INTRODUCTION TO COMPUTER SCIENCE: PROGRAMMING METHODOLOGY

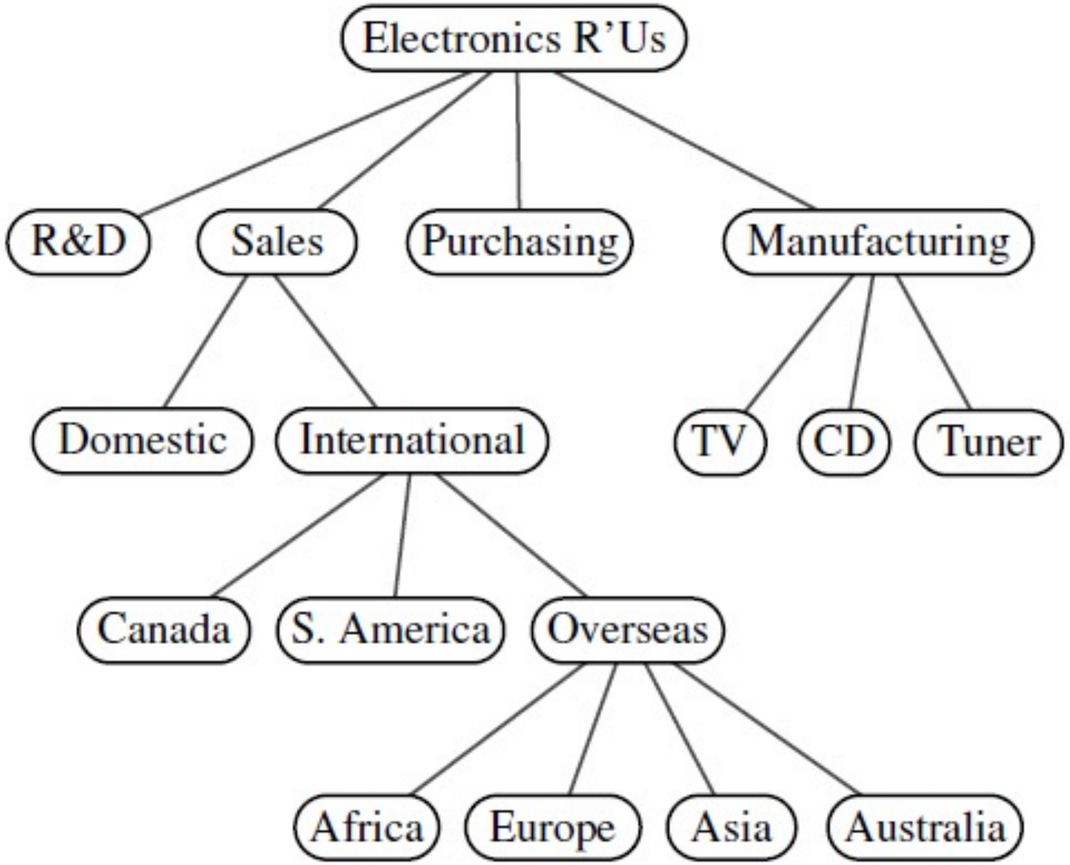
# Lecture 12 Tree

**Prof. Wei Cai**

**School of Science and Engineering**

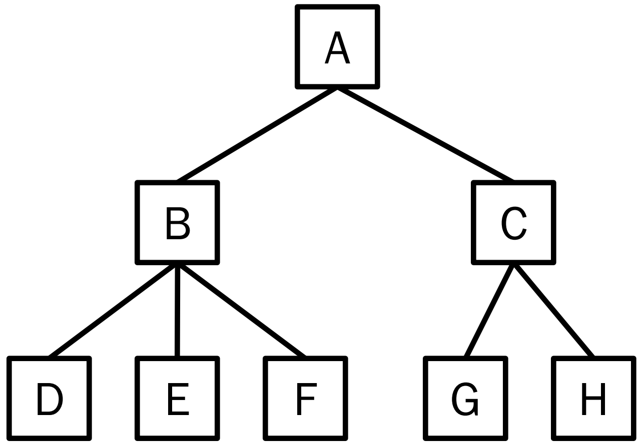
香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

# THE ORGANIZATION OF AN INTERNATIONAL COMPANY



# TREE

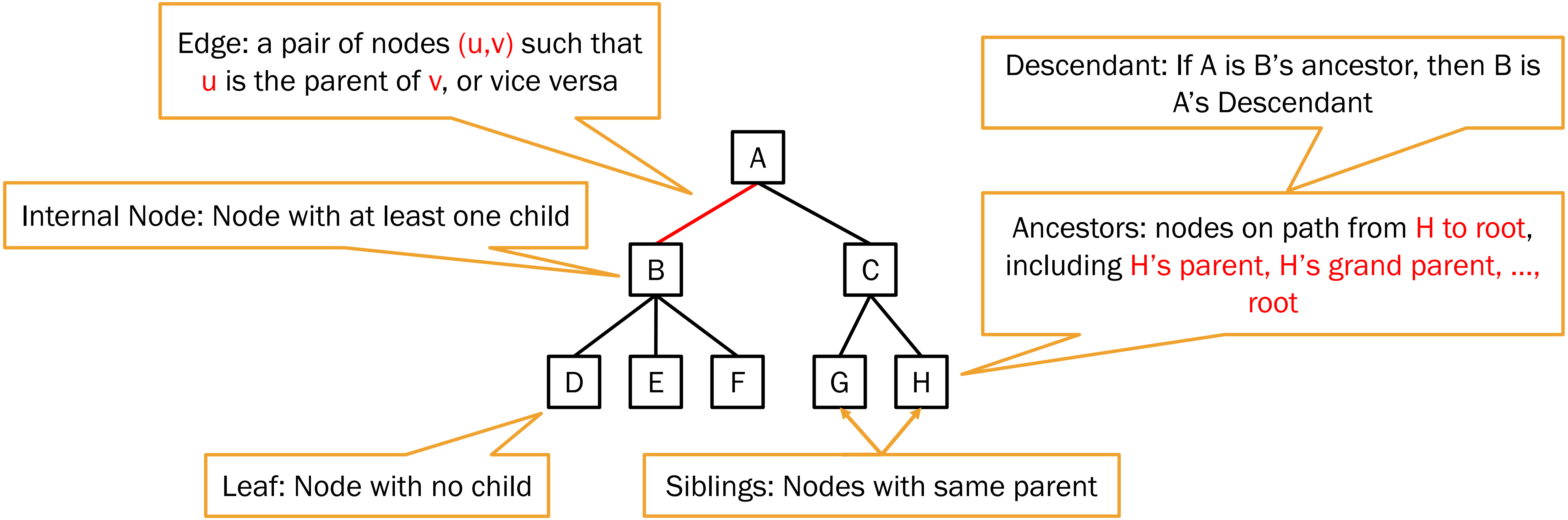
- a data structure that stores elements *hierarchically*
- the root of the tree
  - the top element
  - drawn as the highest
- except the root, each element has
  - a parent element
  - zero or more children elements



