



《数据结构》 期末复习

- 期末考试题型：与期中样卷类似，题目主要是算法应用题和设计题，再加上一些小题
- 考试范围：第一章到第十章（除掉第八章、第十一章、第十二章的内容）
- 算法题：面向问题，解决问题。作答最好是先给出算法描述，再给出具体的程序（可以利用书上的数据结构和基本操作）。
- 小题：算法的例子，比如哈曼树，折半查找，HASH表，二叉排序树，B树等等
- 期末成绩：平时成绩+期末成绩

- 数据结构的章节结构及重点构成
- 数据结构学科的章节划分基本上为：概论，线性表，栈和队列，串，多维数组和广义表，树和二叉树，图，查找，内排，外排，文件，动态存储分配。
- 对于绝大多数的学校而言，“外排，文件，动态存储分配”三章基本上是不考的，在大多数高校的计算机本科教学过程中，这三章也是基本上不作讲授的。所以，大家在这三章上可以不必花费过多的精力，只要知道基本的概念即可。但是，对于报考名校特别是该校又有在试卷中对这三章进行过考核的历史，那么这部分同学就要留意这三章了。

■ 对于每一章，要清楚

- 1 想解决的问题
- 2 基本的数据结构
- 3 基本的操作
- 4 算法（特别是经典算法）的解决思路，程序实现及时间复杂性

经典算法比如：有序表的归并，**KMP**算法，矩阵的快速转置，二叉树（递归或非递归）的遍历，图的（广度优先和深度优先）遍历，最小生成树，最短路径，折半查找，冒泡排序，快速排序等等

- **概论**：内容很少，概念简单，分数大多只有几分，有的学校甚至不考。
- **线性表**：基础章节，必考内容之一。考题多数为基本概念题，名校考题中，鲜有大型算法设计题。如果有，也是与其它章节内容相结合。
- **栈和队列**：基础章节，容易出基本概念题，必考内容之一。而栈常与其它章节配合考查，也常与递归等概念相联系进行考查。
- **串**：基础章节，概念较为简单。专门针对于此章的大型算法设计题很少，较常见的是根据**KMP**进行算法分析。
- **多维数组及广义表**：基础章节，基于数组的算法题也是常见的，分数比例波动较大，是出题的“可选单元”或“侯补单元”。一般如果要出题，多数不会作为大题出。数组常与“查找，排序”等章节结合来作为大题考查。

- **树和二叉树**：重点难点章节，各校必考章节。各校在此章出题的不同之处在于，是否在本章中出一到两道大的算法设计题。通过对多所学校的试卷分析，绝大多数学校在本章都曾有过出大型算法设计题的历史。
- **图**：重点难点章节，名校尤爱考。如果作为重点来考，则多出现于分析与设计题型当中，可与树一章共同构成算法设计大题的题型设计。
- **查找**：重点难点章节，概念较多，联系较为紧密，容易混淆。出题时可以作为分析型题目给出，在基本概念型题目中也较为常见。算法设计型题中可以数组结合来考查，也可以与树一章结合来考查。
- **排序**：与查找一章类似，本章同属于重点难点章节，且概念更多，联系更为紧密，概念之间更容易混淆。在基本概念的考查中，尤爱考各种排序算法的优劣比较之类的题。算法设计大题中，如果作为出题，那么常与数组结合来考查。

第一章 概述

- 本章主要起到总领作用，为读者进行数据结构的学习进行了一些先期铺垫。大家主要注意以下几点：数据结构的基本概念，时间和空间复杂度的概念及度量方法，算法设计时的注意事项。本章考点不多，只要稍加注意理解即可。
- 主要考时间复杂性的计算，时间复杂性的比较

第一章 概述——逻辑结构

- 数据结构包括**逻辑结构**和**物理结构**
- 数据之间的相互关系称为**逻辑结构**。通常分为四类基本结构：
 - **集合**：结构中的数据元素除了同属于一种类型外，别无其它关系。
 - **线性结构**：结构中的数据元素之间存在一对一的关系。
 - **树型结构**：结构中的数据元素之间存在一对多的关系。
 - **图状结构或网状结构**：结构中的数据元素之间存在多对多的关系。

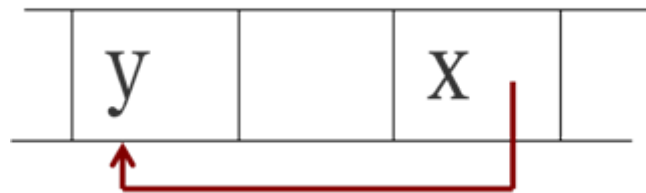
第一章 概述——存储结构

逻辑结构在存储器中的映像：数据元素映像和关系的映像

- 数据结构在计算机中的表示称为数据的**物理结构**，又称为**存储结构**。
- **数据元素的映象方法**：用二进制位(**bit**)的位串表示数据元素
$$(321)_{10} = (501)_8 = (101000001)_2$$
$$A = (101)_8 = (001000001)_2$$
- **数据元素之间的关系**在计算机中有两种不同的表示方法：顺序表示和非顺序表示——由此得出两种不同的存储结构：**顺序存储结构**和**链式存储结构**

第一章 概述——存储结构

- **顺序存储结构：**用数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系。
- **链式存储结构：**在每一个数据元素中增加一个存放地址的指针(**pointer**)，用此指针来表示数据元素之间的逻辑关系。



第一章 概述——数据类型 vs. 抽象数据类型

- 在不同的编程环境中，**存储结构可有不同的描述方法**。
——当用高级程序设计语言进行编程时，通常可用高级编程语言中提供的**数据类型**描述之。
- **数据类型**：在一种程序设计语言中，变量所具有的数据种类。
- 例1：在**FORTRAN**语言中，变量的数据类型有整型、实型、和复数型
- 例2：在**C**语言中
 - 数据类型：基本类型和构造类型
 - 基本类型：整型、浮点型、字符型、枚举型、指针、空类型
 - 构造类型：数组、结构、联合、自定义

第一章 概述——数据类型 vs. 抽象数据类型

- 抽象数据类型(**Abstract Data Type** 简称**ADT**):
一个数学模型以及定义在该模型上的一组操作。
——抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关。
- 抽象数据类型实际上就是对数据结构的表示以及在此结构上的一组算法。用三元组描述如下：
 (D, S, P)
D是数据对象，**S**是**D**上的关系集，**P**是对**D**的基本操作集，见例**P9**

第一章 概述——时间复杂度

- 一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，算法的时间量度记作 $T(n)=O(f(n))$ ，称作算法的渐近时间复杂度，简称**时间复杂度**。
- 显然，被称做基本操作的原操作应是其重复执行次数和算法的执行时间成正比的原操作，多数情况下它是**最深层循环内的语句中的原操作**，它的执行次数和包含它的语句的频度相同。
- 语句的频度：是指该语句重复执行的次数。

■ 例1: `for(i=1;i<=n;++i)`
 `for(j=1;j<=n;++j)`
 `{++x;s+=x;}`

语句频度为 n^2 ，其时间复杂度为： $O(n^2)$ ，即时间复杂度为平方阶。

■ 例2: `for(i=2;i<=n;++i)`
 `for(j=2;j<=i-1;++j)`
 `{++x;a[i, j]=x;}`

语句`++x`的执行次数关于 n 的增长率为 n^2 ，它是语句频度表达式中增长最快的项。

总的语句频度为：

$$1+2+3+\dots+n-2=(1+n-2)\times(n-2)/2=(n-1)(n-2)/2$$
$$=(n^2-3n+2)/2$$

此时，时间复杂度为： $O(n^2)$ ，即时间复杂度为平方阶。

第二章 线性表

- 作为线性结构的开篇章节，线性表一章在线性结构的学习乃至整个数据结构学科的学习中，其作用都是不可低估的。在这一章，第一次系统性地引入链式存储的概念，**链式存储概念将是整个数据结构学科的重中之重**，无论哪一章都涉及到了这个概念。
- 总体来说，线性表一章可供考查的重要考点有以下几个方面：
 - 1.线性表的相关基本概念**，如：前驱、后继、表长、空表、首元结点，头结点，头指针等概念。
 - 2.线性表的结构特点**，主要是指：除第一及最后一个元素外，每个结点都只有一个前趋和只有一个后继。
 - 3.线性表的顺序存储方式及其在具体语言环境下的两种不同实现**：**表空间的静态分配**和**动态分配**。静态链表与顺序表的相似及不同之处。

4.线性表的链式存储方式及以下几种常用链表的特点和运算

单链表、循环链表，双向链表，双向循环链表。其中，单链表的归并算法、循环链表的归并算法、双向链表及双向循环链表的插入和删除算法等都是较为常见的考查方式。此外，近年来在不少学校中还多次出现要求用递归算法实现单链表输出（可能是顺序也可能是倒序）的问题。

在链表的小题型中，经常考到一些诸如：**判表空的题**。在不同的链表中，其判表空的方式是不一样的，请大家注意。

5.线性表的顺序存储及链式存储情况下，其不同的优缺点比较，即其各自适用的场合

单链表中设置头指针、循环链表中设置尾指针而不设置头指针的各自好处。

第二章 线性表——链表

- 线性表的顺序表示的特点是用物理位置上的邻接关系来表示结点间的逻辑关系，这一特点使我们可以随机存取表中的任一结点，但它也使得插入和删除操作会移动大量的结点。**为避免大量结点的移动**，引入线性表的另一种存储方式，链式存储结构，简称为**链表 (Linked List)**。
- **链表**是指用一组**任意的存储单元**来依次存放线性表的结点，这组存储单元即可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- 因此，链表中结点的**逻辑次序**和**物理次序**不一定相同。

第二章 线性表——单链表的归并

Void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc)

{ //已知单链线性表La和Lb的元素按值非递减排列。

//归并La和Lb得到新的单链线性表Lc，Lc的元素也按值非递减排列。

pa=La->next; pb=Lb->next;

Lc=pc=La; //用La的头结点作为Lc的头结点

While (pa&&pb) {

If (pa->data<=pb->data){

pc->next=pa; pc=pa; pa=pa->next;

}

Else {pc->next=pb; pc=pb; pb=pb->next;}

}

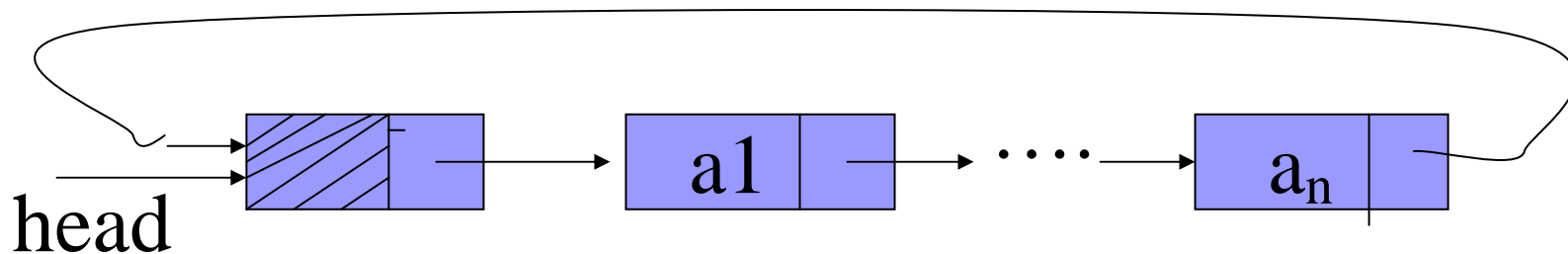
Pc->next=pa?pa:pb; //插入剩余段

Free(Lb); //释放Lb的头结点

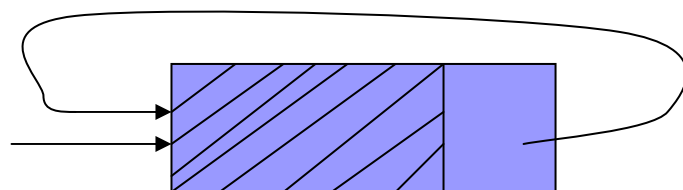
}//MergeList_L

第二章 线性表——循环链表的判空

- 循环链表是一种头尾相接的链表。其特点是无须增加存储量，仅对表的链接方式稍作改变，即可使得表处理更加方便灵活。
- **单循环链表：**在单链表中，将终端结点的指针域NULL改为指向表头结点或开始结点，就得到了单链形式的循环链表，并简单称为单循环链表。
- 为了使空表和非空表的处理一致，循环链表中也可设置一个头结点。这样，空循环链表仅有一个自成循环的头结点表示。如下图所示：



(1) 非空表



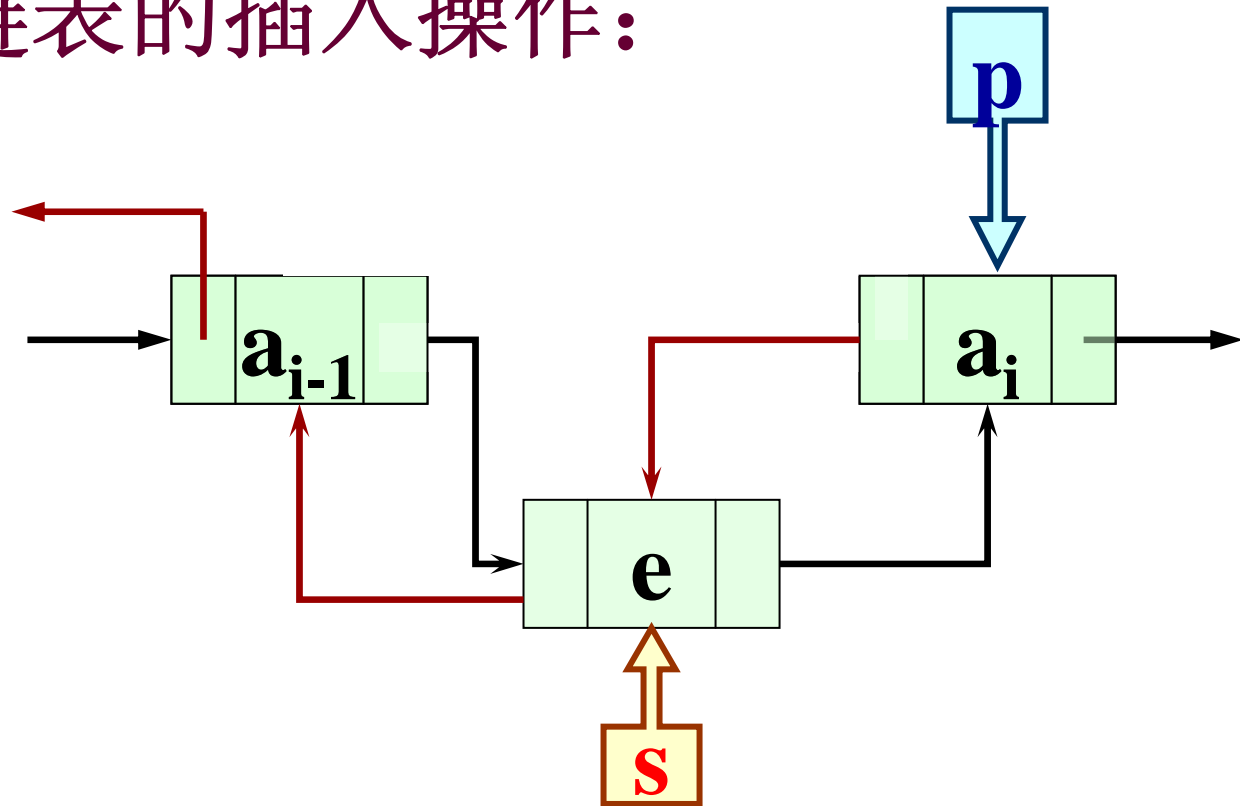
(2) 空表

- 在用头指针表示的单链表中，找开始结点 a_1 的时间是 $O(1)$ ，然而要找到终端结点 a_n ，则需从头指针开始遍历整个链表，其时间是 $O(n)$

第二章 线性表——循环链表的判空

- 头指针表示的单循环链表对表的操作不够方便
 - 因为表的操作常常是在表的首尾位置上进行
- 改用尾指针rear来表示单循环链表
 - 查找开始结点 a_1 和终端结点 a_n 都会方便，它们的存储位置分别是 $(\text{rear} \rightarrow \text{next}) \rightarrow \text{next}$ 和 rear ，显然，查找时间都是 $O(1)$ 。
- 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 $p \rightarrow \text{next}$ 是否为空，而是判断它们是否等于某一指定指针，如头指针或尾指针等。

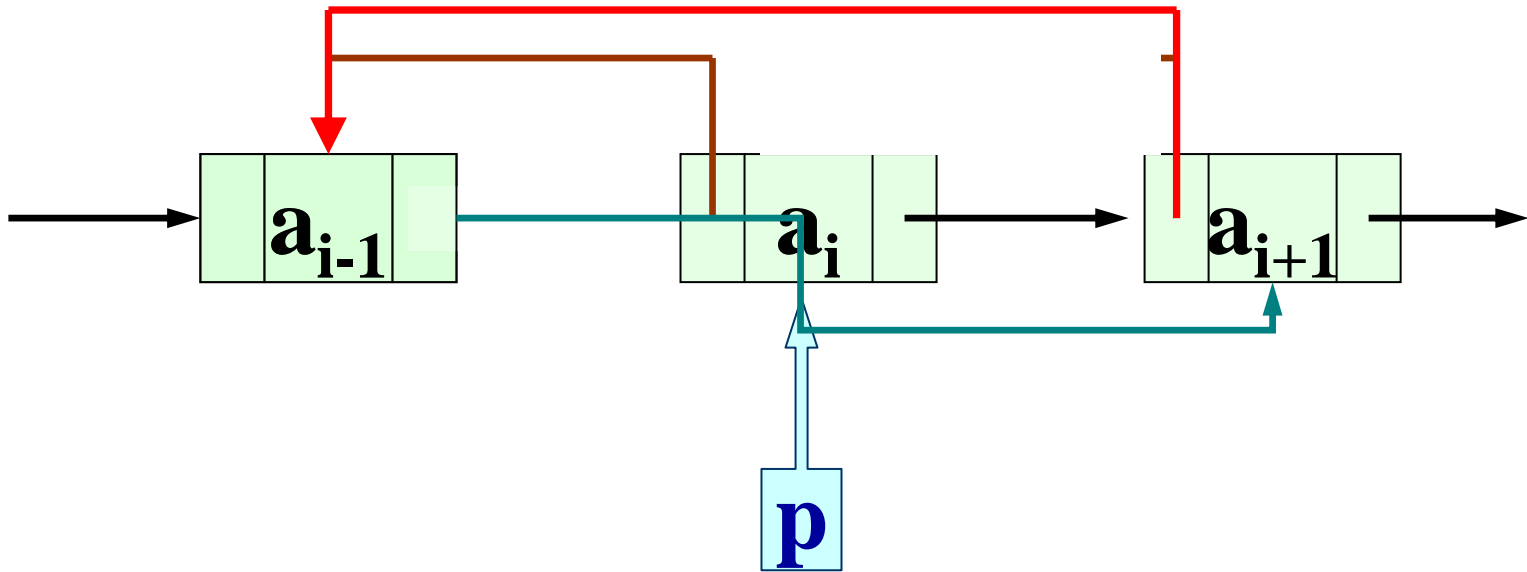
双向链表的插入操作:



$s \rightarrow \text{prior} = p \rightarrow \text{prior};$ $p \rightarrow \text{prior} \rightarrow \text{next} = s;$

$s \rightarrow \text{next} = p;$ $p \rightarrow \text{prior} = s;$

双向链表的删除操作:



$p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$

$p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$

第二章 线性表

——顺序存储及链式存储的优缺点比较

■ 链表的优点:

- 空间的合理利用;
- 插入、删除时不需移动

——线性表的首选存储结构

■ 链表的缺点:

- 求线性表的长度时不如顺序存储结构
- 数据元素在线性表中的“位序”的概念已淡化

■ 为此, 从实际应用出发重新定义一个带头结点的线性链表类型P37

第三章 栈和队列

- 栈与队列，是很多学习DS的同学遇到第一只拦路虎，很多人从这一章开始坐晕车，一直晕到现在。所以，理解栈与队列，是走向DS高手的一条必由之路，。
- 学习此章前，你可以问一下自己是不是已经知道了以下几点：
 1. **栈、队列的定义及其相关数据结构的概念**，包括：顺序栈，链栈，共享栈，循环队列，链队等。栈与队列**存取数据**（请注意包括：存和取两部分）的特点。
 2. **递归算法**。栈与递归的关系，以及借助栈将递归转向于非递归的经典算法： $n!$ 阶乘问题，fib数列问题，hanoi问题，背包问题，二叉树的递归和非递归遍历问题，图的深度遍历与栈的关系等。其中，涉及到树与图的问题，多半会在树与图的相关章节中进行考查。
 3. **栈的应用**：数值表达式的求解，括号的配对等的原理，只作原理性了解，具体要求考查此为题目的算法设计题不多。
 4. **循环队列中判队空、队满条件，循环队列中入队与出队算法。**
 5. **栈的InitStack, Push, Pop, StackEmpty等基本操作，队列的InitQueue, QueueEmpty, EnQueue, DeQueue等基本操作**
- 如果你已经对上面的几点了如指掌，栈与队列一章可以不看书了。注意，我说的是可以不看书，并不是可以不作题哦。

通常称，栈和队列是限定插入和删除只能在表的“端点”进行的线性表。

线性表

Insert(L, **i**, x)

$1 \leq i \leq n+1$

Delete(L, **i**)

