Cheatsheet

overall:

数算B 争取双十协定，包括10个左右数据结构，10个左右算法。例如：
数据结构：Stack, Queue,Deque,Linkedlist, Tree, Binary Search Tree, Heap, AVL, Disjoint set/Union-Find, Segment Tree, Binary Indexed Tree, Trie, 单调栈
算法：Sorting, Shutting yard, Huffman, heap懒更新, DFS, BFS, Dijkstra, Prim, Krusal, 拓扑排序, KMP

栈

中序转后序shunting yard

数学表达式

```
n=int(input())
def flush_num_buffer():
    if num_buffer:
        postfix_expre.append(''.join(num_buffer))
        num_buffer.clear()
for _ in range(n):
    infix_expre=input()
    postfix_expre=[]
    numbers='0123456789.'
    operators='+-*/'
    priority={}
    priority["+"],priority["-"]=1,1
    priority["*"], priority["/"] = 2,2
    priority['('],priority[')']=0,0 #stack里的'('相当于不存在，无优先级，只起分割作用
    stack=[]
    num_buffer=[]

    for i in infix_expre:
        if i in numbers:
            num_buffer.append(i)
        elif i in operators:
            flush_num_buffer()
            #此处必须while不能if：[+,*] ←+，需弹出两次
            while stack and priority[i]<=priority[stack[-1]]:
                postfix_expre.append(stack.pop())
            stack.append(i)
        elif i == "(":
            stack.append(i)
        elif i == ')':
            flush_num_buffer()
            in_bracket=stack.pop()
            while in_bracket!='(':
                postfix_expre.append(in_bracket)
                in_bracket=stack.pop()
    flush_num_buffer()
    while stack:
        postfix_expre.append(stack.pop())
```

```
    print(' '.join(postfix_expre))
```

布尔表达式

```
def ShuntingYard(l:list):
    stack,output=[],[]
    for i in l:
        if i==" ":continue
        if i in 'VF':output.append(i)
        elif i=='(':stack.append(i)
        elif i in '&|!':
            while True:
                if i=='!':break
                elif not stack:break
                elif stack[-1]=="(":
                    break
                else:output.append(stack.pop())
            stack.append(i)
        elif i==')':
            while stack[-1]!='(':
                output.append(stack.pop())
            stack.pop()
    if stack:output.extend(reversed(stack))
    return output
```

排序

求逆序对数（merge sort)

```
def merge_sort(a):
    if len(a)<=1:
        return a,0
    mid=len(a)//2
    l,l_cnt=merge_sort(a[:mid])
    r,r_cnt=merge_sort(a[mid:])
    merged,merge_cnt=merge(l,r)
    return merged,l_cnt+r_cnt+merge_cnt
def merge(l,r):
    merged=[]
    l_idx,r_idx=0,0
    inverse_cnt=0
    while l_idx<len(l) and r_idx<len(r):
        if l[l_idx]<=r[r_idx]:
            merged.append(l[l_idx])
            l_idx+=1
        else:
            merged.append(r[r_idx])
            r_idx+=1
            inverse_cnt+=len(l)-l_idx
    merged.extend(l[l_idx:])
    merged.extend(r[r_idx:])
```

```
    return merged,inverse_cnt
```

