

# 拓扑排序&查找

**Zibin Zheng（郑子彬）**

**School of Data and Computer Science，SYSU**

**http://www.inpluslab.com**

课程主页：http://inpluslab.sysu.edu.cn/dsa2016/

# 有向无环图及其应用

- 有向无环图(Directed Acycling Graph)：是图中没有回路(环)的有向图。是一类具有代表性的图，主要用于研究工程项目的工序问题、工程时间进度问题等。
- 一个工程(project)都可分为若干个称为活动(active)的子工程(或工序)，各个子工程受到一定的条件约束：某个子工程必须开始于另一个子工程完成之后；整个工程有一个开始点(起点)和一个终点。人们关心：
  - 工程能否顺利完成?影响工程的关键活动是什么?
  - 估算整个工程完成所必须的最短时间是多少?
- 对工程的活动加以抽象：图中顶点表示活动，有向边表示活动之间的优先关系，这样的有向图称为顶点表示活动的网(Activity On Vertex Network ，AOV网) 。

# 拓扑排序

在AOV网中，若有有向边<i, j>，则i是j的直接前驱，j是i的直接后继；推而广之，若从顶点i到顶点j有有向路径，则i是j的前驱，j是i的后继。

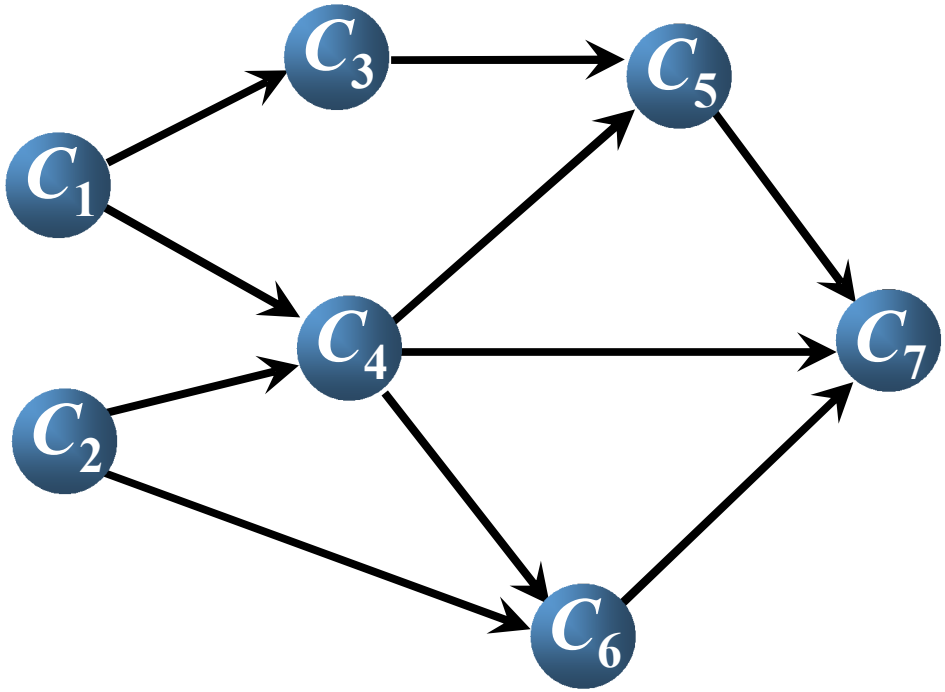
在AOV网中，不能有环，否则，某项活动能否进行是以自身的完成作为前提条件。

检查方法：对有向图的顶点进行拓扑排序，若所有顶点都在其拓扑有序序列中，则无环。

有向图的拓扑排序：构造AOV网中顶点的一个拓扑线性序列($v'_1, v'_2, \cdots, v'_n$)，使得该线性序列不仅保持原来有向图中顶点之间的优先关系，而且对原图中没有优先关系的顶点之间也建立一种(人为的)优先关系。

# AOV网

| 编号 | 课程名称 | 先修课 |
|---|---|---|
| $C_1$ | 高等数学 | 无 |
| $C_2$ | 计算机导论 | 无 |
| $C_3$ | 离散数学 | $C_1$ |
| $C_4$ | 程序设计 | $C_1$，$C_2$ |
| $C_5$ | 数据结构 | $C_3$，$C_4$ |
| $C_6$ | 计算机原理 | $C_2$，$C_4$ |
| $C_7$ | 数据库原理 | $C_4$，$C_5$，$C_6$ |



