选择填空

1、数据元素：数据的最小单位；
数据项：数据最小单位；

2、逻辑结构
线性结构：有且只有一个开始和一个终端结点，并且所有节点都最多只有一个直接前驱和一个后继
非线性：一个结点可能有多个直接前驱和直接后继

3、存储结构
顺序存储：Loc（元素 i）=Lo+（i-1）*m
链式存储

4、数据的运算：插入、删除、修改、查找、排序

5、逻辑结构唯一，存储结构不唯一，运算的实现依赖于存储结构

6、抽象数据类型 ADT=(D ,S,P)数据对象 D 上的关系集 D 上的操作集

7、抽象数据类型的表示与实现

（1） 预定义常量及类型

```
#define OK 1
#define ERROR 0
#define OVERFLOW -2

Typedef int Status;
```

Status 是函数返回值类型，其值是函数结果状态代码。

（2） 数据元素被约定为 ElemType 类型，用户选哟更急具体情况，自行定义该数据类型

（3） 算法描述为一下的函数形式：
函数类型 函数名（函数参数表）
{ 语句序列；}

（4） 内部的动态分配与释放
分配空间：指针变量=new 数据类型
释放空间：delete 指针变量

（5） 函数结束语句 return
循环结束语句 break
异常结束语句 exit（异常代码）

8、算法特性
输入（有 0 个或多个输入）输出（有一个或多个输出（处理结果））确定性、有穷性、有效性

9、算法的时间量度：T（n）=O（f(n)）
N 越大，执行时间越长
时间复杂度由嵌套最深语句的频度决定，取决于问题规模和初始状态

10、空间复杂度：S（n）=O(f(n))

11、线性结构反应结点间的逻辑关系是一对一的

12、n=0 时为空表

13、同一线性表中的元素必定具有相同特性
元素间关系是线性

14、线性表的操作：初始化、取值、查找、插入、删除

15、线性表插入在第 i 个结点之前，移动 n-i+1 次

各种位置插入（共 n+1 种可能）的平均移动次数：n/2

16、删除第 i 个结点，移动 n-i 次

各种位置删除（共 n 种可能）的平均移动次数：（n-1）/2

17、线性表查找、插入、删除算法的平均时间复杂度为：O（n），

空间复杂度 S (n)=O(1) 没有占用辅助空间

18、顺序表的特点

逻辑结构与存储结构一致

访问每个元素所花时间相等

随机存取法

19、顺序表的优缺点

优点：存储密度大，可以随机存取表中任一元素

缺点：在插入、删除某一元素时，需要移动大量元素；浪费存储空间；属于静态存储形式，数据元素的格数不能自由扩充

20、链式存储结构特点

节点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻

访问时只能通过头指针进入链表，并通过每个结点的指针域向后扫描其余结点

顺序存取法

21、结点=数据域+指针域（存储直接后继结点的存储位置）

22、头指针（数据域内只放空表表直和表长等信息）->头结点->首元结点

头节点不计入链表长度值

23、有头结点时，当头节点的指针域为空时表示空表

24、设置头结点的好处：便于首元结点的处理、便于空表和非空表的统一处理

25、链表的优缺点

优点：数据元素的个数可以自由扩充，插入、删除等操作不必移动数据，只需修改连接指针，修改效率较高

缺点：存储密度小，存储效率不高

26、链表的每个结点中都恰好包含一个指针（错）

27、线性表的每个结点只能时一个简单类型，而链表的每个结点可以是一个复杂类型（错）

28、单链表的存储密度小于 1

29、已知两个长度分别为 m 和 n 的升序链表，若将他们合并为一个 长度为 m+n 的降序链表，则最坏情况下的时间复杂的为 O（max(m,n)

