

**華東理工大學**  
EAST CHINA UNIVERSITY OF SCIENCE AND TECHNOLOGY

# 《 算法与数据结构 》

## 实验报告本

班 级： 计算机类 2105

学 号： 21013134

姓 名： 徐昊博

指导教师： 张静

信息科学与工程学院

2022 年 5 月

# 实验报告（1）

实验名称：栈和队列	实验地点：在线实验
所使用的工具软件及环境 Win10/Win 7, Visual C++/Java	
<b>一、实验目的：</b> 通过上机实验，要求掌握线性表、栈、队列和字符串的应用算法。	
<b>二、实验内容描述：</b> （填写题目内容及输入输出要求） （1）使用顺序表或链表实现多项式的加法运算。 输入：在一行输入一串数字，两两一组，分别为多项式系数和指数，中间用空格分隔，结束符为“0 0”。如：“9 3 1 1 1 0 0 0”表示“多项式 $9x^3+x+1$ ”。然后输入一个“+”号，继续输入一行多项式数据。 输出：多项式相加后的多项式。如“相加后多项式 $9x^3+x+1$ ” （2）假设某火车站采用后进先出的模式。现有 n 列火车，调度人员给出火车进站的序列，并给出火车出站的序列，判断在这个调度要求能否实现，如果能实现写出火车进站、出站的操作序列。（栈） 输入：第一行为一个正整数 N 代表火车数量；第二行为 N 个字母，中间用空格分开，代表 N 个火车的进站顺序；第三行为 N 个字母，中间用空格分开，代表 N 个火车的离站顺序。 输出：第一行输出 0 或 1，0 代表该调度无法实现，1 代表可以实现；如果可以实现，请在第二行输出进站出站序列。表示进站时在字母后加上 ‘_in’，出站加上 ‘_out’。	
<b>三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）</b>  <b>第一题：使用顺序表或链表实现多项式的加法运算。</b>  <b>源代码：</b>  <pre>#include&lt;stdio.h&gt;  #include&lt;stdlib.h&gt;  #define INFINITY 1000000000  typedef struct LNode *PtrtoNode;  struct LNode {     int xishu,zhishu;      PtrtoNode Next; };  typedef PtrtoNode List;  int main() {     int a,b;</pre>	

```

PtrtoNode L=(List)malloc(sizeof(struct LNode));//头结点申请空间

PtrtoNode Head=L;

L->Next=NULL;

L->xishu=INFINITY;

L->zhishu=INFINITY;

scanf("%d%d",&a, &b);

if(a==0&&b==0) { printf("0"); return 0;}

A:while(a!=0 | b!=0)

{

    List i=Head;

    while(i!=NULL)

    {

        if(i->zhishu==b) //判断是否可以合并同类项

        {

            i->xishu=i->xishu+a;

            scanf("%d%d",&a,&b);

            goto A;

        }

        else i=i->Next;

    }

    PtrtoNode p=(List)malloc(sizeof(struct LNode));

    p->xishu=a;

    p->zhishu=b;

    p->Next = NULL;

    L->Next=p;

    L=L->Next;

    scanf("%d%d",&a,&b);

}

```

```

char p;

p=getchar();

scanf("%d%d",&a,&b);

if(a==0&&b==0) {List i=Head->Next; goto B;}

A1:while(a!=0 | b!=0)
{
    List i=Head;
    while(i!=NULL)
    {
        if(i->zhishu==b) //判断是否可以合并同类项
        {
            i->xishu=i->xishu+a;
            scanf("%d%d",&a,&b);
            goto A1;
        }
        else i=i->Next;
    }

    PtrtoNode p=(List)malloc(sizeof(struct LNode));

    p->xishu=a;
    p->zhishu=b;
    p->Next=NULL;
    L->Next=p;
    L=L->Next;
    scanf("%d%d",&a,&b);
}

List i = Head->Next;

B:if(i->xishu==1)

