



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

Introduction to Computer Science: Programming Methodology

Lecture 11

Tree



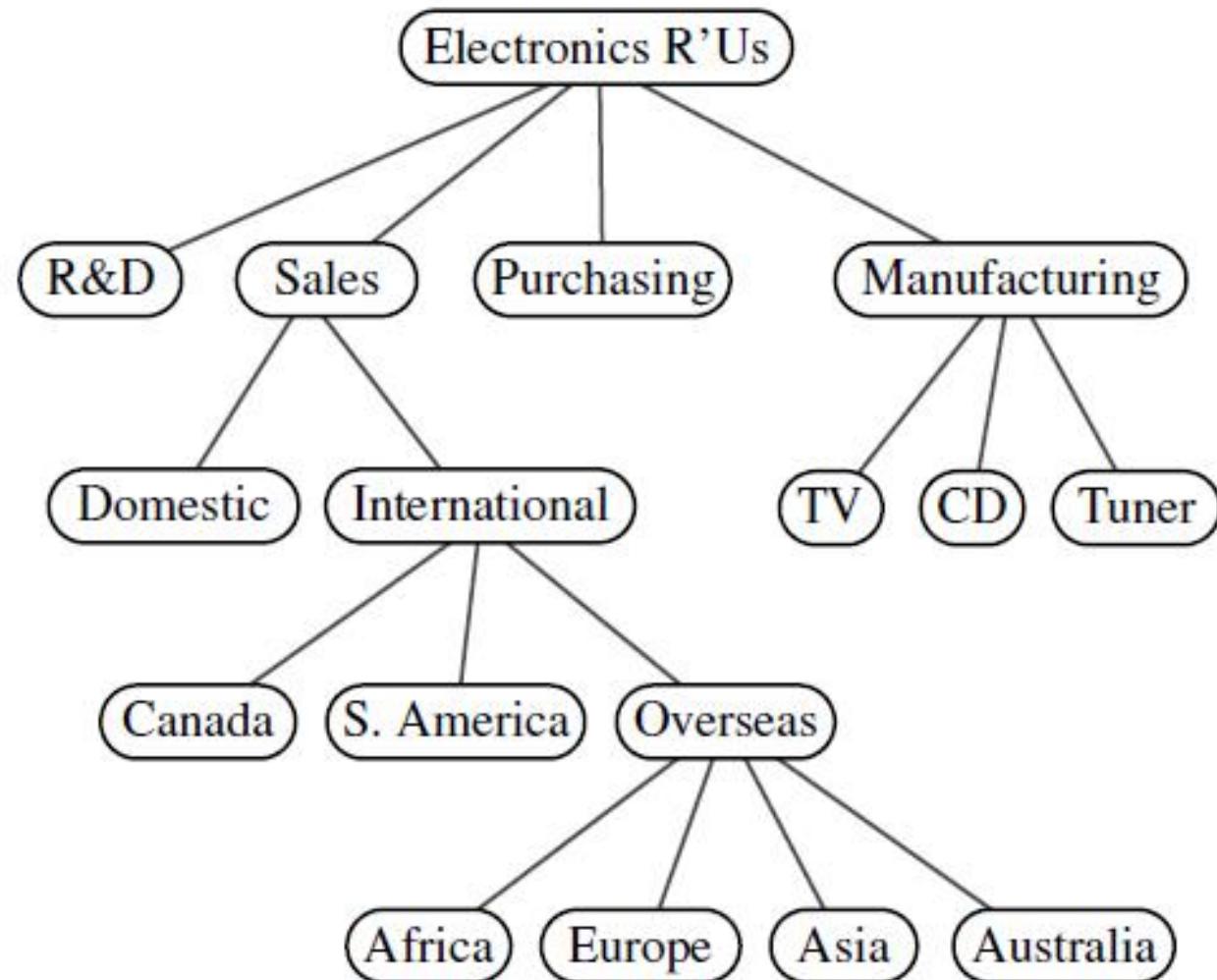
Prof. Junhua Zhao

School of Science and Engineering

Tree

- A **tree** is a data structure that stores elements hierarchically
- With the exception of the top element, each element in a tree has a **parent** element and zero or more **children** elements
- We typically call the top element the **root** of the tree, but it is drawn as the highest element

Example: The organization of a company



Formal definition of a tree

(linked list)

- Formally, 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 nonempty, 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 .

