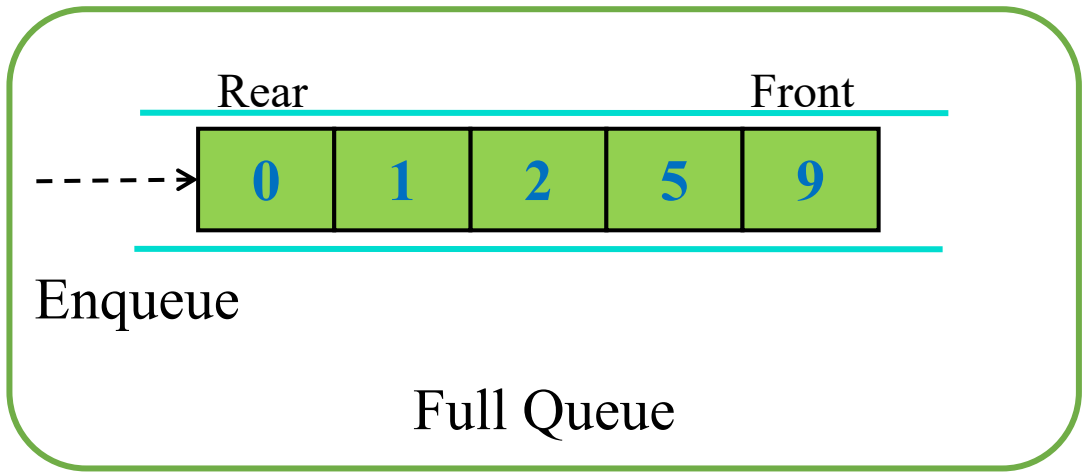
```
1 data = [6, 5, 7, 1, 9, 2]
2 print(data)
3 data.pop(0)
4 print(data)
```

```
[6, 5, 7, 1, 9, 2]
[5, 7, 1, 9, 2]
```

Queue Data Structure

❖ enqueue()





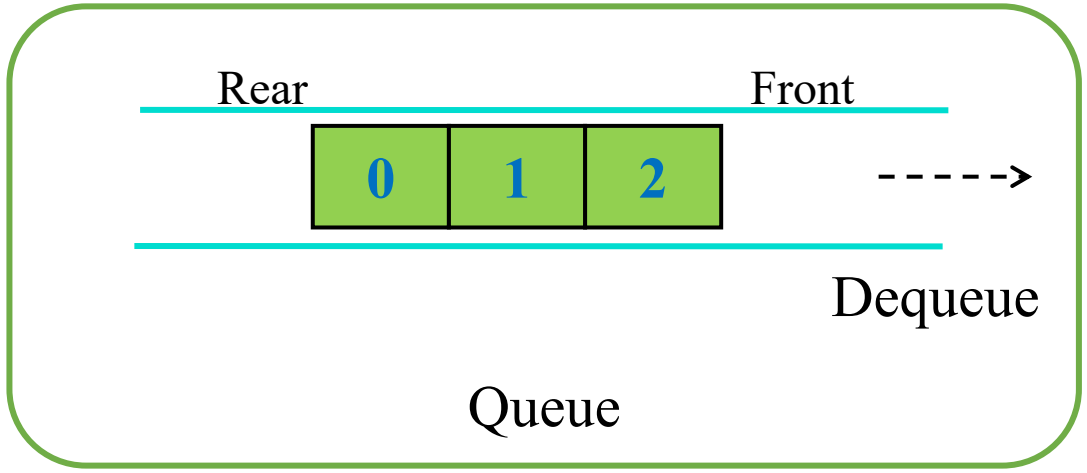

```python
class MyQueue:
    def __init__(self, capacity):
        self.__capacity = capacity
        self.__data = []

    def is_full(self):
        return len(self.__data) == self.__capacity

    def enqueue(self, value):
        if self.is_full():
            print('Do nothing!')
        else:
            self.__data.append(value)

    def print(self):
        print(self.__data)
```

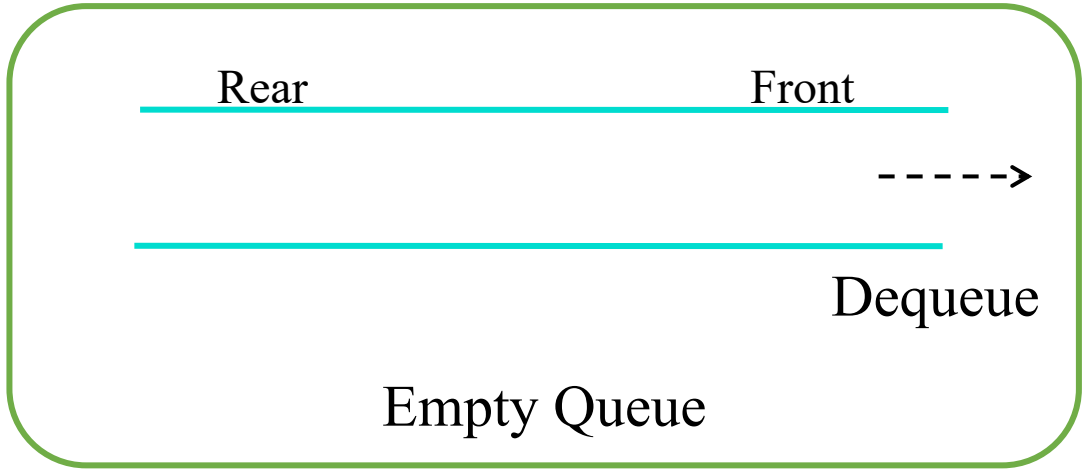
```python
queue = MyQueue(5)
queue.print()

queue.enqueue(9)
queue.enqueue(5)
queue.enqueue(2)
queue.enqueue(1)
queue.enqueue(0)
queue.enqueue(6)
queue.print()
```

```
[]
Do nothing!
[9, 5, 2, 1, 0]
```

Queue Data Structure

❖ dequeue()






```python
class MyQueue:
    def __init__(self, capacity):
        self.__capacity = capacity
        self.__data = []

    def is_empty(self):
        return len(self.__data) == 0

    def dequeue(self):
        if self.is_empty():
            print('Do nothing!')
            return None
        else:
            return self.__data.pop(0)

    def print(self):
        print(self.__data)
```

```python
queue = MyQueue(5)
#...
queue.print()

queue.dequeue()
queue.dequeue()
queue.dequeue()
queue.print()
```

```
[2, 1, 0]
[]
```

Queue Implementation

❖ Using List

```
queue = []
queue.append('a')
queue.append('b')
queue.append('c')
print("Initial queue")
print(queue)
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))
print("\nQueue after removing elements")
print(queue)
```

❖ Using deque

```
from collections import deque
q = deque()
q.append('a')
q.append('b')
q.append('c')
print("Initial queue")
print(q)
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nQueue after removing elements")
print(q)
```

❖ Using Queue

```
from queue import Queue
q = Queue(maxsize = 3)
print(q.qsize())
q.put('a')
q.put('b')
q.put('c')
print("\nFull: ", q.full())
print("\nElements dequeued from the queue")
print(q.get())
print(q.get())
print(q.get())
print("\nEmpty: ", q.empty())
q.put(1)
print("\nEmpty: ", q.empty())
print("Full: ", q.full())
```

Results

```
Initial queue
['a', 'b', 'c']

Elements dequeued from queue
a
b
c

Queue after removing elements
[]
```

Queue Implementation

❖ Using List (OOP)

```
class QueueList:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if self.is_empty():
            return None
        return self.queue.pop(0)

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)
```

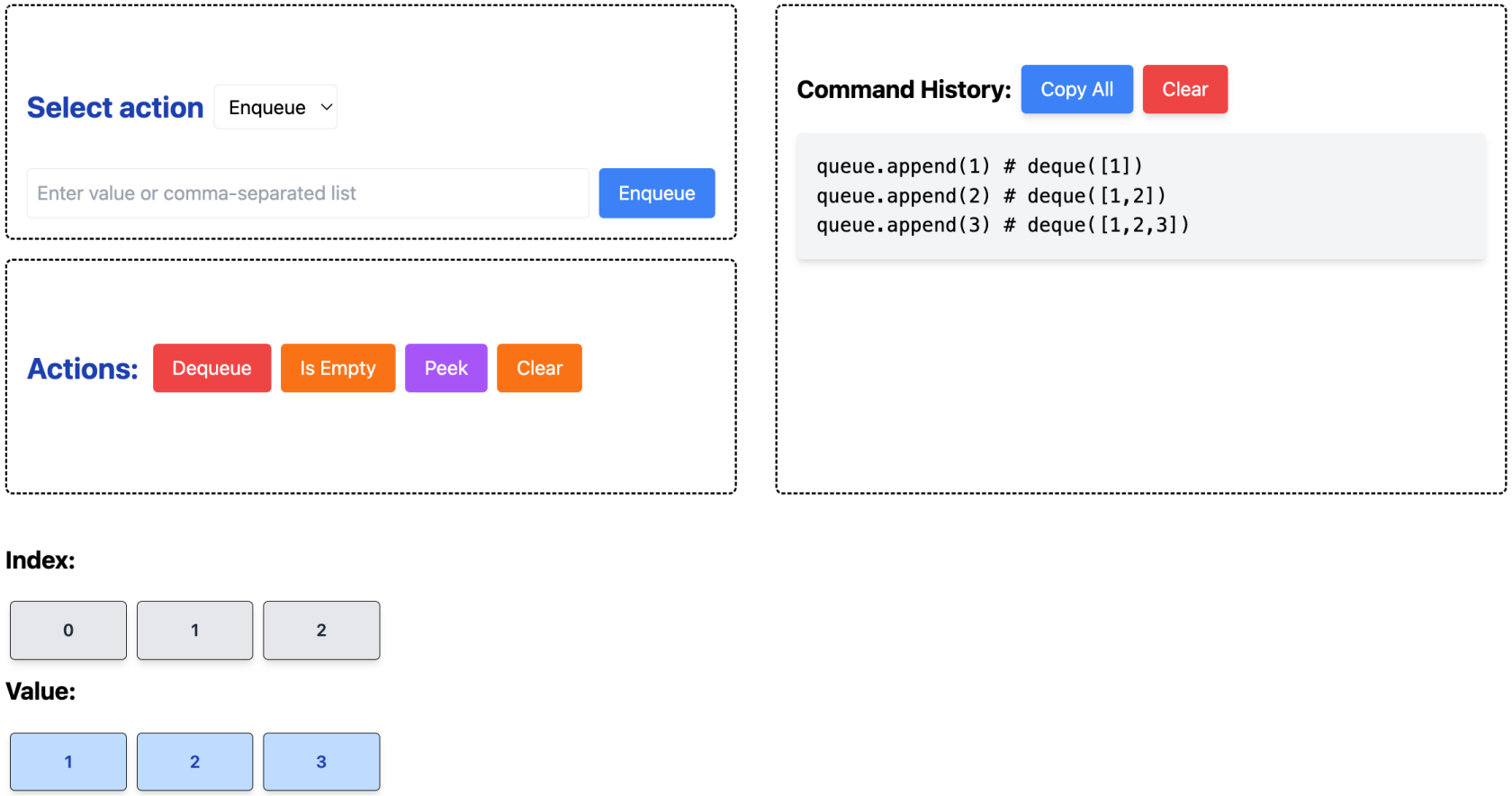
❖ Using dequeue (OOP)

```
from collections import deque

class QueueDeque:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if self.is_empty():
            return None
        return self.queue.popleft()

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)
```

