



中国科学技术大学  
University of Science and Technology of China

# 数据结构

线性表 (5 学时)

September 5, 2022

# 目录

---

- ① 类型定义
- ② 顺序表
- ③ 链表
- ④ 线性表的应用与习题

# 认识线性表

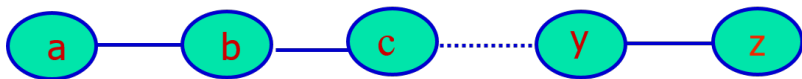
## 定义

- 线性表 (Linear list) 是  $n(\geq 0)$  个数据元素的有限序列, 记作  $(a_1, a_2, \dots, a_n)$ ,  $a_i, 1 \leq i \leq n$  是线性表中的数据元素,  $n$  是表长
- $i$  称为  $a_i$  在线性表中的位序;  $n = 0$  是空表。

## 特点

- 同一线性表中的元素具有相同特性
- 复杂的数据元素可由若干个数据项组成
- 相邻数据元素之间存在序偶关系

## 例子



# 认识线性表

例子：学生健康情况登记表，每一行是一个数据元素

姓 名	学 号	性 别	年 龄	班 级	健康状况
王小林	790631	男	18	计 91	健康
陈红	790632	女	20	计 91	一般
刘建平	790633	男	21	计 91	健康
张立立	790634	男	17	计 91	神经衰弱
⋮	⋮	⋮	⋮	⋮	⋮

## 特点

- 存在唯一的一个被称作“第一个”的数据元素；存在唯一的一个被称做“最后一个”的数据元素；
- 除第一个元素外，其他每一个元素有一个且仅有一个直接前驱；除最后一个元素外，其他每一个元素有一个且仅有一个直接后继。

## 错误的例子

- $(a,b,c,d)$ , 其中  $a=3, b=5, c='a', d=(1,2)$
- 自然数

# 线性表的抽象数据类型

## 线性表的 ADT

ADT List{

数据对象:  $D = \{a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0\}$

数据关系:  $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本操作:

InitList(&L)

DestroyList(&L)

ClearList(&L)

ListEmpty(L)

ListLength(L)

GetElem(L,i,&e)

LocateElem(L,e,compare( ))

PriorElem(L,cur\_e,&pre\_e)

NextElem(L,cur\_e,&next\_e)

ListInsert(&L,i,e)

ListDelete(&L,i,&e)

ListTraverse(L,visit( ))

} ADT List

在此ADT基础上还可定义更复杂的一些操作

## 逻辑结构的形式化

## 利用 ADT 定义更复杂的操作

### 复杂操作：集合的并

```
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, equal))  
            ListInsert(La, ++La_len, e);  
    }  
} //union
```

### 解释说明

- 上述算法的时间复杂度依赖 ADT List 定义中基本操作的执行时间
- 不同的数据结构影响基本操作的实现方式，导致基本操作的执行时间可能相差很大，从而导致上述算法的时间代价相差很大

# 利用 ADT 定义更复杂的操作

## 复杂操作：两个有序表的归并

两个有序表的归并(merge)操作

```
void MergeList(List La, List Lb, List & Lc){
    //非递减表La, Lb归并为非递减表Lc
    InitList(Lc);
    i=j=1; k=0;
    La_len=ListLength(La);
    Lb_len=ListLength(Lb);
    //指针平行移动，一次扫描完成
    while((i<=La_len) && (j <= Lb_len)){
        GetElem(La, i, ai); GetElem(Lb, j, bj);
        if (ai <= bj){
            ListInsert(Lc, ++k, ai); ++i;
        }
        else {ListInsert(Lc, ++k, bj); ++j;}
    }
    //若La或Lb还有剩余部分，则直接把这些元素
    //依次加到Lc表尾
    while (i<=La_len){ //La有剩余
        GetElem(La, i++, ai);
        ListInsert(Lc, ++k, ai);
    }
    while (j<=Lb_len){ //Lb有剩余
        GetElem(Lb, j++, bj);
        ListInsert(Lc, ++k, bj);
    }
} //union
```

## 算法思想:

- 用两个下标  $i, j$  分别指向有序表  $La, Lb$  的起始位置，将较小的表元素复制到  $Lc$  中，同时对应有有序表的下标加 1；
- 若其中任何一个有序表访问完毕，则将另一个有序表剩下的元素直接复制到  $Lc$  的末尾（用下标  $k$  指示该位置）。



