

Day02笔记

作业讲解

链表作业题一

■ 题目描述 + 试题解析

- | | |
|---|-----------------------------------|
| 1 | 【1】题目描述 |
| 2 | 输入一个链表，输出该链表中倒数第 k 个节点 |
| 3 | |
| 4 | 【2】试题解析 |
| 5 | 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

```
1  """
2  输入一个链表，输出该链表中倒数第 k 个节点
3  思路：
4      1、链表只能从头到尾遍历，从尾到头遍历存在难度
5      2、从头到尾遍历，将节点数据添加到一个列表中
6      3、利用列表的下标索引取出对应的节点数据
7  """
8  class Node:
9      """节点类"""
10     def __init__(self, value):
11         self.value = value
12         self.next = None
13
14     class Solution:
15         def get_k_node(self, head, k):
16             # 1.把链表中节点数据添加到列表中
17             li = []
18             cur = head
19             while cur:
20                 li.append(cur.value)
21                 cur = cur.next
22             # 2.利用列表的索引取出对应值
23             if k > len(li):
24                 raise IndexError('list index out of range')
25
26             return li[-k]
27
28 if __name__ == '__main__':
29     s = Solution()
```

```

30 # 创建链表: 100 -> 200 -> 300 -> None
31 head = Node(100)
32 head.next = Node(200)
33 head.next.next = Node(300)
34 # 终端1: 200
35 print(s.get_k_node(head, 2))
36 # 终端2: list index out of range
37 print(s.get_k_node(head, 8))

```

链表作业题二

■ 题目描述 + 试题解析

```

1 【1】题目描述
2     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
3
4 【2】试题解析
5     a> 比较两个链表的头节点，确认合成后链表的头节点
6     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

```

1 """
2 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
3 思路:
4     1、程序最终返回的是：合并后的链表的头节点
5     2、先确定新链表的头节点
6     3、互相比对,移动值小的游标
7 """
8 class Node:
9     """节点类"""
10     def __init__(self, value):
11         self.value = value
12         self.next = None
13
14 class Solution:
15     def merge_two_link_list(self, head1, head2):
16         # 1.确定新链表的头节点
17         h1 = head1
18         h2 = head2
19         if h1 and h2:
20             if h1.value >= h2.value:
21                 merge_head = h2
22                 h2 = h2.next
23             else:
24                 merge_head = h1
25                 h1 = h1.next
26         # p即为最终返回的结果
27         p = merge_head
28         elif h1:
29             return h1
30         else:

```

