

## 数据结构与算法

## 2.1 排序

1. 插入排序  $O(n^2)$  memory = 1eg. 3, 2, 10, 12, 1, 5, 6  $\rightarrow$  3, 4, 2, 10, 12, 1, 5, 6  $\rightarrow$  2, 3, 4, 10, 12, 1, 5, 6 ...

把一个数字放到正确位置，算一轮排序。

2. 冒泡排序(略)  $O(n^2)$  m=13. 选择排序  $O(n^2)$  m=1

即与未排序部分最后一个/第一个元素交换。

从未排序部分重复选择最大(或最小)元素，放在未排序部分最后(或前)。

eg. 64, 25, 12, 22, 11  $\rightarrow$  25, 12, 22, 11, 64  $\rightarrow$  25, 12, 22, 11, 25, 64  $\rightarrow$  12, 11, 22, 25, 64 ...4. 快速排序  $\Theta(n \log n)$   $\Omega(n \log n)$   $\mathcal{O}(n^2)$ . m =  $\log n$ 

基本思想：(1)选定 pivot 中心轴 (一般为最左/右的元素).

(2) 将大于 pivot 的数字放在 pivot 右边 (当前右指针处)

(3) 将小于 pivot 的数字放在 pivot 左边 (当前左指针处)

(4) 将 pivot 放在  
更新左右指针重合处，并分别对左右子序列重复(1)~(3).

把一个子列排好，算一轮。

eg. 14, 17, 13, 15, 19, 10, 3, 6, 9, 12  $\rightarrow$  10, 12, 9, 3, 13, 14, 19, 16, 15, 17 $\rightarrow$  9, 3, 10, 13, 12, 14, 19, 16, 15, 17.

代码实现(双指针):

def quicksort(arr, left, right):

if left &lt; right:

partition\_pos = partition(arr, left, right)

quicksort(arr, left, partition\_pos - 1)

quicksort(arr, partition\_pos + 1, right).

def partition(arr, left, right):

i = left, j = right - 1.

pivot = arr[right]

while i &lt;= j:

while i &lt;= right and arr[i] &lt; pivot: i += 1.

O: worst case

Ω: best case

Θ: average case



扫描全能王 创建

```

while j >= left and arr[j] >= pivot: j -= 1
    if i < j: arr[i], arr[j] = arr[j], arr[i]
if arr[i] > pivot: arr[i], arr[right] = arr[right], arr[i]
return i

```

5. 归并排序  $O(n \log n)$ .  $m = n$

先拆分(拆到子列表长度为1为止)再归并

eg. 21, 1, 26, 45, 29, 28, 2, 9  $\rightarrow$  21, 1, 26, 45  $\xrightarrow{21, 1 \xrightarrow{21} 1, 21}$   
 $\downarrow$   $\xrightarrow{26, 45 \xrightarrow{26} 26, 45}$   $\xrightarrow{1, 21, 26, 45}$   
 $\xrightarrow{29, 28, 2, 9 \xrightarrow{29, 28} \dots}$

6. 布尔排序  $O(n^2)$ .  $m = 1$

eg. 5, 3, 18, 17, 16, 19, 9, 12. length = 8

\* ①每隔4个元素分一组: 5 & 16  $\rightarrow$  5, 16    3 & 19  $\rightarrow$  3, 19  
 $\downarrow$  8/12                  18 & 9  $\rightarrow$  9, 18    17 & 12  $\rightarrow$  12, 17.

原序列变为: 5, 3, 9, 12, 16, 19, 18, 17.

②每隔4/12 = 2个元素分一组: 5 & 18  $\rightarrow$  5, 18    5, 18, 16, 9  $\rightarrow$  5, 9, 16, 18, ...

原序列变为: 5, 3, 18, 17    5, 3, 18, 12, 16, 17, 18, 19.

③ 1个元素分1组, 用插入排序排好: 3, 5, 9, 12, 16, 17, 18, 19.

## 7. 相关概念:

稳定性: 若有两个相等元素 A, B, 且排序前 A 出现在 B 前面, 则能保证排序后 A 仍然出现在 B 前面.

包括归并、插入、冒泡排序.

不稳定排序无法保证 A, B 顺序, 包括选择、希尔、快速排序和堆排序



扫描全能王 创建

## 2. 基本数据结构 (栈、队列)

### 1. 线性数据结构

包含栈、队列、双向队列、列表，指元素的顺序只与其被加入/移除的顺序有关。

### 2. 栈

LIFO — last-in first-out.

`stack()`: create an empty stack.

### 3. 队列

FIFO — first-in first-out / first-come first-served.

元素从 `rear` 向 `front` 移动：push 到最右边，pop 从最左边取出。

### 4. 数组和链表 (均属于线性表) 的比较：

	插入/删除	访问
(顺序存储) 数组 (接近于 List)	$O(n)$ , 必须从头遍历所有元素 移动其他元素	$O(1)$ , 可直接计算元素地址
链表 (链式存储)	$O(1)$ , 只需调整相应节点的指针 通过节点的相互连接实现元素存储	$O(n)$ , 必须从头遍历 找到相应元素

