```
#输出格式化：s.format(1,2):替换大括号
#          name = "Alice",age = 25,print(f"My name is {name} and I'm {age} years old.")
#          f"anything{x:.nf}"
#          进制：f"{x:.b/o/x}"
#print(objects,sep=''(单次 print）,end=''(循环 print))

#字符串操作：\n,s.lower(),s.upper(),s.strip(),s.split(sep)
#，sep.join(list),s.replace(old,new(,count)),s[1:2:step]

#list 操作:list.sort(key=type/lambda x:f(x)/(x[1],x[0]),reverse=True)
#sum,max,min,del s[i],insert(i,x),s.pop().s.remove(),s.count(x)
#创建矩阵：matrix = [[''for _ in range(cols)] for _ in range(rows)]
#dic 操作:注意无顺序
#      dic[key]=value,key in dic,del dic[key],
#      dic.get(k,default),value=dic.pop(k,default）
#      list(dic)->list(keys),dic.keys()/values()/items()->生成迭代对象，用 list()汇总

#集合：set():s<=s1:子集判断，s1|s2:并集，s1&s2:交集,s1-s2:差集,s.add(x),s.remove(x)
#     s.discard(x),s.pop()
#定义函数：def f(a,b=x,c),or f(*args,*kargs)->当作元组处理
#类：
class Force:
    def __init__(self, fx, fy):
        self.fx = fx
        self.fy = fy

    def show(self):          #(self 为参数的函数，即为 f1.f())
        print((self.fx, self.fy))
#另一种方法：定义类的字符串表示（魔术方法）
    def __str__(self):
        return f"Point({self.fx}, {self.fy})"

    def add(self,f2):
        self.fx += f2.fx
        self.fy += f2.fy
        return self          #更改被操作对象自己的值
f1=Force(0,1)
f1.show()
f2=Force(1,0)
f1.add(f2)
f1.show()
print(f1)
```

线性结构

#顺序表 array：

#单链表：简单的单链表

#整个单链表：head 头指针

#第一个结点：head

空表判断：head==NULL

当前结点 a1：curr 尾指针（有的话）：tail 从当前一个元素很容易找到下一个元素，反之很麻烦顺序表正反都很容易

如果要在 curr 之前插入一个元素怎么办

list 实现顺序表 / 动态数组

```
arr = [1, 2, 3]
arr.append(4)          # 末尾添加
arr.insert(1, 99)      # 在索引 1 插入
arr.remove(2)          # 移除值为 2 的元素
arr.pop(0)             # 按索引删除
arr[1] = 100           # 修改索引值
arr.index(100)         # 查找元素位置
```

collections.deque 实现 Queue、Deque

```
from collections import deque
# 队列 FIFO
queue = deque()
queue.append('A')      # 入队 enqueue
queue.popleft()        # 出队 dequeue
```

 # 双端队列

```
dq = deque()
dq.append(1)           # 从右加入
dq.appendleft(0)       # 从左加入
dq.pop()               # 从右弹出
dq.popleft()           # 从左弹出
# 公共操作
len(dq)                # 长度
not dq                 # 是否为空
dq[0], dq[-1]          # 查看两端元素
```