# FORMAL DEFINITION OF A TREE

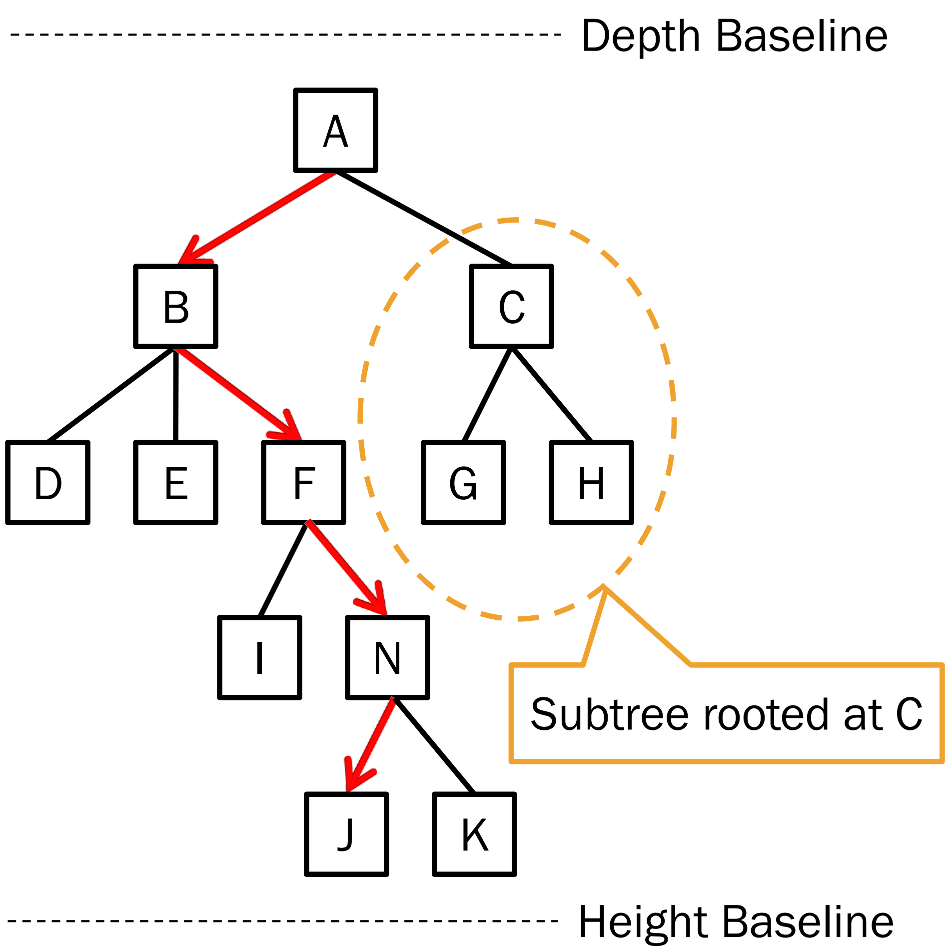
- We define a tree *T* as a set of nodes storing elements such that the nodes have a *parent-child relationship* that satisfies the following properties:
  - If T is non-empty, it has a special node, called the root of T, that has no parent
  - Each node v of T different from the root has a unique parent node w
  - Every node with parent w is a child of w

# TERMS IN TREE DATA STRUCTURE



# TERMS IN TREE DATA STRUCTURE

- A path is a sequence of nodes such that any two consecutive nodes form an edge
  - Length of path: number of edges in path
- Depth of node v
  - Length of path from v to root
  - Depth of root is zero
- Height of node v
  - Length of path from v to its deepest descendant
  - Height of any leaf is zero
  - Height of a tree = Height of the root
- Subtree rooted at n
  - The tree formed by n and its descendants



# FAMILY TREE: IS IT A TREE IN COMPUTER SCIENCE?


Each node v of T different from the root has a unique parent node w

# HOW CAN WE IMPLEMENT A TREE?

?

