九章算法

九章算法基础班

第五讲　二叉树及深度优先遍历

课程版本：v3.0　　张三疯 老师

扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: ninechapter
知乎专栏：http://zhuanlan.zhihu.com/jiuzhang
微博: http://www.weibo.com/ninechapter
官网: www.jiuzhang.com

版权声明

九章课程不提供视频，也严禁录制视频的侵权行为

否则将追求法律责任和经济赔偿

请不要缺课

本节重点

- 什么是树（Tree）？
- 深度优先遍历（Depth-first traverse）
- 递归算法及其复杂度

课程回顾

链表（Linked list）

九章算法

- 什么是链表（linked list）
  - 由节点构成的列表
  - 线性的数据结构
  - 自定义数据结构

```
1  class ListNode:
2
3      def __init__(self, val):
4          self.val = val
5          self.next = None
```

链表操作

- 链表的操作
  - 遍历（traverse）
  - 插入（insert）
  - 查找（find）
  - 更新（update）
  - 删除（delete）

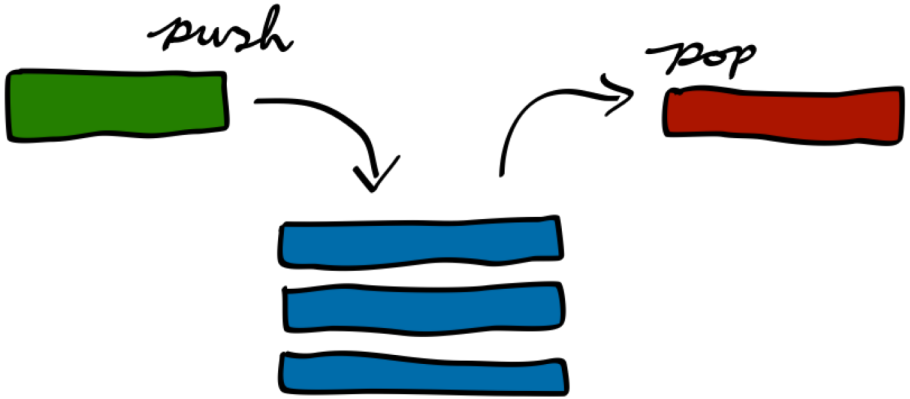
时间复杂度（Time complexity）

- 评估算法时间效率的标准
- 算法的“执行时间”和输入问题规模之间的关系
  - 执行时间：不是实际的时间
  - 输入问题规模：具体问题具体分析

栈（Stack）

- 什么是栈（stack）
  - 栈是一种后进先出（last in first out，LIFO）的线性数据结构

- 栈的操作
  - push(val)
  - pop()
  - peek()
  - is_empty()



队列（Queue）

- 什么是队列（queue）
  - 队列是一种先进先出（first in first out，FIFO）的线性数据结构

- 队列的操作
  - enqueue(val)：进队列
  - dequeue()：出队列
  - size()：队列中元素个数
  - is_empty()：队列是否为空