$1 \leq i \leq n$

栈

Insert(S, **n+1**, x)

Delete(S, **n**)

队列

Insert(Q, **n+1**, x)

Delete(Q, **1**)

栈和队列是两种常用的数据类型

第三章 栈和队列——顺序栈

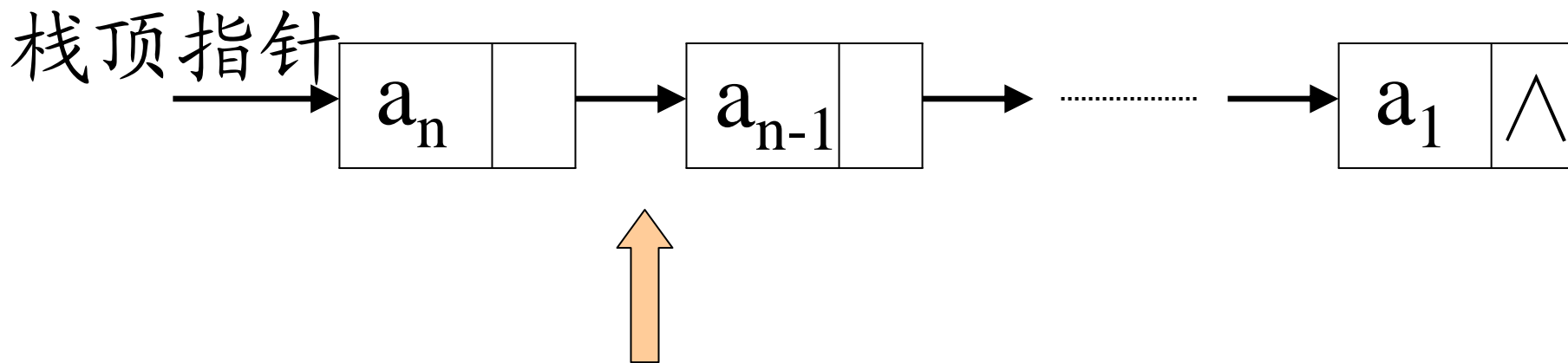
- 由于栈是运算受限的线性表，因此线性表的**存储结构**对栈也适应。
- 栈的顺序存储结构简称为**顺序栈**，它是**运算受限的线性表**。因此，**可用数组来实现顺序栈**。因为栈底位置是固定不变的，所以可以将栈底位置设置在数组的两端的任何一个端点；栈顶位置是随着进栈和退栈操作而变化的，故需用一个整型变量top来表示。

第三章 栈和队列——顺序栈

- 设S是**SqStack**类型的指针变量。
 - 若栈底位置在向量的低端，即**s->base**是栈底元素，那么栈顶指针**s->top**是正向增加的，即进栈时需将**s->top**加1，退栈时需将**s->top**减1。
 - **s->top==s->base**表示空栈，当栈空时再做退栈运算也将产生溢出，简称“下溢”。
 - **s->top-s->base == stacksize**表示栈满。当栈满时再做进栈运算必定产生空间溢出，简称“上溢”。

第三章 栈和队列——链栈

- 栈的链式存储结构称为链栈，它是运算受限的单链表，插入和删除操作仅限制在表头位置上进行。
- 由于只能在链表头部进行操作，故链表没有必要像单链表那样附加头结点。栈顶指针就是链表的头指针。



注意：链栈中
指针的方向

第三章 栈和队列——循环队列（顺序队列）

- 和栈类似，队列中亦有上溢和下溢现象。此外，顺序队列中还存在“**假上溢**”现象。
- 因为在入队和出队的操作中，头尾指针**只增加不减小**，致使被删除元素的空间永远无法重新利用。
- 因此，尽管队列中实际的元素个数远远小于向量空间的规模，但也可能**由于尾指针已超出向量空间的上界而不能做入队操作**。该现象称为假上溢。
- 为充分利用向量空间，克服假上溢现象：
 - 将向量空间想象为一个**首尾相接的圆环**，并称这种向量为**循环向量**，存储在其中的队列称为**循环队列**（Circular Queue）。

第三章 栈和队列——循环队列（顺序队列）

- 在循环队列中进行出队、入队操作时，头尾指针仍要加1，朝前移动。只不过当头尾指针指向向量上界（**QueueSize-1**）时，其加1操作的结果是指向向量的下界0。

- 这种循环意义下的加1操作可以描述为：

if(i+1==QueueSize)

i=0;

else

i++;

利用模运算可简化为： **$i=(i+1)\%QueueSize$**

第三章 栈和队列——循环队列（顺序队列）

- 因为循环队列元素的空间可以被利用，除非向量空间真的被队列元素全部占用，否则不会上溢。因此，除一些简单的应用外，真正实用的顺序队列是循环队列。
- 由于入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等。因此，我们无法通过 $front=rear$ 来判断队列“空”还是“满”。

第三章 栈和队列——循环队列（顺序队列）

■ 解决此问题的方法至少有三种：

1. 另设一个标志位以区别队列的空和满；
2. 少用一个元素的空间，约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队满（注意：rear所指的单元始终为空）；
3. 其三是使用一个计数器记录队列中元素的总数（实际上是队列长度）。

第三章 栈和队列——循环队列（顺序队列）

入队算法:

```
Status EnQueue (SqQueue &Q, ElemType e) {  
    // 插入元素e为Q的新的队尾元素  
    if ((Q.rear+1) % MAXQSIZE == Q.front)  
        return ERROR; //队列满  
    Q.base[Q.rear] = e;  
    Q.rear = (Q.rear+1) % MAXQSIZE;  
    return OK;  
}
```

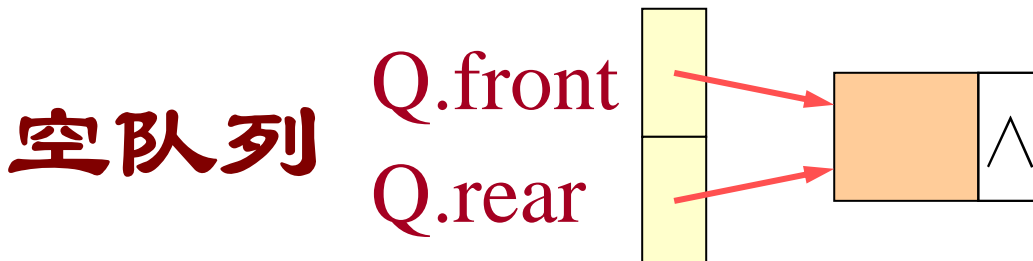
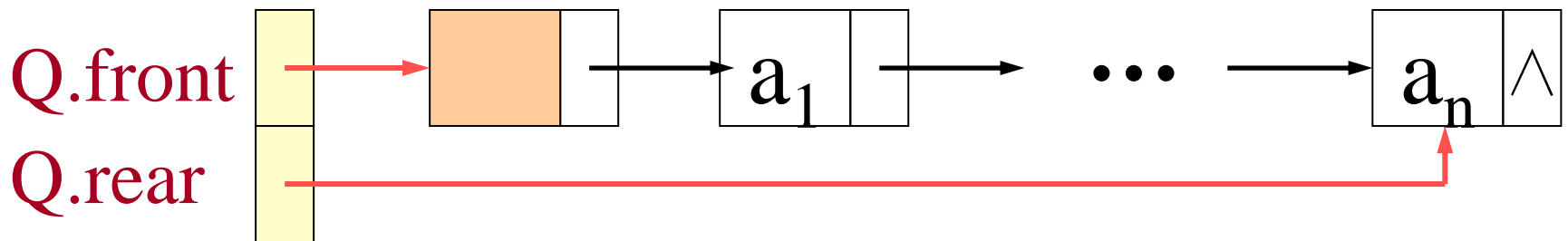
第三章 栈和队列——循环队列（顺序队列）

出队算法:

```
Status DeQueue (SqQueue &Q, ElemType &e) {  
    // 若队列不空，则删除Q的队头元素，  
    // 用e返回其值，并返回OK；否则返回ERROR  
    if (Q.front == Q.rear) return ERROR;  
    e = Q.base[Q.front];  
    Q.front = (Q.front+1) % MAXQSIZE;  
    return OK;  
}
```

第三章 栈和队列——链队列

- 队列的链式存储结构简称为**链队列**，它是限制**仅在表头删除和表尾插入**的单链表。
- 显然仅有单链表的头指针不便于在表尾做插入操作，为此再**增加一个尾指针**，指向链表的最后一个结点。于是，一个链队列由一个头指针唯一确定。
- 和顺序队列类似，我们也是将这两个指针封装在一起，将链队列的类型LinkQueue定义为一个结构类型。



第三章 栈和队列——递归

- 递归程序是一种特别的子函数调用，自己调用自己，但参数会越来越小，直到接近出口。
- 因此，递归程序，可以**实现非递归化**：
 - 1、尾递归可以使用循环实现非递归化，如 $n! = n * (n-1)!$
 - 2、其他递归可以使用栈来实现非递归化

第四章 串

- 经历了栈一章的痛苦煎熬后，终于迎来了串一章的柳暗花明。
- 串，在概念上是比较少的一个章节，也是最容易自学的章节之一，但正如每个过来人所了解的，KMP算法是这一章的重要关隘，突破此关后，走过去又是一马平川的大好山河。
- 串一章需要攻破的主要堡垒有：
 1. **串的基本概念**，串与线性表的关系（串是其元素均为字符型数据的特殊线性表），空串与空格串的区别，串相等的条件。
 2. **串的基本操作，以及这些基本函数的使用**，包括：取子串，串连接，串替换，求串长等等。运用串的基本操作去完成特定的算法是很多学校在基本操作上的考查重点。
 3. **顺序串与链串及块链串的区别和联系，实现方式。**
 4. **KMP算法思想**。KMP中next数组以及nextval数组的求法。明确传统模式匹配算法的不足，明确next数组需要改进之外。其中，理解算法是核心，会求数组是得分点。不用我多说，这一节内容是本章的重中之重。可能进行的考查方式是：求next和nextval数组值，根据求得的next或nextval数组值给出运用KMP算法进行匹配的匹配过程。

第四章 串

- 串 (**String**) 是零个或多个字符组成的**有限序列**。一般记作 **S** = “**a₁a₂a₃...a_n**”
 - 其中S是串名，引号括起来的字符序列是串值；
 - a_i ($1 \leq i \leq n$) 可以是字母、数字或其它字符；
 - 串中所包含的字符个数称为该串的长度。长度为零的串称为**空串** (**Empty String**)，它不包含任何字符。通常将仅由一个或多个空格组成的串称为**空白串** (**Blank String**)
- 注意：**空串和空白串的不同，例如 “ ” 和 “ ” 分别表示长度为1的空白串和长度为0的空串。

串和线性表的异同:

- 串的逻辑结构和线性表极为相似，区别仅在于串的数据对象约束为字符集。
- 然而，串的基本操作和线性表有很大差别。
 - 在线性表的基本操作中，大多以“单个元素”作为操作对象，例如：在线性表中查找某个元素等。
 - 在串的基本操作中，通常以“串的整体”作为操作对象。例如：在串中查找某个子串、插入一个子串等。

第四章 串——存储结构

- 因为串是特殊的线性表，故其存储结构与线性表的存储结构类似。只不过组成串的结点是单个字符。
- 串有三种机内表示方法，下面分别介绍。
 - 定长顺序存储表示
 - 堆分配存储表示
 - 串的链式存储结构

第四章 串——定长顺序存储表示

- 定长顺序存储表示, 也称为静态存储分配的顺应表。它是用一组地址连续的存储单元来存放串中的字符序列。
- 定长顺序存储结构可以直接使用定长的字符数组来定义, 数组的上界预先给出:

```
#define MAXSTRLEN 255
```

```
typedef unsigned char SString[MAXSTRLEN+1];
```

```
SString s; //s是一个可容纳255个字符的顺序串
```

- 特点: 串的实际长度可在这个预定义长度的范围内随意设定, 超过预定义长度的串值则被舍去, 称之为“截断”。

串长的计算??

第四章 串——定长顺序存储表示

- 0号单元存放串的长度。
- 在串值后面加一个不计入串长的结束标记字符：
 - 例如，C语言中以字符 `'\0'` 表示串值的终结，这就是为什么在上述定义中，串空间最大值`MAXSTRLEN+1`为256，但最多只能存放255个字符的原因，因为必须留一个字节来存放 `'\0'` 字符。
- 若不设终结符，也可用一个整数来表示串的长度，那么该长度减1的位置就是串值的最后一个字符的位置。此时顺序串的类型定义和顺序表类似：

```
typedef struct{  
    char ch[MAXSTRLEN];  
    int length;
```

```
}SString; //其优点是涉及到串长操作时速度快。
```

第四章 串——堆分配存储表示

- **特点：**仍以一组地址连续的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配而得。所以也称为**动态存储分配的顺序表**。
- 在C语言中，利用**malloc()**和**free()**等动态存储管理函数，来根据实际需要动态分配和释放字符数组空间。称串值共享的存储空间为“**堆**”。这样定义的顺序串类型也有两种形式。

```
typedef char *string;
```

//c中的串库相当于此类型定义，或者再额外加入一个长度变量。

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

第四章 串——链式存储结构

- 顺序串上的插入和删除操作不方便，需要移动大量的字符。因此，我们可用单链表方式来存储串值，串的这种链式存储结构简称为链串。

```
typedef struct node{  
    char data;  
    struct node *next;  
}LString;
```

- 一个链串由头指针唯一确定。
- 这种结构便于进行插入和删除运算，但存储空间利用率太低。

第四章 串——链式存储结构

- 为了提高存储密度，可使每个结点存放多个字符。通常将结点数据域存放的字符个数定义为**结点的大小**，显然，
 - 当结点大小大于1时，串的长度不一定正好是结点大小的整数倍，因此要用特殊字符来填充最后一个结点，以表示串的终结。
- 对于结点大小不为1的链串，其类型定义只需对上述的结点类型做简单的修改即可。

```
#define CHUNKSIZE 80 // 可由用户定义的块大小
```

```
typedef struct Chunk{ // 结点结构
```

```
    char data[CHUNKSIZE];
```

```
    struct Chunk *next;
```

```
}Chunk;
```

```
typedef struct{ // 串的链表结构
```

```
    Chunk *head, *tail; // 串的头和尾指针
```

```
    int curlen; // 串当前长度
```

```
}LString;
```

第四章 串——链式存储结构

- 在处理串的联结操作时，要注意处理第一个串尾的无效字符
- 使用块链来表示串，要考虑串值的存储密度
$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$
可能浪费的空间包含头结点、指针域和无效字符。
- 块的大小如果设得太大，则浪费空间。块的大小如果设得太小，则访问效率低下。因此，要针对具体的应用场合，选择合适的大小。例如扇区大小为512个字节。
- 块链的插入和删除，同样要涉及到移动元素，其实也具有顺序串的缺点。所以，块链方式不大实用。
- 块链的操作，与链表的操作类似，不作详细讨论。

第四章 串——模式匹配算法

- 子串的定位操作又称为模式匹配（**Pattern Matching**）或串匹配（**String Matching**），其中子串**T**被称为模式串。
- 此操作的应用非常广泛。很多软件，若有“编辑”菜单项的话，则其中必有“查找”子菜单项。
 - 例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

第四章 串——模式匹配算法

首先，回忆一下串匹配(查找)的定义：

INDEX (S, T, pos)

初始条件：串S和T存在，T是非空串，

$1 \leq \text{pos} \leq \text{StrLength}(S)$ 。

操作结果：若主串S中存在和串T值相同的子串返回它的主串S中第pos个字符之后第一次出现的位置；否则函数值为0。

第四章 串——朴素模式匹配算法（穷举法）

- **基本思想（同算法4.1）**：从主串**S**的第**pos**个字符起和模式**T**的第一个字符比较，若相等，则继续逐个比较后继字符，否则从主串的下一个字符起重新和模式**T**的字符比较。依次类推，直到找到匹配成功，或匹配失败。

第1趟 S
T

$i=1=2=3$
a b b a b a
a b a
 $j=1=j=2=j=3$

第2趟 S
T

$i=2$
a b b a b a
a b a
 $j=1$

第3趟 S
T

$i=3$
a b b a b a
a b a
 $j=1$

第4趟 S
T

$i=4=i=5=6=i=7$
a b b a b a
a b a ✓
 $j=1=j=2=j=3=j=4$

返回 $i=4$

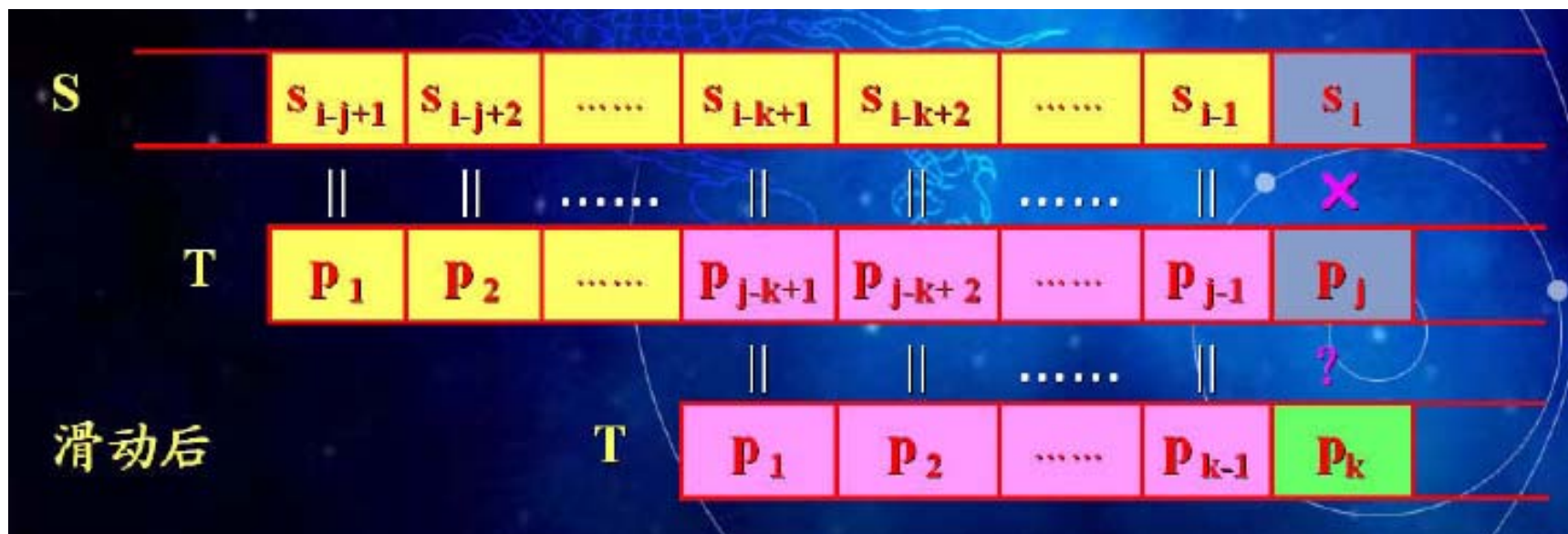
第四章 串——朴素模式匹配算法（穷举法）

- [illegible]

第四章 串——改进的模式匹配算法（KMP算法）

- **D.E.Knuth**与**V.R.Pratt**和**J.H.Morris**同时发现的，故简称为**KMP**算法
- 每当出现失配时，**i指针不回溯**，而是利用已经得到的“部分匹配”结果将模式向右“滑动”**尽可能远的一段距离**后，继续比较。
- 假设主串‘ $s_1s_2\cdots s_n$ ’，模式串‘ $p_1p_2\cdots p_m$ ’，当主串中第**i**个字符与模式串中第**j**个字符“失配”时，主串中第**i**个字符应与模式串中第**k**个字符再比较。

第四章 串——改进的模式匹配算法（KMP算法）



实质： $k-1$ 为' $p_1p_2\cdots p_{j-1}$ '的最大相同真前缀，即模式中头 $k-1$ 个字符的子串' $p_1p_2\cdots p_{k-1}$ '必定与主串中第 i 个字符之前长度为 $k-1$ 的子串' $s_{i-k+1}s_{i-k+2}\cdots s_{i-1}$ '相等，由此，匹配仅需从模式中第 k 个字符与主串中第 i 个字符比较起继续进行。

第1趟 S
T

$i=1=2=3$
a b b a b a
a b a
 $j=1=j=2=3$

第2趟 S
T

$i=3$
a b b a b a
a b a
 $j=1$

i指针不回溯

第3趟 S
T

$i=4=i=5=6=i=7$
a b b a b a
a b a ✓
 $j=1=j=2=3=j=4$

返回 $i=4$

第四章 串——改进的模式匹配算法（KMP算法）

- 定义模式串的**next函数**：若令 $next[j]=k$,则 $next[j]$ 表明当模式串中第 j 个字符与主串中第 i 个字符“失配”时，在模式串中需重新和主串中该字符进行比较的字符的位置。
- **next函数**有时也称为**失效函数**，其值仅取决于模式串本身而和想匹配的主串无关——即**与 i 值无关**！

$$next[j] = \left\{ \begin{array}{ll} 0 & \text{当 } j = 1 \text{ 时} \\ \text{Max}\{k \mid 1 < k < j \text{ 且 } 'p_1p_2 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\} & \\ 1 & \text{当此集合不空时} \\ & \text{其它情况} \end{array} \right.$$