### 5. deque (双端队列)

#### 2. 补充：栈的操作

(1) `isEmpty()` → bool 检验栈是否为空。

(2) `peek()` → 返回最右端的元素，但不将其删除。

(3) `size()`：返回 stack 中元素个数。

(4) 用 class (类) 实现栈及上述操作：

也可以直接用系统的 list 实现。

`class Stack:`

```
def __init__(self):
    self.items = []
```

} 用 `Stack()` 即可创建一个栈 (空列表)，  
并可对栈进行 class 中的操作。

`def is_empty(self):`

```
    return self.items == []
```

`def push(self, item):`

```
    self.items.append(item)
```

`def peek(self):`

或 [-1]

```
    return self.items[len(self.items)-1]
```

`def pop(self):`

```
    return self.items.pop()
```

`def size(self):`

```
    return len(self.items)
```



## 6. 中序、前序、后序表达式

(1) 中序 (infix) eg.  $B^*C$ , 运算符恰好在哪些元素前，就是哪些元素参与运算

(2) 前序 (prefix) eg.  $+A^*BC$  } =  $A + B^*C$

(3) 后序 (postfix) eg.  $ABC^{*+}$  } 运算符紧随哪些元素之后，就是哪些元素参与运算

(4) 将中序转为后序：shunting yard 算法

① 初始化运算符栈和输出栈为空

② 遍历中序表达式，a. 操作数（数字、字母） $\Rightarrow$  push 入输出栈

b. 左括号  $\Rightarrow$  推入运算符栈

c. 运算符  $\Rightarrow$  若优先级大于栈顶运算符，则推入运算符栈，否则

将运算符栈顶元素弹出并推入输出栈，直至输出栈顶运算符优先级更小（或运  
算符栈为空），再将当前这个 ~~及当前运算符~~ 推入这个堆栈并推入运算符栈。

d. 右括号  $\Rightarrow$  将运算符栈顶的运算符弹出并推入输出栈，直  
到遇到左括号。将左括号弹出但不加入输出栈。

③ 遍历后若运算符栈不为空，将运算符依次弹出并推入输出栈。

## 3. 队列：队列的操作

(1) enqueue(item): 向队列 rear 端添加元素

(2) Queue(): 创建新队列，返回一个空队列。

(3) dequeue(): 从 front 端移除一个元素。

(4) isEmpty(): (5) size().

(6) 类实现: class Queue:

```
def __init__(self):
```

```
    self.items = []
```

```
def isEmpty(self):
```

```
    return self.items == []
```

```
def size(self):
```

```
    return len(self.items)
```

```
def enqueue(self, item):
```

```
    self.items.insert(0, item)
```

```
def dequeue(self):
```

```
    return self.items.pop()
```



扫描全能王 创建

(6) 用 deque() 并用 popleft(), O(1). [pop() 为  $O(N)$ ].

deque 对哪一端添加/删除元素无任何限制.

↑ addFront(), addRear(), removeFront(), removeRear().

#### 4-1 单向链表 (LinkedList)

(1) 每个节点包含 { 数据元素 (数据项) : 节点存储的实际情况 }.

指针 (引用) : 指向下一个/前一个节点.

(2) 分类: 单向、双向 (每个节点有 2 个指针)、循环.

(3) 实现: ① 单向链表实现: ② 双向链表实现.

class Node:

def \_\_init\_\_(self, value):

    self.value = value

    self.next = None.

class LinkedList:

def \_\_init\_\_(self, ~~value~~):

~~self.value = value~~

    head  
    self.next = None.

def insert(self, value):

    new\_node = Node(value)

    if self.head is None:

        self.head = new\_node.

    else:

        current = self.head

        while current.next:

            current = current.next

        current.next = new\_node.

def delete(self, value):

    if self.head is None:

        return

class DoublyLinkedList:

    <no: self.tail = None

    ↑ ~~self~~, node 前插 > new-node

def insert\_before(self, node, new\_node):

    if node is None:

        self.head = new\_node

        self.tail = new\_node.

    else:

        new\_node.next = node

        new\_node.prev = node.prev

        if node.prev is not None:

            node.prev.next = new\_node

        else: self.head = new\_node

        node.prev = new\_node.



```
if self.head.value == value:  
    self.head = self.head.next  
else:  
    current = self.head  
    while current.next:  
        if current.next.value == value:  
            current.next = current.next.next  
            break
```

```
def display(self)
```

```
    current = self.head
```

```
    while current:
```

```
        print(current.value, end=" ")
```

```
        current = current.next
```

```
print() # 表示下一次 display 全换行输出
```

## 7. 经典题目用数据结构实现

(1) 模拟括号匹配、前/中/后序表达式转换、八皇后。

(2) 队列：约瑟夫问题、模拟打印机。

从后向

从前向后输出

前输出

def display\_forward(self)/backward(self):

current = self.head / tail

while current is not None:

改成  $\rightarrow$  current = current.next / prev.



扫描全能王 创建

## 1.3. 树

## 1. 基本概念

① generic tree(一般树/造型树)：一个节点可以连多个子树，且子树个数未知。

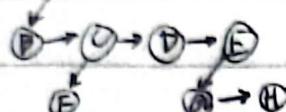
\* 实现： class Node:

```
def __init__(self, data):
    self.data = data
    self.children = []
```

# data 常常写作 value.

② first child / next sibling: 次节点只与第一个子节点相连

③ class Node:



def \_\_init\_\_(self, data):

self.data = data

self.firstchild = None

self.nextsibling = None.

④ 边：连接两个节点。有出入方向，分为出边、入边。

⑤ 叶节点：没有子节点的节点。

⑥ 层级和高度 ≥ 0，深度 ≥ 1. (空树除外)。

## 2. 树的表示方法

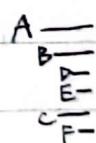
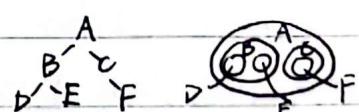
① 嵌套括号表示法。(可以用 stack 游取)。

$a(b(c,d))$ .

