

0. 绪论

0.1 数据结构基本概念

0.1.1 基本概念和术语

术语	定义
数据	数据是信息的载体，是描述客观事物属性的数、字符及所有能输入到计算机中并被计算机程序识别和处理的符号的集合
数据元素	数据元素是数据的基本单位，通常作为一个整体进行考虑和处理，含有多个数据项
数据项	是构成数据元素的不可分割的最小单位
数据对象	具有相同性质的数据元素的集合，是数据的一个子集
数据类型	一个值的集合和定义在此集合上的一组操作的总称 {原子类型、结构类型、抽象数据类型}
数据结构	相互之间存在一种或多种特定关系的数据元素的集合

数据类型

数据类型	定义
原子类型	其值不可再分的数据类型
结构类型	其值可以再分解为若干成分的数据类型
抽象数据类型 ADT	抽象数据组织及与之相关的操作

抽象数据类型的定义格式

```
ADT 抽象数据类型名{ //抽象数据类型定义格式
    数据对象:<数据对象的定义>          //自然语言
    数据关系:<数据关系的定义>          //自然语言
    基本操作:<基本操作的定义>
}ADT 抽象数据类型名;

//基本操作的定义格式
基本操作名(参数表)      //参数表中赋值参数只提供输入值，引用参数以&打头，可提供输入值和返回操作结果
    初始条件:<初始条件描述>
    操作结果:<操作结果描述>
```

0.1.2 数据结构三要素

逻辑结构

定义：逻辑结构是指数据元素之间的逻辑关系，即从逻辑关系上描述数据。

分类：

- 线性结构：一般线性表、受限线性表（栈和队列）、线性表推广（数组）
- 非线性结构：集合结构、树结构、图结构

存储结构

定义：存储结构是指数据结构在计算机中的表示，也称物理结构

分类：

存储结构	定义	优点	缺点
顺序存储	把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来体现	随机存取，占用空间少	使用一整块相邻的存储单元，产生较多碎片
链式存储	不要求逻辑上相邻的元素在物理位置上也相邻，借助指示元素存储地址的指针来表示元素之间的逻辑关系	不会出现碎片，充分利用所有存储单元	需要额外空间，只能顺序存取
索引存储	在存储元素信息的同时，还建立附加的索引表。	检索速度快	附加的索引表需要额外空间。增删数据修改索引表时花费时间
散列存储	根据元素的关键字直接计算出该元素的存储地址，又称哈希（Hash）存储。	检索、增加和删除结点的操作很快	可能出现元素存储单元的冲突，解决冲突会增加时间和空间开销

数据的运算

定义：施加在数据上的运算包括运算的定义和实现。

- 定义是针对逻辑结构的，指出运算的功能；
- 运算的实现是针对存储结构的，指出运算的具体操作步骤。

0.2 算法和算法评价

0.2.1 算法的定义、特性和评价标准

定义：算法是针对特定问题求解步骤的一种描述，它是指令的有限序列，其中的每条指令表示一个或多个操作。

特性：

- 输入，零个或多个
- 输出，一个或多个
- 确定性，每条指令含义确定，相同的输入得出相同的结果
- 有穷性
- 可行性，所有操作可以通过已经实现的基础运算操作有限次来实现

评价标准：

- 正确性：正确结果
- 可读性
- 健壮性：输入数据非法时，能够适当的作出反应或相应处理，不会产生莫名其妙的输出结果
- 高效性：时间和空间

0.2.2 算法效率的度量

- 算法效率分为时间效率和空间效率
- 时间复杂度定义: $T(n) = O(f(n))$
- 空间复杂度定义: $S(n) = O(g(n))$
- 函数渐近的界 $O(g(n))$, 存在正数 c 和 n_0 使得对于一切 $n \geq n_0$, $0 \leq f(n) \leq cg(n)$

注意:

- 原地工作的算法的空间复杂度为 $O(1)$;
- 算法的时间复杂度不仅仅依赖于数据的规模, 也取决于待输入数据的性质, 如数据的初始状态;

算法复杂度分析步骤

- 确定表示输入规模的参数
- 找出算法的基本操作
- 检查基本操作的执行次数是否只依赖于输入规模。这决定是否需要考虑最差、平均以及最优情况下的复杂性
- 对于非递归算法, 建立算法基本操作执行次数的求和表达式;
- 对于递归算法, 建立算法基本操作执行次数的递推关系及其初始条件, 利用求和公式和法则建立一个操作次数的闭合公式, 或者求解递推公式, 确定增长的阶

加法法则:

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

乘法法则:

$$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n)) \times O(g(n))$$

常见的复杂度:

$$\begin{aligned} O(1) &\leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \\ &\leq O(n^3) \leq O(2^n) \leq O(n!) \leq O(n^n) \end{aligned}$$

两类递归算法问题的复杂度求解:

- 线性分解

$$T(n) = \begin{cases} O(1) & n = 1 \\ aT(n-1) + f(n) & n > 1 \end{cases}$$
$$T(n) = a^{n-1}T(1) + \sum_{i=2}^n a^{n-i}f(i)$$

- 指数分解

$$T(n) = \begin{cases} O(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$
$$T(n) = n^{\log_b a}T(1) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$$

1. 线性表

1.1 线性表的定义和基本操作

定义

线性表是具有相同数据类型的 n 个数据元素的有限序列。其中 n 为表长，当 $n=0$ 时线性表是一个空表。若用 L 命名线性表，则其一般表示为 $L = (a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$ 。

特点

- a_1 是唯一的“第一个”数据元素，又称表头元素
- a_n 是唯一的“最后一个”数据元素，又称表尾元素
- 除第一个元素外，每个元素有且仅有一个直接前驱。除最后一个元素外，每个元素有且仅有一个直接后继。

基本操作

```
InitList(&L);      // 初始化表：构造一个空的线性表L，分配内存空间  
DestoryList(&L);  // 销毁操作：销毁线性表，并释放线性表L所占用的内存空间  
  
ListInsert(&L, i, e); // 插入操作：在表L中第i个位置上插入指定元素e  
ListDelete(&L, i, &e); // 删除操作：删除表L中第i个位置的元素，/*并用e返回删除元素的值*/  
  
LocateElem(L, e); // 按值查找操作  
GetElem(L, i);    // 按位查找操作  
  
// 其它常用操作  
Length(L);       // 求表长  
Print(L);         // 输出操作  
Empty(L);        // 判空操作
```

1.2 线性表的顺序表示

顺序表的定义

线性表的顺序存储又称顺序表。它是用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。顺序表的特点是表中元素的逻辑顺序与物理顺序相同。

- 线性表 A 中第 i 个元素的内存地址： $\&(A[0])+i*\text{sizeof}(\text{ElemType})$
- 一维数组可以是静态分配，也可以动态分配
- 静态分配时，数组的大小和空间事先已经固定，一旦空间占满，再加入新的数据就会产生溢出，进而导致程序崩溃
- 动态分配时，存储数组的空间是在程序执行过程中通过动态存储分配语句分配的，一旦数据空间占满，就另外开辟一块更大的存储空间，用以替换原来的存储空间。

静态分配的实现

```
#define MaxSize 50           // 定义线性表的最大长度  
typedef struct{  
    ElemType data[MaxSize];   // 顺序表的元素  
    int length;              // 顺序表的当前长度  
} SqList;                  // 顺序表的类型定义
```

动态分配的实现

```

#define InitSize 100           //表长度的初始定义
typedef struct{
    ElemenType *data;       //指示动态分配数组的指针
    int MaxSize,length;    //数组的最大容量和当前个数
}SqList;                   //动态分配数组顺序表的类型定义
L.data = (ElemenType*)malloc(sizeof(ElemenType)*InitSize);
free(L);

//C++的初始动态分配语句
L.data = new ElemenType[InitSize];
delete L;

```

特点

- 随机访问
- 存储密度高
- 插入删除需要移动大量元素

顺序表的实现

注意算法对i的描述是第i个元素，它是以1为起点的

插入

```

//插入操作：在顺序表L的第i个(位序)上插入x
bool ListInsert(SqList &L, int i,int e){
    if(i<1||i>L.length+1)          //判断i的范围是否有效
        return false;
    if(L.length>=MaxSize)           //当存储空间已满时，不能插入
        return false;
    for(int j=L.length; j>=i; j--)
        L.data[j]=L.data[j-1];     //将第i个及后面的元素后移
    L.data[i-1]=e;                  //将e放到第i个位置
    L.length++;                     //长度+1
}

```

插入的时间复杂度：

最好情况：插到表尾，不需移动元素，循环0次， 最好时间复杂度 =O(1)

最坏情况：插到表头，移动n个元素，循环n次， 最坏时间复杂度 =O(n)

平均情况：设插入概率为p=1/n+1，则循环np+(n-1)p+...+1p=n/2， 平均时间复杂度 =O(n)

删除

```

//删除操作：删除顺序表L中第i个元素并返回其元素值
bool ListDelete(Sqlist &L, int i, int &e){
    if(i<1||i>L.length+1){           //判断i的范围是否有效
        return false;
    }else{
        e = L.data[i-1];             //将被删除的元素赋值给e
        for(int j=i; j<L.length; j++){
            L.data[j]=L.data[j-1];   //将第i个后面的元素前移
        }
        L.length--;
        return true;
    }
}

```

删除的时间复杂度：

最好情况：删除表尾，不需移动元素，循环0次，最好时间复杂度 =O(1)

最坏情况：删除表头，移动n-1个元素，循环n次，最坏时间复杂度 =O(n)

平均情况：设删除概率为p=1/n，则循环 $(n-1)p+(n-2)p+\dots+1p=(n-1)/2$ ，平均时间复杂度 =O(n)

查找

按位查找

```

//按位查找：返回顺序表中第i个元素的元素值
int GetElem(Sqlist L, int i){
    return L.data[i-1];
}

```

按位查找的时间复杂度

时间复杂度=O(1)

按值查找

```

//按值查找：返回顺序表L中第一个值为x的元素的位置
int LocateElem(Sqlist L, int e){
    for(int i=0; i<L.length; i++){
        if(L.data[i] == e)
            return i+1; //返回元素位置
    }
    return -1; //查找失败，返回-1
}

```

按值查找的时间复杂度

最好情况：目标在表头，循环1次，最好时间复杂度 =O(1)

最坏情况：目标在表尾，循环n次，最坏时间复杂度 =O(n)

平均情况：设删除概率为p=1/n，则循环 $(n-1)p+(n-2)p+\dots+1p=(n+1)/2$ ，平均时间复杂度 =O(n)

1.3 线性表的链式表示

单链表

- 结点描述：

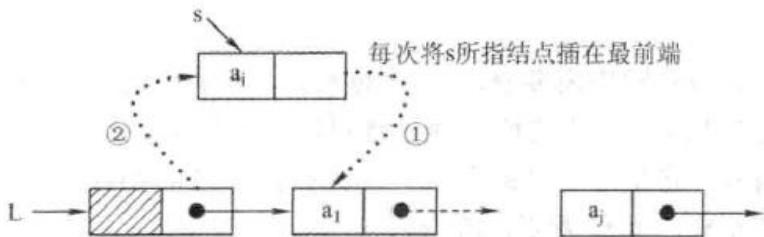
```

typedef struct LNode{           //定义单链表结点类型
    ElemType data;             //数据域
    struct LNode *next;        //指针域
}LNode, *LinkList;            //LinkList为指向结构体LNODE的指针类型

```

- 通常用头指针来标示一个单链表。
- 有头结点或者没头结点之分
- 头结点的作用
 - 便于首元结点的处理，对链表的第一个数据元素的操作与其他数据元素相同，无需特殊处理
 - 便于空表与非空表的统一处理：头指针永远不为空

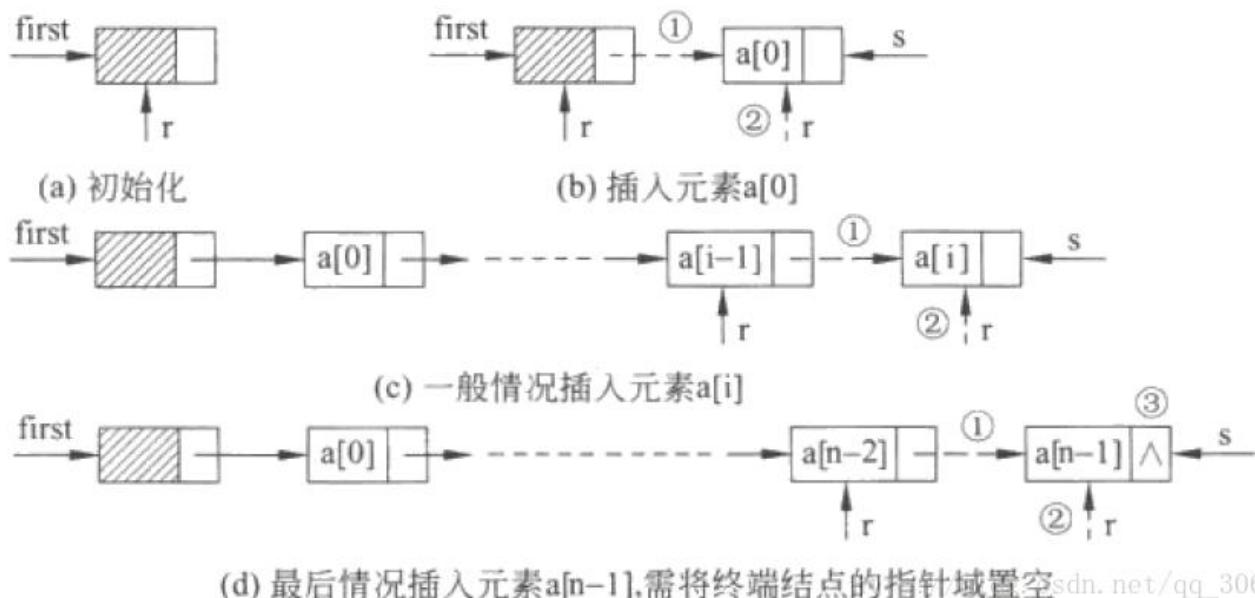
单链表的实现



```

LinkList List_HeadInsert(LinkList &L){
    LNode *s; int x;
    L=(LinkList)malloc(sizeof(LNode)); //创建头结点
    L->next = NULL;                //初始为空链表
    scanf("%d", &x);
    while(x!=9999){
        s = (LNode*)malloc(sizeof(LNode));
        s->data = x;
        s->next = L->next;
        L->next = s;
        scanf("%d", &x);
    }
    return L;
}

```



```

LinkList List_TailInsert(LinkList &L){
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    LNode *s,*r=L;           //r为表尾指针
    scanf("%d",&x);
    while(x!=9999){
        s = (LNode *)malloc(sizeof(LNode));
        s->data = x;
        r->next = s;
        r = s;
        scanf("%d",&x);
    }
    r->next = NULL;          //尾结点指针置空
    return L;
}

```

```

LNode *LocateElem(LinkList L,ElemType e){//按值查找
    LNode *p = L->next;
    while(p!=NULL&&p->data!=e)
        p = p->next;
    return p;
}

```

删除

按位序删除

```

//删除操作:将单链表中的第i个结点删除
bool Delete(LinkList &L, int i int &e){
    if(i<1 || i>Length(L))
        return false;
    LNode *p = GetElem(L,i-1); //查找第i个位置
    LNode *q = p->next;
    e = q->data;
    p->next = q->next;
    free(q);
    return true;
}

```

按位序删除的时间复杂度:

最好情况：删除第一个，不需查找位置，循环0次，最好时间复杂度 =O(1)

最坏情况：删除最后一个，需查找第n位，循环n次，最坏时间复杂度 =O(n)

平均情况：删除任意一个，平均时间复杂度 =O(n)

指定结点的删除

时间复杂度 =O(n)

方法：p的后一个为q，p指向q的下一个，把q的值给p，最后释放q

```
//删除指定结点p
bool Delete(LNode *p){
    if(p==NULL) return false;
    LNode *q = p->next;
    p->data = q->data;
    p->next = q->next;
    free(q);
    return true;
}
```

查找

按位查找

平均时间复杂度 = $O(n)$

```
//按位查找：查找在单链表L中第i个位置的结点
LNode *GetElem(LinkList L, int i){
    int j=0;
    LNode *p = L;
    if(i<0) return NULL;
    while(p && j<i){
        p = p->next;
        j++;
    }
    return p; //如果i大于表长，p=NULL,直接返回p即可
}
```

按值查找

平均时间复杂度 = $O(n)$

```
//按值查找：查找e在L中的位置
LNode *LocateElem(LinkList L, int e){
    LNode *p = L->next;
    while(p && p->data != e){
        p = p->next;
    }
    return p;
}
```

求表长

平均时间复杂度 = $O(n)$

```
//求表的长度
int Length(LinkList L){
    int len = 0;
    LNode *p = L;
    while(p->next){
        p = p->next;
        len++;
    }
    return len;
}
```

遍历

```

//遍历操作
void PrintList(LinkList L){
    LNode *p = L->next;
    while(p){
        printf("%d ", p->data);
        p = p->next;
    }
}

```

双链表

```

typedef struct DNode{
    ElemenType data;
    struct DNode *prior,*next; //前驱和后继指针
}DNode,*DLinkList;

```

循环链表

- 循环单链表

初始化和判空(与单链表不一样)

L->next = NULL改为L->next = L

- 循环双链表

初始化和判空(与双链表不一样)

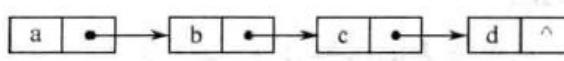
L->next = NULL改为L->next = L
L->prior = NULL改为L->prior = L

静态链表

借助数组来描述线性表的链式存储结构，结点也有数据域 data 和指针域 next，这里的指针是节点的相对地址（数组下标），又称游标

0		2
1	b	6
2	a	1
3	d	-1
4		
5		
6	c	3

(a)静态链表示例



(b)静态链表对应的单链表

```

#define MaxSize 50
typedef struct{
    ElemenType data;
    int next;
}SLinkList[MaxSize];

```

2. 栈和队列

2.1 栈

2.1.1 栈的基本概念

栈的定义

- 栈是只允许在一端进行插入或删除操作的线性表。后进先出 LIFO
- 栈顶 (Top) : 线性表允许进行插入删除的那一端。
- 栈底 (Bottom) : 固定的，不允许进行插入和删除的另一端
- 空栈: 不包含任何元素的空表

N个不同元素进栈出栈，出栈的序列个数为： $\frac{1}{n-1} C_{2n}^n = \frac{1}{n-1} \frac{(2n)!}{n! \times n!}$

栈的基本操作

InitStack(&S): 初始化一个空栈S
StackEmpty(S): 判断一个栈是否为空，若栈S为空则返回true，否则返回false
Push(&S, x): 进栈，若栈S未满，则将x加入使之成为新栈顶
Pop(&S, &x): 出栈，若栈S非空，则弹出栈顶元素，并用x返回
GetTop(S, &x): 读取栈顶元素，若栈S非空，则用x返回栈顶元素
DestroyStack(&S): 销毁栈，并释放栈S占用的存储空间

2.1.2 栈的顺序存储结构

顺序栈的实现

利用一组地址连续的存储单元存放自栈底到栈顶的数据元素，并附设一个指针 top 指示当前栈顶元素的位置

```
#define MaxSize 50           //定义栈中元素最大个数
typedef struct{
    Elemtyp data[MaxSize];   //存放栈中元素
    int top; //栈顶指针
}
```

- 栈顶指针: S.top，初始时设置 S.top=-1；栈顶元素: S.data[S.top]
- 进栈操作: 栈不满时，栈顶指针先加 1，再送值到栈顶元素
- 出栈操作: 栈非空时，先取栈顶元素值，再将栈顶指针减 1
- 栈空条件: S.top== -1；栈满条件: S.top==MaxSize-1；栈长: S.top+1

共享栈

利用栈底位置相对不变的特性，可让两个顺序栈共享一个一维数组空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶共享空间的中间延伸。

- 两个栈的栈顶指针都指向栈顶元素
- top0=-1 时 0 号栈为空，top1=MaxSize 时 1 号栈为空
- top1-top0==1 为栈满
- 当 0 号栈进栈时 top0 先加 1 再赋值，1 号栈进栈时 top1 先减 1 再赋值；出栈是刚好相反

2.1.3 栈的链式存储结构

采用链式存储的栈称为**链栈**，链栈的优点是便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况。这里规定链栈没有头结点，Lhead 指向栈顶元素

```

typedef struct Linknode{
    ElemenType data;//数据域
    struct Linknode *next;//指针域
} *LiStack;//栈类型定义

```

2.2 队列

2.2.1 队列的基本概念

队列的定义

- 队列简称队，也是一种操作受限的线性表，只允许在表的一端进行插入，而在表的另一端进行删除。
- 向队列中插入元素称为**入队或进队**
- 删除元素称为**出队或离队**
- 操作的特性是先进先出

队列常见的基本操作

```

InitQueue(&Q) : 初始化队列，构造一个空队列Q
QueueEmpty(Q) : 判队列空
EnQueue(&Q, x) : 入队，若队列Q非满，将x加入，使之成为新的队尾
DeQueue(&Q, &x) : 出队，若队列Q非空，删除队头元素，并用x返回
GetHead(Q, &x) : 读队头元素，若队列Q非空，则将队头元素赋值给x

```

2.2.2 队列的顺序存储结构

队列的顺序存储

队列的顺序实现是指分配一块连续的存储单元存放队列中的元素，并附设两个指针：队头指针 `front` 指向队头元素，队尾指针 `rear` 指向队尾元素的下一个位置

```

#define MaxSize 50//定义队列中元素的最大个数
typedef struct{
    ElemenType data[MaxSize];//存放队列元素
    int front, rear;//队头指针和队尾指针
} SqQueue;

```

- 初始状态： `Q.front==Q.rear==0`
- 进队操作：队不满时，先送值到队尾元素，再将队尾指针加 1
- 出队操作：队不空时，先取队头元素值，再将队头指针加 1

循环队列

将顺序队列臆造为一个环状的空间，即把存储队列元素的表从逻辑上视为一个环，称为循环队列。当队首指针 `Q.front=MaxSize-1` 后，再前进一个位置就自动到 0，这可以利用除法取余运算 `%` 来实现

- 初始状态： `Q.front=Q.rear=0`
- 队首指针进 1： `Q.front=(Q.front+1)%MaxSize`
- 队尾指针进 1： `Q.rear=(Q.rear+1)%MaxSize`
- 队列长度： `(Q.rear+MaxSize-Q.front)%MaxSize`
- 出队入队时：指针都按顺时针方向进 1

判断循环队列队空或队满的三种方式

1. 牺牲一个单元来区分队空和队满，入队时少用一个队列单元，约定以“队头指针在队尾指针的下一位置作为队满的标志”

- 队满条件: $(Q.\text{rear}+1)\% \text{MaxSize} == Q.\text{front}$
- 队空条件: $Q.\text{front} == Q.\text{rear}$
- 队列中元素的个数: $(Q.\text{rear}-Q.\text{front}+\text{MaxSize})\% \text{MaxSize}$

2. 类型中增设表示元素个数的数据成员。

- 队空条件: $Q.\text{size} == 0$
- 队满条件: $Q.\text{size} == \text{MaxSize}$

3. 类型中增设 tag 数据成员，以区分是队满还是队空。

- $\text{tag}=0$ 时，若因删除导致 $Q.\text{front} == Q.\text{rear}$ ，则为队空
- $\text{tag}=1$ 时，若因插入导致 $Q.\text{front} == Q.\text{rear}$ ，则为队满

2.2.3 队列的链式存储结构

队列的链式存储

队列的链式表示称为链队列，它实际是一个同时带有队头指针和队尾指针的单链表。头指针指向队头结点，尾指针指向队尾结点。

```
typedef struct{//链式队列结点
    ElemenType data;
    struct LinkNdoe *next;
}LinkNode;
typedef struct{//链式队列
    LinkNode *front,*rear;//队列的队头和队尾指针
}LinkQueue;
```

通常将链式队列设计成一个带头结点对的单链表，这样插入和删除就统一了

2.2.4 双端队列

双端队列是指允许两端都可进行入队和出队操作的队列，其元素的逻辑结构仍是线性结构。将队列的两端分别称为前端和后端。

- 输出受限的双端队列：允许在一端进行插入和删除，另一端只允许插入的双端队列
- 输入受限的双端队列：允许在一端进行插入和删除，另一端只允许删除的双端队列

2.3 栈和队列的应用

栈在括号匹配中的应用

- 初始设置一个空栈，顺序读入括号
- 若是右括号，则或者置于栈顶的最急迫期待得以消解，或者是不合法的情况
- 若是左括号，则作为一个新的更急迫的期待压入栈中
- 算法结束时，栈为空，否则括号序列不匹配

栈在表达式求值中的应用

后续表达式计算方式

顺序扫描表达式的每一项，然后根据它的类型作出如下相应操作：若该项是操作数，则将其压入栈中；若该项是操作符 $<\text{op}>$ ，则连续从栈中退出两个操作数 Y 和 X ，形成运算指令 $X<\text{op}>Y$ ，并将计算结果重新压入栈中。当表达式的所有项扫描并处理完毕后，栈顶存放的就是最后的结果

中缀表达式转换为前缀或后缀表达式的手工做法

- 按照运算符的优先级对所有的运算单位加括号
- 转换为前缀或后缀表达式。前缀把运算符移动到对应的括号前面，后缀把运算符移动到对应的括号后面
- 把括号去掉

中缀表达式转换为后缀表达式的算法思路

- 从左向右开始扫描中缀表达式
- 遇到数字时，加入后缀表达式
- 遇到运算符时
 - 若为 (，入栈
 - 若为)，则依次把栈中的运算符加入后缀表达式，直到出现 (，从栈中删除 (
 - 若为除括号外的其他运算符，当其优先级高于除 (外的栈顶运算符时，直接入栈。否则从栈顶开始，依次弹出比当前处理的运算符优先级高和优先级相等的运算符，直到一个比它优先级低的或遇到一个左括号为止。

待处理序列	栈	后缀表达式	当前扫描元素	动 作
a/b+(c*d-e*f)/g			a	a 加入后缀表达式
/b+(c*d-e*f)/g		a	/	/入栈
b+(c*d-e*f)/g	/	a	b	b 加入后缀表达式
+(c*d-e*f)/g	/	ab	+	+优先级低于栈顶的 /, 弹出 /
+(c*d-e*f)/g		ab/	+	+入栈
(c*d-e*f)/g	+	ab/	((入栈
c*d-e*f)/g	+()	ab/	c	c 加入后缀表达式
*d-e*f)/g	+()	ab/c	*	栈顶为 (, *入栈
d-e*f)/g	+(*)	ab/c	d	d 加入后缀表达式
-e*f)/g	+(*)	ab/cd	-	-优先级低于栈顶的 *, 弹出 *
-e*f)/g	+()	ab/cd*	-	栈顶为 (, -入栈
e*f)/g	+(-)	ab/cd*	e	e 加入后缀表达式
*f)/g	+(-)	ab/cd*e	*	*优先级高于栈顶的 -, *入栈
f)/g	+(-*)	ab/cd*e	f	f 加入后缀表达式
) / g	+(-*)	ab/cd*ef)	把栈中 (之前的符号加入表达式
/ g	+	ab/cd*ef*-	/	/优先级高于栈顶的 +, /入栈
g	+/	ab/cd*ef*-	g	g 加入后缀表达式
	+/	ab/cd*ef*-g		扫描完毕, 运算符依次退栈加入表达式
		ab/cd*ef*-g/+		完成

由此可知, 当扫描到 f 时, 栈中的元素依次是 + (-*, 选 B。

在此, 以上面给出的中缀表达式为例, 给出中缀表达式转换为前缀或后缀表达式的手工做法。

步骤 1: 按照运算符的优先级对所有的运算单位加括号。

式子变成 ((a/b)+(((c*d)-(e*f)) /g))。

步骤 2: 转换为前缀或后缀表达式。

前缀: 把运算符号移动到对应的括号前面, 式子变成 +(/(ab) / (-(* (cd) * (ef)) g))。

把括号去掉: +/ab/-*cd*efg 前缀式子出现。

后缀: 把运算符号移动到对应的括号后面, 式子变成 ((ab) / (((cd) * (ef) *) -g) /) +。

把括号去掉: ab/cd*ef*-g/+后缀式子出现。

栈在递归中的应用

可以将递归算法转换为非递归算法。通常需要借助栈来实现这种转换

队列在层次遍历中的应用

1. 根节点入队
2. 若队空, 则结束遍历; 否则重复 3 操作
3. 队列中第一个结点出队, 并访问之。若其没有左孩子, 则将左孩子入队, 若其有左孩子, 则将其右孩子入队, 返回 2

队列在计算机系统中的应用

- 解决主机与外部设备之间速度不匹配的问题
- 解决由多用户引起的资源竞争问题

2.4 特殊矩阵的压缩存储

数组的定义

数组是由 n 个相同类型的数据元素构成的有限序列，每个数据元素成为一个**数据元素**，每个元素在 n 个线性关系中的序号称为该元素的**下标**，下标的取值范围称为数组的**维界**

数组是线性表的推广。一维数组可视为一个线性表；二维数组可视为其元素也是定长线性表的线性表。

2.4.1 数组的存储结构

一维数组： $a[N]$

逻辑上连续存放，物理上（内存中）也连续存放

数组元素 $a[i]$ 的物理地址 = $\text{LOC} + i * \text{sizeof}(\text{ElemType})$

二维数组： $a[N][M]$

逻辑上是 n 行 m 列的矩阵，物理上（内存中）是 行优先存储 和 列优先存储 的连续存放

行优先存储： 数组元素 $a[i][j]$ 的物理地址 = $\text{LOC} + (i * N + j) * \text{sizeof}(\text{ElemType})$

列优先存储： 数组元素 $a[i][j]$ 的物理地址 = $\text{LOC} + (j * M + i) * \text{sizeof}(\text{ElemType})$

普通矩阵的存储可用二维数组存储。

2.4.2 特殊矩阵的存储

- ① 对称矩阵
- ② 三角矩阵
- ③ 三对角矩阵
- ④ 稀疏矩阵

对称矩阵的压缩存储

对称矩阵： $a_{i,j} = a_{j,i}$

方法： 一维数组 $a[N]$ 只存主对角线+下三角区（或主对角线+上三角区）

存储数组的大小： $N = \frac{n(n+1)}{2}$

数组下标范围： $0 \sim \frac{n(n+1)}{2} - 1$

行优先存储： 数组下标： $k = \begin{cases} \frac{i(i-1)}{2} + j - 1, & i \geq j \text{(下三角区和主对角线元素)} \\ \frac{j(j-1)}{2} + i - 1, & i < j \text{(上三角区元素 } a_{i,j} = a_{j,i} \text{)} \end{cases}$

三角矩阵的压缩存储

① 下三角矩阵： 除主对角线和下三角区，其余的元素都相等

方法： 一维数组 $a[N]$ 存主对角线+下三角区，在最后多加一个位置存常其余相等元素

存储数组的大小： $N = \frac{n(n+1)}{2} + 1$

数组下标范围： $0 \sim \frac{n(n+1)}{2}$

行优先存储： 数组下标： $k = \begin{cases} \frac{i(i-1)}{2} + j - 1, & i \geq j \text{(下三角区和主对角线元素)} \\ \frac{n(n+1)}{2}, & i < j \text{(上三角区元素)} \end{cases}$

② 上三角矩阵：除主对角线和上三角区，其余的元素都相等

方法：一维数组 $a[N]$ 存主对角线+下三角区，在最后多加一个位置存常其余相等元素

存储数组的大小： $N = \frac{n(n+1)}{2} + 1$

数组下标范围： $0 \sim \frac{n(n+1)}{2}$

行优先存储：数组下标： $k = \begin{cases} \frac{(i-1)(2n-i+2)}{2} + (j-i), & i \geq j (\text{下三角区和主对角线元素}) \\ \frac{n(n+1)}{2}, & i < j (\text{上三角区元素}) \end{cases}$

