



小红书

小红书号：187309931

1.2.2

数据

数据元素 (是数据的基本单位)

数据项 (构成数据元素不可分割的最小单位)

三者关系: 数据 > 数据元素 > 数据项

逻辑结构: 数据元素之间的逻辑关系

存储结构: 数据元素及其关系在计算机内的表示

逻辑结构 { 线性结构: 有且仅有一个开始和终端结点, 并且所有结点都只有一直前一直后
(一对一)
非线性结构: 一个结点可能有多个直前或直后
(一对多)

存储结构 { 顺序存储结构: 用一组连续的存储单元依次存储数据元素 (用数组实现)
链接存储结构: 用一组任意的存储单元存储数据元素 (用指针实现)
索引存储结构: 在存储结点的同时建立附加的索引表
散列存储结构: 根据结点的关键字直接计算出该结点的存储地址

1.2.3

数据类型 = 值集合 + 值集合上的一组操作

抽象数据类型: 指一个数学模型以及定义在此数学模型上的一组操作

★ 抽象数据类型可用 (D, S, P) 三元组表示

D: 数据对象

S: 是 D 上的关系集合

P: 是对 D 的操作集合

抽象数据类型定义格式:

ADT 抽象数据类型名 {

数据对象: <数据对象的定义> $\triangleright$
数据关系: <数据关系的定义> $\mathcal{S}$
基本操作: <基本操作的定义> $\triangleright$

} ADT 抽象数据类型名

基本操作格式说明:

参数表: 赋值参数 只为操作提供输入值

引用参数 以 & 打头, 除又提供输入值外, 还将返回操作结果

eg: $\text{area}(x, y)$ $\text{scale}(\&G, n)$
 $\downarrow$ $\downarrow$
 赋值参数 引用参数

初始条件: 描述操作执行之前数据结构和参数应满足的条件

操作结果: 说明操作正常完成后数据结构的状况和应返回的结果

ADT 抽象数据类型名 {

Data

数据对象的定义

数据元素之间逻辑关系的定义

Operation

操作 1

初始条件

操作结果描述

操作 2

.....

操作 n

.....

} _____

ADT Circle {

数据对象: $D = \{r, x, y \mid r, x, y \text{ 均为实数}\}$

数据关系: $R = \{ \langle r, x, y \rangle \mid r \text{ 是半径, } \langle x, y \rangle \text{ 是圆心坐标} \}$

基本操作:

Circle(&C, r, x, y)

操作结果: 构造一个圆。

double Area(C)

初始条件: 圆已存在。

操作结果: 计算面积。

double Circumference(C)

初始条件: 圆已存在。

操作结果: 计算周长。

.....

1.4 算法和算法分析

算法：对特定问题求解方法和步骤的一种描述，它是指令的有限序列。其中一个指令表示一个或多个操作

程序 = 数据结构 + 算法

算法特性：① 有穷性 ② 确定性 ③ 可行性 ④ 输入 ⑤ 输出

通过算法效率的高低来评判不同算法优劣

① 时间效率 ② 空间效率

时间效率：用该程序在计算机上执行所消耗的时间来度量

两种度量方法

- 事后统计：将算法实现，测量时间和空间开销
- 事前分析：对算法所消耗的资源估算

运行时间 = 一个简单操作所需的时间 × 简单操作次数
语句频度

(又：一次操作所需时间与计算机本身条件有关，所以还有语句频度)

1.4 算法和算法分析

■ 例如：两个 $n \times n$ 矩阵相乘的算法可描述为：

```
1 for(i=1; i<=n; i++)           // n+1次
2   for(j=1; j<=n; j++)           // n(n+1)次
3     c[i][j]=0;                   // n*n次
4     for(k=0; k<=n; k++)           // n*n*(n+1)次
5       c[i][j]=c[i][j]+a[i][k]*b[k][j]; // n*n*n次
6   }
```

■ 我们把算法所耗费的时间定义为该算法中每条语句的频度之和，则上述算法的时间消耗 $T(n)$ 为：

$$T(n) = 2n^3 + 3n^2 + 2n + 1$$

这是一个关于 n 的函数

算法时间复杂度的渐进表示法

- 为了便于比较不同算法的时间效率，我们仅比较它们的数量级

例如：两个不同的算法，时间消耗分别是：

$$T_1(n) = 10n^2 \text{ 与 } T_2(n) = 5n^2$$

哪个好？

- 若有某个辅助函数 $f(n)$ ，使得当 n 趋近于无穷大时， $T(n)/f(n)$ 的极限值为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n) = O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度（ O 是数量级的符号），简称时间复杂度。

eg: $T_n = 2n^3 + 3n^2 + 2n + 1$

$n \rightarrow \infty \quad T_n/n^3 \rightarrow 2$ 即 T_n 与 n^3 为同阶，引入“ O ”记号
记作 $T(n) = O(n^3)$

定理：若 $f(n) = \cancel{O(n^m)} + \cancel{O(n^m)}$ 是 m 次多项式，则 $T_n = O(n^m)$
(忽略低次项与最高次项的系数)

时间复杂度基本方法：

- ① 找出语句频度最大的那条语句作为基本语句
- ② 计算基本语句的频度得到 n 的某个函数 $f(n)$
- ③ 取其数量级用符号“ O ”表示

分析算法时间复杂度的基本方法

1. 找出语句频度最大的那条语句作为基本语句

2. 计算基本语句的频度得到问题规模 n 的某个函数 $f(n)$

3. 取其数量级用符号“ O ”表示

```
x = 0; y = 0;
for (int k = 0; k < n; k++)
    x++;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        y++;
```

$n+1$

$n+1$

$n \times (n+1)$

$n \times n$

$n^2 + n$ (累加) $T_n = O(n^2)$

```

void exam ( float x[ ][ ], int m, int n ) {
    float sum [ ];
    for ( int i = 0; i < m; i++ ) {
        sum[i] = 0.0;
        for ( int j = 0; j < n; j++ )
            sum[i] += x[i][j];
    }
    for ( i = 0; i < m; i++ )
        cout << i << " : " << sum [i] << endl;
}

```

$T(n) = O(m \cdot n)$

$f(n) = m \cdot n$

时间复杂度是由嵌套最深层语句的频度决定的

算法时间复杂度分析例题

例3：分析以下程序段的时间复杂度

```

i=1;
while(i<=n)
    i=i*2;

```

关键是要找出来执行次数 x 与 n 的关系，并表示成 n 的函数

若循环执行1次： $i=1 \cdot 2=2$ ，

若循环执行2次： $i=2 \cdot 2=2^2$ ，

若循环执行3次： $i=2 \cdot 2=2^3$ ，.....，

若循环执行 x 次： $i=2^x$

设语句②执行次数为 x 次，由循环条件

$i \leq n$ ， $\therefore 2^x \leq n \therefore x \leq \log_2 n$

1 2 3 5 ... X
1 2 4 8
 $\sum x = n$

$x = \log_2 n$

$\therefore$ 时复 $T_n = O(\log_2 n)$

一般考虑最坏时间复杂度

空间复杂度

【例】将一维数组 a 中的 n 个数逆序存放到原数组中。

【算法1】

```

for(i=0; i<n/2; i++) {
    t=a[i];
    a[i]=a[n-i-1];
    a[n-i-1]=t;
}

```

只需一个

$S(n) = O(1)$
原地工作

【算法2】

```

for(i=0; i<n; i++)
    b[i]=a[n-i-1];
for(i=0; i<n; i++)
    a[i]=b[i];

```

$S(n) = O(n)$

需一组数组有储

$\therefore$ 算法1效率更高

2.1 线性表的定义和特征

线性表是具有相同特性的数据元素的一个有限序列

eg $(a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$ 元素个数即为表长

a_1 ↓ 起始结点
 a_i ↓ a_{i-1} 的直接前驱
 a_{i+1} ↓ a_i 的直接后继
 a_n ↓ 终止结点

同一线性表中的元素必有相同特性，数据元素间为线性关系。

多项式非零项的数组表示

$$P_n(x) = P_1x^{e_1} + P_2x^{e_2} + \dots + P_mx^{e_m}$$

线性表 $P = ((P_1, e_1), (P_2, e_2), \dots, (P_m, e_m))$

eg: $A((7, 0), (3, 1), (9, 8), (5, 17))$

$B((8, 1), (2, 7), (-9, 8))$

创建一个新数组C:

0	1	7	5				
7	11	2	17				

step: ① 分别从头遍历A和B每一项

② { 指数相同: 对应系数相加, 若其和不为零则填入, 为零则略
| 指数不相同: 则将指数较小的系数列入C中

③ 一个多项式已遍历完毕时, 将另一个剩余项依次填入C中

eg 先用 $(7, 0)$ 与 $(8, 1)$ 比较, $0 < 1$ ∴ 填入 $(7, 0)$

再用 $(8, 1)$ 与A中下一个 $(3, 1)$ 比较

指数相同: $8+3=11$ 填入 $(11, 1)$

由上顺序存储存在问题

{ 存储空间分配不灵活
运行空间复杂度高

⇒ 应用链式存储

数组

2.3 线性表的类型定义

基本操作(一)

▪ InitList(&L) (Initialization List)

- 操作结果: 构造一个空的线性表L。

▪ DestroyList(&L)

- 初始条件: 线性表L已经存在。
- 操作结果: 销毁线性表L。

表也不在了

▪ ClearList(&L)

- 初始条件: 线性表L已经存在。
- 操作结果: 将线性表L重置为空表。

表还在数据都不存在

基本操作(二)

▪ ListEmpty(L)

- 初始条件: 线性表L已经存在。
- 操作结果: 若线性表L为空表, 则返回TRUE; 否则返回FALSE。

$n=0$

判断是否存在

▪ ListLength(L)

- 初始条件: 线性表L已经存在。
- 操作结果: 返回线性表L中的数据元素个数。

n

基本操作(三)

▪ GetElem(L, i, &e)

- 初始条件: 线性表L已经存在, $1 \leq i \leq \text{ListLength}(L)$ 。
- 操作结果: 用e返回线性表L中第i个数据元素的值。

n
 $-1 \ 0 \ 1 \sim n \ n+1$

▪ LocateElem(L, e, compare())

- 初始条件: 线性表L已经存在, compare()是数据元素判定函数。
- 操作结果: 返回L中第1个与e满足compare()的数据元素的位置。若这样的数据元素不存在则返回值为0。

$=, <, >?$

基本操作(四)

▪ PriorElem(L, cur_e, &pre_e)

- 初始条件: 线性表L已经存在。
- 操作结果: 若cur_e是L的数据元素, 且不是第一个, 则用pre_e返回它的前驱, 否则操作失败, pre_e无意义。

表前驱

▪ NextElem(L, cur_e, &next_e)

- 初始条件: 线性表L已经存在。
- 操作结果: 若cur_e是L的数据元素, 且不是最后一个, 则用next_e返回它的后继, 否则操作失败, next_e无意义。

表后继

基本操作(五)

▪ ListInsert(&L, i, e)

- 初始条件: 线性表L已经存在, $1 \leq i \leq \text{ListLength}(L)+1$ 。
- 操作结果: 在L的第i个位置之前插入新的数据元素e, L的长度加一。

插入元素e之前 (长度为n) : $(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$

插入元素e之后 (长度为n+1) : $(a_1, a_2, \dots, a_{i-1}, \underline{e}, a_i, \dots, a_n)$

基本操作(六)

▪ ListDelete(&L, i, &e)

- 初始条件: 线性表L已经存在, $1 \leq i \leq \text{ListLength}(L)$ 。
- 操作结果: 删除L的第i个数据元素, 并用e返回其值, L的长度减一。
- 删除前 (长度为n) : $(a_1, a_2, \dots, a_{i-1}, \underline{a_i}, a_{i+1}, \dots, a_n)$
- 删除后 (长度为n-1) : $(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

▪ ListTraverse(&L, visited())

- 初始条件: 线性表L已经存在
- 操作结果: 依次对线性表中每个元素调用visited()

2.4 线性表的顺序表示和实现

线性表存储 { 顺序存储: 逻辑上相邻的数据元素存储在物理上相邻单元中
(逻辑上相邻, 物理位置也相邻)

typedef struct {
ElemType data[];
int length;
} SqList

数组
顺序表
地址连续
依次有序
随机存取
类型相同
用一维数组表示
顺序表
长度

SqList L 线性表长变量(用除) } 用一变量表示, 顺序表长度
L.data = (ElemType) 数组长度不可动态定义 } 属性

malloc(sizeof(ElemType)*MaxSize);
#define List_init_size 100
typedef struct {
ElemType elem[List_INIT_SIZE];
int length;
} SqList

构造结构

多项式的顺序存储结构类型定义

$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_mx^{e_m}$

线性表 $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

```
#define MAXSIZE 1000 //多项式可能达到的最大长度

typedef struct {
    float p; //系数
    int e; //指数
} Polynomial;

typedef struct {
    Polynomial *elem //存储空间的基础地址
    int length; //多项式中当前项的个数
} SqList; //多项式的顺序存储结构类型为SqList
```

图书表的顺序存储结构类型定义

booklist - 记事本

ISBN	书名	定价
9787302257646	程序设计基础	25
9787302219872	单片机技术及应用	32
9787302203513	编译原理	46
9787811234923	汇编语言程序设计教程	21
9787512100831	计算机操作系统	17
9787302265436	计算机导论实验指导	18
9787302180630	实用数据结构	29
9787302225065	数据结构 (C语言版)	38
9787302171676	C#面向对象程序设计	39
9787302250692	C语言程序设计	42
9787302150664	数据库原理	35
9787302260806	Java编程与实践	56
9787302252887	Java程序设计与应用教程	39
9787302198505	嵌入式操作系统及编程	25
9787302169666	软件测试	24
9787811231557	Eclipse基础与应用	35

```
#define MAXSIZE 10000 //图书表可能达到的最大长度

typedef struct { //图书信息定义
    char no[20]; //图书ISBN
    char name[50]; //图书名字
    float price; //图书价格
} Book;

typedef struct {
    Book *elem; //存储空间的基础地址
    int length; //图书表中当前图书个数
} SqList; //图书表的顺序存储结构类型为SqList
```

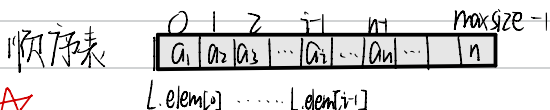
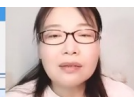
补充: C语言的内存动态分配

SqList L;

L.data=(ElemType*)malloc(sizeof(ElemType)*MaxSize);

- malloc(m)函数, 开辟m字节长度的地址空间, 并返回这段空间的首地址
- sizeof(x)运算, 计算变量x的长度
- free(p)函数, 释放指针p所指变量的存储空间, 即彻底删除一个变量