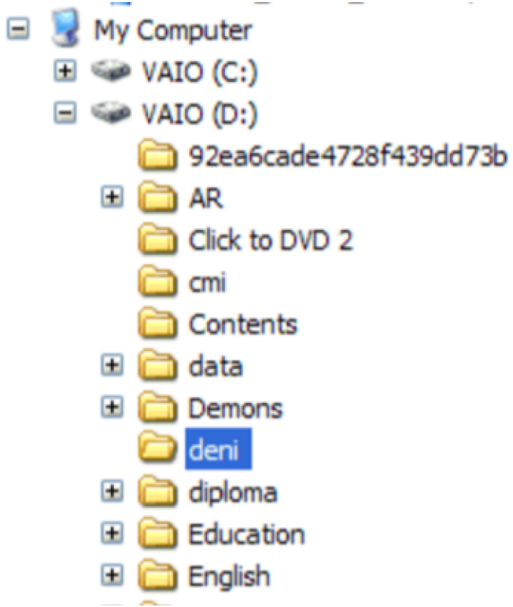
什么是树（Tree）？

树（Tree）

- 什么是树（tree）
  - 树是一种用来表示层次关系的非线性数据结构

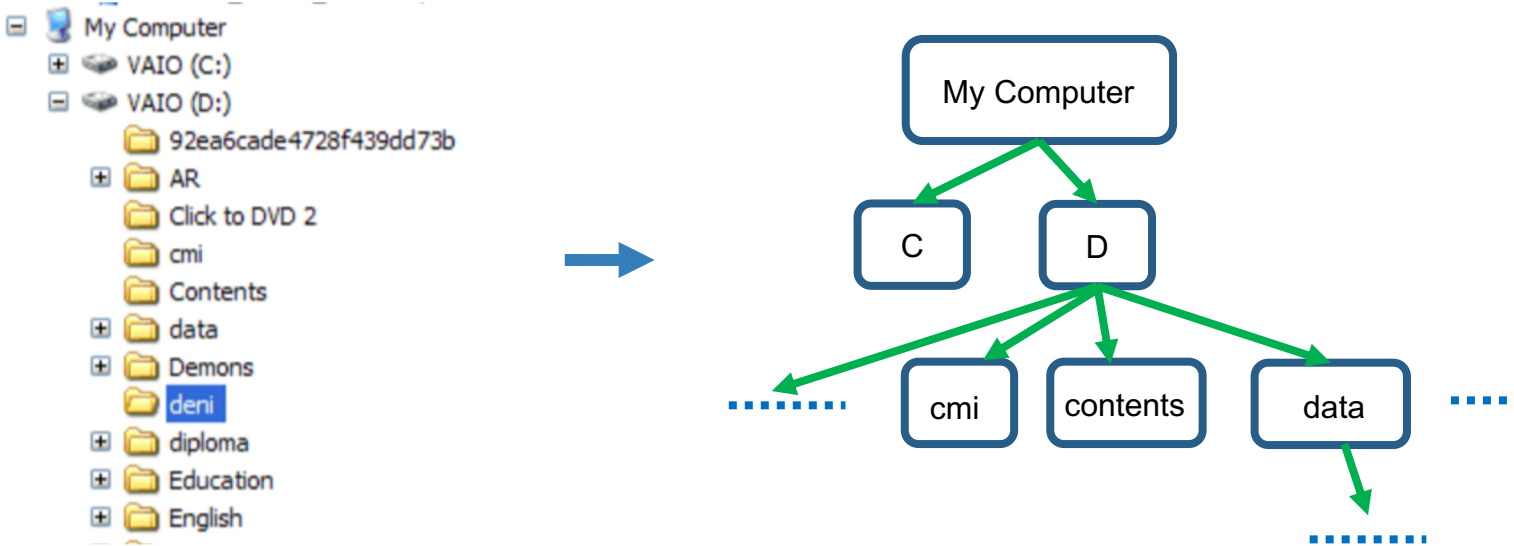
树（Tree）

- 随处可见的数据结构
  - 文件系统
  - 数据库的索引
  - 字典树 - 算法强化班



树（Tree）

- 结构化存储



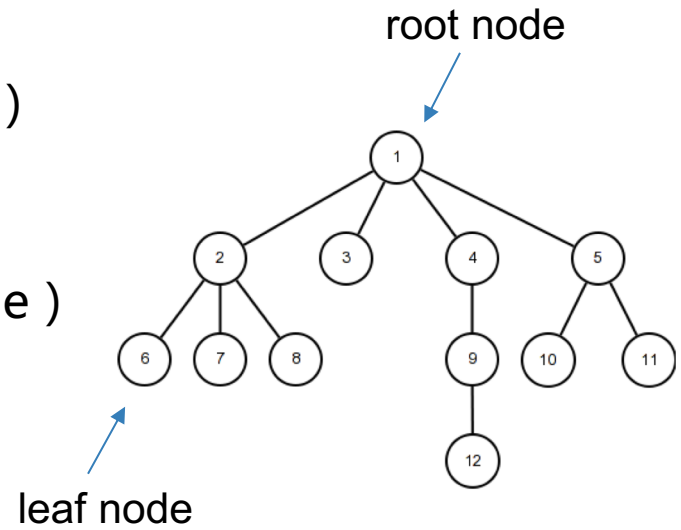
树（Tree）

九章算法

- 树的定义
  - 由节点（node）构成
  - 每个节点有零个或多个子节点（child node）

    没有子节点的是叶子节点（leaf node）
  - 每个非根节点只有一个父节点（parent node）

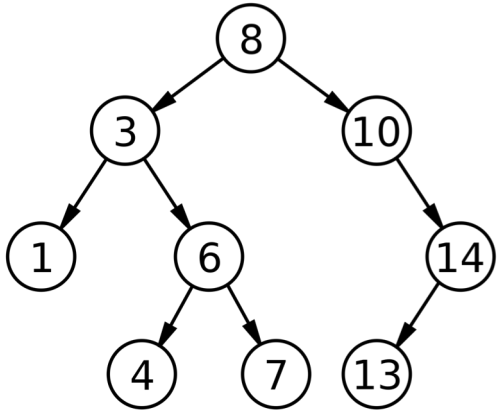
    没有父节点的是根节点（root node）



二叉树（Binary tree）

- 二叉树（Binary tree）
  - 每个节点最多有两个子节点
  - 两个子节点分别被称为左孩子（left child）和右孩子（right child）

- 不特别说明的话，我们提到的树都是指二叉树



二叉树（Binary tree）

九章算法

- 二叉树的构建
  - 先建立单个节点
  - 再将节点连接起来

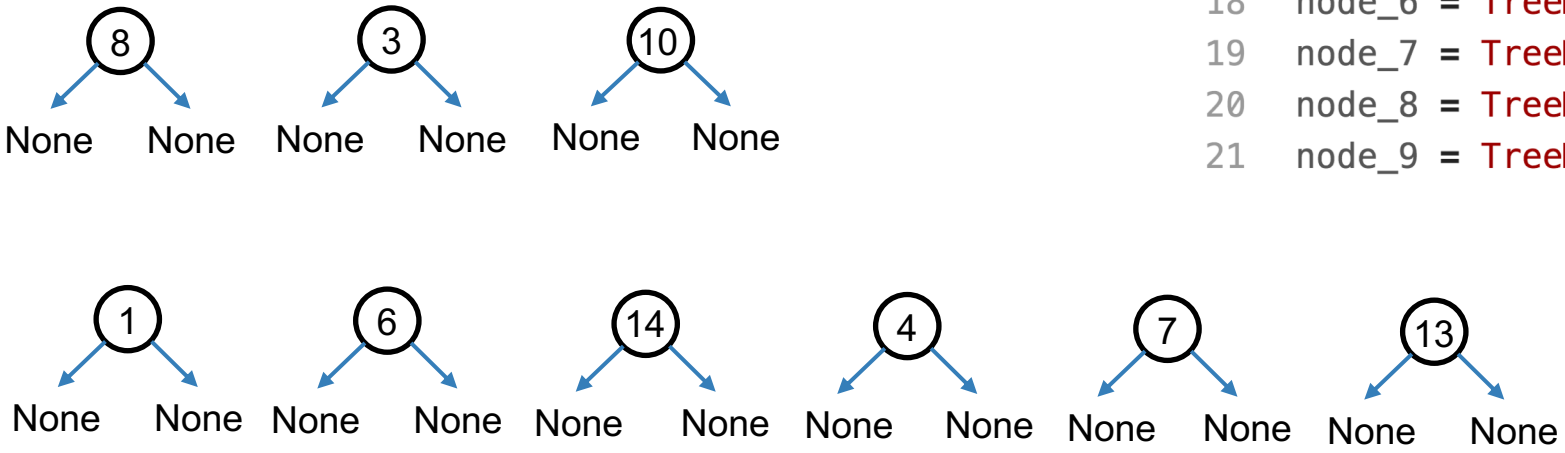
```
class TreeNode:

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

二叉树（Binary tree）

九章算法

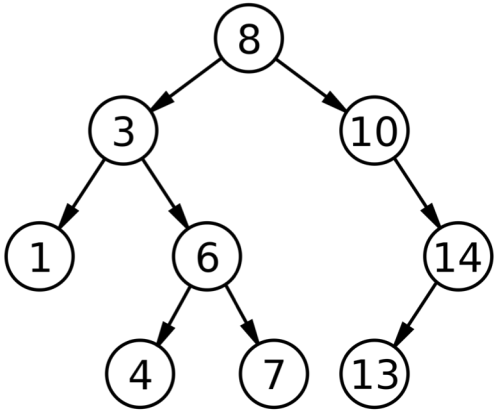
- 二叉树的构建
  - 先建立单个节点