## 从逻辑结构到存储结构

给出 ADT，就给出了线性表的逻辑结构，意味着：

- 人脑中对数据的概念和基本操作都已经明确了，可以借用这些概念和基本操作，去编写功能更强大的操作
- 但是，基本操作和强大的复杂操作的性能如何？（需要明了逻辑结构如何物理实现的）

这种逻辑结构如何在计算机/内存中进行物理实现？

- 了解物理实现的目的：理解和掌握原理，为将来高效利用内存空间，编写与物理实现密切相关的高效算法奠定基础
- 内存是线性的结构：将一排或明或灭的小灯想象成一串 0-1
- 顺序表：数据元素存储在内存的不同位置，位置关系对应逻辑关系（即 ADT 中定义的关系），通常呈序偶关系的数据元素其内存中的存储位置也相邻
- 链式表：数据元素存储在内存的不同位置，位置关系无法体现序偶关系，需要额外的存储空间或信息来标记序偶关系

## 顺序表：在内存中顺序表示线性表

### 定义

- 将线性表中的元素相继存放在一个连续的存储空间中

### 特点

- 存储结构：类似一维数组
- 存取方式：顺序访问，可以**随机存取**（指定访问第  $i$  个元素时，一步即可访问到）

例子，如图

1	2	3	4	5	6
45	89	90	67	40	78

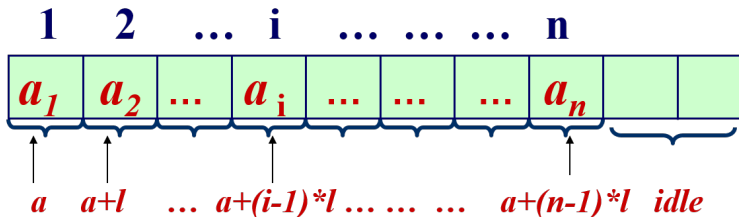
## 顺序表的存储方式

### 详细说明

假设每个元素占用  $l$  个存储地址，则

$$\text{LOC}(a_{i+1}) = \text{LOC}(a_i) + l$$

$$\text{LOC}(a_i) = \text{LOC}(a_1) + (i-1) * l$$



## C 语言中的顺序表

### C 语言的顺序表：一维数组

- 一般常用的程序设计语言都有一种数据类型：“数组”
- C 语言中的一维数组，就是顺序表的一种具体实现

### 接下来的目标

- 用程序设计语言 C 来实现线性表的 ADT，在内存的堆中“模拟”数组（这种数据类型）
  - 实现数据对象的存储
  - 实现数据关系的描述
  - 实现基本操作

# 顺序表的实现：静态部分

## 数据对象和数据关系的实现

```
1  #define LIST_INIT_SIZE 100 //线性表存储空间初始分配量
2  #define LISTINCREMENT 10 //存储空间分配增量
3
4  typedef struct {
5      ElemType *elem; //存储空间基址
6      int length;      //当前元素个数(表长)
7      int listsize;    //当前分配存储容量
8                      //以sizeof(ElemType)为单位
9  } SqList
10
11 //如果要实现一个存放整数的顺序表，可以用
12 typedef int ElemType;
13 //或者
14 #define ElemType int
```

## 顺序表的实现：动态部分

初始化线性表：从无到有，在内存中创建一个线性表

```
1 void InitList_sq(SqList &L){
2     L.elem =(ElemType *) malloc(LIST_INIT_SIZE*
        sizeof(ElemType)); //在堆中分配空间，字节数
3     if (!L.elem) exit(OVERFLOW); //分配失败就退出程序
4     L.length = 0; //分配成功，设置顺序表中元素个数为0
5     L.listsize = LIST_INIT_SIZE ; //设置表的最大存储容量
6 }
```

按值查找：在顺序表中从头查找结点值等于给定值  $x$  的结点

```
1 int LocateElem(SqList L,ElemType x,equal){
2     //顺序表遍历结束或找到 $x$ 就退出 for 循环
3     int i;
4     for(i=0;i<L.length && *(L.elem+i)!=x;i++) ; //循环体为空语句
5
6     //for 循环的不同退出原因，对应不同查找结果
7     return i<L.length ? ++i:0;
8 } //O(n)
```