# 拓扑排序

**拓扑序列：** 设$G=(V, E)$是一个具有$n$个顶点的有向图，$V$中的顶点序列$v_1, v_2, \ldots, v_n$称为一个拓扑序列，当且仅当满足下列条件：若从顶点$v_i$到$v_j$有一条路径，则在顶点序列中顶点$v_i$必在顶点$v_j$之前。

**拓扑排序：** 对一个有向图构造拓扑序列的过程称为拓扑排序 。

拓扑序列使得AOV网中所有应存在的前驱和后继关系都能得到满足。

# 拓扑排序

**基本思想：**

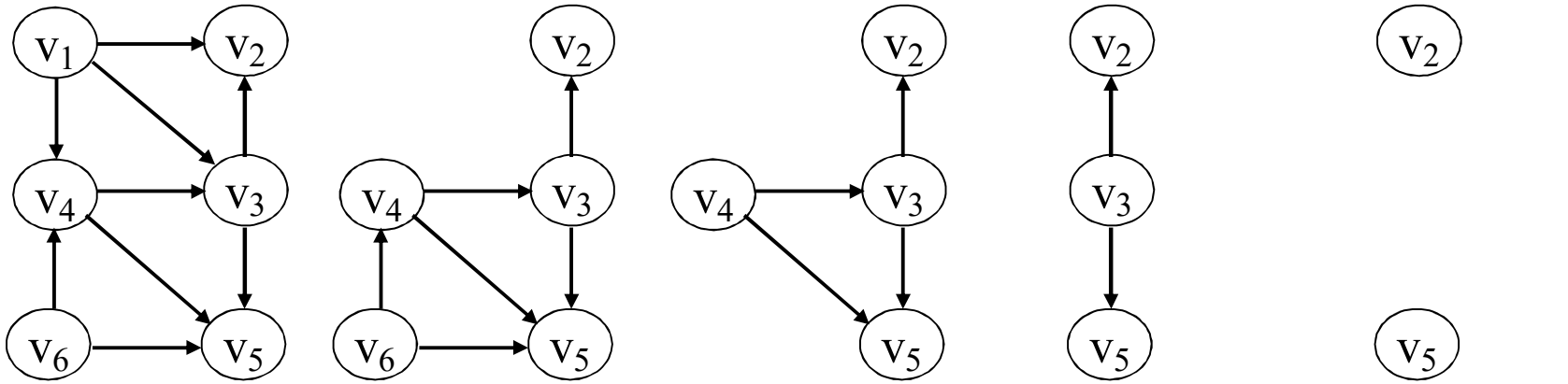
(1) 从AOV网中选择一个没有前驱的顶点并且输出；

(2) 从AOV网中删去该顶点，并且删去所有以该顶点为尾的弧；

(3) 重复上述两步，直到全部顶点都被输出，或AOV网中不存在没有前驱的顶点。

# 拓扑排序算法

## 2 算法思想




(a) 有向图　(b) 输出$v_1$后　(c) 输出$v_6$后　(d) 输出$v_4$后　(e) 输出$v_3$后

有向图的拓扑排序过程，其拓扑序列是：(v1,v6,v4,v3,v2,v5)

