#专业课代码稿

顺序表与链表的定义

```
//顺序表定义
typedef struct
{
   int data[maxSize];
   int length;
}Sqlist;
//在考试时直接使用下面定义即可
int A[maxSize];
int n;

//单链表定义
typedef struct LNode
{
   int data;
   struct LNode *next;
}LNode;
```

头插法与尾插法关键代码（一般重要，但必须要会）

```
//头插法
void headinsert(LNode *&L, int a[], int n)
{
  LNode *p;//用于指向插入的节点
  int i;
  L=(LNode*)malloc(sizeof(LNode));//申请L的头节点空间
  L->next=NULL;//L初始化为空
  for(i=0;i<n;++i)
  {
      p=(LNode*)malloc(sizeof(LNode));
      p->data=a[i];
      //以下为核心代码
```

```
        p->next=L->next;//p的指针指向那个L的开始节点
        L->next=p;//L指针指向p
    }
}

//尾插法
void rearinsert(LNode *&L, int a[], int n)
{
    LNode *p, *q; //p指向要插入的节点，q指向L的最后一个节点
    int i;
    L=(LNode*)malloc(sizeof(LNode));
    L->next=NULL;//这两句和头插法作用相同
    q=L;
    for(i=0;i<n;++i)
    {
        p=(LNode*)malloc(sizeof(LNode));
        p->data=a[i];
        //以下为核心代码
        q->next=p;//q的next指针指向新节点
        q=q->next;//q指向新的最后一个节点
    }

}
//特别注明：头插法与尾插法最后得到的序列正好完全相反，可以用以某些题目：如将递增链表修改成递减链表
```

栈与队列的定义

```
//顺序栈定义
typedef struct
{
    int data[maxSize];
    int top;//栈顶指针
}SqStack;
//假设元素为int型，可以直接使用下面方法进行定义并初始化
int stack[maxStack];
```

```
int top=-1;
//链栈定义——其实就是单链表，这里就不写了

//顺序队列定义
typedef struct
{
  int data[maxSize];
  int front;
  int rear;
}SqQueue;

//链队定义
//1.队节点定义
typedef struct QNode
{
  int data;//数据域
  struct QNode *next;//指针域
}QNode;
//2.类型定义
typedef struct
{
  QNode *front;//队头指针
  QNode *rear;//队尾指针
}LiQueue;

//基本上不会单独出题，应该会结合其他的考查
```

矩阵与三元组

```
//给定一个稀疏矩阵A（float型），其尺寸为m*n，1.建立其对应三元组存储，2.并通过三元组打印输出矩阵A。

/*建立三元组*/
void createtrimat(float A[][maxSize],int m,int n,float B[][3])
{
  int k=1;
```

```
  for(int i=0;i<m;++i)
      for(int j=0;j<n;++j)
          if(A[i][j]!=0)//如果元素不为零，存入三元组
          {
              B[k][0]=A[i][j];//第一位存元素值
              B[k][1]=i;//第二位存行号
              B[k][2]=j;//第三位存列号
              ++k;
          }
      B[0][0]=k-1;//三元组第一行第一个元素记录：非零元素个数
      B[0][1]=m;//第一行第二个元素记录：总行数
      B[0][2]=n;//第一行第三个元素记录：总列数
}

/*通过三元组输出矩阵A*/
void outputmatrix(float B[][3])
{
  int k=1;
  for(int i=0;i<B[0][1];++i)
      for(int j=0;j<B[0][2];++j)
          if(i==(int)B[k][1] && j==(int)B[k][2])
          {
              cout<<B[k][0];
              ++k;
          }else{
              cout<<“0 “;
          }
      cout<<endl;
}

//不是太重要，考408必须要会
```

