Part IV 数据结构

一、遍历链表

建立一个升序链表并遍历输出。 **输入描述:**

> 输入的每个案例中第一行包括1个整数：$n(1 \leq n \leq 1000)$，接下来的一行包括n个整数。

输出描述:

> 可能有多组测试数据，对于每组数据， 将n个整数建立升序链表，之后遍历链表并输出。

示例1 输入

```
4 3 5 7 9
```

输出

```
3 5 7 9
```

完整代码实现：

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 1005
typedef struct LNode {
    int data;
    struct LNode* next;
}LNode, *LinkList;

// 链表初始化
LinkList initList(){
    LinkList L = (LNode*)malloc(sizeof(LNode));
    L -> next = NULL;
    return L;
}

// 插入结点
void insertNode(LNode* L,int num){
    LNode *p = L -> next;
    LinkList pre = L;
    LinkList q;
    while(p) {
        if(p -> data < num) {
            pre = p;
            p = p -> next;
```

```
        } else {
            break;
        }
    }
    q = (LNode*)malloc(sizeof(LNode));
    q -> data = num;
    pre -> next = q;
    q -> next = p;
}

// 输出链表
void print(LinkList L){
    LinkList p = L -> next;
    while(p -> next) {
        printf("%d ",p -> data);
        p = p -> next;
    }
    printf("%d",p -> data);
}

// 销毁链表
void destroyList(LinkList L) {
    LNode *p = L, *temp;
    while (p) {
        temp = p;
        p = p -> next;
        free(temp);
    }
}

int main(){
    int i, n, num;
    while(scanf("%d",&n)!=EOF){
        LinkList L;
        L = initList();
        for(i = 0;i < n;i++) {
            scanf("%d", &num);
            insertNode(L, num);
        }
        print(L);
        destroyList(L);             // 销毁链表，防止内存泄漏
    }
    return 0;
}
```

二、堆栈的使用

堆栈是一种基本的数据结构。堆栈具有两种基本操作方式，push 和 pop。push一个值会将其压入栈顶，而 pop 则会将栈顶的值弹出。现在我们就来验证一下堆栈的使用。 **输入描述:**

> 对于每组测试数据，第一行是一个正整数 n，0<n<=10000(n=0 结束)。而后的 n 行，每行的第一个字符可能是'P'或者'O'或者'A'；如果是'P'，后面还会跟着一个整数，表示把这个数据压入堆栈；如果是'O'，表示将栈顶的值 pop 出来，如果堆栈中没有元素时，忽略本次操作；如果是'A'，表示询问当前栈顶的值，如果当时栈为空，则输出'E'。堆栈开始为空。

输出描述:

> 对于每组测试数据，根据其中的命令字符来处理堆栈；并对所有的'A'操作，输出当时栈顶的值，每个占据一行，如果当时栈为空，则输出'E'。当每组测试数据完成后，输出一个空行。

示例1 输入

```
3 A P 5 A 4 P 3 P 6 O A
```

输出

```
E 5
```

```
3
```

完整代码实现(C语言)

```c
#include <stdio.h>
#include <string.h>
#define N 10005
int top;

void push(int stack[N], int num) {
    stack[top++] = num;
}

void pop() {
    if(top > 0) {
        top--;
    }
}

void output(int stack[N]) {
    if(top == 0) {
        printf("E\n");
    } else {
        printf("%d\n", stack[top - 1]);
    }
}

int main() {
    int i, n, num;
    int stack[N];
    char c;
```

```
    while(scanf("%d", &n) != EOF && n > 0) {
        top = 0;                //栈顶指针初始化
        for(i = 0; i < n; i++) {
            c = getchar();
            scanf("%c", &c);
            switch(c){
                case 'A' : output(stack);
                           break;
                case 'P' : scanf("%d", &num);
                           push(stack, num);
                           break;
                case 'O' : pop();
                           break;
            }
        }
        printf("\n");
    }
    return 0;
}
```

完整代码实现(C++语言)

