CS201 Cheating sheet

by jerry 2024.6.5

> 主要是一些本学期的新内容

一、数据结构及基础操作

1. 队列、链表、栈

双端队列 `deque`

```python
from collections import deque

# 初始化deque
d = deque([1, 2, 3])

```

```python
# 添加元素
d.append(4)  # deque变为[1, 2, 3, 4]
d.appendleft(0)  # deque变为[0, 1, 2, 3, 4]

# 移除元素
d.pop()  # 返回 4, deque变为[0, 1, 2, 3]
d.popleft()  # 返回 0, deque变为[1, 2, 3]

# 扩展
d.extend([4, 5])  # deque变为[1, 2, 3, 4, 5]
d.extendleft([0])  # deque变为[0, 1, 2, 3, 4, 5]

# 旋转
d.rotate(1)  # deque变为[5, 0, 1, 2, 3, 4]
d.rotate(-2)  # deque变为[1, 2, 3, 4, 5, 0]

# 清空
d.clear()  # deque变为空
```

queue

- `put(item, block=True, timeout=None)`：将 `item` 放入队列中。如果可选参数 `block` 设为 `True`，并且 `timeout` 是一个正数，则在超时前会阻塞等待可用的槽位。
- `get(block=True, timeout=None)`：从队列中移除并返回一个元素。如果可选参数 `block` 设为 `True`，并且 `timeout` 是一个正数，则在超时前会阻塞等待元素。
- `empty()`：判断队列是否为空。
- `full()`：判断队列是否已满。
- `qsize()`：返回队列中的元素数量。注意，这个大小只是近似值，因为在返回值和队列实际状态间可能存在时间差。

```python
import queue

# 创建一个先进先出队列
pq1 = queue.Queue()
# 创建一个优先级队列
pq = queue.PriorityQueue()

# 添加元素及其优先级
pq.put((3, 'Low priority'))
pq.put((1, 'High priority'))#先输出这个
pq.put((2, 'Medium priority'))

# 依次取出元素
while not pq.empty():
    print(pq.get()[1])  # 输出元素的数据部分
```

最小堆 `heapq`

```python
import heapq
data = [3, 1, 4, 1, 5, 9, 2, 6, 5]
heapq.heapify(data)
heapq.heappush(data, 3)
print(heapq.heappop(data))
heapreplace(heap, item)  # 弹出最小元素并插入新元素
print(data)  # 输出将是堆，但可能不是完全排序的
```

`defaultdict`

`defaultdict` 是另一种字典子类，它提供了一个默认值，用于字典所尝试访问的键不存在时返回。

```python
from collections import defaultdict

# 使用 lambda 来指定默认值为 0
d = defaultdict(lambda: 0)

d['key1'] = 5
print(d['key1'])  # 输出: 5
print(d['key2'])  # 输出: 0，因为 key2 不存在，返回默认值 0
```

2. 树

左孩子右兄弟

```python
class fNode:
    def __init__(self, key):
        self.key = key
        self.first = None
        self.next = None
        self.parent = None
        self.height = 0

    # 给self节点加个子节点
    def put(self, p_key):
        if self.first is None:
            self.first = fNode(p_key)
            self.first.parent = self
            self.first.checkheight()
            return self.first
        pos_node = self.first
        while pos_node.next is not None:
            pos_node = pos_node.next
        pos_node.next = fNode(p_key)
        pos_node.next.parent = pos_node.next
        return pos_node.next

    # 自动修正层高
    def checkheight(self):
        pos_node = self
        while pos_node.parent is not None and \
            pos_node.height == pos_node.parent.height:
```

```
            pos_node.parent.height += 1
            pos_node = pos_node.parent
```

不排序二叉树

```python
# 二叉树
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0

    def put_left(self, p_key):
        self.left = Node(p_key)
        self.left.parent = self
        self.left.checkheight()
        return self.left

    def put_right(self, p_key):
        self.right = Node(p_key)
        self.right.parent = self
        self.right.checkheight()
        return self.right

    def checkheight(self):
        pos_node = self
        while pos_node.parent is not None and \
            pos_node.height == pos_node.parent.height:
            pos_node.parent.height += 1
            pos_node = pos_node.parent
```