# 顺序表的实现：动态部分

## 求表的长度

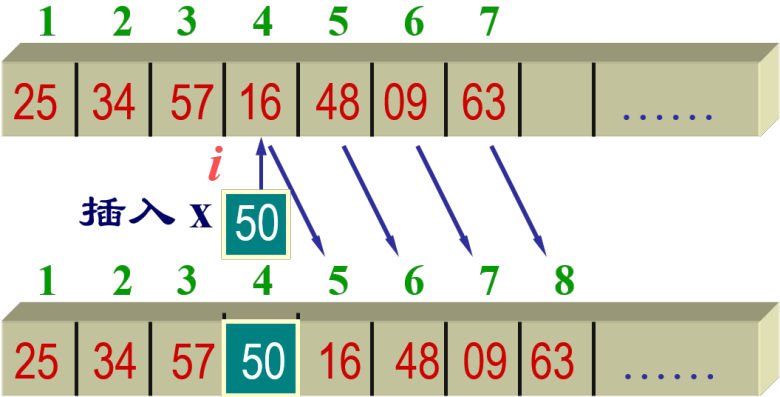
```
1 int ListLength(SqList L){
2     return L.length;
3 }//O(1)
```

## 提取函数：在表中提取第 $i$ 个元素的值

```
1 Status GetElem(SqList L, int i, ElemType &e){
2     if ( i >= 1 && i <= L.length ){//输入位置是否有效的判定
3         e= *(L.elem+i-1);
4         return(OK); //范围内有效位置，返回e
5     }
6     else{
7         printf("参数 i 不合理! \n");
8         return ERROR;}
9 }//O(n), 返回值Status表示程序执行状态，e是正确执行后返回的计算结果
```

# 顺序表的实现：动态部分

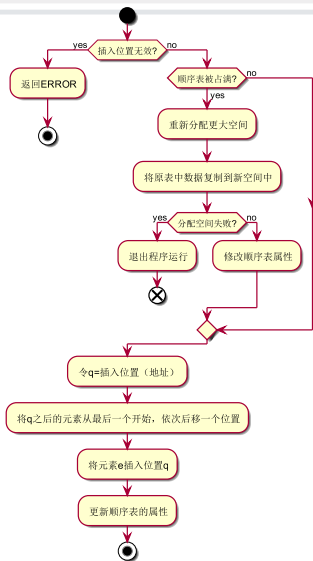
插入元素：例子





# 顺序表的实现：动态部分

## 插入元素



# 顺序表的实现：动态部分

## 插入元素

```
1  Status ListInsert_Sq(SqList &L, int i, ElemType e){
2      //在表L中第i个位置前插入新元素e
3      if (i<1 || i>L.length+1) return ERROR;
4      if (L.length>=L.listsize){//当前存储空间满，增加分配
5          newbase=(ElemType *) realloc(L.elem,
6              (L.listsize+LISTINCREMENT)*sizeof(ElemType));
7          if (!newbase) exit(OVERFLOW);//存储分配失败
8          L.elem=newbase;L.listsize+=LISTINCREMENT;
9      }
10     q=L.elem+i-1; //q为插入位置
11     for (p=L.elem+L.length-1;p>=q;--p)
12         *(p+1)=*p;    //插入位置及之后的元素右移
13     }
14     *(L.elem+i-1)=e;
15     L.length++;
16 }
```

//课后感兴趣的同学可以调查一下realloc的详细功能



## 顺序表的实现：动态部分

### 插入元素：性能分析

- 基本操作为“移动”（实质为赋值）操作；
- 在第  $i$  个位置前插入需移动  $n - i + 1$  个元素
- 假设在任何位置插入元素都是等概率的，则移动次数的期望值（平均次数）为：

$$\begin{aligned} E_{is} &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) = \frac{1}{n+1} (n + \dots + 1 + 0) \\ &= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2} \end{aligned}$$

- 故，此假设条件下，插入算法的平均时间复杂度为  $O(n)$

## 顺序表的实现：动态部分

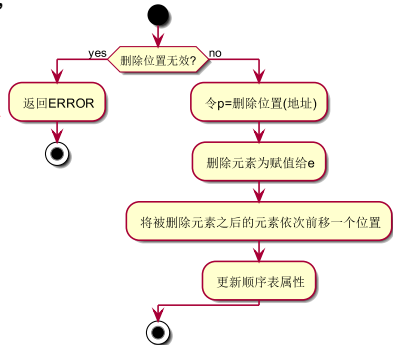
删除元素：例子



# 顺序表的实现：动态部分

## 删除元素

```
1 Status ListDelete_Sq(SqList &L, int i, ElemType &e){
2     //在表L中删除第i个元素,并用e返回其值
3     if ((i<1) || (i>L.length)) return ERROR;
4     p=L.elem+i-1; //p 为被删除元素的位置
5     e=*p;
6     q=L.elem+L.length-1; //表尾元素位置
7     for (++p;p<=q;++p) *(p-1)=*p; //元素左移
8     --L.length; //表长减一
9     return OK;
10 }
```



