数据结构

陈小柏
chenxb86@njupt.edu.cn

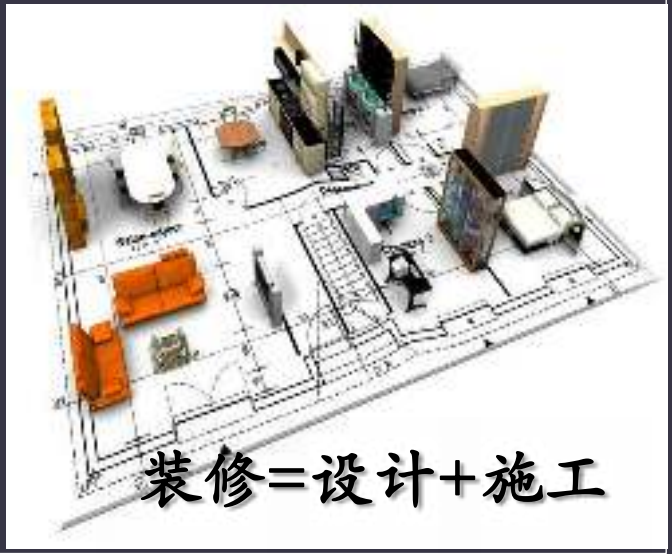
目录

- 算法与数据结构
- 数据结构的定义
- 数据抽象和抽象数据类型
- 描述数据结构和算法
- 算法分析的基本方法

算法和数据结构

- 数据：计算机加工处理的对象
- 数据结构：存在特定关系的数据元素集合
- 算法：一系列解决问题的清晰指令
- 数据结构与算法的关系
  - ✓数据结构是算法的实现基础
  - ✓精心选择的数据结构可以提高算法效率

程序=数据结构+算法



目录

- 算法与数据结构
- **数据结构的定义**
- 数据抽象和抽象数据类型
- 描述数据结构和算法
- 算法分析的基本方法

数据结构的基本概念

- 数据（Data）：计算机加工的对象，是数据元素的集合
- 数据元素（Data Element）：数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理
- 数据项（Data Item）：不可再分割

商品	材质	价格
组合沙发	牛皮	7000
餐桌椅	榉木	5000
衣橱	橡木	3000

数据结构定义

- 定义1：相互之间存在一种或多种特定关系的数据元素的集合。
- 定义2：数据结构是数据对象（Data Object，实例或值的集合），以及存在于该对象的实例和组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。
- 定义3：按某种逻辑关系组织起来的数据元素的集合，使用计算机语言描述并按一定的存储方式存储在计算机中，并在其上定义了一组运算

数据结构 {
逻辑结构：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形式
运算结构：在数据上执行的操作

数据的逻辑结构

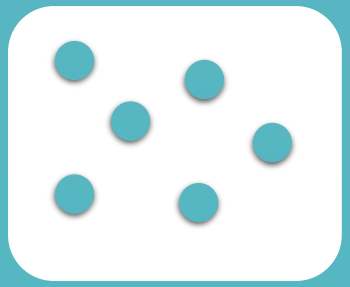
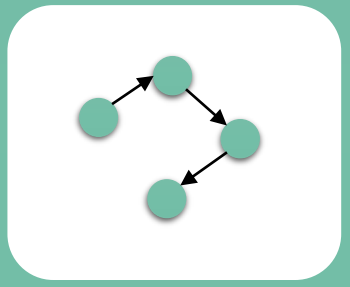
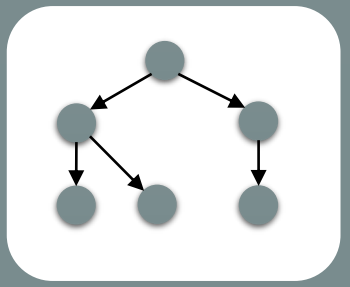
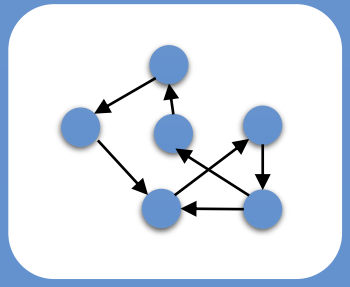
- 数据的逻辑结构是从具体问题抽象出来的数学模型，是描述数据元素及其关系的数学特性的，有时就把逻辑结构简称为数据结构
- 二元组表示　DS = (D, R)

D是数据元素的有限集合，R是D中数据元素序偶的集合

D={a, b, c, d}, R={<a, b>, <c, d>, <b, c>}

直接前驱 直接后继 边

基本逻辑结构

集合结构	线性结构	树形结构	图形结构
			
无关系 0前驱 and 0后继	一对一关系 1前驱 or 1后继	一对多关系 根：0前驱n后继 非根：1前驱 n后继 路径唯一	多对多关系 n前驱 or n后继

D={a, b, c, d}, R={}

D={a, b, c, d}, R={<a, b>, <c, d>, <b, c>}

D={a, b, c, d}, R={<a, b>, <a, d>, <a, c>}

D={a, b, c, d}, R={<a, b>, <a, c>, <d, a>, <d, c>}

数据结构示例

商品	材质	价格
组合沙发	牛皮	7000
餐桌椅	榉木	5000
衣橱	橡木	3000

逻辑结构：线性结构
按照价格从高到低排列

数据的存储表示

- 存储结构:数据结构的实现形式，是数据结构在计算机内的表示，即数据元素及其关系在计算机存储器中的存储方式
- 数据元素的机内表示： 用二进制位（bit）的位串表示数据元素
- 关系的机内表示
  - 顺序存储结构★
  - 链接存储结构★
  - 索引存储结构
  - 散列存储结构

顺序存储结构

- 将逻辑上相关的数据元素依次存储在一块连续的存储空间中

	商品	材质	价格
a_0	组合沙发	牛皮	7000
a_1	餐桌椅	榉木	5000
a_2	衣橱	橡木	3000

设：申请到的连续空间首地址为L

每个数据元素占A个存储单元

$Location(a_k) = L + A * k$

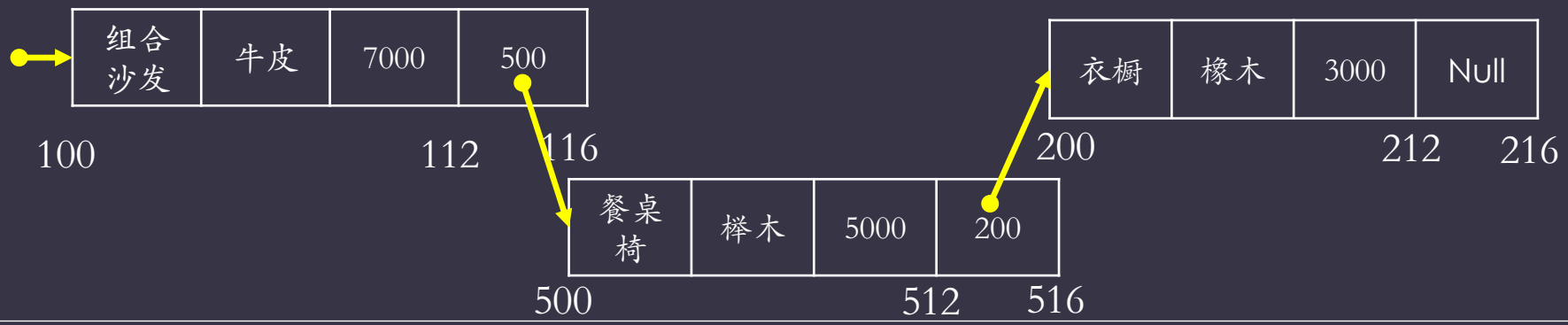
线性结构的顺序存储方式

链接存储结构

- 存储数据元素的**结点**，除了存放数据元素本身外，同时还存放了与该数据元素有关系的其它数据元素地址信息

	商品	材质	价格
a_0	组合沙发	牛皮	7000
a_1	餐桌椅	榉木	5000
a_2	衣橱	橡木	3000

a_2	Null
a_1	Location(a_2)
a_0	Location(a_1)



索引/散列存储结构

- 索引：建立存储结点信息外，还建立附加的索引表来标识结点的地址
- 散列：根据结点的关键字直接计算出该结点的存储地址

  若关键字为k，则其值存放在f(k)的存储位置上

数据结构的运算

创建运算

商品	材质	价格

插入运算

商品	材质	价格
组合沙发	牛皮	7000
餐桌椅	榉木	5000
衣橱	橡木	3000

清除运算

商品	材质	价格
组合沙发	牛皮	10000
餐桌椅	榉木	5000

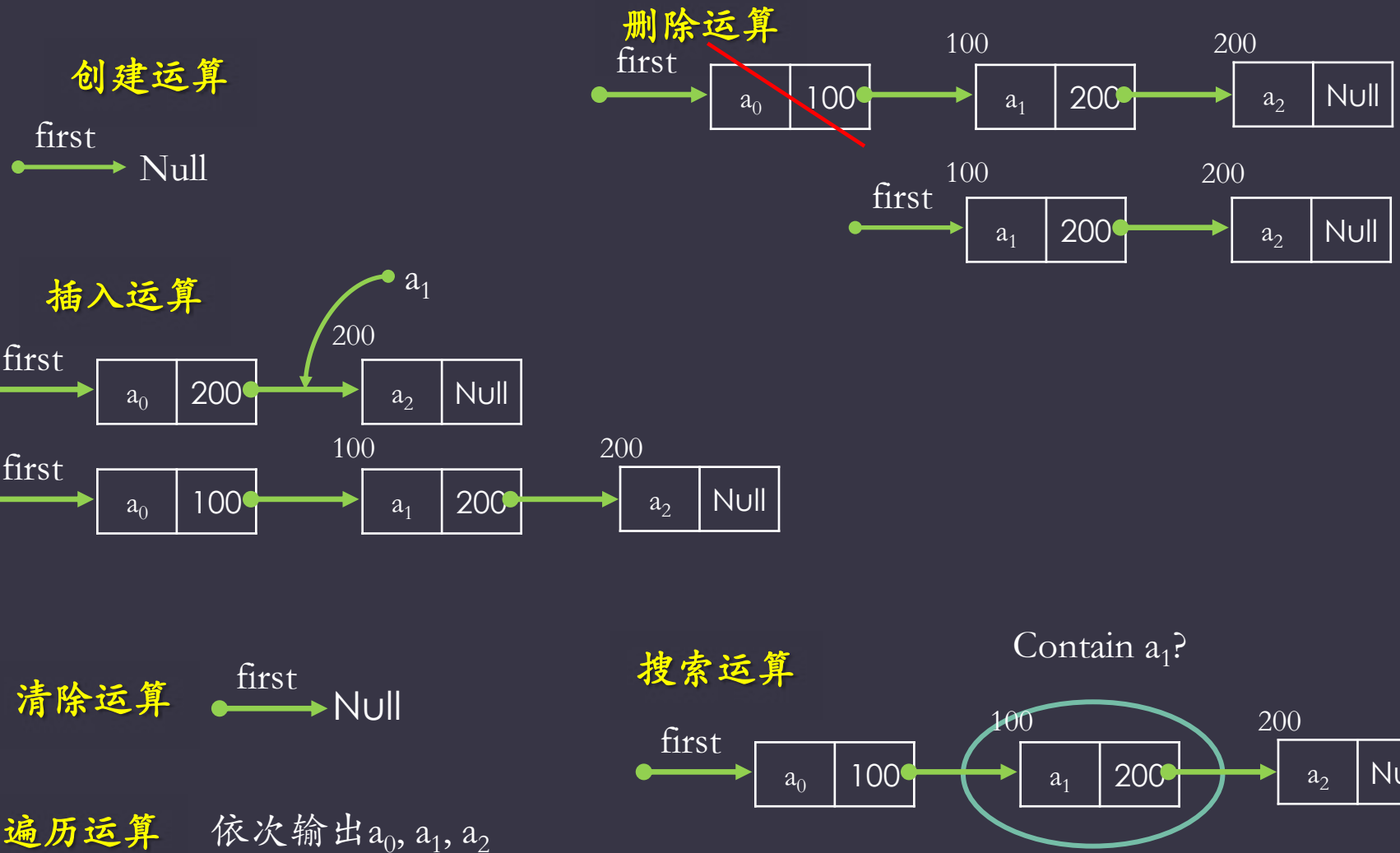
更新运算

商品	材质	价格
组合沙发	牛皮	7000
餐桌椅	榉木	5000

删除运算

- 搜索运算
- 访问运算
- 遍历运算

数据结构的运算


创建运算
first
Null
删除运算
first
100
200
a_0
a_1
a_2
Null
插入运算
a_1
200
清除运算
first
Null
遍历运算
依次输出a_0, a_1, a_2
搜索运算
Contain a_1?

数据结构的运算

- 静态数据结构
  - 系统分配固定大小存储空间，在程序运行过程中存储空间位置和容量都不再改变
  - 数组、简单类型（int, float, char……）
- 动态数据结构
  - 不确定总的存储量，仅为每个数据元素定义初始大小空间，随着数据元素的增加，不断分配新的存储空间，当数据元素减少，则回收存储空间
  - 链表

目录

- 算法与数据结构
- 数据结构的定义
- **数据抽象和抽象数据类型**
- 描述数据结构和算法
- 算法分析的基本方法

从过程抽象到数据抽象

- 过程抽象
  - 通过函数提供信息的隐蔽和重用性，无需了解函数所包含的运算步骤，仅需了解其实现功能，在需要该功能的地方调用该函数即可
- 数据抽象
  - 定义数据类型，将数据和其上的运算组合成一个整体（封装），实现更好的信息隐蔽和重用

数据类型

- C语言的数据类型
  - 基本类型：字符、整型 ……
  - 构造类型：数组、结构和联合
  - 指针类型：指针

数据类型：定义了一个值的集合以及作用于该值集的操作的集合

整型 int

值的集合：$\{x \mid x \in Z \cup -32768 \leq x \leq 32767\}$

操作的集合：{+, -, *, /, %, <, =, >, <=, >=, ==, !=}

抽象数据类型

- 抽象数据类型（Abstract Data Type, ADT）是一个数据类型，其主要特征是该类型的对象及其操作的规范，与该类型对象的表示和操作的实现分离，实行封装和信息隐蔽，即使用和实现分离

> 使用和实现分离：使用者通过规范使用该类型的数据，而不必考虑其实现细节；改变实现将不影响使用

数据结构与抽象数据类型

- 抽象层
  - 逻辑结构和运算定义组成了数据结构的规范---“做什么”
- 实现层
  - 数据存储方式与运算方式构成了数据结构的实现---“怎么做”

目录

- 算法与数据结构
- 数据结构的定义
- 数据抽象和抽象数据类型
- **描述数据结构和算法**
- 算法分析的基本方法

描述数据结构与算法

- 数据结构的规范
  - 如何定义数据的逻辑结构
  - 如何描述数据的运算方法
- 数据结构的实现
  - 如何实现数据的存储结构
  - 如何实现数据的运算方法

数据结构的描述方法

数据结构被看成是一个抽象数据类型(ADT)，用格式化的自然语言来描述。

数据结构的ADT描述的是ADT的接口，它包括；

ADT名称

对数据的逻辑结构关系的简单陈述

该ADT上定义的一组运算的规范

ADT 1－1 Complex{

数据：

由一对实数(x，y)构成，x为实部，y为虚部。

运算：设两个复数分别为a=(a1，a2)和b=(b1，b2)。

Complex Comp(float x，float y)

功能: 构造函数，函数返回复数(x，y)；

Complex Add(Complex a，Complex b)

功能: 返回复数(a1+b1，a2+b2)；

Complex Sub(Complex a，Complex b)

功能: 返回复数(a1-b1，a2-b2)；

….….}

使用C语言的数据类型(结构、数组和指针)描述数据的存储表示方法，并使用C语言语句描述实现运算的算法。可以认为，这也是一个抽象数据类型的C语言实现。

假定C语言的数组和结构都是顺序存储的：

数组元素具有相同的数据类型，按照下标的次序存储在连续的存储区中；

结构可以由不同类型的成分(域)组成，按照域表的次序顺序存放

【程序1－1】 Complex的实现。

```
#include <stdio.h>
#include <stdlib.h>
typedef struct complex
{
        float x, y;
}Complex;
Complex Comp(float x，float y)
{
    Complex c;
    c.x=x; c.y=y;
    return c;
}
```

```
Complex Add (Complex a，Complex b)
 {
      Complex c;
      c.x=a.x+b.x;
      c.y=a.y+b.y;
      return c;
 }
      ……
```

目录

- 算法与数据结构
- 数据结构的定义
- 数据抽象和抽象数据类型
- 描述数据结构和算法
- 算法分析的基本方法

算法及其性能标准

- 一个算法(algorithm)是对特定问题的求解步骤的一种描述，是指令的有限序列
  - 输入：算法有零个或多个输入
  - 输出：算法至少产生一个输出
  - 确定性：算法的每一条指令都有确切的定义，没有二义性。
  - 能行性/可行性：可以通过已经实现的基本运算执行有限次来实现
  - 有穷性：算法必须总能在执行有限步之后终止

下列算法有什么问题？

```
void getSum(int num)
{
    int i, sum = 0;
    for(i=0; i<=num; i++)
    { sum += i;}
    return;
}
```

没有输出的算法是无意义的

下列算法有什么问题？

```
void example()
{
    while(true)
    {printf(“%s”, “Hi”);}
    return;
}
```

违反算法有穷性

下列算法有什么问题？

```
float example(float y)
{
    float x=0;
    return y/x;
}
```

违反算法可行性

下列算法有什么问题？

```
float avg(int *a, int num)
{ int i;
   double sum = 0;
   for(i=0;i<=num; i++)
      sum += *(++a);
      return sum/num;
}
int main(int arg, char* argv[])
{ int score[4] = {1,2,3,4};
   printf( “%f” , avg(score,4));
}
```

违反算法确定性

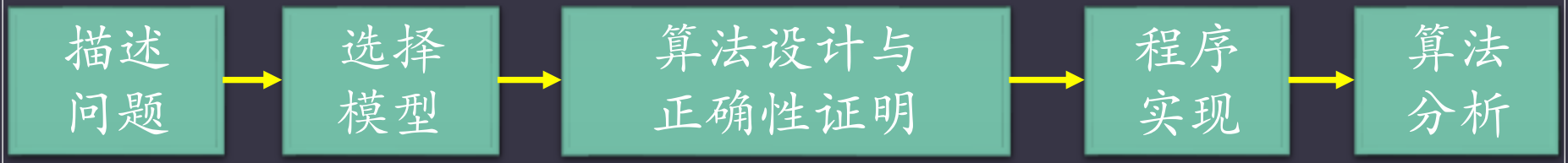
算法的描述

- 自然语言：容易理解，但是冗长、二义性、无法直接转换成可执行程序
- 流程图：流程直观，但仅适用于描述简单算法
- 伪代码（Pseudocode）：介于自然语言和程序设计语言之间的方法，采用程序设计基本语法，操作指令可以结合自然语言来设计
- 程序设计语言：可执行程序

算法性能标准

- 正确性：算法的执行结果应当满足功能需求，无语法错误，无逻辑错误
- 简明性：思路清晰、层次分明、易读易懂，有利于调试维护
- 健壮性：当输入不合法数据时，应能做适当处理，不至于引起严重后果
- 效率：有效使用存储空间和有高的时间效率
- 最优性：解决同一个问题可能有多种算法，应进行比较，选择最佳算法
- 可使用性：用户友好性

算法分析



- 算法分析（Algorithm Analysis）：对算法所需要的计算资源—**时间**和**空间**进行估算
  - 算法的时间复杂度（Time Complexity）是程序从运行开始到结束所需时间
  - 算法的空间复杂度（Space Complexity）是程序从运行开始到结束所需存储空间

算法的时间复杂度

- 程序步：在语法或语义上有意义的程序段，该程序段的执行时间与问题规模无关

```
float sum(float list[],const int n)
{   float tempsum=0.0;          ←—— 1
 for(int i=0;i<n;i++)           ←—— n+1
   tempsum+=list[i];            ←—— n
 return tempsum;     }          ←—— 1
```

T(n)=2n+3

算法的时间复杂度

- 程序步：在语法或语义上有意义的程序段，该程序段的执行时间与问题规模无关

```
float sum(float list[],const int n)
{   float tempsum=0.0;          1
    int i=0;                    1
    while(i<n)                  n+1
     { tempsum+=list[i];        n
       i++;}                    n
    return tempsum;     }       1
```

$T(n)=3n+4$

算法的渐近时间复杂度

- 忽略算法时间复杂度的低次幂和最高次幂的系数，使用大O记号表示

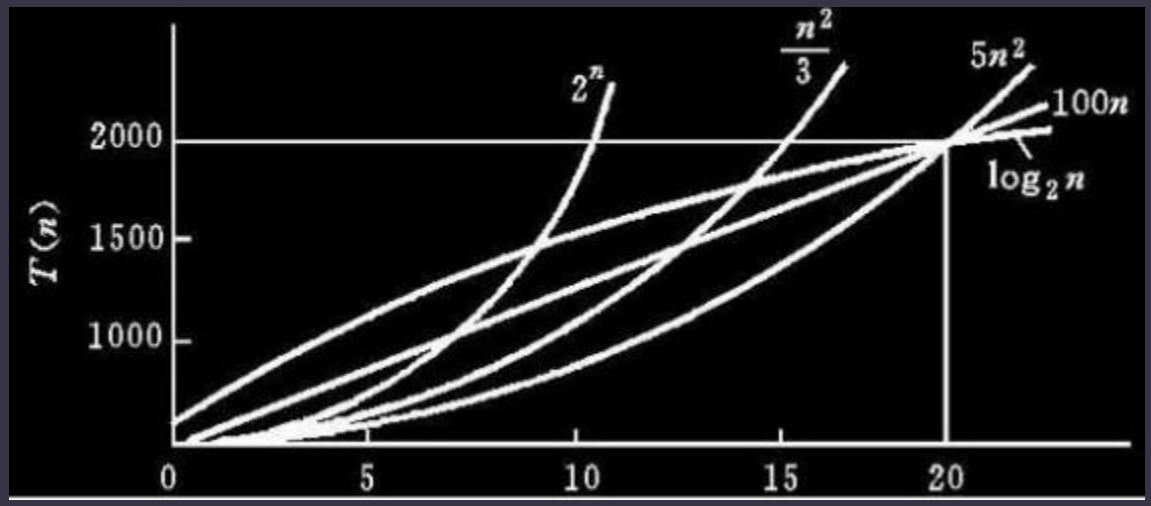
$T(n)=3.6n^3+2.5n^2+2$ ⇨ $T(n)=O(n^3)$

算法的渐近时间复杂度

- 忽略算法时间复杂度的低次幂和最高次幂的系数，使用大O记号表示

$$O(1)< O(\log_2 n) < O(n)< O(n\log_2 n)< O(n^2)< O(n^3)<O(2^n)$$

一个没有循环的算法，其渐近时间复杂度为常数计算时间O(1)



算法的渐近时间复杂度

- 问题规模：输入量的多少
- 关键操作/基本语句
  - 最深层循环内的语句

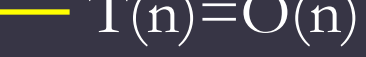
```
for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
      x+=i*j;
```

问题规模：n

基本语句：x+=i*j

算法的渐近时间复杂度

```
float sum(float list[],const int n)
{   float tempsum=0.0;
 for(int i=0;i<n;i++)
   tempsum+=list[i];
 return tempsum;     }
```



算法的渐近时间复杂度

```
void Mult(int a[n][n], b[n][n], c[n][n], int n)
{ for (int i=0;i<n;i++)
    for(int j=0;j<n;j++)
    { c[i][j]=0;
      for (int k=0;k<n;k++)
          c[i][j]+=a[i][k]*b[k][j]; }
}
```

$T(n)=O(n^3)$

最好、最坏和平均时间复杂度

- 问题实例的规模相同，算法所需时间开销与输入的数据相关
  - 最好情况：仅在发生概率较大情况下分析
  - 最坏情况：实时系统
  - 平均情况：已知输入数据的分布概率，通常假设平均分布

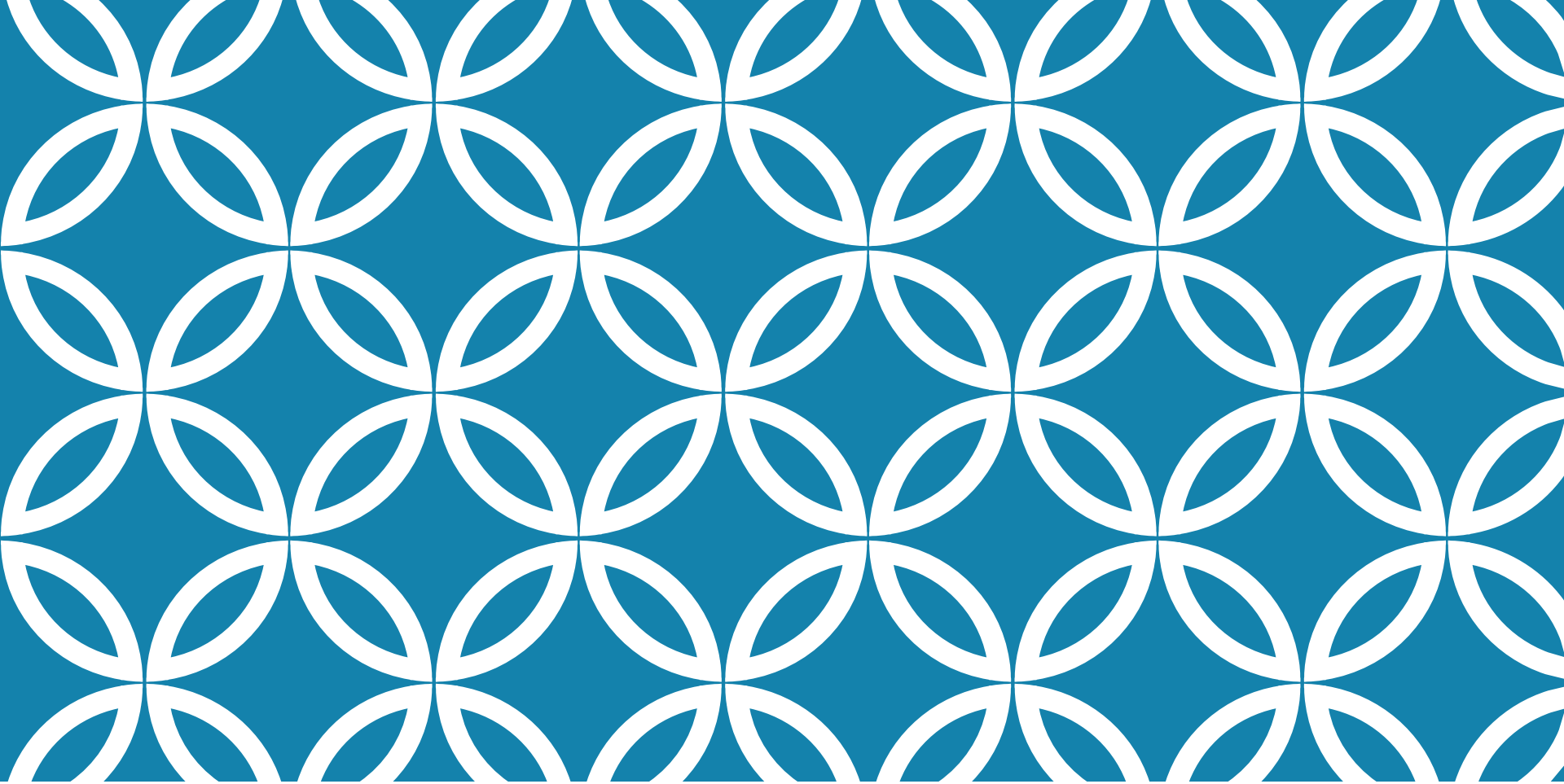
Search(x)

最好情况：1
最坏情况：n
平均情况：≈n/2

算法的空间复杂度

- 固定部分：
  - 与问题规模无关，主要包括程序代码、常量、简单变量、定长成分的结构变量所占的空间
- 可变部分—空间复杂度 S(n)
  - 这部分空间大小与算法在某次执行中处理的特定数据的大小和规模有关
  - 一般按最坏情况分析

```
for(i=0; i<n; i++)
  for(j=i+1; j<n; j++)
 {
      if(a[i]>a[j])
      {
          x = a[i];
          a[i] = a[j];
          a[j] = x;
      }
 }
```



数组和链表

目录

结构与结构变量

定义结构

```
struct student
{
    char name[20];
    char gender;
    int age;
}
```

定义结构变量

```
struct student studA;
struct student students[20];
```

定义结构

```
typedef struct student
{
    char name[20];
    char gender;
    int age;
} Student
```

结构名

结构类型名

定义结构变量

```
Student studA;
Student *stud;
```

结构与结构变量

定义结构

```
struct student
{
    char name[20];
    char gender;
    int age;
}
```

```
struct student
{
    char name[20];
    char gender;
    int age;
}studA, students[20];
```

定义结构变量

```
struct student studA;
struct student students[20];
```

结构自引用

```
struct node
{
    char key[20];
    char value[20];
    struct node next;
}
```

```
struct node
{
    char key[20];
    char value[20];
    struct node *next;
}
```

结构的使用

定义结构

```
struct student
{
    char name[20];
    char gender;
    int age;
}
```

定义结构变量

```
struct student studA, studB;
Struct student *stud;
```

- “.” 直接访问结构变量

```
studA.age = 18;
printf( "%c\n", studA.name[0]);
strcpy(studA.name, "LeLe") ;
```

- “->” 间接访问

```
stud = &studB;
stud->age = 22;
```

结构的使用

定义结构

```
struct student
{
    char name[20];
    char gender;
    int age;
}
```

定义结构变量

```
struct student studA, studB;
Struct student *stud;
```

- 结构变量整体初始化

  studA={“Mike”，‘M’,2};

- 结构变量整体赋值

  stud = &studB;

  studA = studB;

目录

数组 VS 结构

相同点：

- 数组类型变量一旦定义，则一次性申请连续存储空间用于存放指定数量的数据元素

不同点：

- 数组的元素具有相同的数据类型，而结构的成员(域)可以是不同类型

数组元素用下标(index)标识，而结构成员由域名(field name)引用。

一维数组

```
int one[5];
```

定义了5个整数组成的一个数组，下标从0到4

数组可以在定义时集体赋值

```
int one[5]={0, 1, 2, 3, 4};
```

可以依次对每个数据元素赋值

```
for ( i=0; i<5; i++) one[i]=i;
```

一维数组的存储结构

数组类型采用顺序方式存储：数组中的元素按一定顺序存放在一个连续的存储空间

计算机中的存储空间是一维的，一维数组元素可直接映射到存储空间中

设给长度为n的一维数组a所分配的存储块的起始地址是Loc(a[0])，若已知a的每个元素占k个存储单元，则下标为i的数组元素a[i]的存放地址Loc(a[i])是

$$Loc(a[i]) = Loc(a[0]) + i*k \qquad 0 \leqslant i < n$$

二维数组

```
int one[2][3];
```

定义了包含2个整型一维数组的数组，下标从0到1

每个一维数组又包含了3个整型，下标从0到2

数组可以在定义时集体赋值

```
int one[2][3]={{0, 1, 2}, {3, 4, 6}};
```

可以依次对每个数据元素赋值

```
for ( i=0; i<2; i++)
    for ( j=0; j<3; j++)  one[i][j]=i*j;
```

二维数组的存储结构

数组类型采用顺序方式存储：数组中的元素按一定顺序存放在一个连续的存储空间

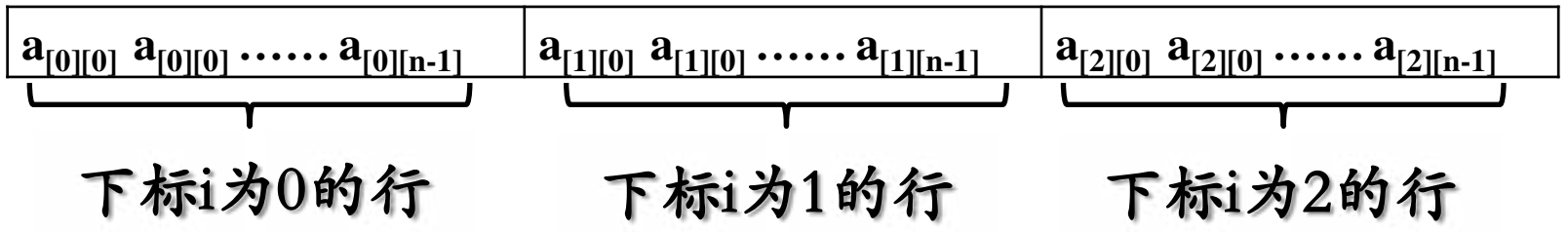
计算机中的存储空间是一维的，二维数组元素需要按照一定规则映射到一维存储空间中

- 行优先存储
- 列优先存储

数组的顺序存储

二维数组a[m][n]需按照行优先映射到一维的存储空间

$$\begin{bmatrix} a_{[0][0]} & a_{[0][1]} & \cdots & \cdots & \cdots & a_{[0][n-1]} \\ a_{[1][0]} & a_{[1][1]} & \cdots & \cdots & \cdots & a_{[1][n-1]} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{[i][j]} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{[n-1][0]} & a_{[n-1][1]} & \cdots & \cdots & \cdots & a_{[n-1][n-1]} \end{bmatrix}$$



数组的顺序存储

行优先顺序的地址计算

若对于二维数组a[m][n],已知每个数组元素占k个存储单元，第一个数组元素a[0][0]的存储地址是loc(a[0][0]),则数组元素a[i][j]的存储地址loc(a[i][j])为

loc(a[i][j])=loc(a[0][0])+(i*n+j)*k　(0≤i<m;0≤j<n)

$$\begin{bmatrix} a_{[0][0]} & a_{[0][1]} & \cdots & \cdots & \cdots & a_{[0][n-1]} \\ a_{[1][0]} & a_{[1][1]} & \cdots & \cdots & \cdots & a_{[1][n-1]} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{[i][j]} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{[n-1][0]} & a_{[n-1][1]} & \cdots & \cdots & \cdots & a_{[n-1][n-1]} \end{bmatrix}$$

数组的顺序存储

列优先顺序的地址计算

$$loc(a[i][j])=loc(a[0][0])+(j*m+i)*k \quad (0\le i<m;0\le j<n)$$

$$\begin{bmatrix} a_{[0][0]} & a_{[0][1]} & \cdots & \cdots & \cdots & a_{[0][n-1]} \\ a_{[1][0]} & a_{[1][1]} & \cdots & \cdots & \cdots & a_{[1][n-1]} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{[i][j]} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{[n-1][0]} & a_{[n-1][1]} & \cdots & \cdots & \cdots & a_{[n-1][n-1]} \end{bmatrix}$$

例1：10×10的整型数组A，其每个数组元素占2个字节，已知A[0][0]在内存中的地址是100，按行优先，A[4][6]的地址是____。

A．228　　B．192　　C．124　　D．138

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	[0][8]	[0][9]
[1][0]						[1][6]			
[2][0]						[2][6]			
[3][0]						[3][0]			
[4][0]						[4][6]			
[5][0]						[5][0]			
[6][0]						[6][0]			
[7][0]						[7][0]			
[8][0]						[8][0]			
[9][0]						[9][0]			

loc(a[0][0])+(i×n+j)×k＝100+（4*10+6）*2＝192

二维数组的顺序存储

Loc(a[0][0])被称为基地址，它是存储数组的存储空间的起始地址。

数组一旦规定了它的维数和各维的长度，便可为它分配存储空间

只要给出数组元素下标，就可根据相应的地址计算公式求得数组元素的存储位置来存取元素

存取数组中任何一个元素所需的时间是相同的，具有这一存取特点的存储结构为随机存取的存储结构。

多维数组的顺序存储

多维数组$a[m_1][m_2]\cdots[m_n]$，数组元素$a[i_1][i_2]\cdots[i_n]$的存储地址为

$$loc(a[i_1][i_2]\cdots[i_n])=loc(a[0]\cdots[0])+(i_1*m_2*m_3*\cdots*m_n+i_2*m_3*m_4*\cdots*m_n+\cdots+i_{n-1}*m_n+i_n)*k$$

$(0\le i_j<m_j,\ 1\le j\le n)$

i_1	m_2	m_3	m_4	…	m_n
1	i_2	m_3	m_4	…	m_n
1	1	i_3	m_4	…	m_n
1	1	1	i_4	…	m_n
1	1	1	1	⋱	m_n
1	1	1	1	…	i_n

- 数组元素的存储位置是其下标的线性函数。通过计算地址便可实现对数组元素的随机存取。
- C语言中数组是不允许整体赋值的。
- C语言中对数组下标超出正常范围的访问并不限制

  例如对长度为5的一维数组a，不合法的访问a[-3]，a[7]
- 顺序存储一般借助数组来实现
- 数组是静态数据结构，其存储空间的大小需具体确定，并预先分配，一旦分配则难以扩充
- 数组名本身存储了指针值，其保存数组的首地址

C语言提供的数组并非一个完备的数据结构

（1）不能实现数组的整体赋值；

（2）不能将数组作为函数值返回；

（3）对数组元素下标不提供边界检查；

（4）将数组名作为变量进行传递时，传递的实际上是数组的基地址。

目录

指针

int i, *pi;

定义一个整型变量i 和一个指向整型变量的指针变量pi

任意类型都可定义指向该类型的指针类型

指针类型变量的值是一个存储地址

与指针类型相关的运算符:

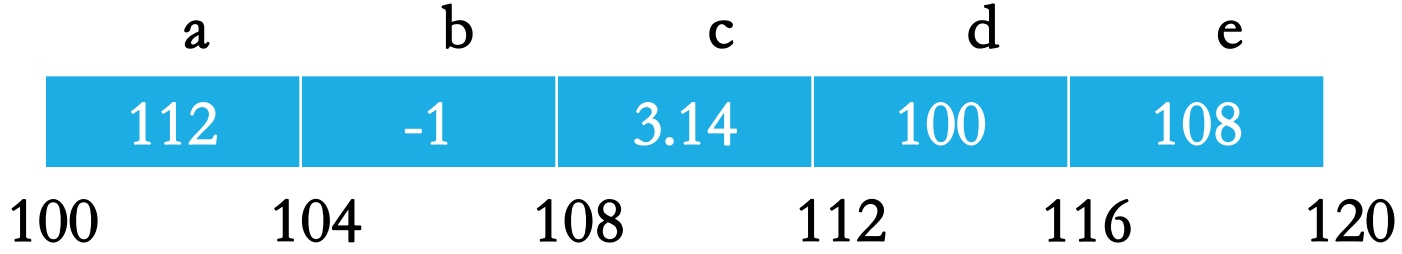
- 取地址运算符 &
- 间接引用运算符 *

间接引用运算符用来对指针变量所指示的地址空间中存储的数据元素进行操作。

指针

```
int a = 112, b=-1;
float c = 3.14;
int *d = &a;
float *e = &c;
```

指针



int a = 112, b=-1;

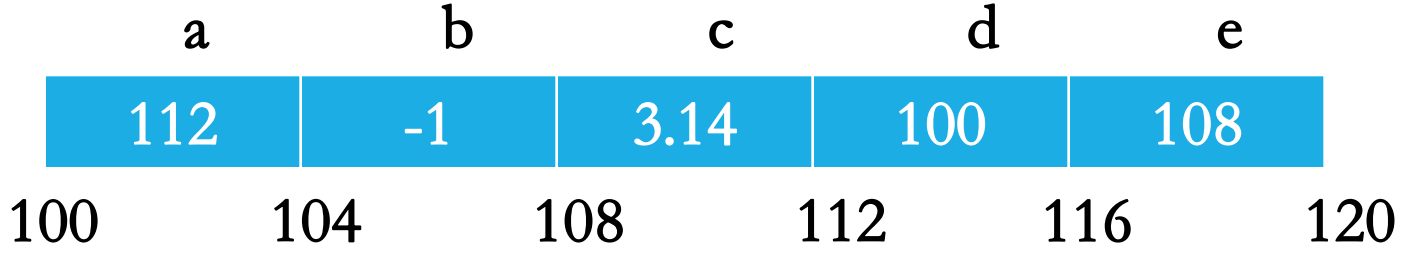
int *d = &a;

float c = 3.14;

float *e = &c;

表达式	右值	类型	表达式	右值	类型	表达式	右值	类型
a			&a			*a		
b			&b			*b		
c			&c			*c		
d			&d			*d		
e			&e			*e		

指针



int a = 112, b=-1;

float c = 3.14;

int *d = &a;

float *e = &c;

表达式	右值	类型	表达式	右值	类型	表达式	右值	类型
a	112	int	&a	100	int *	*a	-	-
b	-1	int	&b	104	int *	*b	-	-
c	3.14	float	&c	108	float *	*c	-	-
d	100	int *	&d	112	int **	*d	112	int
e	108	float *	&e	116	float **	*e	3.14	float

下面哪些是非法的指针？

```
int *a = 12;
```

```
int *a;
a = 12;
```

```
int *a;
*a = 12;
```

```
int *a, b=12;
a = &b;
```

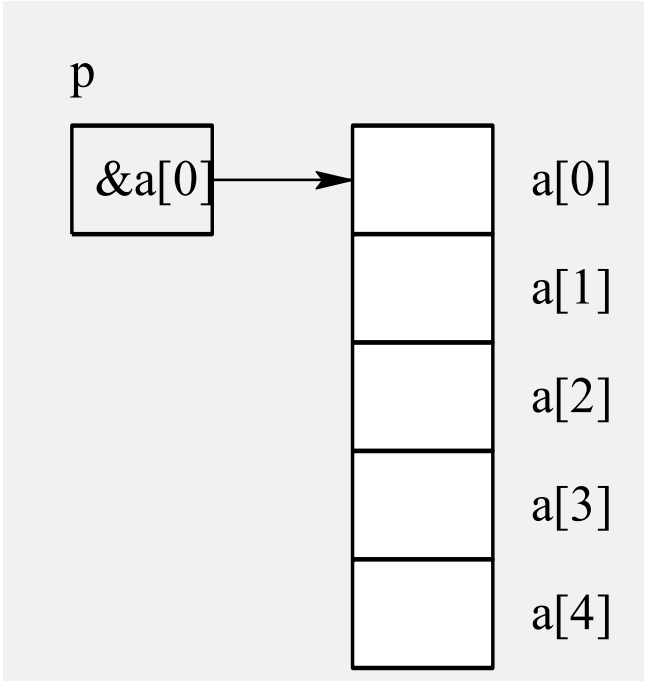
```
int *a, b;
a = &b;
*a = 12;
```

指针和数组

数组名本身存储了指针值，其保存数组的首地址

指针等于数组吗?

```
int a[5];
int *p;
p = a;
p = &a[0];
```



借助指针可以实现动态数组

```
int a[n]; /*错误*/
int *b = (int *) malloc(n*sizeof(int));
```

指向结构的指针

结构的自引用

```
struct node{
        int Data;
        struct node* Link;
};
```

当一个结构中有一个或多个成员是指向它自身的指针时，称这种结构为**自引用结构**。

动态存储分配

C语言中的变量可分为两类：静态变量和动态变量

- 静态变量是存储空间可预先估算且固定不变的变量
- 动态变量：只在程序运行时才创建，存储空间大小可由变量控制，动态变量只能通过指针访问

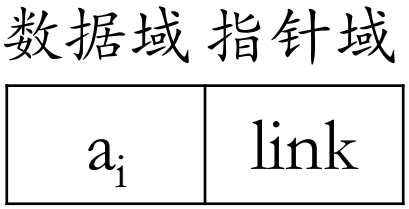
动态存储分配：在运行时根据程序要求为变量分配存储空间的方法。

使用C语言的标准函数malloc和free动态地创建和撤销一个动态变量

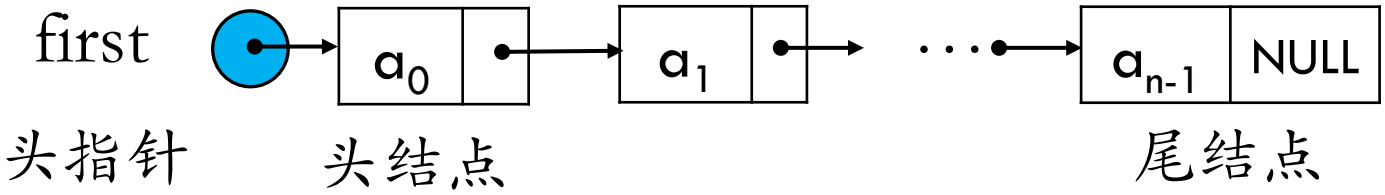
```
Node* p=(Node*)malloc(sizeof(Node));
if (!p)
{
    fprintf(stderr, "The memeny is full\n");
    exit(1);
}
free(p);
```

单链表

单链表结点结构



单链表结构



空单链表
first == NULL

单链表

注意：

- 单链表头指针first不能少
- 最后一个结点的指针域为∧
- 逻辑上相邻的元素在物理上不一定相邻
- 不能出现“断链”现象

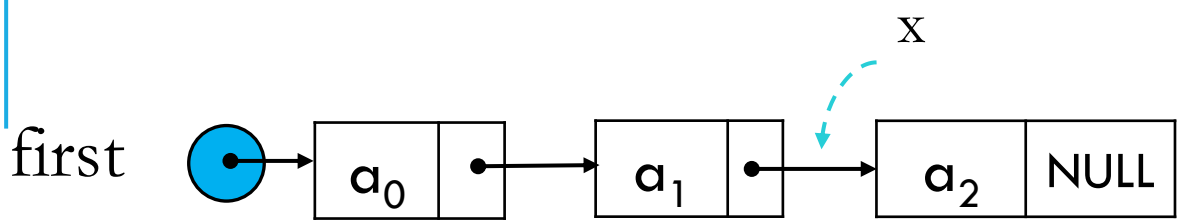
单链表的结点结构实现

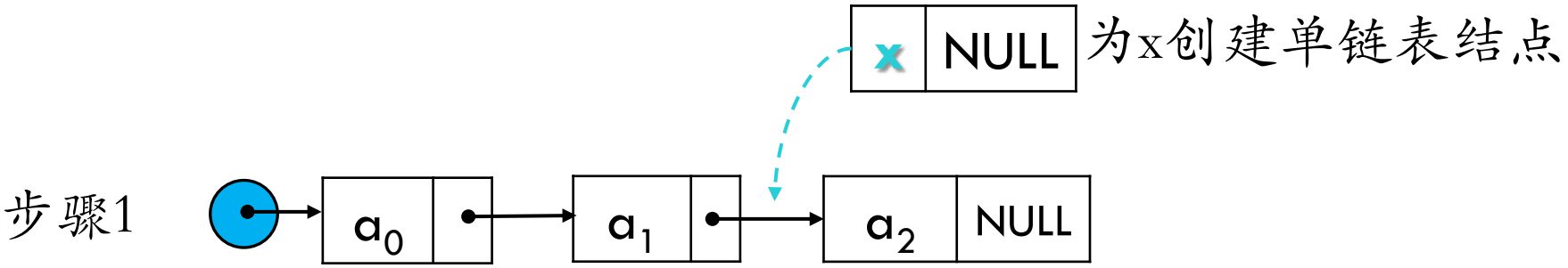
```
typedef struct node{
        ElemType element;
        struct node* link;
    } Node;
```

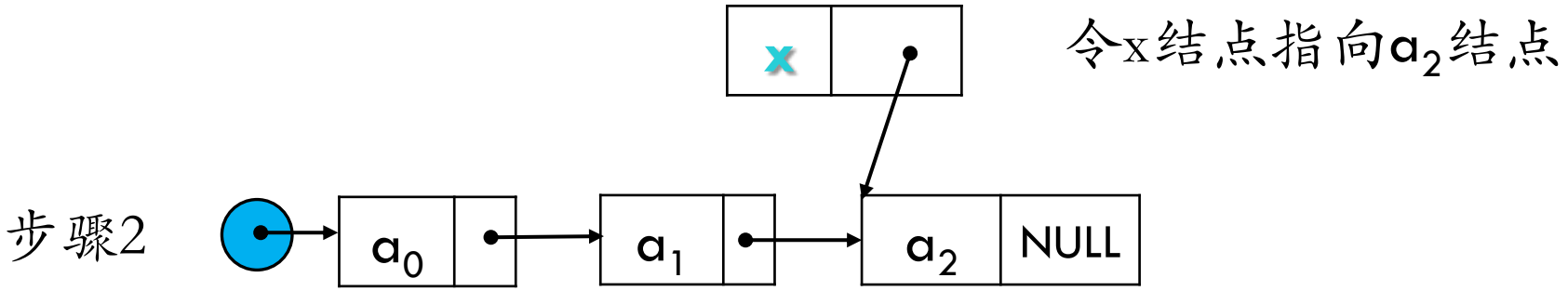
```
typedef  int ElemType ;
//类型别名
```

包括两个域：元素域element和指针域link。

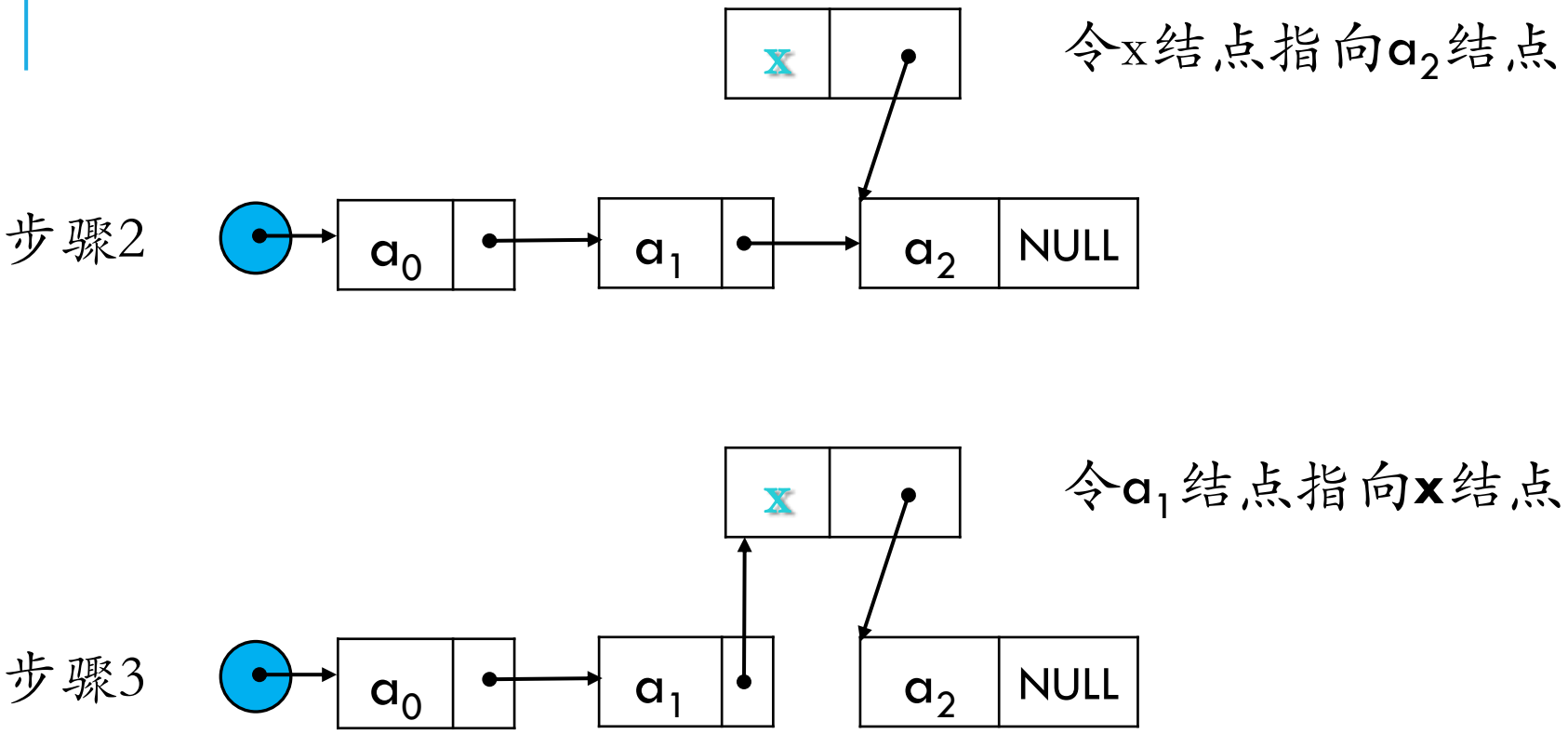
单链表的插入


x
first
a_0
a_1
a_2
NULL


x
NULL
为x创建单链表结点
步骤1
a_0
a_1
a_2
NULL

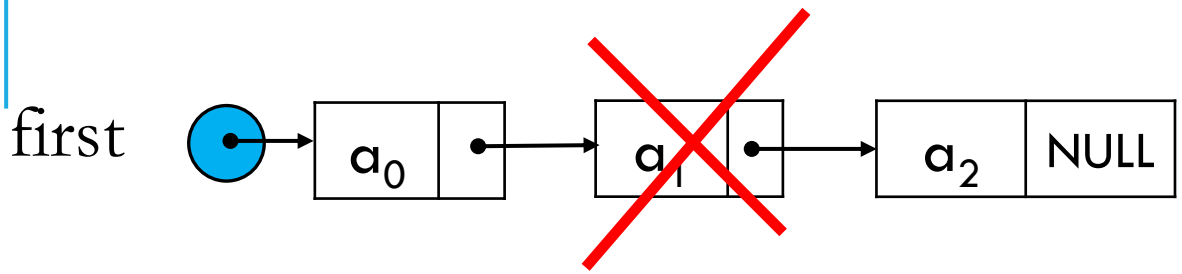

x
令x结点指向a_2结点
步骤2
a_0
a_1
a_2
NULL

单链表的插入



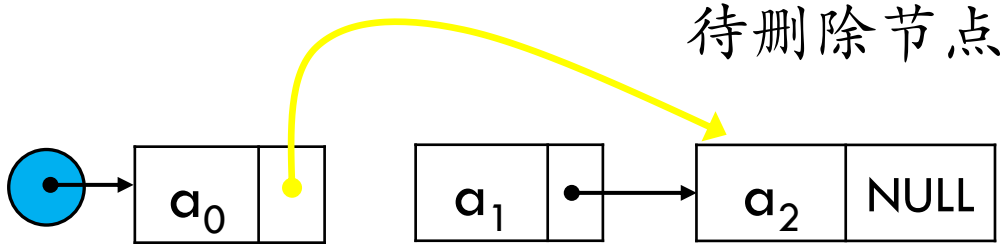
能否对调上述语句执行顺序?

单链表的删除



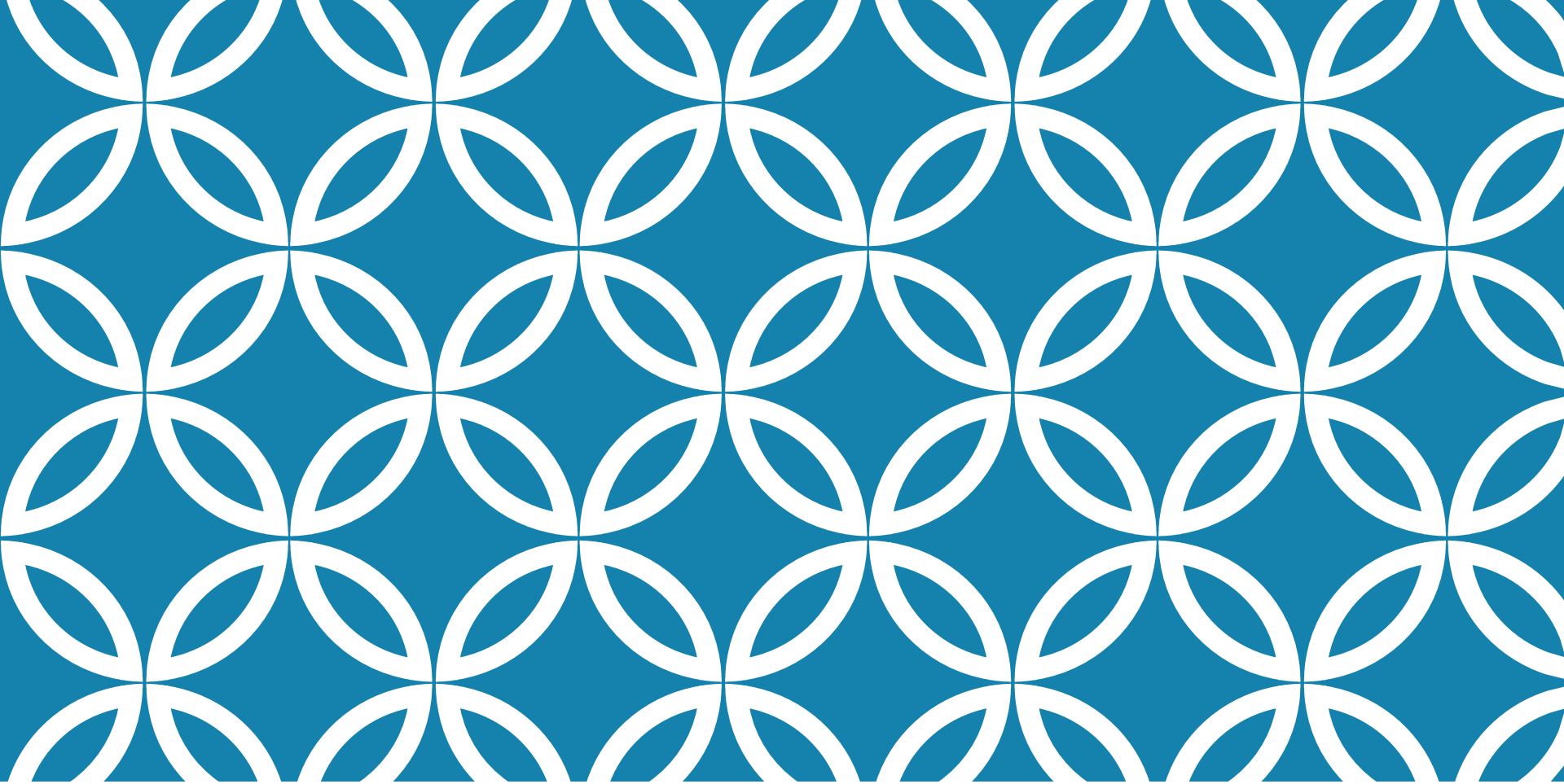
待删除节点直接前驱节点指向
待删除节点的直接后继节点

步骤1



删除节点

步骤2



线性表

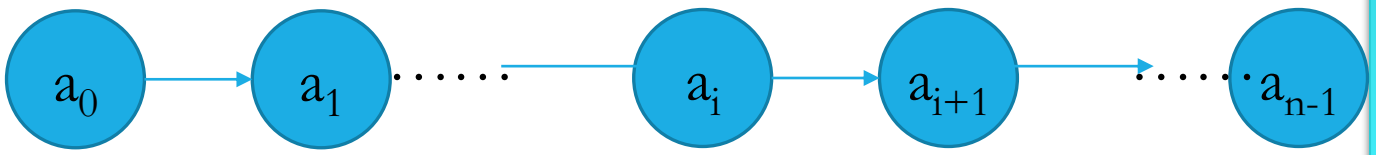
目录

线性表的定义

n个元素a_0，a_1，…，a_{n-1}的有序集合，记为$(a_0, a_1, \ldots, a_{n-1})$

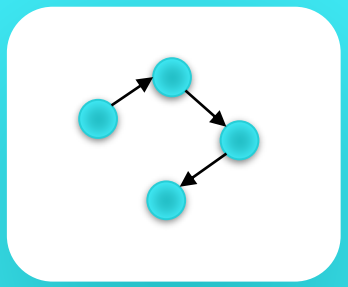
- n≥0是线性表中元素的个数，是线性表的长度，n=0时为空表
- 线性表的表长可以改变。

商品	材质	价格
组合沙发	牛皮	7000
餐桌椅	榉木	5000
衣橱	橡木	3000



直接前驱 → 直接后继

线性结构



一对一关系
1前驱 or 1后继

ADT 2.1 List {

数据：

零个或多个元素的线性序列($a_0, a_1, \cdots, a_{n-1}$)，其最大允许长度为MaxLength。

运算：

Init(L)

构造运算: 构造一个空线性表。

IsEmpty(L)

判空运算：若线性表为空，则返回TRUE，否则返回FALSE。

IsFull(L)

判满运算：若线性表已满，则返回TRUE，否则返回FALSE。

int Size(L)

长度运算: 返回线性表的长度。

Insert(L, i，x)

插入运算：若线性表未满且i位置合法，则将元素x插在位置i，并且函数返回TRUE；否则函数返回FALSE。

Remove(L, i, x)

删除运算：若线性表非空且i位置合法，则位置i处的元素通过x返回，从原表中移去该元素，并且函数返回TRUE；否则函数返回FALSE。

Retrieve(L, i, x)

获取运算：若线性表非空且i合法，则位置i处的元素通过参数x返回，并且函数返回TRUE；否则函数返回FALSE。

Replace(L, i, x)

替代运算： 若线性表非空且i合法，则位置i处的元素值被x替代，并且函数返回TRUE；否则函数返回FALSE。

Clear(L)

清除运算: 移去所有元素，线性表成为空表。

}

目录

线性表的顺序存储表示

是用一组地址连续的存储单元依次存储线性表中元素。

顺序表示的线性表程为顺序表。

线性表的顺序实现可以用下面的C语言结构定义:

```
typedef struct list{
    int n, maxLength;
    ElemType *element;
} seqList;
```

2.2.2 顺序表基本运算的实现

1. 初始化

```
#define ERROR 0
#define OK 1
#define Overflow 2 // Overflow 表示上溢
#define Underflow 3 // Underflow 表示下溢
#define NotPresent 4 // NotPresent 表示元素不存在
#define Duplicate 5 // Duplicate 表示有重复元素
typedef int Status;
Status Init(SeqList *L,int mSize)
{
    L->maxLength= mSize;
    L->n=0;
    L->element=(ElemType*)malloc(sizeof(ElemType)*mSize);
    //动态生成一维数组空间
    if (!L->element)
    return ERROR;
    return OK;
}
```

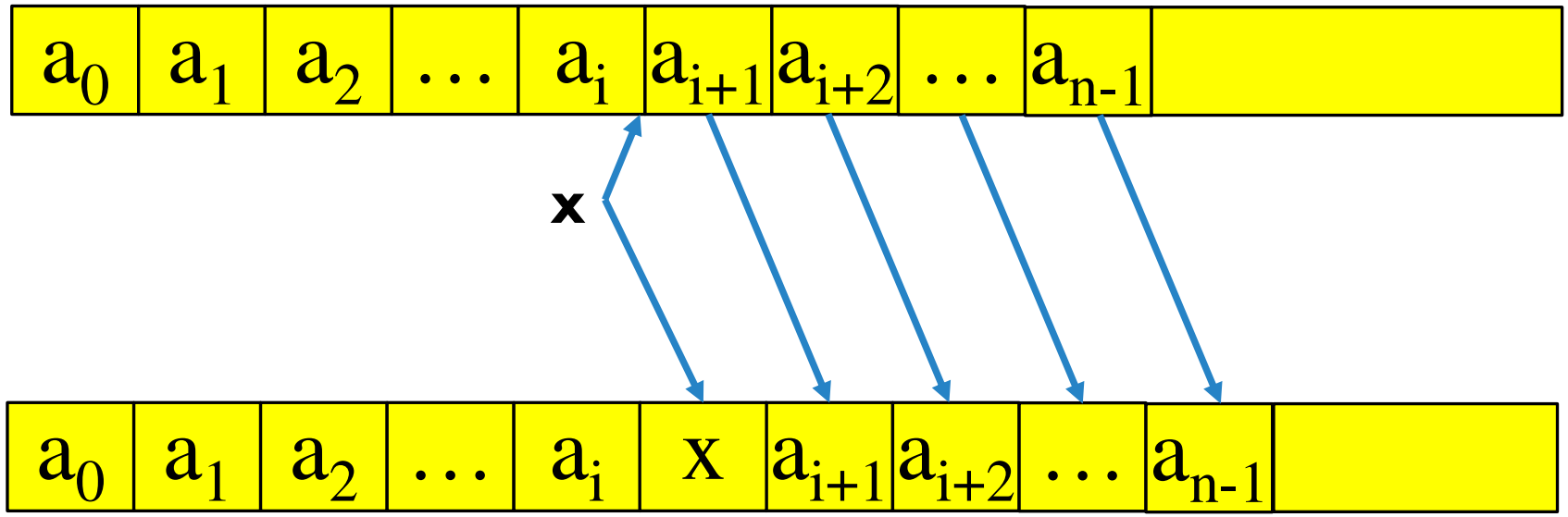
2．查找

顺序表的查找运算是查找表中元素a_i的值。

```
Status Find(SeqList L, int i,ElemType *x)
{
   if (i<0 || i>L.n-1)
   return ERROR;     //判断元素下标i 是否越界
   *x=L.element[i]; //取出element[i]值通过参数x 返回
   return OK;
}
```

3. 插入操作

顺序表的插入运算是在顺序表L 的元素a_i 之后插入新元素x。



```
Status Insert(SeqList *L ,int i ,ElemType x)
{
 int j;
 if (i<-1 | | i>L->n-1) //判断下标i 是否越界
 return ERROR;
 if(L->n== L->maxLength) //判断顺序表存储空间是否已满
 return ERROR;
 for (j= L->n-1;j>i;j--)
 L->element[j+1]= L->element[j]; //从后往前逐个后移元素
 L->element[i+1]=x; //将新元素放入下标为i+1 的位置
 L->n = L->n +1;
 return OK;
}
```

讨论当i=-1且n>0的情况

讨论当i=-1且n=0的情况

分析：

设顺序表长度为n，则在位置i（i=-1,0,…,n-1）后插入一个元素要移动 n-i-1个元素。

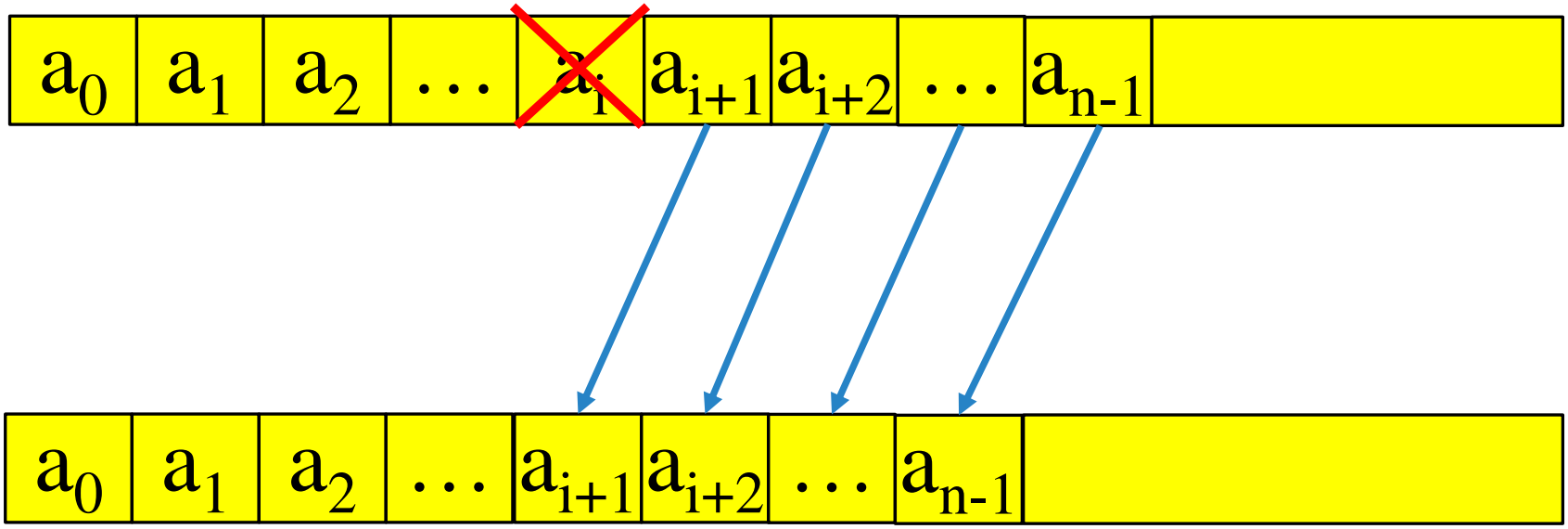
设P_i 是在位置i 之后插入一个新元素的概率，并设在任意位置处插入元素的概率是相等的， 即$P_i=1/(n+1)$.设E_i 是在长度为n 的顺序表中插入一个新元素时所需移动元素的平均次数，则：

$$E_i = \sum_{i=-1}^{n-1} \frac{1}{n+1}(n-i-1) = \frac{n}{2}$$

渐近时间复杂度：O(n)

4．删除

顺序表的删除运算的功能是将元素ai 删除。



删除操作算法：

```
Status Delete(SeqList *L ,int i)
{
  int j;
  if (i<0 | | i> L-> n-1)      //下标i 是否越界
  return ERROR;
  if (!L->n)                   //顺序表是否为空
  return ERROR;
  for ( j=i+1;j< L-> n;j++)
  L->element [j-1]= L->element [j]; //从前往后逐个前移元素
  L->n --; //表长减1
  return OK;
}
```

分析:

设顺序表长度为n,则删除元素a_i（i=0,…,n-1）,要移动n-i-1元素。。设Pi是在位置i删除一个元素的概率，

并设在任意位置处删除元素的概率是相等的，则有P_i=1/n设E_d是在长度为n的顺序表中删除一个元素时所需移动元素的平均次数，则

$$E_d = \sum_{i=0}^{n-1} \frac{1}{n}(n-i-1) = \frac{n-1}{2}$$

渐近时间复杂度：O(n)

```
Status Output(SeqList L)
{
  int i;
  if (!L.n) //判断顺序表是否为空
  return ERROR;
  for (i=0;i<= L.n -1;i++)
  printf("%d ",L.element [i]); //从前往后逐个输出元素
  return OK;
}
```

6．撤销

顺序表的撤销运算的主要功能是释放初始化运算中动态分配的数据元素存储空间，以防止内存泄漏。

```
void Destroy (SeqList *L)
{
  (*L).n=0;
  (*L). maxLength=0;
  free((*L).element);
}
```

7．主函数main

顺序表的撤销运算的主要功能是释放初始化运算中动态分配的数据元素存储空间，以防止内存泄漏。

```
#include<stdlib.h>
#include<stdio.h>
typedef int ElemType;
typedef struct
{
    int n;
    int maxLength;
    ElemType *element;
} SeqList;
```

```
void main()
{
   int i;
   SeqList list;
   Init(&list,10); //对线性表初始化
   for(i=0;i<9;i++)
   Insert(&list,i-1,i); //线性表中插入新元素
   Output(list);
   Delete(&list,0);
   Output(list);
   Destroy(&list);
}
```

线性表的顺序存储表示的优缺点：

（1）优点：

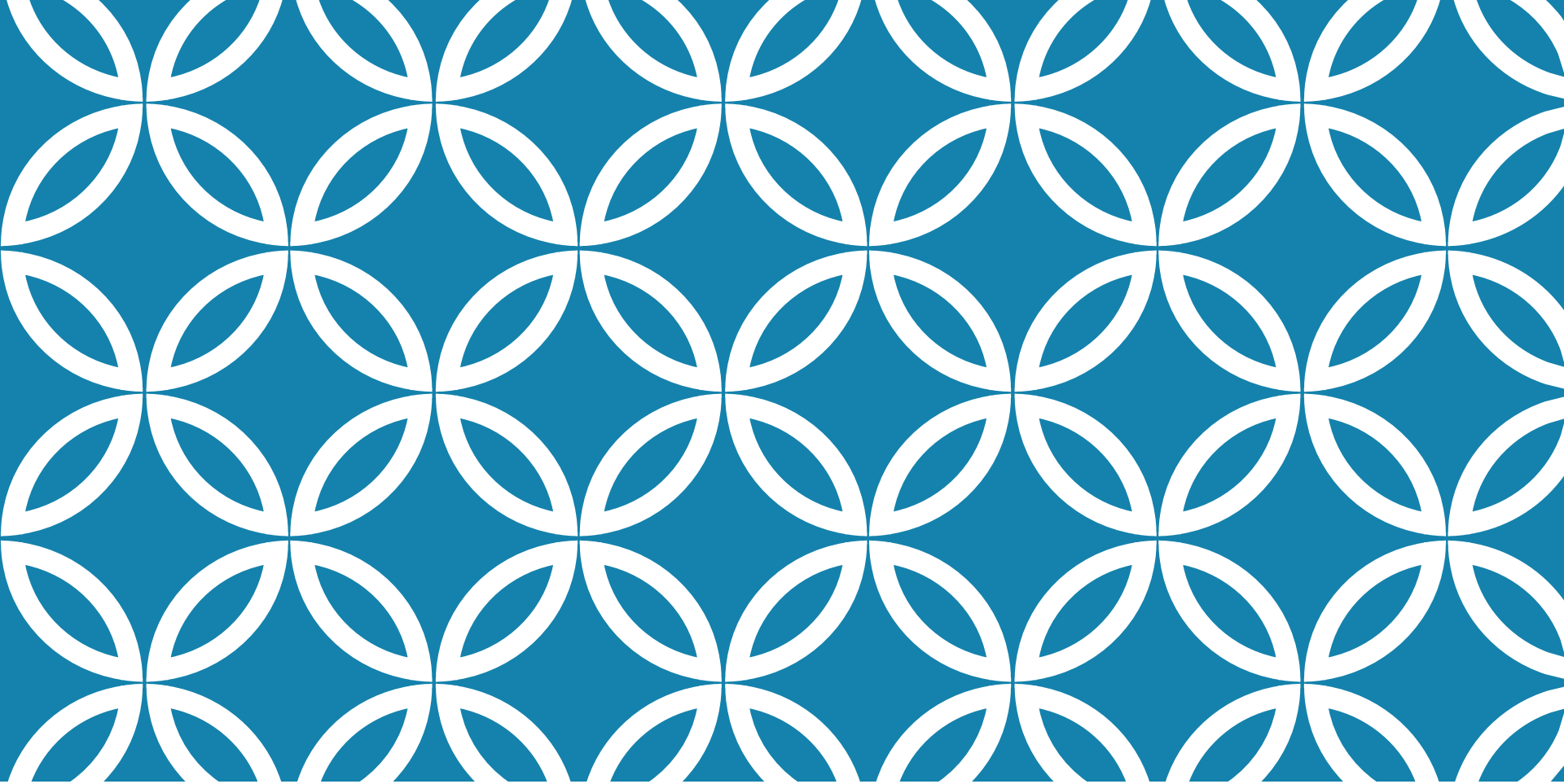
随机存取；

存储空间利用率高。

（2）缺点：

插入、删除效率低；

必须按事先估计的最大元素个数分配连续的存储空间，难以临时扩大。



线性表

目录

线性表的链接存储表示

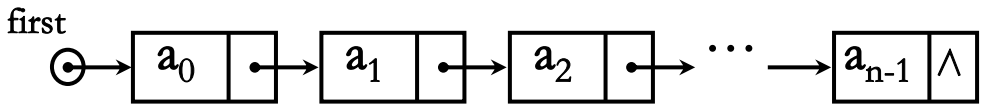
单链表

带表头结点的链表

循环链表

双向链表

单链表结构



非空单链表

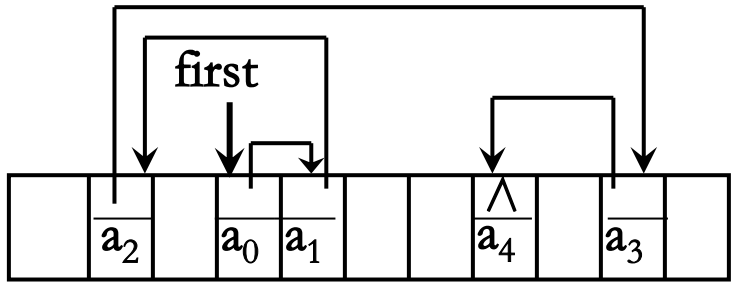
first ∧

空单链表

∧ NULL

first 头指针

绿色为结点的指针域



单链表在内存中的示意图

注意：

- 头结点：第一个结点
- 单链表头指针first不能少,指向头结点（第一个结点）
- 最后一个结点的指针域为∧
- 逻辑上相邻的元素在物理上不一定相邻
- 不能出现“断链”现象

线性表的链接实现可以用下面的C语言结构定义：

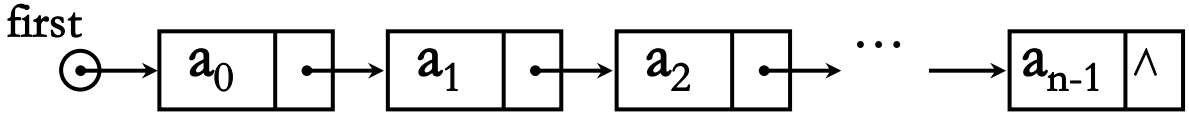
```
typedef struct node{
    ElemType element;
    struct node* link;
} Node;
typedef struct list{
    Node * first;
    int n;
} SingleList;
```

初始化

```
Status Init(SingleList *L)
{
    L->first=NULL;
    L->n=0;
    return OK;
}
```

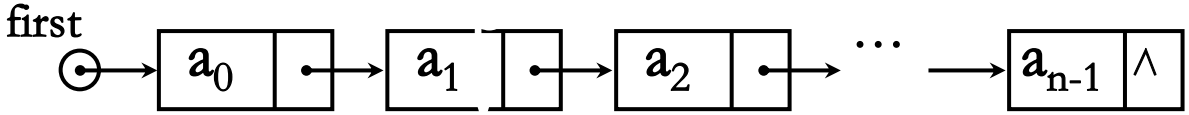
查找运算

由于链表失去了随机查找元素的特性,所以必须从头指针开始沿链逐个计数查找。



查找元素a_i的算法

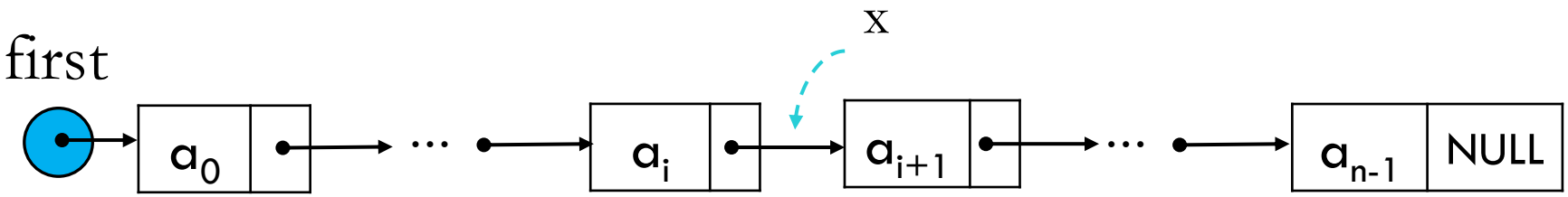
```
Status Find(SingleList L, int i,ElemType *x)
{
    Node *p;
    int j;
    if (i<0 || i>L.n-1) //对i 进行越界检查
    return ERROR;
    p=L.first;
    for ( j=0; j<i; j++) p=p->link; //从头结点开始查找ai
    *x=p->element; //通过x 返回ai 的值
    return OK;
}
```



渐近时间复杂度：O(n)

插入运算

在给定元素a_i之后插入值为x的元素。



插入算法步骤为：

①生成数据域为x的新结点，q指向新结点；

②从first开始找元素a_i，即第i+1个结点，p指向该结点；

③将q插入p之后。

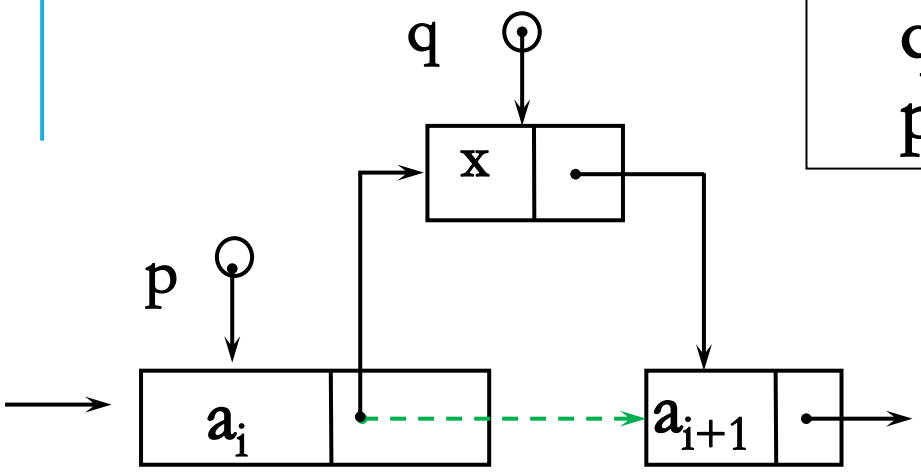
④表长加1。

将q插入p之后:

插入时只要修改二个指针:

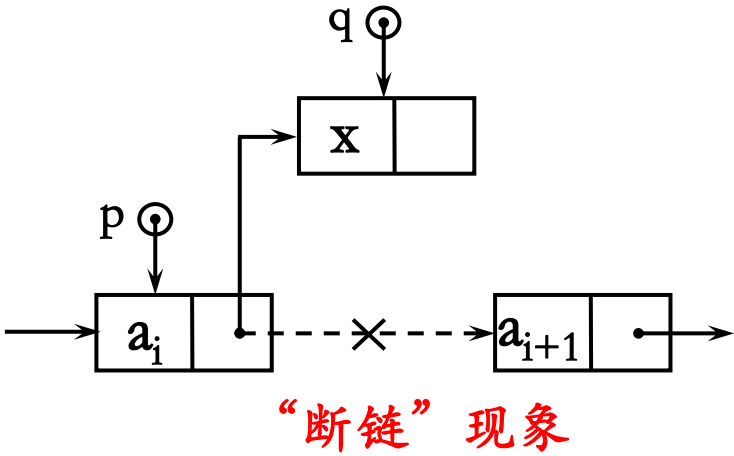
q->link=p->link

p->link=q

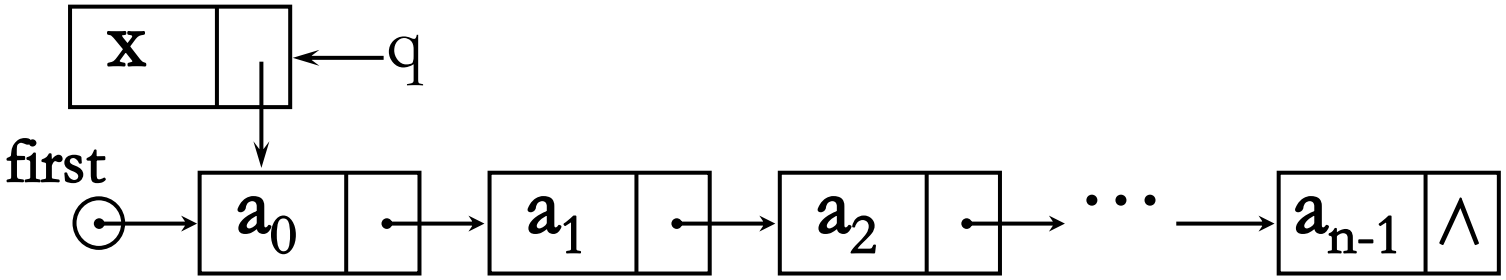


能否对调上述语句执行顺序?

不能，会“断链”现象

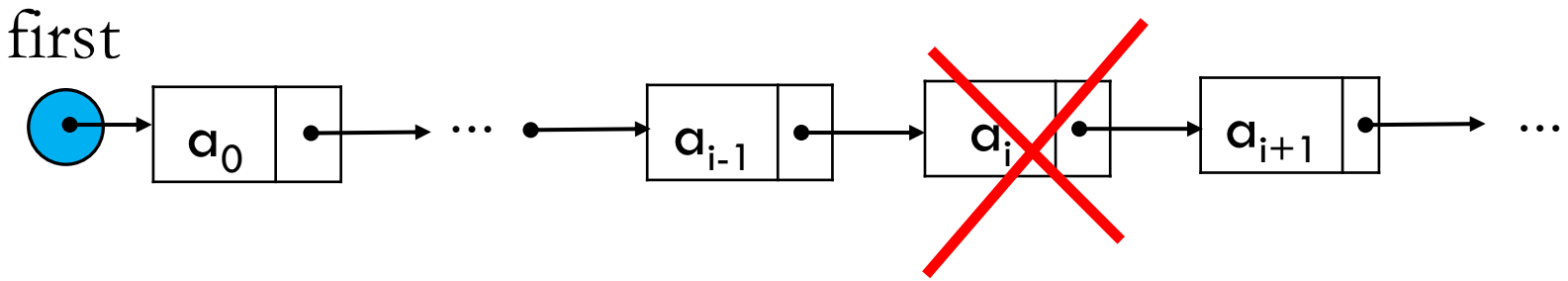


“断链”现象



```
Status Insert(SingleList *L,int i,ElemType x)
{
   Node *p,*q; int j;
   if (i<-1 || i>L->n-1) return ERROR;
   p=L->first;
   for ( j=0; j<i; j++) p=p->link; //从头结点开始查找ai
   q=(Node*)malloc(sizeof(Node)); //生成新结点q
   q->element=x;
   if(i>-1)
   { q->link=p->link; //新结点插在p 所指结点之后
   p->link=q;}
   else
   {q->link=L->first; //新结点插在头结点之前，成为新的头结点
   L->first=q;}
   L->n++;
   return OK;
}
```

删除运算



删除算法步骤为：

①从first开始找到a_{i-1}结点，q指向该结点，p指向q的后继结点a_i;

②从单链表中删除p;

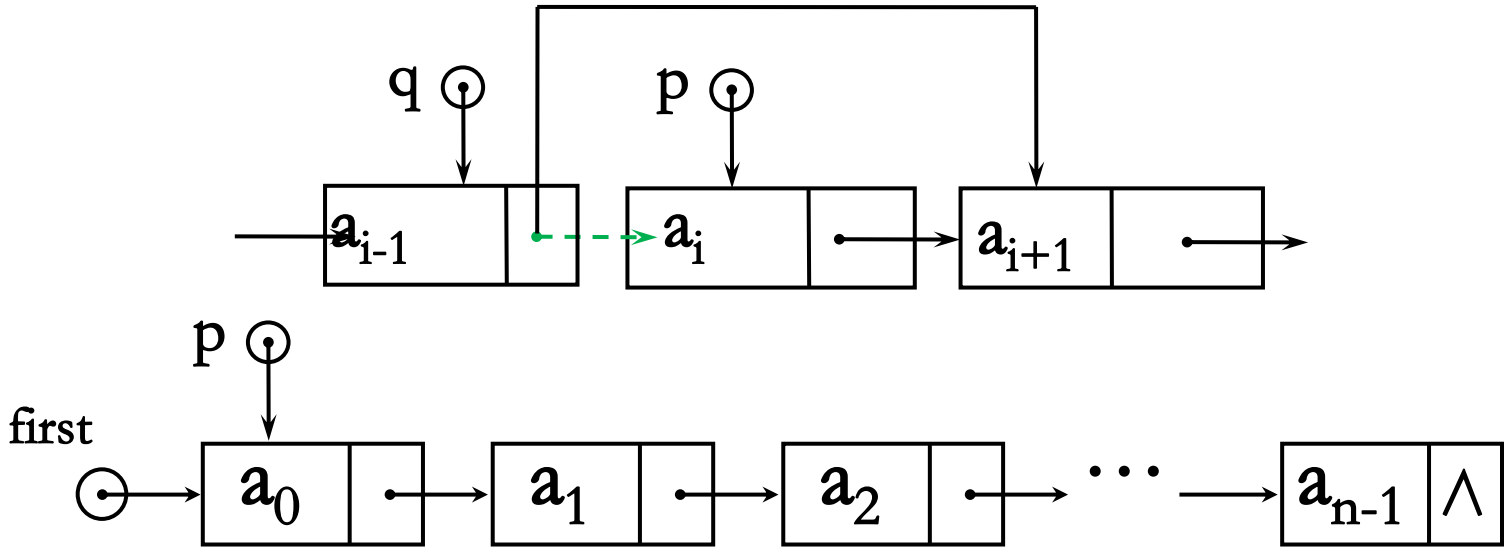
③释放p之空间;

④表长减1。

删除p：

删除时只要修改一个指针：

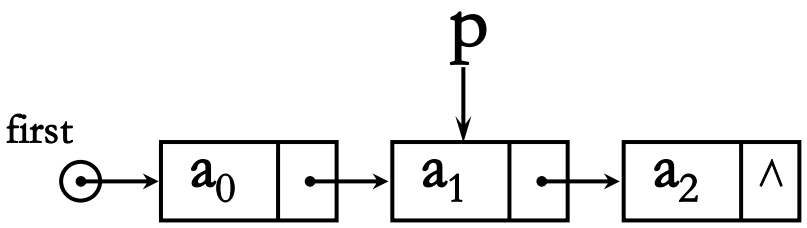
q->link=p->link



```
Status Delete(SingleList *L,int i)
{  int j;  Node *p,*q;
   if (!L->n)  return ERROR;
   if (i<0 || i>L->n-1)  return ERROR;
   q=L->first;
   p=L->first;
   for (j=0; j<i-1; j++) q=q->link;
   if (i==0)
   L->first=q->link; //删除的是头结点
   else
   { p=q->link; //p 指向ai
   q->link=p->link; //从单链表中删除p 所指向的结点}
   free(p); //释放p 所指结点的存储空间
   L->n--;
   return OK; }
```

撤销运算

```
void Destroy (SingleList *L)
{   Node *p;
    while(L-> first )
    { p= L-> first->link;
      free(L->first);
      L-> first=p;
    }
}
```



如何阅读

线性表的链接存储表示的优缺点：

（1）优点：

插入、删除效率高

可动态申请存储空间

（2）缺点：

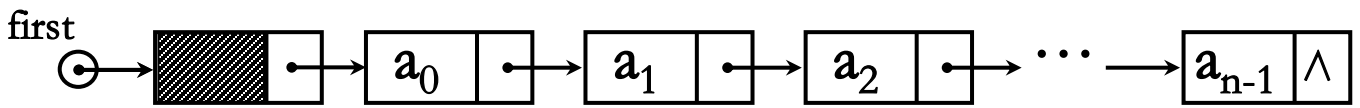
查找指定位置元素效率低；

链接存储需要额外的指针域

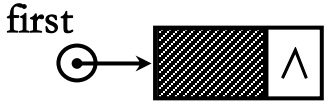
带表头结点的单链表

上一节介绍的单链表在做插入和删除运算时，要考虑一般情况和特殊情况，用带表头结点的单链表可以简化上述两种算法。

在原单链表的头结点之前增加一个结点，但它的element域中不存放线性表中的任何元素，称该结点为表头结点。



非空表



空表

带表头结点的单链表的类型定义

```
typedef struct
{
   struct node * head;
   int n;
}HeaderList;
```

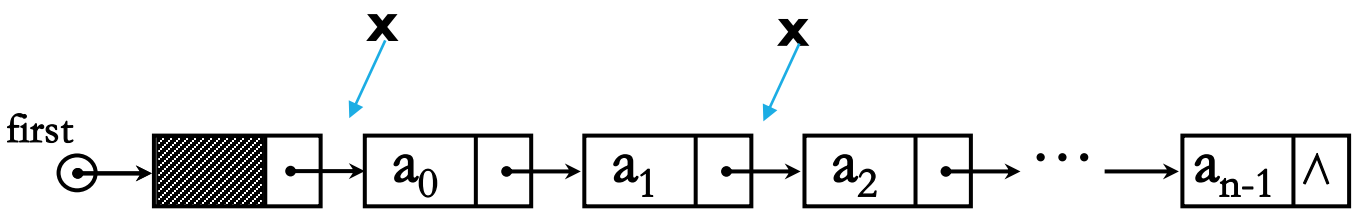
带表头结点的单链表的运算的具体实现

初始化

```
Status Init(HeaderList *h)
{
   h->head=(Node*)malloc(sizeof(Node)); //生成表头结点
   if (!h->head)
   return ERROR;
   h->head->link=NULL; //设置单链表为空表
   h->n=0;
   return OK;
}
```

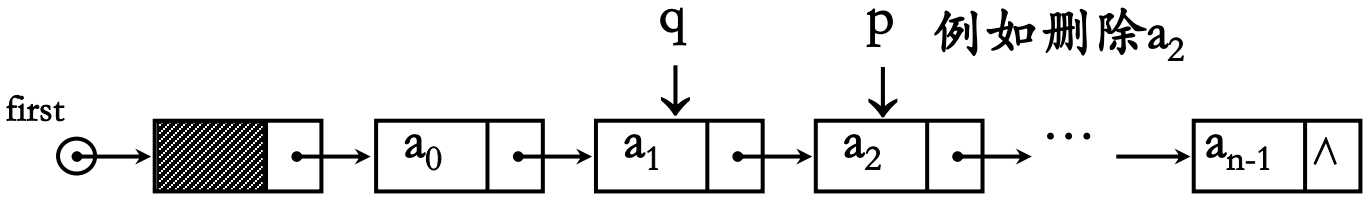
插入操作

```
Status Insert(HeaderList *h, int i,ElemType x)
{
   if( i<-1 || i>n-1 ) return ERROR;
   Node* p=h->head;
   for (int j=0; j<=i; j++) p=p->link;
   Node* q= (Node*)malloc(sizeof(Node)); //生成新结点
   q->element=x;
   q->link=p->link; //新结点插入表中
   p->link=q;
   n++;
   return OK;
}
```



删除操作

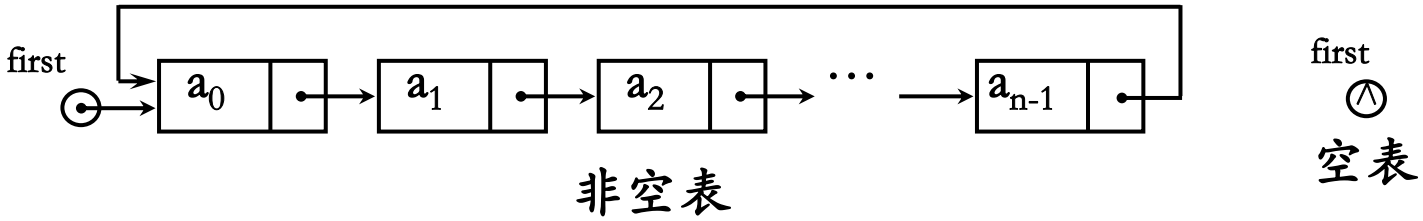
```
Status Delete(HeaderList *h, int i)
{
   int j; Node *p,*q;
   if (!h->n) return ERROR;
   if (i<0 || i>h->n-1) return ERROR;
   q=h->head;
   for (j=0; j<i; j++) q=q->link;
   p=q->link;
   q->link=p->link; //从单链表中删除p 所指结点
   free(p); //释放p 所指结点的存储空间
   h->n--;
   return OK;
}
```



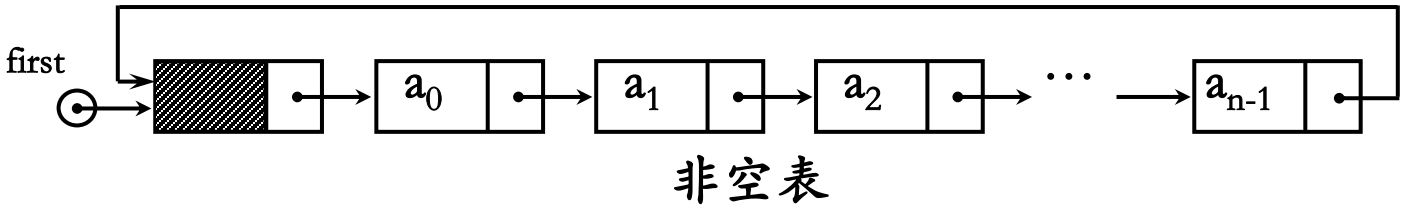
单循环链表

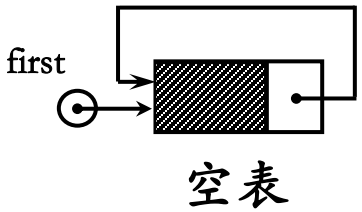
将单链表中最后一个结点的指针域存储头结点的地址，使得整个单链表形成一个环，这种头尾相接的单链表称为单循环链表。

不带表头结点的单循环链表



带表头结点的单循环链表

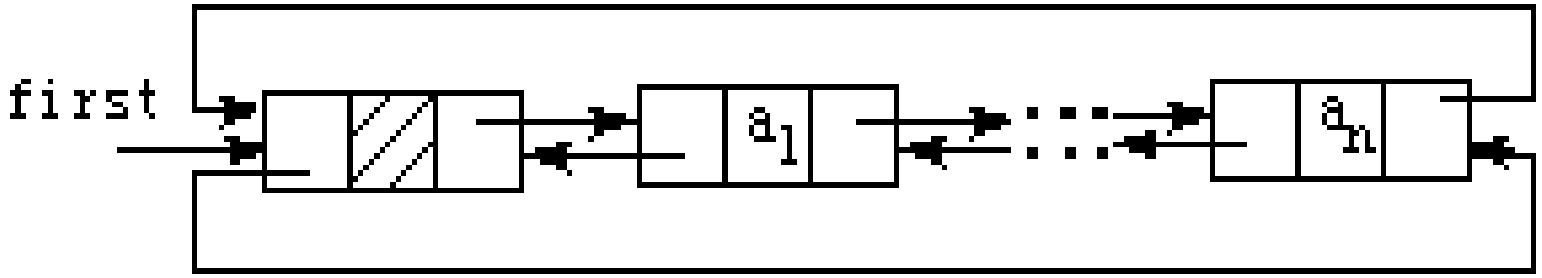




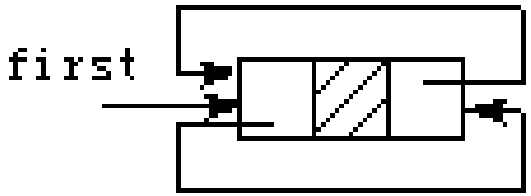
双向链表的结点结构

双向链表的每个结点的结构具有如下定义的结构类型：

```
typedef struct dnode{
    ElemType element;
    struct dnode* rLink，*lLink;
} DNode;
```



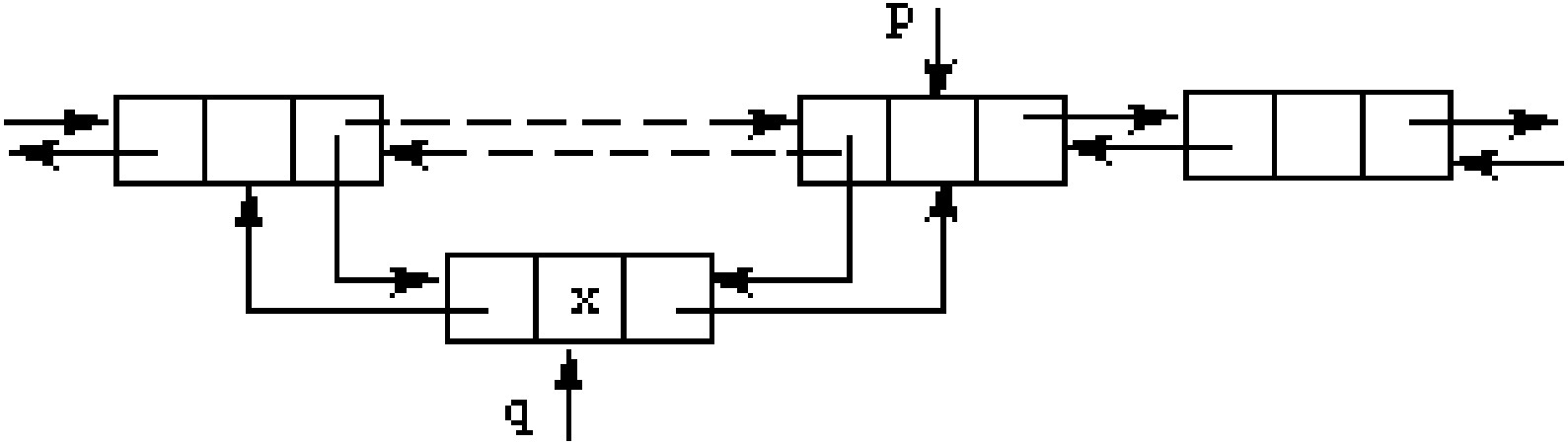
(a) 非空表



(b)空表

图2-8 带表头结点的双向循环链表

双向链表的插入



(a) 双向链表的插入

插入操作的核心步骤是：

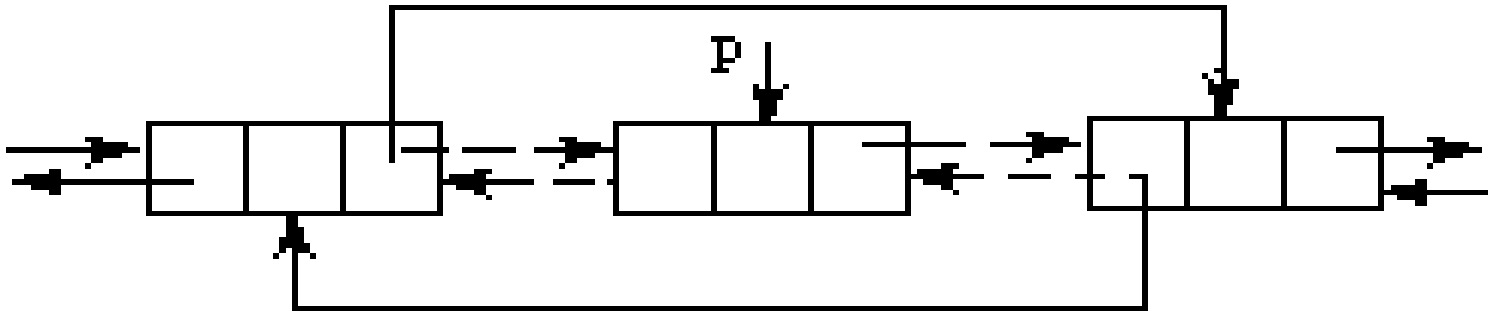
(1) DNode* q= (DNode*) malloc(sizeof(DNode));

(2) q->lLink=p->lLink; q->rLink=p;

p->lLink->rLink=q; p->lLink=q;

<=能否颠倒顺序?

双向链表的删除



(b) 双向链表的删除

删除操作的核心步骤是:

(1) p->lLink->rLink=p->rLink;

p->rLink->lLink=p->lLink; <=能否颠倒顺序?

(2) free(p) ;

目录

一元整系数多项式

$p(x)=3x^{14}-8x^{8}+6x^{2}+2$

要求:(1)按降幂排列　　(2)做q(x)←q(x)+p(x)

分析:

1. 数据元素

该多项式是由一项一项组成的。我们忽略掉各项具体数字的细节，即每一项就是由系数(coef)和指数(exp)组成的：$coef \cdot x^{exp}$ 。

每一项就是一个要处理的数据元素，即

(coef,exp) 。

2. 元素间的逻辑关系

由于多项式按降幂排列，因此每一项都有一个指数比它高的项，有一个比它低的项，除了最高项没有比它高的项，最后一项没有比它低的项外。这样，多项式的每一项就组成了一个线性表，线性表中的每个数据元素是多项式的一项(coef,exp)。

因此，一元整系数多项式可以视为线性表。

3. 存储表示

$q(x)=2x^{10}+4x^{8}-6x^{2}$，$p(x)=3x^{14}-8x^{8}+6x^{2}+2$

做$q(x)+p(x)\Rightarrow q(x)$，

结果为：$q(x)=3x^{14}+2x^{10}-4x^{8}+2$

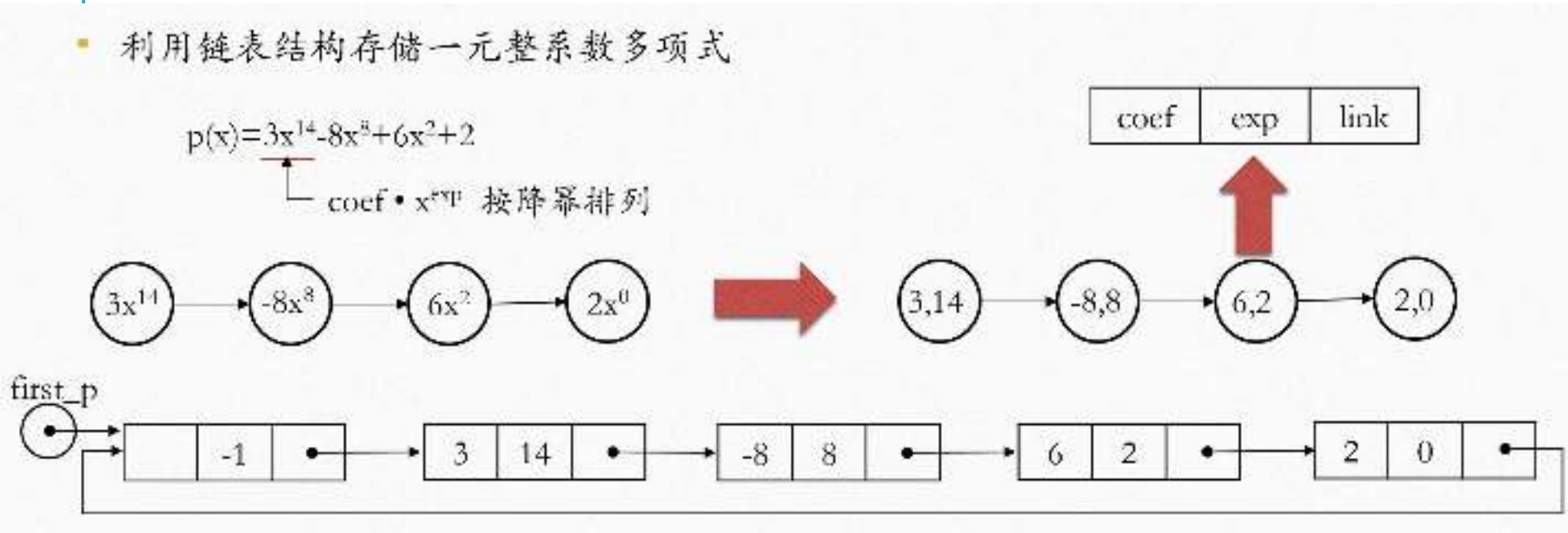
线性表有顺序和链接存储：

1.用顺序存储：

q(x)	(2,10)	(4,8)	(-6,2)	
p(x)	(3,14)	(-8,8)	(6,2)	(2,0)
结果q(x)	(3,14)	(2,10)	(-4,8)	(2,0)

从结果中可以发现，在q(x)上做了插入和删除运算，因此不宜采用顺序存储。

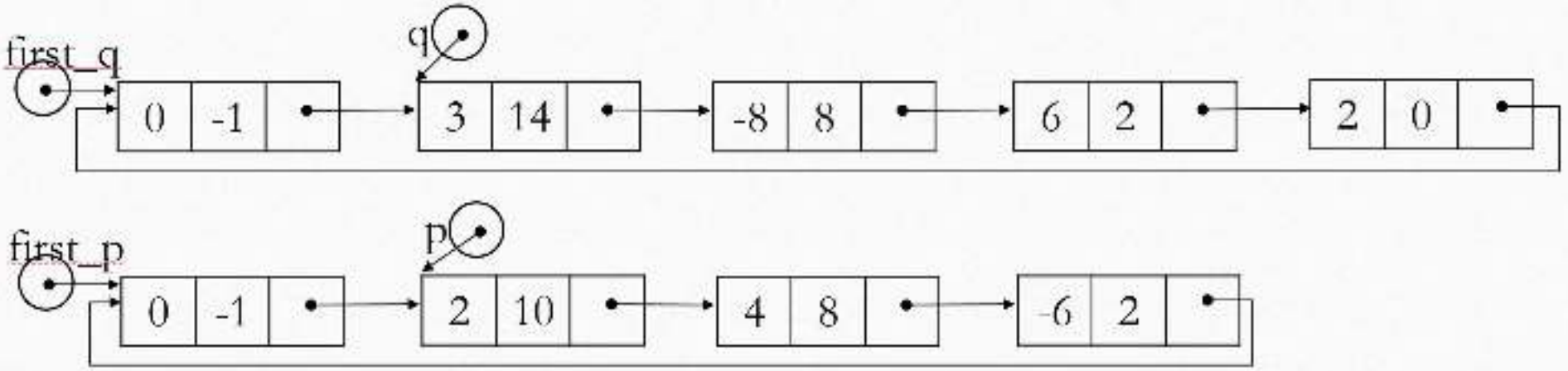
多项式的算术运算



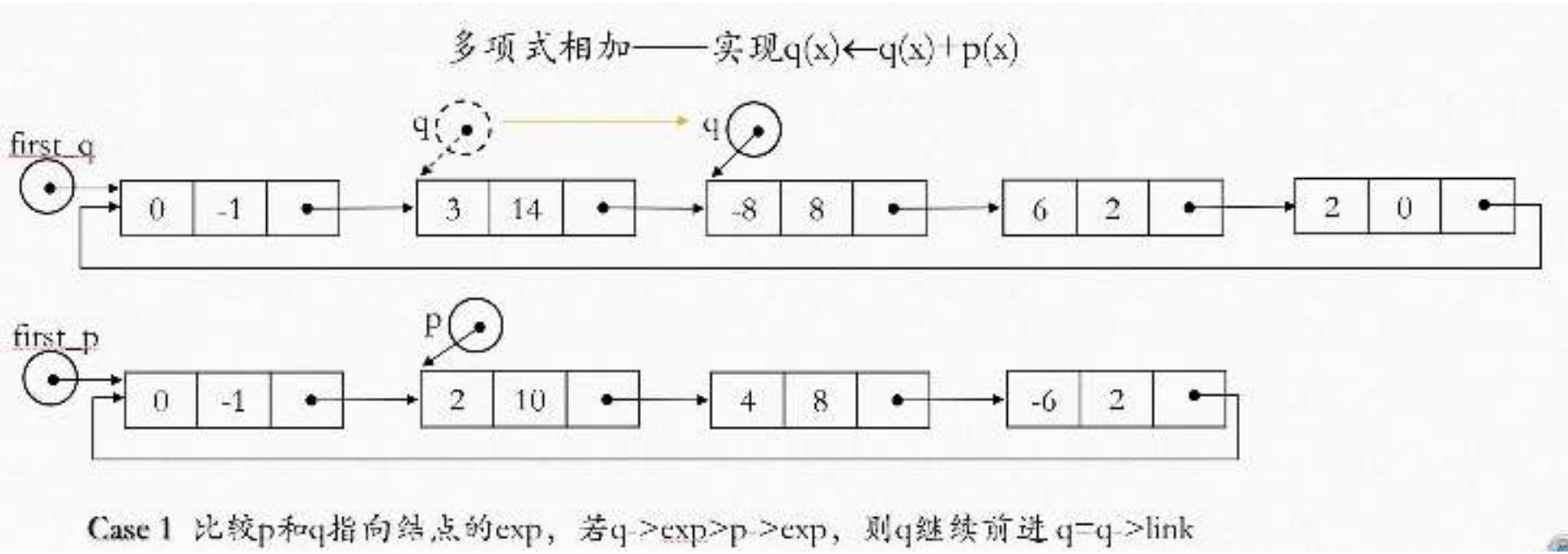
多项式相加——实现q(x)←q(x)+p(x)

- 设p和q分别指向多项式p(x)和q(x)的当前正进行比较的项
- 初始时分别指向两多项式中最高幂次的项
- 对p(x)进行遍历，根据指针p、q的exp域的大小情况做相应处理

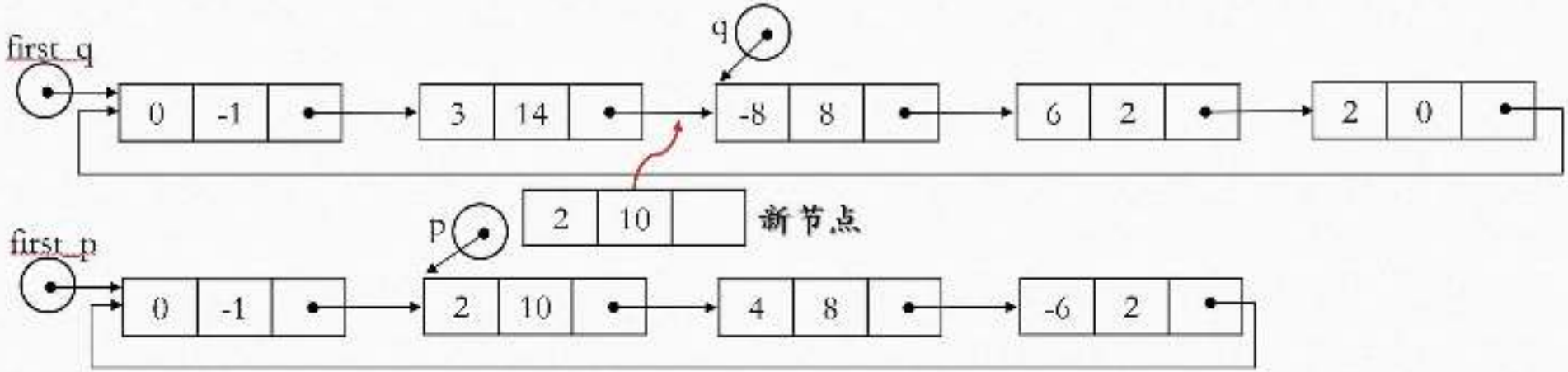
多项式相加——实现q(x)←q(x)+p(x)



初始时，p和q指向头结点

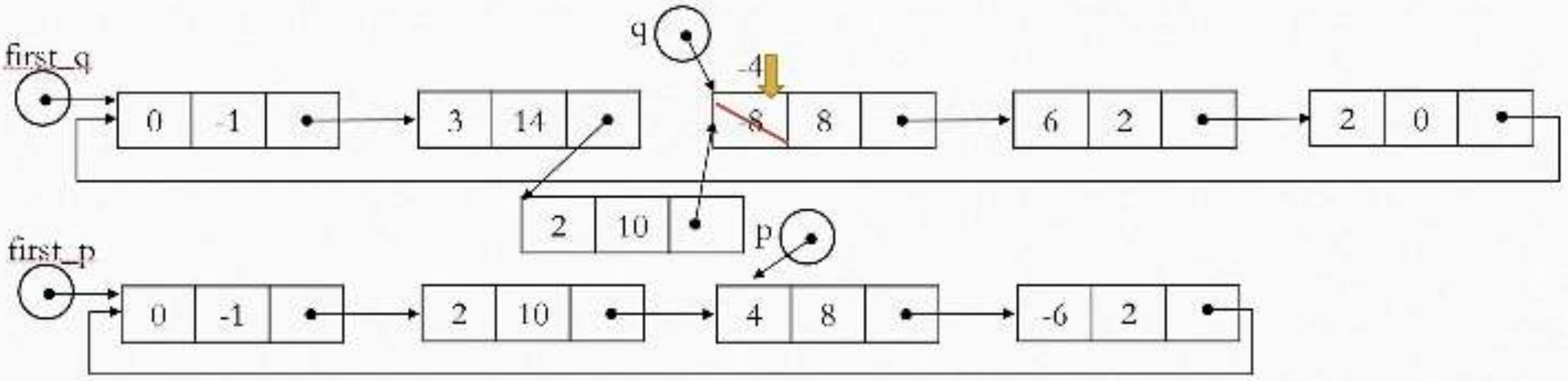

多项式相加——实现q(x)←q(x)+p(x)
first_q
q
q
0
-1
3
14
-8
8
6
2
2
0
first_p
p
0
-1
2
10
4
8
-6
2
Case 1 比较p和q指向结点的exp，若q->exp>p->exp，则q继续前进 q=q->link

多项式相加——实现q(x)←q(x)+p(x)



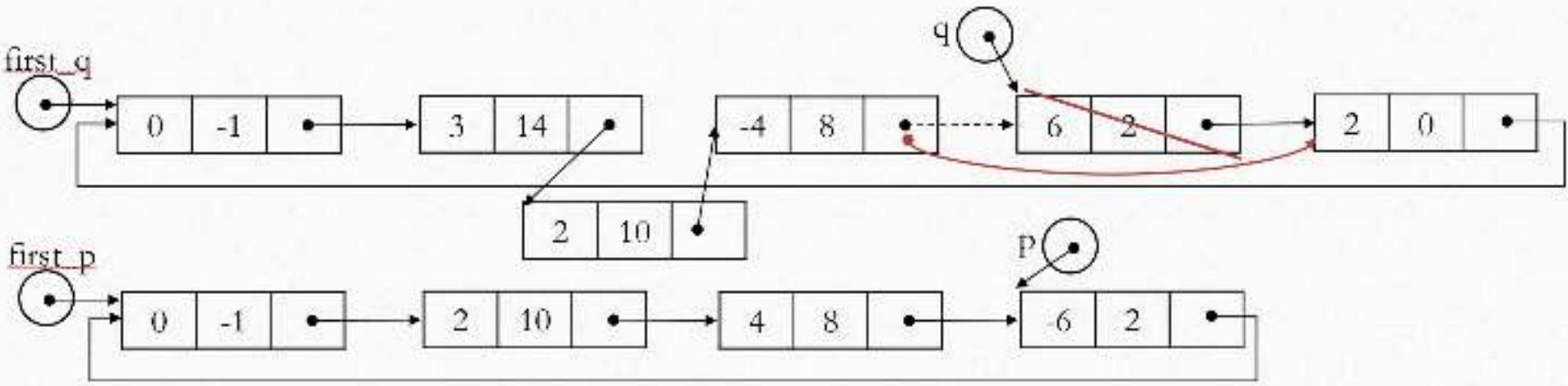
Case 2 比较p和q指向结点的exp，若q->exp<p->exp，则创建一个新的term结点插入到q所指向结点之前，并将q所指向结点的coef和exp赋值给新结点，p继续前进

多项式相加——实现q(x)←q(x)+p(x)



Case 3 比较p和q指向结点的exp，若p->exp == q->exp，则令q->coef = q->coef + p->coef，p和q继续前进

多项式相加——实现q(x)←q(x)+p(x)



Case 4 比较p和q指向结点的exp，若p->exp == q->exp，则令q->coef = q->coef + p->coef，若相加后，q->coef ==0，则删除q所指向结点，p和q继续前进

表达式计算

- 后缀表达式求值算法:

abc-/de*+

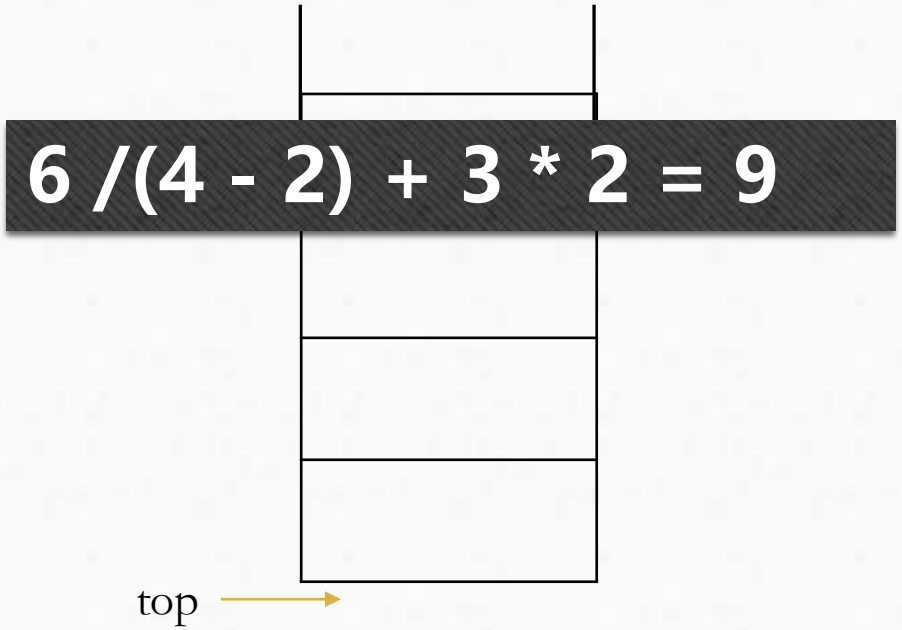
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



表达式计算

- 后缀表达式求值算法:

abc-/de*+

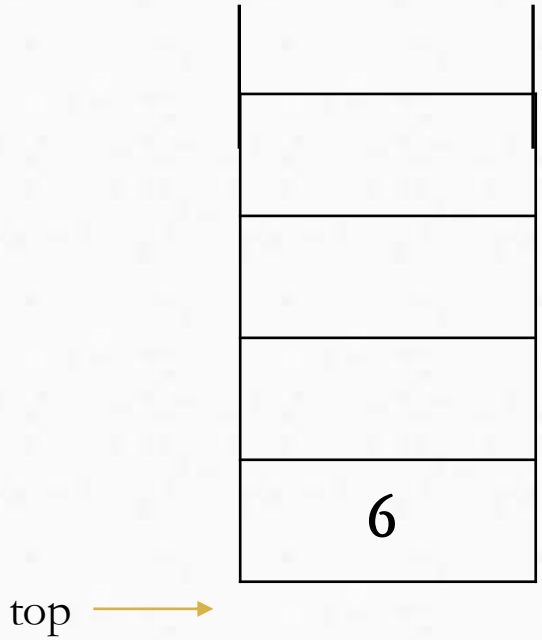
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



表达式计算

- 后缀表达式求值算法:

abc-/de*+

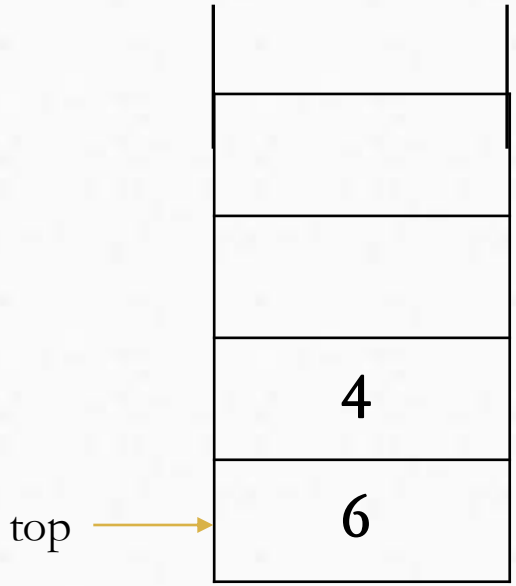
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



表达式计算

- 后缀表达式求值算法:

abc-/de*+

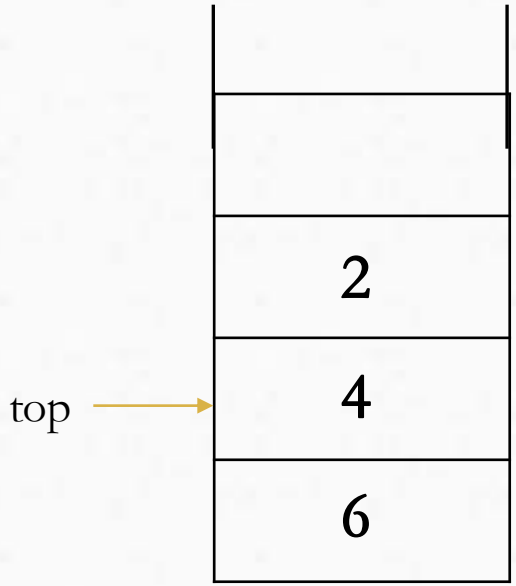
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



表达式计算

- 后缀表达式求值算法:

abc-/de*+

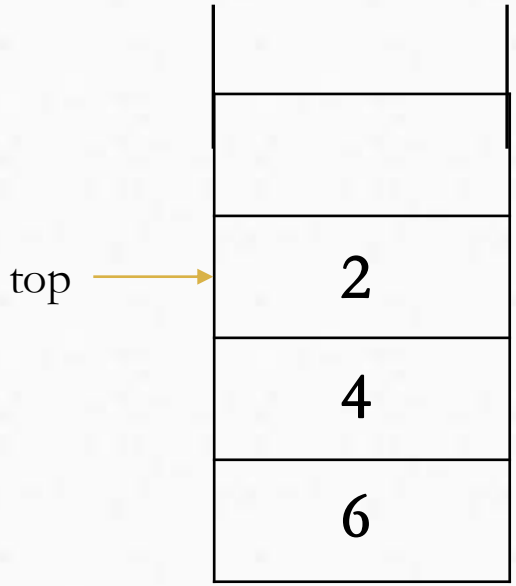
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



4-2=2

表达式计算

- 后缀表达式求值算法:

abc-/de*+

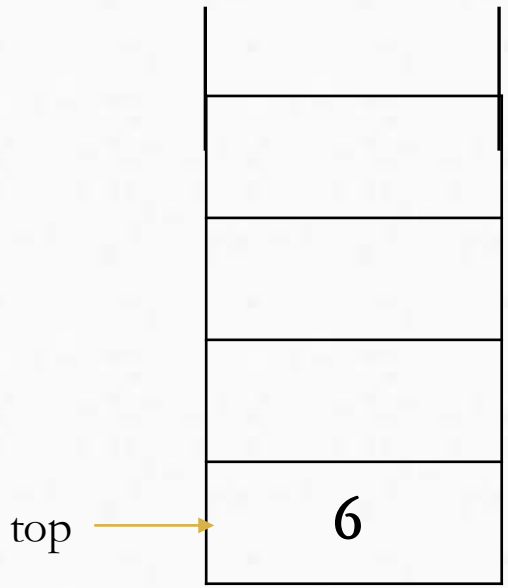
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



4-2=2

表达式计算

- 后缀表达式求值算法:

abc-/de*+

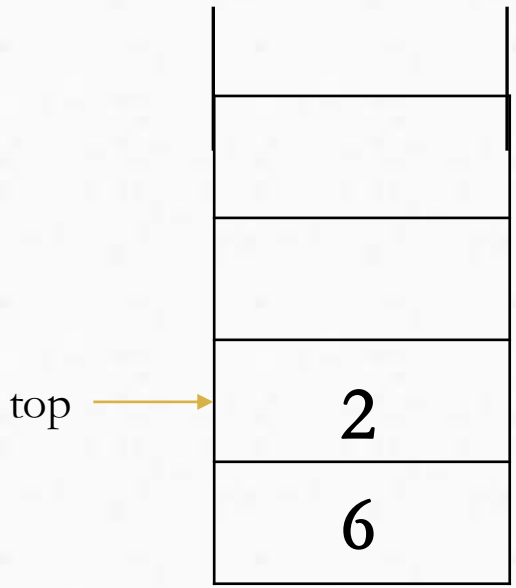
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



6/2=3

表达式计算

- 后缀表达式求值算法:

abc-/de*+

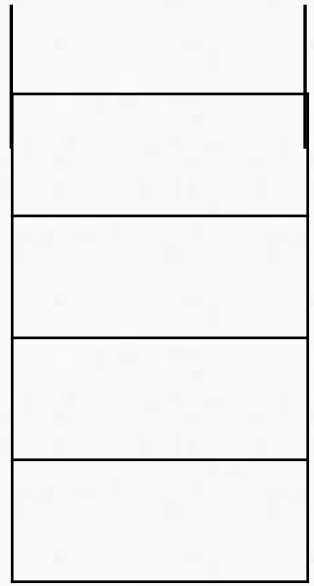
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



top

6/2=3

表达式计算

- 后缀表达式求值算法:

abc-/de*+

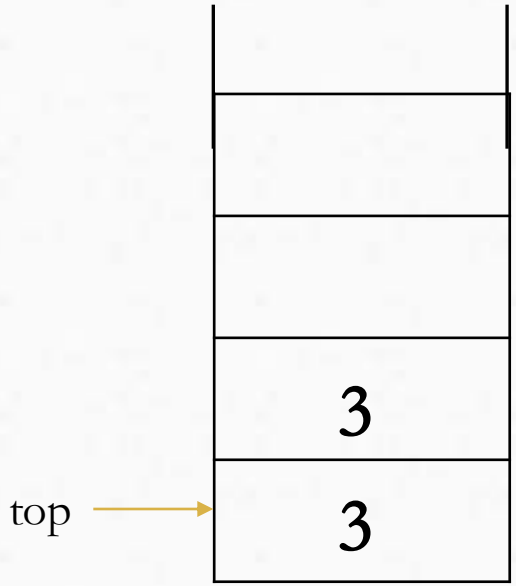
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



表达式计算

- 后缀表达式求值算法:

abc-/de*+

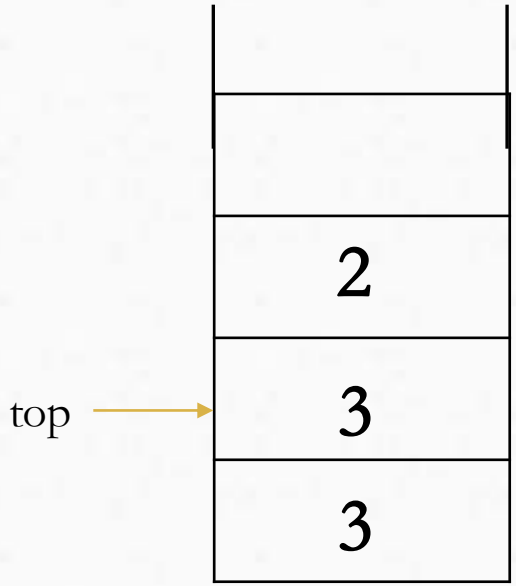
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



表达式计算

- 后缀表达式求值算法:

abc-/de*+

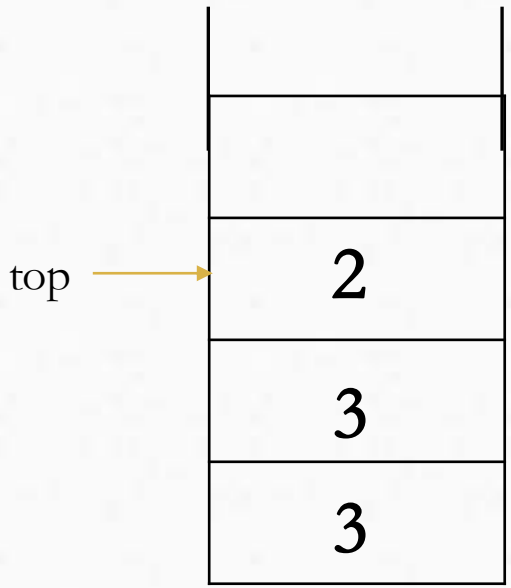
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



3*2=6

表达式计算

- 后缀表达式求值算法:

abc-/de*+

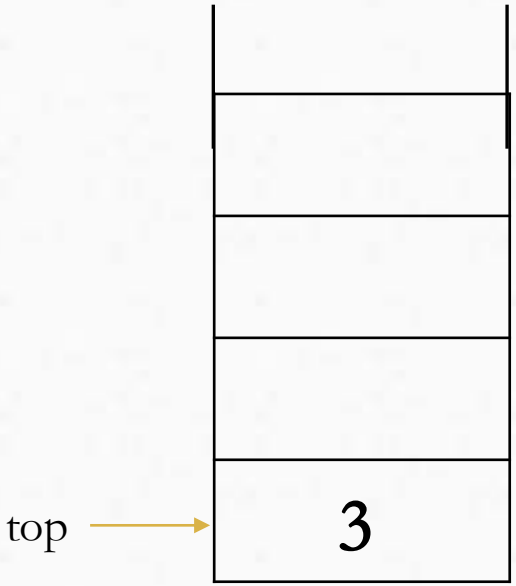
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



3*2=6

表达式计算

- 后缀表达式求值算法:

abc-/de*+

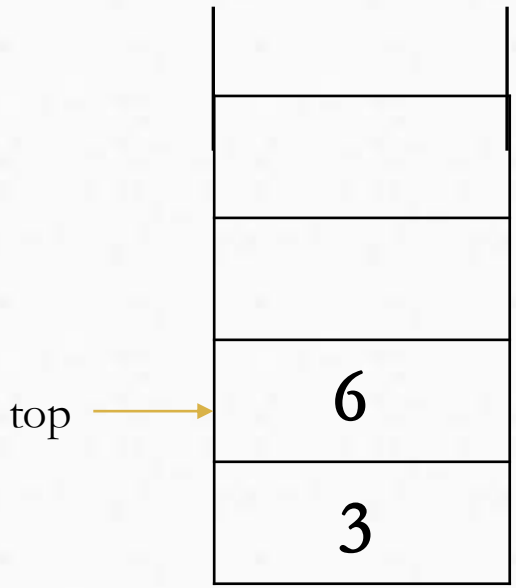
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



3+6=9

表达式计算

- 后缀表达式求值算法:

abc-/de*+

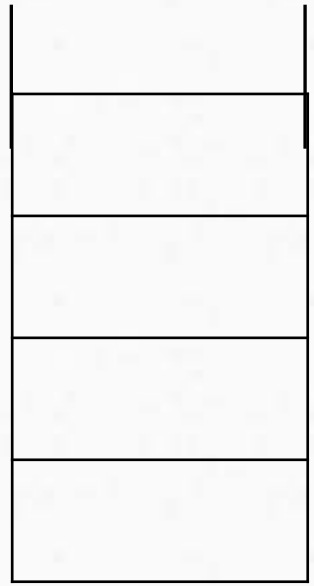
① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #



top

3+6=9

表达式计算

- 后缀表达式求值算法:

abc-/de*+

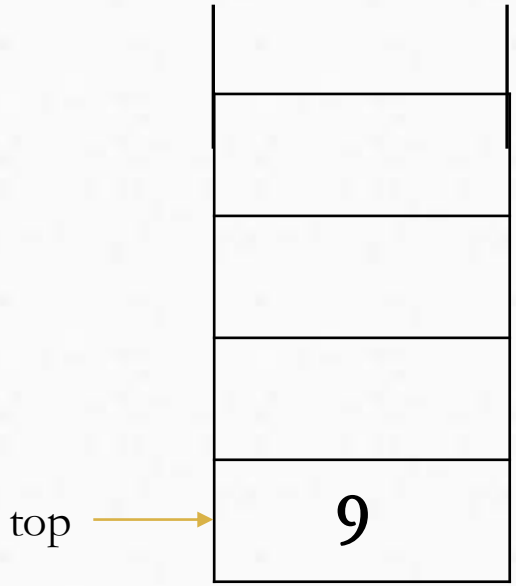
① 从左往右顺序扫描后缀表达式;

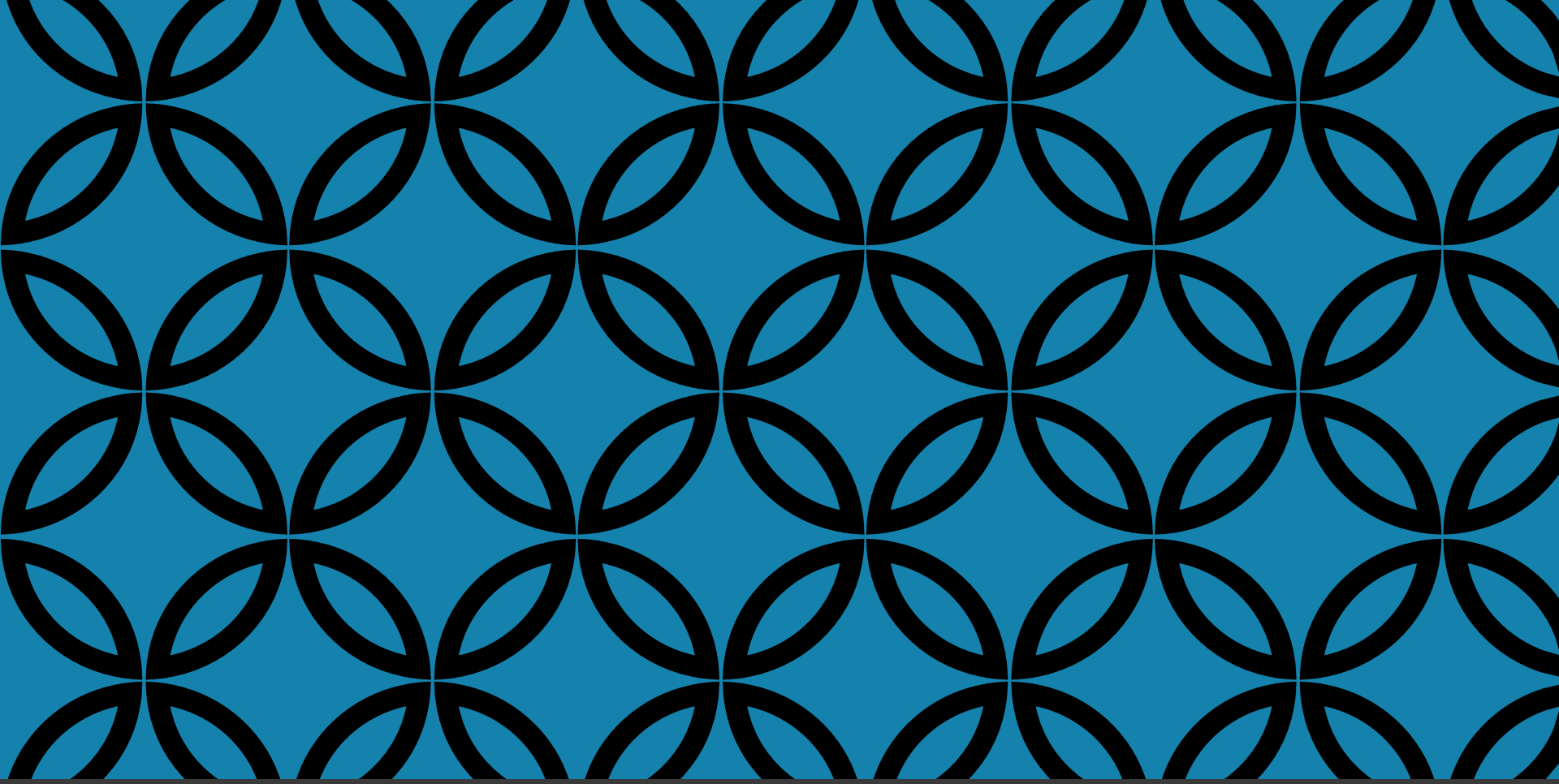
② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

6 4 2 - / 3 2 * + #





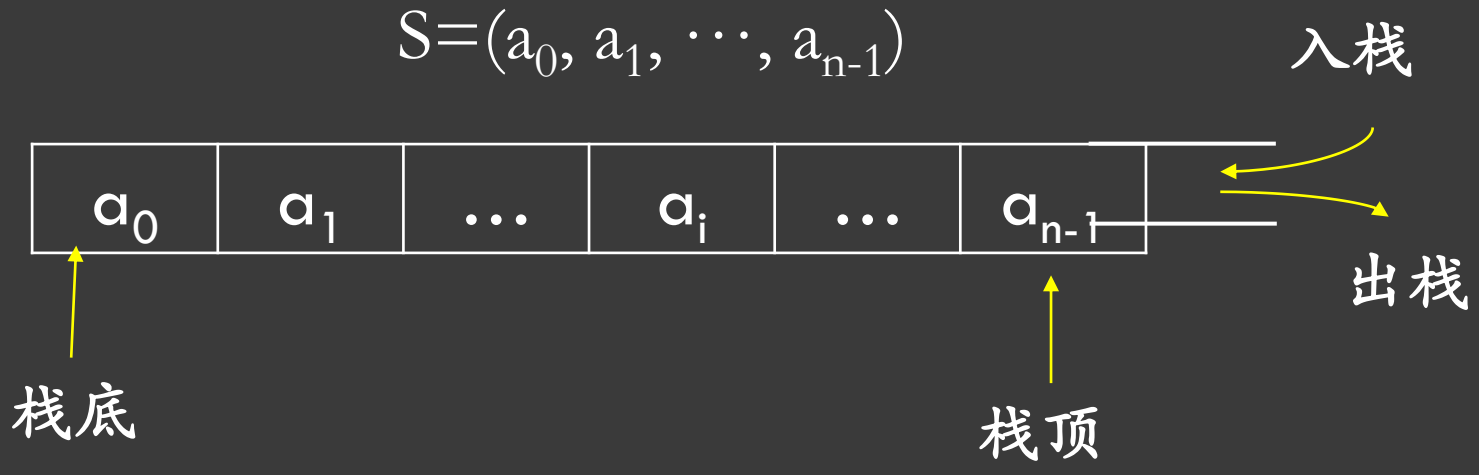
堆栈和队列

目录

堆栈的基本概念

堆栈（Stack，栈）限定插入和删除操作都在同一端进行的线性表

后进先出(LIFO)的线性数据结构



堆栈的基本概念

为什么需要有栈这样的数据结构?
存在一些场景：数据写入一块内存的顺序决定了读出顺序，不允许随机存取这块内存上的数据，仅提供内存一端的读写接口

堆栈的抽象数据类型

ADT Stack{

数据:

n个元素的线性序列(a_0,a_1,...,a_{n-1})，其中线性序列的长度上限为maxSize，且0≤n<maxSize。

运算:

Create(S,maxSize)：建立一个最多能存储maxSize个元素的空堆栈S

Destroy(S)：释放堆栈所占的存储空间

IsEmpty(S)：判断堆栈S是否为空

IsFull(S)：判断堆栈是否已满

Top(S，x)：获取栈顶元素，并通过x返回

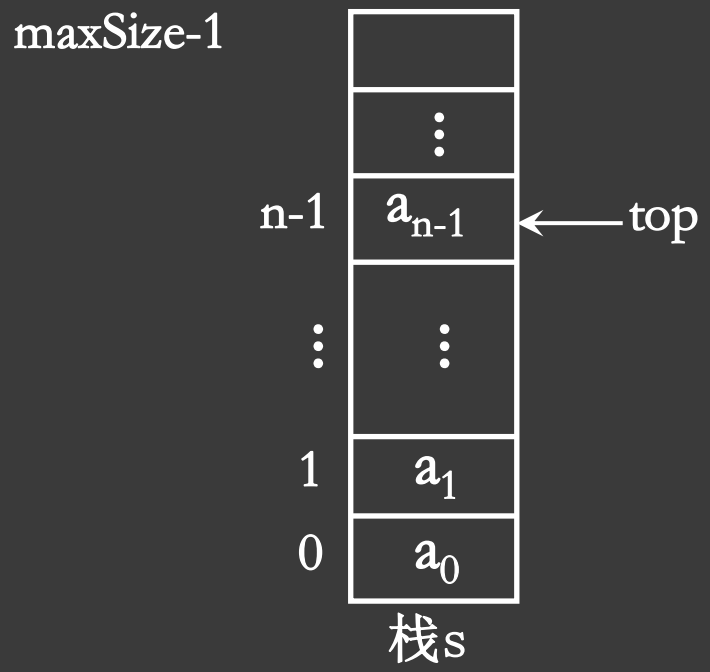
Push(S，x)：在栈顶位置插入元素x（入栈操作）

Pop(S)：删除栈顶元素（出栈操作）

Clear(S)：清除堆栈S中全部元素};

目录

堆栈的顺序存储




```
typedef struct stack{
   int top, maxSize;
   ElemType *element;
 } Stack ;
```

顺序栈

在顺序存储表示下实现栈上定义的操作

（1）创建一个能容纳mSize个单元的空堆栈

```
void Create(Stack *S, int mSize)
{
    S->maxSize=mSize;
    S->element = (ElemType*)malloc(sizeof(ElemType)*mSize);
    S->top=-1;
}
```

（2）销毁一个已存在的堆栈，即释放堆栈占用的数组空间

```
void Destroy(Stack *S)
{
    free(S->element);
    free(S);
}
```

（3）判断堆栈是否为空栈

```
BOOL IsEmpty(Stack *S)
{
    return S->top==-1;
}
```

（4）判断堆栈是否已满

```
BOOL IsFULL(Stack *S)
{
    return S->top==S->maxSize-1;
}
```

（5）获取栈顶元素，并通过x返回

```
BOOL Top(Stack *S, ElemType *x)
{
    if(IsEmpty(S))    return FALSE;   //空栈处理
    *x=S->element[S->top];
    return TRUE;
}
```

（6）在栈顶位置插入元素x（入栈操作）

```
BOOL Push(Stack *S, ElemType x)
{
    if(IsFull(S))  return FALSE;  //溢出处理
    S->top++;
    S->element[S->top]=x;
     return TRUE;
}
```

（7）删除栈顶元素（出栈操作）

```
BOOL Pop(Stack *S)
{
    if(IsEmpty(S))     return FALSE;       //空栈处理
    S->top--;
    return TRUE;
}
```

（8）清除堆栈中全部元素，但并不释放空间

```
void Clear(Stack *S)
{
   S->top=−1;
}
```

目录

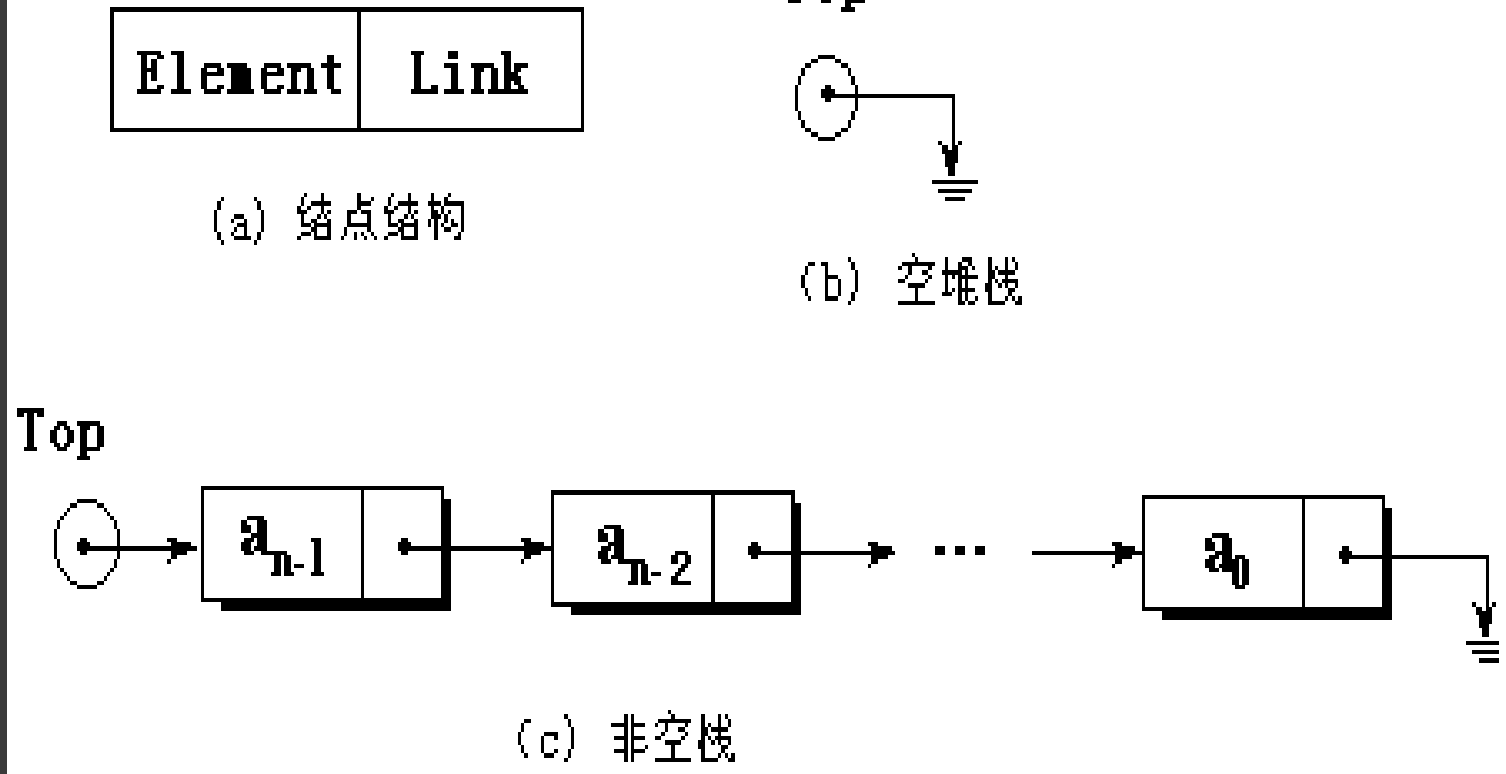


图 3-3 栈的链接表示

其结点类型与单链表相同，

```
typedef struct node{
    ElemType element;
    struct node* link;
} Node;
typedef struct stack{
    Node* top;
}Stack;
```

链式栈的进栈和出栈运算

```
void Push(Stack *S, ElemType x)
{
    Node* p= (Node*)malloc(sizeof(Node));
    p->element = x; p->link = NULL;
    p->link=S->top; S->top=p;
}
 void Pop(Stack *S)
 {
     Node *p=S->top;
     S->top=p->link;
     free(p);
 }
```

目录

表达式计算

中缀表达式：操作符在两个操作数之间的表达式

a/(b-c)+d*e　　操作数 操作符 界限符

后缀表达式：操作符放在两个操作数之后的表达式（逆波兰表达式）

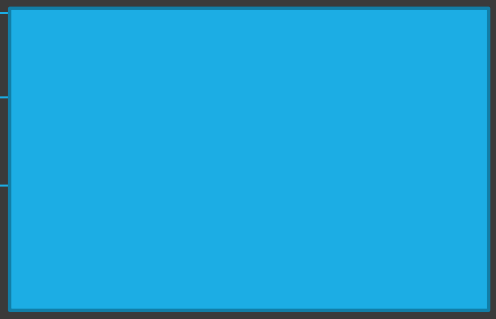
abc-/de*+　　操作数 操作符

后缀表达式的优点

- 无界限符
- 求值时无需考虑操作符的优先级

表达式计算

a*b+c	ab*c+
a*b/c	ab*c/
a*b*c*d*e*f	ab*c*d*e*f*
a+(b*c+d)/e	
a*((b+c)/(d-e)-f)	
a/(b-c)+d*e	



表达式计算

后缀表达式求值算法: abc-/de*+

① 从左往右顺序扫描后缀表达式;

② 遇到操作数就进栈;

③ 遇到操作符就从栈中弹出两个操作数，并执行该操作符规定的运算；并将结果进栈;

④ 重复上述操作，直到表达式结束，弹出栈顶元素即为结果。

表达式计算

后缀表达式计算过程演示

中缀表达式转换成后缀表达式

转换的关键：确定操作符的优先级

- 优先级决定操作符是进栈或出栈。
- 操作符在栈内外的优先级应该不同

  “(”的优先级在栈外最高，但进栈后应该比除#外的操作符低，便于括号内的其它操作符进栈。

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

后缀计算：操作数入栈出栈
中转后：操作符入栈出栈

中缀表达式转换成后缀表达式

1. 从左到右扫描中缀表达式，遇到#转（2）；
   ①遇到操作数直接输出；
   ②遇到“)"，则连续出栈，直到“("为止
   ③遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈
2. 输出栈中剩余操作符(#除外)。

表达式计算

中缀表达式转后缀表达式过程演示

目录

队列的基本概念

队列（Queue）是一种限定在表的一端插入，在表的另一端删除的线性表

$$Q=(a_0, a_1, \cdots, a_{n-1})$$

先进先出(FIFO)的线性数据结构

为什么需要有**队列**这样的数据结构?
存在一些任务调度的场景，需要保证先到的任务先处理
例如：CPU资源调度、打印任务调度

ADT Queue{

数据:

n个元素的线性序列($a_0,a_1,...,a_{n-1}$)，其最大允许长度为maxSize，且$0 \leqslant n < maxSize$。元素插入在一端进行，而删除在另一端进行，并遵循FIFO原则。

操作:

Create(Q,maxSize)：建立最多能存储maxSize个元素的空队列Q

Destroy(Q)：释放队列Q申请的存储空间

IsEmpty(Q)：判断队列是否为空

IsFull(Q)：判断队列是否已满

Front(Q,x)：获取队列Q的队头元素，并通过x返回

EnQueue(Q，x)：在队列Q的队尾插入元素x（入队操作）

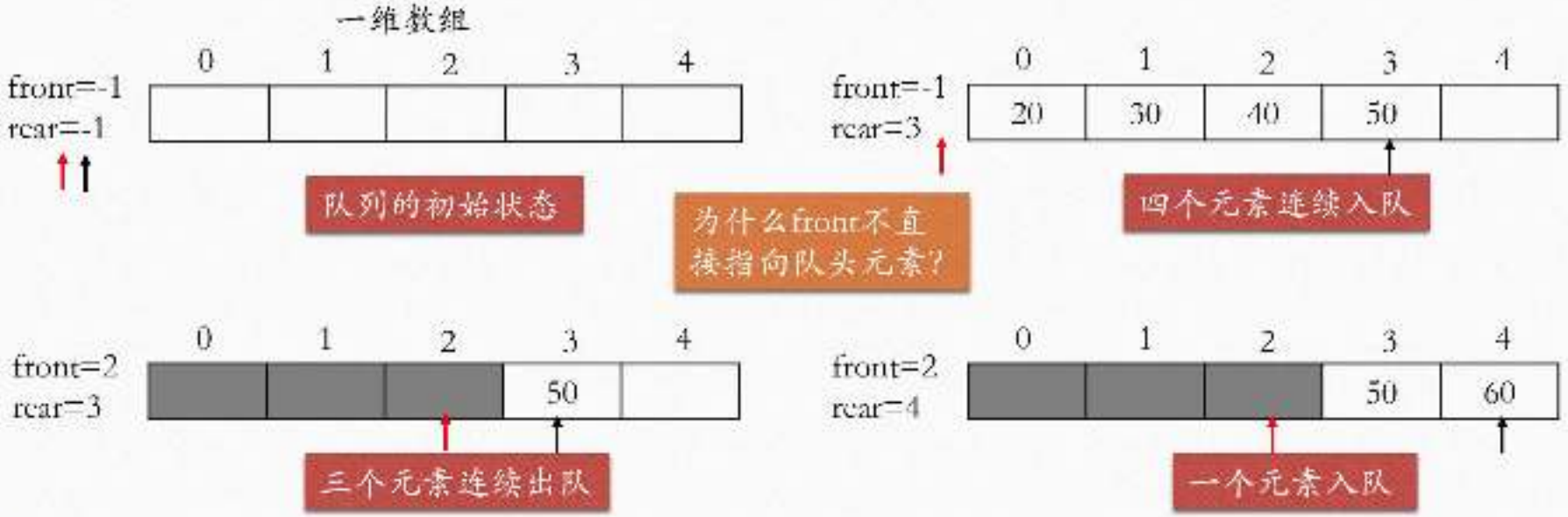
DeQueue(Q)：从队列Q中删除队头元素（出队操作）

Clear(Q)：清除队列中全部元素

}

目录

队列的顺序存储表示

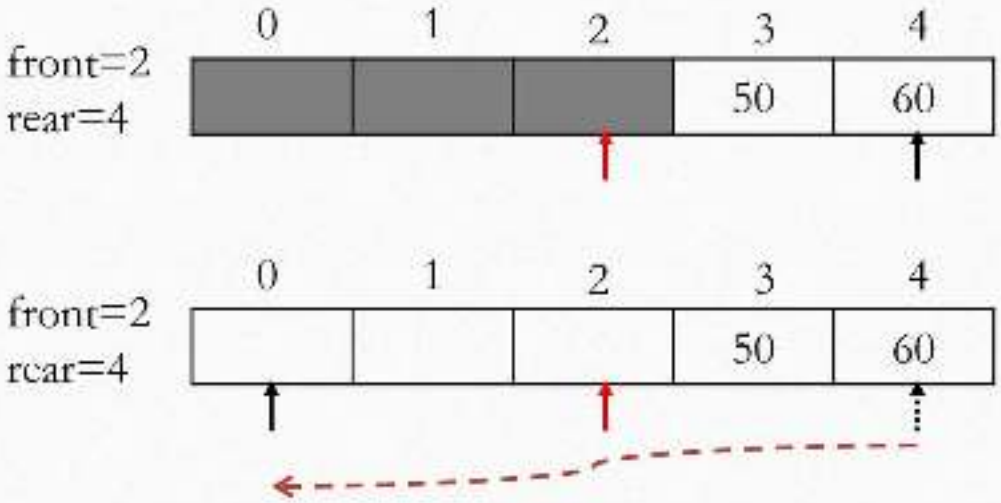


元素入队时：rear++

元素出队时：front++

如果front直接指向队头元素，此时最后一个元素50出队后，会发生什么情况?

队列的顺序存储表示



此时再有元素入队发生“假溢出”

解决方案：想象队列首位相连，令rear“前进”至队头

队头队尾标识前进方式：

front = (front+1)%maxQueue

rear = (rear+1)%maxQueue

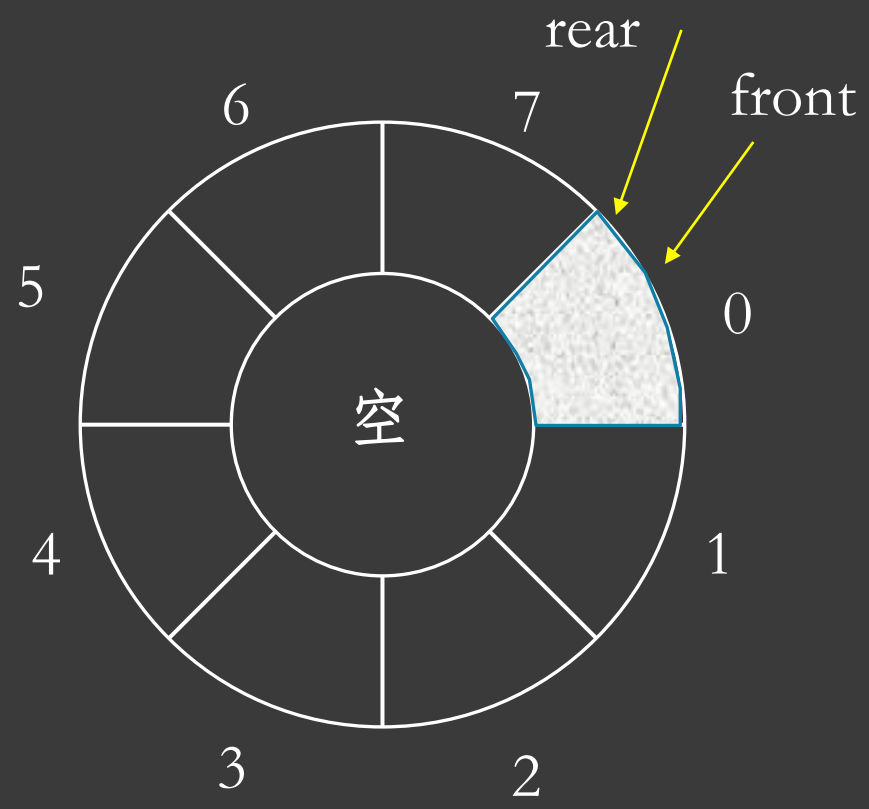
队列的顺序存储表示

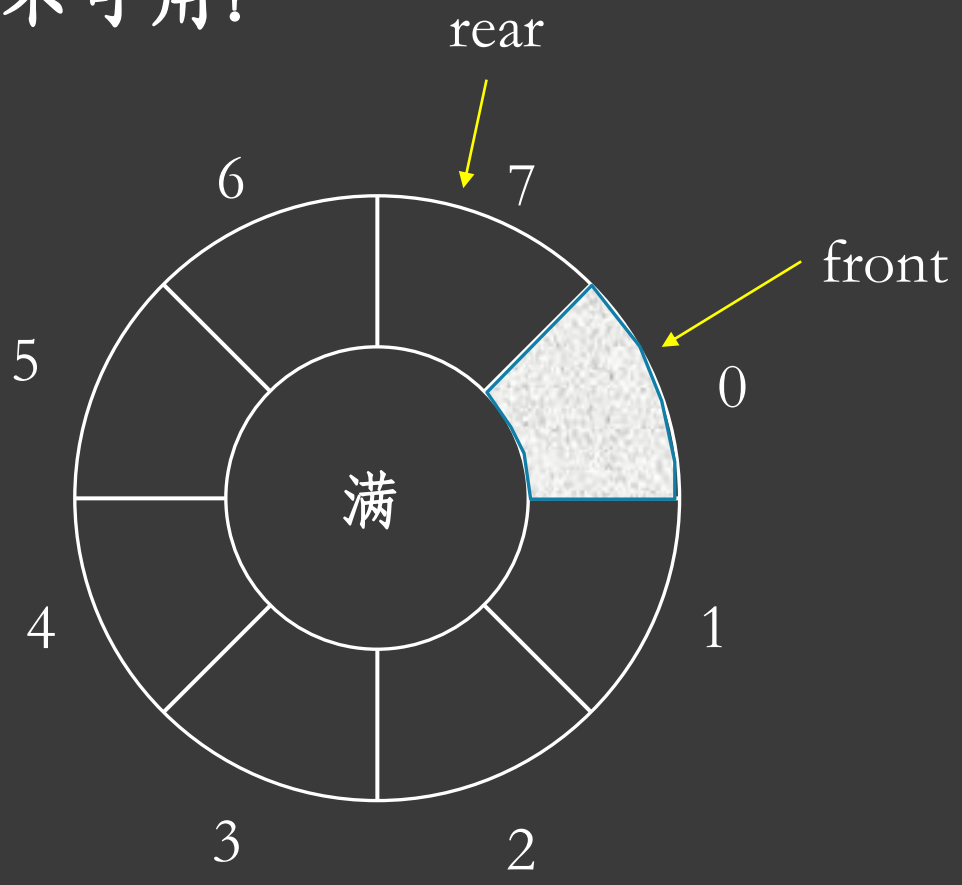
循环队列元素出入演示

思考

需要一个不可用的空间，
来区别判空和判满条件

为什么front指向的空间不可用？





实现循环队列操作：

(1)为使入队和出队实现循环，可以利用取余运算符%

(2)队头标识进一（出队）：

front=(front+1) % maxSize;

(3)队尾标识进一（入队）：

rear=(rear+1) % maxSize ;

(4)空队列： 当front==rear 时为空队列，

(5)满队列： 当(rear+1) % maxSize == front 时为满队列。满队列时实际仍有一个元素的空间未使用。

队列的顺序实现可以用下面的C语言结构定义:

```
typedef struct queue{
   int front，  rear, maxSize;
   ElemType *element;
 } Queue
```

循环队列算法实现

```
void Create(Queue *Q,int maxsize)
 {
      Q->maxSize=mSize;
      Q->element=(ElemType*)malloc(sizeof(ElemType)*mSize;
      Q->front=Q->rear=0;
 }

void Clear(Queue *Q){
    Q->front = Q->rear = 0;
}
```

循环队列算法实现

```
BOOL IsEmpty(Queue Q)
 {
     return Q.front==Q.rear;
 }
 BOOL IsFull(Queue Q)
 {
     return (Q.rear+1) % Q.maxSize==Q.front;
 }
```

在队列Q的队尾插入元素x（入队操作）

```
BOOL EnQueue(Queue *Q, ElemType x)
{
    if (IsFull(Q))   return FALSE;   //溢出处理
    Q->rear=(Q->rear+1)%Q->maxSize;
    Q->element[Q->rear]=x;
    return TRUE;
}
```

从队列Q中删除队头元素（出队操作）

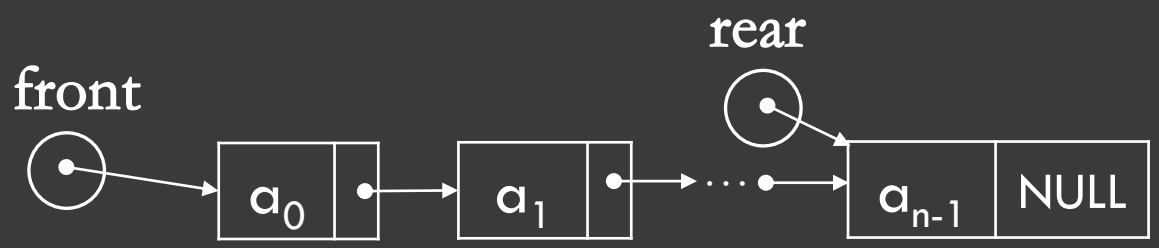
```
BOOL DeQueue(Queue *Q)
{
    if (IsEmpty(Q))  return FALSE;//空队列处理
    Q->front=(Q->front+1)%Q->maxSize;
    return TRUE;
}
```

```
BOOL Front(Queue *Q, ElemType *x)
{
    if(IsEmpty(Q))    return FALSE;   //空队列处理
    *x=Q->element[(Q->front+1)%Q->maxSize];
    return TRUE;
}
```

```
void Destroy(Queue *Q)
{
    free(Q->element);
    free(Q)
}
```

目录

队列的链式存储表示




```
typedef struct queue{
        Node* front,*rear;
} Queue;
```

目录

递归与递归程序设计

直接递归：函数的实现过程中出现了对自己本身的调用

间接递归：函数调用形成了一个环状调用链

- 函数A的实现过程中调用的B，而B的实现过程中又调用了A

递归与递归程序设计

斐波那契级数

$$\begin{cases} F_0=0 \\ F_1=1 \\ F_h=F_{h-1}+F_{h-2} \end{cases}$$

$$F_h=\frac{(\frac{1+\sqrt{5}}{2})^h-(\frac{1-\sqrt{5}}{2})^h}{\sqrt{5}}$$

黄金分割级数

F_{h-1}/F_{h-2} 无限接近 $(1+\sqrt{5})/2 \approx 0.618$

递归与递归程序设计

```
long Fib(long n)
{
    if(n<=1) return n;
    return Fib(n-2)+Fib(n-1);
}
```

递归出口：定义了递归的停止条件，出口可以不止一个

子问题划分

（1）子问题结构与原问题相同

（2）子问题规模比原问题小

（3）子问题通过一定形式修改参数调用自身函数

（4）若干子问题的解以一定方式组成原问题的解

递归与递归程序设计

```
long Fib(long n)
{
    if(n<=1) return n;
    return Fib(n-2)+Fib(n-1);
}
```

优点：程序简洁清晰，易于分析
缺点：费时、费空间

递归程序执行过程分析

系统栈：程序运行时存放函数调用与返回所需各种数据

工作记录：

- 函数调用执行完成时的返回地址：即上层中本次调用的语句的下一条语句或指令地址
- 每次函数调用的实参与局部变量参数

每进入一层递归就产生一个新的工作记录压入系统栈顶，每完成一次递归调用就从栈顶弹出一个工作记录

递归程序执行过程分析

递归过程演示

递归程序到非递归程序

WHY?