```

```

{
    if(i->zhishu==1) printf("x");

    else if (i->zhishu==0) printf("1");

    else printf("x^%d",i->zhishu);
}

else if (i->xishu==0)
{
    i=i->Next;

    goto B;
}

else if (i->zhishu==1) printf("%dx",i->xishu);
else if (i->zhishu==0) printf("%d",i->xishu);
else printf("%d^x%d",i->xishu,i->zhishu);

i=i->Next;

for(;i!=NULL;i=i->Next)
{
    if(i->xishu==1)
    {
        if(i->zhishu==1) {printf("+x"); continue;}

        else if (i->zhishu==0) {printf("+1"); continue;}

    }

    else if (i->xishu==0) continue;

    if (i->zhishu==1) printf("+%dx",i->xishu);

    else if (i->zhishu==0) printf("+%d",i->xishu);

    else printf("+%d^x%d",i->xishu,i->zhishu);

}

return 0;
}

```

程序运行结果：

C:\Users\Bo\Desktop\数据结构代码第一章\实验一第一题.exe

```
9 3 1 1 0 0+9 3 1 4 0 0
18^x3+x+x^4
Process returned 0 (0x0)    execution time : 9.996 s
Press any key to continue.
```

实验第一题采用的数据结构是栈的数据结构，通过每一次数据的输入，包括多项式的系数与指数，在每一次的输入时对输入的数据进行检验，检验过程为：将当前已存入数据的栈进行遍历，访问其栈中指数这一成员与当前输入的指数进行对比，如果发生指数相同的情况就对该节点的系数这一成员变量进行更新，输入的数据不用再进栈；如果输入的数据中指数成员变量与当前栈中所有节点的指数节点变量都不相同，则将输入的数据作为新的节点入栈，将加号作为一个单独的字符变量进行输入。在输入和处理调整完毕后，将结构体变量的指针再次调整为栈底的指针，通过指针的遍历将每一个节点内的数据进行输出，同时在输出时要注意一些特殊情况：一是第一项前面没有加号；二是系数为 0 的时候直接跳过系数为 1 时不输出系数；三是指数为 1 时不输出<sup>^</sup>以及指数，指数为 0 时仅仅输出该节点的系数，完成输出。

在时间复杂度方面，在输入数据的时候每一次输入都要对栈进行一次遍历，所以时间复杂度为  $O(n^2)$ 。

第二题：调度火车

源代码：

```
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 1000
char a[MAXSIZE];
char b[MAXSIZE];
typedef struct SNode *PtrtoNode;
typedef PtrtoNode Stack;
struct SNode{
    char *Data;
    int top;
    int maxsize;
};
int main()
{
    int N;
    scanf("%d",&N);
    char temp;
    int i,j,count,flag=1;
    Stack S1=(Stack)malloc(sizeof(struct SNode));
    S1->Data=(char*)malloc(N*sizeof(char));
```

```

S1->top=-1;
S1->maxsize=N;
for(i=0;i<2*N;i++)
{
    scanf("%c",&a[i]);//入栈序列输入
}
for(i=0;i<2*N;i++)
{
    scanf("%c",&b[i]);//出栈序列输入
}
j=1;//出栈序列栈尾指针
for(count=1,i=1;count<=N;count++,i+=2)//遍历入栈序列
{
    if(a[i]!=b[j]&& i<=2*N-1)
    {
        S1->Data[++(S1->top)]=a[i];//压栈
        //printf("%c-in\n",a[i]);
    }
    else if(a[i]==b[j]&& i<=2*N-1)
    {
        S1->Data[++(S1->top)]=a[i];
        //printf("%c-in\n",a[i]);
        while(S1->Data[S1->top]==b[j])
        {
            //printf("%c-out\n",S1->Data[S1->top]);
            S1->Data[S1->top]==' ';//出栈清空
            if(S1->top!=-1) (S1->top)--;//栈顶指针回溯
            j+=2;//出栈序列指针后移，准备判断下一阶段是否可以出栈
        }
    }
}
if(j==2*N+1) printf("1\n");
else {printf("0"); return 0;}
//和上述代码块一样 只不过增加输出语句
j=1;
S1->top=-1;//栈顶指针归位，准备遍历
for(count=1,i=1;count<=N;count++,i+=2)
{
    if(a[i]!=b[j]&& i<=2*N-1)
    {
        S1->Data[++(S1->top)]=a[i];
        printf("%c-in\n",a[i]);
    }
    else if(a[i]==b[j]&& i<=2*N-1)