二叉搜索树

```python
import queue

# 定义节点
class Node:
    # key用于判断，没有即退化为用value判断
    # 需要的话后面算法自己加parent!
    def __init__(self, key, value=None, parent=None):
        self.key = key
        self.value = value if value is not None else key
        self.left = None
        self.right = None
        self.parent = parent

    # 节点大小比较定义（按需取用）
    def __lt__(self, other):
        return self.key < other.key
    def __gt__(self, other):
        return self.key > other.key
    def __eq__(self, other):
```

```python
        return self.key == other.key
    def __le__(self, other):
        return self.key <= other.key
    def __ge__(self, other):
        return self.key >= other.key
    def __ne__(self, other):
        return self.key != other.key

# 二叉搜索树
class BinarySearchTree:
    # 定义根节点
    def __init__(self):
        self.root = None

    # 插入节点总算法
    def put(self, key, val=None):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = Node(key,val)

    def _put(self, key, val=None, pos_node=None):
        put_node = Node(key, val)
        while True:
            if put_node > pos_node:
                if pos_node.right:
                    pos_node = pos_node.right
                else:
                    pos_node.right = put_node
                    break
            elif put_node < pos_node:
                if pos_node.left:
                    pos_node = pos_node.left
                else:
                    pos_node.left = put_node
                    break
            # 大小相等的节点不重复插入树
            else:break

    # 前序遍历（列表输出，中后序自己换位置，记得改代码里的方法名pre）
    def preorder(self,node=None):
        if node is None:
            if self.root:
                node = self.root
            else: return None
        output = [node.value]
        if node.left:
            output += self.preorder(node.left)
        if node.right:
            output += self.preorder(node.right)
        return output

    # 层次遍历，同层从左到右，需要queue库
    def levelorder(self):
        pos = queue.Queue()
        pos.put(self.root)
```

```
        output = []
        while not pos.empty():
            pos_node = pos.get()
            output.append(pos_node.value)
            if pos_node.left:
                pos.put(pos_node.left)
            if pos_node.right:
                pos.put(pos_node.right)
        return output
```

3. 图

邻接表实现图

```
# 无权图
graph1 = {'A': ['B', 'C', 'E'],
        'B': ['A','D', 'E'],
        'C': ['A', 'F', 'G'],
        'D': ['B'],
        'E': ['A', 'B','D'],
        'F': ['C'],
        'G': ['C']}
# 有权图
graph2 = {'A': {'B': 1, 'C': 2, 'E': 3}..................}
```

字典树

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_number = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, number):
        node = self.root
        for digit in number:
            if digit not in node.children:
                node.children[digit] = TrieNode()
            node = node.children[digit]
            if node.is_end_of_number:
                return False
        node.is_end_of_number = True
        return len(node.children) == 0

def check_consistency(t, test_cases):
    results = []
    for i in range(t):
        n = test_cases[i][0]
        numbers = test_cases[i][1:]
```

```python

        trie = Trie()
        consistent = True
        for number in numbers:
            if not trie.insert(number):
                consistent = False
                break

        if consistent:
            results.append("YES")
        else:
            results.append("NO")

    return results

# 读取输入
t = int(input())
test_cases = []
for _ in range(t):
    n = int(input())
    numbers = [input() for _ in range(n)]
    test_cases.append([n] + numbers)

# 检查一致性并输出结果
results = check_consistency(t, test_cases)
for result in results:
    print(result)
```

图的类实现

```python
# 定义节点
class Vertex:
    def __init__(self,key):
        self.key = key
        self.color = 'white'
        self.neighbors = {}
        self.previous = None
        # DFS 专用
        self.discovery_time = -1
        self.closing_time = -1
        # BFS 专用
        self.distance = 0

    # 线的权重
    def addneighbor(self,nbr,weight=0):
        self.neighbors[nbr] = weight

    def getneighbors(self):
        return self.neighbors.keys()

    def getweight(self,nbr):
        return self.neighbors[nbr]

```

```python
# 定义图
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

    # 就是 in
    def __contains__(self,n):
        return n in self.vertList

    # 为无向图必须再加一项
    def addEdge(self,f,t,weight=0):
        self.vertList[f].addneighbor(self.vertList[t], weight)

        # 回溯路径
    def traverse(self,starting_vertex):
        ans = []
        current = starting_vertex
        while (current.previous):
            ans.append(current.key)
            current = current.previous
        ans.append(current.key)
        # 正序输出
        ans.reverse()
        return ans
```

二、算法

1. 排序

归并排序（稳定）