② 树形表示

③ Venn 圈

④ 四入表



⑤ 邻接表表示法：适用于稀疏树(树中节点度数相对较小)

$A$	$A: ['B', 'C', 'D']$	$F: [ ]$	$C: [ ]$
$B$	$B: ['E', 'F']$	$D: [G]$	$E: [ ]$
$C$		$G: [ ]$	

⑥ 实现： class TreeNode:

```
def __init__(self, value):
    self.value = value
    self.children = [] # 每个节点用一个数组表示.
```



```

def build_tree():
    root = TreeNode('A')
    node_b = TreeNode('B')
    node_g = TreeNode('G')
    root.children.extend([node_b, node_c, node_d])
    # 数组中每个元素都造
    # 一个链表，包含其子节点。
    node_b.children.extend([node_e, node_f])
    node_d.children.append(node_g)

```

→ return root.

```

def print_tree_adjacency_list(node):
    adjacency_list[node.value] = [child.value for child in node.children]
    for child in node.children:
        build_adjacency_list(child)      # 遍历构建邻接表。
    build_adjacency_list(root)

for node, children in adjacency_list.items():
    print(f'{node}: {children}')

root_node = build_tree
print_tree_adjacency_list(root_node)

```

补充： $\Delta = \text{DFS}$  def print\_tree\_adjacency\_list(root):
 adjacency\_list = {}

1-补充：由树的遍历（以 2(5) 为例）。

① 前序遍历：先访问根节点，然后遍历地前序遍历左子树，最后遍历地前序遍历右子树。 eg. ABECFDG.

② 中序遍历：左子树  $\rightarrow$  根节点  $\rightarrow$  右子树 (= 又树?)

③ 后序遍历：左子树  $\rightarrow$  右子树  $\rightarrow$  根节点 eg. GDCFEDA.

④ PRYR 遍历：

bfs.

实现：前序： def preorder(node):

output = [node.value]

for child in node.children:

output.extend(preorder(child))

return ''.join(output).

def postorder(node):

output = []

for child in node.children:

output.extend(postorder(child))

output.append(node.value)

return ''.join(output).

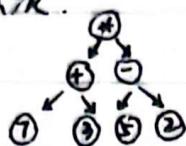


扫描全能王 创建

## 3. 二叉树的应用

(parse tree)

① 解析树 用于表示多级层次。

eg.  $(7+3) * (5-2)$ 

用 getLeftChild, getRightChild 实现完全括号表达式解析树的构建：

② 二叉树的实现：

class BinaryTree:

def \_\_init\_\_(self, rootobj):

self.key = rootobj

self.leftchild = None

self.rightchild = None

def insertleft(self, newNode):

if self.leftchild == None:

self.leftchild = BinaryTree(newNode) # 递归思想.

elif else:

t = BinaryTree(newNode)

t.leftchild = self.leftchild

# insertRight 同理,略

self.leftchild = t.

def getRightChild(self):

left 同理

return self.getrightchild.

def setRootVal(self, obj):

self.key = obj

def getRootVal(self):

return self.key.

② 根据完全括号表达式构建解析树的规则 (currentTree = BinaryTree(" "))

i. 当前标记为 "(" → 为当前节点添加一个左子节点 (insertLeft), 并将栈顶孩子

节点 (getLeftChild).

将 currentTree stack.push(currentTree)

ii. 当前标记在 "+", "-", "\*", "/", "//" 中 → 将当前节点的值设置为当前标记的值



字符 (setRootVal), 为当前节点 insertRightChild 和 getRightChild

iii. 当前标记是数字  $\rightarrow$  将当前节点的值 (setRootVal) 并返回父节点 (currentTree =  $\checkmark$ )  $\stackrel{\text{将为当前节点的值}}{\text{stack.pop()}}$

iv. 当前标记是")  $\rightarrow$  跳到当前节点的父节点 (currentTree =  $\&$  stack.pop(1)).

1(b) - 补充: BinaryTree 类实现前序遍历 (更直观). (后序同理)

def preOrder(tree):

if tree:

print(tree.getRootVal())

preOrder(tree.getLeftChild())

preOrder(tree.getRightChild())

(3) Huffman 算法.

(应用 eg. 勿施子).

① 定义: 将字符串转化为二进制编码. 首先统计字符出现频率, 按频率构建二叉树.

叉树: 频率越高, 离根越近. 编码时, 左孩子编为0, 右孩子编为1.

从而节约内存.

② 实现:

(思路: 使用最小堆, 每次从堆中取出两个频率最小的节点进行合并, 直到堆中只剩下一个节点, 即哈夫曼树的根节点)

import heapq (与 "class Node" (缩进相同))

class Node:

def \_\_init\_\_(self, char, freq):

self.char = char self.freq = freq

self.left = None self.right = None

def \_\_lt\_\_(self, other): return self.freq < other.freq

def huffman\_encoding(char freq):

heap = [Node(char, freq) for char, freq in char\_freq.items()]

heapq.heapify(heap)

while len(heap) > 1: # bfs

left = heapq.heappop(heap) right = heapq.heappop(heap)

merged = Node(None, left.freq + right.freq)



扫描全能王 创建

merged.left = left, merged.right = right  
heapq.heappush(heap, merged).

return heap[0].

