

# 数据结构与算法大全

## 目 录

开始之前	1
序言	1.1
作者的话	1.2
本书的前情提要	1.3
参考资料	1.4
注意事项	1.5
入门	2
数据的基本概念	2.1
算法的基础知识	2.2
线性结构（线性表）	3
线性表入门	3.1
动态分配	3.2
顺序表	3.3
链表入门	3.4
循环链表	3.5
双向链表	3.6
静态链表	3.7
栈	3.8
队列	3.9
串	3.10
总结与整理	3.11
过渡	4
数组	4.1
矩阵压缩	4.2
广义表	4.3
总结与整理	4.4
树	5
树的入门	5.1
森林与树	5.2
二叉树的入门	5.3
树以及二叉树的相关计算	5.4
二叉树的创建	5.5
遍历二叉树	5.6

树的遍历还原	5.7
二叉树的计数 ( 用代码 )	5.8
线索二叉树	5.9
赫夫曼树 ( 最优二叉树 )	5.10
回溯法	5.11
总结与整理	5.12
<b>图</b>	<b>6</b>
图的入门	6.1
图的储存结构	6.2
图的创建	6.3
图的遍历	6.4
最小生成树	6.5
有向无环图	6.6
拓扑排序与关键路径	6.7
最短路径	6.8
总结与整理	6.9
<b>查找</b>	<b>7</b>
一些想法	7.1
查找表及顺序查找	7.2
折半查找及查找树表	7.3
二叉排序树	7.4
哈希表	7.5
<b>排序</b>	<b>8</b>
内部排序的概述	8.1
直接插入排序	8.2
希尔插入排序	8.3
快速排序	8.4
简单选择排序	8.5
堆排序	8.6
<b>尾声</b>	<b>9</b>
中英文对照表	9.1
后记	9.2

# 序言

在浩瀚的计算机科学领域中，数据结构与算法无疑是其中最璀璨夺目的明珠。然而，想要真正掌握这门学科，却并非一撮而就的易事。它需要我们有足够的耐心，去一步步探寻每一个细节；它需要我们有足够的毅力，去克服每一个难关。因此，学习数据结构与算法，更像是一场循序渐进的修行，需要我们在实践中不断积累、反思与提升。

本书《数据结构与算法大全》正是基于这样的理念而诞生。它不仅仅是一本简单的教程，更是一部集大成之作。在编写过程中，我们深入研究了各种数据结构与算法的原理、特性及应用场景，力求将最精华、最实用的内容呈现给读者。同时，我们也参考了大量的专业书籍、学术论文以及实际项目经验，以期为读者提供一个全面、深入的学习平台。

在书中，你将看到各种经典的数据结构，如数组、链表、栈、队列、树和图等，以及与之相关的算法实现和优化方法。我们将从基础概念讲起，逐步深入到高级应用和算法设计技巧，让你能够逐步建立起自己的知识体系和技能体系。

此外，本书还注重理论与实践的结合。我们不仅在理论层面详细讲解了数据结构与算法的基本原理和特性，还通过大量的示例和案例，让读者能够更好地理解其在实际问题中的应用。相信通过本书的学习，你不仅能够掌握数据结构与算法的核心知识，还能够将其应用于实际项目中，提升自己的编程能力和问题解决能力。

最后，我要感谢所有为这本书付出努力的人，也要感谢每一位选择阅读这本书的读者。希望它能够成为你学习数据结构与算法的良师益友，陪伴你走过一段充满挑战与收获的学习旅程。愿你在这个领域中不断前行，创造出更加辉煌的未来。

# 作者的话

我依稀记得，小时候的我，便与计算机结下了不解之缘。那时的我，对计算机充满了好奇，总是想要探索这个神奇的小盒子里的奥秘。每次坐在电脑前，我都会瞪大眼睛，仔细观察屏幕上的每一个图标，尝试着去理解它们的功能和用途。

随着时间的推移，我逐渐对计算机产生了热爱之情。我开始沉迷于计算机的命令行，那些看似复杂的指令和代码，在我眼中却充满了魅力。我喜欢通过命令行来控制计算机，完成各种任务，那种掌控感让我兴奋不已。我开始研究各种计算机知识，从硬件到软件，从操作系统到编程语言，我都想要了解得更多。

在小学五年级的时候，我第一次接触到了C语言。那时的我，满怀激情地投入到学习中，想要掌握这门强大的编程语言。然而，现实却给我泼了一盆冷水。我发现，我所使用的书籍中充满了错误，这让我感到十分沮丧。最终，我无奈地选择了放弃，转而寻找其他更适合我的编程语言。

初中的时候，我接触到了Python。与C语言相比，Python的语法更加简洁易懂，让我越用越得心应手。我开始用Python编写各种小程序，解决实际问题。我喜欢Python的灵活性和强大性，它让我感受到了编程的乐趣。

在填报高考志愿时，我坚决地选择了计算机专业，不顾家人的阻拦。他们总是以我那高考成绩只够上二本末的堂哥为例子，试图说服我放弃这个选择。然而，我深知自己的兴趣和优势所在，我坚信自己能够在计算机专业中取得好成绩。最终，我成功地进入了大学，开始了我的计算机学习之旅。

在上海的校园里，我遇到了一些同样是学计算机的同学。他们中有些人总是喜欢在我面前炫耀自己的成绩和成就，搞些所谓的“凡尔赛”。然而，我对此却甚为厌恶。我深知，真正的进步和成就不是靠炫耀得来的，而是需要脚踏实地地学习和实践。因此，我选择了超越他们，用自己的努力和成果来证明自己的实力。我坚信，只要我坚持不懈地努力下去，我一定能够在计算机领域取得属于自己的辉煌成就。

在未来的日子里，我将会继续深耕计算机领域，不断提升自己的技能和知识。无论是面对复杂的算法问题，还是应对日新月异的科技变革，我都会保持一颗求知的心，勇于挑战自我，不断超越极限。

或许，我会遇到更多的困难和挑战，但我相信，只要我坚定信念，勇往直前，就一定能够克服一切难关，实现自己的梦想。我期待着在计算机的世界里，书写属于自己的精彩篇章，用智慧和汗水创造出更多的可能。

作者的话

回首过去，我为自己的选择和坚持感到自豪。展望未来，我充满了信心和期待。我相信，在不久的将来，我将成为一名优秀的计算机专业人才，为社会的进步和发展贡献自己的力量。

# 本书的前情提要

对于一些特殊的东西，如 `ElemType` 等等的，需要在运行前声明（使用 `#define` ) 的，在这里都有说明。

名称	解释
ElemType	表示项的类型
null	与NULL相类似，就是表示空的意思
INITALSIZE	动态分配中的顺序线性结构初始大小
MAXSIZE	顺序储存结构中的最大储存容量
True	为真
False	为假
status	表示状态的数据类型
ElementType	与ElemType一样，都是表示项的类型

# 参考资料

## 书籍



数据结构 ( C语言版 ) | 作者 : 严蔚敏女士



大话数据结构 | 作者 : 程杰先生



数据结构与算法图解 | 作者 : [美国] 杰伊•温格罗



算法图解 | 作者 : [美国] Aditya Bhargava

## 博客



史上最详细!用栈实现二叉树的建立以及先/中/后序遍历\_C语言

## 课程



数据结构与算法 | 主讲 : 杨雯静老师



数据结构 | 主讲 : 陈越、何钦铭

# 注意事项

## 一些要注意的事情

1. 本书将会加入水印
2. 本书是基于严蔚敏女士的《数据结构（C语言版）》编写的，有一些地方会选择直接引用。
3. 禁止商业用途。

## 相关条文

《中华人民共和国著作权法》第二十二条规定：

在下列情况下使用作品，可以不经著作权人的许可，不向其支付报酬，但应当指明作者姓名、作品名称，并且不得侵犯著作权人依照本法享有的其他权利：

（一）为个人学习、研究或者欣赏，使用他人已经发表的作品；

# 数据的基本概念

## 数据 ( data )

能输入到计算机当中的并能被处理的符号的总称。

## 数据元素 ( data item )

数据的基本单位，通常是以一个整体去考虑，有若干个数据项组成，又名记录。

在计算机当中，最小的储存单位是位 ( bit )，就是二进制数的一位。将若干个位组合起来连成一个位串表示一个数据元素，通常这个位串为一个元素 ( element ) 或结点 ( node )，有时可以看成是数据元素在计算机中的映像。

当数据元素组成各个数据项时，位串应于各个数据项的子串称为数据域 ( data field )

## 数据对象 ( data object )

性质相同的数据元素的集合，是数据的子集。

## 数据结构 ( data structure )

存在一种或一种以上的特殊关系的数据元素的集合—— $Data\_Structure = (D, S)$

D是数据元素的有限集，S是D上关系的有限集

数据结构通常分为4种基本结构：

1. 集合
2. 线性
3. 树
4. 图

数据结构在计算机的表示成为数据的物理结构 ( physical structure )，又称为储存结构 ( storage structure )，同时，这也包括了数据元素的表示和关系的表示。

## 数据项 ( data element )

## 数据的最小单位

# 数据类型 ( data type )

用于刻画操作对象的特性：譬如，整型数据可以进行加减乘除，字符型的只能进行加减  
两大分类：

1. 原子类型，譬如：C语言中的整型、浮点型、字符型、枚举型
2. 结构类型：若干个数据类型组成（可以是结构的、也可以是非结构的）

# 抽象数据类型 [ ADT ] ( abstract data type )

分为两种类型：

1. 原子类型
2. 结构类型
  1. 固定聚合类型，“值”确定的
  2. 可变聚合类型，“值”的成分不确定

定义上分为三个部分：数据对象 ( data object )、数据关系 ( data relation )、基本操作。

数据的关系描述的是数据元素之间的逻辑关系，由此被称为逻辑关系 ( logical structure )。

数据关系在计算机的储存方法有两种：

1. 顺序映像
  2. 非顺序映像
- 同时，也有两种储存结构：
3. 顺序储存结构 ( sequential storage structure )
  4. 链式储存结构 ( linked storage structure )

# 算法的基础知识

## 算法的五大特性

1. 确定性
2. 有穷性
3. 可行性
4. 输入和输出 ( 可以没有输入 , 但是得要有输出 )

## 算法的要求

1. 正确性 ( correctness )
2. 可读性 ( readability )
3. 健壮性 ( robustness )
4. 效率与低储存量需求

## 辨析

1. 程序是不是算法 ?

答案是False , 因为程序可以没有输入。

## 算法的书写

我们要把它写成一个函数。

声明行由返回值类型 ( int、char、double、float、void、long int ) 、函数名称和形式参数声明区构成

```
returnValueType functionName(type1 value1; type2 value2) {  
    command 1;  
    command 2;  
    ....
```

```
    command n;  
}
```

如果没有声明行，那一定要原型声明，如果不进行原型声明，就有可能造成C语言去猜测变量。

如果输入参数与定义的不匹配，那么会发生变量类型转换。

当然如果你把要声明的函数放到main函数的前面或者是放到头文件当中就不会需要声明行了。

总之，要知道，我们要用到函数来表示一个算法。譬如，我们要从顺序表（从零号位开始存数据，里面的数值都大于等于0）中返回指定位置的值：

```
/*  
#define maxSize 100  
typedef struct {  
    elementType data[maxSize];  
    int length;  
};  
*/  
elementType initAlist(seqList *L, int i) {  
    if(i > L->length) return -1;  
    return L->data[i-1];  
}
```

至于顺序表是什么，为什么要特别声明是从0号位开始，我们以后再说。

不过，在这里形式参数声明区代表的是要输入的值的区域，函数名称得表达出算法的用途，实在不行可以随意。

## 算法的衡量标准

即时间复杂度（time complexity）和空间复杂度（space complexity）。

### 时间复杂度（time complexity）

又分为事前估计和事后估计

# 事前估计

## 语句频度

1. 顺序结构、分支结构、循环结构——运行次数会有变化，取最大的运行次数。

## 渐进时间复杂度 ( asymptotic time complexity )

2.  $T(n) = O(f(n))$ ，渐进时间复杂度。

3. 主要是要找到关键操作（递归和循环），就是嵌套最深的语句，可以是判断、也可以是普通语句。

4. 当存在最好和最坏情况后，用平均复杂度。

## 例题

### 1. 多重循环

#### 例题分析

##### 例一：

```

1 | int m=0,i,j;
2 | for (i=1;i<=n;i++)
3 |   for(j=1;j<=2*i;j++)
4 |     m++;

```

第一步列出循环中的变化值：

第二步列出内层语句的执行次数：

i	1	2	3	4	5	.....	n
内层语句执行次数	2	4	6	8	10	.....	$2^n$ 次

第三步 求和，写结果

$$2 + 4 + \dots + 2n = \frac{2+2n}{2}n = n(n+1)$$

$$T = O(n^2)$$

**例二:**

```

1 | for (i=0;i<n;i++)
2 |   for(j=0;j<m;j++)
3 |     a[i][j] = 0;

```

**第一步**列出循环中的变化值:

**第二步**列出内层语句的执行次数:

i	0	1	2	3	4	.....	n-1
内层语句执行次数	m	m	m	m	m	.....	m次

**第三步**求和, 写结果

$$m * (n - 1 - 0 + 1) = m * n$$

$$T = O(mn)$$

## 2. 一重循环

**例一:**

```

1 | i = n*n;
2 | while(i != 1)
3 |   i = i/2;

```

**第一步:**列出循环趟数t及每轮循环i的变化值:

t	0	1	2	3
i	$n^2$	$\frac{n^2}{2}$	$\frac{n^2}{4}$	$\frac{n^2}{8}$

**第二步:**找到t与i的关系:

$$i = \frac{n^2}{2^t}$$

**第三步:**确定循环停止条件:

$$i = 1$$

**第四步:**联立第二步第三步两式解方程:

$$\frac{n^2}{2^t} = 1 \quad n^2 = 2^t \quad t = \log_2 n^2$$

$$t = \log_2 n^2 = 2 \log_2 n$$

所以得到时间复杂度为:

$$T = O(\log_2 n)$$



d9379049-6e45-4356-a8b0-f19cb2ecb852.png

**例三:**

```

1 | int i = 1;
2 | while (i<=n)
3 |     i = i *2

```

**第一步:** 列出循环趟数t及每轮循环i的变化值:

t	0	1	2	3	4
i	0	1	2	3	4

**第二步:** 找到t与x的关系:

$$i = 2^t$$

**第三步:** 确定循环停止条件:

$$i = n$$

**第四步:** 联立第二步第三步两式解方程:

$$2^t = n$$

$$t = \log_2 n$$

所以得到时间复杂度为:

$$T = O(\log_2 n)$$

**例四:**

```

1 | int i = 0;
2 | while (i*i*i<=n)
3 |     i++;

```

**第一步:** 列出循环趟数t及每轮循环i的变化值:

t	0	1	2	3	4
i	0	1	2	3	4

**第二步:** 找到t与x的关系:

$$i = t$$

**第三步:** 确定循环停止条件:

$$i^3 = t$$

**第四步:** 联立第二步第三步两式解方程:

$$t^3 = n$$

$$t = \sqrt[3]{n}$$

所以得到时间复杂度为:

$$T = O(\sqrt[3]{n})$$

**例五：**

```
1 | y = 0;
2 | while (y+1)*(y+1) <= n
3 |   y = y+1;
```

第一步：列出循环趟数t及每轮循环y的变化值：

t	0	1	2	3	4
y	0	1	2	3	4

第二步：找到t与x的关系：

$$t = y$$

第三步：确定循环停止条件：

$$(y + 1)^2 = n$$

第四步：联立第二步第三步两式解方程：

$$(t + 1)^2 = n$$

$$t = \sqrt{n} - 1$$

所以得到时间复杂度为：

$$T = O(\sqrt{n})$$

## 事后统计

1. 利用计算机的计时工具，用一组或多组数据去测。缺点是要运行程序，还会依赖于硬件、软件等因素。

## 空间复杂度（ space complexity ）

算法本身会占用：输入、输出、指令、常数、变量等。

看看弄出了多少空间被占用。

注意：如果是递归的算法，那就是看层数，不是看节点数，因为递归算法是单线程的弄完一个以后，就会删除掉。

# 线性表入门

## 定义

线性表 (linear list) 有以下三个规则：

1. 存在唯一的一个“第一个”数据元素
2. 存在唯一的一个“最后一个”数据元素
3. 除“第一个”和“最后一个”元素均只有一个直接前驱 (immediate predecessor) 和一个直接后继 (immediate successor)。

## 一些参数

线性表长度为 `n`，也可以直接用 `xxLen` 表示

当 `n=0` 时，就是空表

`a` 的下标 `i` 表示的是 `a(i)` 在线性表的位序

## 一些要说的东西

对于线性表存在两种输入的情况：

1. 不修改内容，只是把内容传入，如 `List L`
2. 譬如，`getLength(List L)`
3. 可修改内容，也可把内容传入，就传地址（指针），如 `List *L`
4. 譬如，`initList(List *L)`
5. 但是，在此后的访问要用到 `L->`

但是，我们要注意的是结构体。

举个例子

这是一个动态分配

```
typedef struct {  
    ElemtType *data;
```

```
int length;
int listSize;
} DA;
```

注意，如果我们要修改结构体变量的内容（即data、length、listSize），那就直接传地址，如果我们只是读取，那就传变量即可。

譬如，我们的读取函数

```
void readDA(DA da) {
    for(int i = 0; i < da.length; i++) printf("%d", da.data[i])
}
```

这就是直接传变量本身。

如果们要对该变量本身进行一系列的修改的时候，下面的增加函数代码

```
void addContent(DA *da, ElemenType content) {
    ElemenType *p = da->data+da->length;
    *p = content;
    da->length++;
}
```

这就是直接传地址的。

# 动态分配

## 前情提要

在之前的C语言的学习当中，我们提到了 `malloc` 和 `free`，这两个函数是用来动态分配内存的。

看好。

动态分配的代码是这样写的。

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    scanf("%d", &n);
    int *a = (int*)malloc(sizeof(int)*n);
    free(a);
}
```

## 结构体结构

```
typedef struct {
    ELEMType *data;
    int length;
    int listSize;
} DA;
```

我来解释一下，这个结构体声明。

`length`是你实际存了多少的东西，`listSize`是你目前表的最大长度。

## 动态分配的使用

其实，无非就是创、增、删、改、读、销。

## 创建一个动态分配

其实，就是借空间，然后初始化。当然，有的是选择将借空间和初始化这两个操作分开来做。

```
#define INITIALSIZE 100  
DA da;  
da.listSize = INITIALSIZE;  
da.length = 0;
```

## 初始化动态分配

算了，我还是写出来吧。

```
void initDynamicAllocation(DA *da) {  
    da->data = (ElemType*)malloc(sizeof(ElemType)*INITIALSIZE);  
    /* initial process */  
}  
  
### 添加内容  
```c  
void addContent(DA *da, ElemType content) {  
    ElemType *p = da->data+length;  
    *p = content;  
    da->length++;  
}
```

# 顺序表

## 别称

线性表的顺序表示、顺序存储结构、顺序映像、随机存取结构的储存结构

## 作用

利用数组的连续存储空间顺序存放线性表的各个元素

$a[n-1]$  是  $a[n]$  的直接前趋，  $a[n+1]$  是  $a[n]$  的直接后继。

## 结构体代码

### 第一种写法

```
typedef struct sqList {  
    ElementType Data[MAXSIZE];  
    int Last;  
} sqList;
```

#### ▼ 其他语言

### Python

```
class LNode:  
    def __init__(self):  
        self.Data = []  
        self.last = -1
```

## 第二种写法

```
typedef struct sqList {  
    ElementType Data[MAXSIZE];  
    int length;  
} sqList;
```

▼ 其他语言

## Python

```
class LNode:  
    def __init__(self):  
        self.Data = []  
        self.length = 0
```

## 第三种写法

```
typedef struct sqList {  
    ElementType *Data;  
    int length;  
} sqList;
```

▼ 其他语言

## Python

```
class LNode:  
    def __init__(self):  
        self.Data = None  
        self.length = 0  
    ...
```

# 初始化

## 通过输入来实现的初始化

```
int initList(sqList *List, int n) {
    if(MAXSIZE < n) return 1;
    ElementType *p = List->Data;
    for(int i = 0; i < n; i++, List->length = i) scanf("%d", p+
    return 0;
}
```

### ▼ 其他语言

## Python

```
def initList(List: LNode, n):
    for i in range(0,n):
        List.Data.append(int(input()))
        List.length += 1
    ...
```

## 通过读取文件实现的初始化

```
int initList(sqList *List, int n, FILE *fp) {
    if(MAXSIZE < n || fp == NULL) return 1;
    ElementType *p = List->Data;
    for(int i = 0; i < n; i++, List->length = i) {
        if(sizeof(ElementType) == 4) *(p++) = fgetc(fp) - '0';
        else if(sizeof(ElementType) == 1) *(p++) = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
    }
```

```
    }
    return 0;
}
```

## 增加

### 行尾增加

#### ▼ 代码

C

```
int insertNodeFromEnd(sqList *List, int n) {
    if(MAXSIZE < List->length+n) return 1;
    ElementType *p = List->Data + List->length;
    for(int i = 0; i < n; i++) scanf("%d", p++);
    List->length = n + List->length;
    return 0;
}
```

Python

```
def insertNodeFromEnd(List, n):
    for i in range(1, n+1):
        List.Data.append(input())
        List.length += 1
```

### 行首增加

主要原理就是在添加一个Node时，先将原有的向后移一个，再添加。

## ▼ 代码

C

```

int insertNodeFromHead(sqList *List, int n) {
    if(MAXSIZE < List->length+n) return 1;
    ElementType *p = List->Data;
    for(int i = 0; i < n; i++) {
        for(ElementType *q = List->Data+List->length; q !=
            scanf("%d", p);
            List->length++;
    }
    return 0;
}

```

Python

```

def insertNodeFromHead(List, n):
    for i in range(0, n):
        List.Data.insert(0, input())
        List.length += 1

```

中间插入

## ▼ 代码

C

```

int insertNodeFromMiddle(sqList *List, int insertPostion, i
    if(MAXSIZE < insertPostion+n || MAXSIZE < insertPostion
        printf("insertPostion > List->length");
        return 1;

```

```

    }
    ElementType *p = List->Data + insertPostion;
    for(int i = 0; i < n; i++) {
        for(ElementType *q = List->Data+List->length; q != 
            scanf("%d", p);
            List->length++;
    }
    return 0;
}

```

## Python

```

def insertNodeFromMiddle(List, insertPostion, n):
    for i in range(0, n):
        List.Data.insert(insertPostion, input())
        List.length += 1

```

## 有序插入

原理：从后往前找——>比它小的数的后面

### ▼ 代码

## C

```

int insertANode(List va, ElementType a) {
    if(va->length > MaxSize) {
        printf("序列已满");
        return 1;
    }
    ElementType *p = va->data + va->length-1;
    while(p != va->data && a <= *p) {
        *(p+1) = *p;

```

```
    p--;  
}  
*(p+1) = a;  
va->length++;  
return 0;  
}
```

## 删除

### 通过序号删除

原理：从后往前找——>比它小的数的后面

#### ▼ 代码

C

```
int removeNodeThroughPosition(List *L, int position) {  
}
```

### 通过匹配结果删除

原理：

#### ▼ 代码

C

```
int removeNodeThroughContent(List *L, ElementType data) {  
    node *p = *L, *q;
```

```

int positionCurrent = 0;
while(p && p->next) {
    p = p->next;
    if(p->data == data) {
        q = p->next;
        p->next = q->next;
        free(q);
        return 0;
    }
}
return 1;
}

```

## 从i开始删除k个

原理：

▼ 代码

C

```

int deleteNodeFromItoK(sqList *List, int i, int k) {
    if(i > List->length || i+k-1 > List->length) return 1;
    int *p = List->Data+(i-1);
    for(int *q = List->Data+i+k-1; q != List->Data+List->length;
        List->length -= k;
    return 0;
}

```

## 全部删除

▼ 代码

C

```
int deleteAllNodes(sqList *List) {  
    List->length = 0;  
}
```

python

```
def deleteAllNodes(List):  
    List.length = 0;
```

修改

通过序号修改

▼ 代码

C

```
int changeValue(sqList *List, int insertPostion, ElementType value){  
    if(insertPostion > List->length) {  
        printf("insertPostion > List->length");  
        return 1;  
    }  
    ElementType *p = List->Data+(insertPostion-1);  
    *p = value;  
  
    return 0;  
}
```

## python

```
def changeValue(List, insertPostion, value):
    if insertPostion > List.length:
        print("插入位置超过表长")
        return 1
    List[insertPostion-1] = value
    return 0
```

## 通过匹配结果修改

### ▼ 代码

### C

```
int changeValue(sqList *List, ElementType value1, ElementType value2)
{
    for(int i = 0; i < List->length; i++) if(*(List->Data+i) == value1)
        *(List->Data+i) = value2;
    return 0;
}
printf("not find");
return 1;
}
```

## python

```
def changeValue(List, value1, value2):
    try:
        List.Data[List.Data.index(value1)] = value2
    except ValueError:
        print("找不到该值")
    return 1
```

