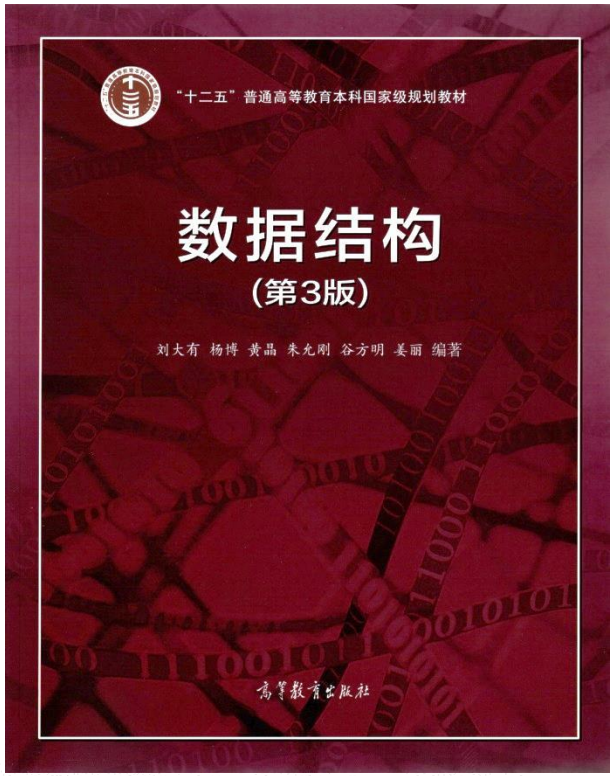
计算机学院王湘浩班

2024级



数据之法

结构之美

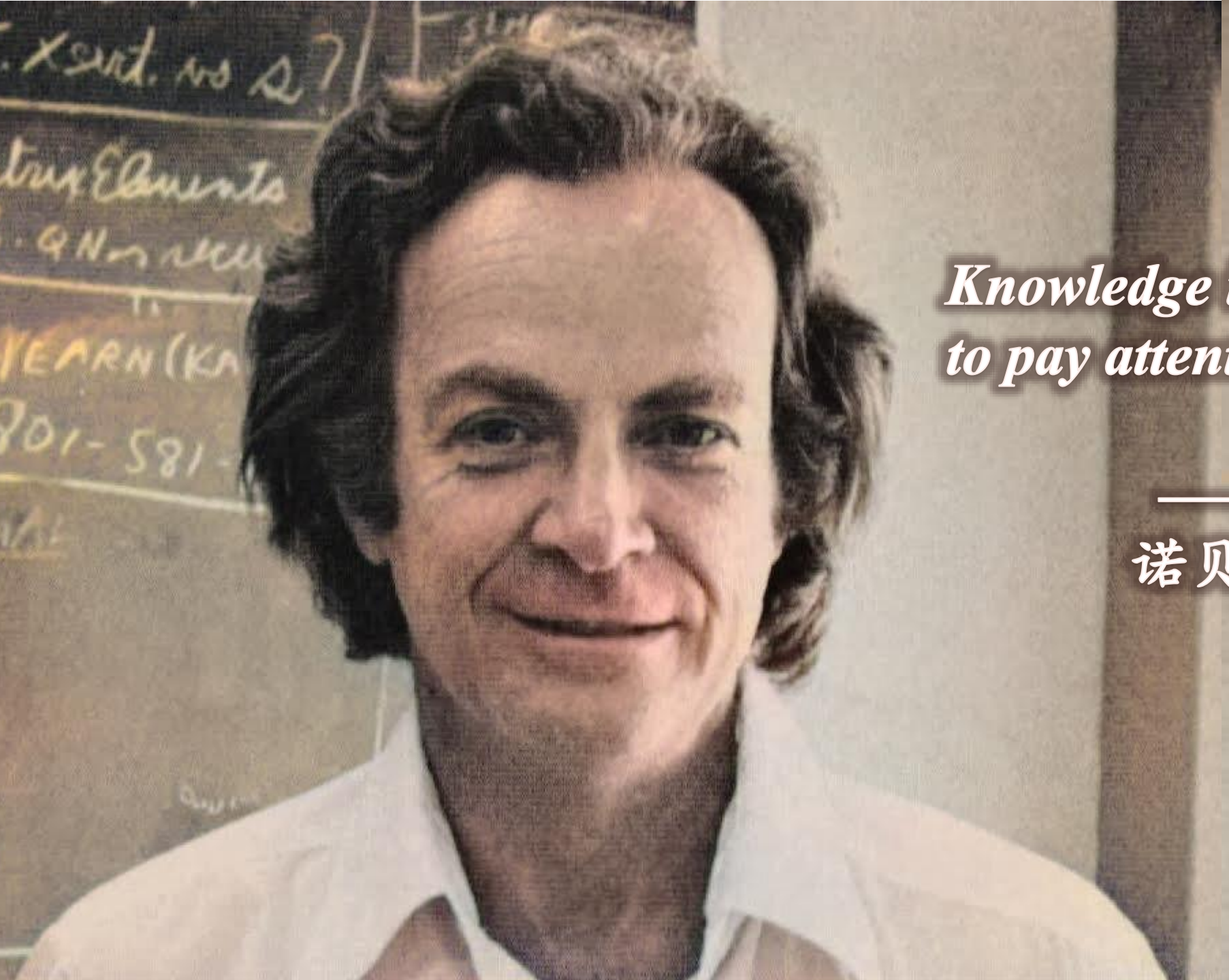
算法之道

树和二叉树的应用

- 压缩与哈夫曼树
- 表达式树
- 并查集

zhuyungang@jlu.edu.cn

Last updated on 2025.4



Knowledge is not free. You have to pay attention.

——*Richard Feynman*

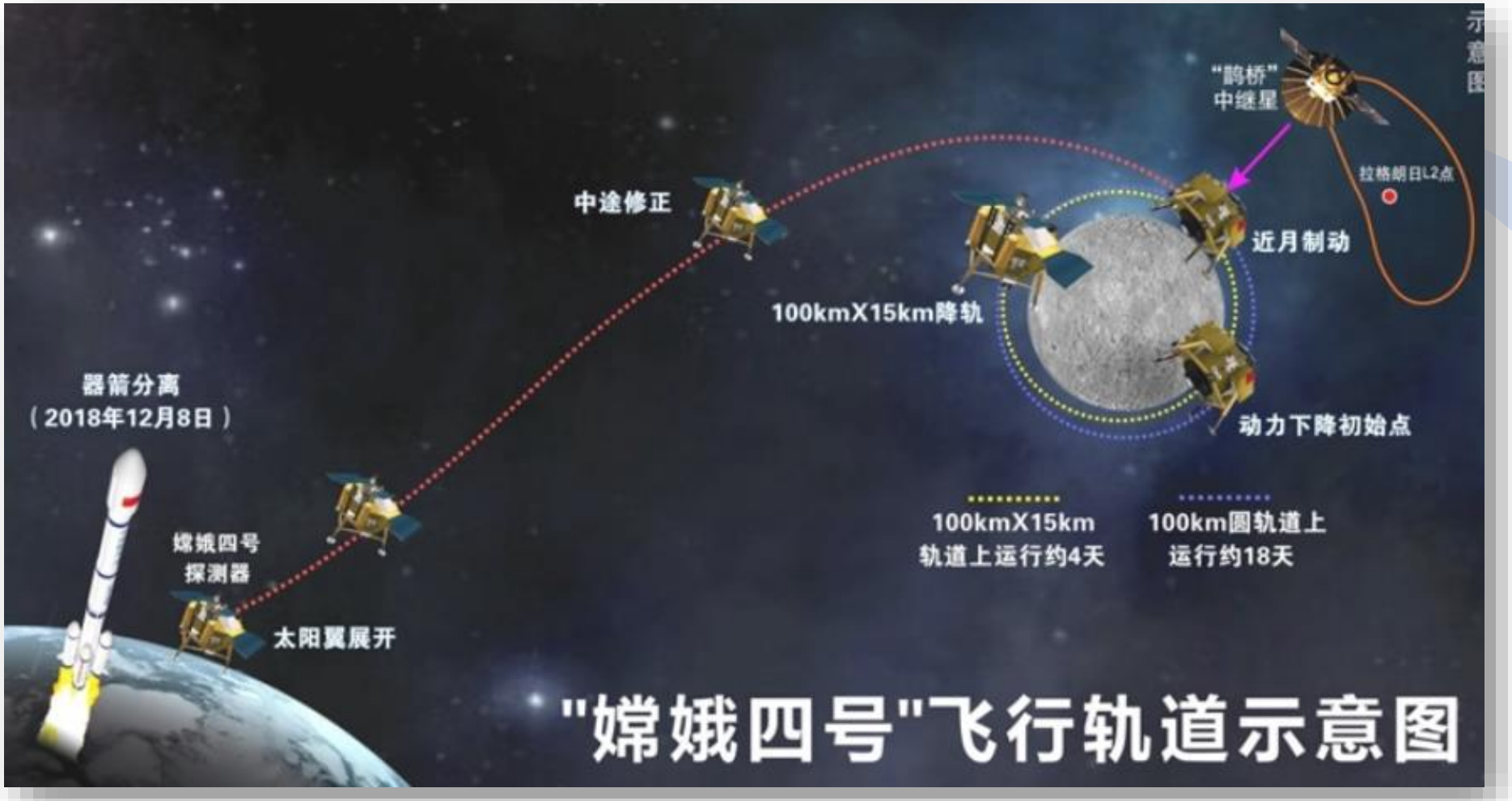
诺贝尔物理学奖获得者

美国科学院院士

加州理工学院教授

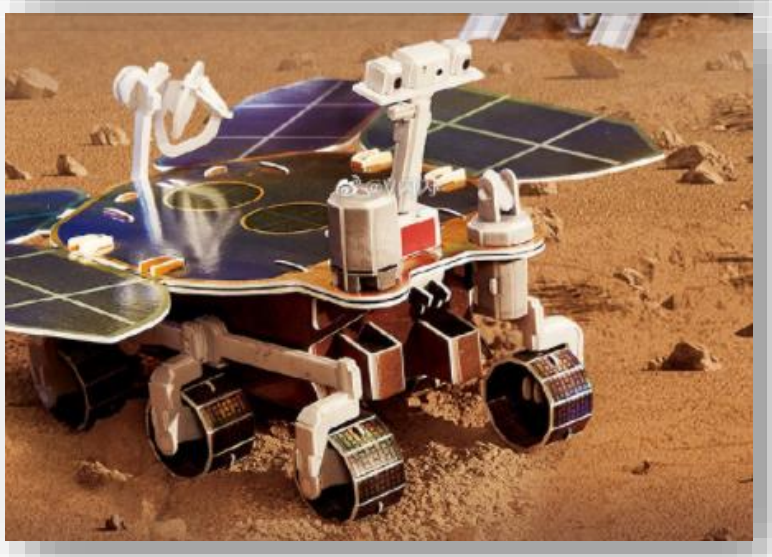
D

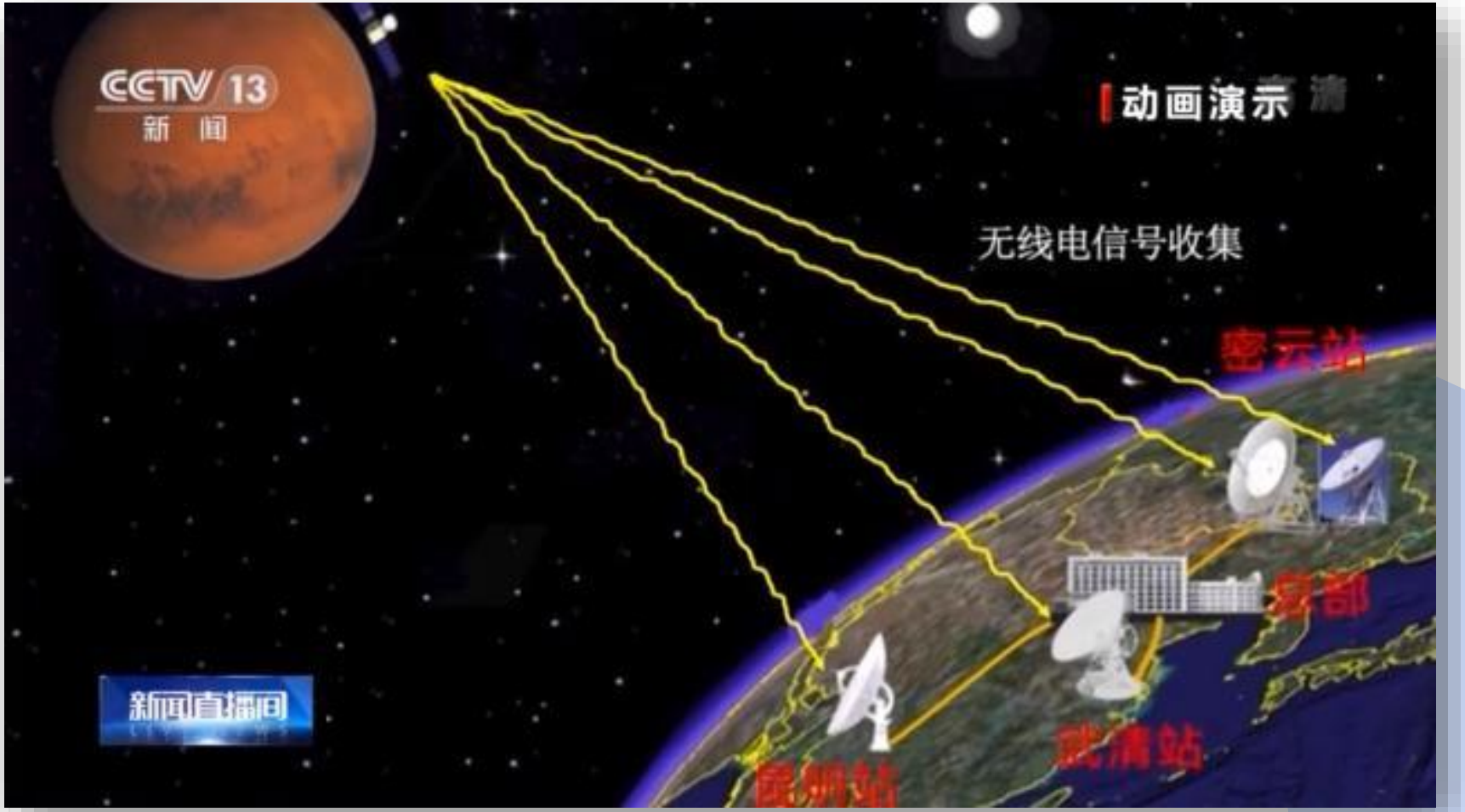
嫦娥月球探测器，与地球传输图像，需进行数据压缩



D

天问一号火星探测器与祝融号火星车，与地球传输图像，需进行数据压缩





A

数据压缩

- 数据压缩是计算机科学中的重要技术。
- 数据压缩过程称为编码，即将文件中的每个字符均转换为一个唯一的二进制位串。
- 数据解压过程称为解码，即将二进制位串转换为对应的字符。

信息编码

A

假设有一个文本文件仅包含4种字符：*A*、*B*、*C*、*D*，且文件中有11个*A*，4个*B*，3个*C*，2个*D*。

若采用等长编码，因为$\lceil \log_2 4 \rceil = 2$，所以每个字符都至少由一个2位的二进制数表示。于是文件所需存储空间（文件的总编码长度）为：

$$(11 + 4 + 3 + 2) \times 2 = 40 \text{ bit} = 5 \text{ Byte}$$

还有更好的方案么？

信息编码

A

假设有一个文本文件仅包含4种字符：*A*、*B*、*C*、*D*，且文件中有11个*A*，4个*B*，3个*C*，2个*D*。

若采用等长编码，因为$\lceil \log_2 4 \rceil = 2$，所以每个字符都至少由一个2位的二进制数表示。于是文件所需存储空间（文件的总编码长度）为：

$$(11 + 4 + 3 + 2) \times 2 = 40 \text{ bit} = 5 \text{ Byte}$$

字符出现的频率不同，可否借助这一信息，设计不等长编码，使文件总长度更短？

如何才能压缩总编码长度？

A

假设有一个文件中有100个*A*，1个*B*，1个*C*，1个*D*。

编码策略1：

A、*B*、*C*、*D*都用2个二进制位表示。

总编码长度：103×2 = 206 bit

编码策略2：

*A*用1个二进制位表示；*B*、*C*、*D*用5个二进制位表示。

总编码长度：100×1+3×5=115 bit

采用不等长编码，希望：

① 熵编码：文件中出现频率高的字符的编码长度尽可能短。

解码过程不能出现歧义性

A

字符	编码
A	10
B	01
C	1001

歧义：1001 = *C* 还是 *AB*?

原因：*A*的编码是 *C* 的前缀。

采用不等长编码，希望：

② 前缀码（无前缀冲突编码，Prefix-Free Codes）：字符集中任何字符的编码都不是其它字符的编码的前缀。

怎样的前缀码才能使文件的总编码长度最短？

A

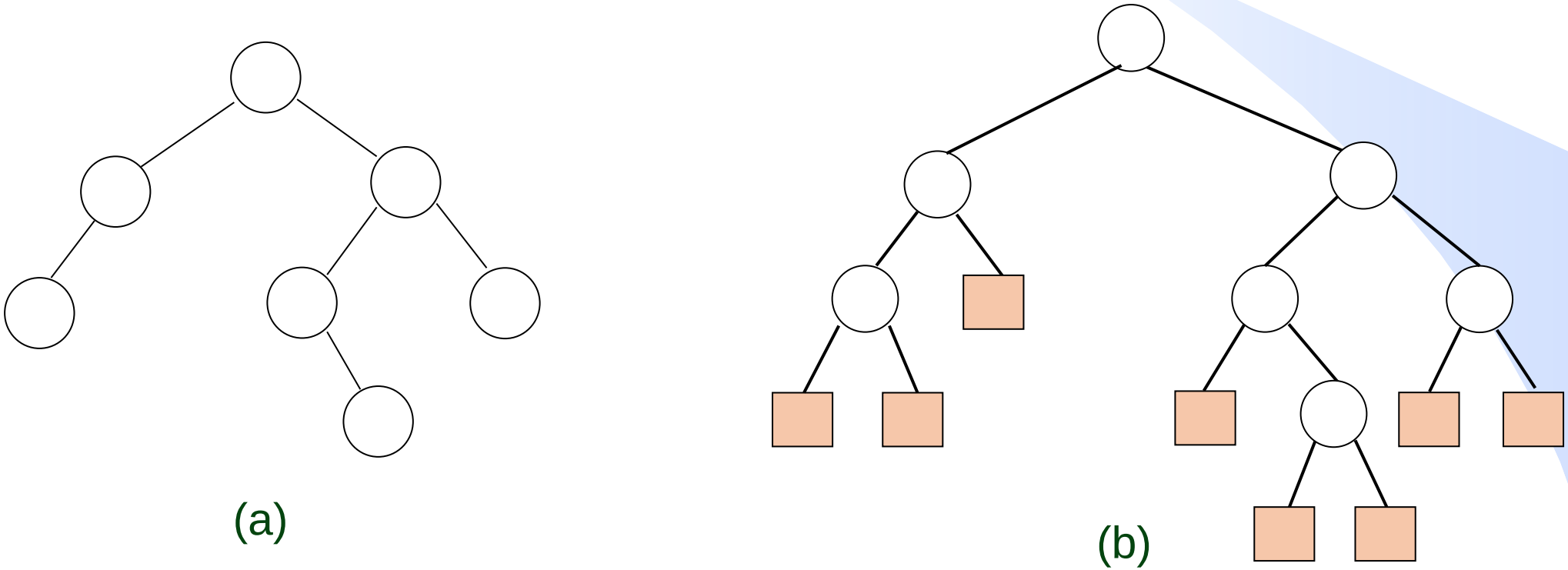
最优编码问题描述：

设组成文件的字符集$A=\{a_1,a_2,\ldots,a_n\}$，其中a_i出现的次数为c_i，a_i的编码长度为l_i。设计一个前缀码方案，使文件的总编码长度最小：