```
13  node_1 = TreeNode(8)
14  node_2 = TreeNode(3)
15  node_3 = TreeNode(10)
16  node_4 = TreeNode(1)
17  node_5 = TreeNode(6)
18  node_6 = TreeNode(14)
19  node_7 = TreeNode(4)
20  node_8 = TreeNode(7)
21  node_9 = TreeNode(13)
```



二叉树（Binary tree）

- 二叉树的构建
  - 再将节点连接起来




```
23  node_1.left = node_2
24  node_1.right = node_3
25
26  node_2.left = node_4
27  node_2.right = node_5
28
29  node_3.right = node_6
30
31  node_5.left = node_7
32  node_5.right = node_8
33
34  node_6.left = node_9
```

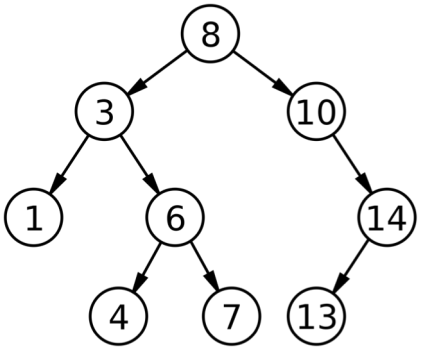
深度优先遍历

树的遍历

- 二叉树的遍历
  - 访问每一个节点，不重不漏
  - 以什么样的顺序来遍历二叉树？

树的遍历

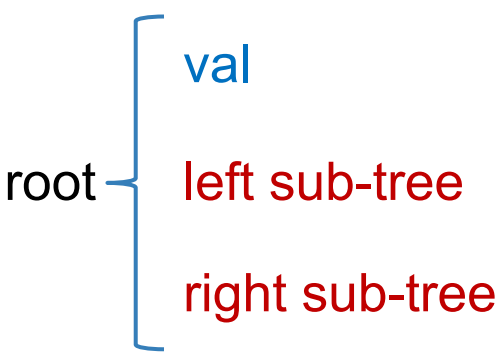
- 子树（sub-tree）
  - 树中的每个节点代表以它为根的一棵树
  - 左孩子所代表的树称为左子树（left sub-tree）
  - 右孩子所代表的树称为右子树（right sub-tree）

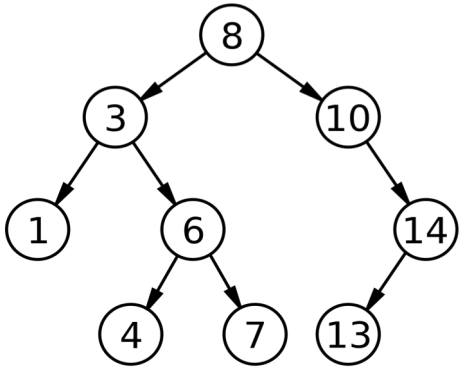


- 例子
  - 节点3代表8的左子树，节点10代表8的右子树

树的遍历

- 递归遍历
  - 代码演示

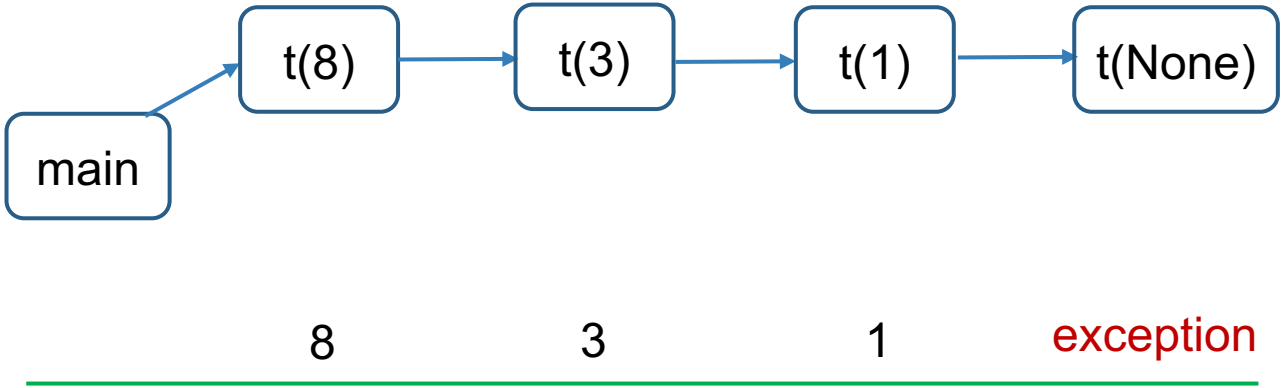


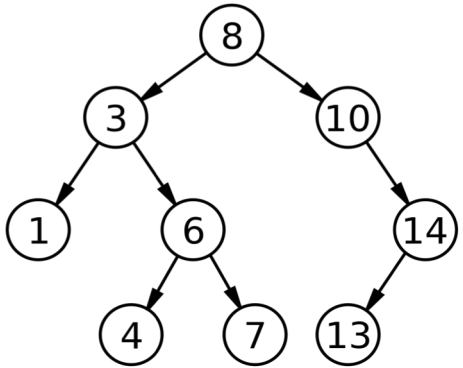


树的遍历

- 递归遍历