- 递归程序空间需求和时间需求都较高

HOW?

- 采用循环方法解决

递归程序到非递归程序

```
long Fib(long n)
{
    if(n<=1) return n;
    return Fib(n-2)+Fib(n-1);
}
```

```
long Fib(long n)
{
    if(n<=1) return n;
    long f_a = 0;
    long f_b = 1;
    long m = 0;
    for(int i =2;i<=n;i++)
    {m = f_a+f_b;
     f_a = f_b;
     f_b = m;}
     return m;
}
```

中缀表达式转换成后缀表达式

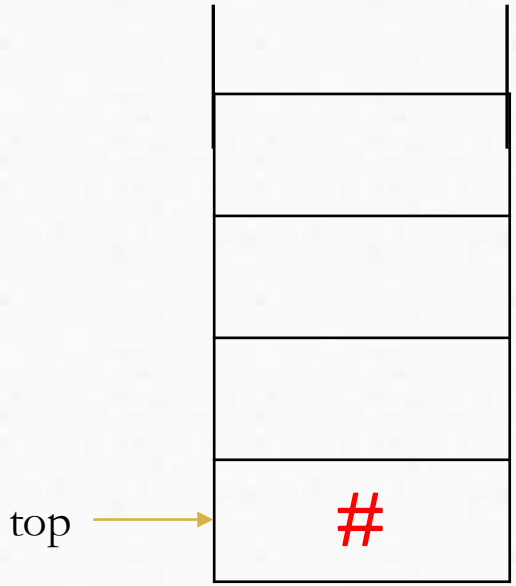
① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #



中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

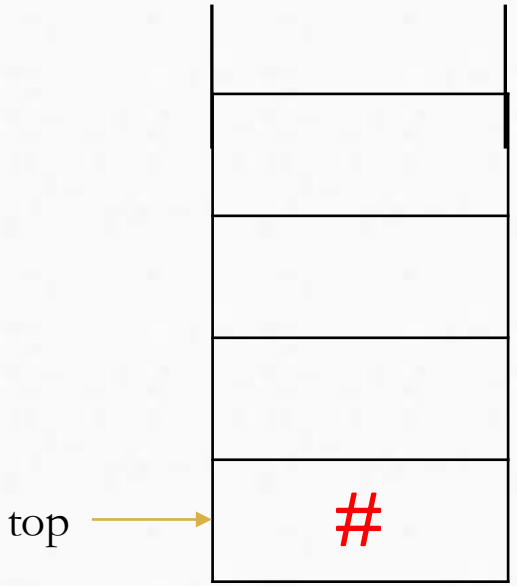
③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a/(b-c)+d*e#

输出

a



中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

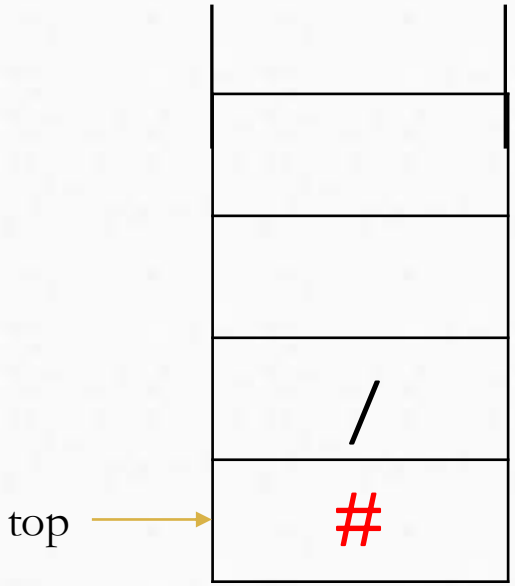
③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #

输出

a



中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

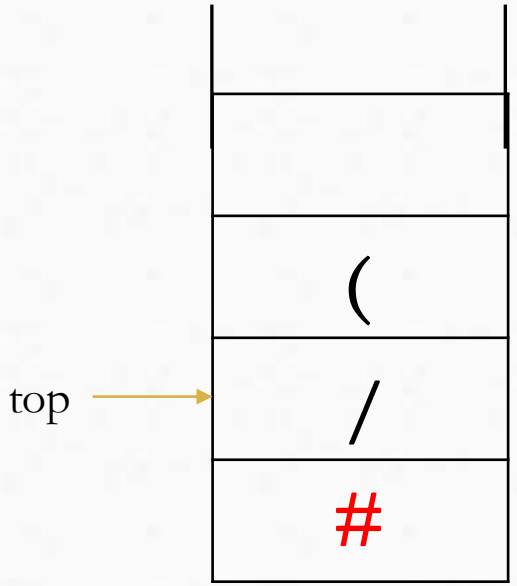
③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #

输出

a



中缀表达式转换成后缀表达式

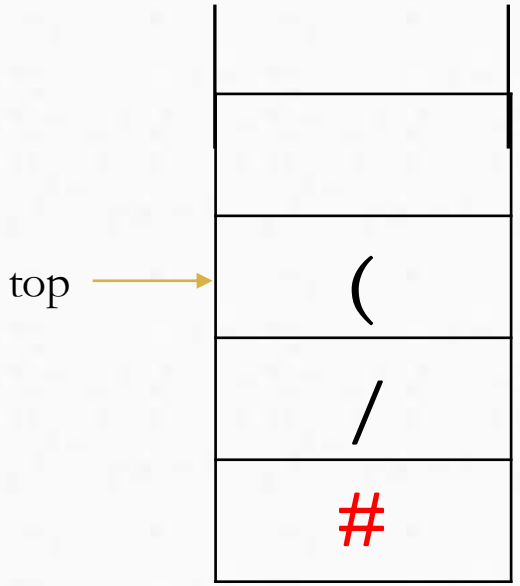
① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #



输出

a

b

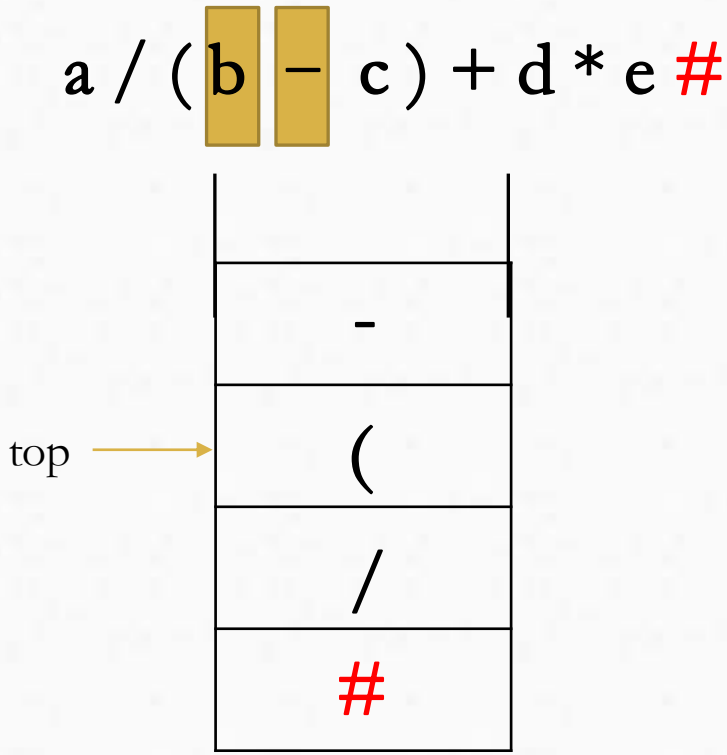
中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7



输出

a

b

中缀表达式转换成后缀表达式

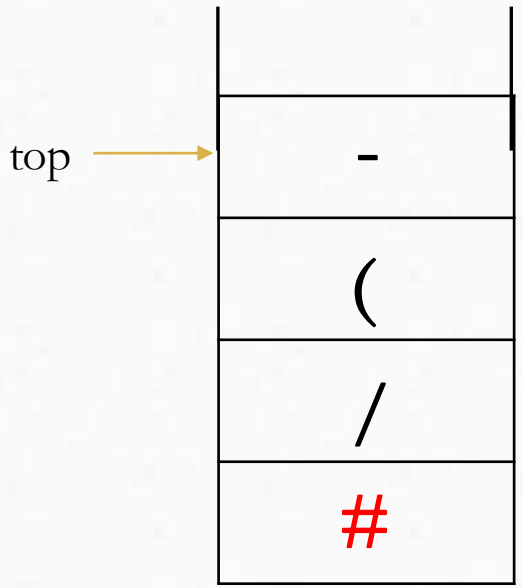
① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b − c) + d * e #



输出

a

b

c

中缀表达式转换成后缀表达式

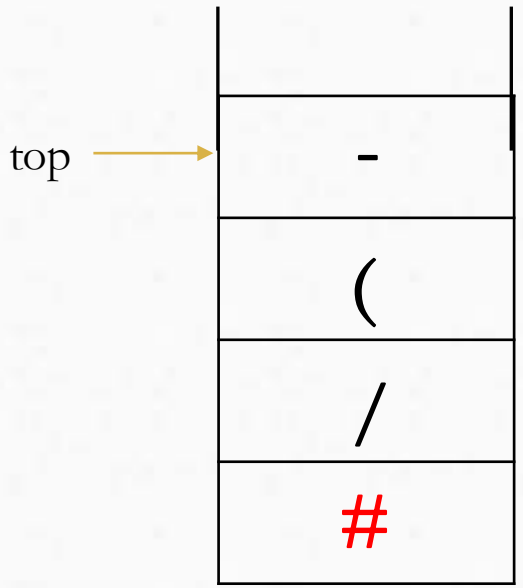
① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #



输出

a

b

c

中缀表达式转换成后缀表达式

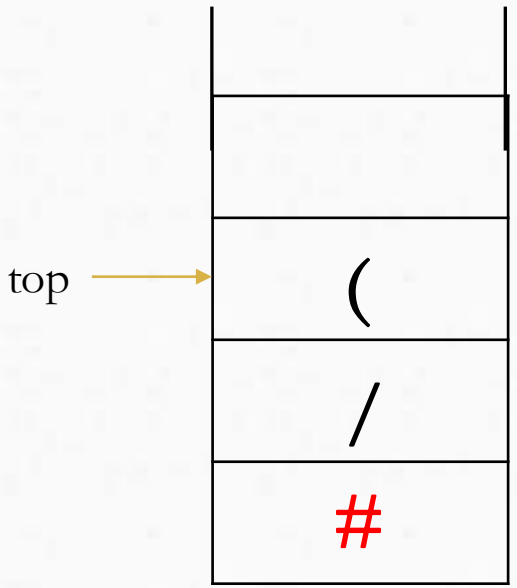
① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #



输出

a

b

c

-

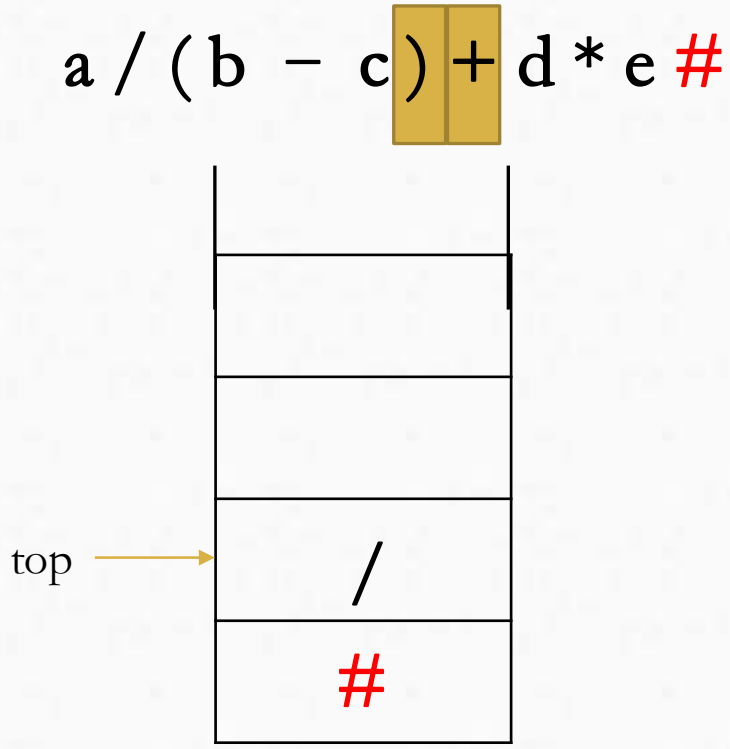
中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7



输出

a

b

c

-

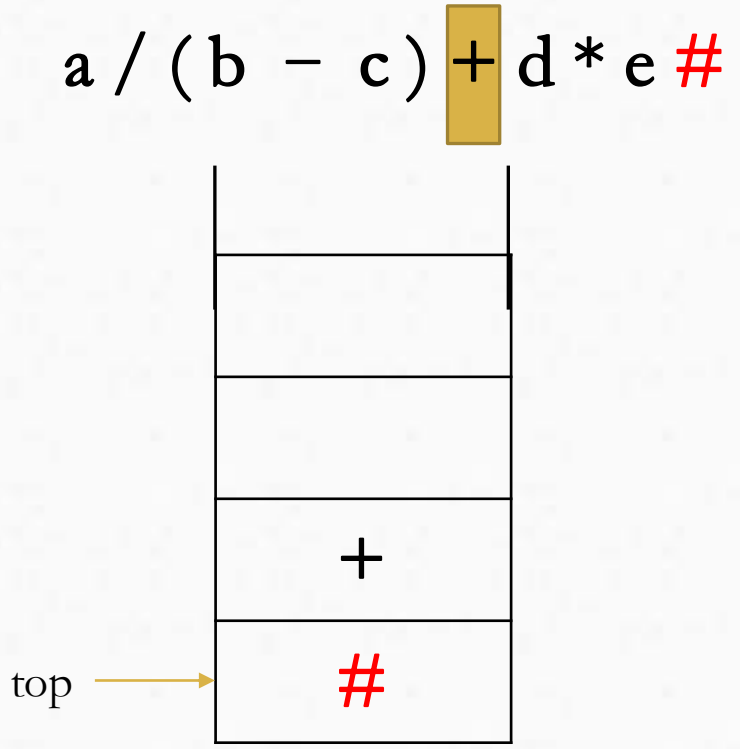
中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7



输出

a

b

c

-

/

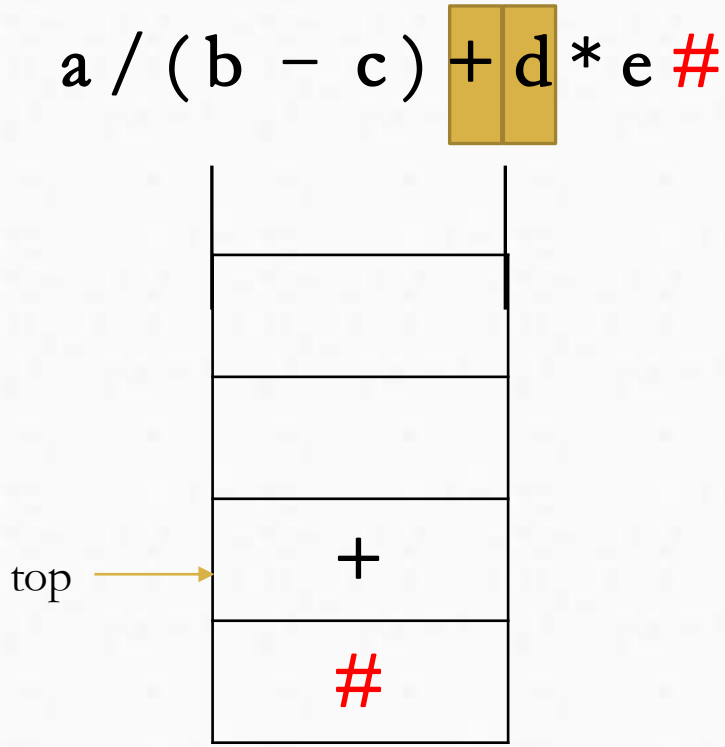
中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7



输出

a

b

c

-

/

d

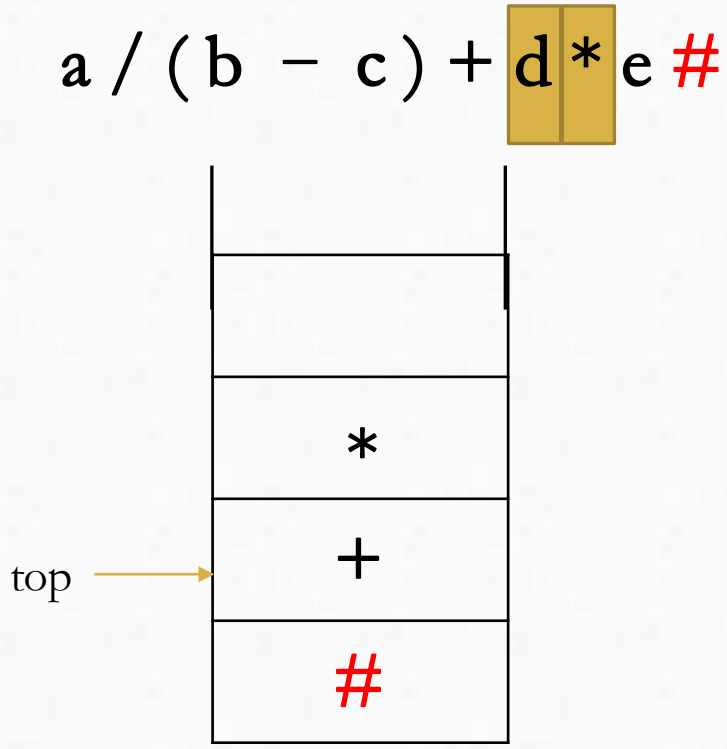
中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7



输出

a

b

c

-

/

d

中缀表达式转换成后缀表达式

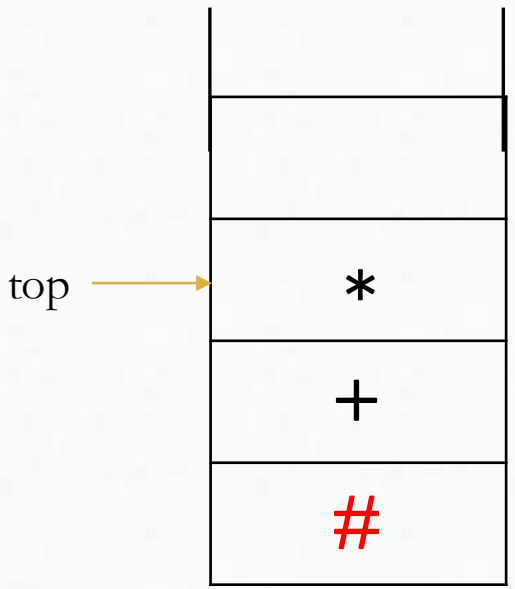
① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7

a / (b - c) + d * e #



输出

a

b

c

-

/

d

e

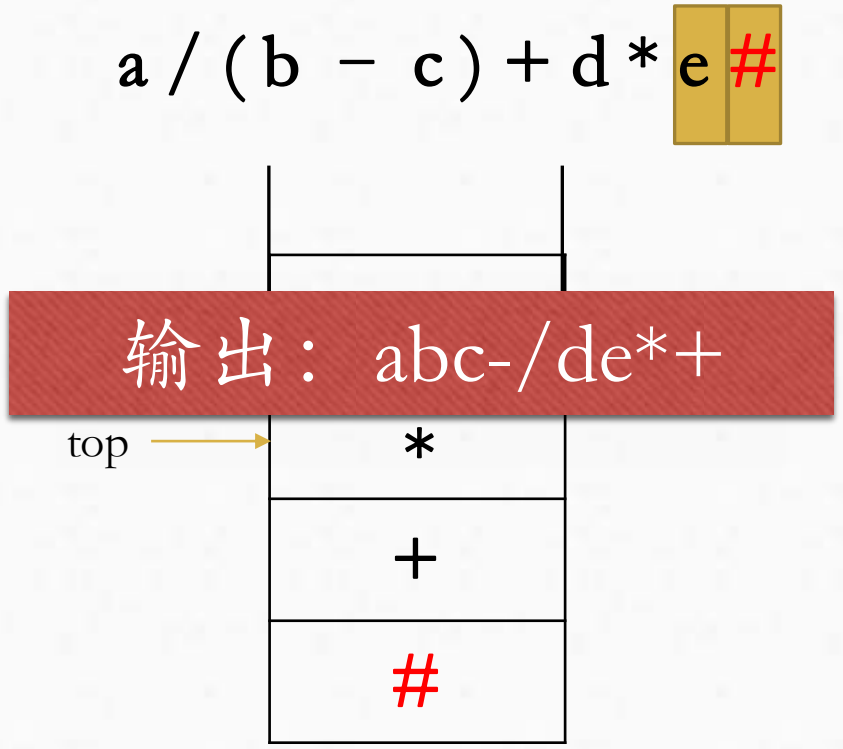
中缀表达式转换成后缀表达式

① 遇到操作数直接输出；

② 遇到“)"，则连续出栈，直到“("为止

③ 遇到其它操作符，与栈顶的操作符比较优先级；若优先级<=栈顶的优先级，则连续出栈，直到>栈顶，操作符进栈

操作符	#	(	*/	+-	)
icp(外)	0	7	4	2	1
isp(内)	0	1	5	3	7



输出

a

b

c

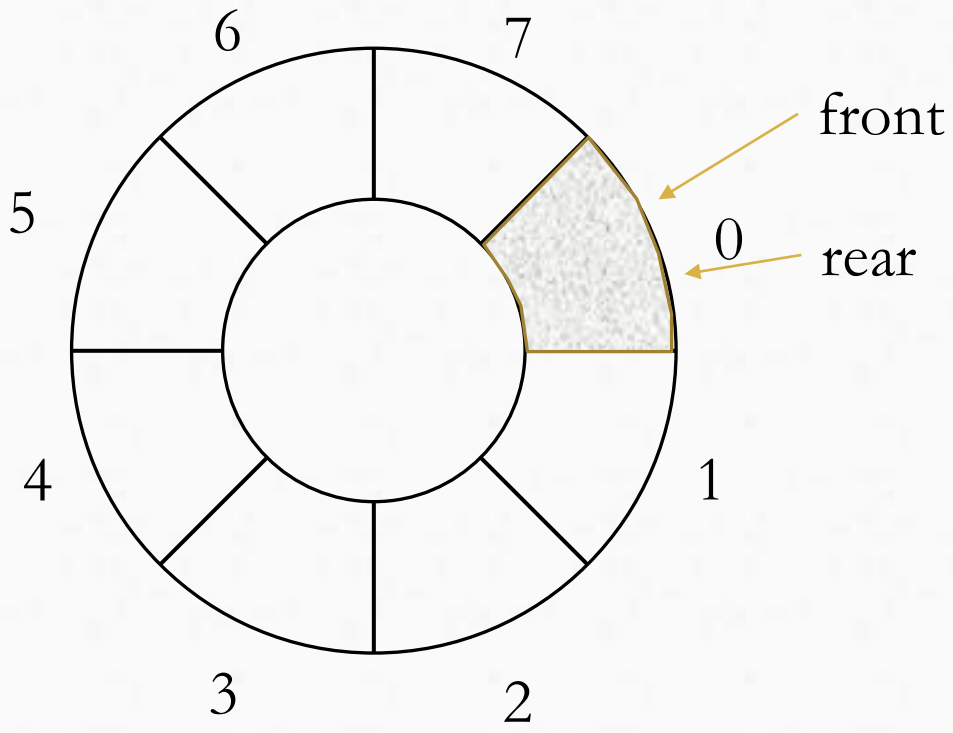
-

/

d

e

队列的顺序存储表示



初始时：

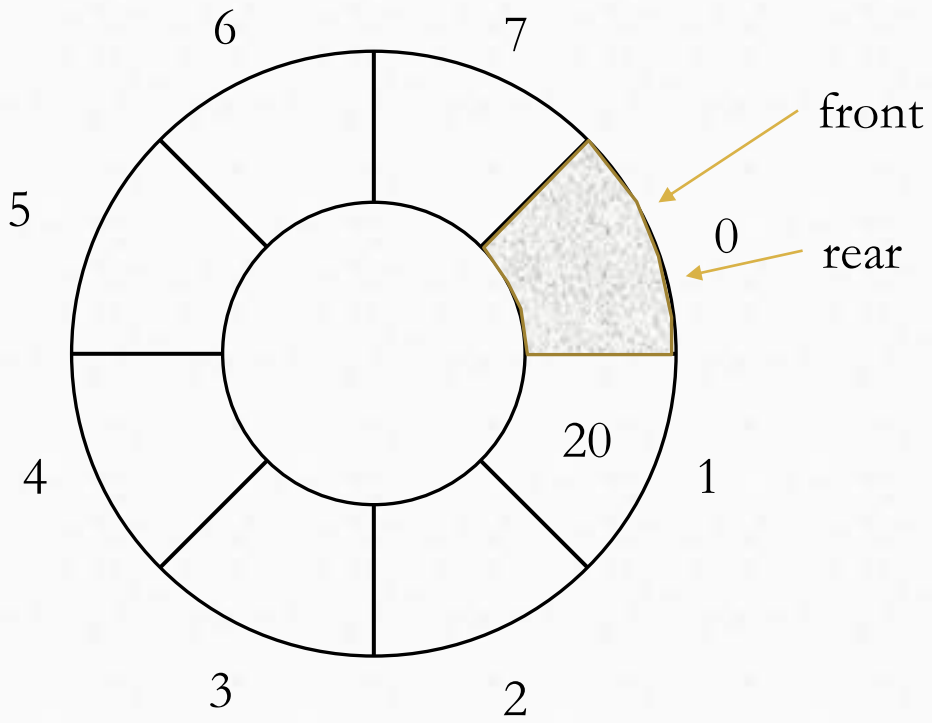
front = rear = 0

指针前进

front = (front+1)%maxQueue

rear = (rear+1)%maxQueue

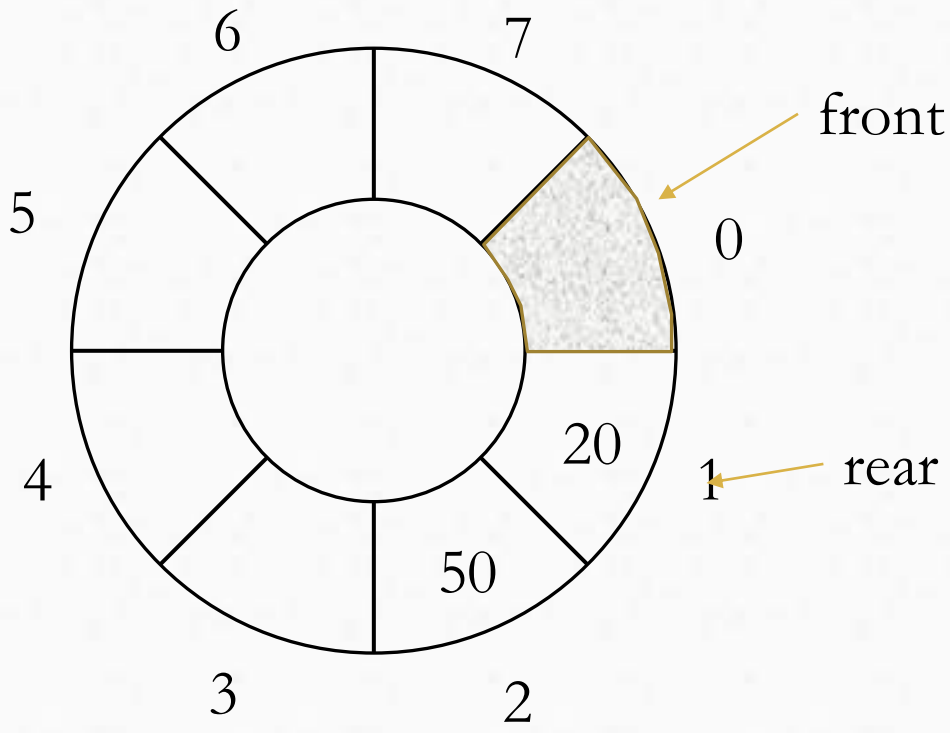
队列的顺序存储表示



先前进、后赋值
rear = (rear+1)%maxQueue

20 入队

队列的顺序存储表示



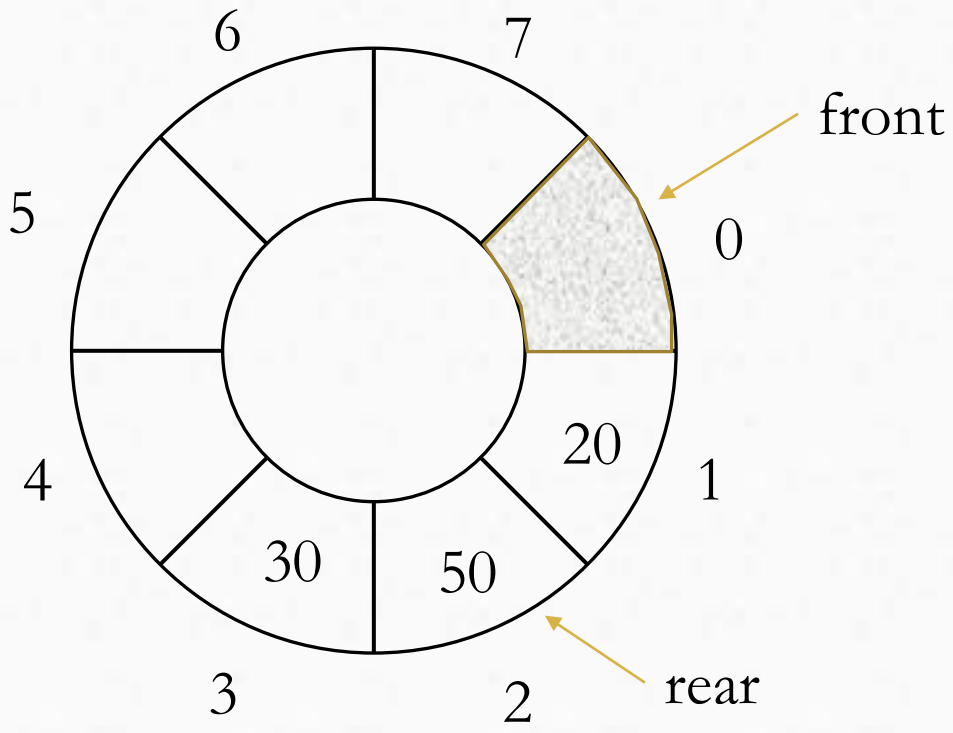
先前进、后赋值

rear = (rear+1)%maxQueue

20 入队

50 入队

队列的顺序存储表示



先前进、后赋值

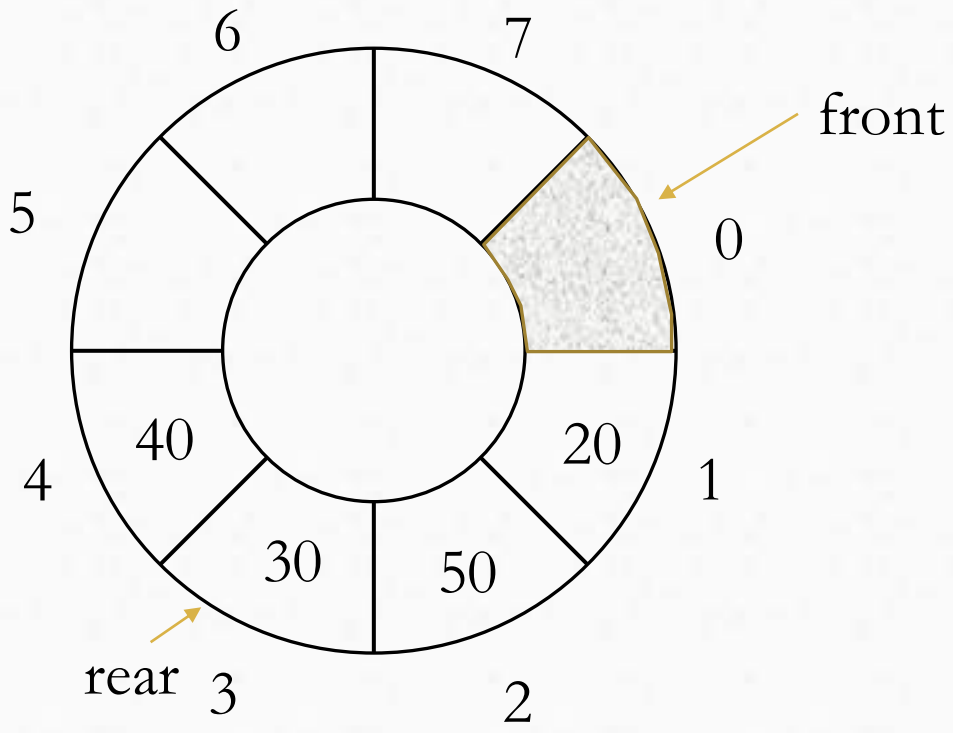
rear = (rear+1)%maxQueue

20 入队

50 入队

30 入队

队列的顺序存储表示



先前进、后赋值

rear = (rear+1)%maxQueue

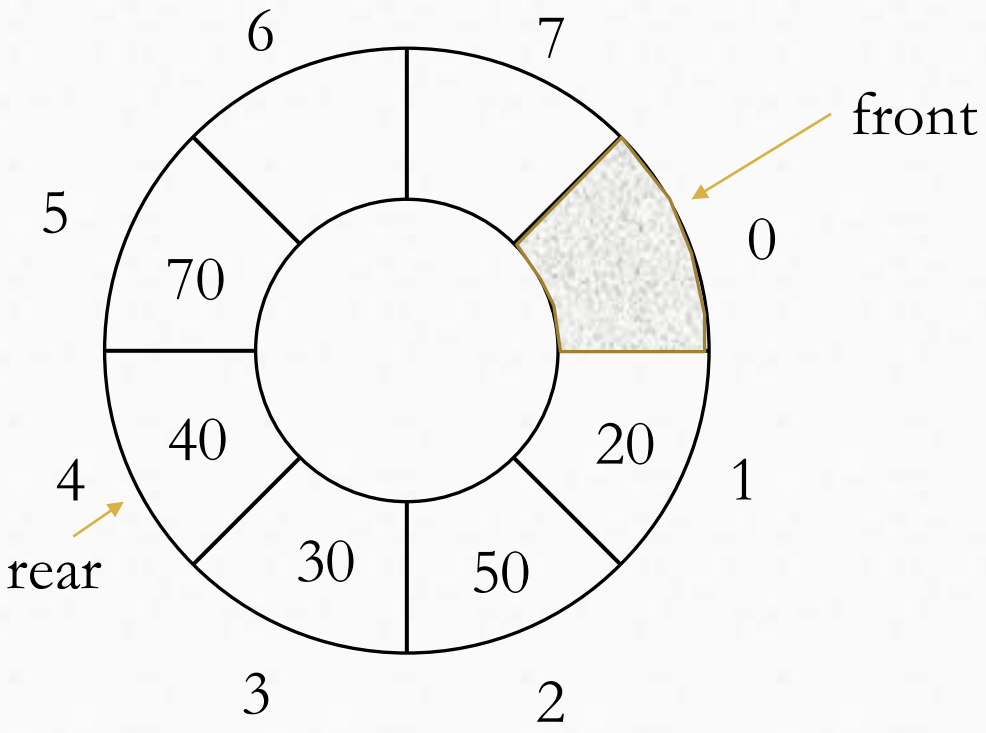
20 入队

50 入队

30 入队

40 入队

队列的顺序存储表示



先前进、后赋值

rear = (rear+1)%maxQueue

20 入队

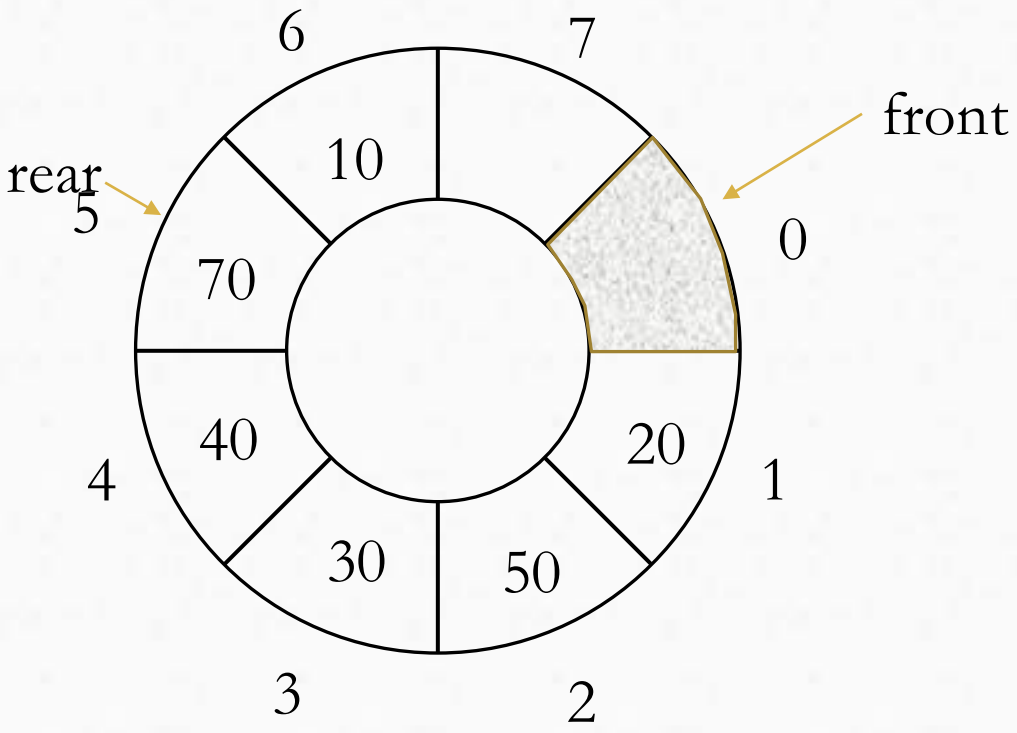
50 入队

30 入队

40 入队

70 入队

队列的顺序存储表示



先前进、后赋值

rear = (rear+1)%maxQueue

20 入队

50 入队

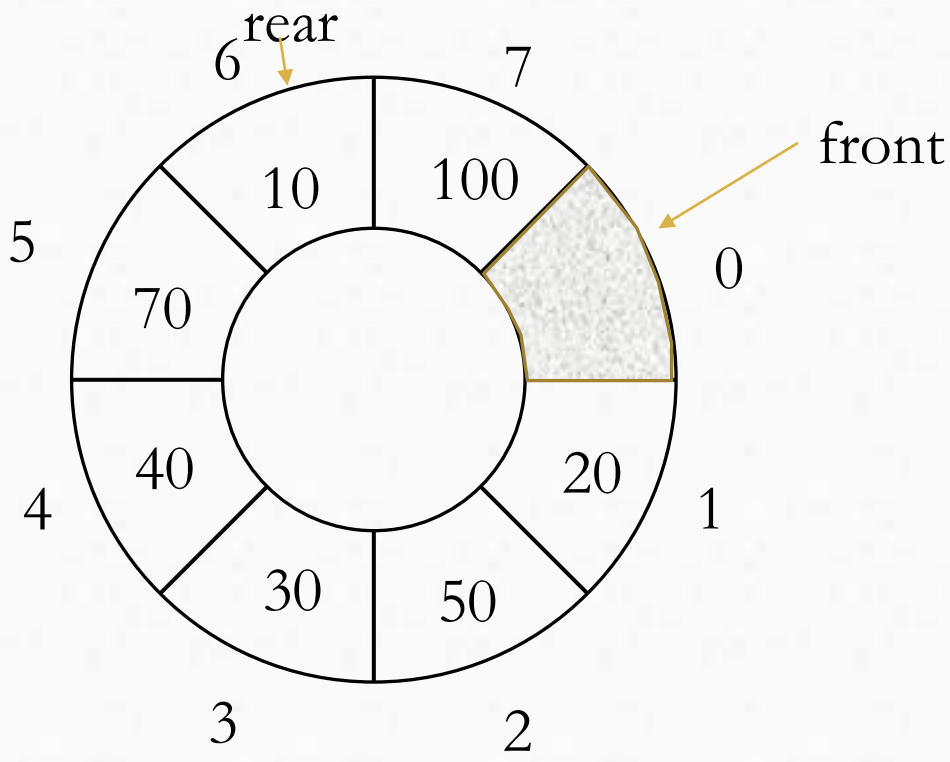
30 入队

40 入队

70 入队

10 入队

队列的顺序存储表示



先前进、后赋值

rear = (rear+1)%maxQueue

队列满的判断条件

(rear+1)%maxQueue == front

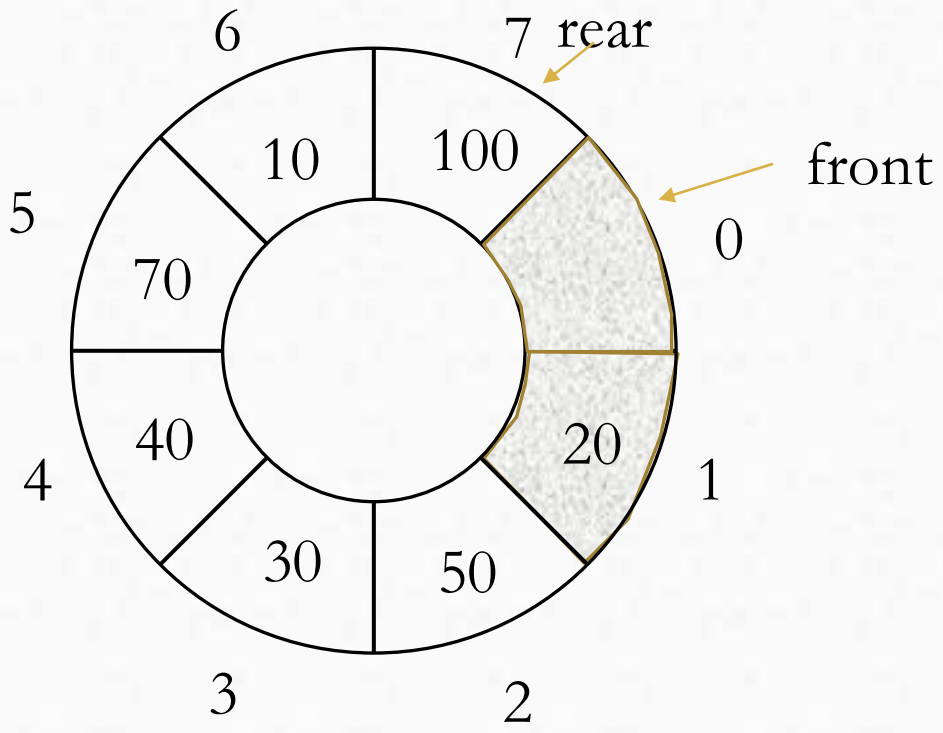
rear == 7

maxQueue == 8

front == 0

满足队列满的判断条件

队列的顺序存储表示

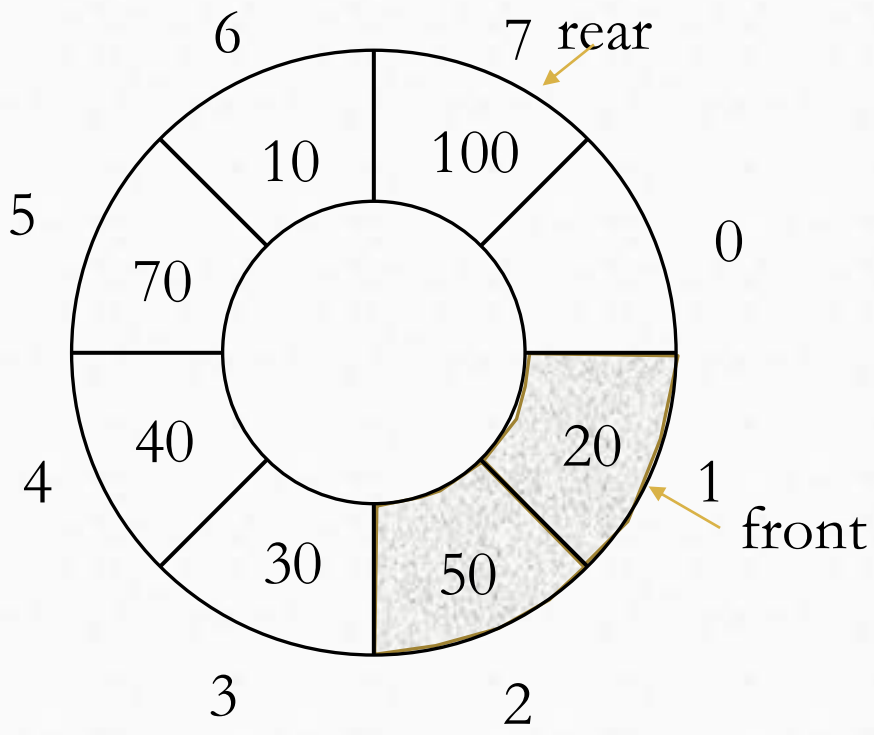


front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

20 出队

此时(rear+1)%maxQueue = 0
front == 1
队列不满，0号位置可用

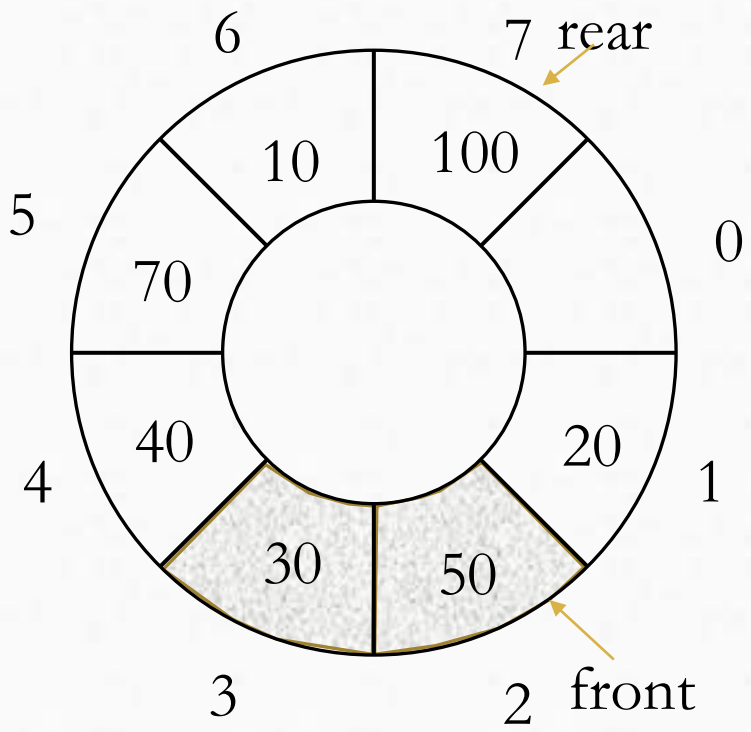
队列的顺序存储表示



front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

20 出队
50 出队

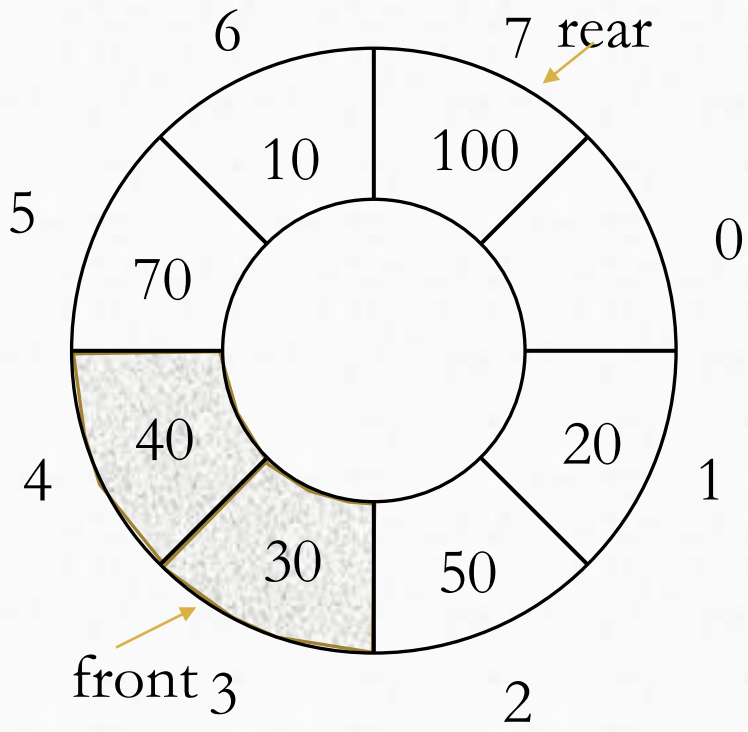
队列的顺序存储表示



front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

20 出队
50 出队
30 出队

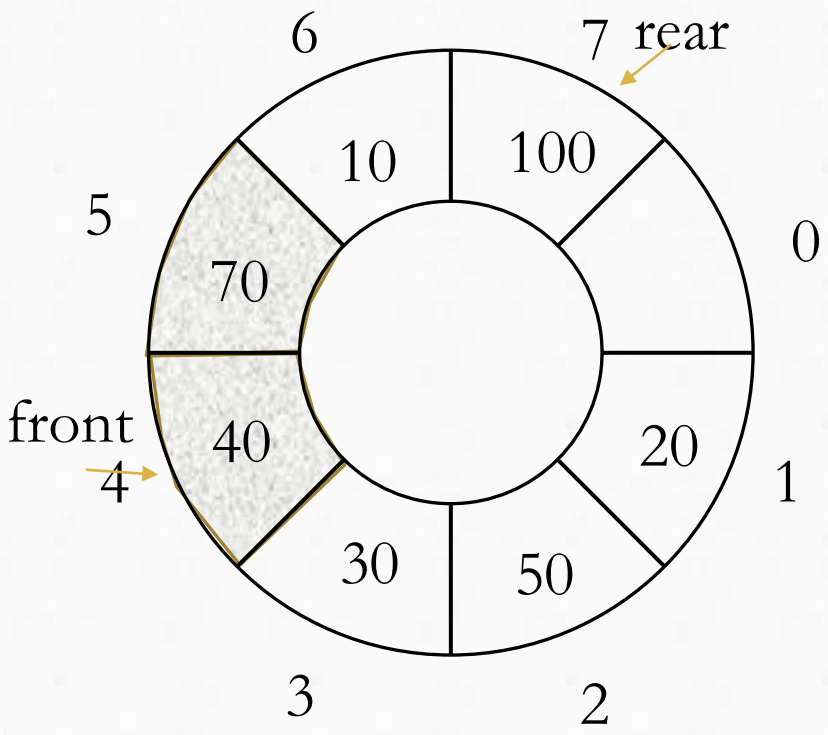
队列的顺序存储表示



front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

20 出队
50 出队
30 出队
40 出队

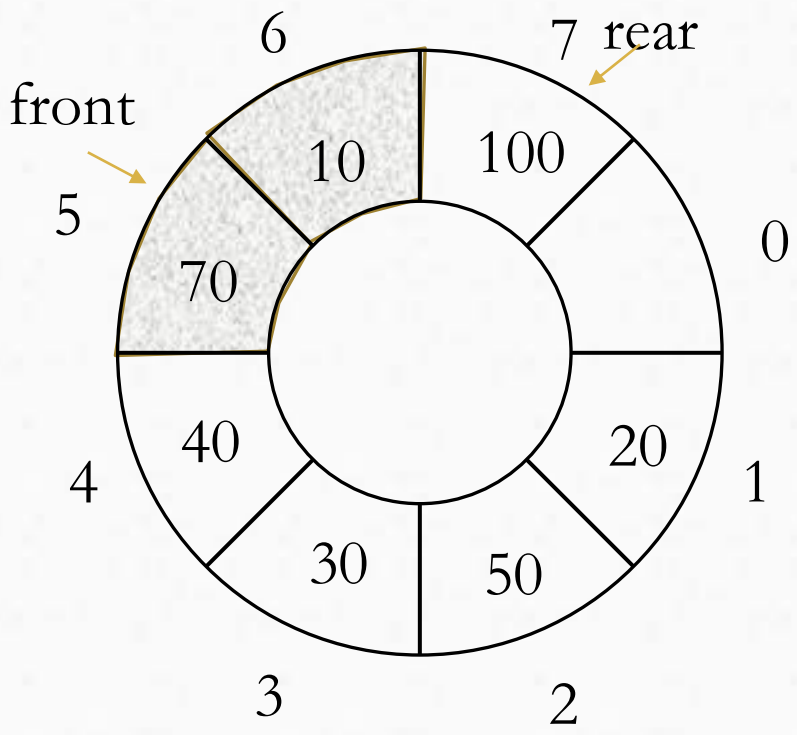
队列的顺序存储表示



front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

20 出队
50 出队
30 出队
40 出队
70 出队

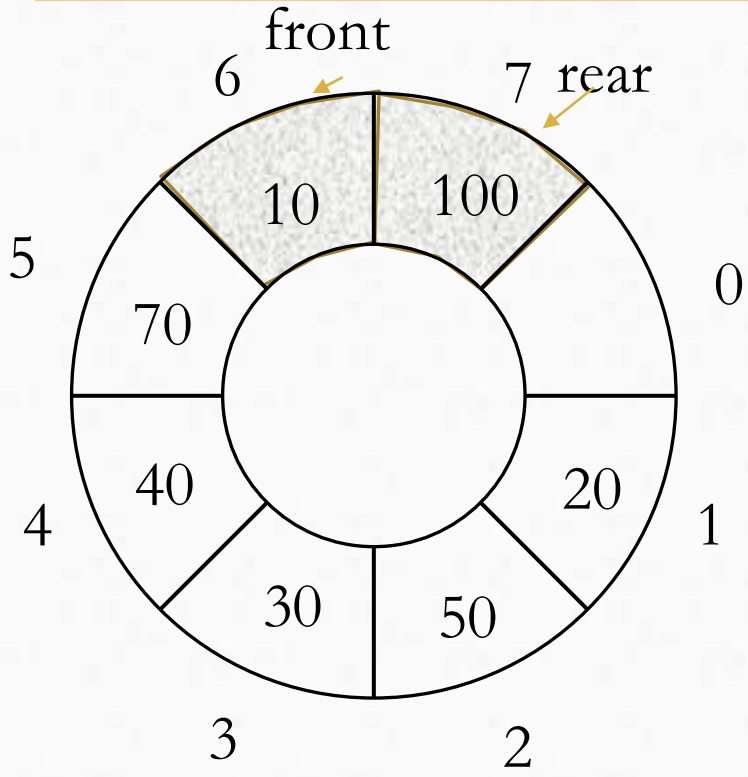
队列的顺序存储表示



front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

20 出队
50 出队
30 出队
40 出队
70 出队
10 出队

队列的顺序存储表示

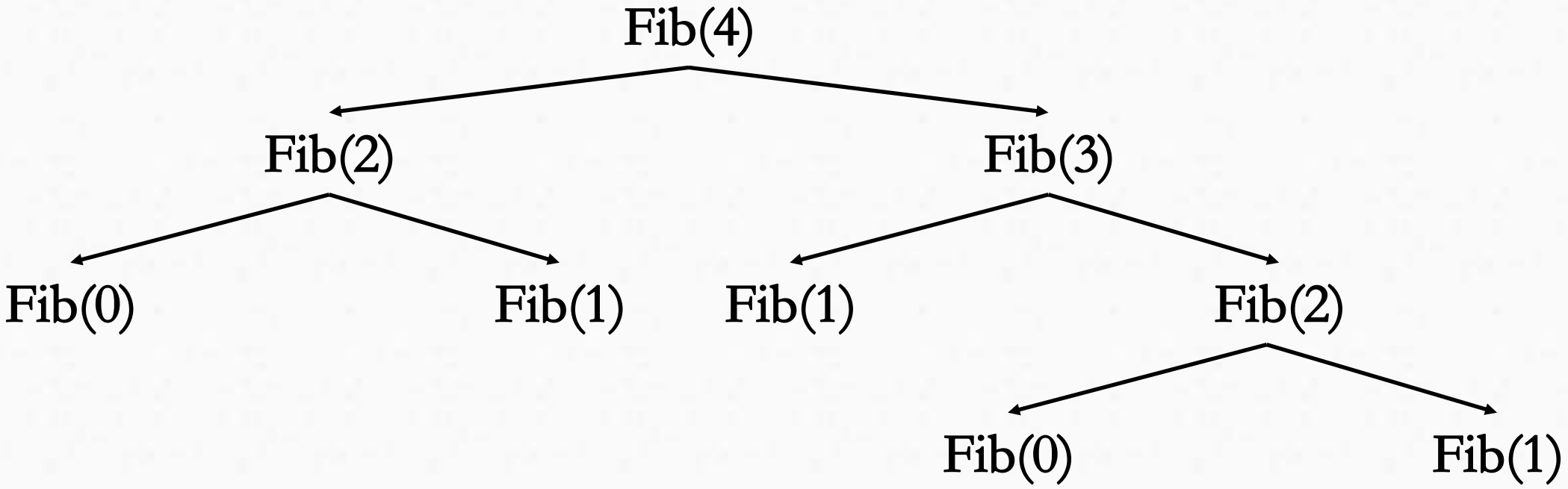


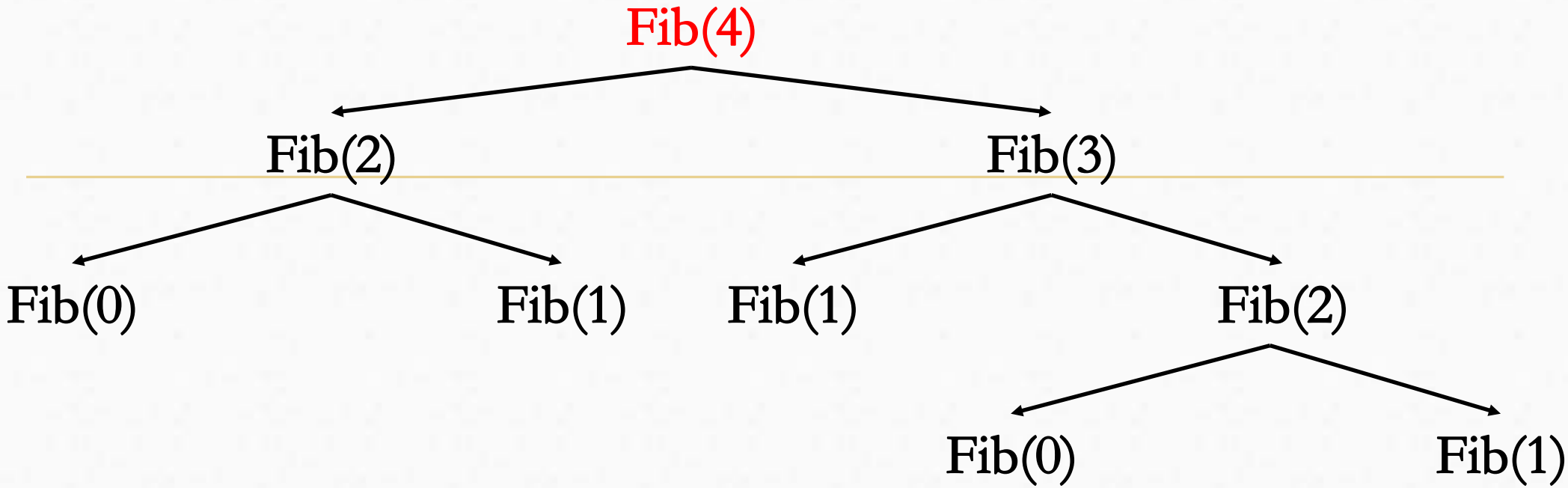
front指向的空间不可用
rear与front之间的空间可用
front = (front+1)%maxQueue

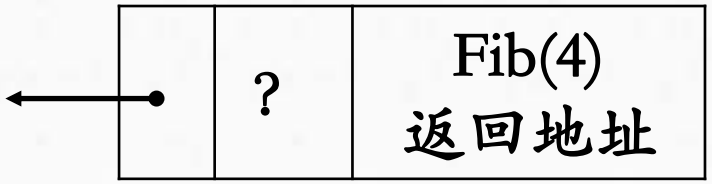
队列空的判断条件
rear == front

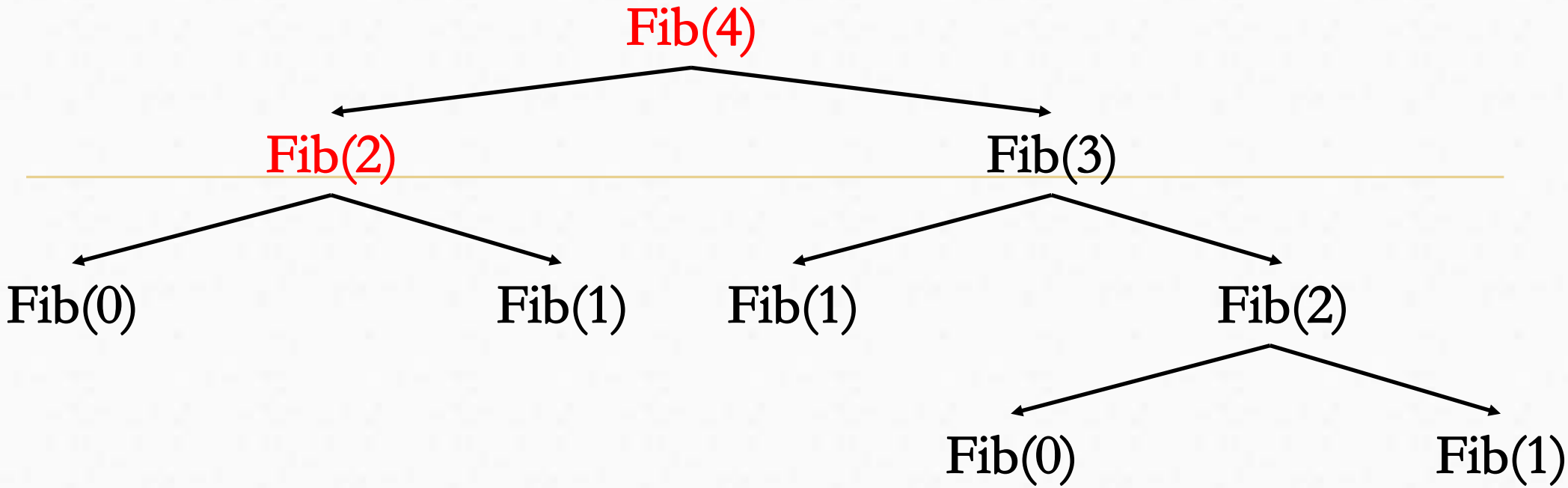
rear == 7
front == 7
满足队列空的判断条件

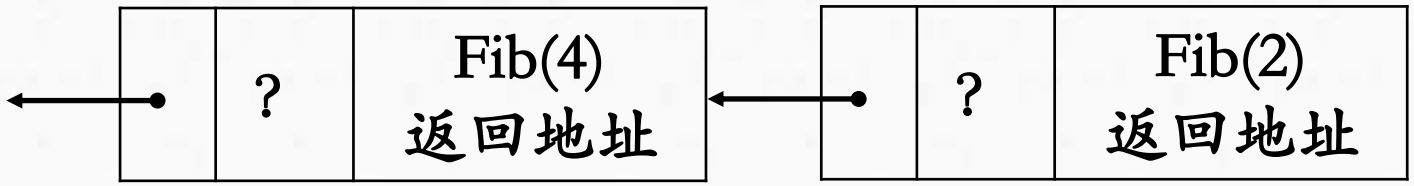
递归程序执行过程分析

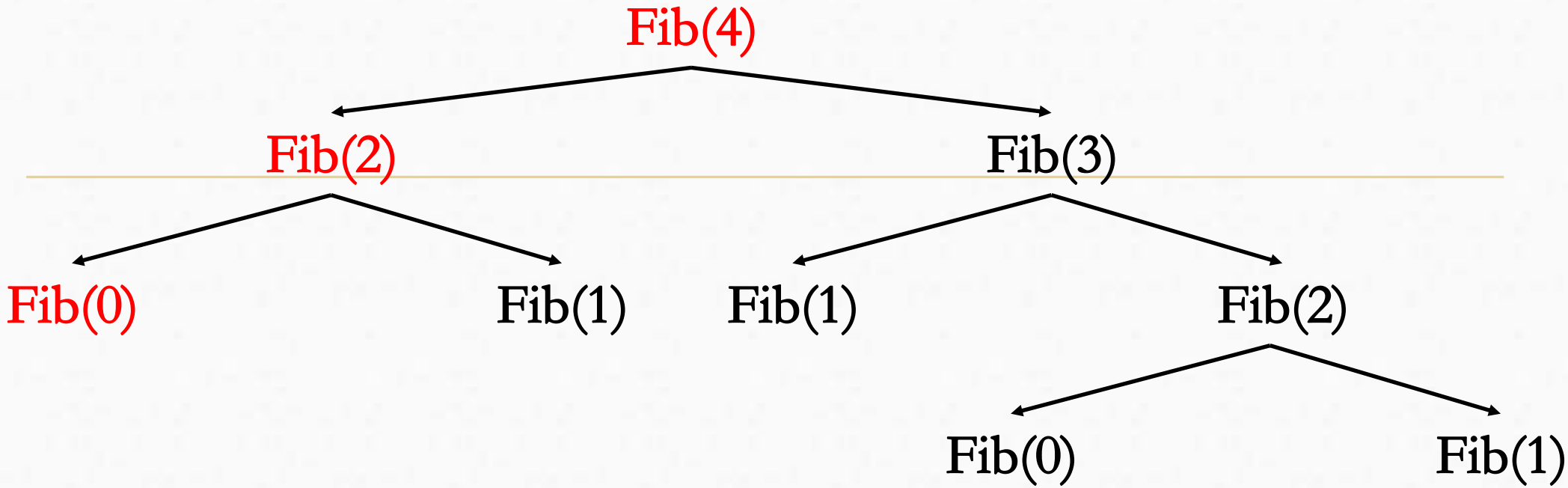


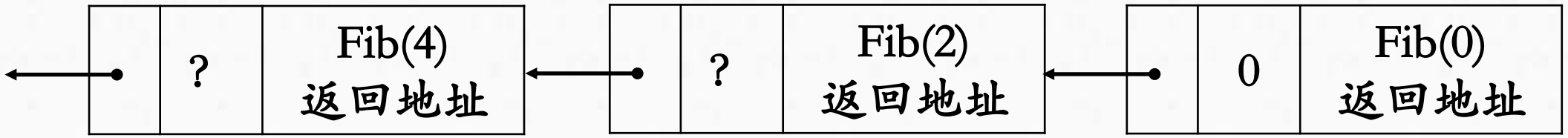

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

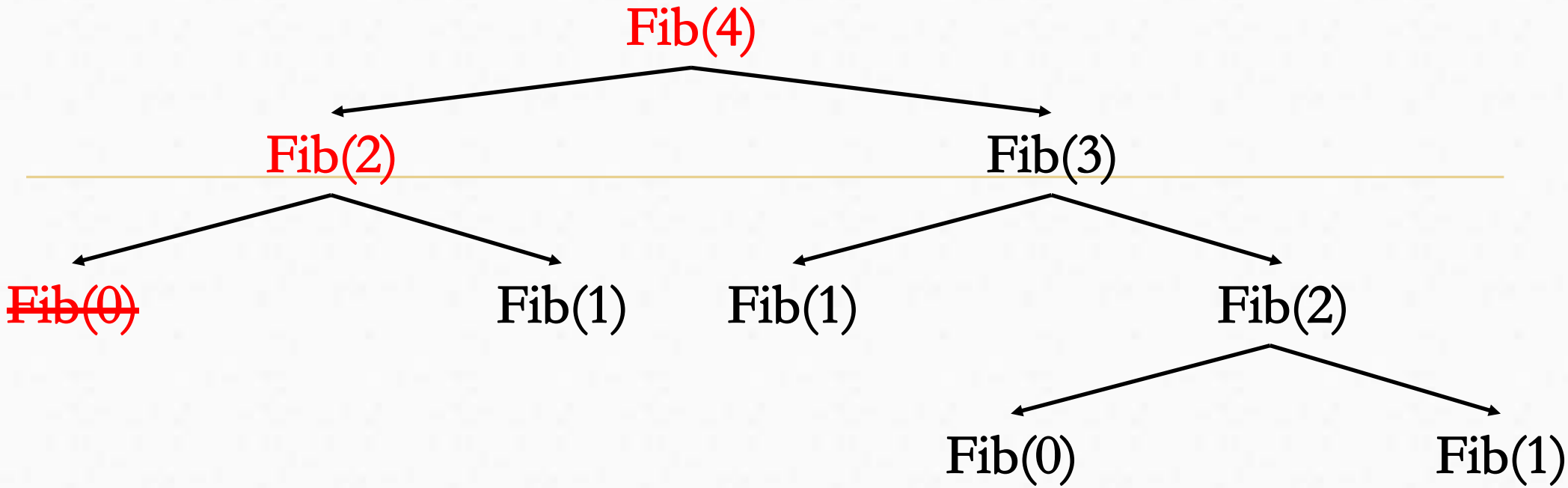

?
Fib(4)
返回地址

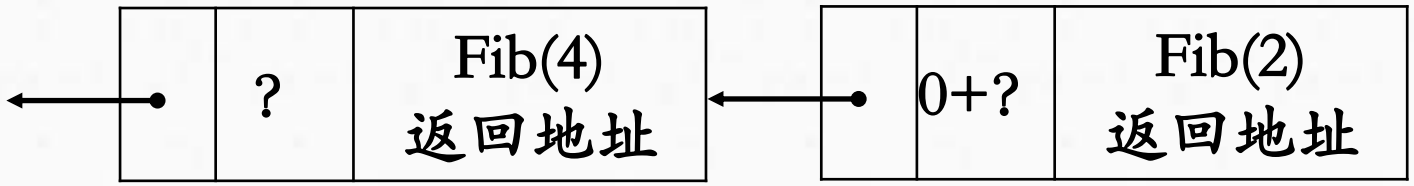

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

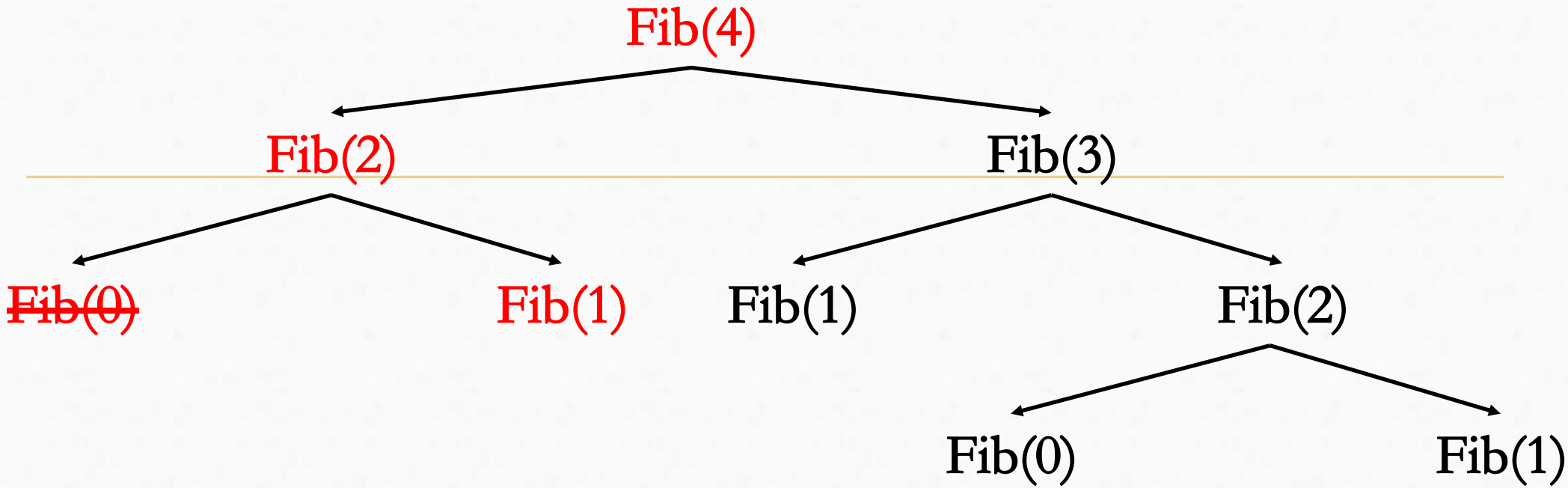

?
Fib(4)
返回地址
?
Fib(2)
返回地址

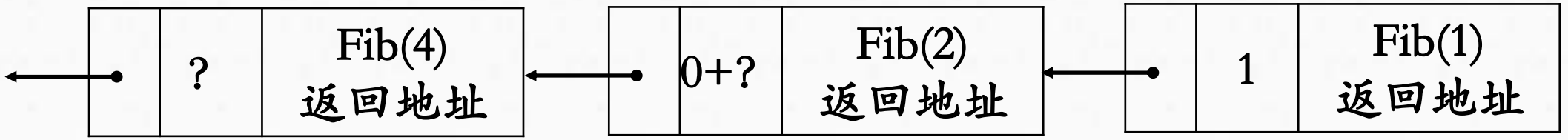

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

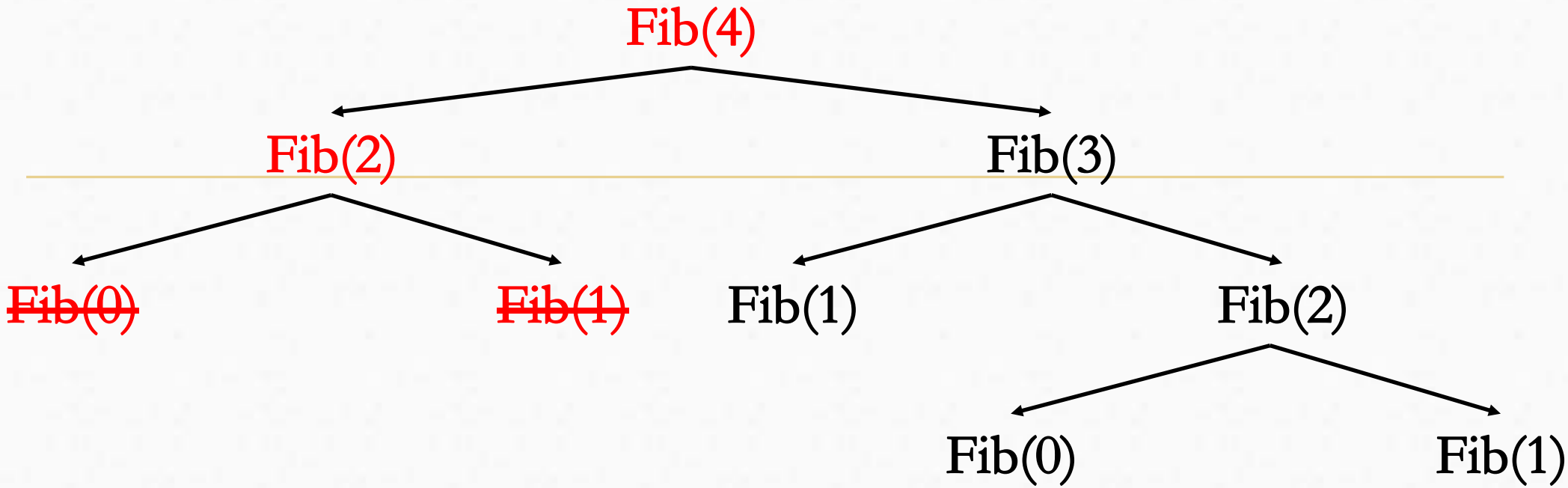

?
Fib(4)
返回地址
?
Fib(2)
返回地址
0
Fib(0)
返回地址

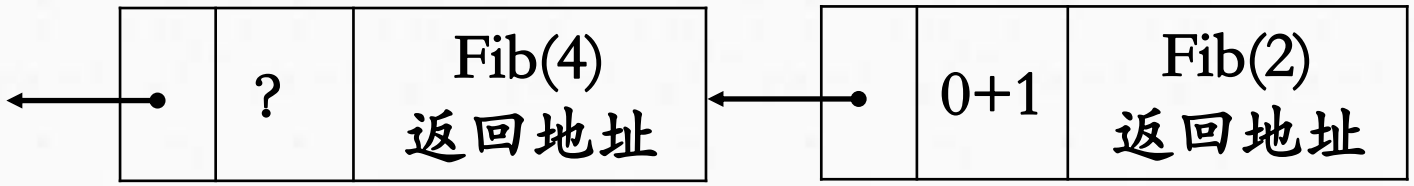

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

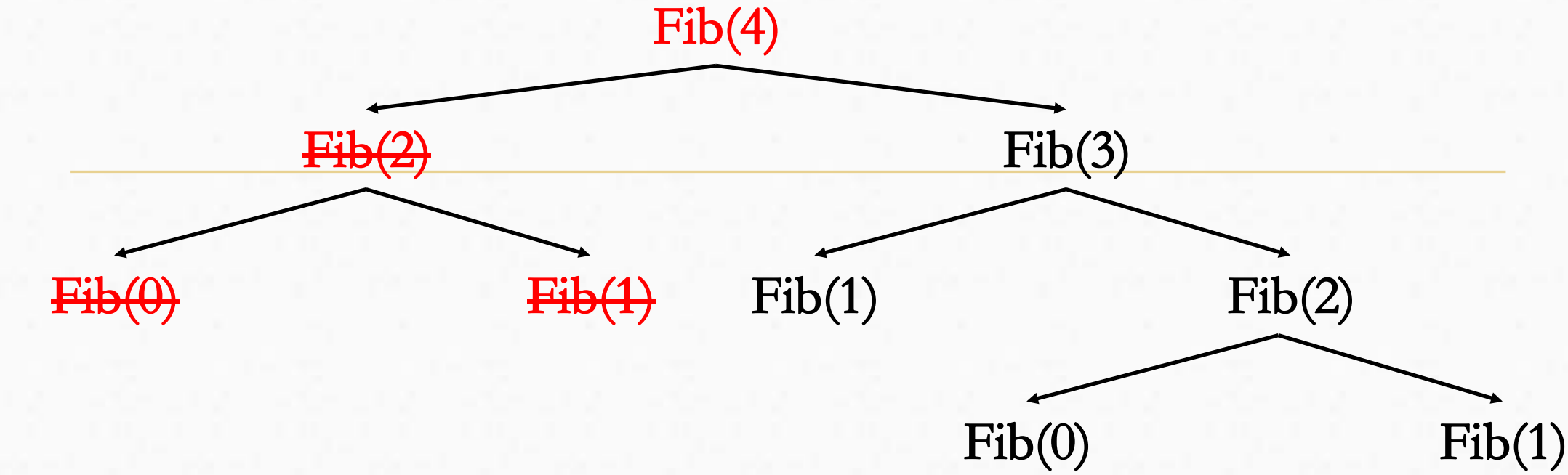

?
Fib(4)
返回地址
0+?
Fib(2)
返回地址

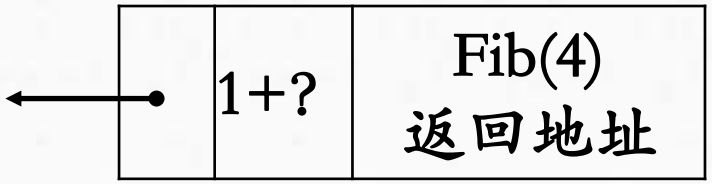

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

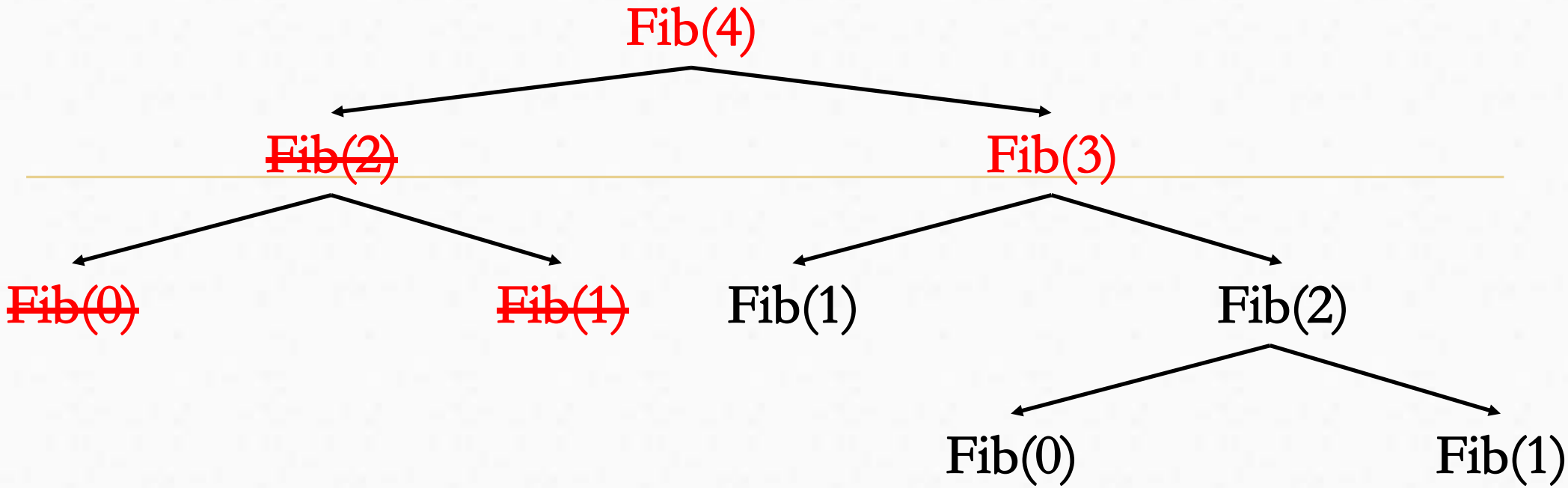

?
Fib(4)
返回地址
0+?
Fib(2)
返回地址
1
Fib(1)
返回地址

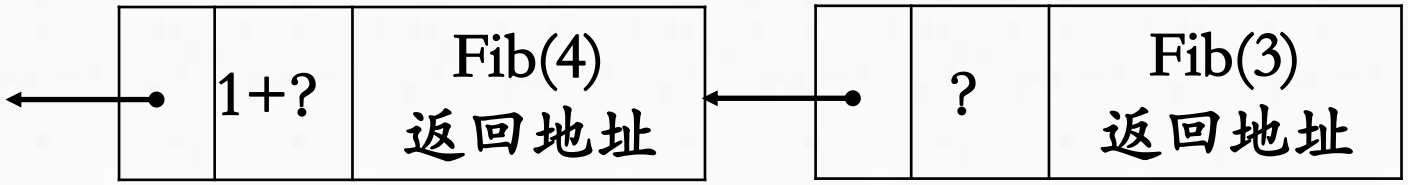

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

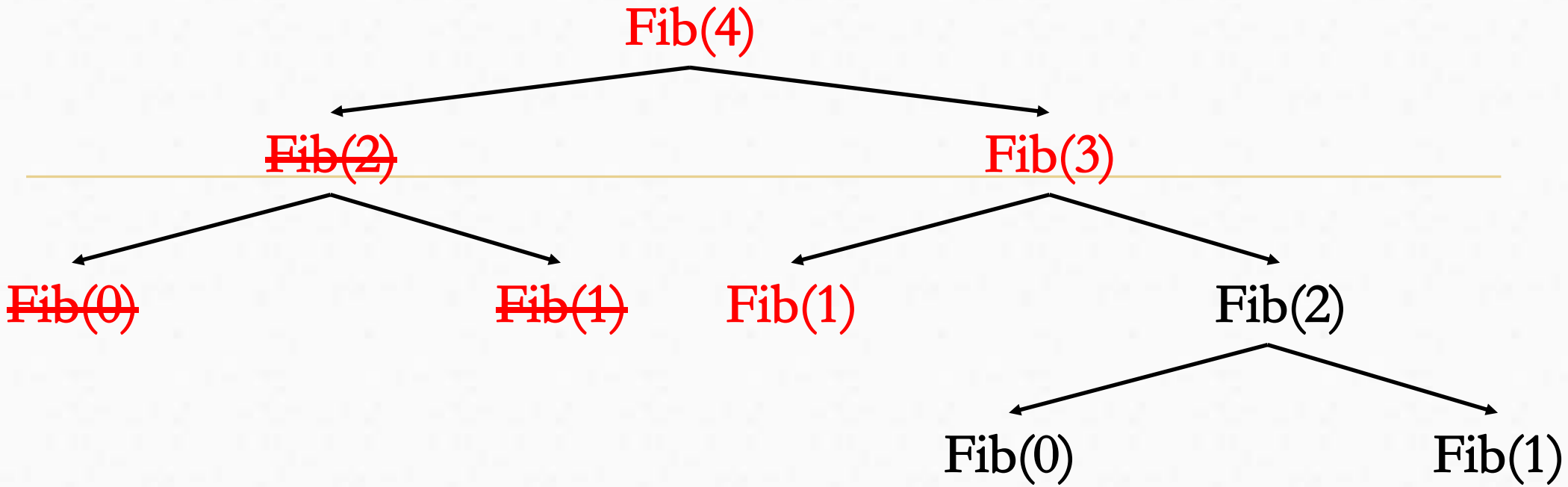

?
Fib(4)
返回地址
0+1
Fib(2)
返回地址

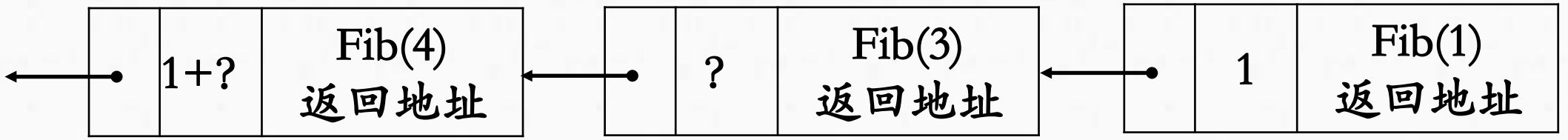

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

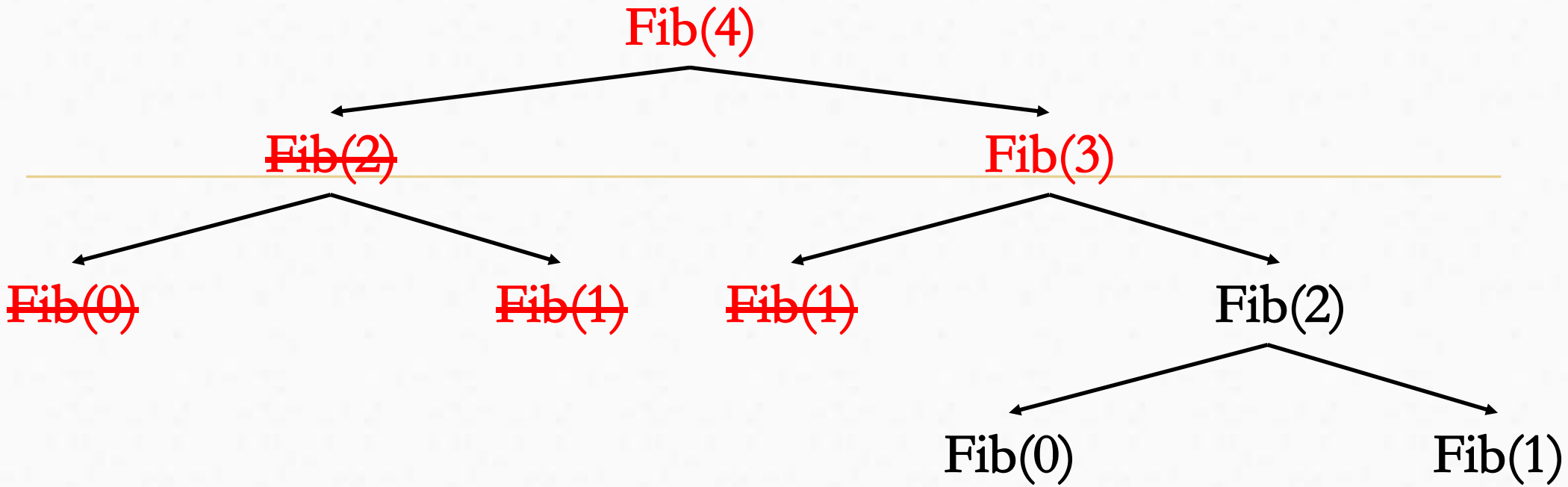

1+?
Fib(4)
返回地址

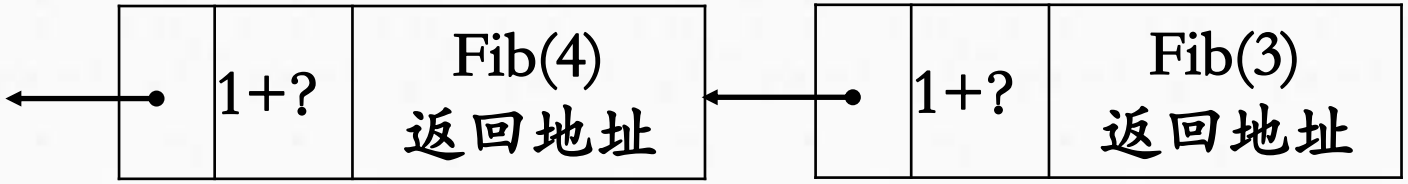

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

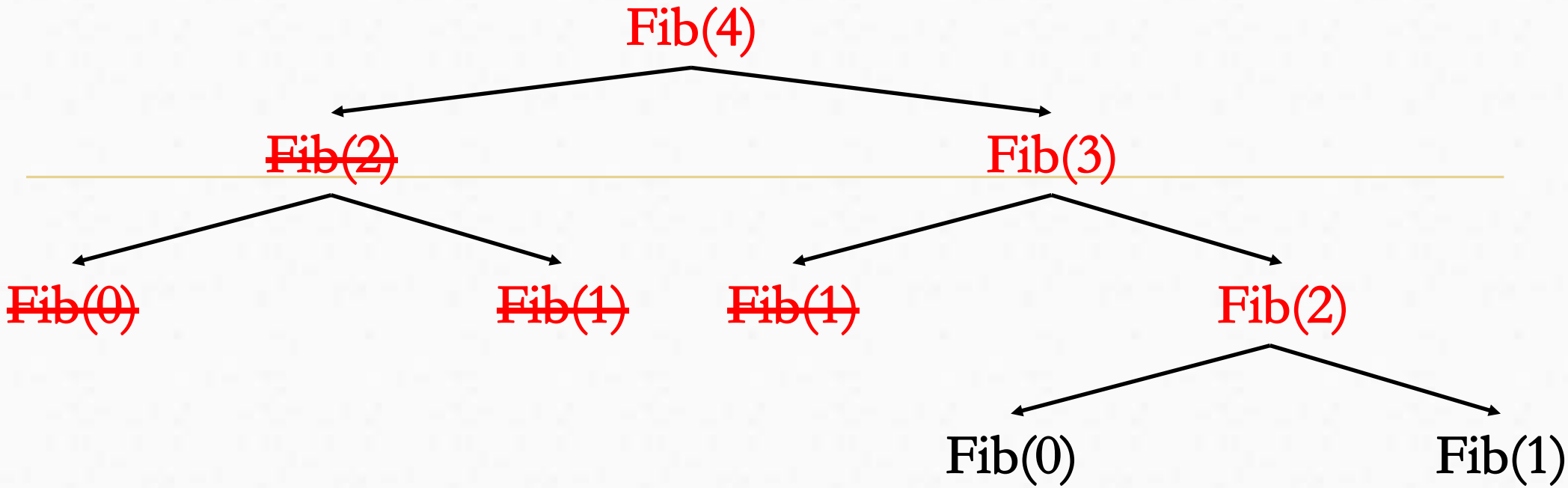

1+?
Fib(4)
返回地址
?
Fib(3)
返回地址

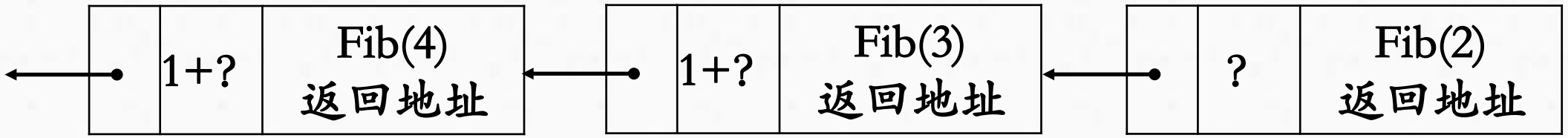

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

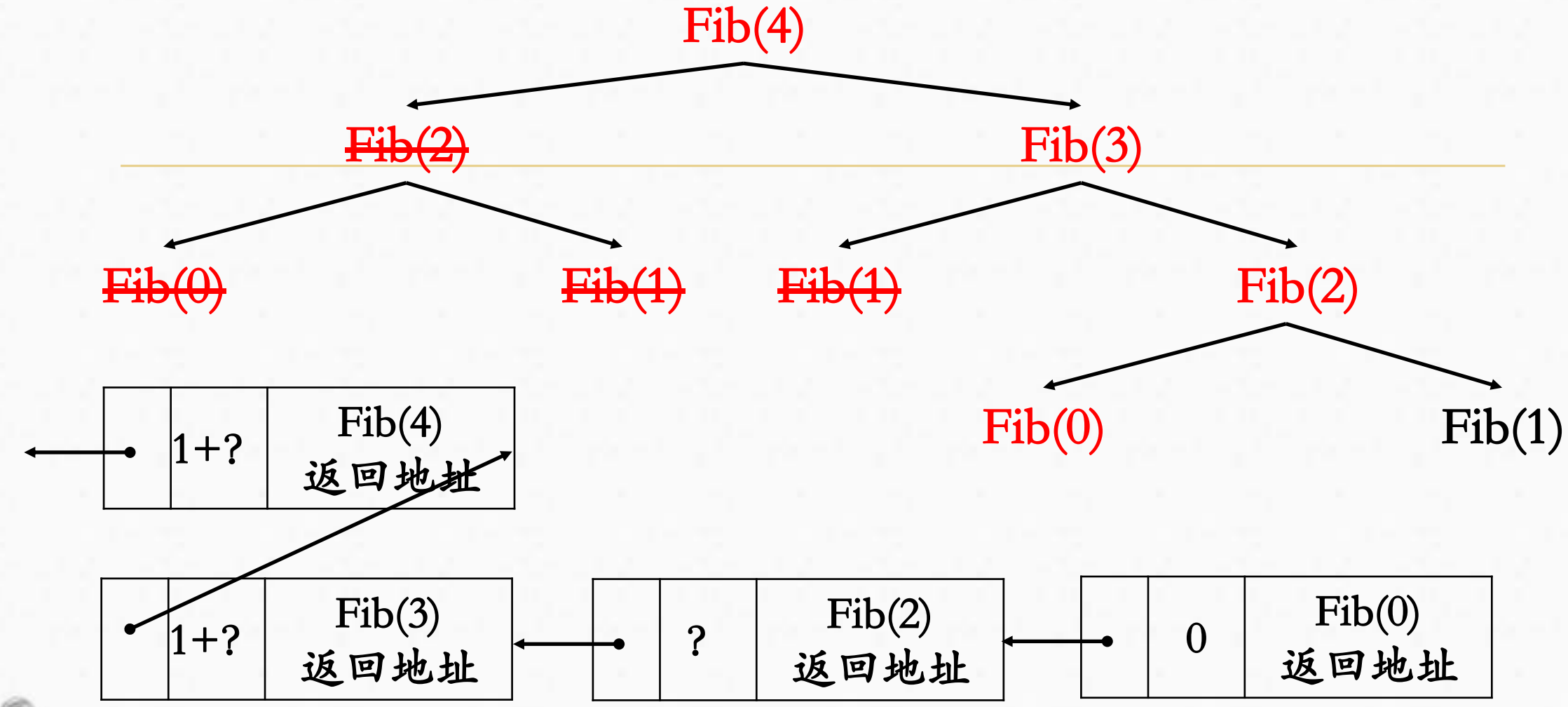

1+?
Fib(4)
返回地址
?
Fib(3)
返回地址
1
Fib(1)
返回地址

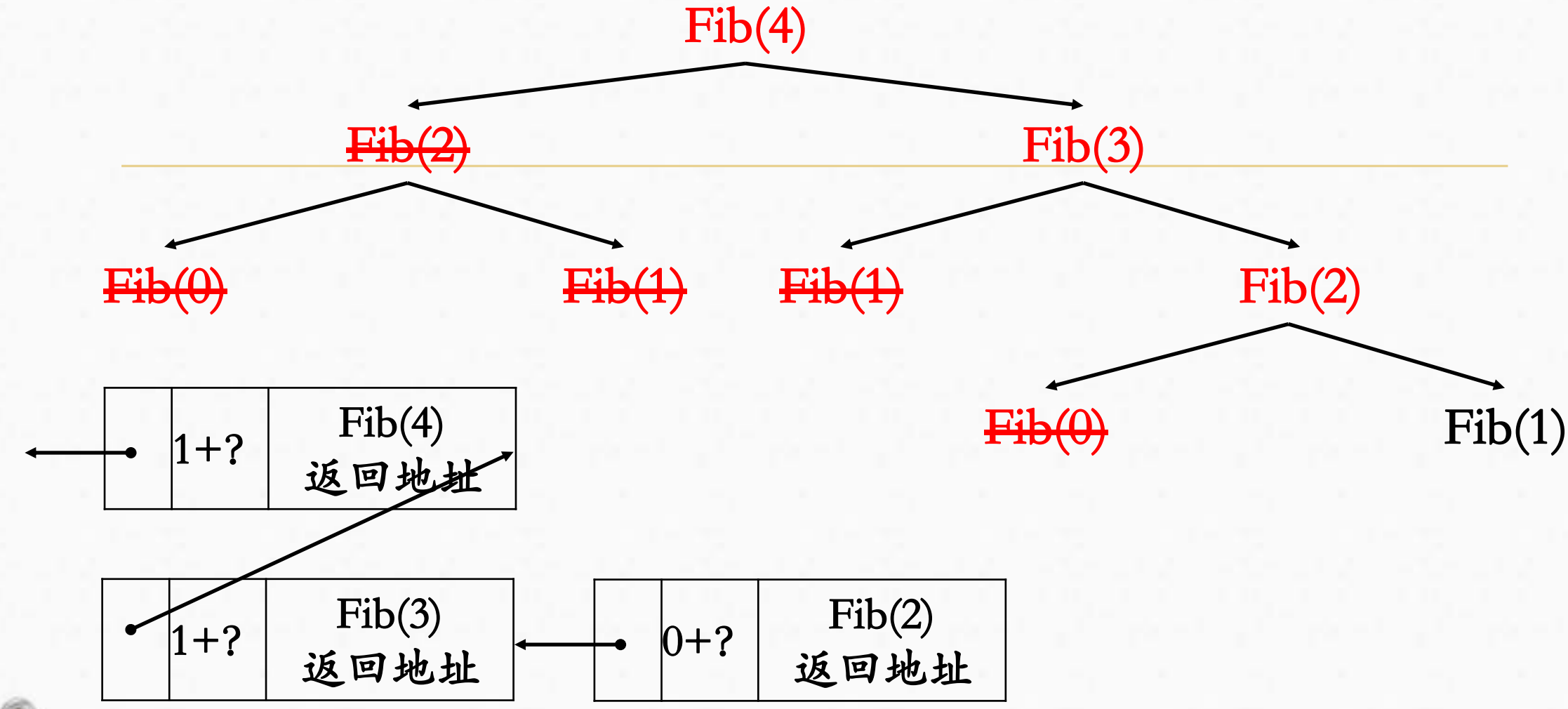

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

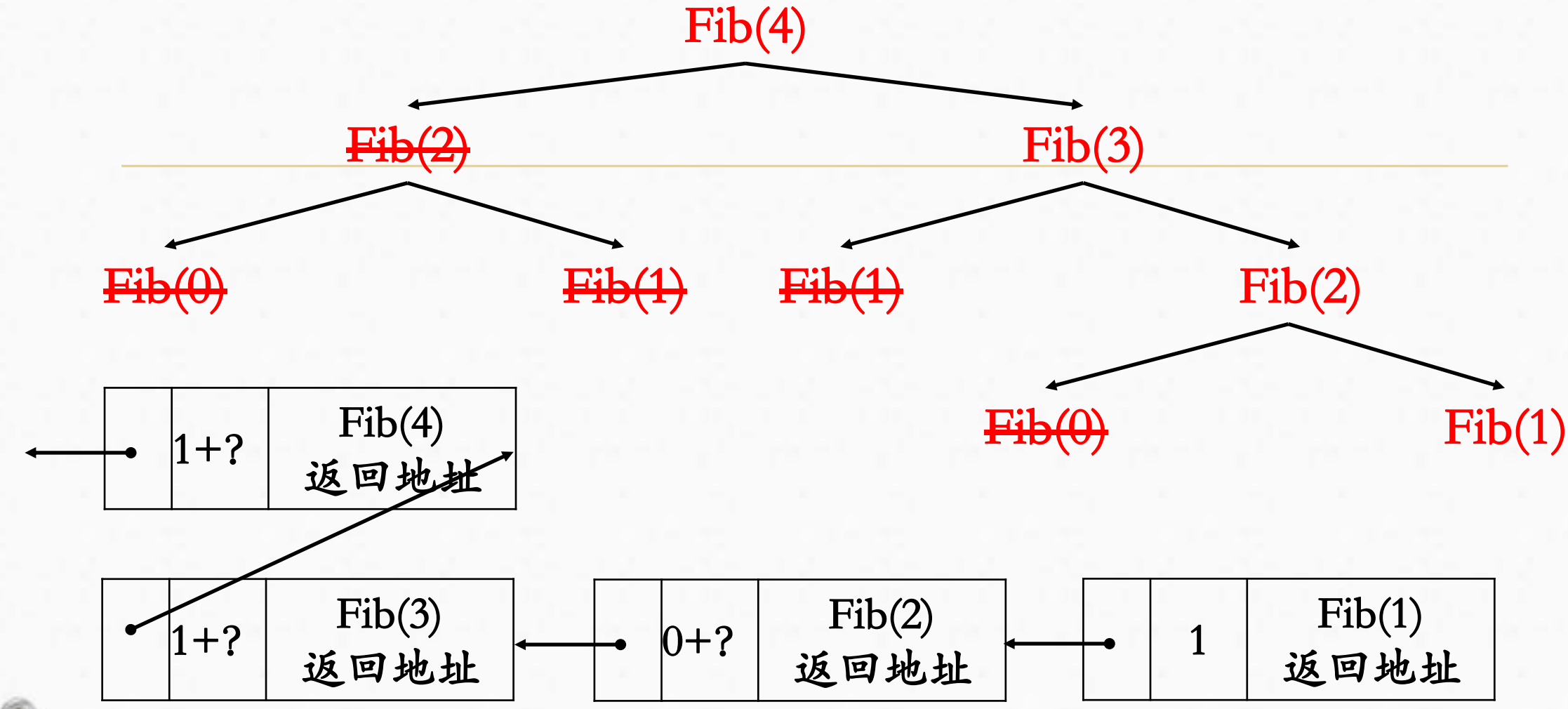

1+?
Fib(4)
返回地址
1+?
Fib(3)
返回地址

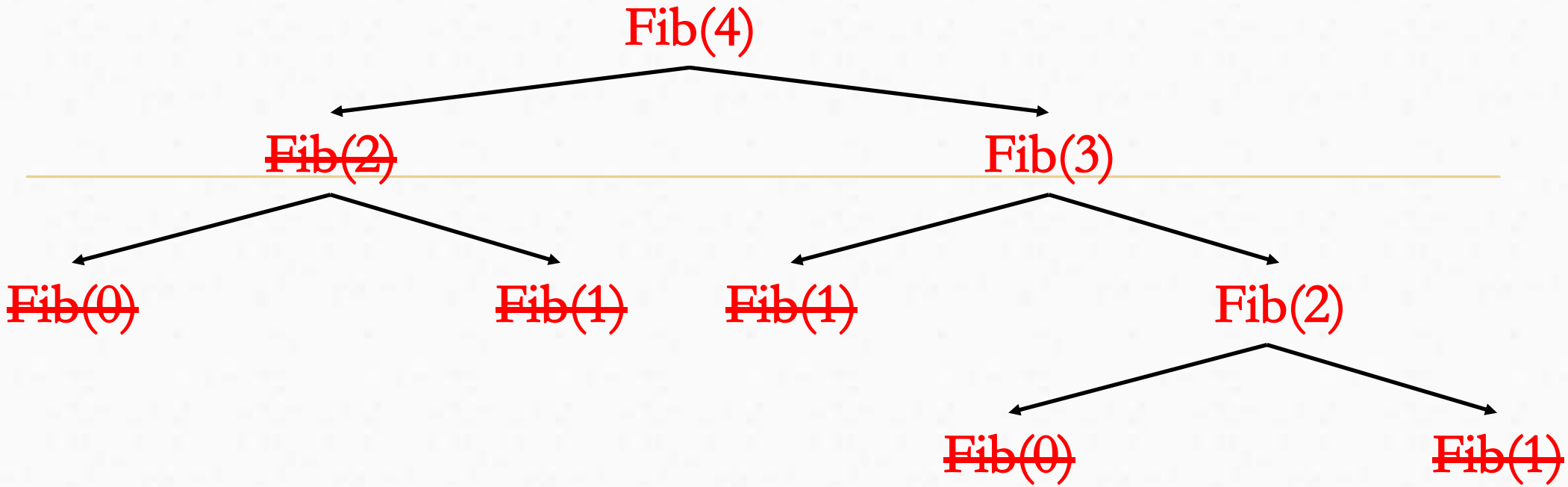

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

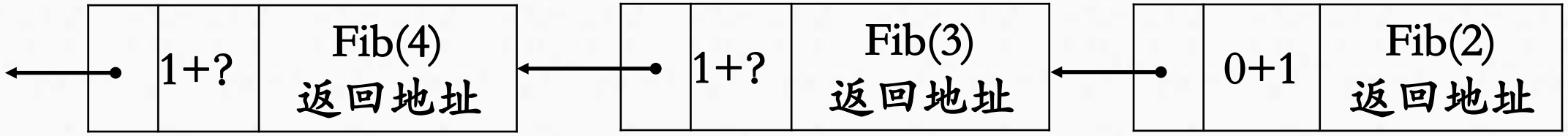

1+?
Fib(4)
返回地址
1+?
Fib(3)
返回地址
?
Fib(2)
返回地址

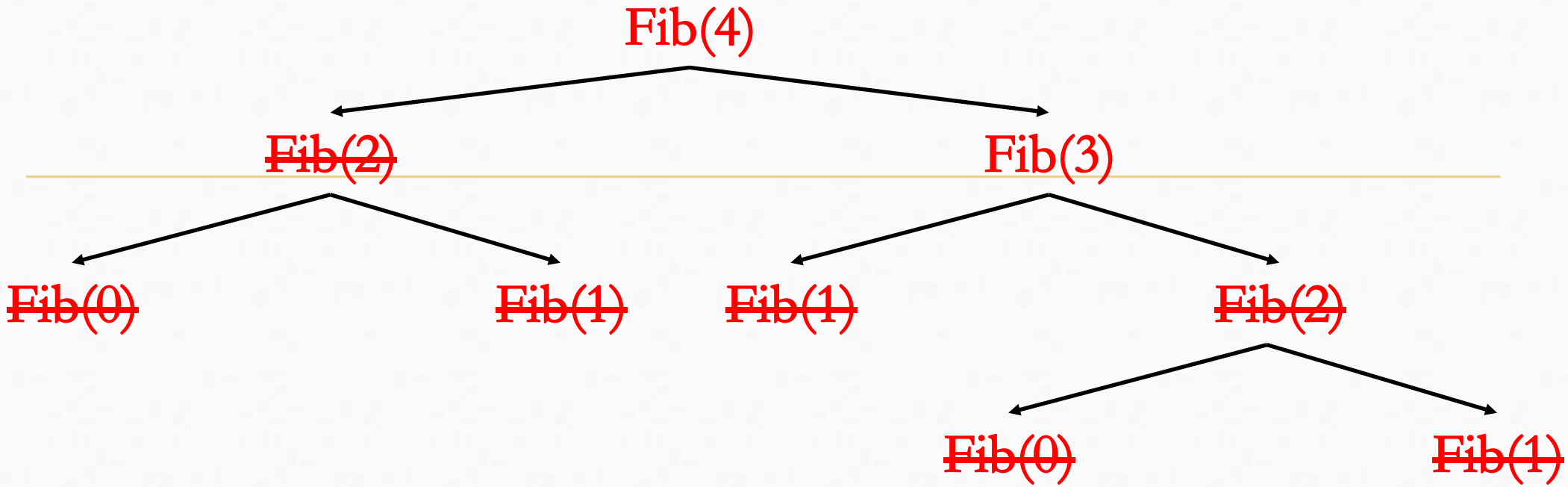

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)
1+?
Fib(4)
返回地址
1+?
Fib(3)
返回地址
?
Fib(2)
返回地址
0
Fib(0)
返回地址

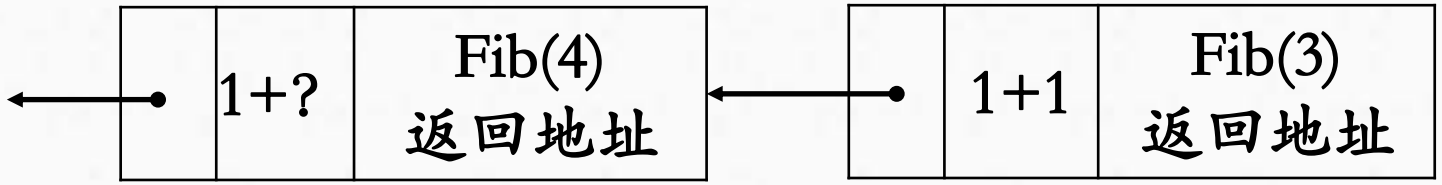

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)
1+?
Fib(4)
返回地址
1+?
Fib(3)
返回地址
0+?
Fib(2)
返回地址

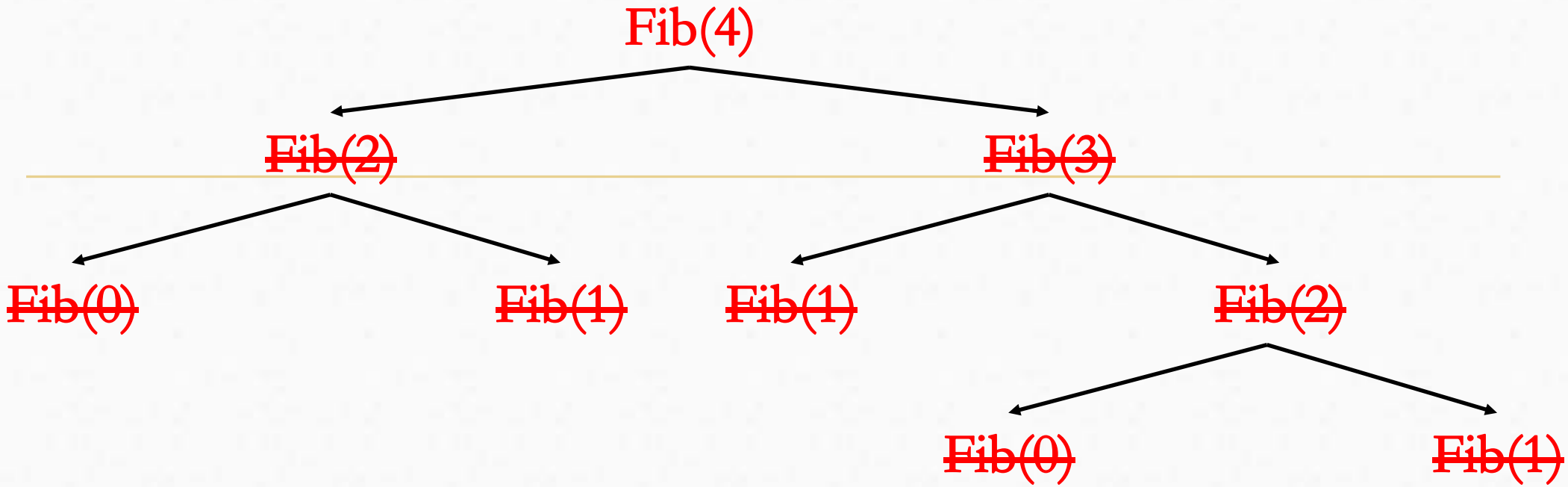

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)
1+?
Fib(4)
返回地址
1+?
Fib(3)
返回地址
0+?
Fib(2)
返回地址
1
Fib(1)
返回地址

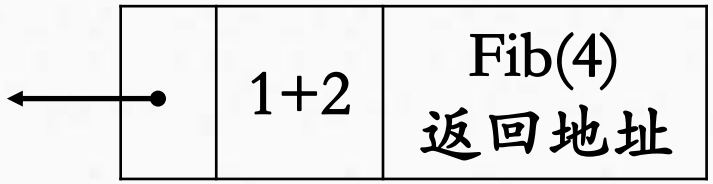

Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)

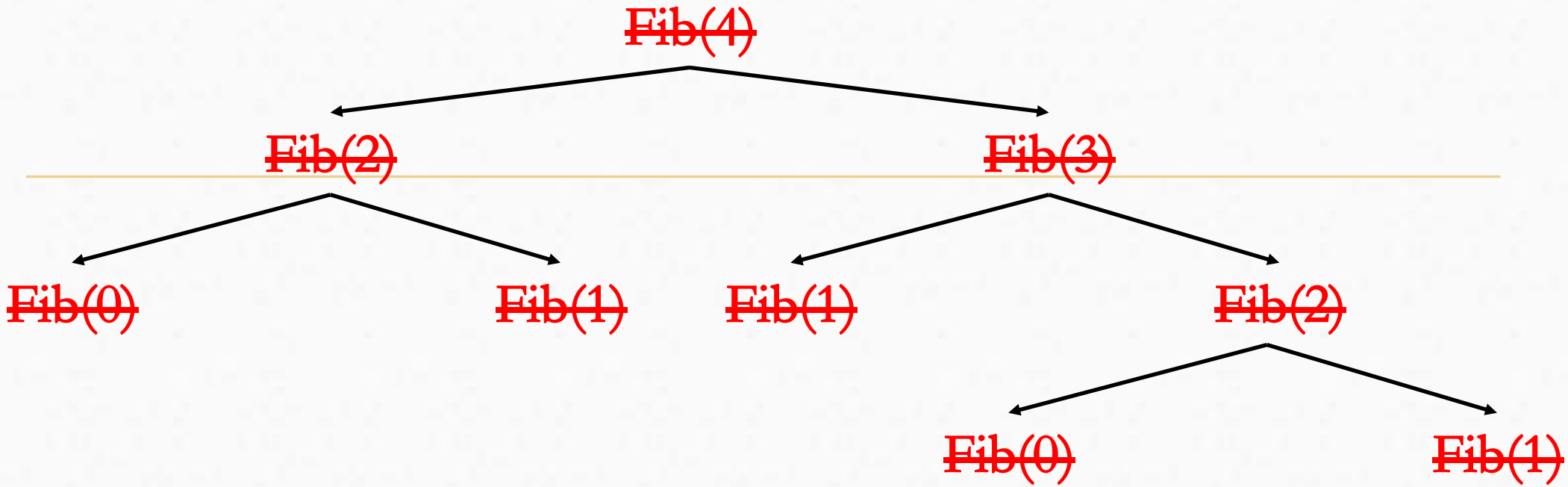

1+?
Fib(4)
返回地址
1+?
Fib(3)
返回地址
0+1
Fib(2)
返回地址


Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)


1+?
Fib(4)
返回地址
1+1
Fib(3)
返回地址


Fib(4)
Fib(2)
Fib(3)
Fib(0)
Fib(1)
Fib(1)
Fib(2)
Fib(0)
Fib(1)


1+2
Fib(4)
返回地址



Fib(4)=3

数组

目录

- 数组
- 数组的抽象数据类型
- 特殊矩阵
- 稀疏矩阵

一维数组

```
int one[5];
```

- 定义了5个整数组成的一个数组，下标从0到4
- 数组可以在定义时集体赋值

```
int one[5]={0, 1, 2, 3, 4};
```

- 可以依次对每个数据元素赋值

```
for ( i=0; i<5; i++) one[i]=i;
```

一维数组的存储结构

- 数组类型采用顺序方式存储：数组中的元素按一定顺序存放在一个连续的存储空间
- 计算机中的存储空间是一维的，一维数组元素可直接映射到存储空间中

设给长度为n的一维数组a所分配的存储块的起始地址是Loc(a[0])，若已知a的每个元素占k个存储单元，则下标为i的数组元素a[i]的存放地址Loc(a[i])是

$$Loc(a[i]) = Loc(a[0]) + i*k \qquad 0 \leqslant i < n$$

二维数组

```
int one[2][3];
```

- 定义了包含2个整型一维数组的数组，下标从0到1
- 每个一维数组又包含了3个整型，下标从0到2
- 数组可以在定义时集体赋值

```
int one[2][3]={{0, 1, 2}, {3, 4, 6}};
```

- 可以依次对每个数据元素赋值

```
for ( i=0; i<2; i++)
        for ( j=0; j<3; j++)  one[i][j]=i*j;
```

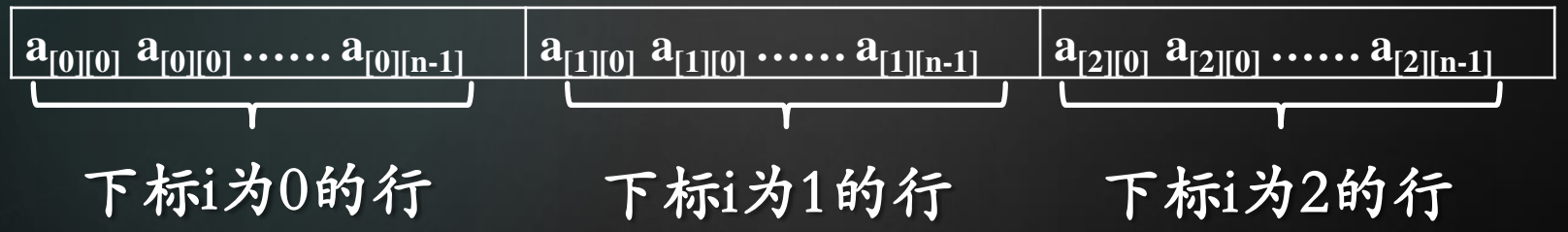
二维数组的存储结构

- 数组类型采用顺序方式存储：数组中的元素按一定顺序存放在一个连续的存储空间
- 计算机中的存储空间是一维的，二维数组元素需要按照一定规则映射到一维存储空间中
  - 行优先存储
  - 列优先存储

数组的顺序存储

二维数组a[m][n]需按照行优先映射到一维的存储空间

$$\begin{bmatrix} a_{[0][0]} & a_{[0][1]} & \cdots & \cdots & \cdots & a_{[0][n-1]} \\ a_{[1][0]} & a_{[1][1]} & \cdots & \cdots & \cdots & a_{[1][n-1]} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{[i][j]} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{[n-1][0]} & a_{[n-1][1]} & \cdots & \cdots & \cdots & a_{[n-1][n-1]} \end{bmatrix}$$



数组的顺序存储

行优先顺序的地址计算

若对于二维数组a[m][n],已知每个数组元素占k个存储单元，第一个数组元素a[0][0]的存储地址是loc(a[0][0]),则数组元素a[i][j]的存储地址loc(a[i][j])为

loc(a[i][j])=loc(a[0][0])+(i*n+j)*k　(0≤i<m;0≤j<n)

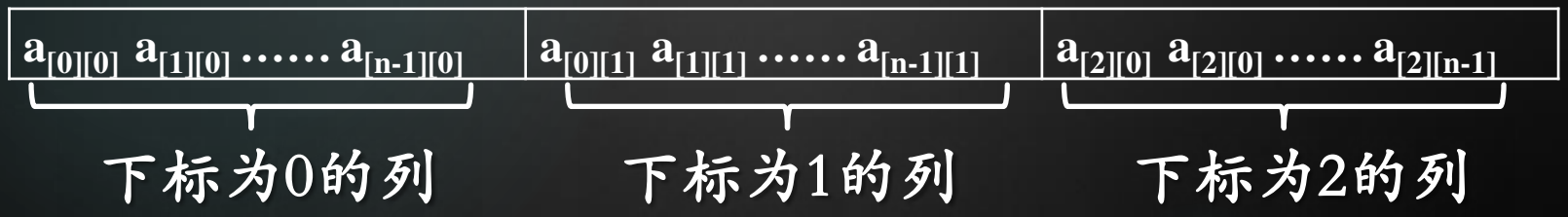
$$\begin{bmatrix} a_{[0][0]} & a_{[0][1]} & \cdots & \cdots & \cdots & a_{[0][n-1]} \\ a_{[1][0]} & a_{[1][1]} & \cdots & \cdots & \cdots & a_{[1][n-1]} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{[i][j]} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{[n-1][0]} & a_{[n-1][1]} & \cdots & \cdots & \cdots & a_{[n-1][n-1]} \end{bmatrix}$$

数组的顺序存储

列优先顺序的地址计算

$$loc(a[i][j])=loc(a[0][0])+(j*m+i)*k \quad (0\le i<m;0\le j<n)$$

$$\begin{bmatrix} a_{[0][0]} & a_{[0][1]} & \cdots & \cdots & \cdots & a_{[0][n-1]} \\ a_{[1][0]} & a_{[1][1]} & \cdots & \cdots & \cdots & a_{[1][n-1]} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & a_{[i][j]} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ a_{[n-1][0]} & a_{[n-1][1]} & \cdots & \cdots & \cdots & a_{[n-1][n-1]} \end{bmatrix}$$



例1：10×10的整型数组A，其每个数组元素占2个字节，已知A[0][0]在内存中的地址是100，按行优先，A[4][6]的地址是____。

A．228　　B．192　　C．124　　D．138

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	[0][8]	[0][9]
[1][0]						[1][6]			
[2][0]						[2][6]			
[3][0]						[3][0]			
[4][0]						[4][6]			
[5][0]						[5][0]			
[6][0]						[6][0]			
[7][0]						[7][0]			
[8][0]						[8][0]			
[9][0]						[9][0]			

loc(a[0][0])+(i×n+j)×k＝100+（4*10+6）*2＝192

二维数组的顺序存储

- Loc(a[0][0])被称为基地址，它是存储数组的存储空间的起始地址。
- 数组一旦规定了它的维数和各维的长度，便可为它分配存储空间
- 只要给出数组元素下标，就可根据相应的地址计算公式求得数组元素的存储位置来存取元素
- 存取数组中任何一个元素所需的时间是相同的，具有这一存取特点的存储结构为随机存取的存储结构。

多维数组的顺序存储

多维数组$a[m_1][m_2]\cdots[m_n]$，数组元素$a[i_1][i_2]\cdots[i_n]$的存储地址为

$$loc(a[i_1][i_2]\cdots[i_n]) = loc(a[0]\cdots[0]) + (i_1 * m_2 * m_3 * \cdots * m_n + i_2 * m_3 * m_4 * \cdots * m_n + \cdots + i_{n-1} * m_n + i_n) * k$$

$(0 \le i_j < m_j,\ 1 \le j \le n)$

i_1	m_2	m_3	m_4	…	m_n
1	i_2	m_3	m_4	…	m_n
1	1	i_3	m_4	…	m_n
1	1	1	i_4	…	m_n
1	1	1	1	⋱	m_n
1	1	1	1	…	i_n

- 数组元素的存储位置是其下标的线性函数。通过计算地址便可实现对数组元素的随机存取。
- C语言中数组是不允许整体赋值的。
- C语言中对数组下标超出正常范围的访问并不限制

  例如对长度为5的一维数组a，不合法的访问a[-3]，a[7]
- 顺序存储一般借助数组来实现
- 数组是静态数据结构，其存储空间的大小需具体确定，并预先分配，一旦分配则难以扩充
- 数组名本身存储了指针值，其保存数组的首地址

- C语言提供的数组并非一个完备的数据结构

（1）不能实现数组的整体赋值；

（2）不能将数组作为函数值返回；

（3）对数组元素下标不提供边界检查；

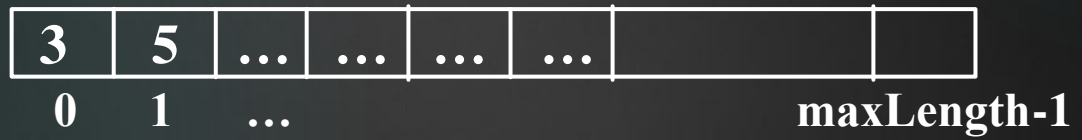
（4）将数组名作为变量进行传递时，传递的实际上是数组的基地址。

目录

- 数组
- 数组的抽象数据类型
- 特殊矩阵
- 稀疏矩阵

数组的抽象数据类型

- 数组是下标index 和值value 组成的偶对的集合。
  - 每个下标都与一个值对应，该值称做数组元素。
  - 每个偶对形如：(index,value)



数组的抽象数据类型

上一节我们讨论了C语言对数组所提供的支持。C语言提供的数组并非一个完备的数据结构，主要体现在：

（1）不能实现数组的整体赋值；

（2）不能将数组作为函数值返回；

（3）对数组元素下标不提供边界检查；

（4）将数组名作为变量进行传递时，传递的实际上是数组的基地址。

如果我们需要像使用int、float、char这些基本类型一样将数组作为一种数据类型来使用，则需要基于C语言所提供的数组支持，定义一个数组的抽象数据类型并实现。

ADT 4-1 Array{

数据：

下标$i=<i_1, i_2, \cdots, i_n>$和元素$v$的偶对<i, v>集合。其中n是数组维度的数量，$i_1, i_2, \cdots, i_n$ 表示元素在数组各个维度的下标值；数组在各个维度的长度分别是m_1，m_2，$\cdots$，m_n；数据元素类型为ElemType。

运算：

CreateArray (A, $m_1, m_2, \cdots, m_n$)

创建运算：申请n维数组A所需存储空间并分配给数组A，成功分配则函数返回OK，否则，函数返回ERROR。

DestroyArray (A)

清除运算：判断数组A是否存在，若存在，则撤销数组，函数返回OK；否则，函数返回ERROR。

RetrieveArray (A, $i_1, i_2, \cdots, i_n$, x)

数组元素查询运算：判断数组A是否存在，若不存在，则函数返回ERROR；否则，对$i_1, i_2, \cdots, i_n$进行边界检查，若下标非法，则函数返回ERROR，否则在参数x中返回下标为$i_1, i_2, \cdots, i_n$的数组元素，函数返回OK。

StoreArrayItem (A, i_1, i_2, …, i_n, x)

数组元素赋值运算：判断数组A是否存在，若不存在，则函数返回ERROR；否则，对i_1, i_2, …, i_n进行边界检查，若下标非法，则函数返回ERROR，否则将下标为i_1, i_2, …, i_n的数组元素值设置为x，函数返回OK。

OutputArray (A)

数组输出运算：判断数组A是否存在，若不存在，则函数返回；否则，将数组中所有元素依次输出。

CopyArray (A, B)

数组拷贝运算：判断数组A和B是否存在，若A或B不存在，则函数返回ERROR；否则，判断数组A和B是否大小相同（指每一个维度的大小都相同），若不同，则函数返回ERROR；否则，将数组A中元素依次拷贝到数组B中；

……

}

借助C语言的结构体和动态数组，实现一个三维整型数组类型TDArray及其运算。

```
typedef struct tdarray
    {
        int m1;
        int m2;
        int m3;
        int *array;
}TDArray;
```

```
Status CreateArray (TDArray *tdarray, int m1, int m2, int m3)
{
    tdarray->m1 = m1;
    tdarray->m2 = m2;
    tdarray->m3 = m3;
    tdarray->array = (int *)malloc(m1*m2*m3*sizeof(int));
    if(!tdarray->array) return ERROR;
    return OK;
}

Status DestroyArray (TDArray * tdarray)
{
    if(!tdarray) return ERROR;
    if(tdarray->array) free(tdarray->array);
    free(tdarray);
    return OK;
}
```

```
Status RetrieveArray (TDArray tdarray, int i1, int i2, int i3, int *x)
{
    if(!tdarray.array) return NotPresent;
    if( i1<0 || i2<0 || i3<0
            ||i1>=tdarray.m1||i2>=tdarray.m2||i3>=tdarray.m3)
            return IllegalIndex;
    *x = *(tdarray.array+i1*m2*m3+i2*m3+i3);
    Return OK;
}
```

```
Status StoreArrayItem (TDArray *tdarray, int i1, int i2, int i3, int x)
{
    if(!tdarray.array) return NotPresent;
    if(i1<0 ||i2<0||i3<0||i1>=tdarray->m1
            ||i2>=tdarray->m2||i3>=tdarray->m3)
            return IllegalIndex;
    *(tdarray.array+i1*m2*m3+i2*m3+i3) = x;
    return OK;
}
```

目录

- 数组
- 数组的抽象数据类型
- 特殊矩阵
- 稀疏矩阵

矩阵

- 矩阵 (Matrix) 在数学中描述为一个按照长方阵列排列的复数或实数集合。
- 矩阵的概念被广泛应用于数学、物理学、计算机科学等学科中。在计算机科学中，图像处理、三维动画制作等都需要应用矩阵及其运算。
- 基于矩阵具有行与列的概念，二维数组非常适合于描述矩阵。

特殊矩阵之对称矩阵

- 在n×n的矩阵A中，若$a_{[i][j]}=a_{[j][i]}(1\le i,j\le n)$，则称其为n阶对称矩阵
- 只存储上三角(或下三角)中的元素(包括对角线上的元素)
- n×n的三角矩阵，需要n(n+1)/2个元素的存储空间，提高了存储效率

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

特殊矩阵之对称矩阵

- 以行优先顺序在数组B中存储下三角元素，则矩阵元素$a_{[i][j]}$在数组b中的存储位置k为：

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & i\geq j\\ \dfrac{j(j+1)}{2}+i & j\geq i\end{cases}$$

$$\begin{pmatrix} a_{[0][0]} & & & & \\ a_{[1][0]} & a_{[1][1]} & & & \\ a_{[2][0]} & a_{[2][1]} & a_{[2][2]} & & \\ a_{[3][0]} & a_{[3][1]} & a_{[3][2]} & a_{[3][3]} & \\ a_{[4][0]} & a_{[4][1]} & a_{[4][2]} & a_{[4][3]} & a_{[4][4]} \end{pmatrix}$$

$a_{[0][0]}$	$a_{[1][0]}$ $a_{[1][1]}$	$a_{[2][0]}$ $a_{[2][1]}$ $a_{[2][2]}$	$a_{[3][0]}$ $a_{[3][1]}$ $a_{[3][2]}$ $a_{[3][3]}$	$a_{[4][0]}$ $a_{[4][1]}$ $a_{[4][2]}$ $a_{[4][3]}$ $a_{[4][4]}$
1	2	3	4	5

特殊矩阵之上下三角矩阵

- 主对角以下元素全为0的方阵称为上三角矩阵。
- 主对角以上元素全为0的方阵称为下三角矩阵。
- 上、下三角矩阵采用对称矩阵的存储方式，只存储主对角线及其以上（或以下）的元素。

$$\begin{pmatrix} a_{[0][0]} & 0 & 0 & 0 & 0 \\ a_{[1][0]} & a_{[1][1]} & 0 & 0 & 0 \\ a_{[2][0]} & a_{[2][1]} & a_{[2][2]} & 0 & 0 \\ a_{[3][0]} & a_{[3][1]} & a_{[3][2]} & a_{[3][3]} & 0 \\ a_{[4][0]} & a_{[4][1]} & a_{[4][2]} & a_{[4][3]} & a_{[4][4]} \end{pmatrix}$$

$$k = \begin{cases} \dfrac{i(i+1)}{2} + j & \text{行优先存储的下三角矩阵} \\ \dfrac{j(j+1)}{2} + i & \text{列优先存储的上三角矩阵} \end{cases}$$

目录

- 数组
- 数组的抽象数据类型
- 特殊矩阵
- 稀疏矩阵

稀疏矩阵

- 矩阵中非零元素数量占元素总数的比例称为矩阵的**稠密度**
- 当矩阵的稠密度很小，即包含大量零元素的矩阵称为**稀疏矩阵** (sparse matrix)
- 通常认为稠密度**小于5%**的矩阵即可视为稀疏矩阵
- 稀疏矩阵中**零元素**的位置**分布没有规律**。
- 稀疏矩阵常出现于大规模集成电路设计、图像处理等应用领域
- 由于稀疏矩阵中只有少量非零元素，为了节省存储空间，对稀疏矩阵可以**只存储非零元素**。

ADT SparseMatrix{

数据：

大多数元素为零的矩阵。

运算：

void CreateMatrix(M, m,n)

构造运算: 构造一个m×n的空稀疏矩阵。

void Clear(M)

清除运算： 清除稀疏矩阵中的所有非零元素。

SparseMatrix Transpose(a)

转置运算： 返回稀疏矩阵a的转置矩阵。

SparseMatrix Madd(a, b)

加法运算： 返回稀疏矩阵a和b的和。

SparseMatrix MMulti(a, b)

乘法运算： 返回稀疏矩阵a和b的积。

StoreSparseMatrixItem (M, i, j, x)

稀疏矩阵元素赋值运算：判断稀疏矩阵M是否存在，若不存在，则函数返回ERROR；否则，判断x是否为非零元，是非零元，则设置稀疏矩阵中下标为i, j的元素值为x，函数返回OK；否则，函数返回ERROR。

RetrieveSparseMatrix (M, i, j, x)

稀疏矩阵元素查找运算：判断稀疏矩阵是否存在，若不存在，则函数返回ERROR；否则，对i, j进行边界检查，若下标非法，则函数返回ERROR；否则，在稀疏矩阵中查找下标为i, j的元素，若存在下标为i, j的元素，在参数*x中返回该元素，函数返回OK；否则，在参数*x中返回零值，函数返回OK。

OutputSparseMatrix (A)

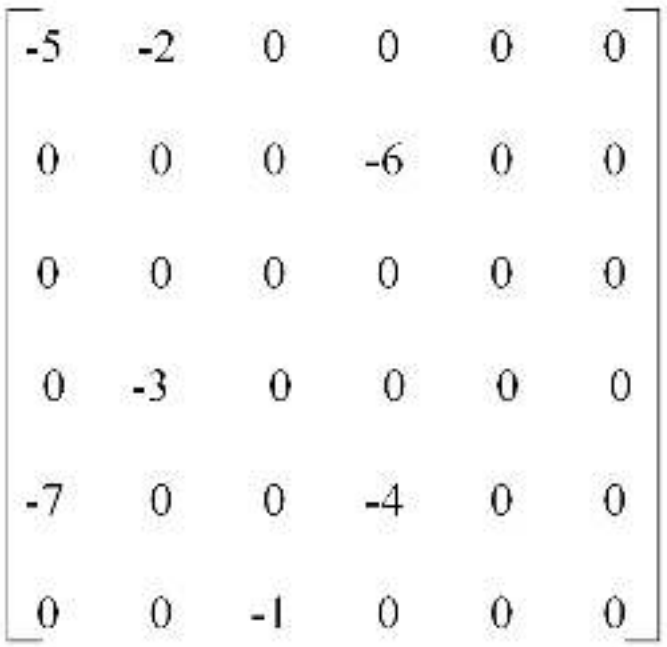
稀疏矩阵输出运算：判断稀疏矩阵A是否存在，若不存在，则函数返回；否则，将矩阵所有非零元素依次输出。

}

稀疏矩阵的顺序存储

- 非零元的位置分布没有规律，需连同其位置信息一起存储，否则无法实现无损解压
- 将稀疏矩阵中非零元a_{ij}以三元组 $<i, j, a_{ij}>$ 表示

```
#define maxSize 100 /*可存储的非零元数量上限*/
typedef int ElemType;

typedef struct term{
        int col, row;     /*非零元在稀疏矩阵中的
                            行下标col和列下标row*/
        ElemType value;       /*非零元的值*/
}Term;
```



-5	-2	0	0	0	0
0	0	0	-6	0	0
0	0	0	0	0	0
0	-3	0	0	0	0
-7	0	0	-4	0	0
0	0	-1	0	0	0

(a) 稀疏矩阵

	i	j	a_{ij}
0	0	0	-5
1	0	1	-2
2	1	3	-6
3	3	1	-3
4	4	0	-7
5	4	3	-4
6	5	2	-1

(b) 行三元组表

	i	j	a_{ij}
0	0	0	-5
1	4	0	-7
2	0	1	-2
3	3	1	-3
4	5	2	-1
5	1	3	-6
6	4	3	-4

(c) 列三元组表

```
typedef struct sparsematrix{
        int m, n, t;            /*m是矩阵行数，n是矩阵列数，
                                  t是实际非零元个数*/
        Term table[maxSize];   /*存储非零元的三元组表*/
}SparseMatrix;
```

稀疏矩阵的简单转置算法

- 采用二维数组A存储一个普通 m×n 矩阵，假设将矩阵转置结果存储到 n×m 矩阵B中

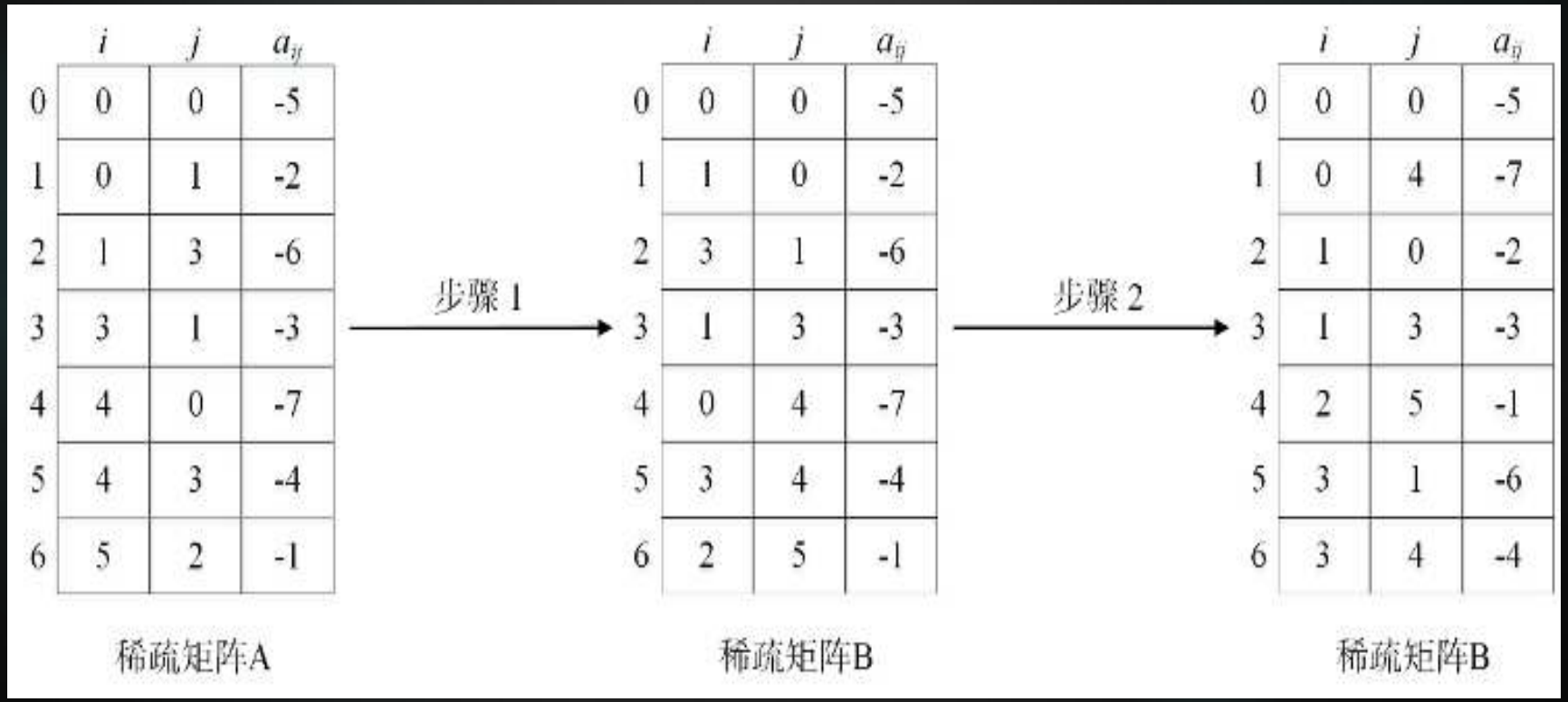
```
for(int i=0; i<m; i++)
        for(int j=0; j<n; j++)
            B[j][i] = A[i][j];
```

上述程序段需要访问矩阵中的每一个元素，其时间复杂度为O(mn)

第一种简单转置算法

步骤1：依次访问A的行三元组表中各个三元组 <i, j, a_{ij}>，交换元素行列号后依次保存到B的行三元组表中。

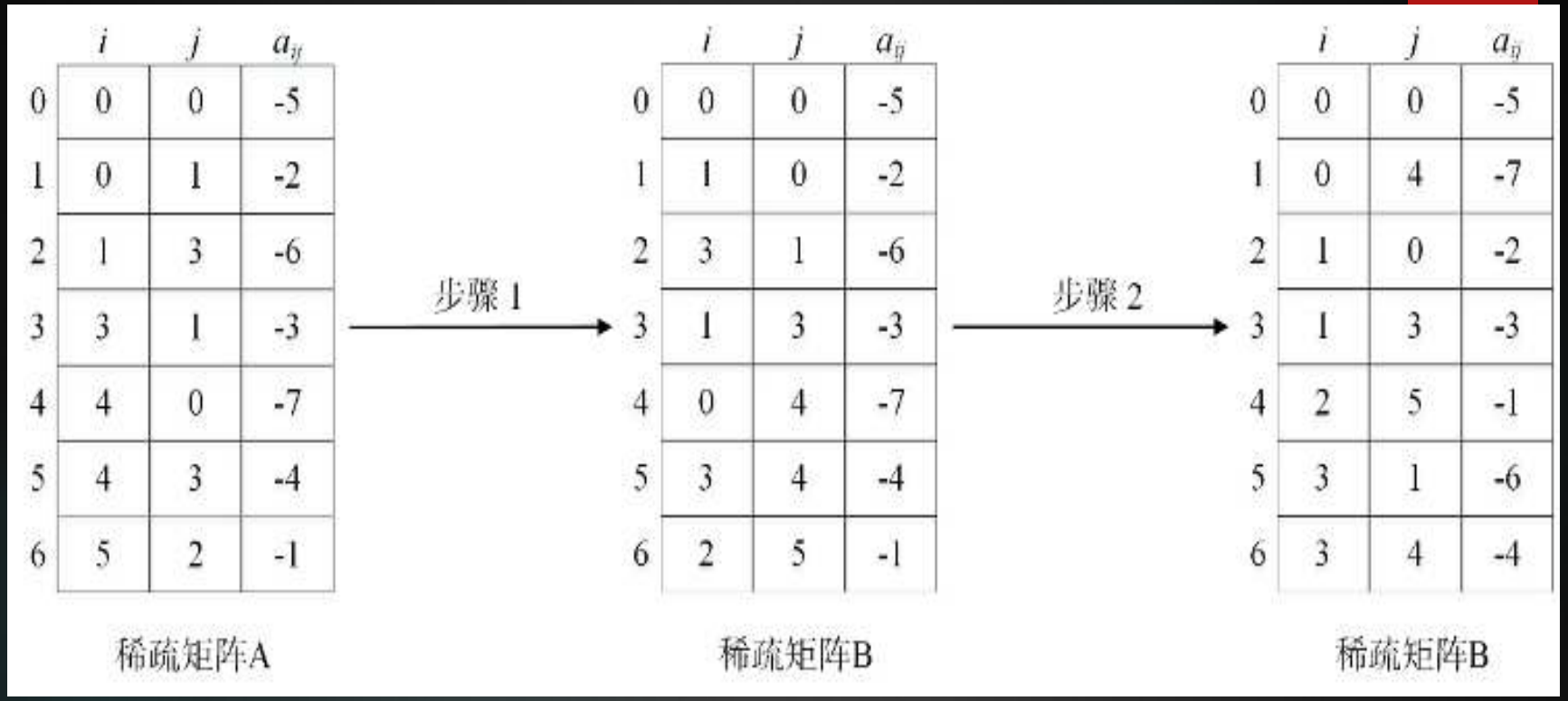
步骤2：将B的行三元组表中的行三元组按照下标i值从小到大重新排序。



	i	j	a_{ij}
0	0	0	-5
1	0	1	-2
2	1	3	-6
3	3	1	-3
4	4	0	-7
5	4	3	-4
6	5	2	-1

	i	j	a_{ij}
0	0	0	-5
1	1	0	-2
2	3	1	-6
3	1	3	-3
4	0	4	-7
5	3	4	-4
6	2	5	-1

	i	j	a_{ij}
0	0	0	-5
1	0	4	-7
2	1	0	-2
3	1	3	-3
4	2	5	-1
5	3	1	-6
6	3	4	-4



- 步骤2决定算法时间复杂度，步骤2是一个排序过程，采用不同排序算法的时间复杂度为$O(t^2)$ 或$O(t\log_2 t)$

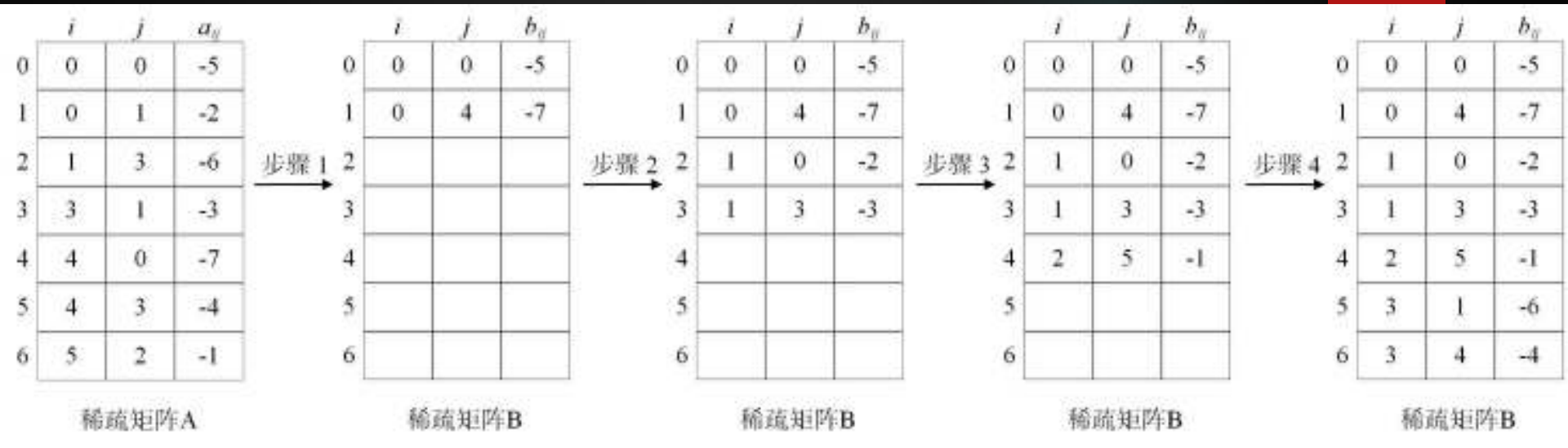
第二种简单转置算法

步骤1：对A的行三元组表进行第1次扫描，找到列下标j = 0的所有三元组 <i, 0, a_{i0}>，交换元素行列号后依次保存到B的行三元组表中。

步骤2：对A的行三元组表进行第2次扫描，找到列下标j = 1的所有三元组 <i, 1, a_{i1}>，交换元素行列号后依次保存到B的行三元组表中。

……

步骤n：对A的行三元组表进行第n次扫描，找到列下标j = n-1的所有三元组 <i, n-1, $a_{i,n-1}$>，交换元素行列号后依次保存到B的行三元组表中。



上述算法对A的行三元组表进行了n次扫描，时间复杂度为O(nt)

稀疏矩阵的快速转置算法

剖透：通过增加适量额外存储空间，存储预先计算的辅助信息，能够实现快速稀疏矩阵转置，其算法时间复杂度可以降低至O(n+t)

稀疏矩阵的快速转置算法

实现快速转置算法需要借助两个一维辅助数组***num***和***k***，这两个数组长度都为*n*（稀疏矩阵A的列数）

数组num的数组元素num[j] 统计稀疏矩阵A中列号为 *j*的非零元个数

只需要对A的行三元组表进行一次扫描，即可统计出A的每一列非零元个数

```
for(int j=0; j<n; j++) num[j] = 0;  /* num初始化*/
for(int i=0; i<t; i++) num[A.table[i].col]++;
```

稀疏矩阵的快速转置算法

- 数组k的数组元素k[j] 统计稀疏矩阵A中列号从0到j-1列的非零元个数总和（第1列至第j列）
- 该值也表示本列第一个非零元在转置稀疏矩阵B的行三元组表中的位置

	i	j	a_{ij}
0	0	0	-5
1	0	1	-2
2	1	3	-6
3	3	1	-3
4	4	0	-7
5	4	3	-4
6	5	2	-1

A

	i	j	b_{ij}
0	0	0	-5
1	0	4	-7
2	1	0	-2
3	1	3	-3
4	2	5	-1
5	3	1	-6
6	3	4	-4

B

思考：

转置矩阵B中i=0的第一个元素在行三元组表中的位置；

转置矩阵B中i=1的第一个元素在行三元组表中的位置；

转置矩阵B中i=2的第一个元素在行三元组表中的位置；

转置矩阵B中i=3的第一个元素在行三元组表中的位置；

演示

稀疏矩阵的快速转置算法

- 只需要对辅助数组num进行一次扫描，即可完成数组k中各元素值的计算，程序段如下所示：

```
for(int j = 0; j<n; j++) k[j] = 0;  /* k初始化*/
for(int j = 1; j<n; j++) k[j] = k[j-1] + num[j-1];
```

j	0	1	2	3	4	5
num[j]	2	2	1	2	0	0
k[j]	0	2	4	5	7	7

计算数组num和k的两个程序段时间复杂度为O(n+t)。

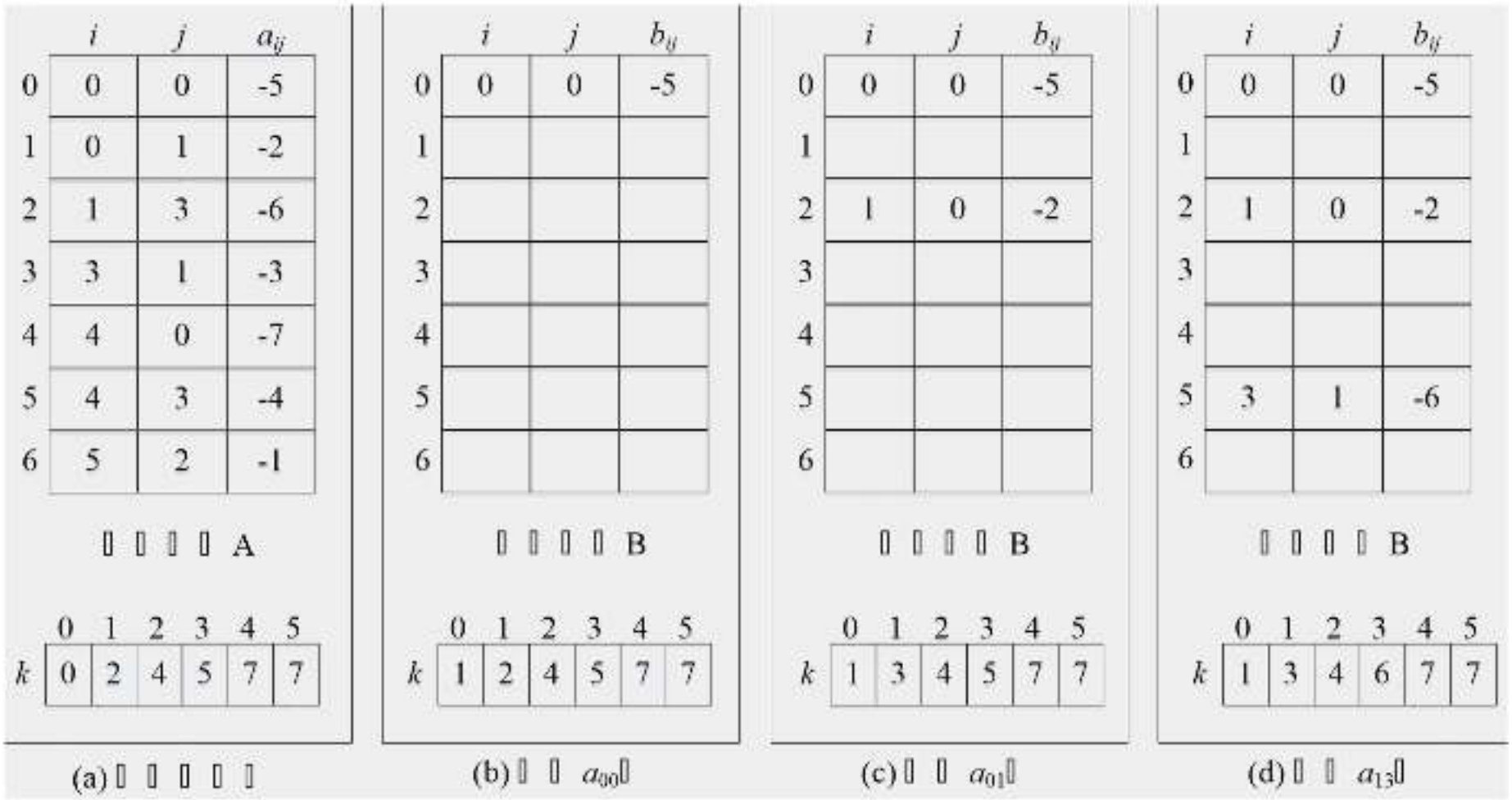
借助辅助数组k，即可完成快速转置，程序段如下所示：

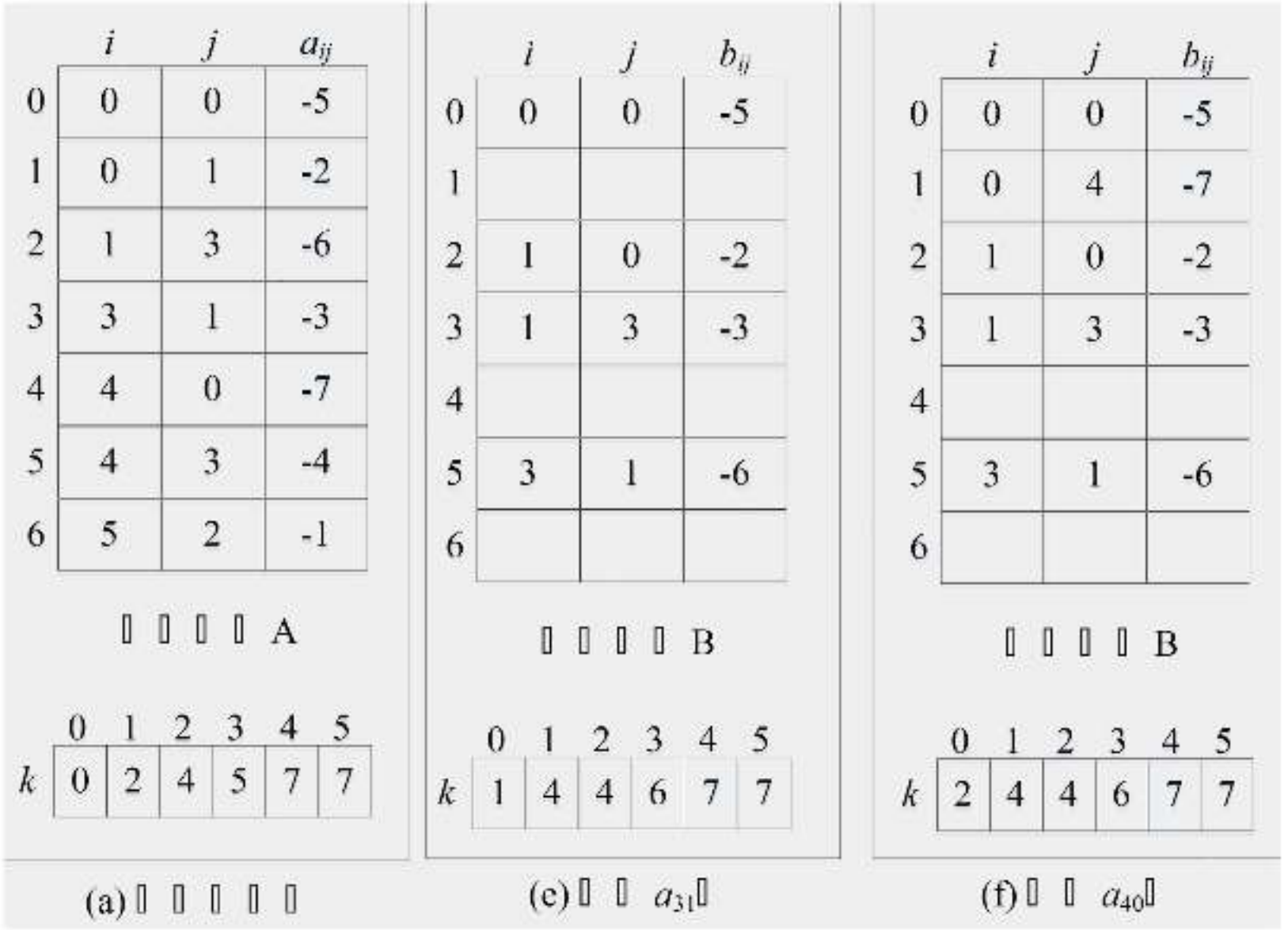
```
for(int i=0; i<t; i++){
    int index = k[A. table[i].col]++;
    B. table[index].col = A. table[i].row;
    B. table[index].row = A. table[i].col;
    B. table[index].value = A. table[i].value;
}
```

注意在快速转置开始前，k[j]的值表示稀疏矩阵列号为j的列中第一个非零元在转置矩阵中的存储位置

在快速转置执行的过程中，k[j]每被访问一次，都需要执行一次自加操作，表示该列中下一个非零元在转置矩阵中的存储位置。

上述程序段只需要对稀疏矩阵A的行三元组表进行一遍扫描，时间复杂度为O(t)。因此，稀疏矩阵的快速转置矩阵时间复杂度为O(n+t)（包含辅助数组的计算时间）


i j a_{ij}
0 0 0 -5
1 0 1 -2
2 1 3 -6
3 3 1 -3
4 4 0 -7
5 4 3 -4
6 5 2 -1
A
0 1 2 3 4 5
k 0 2 4 5 7 7
(a)
i j b_{ij}
0 0 0 -5
1
2
3
4
5
6
B
0 1 2 3 4 5
k 1 2 4 5 7 7
(b) a_{00}
i j b_{ij}
0 0 0 -5
1
2 1 0 -2
3
4
5
6
B
0 1 2 3 4 5
k 1 3 4 5 7 7
(c) a_{01}
i j b_{ij}
0 0 0 -5
1
2 1 0 -2
3
4
5 3 1 -6
6
B
0 1 2 3 4 5
k 1 3 4 6 7 7
(d) a_{13}



	i	j	a_{ij}
0	0	0	-5
1	0	1	-2
2	1	3	-6
3	3	1	-3
4	4	0	-7
5	4	3	-4
6	5	2	-1

A

k	0	1	2	3	4	5
	0	2	4	5	7	7

(a)

	i	j	b_{ij}
0	0	0	-5
1			
2	1	0	-2
3	1	3	-3
4			
5	3	1	-6
6			

B

k	0	1	2	3	4	5
	1	4	4	6	7	7

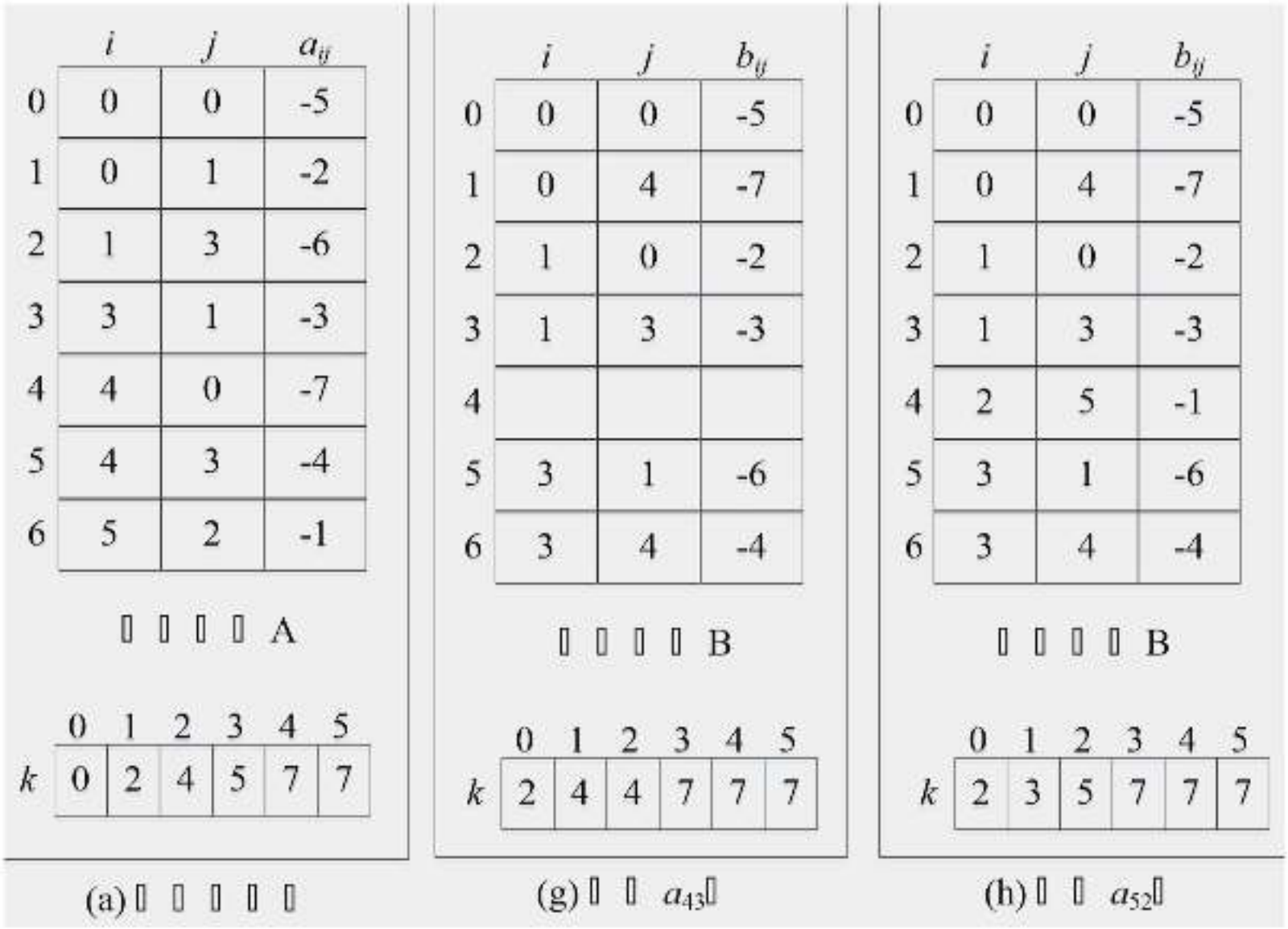
(c) a_{31}

	i	j	b_{ij}
0	0	0	-5
1	0	4	-7
2	1	0	-2
3	1	3	-3
4			
5	3	1	-6
6			

B

k	0	1	2	3	4	5
	2	4	4	6	7	7

(f) a_{40}



	i	j	a_{ij}
0	0	0	-5
1	0	1	-2
2	1	3	-6
3	3	1	-3
4	4	0	-7
5	4	3	-4
6	5	2	-1

A

k	0	1	2	3	4	5
	0	2	4	5	7	7

(a)

	i	j	b_{ij}
0	0	0	-5
1	0	4	-7
2	1	0	-2
3	1	3	-3
4			
5	3	1	-6
6	3	4	-4

B

k	0	1	2	3	4	5
	2	4	4	7	7	7

(g) a_{43}

	i	j	b_{ij}
0	0	0	-5
1	0	4	-7
2	1	0	-2
3	1	3	-3
4	2	5	-1
5	3	1	-6
6	3	4	-4

B

k	0	1	2	3	4	5
	2	3	5	7	7	7

(h) a_{52}

稀疏矩阵的快速转置算法

- 演示

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1

	0	1	2	3	4	5
0	16	0	0	22	0	−16
1	0	12	3	0	0	0
2	0	0	0	−8	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	15	0	0	0

稀疏矩阵*M*

Step 1: 计算每列非零元素个数

```
for (i=0; i<t; i++)
  num[A. table[i].col]++;
```

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	0	2	3	5	7	7

Step 2: 计算k数组

k[0] = 0 表示第0列首个非零元素存储在B中的0号行

k[1] = 2 表示1号列首个非零元素存储在B中的2号行

k[i] = x 表示i号列首个非零元素存储在B中的x号行

原第0列的非零元素

原第1列的非零元素

原第2列的非零元素

原第3列的非零元素

原第5列的非零元素

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	0	2	3	5	7	7

Step 2: 计算k数组

$$k[i] = \begin{cases} 0 & i = 0 \\ k[i-1] + num[i-1] & i > 0 \end{cases}$$

原第0列的非零元素

原第1列的非零元素

原第2列的非零元素

原第3列的非零元素

原第5列的非零元素

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	0	2	3	5	7	7

Step 2: 计算k数组

```
for (i=1; i<n; i++)
k[i]=k[i-1]+num[i-1];
```

原第0列的非零元素

原第1列的非零元素

原第2列的非零元素

原第3列的非零元素

原第5列的非零元素

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	0	2	3	5	7	7

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素16存放在B的第k[0]=0个位置

元素16存储到B时行列转置

x存放完毕后，对k[x.col]加1，表示该列下一个非零元素在B的位置

0	0	16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	1	2	3	5	7	7

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素22存放在B的第k[3]=5个位置

元素22存储到B时行列转置

k[3]=5+1=6，表示下一个第3列的非零元素在B的存放位置

0	0	16
3	0	22

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	1	2	3	6	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素-16存放在B的第k[5]=7个位置