根节点

Edge and path

3b

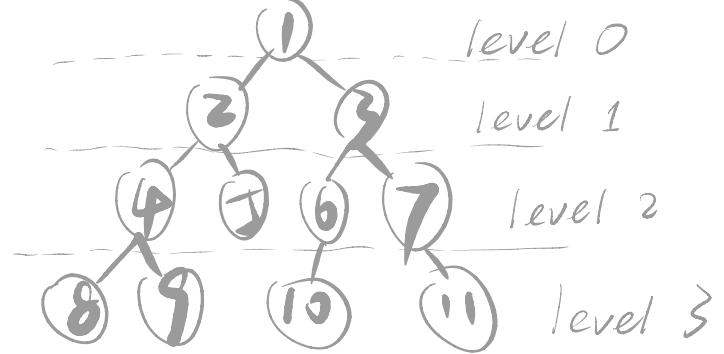
- An **edge** of tree T is a pair of nodes (u,v) such that u is the parent of v , or vice versa



- A **path** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge


 $D=3$

- The **depth** of a node v is the length of the path connecting root node and v



Internal and leaf nodes

叶子节点

(上五图)

eg. 5 8 9 10 11

- A node is called a **leaf node** if it has no child

内部节点

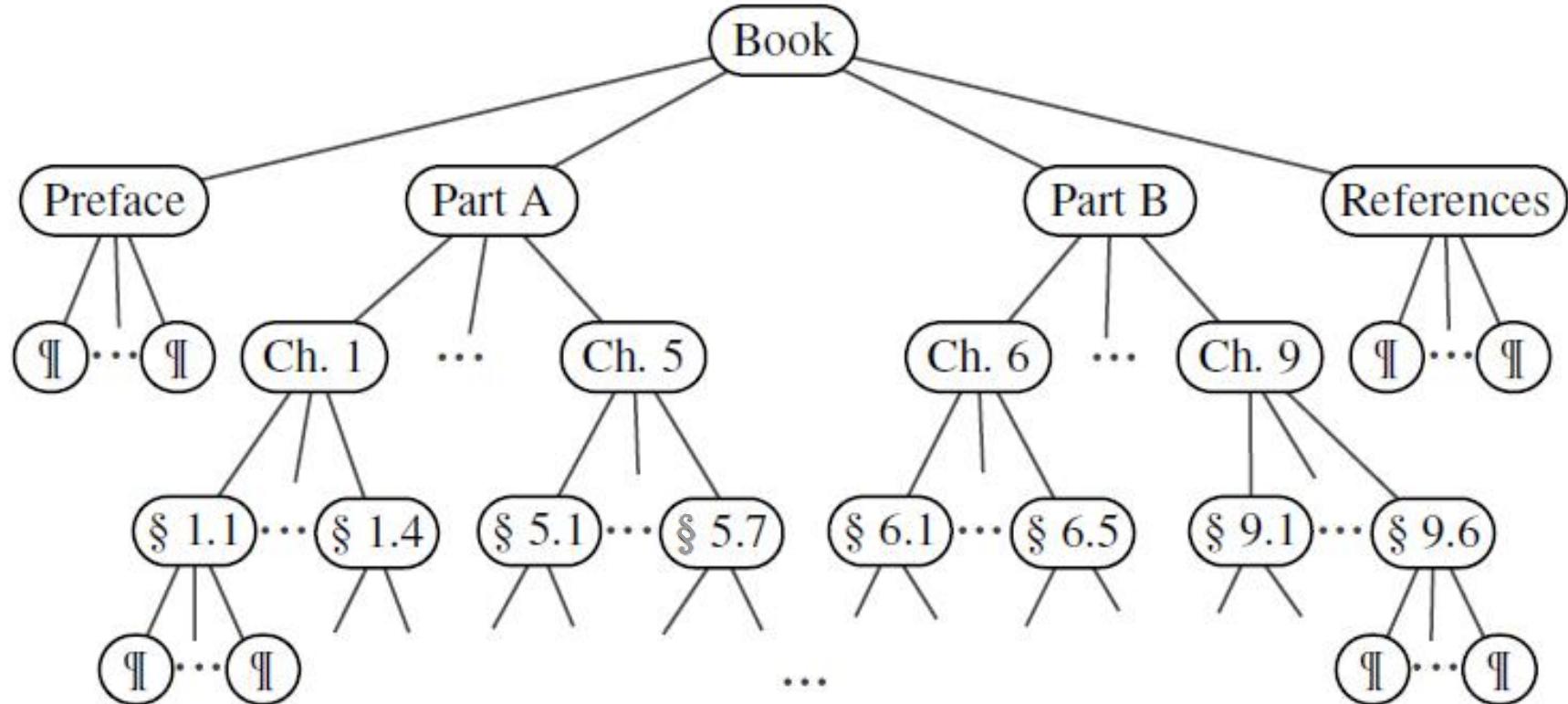
- If a node has at least one child, it is an **internal node**