需要加载头文件: <stdlib.h>



逻辑位序 = 物理位序 - 1

顺序表示意图

```
#define MAXSIZE 100
typedef struct {
    ElemType *elem;
    int length;
} SqList; //定义顺序表类型
```

就像:

```
int a; //定义变量a, a是int型
SqList L; //定义变量L, L是
SqList这种类型的, L是个顺序表
```



线性表的基本操作

- InitList(&L) // 初始化操作, 建立一个空的线性表L
- DestroyList(&L) // 销毁已存在的线性表L
- ClearList(&L) // 将线性表清空
- ListInsert(&L, i, e) // 在线性表L中第i个位置插入新元素e
- ListDelete(&L, i, &e) // 删除线性表L中第i个位置元素, 用e返回
- IsEmpty(L) // 若线性表为空, 返回true, 否则false
- ListLength(L) // 返回线性表L的元素个数
- LocateElem(L, e) // L中查找与给定值e相等的元素, 若成功
返回该元素在表中的序号, 否则返回 0
- GetElem(L, i, &e) // 将线性表L中的第i个位置元素返回给e



线性表L的初始化

返回值类型

Status InitList_Sq (SqList &L)

构造一个空的顺序表

L.elem = new ElemType [maxsize];

为顺序表分配空间

if (!L.elem) exit (OVERFLOW);

有内存分配失败

代表指针

1111 → elem

L.length = 0

内表长度为0

return OK;

销毁线性表L

void DestroyList (SqList &L) {

if (!L.elem) delete L.elem // 释放内存空间

清空线性表

void ClearList (SqList &L) {

L.length = 0 // 将线性表空间释放

}

求线性表的长度

int GetLength (SqList L) {

return (L.length);

}

判断线性表是否为空

```
int IsEmpty(SqList L){  
    if (L.elem == 0) return 1  
    else return 0;  
}
```

【算法2.2】顺序表的取值 (根据位置 i 获取相应位置数据元素的内容)

```
int GetElem(SqList L, int i, ElemType &e){  
    if (i < 1 || i > L.length) return ERROR;  
    //判断值是否合理, 若不合理, 返回ERROR  
    e = L.elem[i-1]; //第i-1的单元存储着第i个数据  
    return OK;  
}
```

$a_1 \quad a_2 \quad \dots \quad a_i$
 $0 \quad 1 \quad \dots \quad i-1$

随机存取

时间复杂度

$O(1)$ 都只执行一次

【算法2.3】顺序表的查找

- 在线性表L中查找与指定值e相同的数据元素的位置
- 从表的一端开始, 逐个进行记录的关键字和给定值的比较。找到, 返回该元素的位置序号, 未找到, 返回0。

```
int LocateElem(SqList L, ElemType e){  
    //在线性表L中查找值为e的数据元素, 返回其序号 (是第几个元素)  
    for (i=0; i < L.length; i++)  
        if (L.elem[i] == e) return i+1; //查找成功, 返回序号  
    return 0; //查找失败, 返回0  
}
```

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

若求其时间复杂度 (恒不知会出现哪)

$\therefore$ 用假设从 $a \rightarrow h$ 出现的查找次数 $\div 7$
 $= (1 + 2 + 3 + 4 + 5 + 6 + 7) \div 7$

用数学方法记为：

$$ASL = \sum_{i=1}^n P_i C_i = P_1 \times 1 + P_2 \times 2 + \dots + P_n \times n$$

找到第几个记录比较的次数
第几个记录被查找的概率

eg (上题) : $\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3 + \dots + \frac{1}{7} \times 7$
($\frac{1}{7}$ 为出现在 7 个格子之一的概率，
后面为每个格子需查找次数)

顺序表的插入

- 插入在最后

1	2	3	4	5
---	---	---	---	---
- 插入在中间

1	2	5	3	4
---	---	---	---	---

^{3,4}
(依次后移)
- 插入在开头

5	1	2	3	4
---	---	---	---	---

 (全部依次后移)

模板

★ Status ListInsert_Sq (SqList & L, int i, ElemType e) {

① if (i < 1 || i > L.length + 1) return ERROR; ① 判断 i 值是否合法

② if (L.length == MAXSIZE) return ERROR; ② 判断存储空间

③ for (j = L.length - 1; j >= i - 1; j--) ③ 插入位置及后移元素

L.elem[j+1] = L.elem[j];

L.elem[i-1] = e

L.length++;

④ 将新元素 e 放入第 i 个位置

⑤ 表长增 1

其时间复杂度：
 对插入的位置共 $n+1$ 种
 ∴ 插入其中之一概率为 $\frac{1}{n+1}$

∴ 其时间复杂度： $\frac{1}{n+1} \times 0 + \frac{1}{n+1} \times 1 + \frac{1}{n+1} \times 2 + \dots + \frac{1}{n+1} \times n$
 $= \frac{n}{2}$ ∴ $O(n)$

顺序表的删除

删除的位置在最后

1	2	3	4	5	6
---	---	---	---	---	---

删除的位置在中间

1	2	3	4	5	6
---	---	---	---	---	---

删除的位置在开头

1	2	3	4	5	6
---	---	---	---	---	---

模板

Status ListDelete_Sq (SqList & L, int i)

① if ($i < 1$ || $i > L.length$) return ERROR

① 判断 i 值是否合法

② for ($j = i$; $j \leq L.length - 1$; $j++$)

② 从删除那位开始
后面依次前移

$L.elem[j-1] = L.elem[j]$

③ $L.length--$

③ 表长减 1

return OK;

}

其时间复杂度

$$\frac{1}{n} \times 0 + \frac{1}{n} \times 1 + \dots + \frac{1}{n} \times n-1 = \frac{n-1}{2}$$

$O(n)$

小结

(顺序表)

查找、插入、删除算法的平均时间复杂度为 $O(n)$

空间复杂度 $O(1)$

(没有应用辅助空间)

顺序表优缺点

优点:

- 存储密度大 (结点本身所占存储量/结点结构所占存储量)
- 可以随机存取表中任一元素

缺点:

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

2.5 线性表的链式表示和实现

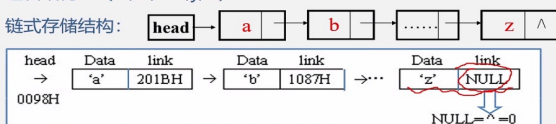
用一组物理位置任意的存储单元来存放线性表的数据元素、

单链表是由头指针唯一确定, 因此单链表可以用头指针的名字来命名

例: 26 个英文小写字母表的链式存储结构

逻辑结构: (a, b, ..., y, z)

链式存储结构:



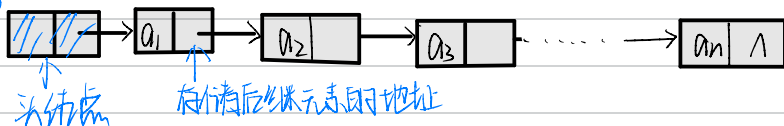
各结点由两个域组成 ① 数据域: 存储元素数值数据

② 指针域: 存储后续结点位置

数据域 指针域

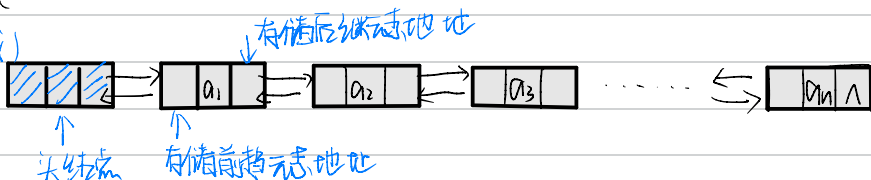
单链表：结点只有一个指针域的链表称为单链表或线性链表

(一个指针域)

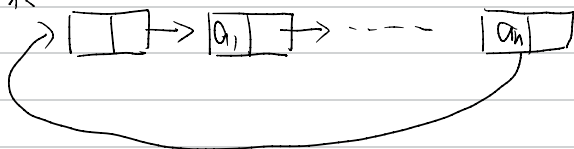


双链表：

(两个指针域)



循环链表



2.5 线性表的链式表示和实现

4. 头指针、头结点和首元结点：

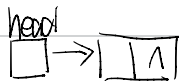


头指针：是指向链表中第一个结点的指针

首元结点：是指链表中存储第一个数据元素 a_1 的结点

头结点：是在链表的首元结点之前附设的一个结点；

空表：① 带头结点



② 不带头结点



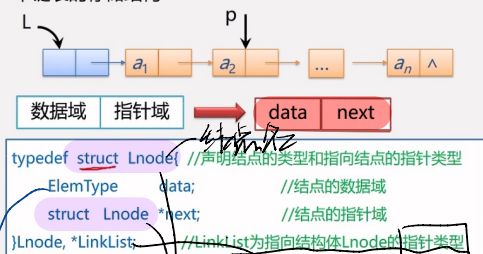
链表的特点 ① 结点在存储器中位置是任意的，即逻辑上相邻的数据元素，在物理上不一定相邻

② 只能通过头指针进入链表，并通过每个结点的指针域依次访问顺序遍历其余结点

(顺序表：随机存取)

顺序存取法

单链表的存储结构



具体数据类型

嵌套定义

指针类型

eg: 存学生学号 姓名 成绩的单链表

```
typedef struct student {
```

```
    char num[8];
```

```
    char name[8];
```

```
    int score;
```

} 数据域

```
    struct student *next
```

指针域

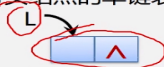
```
}
```

```
LNode * LinkList
```

单链表基本操作的实现

单链表的初始化 (算法2.6) (带头结点的单链表)

- 即构造一个如图的空表



【算法步骤】

- (1) 生成新结点作头结点, 用头指针L指向头结点。
- (2) 将头结点的指针域置空。

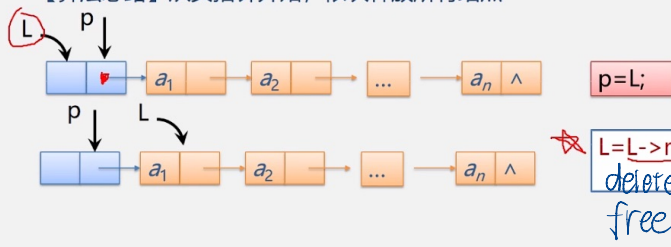
【算法描述】

```
Status InitList_L(LinkList &L){
    L=new LNode; //或L= (LinkList) malloc ( sizeof ( LNode));
    L->next=NULL; (将指针域置空)
    return OK;
}
```

返回值

2. 【补充算法2】——单链表的销毁：链表销毁后不存在

【算法思路】从头指针开始，依次释放所有结点



不断 $L = L \rightarrow \text{next}$

结束条件 $L == \text{NULL}$

3. 【算法】销毁单链表L

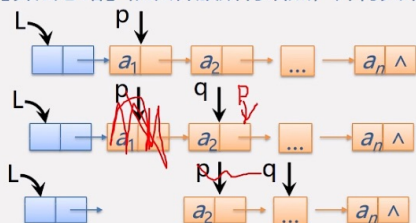
```
Status DestroyList_L(LinkList &L){ //销毁单链表L
    Lnode *p; //或LinkList p;
    while(L){
        p=L;
        L=L->next;
        delete p;
    }
    return OK;
}
```

Handwritten note: 要清空的结点 (The nodes to be cleared)

3. 【补充算法3】——清空链表:

链表仍存在，但链表中无元素，成为空链表（头指针和头结点仍然存在）

【算法思路】依次释放所有结点，并将头结点指针域设置为空



结束条件 $p == \text{NULL}$

▪ 【算法】清空链表L:

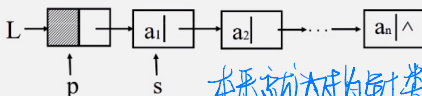
```
Status ClearList(LinkList & L){ // 将L重置为空表
    LNode *p,*q; //或LinkList p,q;
    p=L->next;
    while(p) { //没到表尾
        q=p->next;
        delete p;
        p=q;
    }
    L->next=NULL; //头结点指针域为空
    return OK;
}
```

4. ▪ 【算法】求单链表L的表长

```
int ListLength_L(LinkList L){ //返回L中数据元素个数
    LinkList p;
    p=L->next; //p指向第一个结点
    i=0;
    while(p) //遍历单链表,统计结点数
    {
        i++;
        p=p->next;
    }
}
```

回顾

带头结点的单链表:



类型定义:

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
```

变量定义:

```
LinkList L ;
LNode *p,*s ;
```

重要操作:

```
p=L; //p指向头结点
s=L->next; //s指向首元结点
p=p->next; //p指向下一结点
```

链表: 带型.

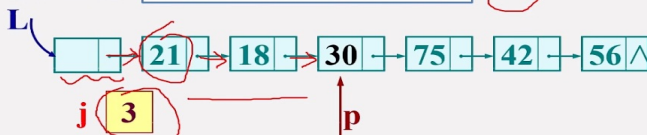
本来就是指针类型 不用加*

取值

【算法2.7】取值——取单链表中第*i*个元素的内容

【算法思路】分别取出表中第3个元素和第15个元素

$i=3$



从链表的头指针出发，顺着链域next逐个结点往下搜索，直至搜索到第*i*个结点为止。因此，链表不是随机存取结构

【算法描述】

```

Status GetElem_L(LinkList L, int i, ElemType &e){ //获取线性表L中的
    某个数据元素的内容，通过变量e返回
    p=L->next; j=1; //初始化
    while(p && j < i){ //向后扫描，直到p指向第i个元素或p为空
        p=p->next; ++j;
    }
    if(!p || j > i) return ERROR; //第i个元素不存在
    e=p->data; //取第i个元素
    return OK;
} //GetElem_L
  
```

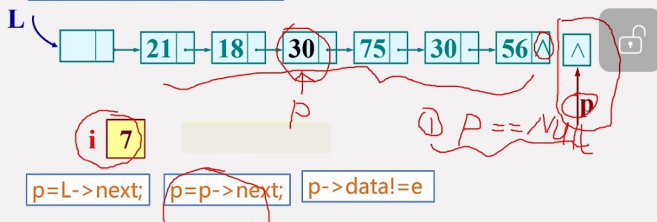
6. 查找

【算法2.8】按值查找——根据指定数据获取该数据所在的位置

分别查找值为30和值为15的元素

$e=30$

$e=15$



①从第一个结点起依次与e相比较

②如果找到一个某值与e相等的数据元素，则返回其在链表中的“位置”或“地址”

③若遍历整个链表都没有与e相等的数据，返回NULL

【算法描述】