- ④ 节点的带权路径长度：该节点到树根之间的路径长度与该节点权值(频率)之积。  
树的带权外部路径长度：树中所有叶子结点的带权路径长度之和。  
\*注：WPL 表示带权外部路径长度。

(不同于二叉树的深度，深度的计算：

def depth(Node):

```
if Node == None: return 0
else:
    left_depth = depth(Node.left)
    right_depth = depth(Node.right)
    return max(left_depth, right_depth) + 1.
```

#### 4. 利用二叉堆实现优先级队列

- (1) 优先级队列：元素的更新操作由优先级决定（元素入队时可能被直接移到头部）  
(2) 二叉堆出入队操作均为  $O(\log N)$ 。  
(3) 用完全二叉树（除最底层外，每一层的节点都是满的）实现。  
（完全二叉树可用一个列表表示，列表中位置为  $p$ ，则左子节点为  $2p$ ，右子节点为  $2p+1$ ）。

(4) 代码：class BinHeap:

```
def __init__(self):
    self.heaplist = [0] # 0 用来占位，使后续元素可使用整数除法。
    self.currentsize = 0.
```

- (5) insert 方法：比较新元素与其父元素（新元素初始时在列表末尾），若新元素  
小于/优先级高于父元素，就将二者交换。

def percup(self, i):

while i//2 > 0:

if self.heaplist[i] < self.heaplist[i//2]:

tmp = self.heaplist[i//2] # 先将 heaplist[i//2] copy一下

self.heaplist[i//2] = self.heaplist[i]

self.heaplist[i] = tmp

i = i//2.

\* 堆的结构性质  
要求根节点遵循的  
最小元素



扫描全能王 创建

```
def insert(self, k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.perchUp(self.currentSize)
```

b) delWith 先定义 minChild(找出两个子节点中较小的那个), 然后定义 perchDown  
(把靠近根部的元素向下调整到合适位置), 最后可定义 delMin.

```
def minChild(self, i): # 一起 minChild 只能向下找一层
```

```
if i * 2 + 1 > self.currentSize: # 防止越界
    return i * 2
```

```
else:
```

```
    if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
        return i * 2
```

```
else: return i * 2 + 1.
```

```
def perchDown(self, i): # 整体框架与 perchUp - 致
```

```
while (i * 2) <= self.currentSize: # 多次使用 minChild 把
    mc = self.minChild(i) # heapList[i] 不断向下
    if self.heapList[i] > self.heapList[mc]: # 交换
        tmp = self.heapList[i]
        self.heapList[i] = self.heapList[mc]
        self.heapList[mc] = tmp
```

```
self.heapList[mc] = tmp
```

```
i = mc.
```

```
def delMin(self): # 只能删除最小的 (即根节点上的) 元素
```

```
retval = self.heapList[1]
```

```
self.heapList[1] = self.heapList[self.currentSize]
```

```
self.currentSize = self.currentSize - 1
```

```
self.heapList.pop()
```

```
self.perchDown(1)
```

```
return retval
```



扫描全能王 创建

## 2. 根据元素列表构建堆

```

def buildHeap(self, alist):
    i = len(alist) // 2 # 超过中点的节点都是叶节点
    self.currentSize = len(alist)
    self.heapList = [0] + alist + [-1]
    while i > 0:
        self.perDown(i) # 此行运行完后, 堆 heapList[i:] 已移到合适位置
        i = i - 1. # 将不是叶节点的节点都往下移(如果它们较大).
    
```

时间

⑥ 构建堆的复杂度为  $O(N)$ , 排序为  $O(N \log N)$ .

## 5. 二叉搜索树 (BST)

① 定义及特征: 小于父节点的键都在左子树中, 大于父节点的键都在右子树中  
(称为二叉搜索性).

② 可实现排序 (quicksort):  $O(N \log N)$ .

通过选择一个元素作为 pivot, 将数组分割为两个子数组, 其中一个子数组的  
元素都小于 pivot, 另一个都大于 pivot. 然后对两个子数组应用相同过程递归地  
(但若 BST 不平衡, 将变为  $O(N^2)$ ).

## ③ 代码实现

```
class TreeNode:
```

```
    def __init__(self, val):
```

```
        self.val = val
```

```
        self.left = None
```

```
        self.right = None
```

```
    def insert(self, root, val): # val 为待插入的值, root 为树根节点.
```

```
        if root is None: # 树中还没有任何元素(节点)时.
```

```
            return TreeNode(val)
```

```
        if val < root.val: root.left = insert(root.left, val)
```

```
        else: root.right = insert(root.right, val)
```

```
        return root
```



```

    (root, result)
def inorder_traversal(): # 前序遍历 (中, 后序类似)
    if root:
        inorder_traversal(root.left, result) # 树的程序中递归很重要.
        result.append(root.val)
        inorder_traversal(root.right, result)

def quicksort(nums): # nums 为输入的一个乱序数字列表
    if not nums: # 树中无节点时 nums 为空, 对应所建树中无节点时
        return []
    root = TreeNode(nums[0])
    for num in nums[1:]: insert(root, num)
    result = []
    inorder_traversal(root, result)
    return result

```

b. 平衡二叉搜索树 (AVL 树, 也是一种 BST, 有二叉搜索性).

(1) 概述: 通过在每个节点上维护一个平衡因子 (节点左子树与右子树高度之差的绝对值) 来实现平衡, 使最坏情况下查找、插入、删除保持  $O(\log N)$ .

在每次插入 / 删除后, 会调整树的结构使 balanceFactor 为  $0, -1, +1$ .

$\text{balanceFactor} = \text{height}(\text{leftSubTree}) - \text{height}(\text{right SubTree})$ .

$bf > 0$  称左倾,  $< 0$  称右倾,  $= 0$  称完全平衡.  $|bf| \leq 1$  称平衡树.