## 返回信息

### 返回长度

#### ▼ 代码

C

```
int getListLength(sqList List) {  
    return List.length;  
}
```

python

```
def getListLength(List):  
    return List.length
```

## 返回某一元素的位置

#### ▼ 代码

C

```
int getThePositionOfNode(sqList List, int value) {  
    ElementType *p = List.Data;  
    for(int i = 0; i < List->length; i++) if(p[i] == value)
```

```
    return -1  
}
```

## Python

```
def getThePositionOfNode(List, int value):  
    try:  
        return List.Data.index(value)+1  
    except ValueError:  
        print("找不到该值")  
    return 1
```

## 返回某一位置元素的信息

### ▼ 代码

## C

```
int getTheContactOfNode(sqList List, int postion) {  
    ElementType *p = List.Data;  
    if(postion > List.length) {  
        return p[postion-1];  
    }  
    return NULL;  
}
```

## Python

```
def getTheContactOfNode(List, postion):  
    if postion > List.length:
```

```
        return False  
    return List.Data[postion-1]
```

## 是否是空表

### ▼ 代码

C

```
#define OK 1  
#define NO 0  
int isEmpty(sqList List) {  
    if(List.length) return OK;  
    else return NO;
```

Python

```
def isEmpty(List):  
    if len(List):  
        return 1  
    else:  
        return 0
```

## 获得最大长度

### ▼ 代码

C

```
int getMaxSize() {  
    return MAXSIZE;  
}
```

## 对内部数据进行操作

### 排序

### 逆置

#### ▼ 代码

C

```
void f(sqList *L) {  
    ElementType *p = L->Data, temp;  
    for(ElementType *q = L->Data + L->length-1; q > p; q--,  
        temp = *q;  
        *q = *p;  
        *p = temp;  
    }  
}
```

# 链表入门

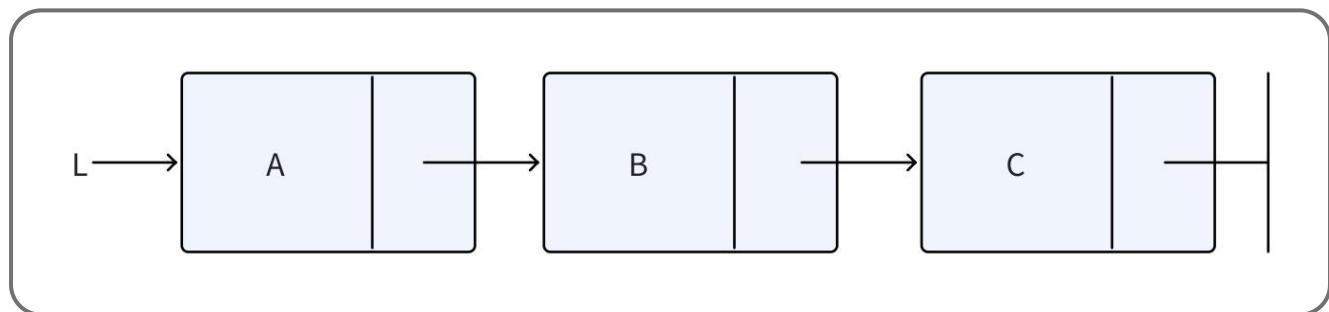
## 链表 (linked list) 的别称

线性表的链式表示、链式存储结构、链式映像、随机存取结构的储存结构

### 单向链表

长相

### 无头链表



指向首元结点的指针叫做头指针 ( head pointed )

### 有头链表

第一个结点叫头结点 ( head node )

### 结构体定义

```
typedef struct node {  
    ElementType data;  
    struct node *next;  
} node;  
typedef node* List;
```

# 初始化

```

void initList(List *L) {
    List p = *L;
    while(p && p->next) p = p->next;
    if(!p) {
        *L = (node*)malloc(sizeof(node));
        p = *L;
        scanf("%d", &p->data);
        p->next = NULL;
    } else {
        p->next = (node*)malloc(sizeof(node));
        p = p->next;
        scanf("%d", &p->data);
        p->next = NULL;
    }
}

```

# 增加

## 行尾增加

### ▼ 代码

```

int addNodeFromLast(List *L, ElementType data) {
    List p = *L;
    while(p && p->next) p = p->next;
    p->next = (node*)malloc(sizeof(node));
    p = p->next;
    p->data = data;
    p->next = NULL;
}

```

## 行首增加

## ▼ 代码

```
int addNodeFromHead(List *L, ElementType data) {
    node *p = (node*)malloc(sizeof(node));
    p->data = data;
    p->next = *L;
    *L = p;
}
```

## 中间插入

## ▼ 代码

```
int addNodeFromMiddle(List *L, ElementType data, int position) {
    node *p = *L, *q = (node*)malloc(sizeof(node));
    int positionCur = 0;
    while(position-1 > positionCur) {
        positionCur++;
        p = p->next;
        if(!p) return 0;
    }
    q->data = data;
    q->next = p->next;
    p->next = q;
    return 1;
}
```

## 有序插入

原理：找到比它大的数，插在比它大的数的前面

## ▼ 代码

```
void insertNode(List *L, ElementType v)
```

```

void insertNode(List *L, ElementType x) {
    node *p = (node*)malloc(sizeof(node)), *q = *L, *k = NULL;
    p->data = x;
    while(x > q->data && q->next) {
        k = q;
        q = q->next;
    }
    if(q != *L) {
        p->next = k->next;
        k->next = p;
    } else {
        p->next = *L;
        *L = p;
    }
}

```

## 删除

### 通过序号删除

原理：寻找——>删除

#### ▼ 代码

```

int removeNode(List *L, int position) {
    if(position < 1) return 1;
    node *p = *L, *q = *L;
    int curPosition = 1;
    while(p && curPosition < position) {
        curPosition++;
        q = p;
        p = p->next;
    }
    if(p) {
        if(p != *L) q->next = p->next;
        else *L = p->next;
    }
}

```

```

        free(p);
        return 0;
    } else return 1;
}

```

## 通过匹配结果删除

原理：寻找（需要删除的位置和需要删除的位置的前一个位置）——>删除

### ▼ 代码

```

int removeNodeThroughContent(List *L, ElementType data) {
    node *p = *L, *k = *L;
    int positionCurrent = 0;
    while(p && p->data != data) {
        k = p;
        p = p->next;
    }
    if(p) {
        if(p != *L) k->next = p->next;
        else *L = p->next;
        free(p);
        return 0;
    } else {
        k->next = NULL;
        free(p);
        return 0;
    }
    return 1;
}

```

## 全部删除

### ▼ 代码

```
int removeAllNode(List *L) {  
    node *p = *L, *k = *L;  
    while(k) {  
        k = p;  
        if(p) {  
            p = p->next;  
        }  
        free(k);  
    }  
    *L = NULL;  
    return 1;  
}
```

## 返回信息

## 返回长度

### ▼ 代码

C

```
int getListLength(sqList List) {  
    int length = 0;  
    node *p = List;  
    while(p) {  
        p = p->next;  
        length++;  
    }  
    return length;  
}
```

Python

```
def getListLength(List):
```

## 返回某一元素的位置

▼ 代码

## 返回某一位置元素的信息

▼ 代码

## 是否是空表

▼ 代码

C

```
#define OK 1
#define NO 0
int isEmpty(sqList List) {
    if(List) return OK;
    else return NO;
}
```

Python

```
def isEmpty(List):
```

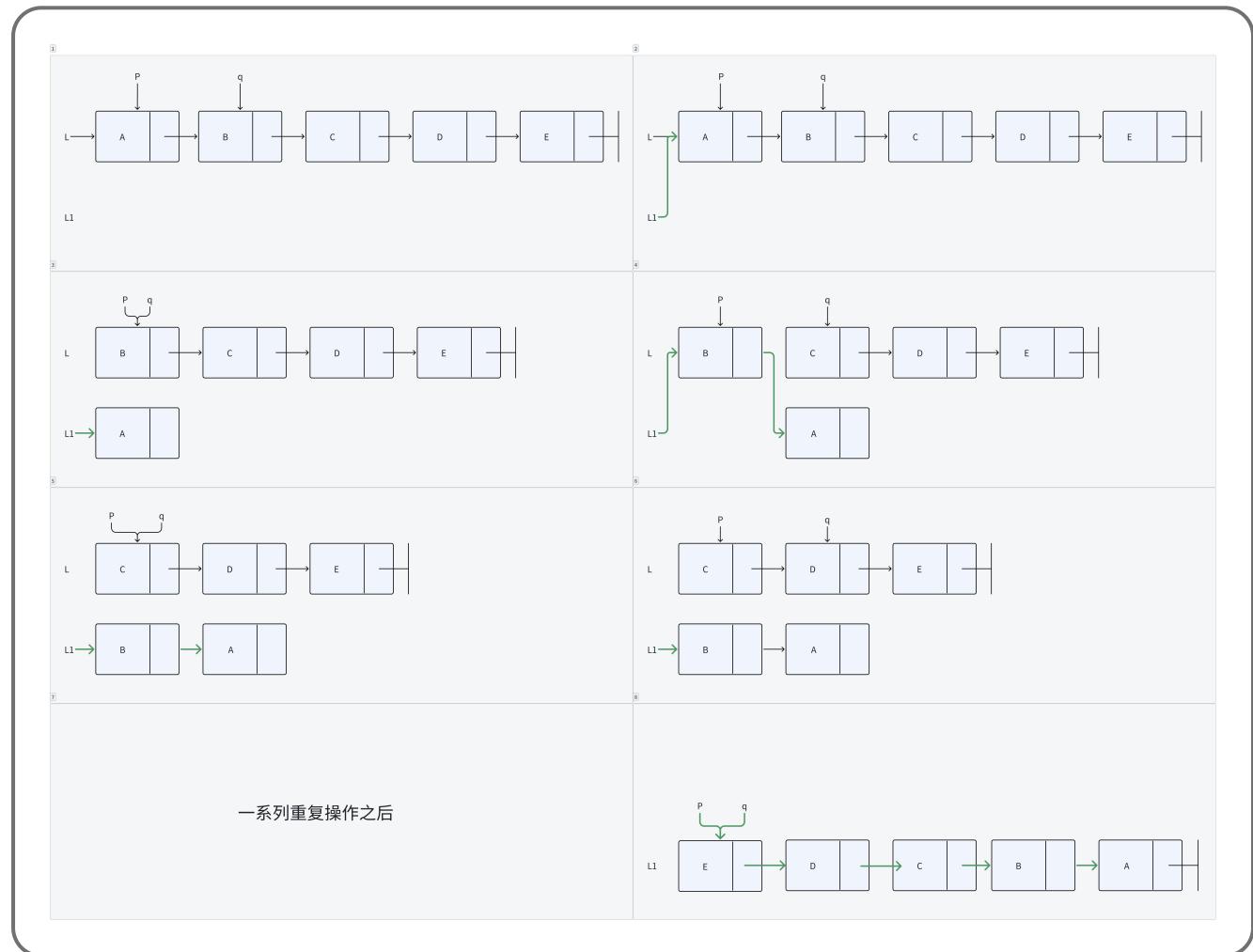
# 对内部数据进行操作

## 排序

## 逆置

## 方法一

### 原理



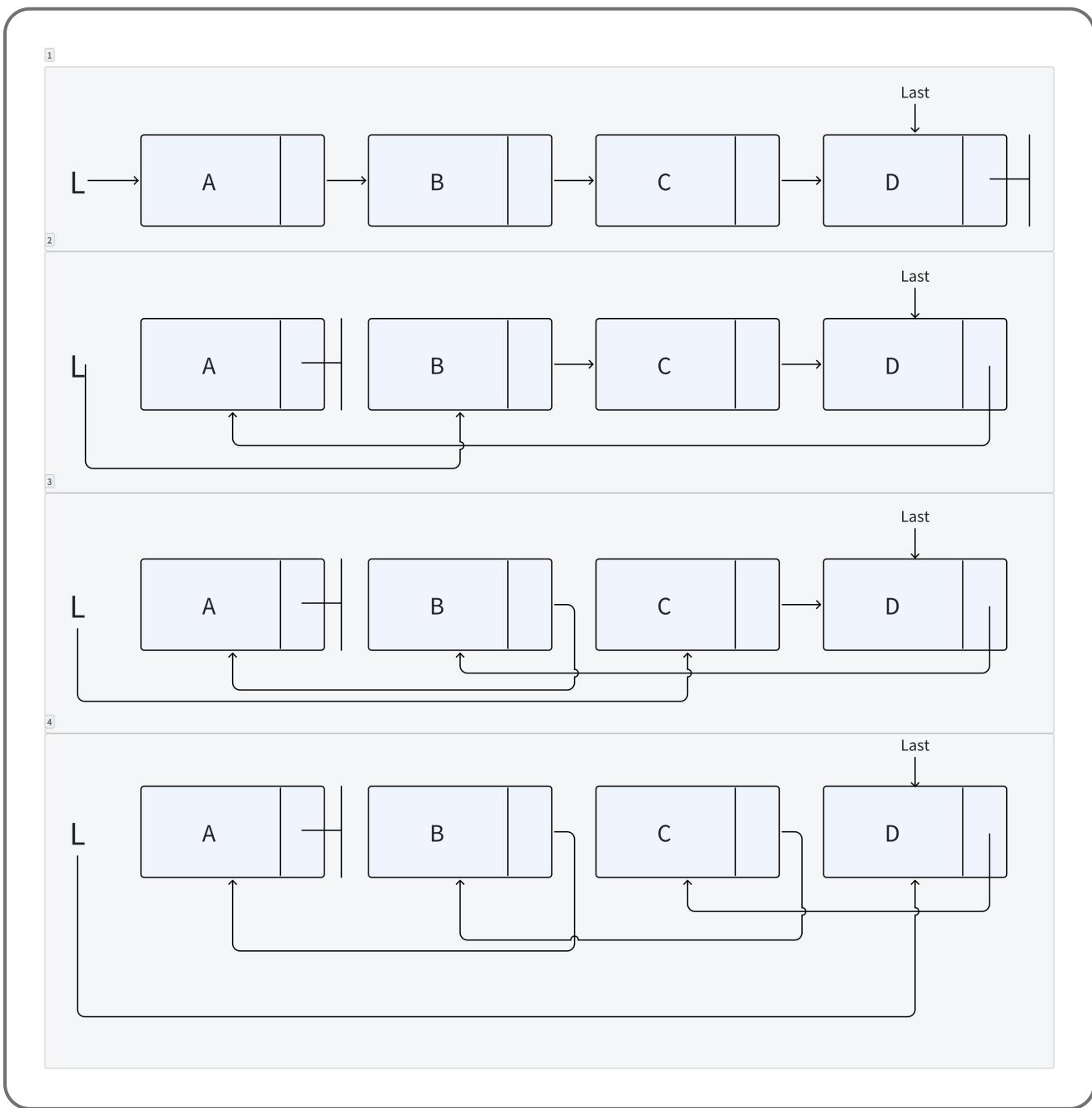
### ▼ 代码

C

```
List f(List *L) {  
    List p = *L, L1 = NULL, q = NULL;  
    while(p) {  
        q = p->next;  
        if(!L1) {  
            L1 = p;  
            L1->next = NULL;  
        } else {  
            p->next = L1;  
            L1 = p;  
        }  
        p = q;  
    }  
    return L1;  
}
```

## 方法二

### 原理



## ▼ 代码

```

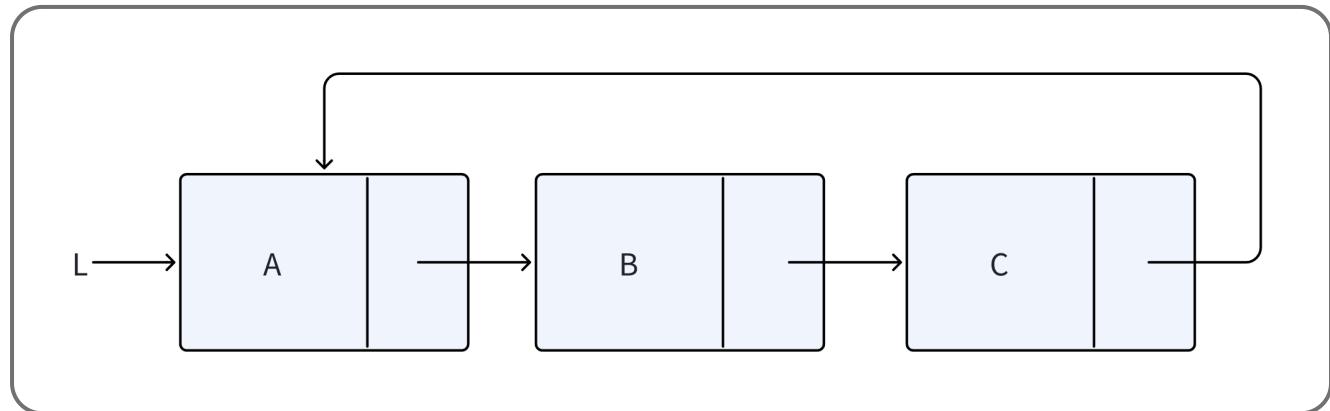
List f(List *L) {
    List p = *L, Last = *L;
    while(Last->next) Last = Last->next;
    while(p != Last) {
        *L = (*L)->next;
        p->next = Last->next;
    }
}

```

```
    Last->next = p;
    p = *L;
}
return *L;
}
```

# 循环链表

## 长相



## 结构体定义

```

typedef struct node {
    ElementType data;
    struct node *next;
} node;
typedef node* List;
  
```

## 初始化

```

void initList(List *L) {
    List p = *L;
    while(p && p->next) p = p->next;
    if(!p) {
        *L = (node*)malloc(sizeof(node));
        p = *L;
        scanf("%d", &p->data);
        p->next = NULL;
    } else {
        p->next = (node*)malloc(sizeof(node));
        p = p->next;
        scanf("%d", &p->data);
    }
}
  
```

```
    p->next = *L;  
}  
}
```

## 增加

### 行尾增加

```
int addNodeFromLast(List *L, ElementType data) {  
    List p = *L;  
    while(p && p->next != *L) p = p->next;  
    p->next = (node*)malloc(sizeof(node));  
    p = p->next;  
    p->data = data;  
    p->next = *L;  
}
```

### 行首增加

### 中间插入

### 删除

#### 通过序号删除

#### 通过匹配结果删除

## 全部删除

返回信息

返回长度

```
int getListLength(sqList List) {  
}
```

```
def getListLength(List):
```

返回某一元素的位置

返回某一位置元素的信息

是否是空表

```
#define OK 1  
#define NO 0  
int isEmpty(sqList List) {
```

```
def isEmpty(List):
```

获得最大长度

```
int getMaxSize() {  
}
```

## 对内部数据进行操作

### 排序

### 逆置

### 方法一

```
List f(List *L) {  
    List p = *L, L1 = NULL, q = NULL;  
    while(p) {  
        q = p->next;  
        if(!L1) {  
            L1 = p;  
            L1->next = NULL;  
        } else {  
            p->next = L1;  
            L1 = p;  
        }  
        p = q;  
    }  
    return L1;  
}
```

### 方法二

```
List f(List *L) {  
    List p = *L, Last = *L;  
    while(Last->next) Last = Last->next;  
    while(p != Last) {  
        *L = (*L)->next;  
        p->next = Last->next;  
        Last->next = p;  
        p = *L;  
    }  
    return *L;  
}
```

# 双向链表

## 长相

### 结构体定义

```
typedef struct node {  
    ElemtType data;  
    struct node *prior;  
    struct node *next;  
} LNode, *LList;
```

### 添加结点

#### 默认尾插法

```
void addElem(LList *L, ElemtType data) {  
    if(*L) {  
        LNode *p = (LNode*)malloc(sizeof(LNode));  
        LNode *q = *L;  
        p->data = data;  
        p->prior = NULL;  
        p->next = NULL;  
        while(q->next) q = q->next;  
        q->next = p;  
        p->prior = q;  
    } else {  
        *L = (LNode*)malloc(sizeof(LNode));  
        (*L)->next = NULL;  
        (*L)->prior = NULL;  
    }  
}
```

## 删除结点

我们选择删除后返回data域的做法

### 由位置决定

```
ElementType removeNode(LLList *L, int pos) {  
    ElementType data;  
    LNode *p = *L;  
    for(int i = 1; i < pos; i++, p++);  
    LNode *q = p->next;  
    p->next = q->next;  
    p->next->prior = p;  
    data = q->data;  
    free(q);  
    return data;  
}
```

### 由data域决定

```
void removeNode(LLList *L, ElementType data) {  
    LNode *p = *L;  
    while(p) {  
        if(p->data )
```

# 静态链表

---

# 栈

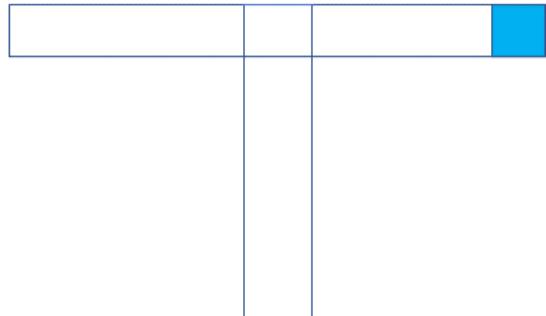
## 定义

栈 ( stack ) 是具有一定操作约束的线性表——只在一端 ( 栈顶、Top、表尾 ) 做插入、删除操作，这也就出现了 后进先出，又称LIFO ( Last In First Out ) 的情况。

一般情况下，称表尾端被称为“栈顶 ( top )”，表头端被称为“栈底 ( bottom )”。

不含元素的栈叫做“空栈”。

比如，铁路调度站



通俗来讲，栈其实是一个很简单的结构。

## 顺序表形态

## 结构体形态

```
#define MAXSIZE 100  
typedef struct {
```

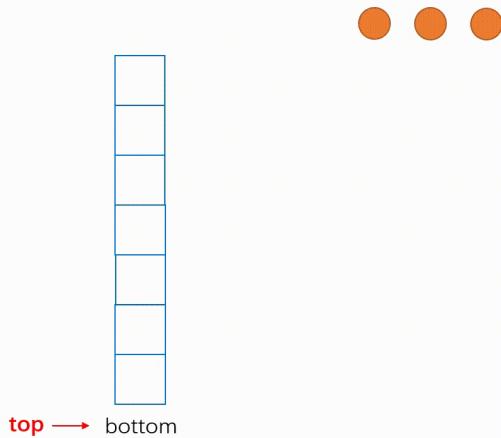
栈

```
ElemType data[MAXSIZE];  
int top;  
} Stack;
```

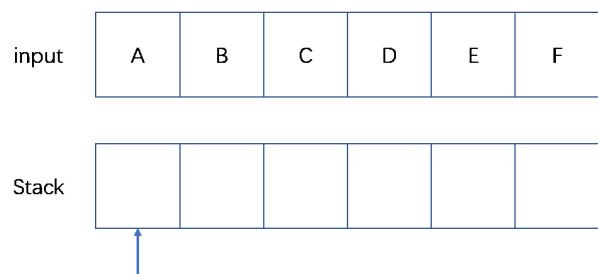
它的栈顶有初始情况：

1. Top在-1处

栈的顺序存储结构——入栈操作O(1)



2. Top在0处



# 相关操作

## 初始化栈

其实就是对top指针进行一些设置

```
void initStack(Stack *s) {  
    s->top = 0;  
}
```

或者是

```
void initStack(Stack *s) {  
    s->top = -1;  
}
```

由于我的老师在上课的时候，约定Top是0，所以，我也就设Top是从0开始吧。

## 获取栈顶

其实，一开始我是不以为意的，因为直接pop就好了嘛，但是当我开始编写[Image-Train-Tool-for-YOLO](#)的时候，我开始发现这还是很重要的。

先说一下原理吧。其实，原理很简单，就是获取[顺序表中最末尾的元素](#)

### ▼ Image-Train-Tool-for-YOLO

至于在[Image-Train-Tool-for-YOLO](#)中的那部分的代码片段。

看下面吧

先看一下，栈的类声明。

```
class Stack:  
    def __init__(self):  
        self.stack = []  
  
    def push(self, item: np.ndarray):
```

```

        self.stack.append(item)

def pop(self) -> np.ndarray:
    if len(self.stack) == 0:
        return np.array([])
    return self.stack.pop(-1)

def top(self) -> np.ndarray:
    if len(self.stack) == 0:
        return np.array([])
    return self.stack[-1]

def clear(self):
    self.stack = []

```

至于用到获取栈顶的部分是以下代码

```

elif a == ord('z') or a == ord('Z'):
    if self.workStack.stack.__len__() > 1:
        self.workStack.pop()
        image_cv2: np.ndarray = self.workStack.top().copy()
        self.saveRetangle.pop()
    else:
        print("Cannot undo.")