# 拓扑排序算法实现说明

- 算法实现
  - 采用正邻接链作为AOV网的存储结构；
  - 设立堆栈，用来暂存入度为0的顶点；
  - 删除顶点以它为尾的弧：弧头顶点的入度减1。

# 设计数据结构

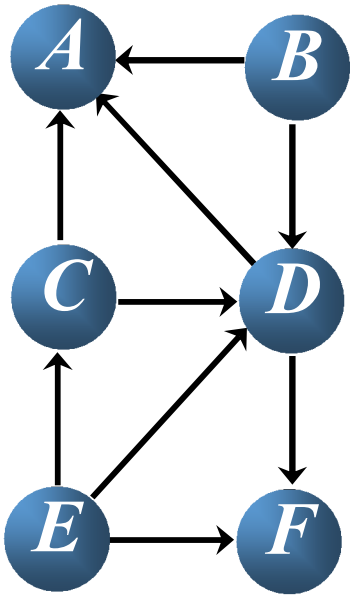
1. 图的存储结构：采用邻接表存储，在顶点表中增加一个入度域。

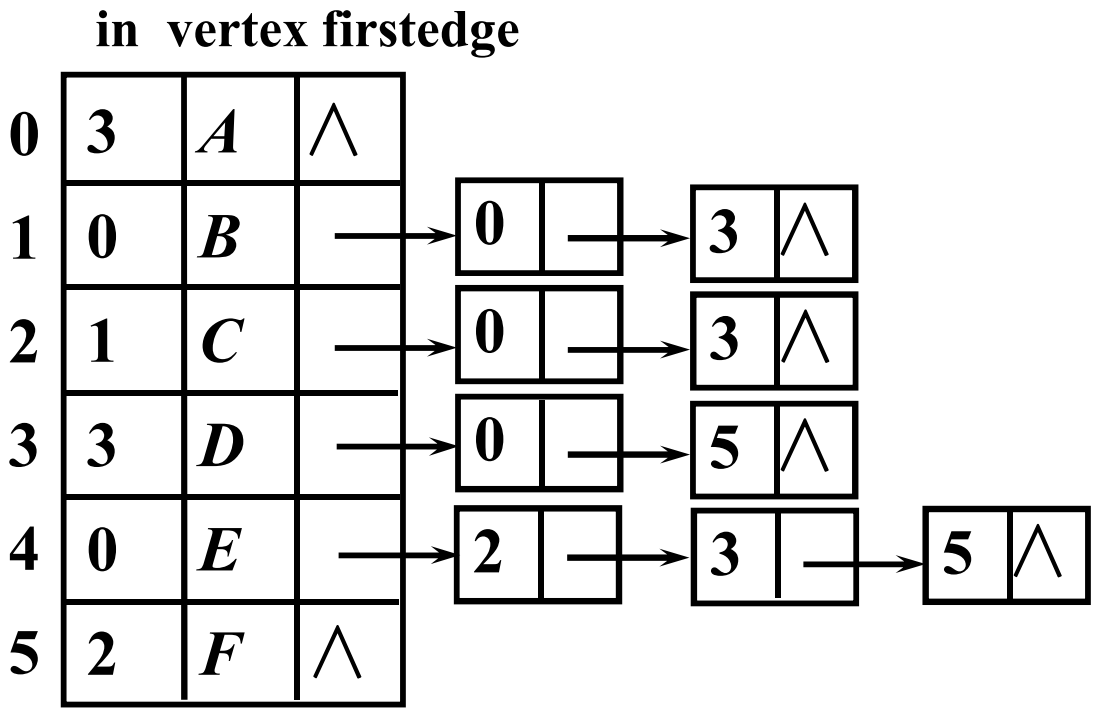
| in | vertex | firstedge |
|---|---|---|