$$\min \sum_{i=1}^{n} c_i \cdot l_i$$

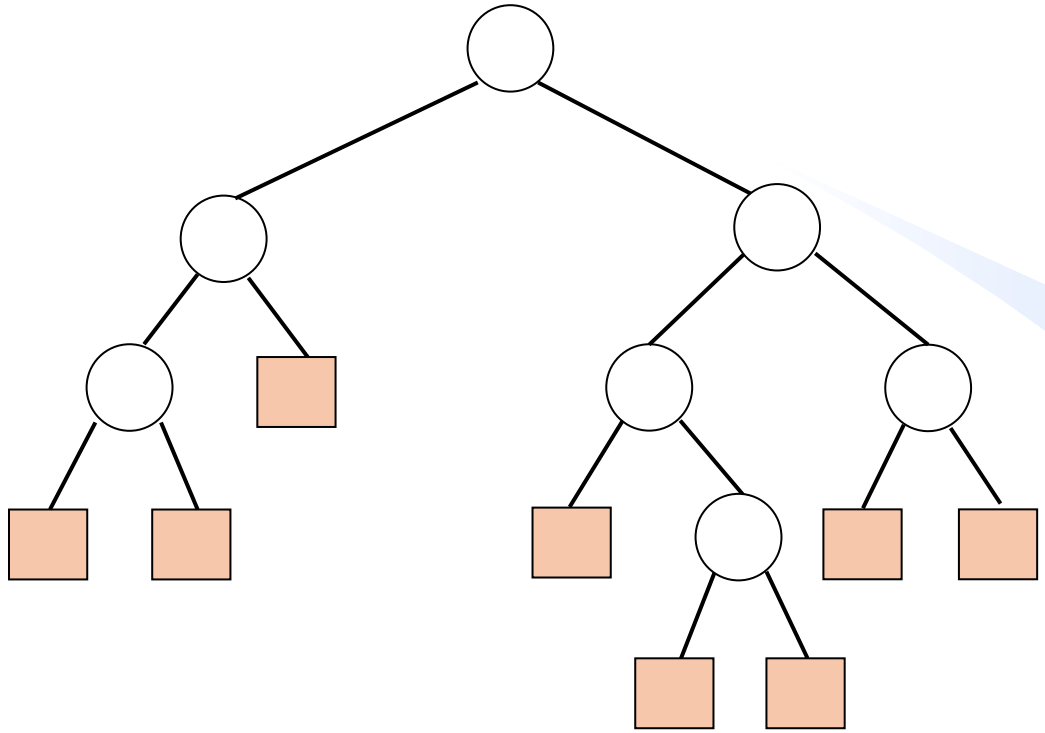
扩充二叉树

定义 在二叉树中**空指针**的位置，都增加特殊的结点（空叶结点），由此生成的二叉树称为**扩充二叉树**。



二叉树及其对应的扩充二叉树

扩充二叉树



- 称圆形结点为内结点，方形结点为外结点。
- 每个内结点都有2个孩子，每个外结点没有孩子。
- 规定空二叉树的扩充二叉树是只有一个外结点。

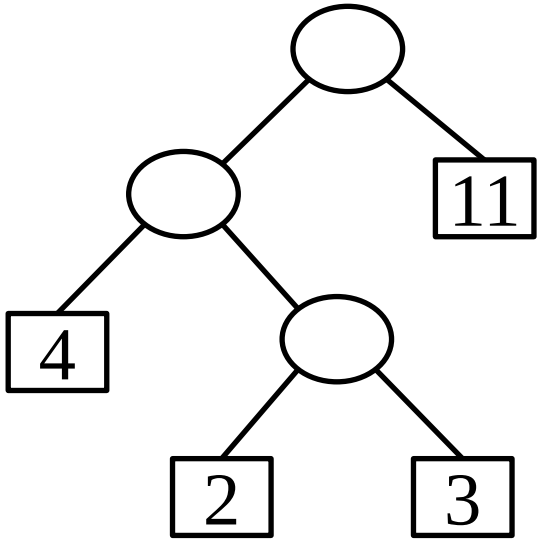
加权路径长度（*Weighted Path Length*, *WPL*）

A

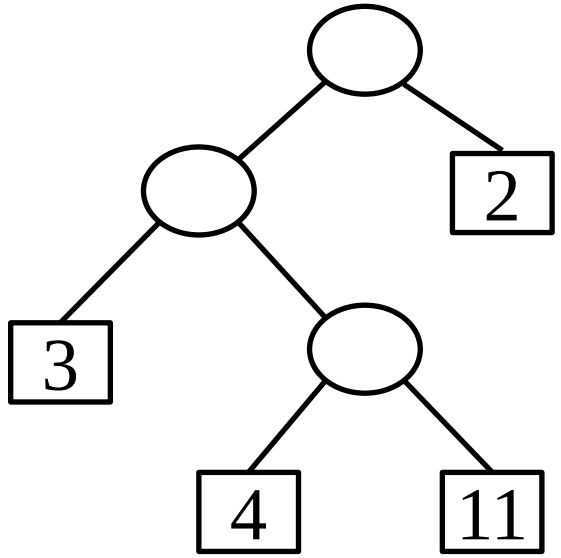
设扩充二叉树有n个外结点，为每个外结点赋予一个实数，称为该结点的权值，第i个外结点的权值为w_i，深度为L_i，则**加权路径长度**定义为：

$$WPL = \sum_{i=1}^{n} w_i L_i$$

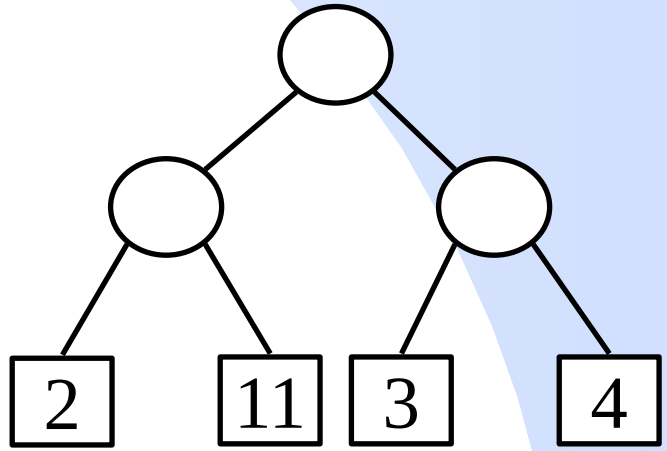
n个带权外结点构成的所有扩充二叉树中，WPL值最小者称为最优二叉树。



4×2+2×3+3×3+11×1=34



3×2+4×3+11×3+2×1=53

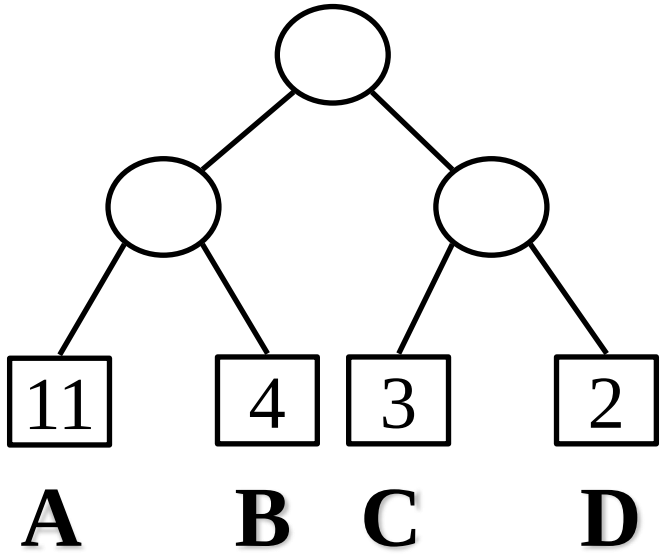


2×2+11×2+3×2+4×2=40

B

➢ 一种文件编码方案可以映射为一棵扩充二叉树。

文件编码		扩充二叉树
字符a_i	⟷	外结点i
字符的出现次数c_i	⟷	外结点的权值 w_i
字符的编码长度l_i	⟷	外结点的深度 L_i
文件总编码长度	⟷	扩充二叉树的 WPL值



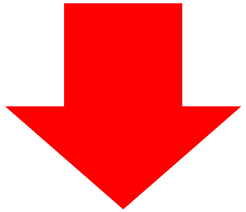
$$\text{文件编码长度}=\sum_{i=1}^{n} c_i \cdot l_i$$

$$WPL=\sum_{i=1}^{n} w_i \cdot L_i$$

最优编码问题

给定*n*种字符和每种字符出现的次数

构建一种总编码长度最短的编码方案（最优编码方案）

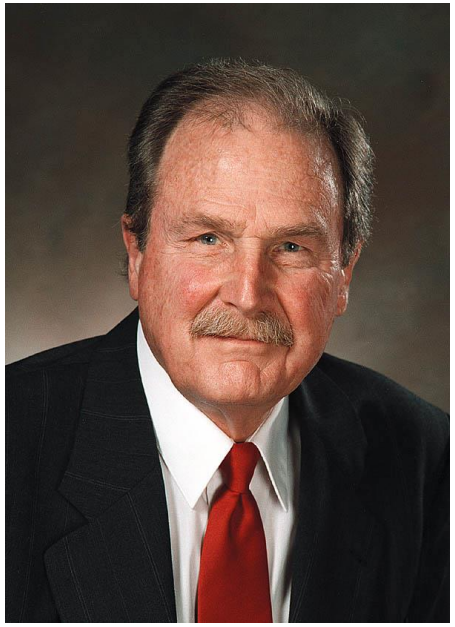


构造最优二叉树问题

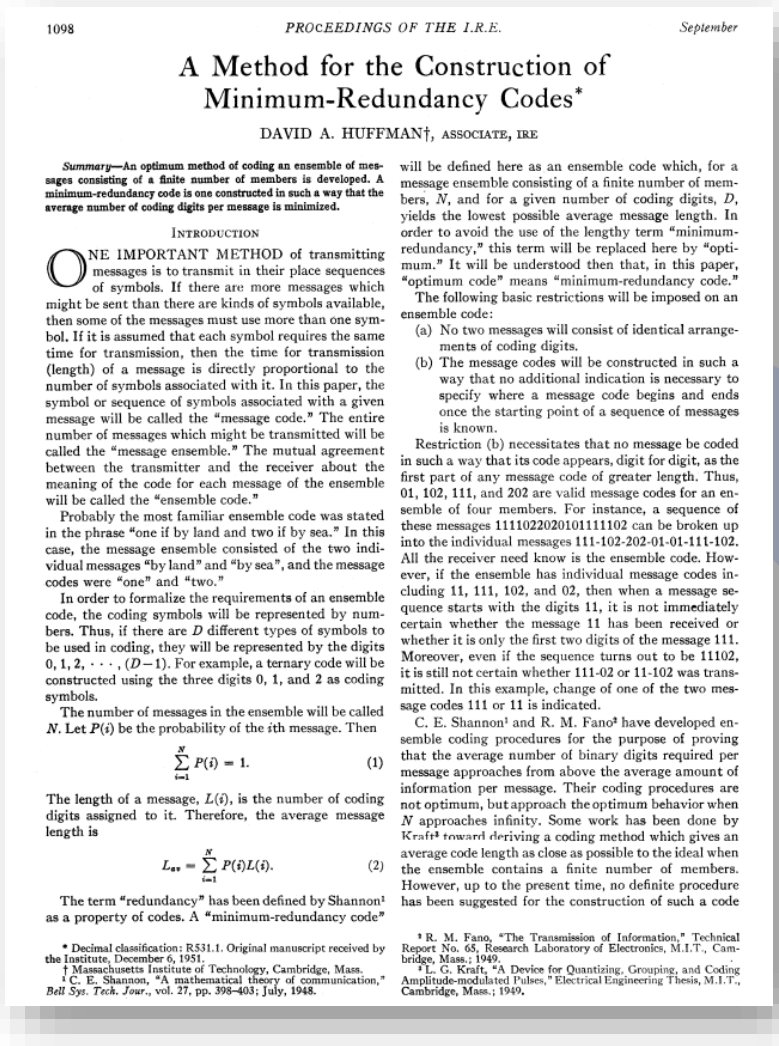
给定*n*个外结点和每个结点的权值

构建一棵WPL值最小的扩充二叉树（最优二叉树）

Huffman算法



David Huffman
（1925-1999）
麻省理工学院 博士
加州大学圣克鲁兹分校 教授



1098 PROCEEDINGS OF THE I.R.E. September

A Method for the Construction of Minimum-Redundancy Codes*

DAVID A. HUFFMAN†, ASSOCIATE, IRE

Summary—**An optimum method of coding an ensemble of messages consisting of a finite number of members is developed. A minimum-redundancy code is one constructed in such a way that the average number of coding digits per message is minimized.**

Introduction

ONE IMPORTANT METHOD of transmitting messages is to transmit in their place sequences of symbols. If there are more messages which might be sent than there are kinds of symbols available, then some of the messages must use more than one symbol. If it is assumed that each symbol requires the same time for transmission, then the time for transmission (length) of a message is directly proportional to the number of symbols associated with it. In this paper, the symbol or sequence of symbols associated with a given message will be called the "message code." The entire number of messages which might be transmitted will be called the "message ensemble." The mutual agreement between the transmitter and the receiver about the meaning of the code for each message of the ensemble will be called the "ensemble code."

Probably the most familiar ensemble code was stated in the phrase "one if by land and two if by sea." In this case, the message ensemble consisted of the two individual messages "by land" and "by sea", and the message codes were "one" and "two."

In order to formalize the requirements of an ensemble code, the coding symbols will be represented by numbers. Thus, if there are D different types of symbols to be used in coding, they will be represented by the digits $0, 1, 2, \cdots, (D-1)$. For example, a ternary code will be constructed using the three digits 0, 1, and 2 as coding symbols.

The number of messages in the ensemble will be called N. Let $P(i)$ be the probability of the ith message. Then

$$\sum_{i=1}^{N} P(i) = 1. \qquad (1)$$

The length of a message, $L(i)$, is the number of coding digits assigned to it. Therefore, the average message length is

$$L_{av} = \sum_{i=1}^{N} P(i)L(i). \qquad (2)$$

The term "redundancy" has been defined by Shannon[1] as a property of codes. A "minimum-redundancy code" will be defined here as an ensemble code which, for a message ensemble consisting of a finite number of members, N, and for a given number of coding digits, D, yields the lowest possible average message length. In order to avoid the use of the lengthy term "minimum-redundancy," this term will be replaced here by "optimum." It will be understood then that, in this paper, "optimum code" means "minimum-redundancy code."

The following basic restrictions will be imposed on an ensemble code:

(a) No two messages will consist of identical arrangements of coding digits.

(b) The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of messages is known.

Restriction (b) necessitates that no message be coded in such a way that its code appears, digit for digit, as the first part of any message code of greater length. Thus, 01, 102, 111, and 202 are valid message codes for an ensemble of four members. For instance, a sequence of these messages 111022020101111102 can be broken up into the individual messages 111-102-202-01-01-111-102. All the receiver need know is the ensemble code. However, if the ensemble has individual message codes including 11, 111, 102, and 02, then when a message sequence starts with the digits 11, it is not immediately certain whether the message 11 has been received or whether it is only the first two digits of the message 111. Moreover, even if the sequence turns out to be 11102, it is still not certain whether 111-02 or 11-102 was transmitted. In this example, change of one of the two message codes 111 or 11 is indicated.

C. E. Shannon[1] and R. M. Fano[2] have developed ensemble coding procedures for the purpose of proving that the average number of binary digits required per message approaches from above the average amount of information per message. Their coding procedures are not optimum, but approach the optimum behavior when N approaches infinity. Some work has been done by Kraft[3] toward deriving a coding method which gives an average code length as close as possible to the ideal when the ensemble contains a finite number of members. However, up to the present time, no definite procedure has been suggested for the construction of such a code

* Decimal classification: R531.1. Original manuscript received by the Institute, December 6, 1951.

† Massachusetts Institute of Technology, Cambridge, Mass.

[1] C. E. Shannon, "A mathematical theory of communication," *Bell Sys. Tech. Jour.*, vol. 27, pp. 398–403; July, 1948.

[2] R. M. Fano, "The Transmission of Information," Technical Report No. 65, Research Laboratory of Electronics, M.I.T., Cambridge, Mass.; 1949.

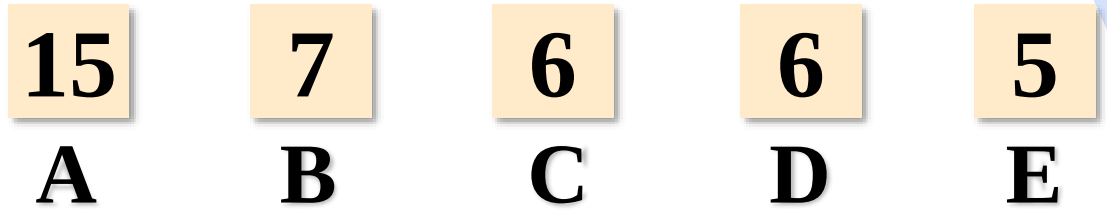
[3] L. G. Kraft, "A Device for Quantizing, Grouping, and Coding Amplitude-modulated Pulses," Electrical Engineering Thesis, M.I.T., Cambridge, Mass.; 1949.

Huffman算法

A

重复做：选择权值最小的两个结点生成新结点，新结点作为原结点的父亲，权值是原来两个结点权值之和。

字符	出现次数
A	15
B	7
C	6
D	6
E	5

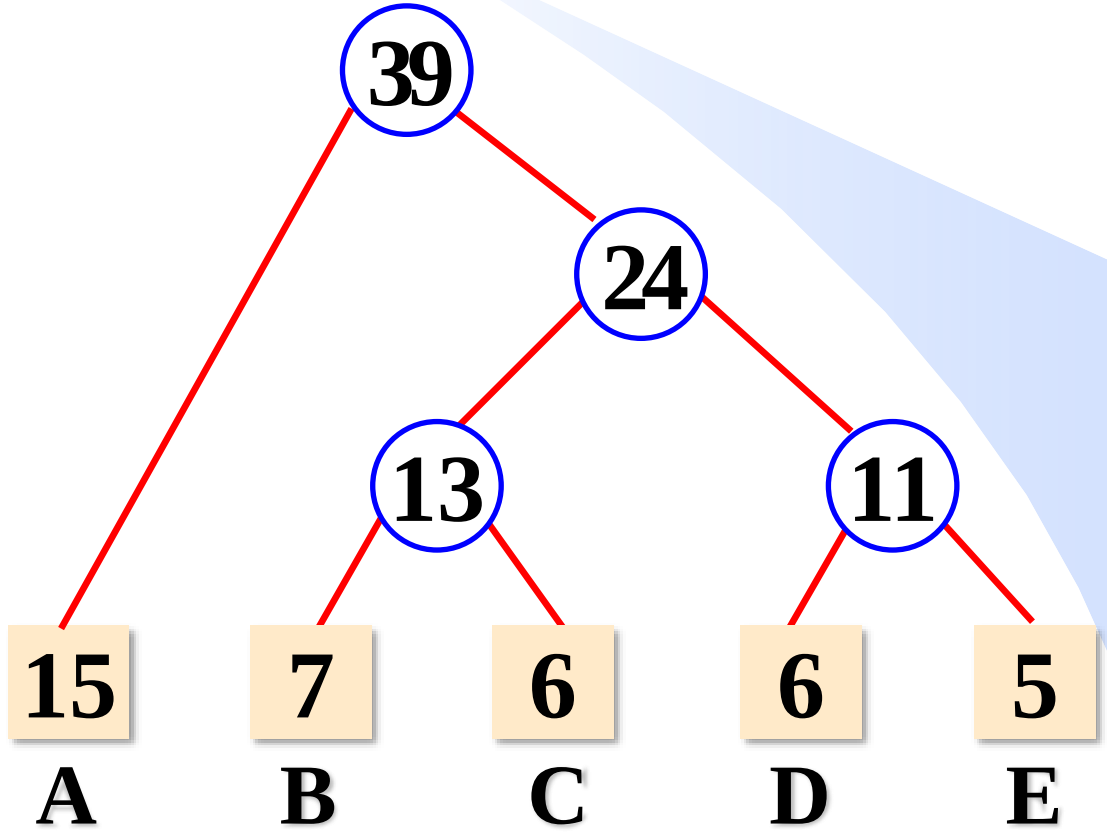


Huffman算法

A

重复做：选择权值最小的两个结点生成新结点，新结点作为原结点的父亲，权值是原来两个结点权值之和。

字符	出现次数
A	15
B	7
C	6
D	6
E	5



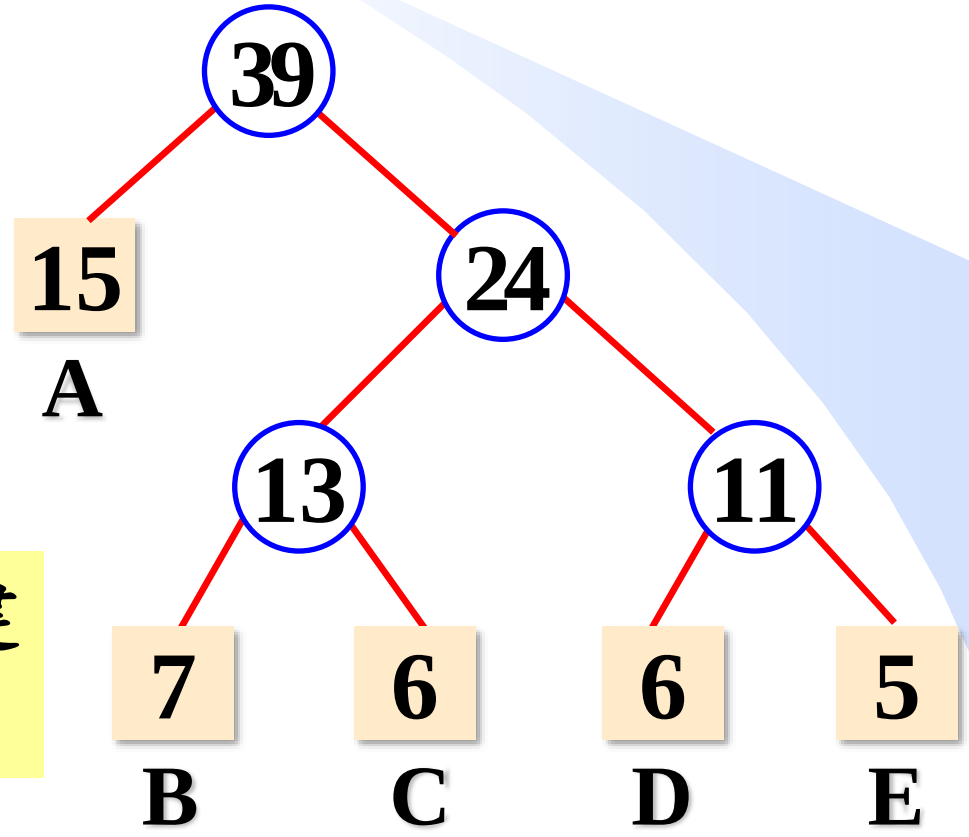
Huffman算法

A

重复做：选择权值最小的两个结点生成新结点，新结点作为原结点的父亲，权值是原来两个结点权值之和。

字符	出现次数
A	15
B	7
C	6
D	6
E	5

自下而上构建Huffman树

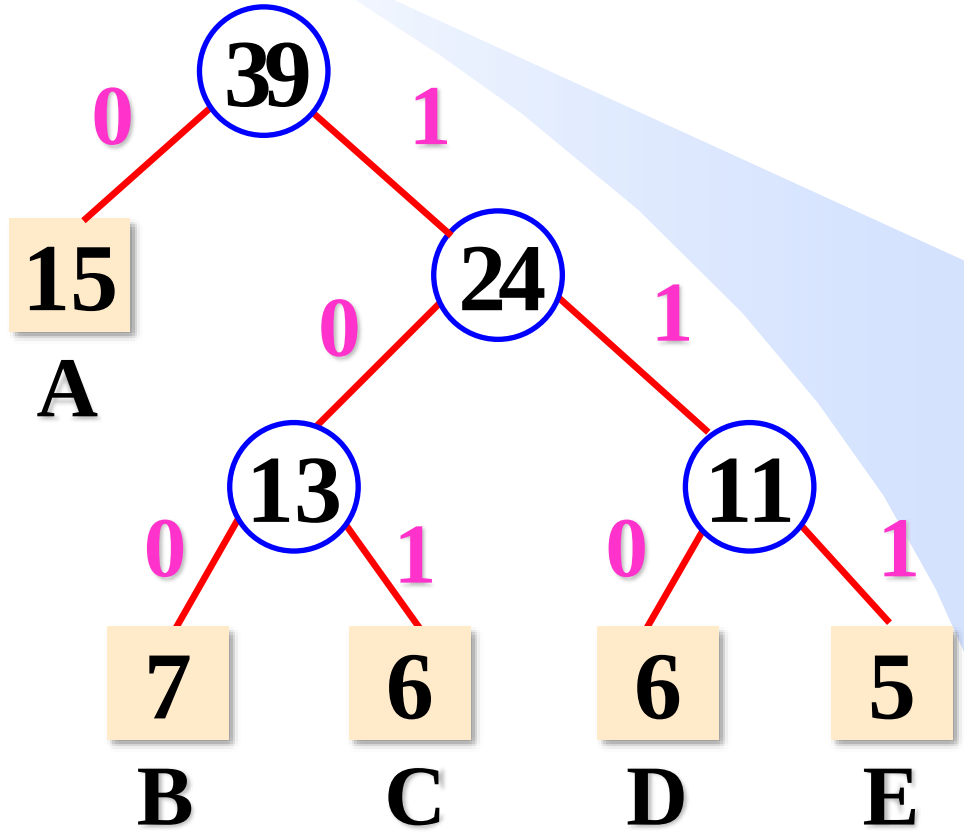


Huffman编码

A

每个左分支标记0，右分支标记1。把从根到叶的路径上的标号连接起来，作为该叶结点所代表的字符的编码。

字符	出现次数	编码
A	15	0
B	7	100
C	6	101
D	6	110
E	5	111



WPL=15*1+(7+6+6+5)*3=87

Huffman算法

A

① 根据给定的n个权值$w_1, w_2, \ldots, w_n$构成n棵二叉树$T_1, T_2, \ldots, T_n$，每棵二叉树T_i中都只有一个结点，其权值为w_i；