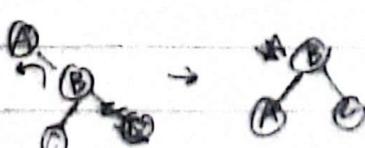
(2) 最少的节点数 (最不平衡的状态)

给定 ~~最小~~ 高度  $h$ , 最小节点数  $N_h = 1 + N_{h-1} + N_{h-2}$  ( $N_0=1, N_1=2$ ).

$0^0 \quad 0^1 \quad 0^0 \quad 0^0 \quad 0^0 \quad \leftarrow N=0 \rightarrow$  时最平衡的情况  
(节点旁的数字为  $bf$ ).

### 3. 实现

① 左旋: i. 将右子节点 (B) 提升为根节点



ii. 将旧的根节点 (A) 作为新根节点的左子节点.

iii. 原生新根节点 (B) 已有一个左子节点 (C).

将其作为右子节点.

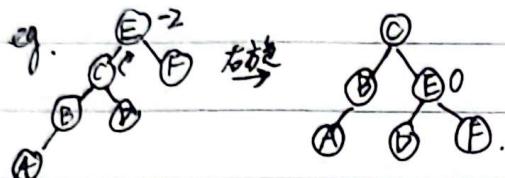


②右旋：将左子节点提升为新根节点.

i. 将旧根节点作为新根节点的右子节点

ii. 若新根节点原先已有一个右子节点，将其作为新右子节点的左子节点.

注：节点移动时，都需带着其子节点.



③实践中对失衡节点的调整.

最近插入节点的失衡节点，平衡因子只能为±2.

若为2，分 LL 型 和 LR 型； L: 左倾 R: 右倾.

若为-2，分 RR, RL.

LL: 根节点处右旋一次

(同理，RR: 根节点处左旋一次)

LR: 节点 A 处左旋一次转化为 LRL，再以根节点右旋一次

(同理，RL型先左旋一次，再右旋一次).

(4)代码

c9. 将 n 个互不相同的正整数先后插入一棵空的 AVL 树.

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

self.height = 1

class AVL:

def \_\_init\_\_(self):

self.root = None

def insert0(self, value):

if not node:

return Node(value)

# 此处 insert0 是经典.

将元素插入 BST 底层  
的方法.

elif value < node.value:

node.left = self.insert0(value, node.left)

else: node.right = self.insert0(value, node.right)

node.height = 1 + max(self.height(node.left),

self.height(node.right)).

balance = self.getBalance(node).

if balance > 1:

if value < node.left.value: # 树为 LL.



```

    return self.rotate_right(node)
else: # 树形为 LR
    node.left = self.rotate_left(node.left)
    return self.rotate_right(node)
if balance < -1:
    if value > node.right.value: # 树形为 RR
        return self.rotate_left(node)
    else: # 树形为 RL
        node.right = self.rotate_right(node.right)
    return self.rotate_left(node)
return node

```

```

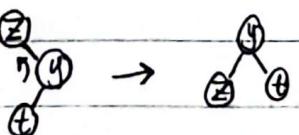
def insert(self, value): # insert 利用 insert() 维持二叉搜索性。
    if not self.root: self.root = Node(value)
    else: self.root = self.insert0(value, self.root)
def height(self, node):
    if not node: return 0
    return node.height
def getBalance(self, node):
    if not node: return 0
    return self.height(node.left) - self.height(node.right)

```

```

def rotate_left(self, z):
    y = z.right
    t = y.left
    y.left = z
    y.height = t
    z.height = max(self.height(z.left), self.height(z.right))
    y.height = 1 + max(self.height(y.left), self.height(y.right))
    return y

```



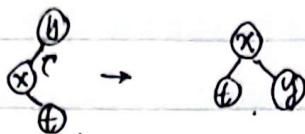
```
def rotate_right(self, y):
```

$x = y.left$

$t = x.right$

$x.right = t$

$y.left = t$



$y.height = 1 + \max(\text{self.height}(y.left), \text{self.height}(y.right))$

~~$x.height = 1 + \max(\text{self.height}(x.left), \text{self.height}(x.right))$~~

return  $x$

\*sequence = list(map(int, input().strip().split())) # strip: 去除字符串开头和结尾的空格和换行符.

avl = AVL()

for value in sequence: avl.insert(value)

## 7. 并查集 (Disjoint Set).

(1) 使用邻接表表示 (仍是一种树形结构, 每个节点存储其父节点的指针)

(2) 每个元素都属于一个集合, 且集合间不相交.

树形结构中, 每个集合用一棵树表示, 树的根节点是集合代表元素.

(3) 代码实现:

class \*DisjSet:

def \_\_init\_\_(self, n): # n为元素数.

self.rank = [1] \* n

self.parent = [i for i in range(n)]

def find(self, x): # 找到给定元素 x.

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x].

def union(self, x, y):

xset = self.find(x)

yset = self.find(y)

if xset == yset: return



```

if self.rank[xset] < self.rank[yset]
    self.parent[xset] = yset
elif self.rank[xset] > self.rank[yset]
    self.parent[yset] = xset
else:
    self.parent[yset] = xset
    self.rank[xset] += 1.

```

## 14. 图

### 一、概述

1. 概念 由顶点和边组成

$G(V, E)$ : 图  $G$  的顶点集为  $V$ , 边集为  $E$ .

2. 有向图: 所有边有向

无向图: 所有边为双向.