三对角矩阵的压缩存储

三对角矩阵，又称 带状矩阵。

方法：一维数组 $a[N]$ 存带状部分

存储数组的大小： $N = 3n - 3 + 1$

数组下标范围： $0 \sim 3n - 3$

行优先存储：

i 和 j 计算数组下标 k： $k = 2i + j - 3$

数组下标 k 计算 i 和 j： $i = \lceil (k+2)/3 \rceil, j = k - 2i + 3$

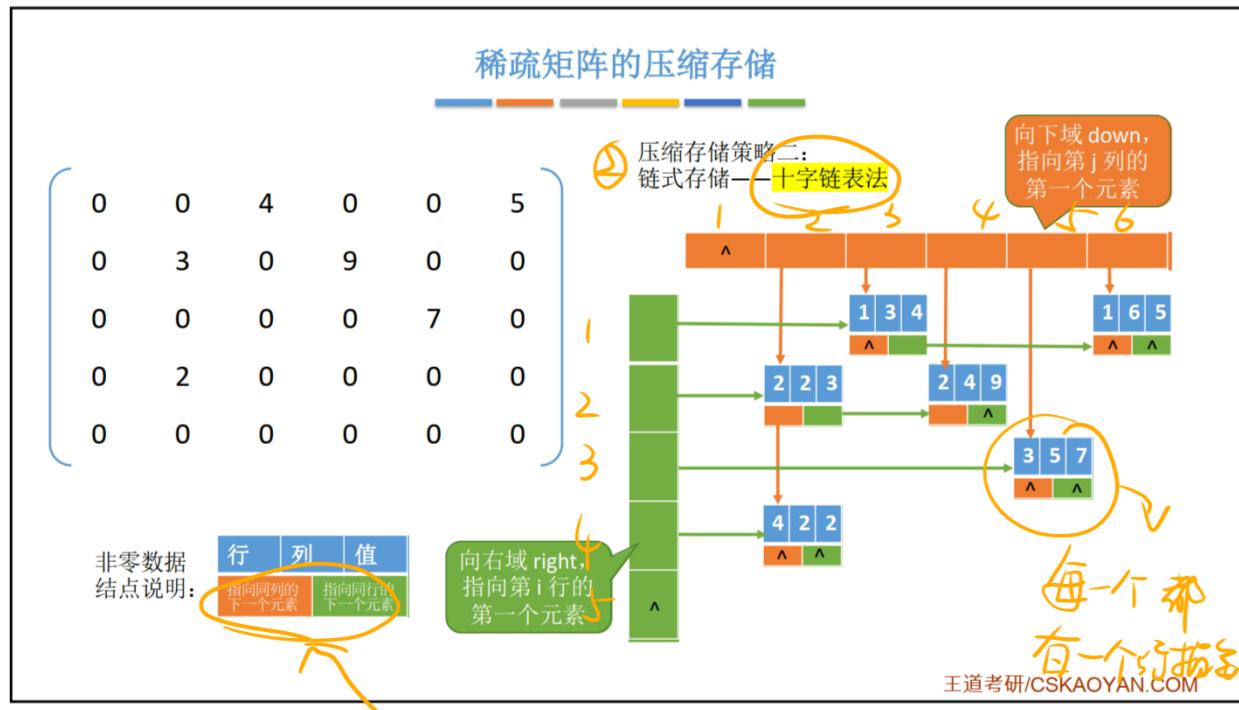
稀疏矩阵的压缩存储

稀疏矩阵：非零元素的个数远远少于矩阵元素的个数。

方法一：顺序存储—三元组<行，列，值>，注：行列从1开始

i (行)	j (列)	v (值)
1	3	4
1	6	5
2	2	3

方法二：链式存储—十字链表法



22

11

王道考研/cskaoyan.com

2.5 广义表

2.5.1 定义

- $A = ()$ ：广义表 A 是一个空表
- $B = (d, e)$ ：广义表 B 的元素全是原子 d 和 e
- $C = (b, B, (c, d))$ ：广义表 C 的元素分别是原子 b、广义表 B、广义表 (c, d)

2.5.2 广义表的长度

- $A = ()$ ：长度为 0
- $B = (d, e)$ ：长度为 2
- $C = (b, B, (c, d))$ ：长度为 3

2.5.3 广义表的深度

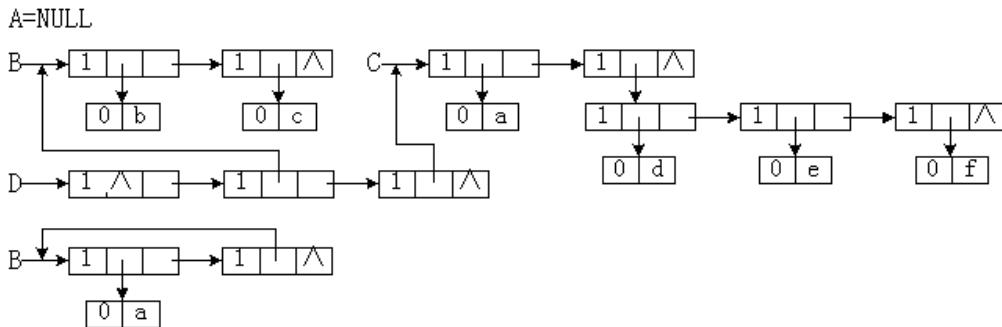
嵌套几个括号深度就是几

- $A = ()$ ：深度为 1
- $B = (d, e)$ ：深度为 1
- $C = (a, B)$ ：深度为 2
- $D = (a, (d, e))$ ：深度为 2
- $E = ((a, D), c)$ ：深度为 4

2.5.4 广义表获取表头和表尾

- $A = (a, b, c)$
 - GetHead(A) = a
 - GetTail(A) = (b, c)
- $B = ()$
 - GetHead(B) = NULL
 - GetTail(B) = ()
- $C = (a, b)$
 - GetHead(C) = a
 - GetTail(C) = (b)

2.5.5 存储结构



广义表的存储结构示例

最后一个是一个无限深度的广义表 $B = (a, B)$

2.5.6 例题

已知广义表 $L = (a, (b, c), (d, e))$ 使用取头函数 GetHead 和取尾函数 GetTail 求出原子 e

```
e = GetHead(GetTail(GetHead(GetTail(GetTail(L)))))
```

3. 串

3.1 串的定义和实现

串的定义

- **串 (String)** 是由零个或多个字符组成的有限序列。记为 $S = ' a_1 a_2 \dots a_n'$
- 串中任意多个连续的字符组成的子序列称为该串的**子串**
- 包含子串的串称为主串
- 由一个或多个空格组成的串称为空格串

串的存储结构

定长顺序存储表示

用一组地址连续的存储单元存储串值的字符序列

```
#define MAXLEN 255 //预定义最大串长为255
typedef struct{
    char ch[MAXLEN]; //每个分量存储一个字符
    int length; //串的实际长度
}SString;
```

堆分配存储表示

堆分配存储仍然以一组地址连续的存储单元存放串值的字符序列，但他们的存储空间是在程序执行过程中动态分配得到的

```
typedef struct{
    char *ch; //按串分配存储区，ch指向串的基地址
    int length; //串的长度
}HString;
```

在 C 语言中，存在一个称之为**堆**的自由存储区，并用 `malloc()` 和 `free()` 函数来完成动态存储管理

利用 `malloc()` 为每个新产生的串分配一块实际串长所需要的存储空间，若分配成功，则返回一个指向起始地址的指针，称为串的基地址，这个串由 `ch` 指针来指示；若分配失败，则返回 `NULL`。已分配的空间可用 `free()` 释放掉

块链存储表示

类似于线性表的链式存储结构，也可采用链表方式存储串值。由于串的特殊性，在具体实现时，每个节点即可以存放一个字符，也可以存放多个字符，每个节点称为**块**，整个链表称为**块链结构**

串的基本操作

```
StrAssign(&T, chars): 赋值操作。把串T赋值为chars
StrCopy(&T, S): 复制操作。由串S复制得到串T
StrEmpty(S): 判空操作。若S为空串，则返回TRUE，否则返回FALSE
StrCompare(S, T): 比较操作，S>T则返回值>0；若S=T，返回0，否则返回<0
StrLength(S): 求串长
SubString(&Sub, S, pos, len): 求子串。用Sub返回串S的第pos个字符起长度为len的子串
Concat(&T, S1, S2): 串联接。用T返回S1和S2的联接
Index(S, T): 定位操作
ClearString(&S): 清空
DestroyString(&S): 销毁串
```

3.2 串的模式匹配

简单的模式匹配算法 BF

子串的定位操作通常称为串的**模式匹配**，它求的是子串在主串中的位置

KMP 算法

基础概念

- 前缀：除最后一个字符以外，字符串的所有头部子串
- 后缀：除第一个字符外，字符串的所有尾部子串
- 部分匹配值 PM：字符串的前缀和后缀的最长相等前后缀长度

算法原理

编号	描述	1	2	3	4	5
s	字符	a	b	c	a	c
PM	子串右移位数=已匹配的字符数-对应的部分匹配值: $Move=(j-1)-PM[j-1]$	0	0	0	1	0
next (PM 右移一位)	子串右移位数: $Move=(j-1)-next[j]$, 子串的比较指针回退到: $j=next[j]+1$	-1	0	0	0	1
next=next+1	在子串的第j个字符与主串发生失配时, 则跳到子串的 $next[j]$ 位置重新与主串当前位置进行比较	0	1	1	1	2

next[] 推导方法

- $$next[j] = \begin{cases} 0 & j = 1 \\ max\{k | 1 < k < j \text{ 且 } p_1...p'_{k-1} = p_{j-k+1}...p'_{j-1}\} & \text{当此集合不空时 (next[]推导公式)} \\ 1 & \text{其他情况} \end{cases}$$
- next 的推导步骤, $next[j]=k$, 求 $next[j+1]$

$next[j]=k$ 表明 $p_1...p_{k-1} = p_{j-k+1}...p_{j-1}$

 - 若 $p_k = p_j$, 则 $next[j+1]=next[j]+1$
 - 若 $p_k \neq p_j$. 用前缀 $p_1...p_k$ 去跟后缀 $p_{j-k+1}...p_j$ 匹配, 则当 $p_k \neq p_j$ 是应将 $p_1...p_k$ 向右滑动至以第 $next[k]$ 个字符与 p_j 比较, 如果 $p_{next[k]}$ 与 p_j 还是不匹配, 那么需要寻找长度更短的相等前后缀, 下一步继续用 $P_{next[next[k]]}$ 与 p_j 比较, 直到找到 $k'=next[next...[k]]$ 满足条件 $p_1...p'_{k'} = p_{j-k'+1}...p'_{j-1}$, 则 $next[j+1]=k'+1$

说明

- 为什么 $next[1]=0$: 当模式串中的第一个字符与主串的当前字符比较不相等时, $next[1]=0$, 表示模式串应该右移一位, 主串当前指针后移一位, 再和模式串的第一个字符进行比较
- 为什么要取 $max\{k\}$: 当主串的第 i 个字符与模式串的第 j 个字符失配时, 主串 i 不回溯, 则假定模式串的第 k 个字符与主串的第 i 个字符比较, k 应满足条件 $1 < k < j$ 且 $p_1...p'_{k-1} = p_{j-k+1}...p'_{j-1}$ 。为了不使向右移动丢失可能的匹配, 右移距离应该取最小, 由于 $j - k$ 表示右移距离, 所以取 $max\{k\}$ 。

```
//next数组的建立
int[] getNext(String ps){
    char[] p = ps.toCharArray();
    int[] next = new int[p.length];
    next[1] = 0;
    int j = 1, k = 0;
    while (j < p.length) {
        if (k == 0 || p[j] == p[k]) {
            next[++j] = ++k;
        } else{
            k = next[k];
        }
    }
    return next;
}
```

```

int Index(SSString S, SString T){
    int i = 1, j = 1;
    while(i<=S.length && j<=T.length){ //跳出循环情况: j>T.length, 匹配成功
        if(S.ch[i]==T.ch[j]){
            //i>S.length, 匹配失败
            ++i; ++j; //继续比较后面的字符
        }else{
            //匹配失败, 指针i不变, j后退, 匹配下一个
            j=next[j];
        }
    }
    if(j > T.length){ //j>T.length, 匹配成功
        return i-T.length;
    }else{
        return 0;
    }
}

```

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1	2	1	2	3

也可以通过求部分匹配值表的方法来求 next 数组。

2) 利用 KMP 算法的匹配过程如下。

第一趟：从主串和模式串的第一个字符开始比较，失配时 $i=6$ 、 $j=6$ 。

主串	a	a	b	a	a	b	a	a	b	a	a	c
	a	a	b	a	a	c						

第二趟： $\text{next}[6]=3$ ，主串当前位置和模式串的第 3 个字符继续比较，失配时 $i=9$ 、 $j=6$ 。

主串	a	a	b	a	a	<u>b</u>	a	a	b	a	a	c
	a	a	b	a	a	c						

第三趟： $\text{next}[6]=3$ ，主串当前位置和模式串的第 3 个字符继续比较，匹配成功。

主串	a	a	b	a	a	b	a	a	<u>b</u>	a	a	c
	a	a	<u>b</u>	a	a	c						

KMP 算法的时间复杂度：

设主串长度为 n ，模式串长度为 m ，则

最好情况：每次匹配第一个字符就匹配失败，直到最后才匹配成功，循环 n 次，最好时间复杂度 = $O(n)$

最坏情况：到最后也没找到，主串移动 n 个元素，模式串移动 m 个元素，最坏时间复杂度 = $O(m+n)$

KMP 算法的进一步优化

$\text{nextval}[]$ ，如果出现了 $p_j = p_{\text{next}[j]}$ ，则将 $\text{next}[j]$ 修正为 $\text{next}[\text{next}[j]]$ ，直到两者不相等

主串 <i>S</i>	a	a	a	b	a	a	a	a	b
模式 <i>P</i>	a	a	a	a	b				
<i>j</i>	1	2	3	4	5				
<i>next[j]</i>	0	1	2	3	4				
<i>nextval[j]</i>	0	0	0	0	4				

图 4.7 KMP 算法进一步优化示例

4. 树

4.1 树

4.1.1 树的定义

- 有且仅有一个特定的称为根的结点
- 当 $n > 1$ 时，其余结点可分为 m 个互不相交的有限集 T_1, T_2, \dots, T_m ，其中每个集合本身又是一棵树，并且称为根的子树

4.1.2 基本术语

1. 祖先、子孙、双亲、孩子、兄弟
2. 结点的度、树的度
3. 分支结点、叶子节点
4. 节点的深度、高度、层次
5. 有序树和无序树
6. 路径和路径长度
7. 森林

4.1.3 树的性质

- 树中的节点数等于所有结点的度数之和加 1
- 总结点数 = $n_0 + n_1 + n_2 + \dots + n_m$
- 总分支数 = $1n_1 + 2n_2 + \dots + mn_m$
- 总结点数 = 总分支数 + 1

4.2 二叉树

4.2.1 二叉树的定义及其主要特性

定义

每个节点至多只有两棵子树，并且二叉树的子树有左右之分，不能颠倒

几个特殊的二叉树

- 满二叉树：一棵高度为 h ，且含有 2^{h-1} 个结点的二叉树称为满二叉树，除叶子结点外每个结点度数为 2
- 完全二叉树：每个结点都与同等高度的满二叉树有同样的编号
 - 若 $i \leq \lfloor n/2 \rfloor$, 则结点 i 为分支结点，否则为叶子结点
 - 当 $2i \leq n$ 时，结点 i 的左孩子编号为 $2i$, 否则无左孩子
 - 当 $2i + 1 \leq n$ 时，结点 i 的右孩子编号为 $2i + 1$, 否则无右孩子
 - 结点 i 所在的层次为 $\lfloor \log_2 i \rfloor$
 - 叶子结点只可能在层次最大的两层上出现，对于最大层次中的叶子结点，都依次排列在该层最左边的位置上
 - 若有度为 1 的节点，则只可能有一个，且只有左孩子没有右孩子
 - 若 n 为奇数，则每个分支结点都有左孩子和右孩子；若 n 为偶数，则编号最大的分支结点只有左孩子，没有右孩子
- 二叉排序树：左子树上所有节点的关键字小于根节点的关键字；右子树上的所有节点的关键字均大于根节点的关键字
- 平衡二叉树：树上任一节点的左子树和右子树的深度之差不超过 1

二叉树的性质

- 非空二叉树上的叶子结点数等于度为 2 的结点数 + 1，即 $n_0 = n_2 + 1$

4.2.2 二叉树的存储结构

顺序存储结构

将完全二叉树上编号为 i 的结点元素存储在一维数组下标为 $i-1$ 的分量中

注意：这种存储结构建议从数组下标 1 开始存储树中的结点，若从数组下标 0 开始存储，则计算其孩子结点时与之前描述的计算公式不一致，在书写程序时需要注意。

链式存储结构

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

```
typedef struct BiTNode{  
    ELEMTYPE data; // 数据域  
    struct BiTNode *lchild,*rchild; // 左、右孩子指针  
}BiTNode,*BiTree;
```

n 个结点的二叉链表中，含有 $n+1$ 个空链域

4.3 二叉树的遍历和线索二叉树

4.3.1 二叉树的遍历

二叉树的遍历是按照某条搜索路径访问树中每个结点，使得每个结点均被访问一次，而且仅被访问一次。共有先序遍历（NLR）、中序（LNR）、后续（LRN）三中遍历方法

递归遍历算法

```

void PreOrder(BiTree T){//PreOrder-先序、InOrder-中序、PostOrder-后序
    if(T!=NULL){
        visit(T);//访问根结点
        PreOrder(T->lchild);//遍历访问左子树
        PreOrder(T->rchild);//遍历访问右子树
    }
}

```

非递归遍历算法

- 先序遍历

```

void PreOrder2(BiTree T){
    InitStack(S);BiTree p=T;
    while(p||!IsEmpty(S)){
        if(p){
            visit(p);Push(S,p);
            p = p->lchild;
        }
        else{
            Pop(S,p);
            p = p->rchild;
        }
    }
}

```

- 中序遍历

```

void Inorder2(BiTree T){
    InitStack(S);BiTree p = T;//初始化S, p是遍历指针
    while(p||!IsEmpty(S)){
        if(p){//一路向左
            Push(S,p);//当前结点入栈
            p = p->lchild;//左孩子不空，一直往左走
        }
        else{//出栈，并转向出栈结点的左子树
            Pop(S,p);visit(p);//栈顶元素出栈，访问出栈结点
            p = p->rchild;//向右子树走
        }
    }
}

```

- 后序遍历

```

void PostOrder(BiTTree T){
    InitStack(S);
    P = T;
    r = NULL;
    while(p || !IsEmpty(S)){
        if(p){           //走到最左边
            push(S,p);
            p = p->lchild;
        }
        else{           //向右
            GetTop(S,p); //读栈顶结点(非出栈)
            if(p->rchild&&r->rchild!=r) //若右子树存在,且未被访问过
                p = p->rchild;          //转向右
            else{                     //否则,弹出结点并访问
                pop(S,p);           //将结点弹出
                visit(p->data);     //访问该结点
                r = p;                //记录最近访问过的结点
                p = NULL;             //结点访问完,重置p指针
            }
        } //else
    } //while
}

```

- 层次遍历

```

void LevelOrder(BiTTree T){
    InitQueue(Q);
    BiTree p;
    EnQueue(Q,T); //将根结点入队
    while(!IsEmpty(Q)){
        DeQueue(Q,p);
        visit(p);
        if(p->lchild!=NULL)EnQueue(Q,p->lchild);
        if(p->rchild!=NULL)EnQueue(Q,p->rchild);
    }
}

```

由遍历序列构造二叉树

由二叉树中序遍历结果和前序、后序、层次中的一个组合，就可唯一确定一棵二叉树。
例如，求先序序列 (ABCDEF~~GHI~~) 和中序序列 (BCAEDGHFI) 所确定的二叉树。

首先，由先序序列可知 A 为二叉树的根结点。中序序列中 A 之前的 BC 为左子树的中序序列，DEF GHI 为右子树的中序序列。然后由先序序列可知 B 是左子树的根结点，D 是右子树的根结点。以此类推，就能将剩下的结点继续分解下去，最后得到的二叉树如图 5.9(c)所示。

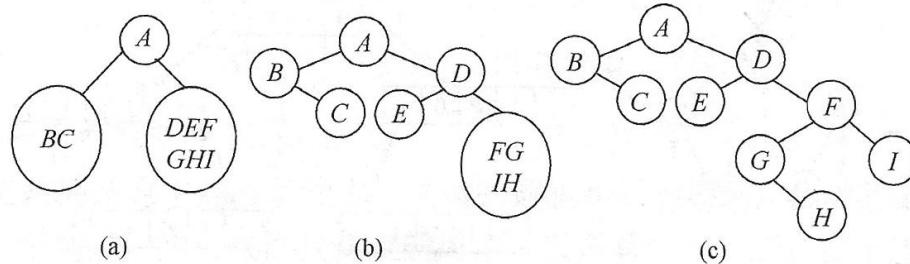


图 5.9 一棵二叉树的构造过程

4.3.2 线索二叉树

线索二叉树的基本概念

在含 n 个结点的二叉树中，有 n+1 个空指针。引入线索二叉树正是为了加快查找结点前驱和后继的速度。

规定：若无左子树，令 lchild 指向其前驱结点；若无右子树，令 rchild 指向其后继结点

lchild	ltag	data	rtag	rchild
--------	------	------	------	--------

$$ltag = \begin{cases} 0, & lchild \text{域指示结点的左孩子} \\ 1, & lchild \text{域指示结点的前驱} \end{cases}$$

$$rtag = \begin{cases} 0, & rchild \text{域指示结点的右孩子} \\ 1, & rchild \text{域指示结点的后继} \end{cases}$$

```
typedef struct ThreadNode{
    ElemtType data;
    struct ThreadNode *lchild,*rchild;
    int ltag,rtag;
}ThreadNode,*ThreadTree;
```

中序线索二叉树的构造

线索化的实质就是遍历一次二叉树

中序线索二叉树的遍历

在对其进行遍历时，只要先找到序列中的第一个节点，然后依次找结点的后继，直至后继为空。在中序线索二叉树中找结点后继的规律是：若其右标志为“1”，则右链为线索，指示其后继，否则遍历右子树中第一个访问的结点为其后继。

先序线索二叉树和后序线索二叉树

后序线索二叉树上找后继时需要知道结点双亲，即需采用带标志域的三叉链表作为存储结构。

4.4 树、森林

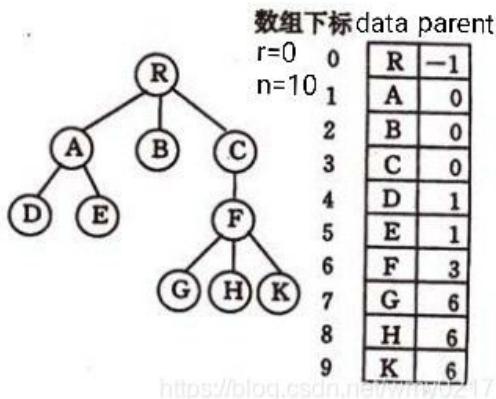
4.4.1 树的存储结构

顺序存储 和 链式存储

方法：

- 双亲表示法（顺序存储）
- 孩子表示法（顺序+链式存储）
- 孩子兄弟表示法（链式存储）

4.4.1.1 双亲表示法（顺序存储）



双亲表示法：顺序存储 结点数据，结点中保存父结点在数组中的下标

采用一组连续空间来存储每个结点，同时在每个结点中增设一个伪指针，指示其双亲结点在数组中的位置。

根节点的下标为 0，其伪指针域为 -1

该存储结构利用了每个结点只有唯一双亲对的性质，可以很快的得到每个结点的双亲结点，但求结点的孩子需要遍历整个结构。

优点：找父节点方便。

缺点：找孩子不方便。

注：双亲表示法与二叉树的顺序存储不一样，双亲表示法也可表示二叉树

类型描述

结点 包括 数据 和 父亲下标，

树 包括 结点数组 和 结点个数

```
#define MAX_TREE_SIZE 100           //树中最多结点数
typedef struct{                     //树的结点定义
    ElemtType data;               //数据元素
    int parent;                  //双亲位置域
}PTNode;
typedef struct{                   //树的类型定义
    PTNode nodes[MAX_TREE_SIZE]; //双亲表示
    int n;                      //结点数
}PTree;
```

增加一个结点

新增元素，无需按逻辑次序存储，可以放到删除结点留下的存储空间里

删除一个结点

方案一：数据取出，双亲指针改为-1

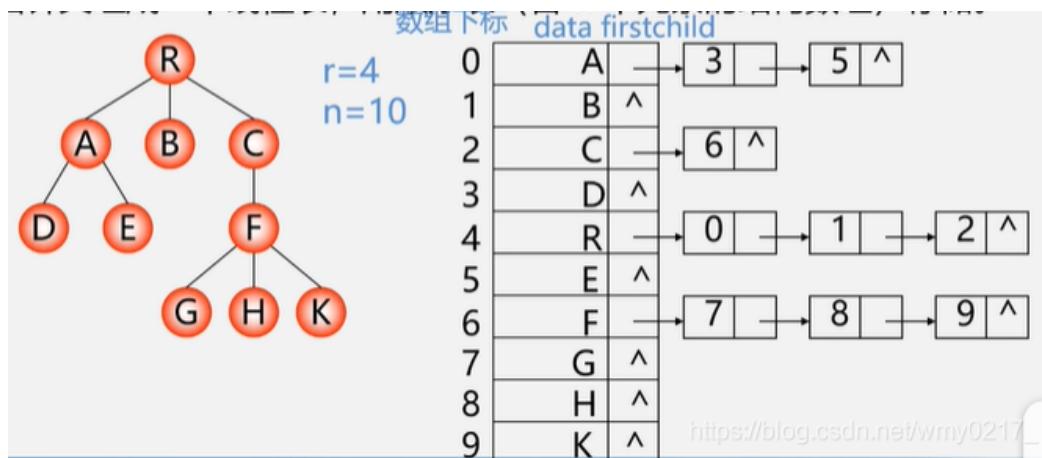
方案二：用存储空间中最后一个存的结点把要删的结点覆盖

查找一个结点

找父结点方便、找孩子不方便。

空数据导致遍历慢。

4.4.1.2 孩子表示法（顺序+链式存储）



孩子表示法：顺序存储 结点数据，结点中保存孩子 链表 头指针（链式存储）

优点：找孩子方便。

缺点：找父节点不方便。

类型描述

孩子结点 包括 孩子下标 和 下一个孩子指针，

数组 包括 数据 和 孩子结点，

树 包括 数组 和 数组元素（结点）个数 及 根的下标

```
#define MAX_TREE_SIZE 100           //树中最多结点数
struct CTNode{
    int child;                   //孩子结点在数组中的位置
    struct CTNode *next;         //下一个孩子
};
typedef struct{
    ELEMTYPE data;
    struct CTNode* firstchild;   //第一个孩子
}CTBox;
typedef struct{
    CTBox nodes[MAX_TREE_SIZE];
    int n, r;                   //结点数和根的位置
}CTree;
```

增加一个结点

新增元素，父结点后新增一个孩子结点，数组中加一个数组元素

删除一个结点

父结点后的链表中将此结点删除

数组中：

- ①若此结点后无链表，则直接删除
- ②若此结点后有链表，再处理子树

查找一个结点

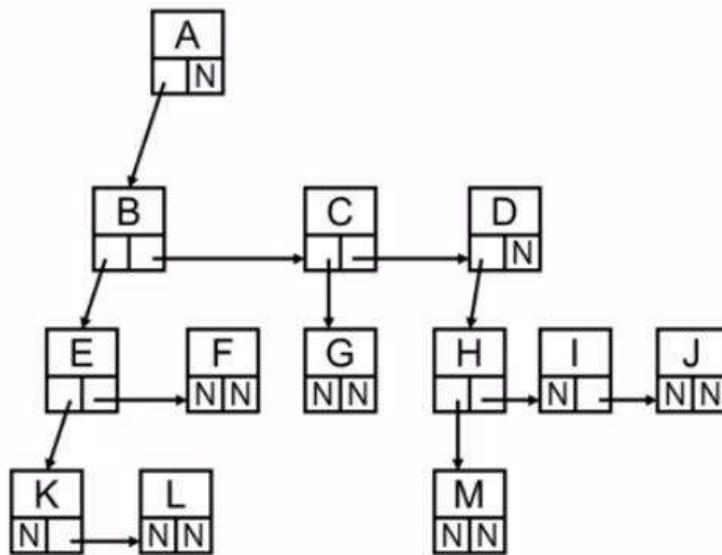
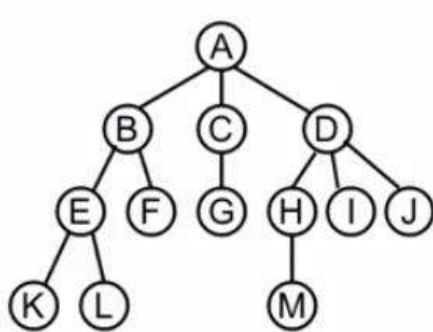
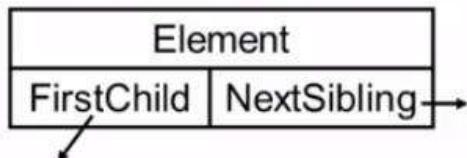
按图一行一行遍历

找孩子结点方便，找父结点不方便

4.4.1.3 孩子兄弟表示法（顺序+链式存储）

❖ 儿子-兄弟表示法

中国



孩子兄弟表示法：用二叉链表存储树——两个指针：第一个孩子和右兄弟

用此方法存储的树，形态上和二叉树类似

类型描述

由二叉树的链式存储（二叉链表）改变而来

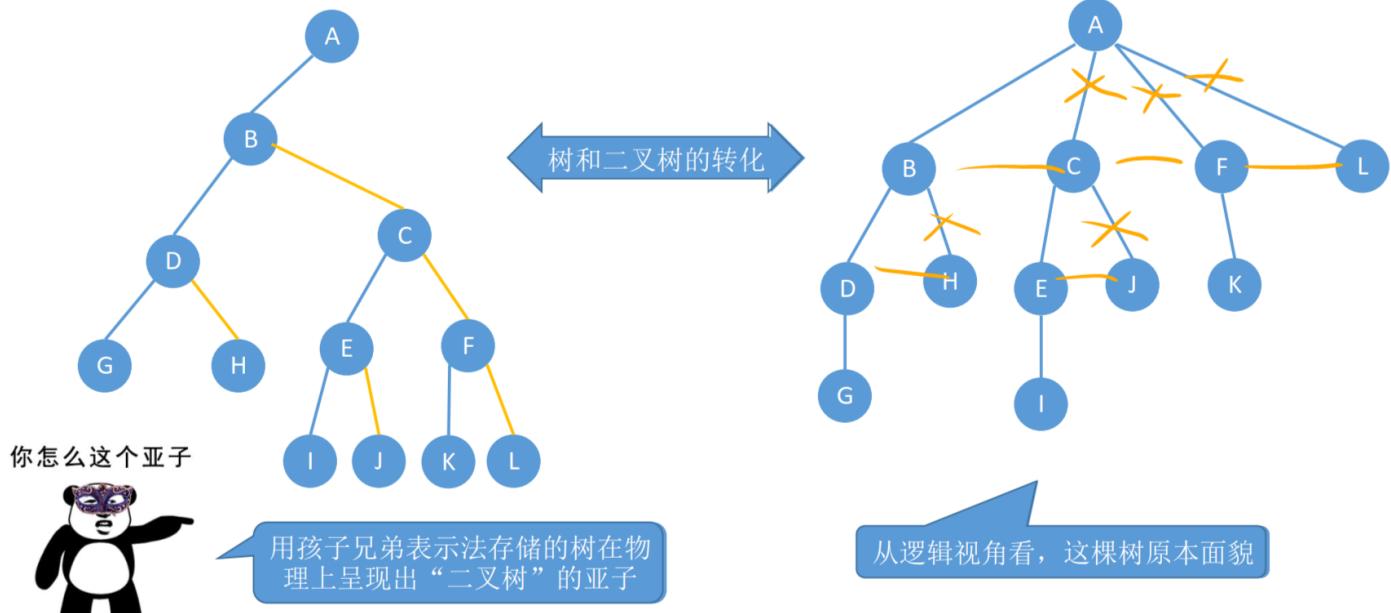
```
typedef struct CSNode{
    Elemtyp data; //数据域
    struct CSTNode *firstchild, *nextsibling; //第一个孩子和右兄弟指针
}CSTNode, *CSTree;
```

4.4.2 树、森林与二叉树的转化

树转换为二叉树

- 规则：**左孩子右兄弟**。每个结点左指针指向它的第一个孩子，右指针指向它在树中的相邻右兄弟
- 画法：
 - 1. 在兄弟结点之间加一连线；
 - 2. 对每个结点，只保留它与第一个孩子的连线，而与其他孩子的连线全部抹掉；
 - 3. 以树为轴心，顺时针旋转 45°
- 特点：根无右子树
-

孩子兄弟表示法（链式存储）



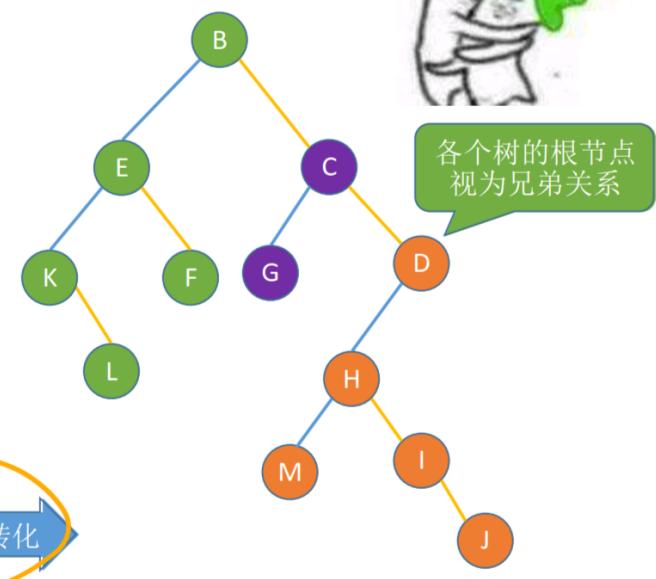
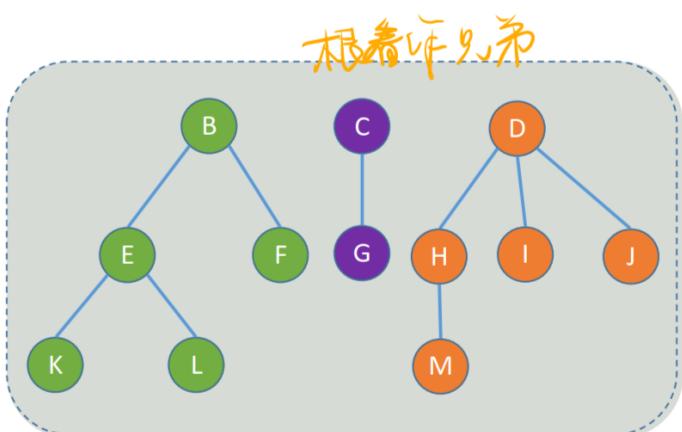
森林转换二叉树