```

31         return h2
32     # 2.遍历两个链表进行比较合并
33     while h1 and h2:
34         if h1.value <= h2.value:
35             merge_head.next = h1
36             h1 = h1.next
37         else:
38             merge_head.next = h2
39             h2 = h2.next
40         # 移动新链表的游标
41         merge_head = merge_head.next
42
43     # 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)
44     if h2:
45         merge_head.next = h2
46     elif h1:
47         merge_head.next = h1
48
49     # 4.最终返回新链表的头节点
50     return p
51
52 if __name__ == '__main__':
53     s = Solution()
54     # 链表1: 100 -> 200 -> 300 -> 400 -> None
55     head1 = Node(100)
56     head1.next = Node(200)
57     head1.next.next = Node(300)
58     head1.next.next.next = Node(400)
59     # 链表2: 1 -> 200 -> 600 -> 800 -> None
60     head2 = Node(1)
61     head2.next = Node(200)
62     head2.next.next = Node(600)
63     head2.next.next.next = Node(800)
64     # 合并
65     p = s.merge_two_link_list(head1, head2)
66     # 结果: 1 100 200 200 300 400 600 800
67     while p:
68         print(p.value, end=' ')
69         p = p.next

```

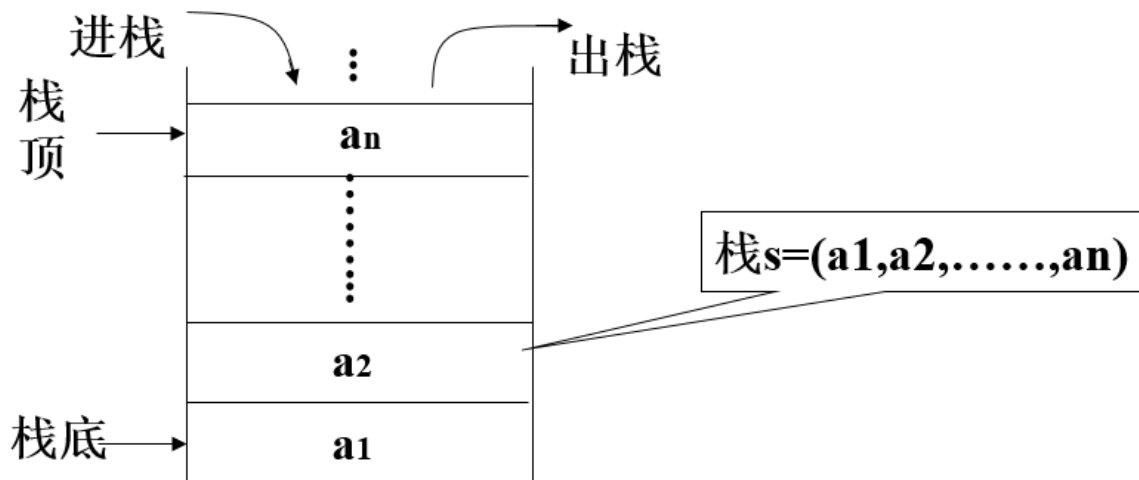
线性表 - 栈 (LIFO)

■ 定义

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



■ 顺序栈代码实现

```

1  """
2  顺序存储的方式实现栈
3  思路:
4      1、栈 : LIFO 后进先出
5      2、设计
6          列表尾部作为栈顶 (入栈、出栈操作)
7          列表头部作为栈底 (不进行任何操作)
8  """
9  class Stack:
10     def __init__(self):
11         """初始化一个空栈"""
12         self.elems = []
13
14     def is_empty(self):
15         """判断栈是否为空栈"""
16         return self.elems == []
17
18     def push(self, item):
19         """入栈: 相当于在链表尾部添加1个元素"""
20         self.elems.append(item)
21
22     def destack(self):
23         """出栈: 相当于在列表尾部弹出1个元素"""
24         if self.is_empty():
25             raise Exception('destack from empty stack')
26         return self.elems.pop()
27
28  if __name__ == '__main__':
29      s = Stack()
30      # 栈(栈底->栈顶): 100 200 300
31      s.push(100)
32      s.push(200)
33      s.push(300)
34      # 终端1: 300 200 100 异常
35      print(s.destack())
36      print(s.destack())
37      print(s.destack())
38      print(s.destack())

```

■ 链式栈代码实现