### 删除元素：性能分析

- 删除第  $i$  个位置的元素需（向左）移动  $n - i$  个元素
- 假设在任何位置删除元素都是等概率的，则移动次数的期望值（平均次数）为：

$$E_{dl} = \frac{1}{n} \sum_{i=1}^n (n - i) = \frac{1}{n} \frac{(n-1)n}{2} = \frac{n-1}{2}$$

- 故，在此假设条件下，删除算法的平均时间复杂度为  $O(n)$

# 顺序表的应用 1

## 定义更复杂的操作：集合的并

```
1 void Union(SqList &La, SqList Lb){
2     int n = ListLength(La);
3     int m = ListLength(Lb);
4     for (int i=1; i<=m; i++){
5         GetElem(Lb,i,e);           //在Lb中取一元素
6         if (!LocateElem(La,e,equal)) // La中若没有
7             ListInsert_Sq(La,++n,e); //则向La表尾插入该元素,时间复杂度O(1)
8     }
9 }
```

## 时空复杂度分析

- 时间复杂度:  $O(La.length * Lb.length)$
- 空间复杂度:  $O(1)$

## 顺序表的应用 2

### 定义更复杂的操作：集合相减

```
1 void Intersection(SqList &La, SqList Lb){
2     int n = ListLength(La);
3     int m = ListLength(Lb);
4     for (int i=1; i<=m; i++){
5         GetElem(Lb,i,e);           //在Lb中取一元素
6         if (k=LocateElem(La,e,equal)) // La中若有
7             ListDelete_Sq(La,k,e); //则从La中删除该元素
8     }
9 }
```

### 时空复杂度分析

- 时间复杂度：  $O(La.length * Lb.length)$
- 空间复杂度：  $O(1)$



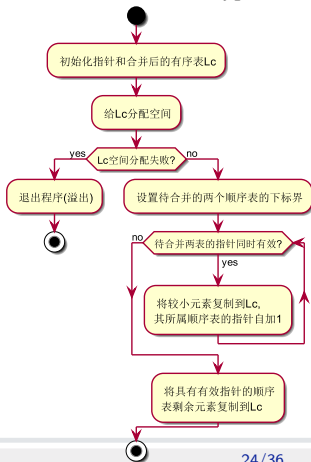
## 顺序表的应用 3

定义更复杂的操作：有序表的归并（两个有序表拼成更大的有序表）

```
1 void MergeList_Sq(SqList La,SqList Lb,SqList &Lc){
2     pa=La.elem;pb=Lb.elem;
3     Lc.listsize=Lc.length=La.length+Lb.length;
4     pc=Lc.elem=(ElemType *)malloc(Lc.listsize * sizeof(ElemType));
5     if (!Lc.elem) exit(OVERFLOW);
6     pa_last =La.elem+La.length-1;
7     pb_last =Lb.elem+Lb.length-1;
8     while (pa <=pa_last &&pb<=pb_last){
9         if (*pa<=*pb) *pc++=*pa++;
10        else *pc++=*pb++; }
11    while (pa<=pa_last) * pc++=*pa++;
12    while (pb<=pb_last) * pc++=*pb++;
13 }//MergeList_Sq
```

### 时空复杂度分析

- 时间复杂度：  $O(La.length + Lb.length)$
- 空间复杂度：  $O(La.length + Lb.length)$



# 顺序表的总结与思考

## 线性结构：定义与例子

- 线性结构的顺序表示方法
- 线性结构的顺序表示方法的实现
- 线性结构的顺序表示的应用与例子

## 顺序表的优缺点

- 优点：逻辑上相邻，物理上也相邻，可随机存取其中任一元素，存储位置可用公式计算
- 缺点：插入删除的附加操作过多，移动大量的元素

## 思考

- 分析顺序表的插入与删除操作的最坏时间复杂度

# 链表：在内存中链式表示线性表

## 单链表

- 用一组任意的存储单元存放线性表中的数据元素
- 用额外的方式（指针/地址/位置等）存储数据元素间的关系

## 特点

- 链式存储结构
- 存储单元可以不连续
- 顺序存取：访问第  $k$ ，必须要从第 1 个开始，沿着逻辑顺序关系依次访问第 2, 3, ...,  $k$  个元素