Time complexity

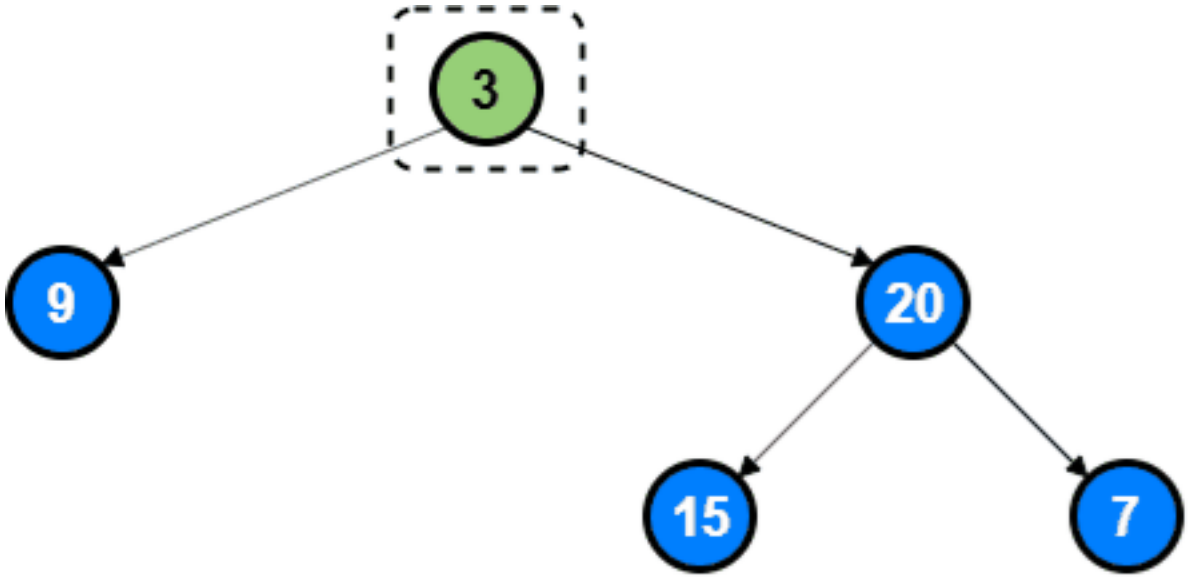
AI VIET NAM
@aivietnam.edu.vn

Queue Visualizer



https://ducnd58233.github.io/python-viz/#/queue

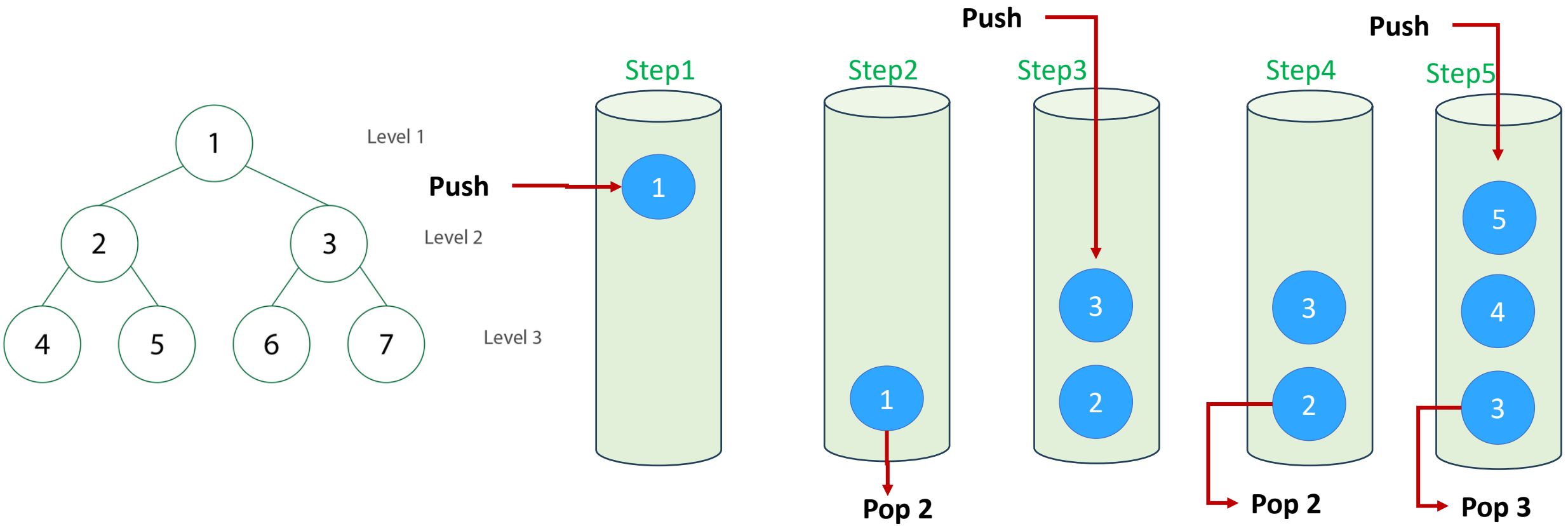
Breadth First Search



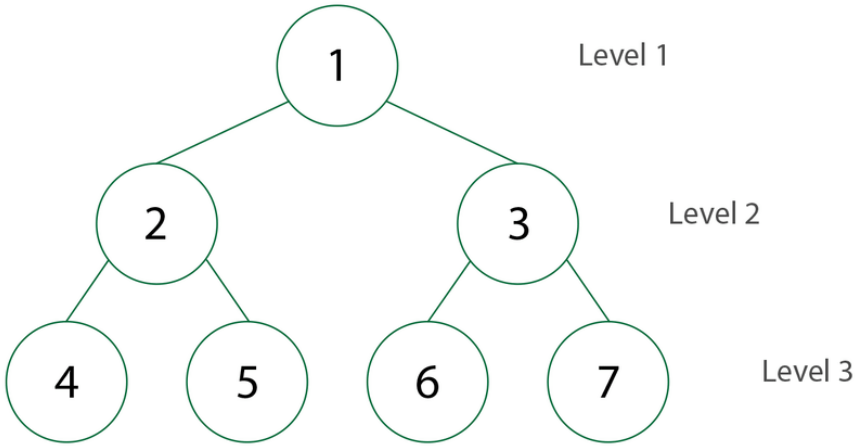
Level Size: 1

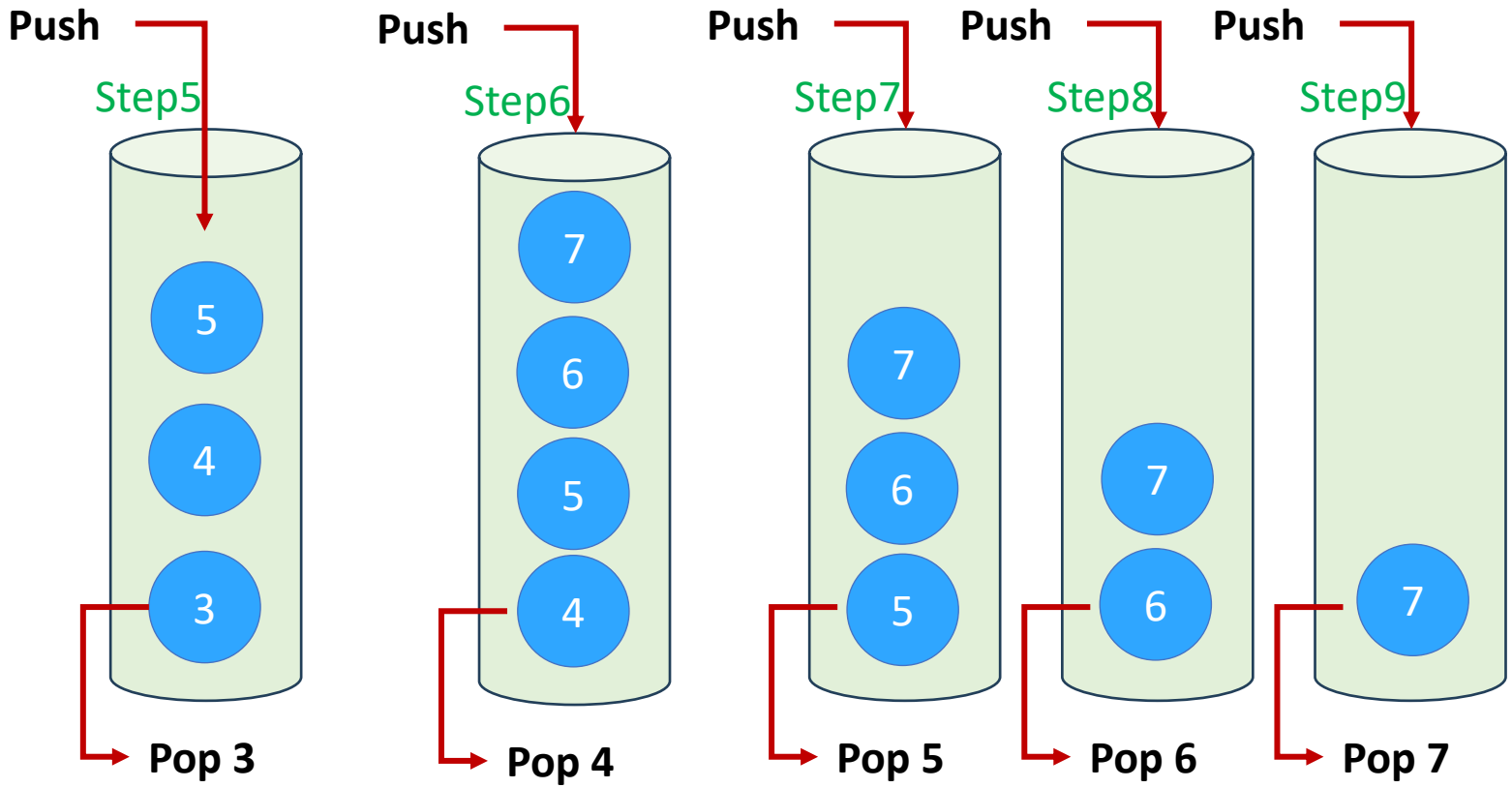
Queue:

3


1
2
3
4
5
6
7
Level 1
Level 2
Level 3
Push
Step1
Step2
Step3
Step4
Step5
Push
Push
Pop 2
Pop 2
Pop 3

Visited : 1, 2, 3



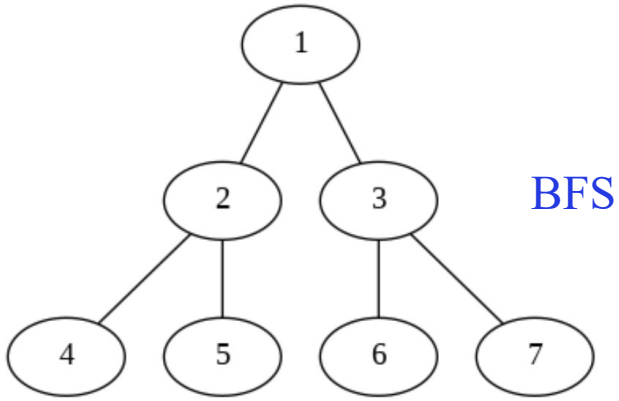


Visited : 1, 2, 3, 4, 5, 6, 7

Breadth First Search

```python
from collections import deque

def bfs(root):
    if root is None:
        return []

    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        result.append(node.val)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return result
```

```python
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

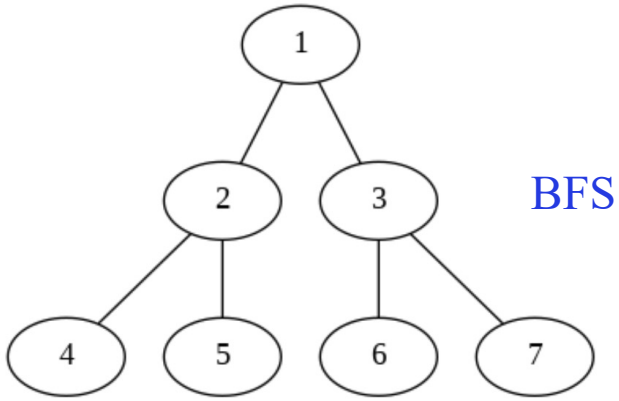


BFS traversal result: [1, 2, 3, 4, 5, 6, 7]

Breadth First Search

```python
from collections import deque

def bfs(root):
    if root is None:
        return []

    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        result.append(node.val)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return result
```

```python
if __name__ == '__main__':
    # Create the root of the tree
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)
    root.right.left = TreeNode(6)
    root.right.right = TreeNode(7)