```
39  def traverse_tree(root):
40      print(root.val)
41      traverse_tree(root.left)
42      traverse_tree(root.right)
```





树的遍历

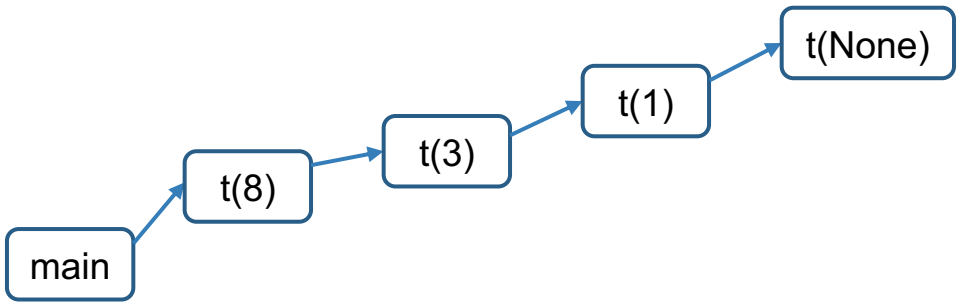
- 递归遍历
  - 终止条件（End case）
  - 最基本情况（Base case）

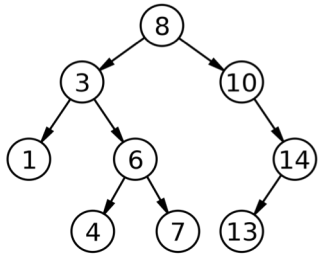
```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```

树的遍历

- 递归遍历

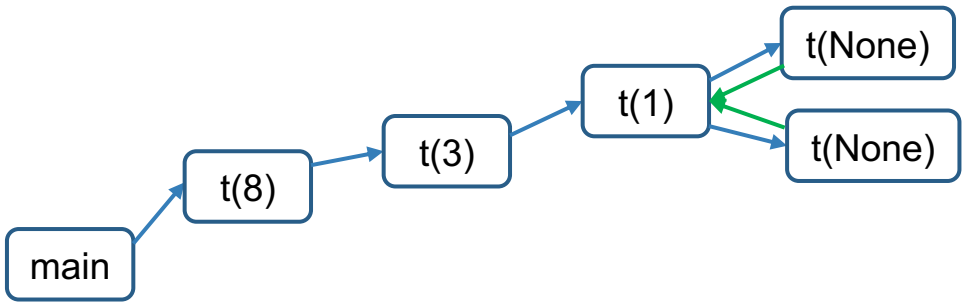
```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```



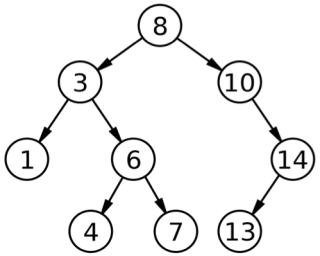


树的遍历

- 递归遍历

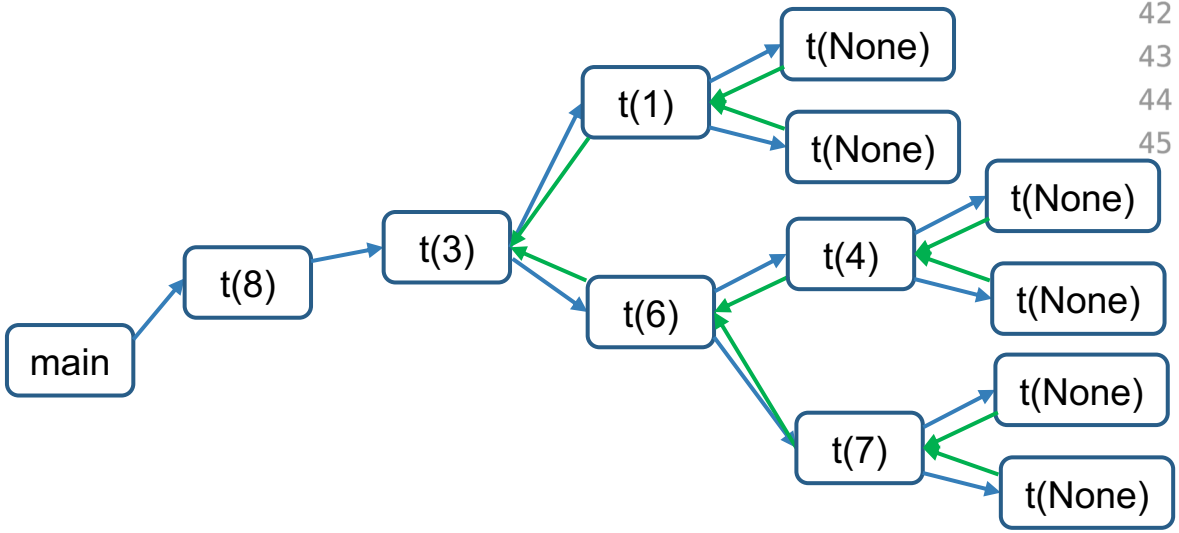