```
1  """
2  链式存储方式实现栈
3  思路:
4      1、栈: LIFO 后进先出
5      2、设计
6          链表头部作为栈顶 (入栈、出栈操作)
7          链表尾部作为栈底 (不进行任何操作)
8  """
9  class Node:
10     """节点类"""
11     def __init__(self, value):
12         self.value = value
13         self.next = None
14
15  class LinkListStack:
16     def __init__(self):
17         """初始化一个空栈"""
18         self.head = None
19
20     def is_empty(self):
21         """判断是否为空栈"""
22         return self.head == None
23
24     def push(self, item):
25         """入栈操作: 相当于在链表的头部添加一个节点"""
26         node = Node(item)
27         node.next = self.head
28         self.head = node
29
30     def pop(self):
31         """出栈操作: 相当于删除链表头节点"""
32         if self.is_empty():
33             raise Exception('pop from empty LinkListStack')
34         item = self.head.value
35         self.head = self.head.next
36
37         return item
38
39  if __name__ == '__main__':
40     s = LinkListStack()
41     # 栈 (栈底->栈顶) : 300 200 100
42     s.push(100)
43     s.push(200)
44     s.push(300)
45     # 终端1: 300
46     print(s.pop())
47     # 终端2: False
48     print(s.is_empty())
```

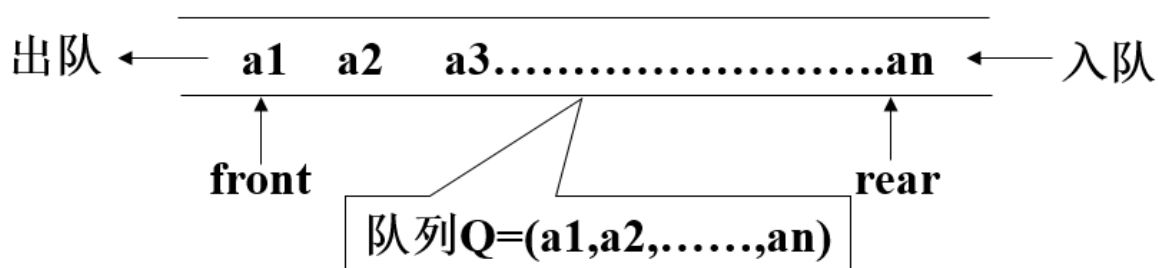
线性表 - 队列 (FIFO)

▪ 定义

- 1 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

```
1 """
2 顺序存储方式去实现队列模型
3 思路:
4     1、队列: FIFO 先进先出,队尾负责入队,队头负责出队
5     2、设计:
6         列表头部作为队头,负责出队
7         列表尾部作为队尾,负责入队
8 """
9 class Queue:
10     def __init__(self):
11         """初始化一个空队列"""
12         self.elems = []
13
14     def is_empty(self):
15         """判断队列是否为空"""
16         return self.elems == []
17
18     def enqueue(self, item):
19         """队尾入队: append(item)"""
20         self.elems.append(item)
21
22     def dequeue(self):
23         """队头出队: pop(0)"""
24         if self.is_empty():
25             raise Exception('dequeue from empty Queue')
26         return self.elems.pop(0)
27
28 if __name__ == '__main__':
29     q = Queue()
30     # 队列: 100 200 300
31     q.enqueue(100)
```

```

32     q.enqueue(200)
33     q.enqueue(300)
34     # 终端1: 100
35     print(q.dequeue())
36     # 终端2: False
37     print(q.is_empty())

```

■ 链式队列代码实现

```

1  """
2  链式存储方式去实现队列
3  思路:
4      1、队列: FIFO 先进先出
5      2、设计:
6          链表头部作为队头,负责出队操作
7          链表尾部作为队尾,负责入队操作
8  """
9  class Node:
10     def __init__(self, value):
11         self.value = value
12         self.next = None
13
14  class LinkListQueue:
15     def __init__(self):
16         """初始化一个空队列"""
17         self.head = None
18
19     def is_empty(self):
20         """判断队列是否为空"""
21         return self.head == None
22
23     def enqueue(self, item):
24         """队尾入队: 相当于在链表尾部添加一个节点"""
25         node = Node(item)
26         # 空队列情况
27         if self.is_empty():
28             self.head = node
29             return
30         # 非空队列
31         cur = self.head
32         while cur.next:
33             cur = cur.next
34         # 循环结束后,cur一定是指向了原链表尾节点
35         cur.next = node
36         node.next = None
37
38     def dequeue(self):
39         """队头出队: 相当于删除链表头节点"""
40         if self.is_empty():
41             raise Exception('dequeue from empty LinkListQueue')
42         cur = self.head
43         # 删除头节点
44         self.head = self.head.next
45
46         return cur.value
47

```