第四章 串——改进的模式匹配算法（KMP算法）

【例1】

j	1	2	3	4	5
模式串	a	b	c	a	c
next[j]	0	1	1	1	2

【例2】

j	1	2	3	4	5	6	7	8
模式串	a	b	a	b	c	a	b	c
next[j]	0	1	1	2	3	1	2	3

求 $next$ 函数值的过程是一个递推过程，分析如下：

已知： $next[1] = 0$;

假设： $next[j] = k$;

1) 若： $p_j = p_k$

则： $next[j+1] = k+1$

2) 若： $p_j \neq p_k$ ，且 $next[k]=k'$ ，则：

a) 若 $p_j = p_{k'}$ ，则： $next[j+1] = next[k]+1$;

b) 若 $p_j \neq p_{k'}$ ，则：将模式串继续向右滑动直至将模式中的第 $next[k']$ 个字符和 p_j 对齐，.....，以此类推，直至 p_j 和模式串中某个字符匹配成功或者不存在任何 $k'(1 < k' < j)$ 满足

$'p_1 \dots p_k' = 'p_{j-k'+1} \dots p_j'$ ，则 $next[j+1]=1$ 。



这实际上也是一个模式匹配的过程，不同在于：主串和模式串是同一个

■ 例子：模式串的next函数值

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

已知前6个字符的next函数值，求next[7]？

- 由next[6]=3, 又 $p_6 \neq p_3$, 则需比较 p_6 和 p_1 (因为next[3]=1), 这相当于将子串模式向右滑动。
- 由于 $p_6 \neq p_1$, 而且next[1]=0, 所以next[7]=1;

- 因为 $p_7 = p_1$, 则next[8]=2。

next函数的改进:

例如:

$S = \text{'aaabaaabaaabaaab'}$

$T = \text{'aaaab'}$

- 当 $i=4$ 、 $j=4$ 时, $S[4] \neq T[4]$, 由 $\text{next}[j]$ 的指示还需进行 $S[4]$ 和 $T[3]$ 、 $S[4]$ 和 $T[2]$ 、 $S[4]$ 和 $T[1]$ 三次的比较。
- 但是由 $T[1]=T[2]=T[3]=T[4]$, 可知不需要再和主串的 $S[4]$ 比较, 可以将模式一气向右滑动4个字符直接进行 $S[5]$ 和 $T[1]$ 字符的比较。

j	1	2	3	4	5
模式串	a	a	a	a	b
next[j]	0	1	2	3	4

若 $\text{next}[j]=k$, 而模式中 $p_k=p_j$, 则当主串中字符 s_i 和 p_j 比较不等时, 不需要再和 p_k 进行比较, 而直接和 $p_{\text{next}[k]}$ 进行比较, 即应使 $\text{next}[j]$ 和 $\text{next}[k]$ 相同。

next函数的改进算法:

```
void get_nextval(SString &T, int &nextval[])
```

```
{ // 求模式串T的next函数修正值并存入数组nextval.
```

```
    i = 1; nextval[1] = 0; j = 0;
```

```
    while (i < T[0]) {
```

```
        if (j = 0 || T[i] == T[j]) {
```

```
            ++i; ++j;
```

```
            if (T[i] != T[j]) next[i] = j;
```

```
            else nextval[i] = nextval[j];
```

```
        }
```

```
        else j = nextval[j];
```

```
    }
```

```
} // get_nextval
```

第五章 数组和广义表

- 学过程序语言的朋友，数组的概念已经不是第一次见到了，应该已经“一回生，二回熟”了，所以，在概念上，不会存在太大障碍。但作为**DS**课程来说，本章的考查重点可能与程序语言所关注的不太一样，下面会作介绍。
- 广义表的概念，是数据结构里第一次出现的。它是线性表或表元素的有限序列，构成该结构的每个子表或元素也是线性结构的。
- 本章的考查重点有：
 1. **多维数组中某数组元素的position求解**。一般是给出数组元素的首元素地址和每个元素占用的地址空间并给出多维数组的维数，然后要求你求出该数组中的某个元素所在的位置。
 2. **明确按行存储和按列存储的区别和联系**，并能够按照这两种不同的存储方式求解1中类型的题。

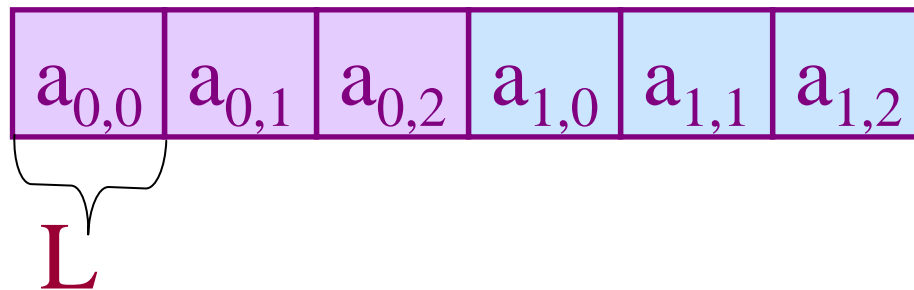
3.将特殊矩阵中的元素按相应的换算方式存入数组中。这些矩阵包括：对称矩阵，三角矩阵，具有某种特点的稀疏矩阵等。熟悉稀疏矩阵的三种不同存储方式：三元组，带辅助行向量的二元组，十字链表存储。掌握将稀疏矩阵的三元组或二元组向十字链表进行转换的算法。

4.广义表的概念，特别应该明确表头与表尾的定义。这一点，是理解整个广义表一节算法的基础。传统经常给出某个广义表，要求给出对应的广义表存储结构，注意两种存储结构不要混淆。近来，在一些学校中，出现了这样一种题目类型：给出对某个广义表L若干个求了若干次的取头和取尾操作后的串值，要求求出原广义表L。大家要留意。

5.与广义表有关的递归算法。由于广义表的定义就是递归的，所以，与广义表有关的算法也常是递归形式的。比如：求表深度，复制广义表等。这种题目，可以根据不同角度广义表的表现形式运用两种不同的方式解答：一是把一个广义表看作是表头和表尾两部分，分别对表头和表尾进行操作；二是把一个广义表看作是若干个子表，分别对每个子表进行操作。

例如： 二维数组 $A[0..b_1, 0..b_2]$ 按“行优先顺序”存储在内存中，假设每个元素占用 L 个存储单元。

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$



元素 $a_{i,j}$ 的存储地址应是数组的基地址加上排在 $a_{i,j}$ 前面的元素所占用的单元数，即：

$$\text{LOC}(i,j) = \text{LOC}(0,0) + (b_2 \times i + j) \times L$$

↑ 称为基地址或基址。

推广到一般情况，可得到 n 维数组数据元素存储位置的映象关系

$$\begin{aligned}\text{LOC}(j_1, j_2, \dots, j_n) \\ &= \text{LOC}(0, 0, \dots, 0) + (b_2 * b_3 * \dots * b_n * j_1 + b_3 * \dots * b_n * j_2 + \dots + b_n * j_{n-1} + j_n) * L \\ &= \text{LOC}(0, 0, \dots, 0) + \sum_{i=1}^n c_i j_i\end{aligned}$$

其中 $c_n = L$, $c_{i-1} = b_i \times c_i$, $1 < i \leq n$ 。

称为 n 维数组的映象函数。数组元素的存储位置是其下标的线性函数。

第五章 数组和广义表——特殊矩阵

■ 特殊矩阵

- 对称矩阵
- 三角矩阵
- 对角矩阵

- 上述的各种特殊矩阵，其非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个向量中，并且一般都能找到矩阵中的元素与该向量的对应关系，通过这个关系，仍能对矩阵的元素进行随机存取。

第五章 数组和广义表——对称矩阵

- 不失一般性，我们按“行优先顺序”存储主对角线（包括对角线）以下的元素，其存储形式如图所示：

$$\begin{pmatrix} \mathbf{1} & \mathbf{5} & \mathbf{1} & \mathbf{3} & \mathbf{7} \\ \mathbf{5} & \mathbf{0} & \mathbf{8} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{8} & \mathbf{9} & \mathbf{2} & \mathbf{6} \\ \mathbf{3} & \mathbf{0} & \mathbf{2} & \mathbf{5} & \mathbf{1} \\ \mathbf{7} & \mathbf{0} & \mathbf{6} & \mathbf{1} & \mathbf{3} \end{pmatrix}$$

- 在这个下三角矩阵中，第 i 行恰有 i 个元素，元素总数为：

$$1+2+\cdots+n=n(n+1)/2$$

- 因此，我们可以按图中箭头所指的次序将这些元素存放在一个向量 $sa[0..n(n+1)/2-1]$ 中。

第五章 数组和广义表——对称矩阵

- 为了便于访问对称矩阵**A**中的元素，我们必须在 a_{ij} 和 $sa[k]$ 之间找一个对应关系：

- 若 $i \geq j$ ，则 a_{ij} 在下三角形中。 a_{ij} 之前的 $i-1$ 行（从第1行到第 $i-1$ 行）一共有 $1+2+\dots+i-1=i(i-1)/2$ 个元素，在第 i 行上， a_{ij} 之前恰有 $j-1$ 个元素（即 $a_{i1}, a_{i2}, \dots, a_{ij-1}$ ），因此有：

$$k=i*(i-1)/2+j-1 \quad 0 \leq k < n(n+1)/2$$

- 若 $i < j$ ，则 a_{ij} 是在上三角矩阵中。因为 $a_{ij}=a_{ji}$ ，所以只要交换上述对应关系式中的 i 和 j 即可得到：

$$k=j*(j-1)/2+i-1 \quad 0 \leq k < n(n+1)/2$$

- 令 $i=\max(i,j)$ ， $j=\min(i,j)$ ，则 k 和 i, j 的对应关系可统一为：

$$k=i*(i-1)/2+j-1 \quad 0 \leq k < n(n+1)/2$$

第五章 数组和广义表——对称矩阵

- 因此， a_{ij} 的地址可用下列式计算：

$$\text{LOC}(a_{ij}) = \text{LOC}(\text{sa}[k])$$

$$= \text{LOC}(\text{sa}[0]) + k * L = \text{LOC}(\text{sa}[0]) + [i * (i-1)/2 + j-1] * L$$

- 有了上述的下标交换关系，

- 对于任意给定一组下标(i, j)，均可在 $\text{sa}[k]$ 中找到矩阵元素 a_{ij} ；
- 反之，对所有的 $k=0, 1, 2, \dots, n(n-1)/2-1$ ，都能确定 $\text{sa}[k]$ 中的元素在矩阵中的位置(i, j)。

- 由此，称 $\text{sa}[n(n+1)/2]$ 为阶对称矩阵 A 的压缩存储，见下图：

a_{11}	a_{21}	a_{22}	a_{31}	a_{n1}	...	a_{nn}
$k=0$	1	2	3		$n(n-1)/2$...	$n(n+1)/2-1$

第五章 数组和广义表——稀疏矩阵

- 为了节省存储单元，很自然地想到使用压缩存储方法。但由于非零元素的分布一般是没有规律的，因此在存储非零元素的同时，还必须同时记下它所在的行和列的位置 (i,j) 。
- 一个三元组 (i,j,a_{ij}) 唯一确定了矩阵 A 的一个非零元。因此，稀疏矩阵可由表示非零元的三元组及其行列数唯一确定。

例如，下列三元组表

$((1,2,12)(1,3,9),(3,1,-3),(3,6,14),(4,3,24),$
 $(5,2,18),(6,1,15),(6,4,-7))$

加上(6,7)这一对行、列值便可作为下列矩阵M的另一种描述。而由上述三元组表的不同表示方法可引出稀疏矩阵不同的压缩存储方法。

$$M = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

$$T = \begin{pmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

第五章 数组和广义表——稀疏矩阵

■ 稀疏矩阵的三种压缩存储方法：

1. 三元组顺序表
2. 行逻辑链接的顺序表
3. 十字链表

第五章 数组和广义表——三元组顺序表

- 假设以**顺序存储结构**来表示三元组表，则可得到稀疏矩阵的一种压缩存储方法——三元顺序表。

```
#define MAXSIZE 12500           //非零元个数的最大值

typedef struct{
    int  i,j; //非零元的行下标和列下标
    ElemType e;
}Triple;

typedef struct{
    Triple data[MAXSIZE+1]; //非零元三元组表，data[0]未用
    int mu,nu,tu; //矩阵的行数、列数和非零元个数
}TSMatrix;
```

图5.5中所示的稀疏矩阵M及其对应转置矩阵T的三元组的表示分别为：

i	j	v
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

i	j	v
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14

第五章 数组和广义表——三元组顺序表

- 矩阵的**快速转置**的算法。
- **具体实施如下**：一遍扫描先确定三元组的位置关系，二次扫描由位置关系装入三元组。可见，位置关系是此种算法的关键。
- 为了预先确定矩阵M中的每一列的第一个非零元素在数组B中应有的位置，**需要先求得矩阵M中的每一列中非零元素的个数**，因为：
 - 矩阵M中每一列的第一个非零元素在数组B中应有的位置等于前一列第一个非零元素的位置加上前列非零元素的个数。

第五章 数组和广义表——三元组顺序表

- 为此，需要设置两个一维数组 **num[1..n]** 和 **cpot[1..n]**
 - num[1..n]**: 统计M中每列非零元素的个数，例如num[1]记录第1列非零元素的个数。
 - cpot[1..n]**: 由递推关系得出M中的每列第一个非零元素在B中的位置。
- 算法通过cpot数组建立位置对应关系:
$$\begin{aligned} \text{cpot}[1] &= 1 \\ \text{cpot}[\text{col}] &= \text{cpot}[\text{col}-1] + \text{num}[\text{col}-1] \\ 2 \leq \text{col} &\leq a.n \end{aligned}$$

首先应该确定每一列的第一个非零元在三元组中的位置。

1	2	15
1	5	-5
2	2	-7
3	1	36
3	4	28

col	1	2	3	4	5
Num[pos]	1	2	0	1	1
Cpot[col]	1	2	4	4	5

$\text{cpot}[1] = 1;$

for (col=2; col<=M.nu; ++col)

$\text{cpot}[\text{col}] = \text{cpot}[\text{col}-1] + \text{num}[\text{col}-1];$

A	i	j	v
	1	2	v

p ↙

B	i	j	v
	2	1	v

↘ q

第一列元素个数

第二列元素个数

第三列元素个数

num

--	--	--	--

col=2

cpot

			...
--	--	--	-----

$q = \text{cpot}[\text{col}]$

A的第二列第一个非零元素在B中的位置

Status FastTransposeSMatrix(TSMatrix M, TSMatrix &T)

```
{ //采用三元组顺序表存储表示, 求稀疏矩阵M的转置矩阵T。
    T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;
    if (T.tu){
        //求M中每一列含非零元个数
        for (col=1;col<=M.nu;++col) num[col]=0;
        for (t=1;t<=M.tu;++t) ++num[M.data[t].j];
        cpot[1]=1;
        //求第col列中的一个非零元在b.data中的序号
        for (col=2;col<=M.nu;++col) cpot[col]=cpot[col-1]+num[col-1];
        for (p=1;p<=M.tu; ++p){ //转置矩阵元素
            col=M.data[p].j; q=cpot[col];
            T.data[q].i=M.data[p].j; T.data[q].j=M.data[p].i;
            T.data[q].e=M.data[p].e; ++cpot[col];
        }
    }return OK;
}
```

第五章 数组和广义表——行逻辑链接的顺序表

- 有时为了方便某些矩阵运算，我们在按行优先存储的三元组中，加入一个行表来记录稀疏矩阵中每行的非零元素在三元组表中的起始位置。
- 当将行表作为三元组表的一个新增属性加以描述时，我们就得到了稀疏矩阵的另一种顺序存储结构：**行逻辑链接的三元组表**。其类型描述如下：

```
typedef struct{
```

```
    Triple data[MAXSIZE+1]; //非零元三元组表
```

```
    int    rpos[MAXRC+1]; //各行第一个非零元的位置表
```

```
    int    mu,nu,tu; //矩阵的行数、列数和非零元个数
```

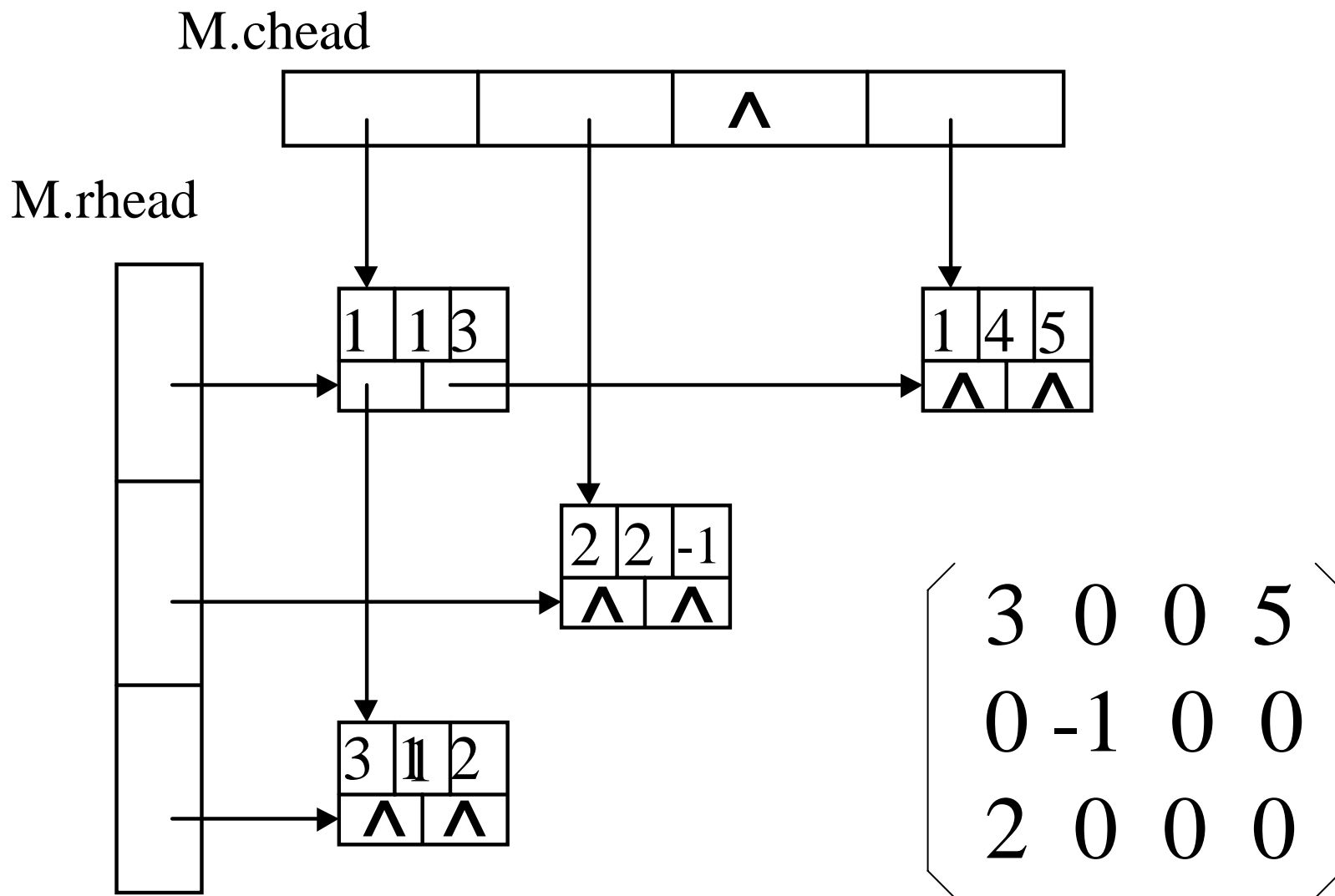
```
}RLSMatrix;
```


第五章 数组和广义表——十字链表

- 当矩阵的非零元个数和位置在操作过程中变化较大时，就不宜采用顺序存储结构来表示三元组的线性表，而应采用链式存储结构的十字链表。

```
typedef struct OLNode{  
    int i,j;           //该非零元的行和列下标  
    ElemType e;        //该非零元的值  
    struct OLNode *right, *down; //行和列的后继  
}OLNode; *OLink;  
typedef struct{  
    OLink *rhead, *chead; //行和列的指针向量基址  
    int mu, nu, tu; //矩阵的行数、列数和非零元个数  
}CrossList;
```

第五章 数组和广义表——十字链表



第六章 树与二叉树

- 从对线性结构的研究过度到对**树形结构**的研究，是数据结构课程学习的一次跃变，此次跃变完成的好坏，将直接关系到你的实际的考试中是否可以拿到高分。所以，树这一章的重要性，已经不说自明了。
- 总体来说，**树这一章的知识点包括**：二叉树的概念、性质和存储结构，二叉树遍历的三种算法（递归与非递归），在三种基本遍历算法的基础上实现二叉树的其它算法，线索二叉树的概念和线索化算法以及线索化后的查找算法，最优二叉树的概念、构成和应用，树的概念和存储形式，树与森林的遍历算法及其与二叉树遍历算法的联系，树与森林和二叉树的转换。

第六章 树与二叉树

■ 下面我们来看考试中对以上知识的主要考查方法：

1. 二叉树的概念、性质和存储结构

考查方法可有：

- 直接考查二叉树的定义，让你说明二叉树与普通双分支树的区别；考查满二叉树和完全二叉树的性质，普通二叉树的**五个性质**：第 i 层的最多结点数，深度为 k 的二叉树的最多结点数， $n_0 = n_2 + 1$ 的性质， n 个结点的完全二叉树的深度，顺序存储二叉树时孩子结点与父结点之间的换算关系（左为： $2*i$ ，右为： $2*i+1$ ）。
- 二叉树的顺序存储和二叉链表存储的各自优缺点及适用场合，二叉树的三叉链表表示方法。