顶点表结点

2. 栈S：存储所有无前驱的顶点。也可以用队列。

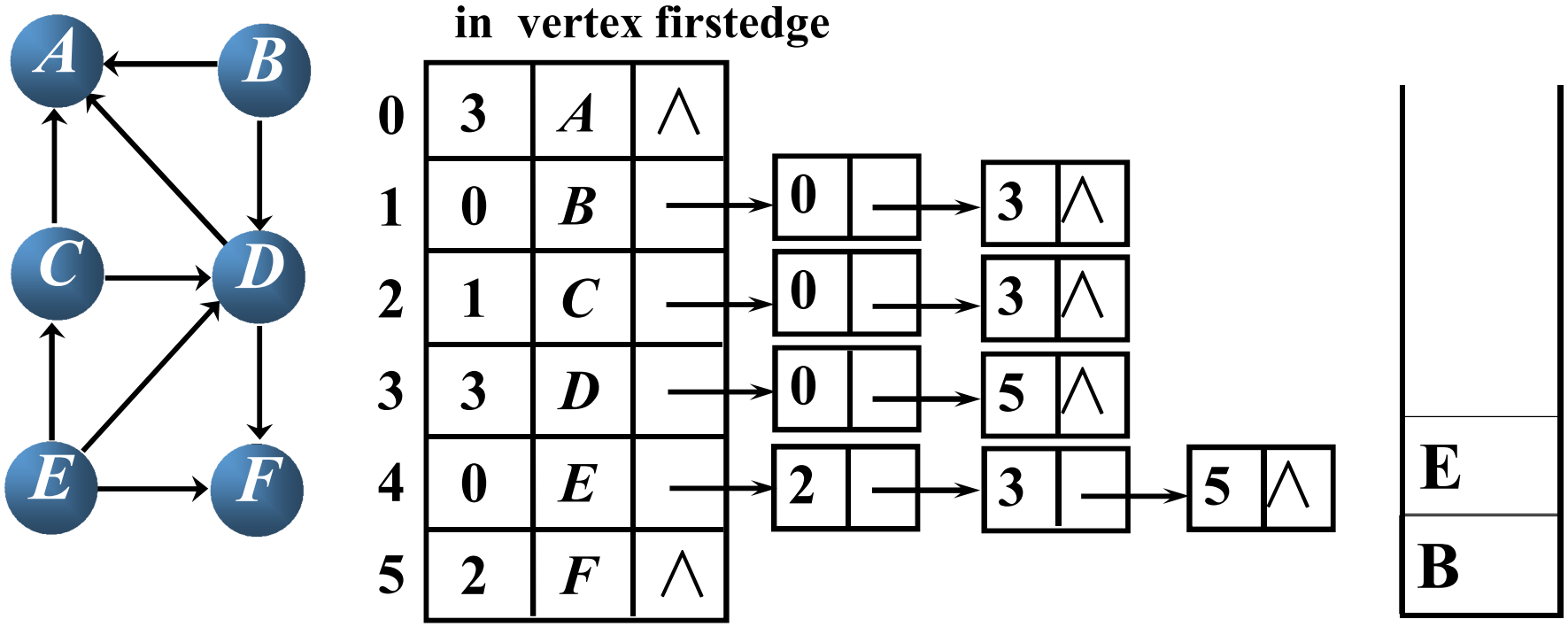
# 拓扑排序



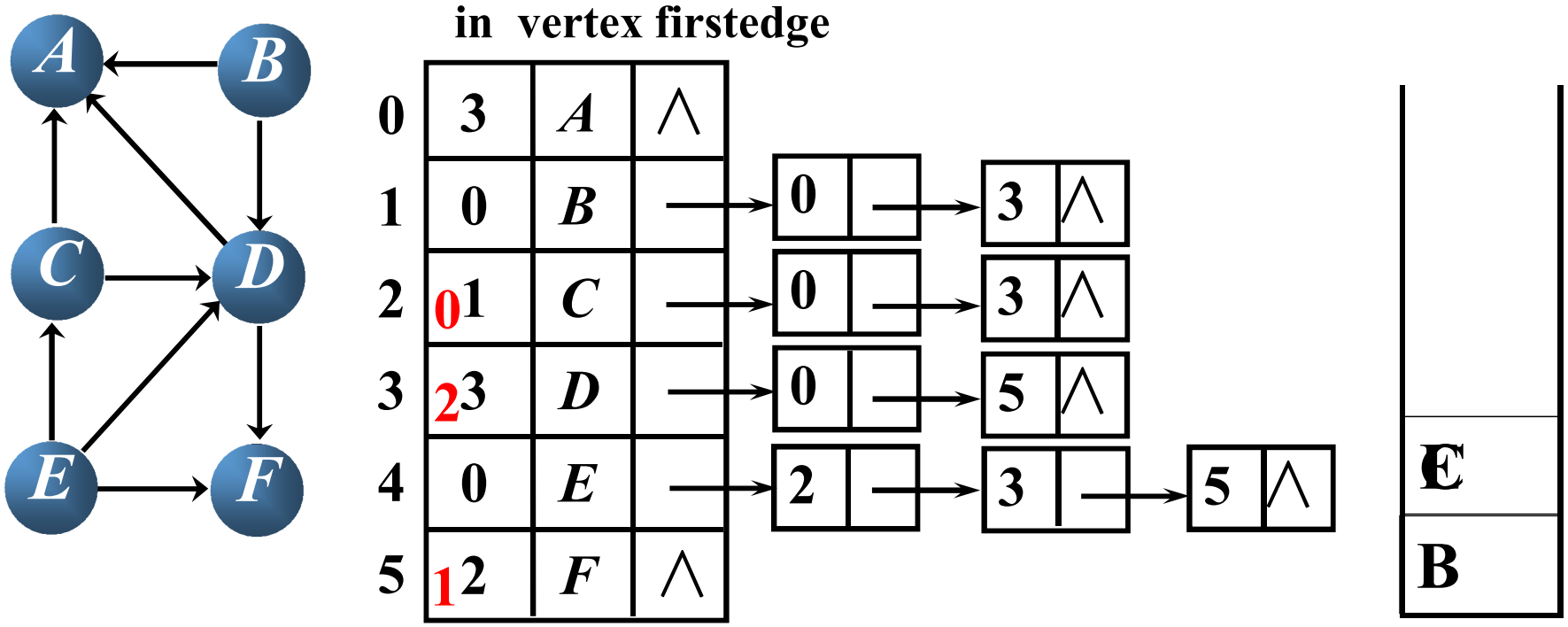




(a) 一个AOV网　　(b) AOV网的邻接表存储