```

```

    {
        S1->Data[++(S1->top)]=a[i];
        printf("%c-in\n",a[i]);
        while(S1->Data[S1->top]==b[j])
        {
            printf("%c-out\n",S1->Data[S1->top]);
            S1->Data[S1->top]=' ';
            if(S1->top!=-1) (S1->top)--;
            j+=2;
        }
    }
}
return 0;
}

```

程序运行结果:

```

4
A B C D
D C B A
1
A-in
B-in
C-in
D-in
D-out
C-out
B-out
A-out

Process returned 0 (0x0)   execution time : 60.815 s
Press any key to continue.

```

```

4
A B C D
D B C A
0

Process returned 0 (0x0)   execution time : 59.336 s
Press any key to continue.

```

第二题主要是实现栈的先入先出是否能够实现某一种输出顺序, 本题我采用以下方法解决: 首先准备两个数组将所给的火车入栈顺序和出栈顺序输入, 然后用一个变量指向



输出序列的首位，再对入栈序列进行一次遍历，每一次遍历的过程中都和输出序列当前指向的变量进行比较如果不相同，意味着现在不能出栈，将入栈的节点压栈，并且入栈序列向后移动一位，如果某时刻入栈序列与出栈序列相同，意味着可以出栈，此时进行一个循环，使得入栈序列的栈顶元素出栈并且输出序列再后移一位，反复进行这一过程直到所比较的数据不相同为止，然后再进行下一项数据的进栈。直到所输入的入栈序列全部遍历完毕，此时检验指向出栈序列的变量是否已达到最后一位，若达到说明能够使得所有节点按照输入的出栈序列输出，则输出 1 并进行输出，输出的代码和判断一样，只要加上输出语句即可。

时间复杂度：由于在输出时需要一直更新栈序列中输出满足条件的节点，所以时间复杂度为  $O(n^2)$ 。

## 实验报告（2）

实验名称：树的应用	实验地点：线上实验
所使用的工具软件及环境：Win7, Visual C++/Java	
<p><b>一、实验目的：</b></p> <ol style="list-style-type: none"><li>1、掌握二叉树的结构特征，以及各种存储结构的特点及使用范围。</li><li>2、掌握用指针类型描述、访问和处理二叉树的运算。</li><li>3、掌握树的应用算法。</li></ol>	
<p><b>二、实验内容描述：</b>（填写题目内容及输入输出要求）</p> <p>（1）设计算法计算二叉树最大的宽度（二叉树的最大宽度是指二叉树所有层中结点个数的最大值）。</p> <p>输入：创建的二叉树，先序排列，以#虚设结点。</p> <p>输出：一个正整数 N，代表着该二叉树的最大宽度。</p> <p>（2）编写程序实现平衡二叉树 AVL 树。</p> <p>输入：第一行为一个正整数 N 代表二叉树序列长度；第二行为 N 个数字，中间用空格分开，代表输入的 N 个二叉树顺序。</p> <p>输出：生成的平衡二叉树序列，先序排列，以#表示空结点。</p> <p>（3）已知一棵树的由根至叶子结点按层次输入的结点序列及每个结点的度（每层中自左至右输入），试写出构造此树的孩子-兄弟链表的算法。</p> <p>输入：一个正整数 N 结点数；然后输入 N 行，每行输入两个数字，中间用空格分开，代表节点及其对应的度。</p> <p>输出：若该树有 M 个叶结点，则输出 M 行，每行是一条从根到叶子结点的路径，然后按照先序遍历的方式输出每一行。</p>	