- 规则：先将森林中的每棵树转换为二叉树；把第二棵树的根作为第一棵树根的右兄弟，以此类推
- 画法：
 - 森林中的每棵树转换为相应的二叉树
 - 每棵树的根也可视为兄弟关系，在每棵树的根之间加一根连线；
 - 以第一棵树的根为轴心顺时针旋转 45°
- 特点：森林中每棵树的根节点从第二个开始依次连接到前一棵树的根的右孩子，因此最后一棵树的根节点的右指针为空。另外，每个非终端节点，其所有孩子结点在转换后，最后一个孩子的右指针也为空。

森林和二叉树的转换

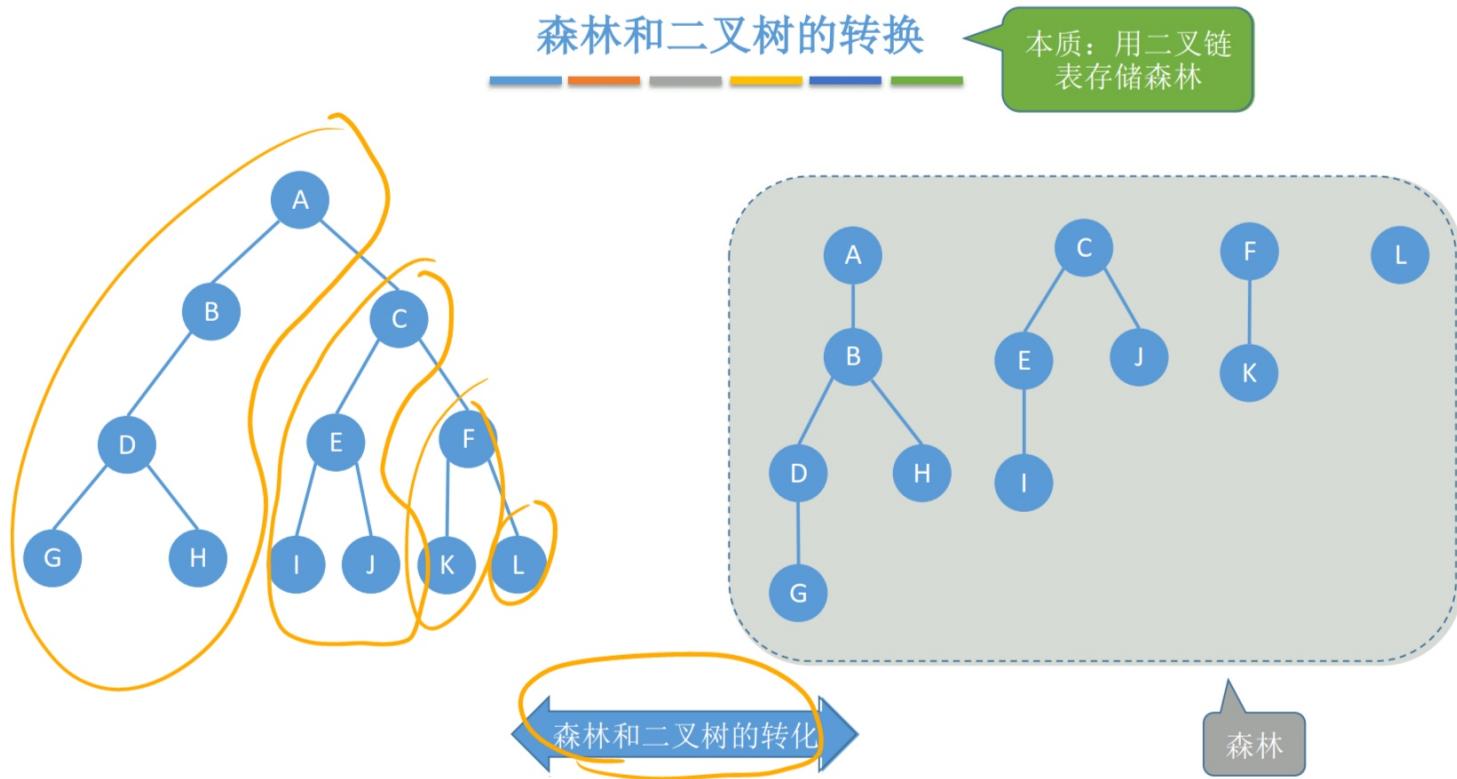
本质：用二叉链表存储森林

森林。森林是 m ($m \geq 0$) 棵互不相交的树的集合



二叉树转换为森林

若二叉树非空，则二叉树的根及其左子树为第一棵树的二叉树形式，故将根的右链断开。二叉树根的右子树又可视为由除第一棵树外的森林转换后的二叉树。



4.4.3 树和森林的遍历

树的遍历

- 先根遍历
- 后根遍历

森林的遍历

- 先序遍历森林
 - 访问森林中第一棵树的根节点
 - 先序遍历第一棵树的根节点的子树森林
 - 先序遍历除去第一棵树后剩余的树构成的森林
- 中序遍历森林（又称后根遍历）
 - 中序遍历森林中第一棵树的根节点的子树森林
 - 访问第一棵树的根结点
 - 中序遍历除去第一棵树后剩余的树构成的森林
- 树和森林的遍历与二叉树遍历的对应关系

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

4.5 树与二叉树的应用

4.5.1 二叉排序树 BST

二叉排序树的定义

左子树上所有节点的关键字小于根节点的关键字；右子树上的所有节点的关键字均大于根节点的关键字

二叉排序树的查找、插入、构造

二叉排序树的删除

- 若被删除结点 z 是叶子结点，则直接删除
- 若结点 z 只有一棵左子树或右子树，则让 z 的子树成为 z 父节点的子树，替代 z 的位置
- 若结点 z 有左、右两棵子树，则令 z 的直接后继替代 z，然后从二叉排序树中删去这个直接后继，这样就转换成了上面的两种情况

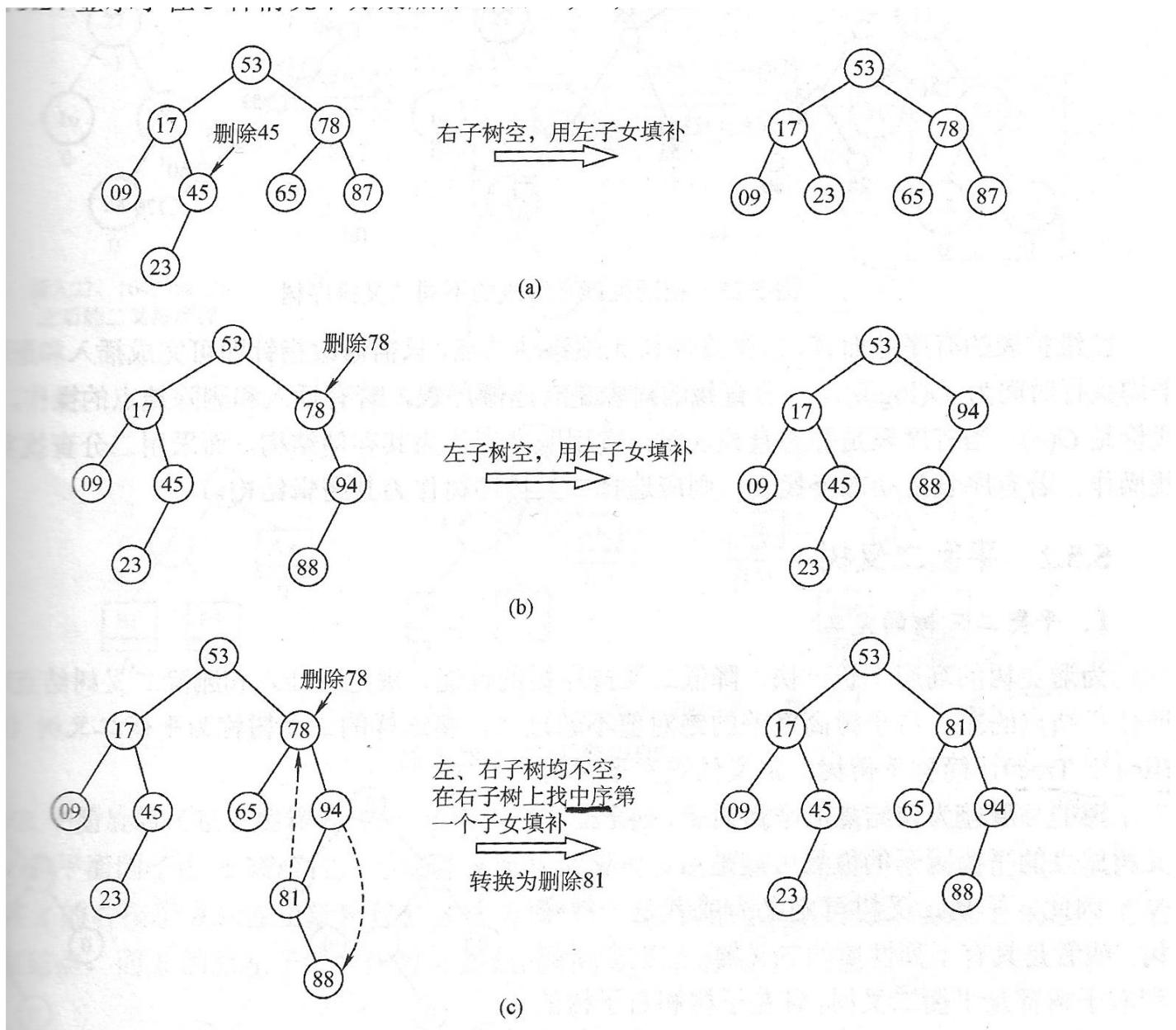


图 5.24 3 种情况下的删除过程

二叉排序树的查找效率分析

- $O(\log_2 n)$ (平衡二叉树) $\sim O(n)$ (链表)
- ASL_a : 平均查找长度

4.5.2 平衡二叉树

定义

任意结点的左右子树高度差的绝对值不超过 1 的二叉排序树

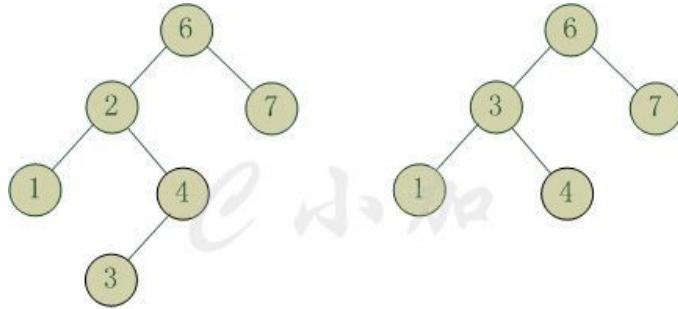


图1 两颗二叉查找树，只有右边的是AVL树

二叉排序树的插入

- ①LL: 在A的左孩子的左子树中插入导致A的不平衡，将A的左孩子右上旋。

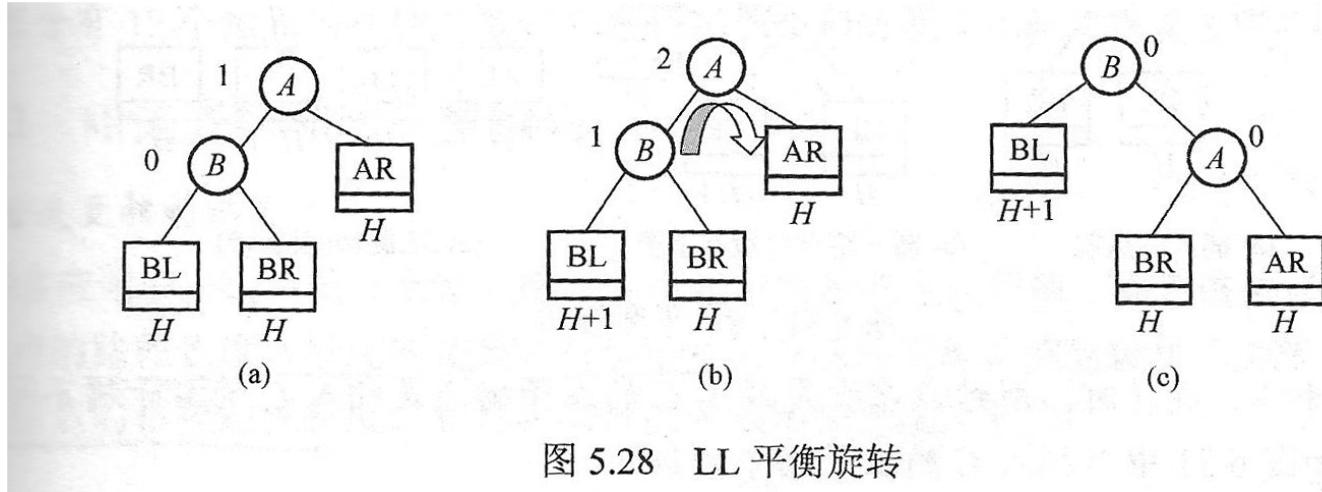


图 5.28 LL 平衡旋转

- ②RR: 在A的右孩子的右子树中插入导致A的不平衡，将A的右孩子左上旋。

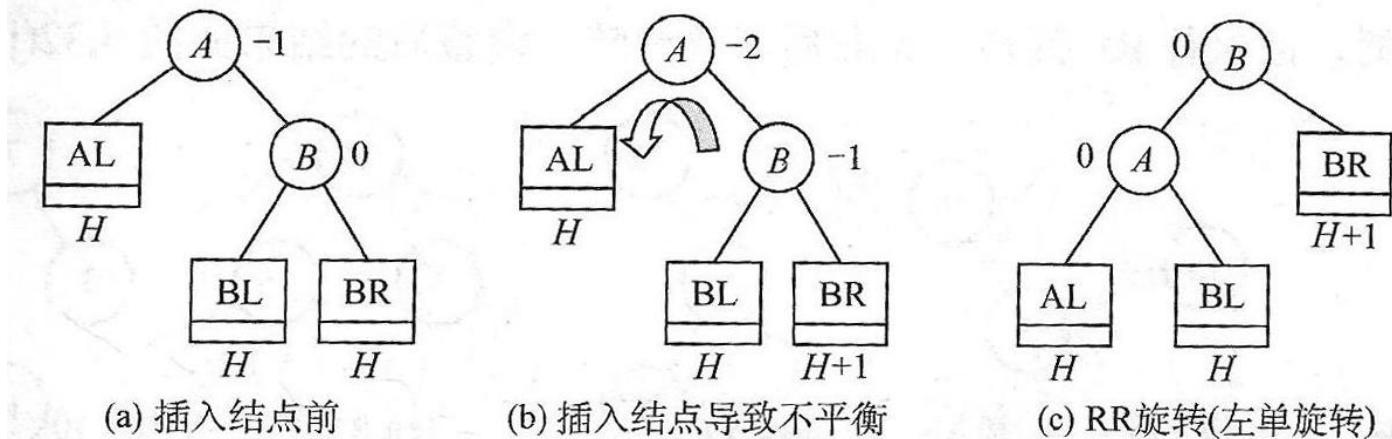


图 5.29 RR 平衡旋转

- ③LR: 在A的左孩子的右子树中插入导致A的不平衡, 将A的左孩子的右孩子, 先左上旋再右上旋。

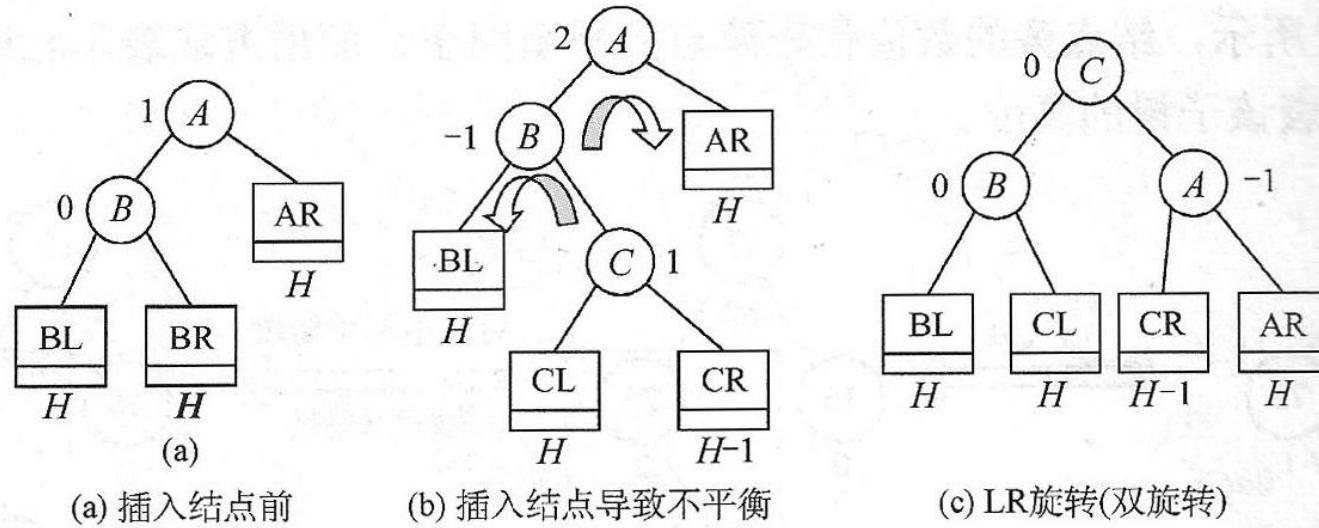


图 5.30 LR 平衡旋转

- ④RL: 在A的右孩子的左子树中插入导致A的不平衡, 将A的右孩子的左孩子, 先右上旋再左上旋。

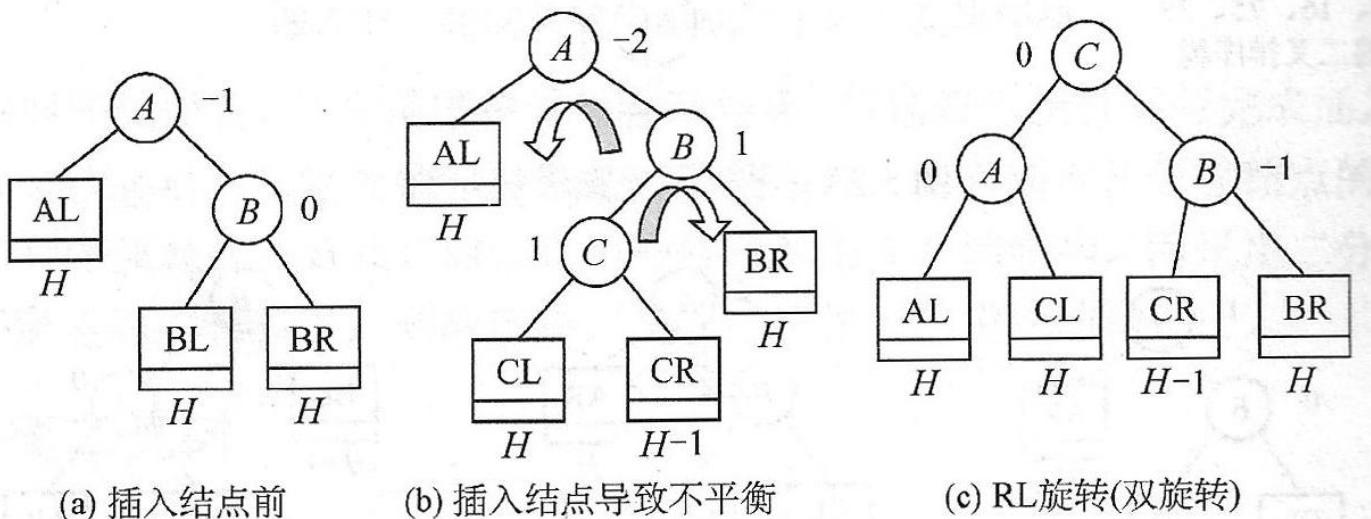


图 5.31 RL 平衡旋转

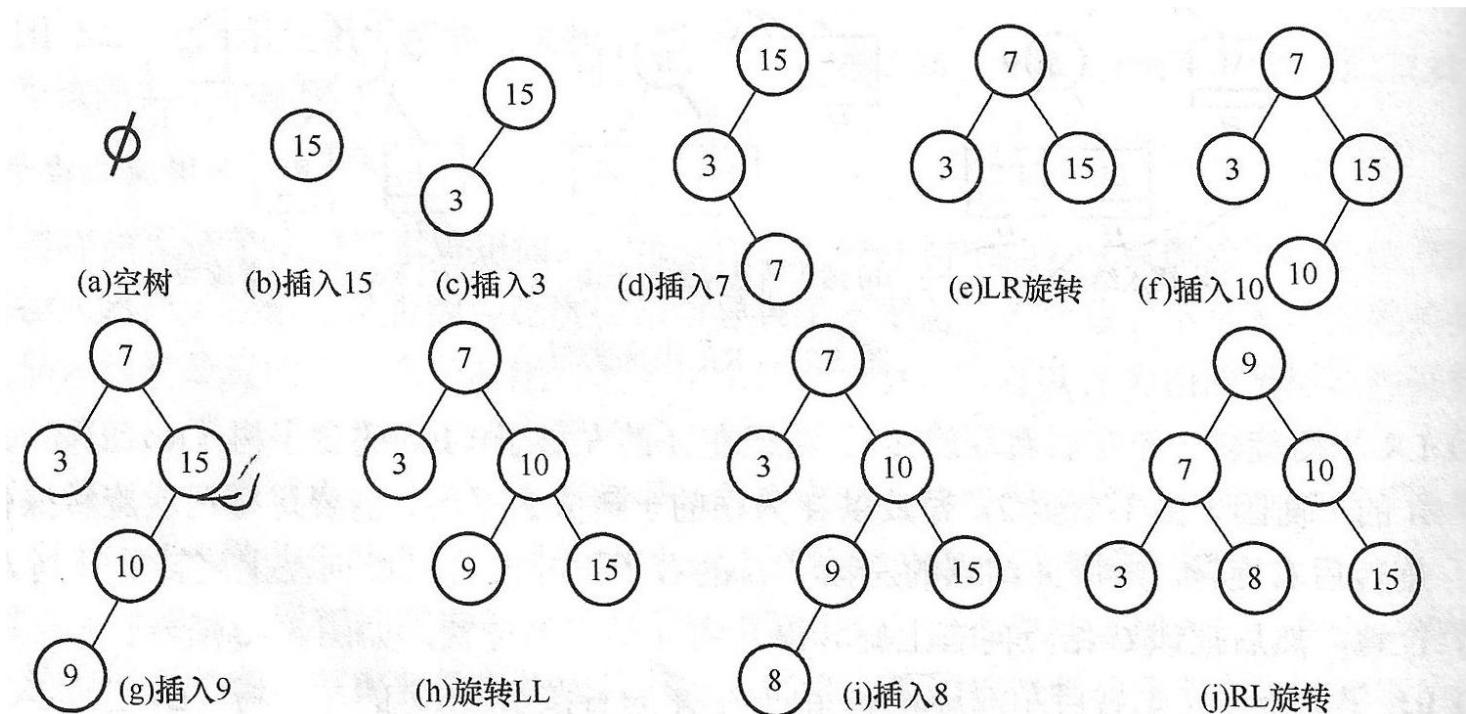


图 5.32 平衡二叉树的生成过程

平衡二叉树的查找

- 含有 n 个结点的平衡二叉树的最大深度为 $O(\log_2 n)$, 平均查找长度为 $O(\log_2 n)$
- 平衡二叉树结点数的递推关系 $n_h = 1 + n_{h-1} + n_{h-2}$, $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, n_h 为构造此高度的平衡二叉树所需的最少结点数

二叉树的关键字个数不超过树的深度。假设以 n_h 表示深度为 h 的平衡树中含有的最少结点数。显然, $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, 并且有 $n_h = n_{h-1} + n_{h-2} + 1$ 。可以证明, 含有 n 个结点的平衡二叉树的最大深度为 $O(\log_2 n)$, 因此平衡二叉树的平均查找长度为 $O(\log_2 n)$, 如图 5.33 所示。

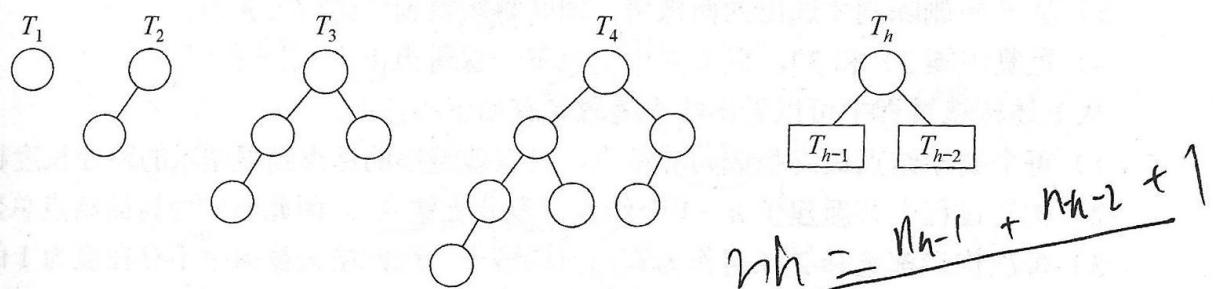


图 5.33 结点个数 n 最少的平衡二叉树

4.5.3 哈夫曼树和哈夫曼编码

定义

- 带权路径长度: $WPL = \sum_{i=1}^n w_i l_i$
- 含有 n 个带权结点的二叉树中, 带权路径长度 WPL 最小的二叉树称为哈夫曼树

哈夫曼树的构造

- 将 n 个结点分别作为 n 棵仅含有一个结点的二叉树, 构成森林 F

2. 构造一个新结点，从 F 中选取两棵根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根结点的权值之和
3. 从 F 中删除刚才选出的两棵树，同时将新得到的树加入 F 中
4. 重复 2-3 步骤

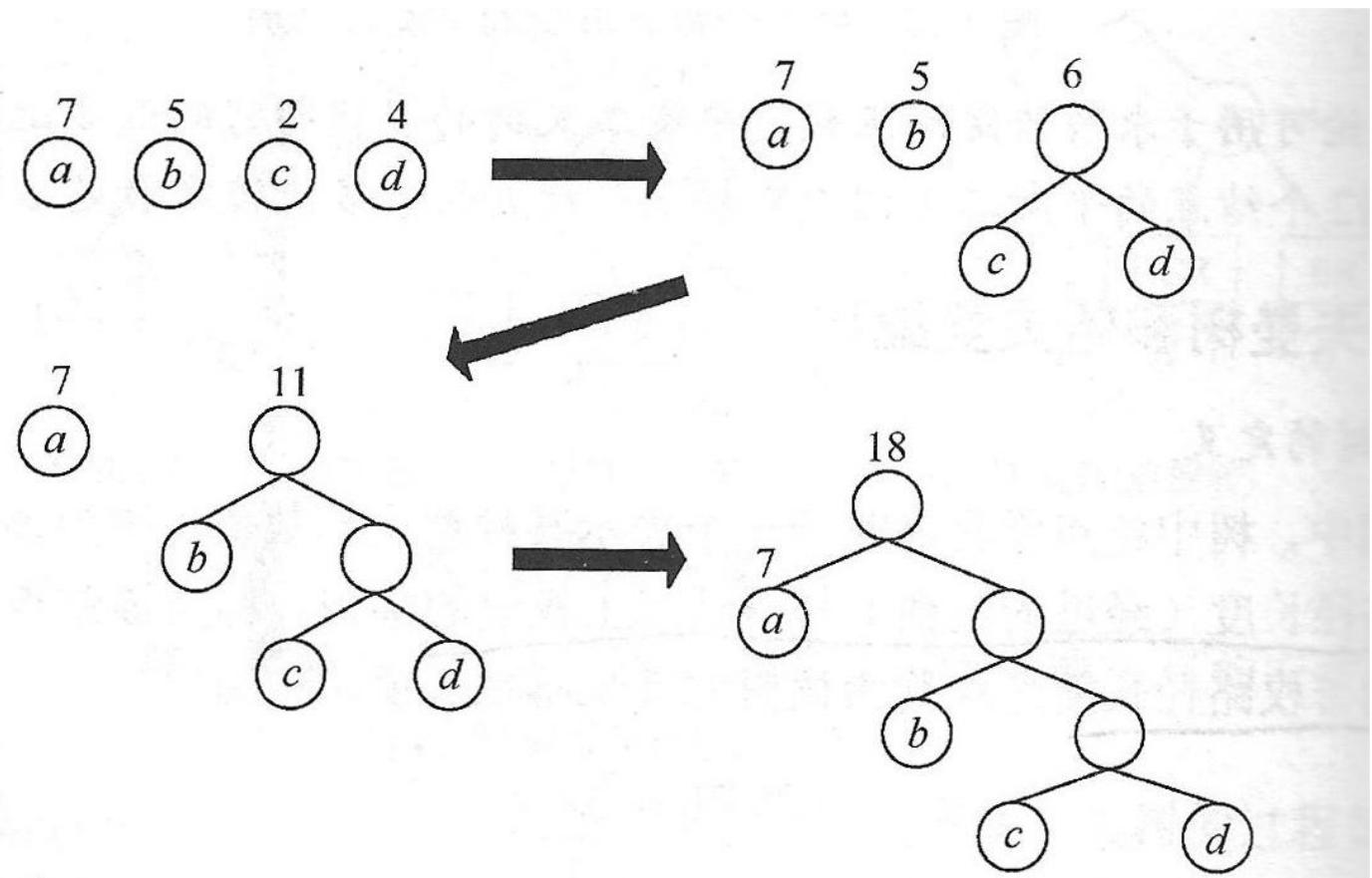


图 5.35 哈夫曼树的构造过程

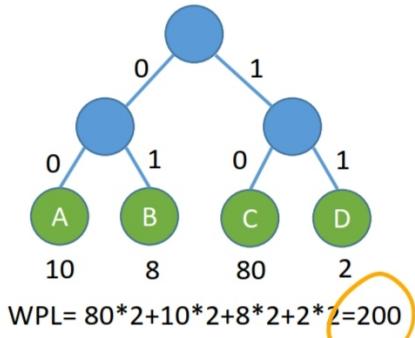
哈夫曼树特点

- 每个初始节点最终都成为叶结点
- 构造过程中新建了 $n-1$ 个结点，哈夫曼树中结点总数为 $2n-1$
- 不存在度为 1 的节点

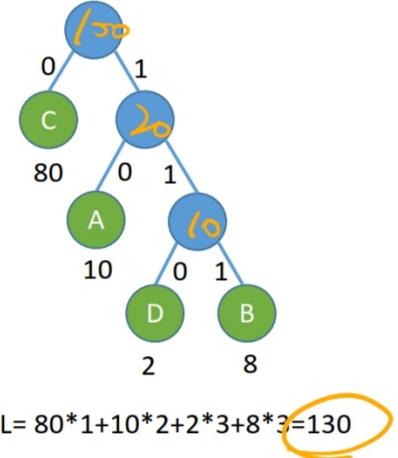
哈夫曼编码

- 固定长度编码：对每个字符用相等长度的二进制位表示
- 可变长度编码：允许对不同字符用不等长的二进制位表示
- 前缀编码：没有一个编码是另一个编码的前缀
- 使用哈夫曼树得到哈夫曼编码：默认为左边为 0，右边为 1（不唯一，没明确规定）

假设，100题中有80题选C，10题选A，8题选B，2题选D
所有答案的二进制长度=80*2+10*2+8*2+2*2=200 bit



可变长度编码——允许对不同字符用不等长的二进制位表示



5. 图

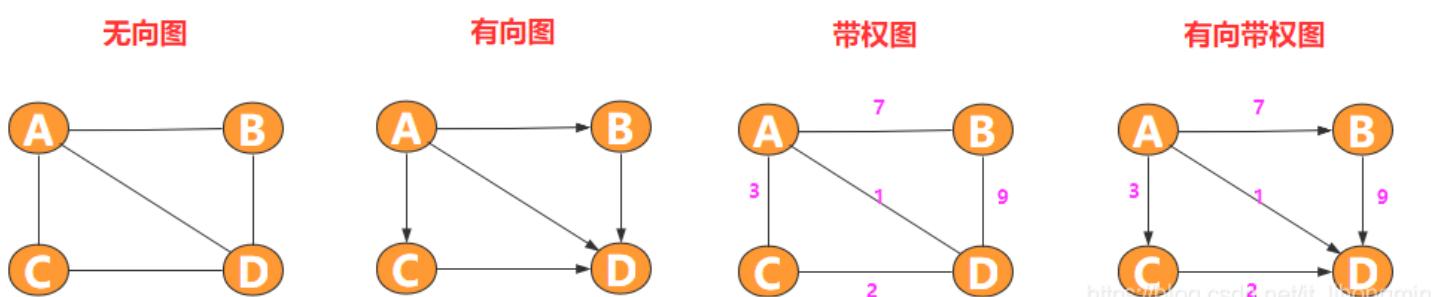
5.1 图的基本概念

图 G 由顶点集 V 和边集 E 组成，记为

$G = (V, E)$ ，其中 $V(G)$ 表示图 G 中顶点的有限非空集； $E(G)$ 表示图 G 中顶点之间的关系集合。

- $V = \{v_1, v_2, \dots, v_n\}$, $|V|$ 表示顶点个数
- $E = \{(u, v) | u \in V, v \in V\}$, $|E|$ 表示图 G 中边的条数

