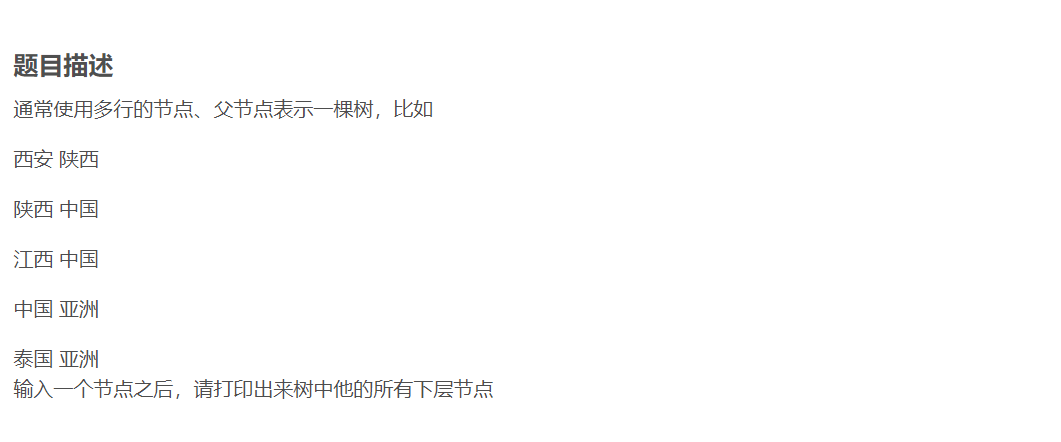
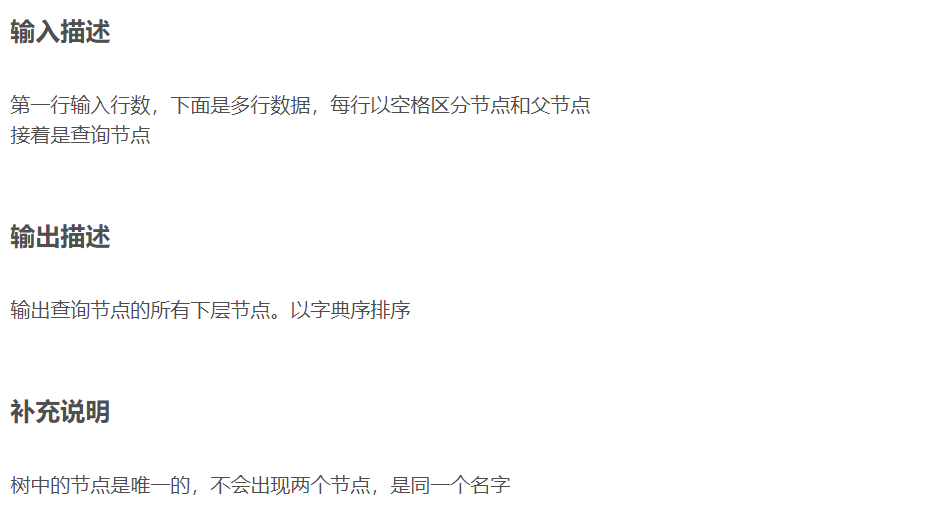
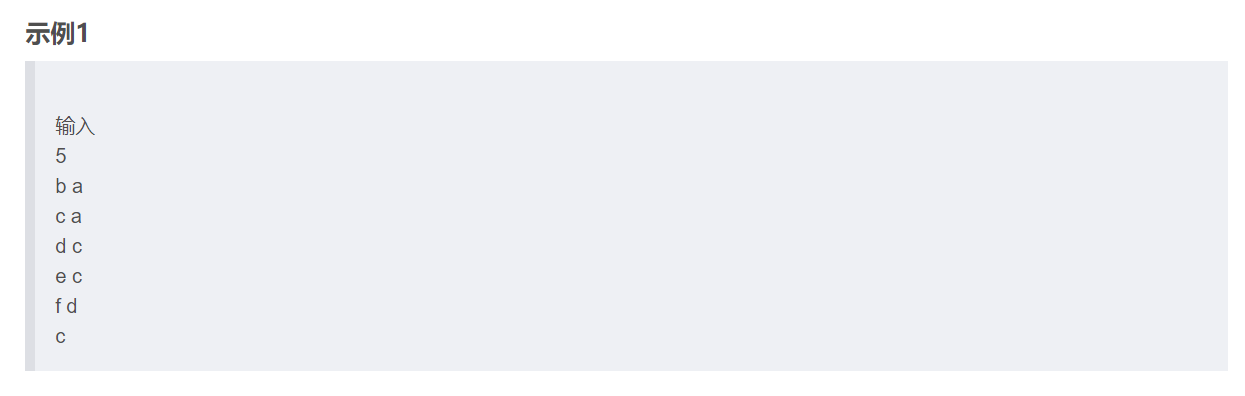
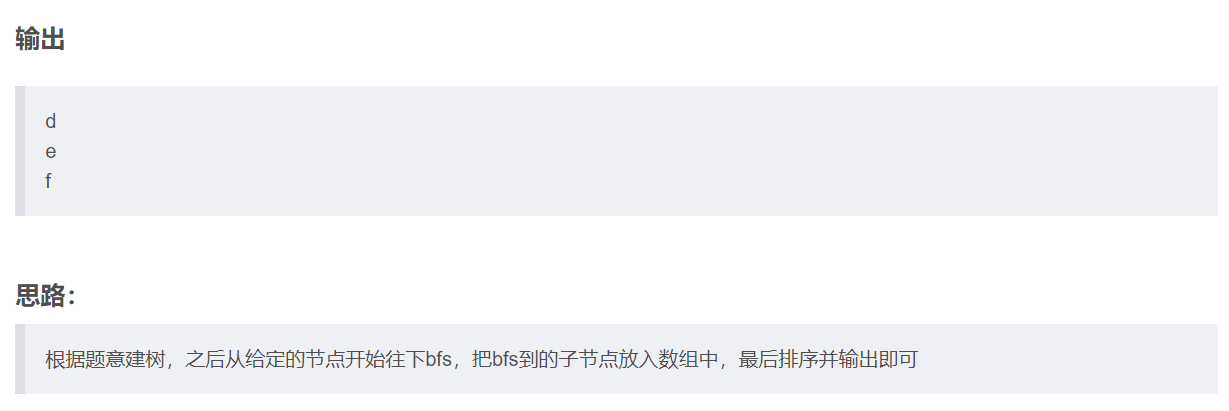
# **E卷-树状结构查询[200分]（ Java | Python3 | C++ | JsNode | Go | C语言）**











