Day01回顾

- **数据结构、算法、程序**

```
【1】数据结构：解决问题时使用何种数据类型，数据到底如何保存，只是静态描述了数据元素之间的关系
【2】算法：    解决问题的方法，为了解决实际问题而设计的，数据结构是算法需要处理的问题载体
【3】程序：    数据结构 + 算法
```

- **数据结构分类**

```
【1】线性结构 ： 多个数据元素的有序集合
    1.1) 顺序存储 - 线性表
        a> 定义：将数据结构中各元素按照其逻辑顺序存放于存储器一片连续的存储空间中
        b> 示例：顺序表、列表
        c> 特点：内存连续，溢出时开辟新的连续内存空间进行数据搬迁并存储

    1.2) 链式存储 - 线性表
        a> 定义：将数据结构中各元素分布到存储器的不同点，用记录下一个结点位置的方式建立联系
        b> 示例：单链表、单向循环链表
        c> 特点：
            单链表：内存不连续，每个节点保存指向下一个节点的指针，尾节点指针指向"None"
            单向循环链表：内存不连续，每个节点保存指向下一个节点指针，尾节点指针指向"头节点"
    1.3) 栈 - 线性表
        a> 后进先出 - LIFO
        b> 栈顶进行入栈、出栈操作，栈底不进行任何操作
        c> 顺序存储实现栈、链式存储实现栈
    1.4) 队列 - 线性表
        a> 先进先出 - FIFO
        b> 队尾进行入队操作、队头进行出队操作
        c> 顺序存储实现队列、链式存储实现队列
```

- **算法效率衡量-时间复杂度T(n)**

```
【1】定义：算法执行步骤的数量

【2】分类
    2.1) 最优时间复杂度
    2.2) 最坏时间复杂度 - 平时所说
    2.3) 平均时间复杂度

【3】时间复杂度大O表示法 T(n) = O(??)
    去掉执行步骤的系数、常数、低次幂

【4】常见时间复杂度排序
    O(1)<O(logn)<O(n)<O(nlogn)<O(n2)<O(n2logn)<O(n3)
```

Day02笔记

作业讲解

链表作业题一

- **题目描述 + 试题解析**

```
【1】题目描述
    输入一个链表，输出该链表中倒数第 k 个节点

【2】试题解析
    可将链表中的每一个元素保存到列表中，在列表中寻找倒数第 k 个元素
```

- **代码实现**

```
"""
输入一个链表，输出该链表中倒数第 k 个节点
思路:
    1、链表只能从头到尾遍历,从尾到头遍历存在难度
    2、从头到尾遍历,将节点数据添加到一个列表中
    3、利用列表的下标索引取出对应的节点数据
"""
class Node:
    """节点类"""
    def __init__(self, value):
        self.value = value
        self.next = None

class Solution:
    def get_k_node(self, head, k):
        # 1.把链表中节点数据添加到列表中
        li = []
        cur = head
        while cur:
            li.append(cur.value)
            cur = cur.next
        # 2.利用列表的索引取出对应值
        if k > len(li):
            raise IndexError('list index out of range')

        return li[-k]

if __name__ == '__main__':
    s = Solution()
    # 创建链表: 100 -> 200 -> 300 -> None
    head = Node(100)
    head.next = Node(200)
    head.next.next = Node(300)
    # 终端1: 200
    print(s.get_k_node(head, 2))
```

```
    # 终端2: list index out of range
    print(s.get_k_node(head, 8))
```

链表作业题二

- **题目描述 + 试题解析**

```
【1】题目描述
    输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则

【2】试题解析
    a> 比较两个链表的头节点，确认合成后链表的头节点
    b> 继续依次比较两个链表元素的大小，将元素小的结点插入到新的链表中，直到一个链 表为空
```