```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```

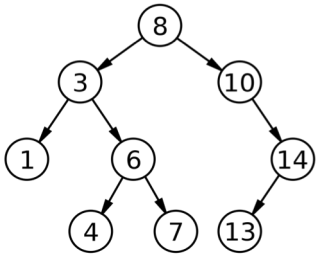


树的遍历

- 递归遍历

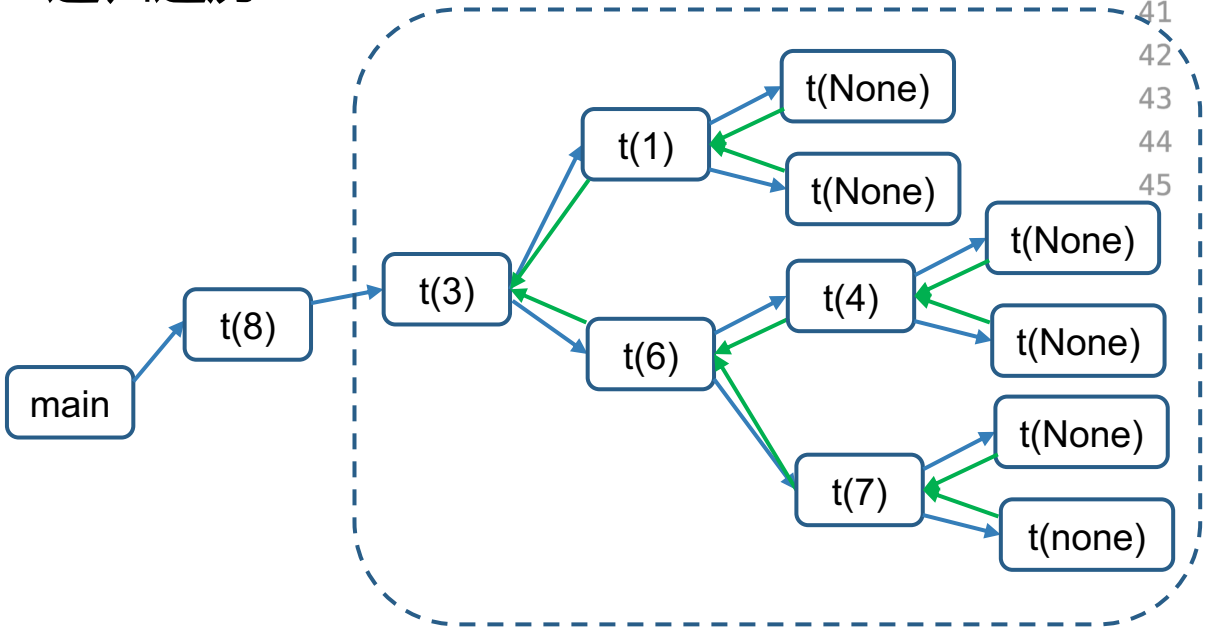



```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```



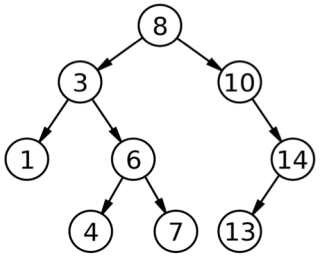
树的遍历

- 递归遍历



traverse_tree(3)

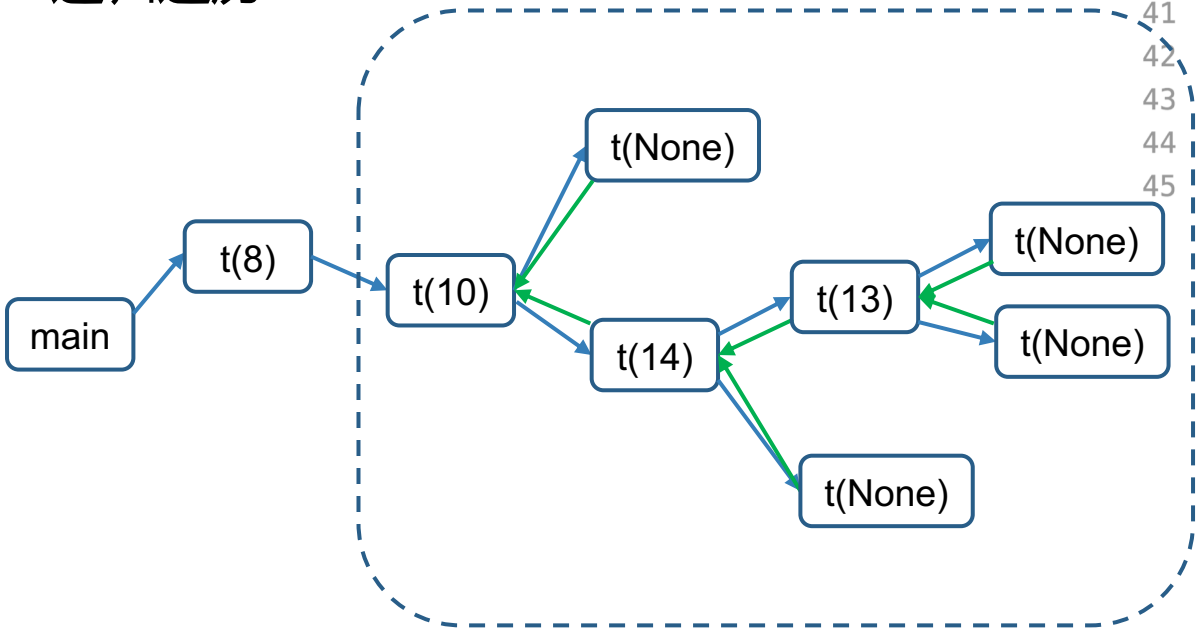
```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```