三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）

第一题：计算二叉树的宽度

源代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<deque>
#include<iostream>
int ceng[999999999];
typedef struct TNode *BinTree;
struct TNode{
    char Data;
    BinTree Left;
    BinTree Right;
}T1;
//求二叉树的宽度
int WidthOfBiTree(BinTree root)
{
    if(root == NULL)
        return 0;
    int maxWidth = 0;
    std::deque<BinTree> d;//利用双端队列实现层序遍历
    d.push_back(root);
    while(true)
    {
        int len = d.size();
        if(len == 0)
            break;
        while(len > 0)
        {
            BinTree temp = d.front();
            d.pop_front();
            len--;
            if(temp->Left)
                d.push_back(temp->Left);
            if(temp->Right)
                d.push_back(temp->Right);
        }
        maxWidth = maxWidth > d.size() ? maxWidth : d.size();
    }
    return maxWidth;
}
BinTree CreateBinTree()
```

```

{
    char ch;
    BinTree T=NULL,left=NULL,right=NULL;
    ch=getchar();
    if(ch=='#') return NULL;
    else
    {
        T=(BinTree)malloc(sizeof(struct TNode));
        T->Data=ch;
        left=CreateBinTree();
        if(left!=NULL) T->Left=left;
        else T->Left=NULL;
        right=CreateBinTree();
        if(right!=NULL) T->Right=right;
        else T->Right=NULL;
    }
    return T;
}
int main()
{
    int ans;
    BinTree T1;
    T1=CreateBinTree();
    ans=WidthOfBiTree(T1);
    printf("%d",ans);
    return 0;
}

```

程序运行结果：

```

C:\Users\Bo\Desktop\数据结构代码第二章\实验二第一题.exe
11111##1###111##1##11##1##11##1##
6
Process returned 0 (0x0) execution time : 1.471 s
Press any key to continue.

```

这道题主要运用了二叉树的数据结构来进行树的构建，用一个双端队列用来存储每一层的节点个数。在解决问题中，先用递归定义的结构体来作为树的结构将数据存入其中并返回头节点。然后从头节点开始遍历首先头结点构成第一层，将头结点录入双端队列，记录队列长度此时队列长度就是当前树的宽度。然后对于每一层的节点都进行如下递推操作：记录队列长度，弹出队首元素，遍历其左子树和右子树如果存在节点则加入队列之中直到原来记录的队列长度的队列中所有元素都被弹出，说明上一层已全部清空，而上一层的所有子节点均已入队，构成新的队列，新的队列的长度就是树对应下一层的宽度，不断更新宽度最大值直到队列全部为空，得到出其宽度最大值。

时间复杂度：对所给出的树的先序输出只进行一次遍历，所以时间复杂度为  $O(n)$ ;

## 第二题：AVL 树构造

源代码：

```
#include<stdio.h>
#include<stdlib.h>
typedef struct AVLNode * AVLTree;//注意 AVLTree 的类型是结构体指针，指向对象是结构体
struct AVLNode{
    int Data;
    AVLTree Left;
    AVLTree Right;
    int Height;
};
int max(int a,int b)
{
    return a>b?a:b;
}
int GETHeight(AVLTree T)//p117
{
    int hl,hr,maxh;
    if(T)
    {
        hl=GETHeight(T->Left);
        hr=GETHeight(T->Right);
        maxh=hl>hr?hl:hr;
        return (maxh+1);
    }
    else return 0;
}
AVLTree Singleleft(AVLTree A)//p141
{
    AVLTree B=A->Left;
    A->Left=B->Right;
    B->Right=A;
    A->Height=max(GETHeight(A->Left),GETHeight(A->Right))+1;
    B->Height=max(GETHeight(B->Left),A->Height)+1;
    return B;
}
AVLTree Singleright(AVLTree A)
{
    AVLTree B=A->Right;
    A->Right=B->Left;
    B->Left=A;
    A->Height=max(GETHeight(A->Left),GETHeight(A->Right))+1;
```