    # Perform BFS
    bfs_result = bfs(root)
    print("BFS traversal result:", bfs_result)
```

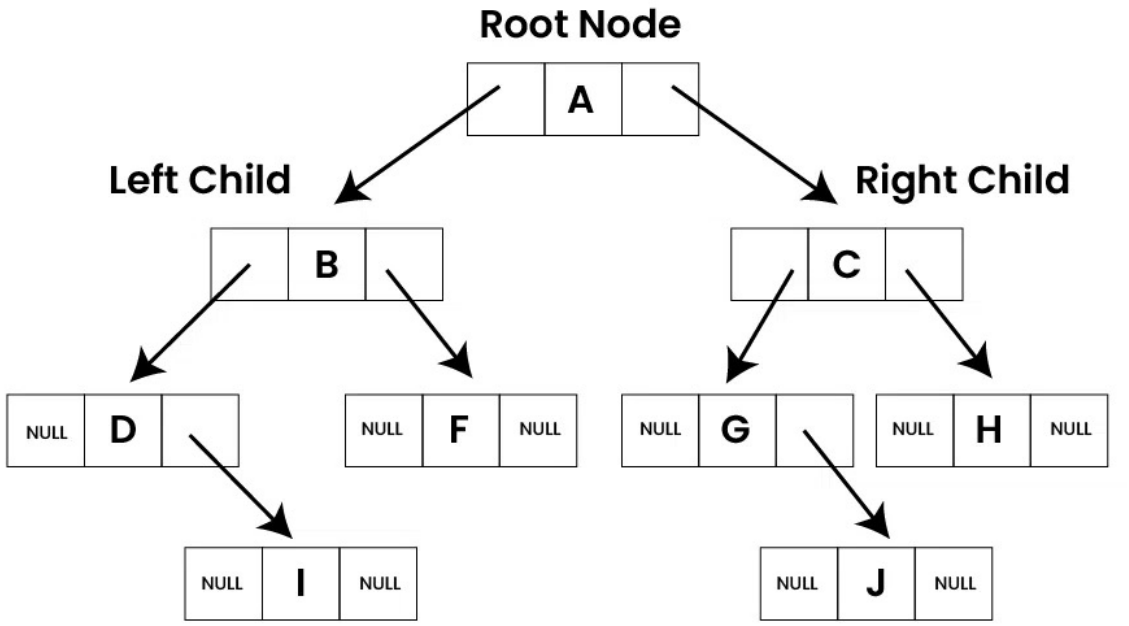


BFS traversal result: [1, 2, 3, 4, 5, 6, 7]

Outline

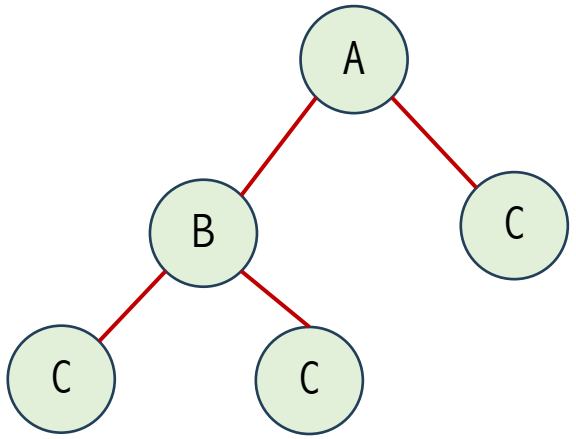
- What is a Tree Data Structure
- Algorithms on Trees
- Stack
- Queue
- What is a Binary Tree
- What is a Binary Search Tree
- Summary

Binary Tree

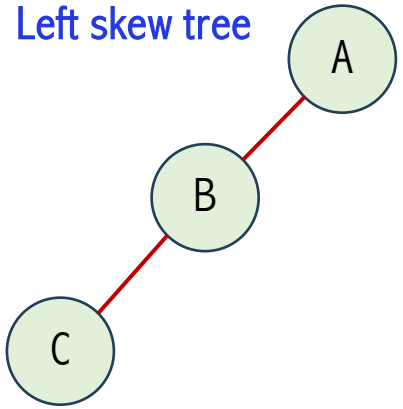


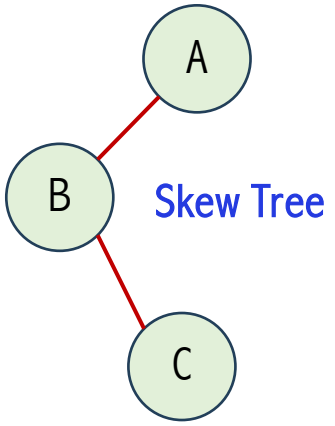
Binary tree is a tree data structure(non-linear) in which each node can have at most two children which are referred to as the left child and the right child

Full binary tree

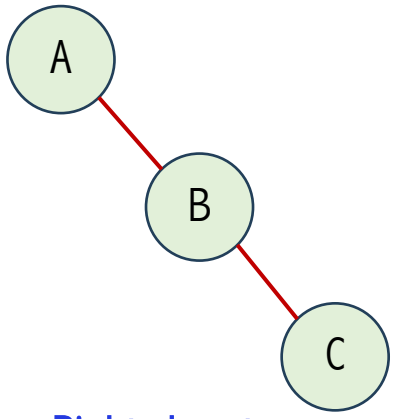


Left skew tree





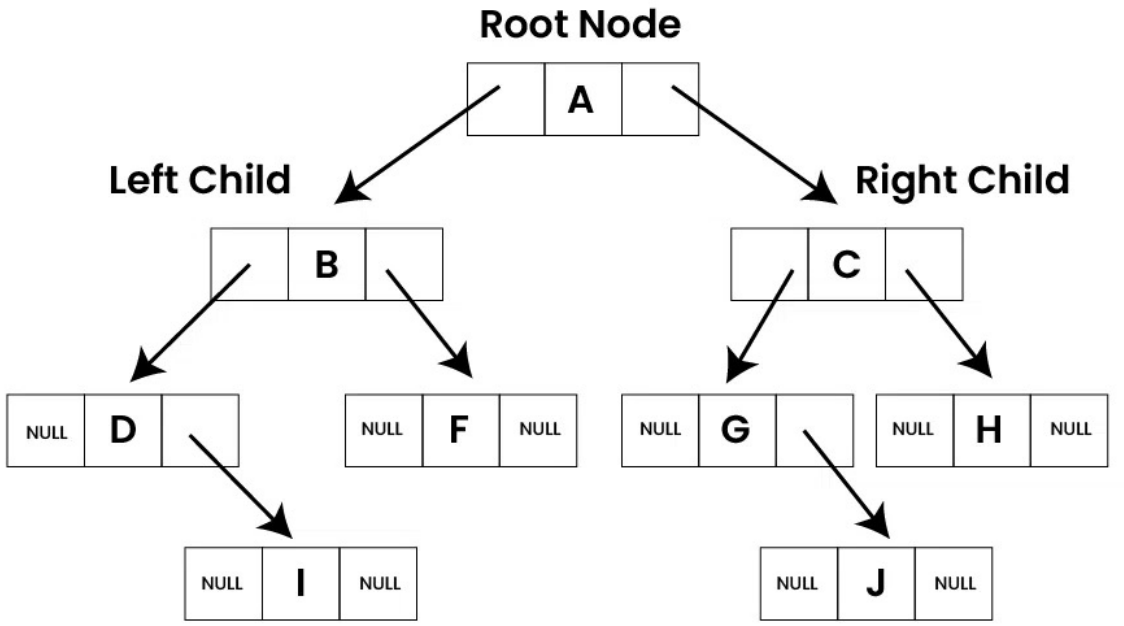
Skew Tree



Right skew tree

AI VIET NAM
@aivietnam.edu.vn

Balanced Binary Tree



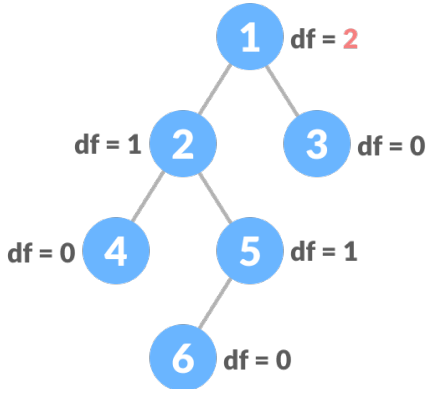