元素-16存储到B时行列转置

k[5]=7+1=8，表示下一个第5列的非零元素在B的存放位置

0	0	16
3	0	22
5	0	-16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	1	3	3	6	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素12存放在B的第k[1]=2个位置
元素12存储到B时行列转置

k[1]=2+1=3，表示下一个第1列的非零元素在B的存放位置

0	0	16
1	1	12
3	0	22
5	0	-16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	1	3	4	6	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素3存放在B的第k[2]=3个位置

元素3存储到B时行列转置

k[2]=3+1=4，表示下一个第2列的非零元素在B的存放位置

0	0	16
1	1	12
2	1	3
3	0	22
5	0	-16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	1	3	4	6	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素-8存放在B的第k[3]=6个位置

元素-8存储到B时行列转置

k[3]=6+1=7，表示下一个第3列的非零元素在B的存放位置

0	0	16
1	1	12
2	1	3
3	0	22
3	2	-8
5	0	-16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	1	3	4	7	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素91存放在B的第k[0]=1个位置

元素91存储到B时行列转置

k[0]=1+1=2，表示下一个第0列的非零元素在B的存放位置

0	0	16
0	4	91
1	1	12
2	1	3
3	0	22
3	2	-8
5	0	-16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	2	3	5	7	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

元素x存放在B的第k[x.col]个位置

元素15存放在B的第k[2]=4个位置

元素15存储到B时行列转置

k[2]=4+1=5，表示下一个第2列的非零元素在B的存放位置

0	0	16
0	4	91
1	1	12
2	1	3
2	6	15
3	0	22
3	2	-8
5	0	-16

转置三元组

快速转置算法

- 增加适量存储空间，提高时间效率

0	0	16
0	3	22
0	5	-16
1	1	12
1	2	3
2	3	-8
4	0	91
6	2	15

行三元组A

num	2	1	2	2	0	1
k	2	3	5	7	7	8

Step 3: 依次扫描A中各三元式，根据数组k找到三元式在B中的存放位置

对于三元组A中的第i个元素

```
int j=k[A.table[i].col]++;
B.table[j].row= A.table[i].col;
B.table[j].col= A.table[i].row;
B.table[j].value= A.table[i].value;
```

0	0	16
0	4	91
1	1	12
2	1	3
2	6	15
3	0	22
3	2	-8
5	0	-16

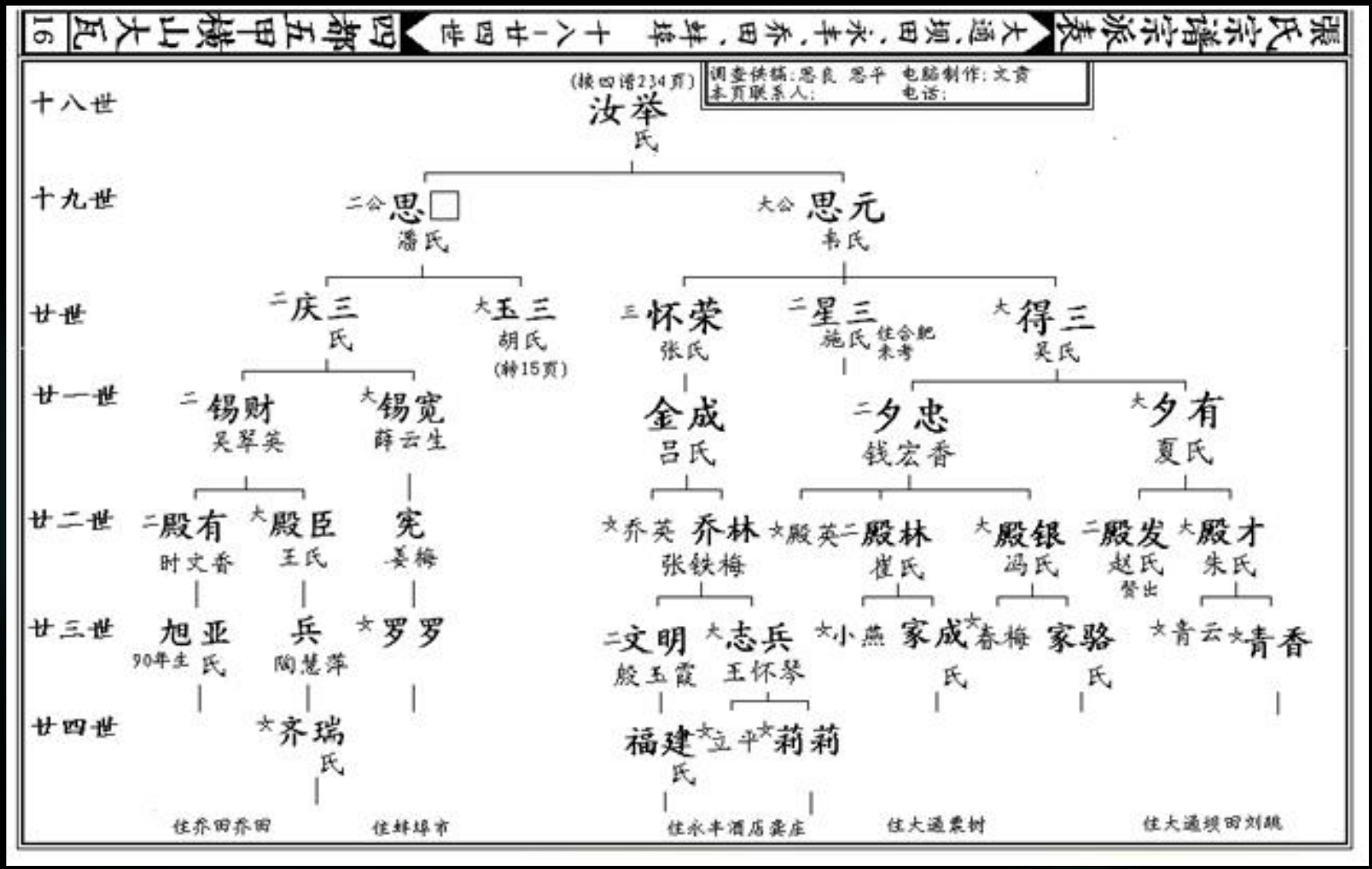
转置三元组

树

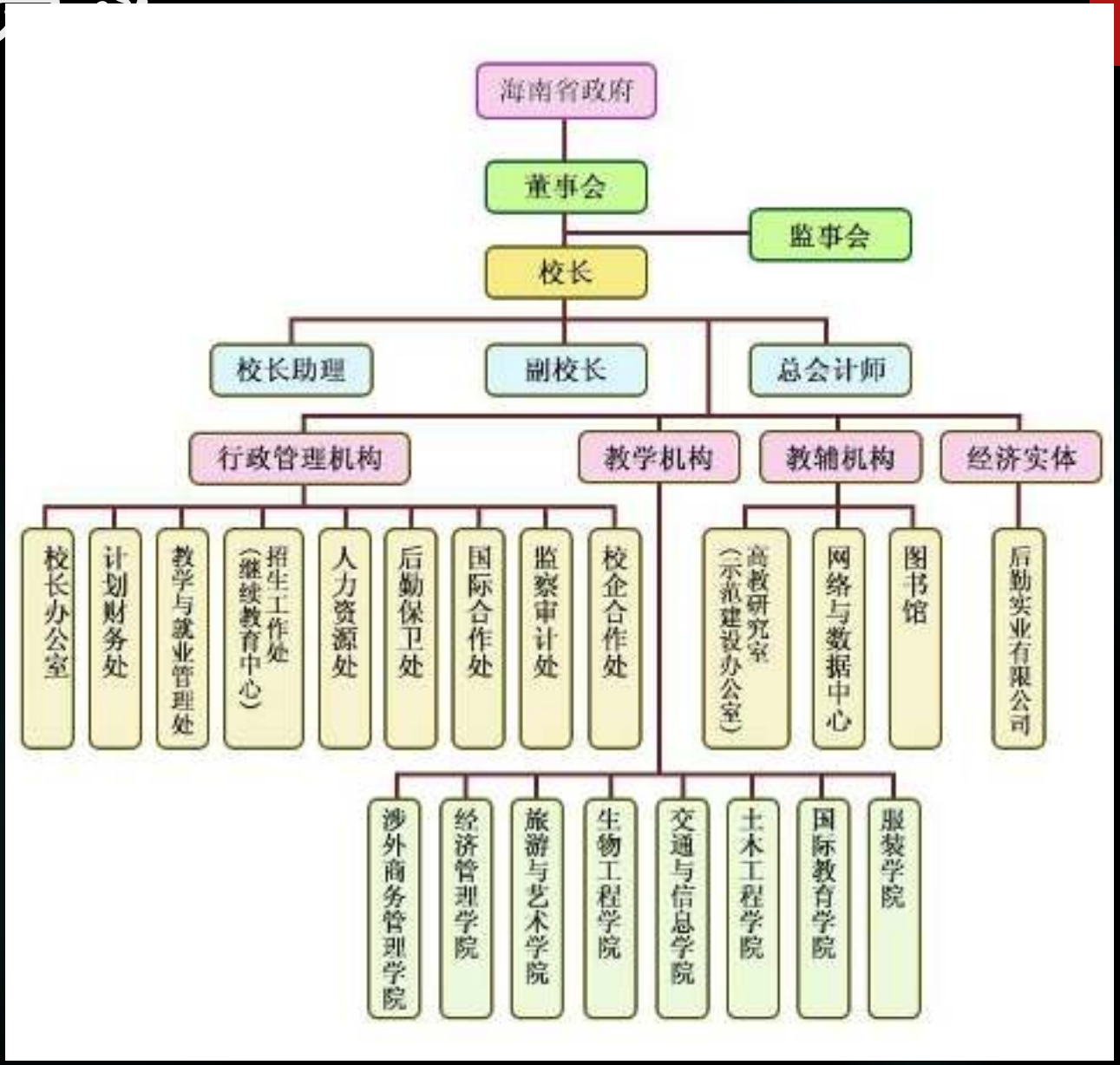
目录

树的定义

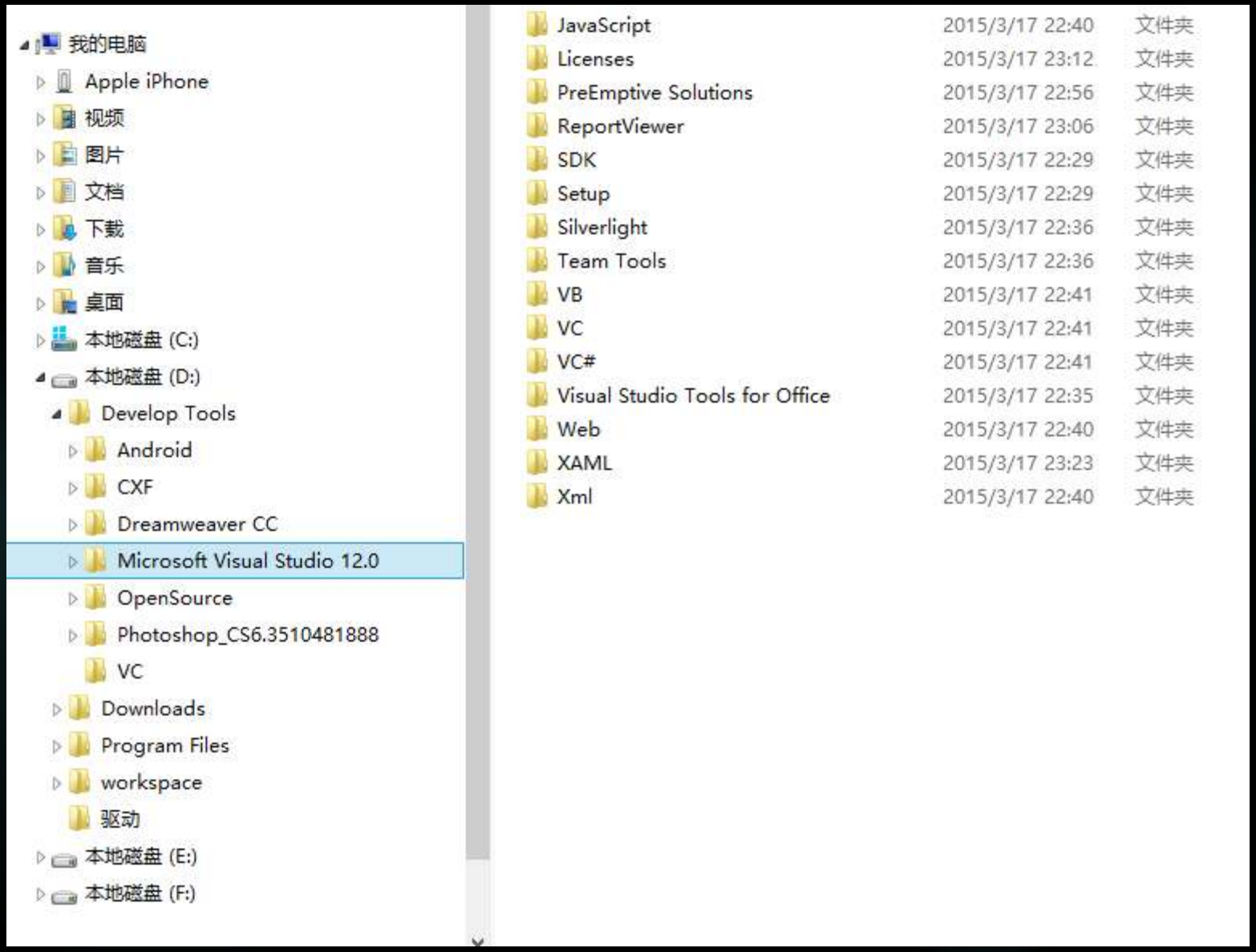
树的定义


十八世
十九世
廿世
廿一世
廿二世
廿三世
廿四世
汝举 氏
思□ 潘氏
思元 韦氏
庆三 氏
玉三 胡氏
怀荣 张氏
星三 施氏
得三 吴氏
锡财 吴翠英
锡宽 薛云生
金成 吕氏
夕忠 钱宏香
夕有 夏氏
殷有 时文香
殷臣 王氏
宪 姜梅
乔英 乔林 张铁梅
殷英 殷林 崔氏
殷银 冯氏
殷发 赵氏
殷才 朱氏
旭亚 90年生 氏
兵 陶慧萍
罗罗
文明 殷玉霞
志兵 王怀琴
家成 氏
家骆 氏
青云
青香
齐瑞 氏
福建 氏
莉莉
住乔田乔田
住蚌埠市
住永丰酒店套座
住大通素村
住大通坝田刘靓

树的定义


海南省政府
董事会
监事会
校长
校长助理
副校长
总会计师
行政管理机构
教学机构
教辅机构
经济实体
校长办公室
计划财务处
教学与就业管理处
招生工作处（继续教育中心）
人力资源处
后勤保卫处
国际合作处
监察审计处
校企合作处
高教研究室（示范建设办公室）
网络与数据中心
图书馆
后勤实业有限公司
涉外商务管理学院
经济管理学院
旅游与艺术学院
生物工程学院
交通与信息学院
土木工程学院
国际教育学院
服装学院

树的定义


我的电脑
Apple iPhone
视频
图片
文档
下载
音乐
桌面
本地磁盘 (C:)
本地磁盘 (D:)
Develop Tools
Android
CXF
Dreamweaver CC
Microsoft Visual Studio 12.0
OpenSource
Photoshop_CS6.3510481888
VC
Downloads
Program Files
workspace
驱动
本地磁盘 (E:)
本地磁盘 (F:)
JavaScript 2015/3/17 22:40 文件夹
Licenses 2015/3/17 23:12 文件夹
PreEmptive Solutions 2015/3/17 22:56 文件夹
ReportViewer 2015/3/17 23:06 文件夹
SDK 2015/3/17 22:29 文件夹
Setup 2015/3/17 22:29 文件夹
Silverlight 2015/3/17 22:36 文件夹
Team Tools 2015/3/17 22:36 文件夹
VB 2015/3/17 22:41 文件夹
VC 2015/3/17 22:41 文件夹
VC# 2015/3/17 22:41 文件夹
Visual Studio Tools for Office 2015/3/17 22:35 文件夹
Web 2015/3/17 22:40 文件夹
XAML 2015/3/17 23:23 文件夹
Xml 2015/3/17 22:40 文件夹

树的定义

- 树（有根树）是包括n（n≥1）个元素的有限非空集合D，R是D中元素的序偶的集合，R满足

  有且只有一个树根结点，该结点没有前驱结点

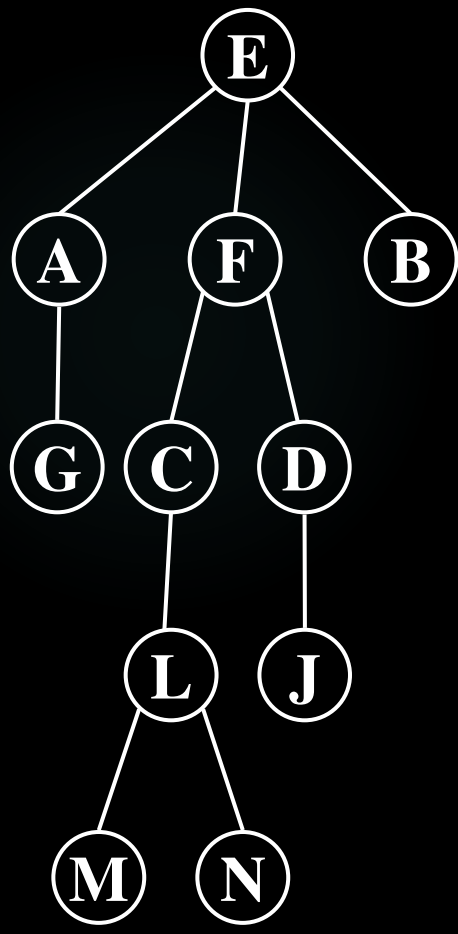
  其他结点，有且只有一个前驱结点

  结点v ∈ D， v≠u，使得<v,u> ∈ R。

- 树不为空集合，树中至少有一个根结点

树的定义

- 树是包括n个结点的有限非空集合T，其中一个特定的结点r称为根，其余结点（T-{r}）划分成m（m≥0）个互不相交的子集T_1，T_2，…T_m，其中，每个子集都是树，被称为树根r的子树。
- 树是递归数据结构：树的定义中引用了树概念本身

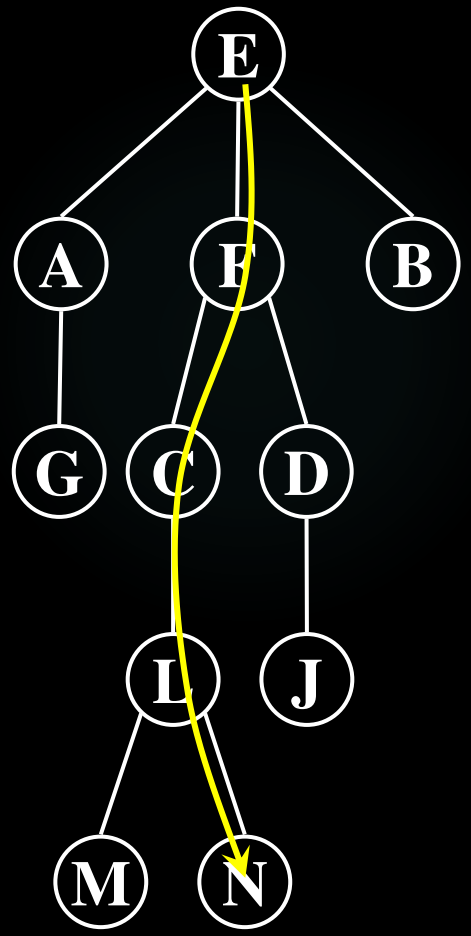


树的定义

结点(node)：树中的元素。

根结点和它的子树根（如果存在）之间形成一条边。

路径(path)：从某个结点沿树中的边可到达另一个结点，则称这两个结点之间存在一条路径。



树的定义

双亲(parent)：若一个结点有子树，那么该结点称为子树**根**的

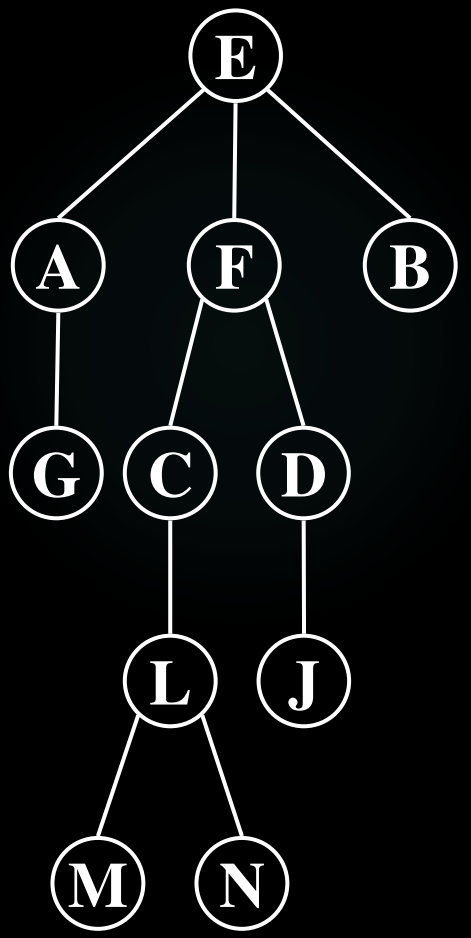
E是A、F、B的双亲

孩子(child)：某结点子树的根是该结点的孩子

C、D是F的孩子

兄弟(sibling)：有相同双亲的结点互为兄弟

M、N是兄弟



结点G和C互为兄弟否？

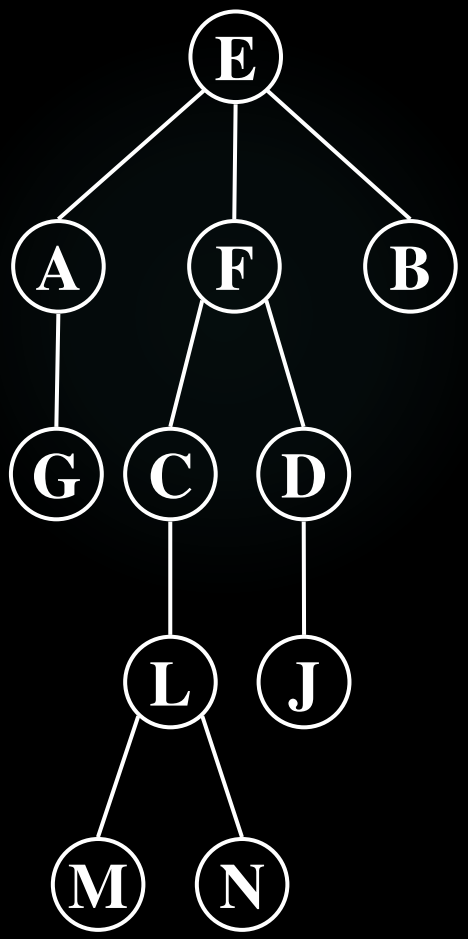
树的定义

后裔(decendent)：一个结点的所有子树上的任何结点都是该结点的后裔。

C的后裔：L、M、N

祖先(ancestor)：从根结点到某个结点的路径上的所有结点都是该结点的祖先。

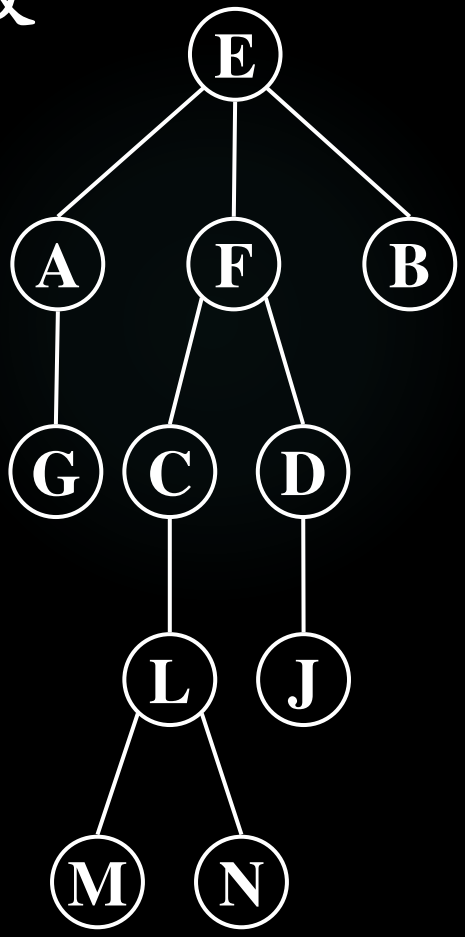
J的祖先：D、F、E



树的定义

结点的度(degree)：结点拥有的子树数

结点E的度为3，结点F的度为2，
结点A的度为1，结点G的度为0。



叶子(leaf)：度为零的结点

B、G、J、M、N均为叶子结点

树的度：树中结点的最大的度

该树的度为3

分支结点(branch)：度不为零的结点。

E、A、F、C等为分支结点

树的定义

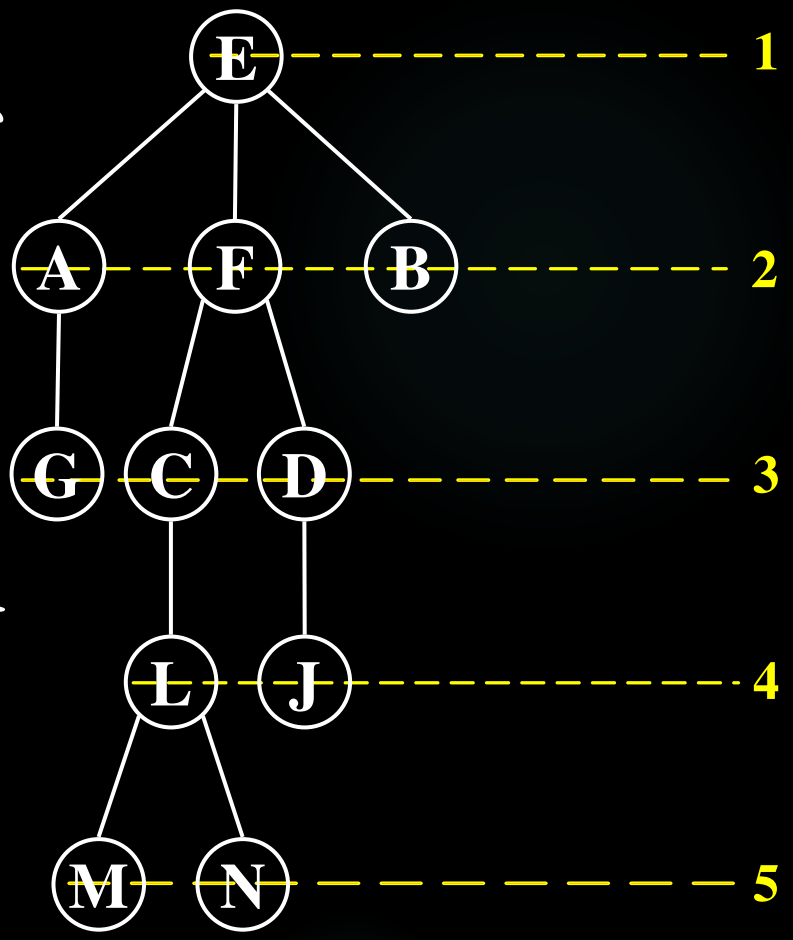
结点的层次：根结点的层次为1，其余结点的层次等于其双亲结点的层次加1

结点E的层次为1

结点M的层次为5

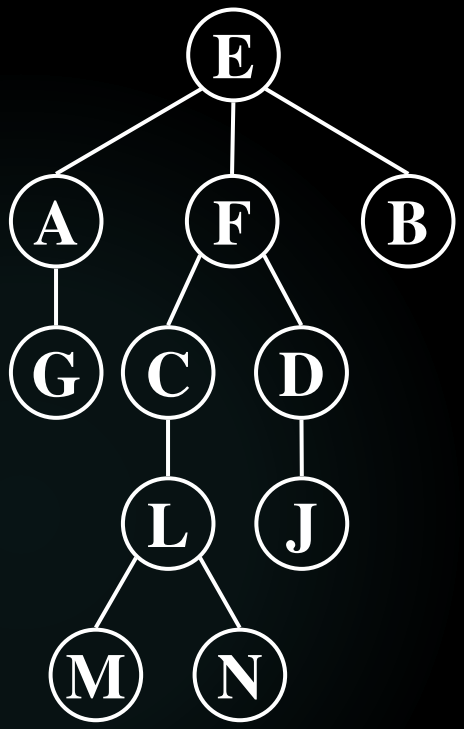
树的高度：树中结点的最大层次

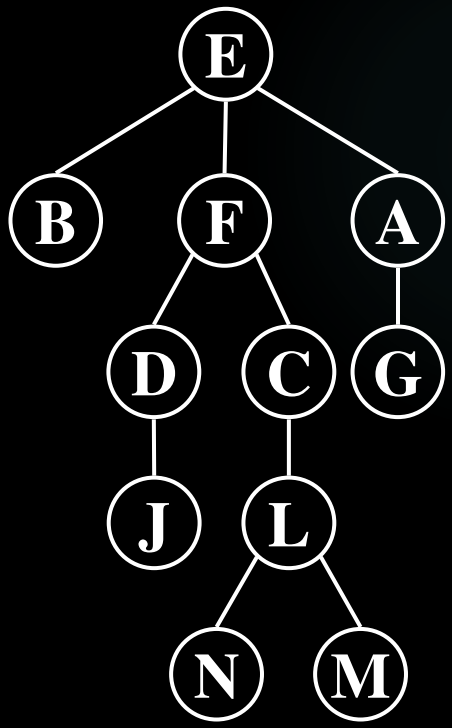
树的高度为5



树的定义

无序树：如果树中结点的各子树之间的次序是不重要的，可以交换位置。

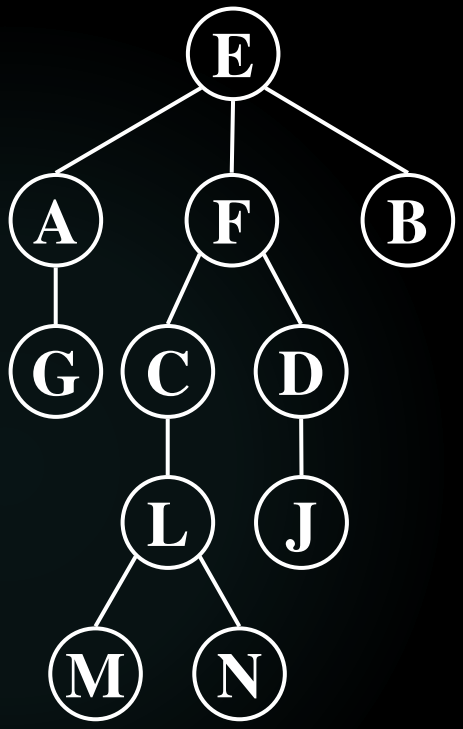


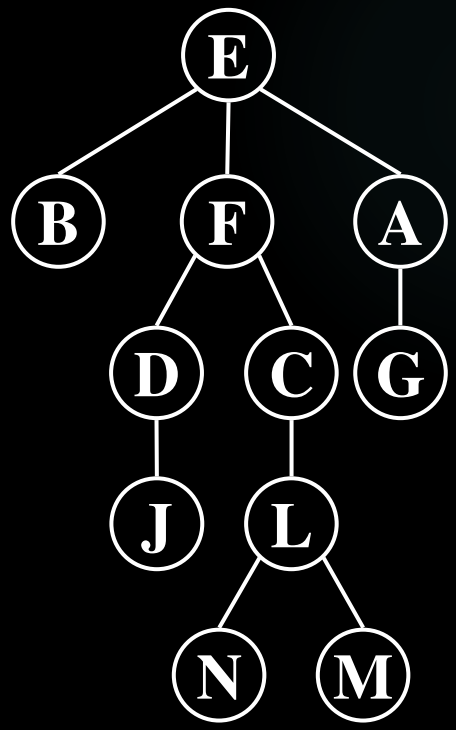


将左边树中所有结点的子树互换次序就是右边的树

树的定义

有序树：如果树中结点的各棵子树看成是从左到右有次序的，则称该树为有序树





有序树的各子树从左到右为第一棵子树、第二棵，…

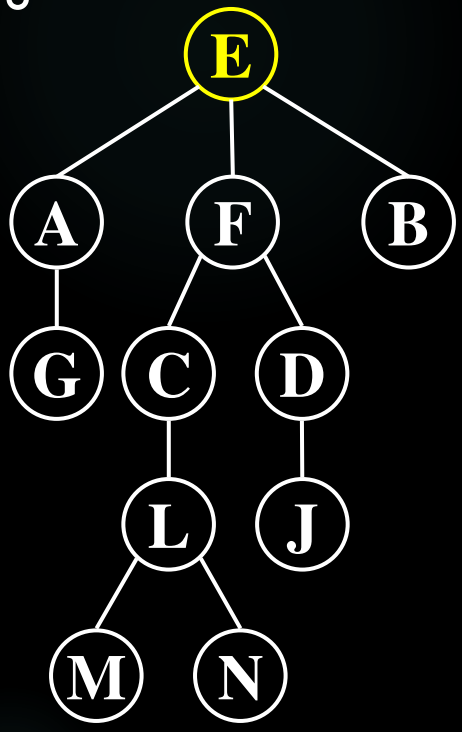
树的定义

森林：是树的集合。0个或多个不相交的树组成森林。

果园或有序森林：有序树的有序集合。

若增加一个结点，将森林中各树的根作为新增结点的孩子，则森林即成为树

将树中的根去掉，则得到根的子树组成的森林。



树

目录

- 树的定义
- 二叉树
- 二叉树的遍历
- 树和森林
- 堆和优先级队列
- 哈夫曼编码

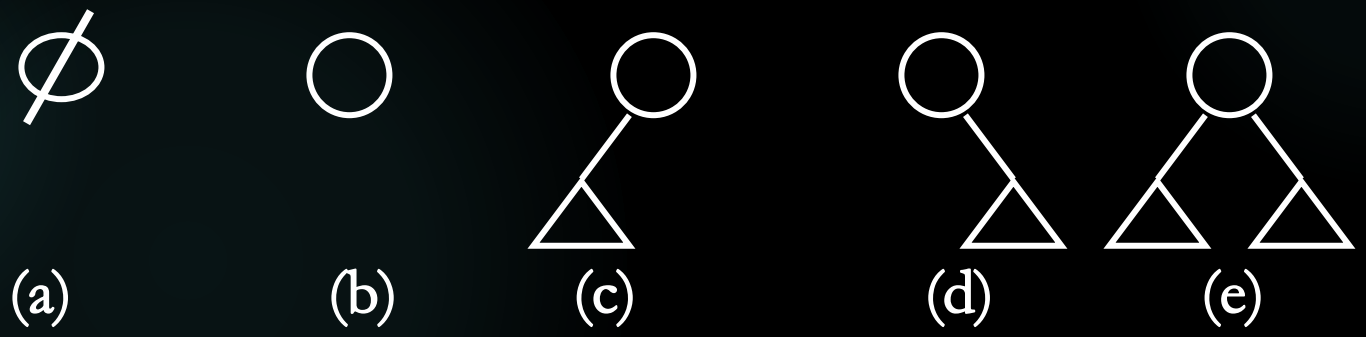
二叉树的定义与性质

二叉树的定义

二叉树(binary tree)结点的有限集合

- 可以为空集
- 可以是由一个根和两个子树组成

  左子树+右子树



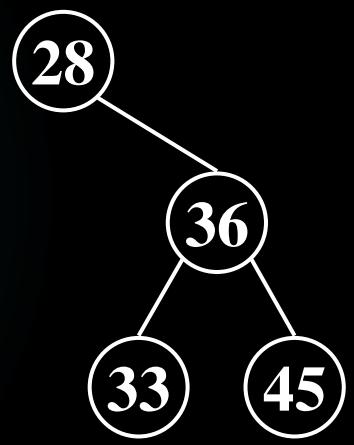
二叉树的五种基本形态

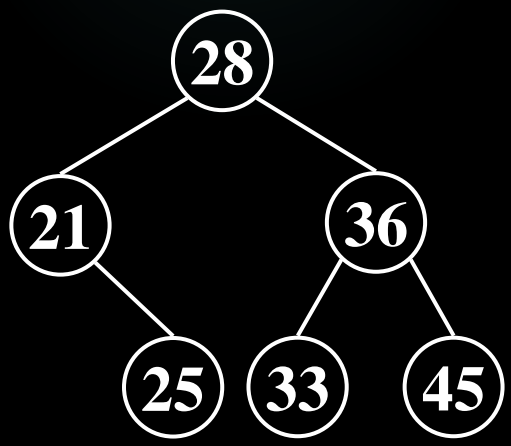
二叉树的定义

二叉树(binary tree)结点的有限集合

- 可以为空集
- 可以是由一个根和两个子树组成

  左子树+右子树

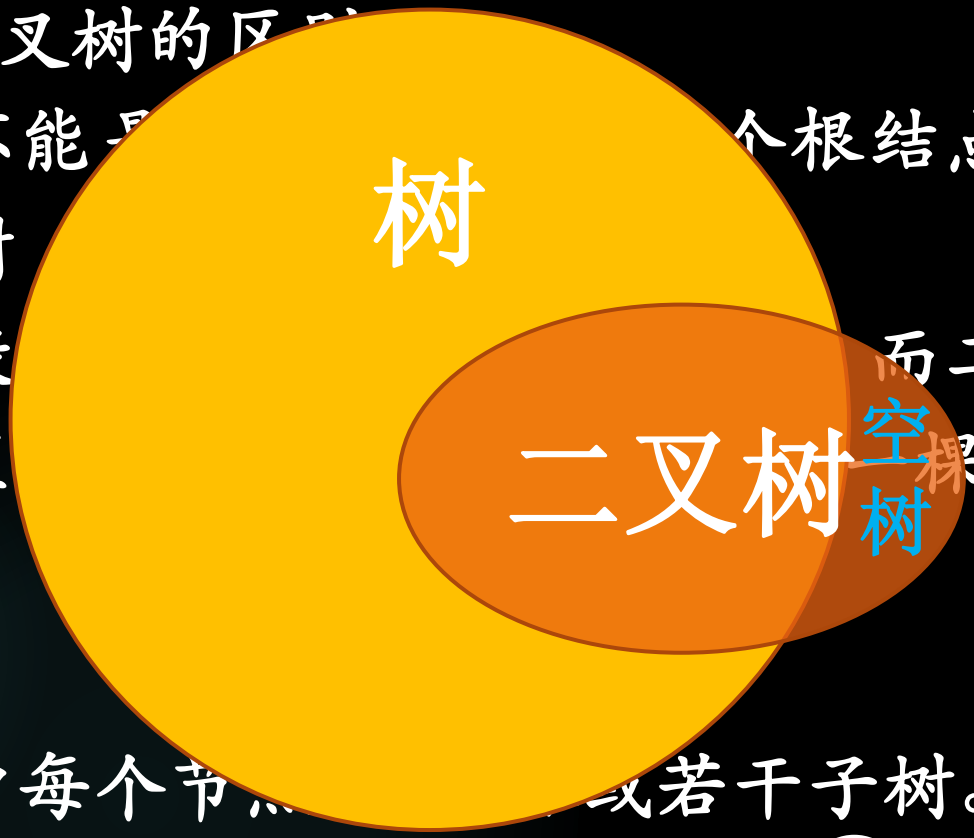




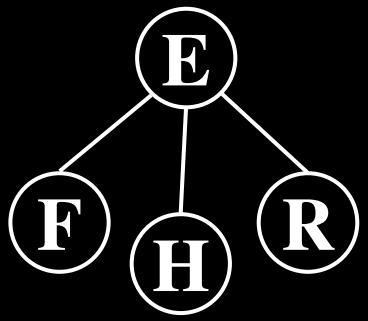
二叉树的定义

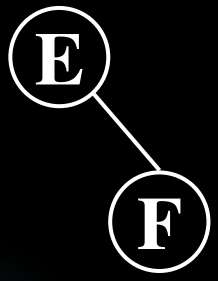
树与二叉树的区别

- 树不能是空树，至少有一个根结点。而二叉树可以是空树
- 一般树中结点的子树没有顺序，而二叉树中结点的子树要区分左子树和右子树



- 树中每个节点可以有任意多个分支或若干子树。而二叉树的每个节点最多只能有2棵子树。





二叉树的定义

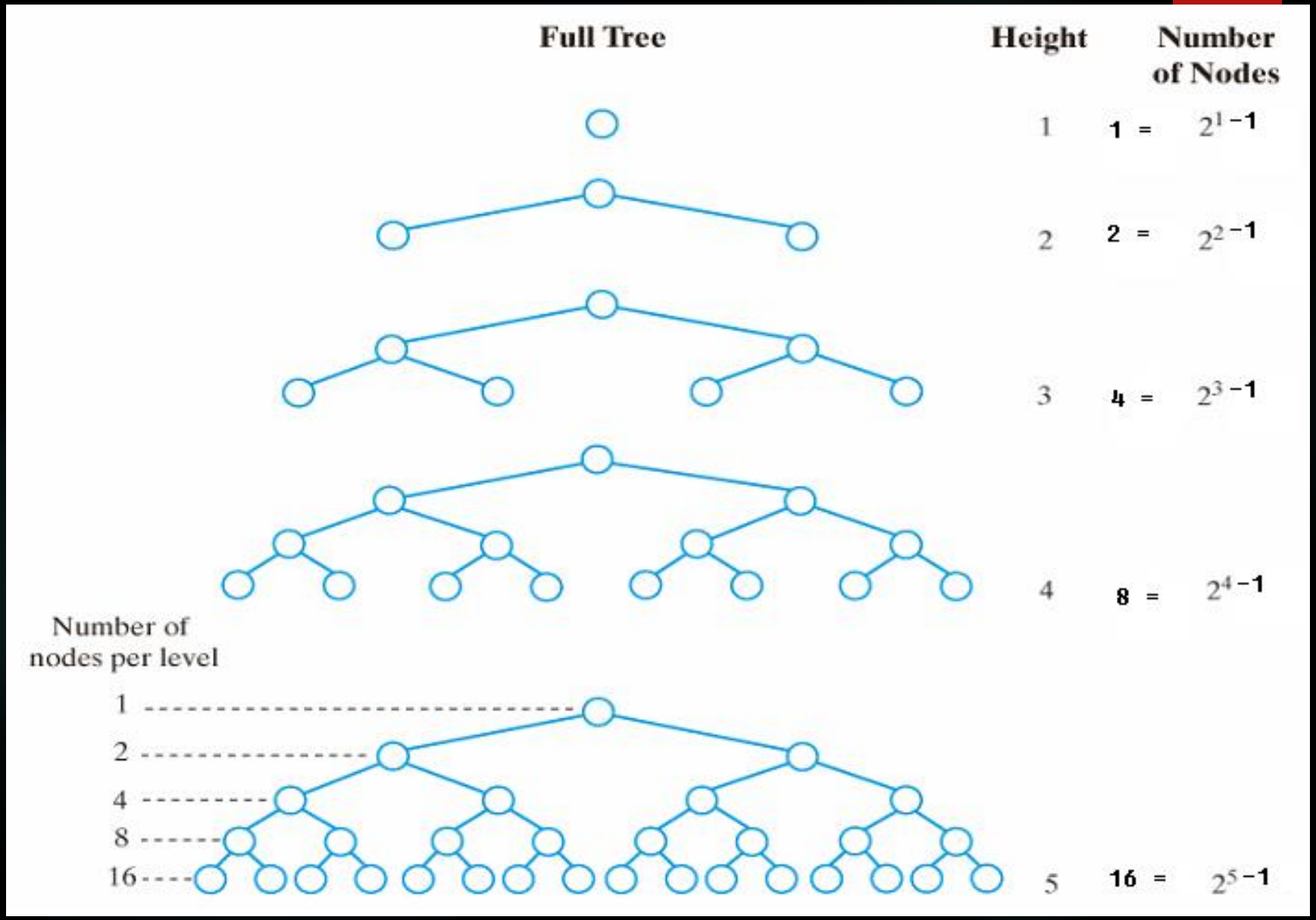
性质1：二叉树的第i(i≥1)层上至多有2^{i-1} 个结点（归纳法证明）

- 当i=1时，二叉树至多只有一个结点，结论成立。
- 设当i=k时结论成立，即二叉树第k层至多有2^{k-1}个结点
- 当i=k+1时

  ∵每个结点最多只有两个孩子，

  ∴第k+1层上至多有$2*2^{k-1}=2^k$个结点，性质成立

性质 二叉树的第i(i≥1)层上至多有2^{i-1}个结点。



二叉树的定义

性质2 高度为h的二叉树上至多有2^h-1个结点。

当h=0时，二叉树为空二叉树。

当h>0时，利用性质1，高度为h的二叉树中结点的总数最多为：

性质1：二叉树的第i(i≥1)层上至多有2^{i-1}个结点

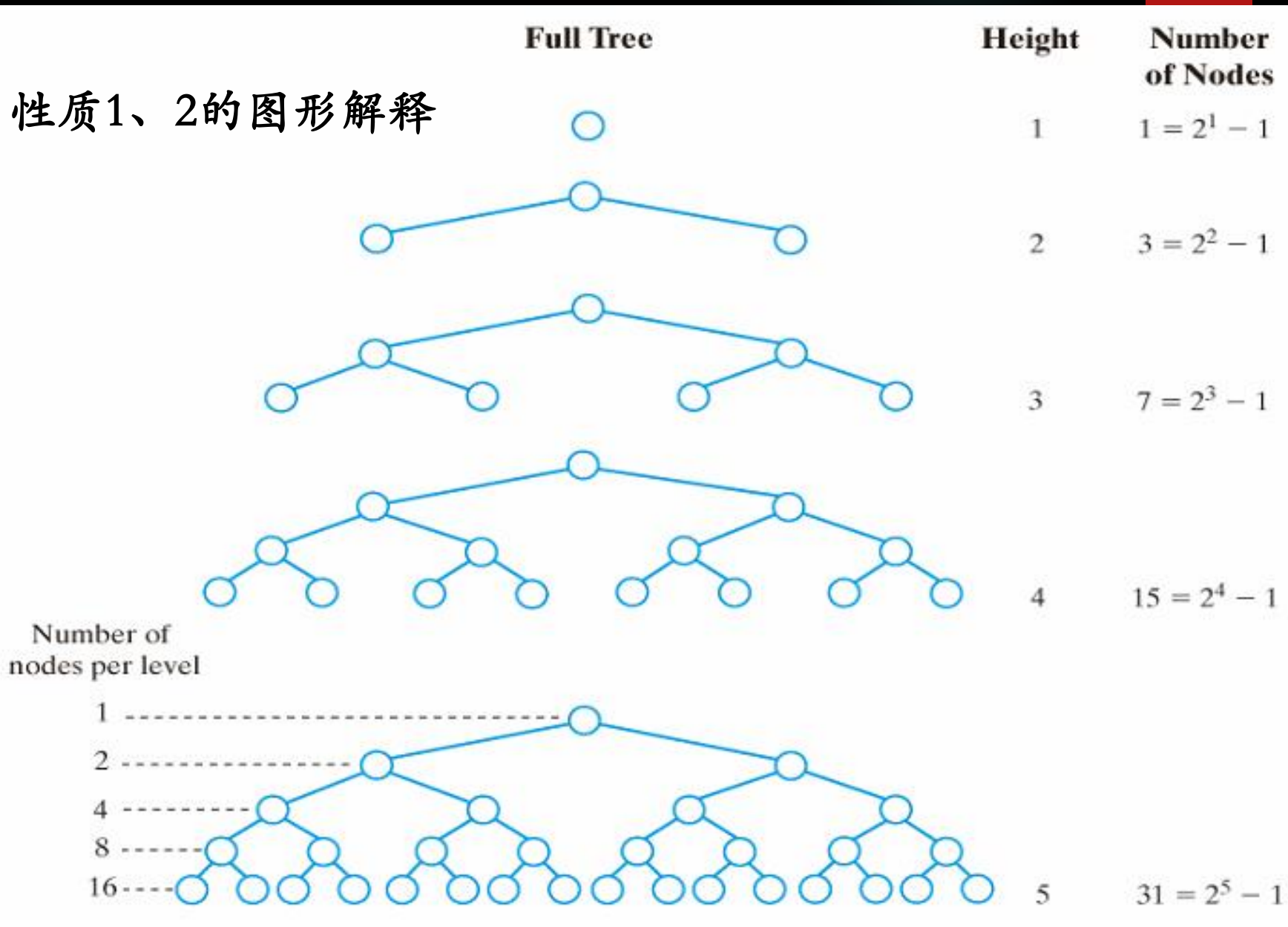
$$\sum_{i=1}^{h} 2^{i-1} = (2^0 + 2^1 + 2^2 + \ldots + 2^{h-1}) = 2^h - 1$$

补充：

等比数列的求和公式是

$$1 + a + a^2 + a^3 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

性质1、2的图形解释


Full Tree
Height
Number of Nodes
1
1 = 2^1 − 1
2
3 = 2^2 − 1
3
7 = 2^3 − 1
4
15 = 2^4 − 1
Number of nodes per level
1
2
4
8
16
5
31 = 2^5 − 1

性质3 任意一棵二叉树中，若叶结点的个数（度为0）为n_0，度为2的结点的个数为n_2，则必有$n_0=n_2+1$

设二叉树的度为1的结点数为n_1，树中结点总数为n，则$n=n_0+n_1+n_2$①（∵二叉树中只有度为0、1、2三种类型的结点）

设分支数为B（树枝），n个结点的二叉树，除了根结点外，每个结点都有一个分支进入，则B=n-1；分支是由度为1或者度为2的射出的，又有$B=2n_2+n_1$；

则有：$n-1=2n_2+n_1$ ➔$n=2n_2+n_1+1$……②

由①②可得到：

$n_0+n_1+n_2=2n_2+n_1+1$ ➔ $n_0+n_2=2n_2+1$ 即$n_2=n_0-1$。

性质4 包含n个元素的二叉树的高度至少为$\lceil \log_2(n+1) \rceil$

根据性质2，高度为h的二叉树最多有$2^h - 1$个结点，因而$n \leq 2^h - 1$，则有$h \geq \log_2(n+1)$。

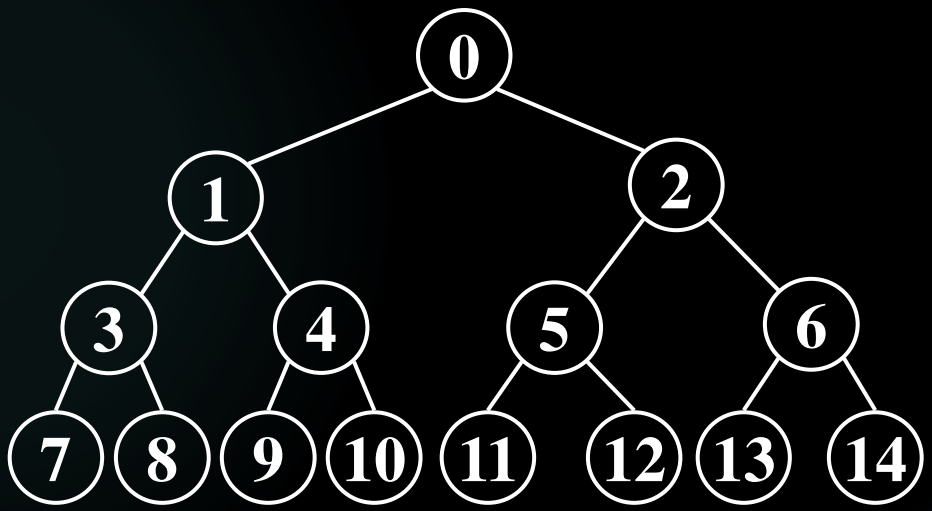
由于h是整数，所以$h \geq \lceil \log_2(n+1) \rceil$。

特殊的二叉树

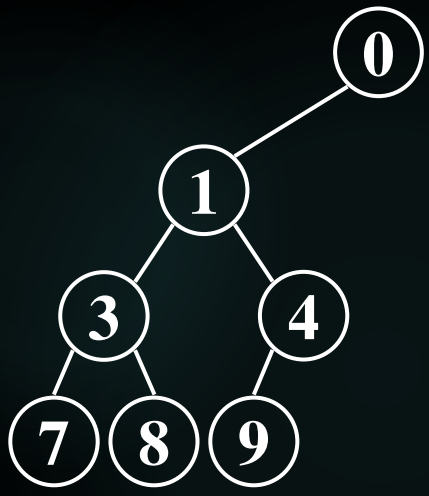
- 满二叉树
- 完全二叉树
- 2-树

定义 高度为h的二叉树恰好有2^h-1个结点时称为**满二叉树**

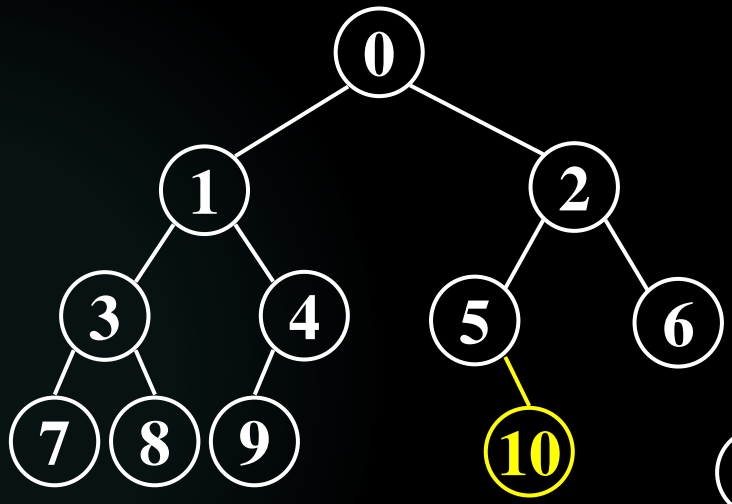
性质2 高度为h的二叉树上**至多**有2^h-1个结点。



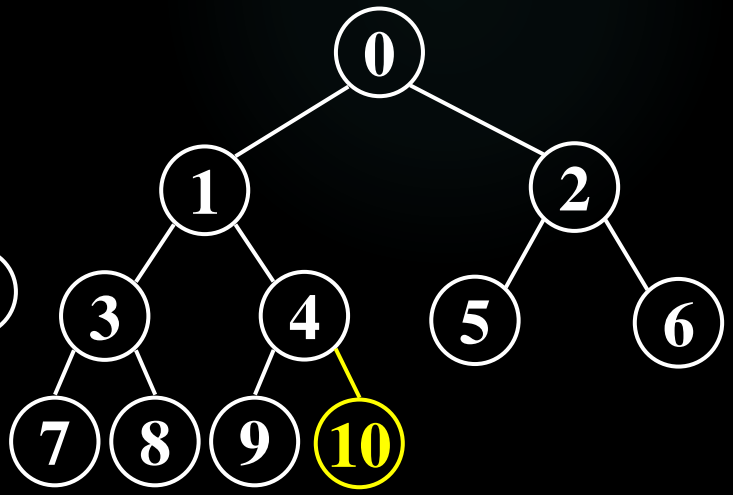
定义 一棵二叉树中，只有**最下面两层结点**的度**可以**小于2，并且最下一层的**叶结点**集中在**靠左**的若干位置上。这样的二叉树称为**完全二叉树**



非完全二叉树

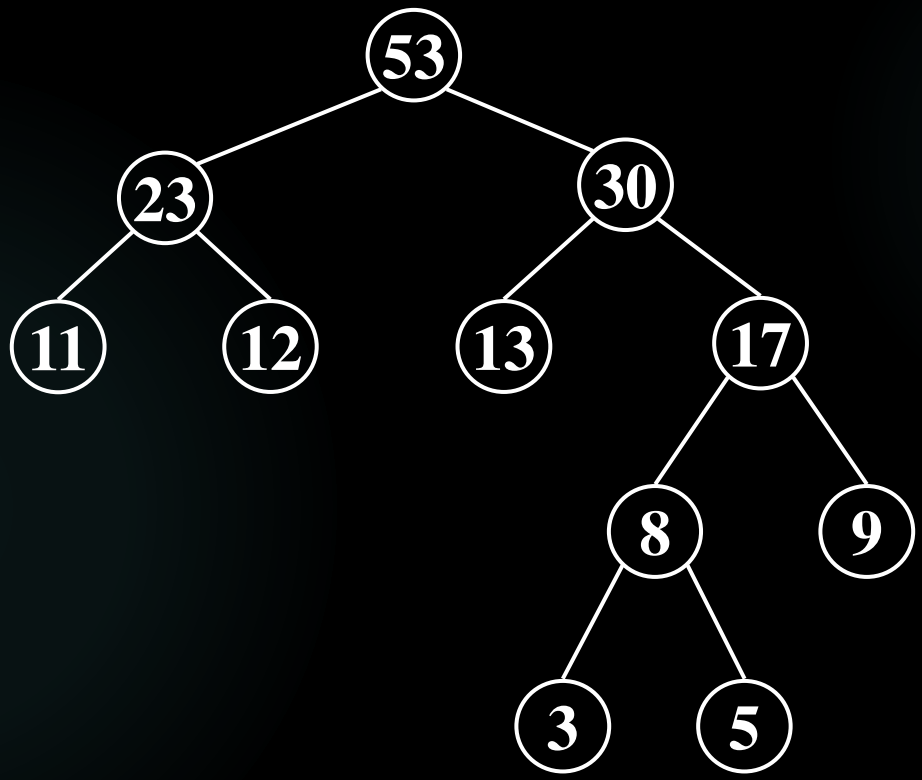


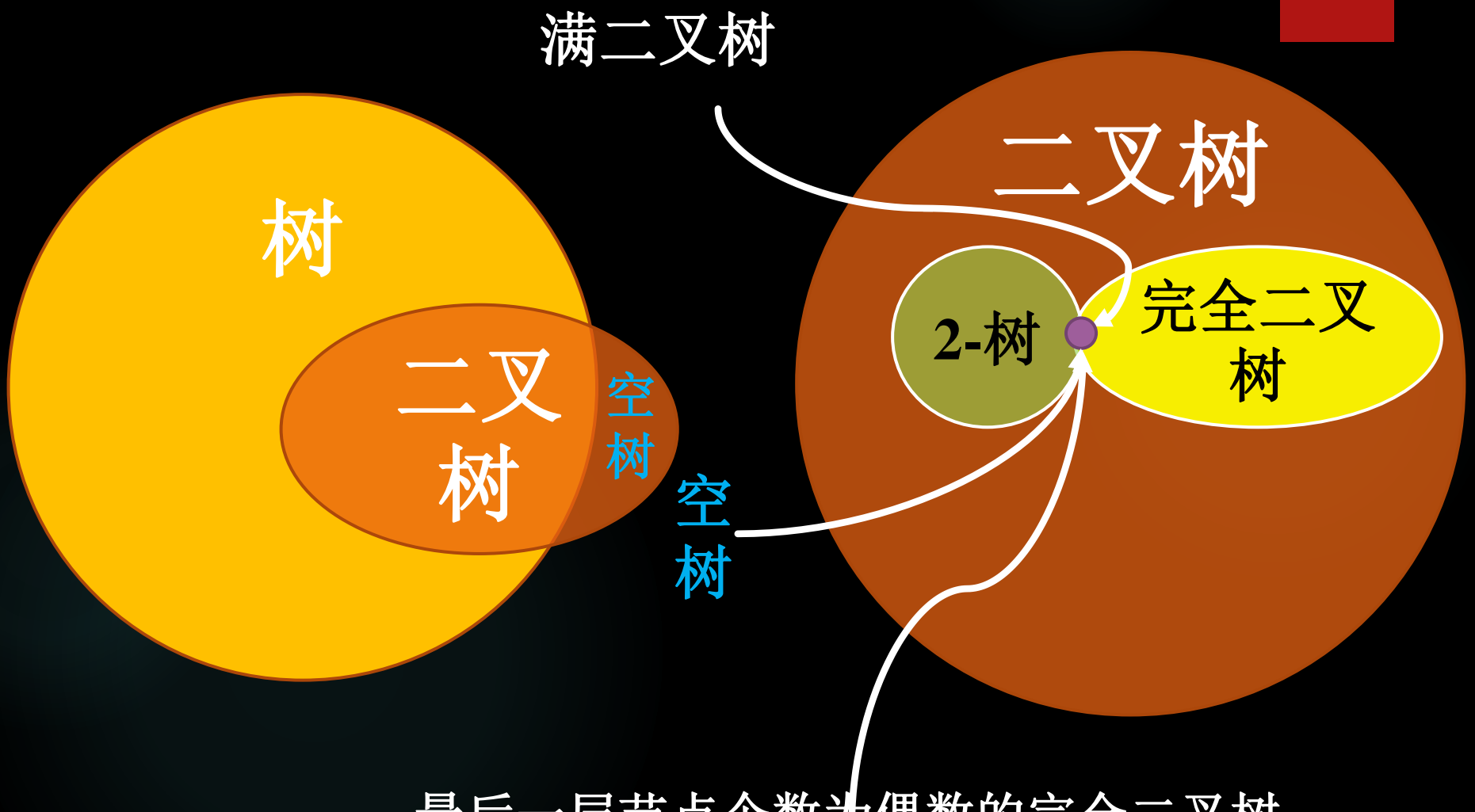
非完全二叉树



完全二叉树

定义 扩充二叉树也称2-树，其中除叶子结点外，其余结点都必须有两个孩子。

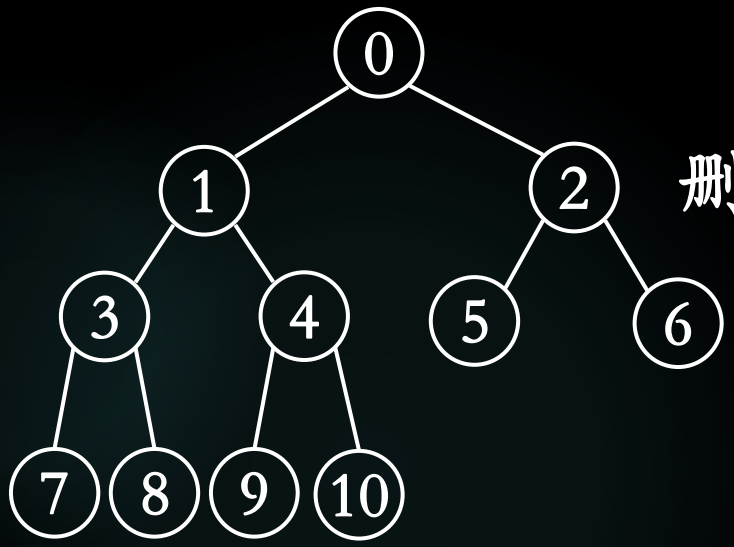



满二叉树
树
二叉树
空树
空树
二叉树
2-树
完全二叉树
最后一层节点个数为偶数的完全二叉树

完全二叉树的性质

完全二叉树的性质

性质 具有n个结点的完全二叉树的高度为 $\lceil \log_2(n+1) \rceil$



根据完全二叉树定义：除最后两层外，其他层结点都是满度（2）

删除最后一层，得到的是一个满树

高度为h的二叉树恰好有 $2^h - 1$ 个结点时称为**满二叉树**

删除最后一层，剩下结点树是 $2^{h-1} - 1$

二叉树的第i(i≥1)层上至多有 2^{i-1} 个结点

h层结点数 $[1, 2^{h-1}]$

完全二叉树结点个数 $2^{h-1} \leqslant n \leqslant 2^h - 1$

完全二叉树结点个数 $2^{h-1} \le n \le 2^h - 1$

$\log_2(n+1) \le h \le \log_2 n+1$

h是整数

$\lceil \log_2(n+1) \rceil \le h \le \lfloor \log_2 n + 1 \rfloor$

$h = \lceil \log_2(n+1) \rceil = \lfloor \log_2 n + 1 \rfloor$

n个结点的二叉树中完全二叉树最矮

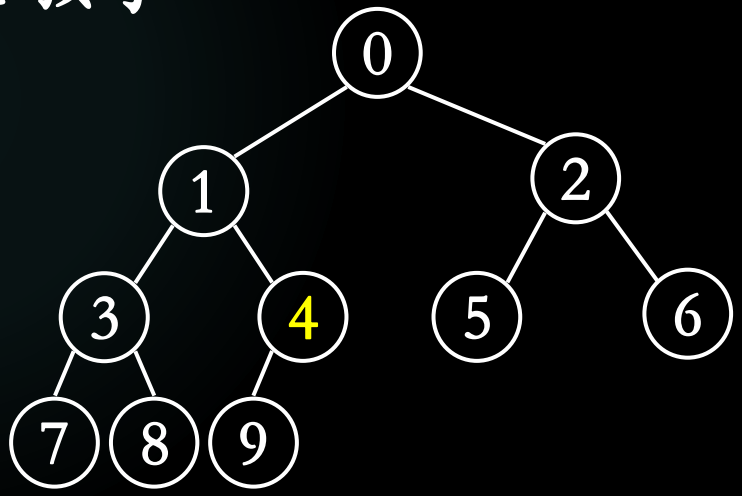
性质5 假定对一棵有n个结点的完全二叉树中的结点，按从上到下、从左到右的顺序，从0到n-1编号，设结点序号为i，则有以下关系成立：

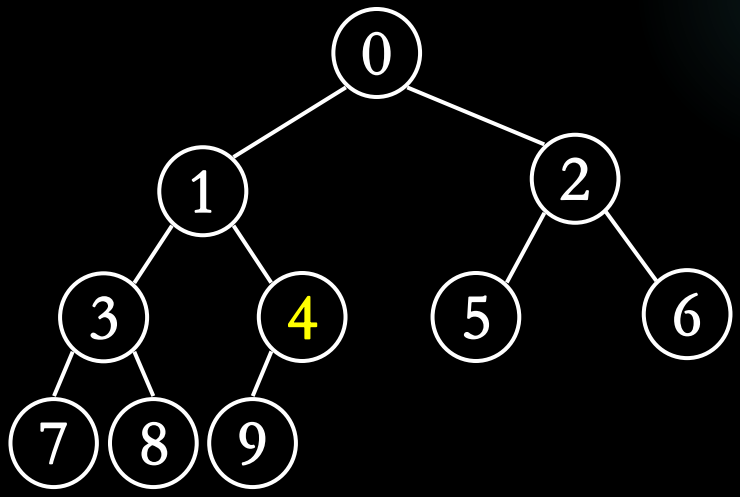
（1）当i=0时，该结点为二叉树的根。

（2）若i>0，则该结点的双亲的序号为$\lfloor (i-1)/2 \rfloor$

（3）若2i+1＜n，则该结点左孩子的序号为2i+1，否则该结点无左孩子

（4）若2i+2＜n，则该结点右孩子的序号为2i+2，否则该结点无右孩子





设结点*i*的层号是*k*，*k*层结点编号范围

$$2^{k-1}-1 \le i \le 2^k-2$$

设结点*i*的孩子编号取决于

- 与*i*同层，在i之后结点个数$af(i)=2^k-2-\mathrm{i}$
- 与*i*同层，在i之前结点的孩子个数

$$bc(i)=2\times(\mathrm{i}-2^{k-1}+1)$$

*i*的左孩子编号=*i*+*af*(*i*)+*bc*(*i*)+1=2*i*+1

二叉树的抽象数据类型

ADT BinaryTree {

数据：

二叉树是结点的有限集合，它或者为空，或者由一个根结点和左、右子二叉树组成。

运算：

CreateBT(BTree *Bt);

创建运算: 构造一个空二叉树。

BOOL IsEmpty(BTree bt)

判空运算: 若二叉树为空，则返回TRUE，否则返回FALSE。

void MakeBT(BTree * Bt，K x, BTree *Lt，BTree *Rt)

构造运算：构造一棵二叉树*Bt，x为*Bt的根结点的元素值，*Lt成为*Bt的左子树，*Rt成为右子树，*Lt和*Rt都成为空二叉树。

void BreakBT(BTree *Bt，K *x, BTree *Lt，BTree *Rt)
拆分运算：二叉树非空时，拆分二叉树*Bt成为三部分，*x为根结点的值，*Lt为原*Bt的左子树，*Rt为原*Bt的右子树，*Bt自身成为空二叉树。

BOOL Root(BTree Bt, K * x)

取根运算：若二叉树非空，则*x为根结点的值，返回TRUE，否则返回FALSE。

void InOrder(BTree Bt, BTNode* u)

中序遍历运算：中序遍历二叉树Bt。

void PreOrder(BTree Bt, BTNode* u)

先序遍历运算：先序遍历二叉树Bt。

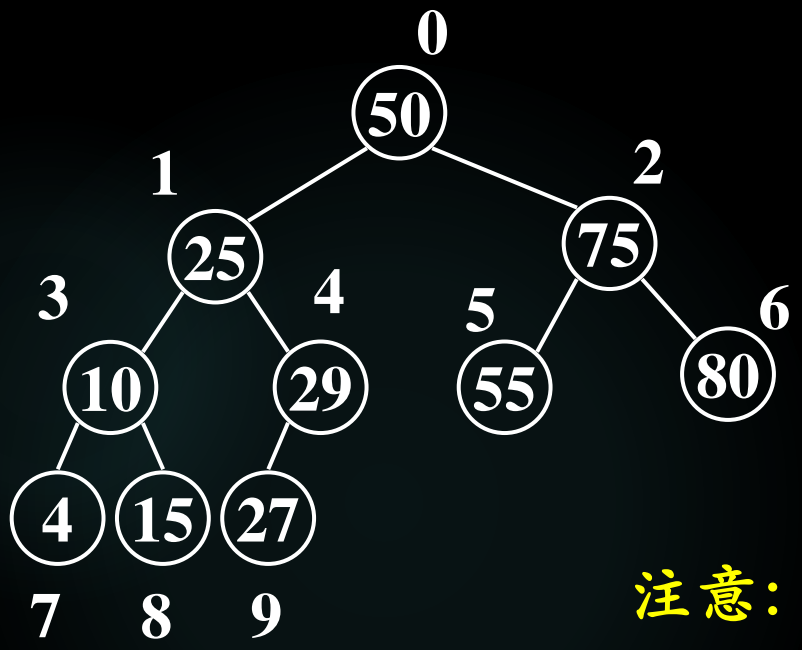
void PostOrder(BTree Bt, BTNode* u)

后序遍历运算：后序遍历二叉树Bt。

二叉树的存储表示

完全二叉树的顺序表示

完全二叉树中的结点可以按层次顺序存储在一片连续的存储单元中。根结点保存在编号为0的位置上。



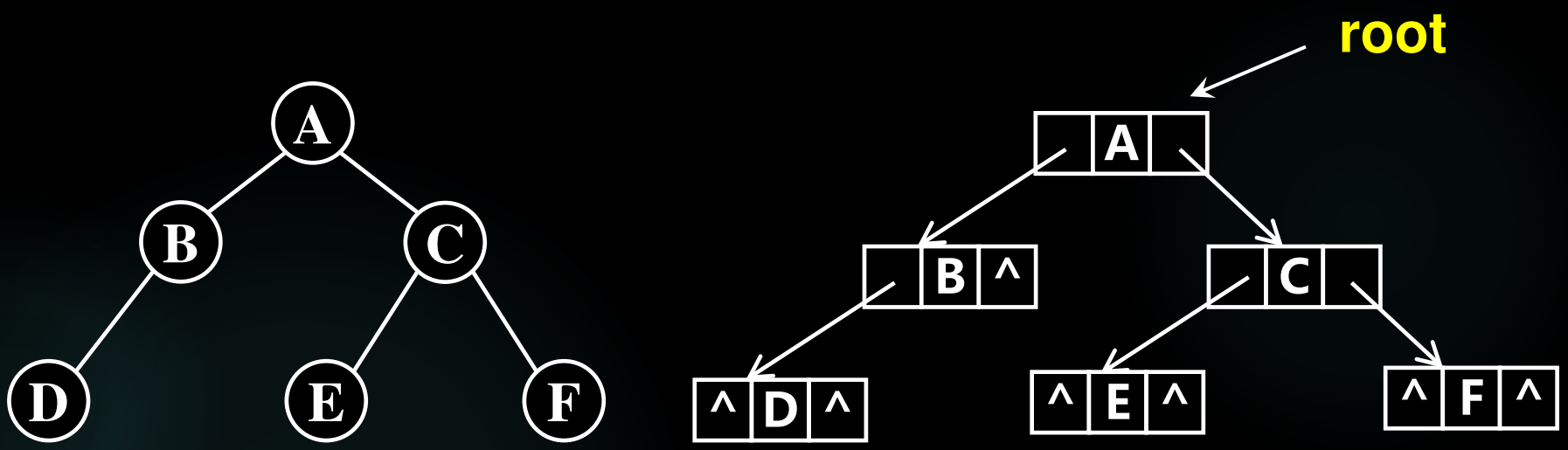
0	1	2	3	4	5	6	7	8	9
50	25	75	10	29	55	80	4	15	27

注意：一般的二叉树不适合用这种存储结构

WHY？

二叉树的链接表示

leftChild	value	rightChild



二叉树的二叉链表结构有利于从双亲到孩子方向的访问。采取从根开始，遍历整个二叉树

二叉树的链接表示

value	parent	leftChild	rightChild

- 如果应用程序需要经常执行从孩子到双亲访问，可在二叉链表结点中增加一个parent域，令它指向该结点的双亲结点。这就实现了从孩子到双亲，以及从双亲到孩子的双向链接结构，形成多重链表。

链接存储的二叉树实现

BTNode是二叉树的结点类型，BTree为二叉树类型。

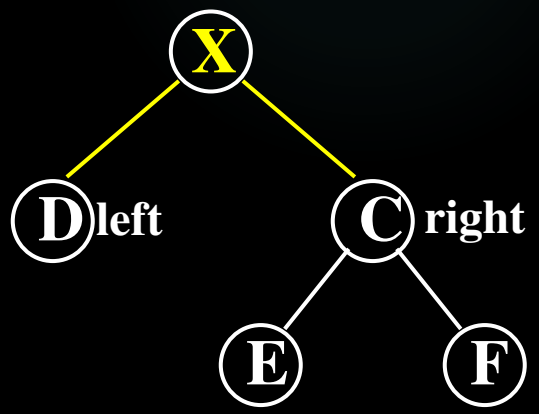
```
typedef struct btnode
{
    T Element;
    struct btnode* Lchild, *Rchild;
}BTNode;

typedef  struct btree{
    struct btnode* Root;
}BTree;
```

链接存储的二叉树的基本运算

函数CreateBT的目的是构造一棵空二叉树，所以只需让二叉树结构中指向根结点的指针Root为空指针即可，通过执行语句Bt->Root=NULL; 来实现。

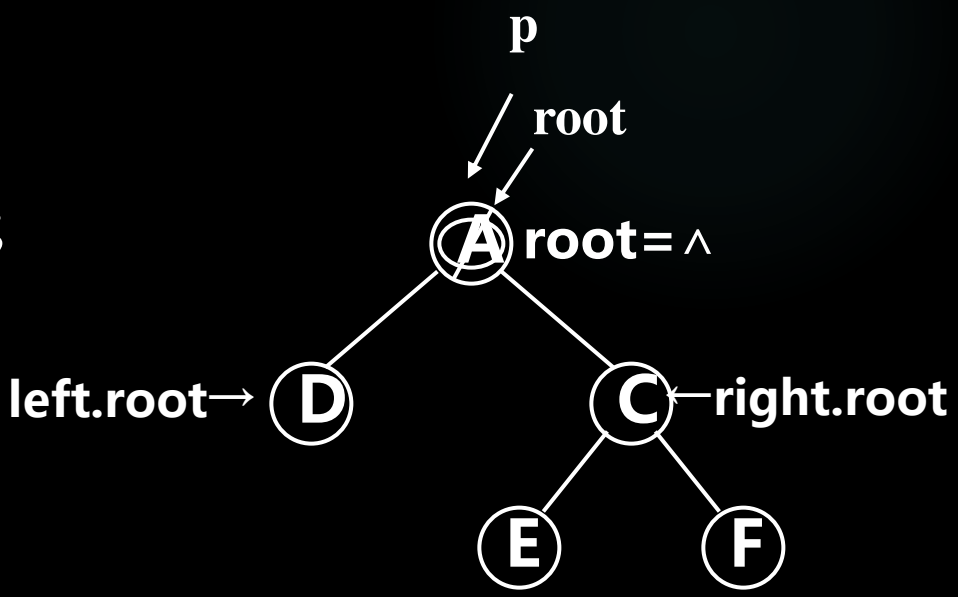
```
void CreateBT(BTree * Bt)
{
        Bt->Root=NULL;
}
```

```
void MakeBT(BTree* Bt, T x, BTree *Lt, BTree *Rt)
{
    BTNode* p＝NewNode( );
    p->Element=x;
    p->LChild=Lt->Root;
    p->RChild=Rt->Root;
    Lt->Root=Rt->Root=NULL;
        Bt->Root=p;
}
```



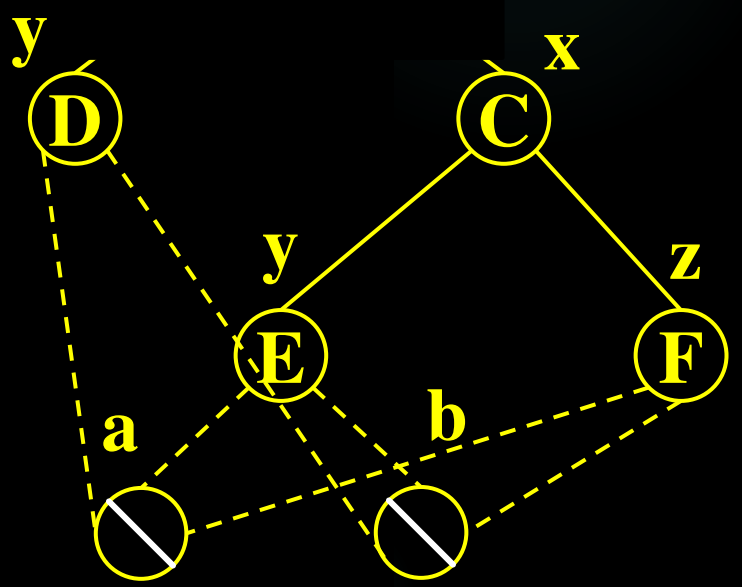
```
BTNode* NewNode( )
{
   BTNode* p=(BTNode*)malloc(sizeof(BTNode));
   if (!p){
      fprintf(stderr, "The memeny is full\n");
       exit(1);
   }
   return p;
}
```

```
void BreakBT(BTree* Bt, T *x, BTree *Lt, BTree *Rt)
{
    BTNode * p=Bt->Root;
    if(p){
       *x=p->Element;
       Lt->Root=p->LChild;
       Rt->Root=p->RChild;
       Bt->Root=NULL;
       free(p);
    }
}
```



一个测试程序

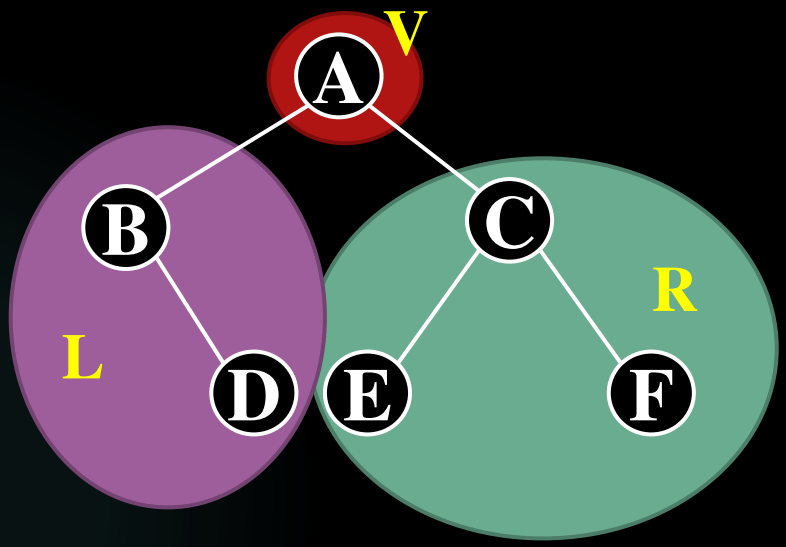
```
int main(void)
{ Btree a, b, x, y, z;  char e;
  CreateBT(&a); CreateBT(&b);
  CreateBT(&x); CreateBT(&y);
  CreateBT(&z);
  MakeBT(&y, 'E', &a, &b);
  MakeBT(&z, 'F', &a, &b);
  MakeBT(&x, 'C', &y, &z);
  MakeBT(&y, 'D', &a, &b);
  MakeBT(&z, 'B', &y, &x);
  BreakBT(&z, &e, &y, &x);
  return 0;}
```



目录

- 树的定义
- 二叉树
- 二叉树的遍历
- 树和森林
- 堆和优先级队列
- 哈夫曼编码

遍历（traverse）一个有限结点的集合，意味着对该集合中的每个结点访问且仅访问一次。



二叉树遍历算法:

(I) 先左后右：VLR，LVR，LRV

(II)先右后左：VRL，RVL，RLV

(1) 先序遍历（VLR）

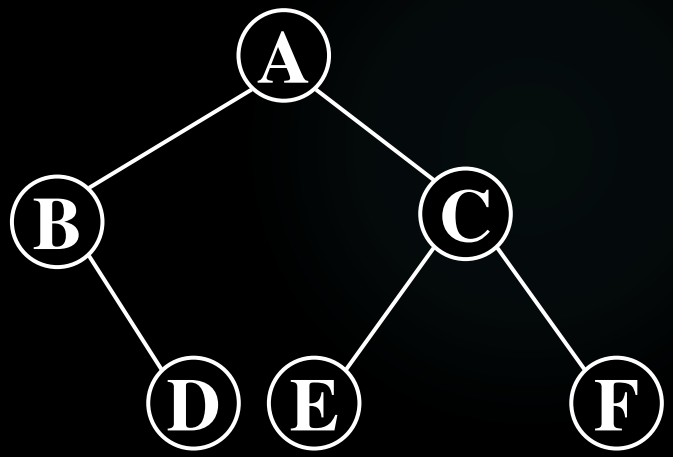
IF二叉树为空，则什么也不做

ELSE

访问根结点；

先序遍历（左子树）；

先序遍历（右子树）。



先序遍历序列：A B D C E F

(2) 中序遍历（LVR）

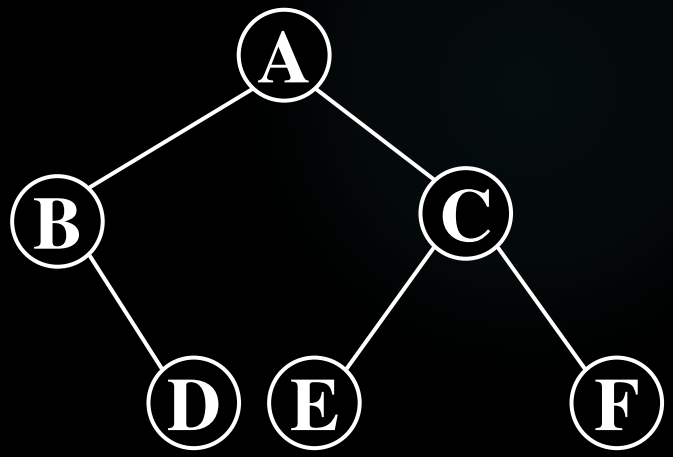
IF二叉树为空，则什么也不做；

ELSE

中序遍历（左子树）；

访问根结点；

中序遍历（右子树）。



中序遍历序列：B D A E C F

(3) 后序遍历 （LRV）

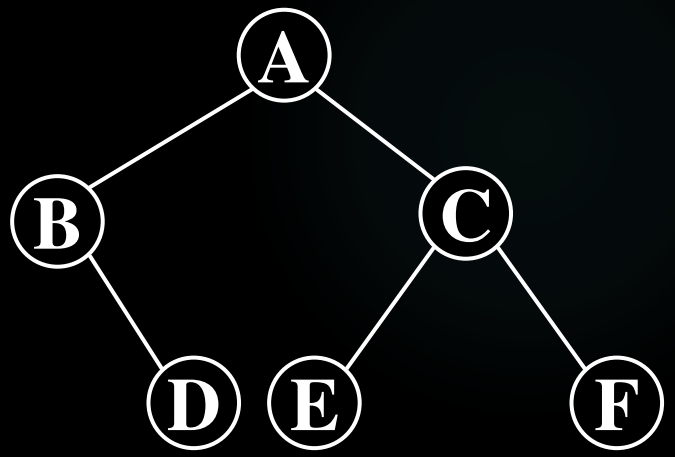
IF二叉树为空，则什么也不做；

ELSE

后序遍历（左子树）；

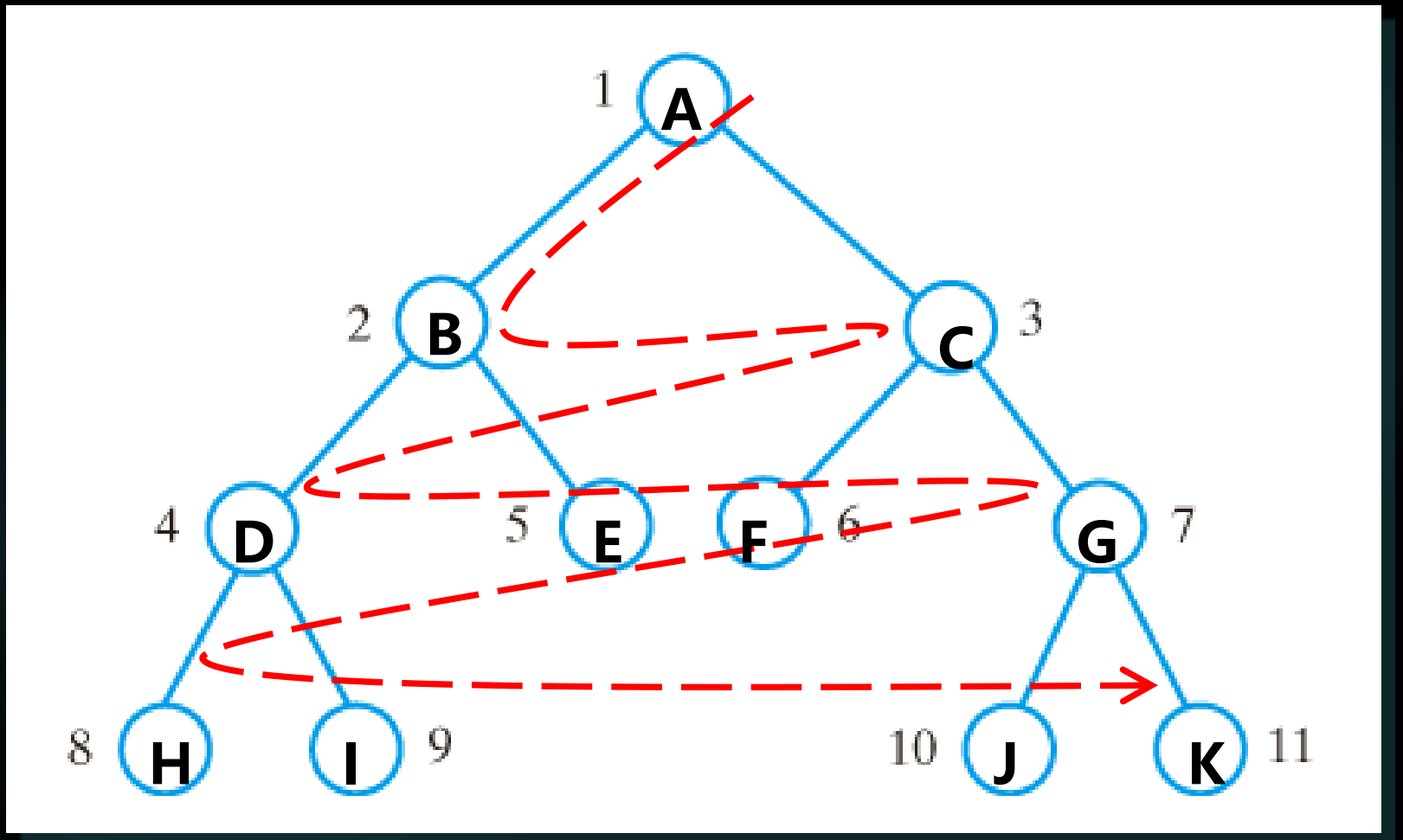
后序遍历（右子树）；

访问根结点。



后序遍历序列：D B E F C A

层次遍历



二叉树遍历的递归算法

对于遍历运算，设计了一个面向用户的主要函数和一个具体实现遍历操作的递归函数，两者共同完成遍历运算的功能。

主要函数：非递归函数，作为与用户的接口。它调用递归函数。

递归函数：具体实现遍历操作。被主要函数调用。

程序 访问元素函数

```
void Visit(BTNode * p)
{
   printf("%c ", p->Element);
}
```

程序 先序遍历

```
void PreOrd(BTNode *t) //递归函数
{
      if (t){
          Visit(t);
          PreOrd(t->LChild);
          PreOrd(t->RChild);
      }
 }
void PreOrder(BTree Bt) //主要函数
 {
       PreOrd(Bt.Root);
 }
```

中序遍历

```
void InOrd(BTNode *t)
 {
       if (t){
          InOrd(t->LChild);
          Visit(t);
          InOrd(t->RChild);
       }

}
void InOrder(BTree Bt)
 {
       InOrd(Bt.Root);
 }
```

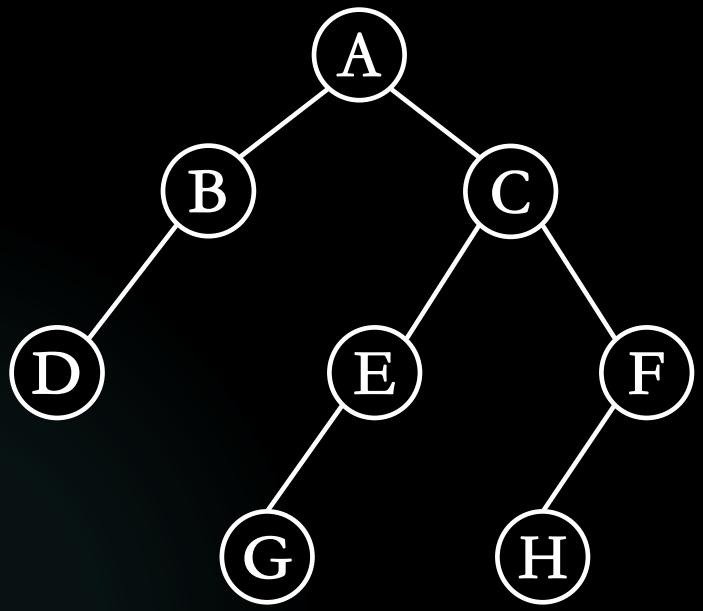
后序遍历

```
void PostOrd(BTNode*t)
 {
       if (t){
          PostOrd(t->LChild);
          PostOrd(t->RChild);
           Visit(t);
       }
   }
void PostOrder(BTree Bt)
 {
       PostOrd(Bt.Root);
 }
```

显然，二叉树遍历算法基本操作是访问结点，不论按何种次序遍历，对含有n个结点的二叉树，其时间复杂度均为O(n)。

关于三种遍历算法对大家的学习要求

给定一棵二叉树，能写出它的三种遍历序列



先序遍历序列：A B D C E G F H

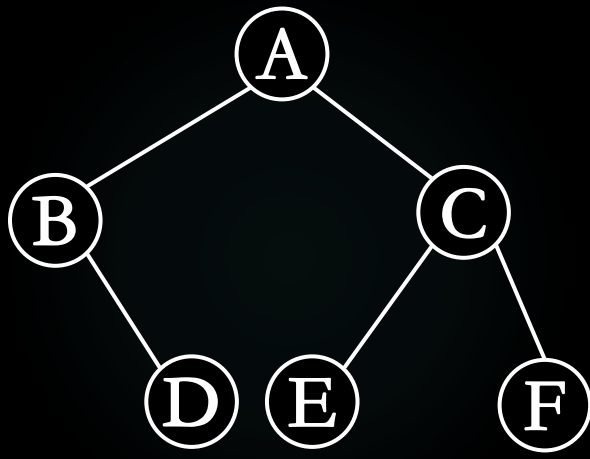
中序遍历序列：D B A G E C H F

后序遍历序列：D B G E H F C A

给出二叉树的先序遍历序列和中序遍历序列可以唯一确定一棵二叉树。

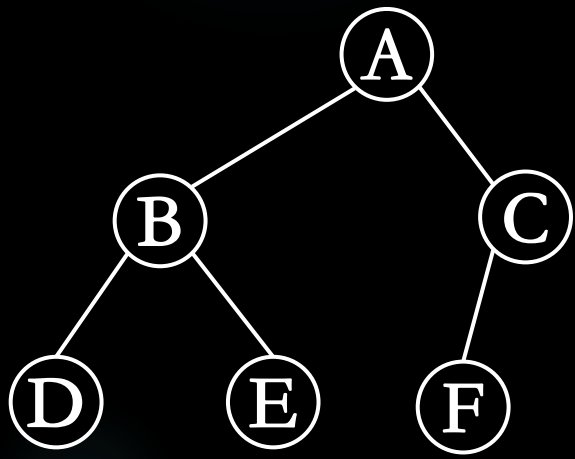
先序遍历序列：A B D C E F

中序遍历序列：B D A E C F



先序遍历序列：A B D E C F

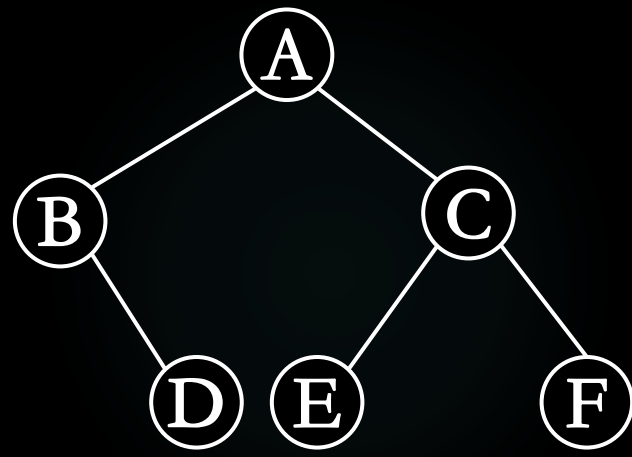
中序遍历序列：D B E A F C



给出二叉树的后序遍历序列和中序遍历序列可以唯一确定一棵二叉树。

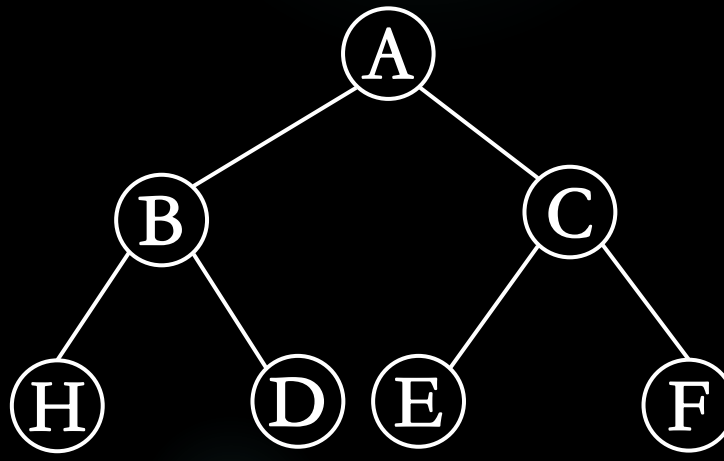
后序遍历序列：D B E F C A

中序遍历序列：B D A E C F



后序遍历序列：H D B E F C A

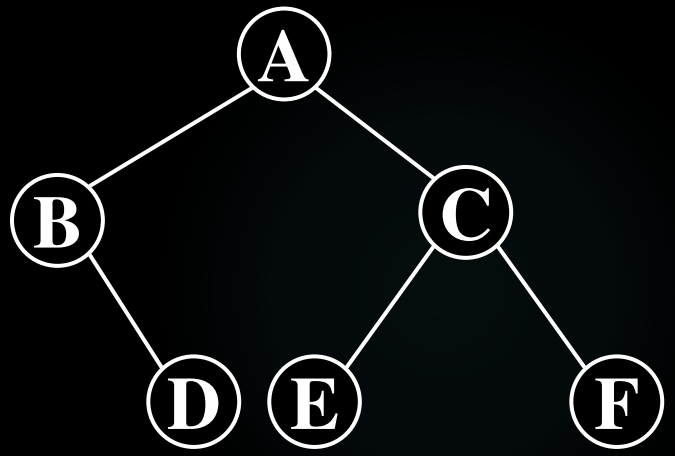
中序遍历序列：H B D A E C F



当n>1时，给出二叉树的先序遍历序列和后序遍历序列不可以唯一确定一棵二叉树。

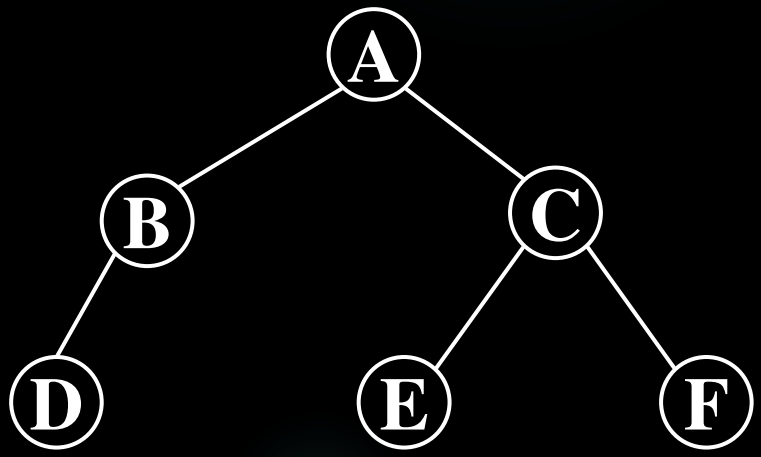
先序遍历序列：A B D C E F

后序遍历序列：D B E F C A



先序遍历序列：A B D C E F

后序遍历序列：D B E F C A

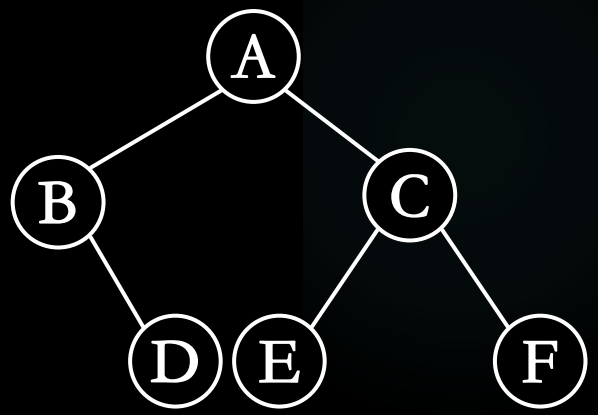


二叉树遍历的应用

1.计算二叉树的结点数

程序 求二叉树的结点数

```
int Size(BTree bt)
{ return Size(bt.Root);}

int Size(BTNode* t)
{ if(!t) return 0;
  return Size(t->LChild) + Size(t->RChild) + 1;
}
```



后序遍历

如何改造成其他遍历方式?

2. 二叉树复制

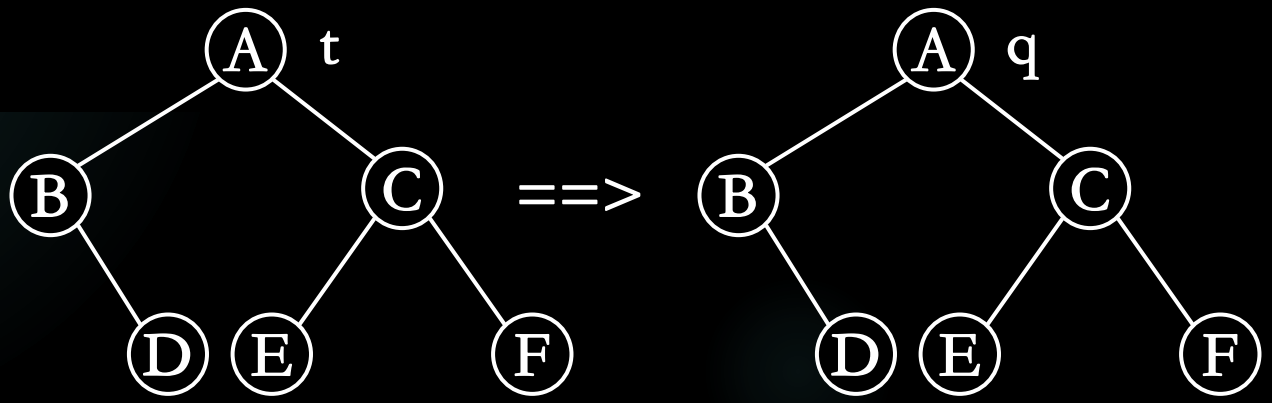
程序 二叉树复制

```
BTNode* Copy(BTNode* p)
{ if(!p) return NULL;
  BTNode* q = NewNode();
  q->Element = p->element;
  q->LChild = Copy(p->LChild);
  q->RChild = Copy(p->RChild);
  return q;}
```

```
BTree Copy (BTree bt)
{
    BTree a;
    a.Root = Copy(bt.Root);
    return a;
}
```

先序遍历

能否改造成
其他遍历方式?



3. 补充：Clear函数

```
void Clear(BTree bt)
{Clear(bt.Root); root=NULL;}

void Clear(BTNode* t)
{  if(t)
   { Clear(t->LChild);
     Clear(t->RChild);
     free(t);
   }
}
```

后序遍历

可以改造成其他遍历方式吗?

二叉树在很多领域都有着广泛的应用——

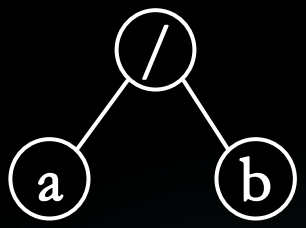
编译原理中的表达式树

专家系统中的决策树

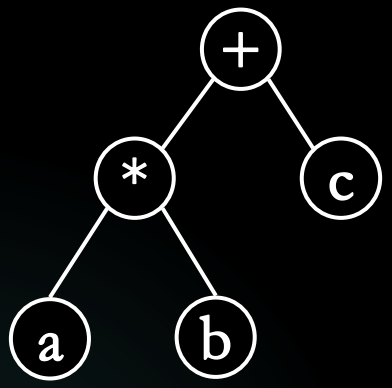
猜谜游戏的决策树

编译原理中的表达式树

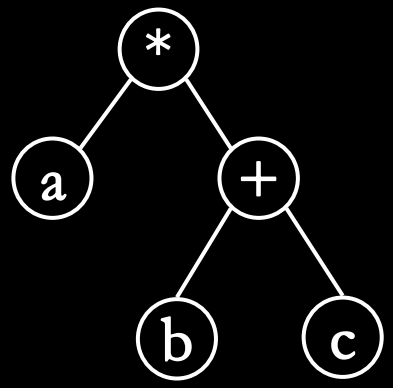
(a) a/b



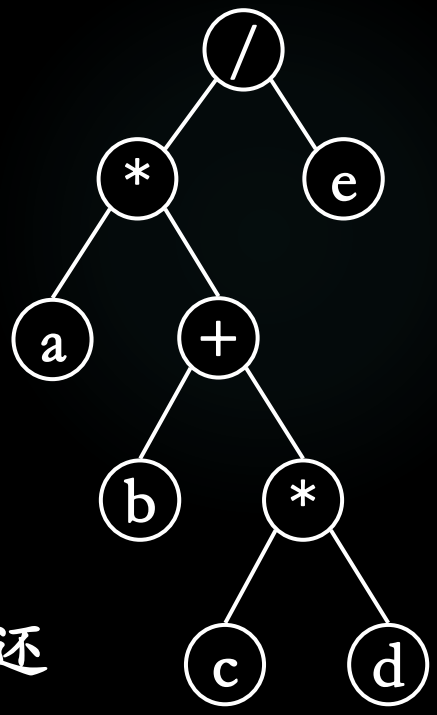
(b) a*b+c



(c) a*(b+c)



(d) a*(b+c*d)/e



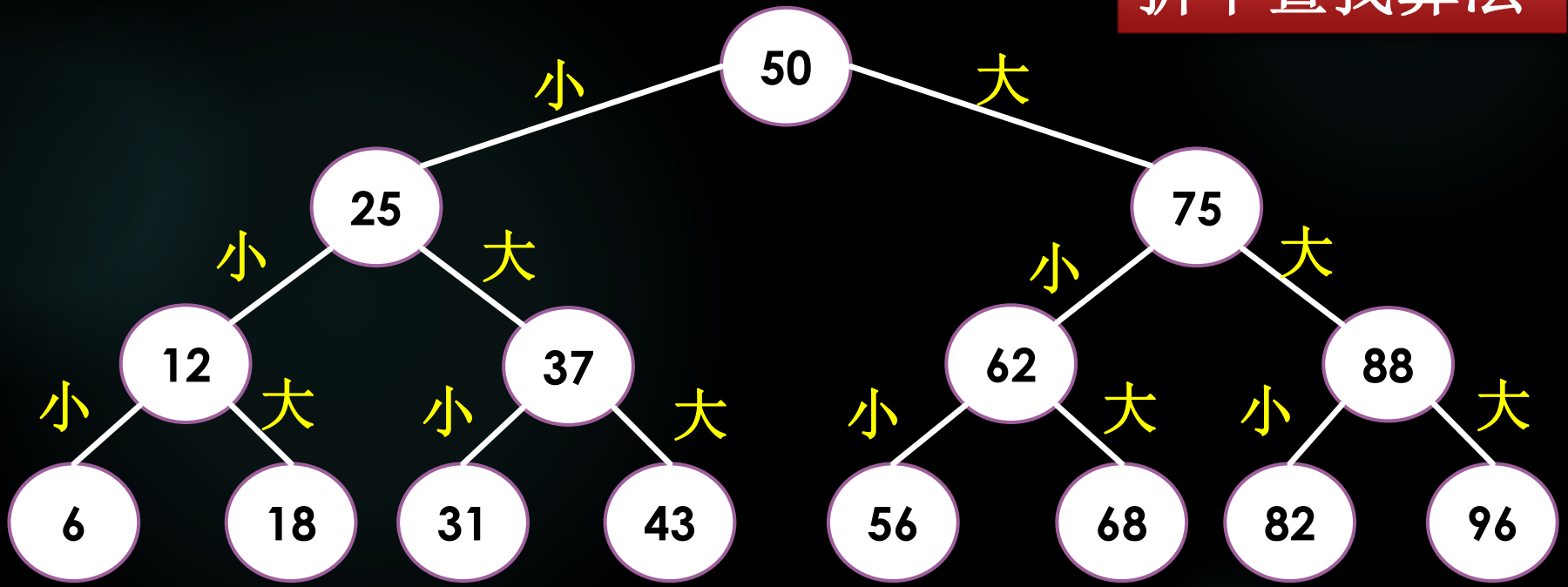
先序遍历→前缀表达式
中序遍历→中缀表达式
后序遍历→后缀表达式

编译原理中还用了语法树

决策树

- 如果我在纸上写一个100以内的数字，给大家来猜，我只回答“大了”或“小了”，怎样才能最快猜中？

折半查找算法



树

目录

- 树的定义
- 二叉树
- 二叉树的遍历
- 树和森林
- 堆和优先级队列
- 哈夫曼编码

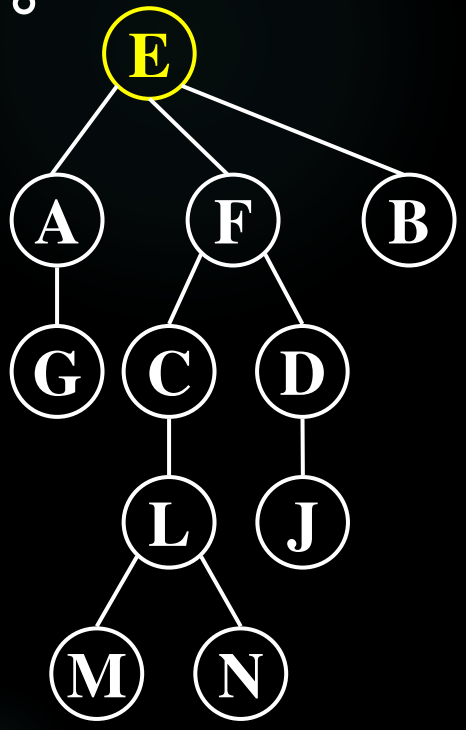
回顾

森林：是树的集合。0个或多个不相交的树组成森林。

果园或有序森林：有序树的有序集合。

若增加一个结点，将森林中各树的根作为新增结点的孩子，则森林即成为树

将树中的根去掉，则得到根的子树组成的森林。



森林与二叉树的转换

为什么要转换?

如果树和森林能够用二叉树表示，则前面对二叉树的讨论成果可应用于一般树和森林。

注意：树和二叉树的主要区别

二叉树有左右孩子之分，而树没有左右孩子之分

森林(Forest)转换成二叉树(BTree)

可以将任何森林唯一地表示成一棵二叉树。

方法如下：

(1)若F为空，则B为空二叉树

(2)若F非空，则

B的根是F中第一棵子树T1的根R_1

我们讨论的是有序森林

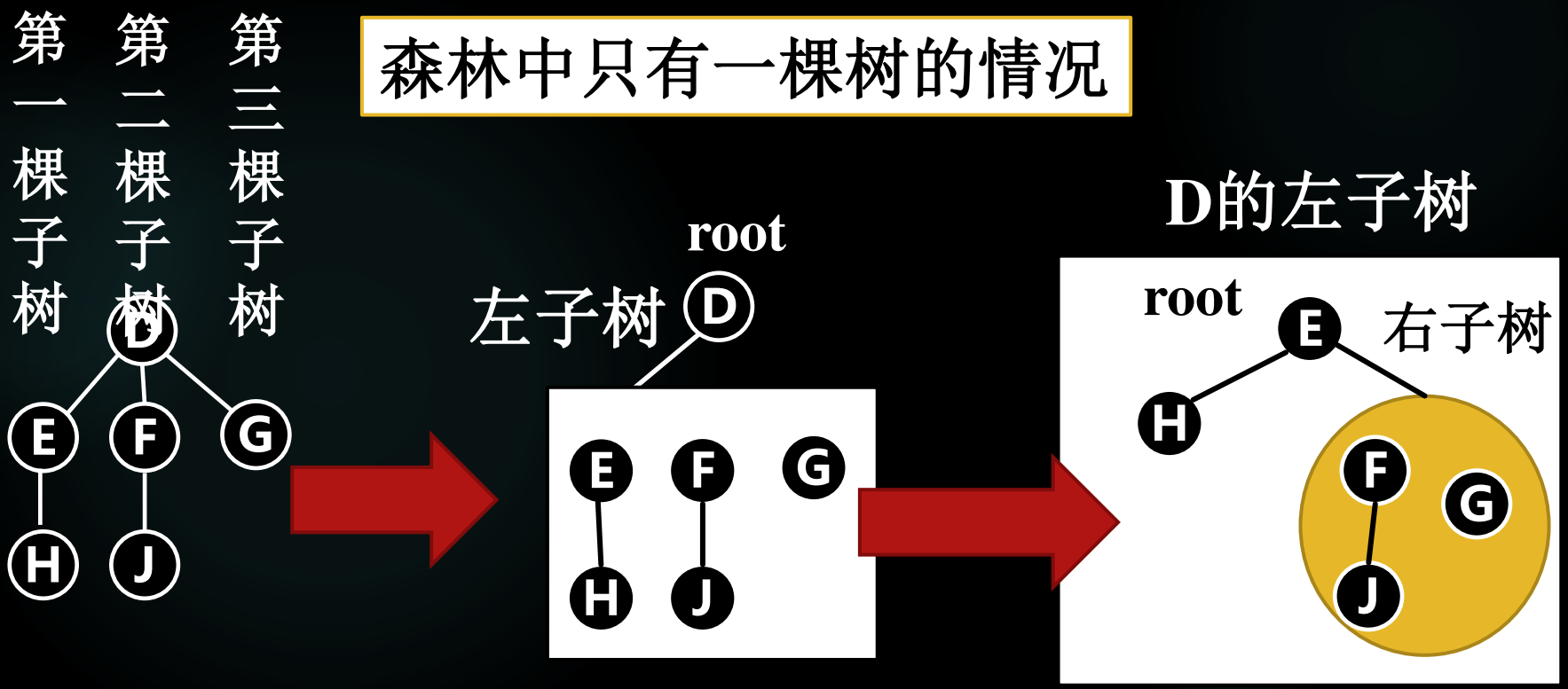
B的左子树是R_1的子树森林（T_{11}，T_{12}，…，T_{1m}）所对应的二叉树，B的右子树是森林（T_2，…，T_n）所对应的二叉树

最后所形成的二叉树就是森林所对应的二叉树。

我们先讨论单棵树如何转换成一棵二叉树

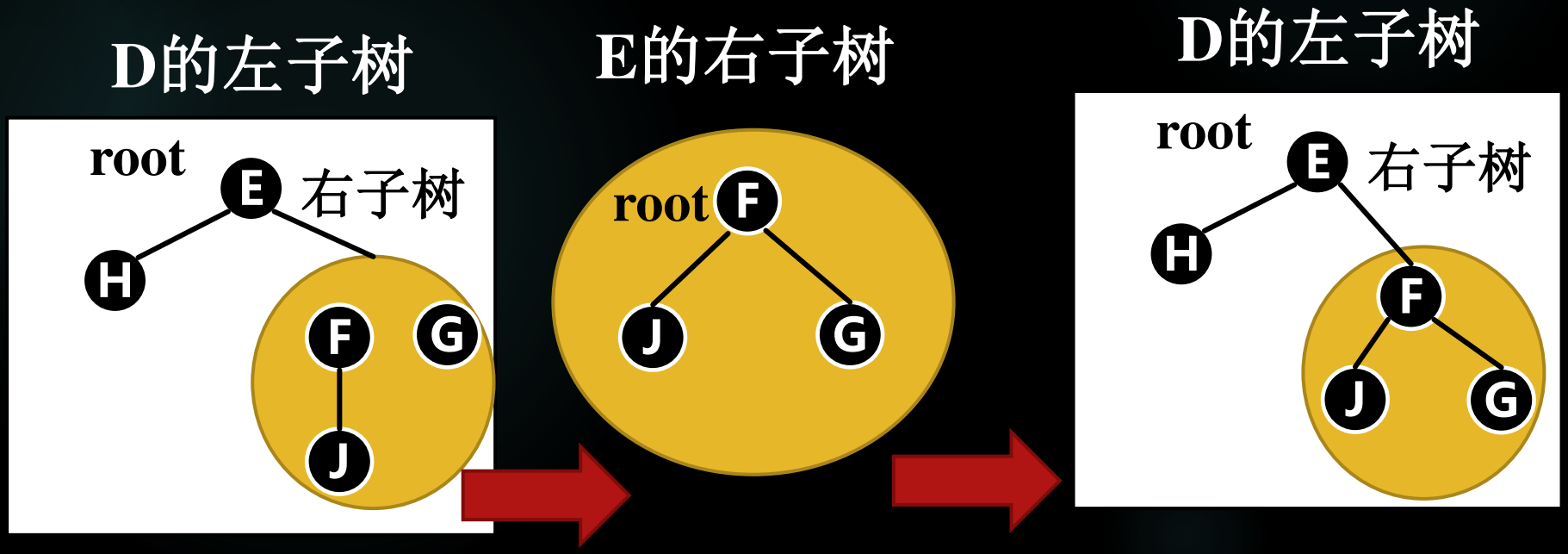
再讨论多棵树如何转换成一棵二叉树

- B的根是F中第一棵子树T1的根R_1
- B的左子树是R_1的子树森林（T_{11}，T_{12}，…，T_{1m}）所对应的二叉树，B的右子树是森林（T_2，…，T_n）所对应的二叉树
- 最后所形成的二叉树就是森林所对应的二叉树。



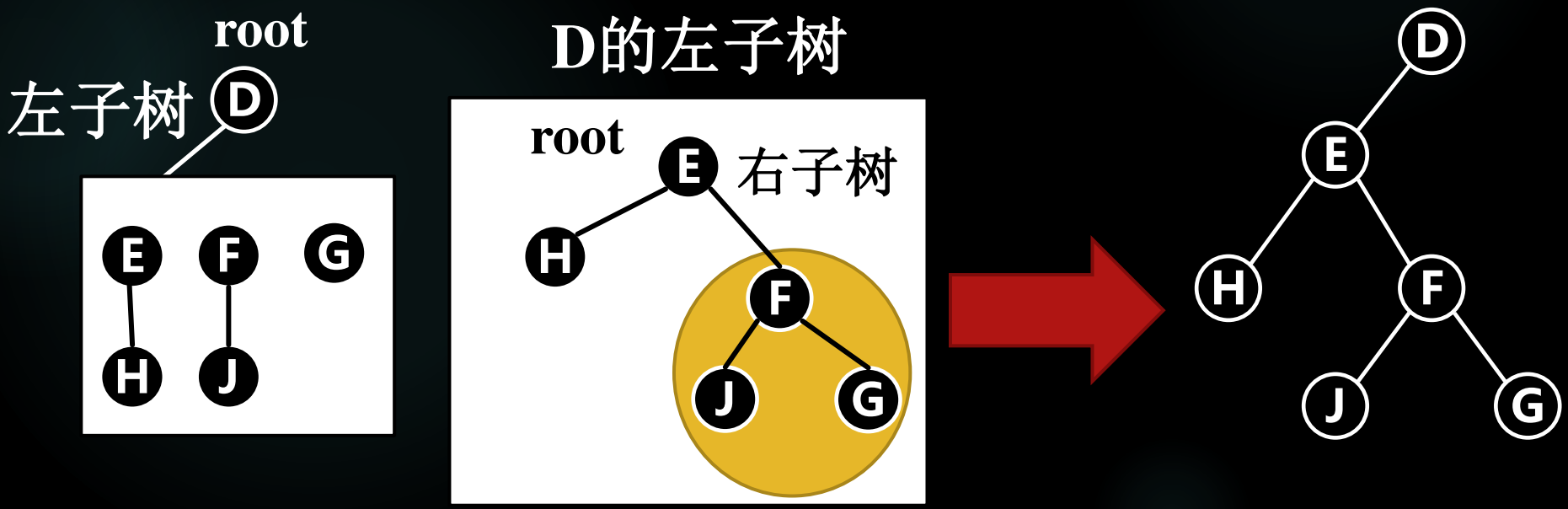
- B的根是F中第一棵子树T1的根R_1
- B的左子树是R_1的子树森林（T_{11}，T_{12}，…，T_{1m}）所对应的二叉树，B的右子树是森林（T_2，…，T_n）所对应的二叉树
- 最后所形成的二叉树就是森林所对应的二叉树。

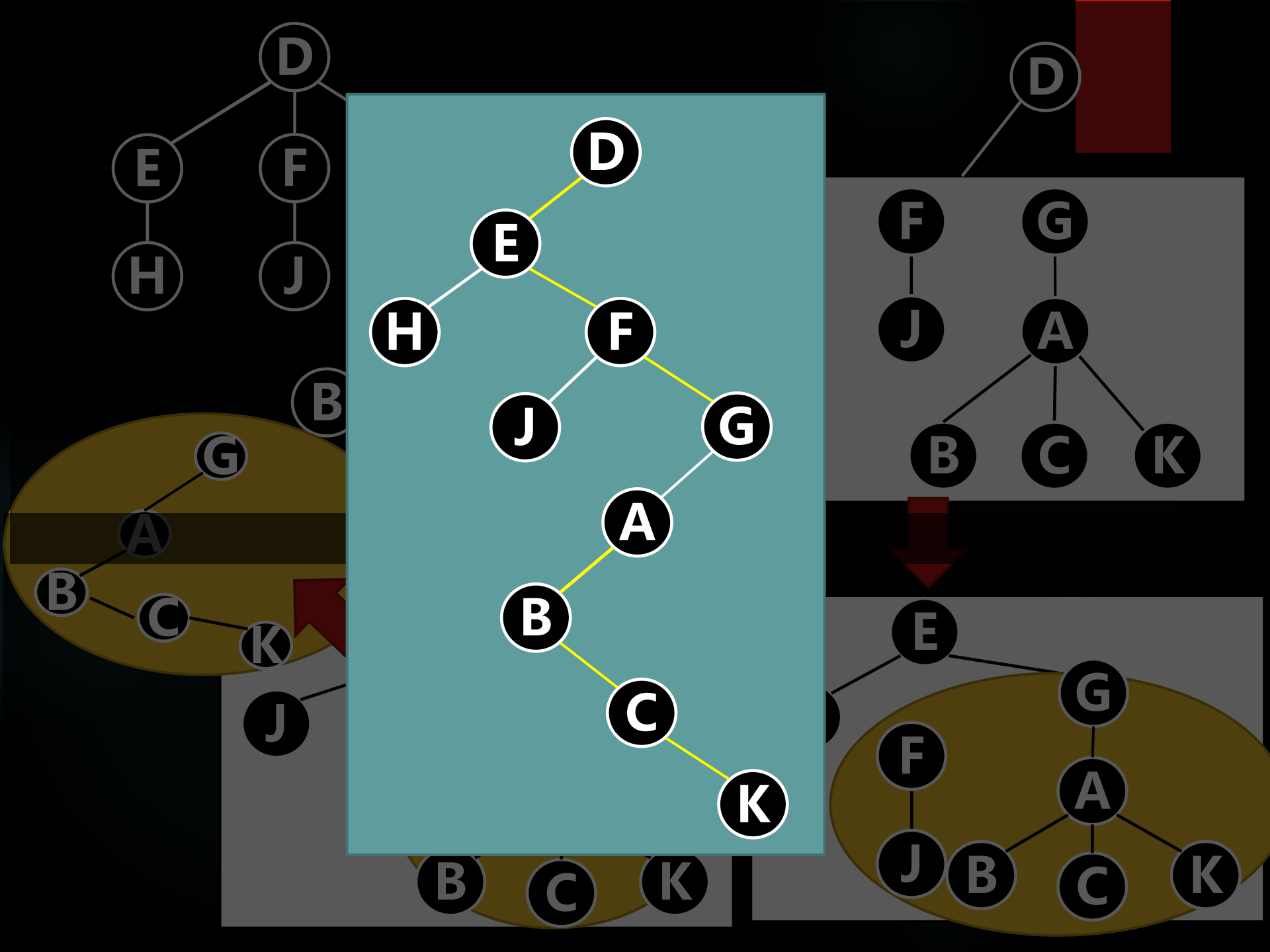
森林中只有一棵树的情况



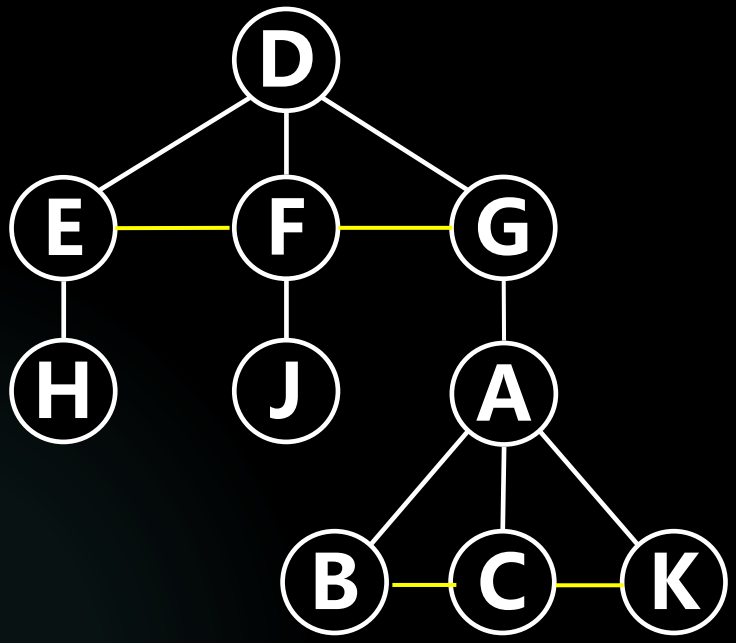
- B的根是F中第一棵子树T1的根R_1
- B的左子树是R_1的子树森林（T_{11}，T_{12}，…，T_{1m}）所对应的二叉树，B的右子树是森林（T_2，…，T_n）所对应的二叉树
- 最后所形成的二叉树就是森林所对应的二叉树。

森林中只有一棵树的情况

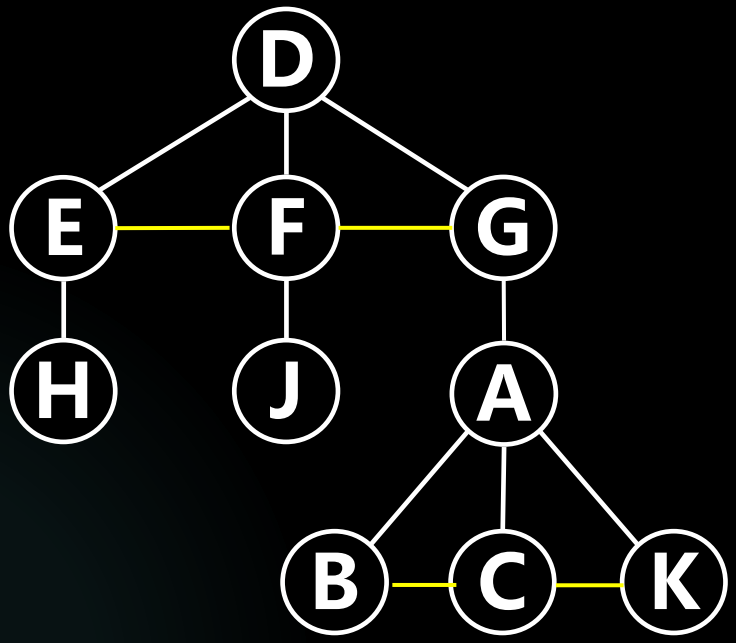



D
E
F
H
J
B
G
A
B
C
K
J
D
E
H
F
J
G
A
B
C
K
D
F
G
J
A
B
C
K
E
G
F
A
J
B
C
K

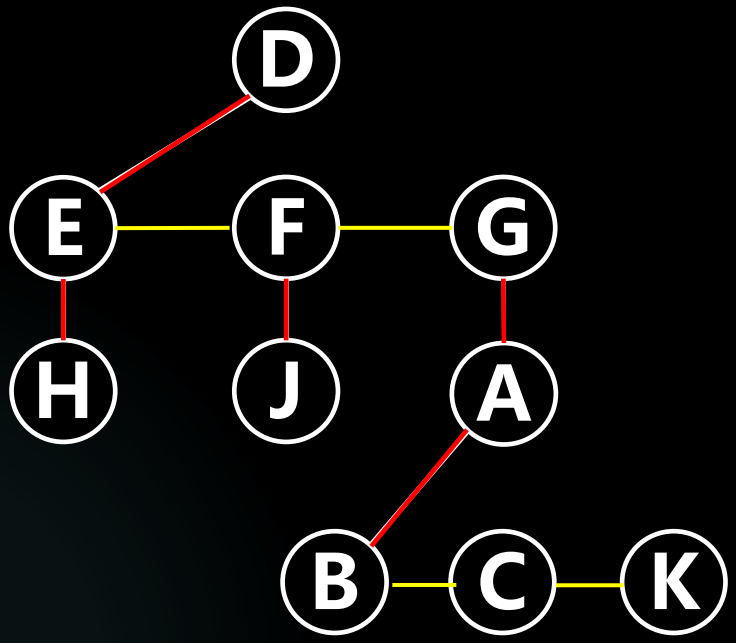
有没有更加直观的方式进行转换?



亲兄弟之间画上黄线



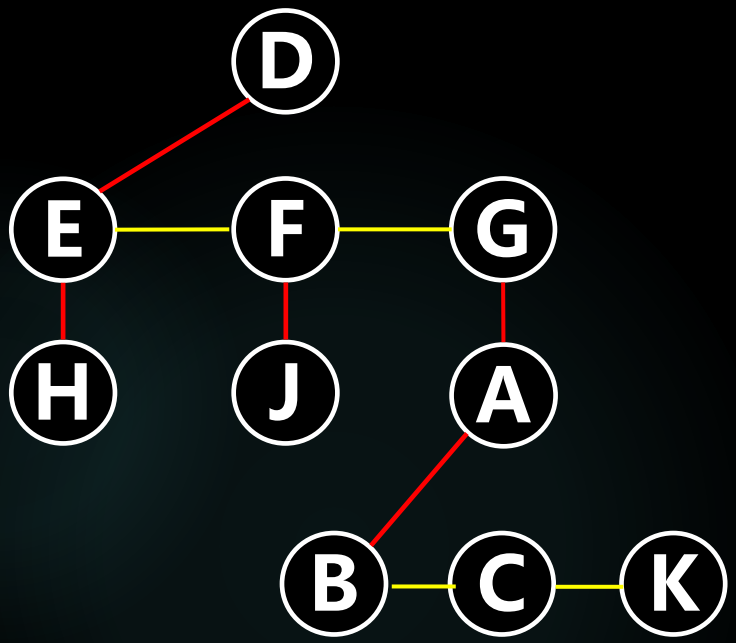
断开第二个孩子、第三个孩子…与双亲的关系
只有老大与双亲保留关系

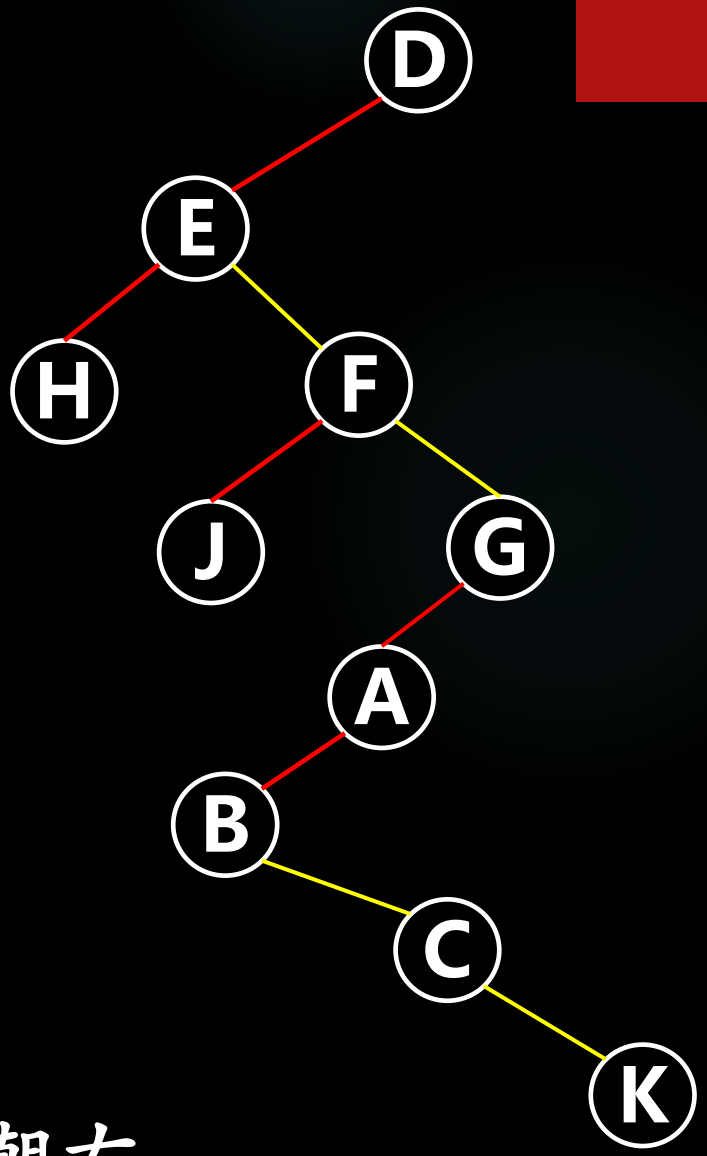


老大成为双亲的左孩子

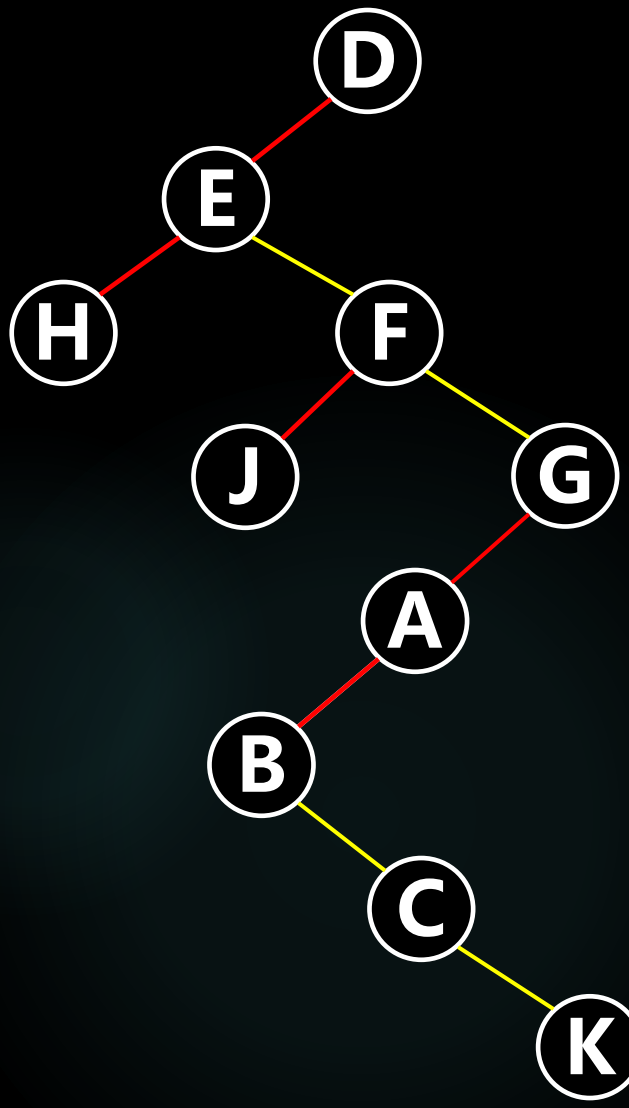
左

右

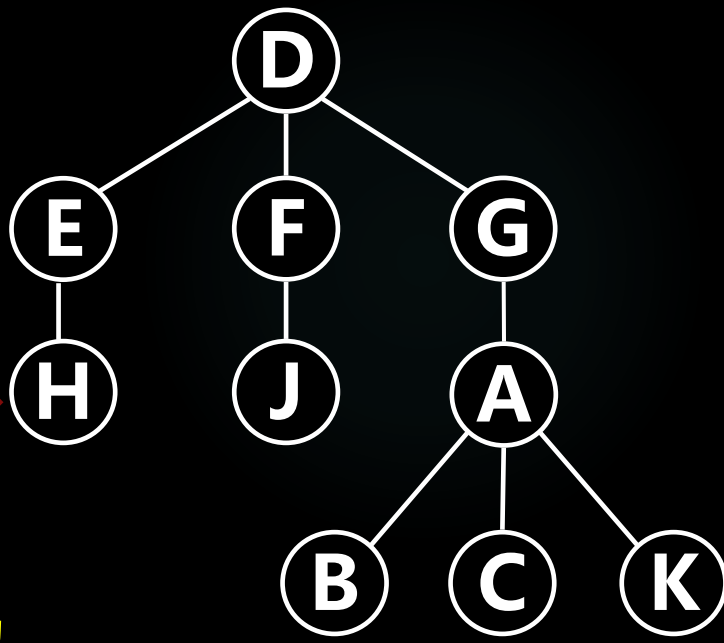




红线朝左，黄线朝右

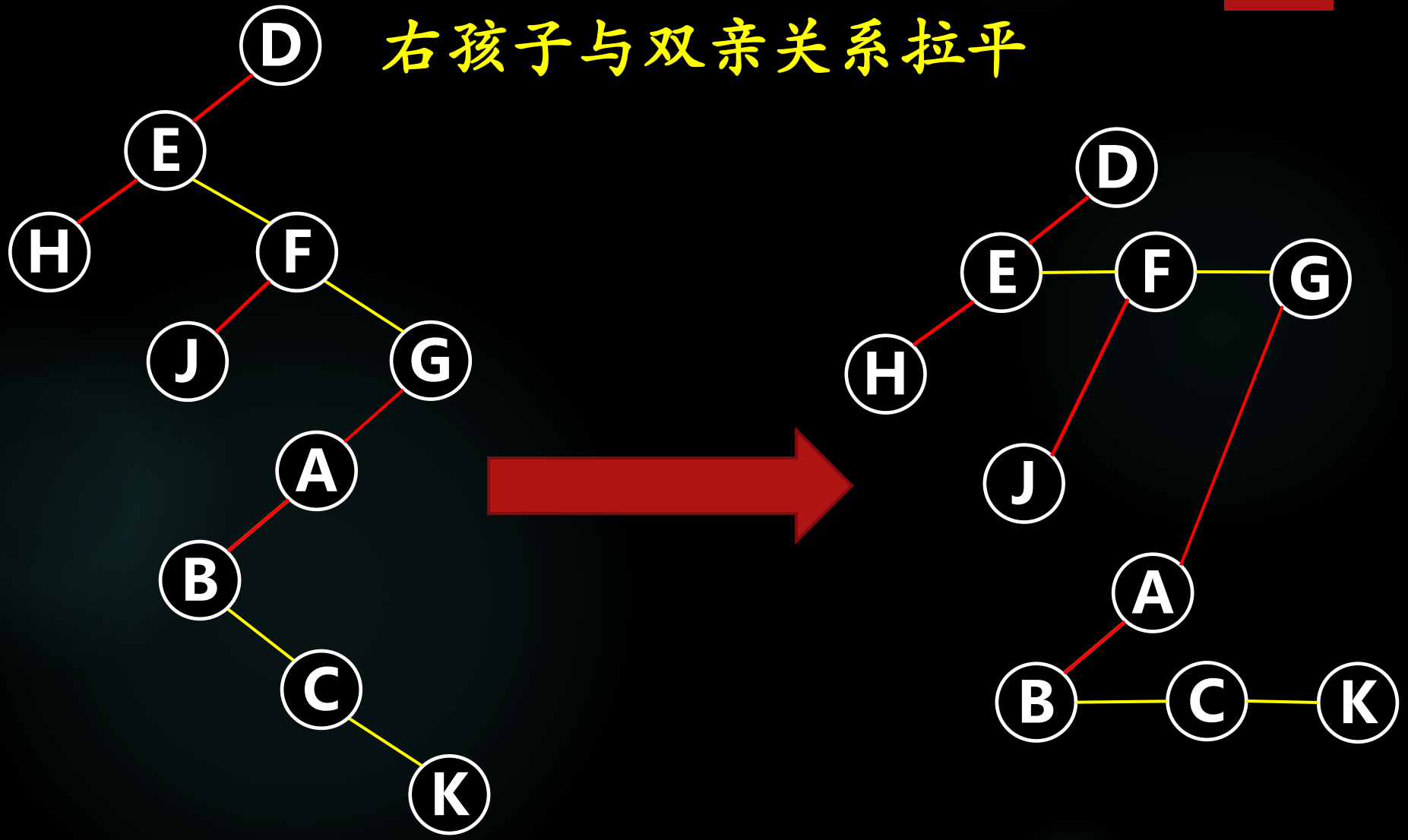


如何还原

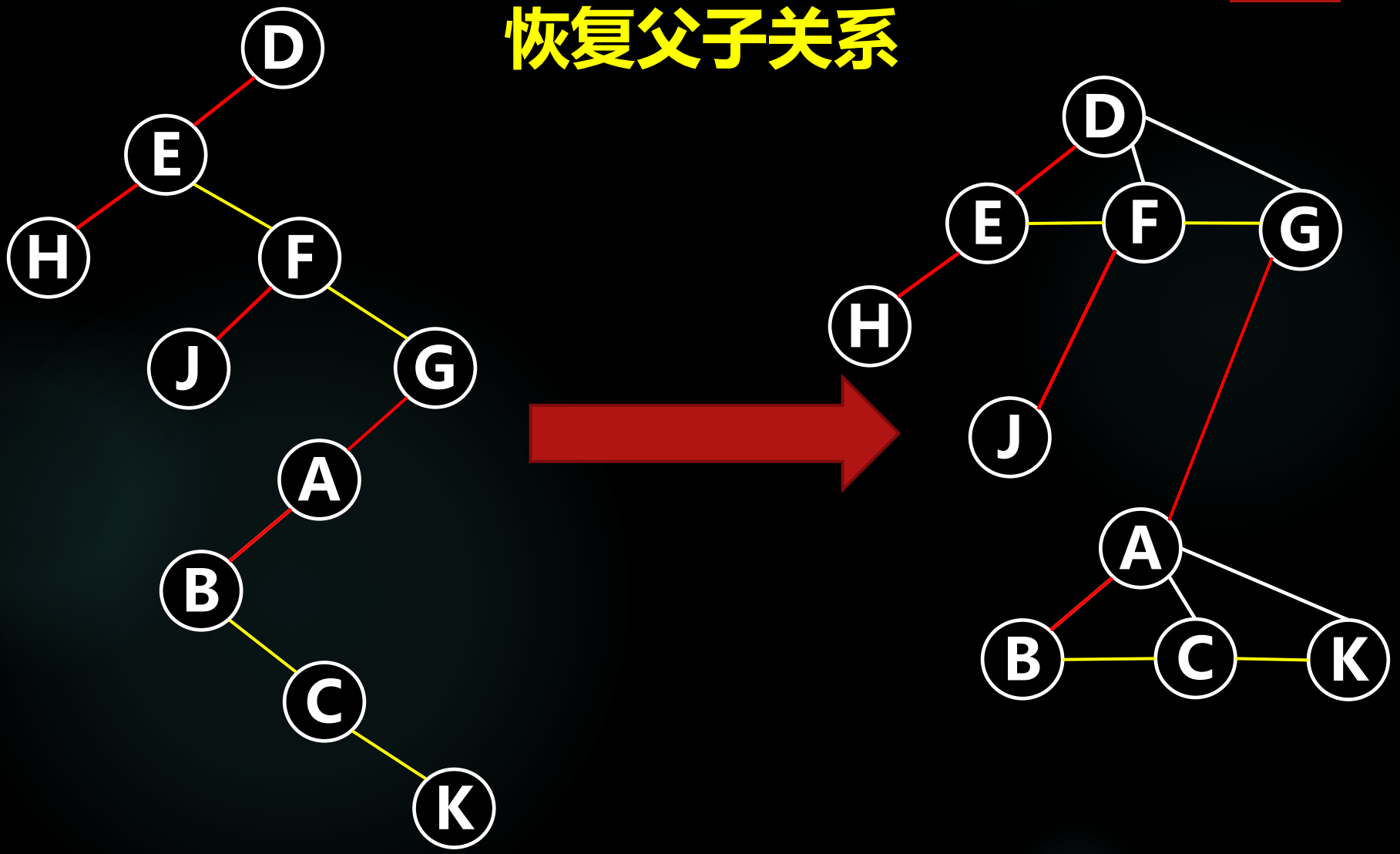


左孩子右兄弟

黄线拉平
右孩子与双亲关系拉平



黄线连接的其实是亲兄弟
恢复父子关系



单棵树转换成二叉树规则总结

(1)左孩子右兄弟

二叉树中有两个结点X和Y，且X是Y的双亲

二叉树中	树中
Y是X左孩子	Y是X孩子
Y是X右孩子	Y是X兄弟

(2) 树的根结点没有兄弟，所以树对应的二叉树根结点没有右子树

如果将单独的一棵树转换成二叉树，则该二叉树的根只有左孩子

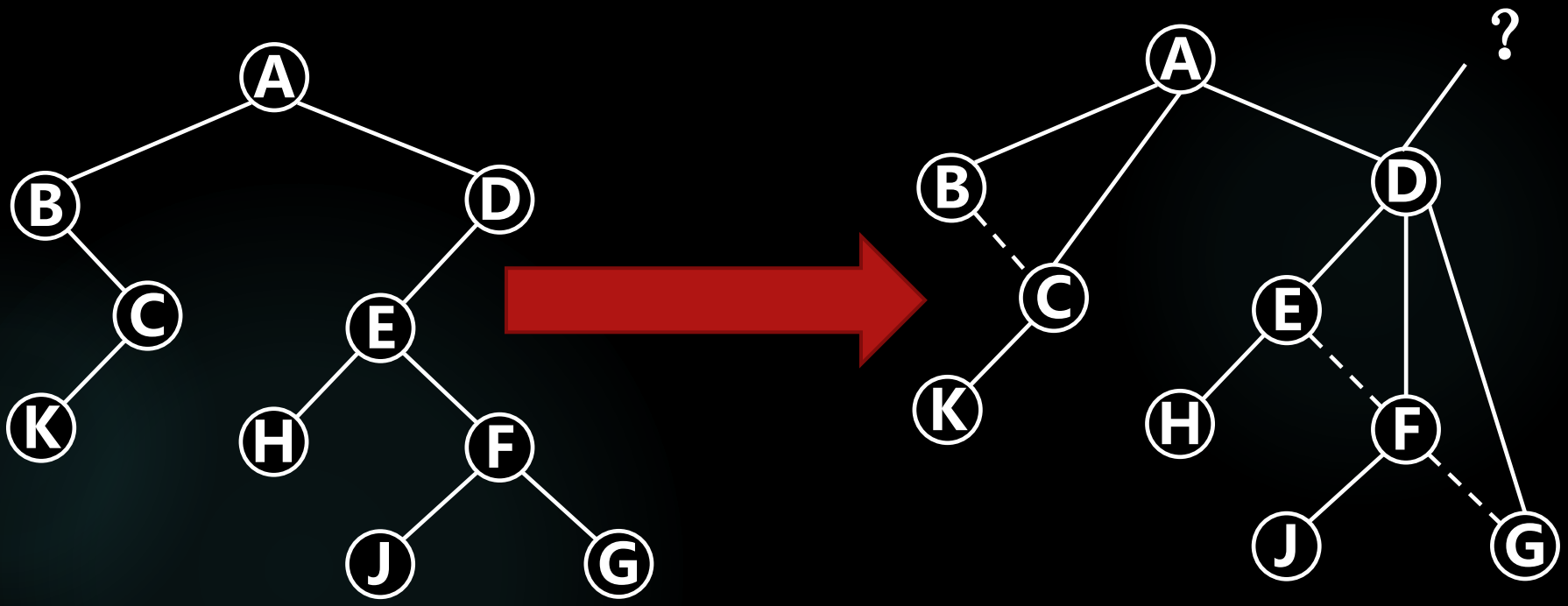
已经学会了单棵树的二叉树转换

下面开始多棵树的二叉树转换

一个简单的开始

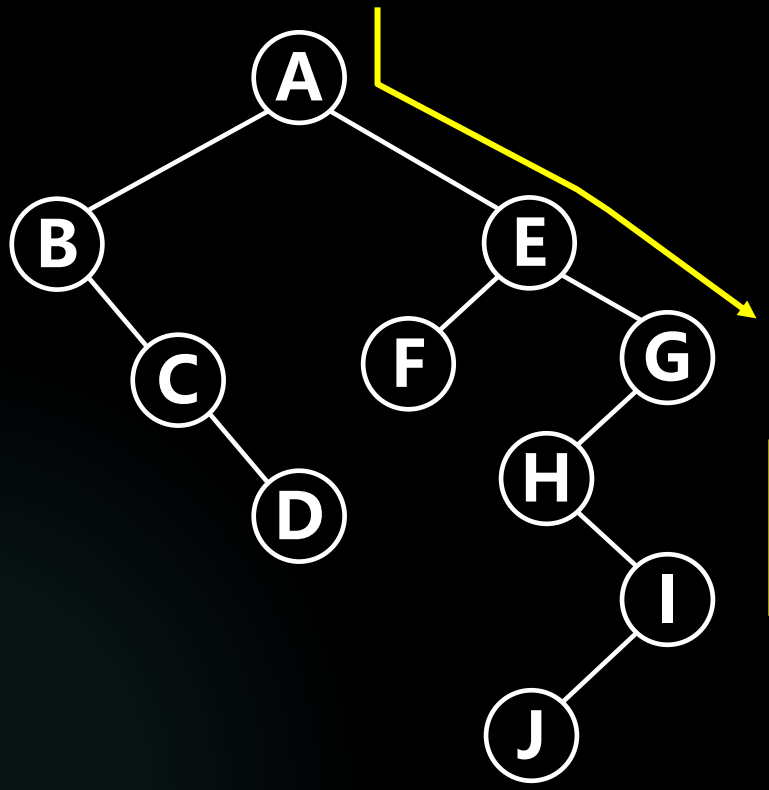
二叉树转换成森林

左孩子仍是孩子，右孩子变为兄弟



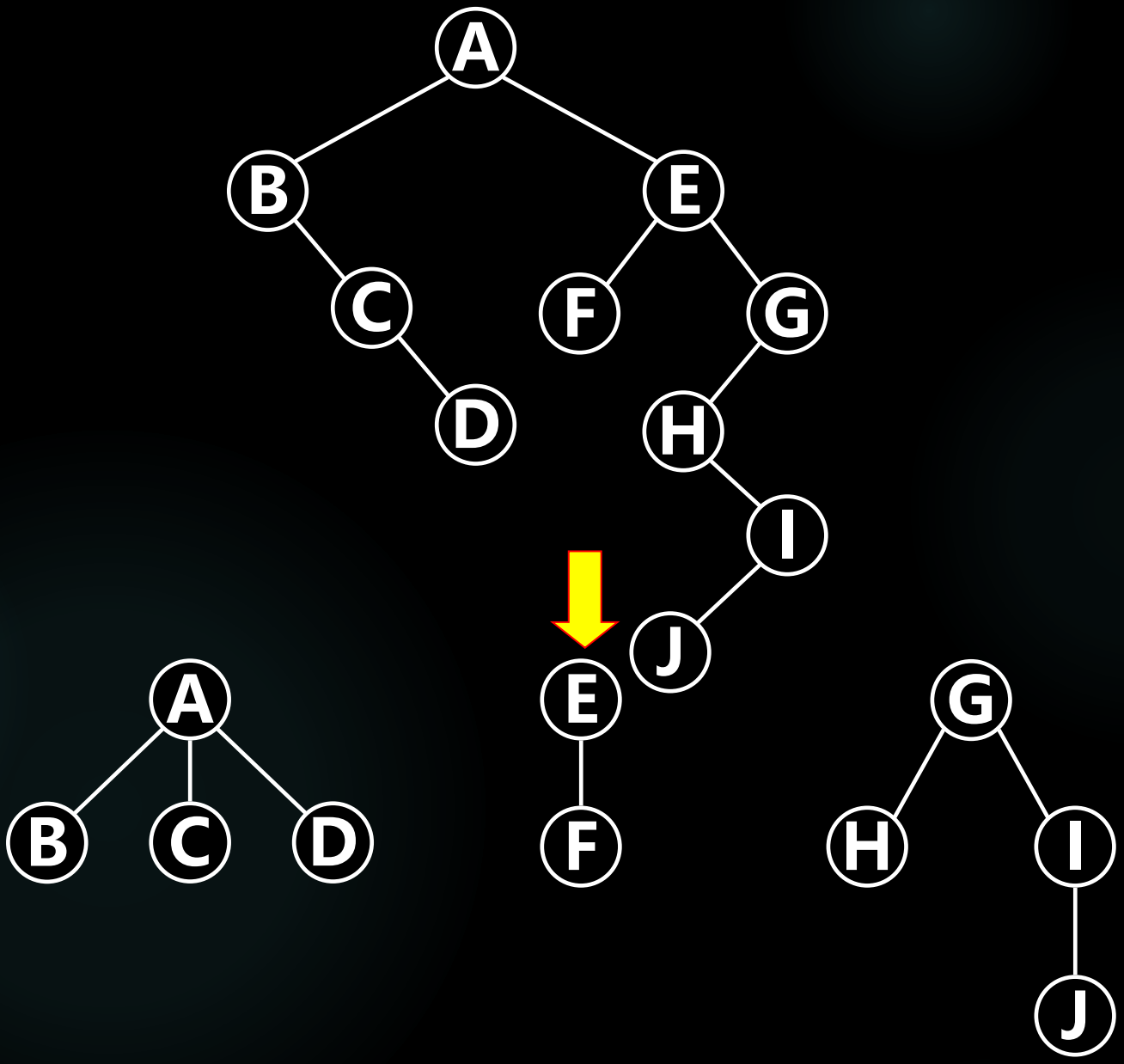
右孩子与双亲变成亲兄弟

一棵二叉树转换成的森林中有多少棵树?



经过3个结点，
故森林中3棵树

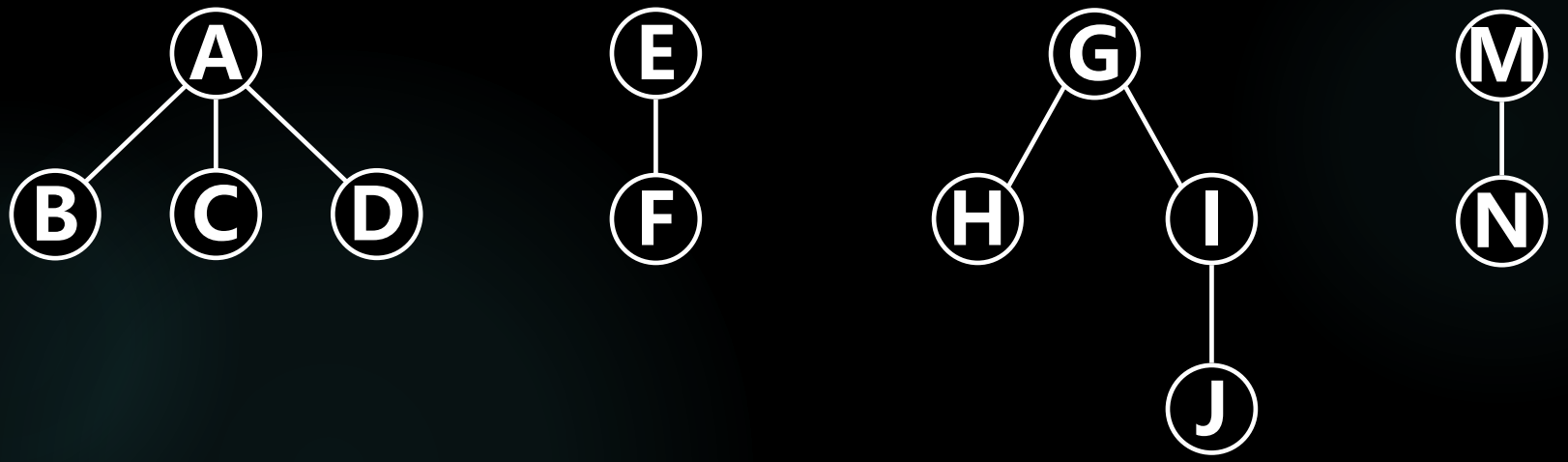
一棵二叉树转化成的森林中所具有的树的数目，等于二叉树从根结点开始沿右链到第一个没有右孩子的结点所经过的结点数目。

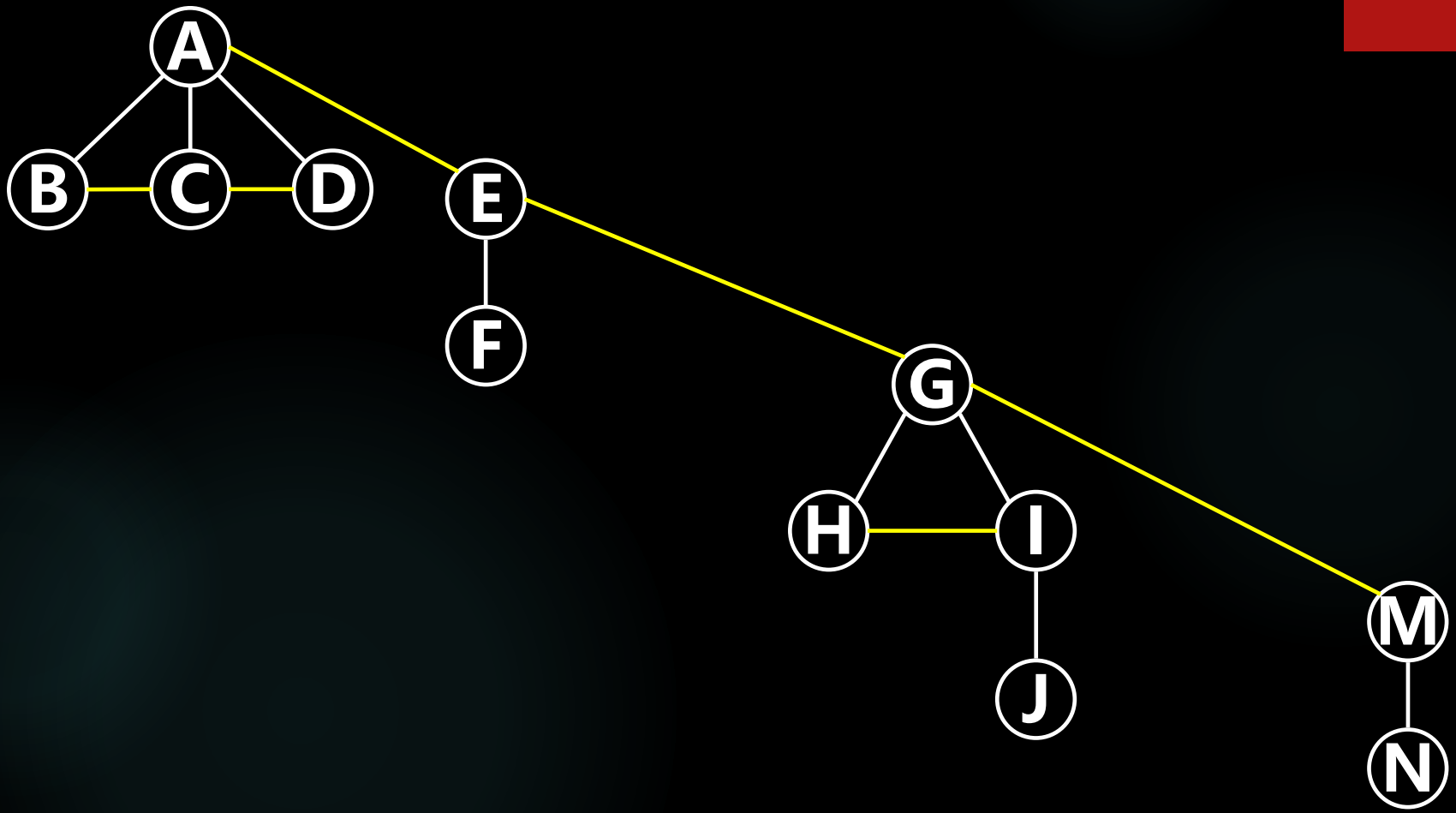

A
B
E
C
F
G
D
H
I
J
A
B
C
D
E
F
G
H
I
J

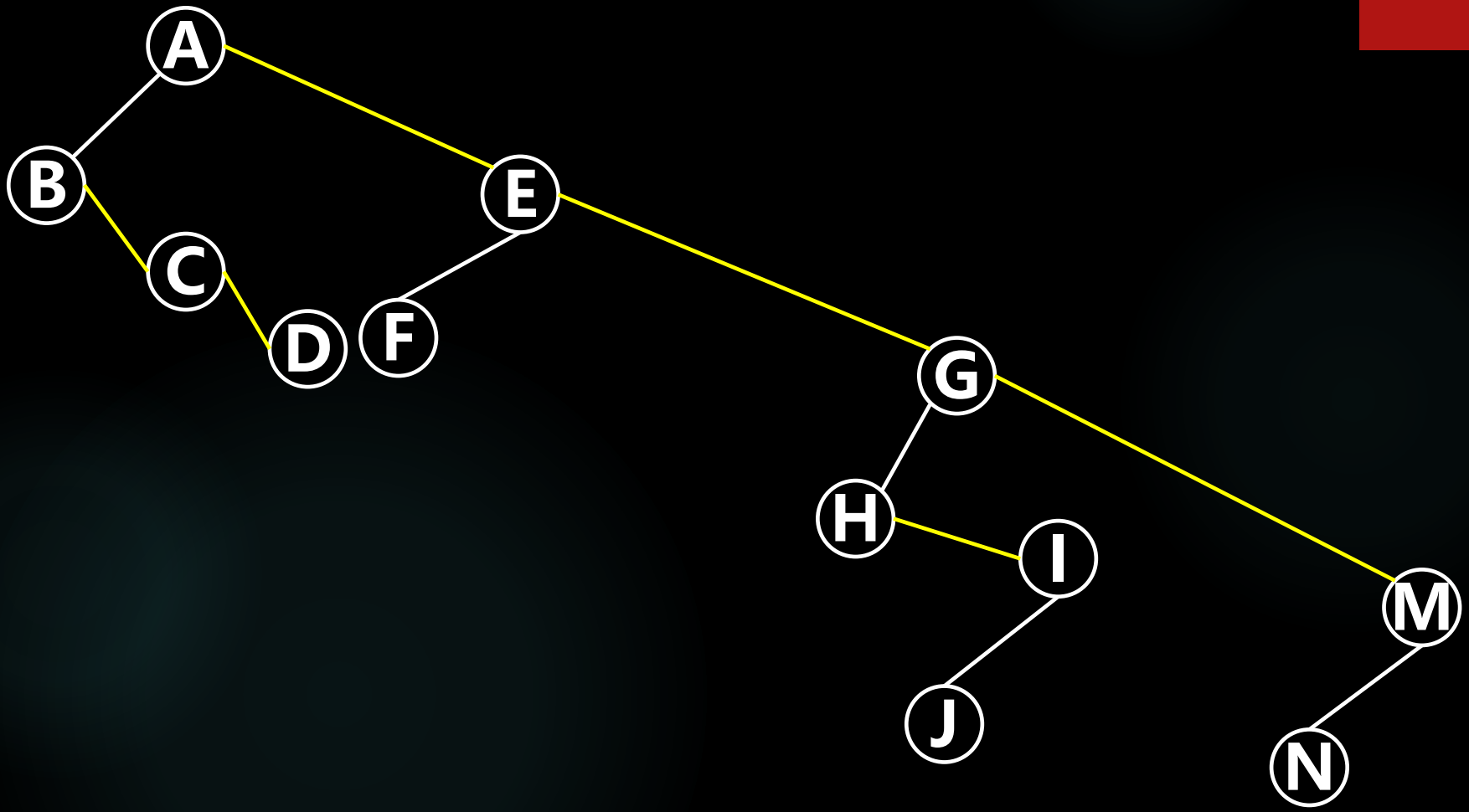
森林如何转换成二叉树

你们能够根据之前的知识自己领悟吗?

- 单棵树转换成二叉树
- 二叉树转换成树/森林
- 森林转换成二叉树?


A
B
C
D
E
F
G
H
I
J
M
N


A
B
C
D
E
F
G
H
I
J
M
N


A
B
E
C
D
F
G
H
I
M
J
N

树与森林的存储表示

树与森林的存储表示

- 链接存储

1. 多重链表表示法

element	$child_1$	$child_2$	……	$child_m$

其中m是树的度。每个结点的指针域个数均为m，故又称为等长的多重链表。

优点：处理简单。

缺点：空指针域多，有浪费。

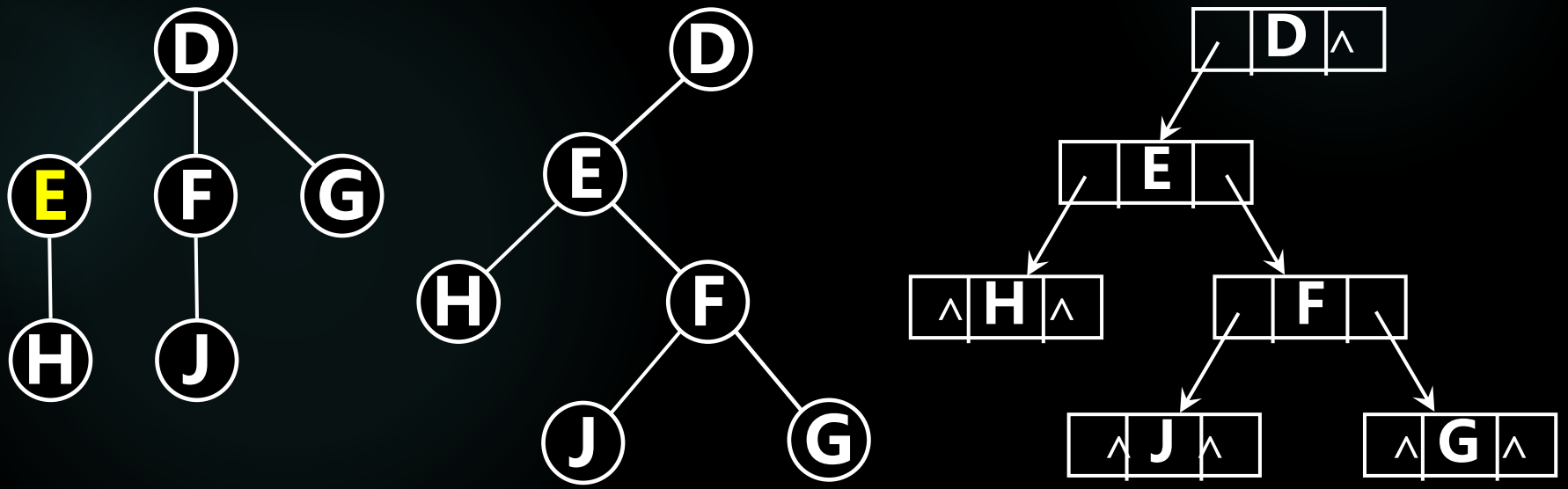
设树中有n个结点，总共有n*m个指针域，其中，只有n-1个非空指针域，故空指针域个数为：

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

2. 孩子兄弟链表表示法

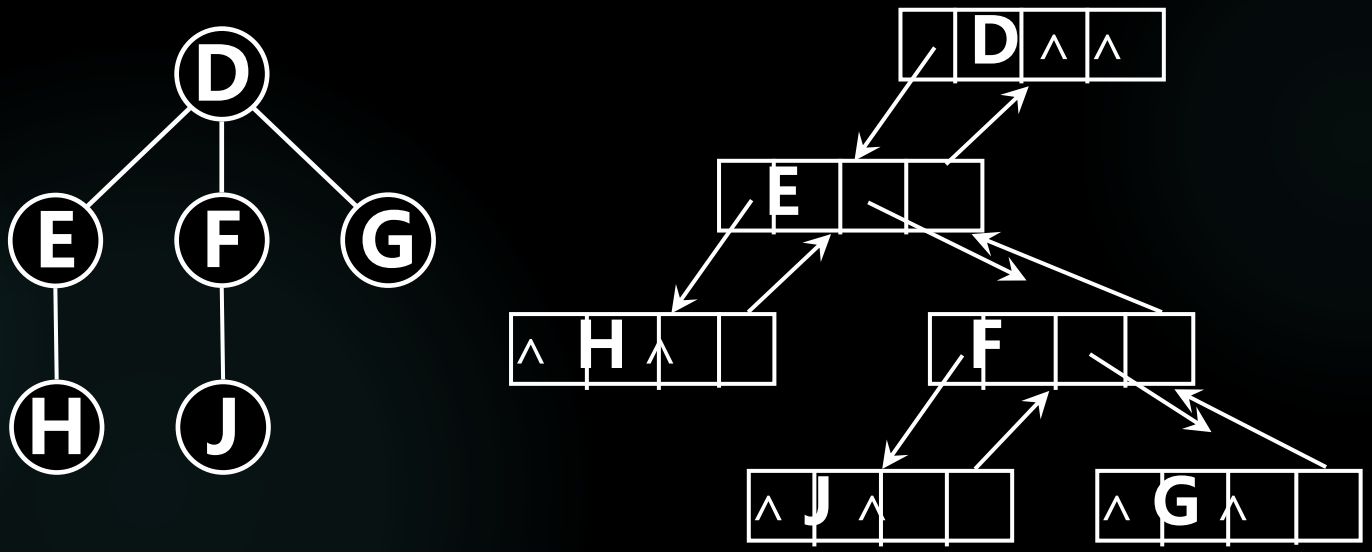
孩子兄弟(左子/右兄弟)表示法实质上就是树所对应的二叉树的二叉链表表示法。其每个结点为：

leftChild	element	rightSibling



4. 三重链表表示法

leftChild	element	rightSibling	parent



优点：可以很方便地得到节点的双亲和孩子信息。

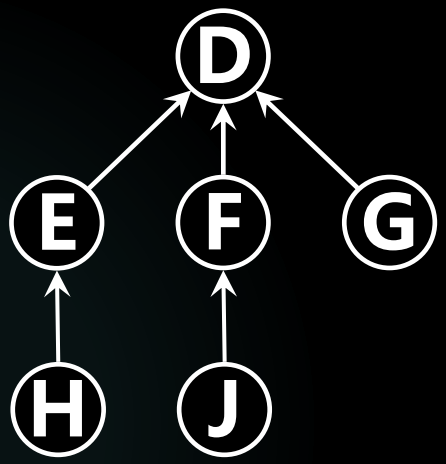
树与森林的存储表示

- 顺序存储

1. 双亲顺序存储表示法

每个结点有两个域：element和parent。parent域为指向该结点的双亲结点的下标。

可以对树中结点按自上而下、自左向右(按层次)的次序顺序存储起来。



双亲表示法

	element	parent
0	D	-1
1	E	0
2	F	0
3	G	0
4	H	1
5	J	2

顺序存储的双亲表示法

思考：如何查找结点的双亲和孩子?