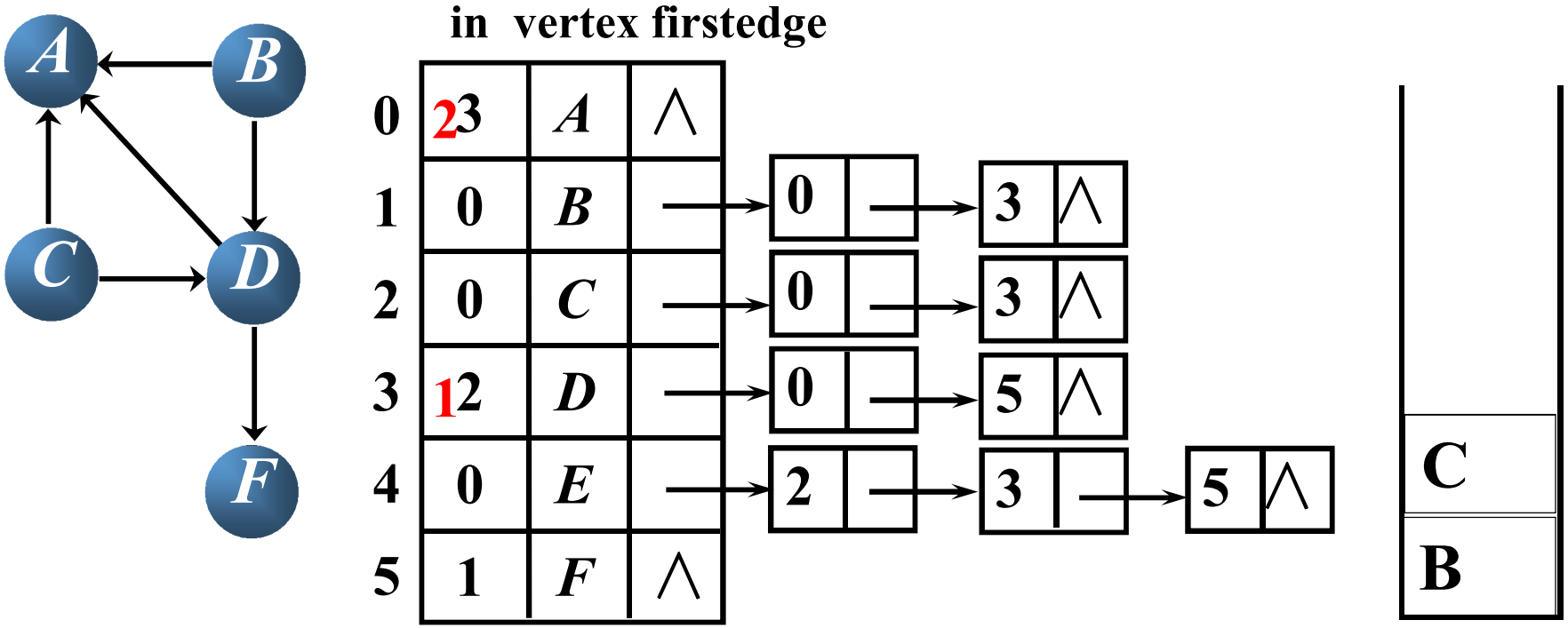
# 拓扑排序


A
B
C
D
E
F
in vertex firstedge
0 3 A ∧
1 0 B
2 1 C
3 3 D
4 0 E
5 2 F
0 3 ∧
0 3 ∧
0 5 ∧
2 3 5 ∧
E
B