树

二叉堆

```
class BinHeap:
    def __init__(self):
        self.heaplist=[0]
        self.heaplen=0
    #节点上移
    def percolate_up(self,pos):
        while pos>1:
            if self.heaplist[pos]<self.heaplist[pos>>1]:

self.heaplist[pos],self.heaplist[pos>>1]=self.heaplist[pos>>1],self.heaplist[pos]
            else:
                break
            pos>>=1
    def find_min_child(self,pos):
        if pos<<1==self.heaplen:
            return pos<<1
        else:
            if self.heaplist[pos<<1]<self.heaplist[pos<<1|1]:
                return pos<<1
            else:
                return pos<<1|1
    def percolate_down(self,pos):
        #确保pos至少有左儿子（否则到底了）
        while (pos<<1)<=self.heaplen:
            minchild=self.find_min_child(pos)
            if self.heaplist[pos]>self.heaplist[minchild]:

self.heaplist[pos],self.heaplist[minchild]=self.heaplist[minchild],self.heaplist[
pos]
            else:
                break
            pos=minchild
    def insert_element(self,element):
        self.heaplist.append(element)
        self.heaplen+=1
        self.percolate_up(self.heaplen) #调用自己的方法也需要self!！

    def del_min(self):
        min_element=self.heaplist[1]
        self.heaplist[1]=self.heaplist[-1]
        self.heaplist.pop()
        self.heaplen-=1
        self.percolate_down(1)
        return min_element
```

```
    #对一个列表建堆（从完全二叉树的最后一个非叶子节点开始，向上遍历每个节点。对于每个节点，进行下沉操作，将节点与其子节点进行比较，并交换位置直到满足小根堆的性质。重复步骤，直到根节点。
    def buildHeap(self, alist):
        i = len(alist) // 2
        self.heaplen = len(alist)
        self.heaplist = [0] + alist[:]
        while (i > 0):
            print(f'i = {i}, {self.heaplist}')
        self.percolate_down(i)
        i = i - 1
        print(f'i = {i}, {self.heaplist}')

n=int(input())
binary_heap=BinHeap()
for _ in range(n):
    operation=input()
    if operation[0]=='1':
        #print(operation.split())
        new_element=int(operation.split()[1])
        #print(new_element)
        binary_heap.insert_element(new_element)
    else:
        print(binary_heap.del_min())
```

递归建树+二叉与多叉转换后层次遍历

```
#建二叉树
def build_tree(nodes,i):
    val,prop=nodes[i][0],nodes[i][1]
    node=Node(val)
    #说明是内部节点，有子节点，继续递归建树
    if prop=='0':
        i+=1
        new_node,i=build_tree(nodes,i)
        node.left=new_node
        i+=1
        new_node,i=build_tree(nodes,i)
        node.right=new_node
    #若已到叶子节点，直接return
    return node,i
```

```
def level_order_traversal(root):
    queue=deque()
    traversal=[]
    stack=[]
    queue.append(root)
    while queue:
        for _ in range(len(queue)):
            current_node=queue.popleft()
            stack.append(current_node.value)
            #遍历这一层的同时，加入下一层的所有节点
            while current_node.right!=None and current_node.right.value!='$':
                if current_node.left != None and current_node.left.value != '$':
                    queue.append(current_node.left)
```

```
                stack.append(current_node.right.value)
                current_node=current_node.right
            #别忘了检查最后一个节点的左儿子
            if current_node.left!=None and current_node.left.value!='$':
                queue.append(current_node.left)
        #print(stack)
        while stack:
            traversal.append(stack.pop())
    return traversal
```

表达式树（unfini）

前中后序转换+后序输出

```
def parse_tree(preorder,inorder):
    if not inorder:
        return None,preorder
    root_val=preorder.pop(0)
    root=Node(root_val)
    root_index=inorder.index(root_val)
    left_tree,right_tree=inorder[:root_index],inorder[root_index+1:]
    root.left,preorder=parse_tree(preorder,left_tree)
    root.right,preorder=parse_tree(preorder,right_tree)
    return root,preorder
```

```
def post_order_traversal(root):
    if root==None:
        return []
    return post_order_traversal(root.left)+post_order_traversal(root.right)+
[root.value]
```

用stack来解析树（括号嵌套树）

```
A(B(E),C(F,G),D(H(I)))
def parse_tree(l):
    stack=[]
    for i in l:
        if i =='(':
            stack.append(node)
        elif i.isalpha():
            node=Node(i)
            if stack:
                stack[-1].child.append(node)
            else:
                root=node
        elif i ==')':
            stack.pop()
    return root
```