3. 顶点的度 (degree): 和某顶点相连的边的条数

(对于有向图, 还分出度、入度)

4. 权值: 顶点和边被量化的属性, (点权/边权)

5. 路径: 由边连接的顶点组成的序列 (长度边数 / 边数权值之和).

6. 环: 起终点相同的路径.

DA G: 有向图无环图

### 二、表示方法

1. 图相关函数定义

(1) Graph(): 创建空图

(2) addVertex(vert) 向图中添加一个顶点 vert.

(3) addEdge(fromVert, toVert, weight)

(4) getVertex(vertkey) 在图中找到名为 vertkey 的顶点

(5) getVertices() 以列表形式返回图中所有顶点

(6) vertexInGraph(vert, graph)  $\rightarrow$  bool



扫描全能王 创建

图的常用的实现：邻接矩阵 (adjacency matrix), 邻接表 (adjacency list).

b. 邻接表 dict 框架 (list / set) (将每个 vertex 的邻接值为 list)

e.g. graph: { 'A': ['B', 'C'], 'B': ['A', 'D'], ... }

\* 宽度树 / 最短树 (Tree).: dict 套 dict.

graph = { 'A': { 'B': 1, 'C': 2, 'D': { 'is\_end': True } } }

### (2) 邻接矩阵

每行、每列均表示图中一个顶点，第 V 行、第 W 列交叉表示顶点 V 到 W 的权值。只有相邻 (直接被一条边相连) 顶点对在朋友才有权值  
（孤立、稀疏（多数单元格为空））。

(无权值处可赋为 -1)

缺省：稀疏（多数单元格为空）。

△ 实现 (代码清单).

class Vertex:

def \_\_init\_\_(self, key):

self.id = key

self.connectedTo = {}

def addNeighbour(self, nbr, weight=0):

self.connectedTo[nbr] = weight. #记录 self 与 nbr 的连接及权值

def \_\_str\_\_(self):

return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

def getConnections(self):

return self.connectedTo.keys() # 返回与 self 相邻的所有顶点

def getId(self):

return self.id

def getWeight(self, nbr):

return self.connectedTo[nbr] # self 与 nbr 之间权值

class Graph:

def \_\_init\_\_(self):

self.vertList = {}

self.numVertices = 0



```

    . def addvertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    . def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else: return None

    . def contains_(self, n):
        return n in self.vertList

    . def addEdge(self, f, t, weight=0):
        if f not in self.vertList:
            newv = self.addvertex(f)
        if t not in self.vertList:
            newv = self.addvertex(t)
        self.vertList[f].addNeighbour(self.vertList[t], weight)

    . def getvertices(self):
        return self.vertList.keys()

    . def iter_(self):
        return iter(self.vertList.values())

```

#iter\_ 使遍历图中所有顶点对更加方便。

### 三. 图的遍历

#### 1. 宽度优先搜索 (BFS)

(1) 优点：可寻找最短路径。

缺点：时间复杂度高。

(2) 时间复杂度为  $O(|V| + |E|)$ ,

$|V|$  为 node (vertex) 数,  $|E|$  为 edge 数



(3) 代码实现:

```
def bfs(graph, initial):
    visited = []
    queue = [initial]
    while queue:
        node = queue.pop(0)
```

# 使用 deque.

if node not in visited:

visited.append(node)

~~neighbours = graph[node]~~

for neighbour in neighbours[node].getNeighbours:

queue.append(neighbour)

return visited.

print(bfs(graph, 'A')).

(4) 深度优先遍历应用.

① “字典”：通常是字典，将有某种共同属性的元素放入一个 key 下（可用列表储存），可更高效地构建关系图。

② 标记颜色

i. 构建时初始化为白色，代表未被访问。

ii. 第一次访问后，标记为灰色。

iii. 完成对该顶点的访问（没有白色顶点与之相连）后，标记为灰色。

③ 记录前驱 (previous, 上一个访问的顶点)

④ 记录距离

② ~ ④ 的实现：

```
class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectTo = {}
        self.color = 'white'
        self.previous = None
        self.distance = float('inf')
```



self.distance = 0 (或如果要找最小步数, 可将距离初始化为由  
self.previous = None 题中给出的最大可能值, 后续每次取min).

在BFS部分中, 将(3)中的循环改成:

while queue:

~~node~~ = queue.pop(0)

for neighbour in node.getNeighbours():

if neighbour.color == 'white':

neighbour.color = 'grey'

neighbour.distance = ~~node~~.distance + 1

neighbour.previous = node

queue.append(neighbour)

~~node~~.color = 'black' # 表示当前节点已处理完

## 2. 深度优先搜索 (DFS).

1. 骑士周游问题 - Warnsdorff 算法. (属于启发式技术).

主要思想: 优先选择下一步可行路径中具有较少可选路径的顶点 (子节点中可行节点少的), 从而尽可能减少搜索空间.

(注: 骑士周游指在8x8棋盘上, 按“马走日”规则周游, 找到使每一格都恰好访问过一次的方法; 本题给定起点, 找求能否周游).

代码: class graph:

def \_\_init\_\_(self): self.vert = {}, self.num\_vert = 0.

def add\_vertex(self, key):

self.num\_vert += 1, new\_vert = vertex(key)

self.vert[key] = new\_vert, return new\_vert

def get\_vert(self, n):

if n in self.vert: return self.vert[n]

def len(self): return self.num\_vert

def contains\_(self, n): return n in self.vert

def add\_edge, get\_vert\_index\_ (略)



扫描全能王 创建

class vertex:

def \_\_init\_\_(self, num):

self.key = num, self.connectTo = {}, self.color = 'white'

self.previous = None,

def \_\_lt\_\_(self, x): return self.key < x.key.

def add\_nbr(self, nbr, weight): self.connectTo[nbr] = weight.

def get\_weighter, def \_\_str\_\_(self) 同前.