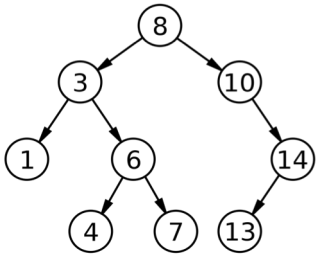
树的遍历

- 递归遍历

```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```



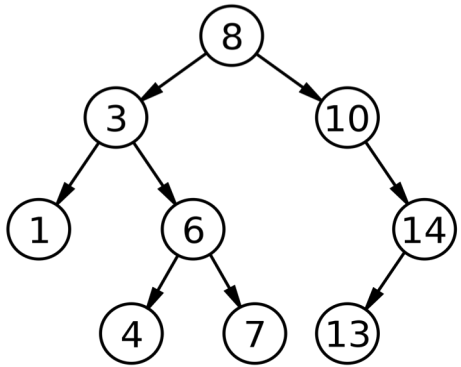
traverse_tree(10)



深度优先遍历

- 先序遍历（preorder）
  - root – left – right
  - 8 3 1 6 4 7 10 14 13
- 中序遍历（inorder）
- 后序遍历（postorder）

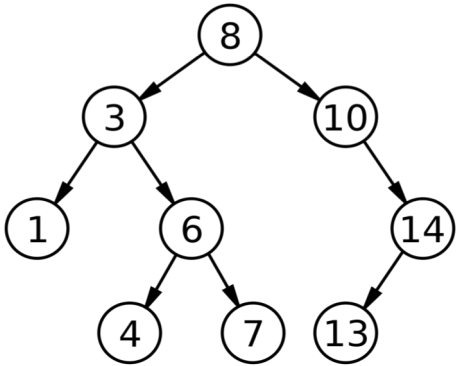
```
48  def preorder_traverse(root):
49      if root is None:
50          return
51
52      print(root.val, end=' ')
53      preorder_traverse(root.left)
54      preorder_traverse(root.right)
```



深度优先遍历

- 中序遍历（inorder）
  - left – root – right
  - 1 3 4 6 7 8 10 13 14

- 后序遍历（postorder）
  - left – right – root
  - 1 4 7 6 3 13 14 10 8



深度优先遍历

- 练习一：
  - Binary tree preorder traversal
  - https://www.lintcode.com/en/problem/binary-tree-preorder-traversal/
  - Binary tree inorder traversal
  - https://www.lintcode.com/en/problem/binary-tree-inorder-traversal/
  - Binary tree postorder traversal
  - https://www.lintcode.com/en/problem/binary-tree-postorder-traversal/

递归算法及其复杂度

递归（Recursive）

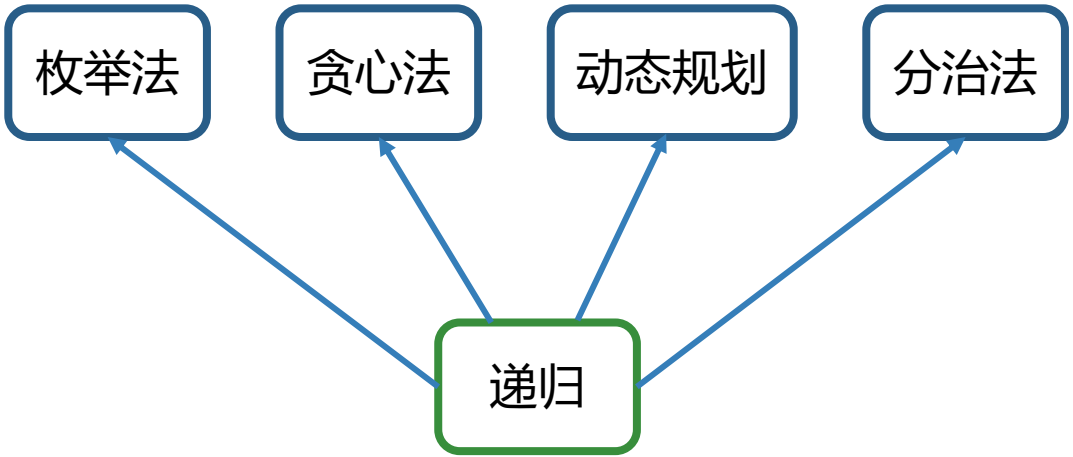
- 什么是递归 (recursive)？
  - 数据结构的递归

    树就是一种递归的数据结构
  - 算法（程序）的递归

    函数自己调用自己

递归（Recursive）

- 算法分类
  - 递归是一种程序的写法



递归三要素

- 递归的定义
  - 首先这个问题或者数据结构需要是递归定义的
- 递归的出口
  - 什么时候递归终止
- 递归的拆解
  - 递归不终止的时候，如何分解问题

递归三要素

- 经典例题Fibonacci
  - https://www.lintcode.com/en/problem/fibonacci/