构建二叉搜索树

```
def insert(root,num):
    if not root:
        return Node(num)
    if num<root.val:
        root.left=insert(root.left,num)
    else:
        root.right=insert(root.right,num)
    return root
root=insert(root,num)
```

huffman

```
import heapq
class Node:
    def __init__(self,value,freq):
        self.value=value
        self.freq=freq
        self.left=None
        self.right=None
    def __lt__(self, other):
        if self.freq!=other.freq:
            return self.freq < other.freq
        else:
            return self.value < other.value
def build_huffman(l):
    #小根堆
    while len(l)>1:
        a=heapq.heappop(l)
        b=heapq.heappop(l)
        merge=Node(None,a.freq+b.freq) #非叶子结点的value=None便于区分
        merge.left,merge.right=a,b
        heapq.heappush(l,merge)
        #print(merge.freq)
    return merge
def traverse(node,codes,huffman_code):
    if node.value!=None:
        codes[node.value]=huffman_code
    else:
        traverse(node.left,codes,huffman_code+'0')
        traverse(node.right,codes,huffman_code+'1')

def decode(l,root):
    decoded_s=''
    node=root
    for i in l:
        if i=='0':
            node=node.left
        elif i=='1':
            node=node.right
        if node.value!=None:
```

```
            decoded_s+=node.value
            node=root
    return decoded_s

n=int(input())
char_freq=[]
for _ in range(n):
    char,freq=input().split()
    node=Node(char,int(freq))
    char_freq.append(node)
heapq.heapify(char_freq)
root=build_huffman(char_freq)
codes={}
traverse(root,codes,'')
while True:
    try:
        s=list(input())
        #print(s)
        if s[0].isalpha():
            huffman_code=''
            for i in s:
                huffman_code+=codes[i]
            print(huffman_code)
        else:
            print(decode(s,root))
    except EOFError:
        break
```

AVL树（还需复习！！！）

```
class Node:
    def __init__(self,value):
        self.value=value
        self.left=None
        self.right=None
        self.height=1
class AVL_tree:
    def __init__(self):
        self.root=None
    #初始化插入（用于区分是否为根节点）
    def insert(self,num):
        if self.root ==None:
            self.root=Node(num)
        else:#此处注意root要传过来（每次插入新节点，root都有可能变化）
            self.root=self.real_insert(self.root,num)
    #真正的插入
    def real_insert(self,node,num):
        if node==None:
            return(Node(num))
        if num<node.value:
            node.left=self.real_insert(node.left,num)
        else:
            node.right=self.real_insert(node.right,num)

        #检查是否平衡（自下而上）这样确保height逐级更新上去
```

```
        node.height=1+max(self.get_height(node.left),self.get_height(node.right))
        balance=self.get_height(node.left)-self.get_height(node.right)

        if balance>1:
            if num<node.left.value: #LL,右旋一次
                node=self.rotate_right(node)
            else: #LR,先左后右
                node.left=self.rotate_left(node.left)
                node=self.rotate_right(node)
        elif balance<-1:
            if num>node.right.value: #RR,左旋一次
                node=self.rotate_left(node)
            else: #RL,先右后左
                node.right=self.rotate_right(node.right)
                node=self.rotate_left(node)
        #返回（子树）根节点
        return node

    def rotate_right(self,node):
        new_root=node.left
        tmp=new_root.right
        node.left=tmp
        new_root.right=node
        #更改节点高度（个人认为new_root不需要改？）
        node.height=1+max(self.get_height(node.left),self.get_height(node.right))

new_root.height=1+max(self.get_height(new_root.left),self.get_height(new_root.rig
ht))
        return new_root
    def rotate_left(self,node):
        new_root = node.right
        tmp = new_root.left
        node.right = tmp
        new_root.left = node
        # 更改节点高度（个人认为new_root不需要改？）
        node.height = 1 + max(self.get_height(node.left),
self.get_height(node.right))
        new_root.height = 1 + max(self.get_height(new_root.left),
self.get_height(new_root.right))
        return new_root
    #获取高度函数的作用：遇到空节点时返回0（否则会报错）
    def get_height(self,node):
        if node==None:
            return 0
        return node.height

    def preorder(self):
        return self.real_preorder(self.root)
    def real_preorder(self,node):
        if node==None:
            return []
        return
[node.value]+self.real_preorder(node.left)+self.real_preorder(node.right)

n=int(input())
numbers=list(map(int,input().split()))
```

```
avl_tree=AVL_tree()
for num in numbers:
    avl_tree.insert(num)
output=avl_tree.preorder()
#print(output)
output=map(str,output)
print(' '.join(output))
```