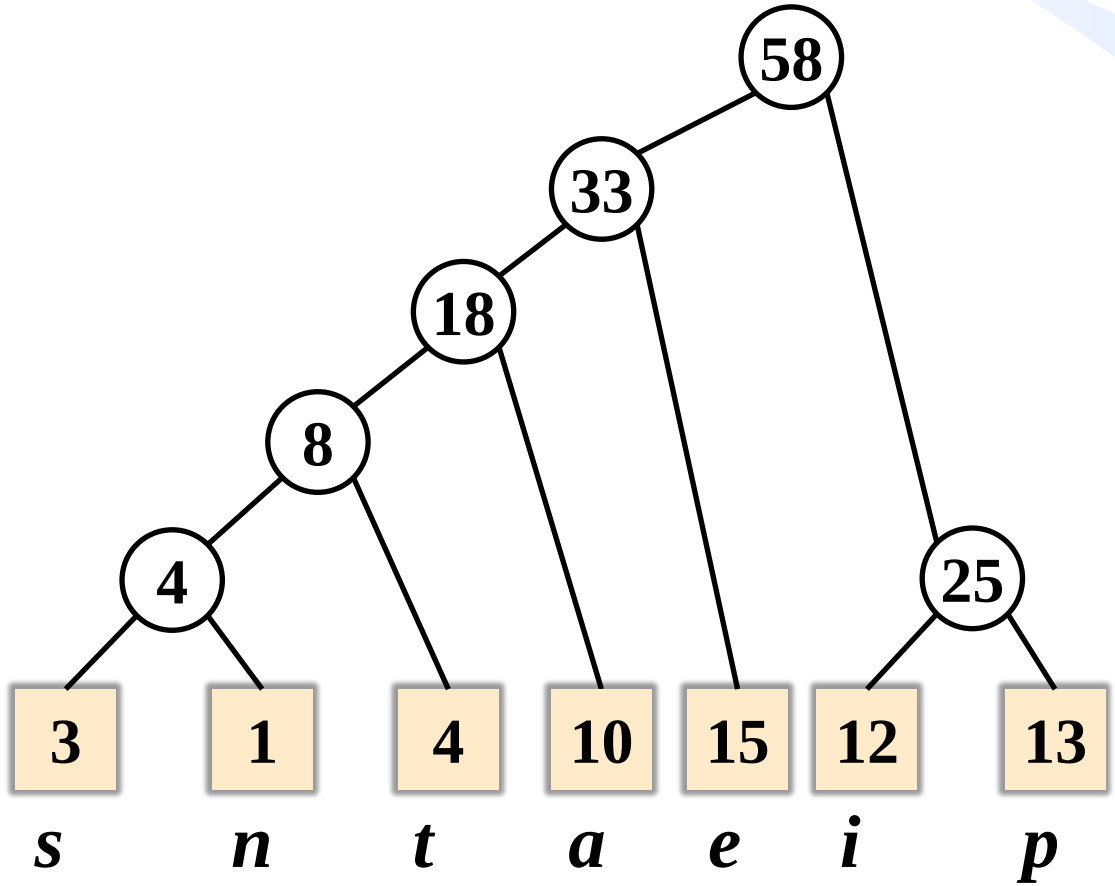
② 选出权值最小的两个根结点合并成一棵二叉树：生成一个新结点作为这两个结点的父结点，新结点的权值为其两个孩子的权值之和。

③ 重复步骤②，直至只剩一棵二叉树为止，此二叉树便是哈夫曼树。

构建Huffman树示例

A

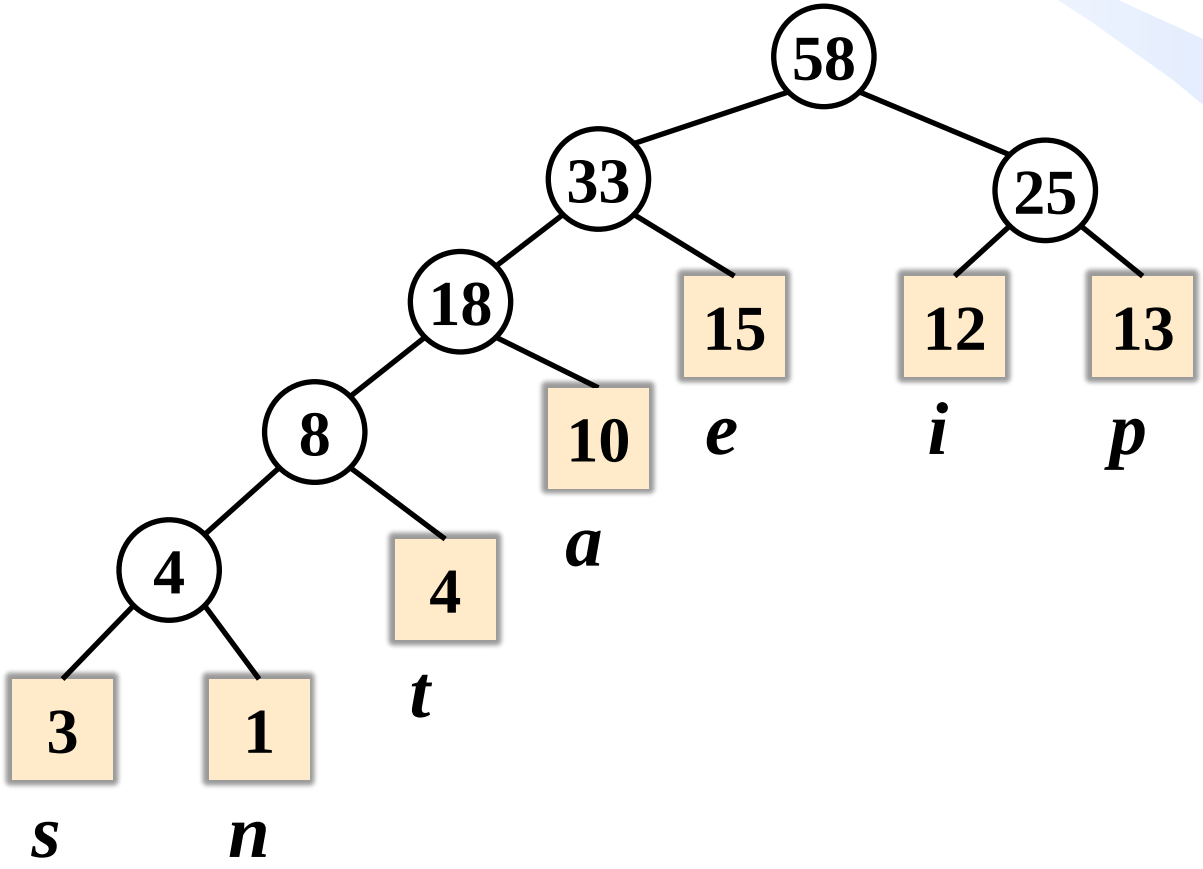
假设有一文件包含7种字符：*a*、*e*、*i*、*s*、*t*、*p*、*n*, 且文件中有10个*a*, 15个*e*, 12个*i*, 3个*s*, 4个*t*, 13个*p*, 1个*n* .



构建Huffman树示例

A

假设有一文件包含7种字符：*a*、*e*、*i*、*s*、*t*、*p*、*n*，且文件中有10个*a*, 15个*e*, 12个*i*, 3个*s*, 4个*t*, 13个*p*, 1个*n* .

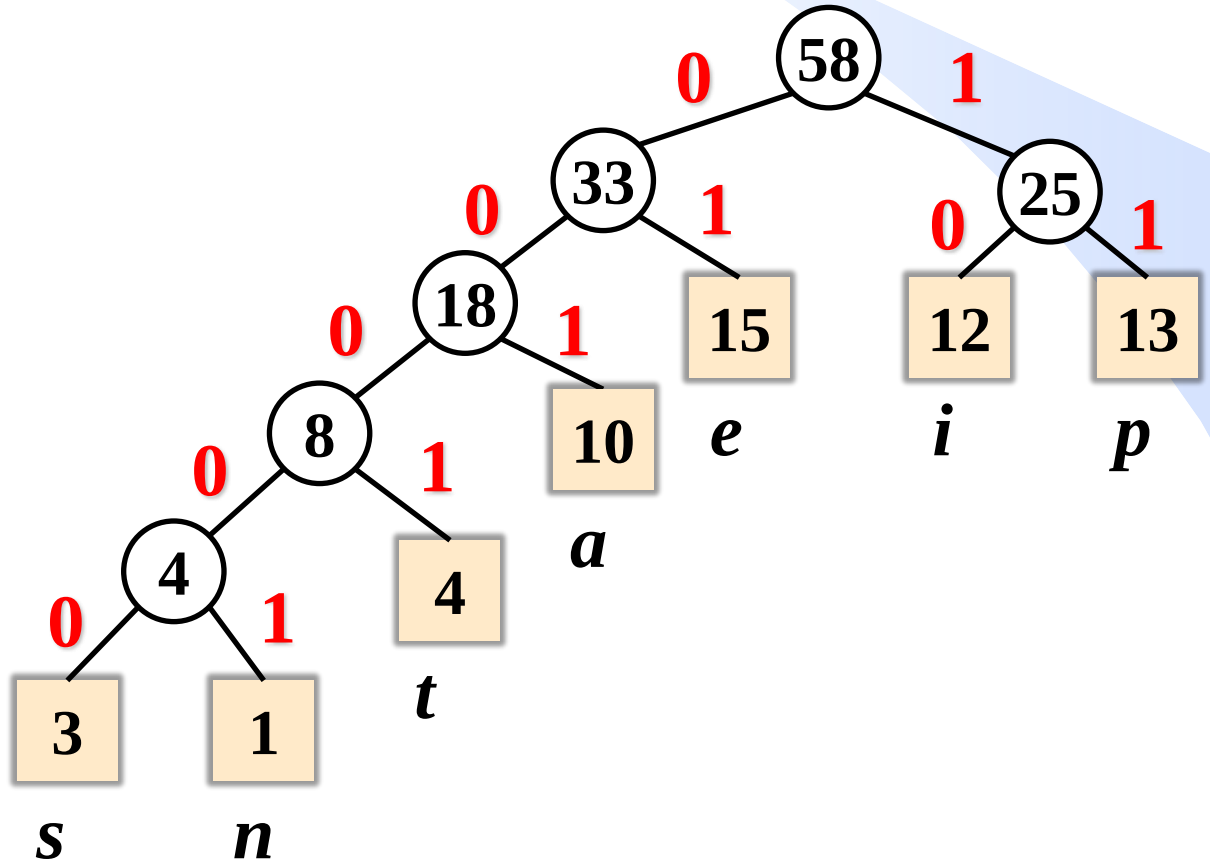


Huffman编码

A

- 对哈夫曼树每个非叶结点的左分支标记0，右分支标记1。
- 把从根到叶的路径上的标号连接起来，作为该叶结点所代表的字符的编码。

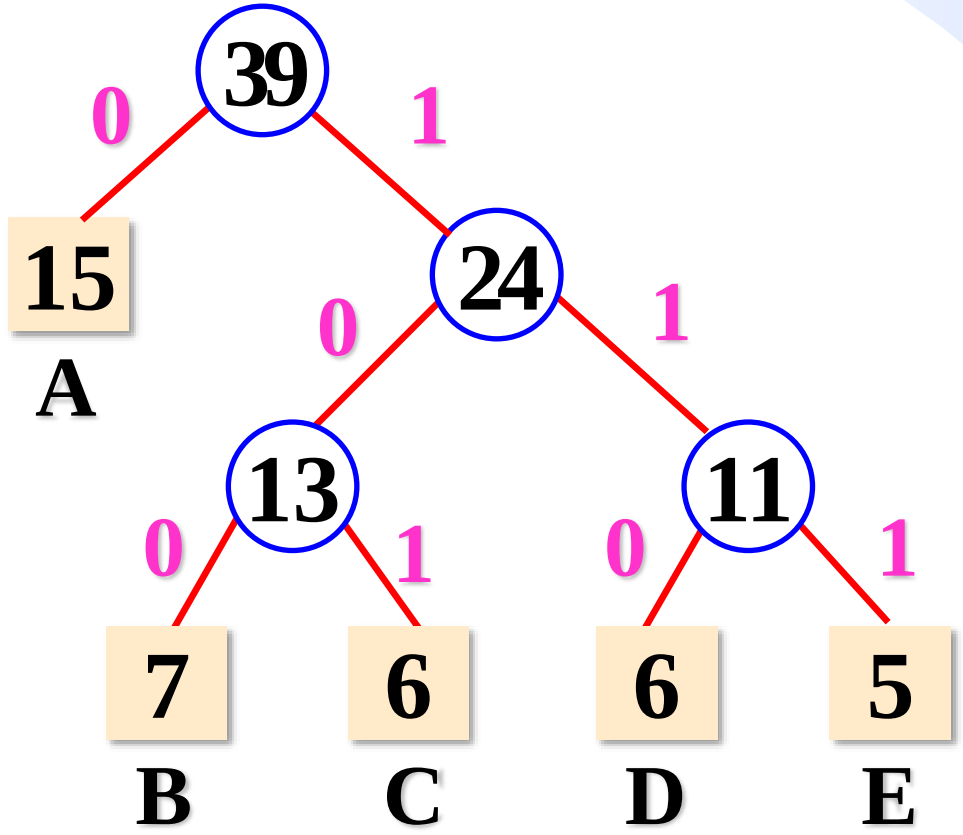
s的编码是：	00000
n 的编码是	00001
t的编码是：	0001
a的编码是：	001
e的编码是：	01
i的编码是：	10
p的编码是：	11



哈夫曼编码是否是“无前缀冲突编码”？

A

字符对应叶结点，任意一个叶结点不可能是其他叶结点的祖先，每个叶结点对应的编码不可能是其他叶结点对应的编码的前缀，故哈夫曼编码是无前缀冲突编码。



课下思考

A

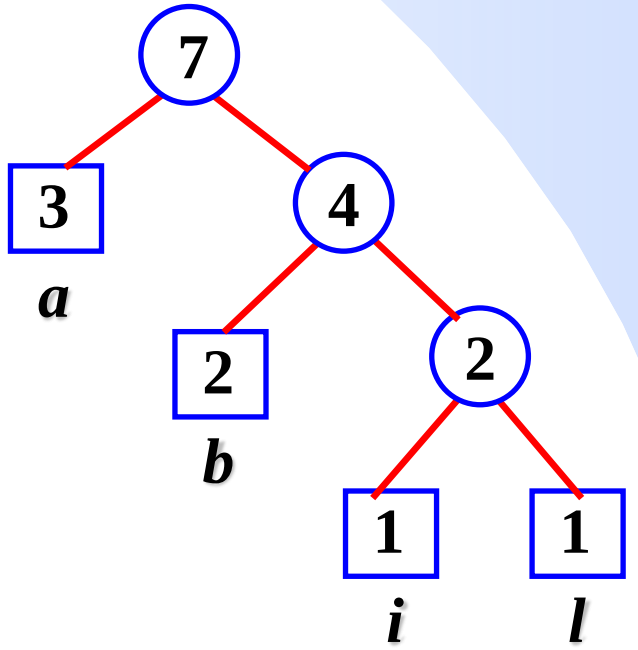
字符串"*alibaba*"的Huffman编码总长度有___位（bit）。

【阿里笔试题】

A. 11　　B. 12　　C. 13　　D. 14

文本中有3个*a*，2个*b*，1个*i*，1个*l*

WPL=13



课下思考

A

在有6个字符组成的字符集S中，各字符出现的频次分别为3、4、5、6、8、10，为S构造的哈夫曼树的加权路径长度WPL为____。【2023年考研题全国卷】

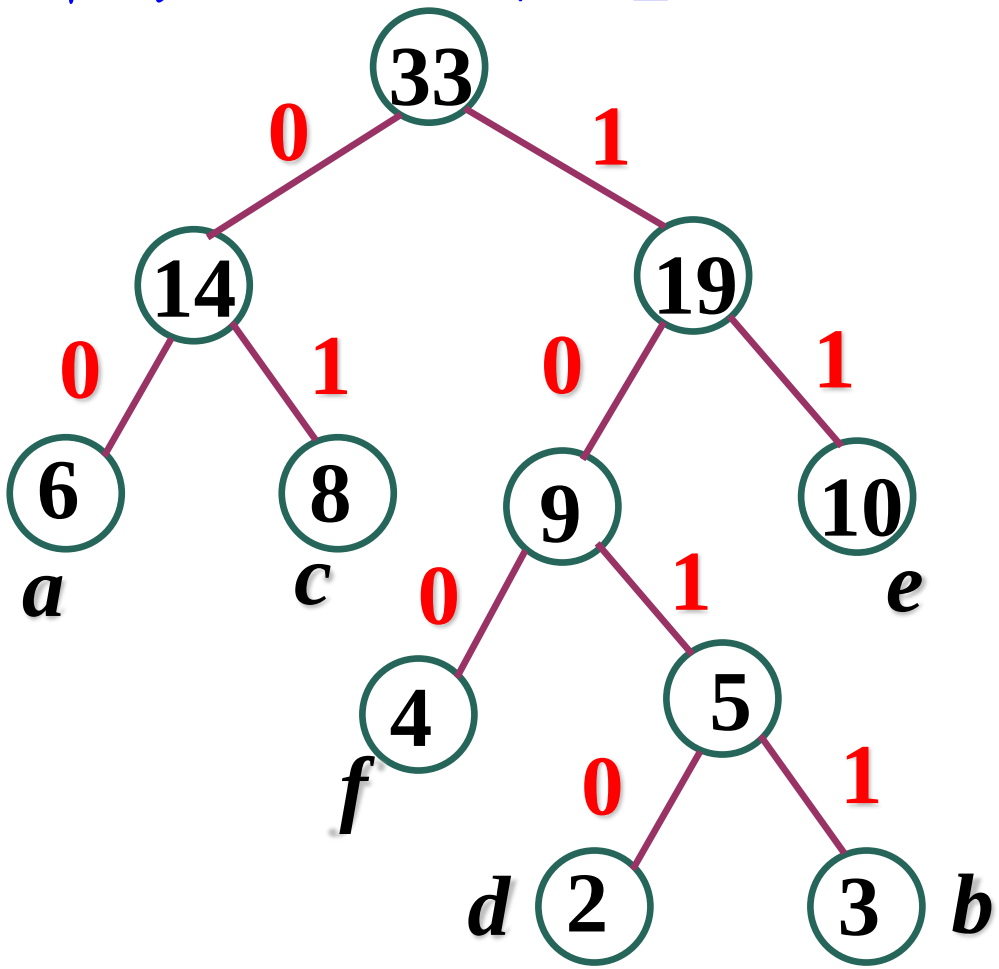
A. 86　　B. 90　　C. 96　　D. 99

课下思考

A

已知字符集合是{ a, b, c, d, e, f }，各个字符出现的次数分别是6，3，8，2，10，4，则各字符对应的哈夫曼编码为_____

【2018年考研题全国卷】



编码

a (00)

b (1011)

c (01)

d (1010)

e (11)