```

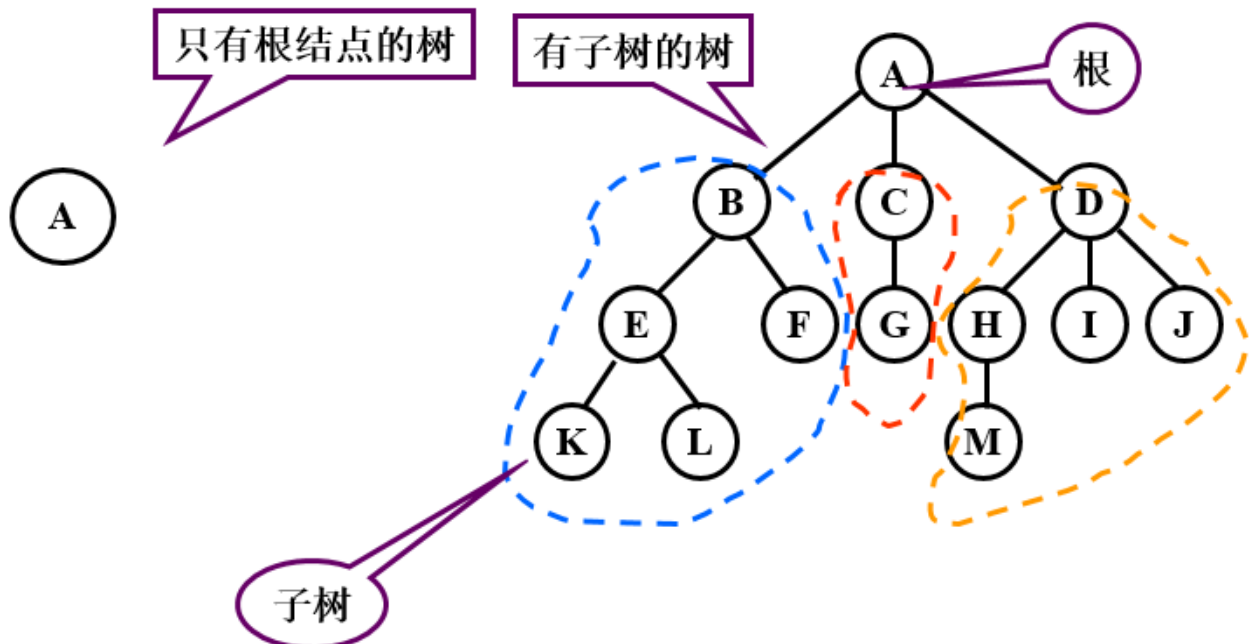
48 if __name__ == '__main__':
49     q = LinkListQueue()
50     # 队列: 100 200 300
51     q.enqueue(100)
52     q.enqueue(200)
53     q.enqueue(300)
54     # 终端1: 100
55     print(q.dequeue())
56     # 终端2: False
57     print(q.is_empty())

```

树形结构

定义

- 1 树 (Tree) 是 n ($n \geq 0$) 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 ($m \geq 0$) 个互不相交的有限集合 T_1 、 T_2 、.....、 T_m ，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 # 1. 树的特点
- 2 * 每个节点有零个或者多个子节点
- 3 * 没有父节点的节点称为根节点
- 4 * 每一个非根节点有且只有一个父节点
- 5 * 除了根节点外, 每个子节点可以分为多个不相交的子树
- 6
- 7 # 2. 相关概念
- 8 1) 节点的度: 一个节点的子树的个数

- 9 2) 树的度：一棵树中，最大的节点的度成为树的度