heapq 实现最小堆 / 优先队列

```
import heapq
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)
heapq.heappop(heap)        # -> 1，最小值出队
```

```
heap[0]                    # 查看最小值
#括号匹配
def f(s:str):
    stack= []
    pairs = {')': '(', ']': '[', '}': '{'}
    for c in s:
        if c in pairs.values():
            stack.append(c)
        elif not stack or stack.pop() != pairs[c]:
            return False
    return not stack
#前中后缀表达式：
#中->前：全括号表达式将运算符移至左括号处删除右括号即可，反之亦然
#前缀转中缀，从左往右扫描，每次看到一个操作符和两个操作数，就替换为一个带括号的“操作数”，不断进行，直到全部变成中缀表达式
#后缀转中缀，从左往右扫描，每次看到两个操作数和一个操作符，就替换为一个带括号的“操作数”，不断进行，直到全部变成中缀表达式
def infix_to_postfix(expr):
    # 定义运算符优先级
    prec = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack, output = [], []
    # 去除空格并逐个处理字符
    for c in expr.replace(' ', ''):
        if c.isalnum():  # 如果是操作数（字母或数字）
            output.append(c)
        elif c == '(':  # 左括号直接入栈
            stack.append(c)
        elif c == ')':  # 右括号处理
            while stack and stack[-1] != '(':  # 弹出直到遇到左括号
                output.append(stack.pop())
            stack.pop()  # 弹出左括号但不输出
        else:  # 运算符处理
            # 弹出优先级≥当前运算符的栈顶运算符
            while stack and stack[-1] != '(' and prec[c] <= prec[stack[-1]]:
                output.append(stack.pop())
            stack.append(c)  # 当前运算符入栈
    # 将栈中剩余运算符全部弹出
    return output + stack[::-1]
#前缀：关键区别说明
#扫描方向：从右向左扫描表达式
#括号处理：遇到 ) 压栈（相当于后缀中的 ()
#        遇到 ( 弹出直到 )（相当于后缀中的 )）
#优先级比较：使用 < 而不是 <=（保持相同优先级运算符的原顺序）
#结果反转：最终需要反转输出列表
```

```
#后缀转中缀：
def postfix_to_infix(postfix_expr):
    stack = []
    operators = {'+', '-', '*', '/', '^'}
    for token in postfix_expr.split():
        if token not in operators:
            stack.append(token)
        else:
            right = stack.pop()
            left = stack.pop()
            # 根据优先级决定是否加括号
            stack.append(f"({left} {token} {right})")
    return stack[0] if stack else ""
def postfix_to_infix_optimized(postfix_expr):
    stack = []
    prec = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    for token in postfix_expr.split():
        if token not in prec:
            stack.append((token, 0))  # (表达式, 当前优先级)
        else:
            right, right_prec = stack.pop()
            left, left_prec = stack.pop()
            current_prec = prec[token]
            # 左操作数需要括号的情况
            if left_prec and left_prec < current_prec:
                left = f"({left})"
            # 右操作数需要括号的情况（注意减法和除法特殊处理）
            if (right_prec and right_prec < current_prec) or \
                    (right_prec == current_prec and token in {'-', '/'}):
                right = f"({right})"
            stack.append((f"{left} {token} {right}", current_prec))
    return stack[0][0] if stack else ""
#前转中反向即可
def prefix_to_infix_optimized(prefix_expr):
    stack = []
    prec = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    # 从右向左扫描
    for token in reversed(prefix_expr.split()):
        if token not in prec:
            stack.append((token, 0))  # (表达式, 当前优先级)
        else:
            left, left_prec = stack.pop()
            right, right_prec = stack.pop()
            current_prec = prec[token]
```

```
            # 左操作数需要括号的情况
            if left_prec and left_prec < current_prec:
                left = f"({left})"
            # 右操作数需要括号的情况（注意减法和除法特殊处理）
            if (right_prec and right_prec < current_prec) or \
                    (right_prec == current_prec and token in {'-', '/', '^'}):
                right = f"({right})"
            stack.append((f"{left} {token} {right}", current_prec))
    return stack[0][0] if stack else ""
#后缀求值：
def evaluate_postfix(expression):
    stack = []
    operators = {'+', '-', '*', '/', '^'}
    for token in expression.split():
        if token not in operators:
            stack.append(float(token))  # 支持浮点数
        else:
            right = stack.pop()
            left = stack.pop()
            if token == '+':
                stack.append(left + right)
            elif token == '-':
                stack.append(left - right)
            elif token == '*':
                stack.append(left * right)
            elif token == '/':
                stack.append(left / right)
            elif token == '^':
                stack.append(left ** right)
    return stack[0] if stack else 0
#前缀：反向，弹出顺序相反，中缀：转换为后缀求值
```

