链表定义

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

栈的经典应用：括号匹配

```
class Solution:
    def isValid(self, s):
        pairs = {
            ")": "(",
            "]": "[",
            "}": "{",
        }
        a = list()
        for ch in s:
            if ch in pairs:
                if not a or a[-1] != pairs[ch]:
                    return False
                a.pop()
            else:
                a.append(ch)

        return not a
```

自定义的矩阵运算

```
def read_matrix():
    try:
        row, col = map(int, input().split())
        matrix = []
        for _ in range(row):
            matrix.append(list(map(int, input().split())))
        return matrix, row, col
    except:
        return None, 0, 0
def matrix_multiply(A, B):
    """矩阵乘法 A @ B"""
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    if cols_A != rows_B:
        return None
    result = [[0] * cols_B for _ in range(rows_A)]
    for i in range(rows_A):
        for j in range(cols_B):
```

```
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]
    return result
def matrix_add(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    if rows_A != rows_B or cols_A != cols_B:
        return None
    result = [[0] * cols_A for _ in range(rows_A)]
    for i in range(rows_A):
        for j in range(cols_A):
            result[i][j] = A[i][j] + B[i][j]

    return result
```

链表：双指针，反转

```
class Solution(object):
    def getIntersectionNode(self, headA, headB):
        if not headA or not headB:
            return None
        p1, p2 = headA, headB
        while p1 != p2:
            p1 = p1.next if p1 else headB
            p2 = p2.next if p2 else headA
        return p1
```

```
class Solution(object):
    def reverseList(self, head):
        """
        :type head: Optional[ListNode]
        :rtype: Optional[ListNode]
        """
        prev = None
        current = head
        while current:
            next_node = current.next  # 保存下一个节点
            current.next = prev       # 反转指针
            prev = current            # 移动prev
            current = next_node       # 移动current
        return prev
```

卡特兰数：

```
def stack_permutations_memo(n):
    """
    使用记忆化递归计算
```

```
    """
    memo = {}
    def dfs(k):
        if k <= 1:
            return 1
        if k in memo:
            return memo[k]
        total = 0
        for i in range(k):
            left = dfs(i)
            right = dfs(k - 1 - i)
            total += left * right
        memo[k] = total
        return total
    return dfs(n) if n >= 0 else 0
```

二叉树

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class BSTNode(TreeNode):
    """二叉搜索树节点"""
    def __init__(self, val=0, left=None, right=None):
        super().__init__(val, left, right)

class BinarySearchTree:
    """二叉搜索树"""
    def __init__(self):
        self.root = None
    def insert(self, val):
        """插入节点"""
        if not self.root:
            self.root = BSTNode(val)
            return
        current = self.root
        while True:
            if val < current.val:
                if not current.left:
                    current.left = BSTNode(val)
                    break
                current = current.left
            elif val > current.val:
                if not current.right:
                    current.right = BSTNode(val)
                    break
                current = current.right
            else:
                # 值已存在，根据需求处理
                break
```

```
    def search(self, val):
        """查找节点"""
        current = self.root
        while current:
            if val == current.val:
                return current
            elif val < current.val:
                current = current.left
            else:
                current = current.right
        return None
    def delete(self, val):
        """删除节点"""
        def find_min(node):
            while node.left:
                node = node.left
            return node
        def delete_node(node, key):
            if not node:
                return None
            if key < node.val:
                node.left = delete_node(node.left, key)
            elif key > node.val:
                node.right = delete_node(node.right, key)
            else:
                # 找到要删除的节点
                if not node.left:
                    return node.right
                elif not node.right:
                    return node.left
                else:
                    # 有两个子节点
                    min_node = find_min(node.right)
                    node.val = min_node.val
                    node.right = delete_node(node.right, min_node.val)
            return node
        self.root = delete_node(self.root, val)
```

逆波兰表达式

```
class TreeNode:
    def __init__(self, val=None, left=None, right=None,symbol=None):
        self.val = val
        self.left = left
        self.right = right
def postorder(sentence):
    operators=[]
    ans=[]
    s=sentence.split()
    weight={'not':3,'and':2,'or':1,'(':0}
    for word in s:
        if word in ('True','False'):
```