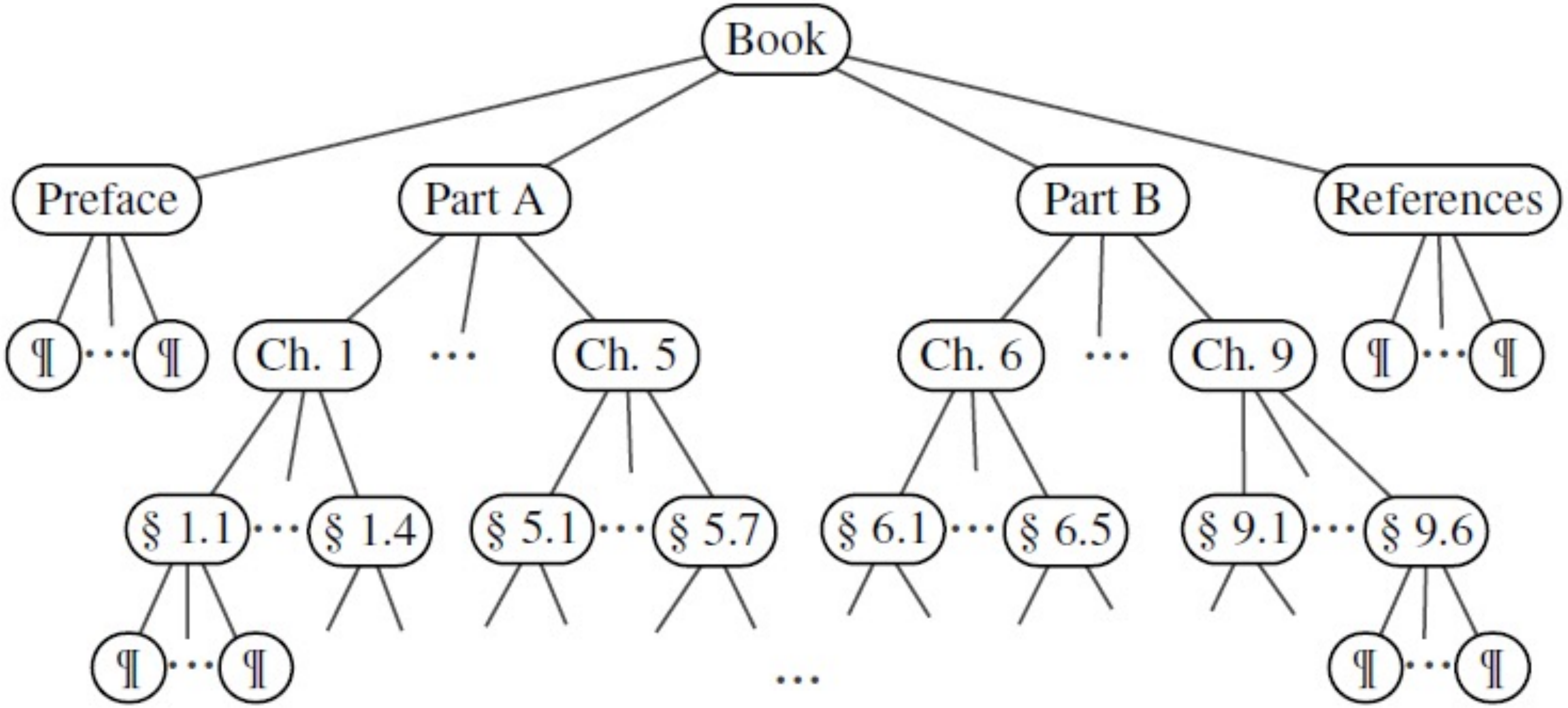
# EXAMPLE OF A TREE

```
class Node:
    def __init__(self, element, parent= None, children=None):
        self.element = element
        self.parent = parent
        self.children = children

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

```
n1 = Node(1)
n2 = Node(2, n1)
n3 = Node(3, n1)
n4 = Node(4, n1)
n1.children.append(n2)
n1.children.append(n3)
n1.children.append(n4)
t = Tree(n1)