```

Lnode *LocateElem_L(LinkList L, ElemType e){
    //在线性表L中查找值为e的数据元素
    //找到，则返回L中值为e的数据元素的地址，查找失败返回NULL
    p=L->next;
    while(p && p->data!=e){
        p=p->next;
    }
    return p;
}
  
```

时间复杂度： $O(n)$

（可能从头找到尾）

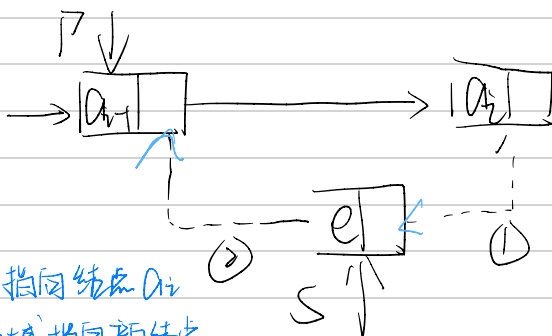
7. 插入

① 首先找到 A_{i-1} 的存储位置 P

② 生成一个数据域为 e 的新结点 s

③ 插入新结点 ① 新结点的指针域指向结点 A_i

② 结点 A_{i-1} 的指针域指向新结点



☆ ① $s \rightarrow \text{next} = P \rightarrow \text{next}$ (A_i 的地址赋给 s 的 next 域)

☆ ② $P \rightarrow \text{next} = s$ (s 的地址赋给 P 的 next 域)

【算法描述】

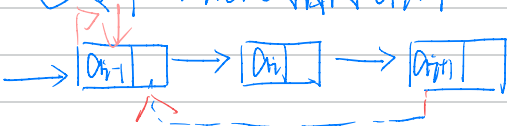
```
//在L中第i个元素之前插入数据元素e
Status ListInsert_L(LinkList &L, int i, ElemType e){
    p=L; j=0;
    while(p && j<i-1) { p=p->next; ++j; } //寻找第i-1个结点, p指向i-1结点
    if(!p || j>i-1) return ERROR; //i大于表长+1或者小于1, 插入位置非法
    s=new LNode; s->data=e; //生成新结点s, 将结点s的数据域置为e
    s->next=p->next; //将结点s插入L中
    p->next=s;
    return OK;
} //ListInsert_L
```

时间复杂度 $O(i)$
(用移动链表)

8. 删除

① 找到 A_{i-1} 位置 P , 保存要删除的 A_i 的值

② 令 $P \rightarrow \text{next}$ 指向 A_{i+1}



时间 $O(n)$
(要查找 A_i)

☆ $P \rightarrow \text{next} = P \rightarrow \text{next} \rightarrow \text{next}$
 A_i A_{i+1}

【算法描述】

//将线性表L中第i个数据元素删除

```
Status ListDelete_L(LinkList &L, int i, ElemType &e){
```

p=L; j=0; $q=i$

while(p->next && j<i-1) { p=p->next; ++j; }

//寻找第i个结点, 并令p指向其前驱

if(!(p->next) || j>i-1) return ERROR; //删除位置不合理

q=p->next;

//临时保存被删结点的地址以备释放

p->next=q->next;

//改变删除结点前驱结点的指针域

e=q->data;

//保存删除结点的数据域

delete q;

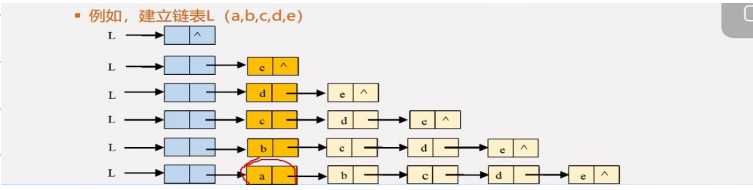
//释放删除结点的空间

return OK;

//ListDelete_L

头插法

- ① 从一个结点开始，重复读入数据
- ② 生成新结点，将读入数据有放到新结点数据域中
- ③ 从最后一个结点开始，依次将各结点插入到链表的前端

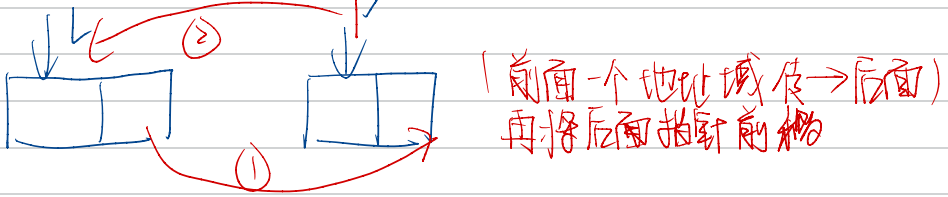


```
graph LR
    L1[L] --> A1[a_n]
    L2[L] --> P1[p] --> A1
    L3[L] --> P2[p] --> A2[a_{n-1}]
    L4[L] --> P3[p] --> A3[a_{n-2}] --> A2
```

```
L = new LNode; //或C语言中如下:
L = (LinkedList)malloc(sizeof(LNode));
p = new LNode;
p->data = a_n;
p->next = L->next;
L->next = p;
```

```
p = new LNode;
p->data = a_{n-1};
p->next = L->next;
L->next = p;
```

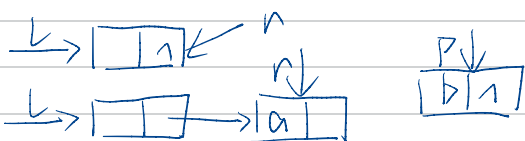
```
void CreateList_H(LinkList &L,int n){
    L=new LNode;
    L->next=NULL; //先建立一个带头结点的单链表
    for(i=n;i>0;--i){
        p=new LNode; //生成新结点 p=(LNode*)malloc(sizeof(LNode))
        cin>>p->data; //输入元素值 scanf(&p-> data);
        p->next=L->next; //插入到表头
        L->next=p;
    }
```



尾插法

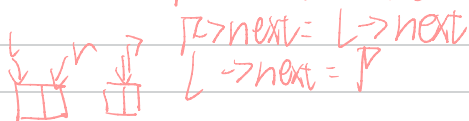
① 从一个空表 L 开始, 将新结点逐个插入到链表尾部, 尾指针 r 指向链表的尾结点。

② 初始时, r 同 L 均指向头结点。每读入一个数据元素则申请一个新结点, 将新结点插入到尾结点后, r 指向新结点。



头插 $r \rightarrow data = a_n$
 $r \rightarrow next = NULL$

- ① $r \rightarrow data = a$
 - ② $r \rightarrow next = NULL$
 - ③ $r \rightarrow next = r$
 - ④ $r = r$
- } 建立新结点



尾插: $r \rightarrow next = r$
 $r = r$

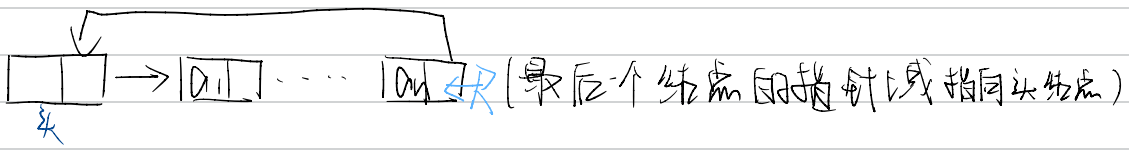
【算法描述】 //正位序输入 n 个元素的值, 建立带表头结点的单链表 L

```
void CreateList_R(LinkList &L, int n){
    L = new LNode; L->next = NULL;
    r = L; //尾指针r指向头结点
    for(i=0; i<n; ++i){
        p = new LNode; cin >> p->data; //生成新结点, 输入元素值
        p->next = NULL;
        r->next = p; //插入到表尾
        r = p; //r指向新的尾结点
    }
}
```

时间 $O(n)$

(后面的传入前一个地址域
 指针后移)

2.3 循环链表



优点: 从表中任一结点出发均可找到表中其他结点

判断

注意:

由于循环链表中没有 **NULL** 指针, 故涉及遍历操作时, 其终止条件就不再像非循环链表那样判断 p 或 $p \rightarrow next$ 是否为空, 而是判断它们是否等于头指针。

循环条件:

$p \neq \text{NULL}$	$\rightarrow$	$p \neq L$
$p \rightarrow next \neq \text{NULL}$	$\rightarrow$	$p \rightarrow next \neq L$
单链表		单循环链表

a_1 存储位置: $R \rightarrow next \rightarrow next$
 a_n 存储位置: R

找 a_1 时间复杂度 $O(1)$
 找 a_n 时间复杂度 $O(n)$

这里也可以看教材上

2.5.3 循环链表

带尾指针循环链表的合并(将Tb合并到Ta之后)

分析有哪些操作?

- ① p 存表头结点
- ② Tb表头连接到Ta表尾
- ③ 释放Tb表头结点
- ④ 修改指针

$p = Ta \rightarrow next$; $Ta \rightarrow next = Tb \rightarrow next \rightarrow next$; $delete Tb \rightarrow next$; $Tb \rightarrow next = p$

2.5.4 双向链表

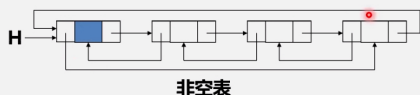
结点结构:

prior	data	next
-------	------	------

双向循环链表

和单链的循环表类似，双向链表也可以有循环表

- 让头结点的前驱指针指向链表的最后一个结点
- 让最后一个结点的后继指针指向头结点。



比较

单链表、循环链表和双向链表的时间效率比较

	查找表头结点 (首元结点)	查找表尾结点	查找结点 * P 的前驱结点
带头结点的单链表 L	L->next 时间复杂度O(1)	从L->next依次向后遍历 时间复杂度O(n)	通过p->next无法找到其前驱
带头结点仅设头指针L的循环单链表	L->next 时间复杂度O(1)	从L->next依次向后遍历 时间复杂度O(n)	通过p->next可以找到其前驱 时间复杂度O(n)
带头结点仅设尾指针R的循环单链表	R->next 时间复杂度O(1)	R 时间复杂度O(1)	通过p->next可以找到其前驱 时间复杂度O(n)
带头结点的双向循环链表L	L->next 时间复杂度O(1)	L->prior 时间复杂度O(1)	p->prior 时间复杂度O(1)

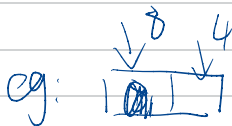


顺序表与链表比较

链式存储优点: ① 结点空间可以动态申请和释放
② 插入删除不需移动数据元素

缺点: ① 存储密度小 (指针占略)

结点数据本身占用空间
结点占用空间略



$$\frac{8}{12} = 67\%$$

② 链式存储非随机存取，对任一结点操作者需从头指针或
通过下标获取
元素。

2.6 顺序表和链表的比较

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

2.7 线性表的应用

1. 线性表的合并 (不按顺序, 谁小谁就先取)

$$La = (17, 5, 3, 11) \quad Lb = (12, 6, 3) \Rightarrow Lc = (17, 5, 3, 11, 2, 6)$$

算法步骤: ①依次取出 Lb 中的每个元素

②在 La 中查找该元素

③如果找不到则将其插入 La 最后

2. 有序表的合并 (按顺序合并, 谁先创建一个新表)

$$La = (11, 7, 8) \quad Lb = (12, 4, 6, 8, 10, 11) \Rightarrow Lc = (1, 2, 4, 6, 7, 8, 8, 10, 11)$$

算法步骤: ①创建一个空表 Lc

②依次从 La 或 Lb 中摘取元素较小的结点, 插入 Lc 表最后, 直到其中一个表取完为止

③继续将 La 或 Lb 中未取完的结点插入 Lc 中

有序链表合并 - 用链表实现

$L_a \rightarrow [1] \rightarrow [1] \rightarrow [7] \rightarrow [8]$

$L_b \rightarrow [] \rightarrow [2] \rightarrow [4] \rightarrow [6] \rightarrow [8] \rightarrow [10] \rightarrow [11]$

$L_c = P_c = L_a$

用 L_a 头结点作为 L_c 的头结点

$\because P_a \rightarrow data < P_b \rightarrow data$

$\rightarrow P_c \rightarrow data = P_a$

$P_c = P_a, P_a = P_a \rightarrow next$

$P_c \rightarrow next = P_b, P_c = P_b$

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc){
    pa = La->next; pb = Lb->next;
    pc = Lc = La; //用La的头结点作为Lc的头结点
    while( pa && pb){ pa 且 pb 不为空
        if(pa->data <= pb->data) { pc->next = pa; pc = pa; pa = pa->next; }
        else { pc->next = pb; pc = pb; pb = pb->next; }
    }
    pc->next = pa ? pa : pb; //插入剩余段 delete Lb; //释放Lb的头结点
}
```

时间复杂度: $O(\text{length}(a) + \text{length}(b))$

2.8 案例

1. 一元多项式运算, 实现两个多项式相加, 减法运算

$$P_n = P_0 + P_1x + P_2x^2 + \dots + P_nx^n$$

将系数写为一个线性表 $P = [P_0, P_1, P_2, \dots, P_n]$

每一项的指数: 隐含在其系数下标之中

eg: $P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$

下标: 0 1 2 3 4

系数 $P[i]$: 10 5 -4 3 2

eg: $P_a(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$

$P_b(x) = -3 + 8x + 4x^2 - 5x^4 + 7x^5 - 2x^6$

系数 P_a 10 5 -4 3 2
 系数 P_b -3 8 4 0 -5 7 -2

P_c 7 13 0 3 -3 7 -2

P_c 剩余的
直接输入

2. 稀疏多项式

多项式非零项的数组表示 (有许多项为0)

$A(x) = 7 + 3x + 9x^8 + 5x^{17}$

(b) $B(x) = 8x + 22x^7 - 9x^8$

下标: 0 1 2 3

系数: 7 3 9 5

指数: 0 1 8 17

下标: 0 1 2

系数: 8 22 -9

指数: 1 7 8

$P_n(x) = P_1x^{e_1} + P_2x^{e_2} + \dots + P_mx^{e_m}$

线性表 $P[(P_1, e_1), (P_2, e_2), \dots, (P_m, e_m)]$

$P_A[(7, 0), (3, 1), (9, 8), (5, 17)]$

$P_B[(8, 1), (22, 7), (-9, 8)]$

① 创建一个新数组 C

② 分别从头遍历比较 a 和 b 的每一项

✓ 指数相同, 对应系数相加, 若其和不为零, 则在 c 中增加一个新项

✓ 指数不相同, 则将指数较小的项复制到 c 中

多项式相加

多项式相加---【算法步骤】

① 指针 p_1 和 p_2 初始化, 分别指向 P_a 和 P_b 的首元结点。

② p_3 指向和多项式的当前结点, 初值为 P_a 的头结点。

③ 当指针 p_1 和 p_2 均未到达相应表尾时, 则循环比较 p_1 和 p_2 所指结点对应的指数值

($p_1 \rightarrow \text{expn}$ 与 $p_2 \rightarrow \text{expn}$), 有下列3种情况:

▪ 当 $p_1 \rightarrow \text{expn} = p_2 \rightarrow \text{expn}$ 时, 则将两个结点中的系数相加

▪ 若和不为零, 则修改 p_1 所指结点的系数值, 同时删除 p_2 所指结点

▪ 若和为零, 则删除 p_1 和 p_2 所指结点;

▪ 当 $p_1 \rightarrow \text{expn} < p_2 \rightarrow \text{expn}$ 时, 则应摘取 p_1 所指结点插入到“和多项式”链表中去;

3.1 栈和队列

栈 — 后进先出



队列 — 先进先出



(栈和队列是线性表的子集)

栈：是一个特殊线性表，是限定仅在一端进行插入和删除操作的线性表。

(后进先出) LIFO结构

表尾称为栈顶(TOP)

表头 栈底

联想 线性表
合并

队列：先进先出的线性表，在表一端插入，另一端删除

$\leftarrow a_1 a_2 a_3 a_4 a_5 a_6 \dots a_n \leftarrow$

3.2 案例引入

1. 进制转换(栈)

十进制整数N向其它进制数d(=2, 8, 16)的转换

转换法则：除以d倒取余

eg: 十进制159转8进制

解：
$$\begin{array}{r} 159 \\ 8 \overline{) 159} \\ \underline{80} \\ 79 \\ \underline{72} \\ 7 \end{array}$$
 依次 $\begin{array}{r} 2 \\ 8 \overline{) 19} \\ \underline{16} \\ 3 \end{array}$ 依次 $8 \overline{) 2}$

$\therefore$ 倒过来 237



$\therefore 237$

2. 括号匹配的检验

括号：方括号，圆括号

其嵌套的顺序随意 eg: $[\] ()$, $[\] ([])$

但注意 匹配问题 与 数量问题
() 数量

eg: 检验 $(()])$ 是否匹配

$\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}$ $\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}$ $\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}$ $\xrightarrow{\text{成功删除}}$ $\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}$ $\rightarrow$ $\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}$ $\rightarrow$ $\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}$ $\rightarrow$ 不匹配

3. 表达式求值 (栈)

3.1 栈的定义和抽象操作实现

ADT Stack
(栈)

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

$R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2, \dots, n \}$
约定 a_n 端为栈顶, a_1 端为栈底.

基本操作: 初始化、进栈、出栈、取栈顶元素
} ADT Stack.

3.3.1 栈的抽象数据类型的类型定义



InitStack(&S) 初始化操作

操作结果：构造一个空栈 S。

DestroyStack(&S) 销毁栈操作

初始条件：栈 S 已存在。

操作结果：栈 S 被销毁。

StackEmpty(S) 判定 S 是否为空栈

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 TRUE，

否则 FALSE。

StackLength(S) 求栈的长度

初始条件：栈 S 已存在。

操作结果：返回 S 的元素个数，即栈的长度。

GetTop(S, &e) 取栈顶元素

初始条件：栈 S 已存在且非空。

操作结果：用 e 返回 S 的栈顶元素。

ClearStack(&S) 栈置空操作

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

Push(&S, e) 入栈操作

初始条件：栈 S 已存在。

操作结果：插入元素 e 为新的栈顶元素。

Pop(&S, &e) 出栈操作

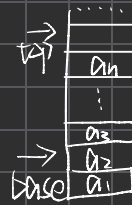
初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素 a_n ，并用 e 返回其值。

线性表
树
图
集合

顺序
链式

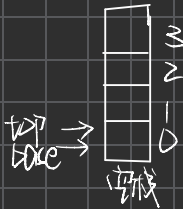
有储方式



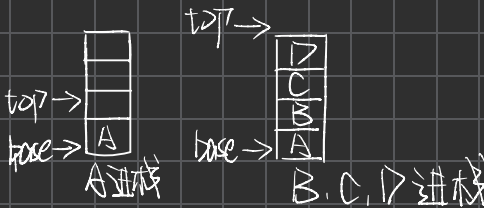
栈本身是线性表，于是栈也有顺序存储和链式存储两种
有储方式：栈底一般在低地址处

附设 top 指针，指示栈顶元素在顺序栈中的位置
另设 base 指针，指示栈底元素在顺序栈中的位置
stacksize 表示长度

eg: stacksize = 4



空栈: $base == top$ 是栈内结构



栈满: $top - base == stacksize$

下溢: 没有元素还因出栈 (一般认为是一种结束条件)

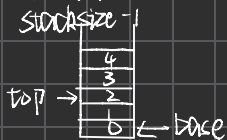
上溢: 已经满了还要压入 (一般认为是一种错误)

3.3.2 顺序栈的表示和实现

顺序栈的表示

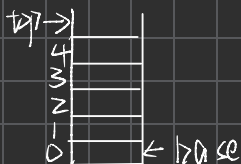
```
#define MAXSIZE 100
typedef struct {
    SElemType *base; //栈底指针
    SElemType *top; //栈顶指针
    int stacksize; //栈可用最大容量
} SqStack;
```

顺序栈的表示



Stack size: 5
top: 4
base: 1

栈中元素个数 = top - base = 3

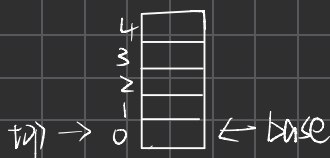


Stack size: 5
top: 4
base: 1

栈满:

栈中元素个数 = top - base = 3

顺序栈初始化



栈空, top == base

3.3.2 顺序栈的表示和实现

顺序栈的表示

```
#define MAXSIZE 100
typedef struct {
    SElemType *base; //栈底指针
    SElemType *top; //栈顶指针
    int stacksize; //栈可用最大容量
} SqStack;
```

判断顺序栈是否为空

Status StackEmpty (SqStack s)

if (s.top == s.base) S指针

return TRUE

else

return FALSE;

求顺序栈长度

int StackLength (SqStack s)

{ return s.top - s.base

}

清空顺序栈 (栈保留)

```
Status ClearStack (SqStack)
```

```
if (S.base)
```

```
S.top = S.base
```

```
return OK;
```

销毁顺序栈 (栈不保留)

```
Status DestroyStack (SqStack &S)
```

```
if (S.base) {
```

```
delete S.base;
```

```
S.stacksize = 0;
```

```
S.base = S.top = NULL
```

```
}
```

```
return OK;
```

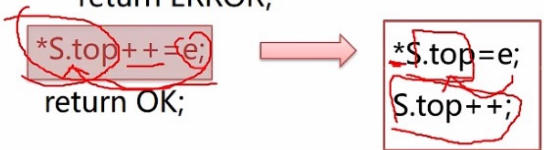
顺序栈自由入栈

① 判断是否栈满 ($S.top - S.base == S.stacksize$)

② 元素存入 top 指针指向位置

③ top 指针后移一位

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



顺序栈的出栈

① 判断栈空, 若空则下溢。 ($S.top == S.base$)

② 获取栈顶元素 e

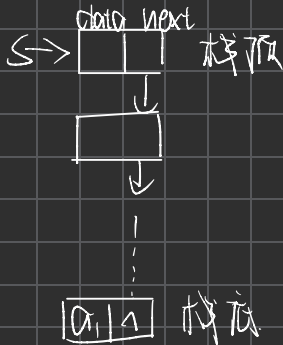
③ 栈顶指针减一

```
Status Pop(SqStack &S, SElemType &e) {  
    // 若栈不空, 则删除S的栈顶元素, 用e返回其值, 并返回OK;  
    // 否则返回ERROR  
    if (S.top == S.base) // 等价于 if (StackEmpty(S))  
        return ERROR;  
    e = *S.top;  
    return OK;  
}
```

$--S.top;$
 $e = *S.top;$

3.3 链栈表示和实现

链栈是运算受限的单链表, 只能在链表头部进行操作



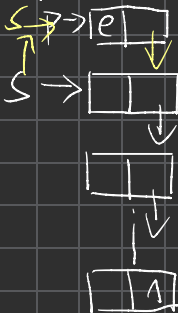
(指针方向与单链表相反)

单: 头 $\rightarrow$ 尾

链栈: 尾 $\rightarrow$ 头

- 链表的头指针就是栈顶
- 不需要头结点
- 基本不存在栈满的情况
- 空栈相当于头指针指向空
- 插入和删除仅在栈顶处执行

链栈的入栈



```
Status Push(LinkStack &S, SElemType e) {  
    p = new StackNode // 生成新结点p  
    p->data = e; // 将新结点数据域置为e  
    p->next = S; // 将新结点插入栈顶  
    S = p; // 修改栈顶指针  
    return OK;  
}
```

链栈的出栈



$e = 'A'$

出栈

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

3.4 栈与递归

递归: 若一个对象或过程包含它自己, 或用它自己作自己定义

分治法: 对于一个较为复杂的问题, 能将其分解成几个相对简单且解法相同或类似子问题

- 必备三个条件
- ① 能将一个问题转变成为一个新问题, 且解法相同或类似
 - ② 可以通过上述转化而使问题简化
 - ③ 必须有一个明确的递归出口

分治法求解递归问题

Void P(参数表) {

if (递归结束条件) 可直接求解步骤 ——— 基本顶

else P(较小的参数) ——— 归纳顶

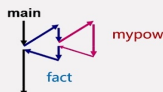
3.4 栈与递归

当多个函数构成嵌套调用时:

```
int main(void) {
    .....
    y = fact(3);
    .....
}

double fact(int n) {
    .....
    z = mypow(3.5, 2);
    .....
}

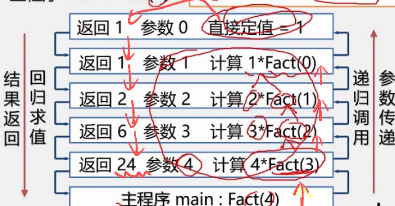
double mypow(double x, int n) {
    .....
}
```



求解阶乘 $n!$ 的过程

if ($n == 0$) return 1;
else return $n * \text{Fact}(n-1)$;

主程序 main : Fact(4)



最后调用的最先返回

递归 优点: 结构清晰, 程序易读

缺点: 每次调用都占用生成工作记录, 保存状态、信息, 入栈, 返回时再出栈
恢复状态信息, 时间开销大

递归 $\rightarrow$ 非递归

方法1: 尾递归, 单向递归 $\rightarrow$ 循环结构

方法2: 自顶向下模拟, 递归运行了

3.5 队列的表示和操作实现

表尾输入, 表头删除, 的线性表

```
ADT Queue{
    数据对象: D={a|a∈ElemSet, i=1,2,...,n, n≥0}
    数据关系: R={<ai-1, ai>|ai-1, ai∈D, i=2,...,n}  约定其中a1端为队头, an端为队尾。
    基本操作:
        InitQueue(&Q)  操作结果: 构造空队列Q
        DestroyQueue(&Q)  条件: 队列Q已存在; 操作结果: 队列Q被销毁
        ClearQueue(&Q)  条件: 队列Q已存在; 操作结果: 将Q清空
        QueueLength(Q)  条件: 队列Q已存在 操作结果: 返回Q的元素个数, 即队长
        GetHead(Q, &e)  条件: Q为非空队列 操作结果: 用e返回Q的队头元素
        EnQueue(&Q, e)  条件: 队列Q已存在 操作结果: 插入元素e为Q的队尾元素
        DeQueue(&Q, &e)  条件: Q为非空队列 操作结果: 删除Q的队头元素, 用e返回该值
        ..... 还有将队列置空、遍历队列等操作.....
} ADT Queue
```

队列 可顺序存储也可链式存储
顺序队列 链式队列

队列的顺序表示——用一维数组 base

```
#define MAXQSIZE 100 //最大队列长度
typedef struct {
    QElemType *base; //初始化的动态分配存储空间
    int front; //头指针
    int rear; //尾指针
} SqQueue;
```

base rear top front. 一维

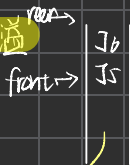
初始时: front = rear = 0

入队: base[rear] = x; rear++;

出队: x = base[front]; front++;

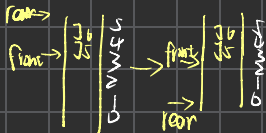
空队: front = rear

假上溢



还留着但无法插入

解决方法：将空间设想成一个循环链表



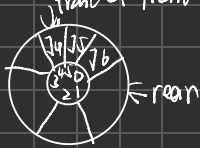
base[0] 接在 base[MaxSize-1] 之后，若 $rear+1 == M$ 则令 $rear=0$

插入元素 Q $base[Q.rear] = x$; (新元素指向尾指针位置)

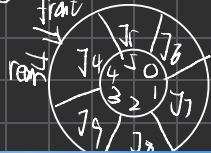
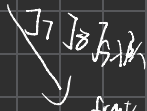
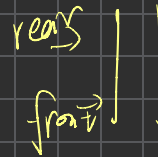
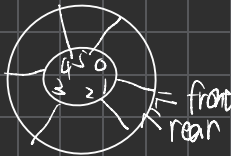
$Q.rear = (Q.rear + 1) \% MaxQSize$
(让它回到 0)

删除元素 $x = Q.base[s.front]$

$Q.front = (Q.front + 1) \% MaxQSize$



J4, J5, J6, J7, J8, J9, J0, J1

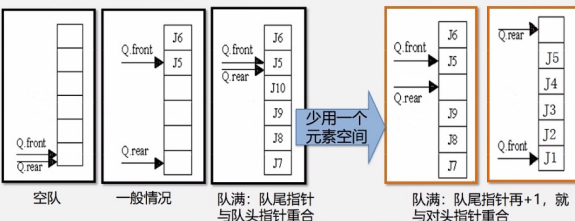


队空 $front == rear$
队满 $front == rear$

有问题

5.2 队列的顺序表示和实现

循环队列解决队满时判断方法——少用一个元素空间：



队空: $front == rear$

队满: $(rear + 1) \% MAXQSIZE == front$

- 决方案：
- ① 另外设一个标志，以区分满/空
 - ② 另设一个变量记录初始值
 - ③ 少用一个元素空间

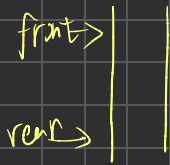
3.5 循环队列的类型定义

初始化:

```
Status InitQueue (SqQueue &Q){
    Q.base = new QElemType[MAXQSIZE] //分配数组空间
    //Q.base = (QElemType*)
    malloc(MAXQSIZE*sizeof(QElemType));
    if(!Q.base) exit(OVERFLOW); //存储分配失败
    Q.front = Q.rear = 0; //头指针尾指针置为0, 队列为空
    return OK;
}
```