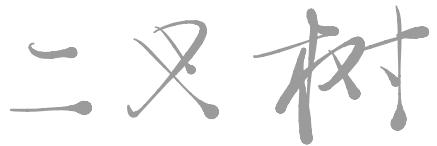
eg. 1, 2, 3, 4, 5, 7

Ordered tree

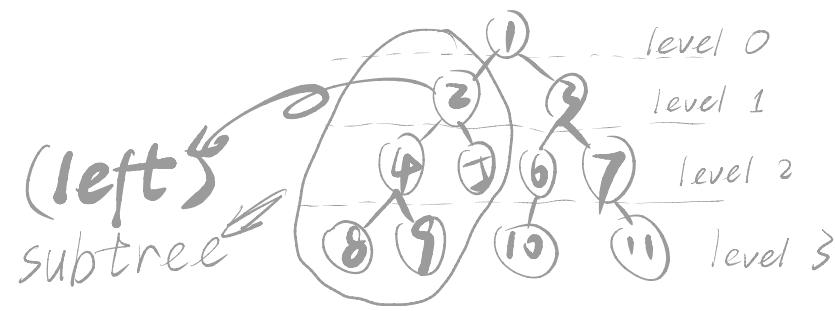
有序树

- 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 labeled 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

左 / 右 子 树

子 树

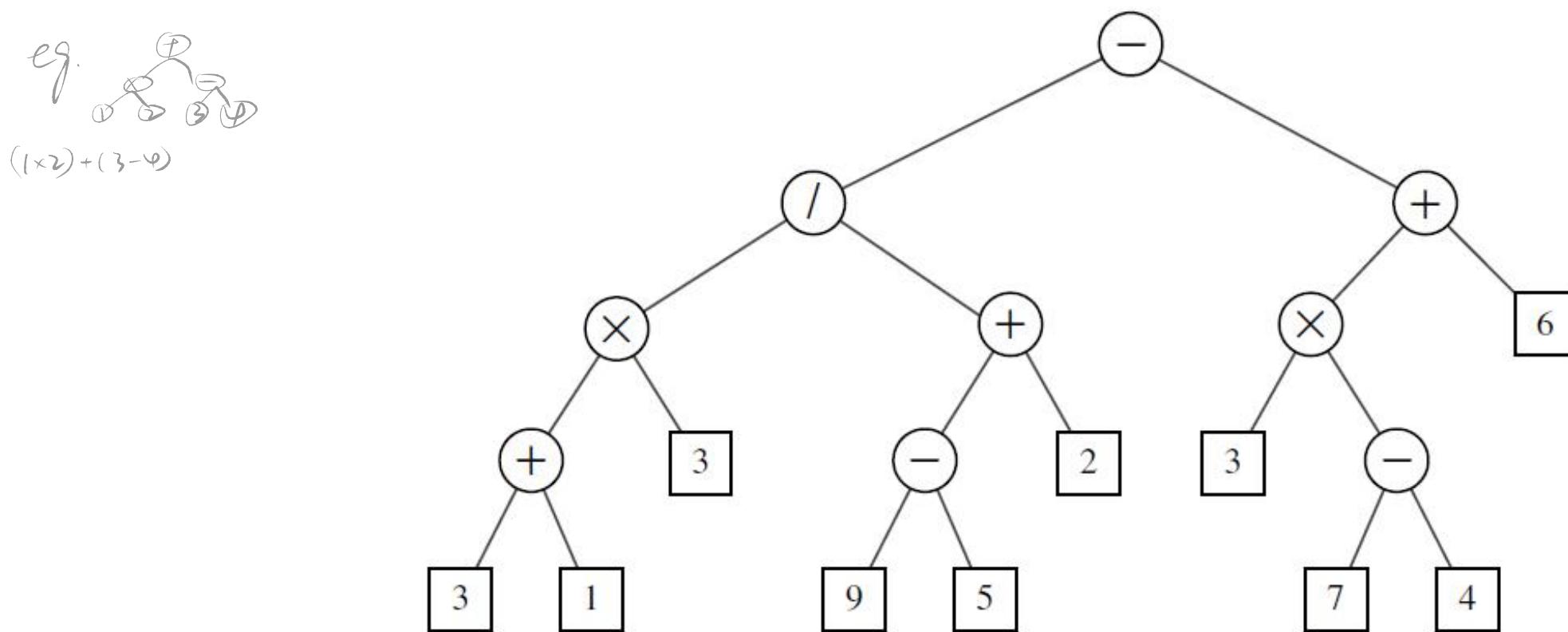
The subtree rooted at a left or right child of an internal node v is called a **left subtree** or **right subtree**, respectively, of v