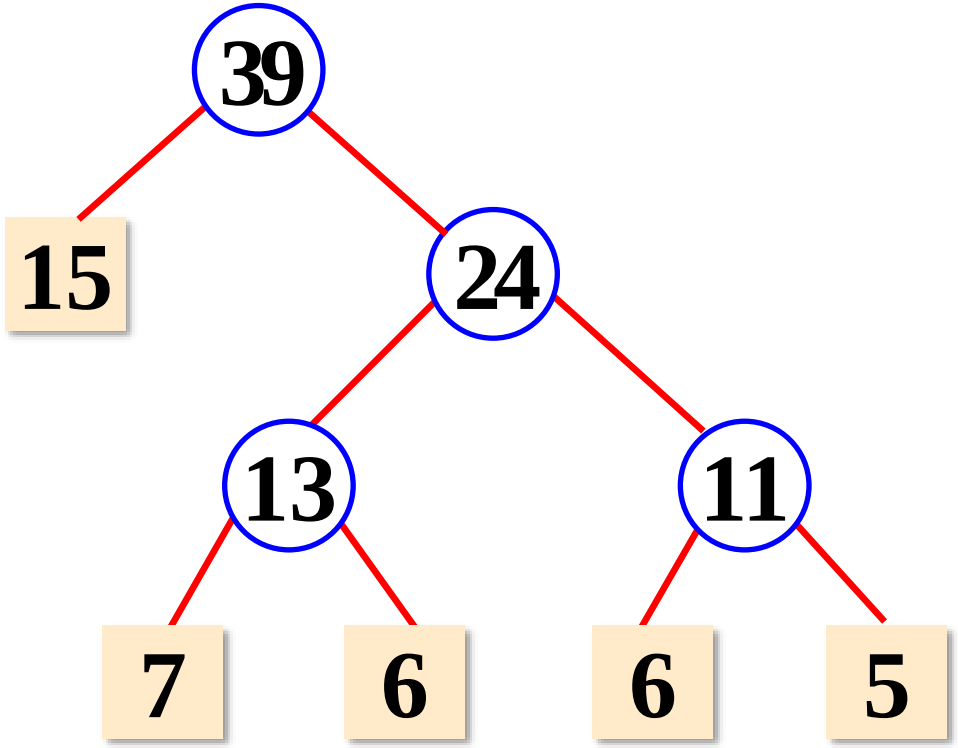
f (100)

哈夫曼树——二子性

A

- 哈夫曼树中包含度为1的结点么？
  - ✓ 哈夫曼树不包含度为1的结点。
  - ✓ 若哈夫曼树n个叶结点，则必有n-1个非叶结点，一共2n-1个结点。

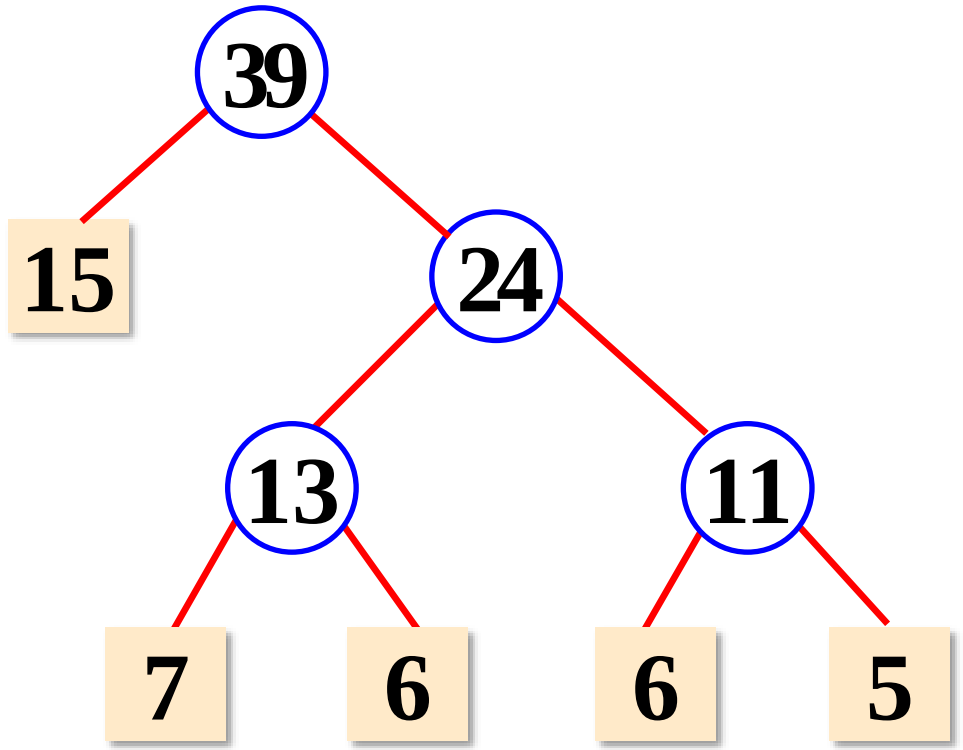
引理 $n_0=n_2+1$.



哈夫曼树——同权不同构

A

- 哈夫曼树、哈夫曼编码、最小编码长度唯一么？
  - ✓哈夫曼树形态不唯一、编码不唯一。
  - ✓对任意内结点而言，其左右子树互换后WPL（加权路径长度）不变，故最小编码长度唯一。



Huffman算法实现

B

哈夫曼树中每个结点的结构为：

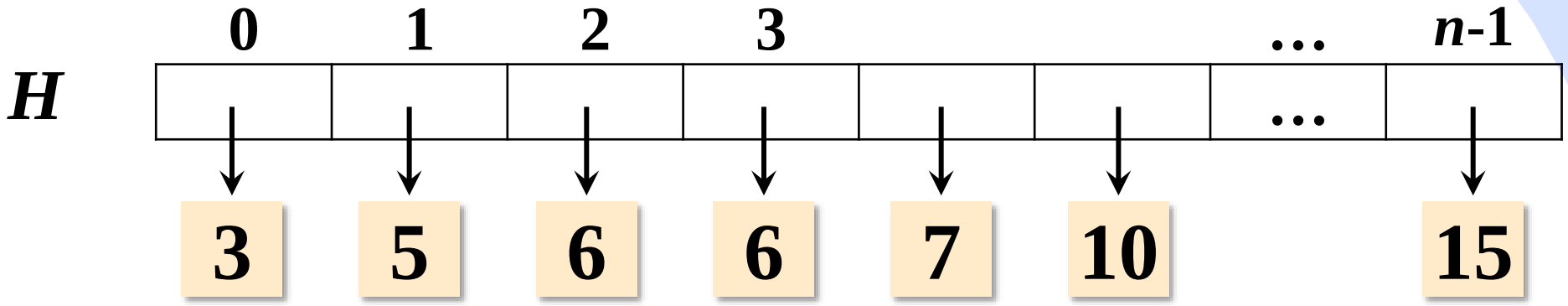
left	*data*	*weight*	*right*

其中*left*和*right*为链接域，*data*为字符值，*weight*为该结点的权值。

```
struct HuffmanNode{
   char data=' ';
   int weight;
   HuffmanNode *left,*right;
};
```

将哈夫曼树结点的指针存于一维数组*H*[ ]中

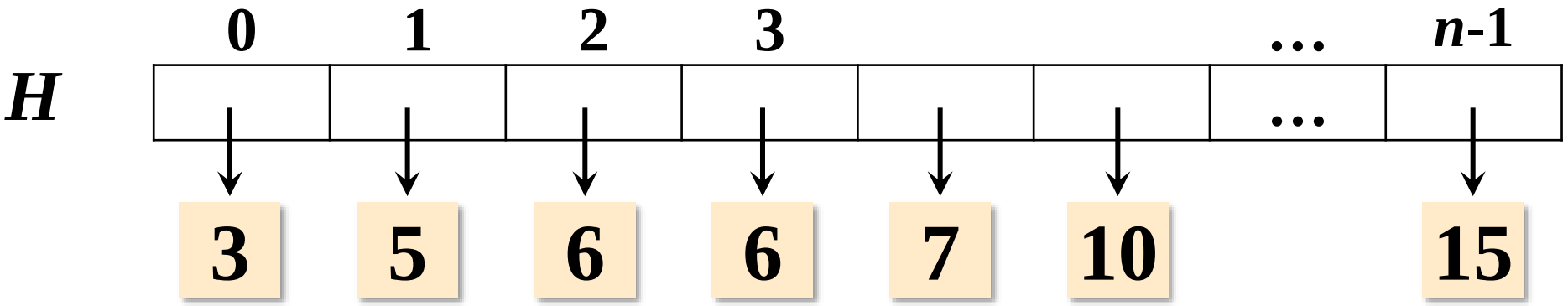
```
const int maxn=300;
HuffmanNode *H[maxn];
```



Huffman算法实现

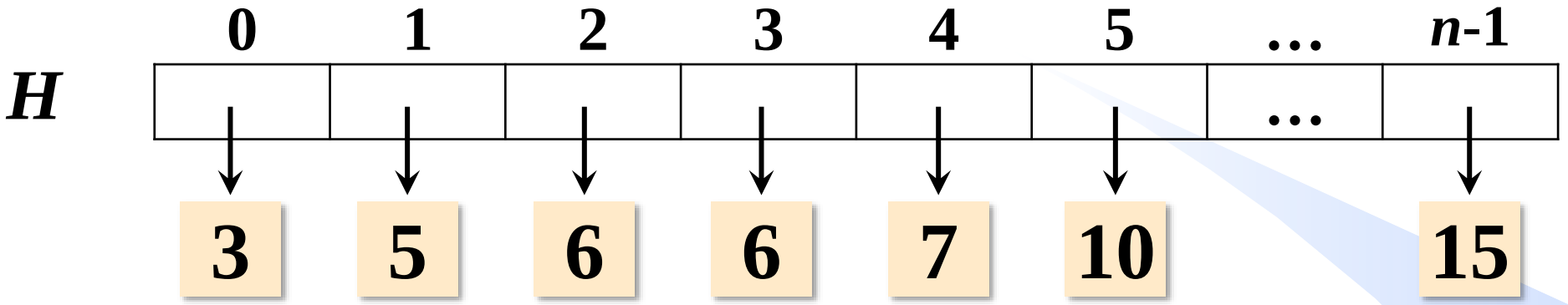
B

```
const int maxn=300; HuffmanNode *H[maxn];
void CreateHuffmanNode(char ch[], int freq[], int n){
   //有n种字符，第i种字符存放在ch[i]，其频率存放在freq[i]中
   for(int i = 0; i < n; i++) {
      H[i] = new HuffmanNode;
      H[i]->data = ch[i];  H[i]->weight = freq[i];
      H[i]->left = H[i]->right = NULL;
   }
   sort(H, n); //对H按weight值递增排序，sort函数需预先实现
}
```



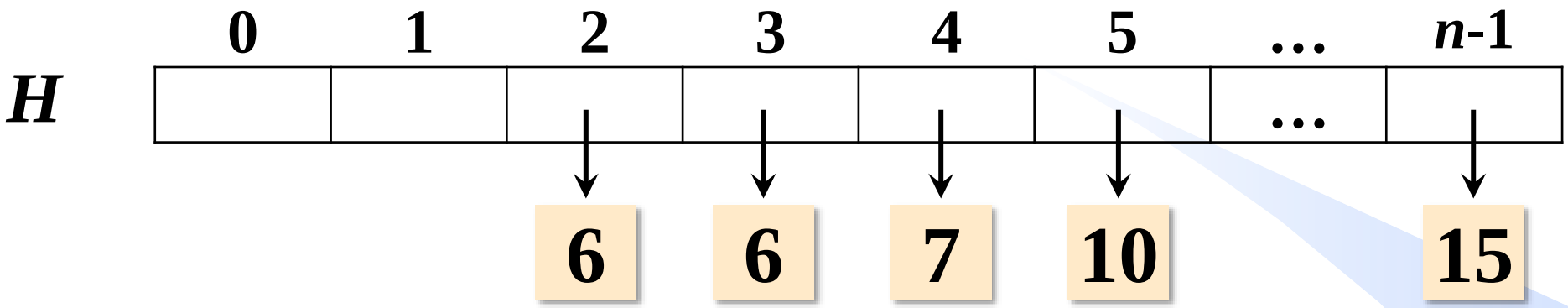
Huffman算法实现

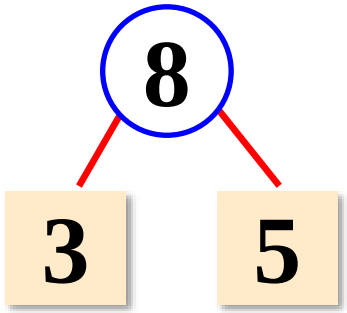
B



Huffman算法实现

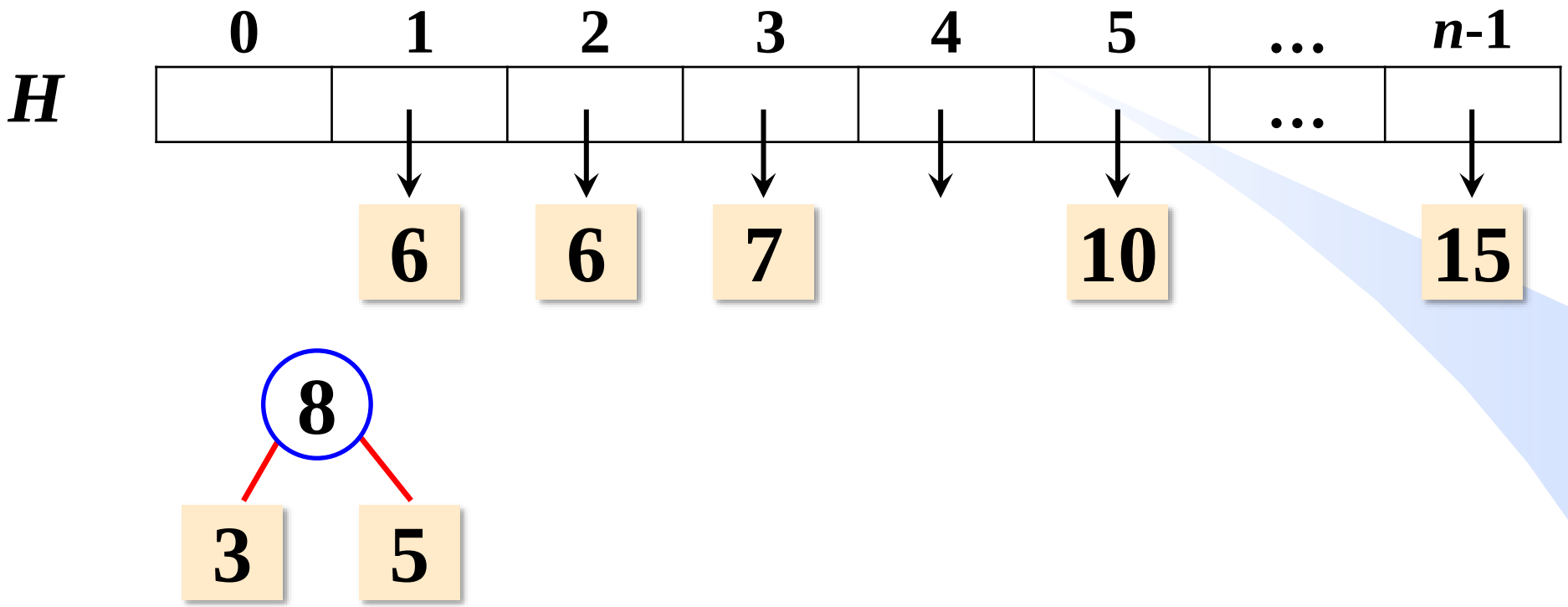
B





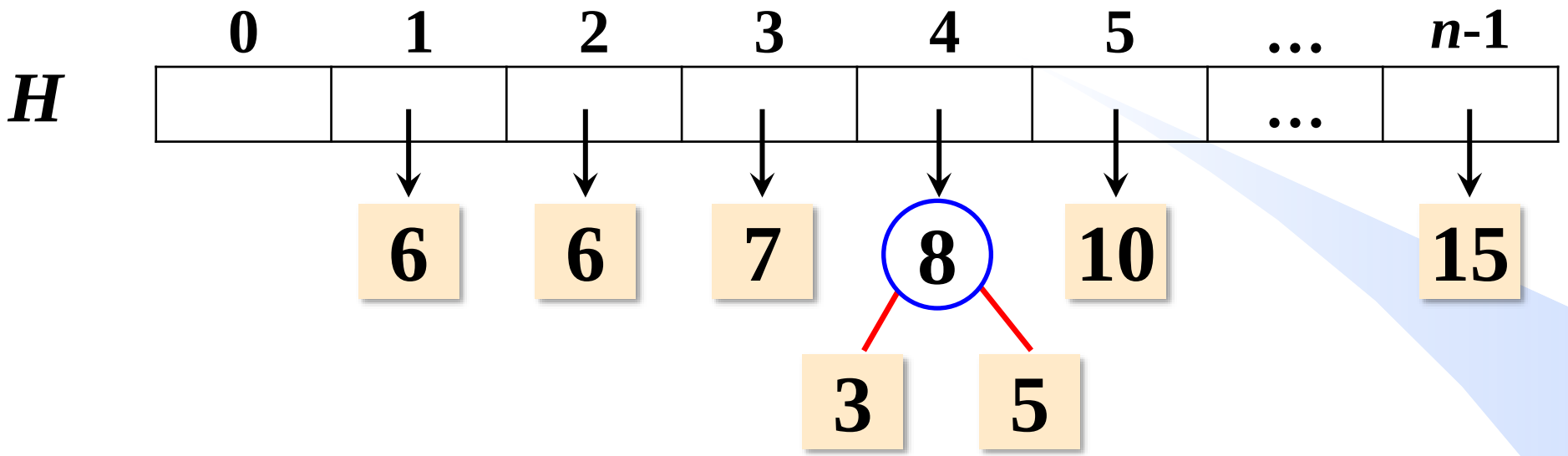
Huffman算法实现

B



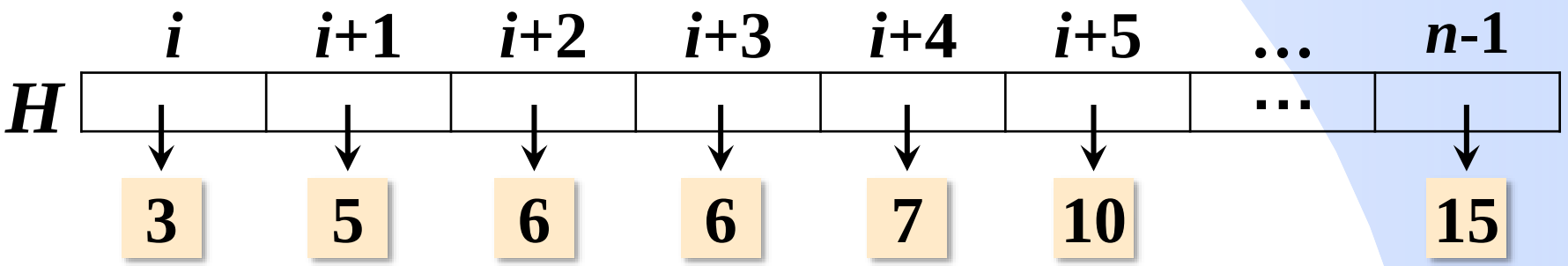
Huffman算法实现

B



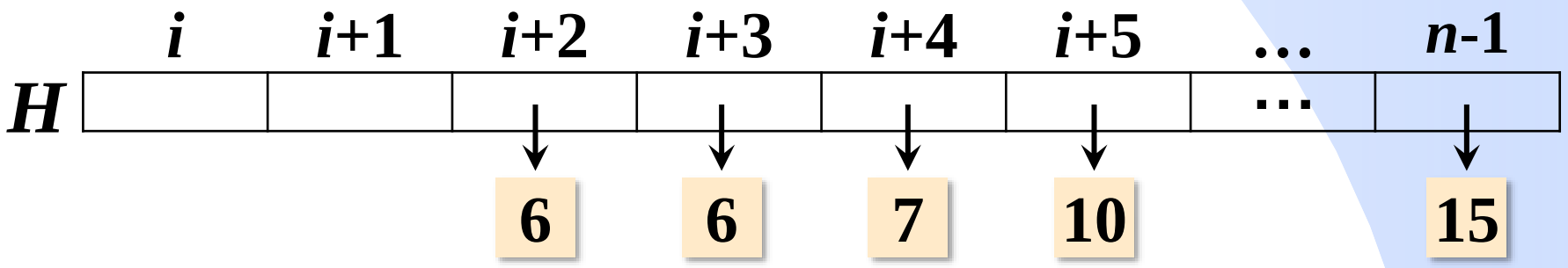
```
HuffmanNode* BuildHuffanmanTree(HuffmanNode *H[],int n){
   for(int i=0; i<n-1; i++){
      //未完待续
```

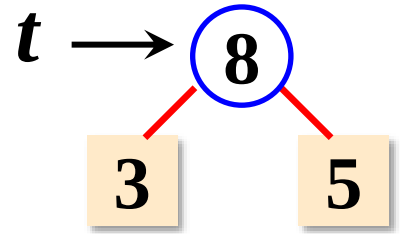
B



```
HuffmanNode* BuildHuffanmanTree(HuffmanNode *H[],int n){
   for(int i=0; i<n-1; i++){
      HuffmanNode *t = new HuffmanNode;
      t->left=H[i];  t->right=H[i+1];
      t->weight = t->left->weight + t->right->weight;
      //把新结点t插入到数组H中，使得数组保持按权值升序排列
      int j = i+2;
      while(j < n && H[j]->weight < t->weight){
         H[j-1]=H[j];  j++;
      }
      //未完待续
```