- **代码实现**

```
"""
输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则
思路:
    1、程序最终返回的是：合并后的链表的头节点
    2、先确定新链表的头节点
    3、互相比较,移动值小的游标
"""
class Node:
    """节点类"""
    def __init__(self, value):
        self.value = value
        self.next = None

class Solution:
    def merge_two_link_list(self, head1, head2):
        # 1.确定新链表的头节点
        h1 = head1
        h2 = head2
        if h1 and h2:
            if h1.value >= h2.value:
                merge_head = h2
                h2 = h2.next
            else:
                merge_head = h1
                h1 = h1.next
            # p即为最终返回的结果
            p = merge_head
        elif h1:
            return h1
        else:
            return h2
        # 2.遍历两个链表进行比较合并
        while h1 and h2:
            if h1.value <= h2.value:
                merge_head.next = h1
                h1 = h1.next
```

```
            else:
                merge_head.next = h2
                h2 = h2.next
            # 移动新链表的游标
            merge_head = merge_head.next

        # 3.循环结束后,一定有一个游标为None(或者说一定有一个链表遍历完了)
        if h2:
            merge_head.next = h2
        elif h1:
            merge_head.next = h1

        # 4.最终返回新链表的头节点
        return p

if __name__ == '__main__':
    s = Solution()
    # 链表1: 100 -> 200 -> 300 -> 400 -> None
    head1 = Node(100)
    head1.next = Node(200)
    head1.next.next = Node(300)
    head1.next.next.next = Node(400)
    # 链表2: 1 -> 200 -> 600 -> 800 -> None
    head2 = Node(1)
    head2.next = Node(200)
    head2.next.next = Node(600)
    head2.next.next.next = Node(800)
    # 合并
    p = s.merge_two_link_list(head1, head2)
    # 结果: 1 100 200 200 300 400 600 800
    while p:
        print(p.value, end=' ')
        p = p.next
```

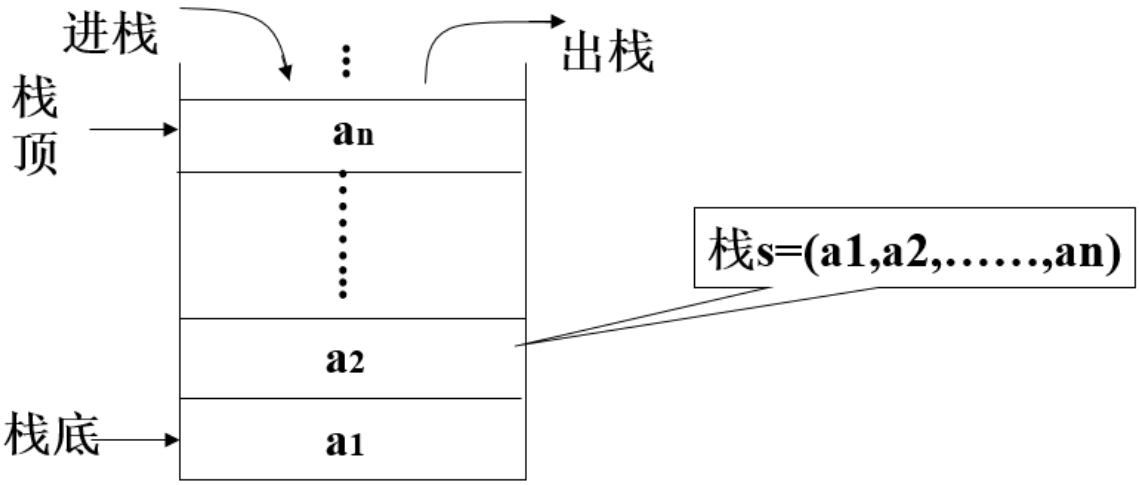
线性表 - 栈（LIFO）

- **定义**

```
栈是限制在一端进行插入操作和删除操作的线性表（俗称堆栈），允许进行操作的一端称为"栈顶"，另一固定端称为"栈底"，当栈中没有元素时称为"空栈"
```