例子，如图

31

头指针

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
...	...	...
19	WANG	NULL

# 链表的实现：静态部分

## 单链表的数据对象和数据关系的实现

```
1      typedef struct Lnode { //链表结点
2          ElemType data;      //结点数据域
3          struct Lnode * next; //结点链域
4      } ListNode, * LinkList;
```

Node

**data**    **next**

# 单链表基本操作的实现及应用

## 基本操作的实现（略，课后阅读教材）

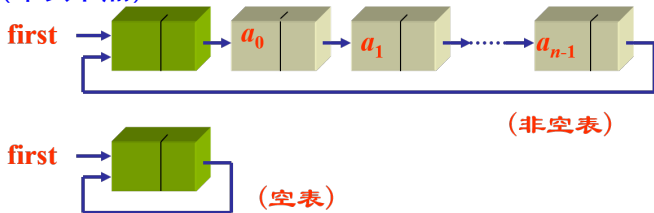
- 插入
- 删除
- 构建链表
- 清空链表
- 求链表长度
- 按值查找
- 按序号查找

## 应用

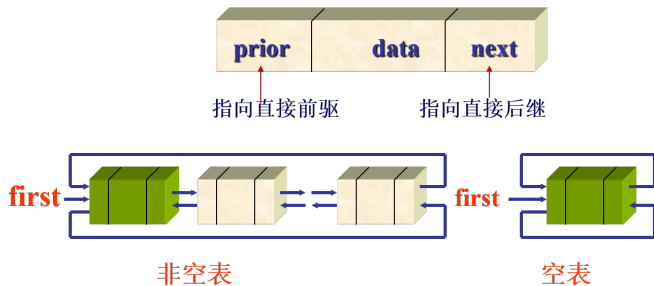
- 链表的归并

# 不同类型的链表

## 循环链表（带头节点）



## 双向链表



# 双向循环链表的实现

## 实现数据对象和数据关系

```
1 typedef int ListData;
2 typedef struct dnode {
3     ListNode data;
4     struct dnode * prior, * next;
5 } DblNode;
6 typedef DblNode * DblList;
```

## 实现基本操作

- 课后阅读教材相关章节

# 顺序表和链表的比较

## 基于空间的比较：存储密度

- 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量
- 顺序表的存储密度  $\approx 1$  （当数据存满时约等于 1）
- 链表的存储密度  $< 1$

## 基于时间的比较：插入/删除时移动元素个数

- 顺序表平均需要移动近一半元素
- 链表不需要移动元素，只需要修改指针

## 存取方式

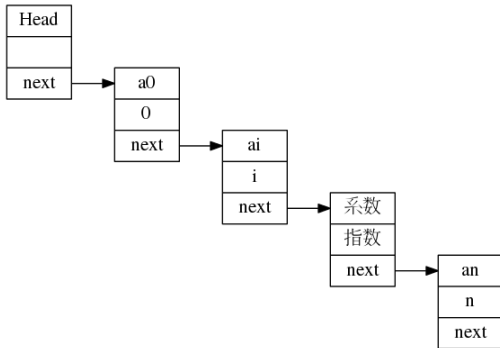
- 顺序表可以随机存取，也可以顺序存取
- 链表是顺序存取的，不能随机存取



## 练习题

### 单变量多项式 $a_0x^0 + a_1x^1 + \dots + a_nx^n$ 的表示和运算

- 表示方法一：顺序表  $(a_0, a_1, \dots, a_n)$
- 表示方法二：单链表，如下图所示，要保存系数和指数，系数为 0 的项不需要出现在链表里