```

这个是用来干嘛的呢？没错就是我们最常用的Ctrl+Z。

其实，这部分完全可以不用专门获取以下栈顶的，不过，那都是后话了。

#### ▼ 实现代码

至于在C语言上，它是这样的实现的

假设 `null` 表示空。

```

ElemType getTop(Stack S) {
    ElemenType data = null;
    if(S.Top) {
        data = S.data[Top-1];
    }
}

```

```
    }
    return data;
}
```

## 弹出 ( Pop )

就是在获取栈顶的基础上，将Top-1。

### ▼ 实现代码

```
ElemType Pop(Stack S) {
    ElemType data = null;
    if(S.Top) {
        data = S.data[Top--];
    }
    return data;
}
```

## 压栈 ( Push )

由于前面说了，Top是从0开始，那就是先放东西，然后，Top++。

### ▼ 实现代码

```
void Push(Stack S, ElemType data) {
    if(S.Top <= MAXSIZE) {
        S.data[Top++] = data;
    }
}
```

## 判断是否是栈空

看Top是否为零。

▼ 实现代码

```
int IsStackEmpty(Stack S) {
    int res = True;
    if(S.Top) {
        res = False;
    }
    return res;
}
```

## 获得栈长

其实就是返回Top就可以了。

▼ 实现代码

```
int StackLength(Stack S) {
    return S.Top;
}
```

## 链式形态

这个是我较为习惯的。

在这个地方，我得拿出我的秘密武器了。

没错，[CI-CLI-Simulator](#)，我写的，就是用于模拟Linux系统的软件。

```
typedef struct treeStackFatherNode {
    treeNode *treenode;
    struct treeStackFatherNode *next;
} treeStackFatherNode;
typedef treeStackFatherNode* treeStackFather;

void treeStackPush(treeStackFather *stack, treeNode *treenode)
    if(!*stack) {
        *stack = (treeStackFatherNode*)malloc(sizeof(treeStackF
```

栈

```
(*stack)->treenode = treenode;
(*stack)->next = NULL;
} else {
    treeStackFatherNode *p = (treeStackFatherNode*)malloc(s
p->treenode = treenode;
p->next = *stack;
*stack = p;
}
}

treeNode *treeStackPop(treeStackFather *stack) {
if(!*stack) return NULL;

else {
    treeStackFatherNode *p = *stack;
    treeNode *q = p->treenode;
    (*stack) = (*stack)->next;
    free(p);
    return q;
}
}
```

这个东西主要是用于树的遍历上的，至于什么是树，后面再说吧。

不过，你注意看，这里面的栈就是用了链式的形式。

至于，如何使用其实，在链表的那一章就讲得很仔细了，不过，这里的栈与链表的有一点差别——头指针（或头结点）所指的就是栈顶。

## 定义结构体

```
typedef struct StackNode {
    ELEMType data;
    struct StackNode *next;
} StackNode;
typedef StackNode* Stack;
```

# 相关操作

## 初始化操作

### 有头结点的链表

```
void initStack(Stack *s) {  
    *s = (stackNode*)malloc(sizeof(stack));  
    s->next = NULL;  
}
```

### 没有头结点的链表（推荐使用）

```
void initStack(Stack *s) {  
    *s = NULL;  
}
```

## 获取栈顶

其实就是获取[链表中的首元结点](#)，下面是代码。  
实例代码基本上都是使用没有头结点的链表。

### ▼ 代码

```
ElemType getTop(Stack S) {  
    ElemType data = null;  
    if(S) {  
        data = S->data;  
    }  
    return data;  
}
```

## 弹出 ( Pop )

具体操作就是将首元结点从栈中取出，然后头指针前移一步。

### ▼ 代码

```
ElemType Pop(Stack *S) {
    ElemType data = null;
    if(*S) {
        data = (*S)->data;
        *S = (*S)->next;
    }
    return data;
}
```

## 压栈 ( Push )

就是将内容放入新的结点，然后用头插法放入栈中。

### ▼ 代码

```
void Push(Stack *S, ElemType data) {
    StackNode *p = (StackNode*)malloc(sizeof(StackNode));
    p->data = data;
    p->next = *S;
    *S = p;
}
```

## 判断是否是栈空

就是看头指针是否为NULL

### ▼ 代码

```
void IsStackEmpty(Stack S) {
    int res = False;
```

```
if(S) res = True;  
return res;  
}
```

## 获得栈长

遍历一次吧（其实，我纠结于一点，就是栈只能对栈顶操作，所以担心遍历会违规）

### ▼ 代码

```
void StackLength(Stack S) {  
    int length = 0;  
    StackNode *p = S;  
    while(S) {  
        length++;  
        p = p->next;  
    }  
    return length;  
}
```

# 队列

## 定义

这与在电影院，饭堂等地方排队的道理是一样的——先进先出，又称FIFO ( first in , first out )。

只能在队尾插入，队头取出，读取队头

## 普通队列

### 基于顺序表

### 结构体的定义

```
typedef struct Queue {
    ElemType data[MAXSIZE];
    int rear;
    int font;
} Queue;
```

## 入队

就是在末尾添加就好了。

### ▼ 代码

```
void EnQueue(Queue *q, ElemType data) {
    if(q.rear < MAXSIZE): q->data[q.rear++] = data;
}
```

## 出队

队列

取出下标为 `i` 的数据，然后 `i++`。

▼ 代码

```
ElementType DeQueue(Queue *q, ElementType data) {
    ElementType data = null;
    if(q.front <= q.rear) q->data[q.front++] = data;
    return data;
}
```

## 判断队伍是否为空

看看 `rear` 和 `front` 是否相等

▼ 代码

```
status isQueueEmpty(Queue q) {
    status res = False;
    if(q.rear == q.front) res = True;
    return res;
}
```

## 返回队长

这个还算简单，由于是在顺序表内，直接 `rear - front` 就好了。

▼ 代码

```
int QueueLength(Queue q) {
    return q.rear - q.front
}
```

## 清空队列

## 队列

我的想法就是将 `rear` 和 `front` 都设置为0就好了。

### ▼ 代码

```
void QueueClear(Queue *q) {  
    q->rear = q->front = 0;  
}
```

## 获取队头

就是返回`front`所指的那个元素就好了，不过，要注意这队列是否是队空。

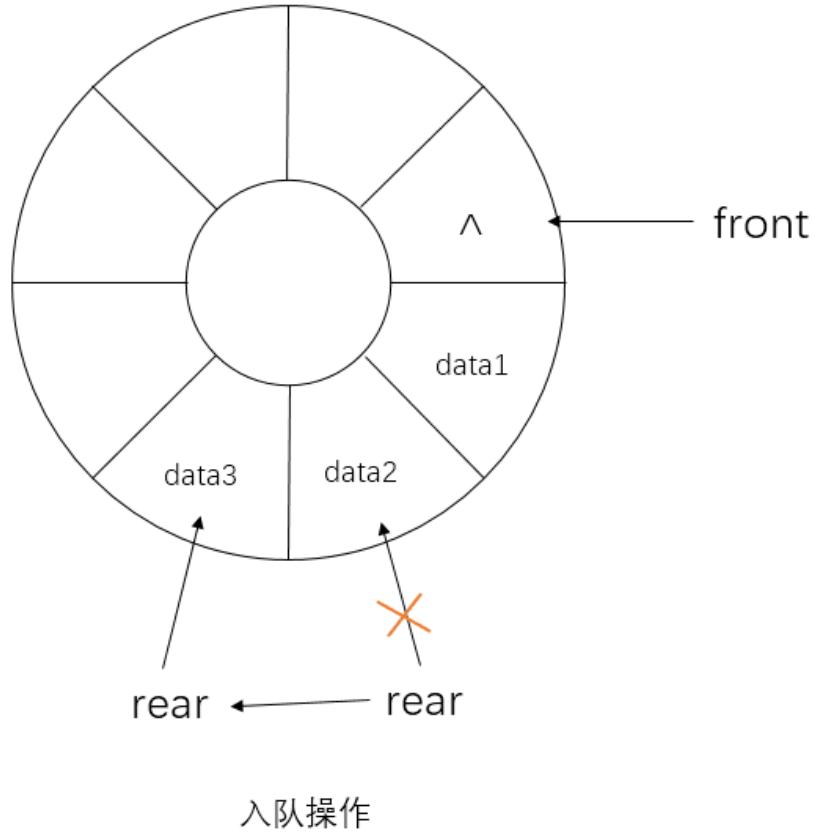
### ▼ 代码

```
ElemType GetQueueHead(Queue *q) {  
    if(!isQueueEmpty(*q)) {  
        return q->data[q->front];  
    } else {  
        return null;  
    }  
}
```

## 循环队列

要知道除了链式队列，顺序队列有一个缺陷——容易出现假队满，当`rear == front`且`rear`和`front`都等于`MAXSIZE`的时候，队列是空的，但系统判定是满的。

由此，想出了一个办法——用循环队列。



其实，有大部分的与顺序队列相似，我们说不相似的。

## 是否队满

这里有两种办法的

### 标记法

假如发生入队事件，那么flag=0，当发生出队以后，flag=1，然后当rear==front且flag==1时，队空。

当出现到达MAXSIZE的时候，只要flag=1，就可以发生循环。

### 牺牲一个空间法（推荐）

就是牺牲一个空间作为判断队列是否满了。这是由于队列的front和rear初始时都是0，在添加一个元素以后，rear+1。

说白了就是，我前进一步是否会碰到front。

看代码吧。

```
status isQueueEmpty(Queue q) {
    status res = (q.rear+1) % MAXSIZE == q.front? True: False;
    return res;
}
```

假如现在rear是下标5，front是下标6，那么，我们要添加一个元素的时候，程序会先检查是否队满，由于 $(5 + 1)$ ，此时，无法向下标为5的位置存放东西。

## 入队

在还没到达 `MAXSIZE` 之前的操作与顺序队列相同，但是到达 `MAXSIZE` 之后，就有差别了。

我们一般选择牺牲一个空间法去实现。

每次存东西之前我们要先判断是否队满（不过，话说回去，我在课堂上经常犯的错误，就是判断是否队满），当队不满的时候，放东西。

### ▼ 代码

```
void EnQueue(Queue *q, ElemtType data) {
    if(isQueueEmpty(*q)) {
        q->data[q.rear] = data;
        q.rear = (q.rear++) % MAXSIZE;
    } else printf("队满");
}
```

## 出队

主要是动front变量。

### ▼ 代码

```

ElemType EnQueue(Queue *q) {
    ElemType data = null
    if(!isQueueEmpty(*q)) {
        data = q->data[q.front];
        q.front = (q.front++) % MAXSIZE;
    } else printf("队空");
}

```

## 计算队长

就是使用公式  $(rear - front + MAXSIZE) \% MAXSIZE$ 。

假设  $rear = 5$ ,  $front = 2$ ,  $MAXSIZE = 7$ ;

那么直接数出来是2、3、4，3个，要知道  $rear=5$  的地方是空的，只有2、3、4上有东西。  
用这个公式就是  $(5 - 2 + 7) \% 7 = 3$

### ▼ 代码

```

int QueueLength(Queue q) {
    return (q.rear - q.front + MAXSIZE) % MAXSIZE;
}

```

## 基于链表

### 结构体的定义

```

typedef struct LinkQueneNode {
    ElemType data;
    struct LinkQueneNode *next;
} QueneNode;
typedef struct Quene {
    QueneNode *head;
    QueneNode *last;
} Quene;

```

## 初始化队列

```
void initQuene(Quene *q) {
    q->head = q->last = NULL;
}
```

## 入队

就是用尾插法插入

### ▼ 代码

```
void EnQueue(Queue *q, ElemtType data) {
    QueneNode *p = (QueneNode*)malloc(sizeof(QueneNode));
    p->data = data;
    p->next = NULL;
    if(!q->head) q->last = q->head = p;
    else {
        q->last->next = p;
        q->last = q->last->next;
    }
}
```

## 出队

先取出数据，然后head指针后移一步。

### ▼ 代码

```
ElemtType DeQueue(Queue *q) {
    QueueNode *p = q->head;
    ElemtType data = null;
    if(p) {
        data = p->data;
```

```

        q->head = p->next;
        free(p);
    }
    if(!q->head) q->head = q->last;
    return data;
}

```

## 返回队长

遍历一次就可以了。

### ▼ 代码

```

int QueueLength(Queue q) {
    QueueNode *p = q->head;
    int length = 0;
    while(p) {
        length++;
        p = p->next;
    }
    return length;
}

```

## 队列是否为空

看head和last是否都指向NULL。

### ▼ 代码

```

status IsQueueEmpty(Queue q) {
    status res = !q.head && !q.last? True: False;
    return res;
}

```

## 清空队列

有一个很方便的方法，就是调用DeQueue直到head和last都指向NULL。

### ▼ 代码

```
void QueueClear(Queue q) {  
    while(isQueueEmpty(q)) DeQueue(&q);  
}
```

## 获取队头

就是获取指针head所指的内容。

### ▼ 代码

```
ElemType GetHead(Queue q) {  
    ElemType data = null  
    if(q.head) data = (q.head)->data;  
    return data;  
}
```

# 串

## 定义

其实就是字符串。

有空串、空格串和非空串之分：

1. 假如没有一个字符在串当中，那就是空串
2. 假如全是空格在里面（怎么说呢，有一个空格在里面或者是有多个空格在里面），那就是空格串。

至于子串，其实类似于集合的子集。

## 串的表示

### 定长的顺序储存

```
#define MAXSIZE 256
typedef char string[MAXSIZE];
```

### 堆分配储存

```
typedef struct str {
    char *data;
    int length;
} string;
```

### 串的块链储存

```
typedef struct str {
    char data[MAXSIZE];
```

串

```
    struct str *next;
} stringBlock;
typedef struct String {
    stringBlock *head, *tail;
    int curlen;      // 串的长度
} string;
```

## 串的相关操作

### 截取

### 替换

### 模式匹配

无论是什么算法，只要是设计文本匹配的，就必须是这样的。

文本串	ABCDEACABABCDEF
模式串	ABCDEF

### 一般的模式匹配（暴力算法）

主要是用于子串在串的位置的定位。

### 基本思想

是先匹配第一个字符，当匹配成功后，再匹配第二个字符，第二个字符匹配成功后，在继续匹配下一个直至结束；

当然，也有一种情况，那就是发生匹配失败，其实，那种解决办法是很简单的，就是回到要匹配的子串的第一个字符再重新匹配。

倘若找到，会返回字串第一个字符所在的位置。

## 代码实现

```

/*
typedef struct String {
    char str[120];
    int length;
} String;
*/
int Index(String S, String T, int pos) {
    int res = 0;
    for(char *p = S.str+(pos-1), *q = T.str; *p != 0; p++) {
        if(*p == *q) {
            if(q == T.str) res = p - S.str;
            q++;
            if(*q == 0) break;
        } else q = T.str;
    }
    if(*q != 0) res = 0;
    return res;
}

```

## 时间复杂度

$$O(S.length * T.length)$$

## KMP算法

就是解决了需要匹配的子串指针回溯的问题，也就是两层循环的问题。

## 基本思想

基本思想就是跳回到之前匹配过的地方  
还是这个例子

## 串

文本串	ABCDABCABABCDAF
模式串	ABCDAF
文本串中的 ABCDA 与模式串中的 ABCDA 相对应，然后，在遇到文本串中的 B 与模式串中 的 F 不对应的时候，模式串指针跳回到模式串中 的 B 。	
但是要是发生了这种情况怎么办？	

文本串	ABCDEACABABCDEF
模式串	ABCDEF
这里要引入一个前缀表。	
用于记录前缀，每当发生无法匹配时，就去找最长相 等前后缀。	

# 总结与整理

## 线性表

线性表 ([linear list](#)) 有以下三个规则：

1. 存在唯一的一个“第一个”数据元素
2. 存在唯一的一个“最后一个”数据元素
3. 除“第一个”和“最后一个”元素均只有一个[直接前驱 \(immediate predecessor\)](#) 和一个[直接后继 \(immediate successor\)](#)。

## 动态分配

### 结构体结构

```
typedef struct {  
    ElemtType *data;  
    int length;  
    int listSize;  
} DA;
```

我来解释一下，这个结构体声明。

length是你实际存了多少的东西，listSize是你目前表的最大长度。

## 顺序表

### 别称

[线性表的顺序表示](#)、[顺序存储结构](#)、[顺序映像](#)、[随机存取结构](#)的储存结构

## 作用

利用数组的连续存储空间顺序存放线性表的各个元素

$a[n-1]$  是  $a[n]$  的直接前趋，  $a[n+1]$  是  $a[n]$  的直接后继。

## 链表

## 栈

### 定义

栈 ( stack ) 是具有一定操作约束的线性表——只在一端 ( 栈顶、Top、表尾 ) 做插入、删除操作，这也就出现了 后进先出，又称LIFO ( Last In First Out ) 的情况。

一般情况下，称表尾端被称为“栈顶 ( top )”，表头端被称为“栈底 ( bottom )”。

不含元素的栈叫做“空栈”。

## 队列

### 定义

依照先进先出 ( FIFO ( first in , first out ) 的有限制的线性表。

只能在队尾插入，队头取出，读取队头

## 形态

### 基于顺序表的直线型队列

### 基于顺序表的循环型队列

要知道除了链式队列，顺序队列有一个缺陷——容易出现假队满，当  $rear == front$  且  $rear$  和  $front$  都等于  $MAXSIZE$  的时候，队列是空的，但系统判定是满的。  
循环队列就是解决这个问题的。

既然是循环，那就会出现fron > rear的情况。

入队是这样的

```
q->data[q.rear] = data;  
q.rear = (q.rear++) % MAXSIZE;
```

至于判断队满，是用 `(q.rear+1) % MAXSIZE == q.front`。  
我们可以使用标记法或者是牺牲一个储存空间法。

## 链式队列

在链表的基础上，对指针的定义进行修改

```
typedef struct {  
    node *head;  
    node *tail;  
} Queue;
```

由于是链式，就不存在循环的情况。

## 串

就是指字符串的意思。

如果是没有字符，那就是空串。

只有空格，那就是空格串。

有三种储存方式：

1. 定长的顺序储存
2. 堆分配储存
3. 串的块链储存

# 数组

## 一般情况

在C语言中，有一个很适合表示矩阵的方法——二维数组。  
这我印象还是很深的。  
做法是这样的。

```
int a[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

## 新的数组创建方式

### 多维数组转一维数组

将一个多维数组的东西转到一维数组上是很值得考究的。  
好吧，这样子是有点抽象的。  
还是举个例子吧  
这是一个二维数组。

```
int a[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
/*
数组的样子是这样的
{ 1, 2, 3 },
{ 4, 5, 6 },
{ 7, 8, 9 }
*/
```

那如何转换呢？

其实，要说转换吧，那肯定有一个一一对应的规则。

那如何一一对应呢？

这是很简单的。

假设 `a[0][0]` (即 `1`) 是在一维数组的0号位，那8在第多少号位？

此时，有两种结果：

1. 以行为主序：7号位

2. 以列为主序：5号位

当然，这只是入门。至于进阶版，其实，与入门版大差不差，唯独差在要算的是地址。

假设 `a[0][0]` 的基地址是在100，一个元素占用`sizeof(int)`个字节，即4个字节。

还是两种结果：

1. 以行为主序： $100 + (2 * 3 + 1) * 4 = 128$

2. 以列为主序： $100 + (1 * 3 + 2) * 4 = 120$

由上面两种做法，我们可以知道，要想知道某个元素在哪，得知道两样东西，最前面的  
一个元素的地址和从他到那个元素，包括那个元素的本身之间有多少个元素，这样就可以  
计算出来了。

至于如何去算储存位置（LOC），我们要看它前面有多少行（列、面），然后再看它所  
在的那一行（列）中，它前面有多少个。

# 矩阵压缩

## 上三角矩阵

### 压缩思想

由于上三角矩阵的上半部分是0（也有可能是其他的值），我们可以将上半部分弄到数组0号位，把下半部分从一开始以行为主序存储。

当需要读取的时候，如果是下半部分，那就读取在一维数组相应的值，如果是非下半部分的，那就自动读取数组0号位。

### 实现代码

#### 我所想的

```
#define MAXSIZE 1000
void zipMatrix(intMatrix M) {
    // 假设矩阵从0开始。
    int array[MAXSIZE], pos = 1;
    array[0] = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < i; j++) {
            array[pos++] = Get(M, i, j);
        }
    }
}
```

### 对称矩阵

### 压缩思想

## 矩阵压缩

由于上三角部分与下三角部分相同，所以我们可以将下三角部分列入以为数组中。当需要读取的时候，如果是要取上三角的部分，那就自动引导到下三角部分并返回对应的值。

## 稀疏矩阵

## 压缩思想

由于这矩阵很大，但是基本上也就只有几个元素在矩阵中不为0，所以，我们可以将它压缩。

## 储存方式

### 三元组顺序表

#### 结构体声明

```
typedef struct {
    int i, j;
    ELEMType e;
} Triple;
typedef struct {
    Triple data[MAXSIZE+1];      // 非零元三元组表
    int rops[MAXRC+1]           // 各行第一个非零元素的位置表
    int mu, nu, tu;             // 矩阵行数、列数、非零元个数
} TSMartix;
```

## 十字链表

## 相关操作

# 广义表

## 大致概括

它既可以是存放一个元素，或者是一个子表。

## 基本操作

由于不算重要，暂时不记录。

## 相关操作

### 计算表长

例子	解释
A = ()	A是一个空表，它的长度为0。
B = ( e )	B是一个只有一个原子e，B的长度为1。
C = ( a, ( b, c, d ) )	C是的长度为2，两个元素分别为原子a和子表(b, c, d)
D = ( A, B, C )	D的长度为3，3个元素都是列表，即D = ( ( ), (e), ( a, (b, c, d) ) )
E = (a, E )	E的长度为2，是一个递归表

## 获取表头

### 获取第一个元素。

函数名	结果
GetHead(B)	e
GetHead(D)	A

## 广义表

函数名	结果
GetHead( (B, C) )	B

## 获取表尾

获取除第一个元素以外的元素，并将其组成列表，最终返回此列表。

函数名	结果
GetTail(B)	()
GetTail(D)	(B, C)
GetTail( (B, C) )	(C)

## 计算深度

看括号的数量，然后取最大值，最后结果+1。

# 总结与整理

---

## 数组

## 压缩矩阵

## 广义表

# 树的入门

## 概念及定义

### 树

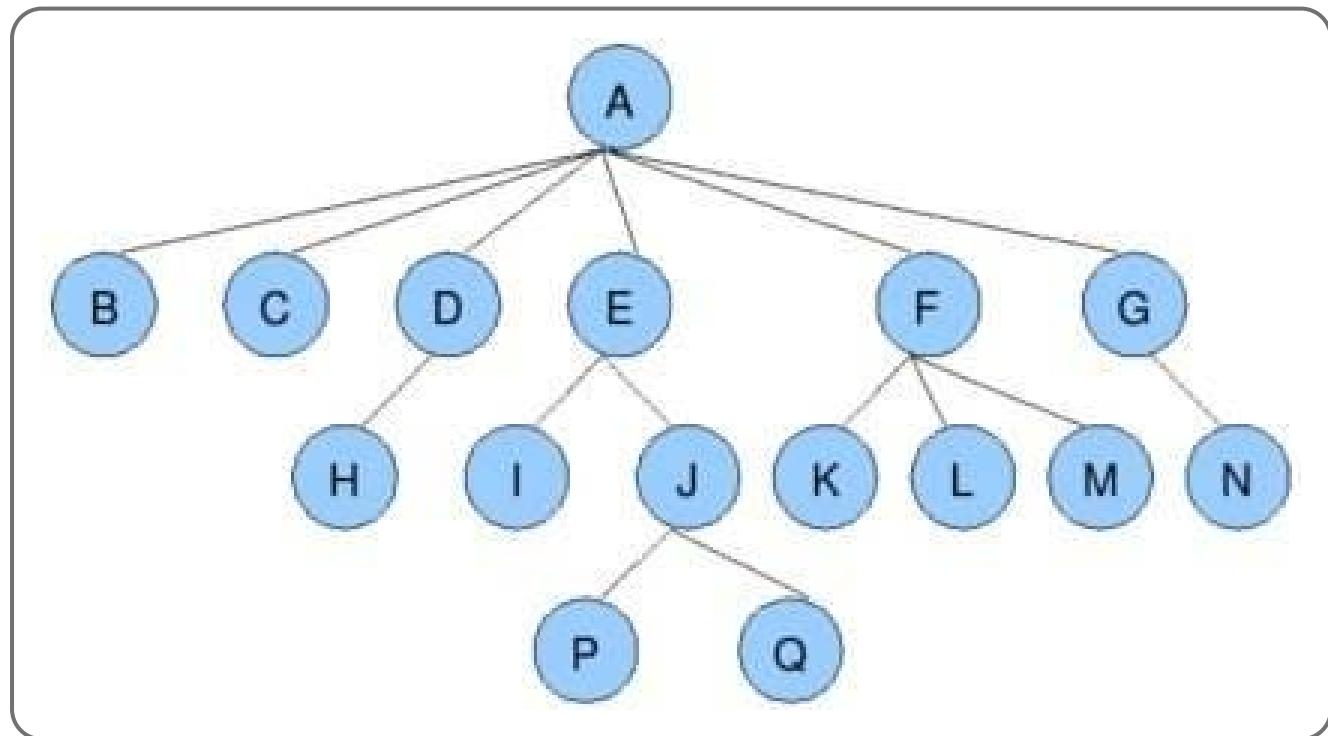
树是n个节点的有限集。

### 特征

1. 有且仅有一个为\*\*根 ( Root ) \*\*的结点。
2. 当 $n>1$ 时，其余结点可分为 $m(m>0)$ 个互不相交的有限集

$T_1, T_2, T_3, T_4, T_5, T_6, \dots, T_n$ ，由于本身又是一棵树，所以被称为子树  
( SubTree )。

具体看下图



有这样的说法，树由多个互不相交的子集构成。 $T_1 = \{D, H\}$ ,  $T_2 = \{E, I, J, P, Q\}$ ,  $T_3 = \{F, K, L, M\}$ ,  $T_4 = \{G, N\}$ ,  $T_5 = \{C\}$ ,  $T_6 = \{B\}$ ，而  $T_1, T_2, T_3, T_4, T_5, T_6$  是A的子集。在子树D、E、F、G、J中也一样适用。

## 度

就是指子树拥有的结点数。

度为0的结点称为叶子结点或终端结点，度不为0的结点称为非终端结点或分支结点。

树的度是树内各结点的度的最大值。

## 有序树和无序树

有序树是从左到右是有次序的，最左边的为第一个孩子，最右边的为最后一个孩子，并且不可互换。

无序树的概念是与有序树的概念是相反的。

## 结点的层次

以根结点为第一层，根的儿子为第二层，以此类推。

双亲在同一层的称为堂兄弟，

树中结点的最大层次称为树的深度或高度

## 定义

### 数据对象

设数据对象为D，D是具有相同特性的元素集合。

### 数据关系

### 基本操作

#### 构建空树