```
            ans.append(word)
        elif word=='(':
            operators.append(word)
        elif word==')':
            while operators and operators[-1]!='(':
                ans.append(operators.pop())
            operators.pop()
        else:
            while operators and weight[operators[-1]]>=weight[word]:
                ans.append(operators.pop())
            operators.append(word)
    while operators:
        ans.append(operators.pop())
    return ans
def buildTree(inlist):
    stack=[]
    for word in inlist:
        if word in ('True','False'):
            stack.append(TreeNode(word))
        elif word=='not':
            a=stack.pop()
            stack.append(TreeNode('not',a))
        else:
            a=stack.pop()
            b=stack.pop()
            stack.append(TreeNode(word,b,a))
    return stack[0]
level={'or':1,'and':2,'not':3,'True':4,'False':4}
def printTree(root):#返回中序输出
    if root.val in ('True','False'):
        return [root.val]
    elif root.val=='not':
        a=root.left
        b=printTree(a)
        if level[a.val]<level[root.val]:
            b=['(']+b+[')']
        return ['not']+b
    else:
        l=root.left
        r=root.right
        lefttree=printTree(l)
        righttree=printTree(r)
        if level[l.val]<level[root.val]:
            lefttree=['(']+lefttree+[')']
        if level[r.val]<level[root.val]:
            righttree=['(']+righttree+[')']
        return lefttree+[root.val]+righttree
ini=input().strip()
ansl=postorder(ini)
ans2=printTree(buildTree(ansl))
print(' '.join(ans2))
```

堆的创建与转换

操作	函数	时间复杂度	示例	说明
创建堆	`heapq.heapify(lst)`	O(n)	`heapq.heapify([3,1,4])` → `[1,3,4]`	原地转换列表为堆
添加元素	`heapq.heappush(heap, item)`	O(log n)	`heapq.heappush(h, 2)`	添加元素并保持堆属性
弹出最小	`heapq.heappop(heap)`	O(log n)	`heapq.heappop(h)`	弹出并返回最小元素
压入+弹出	`heapq.heappushpop(heap, item)`	O(log n)	`heapq.heappushpop(h, 5)`	先push后pop，比分开调用高效
弹出+压入	`heapq.heapreplace(heap, item)`	O(log n)	`heapq.heapreplace(h, 5)`	先pop后push，堆非空时使用

查询与排序

操作	函数	时间复杂度	示例	说明
查看最小	`heap[0]`	O(1)	`min_val = heap[0]`	查看但不弹出最小元素
前N大	`heapq.nlargest(n, iter)`	O(k + n log k)	`heapq.nlargest(3, nums)`	返回前n个最大元素
前N小	`heapq.nsmallest(n, iter)`	O(k + n log k)	`heapq.nsmallest(3, nums)`	返回前n个最小元素
合并排序	`heapq.merge(*iterables)`	O(k log k)	`list(heapq.merge(lst1, lst2))`	合并多个已排序输入

deque创建与基本属性

操作	方法/属性	时间复杂度	示例	说明
创建deque	`deque(iterable, maxlen)`	O(n)	`d = deque([1,2,3])`	创建双端队列
最大长度	`maxlen` 属性	O(1)	`d.maxlen`	返回最大长度(只读)
长度	`len(deque)`	O(1)	`len(d)`	元素个数
是否为空	布尔转换	O(1)	`if d:`	空为False

左端操作 (Left End)

操作	方法	时间复杂度	示例	说明
左端添加	`appendleft(x)`	O(1)	`d.appendleft(0)`	添加到左侧
左端弹出	`popleft()`	O(1)	`x = d.popleft()`	从左侧弹出
左端扩展	`extendleft(iterable)`	O(k)	`d.extendleft([-1,-2])`	逆序添加到左侧

➡️ 右端操作 (Right End)

操作	方法	时间复杂度	示例	说明
右端添加	`append(x)`	O(1)	`d.append(4)`	添加到右侧
右端弹出	`pop()`	O(1)	`x = d.pop()`	从右侧弹出
右端扩展	`extend(iterable)`	O(k)	`d.extend([5,6])`	添加到右侧

🔍 查询与访问

操作	方法	时间复杂度	示例	说明
索引访问	`deque[i]`	O(1)	`x = d[0]`	支持正负索引
索引赋值	`deque[i]=x`	O(1)	`d[0] = 100`	修改元素
计数	`count(x)`	O(n)	`d.count(1)`	统计出现次数
查找索引	`index(x, start, end)`	O(n)	`d.index(2)`	查找元素位置
是否包含	`in` 操作符	O(n)	`if 2 in d:`	成员检查

bfs