30、将长度为 n 的单链表链接在长度为 m 的单链表之后的算法的 时间复杂度为 O（m）

31、线性表的（查找 ）运算，不改变数据元素之间的结构关系

32、单链表由表头唯一确定，可以用头指针的名字来命名

33、链表的时间效率

查找 O（n）

插入和删除 O（1）

34、循环链表为空表：L->next=L

35、从循环链表中的任何一个结点的位置都可 以找到其他所有结点，而单链表做不到

36、循环条件：p!=NULL 或 p->next!=NUL 单链表； p!=L 或 p->next!=L 循环链表

37、对循环链表，有时不给出头指针，而给出尾指针 可以更方便的找到第一个和最后一个结点

38、终端结点：rear 开始结点：rear->next->next

39、线性表的合并

在 La 中查找该元素

如果找不到，则将其插入 La 的最后

40、有序顺序表合并 S(n)=O(n)

41、有序链表合并 S(n)=O(1)

42、栈

只能在栈顶进行插入和删除运算的线性表

逻辑关系仍为一对一关系

只能在栈顶运算，后进先出，先进后出

43、队列

只能在队尾进行插入，在队头进行删除运算的线性表

逻辑结构仍为一对一

先进先出

44、顺序栈一定要预设栈顶指针 top

45、base==top 是空栈标志

46、链栈的表示：运算是受限的单链表，只能在链表头部进行操作，故没有必要附加头结点。栈顶指针就是链表的头指针

47、入队列 EnQueue (&Q,e)

出队列 DeQueue (&Q,&e)

48、空队标志：front==rear

入队：base[rear++]=x;

出队：x=base[front++];

49、front=0 rear=M 时 再入队—-真溢出

Front>0 rear=M 时 再入队—-假溢出

50、循环队列

实现：利用“模”运算

入队： base[rear]=x; rear=(rear+1)%M;

出队： x=base[front]; front=(front+1)%M;

队满：(rear+1)%M==front

51、子串的位置：Index(S,T,pos)

52、串链式存储表示

优点：操作方便；缺点：存储密度较低（可将多个字符存放在一个节点中，以克服其缺点）

存储密度=串值所占的存储位/实际分配的存储位

53、串值必须用一堆单引号括起来，但单引号本身不属于串

54、串的模式匹配算法

算法目的：确定主串中所含子串第一次出现的位置（定位）

算法种类：BF 算法、KMP 算法（速度快）

55、BF 算法时间复杂度：若 n 为主串长度，m 为字串长度，最坏情况是
主串前面 n-m 个位置都匹配到子串的最后一位，即这 n-m 位各比较了 m 次，最后 m 位也各比较了 1 次。
总次数为（n-m）*m+m=（n-m+1）*m
若 m<<n,则算法复杂度 O（n*m）

56、二维数组 a[n][m]的行序优先表示
设数组开始存放位置 LOC（0，0）=a，LOC（j,k）=a+j*m+k

57、按页、行、列存放，页优先的顺序存储

58、三维数组 LOC（i1, i2, i3 ） = a + i1* m2 * m3 + i2* m3 + i3

59、对称矩阵
特点：在 n*n 的矩阵 a 中，满足 aij=aji (1≤ i,j≤n)
存储方法：只存储下(或者上)三角(包括主对角线)的数据元 素。共占用 n(n+1)/2 个元素空间

60、三角矩阵
特点：对角线以下(或者以上)的数据元素(不包括对角线)全部 为常数 c 或 0
存储方法：重复元素 c 共享一个元素存储空间，共占用 n(n+1)/2+1 个元素空间: sa[1.. n(n+1)/2+1]

61、对角矩阵（带状矩阵）
特点：在 n*n 的方阵中，非零元素集中在主对角线及其两侧 共 L(奇数)条对角线的带状区域内 — L 对角矩阵
存储方式：以对角线的顺序存储
只存储带状区内的元素：除首行和末行，按每行 L 个元素，共(n-2)L+(L+1)个元素。

62、稀疏矩阵
特点：大多数元素为零
常用存储方法：只记录每一非零元素(i,j,aij) 节省空间，但丧失随机存取功能

63、广义表（列表）：n （ 0 ）个表元素组成的有限序列， 记作 LS = (a0,a1, a2,…, an-1)
LS 是表名，ai 是表元素，它可以是表 (称为子表)，可以是 数据元素(称为原子)。n 为表的长度。n=0 的广义表为空表。