print(t.root.element)
print(t.root.children[2].element)
```

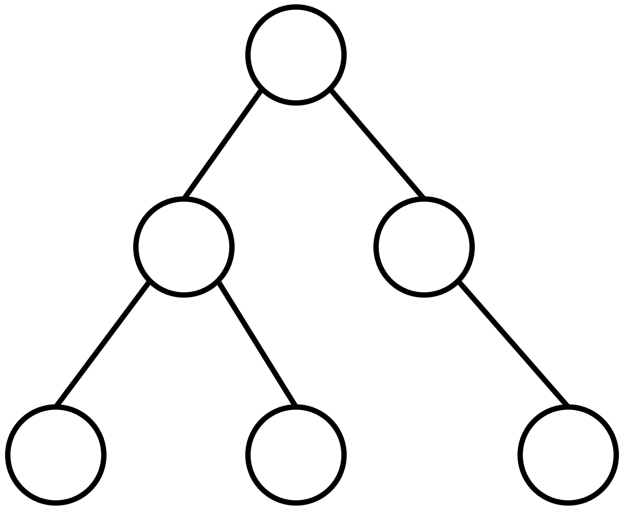
# ORDERED TREE

- A tree is ordered if there is a meaningful linear order among the children of each node; such an order is usually visualized by arranging children from left to right, according to their order



# BINARY TREE

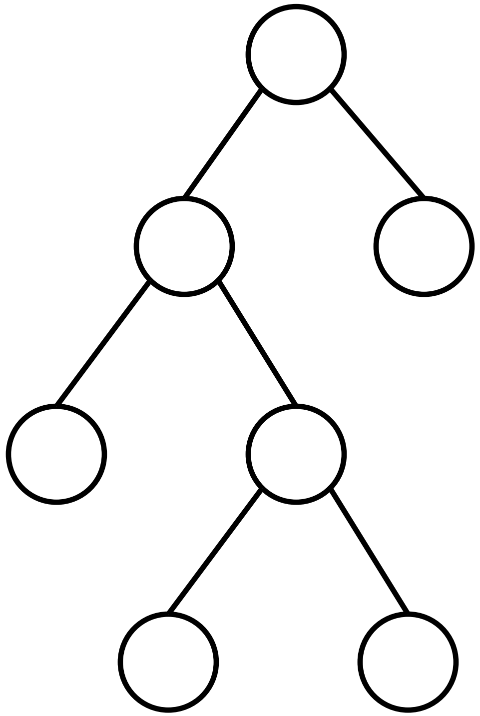
- A binary tree is an ordered tree with the following properties:
  1. Every node has at most two children
  2. Each child node is labelled as being either a left child or a right child
  3. A left child precedes a right child in the order of children of a node



The subtree rooted at a left or right child of an internal node v is called a left subtree or right subtree, respectively, of v

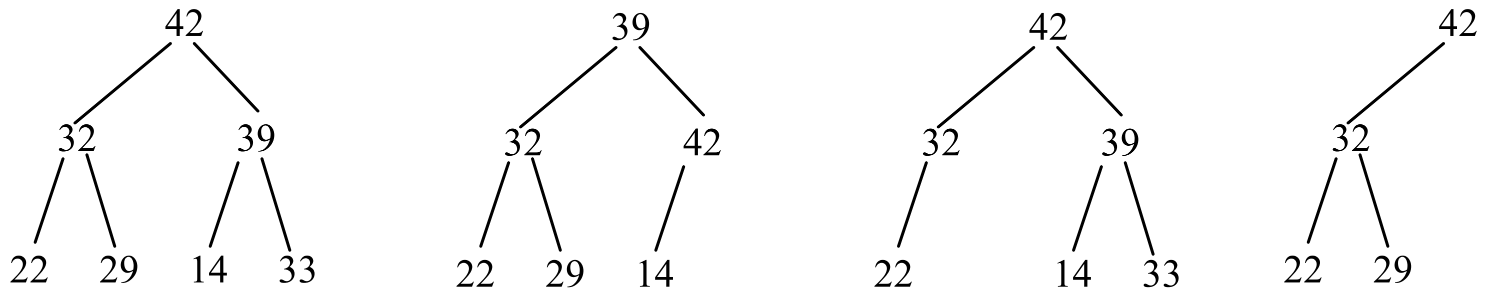
# FULL BINARY TREE

A binary tree is full or proper if each node has either zero or two children.



# COMPLETE BINARY TREE

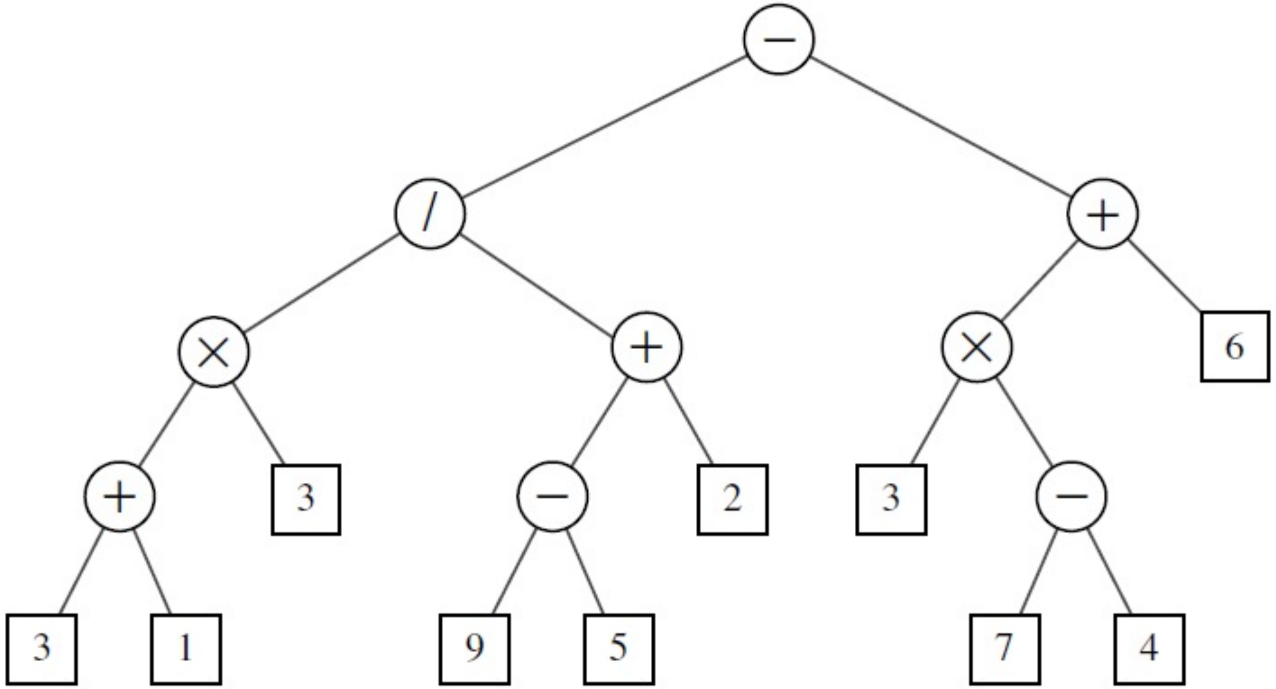
- A binary tree is *complete* if
  - every level of the tree is full
  - except that the last level may not be full and all the leaves on the last level are placed left-most




Complete Complete Not Complete Not Complete

# EXAMPLE: REPRESENT AN EXPRESSION WITH BINARY TREE

- An arithmetic expression can be represented by a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators +, −, ×, and /



## BINARY TREE CLASS

- We define a tree class based on a class called Node; an element is stored as a node

- Each node contains three references, one pointing to the parent node, two pointing to the child nodes

# IMPLEMENTING THE BINARY TREE