The height of a node in a tree is the length of the longest path from that node downward to a leaf.

Height of a **tree with one node** (just the root) = **0**

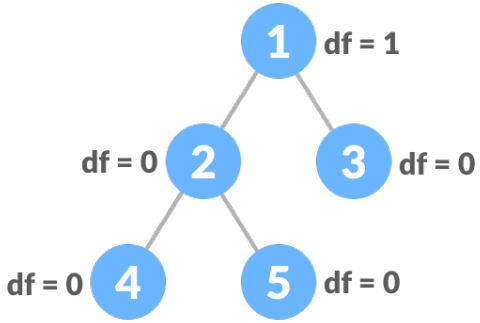
Conditions for a height-balanced binary tree

- The difference **(df)** between the heights of the left and the right subtree for any node is not more than one.
- The left subtree is balanced.
- The right subtree is balanced.

Unbalanced Binary Tree

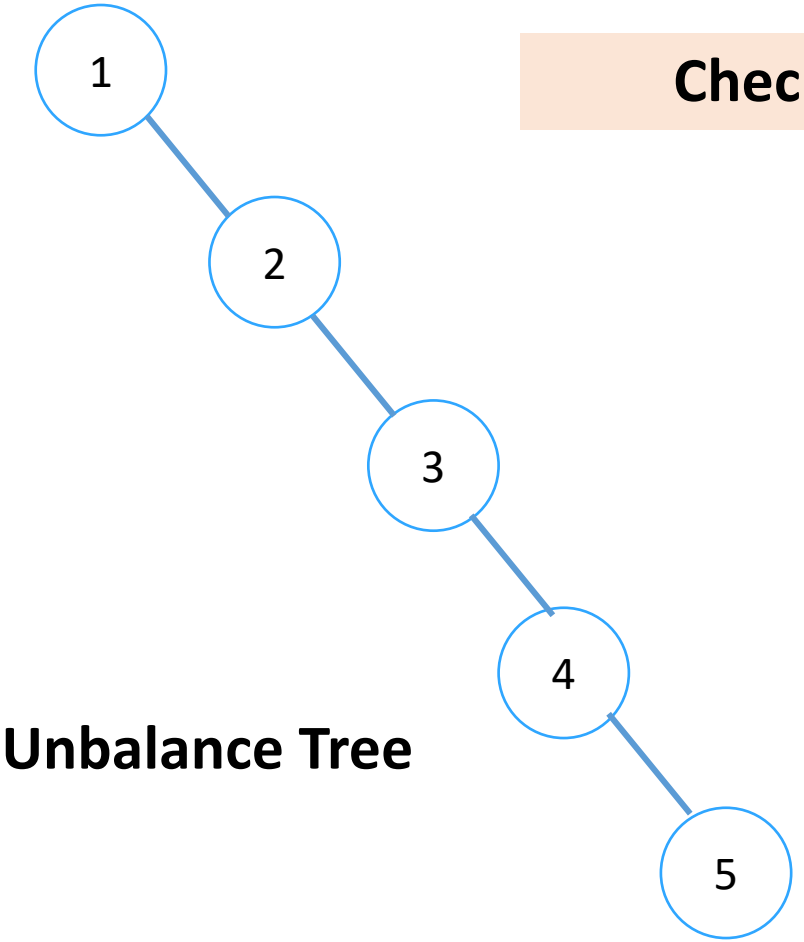


Balanced Binary Tree



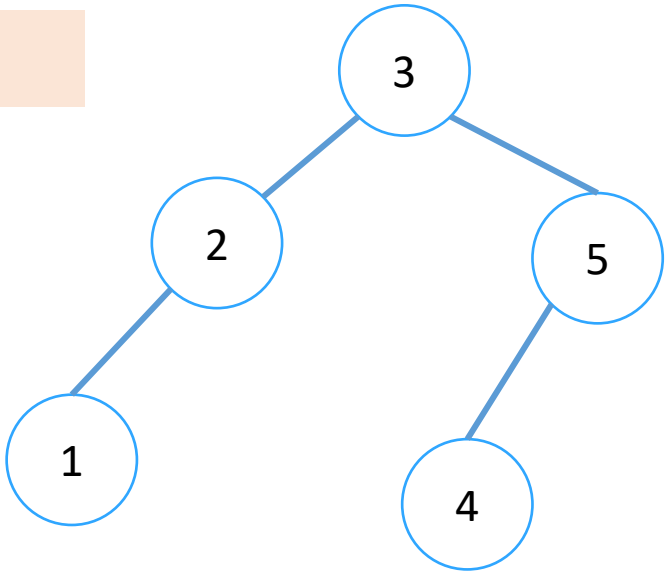
Why Balanced Binary Tree?

Dataset: 1 → 2 → 3 → 4 → 5



Unbalance Tree

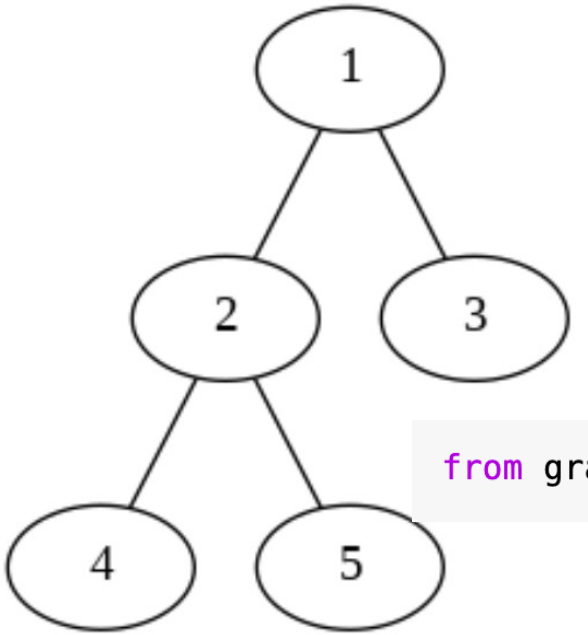
Check if 5 exist?



Balance Tree

Operations On Binary Tree

```python
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```




```python
from graphviz import Graph
```