- **特点**

```
【1】栈只能在一端进行数据操作
【2】栈模型具有后进先出的规律（LIFO）
```



- **顺序栈代码实现**

```
"""
顺序存储的方式实现栈
思路:
    1、栈 : LIFO 后进先出
    2、设计
        列表尾部作为栈顶（入栈、出栈操作）
        列表头部作为栈底（不进行任何操作）
"""
class Stack:
    def __init__(self):
        """初始化一个空栈"""
        self.elems = []

    def is_empty(self):
        """判断栈是否为空栈"""
        return self.elems == []

    def push(self, item):
        """入栈：相当于在链表尾部添加1个元素"""
        self.elems.append(item)

    def destack(self):
        """出栈：相当于在列表尾部弹出1个元素"""
        if self.is_empty():
            raise Exception('destack from empty stack')
        return self.elems.pop()

if __name__ == '__main__':
    s = Stack()
    # 栈(栈底->栈顶): 100 200 300
    s.push(100)
    s.push(200)
    s.push(300)
    # 终端1: 300 200 100 异常
    print(s.destack())
    print(s.destack())
    print(s.destack())
    print(s.destack())
```

- **链式栈代码实现**

```python
"""
链式存储方式实现栈
思路:
    1、栈: LIFO 后进先出
    2、设计
        链表头部作为栈顶（入栈、出栈操作）
        链表尾部作为栈底（不进行任何操作）
"""
class Node:
    """节点类"""
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkListStack:
    def __init__(self):
        """初始化一个空栈"""
        self.head = None

    def is_empty(self):
        """判断是否为空栈"""
        return self.head == None

    def push(self, item):
        """入栈操作: 相当于在链表的头部添加一个节点"""
        node = Node(item)
        node.next = self.head
        self.head = node

    def pop(self):
        """出栈操作: 相当于删除链表头节点"""
        if self.is_empty():
            raise Exception('pop from empty LinkListStack')
        item = self.head.value
        self.head = self.head.next

        return item

if __name__ == '__main__':
    s = LinkListStack()
    # 栈（栈底->栈顶）: 300 200 100
    s.push(100)
    s.push(200)
    s.push(300)
    # 终端1: 300
    print(s.pop())
    # 终端2: False
    print(s.is_empty())
```

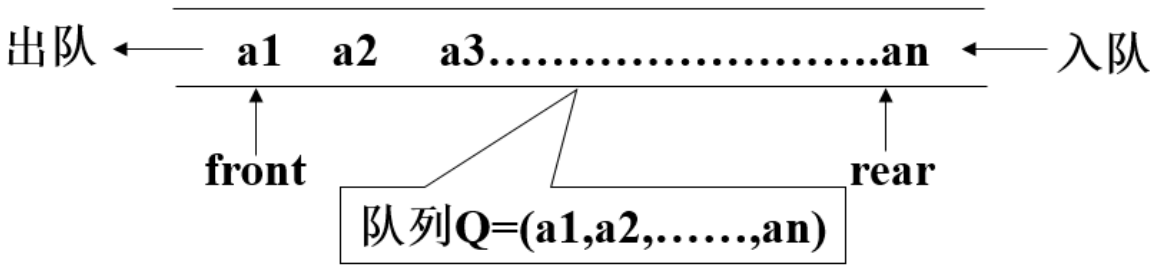
线性表 - 队列（FIFO）

- **定义**

```
队列是限制在两端进行插入操作和删除操作的线性表，允许进行存入操作的一端称为"队尾"，允许进行删除操作的一端称为"队头"
```

- **特点**

```
1) 队列只能在队头和队尾进行数据操作
2) 队列模型具有先进先出规律（FIFO）
```



- **顺序队列代码实现**

```python
"""
顺序存储方式去实现队列模型
思路:
    1、队列: FIFO 先进先出,队尾负责入队,队头负责出队
    2、设计:
       列表头部作为队头,负责出队
       列表尾部作为队尾,负责入队
"""
class Queue:
    def __init__(self):
        """初始化一个空队列"""
        self.elems = []

    def is_empty(self):
        """判断队列是否为空"""
        return self.elems == []

    def enqueue(self, item):
        """队尾入队: append(item)"""
        self.elems.append(item)

    def dequeue(self):
        """队头出队: pop(0)"""
        if self.is_empty():
            raise Exception('dequeue from empty Queue')
        return self.elems.pop(0)

if __name__ == '__main__':
    q = Queue()
    # 队列: 100 200 300
    q.enqueue(100)
```

```
    q.enqueue(200)
    q.enqueue(300)
    # 终端1: 100
    print(q.dequeue())
    # 终端2: False
    print(q.is_empty())
```