A binary tree is proper if each node has either zero or two children. Some people also refer to such trees as being full binary trees

滿 二 叉 树

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

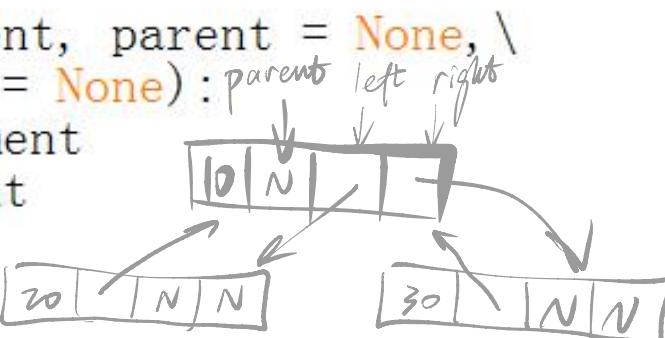
Implementing the binary tree

私有方法没有名字意义

公有时其实可以重写的方法

```
class Node:
```

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

```

$\ell = LBTree$

$\ell.add_root(1)$

$\ell.add_left(1, 2)$

return None
~~self.size+=1~~
 p.left = Node(e, p)
 return p.left



~~l.root = φ~~
~~l.size = 0~~
 l.root = 40
 l.size = 1

```
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
```

```
l.add_root(10) def delete(self, p):  
l.add_left(l.root, 20) if p.parent.left is p:  
l.add_right(l.root, 30)     p.parent.left = None  
l.add_left(l.root.left, 40) if p.parent.right is p:  
a=l.delete(l.root.left)    p.parent.right = None  
                           return p.element
```

Example: Use the binary tree class