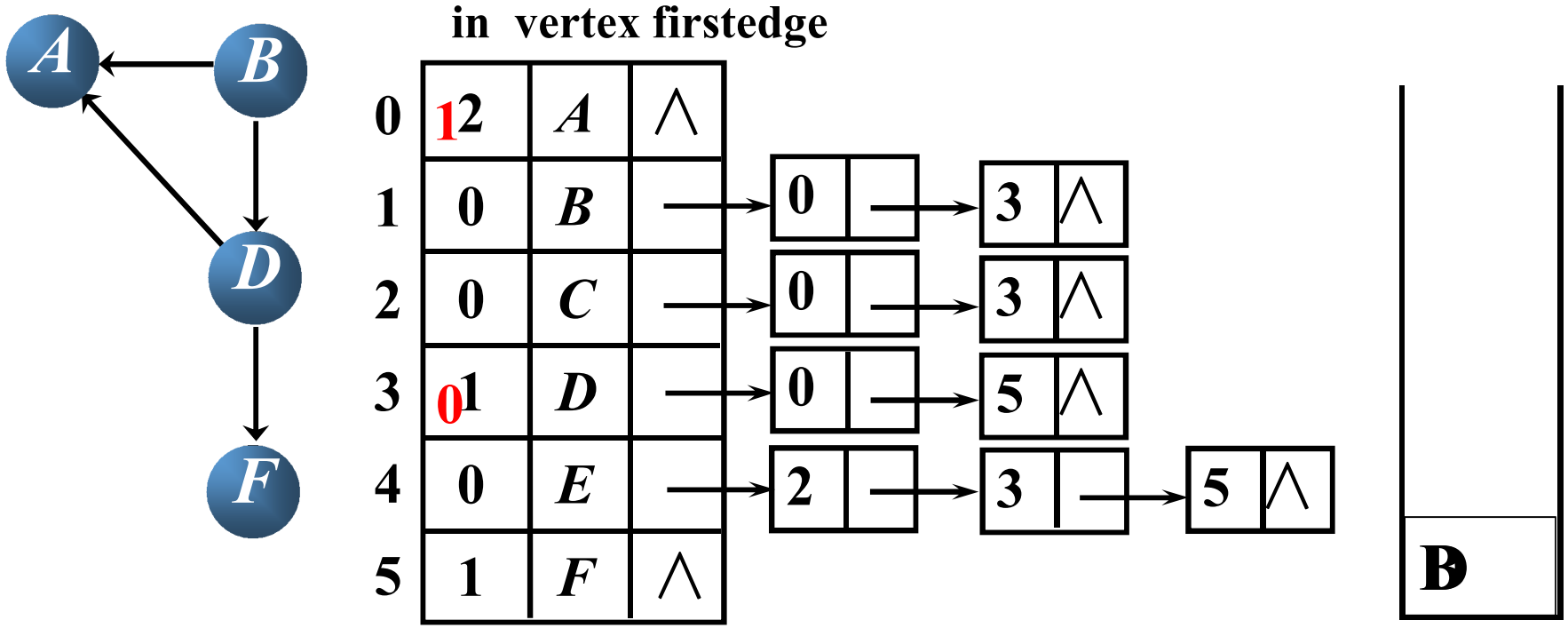
# 拓扑排序


A
B
C
D
E
F
in vertex firstedge
0 3 A ∧
1 0 B
2 01 C
3 23 D
4 0 E
5 12 F
0 3 ∧
0 3 ∧
0 5 ∧
2 3 5 ∧
E
B