树与二叉树存储结构

```
//二叉链表存储结构，几乎考树的代码题，都得用到这个，必须要会
typedef struct BTNode
```

```
{
  char data;
  struct BTNode *lchild;
  struct BTNode *rchild;
}BTNode;
```

```
/*树的遍历算法*/
//先序遍历（递归版）
void preorder(BTNode *p)
{
  if(p!=NULL)
  {
      Visit(p);
      preorder(p->lchild);
      preorder(p->rchild);
  }
}
//中序遍历（递归版）
void midorder(BTNode *p)
{
  if(p!=NULL)
  {
      midorder(p->lchild);
      Visit(p);
      midorder(p->rchild);
  }
}
//后序遍历（递归版）
void postorder(BTNode *p)
{
  if(p!=NULL)
  {
      postorder(p->lchild);
      postorder(p->rchild);
      Visit(p);
  }
```

```
}
//先序遍历（非递归版——借助栈）
void preorder1(BTNode *p)
{
  if(p!=NULL)
  {
      BTNode *stack[maxSize];
      int top=-1;
      BTNode *q;//指向要出栈元素
      stack[++top]=p;
      while(top!=-1)
      {
          q=stack[top--];
          Visit(q);
          //注意下面的代码，先入栈右子树，再入栈左子树，切记别写错
          if(q->rchild!=NULL)
              stack[++top]=q->rchild;
          if(q->lchild!=NULL)
              stack[++top]=q->lchild;
      }
  }
}
//中序遍历的非递归遍历代码是先入栈左子树，再入栈右子树
//注意判断当栈空时，遍历不一定结束，所以要while判断时要加上
while(top!=-1 ||p!=NULL)
{
  while(q!=NULL)
  {
      Stack[++top]=q;
      q=q->lchild;//左子树存在则左子树入栈
  }
  if(top!=-1)
  {
      q=stack[top--];
      Visit(q);
      q=q->rchild;
  }
```

```
}
//后序遍历非递归方式，考的最多，由于逆后序遍历序列和先序遍历序列只有左右子树访问次序不同，所以，借助两个即可实现后序遍历的非递归版
//第一个栈要求先入栈左子树，再入栈右子树，代码就不写了，按照先序遍历来即可
```

典型题目有：求二叉树深度，二叉树查找，输出某个值，都可以使用递归求得

```
//写一个算法求一棵二叉树深度，存储方式为二叉链表
int getDepth(BTNode *p)
{
  int LD,RD;//用来接收左右子树深度的
  if(p==NULL)//如果为空，输出深度为0
      return 0;
  else{
      LD=getDepth(p->lchild);
      RD=getDepth(p->rchild);
      return (LD>RD?LD:RD)+1;//递归求法，最后返回左右子树深度的最大值+1
  }
}
//在二叉树p中查找data域值为key的节点是否存在，存储方式为二叉链表，如果存在用q指向它
void searchkey(BTNode *p,BTNode *&q,int key)
{
  if(p!=NULL)
  {
      if(p->data==key)
          q=p;
      else{
          searchkey(p->lchild,q,key);
          if(q==NULL)//剪枝，如果左子树没找到，才去右子树，增加效率
              searchkey(p->rchild,q,key);
      }
  }
}
```

层次遍历

```
//层次遍历需要借助一个循环队列，先将二叉树的头节点入队，然后出队访问，如果有左子树则左子树根节点入队，如果有右子树，则右子树根节点入队，然后出队，并进行访问，循环，直至队空为止
void levelorder(BTNode *p)
{
  BTNode *que[maxSize];
  int front=rear=0;//定义一个队列
  BTNode *q;//用于指向出队元素
  if(p!=NULL)
  {
      rear=(rear+1)%maxSize;
      que[rear]=p;//根节点入队
      while(front!=rear)//队不空循环
      {
          front=(front+1)%maxSize;
          q=que[front];
          Visit(q);//如果队不为空，开始出队访问
          if(q->lchild!=NULL)//如果左子树不空，则入队
          {
              rear=(rear+1)%maxSize;
              que[rear]=q->lchild;
          }
          if(q->rchild!=NUll)//如果右子树不空，则入队
          {
              rear=(rear+1)%maxSize;
              que[rear]=q->rchild;
          }
      }
  }
}
```