- **链式队列代码实现**

```
"""
链式存储方式去实现队列
思路:
    1、队列: FIFO 先进先出
    2、设计:
       链表头部作为队头,负责出队操作
       链表尾部作为队尾,负责入队操作
"""
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkListQueue:
    def __init__(self):
        """初始化一个空队列"""
        self.head = None

    def is_empty(self):
        """判断队列是否为空"""
        return self.head == None

    def enqueue(self, item):
        """队尾入队: 相当于在链表尾部添加一个节点"""
        node = Node(item)
        # 空队列情况
        if self.is_empty():
            self.head = node
            return
        # 非空队列
        cur = self.head
        while cur.next:
            cur = cur.next
        # 循环结束后,cur一定是指向了原链表尾节点
        cur.next = node
        node.next = None

    def dequeue(self):
        """队头出队: 相当于删除链表头节点"""
        if self.is_empty():
            raise Exception('dequeue from empty LinkListQueue')
        cur = self.head
        # 删除头节点
        self.head = self.head.next

        return cur.value

```

```python
if __name__ == '__main__':
    q = LinkListQueue()
    # 队列: 100 200 300
    q.enqueue(100)
    q.enqueue(200)
    q.enqueue(300)
    # 终端1: 100
    print(q.dequeue())
    # 终端2: False
    print(q.is_empty())
```

栈和队列练习一

- **题目描述+试题解析**

```
【1】题目描述
    用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型

【2】试题解析
    1、队列特点：先进先出（时刻铭记保证先进先出）
    2、栈 A 用来做入队列，栈 B 用来做出队列，当栈 B 为空时，栈 A 全部出栈到栈 B，栈B 再出栈(即出队列)
    3、精细讲解
      stack_a：入队列（队尾）
      stack_b：出队列（队头）

      stack_a队尾：[1,2,3]
      stack_b队头：[]
         stack_b.append(stack_a.pop()) # [3,2,1]
         stack_b.pop()  # 1
```

- **代码实现**

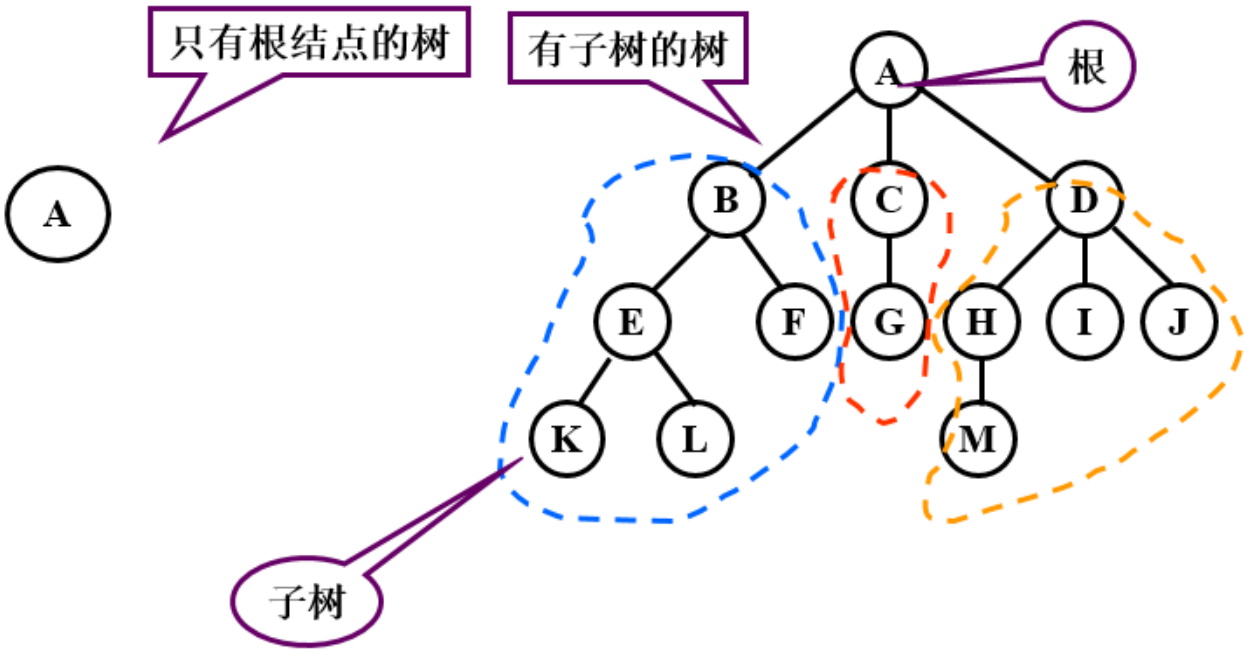
```python
"""
用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型
思路：
    1、明确目标：实现队列(FIFO)
    2、实现载体：2个栈(LIFO)
    3、设计(计划使用顺序栈模型,暂时不使用链式栈模型)
       栈A(列表A) ： 入队
       栈B(列表B) ： 出队
"""
class Solution:
    def __init__(self):
        """先初始化两个空栈"""
        self.stack_a = []
        self.stack_b = []

    def push(self, item):
        """入队操作：相当于在栈stack_a中添加一个元素"""
        self.stack_a.append(item)