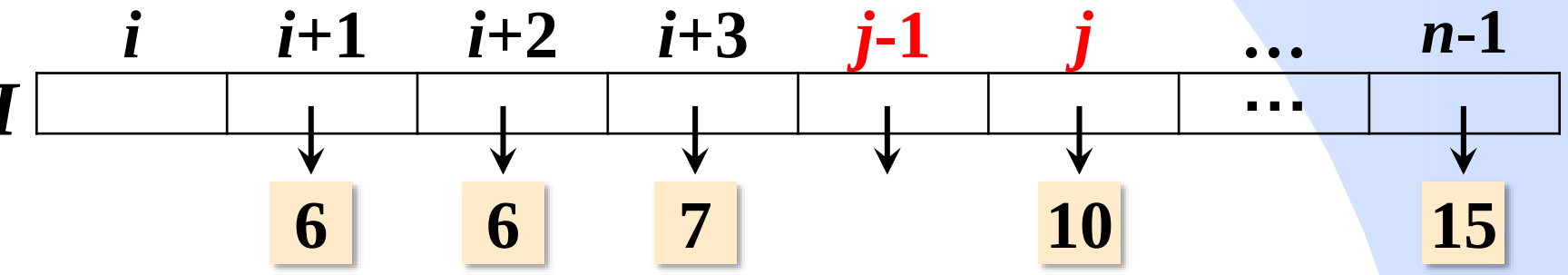
B

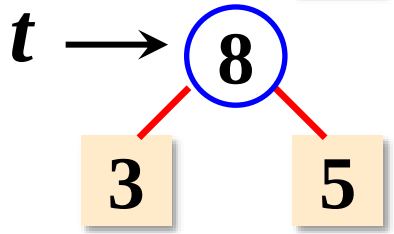




```
HuffmanNode* BuildHuffanmanTree(HuffmanNode *H[],int n){
   for(int i=0; i<n-1; i++){
      HuffmanNode *t = new HuffmanNode;
      t->left=H[i];  t->right=H[i+1];
      t->weight = t->left->weight + t->right->weight;
      //把新结点t插入到数组H中，使得数组保持按权值升序排列
      int j = i+2;
      while(j < n && H[j]->weight < t->weight){
         H[j-1]=H[j];  j++;
      }
      H[j-1] = t;
}
//未完待续
```

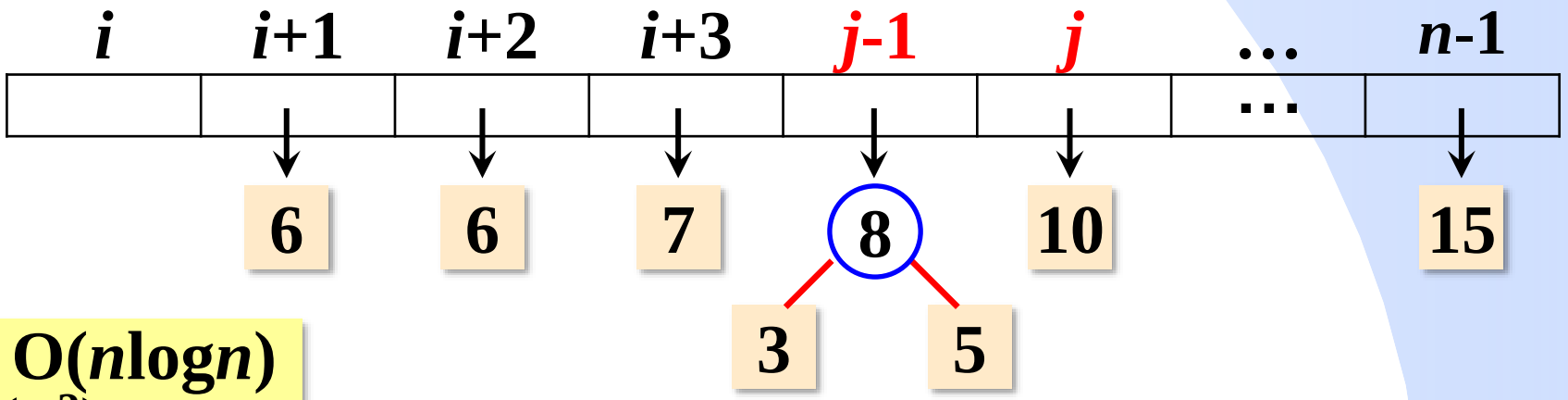
B





```
HuffmanNode* BuildHuffanmanTree(HuffmanNode *H[],int n){
   for(int i=0; i<n-1; i++){
      HuffmanNode *t = new HuffmanNode;
      t->left=H[i];  t->right=H[i+1];
      t->weight = t->left->weight + t->right->weight;
      //把新结点t插入到数组H中，使得数组保持按权值升序排列
      int j = i+2;
      while(j < n && H[j]->weight < t->weight){
         H[j-1]=H[j];  j++;
      }
      H[j-1] = t;
   }
   return H[n-1];
}
```

B



- ✓ 预处理（排序）：$O(n\log n)$
- ✓ 构建哈夫曼树：$O(n^2)$
- ✓ 总时间复杂度$O(n^2)$

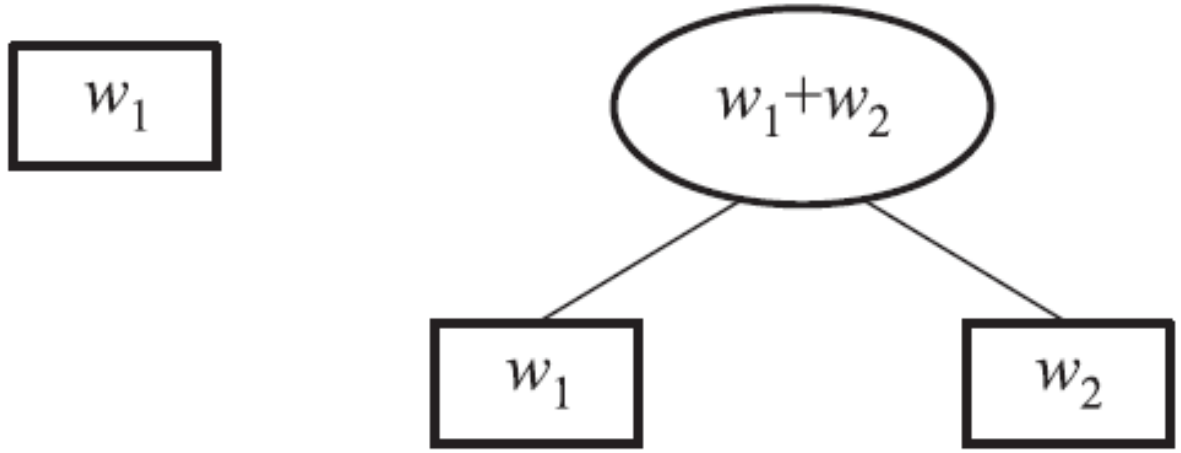
Huffman算法的正确性

C

定理 外结点权值$w_1 \le w_2 \le \ldots \le w_n$的扩充二叉树中，由Huffman算法构造出的是一棵最优二叉树。

证明：数学归纳法。

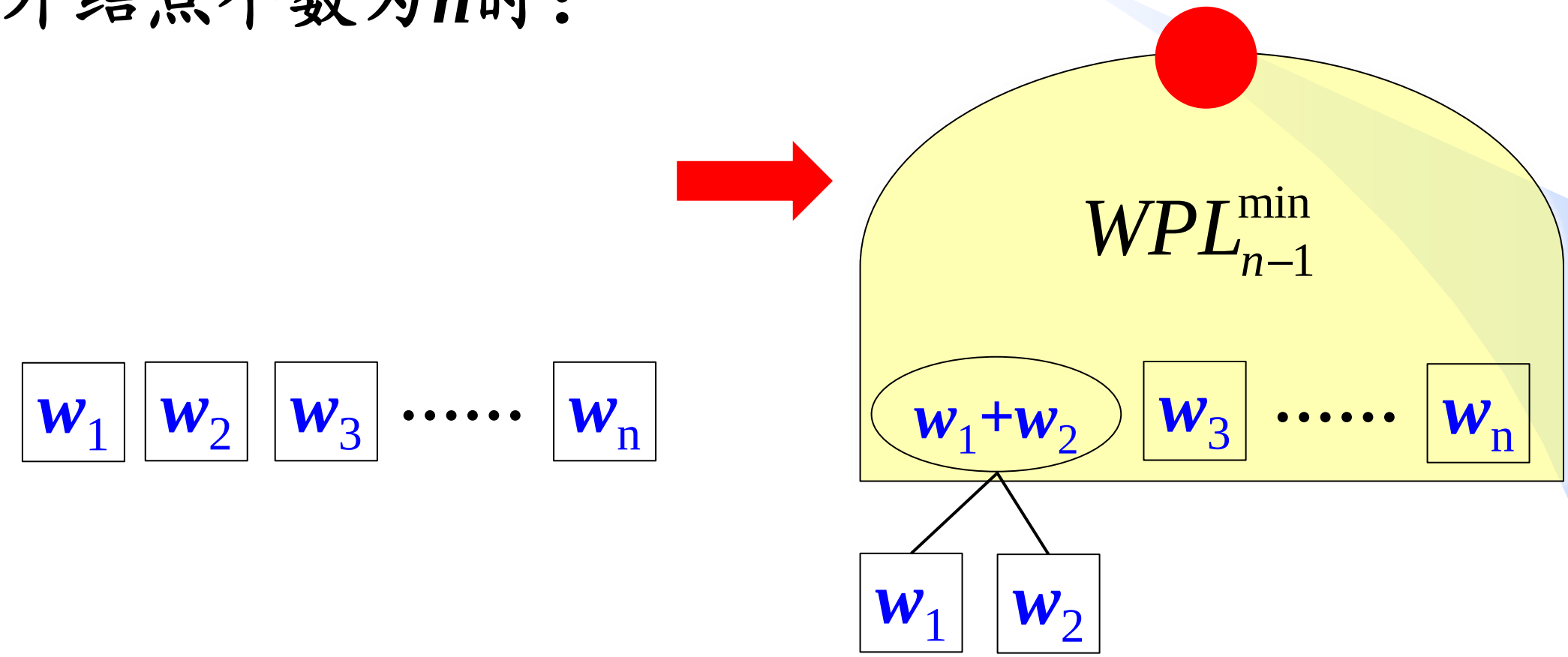
n=1, 2时



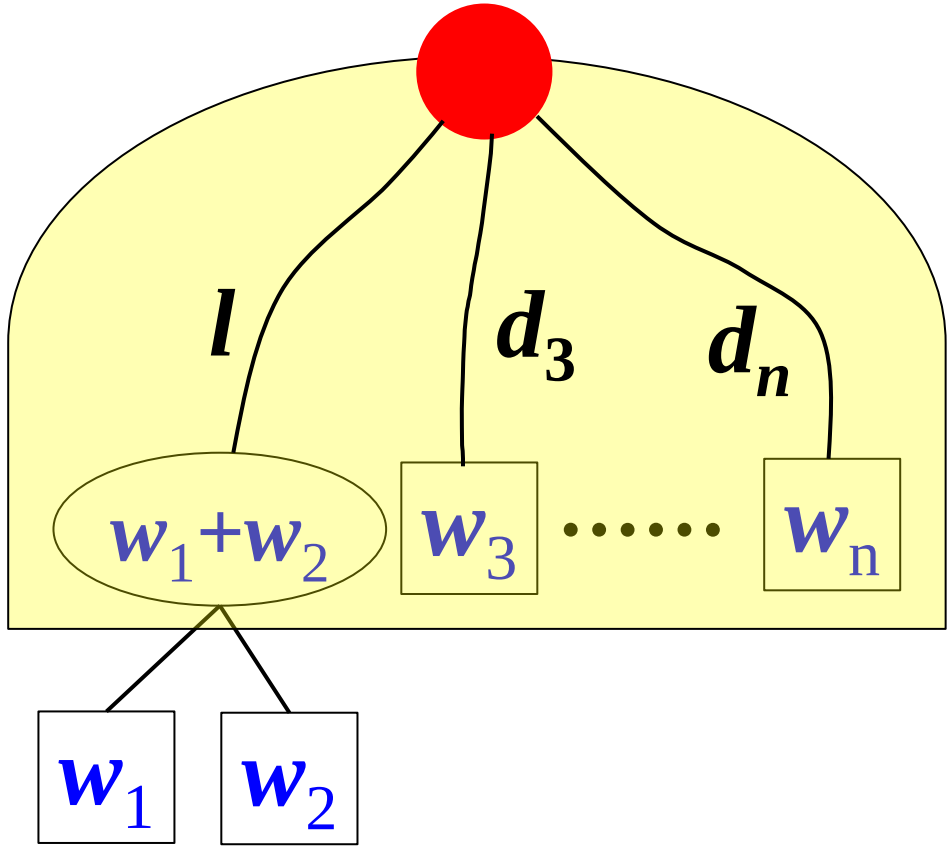
Huffman算法的正确性

C

- 设外结点个数为n-1时算法正确。
- 外结点个数为n时：



C



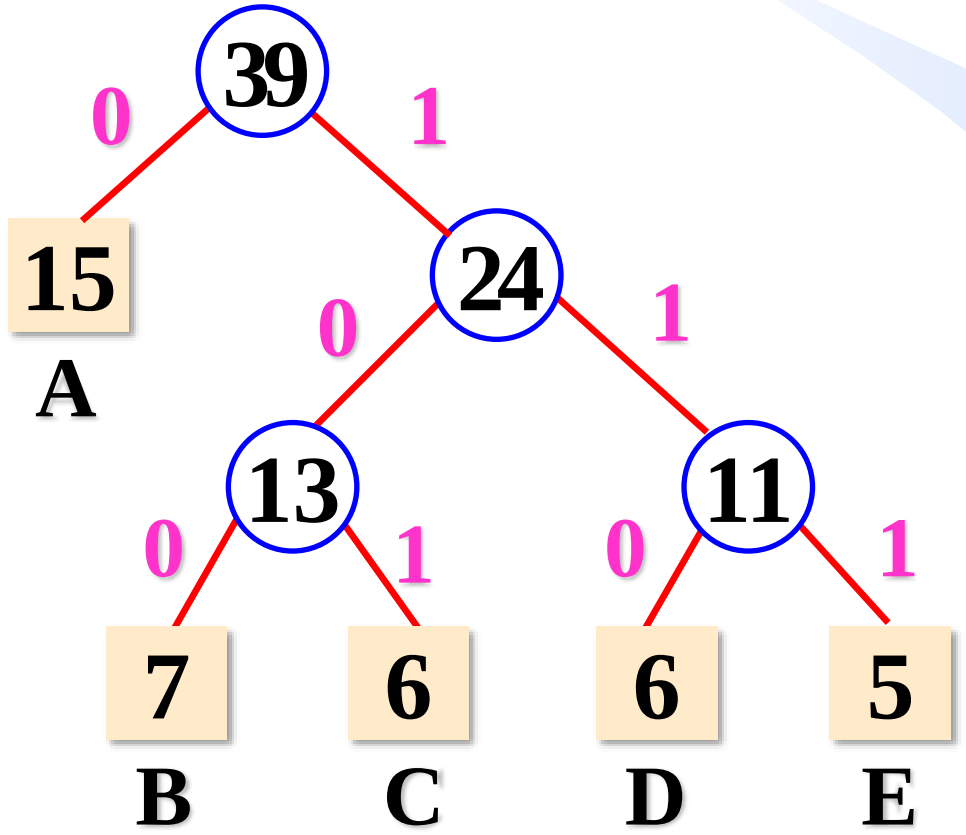
$$WPL_n$$
$$= w_1(l+1) + w_2(l+1) + w_3 d_3 + \ldots + w_n d_n$$
$$= (w_1 + w_2)(l+1) + w_3 d_3 + \ldots + w_n d_n$$
$$= (w_1 + w_2)l + w_1 + w_2 + w_3 d_3 + \ldots + w_n d_n$$
$$= WPL_{n-1}^{\min} + w_1 + w_2$$

$$WPL_n^{\min} = WPL_{n-1}^{\min} + w_1 + w_2$$

哈夫曼算法——贪心算法

C

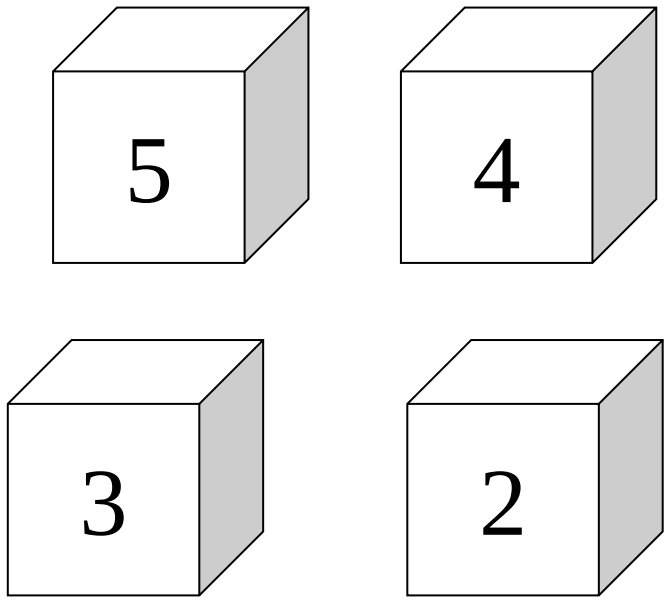
- 每步都选最优的方案



贪心算法

C

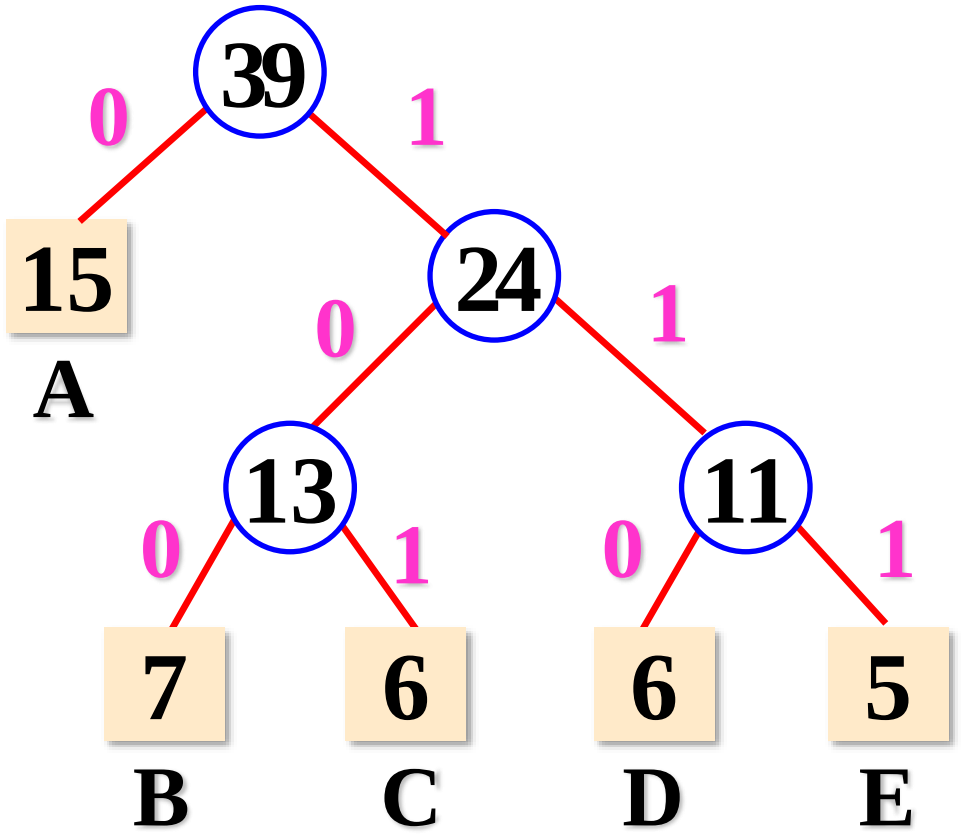
- 每步都选最优的方案，最后不见得总能得到最优解
- 步步最优，并不一定全局最优



解码

A

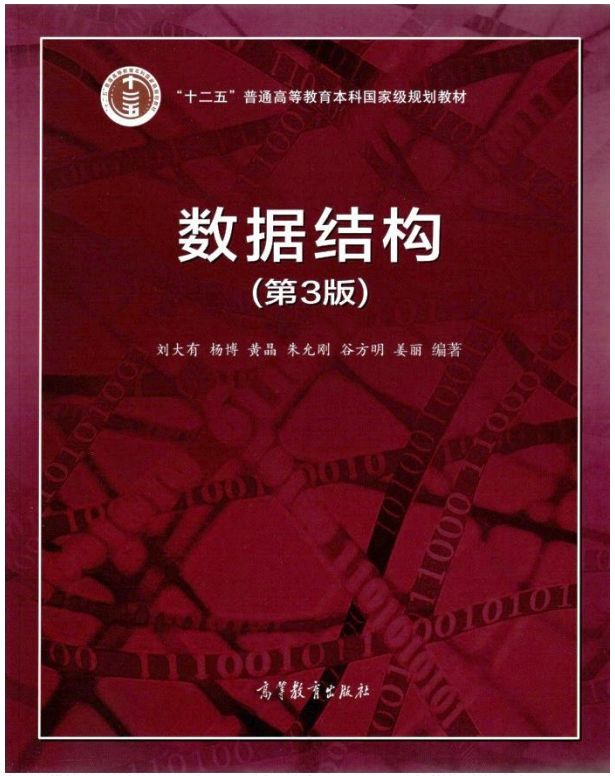
解码过程：依次读入文件的二进制码，从哈夫曼树的根结点出发，若当前读入0，则走向其左孩子，否则走向其右孩子，到达某一叶结点时，便可以译出相应的字符。