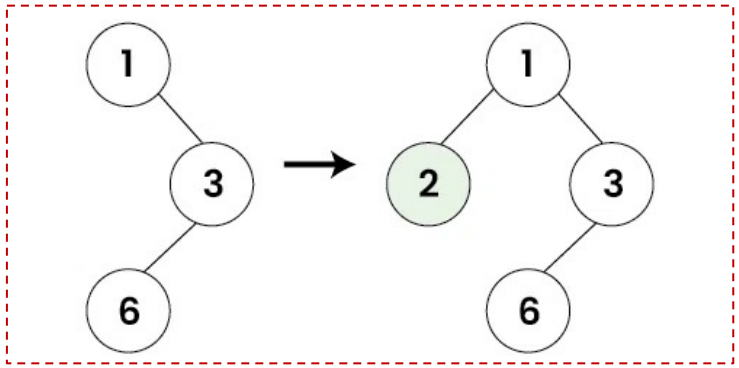
```python
# Create the root of the tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

dot = draw_tree(root)
dot.render('orignial_tree', format='png', view=True)
```

Tree visualization

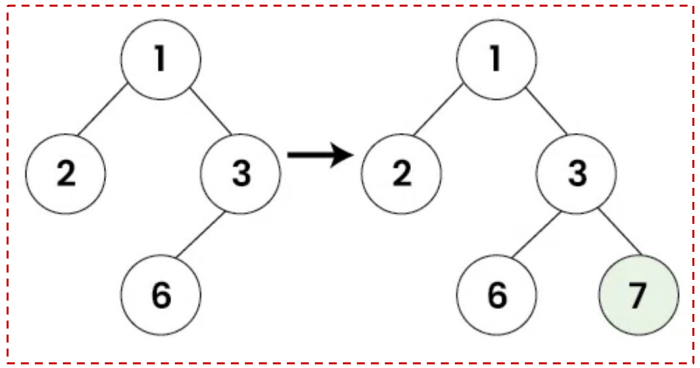
```python
def add_edges(dot, node):
    if node is None:
        return
    if node.left:
        dot.edge(str(node.val), str(node.left.val))
        add_edges(dot, node.left)
    if node.right:
        dot.edge(str(node.val), str(node.right.val))
        add_edges(dot, node.right)

def draw_tree(root):
    dot = Graph()
    dot.node(str(root.val))
    add_edges(dot, root)
    return dot
```

Operations On Binary Tree

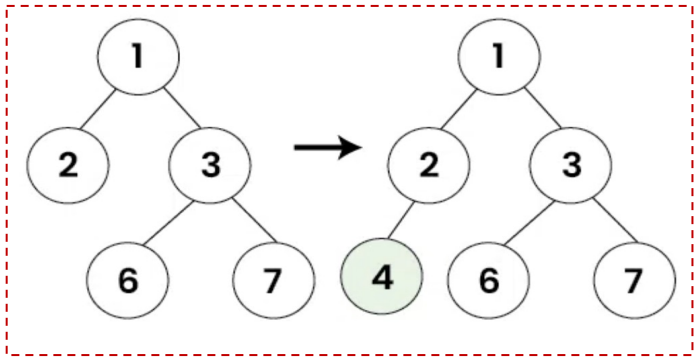
Insert a node in a Binary Tree



Node 2 is inserted to the left of Node 1 as it has a missing left child



Node 7 is inserted to the right of Node 3 as it has a missing left child



Node 4 is inserted to the right of Node 2 as it has a missing left child

Insert a Node

```
def insert_node(root, key):
    # Create a new node
    new_node = TreeNode(key)

    # If the tree is empty, the new node becomes the root
    if root is None:
        return new_node

    # Use a queue to perform level-order traversal
    queue = []
    queue.append(root)

    # Traverse the tree
    while queue:
        # Dequeue a node
        temp = queue.pop(0)

        # Check if the left child is empty
        if temp.left is None:
            temp.left = new_node
            return root
        else:
            queue.append(temp.left)

        # Check if the right child is empty
        if temp.right is None:
            temp.right = new_node
            return root
        else:
            queue.append(temp.right)

    return root
```

Algorithm

1. Initialize:

- Create a new node with the given value.
- If the tree is empty, the new node becomes the root of the tree

.

2. Level-Order Traversal:

- Use a queue to perform a level-order traversal (BFS).
- Start by enqueueing the root node.

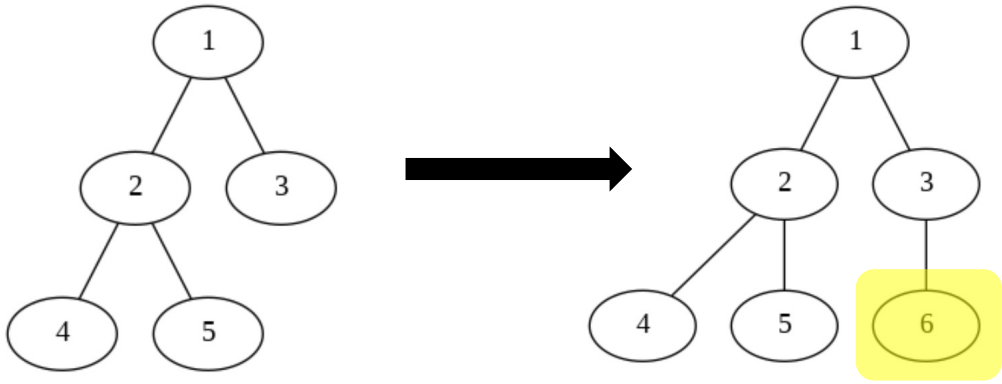
3. Traversal Loop:

- While the queue is not empty:
  - Dequeue a node from the front of the queue.
  - Check if the dequeued node has a left child:
    - If not, set the left child to the new node and return.
    - If yes, enqueue the left child to the queue.
  - Check if the dequeued node has a right child:
    - If not, set the right child to the new node and return.
    - If yes, enqueue the right child to the queue.

AI VIET NAM
@aivietnam.edu.vn

Insert a Node

```
# Driver code
if __name__ == '__main__':
    # Create the root of the tree
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)

    dot = draw_tree(root)
    dot.render('orignial_tree', format='png', view=True)

    key = 6
    print(f"\nInserting {key} into the binary tree.")
    root = insert_node(root, key)

    dot = draw_tree(root)
    dot.render('after_tree', format='png', view=True)
```

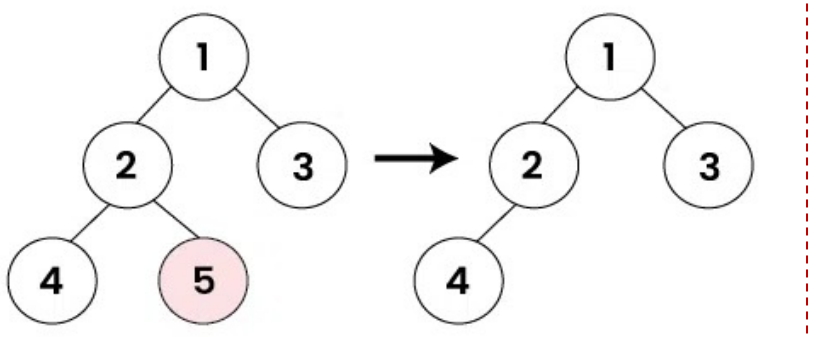


Visualization using `graphviz`

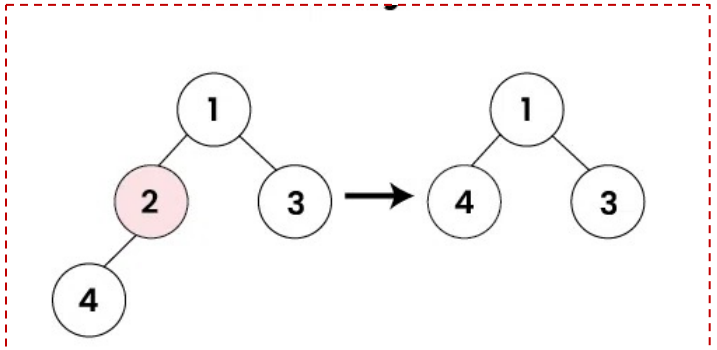
```
def add_edges(dot, node):
    if node is None:
        return
    if node.left:
        dot.edge(str(node.val), str(node.left.val))
        add_edges(dot, node.left)
    if node.right:
        dot.edge(str(node.val), str(node.right.val))
        add_edges(dot, node.right)

def draw_tree(root):
    dot = Graph()
    dot.node(str(root.val))
    add_edges(dot, root)
    return dot
```