前缀树

并查集

```
class UnionFind:
    def __init__(self,n):
        self.parent=list(range(n))
        self.rank=[0]*n

    def find(self,x):
        if self.parent[x]!=x:
            self.parent[x]=self.find(self.parent[x])
        return self.parent[x]

    def union(self,x,y):
        px,py=self.find(x),self.find(y)
        if self.rank[x]>self.rank[y]:
            self.parent[py]=px
        elif self.rank[x]<self.rank[y]:
            self.parent[px] = py
        else:
            self.parent[px]=py
            self.rank[py]+=1
```

图

建图常用方法

```
vertex={i:[] for i in range(n)}
edges=[]
edges.append((u,v))
```

prim算法（最小生成树）兔子与星空

```
import heapq
def prim(start,graph):
    mst=[]
    path=[]
    visited=set()
    visited.add(start)
    for to,cost in graph[start].items():
        path.append((cost,start,to))
```

```
    heapq.heapify(path)
    while path:
        cost,frm,cur=heapq.heappop(path)
        if cur not in visited:
            visited.add(cur)
            mst.append((frm,cur,cost))
            for to,cost in graph[cur].items():
                heapq.heappush(path,(cost, cur, to))
    return mst


n=int(input())
graph={chr(i+65):{} for i in range(n)}
for i in range(n-1):
    info=input().split()
    for j in range(int(info[1])):
        #无向：正反都要连线!
        graph[info[0]][info[2*j+2]] = int(info[2*j+3])
        graph[info[2*j+2]][info[0]] = int(info[2*j+3])
mst=prim("A",graph)
tot_weight=0
for _,_,cost in mst:
    tot_weight+=cost
print(tot_weight)
```

kruskal算法+并查集（最小生成树）

```
def kruskal(n,edges):
    union_set=UnionFind(n)
    edges.sort(key=lambda x:x[2])
    res,line=0,0
    for u,v,cost in edges:
        if union_set.find(u)!=union_set.find(v):
            union_set.union(u,v)
            res+=cost
            line+=1
        if line==n-1:
            return res
while True:
    try:
        n=int(input())
        matrix=[]
        for i in range(n):
            row=list(map(int,input().split()))
            matrix.append(row)
        edges=[]
        for i in range(n):
            for j in range(i+1,n):
                edges.append((i,j,matrix[i][j]))
        print(kruskal(n,edges))
    except EOFError:
        break
```

dijkstra

bfs+三维visited（条件走迷宫：鸣人和佐助）

理解：dij和deque/heap+visited达成的效果是一样的

```
def bfs(map,start_x,start_y,end_x,end_y,t):
    queue=deque()
    visited=set()
    # 加一个元素：当前查克拉数量t，如果走的时候发现查克拉和之前一样，则不用再走（因为之后的情形是一样的）
    visited.add((start_x,start_y,t))
    queue.append((start_x,start_y,t,0,visited))
    path=[[-1,0],[0,1],[1,0],[0,-1]]
    while queue:
        #flag=True
        for _ in range(len(queue)):
            x,y,remain_t,time,visited=queue.popleft()
            for dx,dy in path:
                next_x,next_y=x+dx,y+dy
                if next_x==end_x and next_y==end_y:
                    return time + 1
                if 0<=next_x<=m-1 and 0<=next_y<=n-1 and (next_x,next_y,remain_t) not in visited:
                    if map[next_x][next_y] == '#' and remain_t>0:
                        visited.add((next_x,next_y,remain_t-1))
                        queue.append((next_x,next_y,remain_t-1,time+1,visited))

                    elif map[next_x][next_y] == '*':
                        visited.add((next_x, next_y,remain_t))
                        queue.append((next_x,next_y,remain_t,time+1,visited))

    return -1
```