```

```
    def pop(self):
        """出队操作：相当于在栈stack_b中弹出一个元素"""
        # 先从栈stack_b中pop()出栈
        if self.stack_b:
            return self.stack_b.pop()

        # 栈stact_b为空时,把栈stack_a中的所有元素再添加到栈stack_b中
        while self.stack_a:
            self.stack_b.append(self.stack_a.pop())

        # 循环结束后,把stack_a中的所有元素添加到了stack_b中
        if self.stack_b:
            return self.stack_b.pop()

if __name__ == '__main__':
    s = Solution()
    # 入队：100 200 300
    s.push(100)
    s.push(200)
    s.push(300)
    # 出队：100 200 300
    print(s.pop())
    print(s.pop())
    print(s.pop())
```

树形结构

- **定义**

```
树（Tree）是n（n≥0）个节点的有限集合T，它满足两个条件：有且仅有一个特定的称为根（Root）的节点；其余的节点可以分为m（m≥0）个互不相交的有限集合T1、T2、……、Tm，其中每一个集合又是一棵树，并称为其根的子树（Subtree）
```



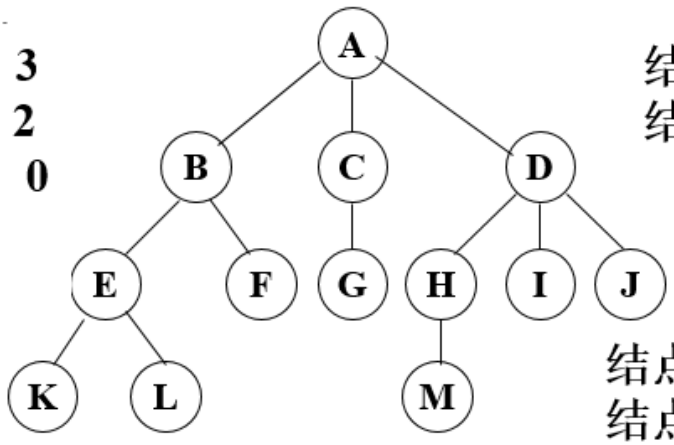
- **基本概念**

```
# 1. 树的特点
* 每个节点有零个或者多个子节点
* 没有父节点的节点称为根节点
* 每一个非根节点有且只有一个父节点
* 除了根节点外,每个子节点可以分为多个不相交的子树

# 2. 相关概念
1) 节点的度：一个节点的子树的个数
2) 树的度：一棵树中,最大的节点的度成为树的度
3) 叶子节点：度为0的节点
4) 父节点
5) 子节点
6) 兄弟节点
7) 节点的层次：从根开始定义起,根为第1层
8) 深度：树中节点的最大层次
```