Operations On Binary Tree

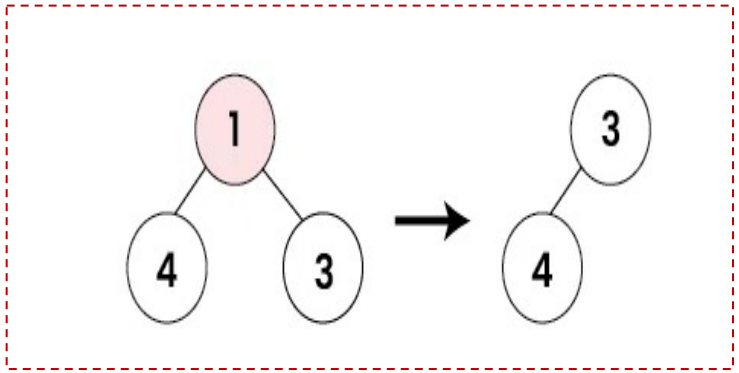
Delete a node in a Binary Tree



Leaf nodes can be deleted without any shifting of nodes



Node 2 is deleted and replaced with the deepest node 4

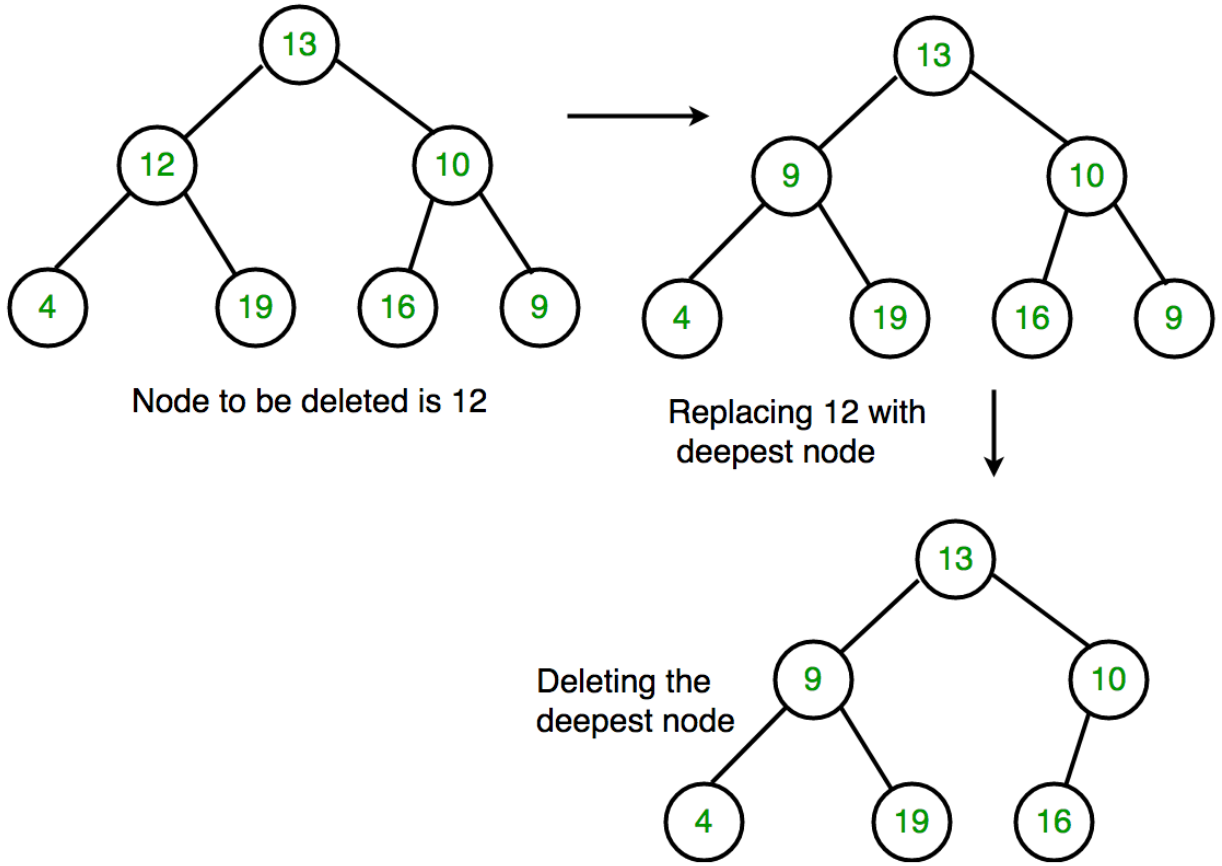


Node 1 is deleted and relaced with deepest node 3

We can delete any node in the binary tree and rearrange the nodes after deletion to again form a valid binary tree.

Operations On Binary Tree

Delete a node in a Binary Tree



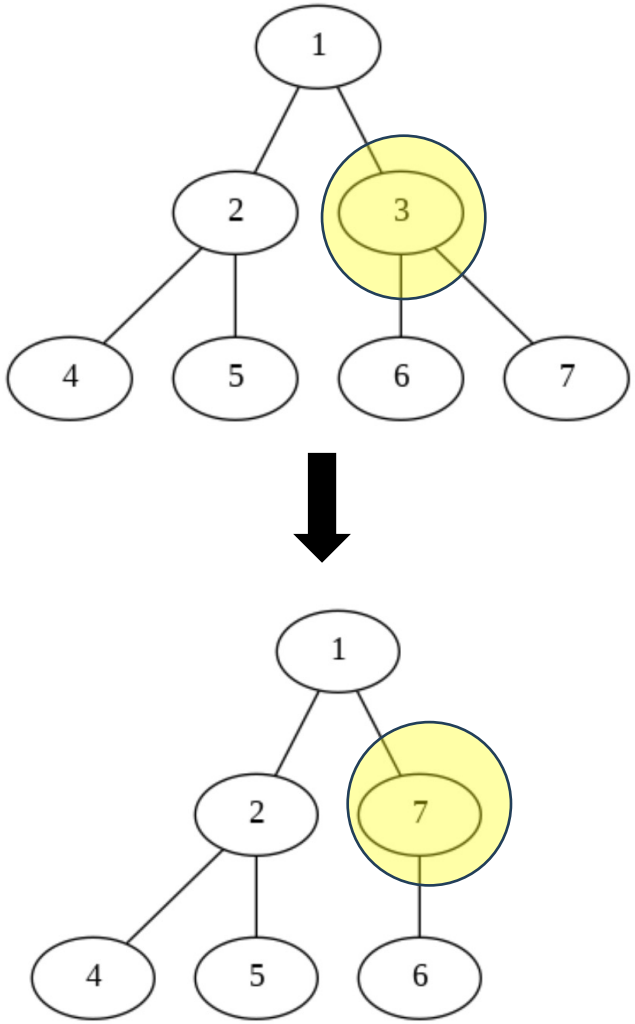
Algorithm to delete a node in a Binary Tree:

- **Find the Node to be Deleted**: Traverse the tree to find the node that needs to be deleted.
- **Replace the Node**: If the node to be deleted is not a leaf node, replace it with the deepest rightmost node in the tree.
- **Delete the Deepest Node**: Remove the deepest rightmost node from the tree.

AI VIET NAM
@aivietnam.edu.vn

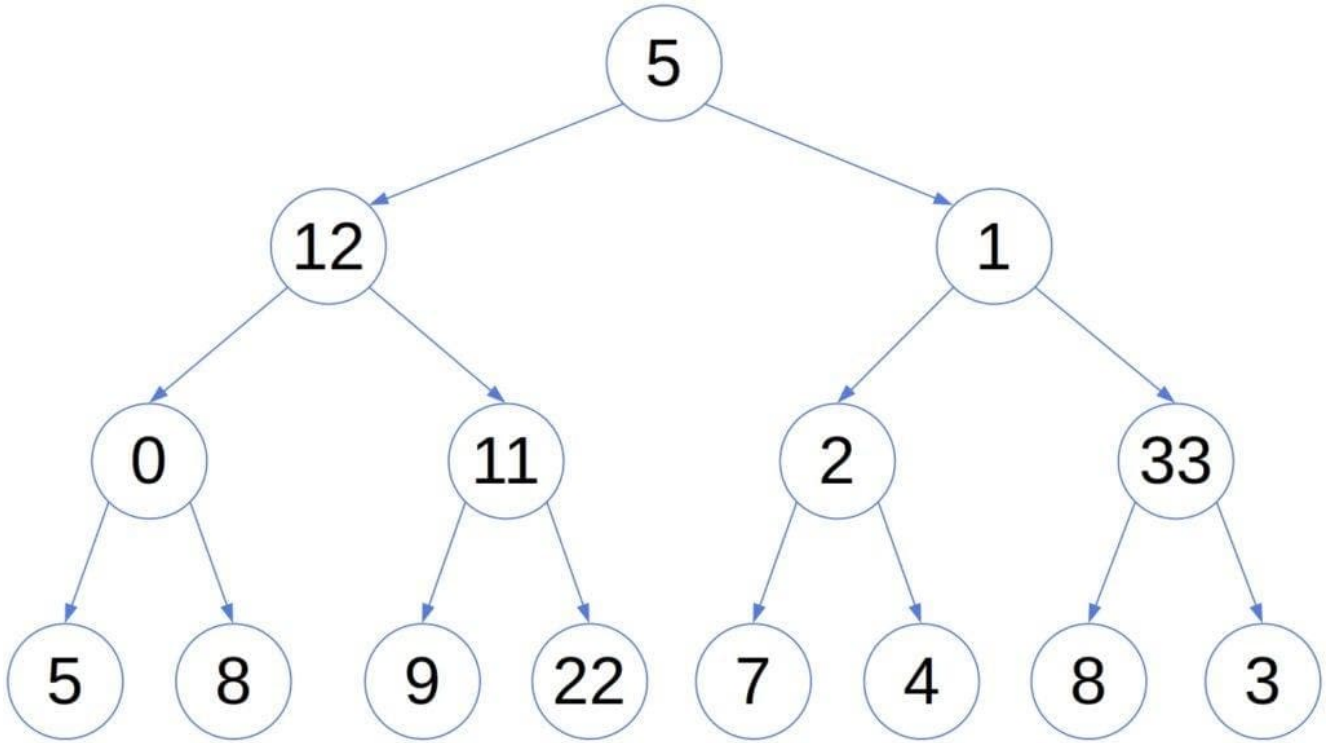
Delete a Node in Binary Tree

Delete node “3” from a Binary Tree




```python
if __name__ == '__main__':
    # Create the root of the tree
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)
    root.right.left = TreeNode(6)
    root.right.right = TreeNode(7)

    dot = visualize_tree(root)
    dot.render('original_delete_tree', format='png', view=True)

    key = 3
    print(f"\nDeleting node with key {key}")
    root = delete_node(root, key)
    dot = visualize_tree(root)
    dot.render('after_delete_tree', format='png', view=True)
```