bfs+heapq(带权路径：走山路)

```
def bfs(x,y):
    if board[x][y]==-1e5 or board[x2][y2]==-1e5:
        print("NO")
        return True
    res=[(0,x,y)]
    heapq.heapify(res)
    vis=[[0]*m for _ in range(n)]
    while res:
        for _ in range(len(res)):
            strength,x,y=heapq.heappop(res)
            vis[x][y]=1 #出队标记！每次走过一个点时才标记visited，否则可能遗漏
            if x==x2 and y==y2:
                print(strength)
                return True
            else:
                for k in range(4):
                    nx,ny=x+dx[k],y+dy[k]
```

```
                    if 0<=nx<=n-1 and 0<=ny<=m-1 and board[nx][ny]!=-1e5 and
vis[nx][ny]==0:
                        heapq.heappush(res,(strength+abs(board[nx][ny]-board[x]
[y]),nx,ny))
    return False
```

拓扑排序（两种）

```
import heapq
def topological_sort(vertice,edges):
    in_degree=[0]*(vertice+1)
    connect=[[0]*(vertice+1) for _ in range(vertice+1)]
    if_isolated=[1]*(vertice+1)
    for u,v in edges:
        if_isolated[u],if_isolated[v]=0,0
        in_degree[v]+=1
        connect[u][v]+=1
    to_sort=[]
    heapq.heapify(to_sort)
    for i in range(1,vertice+1):
        if in_degree[i]==0:
            heapq.heappush(to_sort,i)

    order=[]
    while to_sort:
        u=heapq.heappop(to_sort)
        order.append(u)
        for v in range(1,vertice+1):
            if connect[u][v]>0:
                in_degree[v]-=connect[u][v]
                if in_degree[v]==0:
                    heapq.heappush(to_sort,v)
    if len(order)==vertice:return order
    #成环
    else: return None

vertice,num_edges=map(int,input().split())
edges=[]
for _ in range(num_edges):
    u,v=map(int,input().split())
    edges.append((u,v))
order=topological_sort(vertice,edges)
for v in order:
    print(f"v{v}",end=' ')
```

```
from collections import deque, defaultdict

def topological_sort(graph):
    indegree = defaultdict(int)
    result = []
    queue = deque()

    # 计算每个顶点的入度
```

```
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1

    # 将入度为 0 的顶点加入队列
    for u in graph:
        if indegree[u] == 0:
            queue.append(u)

    # 执行拓扑排序
    while queue:
        u = queue.popleft()
        result.append(u)

        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)

    # 检查是否存在环
    if len(result) == len(graph):
        return result
    else:
        return None

# 示例调用代码
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['F'],
    'E': ['F'],
    'F': []
}
```

判断无向图是否有环

```
#key:记录父节点！！
def loop_dfs(graph,visited,vertex,p_v):
    for edge in graph[vertex]:
        if visited[edge]==0:
            visited[edge]=1
            if(loop_dfs(graph,visited,edge,vertex)):
                return True
        elif visited[edge]==1 and p_v!=edge:
            return True
    return False
```

判断有向图是否有环（dfs+着色）or拓扑排序

```
def dfs_check_loop(graph,visited,i):
    for neighbor in graph[i]:
        if visited[neighbor]==0:
            visited[neighbor]=1
            if dfs_check_loop(graph,visited,neighbor):
                return True
        elif visited[neighbor]==1:
            return True
    #i的所有后序节点全部走完后：
    visited[i]=2
    return False
```

二分法

```
while l<=r:
    #print(l,r)
    mid=(l+r)//2
    if check(mid):
        r=mid-1
    else:
        l=mid+1
```

bisect

```
import bisect

a = [1, 3, 5, 7, 9]

# 查找元素 4 的插入位置
index = bisect.bisect_left(a, 4)
print(index)  # 输出: 2
```