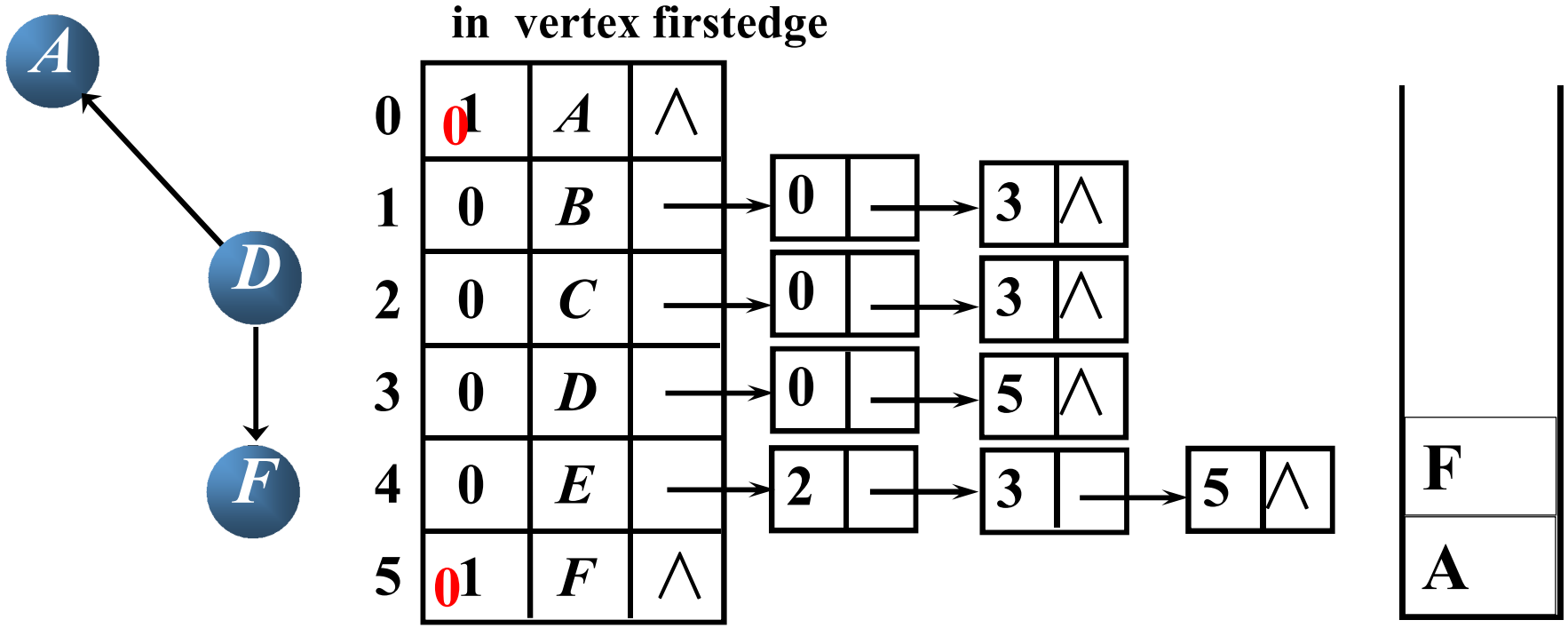
# 拓扑排序


A
B
C
D
F
in vertex firstedge
0 23 A ∧
1 0 B
2 0 C
3 12 D
4 0 E
5 1 F
0
3 ∧
0
3 ∧
0
5 ∧
2
3
5 ∧
C
B