2. 带右链的先序顺序表示法

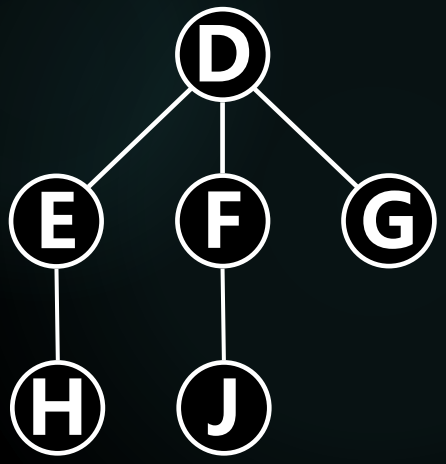
数据元素有三个数据项，element, LTag, sibling。

1.sibling 指向结点的兄弟结点下标

2.LTag为0表示有孩子，孩子存于相邻单元

LTag为1表示无孩子

3.数据元素按对应二叉树的先序遍历的顺序存储结点。

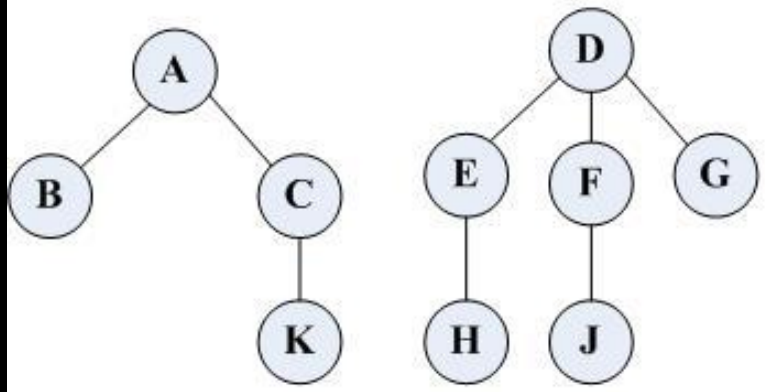


	element	LTag	sibling
0	D	0	-1
1	E	0	3
2	H	1	-1
3	F	0	5
4	J	1	-1
5	G	1	-1

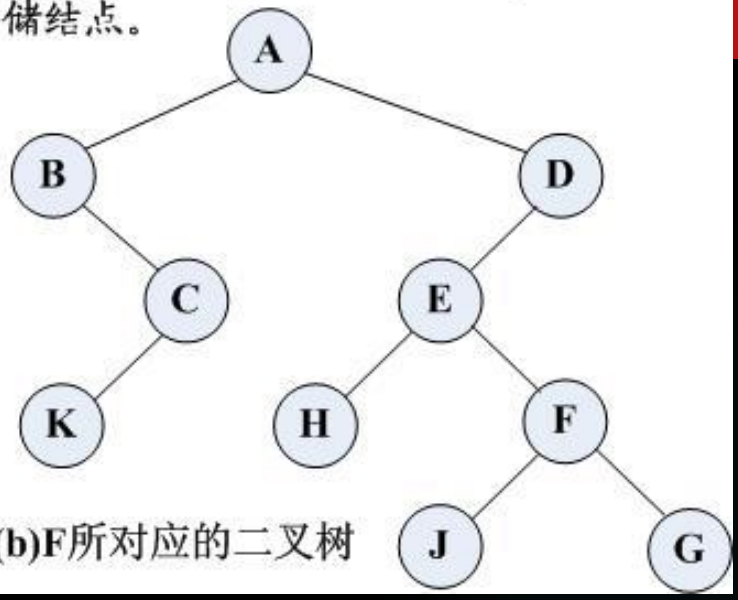
数据元素有三个数据项，element,ltag,sibling。

sibling 指向结点的兄弟　　ltag= 0 有孩子，孩子存于相邻单元　　ltag= 1 无孩子

数据元素按对应二叉树的先序遍历的顺序存储结点。



(a)森林F=(T1,T2)



(b)F所对应的二叉树

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

已经学习过二叉树的遍历

树和森林如何遍历？

1．按深度方向遍历

对森林的深度遍历与二叉树类似，根据树的递归定义，可以有两种遍历次序：先序遍历和中序遍历。

森林(F)	对应	二叉树(B)
先序遍历	等价于	先序遍历
中序遍历	等价于	中序遍历

先序遍历（第一棵树、第二棵树、…）

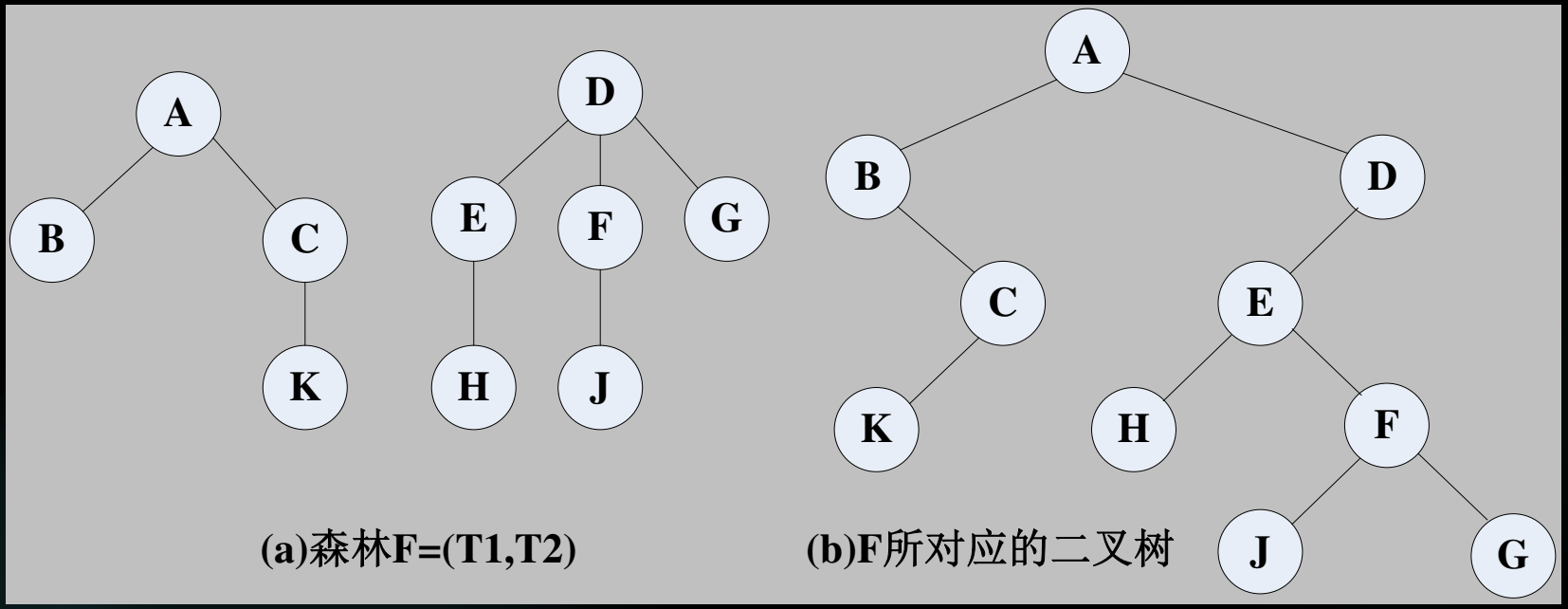
IF森林为空，则遍历结束

ELSE

a) 访问第一棵树的根；

b) 先序遍历（第一棵树的子树森林）；

c) 先序遍历（第二棵树、第三棵树、…）。



对上图（a)的森林的先序遍历的结果是:

A B C K D E H F J G

它等同于对(b)的二叉树的先序遍历。

对森林的先序遍历等于对每棵树先序遍历的简单拼接

中序遍历（第一棵树、第二棵树、…）

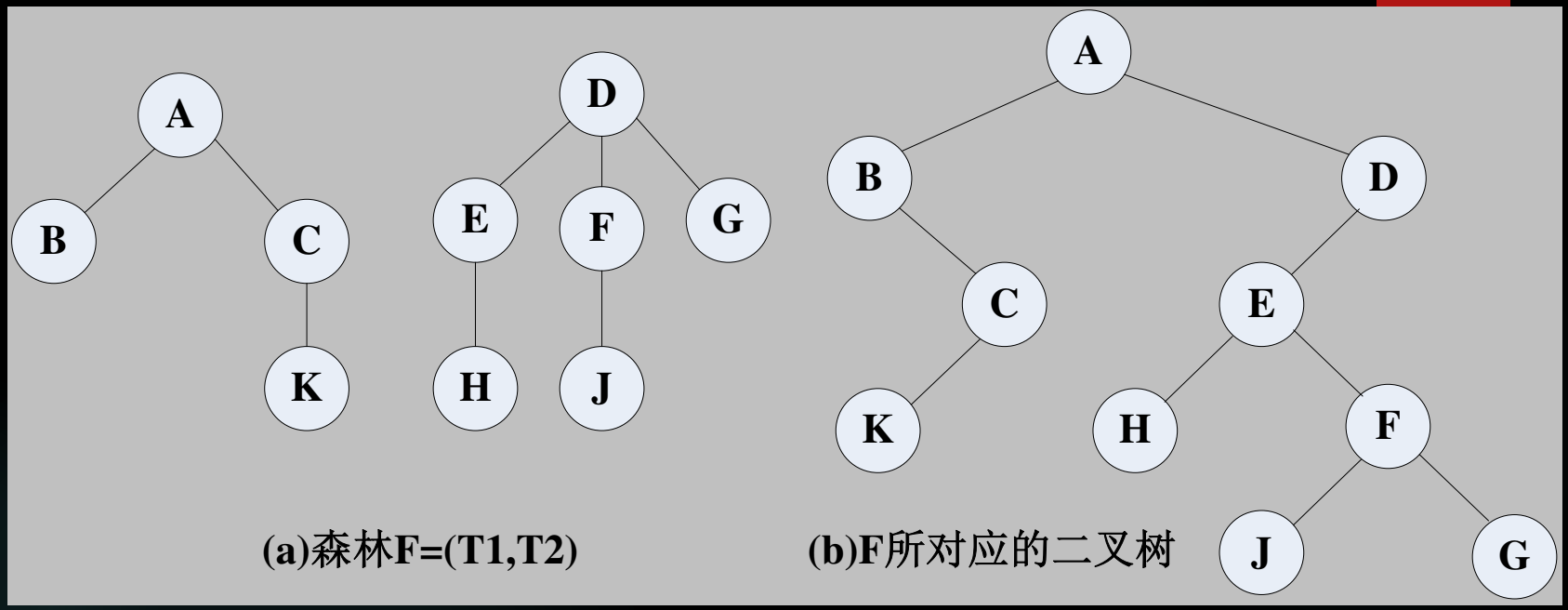
IF森林为空，则遍历结束

ELSE

a)中序遍历（第一棵树的子树森林）；

b)访问第一棵树的根;

c) 中序遍历（第二棵树、第三棵树、…）。



对上图（a)的森林的中序遍历的结果是:

B K C A H E J F G D

它等同于对(b)的二叉树的中序遍历。

对森林的中序遍历等于对每棵树中序遍历的简单拼接

为什么不讨论树和森林的后序遍历?

后序遍历（第一棵树、第二棵树、…）

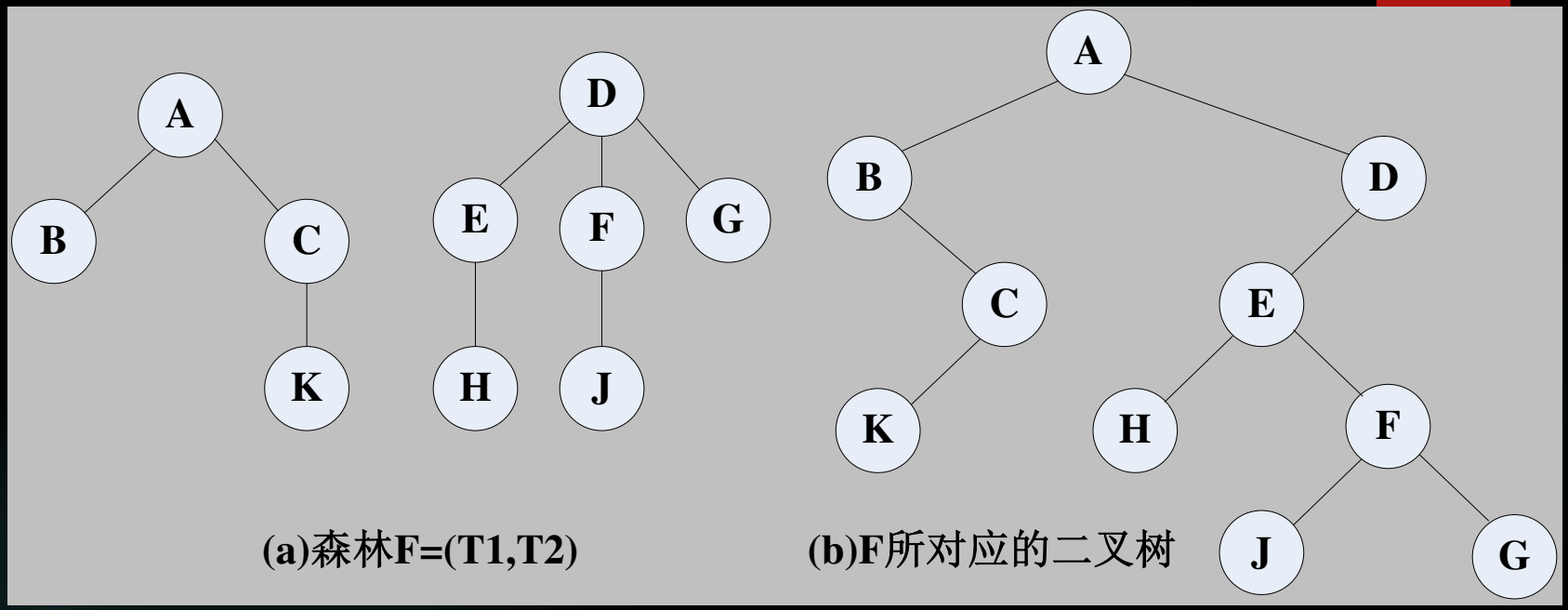
IF森林为空，则遍历结束

ELSE

a)后序遍历（第一棵树的子树森林）；

b)后序遍历（第二棵树、第三棵树、…）；

c)访问第一棵树的根。



对上图（a)的森林的后序遍历的结果是:

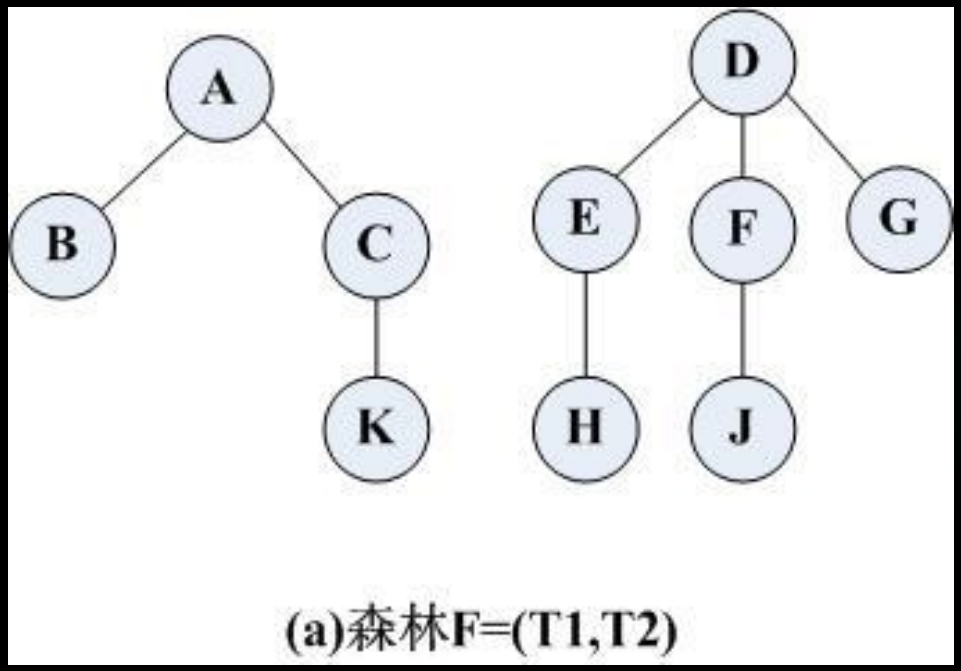
K C B H J G F E D A

它等同于对(b)的二叉树的后序遍历。

对森林的后序遍历不等于对每棵树后序遍历的简单拼接

2. 按宽度方向遍历

首先访问处于第一层的结点，然后访问处于第二层的结点，再访问第三层，…，等，最后访问最下层的结点。



(a)森林F=(T1,T2)

对上图的森林按宽度方向的遍历结果是:

A D B C E F G K H J

树

目录

- 树的定义
- 二叉树
- 二叉树的遍历
- 树和森林
- 堆和优先级队列
- 哈夫曼编码

回顾

堆栈：先进后出的线性结构

队列：先进先出的线性结构

数据元素加入的次序是重要的

每个数据元素都被赋予优先级，数据元素的到来顺序是不重要的，但是离开队伍的顺序以优先级决定

场景

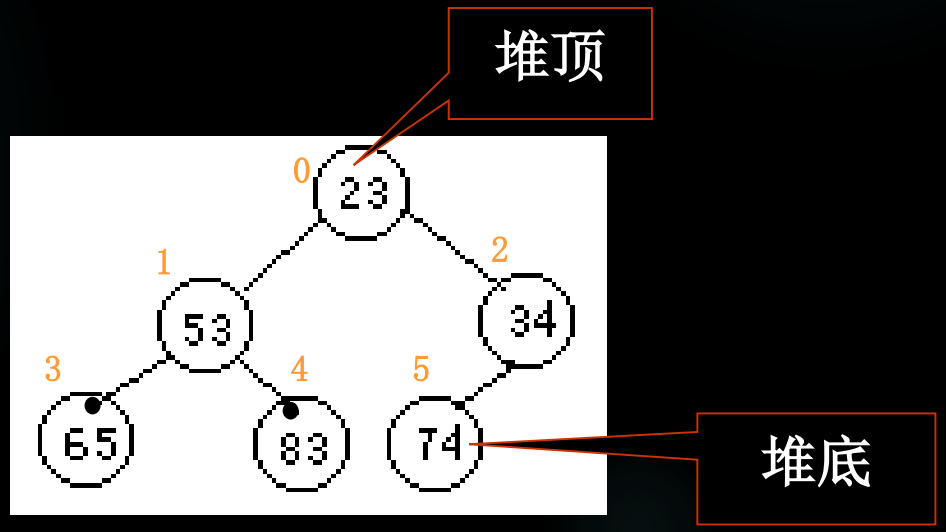
后到达的银行VIP客户可提前办理业务

邮件的处理顺序

到达顺序	0	1	2	3	4	5	6	7	8	9
关键字	**18**	**15**	**1**	**3**	**10**	**22**	**12**	**16**	**20**	**7**
数据元素	A	B	C	D	E	F	G	H	I	J

堆

- 一个大小为n的**堆**是一棵包含n个结点的**完全二叉树**，该树中每个结点的关键字值**大于等于**其**双亲**结点的关键字值
- 完全二叉树的根称为**堆顶**，它的关键字值是整棵树上最小的-**最小堆**
- **最大堆**



回顾

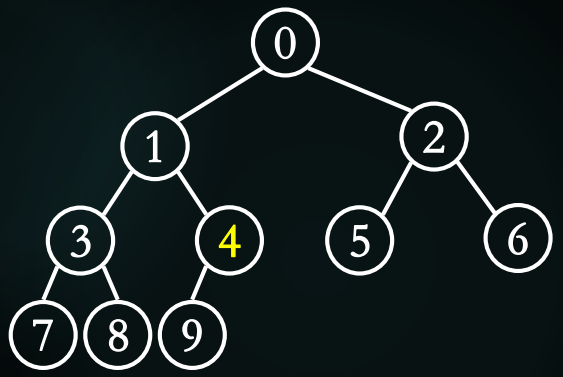
性质 假定对一棵有n个结点的完全二叉树中的结点，按从上到下、从左到右的顺序，从0到n-1编号，设结点序号为i，则有以下关系成立：

（1）当i=0时，该结点为二叉树的根。

（2）若i>0，则该结点的双亲的序号为$\lfloor (i-1)/2 \rfloor$

（3）若2i+1＜n，则该结点左孩子的序号为2i+1，否则该结点无左孩子

（4）若2i+2＜n，则该结点右孩子的序号为2i+2，否则该结点无右孩子



设结点*i*的层号是*k*，*k*层结点编号范围

$$2^{k-1}-1 \le i \le 2^k-2$$

设结点*i*的孩子编号取决于

- 与*i*同层，在i之后结点个数$af(i)=2^k-2-\mathrm{i}$
- 与*i*同层，在i之前结点的孩子个数

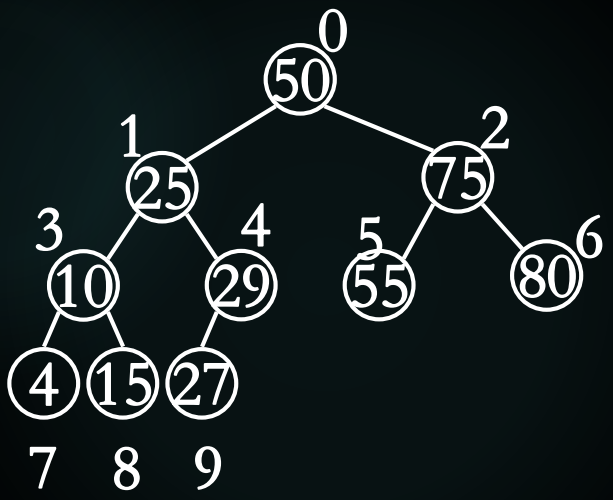
$$bc(i)=2\times(\mathrm{i}-2^{k-1}+1)$$

*i*的左孩子编号=*i*+*af*(*i*)+*bc*(*i*)+1=2*i*+1

完全二叉树的顺序表示

完全二叉树中的结点可以按层次顺序存储在一片连续的存储单元中。根结点保存在编号为0的位置上。

0	1	2	3	4	5	6	7	8	9
50	25	75	10	29	55	80	4	15	27



注意：一般的二叉树不适合用这种存储结构

WHY?

优先级队列 —逻辑结构— 堆数据结构

堆数据结构 —继承— 完全二叉树

完全二叉树 —物理存储— 顺序存储

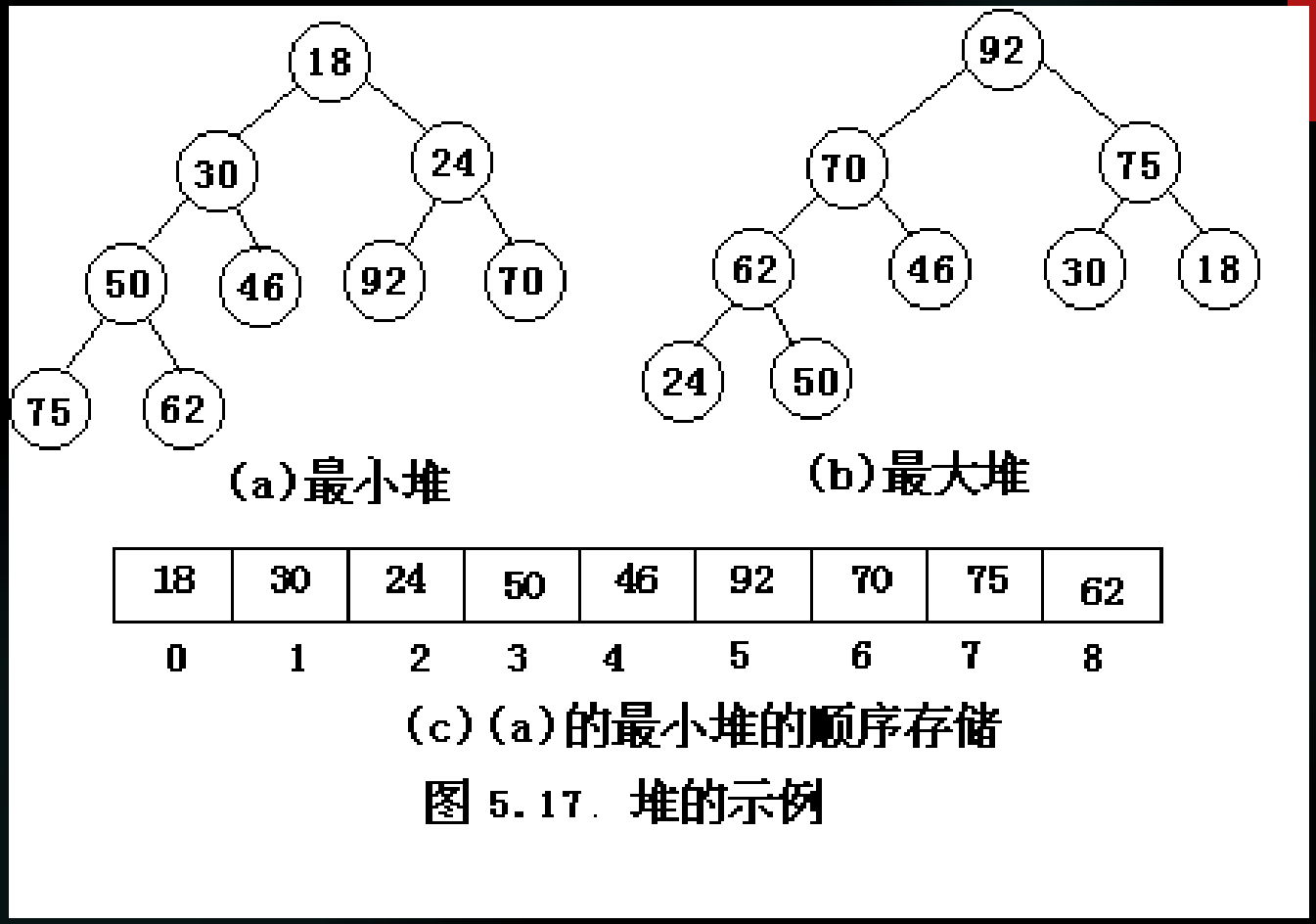


图 5.17 堆的示例

最小堆的特点：堆顶元素是整个堆的最小元素

最小堆

完全二叉树可以以顺序方式存储

堆是n个元素的序列（$k_0,k_1,\ldots,k_{n-1}$），当且仅当满足

(1) 2i+1 <n时， $k_i \le k_{2i+1}$

(2) 2i+2 <n时， $k_i \le k_{2i+2}$

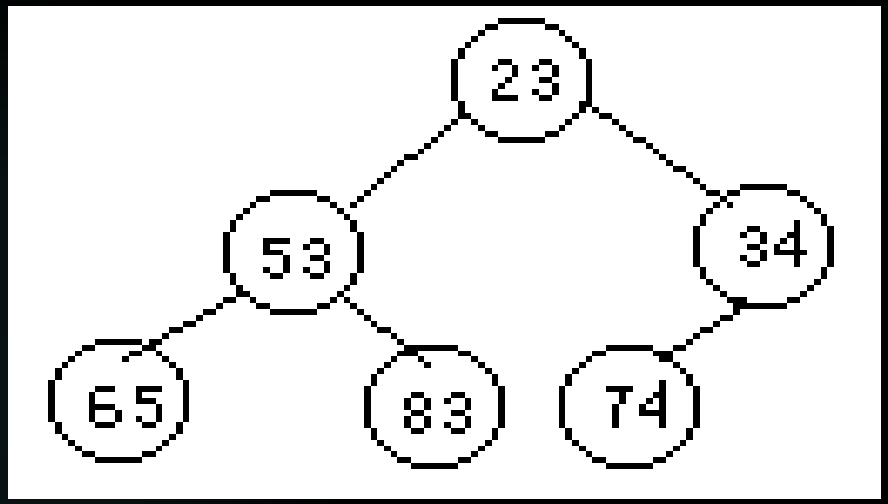
最小堆

```
Typedef struct minheap{
    int Size, MaxHeap;
    T Elements[MaxSize];
}MinHeap;
```

考查1

给定序列，判断是否最小堆?

判断序列（23，53，34，65，83，74）是最小堆吗



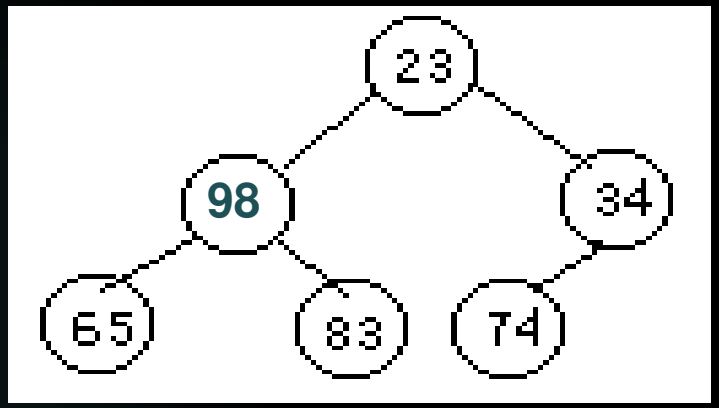
判断方法

先将序列画成完全二叉树形式

判断最小堆条件是否成立

双亲≤子女

判断序列（23，98，34，65，83，74）是最小堆吗?



考查2

给定序列不是最小堆

如何改造成唯一的最小堆?

1.将序列画成堆（完全二叉树）的形式

2.将堆调整为最小堆：

从⌊(n-2)/2⌋到0

结点i的双亲的序号为⌊(i-1)/2⌋

从完全二叉树最后一个叶子的双亲开始往前访问直到根结点

每访问一个结点，判断是否满足最小堆条件

双亲≤子女

IF 不满足，将该结点与最小孩子交换（向下调整）

交换完后再判断该结点是否满足最小堆条件

IF 不满足，继续向下调整，直到满足

3.将获得的最小堆中元素按照层次遍历顺序依次编号，并表示为一个序列

动作分解

任给定一个不满足最小堆的结点
如何向下调整

元素92向下调整的过程

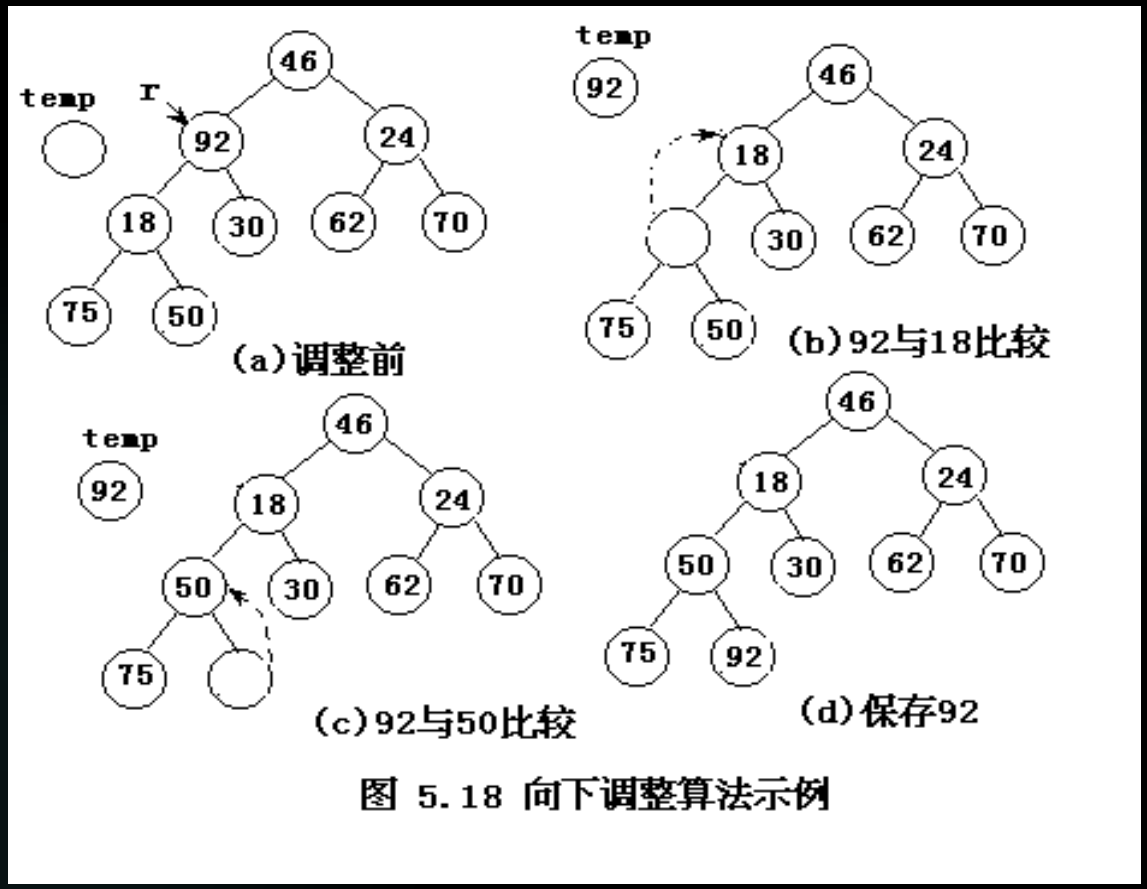
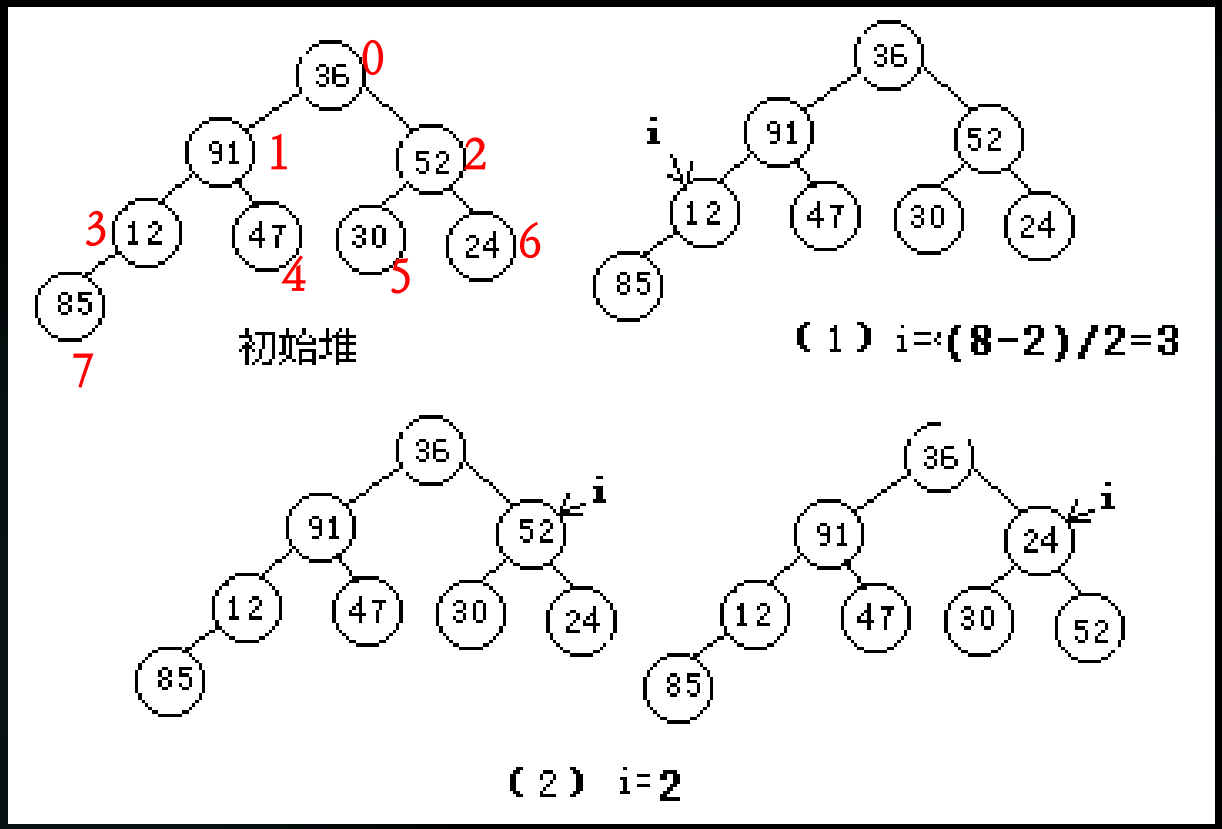
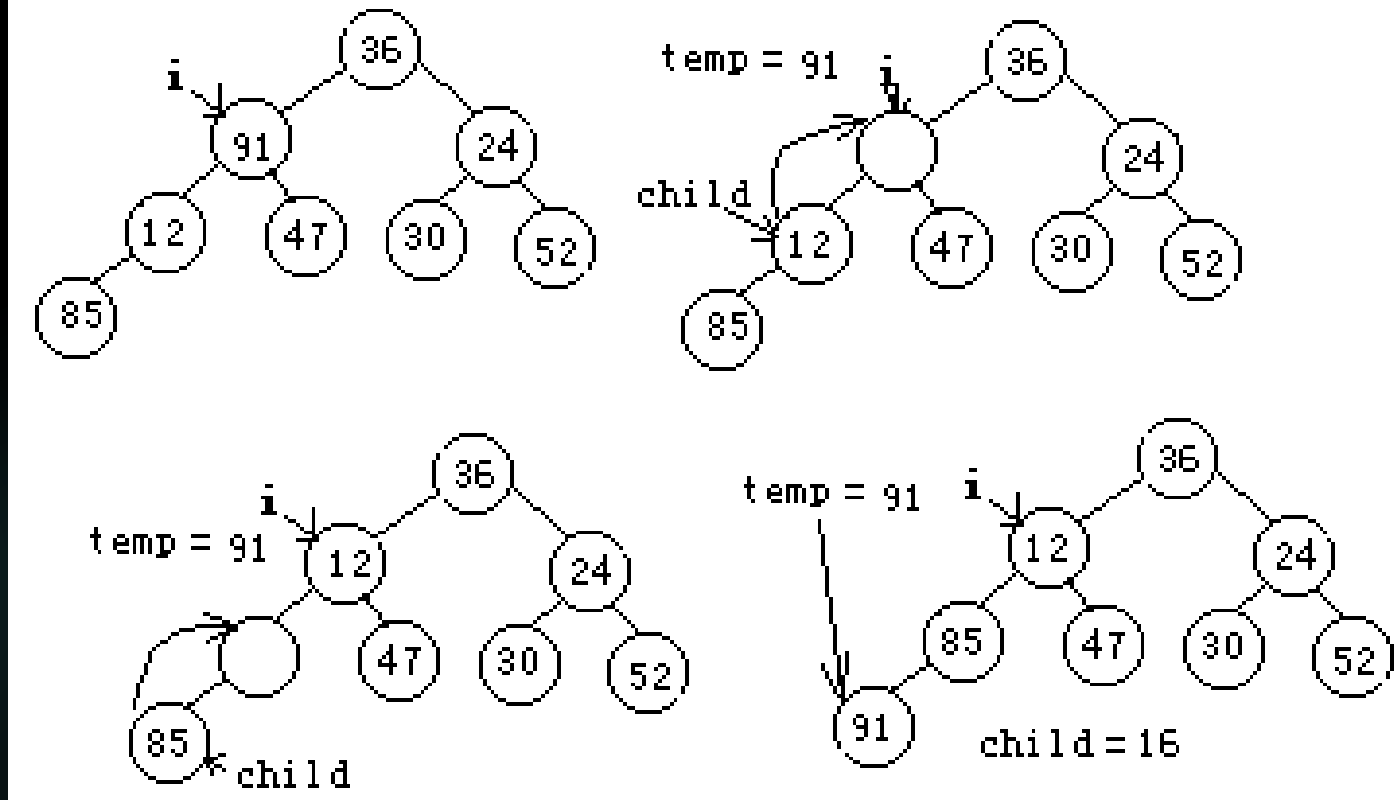


图 5.18 向下调整算法示例

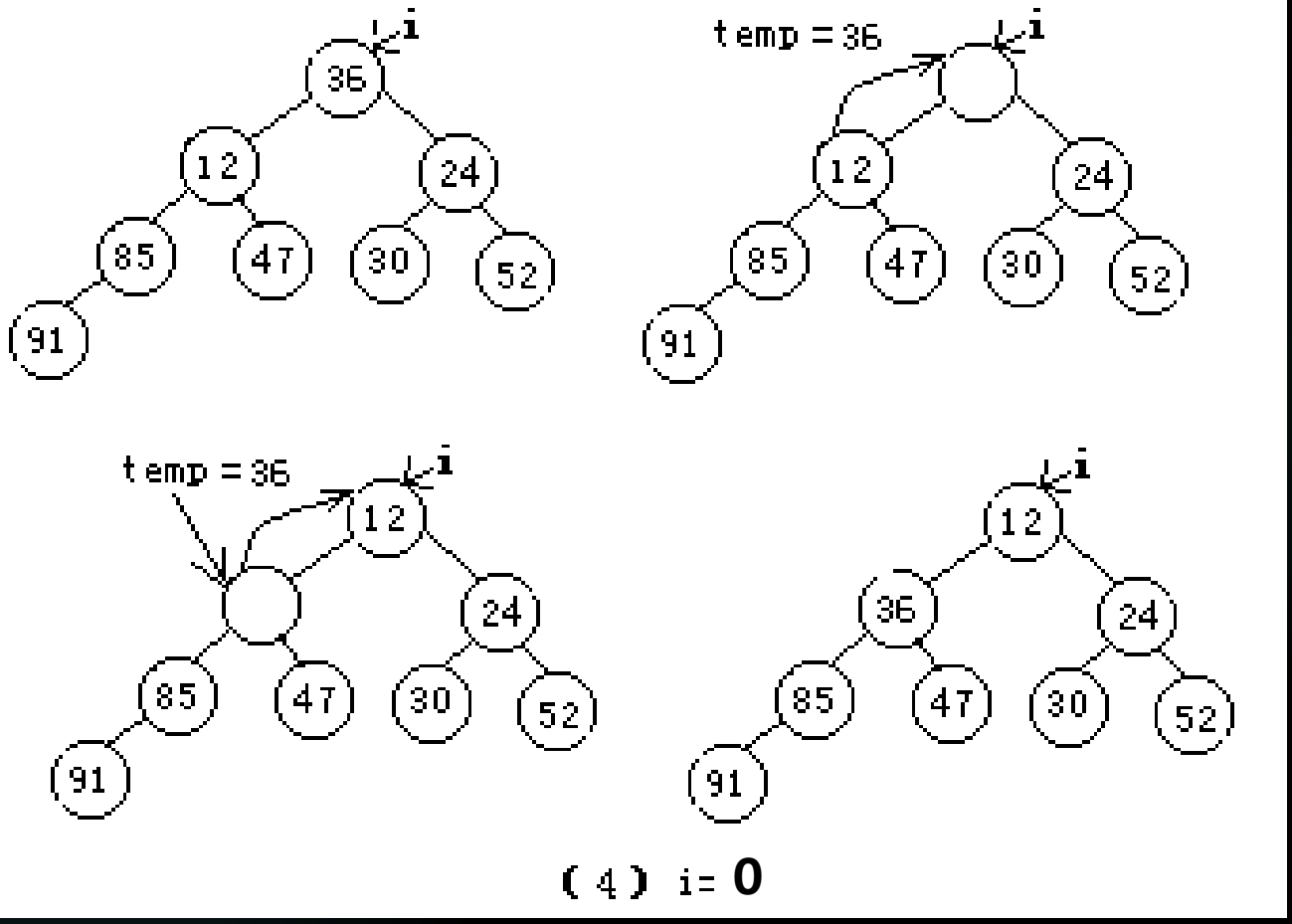
将序列(36,91,24,12,47,30,52,85）调整为最小堆

从⌊(n-2)/2⌋处的元素开始，调整⌊(n-2)/2⌋到0的每个元素。




36
i
91
24
12
47
30
52
85
temp = 91
i
36
child
24
12
47
30
52
85
temp = 91
i
36
12
24
47
30
52
85
child
temp = 91
i
36
12
24
85
47
30
52
91
child = 16

(3) i=1


i
36
12
24
85
47
30
52
91
temp = 36
i
12
24
85
47
30
52
91
temp = 36
i
12
24
85
47
30
52
91
i
12
36
24
85
47
30
52
91

(4) i= 0

优先权队列的抽象数据类型

ADT PQueue{

数据：

n≥0个元素的最小堆。

运算：

CreatePQ(PQueue *pq,int maxsize)：建立一个空队列。

IsEmpty(PQueue pq)：若队列空，则返回true；否则返回false。

IsFull(PQueue pq)：若队列满，则返回true；否则返回false

Append(PQueue *pq, T x):元素值为x的新元素入队列。

Serve(PQueue *pq, T x): 在x中返回具有最高优先权的元素值,并从优先权队列中删除该元素。

}

优先权队列中插入一个新元素的算法步骤：

1.首先将新元素插入到优先权队列的最后

2.检查新元素插入后，队列是否保持优先权队列的特点，若不能则需调整：

调整过程是由下向上，与双亲结点比较

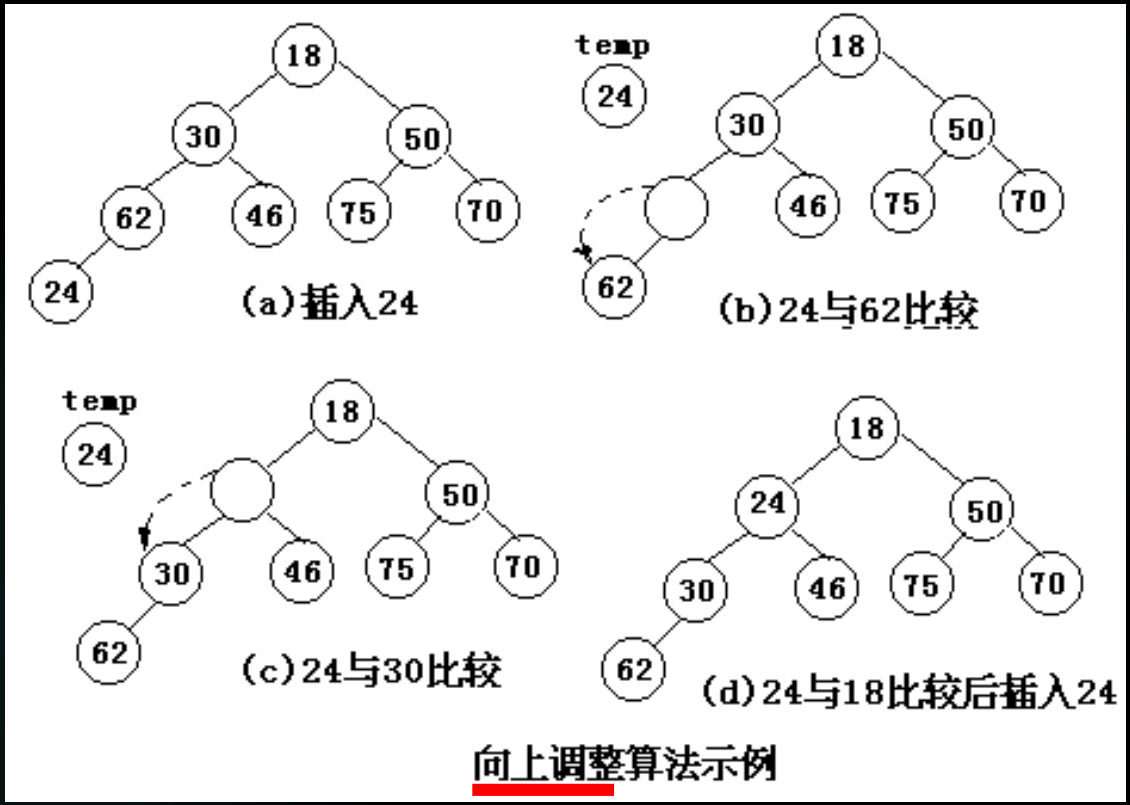
若双亲结点大则新元素上浮，双亲结点下沉。

注：这一过程中与AdjustDown相反的比较路径

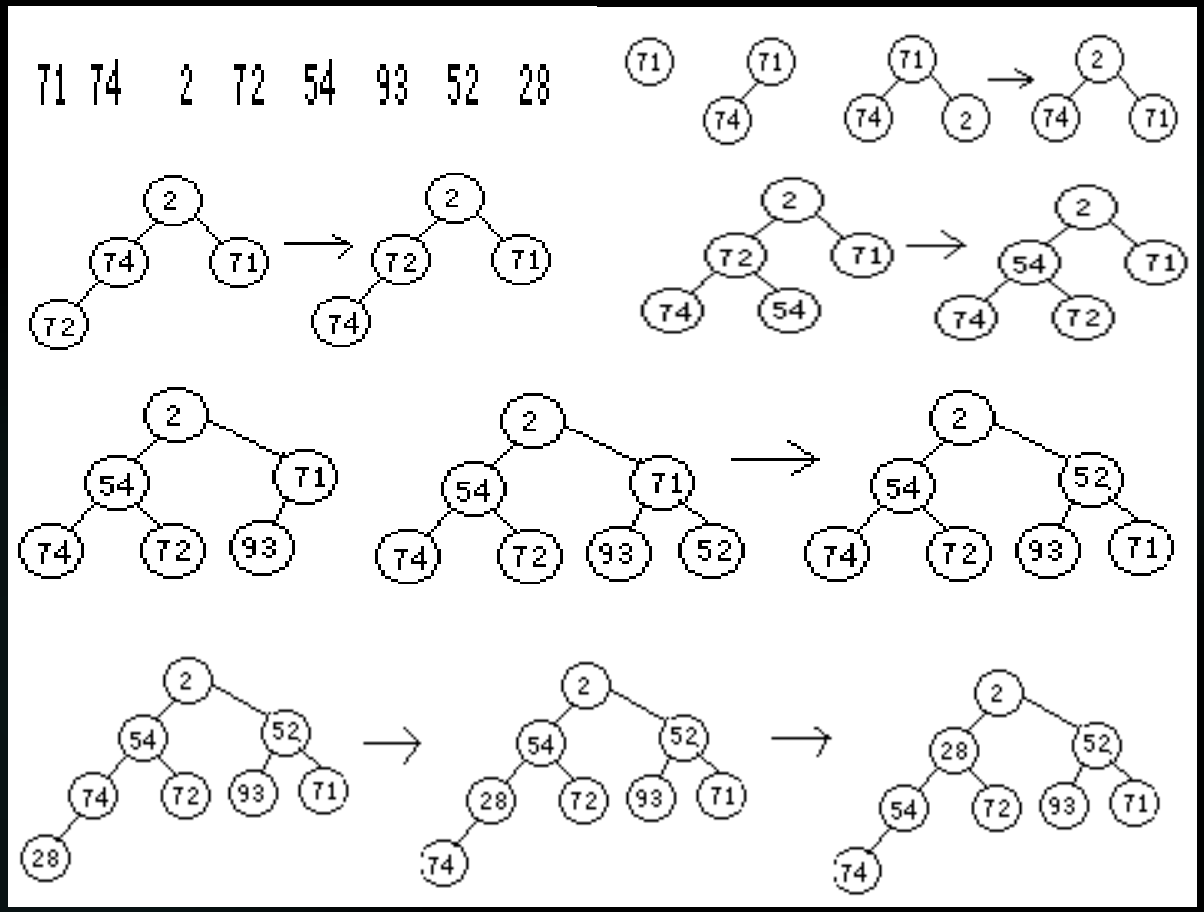
AdjustDown中调整结点与其孩子比较，不断下沉

本算法中调整结点与双亲比较，不断上浮

例 1： 向优先权队列中插入一个新元素24



从空队列开始，依此向队列中插入元素的过程



设从空队列开始，依此向队列中插入元素：71，74，2，72，54，93，52，28，则每次插入后队列的状态如下表

	0	1	2	3	4	5	6	7
1	71							
2	71	74						
3	2	74	71					
4	2	72	71	74				
5	2	54	71	74	72			
6	2	54	71	74	72	93		
7	2	54	52	74	72	93	71	
8	2	28	52	54	72	93	71	74

优先权队列中删除堆顶元素的算法步骤：

1. 将堆顶元素赋值给x，然后将堆底元素覆盖堆顶元素

2.检查堆顶元素被覆盖后，新堆顶元素是否保持优先权队列的特点，若不能则需调整：

调整过程是由上向下，与孩子结点比较

若堆顶结点大则堆顶元素下沉，孩子中小者上浮

注：这一过程中即AdjustDown过程

0	1	2	3	4	5	6	7
2	28	52	54	72	93	71	74

连续调用Serve，堆中元素如何变化？

Append 和Serve函数的时间

容易分析优先权队列的插入和删除运算的时间复杂度。

由于具有n个结点的完全二叉树的高度为

$$\lceil \log_2 (n+1) \rceil,$$

所以AdjustDown和AdjustUp两个算法中，比较和移动元素的次数均不会超过该高度。

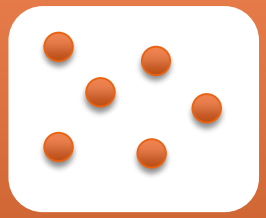
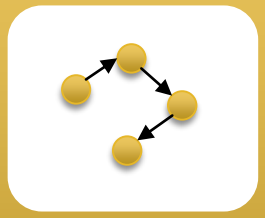
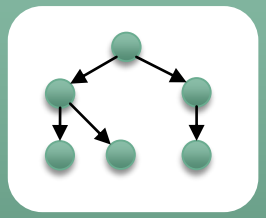
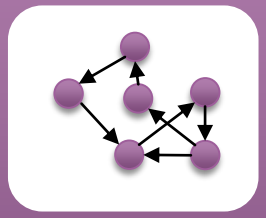
Append和Serve运算分别用一常数阶调用上述两个运算，所以它们的时间复杂度为 $O(\log_2 n)$。

集合和搜索

目录

- 集合的基本概念
- 集合的抽象数据类型
- 集合的表示形式
- 顺序搜索
- 二分搜索

基本逻辑结构

集合结构	线性结构	树形结构	图形结构
			
无关系	一对一关系	一对多关系	多对多关系

基本概念

- 集合：在数学上，集合是不同对象的无序汇集。

  例如：集合{1,2,3}与{3,2,1}相同。
- 元素：集合的对象。在集合中，每个元素仅出现一次。
- 集合运算

  求集合的并

  求集合的差

  求集合的交

  判断两集合是否相等

动态集：数据结构意义上，集合通常是动态的，在集合中可以插入和删除元素，因而称为动态集

基本概念

- 有序集：元素的汇集，其中每个元素可以出现一次或多次，并且出现次序是重要的，用()表示有序集

  (1,2,3)与(3,2,1)不同

- 多重集：元素的汇集，其中每个元素可以出现一次或多次，并且出现次序是不重要的，用()表示有序集

  集合{1,1,2,3}与{3,2,1,1}相同，但与{1,2,3} 不同

目录

- 集合的基本概念
- 集合的抽象数据类型
- 集合的表示形式
- 顺序搜索
- 二分搜索

集合元素定义

```
typedef struct entry
{
    KeyType Key;
    DataType data;
} Entry;
```

其中， KeyType 称为关键字类型，应为可比较大小的类型。

key 为关键字。

除关键字外的其他数据项归入data中。

ADT Set{

数据: 同类元素的有限汇集，其最大允许长度为MaxSet。元素由关键字标识，集合的元素各不相同。

运算:

Create()：创建一个空集合。

Destroy()：撤消一个集合。

IsEmpty()：若集合空，则返回true，否则返回false。

IsFull()：若集合满，则返回true，否则返回false。

Search(x)：在集合中搜索与x的关键字值相同的元素。如果存在该元素，则将其值赋给x，并且返回Success；否则返回NotPresent。

Insert(x): 在集合中搜索与x的关键字相同的元素。若集合中存在该元素，则将其值赋给x，函数返回Duplicate。否则，若集合已满，则函数返回Overflow；若集合未满，则在表中插入值为x的元素，函数返回Success。

Remove(x): 在集合中搜索与x的关键字值相同的元素。如果存在该元素，则将其值赋给x，并从集合中删除之，函数返回Success；否则返回NotPresent。

}

目录

- 集合的基本概念
- 集合的抽象数据类型
- 集合的表示形式
- 顺序搜索
- 二分搜索

集合的表示形式

组织集合的方法很多，组织的方法不同，实现搜索等运算的方法也不同，它直接影响运算的效率。

集合可以用线性表、搜索树、跳表和散列表表示。

本章讨论集合的线性表表示，重点讨论在于顺序表表示方式下的搜索算法。

下一章将介绍集合的搜索树表示；第八章将讨论散列表的集合。

目录

- 集合的基本概念
- 集合的抽象数据类型
- 集合的表示形式
- 顺序搜索
- 二分搜索

顺序搜索

集合可以用线性表表示。如果线性表中元素已按关键字值的大小次序排列，则称为有序表。否则为无序表。

本节将分别讨论无序表和有序表的顺序搜索算法。

无序表的顺序搜索

从头开始检查无序表，将指定元素x的关键字与表中元素的关键字比较，若相等，搜索成功；若搜索完整个表，不存在关键字值等于给定值的元素，搜索失败。

例如，在下表中分别搜索33和 35。

33 搜索成功！

(41，25，28，33，36，15)

35

(41，25，28，33，36，15)

搜索失败！

顺序搜索无序表

T为集合元素的数据类型

```
BOOL SeqSearch(List lst, KeyType k, Entry* x)
{
    int i;
    for (i=0; i<lst.Size; i++)
        if (lst.Elements[i].Key==k) {
         *x = lst.Elements[i];
         return TRUE;                    //搜索成功
    }
    return FALSE;                        //搜索失败
}
```

有序表的顺序搜索

一个有序表是一个线性表（$a_0,a_1,\cdots,a_{n-1}$），并且表中元素的关键字值有如下关系：

$$a_i.key \le a_{i+1}.key \quad (0 \le i < n-1)$$

$a_i.key$表示元素a_i的关键字域。

一个有序表可视为一个已按关键字排序的有序集

有序表的顺序搜索算法

从头开始检查有序表，将指定元素x的关键字与表中元素的关键字比较，若相等，搜索成功；若搜索到某个元素关键字大于指定元素x时，搜索失败。

例如，在下表中分别搜索33和 35。

（21，25，28，33，36，45）

33　搜索成功！

（21，25，28，33，36，45）

35 搜索失败！

顺序搜索有序表（无哨兵）

```
BOOL SeqSearch(List lst, KeyType k, Entry* x)
{
   int i;
   for (i=0; i<lst.Size; i++)
   {
         if (lst.Elements[i].Key==k) {
               *x = lst.Elements[i];
          return TRUE; }              //搜索成功
         else if(lst.Elements[i].Key>k)
               return FALSE;      //搜索失败
         }
}
```

顺序搜索有序表（有哨兵，设关键字为整数）

```
BOOL SeqSearch(List lst, KeyType k, Entry* x)
{
  int i;
  lst.Elements[lst.Size].Key = MaxNum；  //MaxNum正无穷
  for (i=0; lst.Elements[i].Key<k; i++);
  if (lst.Elements[i].Key==k) {
              *x = lst.Elements[i];
        return TRUE; }              //搜索成功
  else return FALSE;      //搜索失败
}
```

平均搜索长度

分析一个搜索算法的时间复杂度通常分成功搜索以及搜索失败两种情况加以讨论。

为了确定一个指定关键字值的记录在表中的位置所需的关键字值之间的比较次数的期望值称为搜索算法的平均搜索长度(average search length ASL) 。

1. 无序表顺序搜索

（1）成功搜索的平均搜索长度　　(21，25，28，33，36，45)

计算算法的平均搜索时间，需要给定表中元素a_i被搜索的概率p_i，假定每个元素的搜索概率是相等的，即$p_i=1/n$，则搜索成功时的平均搜索长度为

$$ASL_s=\sum_{i=1}^{n} i\times p_i=\frac{1}{n}\sum_{i=1}^{n} i=\frac{n+1}{2}$$

（2）搜索失败的平均搜索长度

该函数在搜索失败的情况下，总要进行n 次关键字值之间的比较。

2. 有序表顺序搜索

（1）成功搜索的平均搜索长度

搜索成功时的平均搜索长度大致与搜索无序表相同。

$$ASL_s = \sum_{i=1}^{n} i \times p_i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

（2）搜索失败的时间复杂度

在搜索失败的情况下，平均搜索长度大约比无序时快一倍。

$$(21，25，28，33，36，45，\ +\infty)$$

$$(-\infty, 21), (21,25), (25,28), (28,33), (33,36), (36,45), (45, +\infty)$$

$$ASL_F = 1 + \sum_{i=1}^{n+1} i \times \frac{1}{n+1} = 2 + \frac{n}{2}$$

目录

- 集合的基本概念
- 集合的抽象数据类型
- 集合的表示形式
- 顺序搜索
- 二分搜索

二分搜索基本思想

有序表（$a_0,a_1,a_2,\ldots,a_{n-1}$）中搜索x

- 如果有序表长等于0，搜索失败
- 二分搜索有序表($a_0,a_1,a_2,\ldots,a_{n-1}$)
  - 如果有序表长>0，取表中某个元素a_m与x进行比较
  - 如果如果a_m.key =x.key，搜索成功，返回
  - 如果a_m.key >x.key，二分搜索有序表($a_0,a_1,a_2,\ldots,a_{m-1}$)
  - 如果a_m .key <x.key，二分搜索有序表($a_{m+1},a_{m+2},\ldots,a_{n-1}$)

选取分割点

对半搜索

- 对半搜索是二分搜索中的一种。分割点为表的中点元素

$$(a_{low}, a_{low+1}, \ldots, a_{high})$$

m=(low+high)/2

对半搜索基本思想

有序表（$a_0,a_1,a_2,\ldots,a_{n-1}$）中搜索x

- 如果有序表长等于0，搜索失败
- 二分搜索有序表($a_0,a_1,a_2,\ldots,a_{n-1}$)

Low=0,high=n-1

  - 如果有序表长>0，取表元素$a_{(low+high)/2}$与x进行比较
  - 如果如果a_m.key =x.key，搜索成功，返回
  - 如果a_m.key > x.key ，二分搜索有序表($a_0,a_1,a_2,\ldots,a_{(low+high)/2-1}$)
  - 如果a_m .key < x.key ，二分搜索有序表($a_{(low+high)/2+1},a_{(low+high)/2+2},\ldots,a_{n-1}$)

查找key=66的数据元素

元素下标	0	1	2	3	4	5	6	7	8	9
元素key	21	30	36	41	52	54	66	72	83	97

第一趟 Low=0, high=9, m=4

取出a_4的key与66比较：52<66

5	6	7	8	9
54	66	72	83	97

第二趟 Low=5, high=9, m=7

取出a_7的key与66比较：72>66

查找key=66的数据元素

元素下标	0	1	2	3	4	5	6	7	8	9
元素key	21	30	36	41	52	54	66	72	83	97

5	6
54	66

第三趟 Low=5, high=6, m=5

取出a_5的key与66比较：54<66

6
66

第四趟 Low=6, high=6, m=6

取出a_6的key与66比较：搜索成功！

查找key=35的数据元素

元素下标	0	1	2	3	4	5	6	7	8	9
元素key	21	30	36	41	52	54	66	72	83	97

第一趟 Low=0, high=9, m=4

取出a_4的key与35比较：52>35

0	1	2	3
21	30	36	41

第二趟 Low=0, high=3, m=1

取出a_1的key与35比较：30<35

查找key=35的数据元素

元素下标	0	1	2	3	4	5	6	7	8	9
元素key	21	30	36	41	52	54	66	72	83	97

2	3
36	41

第三趟 Low=2, high=3, m=2
取出a_2的key与35比较：36>35

第四趟 空表，搜索失败！

对半搜索的递归算法-主函数调用

```
BOOL BSearch(List lst, KeyType k, Entry* x)
 {
   int i=BSearch(lst, k, 0, lst.Size-1);
   if (i==-1) return FALSE;
   *x=lst.Elements[i];
   return TRUE;
 }
```

对半搜索的递归算法-递归函数调用

```
int BSearch(List lst, KeyType k, int low, int high)
{
  if  (low<=high)
  {
      int mid=(low+high)/2;                              //对半分割
      if (k<lst.Elements[mid].Key) return BSearch(lst, k, low, mid-1);
    else if (k>lst.Elements[mid].Key) return BSearch(lst, k, mid+1, high);
        else return mid;                                 //搜索成功
  }
   return -1;                                            //搜索失败
}
```

对半搜索的迭代算法

```
BOOL BSearch(List lst, KeyType k, Entry* x)
{
      int mid, low=0, high=lst.Size-1;
      while  (low<=high)
     {
          mid=(low+high)/2;
          if (k<lst.Elements[mid].Key) high=mid-1;
          else if (k>lst.Elements[mid].Key) low=m+1;
               else {
               *x=lst.Elements[mid].;
               return TRUE;            //搜索成功
              }
     }
     return FALSE;                                  //搜索失败
}
```

对半搜索的搜索长度分析

- 可以建立一棵二叉判定树来模拟对半搜索执行过程

- 二叉判定树的构造

（1）根结点为首次选取的分割点r，分割点r将表T分为左表与右表

（2）在左表中选取分割点s作为上一个分割点r的左孩子，分割点s将表T分为左表与右表，继续步骤（2）直到左表与右表为空

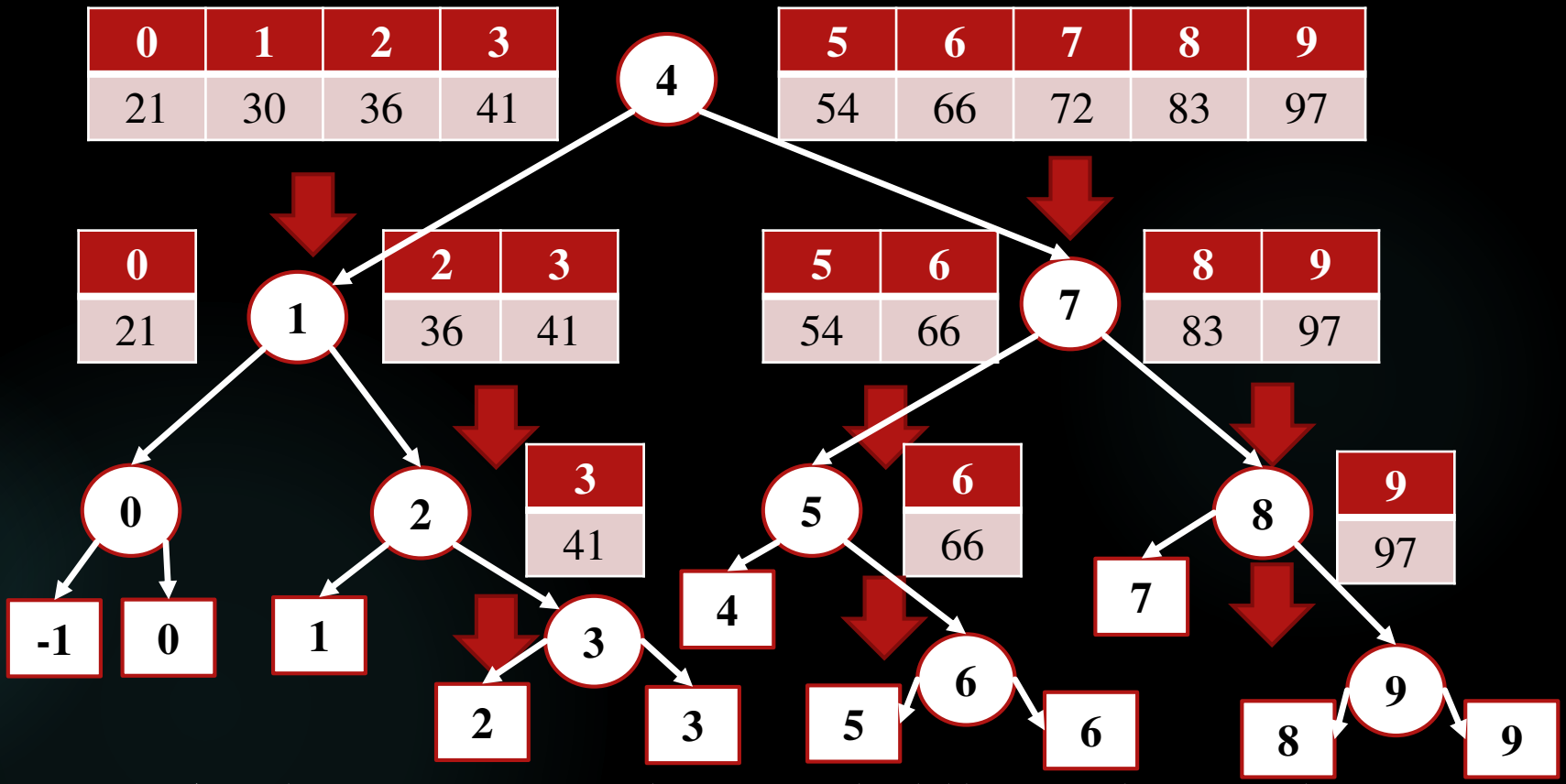
（3）在右表中选取分割点s作为上一个分割点r的右孩子，分割点s将表T分为左表与右表，继续步骤（3）直到左表与右表为空

0	1	2	3	4	5	6	7	8	9
21	30	36	41	52	54	66	72	83	97

0	1	2	3
21	30	36	41

4

5	6	7	8	9
54	66	72	83	97



（5）构造到某个分割点m时，左表为空，则为其构造外结点m-1为左孩子

（6）构造到某个分割点m时，右表为空，则为其构造外结点m为右孩子

- 搜索（T, k）

（1）将（l=根结点的关键字）与（k）进行比较

（2）如果l=k，成功搜索，返回

（4）如果l>k，如果T的左子树不为空，搜索（T的左子树，k）

否则，搜索失败，返回

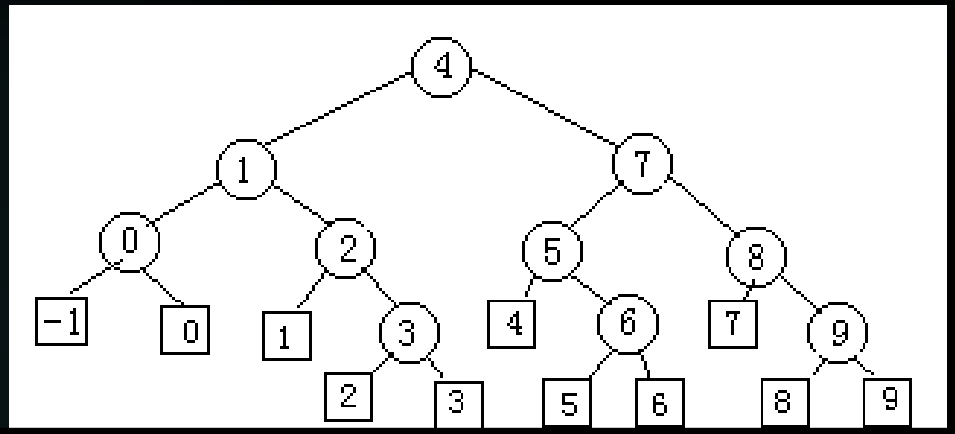
（5）如果l<k，如果T的右子树不为空，搜索（T的右子树，k）

否则，搜索失败，返回

如果搜索成功，则算法在内结点处终止；否则算法在外结点处终止

定理 对半搜索算法在成功搜索的情况下，关键字值之间的比较次数不超过$\lfloor \log_2 n \rfloor + 1$。对于不成功的搜索，算法需要作$\lfloor \log_2 n \rfloor$或$\lfloor \log_2 n \rfloor + 1$次比较。

定理 对半搜索算法在搜索成功时的平均时间复杂度为$O(\log_2 n)$



搜索树

目录

- 二叉搜索树
- 二叉平衡树
- B-树

用树形结构表示集合，可有效提高搜索效率，因此比线性表更适合表示动态集

采用什么样的树形结构表示集合?

如何在树形结构上进行数据元素的插入和删除?

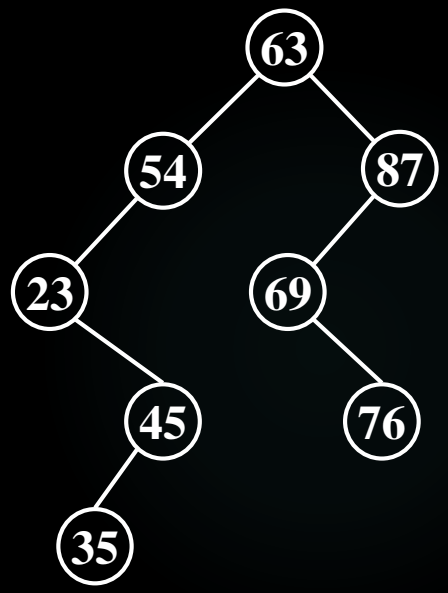
二叉搜索树

定义 设结点由关键字值表征，假定所有结点的**关键字值各不相同**，二叉搜索树或者是一棵空二叉树，或者是具有下列性质的二叉树：

(1)若左子树不空，则**左**子树上**所有**结点的关键字值均**小于根**结点关键字值；

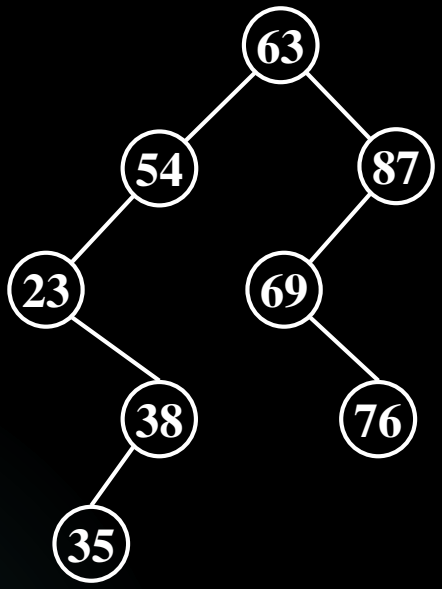
(2)若右子树不空，则**右**子树上**所有**结点的关键字值均**大于**根结点关键字值；

(3)左、右子树也分别是二叉搜索树。



左小右大

给出二叉搜索树的递归定义



中序遍历:23 35 38 54 63 69 76 87

性质 若以中序遍历一棵二叉搜索树，将得到一个以关键字值**递增**排列的有序序列。

二叉搜索树类型实现动态集

```
typedef int KeyType;
typedef struct entry{
    KeyType Key;
    DataType Data ;
 }Entry;
typedef Entry T;
```

定义值集合项类型K，包含值与键

```
typedef  struct btnode{
    T Element;
    struct btnode*LChild,*RChild;
}BTNode;
```

定义值搜索树节点类型BTNode，包含集合项、左右子树根节点指针

```
typedef  struct btree{
     BTNode* Root;
}BTree;
```

定义值搜索树类型BTree，包含根节点指针

1. 二叉搜索树搜索的递归算法
2. 二叉搜索树搜索的迭代算法

1. 二叉搜索树搜索的递归算法

在一棵二叉搜索树上，查找与关键字为x的元素

(1) 若二叉树为空，则搜索失败。

(2) 否则，将x与根结点比较，

- 若x小于该结点的关键字，则以同样的方法搜索左子树，而不必搜索右子树；
- 若x大于该结点的关键字，则以同样的方法搜索右子树，而不必搜索左子树；
- 若x等于该结点的关键字，则搜索成功终止。

二叉搜索树搜索的递归算法

```
BTNode* Find( BTNode *p, KeyType k )
{       if ( !p ) return NULL;                              /*搜索失败*/
        if ( k==p->Element.Key) return p;                   /*搜索成功*/
        if ( k < p->Element.Key) return Find ( p->LChild, k );
        return Find( p->RChild, k); }

BOOL BtSearch(BTree Bt, KeyType k, K *x)
{
        BTNode* p= Find(Bt.Root, k);
        if (p) { *x=p->Element;   return TRUE; }   /*搜索成功*/
    else return FALSE;                                  /*搜索失败*/
}
```

2.二叉搜索树的迭代算法

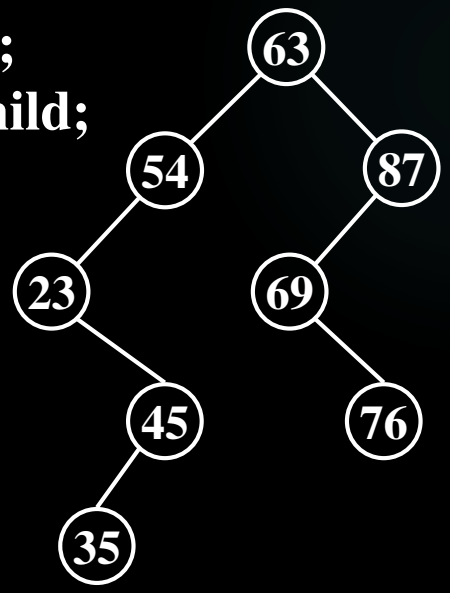
二叉搜索树的迭代算法使用while循环，从根结点开始搜索，将待查元素x与当前元素比较。

- 若x等于该结点的值，则搜索成功终止；
- 若x小于该结点的值，则继续搜索左子树；
- 若x大于该结点的值，则继续搜索右子树。

只有搜索到空子树时，才失败终止。

程序 二叉搜索树搜索的迭代算法

```
BOOL BtSearch(Btree Bt, KeyType k, T* x)
{
   BTNode *p=Bt.Root;
   while (p)
   {   if ( k<p->Element.Key)  p=p->lChild;
       else if(k>p->Element.Key)  p=p->rChild;
            else
            {  *x=p->element;
               return TRUE;
            }
   }
   return FALSE;
}
```



二叉搜索的插入操作

1. 可用于从空树开始构造二叉搜索树

2. 可用于向集合插入具有新关键字的数据元素

二叉搜索树的插入步骤与单链表的插入步骤类似。

(1) 查找插入元素的位置；

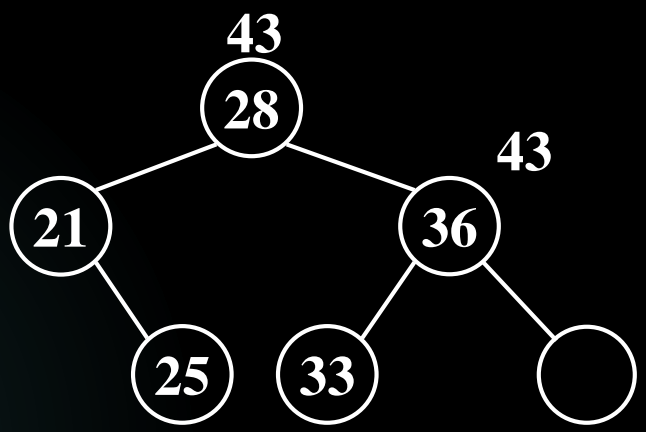
怎么选择该位置?

(2) 生成新结点；

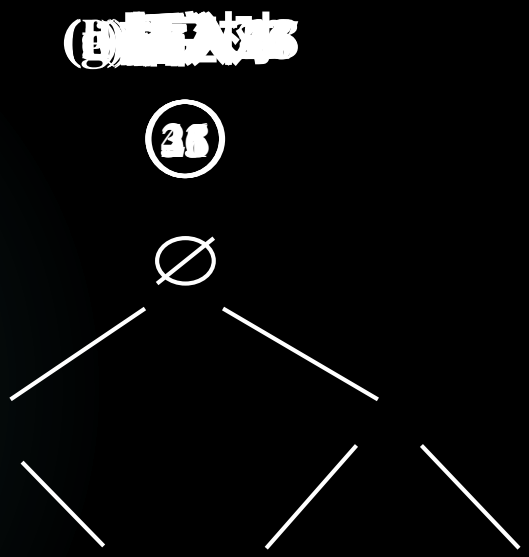
(3) 插入新结点(有可能修改root指针)

插入后是否有后续调整动作?

在二叉搜索树中插入43的执行过程:



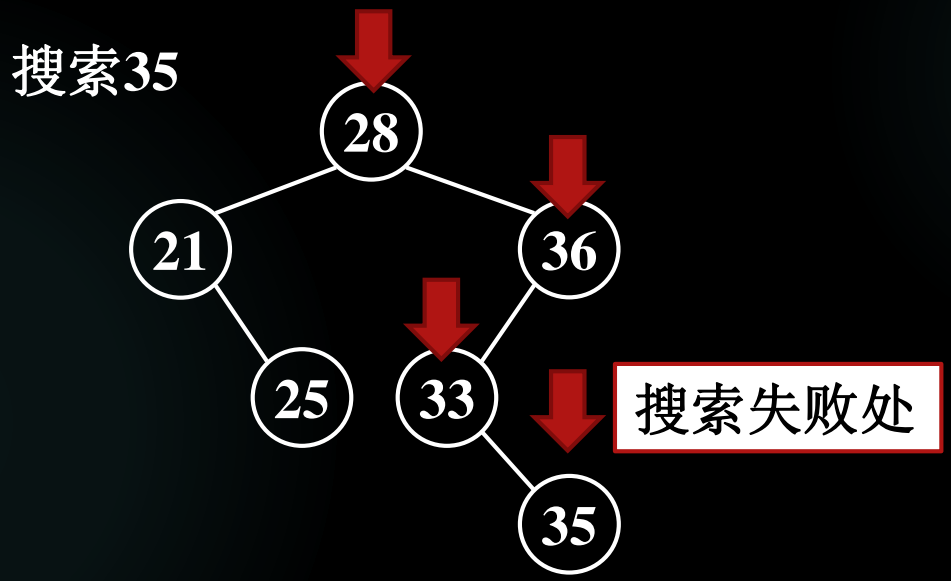
下图显示了从空树开始通过依次**插入**元素，**构造**一棵二叉搜索树的过程。



(1)查找插入元素x的位置

尝试自根结点相下搜索x:

搜索失败的位置处作为x的插入位置



(1)查找插入元素x的位置

尝试自根结点相下搜索x:

搜索失败的位置处作为x的插入位置

- 如果二叉树中有重复元素，应返回Duplicate。
- 搜索到达空子树，则表明树中不包含重复元素。

(2)生成新结点。算法构造一个新结点p存放新元素x

(3)插入新结点。新结点连至双亲结点q，成为q的孩子。至于p是q的左孩子还是右孩子，这取决于x与q关键字值的大小。

如果原二叉搜索树是空树，则新结点p成为新二叉搜索树的根。

程序 二叉搜索树的插入运算

```
BOOL  Insert(Btree *Bt, T x)
{  BTNode *p=Bt->Root,*q, *r;
   KeyType k = x.Key;
   while (p)
    {  q=p;
      if (k<p->Element.Key) p=p->LChild;
      else if(k>p->Element.Key) p=p->RChild;
           else {return FALSE; }
    }
    r=NewNode2(x);
    if(!Bt->Root) root=r;
    else if(k<q->Element.Key) q->LChild=r;
         else q->RChild=r;
    return TRUE;
}
```

q指向p的双亲

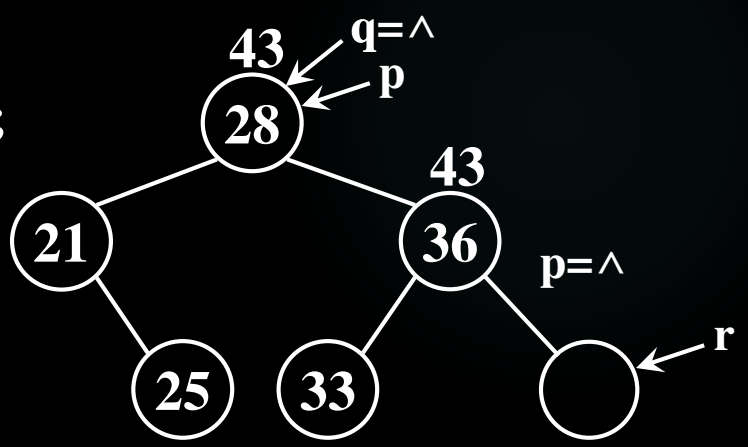
查找插入位置

生成新结点

插入新结点

```
BOOL  Insert(Btree *Bt, T x)
{  BTNode *p=Bt->Root,*q, *r;
   KeyType k = x.Key;
   while (p)
    {  q=p;
      if (k<p->Element.Key) p=p->LChild;
      else if(k>p->Element.Key) p=p->RChild;
            else {return FALSE; }
    }
   r=NewNode2(x);
   if(!Bt->Root) root=r;
   else if(k<q->Element.Key) q->LChild=r;
         else q->RChild=r;
   return TRUE;
}
```

在二叉搜索树中插入43的执行过程:

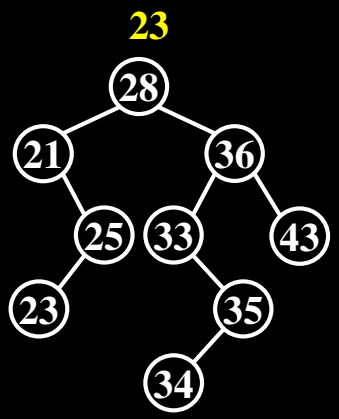


二叉搜索的删除操作

1. 可能是简单操作：删除叶子
2. 可能是复杂操作：删除有一个孩子的结点
3. 可能是更加复杂的操作：删除有两个孩子的结点

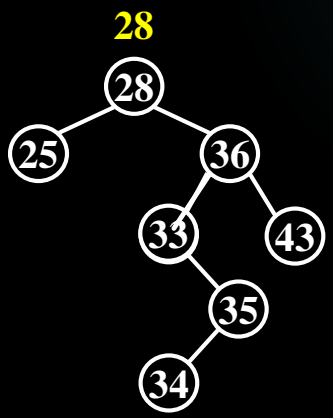
二叉搜索树上删除元素

删除叶子结点



删除只有一个孩子的结点

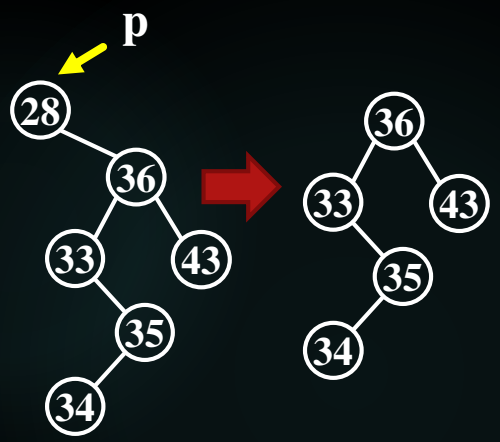
删除有两个孩子的结点



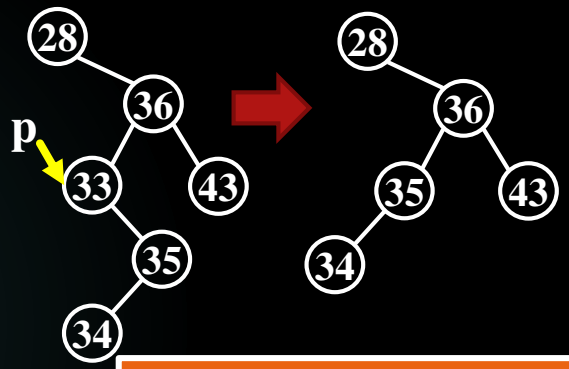
如果不存在指定删除的元素，应返回NotPresent。

删除结点p的操作由下列几步组成：

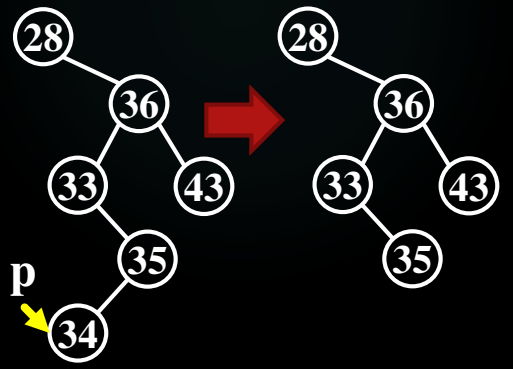
(1)若结点p只有一棵非空子树或p是叶子，以结点p的唯一孩子（设为结点c）或空子树（c=NULL）取代p的位置。



原来p是根，现在p的唯一孩子36变成根



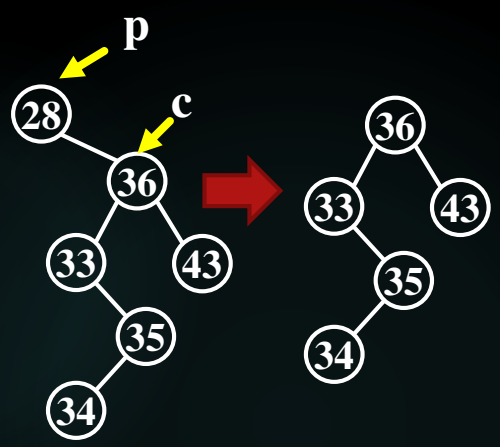
原来p是36的左孩子，现在p的唯一孩子35变成36的左孩子



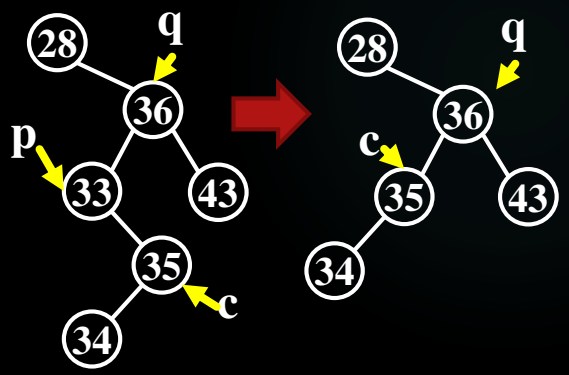
叶子结点直接删除

若被删除的结点p是根结点，则删除后，结点c成为新的根

否则，若p是其双亲q的左孩子，则结点c也应该成为q的左孩子，不然c成为q的右孩子



原来p是根，现在p的唯一孩子36变成根

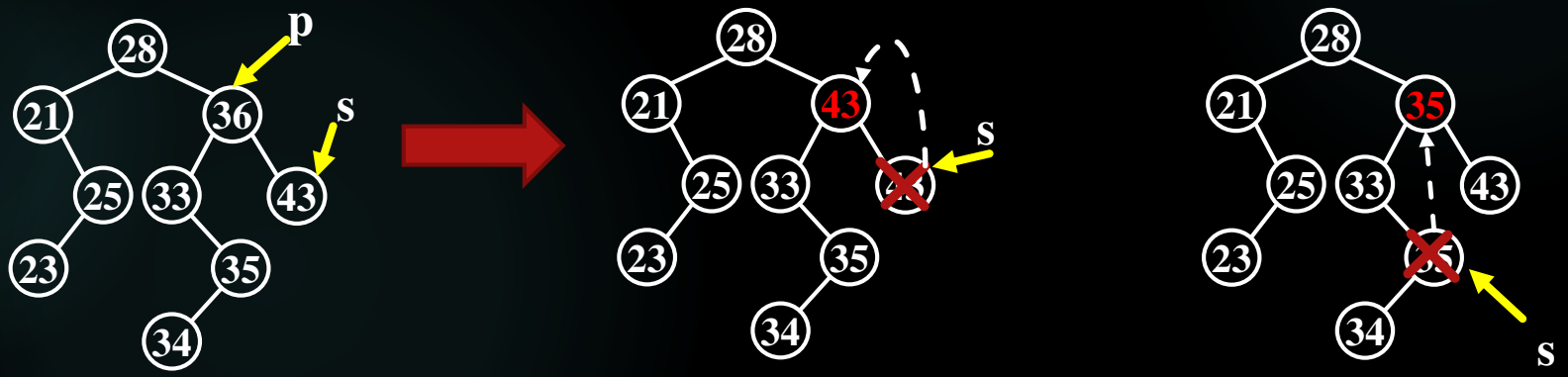


原来p是36的左孩子，现在p的唯一孩子35变成36的左孩子

(2) 若结点p有**两棵非空子树**，

搜索结点p的中序遍历次序下的直接后继（或直接前驱）结点s

将s中的值复制到p中，删除s结点。s结点如果是p的直接后继（前驱），则s结点肯定没有左孩子（右孩子）。这样，问题就化为“被删除的结点最多只有一棵非空子树”的情形。



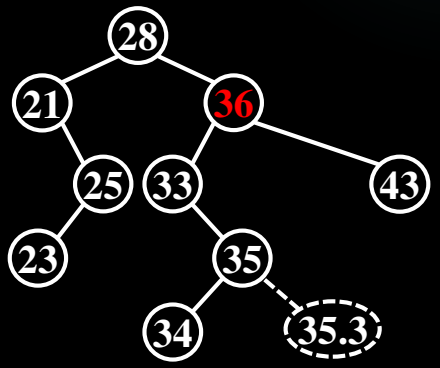
中序遍历：21, 23, 25, 28, 33, 34, 35, 36, 43

s结点如果是p的直接后继（前驱），则s结点肯定没有左孩子（右孩子）。这样，问题就化为“被删除的结点最多只有一棵非空子树”的情形。

中序遍历：21, 23, 25, 28, 33, 34, 35, 36, 43

任意一个结点q，如果其直接后继结点s有左孩子l，那么中序遍历完q之后一定先遍历l，后遍历s，则s不可能成为q的直接后继结点

任意一个结点q，如果其直接前驱结点s有右孩子r，那么中序遍历完s之后一定先遍历r，后遍历q，则s不可能成为q的直接前驱结点



不要忘记最后释放结点p所占的空间

练习

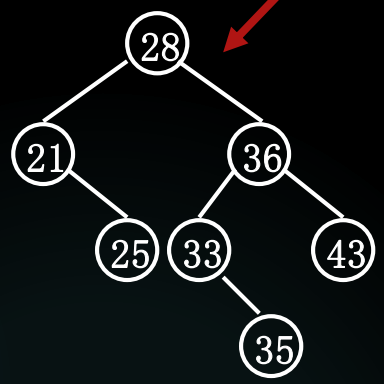
28,21,36,25,33,35,43　构造二叉搜索树之后，删除36

28,21,25,33,43,36,35　构造二叉搜索树之后，删除28

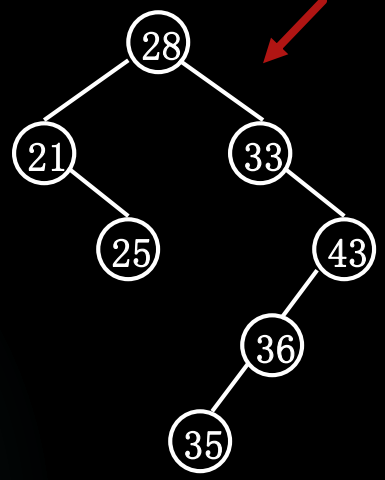
21,25,28,33,35,36,43　构造二叉搜索树之后，删除33

二叉搜索树搜索算法分析

输入：28, 21, 36, 25, 33, 35, 43.



输入：28, 21, 25, 33, 43, 36, 35.



退化成线性表顺序搜索

输入：21, 25, 28, 33, 35, 36, 43.

最好情况和一般情况：$O(\log_2 n)$

最坏情况：单支树，$O(n)$

搜索树

目录

- 二叉搜索树
- 二叉平衡树
- B-树

二叉平衡树是一种特殊的二叉搜索树

二叉平衡树能有效地控制树的高度，避免产生普通搜索树的“退化”树形。

只介绍二叉平衡树的插入，不介绍二叉平衡树的删除。

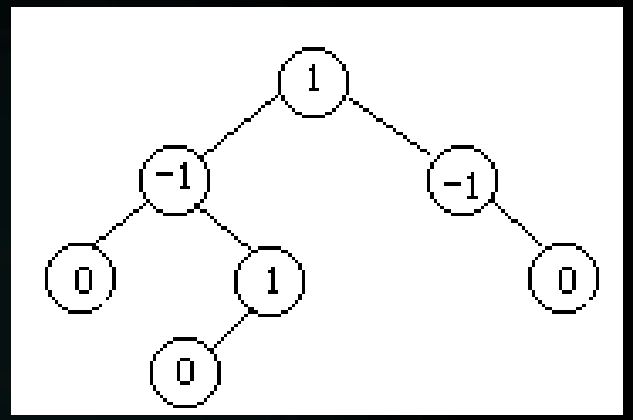
二叉平衡树的定义

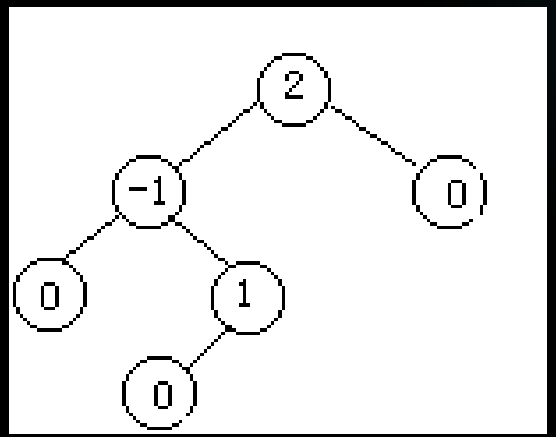
- 定义　二叉平衡树又称AVL树

  它或者是一棵空二叉树，或者是具有下列性质的二叉树：

  （1）其根的左、右子树高度之差的绝对值不超过1；

  （2）其根的左、右子树都是二叉平衡树。





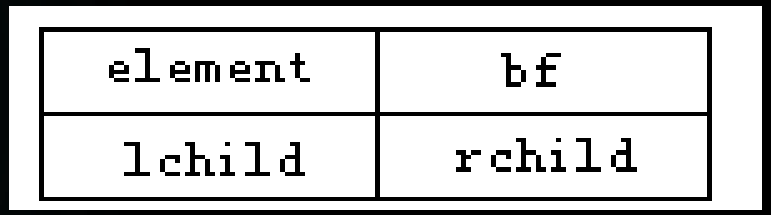
- 结点的平衡因子定义为该结点的左子树的高度减去右子树的高度
- AVL二叉搜索树既是二叉搜索树又是AVL树，具有平衡性和排序性。

二叉平衡树的存储表示

AVL树可以采用普通二叉树的二叉链表存储

每个结点增加一个成员变量：平衡因子bf

二叉平衡树结点结构



AVLNode 平衡树结点类型

```
Typedef struct avlnode
{
    K element;
    int Bf;              //结点的平衡因子
    AVLNode* LChild,*RChild;
}AVLNode;
```

二叉平衡树的搜索和一般二叉树的搜索一样

回顾

二叉搜索树的插入步骤与单链表的插入步骤类似。

(1) 查找插入元素的位置；

怎么选择该位置？

(2) 生成新结点；

(3) 插入新结点(有可能修改root指针)

插入后是否有后续调整动作？

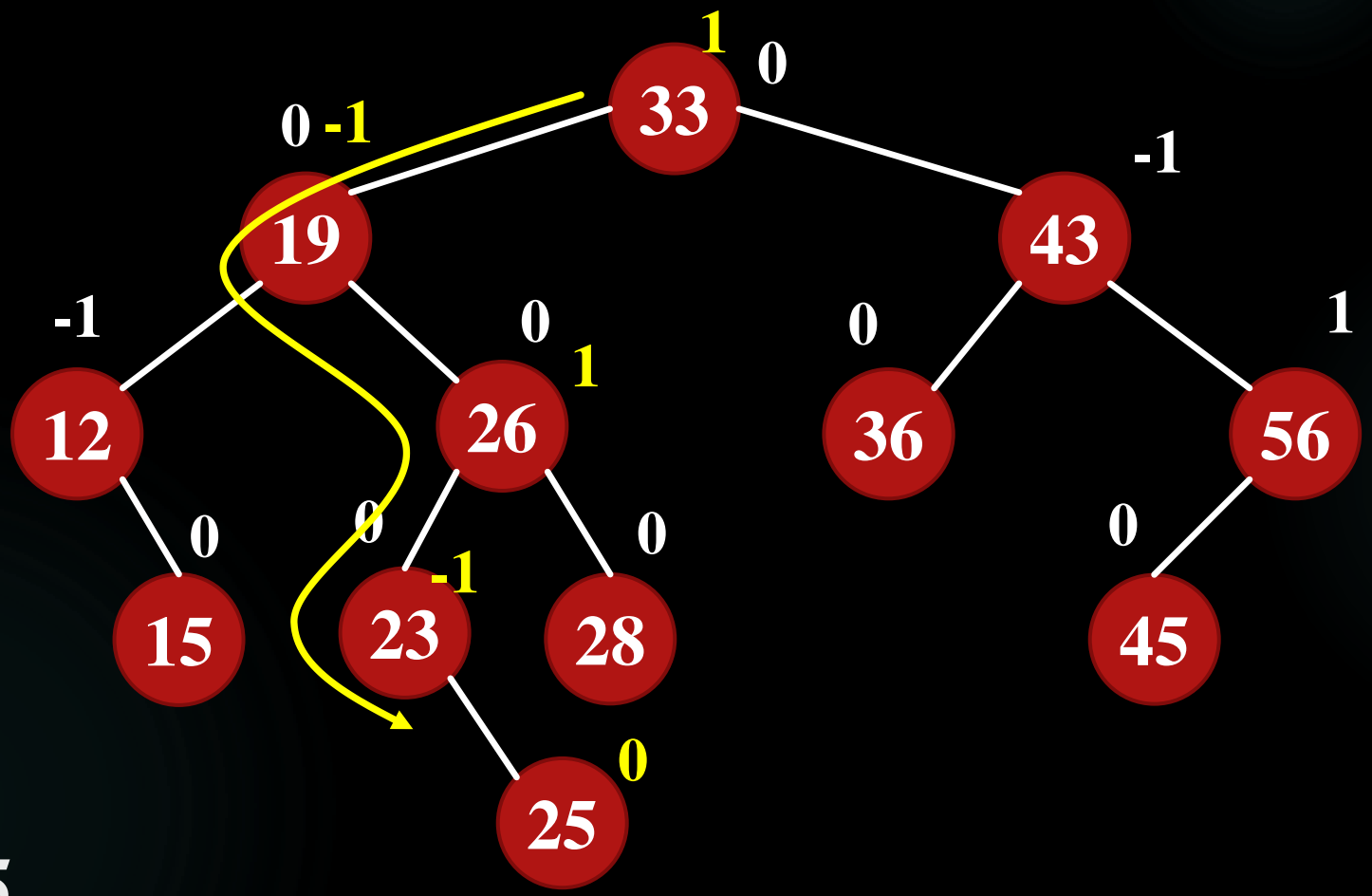
二叉平衡树的平衡旋转

二叉平衡树插入新元素步骤：

（1）二叉平衡树的插入可先按普通二叉搜索树的插入方法插入结点

（2）新结点设为q，但插入新结点后的新树可能不再是AVL树，这时需要重新平衡，使之仍具平衡性和排序性

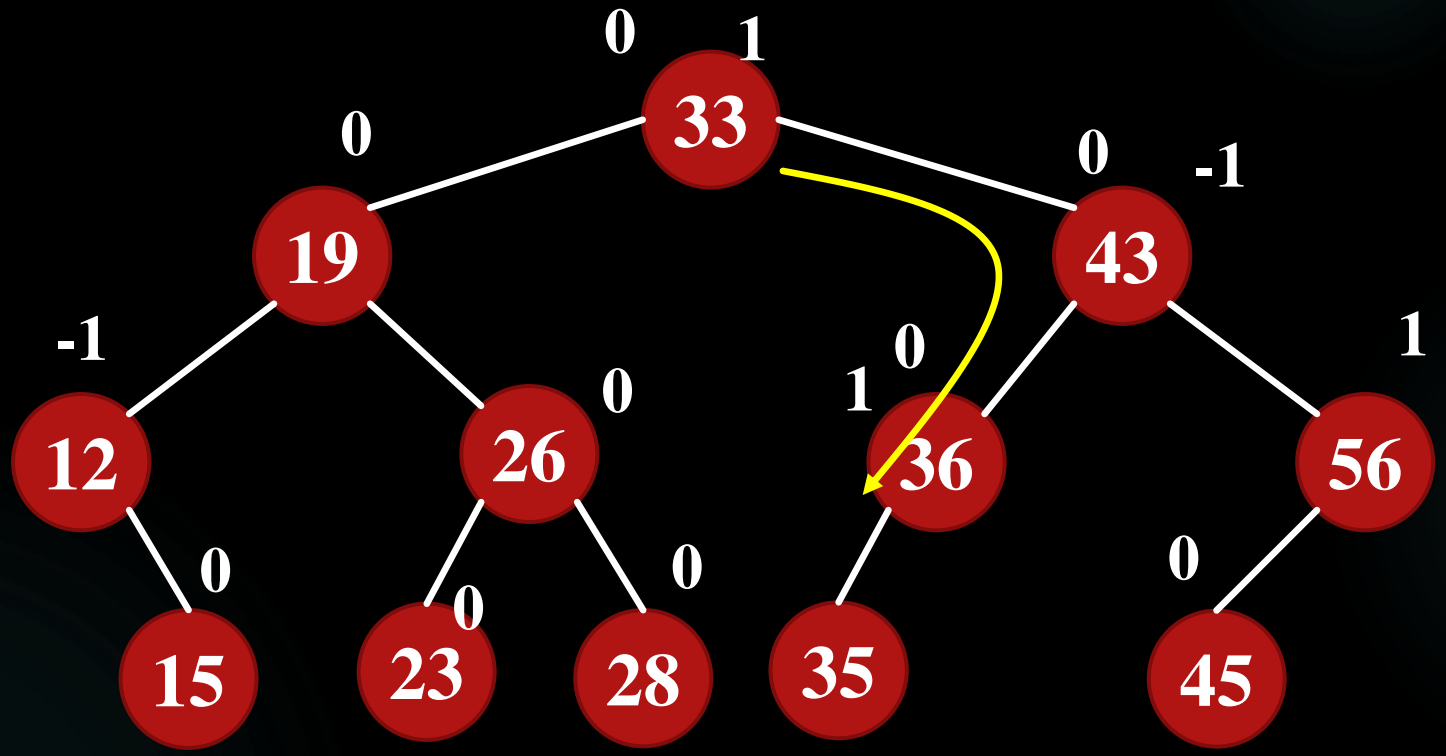
插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。



插入：25

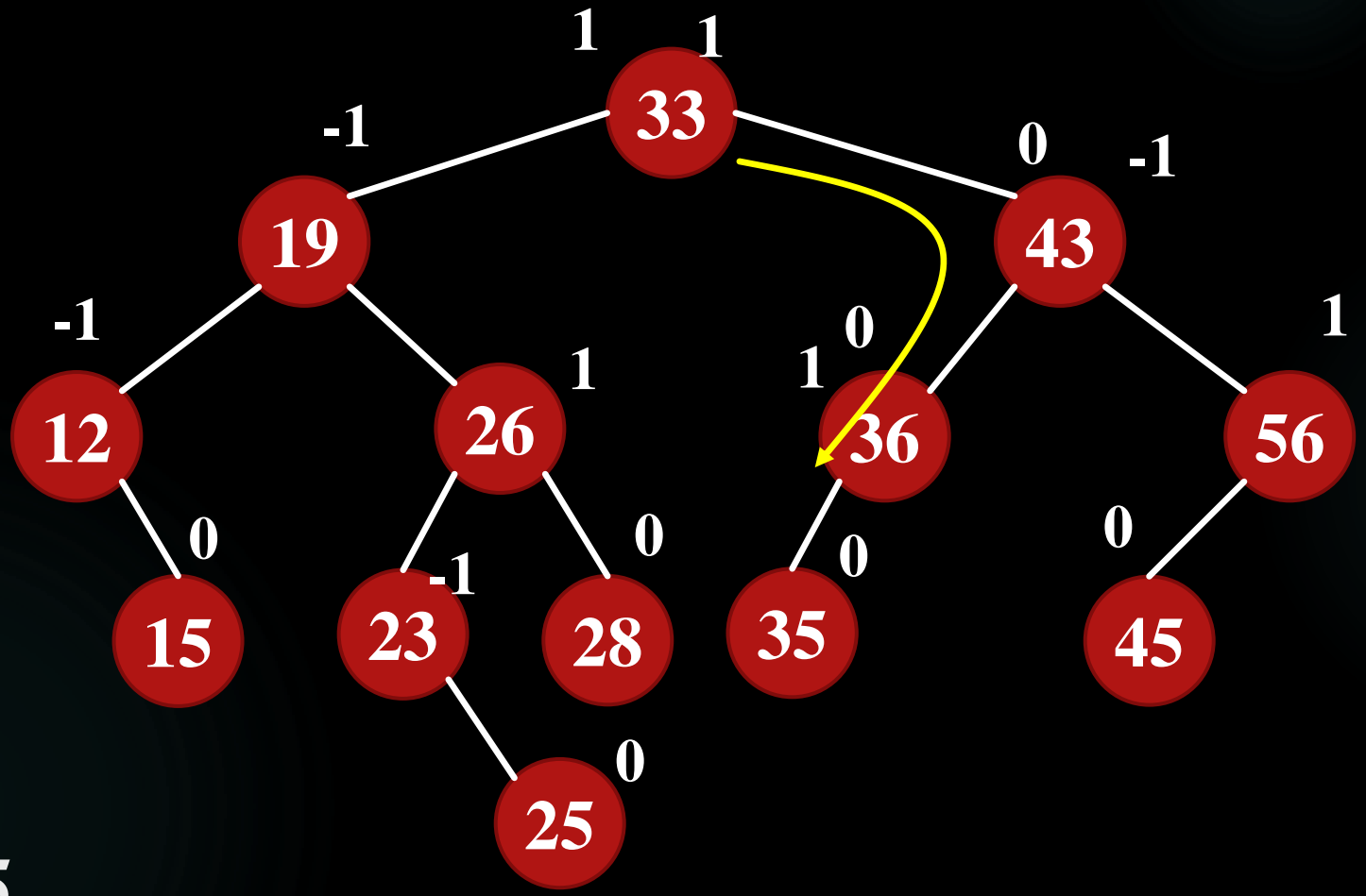
依然是AVL树，插入后树高加1

插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。



插入：35　　依然是AVL树，插入后树高不变

插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。

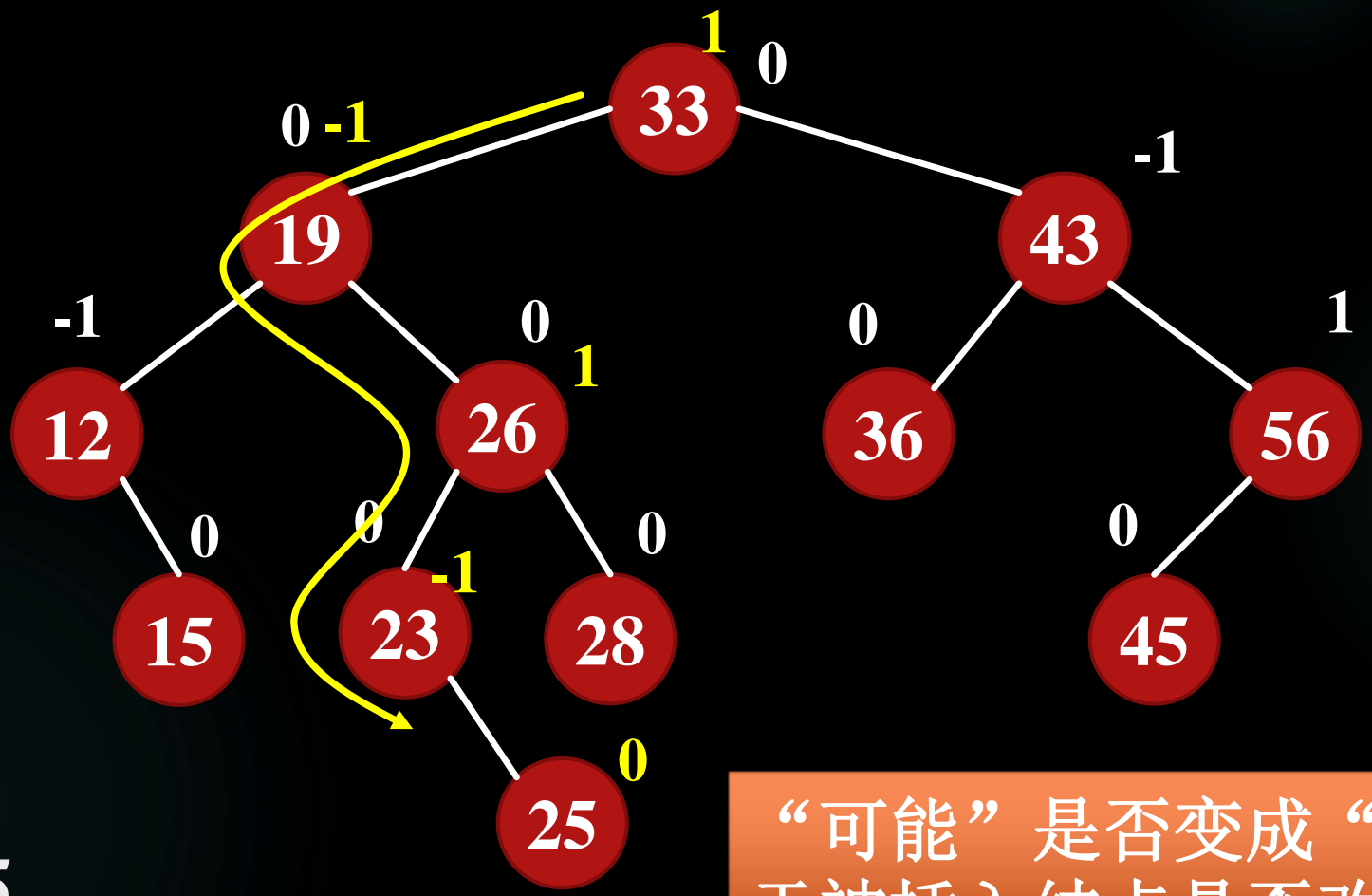


插入：35

根结点的平衡因子值不变

总结平衡因子改变规律

插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。

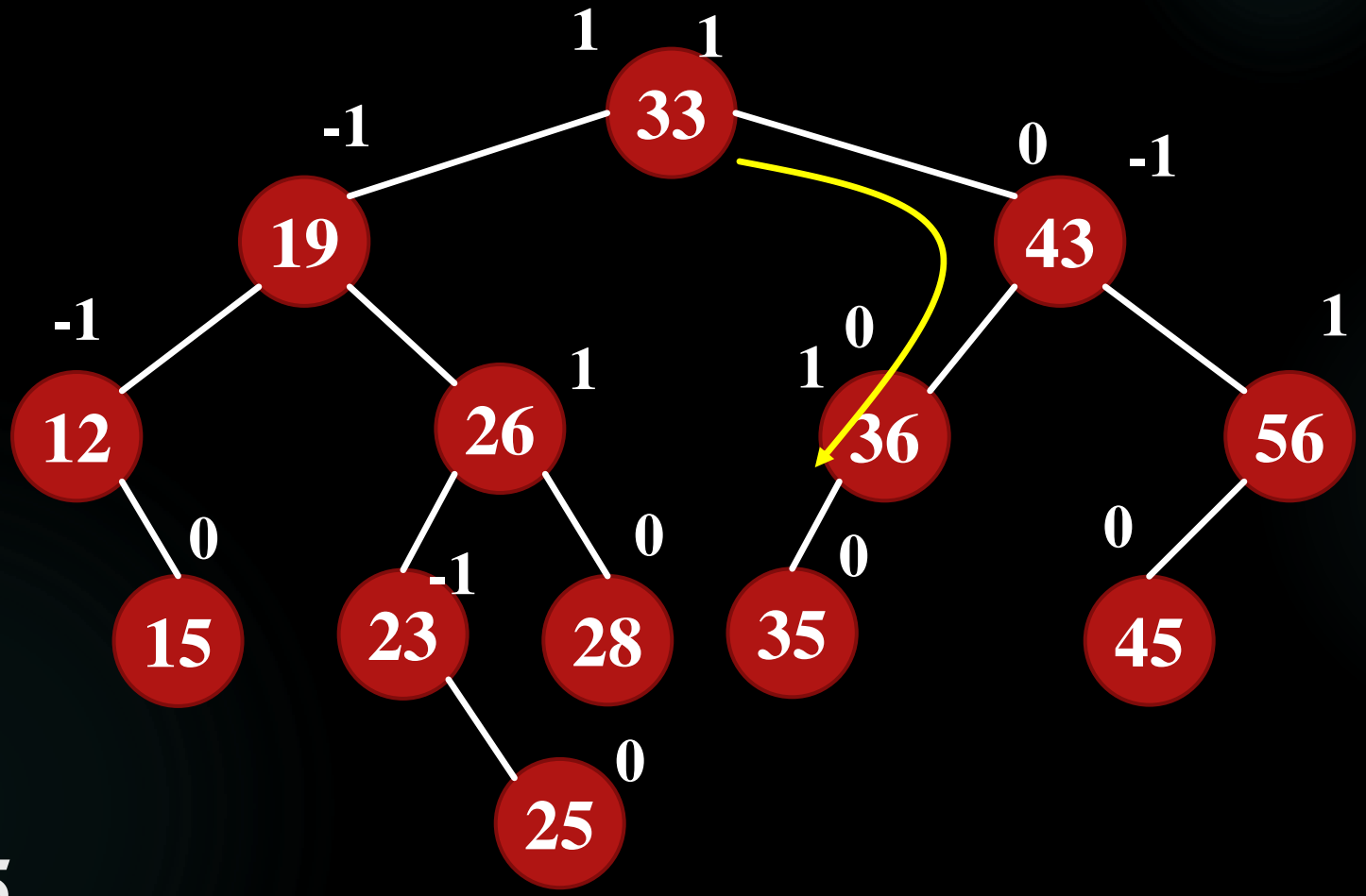


插入：25

“可能”是否变成“肯定”，取决于被插入结点是否改变子树高度

如果被插入的结点在这条路径上某个结点的左子树上，那么该结点的平衡因子可能加1，否则(插入在右子树上)可能减1。

插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。

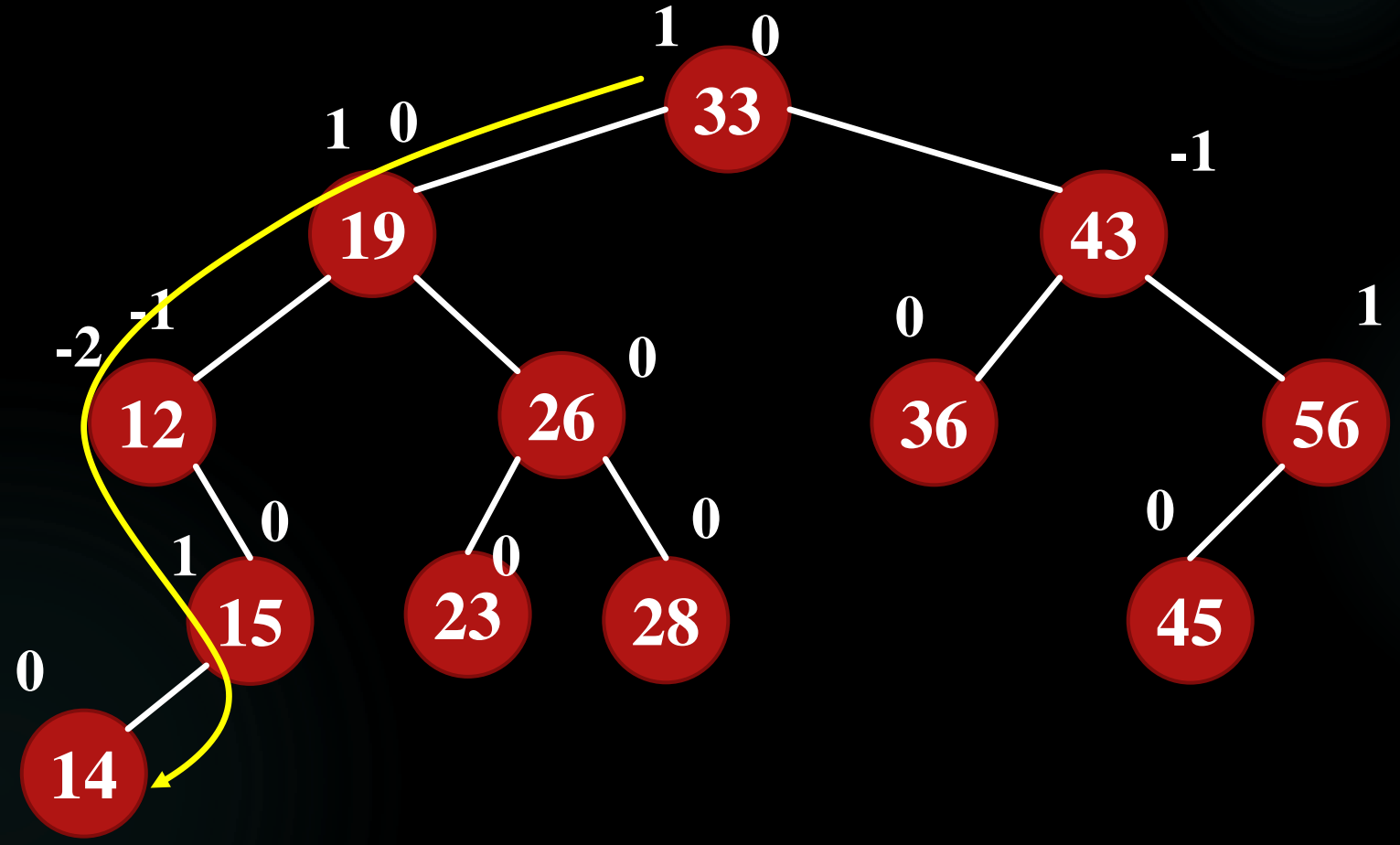


插入：35

如果被插入的结点在这条路径上某个结点的左子树上，那么该结点的平衡因子可能加1，否则(插入在右子树上) 可能减1。

插入后平衡树不再平衡的情况

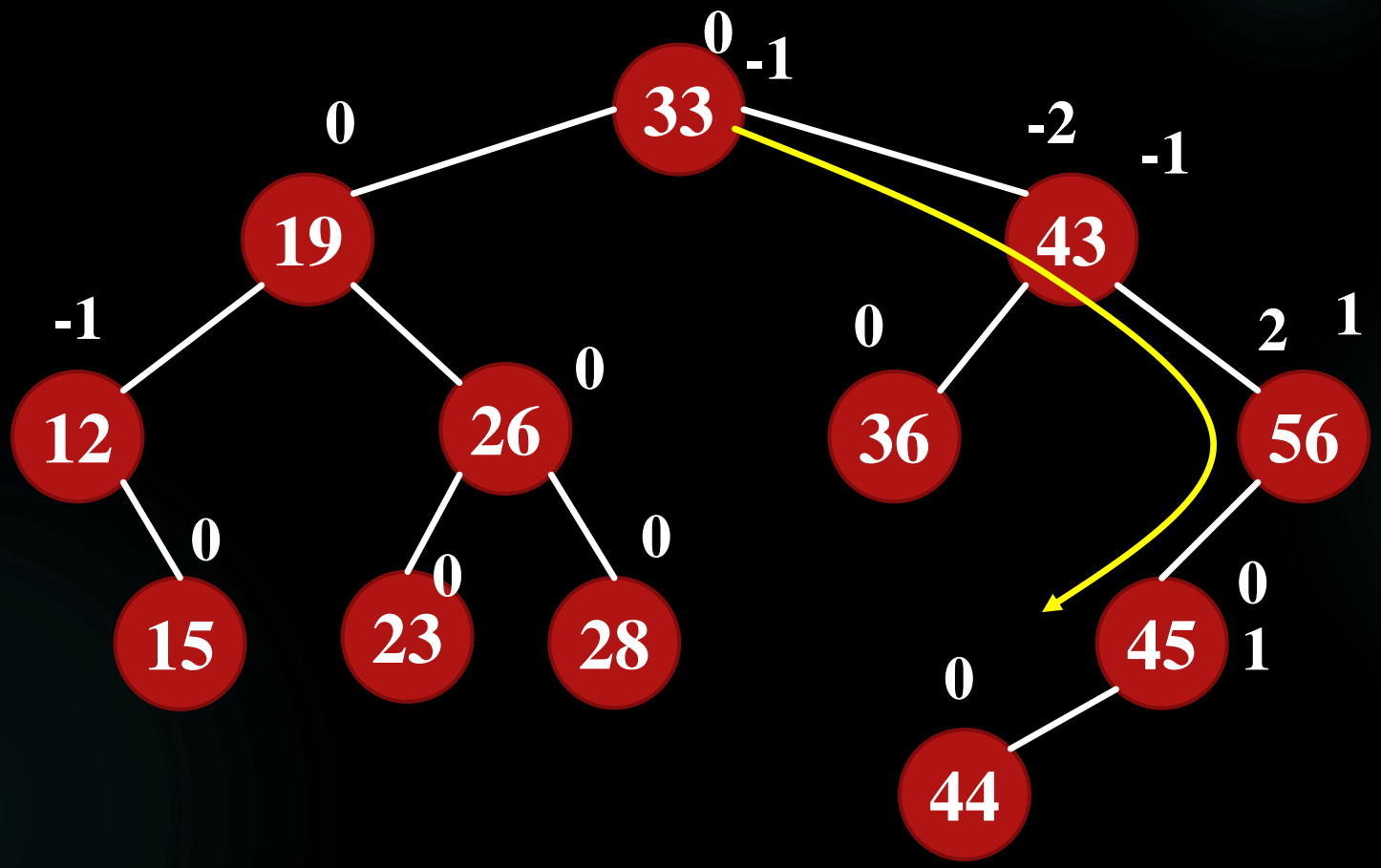
插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。



不再是二叉平衡树

插入：14

插入后可能影响从根到插入位置的路径上所有结点的平衡因子的值。



不再是二叉平衡树

插入：44

4个元素的插入分别代表4种不同情况：

（1）插入25 树仍然是二叉平衡树，树高度加1

（2）插入35 树仍然是二叉平衡树，高度不变

（3）插入14 树不再是二叉平衡树

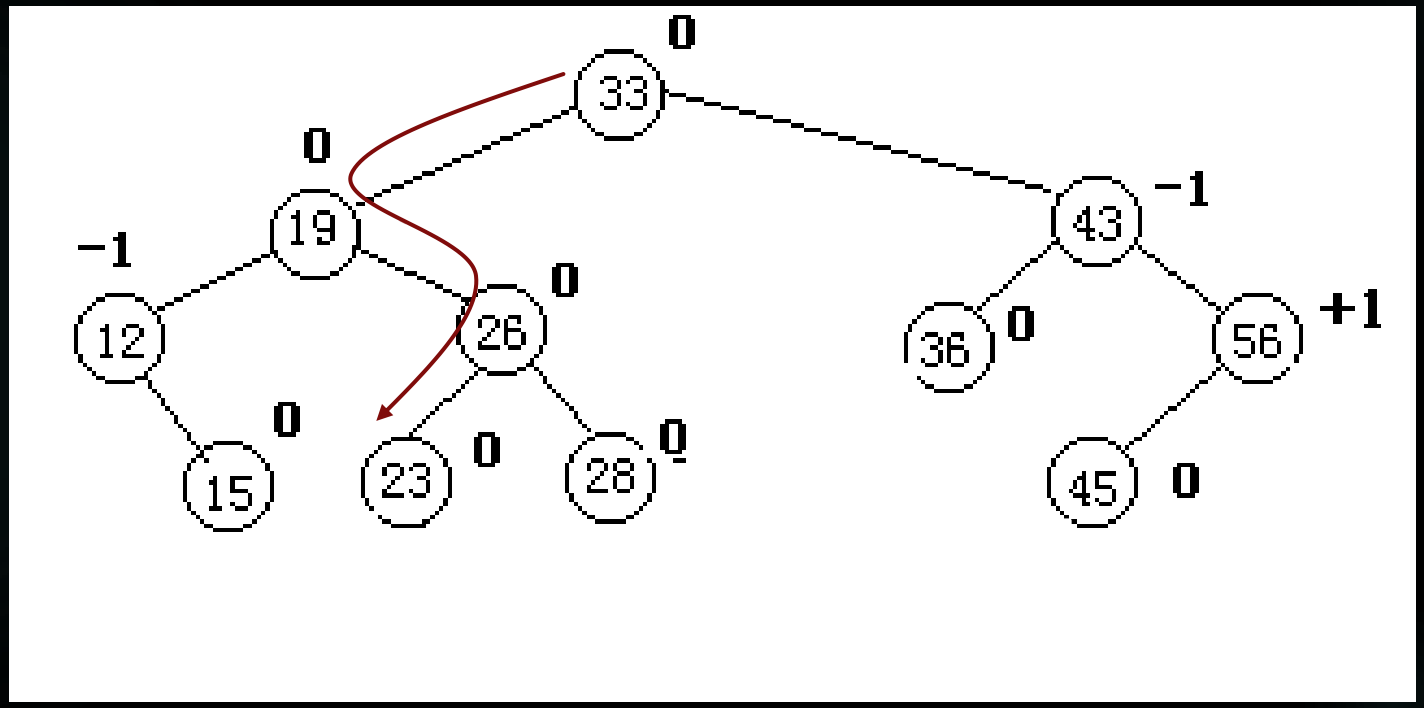
（4）插入44 树不再是二叉平衡树

一般情况，插入新结点后会影响从根结点到新结点的路径上所有结点的平衡因子。插入在某结点的左子树上，该结点的平衡因子可能+1，插入在某结点的右子树上，该结点的平衡因子可能-1，

插入结点后必须进行一系列处理以保持平衡性

重新平衡

在根到新插入结点的路径上，**所有**结点的平衡因子都为0，结点插入后依然是平衡二叉树



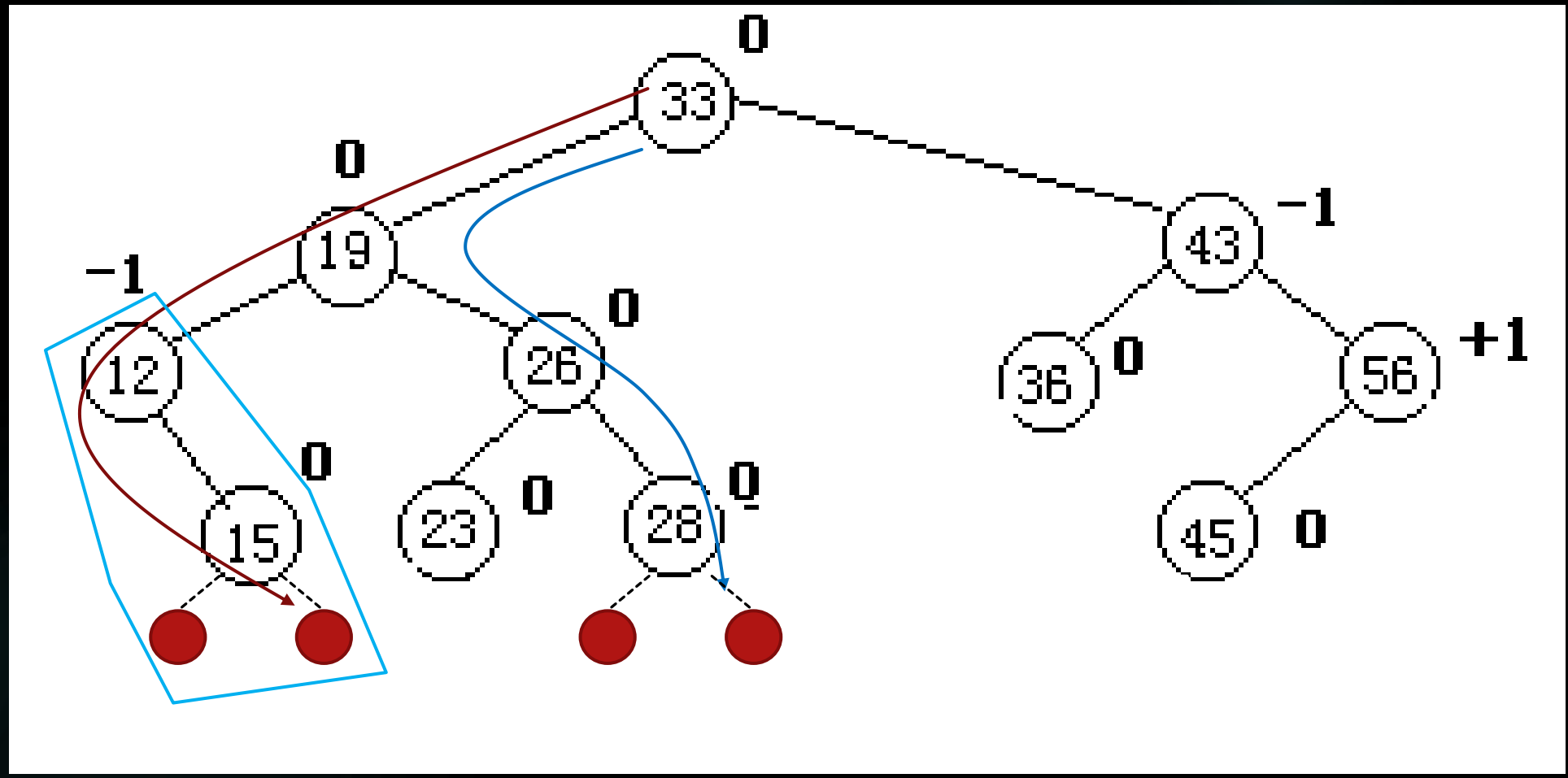
重新平衡

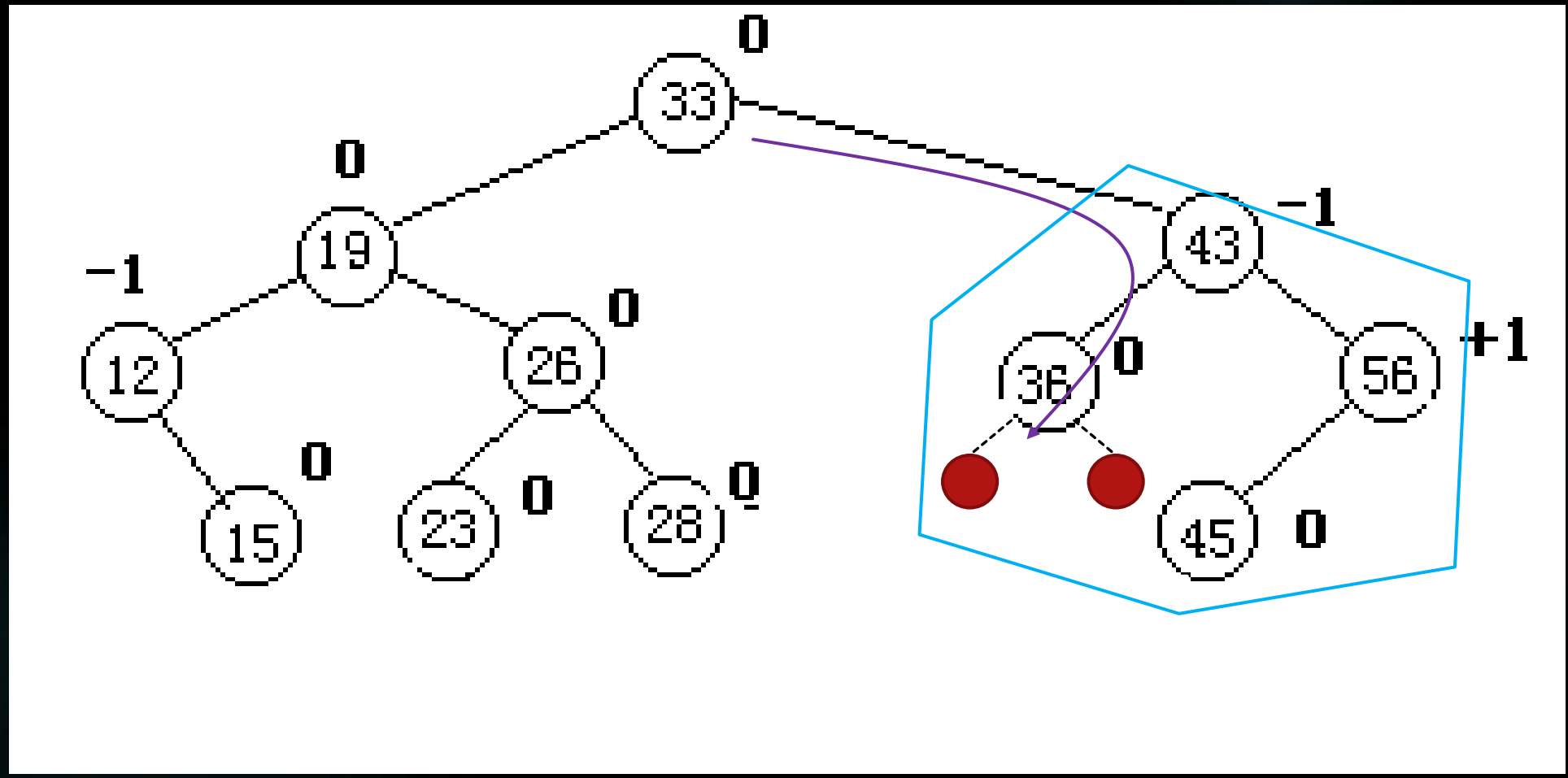
最近不平衡祖先：**从根结点到插入节点的路径上，离插入结点最近的平衡因子不为0的结点（没有插入前）**

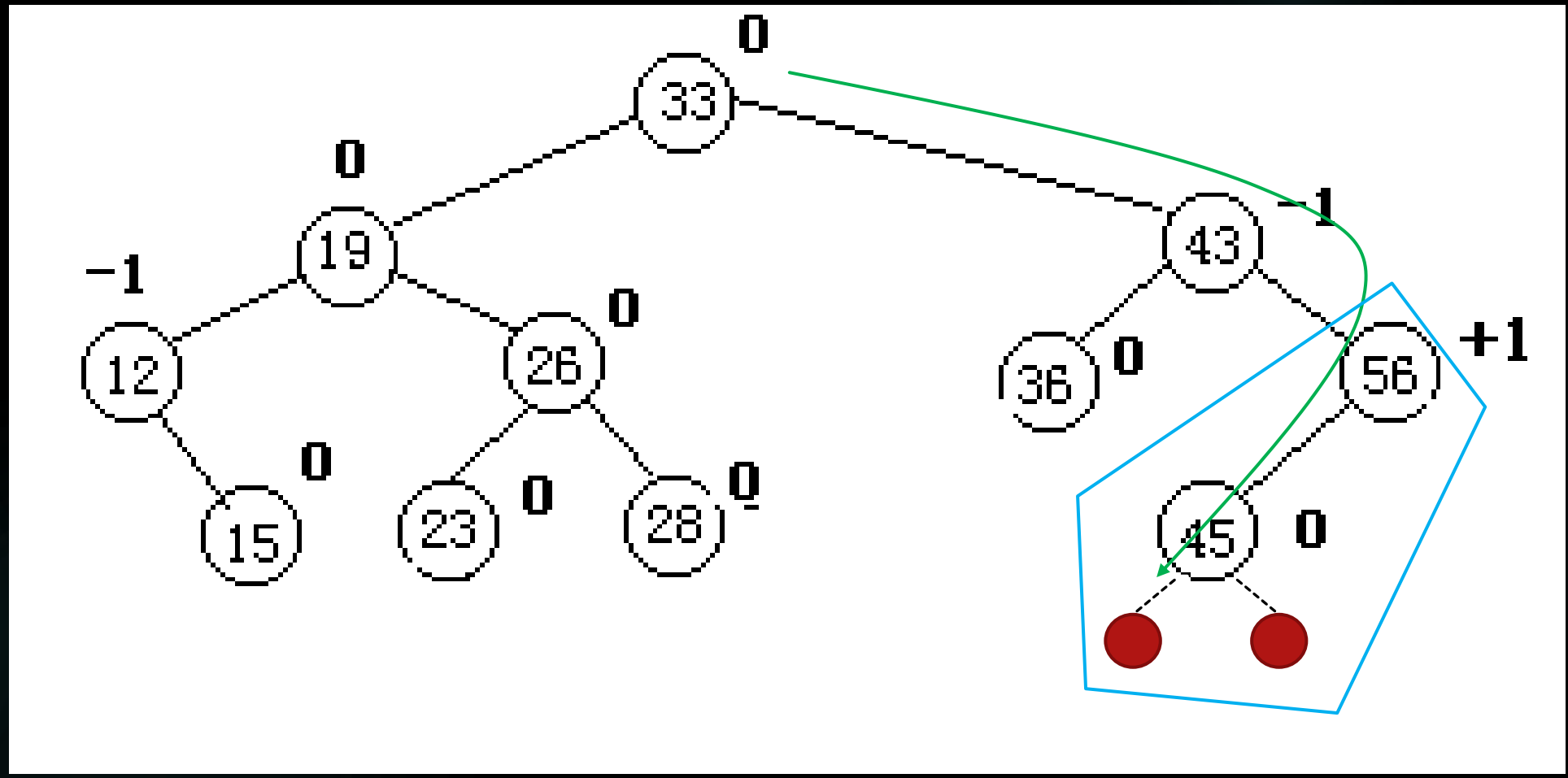
最小不平衡子树：以最近不平衡祖先为根的子树

我们将在最小不平衡子树的范围内进行调整，使得整个树达到平衡

为方便起见，用s表示最小不平衡子树


0
33
0
19
-1
43
-1
12
0
26
0
36
0
56
+1
0
15
0
23
0
28
0
45


0
33
0
19
-1
43
-1
12
26
0
36
0
56
+1
0
0
0
15
23
28
45
0


0
33
0
19
-1
43
-1
12
0
26
0
36
0
56
+1
0
15
0
23
0
28
0
45
0

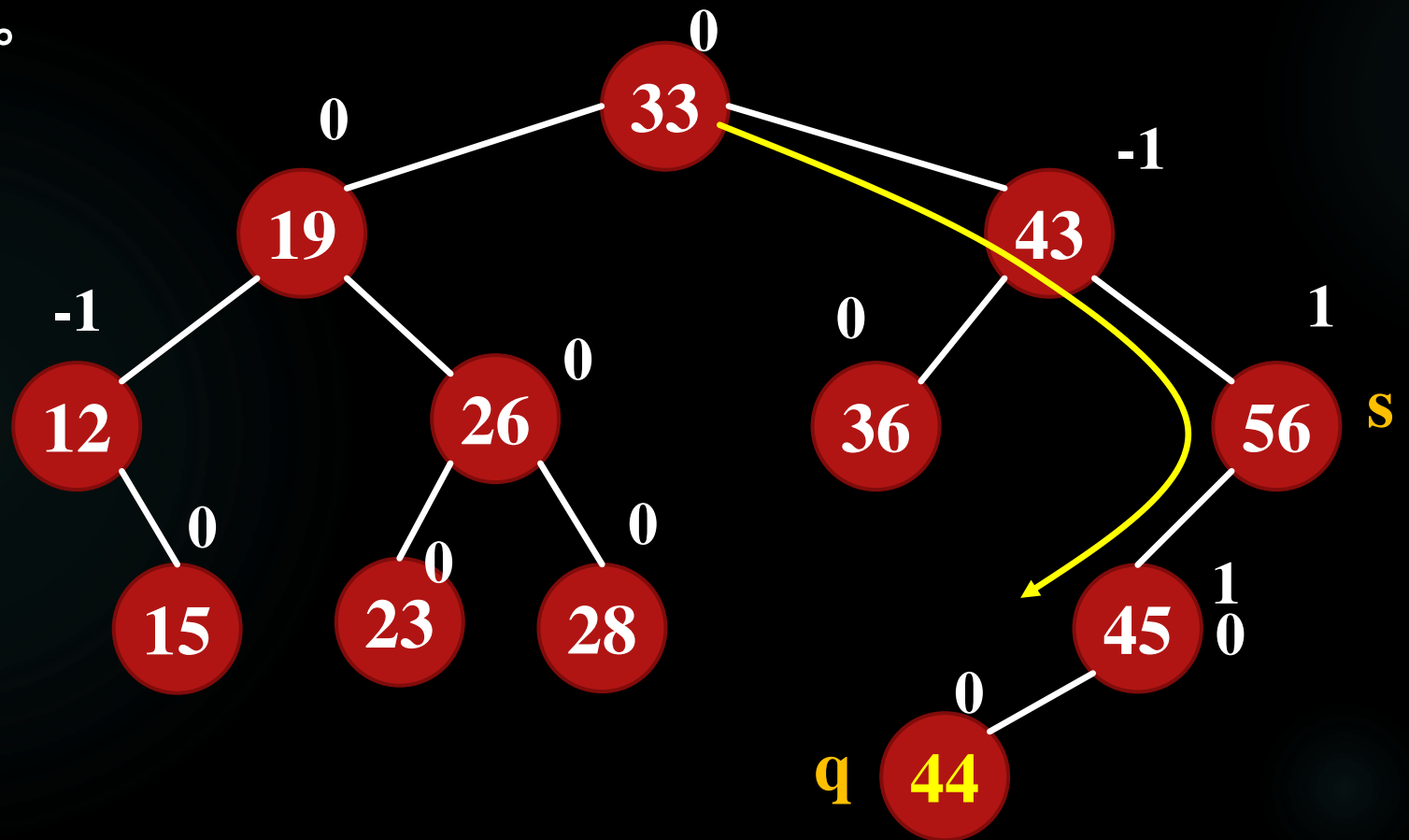
重新平衡-为讨论插入算法，作以下准备工作

（1）新结点为q，q的最近不平衡祖先为s

（2）q按照二叉搜索树的插入方式插入到s的左子树下

（3）从结点s到新结点q的路径上所有结点（不含s）的平衡因子值均已修正。

右子树情况下的分析是类似的

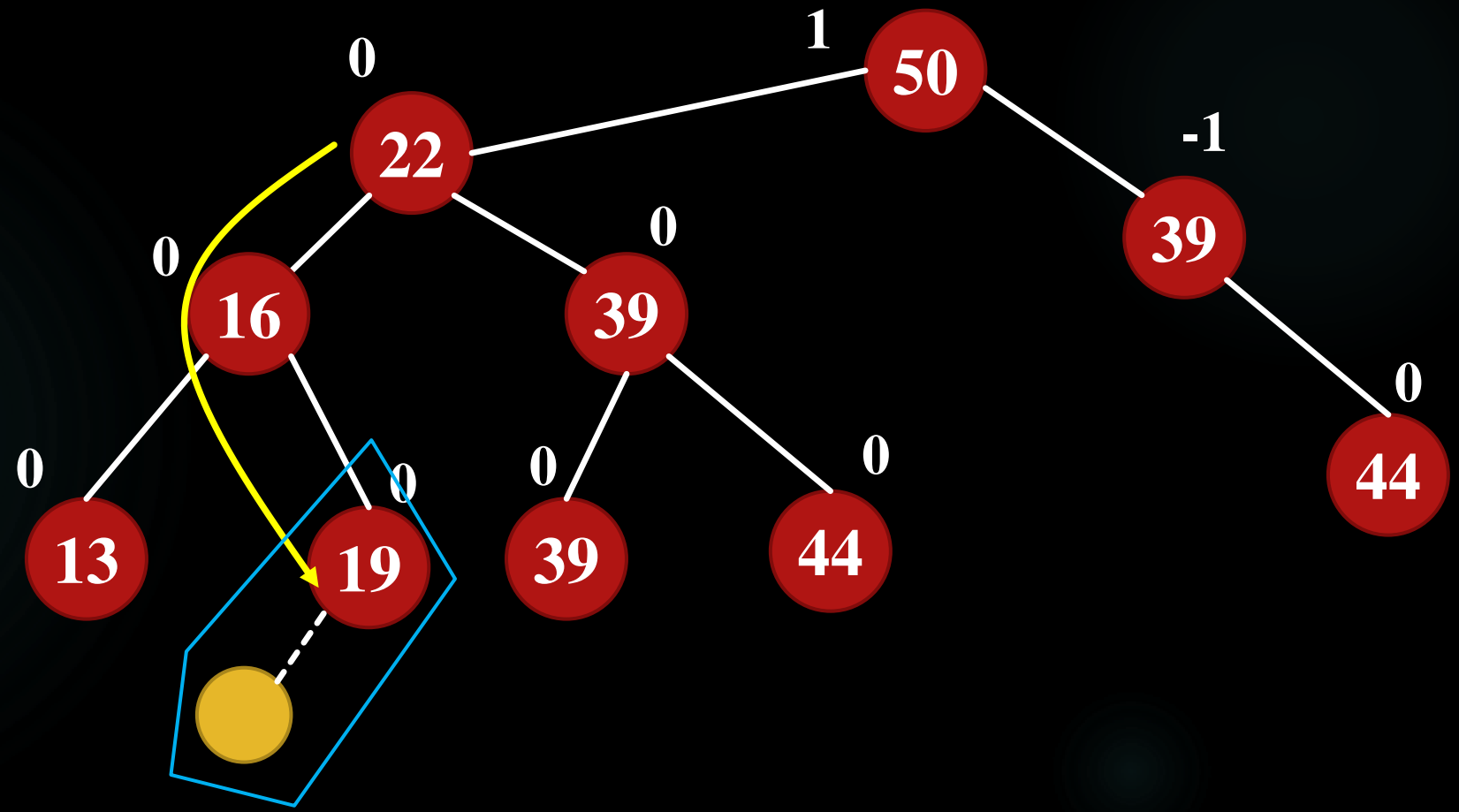


在插入q之前根据平衡因子找到s

修改平衡因子

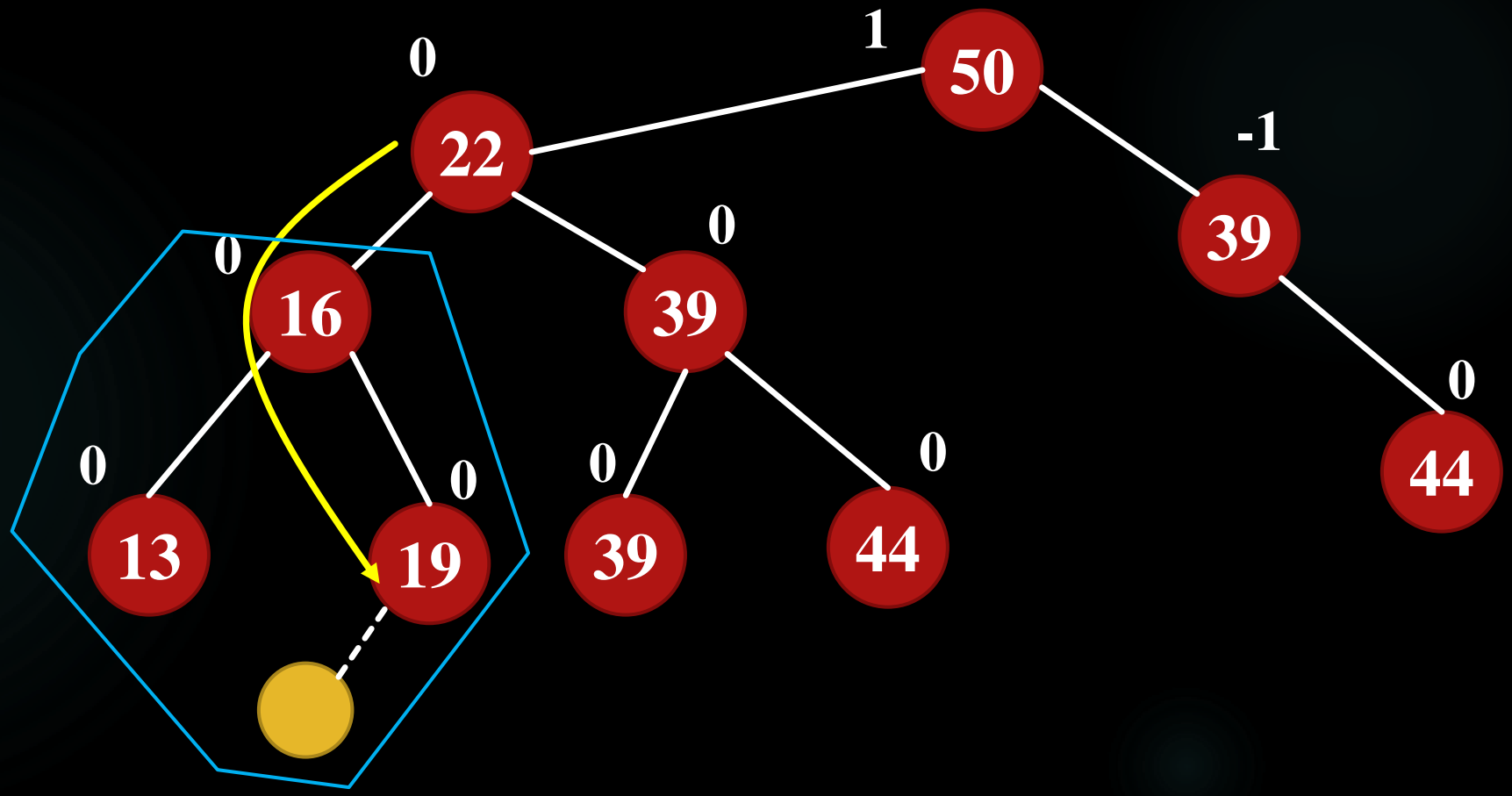
一个规律

如果一条路径上所有结点平衡因子为0，则在这条路径的最后插入一个结点，一定会使得这条路径上所有结点为根的子树的高度+1



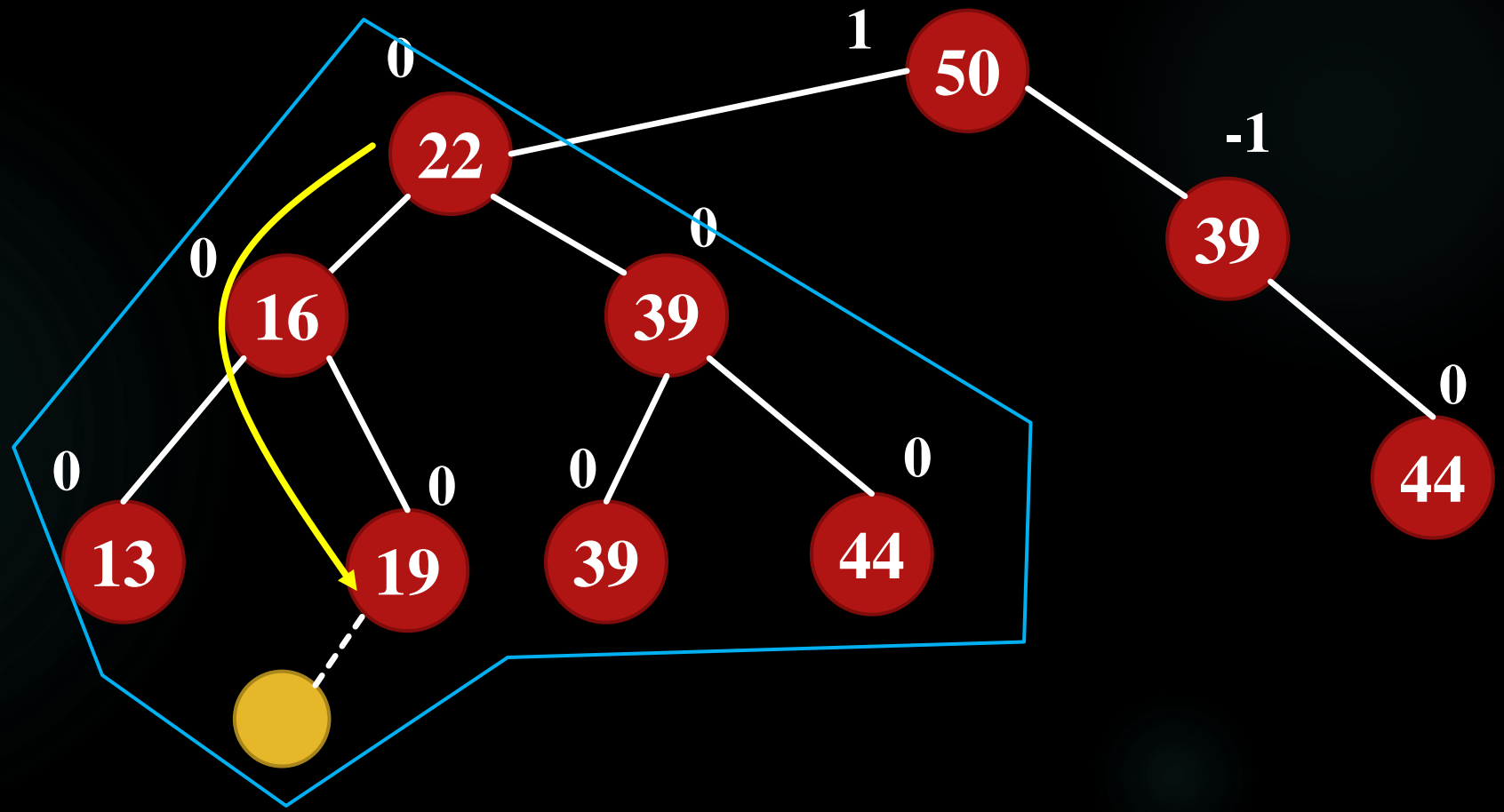
一个规律

如果一条路径上所有结点平衡因子为0，则在这条路径的最后插入一个结点，一定会使得这条路径上所有结点为根的子树的高度+1



一个规律

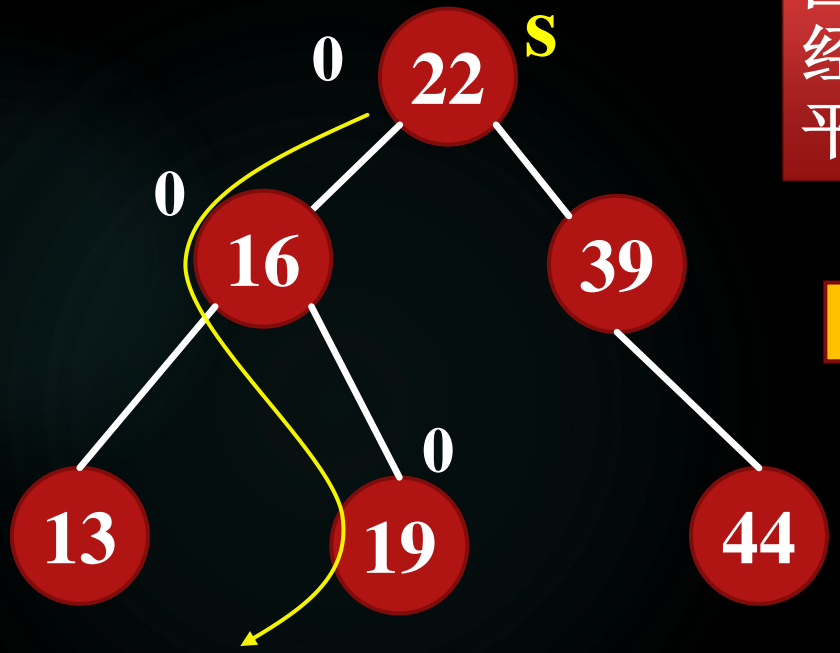
如果一条路径上所有结点平衡因子为0，则在这条路径的最后插入一个结点，一定会使得这条路径上所有结点为根的子树的高度+1



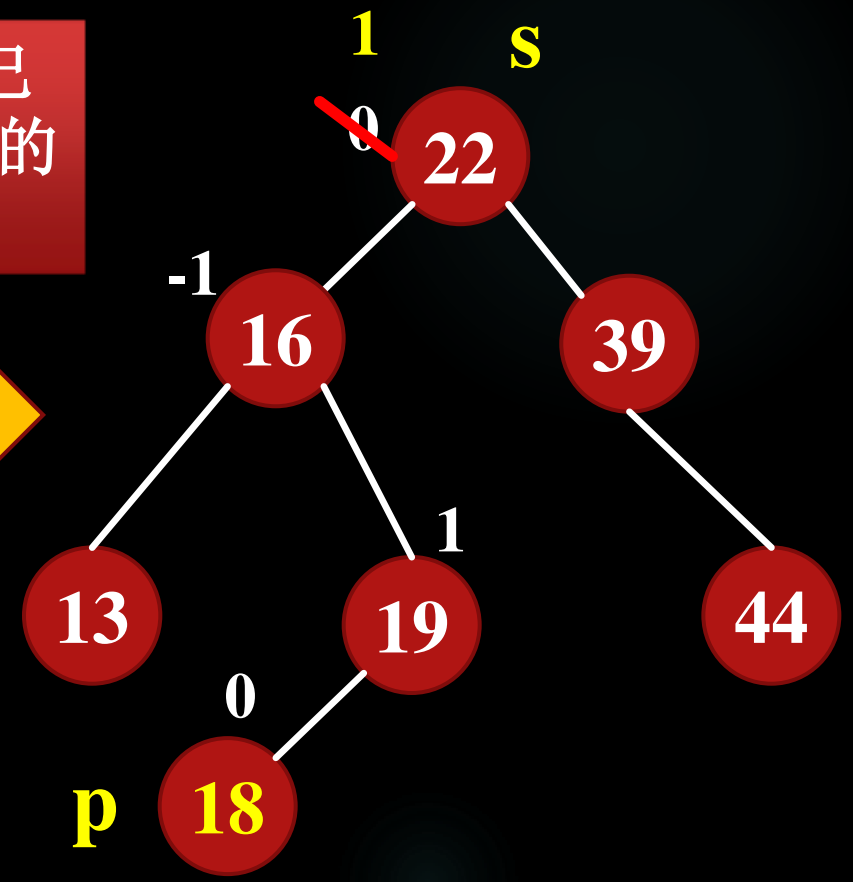
基于上述假设与准备，二叉平衡树的插入算法可分三种情况讨论

情况一（s的平衡因子等于0）

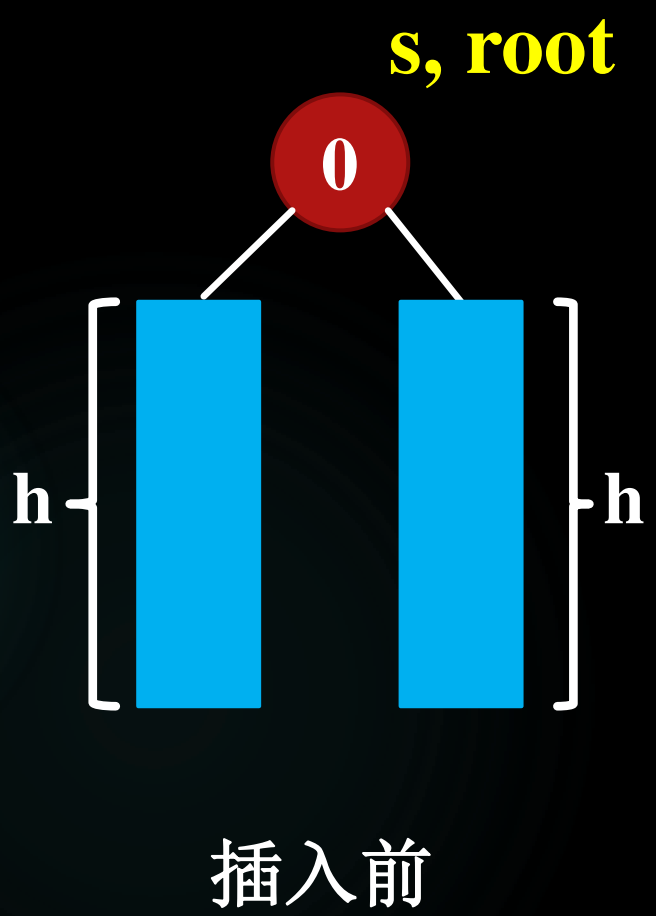
- s肯定为根，且从根结点到新结点q的插入位置的路径上，所有结点的平衡因子值均为0
- 插入q后，只需将根结点的平衡因子改为+1，并且AVL树的高度加1，插入操作完成。



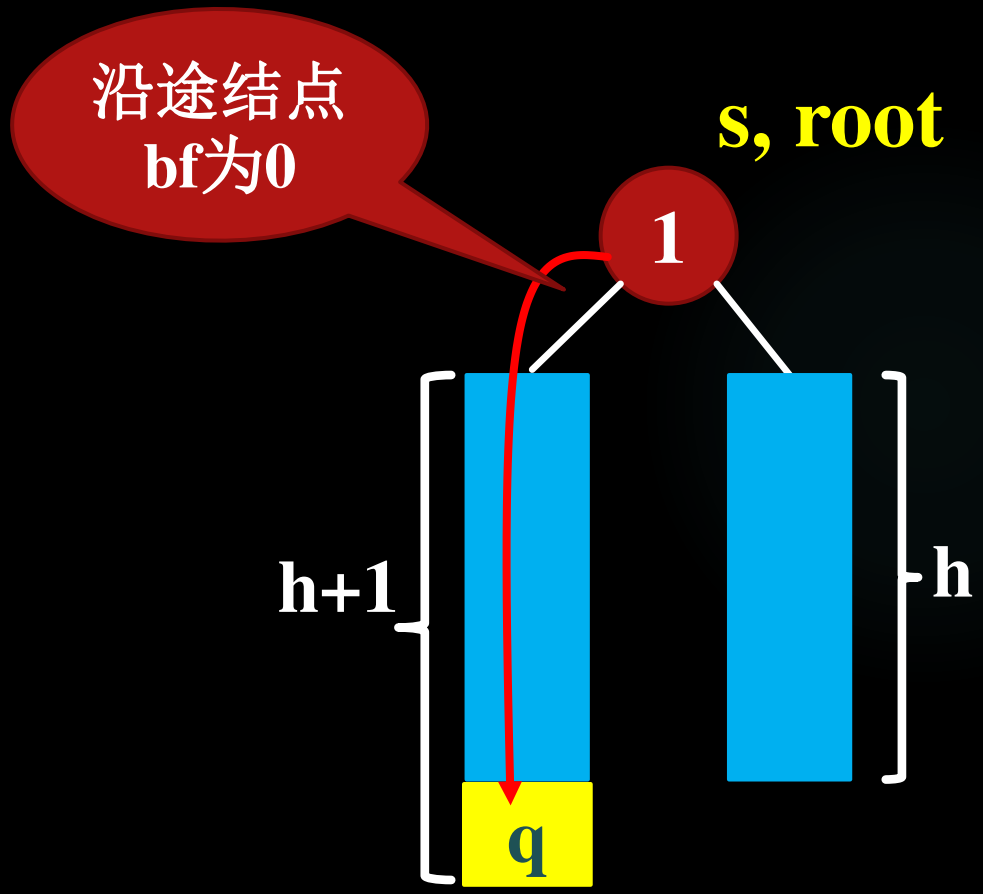
因为在准备工作时，已经将除s之外其他结点的平衡因子修改完成



情况一（s的平衡因子等于0）



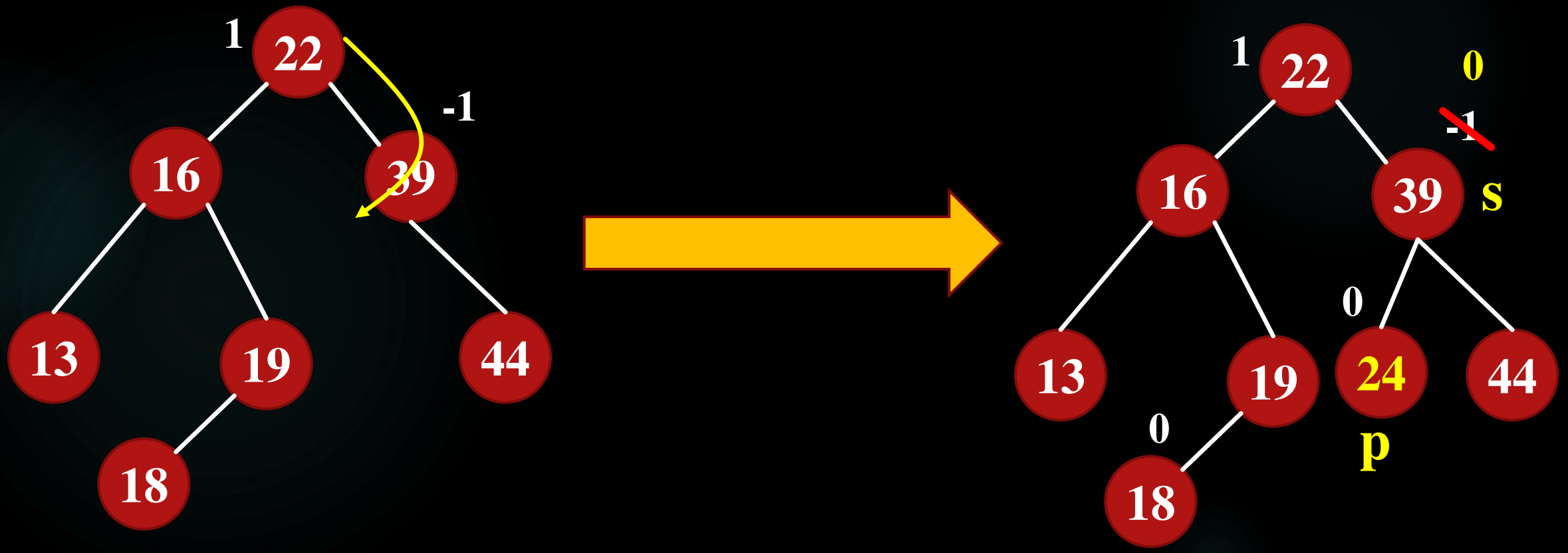
插入前



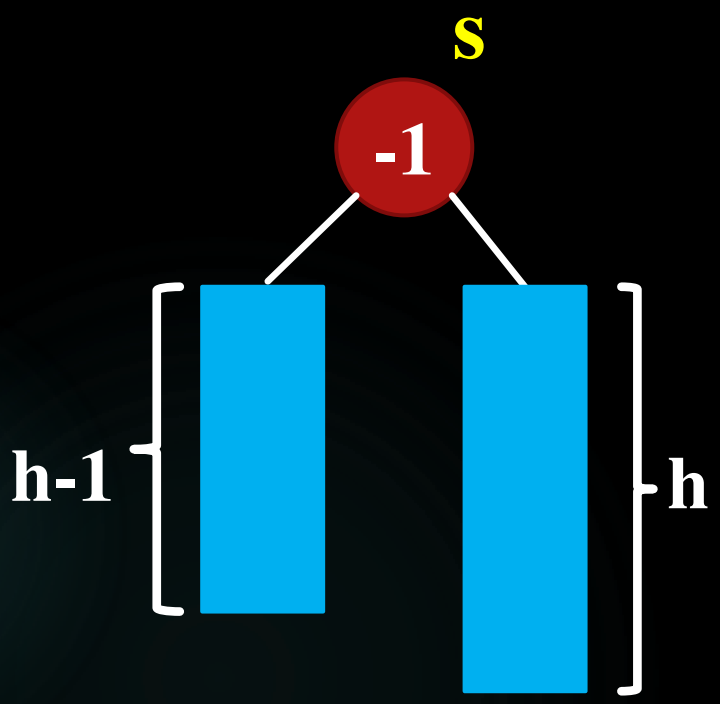
插入后

情况二（s的平衡因子等于-1）

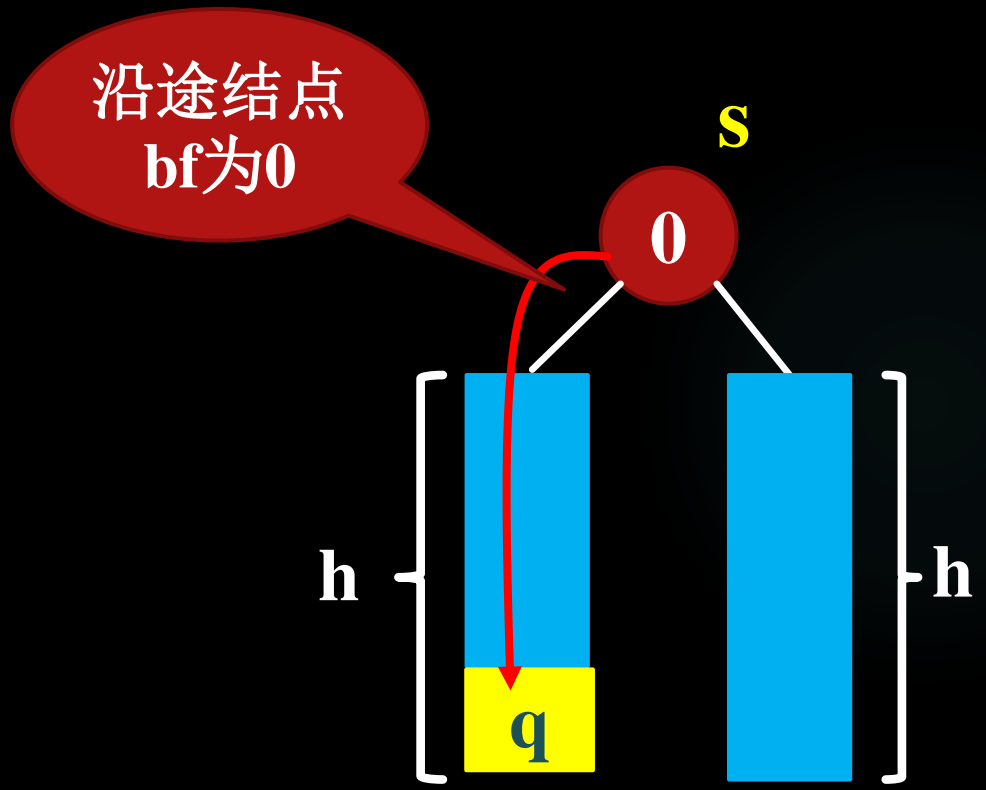
s的平衡因子为-1，并假定q插在s的左子树上，则新结点q插在结点s较矮的子树上，则插入后只需令s的平衡因子为0，插入算法终止。



情况二（s的平衡因子等于-1）



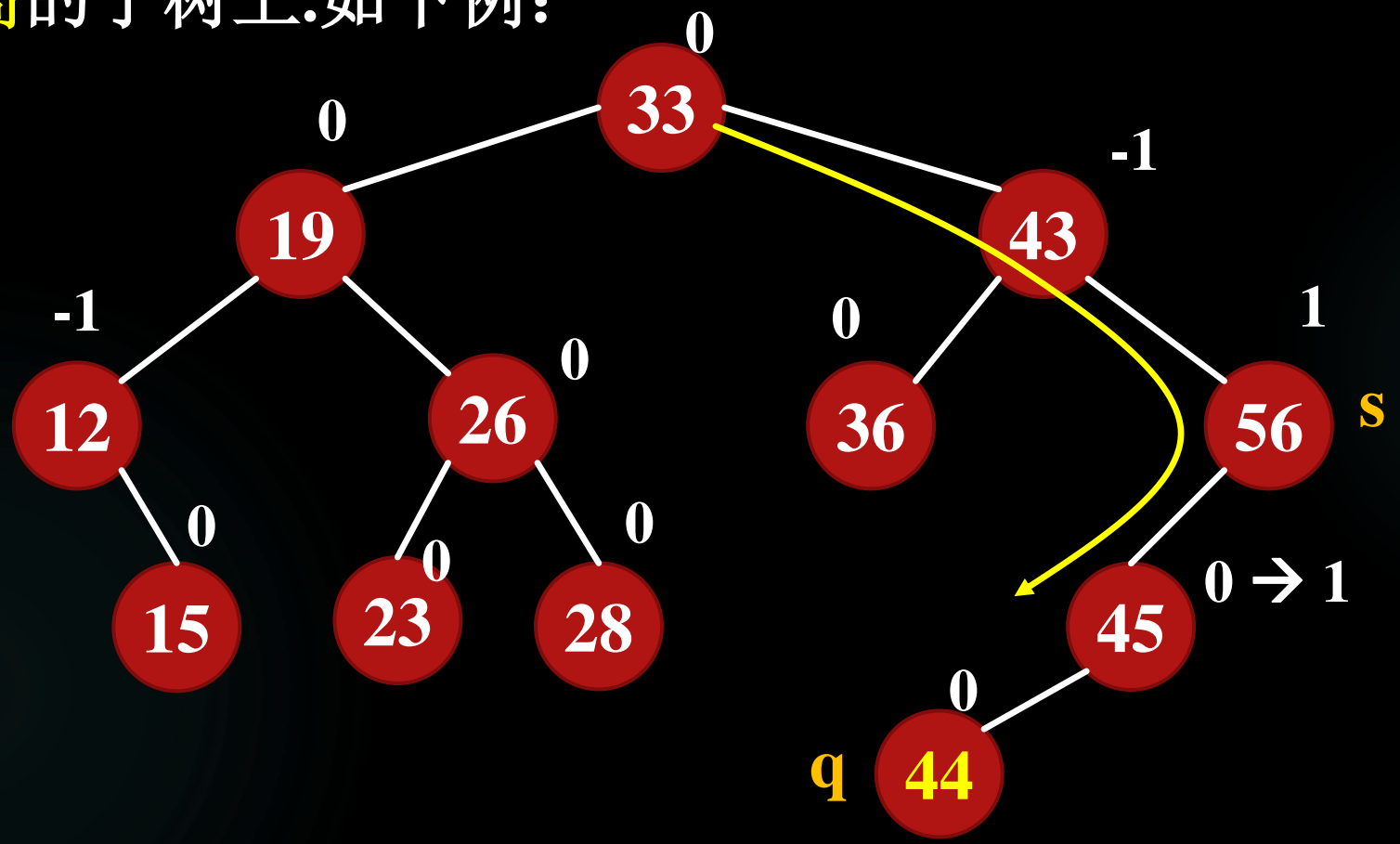
插入前



插入后

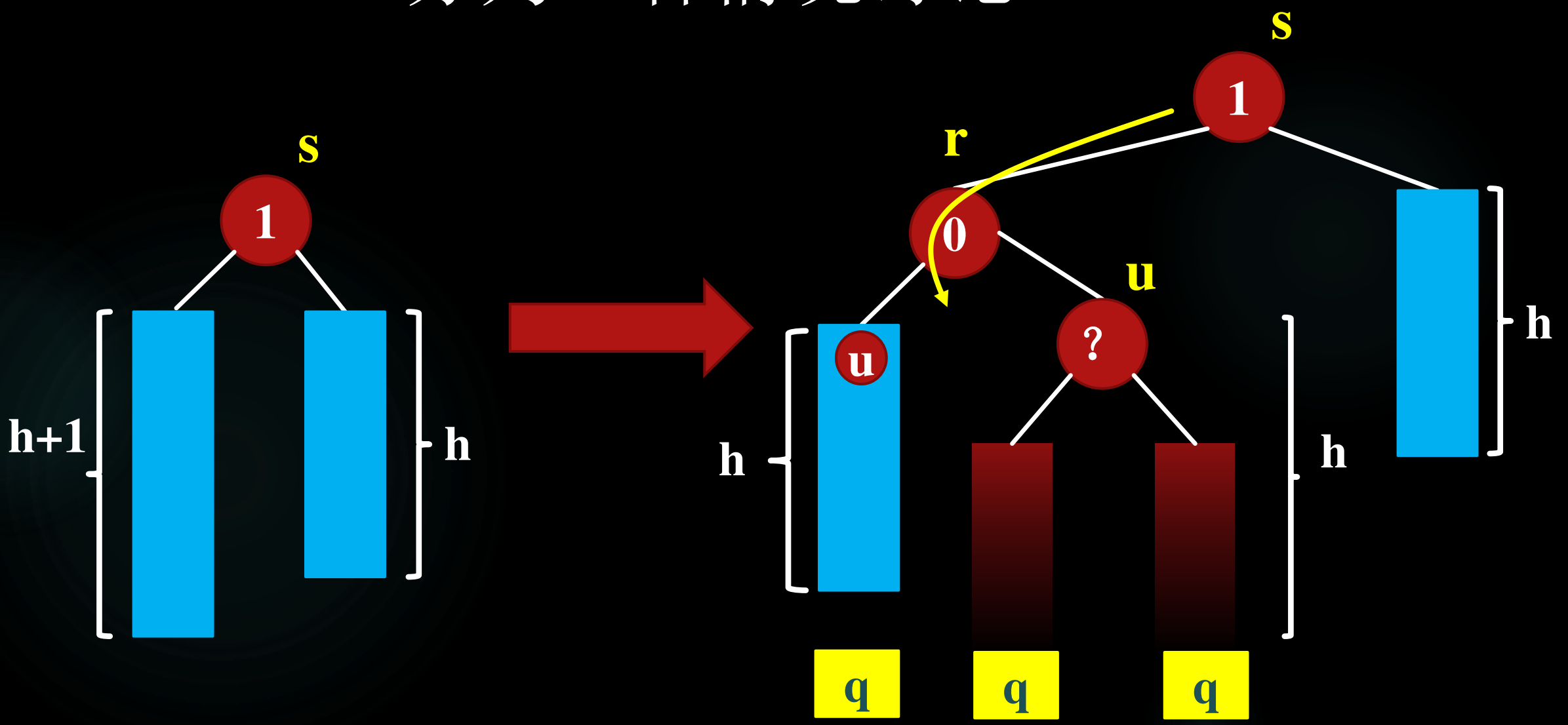
情况三：（s的平衡因子等于1）

因为s的左树比右树高，且我们已经假定新结点插在左树上，因此新结点q插在较高的子树上.如下例：

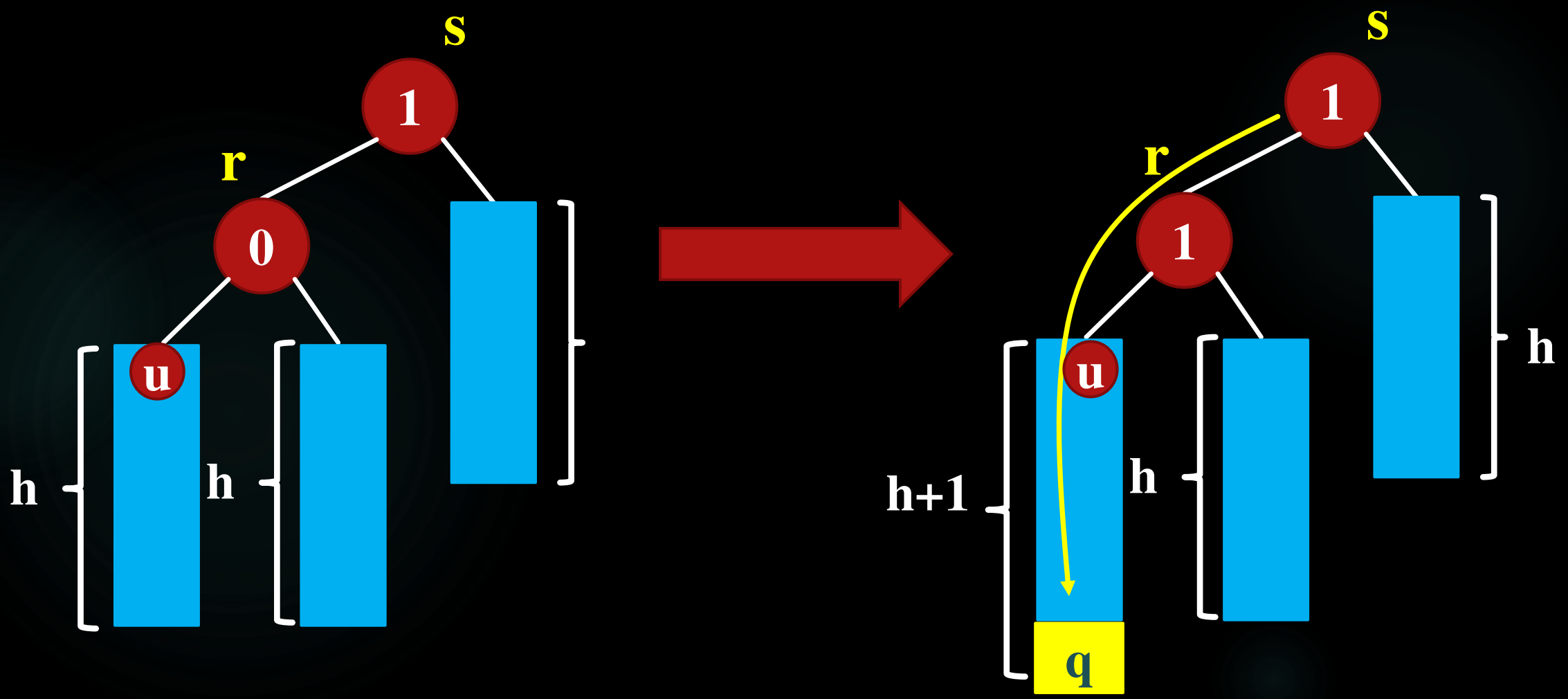


如何恢复平衡性?

