答案解析

卷1

数据结构

设计算法判断这棵树是否为完全二叉树。使用C语言编写

```
bool isCompleteTree(TreeNode* root) {
    if (root == NULL) {
        return true;
    }
    bool flag = false; // 是否遇到过不满足完全二叉树的情况
    TreeNode* queue[10000];
    int front = 0, rear = 0;
    queue[rear++] = root;
    while (front < rear) {
        TreeNode* node = queue[front++];
        if (node == NULL) {
            flag = true;
        } else {
            if (flag) { // 如果已经遇到过空节点，那么当前节点不应该存在左右子节点
                return false;
            }
            queue[rear++] = node->left;
            queue[rear++] = node->right;
        }
    }
    return true;
}
```

已知图的领接链表，设计算法生成对应的逆领接表，并要求算法时间复杂度为O(n+e),其中n和e为图中顶点个数和边的条数。

```

// 定义图的边的结构体
typedef struct Edge {
    int src; // 边的起点
    int dest; // 边的终点
    struct Edge *next; // 指向下一条边的指针
} Edge;

// 创建一个新的边
Edge *createEdge(int src, int dest) {
    Edge *newEdge = (Edge *)malloc(sizeof(Edge));
    newEdge->src = src;
    newEdge->dest = dest;
    newEdge->next = NULL;
    return newEdge;
}

// 为顶点创建逆邻接表
```

```
Edge **createReverseAdjList(Edge **adjList, int n) {
    Edge **reverseAdjList = (Edge **)malloc(n * sizeof(Edge *));
    for (int i = 0; i < n; i++) {
        reverseAdjList[i] = NULL;
    }

    for (int i = 0; i < n; i++) {
        Edge *current = adjList[i];
        while (current != NULL) {
            // 为每条边的终点创建一个新的边，并将其添加到逆邻接表中
            Edge *newEdge = createEdge(current->dest, i);
            newEdge->next = reverseAdjList[current->dest];
            reverseAdjList[current->dest] = newEdge;
            current = current->next;
        }
    }

    return reverseAdjList;
}
```

高级语言

malloc用法

```
#include <stdlib.h>

typedef struct TreeNode {
    int data; // 节点存储的数据
    struct TreeNode *left; // 左子节点指针
    struct TreeNode *right; // 右子节点指针
} TreeNode;

TreeNode* createTreeNode(int data) {
    TreeNode *node = (TreeNode*)malloc(sizeof(TreeNode)); // 申请内存
    return node;
}

typedef struct ListNode {
    int data; // 节点存储的数据
    struct ListNode *next; // 指向下一个节点的指针
} ListNode;

ListNode* createListNode(int data) {
    ListNode *node = (ListNode*)malloc(sizeof(ListNode)); // 申请内存
    return node;
}
```

卷2

数据结构

2、满k二叉树组成的完全k叉树（k>=1),n个节点，判断第i个节点（从1开始每个节点排序）

（1） 是否为根节点

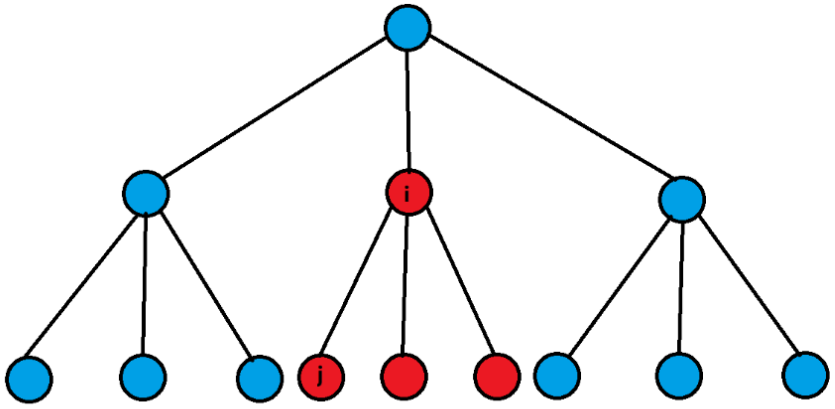
（2）是否为叶子结点

（3）求其第1个孩子的序号

（4）求双亲节点的序号

答案：

i = 1 就是根节点



设i的位置如上，j是i的第一个子节点。
i 的前面有（i-1）个节点，每一个节点有k 个子节点
所以j 前面有（（i-1）*k）+1 个节点*
*因为（i-1）*k 没有包括根节点。所以j 前面的节点再加一，`（（i-1）*k）+2` 就是j 节点的编号。