递归

```
#1，递归算法必须有一个基本结束条件（最小规模问题的直接解决）
#2，递归算法必须能改变状态向基本结束条件演进（减小问题规模）
#3，递归算法必须调用自身（解决减小了规模的相同问题）
#进制转换
def conversion(n,base):
    converString = '0123456789ABCDEF'
    if n < base:
        return converString[n]
    else:
        return conversion(n//base,base) + converString[n%base]
#河内塔：
```

```
#将盘片塔从开始柱，经由中间柱，移动到目标柱：
#先将上层 N-1 个盘片的盘片塔，从开始柱，经由目标柱，移动到中间柱；
#然后将第 N 个（最大的）盘片，从开始柱，移动到目标柱；
#最后将放置在中间柱的 N-1 个盘片的盘片塔，经由开始柱，移动到目标柱。
def moveTower(height,fromPole,toPole,withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
def moveDisk(fromPole,toPole):
    print(fromPole+'->'+toPole)
moveTower(3,'A','B','C')
#兑换硬币的递归，cache，dp
import sys
def recMC(coinValueList,change):
    if change in coinValueList:
        return 1
    ntry = [reMC(coinValueList,change-c) for c in coinValueList if c<=change]
    return 1+min(ntry,default= sys.maxsize)
def recMC2(coinValueList,change,known):
    if known[change] > 0:
        return known[change]
    elif change in coinValueList:
        known[change] = 1
        return 1
    ntry = [recMC2(coinValueList,change-c,known) for c in coinValueList if
c<=change]
    known[change] = 1 + min(ntry,default= sys.maxsize)
    return known[change]
print(recMC2([1,5,10,25],63,[0]*64))
#DP:兑换硬币的动态规划算法从最简单的“1 分钱找零”的最优解开始，逐步递加上去，直到
我们需要的找零钱数
#还是归纳法：a1 显而易见；在 ai,1≤i≤n-1 已知的情况下，求解 an
def dpMC(coinValueList,change):
    minCoins = [0]*(change+1)
    for cents in range(1,change+1):
        ntry = [minCoins[cents-c] for c in coinValueList if c<=cents]
        minCoins[cents] = 1 + min(ntry,default= sys.maxsize)
    return minCoins[change]
print(dpMC([1,5,10,25],63))
#动态规化问题的一般套路
#把原问题扩展为具有最优子结构的问题 Q。
#一维情景：Q=Q(n),二维情景：Q=Q(n,m)
#找到 Q 的递归推导式：
```

#一维情景：Q(n)=F(Q(n1),Q(n2),...,Q(ni)),n1,n2,...,ni < n
#二维情景：Q(n,m) = F(Q(ni,mj),...),(ni < n ∧ mj ⩽ m)∨(ni ⩽ n∧mj < m)
#当 n 和 m 很小的时候，Q 易得
#算法实现
#动态规划：用一个数组 dp 存储 Q 的结果，从“前”往“后”算
#递归：用递归函数“原则上也可以”实现
例题：
线性 01 背包：

```
def maxvalue(T, M, herbs):
    dp = [0]* (T + 1)
    for t, v in herbs:
        for j in range(T, t - 1, -1):
            dp[j] = max(dp[j], dp[j - t] + v)
    return dp[T]
T, M = map(int, input().split())
herbs =[tuple(map(int, input().split())) for _ in range(M)]
print(maxvalue(T, M, herbs))
```

线性：

```
def mi(mis):
    N = len(mis)
    dp = [1]*N
    for i in range(1,N):
        for j in range(i):
            if mis[j]>=mis[i]:
                dp[i]=max(dp[i],dp[j]+1)
    return max(dp)
N = int(input().strip())
mis = list(map(int, input().split()))
print(mi(mis))
def ma(s1: str, s2: str) -> int:
    m, n = len(s1), len(s2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]


s1 = input().strip()
```

```
s2 = input().strip()
print(ma(s1, s2))
```

无序二维划分：

```
def partition_count(n):
    dp = [[0] * (n + 1) for _ in range(n + 1)]



    for j in range(n + 1):
        dp[0][j] = 1

    for i in range(1, n + 1):
        for j in range(1, n + 1):
            dp[i][j] = dp[i][j - 1]
            if i >= j:
                dp[i][j] += dp[i - j][j]

    return dp[n][n]
```

查找

#1.顺序查找：略

#2 二分查找：若为给定列表，使用：

```
import bisect
a = [1,4,6,7,99,99,191]
print(bisect.bisect_left(a, 99))
print(bisect.bisect_right(a, 99))
#给定范围满足某条件：传统方法
n , k = map(int, input().split())
logs = []
for i in range(n):
    logs.append(int(input()))
s = sum(logs)
result = 0
if s<k:
    print(0)
else:
    maxlen0 = s//k
    l,r = 1,maxlen0
    while l<=r:
        mid = (l+r)//2
        k1 = sum(i//mid for i in logs)
        if k1>=k:
```

```
            result = mid
            l = mid+1
        else:
            r = mid-1
    print(result)
#散列：根据某种关系一一对应的表
#稳定：冒泡、插入、归并、分配、sort
#不稳定：选择、希尔、快速
```