5.1.2 基本术语



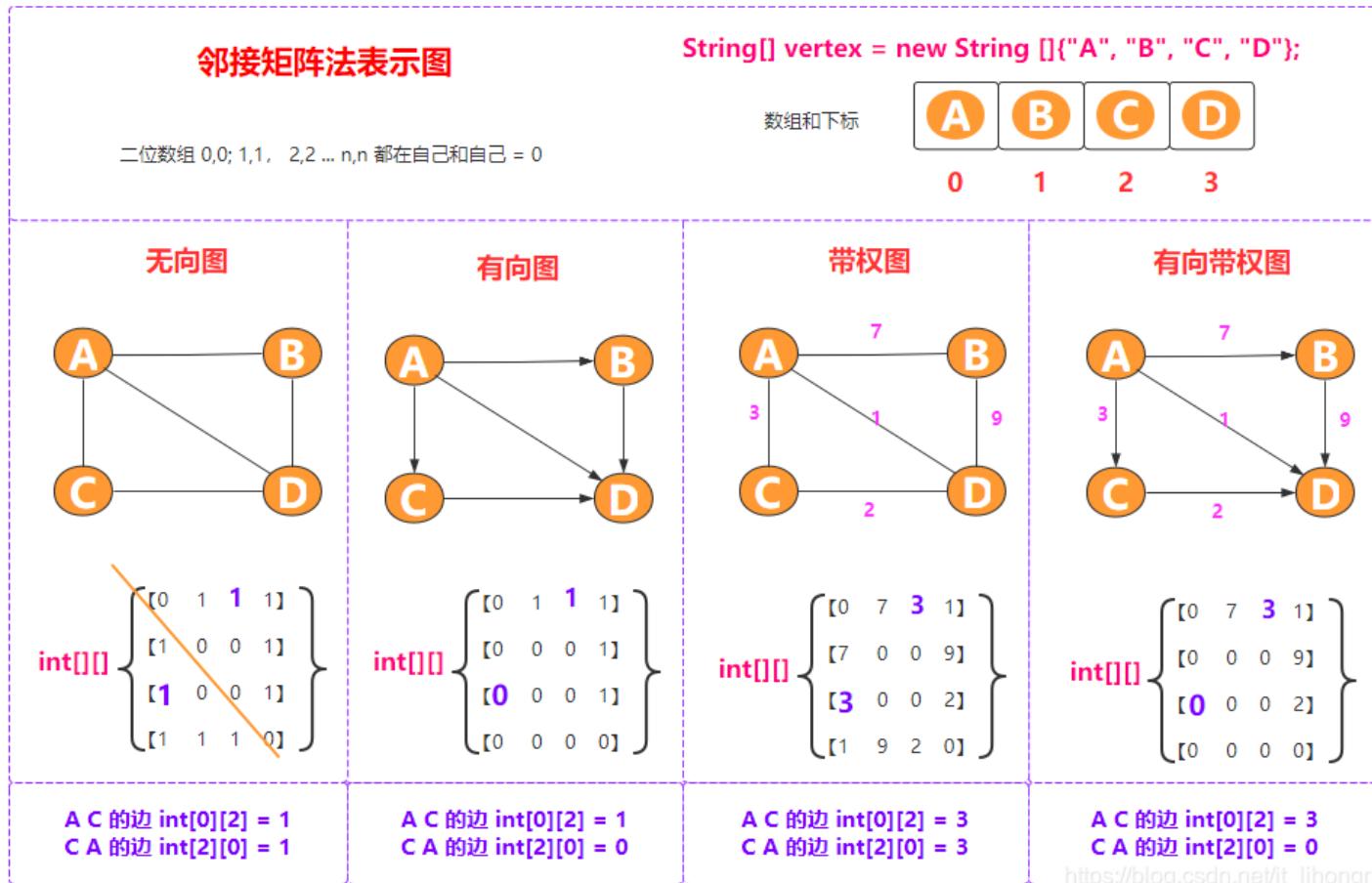
- 有向图
- 无向图
- 简单图、多重图
- 完全图
- 子图
- 连通、连通图和连通分量
- 强连通图、强连通分量
- 生成树、生成森林
- 顶点的度、入度和出度
- 边的权和网
- 稠密图、稀疏图

- 路径、路径长度和回路
- 简单路径、简单回路
- 距离
- 有向树

5.2 图的存储及基本操作

- 必须完整、准确地反映顶点集合边集的信息

5.2.1 邻接矩阵法



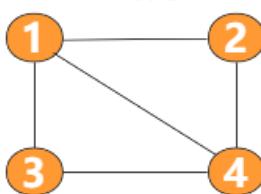
- 一个一维数组存储图中顶点的信息
- 一个二维数组存储图中边的信息，称为**邻接矩阵**
- 设图 G 的邻接矩阵为 A , A^n 的元素等于从顶点 i 到 j 的长度为 n 的路径数目

5.2.2 邻接表法

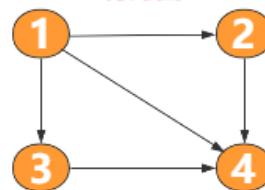
邻接表-表示图

LinkedList<Integer>[] data;

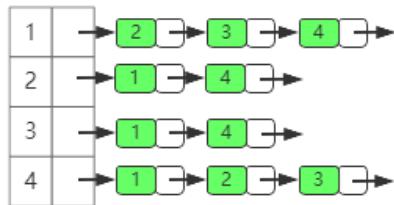
无向图



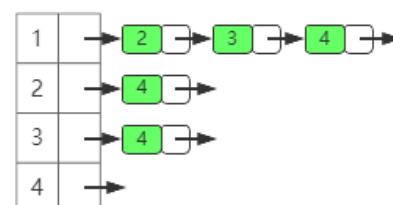
有向图



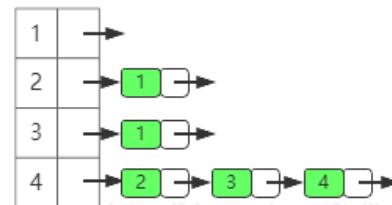
邻接表



邻接表



逆邻接表



https://blog.csdn.net/f_ljihongmin

- 顶点表结点

边表的头指针和顶点的数据信息采用顺序存储

顶点域	边表头指针
data	firstarc

- 边表结点

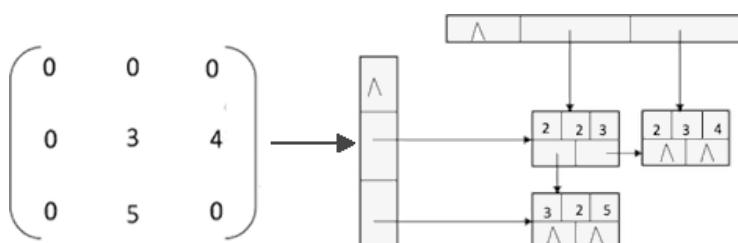
对每个顶点 v_i 建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边，这个单链表为顶点 v_i 的边表

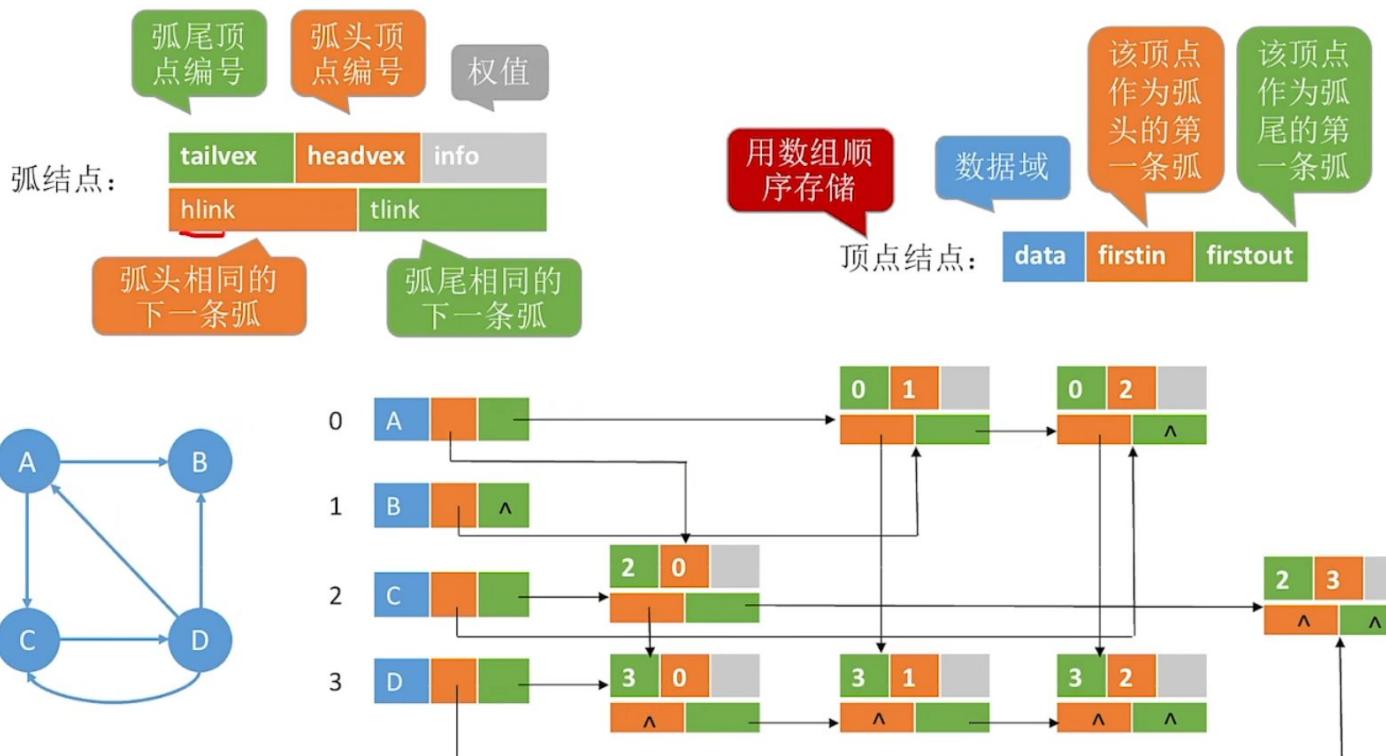
邻接点域	指针域
adjvex	nextarc

- 若存储的是无向图，空间复杂度为 $O(|V| + 2|E|)$ ；若为有向图，空间复杂度为 $O(|V| + |E|)$

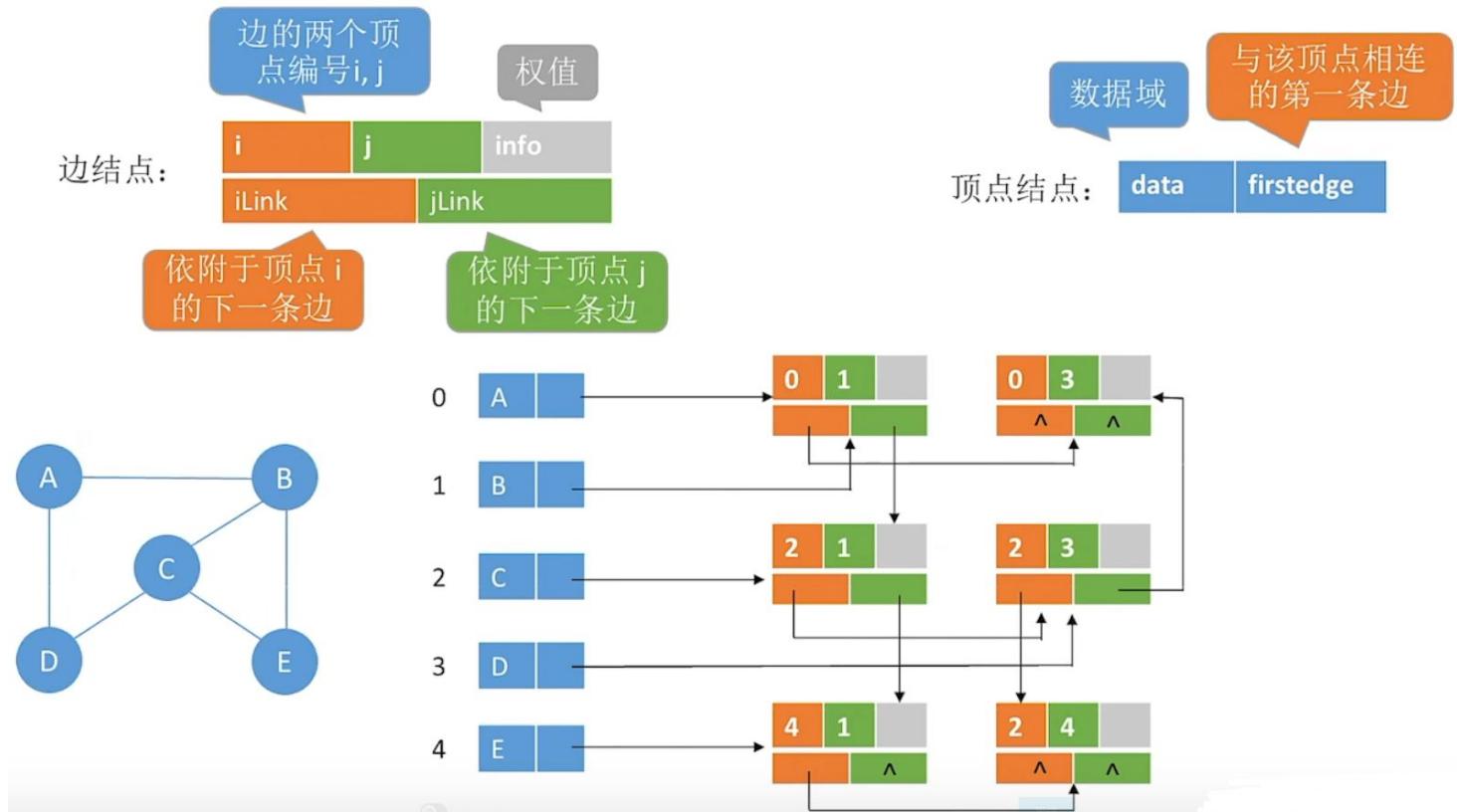
5.2.3 十字链表

- 有向图的一种链式存储结构
- 对于有向图中的每条弧有一个结点，对于每个顶点也有一个结点





5.2.4 邻接多重表



- 无向图的链式存储结构
- 与邻接表的区别是，同一条边在邻接多重表中只有一个结点

5.2.5 图的基本操作

`Adjacent(G,x,y)`: 判断图G是否存在边 $\langle x, y \rangle$
`Neighbors(G,x)`: 列出图G中与结点x邻接对的边
`InsertVertex(G,x)`: 在图G中插入顶点x
`DeleteVertex(G,x)`: 从图G中删除顶点x
`AddEdge(G,x,y)`: 若边 $\langle x, y \rangle$ 不存在, 则向图G中添加该边
`RemoveEdge(G,x,y)`: 若边 $\langle x, y \rangle$ 存在, 则从图G中删除该边
`FirstNeighbor(G,x,y)`: 求图G中顶点x的第一个邻接点, 若有则返回顶点号。否则返回-1
`NextNeighbor(G,x,y)`: 假设图G中顶点y是顶点x的一个邻接点, 返回除y外顶点x的下一个邻接点的顶点号, 若没有返回-1
`Get_edge_value(G,x,y)`: 获取图G中边 $\langle x, y \rangle$ 的权值
`Set_edge_value(G,x,y,v)`: 设置图G中边 $\langle x, y \rangle$ 对应的权值为v

5.3 图的遍历

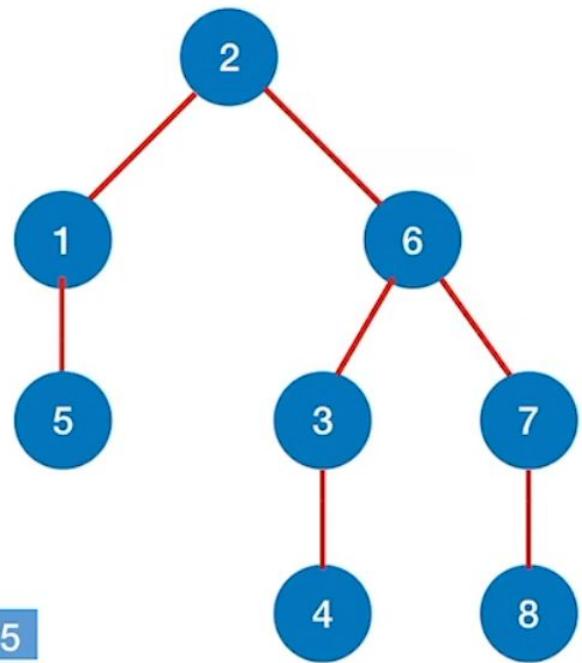
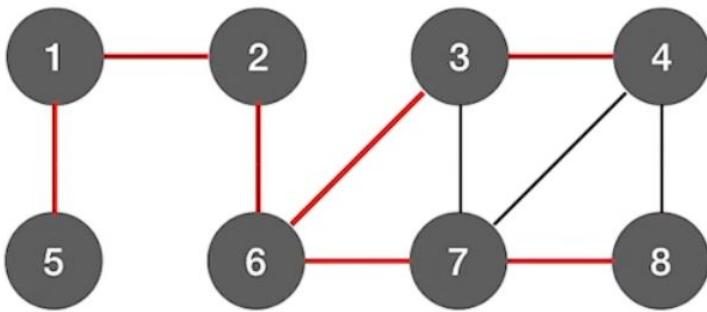
图的遍历是从图中某一点出发, 按照某种搜索方法沿着图中的对边对图中所有顶点访问且只访问一次。

5.3.1 广度优先算法

- 基本思想

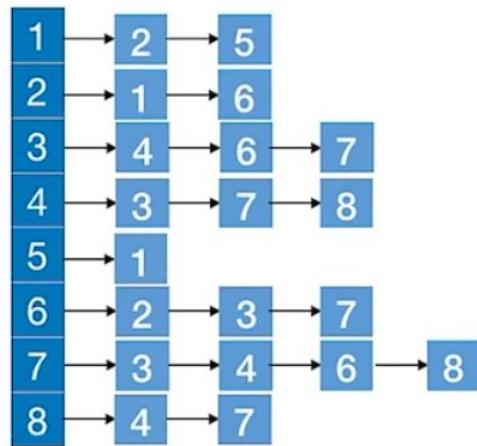
首先访问起始顶点 v , 接着由 v 出发, 依次访问 v 的各个未访问过的邻接顶点 w_1, w_2, \dots, w_i , 然后一次访问 w_1, w_2, \dots, w_i 的所有未被访问的邻接顶点。

换句话说, BFS 是以 v 为起始点, 由近及远依次访问和 v 有路径相通且路径长度为 $1, 2, \dots$ 的顶点



	1	2	3	4	5	6	7	8
1	0	1	0	0	1	0	0	0
2	1	0	0	0	0	1	0	0
3	0	0	0	1	0	1	1	0
4	0	0	1	0	0	0	1	1
5	1	0	0	0	0	0	0	0
6	0	1	1	0	0	0	1	0
7	0	0	1	1	0	1	0	1
8	0	0	0	1	0	0	1	0

邻接矩阵



邻接表

```

bool visited[MAX_VERTEX_NUM];//访问标记数组
void BFSTraverse(Graph G){ //对图G进行广度优先遍历
    for(i=0;i<G.vexnum;i++)
        visited[i] = FALSE;
    InitQueue(Q);
    for(i=0;i<G.vexnum;++i)
        if(!visited[i]) //对每个连通分量调用一次BFS
            BFS(G,i);
}
void BFS(Graph G,int v){//从顶点v出发, 广度优先遍历图G
    visit(v);
    visited[v] = TRUE;
    EnQueue(Q,v);
    while(!isEmpty(Q)){
        DeQueue(Q,v);
        for(w=FirstNeighbor(G,v);w>=0;w=Neighbor(G,v,w))
            if(!visited[w]){
                visit(w);
                visited[w] = TRUE;
                EnQueue(Q,w);
            }
    }
}
  
```

5.3.2 深度优先搜索

Depth-First-Search,DFS

- 基本思想

首先访问图中某一起始顶点 v , 然后由 v 出发, 访问与 v 邻接且未被访问的任一顶点 w_1 , 再访问与 w_1 邻接且未被访问的任一顶点, 重复上述过程

```
bool visited[MAX_VERTEX_NUM];
void DFSTraverse(Graph G){
    for(v=0;v<G.vexnum;++v)
        visited[v] = FALSE;
    for(v=0;v<G.vexnum;++v)
        if(!visited[v])
            DFS(G,v);
}
void DFS(Graph G,int v){
    visit(v);
    visited[v] = TRUE;
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
        if(!visited[w]){
            DFS(G,w);
        }
}
```

性能

性能	广度优先搜索	深度优先搜索
空间复杂度	$O(V)$	$O(V)$
时间复杂度-邻接矩阵	$O(V ^2)$	$O(V ^2)$
时间复杂度-邻接表	$O(V + E)$	$O(V + E)$
生成树	生成树, 邻接表不唯一, 邻接矩阵唯一	

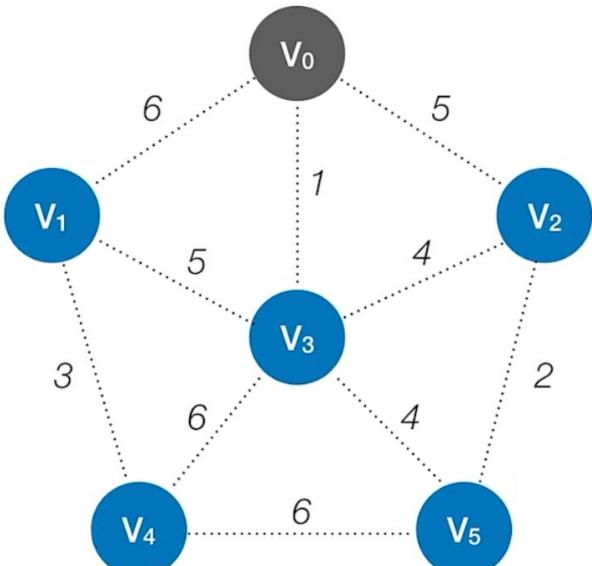
5.4 图的应用

5.4.1 最小生成树

求一个带权连通图的最小生成树 Minimum-Spanning-Tree,MST

Prim 算法

- 基本思想



初始：从V₀开始

标记各节点是否已加入树

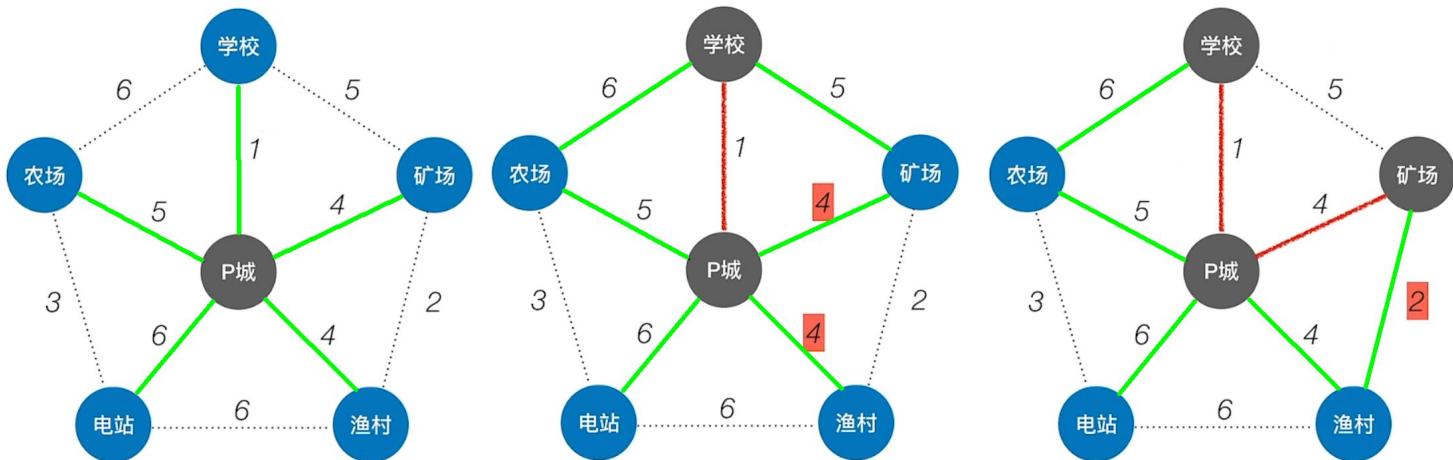
isJoin[6]

V0	V1	V2	V3	V4	V5
✓	✗	✗	✗	✗	✗

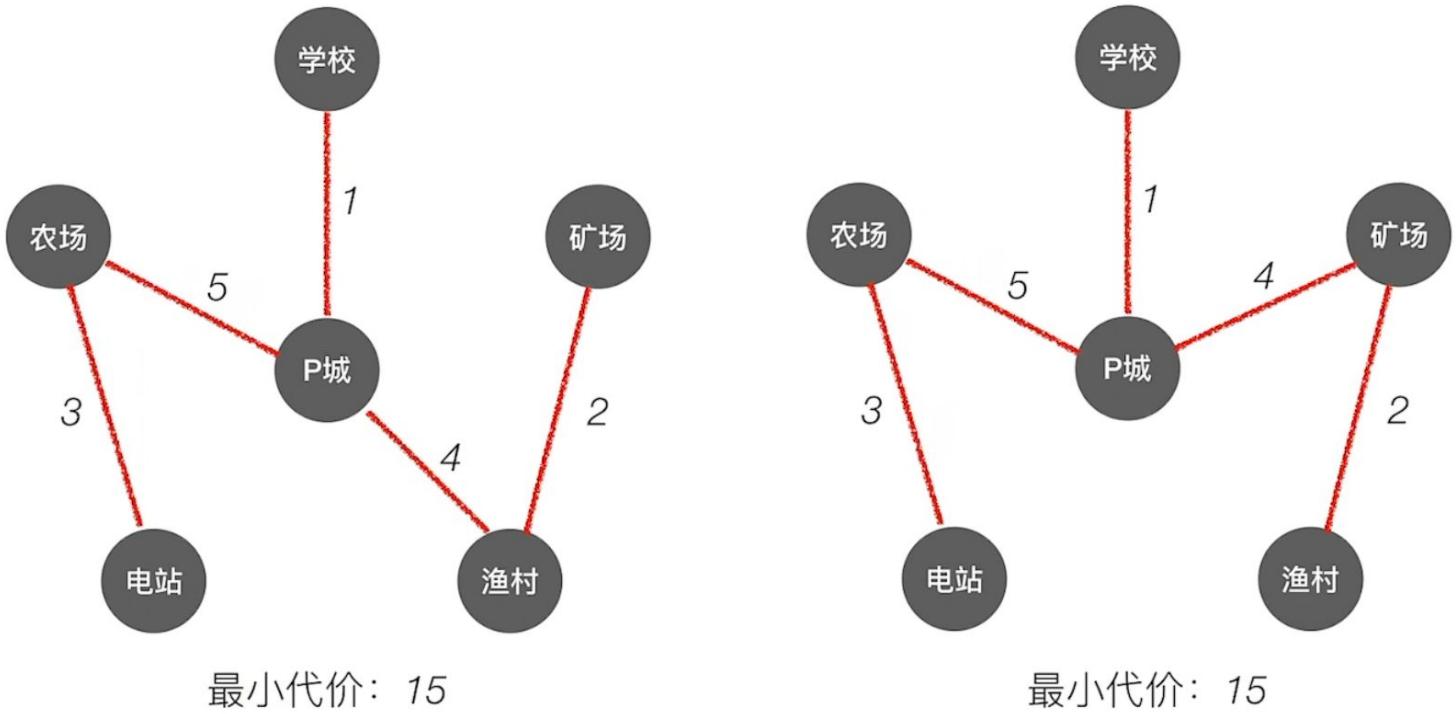
lowCost[6]

各节点加入树的最低代价

- 初始时从图中任取一顶点加入树 T, 此时树中只含有一个顶点
- 之后选择一个与当前 T 中顶点集合距离最近的顶点, 且加入后不能出现环, 并将该顶点和相应的边加入 T, 每次操作后 T 中的顶点数和边数都增 1。
- 重复直到加满
- 时间复杂度: $O(|V|^2)$



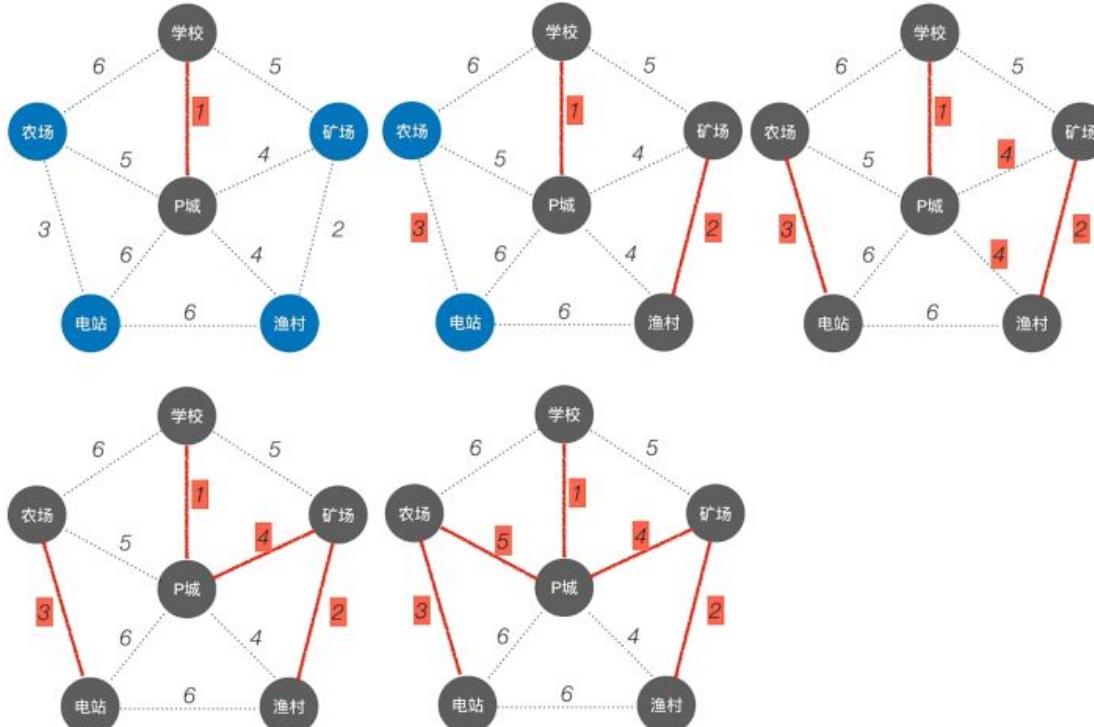
最终可能的结果：



Kruskal 算法

基本思想

- 初始时为只有 n 个顶点而无边的非连通图 T , 每个顶点自成一个连通分量
- 按照边的权值由小到大, 加入到非连通图 T 中, 不能形成环
- 重复直到加满
- 时间复杂度: $O(|E|\log|E|)$, 每轮判断是否属于同一个集合, 需要 $O(\log|E|)$



Kruskal 算法时间复杂度: $O(|V|^2)$ 适合用于边稠密图

Kruskal 算法时间复杂度: $O(|E|\log|E|)$ 适合用于边稀疏图

5.4.2 最短路径

Dijkstra 算法求单源最短路径

辅助数组

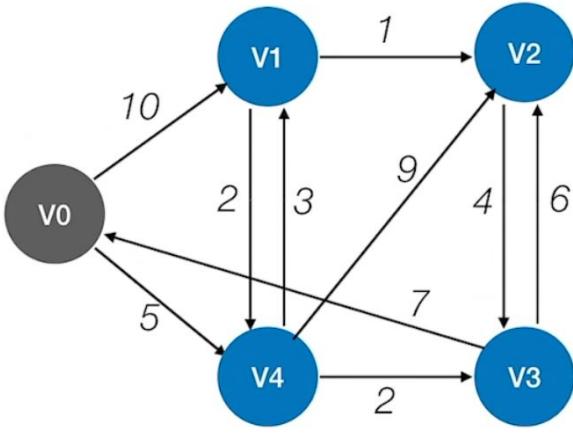
- 集合 s ：记录以求得的最短路径的顶点
- $dist[]$ ：记录从源点 v_0 到其他各顶点当前的最短路径长度
- $path[]$ ： $path[i]$ 表示从源点到顶点 i 之间的最短路径的前驱结点。可用于回溯找最短路径

算法步骤

- 初始化：集合 s 初始化为 $\{v_0\}$, $dist[i] = \infty$ 的初始值 $dist[i] = arcs[0][i]$
 - 从顶点集合 $V - s$ 中选出 v_j , 满足 $dist[j] = \min\{dist[i] | v_i \in V - S\}$, 令 $S = S \cup \{j\}$
 - 根据公式修改从 v_0 出发到集合 $V - S$ 上任一顶点 v_k 可达的最短路径长度, 若 $dist[j] + arcs[j][k] < dist[k]$, 则更新
 - 重复步骤 2-3 操作共 $n-1$ 次
- 时间复杂度: $O(|V|^2)$
 - 注意:Dijkstra 算法不适用于有负权值的带权图