Binary Tree Limitations



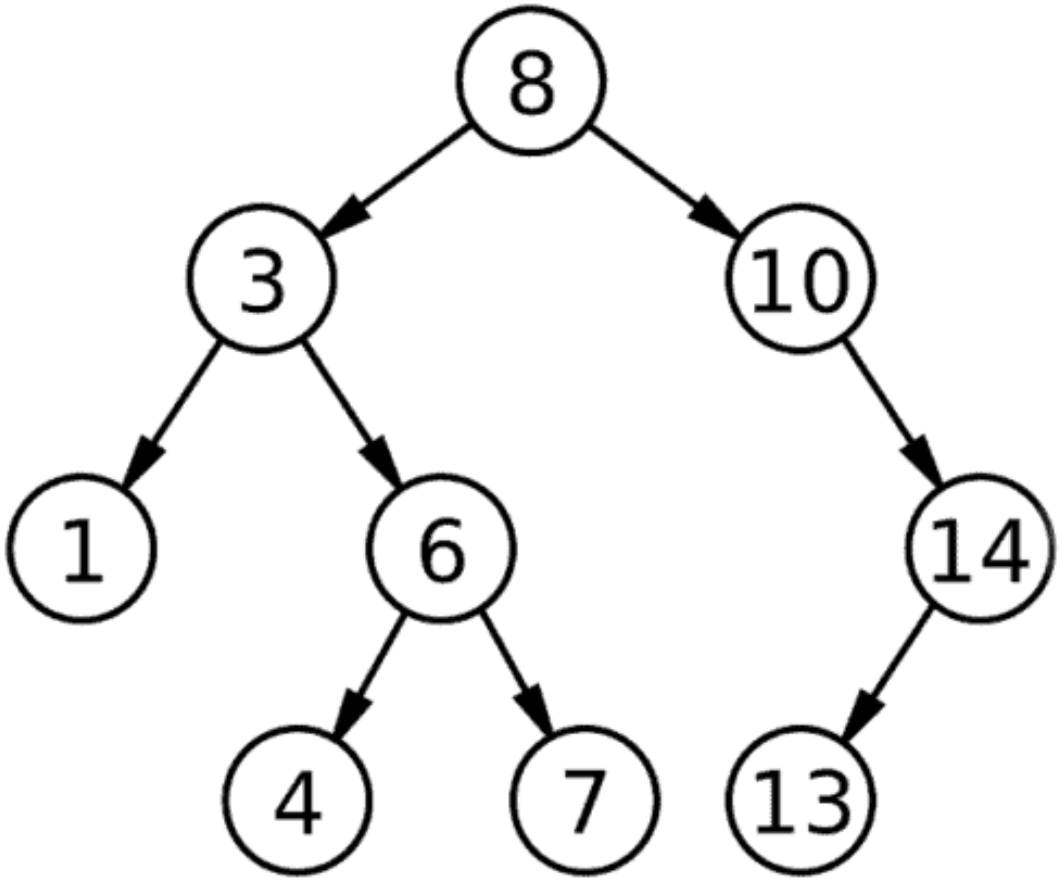
Although suitable for storing hierarchical data, **binary trees of this general form don't guarantee a fast lookup**. Let's take as the example the search for number 9 in the above tree.

Whichever node we visit, we don't know if we should traverse the left or the right sub-tree next. That's because the tree hierarchy doesn't follow the relation

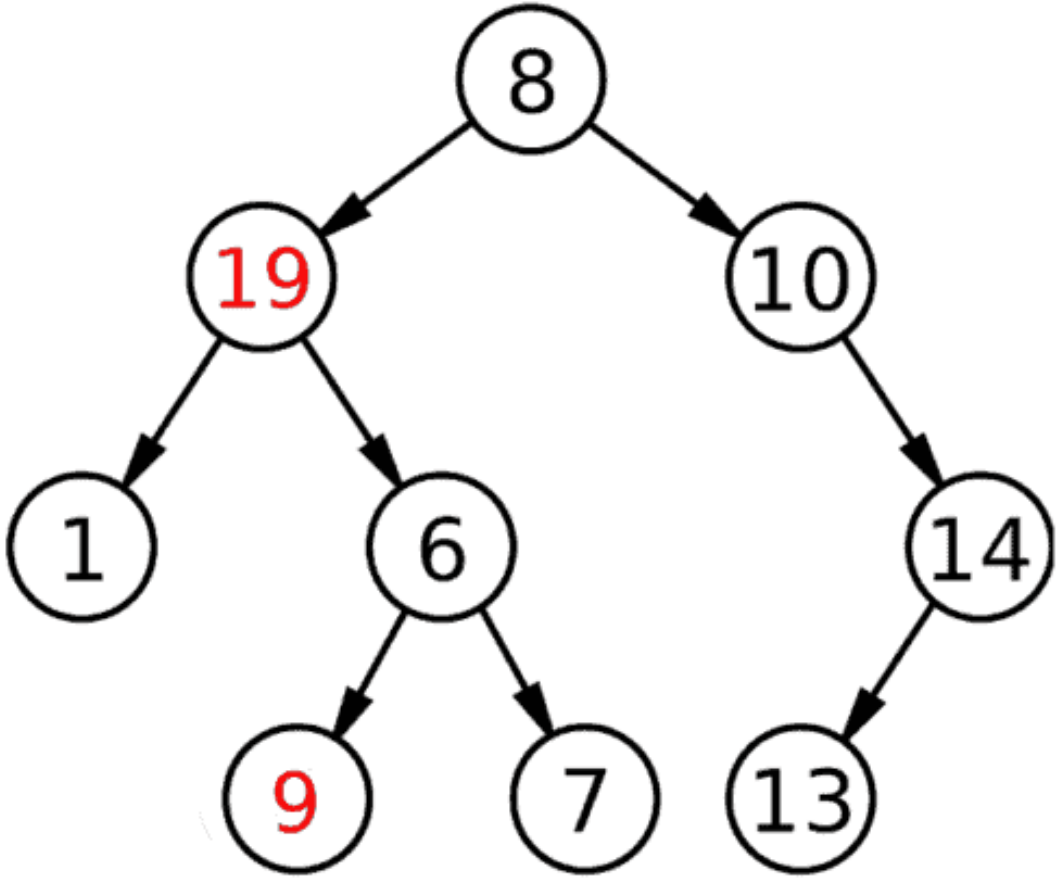
Outline

- What is a Tree Data Structure
- Algorithms on Trees
- Stack
- Queue
- What is a Binary Tree
- What is a Binary Search Tree
- Summary

Binary Search Tree



(a)



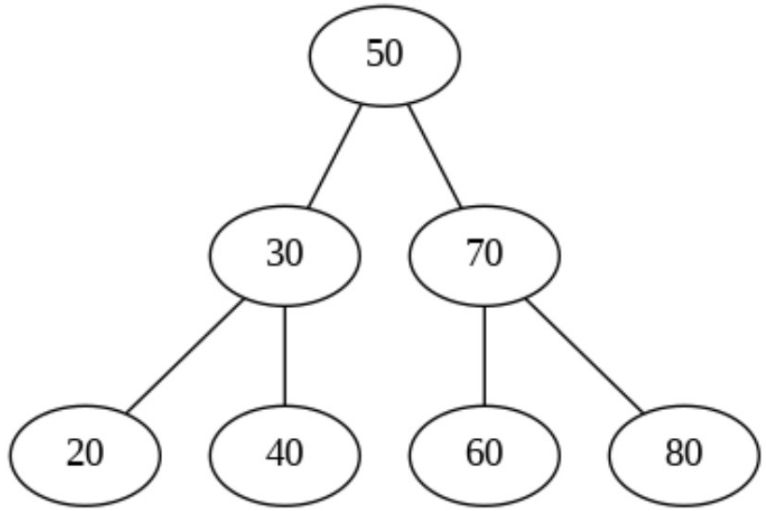
(b)

Which one is the binary search tree? and Why?

Binary Search Tree Implementation

```python
class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert(node.right, key)
```

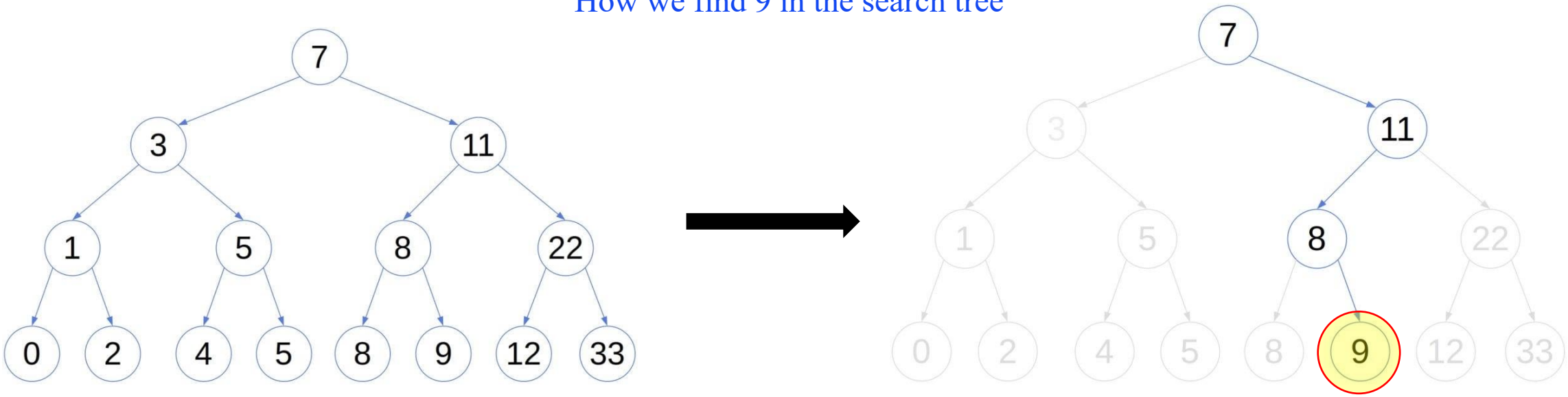
```python
if __name__ == '__main__':
    bst = BST()

    # Insert elements
    elements = [50, 30, 20, 40, 70, 60, 80]
    for element in elements:
        bst.insert(element)
```