图与算法常用模板代码参考

1. 图的邻接表表示

```
class Graph:
    def __init__(self):
        self.adj = {}
    def add_edge(self, u, v, weight=1):
        self.adj.setdefault(u, []).append((v, weight))
    def get_neighbors(self, u):
        return self.adj.get(u, [])
```

2. Word Ladder 构图与 BFS 最短路径

```
from collections import deque, defaultdict

def build_graph(words):
    buckets = defaultdict(list)
    for word in words:
        for i in range(len(word)):
            key = word[:i] + "_" + word[i+1:]
            buckets[key].append(word)

    graph = Graph()
    for bucket in buckets.values():
        for i in range(len(bucket)):
            for j in range(i + 1, len(bucket)):
                graph.add_edge(bucket[i], bucket[j])
                graph.add_edge(bucket[j], bucket[i])
    return graph
```

```python
def bfs(graph, start, end):
    visited = set()
    parent = {start: None}
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        if node == end:
            break
        for neighbor, _ in graph.get_neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = node
                queue.append(neighbor)
    return parent

def reconstruct_path(parent, end):
    path = []
    while end:
        path.append(end)
        end = parent[end]
    return path[::-1]
```

3. 骑士周游 Knight's Tour（DFS + 回溯）

```python
def knight_moves(x, y, board_size):
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1),
             (-2, -1), (-1, -2), (1, -2), (2, -1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < board_size and 0 <= ny < board_size:
            yield nx, ny

def knight_tour(x, y, board_size, path, visited):
    if len(path) == board_size * board_size:
```

```
        return True
    for nx, ny in knight_moves(x, y, board_size):
        if (nx, ny) not in visited:
            visited.add((nx, ny))
            path.append((nx, ny))
            if knight_tour(nx, ny, board_size, path, visited):
                return True
            path.pop()
            visited.remove((nx, ny))
    return False
```

4. 拓扑排序 Topological Sort (DFS)

```
def topological_sort(graph):
    visited = set()
    result = []

    def dfs(u):
        visited.add(u)
        for v, _ in graph.get_neighbors(u):
            if v not in visited:
                dfs(v)
        result.append(u)

    for node in graph.adj:
        if node not in visited:
            dfs(node)
    return result[::-1]
```

5. 强连通分支 Kosaraju 算法

```
def transpose(graph):
    t_graph = Graph()
    for u in graph.adj:
        for v, _ in graph.get_neighbors(u):
            t_graph.add_edge(v, u)
```

```
    return t_graph

def kosaraju_scc(graph):
    stack, visited = [], set()

    def dfs(u):
        visited.add(u)
        for v, _ in graph.get_neighbors(u):
            if v not in visited:
                dfs(v)
        stack.append(u)

    for node in graph.adj:
        if node not in visited:
            dfs(node)

    t_graph = transpose(graph)
    visited.clear()
    scc_list = []

    def dfs2(u, comp):
        visited.add(u)
        comp.append(u)
        for v, _ in t_graph.get_neighbors(u):
            if v not in visited:
                dfs2(v, comp)

    while stack:
        u = stack.pop()
        if u not in visited:
            comp = []
            dfs2(u, comp)
            scc_list.append(comp)
    return scc_list
```

6. 最短路径 Dijkstra 算法

```python
import heapq

def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph.adj}
    dist[start] = 0
    heap = [(0, start)]

    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
            continue
        for v, w in graph.get_neighbors(u):
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                heapq.heappush(heap, (dist[v], v))
    return dist
```

7. Prim 最小生成树

```python
def prim_mst(graph, start):
    visited = set()
    mst = []
    heap = [(0, start, None)]

    while heap:
        weight, u, prev = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        if prev is not None:
            mst.append((prev, u, weight))
        for v, w in graph.get_neighbors(u):
            if v not in visited:
                heapq.heappush(heap, (w, v, u))
```

```
    return mst
```