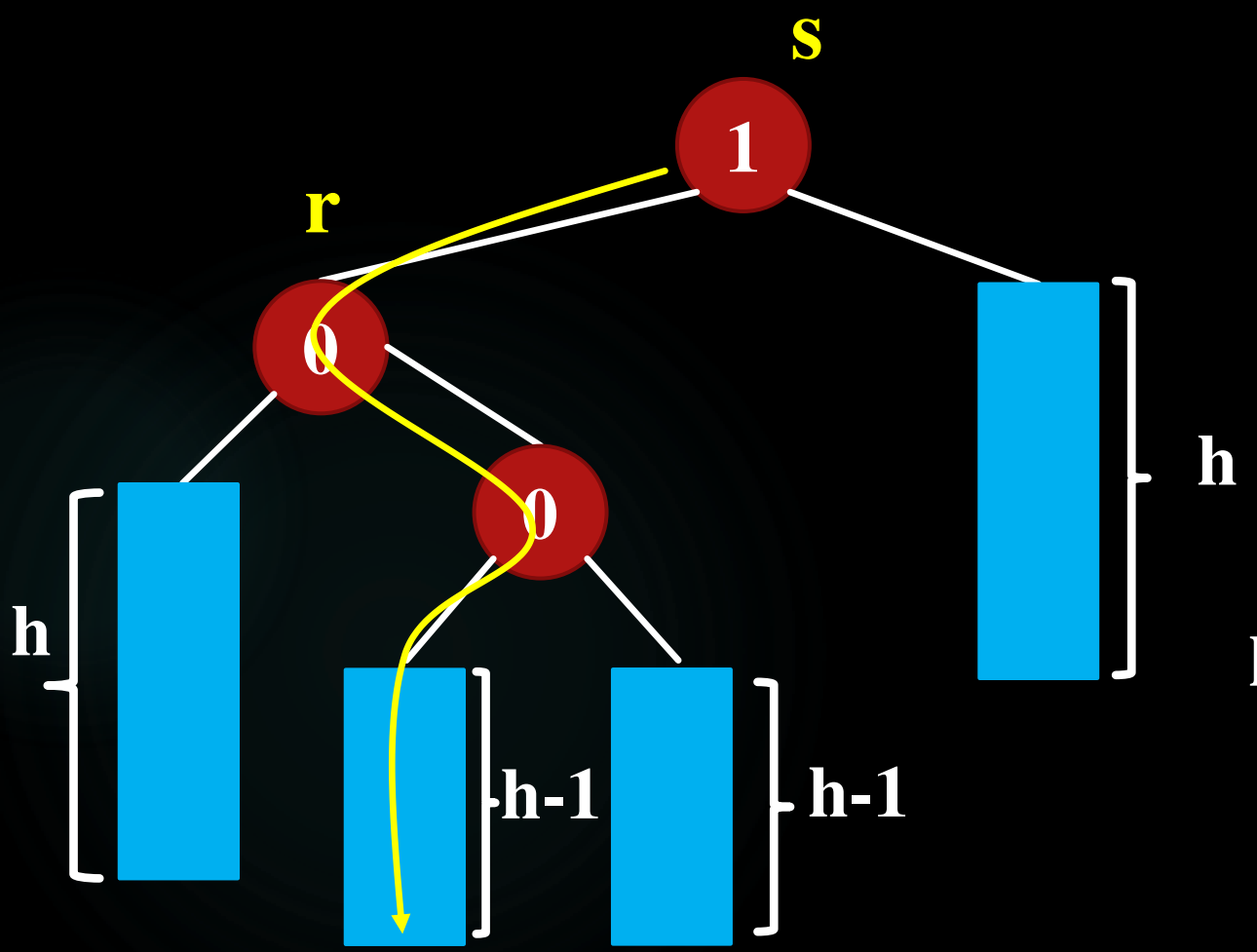
分为三种情况讨论

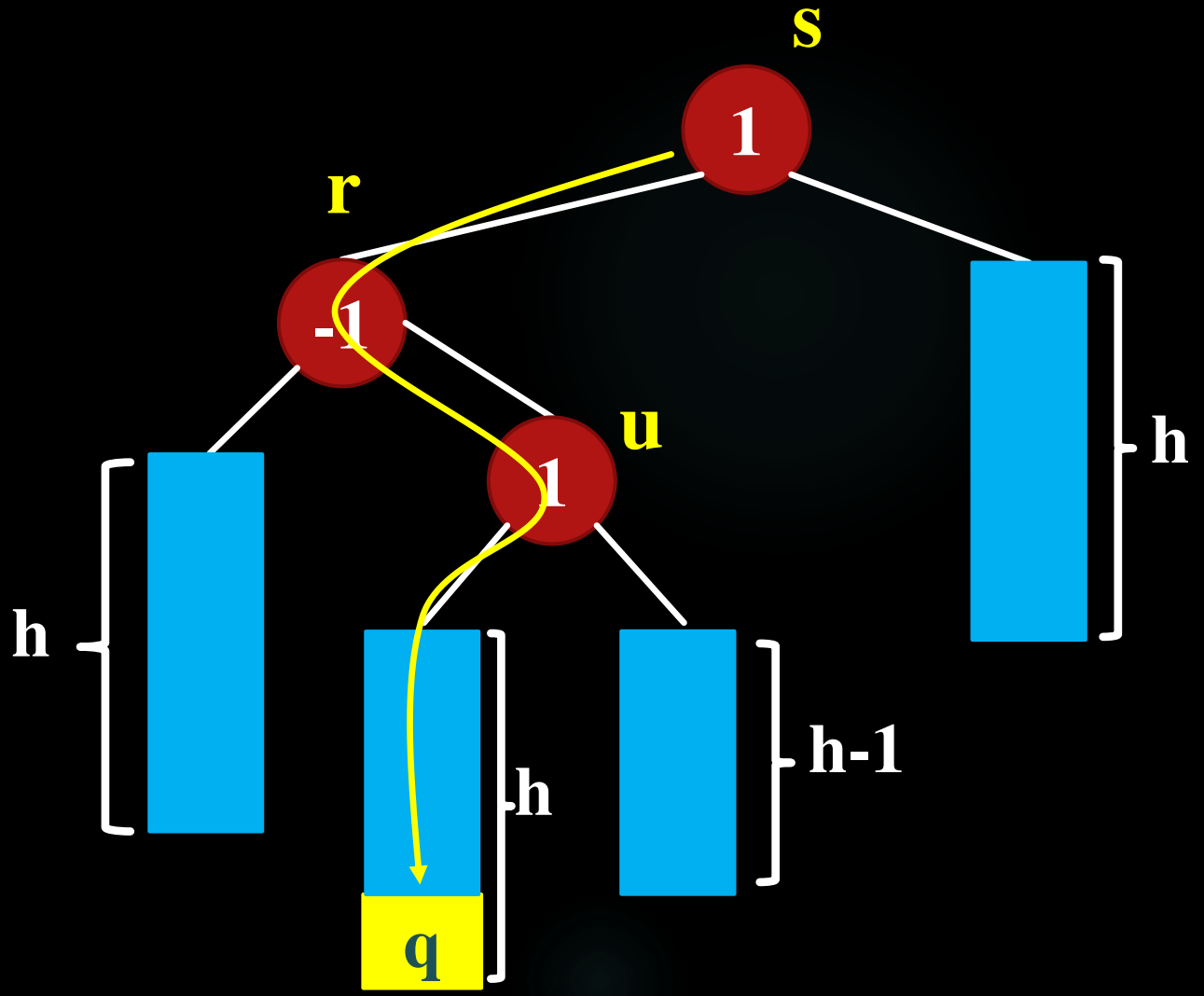


情况三（1）：q添加到r的左子树上

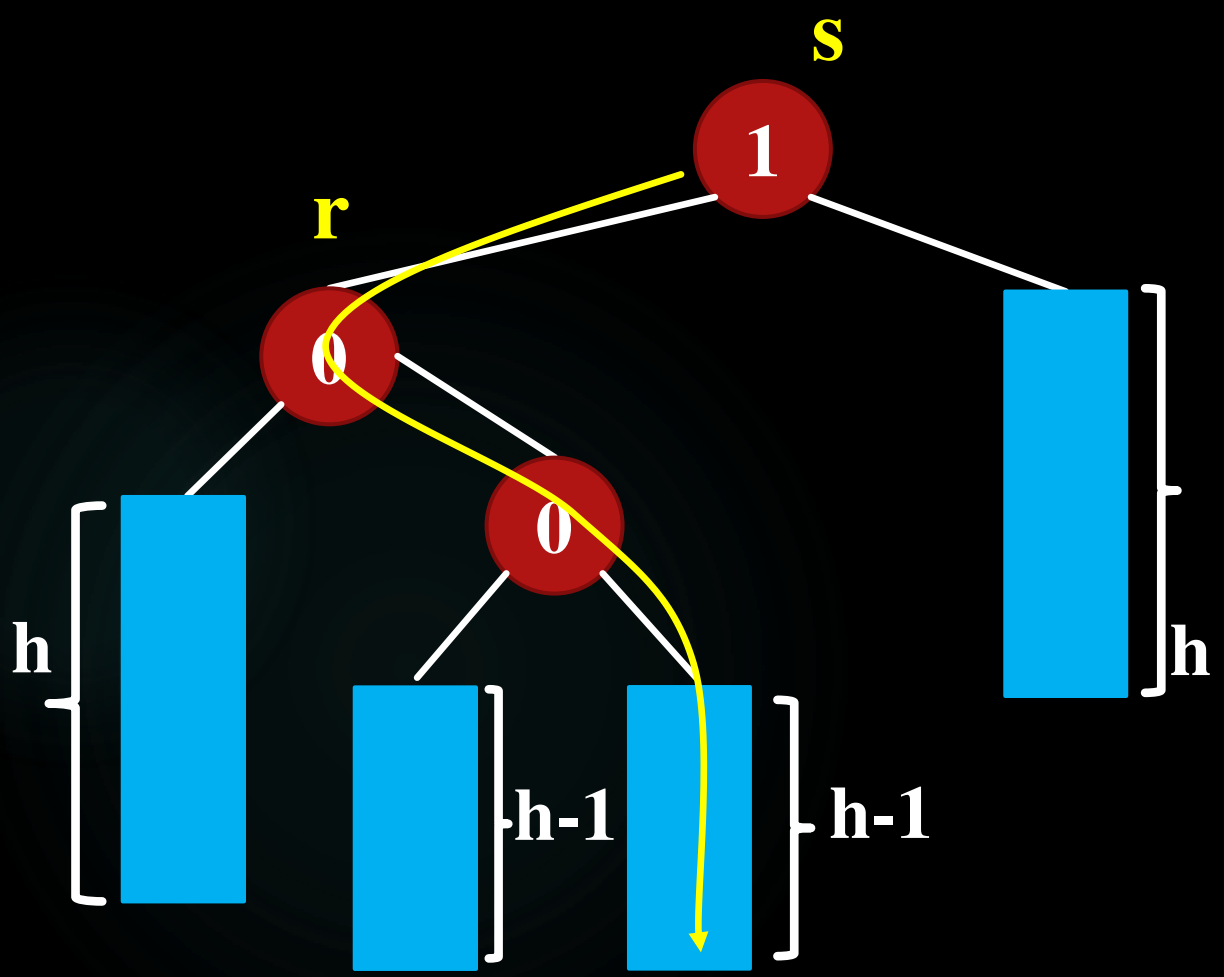


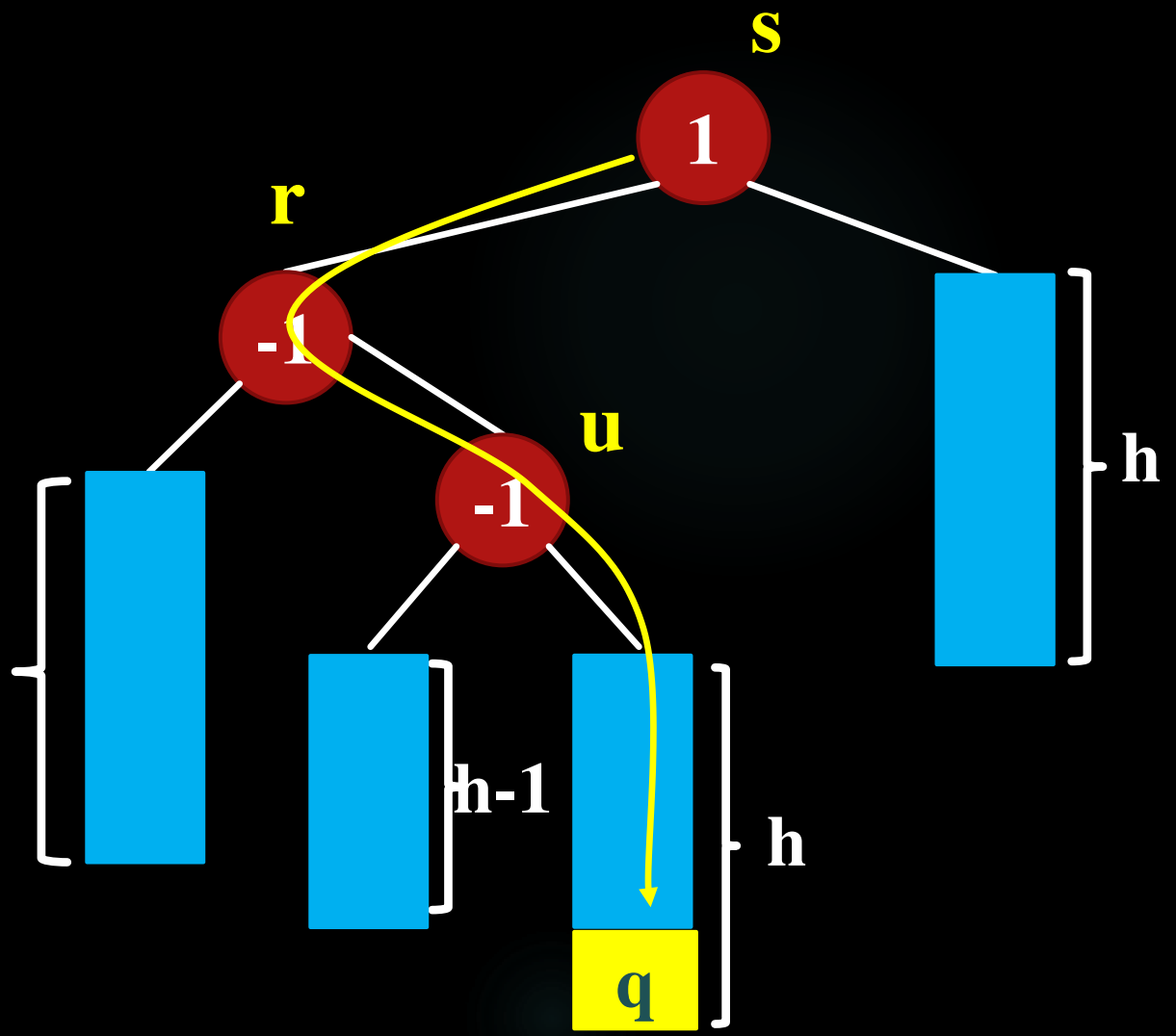
情况三（2）：q添加到r的右子树的左子树上





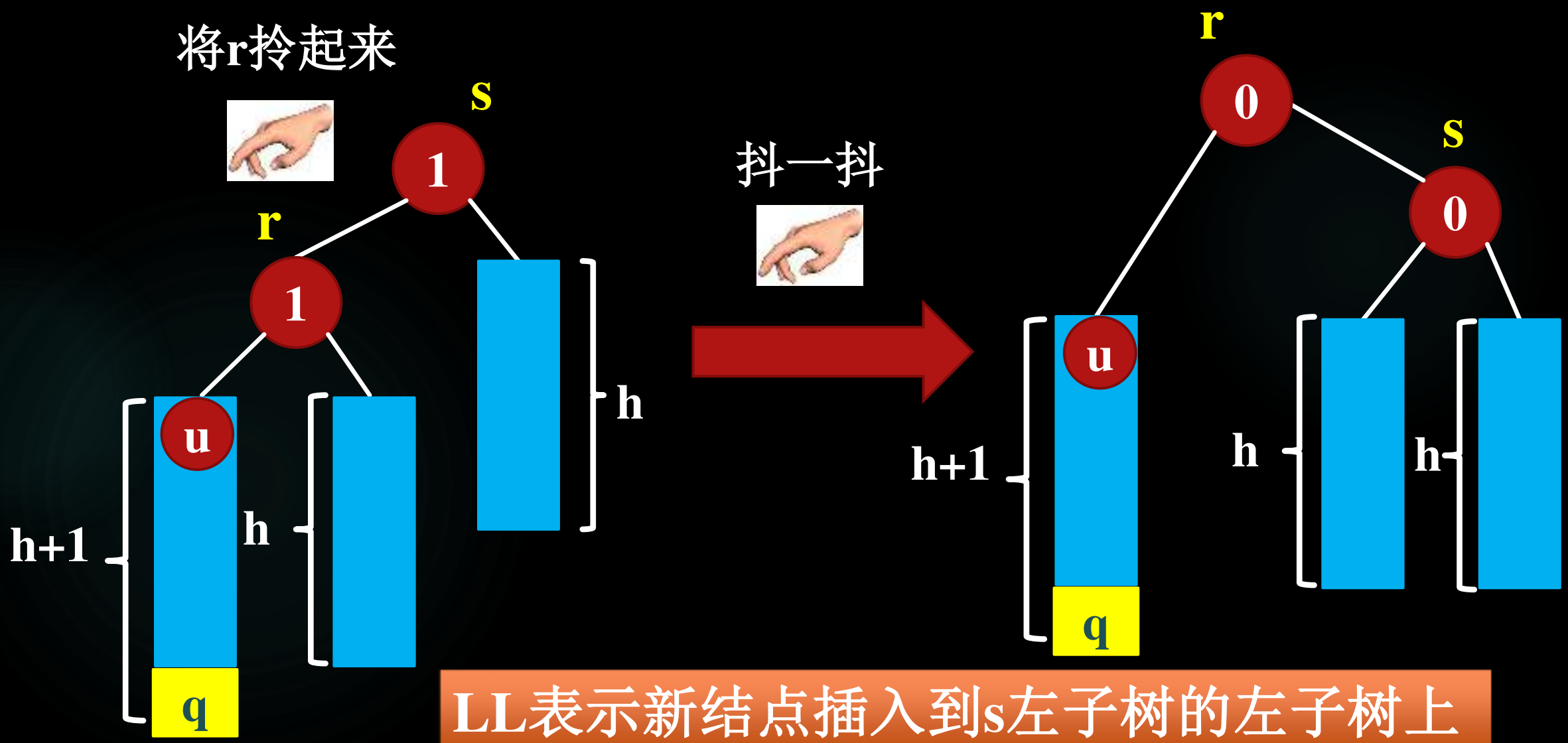
情况三（3）：q添加到r的右子树的右子树上





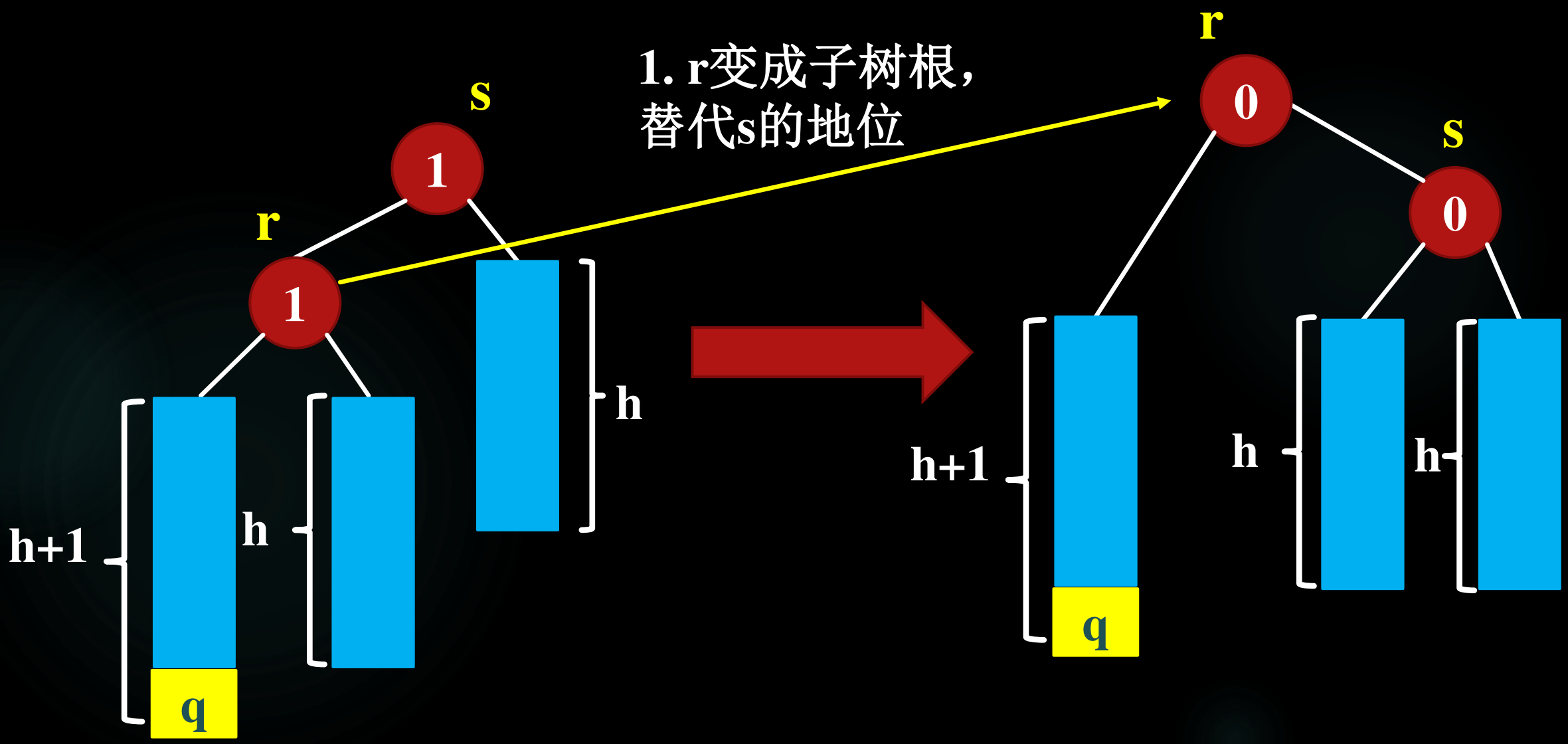
分别就这三种情况进行旋转方案讨论

情况三（1）：LL旋转



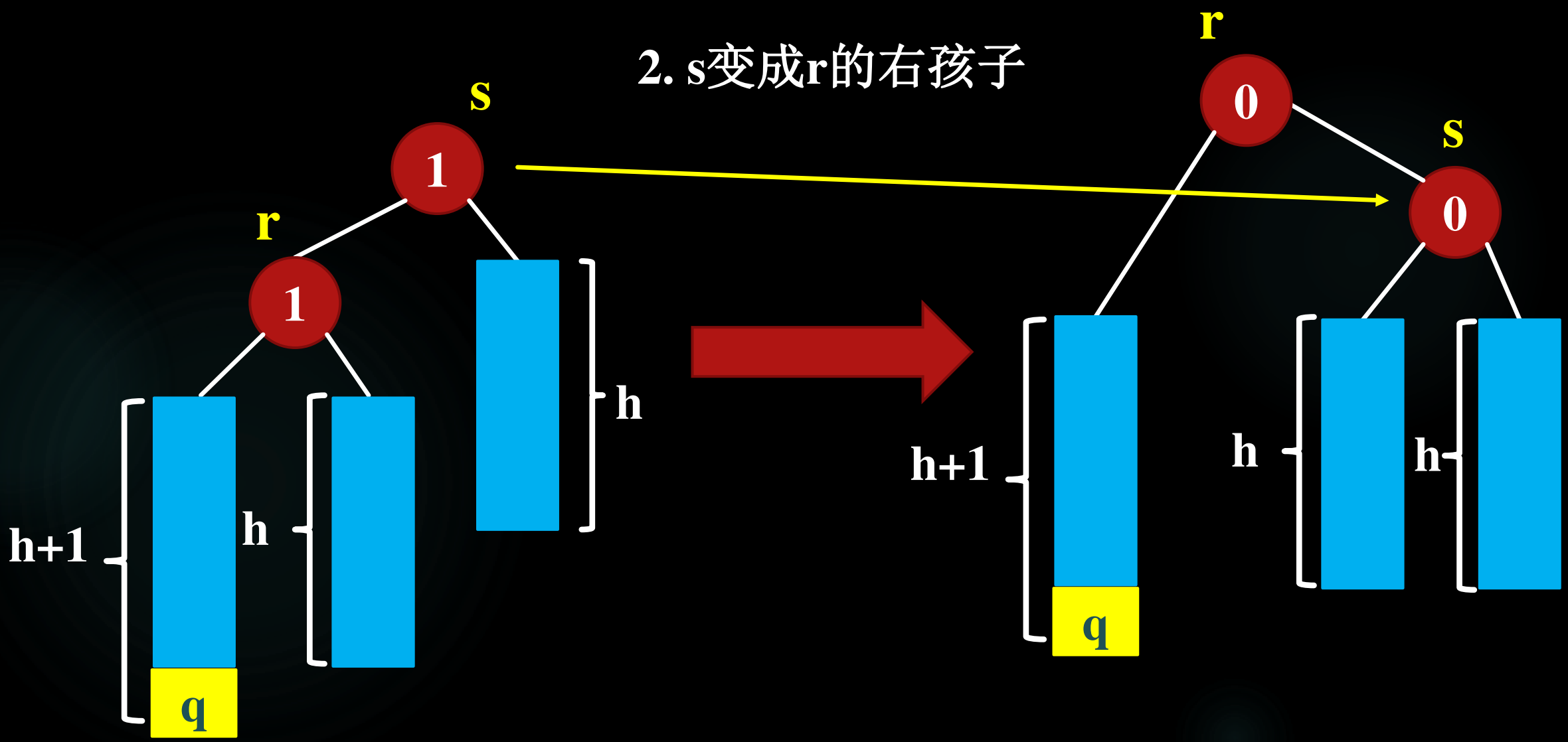
LL表示新结点插入到s左子树的左子树上

情况三（1）：LL旋转

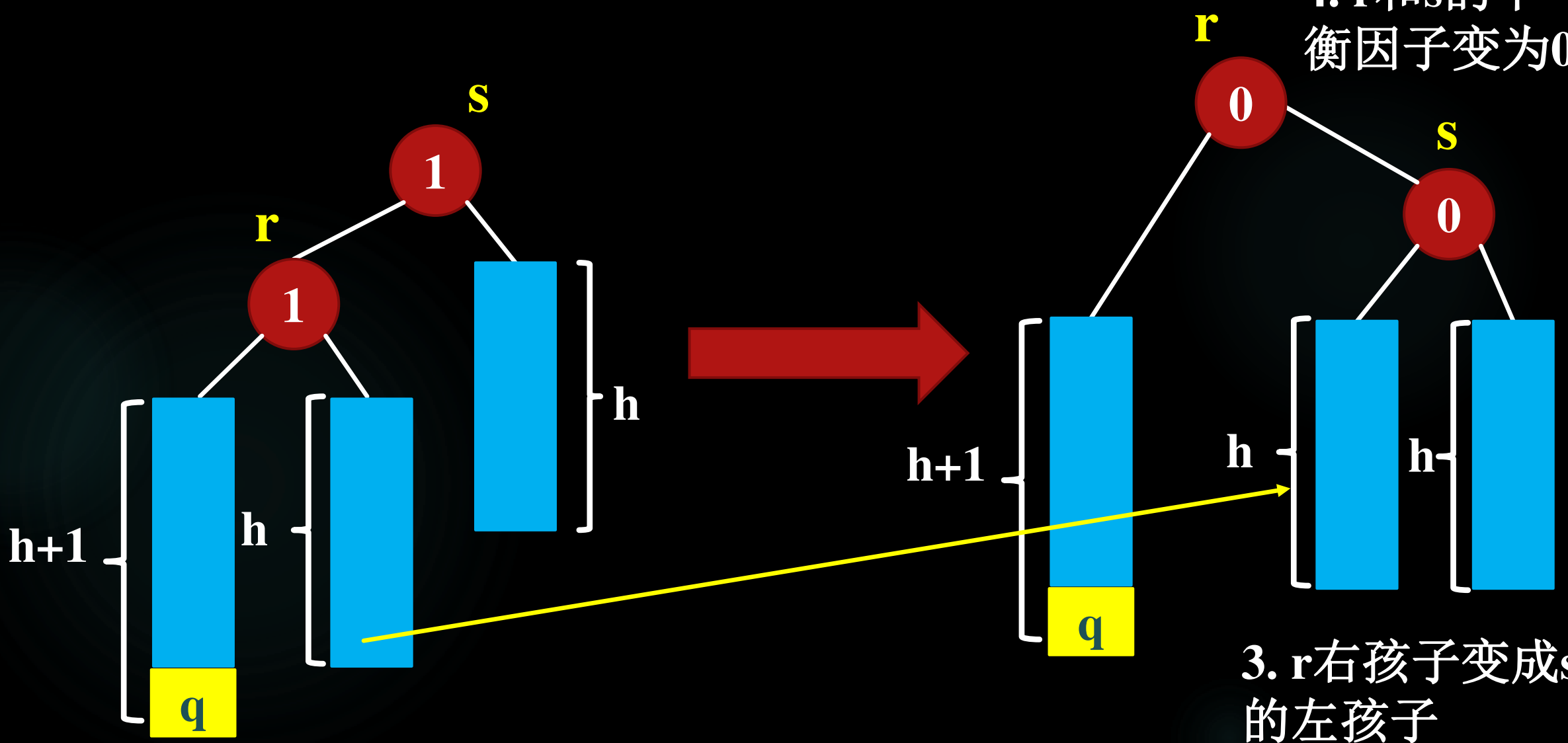


情况三（1）：LL旋转

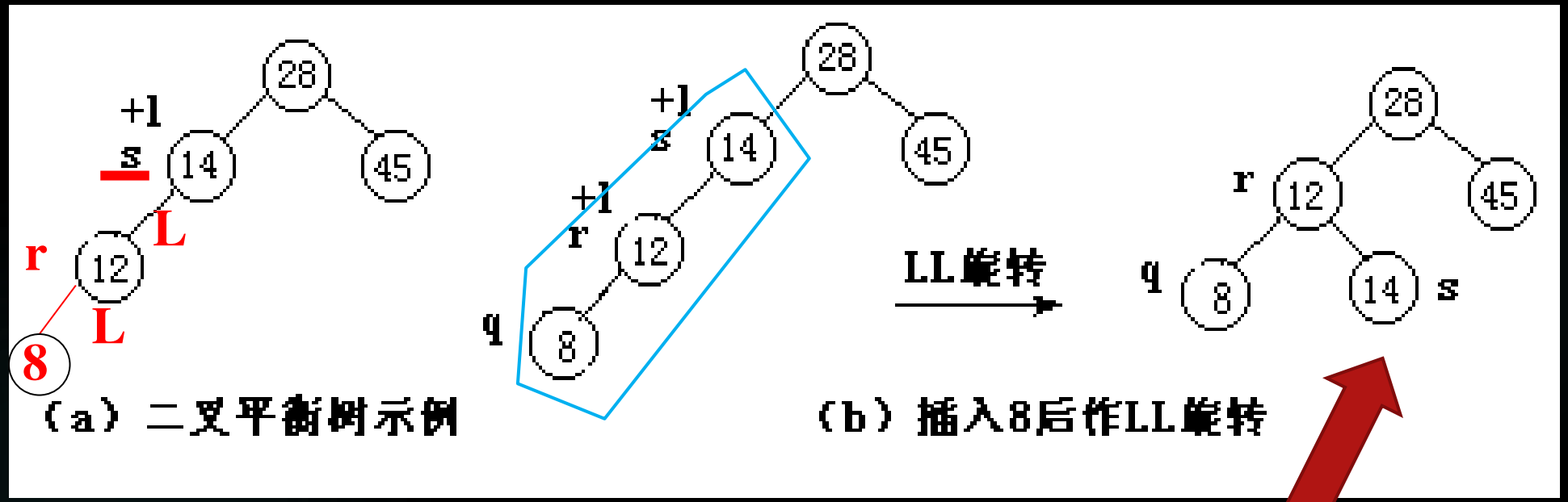
2. s变成r的右孩子

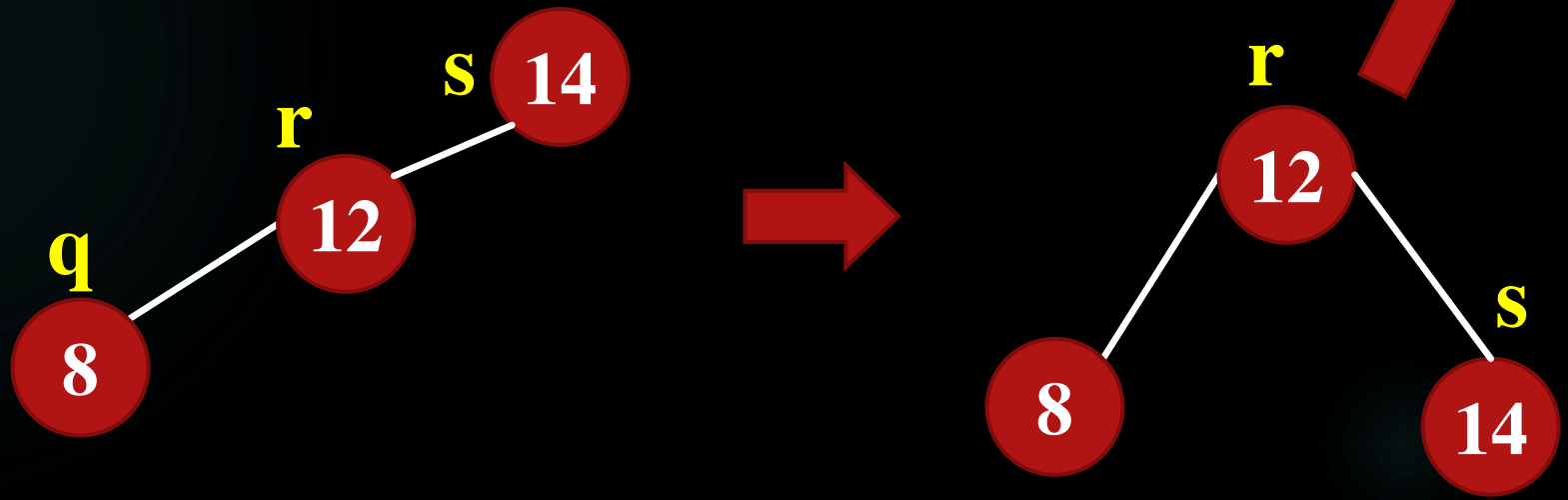


情况三（1）：LL旋转

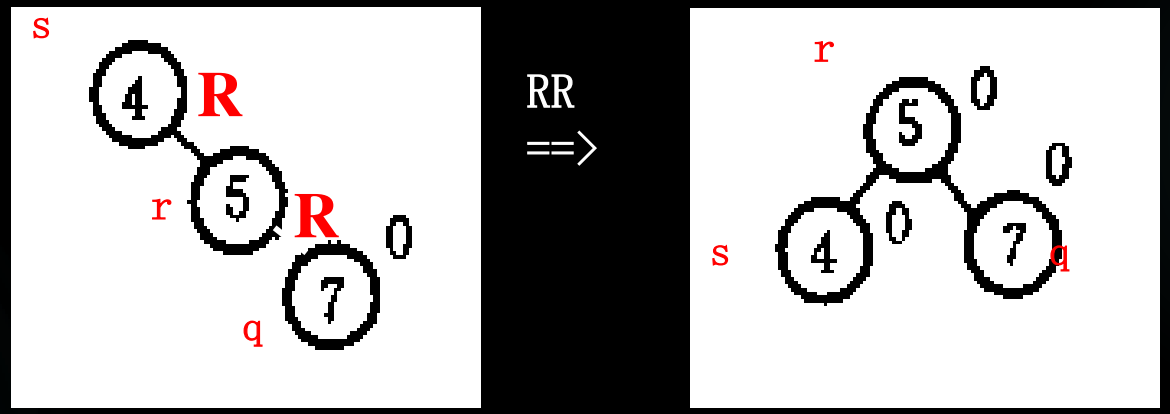


LL-旋转（s->bF=+1， r->bF=+1）

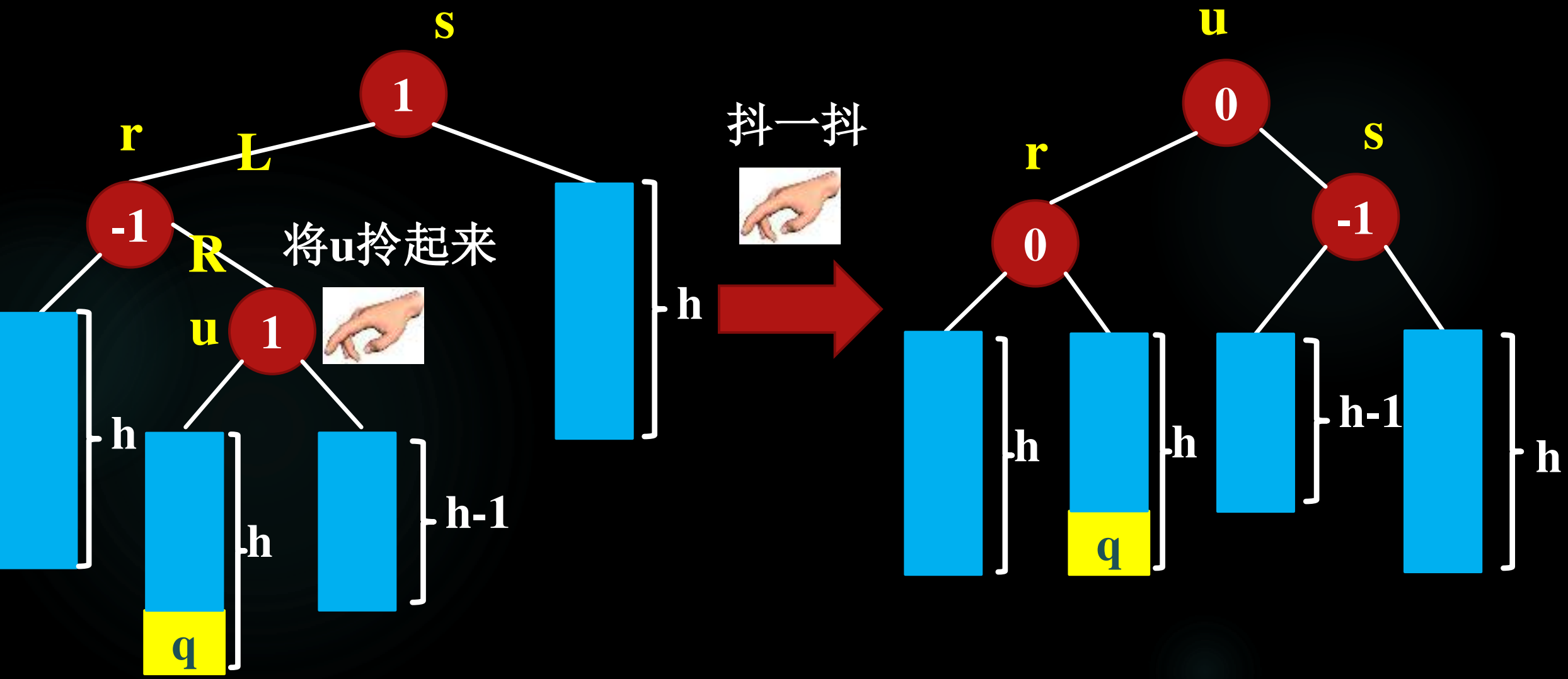




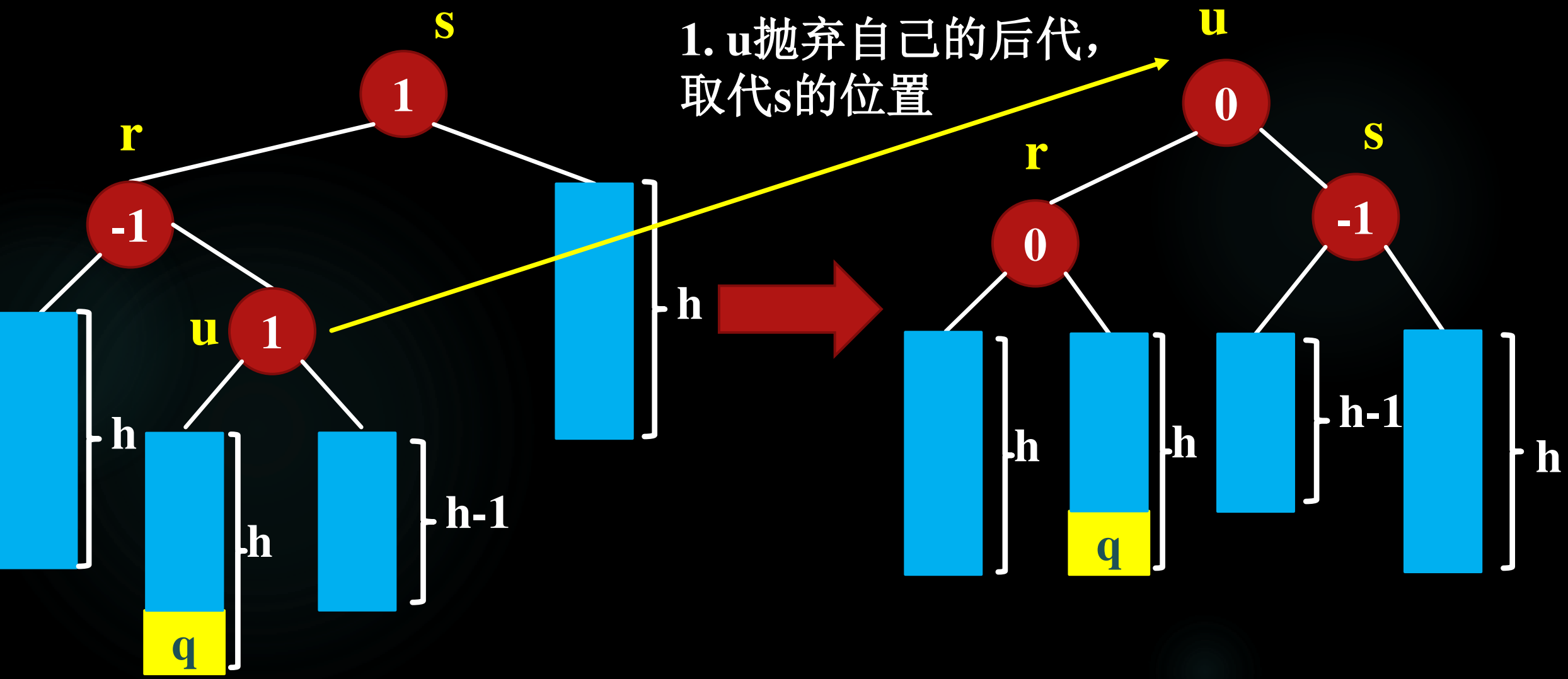
与LL旋转相对应的平衡操作是**RR旋转**，RR旋转是指新结点q插入在 s的**右孩子r的右子树**上时使用的平衡操作。



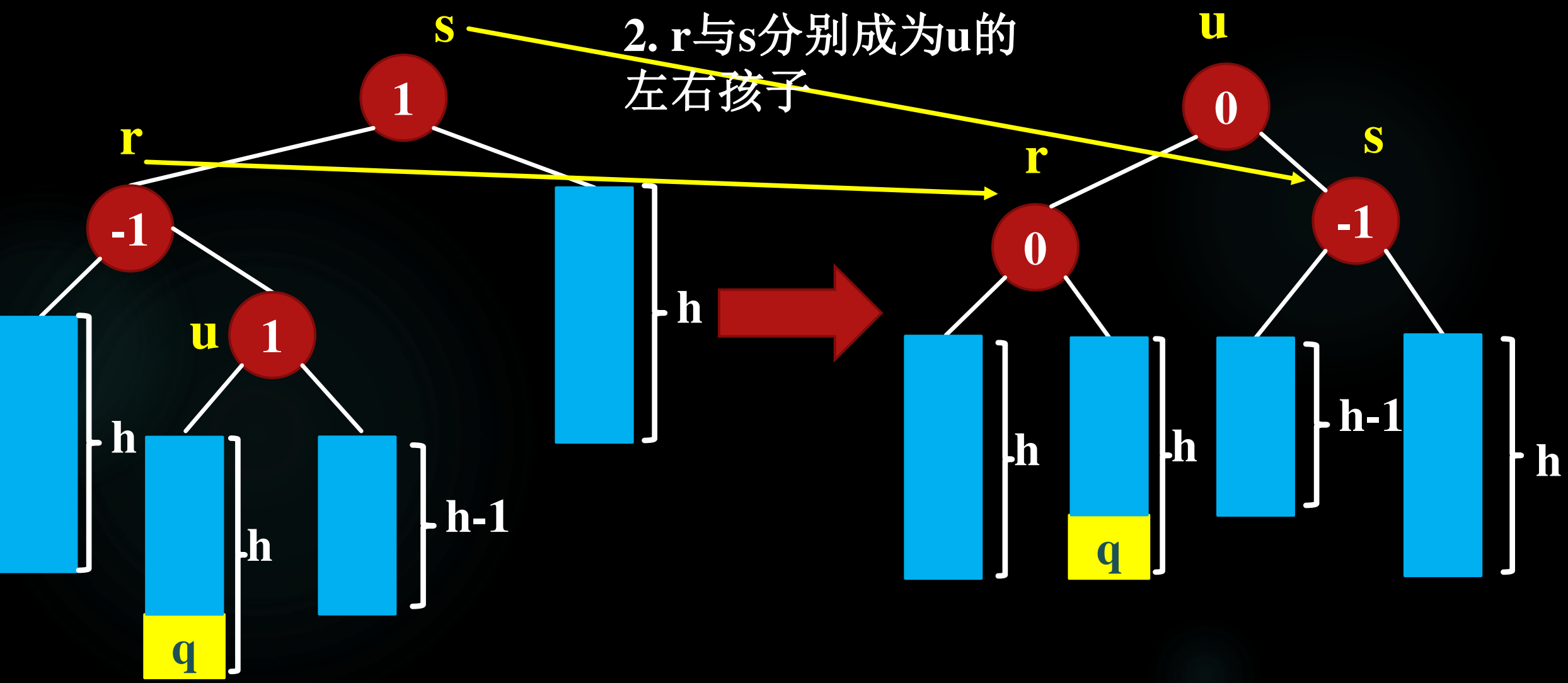
情况三（2）：LR旋转



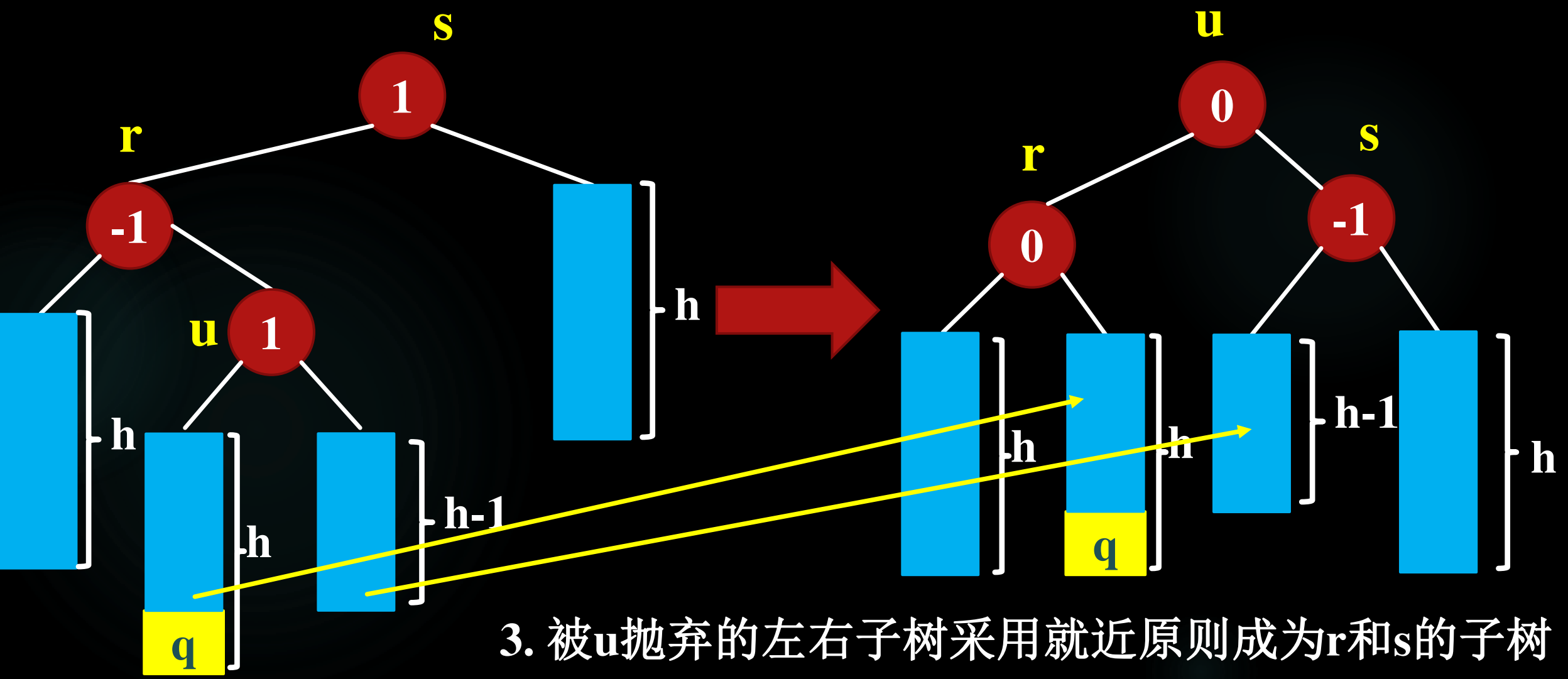
情况三（2）：LR旋转



情况三（2）：LR旋转



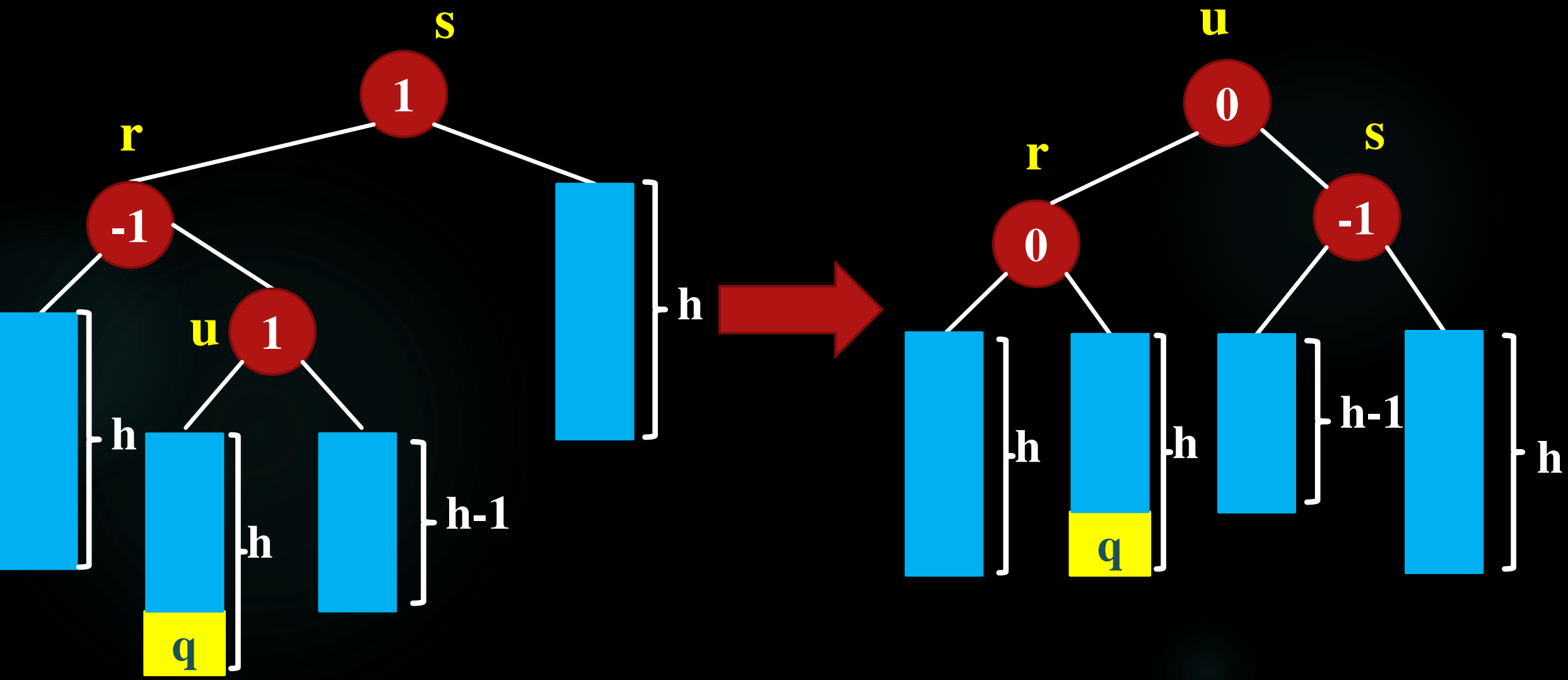
情况三（2）：LR旋转



3. 被u抛弃的左右子树采用就近原则成为r和s的子树

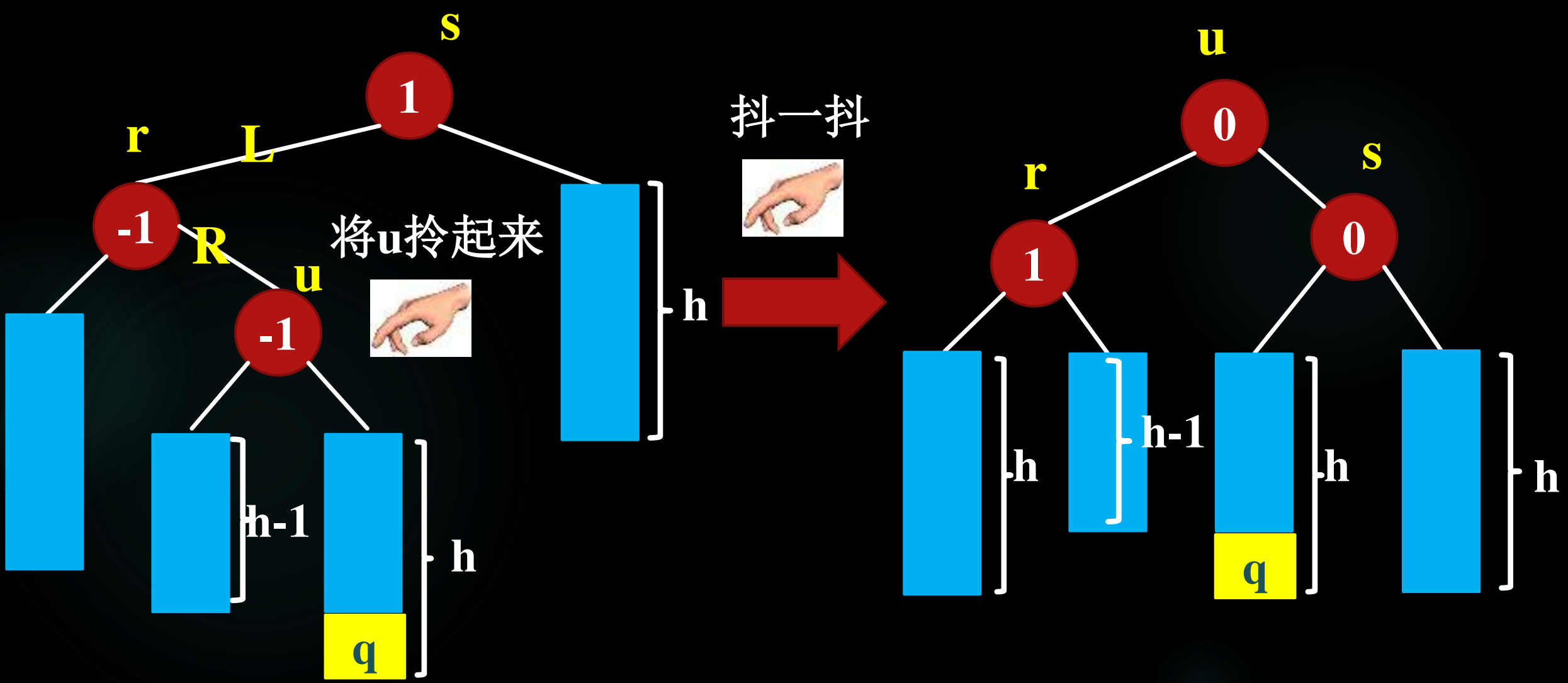
情况三（2）：LR旋转

4. 修改平衡因子

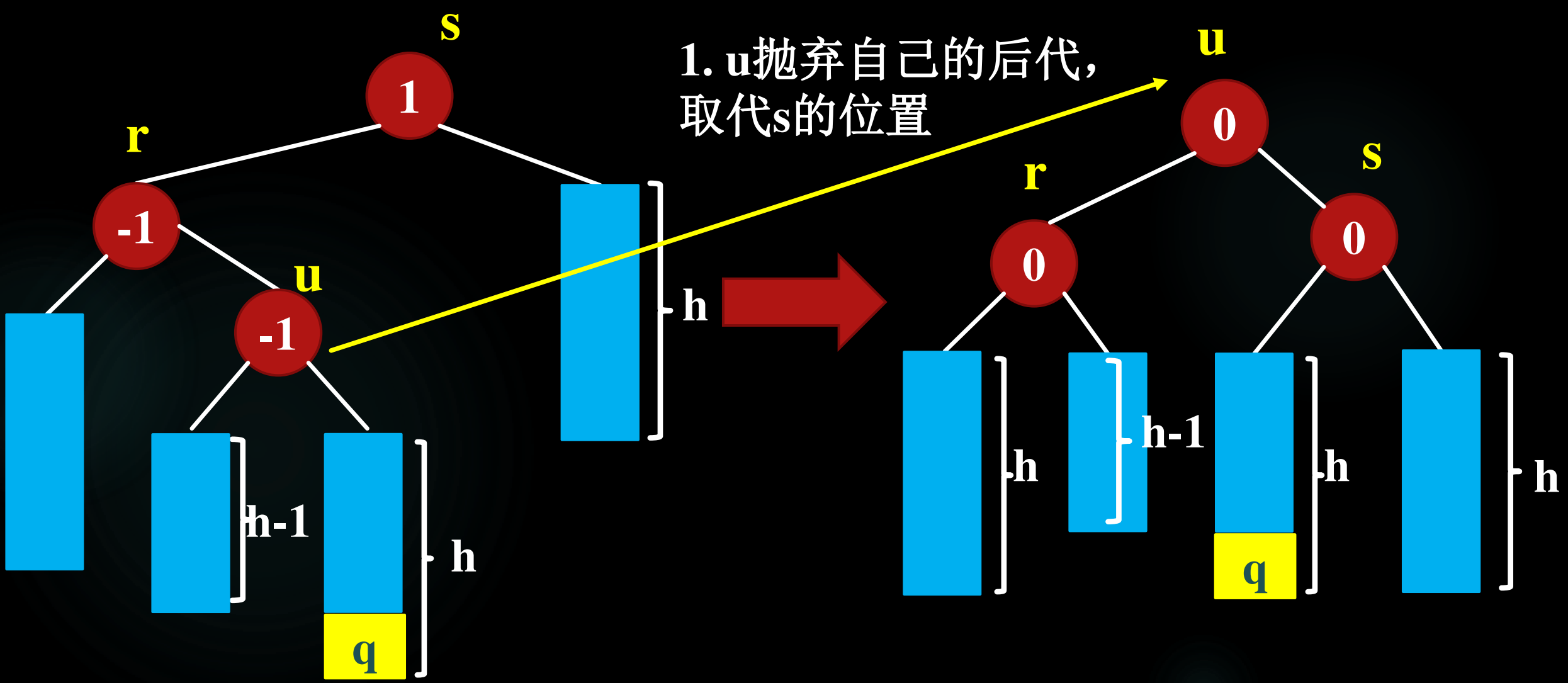


情况三（3）：也是LR旋转

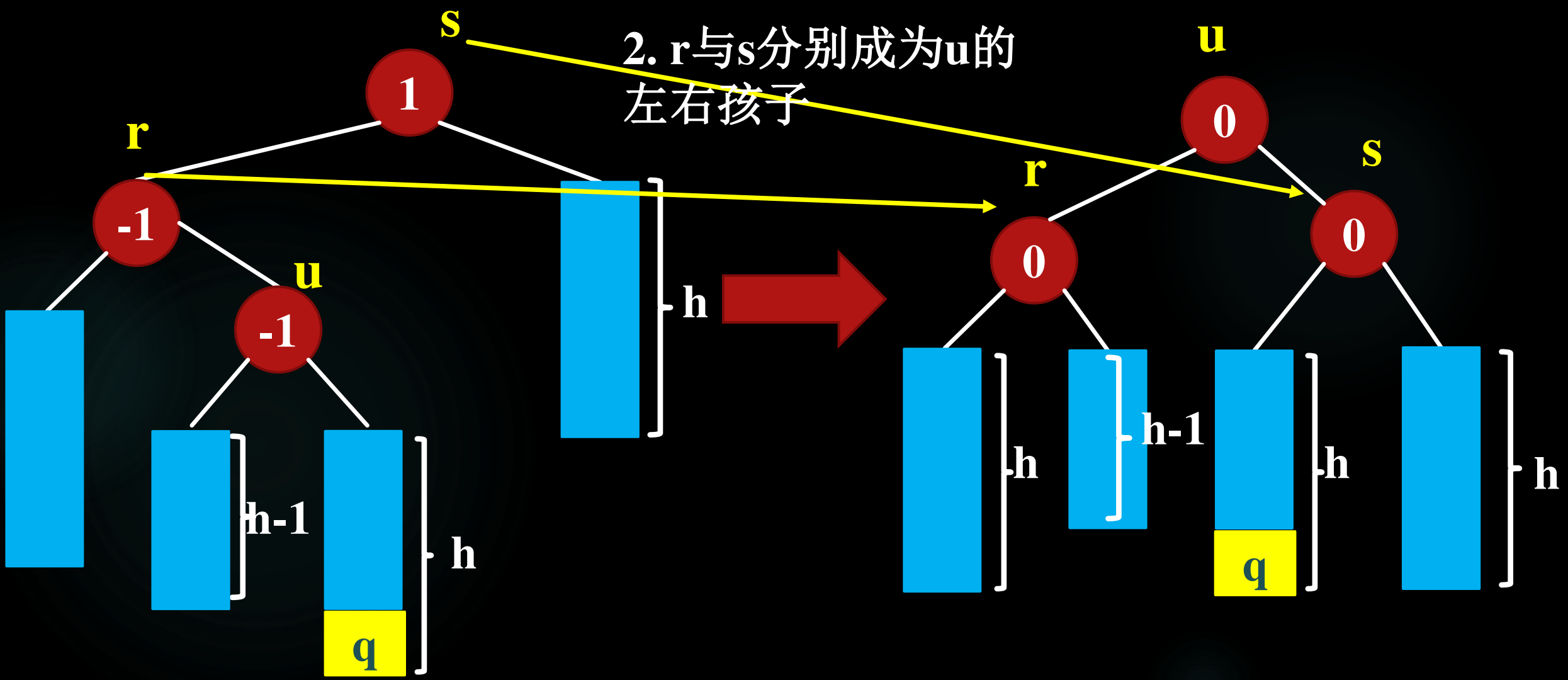
情况三（3）：LR旋转



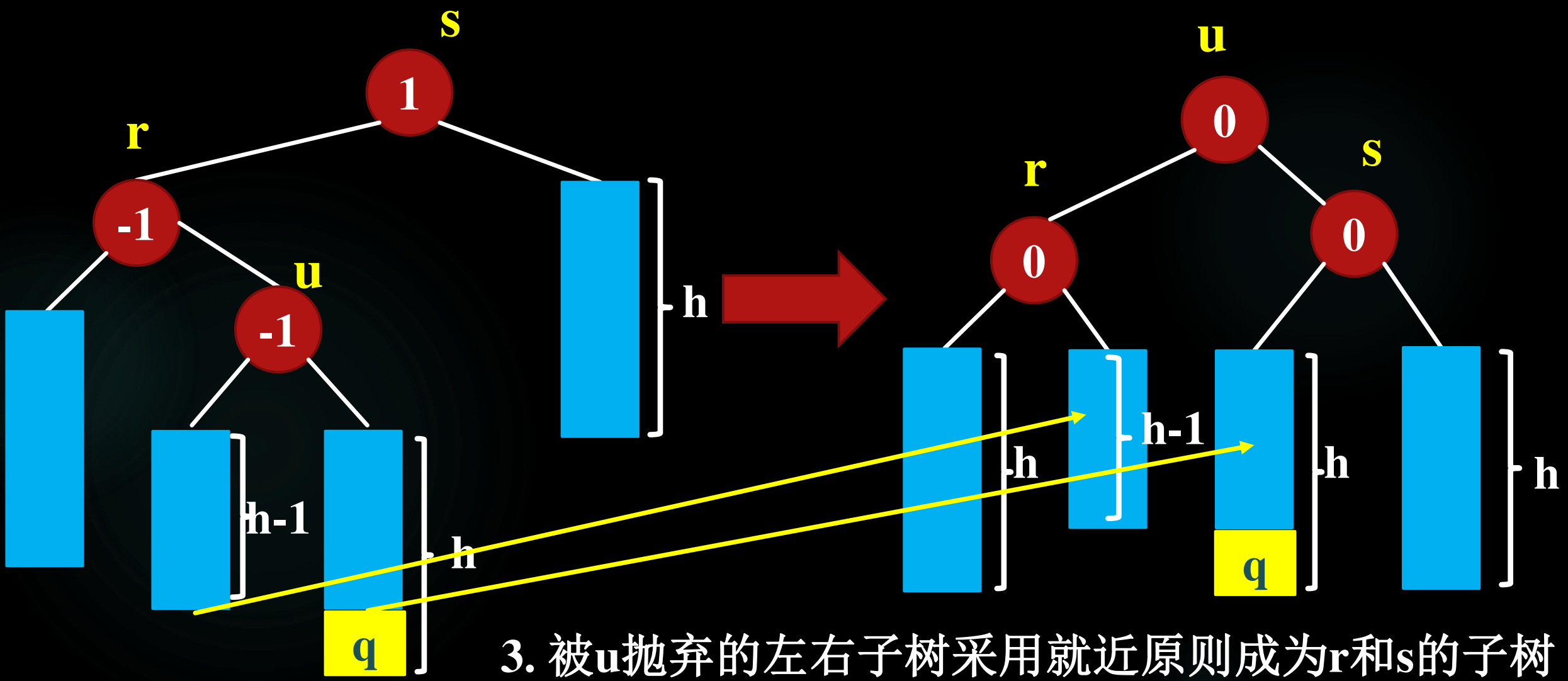
情况三（3）：LR旋转



情况三（3）：LR旋转

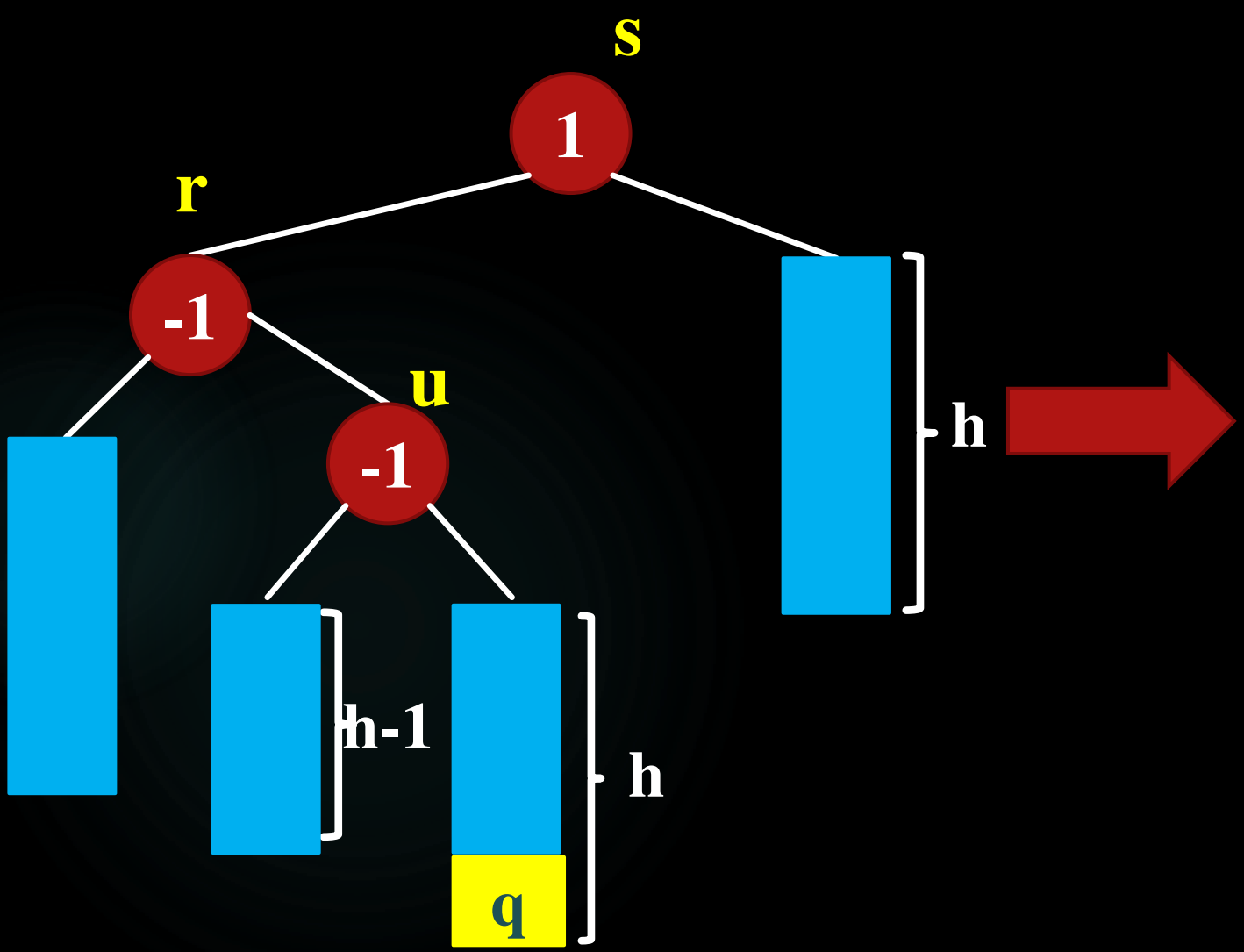


情况三（3）：LR旋转

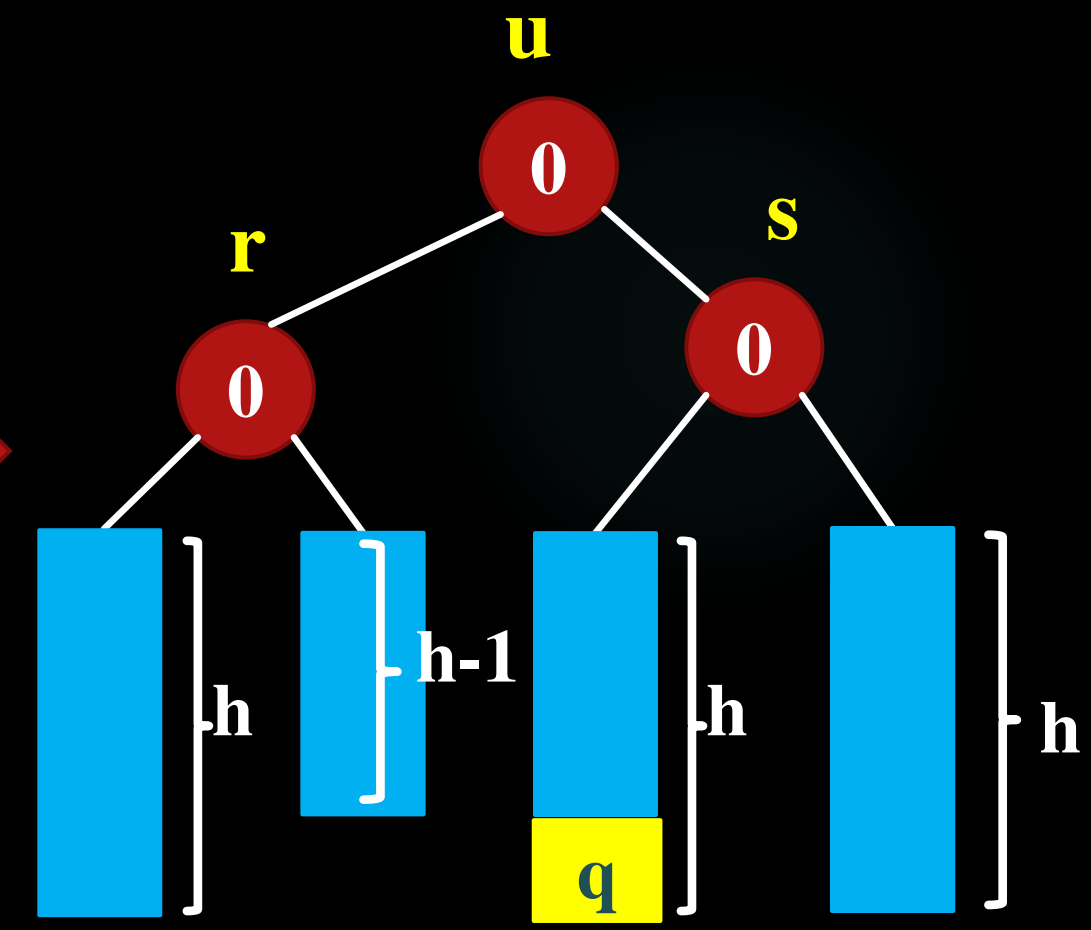


3. 被u抛弃的左右子树采用就近原则成为r和s的子树

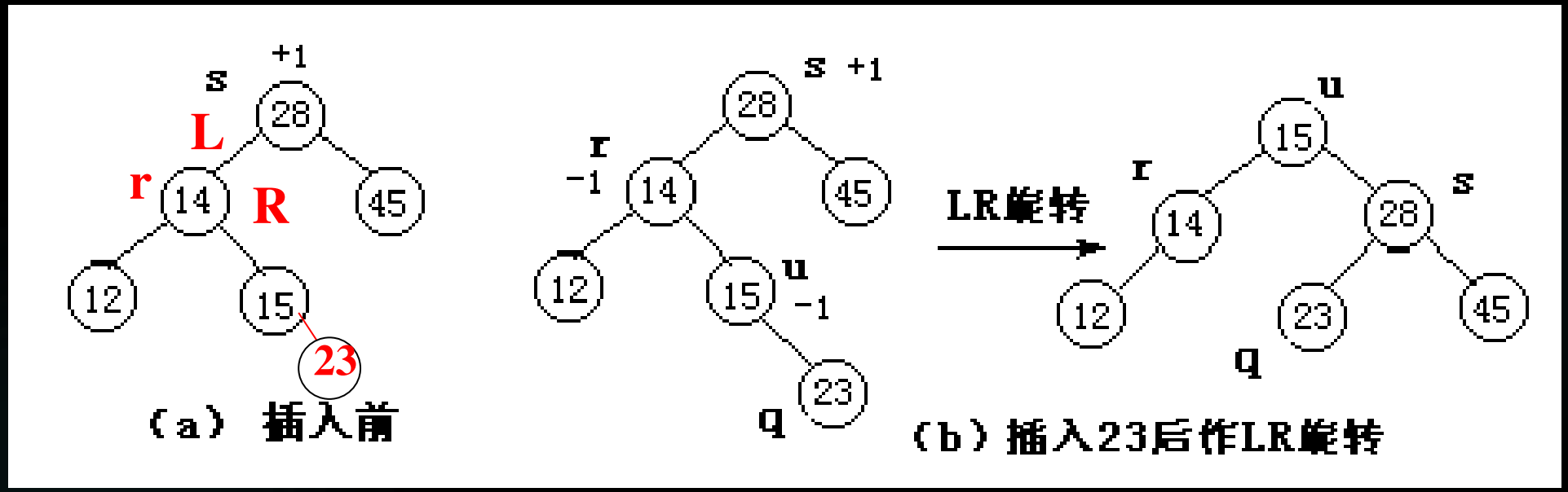
情况三（3）：LR旋转



4. 修改平衡因子

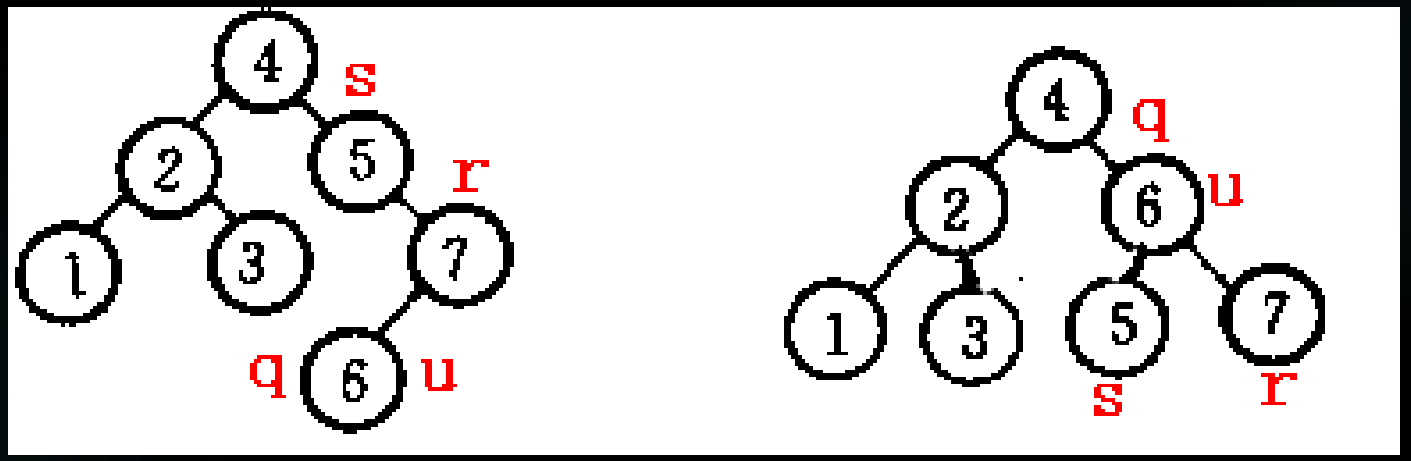


LR-旋转（ s->bf=+1, r->bF=-1）



与LR旋转相对应的平衡操作是**RL旋转**

RL旋转是指新结点q插入在 s的右孩子r的左子树上时使用的平衡操作。



RL型旋转

学习一下，完整的旋转算法

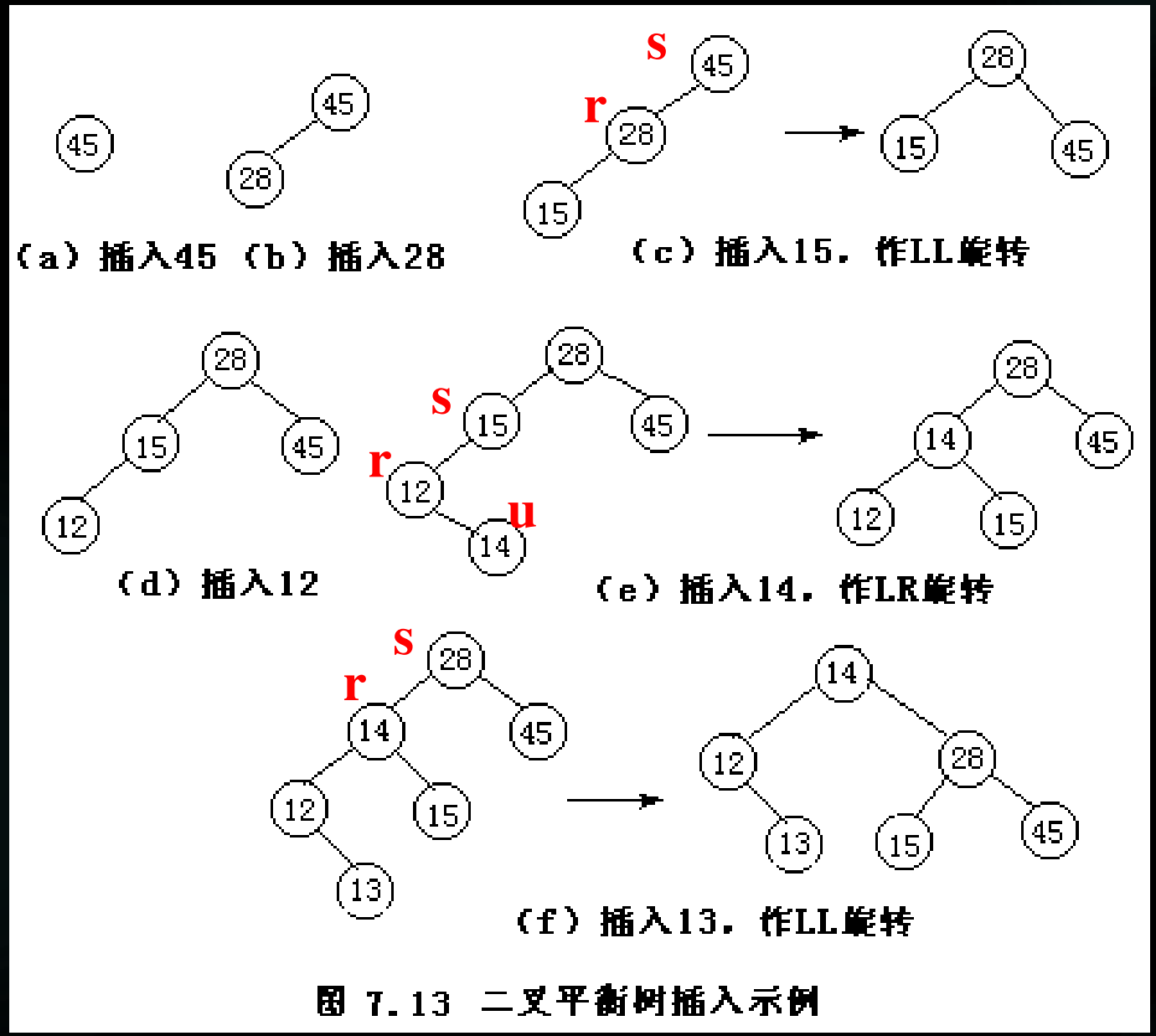
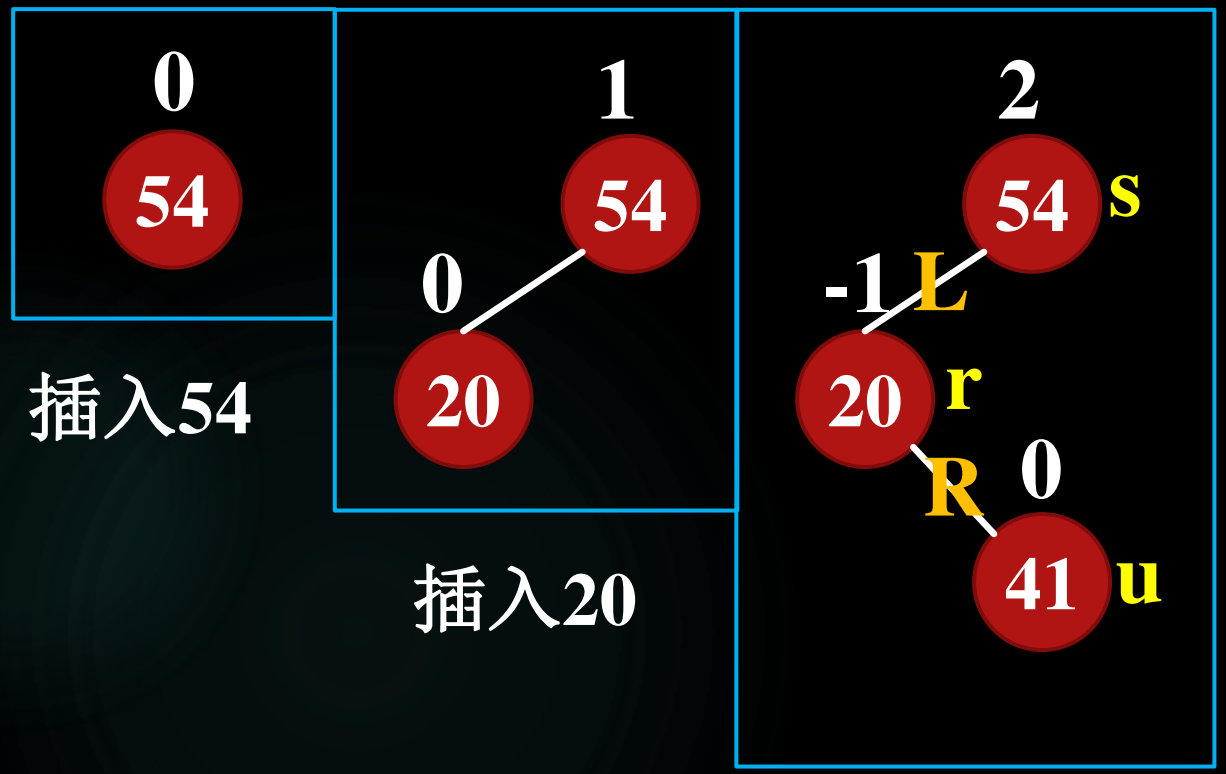


图 7.13 二叉平衡树插入示例

AVL构造步骤

1. 先进行二叉搜索树的插入操作
2. 修改平衡因子
3. 取出最小不平衡子树
4. 在最小不平衡子树上标记s、r、u
5. 判断是LL、RR、LR、RL
6. 进行相应旋转动作
7. 将平衡后的子树放回原树中
8. 检查：平衡性、有序性

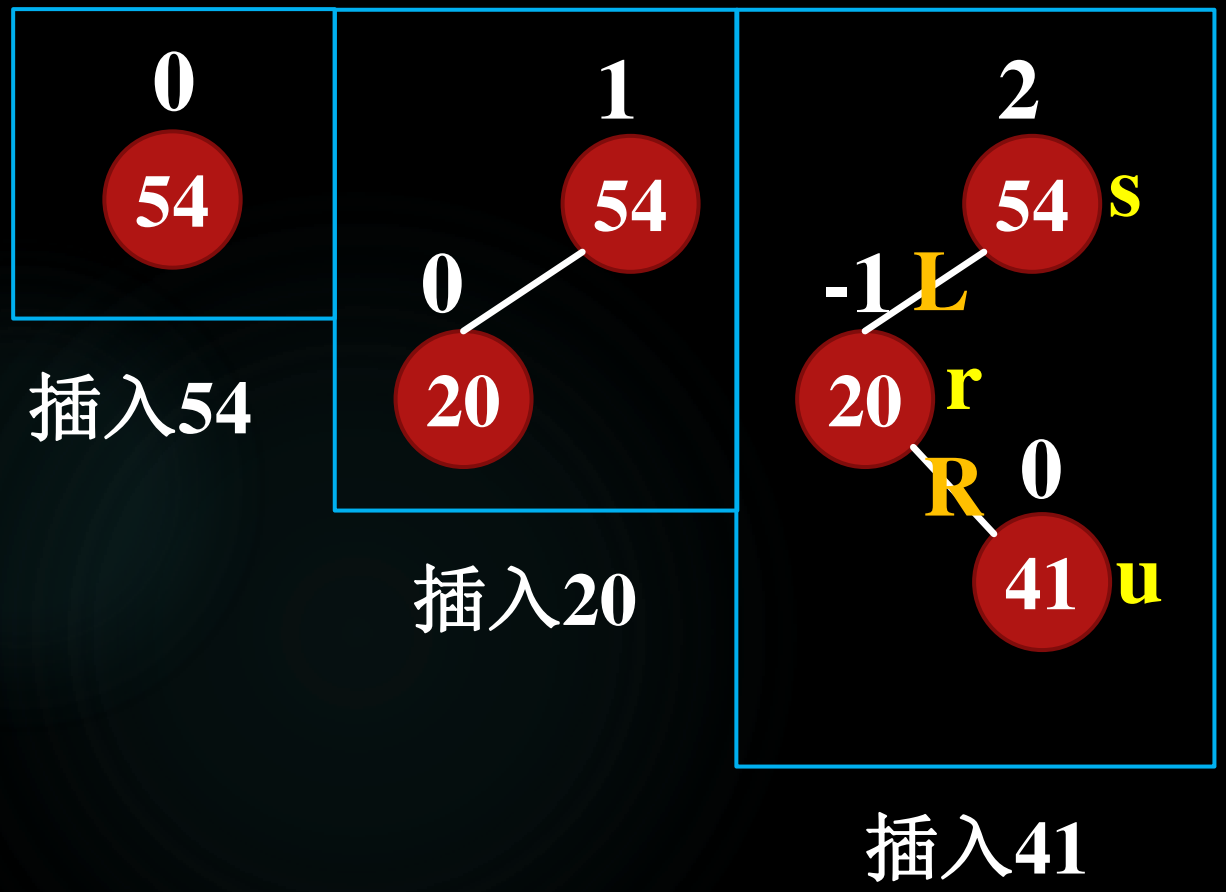
输入关键码序列：54，20，41，80，62，73
画出二叉平衡树的建立过程



最近不平衡祖先怎么找？

1. 从插入新结点位置往根结点方向遇到的第一个平衡因子（尚未更新）非零的结点
2. 插入后更新平衡因子，离插入结点最近的平衡因子为2或-2的结点

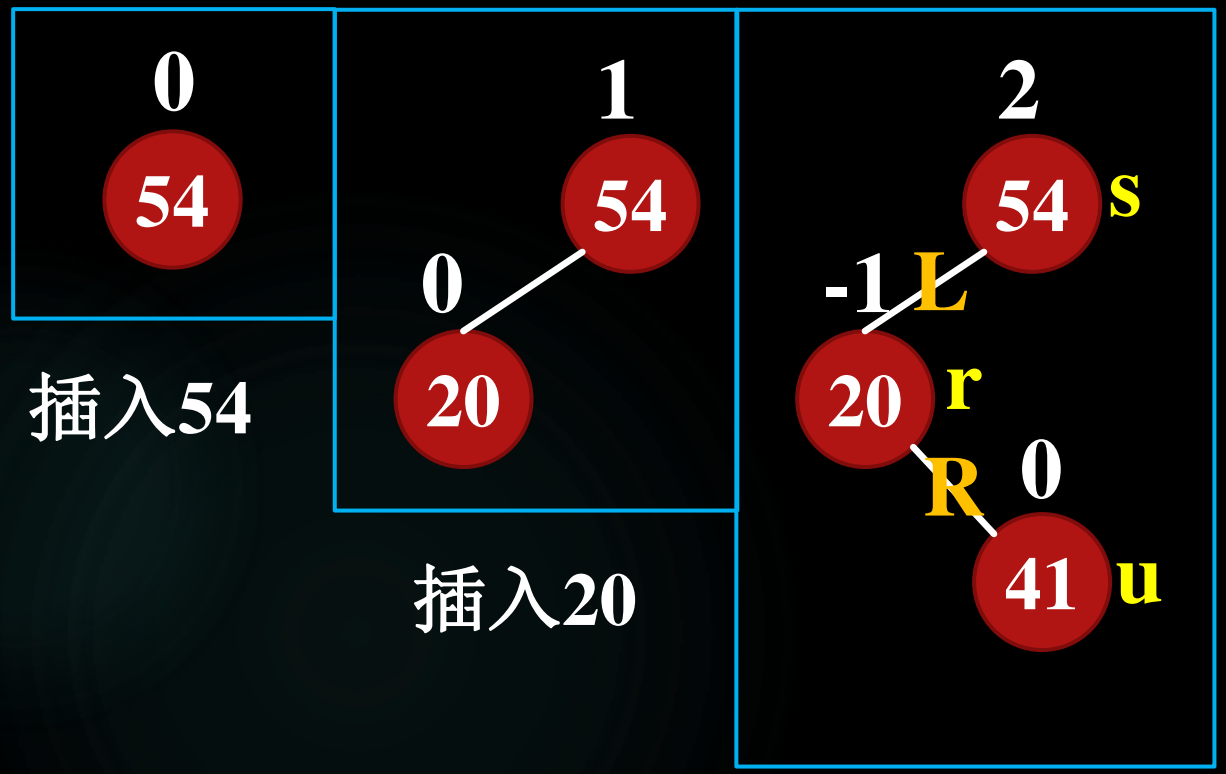
输入关键码序列：54，20，41，80，62，73
画出二叉平衡树的建立过程



如何标记s、r和u？

1. 最近不平衡祖先为s
2. 从s到插入的新结点路径上，紧靠s的后两个结点分别是r和u

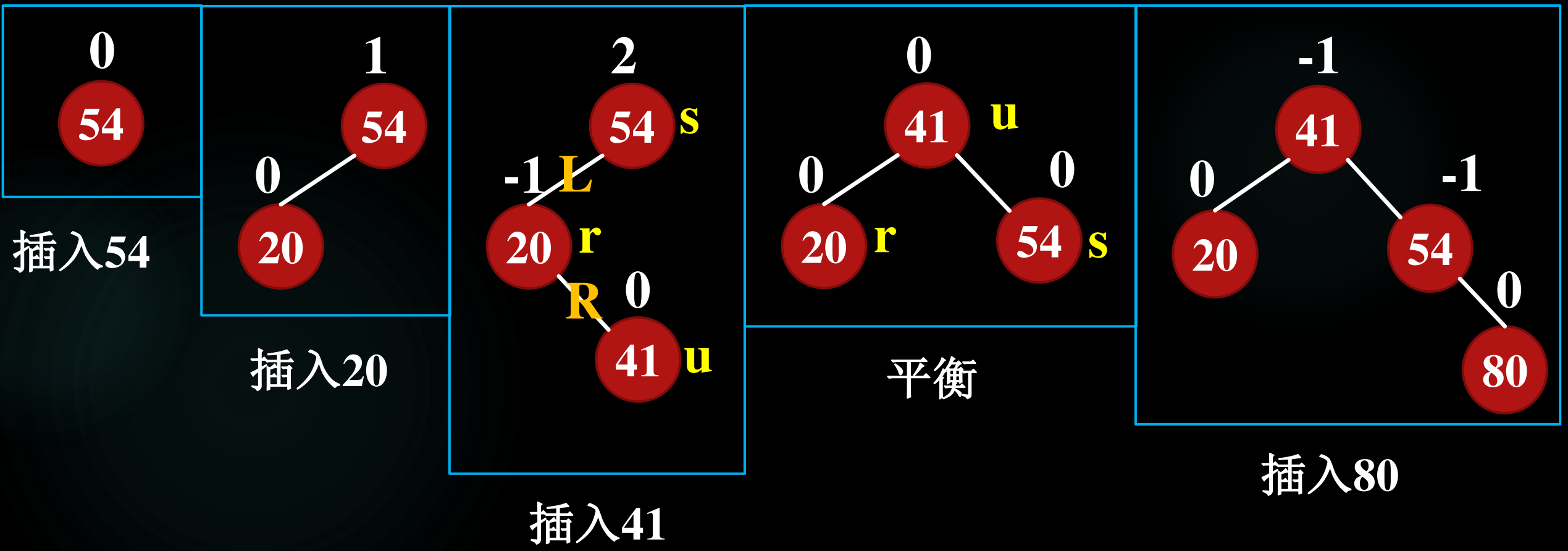
输入关键码序列：54，20，41，80，62，73
画出二叉平衡树的建立过程



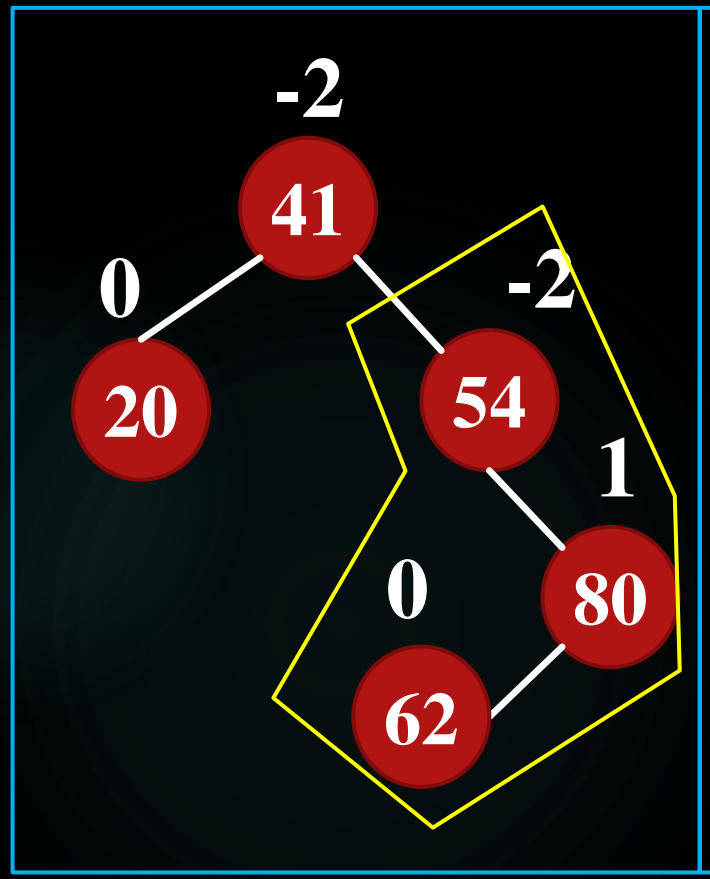
如何标记L和R？

1. s与r之间的路径如果为左分支，则为L，否则为R
2. r与u之间的路径如果为左分支，则为L，否则为R

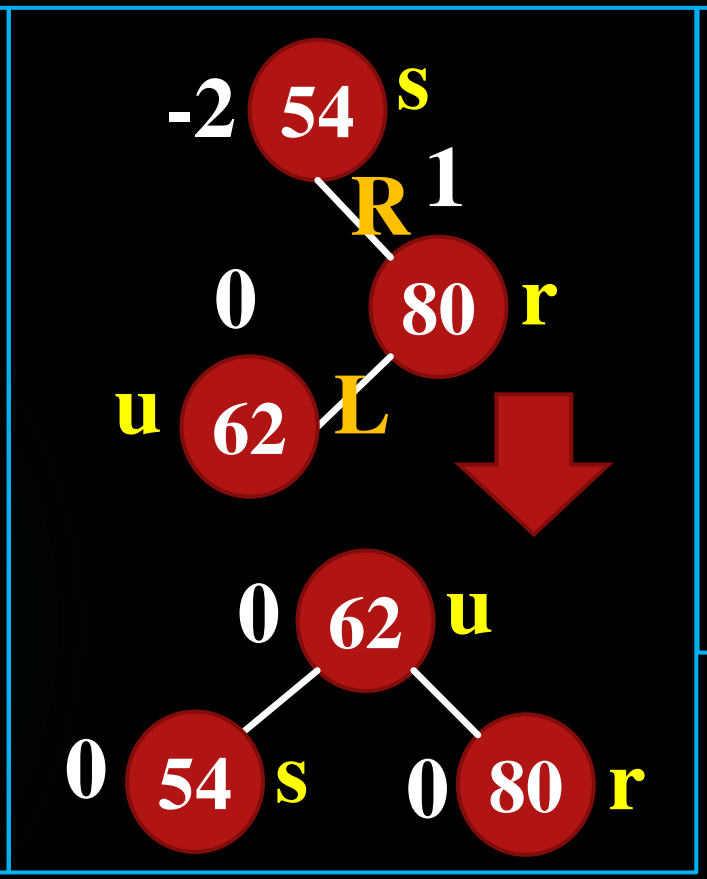
输入关键码序列：54，20，41，80，62，73
画出二叉平衡树的建立过程



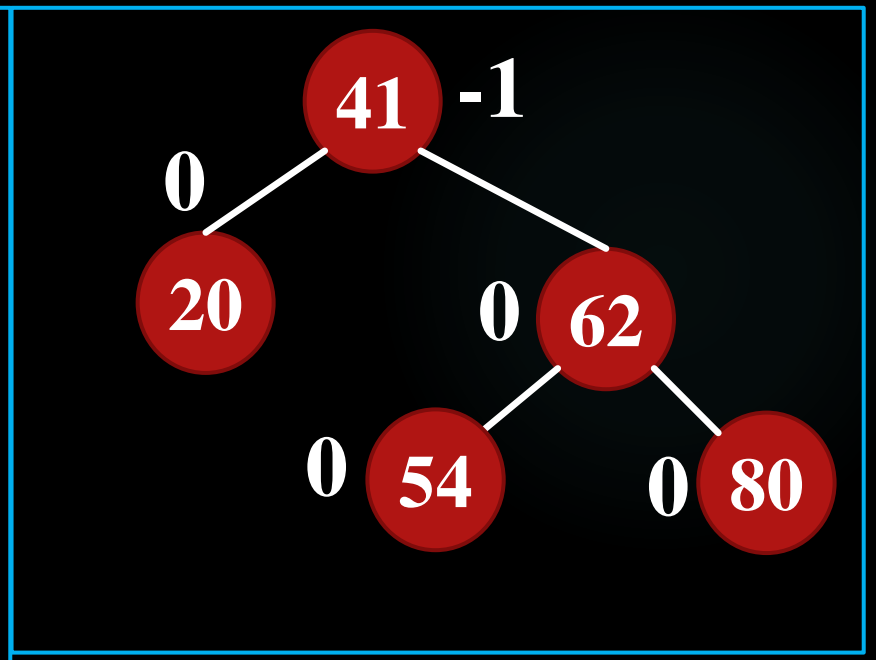
输入关键码序列：54，20，41，80，62，73
画出二叉平衡树的建立过程



插入62

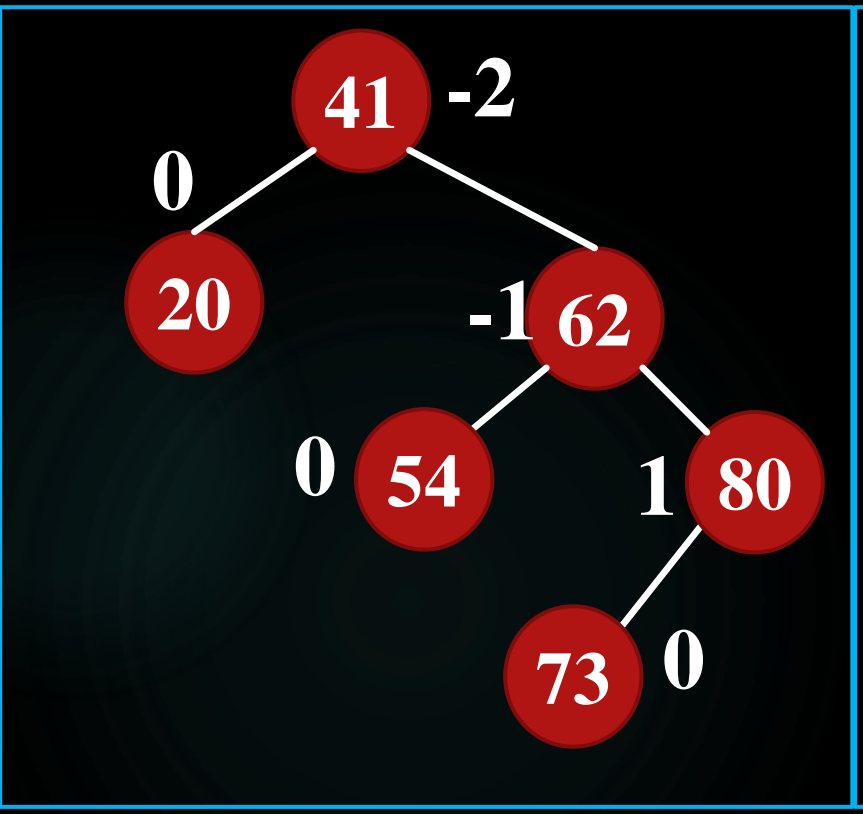


平衡

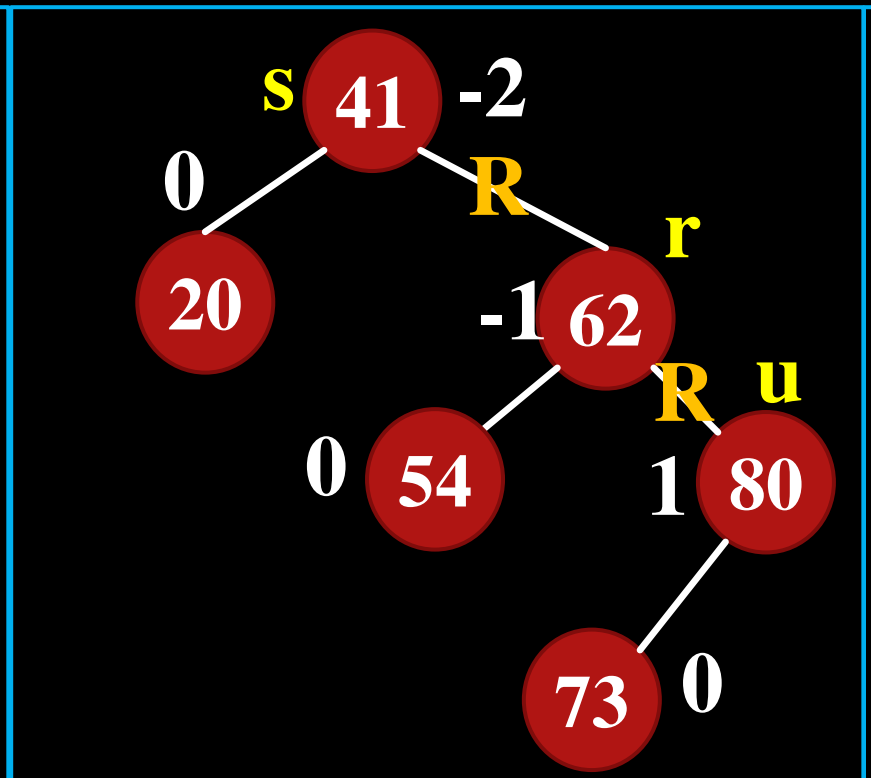


放回原树，检查

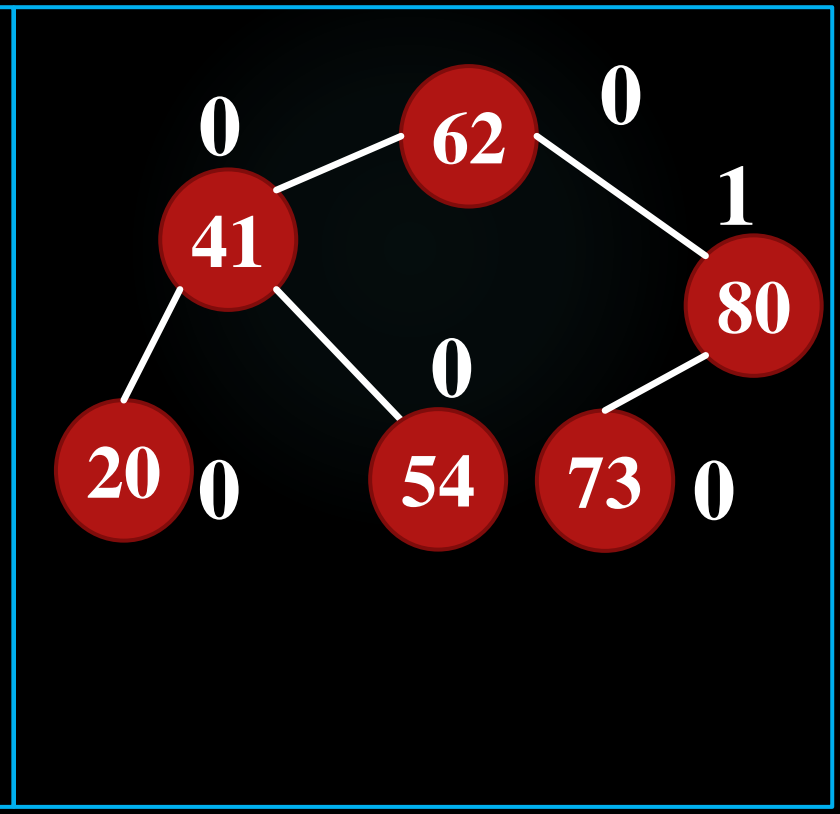
输入关键码序列：54，20，41，80，62，73
画出二叉平衡树的建立过程



插入73



准备工作



调整

输入关键码序列：16,3,7,11,9,26,18,14,15
画出二叉平衡树的建立过程

从空树开始依次向二叉平衡树插入关键字为0,4,8,3,2,1,7,6的数据元素，画出二叉平衡树的构建过程（要求给出每一步插入后的树形）。

下述二叉树中，哪一种满足性质：从任意结点出发到根结点的路径上所经过的结点关键字有序排列？

A：AVL树

B：哈夫曼树

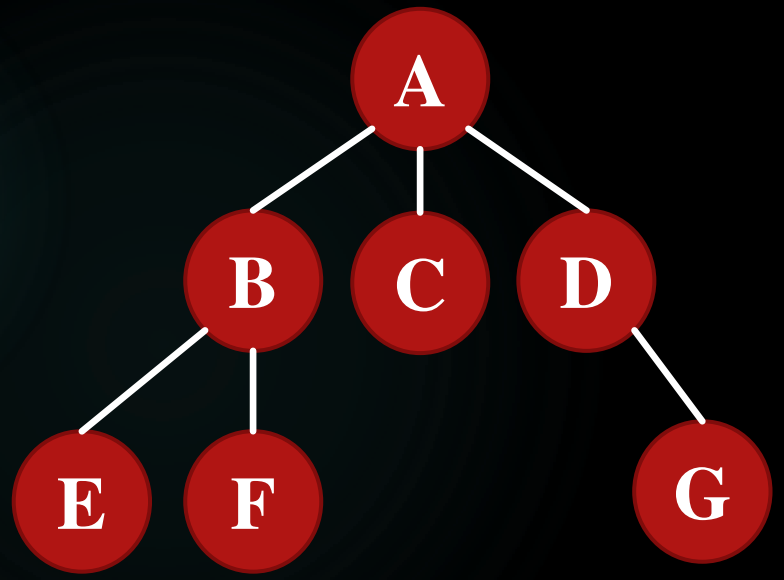
C：二叉排序树

D：最小堆

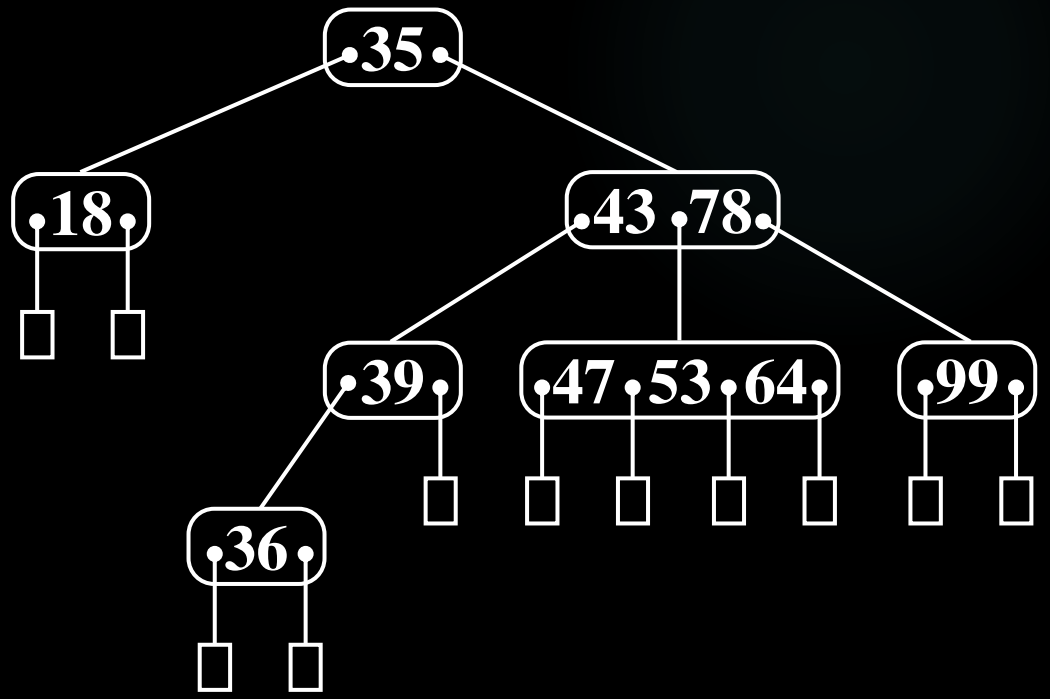
搜索树

M叉搜索树（M>2）

- M叉搜索树 VS M叉树

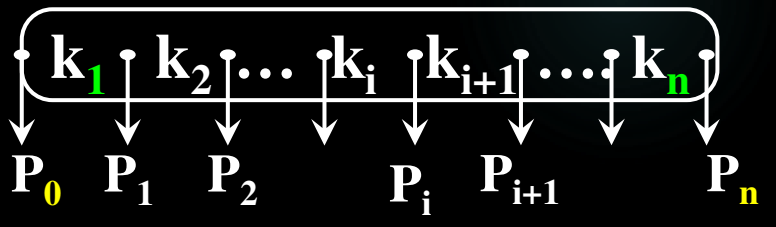


搜索树必须能够表达某种顺序性



四叉搜索树

m叉搜索树的定义



定义 m叉搜索树或者是空m叉搜索树，或者是一棵满足下列特性的树

(1) 根结点**最多**有**m棵子树**，并具有如下结构：

P_0，（K_1，P_1），（K_2，P_2），…，（K_n，P_n）

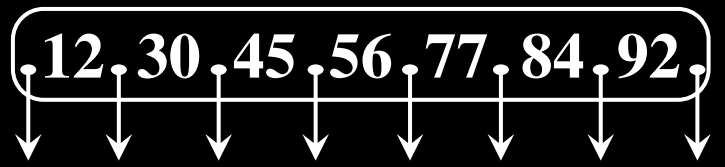
K关键字
P指向子树的指针

(2) $K_i<K_{i+1}$，$1 \le i<n$ （结点中的关键字是有序递增的）

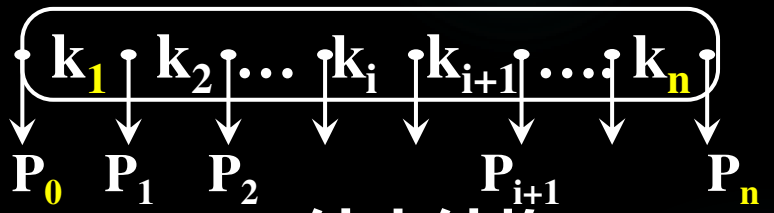
(3) 子树P_i上的所有关键字值都大于K_i，小于K_{i+1}，$0<i<n$。

(4) 子树P_0上的所有关键字值都小于K_1，
子树P_n上的所有关键字值都大于K_n。

(5) 子树P_i（$0 \le i \le n$）也是m叉搜索树。



(a) 结点示例



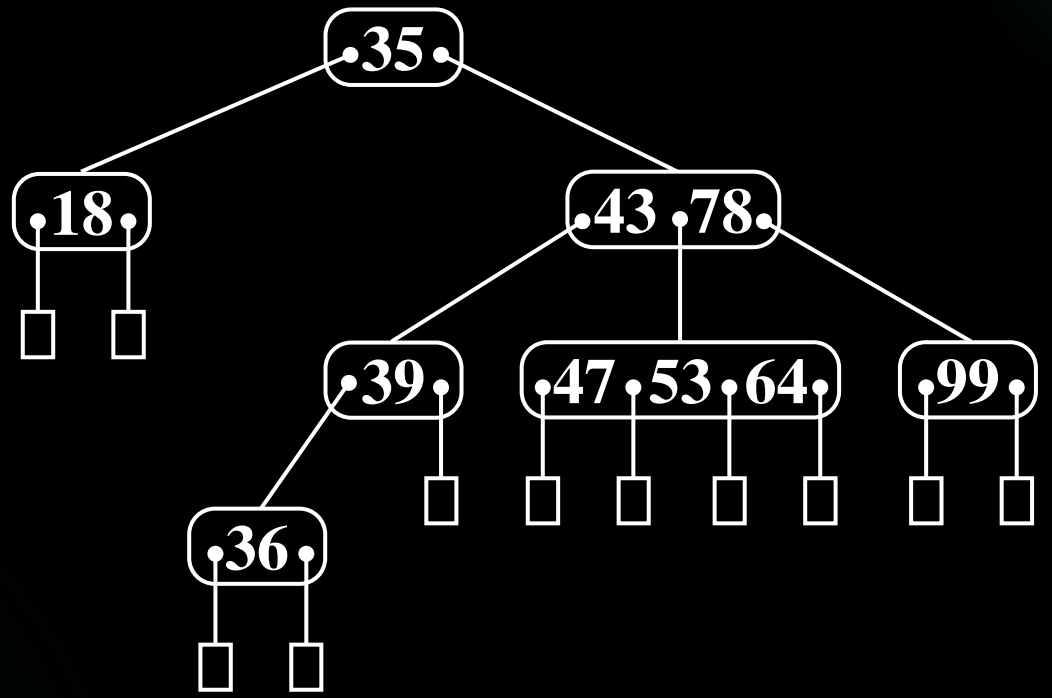
(b) 结点结构

从定义中可以得到：

(1)一个m叉搜索树的结点中，最多存放m-1个元素和m个指向子树的指针。

(2) 每个结点中包含的元素个数=它包含的指针数-1。

(3) 每个结点中元素按关键字值递增排列，一个元素的关键字值大于它的左子树上所有结点中元素的关键字值，小于它的右子树上所有结点中元素的关键字值。



四叉搜索树

图中的方块代表空树。**空树**也称为**失败结点**，因为这是当搜索某个关键字值x不在树中时到达的子树。

失败结点中不包含元素，**失败结点不是叶子结点**！

为什么要有M叉搜索树？

内搜索:当集合足够小，可以驻留在内存中时，相应的搜索方法称为内搜索。

外搜索：如果文件很大，以至于计算机内存容不下时，它们必须存放在外存中。在外存中搜索给定关键字值的元素的方法称为外搜索。

内存中集合用二叉平衡树表示。外存中，集合可以用B-树来表示。

典型的磁盘存取时间是1ms ~10ms，而典型的内存存取时间是10ns~100ns。内存的存取速度比磁盘快1万至百万倍。

设法减少磁盘存取操作的次数是外搜索算法设计应充分考虑的问题。

采用多叉树代替二叉树，在一个结点中存放多个元素而不是一个元素是明智的做法。例如，可将7个元素组织在一个结点（也称为1页）中，如图7.20所示。

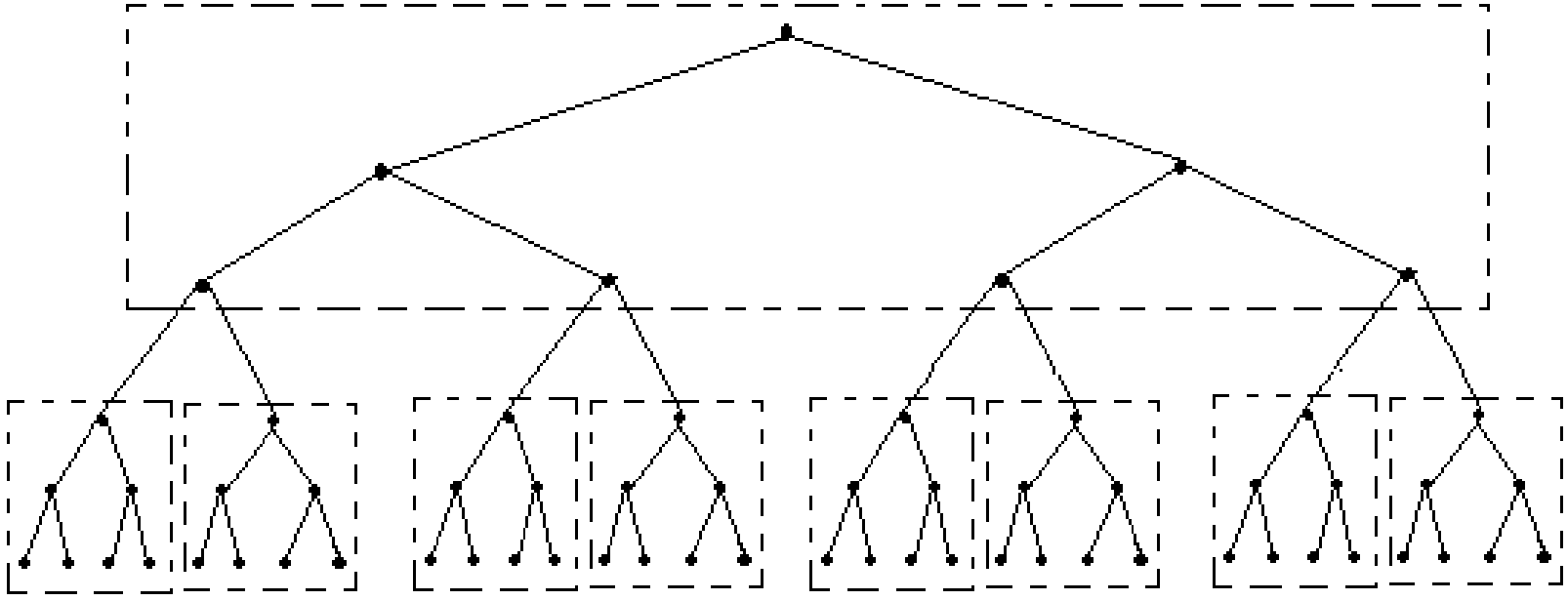
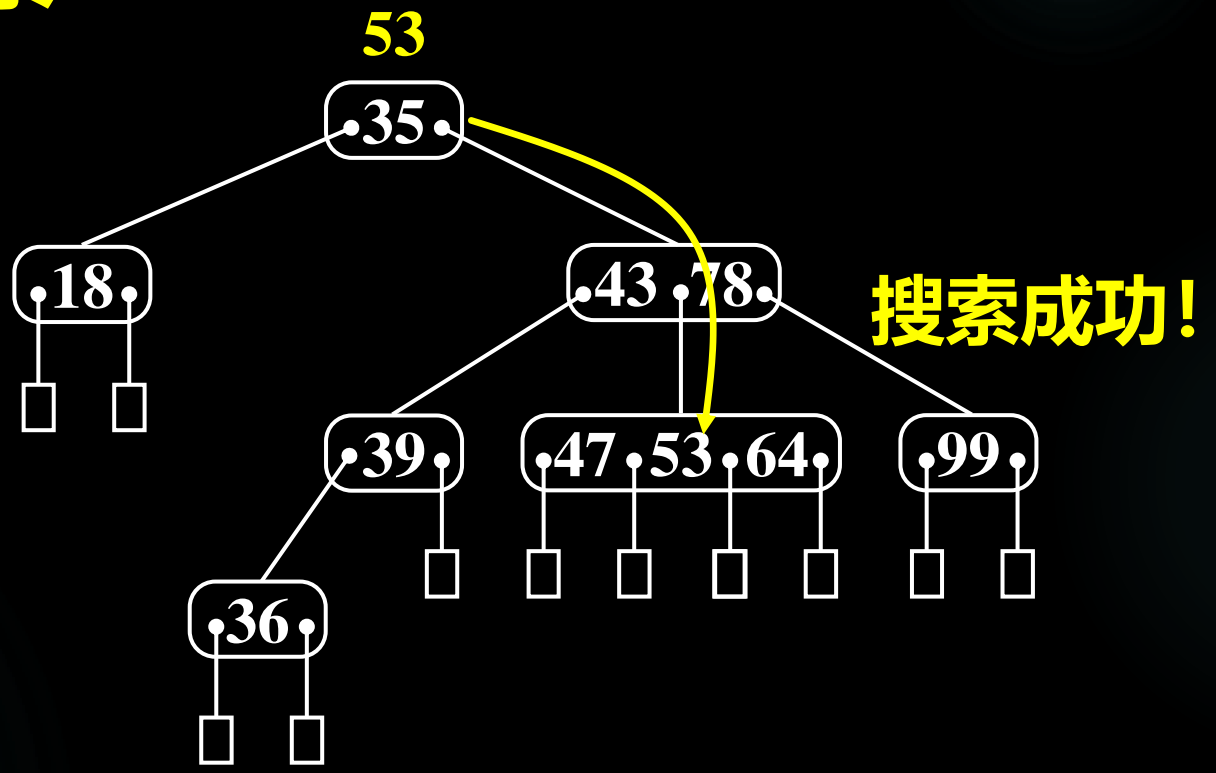


图 7.20　二叉搜索树的分页

二叉树被分成许多包含7个元素的页。每次从磁盘存取一页（而不是一个记录），即7个记录，从而使读取磁盘的次数减少到原来的三分之一，大大提高了搜索速度。

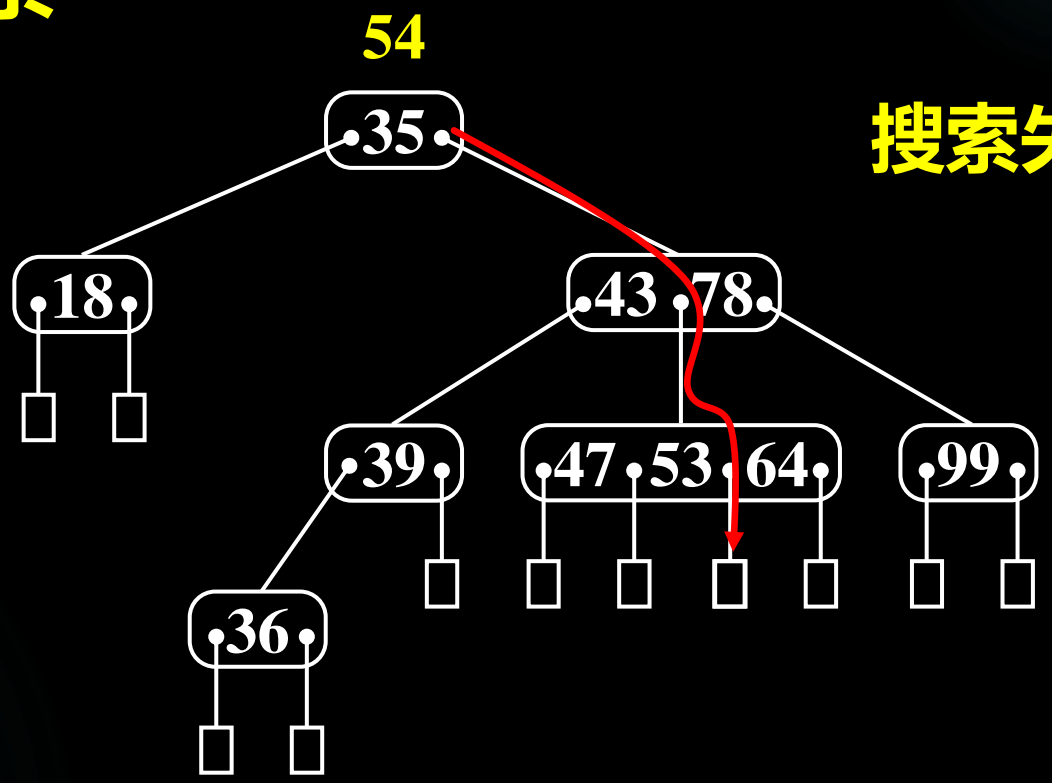
m叉搜索树的搜索



搜索成功！

搜索53

m叉搜索树的搜索



搜索失败!