import java.util.\*;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

scanner.nextLine();

HashMap<String, List<String>> mp = new HashMap<>();

for (int i = 0; i < n; i++) {

String[] s = scanner.nextLine().split(" ");

if (!mp.containsKey(s[1])) {

mp.put(s[1], new ArrayList<>());

}

mp.get(s[1]).add(s[0]);

}

String go = scanner.nextLine();

List<String> ans = new ArrayList<>();

Deque<String> que = new LinkedList<>();

que.add(go);

while (!que.isEmpty()) {

String now = que.pollFirst();

if (mp.containsKey(now)) {

for (String x : mp.get(now)) {

ans.add(x);

que.add(x);

}

}

}

ans.sort(String::compareTo);

for (String s : ans) {

System.out.println(s);

}

scanner.close();

}

}



from collections import defaultdict, deque

n = int(input())

mp = defaultdict(list)

for i in range(n):

s = input().split()

mp[s[1]].append(s[0])

go = input()

ans = []

que = deque()

que.append(go)

while len(que):

now = que.popleft()

for x in mp[now]:

ans.append(x)

que.append(x)

ans.sort()

print("\n".join(ans))



#include <iostream>

#include <vector>

#include <unordered\_map>

#include <queue>

#include <algorithm>

using namespace std;

int main() {

int n;

cin >> n;

unordered\_map<string, vector<string>> mp;

for (int i = 0; i < n; ++i) {

string s1, s2;

cin >> s1 >> s2;

mp[s2].push\_back(s1);

}

string go;

cin >> go;

vector<string> ans;

queue<string> que;

que.push(go);

while (!que.empty()) {

string now = que.front();

que.pop();

for (const string& x : mp[now]) {

ans.push\_back(x);

que.push(x);

}

}

sort(ans.begin(), ans.end());

for (const string& x : ans) {

cout << x << endl;

}

return 0;

}



const readline = require('readline');

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout

});

let input = [];

rl.on('line', (line) => {

input.push(line);

}).on('close', () => {

let n = parseInt(input[0]);

let mp = {};

for (let i = 1; i <= n; i++) {

let [a, b] = input[i].split(' ');

if (!mp[b]) {

mp[b] = [];

}

mp[b].push(a);

}

let go = input[n + 1];

let ans = [];

let que = [go];

while (que.length > 0) {

let now = que.shift();

if (mp[now]) {

for (let x of mp[now]) {

ans.push(x);

que.push(x);

}

}

}

ans.sort();

console.log(ans.join('\n'));

});



package main

import (

"fmt"

"sort"

)

func main() {

var n int

fmt.Scan(&n)

mp := make(map[string][]string)

for i := 0; i < n; i++ {

var a,b string

fmt.Scan(&a)

fmt.Scan(&b)

if \_, ok := mp[b];!ok {

mp[b] = []string{}

}

mp[b] = append(mp[b], a)