案例：

- 初始：从 V_i 开始，初始化三个数组信息如下



标记各顶点是否已找到最短路径

final[5]

V0	V1	V2	V3	V4
✓	✗	✗	✗	✗

最短路径长度

dist[5]

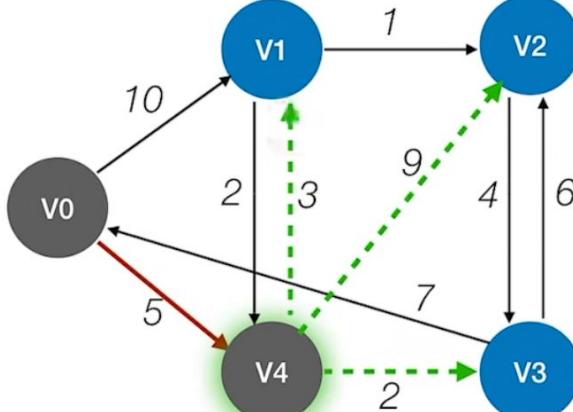
0	10	∞	∞	5
---	----	----------	----------	---

路径上的前驱

path[5]

-1	0	-1	-1	0
----	---	----	----	---

- 第 1 轮：循环遍历所有结点，找到还没确定最短路径，且 $dist$ 最小的顶点 V_i , 令 $final[i] = true$



标记各顶点是否已找到最短路径

final[5]

V0	V1	V2	V3	V4
✓	✗	✗	✗	✓

最短路径长度

dist[5]

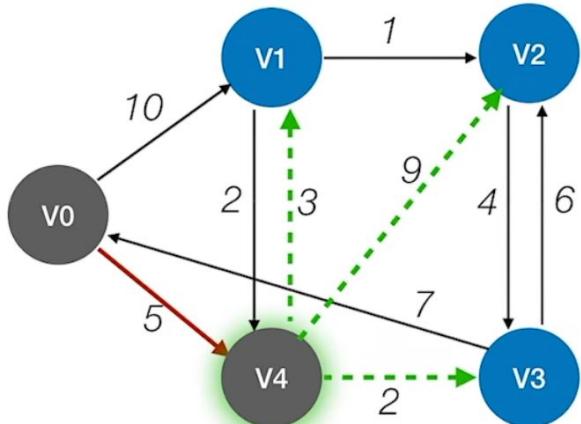
0	10	∞	∞	5
---	----	----------	----------	---

路径上的前驱

path[5]

-1	0	-1	-1	0
----	---	----	----	---

检查所有邻接自 V_i 的顶点，若其 $final$ 值为 false，则更新 $dist$ 和 $path$ 信息



标记各顶点是否已找到最短路径

final[5]

V0	V1	V2	V3	V4
✓	✗	✗	✗	✓

最短路径长度

dist[5]

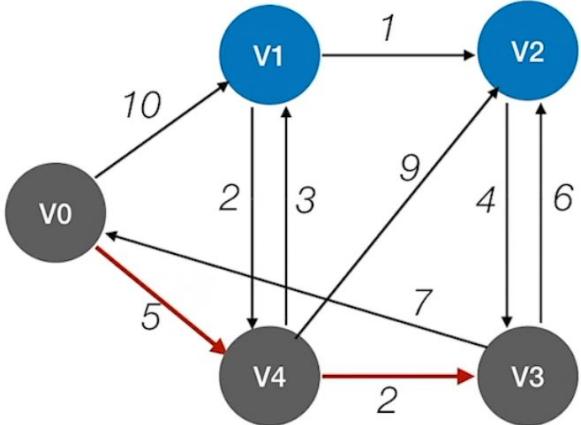
0	8	14	7	5
---	---	----	---	---

路径上的前驱

path[5]

-1	4	4	4	0
----	---	---	---	---

- 第 2 轮：循环遍历所有结点，找到还没确定最短路径，且 dist 最小的顶点 V_i ，令 $\text{final}[i] = \text{ture}$



标记各顶点是否已找到最短路径

final[5]

V0	V1	V2	V3	V4
✓	✗	✗	✓	✓

最短路径长度

dist[5]

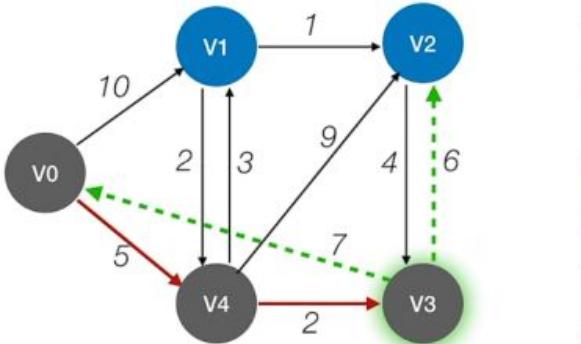
0	8	14	7	5
---	---	----	---	---

路径上的前驱

path[5]

-1	4	4	4	0
----	---	---	---	---

检查所有邻接自 V_i 的顶点，若其 final 值为 false，则更新 dist 和 path 信息



标记各顶点是否已找到最短路径

final[5]

V0	V1	V2	V3	V4
✓	✗	✗	✓	✓

最短路径长度

dist[5]

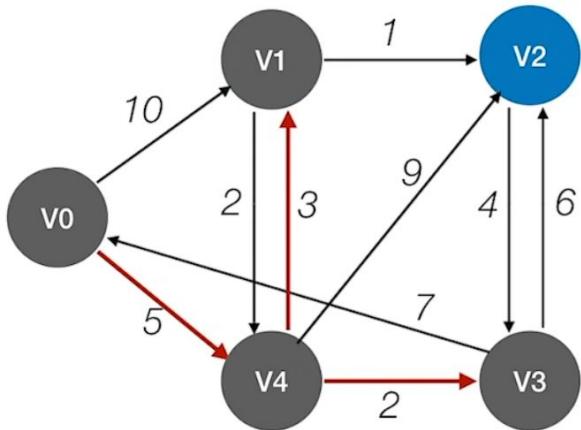
0	8	13	7	5
---	---	----	---	---

路径上的前驱

path[5]

-1	4	3	4	0
----	---	---	---	---

- 第 3 轮：循环遍历所有结点，找到还没确定最短路径，且 dist 最小的顶点 V_i ，令 $\text{final}[i] = \text{ture}$



标记各顶点是否已找到最短路径

final[5]

最短路径长度

dist[5]

路径上的前驱

path[5]

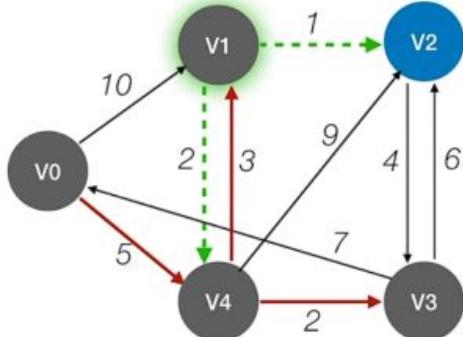
V0	V1	V2	V3	V4
✓	✓	✗	✓	✓



0	8	13	7	5
---	---	----	---	---

-1	4	3	4	0
----	---	---	---	---

检查所有邻接自 V_i 的顶点，若其 final 值为 false，则更新 dist 和 path 信息



标记各顶点是否已找到最短路径

final[5]

最短路径长度

dist[5]

路径上的前驱

path[5]

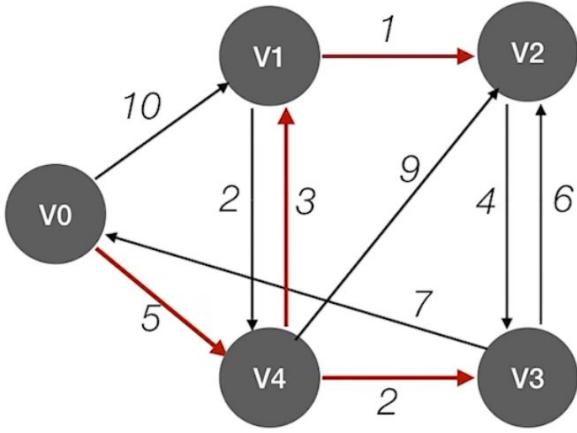
V0	V1	V2	V3	V4
✓	✓	✗	✓	✓



0	8	9	7	5
---	---	---	---	---

-1	4	1	4	0
----	---	---	---	---

- 第 4 轮：循环遍历所有结点，找到还没确定最短路径，且 dist 最小的顶点 V_i ，令 $\text{final}[i] = \text{ture}$



标记各顶点是否已找到最短路径

final[5]

最短路径长度

dist[5]

路径上的前驱

path[5]

V0	V1	V2	V3	V4
✓	✓	✓	✓	✓



0	8	9	7	5
---	---	---	---	---

-1	4	1	4	0
----	---	---	---	---

因为找不到其他顶点了，也不用更新了，算法结束

- 结果：

- V_0 到 v_2 的最短（带权）路径长度为： $\text{dist}[2] = 9$
- 通过 $\text{path}[]$ 可知， V_0 到 V_2 的最短（带权）路径： $V_2 \leftarrow V_4 \leftarrow V_1 \leftarrow V_0$

Floyd 算法求个定点之间最短路径

算法描述

- 定义一个 n 阶方阵 $A^{(-1)}, A^{(0)}, \dots, A^{(n-1)}$ ，其中 $A^{(-1)}[i][j] = \text{arcs}[i][j]$ ，
- 根据递推公式重复 n 次，计算出 $A^{(0)}, \dots, A^{(n-1)}$

- $A^{(k)}[i][j] = \min\{A^{(k-1)}[i][j], A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\}, k = 0, 1, \dots, n - 1$

其中，

◦ $A^{(0)}[i][j]$ 是从顶点 v_i 到 v_j 、中间路径是 v_0 的最短路径的长度

◦ $A^{(k)}[i][j]$ 是从顶点 v_i 到 v_j 、中间顶点的序号不大于 k 的最短路径的长度

- 时间复杂度 $O(|V|^3)$

案例：

- 初始：不允许在其他顶点中转，最短路径如下， $\text{path} = -1$ 表示没有中转点



- 第一步：允许在 V_0 中转，最短路径是 $A^{(0)}$ ，在 $\text{path}^{(0)}$ 中记录中转的点 V_2 到 V_1 之间没有直接通路，但是能经过 V_0 中转的话， V_2 到 V_1 之间的距离从 ∞ 变为 11，公式化是，

若 $A^{(k-1)}[i][j] > A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$

则 $A^{(k)}[i][j] = A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$ ；

$\text{path}^{(k)}[i][j] = k$

否则 $A^{(k)}$ 和 $\text{path}^{(k)}$ 保持原值

其中上述 V_0 在中转后从无穷大变为 11 的路径被找到了：

$A^{(-1)}[2][1] > A^{(-1)}[2][0] + A^{(-1)}[0][1] = 11$ ，修改矩阵的值为 $A^{(0)}[2][1] = 11$, $\text{path}^{(0)}[2][1] = 0$:



- 第二步：允许在 V_0, V_1 中转，求 $A^{(1)}$ 和 $\text{path}^{(1)}$ ，继续根据公式

若 $A^{(k-1)}[i][j] > A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$
 则 $A^{(k)}[i][j] = A^{(k-1)}[i][k] + A^{(k-1)}[k][j];$
 $\text{path}^{(k)}[i][j] = k$
 否则 $A^{(k)}$ 和 $\text{path}^{(k)}$ 保持原值

找到了：

$$A^{(0)}[0][2] > A^{(0)}[0][1] + A^{(0)}[1][2] = 10$$

$$A^{(1)}[0][2] = 10$$

$$\text{path}^{(1)}[0][2] = 1;$$

目前来看，各顶点间的最短路径长度

	V0	V1	V2
V0	0	6	10
V1	10	0	4
V2	5	11	0

$A^{(1)} =$

两个顶点之间的中转点

	V0	V1	V2
V0	-1	-1	1
V1	-1	-1	-1
V2	-1	0	-1

$\text{path}^{(1)} =$

- 第三步：允许在 V_0 、 V_1 、 V_2 中转，求 $A^{(2)}$ 和 $\text{path}^{(2)}$ ，找到了

$$A^{(1)}[1][0] > A^{(1)}[1][2] + A^{(1)}[2][0] = 9$$

$$A^{(2)}[1][0] = 9$$

$$\text{path}^{(2)}[1][0] = 2;$$

目前来看，各顶点间的最短路径长度

	V0	V1	V2
V0	0	6	10
V1	9	0	4
V2	5	11	0

$A^{(2)} =$

两个顶点之间的中转点

	V0	V1	V2
V0	-1	-1	1
V1	2	-1	-1
V2	-1	0	-1

$\text{path}^{(2)} =$

- 根据可知， V_1 到 V_2 最短路径长度为 4，根据可知，完整路径信息为 $V_1 - V_2$
- 根据可知， V_0 到 V_2 最短路径长度为 10，根据可知，完整路径信息为 $V_0 - V_1 - V_2$
- 根据可知， V_1 到 V_0 最短路径长度为 9，根据可知，完整路径信息为 $V_1 - V_2 - V_0$

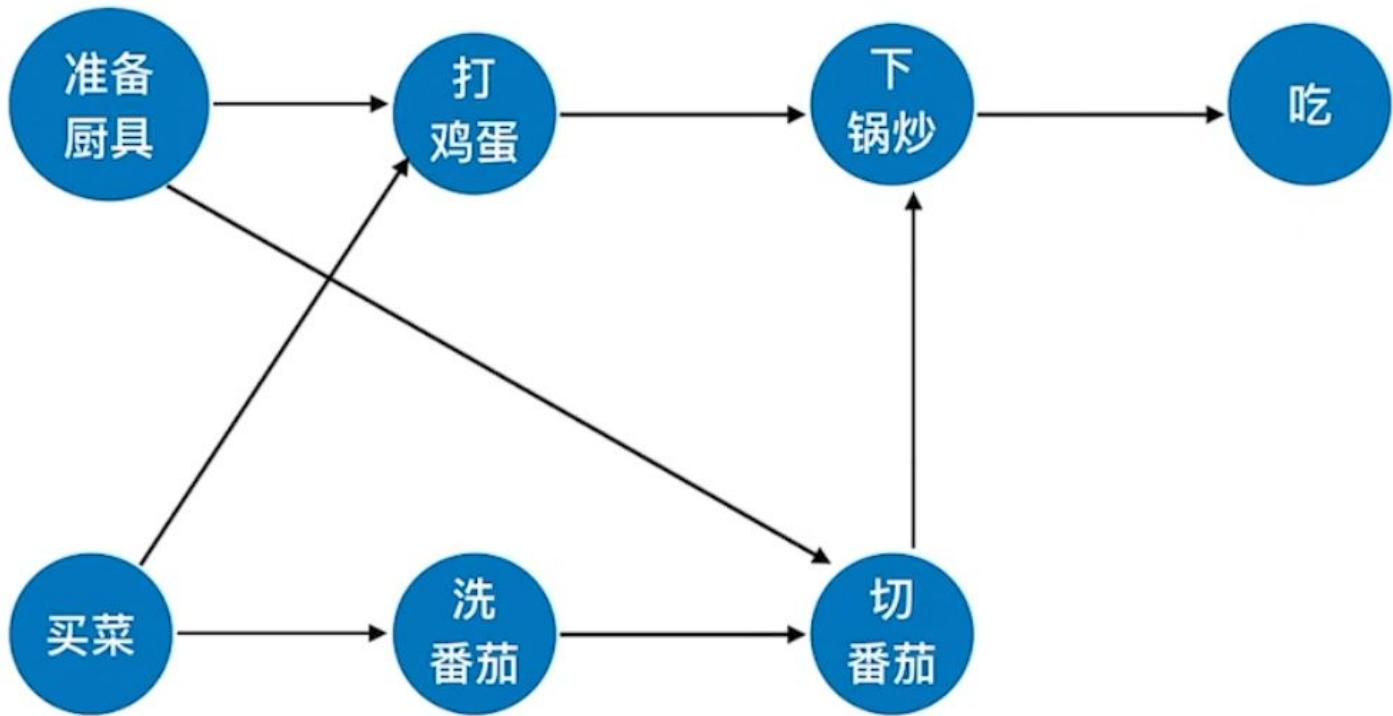
5.4.3 有向无环图描述表达式

有向无环图 DAG

有向无环图是描述含有公共子式的表达式的有效工具，可实现对相同子式的共享，从而节省存储空间

5.4.4 拓扑排序

- AOV网
 - 若用 AOV 表示一个工程，其顶点表示活动，用有向边 $< V_i, V_j >$ 表示活动 V_i 必须先于活动 V_j 进行的这样一种关系，则将这种有向图称为**顶点表示活动的网络**
- 拓扑排序：在图论中，由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序
 - 每个顶点出现且只出现一次
 - 若顶点 A 在序列中排在顶点 B 的前面，则在图中不存在从顶点 B 到顶点 A 的路径
- 拓扑排序算法
 1. 从 AOV 网中选择一个没有前驱的顶点并输出
 2. 从网中删除该顶点和所有以它为起点的有向边
 3. 重复 1-2，直到当前 AOV 网为空
- 时间复杂度 $O(|V| + |E|)$.采用邻接矩阵时间复杂度 $O(|V|^2)$
- 逆拓扑排序
 1. 从 AOV 网中选择一个没有后继的顶点并输出
 2. 从网中删除该顶点和所有以它为重点的有向边
 3. 重复 1-2 直到 AOV 为空
- 案例：

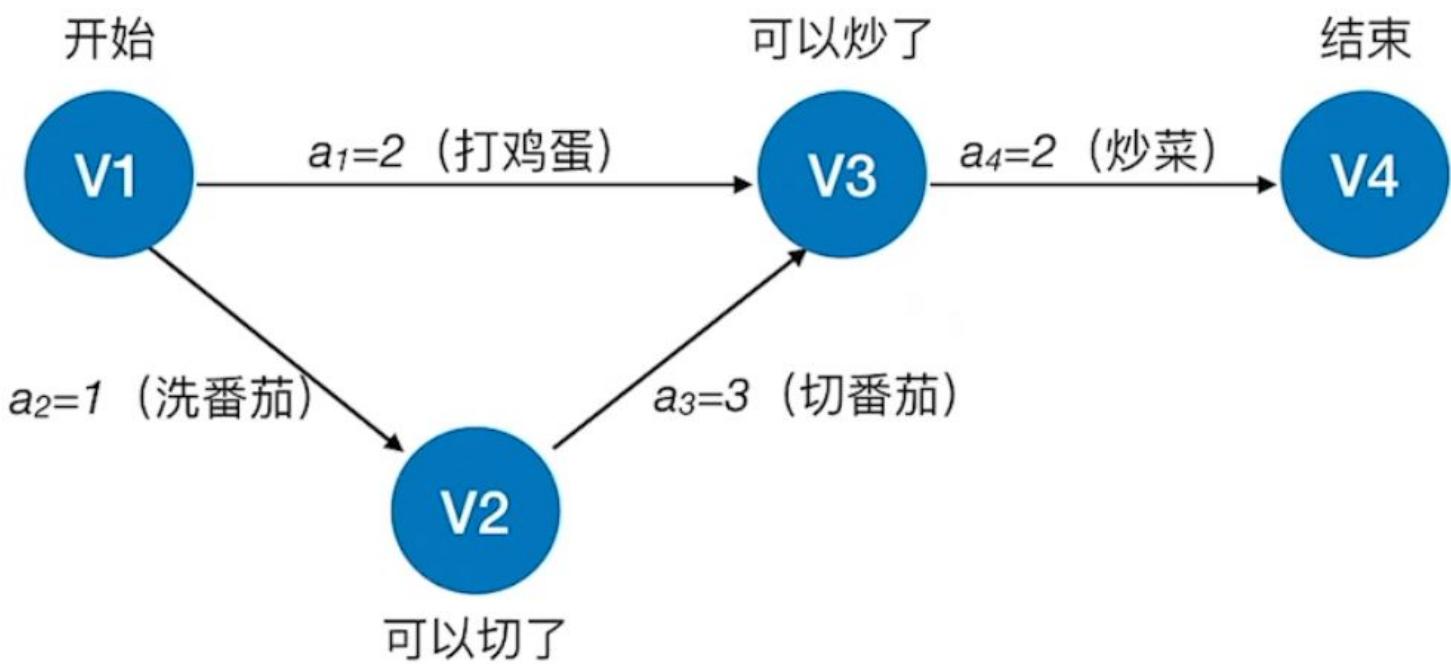


表示“番茄炒蛋工程”的AOV网



5.4.5 关键路径

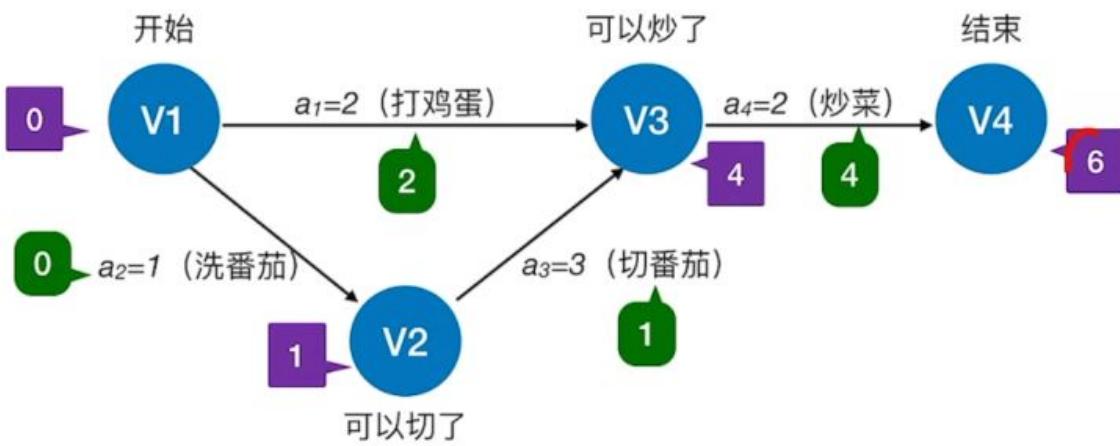
在带权有向图中，以顶点表示事件，以有向边表示活动，以边上的权值表示完成该活动的开销，称之为**用边表示活动的网络**，简称AOE网



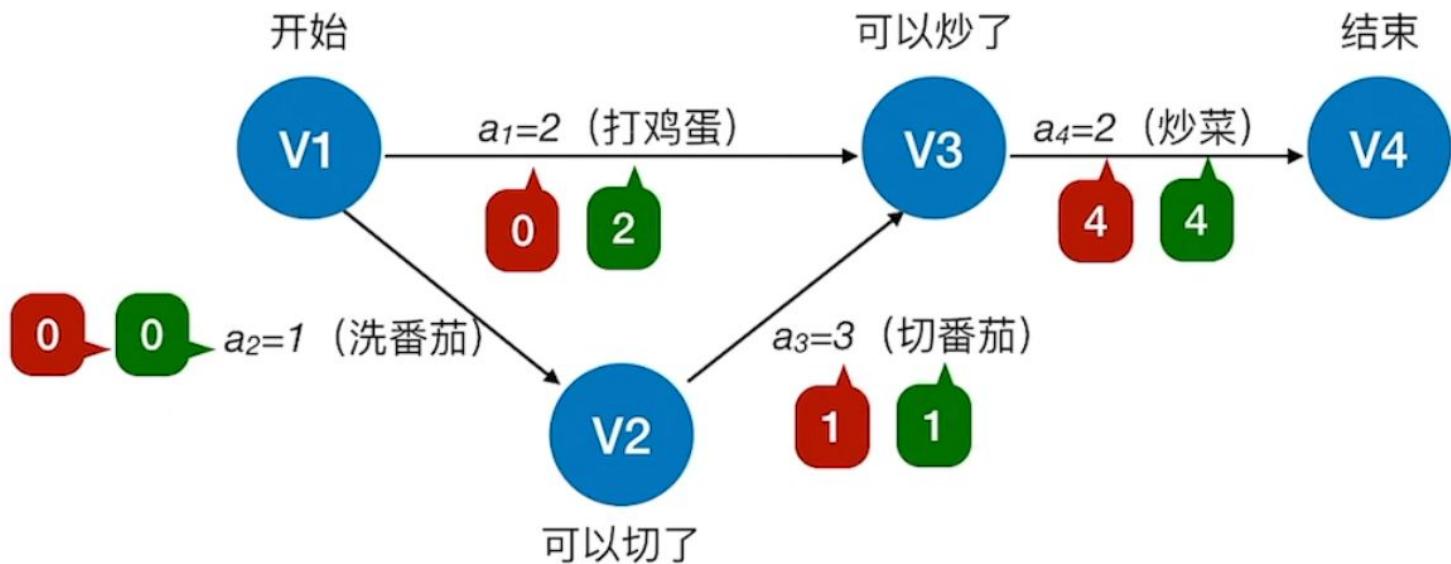
- AOE网 中仅有一个入度为 0 的顶点，称为开始顶点（源点）；只存在一个出度为 0 的顶点，称之为结束顶点（汇点）
- 具有最大路径长度的路径称为**关键路径**，而把关键路径上的活动称为**关键活动**
- 关键路径并不唯一，只提高其中一条关键路径上的关键活动速度不能缩短整个工程的工期。

重要概念：

- 事件 v_i 的最迟发生时间 $v(k)$: 它是指在不推迟整个工程完成的前提下，该事件最迟必须发生的时间（紫色标注）
- 活动 a_i 的最迟开始时间 $l(i)$: 它是指该活动弧的终点所表示事件的最迟发生时间与该活动所需时间之差（绿色标注）



- 活动 a_i 的最早开始时间 $e(i)$: 指该活动弧的起点所表示的事件的最早发生时间 (红色标注)
- 活动 a_i 的时间余量 $d(i) = l(i) - e(i)$, 表示在不增加完成整个工程所需总时间的情况下, 活动 a_i 可以拖延的时间
- 若 $d(i) = 0$ 即 $l(i) = e(i)$ 的活动 a_i 是关键活动
- 由关键活动组成的路径就是关键路径



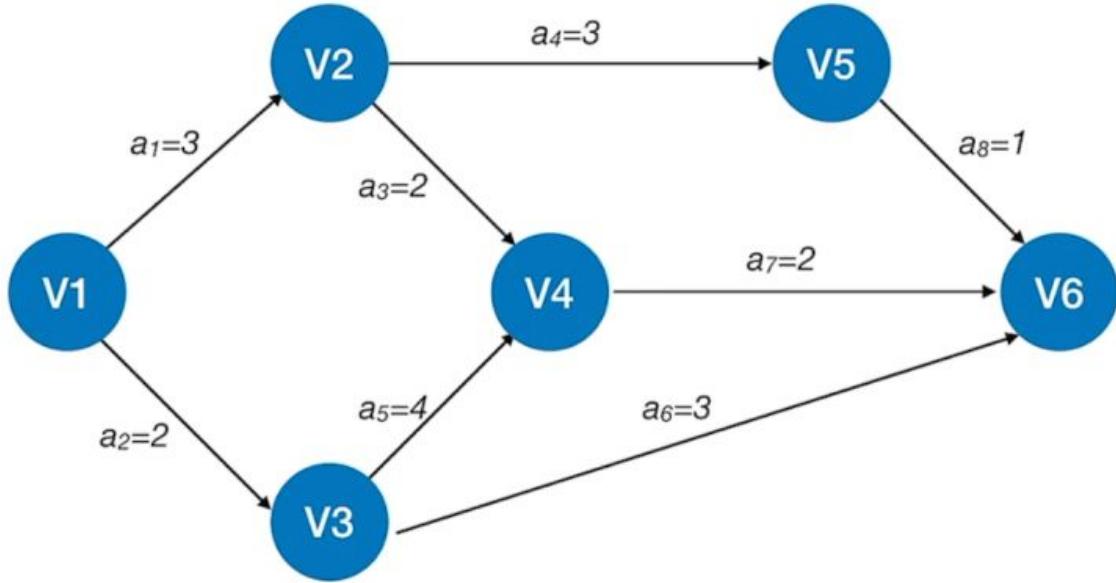
计算步骤

- 事件 v_k 的最早发生时间 $ve(k)$
 - $ve(\text{源点}) = 0$
 - $ve(k) = \max\{ve(j) + \text{Weight}(v_j, v_k)\}$
- 事件 v_k 的最迟发生时间 $vl(k)$
 - $vl(\text{汇点}) = ve(\text{汇点})$
 - $vl(k) = \min\{vl(j) - \text{Weight}(v_k, v_j)\}$
- 活动 a_i 的最早开始时间 $e(i)$
 - 它是指该活动弧的起点所表示的事件最早发生时间。
 - 若边 $< v_k, v_j >$ 表示活动 a_i , 则有 $e(i) = ve(k)$
- 活动 a_i 的最迟开始时间 $l(i)$
 - 它是指该活动弧的终点所表示事件的最迟发生时间与该活动所需时间之差
 - 若边 $< v_k, v_j >$ 表示活动 a_i , 则有 $l(i) = vl(j) - \text{Weight}(v_k, v_j)$
- 求所有活动的时间余量 $d(i) = l(i) - e(i)$, $d(i) = 0$ 的活动 a_i 是关键活动

案例:

1. 求所有事件的最早发生时间 ve

- (1) 计算拓扑排序序列:



$V_1, V_3, V_2, V_5, V_4, V_6$

- (2) 按照序列计算最早发生时间

- $ve(1) = 0$
- $ve(3) = 2$
- $ve(2) = 3$
- $ve(5) = 6$
- $ve(4) = \max\{ve(2) + 2, ve(3) + 4\} = 6$
- $ve(6) = \max\{ve(5) + 1, ve(4) + 2, ve(3) + 3\} = 8$

2. 求所有事件的最迟发生时间 vl

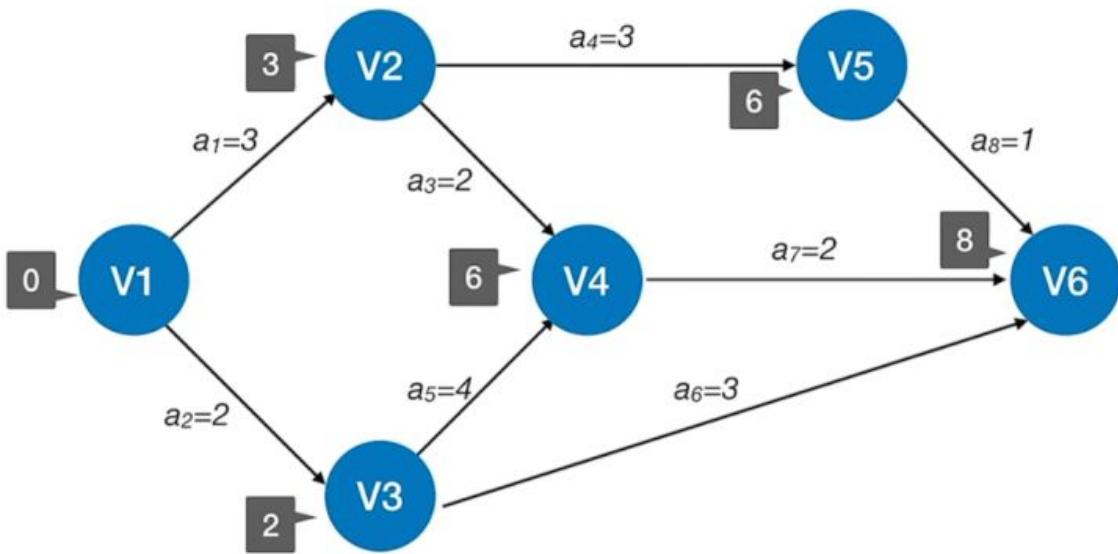
- (1) 计算逆拓扑排序序列: $V_6, V_5, V_4, V_2, V_3, V_1$

- (2) 按照序列计算最迟发生时间

- $vl(6) = ve(6) = 8$
- $vl(5) = vl(6) - 1 = 7$
- $vl(4) = vl(6) - 2 = 6$
- $vl(2) = \min\{vl(5) - 1, vl(4) - 2\} = 4$
- $vl(3) = \min\{vl(4) - 4, vl(6) - 3\} = 2$
- $vl(1) = \min\{vl(2) - 3, vl(3) - 2\} = 0$

3. 求所有活动的最早发生时间 e

	V1	V2	V3	V4	V5	V6
$ve(k)$	0	3	2	6	6	8
$vl(k)$	0	4	2	6	7	8



所有活动的最早发生时间, 若边 $< v_k, v_j >$ 表示活动 a_i , 则有 $e(i) = ve(k)$

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(k)$	0	0	3	3	2	2	6	6

4. 求所有活动的最迟发生时间 l

- 边 $< v_k, v_j >$ 表示活动 a_i , 则有 $l(i) = vl(j) - Weight(v_k, v_j)$
- 得到以下结果:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(k)$	0	0	3	3	2	2	6	6
$l(k)$	1	0	4	4	2	5	6	7

5. 求所有活动的时间余量: $d(i) = l(i) - e(i)$, 得到:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$e(k)$	0	0	3	3	2	2	6	6
$l(k)$	1	0	4	4	2	5	6	7
$d(k)$	1	0	1	1	0	3	0	1