搜索54

m叉搜索树的性质

设一棵m叉搜索树的高度为h（h层都是非失败结点），则该树上最多的**结点数**目（不包括失败结点）为：

$$\sum_{i=1}^{h} m^{i-1} = 1 + m + \cdots + m^{h-1} = \frac{m^h - 1}{m - 1}$$

m叉搜索树中，每个结点最多有m-1个元素，因此m叉搜索树中最多有m^h-1个**元素**。

m叉搜索树中结点与元素不是1: 1

B-树的定义

定义 一棵m阶B-树是一棵m叉搜索树，它或者是空B-树，或者是满足下列特性的树：

(1) 根结点至少有两个孩子

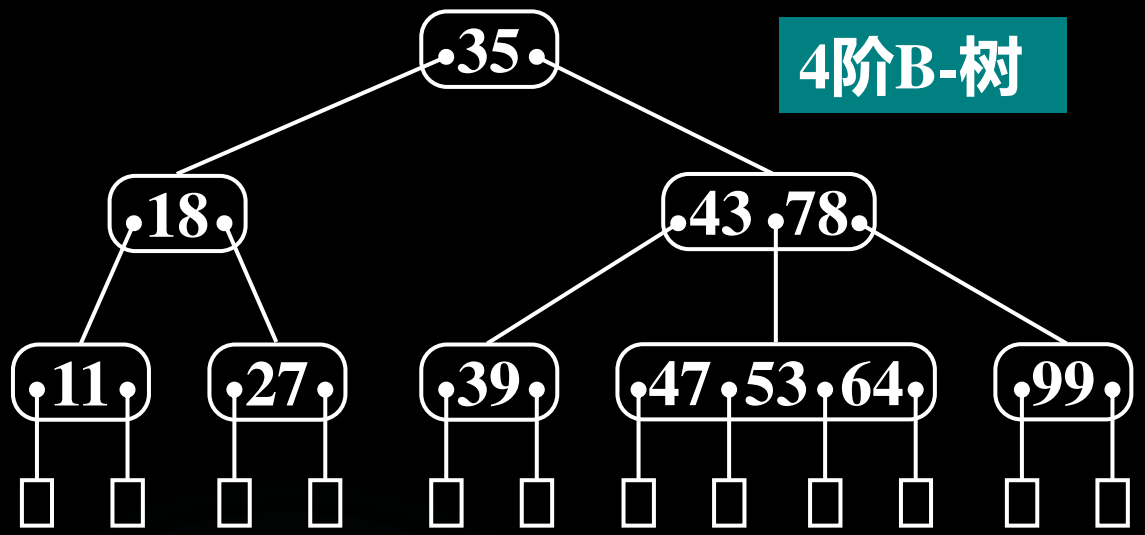
根结点可以只有一个元素

(2) 除根结点和失败结点外的所有结点至少有$\lceil m/2 \rceil$个孩子。

(3) 所有失败结点均在同一层上。

考虑均衡性

其他结点可能不允许只有一个元素，比如6阶-B树



4阶B-树

拿到一个B-树，进行检查工作

(1)首先看m是多少

(2)计算⌈m/2⌉是多少

(3)查看每个结点（根结点除外）的孩子数量有没有少于⌈m/2⌉，或超过m

(4)查看每个结点的关键字数量是否是孩子数量-1

从定义中可以得到，一棵m阶B-树中

(1)一个结点**最多**有m个孩子,**m-1个关键字**

(2)除根结点和失败结点外每个结点**最少**有⌈m/2⌉个孩子，**⌈m/2⌉-1个关键字**

(3)根结点最少有2个孩子

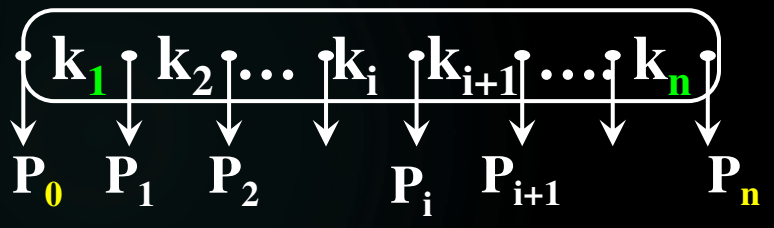
(4) 所有失败结点均在同一层上，**失败结点的双亲是叶子结点**

思考：为什么要限制孩子数量不能太少？

B-树的性质

性质 设B-树失败结点的总数是s，那么，一棵B-树包含的**元素总数N**是B-树的失败结点的总数s减一，即

$$N = s - 1$$



一个结点的指针数量为x，则其保存元素数量为x-1

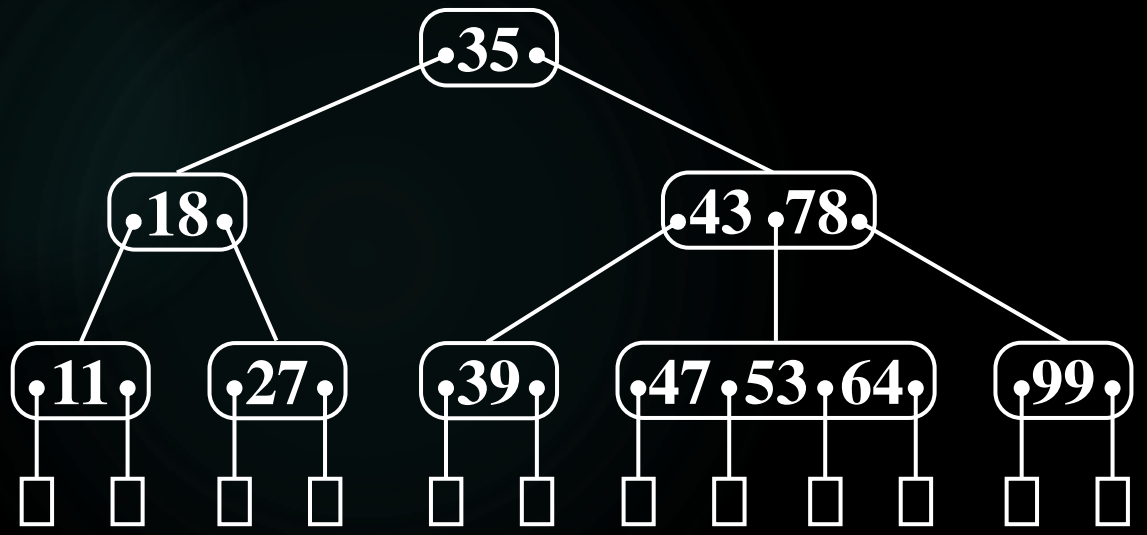
求解：指针总数为t，则非失败的n个结点的元素数量N是多少？

$$N=t-n$$

B-树的高度

性质 设B-树失败结点的总数是s，那么，一棵B-树包含的**元素总数N**是B-树的失败结点的总数s减一，即

$$N = s - 1$$



指针总数t、失败结点个数s，非失败结点数n之间有什么关系？

$$t=n+s-1$$

B-树的高度

性质 设B-树失败结点的总数是s，那么，一棵B-树包含的**元素总数N**是B-树的失败结点的总数s减一，即

$$N = s - 1$$

已知：

N=t-n

t=n+s-1

$N = s - 1$

定理 N个元素的m阶B-树的高度h有 $h \le 1+\log_{\lceil m/2 \rceil}(\frac{N+1}{2})$

证明：

设m阶B-树有N个元素，则失败结点的个数为N+1。

第1层为根结点，根结点至少有2个孩子

第2层至少有2个结点，每个节点至少有$\lceil m/2 \rceil$个孩子

第3层至少$2\times\lceil m/2 \rceil$个结点，每个结点至少有$\lceil m/2 \rceil$个孩子

第4层至少$2\times\lceil m/2 \rceil\times\lceil m/2 \rceil$个结点，每个结点至少有$\lceil m/2 \rceil$个孩子

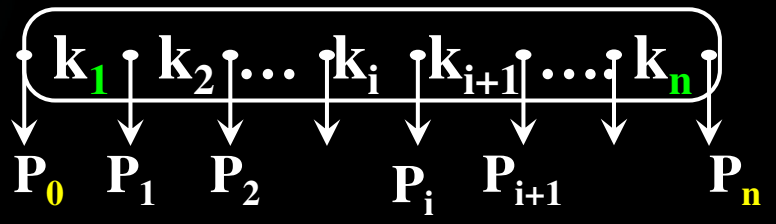
第h+1层至少$2\times(\lceil m/2 \rceil)^{h-1}$个结点，该层全是失败结点

$$N+1 \ge 2\times(\lceil m/2 \rceil)^{h-1}$$

B-树的搜索算法与m叉搜索树的搜索算法相同。但B-树搜索需执行的磁盘访问次数最多是

$1+\log_{\lceil m/2 \rceil}((N+1)/2)$

B-树的每个结点可以看成一个有序表，在一个B-树结点中搜索时在内存中的搜索，因此可以采用顺序搜索和二分搜索等内搜索算法进行。



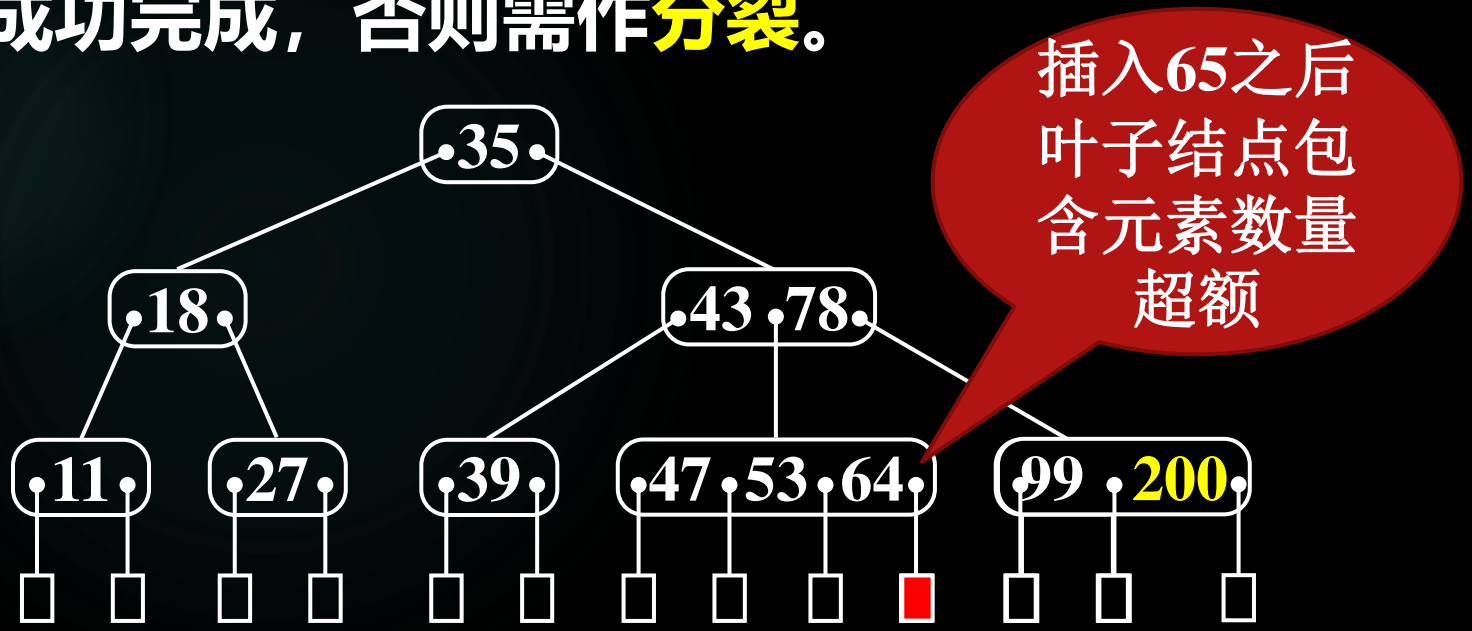
B-树的插入

将一个元素插入B-树的步骤：

（1）查重

（2）查重失败后停止在失败结点处，将新元素插在该**失败结点的上一层叶子结点**中

（3）如果插入后，该叶子结点中包含的元素个数不超过m-1，则插入成功完成，否则需作**分裂**。



4阶-B树

(1) 插入200

(2) 插入65

具体看看插入操作的各种情况及其处理

例1 在图7.23的4阶B-树中插入新的元素59。

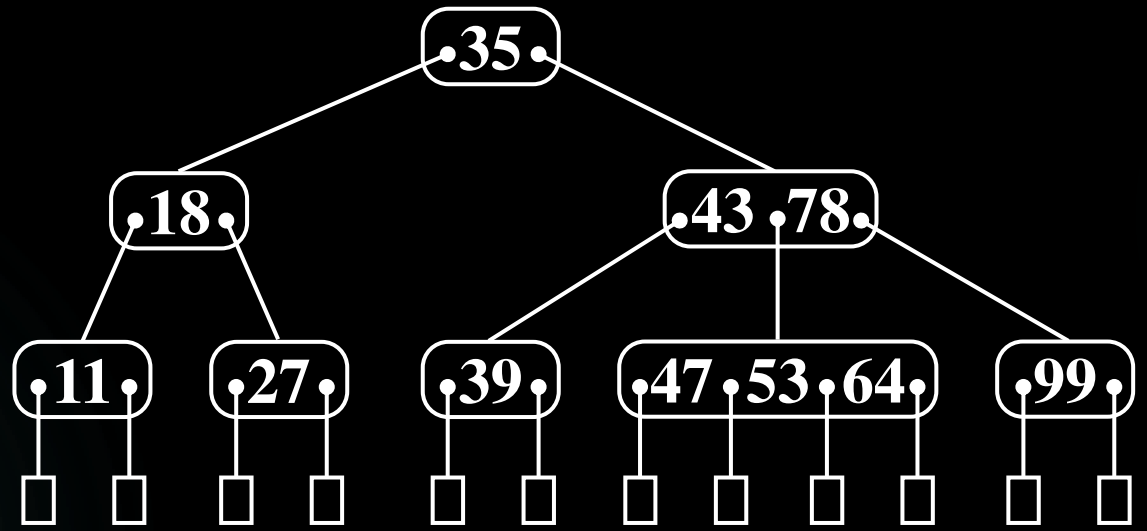
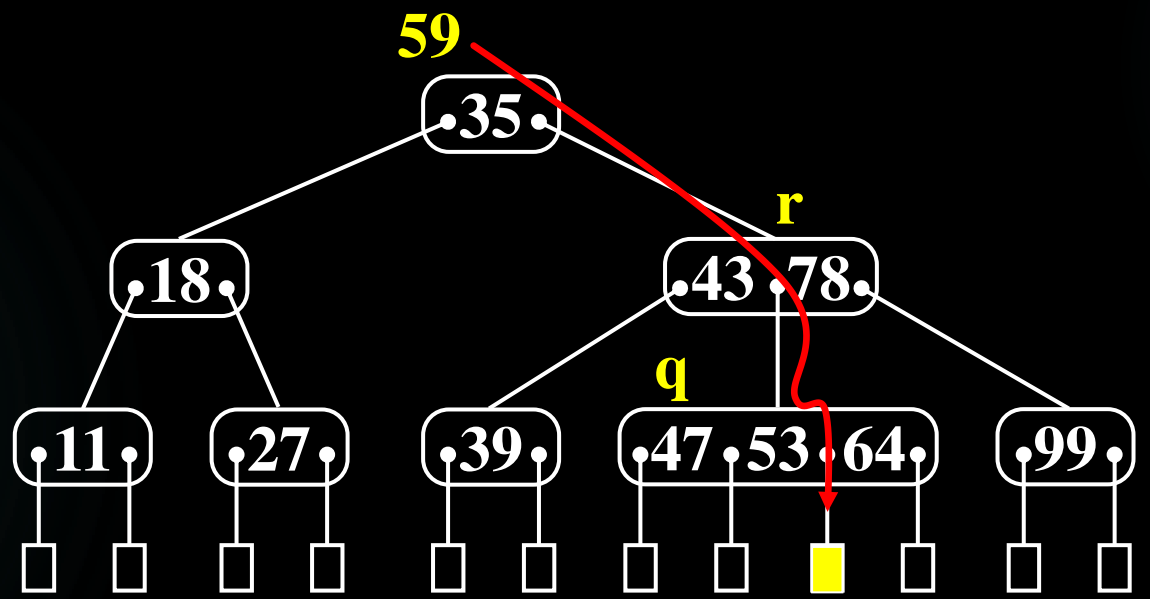


图7-23 4阶B-树

步骤1：搜索新元素

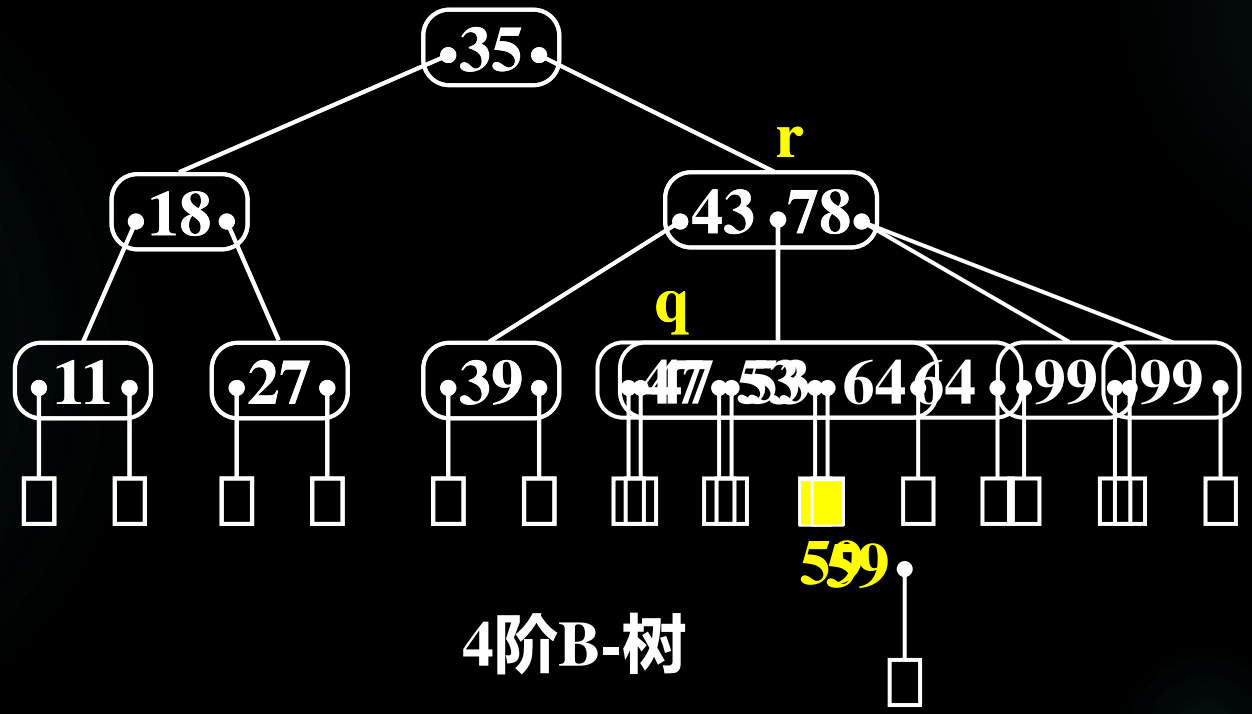
在B-树中搜索元素59。搜索失败处是图中黄色的失败结点。
q是失败结点的上一层叶子结点。
r是q的双亲结点。



4阶B-树

步骤2：将新元素和一个空指针插入到q中

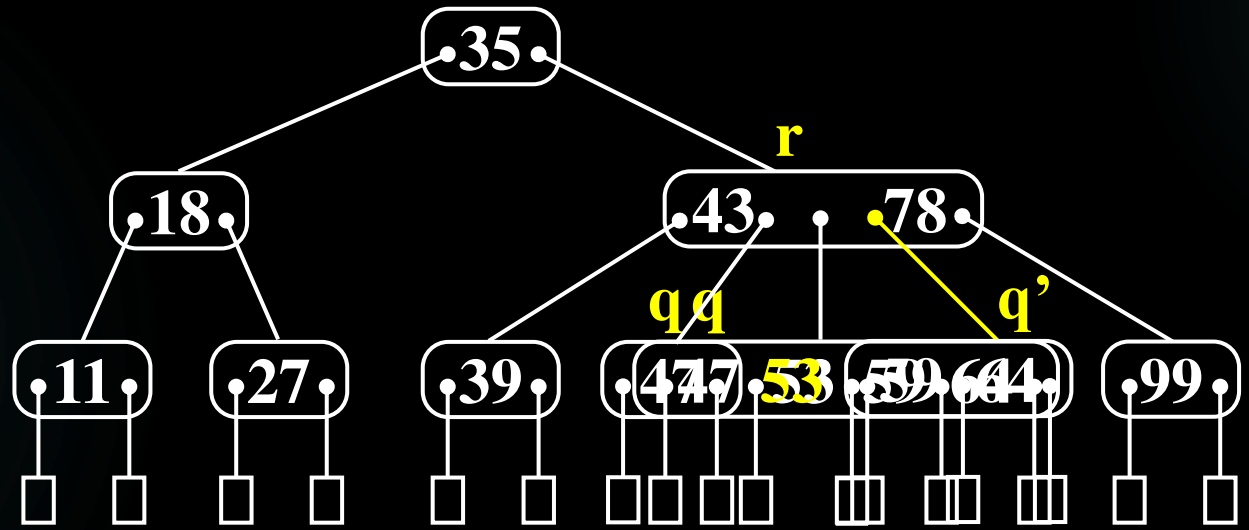
q中插入新元素和失败结点后，如果q没有溢出，即结点中包含的元素个数未超过m-1（指针数没有超过m），则插入运算结束，否则进行步骤3分裂操作。



4阶B-树

步骤3：分裂

分裂发生在q中第$\lceil m/2 \rceil$个元素的位置，结点q被一分为三，即q、$k_{\lceil m/2 \rceil}$和q’。 q中保存前$\lceil m/2 \rceil$-1个元素，q’中保存后$\lceil m/2 \rceil$个元素。多出的$k_{\lceil m/2 \rceil}$和q’一起插入其到双亲结点r中。



4阶B-树

例2 在3阶B-树中插入新元素53

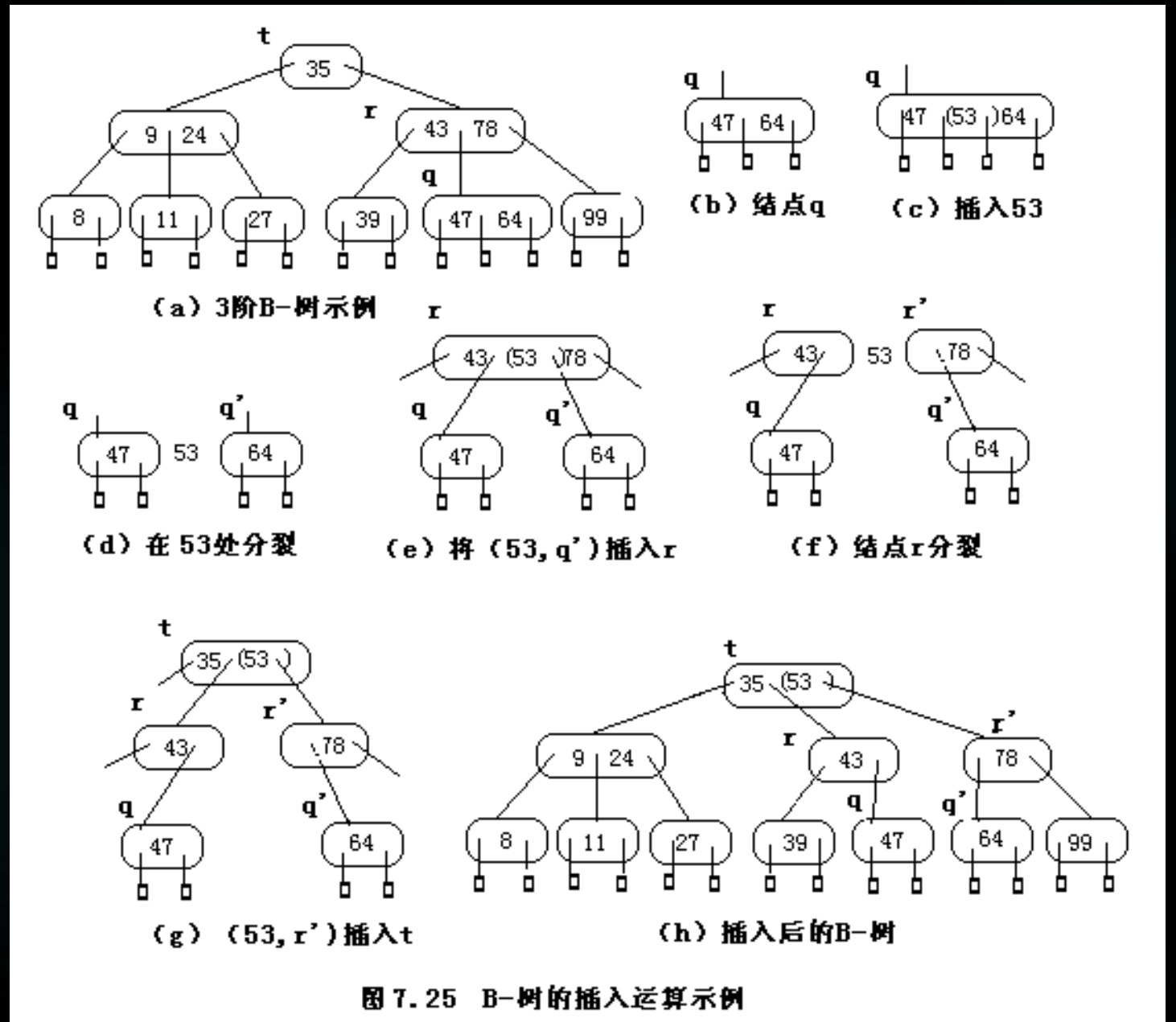


图 7.25 B-树的插入运算示例

集合关键字为{68,54,82,35,75,90,103,22}
构造一棵3-阶B树

B-树插入新元素x的方法。

(1) 搜索x，将x和一个空指针插入搜索失败结点上一层的叶子结点q中

(2) 检查结点q是否溢出，即结点中包含的元素个数未超过m-1（指针数未超过m），未溢出则插入运算成功终止

(3) 若q溢出，则必须进行结点的分裂操作，以⌈m/2⌉处的元素为分割点，将结点一分为三：

（q中前⌈m/2⌉-1个元素）|（⌈m/2⌉处的元素）|（q中后⌈m/2⌉个元素）

将⌈m/2⌉处的元素设为y，插入双亲结点r中

（q中前⌈m/2⌉-1个元素）组成的结点成为y的左侧孩子

（q中后⌈m/2⌉个元素）组成的结点成为y的右侧孩子

(4) 检查双亲结点r是否溢出，如果溢出，则对r继续进行步骤(3)分裂，直到分裂操作不再造成任何结点发生溢出

(5) 如果在步骤(3)中溢出结点为根结点，则生成新的根结点存放y

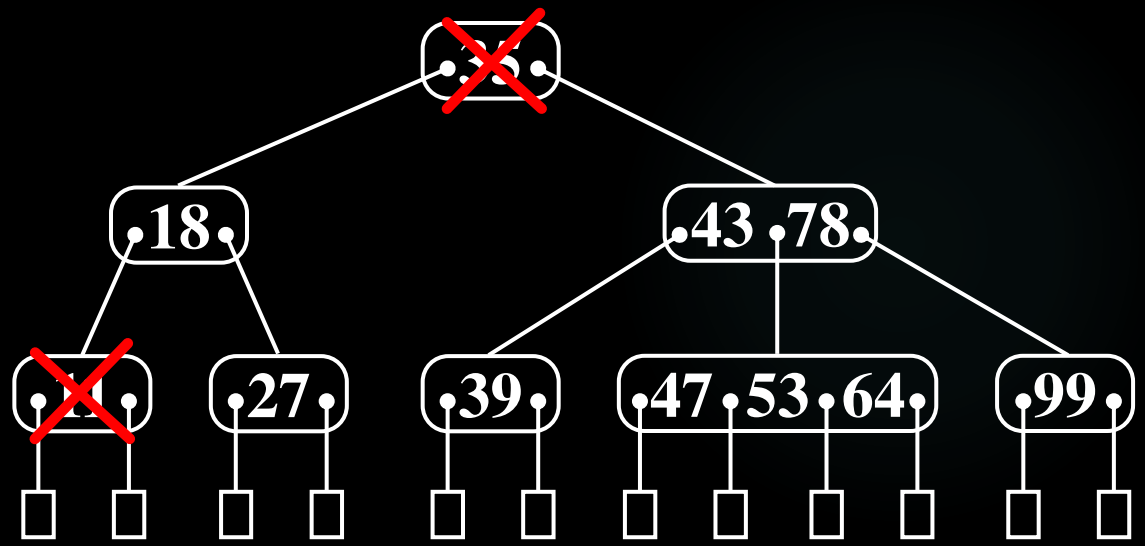
集合关键字为 {75,90,103,22,68,54,82,35}
构造一棵3-阶B树

B-树的删除

B-树删除元素，分情况：

a.元素在叶结点上

b.元素不在叶结点上



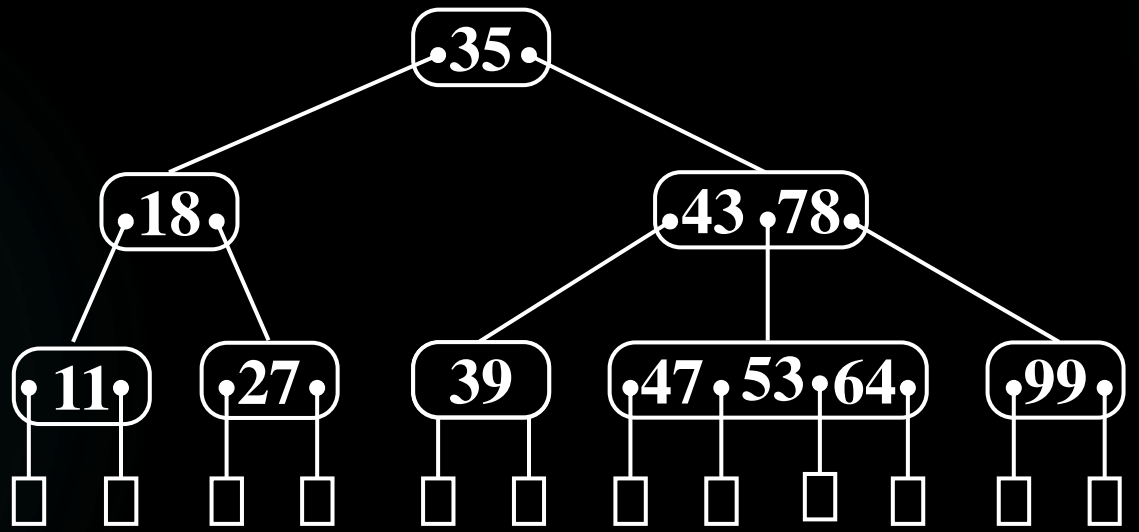
a.元素x在叶结点上

(1) 删除元素x和一个空指针

(2) 检查是否发生下溢，即该叶节点元素个数少于⌈m/2⌉-1

(3) 处理下溢

删除53

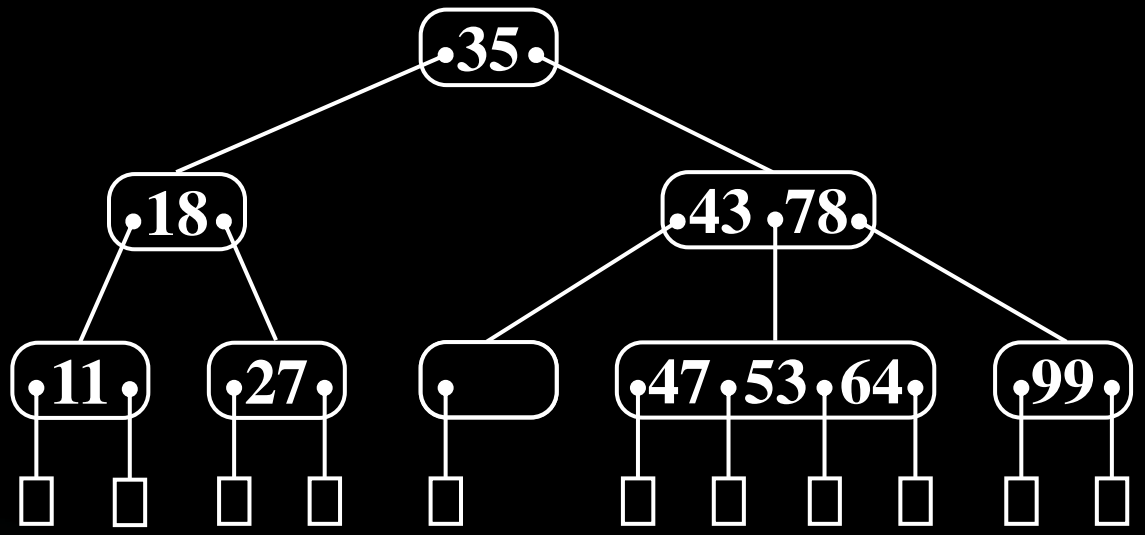


4阶B-树

没有发生下溢，删除成功

删除39

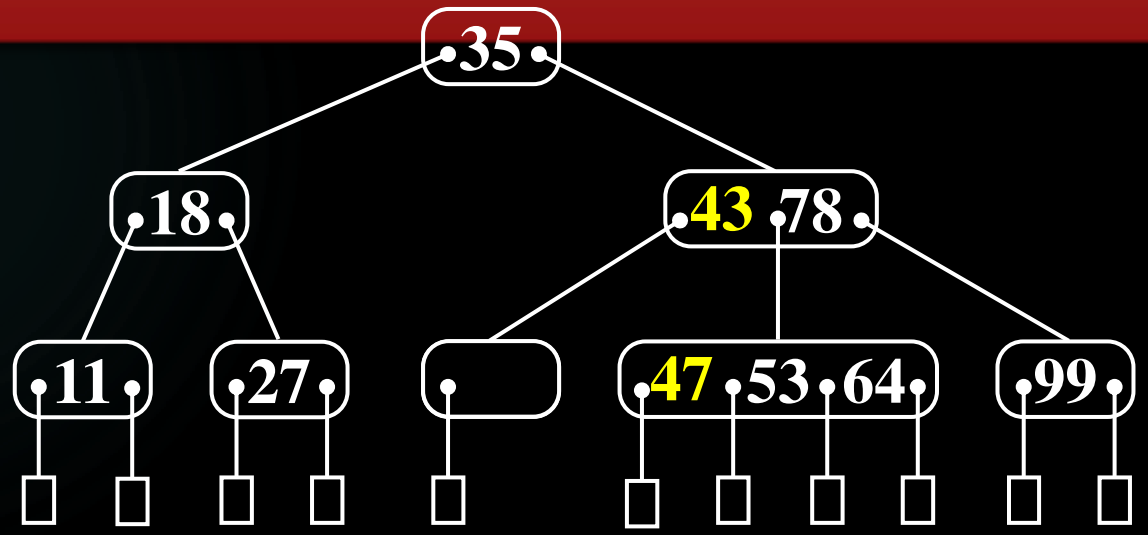
4阶B-树



发生下溢

借

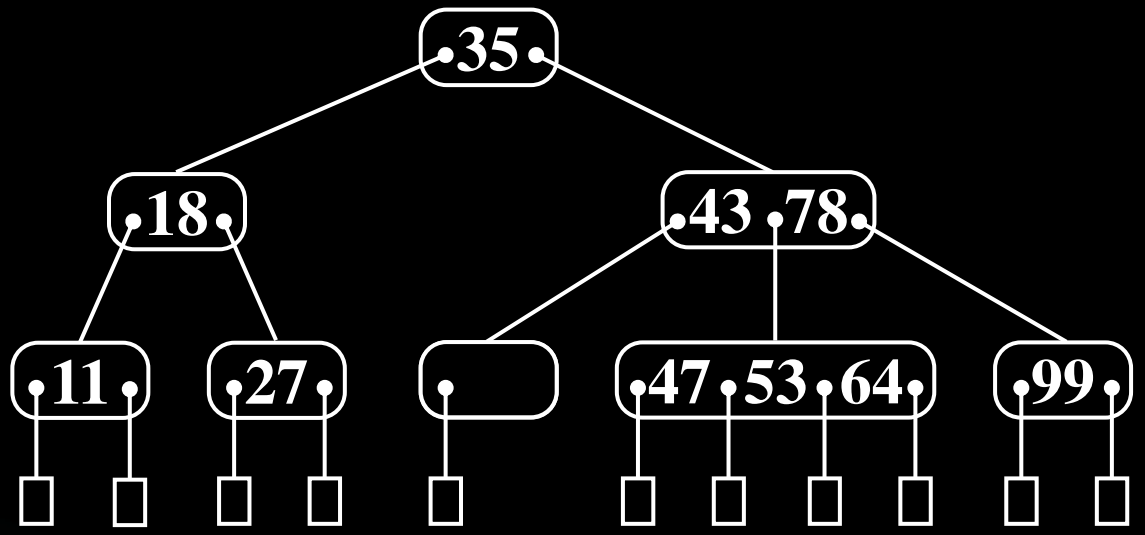
富余兄弟借出一个最靠近贫困兄弟的元素给双亲，将一个最靠近贫困兄弟的指针借给贫困兄弟，双亲将最靠近贫困孩子的元素借给贫困孩子



删除成功

删除39

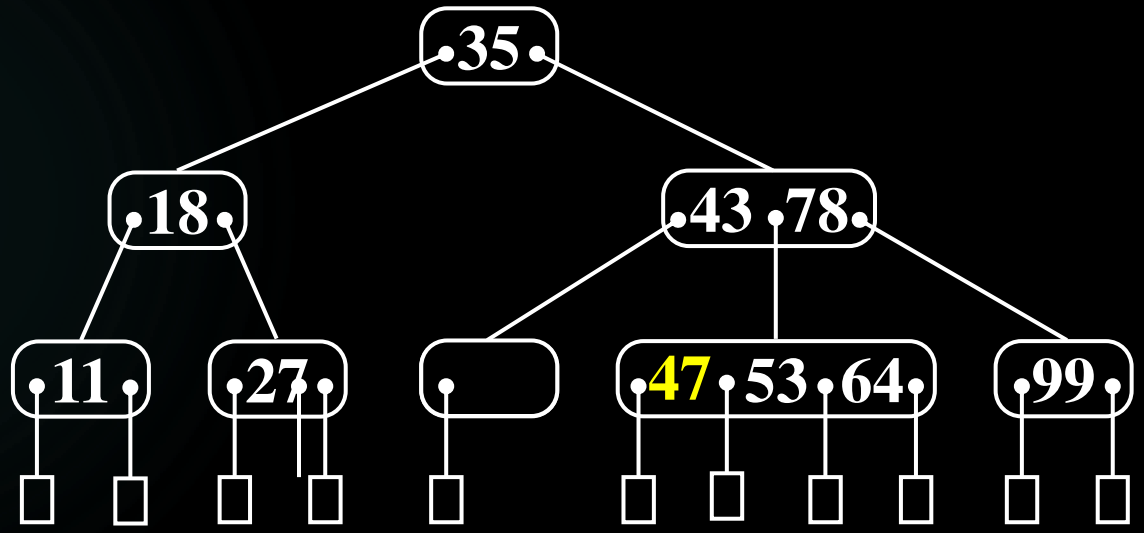
4阶B-树



发生下溢

借

富余的左右兄弟借出一个元素，先左后右



一个错误！！！

直接借造成失序！

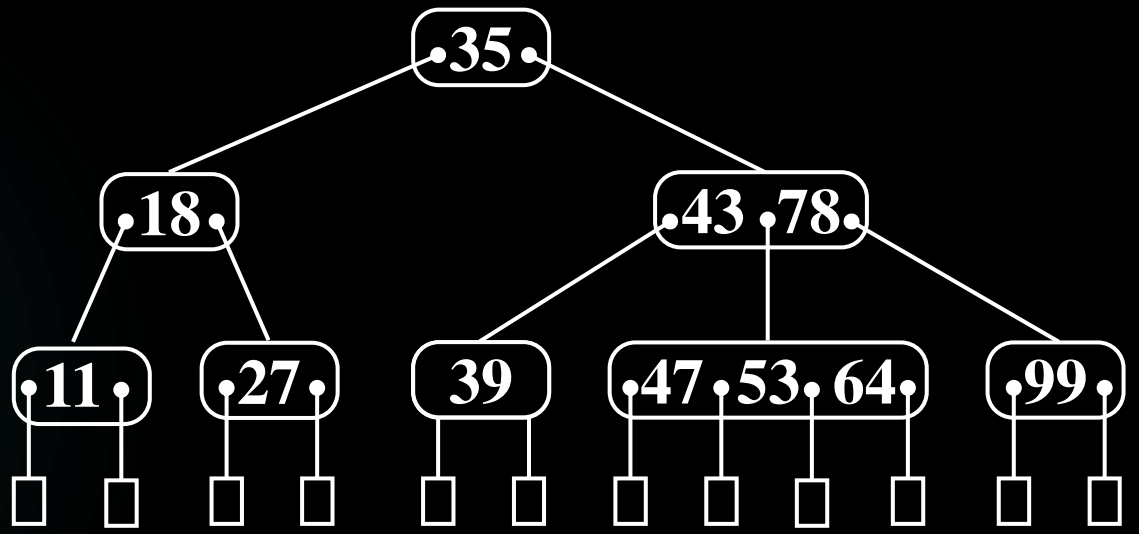
a.元素x在叶结点上

(1) 删除元素x和一个空指针

(2) 检查是否发生下溢，即该叶节点元素个数少于⌈m/2⌉-1

(3) 处理下溢

删除11



4阶B-树

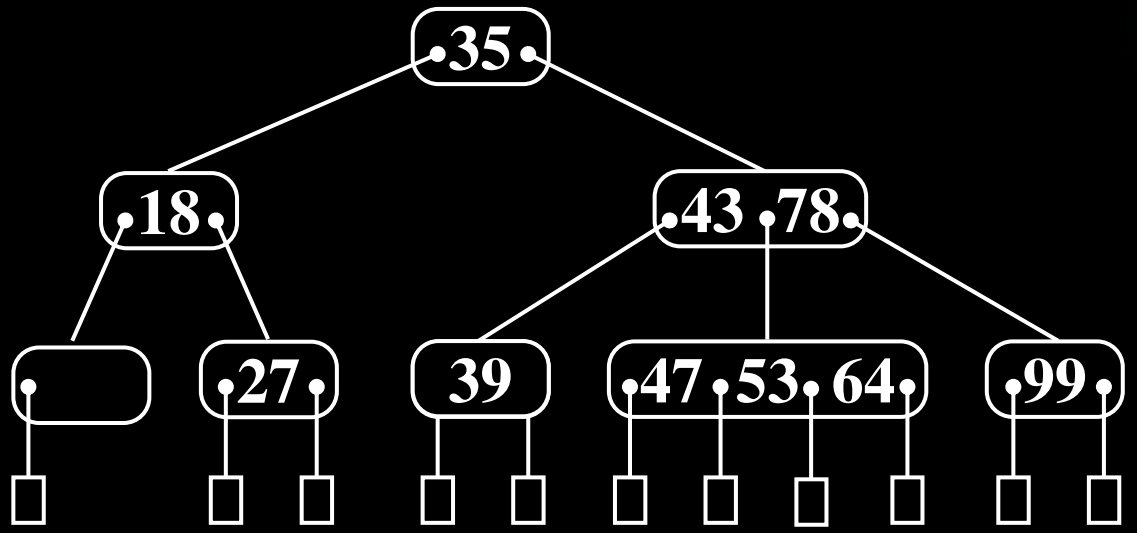
发生下溢

借

没有富余的兄弟

删除11

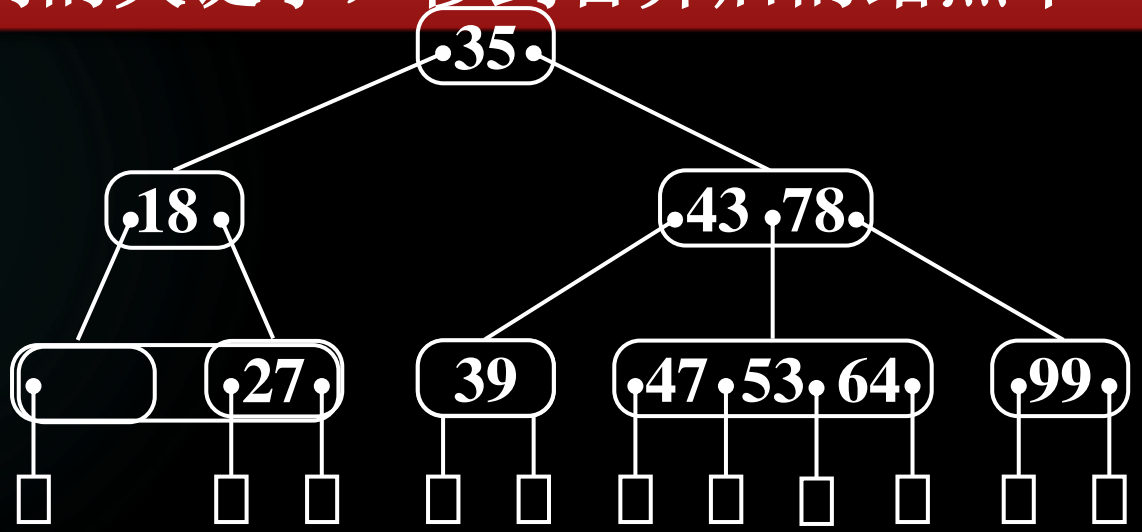
4阶B-树



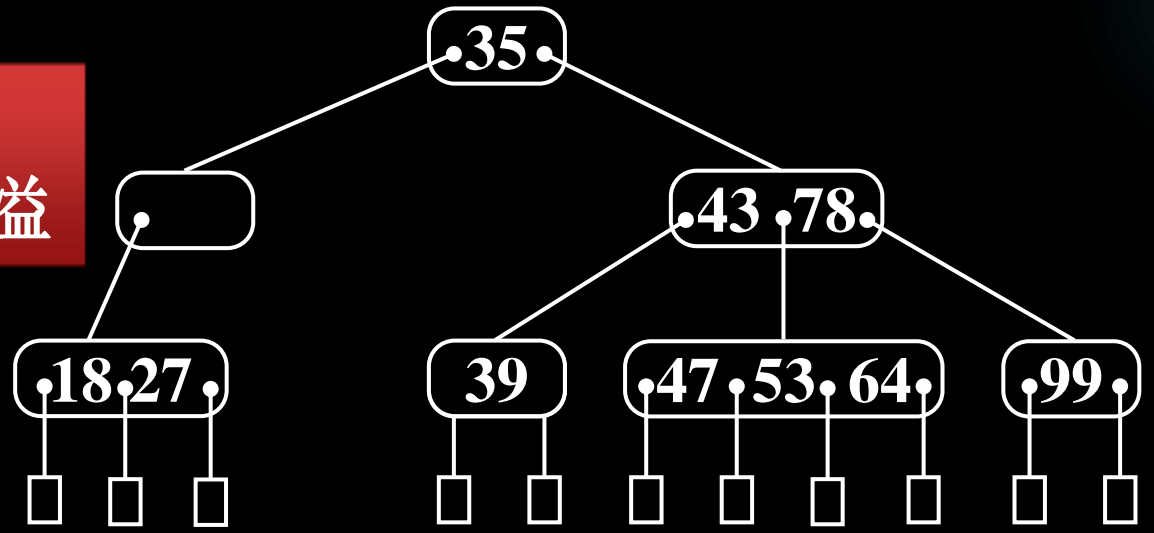
发生下溢，且无富余兄弟

合并

与左右兄弟之一合并，双亲结点的一个元素（夹在合并的左右子树之间的关键字）移到合并后的结点中



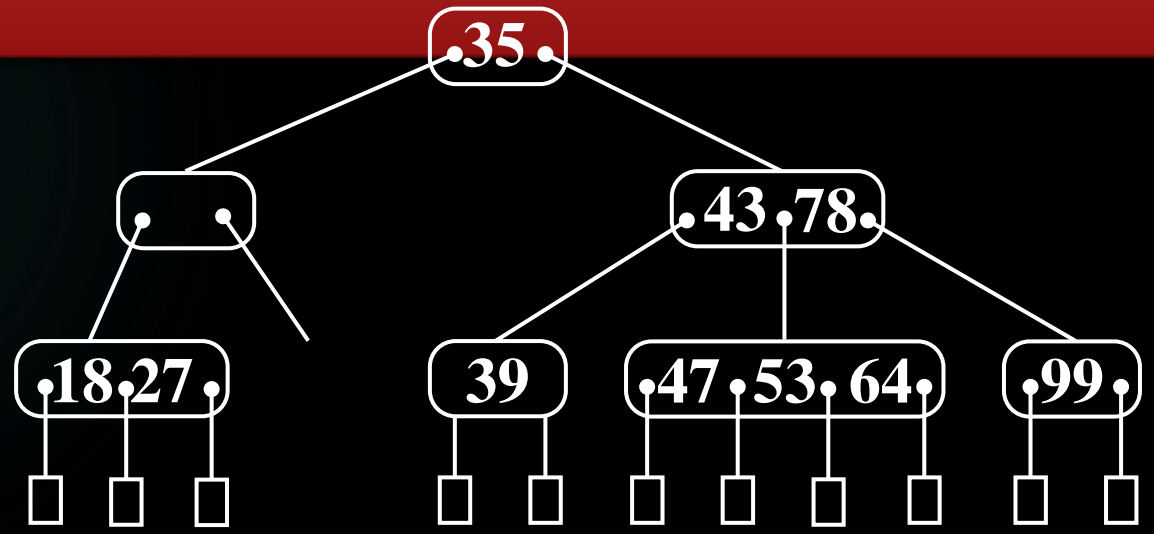
一定要检查
双亲结点是否下溢



发生下溢

借

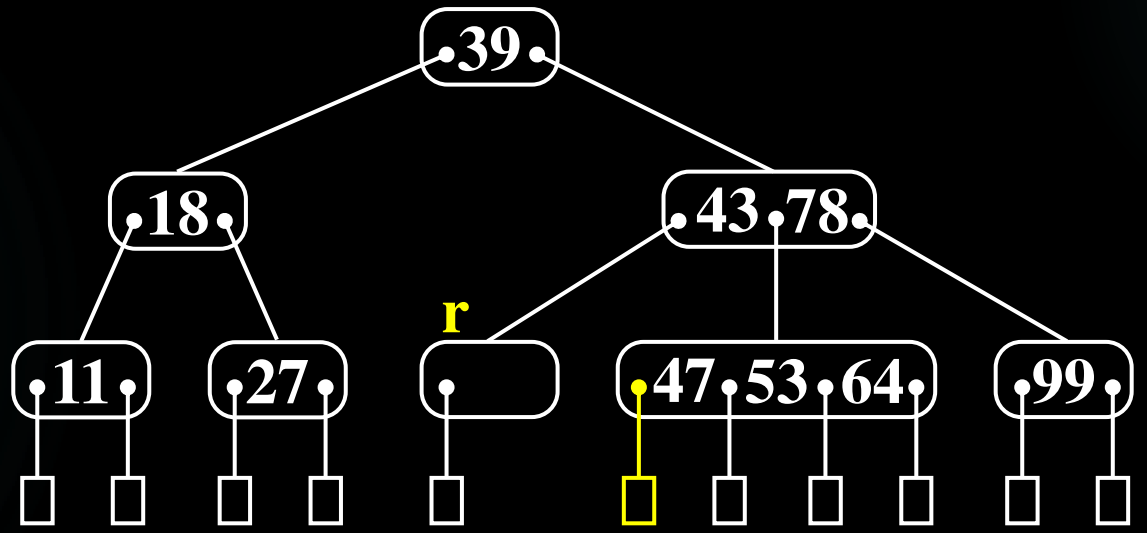
富余兄弟借出一个最靠近贫困兄弟的元素给双亲，将一个最靠近贫困兄弟的指针借给贫困兄弟，双亲将最靠近贫困孩子的元素借给贫困孩子



思考：为什么要限制孩子数量不能太少？
为什么要限制孩子数量下限为$\lceil m/2 \rceil$？

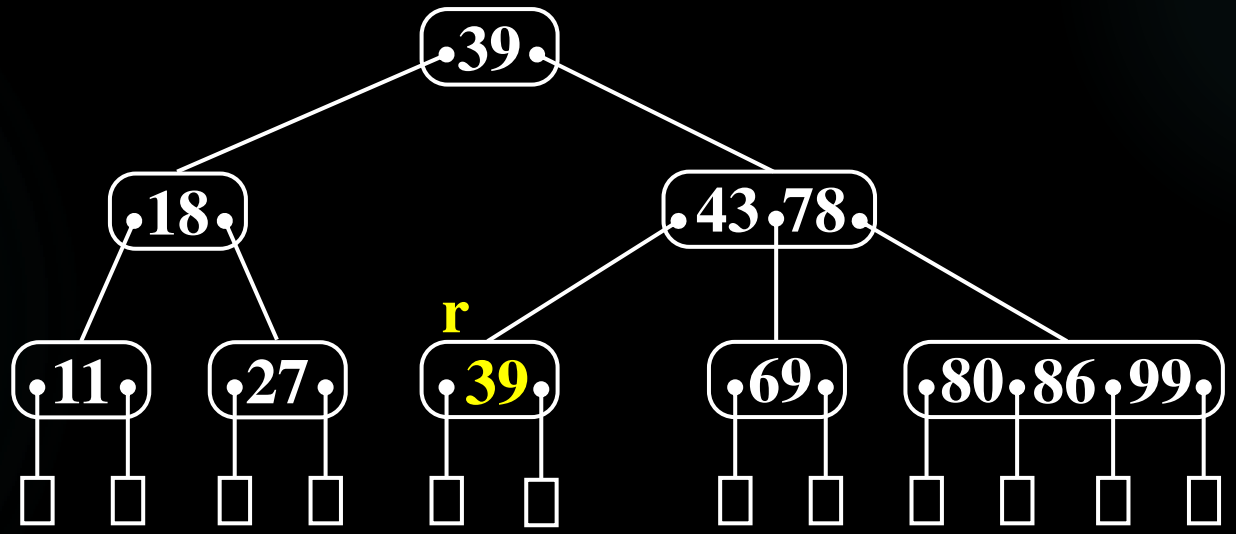
错误的“借”法：

位于2个移动元素间的指针没有跟着移动；



错误的“借”法：

传递地借元素；



b.元素x不在叶结点上

替代

由x的右子树上的最小元素（最靠左的叶子上的第一个元素）取代之

删除用来代替的最小元素---问题转化为删除元素在叶子结点上

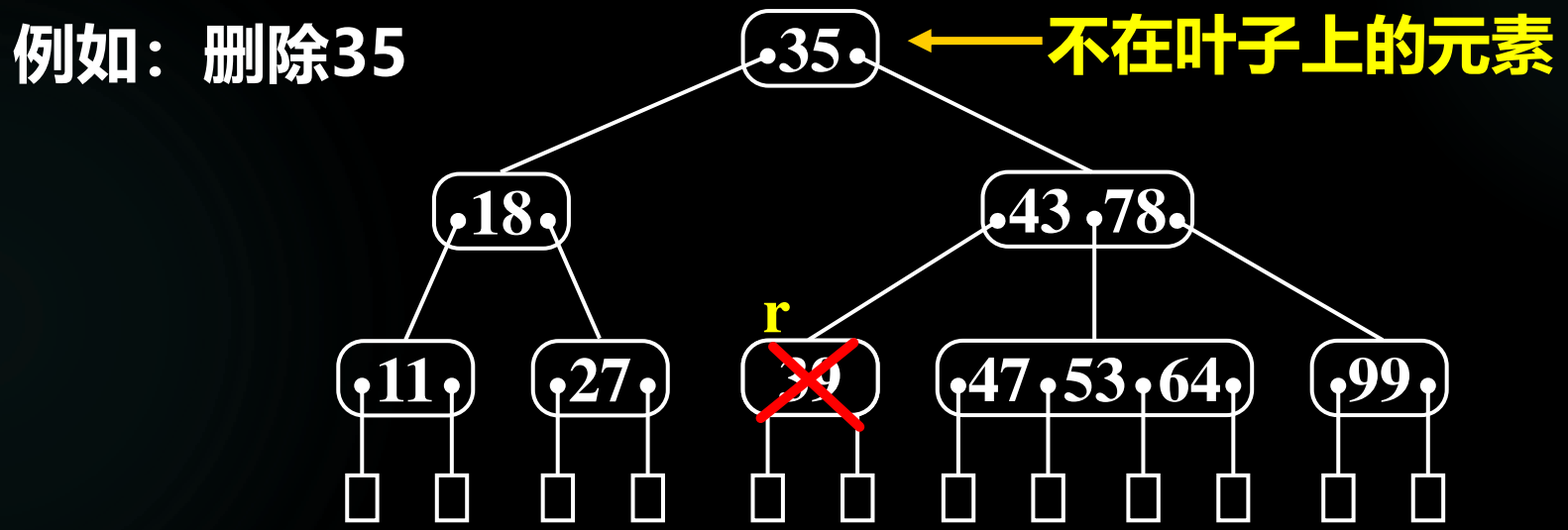


图7-23 4阶B-树

B-树删除元素的方法：

(1) 搜索被删除的元素，判断被删除的元素的位置

(2) 被删除元素在叶子结点中，则从该叶子结点中删除该元素

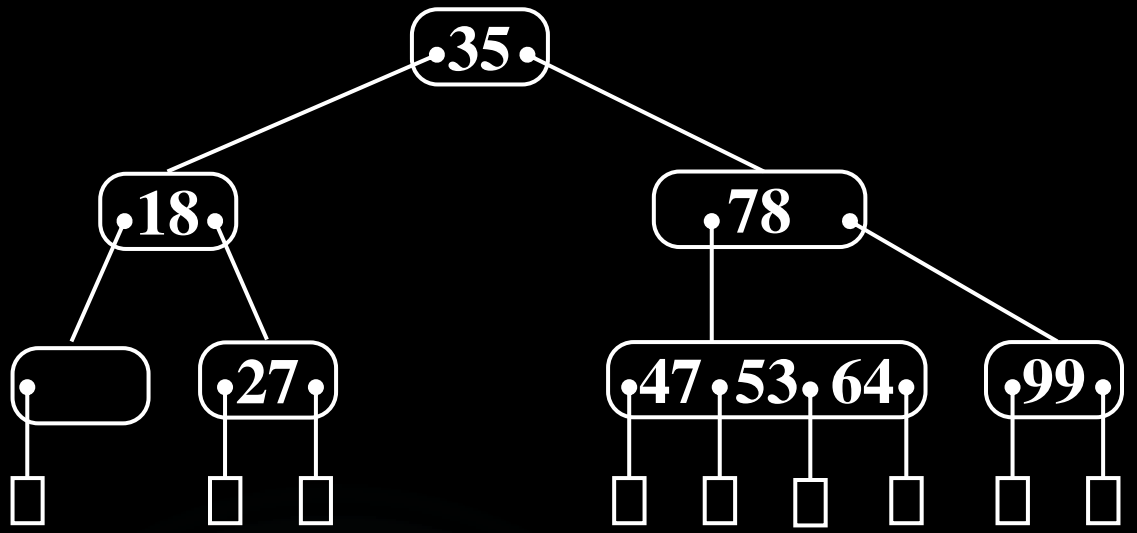
(3) 如果被删除的元素不在叶子结点中，那么用它的右侧子树上的最小元素取代之（必定在叶子结点中），然后从叶子结点中删除该替代元素。

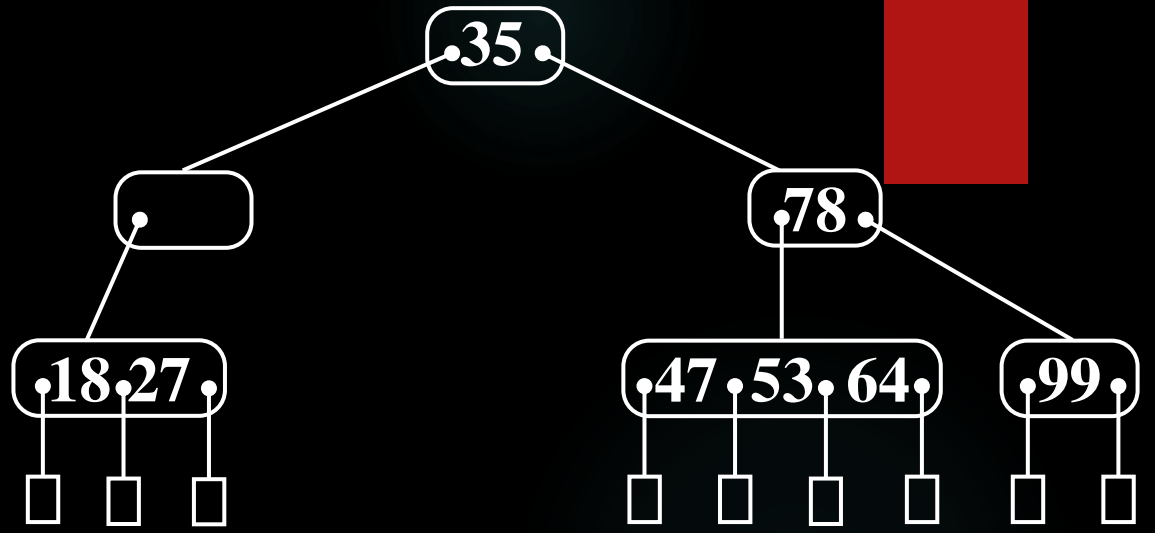
(4) 如果删除元素后，当前结点中包含至少$\lceil m/2 \rceil - 1$个元素，删除运算成功结束。

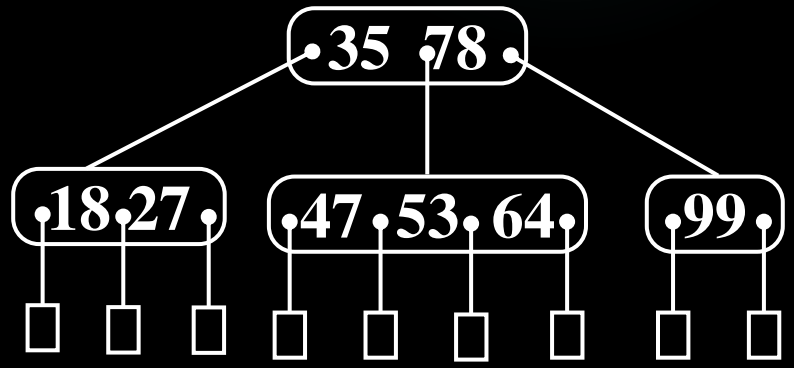
(5) 如果删除元素后，发生下溢。处理的方法首先是借元素：如果其左侧兄弟有至少$\lceil m/2 \rceil$个元素，则可以向其左兄弟“借”一个元素；否则如果其有侧兄弟有多余元素，则向其右侧兄弟借。

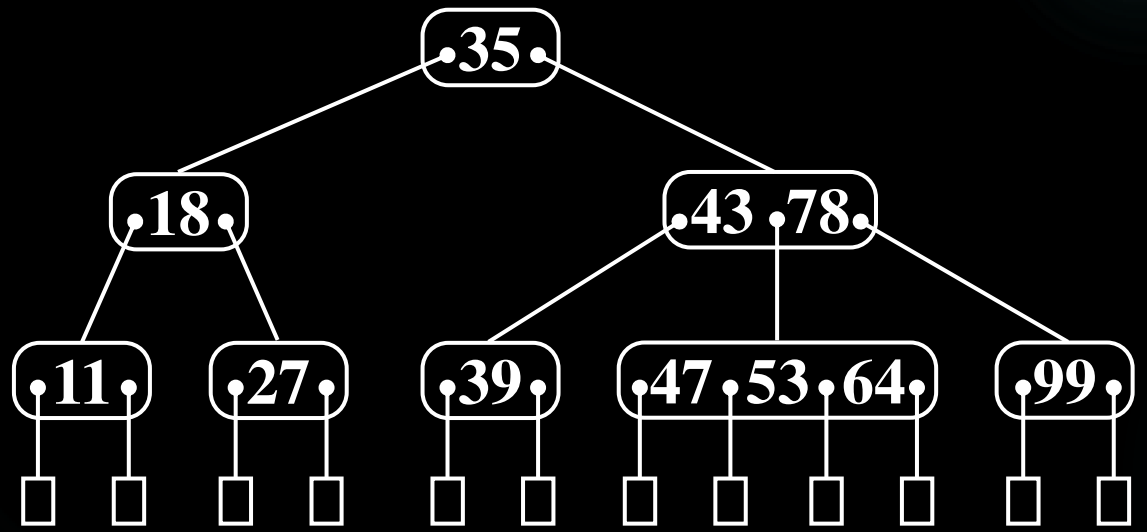
(6) 如果删除元素后，当前结点产生下溢，且左右两侧兄弟结点都只有⌈m/2⌉-1个元素，则只能进行“合并”。

(7) 如果由于合并操作，导致根结点中的一个元素被删除，并且该结点只包含一个元素，则其中的元素被删除后，根结点成为不包含任何元素的空结点，那么刚刚合并的那个结点将成为B-树新的根结点，这时B-树变矮。


35
18
78
27
47
53
64
99

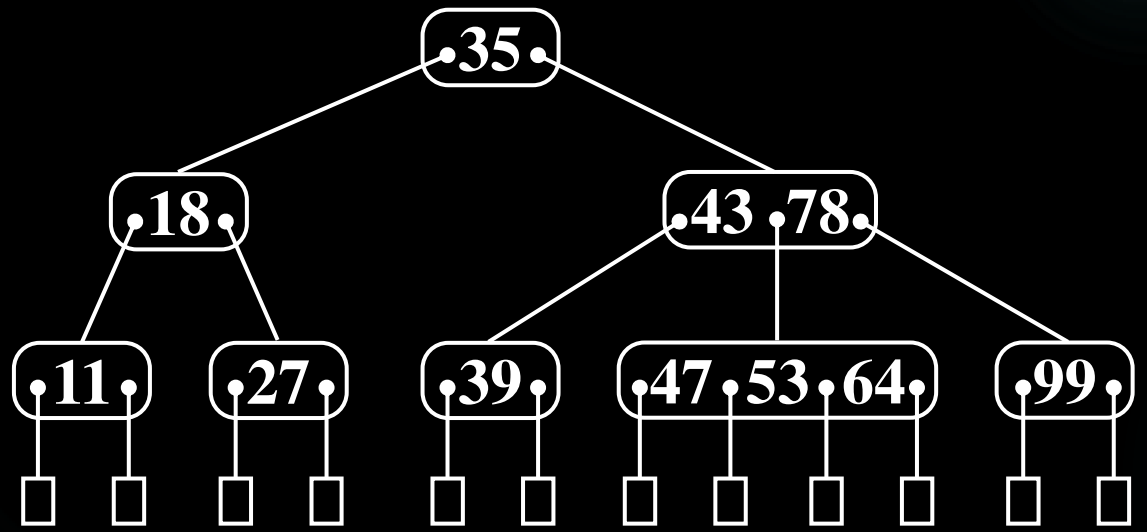

35
78
18
27
47
53
64
99


35
78
18
27
47
53
64
99



依次删除4阶-B树元素关键字：

64,99,78,43,35,27,18



依次删除4阶-B树元素关键字：

18,27,11,35,78,53

散列表

引　言

集合在线性表和树表的表示中，元素在结构中的**位置与元素的关键字间**无直接确定关系，搜索时需通过关键字的一系列比较完成。

散列表，在元素的关键字与其在结构中的位置建立直接联系，以实现直接快速搜索。

目录

- 介绍散列技术
- 常见散列函数
- 冲突解决方法

已经学习的组织集合元素的数据结构：

（1）线性表

（2）二叉排序树、B-树

新的组织集合元素的数据结构：**散列表**

- 使用线性表来存储集合元素
- 元素存储位置与关键字相关

在元素的存储位置和关键字值之间建立一种对应关系h，把关键字值映射到表中的某个位置，即：

集合元素的位置Loc(key) = h(key)

h(key)=(key+5)%10

不需要进行比较即可找到元素位置

0	1	2	3	4	5	6	7	8	9

散列函数(h)：元素的关键字(key)与其存储位置(loc)之间的关系函数

Loc(key)：表示关键字值为key的元素的存储地址。

散列表(hash,哈希表)：用散列函数建立起来的表。

散列表举例

建立31个省、自治区、直辖市的人口统计表。

- 散列表长度30　　关键字：名称的汉语拼音
- 编码：A-Z⇒01-26
- 两种散列函数h:

h_1(key)=首字母的编码

h_2(key)=首字母的编码+尾字母的编码,若和大于30,则减去30

key	BEIJING	JIANGSU	SHANGHAI	SICHUAN	JIANGXI	TIANJIN	SHANXI
h1(key)	02	10	19	19	10	20	19
h2(key)	09	01	28	03	19	04	28

key	BEIJING	JIANGSU	SHANGHAI	SICHUAN	JIANGXI	TIANJIN	SHANXI
h1(key)	02	10	19	19	10	20	19
h2(key)	09	01	28	03	19	04	28

结论一：

对h1: JIANGSHU与JIANGXI等被映射到相同位置，产生冲突

冲突：key1≠key2，但h(key1)=h(key2)的现象。

不允许发生冲突

同义词：对同一散列函数，具有相同h值的关键字。

结论二：

h2(key)冲突少于h1(key)

能否避免冲突？不现实

h(key)的值域应该在地址范围之类

例如：设计H(key)=各字母编码序列使key⇔h(key)
H(SHANXI)=190801142409

没有这么大的地址

分析上表：h_2(key)冲突少于h_1(key)，说明冲突与散列函数有关。
散列函数是一个压缩映象，冲突不可避免！
可以做到的是：
1、选择“好”的h,尽量减少冲突。
2、若发生冲突，如何处理？用冲突处理技术。

压缩映象：key的取值范围可以很广，但是h的取值个数一定不会超过地址空间大小。将大范围的key投射到小范围的h取值上。

“好”的散列函数：

⑴均分布，少冲突；

⑵计算简便，快速。

均分布：元素经散列函数映象到散列表中的任何位置是等概率的。下面介绍几种目前比较通用的散列函数。

1、除留余数法

h(key)=key mod M (M为散列表表长)
一般取不超过M的素数P会更好。

对于素数p, 0<a<p, 任意 0<=i<j<p, a×i 与 a×j 不同余, 即不冲突

例： M=1000, P=997

关键字	内部编码	散列地址
KEYA	11052501	756
KEYB	11052502	757
AKEY	01110502	864
BKEY	02110502	873

散列地址相近，可以先将关键字的内部编码循环移若干位后，再%。

2、平方取中法

$h(key)=(key)^2$的中间若干位k

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

例：n=765, k取3。

$10^{3-1}<765\leq 10^3$，故k=3, 即取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

3、折叠法

把关键字值自左到右，分成位数相等的几部分，每一部分位数与散列表地址的位数相同，只有最后一部分的位数可以短些。把这些部分的数据叠加起来，得到该关键字的散列地址。

（1）移位法：把各部分最后一位对齐相加

（2）分界法：沿各部分的分界来回折叠，然后对齐相加

例：设关键字key=123203241111220，散列地址取3位，则key被划分位5段：

123 203 241 112 20

123	123
203	302
241	241
112	211
＋ 20	＋ 20
699	897
(a)移位法	(b)分界法

适用于关键字很长的情况

4、数字分析法

取分布均匀的若干位。

例如，一组关键字如下：

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

通过分析，可以看出第4、5、6位分布均匀，故取第4、5、6位这三位为散列函数的值。

冲突是不可避免的。当冲突发生时，必须对冲突进行处理。

处理冲突的方法有：

（1）拉链法

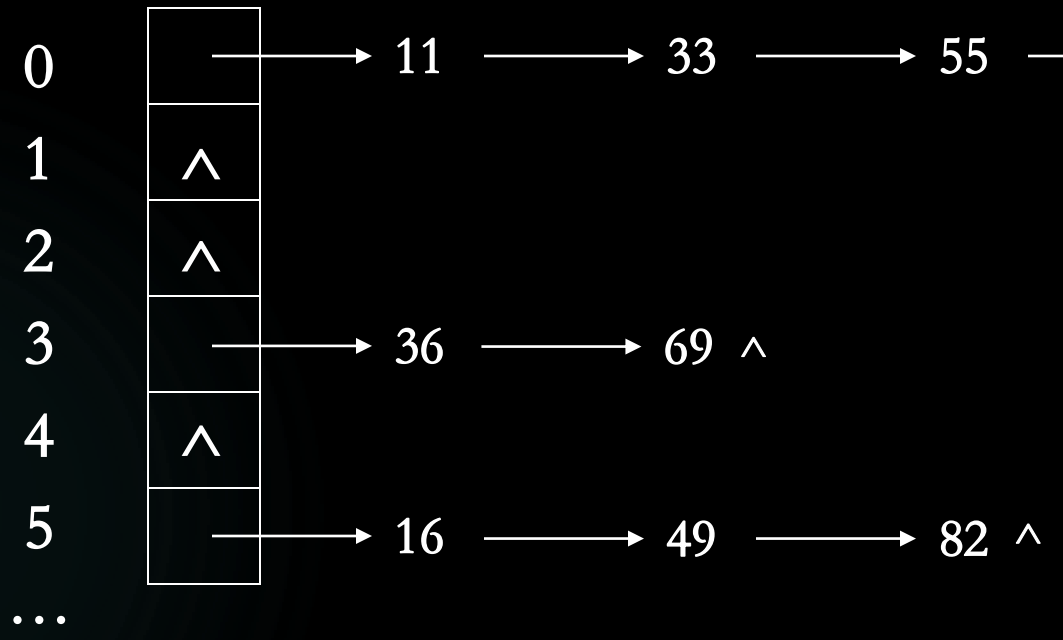
（2）线性探查法

（3）二次探查法

（4）双散列法

拉链法

将所有具有同义词的关键字建立一个单链表，单链表可以按升序排列。所有单链表的头指针存入一长度为M的数组中。



散列函数采用除留余数法，除数取11。H(key)=key % 11

有n个元素的散列表，其每个单链表的平均长度为n/M。

开地址法

对散列表插入新元素的方法：
从h(key)开始，按照某种规定的次序探查允许插入新元素的空位置。

基位置：h(key)　首次探索的地址

探查序列：如果h(key)已经被占用，需要有一种解决冲突的策略来确定如何探查下一个空位置

不同的探查策略，产生不同的探索路径，依次被探索的位置称为探查序列。

根据生成探查序列的规则的不同，开地址法有：

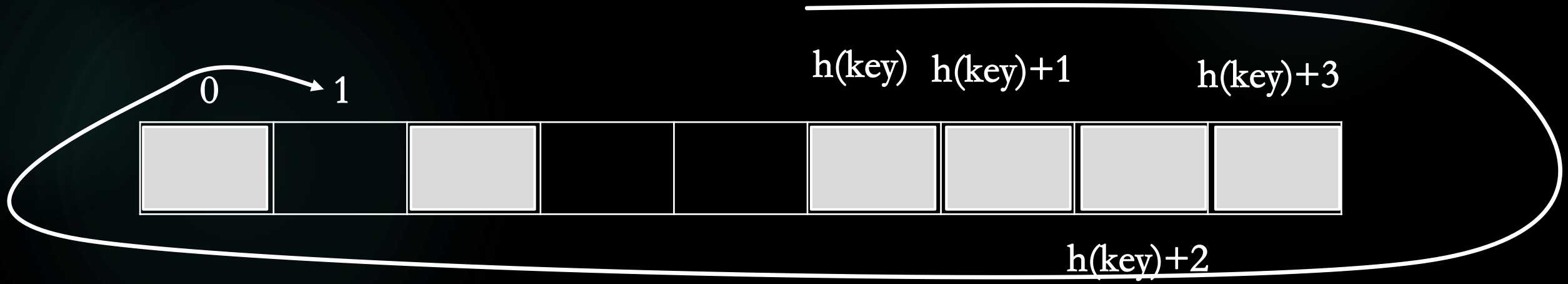
- 1、线性探查法
- 2、二次探查法
- 3、双散列法

1、线性探查法

线性探查法使用的探测序列如下：

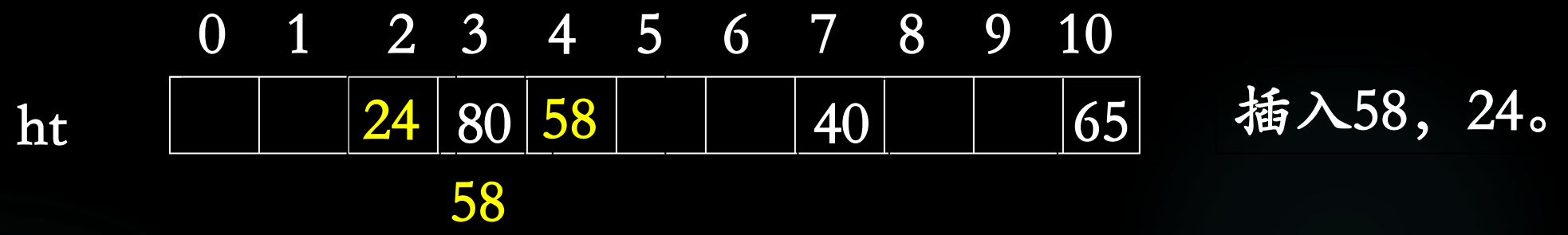
h(key),h(key)+1, h(key)+2 …,*M-1,0*,1,…,h(key)-1

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个位置，直到某个位置为空时，将关键字插入该位置。

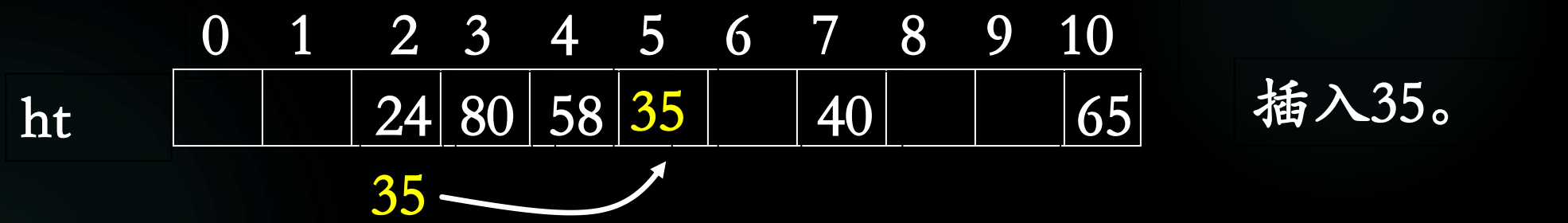


线性探查法的插入

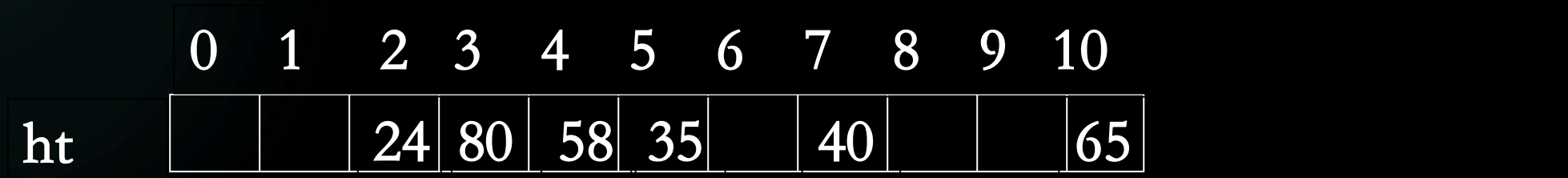
h(key)=key % 11



插入58，24。



插入35。



线性探查散列表的搜索

基本思想：从基位置h(key)开始,按照线性循环探查序列查找该元素。

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、回到h(key)（表满）

线性探查散列表的搜索

散列函数 H(key)=key % 11

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

搜索 35　搜索成功：按照线性循环探查序列找到为key的元素

搜索 25　搜索失败：按照线性循环探查序列遇到空位置

线性探查散列表的搜索

散列函数 H(key)=key % 11

ht

0	1	2	3	4	5	6	7	8	9	10
22	23	24	80	58	35	6	40	8	9	65

搜索 25

搜索失败：按照线性循环探查序列搜索一遍，回到h(key)（表满）

线性探查散列表的删除

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

删除58：　若直接删除，后果是什么？

0	1	2	3	4	5	6	7	8	9	10
		24	80		35					65

搜索35　　遇到空位置，搜索失败！　如何解决？

线性探查散列表的删除

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、删除后，该元素的位置能够重新使用。

解决的办法：

1.为每个元素增加标志域empty，表示该位置是否未使用过

2.删除58时，不改变标志位，仍为false，元素处改为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

删除元素的方法：

1.首先搜索该元素

2.若搜索成功，则置该位置为空值NU，但empty域不作更改！

搜索 35，25

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

线性探查散列表的搜索

改进算法：从基位置h(key)开始,按照线性循环探查序列查找该元素。
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（empty[i]=true）
2、回到h(key)（表满）

线性开地址法散列表的插入

函数探查第一个空值NeverUsed的位置， 插入新元素。

例如：插入2

首先搜索2，若搜索结果为存在重复元素则插入失败

如果搜索不到2，但是表已满，则插入失败

如果搜索不到2，则在搜索到的第一个NU位置插入元素2，empty设为F

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

线性探查法的缺点：很快表中所有位置的empty都变成F

易使元素在表中连成一片，探查次数增加，影响搜索效率

改进方法：1、二次探查法
2、双散列法

1、二次探查法

二次探测法使用下列探测序列进行探测，直到某个位置为空时，将关键字为key的元素插入该位置。

探查序列为： $h(key),h_1(key),h_2(key),\cdots,h_{2i-1}(key),h_{2i}(key),\cdots$。

探测序列由下列函数得到：

$h_{2i-1}(key)=(h(key)+i^2)\ \%\ M$

$h_{2i}(key)=(h(key)-i^2)\ \%\ M$

$i=1,2,\cdots,(M-1)/2$

1、二次探查法

探测序列由下列函数得到：

$h_{2i-1}(key)=(h(key)+i^2)\ \%\ M$

$h_{2i}(key)=(h(key)-i^2)\ \%\ M$

$i=1,2,\cdots,(M-1)/2$

$i=1$ $h_1(key)=(h(key)+1^2)\ \%\ M$

$h_2(key)=(h(key)-1^2)\ \%\ M$

$i=2$ $h_3(key)=(h(key)+2^2)\ \%\ M$

$h_4(key)=(h(key)-2^2)\ \%\ M$

$i=3$ $h_5(key)=(h(key)+3^2)\ \%\ M$

$h_6(key)=(h(key)-3^2)\ \%\ M$

$i=4$ $h_7(key)=(h(key)+4^2)\ \%\ M$

$h_8(key)=(h(key)-4^2)\ \%\ M$

$i=5$ $h_9(key)=(h(key)+5^2)\ \%\ M$

$h_{10}(key)=(h(key)-5^2)\ \%\ M$

1、二次探查法　　散列函数 H(key)=key % 11　i最大为5

i=1　$h_1(key)=(h(key)+1^2)\ \%\ M$

　　$h_2(key)=(h(key)-1^2)\ \%\ M$

i=2　$h_3(key)=(h(key)+2^2)\ \%\ M$

　　$h_4(key)=(h(key)-2^2)\ \%\ M$

i=3　$h_5(key)=(h(key)+3^2)\ \%\ M$

　　$h_6(key)=(h(key)-3^2)\ \%\ M$

i=4　$h_7(key)=(h(key)+4^2)\ \%\ M$

　　$h_8(key)=(h(key)-4^2)\ \%\ M$

i=5　$h_9(key)=(h(key)+5^2)\ \%\ M$

　　$h_{10}(key)=(h(key)-5^2)\ \%\ M$

h(40)=7

i=1　$h_1(40)=(7+1^2)\ \%\ 11=8$

　　$h_2(40)=(7-1^2)\ \%\ 11=6$

i=2　$h_3(40)=(7+2^2)\ \%\ 11=0$

　　$h_4(40)=(7-2^2)\ \%\ 11=3$

i=3　$h_5(40)=(7+3^2)\ \%\ 11=5$

　　$h_6(40)=(7-3^2)\ \%\ 11=9$

i=4　$h_5(40)=(7+4^2)\ \%\ 11=1$

　　$h_6(40)=(7-4^2)\ \%\ 11=2$

i=5　$h_5(40)=(7+5^2)\ \%\ 11=10$

　　$h_6(40)=(7-5^2)\ \%\ 11=4$

$h(key)=key\%11$

0	1	2	3	4	5	6	7	8	9	10
	35	24	80	15		13				65

插入35

$h(key)=35\%11=2$

$h_1(key)=(h(key)+1^2)\%11=3$

$h_2(key)=(h(key)-1^2)\%11=1$

插入13

$h(key)=13\%11=2$

$h_1(key)=(h(key)+1^2)\%11=3$

$h_2(key)=(h(key)-1^2)\%11=1$

$h_3(key)=(h(key)+2^2)\%11=6$

二次探测法能改善“线性聚集”，但是当二个关键字散列到同一位置时，则会有相同的探测序列，产生“二次聚集”。

2、双散列法

具备两个散列函数h1和h2

探查序列为：

$h_1(key),(h_1(key)+ h_2(key))\%M，(h_1(key)+ 2h_2(key))\%M，\cdots。$

$h_2(key)$应该是小于M且与M互质的整数，以保证探测序列能够最多经过M次探测即可遍历表中所有地址。

若M为素数，则可取$h_2(key)=key \% (M-2)+1$

2、双散列法

具备两个散列函数h1和h2

探查序列为：

$h_1(key),(h_1(key)+h_2(key))\%M，(h_1(key)+2h_2(key))\%M，\cdots。$

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

0	1	2	3	4	5	6	7	8	9	10
			80	15				58		65

插入58

$h_1(key)=58\%11=3$

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

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

2、双散列法

具备两个散列函数h1和h2

探查序列为：

$h_1(key),(h_1(key)+h_2(key))\%M，(h_1(key)+2h_2(key))\%M，\cdots。$

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

0	1	2	3	4	5	6	7	8	9	10
		24	80	15				58		65

插入24　$h_1(key)=24\%11=2$

2、双散列法

具备两个散列函数h1和h2

探查序列为：

$h_1(key),(h_1(key)+h_2(key))\%M，(h_1(key)+2h_2(key))\%M，\cdots。$

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

0	1	2	3	4	5	6	7	8	9	10
35		24	80	15				58		65

插入35

$h_1(key)=35\%11=2$

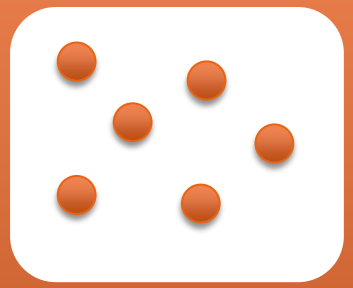
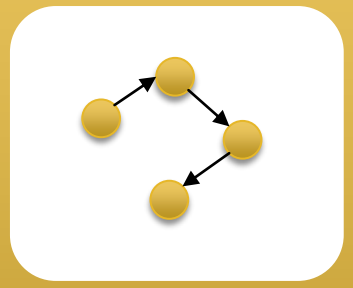
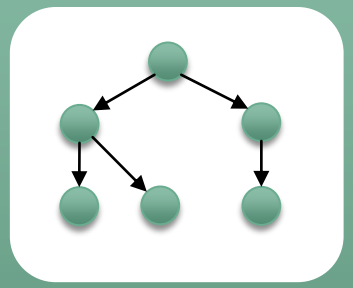
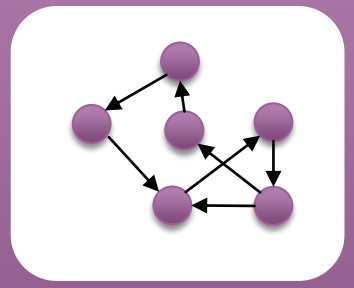
$h_2(key)=35\%9+1=9$

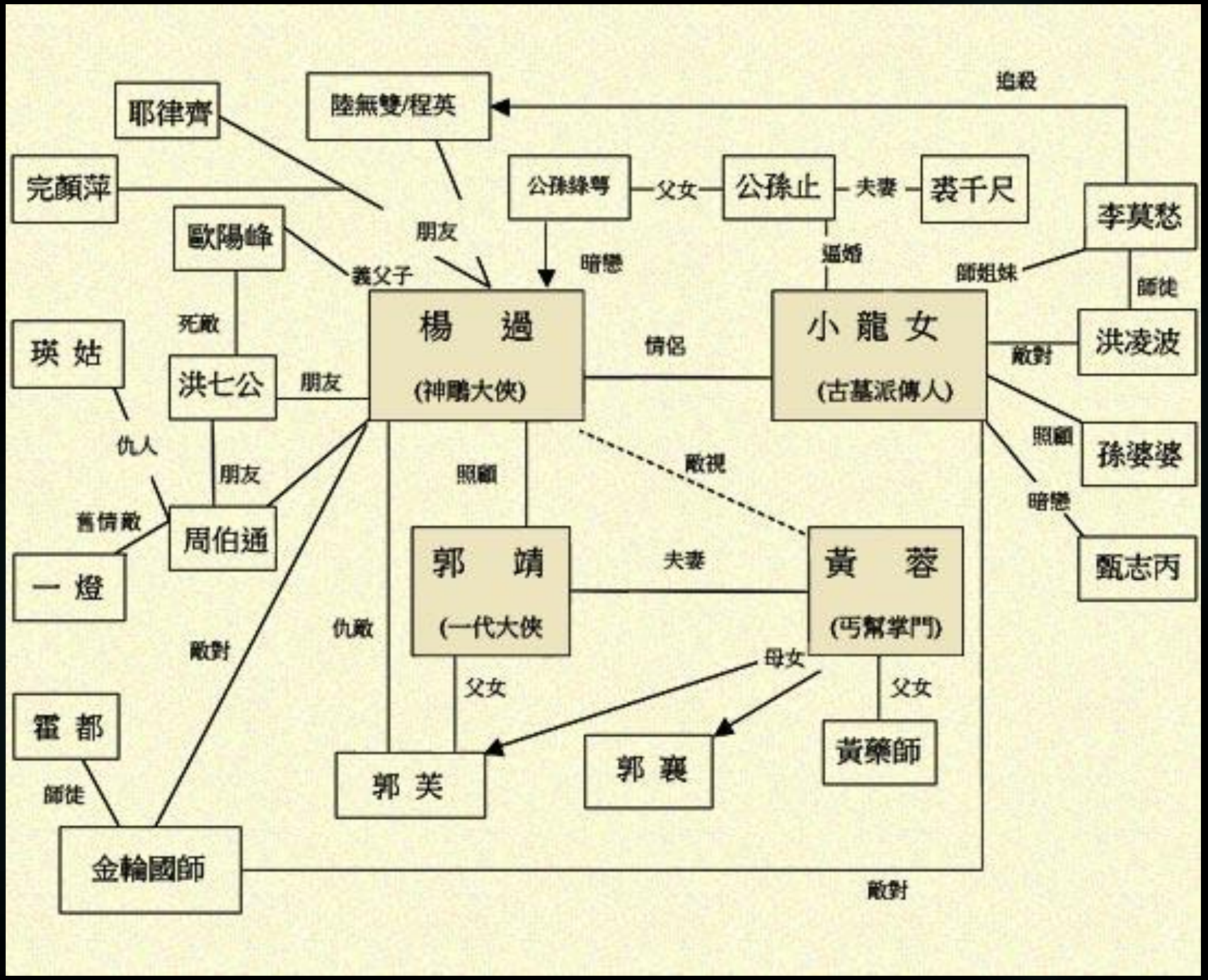
$(h_1(key)+h_2(key))\%11=(2+9)\%11=0$

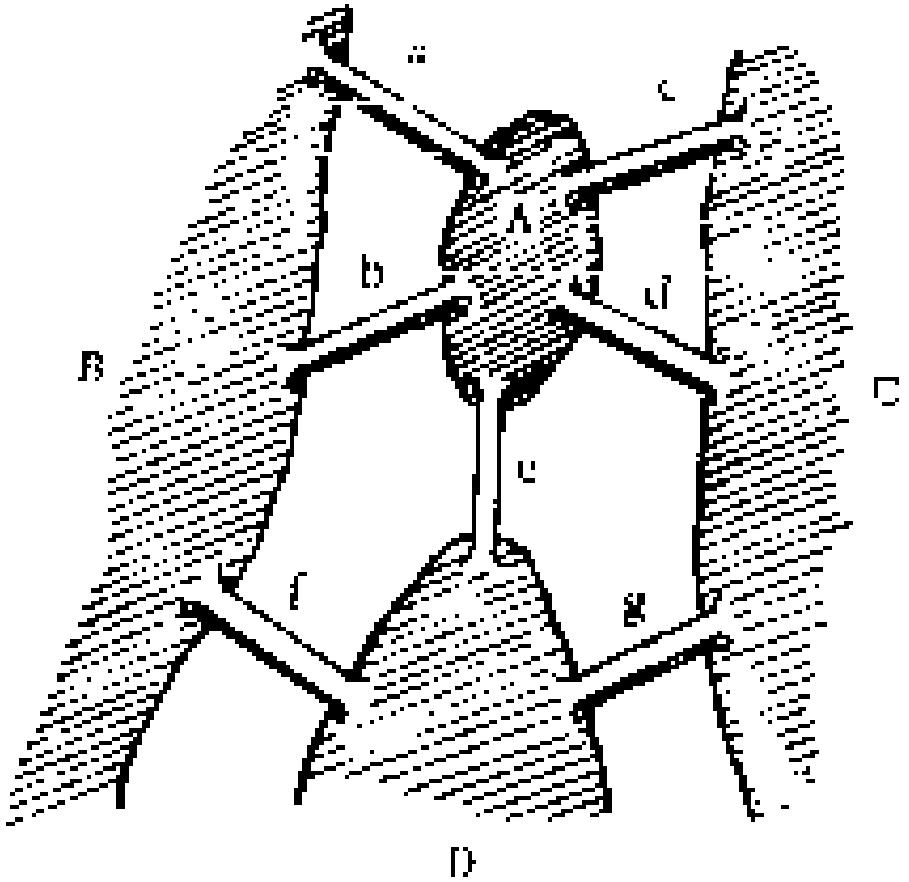
图

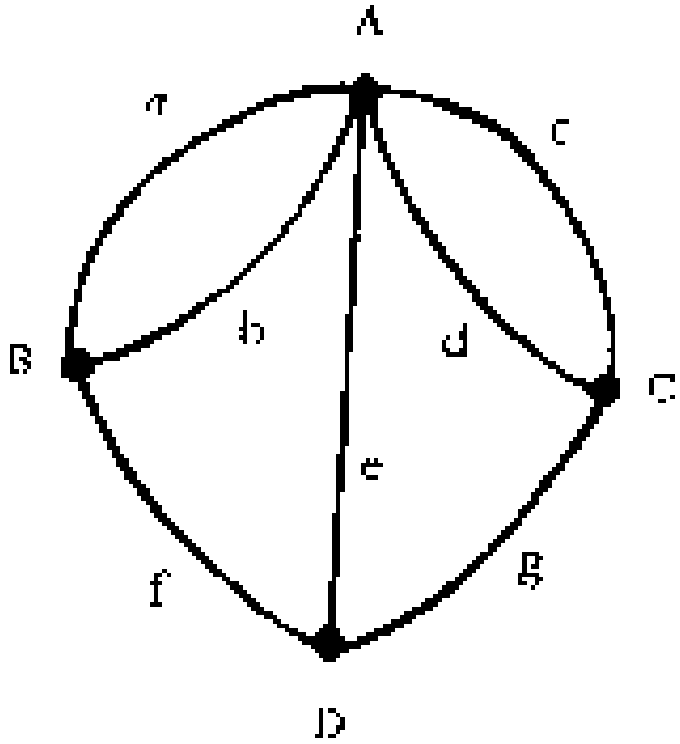
目录

基本逻辑结构

集合结构	线性结构	树形结构	图形结构
			
无关系	一对一关系	一对多关系	多对多关系


耶律齊
陸無雙/程英
追殺
完顏萍
公孫綠萼
父女
公孫止
夫妻
裘千尺
李莫愁
朋友
暗戀
逼婚
師姐妹
師徒
歐陽峰
義父子
楊　過
(神鵰大侠)
小龍女
(古墓派傳人)
洪凌波
敵對
情侶
死敵
瑛姑
洪七公
朋友
照顧
孫婆婆
仇人
敵視
朋友
暗戀
舊情敵
周伯通
甄志丙
一燈
郭　靖
(一代大侠
夫妻
黃　蓉
(丐幫掌門)
仇敵
敵對
母女
父女
父女
霍都
郭芙
郭襄
黃藥師
師徒
金輪國師
敵對

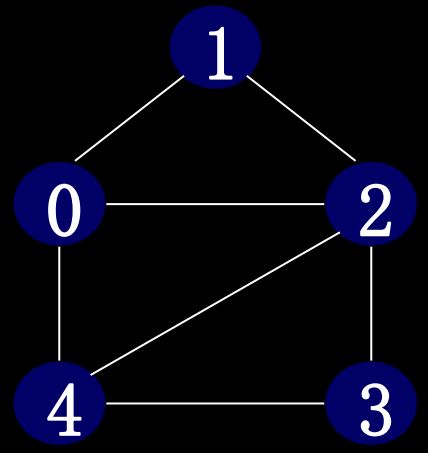




18世纪时，欧洲有一个风景秀丽的小城哥尼斯堡，那里有七座桥。河中的小岛A与河的左岸B、右岸C各有两座桥相连结，河中两支流间的陆地D与A、B、C各有一座桥相连结。当时哥尼斯堡的居民中流传着一道难题：一个人怎样才能一次走遍七座桥，每座桥只走过一次，最后回到出发点？

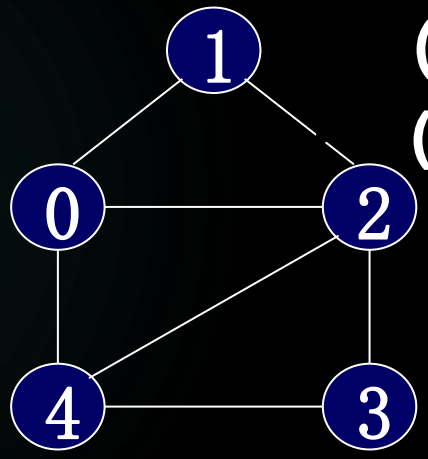
图的定义与术语

图中的结点又称为顶点(vertex)。
结点的偶对称为边(edge)，例如（1，2）；
图(graph)：是数据结构 G=(V,E)，其中V(G)是G中结点的有限非空集合；E(G)是G中边的有限集合。



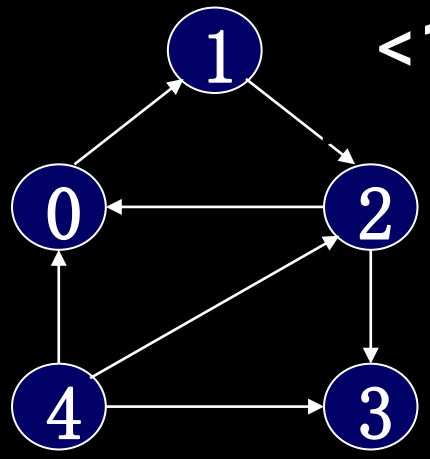
图的定义与术语

有向图(directed graph)：指图中代表边的偶对是有序的。
用<u，v>代表一条有向边（又称为弧），则u称为该边的始点（尾），v称为边的终点（头）。
无向图(undirected graph)：指图中代表边的偶对是无序的
在无向图中边(u，v)和(v，u)是同一条边。



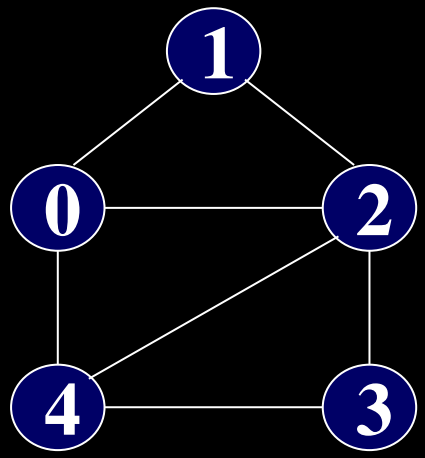
(1，2)
(2，1)

(a)无向图G_1

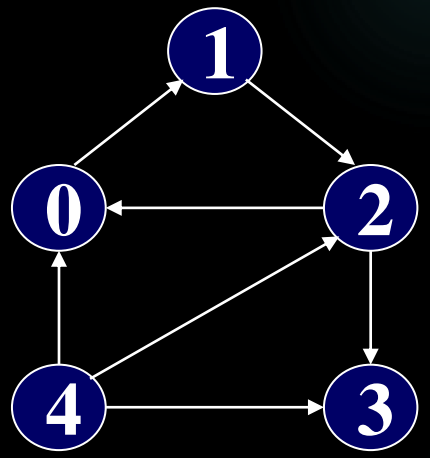


<1，2>

(b)有向图G_2



(a)无向图G_1



(b)有向图G_2

图中

$V(G_1)=V(G_2)=\{0,1,2,3,4\}$

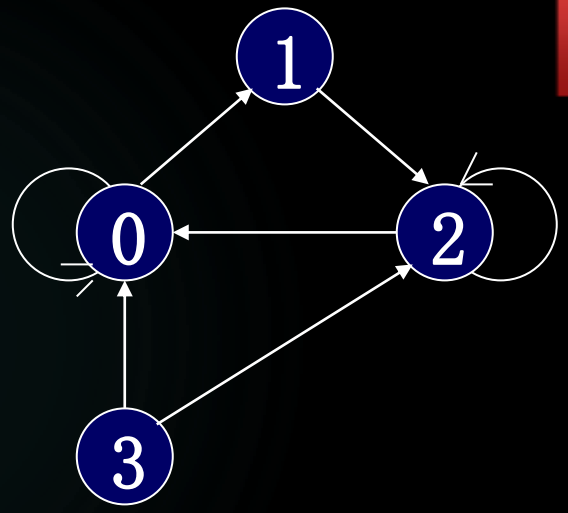
$E(G_1)=\{(0,1),(0,2),(0,4),(1,2),(2,3),(2,4),(3,4)\}$

$E(G_2)=\{<0,1>,<2,0>,<4,0>,<1,2>,<2,3>,<4,2>,<4,3>\}$

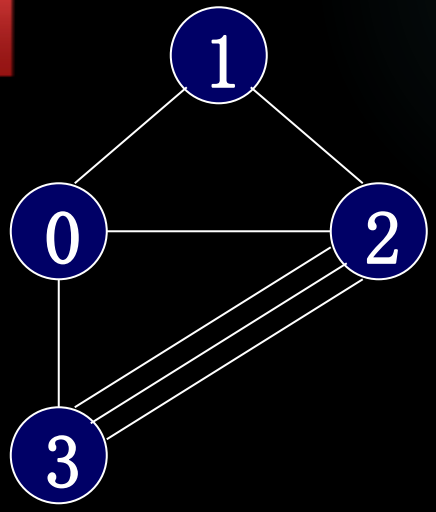
自回路：如果图中存在无向边(u,u)或有向边<u,u>，则称这样的边为自回路。

多重图：指图中两个顶点间允许有多条相同的边。

不予考虑

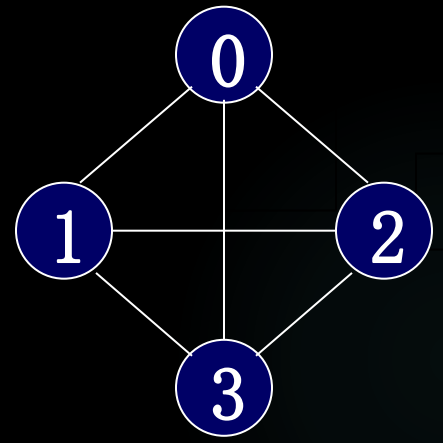


(a) 自回路



(b) 多重图

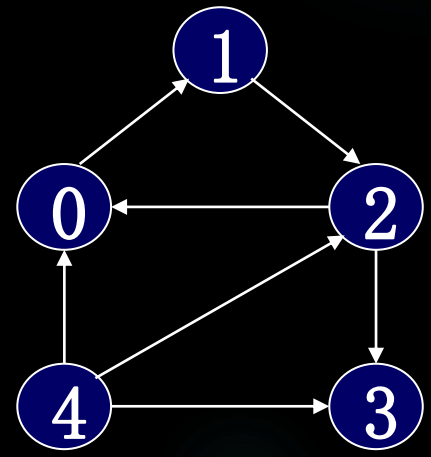
完全图：如果一个图有最多的边数，称为完全图。
无向完全图有n(n-1)/2条边，
有向完全图有n(n-1)条边。
例如：左图是一个完全图。有6条边。



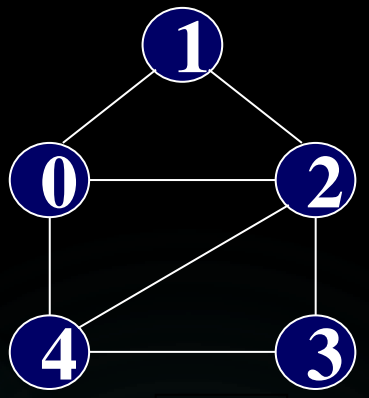
完全图

邻接:如果(u,v)是**无向图**中的一条边，则称顶点u和v相邻接，并称边(u,v)与顶点u和v相关联。
例如：顶点1和2是相邻接的。

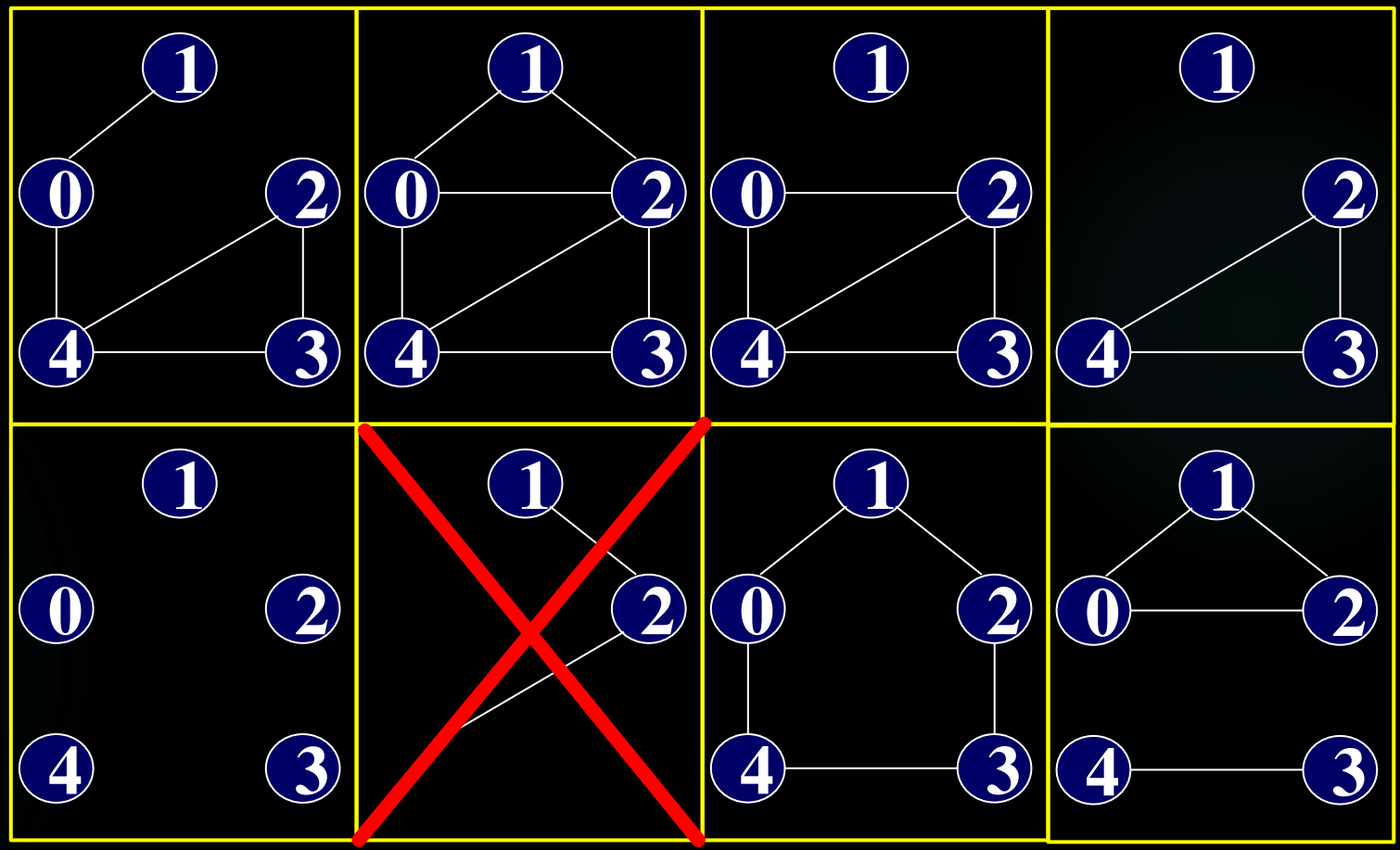
如果<u,v>是**有向图**中的一条边，则称顶点u邻接到v；称顶点v邻接自u，并称边<u,v>与顶点u和v相关联。



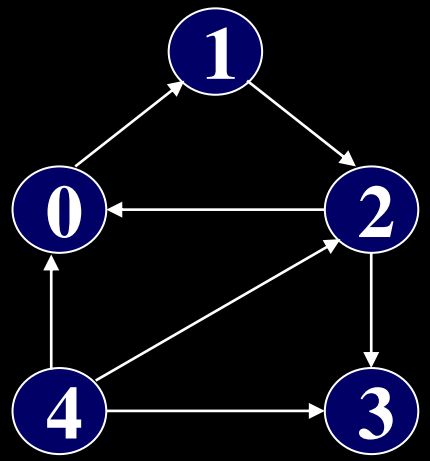
子图：图G的一个子图是图G’=(V’,E’)，其中V’(G’)⊆V(G)，E’(G’)⊆E(G).

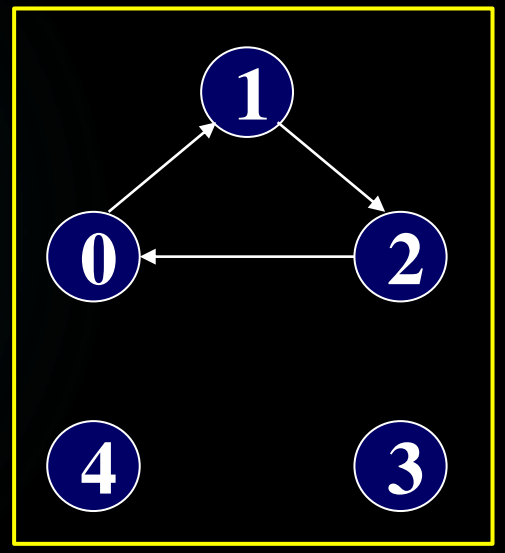


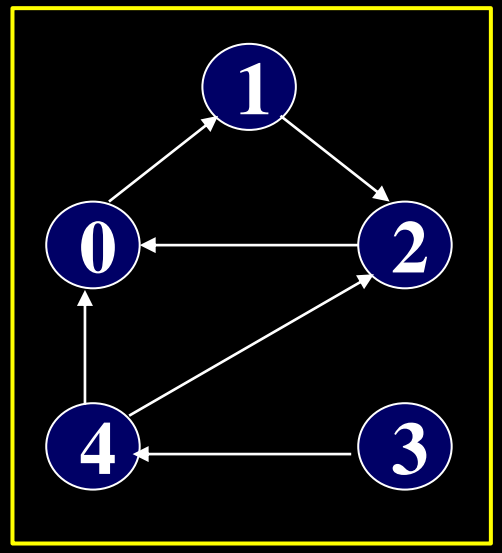
G_1



子图：图G的一个子图是图G’=(V’,E’)，其中V’(G’)⊆V(G)，E’(G’)⊆E(G).

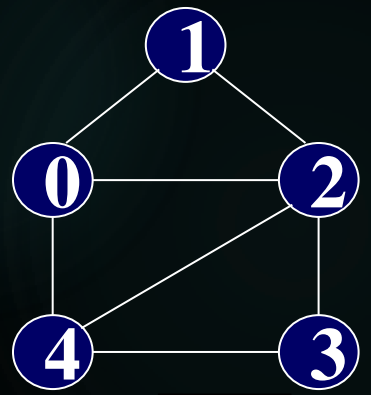






路径：在无向图G中，一条从s到t的路径是存在一个顶点序列$(s,v_1,v_2,\ldots,v_k,t)$，使得$(s,v_1),(v_1,v_2),\ldots,(v_k,t)$都是图G中的边

对于有向图顶点序列$s,v_1,v_2,\ldots,v_k,t$，应使$<s,v_1>,<v_1,v_2>,\ldots,<v_k,t>$都是图G中的边。



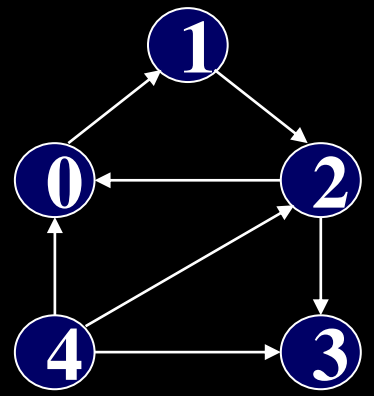
G_1

1与3之间的路径：

(1,2,3)

(1,0,4,3)

(1,2,3,4,2,3)



G_2

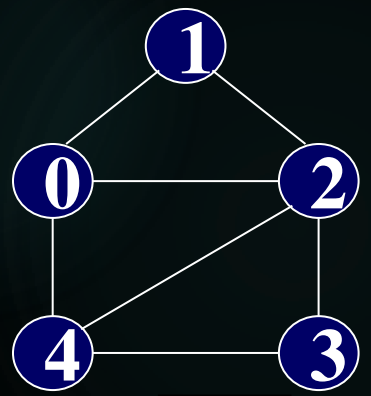
1与3之间的路径：

(1,2,3)

路径长度: 指路径上边的数目。

简单路径: 除起点和终点可以相同外，路径上其余顶点各不相同。

回路: 起点和终点相同的简单路径。



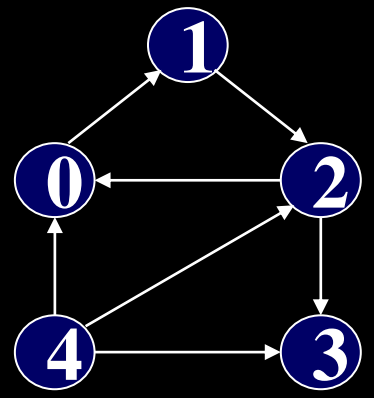
G_1

1与3之间的路径长度

(1,2,3)： 2

(1,0,4,3):3

(1,2,3,4,2,3):5



G_2

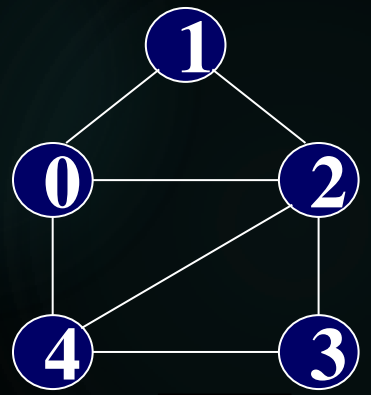
1与3之间的路径：

(1,2,3):2

路径长度: 指路径上边的数目。

简单路径: 除起点和终点可以相同外，路径上其余顶点各不相同。

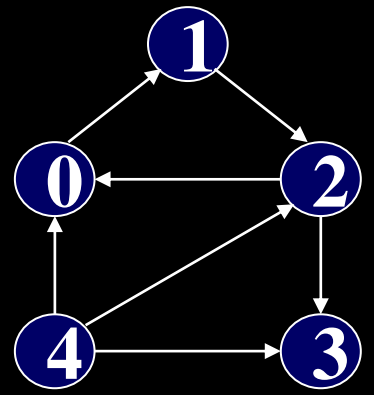
回路: 起点和终点相同的简单路径。



G_1

1与3之间的路径
(1,2,3)： 简单路径
(1,0,4,3):简单路径
(1,2,3,4,2,3):非简单路径

回路：(1,2,1)
非回路：(1,2,3,4,2,1)



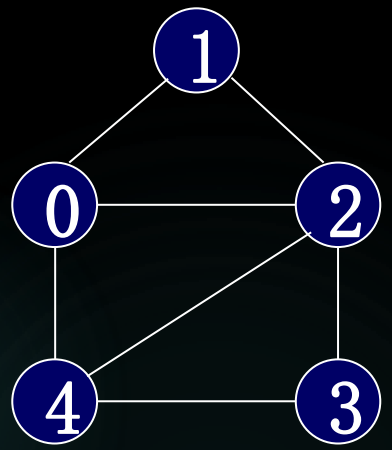
G_2

1与3之间的路径：
(1,2,3):简单路径
无回路

无向图中如果两个顶点u和v之间存在一条路径，则称顶点u和v是连通的，否则是不连通的。

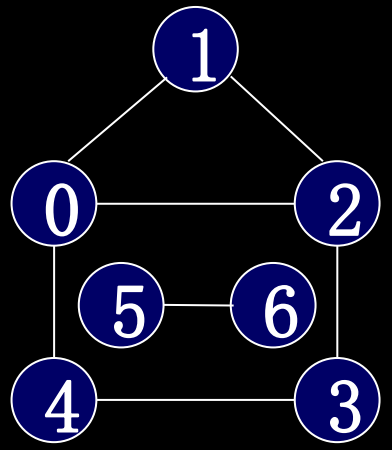
连通图:无向图中如果任意两个顶点之间是连通的。

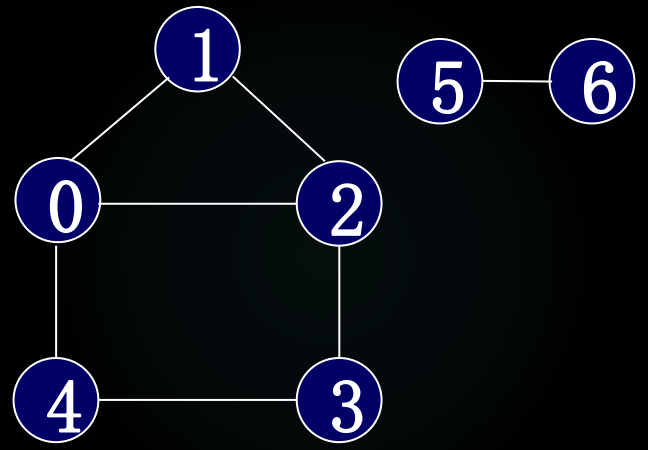
连通分量:无向图的极大连通子图。



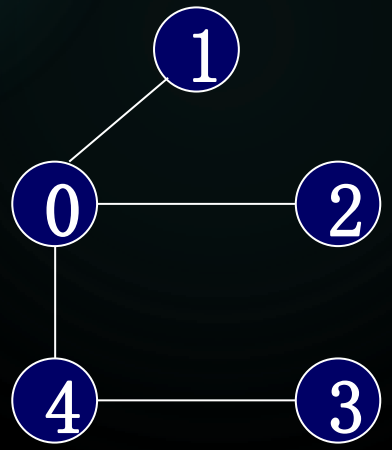
0和3是连通的

是连通图





0和6是不连通的，该图是非连通图，但它存在两个连通分量。



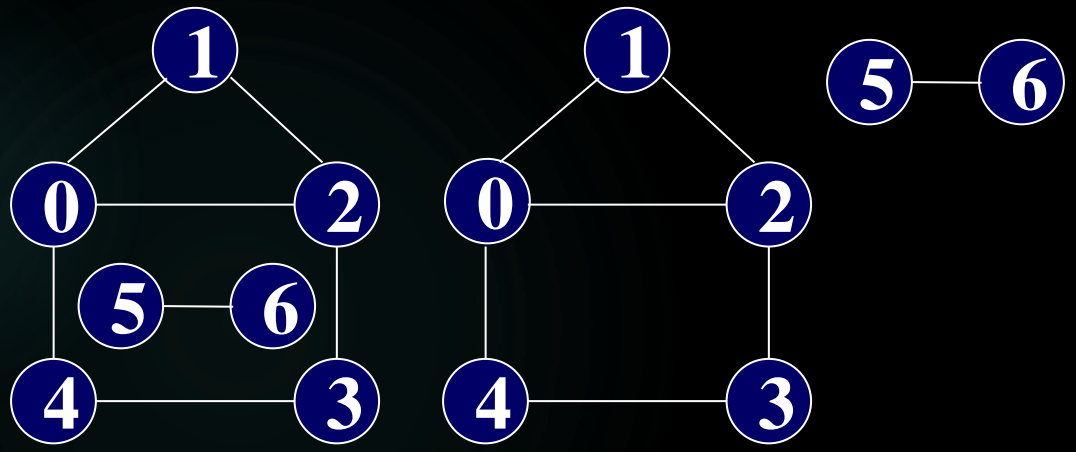
0和3是连通的

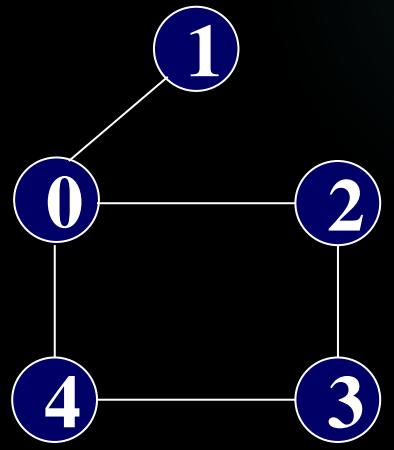
是连通图

注意极大的含义：如果某个连通子图再加上一个顶点后，仍是连通的，则它不是极大的连通子图。

求无向图的连通分量应该注意：

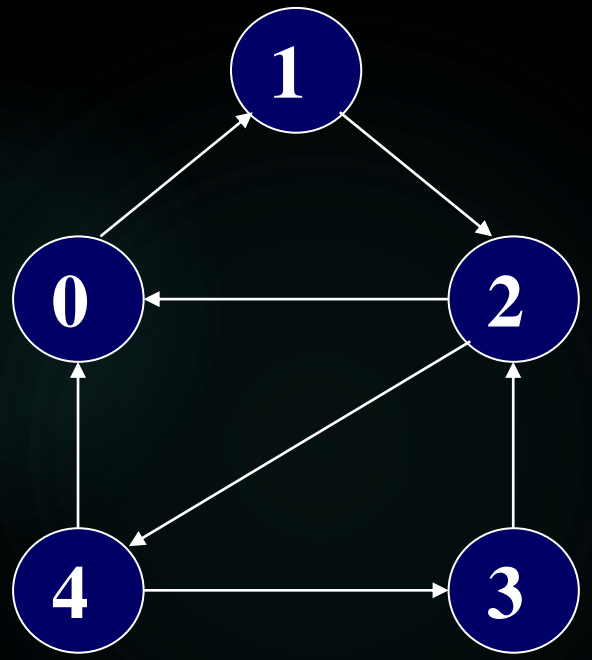
1.图中连通分量可能有多个，不能只写顶点个数最多的那个连通分量。
2. 每个连通分量必须是极大的。
3. 每个连通分量必须写出该（强）连通分量顶点之间所有的边。





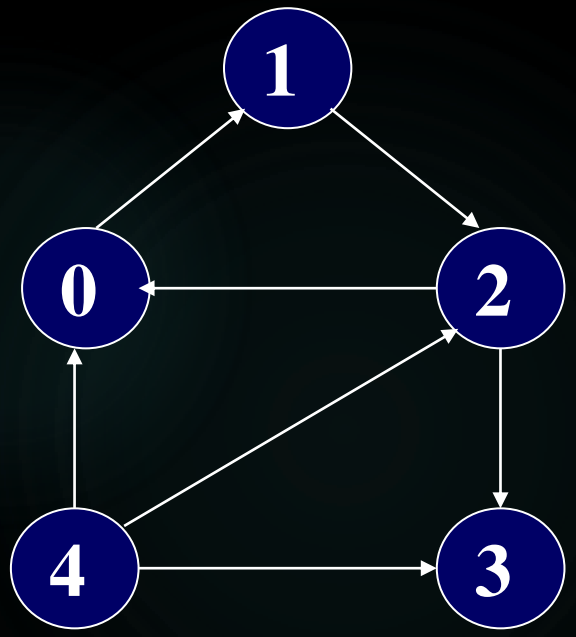
不是连通分量

强连通图：有向图中如果任意两个顶点u和v之间，存在一条从u到v的路径，同时存在一条从v到 u的路径，则称该有向图为强连通图。
强连通分量：有向图的极大连通子图。



	0	1	2	3	4
0	-	有	有	有	有
1	有	-	有	有	有
2	有	有	-	有	有
3	有	有	有	-	有
4	有	有	有	有	-

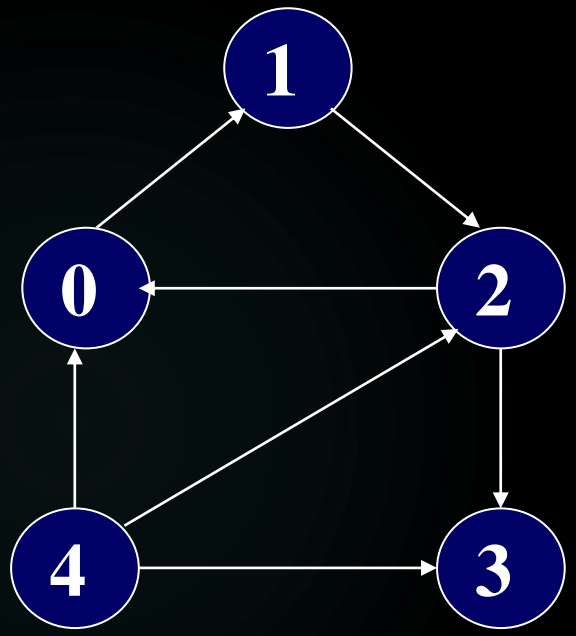
强连通图：有向图中如果任意两个顶点u和v之间，存在一条从u到v的路径，同时存在一条从v到 u的路径，则称该有向图为强连通图。
强连通分量：有向图的极大连通子图。

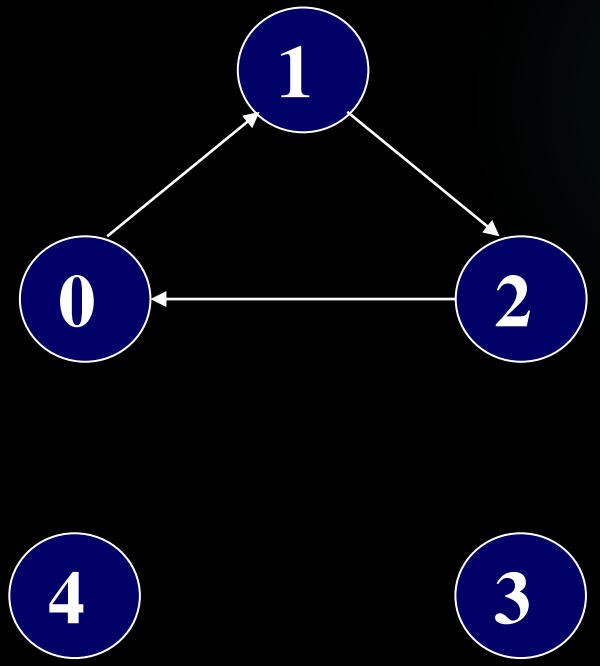


	0	1	2	3	4
0	-	有	有	有	无
1	有	-	有	有	无
2	有	有	-	有	无
3	无	有	无	-	无
4	有	无	无	有	-

强连通图：有向图中如果任意两个顶点u和v之间，存在一条从u到v的路径，同时存在一条从v到 u的路径，则称该有向图为强连通图。

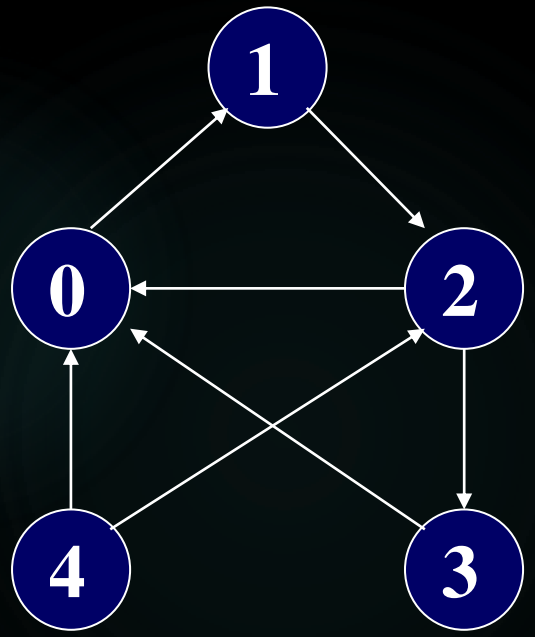
强连通分量：有向图的极大连通子图。

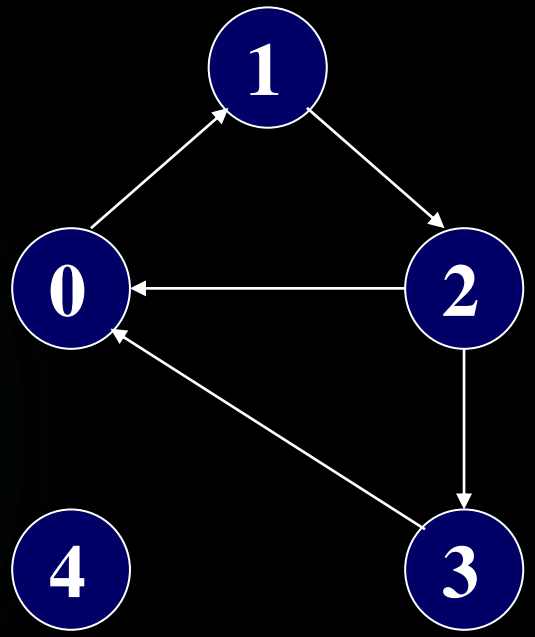


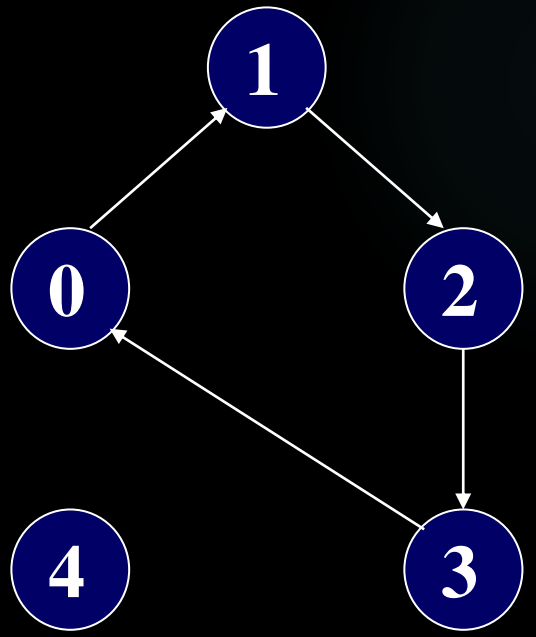


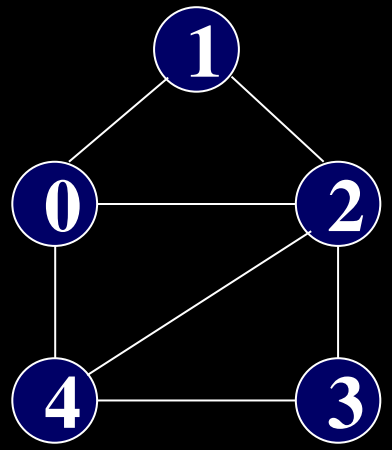
强连通图：有向图中如果任意两个顶点u和v之间，存在一条从u到v的路径，同时存在一条从v到 u的路径，则称该有向图为强连通图。

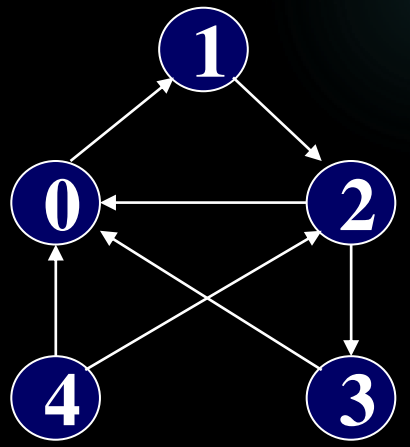
强连通分量：有向图的极大连通子图。











顶点的度：与该顶点相关联的边的数目。

入度：有向图中顶点v的入度指以v为头的弧的数目；

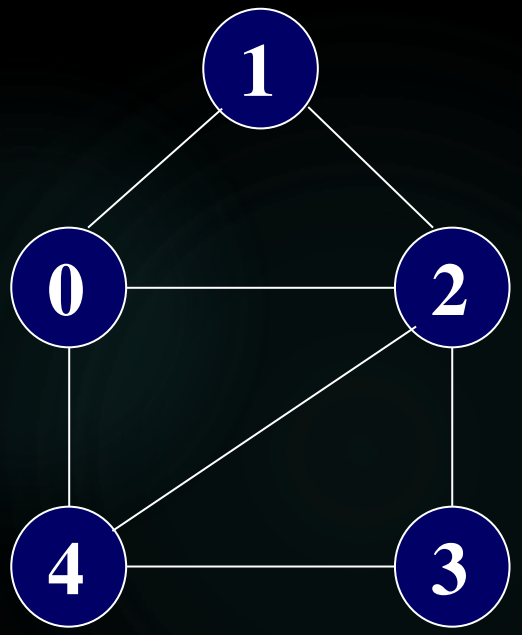
出度：有向图中顶点v的出度指以v为尾的弧的数目。

有向图中，顶点的度=入度+出度。

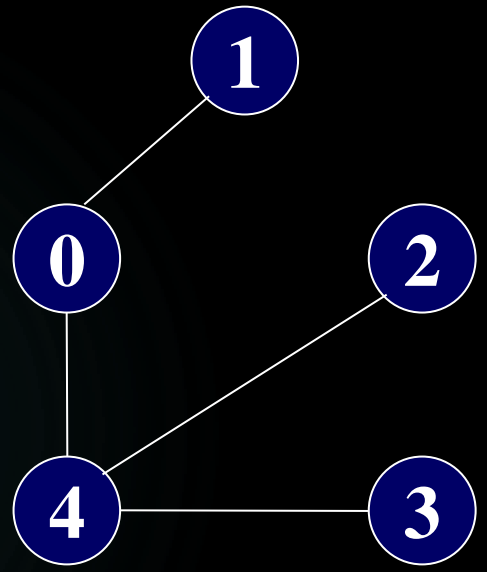
例如左图中，顶点1,2度分别为2和4。

右图中，顶点0的入度和出度分别为3和1，顶点0的度为4

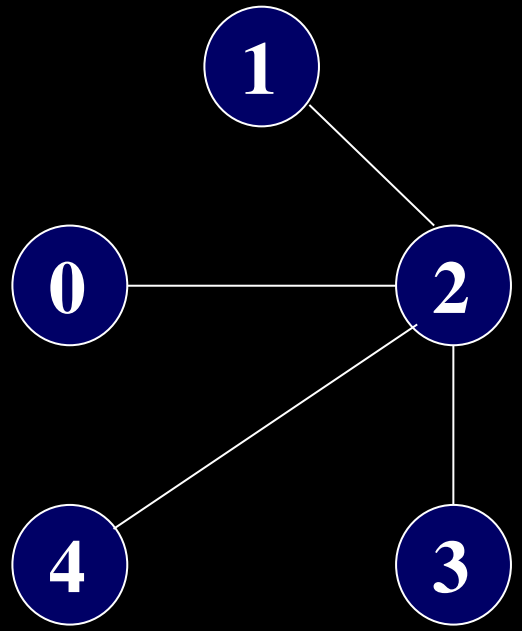
生成树：**无向图**的生成树是一个极小连通子图，它包含图中所有顶点，但只有足以构成一棵树的(n-1)条边。再加上一条边将构成回路。



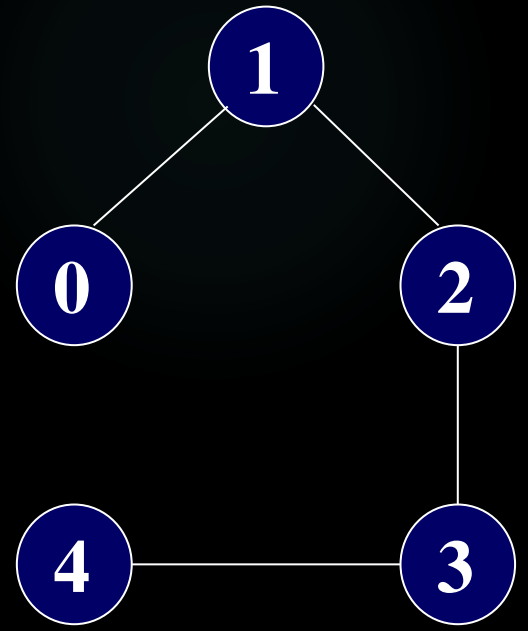
图G_1



图G_1的生成树

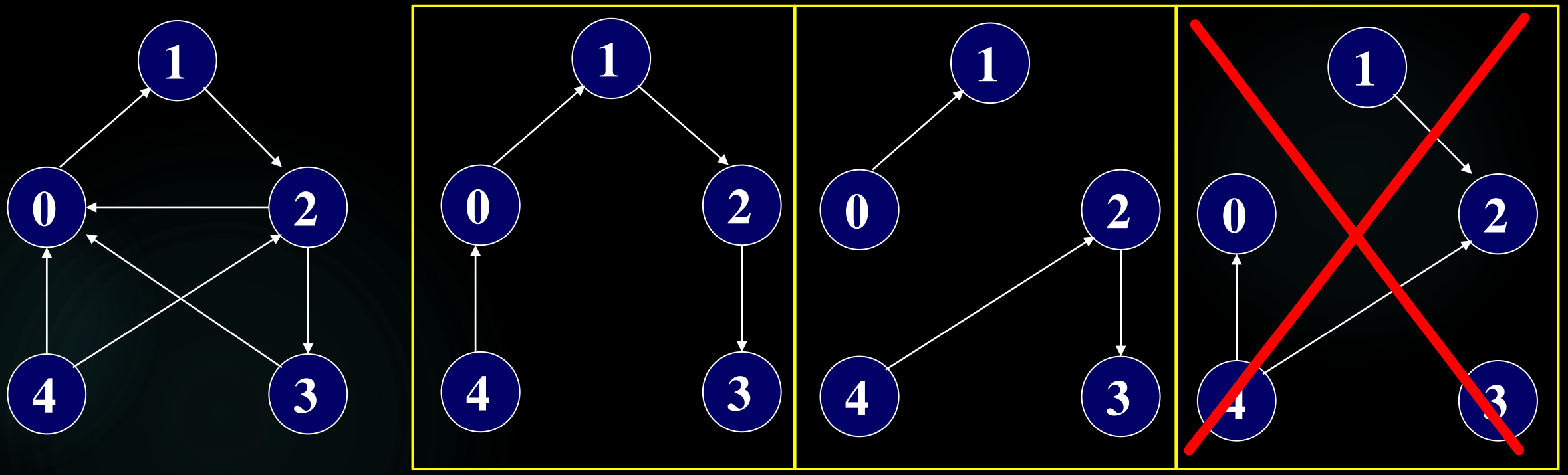


图G_1的生成树



图G_1的生成树

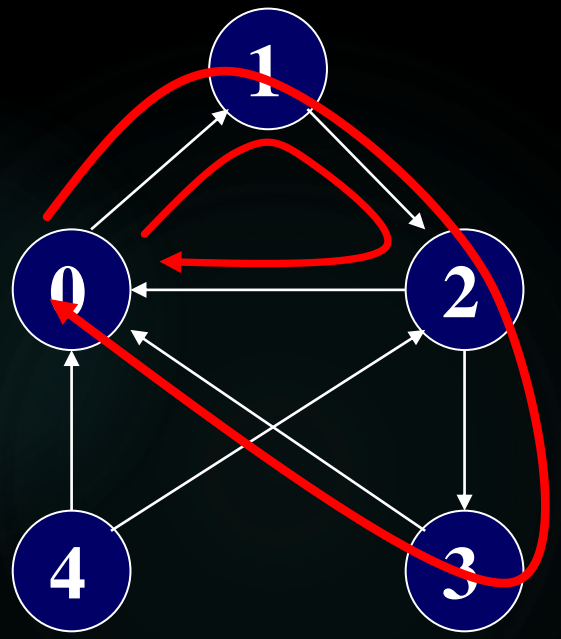
有向图的生成森林：是一个子图，由若干棵互不相交的有根有向树组成，包含图中所有的顶点。



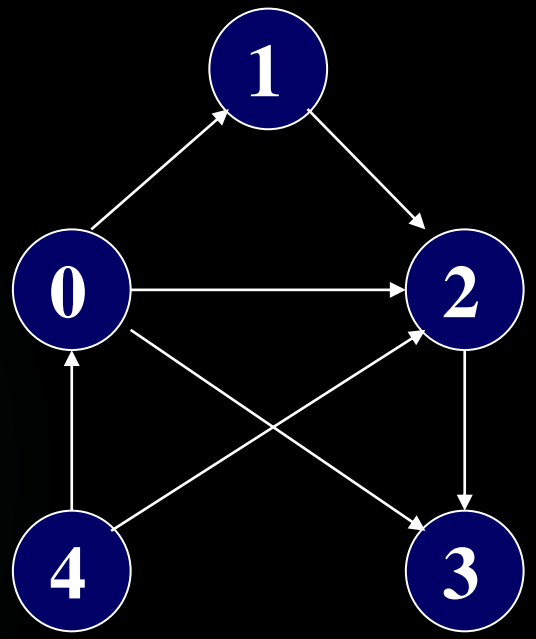
有根有向树：是一个有向图，它恰有一个顶点的入度为0，其余顶点的入度为1。如果略去边的方向，处理成无向图后，则图是连通的。

有向无环图（DAG图）：不包含回路的有向图。

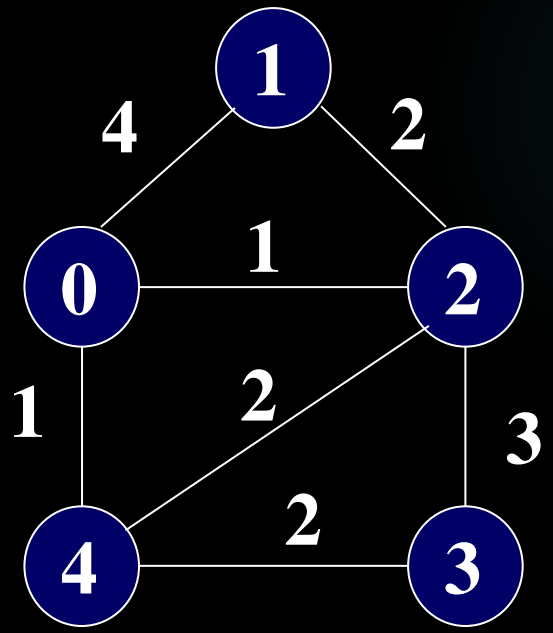
网：在图的每条边上加上一个数字称为权，也称代价，带权的图称为网。



非DAG



DAG



网

	无向图	有向图
	自回路、多重图	自回路、多重图
	完全图	完全图
	邻接	顶点u邻接到v
	子图、路径	子图、路径
	简单路径、回路	简单路径、回路
	连通图、连通分量	强连通图、强连通分量
	顶点的度	顶点的度、入度、出度
	生成树	生成森林
		有向无环图DAG

图

目录

- 图的基本概念
- 图的存储结构
- 图的遍历
- 拓扑排序
- 关键路径
- 最小代价生成树：普里姆算法
- 单源最短路径和所有顶点间的最短路径

带权有向图的抽象数据类型

ADT Graph

{ **数据**：顶点的非空集合V和边集E，每条边由E中偶对<u,v>表示。

运算:

Create()： 构造一个不包含任何边的有向图。

Destroy()：撤销一个有向图。

Exist(u,v): 如果图中存在边<u,v>,则函数返回true,否则返回false。

Add(u,v,w):向图中添加权为w的边<u,v>，若插入成功，返回Success；
若已存在边<u,v>，则返回Duplicate；其他返回Failure。

Delete(u,v): 从图中删除边<u,v>，若图中不存在边<u,v>，则返回
NotPresent；若图中存在边<u,v>，则从图中删除此边，
返回success；其他返回Failure。

Vertices(): 函数返回图中顶点数目。

}

对于图的操作，还有其它成员函数，将在以后陆续介绍。

主要有：

1. void DFS();　　//深度优先搜索图
2. void BFS();　　//宽度优先搜索图
3. void TopoSort(Graph g);　//拓扑排序
4. void CriticalPath(Graph g);　//关键路经
5. void Prim(Graph g, int k);
　　//普里姆算法求最小代价生成树
6. void Kruskal(Graph g, int edges); //克鲁斯卡尔算法求最小代价生成树
7. void Dijkstra(Graph g, int k, T d[], int p[]);
　　//迪杰斯特拉算法求单源最短路经
8. void Floyd(Graph g, T**d, int **p);
　　//弗洛伊德算法求所有顶点之间的最短路经

图的存储结构

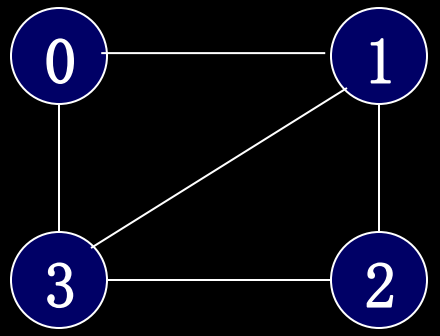
1 图的矩阵表示法及其实现
2 图的邻接表表示法及其实现

图的矩阵表示法及其实现

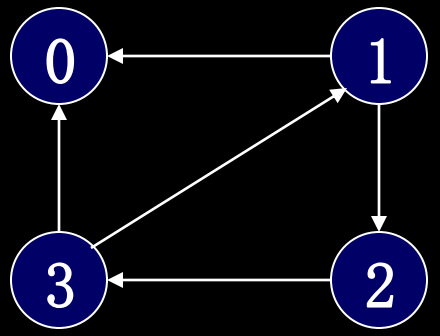
1、邻接矩阵

一个有n个顶点的图G=(V,E)的邻接矩阵是一个n×n的矩阵A，A的每个元素是0或1。

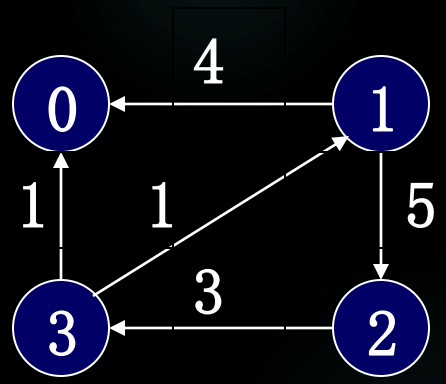
A[u][v]取值	无向图	有向图	带权有向图
0	u和v不邻接	u不邻接v	u=v(对角线)
1	(u,v)∈E或(v,u)∈E	<u,v>∈E	-
∞	-	-	u不邻接v
>0	-	-	w(u,v) <u,v>∈E



(a) 无向图G_1



(b) 有向图G_2



(c) 网G_3

	0	1	2	3
0	0	1	0	1
1	1	0	1	1
2	0	1	0	1
3	1	1	1	0

(d) 图G_1的邻接矩阵

	0	1	2	3
0	0	0	0	0
1	1	0	1	0
2	0	0	0	1
3	1	1	0	0

(e) 图G_2的邻接矩阵

	0	1	2	3
0	0	∞	∞	∞
1	4	0	5	∞
2	∞	∞	0	3
3	1	1	∞	0

(f) 网G_3的邻接矩阵

图的矩阵实现

```
Typedef struct graph{
        T NoEdge;
        int Vertices;
        T**A;
}Graph;
```

建立邻接矩阵

```
void GreateGraph(Graph* g, int n, T noedge)
{
        int i,j;
        g->NoEdge = noedge;
        g->Vertices = n;
        g->A = (T**)malloc(n*sizeof(T*));
        for(i=0;i<n;i++)
        {
                g->A[i] = (T*)malloc(n*sizeof(T));
                for(j=0;j<n;j++) g->g[i][j] = noedge;
                g->A[i][i] = 0;
        }
}
```

判断边是否存在

```
BOOL Exist(Graph g, int u,int v)
{       int n = g.Vertices;
        if(u<0||v<0||u>n-1||v>n-1||u==v||g.A[u][v]==g.NoEdge)
        return false;
  return true; }
```

边的插入

```
BOOL Insert(Graph *g, int u,int v, T w)
{       int n = g->Vertices;
        if(u<0||v<0||u>n-1||v>n-1||u==v) return FALSE;
        if(g.A[u][v]!=g->NoEdge) return FALSE;
        g->A[u][v]=w;
        return TRUE;
}
```

边的删除

```
BOOL Delete(Graph *g, int u,int v)
{       int n = g->Vertices;
        if(u<0||v<0||u>n-1||v>n-1||u==v)
        return FALSE;
        if(g->A[u][v]==g->NoEdge)
        return FALSE;
        g->A[u][v]=g->NoEdge;
        return TRUE;
}
```

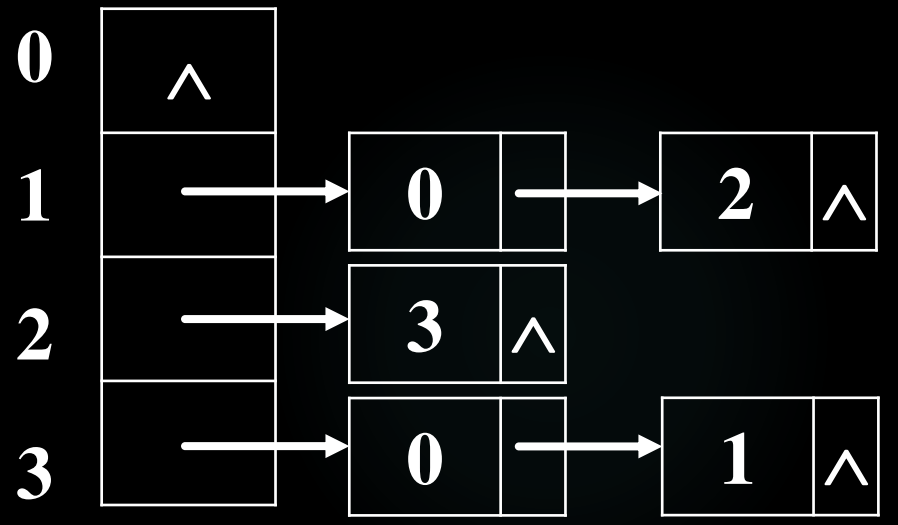
图的邻接表表示法

要点：

1、为图中每个顶点u建立一个单链表；

2、单链表中，每个结点代表一条边<u,v>,称为边结点

3、边结点的结构如下：



AdjVex	nextArc

(a) 边结点

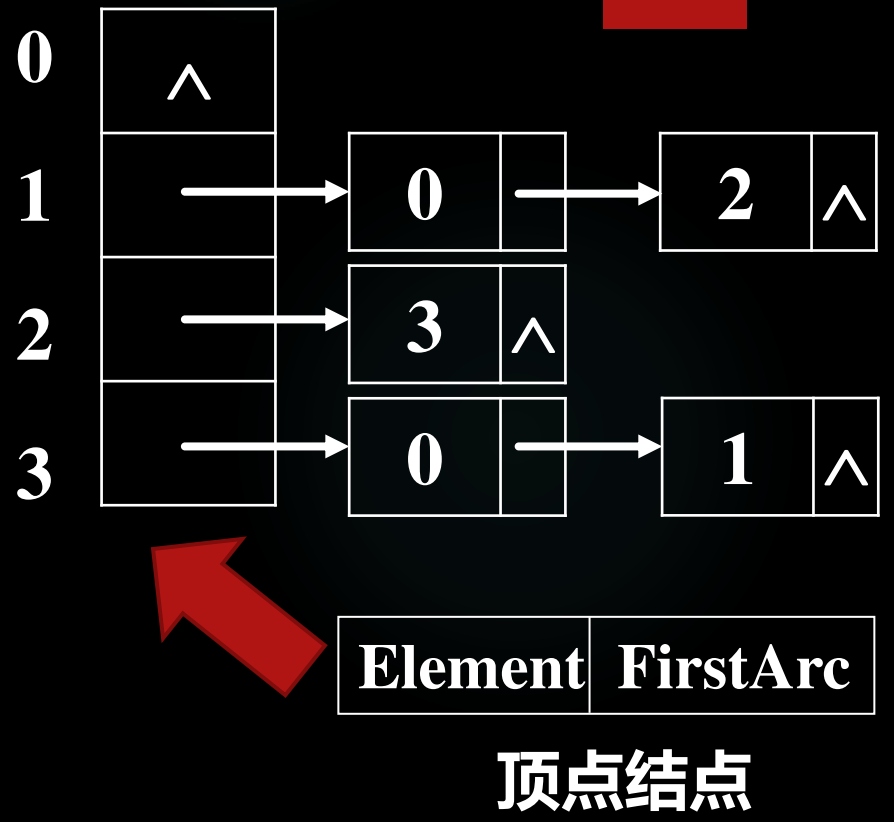
AdjVex	W	NextArc

(b) 带权的边结点

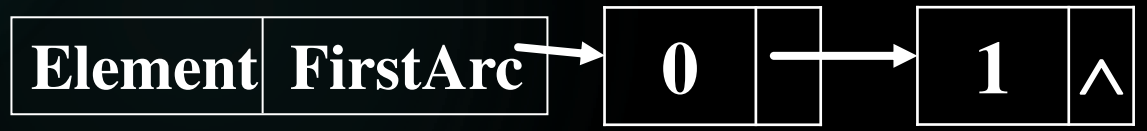
图的邻接表表示法

4、顶点u的单链表中，记录了u邻接到的全部结点

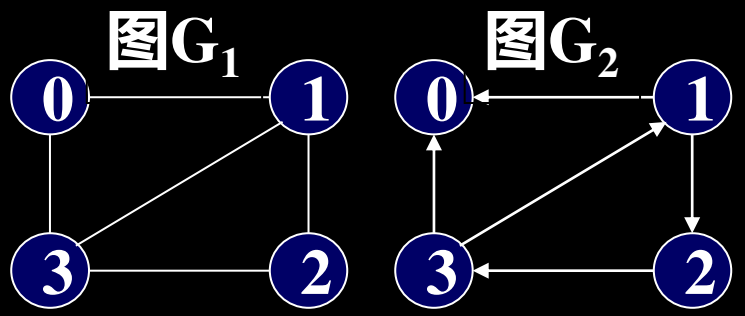
5、每个单链表设立一个存放顶点u相关信息的结点，称为顶点结点。可将顶点结点按顺序存储方式组织起来。

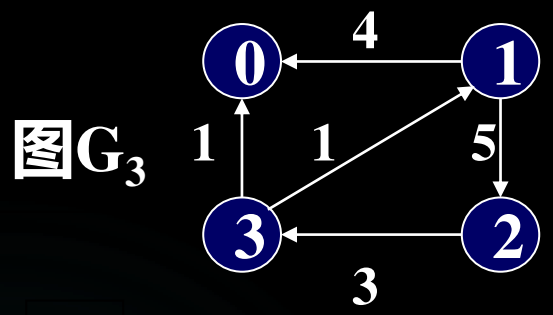


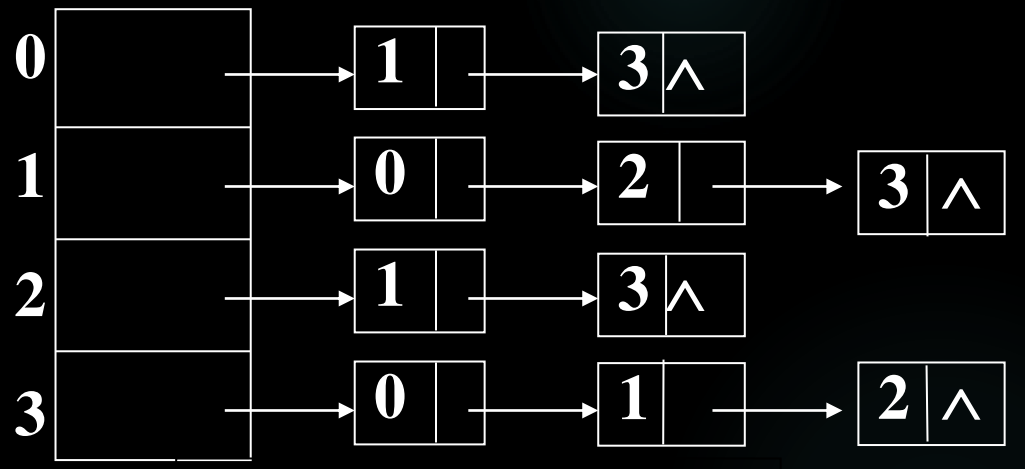
顶点结点



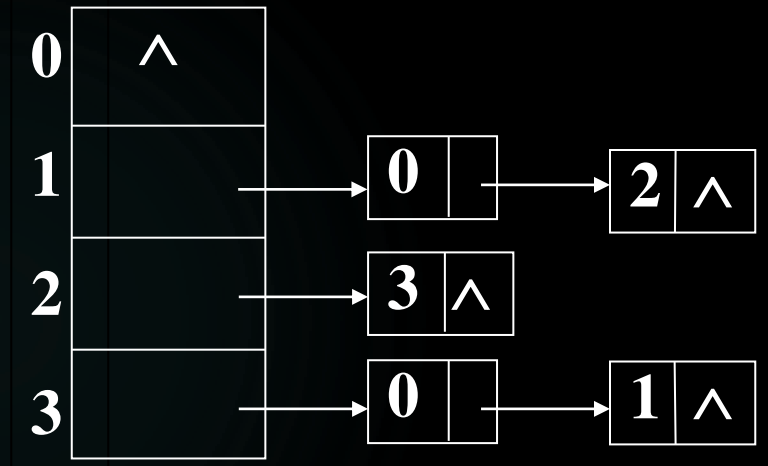
顶点结点



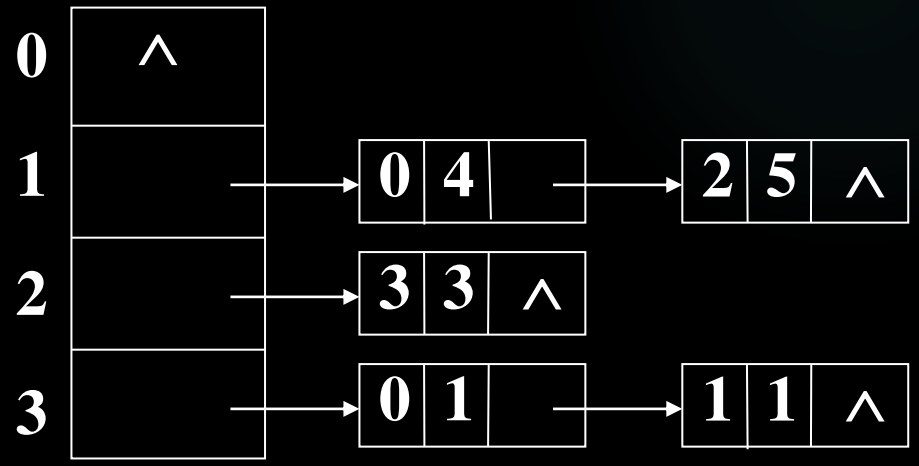




(a)图G_1的邻接表



(b)图G_2的邻接表



(c)图G_3的邻接表

图的邻接表实现

```
typedef struct enode //边结点
{
        int AdjVex;    //指示u邻接到的结点
        T W;
        struct enode* NextArc;
} ENode;
typedef struct graph //邻接表
{
        int Vertices;
        ENode** A;
} ENode;
```

创建邻接表

```
void CreateGraph(Graph*g, int n)
{
        int i;
        g->Vertices = n;
        g->A = (Enode**)malloc(n*sizeof(Enode*));
        for(i=0;i<n;i++) g->A[i] = NULL;
}
```

判断边是否存在

```
BOOL Exist(Graph g, int u,int v)
{       int n = g.Vertices;
        if(u<0||v<0||u>n-1||v>n-1||u==v)
        return FALSE;
        ENode* p = g->A[u];
        while (p && p->AdjVex!=v)
        p=p->NextArc;
        if (!p) return FALSE;
        else return TRUE;
}
```

插入边的函数

```
BOOL Add(Graph *g, int u, int v, T w)
 //将边<u,v>插入指针A[u]所指示的单链表最前面
{       int n = g->Vertices;
         if(u<0||v<0||u>n-1||v>n-1||u==v) return FALSE;
        if(Exist(*g,u,v))return FALSE;
        ENode* p=NewENode(v,w,g->A[u]);
        g->A[u]=p;
        return TRUE;
}
```

删除边的函数

```
BOOL Delete(Graph *g, int u,int v)
{
        int n = g->Vertices;
        if(u<0||v<0||u>n-1||v>n-1||u==v) return FALSE;
        ENode* p=g->A[u],*q=NULL;
         while (p&& p->AdjVex!=v) //查找<u,v>是否存在
         {       q=p;p=p->NextArc;    }
        if (!p) return FALSE;    //p为空，边不存在
         if (q) q->NextArc=p->NextArc;
         else g->A[u]=p->nextArc;
        free(p);
        return TRUE;
    }
```

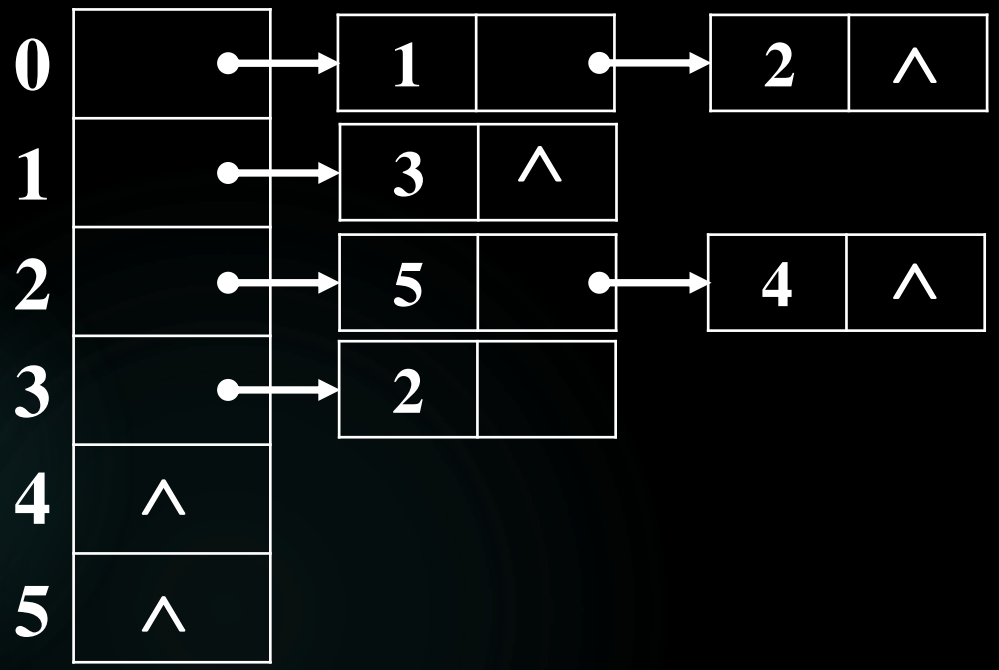
图

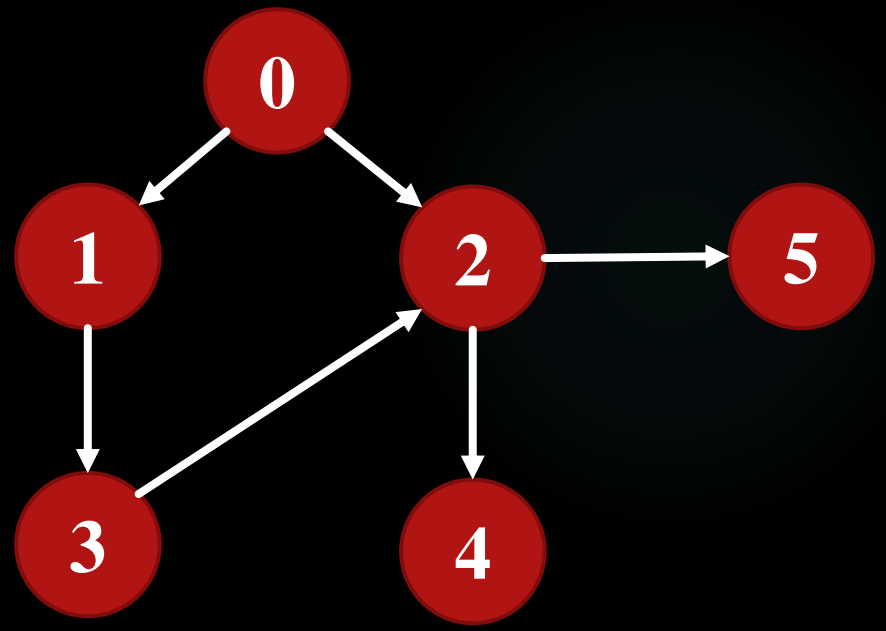
目录

- 图的基本概念
- 图的存储结构
- 图的遍历
- 拓扑排序
- 关键路径
- 最小代价生成树：普里姆算法
- 单源最短路径和所有顶点间的最短路径

图的遍历: 指从图G的任意一个顶点v出发，访问图中所有结点且每个结点仅访问一次的过程。

沿着边的方向行走





图遍历的方法:深度优先搜索（类似于树的先序遍历）和宽度优先搜索（类似于树的按层次遍历）

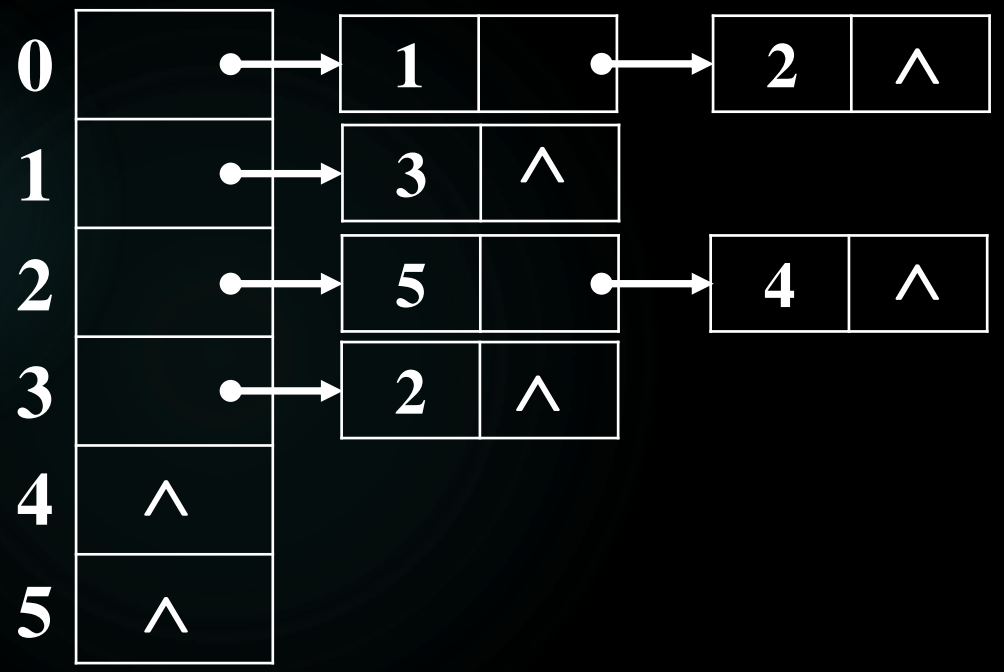
深度优先遍历

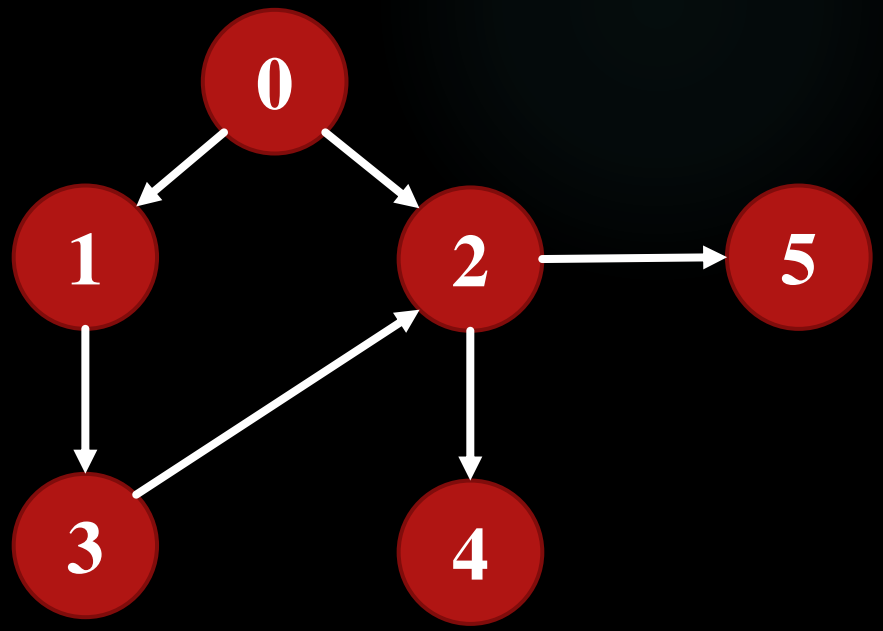
图遍历与树遍历的差异：

沿着边的方向行走

1、从图中任意一个顶点出发未必能到达其它所有顶点；

2、图中存在回路时，又可能多次经过同一个顶点。





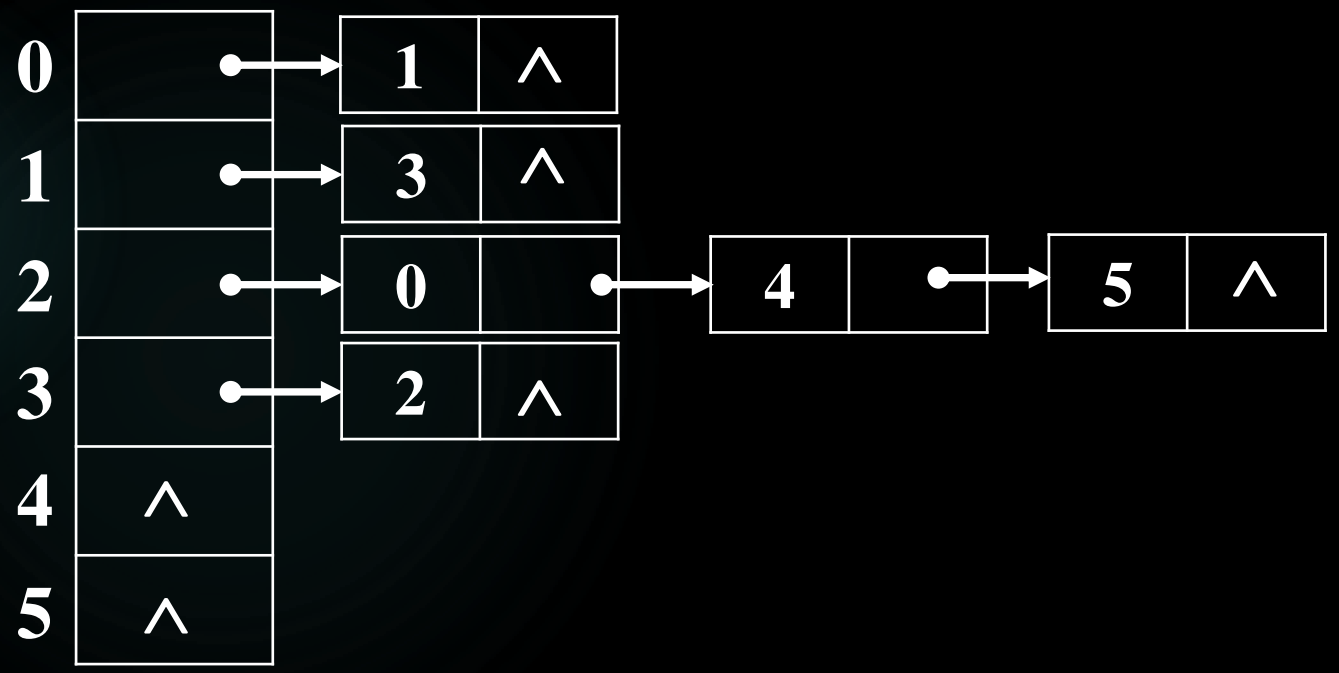
深度优先遍历

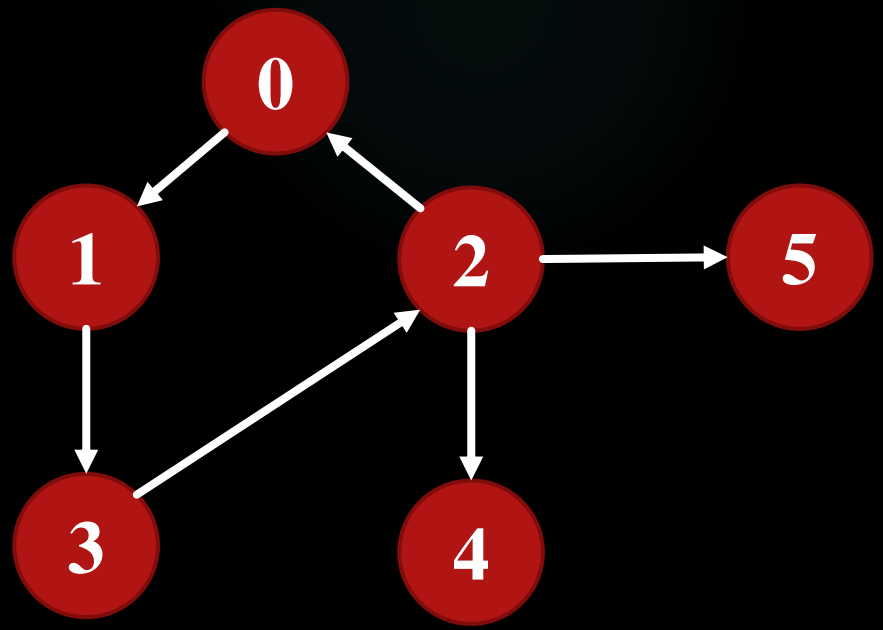
图遍历与树遍历的差异：

沿着边的方向行走

1、从图中任意一个顶点出发未必能到达其它所有顶点；

2、图中存在回路时，又可能多次经过同一个顶点。





深度优先遍历

图遍历与树遍历的差异:

1、从图中任意一个顶点出发**未必**能到达其它所有顶点；

2、图中存在回路时，又可能**多次经过**同一个顶点。

如果多次经过同一个顶点怎么办?

设置辅助数组visited [ ]，它的初始状态为false，在图的遍历过程中，一旦某一个顶点i被访问，则令visited[i]为true

一次遍历结束，visited有标记未被访问的顶点怎么办?

遍历算法应当另选一个未标记的顶点出发，再次执行图的遍历。

深度优先遍历

1、深度优先搜索算法

从图G中某个顶点v出发,深度优先搜索图的DFS算法如下:

1）访问顶点v并打上标记。

2）依次从v的未访问过的邻接点出发，深度优先搜索图G.