`（（i-1）*k）+2 > n` 说明是叶子结点

假设双亲节点为j

$$i = ((j-1)*k)+2$$

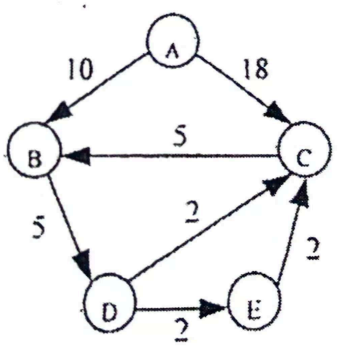
$$i-2 = (j-1)*k$$

$$(i-2)/k = j-1$$

$$j = ((i-2)/k)+1$$

3、无向带权图，使用Dijstra求顶点A到其他所有顶点的最短路径。（需要使用吉大的模板进行书写）

3. （8 分） 采用 Dijkstra 算法计算下图中从顶点 A 到其它各顶点的最短路径和路径长度。



	A	B	C	D	E
s	1	0	0	0	0
dist	0	∞	∞	∞	∞
Path					

	A	B	C	D	E
s	1	1	0	0	0
dist	0	10	18	∞	∞
Path		A	A		

	A	B	C	D	E
s	1	1	0	1	0
dist	0	10	18	15	∞
Path		A	A	B	

	A	B	C	D	E
s	1	1	1	1	0
dist	0	10	17	15	17
Path		A	D	B	D

	A	B	C	D	E
s	1	1	1	1	1
dist	0	10	17	15	17
Path		A	D	B	D

A到其它各顶点的最短路径和路径长度如下

B	A→B	10
C	A→B→D→C	17
D	A→B→D	15
E	A→B→D→E	17

2、指针root指向一颗left/right链接字段表示的二叉树T，data表示该节点的标号。设计算法求出给定节点值为x所有的父节点。编写函数`FindFather(Tree root, int x)`。

```
typedef struct TreeNode {
    int data; // 节点存储的数据
    struct TreeNode *left; // 左子节点指针
    struct TreeNode *right; // 右子节点指针
} *tree;

void FindFather(tree root, int x){
    tree stack[maxsize];
    int top = -1;
    tree p = root,r=NULL;
    while(p || top != -1){
        if(p){
            stack[++top] = p;
            p = p->left;
        }
```

```
        else{
            p = stack[top];
            if(p->right && p->right != r){
                p = p->right;
            }
            else{
                if(p->data == x) break;     //找到了
                top--;
                r = p;
                p = NULL;
            }
        }
    }
    printf("%d的祖先节点有：",x);
    while(top!=-1){
        printf("%d ", stack[top--]->data);
    }
}
```