```
class Node:

    def __init__(self,element, parent = None,\
        left = None, right = None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right

class LBTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size
```

```
    def find_root(self):
        return self.root

    def parent(self,p):
        return p.parent

    def left(self,p):
        return p.left

    def right(self,p):
        return p.right

    def num_child(self,p):
        count = 0
        if p.left is not None:
            count+=1
        if p.right is not None:
            count+=1
        return count
```

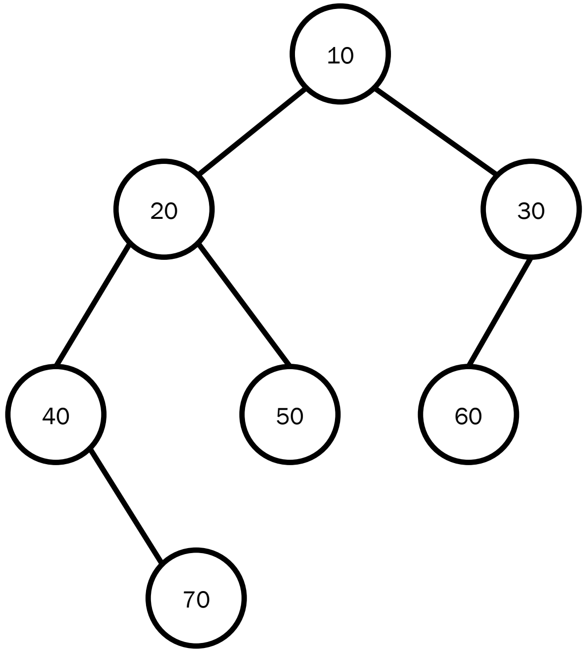
# IMPLEMENTING THE BINARY TREE

```
def add_root(self,e):
    if self.root is not None:
        print('Root already exists.')
        return None
    self.size = 1
    self.root = Node(e)
    return self.root

def add_left(self,p,e):
    if p.left is not None:
        print('Left child already exists.')
        return None
    self.size+=1
    p.left = Node(e,p)
    return p.left
```

```
def add_right(self,p,e):
    if p.right is not None:
        print('Right child already exists.')
        return None
    self.size+=1
    p.right = Node(e,p)
    return p.right

def replace(self,p,e):
    old = p.element
    p.element = e
    return old

def delete(self,p):
    if p.parent.left is p:
        p.parent.left = None
    if p.parent.right is p:
        p.parent.right = None
    return p.element
```

# EXAMPLE: USE THE BINARY TREE CLASS

```python
def main():
    t = LBTree()
    t.add_root(10)
    t.add_left(t.root, 20)
    t.add_right(t.root, 30)
    t.add_left(t.root.left, 40)
    t.add_right(t.root.left, 50)
    t.add_left(t.root.right, 60)
    t.add_right(t.root.left.left, 70)

    print(t.root.element)
    print(t.root.left.element)
    print(t.root.right.element)
    print(t.root.left.right.element)
```

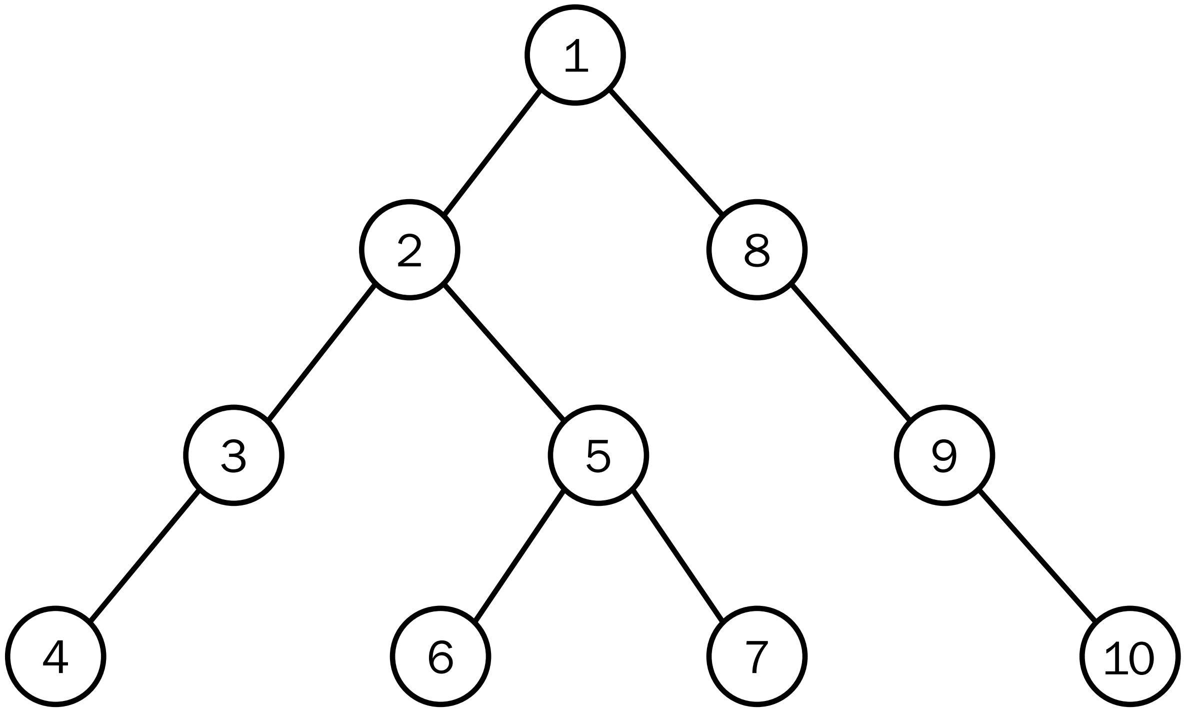