- 关键活动: a_2 、 a_5 、 a_7
- 关键路径: $V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_6$

6. 查找

6.1 查找的概念

- **查找**: 在数据集合中寻找满足某种条件的数据元素的过程称为查找。查找的结果分为**成功**和**失败**
- **查找表**: 用于查找的数据集合称为查找表。对查找表进行的操作一般有四种
 - 查询
 - 查询关键字的其他信息
 - 插入
 - 删除
- 静态查找表: 不涉及插入和删除的查找表
- 关键字: 数据元素中唯一标识该元素的某个数据项的值
- 平均查找长度: $ASL = \sum_{i=1}^n P_i C_i$, P_i 是概率, C_i 是比较次数

6.2 顺序查找和折半查找

6.2.1 顺序查找

一般线性表的顺序查找

```
typedef struct{
    ElemenType *elem;
    int TableLen;
}SSTable;
int Search_Seq(SSTable ST, ElemenType key){
    ST.elem[0] = key; //哨兵
    for(i=ST>TableLen; ST.elem[i]!=key; -i); //从后往前找
    return i; //若表中不存在关键字为key的元素, 将查找到i为0时退出循环
}
```

- 哨兵: 引入它的目的是可以不必判断数组是否会越界。引入哨兵可以避免很多不必要的判断语句, 从而提高程序效率

有序表的顺序查找

- 由于表的关键字是有序的, 查找失败时可以不用比较到表的另一端就能返回失败信息

6.2.2 折半查找

- 基本思想

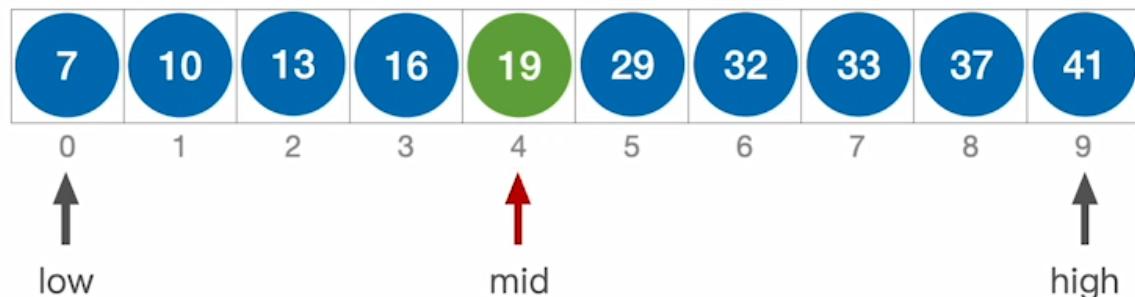
首先将给定的 `key` 与表的中间位置的关键字比较, 成功后返回; 否则根据 `key` 与关键字的大小判断查找左边还是右边
折半查找整个算法中, 关于 `mid` 的取值向上 / 向下需要统一

```
int Binary_Search(SeqList L, ElemenType key){
    int low=0, high=L.TableLen-1, mid;
    while(low<=high){
        mid = (low+high)/2; //取中间位置
        if(L.elem[mid]==key)
            return mid;
        else if(L.elem[mid]>key)
            high = mid-1; //从前半部分继续查找
        else
            low = mid+1; //从后半部分继续查找
    }
    return -1; //查找失败, 返回-1
}
```

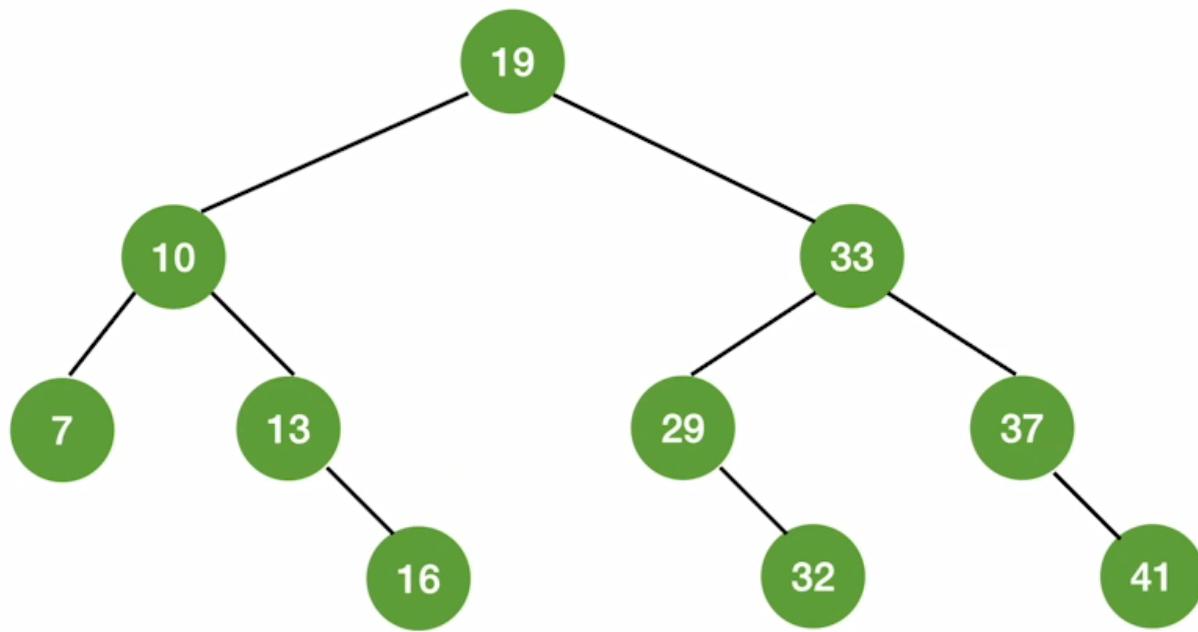
- 折半查找又称二分查找，仅适用于有序的顺序表
- 仅适用于顺序存储结构，不适用于链式存储结构
- 生成判定树

如果当前low和high之间有奇数个元素，则 mid 分隔后，左右两部分元素个数相等

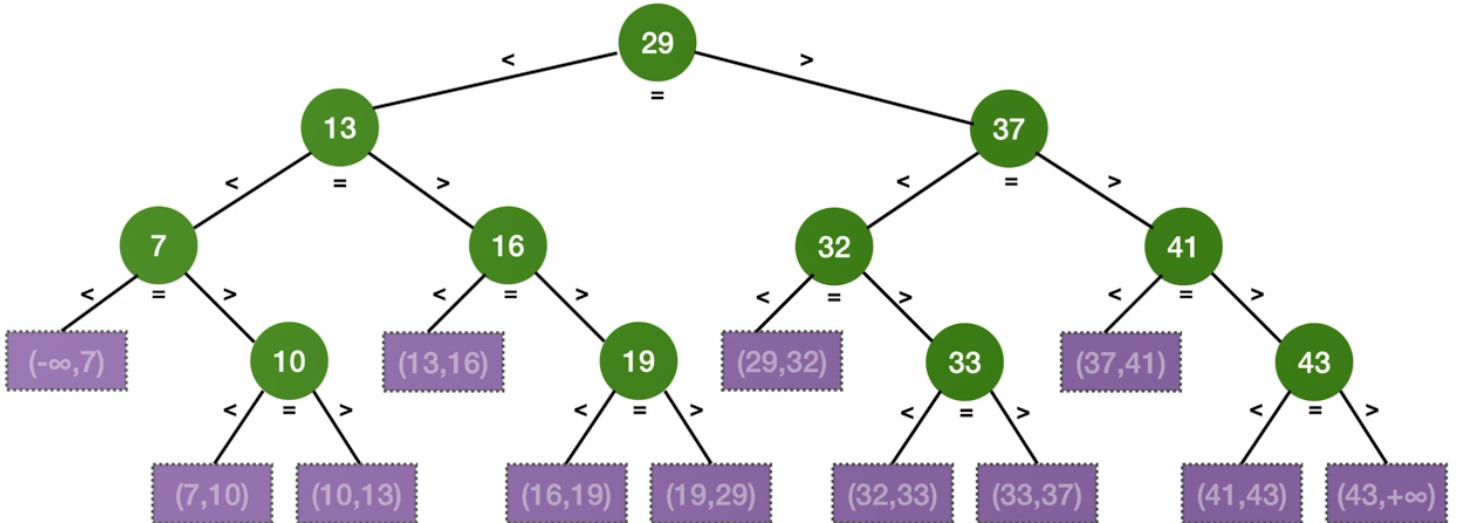
如果当前low和high之间有偶数个元素，则 mid 分隔后，左半部分比右半部分少一个元素



$$mid = \lfloor (low + high)/2 \rfloor$$



判定树特性：



$$ASL_{\text{成功}} = (1*1 + 2*2 + 3*4 + 4*4)/11 = 3$$

$$ASL_{\text{失败}} = (3*4 + 4*8)/12 = 11/3$$

- 是平衡二叉树
- 有 n 个圆形结点代表原数据或成功结点, $n + 1$ 个方框结点代表不成功结点
- 每个圆形结点都不是叶子结点, 一定有方框子结点
- 元素个数为 n 时, 树高度为 $h = \lceil \log_2(n + 1) \rceil$

与 折半查找判定树 的高度 h 有关。高度越小, 查找效率越高

- 最好情况, 平均查找长度 = $O(1)$
- 最坏情况, 平均查找长度 = $O(\log_2 n)$
- 则 时间复杂度 = $O(\log_2 n)$

6.2.3 分块查找

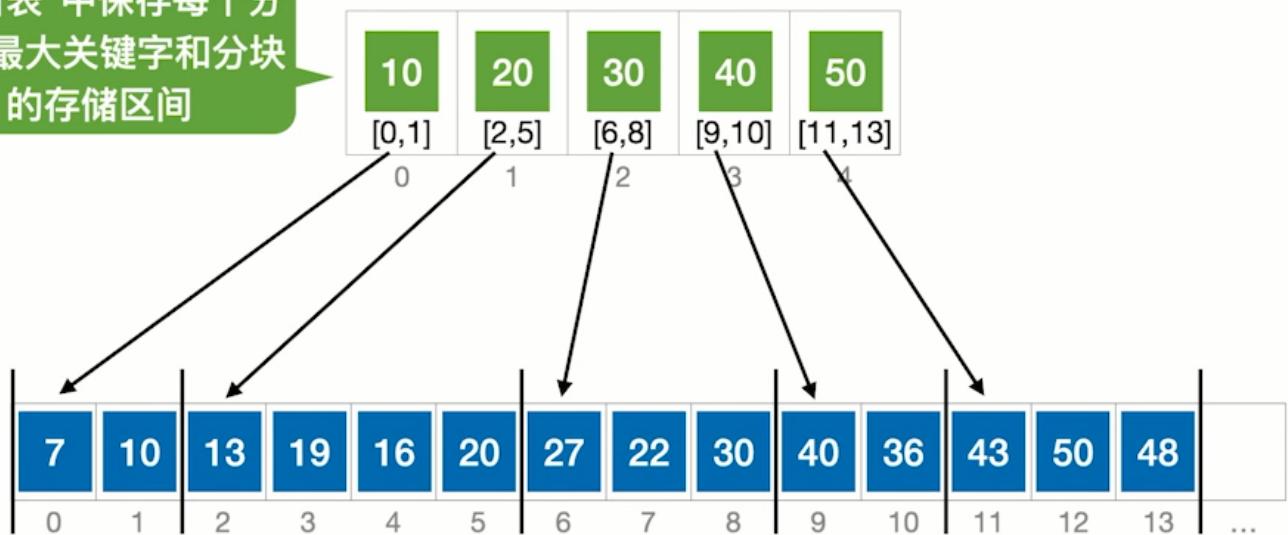
分块查找, 又叫 索引顺序查找。

算法思想：用一个 索引表 给数据归类。

算法过程：

- ① 在 索引表 中确定待查记录所属的分块 (可顺序、可折半)
- ② 在 块内顺序查找

“索引表”中保存每个分块的最大关键字和分块的存储区间

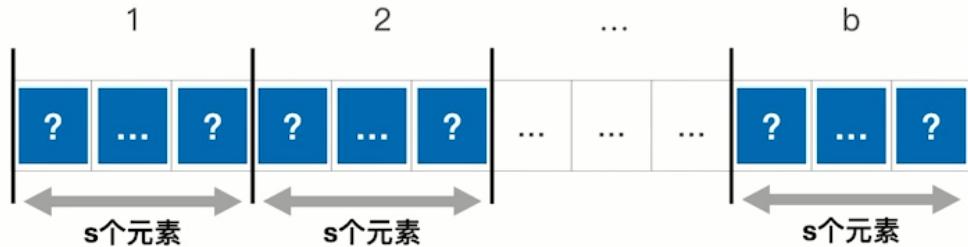


特点：块内无序、块间有序

用折半查找索引表：

若索引表中不包含目标关键字，则折半查找索引表 最终停在 $low > high$ ，要在 low 所指分块中查找

查找效率分析



假设，长度为n的查找表被均匀地分为b块，每块s个元素

设索引查找和块内查找的平均查找长度分别为 L_I 、 L_S ，则分块查找的平均查找长度为

$$ASL = L_I + L_S$$

用顺序查找查索引表，则 $L_I = \frac{(1+2+\dots+b)}{b} = \frac{b+1}{2}$, $L_S = \frac{(1+2+\dots+s)}{s} = \frac{s+1}{2}$

$$\text{则 } ASL = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2+2s+n}{2s}$$

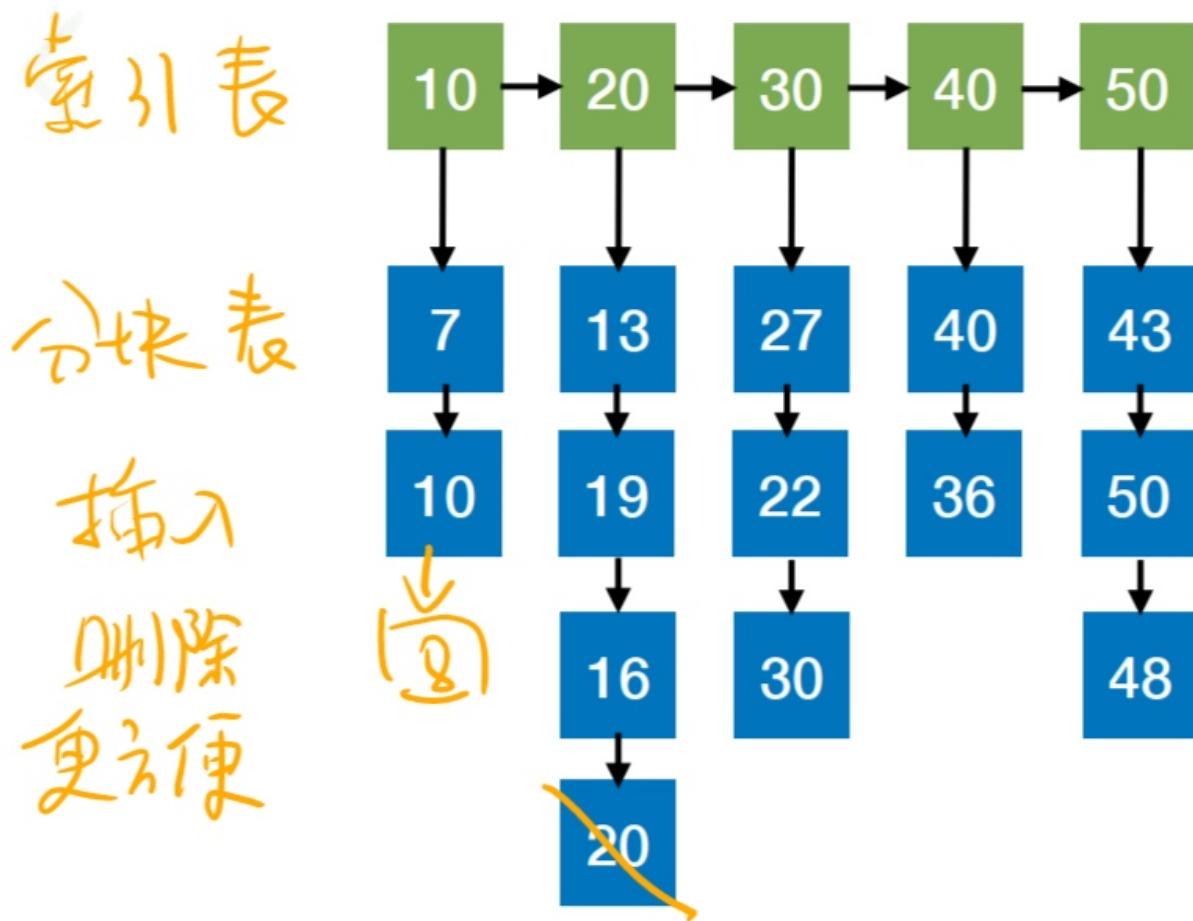
若 $n=10000$, 则
 $ASL_{\min}=101$

用折半查找查索引表，则 $L_I = \lceil \log_2(b+1) \rceil$, $L_S = \frac{(1+2+\dots+s)}{s} = \frac{s+1}{2}$
则 $ASL = \lceil \log_2(b+1) \rceil + \frac{s+1}{2}$

分块查找的优化

上面的分块查找对插入删除不友好。

改进：索引表为顺序表，查找表为链表。



6.2.4 时间复杂度评价

查找算法	$ASL_{\text{成功}}$	$ASL_{\text{失败}}$
顺序查找-无序表	$\frac{n+1}{2}$	$n + 1$
顺序查找-有序表	$\frac{n+1}{2}$	$\frac{n}{2} + \frac{n}{n+1}$
折半查找	$\text{sum}(\text{圆形结点} * \text{对应层数})/n$	$\text{sum}(\text{方结点} * \text{对应层数} - 1)/(n + 1)$
分块查找	$ASL = L_I + L_S$	-

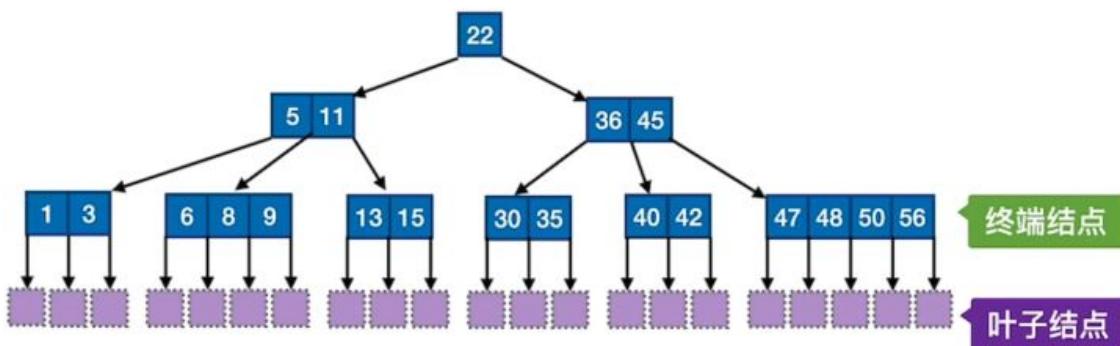
6.3 B 树和 B + 树

6.3.1 B 树及其基本操作

B 树又称**多路平衡查找树**，B 树中所有结点的孩子个数的最大值称为 B 树的阶，通常用 m 表示。

B 树是所有结点的平衡因子都等于 0 的多路平衡查找树

定义



- 树中每个结点至多有 m 棵子树，即至多含有 $m-1$ 个关键字
- 若根结点不是终端节点，则至少有两棵子树
- 除根结点外的所有非叶结点至少有 $\lceil m/2 \rceil$ 棵子树，即至少含有 $\lceil m/2 \rceil - 1$ 个关键字
- 所有非叶子结点的结构如下：

$n | P_0 | K_1 | P_1 | K_2 | P_2 | \dots | K_n | P_n$

其中， K_i 为结点的关键字， P_i 为指向子树根结点的指针，且：

- 指针 P_{i-1} 所指子树中所有结点的关键字均小于 K_i
- 指针 P_i 所指子树中所有结点的关键字均大于 K_i
- 结点的孩子个数等于该节点中关键字个数加 1

- 所有的叶节点都出现在同一层次上，并且不带信息，称为外部结点

B 树的高度

B 树的高度不包括最后的外部结点那一层

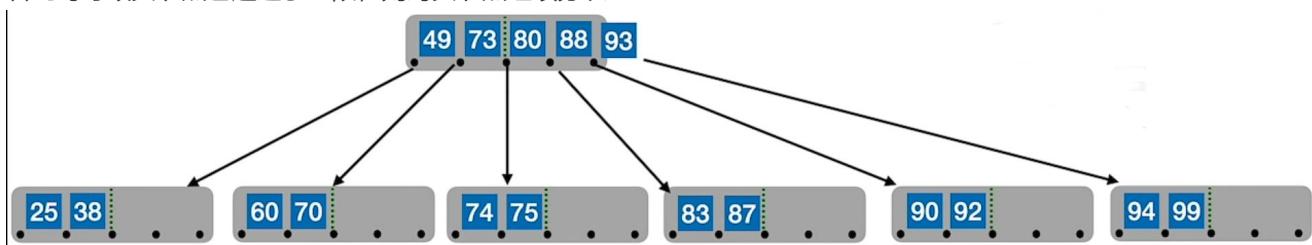
$$\log_m(n+1) \leq h \leq \log_{\lceil m/2 \rceil}((n+1)/2) + 1$$

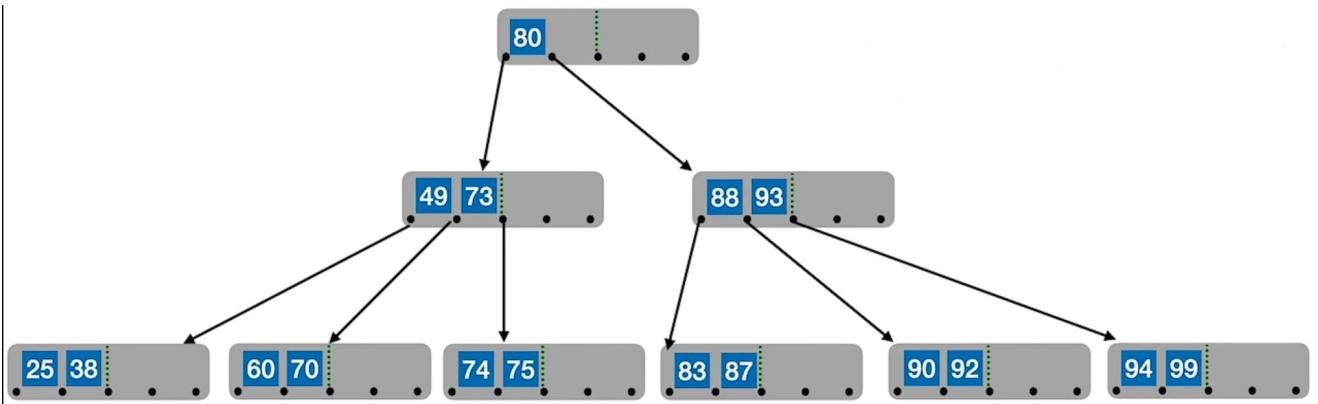
B 树的查找

- 在 B 树中找结点，在磁盘中进行
- 在结点找关键字，在内存中进行

B 树的插入

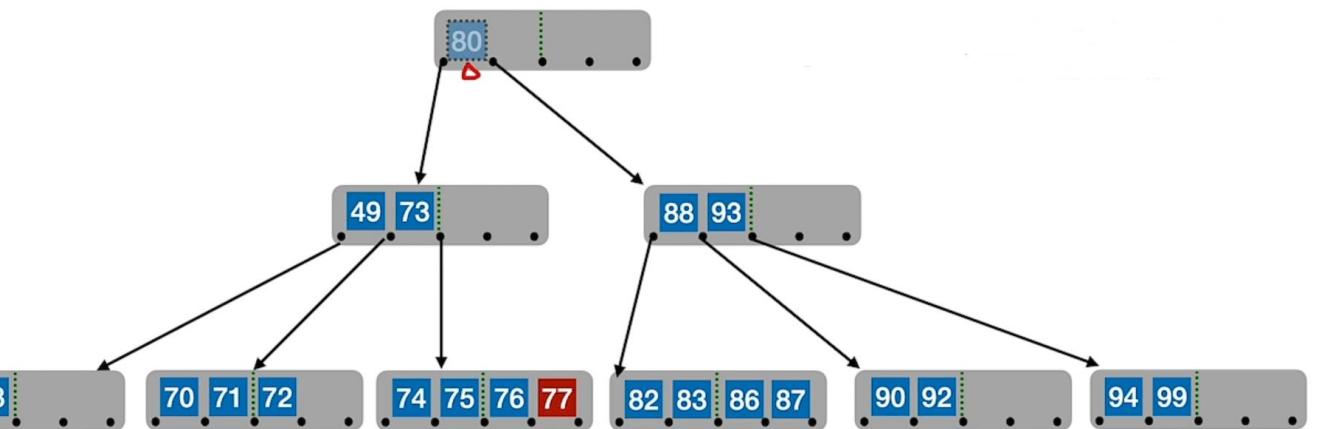
1. 定位：利用 B 树的查找算法，找出插入该关键字的最低层中的某个非叶结点
2. 插入：在 B 树中，每个失败结点的关键字个数都在区间 $[\lceil m/2 \rceil - 1, m - 1]$ 内。插入后的结点关键字个数小于 m ，可以直接插入。如果插入后关键字个数大于 $m-1$ ，必须进行分裂
 - 分裂方法是
 - 取一个新结点，在插入 key 后的原结点，从中间位置将其中的关键字分为两部分
 - 左部分包含的关键字放在原结点中，右部分包含的关键字放到新结点中
 - 中间位置 1 的节点插入原节点的父节点。
 - 若此时导致父节点也超过了上限，则对父节点继续分裂





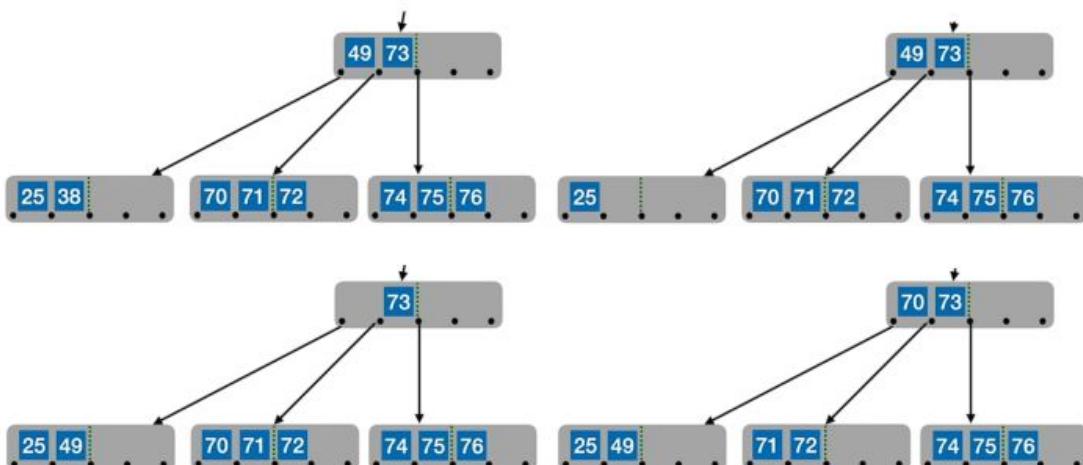
B 树的删除

- 当被删关键字 k 不在终端结点时，可以用 k 的前驱或后继 k' 替代 k ，然后在相应的结点中删除 k' 。关键字 k' 必定落在某个终端节点中，则转换成了被删关键字在终端结点中的情形

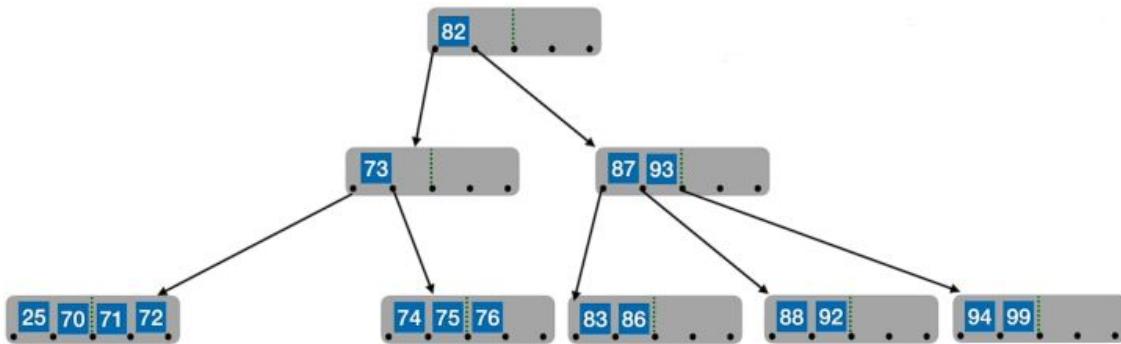
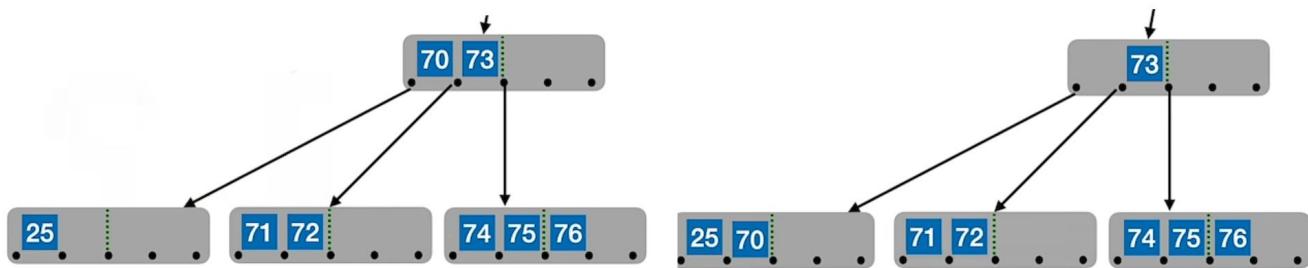


- 当被删关键字 k 在终端结点中时

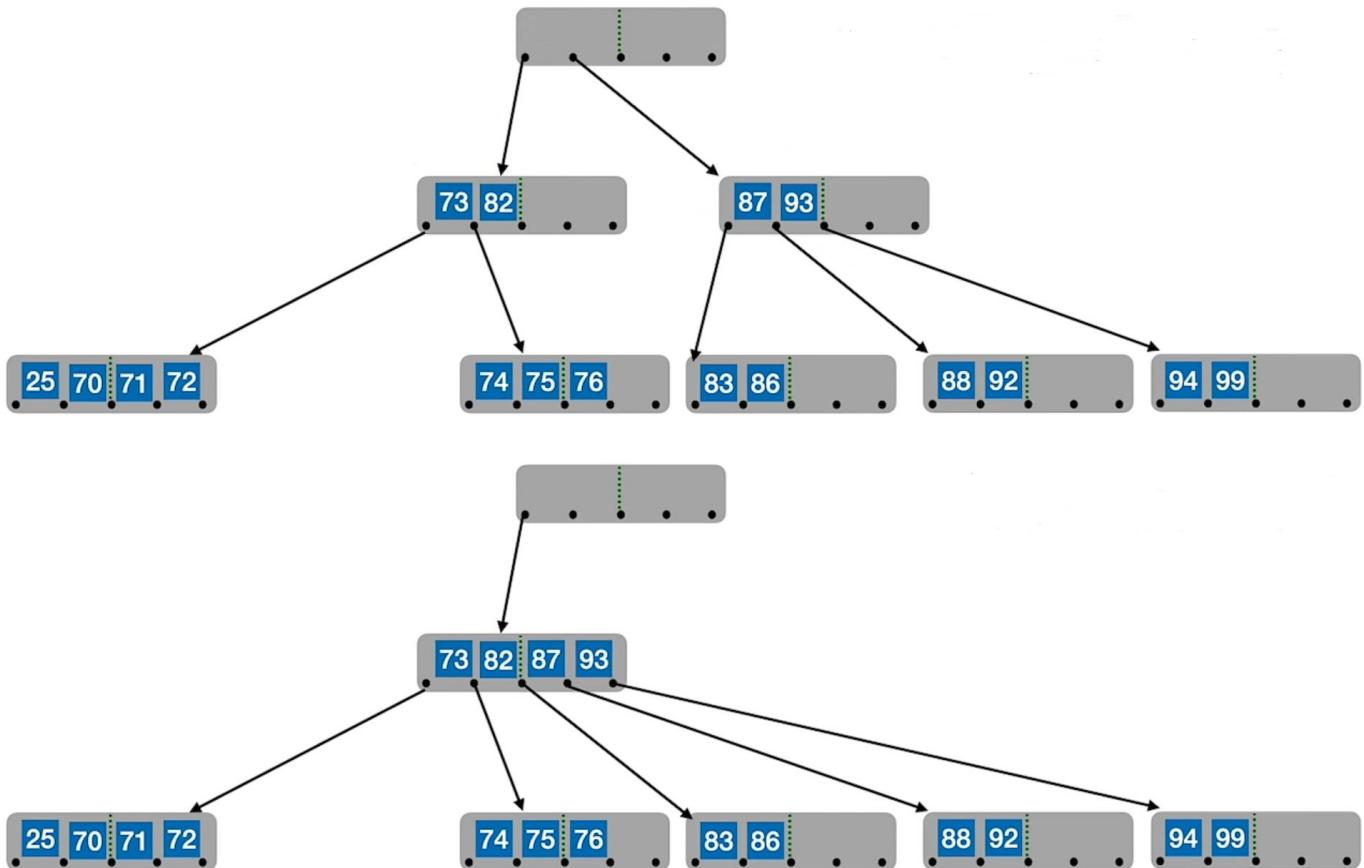
- 直接删除关键字**: 若被删除关键字所在结点的关键字个数 $> \lceil m/2 \rceil$ ，表明删除该关键字后仍满足 B 树的定义，则直接删除该关键字
- 兄弟够借**: 若被删除关键字所在结点的关键字个数 $= \lceil m/2 \rceil - 1$ ，且与此节点相邻的右（左）兄弟节点的关键字个数 $\geq \lceil m/2 \rceil$ ，则需要调整该节点、右（左）兄弟结点及其双亲结点（父子换位法），以达到新的平衡