3、编写算法判断无向带权图G的最小生成树（支撑树）是否唯一。

- 因为kruskal求最小生成树是基于排好序的边权值的，如果说能够取得两个最小值相同的边，那么最小生成树不唯一
- 算法实现就是，在kurskal算法实现过程中，判断相同权值的边是否处于不同的并查集，如果是，那么最小生成树不唯一

```
typedef struct Edge{//并查集的边节点 front,to为边的指向，w代表权值
    int front;
    int to;
    int w;
}Edge;

int father[maxsize] = {0};      //所属根节点

//查找根节点
int Find(int x){                //递归找到所属根节点
    if(father[x]<=0){           //找到了
        return x;
    }                           //递归查找根节点
    return father[x] = Find(father[x]);
}

//按秩合并
void Union(int x, int y){       //按秩合并两个并查集  秩：当前节点深度乘以-1
    int fx = Find(x);
    int fy = Find(y);
    if(fx == fy) return;        //两者属于同一个并查集，无需合并
    if(father[fx] < father[fy]){//fy深度 小于 fx深度 ，fy合并到fx中
        father[fy] = fx;
    }
```

```
    else{                           //否则 fx合并到 fy中
        if(father[fx] == father[fy]){   //秩相同时，合并深度加1 ，秩要减1，
            father[fy]--;
        }
        father[fx] = fy;
    }
}

//对边进行排序
int SortEdge(int graph[][maxsize], int n, Edge edge[]){     //实现两个功能，将图转化为并查集表示，同时对并查集进行排序
    int  edgenum = 0;                   //边的个数
    for(int i = 0; i <= n; i++)         //无向图遍历上半矩阵即可
        for(int j = i+1; j <= n; j++)
            if(graph[i][j]){            //存在边，赋值
                edgenum++;
                edge[edgenum].front = i;
                edge[edgenum].to = j;
                edge[edgenum].w = graph[i][j];
            }
    //冒泡排序对边进行排序
    for(int i = edgenum; i > 1; i--){
        int flag = 0;
        for(int j =1; j < i; j++){
            if(edge[j].w > edge[j+1].w){    //递增排序   求最大生成树只需要改为递减排序即可
                Edge temp = edge[j];
                edge[j] = edge[j+1];
                edge[j+1] = temp;
                flag = 1;                   //本趟进行了交换
            }
        }
        if(flag == 0){                      //该趟未排序，已经有序
            break;
        }
    }
    return edgenum;
}

//判断最小生成树是否唯一
bool Kruskal_Isunique(int graph[][maxsize], int n) {
    edgenum = SortEdge(graph,n,edge);
    int setnum = n;
    int k = 1;
    int ans = 0;
    while (setnum > 1) {        //连通分量个数大于1              构造最小生成树的过程
        int x = edge[k].front;
        int y = edge[k].to;
        int w = edge[k].w;
        k++;
        if (Find(x) == Find(y)) continue;       //这条边的两个顶点从属于一个集合舍弃这条边
        for (int i = k; i <= edgenum; i++) {
            if (edge[i].w == w) {           //如果这条新边的权值与刚才那条边权值相同
                int newa = edge[i].front;
                int newb = edge[i].to;
```

```
                //若两边的所属连通块相同，说明最小生成树不唯一；需注意此时还没有将a，b为顶点的
边合并
                if (Find(x) == Find(newa) && Find(y) == Find(newb) || Find(x) ==
Find(newb) && Find(y) == Find(newa)){
                    printf("边 %d--%d 与边 %d-- %d 可任意选，最小生成树不唯一\n", x,
y, newa, newb);
                    return false;
                }

            }
        }
        Union(x, y);            //已经确保该条边没有可替代的边，将这条边合并
        setnum--;               //合并之后连通分量－1
        ans += w;
    }
    return true;
}
```

高级语言

2、一个大于1的自然数，除了1和它自身外，不能被其他自然数整除的数叫做质数，编写一个完整程序，该程序的功能是，输入一个整数n，若n<2，则输出0，若n是质数，则输出n的值，否则将这个整数分解成质因数相乘，例如90打印出90=2 * 3 * 3 * 5。

```
//将一个正整数分解成质因数
/*思路:
    1、从2开始的质因数不断除以该正整数n,能整除就输出，直到不能整除
    2、换下一个质因数去除，重复上述操作，直到输出最后一个质因数
*/
int main(){
    int i,n;
    printf("请输入n:");
    scanf("%d",&n);
    printf("%d=",n);
    for(int i = 2; i < n; i++){
        while(i){//一直循环到该质因数不能被整除后跳到else中
            if(n%i == 0){
            n=n/i;
            printf("%d*",i);
            }
            else{//该个质因数已经除完了，到下一个质因数
                break;
            }
        }
    }
    printf("%d",n);//只剩下最后一个质因数
    return 0;
}
```