```
>>> main()
10
20
30
50
```

# DEPTH FIRST SEARCH OVER A TREE

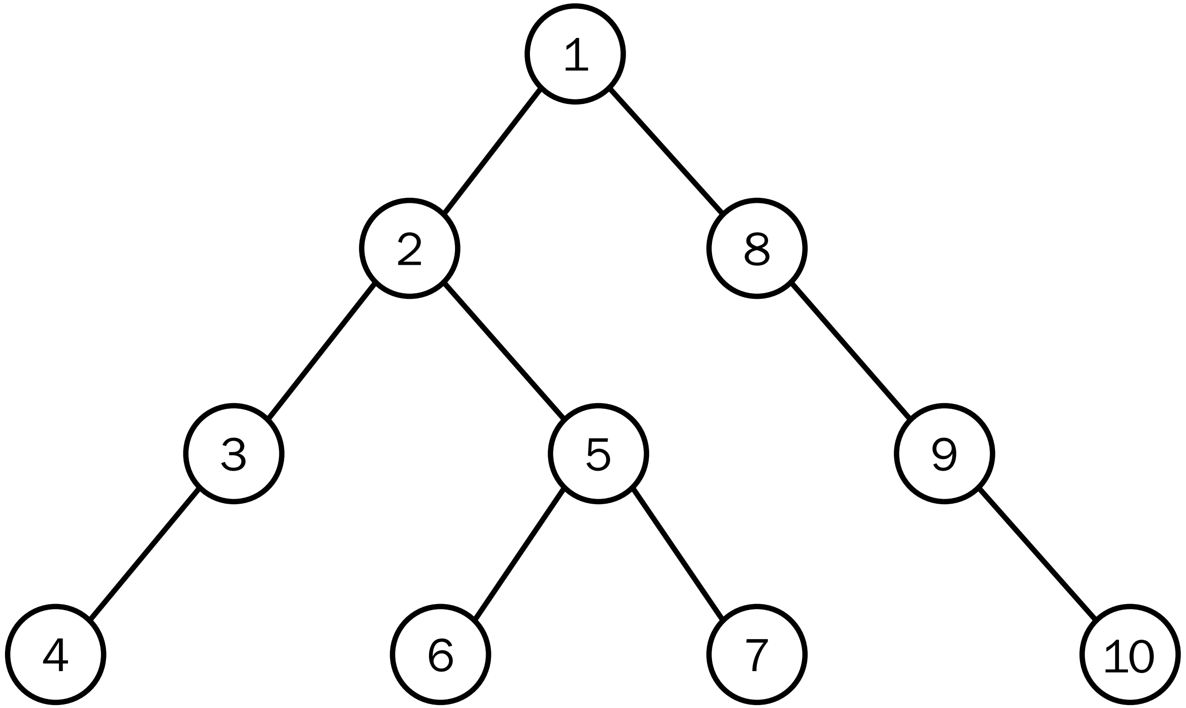
- Depth-first search (DFS) is a fundamental algorithm for traversing or searching tree data structures

- One starts at the root and explores as deep as possible along each branch before backtracking

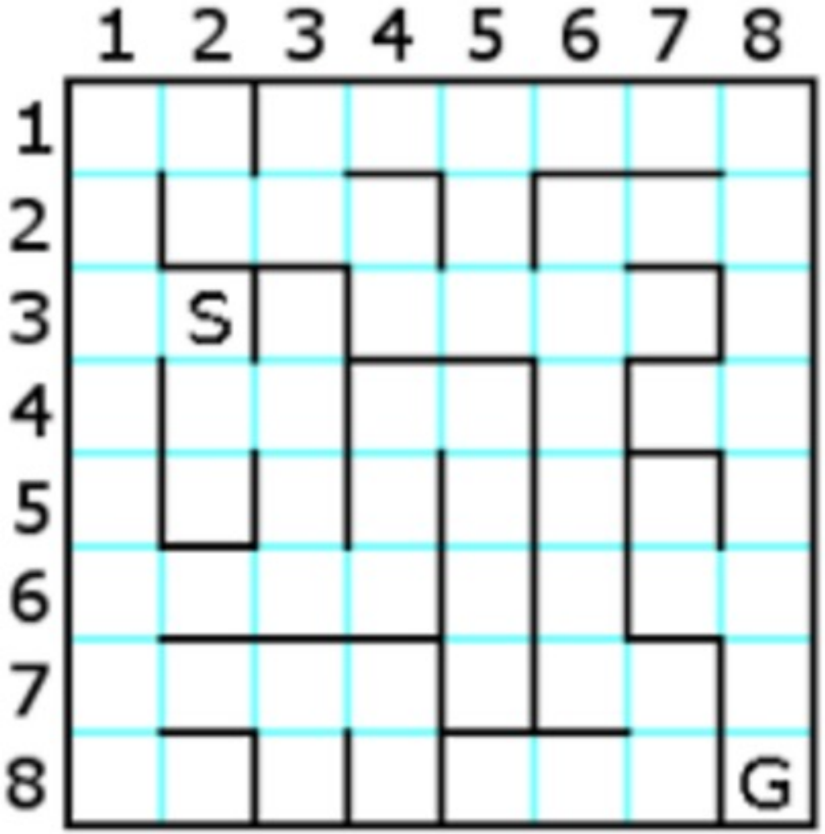


# THE CODE OF DFS OVER A BINARY TREE