8. BFS 和 DFS 模板（独立）

```
from collections import deque

def bfs(graph, start):
    visited = set()
    parent = {start: None}
    distance = {start: 0}
    queue = deque([start])
    visited.add(start)
    while queue:
        u = queue.popleft()
        for v in graph.get(u, []):
            if v not in visited:
                visited.add(v)
                parent[v] = u
                distance[v] = distance[u] + 1
                queue.append(v)
    return parent, distance

def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for v in graph.get(start, []):
        if v not in visited:
            dfs_recursive(graph, v, visited)

def dfs_iterative(graph, start):
    visited = set()
    stack = [start]
    while stack:
        u = stack.pop()
        if u not in visited:
            visited.add(u)
            stack.extend(reversed(graph.get(u, [])))
```

第七章 树与算法——参考代码汇总

一、二叉树的实现

1. 嵌套列表法

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root, newBranch):
    t = root.pop(1)
    if t:
        root.insert(1, [newBranch, t, []])
    else:
        root.insert(1, [newBranch, [], []])
    return root

def insertRight(root, newBranch):
    t = root.pop(2)
    if t:
        root.insert(2, [newBranch, [], t])
    else:
        root.insert(2, [newBranch, [], []])
    return root

def getRootVal(root): return root[0]
def setRootVal(root, val): root[0] = val
def getLeftChild(root): return root[1]
def getRightChild(root): return root[2]
```

2. 结点链接法（类实现）

```
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

```python
    def insertLeft(self, newNode):
        if self.leftChild:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t
        else:
            self.leftChild = BinaryTree(newNode)

    def insertRight(self, newNode):
        if self.rightChild:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t
        else:
            self.rightChild = BinaryTree(newNode)

    def getRightChild(self): return self.rightChild
    def getLeftChild(self): return self.leftChild
    def setRootVal(self, obj): self.key = obj
    def getRootVal(self): return self.key
```

还可实现普通树：

```python
class TreeNode:
    """树节点类"""
    def __init__(self, data):
        self.data = data  # 节点数据
        self.children = []  # 子节点列表
        self.parent = None  # 父节点引用
    def add_child(self, child):
        """添加子节点"""
        child.parent = self  # 设置子节点的父节点为当前节点
        self.children.append(child)
    def get_level(self):
        """获取节点在树中的层级(根节点为0级)"""
        level = 0
        p = self.parent
        while p:
            level += 1
            p = p.parent
        return level
```

```
    def print_tree(self):
        """打印树结构"""
        indent = ' ' * self.get_level() * 3
        prefix = indent + "|__" if self.parent else ""
        print(prefix + self.data)
        for child in self.children:
            child.print_tree()
    def search(self, target_data):
        查询树中是否存在包含目标数据的节点
        返回布尔值表示是否存在
        if self.data == target_data:
            return True
        for child in self.children:
            if child.search(target_data):
                return True
        return False
# 示例：构建树
def build_product_tree():
    root = TreeNode("电子产品")
    laptop = TreeNode("笔记本电脑")
    laptop.add_child(TreeNode("MacBook"))
    laptop.add_child(TreeNode("Surface"))
    laptop.add_child(TreeNode("ThinkPad"))
    root.add_child(laptop)
    return root
if __name__ == '__main__':
    root = build_product_tree()
    root.print_tree()
    # 输出:
    # 电子产品
    #    |__笔记本电脑
    #       |__MacBook
    #       |__Surface
    #       |__ThinkPadT
 from collections import deque

class TreeNode:
    def __init__(self, val):
        self.val = val
        self.children = []
        self.parent = None
        self.depth = 0  # Added depth to help with LCA finding
    def add_child(self, child):
        self.children.append(child)
```

```
        child.parent = self
        child.depth = self.depth + 1
def build_tree(n, R):
    nodes = {}
    root = TreeNode(R)
    nodes[R] = root
    dq = deque()
    for _ in range(n - 1):
        A, B = map(int, input().split())
        a_exists = A in nodes
        b_exists = B in nodes
        if a_exists and b_exists:
            continue
        elif a_exists:
            b_node = TreeNode(B)
            nodes[A].add_child(b_node)
            nodes[B] = b_node
        elif b_exists:
            a_node = TreeNode(A)
            nodes[B].add_child(a_node)
            nodes[A] = a_node
        else:
            dq.append((A, B))
    while dq:
        A, B = dq.popleft()
        a_exists = A in nodes
        b_exists = B in nodes
        if a_exists and b_exists:
            continue
        elif a_exists:
            b_node = TreeNode(B)
            nodes[A].add_child(b_node)
            nodes[B] = b_node
        elif b_exists:
            a_node = TreeNode(A)
            nodes[B].add_child(a_node)
            nodes[A] = a_node
        else:
            dq.append((A, B))
    return root, nodes
def find_lca(nodes, a, b):
    node_a = nodes.get(a)
    node_b = nodes.get(b)
    if not node_a or not node_b:
```

```python
            return -1  # One or both nodes don't exist
        while node_a.depth > node_b.depth:
            node_a = node_a.parent
        while node_b.depth > node_a.depth:
            node_b = node_b.parent
        while node_a != node_b:
            node_a = node_a.parent
            node_b = node_b.parent
        return node_a.val
    n, R = map(int, input().split())
    root, nodes = build_tree(n, R)
    m = int(input())
    for _ in range(m):
        x, y = map(int, input().split())
        print(find_lca(nodes, x, y))
```