```
#include <cstdio>
#include <stack>
using namespace std;

int main() {
    stack<int> S;
    int n, num;
    char c;
    while(scanf("%d", &n) != EOF && n != 0){
        for(int i = 0; i < n; i++){
            c = getchar();
            scanf("%c", &c);
            switch(c){
                case 'A' : if(S.empty()) {
                               printf("E\n");
                           } else {
                                printf("%d\n", S.top());
                           }
                           break;
                case 'P' : scanf("%d", &num);
                           S.push(num);
                           break;
                case 'O' : if(!S.empty()) {
                               S.pop();
                           }
                           break;
            }
        }
        printf("\n");
        while(!S.empty()) {
```

```
            S.pop();
        }
    }
    return 0;
}
```

三、哈夫曼树

> 哈夫曼树，第一行输入一个数n，表示叶结点的个数。需要用这些叶结点生成哈夫曼树，根据哈夫曼树的概念，这些结点有权值，即weight，题目需要输出最小带权路径总和。

输入描述:

> 输入有多组数据。 每组第一行输入一个数n，接着输入n个叶节点（叶节点权值不超过100，$2 \leq n \leq 1000$）。

输出描述:

> 输出权值。

示例1 输入

> 5
> 1 2 2 5 9

输出

> 37

题解：

- 哈夫曼树的带权路径之和为各叶子节点的权值与路径长度（层次数减一或从根节点到达叶子节点路径上的非叶子节点的个数）之积的和。在构造哈夫曼树时，非叶子节点也会带有权值（为其子节点的权值之和），当加上一个非叶子节点的权值时相当于加上以其为根节点的子树的所有叶子节点的权值。而加上从根节点到叶子节点路径上的非叶子节点权值，就包含了该叶子节点的带权路径长度。**所以带权路径之和即为所有非叶子节点的权值之和。**
- 接下来就是按照构造哈夫曼树的方法求出各非叶子节点的权值。关键在于选取节点中最小的两个节点，删除这两节点并添加新的节点。因此可以建立小顶堆，假设元素数为n，堆顶元素即为最小值，可通过将最后的节点提到堆顶，然后进行调整算法建立元素数为n-1的小顶堆，将堆顶元素与第一次选取的值相加便得到了新的节点的权值，与sum相加后，再次调整为小顶堆。重复操作直至元素数为1。而小顶堆可以利用优先队列来实现，使用C++ 标准模板库中的优先队列即可。

完整代码实现：

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
```

```
    int n, temp, a, b;
    int weight;
    priority_queue<int,vector<int>, greater<int> > minheap;
    while(cin >> n) {
        weight = 0;
        for(int i = 0;i < n;i++) {
            cin >> temp;
            minheap.push(temp);
        }
        while(minheap.size() != 1) {
            a = minheap.top();
            minheap.pop();
            b = minheap.top();
            minheap.pop();
            weight = weight + a +b;
            minheap.push(a + b);

        }
        cout << weight << endl;
        minheap.pop();
    }
    return 0;
}
```

四、二叉树遍历

> 编一个程序，读入用户输入的一串先序遍历字符串，根据此字符串建立一个二叉树（以指针方式存储）。 例如如下的先序遍历字符串： ABC##DE#G##F### 其中“#”表示的是空格，空格字符代表空树。建立起此二叉树以后，再对二叉树进行中序遍历，输出遍历结果。

输入描述:

> 输入包括1行字符串，长度不超过100。

输出描述:

> 可能有多组测试数据，对于每组数据， 输出将输入字符串建立二叉树后中序遍历的序列，每个字符后面都有一个空格。 每个输出结果占一行。

示例1 输入

> abc##de#g##f###

输出

> c b e g d f a

完整代码实现：

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 105

typedef struct Node {
    char c;
    struct Node *lchild;
    struct Node *rchild;
} Node;
int pos;            //标记字符串处理到哪了
char str[N];            //读取的字符串

// 创建新节点，返回结点指针
Node* createNode() {
    Node *ret = (Node*)malloc(sizeof(Node));
    ret -> lchild = NULL;
    ret -> rchild = NULL;
    return ret;
}