```
InitTree(&T);
```

#### 摧毁树

树的入门

DestroyTree(&T);

创建树

CreateTree(&T, definition);

清空树

ClearTree(&T);

判断树是否为空

TreeEmpty(T);

返回树的深度

TreeDepth(T);

返回根

Root(T);

返回某个位置的值

Value(T, cur\_e);

将某个位置的值设置为什么

Assign(T, cur\_e, value);

返回某个结点的双亲

Parent(T, cur\_e);

若cur e是T的非根结点，则返回它的双亲，否则函数值为“空”。

返回某个结点的左儿子

树的入门

LeftChild(T, cur. e);

若cur.e是T的非叶子结点，则返回它的最左孩子，否则返回“空”

返回某个结点的右儿子

Rightsibling(T, cur- e);

若cur\_e有右兄弟，则返回它的右兄弟，否则函数值为“空”。

插入c为T中p指结点的第i棵子树。

InsertChild(&T, &p, i, c);

删除T中p所指结点的第i棵子树。

DeleteChild( &T, &p, i);

按某种次序对T的每个结点调用函数visit()一次且至多一次，一旦visit()失败，则操作失败。

TraverseTree(T, Visit());

## 森林

树多了也就有了森林。其实，森林就是互不相交的树的集合。

这就有点类似Windows的文件系统，每一个分区就是一棵树，至于我是怎么知道的，别问，问就是猜的。

## 明辨

### k叉树与度为k的树的差别

1. k叉树可以是空树，可以有1个子结点，可以有\*\*i ( 1 <i=<k ) \*\*个结点。

2. 度为k的树必须要有一个子结点数为k的子树。

# 森林与树

## 树的储存结构

### 双亲表示法

有一个域存数据，一个域存双亲  
结构体声明就是

```
#define MAXSIZE 100
typedef struct PTNode {
    TElemType data;
    int parent;
} PTNode;
typedef struct {
    PTNode nodes[MAXSIZE];
    int root, length;
} PTree;
```

此时，你会直呼熟悉。

一个类似于静态链表的结点结构体，一个不只是存表长的顺序表，两者构成一个双亲表示法的储存结构。

在这种情况下，我们可以知道，找双亲容易，找儿子难。

找双亲只要遍历一遍即可，找儿子要遍历两次。

### 孩子表示法

就是一个域存数据，一个域作为链表的头指针。

结构体是这样的

```
typedef struct CTNode {
    int child;
    struct CTNode *next;
} ChildPtr;
typedef struct {
```

```

TElemType
    ChildPtr *firstChild;
} CTBox;
typedef struct {
    CTBox nodes[MAXSIZE];
    int r, length;
} CTree;

```

这很适用于找孩子，但找双亲是不合适的。

## 孩子兄弟表示法

这是我用过比较舒适的办法了，我将我的[CI-CLI-Simulator](#)当中的这部分奉献出来吧

```

typedef struct treeNode {
    char *data;
    struct treeNode *bro;
    struct treeNode *child;
} treeNode;
typedef treeNode* tree;

```

看吧，孩子兄弟表示法的储存结构是由一个数据域、两个指针域构成，其中，两个指针域分别存放第一个兄弟结点的地址和第一个儿子的地址。

所以，我上面的命名是不大标准的，应该是

```

typedef struct treeNode {
    char *data;
    struct treeNode *firstBro;
    struct treeNode *firstChild;
} treeNode;
typedef treeNode* tree;

```

## 森林与二叉树的相互转换

## 森林与树

要知道，任何的树都可以转化为二叉树，包括森林。

将第一个树的根节点作为根，其他的树的根结点作为子结点，放到树的根结点的右边。

# 二叉树的入门

## 二叉树的种类

### 一般二叉树

### 满二叉树

每一层的结点数=每层的最大结点数 ( $2^{i-1}$ )

### 完全二叉树

与满二叉树相比就是少了些最后一层的右边的结点。

## 二叉树的储存结构

### 顺序结构

主要是通过按照和拟合满二叉树的序列，然后将其存入一维数组中。  
至于在程序中的表示是这样的

```
#define maxTreeSize 100
typedef TElemType sqBiTree[maxTreeSize];
sqBiTree bt;
```

但这也有最坏的情况，那就是存下一个单支树，当然即使不是单支树，是完全二叉树也是会有浪费空间的情况。

### 链式结构

## 声明结构体

其实，怎么设计这个结构体就很明显了。

没错，就是我们要有地方存数据，存左儿子的地址，存右儿子的地址。

所以，我们可以写出这样的代码：

```
typedef struct treeNode {  
    ElemtType data;  
    struct treeNode *LChild;  
    struct treeNode *RChild;  
};  
typedef BiTNode* BiTree;
```

但是，我们知道，当指针访问到子结点的时候是没办法返回双亲结点的，除非你用了递归算法或者是栈。

那该怎么办呢？

其实，已经很明显了，就是再加一个存放双亲结点的地址。

像这样，

```
typedef struct treeNode {  
    ElemtType data;  
    struct treeNode *LChild;  
    struct treeNode *RChild;  
    struct treeNode *Parent;  
} BiTNode;  
typedef BiTNode* BiTree;
```

不过，大部分的会是选择前者。

# 树以及二叉树的相关计算

## 关于结点的计算

### 所有情况下都成立的

$$1. n = n_0 + n_1 + n_2 + \dots + n_k$$

其中， $n$  表示结点总数， $n_i$  表示度为  $i$  的结点， $k$  表示度。

$$n = n_1 + 2n_2 + \dots + in_i + kn_k + 1$$

其中， $n$  表示结点总数， $in_i$  表示分支数， $n_1 + 2n_2 + \dots + in_i + kn_k$  表示分支总数， $k$  表示度。

$$\text{所以}, n_0 + n_1 + n_2 + \dots + n_k = n_1 + 2n_2 + \dots + in_i + kn_k + 1$$

## 二叉树最大结点数计算

这点可以列一个表格去观察

层数	该层最大结点数
1	1
2	2
3	4
4	8
.....	.....

每一层比上一层多乘了一个 2，所以，可以推导出这样的结果——第  $n$  层上的最大结点数为  $2^{n-1}$ 。

## 最大结点总数的计算

树以及二叉树的相关计算

最大结点总数无非就是将各层的最大结点总数加起来就好了。

$$\sum_{i=1}^n 2^{i-1} = 2^n - 1$$

## 【推论】k叉树上的最大结点数

### ▼ 推导例子

假设 $k=3$ ，那就存在这样的树

层数	该层最大结点数
1	1
2	3
3	9
4	27
.....	.....

我是这样想的，在三叉树中一个结点最多可以生三个儿子。

第一层只能有一个结点，第二层可由第一层的结点生出3个子结点；

那第二层的最大结点数为3，第三层可由第二层的3个结点生出分别生出3个节点，所以第三层的最大结点数为——第二层的3个结点\*一个结点最多可以生3个儿子=9；

第四层的也是这样——第3层的9个结点\*一个结点最多可以生3个儿子=27个节点。

其中，我们发现每一层的最大结点数符合等比数列，所以，可以推论出3叉树的每一层最大结点数公式为  $3^{n-1}$ 。

同时，我们发现三叉树的每一层最大结点数公式  $3^{n-1}$  和二叉树的每一层最大结点数公式  $2^{n-1}$  仅仅在底数上有差别，由此，推导出在  $k$  叉树的第*i*层上至多  $k^{i-1}$  个结点。

既然推导出了第*i*层上至多的结点数，那么深度为  $i$  的  $k$  叉树最大的结点总数一样可以推出——即  $k^i - 1$ 。

## 与深度有关的

树以及二叉树的相关计算

具有  $n$  个结点的完全二叉树的深度为  $\lceil \log_2 n \rceil + 1$ 。

假设深度为  $k$  的完全二叉树，根据性质 2（深度为  $i$  的  $k$  叉树最大的结点总数为  $k^i - 1$  个结点，当然，你也可以用等比数列之和计算）完全二叉树的定义，可以得出这样的关系式：

$$2^{k-1} - 1 < n \leq 2^k - 1$$

由此，就可以得到这样的式子  $\lceil \log_2 n \rceil + 1$

上面的解释还是有些复杂了，我们可以举一个简单的例子。

层数	该层结点数
1	1
2	2
3	4

这棵树是满二叉树，树上的结点总数是7个，所以，用  $\lceil \log_2 n \rceil$  运算，就可以知道它是等于2.8074，然后，加1，就为3了，也就是整个二叉树的层数——3层了。

那如果不是满二叉树，是完全二叉树呢？一样。

再举个例子吧

层数	该层结点数
1	1
2	2
3	1

这棵完全二叉树的节点总数是4，用  $\lceil \log_2 n \rceil$  运算后，可以得到2，再加上1，就是3了，没错，这个例子与上一个例子一样还是3层的二叉树。

## 与双亲有关的

对于一棵有  $n$  个结点的完全二叉树的结点按层序编号，则有：

1. 如果  $i = 1$ ，则结点  $i$  是二叉树的根，无双亲；如果  $i > 1$ ，则其双亲的结点是在  $i/2$  位置。

树以及二叉树的相关计算

2. 如果  $2i > n$  , 则结点 i 无左儿子 ; 否则结点 i 的左儿子为  $2i$ 。
3. 如果  $2i + 1 > n$  , 则结点 i 无右儿子 , 否则他的左儿子是  $2I + 1$  。

# 二叉树的创建

## 按照数据大小创建

```
void CreateABinTree(BiTree *T, Queue *Q) {  
    while(Q->head) {  
        BiTNode *p = (BiTNode*)malloc(sizeof(BiTNode)), *q = *T;  
        p->data = DeQueue(Q);  
        p->LChild = p->RChild = NULL;  
        if (*T) {  
            while (q) {  
                t = q;  
                if (q->data > a) q = q->LChild;  
                else q = q->RChild;  
            }  
            if (t->data > a) t->LChild = p;  
            else t->RChild = p;  
        } else {  
            *T = p;  
        }  
    }  
}
```

## 按照先序遍历序列创建

### ▼ 代码

### 递归算法

```
void CreateABinTree(tree *BinTree) {  
    int data;  
    scanf("%d", &data);  
    if(data) {
```

```

    *BinTree = (treeNode*)malloc(sizeof(treeNode));
    (*BinTree)->data = data;
    CreateABinTree(&(*BinTree)->LChild);
    CreateABinTree(&(*BinTree)->RChild);
} else {
    *BinTree = NULL;
}
}

```

## 非递归算法

```

void CreateABinTree(tree *BinTree, Queue *Q) {
    direction pos = Left;
    Stack S = NULL;
    treeNode *p, *q;
    while(S || Q->head) {
        if(!*BinTree) {
            p = (treeNode*)malloc(sizeof(treeNode));
            p->data = DeQueue(Q);
            p->LChild = p->RChild = NULL;
            *BinTree = q = p;
        } else if(GetHead(*Q)) {
            p = (treeNode*)malloc(sizeof(treeNode));
            p->data = DeQueue(Q);
            p->LChild = p->RChild = NULL;
            if(pos == Left) {
                q->LChild = p;
                Push(&S, q);
            } else {
                q->RChild = p;
                pos = Left;
            }
            q = p;
        } else if(pos == Left) {
            pos = Right;
            DeQueue(Q);
        } else if(pos == Right) {

```

## 二叉树的创建

```
    DeQueue(Q);
    q = Pop(&S);
}
}
```

# 遍历二叉树

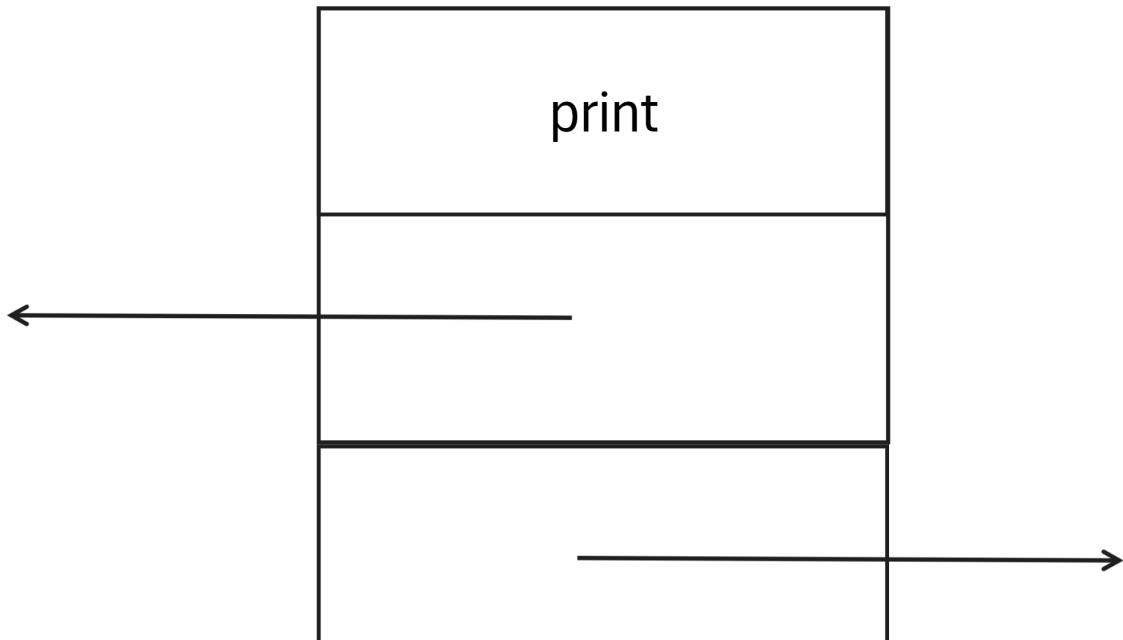
说的很高大上，其实，就是读出结点的 data 部分而已了。

这里有三种办法去读取，则三个读取的方法的命名与根结点被读取的顺序有关。

## 先序遍历

就是一见到根结点就读取，然后再读取左边的，最后读取右边的。

如下图所示



### ▼ 代码示例

## 递归算法

```
void Preorder(BiTee BT) {  
    if(BT) {  
        printf(BT->data);  
        Preorder(BT->LChild);  
        Preorder(BT->RChild);  
    }  
}
```

```

    }
}

```

## 非递归算法

参考[晓逸的博文](#)的解法

```

/* 先声明一个stack及其函数
typedef struct stackNode {
    BiTNode *data;
    struct stackNode *next;
} stackNode;
typedef stackNode* Stack;
void Push(Stack* S, BiTNode *BTNode);
BiTNode *Pop(Stack *S);
*/
void readTreeNode(BiTTree BT) {
    Stack S = NULL;
    BiTNode *p = BT;
    while (S || p != NULL) {
        if (p) {
            printf("%d ", p->data);
            Push(&S, p);
            p = p->LChild;
        } else p = Pop(&S)->RChild;
    }
}

```

## 吾解

```

/* 先声明一个stack及其函数
typedef struct stackNode {
    BiTNode *data;
    struct stackNode *next;
}

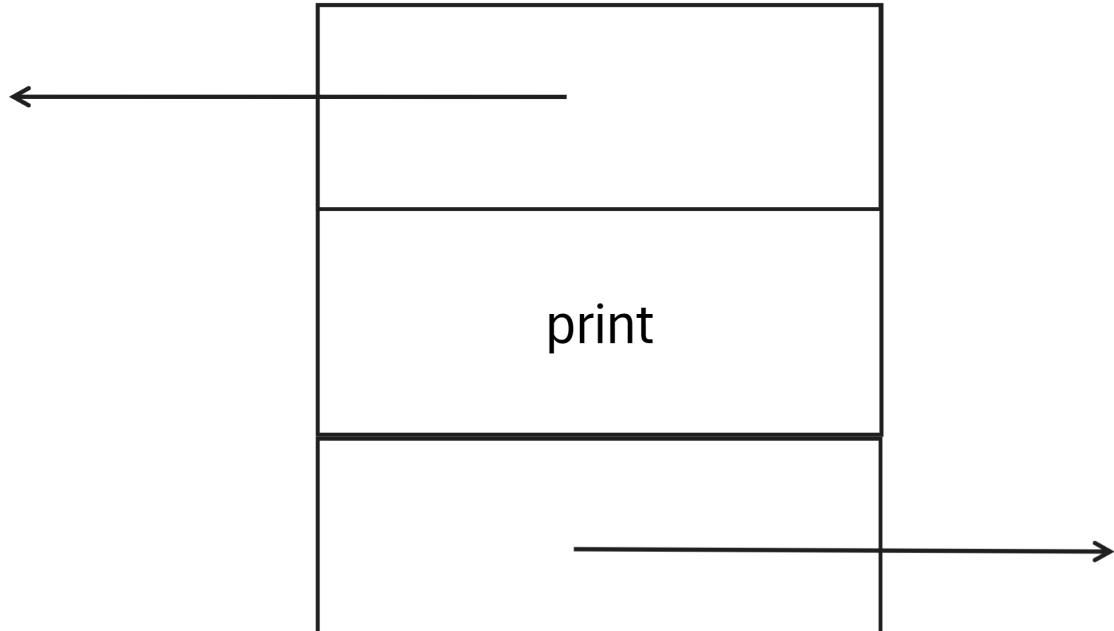
```

```
    } stackNode;
    typedef stackNode* Stack;
    void Push(Stack* S, BiTNode *BTNode);
    BiTNode *Pop(Stack *S);
}

void readTreeNode(BiTTree BT) {
    Stack S = NULL;
    BiTNode *p = BT;
    while (p) {
        printf("%d ", p->data);
        if (p->LChild) {
            Push(&S, p);
            p = p->LChild;
        } else if (p->RChild) p = p->RChild;
        else {
            while (S && !p->RChild) p = Pop(&S);
            if (p->RChild) p = p->RChild;
            else break;
        }
    }
}
```

## 中序遍历

就是先到左边走，直到左儿子为空，然后读取它，然后再读取根结点，最后向右边走，直到右儿子为空，然后读取。



▼ 代码示例

## 递归算法

```
void Preorder(BiTree BT) {  
    if(BT) {  
        Preorder(BT->LChild);  
        printf(BT->data);  
        Preorder(BT->RChild);  
    }  
}
```

## 非递归算法

书解（参考数据结构（C语言版）|作者：严蔚敏女士的）

```
/* 先声明一个stack及其函数  
typedef struct stackNode {  
    BiTNode *data;
```

```

        struct stackNode *next;
    } stackNode;
typedef stackNode* Stack;
void Push(Stack* S, BiTNode *BTNode);
BiTNode *Pop(Stack *S);
*/
void readTreeNode(BiTTree BT) {
    Stack S = NULL;
    BiTNode *p = BT;
    Push(&S, p);
    while(S) {
        while(p) {
            if(p->LChild) Push(&S, p->LChild);
            p = p->LChild;
        }
        p = Pop(&S);
        printf("%d ", p->data);
        if(p->RChild) {
            Push(&S, p->RChild);
            p = p->RChild;
        }
    }
}

```

## 吾解

```

/* 先声明一个stack及其函数
typedef struct stackNode {
    BiTNode *data;
    struct stackNode *next;
} stackNode;
typedef stackNode* Stack;
void Push(Stack* S, BiTNode *BTNode);
BiTNode *Pop(Stack *S);
*/
void readTreeNode(BiTTree BT) {
    Stack S = NULL;

```

```

BiTNode *p = BT;
while(p || S) {
    if(p) {
        Push(&S, p);
        p = p->LChild;
    } else {
        p = Pop(&S);
        printf("%d ", p->data);
        p = p->RChild;
    }
}
}

```

## ▼ 实践

在LeetCode中是有这样的题的，你可以这样

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
/**
 * Note: The returned array must be malloced, assume caller
 */

typedef struct stackNode {
    struct TreeNode* data;
    struct stackNode* next;
} stackNode;
typedef stackNode* Stack;

void Push(Stack* S, struct TreeNode* BTNode) {
    if (*S) {
        stackNode* p = (stackNode*)malloc(sizeof(stackNode))

```

```

        p->data = BTNode;
        p->next = *S;
        *S = p;
    } else {
        stackNode* p = (stackNode*)malloc(sizeof(stackNode));
        p->data = BTNode;
        p->next = NULL;
        *S = p;
    }
}

struct TreeNode* Pop(Stack* S) {
    struct TreeNode* p = (*S)->data;
    stackNode* sp = *S;
    (*S) = (*S)->next;
    free(sp);
    return p;
}

int* inorderTraversal(struct TreeNode* root, int* returnSize) {
    Stack S = NULL;
    struct TreeNode* p = root;
    int *a = (int*)malloc(sizeof(int) * 100), *q = a;
    /* 中序遍历的算法 */
    *returnSize = q - a;
    return a;
}

```

至于题目，看下面吧

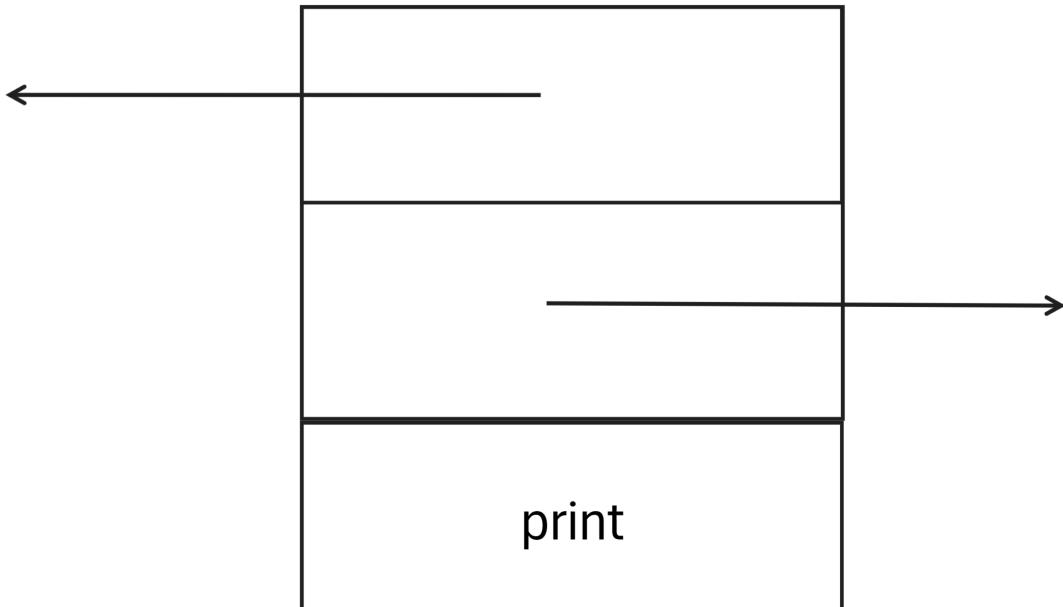


二叉树的中序遍历

## 后序遍历

## 遍历二叉树

就是先到左边走，直到左儿子为空，然后读取它，然后向右边走，直到右儿子为空，然后读取，最后再读取根结点。



### ▼ 代码示例

## 递归算法

```
void Preorder(BiTree BT) {  
    if(BT) {  
        Preorder(BT->LChild);  
        Preorder(BT->RChild);  
        printf(BT->data);  
    }  
}
```