```
def bfs_with_path(start, target):
    """记录路径的BFS"""
    queue = deque([start])
    parent = {start: None}  # 记录父节点（路径回溯）
    visited = set([start])
    while queue:
        current = queue.popleft()
        if current == target:
            # 重构路径
            path = []
            while current is not None:
                path.append(current)
                current = parent[current]
            return path[::-1]  # 反转得到正向路径
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current  # 记录父节点
                queue.append(neighbor)
    return []  # 无路径
```

运算符	符号	名称	示例 (a=5=0101, b=3=0011)	结果	说明
与	`&`	AND	`a & b`	`0001` (1)	两位都为1才为1
或	`|`	OR	`a | b`	`0111` (7)	有一位为1就为1
异或	`^`	XOR	`a ^ b`	`0110` (6)	两位不同为1
取反	`~`	NOT	`~a`	`1010` (-6)	每位取反（补码）
左移	`<<`	SHL	`a << 1`	`1010` (10)	左移n位，低位补0
右移	`>>`	SHR	`a >> 1`	`0010` (2)	右移n位，高位补符号位
无符号右移	Python无	USHR	Python无	-	Python无此运算符

操作	代码	说明
判断奇偶	`x & 1`	0为偶，1为奇
取最低位1	`x & -x`	得到最低位的1
清零最低位1	`x & (x-1)`	将最低位的1变为0
判断2的幂	`x & (x-1) == 0`	2的幂次方
交换两数	`a ^= b; b ^= a; a ^= b`	不用临时变量

🔧 位操作组合技

操作	代码	说明
检查第k位	`(x >> k) & 1`	第k位是否为1
设置第k位为1	`x | (1 << k)`	将第k位置1
设置第k位为0	`x & ~(1 << k)`	将第k位置0
翻转第k位	`x ^ (1 << k)`	第k位取反
最低位1的位置	`(x & -x).bit_length()-1`	从0开始计数

字典树

```
class Trie:
    """完整字典树实现"""
    def __init__(self):
        self.root = TrieNode()
    # ---------- 核心操作 ----------
    def insert(self, word: str) -> None:
        """插入单词"""
```

```
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            node.prefix_count += 1  # 前缀计数增加
        node.is_end = True
        node.word_count += 1
    def search(self, word: str) -> bool:
        """精确搜索单词"""
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end
    def startsWith(self, prefix: str) -> bool:
        """是否存在以prefix为前缀的单词"""
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
    # ---------- 扩展操作 ----------
    def delete(self, word: str) -> bool:
        """删除单词（惰性删除）"""
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        if node.is_end:
            node.is_end = False
            node.word_count = 0
            return True
        return False
    def count_words_with_prefix(self, prefix: str) -> int:
        """统计以prefix为前缀的单词数量"""
        node = self.root
        for char in prefix:
            if char not in node.children:
                return 0
            node = node.children[char]
        return node.prefix_count
    def get_all_words(self):
        """获取所有单词"""
        result = []
        def dfs(node, path):
            if node.is_end:
                result.append("".join(path))
            for char, child in node.children.items():
                path.append(char)
                dfs(child, path)
```

```
                path.pop()
        dfs(self.root, [])
        return result
```

并查集

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
    def find(self, x):
        # 路径压缩
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            self.parent[root_y] = root_x
            return True
        return False
```