卷3

数据结构

3、自由树（无环连通图）`T=(V,E)`的直径是指树种所有顶点之间最短路径的最大值。试设计一个时间复杂度尽可能低的算法求`T`的直径，并分析算法的时间复杂度。

思路：

1. 先从任意一个顶点找到该顶点最短路径中最长的一个，这个点为直径的某个端点
2. 然后再从这个顶点出发，再进行一次BFS,求出距离该顶点最短路径最长的一个，该顶点为直径的另一端
3. 返回两个顶点的路径长度，就是自由树的直径

```
typedef struct ArcNode{     //边节点
    int adjvex;
    struct ArcNode *next;
}ArcNode;

typedef struct VNode{       //顶点节点
    int data;
    struct ArcNode *firstarc;
}VNode;

typedef struct AGraph{      //领接表
    VNode adjlist[maxsize];
    int vexnum, edgenum;
}AGraph;

int MaxLenBFS(AGraph *G, int v, int dist[]){
    int visited[maxsize] = {0};
    int queue[maxsize];
    int front = -1, rear = -1,i,k,temp,max=0;
    ArcNode *p = G->adjlist[v].firstarc;

    for(i = 0; i < G->vexnum; i++){
        dist[i] = -1;
    }

    queue[++rear] = v;
    visited[v] = 1;
    dist[v] = 0;

    while(rear != front){
        k = queue[++front];
        p = G->adjlist[k].firstarc;
        while(p!=NULL){
            temp = p->adjvex;
            if(visited[temp]==0){
                queue[++rear] = temp;
                visited[temp] = 1;
                dist[temp] = dist[k] + 1;
            }
            p = p->next;
```

```
        }
    }

    for(i=0; i < G->vexnum; i++){
        if(dist[i]>dist[max]){
            max = i;
        }
    }
    return max;          //返回端点
}

int Diameter(AGraph *G){
    int dist[maxsize];
    int first = MaxLenBFS(G,0,dist);
    int last = MaxLenBFS(G,first,dist);
    printf("直径为：%d",dist[last]);
    return dist[last];                    //返回直径长度
}
```

高级语言

1、编写对应c语言程序，统计字符串S2,在S1中出现的次数。其中字符串以"\0"为结束符。

```
#include <stdio.h>
#include <string.h>

// 函数用于统计S2在S1中出现的次数
int countOccurrences(char *s1, char *s2) {
    int count = 0;
    char *ptr = s1; // 指向S1的指针

    while (*ptr) { // 遍历S1直到结束符
        // 查找S2在S1中的匹配项
        char *match = strstr(ptr, s2);
        if (match) {
            count++; // 如果找到匹配项，计数增加
            ptr = match + strlen(s2); // 移动指针到S2的下一个可能开始位置
        } else {
            break; // 如果没有找到，退出循环
        }
    }

    return count;
}

int main() {
    char s1[1000]; // 假设S1的最大长度为999字符加上结束符
    char s2[100];  // 假设S2的最大长度为99字符加上结束符

    // 输入S1和S2
    printf("Enter string S1: ");
    fgets(s1, sizeof(s1), stdin);
```

```
    printf("Enter string S2: ");
    fgets(s2, sizeof(s2), stdin);

    // 统计S2在S1中出现的次数
    int occurrences = countOccurrences(s1, s2);

    // 输出结果
    printf("String '%s' appears %d times in string '%s'.\n", s2, occurrences,
s1);

    return 0;
}
```

简单介绍一下c语言的字符串函数

`strcpy(char *dest, char *src)` - 将 `src` 字符串复制到 `dest` 缓冲区。不检查目标缓冲区的大小，可能导致溢出。

`strcat(char *dest, char *src)` - 将 `src` 字符串连接到 `dest` 字符串的末尾。同样不检查大小，可能导致溢出。

`strcmp( char *str1, char *str2)` - 比较两个字符串 `str1` 和 `str2`。如果 `str1` 小于 `str2`，返回负数；如果相等，返回 `0`；如果 `str1` 大于 `str2`，返回正数。

`strlen( char *str)` - 返回字符串 `str` 的长度（不包括结束的 `\0`）。

`strstr( char *haystack, char *needle)` - 在 `haystack` 字符串中查找 `needle` 字符串的第一次出现。