10 3) 叶子节点：度为0的节点
11 4) 父节点
12 5) 子节点
13 6) 兄弟节点
14 7) 节点的层次：从根开始定义起，根为第1层
15 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

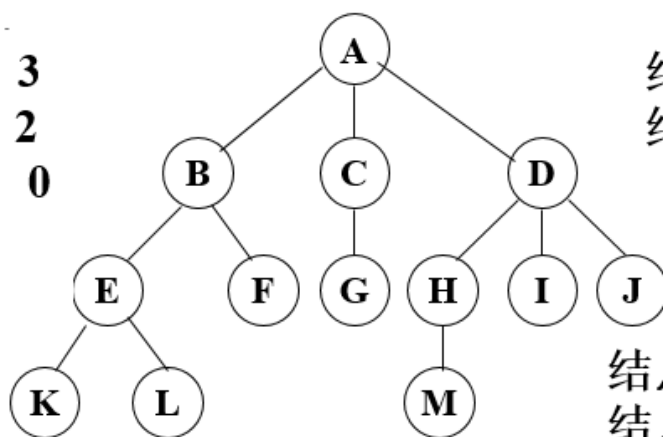
结点B的孩子：E, F

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

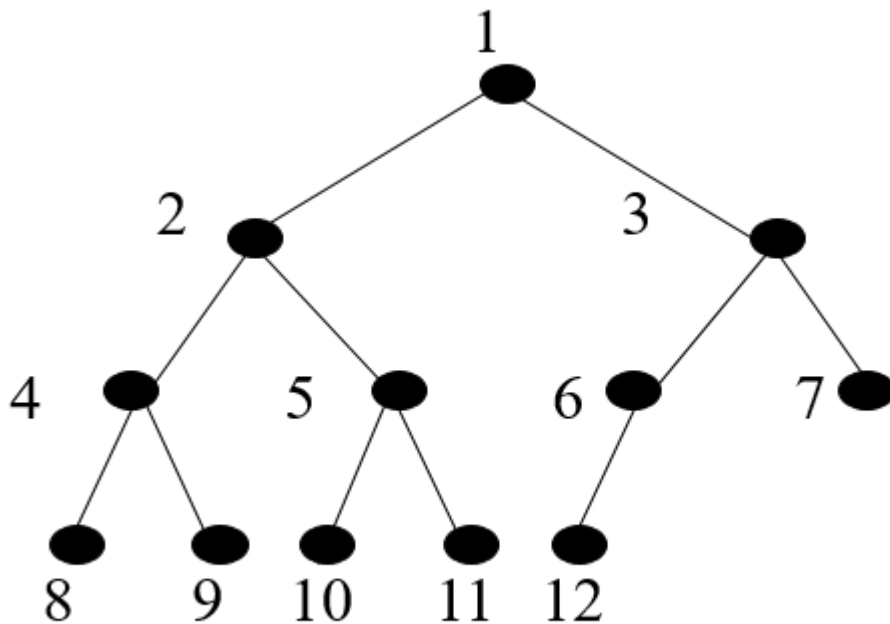
结点A的层次：1

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 ($n \geq 0$) 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 【1】完全二叉树
- 2 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
- 4 【2】满二叉树
- 5 所有叶节点都在最底层的完全二叉树
- 6
- 7 【3】二叉排序树('二叉搜索树')
- 8 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
- 9
- 10 【4】平衡二叉树
- 11 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