```
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
```

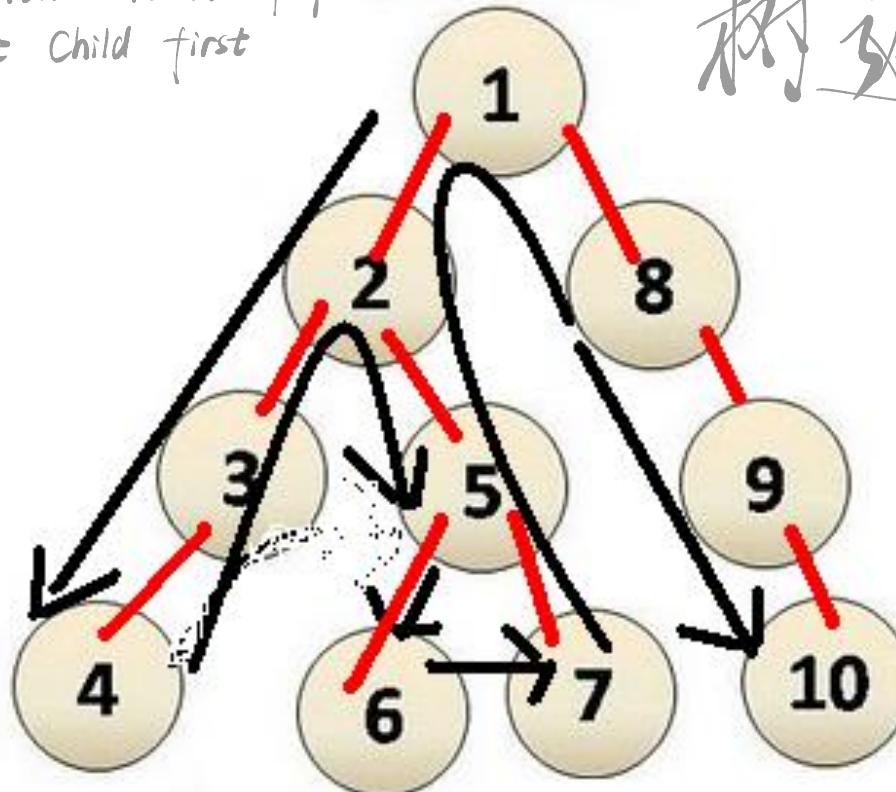
Depth first search over a tree

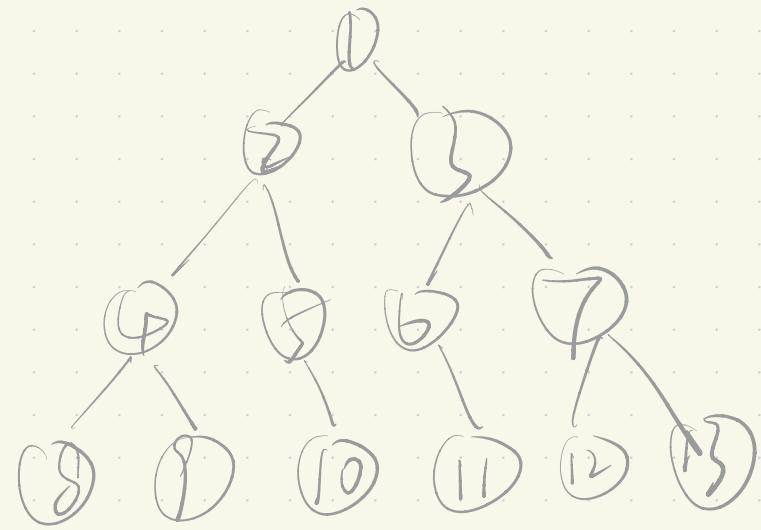
深度优先搜索

- 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**

1. visit the root
2. visit the lower level if possible
3. visit left child first

Tree traversing
树遍历
方法