// 中序遍历
void inOrder(Node* T) {
    if (T) {
        inOrder(T -> lchild);
        printf("%c ", T -> c);
        inOrder(T -> rchild);
    }
}

// 释放二叉树所占内存
void del(Node* T) {
    // 删除左子树
    if(T -> lchild != NULL) {
        del(T -> lchild);
        T -> lchild = NULL;
    }
    // 删除右子树
    if(T -> rchild != NULL) {
        del(T -> rchild);
        T -> rchild = NULL;
    }
    // 删除根节点
    free(T);
}

// 根据字符串创立二叉树，并返回根节点指针
Node* buildTree() {
    // 字符串处理完毕
    if(pos >= strlen(str)) {
        return NULL;
    }
```

```
    //创建空节点，即返回空指针
    if(str[pos] == '#') {
        pos++;              //准备处理下一个字符
        return NULL;
    }
    Node *p = createNode();           //创建一个空节点
    p -> c = str[pos++];              //先序，先获取根节点的字符信息
    p -> lchild = buildTree();        //创建左子树
    p -> rchild = buildTree();        //创建右子树
    return p;                         //返回根节点指针
}

int main() {
    while(gets(str)) {
        pos = 0;                        //标记字符串处理到哪了
        Node* T = buildTree();          //根据字符串构建整棵树
        inOrder(T);                     //中序遍历并输出
        printf("\n");
        del(T);                         //释放内存空间
    }
    return 0;
}
```

五、二叉排序树

> 二叉排序树，也称为二叉查找树。可以是一颗空树，也可以是一颗具有如下特性的非空二叉树：
>
> - 1.若左子树非空，则左子树上所有节点关键字值均不大于根节点的关键字值；
> - 2.若右子树非空，则右子树上所有节点关键字值均不小于根节点的关键字值；
> - 3.左、右子树本身也是一颗二叉排序树。

> 现在给你N个关键字值各不相同的节点，要求你按顺序插入一个初始为空树的二叉排序树中，每次插入后成功后，求相应的父亲节点的关键字值，如果没有父亲节点，则输出-1。

输入描述:

> 输入包含多组测试数据，每组测试数据两行。 第一行，一个数字N（N<=100），表示待插入的节点数。 第二行，N个互不相同的正整数，表示要顺序插入节点的关键字值，这些值不超过10^8。

输出描述:

> 输出共N行，每次插入节点后，该节点对应的父亲节点的关键字值。

示例1 输入

```
5 2 5 1 3 4
```

输出

```
-1 2 2 5 3
```

完整代码实现：

```
#include <stdio.h>
typedef struct TNode {
    int data;
    struct TNode *left, *right;
} *BiTree, TNode;

int loc;
TNode T[101];

BiTree creat() {
    T[loc].left = T[loc].right = NULL;
    return &T[loc++];
}

BiTree insertNode(BiTree T, int x, int father) {
    if(!T) {
        T = creat();
        T -> data = x;
        printf("%d\n",father);
        return T;
    }
    if(x < T -> data) {
        T -> left = insertNode(T -> left, x, T -> data);
    } else if(x > T -> data) {
        T -> right = insertNode(T -> right, x, T -> data);
    }
    return T;
}
int main() {
    int i, n, temp;
    TNode* T;
    while(scanf("%d",&n) != EOF) {
        T = NULL;
        loc = 0;
        for(i = 0;i < n;i++) {
            scanf("%d",&temp);
            T = insertNode(T, temp, -1);
        }
    }
    return 0;
}
```

Part V 图论

一、畅通工程1

> 某省调查城镇交通状况，得到现有城镇道路统计表，表中列出了每条道路直接连通的城镇。省政府“畅通工程”的目标是使全省任何两个城镇间都可以实现交通（但不一定有直接的道路相连，只要互相间接通过道路可达即可）。问最少还需要建设多少条道路?

输入描述:

> 测试输入包含若干测试用例。每个测试用例的第1行给出两个正整数，分别是城镇数目N (< 1000) 和道路数目M；随后的M行对应M条道路，每行给出一对正整数，分别是该条道路直接连通的两个城镇的编号。为简单起见，城镇从1到N编号。 注意:两个城市之间可以有多条道路相通,也就是说 3 3 1 2 1 2 2 1 这种输入也是合法的 当N为0时，输入结束，该用例不被处理。

输出描述:

> 对每个测试用例，在1行里输出最少还需要建设的道路数目。

示例1 输入

> 4 2 1 3 4 3 3 3 1 2 1 3 2 3 5 2 1 2 3 5 999 0 0

输出

> 1 0 2 998

题解： 题目相当于是在问有多少个连通分量，本质上考察的是并查集的基本应用。完整代码实现如下：