64、求广义表表头：非空广义表的第一个元素，可以是一个单元素，也可以是一个子表
求广义表表尾：非空广义表除去表头元素以外其他元素所构成的表。表尾一定是一个表

65、双亲：即上层的那个结点(直接前驱)
孩子：即下层结点的子树的根(直接后继)
结点的度：结点拥有的子树数
结点的层次：从根到该结点的层数（根结点算第一层）
树的度：所有结点度中的最大值
树的深度：指所有结点中最大的层数
叶子结点（终端结点）：度为 0 的结点

66、二叉树基本特点：结点的度小于等于 2，有序树

67、二叉树的性质：

①在二叉树的第 i 层上至多有 2^{i-1} 个结点。第 i 层上至少有 1 个结点

②深度为 k 的二叉树至多有 2^k-1 个结点，至少有 k 个结点

③对于任何一棵二叉树，如果叶子（终端结点）数为 n0，度为 2 的结点数有 n2 个，则叶子数 n0 必定为 n2＋1 （即 n0=n2+1）

④具有 n 个结点的完全二叉树的深度必为[$\log_2 n$]＋1

⑤对完全二叉树，若从上至下、从左至右编号，则编号为 i 的结点，其左孩子编号必为 2i，其右孩子编号必为 2i＋1；其双亲的编号必为 i/2。

68、二叉树的顺序存储特点

结点间关系蕴含在其存储位置中

浪费空间，适于存满二叉树和完全二叉树

69、在 n 个结点的二叉链表中，有 n+1 个空指针域

70、遍历算法分析

时间效率：O（n）

空间效率：O（n）

71、若二叉树中各结点的值均不相同，则：

由二叉树的前序序列和中序序列，或由其后序序列和中序序列均能唯一地确定一棵二叉树，

但由前序序列和后序序列却不一定能唯一地确定一棵二叉树。

72、线索化二叉树

若结点有左子树，则 lchild 指向其左孩子； 否则， lchild 指向其直接前驱(即线索)；

若结点有右子树，则 rchild 指向其右孩子； 否则， rchild 指向其直接后继(即线索) 。

Lchild LTag data RTag rchild

LTag :若 LTag=0, lchild 域指向左孩子；

若 LTag=1, lchild 域指向其前驱。

RTag :若 RTag=0, rchild 域指向右孩子；

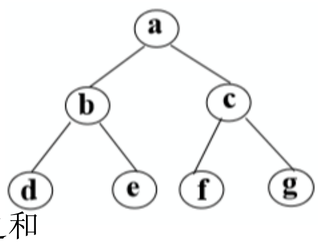
若 RTag=1, rchild 域指向其后继。

73、哈夫曼树应用实例——哈夫曼编码

出现次数较多的字符采用尽可能短的编码

关键：要设计长度不等的编码，则必须使任一字符的编码都不是另一个字符的编码的前缀－前缀编码

74、哈夫曼树的构造



路径：由一结点到另一结点间的分支所构成

路径长度：路径上的分支数目（a->e 的路径长度=2）

带权路径长度：结点到根的路径长度与结点上权的乘积

树的带权路径长度：书中所有叶子结点的带权路径长度之和

哈夫曼树：带权路径长度最小的树

基本思想：使权大的结点靠近根

操作要点：对权值的合并、删除与替换，总是合并当前值最小的两个

75、哈夫曼编码的构造

基本思想：概率大的字符用短码，小的用长码，构造哈夫曼树

76、一棵有 n 个叶子结点的哈夫曼树有 2n-1 个结点
77、哈夫曼编码的几点结论
哈夫曼编码是不等长编码。
哈夫曼编码是前缀编码，即任一字符的编码都不是另一字符编码的前缀。
哈夫曼编码树中没有度为 1 的结点。若叶子结点的个数为 n，则哈夫曼编码树的结点总数为 2n-1。
发送过程：根据由哈夫曼树得到的编码表送出字符数据
接收过程：按左 0、右 1 的规定，从根结点走到一个叶结点，完成一个字符的译码。反复此过程，直到接收数据结束。