```python
class Solution:
    """
    @param: n: an integer
    @return: an ineger f(n)
    """
    def fibonacci(self, n):
        # write your code here
        if n == 1:
            return 0
        if n == 2:
            return 1

        return self.fibonacci(n - 1) + self.fibonacci(n - 2)
```

递归三要素

- 递归的定义
  - 斐波那契数列满足F(n) = F(n - 1) + F(n - 2)
- 递归的出口
  - n = 1 和 n = 2 的时候，问题规模足够小
- 递归的拆解
  - self.fibonacci(n - 1) + self.fibonacci(n - 2)

递归（Recursive）

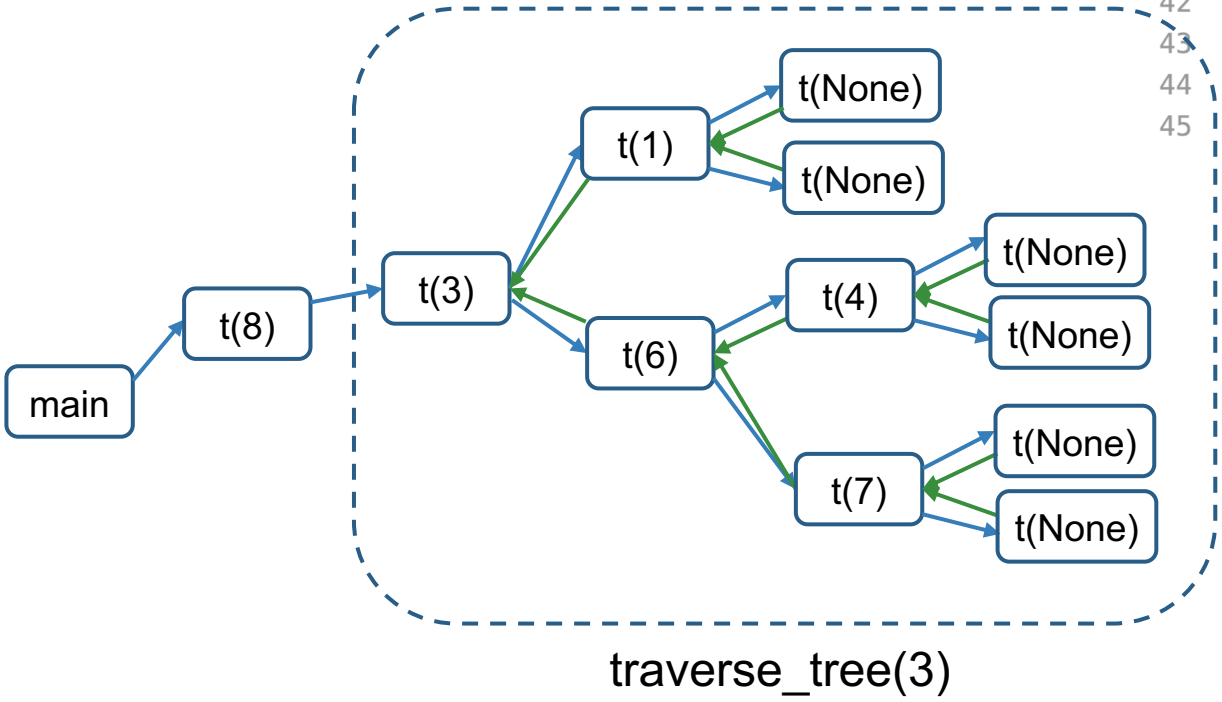
九章算法

- 时间复杂度（Time complexity）
  - 程序执行的时间和输入问题规模之间的关系
  - 函数调用的次数 × 每次函数调用的时间消耗

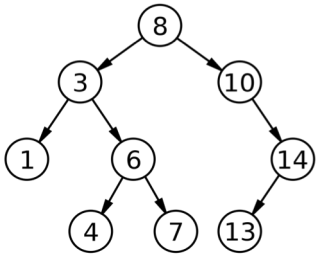
- 空间复杂度（Space complexity）
  - 程序占用的空间和输入问题规模之间的关系
  - 堆（heap）空间 + 栈（stack）空间

二叉树遍历的时间复杂度

- 一次函数调用作为一个时间单位



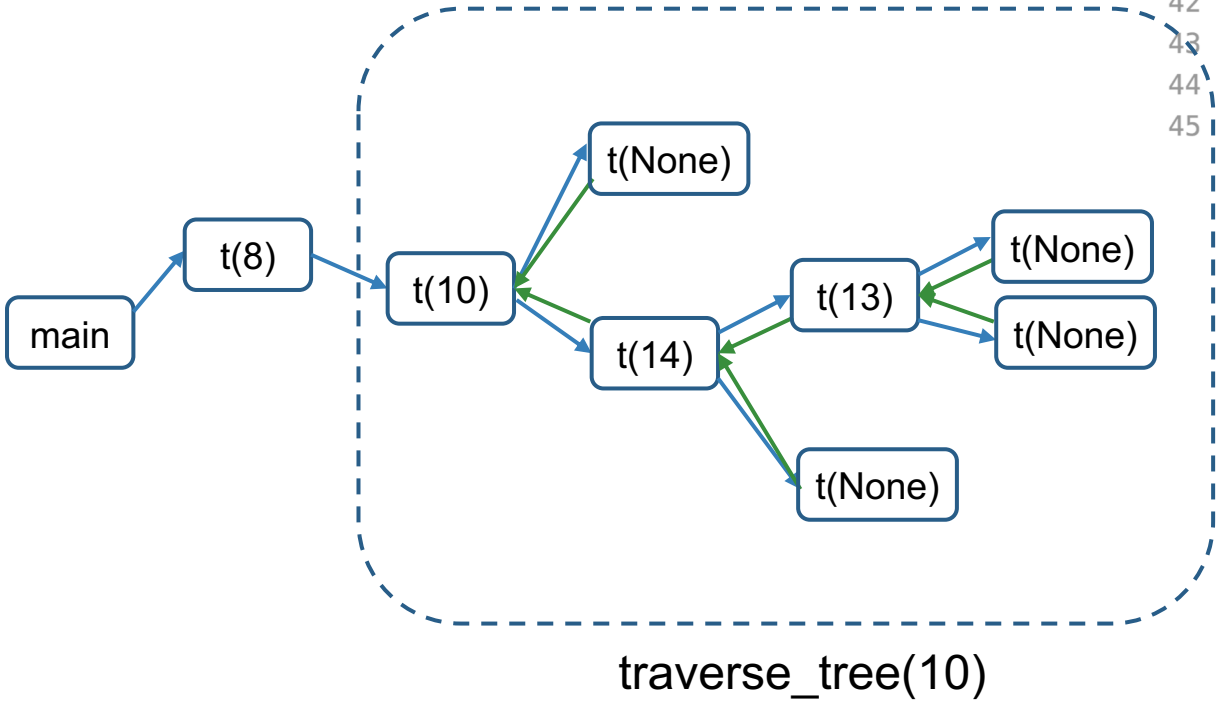