```python
# 逆序数
d = 0

def merge(arr, l, m, r):
    """对l到m和m到r两段进行合并"""
    global d
    n1, n2 = m - l + 1, r - m  # L1和L2的长
    L1, L2 = arr[l:m + 1], arr[m + 1:r + 1]
    # L1和L2均为有序序列
    i, j, k = 0, 0, l  # i为L1指针，j为L2指针，k为arr指针
    '''双指针法合并序列'''
```

```python
    while i < n1 and j < n2:
        if L1[i] <= L2[j]:
            arr[k] = L1[i]
            i += 1
        else:
            arr[k] = L2[j]
            d += n1 - i  # 精髓所在
            j += 1
        k += 1
    while i < n1:
        arr[k] = L1[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = L2[j]
        j += 1
        k += 1

def mergesort(arr, l, r):
    """对arr的l到r一段进行排序"""
    if l < r:  # 递归结束条件，很重要
        m = (l + r) // 2
        mergesort(arr, l, m)
        mergesort(arr, m + 1, r)
        merge(arr, l, m, r)

while True:
    n = int(input())
    if n == 0:
        break
    array = []
    for b in range(n):
        array.append(int(input()))
    d = 0
    mergesort(array, 0, n - 1)
    print(d)
```

2. BFS与DFS

简单BFS

```python
def bfs(graph, initial):
    visited = []
    queue = [initial]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            neighbours = graph[node]

            for neighbour in neighbours:
                queue.append(neighbour)
```

```
    return visited

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

通用BFS

```python
# 接上文图实现
class BFSGraph(Graph):
    def __init__(self):
        super().__init__()

    def bfs(self):
        for vertex in self.vertList.values(): #从每个顶点开始遍历
            if vertex.color == "white":
                self.bfs_visit(vertex) #第一次运行后还有未包括的顶点

    # 建立森林
    def bfs_visit(self, start_vertex):
        start_vertex.color = "gray"
        for next_vertex in start_vertex.getneighbors():
            if next_vertex.color == "white":
                next_vertex.previous = start_vertex
                next_vertex.distance = start_vertex.distance + 1
        start_vertex.color = "black"
        for next_vertex in start_vertex.getneighbors():
            self.bfs_visit(next_vertex) #广度后置递归访问
```

通用DFS

```python
# 接上文图实现
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0 #不是物理世界，而是算法执行步数

    def dfs(self):
        for vertex in self.vertList.values(): #从每个顶点开始遍历
            if vertex.color == "white":
                self.dfs_visit(vertex) #第一次运行后还有未包括的顶点

    # 建立森林
    def dfs_visit(self, start_vertex):
        start_vertex.color = "gray"
        self.time = self.time + 1 #记录算法的步骤
        start_vertex.discovery_time = self.time
        for next_vertex in start_vertex.getneighbors():
            if next_vertex.color == "white":
                next_vertex.previous = start_vertex
                self.dfs_visit(next_vertex) #深度优先递归访问
        start_vertex.color = "black"
        self.time = self.time + 1
        start_vertex.closing_time = self.time
```

3. 并查集

```python
class UnionFind:
    def __init__(self):
        # 上级节点，尽可能为根节点
        self.parent = {}
        # 节点高度
        self.rank = {}

    def find(self, x):
        if x not in self.parent:
            self.parent[x] = x
            self.rank[x] = 0
            return x
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return

        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
        elif self.rank[x_root] > self.rank[y_root]:
            self.parent[y_root] = x_root
        else:
            self.parent[y_root] = x_root
            self.rank[x_root] += 1

# 使用示例
uf = UnionFind()
uf.union('apple', 'banana')
uf.union('cherry', 'durian')
uf.union('banana', 'cherry')

print(uf.find('apple'))  # 输出: 'banana'
print(uf.find('durian'))  # 输出: 'cherry'
print(uf.find('banana'))  # 输出: 'banana'
```

4. 有向图拓扑排序（修课）

```python
from collections import defaultdict

def topological_sort(graph):
    """
    给定一个有向图 graph,返回该图的一个拓扑排序结果。
    graph 是一个字典,key 是节点,value 是该节点的邻居节点列表。
    如果图中存在环,则返回 None。
    """
    in_degree = defaultdict(int)
```

```
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    queue = [node for node in graph if in_degree[node] == 0]
    topological_order = []

    while queue:
        node = queue.pop(0)
        topological_order.append(node)

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

    if len(topological_order) != len(graph):
        return None  # 图中存在环
    return topological_order

# 示例用法
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D', 'E'],
    'D': ['E'],
    'E': []
}

result = topological_sort(graph)
if result:
    print("拓扑排序结果:", " -> ".join(result))
else:
    print("图中存在环,无法进行拓扑排序")
```