第六章 树与二叉树

2. 二叉树的三种遍历算法

- 这一知识点掌握的好坏，将直接关系到树一章的算法能否理解，进而关系到树一章的算法设计题能否顺利完成。二叉树的遍历算法有三种：**先序，中序和后序**。其划分的依据是视其每个算法中**对根结点数据的访问顺序而定**。要熟练掌握三种遍历的递归算法，理解其执行的实际步骤，并且应该熟练掌握三种遍历的非递归算法。

3. 可在三种遍历算法的基础上改造完成的其它二叉树算法：

- 求叶子个数，求二叉树结点总数，求度为1或度为2的结点总数，复制二叉树，建立二叉树，交换左右子树，查找值为n的某个指定结点，删除值为n的某个指定结点，诸如此类等等等等。如果你可以熟练掌握二叉树的递归和非递归遍历算法，那么解决以上问题就是小菜一碟了。

第六章 树与二叉树

4. 线索二叉树:

- 线索二叉树的引出, 是为避免如二叉树遍历时的递归求解。众所周知, 递归虽然形式上比较好理解, 但是消耗了大量的内存资源, 如果递归层次一多, 势必带来资源耗尽的危险, 为了避免此类情况, 线索二叉树便出现了。对于线索二叉树, 应该掌握: 线索化的实质, 三种线索化的算法, 线索化后二叉树的遍历算法, 基本线索二叉树的其它算法问题 (如: 查找某一类线索二叉树中指定结点的前驱或后继结点就是一类常考题)。

5. 最优二叉树 (哈夫曼树):

- 最优二叉树是为了解决特定问题引出的特殊二叉树结构, 它的前提是给二叉树的每条边赋予了权值, 这样形成的二叉树按权相加之和是最小的。最优二叉树一节, 直接考查算法源码的很少, 一般是给你一组数据, 要求你建立基于这组数据的最优二叉树, 并求出其最小权值之和, 此类题目不难, 属送分题。另外, 还有考叶子结点与结点总数的关系。

第六章 树与二叉树

6.树与森林:

- 二叉树是一种特殊的树，这种特殊不仅仅在于其分支最多为2以及其它特征，一个最重要的特殊之处在于：**二叉树是有序的！**即：二叉树的左右孩子是不可交换的，如果交换了就成了另外一棵二叉树，这样交换之后的二叉树与原二叉树我们认为是不相同的两棵二叉树。但是，对于普通的双分支树而言，不具有这种性质。
- 树与森林的遍历，不像二叉树那样丰富，他们只有两种遍历算法：**先根与后根**（对于森林而言称作：先序与后序遍历）。能熟练掌握根据先根或后根写出他们的遍历序列。此二者的先根与后根遍历与二叉树中的遍历算法是有对应关系的：**先根遍历对应二叉树的先序遍历，而后根遍历对应二叉树的中序遍历**。二叉树、树与森林之所以能有以上的对应关系，全拜二叉链表所赐。二叉树使用二叉链表分别存放他的左右孩子，树利用二叉链表存储孩子及兄弟（称孩子兄弟链表），而森林也是利用二叉链表存储孩子及兄弟。
- 树一章，处处是重点，道道是考题，大家务必个个过关。

先序遍历的递归算法，中序与后序与此类似。

```
Status PreOrderTraverse(BiTree T, Status  
    (*Visit) (TElemType e)) {
```

```
//采用二叉链表结构，
```

```
//Visit函数可以是如下打印函数：
```

```
//Status PrintElement(TElemType e)
```

```
//{    printf(e);    return OK; }
```

```
    if (T) {
```

```
        if (Visit(T→data))
```

```
            if (PreOrderTraverse(T→lchild, Visit))
```

```
                if (PreOrderTraverse(T→rchild, Visit))
```

```
                    return OK;
```

```
            return ERROR;
```

```
        } else return OK;
```

```
    }
```

只要调整一下
Visit() 函数的位
置，就可以简单地
转化为中序遍历算
法和后序遍历算法。

第六章 树与二叉树——遍历二叉树

- 观察：中序遍历算法递归工作栈的状态变化。
 - (1) 当栈顶记录中的指针非空时，应遍历左子树，即指向左子树根的指针进栈。
 - (2) 若栈顶记录中的指针值为空，则应退至上一层。
 - 若是从左子树返回，则应访问当前层即栈顶指针上指针所指的根结点；
 - 若是从右子树返回，则表明当前层的遍历结束，应继续退栈。
- 由此可得以下两个中序遍历二叉树的非递归算法，先序遍历二叉树的非递归算法与此类似，但后序遍历二叉树的非递归算法要相对复杂一点。

中序遍历二叉树的非递归算法1:

```
Status InOrderTraverse(BiTree T, Status (*Visit)
    (TElemType e)) {
    InitStack(S); Push(S, T); //根指针进栈
    while (!StackEmpty(S)) {
        while (GetTop(S, p) && p) Push(S, p->lchild);
            //向左走到尽头

        Pop(S, p); //空指针退栈
        if (!StackEmpty(S)) { //访问结点, 向右一步
            Pop(S, p);
            if (!Visit(p->data)) return ERROR;
            Push(S, p->rchild);
        } //if
    } //while
    return OK;
}
```

中序遍历二叉树的非递归算法2:

```
Status InOrderTraverse (BiTree T, Status (*Visit)
    (TElemType e)) {
    InitStack (S); p=T;
    while (p||!StackEmpty (S)) {
        if (p) { //根指针进栈, 遍历左子树
            Push (S, p);
            p=p→lchild;
        }
        else { //根指针退栈, 访问根结点, 遍历右子树
            Pop (S, p);
            if (!Visit (p→data)) return ERROR;
            p=p→rchild;
        }
    }
    return OK;
}
```

遍历右子树时不再需要保存当前层的根结点。

第六章 树与二叉树——遍历二叉树

- 二叉树的先序序列和中序序列能够惟一确定一棵二叉树。
- 二叉树的后序序列和中序序列能够惟一确定一棵二叉树。
- 二叉树的先序序列和后序序列不能惟一确定一棵二叉树。

第六章 树与二叉树——线索二叉树

- 当以二叉链表作为存储结构时, 只能找到结点的左右孩子的信息, 而不能得到结点的任一序列的**前驱**与**后继**信息, 这种信息只有在**遍历**的动态过程中才能得到。
- 为了能保存所需的信息, 在**二叉链表的结点**中增加两个**标志域**, 并作如下规定:
 - 若**该结点的左子树不空**, 则**Lchild**域的指针指向其左子树, 且左标志域的值为“**指针 Link**”; 否则, **Lchild**域的指针指向其“前驱”, 且左标志的值为“**线索 Thread**”。
 - **该结点的右子树不空**, 则**rchild**域的指针指向其右子树, 且右标志域的值为“**指针 Link**”; 否则, **rchild**域的指针指向其“后继”, 且右标志的值为“**线索 Thread**”。

第六章 树与二叉树——线索二叉树

那么，又如何进行二叉树的线索化呢？

- 线索化的实质是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继的信息只有在遍历时才能得到，因此线索化的过程即为在遍历的过程中修改空指针的过程。
- 为了记下遍历过程中访问结点的先后关系，附设一个指针`pre`始终指向刚刚访问过的结点，若指针`p`指向当前访问的结点，则`pre`指向它的前驱。
- 由此可得中序线索化的算法6.6和算法6.7。

第六章 树与二叉树——树和森林的遍历

- 森林遍历与二叉树遍历的对应关系：
森林的先序和中序遍历分别对应二叉树的先序和中序遍历。

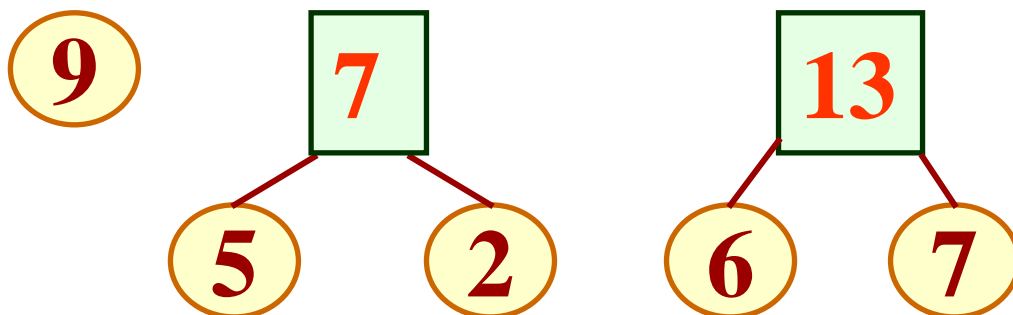
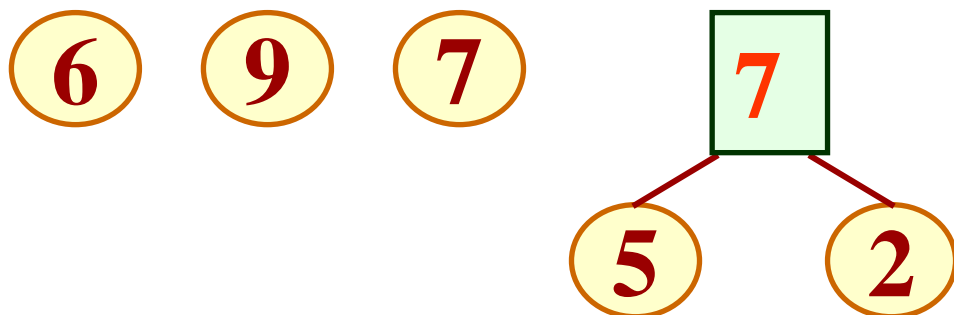
树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

- 由此可见，相当部分森林与树的操作，可以归结为二叉树的操作。

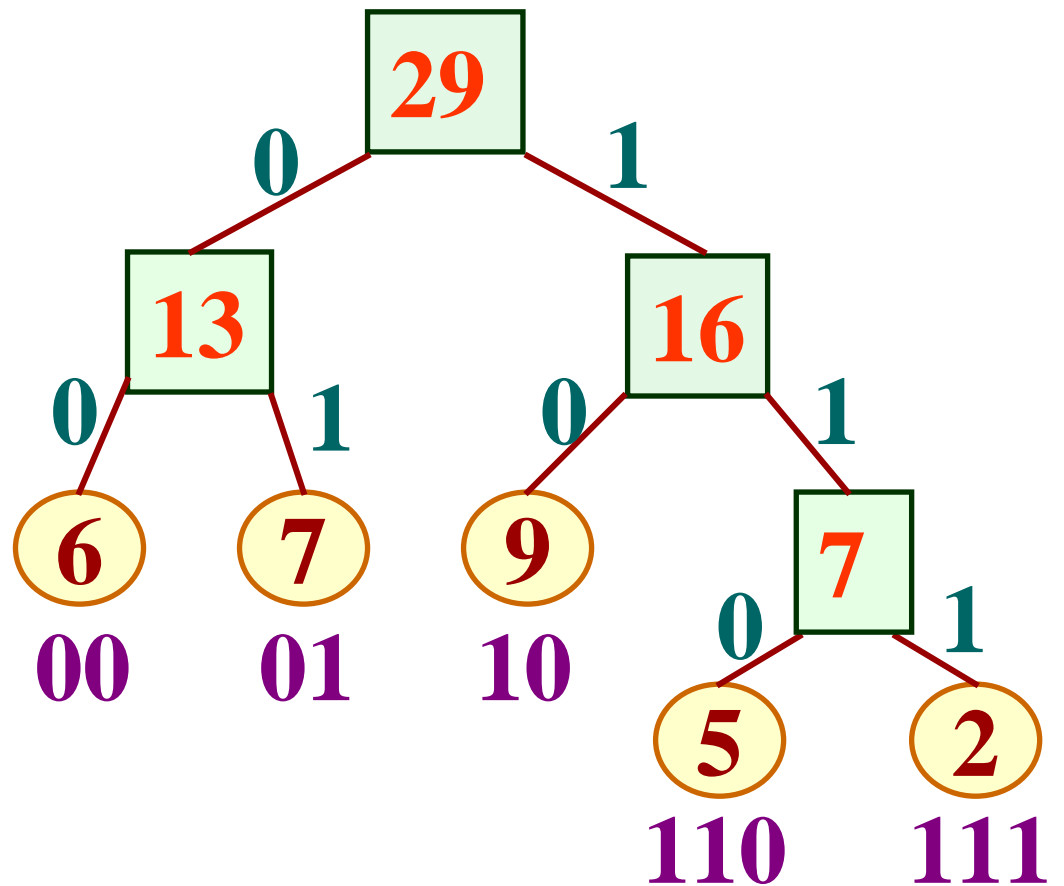
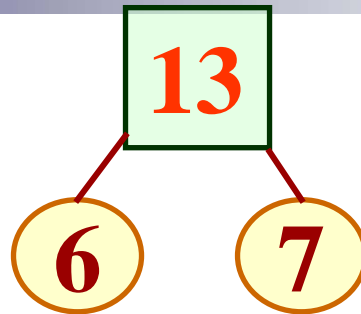
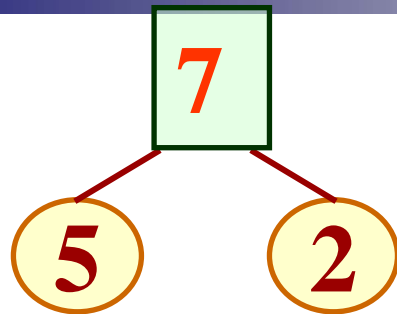
第六章 树与二叉树——最优二叉树

- 赫夫曼树(Huffman)，又称最优树，是一类带权路径长度最短的树，有着广泛的应用。
- 构造赫夫曼树的算法（以二叉树为例）：
 - (1) 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 构成 n 棵二叉树的集合 $F=\{T_1, T_2, \dots, T_n\}$ ，其中每棵二叉树 T_i 中只有一个带权为 w_i 的根结点，其左右子树为空。
 - (2) 在 F 中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的权值为其左、右子树上根结点的权值之和。
 - (3) 在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。
 - (4) 重复(2)和(3)，直到 F 只含一棵树为止。这棵树便为赫夫曼树。
- 图6.24展示了图6.22(c)的赫夫曼树的构造过程。

例如：已知权值 $W=\{ 5, 6, 2, 9, 7 \}$



9



第七章 图

- 如果说，从线性结构向树形结构研究的转变，是数据结构学科对数据组织形式研究的一次升华，那么从树形结构的研究转到图形结构的研究，则进一步让我们看到了数据结构对于解决实际问题的重大推动作用。
- 图这一章的特点是：概念繁多，与离散数学中图的概念联系紧密，算法复杂，极易被考到，且容易出大题。
- 图这一章的主要考点以及这些考点的考查方式：

1.考查有关图的基本概念问题：

- 这些概念是进行图一章学习的基础，这一章的概念包括：图的定义和特点，无向图，有向图，入度，出度，完全图，生成子图，路径长度，回路，（强）连通图，（强）连通分量等概念。与这些概念相联系的相关计算题也应该掌握。

2.考查图的几种存储形式:

- 图的存储形式包括：邻接矩阵，（逆）邻接表，十字链表及邻接多重表。在考查时，题目可能会给出一种存储形式，要求考生用算法或手写出与给定的结构相对应的该图的另一种存储形式。

3.考查图的两种遍历算法：深度遍历和广度遍历

- 深度遍历和广度遍历是图的两种基本的遍历算法，这两个算法对图一章的重要性等同于“先序、中序、后序遍历”对于二叉树一章的重要性。在考查时，图一章的算法设计题常常是基于这两种基本的遍历算法而设计的，比如：“求最长的最短路径问题”和“判断两顶点间是否存在长为K的简单路径问题”，就分别用到了广度遍历和深度遍历算法。

4. 生成树、最小生成树的概念以及最小生成树的构造：Prim算法和Kruskal算法。

- 考查时，一般不要求写出算法源码，而是要求根据这两种最小生成树的算法思想写出其构造过程及最终生成的最小生成树。另外，也经常考MST性质、Prim算法和Kruskal算法可靠性的证明。

5. 拓扑排序问题：

- 拓扑排序有两种方法，一是无前趋的顶点优先算法，二是无后继的顶点优先算法。换句话说，一种是“从前向后”的排序，一种是“从后向前”排。当然，后一种排序出来的结果是“逆拓扑有序”的。
- 注意，也可利用拓扑排序算法来确定有向图是否有环。

6.关键路径问题:

- 这个问题是图一章的难点问题。理解关键路径的关键有三个
方面：**一是何谓关键路径，二是最早时间是什么意思、如何求，三是最晚时间是什么意思、如何求。**简单地说，最早时间是通过“从前向后”的方法求的，而最晚时间是通过“从后向前”的方法求解的，并且，要想求最晚时间必须是在所有的最早时间都已经求出来之后才能进行。这个问题拿来直接考算法源码的不多，一般是要要求按照书上的算法描述求解的过程和步骤。
- 在实际设计关键路径的算法时，还应该注意以下这一点：采用邻接表的存储结构，求最早时间和最晚时间要采用不同的处理方法，即：在算法初始时，应该首先将所有顶点的最早时间全部置为0。关键路径问题是工程进度控制的重要方法，具有很强的实用性。

7.最短路径问题:

- 与关键路径问题并称为图一章的两只拦路虎。概念理解是比較容易的，关键是算法的理解。最短路径问题分为两种：一是求从某一点出发到其余各点的最短路径；二是求图中每一对顶点之间的最短路径。这个问题也具有非常实用的背景特色，一个典型的应该就是旅游景点及旅游路线的选择问题。解决第一个问题用Dijkstra算法，解决第二个问题用Floyd算法。注意区分。

第七章 图——存储结构

图的四种常用的存储形式：

- 数组：邻接矩阵和加权邻接矩阵（**labeled adjacency matrix**）

- 邻接表

- 十字链表

- 邻接多重表

用于有向图，查询
进入结点和离开结
点的边容易

用于无向图，边表中的
边结点只需存放一
次，节约内存。

第七章 图——邻接矩阵

- 图的邻接矩阵存储表示:

```
typedef enum{DG, DN, UDG, UDN}GraphKind;  
    // {有向图, 有向网, 无向图, 无向网}
```

```
typedef struct ArcCell{ // 弧的定义
```

```
    VRType adj; //对于无权图, 用1或0表示相邻否  
                //对于有权图, 则为权值
```

```
    InfoType *info; //该弧相关信息的指针
```

```
}ArcCell, AdjMatrix[MAX_VERTEX_NUM] [MAX_VERTEX_NUM]
```

第七章 图——邻接矩阵

■ 图的邻接矩阵存储表示

```
typedef struct{ //图的定义
```

```
    VertexType vexs[MAX_VERTEX_NUM]; //顶点向量
```

```
    AdjMatrix arcs; //邻接矩阵
```

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

```
    GraphKind kind; // 图的种类标志
```

```
} MGraph;
```

- 算法7.2给出了无向网的邻接矩阵创建算法。构造一个具有 n 个顶点和 e 条边的无向网 G 的时间复杂度是 $O(n^2+e*n)$ ，其中对邻接矩阵 $G.arcs$ 的初始化耗费了 $O(n^2)$ 的时间。

第七章 图——邻接表

结点表中的结点的表示:

- 用数组

data	firstarc
------	----------

data: 结点的数据域, 保存结点的数据值。

firstarc: 结点的指针域, 给出自该结点出发的第一条边的边结点的地址。

- 用单链表

data	firstarc	nextvex
------	----------	---------

data: 结点的数据域, 保存结点的数据值。

firstarc: 结点的指针域, 给出自该结点出发的第一条边的边结点的地址。

nextvex: 结点的指针域, 给出该结点的下一结点的地址。

第七章 图——邻接表

边结点表中的结点的表示：

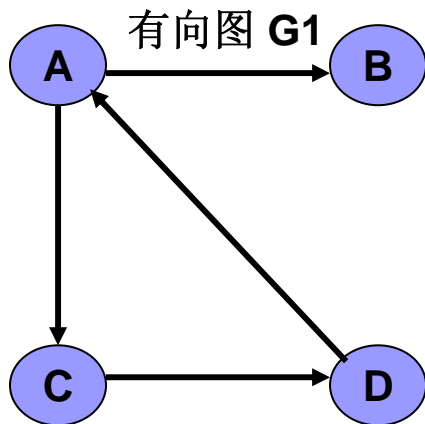
info	adjvex	nextarc
------	--------	---------

info: 边结点的数据域，保存边的权值等。

adjvex: 边结点的指针域，给出本条边依附的另一结点（非出发结点）的地址。

nextarc: 结点的指针域，给出自该结点出发的下一条边的边结点的地址。

第七章 图——邻接表



data firstarc

	data	firstarc
0	A	—
1	B	null
2	C	—
3	D	—

adjvex nextvex

	adjvex	nextvex
1	1	—
2	2	null
3	3	null
0	0	null

寻找进入... 常困难!!!

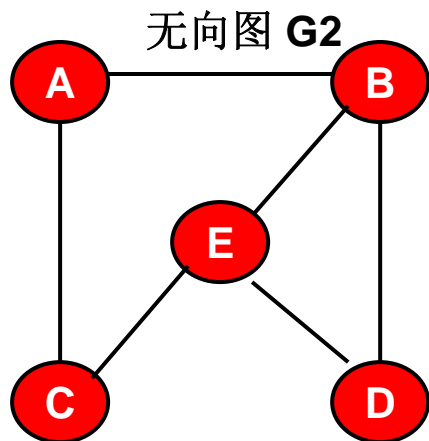


寻找进入结点的边非常困难!!!