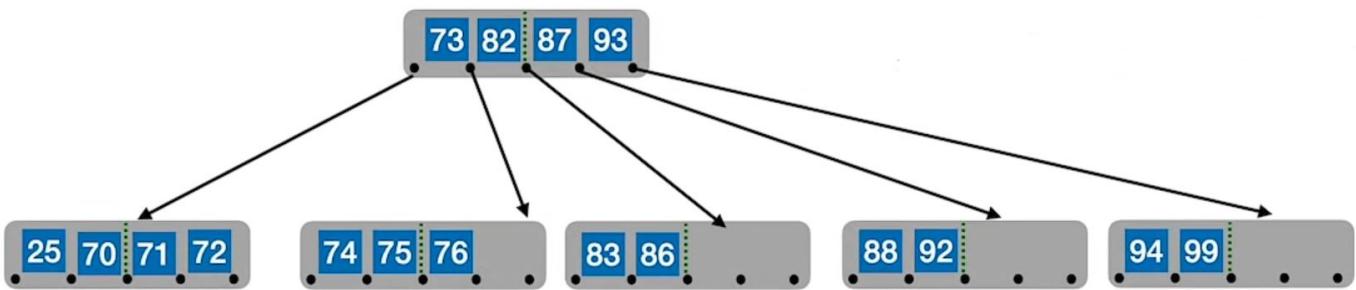


- 兄弟不够借**: 若被删除关键字所在结点的关键字个数 $= \lceil m/2 \rceil - 1$ ，且与此节点相邻的右（左）兄弟节点的关键字个数均 $< \lceil m/2 \rceil - 1$ ，则将关键字删除后与左（或右）兄弟结点及双亲结点中的关键字进行合并。



在合并过程中，双亲结点中的关键字个数会减 1。若其双亲结点是根结点且关键字个数减少至 0，则直接将根结点删除，合并后的新结点成为根；若其双亲结点不是根结点，且关键字减少超过下限，在继续合并操作。

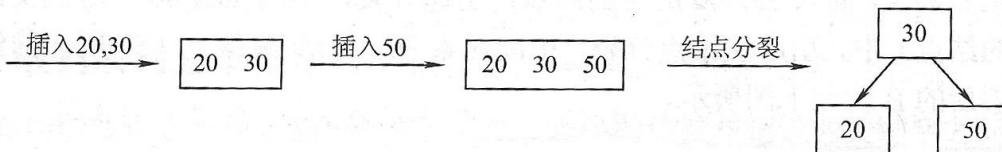




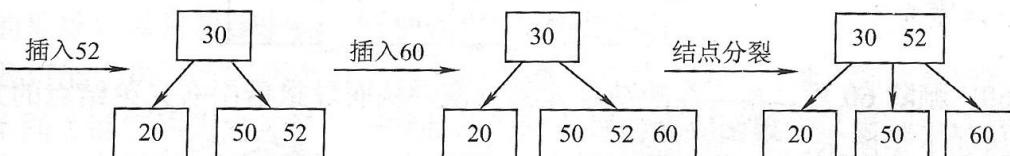
B树的构造及操作案例

1. 解答：

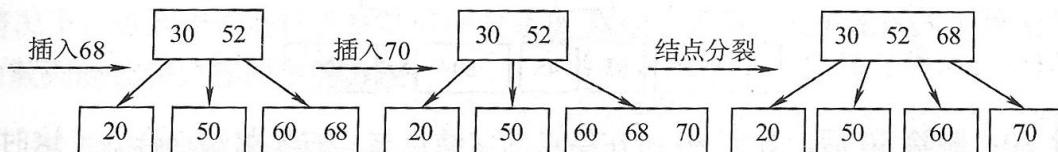
$m = 3$, 因此除根结点外, 非叶子结点关键字个数为 $1 \sim 2$ 。



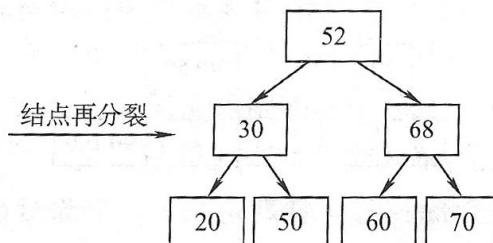
如上图所示, 首先插入 $20, 30$, 结点内关键字个数不超过 $\lceil m/2 \rceil = 2$, 不会引起分裂; 插入 50 , 由于插入 $20, 30$ 所在的结点, 引起分裂, 结点内第 $\lceil m/2 \rceil$ 个关键字 30 上升为父结点。



如上图所示, 插入 52 , 插入 50 所在的结点, 不会引起分裂; 继续插入 60 , 插入 $50, 52$ 所在的结点, 引起分裂, 52 上升到父结点中, 不会引起父结点的分裂。

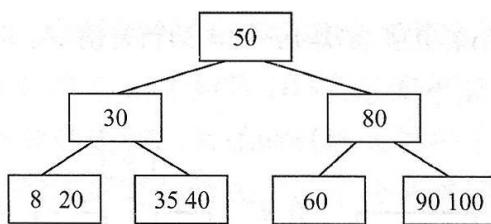


如上图所示, 插入 68 , 插入 60 所在的结点, 不会引起分裂; 继续插入 70 , 插入 $60, 68$ 所在的结点, 引起分裂, 68 上升为新的父结点, 68 上升到 $30, 52$ 所在的结点后, 会继续引起该结点分裂, 故 52 上升为新的根结点。最后得到的 B 树如下图所示。

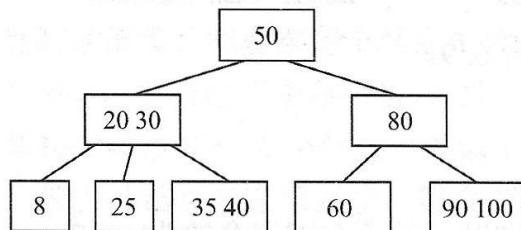


2. 解答：

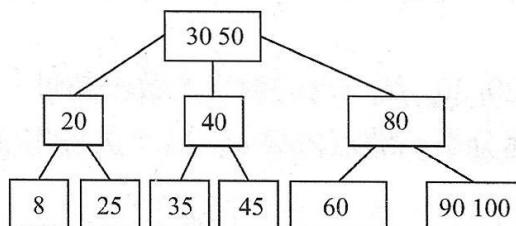
1) 插入 90 : 将 90 插入 100 所在的结点, 插入 90 后该结点中的元素个数不超过 $\lceil 3/2 \rceil = 2$, 不会引起结点的分裂, 插入后的 B 树如下图所示。



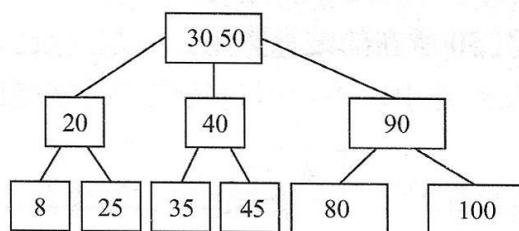
- 2) 插入 25: 将 25 插入 8, 20 所在的结点, 插入后结点内的元素个数为 3, 引起分裂。故将结点内的中间元素 20 上升到父结点中, 此时父结点中的元素个数为 2 (元素 20 和 30), 不会引起继续分裂, 插入 25 后的 B 树如下图所示。



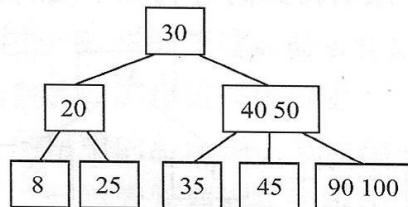
- 3) 插入 45: 将 45 插入 35, 40 所在的结点, 引起分裂, 中间元素 40 上升到父结点 (20, 30 所在的结点) 中, 引起父结点分裂, 中间元素 30 上升到父结点 (50 所在的结点) 中, 两次分裂后的 B 树如下图所示。



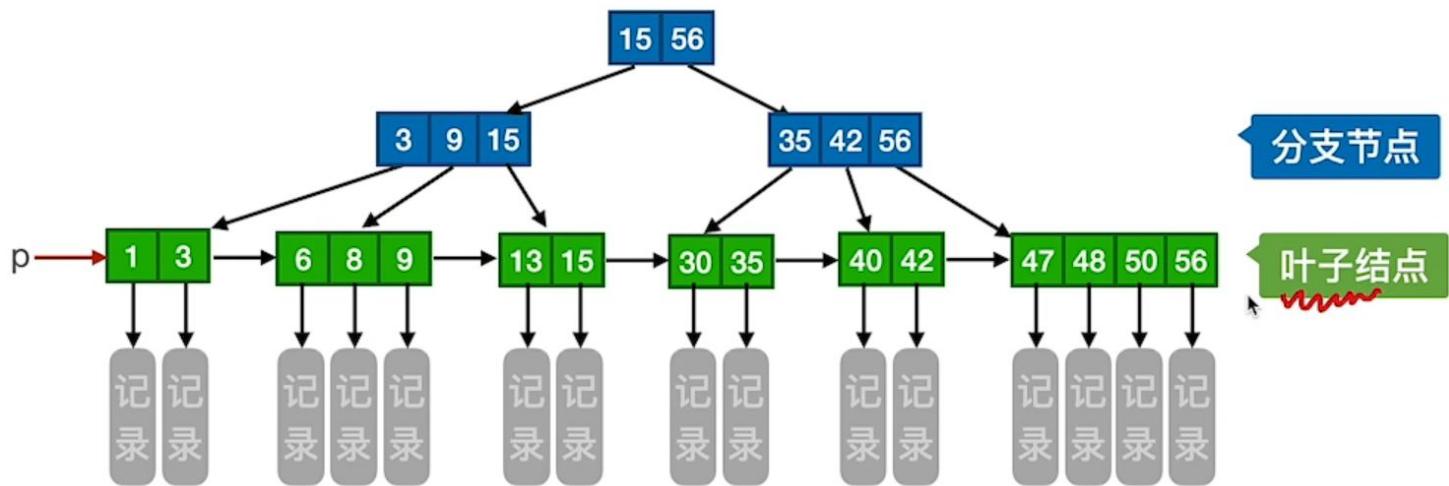
- 4) 删除 60: 删除 60 后, 其所在的结点元素为空, 从而导致借用右兄弟结点的元素, 调整后的 B 树如下图所示。



- 5) 删除 80: 删除 80 后, 导致 80 所在结点的父结点与其右兄弟结点合并, 这时父结点元素个数为 0, 再次对父结点进行调整。将 50 与 40 合并成一个新结点, 则 90, 100 所在结点为这个结点的子结点。从而构造的 B 树如下图所示。注意, 这次调整的过程实际上包含多次调整过程, 希望读者对照考点讲解中的删除过程仔细思考。



6.3.2 B + 树的基本概念



注：以上是一棵4阶B+树

- 每个分支结点最多有 m 棵子树
- 非叶根结点至少有两棵子树，其他每个分支结点至少有 $\lceil m/2 \rceil$ 棵子树
- 结点的子树个数与关键字个数相等
- 所有叶结点包含全部关键字及指向相应记录的指针，叶节点中将关键字按大小顺序排列，并且相邻叶节点按大小顺序相互链接起来
- 所有分支结点中仅包含它的各个子节点中关键字的最大值及指向其子结点的指针

在 B + 树中查找时，非叶结点上的关键字值等于查找值时并不停止，而是继续往下找，直到叶结点上的该关键字为止。无论成功与否，每次查找都是一条从根结点到叶结点的路径

注意B+树存储的记录在叶子节点，路径是索引，不包含记录，相关的插入和删除与B树类似

6.3.3 B 树 VS B+ 树

说明	B树	B+树
关键字个数为n的结点的子树个数	$n-1$	n
结点关键字个数n范围	$\lceil m/2 \rceil \leq n \leq m$	$\lceil m/2 \rceil - 1 \leq n \leq m - 1$
-	-	叶结点包含信息，所有非叶结点仅起索引作用，非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址
-	叶结点包含的关键字和其他结点包含的关键字是不重复的	叶节点包含了全部关键字，即在非叶结点中出现的关键字也会出现在叶结点中

6.4 散列表

6.4.1 散列表的基本概念

- 散列函数：一个把查找表中关键字映射成该关键字对应的地址的函数，记为 $\text{Hash}(\text{key}) = \text{Addr}$
- 冲突：散列函数把两个或两个以上不同关键字映射到同一地址的现象
- 同义词：引起冲突的关键字
- 散列表：根据关键字而直接进行访问的数据结构。散列表建立了关键字和存储地址之间的一种直接映射关系

6.4.2 散列函数的构造方法

散列函数的要求

- 定义域包含全部关键字，值域依赖于散列表的大小或地址范围
- 散列函数计算出的地址应该能等概率、均匀的分布在整个地址空间中，减少冲突发生
- 尽可能简单，能够快速计算出散列地址

常见构造函数	公式	评价
直接定地法	$H(\text{key}) = \text{key}$ 或 $H(\text{key}) = a \times \text{key} + b$	最简单，不会产生冲突。适合关键字的分布基本连续的情况
除留余数法	$H(\text{key}) = \text{key \% } p$, p 为不大于散列表表长 m 但最接近或等于 m 的质数	
数字分析法	设关键字是 n 进制数，选取数码分布比较均匀的若干位作为散列地址	适合于一致的关键字集合，若更换了关键字，则需要重新构造新的散列函数
平方取中法	取关键字对的平方值的中间几位作为散列值	适用于关键字的每位取值都不均匀或均小于散列地址所需的位数

6.4.3 处理冲突的方法

- 开放定址法， $H_i = (H(\text{key}) + d_i) \% m$, m 表示散列表表长， d_i 为增量序列

开放定址法	d_i	补充说明
线性探测法	$0, 1, 2, \dots, m - 1$	可能出现大量元素在相邻地址上聚集，降低查找效率
平方探测法	$0^2, 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$	散列表长度 m 必须是一个可以表示成 $4k+3$ 的素数
再散列法	$i * \text{Hash}_2(\text{key})$	i 是冲突的次数
伪随机序列法	$d_i == \text{随机序列}$	

- 拉链法
把所有的同义词存储在一个线性链表中，这个线性链表由其散列地址唯一标识

6.4.4 散列查找及性能分析

查找过程

- 初始化 $\text{Addr} = \text{Hash}(\text{key})$
- 检测查找表中地址为 Addr 的位置上是否有记录，若无记录，返回查找失败；若有记录，比较它与 key 的值，若相等，则返回查找成功的标志，否则执行步骤 3

3. 用给定的处理冲突方法计算“下一个散列地址”，并将 Addr 置为此地址，转入步骤 2

散列表的查找效率取决于散列函数、处理冲突的方法和装填因子

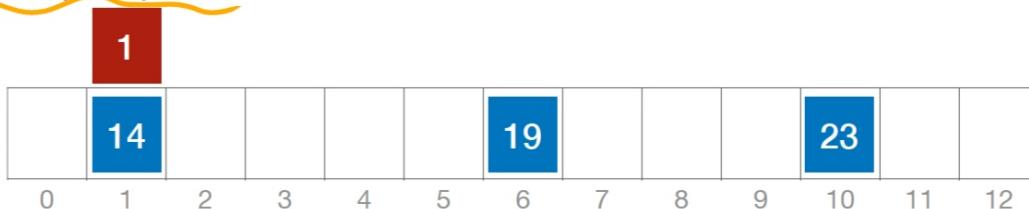
- 装填因子 α 定义为一个表的装满程度

$$\alpha = \frac{\text{表中记录数}n}{\text{散列表长度}m}$$

- 线性探测

例：有一堆数据元素，关键字分别为 {19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数

$$H(key) = key \% 13$$



$$19 \% 13 = 6$$

$$14 \% 13 = 1$$

$$23 \% 13 = 10$$

$$1 \% 13 = 1$$

若不同的关键字通过散列函数映射到同一个值，则称它们为“同义词”
通过散列函数确定的位置已经存放了其他元素，则称这种情况为“冲突”
1 和 14 同义
1 和 14 冲突

①线性探测法—— $d_i = 0, 1, 2, 3, \dots, m-1$ ；即发生冲突时，每次往后探测相邻的下一个单元是否为空

19%13=6——1次	27%13=1——4次	55%13=3——3次
14%13=1——1次	68%13=3——1次	11%13=11——1次
23%13=10——1次	20%13=7——1次	10%13=10——3次
1%13=1——2次	84%13=6——3次	79%13=1——9次

$$ASL_{\text{成功}} = \frac{1 + 1 + 1 + 2 + 4 + 1 + 1 + 3 + 3 + 1 + 3 + 9}{12} = 2.5$$

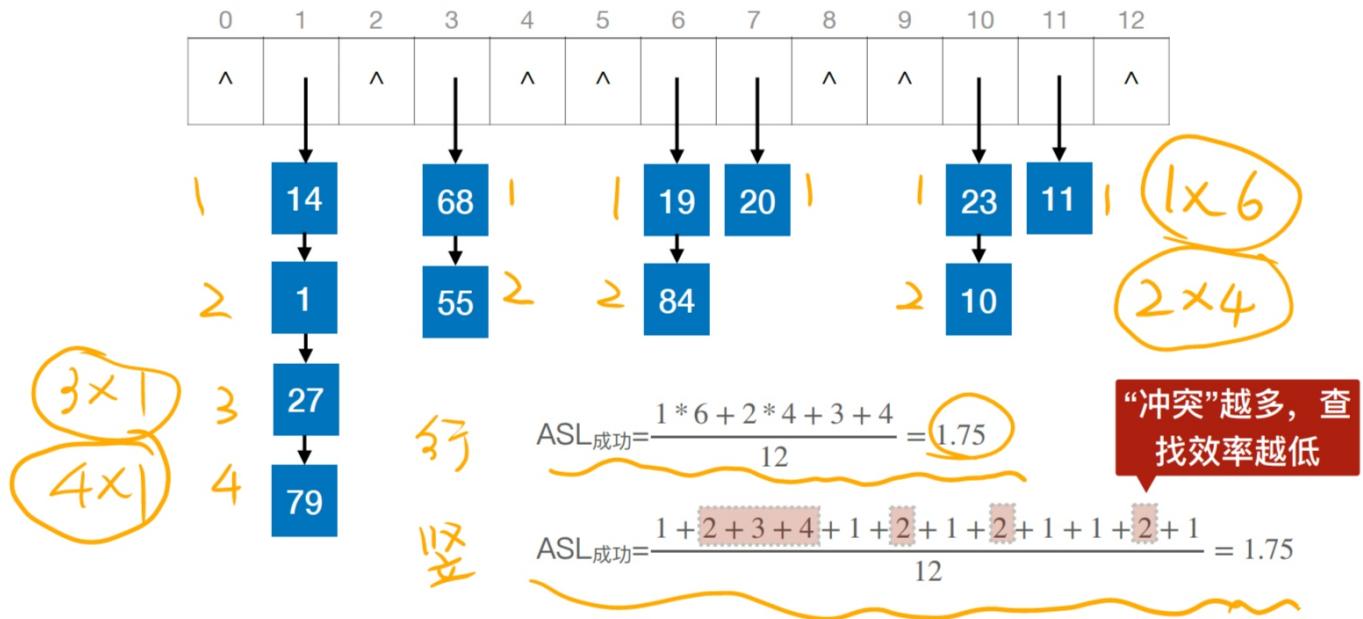
工道考研 | GONGDAOKAOGAN.COM

$$ASL_{\text{失败}} = \frac{1 + 13 + 12 + 11 + 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2}{13} = 7$$

初次探测的地址 H_0 只
有可能在 [0, 12]

- 链地址法

例：有一堆数据元素，关键字分别为 {19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79}，散列函数 $H(key)=key \% 13$



7. 排序

7.1 排序的基本概念

- 排序：就是重新排列表中的元素，使表中的元素满足按关键字有序的过程
- 算法的稳定性：在排序之前关键字相同的元素，在排序后相对位置不变的排序算法是稳定的
- 内部排序：排序期间元素全部存放在内存中的排序
- 外部排序：排序期间元素无法全部同时同放在内存中，必须在排序的过程中根据要求不断的在内、外存之间移动的排序
- 可将排序算法分为：插入排序、交换排序、选择排序、归并排序和基数排序五大类

7.2 插入排序

基本思想：每次讲一个待排序的记录按其关键字大小插入前面已排好序的子序列，直到全部记录插入完成

7.2.1 直接插入排序

要将 $L(i)$ 插入已有序的子序列 $L[1 \dots i-1]$ ，需要执行以下操作

- 查找出 $L(i)$ 在 $L[1 \dots i-1]$ 中的插入位置 k
- 将 $L[k \dots i-1]$ 中的所有元素依次后移一个位置
- 将 $L(i)$ 复制到 $L(k)$

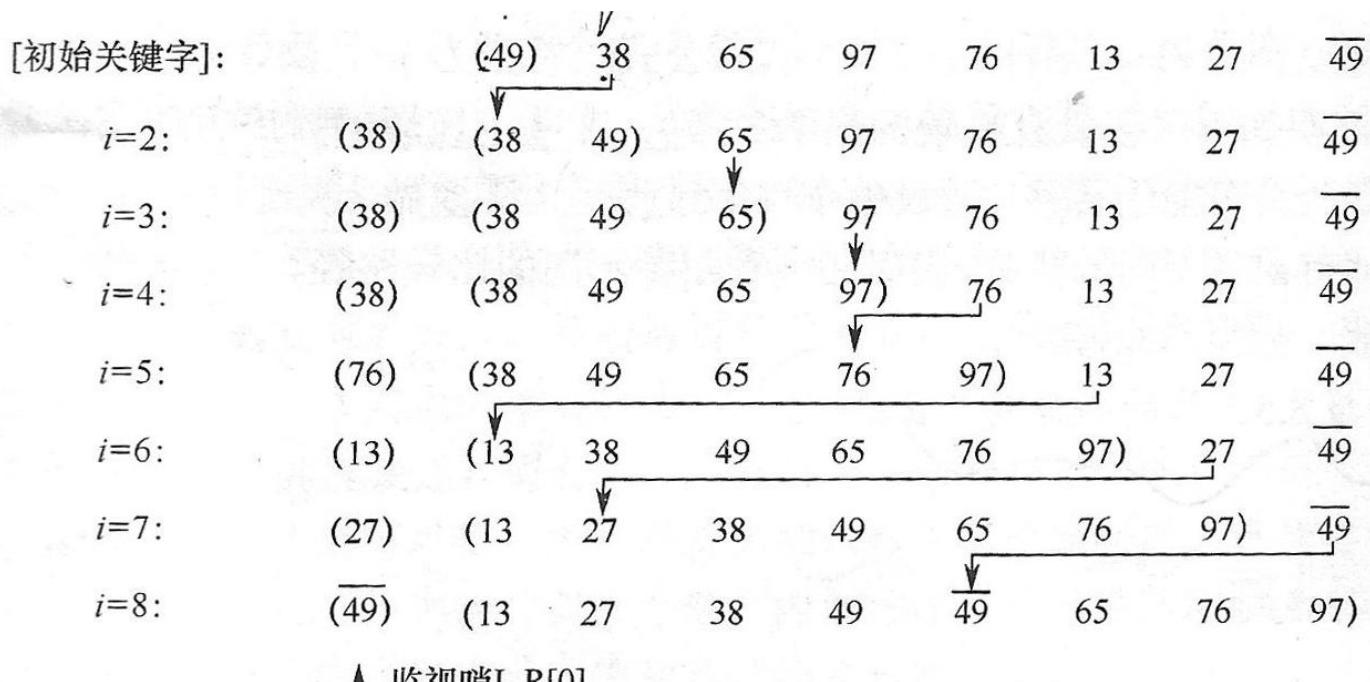


图 8.1 直接插入排序示例

```
void InsertSort(ElemType A[], int n){
    int i, j;
    for(i=2; i<=n; i++) //依次将A[2]...A[n]插入到前面已排序的序列
        if(A[i]<A[i-1]) { //若A[i]小于前驱，则将其插入前面的有序表
            A[0] = A[i]; //哨兵
            for(j=i-1; A[0]<A[j]; --j) //从i-1开始比较，比较一次，向后移动一次
                A[j+1] = A[j];
            A[j+1] = A[0]; //找到插入位置，赋值
        }
}
```

7.2.2 折半插入排序

- 查找有序子表时用折半查找来实现
- 确定待插入位置后，同意以地向后移动元素

```
void InsertSort(ElemType A[], int n){
    int i, j, low, high, mid;
    for(i=2; i<=n; i++) { //依次将A[2]...A[n]插入到前面已排序的序列
        A[0] = A[i]; //暂存单元，不是哨兵
        low = 1; high = i-1;
        while(low<=high) { //折半查找
            mid = (low+high)/2;
            if(A[min]>A[0]) high = mid-1;
            else low = mid+1;
        }
        for(j=i-1; j>=high+1; --j) //统一后移元素
            A[j+1] = A[j];
        A[high+1] = A[0]; //赋值
    }
}
```

7.2.3 希尔排序

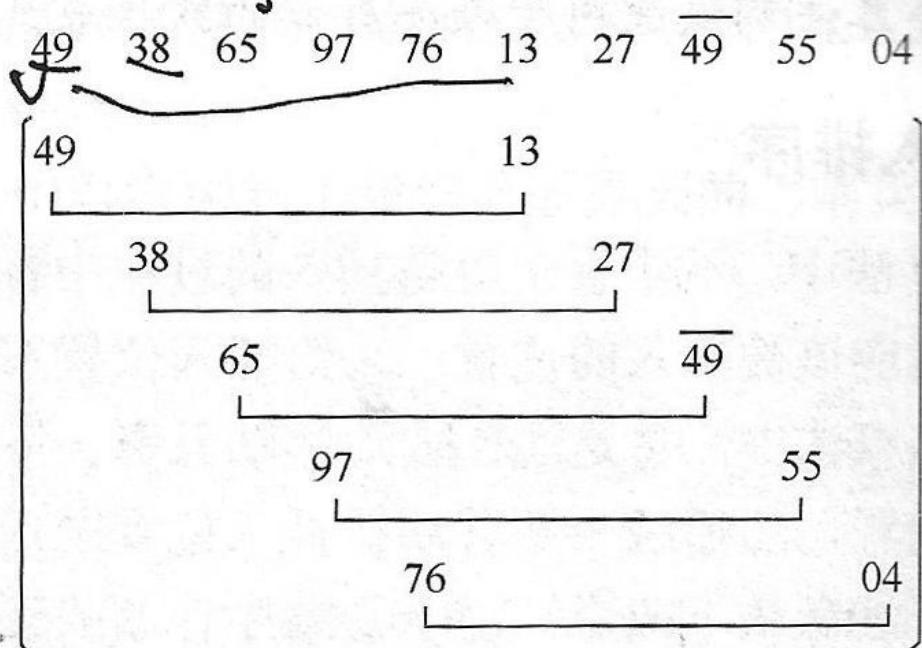
基本思想：先将待排序表分割成若干形如 $\lfloor i, i+d, i+2d, \dots, i+kd \rfloor$ 的特殊子表，即把相隔某个“增量”的记录组成一个子表，对每个子表分别进行直接插入排序，当整个表中的元素已呈“毕本有序”时，再对全体记录进行一次直接插入排序

过程

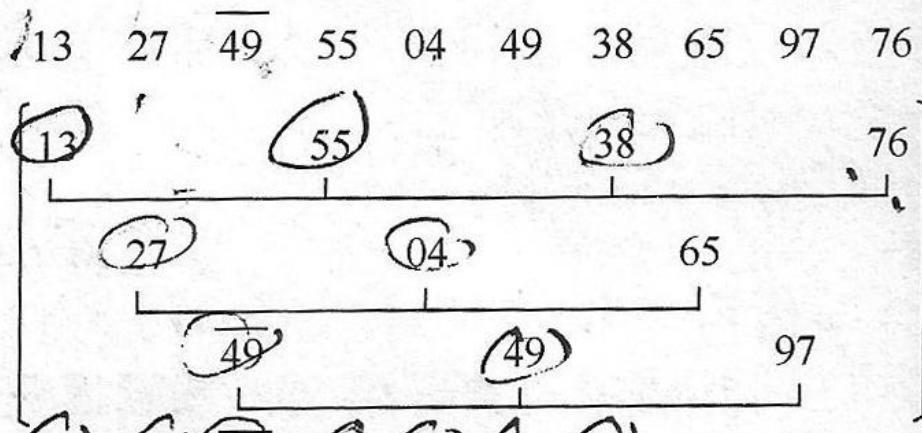
- 先取一个小于 n 的步长 d_1 ，把表中的全部记录分成 d_1 组，所有距离为 d_1 的倍数的记录放在同一组，在各组内进行直接插入排序
- 然后取第二个步长 $d_2 < d_1$ 。
- 重复上述过程，直到所取到的 $d_t = 1$ ，即所有记录已放在同一组，再进行直接插入排序

增量序列： $d_1 = n/2, d_{i+1} = \lfloor d_i/2 \rfloor$ ，最后一个增量等于 1

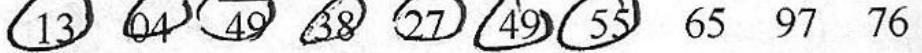
[初始关键字]：



一趟排序结果：



二趟排序结果：



三趟排序结果：

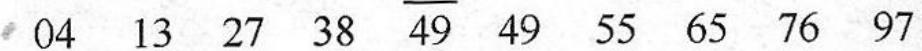


图 8.2 希尔排序示例

```

void ShellSort(ElemType A[], int n){
    //A[0]只是暂存单元，不是哨兵
    for(dk=n/2; dk>=1; dk=dk/2)      //步长变换
        for(i=dk+1; i<=n; i++)
            if(A[i]<A[i-dk]) {          //对d_i个组进行直接插入排序
                A[0] = A[i];             //需要将A[i]插入所在的有序子表中
                for(j=i-dk; j>0&&A[0]<A[j]; j-=dk)//寻找插入位置
                    A[j+dk] = A[j];       //记录后移
                A[j+dk] = A[0];           //插入
            }
}

```

7.3 交换排序

7.3.1 冒泡排序

基本思想：从后往前（或从前往后）两两比较相邻元素的值，若为逆序则交换，指导序列比较完。

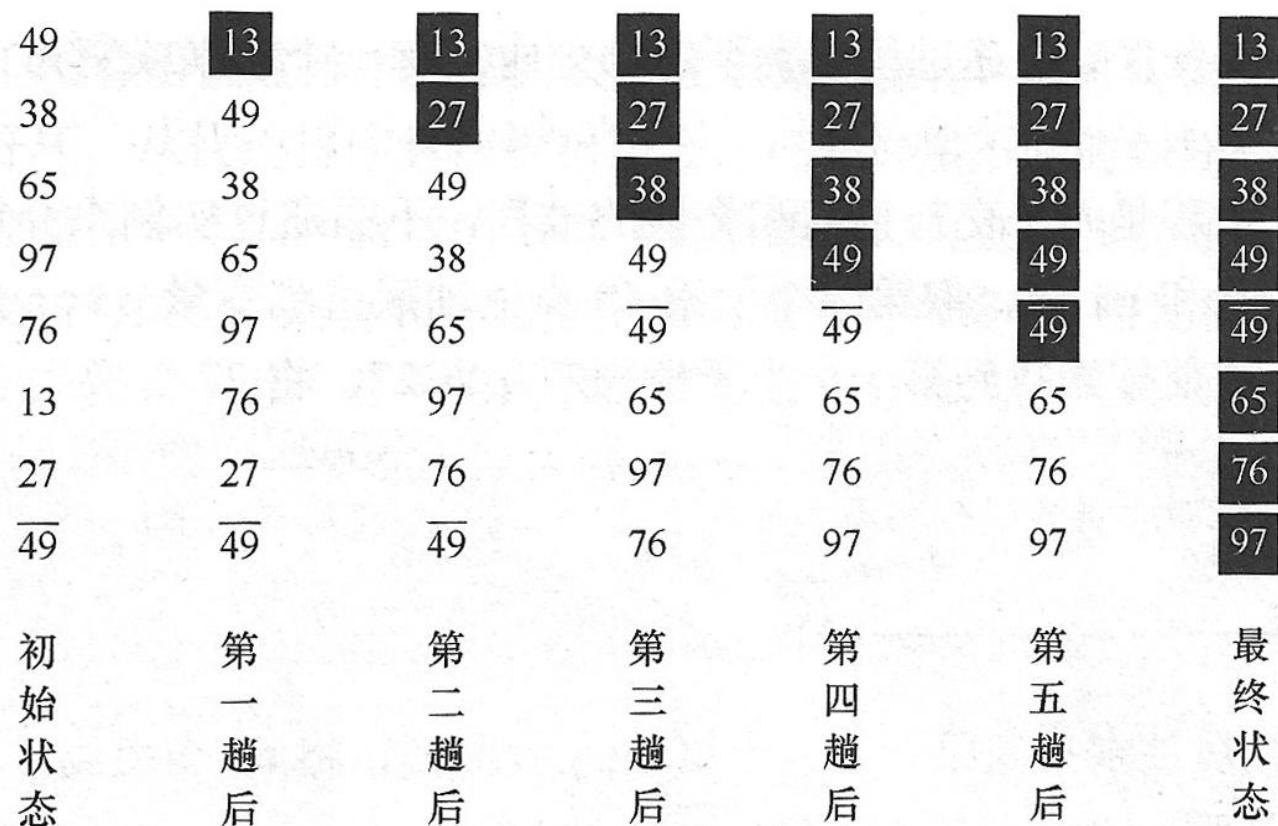


图 8.3 冒泡排序示例

```

void BubbleSort(ElemType A[],int n){
    for(i=0;i<n-1;i++){
        flag = false; //表示本趟冒泡是否发生交换的标志
        for(j=n-1;j>i;j--) //一趟冒泡过程
            if(A[j-1]>A[j]){
                swap(A[j-1],A[j]); //交换
                flag = true;
            }
        if(flag==false)
            return; //本趟遍历后没有发生交换，说明表已经有序
    }
}