```
#include <stdio.h>
#define MAX_SIZE 1005

int parent[MAX_SIZE];       //parent[i]表示节点i的双亲是谁, -1表示其为根节点

//寻找x所在的树的根节点
int findRoot(int x) {
    int ret;
    if(parent[x] == -1) {
        return x;
    } else {
        ret = findRoot(parent[x]);
        parent[x] = ret;              //路径压缩
        return ret;
    }
}

int main() {
    int i, m, n, x, y;
    int count;
    while(scanf("%d", &n) != EOF && n) {
        scanf("%d", &m);
        for(i = 1; i <= n; i++) {
```

```
            parent[i] = -1;             //parent数组初始化，一开始全是根
        }
        while(m--) {
            scanf("%d %d", &x, &y);
            x = findRoot(x);
            y = findRoot(y);
            if(x != y) {                //不在同一棵树那就合并
                parent[y] = x;
            }
        }
        count=0;                        //统计根节点数量，也就是统计连通分量的个数
        for(i = 1; i <= n; i++) {
            if(parent[i] == -1) {
                count++;
            }
        }
        printf("%d\n", count - 1);
    }
    return 0;
}
```

二、畅通工程2

> 省政府“畅通工程”的目标是使全省任何两个村庄间都可以实现公路交通（但不一定有直接的公路相连，只要能间接通过公路可达即可）。经过调查评估，得到的统计表中列出了有可能建设公路的若干条道路的成本。现请你编写程序，计算出全省畅通需要的最低成本。

输入描述:

> 测试输入包含若干测试用例。每个测试用例的第1行给出评估的道路条数 N、村庄数目 $M(N, M \leq 100)$；随后的N行对应村庄间道路的成本，每行给出一对正整数，分别是两个村庄的编号，以及此两村庄间道路的成本（也是正整数）。为简单起见，村庄从1到M编号。当N为0时，全部输入结束，相应的结果不要输出。

输出描述:

> 对每个测试用例，在1行里输出全省畅通需要的最低成本。若统计数据不足以保证畅通，则输出“?”。

示例1 输入

> 3 3 1 2 1 1 3 2 2 3 4 1 3 2 3 2 0 100 **输出** 3 ?

题解： 题目相当于是在问给定图是否可以找到最小生成树，若能找到，则输出最小生成树的边权值之和，若不能，则输出”?“。因此可以采用Kruskal算法求解最小生成树。完整代码实现如下：

```
#include <stdio.h>
#include <algorithm>
#define N 105
```

```cpp
using namespace std;
int Tree[N];
typedef struct Edge {
    int a, b;
    int cost;
} Edge;

int findRoot(int x) {
    if(Tree[x] == -1) {
        return x;
    } else {
        Tree[x] = findRoot(Tree[x]);        // 路径压缩
        return Tree[x];
    }
}

bool cmp(Edge a,Edge b) {
    return a.cost < b.cost;
}

int main() {
    int i, n, m;
    int a, b, ans, count;
    Edge e[N];
    while(scanf("%d%d",&n,&m)!=EOF && n) {
        for(i = 1;i <= m;i++) {
            Tree[i] = -1;
        }
        for(i = 0;i < n;i++) {
            scanf("%d %d %d",&e[i].a, &e[i].b, &e[i].cost);
        }
        sort(e, e+n, cmp);
        ans = 0;
        for(i = 0;i < n;i++) {
            a = findRoot(e[i].a);
            b = findRoot(e[i].b);
            if(a != b) {
                Tree[b] = a;
                ans += e[i].cost;
            }
        }
        count = 0;
        for(i = 1;i <= m;i++) {
            if(Tree[i] == -1) {
                count++;
            }
        }
        if(count == 1) {
            printf("%d\n",ans);
        } else {
            puts("?");
        }
```