```

        B->Height=max(GETHeight(B->Right),A->Height)+1;
        return B;
    }
AVLTree Doublelefttright(AVLTree A)
{
    A->Left=Singleright(A->Left);
    return Singleleft(A);
}
AVLTree Doublerightleft(AVLTree A)
{
    A->Right=Singleleft(A->Right);
    return Singleright(A);
}
AVLTree INSERT(AVLTree T, int X)//p139
{
    if(!T)
    {
        T=(AVLTree)malloc(sizeof(struct AVLNode));
        T->Data=X;
        T->Height=1;
        T->Left=T->Right=NULL;
    }
    else if(X<T->Data)
    {
        T->Left=INSERT(T->Left,X);
        if(GETHeight(T->Left)-GETHeight(T->Right)==2)
        {
            if(X<T->Left->Data) T=Singleleft(T);
            else T=Doublelefttright(T);
        }
    }
    else if(X>T->Data)
    {
        T->Right=INSERT(T->Right,X);
        if(GETHeight(T->Left)-GETHeight(T->Right)==-2)
        {
            if(X>T->Right->Data) T=Singleright(T);
            else T=Doublerightleft(T);
        }
    }
    T->Height=max(GETHeight(T->Left),GETHeight(T->Right))+1;
    return T;
}
void preorder(AVLTree T)//p112

```

```

{
    if(T)
    {
        printf("%d ",T->Data);
        preorder(T->Left);
        preorder(T->Right);
    }
    else printf("# ");
}
int main()
{
    AVLTree T1=NULL;//注意 T1 的数据类型是结构体指针，采用了动态存储方式，所以必须
    初始化！
    int N,i,node;
    scanf("%d",&N);
    for(i=1;i<=N;i++)
    {
        scanf("%d",&node);
        T1=INSERT(T1,node);
    }
    preorder(T1);
    return 0;
}

```

程序运行结果：

```

C:\Users\Bo\Desktop\数据结构代码第二章\实验二第二题.exe
16
3 2 1 4 5 6 7 16 15 14 13 12 11 10 8 9
7 4 2 1 # # 3 # # 6 5 # # # 13 11 9 8 # # 10 # # 12 # # 15 14 # # 16 # #
Process returned 0 (0x0) execution time : 3.116 s
Press any key to continue.

```

本题采用了最经典的树的存储结构，通过结构体嵌套定义，来实现左子树右子树的存储。在本题建立 AVL 树的过程中，对于每一个节点都进行插入的操作。是其满足左子树小于根节点，右子树大于根节点，然后利用递归来计算树的高度，判断树是否平衡，不平衡则根据实际情况进行单旋或双旋。随后对生成的树进行先序遍历即可。

时间复杂度：对每个节点分开处理，所以时间复杂度为  $O(n)$ 。

第三题：孩子兄弟节点构造算法

源代码：

```

#include<iostream>
#include<queue>
using namespace std;
typedef struct Treenode{
    int data;
    struct Treenode *lchild, *rightbro;
}

```