递归算法

前进(v)

1. 从顶点v出发，访问它的任一未访问**邻接顶点**w_1；
2. 从w_1出发，访问与 w_1的未访问邻接顶点w_2；
3. 从w_2出发，…，当到达顶点w_t，其邻接顶点都被访问过，暂停前进

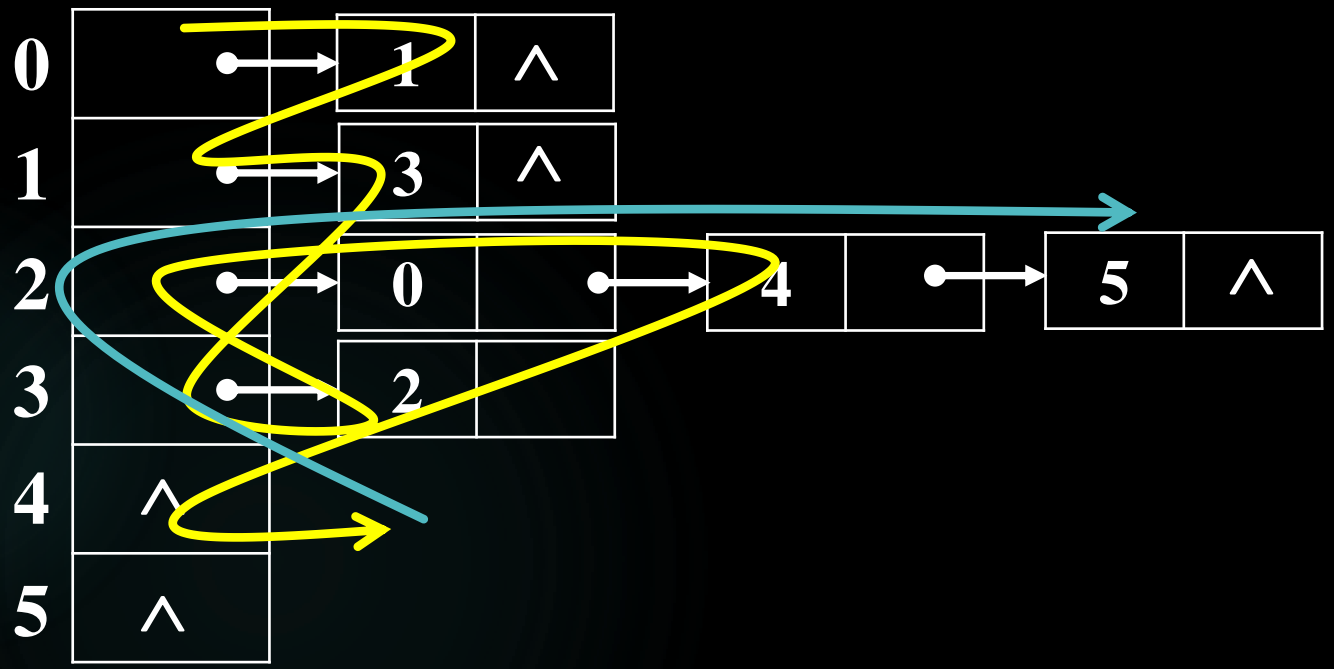
回退(w_t)

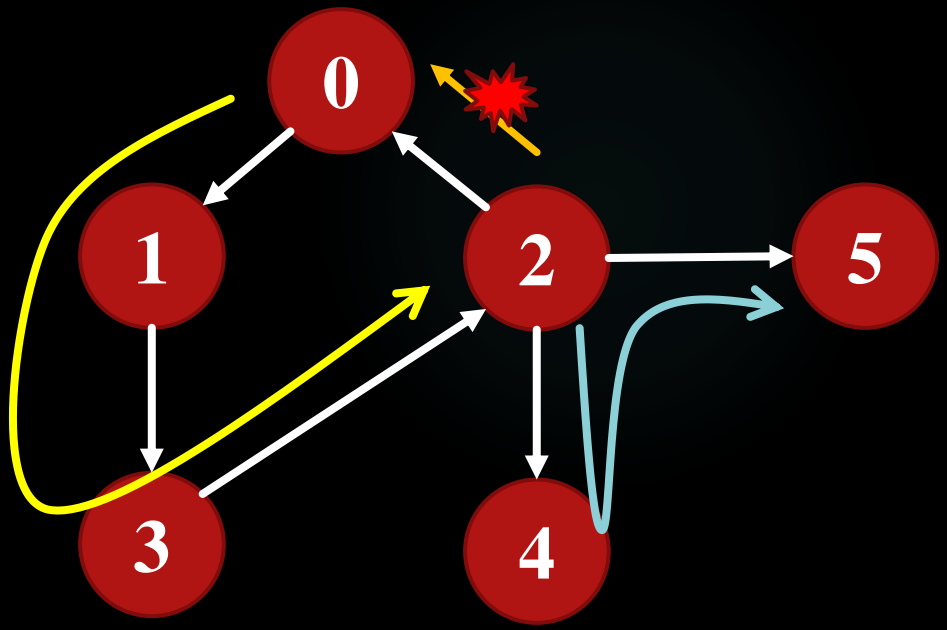
从w_t按刚才的访问路径退到w_{t-1}，若w_{t-1}存在未访问邻接顶点u，**前进(u)**；否则继续回退一步，直到**v所属最大连通子图**中所有顶点都被访问为止

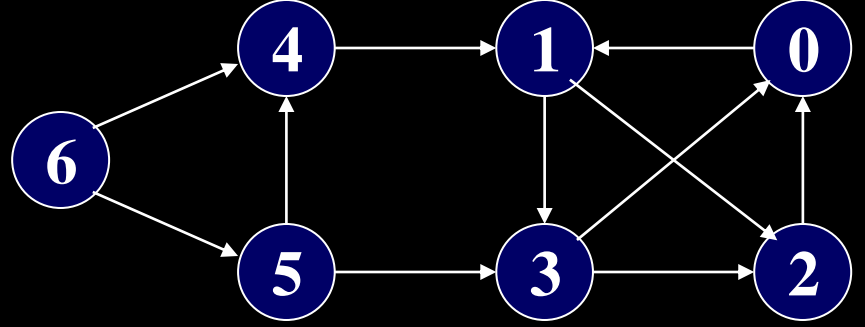
选择图中未被访问的顶点，开始下一次遍历

无向连通图上进行深度遍历，所访问的边和所有顶点组成生成树

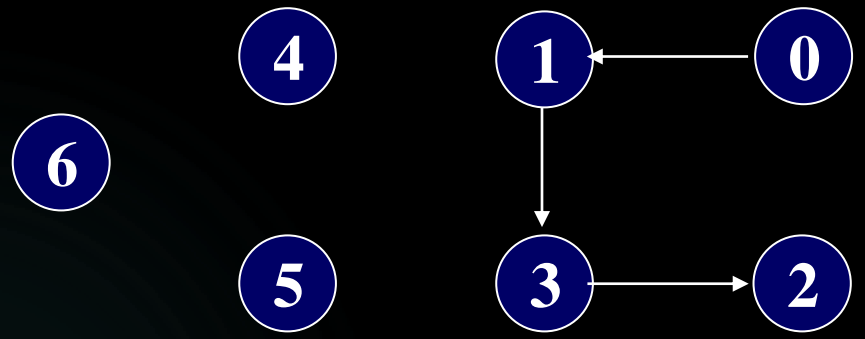
深度优先遍历



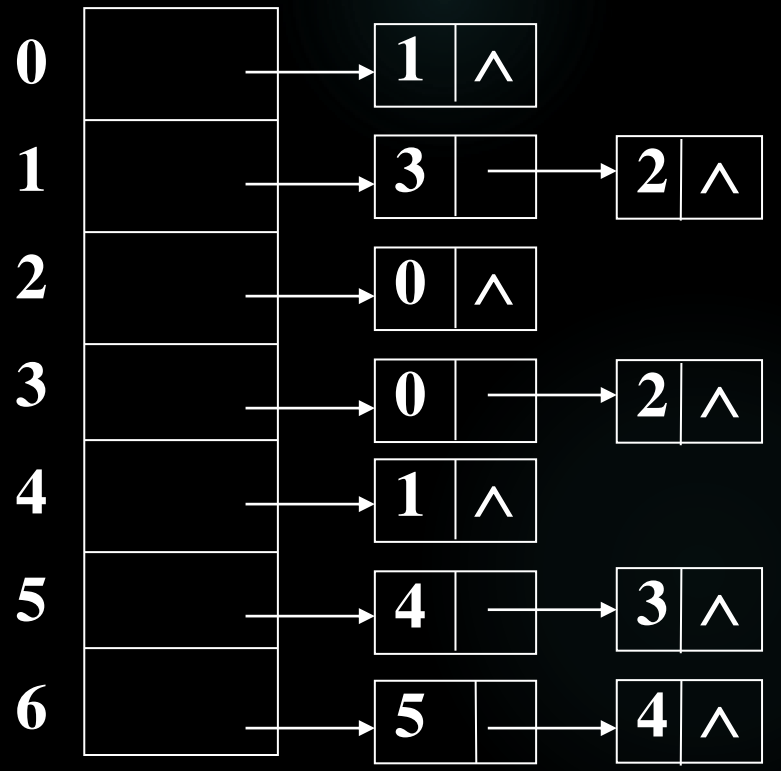




(a) 有向图G



(c) 图G深度优先搜索的生成森林



(b) 图G邻接表

对有向图G，从0出发DFS，访问的次序是0,1,3,2;

另选一个未访问的顶点出发搜索图G的其余部分；结果得到一个生成森林。

思考:1.图的DFS序列是否唯一？(指从同一顶点出发)

2.一次DFS的结果是什么。(**无向图，有向图**)

图的DFS序列是否唯一？(指从同一顶点出发)

图的DFS序列是否唯一？(指从同一顶点出发)
取决于邻接表

深度优先遍历的递归算法

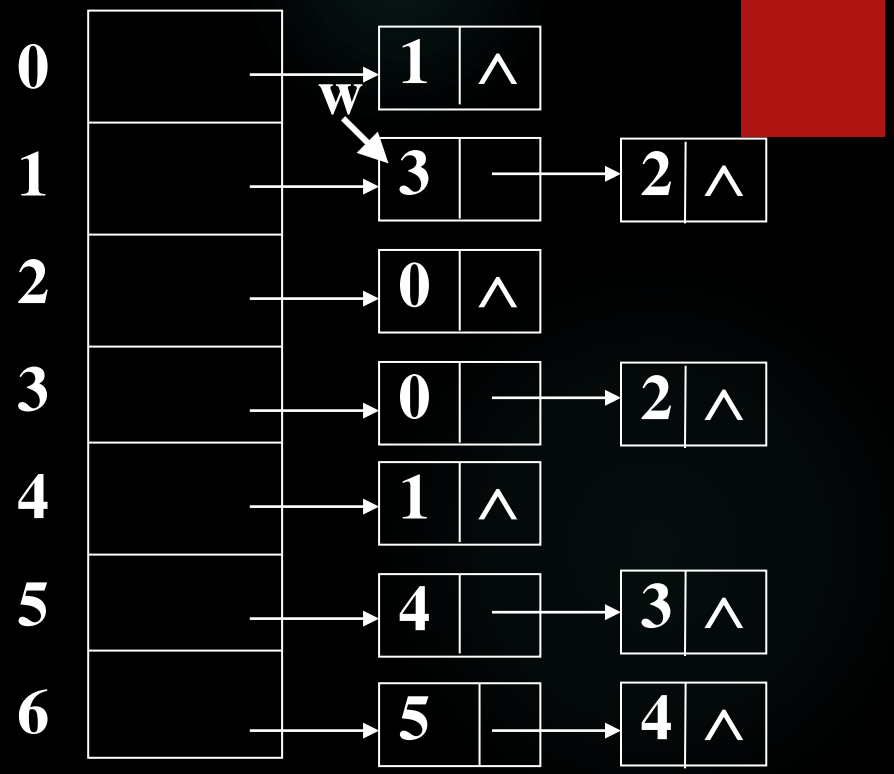
```
void Traversal_DFS(Graph g)
{   BOOL visited[MaxSize]; int i, n = g.Vertices;
    for(i=0;i<n;i++)
        visited[i]=FALSE;
    for (i=0;i<n;i++)
            if (!visited[i]) DFS(g, i, visited);
}
```

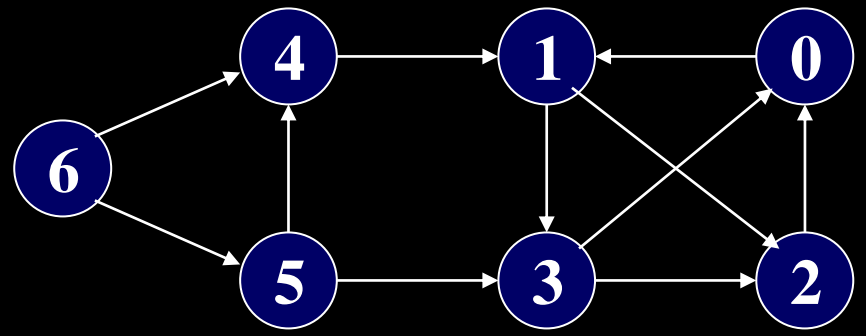
初始化visited

调用递归函数DFS

```
void DFS(Graph g, int v, BOOL* visited)
{   visited[v]=TRUE;
    printf("%d ",v);
    ENode<T> *w;
    for (w=g.A[v]; w; w=w->NextArc)
      if (!visited[w->AdjVex])
          DFS(g, w->AdjVex,visited);}
```

标记当前顶点被访问

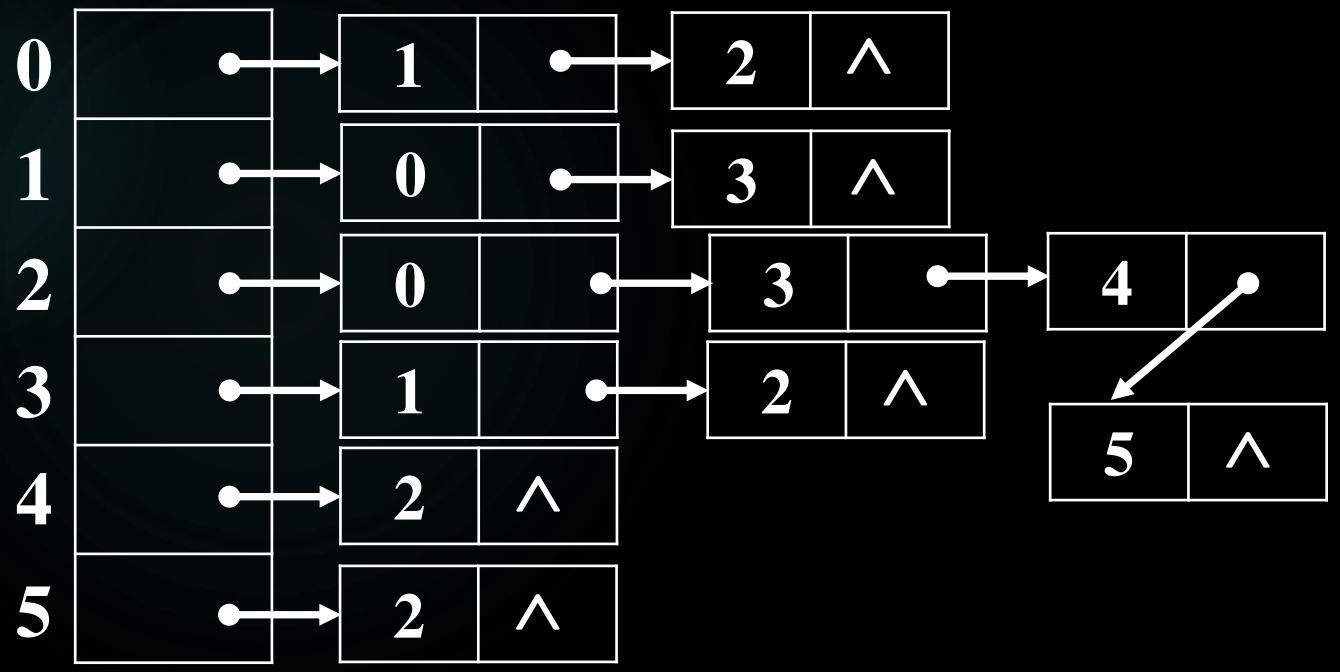


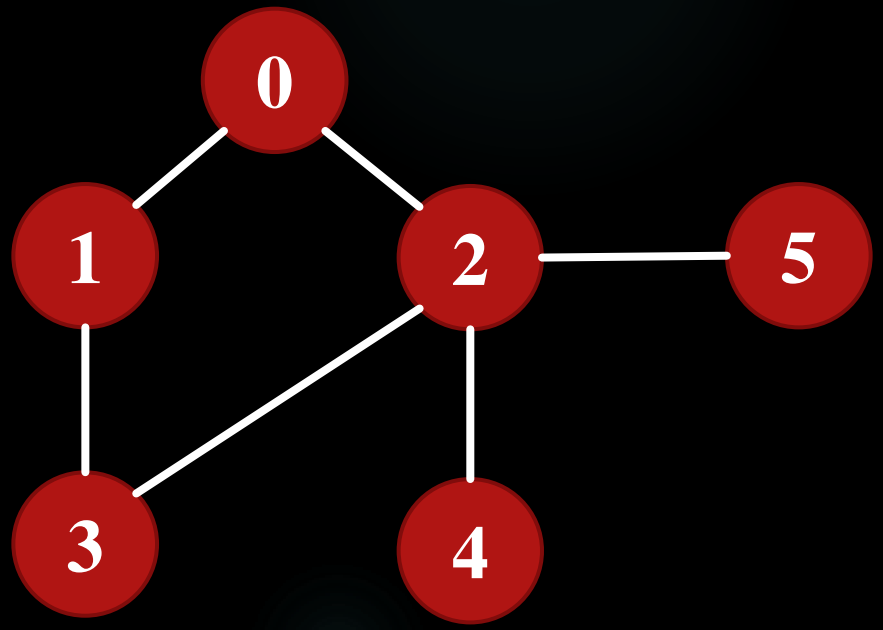


总结：

1 从上面深度优先搜索算法的描述中可以看出，从递归函数DFS外部调用该递归函数与在递归函数内部自己调用自己，所起作用不同。

2 深度优先搜索算法对有向图的每条边只查看1次，而对于无向图，查看2次。

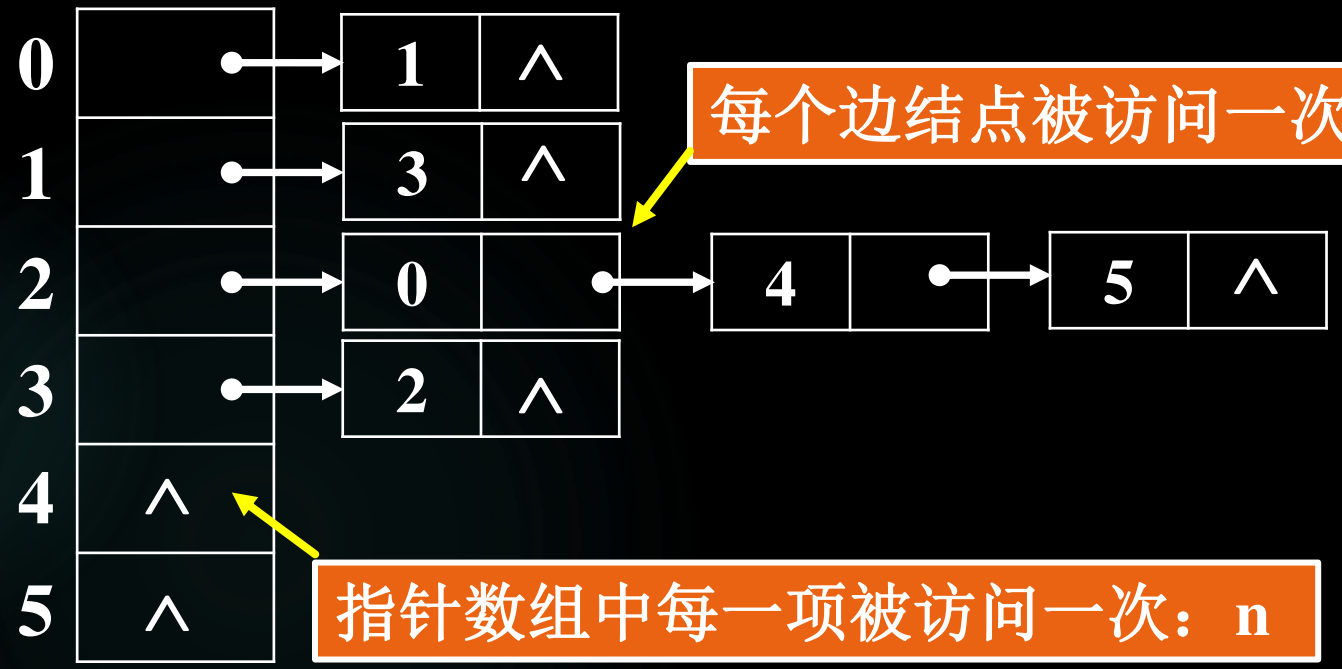


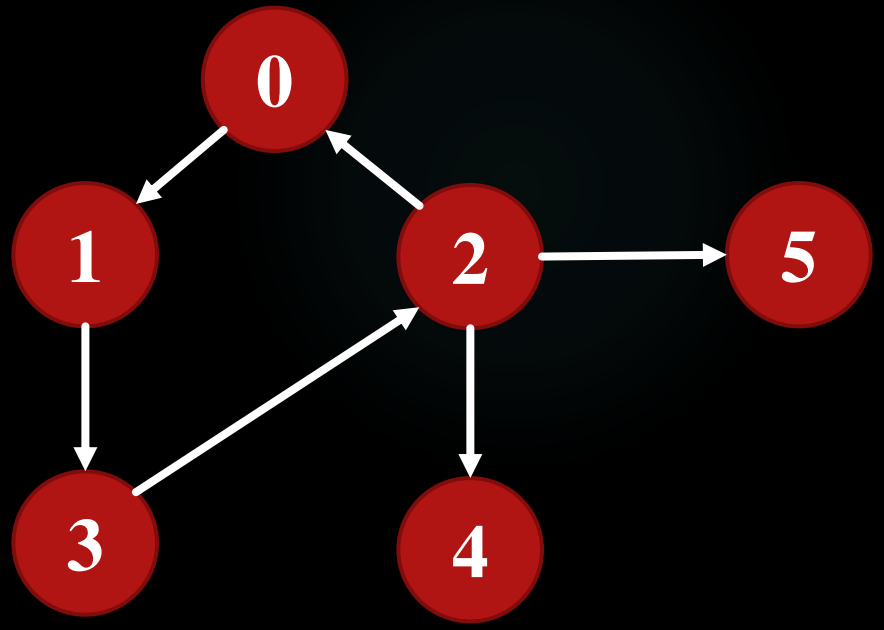


总结：

3 n个顶点、e条边的图采用邻接表存储，时间为O(n+e)，而采用邻接矩阵表示，时间为$O(n^2)$。

邻接矩阵中的每一项都需要被访问一次，以确定两个顶点之间是否有边：n^2





4 图中顶点以及遍历时生成的边所构成的子图称为深度优先搜索生成树。

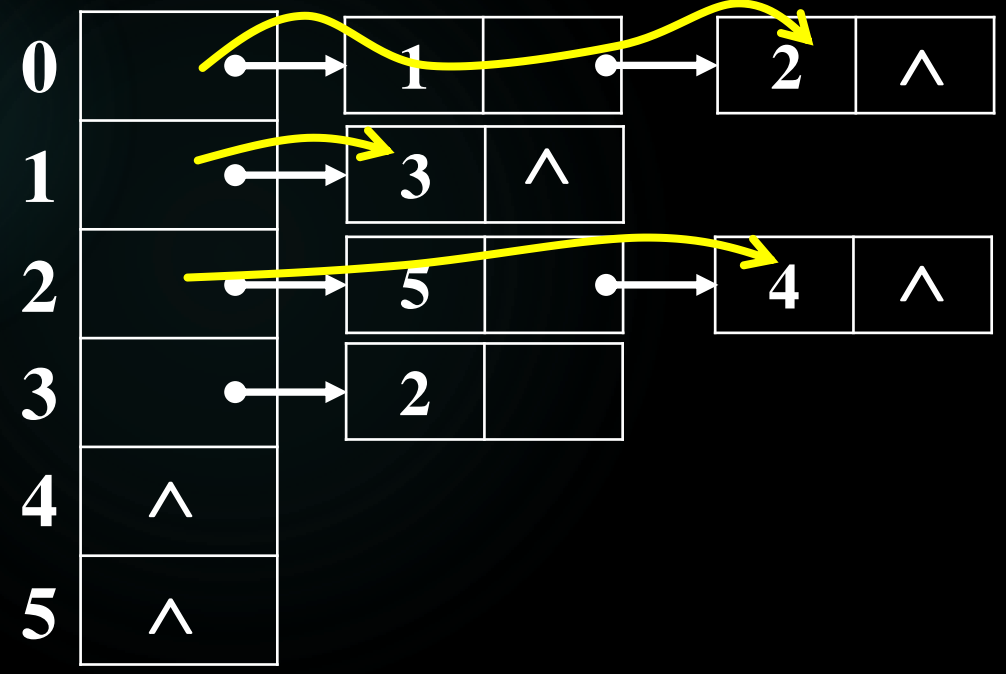
宽度优先遍历

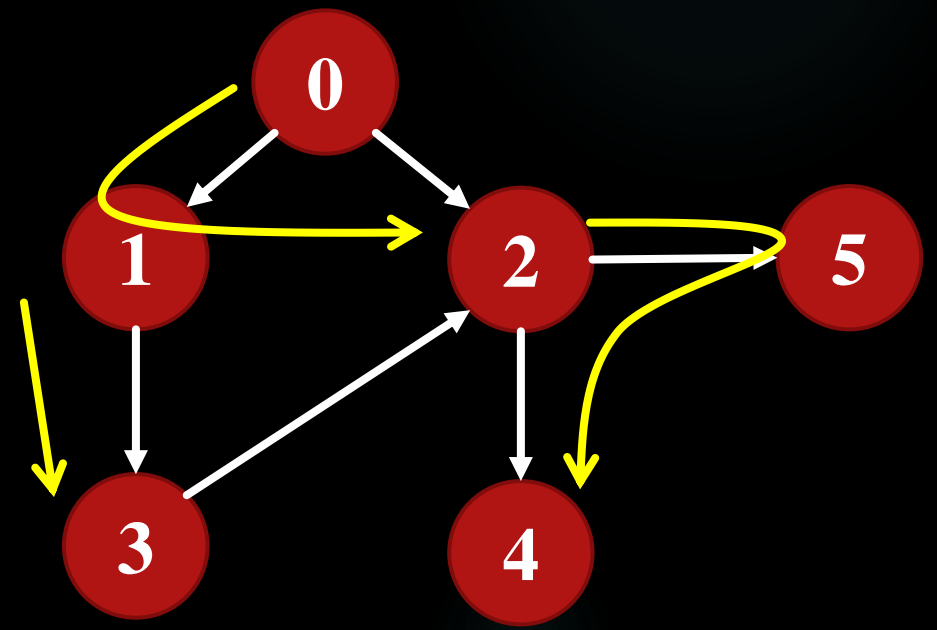
从图中任一顶点v出发遍历图的BFS算法的描述:

1 访问顶点v并打上标记;

2 依次访问顶点v的所有未访问的邻接点，再访问与这些邻接点相邻接且未访问的顶点。

3 若图中还有顶点未被访问，则另选一个未访问的顶点，重新开始上述过程。





宽度优先遍历

从图中任一顶点v出发遍历图的BFS算法的描述:

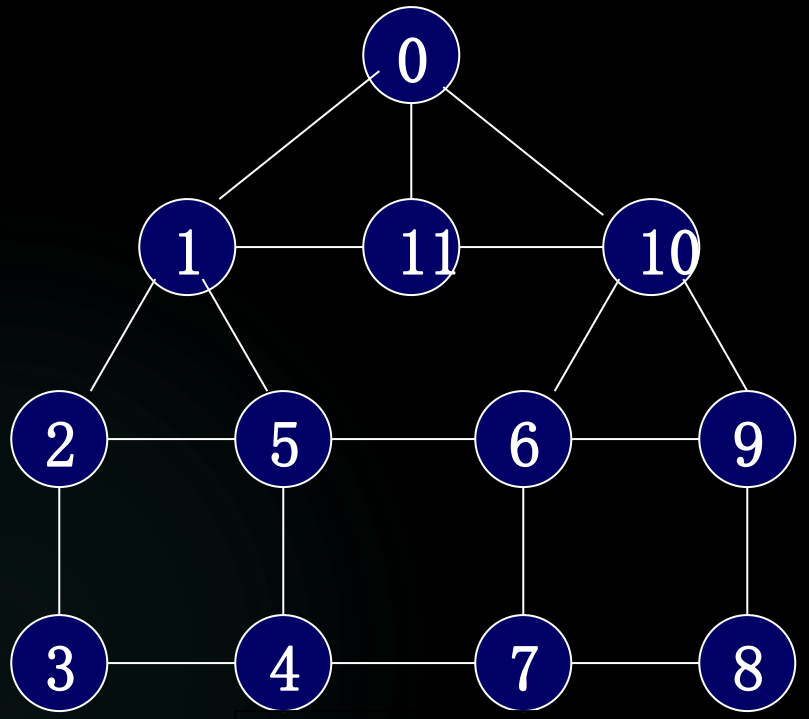
1 访问顶点v并打上标记;

2 依次访问顶点v的所有未访问的邻接点，再访问与这些邻接点相邻接且未访问的顶点。

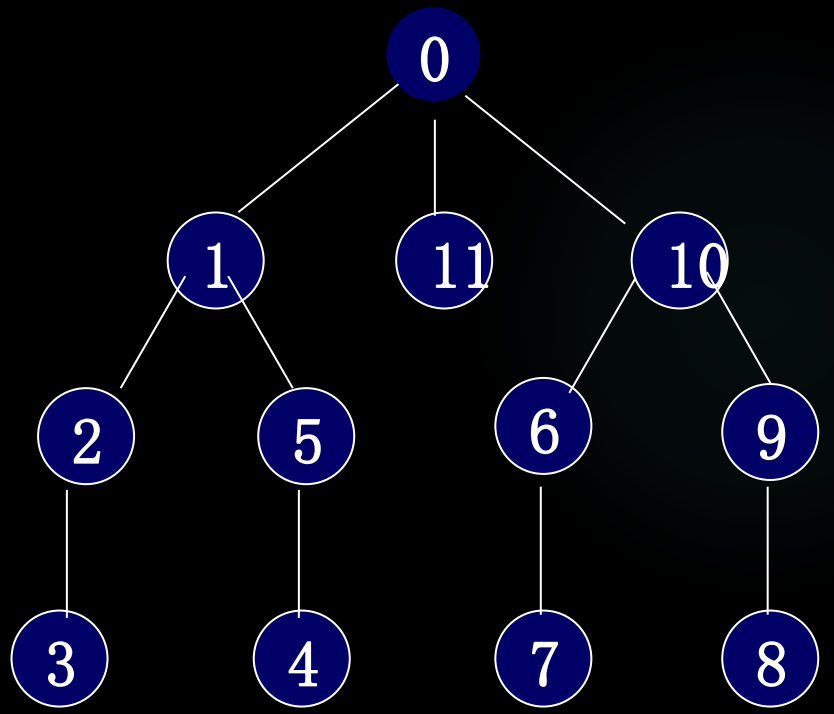
3 若图中还有顶点未被访问，则另选一个未访问的顶点，重新开始上述过程。

宽度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有往回退的情况。因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。

例： 对下图，从顶点0出发BFS遍历，其遍历序列是：
0,1,11,10,2,5,6,9,3,4,7,8。



(a) 无向图G



(b) 图G的宽度优先搜索生成树

图的宽度优先搜索

需要队列保存已访问过但其邻接点未考察的顶点。

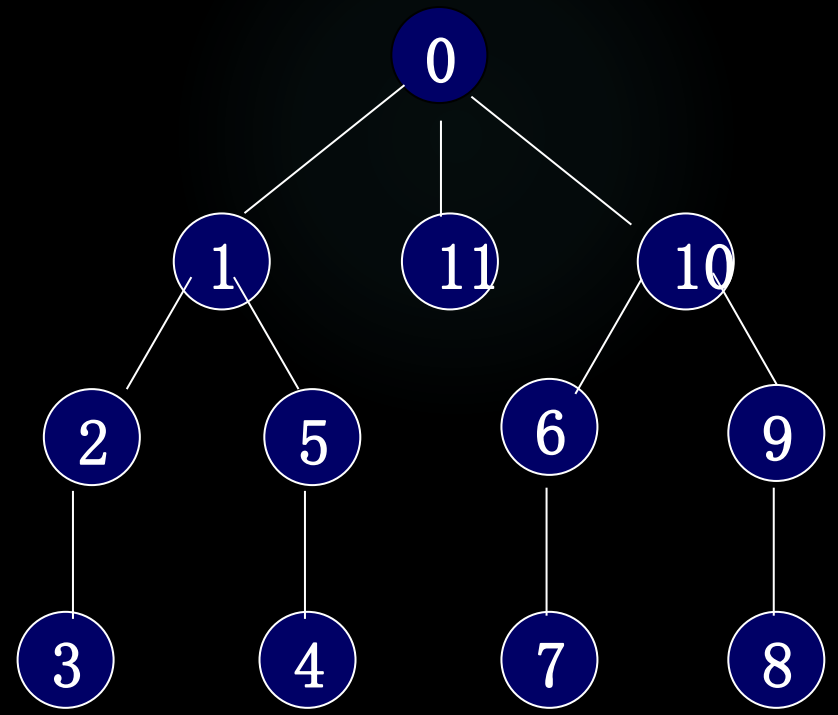
(1)访问0，0入队→(0)。

(2)0出队,访问与0相关联的未访问顶点，访问一个入队一个→(1,11,10)。

(3)重复，直到队列空。

若图中还有顶点未被访问，则另选一个未访问的顶点，重新开始上述过程。

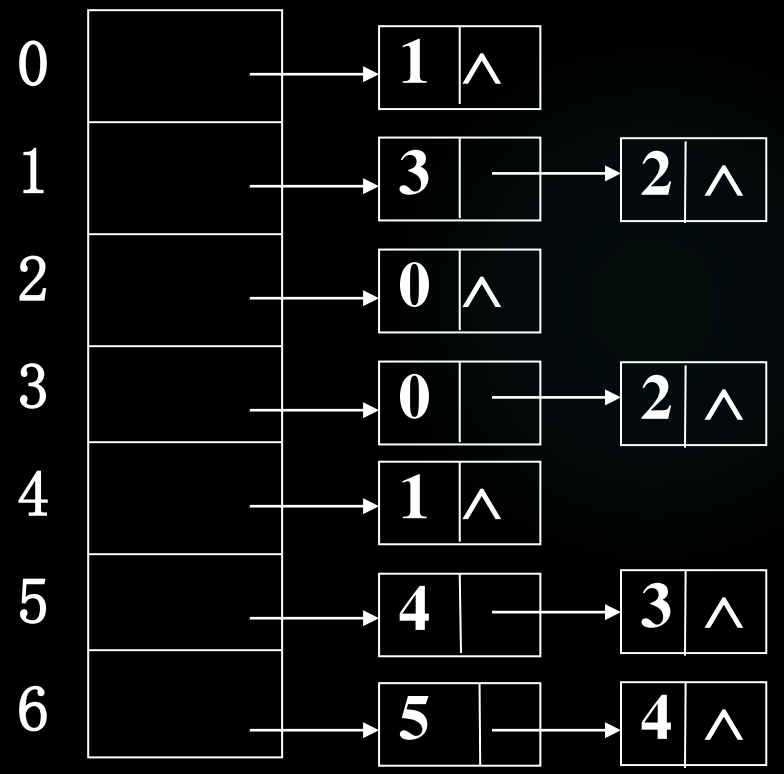
图中顶点以及遍历时生成的边所构成的子图称为宽度优先搜索生成树。



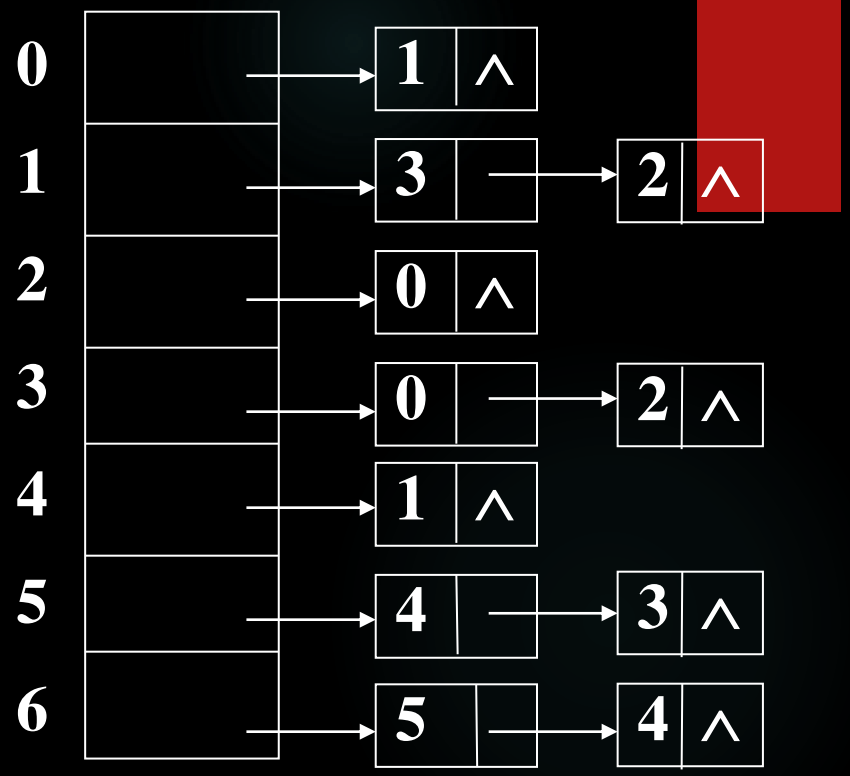
(b) 图G的宽度优先搜索生成树

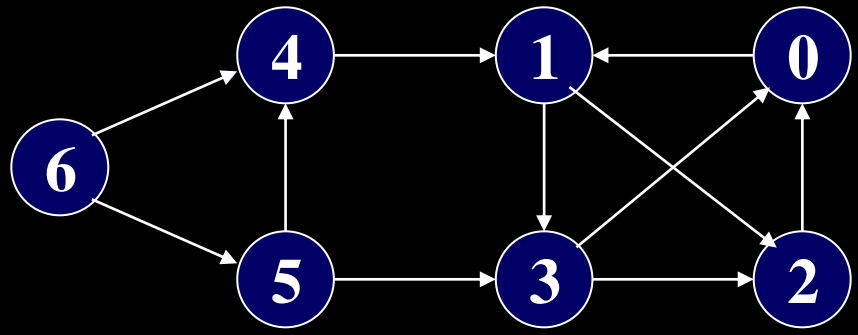
宽度优先搜索的C程序

```
void Traversal_BFS(Graph g)
{
    BOOL visited[MaxSize];
    int i, n = g.Vertices;
    for(i=0;i<n;i++)
        visited[i]=FALSE;
    for (i=0;i<n;i++)
          if (!visited[i])
         BFS(g, i, visited);
}
```



```
void BFS(Graph g, int v, bool* visited)
{  ENode<T> *w;   T u;
   Queue q;
   visited[v]=TRUE;
   printf("%d ", v);
   Append(&q, v);
   while (! IsEmpty(q))
  {  QueueFront(q, &u); Serve(&q);
     for (w=g.A[v];w;w=w->NextArc)
        if (!visited[w->AdjVex])
        {visited[w->AdjVex]=TRUE;
          printf("%d ", w->AdjVex);
         Append(&q,w->AdjVex);}
   }
}
```





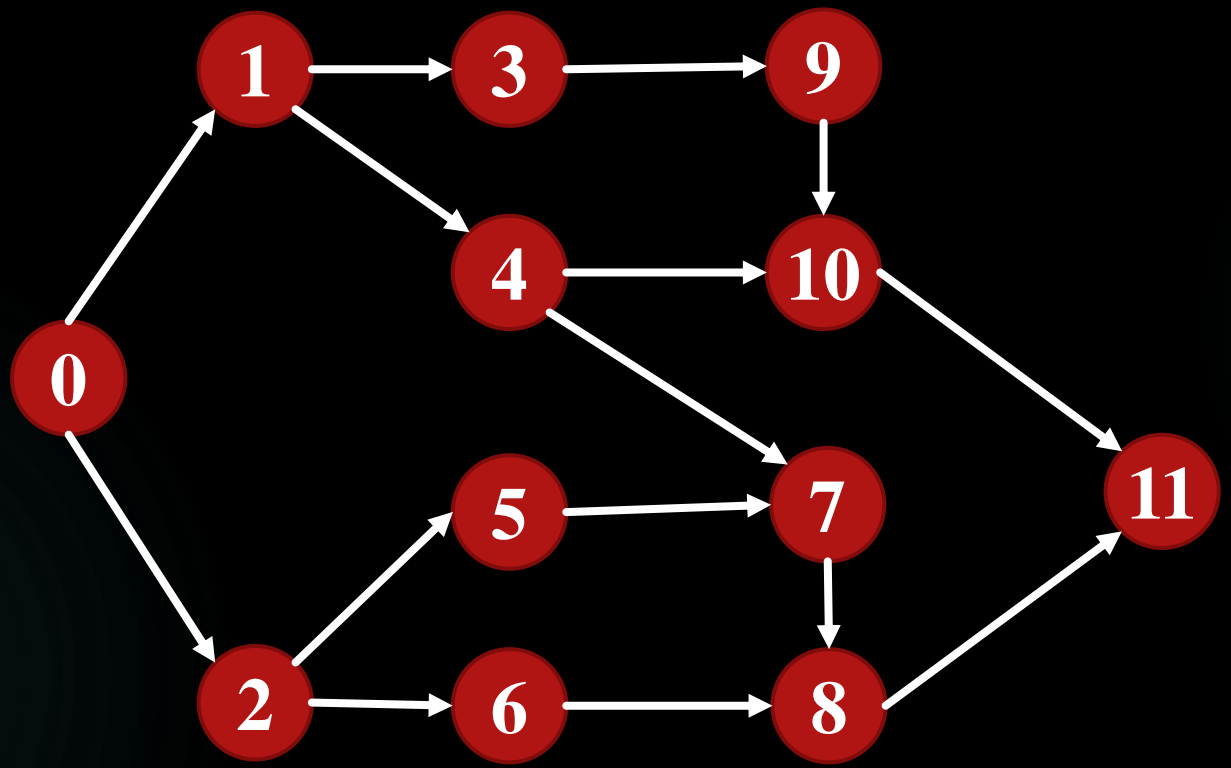
BFS算法的特点：

1、每个顶点进出队列各一次。

2、对于每个出队的顶点，都要检查其所有的邻接点，对于无向图每条边被检查2次。

3、n个顶点，e条边的图采用邻接表存储，BFS算法的时间为O(n+e),而采用邻接矩阵表示，时间为$O(n^2)$。

如同二叉树的遍历算法，图的DFS和BFS算法是最重要、最基本的算法，许多有关图的算法都可以对它们稍加修改得到。例如，求无向图的连通分量、有向图的强连通分量、生成树(森林)等。


1
3
9
4
10
0
5
7
11
2
6
8

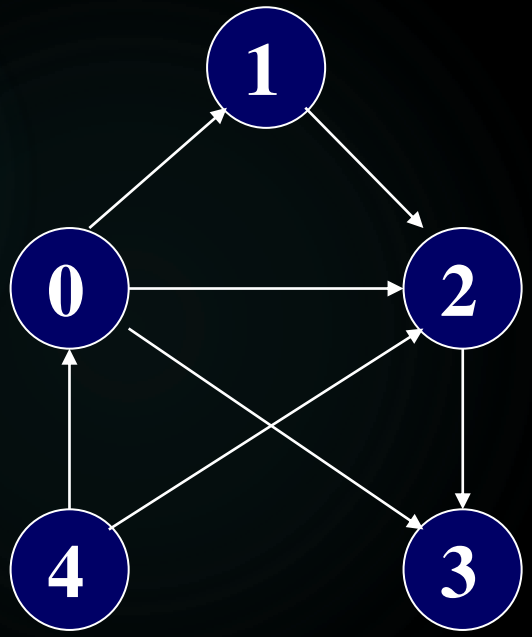
图

目录

- 图的基本概念
- 图的存储结构
- 图的遍历
- 拓扑排序
- 关键路径
- 最小代价生成树：普里姆算法
- 单源最短路径和所有顶点间的最短路径

拓扑排序针对的对象

有向无环图（DAG图）：不包含回路的有向图

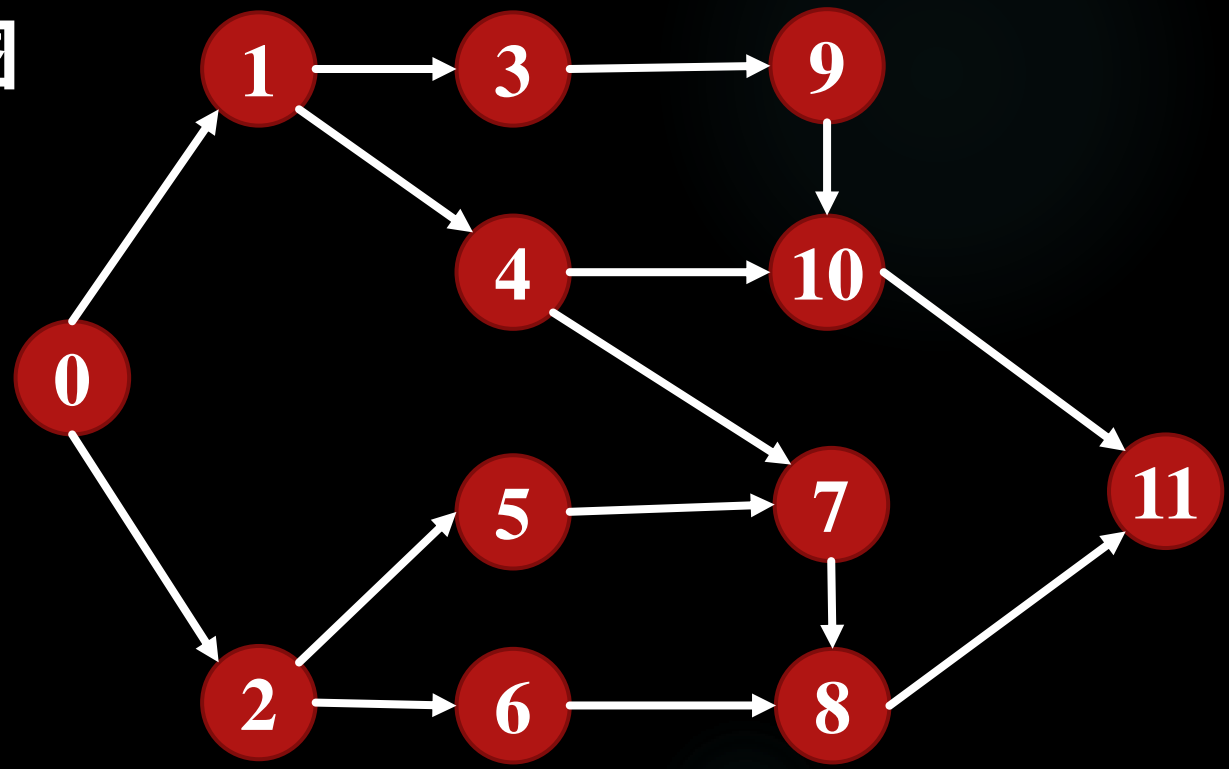


如何判断有向图有没有回路？

不断删除入度为0的顶点，能够最终将图中所有顶点删除

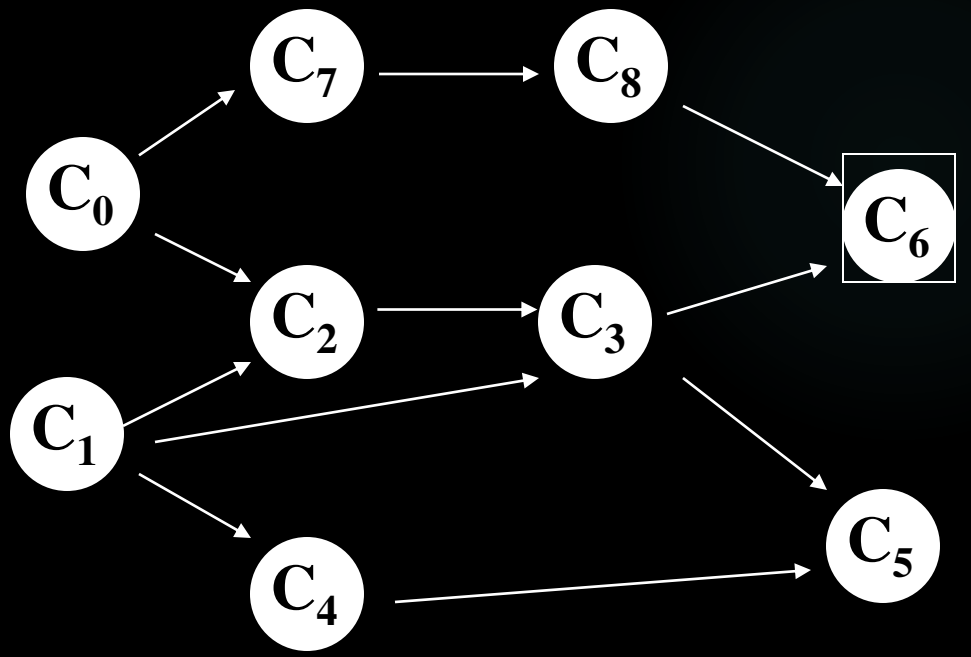
DAG图的应用：AOV网（顶点活动网络）

- 一个工程可以分成若干子工程(**活动**)，活动之间存在领先关系
- 用AOV网络描述一个工程中各个活动之间的领先关系
  - AOV网络是一个DAG图
  - 顶点：表示活动
  - 有向边：表示先决条件



AOV网络举例：计算机专业学生学习的课程。

课程代号	课程名称	先修课程
C_0	高等数学	无
C_1	C++	无
C_2	离散数学	C_0，C_1
C_3	数据结构	C_1，C_2
C_4	高级语言	C_1
C_5	编译方法	C_3，C_4
C_6	操作系统	C_3，C_8
C_7	普通物理	C_0
C_8	计算机原理	C_7



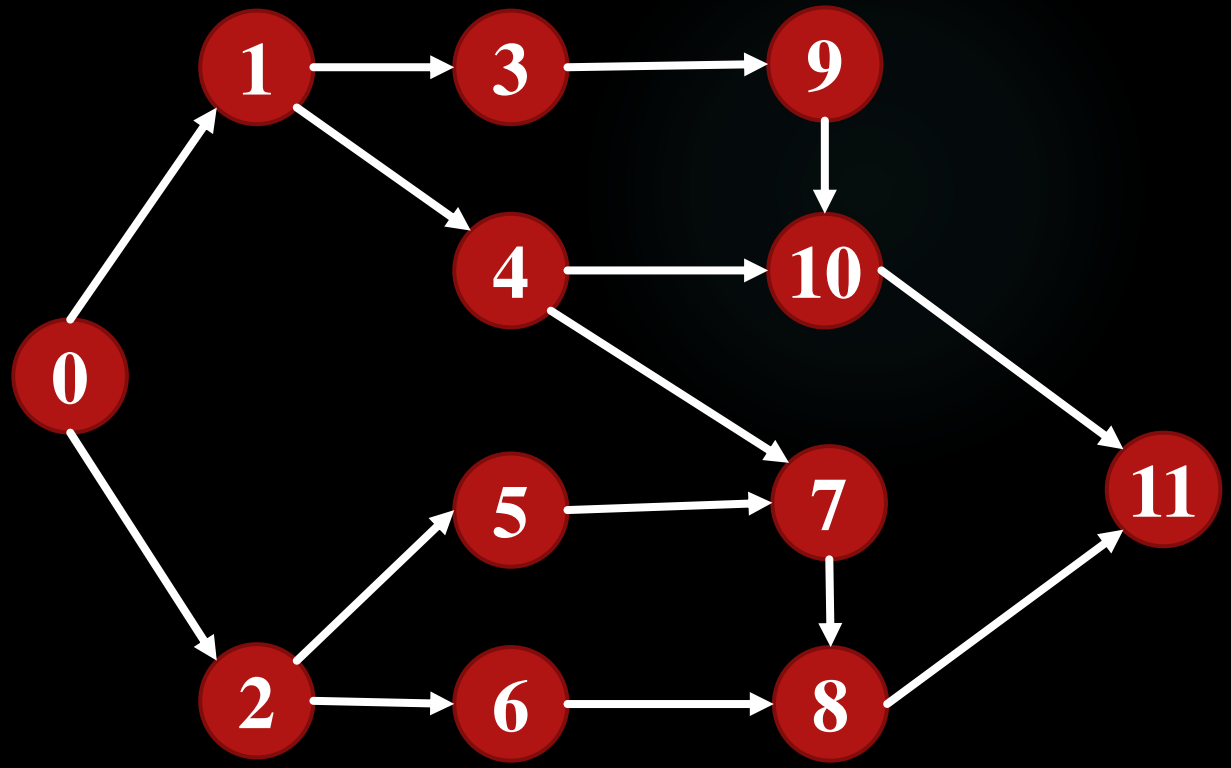
AOV网示例

AOV网中的领先关系是一种拟序关系，即具有传递性和反自反性。

传递性：活动的领先关系可传递的

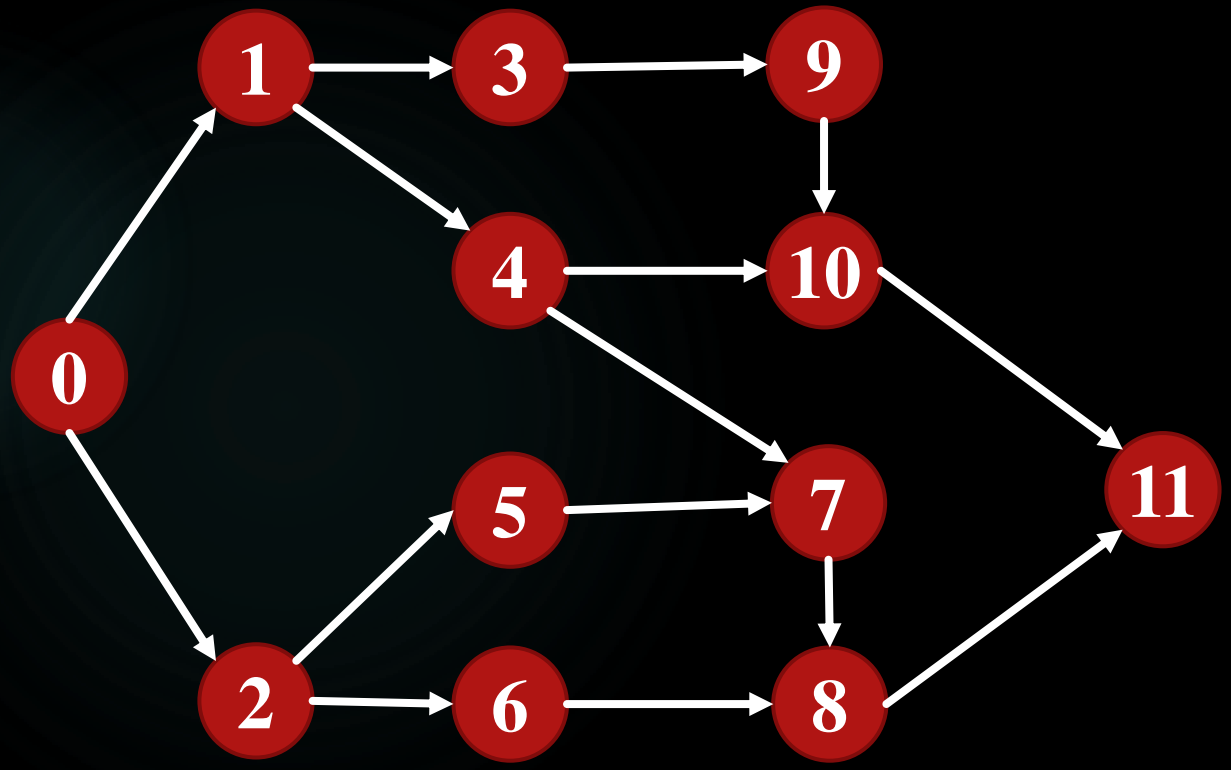
活动1领先于活动4，活动4领先于活动7，
则活动1领先于活动7

反自反性：不允许一个活动在开始之前就完成。要求AOV网是一个有向无回路图



什么是拓扑排序

一个拓扑序列是AOV网中顶点的线性序列，使得对图中任意二个顶点i和j，若在图中，i领先于j，则在线性序列中i是j的前驱结点。



0,1,4,3,2,5,6,7,8,9,10,11
0,1,3,4,9,10,2,6,5,7,8,11

检测拓扑排序的方法

对图中每一条边<i,j>，在序列中i排在j之前

拓扑排序算法

用拓扑排序算法的作用：

（1）测试AOV网的可行性（是否存在回路）

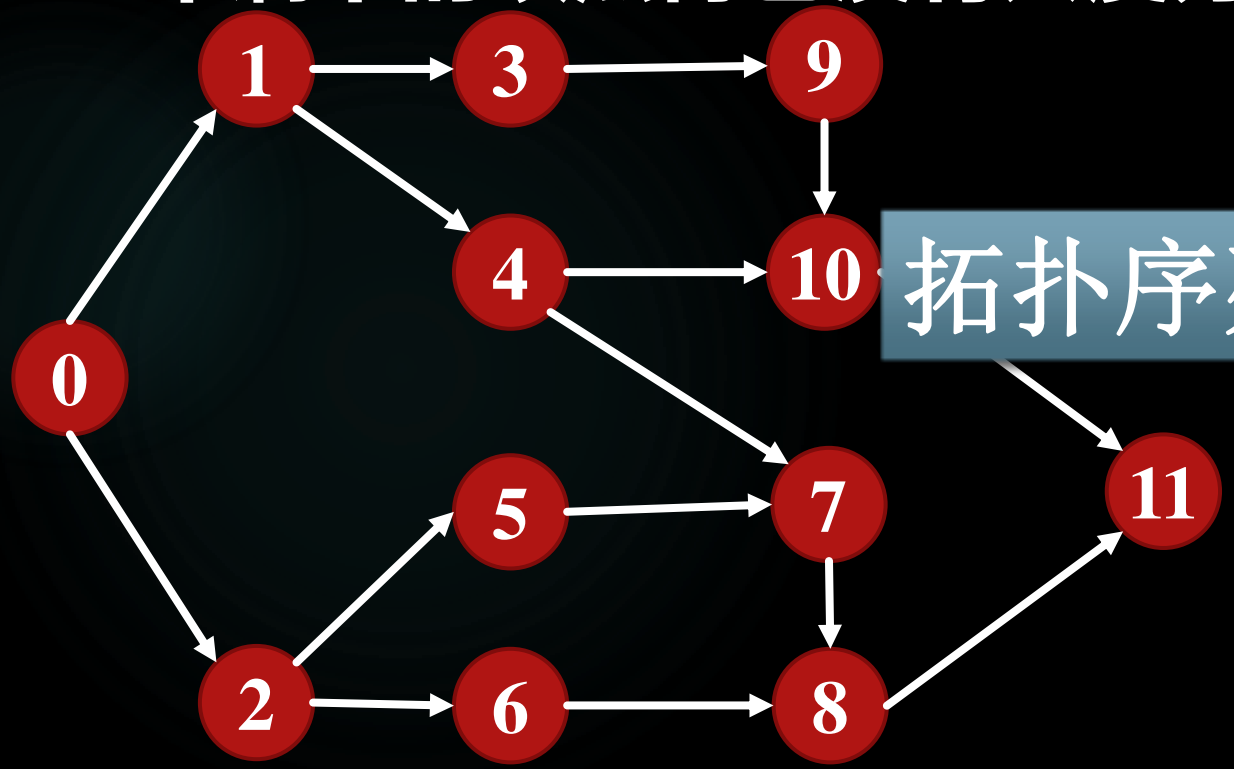
（2）对可行的AOV网求出其拓扑序列。

拓扑排序算法步骤：

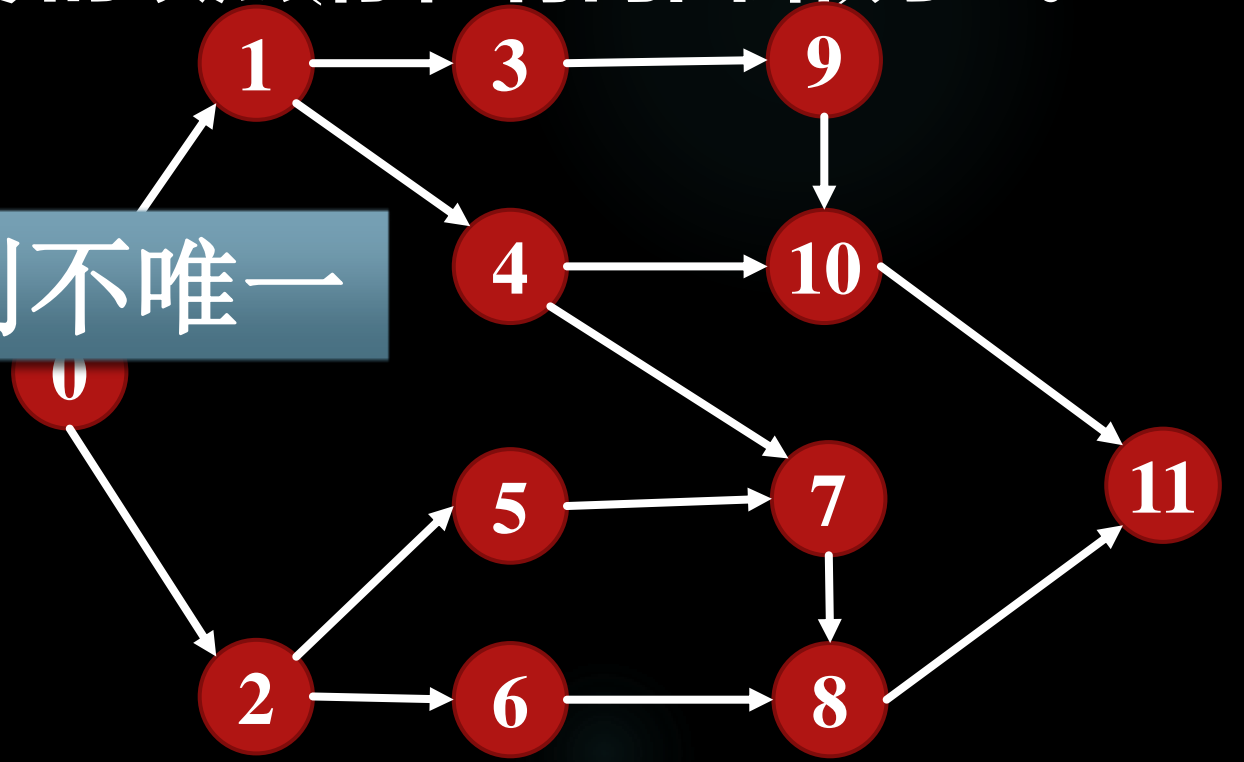
（1）在图中找一个入度为零的顶点，输出之；

（2）从图中删除该顶点及其所有出边；

（3）重复(1)、(2)，直到所有顶点都输出（拓扑序列），或图中剩下的顶点再也没有入度为零的顶点(存在有向回路)为止。



拓扑序列不唯一

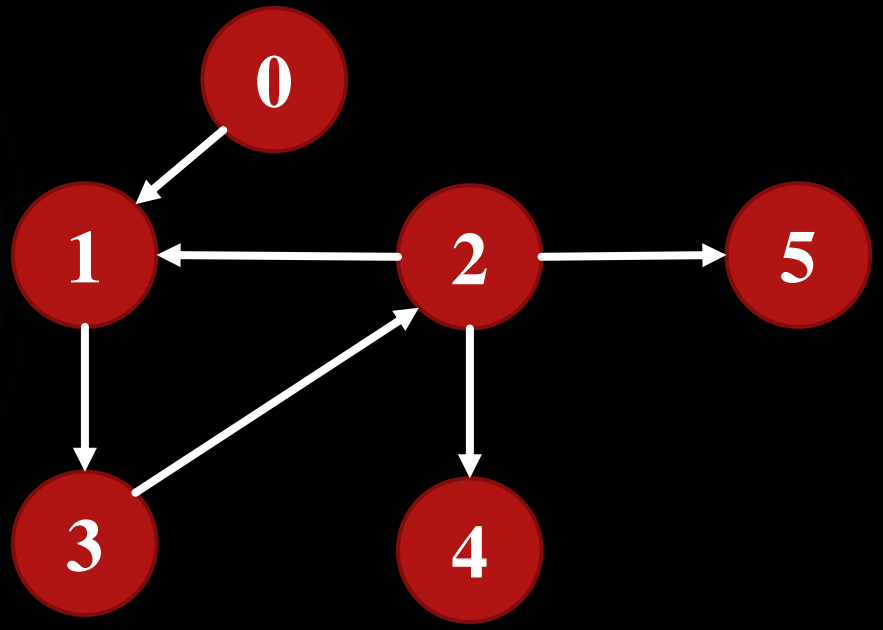


拓扑排序算法步骤：

（1）在图中找一个入度为零的顶点，输出之；

（2）从图中删除该顶点及其所有出边；

（3）重复(1)、(2)，直到所有顶点都输出（拓扑序列），或图中剩下的顶点再也没有入度为零的顶点(存在有向回路)为止。



拓扑排序算法要解决的问题：

（1）如何计算每个顶点的入度？

使用一个数组InDgree保存每个顶点的入度。

（2）如何“删除”一个顶点及其所有出边？

将该顶点的所有邻接到的顶点的入度减一。

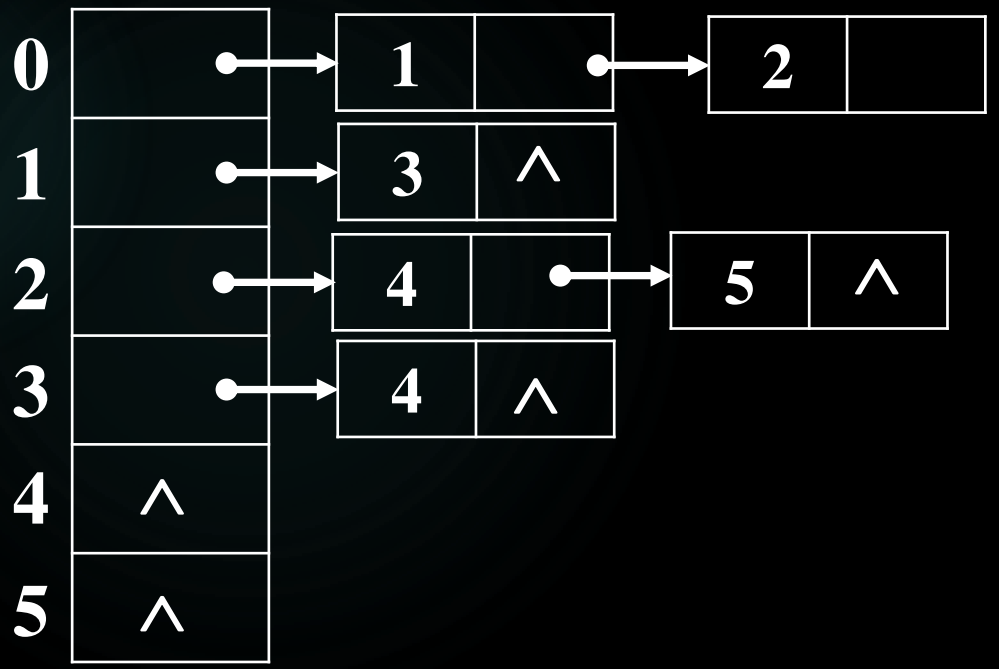
（3）如何保存新产生的入度为0的顶点：

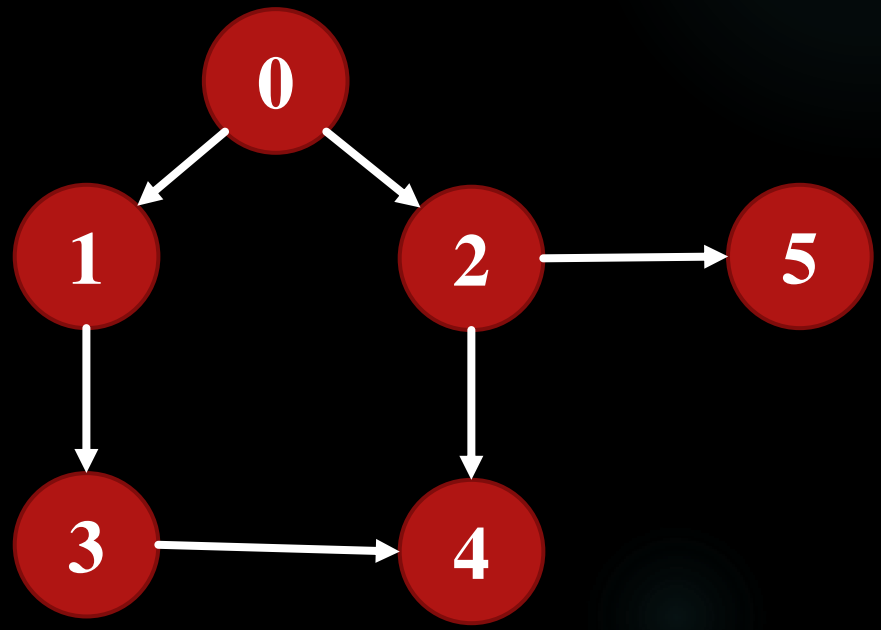
可以用堆栈或队列保存

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

“删除”0及其出边

-1
0
0
1
2
1





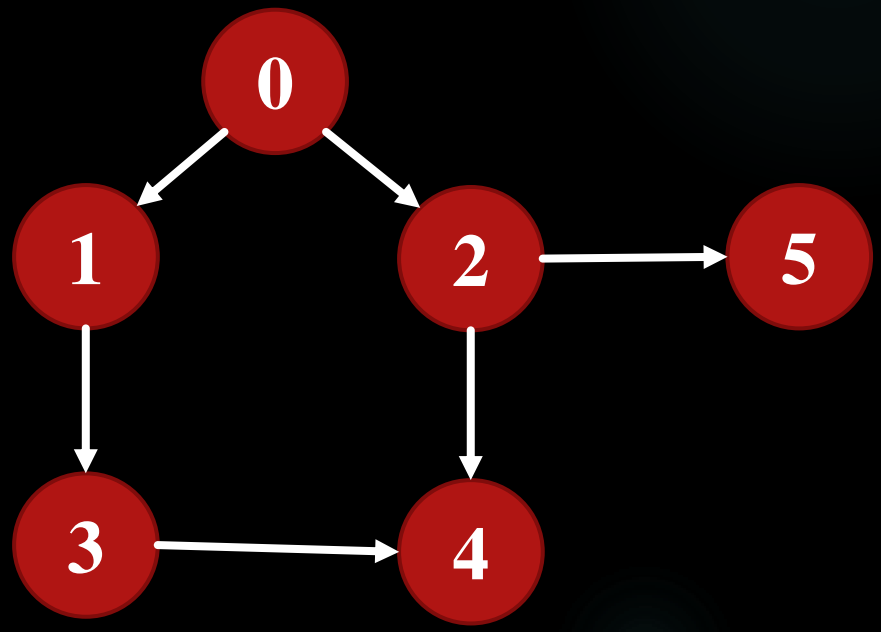
拓扑排序算法

1、设计数据结构

拓扑排序算法采用邻接表存储，用InDegree[i]存储顶点i的入度，数组order被用于保存所求得的一个拓扑序列。

InDegree

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



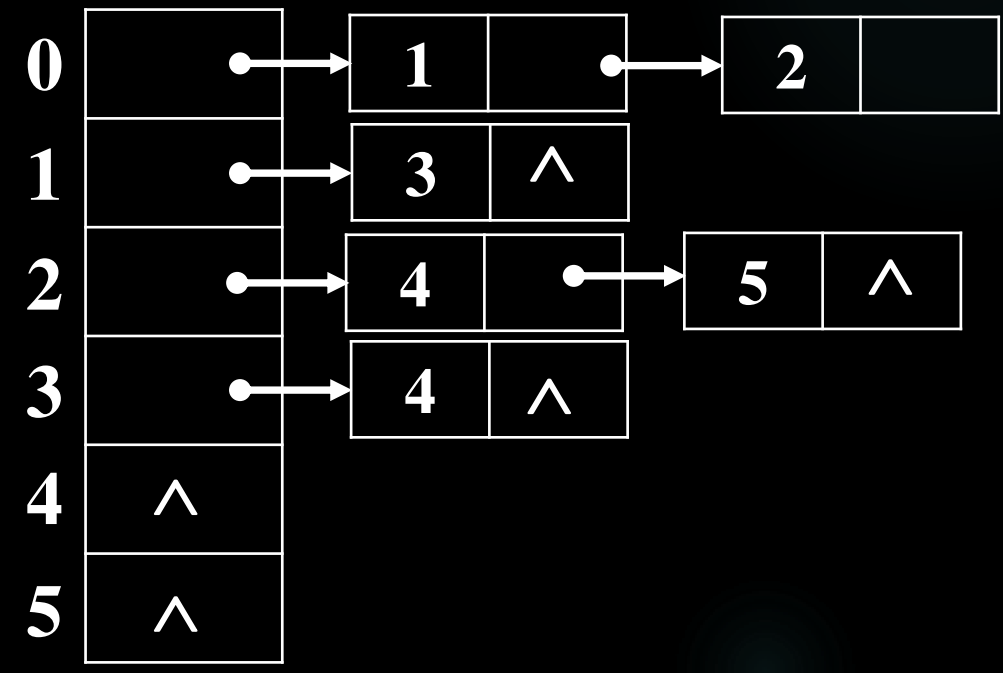
拓扑排序算法

2、算法实现

（1）初始化InDegree数组：扫描邻接表，计算每个顶点入度

InDegree

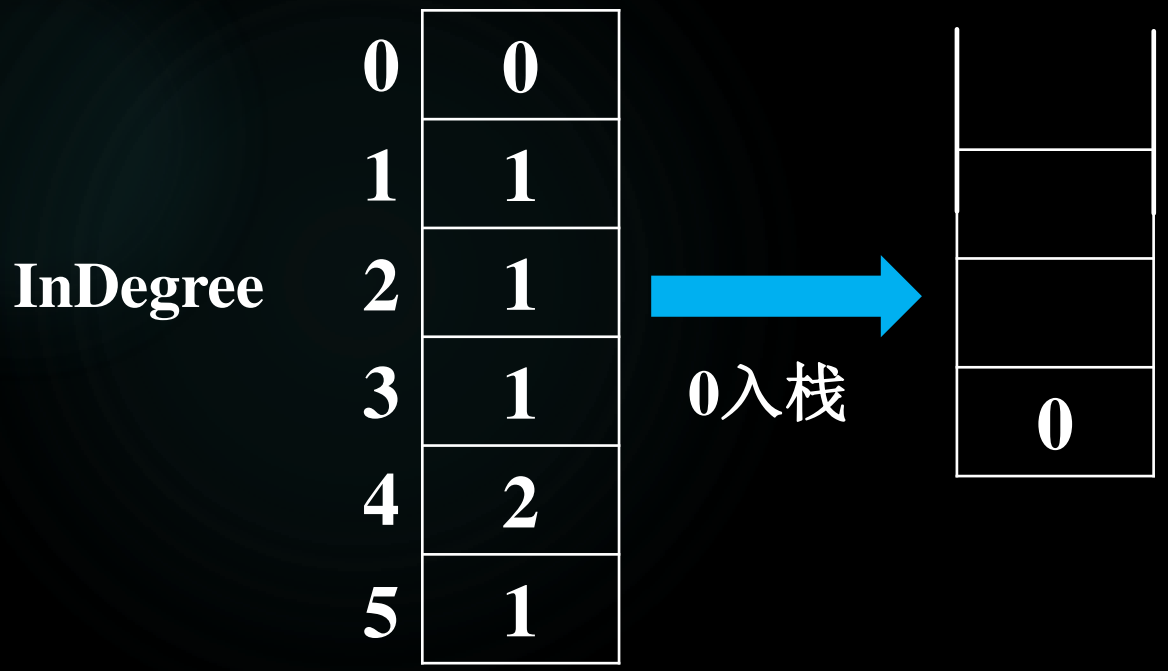
0	0
1	1
2	1
3	1
4	2
5	1



拓扑排序算法

2、算法实现

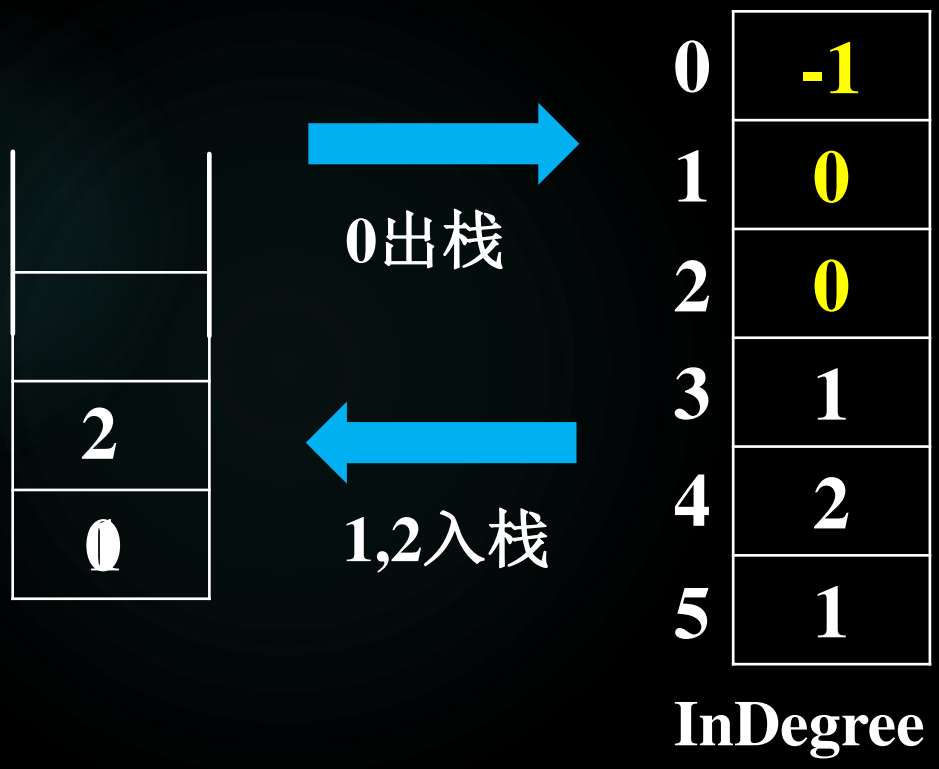
（2）检查InDegree中入度为0的顶点，入栈

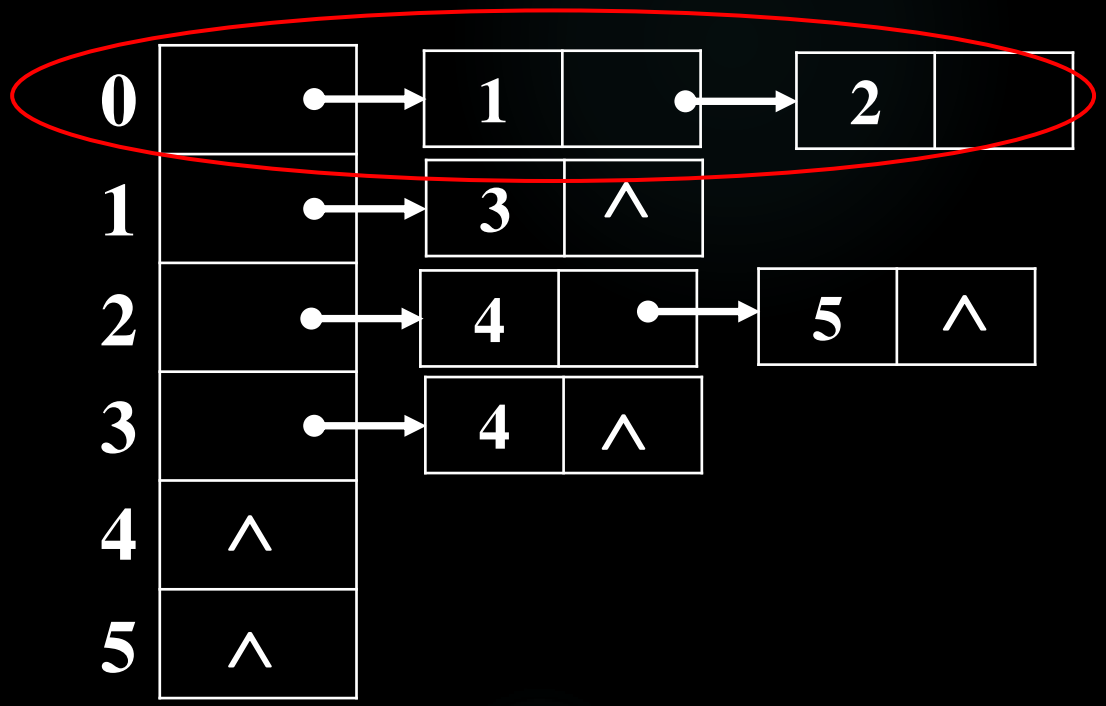


拓扑排序算法

2、算法实现

（3）从堆栈中弹出栈顶元素输出，将栈顶元素及其所邻接的顶点入度减一，同时将入度变成0的顶点入栈

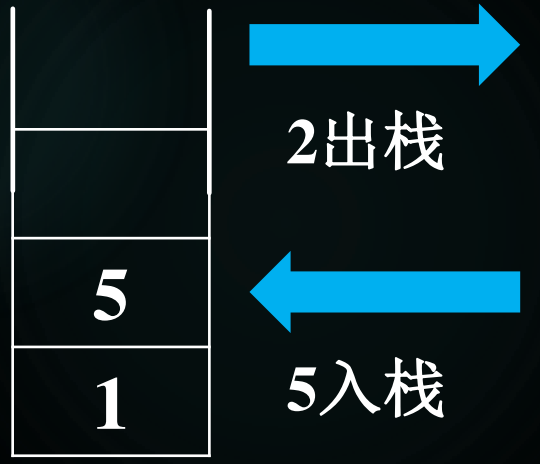




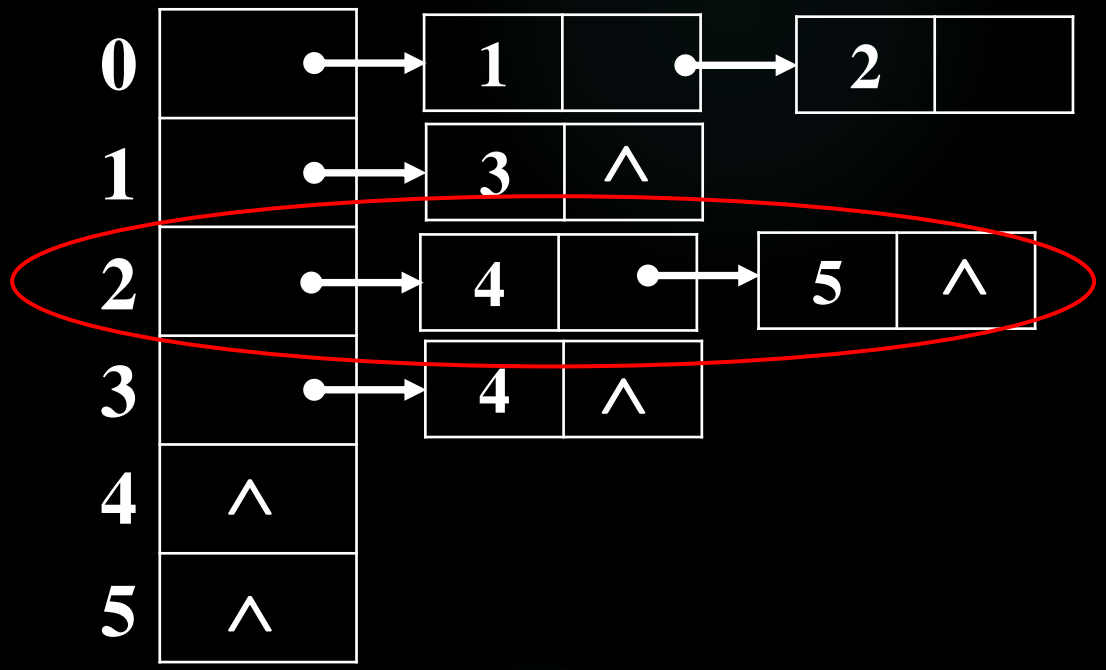
拓扑排序算法

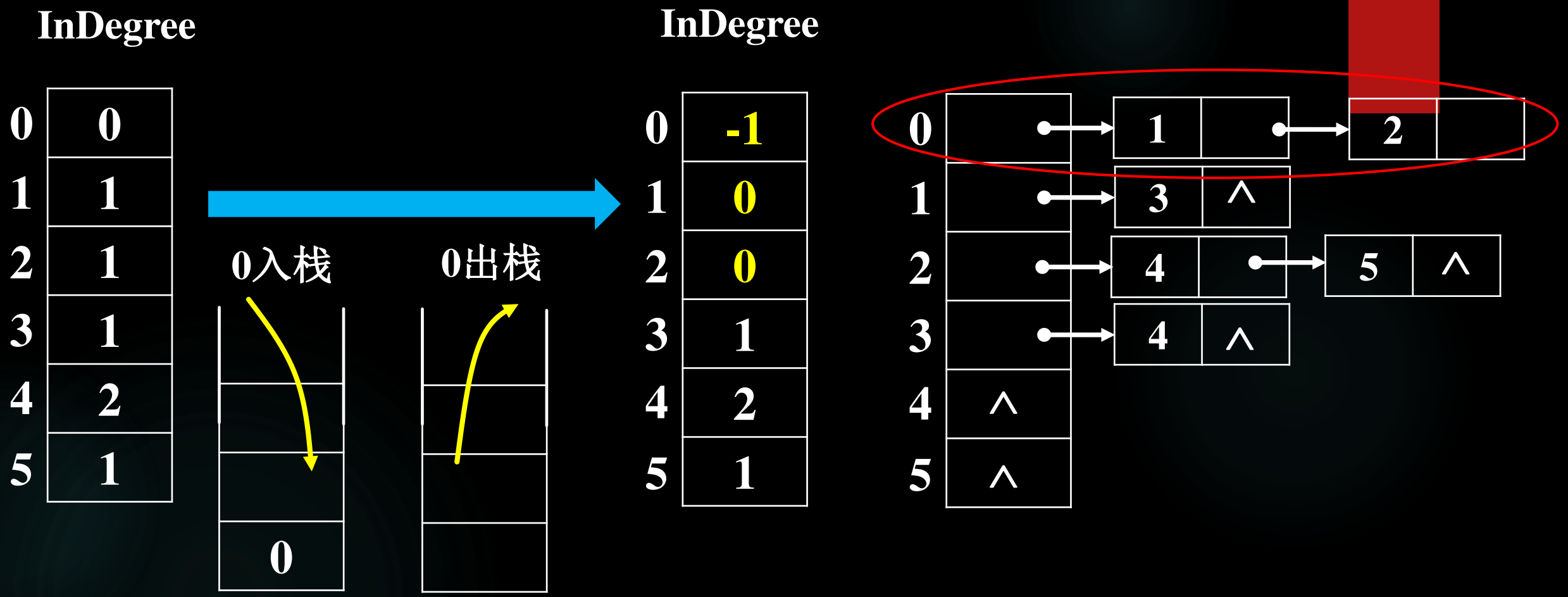
2、算法实现

（4）重复步骤（3）直到栈空为止，若此时所有顶点已经输出，则输出为拓扑序列，否则存在回路

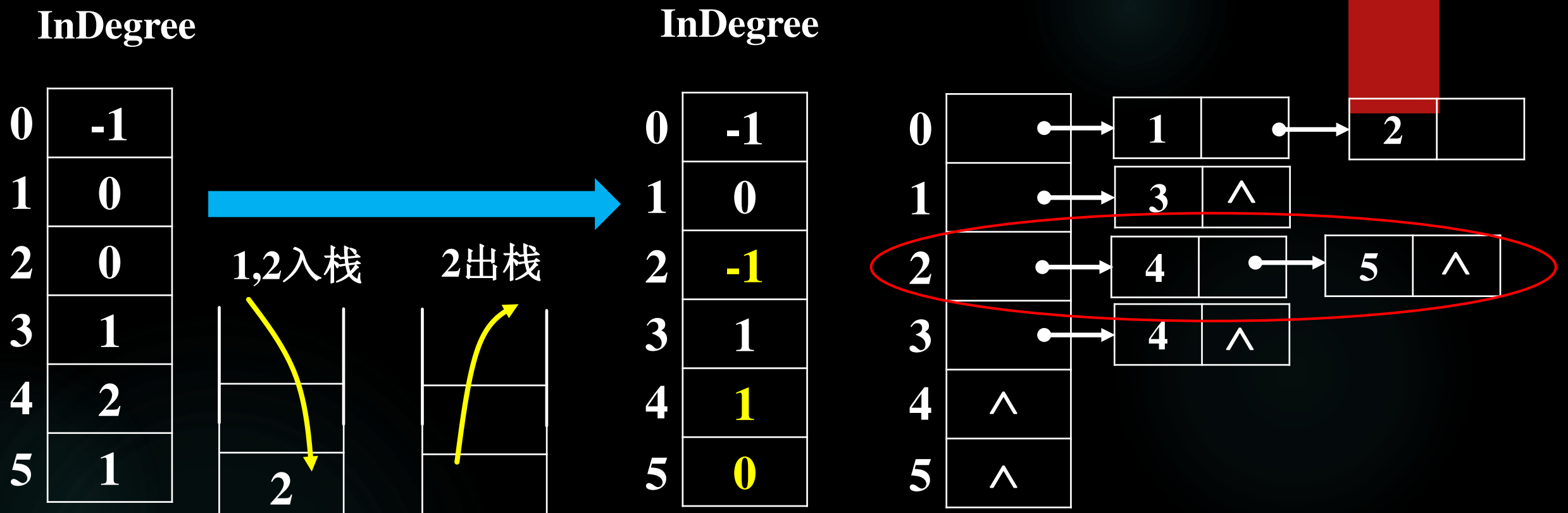


	InDegree
0	-1
1	0
2	-1
3	1
4	1
5	0

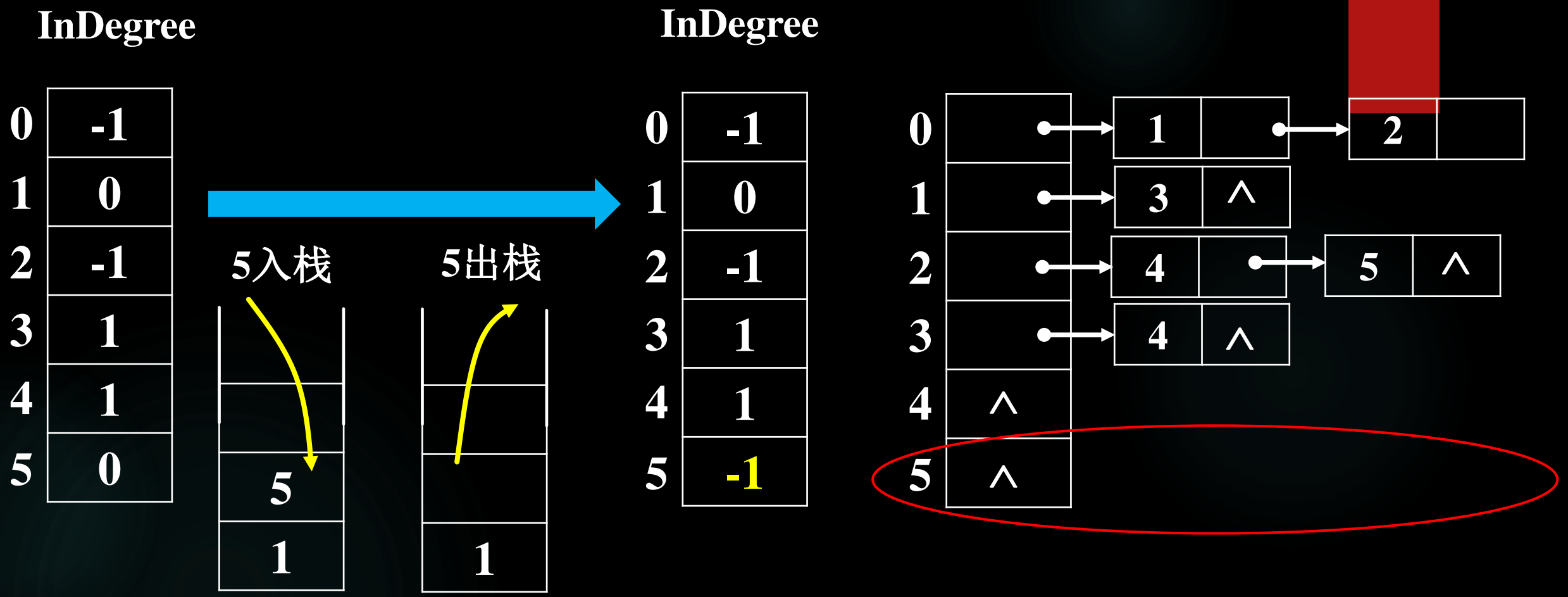



InDegree
0 0
1 1
2 1
3 1
4 2
5 1
0入栈
0出栈
0
InDegree
0 -1
1 0
2 0
3 1
4 2
5 1
0
1
2
1
3 ^
2
4
5 ^
3
4 ^
4 ^
5 ^

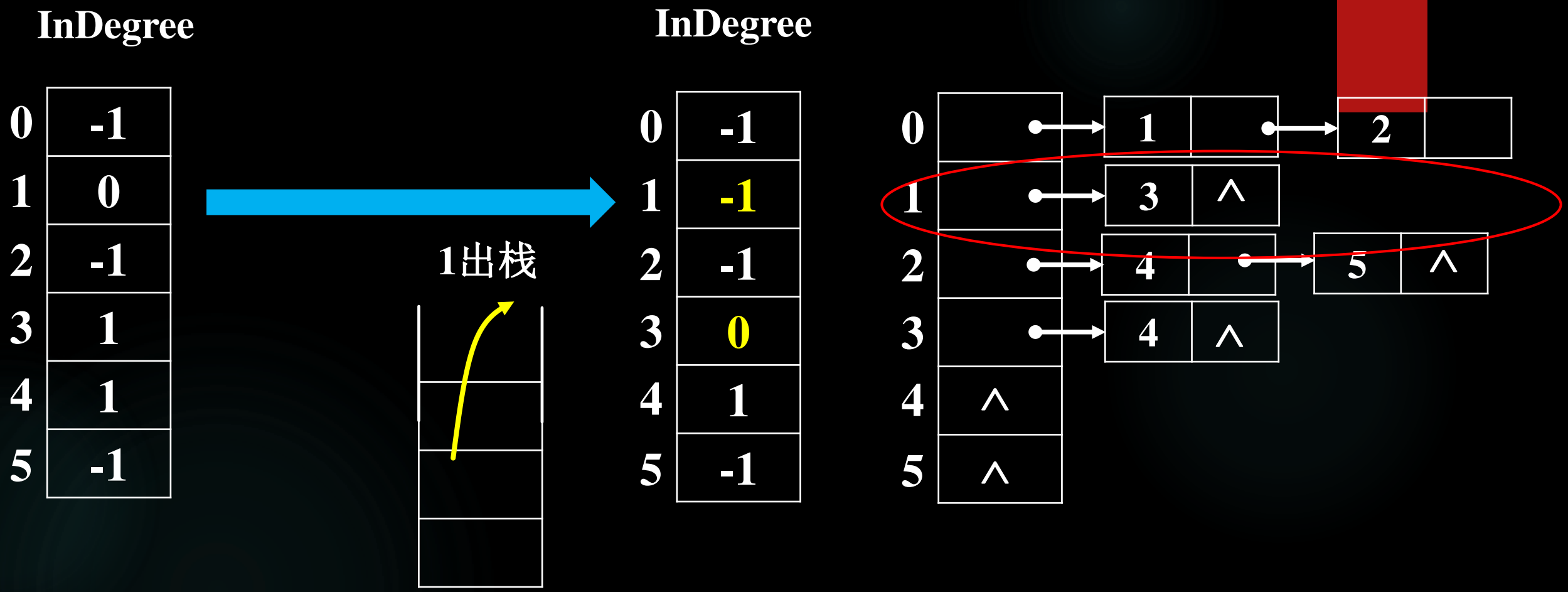
输出：0


InDegree
0 -1
1 0
2 0
3 1
4 2
5 1
1,2入栈
2出栈
2
1
InDegree
0 -1
1 0
2 -1
3 1
4 1
5 0
0
1
2
3
4
5
1
2
3
4
5
4

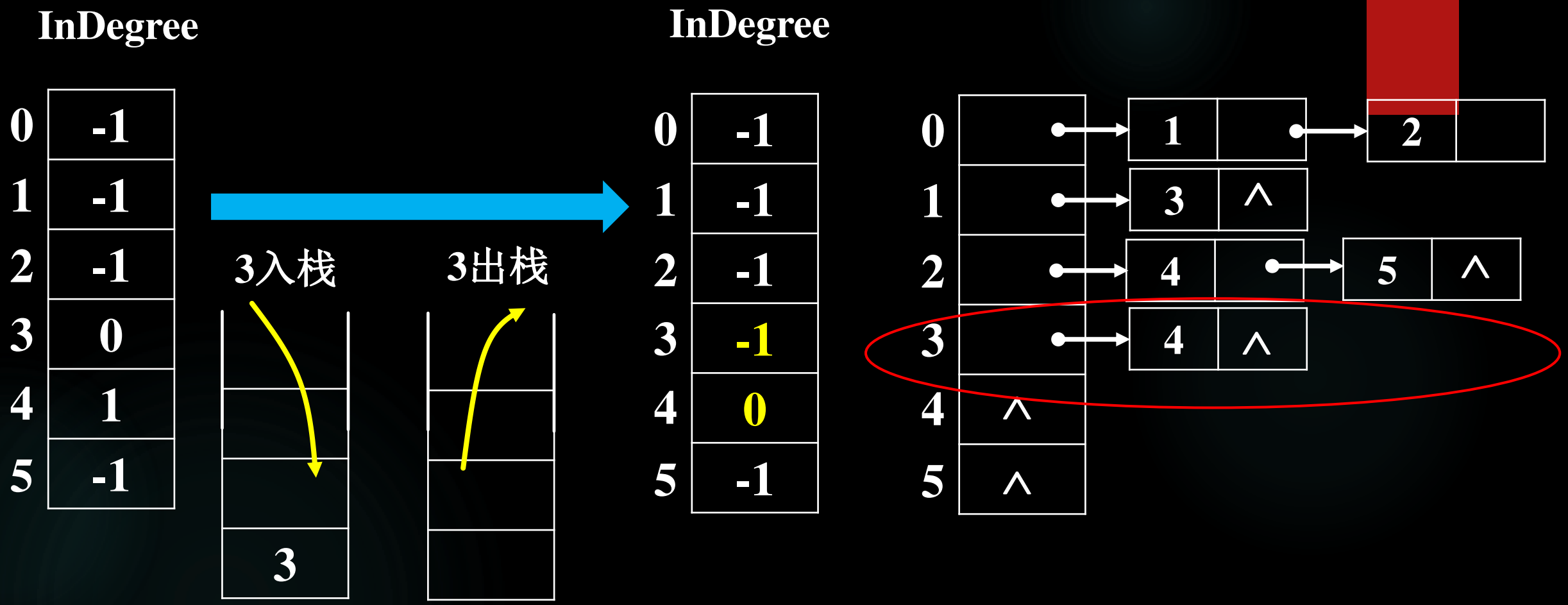
输出：0, 2


InDegree
0 -1
1 0
2 -1
3 1
4 1
5 0
5入栈
5出栈
5
1
1
InDegree
0 -1
1 0
2 -1
3 1
4 1
5 -1
0
1
2
3
4
5
1
2
3
4
5
4
^
^
^
^
^
^

输出：0, 2, 5


InDegree
0 -1
1 0
2 -1
3 1
4 1
5 -1
1出栈
InDegree
0 -1
1 -1
2 -1
3 0
4 1
5 -1
0 ^
1 1 2
2 3 ^
3 4 5 ^
4 4 ^
5 ^

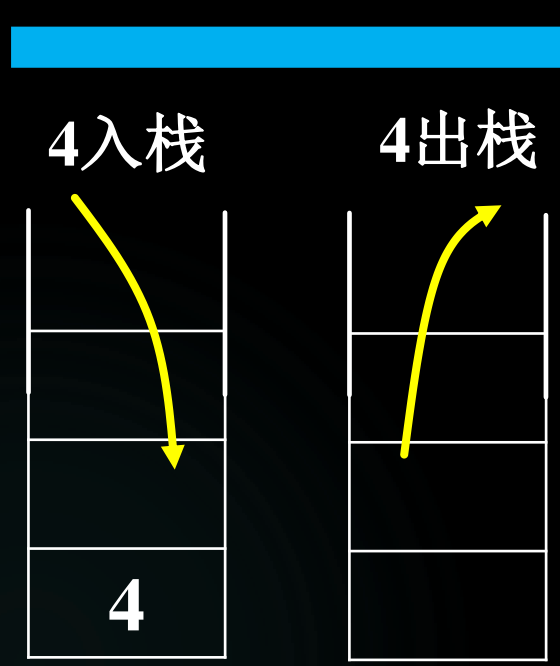
输出：0, 2, 5, 1


InDegree
0 -1
1 -1
2 -1
3 0
4 1
5 -1
3入栈
3出栈
3
InDegree
0 -1
1 -1
2 -1
3 -1
4 0
5 -1
0
1
2
3
4
5
1
2
3
4
5
4
^
^
^
^
^

输出：0, 2, 5, 1, 3

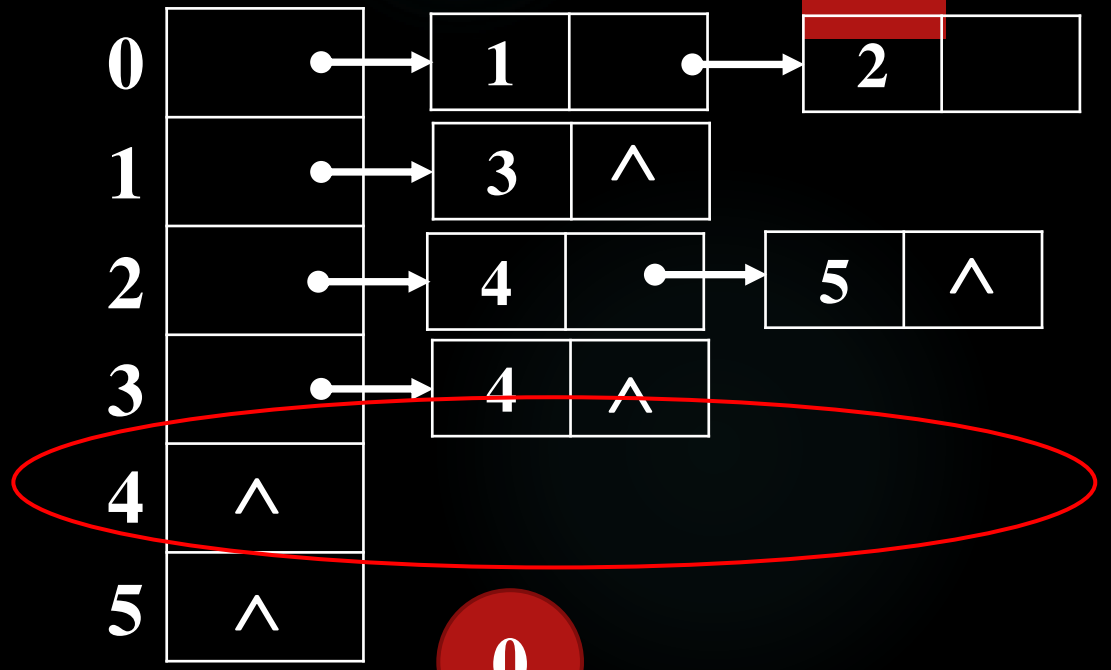
InDegree

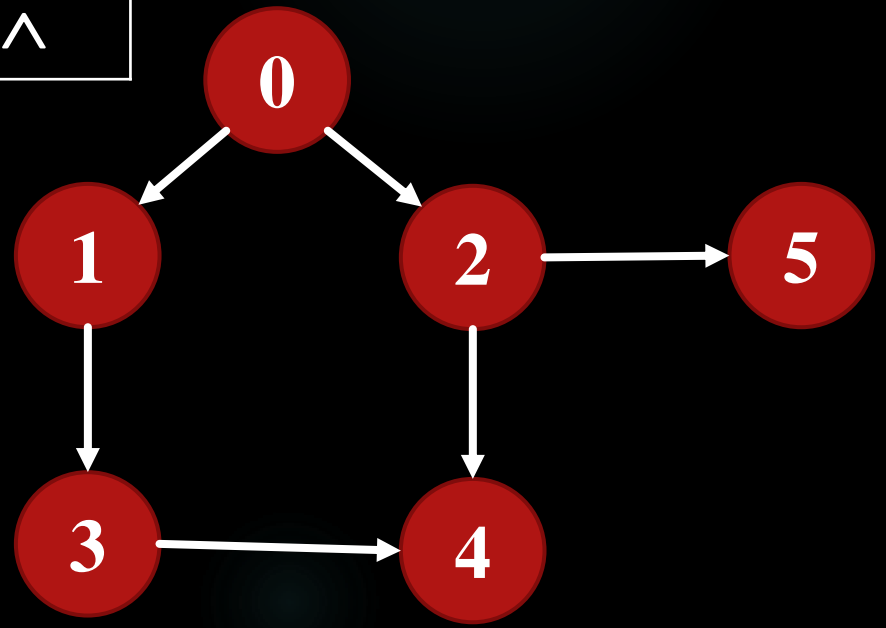
0	-1
1	-1
2	-1
3	-1
4	0
5	-1



InDegree

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1





输出：0, 2, 5, 1, 3, 4

图

目录

- 图的基本概念
- 图的存储结构
- 图的遍历
- 拓扑排序
- 关键路径
- 最小代价生成树：普里姆算法
- 单源最短路径和所有顶点间的最短路径

关键路径

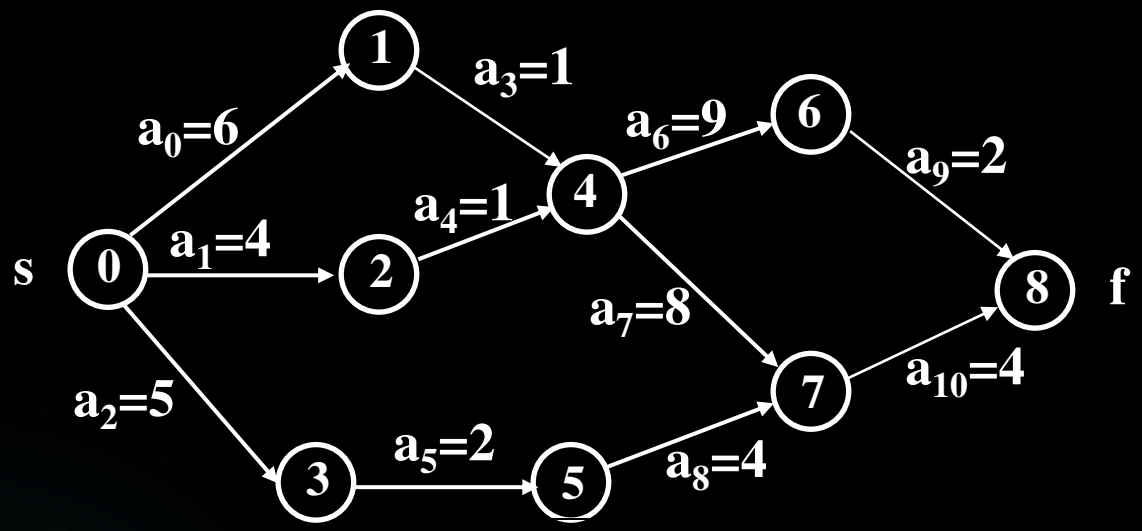
AOV网络，用顶点表示活动，有向边表示先决条件，有向图可以表示活动之间的领先关系。

AOE(边活动网络)：有向图G

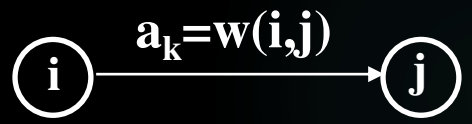
- 顶点代表事件，表示它的入边的活动均已完成，出边的活动可以开始
- 有向边表示活动，有向边上的权值表示一项活动持续的时间

AOE网络主要用于估算一项工程的完成时间。

AOE网络举例



AOE网的例子



边(i,j)的权值w(i,j)

v_0(s): 整个工程的起点(**源点**)， v_8(f): 整个工程的终点(**汇点**)。

v_4: 表示活动a_3和a_4已经完成，a_6和a_7可以开始。

a_5=2: 表示活动a_5持续的时间是2(天)。

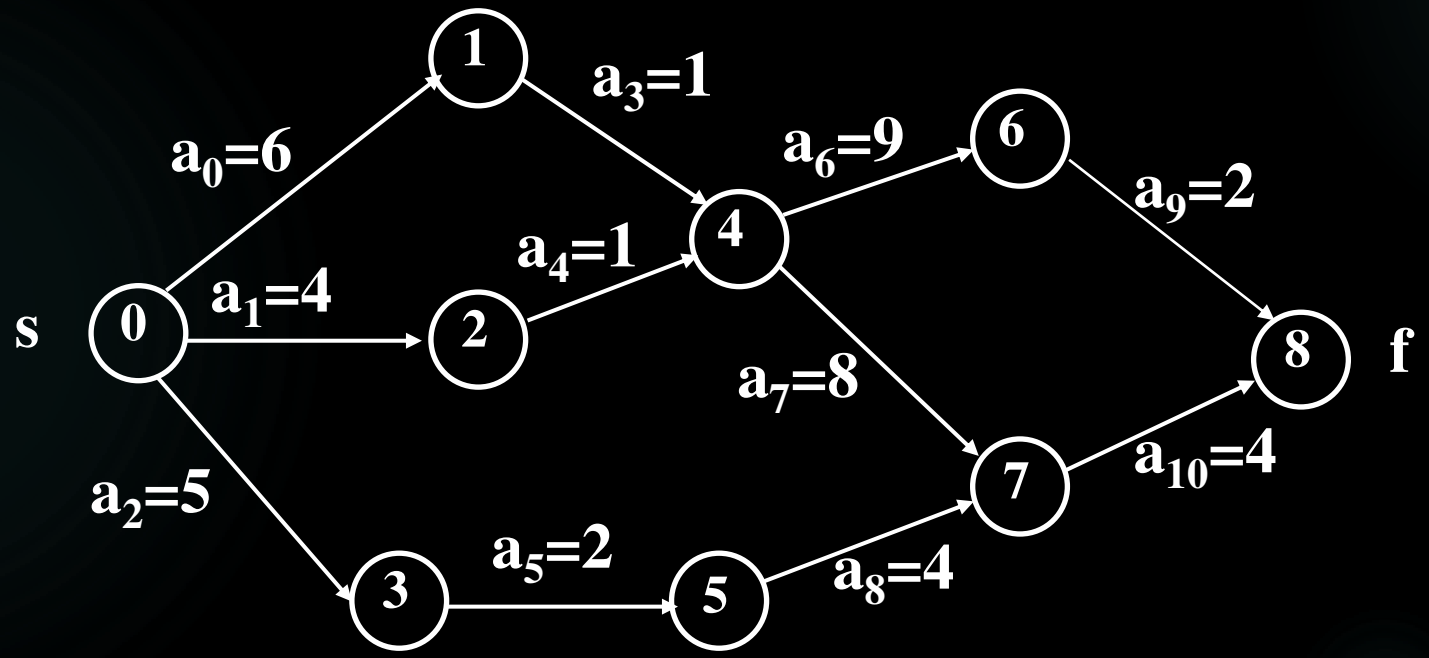
整个工程只有一个开始点，AOE就只有一个入度为0的顶点(源点)

整个工程只有一个完成点，AOE就只有一个出度为0的顶点(汇点)

关键路径与关键活动

(1)整个工程至少需要多少时间；

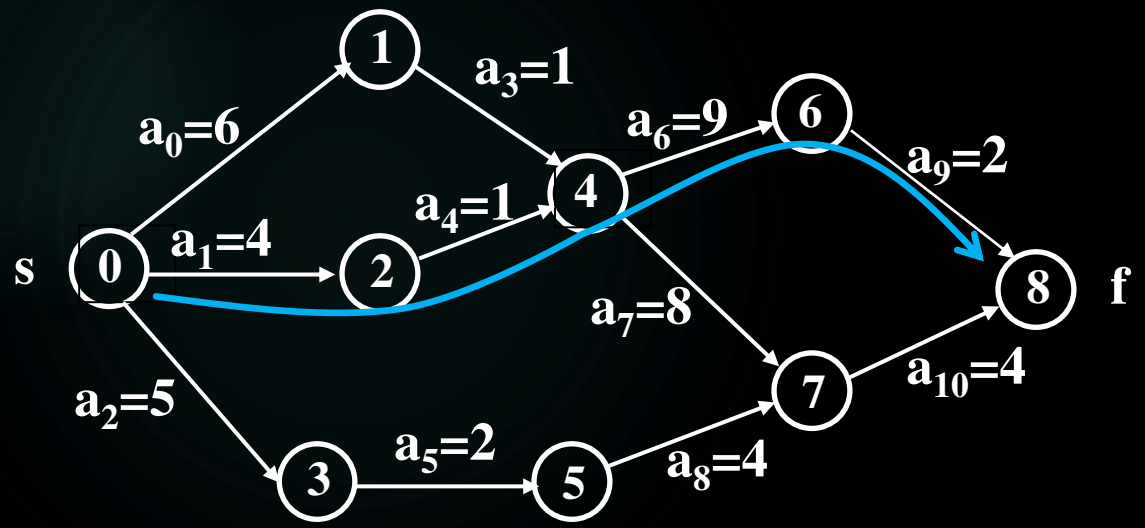
(2)哪些活动是影响工程进度的关键。



关键路径与关键活动

完成工程所需的最短时间是从开始顶点到完成顶点的最长路径的长度（各边的权值之和），这条最长路径称为**关键路径**。

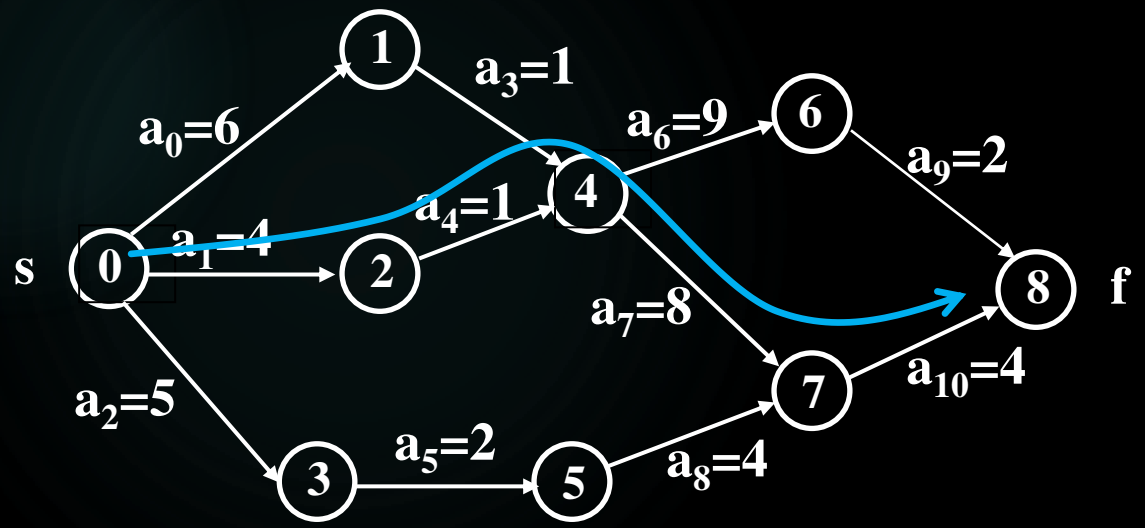
关键活动：关键路径上的活动，对整个工程的最短完成时间有影响，如果它不能按期完成就会影响整个工程。



关键路径与关键活动

完成工程所需的最短时间是从开始顶点到完成顶点的最长路径的长度（各边的权值之和），这条最长路径称为**关键路径**。

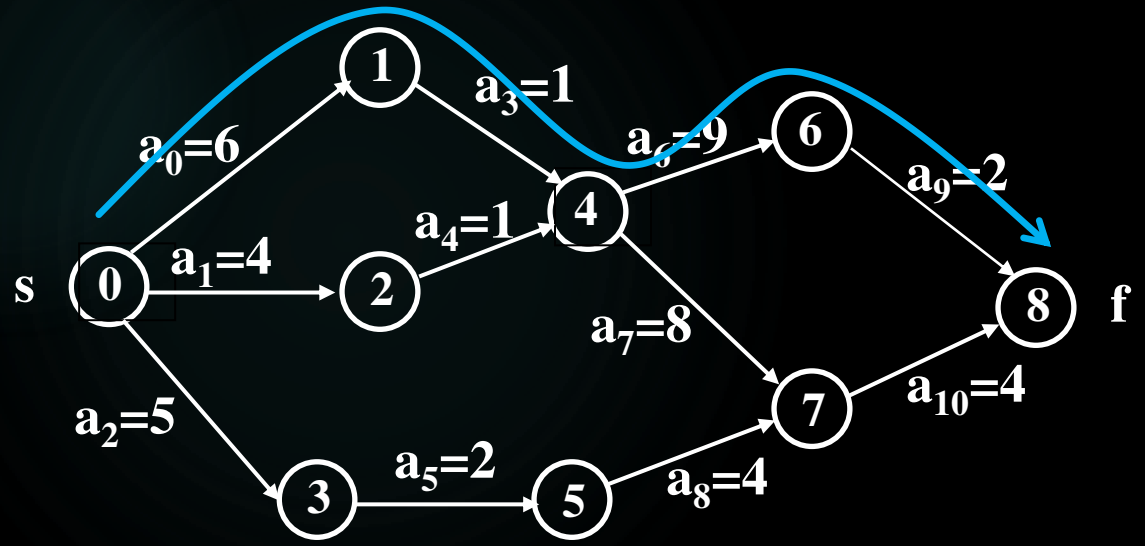
关键活动：关键路径上的活动，对整个工程的最短完成时间有影响，如果它不能按期完成就会影响整个工程。



关键路径与关键活动

完成工程所需的最短时间是从开始顶点到完成顶点的最长路径的长度（各边的权值之和），这条最长路径称为**关键路径**。

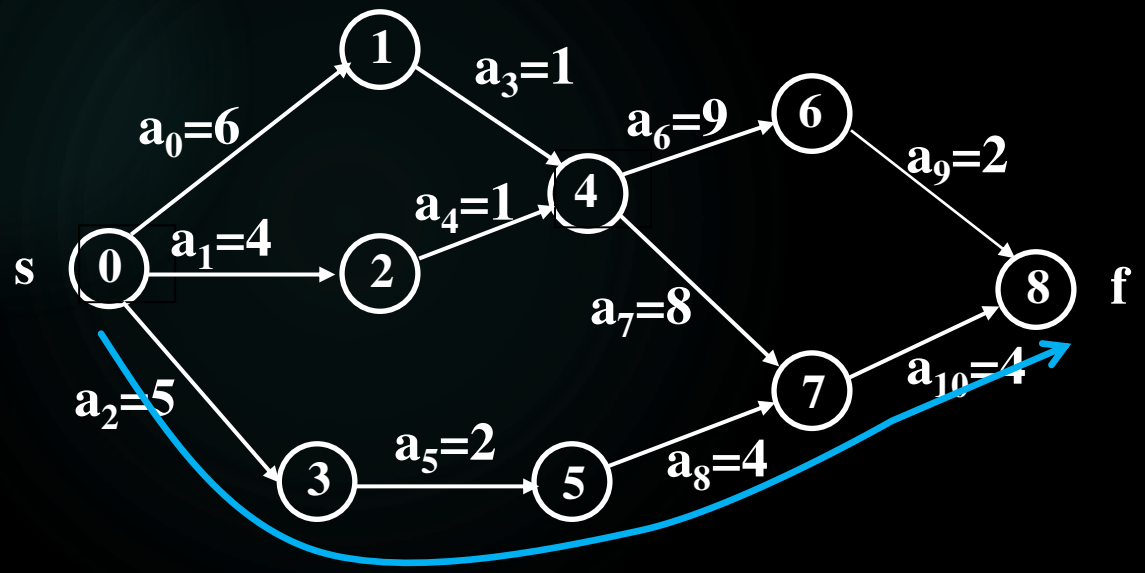
关键活动：关键路径上的活动，对整个工程的最短完成时间有影响，如果它不能按期完成就会影响整个工程。



关键路径与关键活动

完成工程所需的最短时间是从开始顶点到完成顶点的最长路径的长度（各边的权值之和），这条最长路径称为**关键路径**。

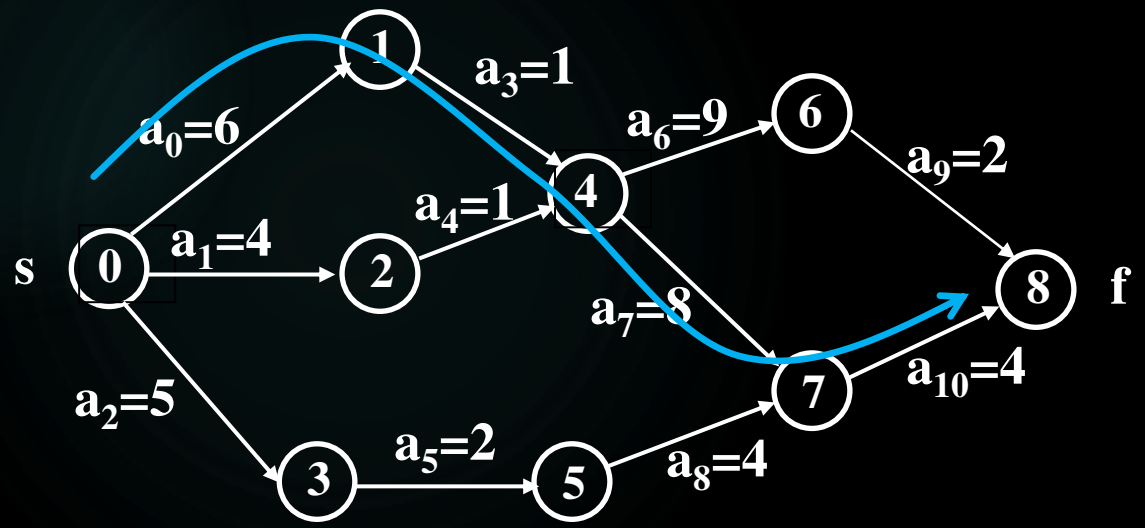
关键活动：关键路径上的活动，对整个工程的最短完成时间有影响，如果它不能按期完成就会影响整个工程。



关键路径与关键活动

完成工程所需的最短时间是从开始顶点到完成顶点的最长路径的长度（各边的权值之和），这条最长路径称为**关键路径**。

关键活动：关键路径上的活动，对整个工程的最短完成时间有影响，如果它不能按期完成就会影响整个工程。



(v_0,v_1,v_4,v_7,v_8)就是一条关键路径。

a_0,a_3,a_7,a_{10}就是关键活动。

路径长度=

$a_0+a_3+a_7+a_{10}=6+1+8+4=19$

找关键活动的目的：

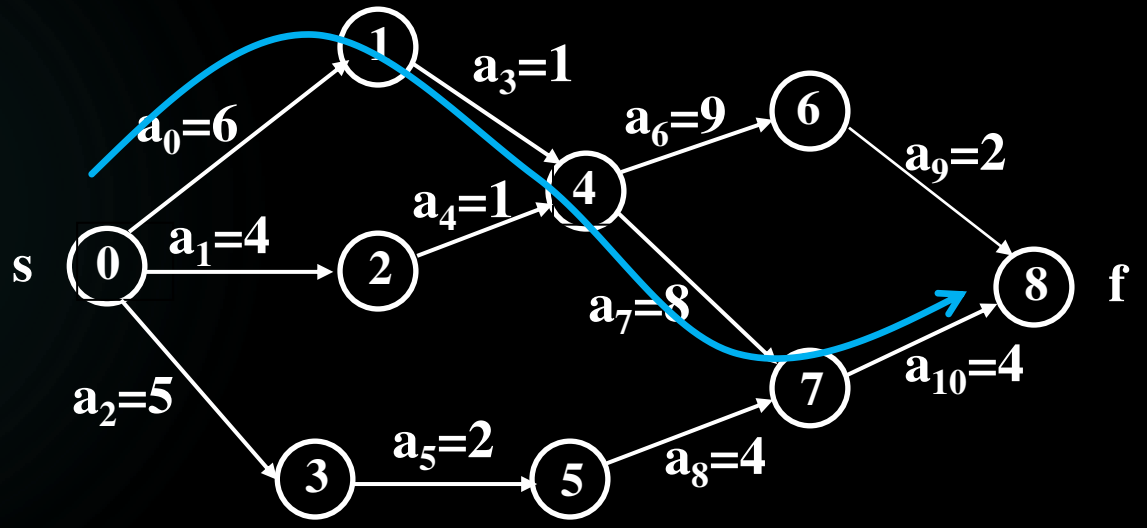
(1)重视关键活动；

(2)缩短整个工期。

关键活动对缩短整个工期起着至关重要的作用；

非关键活动对缩短整个工期不起作用。

因此，对关键活动投入较多的人力和物力，确保工程按期完成，或缩短整个工期。



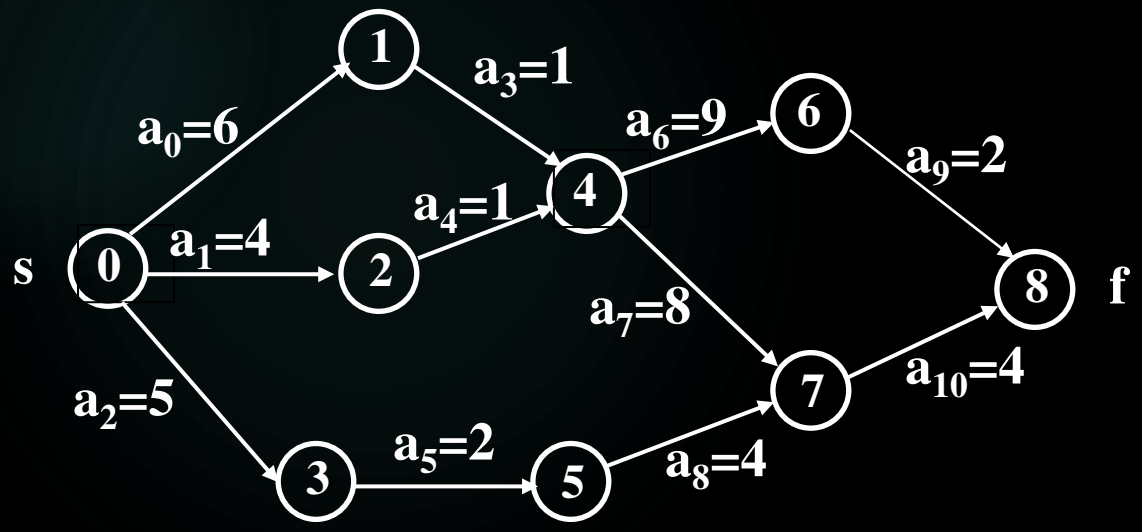
为了定量找出关键路径和关键活动，需要进行以下计算：

(1)事件v_i的可能的最早发生时间

earliest(i)=从源点v_0到v_i的最长路径长度

(2)事件v_i的允许的最晚发生时间。即在不影响工期的条件下事件v_i允许的最晚发生时间。

latest(i)=earliest(n)-从v_i到汇点v_n的最长路径长度



earliest(4)=a_0+a_3=6+1=7

earliest(5)=a_2+a_5=5+2=7

earliest(8)=a_0+a_3+a_7+a_{10}=6+1+8+4=19

latest(4)=19-(8+4)=7

latest(5)=19-(4+4)=11

latest(7)=earliest(8)-4=19-4=15

(3) early(k)：活动a_k的可能的最早开始时间

a_k表示的边为<v_i,v_j>， w(i,j)为a_k的权值。

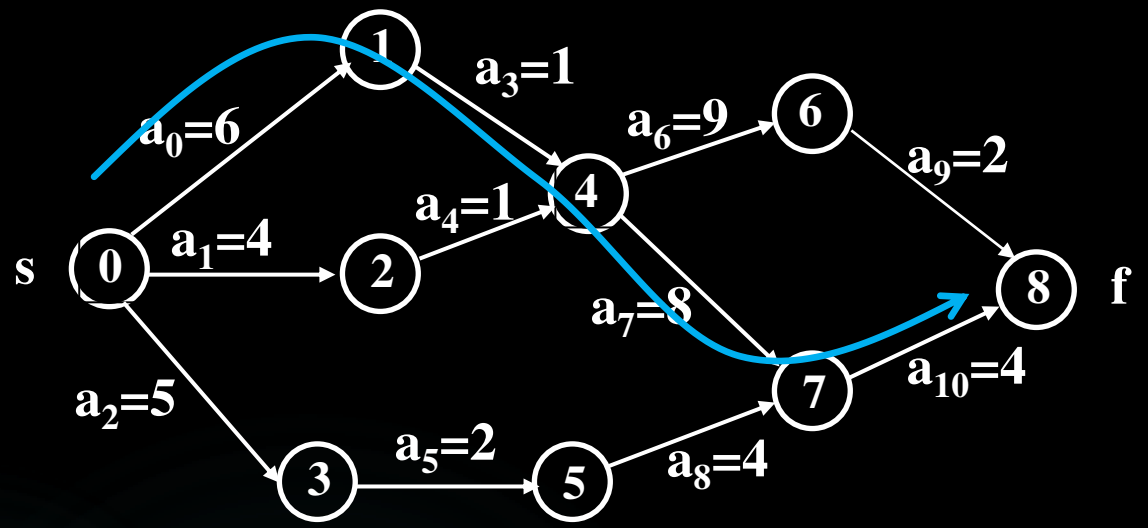
early(k)=事件v_i的可能的最早发生时间

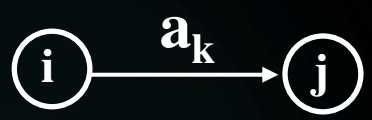
即 early(k)=earliest(i)

(4) late(k):活动a_k的允许的最晚开始时间。

late(k)=latest(j)-w(i,j)

(5)若early(k)=late(k),则a_k是关键活动。





early(k)=earliest(i)

late(k)=latest(j)-w(i,j)

early(7)=earliest(4)=7

late(7)=latest(7)-8=15-8=7

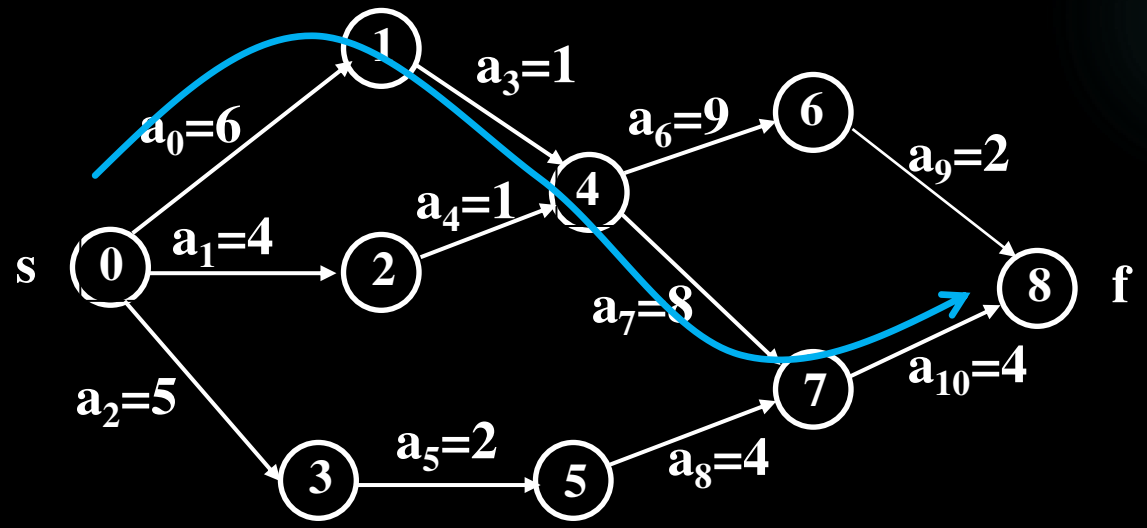
early(8)=earliest(5)=7

late(8)=latest(7)-4=11

因early(7)=late(7),

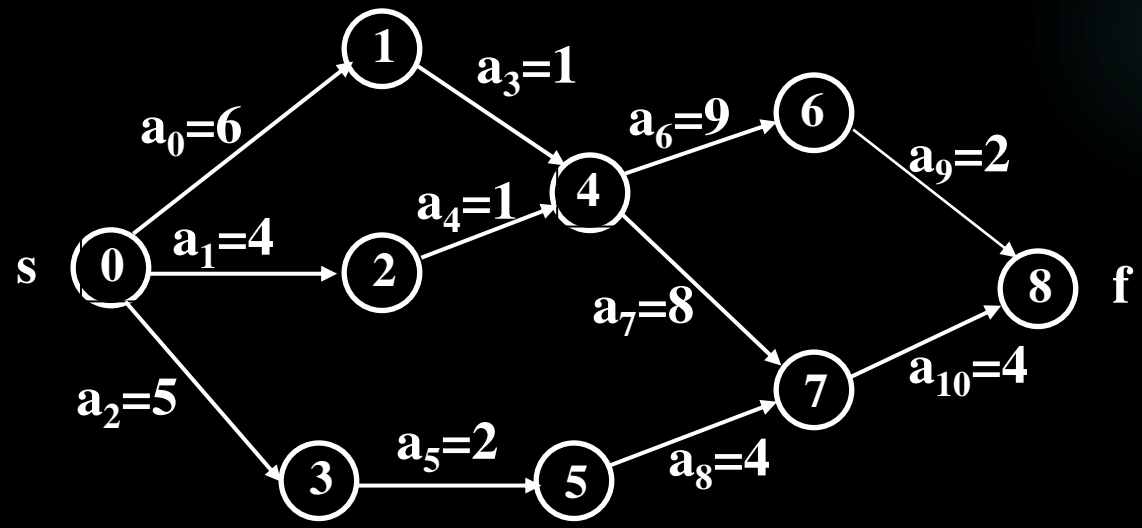
early(8)!=late(8),

故a_7是关键活动，a_8不是关键活动。



事件i:
earliest(i)=从源点v0到vi的最长路径长度
latest(i)=earliest(n)-从vi到汇点vn的最长路径长度

活动a_k 在<i,j>上:
early(k)=earliest(i)
late(k)=latest(j)-w(i,j)
若 early(k)=late(k)，则a_k是关键活动。



项目	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
earliest(i)	0	6	4	5	7	7	16	15	19
latest(i)	0	6	6	9	7	11	17	15	19

项目	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
early(k)	0	0	0	6	4	5	7	7	7	16	15
late(k)	0	2	4	6	6	9	8	7	11	17	15
关键*活动*	*			*				*			*

关键路径： v_0 ,v_1 v_4 ,v_7, v_8

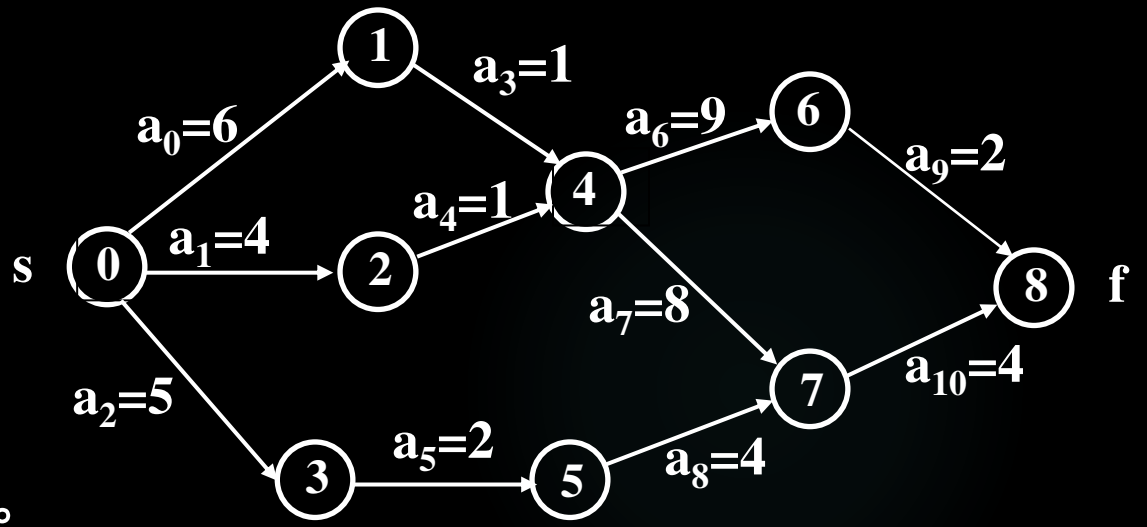
关键路径算法

(1)计算事件的最早发生时间earliest。

从earliest(0)=0开始，自前向后递推计算:

earliest(j)=max{earliest(i)+w(i,j)} i∈P(j)

P(j)是所有以j为头的弧<i,j>的弧尾i的集合。



为保证计算earliest(j)时，所有的earliest(i)(i∈P(j))已经求得。可以利用拓扑排序。

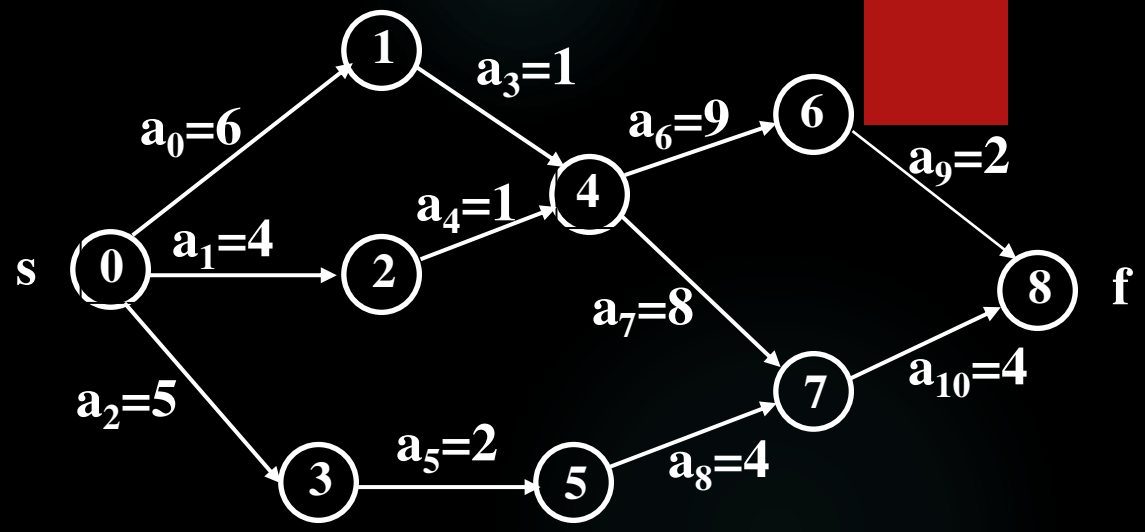
earliest(0) = 0
earliest(1) = max{earliest(0)+w(0,1)}
……
earliest(4) = max{earliest(1)+w(0,1), earliest(2)+w(2,1)}
……
earliest(8)=max{earliest(6)+w(6,8), earliest(7)+w(7,8)}

(2)计算事件的最晚发生时间latest。

从latest(n)=earliest(n)开始，自后向前递推计算:

latest(i)=min{latest(j)-w(i,j)} j∈S(j)

S(i)是所有<i,j>的弧头j的集合。



为保证计算latest(i)时，所有的latest(j)(j∈S(i))已经求得。可以利用逆拓扑排序。

latest(8) = earliest(8) = 19
latest(6) = min{latest(8)-w(6,8)}
latest(7) = min{latest(8)-w(7,8)}
……
latest(4) = min{latest(6)-w(4,6), latest(7)-w(4,7)}
……
latest(0) = 0

(3)计算活动的最早开始时间early(k)和最晚开始时间late(k)。

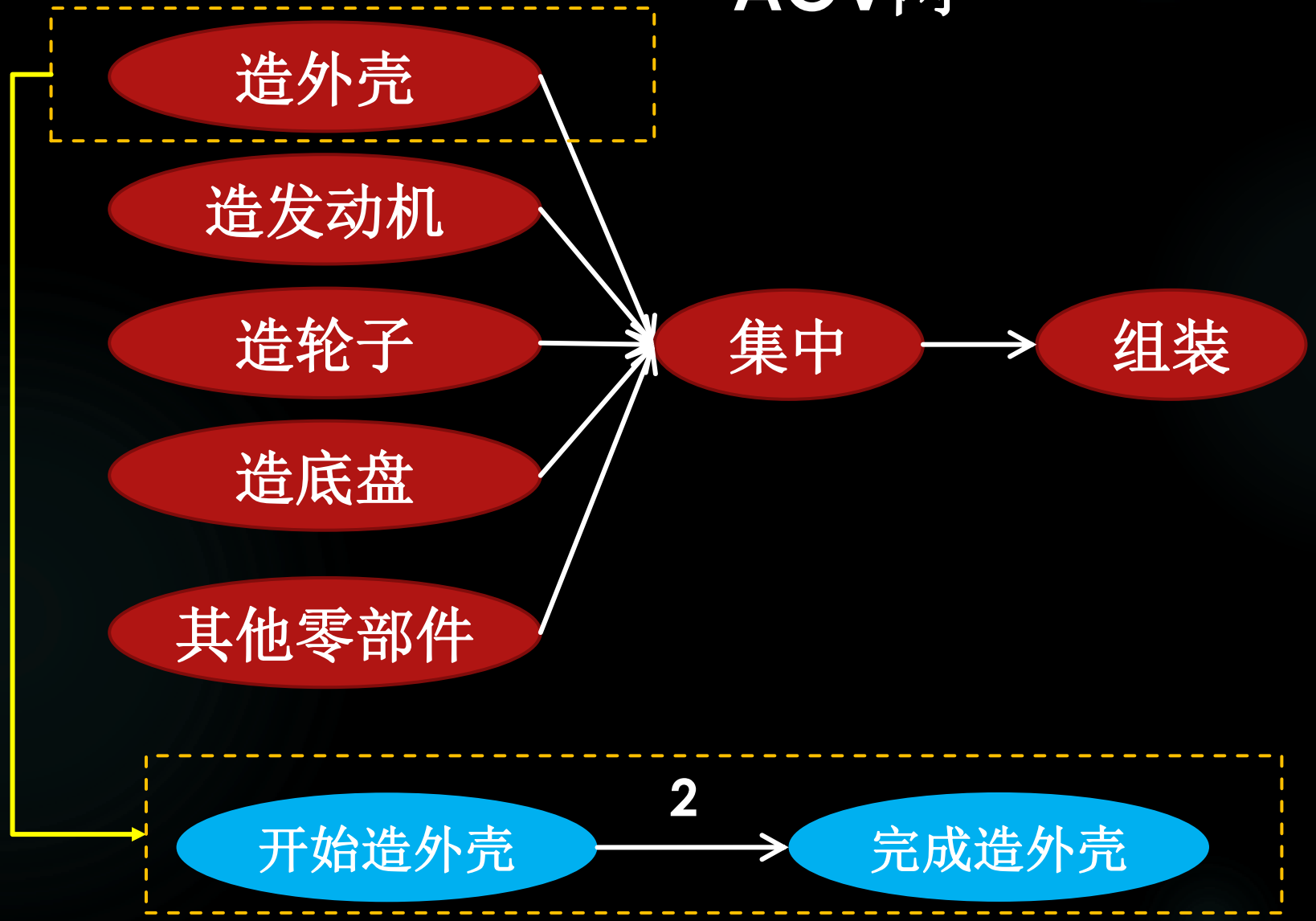
early(k)=earliest(i)

late(k)=latest(j)-w(i,j)

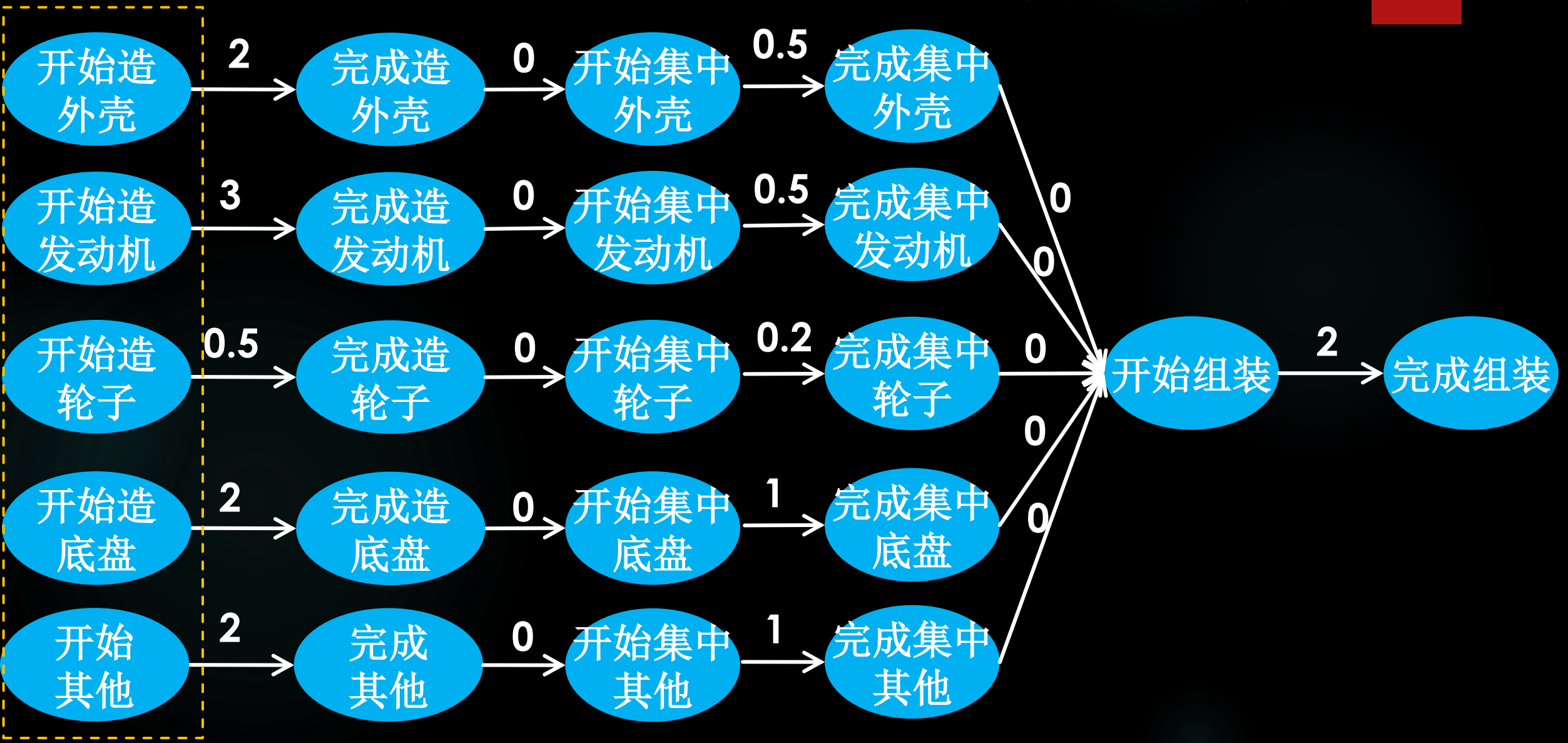
(4)确定关键路径

若early(k)=late(k),则a_k是关键活动。

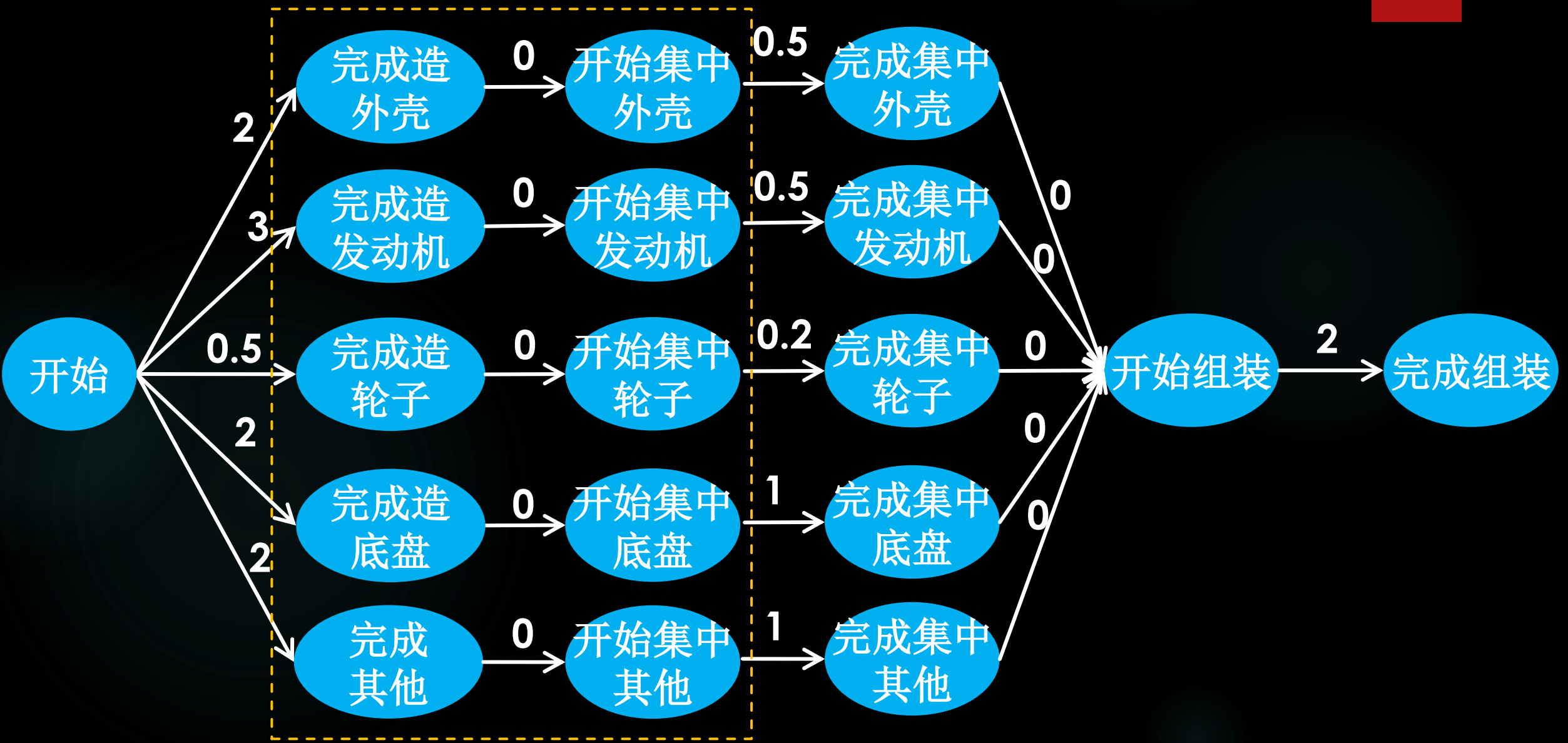
AOV网



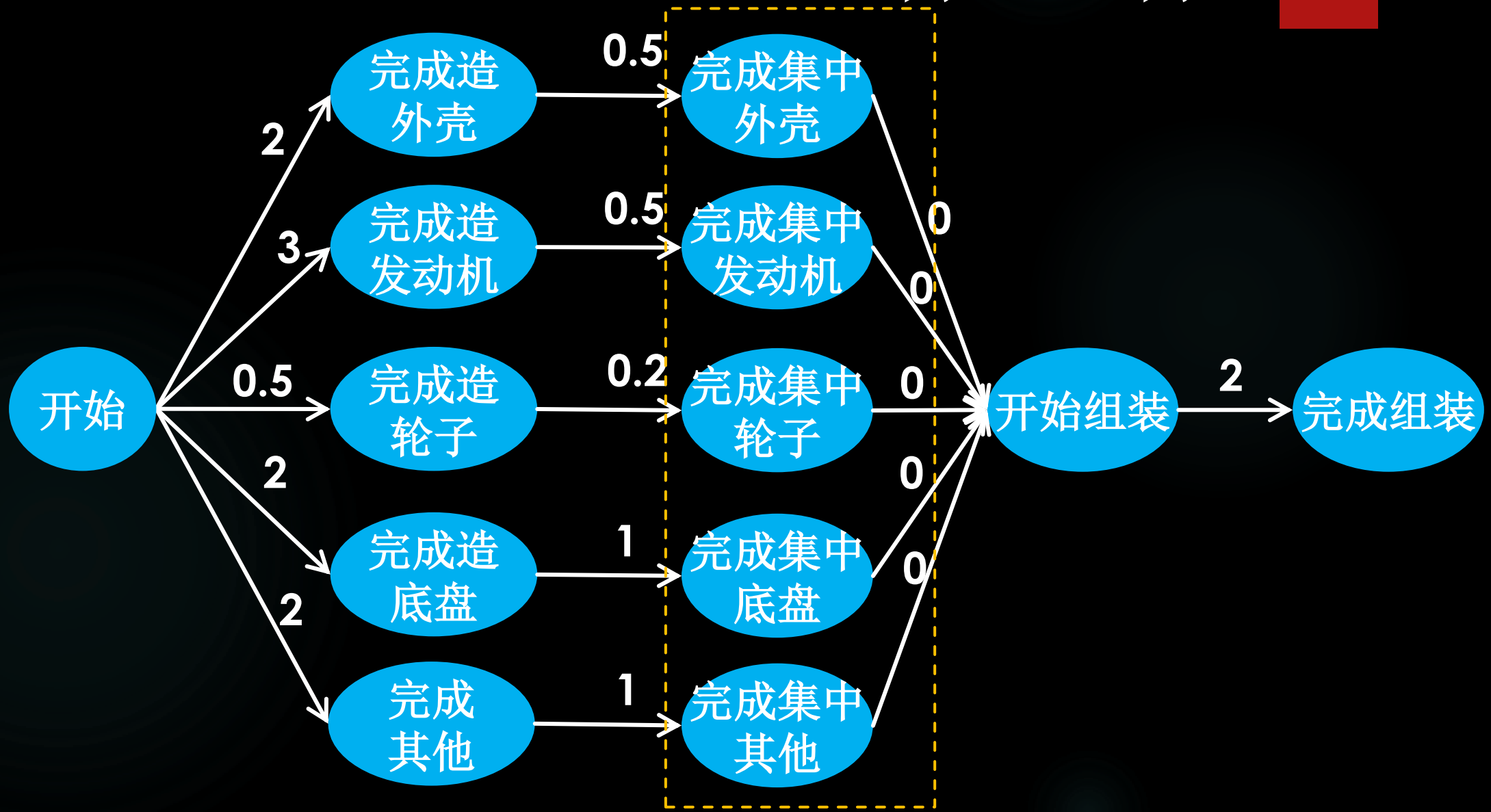
AOV网→AOE网


开始造外壳
2
完成造外壳
0
开始集中外壳
0.5
完成集中外壳
开始造发动机
3
完成造发动机
0
开始集中发动机
0.5
完成集中发动机
开始造轮子
0.5
完成造轮子
0
开始集中轮子
0.2
完成集中轮子
开始造底盘
2
完成造底盘
0
开始集中底盘
1
完成集中底盘
开始其他
2
完成其他
0
开始集中其他
1
完成集中其他
0
0
0
0
0
开始组装
2
完成组装

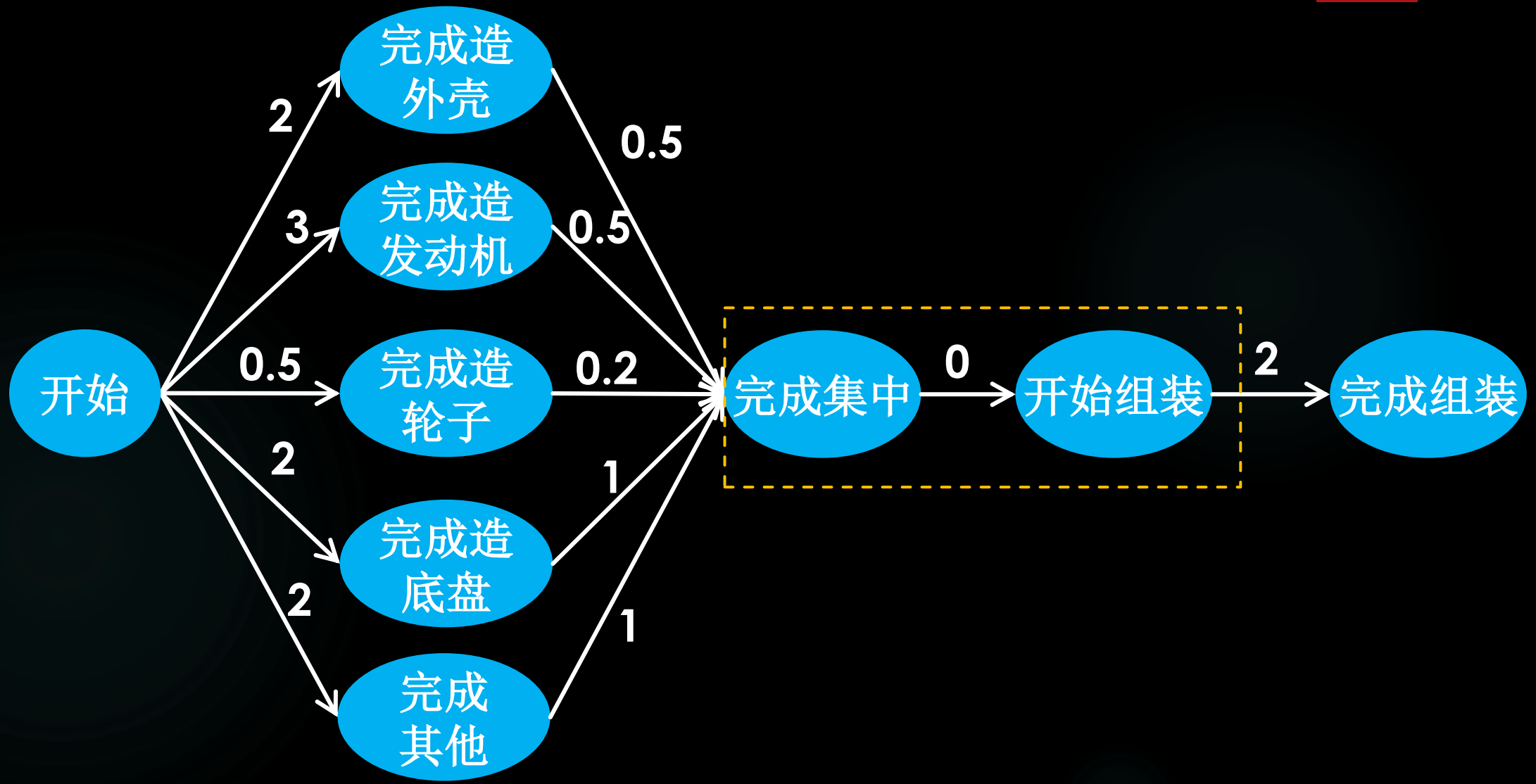
AOV网→AOE网


开始
2
3
0.5
2
2
完成造外壳
完成造发动机
完成造轮子
完成造底盘
完成其他
0
0
0
0
0
开始集中外壳
开始集中发动机
开始集中轮子
开始集中底盘
开始集中其他
0.5
0.5
0.2
1
1
完成集中外壳
完成集中发动机
完成集中轮子
完成集中底盘
完成集中其他
0
0
0
0
0
开始组装
2
完成组装

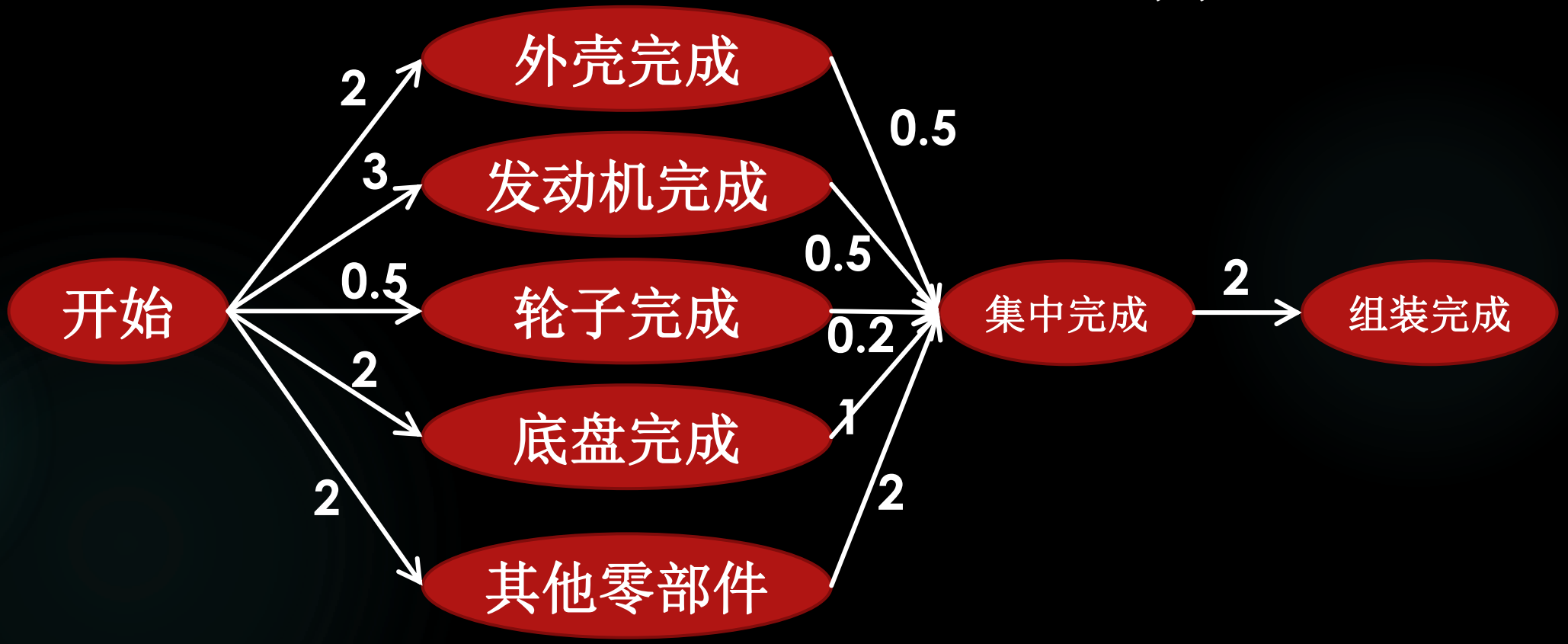
AOV网→AOE网


开始
完成造外壳
完成造发动机
完成造轮子
完成造底盘
完成其他
完成集中外壳
完成集中发动机
完成集中轮子
完成集中底盘
完成集中其他
开始组装
完成组装
2
3
0.5
2
2
0.5
0.5
0.2
1
1
0
0
0
0
2

AOV网→AOE网


开始
完成造外壳
完成造发动机
完成造轮子
完成造底盘
完成其他
完成集中
开始组装
完成组装
2
3
0.5
2
2
0.5
0.5
0.2
1
1
0
2

AOE网



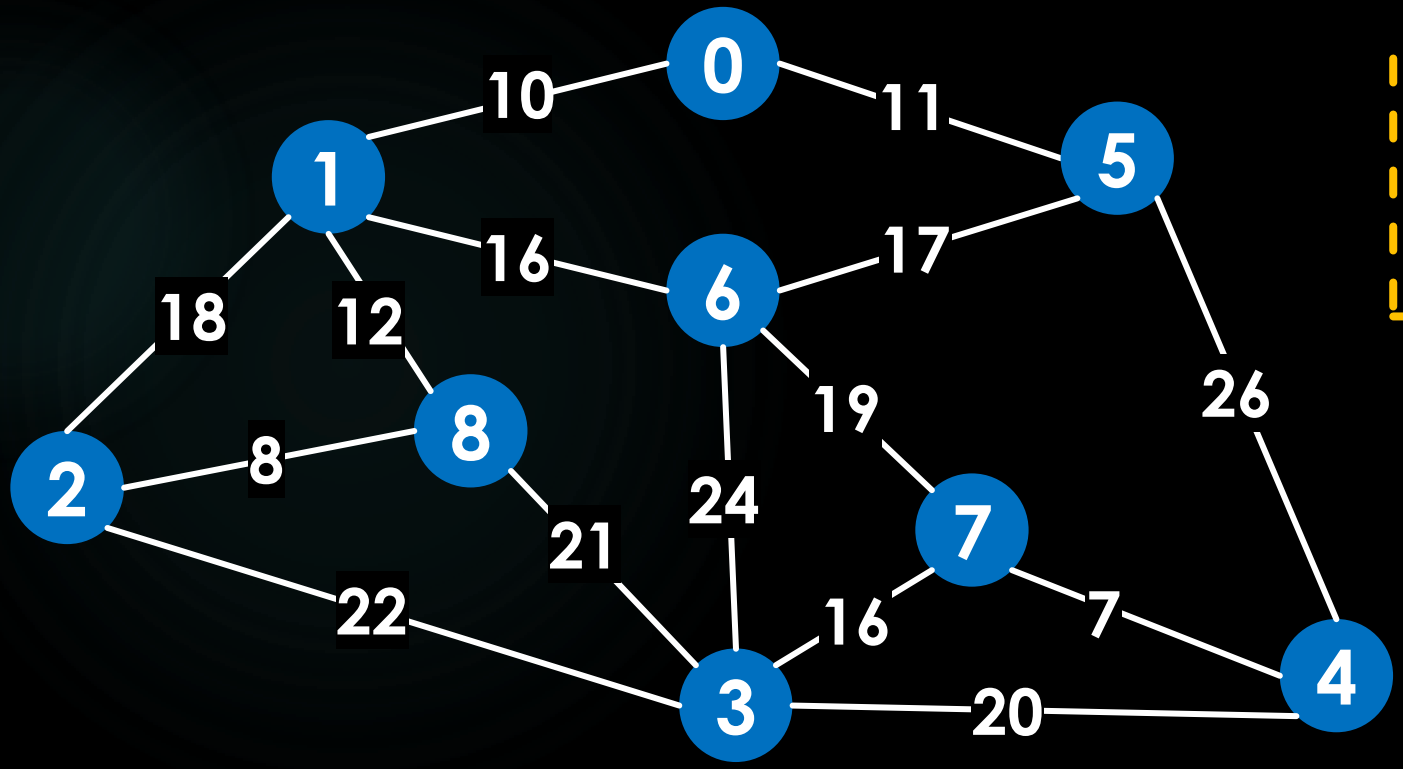
图

目录

- 图的基本概念
- 图的存储结构
- 图的遍历
- 拓扑排序
- 关键路径
- 最小代价生成树
- 单源最短路径和所有顶点间的最短路径

最小代价生成树的问题由来

电信实施工程师要在9个村庄之间架设通信线路。已知每两个村庄间架设线路的代价，问如何选择架设线路，使得9个村庄之间可以通信，切代价最小?