```
def DFSearch(t, value):
    # print("===DFSearch for===:", t.element)
    if t.element == value:
        # print("found!")
        return t
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left, value)
        if t.right is not None:
            DFSearch(t.right, value)
```



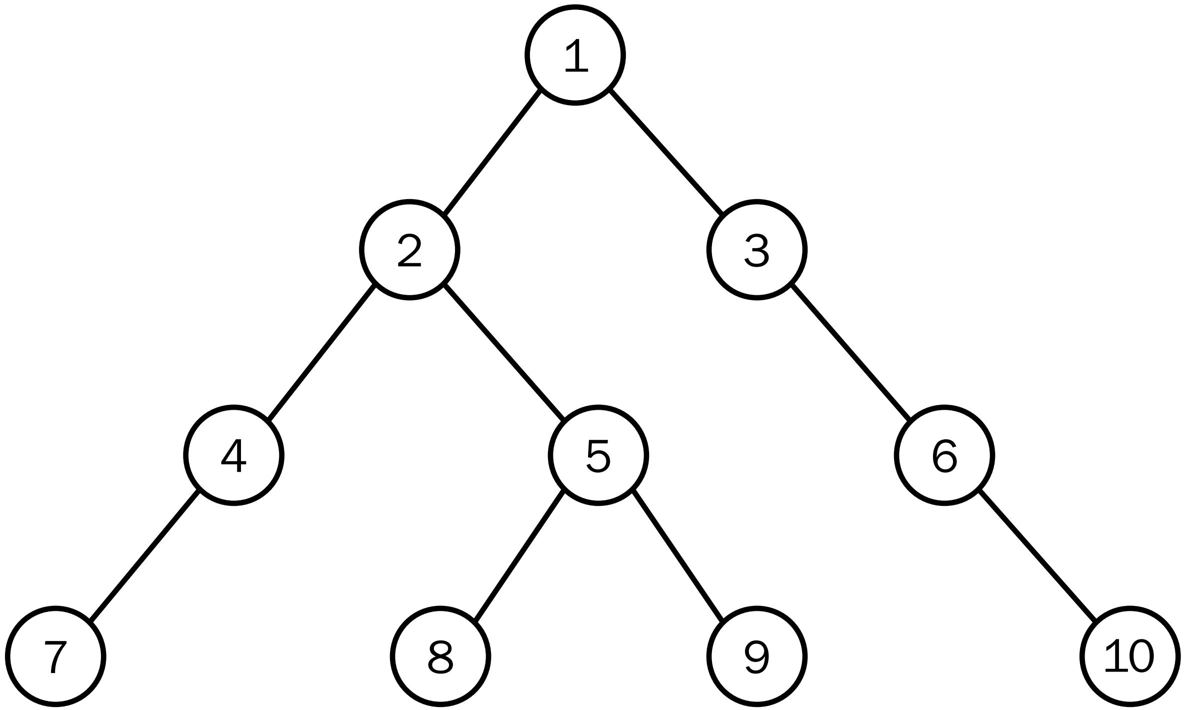
# PROBLEM: SEARCH A PATH IN A MAZE



# BREADTH FIRST SEARCH OVER A TREE

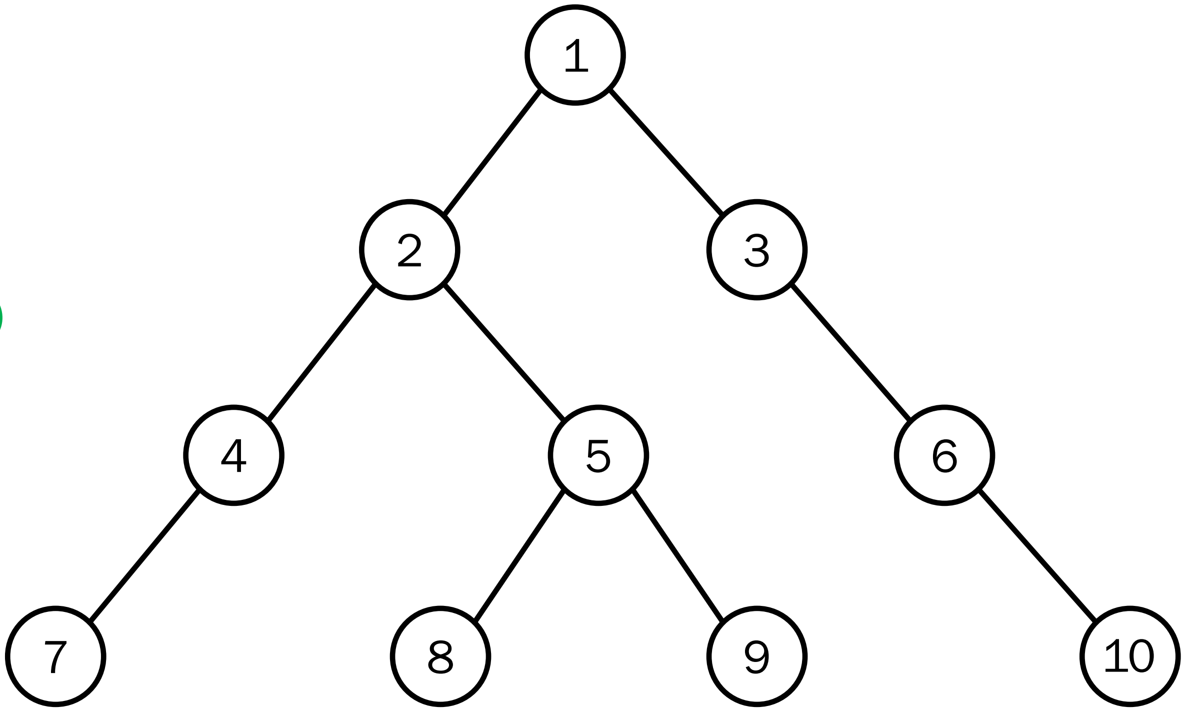
- Breadth-first search (BFS) is another very important algorithm for traversing or searching tree data structures

- Starts at the root and we visit all the positions at depth d before we visit the positions at depth d +1

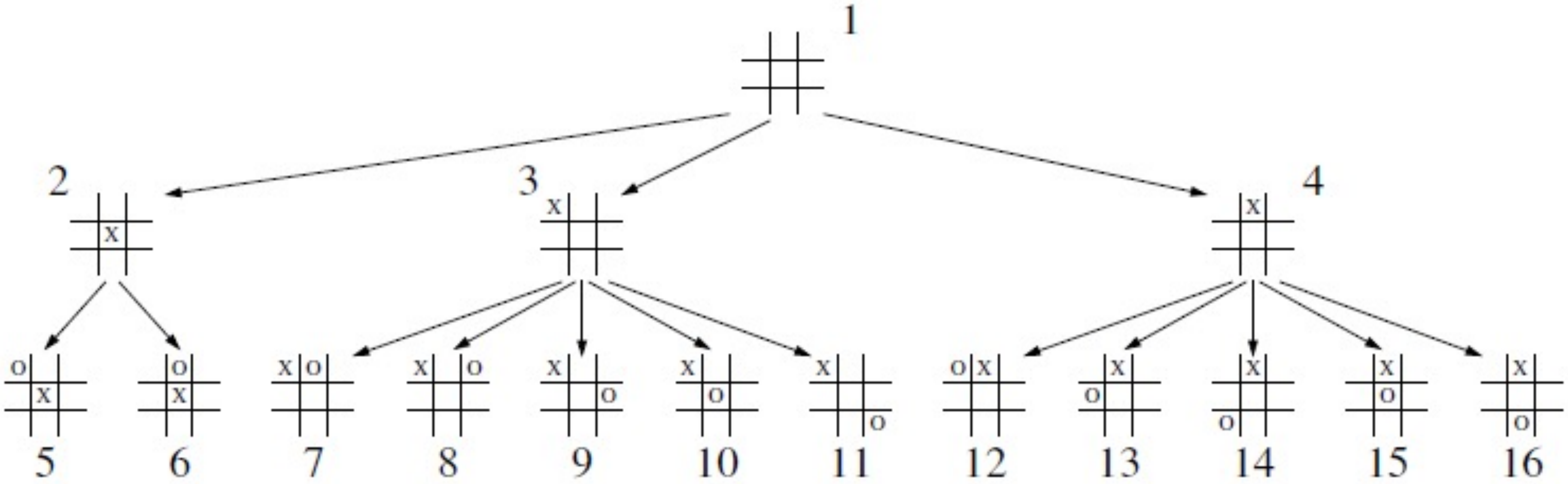


# THE CODE OF BFS OVER A BINARY TREE

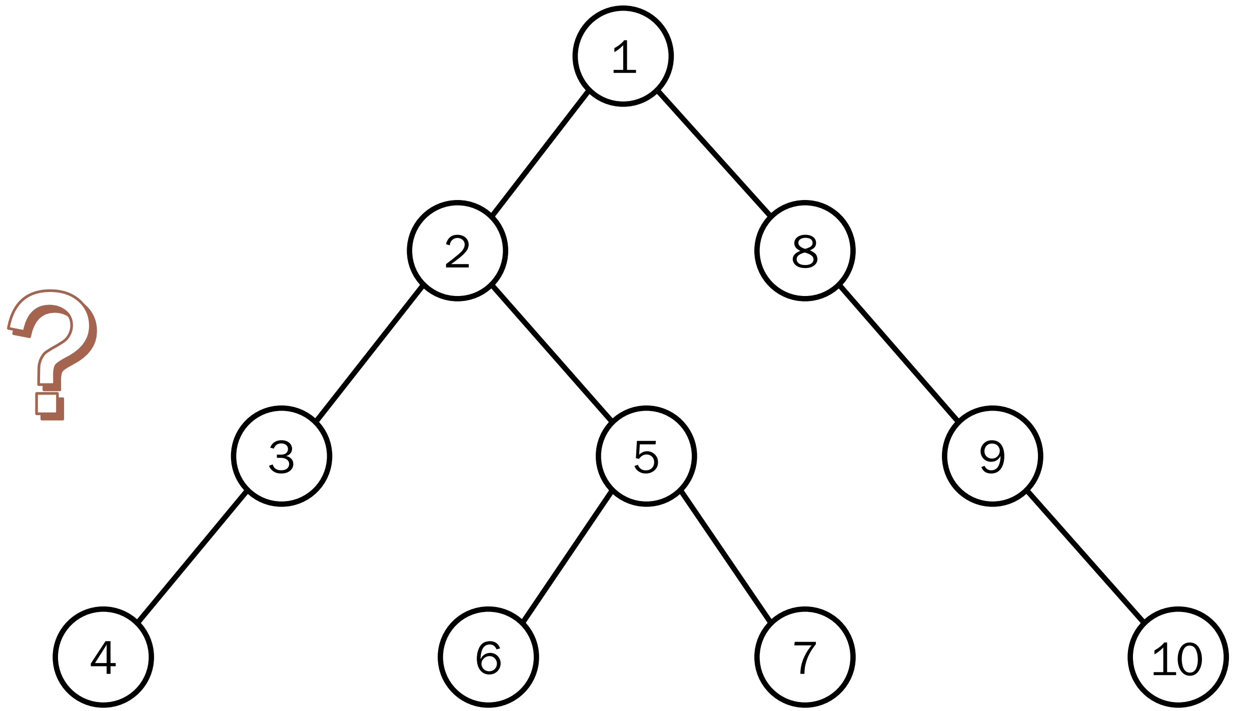
```
def BFSearch(t, value):

    q=ListQueue()
    q.enqueue(t)
    while q.is_empty() is False:
        cNode = q.dequeue()
        # print("===BFSearch for===:", cNode.element)
        if cNode.element==value:
            # print("found!")
            return cNode
        else:
            if cNode.left is not None:
                q.enqueue(cNode.left)
            if cNode.right is not None:
                q.enqueue(cNode.right)
```



# PROBLEM: FINDING THE BEST STRATEGY IN A GAME



# CAN WE IMPLEMENT DFS WITHOUT RECURSION?



# DFS WITHOUT RECURSION

```
def DFSearchStack(t, value):
    s=ListStack()
    s.push(t)
    while s.is_empty() is False:
        cNode = s.pop()
        # print("===DFSearchStack for===:", cNode.element)
        if cNode.element==value:
            # print("found!")
            return cNode
        else:
            if cNode.right is not None:
                s.push(cNode.right)
            if cNode.left is not None:
                s.push(cNode.left)
```