二、表达式解析树

1. 构建解析树

```python
from pythonds.basic.stack import Stack
from pythonds.trees.binaryTree import BinaryTree

def buildParseTree(expr):
    tokens = expr.split()
    stack = Stack()
    tree = BinaryTree('')
    stack.push(tree)
    current = tree

    for t in tokens:
        if t == '(':
            current.insertLeft('')
            stack.push(current)
            current = current.getLeftChild()
        elif t in ['+', '-', '*', '/']:
            current.setRootVal(t)
            current.insertRight('')
            stack.push(current)
```

```
            current = current.getRightChild()
        elif t == ')':
            current = stack.pop()
        else:
            current.setRootVal(int(t))
            current = stack.pop()
    return tree
```

2. 表达式求值

```
import operator

def evaluate(parseTree):
    opers = {'+': operator.add, '-': operator.sub,
             '*': operator.mul, '/': operator.truediv}
    left = parseTree.getLeftChild()
    right = parseTree.getRightChild()

    if left and right:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(left), evaluate(right))
    else:
        return parseTree.getRootVal()
```

三、哈夫曼编码

1. 哈夫曼建树

```
import heapq

def huffman_tree(symbols):
    heap = [[weight, [sym, ""]] for sym, weight in symbols.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]: pair[1] = '0' + pair[1]
        for pair in hi[1:]: pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heap[0][1:], key=lambda p: (len(p[-1]), p))
```

四、二叉搜索树（Binary Search Tree）

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

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key, val=None):
        """插入键值对"""
        def _insert(node, key, val):
            if not node:
                return TreeNode(key, val)
            if key < node.key:
                node.left = _insert(node.left, key, val)
            elif key > node.key:
                node.right = _insert(node.right, key, val)
            else:  # key 已存在，更新值
                node.val = val
            return node

        self.root = _insert(self.root, key, val)

    def search(self, key):
```

```python
        """查找 key 对应的值"""
        node = self.root
        while node:
            if key < node.key:
                node = node.left
            elif key > node.key:
                node = node.right
            else:
                return node.val
        return None

    def delete(self, key):
        """删除指定 key 的节点"""
        def _delete(node, key):
            if not node:
                return None
            if key < node.key:
                node.left = _delete(node.left, key)
            elif key > node.key:
                node.right = _delete(node.right, key)
            else:
                # 找到要删除的节点
                if not node.left:
                    return node.right
                if not node.right:
                    return node.left
                # 有两个子节点，找后继节点(右子树的最小节点)
                temp = node.right
                while temp.left:
                    temp = temp.left
                node.key, node.val = temp.key, temp.val
                node.right = _delete(node.right, temp.key)
            return node

        self.root = _delete(self.root, key)

    def inorder(self):
        """中序遍历生成有序序列"""
        def _inorder(node):
            if node:
                yield from _inorder(node.left)
                yield (node.key, node.val)
                yield from _inorder(node.right)
        return _inorder(self.root)
```

```
def __str__(self):
    return "[" + ", ".join(f"{k}:{v}" for k, v in self.inorder()) + "]"
```