结点A的孩子：B，C，D　　叶子：K，L，F，G，M，I，J

结点B的孩子：E，F

结点A的度：3
结点B的度：2
结点M的度：0
树的度：3



结点I的双亲：D
结点L的双亲：E

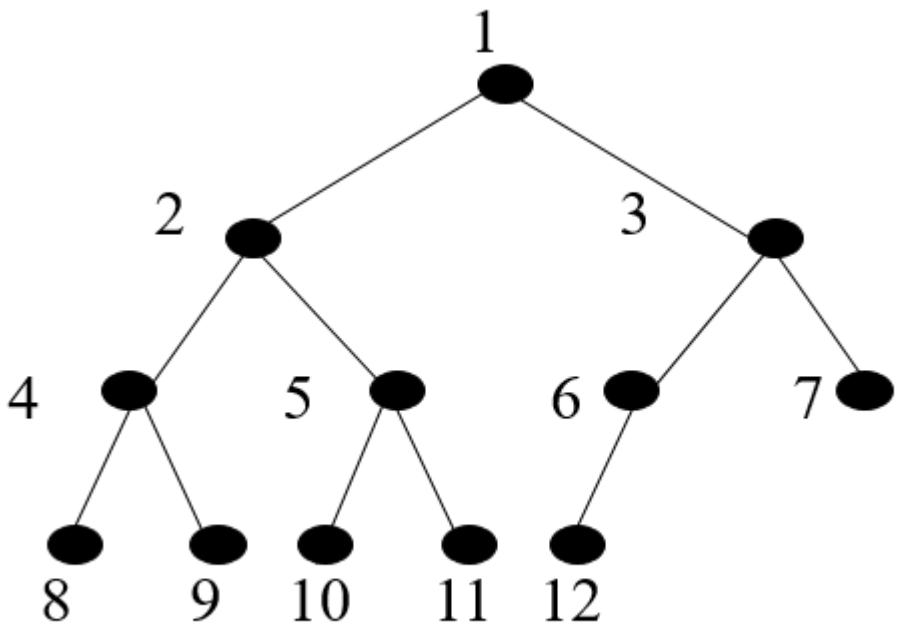
结点B，C，D为兄弟
结点K，L为兄弟

结点A的层次：1
结点M的层次：4　　树的深度：4

二叉树

- **定义**

```
1 二叉树（Binary Tree）是n（n≥0）个节点的有限集合，它或者是空集（n=0），或者是由一个根节点以及两棵互不相交的、分别称为左子树和右子树的二叉树组成。二叉树与普通有序树不同，二叉树严格区分左孩子和右孩子，即使只有一个子节点也要区分左右
```



- **二叉树的分类 - 见图**

```
【1】满二叉树
    所有叶节点都在最底层的完全二叉树

【2】完全二叉树
    对于一颗二叉树，假设深度为d，除了d层外，其它各层的节点数均已达到最大值，并且第d层所有节点从左向右连续紧密排列

【3】二叉排序树
    任何一个节点，所有左边的值都会比此节点小，所有右边的值都会比此节点大

【4】平衡二叉树
    当且仅当任何节点的两棵子树的高度差不大于1的二叉树
```

- **二叉树 - 添加元素代码实现**

```python
"""
二叉树
"""

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class Tree:
    def __init__(self, node=None):
        """创建了一棵空树或者是只有树根的树"""
        self.root = node

    def add(self, value):
        """在树中添加一个节点"""
        node = Node(value)
        # 空树情况
        if self.root is None:
            self.root = node
            return

        # 不是空树的情况
        node_list = [self.root]
        while node_list:
            cur = node_list.pop(0)
            # 判断左孩子
            if cur.left is None:
                cur.left = node
                return
            else:
                node_list.append(cur.left)

            # 判断右孩子
            if cur.right is None:
                cur.right = node
                return
            else:
```

```
                node_list.append(cur.right)
```