AI VIET NAM
@aivietnam.edu.vn

Binary Search Tree

How we find 9 in the search tree



For each node **x** in a BST, all the nodes in the left sub-tree of **x** contain the values that are strictly lower than **x**. Further, all the nodes in the **x**'s right sub-tree are >= **x**

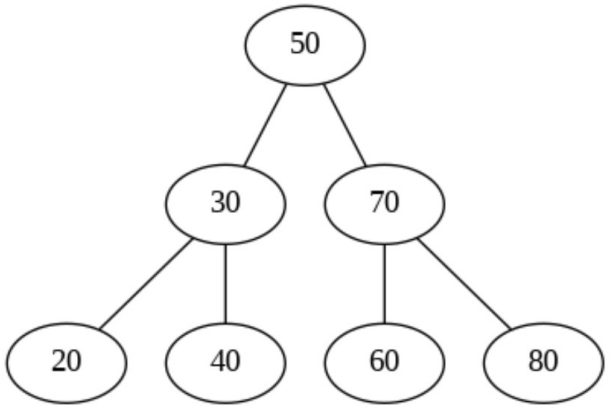
Binary Search Tree

Search algorithm in BST

```
algorithm lookup(tree, key):
    if key = tree.key:
        return true

    if key < tree.key:
        return lookup(tree.leftChild(),key)

    if key > tree.key:
        return lookup(tree.rightChild(), key)
```

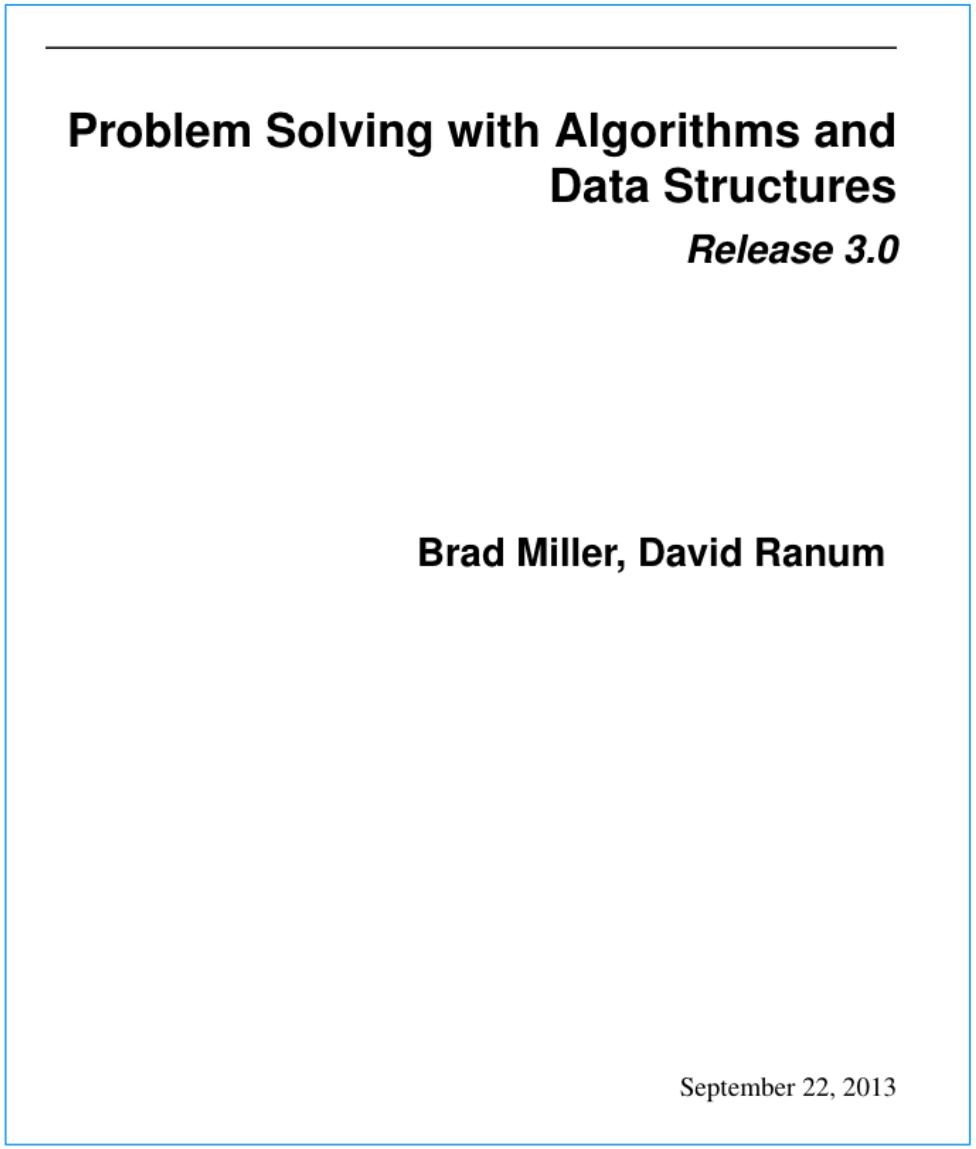
```python
def search(self, key):
    return self._search(self.root, key)

def _search(self, node, key):
    if node is None or node.val == key:
        return node

    if key < node.val:
        return self._search(node.left, key)

    return self._search(node.right, key)
```

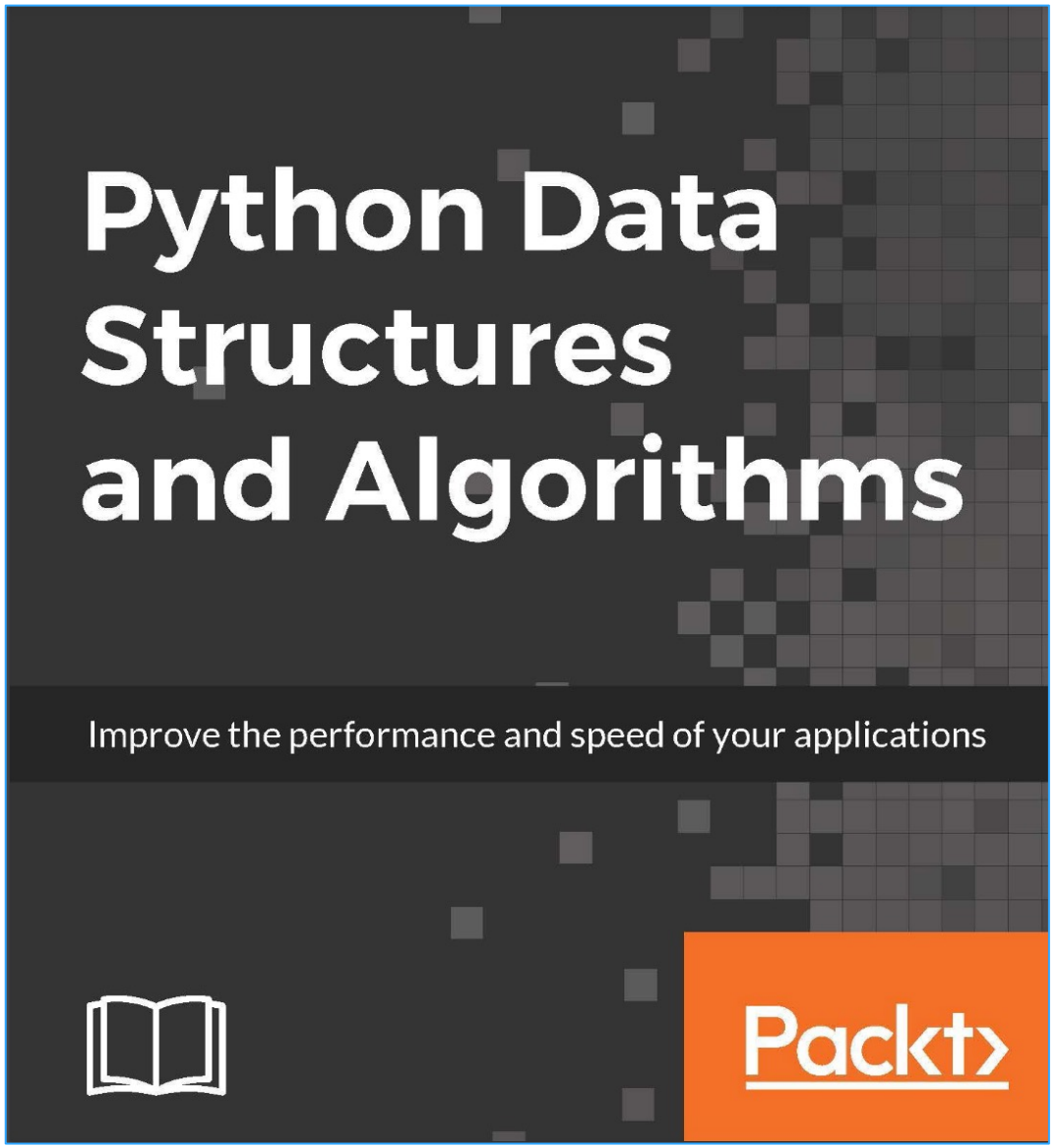
```python
if __name__ == '__main__':
    bst = BST()

    # Insert elements
    elements = [50, 30, 20, 40, 70, 60, 80]
    for element in elements:
        bst.insert(element)


     # Visualize the BST
    dot = visualize_tree(bst.root)
    dot.render('bst', format='png', view=True)

    # Search for elements
    key = 40
    result = bst.search(key)
    if result:
        print(f"Element {key} found in the BST")
    else:
        print(f"Element {key} not found in the BST")
```




Time for
review

References


Problem Solving with Algorithms and Data Structures
Release 3.0
Brad Miller, David Ranum
September 22, 2013


Python Data Structures and Algorithms
Improve the performance and speed of your applications
Packt