解决方案应该具有以下属性

- 选择架设线路应该使得两两连通图的生成树（连通图）
- 架设线路数量最少(8)
- 架设线路代价最小-8条边上权值之和

最小代价生成树

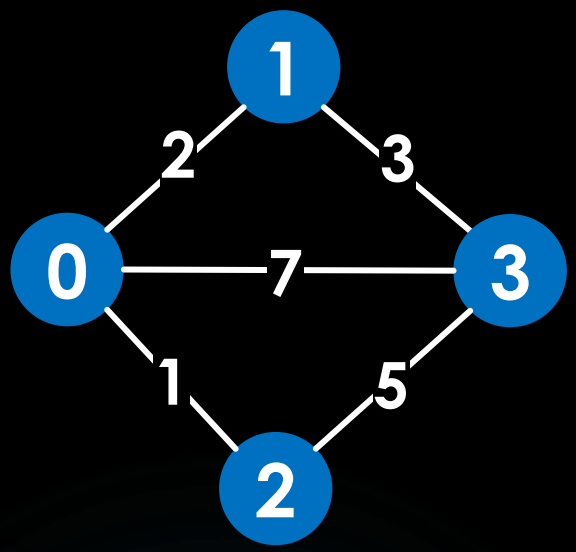
最小代价生成树的概念

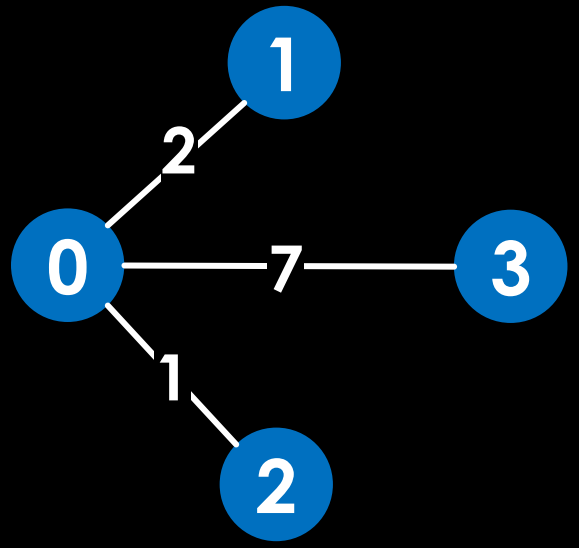
一个连通图的生成树是一个**极小连通子图**，它包括图中全部顶点，但只有足以构成一棵树的**n-1**条边

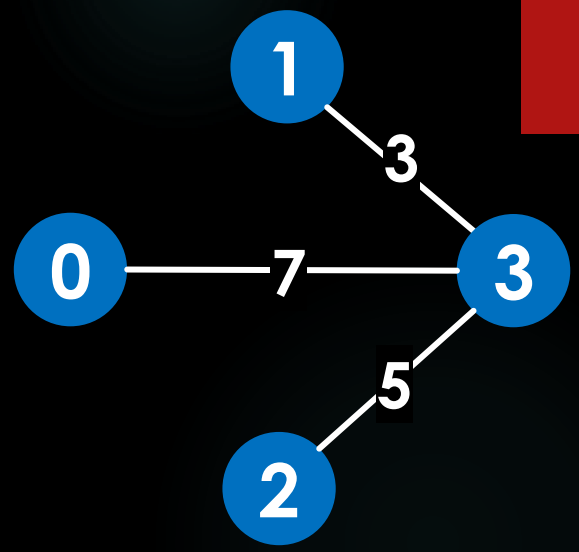
遍历（DFS/BFS）一个连通图可以得到它的生成树

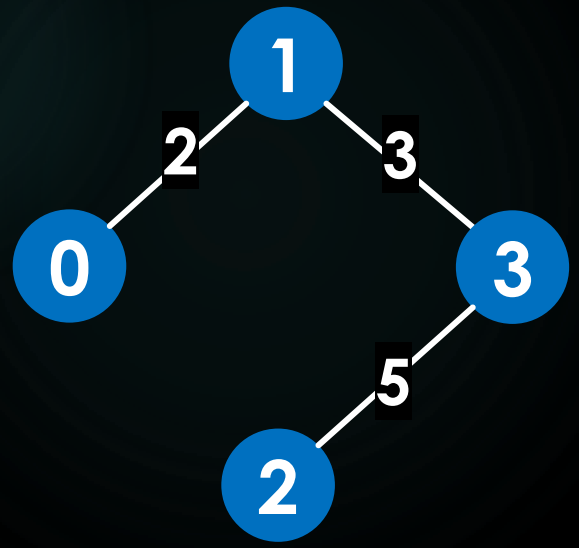
一棵生成树的代价是各条边上的代价之和

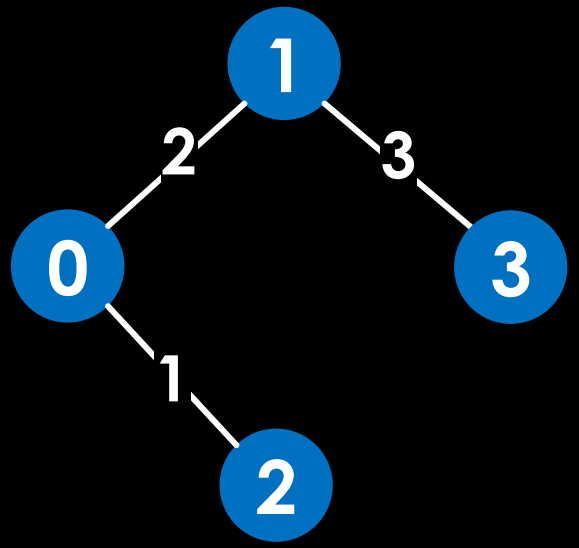
一个网络（**带权的连通图**）的生成树中具有最小代价的生成树称为该网络的最小代价生成树

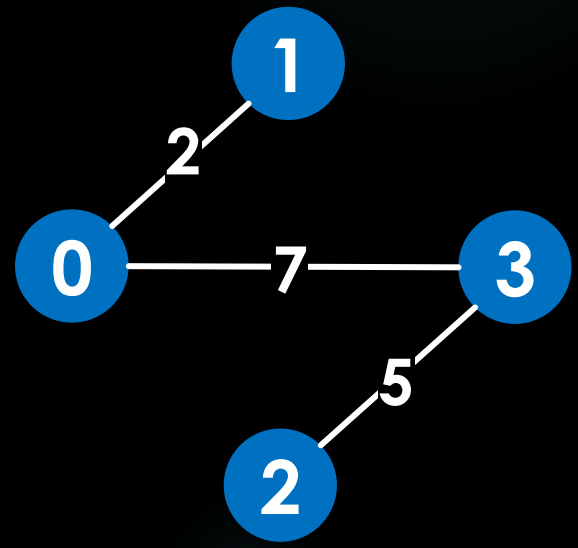

1
2
3
0
7
3
1
5
2


1
2
0
7
3
1
2


1
3
0
7
3
5
2


1
2
3
0
3
5
2

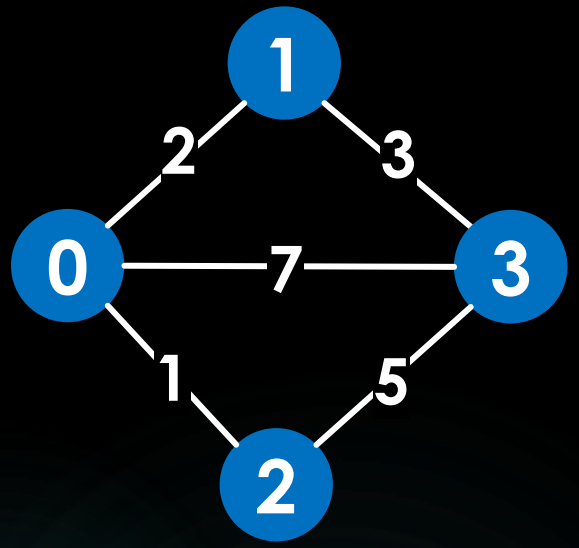

1
2
3
0
3
1
2


1
2
0
7
3
5
2

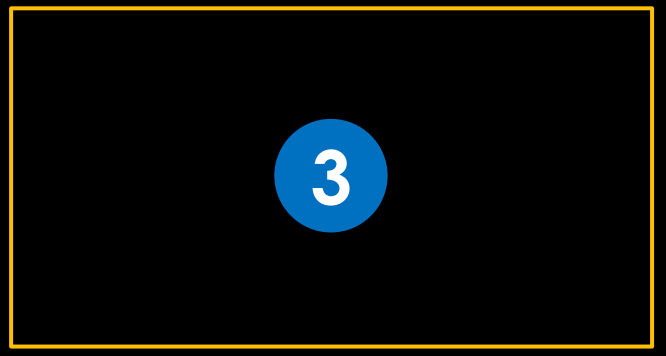
构造最小代价生成树的算法

- **普里姆算法 (Prim)**
- **克鲁斯卡尔算法 (Kruskal)**

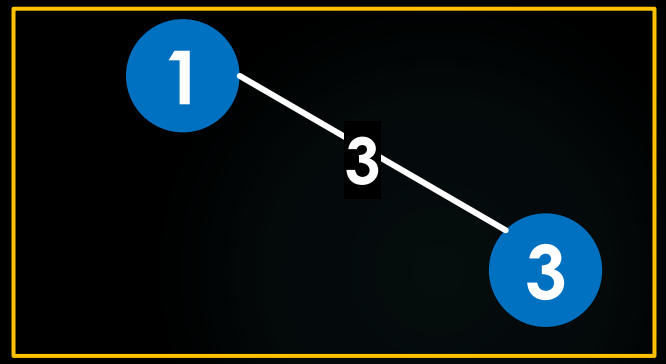
普里姆算法



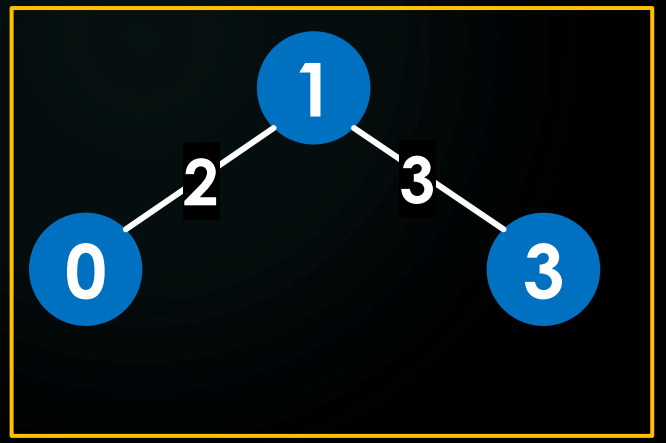
任选一个顶点作为起点



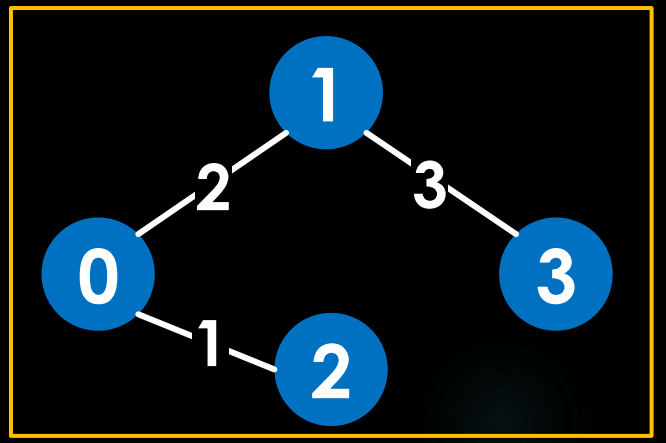
选择一条代价最短的边(3,x)



选择一条代价最短的边(3,x)或(1,x)
新添加的边不能造成回路

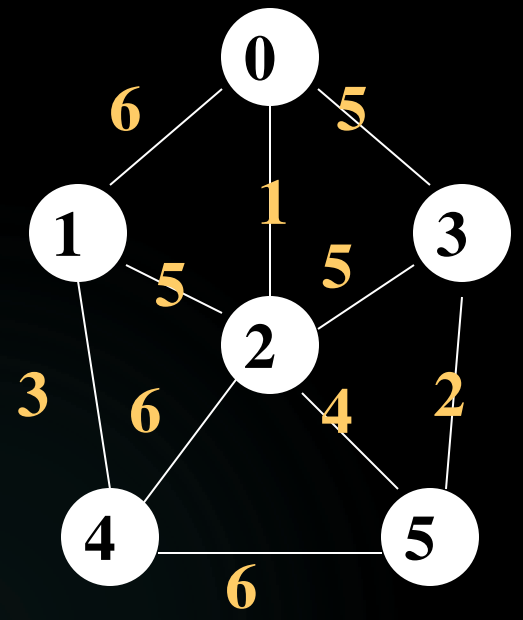


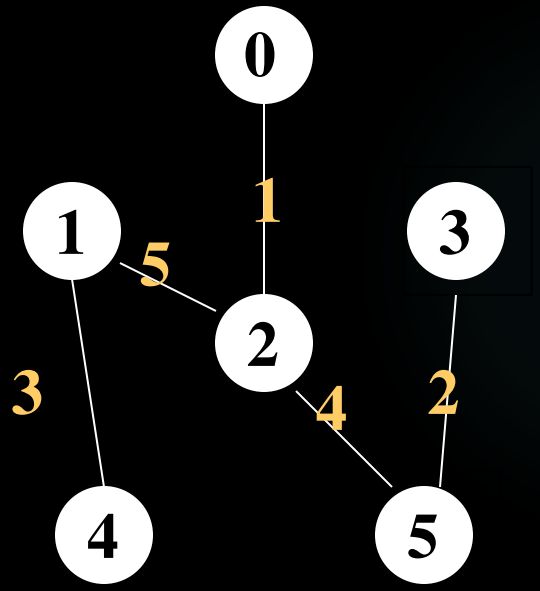
选择一条代价最短的边(3,x)、(1,x)
或(0,x)，新添加的边不能造成回路



普里姆算法

- G=(V,E)是带权的连通图，T=(V’,E’)是正在构造中的生成树
- 初始状态： $T=(\{v_0\},\{\ \})$，v_0是任意选定的起始顶点
- 重复执行下列运算：
  - 在所有u∈V’，v∈V-V’的边(u,v)中找一条代价最小的边(u’,v’)，
  - 边(u’,v’)并入集合E’，将顶点v’并入集合V’
  - 重复执行上述操作，直到V=V’为止。
  - 这时E’中必有n-1条边，T=(V’,E’)是图G的一棵最小代价生成树。

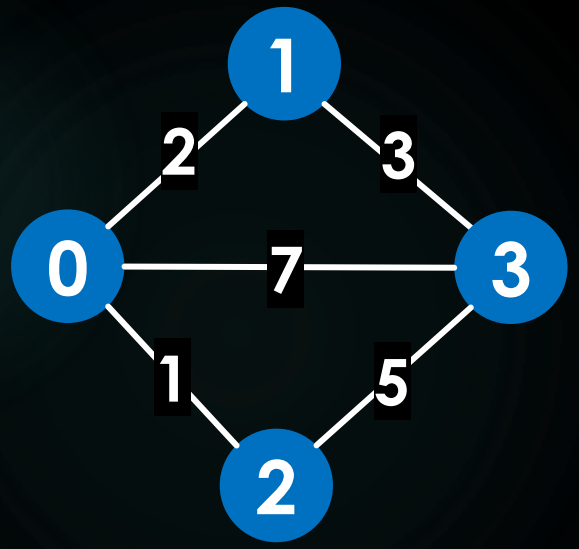




以0为起始结点，用普里姆算法构造最小代价生成树的过程

普里姆算法实现

- 本算法的C实现中图采用**邻接表**存储
- 一维数组nearest，nearest[i]存放与i**距离最近且在生成树**上的顶点
- 一维数组lowcost，lowcost[i]存放边(i,nearest[i])的权值
- 一维数组mark，标记顶点是否已经在生成树上

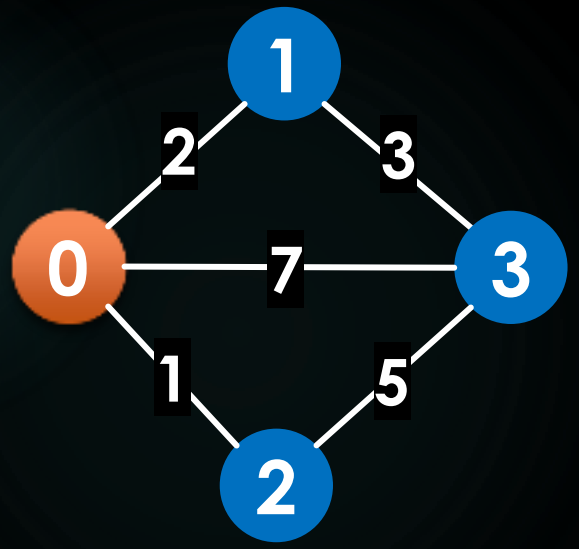


T=({ },{ })

	0	1	2	3
nearest	-1	-1	-1	-1
lowcost	INFTY	INFTY	INFTY	INFTY
mark	false	false	false	false

普里姆算法实现

- 本算法的C实现中图采用**邻接表**存储
- 一维数组nearest，nearest[i]存放与i**邻接最近且在生成树**上的顶点
- 一维数组lowcost，lowcost[i]存放边(i,nearest[i])的权值
- 一维数组mark，标记顶点是否已经在生成树上

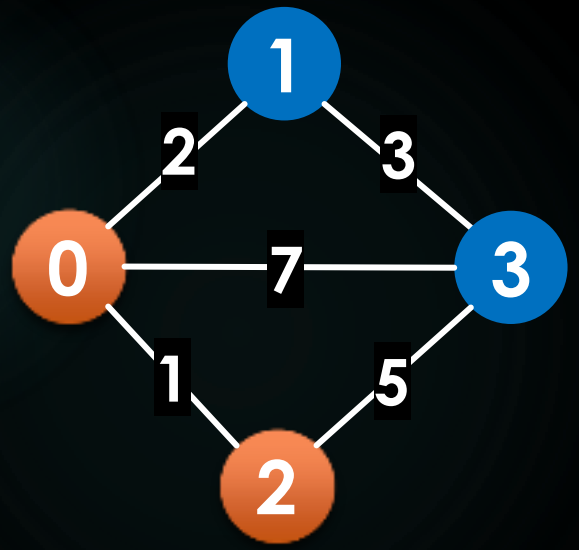


T=({0},{ })

	0	1	2	3
nearest		0	0	0
lowcost		2	1	7
mark	true	false	false	false

普里姆算法实现

- 本算法的C实现中图采用**邻接表**存储
- 一维数组nearest，nearest[i]存放与i**邻接最近且在生成树**上的顶点
- 一维数组lowcost，lowcost[i]存放边(i,nearest[i])的权值
- 一维数组mark，标记顶点是否已经在生成树上

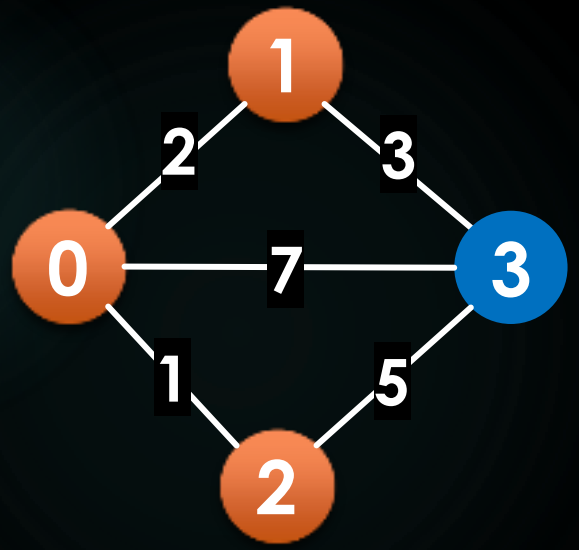


T=({0,2},{(0,2)})

	0	1	2	3
nearest		0		2
lowcost		2		5
mark	true	false	true	false

普里姆算法实现

- 本算法的C实现中图采用**邻接表**存储
- 一维数组nearest，nearest[i]存放与i**邻接最近且在生成树**上的顶点
- 一维数组lowcost，lowcost[i]存放边(i,nearest[i])的权值
- 一维数组mark，标记顶点是否已经在生成树上

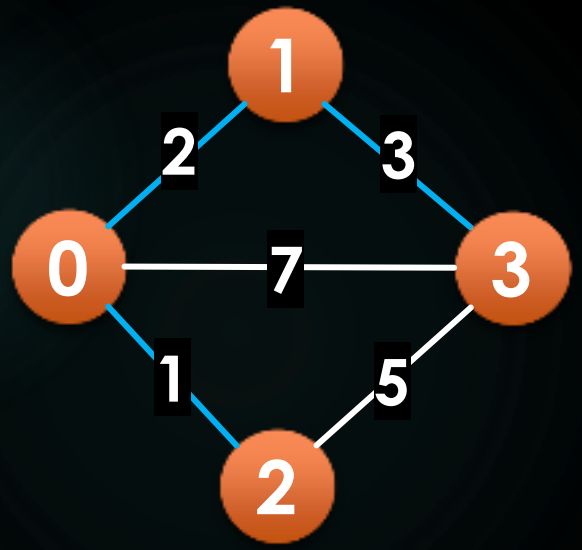


T=({0,2,1},
{(0,2),(1,0)})

	0	1	2	3
nearest				1
lowcost				3
mark	true	true	true	false

普里姆算法实现

- 本算法的C实现中图采用**邻接表**存储
- 一维数组nearest，nearest[i]存放与i**邻接最近且在生成树**上的顶点
- 一维数组lowcost，lowcost[i]存放边(i,nearest[i])的权值
- 一维数组mark，标记顶点是否已经在生成树上

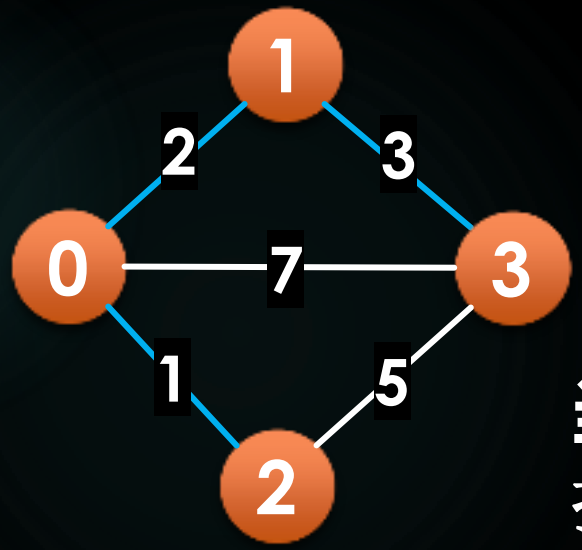


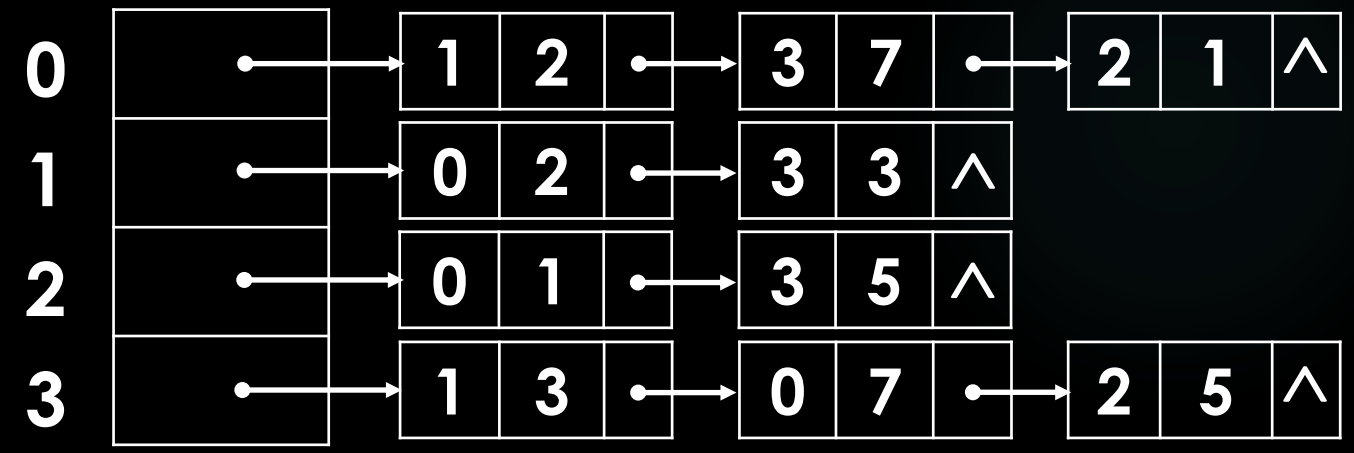
T=({0,2,1,3},
{(0,2),(1,0),(3,1)})

	0	1	2	3
nearest				
lowcost				
mark	true	true	true	true

普里姆算法实现

- 一维数组nearest，nearest[i]存放与i**邻接最近且在生成树**上的顶点
- 一维数组lowcost，lowcost[i]存放边(i,nearest[i])的权值
- 一维数组mark，标记顶点是否已经在生成树上
- **求解nearest与lowcost**





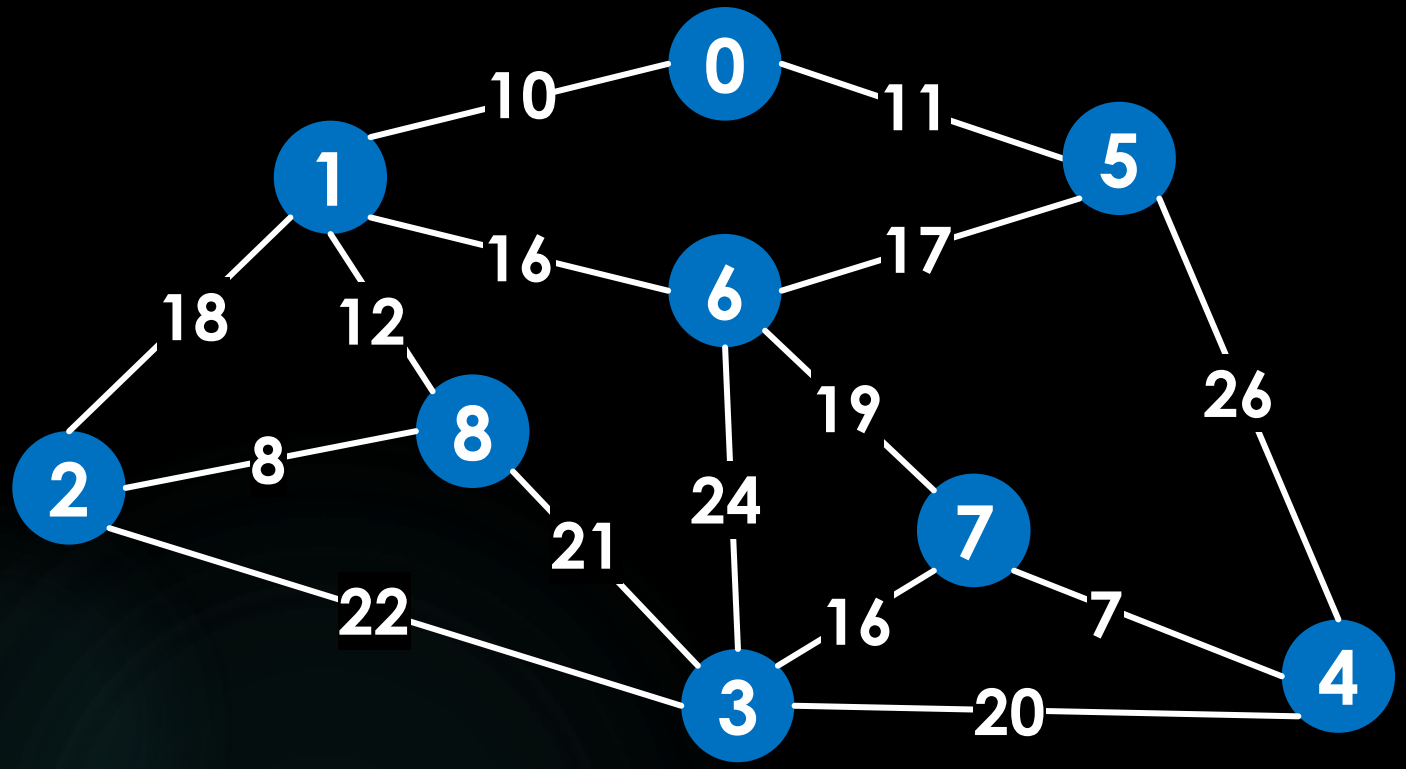
当顶点u加入到生成树之后，对nearest进行重新计算：
找到顶点u的边结点链表，对每个边结点v，检查如果w(u,v)<nearest[v]，则令
nearest[v]=u, lowcost[v]=w(u,v)

构造最小代价生成树的算法

- 普里姆算法 (Prim)
- 克鲁斯卡尔算法 (Kruskal)

克鲁斯卡尔的思想

通过找n-1条不构成回路的最小权值边，来得到最小代价生成树。

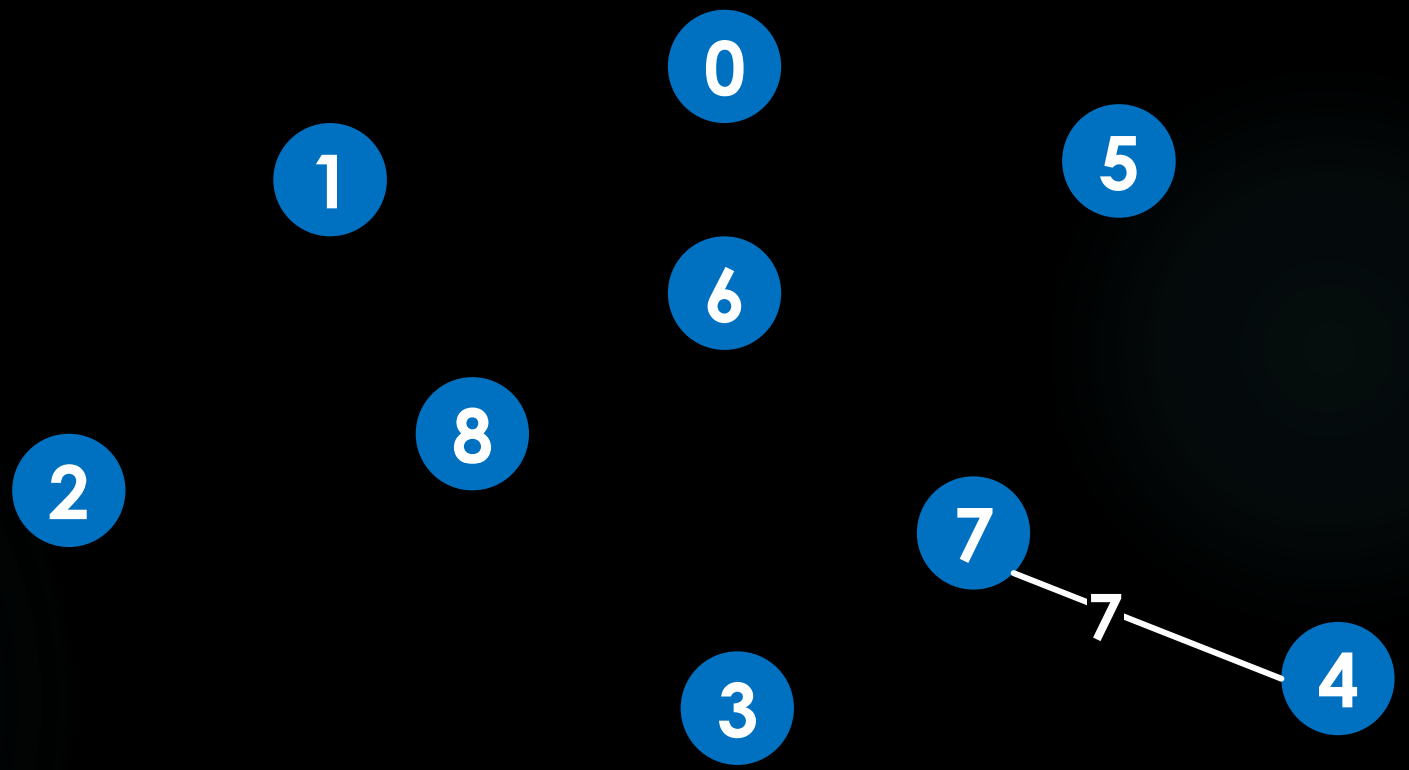


u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

先将边按照权值从小到大排列
初始时最小生成树T=(V,{ })

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

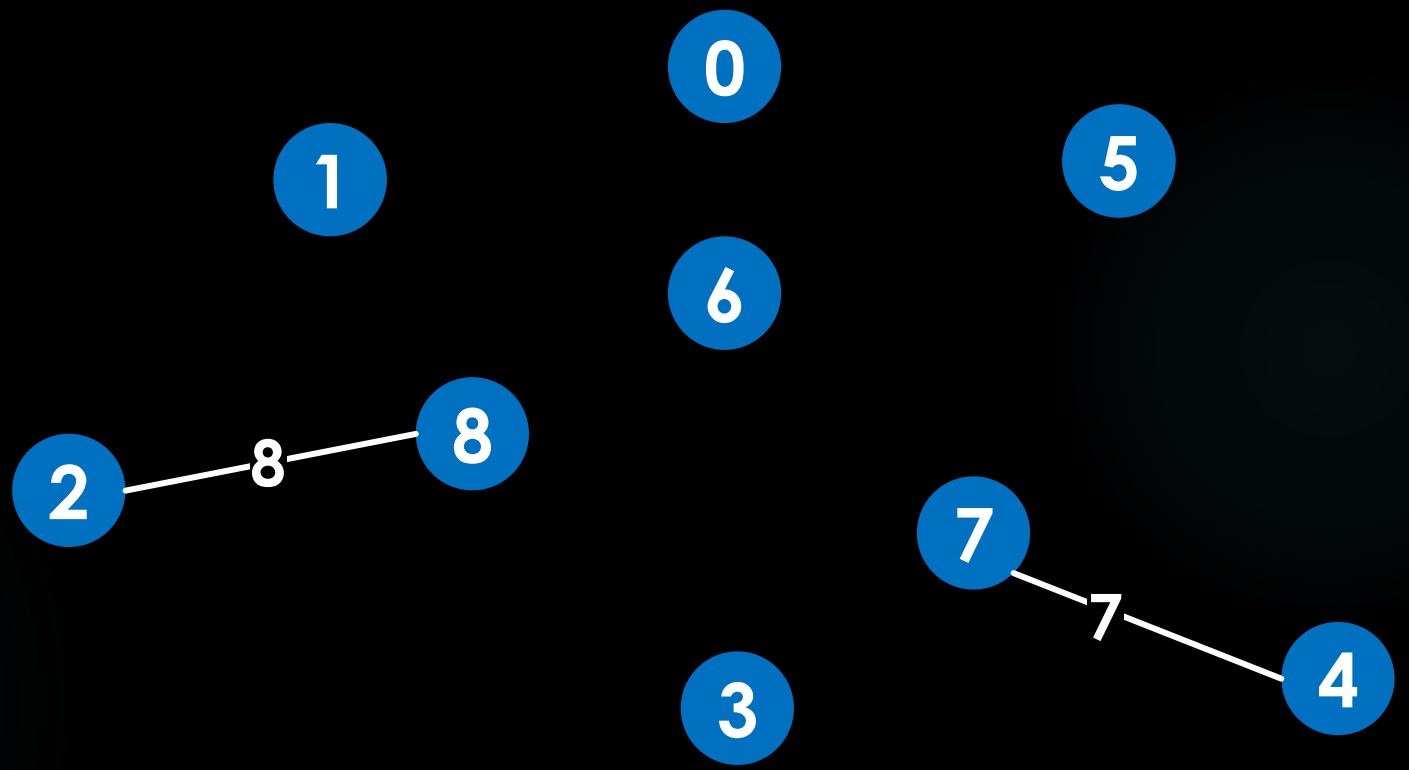


最小生成树

T=(V,{(7,4)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

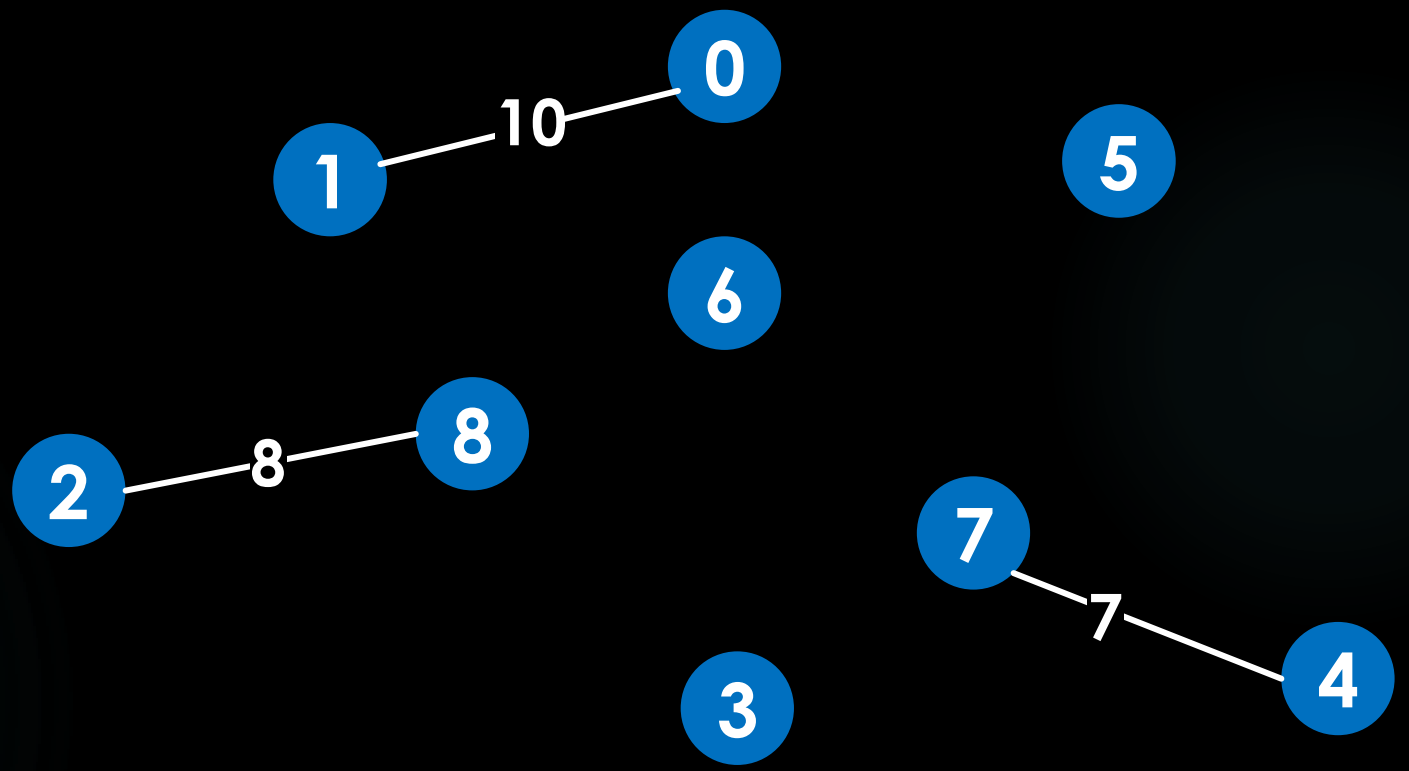


最小生成树

T=(V,{(7,4),(2,8)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

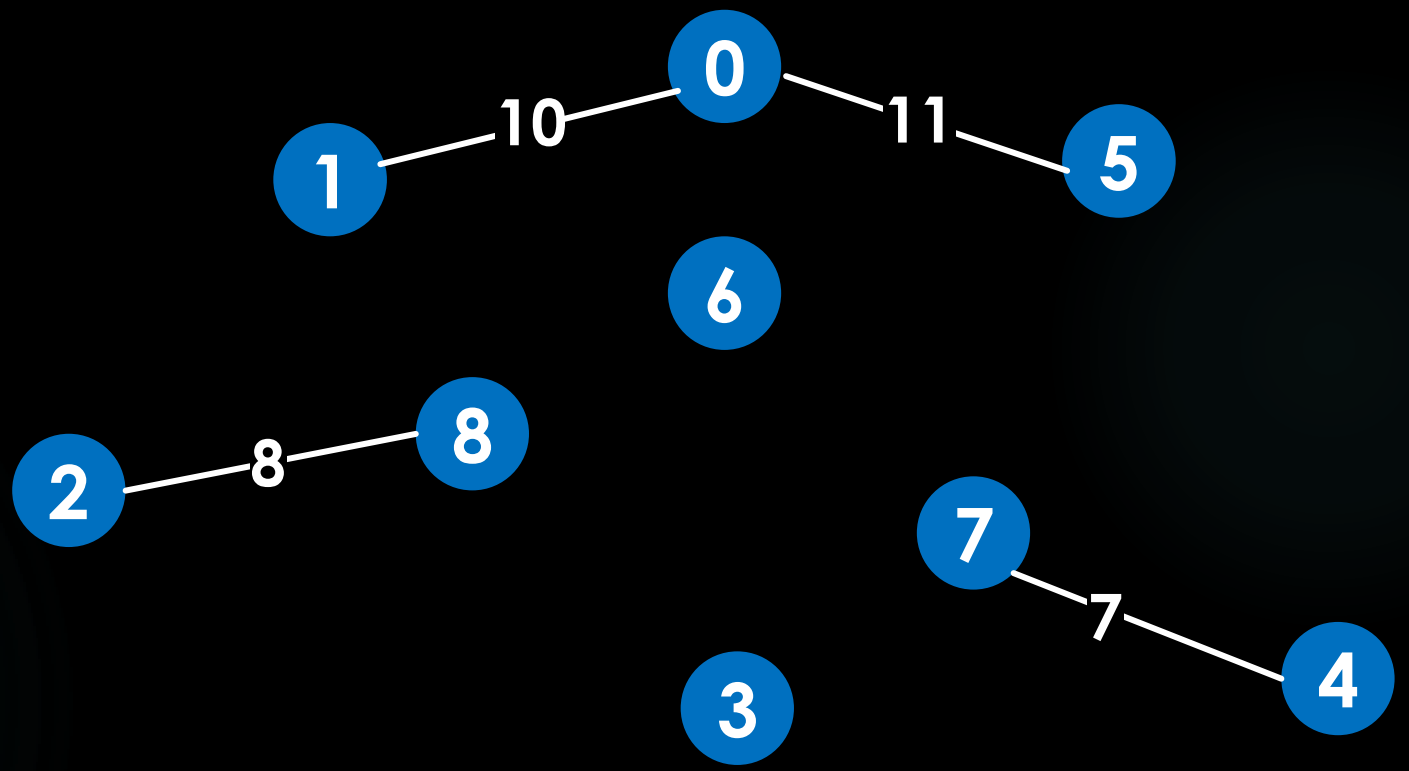


最小生成树

T=(V,{(7,4),(2,8),(1,0)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

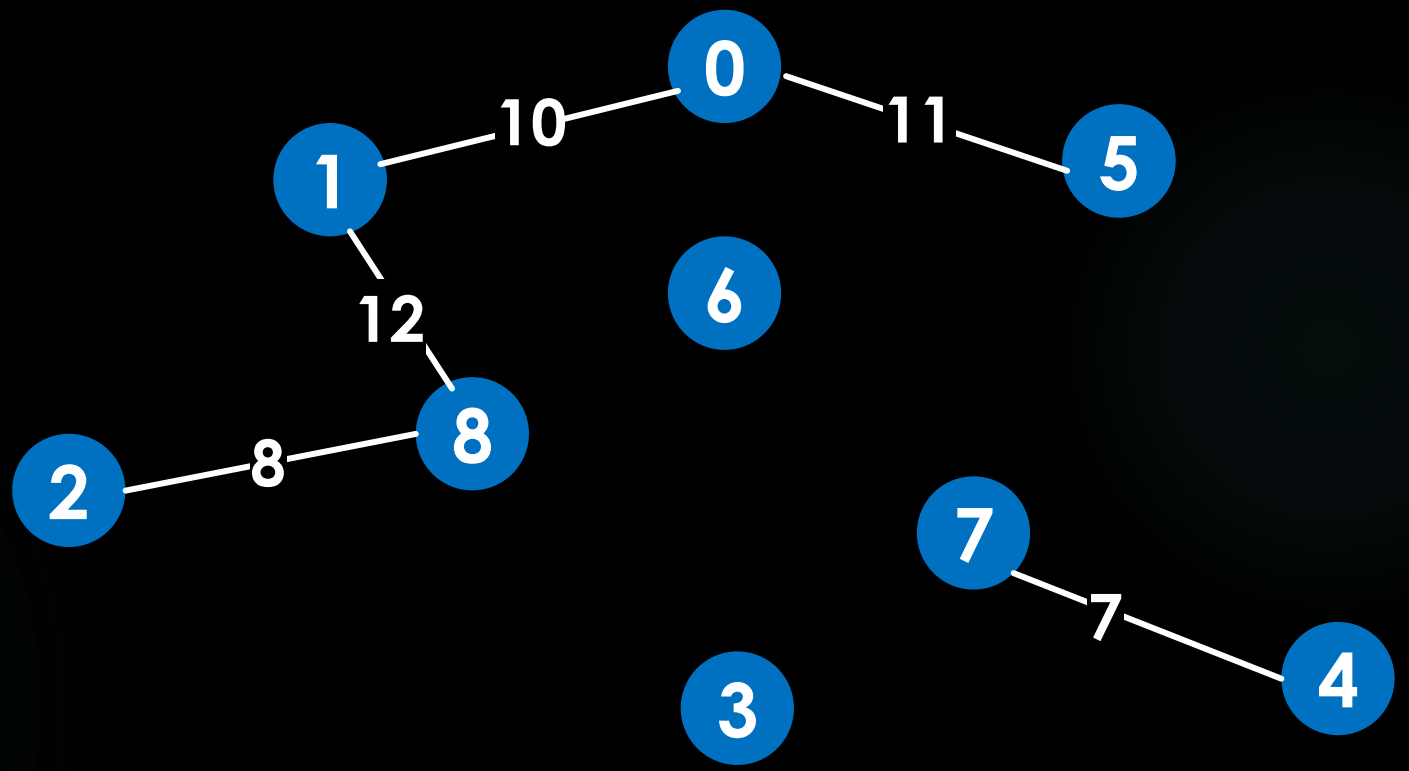


最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

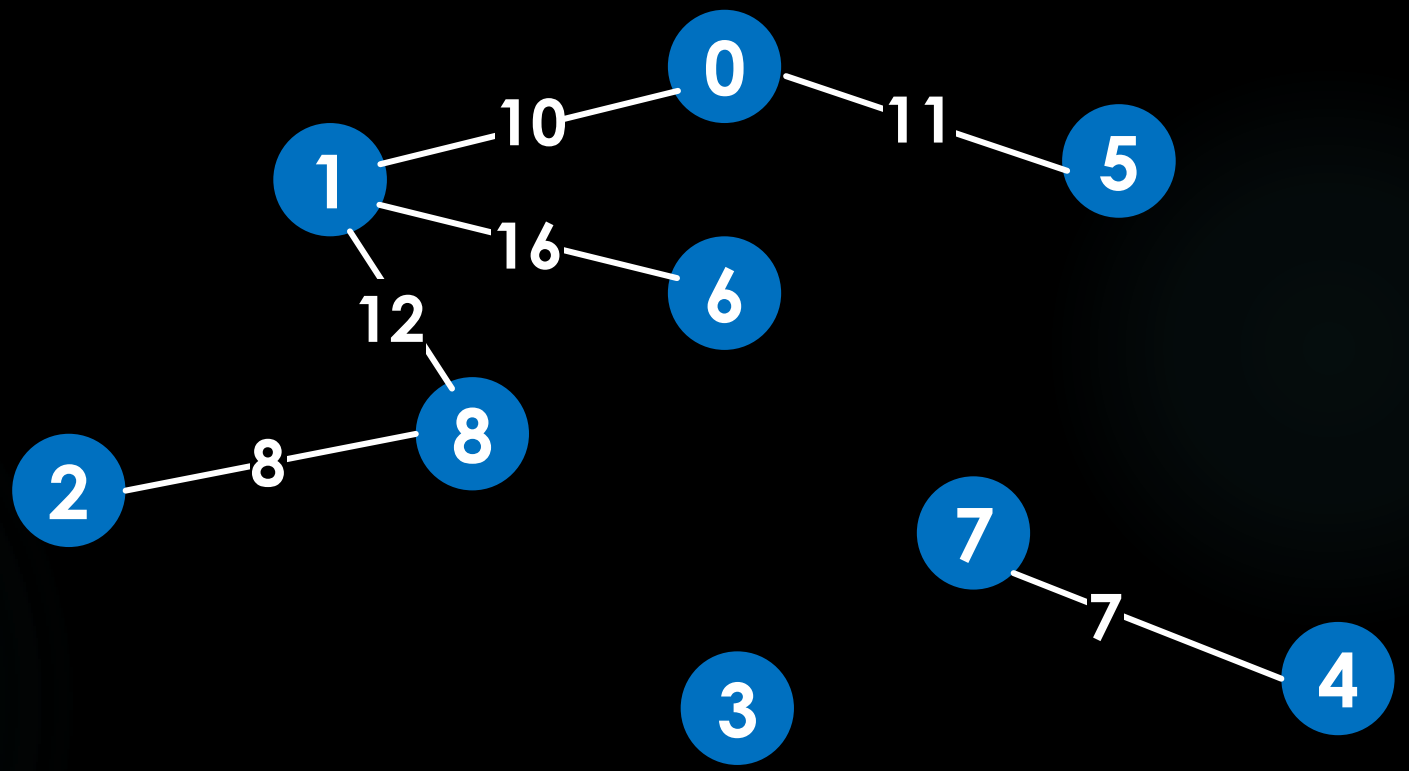


最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5),(1,8)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

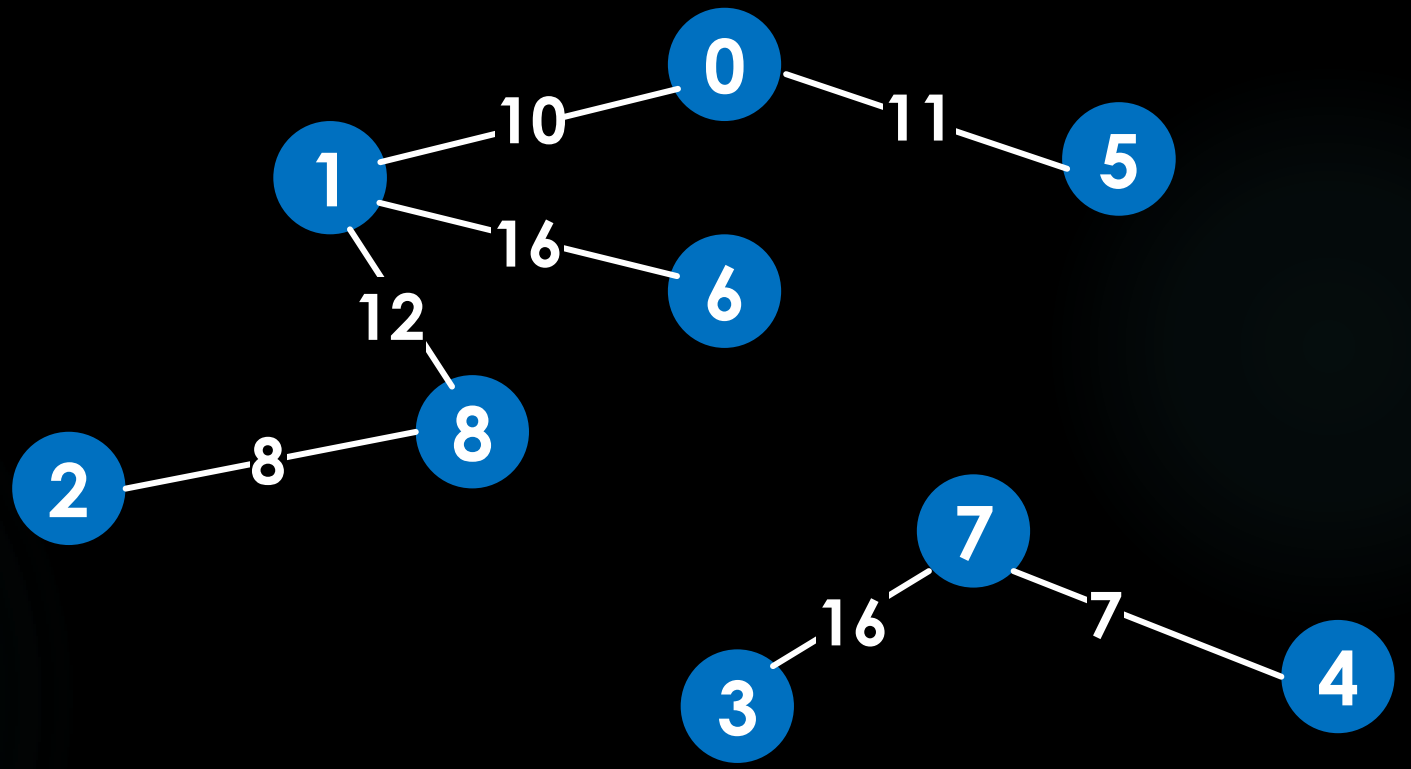


最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

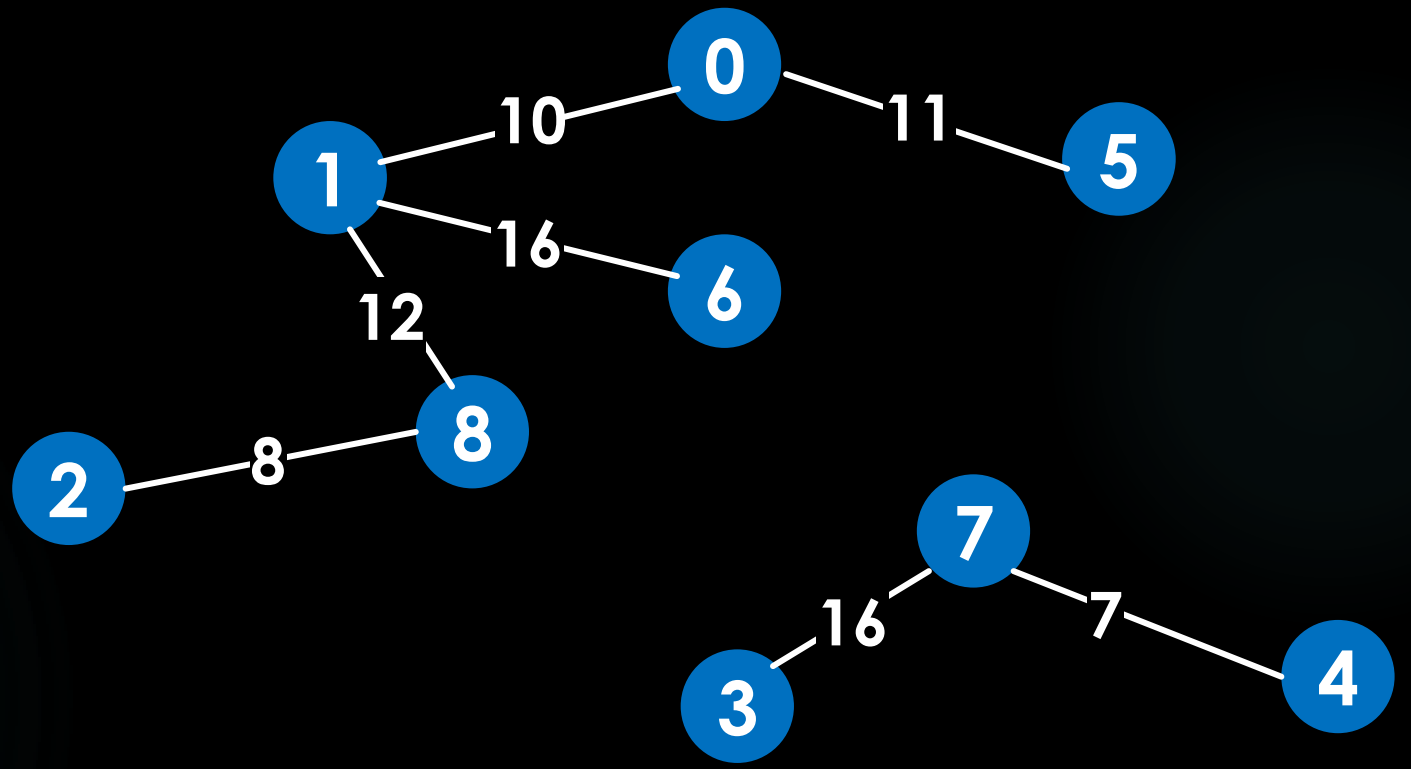


最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

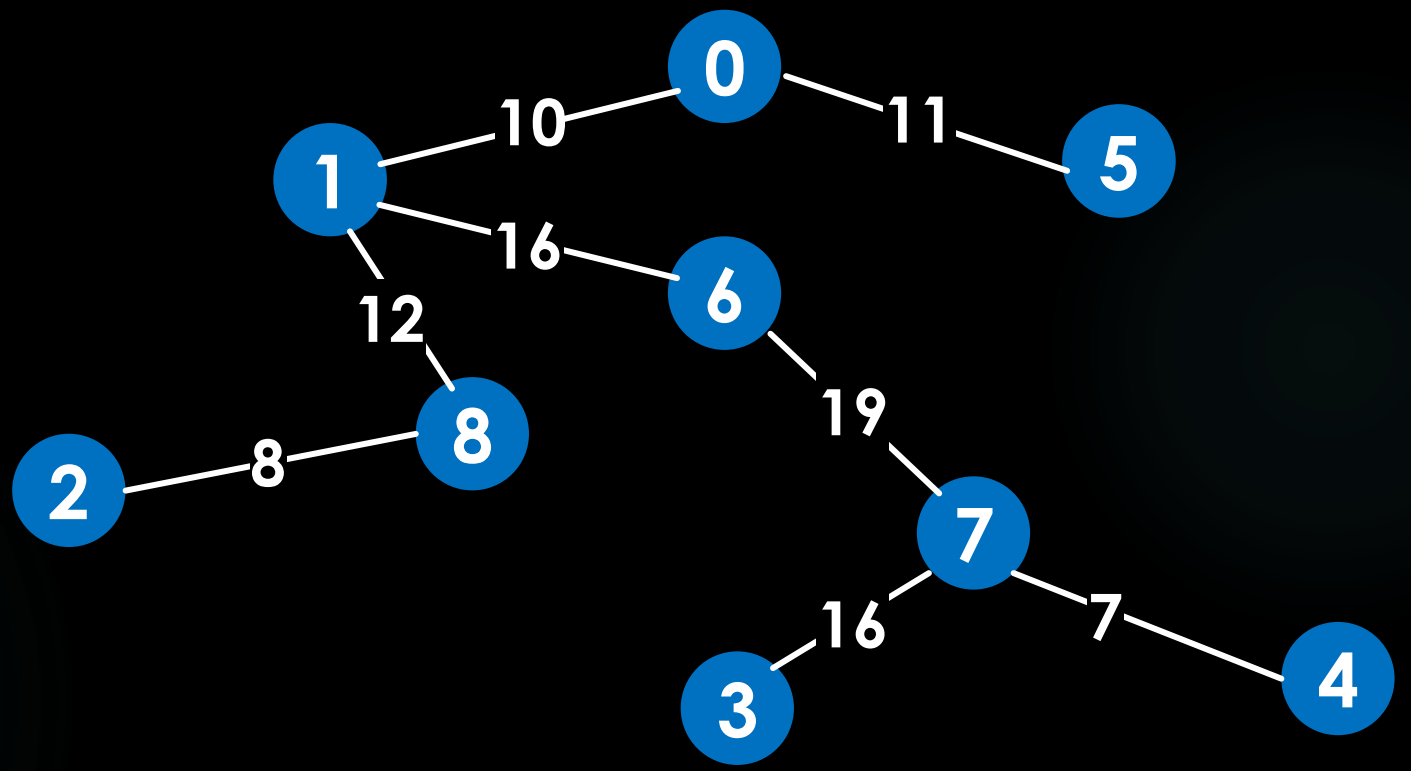


最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路

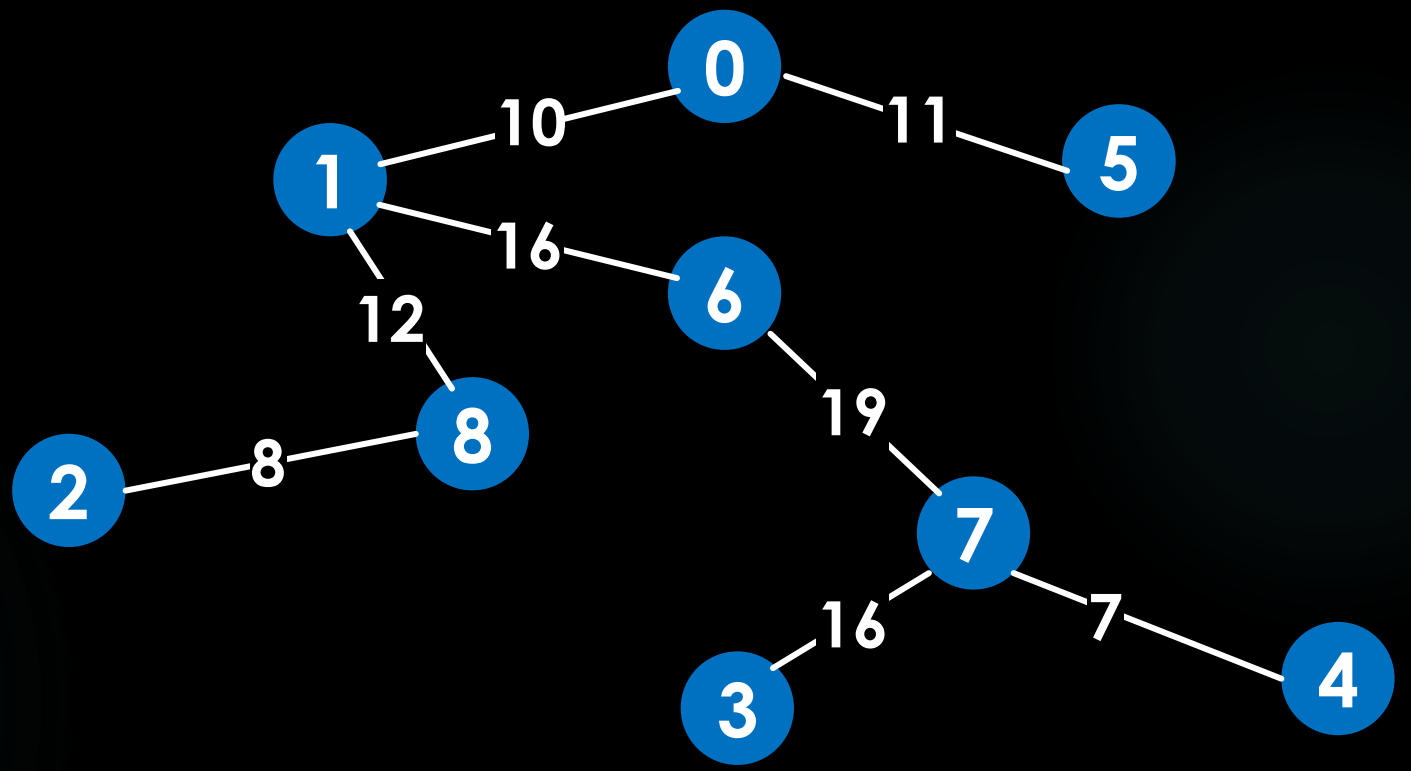


最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7),(6,7)})

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

将权值最小的边加入T，且保证加入后不会造成T中有回路



最小生成树

T=(V,{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7),(6,7)})

停止条件：T中放入n-1个边

如果对边按照权值排序？

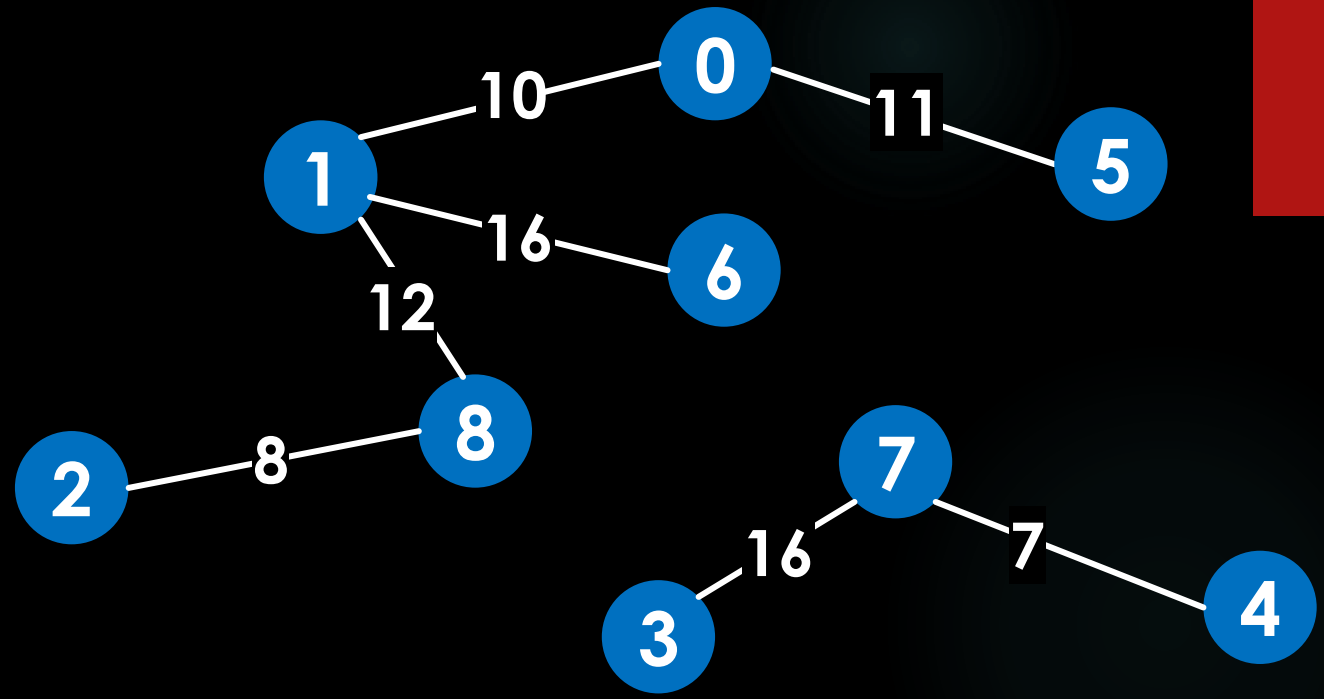
方法一：利用优先权队列，将权值看做优先级，每次serve得到队列中权值小的边

方法二：单独建立一个edge数组，存入所有边，然后选择排序算法进行排序

如何判断向T中加入一个边(u,v)不会造成回路？

1. 因为u和v都在生成树上，可能造成回路，加入前进行如下检查：
   - 设置顶点集合S={u}和边集合P={}
   - 循环：对于T中任意一条边(x,y)
     - 如果x在S中，则将y加入S，(x,y)加入P
     - 如果y在S中，则将x加入S，(x,y)加入P
     - 如果S中出现v，则说明边(u,v)不能加入到T，否则产生回路，跳出循环
     - 直到（T的边集合-P）中的边的顶点与S没有交集
   - 如果上面循环没有提前跳出，说明不会造成回路

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

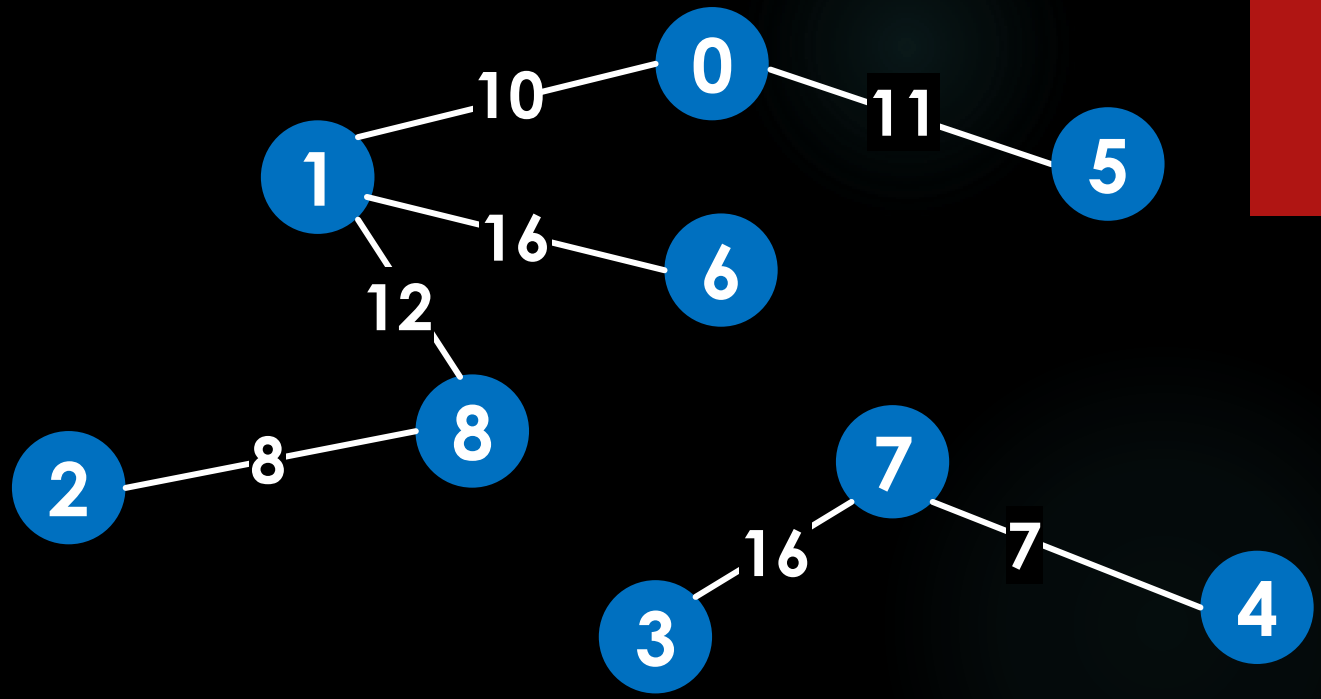


最小生成树T=(V,
{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

是否将(6,5)加入T?

1. S={5}

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

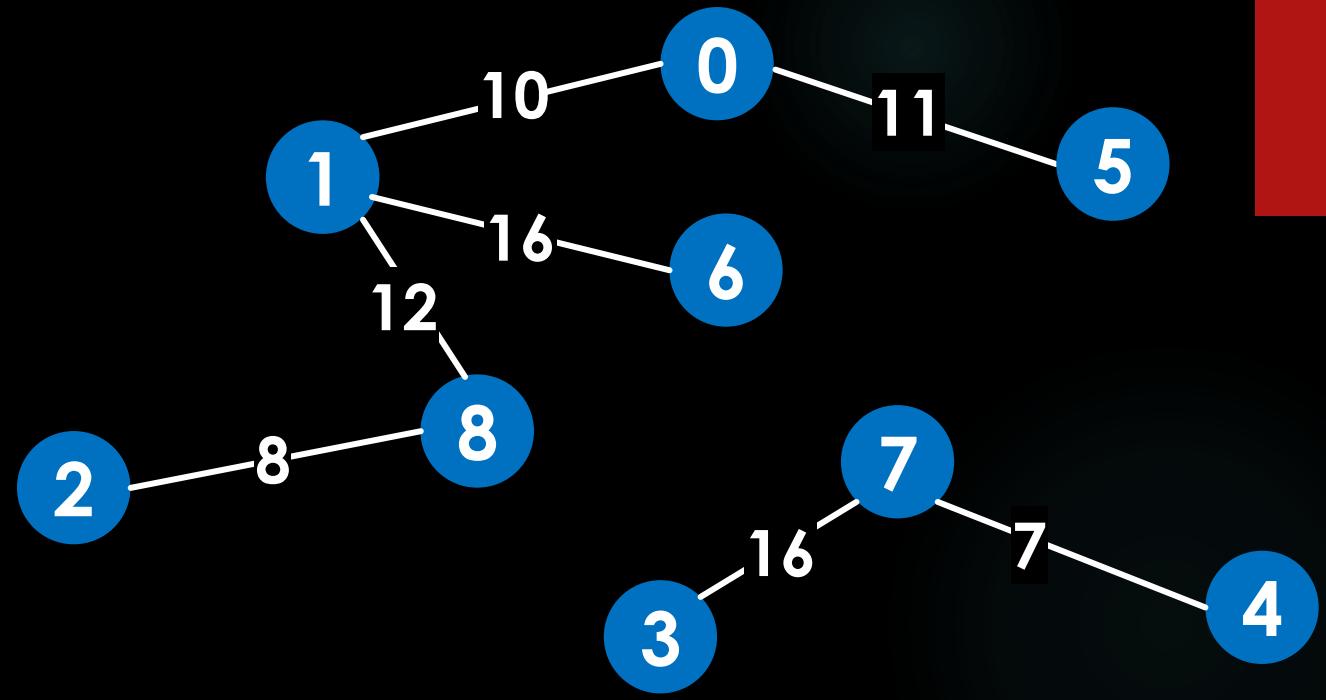


最小生成树T=(V,
{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

是否将(6,5)加入T?

1. S={5}
2. S={5,0}

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

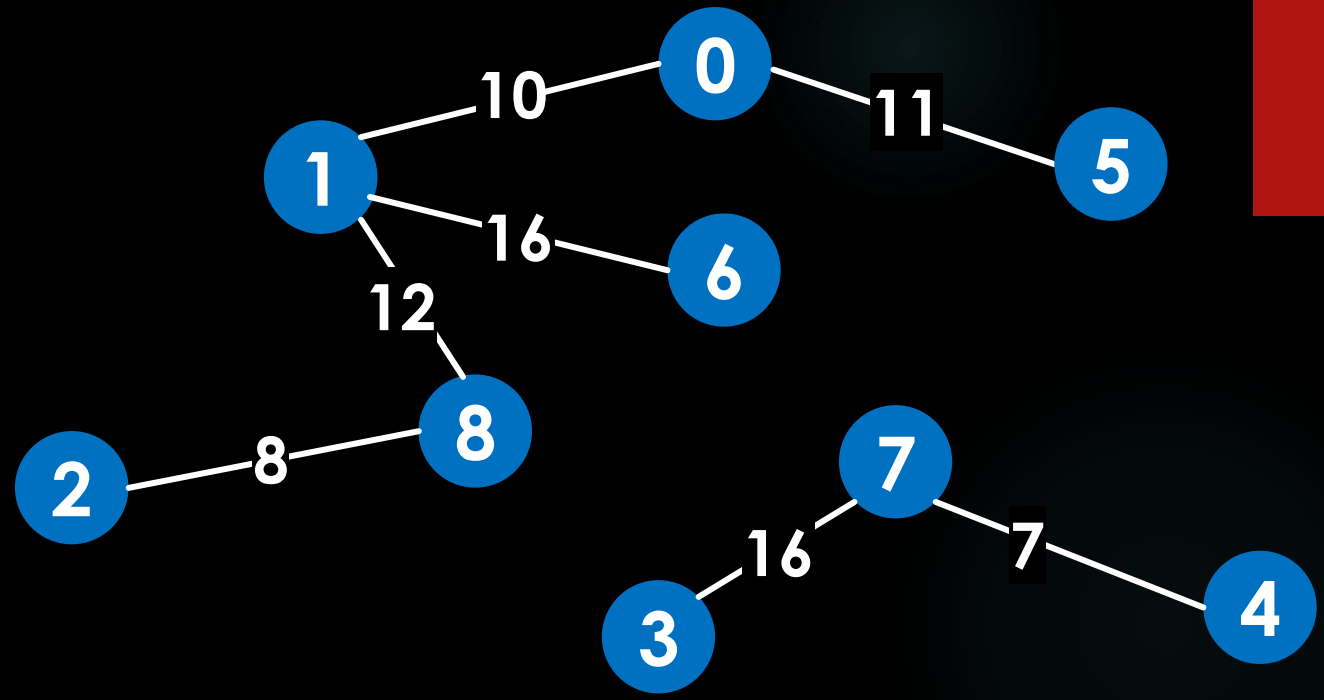


最小生成树T=(V,
{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

是否将(6,5)加入T?

1. S={5}
2. S={5,0}
3. S={5,0,1}

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

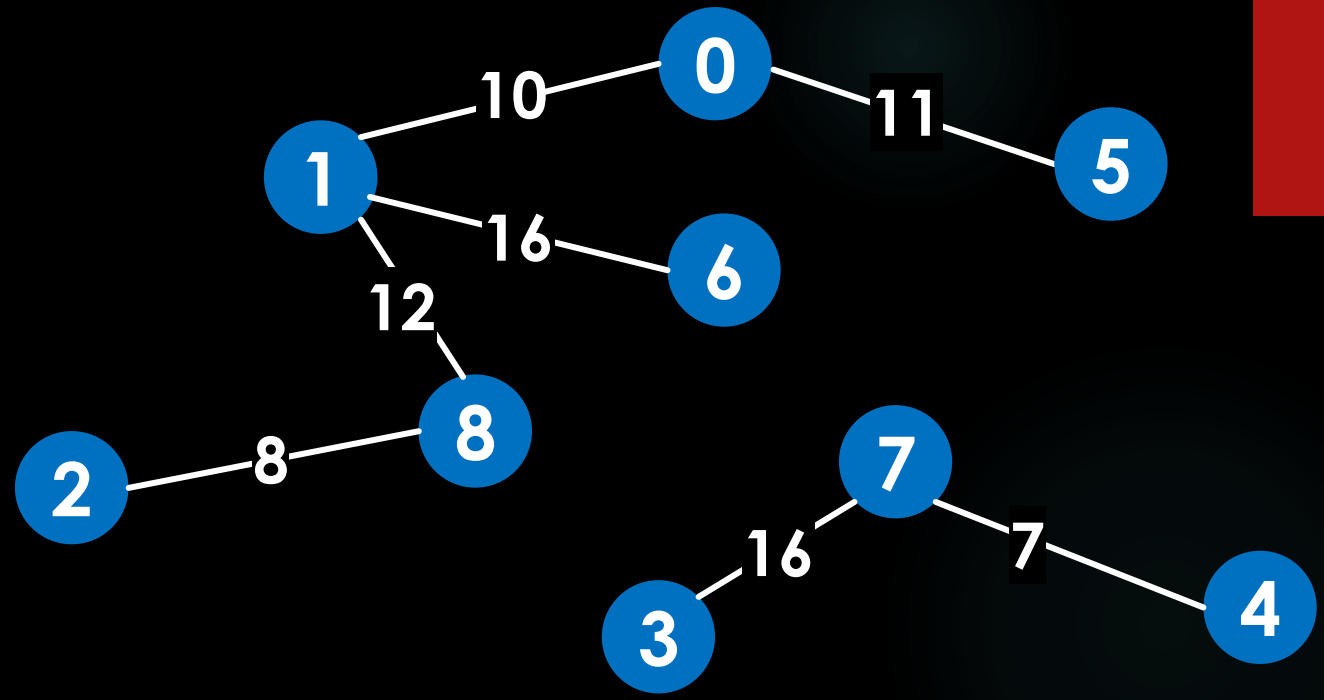


最小生成树T=(V,
{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

是否将(6,5)加入T?

1. S={5}
2. S={5,0}
3. S={5,0,1}
4. S={5,0,1,8}

u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26



最小生成树T=(V,
{(7,4),(2,8),(1,0),(0,5),(1,8),(1,6),(3,7)})

是否将(6,5)加入T?

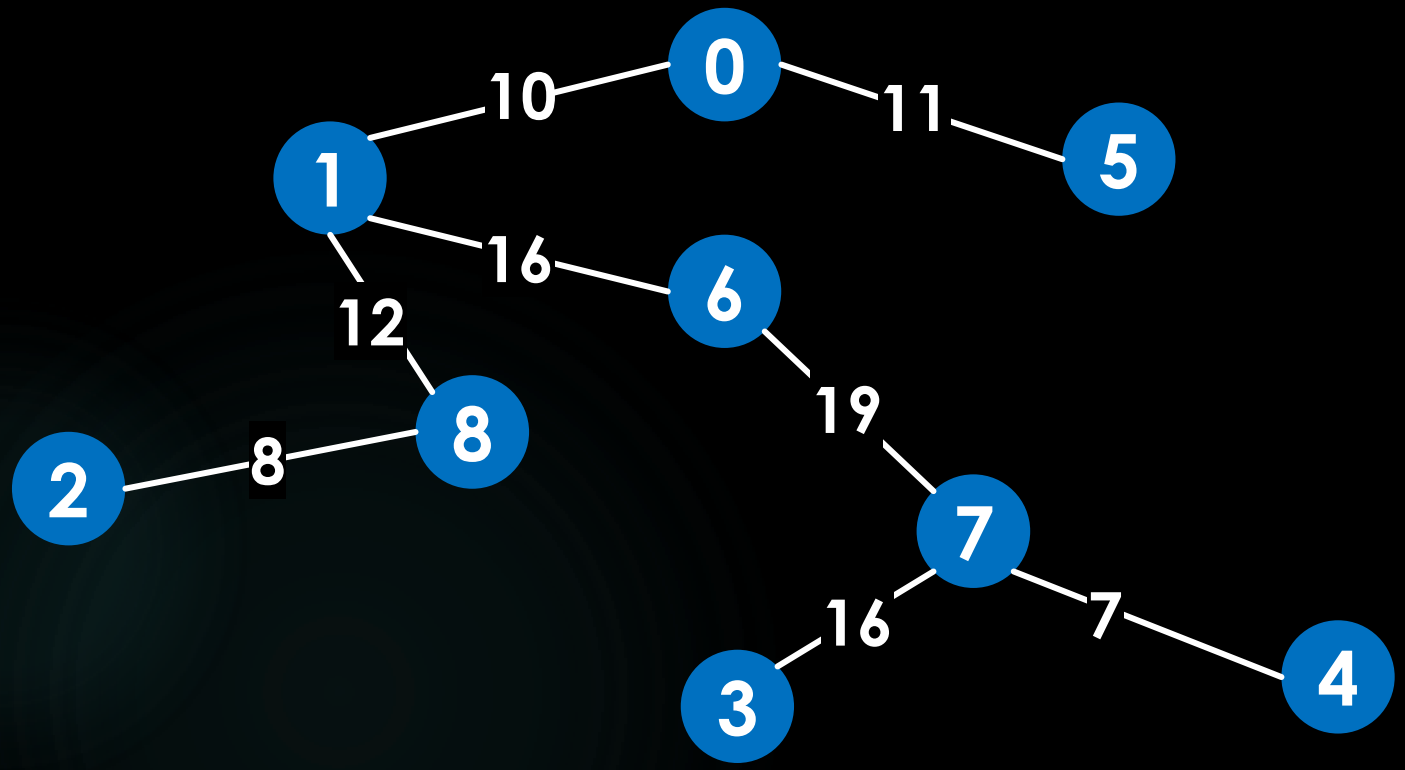
1. S={5}
2. S={5,0}
3. S={5,0,1}
4. S={5,0,1,8}
5. S={5,0,1,8,6}

造成回路，不能加入

设G=(V,E)是带权的连通图，T=(V’,E’)是正在构造中的生成树，即V’={ }，E’={ }。

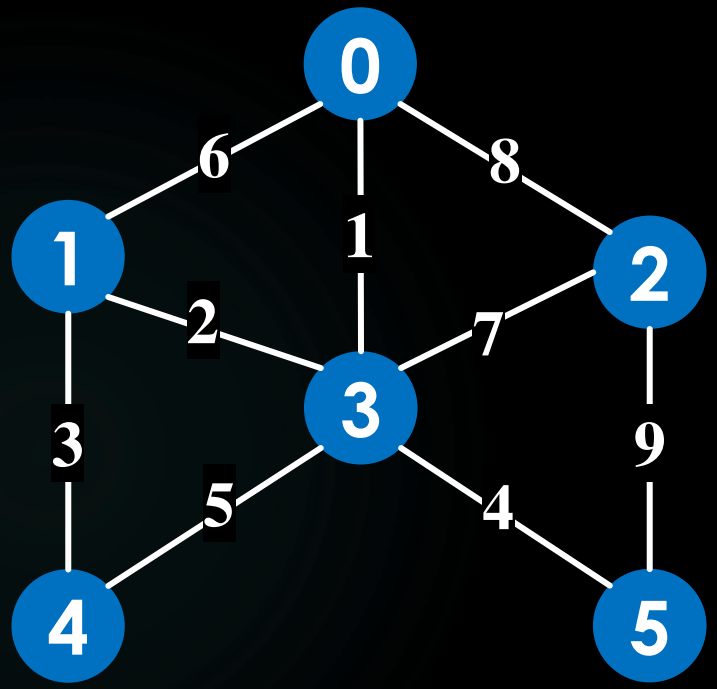
从初始状态开始，重复执行下列运算：
(1)在E中选一条权值最小的边(u,v)
(2)若在T中加入(u,v)后不形成回路，则加入T，并从E中删除；否则继续选另一条边。
重复(1)、(2)，直到T中包含n-1条边为止。
这时T=(V’,E’)是图G的一棵最小代价生成树。

一个简单的手工画最小生成树的方法



u	v	权值
7	4	7
2	8	8
1	0	10
0	5	11
1	8	12
1	6	16
3	7	16
6	5	17
1	2	18
6	7	19
3	4	20
8	3	21
2	3	22
6	3	24
5	4	26

连通图如图所示，使用普里姆算法分别以3、4、5为源点,构建该图的最小代价生成树。

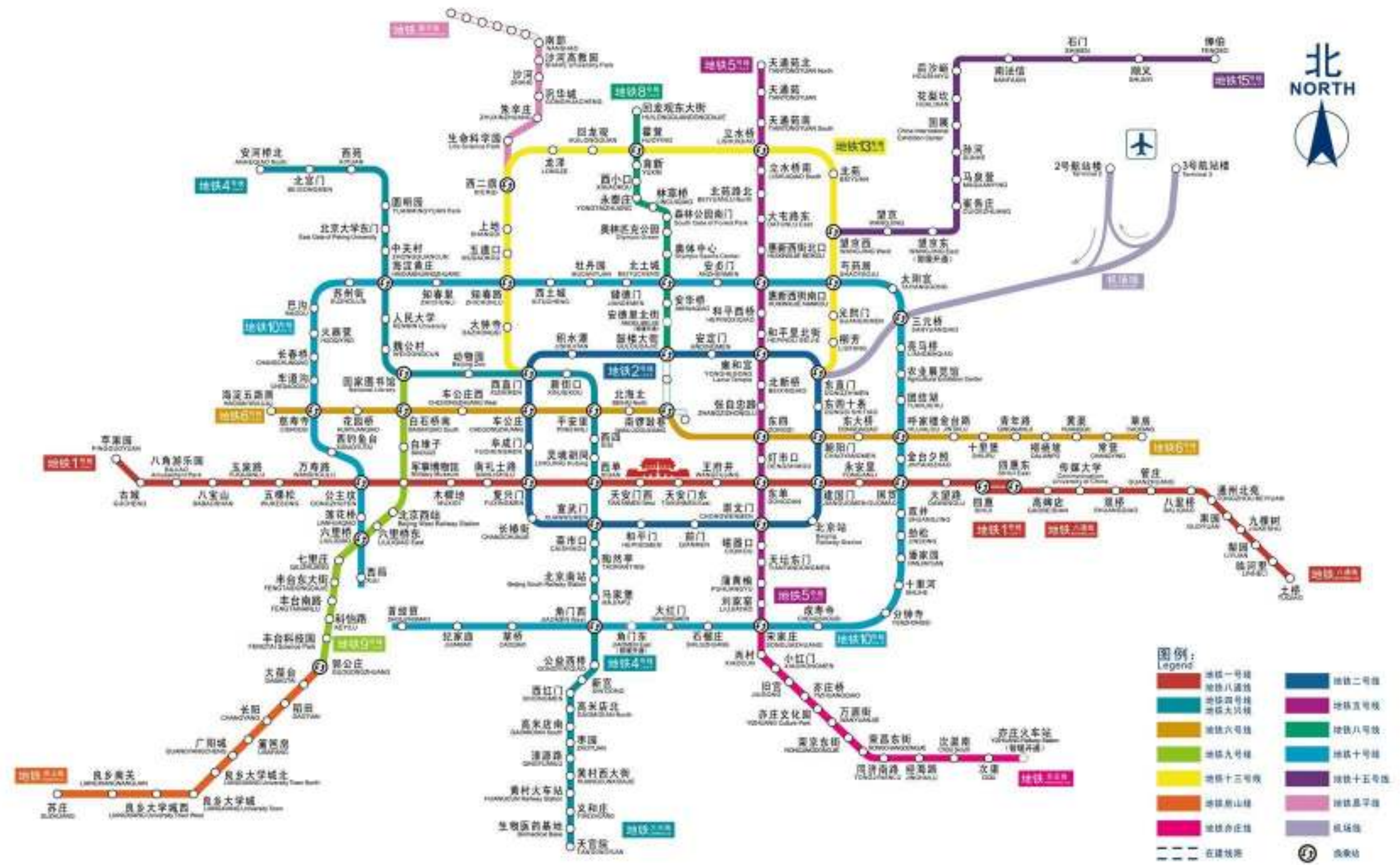


克鲁斯卡尔算法？

图

目录

- 图的基本概念
- 图的存储结构
- 图的遍历
- 拓扑排序
- 关键路径
- 最小代价生成树
- 单源最短路径



北
NORTH
地铁1号线
地铁2号线
地铁4号线
地铁5号线
地铁6号线
地铁8号线
地铁9号线
地铁10号线
地铁13号线
地铁15号线
苹果园
古城
八角游乐园
八宝山
玉泉路
五棵松
万寿路
公主坟
军事博物馆
木樨地
南礼士路
复兴门
西单
天安门西
天安门东
王府井
东单
建国门
永安里
国贸
大望路
四惠
四惠东
高碑店
传媒大学
双桥
管庄
八里桥
通州北苑
果园
九棵树
梨园
临河里
土桥
安河桥北
北宫门
西苑
圆明园
北京大学东门
中关村
海淀黄庄
人民大学
魏公村
国家图书馆
动物园
西直门
新街口
平安里
西四
灵境胡同
宣武门
菜市口
陶然亭
北京南站
马家堡
角门西
公益西桥
新宫
西红门
高米店北
高米店南
枣园
清源路
黄村西大街
黄村火车站
义和庄
生物医药基地
天宫院
天通苑北
天通苑
天通苑南
立水桥
立水桥南
北苑路北
大屯路东
惠新西街北口
惠新西街南口
和平西桥
和平里北街
雍和宫
北新桥
张自忠路
东四
灯市口
崇文门
磁器口
天坛东门
蒲黄榆
刘家窑
宋家庄
南锣鼓巷
北海北
积水潭
鼓楼大街
安定门
东直门
东四十条
朝阳门
北京站
前门
和平门
长椿街
南沙
沙河
沙河高教园
巩华城
朱辛庄
生命科学园
西二旗
上地
五道口
知春路
大钟寺
龙泽
回龙观
霍营
回龙观东大街
育新
西小口
永泰庄
林萃桥
森林公园南门
奥林匹克公园
奥体中心
北土城
安华桥
安德里北街
什刹海
健德门
牡丹园
西土城
知春里
苏州街
巴沟
火器营
长春桥
车道沟
慈寿寺
花园桥
白石桥南
车公庄西
车公庄
阜成门
海淀五路居
西钓鱼台
白堆子
六里桥
六里桥东
北京西站
七里庄
丰台东大街
丰台南路
科怡路
丰台科技园
郭公庄
大葆台
稻田
长阳
篱笆房
广阳城
良乡大学城北
良乡大学城
良乡大学城西
良乡南关
苏庄
西局
草桥
纪家庙
首经贸
角门东
大红门
石榴庄
成寿寺
分钟寺
十里河
潘家园
劲松
双井
金台夕照
团结湖
呼家楼
农业展览馆
亮马桥
三元桥
太阳宫
芍药居
光熙门
柳芳
望京西
望京
望京东
北苑
东大桥
金台路
十里堡
青年路
褡裢坡
黄渠
常营
草房
俸伯
顺义
石门
南法信
后沙峪
花梨坎
国展
孙河
马泉营
崔各庄
2号航站楼
3号航站楼
机场线
肖村
小红门
旧宫
亦庄桥
亦庄文化园
万源街
荣京东街
荣昌东街
同济南路
经海路
次渠南
次渠
亦庄火车站
图例
Legend

最短路径

最短路径是一种重要的图算法。在生活中常常遇到这样的问题，两地之间是否有路可通？在有几条通路的情况下，哪一条路最短？

无权值图上顶点u和v的最短路径长度：路径上边的数量
有权值图上顶点u和v的最短路径长度：路径上边的权值之和

最短路径算法

- **单源最短路径的迪杰斯特拉（Djikstra）算法**

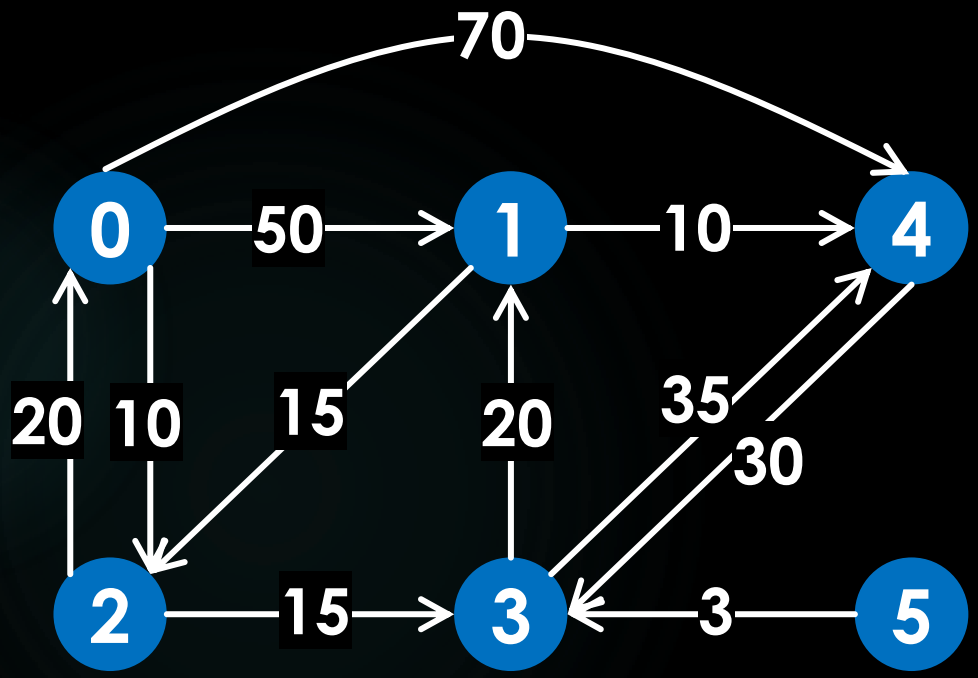
  给定一个顶点，求其与剩下其他顶点之间的最短路径

- **求所有顶点之间的最短路径的弗洛伊德（ Floyd）算法**

  求解所有两两顶点之间的最短路径

单源最短路径算法-迪杰斯特拉

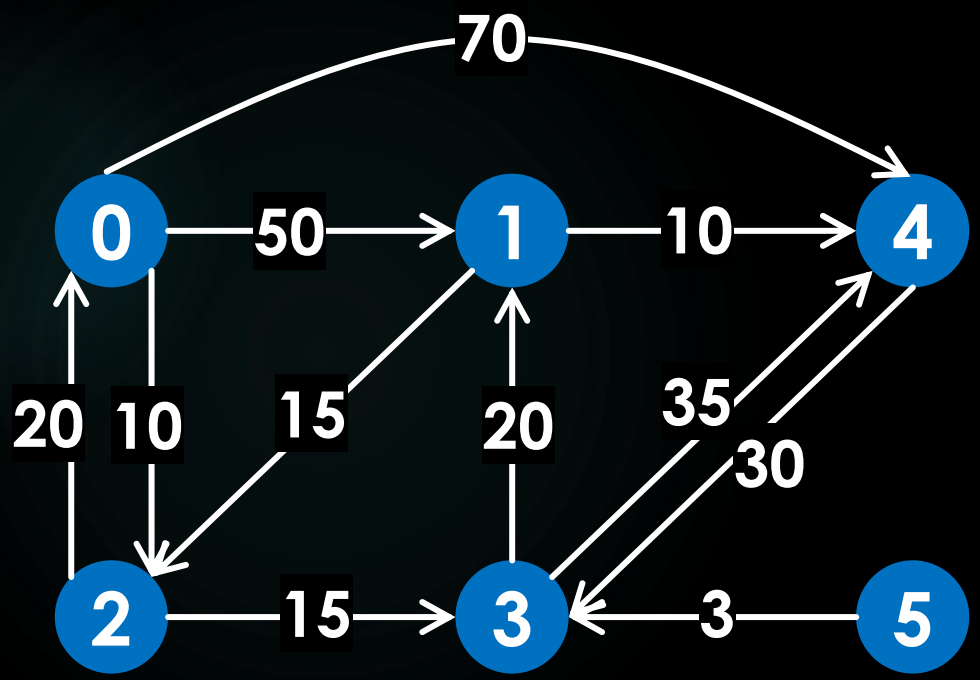
给定带权有向图G=(V,E)，源点v∈V，求从顶点v到顶点集V中其余各顶点的最短路径



源点	终点	最短路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

单源最短路径算法-迪杰斯特拉

算法思想：首先求得长度最短的一条最短路径，再求得长度次短的一条最短路径，依次类推，按照路径长度递增的次序产生最短路径，点之间的最短路径都已求得为止

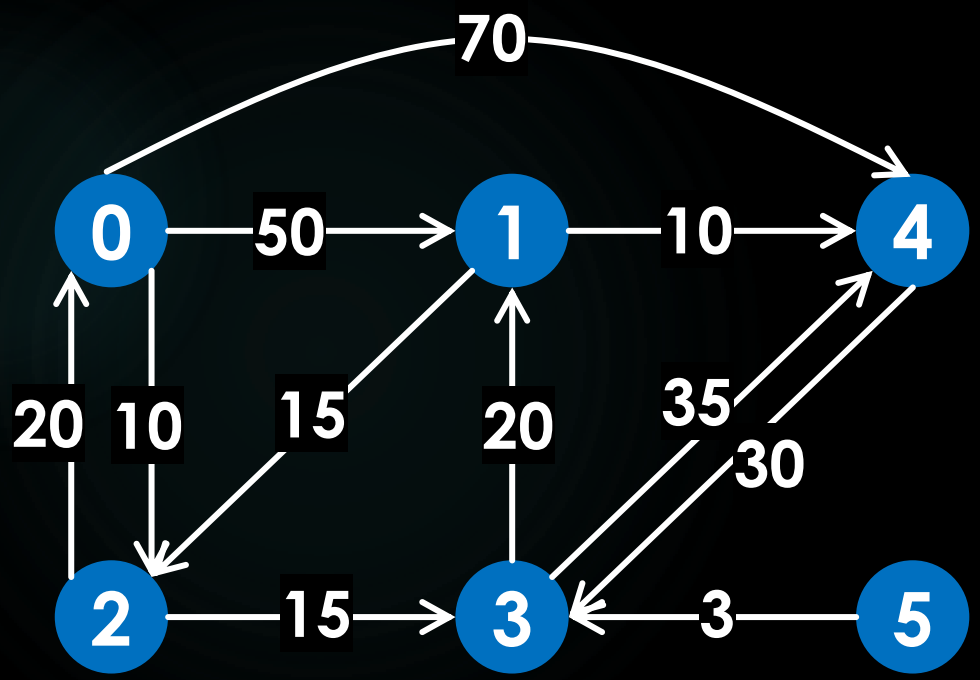


源点	终点	最短路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

单源最短路径算法-迪杰斯特拉

首先，求0到其他顶点，只经过1条边的最短路径

此时求得的多个路径中，长度最短的一定是一条最短路径



源点	终点	路径	路径长度
0	1	0,1	50
	2	0,2	10
	3	-	∞
	4	0,4	70
	5	-	∞

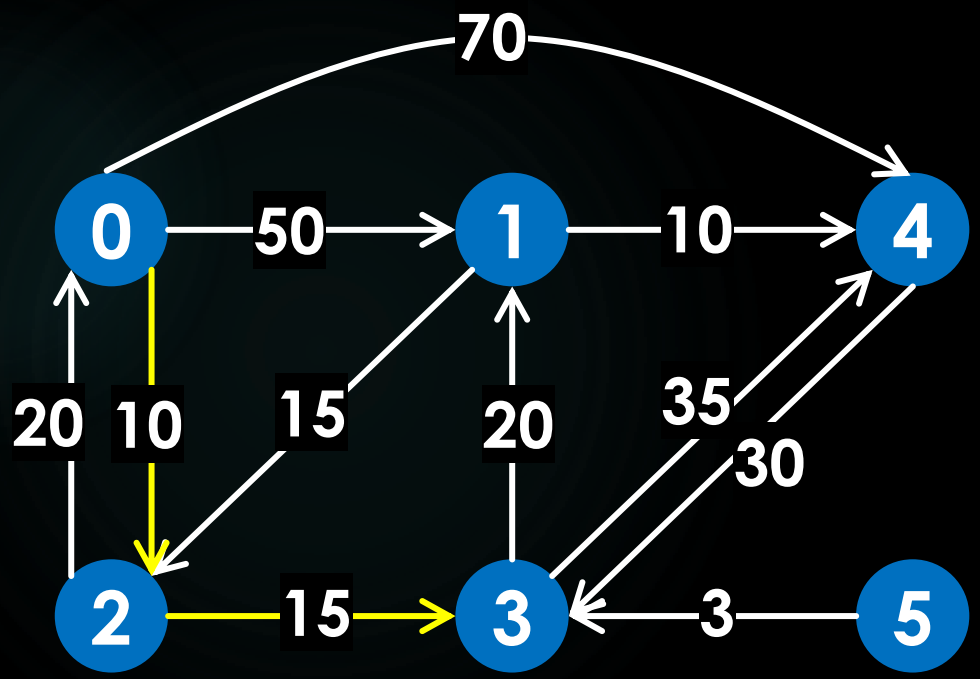
<0,2>一定是0到2的最短路径，为什么？

单源最短路径算法-迪杰斯特拉

当前从0出发的当前最短路径是<0,2>

- 已知0通过2可以到达3和0 （0造成回路，不考虑）
- 0到达3，经过2的路径为<0,2,3>，长度是10+15=25

比<0,3>路径短，可更新！



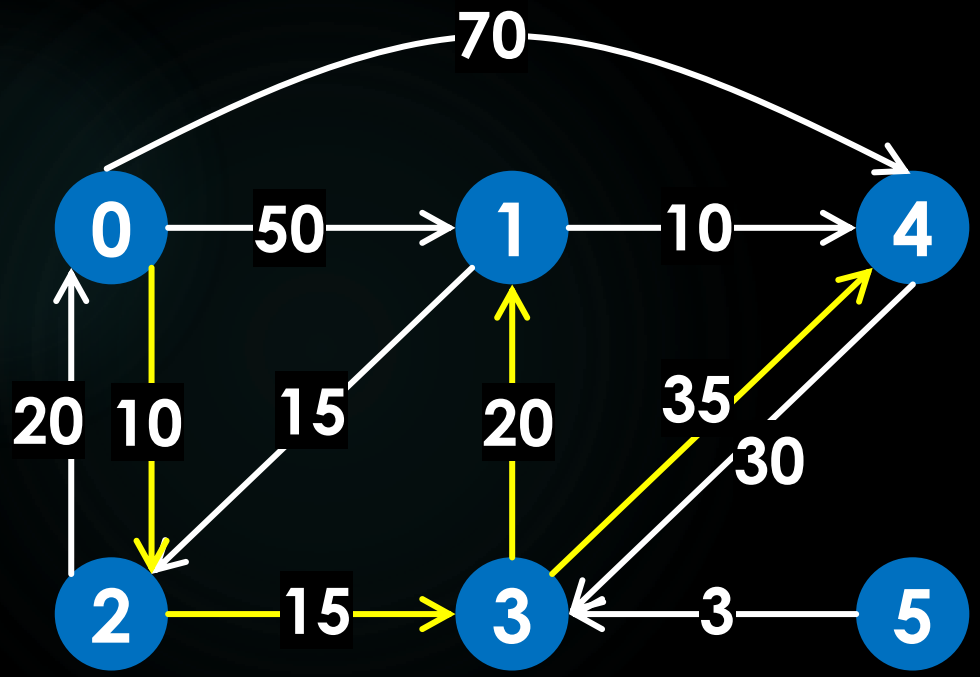
源点	终点	路径	路径长度
0	1	0,1	50
	2	0,2	10
	3	0,2,3	25
	4	0,4	70
	5	-	∞

<0,2,3>一定是0到3的最短路径，为什么？

单源最短路径算法-迪杰斯特拉

当前从0出发的当前最短路径是<0,2,3> <0,2>已经考察过

- 已知0通过2,3可以到达4、1
- 0到达4，经过2,3的路径为<0,2,3,4>，长度是25+35=60 比<0,4>路径短
- 0到达1，经过2,3的路径为<0,2,3,1>，长度为25+20=45 比<0,1>路径短



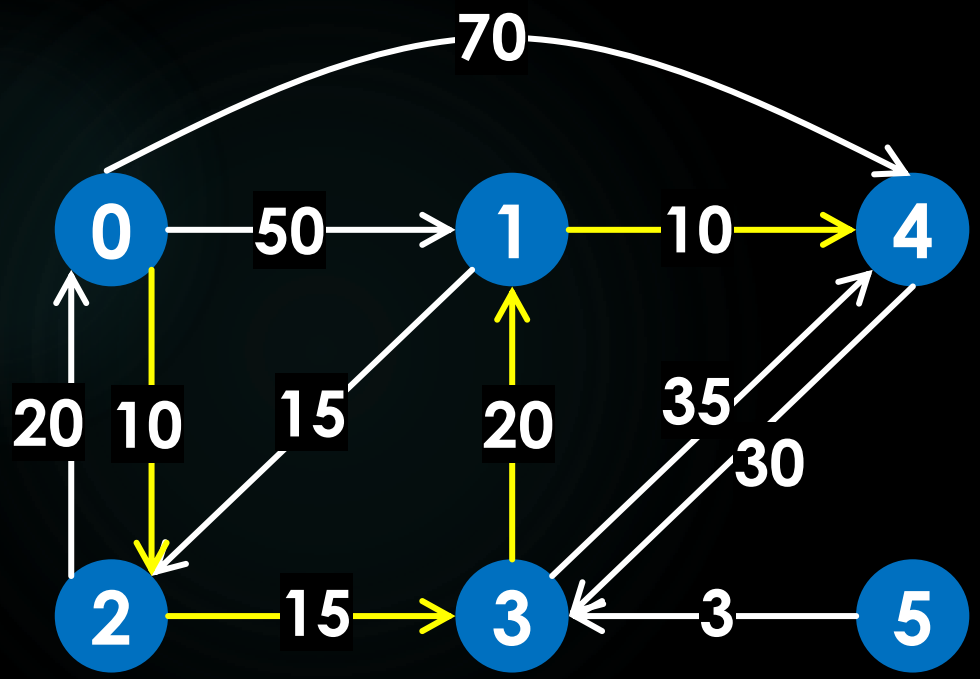
源点	终点	路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,4	60
	5	-	∞

<0,2,3,1>一定是0到1的最短路径，为什么？

单源最短路径算法-迪杰斯特拉

当前从0出发的当前最短路径是<0,2,3,1> <0,2>和<0,2,3>已经考察过

- 已知0通过2,3,1可以到达2和4 （2造成回路，不考虑）
- 0到达4，经过2,3,1的路径为<0,2,3,1,4>，长度是45+10=55 更短！

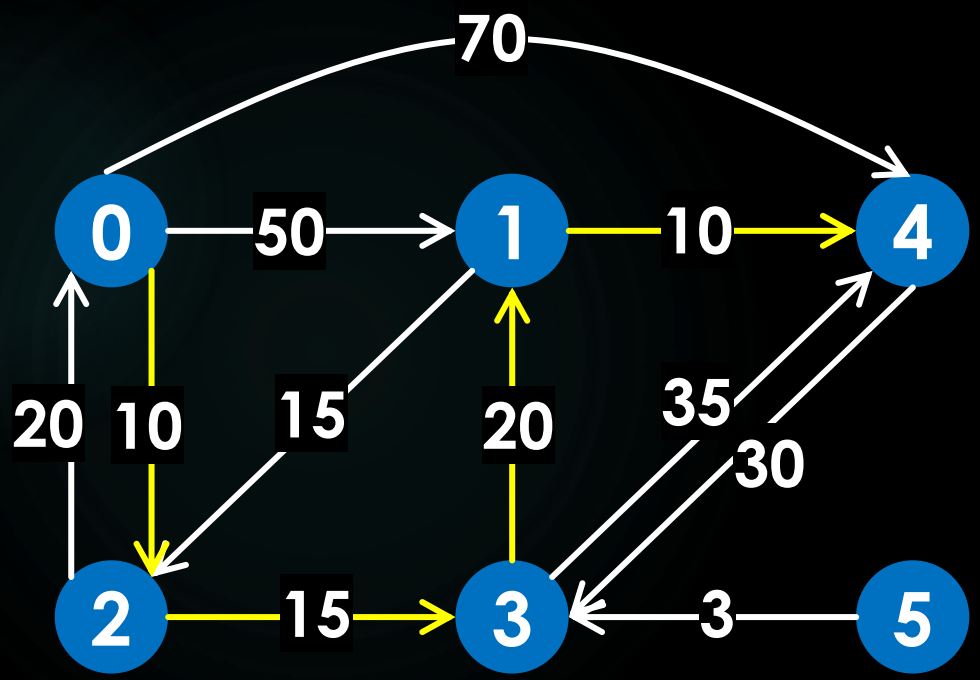


源点	终点	路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

单源最短路径算法-迪杰斯特拉

当前从0出发的当前最短路径是<0,2,3,1,4>

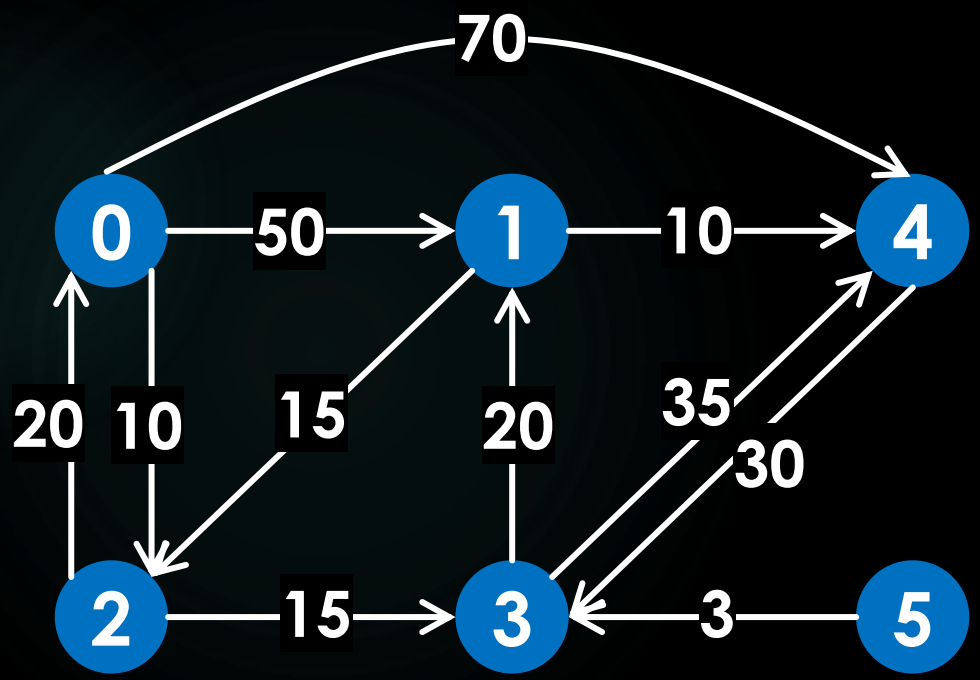
- 已知0通过2,3,1,4可以到达3（3造成回路，不考虑）



源点	终点	路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

单源最短路径算法-迪杰斯特拉

当前从0出发的无法到达5，算法停止
表中当前结果即为从0出发到各个顶点的最短路径



源点	终点	最短路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

迪杰斯特拉算法实现

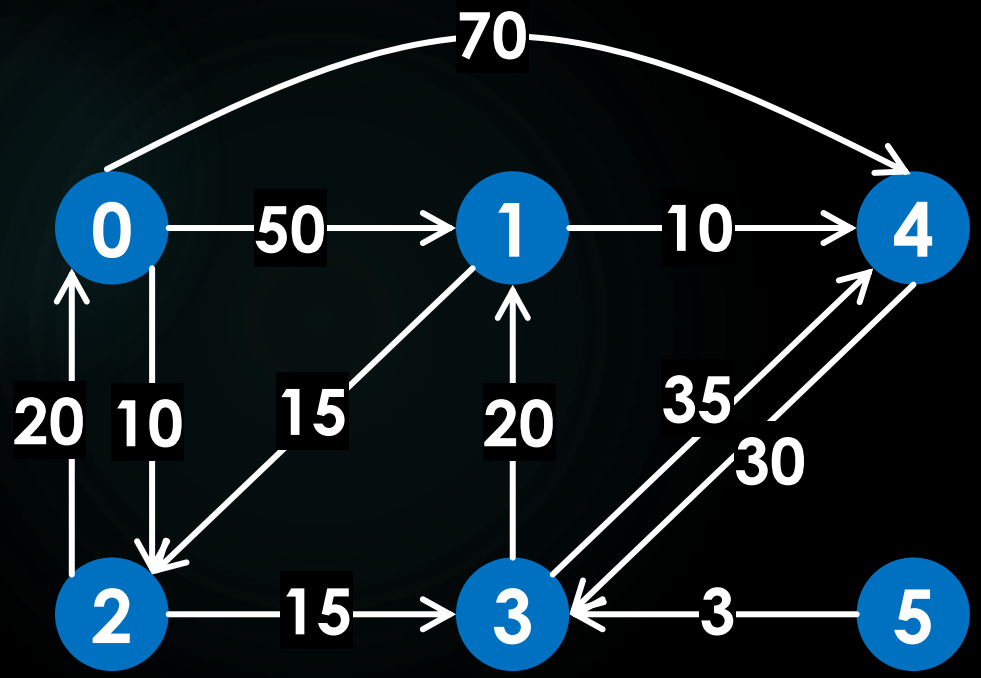
两个集合

- S={源点}：存放已求得最短路径的顶点的集合，初始包含源点
- T=V-S：尚未确定最短路径的顶点集合

S=V时算法停止

初始的当前最短路径： <0,0>

从当前最短路径出发，产生一个新的最短路径，将最短路径的终点从T移动到S中



源点	终点	路径	路径长度
0	1		
	2		
	3		
	4		
	5		

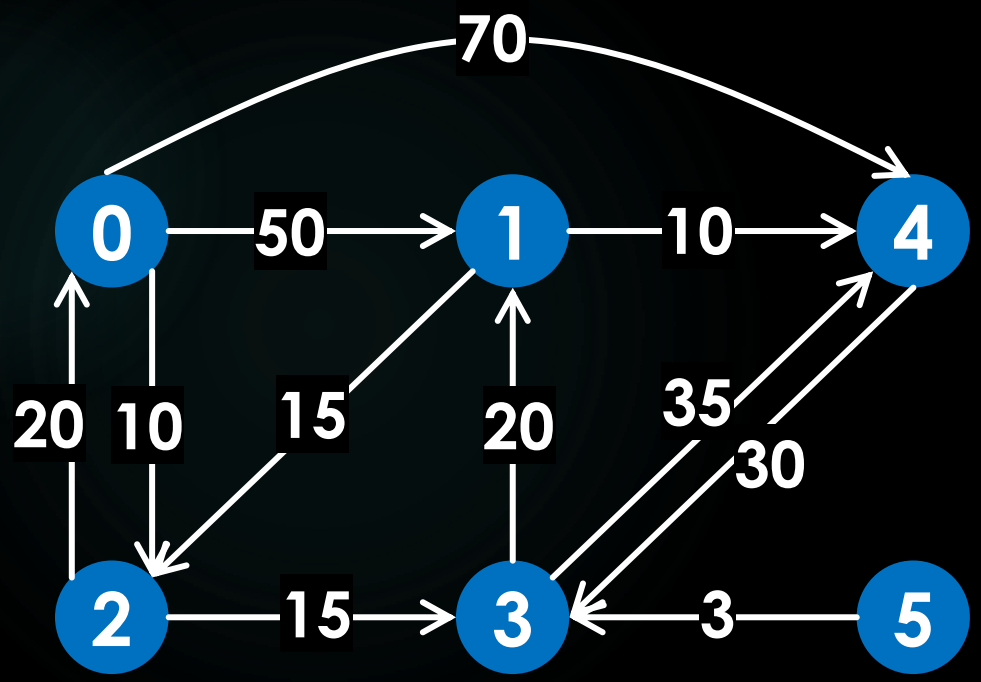
S={0} T={1,2,3,4,5}

迪杰斯特拉算法实现

从当前最短路径出发，如何产生一个新的最短路径？

从当前最短路径的终点出发，行走一个边，产生的新路径（不含回路），更新当前路径后，当前路径中长度最短的那条路径

S={0,2} T={1,3,4,5}



当前最短路径： <0,0>
从0出发再走一条边后，当前路径如下

源点	终点	路径	路径长度
0	1	0,1	50
	2	0,2	10
	3	-	∞
	4	0,4	70
	5	-	∞

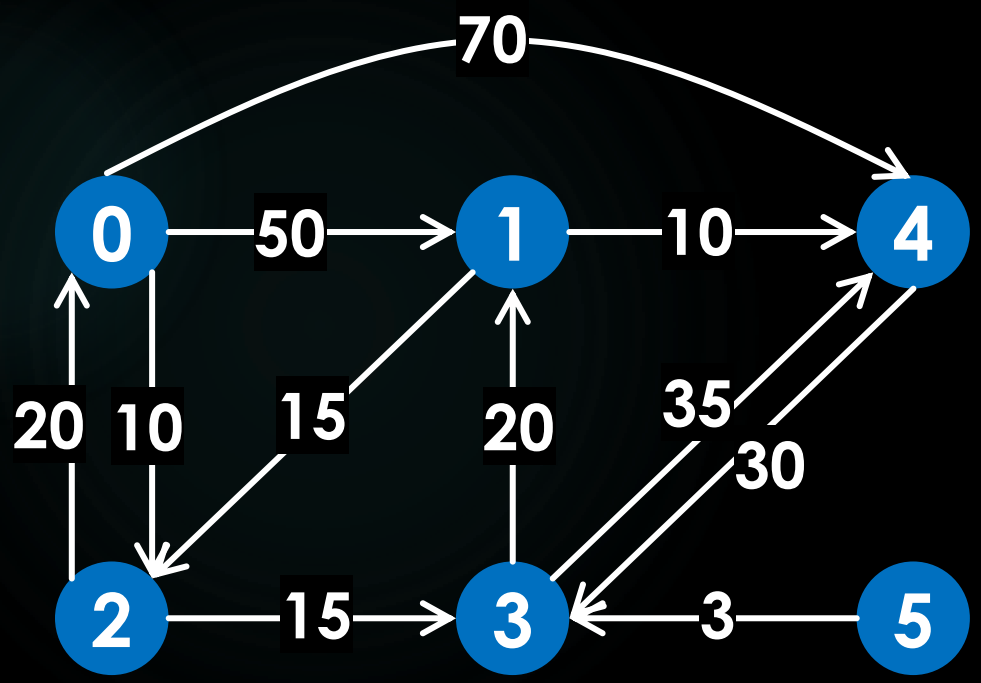
<0,2>最短，成为新的当前最短路径

迪杰斯特拉算法实现

从当前最短路径出发，如何产生一个新的最短路径？

从当前最短路径的终点出发，行走一个边，产生的新路径（不含回路），更新当前路径后，当前路径中长度最短的那条路径

S={0,2,3} T={1,4,5}



当前最短路径： <0,2>
从2出发再走一条边后，当前路径如下

源点	终点	路径	路径长度
0	1	0,1	50
	2	0,2	10
	3	0,2,3	25
	4	0,4	70
	5	-	∞

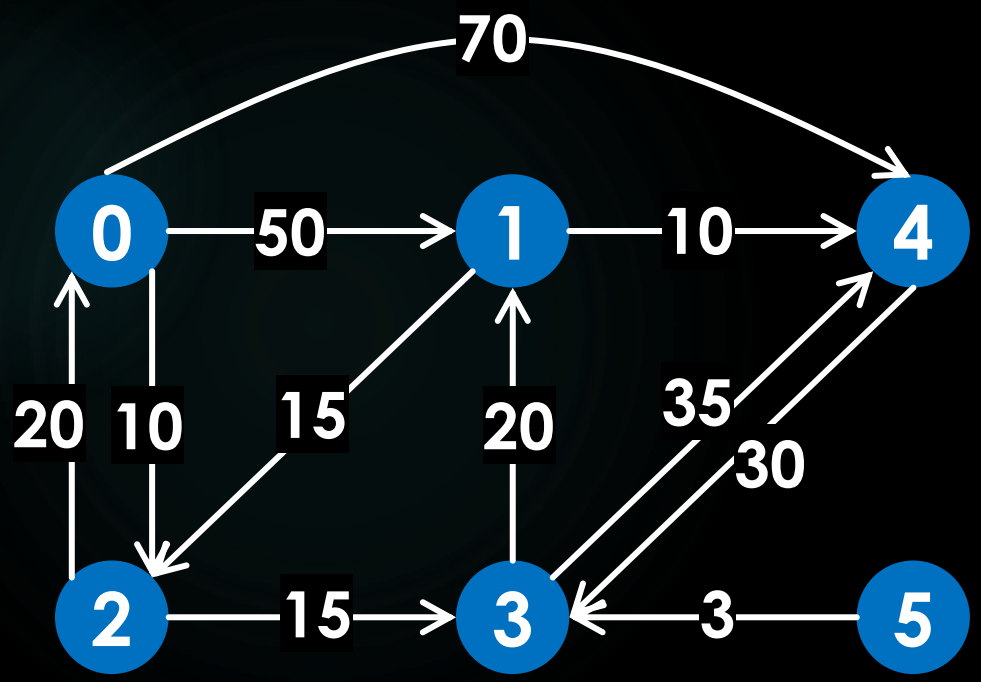
<0,2,3>最短，成为新的当前最短路径

迪杰斯特拉算法实现

从当前最短路径出发，如何产生一个新的最短路径？

从当前最短路径的终点出发，行走一个边，产生的新路径（不含回路），更新当前路径后，当前路径中长度最短的那条路径

S={0,2,3,1} T={4,5}



当前最短路径： <0,2,3>
从3出发再走一条边后，当前路径如下

源点	终点	路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,4	60
	5	-	∞

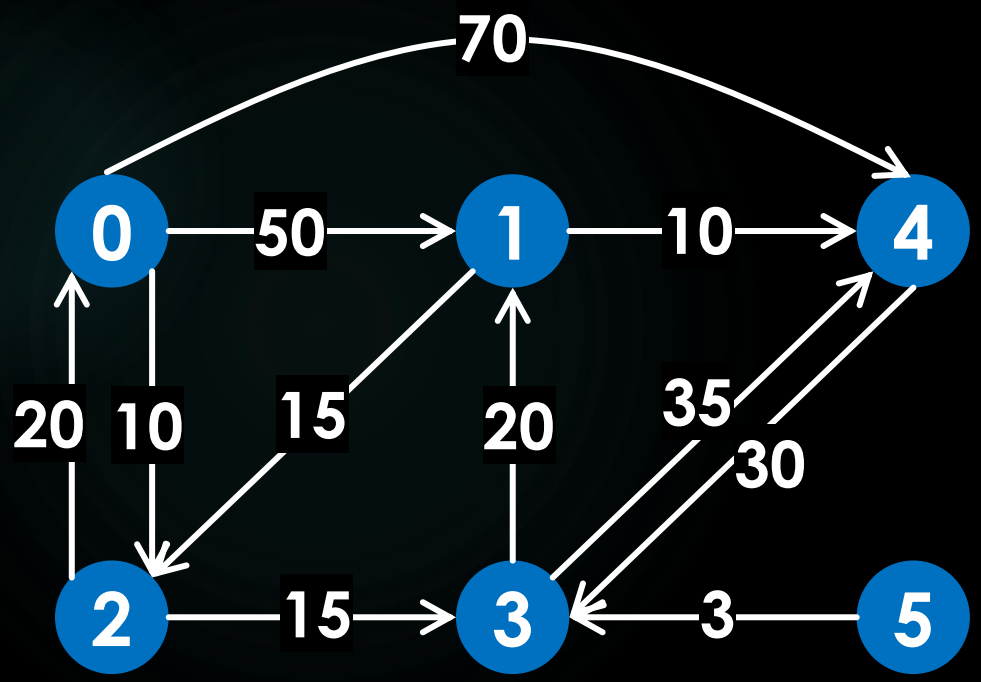
<0,2,3,1>最短，成为新的当前最短路径

迪杰斯特拉算法实现

从当前最短路径出发，如何产生一个新的最短路径？

从当前最短路径的终点出发，行走一个边，产生的新路径（不含回路），更新当前路径后，当前路径中长度最短的那条路径

S={0,2,3,1,4} T={5}



当前最短路径： <0,2,3,1>
从1出发再走一条边后，当前路径如下

源点	终点	路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

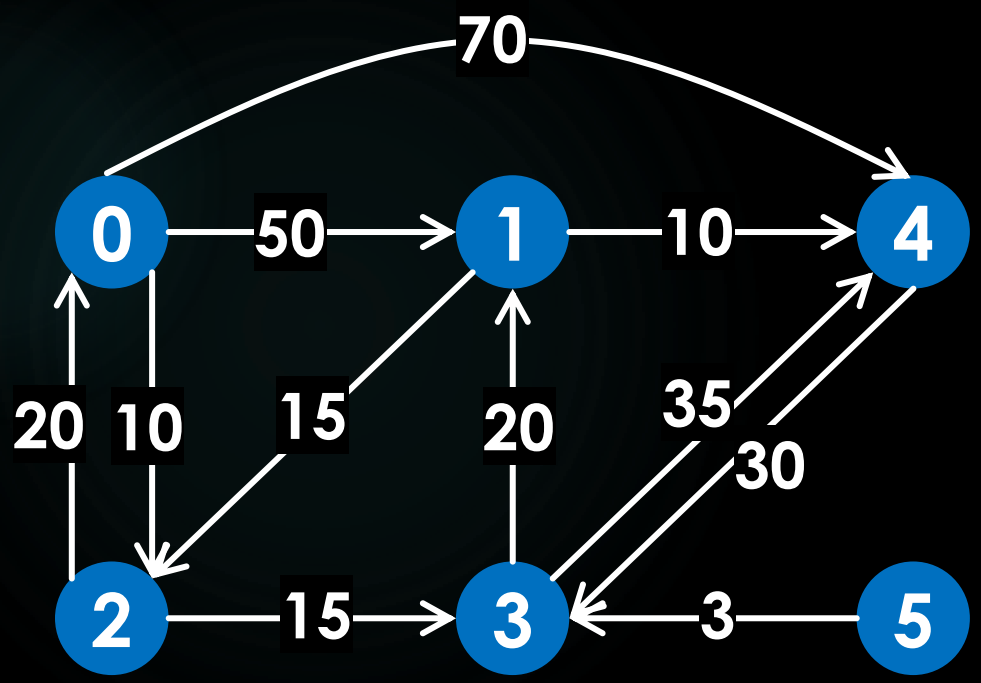
<0,2,3,1,4>最短，成为新的当前最短路径

迪杰斯特拉算法实现

从当前最短路径出发，如何产生一个新的最短路径？

从当前最短路径的终点出发，行走一个边，产生的新路径（不含回路），更新当前路径后，当前路径中长度最短的那条路径

S={0,2,3,1,4,5} T={}



当前最短路径： <0,2,3,1,4>
从4出发再走一条边后，当前路径如下

源点	终点	路径	路径长度
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,1,4	55
	5	-	∞

<0,5>最短，成为新的当前最短路径

把V分成两组：

（1）S：存放已求得最短路径的顶点的集合

（2）V-S=T：尚未确定最短路径的顶点集合

迪杰斯特拉算法的具体步骤：

- 初始状态时，集合S中只有一个源点，设为顶点v_0。首先产生从源点v_0到它自身的路径，其长度为0，将v_0加入S。
- 算法的每一步上，按照最短路径值的非递减次序，产生下一条最短路径，并将该路径的终点$t \in V-S$加入S。
- 直到S＝V时算法结束。

Dijkstra算法用到的数据结构

(1) 图用邻接矩阵；

(2)一维布尔数组s：若s[i]为true，表示顶点i在S中，否则表示i在T中

(3) 一维数组d：保存计算过程中各条最短路径的长度

S={0,2,3} T={1,4,5}

当前最短路径： <0,2,3>
从3出发再走一条边后，
当前路径如右
d[2]和d[3]是真正的最短路径长度
d[1]和d[4]是“临时值”

源点	终点	路径	d[i]
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,4	60
	5	-	∞

Dijkstra算法用到的数据结构

(4) 一维数组path：指示该条最短路径。

path[i]给出：从v_0到v_i的最短路径上，顶点v_i前面的顶点。

S={0,2,3} T={1,4,5}

当前最短路径： <0,2,3>
从3出发再走一条边后，
当前路径如右
path[1]=3 临时最短
path[2]=0 真的最短
path[3]=2 真的最短
path[4]=3 临时最短

源点	终点	路径	d[i]
0	1	0,2,3,1	45
	2	0,2	10
	3	0,2,3	25
	4	0,2,3,4	60
	5	-	∞

可以通过path数组推算出每条最短路径

迪杰斯特拉算法C实现

（1）初始化：$s[v_0]$=true

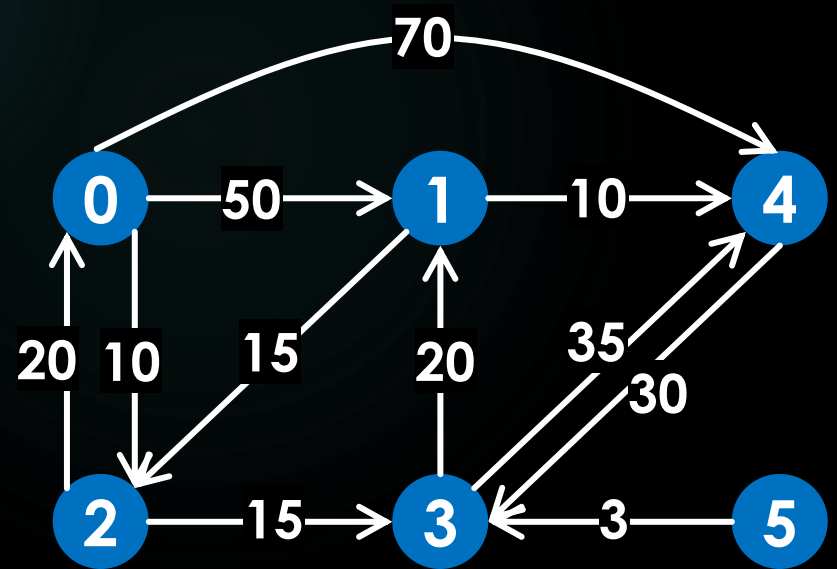
（2）第一次计算d[i]

$$d[i]=\begin{cases} w(v_0,i) & 若 <v_0,i>\in E \\ \infty & 若 <v_0,i>\notin E \end{cases}$$

（3）第一次计算path[i]:

若$<v_0,v_i>\in E$，则 path[i]=0，

否则 path[i]=-1



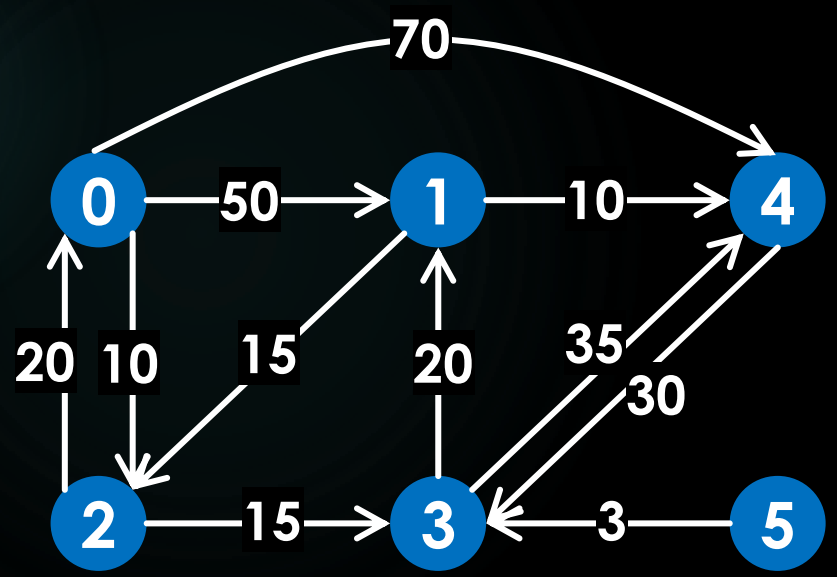
顶点	path	d	s
0	-1	∞	true
1	0	50	false
2	0	10	false
3	-1	∞	false
4	0	70	false
5	-1	∞	false

迪杰斯特拉算法C++实现

（4）求第一条最短路径为 <0,2>

$$d[k]=\min\{\ d[i]\,|\,i\in V-\{v_0\}\ \}$$

s[i]==false



顶点	path	d	s
0	-1	∞	true
1	0	50	false
2	0	10	false
3	-1	∞	false
4	0	70	false
5	-1	∞	false

迪杰斯特拉算法C++实现

（5）将第一条最短路径的终点k加入S　s[k]==true

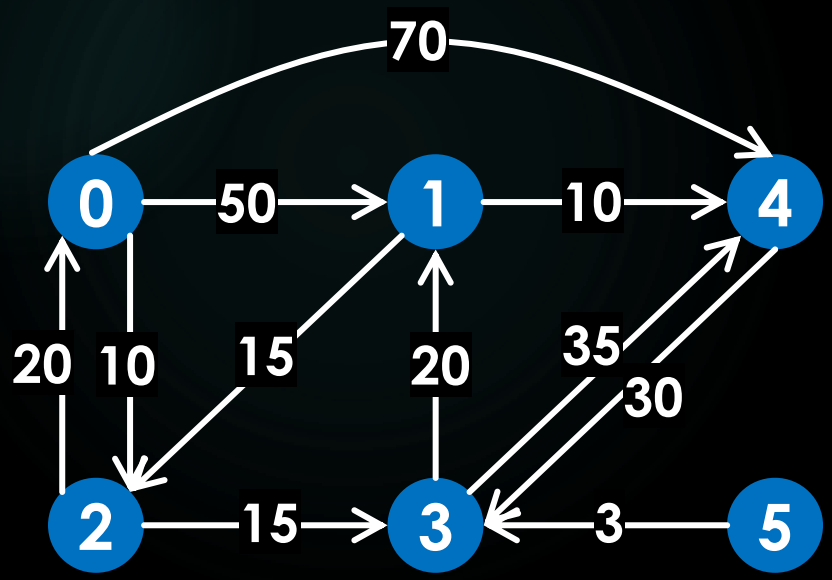
（6）对s[i]==false的顶点修正d值

$$d[i] = \min\{d[i], d[k] + w(k,i)\}$$

例如：d[1] = min{d[1],d[2]+w(2,1)}=25

（7）对于d发生改变的顶点i，令

path[i] = k



顶点	path	d	s
0	-1	∞	true
1	2	25	false
2	0	10	true
3	-1	∞	false
4	0	70	false
5	-1	∞	false

（8）求下一条最短路径，重复上述过程，直到s中所有值都false

■上述算法的执行时间为$O(n^2)$。如果人们只希望求从源点到某一个特定顶点之间的最短路径，也需要与求单源最短路径相同的时间复杂度$O(n^2)$

最短路径算法

- **单源最短路径的迪杰斯特拉（Djikstra）算法**

  给定一个顶点，求其与剩下其他顶点之间的最短路径

- **求所有顶点之间的最短路径的弗洛伊德（ Floyd）算法**

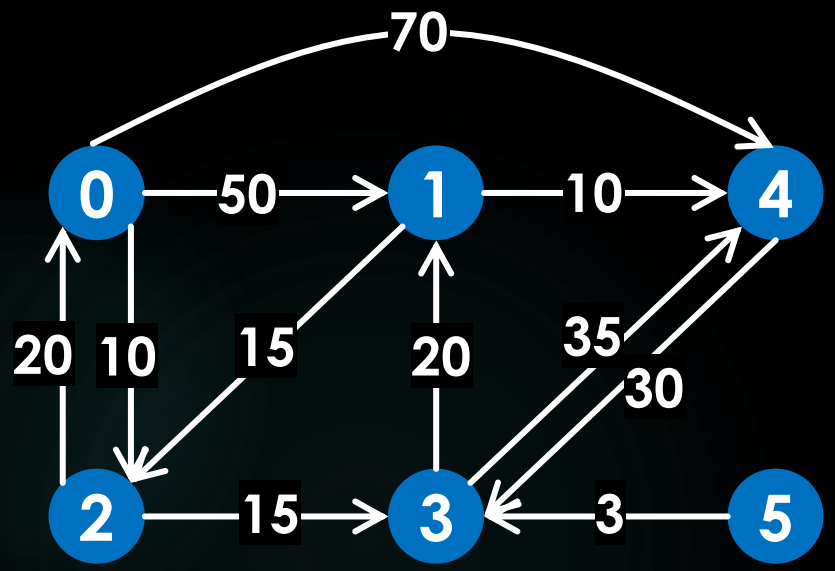
  求解所有两两顶点之间的最短路径

所有顶点之间最短路径

求任意两对顶点之间的最短路径，可以每次选择一个顶点为源点，重复执行迪杰斯特拉算法n次，便可以求得任意两对顶点之间的最短路径，总执行时间为$O(n^3)$

在形式上更直接些的弗洛伊德算法，时间复杂度也是$O(n^3)$
算法非常简洁优雅

弗洛伊德算法

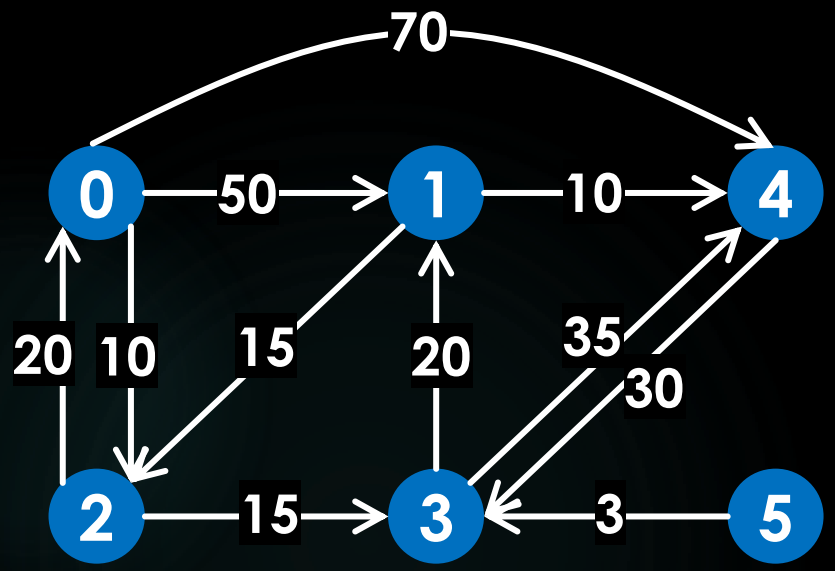


集合S={ }

d[i][j]：从i到j中间只经过S中的顶点的所有可能的路径中的最短路径的长度

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	∞	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

弗洛伊德算法



集合S={ }

p[i][j]：从i到j的最短路径上排在j之前的顶点

如果<4,5,6,7>是4到7的最短路径，则p[4][7]=6

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	-1	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

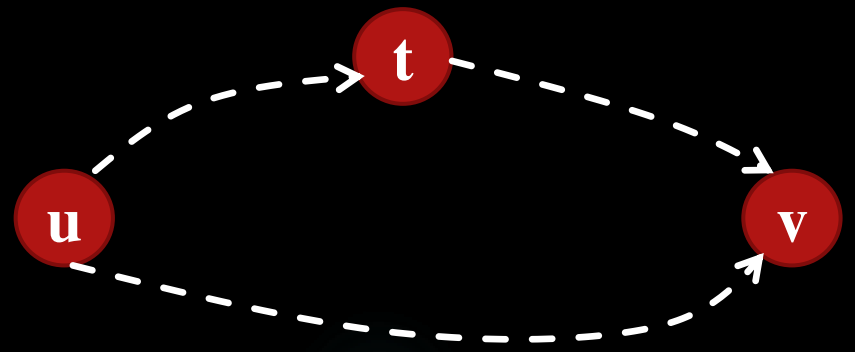
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	-1	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	∞	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



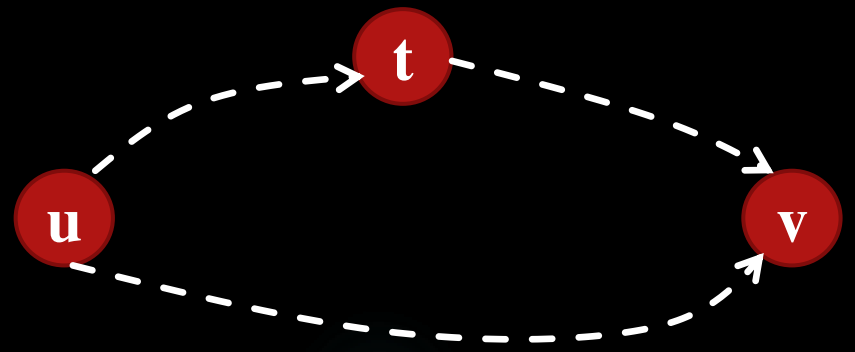
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	-1	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	∞	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



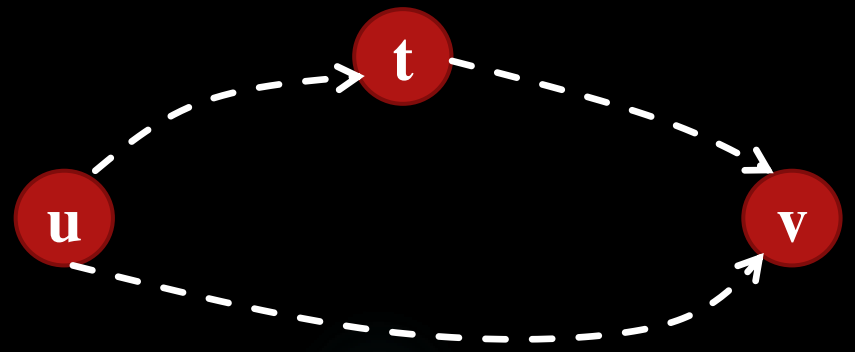
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	-1	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	∞	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



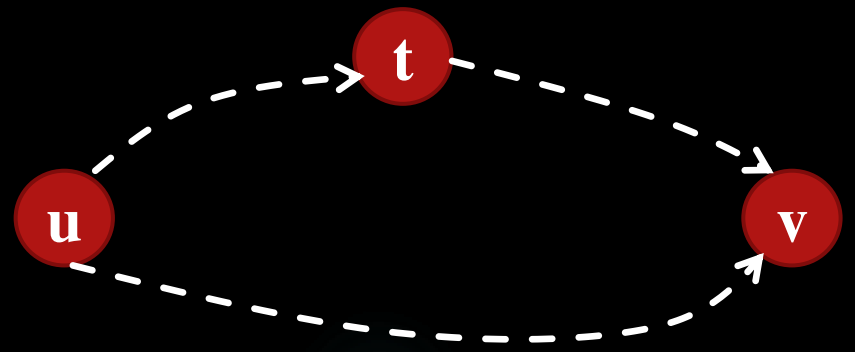
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	-1	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	∞	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}

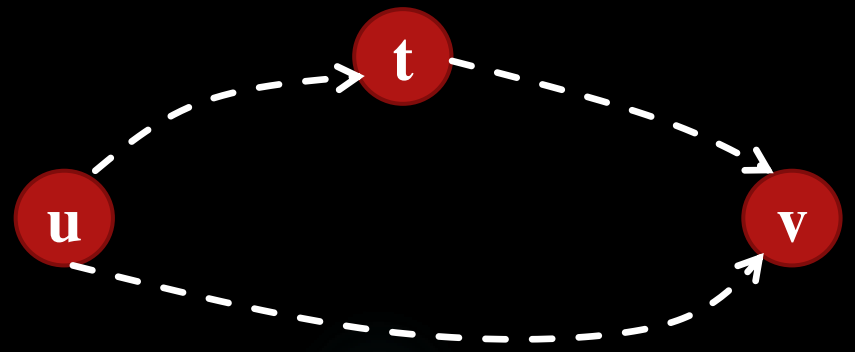


弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，造成d[u][v]改变，
则将p[u][v]=p[t][v]



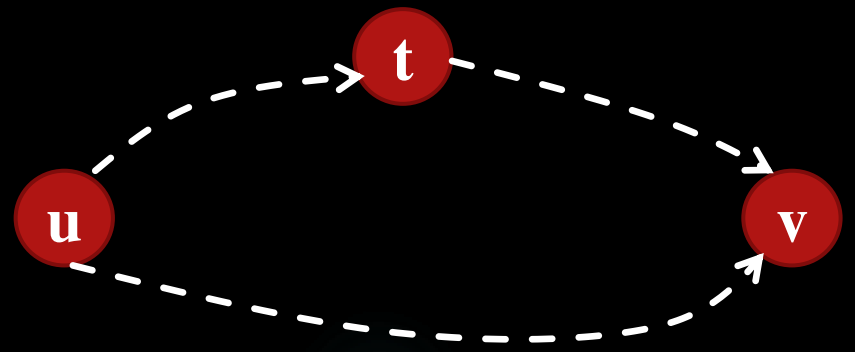
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	-1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	∞	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



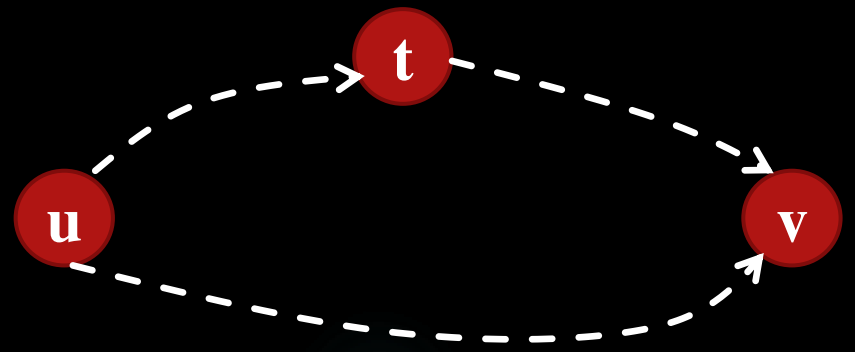
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



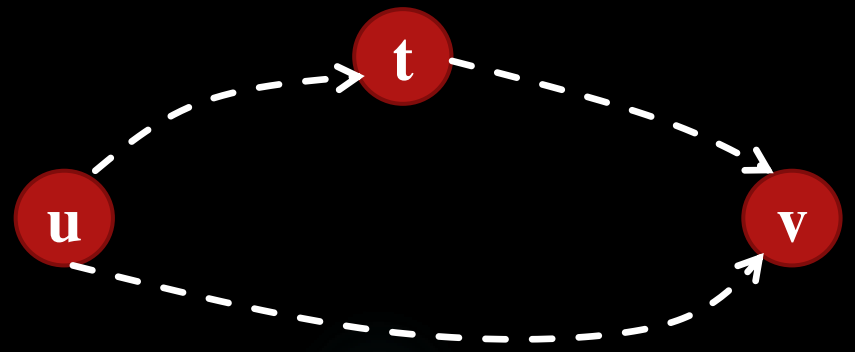
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



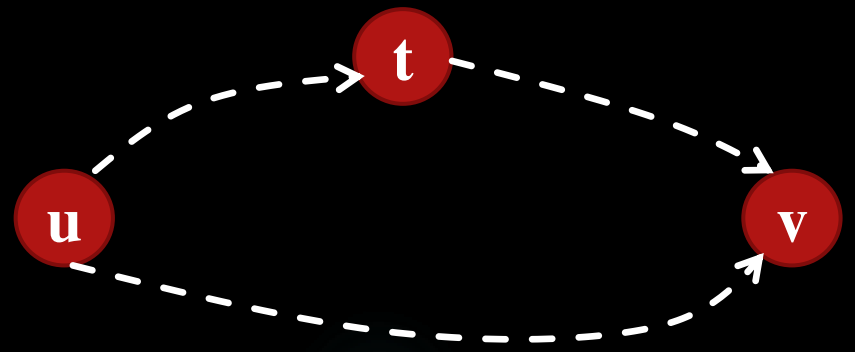
弗洛伊德算法 将0放入集合，集合S={0}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为t，则

d[u][v]=min{d[u][v],d[u][t]+d[t][v]}



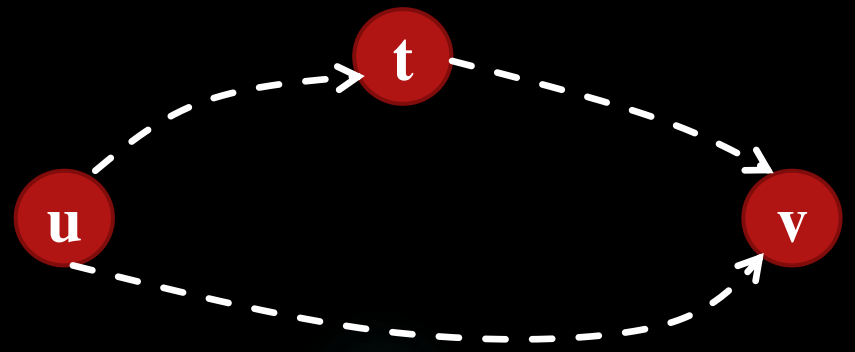
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



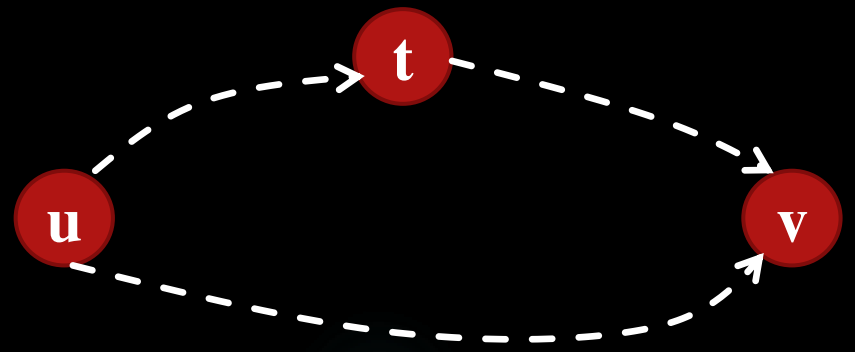
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	0	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	70	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



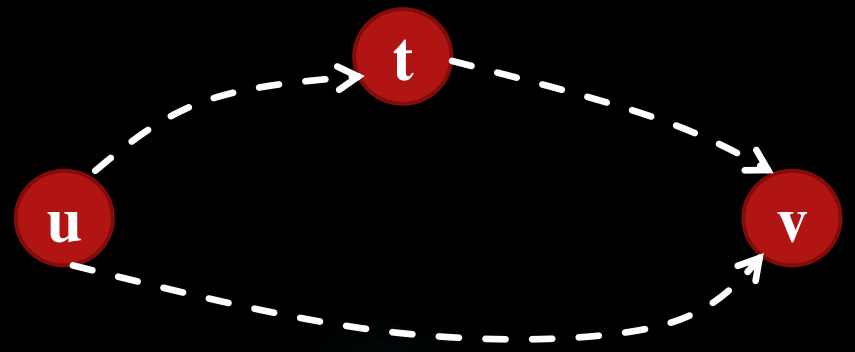
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



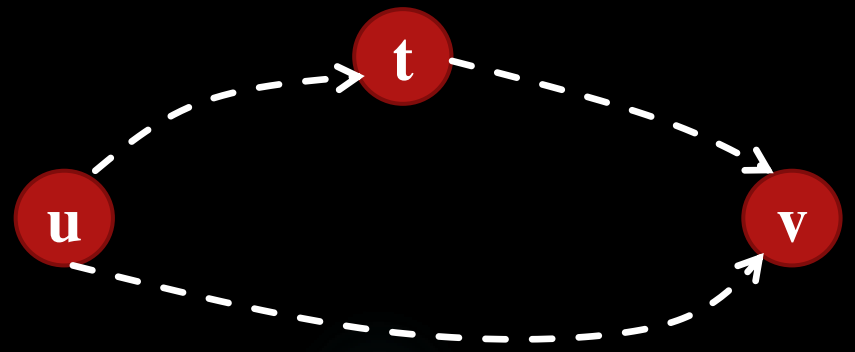
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



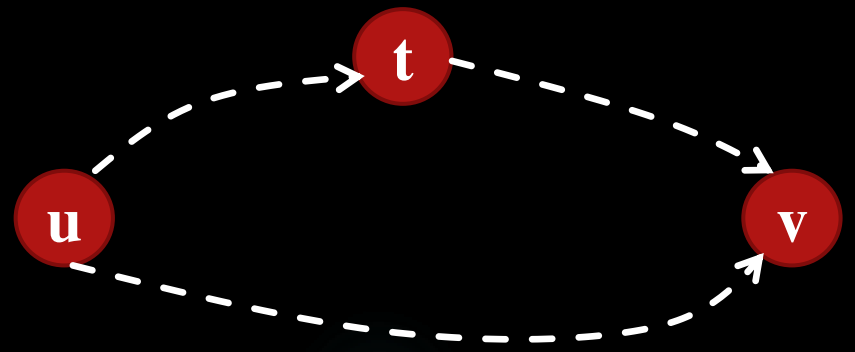
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



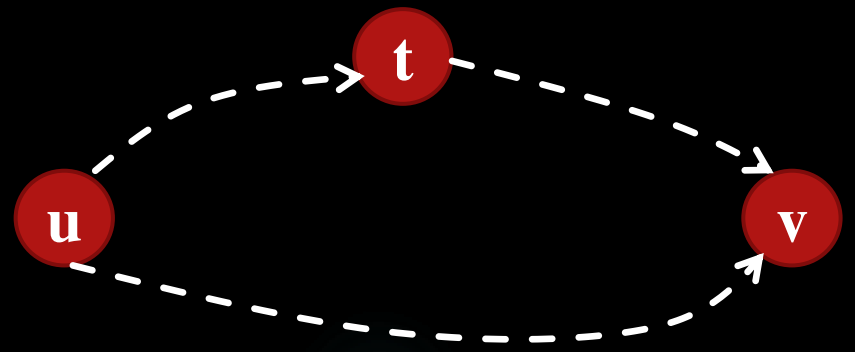
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	0	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	90	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



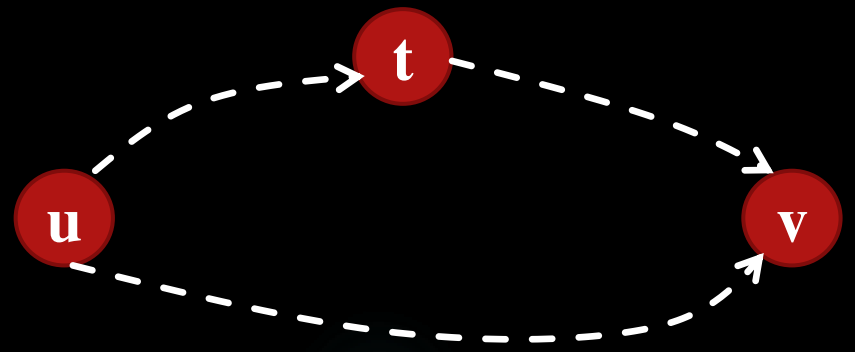
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}

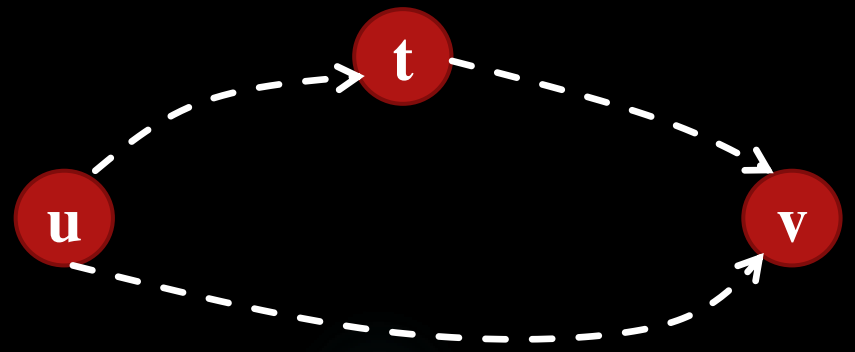


弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则
d[u][v]=min{d[u][v],d[u][1]+d[1][v]}

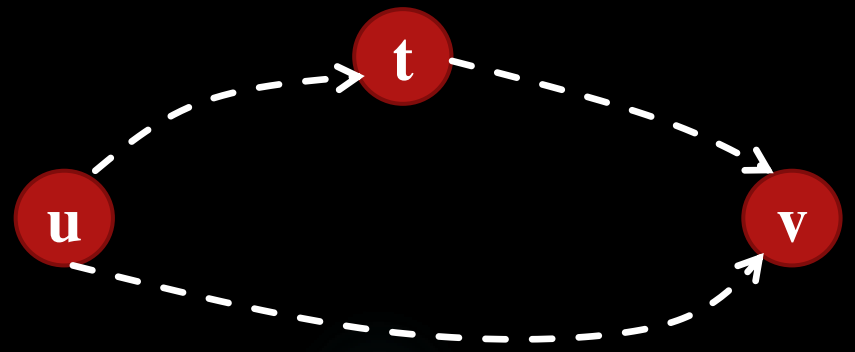


弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	-1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则
d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



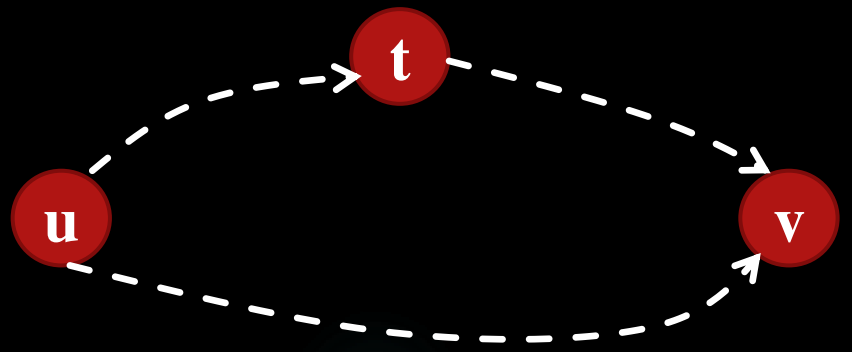
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	1	-1	3	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	35	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



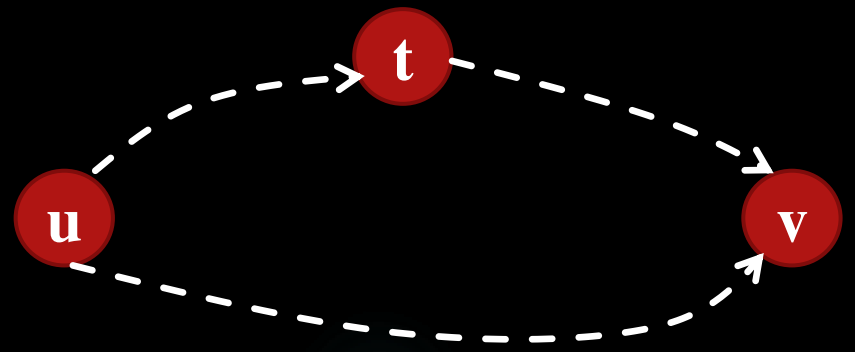
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	1	-1	1	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	35	0	30	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



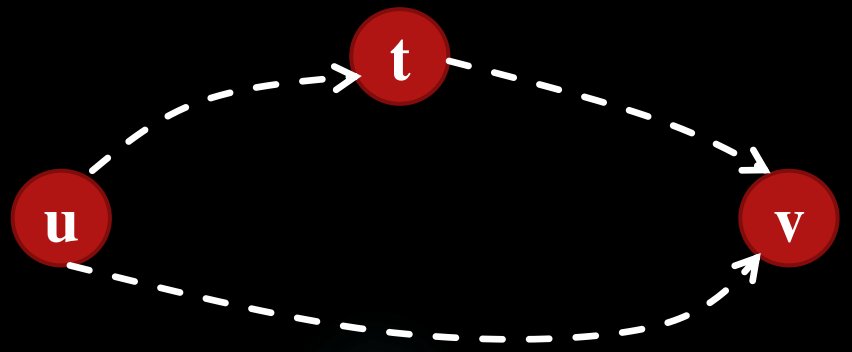
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	1	-1	1	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	35	0	30	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



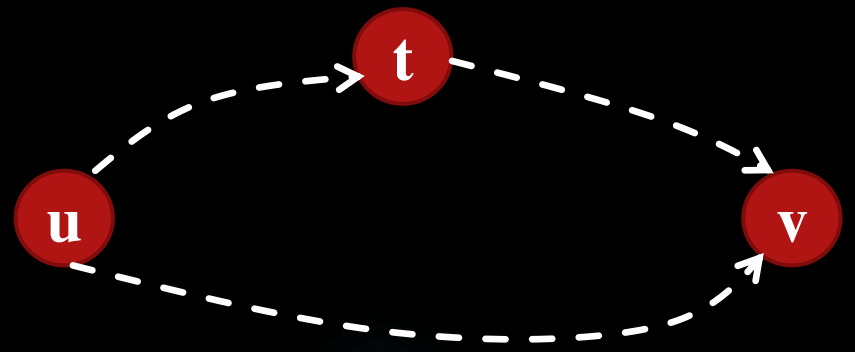
弗洛伊德算法 将1放入集合，集合S={0,1}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	-1	1	-1
1	-1	-1	1	-1	1	-1
2	2	0	-1	2	1	-1
3	-1	3	1	-1	1	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	∞	60	∞
1	∞	0	15	∞	10	∞
2	20	70	0	15	80	∞
3	∞	20	35	0	30	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为1，则

d[u][v]=min{d[u][v],d[u][1]+d[1][v]}



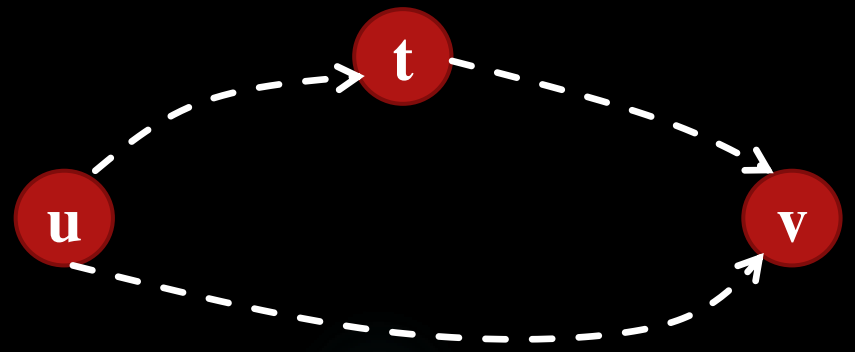
弗洛伊德算法 将2放入集合，集合S={0,1,2}，更新p和d

p	0	1	2	3	4	5
0	-1	0	0	2	1	-1
1	2	-1	1	2	1	-1
2	2	0	-1	2	1	-1
3	2	3	1	-1	1	-1
4	-1	-1	-1	4	-1	-1
5	-1	-1	-1	5	-1	-1

d	0	1	2	3	4	5
0	0	50	10	25	60	∞
1	35	0	15	30	10	∞
2	20	70	0	15	80	∞
3	55	20	35	0	30	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	3	∞	0

新放入S的顶点为2，则

d[u][v]=min{d[u][v],d[u][2]+d[2][v]}



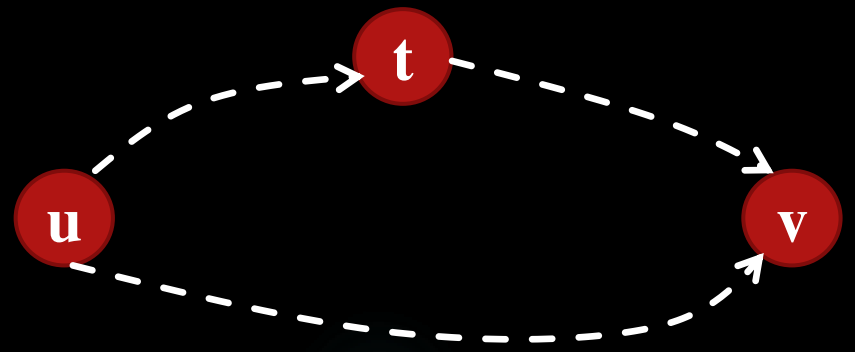
弗洛伊德算法 将3放入集合，集合S={0,1,2,3}，更新p和d

p	0	1	2	3	4	5
0	-1	3	0	2	1	-1
1	2	-1	1	2	1	-1
2	2	3	-1	2	1	-1
3	2	3	1	-1	1	-1
4	2	3	1	4	-1	-1
5	2	3	1	5	1	-1

d	0	1	2	3	4	5
0	0	45	10	25	55	∞
1	35	0	15	30	10	∞
2	20	35	0	15	45	∞
3	55	20	35	0	30	∞
4	85	50	65	30	0	∞
5	58	23	38	3	33	0

新放入S的顶点为3，则

d[u][v]=min{d[u][v],d[u][3]+d[3][v]}



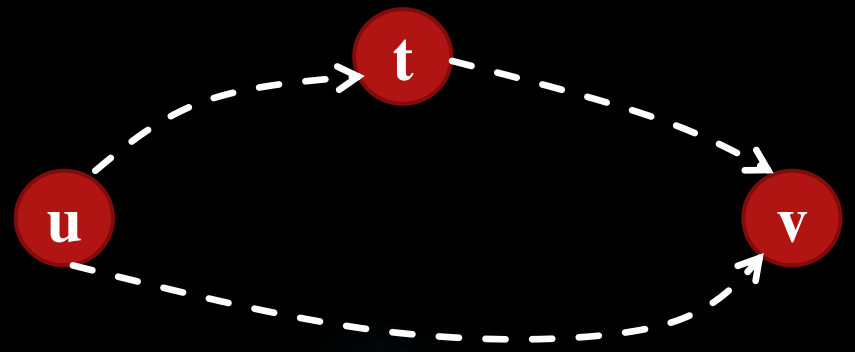
弗洛伊德算法 将4放入集合，集合S={0,1,2,3,4}，更新p和d

p	0	1	2	3	4	5
0	-1	3	0	2	1	-1
1	2	-1	1	2	1	-1
2	2	3	-1	2	1	-1
3	2	3	1	-1	1	-1
4	2	3	1	4	-1	-1
5	2	3	1	5	1	-1

d	0	1	2	3	4	5
0	0	45	10	25	55	∞
1	35	0	15	30	10	∞
2	20	35	0	15	45	∞
3	55	20	35	0	30	∞
4	85	50	65	30	0	∞
5	58	23	38	3	33	0

新放入S的顶点为4，则

d[u][v]=min{d[u][v],d[u][4]+d[4][v]}



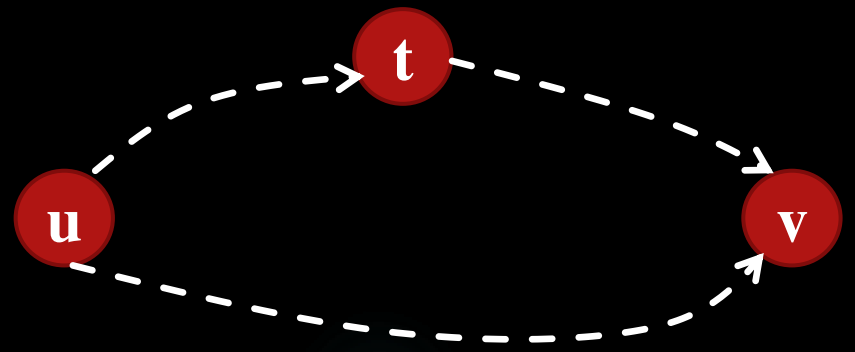
弗洛伊德算法 将5放入集合，集合S={0,1,2,3,4,5}，更新p和d

p	0	1	2	3	4	5
0	-1	3	0	2	1	-1
1	2	-1	1	2	1	-1
2	2	3	-1	2	1	-1
3	2	3	1	-1	1	-1
4	2	3	1	4	-1	-1
5	2	3	1	5	1	-1

d	0	1	2	3	4	5
0	0	45	10	25	55	∞
1	35	0	15	30	10	∞
2	20	35	0	15	45	∞
3	55	20	35	0	30	∞
4	85	50	65	30	0	∞
5	58	23	38	3	33	0

新放入S的顶点为5，则

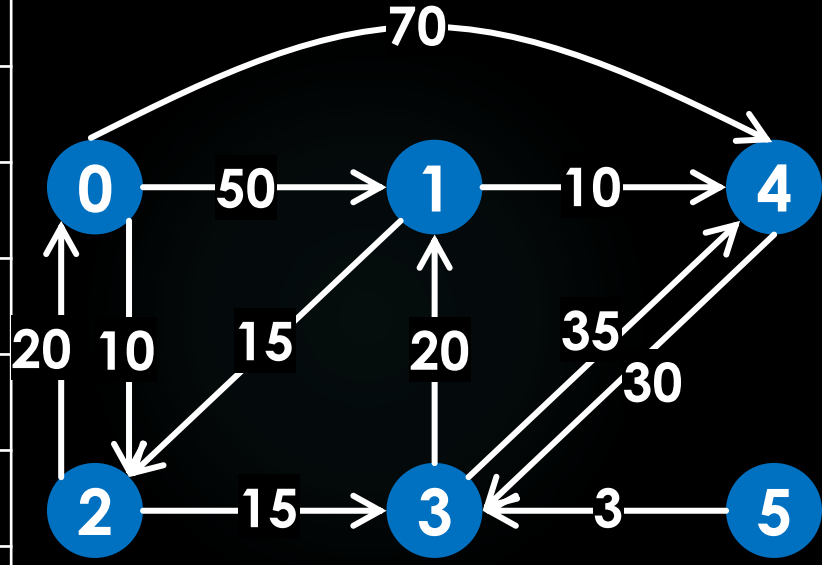
d[u][v]=min{d[u][v],d[u][5]+d[5][v]}



弗洛伊德算法 算法完成

p	0	1	2	3	4	5
0	-1	3	0	2	1	-1
1	2	-1	1	2	1	-1
2	2	3	-1	2	1	-1
3	2	3	1	-1	1	-1
4	2	3	1	4	-1	-1
5	2	3	1	5	1	-1

d	0	1	2	3	4	5
0	0	45	10	25	55	∞
1	35	0	15	30	10	∞
2	20	35	0	15	45	∞
3	55	20	35	0	30	∞
4	85	50	65	30	0	∞
5	58	23	38	3	33	0



验证！！！

弗洛伊德算法基本思想

- 设集合S的初始状态为空集合，依次向集合S中加入顶点0,1,…,n-1，每次加入一个顶点
- 二维数组d中，d[i][j]被定义为：从i到j中间只经过S中的顶点的所有可能的路径中的最短路径的长度
- 二维数组p中，p[i][j]给出从i到j的最短路径上顶点j前面的那个顶点
- 停止条件：随着S中顶点的不断增加， d[i][j]中值不断修正，当S=V时，d[i][j]的值就是i到j的最短路径。