字符	编码
A	0
B	100
C	101
D	110
E	111

解码？

1 0 1 1 0 0 0



树和二叉树的应用

- 压缩与哈夫曼树
- **表达式树**
- 并查集
- 其他应用举例

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2025.4

B

中缀表达式

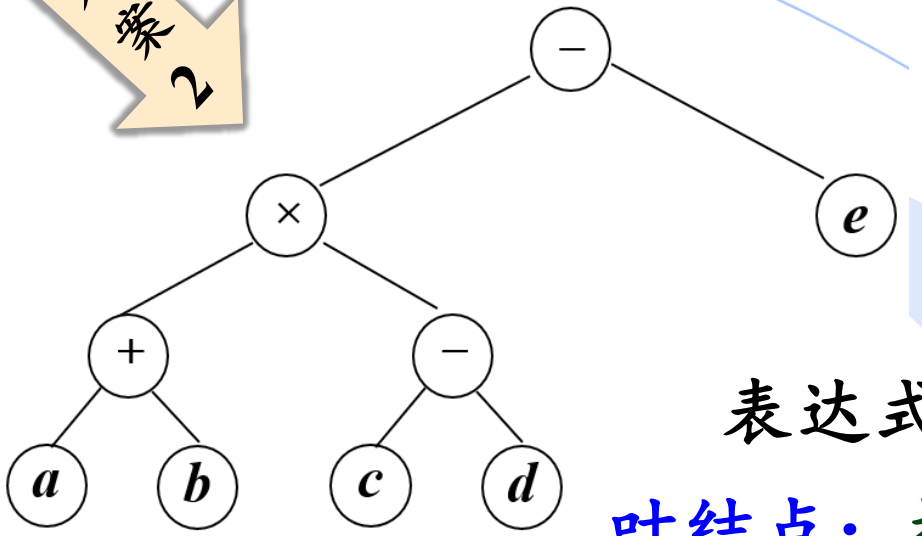
$$(a+b)\times(c-d)-e$$

方案1

后缀表达式

$$a\ b+c\ d-\times e-$$

方案2



表达式树

叶结点：操作数

非叶结点：运算符

中根序列：中缀表达式

后根序列：后缀表达式

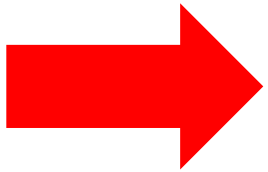
（1）灵活：可方便得到中/后缀表达式

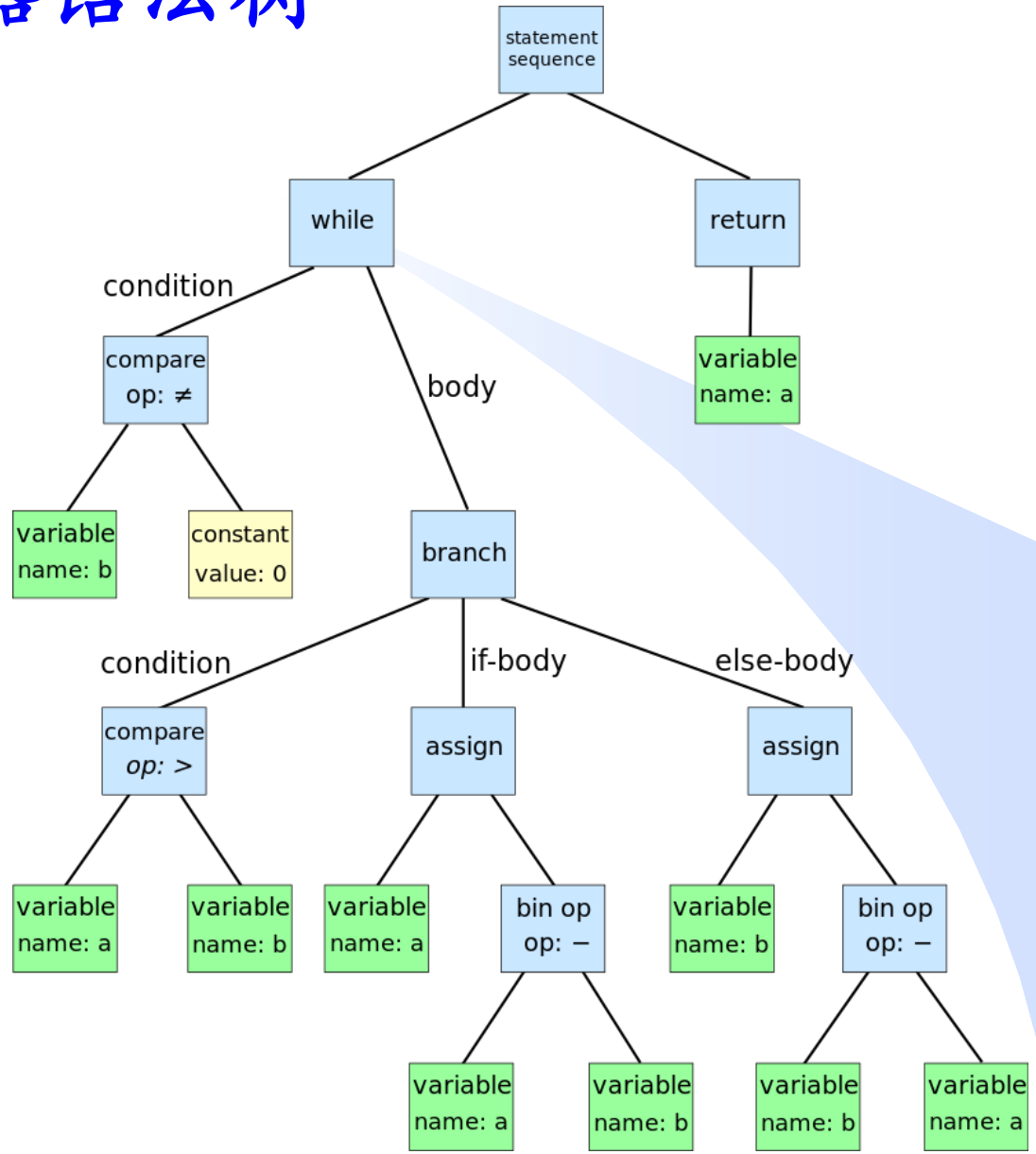
（2）不仅可以分析表达式，也可以表示整个代码（if，for，while）结构，更有利于编译器对整个代码的分析和优化。

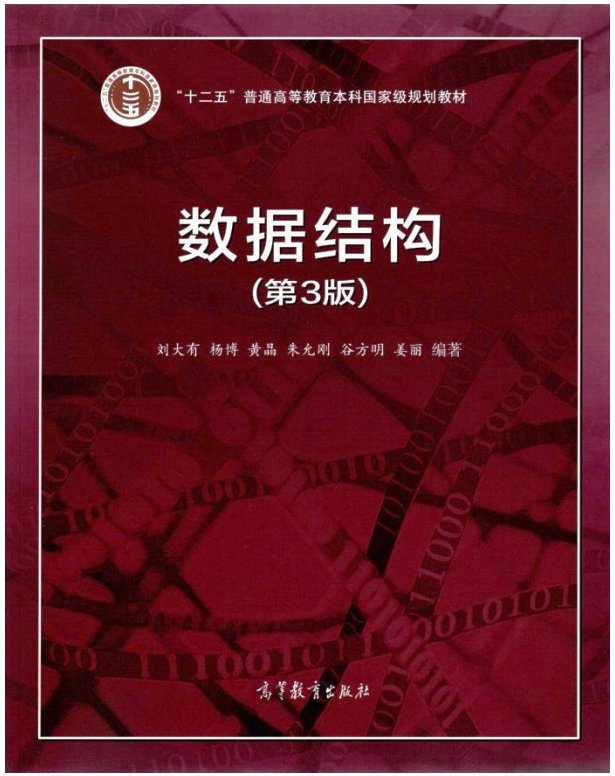
编译器语法树

D

```
while (b != 0)
   if (a > b)
      a = a - b;
   else
      b = b - a;
return a;
```







树和二叉树的应用

- 压缩与哈夫曼树
- 表达式树
- **并查集**

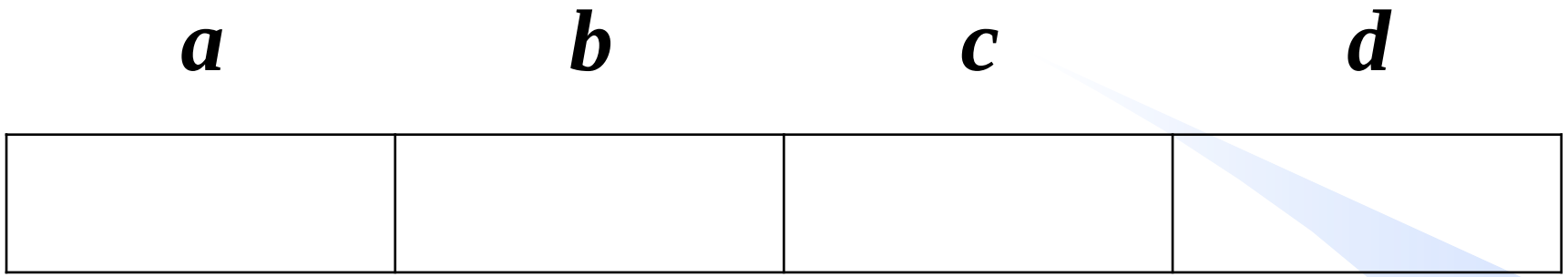
数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2025.4

如何实现集合?

A



如何实现集合?

A

a	b	c	d
1	0	1	1

1	0	0	1

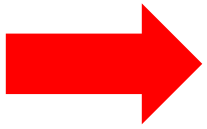
求交集 → O(n)

如何实现集合?

A

a	*b*	*c*	*d*
1	0	1	1

1	0	0	1

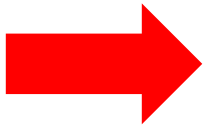
求并集  O(*n*)

如何实现集合?

A

a	b	c	d
1	0	1	1
1	0	0	1
0	0	1	1

判断一个元素
属于哪个集合



O(n)

并查集

A

- 一些应用问题涉及将n个不同的元素分成一组不相交的集合。
- 经常需要进行两种操作：
  ① 求两个集合的并集。
  ② 查询某个元素所属的集合。
- 将维护该不相交集合的数据结构称为并查集。
- 选择集合中的某个元素代表整个集合，称为集合的代表元。

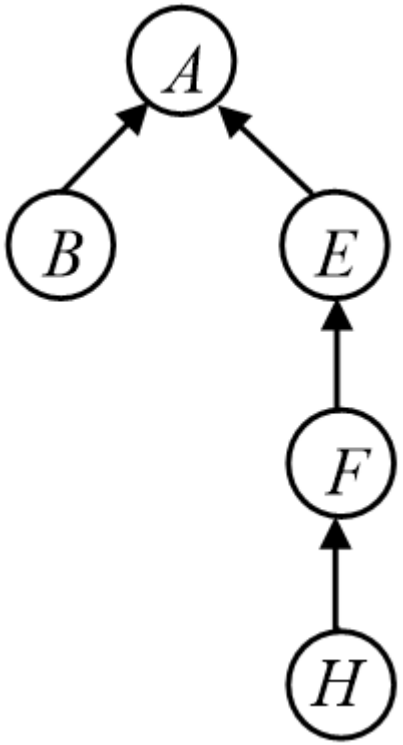
并查集的实现

设x、y表示集合中的元素，并查集的三个操作。

- **MAKE_SET(x)**：建立一个新的集合，它的唯一元素是x，因而x是代表元。
- **UNION(x, y)**：将元素x和y所在集合合并成一个集合。
- **FIND(x)**：找x所在的集合，返回该集合的代表元。

并查集的实现

- 并查集的一种高效实现方式是使用树表示集合。



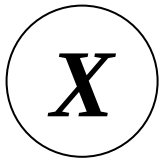
树	⇔	集合
树的结点	⇔	集合的元素
树的根结点	⇔	集合的代表元

并查集的实现

设x、y表示集合中的元素，并查集的三个操作

MAKE_SET(x)：建立一个新的集合，其唯一元素是x

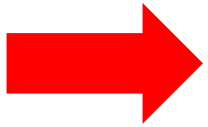
为x生成一棵单结点树，x为根结点。



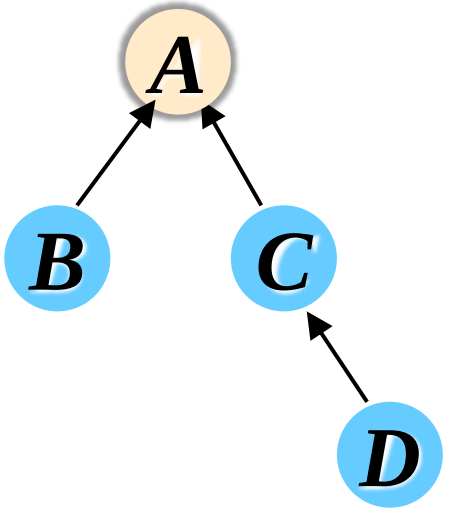
并查集的实现

设x、y表示集合中的元素，并查集的三个操作

FIND(x)：返回x所在集合的代表元



查找x所在的树的根结点。



只需要树的向上访问能力，只需存储每个结点的父结点信息。

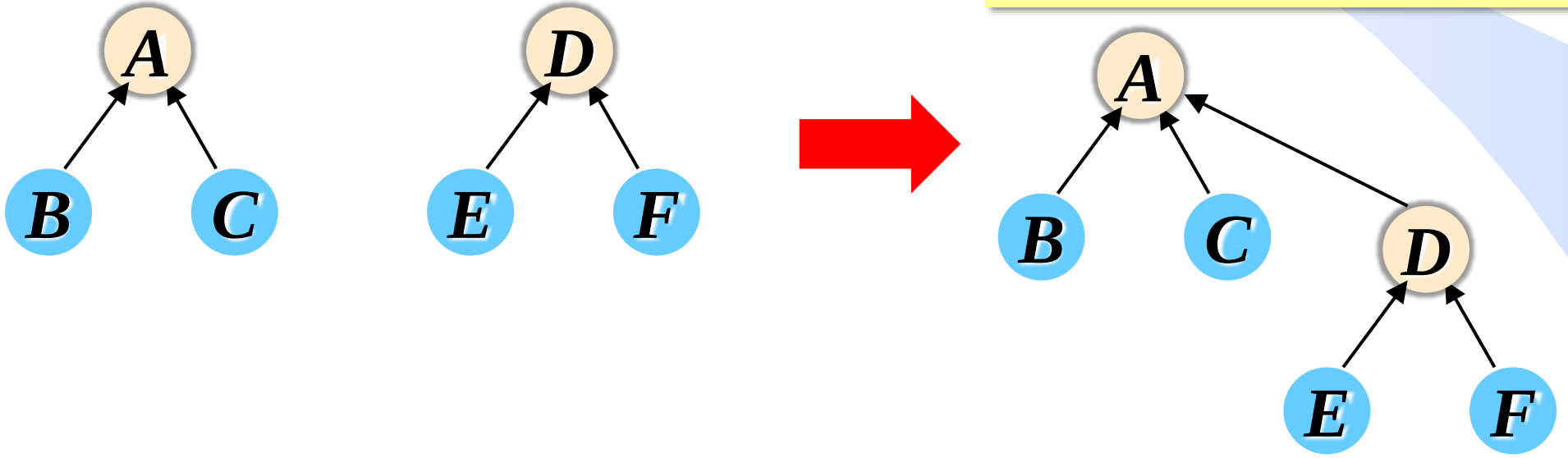
并查集的实现

A

设x、y表示集合中的元素，并查集的三个操作

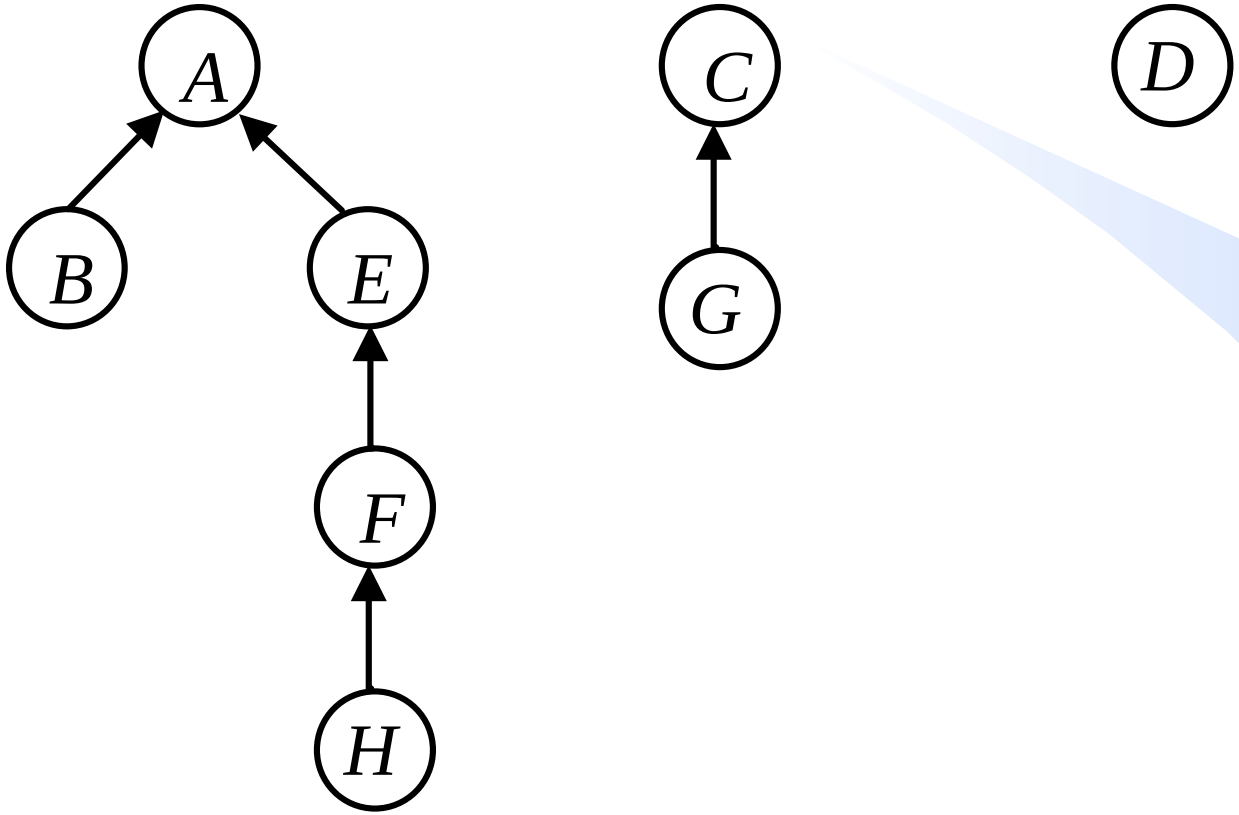
UNION(x, y)：将x和y所在的集合合并成一个集合

合并x所在的树和y所在的树，让其中一棵树的根的父指针指向另一棵树的根。



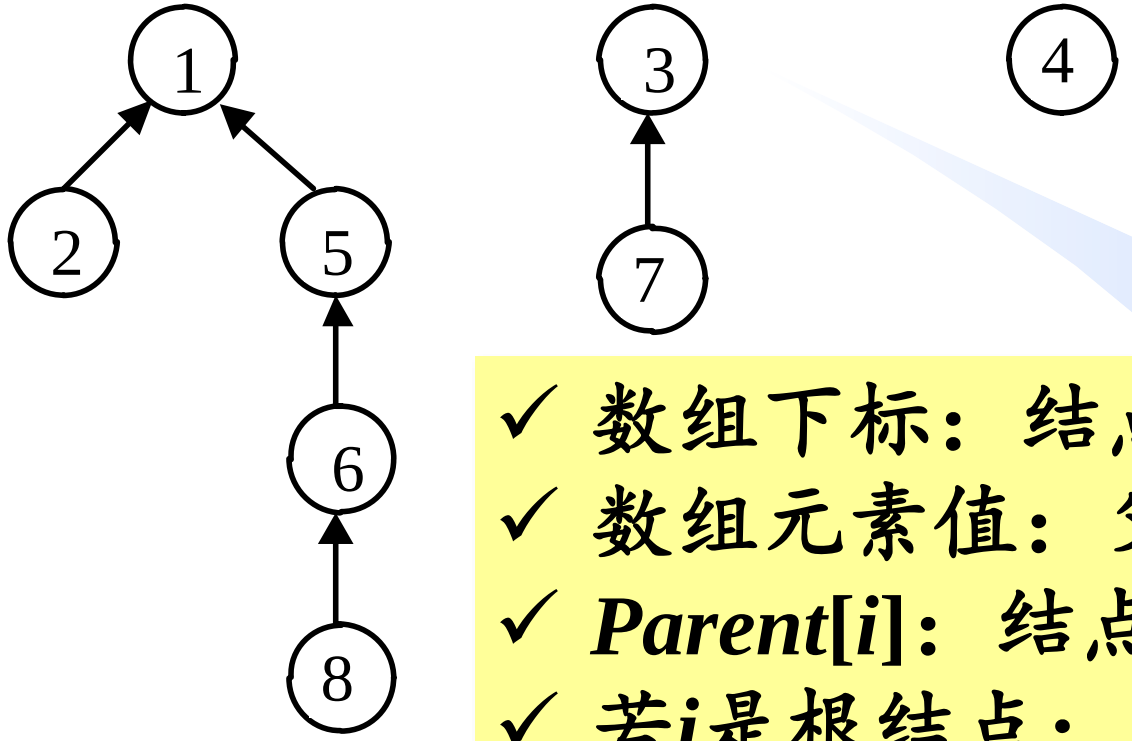
只需要树的向上访问能力，只需存储每个结点的父结点信息。

用*Parent*数组表示集合



A

用*Parent*数组表示集合



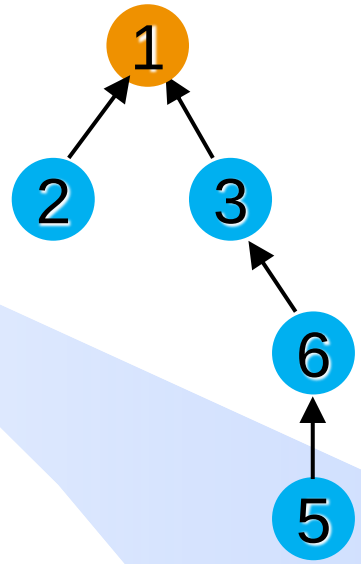
- ✓ 数组下标：结点编号
- ✓ 数组元素值：父结点下标
- ✓ *Parent*[*i*]：结点*i*的父结点
- ✓ 若*i*是根结点： *Parent*[*i*] = -1