2、现在有一批职工需要从键盘中获取信息。职工信息包括：工号（id)、姓名(name)、年龄(age)。定义对应的结构体，从键盘中读入n个职工信息，节点顺序与读入的一致。

```
#include <stdio.h>
#include <stdlib.h>

// 定义链表节点结构体，包含职工信息
typedef struct EmployeeNode {
    int id;    // 工号
    char name[50]; // 姓名
    int age;   // 年龄
    struct EmployeeNode *next; // 指向下一个节点的指针
} EmployeeNode;

// 创建新节点的函数
EmployeeNode* createNode(int id, const char *name, int age) {
    EmployeeNode *newNode = (EmployeeNode *)malloc(sizeof(EmployeeNode));
    if (newNode == NULL) {
        perror("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    newNode->id = id;
```

```
    strncpy(newNode->name, name, sizeof(newNode->name) - 1);
    newNode->name[sizeof(newNode->name) - 1] = '\0'; // 确保字符串以null结尾
    newNode->age = age;
    newNode->next = NULL;
    return newNode;
}

// 插入节点到链表末尾的函数
void appendNode(EmployeeNode **head, EmployeeNode *newNode) {
    if (*head == NULL) {
        *head = newNode;
    } else {
        EmployeeNode *current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

int main() {
    int n; // 职工数量
    scanf("%d", &n); // 读取职工数量
    EmployeeNode *head = NULL; // 链表头指针初始化为NULL

    for (int i = 0; i < n; i++) {
        int id;
        char name[50];
        int age;

        // 使用单个scanf读取一行数据，包含id、name和age
        scanf("%d %49s %d", &id, name, &age)
        // 创建新节点并追加到链表末尾
        EmployeeNode *newNode = createNode(id, name, age);
        appendNode(&head, newNode);
    }

    return 0;
}
```

3、在第二题的基础上，将对应结构体中工号重复的节点删除，并将剩余节点写入到"worker.txt"文件当中。

```
// 删除具有重复工号的节点
void removeDuplicates(EmployeeNode **head) {
    EmployeeNode *current = *head;
    while (current != NULL) {
        EmployeeNode *next = current->next;
        while (next != NULL) {
            if (current->id == next->id) {
                // 移除重复的节点
                EmployeeNode *temp = next;
```

```
                next->next = next->next; // 跳过这个节点
                free(temp);
            } else {
                next = next->next;
            }
        }
        current = current->next;
    }
}

// 写入链表到文件
void writeToFile(EmployeeNode *head, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file");
        return;
    }

    while (head != NULL) {
        fprintf(file, "ID: %d, Name: %s, Age: %d\n", head->id, head->name, head-
>age);
        head = head->next;
    }

    fclose(file);
}
```

简单介绍一下C语言文件操作

打开文件

使用`fopen`函数打开文件，它需要两个参数：文件路径和模式。例如：

```
FILE *fp;

// 以读模式打开文件
fp = fopen("example.txt", "r");

// 以写模式打开文件，如果文件不存在则创建
fp = fopen("example.txt", "w");

// 以追加模式打开文件，如果文件不存在则创建
fp = fopen("example.txt", "a");
```

如果文件成功打开，`fopen`返回一个`FILE`指针；如果失败，返回`NULL`。

读取文件

使用`fscanf`、`fgets`或`fread`等函数从文件中读取数据：

```c
// 使用fscanf按指定格式从文件读取
fscanf(fp, "%d %s %f", &number, buffer, &floatNumber);

// 使用fgets读取一行文本
char line[100];
fgets(line, sizeof(line), fp);

// 使用fread按二进制方式读取数据
size_t result = fread(buffer, sizeof(char), bufferSize, fp);
```

写入文件

使用`fprintf`、`fputs`或`fwrite`等函数向文件写入数据：

```c
// 使用fprintf按指定格式向文件写入
fprintf(fp, "Number: %d, String: %s, Float: %f", number, string, floatNumber);

// 使用fputs写入一行文本
fputs("Hello, world!\n", fp);

// 使用fwrite按二进制方式写入数据
size_t result = fwrite(dataBuffer, sizeof(char), dataSize, fp);
```

关闭文件

操作完成后，使用`fclose`函数关闭文件：

```c
fclose(fp);
```

错误检查

文件操作可能会失败，因此通常需要检查每个操作的返回值：

```c
if (fp == NULL) {
    // 打开文件失败
    printf("Error opening file");
} else if (fscanf(fp, "%d", &number) != 1) {
    // 读取数据失败
    fprintf(stderr, "Error reading from file.\n");
}
```