简答题

1、线性表插入步骤
判断插入位置 i 是否合适
判断书虚表的存储空间是否已满
将第 n 至第 i 位的元素一次向后移动一个位置，空出第 i 个位置
将要插入的新元素 e 放入第 i 个位置
表长加 1，插入成功返回 OK
2、线性表删除步骤
判断删除位置 i 是否合法（合法值为 1≤i≤n）
将欲删除的元素保留在 e 中
将第 i+1 至第 n 位的元素一次向前移动一个位置
表长减 1，删除成功返回 OK
3、单链表插入步骤
找到 ai-1 存储位置 p
生成一个新结点*s
将新结点 *s 的数据域置为 x
新结点*s 的指针域指向 结点 ai
令结点*p 的指针域指向 新结点*s
4、单链表删除步骤
找到 ai-1 存储位置 p
保存要删除的结点的值
令 p->next 指向 ai 的直接后继结点
释放结点 ai 的空间
5、单链表的建立（前插法）
生成新结点
将读入数据存放到新结点的数据域中
将该新结点插入到链表的前端
6、单链表的建立（尾插法）
从一个空表 L 开始，将新结点逐个插入到链表的尾部，尾指 针 r 指向链表的尾结点。 初始时，r 同 L 均指向头结点。每读入一个数据元素则申请一 个新结点，将新结点插入到尾结点后，r 指向新结点

7、

存储结构 / 比较项目		顺序表	链表
空间	存储空间	预先分配，会导致空间闲置或溢出现象	动态分配，不会出现存储空间闲置或溢出现象
	存储密度	不用为表示结点间的逻辑关系而增加额外的存储开销，存储密度等于1	需要借助指针来体现元素间的逻辑关系，存储密度小于1
时间	存取元素	随机存取，按位置访问元素的时间复杂度为O(1)	顺序存取，按位置访问元素时间复杂度为O(n)
	插入、删除	平均移动约表中一半元素，时间复杂度为O(n)	不需移动元素，确定插入、删除位置后，时间复杂度为O(1)
适用情况		① 表长变化不大，且能事先确定变化的范围 ② 很少进行插入或删除操作，经常按元素位置序号访问数据元素	① 长度变化较大 ② 频繁进行插入或删除操作

8、有序顺序表合并

创建一个空表 Lc

依次从 La 或 Lb 中“摘取”元素 值较小的结点插 入到 Lc 表的最 后，直至其中一 个表变空为止

继续将 La 或 Lb 其中一个表的剩 余结点插入在 Lc 表的最后

9、有序链表合并

Lc 指向 La

依次从 La 或 Lb 中“摘取”元素值较小的结点 插入到 Lc 表的最后，直至其中一个表变空为止。

继续将 La 或 Lb 其中一个表的剩余结点插入在 Lc 表的最后。

释放 Lb 表的表头结点

10、栈满时的处理方法

报错，返回操作系统

分配更大的空间，作为栈的存储空间，将原栈的内容移入新栈

11、顺序栈初始化

分配空间并检查空间是否分配失败，若失败则返回错误

设置栈底和栈顶指针 S.top = S.base;

设置栈大小

12、顺序栈进栈

判断是否栈满，若满则出错

元素 e 压入栈顶

栈顶指针加 1

13、顺序栈出栈

判断是否栈空，若空则出错

获取栈顶元素 e

栈顶指针减 1

14、BF 算法设计思想

Index（S，T，pos）

将主串的第迫使个字符和模式的第一个字符比较，若相等，继续租个比较后续字符；若不等，从主串的下一个字符起，重新与模式的第一个字符比较。直到主串的一个连续子串字符序列与模式相等。返回值为 S 中与 T 匹配的子