求队列的长度

$(\text{rear} - \text{Q.front} + \text{MAXQSIZE}) \% \text{MAXQSIZE}$



判满 $(\text{rear} + 1) \% \text{max} = \text{front}$

3.5.2 队列的顺序表示和实现

❖ 循环队列的操作——循环队列入队 (算法3.13)

```
Status EnQueue(SqQueue &Q, QElemType e){
    if((Q.rear+1)%MAXQSIZE==Q.front) return ERROR; //队满
    Q.base[Q.rear]=e; //新元素加入队尾
    Q.rear=(Q.rear+1)%MAXQSIZE; //队尾指针+1
    return OK;
}
```

3.5.2 队列的顺序表示和实现

❖ 循环队列的操作——循环队列出队 (算法3.14)

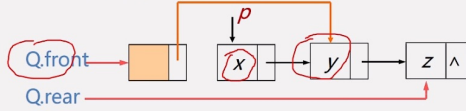
```
Status DeQueue (SqQueue &Q, QElemType &e){
    if(Q.front==Q.rear) return ERROR; //队空
    e=Q.base[Q.front]; //保存队头元素
    Q.front=(Q.front+1)%MAXQSIZE; //队头指针+1
    return OK;
}
```

3.5 队列链式表示和实现



链队列出队

❖ 链队列的操作——链队列出队 (算法3.18)



p = Q.front->next; e = p->data;

Q.front->next = p->next;

初始化

❖ 链队列的操作——链队列初始化 (算法3.16)

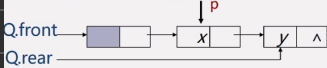


```
Status InitQueue (LinkQueue &Q){
    Q.front = Q.rear = (QueuePtr) malloc(sizeof(QNode));
    if(!Q.front) exit(OVERFLOW);
    Q.front->next = NULL;
    return OK;
}
```

销毁链队列

链队列的操作——销毁链队列 (补充)

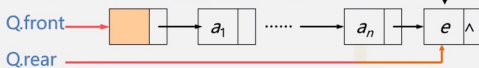
- 算法思想: 从队头结点开始, 依次释放所有结点



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

链队列的入队

❖ 链队列的操作——将元素e入队 (算法3.17)

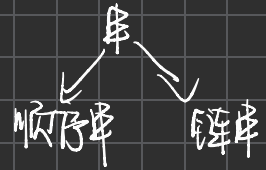


```
Status EnQueue(LinkQueue &Q, QElemType e){
    p = (QueuePtr) malloc(sizeof(QNode));
    if(!p) exit(OVERFLOW);
    p->data = e; p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}
```

线性表

栈和队列 (操作受限的线性表)

串、数组、链表 (内容受限线性表)



串的顺序存储结构:

串: 零个或多个字符 **有序** 组成的有限序列

$S = "a_1 a_2 \dots a_n" \quad (n \geq 0)$

↑ ↑ ↑
串名 串值 串长

空串 $n=0$

子串: 串中任意个连续字符组成的子序列

串相等: ① 长度相等

② 各个对应位置上的字符都相同

串 ADT

ADT String {

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

数据关系: $R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 1, 2, \dots, n \}$

基本操作:

- | | |
|---------------------------|--------|
| (1) StrAssign (&T, chars) | // 串赋值 |
| (2) StrCompare (S, T) | // 串比较 |
| (3) StrLength (S) | // 求串长 |
| (4) Concat (&T, S1, S2) | // 串连接 |

《数据结构与算法基础》

青岛大学 | 数据科学与软件

4.3 串的类型定义、存储结构及运算

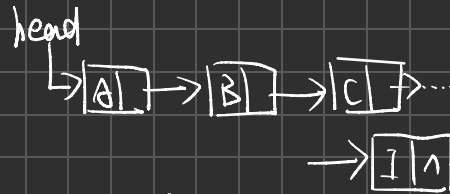
串的顺序存储结构

```

#define MAXLEN 255
typedef struct {
    char ch[MAXLEN+1]; // 存储串的一维数组
    int length; // 串的当前长度
} SString;
  
```

Handwritten note: 256, [0]~[255]

串的链式存储



优点: 操作方便

缺点: 存储空间利用率低



串用一维数组存储

串的模匹配算法

① BF 算法

S: a a a a b c d
↓
T: a b c
↑ ↑ ↑
j
主串: 正文 子串: 模式串
匹配成功

思路: 从S的每一个字符开始依次与T的字符进行匹配

匹配失败: $i = i - j + 1$
 $j = 1$

思路: 将主串的第pos个字符和模式串第一个字符比较

若相等, 继续这个比较后续字符
若不等, 从主串的下一字符起
重新与模式串的第一个
字符比较

```
int Index_BF(SString S, SString T, int pos) {  
    int i = pos, j = 1;  
    while (i <= S.length && j <= T.length) {  
        if (s.ch[i] == t.ch[j]) { ++i; ++j; } // 主串和子串依次匹配下一个字符  
        else { i = i - j + 1; j = 1; } // 主串、子串指针回溯重新开始下一次匹配  
    }  
    if (j > T.length) return i - T.length; // 返回匹配的字符的下标  
    else return 0; // 模式匹配不成功  
}
```

时间复杂度: 最好 $O(1)$ (子串)

最差 $O(n \times m)$ (主串 $\times$ 子串)

平均时间 $\frac{n+m}{2}$

② KMP算法 (提高BF效率)

若匹配失败，不必将主串向前移一位，直接向后移一位

j也不一定回到j=1

i不必回溯，直接到0 (next)

KMP算法

为此，定义 $next[j]$ 函数，表明当模式中第 j 个字符与主串中相应字符

“失配”时，在模式中需重新和主串中该字符进行比较的字符的位置。

前缀 = 后缀
 从头开始的 k-1 个元素 j 前面的 k-1 个元素

$$next[j] = \begin{cases} \max\{k | 1 < k < j, \text{ 且 } "p_1 \dots p_{k-1}" = "p_{j-k+1} \dots p_{j-1}" \} & \text{当此集合非空时} \\ 0 & \text{当 } j=1 \text{ 时} \\ 1 & \text{其他情况} \end{cases}$$

eg:

KMP算法

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
模式串	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	3	4	5	6	7		

首尾 一一比，二二比，三三对
 若匹配✓ 则 k-1 = 前面
 不匹配 则 = 1

数组

按一定格式排列具有相同类型的数据元素。

一维数组声明格式：数据类型 变量名称 [长度]
int num [5] = {0, 1, 2, 3, 4}

二维数组 (矩阵) $A_{m \times n} = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & \dots & \dots & \dots \\ \vdots & \dots & \dots & \dots \\ a_{m0} & \dots & \dots & \dots \end{bmatrix}$

(注：前驱和后续均有2个)

一维数组中的一维数组

声明格式：数据类型 变量名称 [行] [列]
int num [5] [8]

二维数组：二维组中一维数组

n维：n-1维组中一维组

线性表结构是数组结构的一个特例
而数组又是线性表拓展

n维数组的ADT

n维数组的抽象数据类型

ADT Array {

数据对象: $j_i = 0, \dots, b_i - 1, i = 1, 2, \dots, n$

$D = \{a_{j_1 j_2 \dots j_n} \mid a_{j_1 j_2 \dots j_n} \in \text{ElemSet}\}$

数据关系:

$R_1 = \{ \langle a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_{i-1} \dots j_{i+1} \dots j_n} \rangle \mid 0 \leq j_i \leq b_i - 1, 1 \leq i \leq n, \text{且 } k \neq i, 0 \leq j_k \leq b_k - 1, a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_{i-1} \dots j_{i+1} \dots j_n} \in D, i = 2, \dots, n \}$

注：数组可以是多维的，但存储数据元素的内存单元是一维的
因此在存储数组结构之前，需解决将多维关系映射到一维关系，即
存储：

$$\begin{cases} \text{loc}(0) = a \\ \text{loc}(i-1) + L = a + i * L \end{cases}$$

eg: int a[5]

a 内存 2000 每个占 4 求 a[3]

$$a[3] = 2000 + 3 \times 4 = 2012$$

二维 $\rightarrow$ 一维：先行优先，一行一行的存
也可列优先，一列一列的存

三维：按页/行/列存储 [页优先的存储顺序]

矩阵

矩阵的压缩存储：为多个相同的非零元素，只分配一个
存储空间；对零元素不分配空间

1. 对称矩阵：只需存储上/下三角元素 $= \frac{n(n+1)}{2}$

3. 稀疏矩阵存储 (三元组)

以 (i, j, a_{ij}) 来表示
 ↑ ↑ ↑
 行 列 值

i	j	a_{ij}
1	4	6
2	1	12
3	1	3
4	4	4
5	3	6
6	1	16

还原 $\rightarrow$

0	0	0	0
0	0	0	0
12	0	0	0
3	0	0	0
0	0	0	4
0	0	6	0
16	0	0	6

优点：非零元在表中按行有序存放，因此便于进行顺序处理的矩阵运算

缺点：不能随机存取。按行号存取某一行中的非零元则需从头查找。

5.1 树和二叉树的定义

树是 $n (n > 0)$ 个结点的有限集

① $n=0$ 为空树

② $n > 0$ { 有且仅有一个特定的称为根的结点
其余结点可分为 m 个互不相交的有限集

表示方法: ① 嵌套集合 ② 凹入表示 ③ 广义表
(凹入越深层次越低)

二叉树特点 { ① 每个结点最多有两个孩子
② 子树有左右之分, 其次序不能颠倒
③ 二叉树可以是空集合, 根可以有空的左子树或空的右子树

二叉树的抽象数据类型

ADT

数据对象: D : 具有相同特性的数据对象的集合

数据关系: R :

基本操作 P :

○ 性质 1: 在二叉树的第 i 层至多有 2^{i-1} 个结点

○ 性质 2: 若每层都满足 2^{i-1} 则一共有 $2^k - 1$ 个 (满二叉树)

○ 性质 3: 深度为 k 的二叉树至多有 $2^k - 1$ 个结点 ($k \geq 1$)

↓
最多情况 (满二叉树)

性质3: 对任何一棵二叉树T, 如果其叶子数为 n_0 , 度为2的结点数为 n_2 , 则 $n_0 = n_2 + 1$

没有度为1的结点

叶子结点 = 度为2的结点 + 1

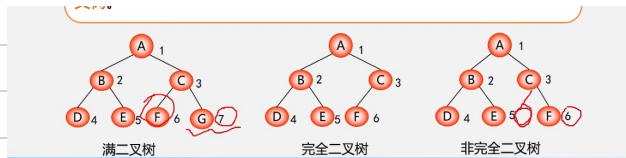
(边数 = 结点数 - 1)

5.4 二叉树的性质 (完全二叉树)

满二叉树: 一棵树有 $2^k - 1$ 个结点
(每一层结点数都最大, 且每层都满)

完全二叉树: 深度为 k 的具有 n 个结点的二叉树当且仅当每一个结点都与深度为 k 的满二叉树中编号为 $1 \sim n$ 的结点一一对应时, 即为完全二叉树。

(仅左边排布)



特点: ① 叶子只可能分布在层次最大的两层上 (最后两层)

② 对任一结点, 如果右子树的最大层次为 i , 则其左子树最大层次必为 i 或 $i+1$ 左子左。

性质4: 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ (结点数 n 与深度 k 关系)

注: $\lfloor x \rfloor$ 代表不大于 x 的最大整数

$$k = \lfloor \log_2 n \rfloor + 1$$

性质5:

性质5: 如果对一棵有 n 个结点的完全二叉树 (深度为 $\lfloor \log_2 n \rfloor + 1$) 按层序编号 (从第1层到第 $\lfloor \log_2 n \rfloor + 1$ 层, 每层从左到右), 则对任一结点 i ($1 \leq i \leq n$), 有:

- (1) 如果 $i = 1$, 则结点 i 是二叉树的根, 无双亲; 如果 $i > 1$, 则其双亲是结点 $\lfloor i/2 \rfloor$.
- (2) 如果 $2i > n$, 则结点 i 为叶子结点, 无左孩子; 否则, 其左孩子是结点 $2i$.
- (3) 如果 $2i + 1 > n$, 则结点 i 无右孩子; 否则, 其右孩子是结点 $2i + 1$.

性质5表明了完全二叉树中双亲结点编号与孩子结点编号之间的关系

深

二叉树的存储结构

- 顺序存储结构
- 链式存储结构
 - 二叉链表
 - 三叉链表

顺序存储

二叉树的顺序存储

实现：按满二叉树的结点层次编号，依次存放二叉树中的数据元素。

//二叉树顺序存储表示

#define MAXSIZE 100

typedef TElemType SqBiTree[MAXSIZE]

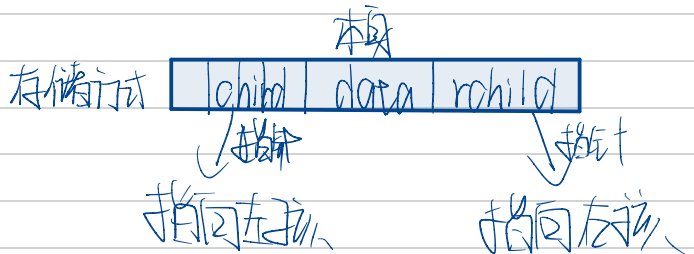
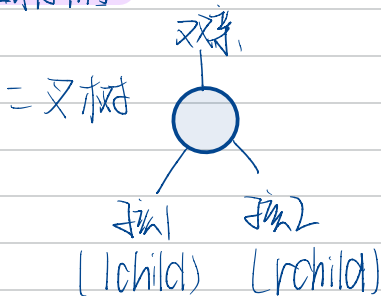
SqBiTree bt;

Binary Tree

缺点：深度为 R 的且只有 R 个结点的单支树需要长度为 $2^R - 1$ 的一维数组，浪费空间

特点：造完全二叉树与满二叉树

链式存储



二叉链表存储结构

typedef struct BiNode{

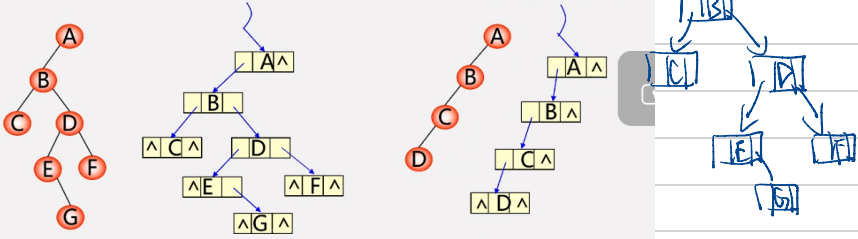
TElemType data;

struct BiNode *lchild, *rchild; //左右孩子指针

}BiNode, *BiTree;