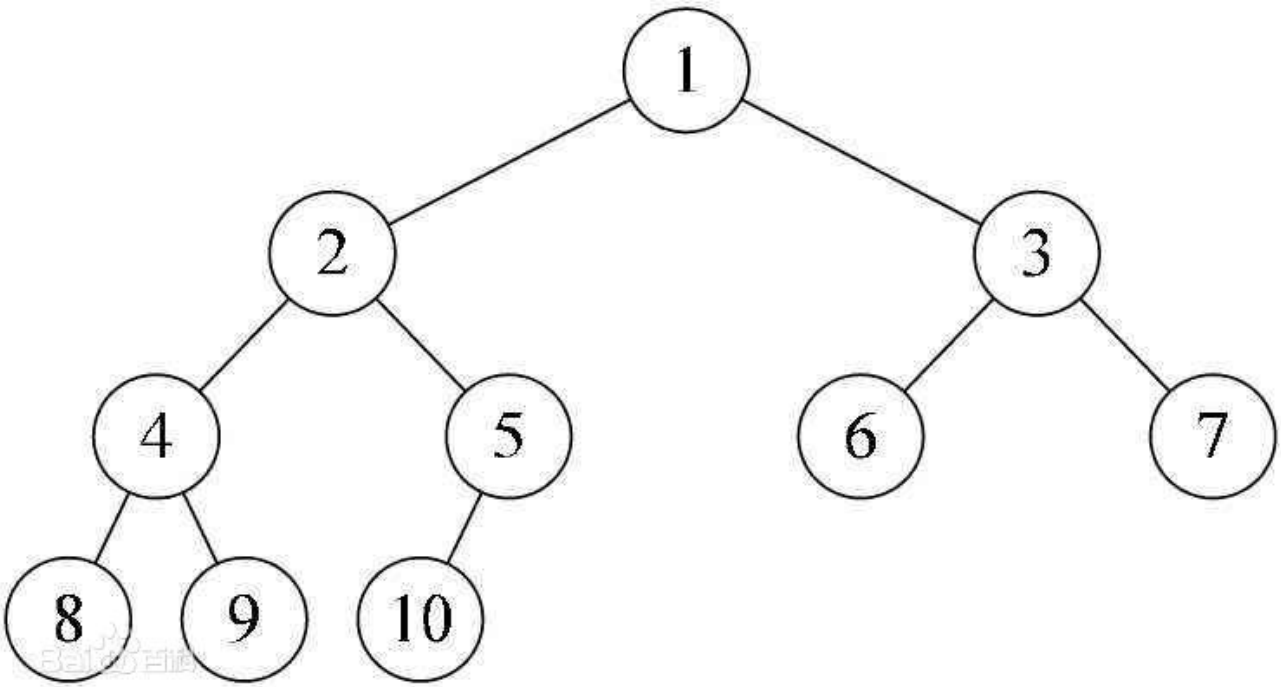
广度遍历 - 二叉树

- **广度遍历 - 代码实现**

```
    def breadth_travel(self):
        """广度遍历 - 队列思想（即：列表的append()方法 和 pop(0) 方法"""
        # 1、空树的情况
        if self.root is None:
            return
        # 2、非空树的情况
        node_list = [self.root]
        while node_list:
            cur = node_list.pop(0)
            print(cur.value, end=' ')
            # 添加左孩子
            if cur.left is not None:
                node_list.append(cur.left)
            # 添加右孩子
            if cur.right is not None:
                node_list.append(cur.right)

        print()
```

深度遍历 - 二叉树

```
【1】遍历
    沿某条搜索路径周游二叉树，对树中的每一个节点访问一次且仅访问一次。

【2】遍历方式
    2.1) 前序遍历：  先访问树根，再访问左子树，最后访问右子树 - 根 左 右
    2.2) 中序遍历：  先访问左子树，再访问树根，最后访问右子树 - 左 根 右
    2.3) 后序遍历：  先访问左子树，再访问右子树，最后访问树根 - 左 右 根
```




```
【1】前序遍历结果: 1 2 4 8 9 5 10 3 6 7
【2】中序遍历结果: 8 4 9 2 10 5 1 6 3 7
【3】后序遍历结果: 8 9 4 10 5 2 6 7 3 1
```

- **深度遍历 - 代码实现**

```python
# 前序遍历
    def pre_travel(self, node):
        """前序遍历 - 根左右"""
        if node is None:
            return

        print(node.value, end=' ')
        self.pre_travel(node.left)
        self.pre_travel(node.right)

# 中序遍历
    def mid_travel(self, node):
        """中序遍历 - 左根右"""
        if node is None:
            return

        self.mid_travel(node.left)
        print(node.value, end=' ')
        self.mid_travel(node.right)

# 后续遍历
    def last_travel(self, node):
        """后序遍历 - 左右根"""
        if node is None:
            return

```

```
        self.last_travel(node.left)
        self.last_travel(node.right)
        print(node.value, end=' ')
```