序列第一个字符的序号，即匹配成功。否则，匹配失败，返回值 0。

15、广义表与线性表的区别

线性表的成分都是结构上不可分的单元素

广义表的成分可以是单元素，也可以是有结构的表

线性表是一种特殊的广义表

广义表不一定是线性表，也不一定是线性结构

16、广义表的特点

有次序性 一个直接前驱和一个直接后继

有长度 ＝表中元素个数

有深度 ＝表中括号的重数

可递归 自己可以作为自己的子表

可共享 可以为其他广义表所共享

程序题

1、抽象数据类型，以复数为例，定义个完整的抽象数据类型：

（1） 定义部分

```
ADT Complex{
数据对象
数据关系
基本操作：
       Creat(&C， x,y)
      GetReal （C）
   }ADT Complex
```

（2） 表示部分

```
Typedef struct
……
```

数据结构的存储结构类型定义通过 typedef 描述

（3） 实现部分

2、销毁线性表 L

```
Void DestroyList(SqList &L)
{if(L.elem)
 Delete[]L.elem;}
```

3、清空线性表 L

```
Void ClearList(SqList &L)
{L.length=0;}
```

4、求线性表 L 的长度

```
Int GetLength(SqList L)
{return (L.length);}
```

4、判断线性表 L 是否为空

```
Int IsEmpty(SqList L)
{if(L.length==0)return 1;
 Else return 0;}
```

5、线性表取值

```
Int GetELen(SqList L,int I,ElemType &e)
{if(i<1||i>L.length) return ERROR;
 e=L.elem[i-1]
 return OK;}
```

6、线性表查找

```
Int LocateElem(SqList L,ElemType e)
{for(i=0;i<L.length:i++)
   If(L.elem[i]==e)return i+1;
Return 0;
}
```

7、线性表插入

```
Status ListInsert_Sq(SqList &L,int I,ElemType e)
{if(i<1||i>L.length+1) return ERROR;
 If(L.length==MAXSIZE) return ERROR;
For(j=L.length-1;j>=j=1;j++)
 L.elem[j+1]=L.elem[j];
L.elem[i-1]=e;
++L.length;
Return OK;
}
```

8、线性表删除

```
Status ListDelete_Sq(SqList &L,int i)
{
  if((i<1||i>L.length))
  return ERROR;
  For(j=i;j<=L.length-1;j++)
  L.elem[j-1]=L.elem[j];
  --L.length;
  Return OK ;
}
```

9、单链表初始化

```
Status InitList_L(LinkList &L)
{
  L=new LNode;
  L->next=NULL;
  Return OK;
}
```

10、单链表销毁

```
Status DestroyList_L(LinkList &L)
{
  LinkList p;
  While(L)
  {
    P=L;
```

```
    L=L->next;
    Delete p;
  Return OK;
 }
```

11、单链表清空

```
Status ClearList (LinkList &L)
{
 LinkList p,q;
P=L->next;
While(p)
{
  Q=p->next;delete p=q;}
 L->next-NULL;
Return OK;
}
```

12、单链表求表长

```
Int ListLength_L(LinkList L){
  LinkList p;
  P=L->next;
  I=0;
  While(p){
   I++;
  p=p->next;}
  return I;
  }
```

13、单链表判断是否为空

```
 Int ListEmpty(LinkList L)
 {
   If(L->next)
     Return 0;
   Else
    Return 1;
   }
```

14、单链表插入

```
  Status ListInsert_L(LinkList &L,int i,ElemType e)
 {
   p=L;  j=0;
   while(p&&j<i-1){p=p->next;++j;}

   if(!p||j>i-1)return ERROR;

   s=new LNode;
   s->data=e;
   s->next=p->next;
```

```
  p->next=s; return OK; }
```

15、单链表删除

```
Status ListDelete_L(LinkList &L,inti,ElemType&e){
  p=L;j=0;
   while(p->next &&j<i-1){
   p=p->next; ++j; }
  if(!(p->next)||j>i-1) return ERROR;
  q=p->next;
  p->next=q->next;
   e=q->data;
   delete q;
   return OK; }
```