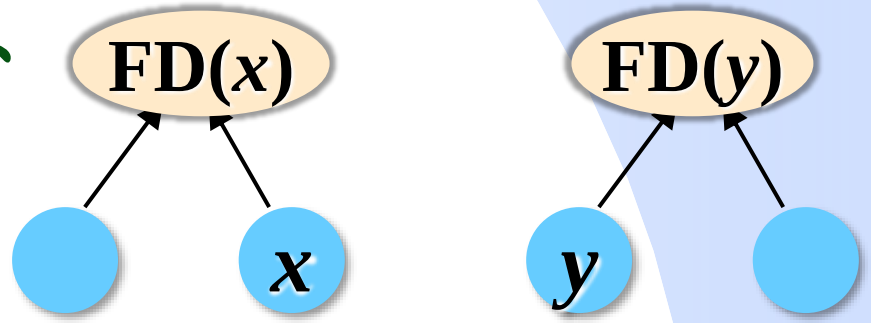
数组下标（结点编号）：

	1	2	3	4	5	6	7	8
Parent	-1	1	-1	-1	1	5	3	6

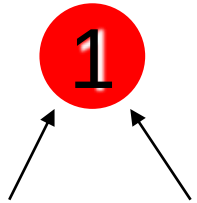
```
void MS(int x){ //MAKE_SET:为元素x生成一棵单结点树
    Parent[x] = -1;
}
int FD(int x){ //FIND:查找x所在树的根结点
    if(Parent[x] == -1) return x;
    return FD(Parent[x]);
}
void UN(int x, int y){ //UNION:合并x和y的树
    //y所在树的根结点指向x所在树根结点
    Parent[FD(y)] = FD(x);
}
```

A





例



2　3

6　5

✓合并1和2
✓合并1和3
✓合并5和6

最坏情况
合并12
合并31
合并53
合并65

FD(x): O(n)
UN(x, y): O(n)

树太高了，退化成一条链

```
void UN(int x, int y){
    Parent[FD(y)]=FD(x);
}
```

FIND和UNION操作时间复杂度取决于树的高度。

A

并查集的按秩合并和路径压缩策略

John Hopcroft
图灵奖获得者
康奈尔大学教授
美国科学院院士
美国工程院院士
中国科学院外籍院士

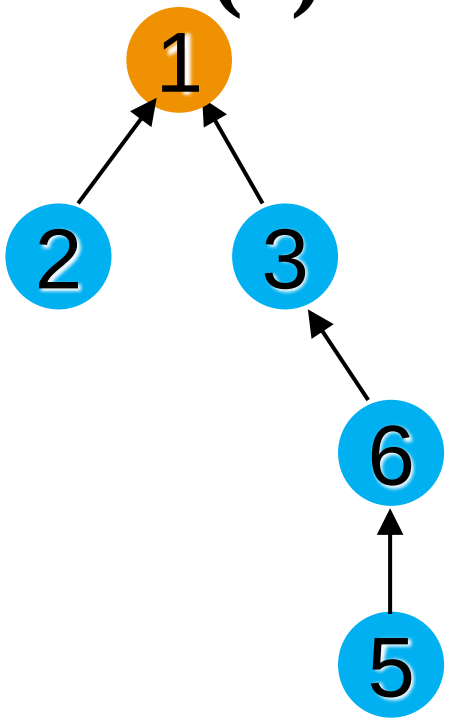
Jeffrey Ullman
图灵奖获得者
斯坦福大学教授
美国科学院院士
美国工程院院士

并查集的优化：策略1 —— 路径压缩

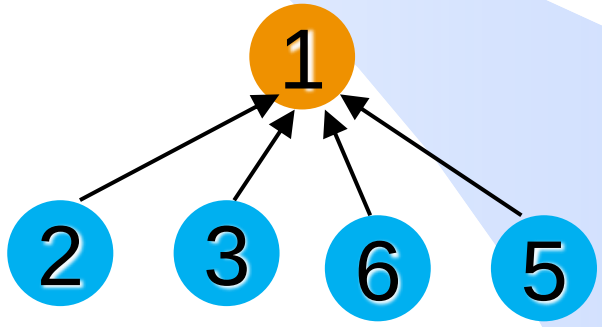
A

在FIND(x)操作中，找到元素x所在树的根fx之后，将x到根fx路径上的所有结点的父亲都改成fx .

例：FIND(5)

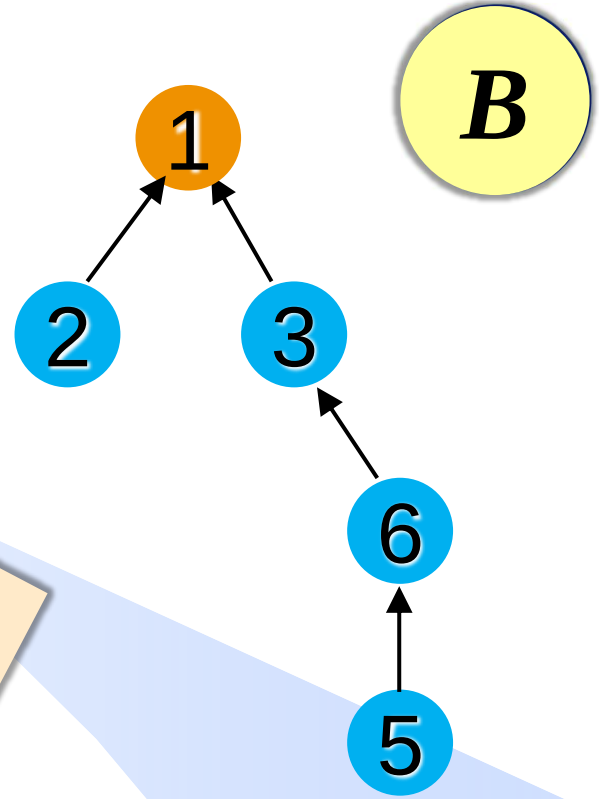


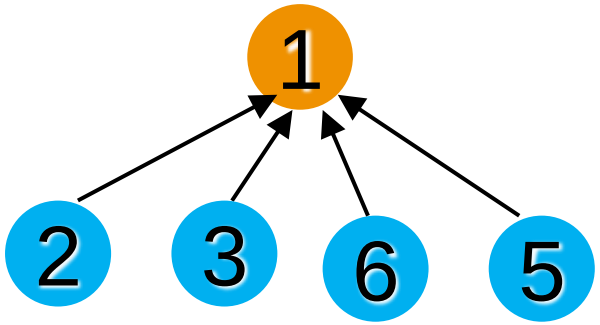
FIND(5)



路径压缩
把一棵很高的树变矮

```
int FIND(int x){ //查找元素x所在集合
    if(Parent[x]==-1) return x; //x是根
    //递归查找根结点，同时路径压缩
    Parent[x]=FIND(Parent[x]); //路径压缩
    return Parent[x];
}
```

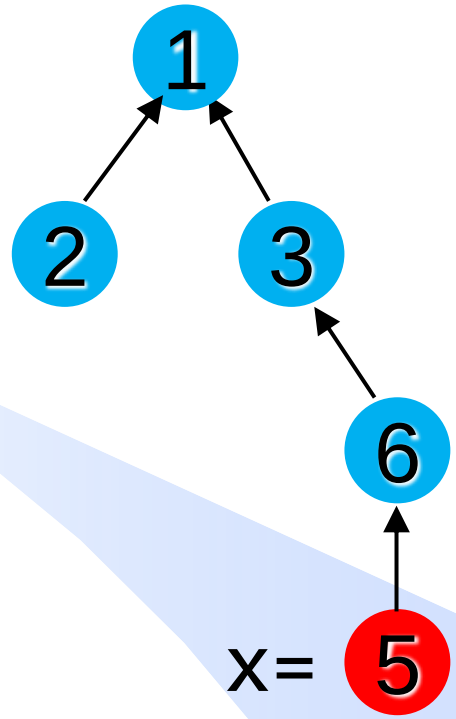




该过程是一个两趟方法，递归时沿着查找路径向上，直到找到根，递归返回时，从根向下顺带更新每个结点的父指针，使其指向根。

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```

```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

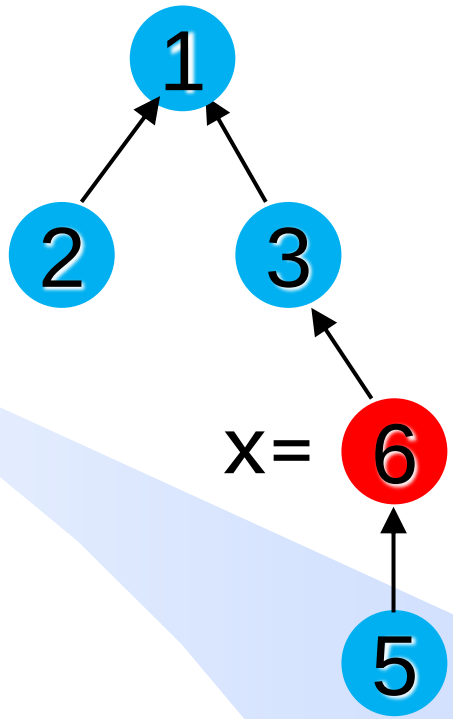


B

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```

```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

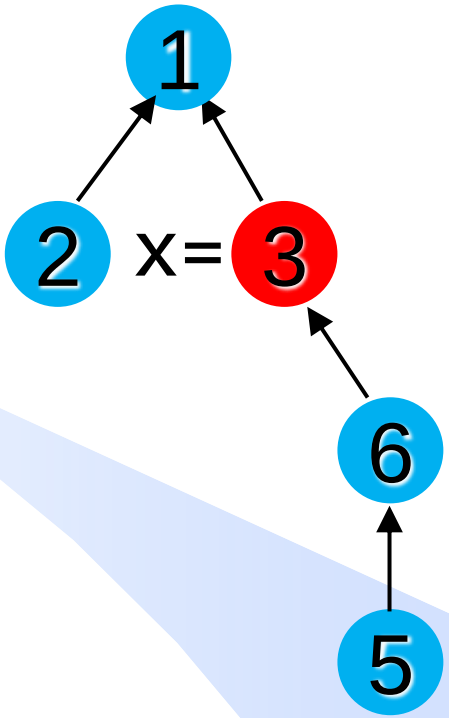
```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```



B

B

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```



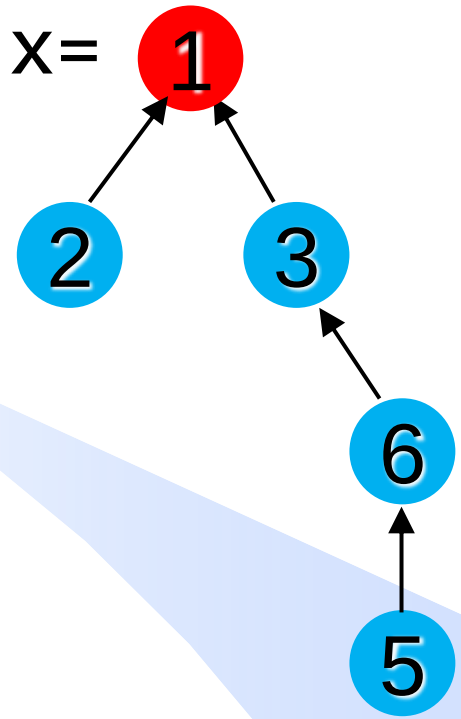

```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```

```
int FIND(3){
  ……
  Parent[3]=FIND(1);
  return Parent[3];
}
```

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```

B




```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

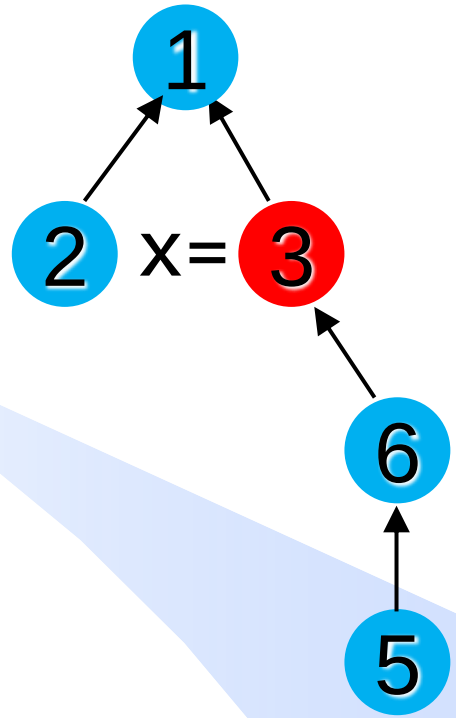
```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```

```
int FIND(3){
  ……
  Parent[3]=FIND(1);
  return Parent[3];
}
```

```
int FIND(1){
  if(Parent[1]==-1)
     return 1;
  ……
}
```

B

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```




```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

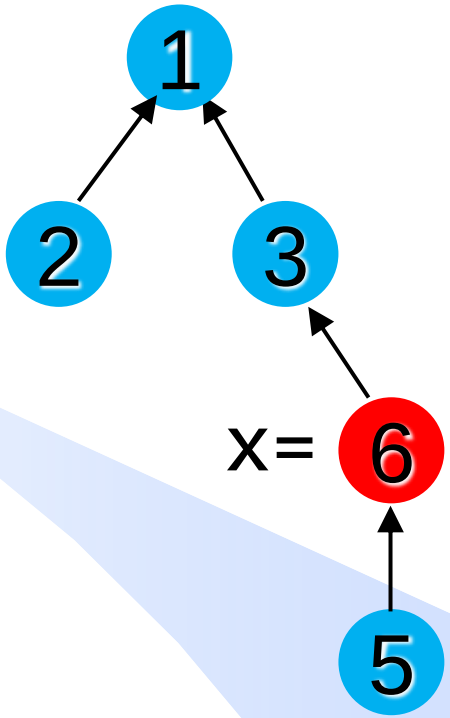
```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```

```
int FIND(3){
  ……
  Parent[3]=FIND(1);
  return Parent[3];
}
```

```
int FIND(1){
  if(Parent[1]==-1)
     return 1;
  ……
}
```

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```

B




```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

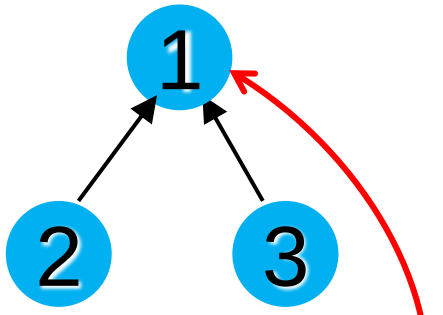
```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```

```
int FIND(3){
  ……
  Parent[3]=FIND(1);
  return Parent[3];
}
```

```
int FIND(1){
  if(Parent[1]==-1)
     return 1;
  ……
}
```

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```

B



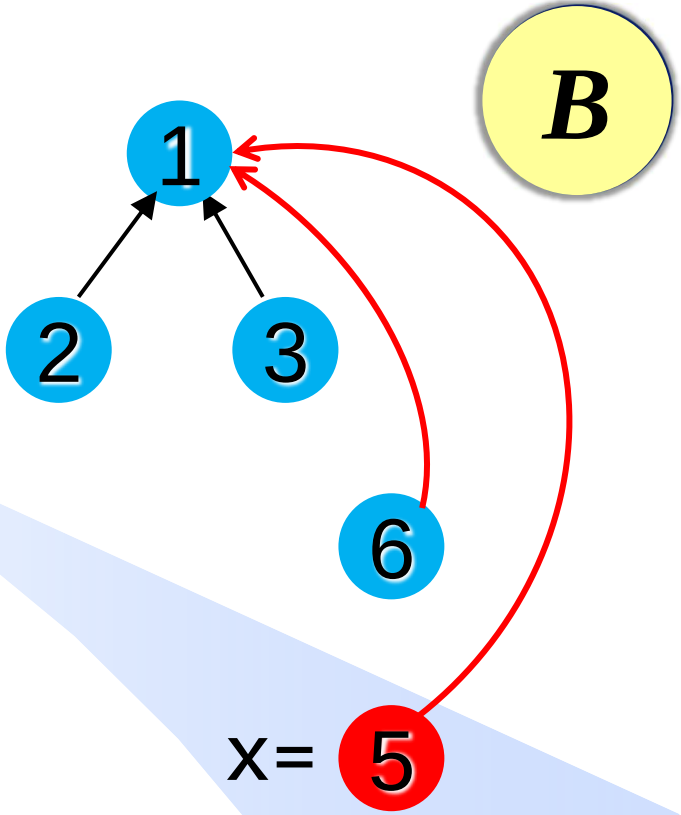

```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```

```
int FIND(3){
  ……
  Parent[3]=FIND(1);
  return Parent[3];
}
```

```
int FIND(1){
  if(Parent[1]==-1)
     return 1;
  ……
}
```

```
int FIND(int x){
    if(Parent[x]==-1) return x;
    Parent[x]=FIND(Parent[x]);
    return Parent[x];
}
```




```
int FIND(5){
  ……
  Parent[5]=FIND(6);
  return Parent[5];
}
```

```
int FIND(6){
  ……
  Parent[6]=FIND(3);
  return Parent[6];
}
```

```
int FIND(3){
  ……
  Parent[3]=FIND(1);
  return Parent[3];
}
```

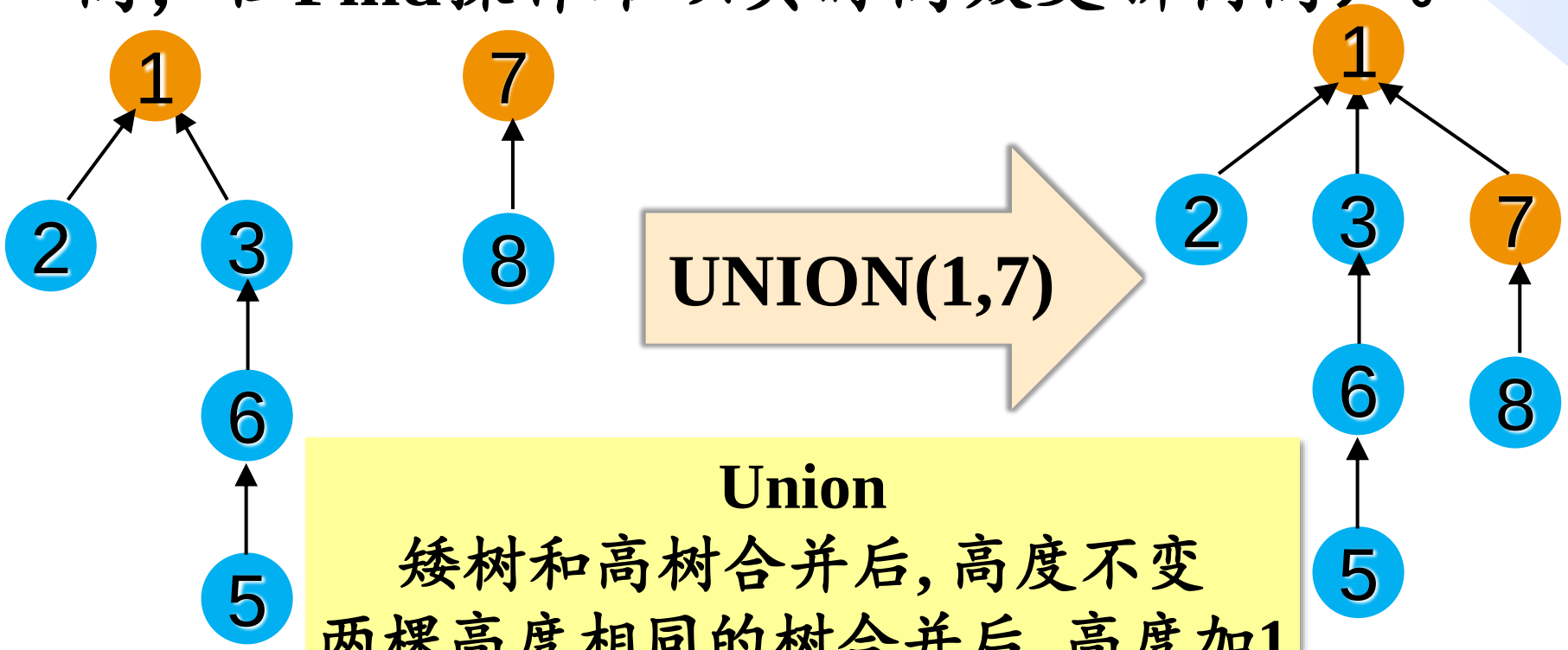
```
int FIND(1){
  if(Parent[1]==-1)
    return 1;
  ……
}
```

通过递归沿着查找路径向上，直到找到根。通过递归的返回过程，从根向下更新每个结点的父指针，使其指向根。

并查集的优化：策略2 —— 按“~~高度~~”合并？

B

- 对每个结点，记录以该结点为根的子树的高度。
- 合并操作：高树的根作为矮树的根的父亲。
- 树高变化时，高度应实时更新（Union操作中可实时更新树高，但Find操作难以实时高效更新树高）。



Union

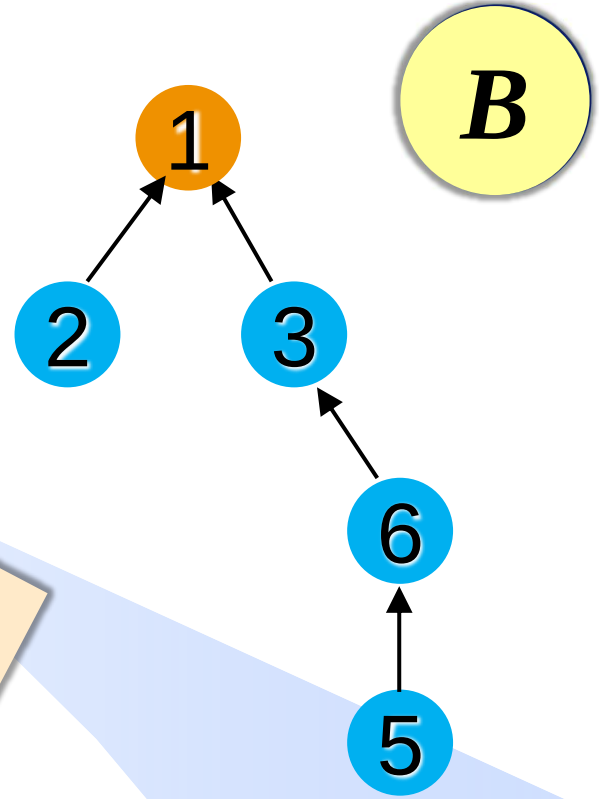
矮树和高树合并后，高度不变

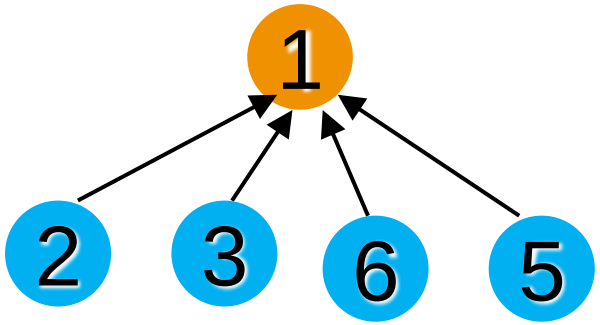
两棵高度相同的树合并后，高度加1

Find

路径压缩后，树高可能减小或不变，很难高效、精确的更新