```

}*Tree;
void printq(queue<int> q)
{
    while(!q.empty())
    {
        cout<<q.front()<<"-";
        q.pop();
    }
    cout<<endl;
}
std::queue<int> q;
void Search(Tree T)
{
    if(T[0].lchild!=NULL)
    {
        q.push(T[0].data);
        Search(T[0].lchild);
    }
    else if(T[0].lchild==NULL)
    {
        printq(q);
        q.pop();
        Search(T[0].rightbro);
    }
}
Tree Getroot(int data[],int degree[],int length)
{
    Tree p=new Treenode[length];
    for(int i=0;i<length;i++)
    {
        p[i].data=data[i];
        p[i].lchild=p[i].rightbro=NULL;
    }
    int d,nodeid=0;
    for(int i=0;i<length;i++)
    {
        d=degree[i];
        if(d)
        {
            nodeid++;
            p[i].lchild=&p[nodeid];
            for(int j=2;j<=d;j++)
            {

```

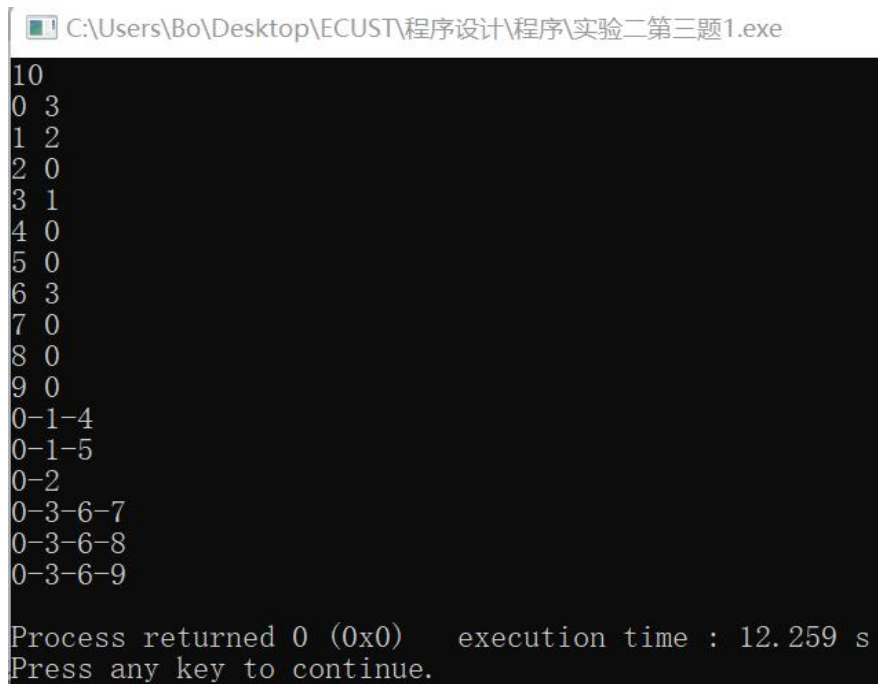


```

        nodeid++;
        p[nodeid-1].rightbro=&p[nodeid];
    }
}
}
return p;
}
int main()
{
    int a[1000],b[1000],len,i;
    scanf("%d",&len);
    for(i=1;i<=len;i++)
    {
        scanf("%d%d",&a[i],&b[i]);
    }
    Tree t=Getroot(a,b,len);
    Search(t);
    return 0;
}

```

程序运行结果：



```

C:\Users\Bo\Desktop\ECUST\程序设计\程序\实验二第三题1.exe
10
0 3
1 2
2 0
3 1
4 0
5 0
6 3
7 0
8 0
9 0
0-1-4
0-1-5
0-2
0-3-6-7
0-3-6-8
0-3-6-9