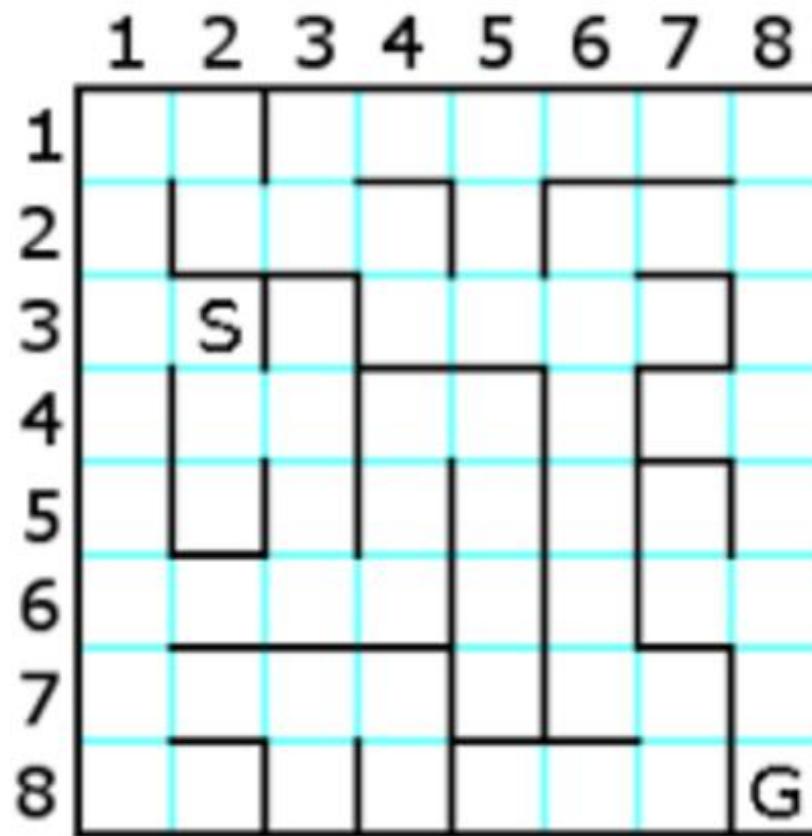
0 → 2 → 4 → 3 → 8 → 9

→ 5 → 10 → 11 → 6 → 12 →

7 → 13 → 14 → 1



Example: search a path in a maze



Recursion

方法:

1. visit the node
2. perform DFS on the left subtree
3. perform DFS on the right subtree

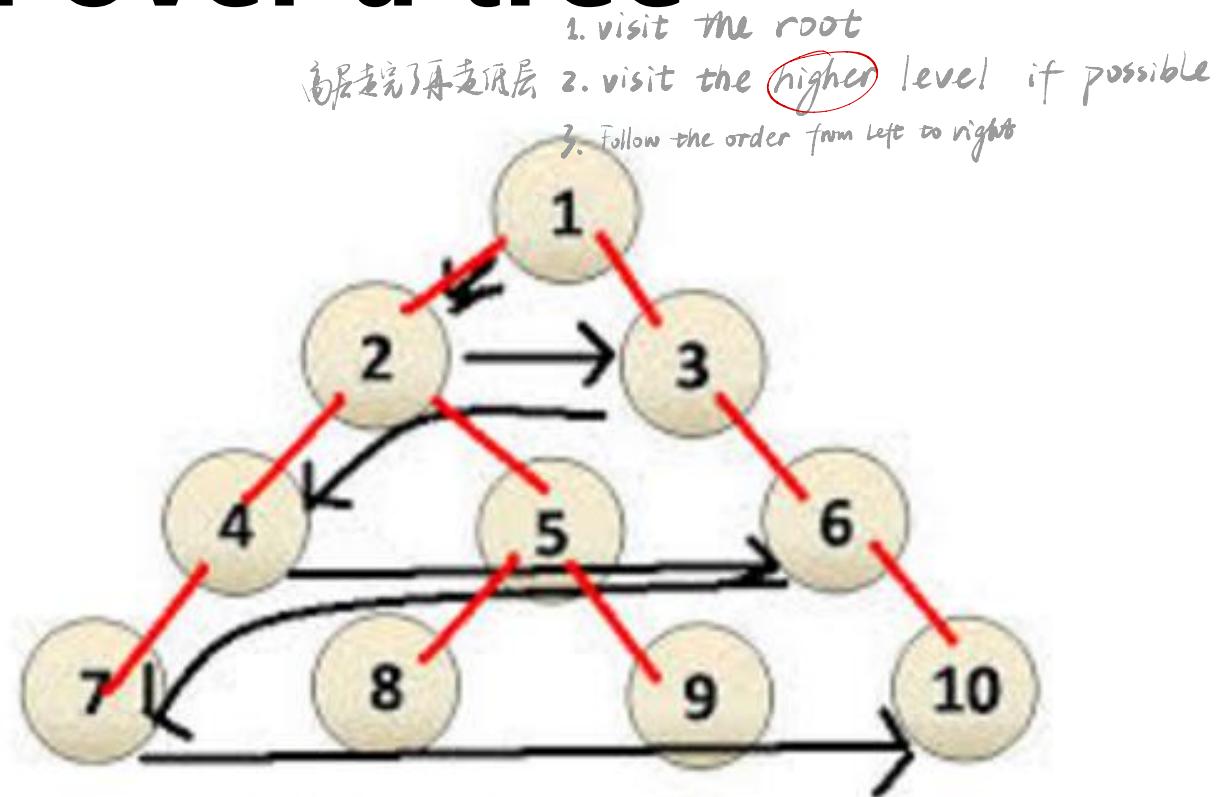
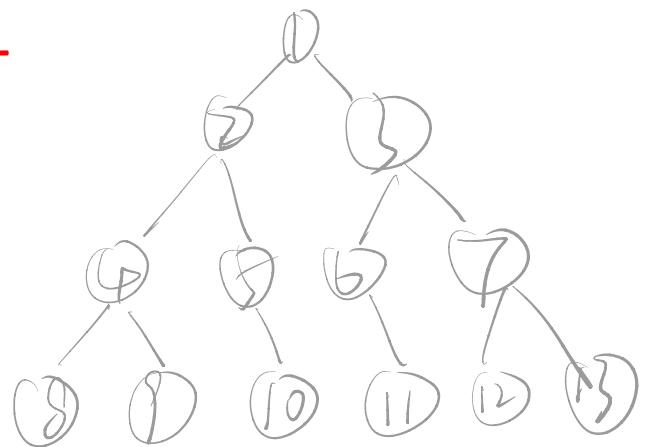
The code of DFS over a binary tree