层次遍历最经典的考题，求二叉树宽度

```
//设计一个算法求二叉树的宽度，用二叉链表存储
//先定义一个顺序非循环队列，可以存储节点节点所在的层次号
typedef struct
{
```

```
  BTNode *b;//节点指针
  int lno;//节点所在层次号
}ST;

int maxwidth(BTNode *p)
{
  ST que[maxSize];
  int front=rear=0;//定义一个顺序非循环队列
  int Lno=0,max=0,n=0;
  BTNode *q;
  //下面其实就是层次遍历，但将循环队列改成了非循环队列而已
  if(p!=NULL)
  {
      ++rear;
      que[rear].b=p;//根节点入队
      que[rear].lno=1;//此时层号为1
      while(front!=rear)
      {
          ++front;
          q=que[front].b;//用q指向出队节点
          Lno=que[front].lno;//Lno用来存当前节点的层数
          if(q->lchild!=NULL)
          {
              ++rear;
              que[rear].b=q->lchild;
              que[rear].lno=Lno+1;//关键句，根据当前节点的层号推其孩子节点层号
          }
          if(q->rchild!=NULL)
          {
              ++rear;
              que[rear].b=q->rchild;
              que[rear].lno=Lno+1;//关键句，根据当前节点的层号推其孩子节点层号
          }
      }
//下面的代码就是来找节点层中的节点数
      for(int i=1;i<=Lno;++i)//层号
      {
```

```
            n=0;
            for(int j=1,j<=rear;++j)//找队列里面与层号相同的节点，每找到一个，n+1
                if(que[j].lno==i)
                    ++n;
            if(max<n)
                max=n;
        }
        return max;
    }
    else
        return 0;//如果是空树直接返回0
}
```

线索二叉树定义

```
typedef struct TBTNode
{
    char data;
    int ltag,rtag;
    struct TBTNode *lchild;
    struct TBTNode *rchild;
}TBTNode;
```

图（非重点，代码题很久没考图了，不保证不考）

```
//图的邻接矩阵定义
typedef struct
{
    int no;//顶点编号
    char info;
}VertexType;
typedef struct
{
    int edges[maxSize][maxSize];
```

```
  int n,e;//顶点个数和边个数
  VertexType vex[maxSize];//存放节点信息
}MGraph;
//上面的定义如果没记住，也要记住里面的元素，如n，e等含义

//图的邻接表，重点中的重点，考的很多
typedef struct ArcNode
{
  int adjvex;//该边所指向的位置
  struct ArcNode *nextarc;//指向下一条边的指针
  int info;
}ArcNode;//边定义
typedef struct
{
  char data;
  ArcNode *firstarc;
}VNode;//顶点定义
typedef struct
{
  int n,e;
  VNode adjlist[maxSize];
}AGraph;//邻接表类型定义

//408邻接多重表（无向图）与十字链表（有向图）几乎不考
```

图的遍历操作

```
//深度优先搜索遍历,使用邻接表为存储结构(递归)
int visited[maxSize];//访问标记
void DFS(AGraph *G,int v)
{
  ArcNode *p;
  visited[v]=1;
  Visit(v);
  p=G->adjlist[v].firstarc;//指向第一条边
  while(p!=NULL)
  {
```

```
        if(visited[p->adjvex]==0)//若顶点未被访问，则递归
            DFS(G,p-adjvex);
        p=p->nextarc;
    }
}
//广度优先搜索遍历，使用邻接表为存储结构,需要借助队列（非递归）
int visited[maxSize];
void BFS(AGraph *G,int v)
{
    ArcNode *p;
    int que[maxSize];
    int front=rear=0;
    int w;
    visited[v]=1;
    Visit(v);
    rear=(rear+1)%maxSize;
    que[rear]=v;//入队
    while(front!=rear)
    {
        front=(front+1)%maxSize;
        w=que[front];//出队
        p=G->adjlist[w].firstarc;
        while(p!=NULL)
        {
            if(visited[p->adjvex]==0)//当前节点未被访问，则入队
            {
                visited[p->adjvex]=1;
                Visit(p->adjvex);
                rear=(rear+1)%maxSize;
                que[rear]=p->adjvex;
            }
            p=p->nextarc;//p指向w下条边
        }
    }
}
```