## 非递归算法

参考晓逸的[博文](#)的解法

```
/* 先声明一个stack及其函数  
typedef struct stackNode {
```

```

BiTNode *data;
struct stackNode *next;
} stackNode;
typedef stackNode* Stack;
void Push(Stack* S, BiTNode *BTNode);
BiTNode *Pop(Stack *S);
BiTNode *GetTop(Stack S);
*/
void PostOrder(BiTTree BT) {
    Stack S
    BiTNode* t = BT;
    BiTNode* pre = NULL;
        // 记录最近一次被访问的结点
    while (t != NULL || S) {
        if (t != NULL) {
            Push(&S, t);
            t = t->LChild;
        }
        else {
            t = GetTop(S);
            // 右孩子空 或 (非空且已被访问过)
            if (t->RChild == NULL || t->RChild == pre) {
                // 出栈访问并记录
                pre = Pop(&S);
                printf("%c ", pre->data);
                t = NULL;
            }
            else {
                // 若右孩子非空且未被访问过
                t = t->RChild;
                Push(&S, t);
                t = t->LChild;
            }
        }
    }
}

```

## 我的解法

```
void PostOrder(tree BinTree) {  
    if(!BinTree) {  
        printf("这是一个空树");  
        return;  
    }  
    treeNode *p = BinTree, *pLast = NULL;  
    Stack S = NULL;  
    Push(&S, p);  
    while(p || S) {  
        if(p && (p->RChild || p->LChild)) {  
            if(p->RChild && pLast == p->RChild) {  
                pLast = p;  
                printf("%d ", pLast->data);  
                if(GetStackHead(S) == pLast) Pop(&S);  
                p = GetStackHead(S);  
                continue;  
            }  
            if(p->LChild && pLast == p->LChild) {  
                pLast = p;  
                printf("%d ", pLast->data);  
                if(GetStackHead(S) == pLast) Pop(&S);  
                p = GetStackHead(S);  
                continue;  
            }  
            if(p->RChild) Push(&S, p->RChild);  
            if(p->LChild) Push(&S, p->LChild);  
            p = p->LChild;  
        } else if(p){  
            pLast = p;  
            printf("%d ", pLast->data);  
            if(GetStackHead(S) == pLast) Pop(&S);  
            p = GetStackHead(S);  
        } else if(!p) p = GetStackHead(S);  
    }  
}
```

## ▼ 实践

在LeetCode中，还是有这样的题，我是这样写的。

```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
typedef struct StackNode {
    struct TreeNode *TreeNode;
    struct StackNode *next;
} StackNode, *Stack;

void Push(Stack *S, struct TreeNode *tNode) {
    StackNode *SN = (StackNode*)malloc(sizeof(StackNode));
    SN->TreeNode = tNode;
    if(!*S) {
        SN->next = NULL;
        (*S) = SN;
    } else {
        SN->next = *S;
        *S = SN;
    }
}

struct TreeNode *Pop(Stack *S) {
    if(*S) {
        struct TreeNode *p = (*S)->TreeNode;
        StackNode *q = *S;
        (*S) = (*S)->next;
        free(q);
        return p;
    } else return NULL;
}
```

```

struct TreeNode *GetStackHead(Stack S) {
    if(S) {
        struct TreeNode *p = S->TreeNode;
        return p;
    } else return NULL;
}

/***
 * Note: The returned array must be malloced, assume caller
 */
int* postorderTraversal(struct TreeNode* root, int* returnSize) {
    int *array = (int*)malloc(sizeof(int)*101), *q = array;
    int count = 0;
    *returnSize = count;
    struct TreeNode *p = root, *pLast = NULL;
    Stack S = NULL;
    Push(&S, p);
    if(!p) return array;
    while(p || S) {
        if(p && (p->right || p->left)) {
            if(p->right && pLast == p->right) {
                pLast = p;
                *(q++) = pLast->val;
                count++;
                if(GetStackHead(S) == pLast) Pop(&S);
                p = GetStackHead(S);
                continue;
            }
            if(p->left && pLast == p->left) {
                pLast = p;
                *(q++) = pLast->val;
                count++;
                if(GetStackHead(S) == pLast) Pop(&S);
                p = GetStackHead(S);
                continue;
            }
        }
        if(p->right) Push(&S, p->right);
        if(p->left) Push(&S, p->left);
    }
}

```

```

        p = p->left;
    } else if(p){
        pLast = p;
        *(q++) = pLast->val;
        count++;
        if(GetStackHead(S) == pLast) Pop(&S);
        p = GetStackHead(S);
    } else if(!p) p = GetStackHead(S);
}
*returnSize = count;
return array;
}

```

[题目链接在这里](#)



二叉树的后序遍历

## (重点) 由遍历序列推出原先的序列

其实，要想推出原有的序列那就必须要知道中序遍历，至于后序和先序，知道其一就可以了。因为二叉树是有序树。

其实，这也不是很困难，几个例子就可以知道了。

开始之前，我们要知道这三个口诀：

1. 前序遍历：根左右
2. 中序遍历：左根右
3. 后序遍历：左右根

ok，现在就差实战了。

现在，我们得到了两个序列：

1. 先序序列：EBADCFHGIJK
2. 中序序列：ABCDEFGHIJK

遍历二叉树

具体是这样的

列一个表格吧

先序序列	中序序列(关注的序列)	选择的元素	操作结果
E B A D C F H G I K J	A B C D E F G H I J K	E	二叉树的根结点是E
B A D C F H G I K J	(A B C D) E	B	E的左儿子是B
A D C F H G I K J	(A) B	A	A是B的左儿子
D C F H G I K J	B (C D)	D	B的右儿子是D
C F H G I K J	(C) D	C	C是D的左儿子
F H G I K J	E (F G H I J K)	F	E的右儿子是F
H G I K J	F (G H I J K)	H	H是F的右孩子
G I K J	(G) H	G	G是H的左儿子
I K J	H (I J K)	I	I就是H的右孩子
K J	I (J K)	K	K就是I的右孩子
J	(J) K	J	J就是K的左孩子
			完美结束

详细解释如下：

1. 先看前序第一个元素——它为E，所以，这颗二叉树的根结点是E。（自此，前序序列可看成B A D C F H G I K J）
2. 由此，我们可以将中序序列变成这样——(ABCD)E(F G H I J K)，E结点有左右两个儿子。
3. 先看序列ABCD，在前序序列中B是E的后继，所以，E的左儿子是B。（自此，前序序列可看成A D C F H G I K J）
4. 然后，对序列ABCD进行这样的处理——(A)B(CD)
5. 由上可知，A是B的左儿子。（自此，前序序列可看成D C F H G I K J）

6. 现在我们看B的右边部分CD，由前序序列可知，D是A的后继，所以，B的右儿子是D。（自此，前序序列可看成CFHGIKJ）
7. 看序列CD，C在D的左边，所以C是D的左儿子。
8. 至此，E左边的部分完成了，我们看E的右边部分——FGHIJK。（自此，前序序列可看成FHGIKJ）
9. 由前序序列可知，F是C的后继，所以，E的右儿子是F。（自此，前序序列可看成HGIKJ）
10. 对于中序序列，我们可以这样看了——F(GHIJK)。
11. 由于H是F的后继，再结合上一步的F(GHIJK)可以知道，H是F的右孩子。（自此，前序序列可看成GIKJ）
12. 自此，对于中序序列，我们可以这样看了——(G)H(IJK)，哦哦哦，我们一眼丁真法，G是H的左儿子。好了，现在前序序列已经可以看成IKJ。
13. I是G的后继，好，I就是H的右孩子，现在，中序序列剩下(JK)了，前序序列为KJ了。
14. 最后一击，K是I的右孩子，J是K的左孩子，哦哦哦哦哦哦哦，完结撒花。

前面的是先序遍历和中序遍历，现在我们举个后序遍历的例子。

还是两个序列：

1. 中序序列：DCBGAEHFIJK
2. 后序遍历：DCEGBFHKJIA

做法呢跟上面有一点差别，但不是很大，我们直接加速。

做法：

1. 找根：由于A是后序序列中的最后一个，所以根为A。
2. 我们括一下中序序列：(DCBGE)A(HFIJK)
3. 先看倒数第二个元素，它是I，ok，A的右儿子是I。
4. 括中序序列：(HF)I(JK)
5. J是I的前驱，所以J是I的右孩子，看，JK，好，K是J的右儿子。

6. 至于HF，看K的前驱，是H，ok，H是I的左儿子，对于HF，我们马上可以知道，F是H的右儿子。
7. 看序列DCBGE，B是F的前驱，A的左儿子是B。
8. OK，继续，括序列——(DC)B(GE)。
9. B的前驱是G，G是B的右儿子，由GE可知，E是G的右儿子。
10. 贯彻星辰的一击，C是E的前驱，C是B的左儿子，对于DC，我们可以知道——D是C的左孩子。
11. OK，完结撒花。

那假如是层序遍历和中序遍历呢？

譬如，有一颗二叉树，层序序列为：ABCDEFGHIJ，中序序列为：DBGEHJACIF。如何还原呢？

要知道层序就是一层层地去遍历，访问。

那就很简单了

还是这个表格

层序序列	中序序列（关注的序列）	选择的元素	操作结果
AABCDEFGHIJ	DBGEHJ <u>A</u> CIF	A	A为根结点
<u>B</u> ABCDEFGHIJ	(D <u>B</u> GEHJ)A(CIF)	B	B为A的左儿子
<u>C</u> ABCDEFGHIJ	A( <u>C</u> I)F	C	C为A的右儿子
<u>D</u> ABCDEFGHIJ	( <u>D</u> )B(GEHJ)	D	D为B的左儿子
<u>E</u> ABCDEFGHIJ	B( <u>G</u> EHJ)	E	E为B的右儿子
<u>F</u> ABCDEFGHIJ	C( <u>I</u> F)	F	F为C的右儿子
<u>G</u> ABCDEFGHIJ	( <u>G</u> )E(HJ)	G	G为E的左儿子
<u>H</u> ABCDEFGHIJ	(G)E( <u>H</u> J)	H	H为E的右儿子
<u>I</u> ABCDEFGHIJ	( <u>I</u> )F	I	I为F的左儿子
<u>J</u> ABCDEFGHIJ	(H) <u>J</u>	J	J为H的右儿子

## 遍历二叉树

层序序列	中序序列 (关注的序列)	选择的元素	操作结果
			完美结束

以上就是二叉树的还原的基本操作。

# 树的遍历还原

已知树的先根次序访问序列和后根次序访问序列还原二叉树

我们要知道，树的先根遍历相当于是还是二叉树的先序遍历，但是树的后根遍历排除掉根节点后却与二叉树的中序遍历相似。

同时要遵循左儿子右兄弟的规则

尽管如此，在还原上基本上是换汤不换药的。

## 例题

树的先根次序访问序列为GFKDAIEBCHJ；

树的后根次序访问序列为DIAEKFCJHBG。

先根次序	后根次序（关注的序列）	选择的元素	操作结果
GFKDAIEBCHJ	DIAEKFCJHBG	G	二叉树的根结点是G
FKDAIEBCHJ	DIAEKFCJHB	F	G的第一个儿子是F
KDAIEBCHJ	(DIAE)F(CJHB)	K	F的第一个儿子是K
DAIEBCHJ	(DIAE)K	D	D是K的第一个儿子
AIEBCHJ	D(AE)	A	A是D的兄弟
IEBCHJ	(I)A(E)	I	I是A的第一个儿子
EBCJH	A(E)	E	E是A的兄弟
BCHJ	F(CJH)B	B	B是F的兄弟
CHJ	(CJH)B	C	C是B的第一个儿子
HJ	C(JH)	H	H是C的兄弟
J	(J)H	J	J是H的第一个儿子
			完美结束

# 二叉树的计数（用代码）

## 叶子结点的计数

对于叶子结点，我们可以知道的是它没有儿子，而这就是突破口。

由于它没有儿子，所以，只要当判断到LChild和RChild。

### ▼ 代码

## 递归算法

### 第一种做法

如果我们以找叶子节点为目的，只要它是叶子结点，那么我们就使a=1，并return；如果存在右儿子或左儿子，那就向那边去走，直到找到叶子结点。

```
int countLeafNodes(BinTree T) {  
    int a = 0;  
    if(T && !T->LChild && !T->RChild) a = 1;  
    else {  
        if(T && T->LChild) a += countLeafNodes(T->LChild);  
        if(T && T->RChild) a += countLeafNodes(T->RChild);  
    }  
    return a;  
}
```

### 第二种

如果我们以顺藤摸瓜为目的，但是如果碰到叶子结点那就使a=1，并return。

```
int countLeafNodes(BinTree T) {  
    int a = 0;  
    if(T && (T->LChild || T->RChild)) {  
        if(T->LChild) a += countLeafNodes(T->LChild);  
    }
```

二叉树的计数(用代码)

```
    if(T->RChild) a += countLeafNodes(T->RChild);
} else if(T) a = 1;
return a;
}
```

## 非递归算法

```
void countLeafNodes(tree BinTree) {
    int count = 0;
    treeNode *p = BinTree;
    Stack S = NULL;
    while(p || S) {
        if(p) {
            Push(&S, p);
            p = p->LChild;
        } else if(GetStackHead(S)->RChild) p = Pop(&S)->RChild;
        else {
            if(!GetStackHead(S)->LChild && !GetStackHead(S)->RChild)
                Pop(&S);
        }
    }
    printf("%d\n", count);
}
```

## 计算总结点数

与先序遍历没有很大的差别，就是把printf()改成count++罢了，这无论是递归算法还是非递归算法都是这样。

### ▼ 代码

## 递归算法

```
int countNodes(BiTTree T) {  
    int a = 0;  
    if(T) {  
        a++;  
        a += countNodes(T->LChild);  
        a += countNodes(T->RChild);  
    }  
    return a;  
}
```

## 非递归算法

```
void countTreeNodes(tree BinTree) {  
    int count = 0;  
    treeNode *p = BinTree;  
    Stack S = NULL;  
    while(p || S) {  
        if(p) {  
            count++;  
            Push(&S, p);  
            p = p->LChild;  
        } else p = Pop(&S)->RChild;  
    }  
    printf("%d\n", count);  
}
```

# 线索二叉树

---

## 大致概括

就是将结构体中可用的指针变成连接前驱和后继的二叉树，左儿子指前驱，右儿子指后继。

# 赫夫曼树（最优二叉树）

## 前情提要

在之前，我们都是在说，如何创建树、如何遍历树，但是我们基本上没有说过结点与结点之间存在长度的概念。现在，我们来学习一下。

## 路径

就是指结点与结点之间的连线。

## 路径长度

就是指经过的分支数目或者是路径的权值。

## 树的路径长度

就是指从树根到每一个结点的路径长度之和。

## 树的带权路径长度计算

就是这条公式： $WPL = \sum_{k=1}^n w_k l_k$   
 $w_k$  表示权值， $l_k$  表示根节点到达目标节点走过的分支数。

## 赫夫曼二叉树

## 构建

### 文字描述

1. 将树的目标节点拆开成一个个的。

赫夫曼树（最优二叉树）

2. 选取权值最小的并将其建成一个二叉树，而那个新建的二叉树的根节点的权值等于两者之和。
3. 重复第二步，直到只剩下一个结点。

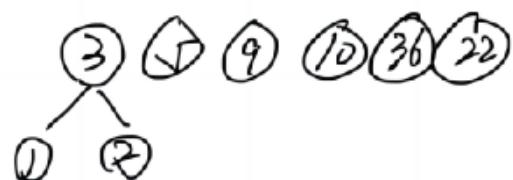
## 图解

赫夫曼树（最优二叉树）

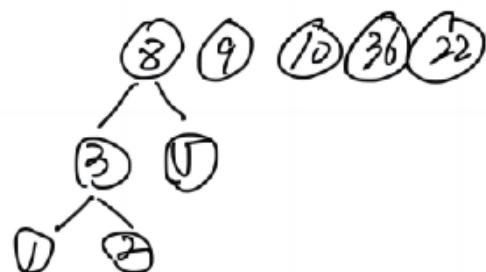
假设我们要创建1、5、9、2、10、36、22的赫夫曼树。

第一步：① ⑤ ⑨ ② ⑩ ③6 ②2

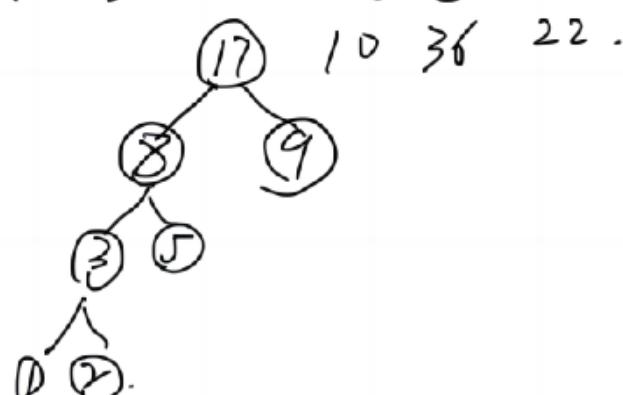
第二步：选取 ① ②.



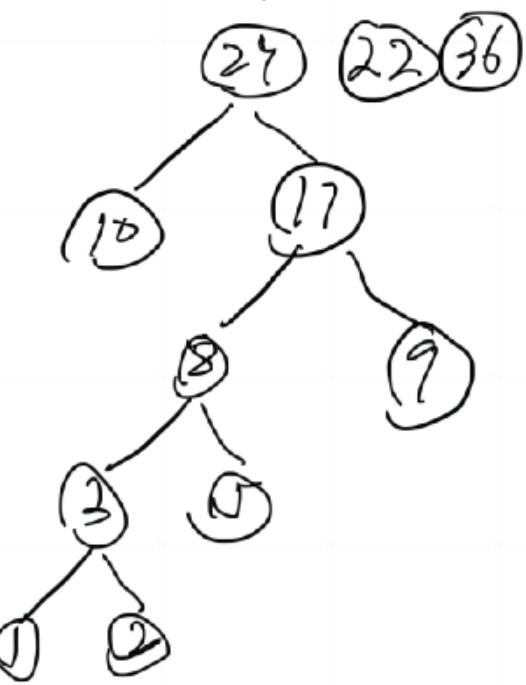
第三步：选取 ③ ⑤



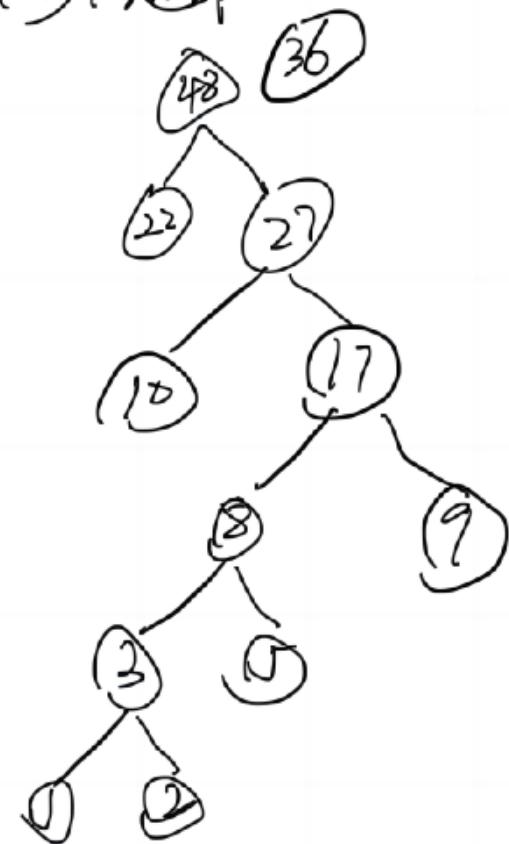
第四步：选取 ⑧ ⑨



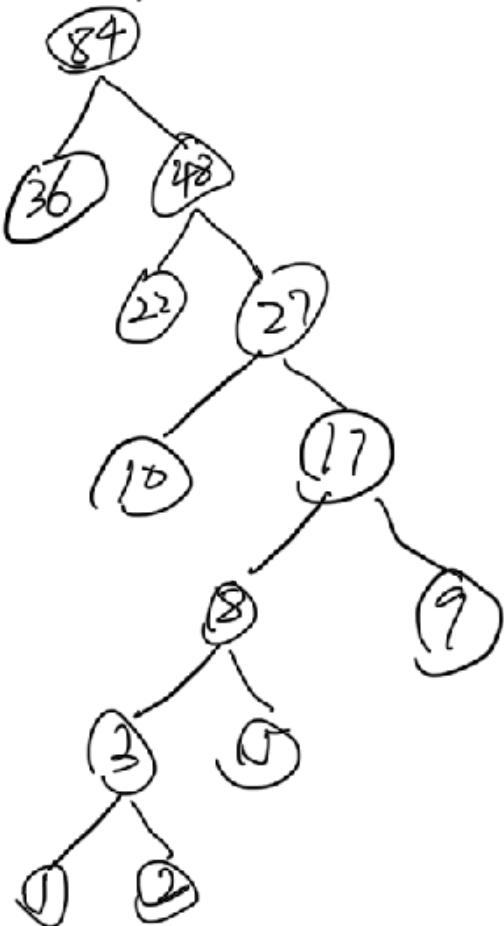
第五步：选取 10 17



第六步：选取 22 27



第七步选取 36 48



不过，无论你以任何形式创建赫夫曼树，它的WPL值是一样的。

## 赫夫曼编码

赫夫曼编码创建出来的目的就是为了减小加密点码的长度。

你想想看，如果我们加密一个内容为“AABBBC”的信息

此刻，我们知道A出现了2次，B出现了3次，C出现了1次。

假如我们要加密这样的内容，有两种方法：

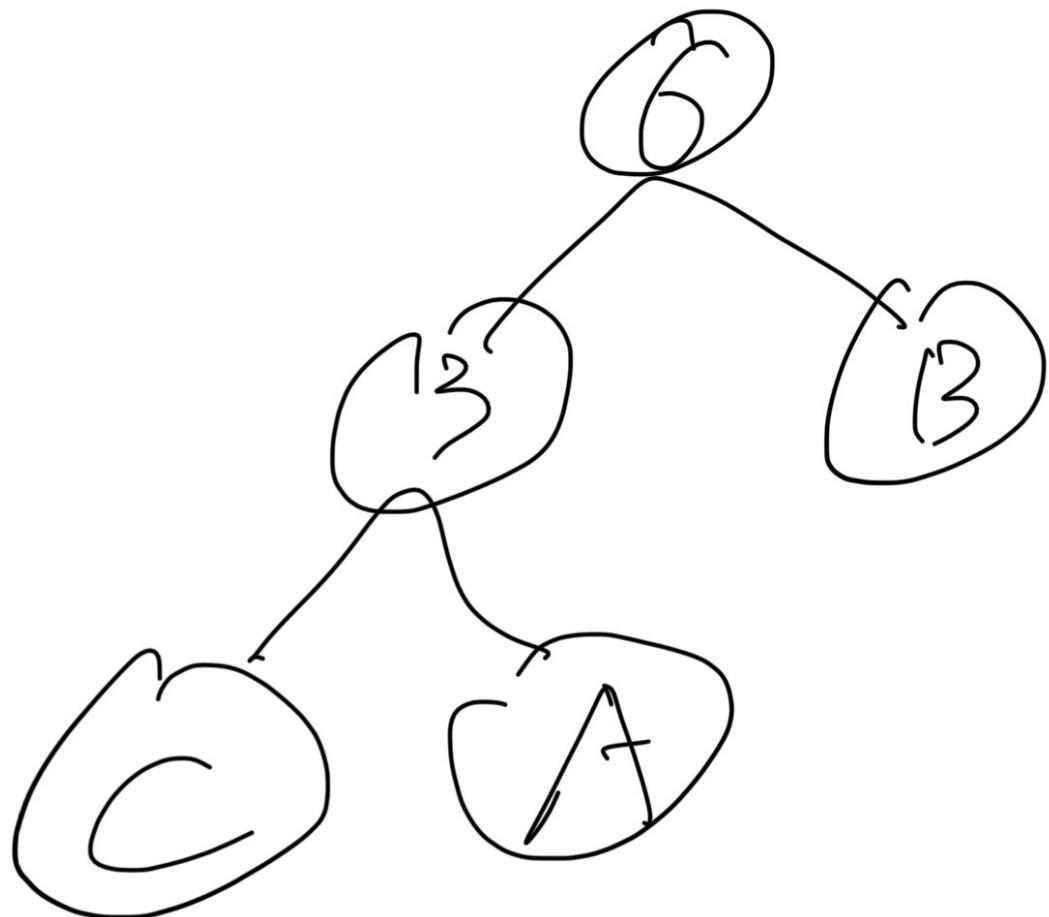
## 1. 等长的二进制序列

设编码表如下

字母	编码
A	00
B	01
C	11

那么，这段编码的序列就是 **00000101011**，长度为12。

## 2. 按照出现频率创建赫夫曼编码



那编码表就是这样的

字母	编码
C	00000
A	01010

字母	编码
A	01
B	1
C	00

那么序列就会是这样的 **010111100** , 长度为9。

那假如我们要加密内容为“I Love You” (至于为什么是这个，那是我兴头上想出来的)的时候，赫夫曼编码编码为什么没有用。因为大多数的字符都是只出现一次，所以导致了加密与没加密是没区别的。

不过，答题的时候，要注意，要把树的形态画出来，还得把赫夫曼编码写出来。

## 创建赫夫曼树

凡是说到数据结构，貌似都离不开这两个存储结构——顺序存储结构和链式存储结构。

## 顺序存储结构的实现

我们以刚才的内容为“AABBBC”的信息为例。

Content	weight	parent	LChild	RChild
A	2	3	0	0
B	3	4	0	0
C	1	3	0	0
	3	4	2	1
	6	0	1	3