```

1  """
2  二叉树
3  """
4
5  class Node:
6      def __init__(self, value):
7          self.value = value
8          self.left = None
9          self.right = None
10
11  class Tree:
12      def __init__(self, node=None):
13          """创建了一棵空树或者是只有树根的树"""
14          self.root = node
15
16      def add(self, value):

```

```

17         """在树中添加一个节点"""
18         node = Node(value)
19         # 空树情况
20         if self.root is None:
21             self.root = node
22             return
23
24         # 不是空树的情况
25         node_list = [self.root]
26         while node_list:
27             cur = node_list.pop(0)
28             # 判断左孩子
29             if cur.left is None:
30                 cur.left = node
31                 return
32             else:
33                 node_list.append(cur.left)
34
35             # 判断右孩子
36             if cur.right is None:
37                 cur.right = node
38                 return
39             else:
40                 node_list.append(cur.right)

```

广度遍历 - 二叉树

■ 广度遍历 - 代码实现

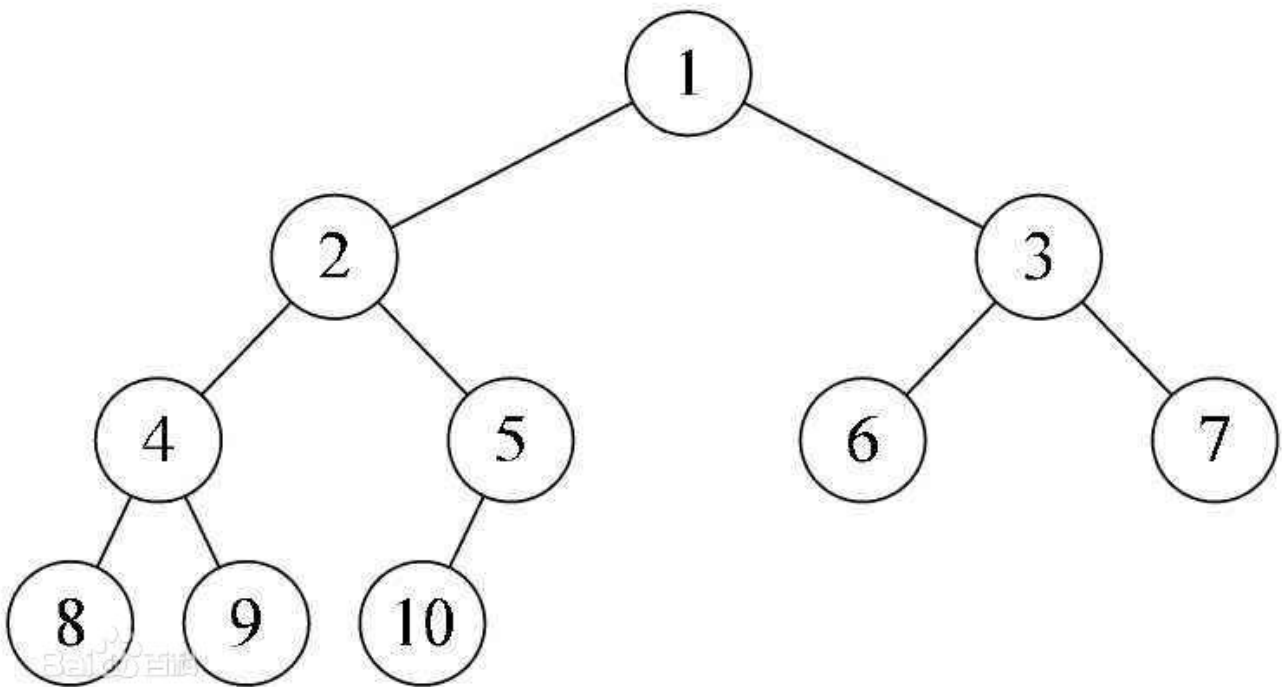
```

1     def breadth_travel(self):
2         """广度遍历 - 队列思想 (即: 列表的append()方法 和 pop(0) 方法)"""
3         # 1、空树的情况
4         if self.root is None:
5             return
6         # 2、非空树的情况
7         node_list = [self.root]
8         while node_list:
9             cur = node_list.pop(0)
10            print(cur.value, end=' ')
11            # 添加左孩子
12            if cur.left is not None:
13                node_list.append(cur.left)
14            # 添加右孩子
15            if cur.right is not None:
16                node_list.append(cur.right)
17
18        print()

```

深度遍历 - 二叉树

- 1 【1】 遍历
- 2 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。
- 3
- 4 【2】 遍历方式
- 5 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
- 6 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
- 7 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 【1】 前序遍历结果：1 2 4 8 9 5 10 3 6 7
- 2 【2】 中序遍历结果：8 4 9 2 10 5 1 6 3 7
- 3 【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