```

注意：冒泡排序所产生的有序子序列是全局有序的。每一趟排序都会将一个元素放置到其最终的位置上

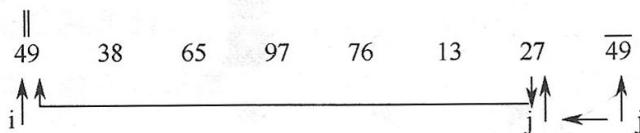
7.3.2 快速排序

基本思想：在待排序表 $L[1 \dots n]$ 中任取一个元素 $pivot$ 作为**枢轴**，通过一趟排序将待排序表划分为独立的两个部分 $L[1 \dots k-1]$ 和 $L[k+1 \dots n]$ ，使得 $L[1 \dots k-1]$ 中的所有元素小于 $pivot$ ， $L[k+1 \dots n]$ 中的所有元素大于等于 $pivot$ ，则 $pivot$ 放在了其最终位置 $L(k)$ 上，这个过程称为一趟快速排序。然后分别对左右两部分重复上述过程，直到每个部分只有一个元素。

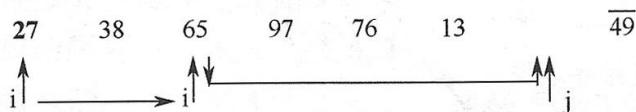
一趟快速排序的过程是一个交替搜索和交换的过程，下面通过实例来介绍，附设两个指针 i 和 j，初值分别为 low 和 high，取第一个元素 49 为枢轴赋值到变量 pivot。

指针 j 从 high 往前搜索找到第一个小于枢轴的元素 27，将 27 交换到 i 所指位置。

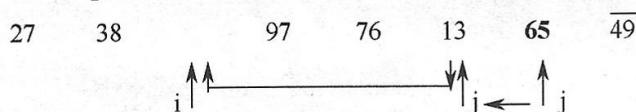
pivot



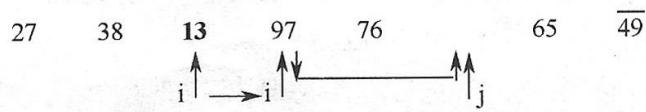
指针 i 从 low 往后搜索找到第一个大于枢轴的元素 65，将 65 交换到 j 所指位置。



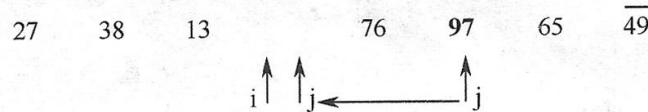
指针 j 继续往前搜索找到小于枢轴的元素 13，将 13 交换到 i 所指位置。



指针 i 继续往后搜索找到大于枢轴的元素 97，将 97 交换到 j 所指位置。



指针 j 继续往前搜索小于枢轴的元素，直至 i==j。



此时，指针 i (==j) 之前的元素均小于 49，指针 i 之后的元素均大于等于 49，将 49 放在 i 所指位置即其最终位置，经过一趟划分，将原序列分割成了前后两个子序列。

{27 38 13} {49} {76 97 65 49}

按照同样的方法对各子序列进行快速排序，若待排序列中只有一个元素，显然已有序。

{13}	27	{38}	{49}	65	76	{97}	
结束		结束		49	{65}	结束	
			结束				
{13}	27	38	49	49	65	76	97

```
void QuickSort(ElemType A[], int low, int high){
    if(low<high){
        //Partition()就是划分操作，将表划分成满足条件的两个子表
        int pivotpos=Partition(A,low,high);      //划分
        QuickSort(A,low,pivotpos-1);
        QuickSort(A,pivotpos+1,high);
    }
}
```

```

int Partition(ElemType A[], int low, int high){    //一趟划分
    ElemenType pivot=A[low];
    while(low<high){
        while(low<high&&A[high]>=pivot) --high;
        A[low] = A[high];           //将比枢轴小的元素移动到左边
        while(low<high&&A[low]<=pivot) ++low;
        A[high] = A[low];          //将比枢轴大的元素移动到右边
    }
    A[low] = pivot;             //枢轴放到最终位置
    return low;
}

```

快速排序是所有内部排序算法中平均性能最优的排序算法

7.4 选择排序

7.4.1 简单选择排序

基本思想：假设排序表为 $L[1 \dots n]$ ，第 i 趟排序即从 $L[i \dots n]$ 中选择关键字最小的元素与 $L(i)$ 交换，每一趟排序可以确定一个元素的最终位置，经过 $n-1$ 趟排序可以使整个排序表有序

```

void SelectSort(ElemType A[], int n){
    for(i=0;i<n-1;i++){           //一共进行n-1趟
        min = i;                  //记录最小元素位置
        for(j=i+1;j<n;j++)        //在A[1\dots n-1]中选择最小的元素
            if(A[j]<A[min]) min = j; //更新最小的元素
        if(min!=i) swap(A[i],A[min]); //封装的swap()函数共移动3次
    }
}

```

7.4.2 堆排序

- 大根堆： $L(i) \geq L(2i) \& L(i) \geq L(2i+1)$ ，最大元素在根结点
- 小根堆： $L(i) \leq L(2i) \& L(i) \leq L(2i+1)$ ，最小元素在根结点
- 堆的插入：把新结点放到堆的末端，后进行向上调整
- 构造初始堆：
 - n 个结点的完全二叉树，最后一个结点是第 $\lfloor n/2 \rfloor$ 个结点的孩子。对第 $\lfloor n/2 \rfloor$ 个结点为根的子树筛选（对于的大根堆，若根结点的关键字小于左右孩子中关键字较大者，则交换），使该子树成为堆。
 - 之后向前依次对各节点 $\lfloor n/2 \rfloor - 1 \sim 1$ 为根的子树进行筛选，看该结点值是否大于其左右子节点的值，不大于的话进行交换
 - 交换后可能会破坏下一级的堆，使用上述办法继续构造下一级的堆，直到以根结点形成堆为止
- 输出堆顶元素，重新构建堆，重复这一过程

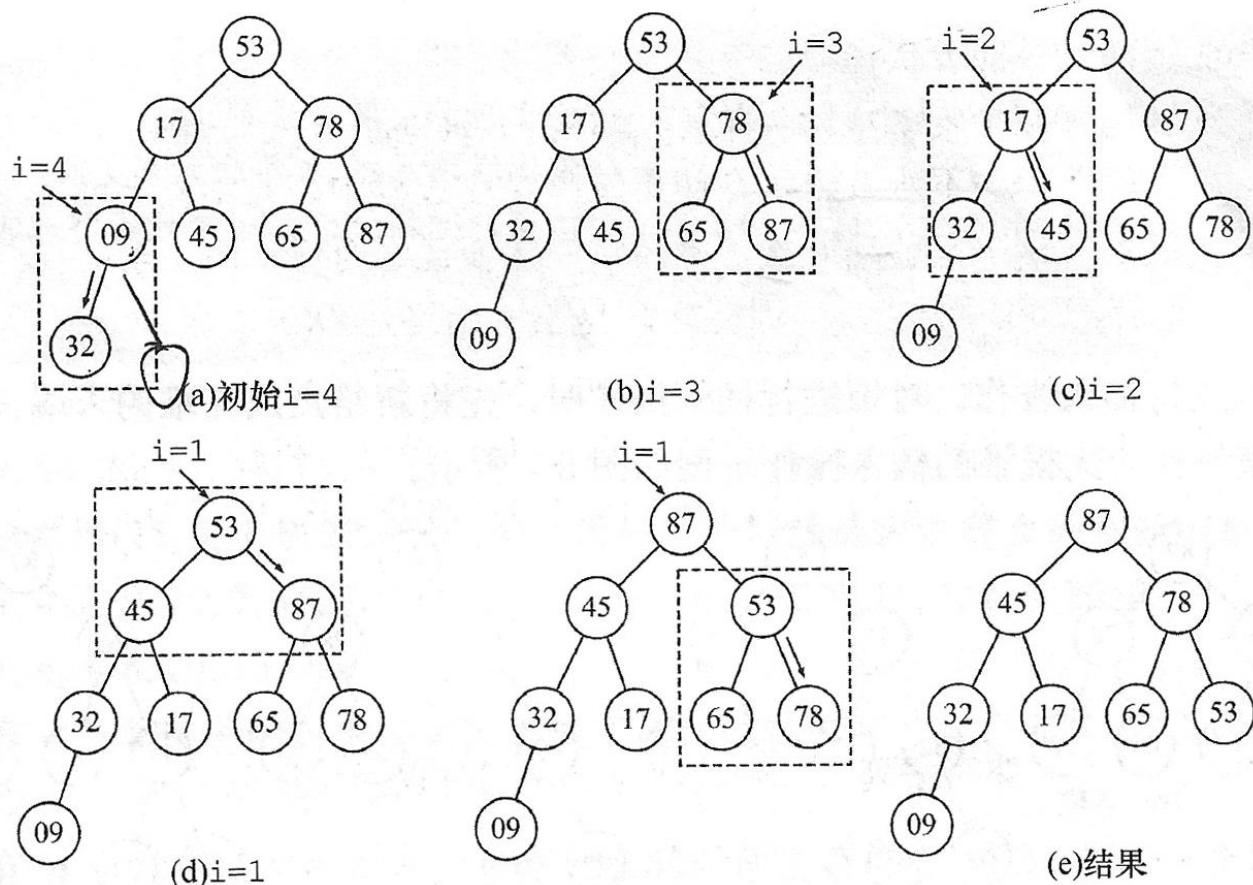


图 8.5 自下往上逐步调整为大根堆

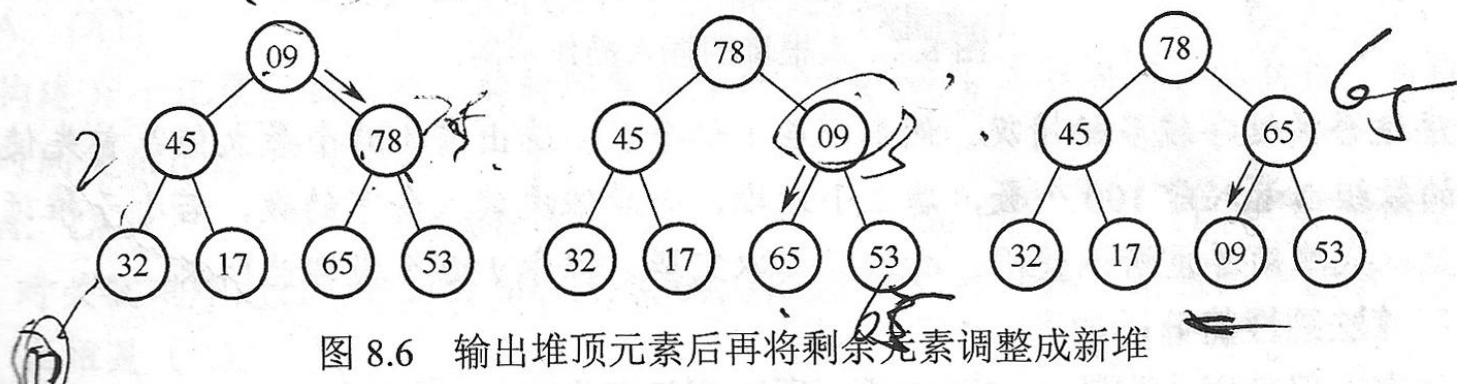


图 8.6 输出堆顶元素后再将剩余元素调整成新堆

```

void BuildMaxHead(ElemType A[], int len){
    for(int i=len/2; i>0; i--) //从i=n/2开始，反复调整堆
        HeadAdjust(A, i, len);
}

void HeadAdjust(ElemType A[], int k, int len){
    //将元素k为根的子树进行调整
    A[0] = A[k]; //A[0]暂存子树的根节点
    for(i=2*k; i<=len; i*=2){ //沿key较大的子节点向下筛选
        if(i<len&&A[i]<A[i+1]) i++; //取i为较大子结点
        if(A[0]>=A[i]) break; //筛选结束
        else{
            A[k]=A[i]; //将A[i]调整到双亲结点上
            k=i; //修改k值，继续向下筛选
        }
    }
    A[k] = A[0]; //被筛选结点的值放入最终位置
}

void HeapSort(ElemType A[], int len){
    BuildMaxHeap(A, len);
    for(i=len; i>1; i--) //n-1趟的交换和建堆过程
        Swap(A[i], A[1]); //输出堆顶元素，和堆底元素交换
        HeadAdjust(A, 1, i-1); //调整，把剩余的i-1个元素整理成堆
}
}

```

7.5 归并排序和基数排序

7.5.1 归并排序

假定待排序序表含有 n 个记录，则可将其视为 n 个有序的子表，每个子表的长度为 1，然后两两合并，得到 $\lceil n/2 \rceil$ 个长度为 2 或 1 的有序表，继续两两合并。这种排序方法称为 2 路归并排序。

一趟归并排序的操作是，调用 $\lceil n/2h \rceil$ 次算法 `merge()`，将 $L[1\dots n]$ 中前后相邻且长度为 h 的有序段进行两两归并，得到前后相邻、长度为 $2h$ 的有序段进行两两归并，得到前后相邻、长度为 $2h$ 的有序段，整个归并排序需要进行 $\lceil \log_2 n \rceil$ 趟。

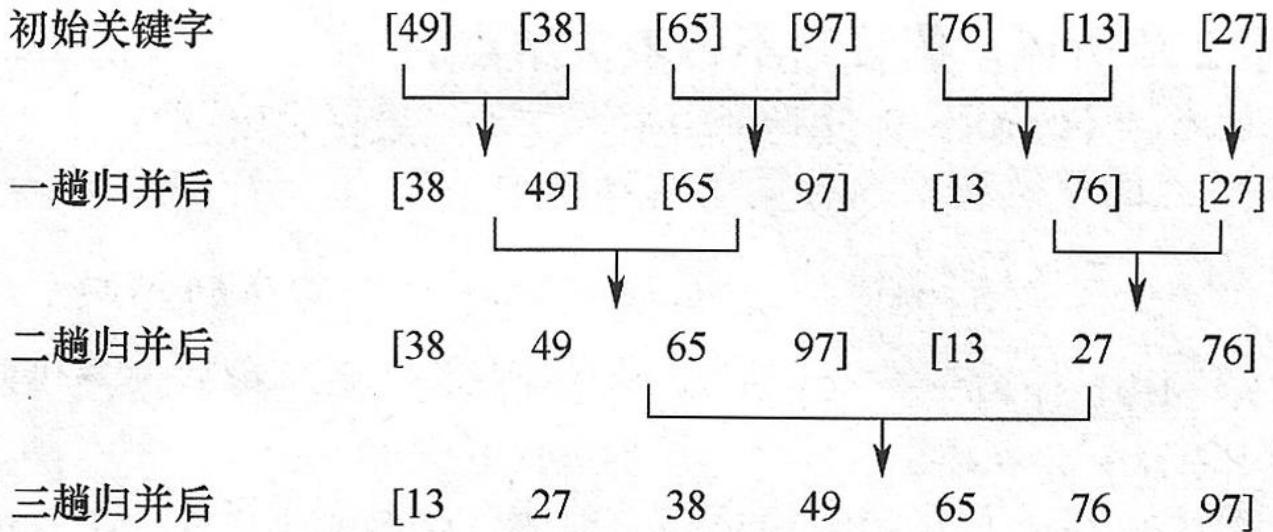


图 8.8 2 路归并排序示例

```

ELemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType)); //辅助数组B
void Merge(ElemType A[],int low,int mid,int high){
    for(int k=low;k<=high;k++)B[k] = A[k]; //将A中元素放到B中
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[i]<=B[j])A[k] = B[i++]; //将较小值赋值到A中
        else A[k] = B[j++];
    }
    while(i<=mid) A[k++] = B[i++]; //若一个表未检测完, 赋值
    while(j<=high) A[k++] = B[j++]; //若第二个表未检测完, 赋值
}
void MergeSort(ElemType A[],int low,int high){
    if(low>high){
        int mid = (low+high)/2;
        MergeSort(A,low,mid);
        MergeSort(A,mid+1,high);
        Merge(A,low,mid,high); //归并
    }
}

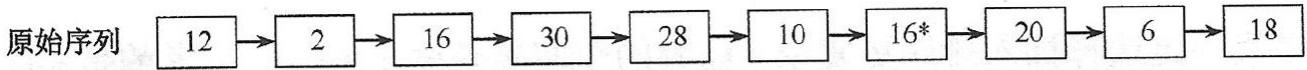
```

7.5.2 基数排序

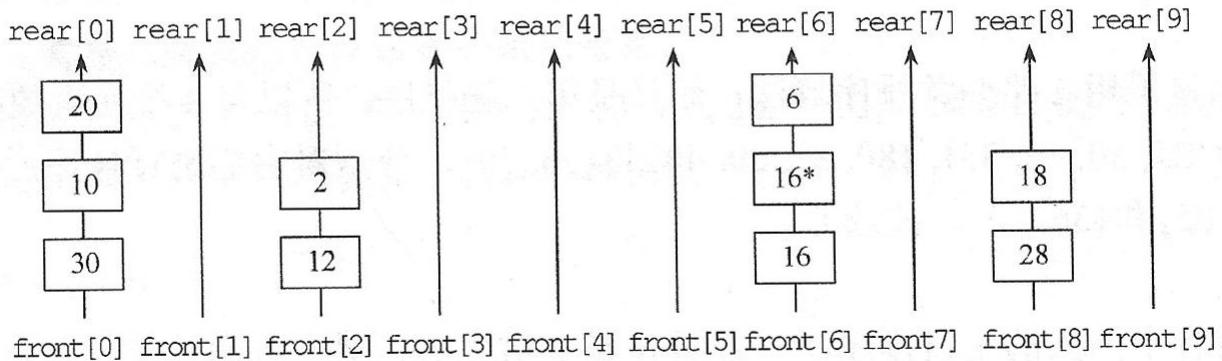
- 最高位优先法 MSD : 将关键字权重递减一次逐层划分成若干更小的子序列，最后将所有子序列依次连接成一个有序序列
- 最低位优先法 LSD : 将关键字权重递增一次进行排序，最后形成一个有序序列

排序过程：

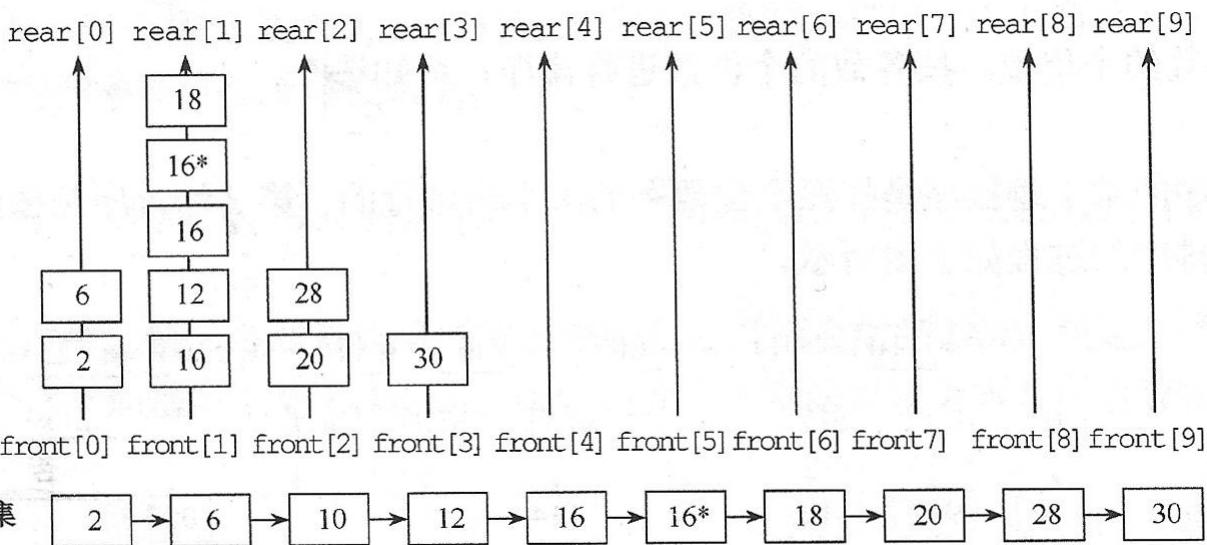
- 在排序中，使用 r 个队列 Q_0, Q_1, \dots, Q_{r-1}
- 对 $i = 0, 1, \dots, d - 1$, 依次做一次**分配和收集**，每个关键字结点 a_j 由 d 元组组成
- 分配：开始时，把 Q_0, Q_1, \dots, Q_{r-1} 各个队列置成空队列，然后依次考察线性表中的每个结点 a_j ，若 a_j 的关键字 $k_j^i = k$ ，就把 a_j 放进 Q_k 队列中
- 收集：把 Q_0, Q_1, \dots, Q_{r-1} 各个队列中的结点依次首尾相连，得到新的结点序列，从而组成新的线性表



按最低位分配



按最高位分配



7.6 各种内部排序算法比较及应用

7.6.1 内部排序算法的比较

算法种类	时间复杂度-最好	时间复杂度-平均	时间复杂度-最坏	空间复杂度	是否稳定
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	否

算法种类	时间复杂度-最好	时间复杂度-平均	时间复杂度-最坏	空间复杂度	是否稳定
2路归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	是
基数排序	$O(d(n + r))$	$O(d(n + r))$	$O(d(n + r))$	$O(r)$	是

选取排序方法需要考虑的因素

- 待排序的元素数目 n : 较小考虑直接插入和简单选择排序, 较大考虑快排】堆排序、归并排序
- 元素本身信息量的大小: 是否选取移动量较少的排序方法
- 关键字的结构及其分布情况: 如已经有序, 则选取直接插入或冒泡排序
- 稳定性的要求
- 语言工具的要求, 存储结构及辅助空间的大小等

7.7 外部排序

- 外部排序指待排序文件较大, 内存一次放不下, 需存放在外存的文件的排序
- 为减少平衡归并中外存读写次数所采取的方法: 增大归并路数和减少归并段个数
- 利用败者树增大归并路数
- 利用置换 - 选择排序增大归并段长度来减少归并段的个数

7.7.1 外部排序的基本概念

在许多应用中, 经常需要对大文件进行排序, 因为文件中的记录很多、信息量庞大, 无法将整个文件复制进内存中进行排序。因此, 需要将待排序的记录存储在外存之上, 排序时再把数据一部分一部分地调进内存进行排序, 在排序过程中需要多次进行内存和外存之间的交换。这种排序方法就称为**外部排序**

7.7.2 外部排序的方法

- 根据内存缓冲区的大小, 将外存上的文件分成若干长度为 l 的子文件, 依次读入内存并利用内部排序方法对它们进行排序, 并将排序后得到的有序子文件重新写回外存, 称这些有序子文件称为归并段或顺串
- 对这些归并段进行逐趟归并, 使归并段逐渐由小到大, 直至得到整个有序文件为止
- 外部排序的总时间=内部排序所需时间+外存信息读写的时间+内部归并所需的时间
- 在进行归并的时候, 需要使用输入缓冲区和输出缓冲区, 在内存和外存中传输数据
- 对 r 个初始段归并, 做 k 路平衡归并, 归并树可用严格 k 叉树来表示, 树的高度 = $\lceil \log_k r \rceil$ = 归并趟数 S

7.7.3 多路平衡归并与败者树

- 做内部归并时, 在 k 个元素中选择关键字最小的记录需要比较 $k-1$ 次, S 趟归并总需的比较次数是 $S(n-1)(k-1) = \lceil \log_k r \rceil(n-1)(k-1) = \lceil \log_2 r \rceil(n-1)(k-1)\lceil \log_2 k \rceil$
- 引入败者树后, 在 k 个元素中选择关键字最小的记录需要比较 $\lceil \log_2 k \rceil$ 次, 内部归并的比较次数与 k 无关。因此只要内存允许, 增大归并路数 k 将有效减少归并树的高度, 提高外部排序的速度
- 败者树
 - k 个叶结点分别存放 k 个归并段在归并过程中当前参加比较的记录
 - 内存结点用来记忆左右子树中的失败者, 而让胜者往上继续比较, 一直到根结点
 - 根结点记录胜者
 - 叶结点进行编号 $b_0 \sim b_k$, 内存结点编号 $ls[0] \sim ls[k]$, $ls[0]$ 为根结点

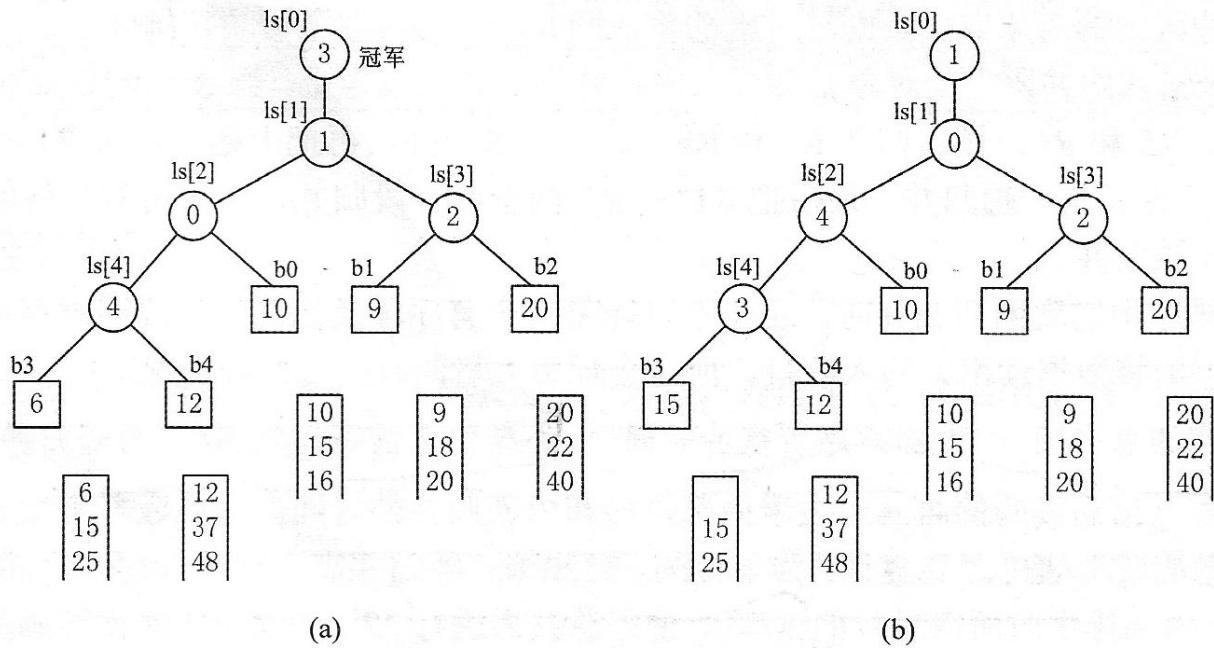


图 8.9 实现 5 路归并的败者树

7.7.4 置换 - 选择排序 (生成初始归并段)

初始待排文件 FI，初始归并输出文件为 FO，内存工作区为 WA，FO 与 WA 的初始状态为空，WA 可容纳 w w w 个记录

1. 从 FI 输入 w 个记录到工作区 WA
2. 从 WA 中选出其中关键字取最小值的记录，记为 MINIMAX
3. 将 MINIMAX 就输出到 FO 中去
4. 若 FI 不为空，则从 FI 输入下一个记录到 WA 中
5. 从 WA 中所有关键字比 MINIMAX 记录的关键字大的记录中选出最小关键字记录，作为新的 MINIMAX
6. 重复 3-5，直至在 WA 中选不出新的 MINIMAX 记录为止，由此得到一个初始归并段，输出一个归并段的结束标志至 FO 中去
7. 重复 2-6，直至 WA 为空，由此得到全部初始归并段

表 8.2 置换-选择排序过程示例

输出文件 FO	工作区 WA	输入文件 FI
—	—	17, 21, 05, 44, 10, 12, 56, 32, 29
—	17 21 05	44, 10, 12, 56, 32, 29
05	17 21 44	10, 12, 56, 32, 29
05 17	10 21 44	12, 56, 32, 29
05 17 21	10 12 44	56, 32, 29
05 17 21 44	10 12 56	32, 29

(续表)

输出文件 FO	工作区 WA	输入文件 FI
05 17 21 44 56	10 12 32	29
05 17 21 44 56 #	10 12 32	29
10	29 12 32	—
10 12	29 32	—
10 12 29	32	—
10 12 29 32	—	—
10 12 29 32 #	—	—

7.7.5 最佳归并树

把归并段的长度作为权值，进行严格 k 叉树的哈夫曼树思想，构造最佳归并树

- $(n_0 - 1)\%(k - 1) = 0$, 不需要添加
- $(n_0 - 1)\%(k - 1) = u \neq 0$, 需要添加 $k - 1 - u$ 个长度为 0 的虚段

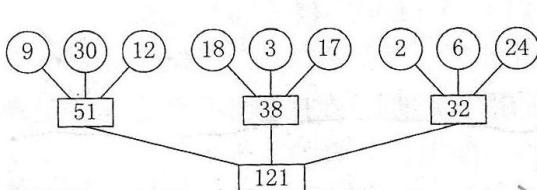


图 8.10 3 路平衡归并的归并树

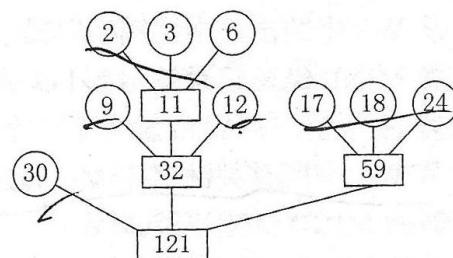


图 8.11 3 路平衡归并的最佳归并树

图 8.11 中的哈夫曼树是一棵严格 3 叉树，即树中只有度为 3 或 0 的结点。若只有 8 个初始归并段，如上例中少了一个长度为 30 的归并段。若在设计归并方案时，缺额的归并段留在最后，即除最后一次做 2 路归并外，其他各次归并仍是 3 路归并，此归并方案的外存读/写次数为 386。显然，这不是最佳方案。

正确的做法是：若初始归并段不足以构成一棵严格 k 叉树时，需添加长度为 0 的“虚段”，按照哈夫曼树的原则，权为 0 的叶子应离树根最远。因此，最佳归并树应如图 8.12 所示。

如何判定添加虚段的数目？

设度为 0 的结点有 $n_0 (=n)$ 个，度为 k 的结点有 n_k 个，则

对严格 k 叉树有 $n_0 = (k-1)n_k + 1$ ，由此可得 $n_k = (n_0 - 1)/(k-1)$ 。

- 若 $(n_0 - 1)%(k-1) = 0$ （% 为取余运算），则说明这 n_0 个叶结点（初始归并段）正好可以构造 k 叉归并树。此时，内结点有 n_k 个。
- 若 $(n_0 - 1)%(k-1) = u \neq 0$ ，则说明对于这 n_0 个叶结点，其中有 u 个多余，不能包含在 k 叉归并树中。为构造包含所有 n_0 个初始归并段的 k 叉归并树，应在原有 n_k 个内结点的基础上再增加 1 个内结点。它在归并树中代替了一个叶结点的位置，被代替的叶结点加上刚才多出的 u 个叶结点，即再加上 $k-u-1$ 个空归并段，就可以建立归并树。

以图 8.12 为例，用 8 个归并段构成 3 叉树， $(n_0 - 1)%(k-1) = (8-1)%(3-1) = 1$ ，说明 7 个归并段刚好可以构成一棵严格 3 叉树（假设把以 5 为根的树视为一个叶子）。为此，将叶子 5 变成一个内结点，再添加 $3-1-1=1$ 个空归并段，就可以构成一棵严格 k 叉树。

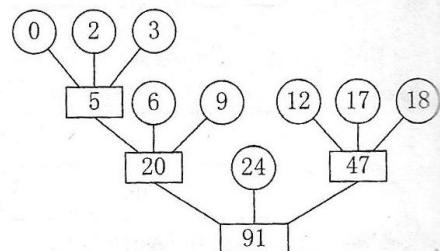


图 8.12 8 个归并段的最佳归并树