拓扑排序

```
from collections import deque

def topological_sort_kahn(n, edges):
    """
    n: 顶点数
    edges: 边的列表，每个元素为 (u, v) 表示 u → v
    """
    # 构建邻接表和入度数组
    graph = [[] for _ in range(n)]
    indegree = [0] * n

    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1

    # 初始化队列（所有入度为0的顶点）
    queue = deque([i for i in range(n) if indegree[i] == 0])
    result = []

    # BFS
    while queue:
        node = queue.popleft()
        result.append(node)

        # 处理该节点的所有邻居
```

```
            for neighbor in graph[node]:
                indegree[neighbor] -= 1
                if indegree[neighbor] == 0:
                    queue.append(neighbor)

        # 检查是否有环
        if len(result) != n:
            return []  # 有环，不存在拓扑排序

        return result
```

语法	名称	匹配内容	示例	匹配结果
`[abc]`	字符集	a、b或c中任意一个	`[aeiou]`	匹配任意元音字母
`[^abc]`	否定字符集	除a、b、c外的任意字符	`[^0-9]`	匹配非数字字符
`[a-z]`	字符范围	a到z的任意小写字母	`[A-Za-z]`	匹配任意字母
`\d`	数字	等价于 `[0-9]`	`\d\d`	"12", "45"
`\D`	非数字	等价于 `[^0-9]`	`\D+`	"abc", "XYZ"
`\w`	单词字符	字母、数字、下划线	`\w+`	"hello_123"
`\W`	非单词字符	非字母、数字、下划线	`\W`	"@", " "
`\s`	空白字符	空格、制表符、换行等	`\s+`	" \t\n"
`\S`	非空白字符	非空白字符	`\S+`	"Hello"
`.`	任意字符	除换行外的任意字符	`a.c`	"abc", "a c"

语法	名称	含义	示例	匹配
`?`	问号	0次或1次（可选）	`colou?r`	"color", "colour"
`*`	星号	0次或多次	`ab*c`	"ac", "abc", "abbc"
`+`	加号	1次或多次	`ab+c`	"abc", "abbc" (不匹配"ac")
`{n}`	精确匹配	恰好n次	`\d{3}`	"123", "456"
`{n,}`	至少n次	n次或更多次	`\d{2,}`	"12", "123", "1234"
`{n,m}`	范围匹配	n到m次	`\d{2,4}`	"12", "123", "1234"

4. 位置锚点

语法	名称	匹配位置	示例	匹配位置
`^`	行首	字符串开始或行开始	`^Start`	匹配行首的"Start"
`$`	行尾	字符串结束或行结束	`end$`	匹配行尾的"end"
`\b`	单词边界	单词的开始或结束	`\bcat\b`	匹配"cat"不匹配"category"
`\B`	非单词边界	非单词边界	`\Bcat\B`	匹配"education"中的"cat"

语法	匹配	说明
`\\`	反斜杠	匹配字面反斜杠
`\.`	点号	匹配字面点号
`\*`	星号	匹配字面星号
`\+`	加号	匹配字面加号
`\?`	问号	匹配字面问号
`\(`	左括号	匹配字面左括号
`\)`	右括号	匹配字面右括号
`\[`	左方括号	匹配字面左方括号
`\]`	右方括号	匹配字面右方括号

7. 常用模式示例

用途	正则表达式	说明
邮箱地址	`\w+@\w+\.\w+`	简单邮箱匹配
手机号码	`1[3-9]\d{9}`	中国大陆手机号
身份证号	`\d{17}[\dXx]`	18位身份证号
中文汉字	`[\u4e00-\u9fa5]`	匹配单个汉字
日期	`\d{4}-\d{2}-\d{2}`	YYYY-MM-DD格式
IP地址	`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`	IPv4地址
URL	`https?://[^\s]+`	HTTP/HTTPS链接
HTML标签	`<[^>]+>`	简单标签匹配

一、`re` 模块核心函数

函数	作用	返回值	常用场景
`re.search()`	在字符串中搜索第一个匹配	Match对象或None	查找是否存在某个模式
`re.match()`	从字符串开头匹配	Match对象或None	验证字符串是否以某模式开头
`re.findall()`	查找所有匹配	列表（字符串或元组）	提取所有符合条件的内容
`re.finditer()`	查找所有匹配（迭代器）	Match对象迭代器	需要获取匹配位置时
`re.sub()`	替换匹配内容	替换后的字符串	文本替换
`re.split()`	按模式分割字符串	分割后的列表	复杂分割
`re.compile()`	编译正则表达式	Pattern对象	重复使用同一模式