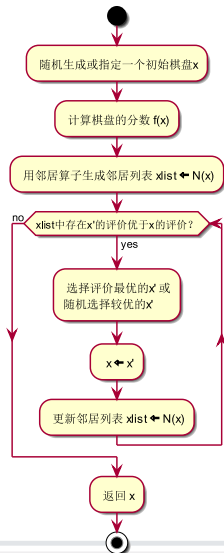


# 练习题

## $n$ 皇后问题（爬山法）

- 用线性表的知识，设计 ADT 和算法，给出一种在  $n \times n$  的国际象棋棋盘上放置  $n$  个彼此攻击不到的皇后的放置方法
- 提示 1：输入  $n$ ，用线性表来设计棋盘或棋局的逻辑结构；例如：线性表  
 $x = (a_1, a_2, \dots, a_n)$ ,  $a_i \in \{1, 2, \dots, n\}$  记录第  $i$  列皇后的行号；或者  $n \times n$  的 0-1 矩阵表示棋盘
- 提示 2：采用随机搜索思想（爬山法），即
  - 设计棋盘的打分机制/评价函数  $f(x)$ ，对放置了  $n$  个皇后的棋盘进行评价；例子：相互能攻击到的皇后对越多，分数越低
  - 设计邻居算子  $N(x)$ ：即从棋盘  $x$  到其他棋盘的函数；例：改变当前棋盘  $x$  中某个皇后位置，得到一系列棋盘
- 提示 3：归纳总结算法过程中需要用到的基本操作
- 思考：在 1 分钟内能求解的  $n$  皇后问题，最大  $n$  是多少？写代码找到自己上述代码能实现的最大  $n$ 。

## 爬山法思想



## 练习题：遗传算法

### 问题描述

- 要求：实现遗传算法 ADT，测试多项式函数在给定区间上的优化问题
- 应用：遗传算法是人工智能/数学优化领域内，模拟生物进化过程，用于实现求任意函数（连续性要求）的最小值或最大值
- 基本概念：一组解构成一个集合，称之为群体/种群，每个解称为个体（或染色体）；一个解  $(x, y)$  包括自变量  $x$  和被称之为“适应度”的实数目标值  $y = f(x)$
- 比如求一元  $k$  次函数的最值，随机选择 20 个解（包括  $x, y$ ）/个体，构成一个种群  $P$ ；机器学习中参数的优化问题

### 关键的第一步：数据对象及关系

- 单个解和群体的逻辑结构和存储结构是什么？单个解/个体可以视为一个数据元素；群体可以视为线性表；存储结构呢？
- 群体内解之间关系：同属一个集合或按适应度值大小排序；存储结构呢？
- 逻辑结构依问题相应设计；存储结构设计需要考虑“操作”及其“性能”

## 练习题：遗传算法

从具体的数值优化问题（求数值函数的最值），感性认识问题

- 例如：求  $y = x^7 + 4x^4 - 6x^2 + 4$  在区间  $[-10, 10]$  上的最小值。
- 动手编程之前，先思考：有哪些对象/数据？这些数据对象如何用逻辑结构描述？逻辑结构采用怎样的存储结构在内存中实现？

### 一种数据对象及关系的实现

- 解：每个解的自变量  $x$  编码为长度为  $n$  的二进制串/染色体，二进制串解码为  $x$ ；存储结构设计
  - $n$  是调节算法性能的参数，决定解的精度
  - 如何确保每个长度为  $n$  的 0-1 串和定义域内的  $x$  一一对应？如果不能一一对应，有没有其它解决策略？
- 数据对象：群体， $k$  个解构成的线性表（单链表或顺序表）， $k$  是超参
- 数据关系：同属一个集合，或序偶（适应度值大小排序）

## 练习题: 遗传算法

### 遗传算法的主要操作及流程

- 算法初始化, 随机生成  $k$  个解, 形成群体  $P$
- 有性繁殖操作/交叉算子: 从种群  $P$  中随机选择两个个体/染色体, 交换两个染色体的尾部 (尾部的起点位置, 随机决定), 得到两个新的染色体; 重复多次得到  $P'$
- 无性繁殖/变异算子: 对  $P'$  中的每个染色体, 以给定的变异概率  $p = 1/n$ , 翻转染色体的每个位置上的 0 或 1, 得到中间群体  $P''$
- 计算  $P''$  中每个染色体的适应度值  $y = f(x), x \in P''$
- 适者生存/选择算子: 将中间群体  $P''$  和种群  $P$  合并, 从并集中选择  $k$  个解, 构成新的种群  $P$ , 选择的方法: 按  $y$  的大小排序, 选择适应度更优的部分。(求最小值, 适应度  $y$  值小的更优)
- 上述三个算子依次不断迭代, 直到群体  $P$  不再变化或满足其它停止条件为止。