```
1  # 前序遍历
2  def pre_travel(self, node):
3      """前序遍历 - 根左右"""
4      if node is None:
5          return
6
7      print(node.value, end=' ')
8      self.pre_travel(node.left)
9      self.pre_travel(node.right)
10
11 # 中序遍历
12 def mid_travel(self, node):
```

```

13         """中序遍历 - 左根右"""
14         if node is None:
15             return
16
17         self.mid_travel(node.left)
18         print(node.value, end=' ')
19         self.mid_travel(node.right)
20
21     # 后续遍历
22     def last_travel(self, node):
23         """后序遍历 - 左右根"""
24         if node is None:
25             return
26
27         self.last_travel(node.left)
28         self.last_travel(node.right)
29         print(node.value, end=' ')

```

■ 二叉树完整代码

```

1     """
2     python实现二叉树
3     """
4
5     class Node:
6         def __init__(self, value):
7             self.value = value
8             self.left = None
9             self.right = None
10
11     class Tree:
12         def __init__(self, node=None):
13             """创建了一棵空树或者是只有树根"""
14             self.root = node
15
16         def add(self, value):
17             """在树中添加一个节点"""
18             node = Node(value)
19             # 空树情况
20             if self.root is None:
21                 self.root = node
22                 return
23
24             # 不是空树的情况
25             node_list = [self.root]
26             while node_list:
27                 cur = node_list.pop(0)
28                 # 判断左孩子
29                 if cur.left is None:
30                     cur.left = node
31                     return
32                 else:
33                     node_list.append(cur.left)
34
35             # 判断右孩子
36             if cur.right is None:

```

```

37         cur.right = node
38         return
39     else:
40         node_list.append(cur.right)
41
42 def breadth_travel(self):
43     """广度遍历 - 队列思想 (即: 列表的append()方法 和 pop(0) 方法)"""
44     # 1、空树的情况
45     if self.root is None:
46         return
47     # 2、非空树的情况
48     node_list = [self.root]
49     while node_list:
50         cur = node_list.pop(0)
51         print(cur.value, end=' ')
52         # 添加左孩子
53         if cur.left is not None:
54             node_list.append(cur.left)
55         # 添加右孩子
56         if cur.right is not None:
57             node_list.append(cur.right)
58
59     print()
60
61 def pre_travel(self, node):
62     """前序遍历 - 根左右"""
63     if node is None:
64         return
65
66     print(node.value, end=' ')
67     self.pre_travel(node.left)
68     self.pre_travel(node.right)
69
70 def mid_travel(self, node):
71     """中序遍历 - 左根右"""
72     if node is None:
73         return
74
75     self.mid_travel(node.left)
76     print(node.value, end=' ')
77     self.mid_travel(node.right)
78
79 def last_travel(self, node):
80     """后序遍历 - 左右根"""
81     if node is None:
82         return
83
84     self.last_travel(node.left)
85     self.last_travel(node.right)
86     print(node.value, end=' ')
87
88 if __name__ == '__main__':
89     tree = Tree()
90     tree.add(1)
91     tree.add(2)
92     tree.add(3)
93     tree.add(4)

```

```
94     tree.add(5)
95     tree.add(6)
96     tree.add(7)
97     tree.add(8)
98     tree.add(9)
99     tree.add(10)
100    # 广度遍历: 1 2 3 4 5 6 7 8 9 10
101    tree.breadth_travel()
102    # 前序遍历: 1 2 4 8 9 5 10 3 6 7
103    tree.pre_travel(tree.root)
104    print()
105    # 中序遍历: 8 4 9 2 10 5 1 6 3 7
106    tree.mid_travel(tree.root)
107    print()
108    # 后序遍历: 8 9 4 10 5 2 6 7 3 1
109    tree.last_travel(tree.root)
```