### ▼ 代码

这部分代码看着很长，但是我如果强行缩减，就会导致PDF文件显示不全。

```
void generateTree(list *list
```

```

void getMin(tree L) {
    int pos[2] ={-1, -1};
    int curPos = 0, n, last;
    last = n = L->length;
    for(int k = 1; k <= n-1; k++) {
        last = L->length;
        for(int i = 0; i < 2; i++) {
            for(int j = 0; j < last; j++)
                if(L->data[j].parent == 0 && j !=pos[0])
                    curPos = j;
            for(int j = 0; j < last; j++)
                if(L->data[curPos].weight>=L->data[j].weight)
                    if(L->data[j].parent==0 && j !=pos[0])
                        curPos = j;
            pos[i] = curPos;
        }
        L->data[last].data = L->data[last].parent = 0;
        L->data[last].weight = 0;
        L->data[last].weight += L->data[pos[0]].weight;
        L->data[last].weight += L->data[pos[1]].weight;
        L->data[last].RChild = pos[1];
        L->data[last].LChild = pos[0];
        last = ++L->length;
        L->data[pos[1]].parent = L->data[pos[0]].parent=last;
    }
}

```

## 链式存储结构

# 回溯法

---

# 总结与整理

## 树

树是n个节点的有限集。

## 特征

1. 有且仅有一个为\*\*根 ( Root ) \*\*的结点。
2. 当 $n>1$ 时，其余结点可分为 $m(m>0)$ 个互不相交的有限集

$T_1, T_2, T_3, T_4, T_5, T_6, \dots, T_n$ ，由于本身又是一棵树，所以被称为子树（SubTree）。

## 度

就是指子树拥有的结点数。

度为0的结点称为叶子结点或终端结点，度不为0的结点称为非终端结点或分支结点。树的度是树内各结点的度的最大值。

## 有序树和无序树

有序树是从左到右是有次序的，最左边的为第一个孩子，最右边的为最后一个孩子，并且不可互换。

无序树的概念是与有序树的概念是相反的。

## 结点的层次

以根结点为第一层，根的儿子为第二层，以此类推。

双亲在同一层的称为堂兄弟。

树中结点的最大层次称为树的深度或高度

## 森林

树多了也就有了森林。其实，森林就是互不相交的树的集合。

## 二叉树

又分为一般二叉树、完全二叉树、满二叉树

### 一般二叉树

### 满二叉树

每一层的结点数=每层的最大结点数 ( $2^{i-1}$ )

### 完全二叉树

与满二叉树相比就是少了些最后一层的右边的结点。

### 遍历二叉树

### 由遍历序列推出原先的序列

其实，要想推出原有的序列那就必须要知道中序遍历，至于后序和先序，知道其一就可以了。因为二叉树是有序树。

其实，这也不是很困难，几个例子就可以知道了。

开始之前，我们要知道这三个口诀：

1. 前序遍历：根左右
2. 中序遍历：左根右
3. 后序遍历：左右根

ok，现在就差实战了。

现在，我们得到了两个序列：

1. 前序序列：EBADCFHGIKJ

## 2. 中序序列：ABCDEFGHIJK

具体是这样的

1. 先看前序第一个元素——它为E，所以，这颗二叉树的根结点是E。（自此，前序序列可看成BADCFHGIKJ）
2. 由此，我们可以将中序序列变成这样——(ABCD)E(FGHIJK)，E结点有左右两个儿子。
3. 先看序列ABCD，在前序序列中B是E的后继，所以，E的左儿子是B。（自此，前序序列可看成ADCFHGIKJ）
4. 然后，对序列ABCD进行这样的处理——(A)B(CD)
5. 由上可知，A是B的左儿子。（自此，前序序列可看成DCFHGIKJ）
6. 现在我们看B的右边部分CD，由前序序列可知，D是A的后继，所以，B的右儿子是D。（自此，前序序列可看成CFHGIKJ）
7. 看序列CD，C在D的左边，所以C是D的左儿子。
8. 至此，E左边的部分完成了，我们看E的右边部分——FGHIJK。（自此，前序序列可看成FHGIKJ）
9. 由前序序列可知，F是C的后继，所以，E的右儿子是F。（自此，前序序列可看成HGIKJ）
10. 对于中序序列，我们可以这样看了——F(GHIJK)。
11. 由于H是F的后继，再结合上一步的F(GHIJK)可以知道，H是F的右孩子。（自此，前序序列可看成GIKJ）
12. 自此，对于中序序列，我们可以这样看了——(G)H(IJK)，哦哦哦，我们一眼丁真法，G是H的左儿子。好了，现在前序序列已经可以看成IKJ。
13. I是G的后继，好，I就是H的右孩子，现在，中序序列剩下(JK)了，前序序列为KJ了。
14. 最后一击，K是I的右孩子，J是K的左孩子，哦哦哦哦哦哦哦，完结撒花。

前面的是先序遍历和中序遍历，现在我们举个后序遍历的例子。

还是两个序列：

1. 中序序列 : DCBGEAHFIJK

2. 后序遍历 : DCEGBFHKJIA

做法呢跟上面有一点差别，但不是很大，我们直接加速。

做法：

1. 找根：由于A是后序序列中的最后一个，所以根为A。

2. 我们括一下中序序列 : (DCBGE)A(HFIJK)

3. 先看倒数第二个元素，它是I，ok，A的右儿子是I。

4. 括中序序列 : (HF)I(JK)

5. J是I的前驱，所以J是I的右孩子，看，JK，好，K是J的右儿子。

6. 至于HF，看K的前驱，是H，ok，H是I的左儿子，对于HF，我们马上可以知道，F是H的右儿子。

7. 看序列DCBGE，B是F的前驱，A的左儿子是B。

8. OK，继续，括序列——(DC)B(GE)。

9. B的前驱是G，G是B的右儿子，由GE可知，E是G的右儿子。

10. 贯彻星辰的一击，C是E的前驱，C是B的左儿子，对于DC，我们可以知道——D是C的左孩子。

11. OK，完结撒花。

以上就是二叉树的还原的基本操作。

## 关于结点的计算

### 所有情况下都成立的

$$1. n = n_0 + n_1 + n_2 + \dots + n_k$$

其中， $n$  表示结点总数， $n_i$  表示度为*i*的结点， $k$  表示度。

$$n = n_1 + 2n_2 + \dots + in_i + kn_k + 1$$

其中， $n$  表示结点总数， $in_i$  表示分支数， $n_1 + 2n_2 + \dots + in_i + kn_k$  表示分

支总数， $k$ 表示度。

所以， $n_0 + n_1 + n_2 + \dots + n_k = n_1 + 2n_2 + \dots + i n_i + k n_k + 1$

## 二叉树最大结点数计算

第 $n$ 层上的最大结点数为  $2^{n-1}$ 。

## 最大结点总数的计算

最大结点总数无非就是将各层的最大结点总数加起来就好了。

$$\sum_{i=1}^n 2^{i-1} = 2^n - 1$$

## 【推论】 $k$ 叉树上的最大结点数

深度为  $i$  的  $k$  叉树最大的结点总数一样可以推出——即  $k^i - 1$ 。

## 与深度有关的

具有  $n$  个结点的完全二叉树的深度为  $\lceil \log_2 n \rceil + 1$ 。

假设深度为  $k$  的完全二叉树，根据性质 2（深度为  $i$  的  $k$  叉树最大的结点总数为  $k^i - 1$  个结点，当然，你也可以用等比数列之和计算）完全二叉树的定义，可以得出这样的关系式：

$$2^{k-1} - 1 < n \leq 2^k - 1$$

由此，就可以得到这样的式子  $\lceil \log_2 n \rceil + 1$

## 与双亲有关的

对于一棵有  $n$  个结点的完全二叉树的结点按层序编号，则有：

1. 如果  $i = 1$ ，则结点  $i$  是二叉树的根，无双亲；如果  $i > 1$ ，则其双亲的结点是在  $i/2$  位置。
2. 如果  $2i > n$ ，则结点  $i$  无左儿子；否则结点  $i$  的左儿子为  $2i$ 。

3. 如果  $2i + 1 > n$  , 则结点 i 无右儿子 , 否则他的左儿子是  $2I + 1$  。

## 赫夫曼树

### 树的带权路径长度计算

就是这条公式 :  $WPL = \sum_{k=1}^n w_k l_k$   
 $w_k$  表示权值 ,  $l_k$  表示根节点到达目标节点走过的分支数。

### 构建

#### 文字描述

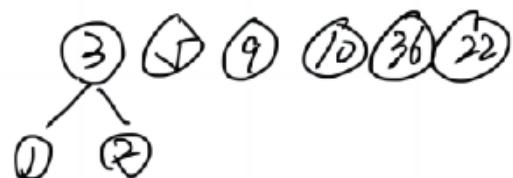
1. 将树的目标节点拆开成一个个的。
2. 选取权值最小的并将其建成一个二叉树 , 而那个新建的二叉树的根节点的权值等于两者之和。
3. 重复第二步 , 直到只剩下一个结点。

#### 图解

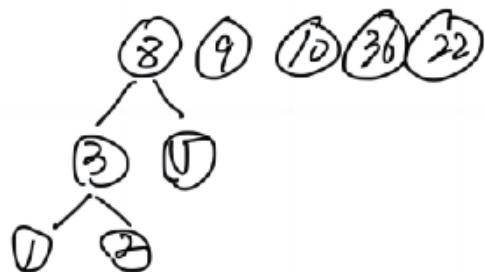
假设我们要创建1、5、9、2、10、36、22的赫夫曼树。

第一步：① ⑤ ⑨ ② ⑩ ③6 ②2

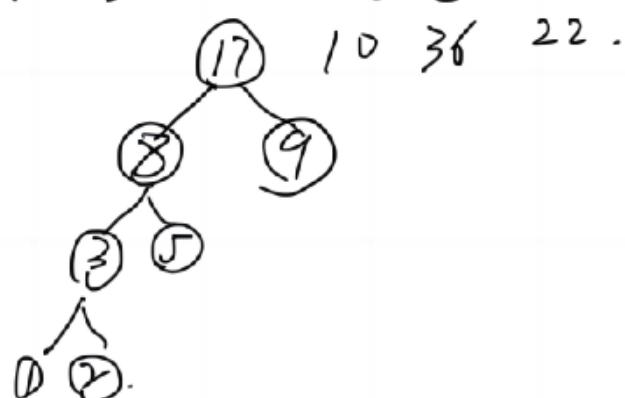
第二步：选取 ① ②.



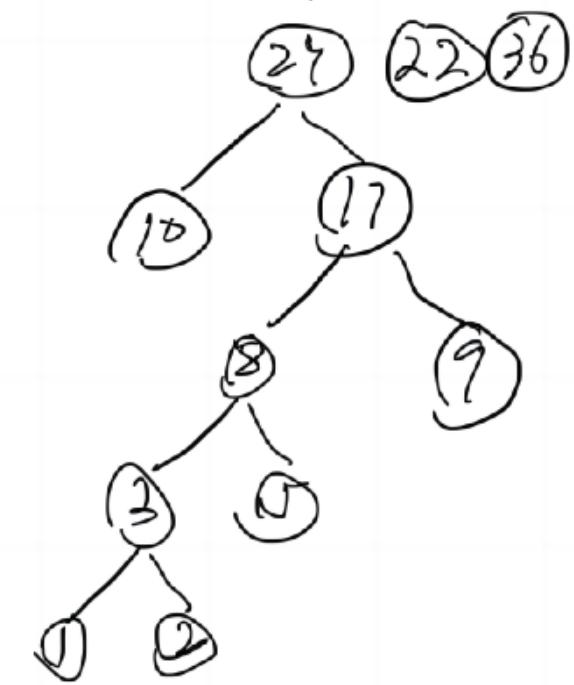
第三步：选取 ③ ⑤



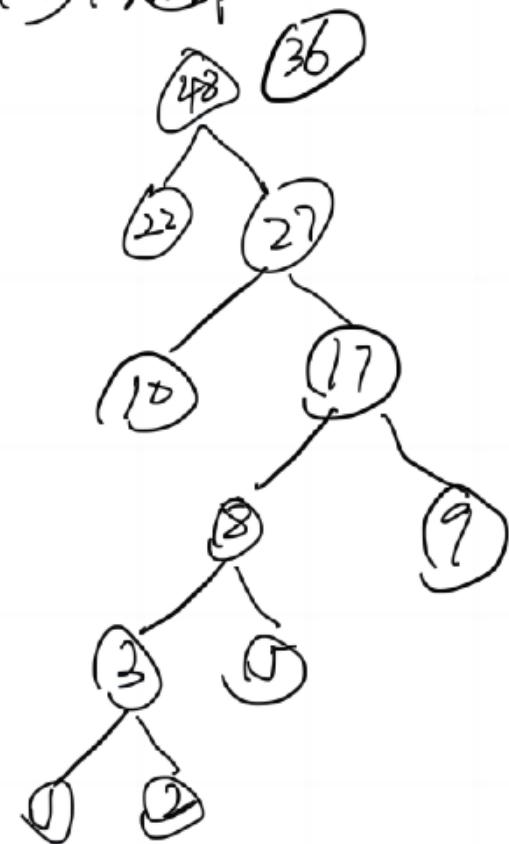
第四步：选取 ⑧ ⑨



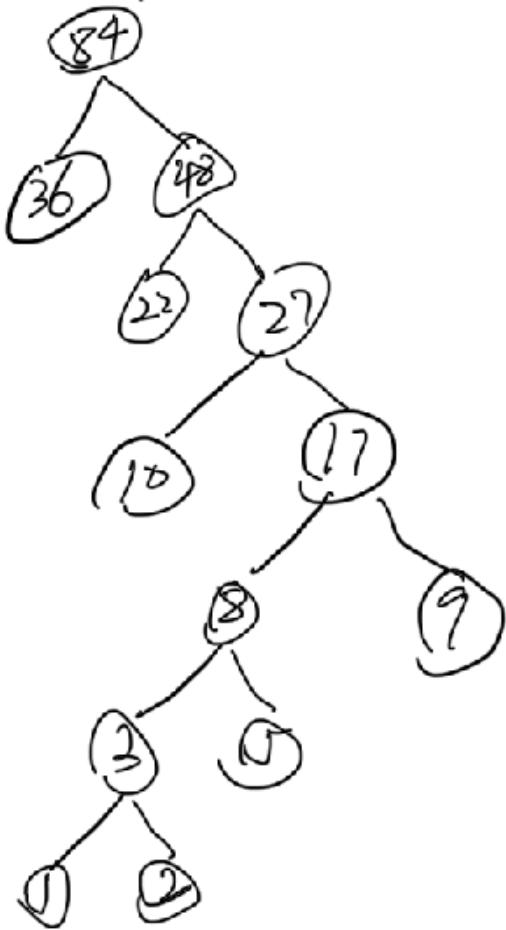
第五步：选取 10 17



第六步：选取 22 27



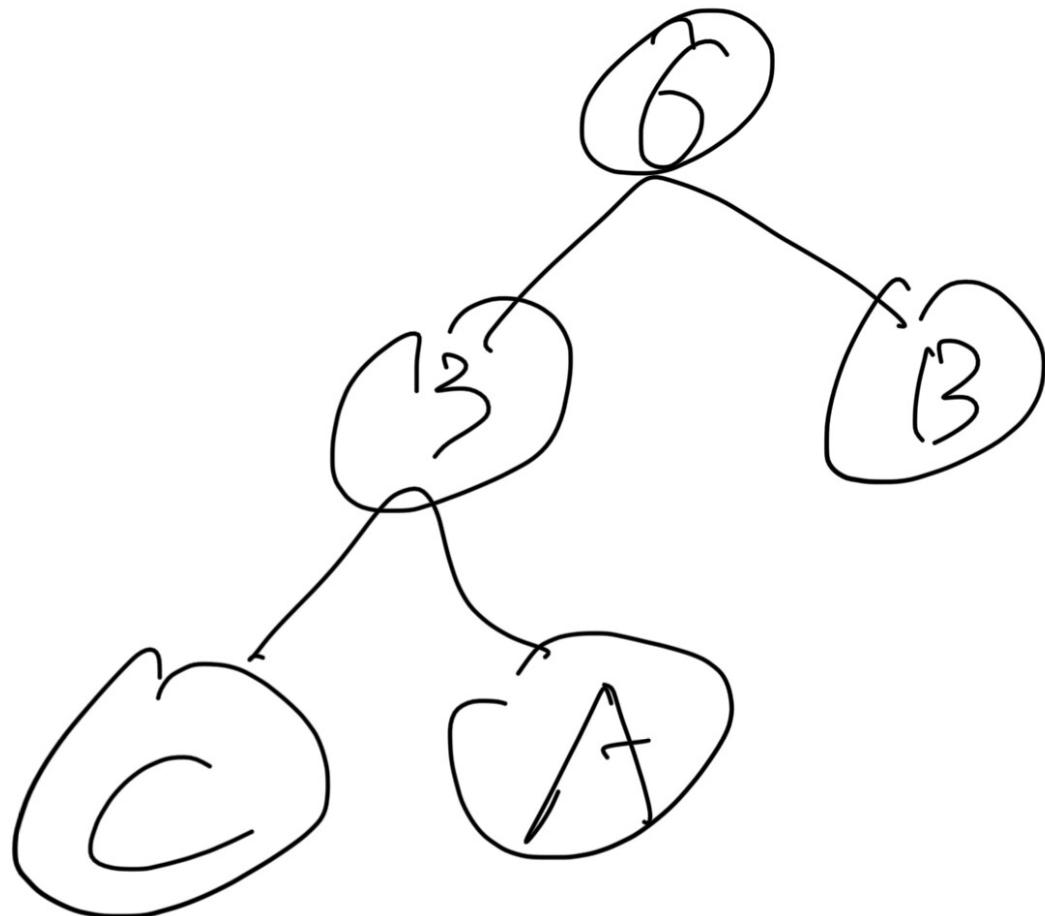
第七步选取 ③6 ④8



不过，无论你以任何形式创建赫夫曼树，它的WPL值是一样的。

## 赫夫曼编码

赫夫曼编码创建出来的目的就是为了减小加密点码的长度。  
你想想看，如果我们加密一个内容为“AABBBC”的信息



那编码表就是这样的

字母	编码
A	01
B	1
C	00

那么序列就会是这样的 **010111100** , 长度为9。

那假如我们要加密内容为“I Love You” (至于为什么是这个，那是我兴头上想出来的) 的时候，赫夫曼编码编码为什么没有用。因为大多数的字符都是只出现一次，所以导致了加密与没加密是没区别的。

# 图的入门

## 定义

数据元素在图中又叫做"顶点"。

若 $\langle v, w \rangle$ 属于VR，则 $\langle v, w \rangle$ 表示从v到w的一条弧 (Arc)，且称v为弧尾 (Tail)，

## 抽象数据类型

## 数据对象

V是具有相同属性的数据元素的集合，成为顶点集。

## 数据关系

## 数据操作

CreateGraph(&G, V, VR)

DestroyGraph(&G)

LocateVex(G, u);

初始条件：图G存在，u和G中顶点有相同特征。

操作结果：若G中存在顶点u，则返回该顶点在图中位置；否则返回其他信息。

GetVex(G, v);

初始条件：图G存在，v是G中某个顶点。

操作结果：返回v的值。

**PutVex(&G, v, value);**

初始条件：图G存在，v是G中某个顶点。

操作结果：对v赋值value

**FirstAdjVex(G, v),**

初始条件：图G存在，v是G中某个顶点。

操作结果：返回v的第一个邻接顶点。若顶点在G中没有邻接顶点，则返回“空”。

**NextAdjvex(G, v, w);**

初始条件：图G存在，v是G中某个顶点，w是v的邻接顶点。

操作结果：返回v的(相对于w的)下一个邻接顶点。若x是v的最后一个邻接点，则返回“空”。

**InsertVex( &.G, v);**

初始条件：图G存在，v和图中顶点有相同特征。

操作结果：在图G中增添新顶点v。

**DeleteVex(&.G, v):**

初始条件：图G存在，v是G中某个顶点。

操作结果：删除G中顶点v及其相关的弧。

**InsertArc(&.G, v,w):**

初始条件：图G存在，v和w是G中两个顶点。

操作结果：在G中增添弧<v,w>,若G是无向的，则还增添对称弧<w,v>。

**DeleteArc( &.G, v, w);**

初始条件：图G存在，v和w是G中两个顶点。

操作结果：在G中删除弧<v,w>,若G是无向的，则还删除对称弧<w,v>。

## DESTraverse(G, Visit()):

初始条件：图G存在，Visit是顶点的应用函数。

操作结果：对图进行深度优先遍历。在遍历过程中对每个顶点调用函数Visit一次且仅一次。一旦visit()失败，则操作失败。

## BESTraverse(G, Visit()):

初始条件：图G存在，Visit是顶点的应用函数。

操作结果：对图进行广度优先遍历。在遍历过程中对每个顶点调用函数Visit一次且仅一次。一旦visit()失败，则操作失败。

# 图的储存结构

## 数组表示法（邻接矩阵）

用两个数组保存顶点信息和数据元素之间的关系的信息

```
#define MAX_VERTEX_NUM 20
typedef enum {DG, DN, UDG, UDN} GraphKind;
// 分别是有向图、有向网、无向图、无向网。
typedef struct ArcCell {
    VRType adj;      // 表示是否相连，无权图用0或1，带权图则为权值。
    InfoType *info;   // 该弧相关的信息指针
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef struct {
    VertexType vexs[MAX_VERTEX_NUM];      // 顶点向量
    AdjMatrix arcs;        // 邻接矩阵
    int vexnum, arcnum;     // 顶点、种类数
    GraphKind kind;        // 种类的标志
} MGraph;
```

一般情况下，在邻接矩阵中的表示是  $A[i][j] = w_{ij}$  或  $\infty$

## 邻接表

它分为表结点和头结点

## 头结点

由data域（数据域）和firstarc域（链域）组成。

## 表结点

由advex域（邻接结点域）、nextarc域（链域）和info域（数据域）组成。

## 代码实现

```
#define MAX_VERTEX_NUM 20
typedef struct ArcNode {
    int adjvex;
    struct ArcNode *nextarc;
    InfoType *info;
} ArcNode;
typedef struct VNode {
    VertexType data;
    ArcNode *firsstarc;
} VNode, AdjList[MAX_VERTEX_NUM];
typedef struct {
    AdjList vertices;
    int vexnum, arcnum;
    int kind;
} ALGraph;
```

## 十字链表

上一次看到它还是在稀疏矩阵。

在这里它有两种结点组成——弧结点和顶点结点。

### 弧结点

### 顶点结点

# 图的创建

---

## 使用邻接矩阵创建

我们可以将表分为4种类型——无向图、有向图、无向网、有向网。

然后，对于无向图、有向图，可以用0，1表示，关系存在；对于无向网、有向网，可以用0、权值、 $\infty$ 表示关系。

# 图的遍历

## 广度最大优先

类似于树的层序遍历，它是逐层访问，先被扫描到的到先被访问。

其实，可以这样理解，广度更像是横着开辟的样子。

所以，在这里要用到一个队列的数据结构。

## 深度最大优先

类似于树的先序遍历。

更贴切的来说，应该是从一个点到一个点，再从这个点开始，到另一个点的遍历方法，我们可以想象成竖着挖东西的样子。

在这个方法中，假如出现到达终点的情况，那就会进行回溯，然后继续遍历，直到所有的点都被访问。

在这里我们要引入一个东西——用于记录被访问的顶点的数组。因为图可以存在多次且无规律地经过这个结点，不像树，最经典的就是二叉树，它无论是先序遍历、中序遍历，还是后序遍历都是经过3次。

# 最小生成树

## 普里姆 ( Prim ) 算法

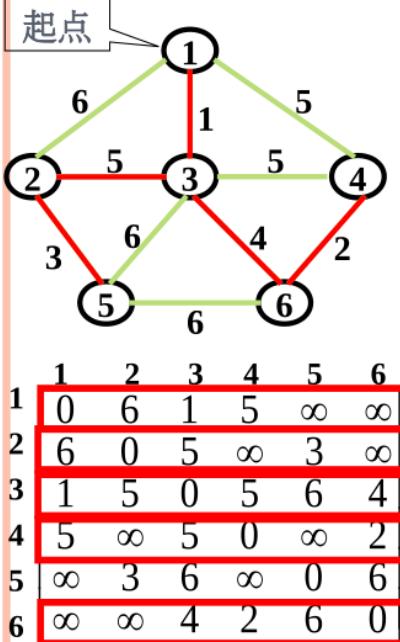
归顶定点

### 操作原理

1. 选择一个结点，将这个结点放入生成树中（初始化最小生成树），并将其 **lowcost** 置为 **0**。
2. 求这一点到未访问的、可到达的顶点的权值。若出现比之前更小的权值，则要修改，否则，不用。
3. 选一条一个顶点在树上，一个顶点在树外的边上的，并且权值最小的顶点（这包括之前的），并加入树中，并将其 **lowcost** 置为 **0**。
4. 重复2、3知道 **U=V**