}

var goStr string

fmt.Scan(&goStr)

ans := []string{}

que := []string{goStr}

for len(que) > 0 {

now := que[0]

que = que[1:]

if values, ok := mp[now]; ok {

for \_, x := range values {

ans = append(ans, x)

que = append(que, x)

}

}

}

sort.Strings(ans)

for \_, s := range ans {

fmt.Println(s)

}

}



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_NAME\_LEN 101

#define MAX\_LIST\_SIZE 1000

// 定义链表节点

typedef struct Node {

char name[MAX\_NAME\_LEN];

struct Node\* next;

} Node;

typedef struct {

char key[MAX\_NAME\_LEN];

Node\* head;

} DefaultDictEntry;

// 定义一个"默认字典"

typedef struct {

DefaultDictEntry\* entries;

int size;

} DefaultDict;

// 队列节点

typedef struct QueueNode {

char name[MAX\_NAME\_LEN];

struct QueueNode\* next;

} QueueNode;

typedef struct {

QueueNode\* front;

QueueNode\* rear;

} Queue;

// 初始化"默认字典"

void init\_defaultdict(DefaultDict\* dict) {

dict->entries = (DefaultDictEntry\*)malloc(MAX\_LIST\_SIZE \* sizeof(

DefaultDictEntry));

dict->size = 0;

}

// 在"默认字典"中查找条目

DefaultDictEntry\* find\_entry(DefaultDict\* dict, const char\* key) {

for (int i = 0; i < dict->size; i++) {

if (strcmp(dict->entries[i].key, key) == 0) {

return &dict->entries[i];

}

}

return NULL;

}

// "默认字典"中添加条目

void add\_to\_defaultdict(DefaultDict\* dict, const char\* key, const char\* value) {

DefaultDictEntry\* entry = find\_entry(dict, key);

if (entry == NULL) {

strcpy(dict->entries[dict->size].key, key);

dict->entries[dict->size].head = NULL;

entry = &dict->entries[dict->size];

dict->size++;

}

Node\* newNode = (Node\*)malloc(sizeof(Node));

strcpy(newNode->name, value);

newNode->next = entry->head;

entry->head = newNode;

}

// 初始化队列

void init\_queue(Queue\* queue) {

queue->front = NULL;

queue->rear = NULL;

}

// 检查队列是否为空

int is\_queue\_empty(Queue\* queue) {

return queue->front == NULL;

}

// 将新的元素添加到队列

void enqueue(Queue\* queue, const char\* name) {

QueueNode\* newNode = (QueueNode\*)malloc(sizeof(QueueNode));

strcpy(newNode->name, name);

newNode->next = NULL;

if (queue->rear) {

queue->rear->next = newNode;

} else {

queue->front = newNode;

}

queue->rear = newNode;

}

// 从队列中删除元素

void dequeue(Queue\* queue, char\* name) {

if (!is\_queue\_empty(queue)) {

QueueNode\* temp = queue->front;

strcpy(name, temp->name);

queue->front = temp->next;

if (!queue->front) {

queue->rear = NULL;

}

free(temp);

}

}

// 比较函数用于qsort

int cmp(const void\* a, const void\* b) {

return strcmp(\*(const char\*\*)a, \*(const char\*\*)b);

}

int main() {

int n;

scanf("%d", &n);

DefaultDict dict;

init\_defaultdict(&dict);

// 读取输入并填充"默认字典"

for (int i = 0; i < n; i++) {

char key[MAX\_NAME\_LEN], value[MAX\_NAME\_LEN];

scanf("%s %s", value, key);

add\_to\_defaultdict(&dict, key, value);

}

char go[MAX\_NAME\_LEN];

scanf("%s", go);

Queue queue;

init\_queue(&queue);

enqueue(&queue, go);

char\*\* ans = (char\*\*)malloc(MAX\_LIST\_SIZE \* sizeof(char\*));

int ans\_count = 0;

// 使用广度优先搜索

while (!is\_queue\_empty(&queue)) {

char now[MAX\_NAME\_LEN];

dequeue(&queue, now);

DefaultDictEntry\* entry = find\_entry(&dict, now);

if (entry) {

Node\* current = entry->head;

while (current) {

ans[ans\_count] = (char\*)malloc(MAX\_NAME\_LEN \* sizeof(char));

strcpy(ans[ans\_count], current->name);

ans\_count++;

enqueue(&queue, current->name);

current = current->next;

}

}

}

// 排序并输出结果

qsort(ans, ans\_count, sizeof(char\*), cmp);

for (int i = 0; i < ans\_count; i++) {

printf("%s\n", ans[i]);

free(ans[i]);

}

free(ans);

// 释放链表节点的内存

for (int i = 0; i < dict.size; i++) {

Node\* current = dict.entries[i].head;

while (current) {

Node\* temp = current;

current = current->next;

free(temp);

}

}

free(dict.entries);

return 0;

}