5. kosaraju/2 DFS（强连通分量）

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
    # Step 1: Perform first DFS to get finishing times
    stack = []
    visited = [False] * len(graph)
```

```
    for node in range(len(graph)):
        if not visited[node]:
            dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs

# Example
graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
sccs = kosaraju(graph)
print("Strongly Connected Components:")
for scc in sccs:
print(scc)
```

6. Dijkstra算法（有权最短路径）

在BFS中采用优先队列存储数据，反复调用路程最短的节点计算。

注意该算法对各节点应初始化distance为一极大值（上面代码初始化为0）

7. Prim（有权图最小生成树）

```
import heapq

def prim(graph, n):
    visited = [False] * n
    min_heap = [(0, 0)] # (weight, vertex)
    min_spanning_tree_cost = 0

    while min_heap:
        weight, vertex = heapq.heappop(min_heap)

        if visited[vertex]:
            continue

        visited[vertex] = True
        min_spanning_tree_cost += weight

        for neighbor, neighbor_weight in graph[vertex]:
            if not visited[neighbor]:
                heapq.heappush(min_heap, (neighbor_weight, neighbor))
```

```python

    return min_spanning_tree_cost if all(visited) else -1

def main():
    n, m = map(int, input().split())
    graph = [[] for _ in range(n)]

    for _ in range(m):
        u, v, w = map(int, input().split())
        graph[u].append((v, w))
        graph[v].append((u, w))

    min_spanning_tree_cost = prim(graph, n)
    print(min_spanning_tree_cost)

if __name__ == "__main__":
    main()
```

8. Kruskal and Disjoint Set 求最小生成树

Kruskal算法的核心思想是通过不断选择权重最小的边，并判断是否会形成环路来构建最小生成树。

```python
class DisjointSet: # 下面代码要求从0开始!
    ......
def kruskal(graph):
    num_vertices = len(graph)
    edges = []

    # 构建边集
    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if graph[i][j] != 0:
                edges.append((i, j, graph[i][j]))

    # 按照权重排序
    edges.sort(key=lambda x: x[2])

    # 初始化并查集
    disjoint_set = DisjointSet(num_vertices)

    # 构建最小生成树的边集
    minimum_spanning_tree = []
    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):
            disjoint_set.union(u, v)
            minimum_spanning_tree.append((u, v, weight))
    return minimum_spanning_tree
```

三、杂点补充

1. try-except

Try except 无法使用break来跳出循环，如果已知结束的特定输入（比如说输入0代表结束），

那么可以用以下代码：

```
while True:
    try:
        Xxxxx
        If input()=='0':
        break
    except EOFFError:
        break
```

2. sys库

```
import sys
# 设置递归深度为 2000
sys.setrecursionlimit(2000)
a = sys.maxsize
sys.exit()
```

3. lambda表达式

Lambda表达式是Python中的一种匿名函数，通常用于简化代码和在需要函数作为参数的地方使用。以下是Lambda表达式的一些常用用法：

1. **作为函数参数**：Lambda表达式可以作为函数的参数传递，特别适用于需要简单函数的情况。

```
# 使用Lambda表达式作为排序函数的key参数
my_list = [1, 5, 3, 9, 2]
sorted_list = sorted(my_list, key=lambda x: x*x)
print(sorted_list)
```

2. **与内置函数结合使用**：Lambda表达式经常与内置函数结合使用，如`map()`、`filter()`等。

```
# 使用Lambda表达式和map()函数对列表中的每个元素进行平方操作
my_list = [1, 2, 3, 4, 5]
squared_list = list(map(lambda x: x*x, my_list))
print(squared_list)
```

3. **条件表达式**：Lambda表达式可以用于简单的条件判断。

```
# 使用Lambda表达式实现简单的条件判断
is_even = lambda x: True if x % 2 == 0 else False
print(is_even(4))  # 输出True
```

4. **字典排序**：Lambda表达式可以用于对字典进行排序。

```python
# 使用Lambda表达式对字典按值进行排序
my_dict = {'a': 5, 'b': 2, 'c': 8}
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1]))
print(sorted_dict)
```

5. **多参数函数**：Lambda表达式可以接受多个参数。

```python
# Lambda表达式接受多个参数并返回它们的和
addition = lambda x, y: x + y
print(addition(3, 4))  # 输出7
```