16、单链表的建立（前插法）

```
void CreateList_F(LinkList&L,int n){
L=new LNode; L->next=NULL;
for(i=n;i>0;--i){
p=new LNode;
cin>>p->data;
p->next=L->next;L->next=p;
 }
}//CreateList_F
```

17、单链表的建立（尾插法）

```
void CreateList_L(LinkList &L,int n) {
L=new LNode;
L->next=NULL;
 r=L;
for(i=0;i<n;++i)
{
p=new LNode;
 cin>>p->data;
 p->next=NULL; r->next=p;
 r=p;  } }
```

18、循环链表的合并

```
LinkList Connect(LinkList Ta,LinkList Tb)
{
  p=Ta->next;
  Ta->next=Tb->next->next;
  Delete Tb->next;
  Tb->next=p;
  Return Tb;
  }
```

19、线性表的合并

```
void union(List&La, List Lb){
 La_len=ListLength(La);
```

```
Lb_len=ListLength(Lb);
for(i=1;i<=Lb_len;i++)
{
  GetElem(Lb, i, e);
  if(!LocateElem( La, e))
  ListInsert(&La,++La_len,e); }
```

20、有序顺序表的合并

```
void MergeList_Sq(SqList LA,SqList LB,SqList &LC)
 { pa=LA.elem;  pb=LB.elem;
  LC.length=LA.length+LB.length;
   LC.elem=new ElemType[LC.length];
  pc=LC.elem;
   pa_last=LA.elem+LA.length-1;
   pb_last=LB.elem+LB.length-1;
  while(pa<=pa_last && pb<=pb_last)
  {  if(*pa<=*pb) *pc++ =*pa++;
   else *pc++ =*pb++;
  }
   while(pb <= pb_last)  *pc++=*pb++;
  while(pa <= pa_last)  *pc++=*pa++; }
```

21、有序链表合并

```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc){
 pa=La->next; pb=Lb->next;
pc=Lc=La;
 while(pa &&pb){
if(pa->data<=pb->data)
{ pc->next=pa;pc=pa;pa=pa->next;}
else{pc->next=pb; pc=pb; pb=pb->next;}
pc->next=pa?pa:pb;
```

22、顺序栈初始化

```
Status InitStack( SqStack &S )
{
S.base =new SElemType[MAXSIZE];
 if( !S.base ) return OVERFLOW;
S.top = S.base;
 S.stackSize = MAXSIZE;
return OK; }
```

23、判断顺序栈是否为空

```
bool StackEmpty( SqStack S )
{
    if(S.top == S.base) return true;
     else return false;
     }
```

24、求顺序栈的长度

```
int StackLength( SqStack S )
{
return S.top - S.base;
}
```

25、清空顺序栈

```
Status ClearStack( SqStack S )
{
if( S.base ) S.top = S.base;
return OK;
}
```

26、销毁顺序栈

```
Status DestroyStack( SqStack &S )
{
if( S.base )
 {
delete S.base ;
S.stacksize = 0;
S.base = S.top = NULL;
}
return OK;
}
```

27、顺序栈进栈

```
Status Push( SqStack &S, SElemType e)
{
  if( S.top - S.base== S.stacksize )
   return ERROR;
  *S.top++=e;
  return OK;
  }
```

28、顺序栈出栈

```
Status Pop( SqStack &S, SElemType &e)
{
if( S.top == S.base)
 return ERROR;
 e= *--S.top;
 return OK; }
```

29、取顺序栈栈顶元素

```
Status GetTop( SqStack S, SElemType &e)
{
  if( S.top == S.base ) return ERROR;
   e = *( S.top - 1 );
   return OK;
  }
```

30、链栈的初始化

```
void InitStack(LinkStack &S )
 { S=NULL; }
```

31、判断链栈是否为空

```
Status StackEmpty(LinkStack S)
{
if (S==NULL) return TRUE;
else return FALSE; }
```

32、链栈进栈

```
Status Push(LinkStack &S , SElemType e)
 {
   p=new StackNode;
   if (!p) exit(OVERFLOW);
   p->data=e; p->next=S; S=p;
   return OK;
   }
```