```
int FIND(int x){ //查找元素x所在集合
    if(Parent[x]==-1) return x; //x是根
    //递归查找根结点，同时路径压缩
    Parent[x]=FIND(Parent[x]); //路径压缩
    return Parent[x];
}
```



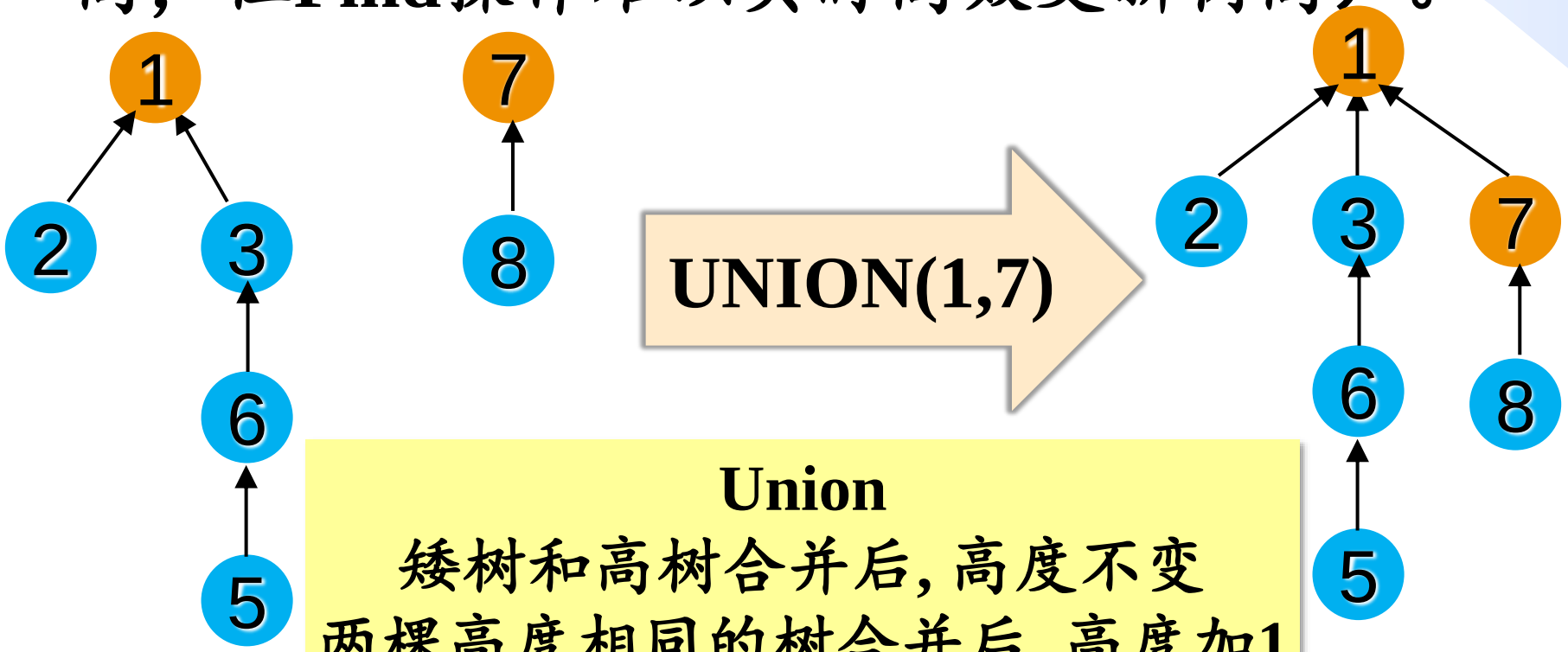


该过程是一个两趟方法，递归时沿着查找路径向上，直到找到根，递归返回时，从根向下顺带更新每个结点的父指针，使其指向根。

并查集的优化：策略2 —— 按“~~高度~~”合并？

B

- 对每个结点，记录以该结点为根的子树的高度。
- 合并操作：高树的根作为矮树的根的父亲。
- 树高变化时，高度应实时更新（Union操作中可实时更新树高，但Find操作难以实时高效更新树高）。



Union

矮树和高树合并后，高度不变

两棵高度相同的树合并后，高度加1

Find

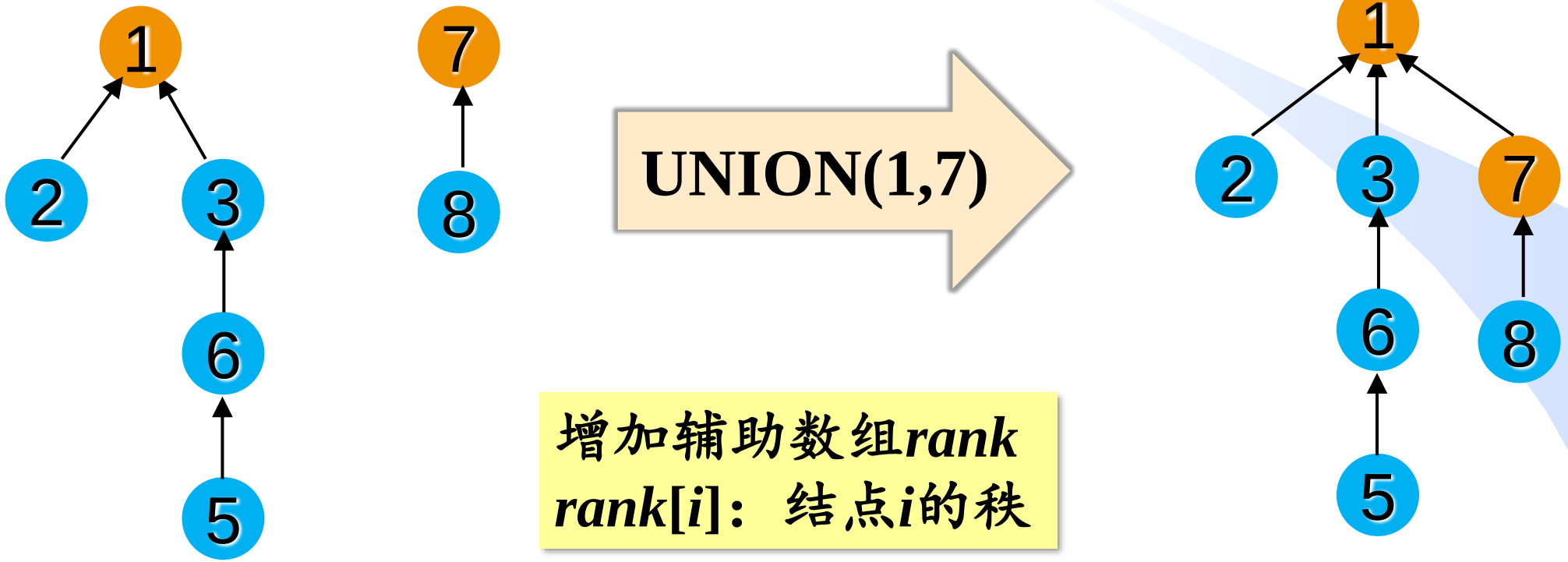
路径压缩后，树高可能减小或不变，很难高效、精确的更新

- ✓ 树高增加时可以更新树高
- ✓ 树高减小时很难更新树高

并查集的优化：策略2 —— 按秩合并

A

- 每个结点维护一个秩，表示以该结点为根的子树的高度的上界。
- 合并操作：秩大的根作为秩小的根的父亲。



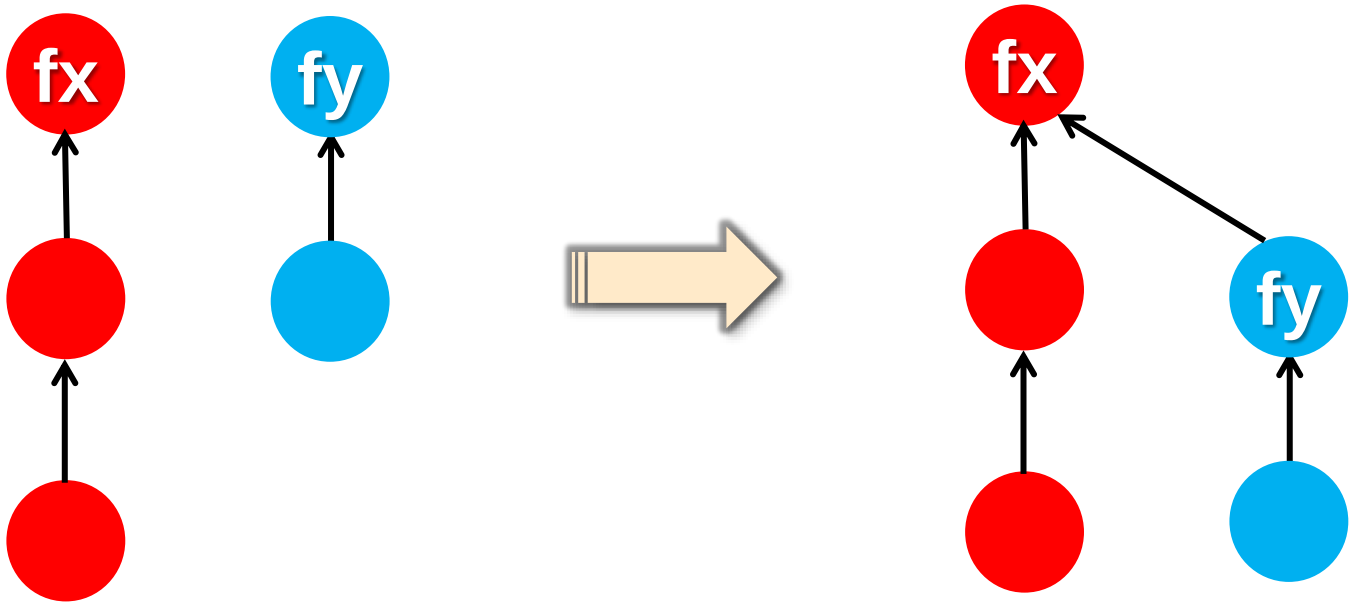
增加辅助数组*rank*

rank[*i*]：结点*i*的秩

树高增加时（可能出现在Union操作时）更新秩、秩增加

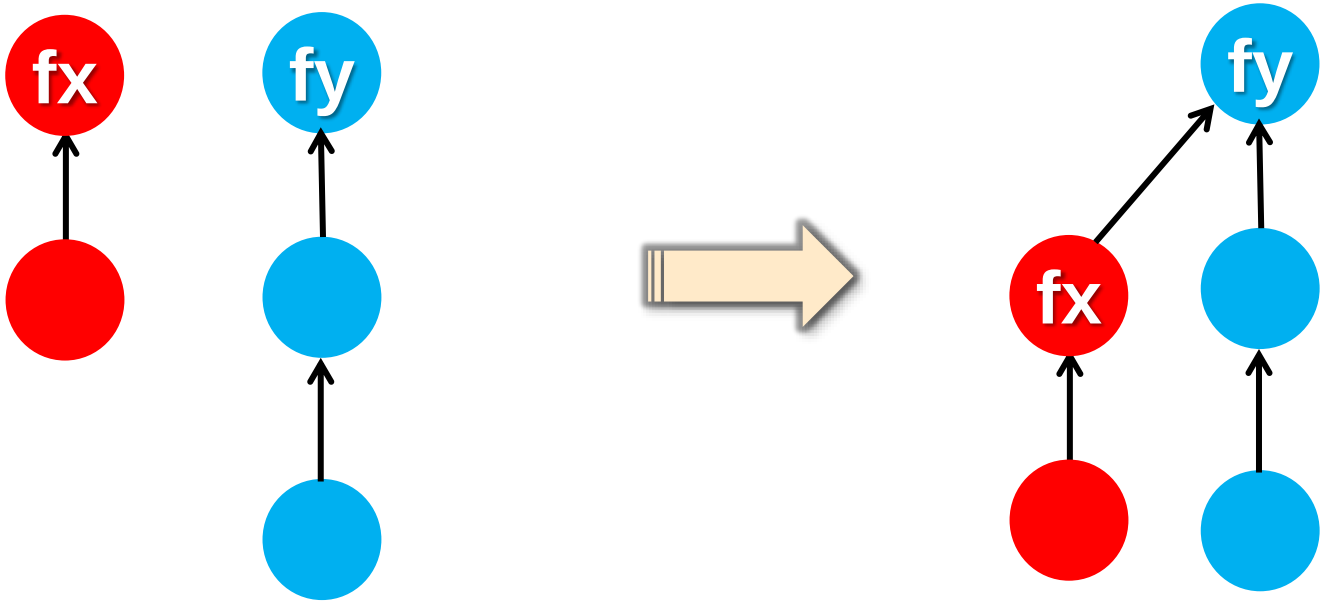
树高减小时（可能出现在Find操作路径压缩时）不更新秩、秩不变

```
void UNION(int x, int y){                  //按秩合并
    int fx=FIND(x), fy=FIND(y);            //取x和y所在树的根
    if(fx==fy) return;                     //x和y在同一棵树
    if(rank[fx]>rank[fy])                  //fy秩小
        Parent[fy]=fx;                     //fx作为fy的父结点
```



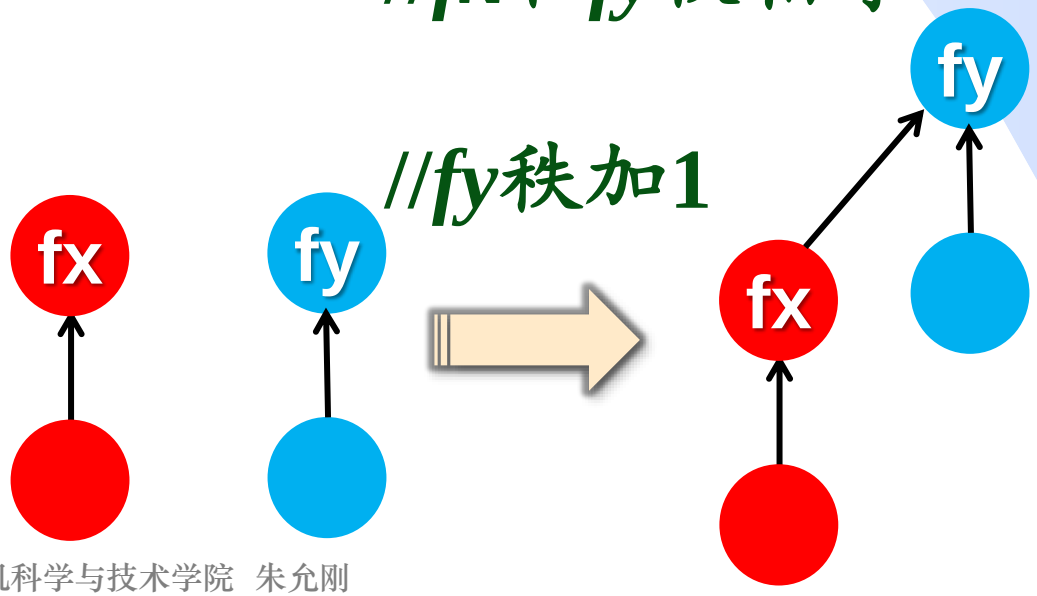
```
void UNION(int x, int y){                  //按秩合并
    int fx=FIND(x), fy=FIND(y);            //取x和y所在树的根
    if(fx==fy) return;                     //x和y在同一棵树
    if(rank[fx]>rank[fy])                  //fy秩小
        Parent[fy]=fx;                     //fx作为fy的父结点
    else if(rank[fx]<rank[fy])             //fx秩小
        Parent[fx]=fy;                     //fy作为fx的父结点
```

B



```
void UNION(int x, int y){                  //按秩合并
    int fx=FIND(x), fy=FIND(y);            //取x和y所在树的根
    if(fx==fy) return;                     //x和y在同一棵树
    if(rank[fx]>rank[fy])                  //fy秩小
        Parent[fy]=fx;                     //fx作为fy的父结点
    else if(rank[fx]<rank[fy])             //fx秩小
        Parent[fx]=fy;                     //fy作为fx的父结点
    else{                                  //fx和fy秩相等
        Parent[fx]=fy;
        rank[fy]++;                        //fy秩加1
    }
}
```

B



B

```
void Make_Set(int x){     //最终版本
    Parent[x] = -1;
    rank[x] = 0;          //根结点的秩为0
}
```

时间复杂度

A

- 单独使用按秩合并或路径压缩，Union和Find操作的均摊时间复杂度为O(log*n*).
- 定理：一组*m*个Make_Set、Union和Find操作的序列，其中Make_Set操作的个数为*n*，在不相交集合上同时使用按秩合并与路径压缩，最坏情况时间复杂度为$O(m\ \alpha(n))$.
- 上述定理表明：同时使用按秩合并和路径压缩两种优化策略，每个操作的均摊时间复杂度接近O(1).

并查集的应用—等价性问题

B

- 等价关系：反身性、对称性、传递性 。
  如：相等关系、亲戚关系
- 等价性问题：给定集合S及其上的若干等价元素对，询问S上的两个元素是否等价。

例：S = {1, 2, 3, 4, 5, 6, 7, 8}

输入：1~2, 1~5, 5~6, 3~7, 6~8

询问：1和6是否等价？ 7和8是否等价？

等价类：{1, 2, 5, 6, 8}, {3, 7}, {4}

- 等价性问题的关键在于等价类的维护，这正是并查集的一个典型应用。

一个等价类 ⇔ 一个并查集

并查集的应用—等价性问题

B

- 初始每个元素生成一个集合，形成若干不相交集合；

```
for(int i=1;i<=n;i++) Make_Set(i);
```

- 输入一对等价元素*x*和*y*，就合并*x*和*y*所在的集合；

```
Union(x,y);
```

- 判断两个元素*x*和*y*是否等价： *x*和*y*是否在同一集合里。

```
Find(x)==Find(y)
```

并查集的应用—等价性问题

B

```
void EquivalenceClass(int n, int m, int q){
     //等价性问题，n为元素个数，m为等价元素对数，q为查询数
     int x,y;
     for(int i=1; i<=n ;i++) Make_Set(i);  //初始化并查集
     while(m--){              //处理等价关系
          scanf("%d %d", &x, &y);      //读入一对等价元素
          Union(x,y);                  //合并x、y所在的集合
     }
     while(q--){              //处理查询
          scanf("%d %d", &x, &y);      //读入一个查询
          if(Find(x)==Find(y))  printf("YES\n");
          else printf("NO\n");
     }
}
```

课下思考

B

有n个城市，其中一些彼此相连，另一些没有相连。如果城市a与城市b直接相连，且城市 b 与城市 c 直接相连，那么城市a与城市c间接相连。“省份”是一组直接或间接相连的城市。给你一个$n\times n$的矩阵C，其中$C[i][j]=1$表示第i个城市和第j个城市直接相连，而$C[i][j]=0$表示二者不直接相连。返回矩阵中“省份”的数量。【华为、字节跳动、滴滴、小米、谷歌、苹果、亚马逊面试题】

省份⇔等价类⇔并查集

课下阅读

B

```
#define maxn 210
int findProvinceCount(int C[maxn][maxn], int n){
    //假设C从下标1开始存储数据
    for(int i=1; i<=n; i++) Make_Set(i);
    for(int i=1; i<=n; i++)
        for(int j=i+1; j<=n; j++)
            if(C[i][j] == 1) Union(i,j); //i和j相连
    int cnt = 0;                         //cnt为集合数目
    for(int i=1; i<=n; i++)//看有多少个集合
        if(Parent[i]<0)    //看有多少个根结点
            cnt++;
    return cnt;
}
```

课下思考

B

操场上有很多同学在玩耍，体育老师要对他们排队。初始时操场上有*n*位同学，自成一列。每次操作，老师的指令是“*x y*”，表示 学生*x* 所在的队列排到 学生*y* 所在的队列的后面，即 *x* 所在队列的队首排在 *y* 所在队列的队尾的后面。（如果 *x* 与 *y* 已经在同一队列，则忽略该指令）。现给定若干指令，请输出执行所有指令后，每位同学所在队列的队首。【北京大学上机练习题，吉林大学上机考试题】

- 一个队列 ⇔ 一个集合
- 指令“*x y*”，*x*所在集合与y所在集合合并Union(*x*, *y*)，注意题目规定了合并次序，故不能用按秩合并。
- 队首：根

自愿性质OJ练习题

- ✓ HDU 1232（并查集）
- ✓ POJ 3253（哈夫曼算法求WPL）
- ✓ POJ 1308（树基础）
- ✓ HDU 1213（并查集）
- ✓ POJ 1521（哈夫曼编码）
- ✓ UVA 12676（哈夫曼编码）