```
    }
    return 0;
}
```

三、最短路径问题

> 给你n个点，m条无向边，每条边都有长度d和花费p，给你起点s终点t，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入描述：

> 输入n, m，点的编号是1~n,然后是m行，每行4个数 a,b,d,p，表示a和b之间有一条边，且其长度为d，花费为p。最后一行是两个数 s,t;起点s，终点t。n和m为0时输入结束。
> $(1 < n \leq 1000, 0 < m < 100000, s \neq t)$

输出描述:

> 输出一行有两个数，最短距离及其花费。

示例1 输入

> 3 2 1 2 5 6 2 3 4 5 1 3 0 0

输出

> 9 11

题解： 注意这道题的测试用例有坑点，两点之间可以有多条边(实际中也要根据题目描述来判断是否可能有重边)，需要保留路径最短的那条(或是路径长度一样但是花费最少的那条)，这需要把输入数据化成“简单无向图”(或者不处理这样的数据，在dijkstra算法中直接处理即可)。由于是单源最短路径问题，因此采用dijkstra算法效率更高，完整代码实现如下：

```
#include<stdio.h>
#include<vector>
#define N 1005
#define INF 0x3f3f3f3f
using namespace std;

struct E{
    int next;
    int cost;
    int dis;
};

vector<E> edge[N];              // 存放图
bool mark[N];                   // 标记点是否已经找到最短路径
int dis[N];                     // 用来存储最短路径，表示从起始点到当前点的最短路径
int cost[N];                    // 用来存储路径上的花费，表示从起始点到当前点的花费
```

```
int main(){
    int m, n;
    int a, b, d, p;
    E temp;
    int s,t;
    while(scanf("%d %d",&n, &m) != EOF) {
        if(m == 0 && n == 0) {
            break;
        }

        for(int i = 1;i <= n;i++) {
            edge[i].clear();
            dis[i] = -1;
            cost[i] = 0;
            mark[i] = false;
        }
        while(m--) {
            scanf("%d %d %d %d",&a, &b, &d, &p);
            temp.dis = d;
            temp.cost = p;
            temp.next = b;
            edge[a].push_back(temp);                // 无向图，所以要添加双向的边
            temp.next = a;
            edge[b].push_back(temp);
        }
        scanf("%d %d",&s, &t);
        int newP = s;
        mark[s] = true;
        dis[s] = 0;
        // 找源点到其他n-1个点的最短路径
        for(int i = 1;i < n;i++) {
            // 遍历newP所有邻接边，重复的边也会遍历
            for(int j = 0;j < edge[newP].size();j++) {
                int next = edge[newP][j].next;
                int p = edge[newP][j].cost;
                int d = edge[newP][j].dis;
                if(mark[next] == true) {                // 已经找到最短路径
                    continue;
                }
                // 满足条件则更新dis数组和cost数组
                if(dis[next] == -1 || dis[next] > dis[newP] + d || (dis[next] ==
dis[newP] + d && cost[next] > cost[newP] + p)) {
                    dis[next] = dis[newP] + d;
                    cost[next] = cost[newP] + p;
                }
            }
            int min = INF;
            // 找到本轮迭代的最短路径
            for(int j = 1;j <= n;j++) {
                if(mark[j] == false && dis[j] != -1 && dis[j] < min) {
                    min = dis[j];
```

```
                    newP = j;
                }
            }
            // 标记已找到最短路径
            mark[newP] = true;
        }
        printf("%d %d\n",dis[t], cost[t]);
    }
    return 0;
}
```