33、链栈出栈

```
Status Pop (LinkStack &S,SElemType &e)
{
 if (S==NULL) return ERROR;
 e = S-> data; p = S; S =S-> next;
 delete p;  return OK;
 }
```

34、取链栈栈顶元素

```
SElemType GetTop(LinkStack S)
 {
   if (S==NULL) exit(1);
    else return S->data; }
```

35、循环队列初始化

```
Status InitQueue (SqQueue &Q)
{
  Q.base =new QElemType[MAXQSIZE]
  if(!Q.base) exit(OVERFLOW);
   Q.front=Q.rear=0;
  return OK; }
```

36、求循环队列的长度

```
int  QueueLength (SqQueue Q)
{ return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE; }
```

37、循环队列入队

```
Status EnQueue(SqQueue &Q,QElemType e) {
if((Q.rear+1)%MAXQSIZE==Q.front)  return ERROR;
Q.base[Q.rear]=e;
Q.rear=(Q.rear+1)%MAXQSIZE;
return OK;
 }
```

38、循环队列出队

```
Status DeQueue (LinkQueue &Q,QElemType &e) {
if(Q.front==Q.rear) return ERROR;
 e=Q.base[Q.front];
 Q.front=(Q.front+1)%MAXQSIZE;
return OK; }
```

39、链队列初始化

```
Status InitQueue (LinkQueue &Q){
Q.front=Q.rear=(QueuePtr) malloc(sizeof(QNode));
if(!Q.front) exit(OVERFLOW);
Q.front->next=NULL;
return OK; }
```

40、销毁链队列

```
Status DestroyQueue (LinkQueue &Q){
while(Q.front){
Q.rear=Q.front->next;
free(Q.front);
 Q.front=Q.rear; }
return OK; }
```

41、判断链队列是否为空

```
Status QueueEmpty (LinkQueue Q)
{
 return (Q.front==Q.rear);
}
```

42、求链队列的对头元素

```
Status GetHead (LinkQueue Q, QElemType &e){
 if(Q.front==Q.rear) return ERROR;
 e=Q.front->next->data;
return OK;
}
```

43、链队列入队

```
Status EnQueue(LinkQueue &Q,QElemType e){
p=(QueuePtr)malloc(sizeof(QNode));
 if(!p) exit(OVERFLOW);
 p->data=e; p->next=NULL;
 Q.rear->next=p;
Q.rear=p;
 return OK; }
```

43、链队列出队

```
Status DeQueue (LinkQueue &Q,QElemType &e){
if(Q.front==Q.rear) return ERROR;
p=Q.front->next;
e=p->data;
Q.front->next=p->next;
```

```
if(Q.rear==p) Q.rear=Q.front;
delete p;
return OK; }
```

44、串的顺序存储

```
typedef struct{
char *ch;
int length; }
HString;
```

45、串的链式存储表示

```
#define CHUNKSIZE 80
typedef structChunk{
char  ch[CHUNKSIZE];
 structChunk *next;
 }Chunk;

typedef struct{
  Chunk *head,*tail;
  int curlen;
 }LString;
```

46、BF 算法描述

```
int Index(Sstring S,Sstring T,int pos){
  i=pos; j=1;
  while (i<=S.length&& j <=T.length){
   if ( S[ i ]=T[ j ]){++i; ++j; }
   else{ i=i-j+2; j=1; }
  if ( j>T.length) return i— T.length ;
  else return 0;
  }
```

遍历算法的分析

```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
    cout<<T->data;
    PreOrderTraverse(T->lchild);
    PreOrderTraverse(T->rchild);   }
}
```

先

```
Status PostOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
    PostOrderTraverse(T->lchild);
    PostOrderTraverse(T->rchild);
    cout<<T->data;   }
}
```

后

```
Status InOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
    InOrderTraverse(T->lchild);
    cout<<T->data;
    InOrderTraverse(T->rchild);}
}
```

中

47、