- **二叉树完整代码**

```
"""
python实现二叉树
"""

class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class Tree:
    def __init__(self, node=None):
        """创建了一棵空树或者是只有树根的树"""
        self.root = node

    def add(self, value):
        """在树中添加一个节点"""
        node = Node(value)
        # 空树情况
        if self.root is None:
            self.root = node
            return

        # 不是空树的情况
        node_list = [self.root]
        while node_list:
            cur = node_list.pop(0)
            # 判断左孩子
            if cur.left is None:
                cur.left = node
                return
            else:
                node_list.append(cur.left)

            # 判断右孩子
            if cur.right is None:
                cur.right = node
                return
            else:
                node_list.append(cur.right)

    def breadth_travel(self):
        """广度遍历 - 队列思想（即：列表的append()方法 和 pop(0) 方法"""
        # 1、空树的情况
        if self.root is None:
            return
        # 2、非空树的情况
        node_list = [self.root]
        while node_list:
            cur = node_list.pop(0)
```

```
            print(cur.value, end=' ')
            # 添加左孩子
            if cur.left is not None:
                node_list.append(cur.left)
            # 添加右孩子
            if cur.right is not None:
                node_list.append(cur.right)

        print()

    def pre_travel(self, node):
        """前序遍历 - 根左右"""
        if node is None:
            return

        print(node.value, end=' ')
        self.pre_travel(node.left)
        self.pre_travel(node.right)

    def mid_travel(self, node):
        """中序遍历 - 左根右"""
        if node is None:
            return

        self.mid_travel(node.left)
        print(node.value, end=' ')
        self.mid_travel(node.right)

    def last_travel(self, node):
        """后序遍历 - 左右根"""
        if node is None:
            return

        self.last_travel(node.left)
        self.last_travel(node.right)
        print(node.value, end=' ')

if __name__ == '__main__':
    tree = Tree()
    tree.add(1)
    tree.add(2)
    tree.add(3)
    tree.add(4)
    tree.add(5)
    tree.add(6)
    tree.add(7)
    tree.add(8)
    tree.add(9)
    tree.add(10)
    # 广度遍历: 1 2 3 4 5 6 7 8 9 10
    tree.breadth_travel()
    # 前序遍历: 1 2 4 8 9 5 10 3 6 7
    tree.pre_travel(tree.root)
    print()
    # 中序遍历:8 4 9 2 10 5 1 6 3 7
    tree.mid_travel(tree.root)
    print()
```

```
    # 后序遍历：8 9 4 10 5 2 6 7 3 1
    tree.last_travel(tree.root)
```

二叉树练习

- **题目描述+试题解析**

```
【1】题目描述
    从上到下按层打印二叉树，同一层结点从左至右输出，每一层输出一行

【2】试题解析
    1、广度遍历，利用队列思想
    2、要有2个队列，分别存放当前层的节点 和 下一层的节点
```

- **代码实现**

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class Solution:
    def print_node_layer(self, root):
        # 空树情况
        if root is None:
            return []
        # 非空树情况
        cur_queue = [root]
        next_queue = []
        while cur_queue:
            cur_node = cur_queue.pop(0)
            print(cur_node.value, end=" ")
            # 添加左右孩子到下一层队列
            if cur_node.left:
                next_queue.append(cur_node.left)
            if cur_node.right:
                next_queue.append(cur_node.right)
            # 判断cur_queue是否为空
            # 为空：说明cur_queue已经打印完成,并且next_queue已经添加完成,交换变量
            if not cur_queue:
                cur_queue, next_queue = next_queue, cur_queue
                print()

if __name__ == '__main__':
    s = Solution()
    p1 = Node(1)
    p2 = Node(2)
    p3 = Node(3)
    p4 = Node(4)
    p5 = Node(5)
    p6 = Node(6)
```

```
    p7 = Node(7)
    p8 = Node(8)
    p9 = Node(9)
    p10 = Node(10)
    p1.left = p2
    p1.right = p3
    p2.left = p4
    p2.right = p5
    p3.left = p6
    p3.right = p7
    p4.left = p8
    p4.right = p9
    p5.left = p10
    s.print_node_layer(p1)
```