```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```

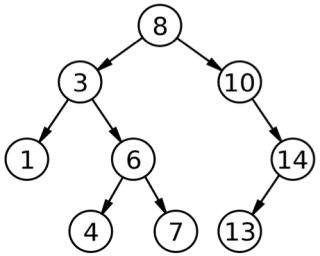


二叉树遍历的时间复杂度

- 时间复杂度：一次函数调用作为一个时间单位

```
39  def traverse_tree(root):
40      if root is None:
41          return
42
43      print(root.val)
44      traverse_tree(root.left)
45      traverse_tree(root.right)
```



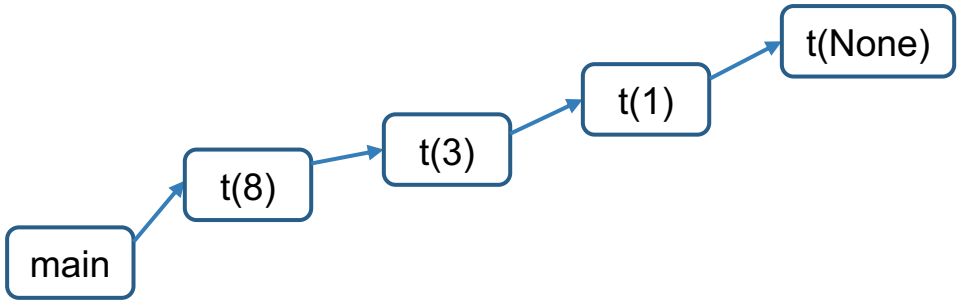


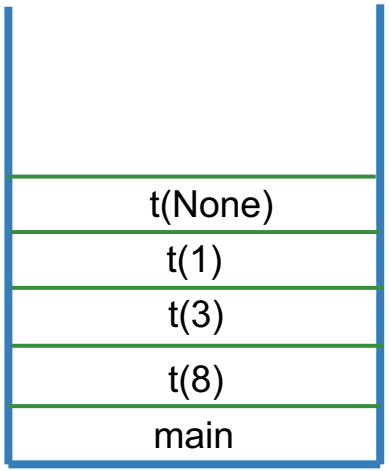
二叉树遍历的时间复杂度

- 二叉树深度优先遍历的时间复杂度
  - 函数会执行2n + 1次
  - 算法的时间复杂度为O(n)

二叉树遍历的空间复杂度

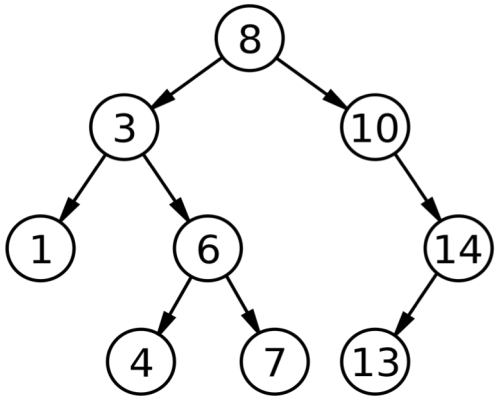
- 二叉树深度优先遍历的空间复杂度





二叉树遍历的空间复杂度

- 二叉树深度优先遍历的空间复杂度
  - 程序最多占用的空间与树的高度成线性关系
  - 算法的空间复杂度为O(h)
  - h介于 logn 和 n 之间



递归（Recursive）

九章算法

- 练习二：面试真题
  - Binary tree leaf sum
  - https://www.lintcode.com/en/problem/binary-tree-leaf-sum/
  - https://www.jiuzhang.com/solution/binary-tree-leaf-sum/

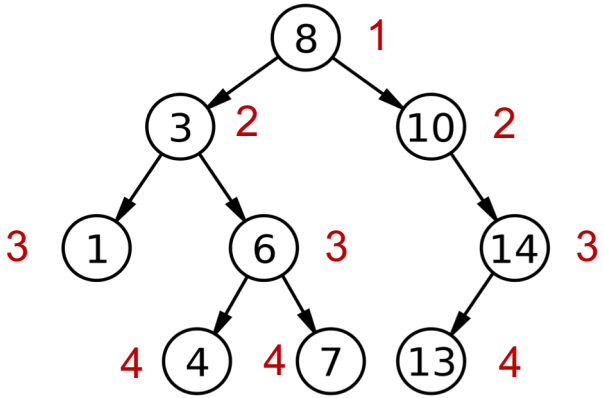
递归（Recursive）

九章算法

- 练习三：面试真题
  - Maximum depth of binary tree
  - https://www.lintcode.com/en/problem/maximum-depth-of-binary-tree/
  - https://www.jiuzhang.com/solution/maximum-depth-of-binary-tree/

递归（Recursive）

- 练习三
  - Maximum depth of binary tree
  - 树的高度 = 节点深度的最大值
  - 借助递归函数参量



总结

- 树是一种用来表示层次关系的非线性数据结构
- 二叉树的深度优先遍历
  - 前序、中序、后序
- 递归算法及其复杂度

Q&A

扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: ninechapter
微博: http://www.weibo.com/ninechapter
官网: www.jiuzhang.com

谢谢大家