例:

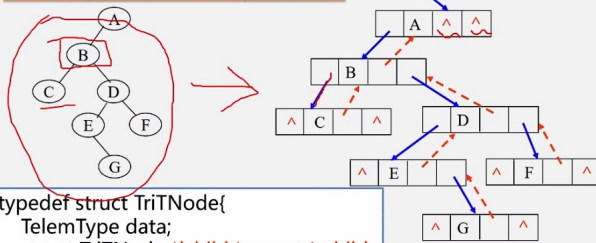
二叉树的链式存储结构



在 n 个结点二叉链表串, 链域为 $2n$, 但除根结点外每个结点有且仅有一个双亲。∴表示前驱和后继的链域至少占 $n-1$ 个

三叉链表

lchild	data	parent	rchild
--------	------	--------	--------



```
typedef struct TriTNode{
    TelemType data;
    struct TriTNode *lchild,*parent,*rchild;
}TriTNode,*TriTree;
```

JS 遍历二叉树

L: 左子树

√: 根结点

R: 右子树

先序遍历: DLR

DLR

中序遍历: LDR

LDR

后序遍历: LRD

LRD

若各结点值不相同 则先中后值都是唯一的

先,中 ; 后中都可确定一棵二叉树 (最后不行)

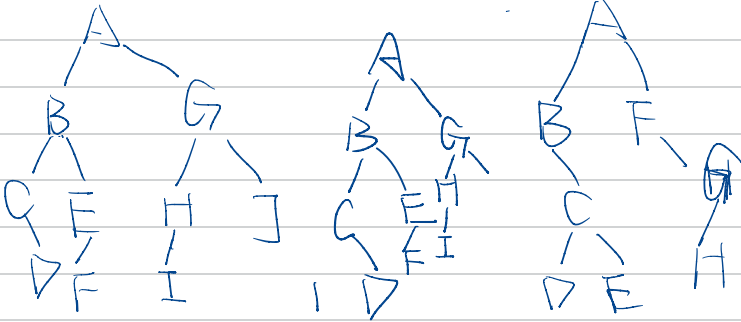
例: 构造二叉树

先: A B C D E F G H I J

中: C D B F E A I H G J

中: B D C E A F H G

后: D E C B H G F A



二叉树先序遍历算法

```
Status PreOrderTraverse(BiTree T){
    if(T==NULL) return OK; //空二叉树
    else{
        visit(T); //访问根结点
        PreOrderTraverse(T->lchild); //递归遍历左子树
        PreOrderTraverse(T->rchild); //递归遍历右子树
    }
}
```

递归 (调用)

遍历算法的分析

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

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

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



如果去掉输出语句,从递归的角度看,三种算法是完全相同的,或者说这三种算法的访问路径是相同的,只是访问的顺序不同

时间效率: $O(n)$ (每个结点只访问一次)

空间效率: $O(n)$ (栈占用的最大辅助空间)

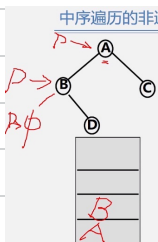
中序遍历非递归算法

基本思想: ① 建立一个栈

② 根结点进栈, 遍历左子树

③ 根结点出栈, 输出根结点, 遍历右子树

中序遍历的非递归算法 (算法5.2)



```
Status InOrderTraverse (BiTree T) {
    BiTree p; InitStack(S); p=T; // 栈指针
    while(p || !StackEmpty(S)) {
        if(p) { Push(S,p); p=p->lchild; }
        else { Pop(S,q); printf("%c", q->data);
            p=q->rchild; }
    } // while
    return OK;
}
```

二叉树的层次遍历

从根结点开始, 从上至下, 从左至右的顺序访问每一个结点 (类似队列)

二叉树的层次遍历

算法设计思路: 使用一个队列

- I. 将根结点进队;
- II. 队不空时循环: 从队列中出队一个结点*p, 访问它;
 - ❶ 若它有左孩子结点, 将左孩子结点进队;
 - ❷ 若它有右孩子结点, 将右孩子结点进队。

二叉树的层次遍历

算法设计思路: 使用一个队列

- I. 将根结点进队;
- II. 队不空时循环: 从队列中出队一个结点*p, 访问它;
 - ❶ 若它有左孩子结点, 将左孩子结点进队;
 - ❷ 若它有右孩子结点, 将右孩子结点进队。

二叉树的层次遍历

二叉树层次遍历算法:

```
void LevelOrder(BTNode *b) {
    BTNode *p;    SqQueue *qu;
    InitQueue(qu);    //初始化队列
    enqueue(qu, b);    //根结点指针进入队列
    while (!QueueEmpty(qu)) {    //队不为空, 则循环
        dequeue(qu, p);    //出队结点p
        printf("%c ", p->data);    //访问结点p
        if (p->lchild != NULL) enqueue(qu, p->lchild);    //有左孩子时将其进队
        if (p->rchild != NULL) enqueue(qu, p->rchild);    //有右孩子时将其进队
    }
}
```

先序遍历序列建立二叉树的二叉链表

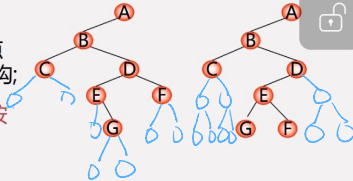
按先序遍历序列建立二叉树的二叉链表

例: 已知先序序列为:

ABCDEGF

(1) 从键盘输入二叉树的结点信息, 建立二叉树的存储结构;

(2) 在建立二叉树的过程中按照二叉树先序方式建立;



问题: 两种不同结构的二叉树先序序列相同

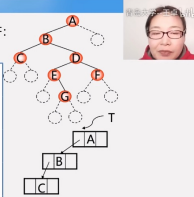
∴ 要给他们补充空结点

∴ 中序宏用#表示: ABC##DE##G##F##

对右图所示二叉树, 按下列顺序读入字符:

ABC##DE##G##F##

```
Status CreateBiTree(BiTree &T) {
    scanf("%c", &ch); //cin >> ch;
    if (ch == "#") T = NULL;
    else {
        if (!T = (BiNode *) malloc(sizeof(BiNode)))
            exit(OVERFLOW); //T = new BiNode;
        T->data = ch; //生成根结点
        CreateBiTree(T->lchild); //构造左子树
        CreateBiTree(T->rchild); //构造右子树
    }
    return OK;
} // CreateBiTree
```



《数据结构与算法基础》

青岛大学 数据科学与软件工程学院

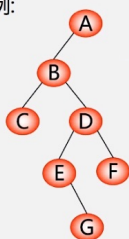
线索二叉树

问: 如何寻找特定遍历序列中的二叉树结点的前驱和后继

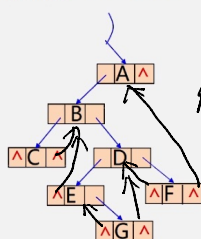
—— 可利用链表中的指针域

若某结点左孩子为空, 则将其左孩子指针域改为指向其前驱; 若右孩子为空, 则改为指向其后继。这种改变指向的指针即为“线索”

例:



中序遍历: CBEGDFA



从A: A在中序中无前驱
但左域满了,不写

看B: 都满 → C: 都空, 但

在中序中后继为B: 右域指B

→ D都满 → E: 左空, 前驱为B: 左子域为B

→ F都空: 前驱D 后继B: 左子域指D, 右指B

→ G都空: 前驱E 后继D: 左子域指E, 右指D

线索二叉树

为区分lchild和rchild指针到底是指向孩子的指针, 还是指向前驱或者后继的指针, 对二叉链表中每个结点增设两个标志域 ltag 和 rtag, 并约定:

ltag = 0 lchild 指向该结点的左孩子

ltag = 1 lchild 指向该结点的前驱

rtag = 0 rchild 指向该结点的右孩子

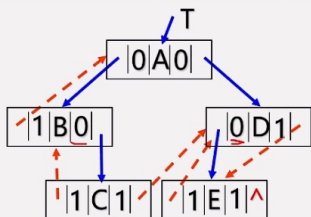
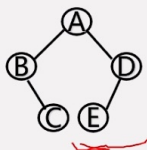
rtag = 1 rchild 指向该结点的后继

结点结构:

lchild	ltag	data	rtag	rchild
	0/1		0/1	

线索二叉树

先序线索二叉树



先序序列: A B C D E

树和森林

森林是树的集合

树的存储结构: ① 双亲表示法

(找双亲易找孩子难)

1 双亲表示法

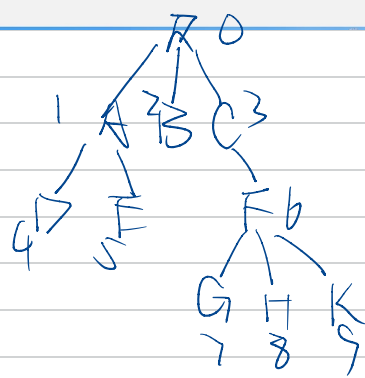
实现: 定义结构数组

存放树的结点,
每个结点含两个域:

- 数据域: 存放结点本身信息。
- 双亲域: 指示本结点的双亲结点在数组中的位置。

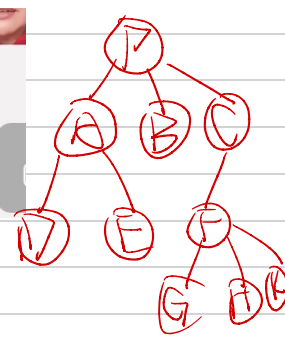
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

相同为兄弟

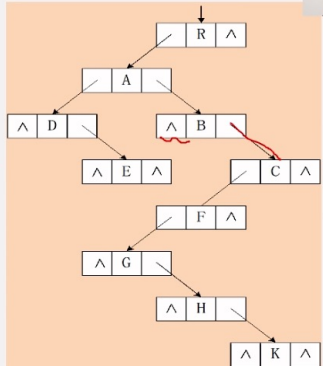
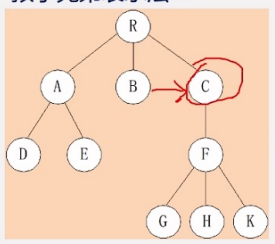


③ 孩子兄弟表示法 (二叉树表示法, 二叉链表表示法)

实现: 用二叉链表作树的存储结构, 链表中每个结点的两个指针域分别指向其第一个孩子结点和下一个兄弟结点

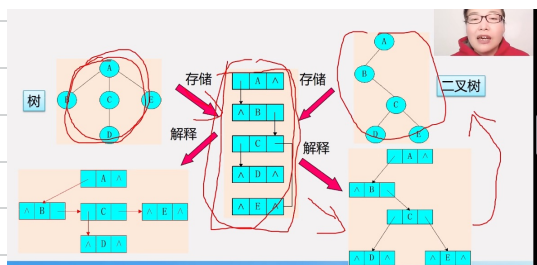


孩子兄弟表示法



左孩子
右兄弟

树转换为二叉树



树与二叉树对应兄弟关系相同，可以相互转换

① 在兄弟之间连线

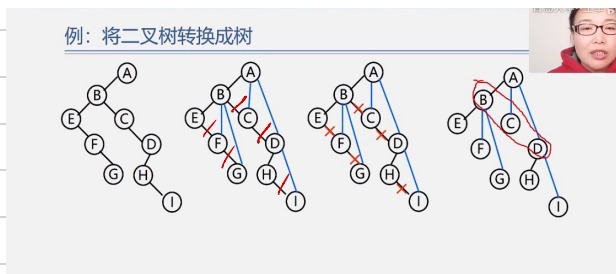
② 除左孩子外所有连线

二叉树转树：

① 若P为双亲的左孩子，则将P的右孩子，右孩子的右孩子……沿分支找到的所有右孩子，都与P用双亲那样连起来

② 抹掉原二叉树中双亲与右孩子的连线

③ 调整



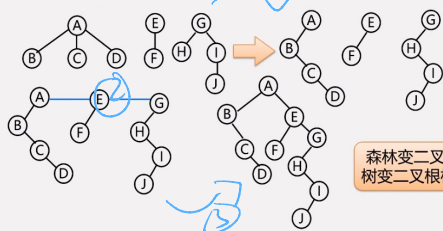
树 → 二叉：连兄弟，砍断左的线

二叉树：P的左孩子，将P的右孩子，右孩子的右孩子……全连

森林转换为二叉树

- ① 将各棵树分别转换为二叉树
- ② 将每颗树的根结点用线相连
- ③ 以第一棵树根结点为二叉树的根，以根为轴心，顺时针旋转，构成二叉树结构

例：森林转换成二叉树



二叉树转换为森林

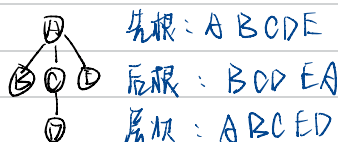
- ① 抹线：将二叉树中根结点与其右孩子连线，及沿左分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立二叉树（去掉右线）
- ② 还原：将孤立的二叉树还原为树。

树的遍历

① 先根遍历：先访问根结点，后依次先遍历各棵子树

② 后根遍历：先依次后根遍历各棵子树，后访问根

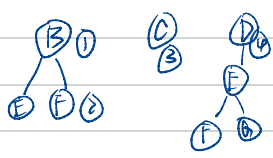
③ 层次遍历：自上而下访问



森林的遍历

- 将森林看作由三部分构成：
- ① 森林中第一棵树的根结点
 - ② 森林中第一棵树的子树森林
 - ③ 其它树构成的森林

- 先序遍历：
- ① 访问第一棵树根结点
 - ② 先序遍历森林中第一棵树的子树森林
 - ③ 先序遍历森林中其它树



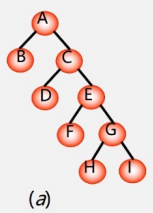
- 中序遍历：② ① ③

哈夫曼树

5.7.1 哈夫曼树的基本概念

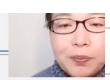
路径：从树中一个结点到另一个结点之间的分支构成这两个结点间的路径。

结点的路径长度：两结点间路径上的分支数。



(a) 从 A 到 B, C, D, E, F, G, H, I 的路径长度分别为 1, 1, 2, 2, 3, 3, 4, 4。

树的路径长度：从树根到每一个结点的
路径长度之和



结点数目相同的二叉树中，完全二叉树是路径长度最短的二叉树

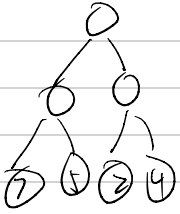
权：将树中结点赋给一个有着某种含义的数值

结点的带权路径长度：从根结点到该结点之间的路径长度与该结点的权的乘积

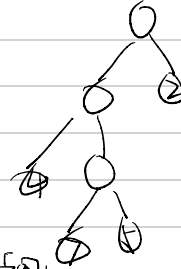
树中带权路径长度：树中所有叶子结点的带权路径长度之和

$$WPL = \sum w_k l_k$$

例



$$\sum (\text{权} \times \text{路径长度})$$
$$W = 7 \times 2 + 5 \times 2 + 2 \times 2 + 4 \times 2$$
$$= 36$$



$$W = 7 \times 3 + 5 \times 3 + 2 \times 1 + 4 \times 2$$
$$= 46$$

(两者 W 不同, 结构同)