图的经典例题

```
//求距离顶点最远的顶点
//广度优先搜索遍历最后一个顶点一定是距离给定顶点最远的一个顶点，所以在广度优先搜索遍历代码后返回最后一个顶点即可。

//判断无向图是否为一棵树
//一个无向图是一棵树的条件是有n-1条边的连通图
int visited[maxSize];
void DFS2(AGraph *G,int v,int &vn,int &en)
{
  //下面的代码基本上就是深度优先遍历，只是多了顶点和边计数
  ArcNode *p;
  visited[v]=1;
  ++vn;//访问的顶点数自增1
  p->G.adjlist[v].firstarc;
  while(p!=NULL)
  {
      ++en;//边数自增1
      if(visited[p->adjvex]==0)
          DFS2(G,p->adjvex,vn,en);
      p=p->nextarc;
  }
}
int judgetree(AGraph *G)
{
  int vn=0,en=0,i;
  for(i=0;i<G->n;++i)
      visited[i]=0//将所有标记初始化为0，表示未被访问
  DFS2(G,1,vn,en);
  //如果遍历过程中访问的顶点数和途中的顶点数相等，且边数等于顶点数-1，则为树
  if(vn==G->n && (G->n-1)==en/2)//无向图边为实际上en计数的一半
      return 1;
  else
      return 0;
}
//判断ij两点是否有路径
//关键代码如下
```

```
for(int i=0;i<G->n;++i)
  visited[i]=0//初始化为0，表示未被访问
DFS(G,i);
if(visited[j]==1)
  return 1;
else
  return 0;
```

最短路径问题，代码推荐只看Floyd，比较简单易懂

```
//算法思想：使用邻接矩阵的存储结构，先初始化A与Path数组，之后循环判断A[i][j]>A[i][k]
+A[k][j]，如果成立则令A[i][j]=A[i][k]+A[k][j]
void Floyd(MGraph g, int Path[][maxSize])//采用邻接矩阵存储
{
  int i,j,k;
  int A[maxSize][maxSize];
  //初始化A与Path数组
  for(i=0;i<g.n;++i)
      for(j=0;j<g.n;++j)
      {
          A[i][j]=g.edges[i][j];
          Path[i][j]=-1;
      }
  //关键代码
  for(k=0;k<g.n;++k)
      for(i=0;i<g.n;++i)
          for(j=0;j<g.n;++j)
              if(A[i][j]>A[i][k]+A[k][j])
              {
                  A[i][j]=A[i][k]+A[k][j];
                  Path[i][j]=k;
              }
}
```

拓扑排序

```
//算法思想：遍历扫描全部节点，将入度为0的节点入栈，之后栈不空出栈，将输出计数n进行加一，和出栈节点链接的nextarc节点，将入度的count--，如果count==0时入栈，while一直循环直到栈空，之后判断n是否等于G->n，如果相等return 1，否则return 0.

//改造VNode定义，加上count
typedef struct
{
  char data;
  ArcNode *firstarc;
  int count;//新加，用以计入度为多少
}VNode;
int TopSort(AGraph *G)
{
  int stack[maxSize];
  int top=-1;//定义一个栈
  ArcNode *p;
  int i,n;//n为访问计数
  for(i=0;i<G->n;++i)//将入度为0的节点入栈
      if(G->adjlist[i].count==0)
          stack[++top]==i;
  //下面是关键代码
  while(top!=-1)
  {
      i=stack[top--];//栈不空，出栈
      ++n;//将访问计数n++
      //output number of i
      p=G->adjlist[i].firstarc;
      while(p!=NULL)
      {
          --(G->adjlist[p->adjvex].count);
          if(G->adjlist[p->adjvex].count)
              stack[++top]=p->adjvex;
          p=p->nextarc;
      }
  }
  if(n==G->n)
      return 1;
```

```
  else
      return 0;
}
```

排序算法

```
//代码主要是掌握插入排序，快速排序（分治思想，考的特别多），选择排序，还得了解一下动态规划思想
void InsertSort(int R[],int n)
{
  int i,j,temp;
  for(i=1;i<n;++i)
  {
      temp=R[i];//每次循环第一步先将待排关键字暂存temp中
      j=i-1;
      while(j>=0&&temp<R[j];
      {
          R[j+1]=R[j];
          --j;
      }
      R[j+1]=temp;
  }
}


//下面是重点，快速排序，不敲了，打印后自己写



```

总结：主要是线性表，链表，二叉链表，还有图（一般重要），部分排序算法（快排等很重要）