```
pointer, pointing to the root of the tree
def DFSearch(t):
    if t:
        print(t.element)
    if (t.left is None) and (t.right is None):
        return
    else:
        if t.left is not None:
            DFSearch(t.left)
        if t.right is not None:
            DFSearch(t.right)
```

Breadth first search over a tree

广度优先搜索

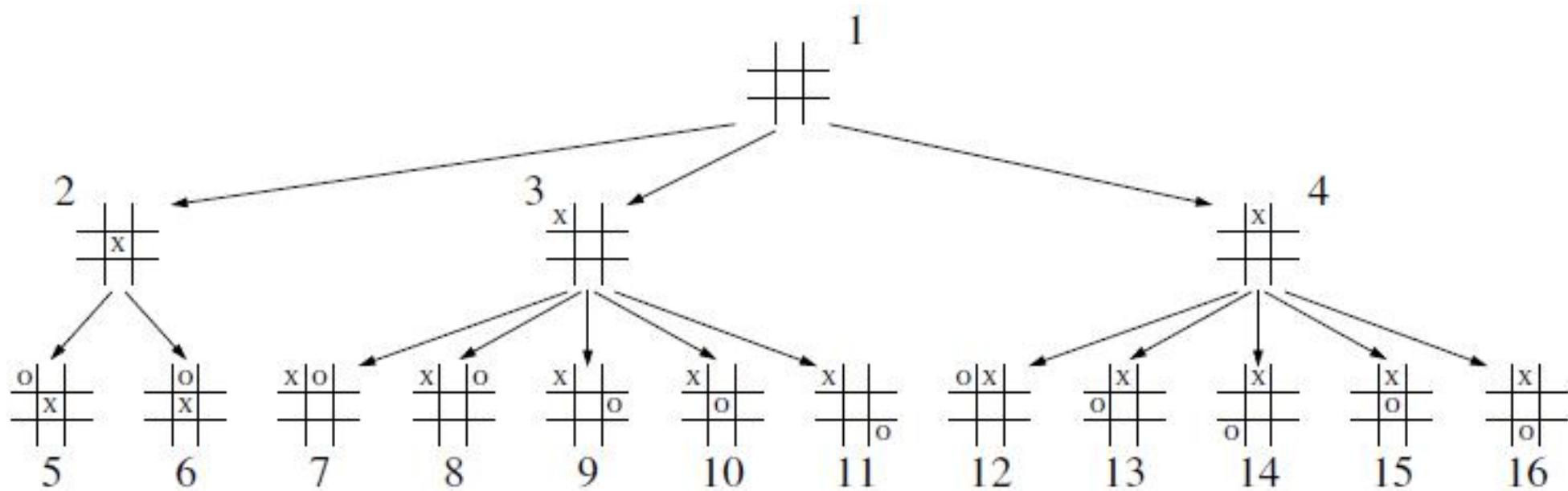
- Breadth-first search (BFS) is another very important algorithm for traversing or searching tree data structures
- Starts at the **root** and we visit all the positions at depth **d** before we visit the positions at depth **$d + 1$**



① → ② → ③ → ④ → ⑤ → ⑥ → ⑦

→ ...

Example: finding the best strategy in a game



The code of BFS over a binary tree

1. Enqueue the root
2. Dequeue a node.
enqueue its children,
visit this node
3. If the queue is not empty -
go to step 2

```
def BFSearch(t):  
    q = ListQueue()  
    q.enqueue(t)  
  
    while q.is_empty() is False:  
        cNode = q.dequeue()  
        if cNode.left is not None:  
            q.enqueue(cNode.left)  
        if cNode.right is not None:  
            q.enqueue(cNode.right)  
        print(cNode.element)
```