改进：建立逆邻接表或十字链表



adjvex 指针域之值为相应结点的数组元素的下标!!!



data		firstarc			adjvex	nextvex					
0	A	—	→	1	—	→	2	null			
1	B	—	→	0	—	→	3	—	→	4	null
2	C	—	→	0	—	→	4	null			
3	D	—	→	1	—	→	4	null			
4	E	—	→	1	—	→	2	—	→	3	null

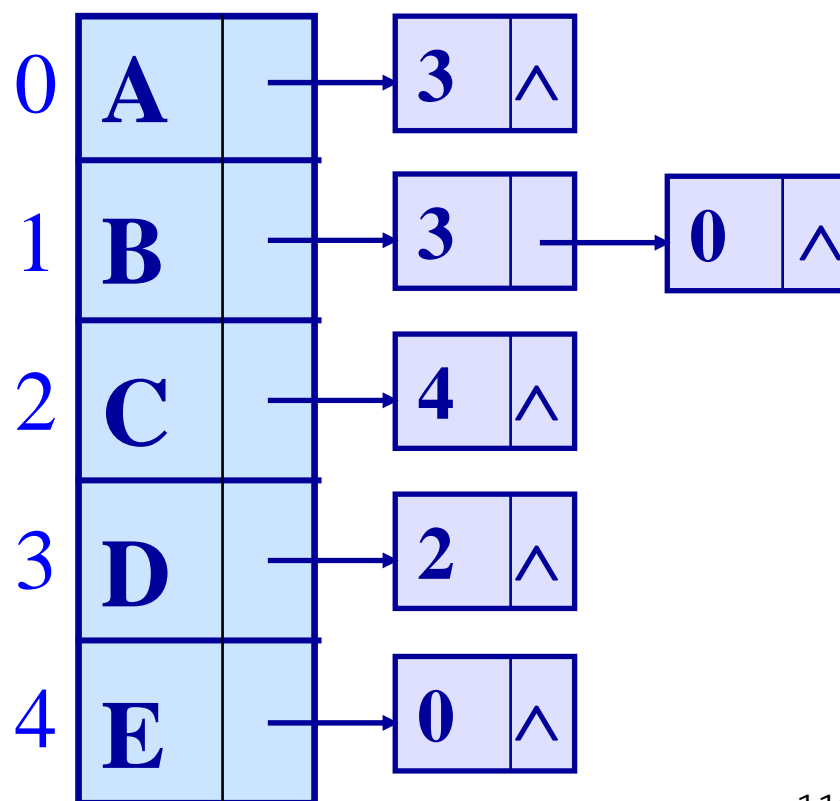
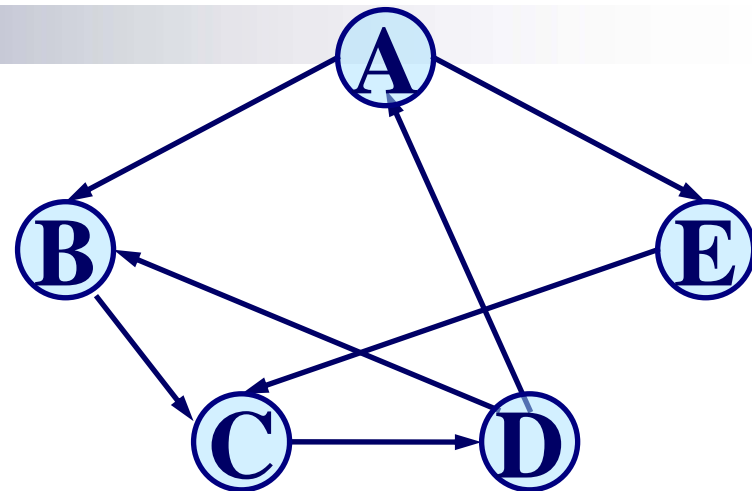


六条边却用了 12 个边结点!!!

改进：建立邻接多重表

有向图的逆邻接表:

在有向图的邻接表中，对每个顶点，链接的是指向该顶点的弧。



第七章 图——十字链表

- 结点表中的结点的表示:

data	firstin	firstout
------	---------	----------

data: 结点的数据域, 保存结点的数据值。

firstin: 结点的指针域, 给出进入该结点的第一条边的边结点的地址。

firstout: 结点的指针域, 给出自该结点出发的第一条边的边结点的地址。

- 边结点表中的结点的表示:

info	tailvex	headvex	hlink	tlink
------	---------	---------	-------	-------

info: 边结点的数据域, 保存边的权值等。

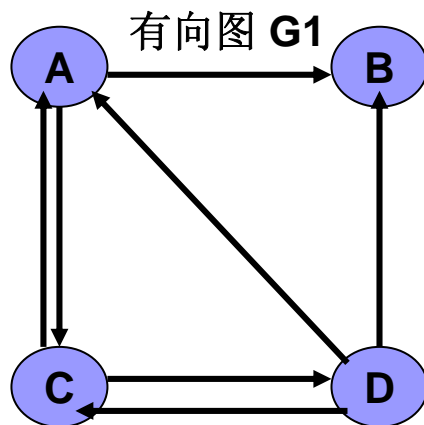
tailvex: 本条边的出发结点的地址。

headvex: 本条边的终止结点的地址。

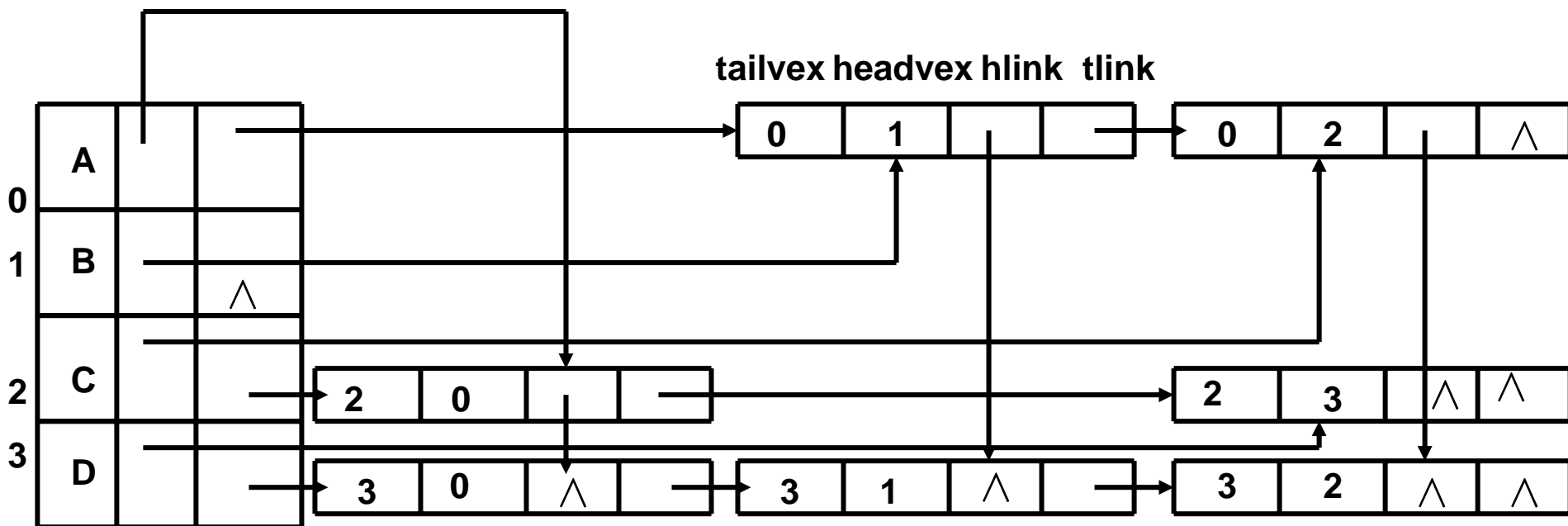
hlink: 出发结点相同的边中的下一条边的地址。

tlink: 终止结点相同的边中的下一条边的地址。

第七章 图——十字链表



- 用途：用于有向图，查询进入结点和离开结点的边容易。



第七章 图——邻接多重表

- 结点表中的结点的表示:

data	firstedge
------	-----------

data: 结点的数据域, 保存结点的数据值。

firstedge: 结点的指针域, 给出自该结点出发的第一条边的边结点的地址。

- 边结点表中的结点的表示:

mark	ivex	ilink	jvex	jlink	info
------	------	-------	------	-------	------

info: 边结点的数据域, 保存边的权值等。

mark: 边结点的标志域, 用于标识该条边是否被访问过。

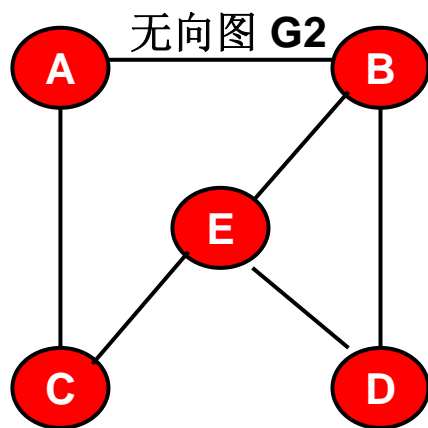
ivex: 本条边依附的一个结点的地址。

ilink: 依附于结点**ivex**的下一条边的地址。

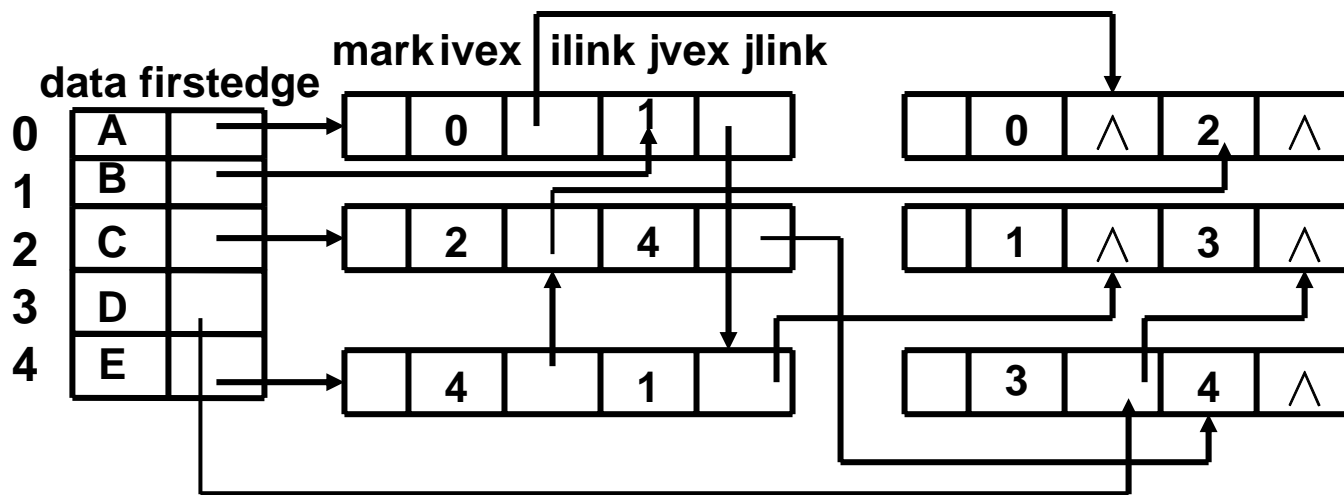
jvex: 本条边依附的另一个结点的地址。

jlink: 依附于结点**jvex**下一条边的地址

第七章 图——邻接多重表



- 用途：用于无向图，边表中的边结点只需存放一次，节约内存。

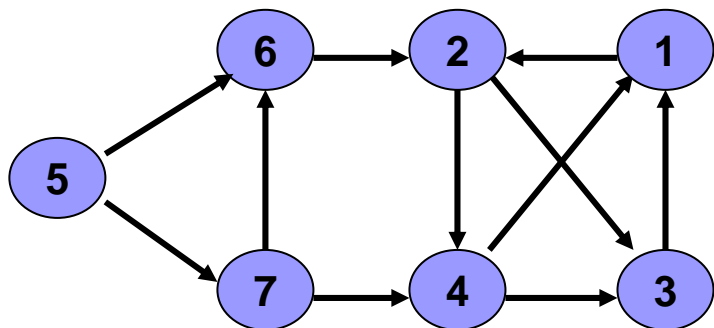


第七章 图——深度优先搜索遍历图

- 深度优先搜索类似于树的**先根遍历**，访问方式：
 - 1、选中第一个被访问的结点。
 - 2、对结点作已访问过的标志。
 - 3、依次从结点的未被访问过的第一个、第二个、第三个……邻接结点出发，进行深度优先搜索。转向2。
 - 4、如果还有顶点未被访问，则选中一个起始结点，转向2。
 - 5、所有的结点都被访问到，则结束。

第七章 图——深度优先搜索遍历图

例子：为了说明问题，邻接结点的访问次序以序号为准。序号小的先访问。
如：结点 **5** 的邻接结点有两个 **6**、**7**，则先访问结点 **6**，再访问结点 **7**。



从结点**1**出发的搜索序列：

1、2、3、4没有搜索

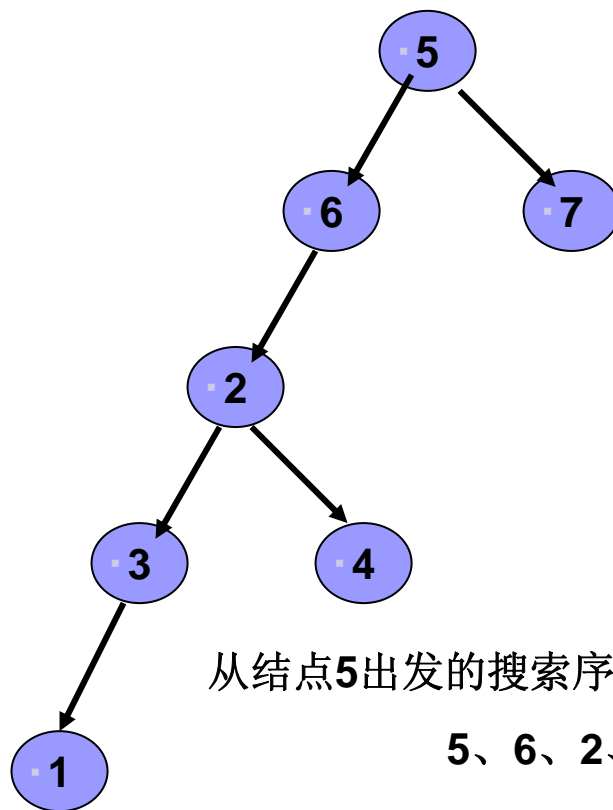
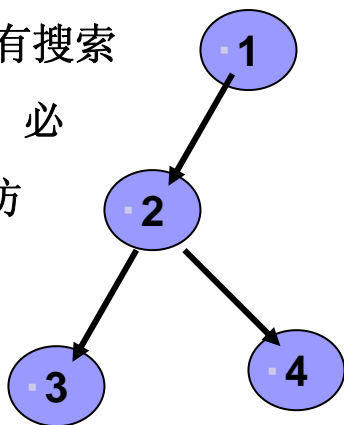
到所有的结点，必

须另选图中未访

问过的结点，

继续进行

搜索。

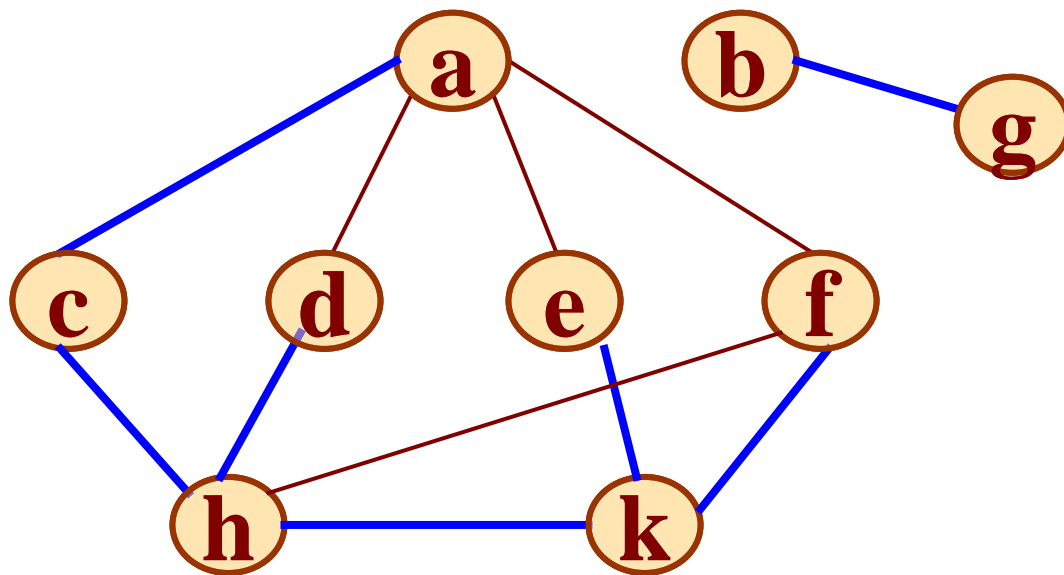


从结点**5**出发的搜索序列：

5、6、2、3、1、4、7

适用的数据结构：栈

例如：



0 1 2 3 4 5 6 7 8

访问标志：

T	T	T	T	T	T	T	T	T
---	---	---	---	---	---	---	---	---

访问次序：

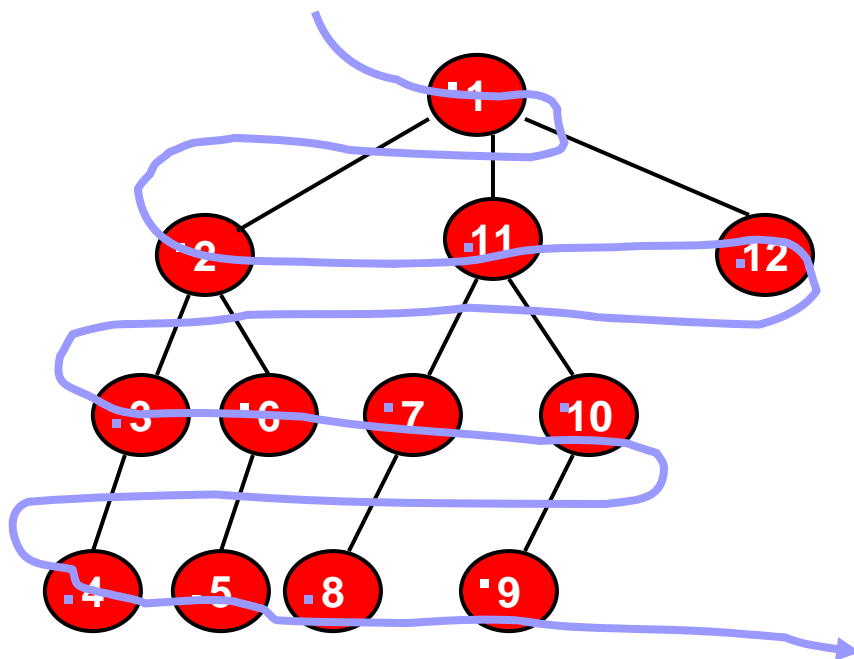
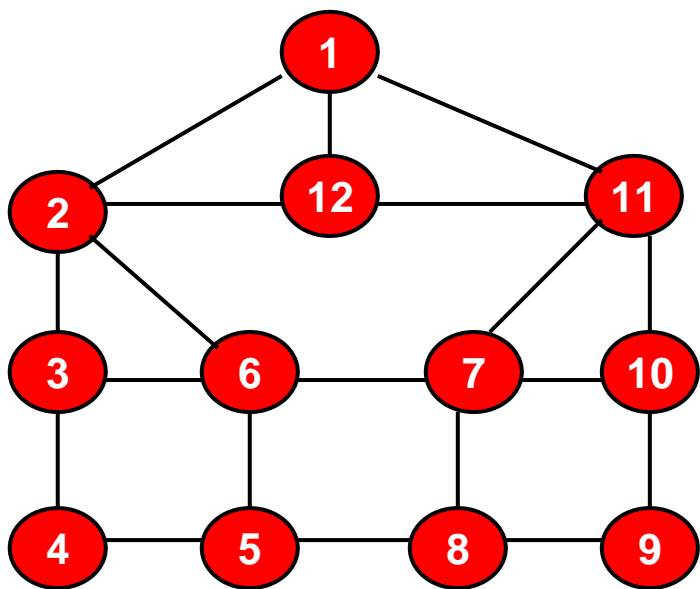
a	c	h	d	k	f	e	b	g
---	---	---	---	---	---	---	---	---

第七章 图——广度优先搜索遍历图

- 结点的邻接结点的次序是任意的，因此广度优先搜索的序列可能有多种。
- 广度优先搜索类似于树的从根出发的按层次遍历。
- 访问方式：
 1. 选中第一个被访问的结点 V
 2. 对结点 V 作已访问过的标志。
 3. 依次访问结点 V 的未被访问过的第一个、第二个、第三个……第 M 个邻接结点 W_1 、 W_2 、 W_3 …… W_m ，且进行标记。
 4. 依次访问结点 W_1 、 W_2 、 W_3 …… W_m 的邻接结点，且进行标记。
 5. 如果还有结点未被访问，则选中一个起始结点，也标记为 V ，转向 2。
 6. 所有的结点都被访问到，则结束。