哈夫曼树: 带权路径长度最短树
(最优树)

B树

(一个有序的多路查询树)

① 树中每个结点至多有 m 个孩子结点 (至多有 $m-1$ 个关键字)

② 每个结点的结构为



↓ 关键字
↓ 找到子树的指针

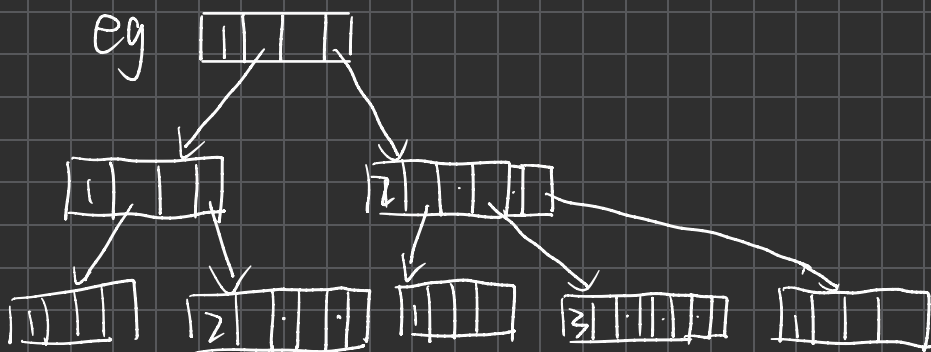
③ 除根与叶子, 其它每个

☆ 结点至少有 $\lceil \frac{m}{2} \rceil$ 个子结点

☆ ④ 除根节点外每个结点, 关键字
 $\lceil \frac{m}{2} \rceil - 1 \leq n \leq m-1$

B树最大高度
$$h \leq \frac{\log_{\lceil \frac{m}{2} \rceil} (M+1)}{1} + 1$$

eg

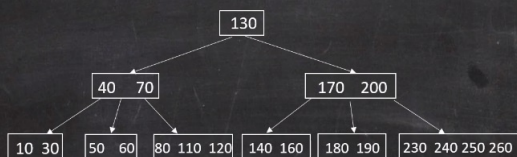


叶子结点, 至多有 $m-1=3$ 个关键字

$\therefore m=4$, 为4阶

例:

已知一棵5阶B树如下图所示, 给出在B树中删除80、200、180、50、140的过程。



叶子结点:

留有: 删一个, 剩余仍对 $\lceil \frac{m}{2} \rceil - 1$ 有用

不留有: 删一个, 不满足 $\lceil \frac{m}{2} \rceil - 1$

↓
借兄弟

↓ 若不行

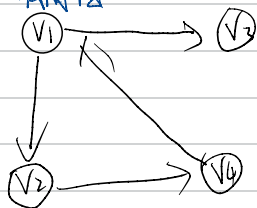
借颜色 → 合并

图 $G=(V, E)$

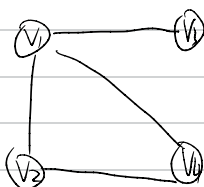
V : 顶点 (数据元) 的有限非空集合

E : 边的有限集合

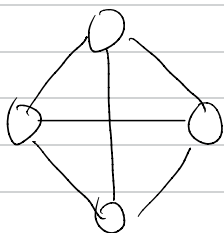
有向图



无向图

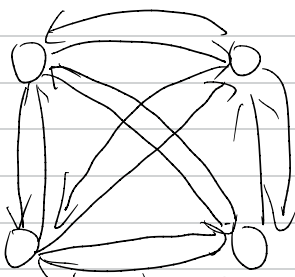


完全图: 任两点之间都有一条边相连



无向完全图

$$\frac{n(n-1)}{2}$$



有向完全图

$$2C_n^2$$

稀疏图: 有很少边或弧的图

稠密图: 有较多边或弧的图

网: 边/弧带权的图

邻接: 有边/弧相连两个顶点之间的关系

存在 (V_i, V_j) 则称 V_i 与 V_j 为邻接点

存在 $\langle V_i, V_j \rangle$ 则称 V_i 邻接到 V_j , V_j 邻接于 V_i

关联：边/弧与顶点之间关系。

存在 $\langle V_p, V_j \rangle$ / (V_p, V_j) 则称该边/弧关联

顶点的度：与该顶点相关联边数/弧数

路径：按顺序边构成的顶点序列

路径长度：路径上边或弧的数目/权之和

回路：第一个顶点和最后一个顶点相同的路径

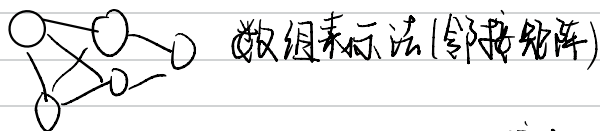
简单路径：除路径起点和终点可以相同外，其余顶点均不相同

连通图：在无向图 $G=(V, E)$ 中，若对任何两个顶点 v, u 都存在从 v 到 u 的路径，则称 G 是连通图

子图：设有两个图 $G=(V, E)$ $G_1=(V_1, E_1)$ 若 $V_1 \subseteq V$
 $E_1 \subseteq E$ 则称 G_1 为 G 的子图

6.4 图的存储结构

图没有顺序存储结构但可以借助二维数组来表示元素间的关系。



数组表示法(邻接矩阵)

链式存储结构

邻接表
十字
邻接多重

数组(邻接矩阵)表示法

① 建立一个顶点表(记录各顶点信息) 和一个邻接矩阵(表示各顶点之间关系)

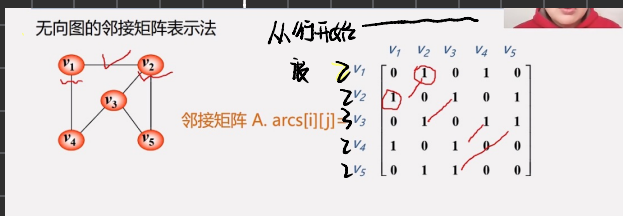
设图 $A=(V,E)$ 有 n 个顶点, 则

顶点表:

v_0	0	1	3	...	$n-1$
v_1					
v_2					
v_3					
v_4					

图的邻接矩阵是一个二维数组 $A.arcs[n][n]$ 定义为:

$$A.arcs[i][j] = \begin{cases} 1 & \text{如果 } \langle i, j \rangle \in E \text{ 或 } (i, j) \in E \\ 0 & \text{其他} \end{cases}$$



邻接矩阵是有意义的

① 无向图的邻接矩阵是对称的

② 顶点 v_i 的度 = 第 i 行/列中 1 的个数



不对称了

PS: 看行从行开始
每个点 发出的边

发出边: 从行开始

有几个 1 就有几条边

收入边: 从列开始

PS: 有向邻接表求度则需分开

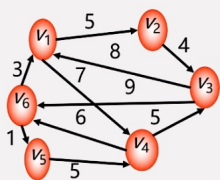
出发: 行
入度: 列

度 = 行 + 列 的 "1" 个数

6.4.1 邻接矩阵

网（即有权图）的邻接矩阵表示法

定义为: $A_{arcs}[i][j] = \begin{cases} W_{ij} & \langle v_i, v_j \rangle \text{ 或 } (v_i, v_j) \in VR \\ \infty & \text{无边 (弧)} \end{cases}$

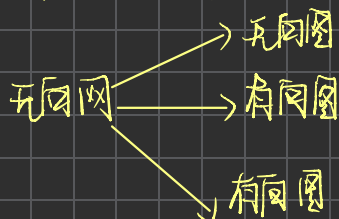


邻接矩阵

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	∞	5	∞	7	∞	∞
v_2	∞	∞	4	∞	∞	∞
v_3	8	∞	∞	∞	∞	9
v_4	∞	∞	5	∞	∞	6
v_5	∞	∞	∞	5	∞	∞
v_6	3	∞	∞	∞	1	∞

空间复杂度 $O(n^2)$

采用邻接矩阵表示法创建无向网



算法: ① 输入总顶点数和边数
② 依次输入边的位置存入顶点表中
③ 初始化邻接矩阵

邻接矩阵好处: ① 直观, 简单, 好理解

② 方便检查任意一对顶点间是否存在边

③ 方便计算度, 任一顶点所有“邻接点”

缺点: ① 增加删除顶点麻烦

② 浪费空间

邻接表 (链式)

data firstarc 头结点

adjvex nextarc 表结点

邻接点无表中原号 权值

6.4.2 邻接表

1. 邻接表表示法 (链式)

头结点

	data	firstarc
0	v_1	4
1	v_2	2
2	v_3	3
3	v_4	0
4	v_5	1
5	v_6	2

表结点

adjvex	nextarc
4	1
2	0
3	0
0	2
1	5
2	5

- 顶点:
 - 按编号顺序将顶点数据存储在二维数组中;
 - 关联同一顶点的边 (以顶点为尾的弧):
 - 用线性链表存储

无向图

如果有向表, 邻接表只按出度

特点: ① 邻接表不唯一

② 若无向图中有 n 个顶点、 e 条边 则其邻接表需 n 个头结点和 $2e$ 个表结点 (无向)

边会重复

③ 有几个表结点就有几个边

有向图

邻接表

	v_1	v_2	v_3	v_4
0	2	1	3	0
1	2	1	3	0
2	2	1	3	0
3	2	1	3	0

看出度

有向 (只记录出度边)

出度: 每个点后的表件数相加

入度: 找所有表结点对应的值

eg 入度 v_1 : 找表结点 4 有几个

度 = 表结点
个数
+ (-) 1
个数

图理 有向邻接表 图只记录入度 (逆邻接表)

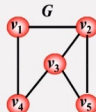
附---

邻接表存储结构

图的邻接表存储表示:

顶点的结点结构

data firstarc



0	v ₁	3	1	△
1	v ₂	4	2	0 △
2	v ₃	4	3	1 △
3	v ₄	2	0	△
4	v ₅	2	1	△

```
typedef struct VNode{
```

```
    VerType data;           //顶点信息
```

```
    ArcNode * firstarc;      //指向第一条依附该顶点的边的指针
```

```
}VNode, AdjList[MVNum]; //AdjList表示邻接表类型
```

弧 (边) 的结点结构

adjvex nextarc info

```
#define MVNum 100           //最大顶点数
```

```
typedef struct ArcNode{      //边结点
```

```
    int adjvex;              //该边所指向的顶点的位置
```

```
    struct ArcNode * nextarc; //指向下一条边的指针
```

```
    OtherInfo info;          //和边相关的信息
```

```
}ArcNode;
```

图的结构定义

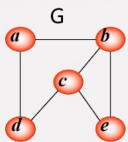
```
typedef struct {
```

```
    AdjList vertices; //vertices--vertex的复数
```

```
    int vexnum, arcnum; //图的当前顶点数和弧数
```

```
} ALGraph;
```

邻接表操作举例说明:



邻接表

0	a	3	1	△
1	b	4	2	0 △
2	c	4	3	1 △
3	d	2	0	△
4	e	2	1	△

ALGraph G;

G.vexnum=5; G.arcnum=5; //定义了邻接表表示的图G

G.vertices[1].data= 'b' ; //图G包含5个顶点, 5条边

p=G.vertices[1].firstarc; //图G中第2个顶点是b

p->adjvex =4; //指针p指向顶点b的第一条边结点

//p指针所指边结点是到下标为4的结点的边

算法思路

① 输入总顶点数
总边数

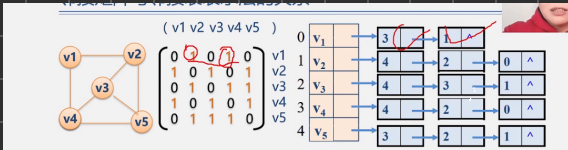
② 建立顶点表
依次输入点信息
顶点表
使每个无头结点的新
域初始化为Null

③ 创建邻接表
依次输入每条边依
附的两个顶点

邻接表特点

- ① 方便找任一顶点的所有“邻接点”
- ② 节约稀疏图的空间
- ③ 方便计算“度”

邻接矩阵与邻接表表示法



联系：邻接矩阵中每一行对应链表中的一行
 结点个数等于此行非0个数

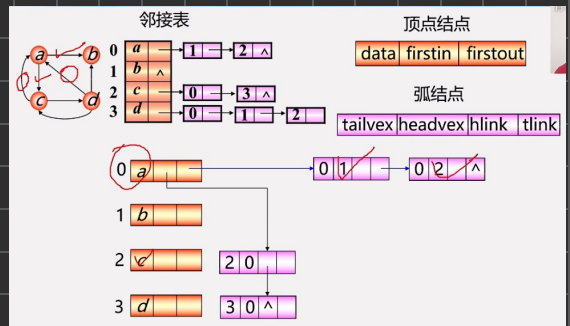
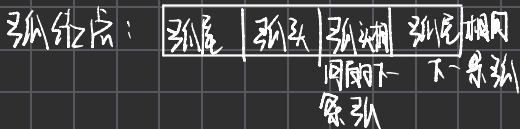
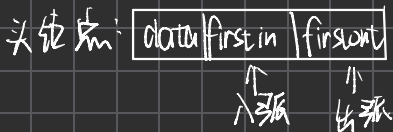
区别：① 对于任一确定的无向图，邻接矩阵唯一
 邻接表不唯一

② 邻接矩阵空间复杂度 n^2 而邻表 $O(n \cdot e)$
 (顶点数 + 边数)

十字链表：存储有图中的邻接表 + 逆邻接表

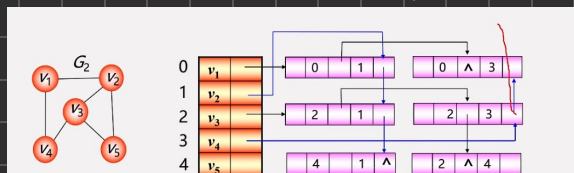
邻接多重表：

十字链表：



邻接矩阵 优点：容易求得顶点与边的信息

缺点：某些操作不方便（删除一边需找此边的两个顶点）



图的遍历实质：找每个顶点的邻接点过程

避免重复访问：建立数组 `visited[]` 每点
若被访问则+1

深度优先 — DFS

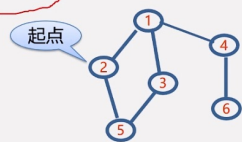
广度优先 — BFS

DFS：一条路走到黑，若无路则返回上一点去走未走过的路
直到访问了所有

（类似于树的先序遍历）

2. 深度优先搜索遍历算法的实现

邻接矩阵表示的无向图深度遍历实现：



辅助数组 `visited[n]`

	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

邻接矩阵

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	1	0	0	0	1	0
3	1	0	0	0	1	0
4	1	0	0	0	0	1
5	0	1	1	0	0	0
6	0	0	0	1	0	0

DFS 结果

2→1→3→5→4→6

若用邻接矩阵来图

遍历图中每个顶点
时间复杂度 $O(n^2)$

用邻接表 时间复杂度 $O(n+e)$

7 广度优先遍历 - BFS

- 访问顶点的所有邻接点，再访问这些邻接点的邻接点
(类似层次遍历)

同 DFS 时间复杂度，矩阵 $O(n^2)$ 表 $O(n \cdot e)$

生成树：所有顶点均由边连接在一起，但不存在回路的图

- 生成树顶点个数与图相同
- 生成树是图的极小连通子图

最小生成树：给定一个无向网络，使权值和最小的那棵树称为该网的最小生成树。
使权值最小

两种算法比较		
算法名	普里姆算法	克鲁斯卡尔算法
算法思想	选择点	选择边
时间复杂度	$O(n^2)$ (n 为顶点数)	$O(e \log e)$ (e 为边数)
适应范围	稠密图	稀疏图

最短路径：有向网中 A 点到 B 点的所有路径中寻找一条
(路径之和 最短) 各边权值之和最小的路径。

第一类：从一点到一点 (迪杰斯特拉)

- ① 初始化，找出源点 V_0 到各终点 V_k 的路径。
- ② 从这些路径找出一条长度最短路径。

第二类：从一点到其他点

有向无环图：DAG图

AOV网：用一个有向图表示活动的网。顶点表示活动，弧表示活动之间（拓扑排序）优先制约关系。

AOE网：用一个有向图表示活动的网。弧表示活动，顶点表示活动（关键路径）开始或结束事件。

AOV网：从 i 到 j 有一条有向路径， i 是 j 的前驱， j 是 i 的后继
不允许有回路!!!