#构建骑士周游图.

get\_nbr(self).

def knight\_graph(board\_size):

kt = graph()

for row in range(board\_size): #遍历每一格.

for col in range(board\_size):

node\_id = get\_id(row, col, board\_size)

new\_pos = get\_legal\_moves(row, col, board\_size)

for row2, col2 in new\_pos:

node\_id2 = get\_id(row2, col2, board\_size)

kt.add\_edge(node\_id, node\_id2)

#在骑士周游图中为两格加一条边.

return kt.

def get\_id(x, y, board\_size): return x \* board\_size + y.

def get\_legal\_moves(x, y, boardsize):

new\_move = []

dxy = [(1, -2), (-1, 2), (-2, -1), (-2, 1), (1, -2), (1, 2), (2, -1), (2, 1)]

for dx, dy in dxy:

if 0 <= x+dx < board\_size and 0 <= y+dy < board\_size:

new\_move.append((x+dx, y+dy)) #一定要检查，不能越界

return new\_move

模量.

#下面是基于dfs的周游过程.



扫描全能王 创建

→ 通用深度优先搜索 (深度优先搜索方法)

```
def knight_tour(n, path, u, limit) # u为当前顶点, n为已周游的  
    u.color = "gray" . path.append(u).
```

```
if n < limit: nbrs = ordered(u). # 对所有合法移动依次深入
```

```
i=0
```

```
for nbr in neighbor_nbrs:
```

```
    if nbr.color == "white" and knight_tour(n+1, path, nbr, limit)
```

```
        return True
```

```
    else: path.pop(), u.color = "white", return False.
```

```
else: return True
```

```
def ordered(n): # 后发式算法的关键函数
```

```
lst = []
```

```
for v in n.get_nbr():
```

```
    if v.color == "white": pass
```

```
c=0
```

```
for w in v.get_nbr():
```

```
    if w.color == "white": c+=1
```

```
    lst.append((c, v))
```

```
lst.sort(key=lambda x: x[0])
```

```
return [y[1] for y in lst]
```

# 可使骑士优先访问合理走法少(即棋盘边缘)的顶点,从而尽早访问

骑士到达的角落,并在需要时通过中间的格子跨越到棋盘另一边

```
def pos(id): return (id//8, id%8) # 获得棋盘边长为8时的坐标
```

```
board g = kt(boardSize)
```

```
start = g.get_vert(get_id(startX, startY, boardSize))
```

```
if start is None: print("fail")
```

```
else: tour_path = []
```

```
judge = knight_tour(0, tour_path, start, boardSize**2 - 1)
```

```
print("success" * if judge else "fail").
```



扫描全能王 创建

## 2) 通用深度优先搜索(深度优先森林)

### ① 特点: 创建多棵深度优先搜索树

b. 使用 Vertex 类中的两个额外变量 —— 发现时间 & 结束时间

```
def setDiscovery(self, dtime): self.disc = dtime
def setFinish(self, ftime): self.fin = ftime
def getDiscovery(self): return self.disc
def getFinish(self): return self.fin.
```

发现时间: 记录第一次访问顶点时的步数

结束时间: 记录顶点被标记为黑色时的步数

### ② 实现:

```
class DFSgraph(graph): # DFSgraph 类的对象是 graph 类
```

```
def __init__(self): self.time = 0
```

```
def dfs(self):
```

```
for vert in self: vert.color = "white", vert.previous = -1
```

```
for vert in self:
```

```
if vert.color == "white": self.dfs_visit(vert)
```

```
def dfs_visit(self, start):
```

```
start.color = "gray"
```

走的物理步数

```
self.time = self.time + 1 # 记录算法执行的步数(不是实际)
```

```
start.disc = self.time setDiscovery(time)
```

```
for next_vert in start.getnbr():
```

```
if next_vert.color == "white":
```

# 遍历

```
next_vert.previous = start, self.dfs_visit(next_vert)
```

```
start.color = "black".
```

```
self.time += 1
```

```
start.setFinish(time).
```

a. 与普通 dfs 不同，遍历所有顶点，而不是从一个指定顶点开始搜索。

b. 开始和结束时间相当于“括号搜索树”中的左右括号；因此说构建出了



## 深度优先搜索树.

C. 与 bfs 很类似，但 bfs 用队列储存待搜索的节点，而 dfs 森林用栈  
(虽然程序中没有特意提到，但后开始搜索的节点先搜完，相当于栈.)

## 四. 图的算法.

1. 拓扑排序 将有向无环图转化为“线性图”.

— kahn 算法

(1) 思路 ① 计算每个顶点的入度(指向该顶点的边的数量)

② 初始化一个空列表(result)、一个队列(queue)

③ 将所有入度为 0 的顶点加入 queue.

④ 当 queue 非空时执行：

a. 从 queue 中取出一个顶点 u      b. 将 u 加入 result

c. 对 u 的每个邻接顶点 v, 减少 v 的入度

d. 若 v 的入度变为 0, 则将 v 加入 queue.

⑤ 若 result 长度等于图中顶点数量，则排序完成，返回 result.