第七章 图——广度优先搜索遍历图

- 例子：为了说明问题，邻接结点的访问次序以序号为准。序号小的先访问。
如：结点 1 的邻接结点有三个 2、12、11，则先访问结点 2、11，再访问12。



图的广度优先的访问次序：

1、2、11、12、3、6、7、10、4、5、8、9

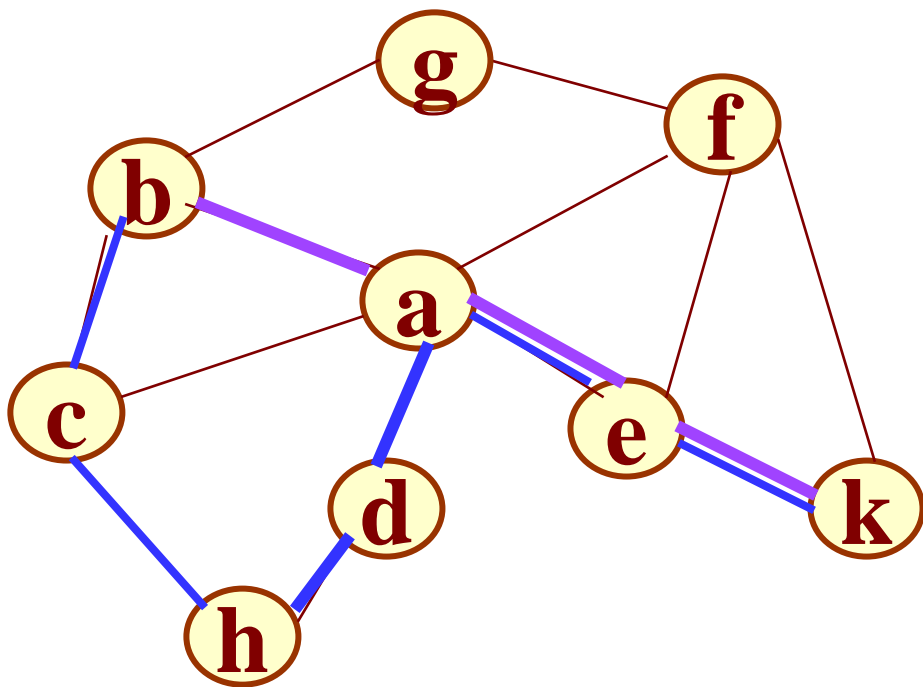
适用的数据结构：队列

第七章 图——遍历应用实例

问题： 求两个顶点之间的一条路径长度最短的路径

若两个顶点之间存在多条路径，则其中必有一条路径长度最短的路径。如何求得这条路径？

第七章 图——遍历应用实例



深度优先搜索访问顶点的次序取决于图的存储结构，而广度优先搜索访问顶点的次序是按“路径长度”渐增的次序。

因此，求路径长度最短的路径可以基于广度优先搜索遍历进行，但需要修改链队列的结点结构及其入队列和出队列的算法。

第七章 图——遍历应用实例

- 1) 将链队列的结点改为“双链”结点。即结点中包含next和prior两个指针;
- 2) 修改入队列的操作。插入新的队尾结点时, 令其prior域的指针指向刚刚出队列的结点, 即当前的队头指针所指结点;
- 3) 修改出队列的操作。出队列时, 仅移动队头指针, 而不将队头结点从链表中删除。

```
typedef DuLinkList QueuePtr;
void InitQueue(LinkQueue& Q) {
    Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
    Q.front->next = Q.rear->next = NULL
}
```

```
void EnQueue( LinkQueue& Q, QelemType e ) {
    p = (QueuePtr) malloc (sizeof(QNode));
    p->data = e; p->next = NULL;
    p->priou = Q.front
    Q.rear->next = p; Q.rear = p;
}
```

```
void DeQueue( LinkQueue& Q, QelemType& e ) {
    Q.front = Q.front->next; e = Q.front->data
}
```

第七章 图——最小生成树

■ 问题:

假设要在 n 个城市之间建立通讯联络网，则连通 n 个城市只需要修建 $n-1$ 条线路，如何在最节省经费的前提下建立这个通讯网？

■ 该问题等价于:

构造网的一棵最小生成树，即：在 e 条带权的边中选取 $n-1$ 条边（不构成回路），使“权值之和”为最小。

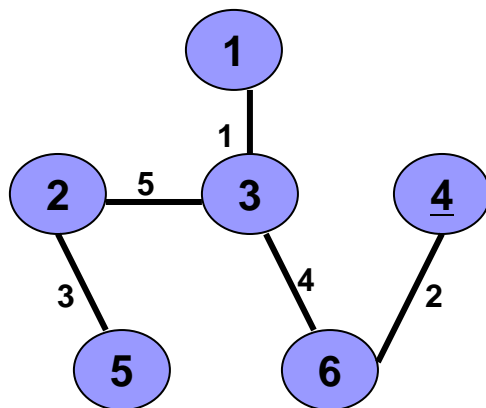
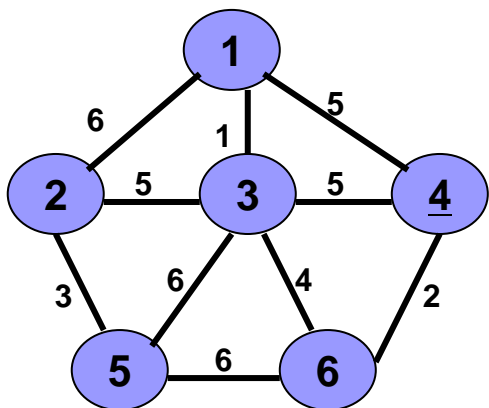
■ 解决方法:

算法一：普里姆（**Prim**）算法

算法二：克鲁斯卡尔（**Kruskal**）算法

第七章 图——最小生成树

- 定义：生成树中边的权值（代价）之和最小的树。
- 实例：

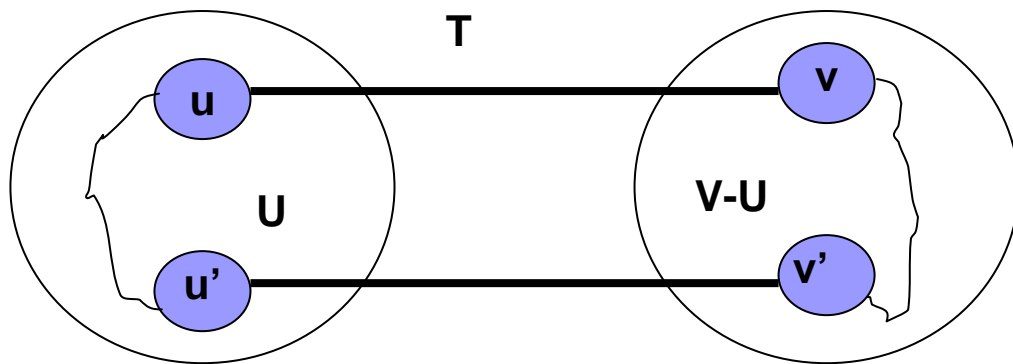


左图的最小代价生成树

第七章 图——最小生成树

■ MST 性质:

假设 $G = \{V, \{E\}\}$ 是一个连通图, U 是结点集合 V 的一个非空子集。若 (u, v) 是一条代价最小的边, 且 u 属于 U , v 属于 $V-U$, 则必存在一棵包括边 (u, v) 在内的最小代价生成树。



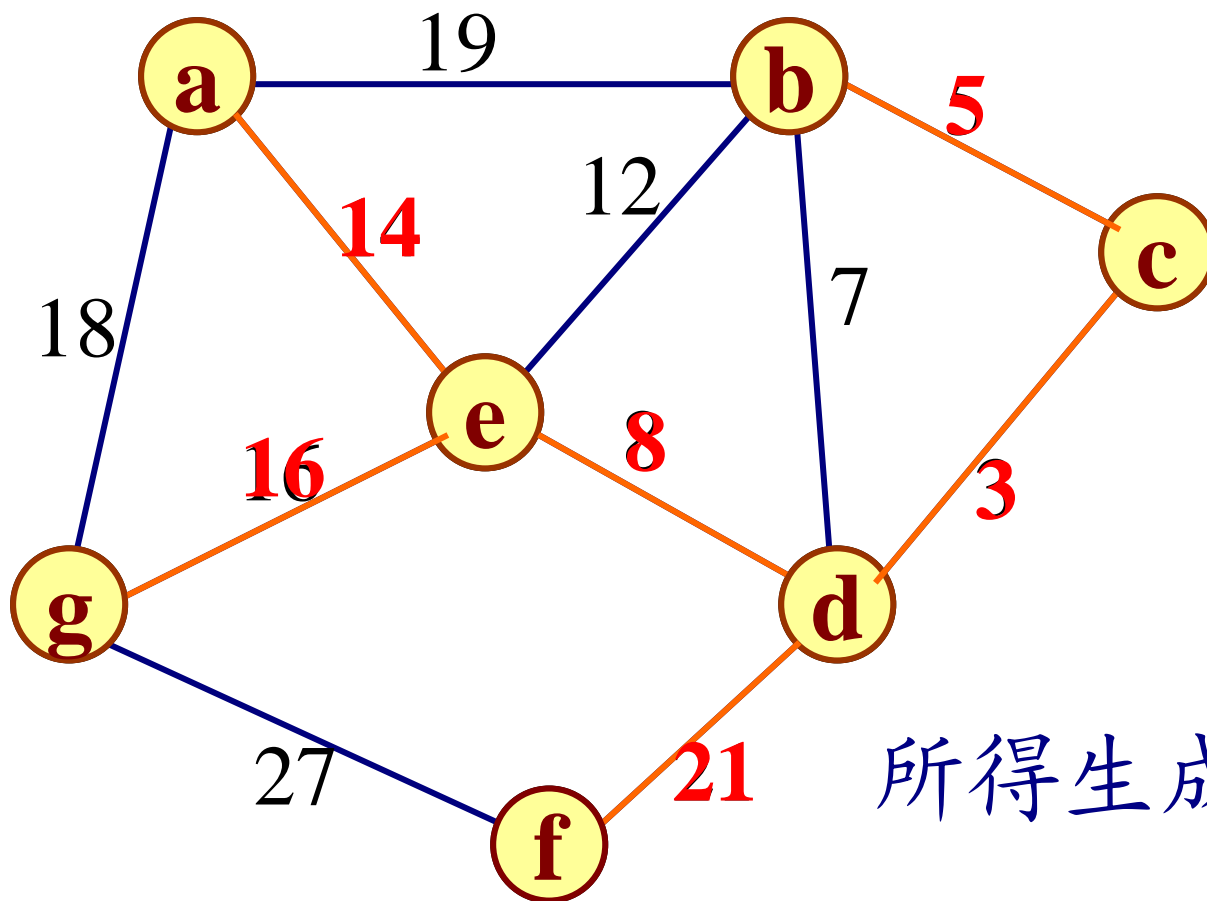
第七章 图——最小生成树 (Prim算法)

普里姆算法的基本思想:

取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w 。在添加的顶点 w 和已经在生成树上的顶点 v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小。之后继续往生成树上添加顶点，直至生成树上含有 $n-1$ 个顶点为止。

第七章 图——最小生成树 (Prim算法)

例如:



所得生成树权值和

$$= 14 + 8 + 3 + 5 + 16 + 21 = 67$$

第七章 图——最小生成树 (Prim算法)

■ Prim 算法的基本描述

设 **G: Graph**, **TE**: 最小生成树的边集

U, V: 顶点集; //初始时, **V** 包含所有顶点

u, v: 顶点

```
{  
    TE =  $\phi$ ;   U = {u0};  
    while ( U != V ) {  
        设(u,v)是一条代价最小的边, 并且u在U中, v在V-U中。  
        TE = TE  $\cup$  ( u,v );  
        U = U  $\cup$  { v };  
    }  
}
```

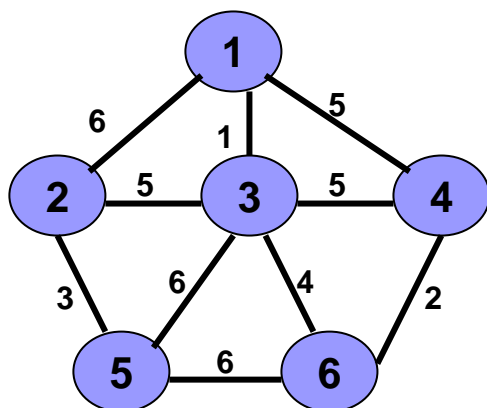
时间复杂性: **$O(n^2)$**

第七章 图——最小生成树（Prim算法）

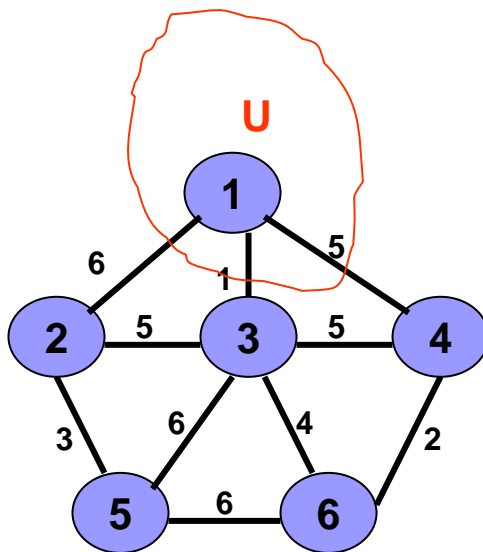
- 设置一个辅助数组**closedge**，对当前**V—U**集中的每个顶点，记录和顶点集**U**中顶点相连接的代价最小的边：

```
struct {  
    VertexType adjvex; // 该边所依附的U中的顶点序号  
    VRType lowcost; // 边的权值  
} closedge[MAX_VERTEX_NUM];
```

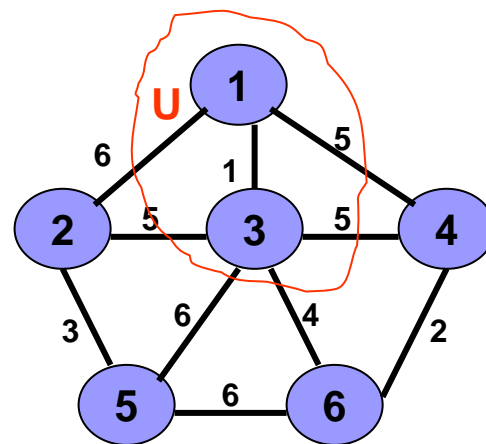
第七章 图——最小生成树 (Prim算法)



图G



图G



图G

注意:

closedge[0]. lowcost = 0且

closedge[2]. lowcost = 0 表示结点1和3 已在U中

数组: closedge[6]

0		0
1	0	6
2	0	1
3	0	5
4	0	∞
5	0	∞

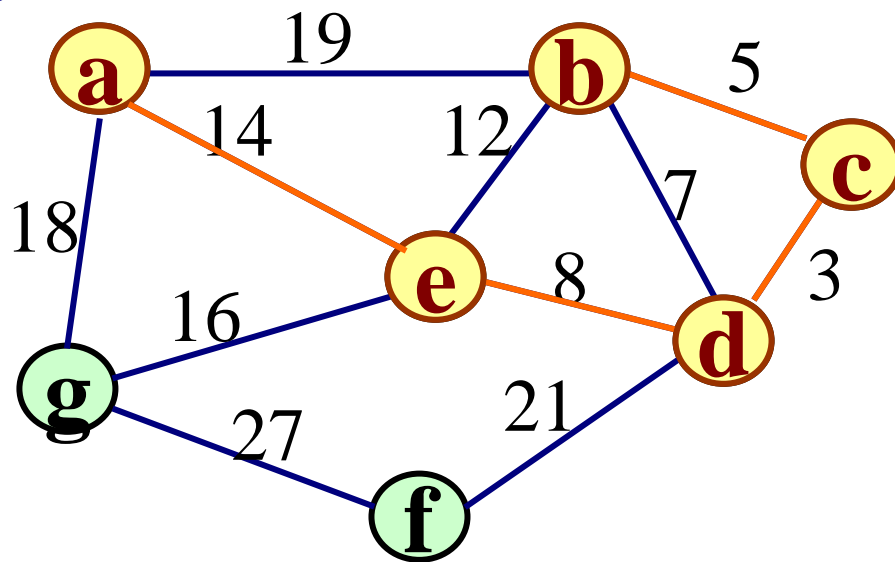
adjvex lowcost

数组: closedge[6]

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

adjvex lowcost

例如:

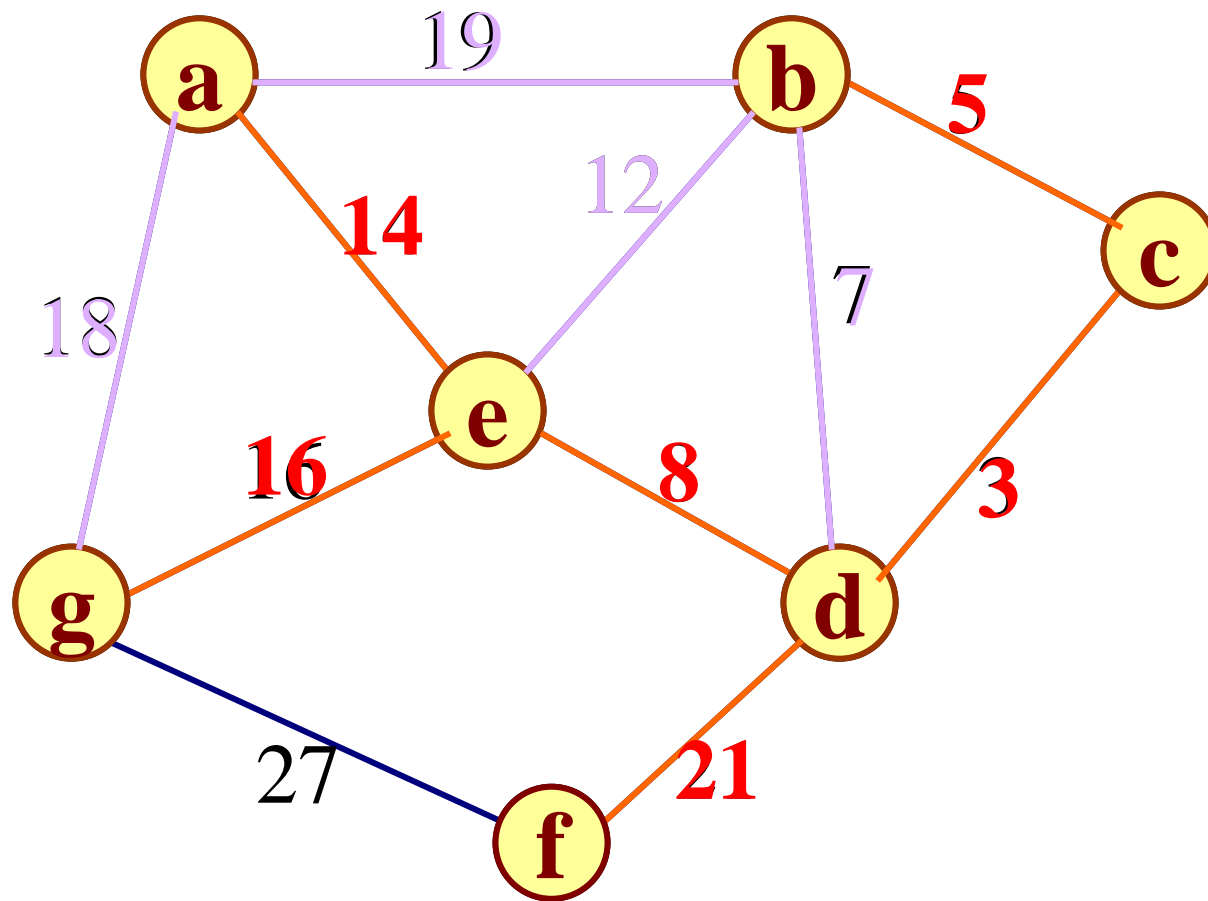


	0	1	2	3	4	5	6
closedge							
Adjvex		c	d	e	a	d	e
Lowcost		5	3	8	14	21	16

第七章 图——最小生成树（Kruscal算法）

- **克鲁斯卡尔算法考虑问题的出发点**：为使生成树上边的权值之和达到最小，则应使生成树中每一条边的权值尽可能地小。
- **具体做法**：先构造一个只含 n 个顶点的子图SG，然后从权值最小的边开始，若它的添加不使SG中产生回路，则在SG上加上这条边，如此重复，直至加上 $n-1$ 条边为止。

例如:



第七章 图——最小生成树 (Kruscal算法)

算法描述:

构造非连通图 $ST=(V,\{\ \});$

$k = i = 0;$ // k 计选中的边数

while ($k < n-1$) {

$++i;$

 检查边集 E 中第 i 条权值最小的边 $(u,v);$

 若 (u,v) 加入 ST 后不使 ST 中产生回路,

 则 输出边 $(u,v);$ 且 $k++;$

}

- 数据结构: 用堆实现, 则每次选择最小代价的边仅需 $O(\log e)$ 的时间。

- 时间复杂性: $O(e \log e)$

比较两种算法

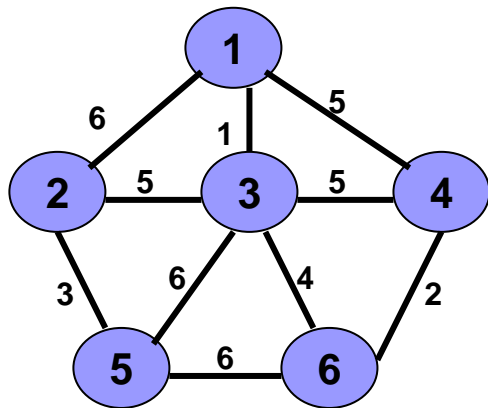
算法	普里姆算法	克鲁斯卡尔算法
----	-------	---------

时间复杂度	$O(n^2)$	$O(e \log e)$
-------	----------	---------------

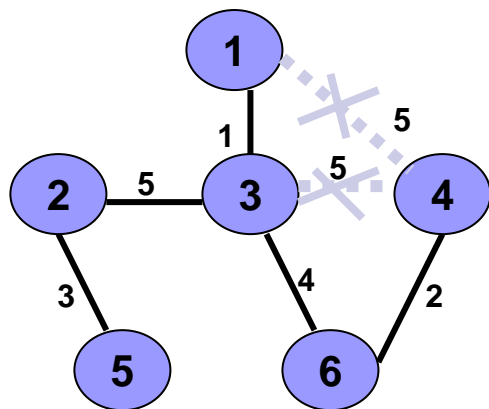
适应范围	稠密图	稀疏图
------	-----	-----

第七章 图——最小生成树（Kruscal算法）

• 实例的执行过程



图G



最小代价生成树

1、初始连通分量: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$

2、反复执行添加、放弃动作。条件: 边数不等于 $n-1$ 时

边	动作	连通分量
(1,3)	添加	$\{1,3\}, \{4\}, \{5\}, \{6\}, \{2\}$
(4,6)	添加	$\{1,3\}, \{4,6\}, \{2\}, \{5\}$
(2,5)	添加	$\{1,3\}, \{4,6\}, \{2,5\}$
(3,6)	添加	$\{1,3,4,6\}, \{2,5\}$
(1,4)	放弃	因构成回路
(3,4)	放弃	因构成回路
(2,3)	添加	$\{1,3,4,5,6,2\}$

第七章 图——拓扑排序

■ 问题:

假设以有向图表示一个工程的施工图或程序的数据流图，则图中不允许出现回路。

检查有向图中是否存在回路的方法之一，是对有向图进行拓扑排序。

第七章 图——拓扑排序

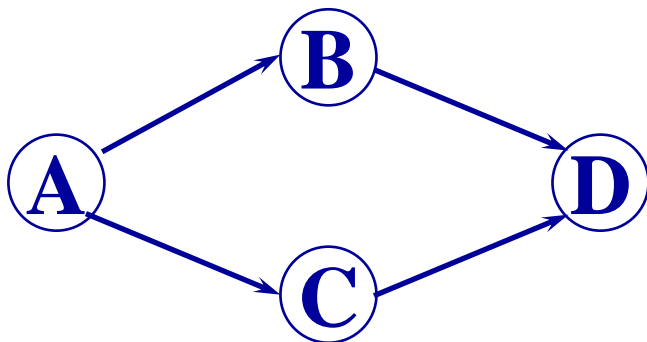
“拓扑排序”对有向图进行如下操作：

按照有向图给出的次序关系，将图中顶点排成一个线性序列，对于有向图中没有限定次序关系的顶点，则可以人为加上任意的次序关系。

由此所得顶点的线性序列称之为拓扑有序序列

第七章 图——拓扑排序

例如：对于下列有向图

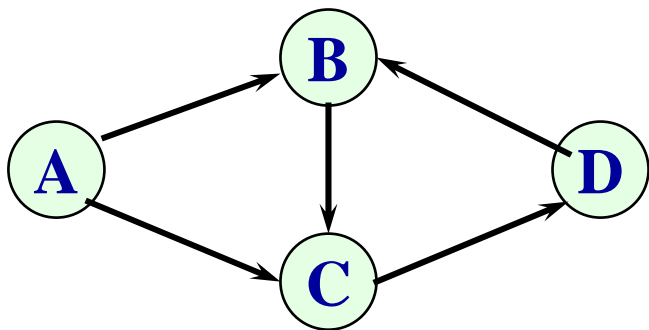


可求得拓扑有序序列：

A B C D 或 **A C B D**

第七章 图——拓扑排序

反之，对于下列有向图



不能求得它的拓扑有序序列。

因为图中存在一个回路 {B, C, D}

第七章 图——拓扑排序

■ 如何进行拓扑排序？

一、从有向图中选取一个没有前驱的顶点，并输出之；

二、从有向图中删去此顶点以及所有以它为尾的弧；

■ 重复上述两步，直至图空，或者图不空但找不到无前驱的顶点为止。

a b h c d g f e

在算法中需要用定量的描述替代定性的概念

没有前驱的顶点 \equiv 入度为零的顶点

删除顶点及以它为尾的弧 \equiv 弧头顶点的入度减1

第七章 图——关键路径

问题：——估算工程项目完成时间

假设以有向网表示一个施工流图，弧上的权值表示完成该项子工程所需时间。

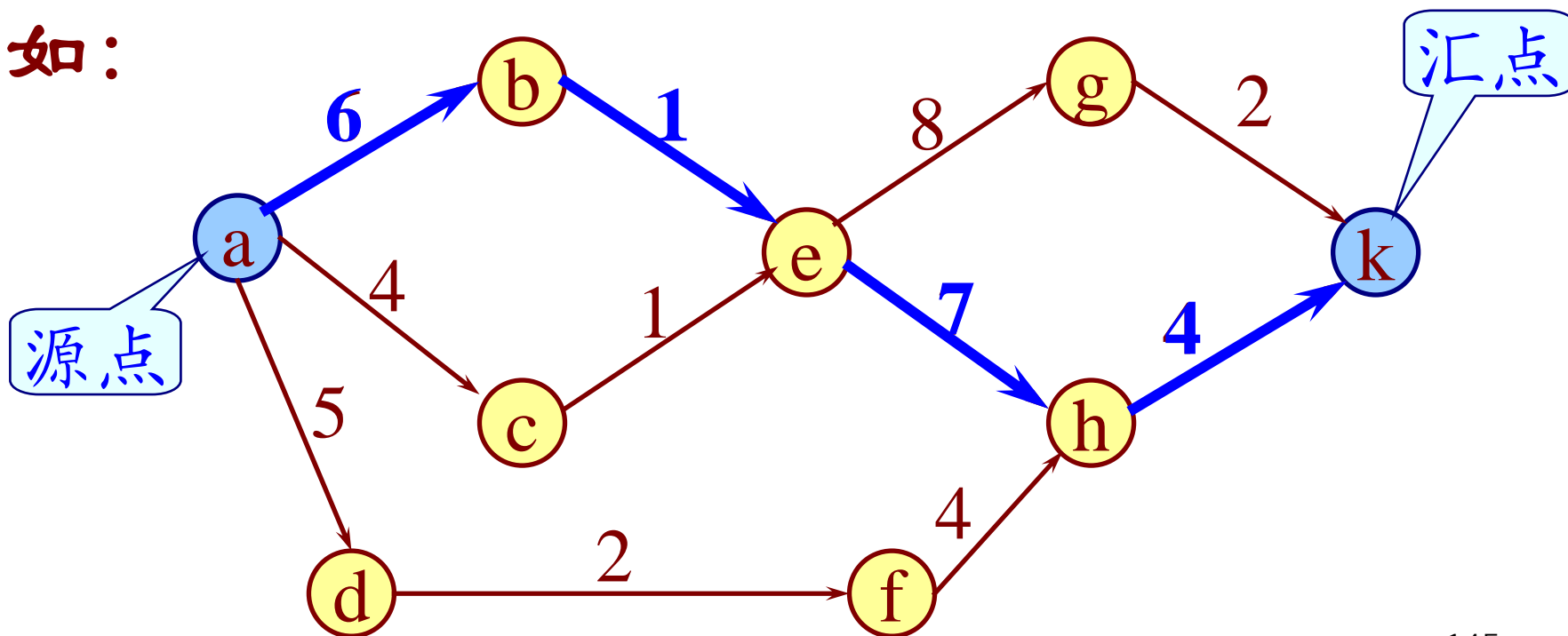
问：（1）完成整项工程至少需要多少时间？

（2）哪些子工程项是“**关键工程**”？即：哪些子工程项将影响整个工程的完成期限的。

第七章 图——关键路径

- 整个工程完成的时间为：从有向图的源点到汇点的最长路径。
- 路径长度是指路径上各活动持续时间之和，不是路径上弧的数目。
- 路径长度最长的路径叫做关键路径（**Critical Path**）。
- “关键活动”指的是：该弧上的权值增加将使有向图上的最长路径的长度增加。

例如：



第七章 图——关键路径

意味着事件最早能够发生的时刻。

如何求关键活动？

“事件(顶点)” 的最早发生时间 $ve(j)$

$ve(j)$ = 从源点到顶点j的最长路径长度；

“事件(顶点)” 的最迟发生时间 $vl(k)$

$vl(k)$ = 从顶点k到汇点的最短路径长度。

不影响工程的如期完工，本结点事件必须发生的时刻。

第七章 图——关键路径

- 假设第*i*条弧为<*j*, *k*>, 其持续时间为 $dut(<j,k>)$, 则对第*i*项活动:

“活动(弧)”的最早开始时间:

$$e(i) = ve(j)$$

“活动(弧)”的最迟开始时间:

$$l(i) = vl(k) - dut(<j,k>)$$

- 关键活动: 最早开始时间=最迟开始时间的活动
- 关键路径: 从源点到收点的最长的一条路径, 或者全部由关键活动构成的路径。

第七章 图——关键路径

■ 事件发生时间 $ve(j)$ 和 $vl(j)$ 的递推计算步骤:

(1) $ve(\text{源点}) = 0$;

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

$<i, j>$ 属于 T , 其中 T 是所有以第 j 个顶点为头的弧的集合, $j=1, 2, \dots, n-1$ 。

(2) $vl(\text{汇点}) = ve(\text{汇点})$;

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

$<i, j>$ 属于 S , 其中 S 是所有以第 i 个顶点为尾的弧的集合, $i=n-2, \dots, 0$ 。

。

第七章 图——关键路径

算法的实现要点:

- 求 ve 的顺序应该是按拓扑有序的次序;
- 求 vl 的顺序应该是按逆拓扑有序的次序;

因为逆拓扑有序序列即为拓扑有序序列的逆序列,
因此应该在拓扑排序的过程中, 另设一个“栈”记下
拓扑有序序列。

第七章 图——关键路径

• 实例：求事件结点的最早发生时间

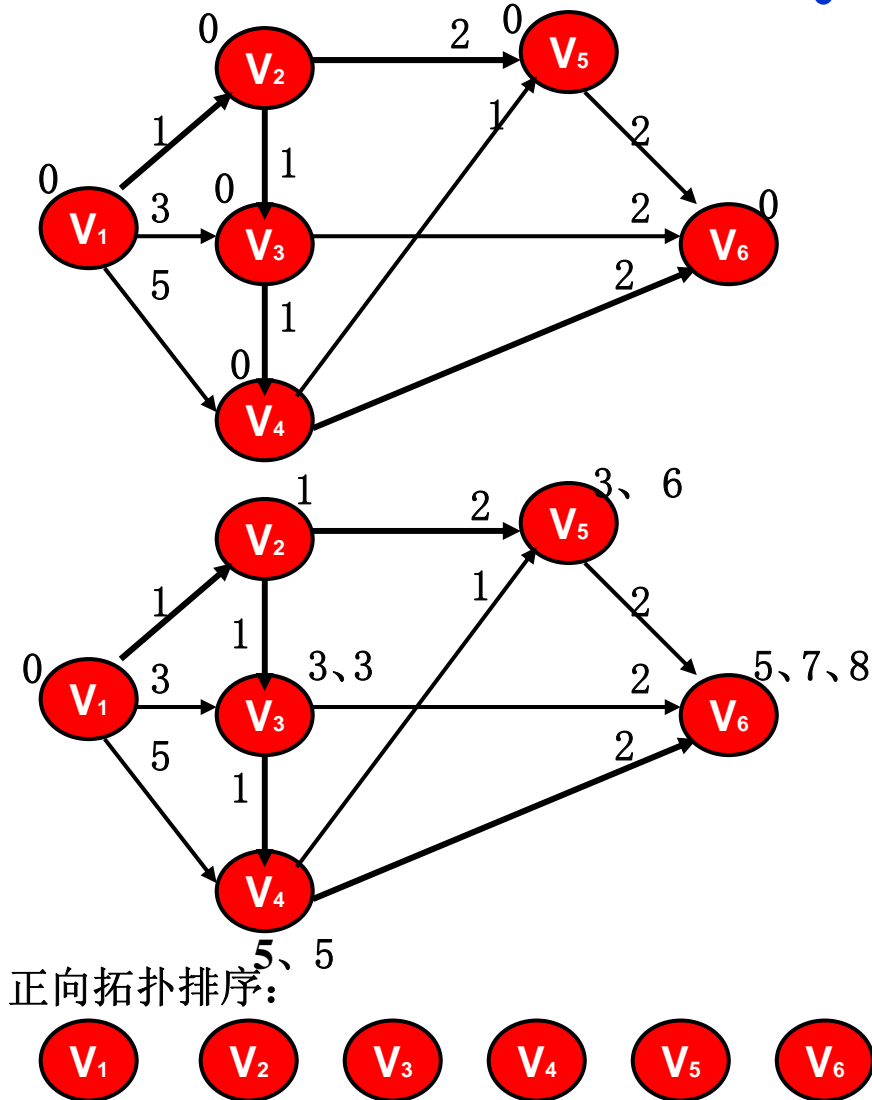
利用拓扑排序算法求事件结点的最早发生时间的执行步骤：

1、设每个结点的最早发生时间为**0**，将入度为零的结点**进栈**。

2、将栈中入度为零的结点**V**取出，并压入另一栈，用于形成逆向拓扑排序的序列。。

3、根据**邻接表**找到结点**V**的所有的邻接结点，将结点**V**的最早发生时间 + 活动的权值 得到的和同邻接结点的原最早发生时间进行比较；**如果该值大，则用该值取代原最早发生时间**。另外，将这些邻接结点的入度减一。如果某一结点的入度变为零，则进栈。

4、反复执行 2、3；直至栈空为止。

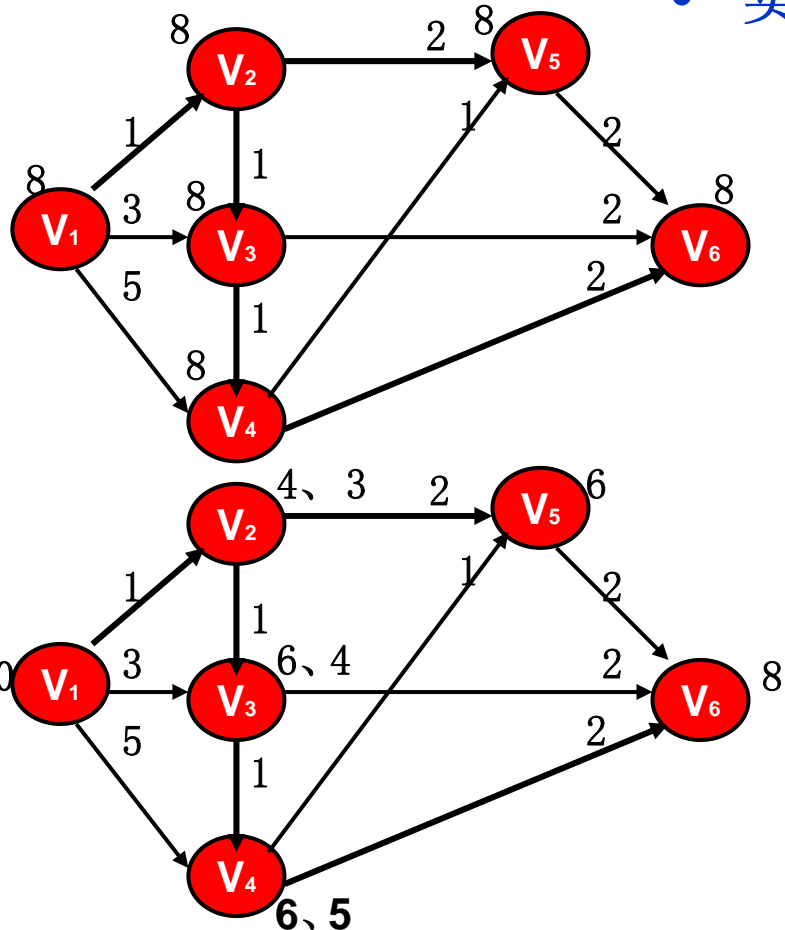


第七章 图——关键路径

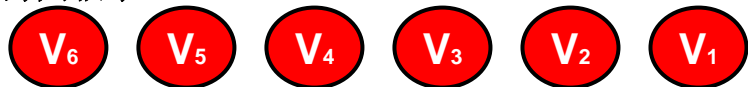
实例：求事件结点的最迟发生时间

利用**逆向**拓扑排序算法求事件结点的最迟发生时间的执行步骤：

- 1、设每个结点的最迟发生时间为收点的最早发生时间。
- 2、将栈中的结点**V**取出。
- 3、根据**逆邻接表**找到结点**V**的所有的起始结点，将结点**V**的最迟发生时间 - 活动的权值得到的差同起始结点的原最迟发生时间进行比较；如果该值小，则用该值取代原最迟发生时间。
- 4、反复执行 2、3；直至栈空为止。

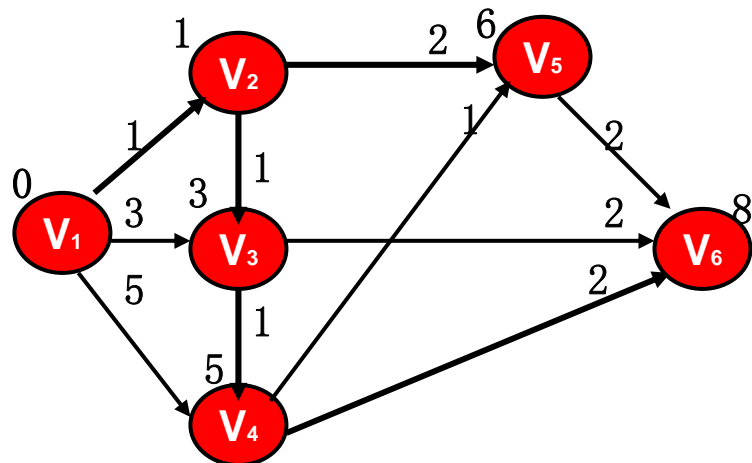


逆向拓扑排序：

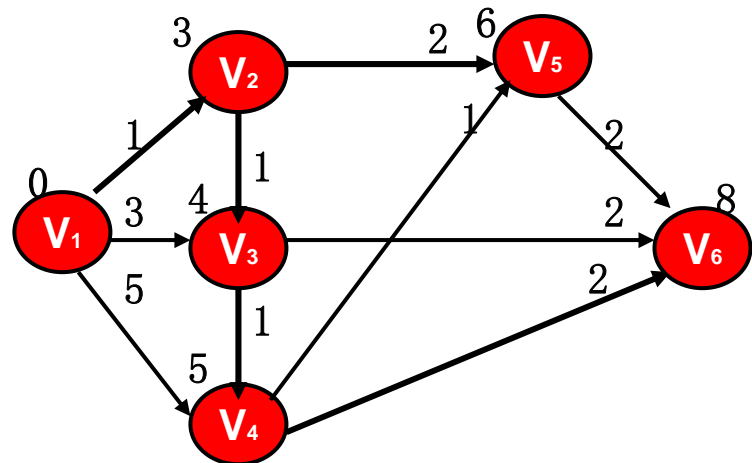


第七章 图——关键路径

- 实例的事件节点的最早发生时间



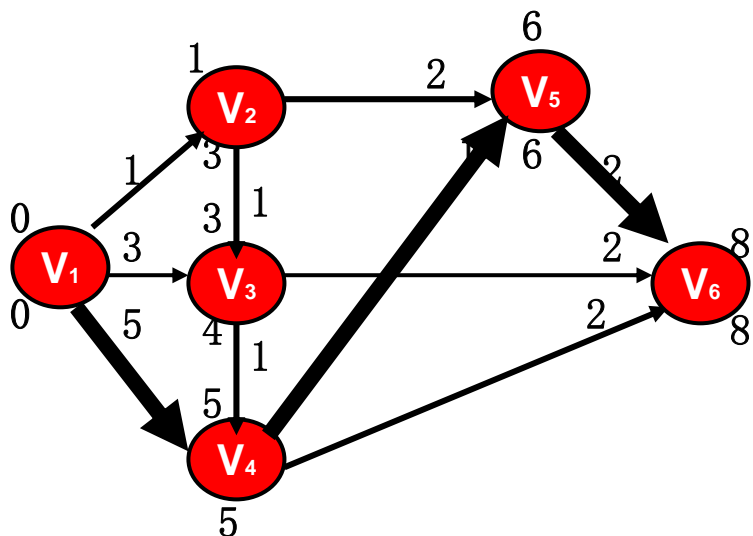
- 实例的事件节点的最迟发生时间



边	最早发生时间	最迟发生时间	
V ₁ → V ₂	0	2	
V ₁ → V ₃	0	1	
V ₁ → V ₄	0	0	关键活动
V ₂ → V ₃	1	3	
V ₂ → V ₅	1	4	
V ₃ → V ₄	3	4	
V ₃ → V ₆	3	6	
V ₄ → V ₅	5	5	关键活动
V ₄ → V ₆	5	6	
V ₅ → V ₆	6	6	关键活动