Process returned 0 (0x0)   execution time : 12.259 s
Press any key to continue.

```

本题用到的数据结构是一般的树的结构，与二叉树不同，一般的树结构的结构体成员变量是其第一个子树和自己的兄弟节点，在这道题解题过程中首先根据左子树是否存在来确定其是否为叶节点，然后从根节点出发，对于有左子树的节点，根据输入的右兄弟节点的个数进行入队操作，直到遇到节点没有左子树证明遇到了叶节点，将队列完整输出，弹出队首节点继续如上操作直到队列全部为空为止。

时间复杂度：本题对于每个等待判断的节点都是需要遍历的所以时间复杂度为  $O(n)$ 。

## 实验报告（3）

实验名称：图的应用	实验地点：线上实验														
所使用的工具软件及环境：Win7, Visual C++/Java															
<p>一、实验目的：</p> <ol style="list-style-type: none"><li>1、理解图的含义；</li><li>2、掌握用邻接矩阵和邻接表的方法描述图的存储结构；</li><li>3、理解并掌握深度优先遍历和广度优先遍历的存储结构；</li></ol>															
<p>二、实验内容描述：（填写题目内容及输入输出要求）</p> <p>1、列出连通集。</p> <p>给定一个有 <math>N</math> 个顶点和 <math>E</math> 条边的无向图，请用深度优先遍历（DFS）和广度优先遍历（BFS）分别列出其所有的连通集。假设顶点从 0 到 <math>N-1</math> 编号。进行搜索时，假设总是从编号最小的顶点出发，按编号递增的顺序访问邻接点。</p> <p>输入：输入第 1 行给出 2 个整数 <math>N(0 &lt; N \leq 10)</math> 和 <math>E</math>，分别是图的顶点数和边数。随后 <math>E</math> 行，每行给出一条边的两个端点。每行中的数字之间用一个空格分隔。</p> <p>输出：按照 “{v1v2...vk}” 的格式，每行输出一个连通集。先输出 DFS 的结果，再输出 BFS 的结果。</p> <p>测试用例：</p> <table><tbody><tr><td>Input: 8    6</td><td>output: {0 1 4 2 7}</td></tr><tr><td>0    7</td><td>{3 5}</td></tr><tr><td>0    1</td><td>{6}</td></tr><tr><td>2    0</td><td>{0 1 2 7 4}</td></tr><tr><td>4    1</td><td>{3 5}</td></tr><tr><td>2    4</td><td>{6}</td></tr><tr><td>3    5</td><td></td></tr></tbody></table>		Input: 8    6	output: {0 1 4 2 7}	0    7	{3 5}	0    1	{6}	2    0	{0 1 2 7 4}	4    1	{3 5}	2    4	{6}	3    5	
Input: 8    6	output: {0 1 4 2 7}														
0    7	{3 5}														
0    1	{6}														
2    0	{0 1 2 7 4}														
4    1	{3 5}														
2    4	{6}														
3    5															
<p>三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）</p> <p>源代码：</p> <pre>#include&lt;stdio.h&gt; #include&lt;stdlib.h&gt; #include&lt;stdbool.h&gt; #include&lt;iostream&gt; #include&lt;queue&gt; typedef struct GNode *MGraph; struct GNode{     int Nv;     int Ne;     int G[100][100]; }; typedef struct ENode *Edge; struct ENode{</pre>															

```

    int V1,V2;
};
bool Visit[100]={false};
int A;
MGraph CreateGraph(int Vnum)
{
    int V,W;
    MGraph Graph;
    Graph=(MGraph)malloc(sizeof(struct GNode));
    Graph->Nv=Vnum;
    Graph->Ne=0;
    for(V=0;V<Graph->Nv;V++)
    {
        for(W=0;W<Graph->Nv;W++)
        {
            Graph->G[V][W]=0;
        }
    }
    return Graph;
}
void Insertedge(MGraph Graph,Edge E)
{
    Graph->G[E->V1][E->V2]=1;
    Graph->G[E->V2][E->V1]=1;
}
MGraph BuildGraph()
{
    MGraph Graph;
    Edge E;
    int Vv;
    int NV,i;
    scanf("%d",&NV);
    A=NV;
    Graph=CreateGraph(NV);
    scanf("%d",&(Graph->Ne));
    if(Graph->Ne!=0)
    {
        E=(Edge)malloc(sizeof(struct ENode));
        for(i=0;i<Graph->Ne;i++)
        {
            scanf("%d %d",&E->V1,&E->V2);
            Insertedge(Graph,E);
        }
    }
}

```

```

        return Graph;
    }
    void DFS(MGraph Graph, int V)
    {
        int W;
        if(Visit[V]==false)
        {
            printf("%d ",V);
            Visit[V]=true;
        }
        for(W=0;W<Graph->Nv;W++)
        {
            if((Visit[W]==false)&&(Graph->G[V][W]==1)) DFS(Graph,W);
        }
    }
    void BFS(MGraph Graph,int V)
    {
        int W;
        std::queue<int> q1;
        printf("%d ",V);
        Visit[V]=true;
        q1.push(V);
        while(!q1.empty())
        {
            for(W=0;W<Graph->Nv;W++)
            {
                if((Visit[W]==false)&&(Graph->G[q1.front()][W]==1))
                {
                    printf("%d ",W);
                    Visit[W]=true;
                    q1.push(W);
                }
            }
            q1.pop();
        }
    }
    int main()
    {
        MGraph G1;
        int i;
        G1=BuildGraph();
        for(i=0;i<A;i++)
        {
            if(Visit[i]==true) continue;