工具

字符串

1. `str.lstrip() / str.rstrip()`：移除字符串左侧/右侧的空白字符。
2. `str.find(sub)`：返回子字符串`sub`在字符串中首次出现的索引，如果未找到，则返回-1。
3. `str.replace(old, new)`：将字符串中的`old`子字符串替换为`new`。
4. `str.startswith(prefix) / str.endswith(suffix)`：检查字符串是否以`prefix`开头或以`suffix`结尾。
5. `str.isalpha() / str.isdigit() / str.isalnum()`：检查字符串是否全部由字母/数字/字母和数字组成。
6. `str.title()`：每个单词首字母大写。

全排列

```
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出：(1, 2, 3)，(1, 3, 2)，(2, 1, 3)，(2, 3, 1)，(3, 1, 2)，(3, 2, 1)
```

逻辑结构指的是数据元素之间的逻辑关系，而不涉及具体的存储结构。常见的逻辑结构包括：

1. 线性结构：数据元素之间存在一对一的关系，如线性表、栈、队列等。
2. 非线性结构：数据元素之间存在一对多或多对多的关系，如树结构、图结构等。
3. 集合结构：数据元素之间没有特定的顺序关系，如集合。

顺序表是一种最基本的线性数据结构,其特点如下:

1. 采用连续的存储空间来存储数据元素。
2. 数据元素在物理上是连续的,逻辑上也是连续的。
3. 每个数据元素都有唯一的序号(下标),通过下标可以快速访问任意元素。
4. 插入和删除操作需要移动大量元素,效率较低。
5. 适合于查找和存取操作,不适合于频繁的插入和删除操作。

在数据结构中，从逻辑上可以把数据结构分成两类：线性结构,非线性结构

二叉树的高（Height）是指从根节点到最远叶子节点的最长路径的边数

二叉树的深度（Depth）是指从根节点到最远叶子节点的最长路径的节点数

满二叉树中，所有的叶子节点都在最底层，非叶子节点的度都是2

完全二叉树：从上到下从左到右逐渐填满，没有度为1 的节点

森林到二叉树的转换
森林是一组不相交的树。将森林转换为二叉树的方法通常称为“孩子-兄弟表示法”：
每个节点的第一个孩子作为该节点的左孩子。
每个节点的兄弟作为该节点的右孩子。
二叉树中右指针域为空的节点
我们需要确定在这种转换中，二叉树中右指针域为空的节点的数量。
分析过程
非终端结点的定义：非终端节点是指在原始森林中有孩子的节点。
转换后的二叉树：在二叉树中，只有那些没有右孩子的节点，其右指针域为空。
如果一个节点没有兄弟节点，那么在二叉树中这个节点的右指针域为空。

在孩子-兄弟表示法中，以下几类节点会导致右指针为空：
每棵树的根节点（因为它们没有兄弟）。
每个非终端节点的最右侧的孩子（因为它们没有右兄弟）。
右指针为空的节点数量
每棵树的根节点的右指针域为空。
每个非终端节点的最后一个孩子的右指针域为空。
考虑所有叶子节点，这些节点也没有右孩子。
由于根节点的右指针域为空，而森林的根节点数就是树的数量，每个树至少有一个根节点。如果我们设 n 是非终端节点的数量，那么根节点（第一层）的数量加上叶子节点的数量会影响右指针为空的节点总数。
关键点
对于森林转换成的二叉树：
如果 F 中有 n 个非终端节点，每个非终端节点的每个最后一个孩子节点的右指针域为空。
每棵树的根节点的右指针域为空，这些根节点数通常为 n+1（根节点数 = 非终端节点数 + 1）。
因此，二叉树中右指针域为空的节点数为 n+1
结论
如果 F 中有 n个非终端结点，则由 F变换得的二叉树 B中右指针域为空的节点数为：
n+1