### 表的写法

具体示例：



i	1(2)	2(3)	3(4)	4(5)	5(6)	U	V-U	k
closedge								
adjvex	1	1	1	1	1	{1}	{2,3,4,5,6}	2(3)
lowcost	6	1	5	$\infty$	$\infty$			
adjvex	3		1	3	3	{1,3}	{2,4,5,6}	5(6)
lowcost	5	0	5	6	4			
adjvex	3		6	3	0	{1,3,6}	{2,4,5}	3(4)
lowcost	5	0	2	6	0			
adjvex	3	0	0	3	0	{1,3,6,6,4}	{2,5}	1(2)
lowcost	5	0	0	3	0	{1,3,6,4,2}	{5}	4(5)
adjvex						{1,3,6,4,2,5}		
lowcost	0	0	0	0	0			

显然，Prim 算法的时间效率 =  $O(n^2)$ 

**U** 是选中的元素， **V-U** 是剩余未选的元素， **k** 是选中的元素， **lowcost** 是最小花费， **adjvex** 是结点的值。

## 时间复杂度

$$O(n^2)$$

## 适用情况

边比较稠密的网

## 实现代码

《数据结构（C语言版）》的

```

/*
typedef struct {
    VertexType vexs[MAX_VERTEX_NUM];
    // 顶点向量
    AdjMatrix arcs;
    // 邻接矩阵
    int vexnum, arcnum;
    // 顶点、种类数
    GraphKind kind;
    // 种类的标志
} MGraph;
struct {
    VertexType adjvex;
    VRTYPE lowcost;
} closedge[MAX_VERTEX_NUM];
*/
void MiniSpanTree_PRIM(MGraph G, VertexType u) {
    /*
        G是网
        u是某个位置的结点
    */
    k = LocateVex(G, u);
    for(int j = 0; j < G.vexnum; ++j) {
        if(j != k)
            closedge[j] = {u, G.arcs[k][j].adj};
    }
    closedge[k].lowcost = 0;
    for(int i = 1; i < G.vexnum; ++i) {
        k = minum(closedge);
        printf(closedge[k].lowcost = 0;
        for(int j = ; j < G.vexnum; ++j)
            if(G.arcs[k][j].adj < closedge[j].lowcost)
                closedge[j] = {G.vexs[k], G.arcs[k][j].adj};
    }
}

```

## 克鲁斯卡尔 ( Kruskal ) 算法

最小生成树

归并边。

## 成立条件

顶点数相同。

## 操作原理

反复选择权值最小的、且为未选择的边。

# 有向无环图

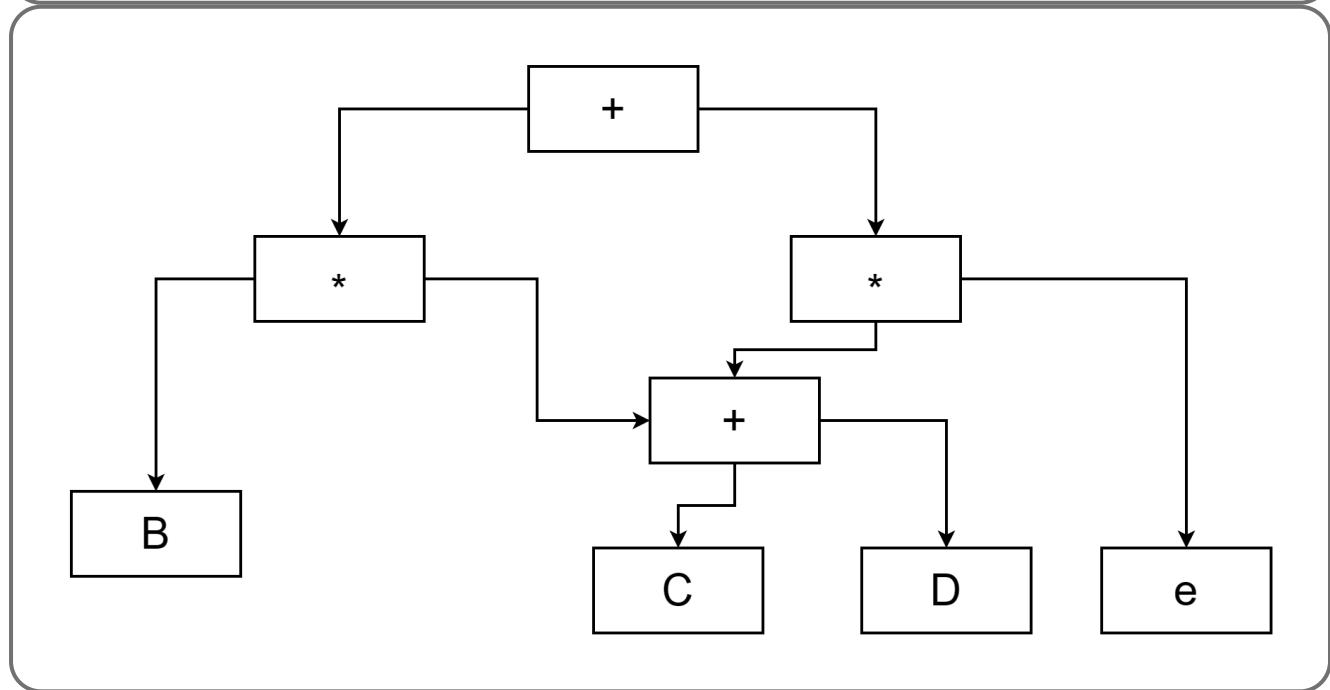
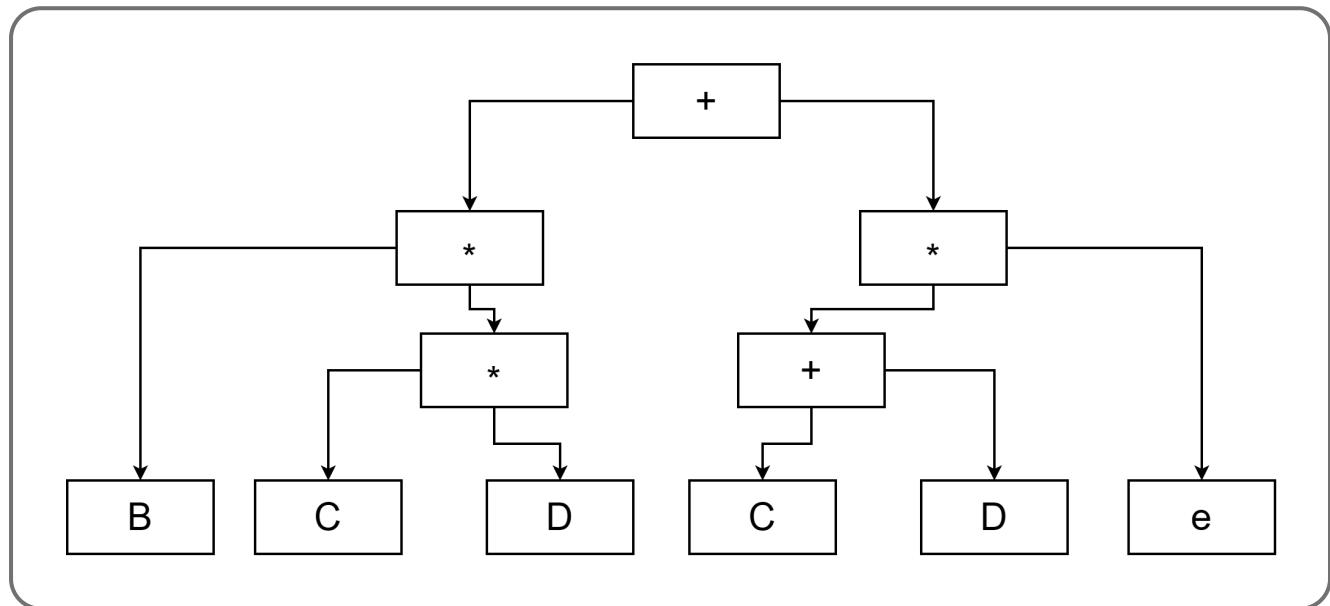
## 大致概括

有向图 + 没有环出现 = 有向无环图 ( DAG )。

## 作用

就是用于描述含有公共子式表达式。

就像这样



当然，也可以是描述一项工程和系统进行过程的有效工具。  
其原理还是一样。

# 拓扑排序与关键路径

## 前情提要

### 偏序

仅有的部分成员可以比较

### 全序

全体成员均可比较

## 拓扑有序

在偏序的基础上人为第加上一条边，使其与全序一样，这种操作叫做拓扑排序，而这个全序叫做拓扑有序。

这可以联想到Project中的前置任务。

## AOV-网

用顶点表示活动，弧表示活动的优先关系  
在AOV网中不存在有向环。

## 操作方法

就是不断地选择没有直接前驱的顶点。

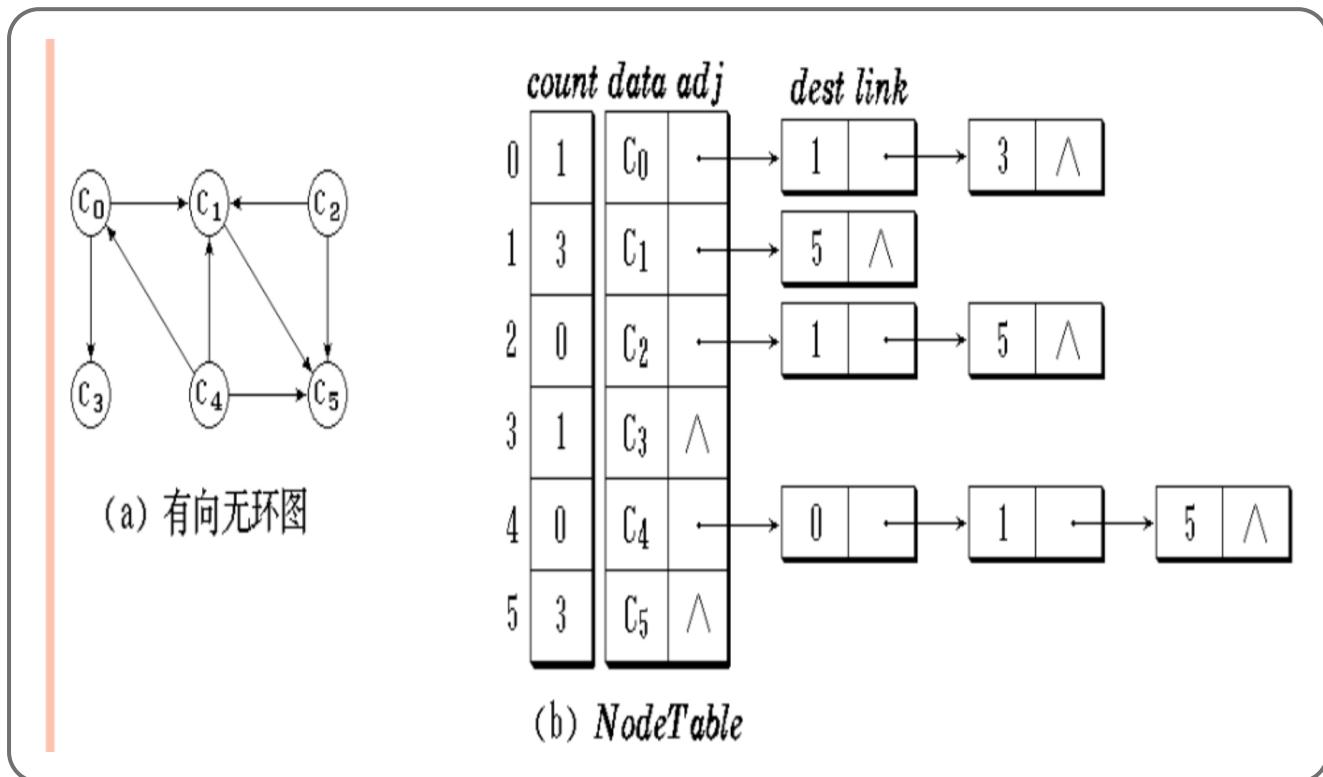
## 操作代码

## 基本原理

1. 创造一个出度表（就是邻接表），再准备一个放入度的数组

2. 输出一个没有直接前驱的结点（即入度为0的顶点），并入栈；
3. 栈不为空，出栈。
4. 由这个点弧所到的对应点的位置上，将其入度值-1。
5. 当入度值减到1的时候就可以将其压入零入度栈中。
6. 循环2、3、4、5步
7. 最后输出

图是这样的



输出的顶点个数小于总顶点个数时，就只能说明存在环了，因为只有是这样才会让其入度值一直无法为0。

## 时间复杂度

总时间是O(n+e)，建立各结点的入度为O(e)，建立0入度栈的时间复杂度为O(n)。

## AOE-网

以顶点表示时间，弧表示活动。

大多数情况下都是用于估算时间的。

# 关键路径

其实，要理解它并不难，关键路径就是最早开始时间  $e(i)$  与最晚开始时间  $l(i)$  的差值等于0的活动及其所联系的点。

先说一些概念吧。

## 一些概念

### 源点与汇点

有源点（源头点）和汇点（最终汇集的点），源点就是指入度为0的、出发的点，汇点是指出度为0的、结束的点。

### 路径长度

路径上持续的时间的总和

### 关键路径

就是指最长的路径

最早发生时间  $v_i$ 、最迟发生时间  $v_j$ 、最早开始时间  $e(i)$ 、最晚开始时间  $l(i)$

最早发生时间  $v_i$ 、最迟发生时间  $v_j$  的对象是事件，最早开始时间  $e(i)$ 、最晚开始时间  $l(i)$  的对象是活动。

前面提到AOE网中，顶点表示事件，弧表示活动。

先说事件的最早发生时间  $v_i$ 、最迟发生时间  $v_j$

$$ve(j) = \text{Max}\{ ve(i) + dut(< i, j >) \}$$

一般来说，就是源点的最早发生时间+权值=汇点的最早发生时间，但为什么会有个Max呢？因为当出现多个源点指向一个汇点的时候，就要取最大值。

对了，一般情况下，  $ve(0) = 0$

$$vl(j) = \min\{vl(i) - dut(<i,j>)\}$$

这条公式与上面的是一样的说法，一般情况下，就是汇点的最迟发生时间-权值=源点的最迟发生时间。Min是用于解决“一个源点指向多个汇点”的情况。

下面是最早开始时间  $e(i)$ 、最晚开始时间  $l(i)$  的计算公式：

$$\begin{aligned} e(i) &= ve(j) \\ l(i) &= vl(k) - dut(<j,k>) \end{aligned}$$

$ve(j)$  是指这条弧上源点的最早开始时间。

$vl(k)$ 是指这条弧上的源点的最迟发生时间， $dut(<j,k>)$ 是指这条弧的权值。

# 最短路径

## dijkstra算法

### 算法思想

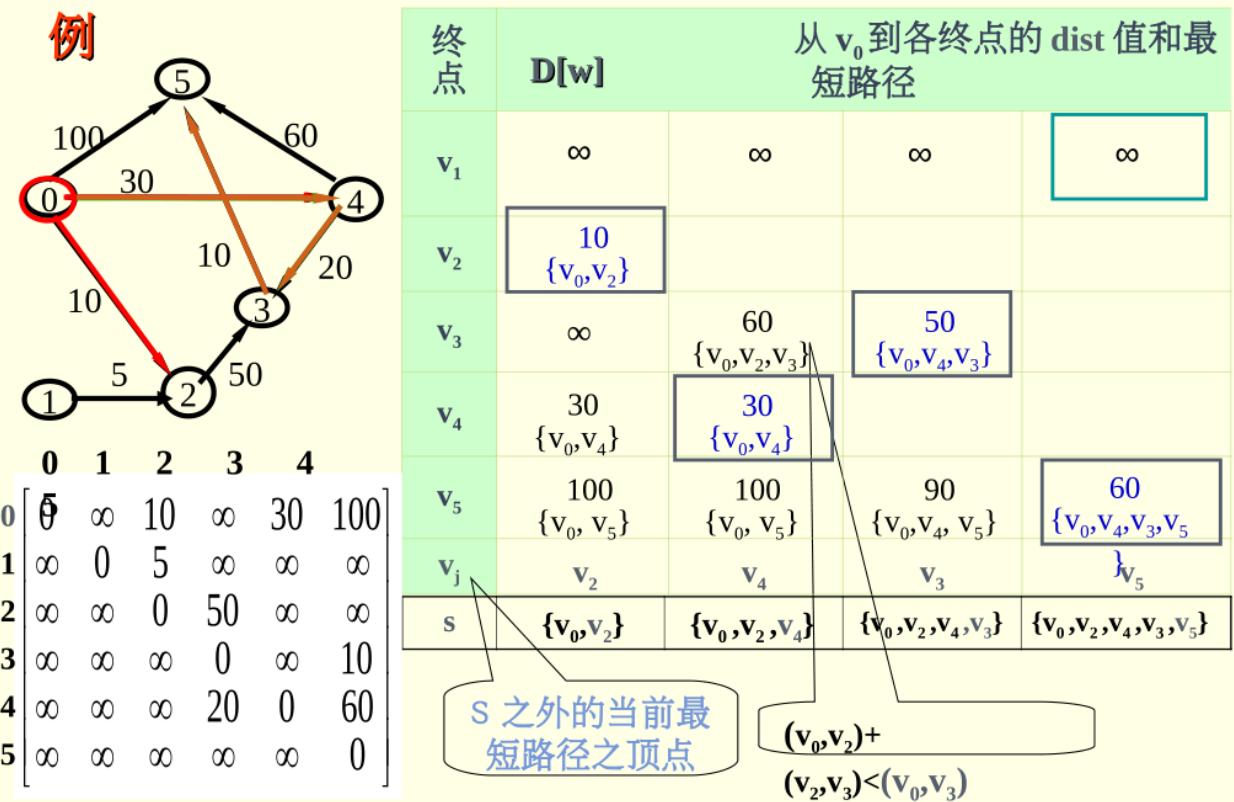
1. 找出从源点 $v_0$ 到各终点 $v_k$ 的直达路径 ( $v_0, v_k$ ) , 即通过一条弧到达的路径。
  2. 从这些路径中找出一条长度最短的路径 ( $v_0, u$ ) , 然后对其余各条路径进行适当调整  
——若在图中存在弧 ( $u, v_k$ ) , 且  $(v_0, u) + (u, v_k) < (v_0, v_k)$  , 则以路径  
( $v_0, u, v_k$ ) 代替 ( $v_0, v_k$ ) 。 ——其实质就是只要有更短的路存在 , 就可以用更短的路  
代替掉之前的路。
  3. 在调整后的各条路径中 , 再找长度最短的路径
- 循环1、2、3步 $n-1$ 次。

### 编程思想

1. 初始化
  1. 将源点 $v_0$ 加到 $S$ 中 , 即 $S[v_0] = \text{true}$  ; (  $v_0$ 一般默认是第一个元素 )
  2. 将 $v_0$ 到各个终点的最短路径长度初始化为权值 , 即 $D[i] = G.\text{arcs}[v_0][v_i]$  ,  
( $v_i \in V - S$ ) ;
  3. 如果 $v_0$ 和顶点 $v_i$ 之间有弧 , 则将 $v_i$ 的前驱置为 $v_0$  , 即 $\text{Path}[i] = v_0$  , 否则  
 $\text{Path}[i] = -1$ 。
2. 选择下一条最短路径的终点 $v_k$  , 使得 :  $D[k] = \text{Min}\{D[i] \mid v_i \in V - S\}$
3. 将 $v_k$ 加到 $S$ 中 , 即 $S[v_k] = \text{true}$ 。
4. 更新从 $v_0$ 出发到集合 $V - S$ 上任一顶点的最短路径的长度 , 同时更改 $v_i$ 的前驱为 $v_k$ 。  
若  $D[k] + G.\text{arcs}[k][i] < D[i]$  , 则  $D[i] = D[k] + G.\text{arcs}[k][i]$ ;  $\text{Path}[i] = k$  ;
5. 重复2 ~ 4  $n - 1$ 次 , 即可按照路径长度的递增顺序 , 逐个求得从 $v_0$ 到图上其余各顶点的最短路径。

注意是n-1次，因为v<sub>0</sub>最早已经被放入了。

## 相关图表



# 总结与整理

---

## 一些想法

---

其实，我想把这个部分给合并到相应的地方，但是，后来我想了一下，还是分开写吧。

# 查找表及顺序查找

## 前情提要

### 关键字

其实就是数据元素的某个数据项的值，起作用是用于识别一个数据元素。

当然对于关键字，也分主关键字和次关键字。

譬如，我们一般会把学号、准考证号等作为关键字。

### 查找

当给定的值与关键字相等时，说明存在这样的记录，就返回整个记录的信息，否则就是查找失败，会返回空值。

### 查找表

### 定义

查找表就是由同一类型的数据元素构成的集合，也是一种非常灵活的数据结构。

### 操作

1. 查找特定的元素是否存在
2. 检索各个元素特有的属性
3. 插入元素
4. 删去某个元素

### 数据元素的结构体声明

```
#define KeyType int
typedef struct {
    KeyType key;      // 关键字域
    .....           // 其他域
} SElemType;
```

## 静态查找表

就是执行“查找特定的元素是否存在”和“检索各个元素特有的属性”的表。

说起静态查找表的结构体，绝对会让你很熟悉，其实，这也是我为什么想要将其放在线性表那个地方。

它有一个特别的东西——哨兵，这个东西是被设置在 0 号位上的东西，当碰到它的时候，就说明不存在该记录。

## 动态查找表

就是在静态查找表的基础上，会进行“插入元素”和“删去某个元素”的操作的表。

## 性能分析

这里我们用到“平均查找长度”这个东西。

对于含有n个记录的表，查找成功时的平均查找长度为

$$ASL = \sum_{i=1}^n P_i C_i$$

在这里  $P_i$  表示查找表的第*i*个记录，且它的总和等于1。假如每次的概率相等，那就是  $P = \frac{1}{n}$ 。

$C_i$  表示找到这个记录所要进行比较的次数，通常是  $n - i + 1$ 。

## 顺序查找

## 结构体声明

```
typedef struct {
    SElemType *elem;
    int length;
} SSTable;
```

## 查找位置的代码

```
int SearchSeq(SSTable ST, KeyType key) {
    ST.elem[0].key = key;
    for(int i = ST.length; ST.elem != key; i--);
    return i;
}
```

## 顺序表的平均查找长度

其实就是

$$\begin{aligned} ASL_{SS} &= \frac{1}{n} \sum_{i=1}^n (n - i + 1) \\ &= \frac{n+1}{2} \end{aligned}$$

有点类似于赫夫曼树的感觉，不过，哈夫曼树是权值根节点到达目标节点走过的分支数。

# 折半查找及查找树表

将有序表、折半查找和二叉查找树放在一起会比较合适。

因为折半查找要在有序且为顺序表的情况下才可以使用，而查找树表上折半查找的延伸。

## 有序表

就是将记录排好顺序的顺序表。

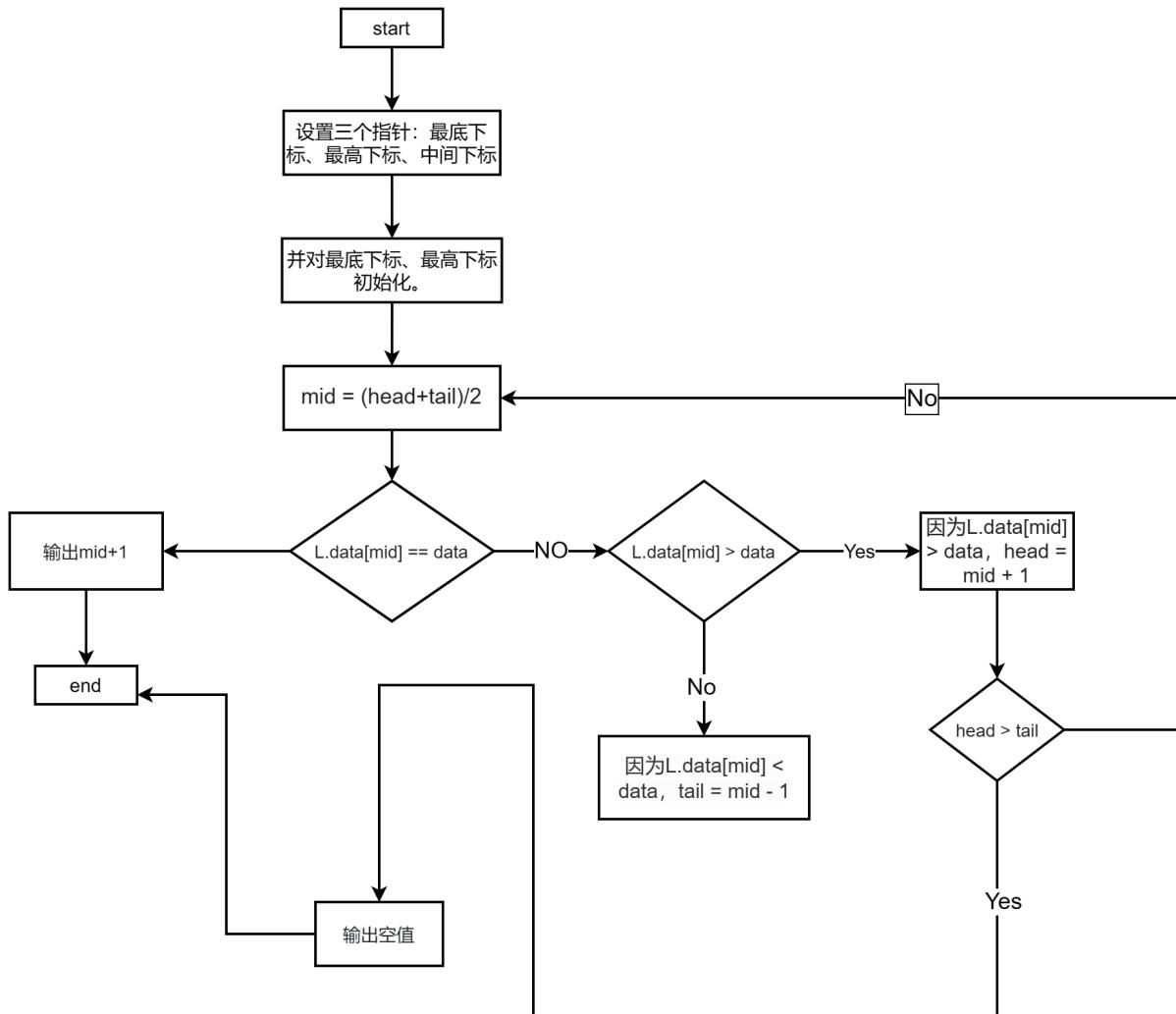
## 折半查找（二分查找）

这让我想起了算法书中提到，假如让你猜一个数，在还没到达正确的数之前主持人只能回答“大了”或“小了”，这个时候，如果从中间开始找那会快很多，但是如果从一开始，那基本上会很慢。