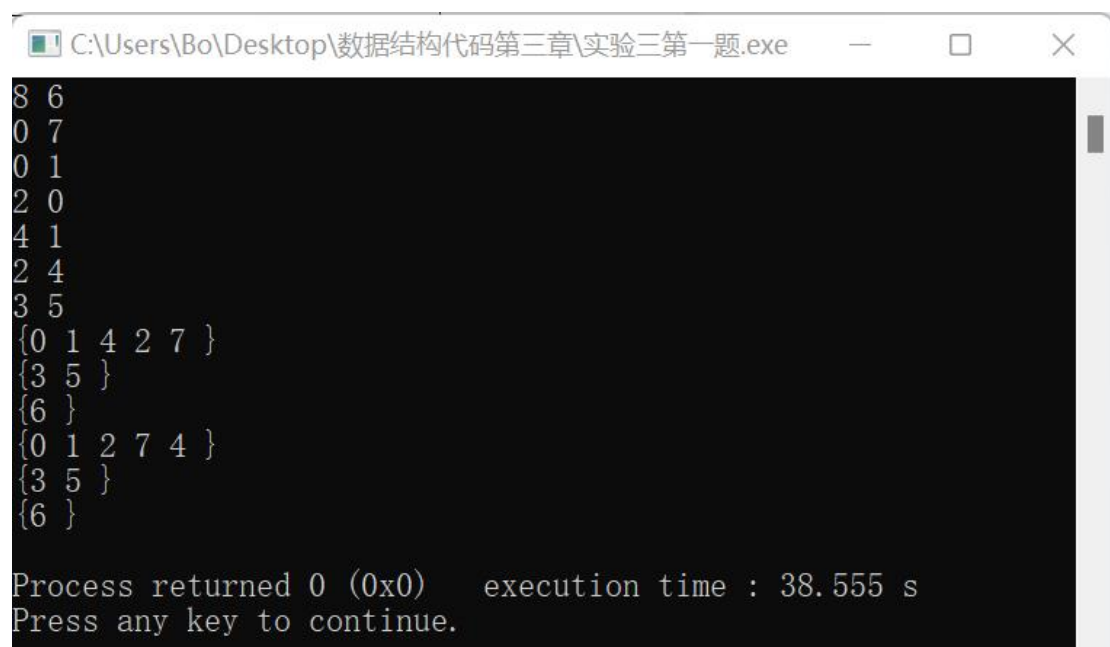
```

```

        else
        {
            printf("{");
            DFS(G1,i);
            printf("}\n");
        }
    }
    for(i=0;i<A;i++)
    {
        Visit[i]=false;
    }
    for(i=0;i<A;i++)
    {
        if(Visit[i]==true) continue;
        else
        {
            printf("{");
            BFS(G1,i);
            printf("}\n");
        }
    }
    return 0;
}

```

程序运行结果：



```

8 6
0 7
0 1
2 0
4 1
2 4
3 5
{0 1 4 2 7 }
{3 5 }
{6 }
{0 1 2 7 4 }
{3 5 }
{6 }

Process returned 0 (0x0)    execution time : 38.555 s
Press any key to continue.

```

本题主要采用图的相关数据结构，对于图的存储方式采用邻接矩阵的方式进行存储对于插入的边只要改变邻接矩阵中相应坐标内的数据即可，深度优先搜采用递推的方式，对于选定的起始点进行每一个点的一一循环遍历判断其和起始点是否连接，如果连接就进行标记，然后再对这一个点进行深度优先遍历，递推直到所有点都被标记。而广度优先搜索

则使用队列的数据结构，将每一次遍历节点的所有邻接的边入队并标记，再对队列进行遍历，从队头开始进行广度优先遍历，注意遍历完后要弹出被检测节点并标记，直到队列为空为止。

时间复杂度：对于深度优先和广度优先，都需要对每个节点进行一次所有边的遍历，所以我设计的算法时间复杂度为  $O(n^2)$ 。

2022 年 6 月 15 日