2. 代码：(非常类似 bfs).

```
from collections import deque, defaultdict
```

```
def topological_sort(graph):
```

```
    indegree = defaultdict(int)
```

```
    result.append(u)
```

```
    result = []
```

```
    for v in graph[u]:
```

```
        queue = deque()
```

```
        indegree[v] -= 1
```

```
    for u in graph:
```

```
        if indegree[v] == 0:
```

```
            for v in graph[u]:
```

```
                queue.append(v)
```

```
            indegree[v] += 1
```

```
        if len(result) == len(graph):
```

```
            return result
```

```
        if indegree[u] == 0:
```

```
            else: return None
```

```
            queue.append(u)
```

```
        if len(result) < len(graph):
```

```
            while queue:
```

```
                v = queue.popleft()
```

```
                n = queue.popleft()
```

```
                if len(result) < len(graph):
```

```
                    result.append(v)
```



扫描全能王 创建

## 2. Dijkstra 算法。——确定带权图的最短路径(总权重最小)

代码

```
def dijkstra(graph, start): # start is graph's 起点
```

```
    priority_queue = [(v.distance, v) for v in graph]
```

```
    start.distance = 0
```

```
    heapify(pq)
```

```
    visited = {}
```

```
    while pq:
```

```
        distance, current_v = heappop(pq)
```

```
        if current_v in visited and visited[current_v] < distance:
```

```
            continue
```

```
        for next_v in current_v.get_neighbours():
```

(next\_v)

```
            new_distance = current_v.distance + current_v.get_neighbours()
```

```
            if new_distance < next_v.distance:
```

```
                next_v.distance = new_distance
```

```
                next_v.previous = current_v
```

```
                heappush(pq, (next_v.distance, next_v))
```

~~print~~

② 时间复杂度  $O((V+E)\log|V|)$ .

## 3. Prim 算法。

① 最小生成树：对于图  $G=(V, E)$ , 最小生成树是  $E$  的无环子集，且连接  $V$  中的所有顶点。（~~不是~~ 最短路径，因为  $T$  中的边不要求必须完全连接，可以是这样的：）。

② Prim 算法是一种贪心算法，每一步都选择代价最小的下一步（即选择权重最小的边）

· 关键在找到“可添加的边”，即一端是生成树的顶点，另一端是还不在树中的顶点。

## ③ 代码。



```
def prim(graph, start):
    pq = []  # start.distance = 0  # pq 是 priority queue (优先级队列),
    heapq.heappush(pq, (0, start))  # 用堆实现.
    visited = set()  # 访问集合可用 set 实现.

    while pq:
        current_dist, current_vert = heapq.heappop(pq)
        if current_vert in visited: continue  # 直接进入下一轮循环
        visited.add(current_vert)

        for next_vert in current_vert.getnbr():
            weight = current_vert.getweight(next_vert)
            if next_vert not in visited and weight < next_vert.distance:
                next_vert.distance = weight
                next_vert.previous = current_vert
                heapq.heappush(pq, (weight, next_vert))

    # 整体框架仍然与 bfs 相似.
```



扫描全能王 创建

39. 邻接矩阵

inDegree[i] += 1;

return None if put(G[i]) ; seq.append(i);

inDegree[e.v] -= 1; q.put(G[e]); len(seq) == n.

40. 堆排序

arr[S] &gt;= arr[R]; arr[S], arr[i] = arr[i], arr[S],

range(heap-size);

41. 无向图判定

True if total == n else False;

u == x; dfs(u, v); not visited[i].

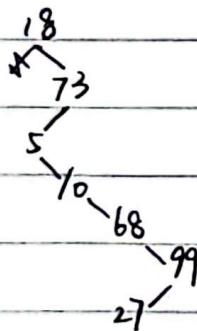
42. 链表操作

next

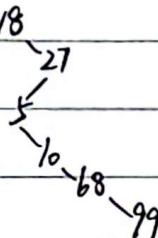
p = Node(a[x]); p.next; p.next = p.next.next;

q = p.next.

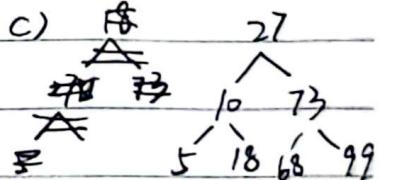
36. a)



b)



c)



37. a) 18, 5, 73, 10, 68, 27, 99, 10; 5, 18, 10, 73, 27, 68, 10, 99;

~~18, 5, 73, 10, 68, 27, 99, 10, 18, 27, 73, 10, 68, 99~~; 5, 10, 18, 27, 10, 73, 68, 99.

b) 不是

c)  $2^{*(n/2)}, 0; 2^{*(n/2)} + n/2; 2^{*(n/2)}$ 38. ~~5~~ {1, 5} {2, 3, 4, 6, 7, 8} 25 ∞ 60 10 17 ∞ ∞+ 2<sup>\*(n/2)</sup>a) {1, 5, 6} {2, 3, 4, 7, 8} 20 ~~20~~ 60 10 17 25 ∞

2 {1, 5, 6, 2} {3, 4, 7, 8} 20 33 60 10 17 25 ∞

7 {1, 5, 6, 2, 7} {3, 4, 8} 20 31 28 10 17 25 35

4 {1, 5, 6, 2, 7, 4} {3, 8} 20 31 28 10 17 25 35

3 {1, 5, 6, 2, 7, 4, 3} {8} 20



扫描全能王 创建

39. Node(\*st[i])

p.next

self.head

p.next

q

p = p.next

40. inorderTraversal(self.left)

inorderTraversal(self.right)

return

BinaryTree(stptr)

tree.left = buildTree()

tree.right = buildTree()

buildTree()



扫描全能王 创建