具体操作方法是

```
int search(list L, int data) {  
    //  
    int head = 0, tail = L.length, mid;  
    // 开始循环，直至找到或tail < head  
    while(head < tail) {  
        mid = L.length/2  
        if(L.data[mid] == data) return mid+1;  
        else if(L.data[mid] < data) tail = mid - 1;  
        else if(L.data[mid] > data) head = mid + 1;  
    }  
    return 0;  
}
```

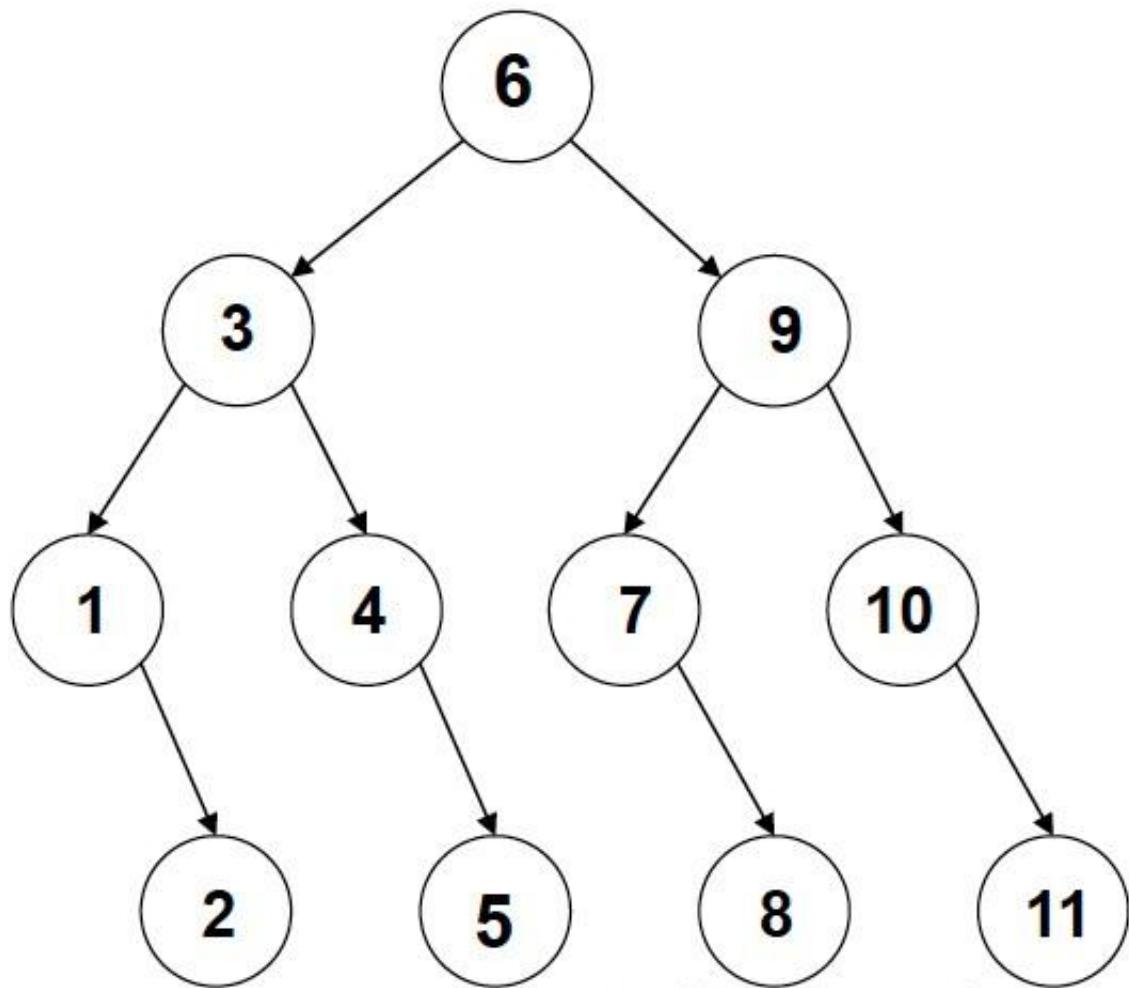
翻译一下这个代码的思路



其实，要找中间的位置，你可以用两只手指摆在开头和末尾，然后同时向中间移去。

## 判定树

它是在二分查找的基础上创造出来的。



<https://www.cnblogs.com/pickchen121/>  
头秃 @十维教育

由图我们可以知道层数就是要比较的次数，最大比较次数为  $\lceil \log_2 n \rceil + 1$ 。

0-1 、 1-2 表示的是head与tail交叉的情况，即找不到值的情况。

## 二叉查找树

它是二分查找与动态查找表的结合体，遵循左小右大的规则。

在这棵树里面，不存在相同的结点，如果有相同的结点，那就会直接返回；对于不存在的结点就会选择添加。

## 平衡二叉查找树

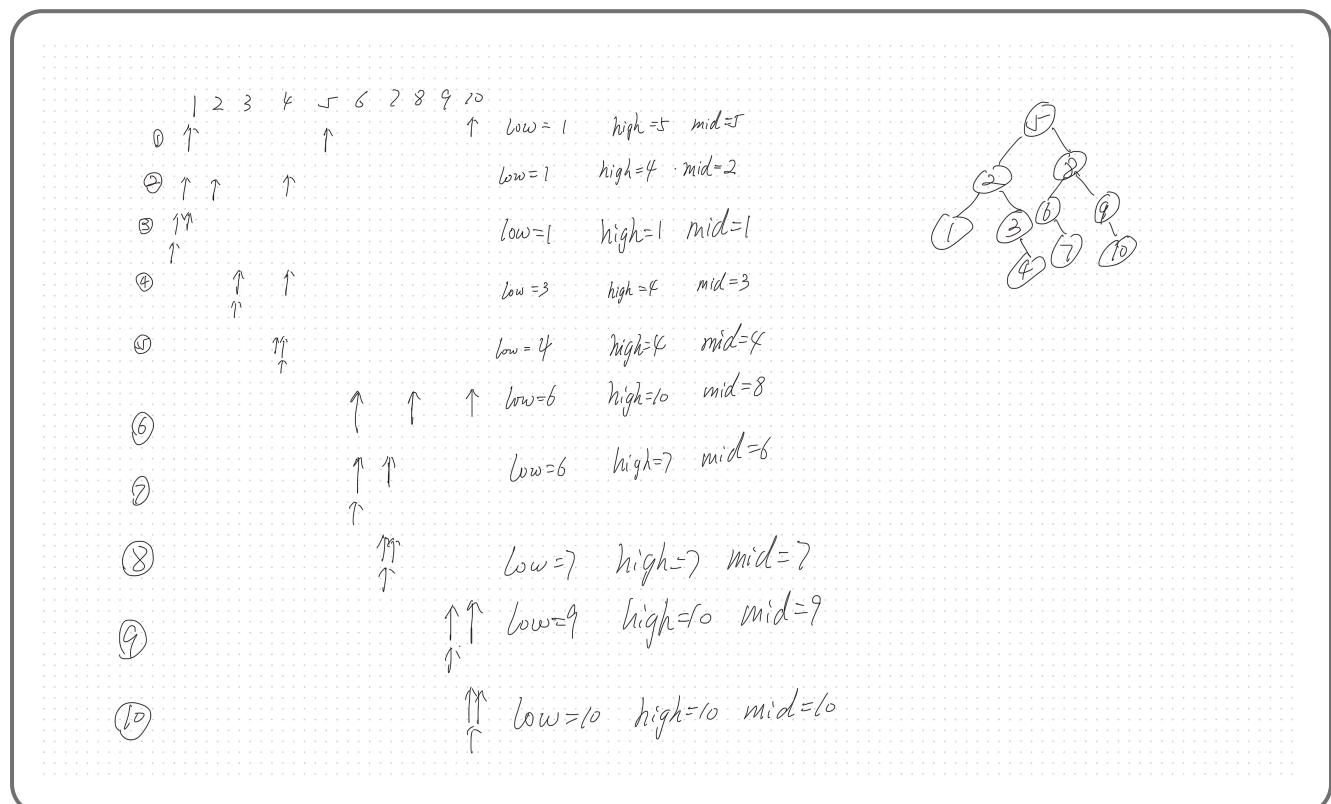
我们都知道，对于平衡查找树而言，左子树的深度与右子树的深度的差的绝对值不大于1，不小于0。但是在计算某个结点的平衡平衡因子的时候不要加上计算绝对值那一步。

# 解题的想法

## 例题一

画出对长度为10的有序表进行折半查找的判定树，并求其等概率时查找成功的平均查找长度。

## 做法



# 二叉排序树

---

# 哈希表

## 前情提要

一般情况下，我们寻找一个记录的时候，大多数情况下都是需要遍历一次才能找到，那有没有一种办法能够使关键字与储存地址是一一对应的（即线性关系），此时，就引出一样东西——哈希表（Hash Table），而用于创造关键字与储存地址是一一对应关系的东西叫做哈希函数。

## 哈希表（Hash Table）

其实，它并不是很难理解。

我们可以选择一些有特征的不是很普遍的部分来作为哈希地址，然后通过创造对应关系，从而实现哈希表的数据存取。

## 构造方法

### 直接定值法

简而言之，就是  $\text{Hash}(\text{key}) = \text{key}$  或者是  $\text{Hash}(\text{key}) = \text{key} \cdot a + b$ 。  
譬如，在一个班级里面找一个学生，就可以这样用，因为学号是符合这个条件的。

### 数学分析法

直观来说，就是找特殊的地方，然后将其作为哈希地址。

### 平方取中法

先平方，然后找特殊的地方，并将其作为哈希地址。

### 折叠法

### 除留余数法

哈希表

这个就是对int类型的关键字进行求余，然后将求余的结果作为哈希地址。至于求余数的设置一般是小于等于m的质数。

## 随机数法

就真的是用随机数来作为哈希地址。

## 冲突

在理想状况下，哈希表的时间复杂度可以当成是  $O(1)$ ，但是实际情况却是在  $O(1)$  和  $O(n)$  之间。

所以，我们要通过一些方法来解决冲突。

## 开放定址法

就是先进行求余，然后看看该地址是否打开，如果打开，那就放入，没打开就向后走一步，再放入。

所以它的公式就是

$$H_i = (\text{Hash}(key) + d_i) \% m \quad (i = 1, 2, 3, \dots)$$

这里要对 `Hash(key) + d_i` 求余的原因是使其相当于一个循环队列。  
其中就这条公式可以引申出三种解决操作：

## 线性探测法

线性探测再散列，其实就是直接使用这条公式，且  $d_i = 1, 2, 3, 4, \dots$ 。

## 例题1

选取哈希函数  $H(k) = (3k) \bmod 11$ 。用开放定址法处理冲突， $d_i = i(7k) \bmod 10 + 1$  ( $i=1,2,3, \dots$ )。试在 0~10 的散列地址空间中对关键字序列 (22, 41, 53, 46, 30, 13, 01, 67) 造哈希表，并求等概率情况下查找成功时的平均查找长

哈希表

度。

0 1 2 3 4 5 6 7 8 9 10  
22 67 41 30 13 46 13 01  
1 3 1 2 1 1 2 6

$$ASL = \frac{1}{8}(4 \times 1 + 2 \times 2 + 3 + 6) \\ = \frac{17}{8}$$

$$\begin{aligned} \textcircled{1} & 22 \times 3 \% 11 = 0 \\ \textcircled{2} & 41 \times 3 \% 11 = 2 \\ \textcircled{3} & 13 \times 3 \% 11 = 1 \\ \textcircled{4} & 46 \times 3 \% 11 = 6 \\ \textcircled{5} & 30 \times 3 \% 11 = 2 \\ & 30 \times 7 \% 10 + 1 = 1 \\ & 2 + 1 = 3 \\ \textcircled{6} & 13 \times 3 \% 11 = 6 \\ & 13 \times 7 \% 10 + 1 = 2 \\ & 2 + 6 = 8 \\ \textcircled{7} & 01 \times 3 \% 11 = 3 \\ & 1 \times 7 \% 10 + 1 = 8 \\ & 8 + 3 = 11 \\ & 11 \% 11 = 0 \\ & (1 \times 7 \% 10 + 1) \times 2 = 16 \\ & 16 + 3 = 19 \% 11 = 8 \\ & (1 \times 7 \% 10 + 1) \times 3 = 24 \\ & 24 + 3 = 27 \% 11 = 5 \\ & (1 \times 7 \% 10 + 1) \times 4 = 32 \\ & 32 + 3 = 35 \% 11 = 2 \\ & (1 \times 7 \% 10 + 1) \times 5 = 40 \\ & (40 + 3) \% 11 = 10 \\ \textcircled{8} & 67 \times 3 \% 11 = 3 \\ & 67 \times 7 \% 10 + 1 = 10 \\ & (10 + 3) \% 11 = 2 \\ & (67 \times 7 \% 10 + 1) \times 2 = 20 \\ & (20 + 3) \% 11 = 1 \end{aligned}$$

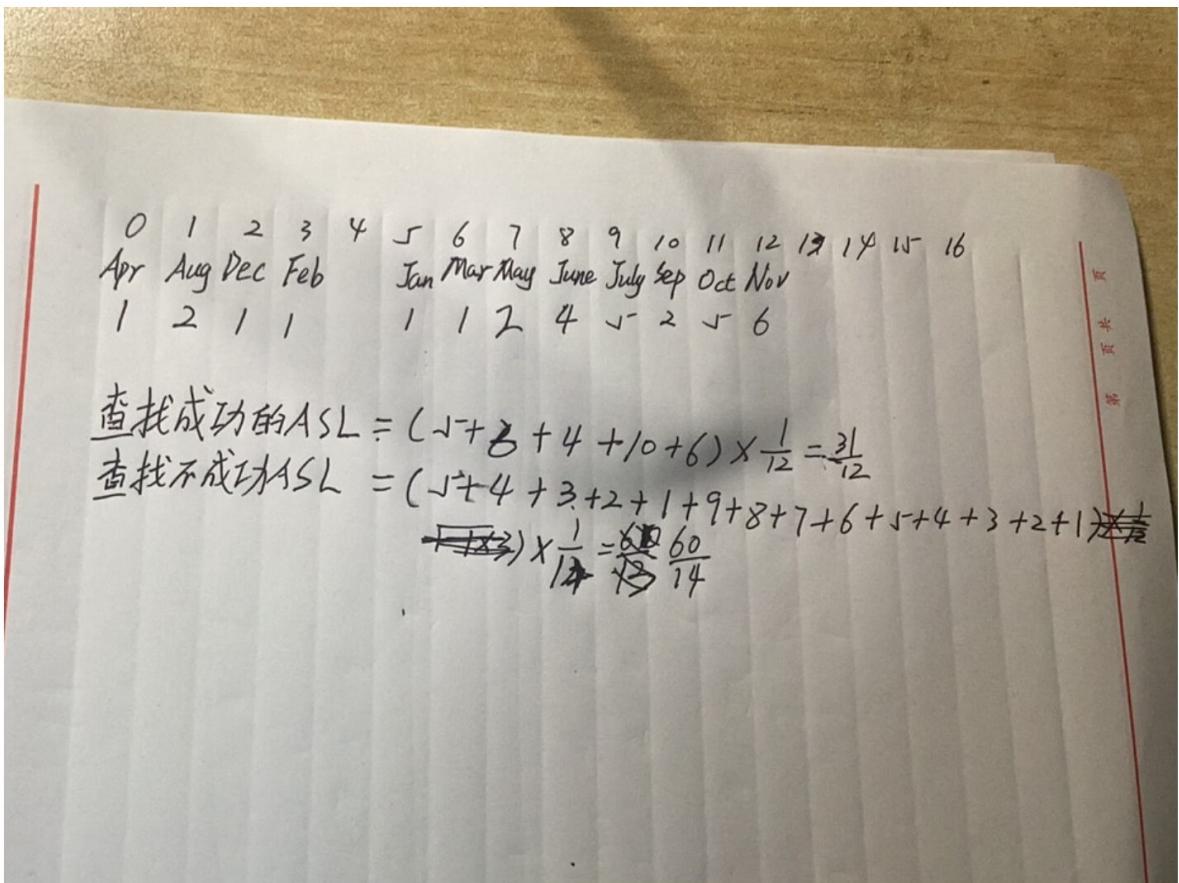
## 例题2

在地址空间为0~16的散列区中，对以下关键字序列构造哈希表：

(Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec)

用线性探测开放定址法处理冲突；求哈希表在等概率情况下查找成功和不成功时的平均查找长度。设哈希函数为 $H(x) = \lfloor i/2 \rfloor$ ，其中*i*为关键字中第一个字母在字母表中的序号的

哈希表  
代码。



```
typedef struct {
    char data[16][5];
    int door[16];
} HashTable;

char words[27] = {' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

int main() {
    char input[6];
    HashTable table;
    for(int i = 0; i < 16; i++) table.door[i] = 0;
    while(1) {
        gets(input);
        for(int i = 1; i < 28; i++) {
            if(input[0] == words[i]) {
                for(int j = 0; j < 1; j++) {
                    if(!table.door[((int)i/2+j)%16]) {
```

```
        strcpy(table.data[((int)i/2+j)%16], inp
        table.door[((int)i/2+j)%16] = 1;
        printf("%d\n", ((int)i/2+j)%16);
        printf("%d\n", j+1);
        break;
    }
}
}
}
}
```

## 二次探测法

与线性探测再散列相似，但是  $d_i$  的取值是  $1, -1^2, 2^2, -2^2, 3^2, -3^2, \dots, \pm k^2$ , ( $k = 1, 2, \dots$ )

## 伪随机数探测法

就是用**随机数序列**来去解决。

# 再哈希法

## 链地址法

链地址法如下图所示



就是创建一个指针数组，假如对应位置的为空，可以直接插入，如果不为空，那就采用尾接法。

至于查找，对于第一个节点，它只要比较一次。查找次数与元素冲突次数有关。

ASL计算

哈希表

假设每个元素的被访问的概率为  $\frac{1}{n}$ ，那剩下的就是与查找次数有关了。

当然，这也与装填因子有关系。

装填因子是  $\frac{\text{表中填入的记录数}}{\text{哈希表表长}}$ ，由公式可知，当表中填入的记录数越小或哈希表表长越大，那发生冲突的可能性越小，反之，越大。

## 查找

查找的结束条件是遇到"空"值或者找到该元素。

# 内部排序的概述

## 大致概括

将一些无序的东西按一定顺序排列起来。

## 排序方法是否稳定

假设有两个下标  $i, j$ ，且  $i < j$ ，如果  $a[i]$  在  $a[j]$  之前，那么排序方法稳定，否则就是不稳定。

一般情况下，证明不稳定只需举出反例即可，而证明稳定则是要对所有的测试用例进行判断才行。

## 排序方法的分类

分为内部排序和外部排序。

## 内部排序

按所需的工作量来分分为以下三类：

1. 简单的排序法，时间复杂度  $O(n^2)$
2. 先进的排序法，时间复杂度  $O(n \log n)$
3. 基数排序法，时间复杂度  $O(d \cdot n)$

## 不变的基本操作

1. 比较关键字大小。
2. 从一个位置移到另一个位置。

## 对于待排序的序列的储存方式

# 直接插入排序

## 操作方案

1. 先选出一个记录，将其放到已排好序的有序表当中。
2. 读取下一个记录的关键词，并将其与上一条记录进行比较，若来得小，就要将其放到零号位，然后，将上一条记录后移，接着，下标继续向前走，并与其所指的记录的关键词进行比较，直到遇到所指记录的关键词小于等于零号位上的后停止，并插入。

## 时间复杂度

其时间复杂度就是  $O(n^2)$

## Code

```
/* typedef RecType SeqList[n + 1]; */
void InsertSort(SeqList R) {
    int j;
    for (int i = 2; i <= N; i++) {
        if (R[i].key < R[i - 1].key) {
            R[0] = R[i];
            for (j = i; i > 1; j--) {
                R[j] = R[j - 1];
                if (R[0].key >= R[j - 1].key) break;
            }
            R[j] = R[0];
        }
    }
}
```

# 希尔插入排序

---

# 快速排序

---

# 简单选择排序

## 定义

简单选择排序，又称为直接选择排序。

简而言之就是先选定一个记录，然后将它后面的关键字与他的关键字进行比较，倘若来的小，替换，但不结束循环。

## Code

```
/* typedef RecType SeqList[n + 1]; */
void InsertSort(SeqList R) {
    int k;
    for (int i = 0; i < n-1; i++) {
        k = i;
        for (int j = i+1; j < n; j++)
            if (R[j].key < R[k].key)
                k = j;
        R[0] = R[i];
        R[i] = R[k];
        R[k] = R[0];
    }
}
```

# 堆排序

---

# 中英文对照表

由于我学校的考试需要考中英文翻译，所以，我还是写这个吧。

中文	英文
数据	data
数据元素	data item
数据对象	data object
数据结构	data structure
物理结构	physical structure
储存结构	storage structure
逻辑结构	logical structure
数据项	data element
数据关系	data relation
项	element
数据类型	data type
顺序存储结构	sequential storage structure
链式存储结构	linked storage structure
时间复杂度	time complexity
空间复杂度	space complexity
抽象数据类型	Abstract Data Type
算法	algorithm
线性表	linear lists
链表	linked list
头指针	head pointed

中英文对照表

中文	英文
头结点	head node
线性链表	linear linked lists
单链表	singly linked lists
双向链表	double linked list
循环链表	circular linked list
直接前驱	immediate predecessor
直接后继	immediate successor
稀疏矩阵	sparse matrix
三元组表	list of 3-tuples
栈	stack
栈顶	top
栈底	bottom
队列	queue
队头	front
队尾	rear
循环队列	circular queue
先进先出	First In First Out
后进先出	Last In First Out
二叉树	binary tree
有序树	ordered tree
完全二叉树	complete binary tree
满二叉树	full binary tree
子树	subtree

中英文对照表

中文	英文
森林	forest
根	root
双亲	parents
孩子	children
祖先	ancestor
层 ( 层次 )	Level
深度	depth
度	degree
树的遍历	traversal of tree
先序遍历	preorder traversal
中序遍历	inorder traversal
后序遍历	postorder traversal
带权路径长度	Weighted Path Length
赫夫曼树	Huffman tree
赫夫曼编码	Huffman codes
图	graph
完全图	complete graph
有向图	directed graph
入度	in-degree
出度	out-degree
顶点	vertex
弧	arc
网	network

中英文对照表

中文	英文
邻接	adjacent
邻接矩阵	adjacency matrix
邻接表	adjacency lists
广度优先搜索	Breadth-First Search
深度优先搜索	Depth-First Search
最小生成树	minimal spanning tree
最短路径	shortest path
拓扑排序	topological sort
关键路径	critical paths
二叉排序树	binary sort tree
平均查找长度	Average Search Length
主关键字	primary key
次关键字	second key
同义词	synonym
冲突	collision
折半查找	binary search
顺序查找	sequential search
哈希表	Hash table
哈希函数	Hash function
装填因子	load factor
判定树	decision tree
排序	sorting
归并排序	merge sort

中英文对照表

中文	英文
希尔排序	Shell's method
快速排序	quicksort
选择排序	selection sort
堆	heap
堆排序	heapsort
基数排序	radix sort
插入排序	insertion sort

# 后记

---