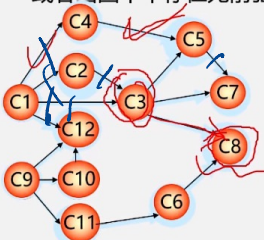
拓扑排序

在AOV网没有回路的前提下，我们将全部活动排列成一个线性序列，使得若AOV网中有弧 $\langle i, j \rangle$ 存在，则在这个序列中， i 一定排在 j 的前面，具有这种性质的线性序列称为拓扑有序序列，相应的拓扑有序排序的算法称为拓扑排序。

拓扑排序方法：

- ① 在有向图中选一个没有前驱的顶点且输出之。
- ② 从图中删除该顶点和所有以它为尾的弧（即删掉它发出的弧）
- ③ 重复上述操作直至输出。

或者当图中不存在无前驱的顶点为止



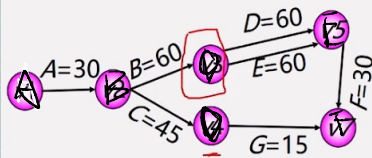
拓扑序列：

C1, C2, C3, C4, C5, C7, C9, C10, C11,
C6, C12, C8
C9, C10, C11, C6, C1, C12, C4, C2,
C3, C5, C7, C8

一个AOV网的拓扑序列不是唯一的

关键路径：用边表示活动网络，用顶点表示事件，弧表示活动，弧的权表示持续时间
(AOE网)

活动代码	活动描述	历时(分钟)	前置任务
A	菜单制定	30	
B	原料采购	60	A
C	餐具准备	45	A
D	甜点准备	60	B
E	原料清洗	60	B
F	烹饪	30	C, D, E
G	桌椅布置	15	C
H	宴会开始	0	FG



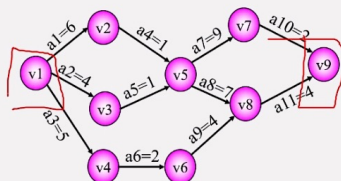
对于AOE网，我们关心两个问题：

- (1) 完成整项工程至少需要多少时间？
- (2) 哪些活动是影响工程进度的关键？



关键路径 —— 路径长度最长的路径。

路径长度 —— 路径上各活动持续时间之和。

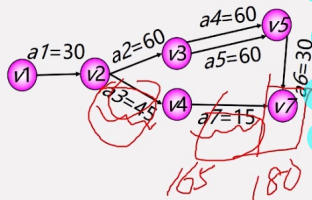


V_i, V_j 表示事件
最早开始时间

$$Ve(V_2) = 6$$

$$Ve(V_1) = 0$$

如何确定关键路径，需要定义4个描述量：



$ve(v_j)$ —— 表示事件 v_j 的最早发生时间。

例： $ve(v_1) = 0$ $ve(v_2) = 30$

$vl(v_j)$ —— 表示事件 v_j 的最迟发生时间。

例： $vl(v_4) = 165$

$e(i)$ —— 表示活动 a_i 的最早开始时间。

例： $e(a_3) = 30$

$l(i)$ —— 表示活动 a_i 的最迟开始时间。

查找表：同一类型的数据元素的集合，数据元素之间存在着松散的联系。

主关键字：唯一地标识

次关键字：识别若干

静态查找表：仅作“查询”操作

动态查找表：作“插入”和“删除”操作的查找表

评价查找算法： $ASL = \frac{\text{平均查找长度}}{\text{比较次数}} \times \text{总个数}$

线性表的查找 { 顺序查找 (线性查找)
折半查找 (二分或对分查找)
分块查找

散列表：通常有顺序表来有诔，存储位置与关键字有关

[顺序表：紧凑

散列表：分散

散列函数 (h, hash): 集合元素的关键字 (key) 与其存储位置 (Loc) 之间的关系函数，散列函数计算出的值也称为散列值。

$\text{Loc}(\text{key})$: 表示关键字值为 key 的集合元素的存储地址。

散列表 (hash table , 哈希表): 用散列函数建立起来的表，用于存储集合元素。

定义域: 关键字取值范围 (1)
值域: 散列表地址范围 (2)

冲突:

key	BEIJING	JIANGSU	SHANGHAI	SICHUAN	JIANGXI	TIANJIN	SHANXI
$h_1(\text{key})$	02	10	19	19	10	20	19
$h_2(\text{key})$	09	01	28	03	19	04	28

发现一:

对 h_1 : JIANGSU 与 JIANGXI 等被映射到相同位置，产生冲突

对 h_2 : SHANGHAI 与 SHANXI 被映射到相同位置，产生冲突

冲突: $\text{key}_1 \neq \text{key}_2$, 但 $h(\text{key}_1) = h(\text{key}_2)$ 的现象

同义词: 对给定 h , 具有相同散列值的不同关键字

办法: ① 选择好的 "h"

② 用冲突处理技术

散列函数选取:

① 确定性: 同一值映射到同一地址

② 快速: 最好是 $O(1)$

③ 满射: 尽可能充分覆盖整个散列表存储空间

④ 均匀分布: 使映射到各个位置的概率接近
避免出现元素聚集扎堆现象

除、留余数法

$$h(key) = key \% M \quad (M \text{ 为散列表长})$$

↓改进: $MAID$

$$h(key) = (key \times a + b) \% P$$

($a > 0, b > 0, P$ 为不超过 M 最大素数)

(b 作为偏移量, 消除了不动点)

a 作为间隔量, 使原本相邻地址隔了 a 个单位)

平方取中法

$$h(key) = (key)^2 \text{ 的中间若干位 } (R) \rightarrow$$

$$10^{k-1} \leq n \leq 10^k$$

$h(key) = (key)^2$ 的中间若干位 k

其中: 位数 k 应满足: $10^{k-1} < n \leq 10^k$, n 为集合中元素个数。

例: $n=765, 10^{3-1} < 765 \leq 10^3$, 故 $k=3$

关键字 (内部编码) ² 散列地址

KEYA 122157778355001 778

KEYB 122157800460004 800

AKEY 001233265775625 265

BKEY 004454315775625 315

折叠法

折叠法

1. 把关键字值自左到右，分成位数相等的几部分；每部分位数与散列表地址位数相同，最后一部分的位数可以短些；
2. 把这些部分的数据叠加起来，得到该关键字的散列地址。

例：设关键字key=12320324111220，散列地址取3位，
则key被划分5段：123 203 241 112 20

折叠法

例：设关键字key=12320324111220，散列地址取3位，
则key被划分5段：123 203 241 112 20

123	123
203	302
241	241
112	211
+ 20	+ 20
699	897

移位法

分界法

数字分析法

数字分析法

观察关键字编码组合规律，取分布均匀的若干位。

位	1	2	3	4	5	6
关键字	9	4	2	1	4	8
	9	4	2	3	5	6
	9	4	2	5	7	2
	9	4	2	6	6	4
	9	4	3	3	9	5
	9	4	2	4	7	2
	9	4	2	7	3	1
	9	4	1	2	8	7
	9	4	2	3	4	5

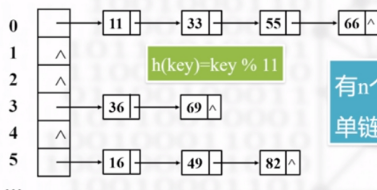
可以看出第4、5、6位取值分布均匀
故取第4、5、6三位为散列函数值

冲突处理技术

1. 拉链法 (开散列法)

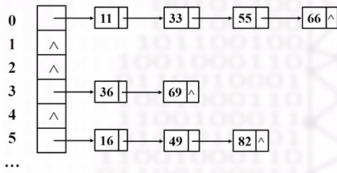
拉链法

- 所有关键字互为同义词的集合元素存储在同一个单链表上
- 所有单链表的头指针存入一个长度为M的散列表中
- 单链表的头指针在散列表中的存储位置通过散列函数计算得到



有n个元素的散列表，其每个
单链表的平均长度为n/M。

拉链法



集合元素的插入操作

- 执行元素查找操作
- 查找不成功，创建新结点
- 执行单链表结点插入操作

如果元素在单链表上无序排列，则必须先遍历链表上所有结点才能确定不重复

如果元素在单链表上有序，则可以提前结束查找，并在查找失败处插入新结点

平均时间复杂度 $O(n)$ ^{元素个数}
空间复杂度 $O(n)$

优点：①简单，无聚集现象，平均查找长度较
②适用于链表前无法确定表长的情况

缺点：指针需要额外有存储空间，当数据元素已有存储空间小时
空间使用率低。

开放地址法

线性探查
二次探查
双散列法

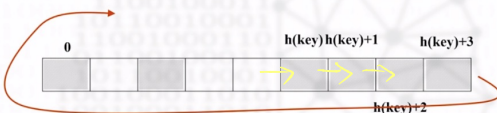
线性探查法

线性探查法

线性探查法使用的探查序列如下：

$h(key), h(key)+1, h(key)+2, \dots, M-1, 0, 1, \dots, h(key)-1$

插入元素时，从 $h(key)$ 开始，若被占用，检查 $h(key)+1$ ，若也占用，再检查探测序列中的下一个位置，直到某个位置为空时，将关键字插入该位置。



散列表的搜索操作

基本思想：从基位置 $h(key)$ 开始，按照线性探查序列查找元素

搜索成功：找到关键字值为 key 的元素

搜索失败：1、遇到一个空位置

2、探查至 $h(key)$ 前一个位置（表满）

改进的散列表删除操作

删除时需考虑两点：

1. 不能简单清除元素，否则会**隔离**探查序列后面的元素，影响搜索；
2. 删除元素后，该位置能够重新使用。

解决的办法：

1. 为每个位置增加标志域empty，表示该位置是否使用过
2. 删除元素时，不改变标志位，仍为false，元素值改为NeverUsed

0	1	2	3	4	5	6	7	8	9	10
NU	NU	24	80	58	35	NU	40	NU	NU	65
T	T	F	F	F	F	T	F	T	T	F

① 未使用过，但标志位就是删除过

缺点：①很快表中T都聚下
②易使线性聚集

搜索时遇到 NU 仍继续
要遇到 NU 或回到 T 内啊

二次探查法

二次探查法的探查序列

$h(key), h_1(key), h_2(key), \dots, h_{2i-1}(key), h_{2i}(key), \dots$

探查序列由下列函数得到
 $h_{2i-1}(key) = (h(key) + i^2) \% M$
 $h_{2i}(key) = (h(key) - i^2) \% M$
 $i = 1, 2, \dots, (M-1)/2$

左右两边同探

$h(40) = 7$

$$i=1 \quad h_1(40) = (7+1^2) \% 11 = 8$$

$$h_2(40) = (7-1^2) \% 11 = 6$$

$$i=2 \quad h_3(40) = (7+2^2) \% 11 = 0$$

$$h_4(40) = (7-2^2) \% 11 = 3$$

$$i=3 \quad h_5(40) = (7+3^2) \% 11 = 5$$

$$h_6(40) = (7-3^2) \% 11 = 9$$

$$i=4 \quad h_7(40) = (7+4^2) \% 11 = 1$$

$$h_8(40) = (7-4^2) \% 11 = 2$$

$$i=5 \quad h_9(40) = (7+5^2) \% 11 = 10$$

$$h_{10}(40) = (7-5^2) \% 11 = 4$$

PS: 先加后减

因此不同之间有相同探查序列

缺点：① 造成二次聚集

双散列法

双散列法的探查序列

具备两个散列函数 h_1 和 h_2 ，探查序列为：

$h_1(\text{key}), (h_1(\text{key}) + h_2(\text{key})) \% M, (h_1(\text{key}) + 2h_2(\text{key})) \% M, \dots,$

$(h_1(\text{key}) + i * h_2(\text{key})) \% M, \dots$

h_2 的作用：

- 对 h_1 散列值产生一个固定增量，实现跳跃式探查
- 改善“二次聚集”，对两个散列函数都为同义词的关键字很少

h_1 用于计算基地址
 h_2 用于计算偏移量

双散列法实例

$(h_1(\text{key}) + i * h_2(\text{key})) \% M, i=0, 1, \dots, M-1$

$h_1(\text{key}) = \text{key} \% 11$ $h_2(\text{key}) = \text{key} \% 9 + 1$

0	1	2	3	4	5	6	7	8	9	10
		24	80	15						65

插入58

$h_1(\text{key}) = 58 \% 11 = 3$

$h_2(\text{key}) = 58 \% 9 + 1 = 5$

$(h_1(\text{key}) + h_2(\text{key})) \% 11 = (3 + 5) \% 11 = 8$