第七章 图——关键路径

- 实例的关键路径（粗大的黑色箭头所示）



- 算法7.14的时间复杂度为 $O(n+e)$ ，其中计算活动最早开始时间和最迟开始时间的复杂度均为 $O(e)$ 。
- 注意：关键路径可有多条。
- 缩短工期必须缩短关键活动所需的时间。

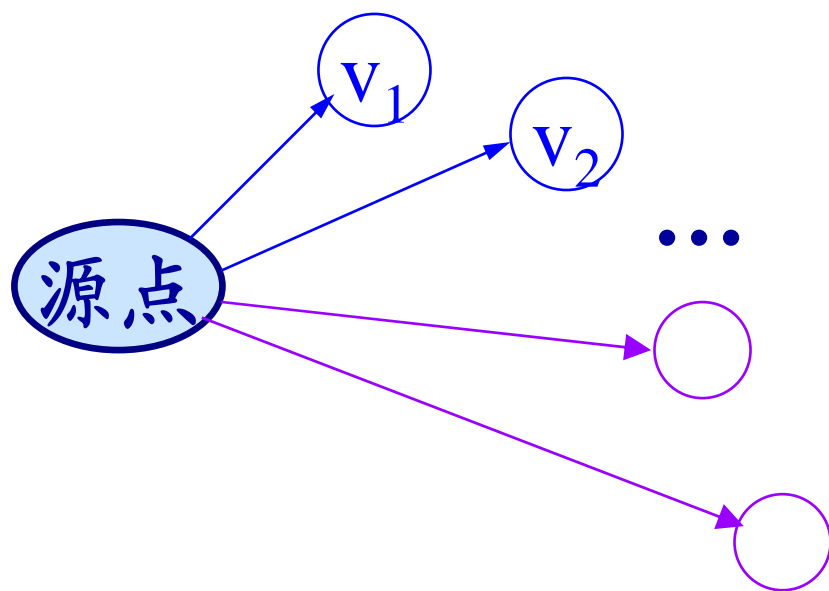
第七章 图——最短路径

- 求从某个源点到其余各点的最短路径
- 每一对顶点之间的最短路径

第七章 图——求从某个源点到其余各顶点的最短路径

算法的基本思想:

依最短路径的长度递增的次序求得各条路径



其中，从源点到
顶点 v 的最短路径
是所有最短路径
中长度最短者。

第七章 图——求从某个源点到其余各顶点的最短路径

■ 求最短路径的迪杰斯特拉(Dijkstra)算法:

设置辅助数组**Dist**，其中每个分量**Dist[k]** 表示当前所求得的从源点到其余各顶点 **k** 的最短路径。

■ 一般情况下，

$$\begin{aligned} \text{Dist}[k] &= \langle \text{源点到顶点 } k \text{ 的弧上的权值} \rangle \\ &\text{或者} = \langle \text{源点到其它顶点的路径长度} \rangle \\ &+ \langle \text{其它顶点到顶点 } k \text{ 的弧上的权值} \rangle。 \end{aligned}$$

第七章 图——求从某个源点到其余各顶点的最短路径

1) 在所有从源点出发的弧中选取一条权值最小的弧，即为第一条最短路径。

$$Dist[k] = \begin{cases} arcs[v0][k] & \text{V0和k之间存在弧} \\ \infty & \text{V0和k之间不存在弧} \end{cases}$$

其中的最小值即为最短路径的长度。

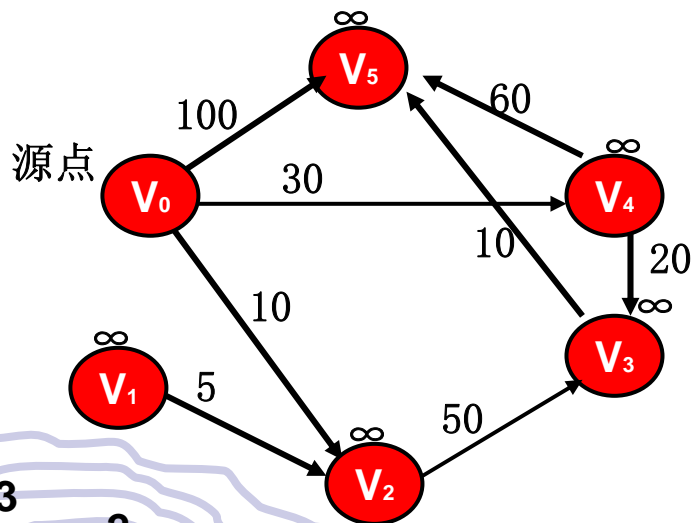
2) 修改其它各顶点的 $Dist[k]$ 值。

假设求得最短路径的顶点为 u ，若

$$Dist[u] + arcs[u][k] < Dist[k]$$

则将 $Dist[k]$ 改为 $Dist[u] + arcs[u][k]$ 。

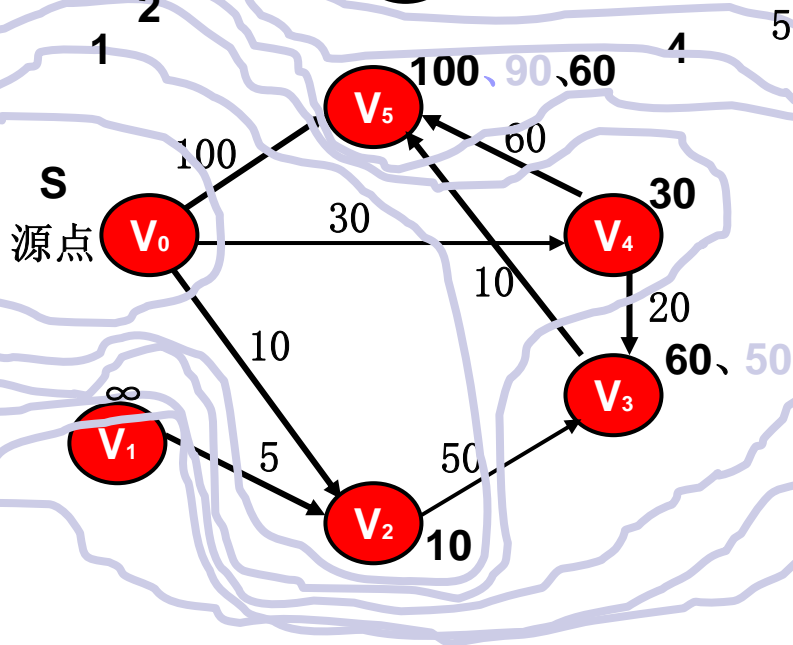
第七章 图——求从某个源点到其余各顶点的最短路径



Dijkstra 算法求单源最短路径:

设 V 是该有向图的结点的集合、集合 S 是已求得最短路径的结点的集合, 求 V_0 至其余各结点的最短距离。

- 1、 $S = \{0\}$; // 结点 V_0 最短路径已求得
- 2、for ($i=1; i<n; i++$)
- 3、 $D[i]=c[0, i]$; // V_0 至 V_i 的边长
- 4、for ($i=1; i<n; i++$)
- 5、{ 在 $V-S$ 中选择一个结点 V_w ; 使得 $D[w]$ 最小。将 W 加入集合 S
- 6、for (每一个在 $V-S$ 中的结点 V)
- 7、{ $D[v]=\text{MIN}(D[v], D[w]+C[w,v])$;
- 8、}
- 9、}



第七章 图——每一对顶点之间的最短路径（Floyd算法）

弗洛伊德（Floyd）算法的基本思想是：

从 V_i 到 V_j 的所有
可能存在的路径中，选
出一条长度最短的路径。

若 $\langle v_i, v_j \rangle$ 存在, 则存在路径 $\{v_i, v_j\}$

// 路径中不含其它顶点

若 $\langle v_i, v_1 \rangle, \langle v_1, v_j \rangle$ 存在, 则存在路径 $\{v_i, v_1, v_j\}$

// 路径中所含顶点序号不大于1

若 $\{v_i, \dots, v_2\}, \{v_2, \dots, v_j\}$ 存在,

则存在一条路径 $\{v_i, \dots, v_2, \dots, v_j\}$

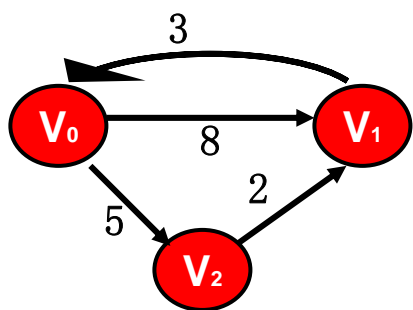
// 路径中所含顶点序号不大于2

...

依次类推, 则 v_i 至 v_j 的最短路径应是上述这些路径中, 路径长度最小者。

第七章 图——每一对顶点之间的最短路径（Floyd算法）

- 加权有向图：求各结点之间的最短路径



	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0

右图的代价矩阵 C

Floyd 算法的求解过程

设 C 为 n 行 n 列的代价矩阵， $c[i, j]$ 为 $i \rightarrow j$ 的权值。如果 $i=j$ ；那么 $c[i, j] = 0$ 。如果 i 和 j 之间无有向边；则 $c[i, j] = \infty$

- 1、使用 n 行 n 列的矩阵 A 用于计算最短路径。

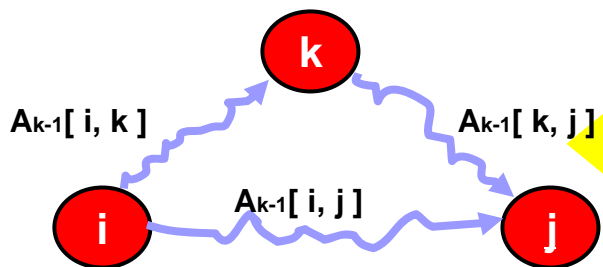
初始时， $A[i, j] = c[i, j]$

- 2、进行 n 次迭代

在进行第 k 次迭代时，我们将使用如下的公式：

$$A_k[i, j] = \min \begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases}$$

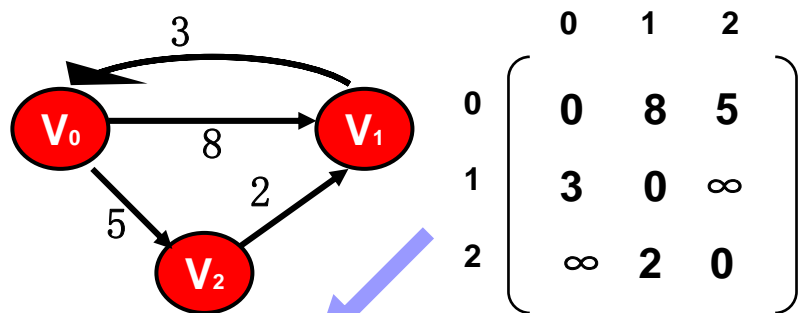
注意：第 k 次迭代时，针对结点 k 进行。原 A_{k-1} 矩阵的第 k 行，第 k 列保持不变。左上至右下的对角线元素也不变。



k 从 0 取到 $n-1$ ，表示路经前 k 个顶点是所能找到的最短路径。

第七章 图——每一对顶点之间的最短路径（Floyd算法）

实例及求解过程:



$$\begin{matrix} & \begin{matrix} 0 & 1 & 2 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} & \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix} \end{matrix}$$

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix}$$

$A_{\text{初值}}$

A_0

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

A_1

A_2

Floyd 算法的求解过程

设 C 为 n 行 n 列的代价矩阵, $c[i, j]$ 为 $i \rightarrow j$ 的权值。如果 $i=j$; 那么 $c[i, j] = 0$ 。如果 i 和 j 之间无有向边; 则 $c[i, j] = \infty$

1、使用 n 行 n 列的矩阵 A 用于计算最短路径。

初始时, $A[i, j] = c[i, j]$

2、进行 n 次迭代

在进行第 k 次迭代时, 我们将使用如下的公式:

$$A_k[i, j] = \min \begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases}$$

注意: 第 k 次迭代时, 针对结点 k 进行。原 A_{k-1} 矩阵的第 k 行, 第 k 列保持不变。左上至右下的对角线元素也不变。

第七章 图——每一对顶点之间的最短路径（Floyd算法）

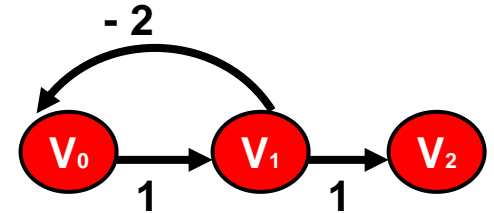
- 算法的程序实现:

Floyd 算法的程序的简单描述:

```
int i, j, k; // 代表结点
for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ ) A[ i, j ] = C[ i, j ];
for ( i = 0; i < n; i++ ) A[ i, i ] = 0;
for ( k = 0; k < n; k++ )
    for ( i = 0; i < n; i++ )
        for ( j = 0; j < n; j++ )
            if ( A[ i, k ] + A[ k, j ] < A[ i, j ] )
                A[ i, j ] = A[ i, k ] + A[ k, j ];
```

- 算法的时间代价: $O(n^3)$

- 应用范围: 无负长度的圈


$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

A初值

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & -1 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

A₀

$A_1[0, 2] :=$
 $\text{MIN} \{ A_0[0, 2],$
 $A_0[0, 1]$
 $+ A_0[1, 2] \}$

因为 $V_0 \rightarrow V_1 \rightarrow V_0$
可以有許多圈。

第九章 查找

- 在不少数据结构的教材中，是把查找与排序放入高级数据结构中的。应该说，查找和排序两章是前面我们所学的知识综合运用，用到了树、也用到了链表等知识，对这些数据结构某一方面的运用就构成了查找和排序。
- 现实生活中，**search**几乎无处不在，特别是现在的网络时代，万事离不开**search**，小到文档内文字的搜索，大到**INTERNET**上的搜索，**search**占据了我們上网的大部分时间。
- 在复习这一章的知识时，你需要先弄清楚以下几个概念：
 - 关键字、主关键字、次关键字的含义；静态查找与动态查找的含义及区别；平均查找长度**ASL**的概念及在各种查找算法中的计算方法和计算结果，特别是一些典型结构的**ASL**值，应该记住。
 - 在**DS**的教材中，一般将**search**分为三类：**1) 在顺序表上的查找；2) 在树表上的查找；3) 在哈希表上的查找。**

- 下面详细介绍其考查知识点及考查方式：

1.线性表上的查找：

- 主要分为三种线性结构：**顺序表**，**有序顺序表**，**索引顺序表**。对于第一种，我们采用传统查找方法，逐个比较。对于有序顺序表我们采用二分查找法。对于第三种索引结构，我们采用索引查找算法。大家需要注意这三种表下的**ASL**值以及三种算法的实现。其中，二分查找还要特别注意适用条件以及其递归实现方法。

2.树表上的查找：

- 这是本章的重点和难点。由于这一节介绍的内容是使用树表进行的查找，所以很容易与树一章的某些概念相混淆。本节内容与树一章的内容有联系，但也有很多不同，应注意归纳。树表主要分为以下几种：**二叉排序树**，**平衡二叉树**，**B树**，**键树**。其中，尤以前两种结构为重。由于二叉排序树与平衡二叉树是一种特殊的二叉树，所以与二叉树的联系就更为紧密，二叉树一章学好了，这里也就不难了。

- **二叉排序树**，简言之，就是“左小右大”，它的中序遍历结果是一个递增的有序序列。常考给定记录序列，建立相应的二叉排序树及其查找长度分析。**平衡二叉树**是二叉排序树的优化，其本质也是一种二叉排序树，只不过，平衡二叉树对左右子树的深度有了限定：深度之差的绝对值不得大于1。对于二叉排序树，“判断某棵二叉树是否二叉排序树”这一算法经常被考到，可用递归，也可以用非递归。平衡二叉树的建立也是一个常考点，但该知识点归根结底还是关注的平衡二叉树的四种调整算法，所以应该掌握平衡二叉树的四种调整算法，调整的一个参照是：**调整前后的中序遍历结果相同**。
- **B树**是二叉排序树的进一步改进，也可以把B树理解为三叉、四叉....排序树，主要用于文件系统，减少外存读次数。除B树的查找算法外，应该特别注意一下B树的插入和删除算法。因为这两种算法涉及到B树结点的分裂和合并，是一个难点。另外，也有考B树的结点数与高度相关关系的题目。B树是报考名校的同学应该关注的焦点之一。
- **键树**也称字符树，特别适用于查找英文单词的场合。一般不要求能完整描述算法源码，多是根据算法思想建立键树及描述其大致查找过程。不列入期末考范围。

3.基本哈希表的查找算法:

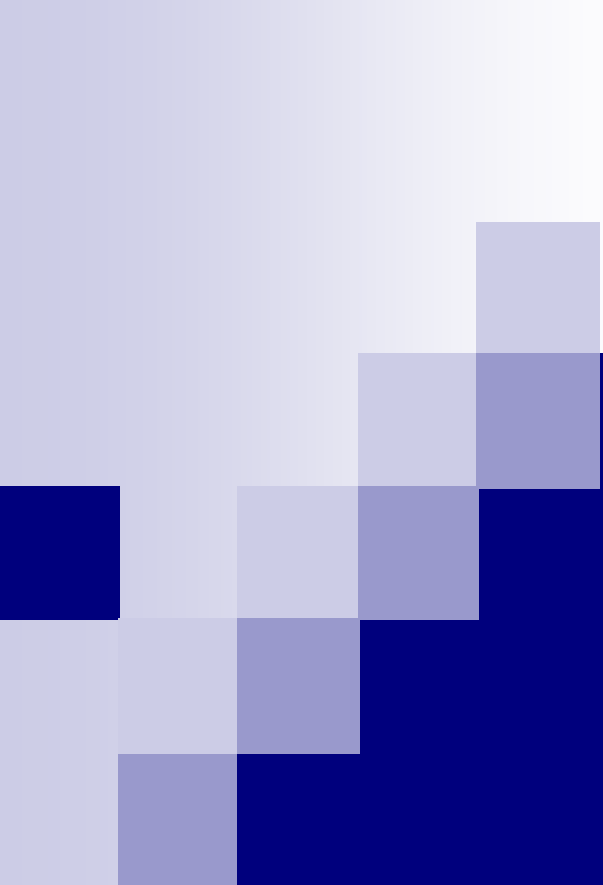
- 哈希一词，是外来词，译自“hash”一词，意为：散列或杂凑的意思。哈希表查找的基本思想是：根据当前待查找数据的特征，以记录关键字为自变量，设计一个 **function**，该函数对关键字进行转换后，其解释结果为待查的地址。基于哈希表的考查点有：哈希函数的设计，冲突解决方法的选择及冲突处理过程的描述。

第十章 内部排序

- 内排是DS课程中最后一个重要的章节，建立在此章之上的考题可以有多种类型：填空，选择，判断乃至大型算法题。但是，归结到一点，就是考查你对书本上的各种排序算法及其思想以及其优缺点和性能指标（时间复杂度）能否了如指掌。
- 这一章，我们对重点的归纳将跟以上各章不同。我们将从以下几个侧面来对排序一章进行不同的规纳，以期能更全面的理解排序一章的总体结构及各种算法。
- 从排序算法的种类来分，本章主要阐述了以下几种排序方法：**插入、选择、交换、归并、基数**等五种排序方法。
- 1. **在插入排序中又可分为：**直接插入、折半插入、2路插入、希尔排序。
 - 这几种插入排序算法的最根本的不同点，说到底就是**根据什么规则寻找新元素的插入点**。直接插入是依次寻找，折半插入是折半寻找。希尔排序，是通过控制每次参与排序的数的总范围“由小到大”的增量来实现排序效率提高的目的。

2. **交换排序**，又称冒泡排序，在交换排序的基础上改进又可以得到快速排序。**快速排序**的思想，一语以蔽之：用中间数将待排数据组一分为二。快速排序，在处理的“问题规模”这个概念上，与希尔有点相反，快速排序，是先处理一个较大规模，然后逐渐把处理的规模降低，最终达到排序的目的。
3. **选择排序**，相对于前面几种排序算法来说，难度大一点。具体来说，它可以分为：**简单选择、树选择、堆排**。这三种方法的不同点是，**根据什么规则选取最小的数**。简单选择，是通过简单的数组遍历方案确定最小数；树选择，是通过“锦标赛”类似的思想，让两数相比，不断淘汰较大（小）者，最终选出最小（大）数；而堆排序，是利用堆这种数据结构的性质，通过堆元素的删除、调整等一系列操作将最小数选出放在堆顶。堆排序中的堆建立、堆调整是重要考点。

4. **归并排序**，故名思义，是通过“归并”这种操作完成排序的目的，既然是归并就必须是两者以上的数据集合才可能实现归并。所以，在归并排序中，关注最多的就是2路归并。算法思想比较简单，有一点，要铭记在心：**归并排序是稳定排序**。
5. **基数排序**，是一种很特别的排序方法，也正是由于它的特殊，所以，基数排序就比较适合于一些特别的场合，比如扑克牌排序问题等。基数排序，又分为两种：多关键字的排序（扑克牌排序），链式排序（整数排序）。基数排序的核心思想也是利用“基数空间”这个概念将问题规模规范、变小，并且，在排序的过程中，只要按照基排的思想，是不用进行关键字比较的，这样得出的最终序列就是一个有序序列。
- 本章各种排序算法的思想以及伪代码实现，及其时间复杂度都是必须掌握的，学习时要注意规纳、总结、对比。此外，对于教材中的**10.7节**，要求必须熟记，在理解的基础上记忆。



最后祝同学们期末考试顺利！