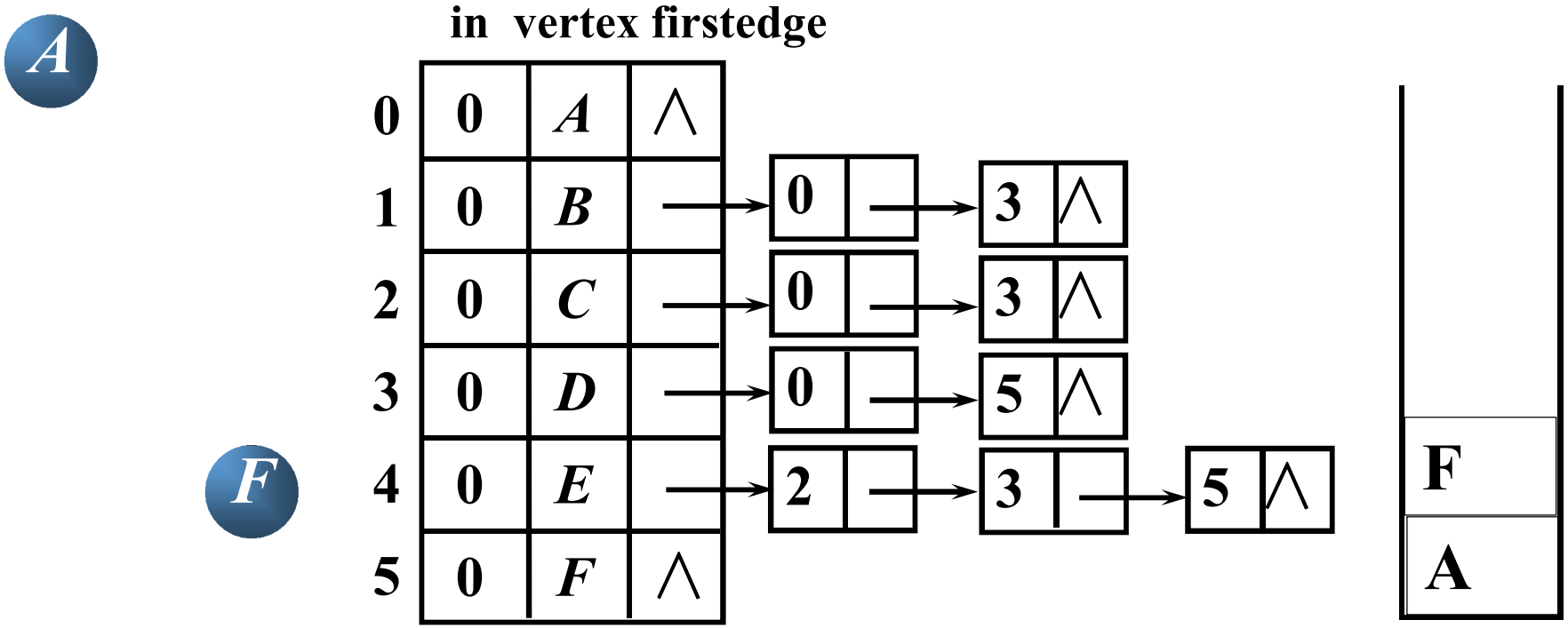
# 拓扑排序


A
B
D
F
in vertex firstedge
0 12 A ∧
1 0 B
2 0 C
3 01 D
4 0 E
5 1 F
0 3 ∧
0 3 ∧
0 5 ∧
2 3 5 ∧
B

# 拓扑排序



# 拓扑排序



# 拓扑排序算法——伪代码

1. 栈S初始化；累加器count初始化；
2. 扫描顶点表，将没有前驱的顶点压栈；
3. 当栈S非空时循环
    3.1 $v_j$=退出栈顶元素；输出$v_j$；累加器加1；
    3.2 将顶点$v_j$的各个邻接点的入度减1；
    3.3 将新的入度为0的顶点入栈；
4. if（count<vertexNum）输出有回路信息；

# 拓扑排序算法实现

**(1) 统计各顶点入度的函数**

```
void count_indegree(ALGraph *G)
{
    int k ; LinkNode *p ;
    for (k=0; k<G->vexnum; k++)
    G->adjlist[k].indegree=0 ;    /* 顶点入度初始化 */
    for (k=0; k<G->vexnum; k++)
    {
        p=G->adjlist[k].firstarc ;
        while (p!=NULL)    /* 顶点入度统计 */
        {
            G->adjlist[p->adjvex].indegree++ ;
            p=p->nextarc ;
        }
    }
}
```

**(2) 拓扑排序算法**

```
int  Topologic_Sort(ALGraph *G, int topol[])
{  int  k, no, vex_no, top=0, count=0, boolean=1 ;
   int  stack[MAX_VEX] ;     /* 用作堆栈 */
   LinkNode *p ;
   count_indegree(G) ;  /* 统计各顶点的入度 */
   for (k=0; k<G->vexnum; k++)
   if  (G->adjlist[k].indegree==0)
       stack[++top]=G->adjlist[k].data ;
    do
    {  if (top==0)  boolean=0 ;
       else
       {  no=stack[top--] ;   /* 栈顶元素出栈 */
          topl[++count]=no ;   /* 记录顶点序列 */
          p=G->adjlist[no].firstarc ;
          while (p!=NULL)    /*删除以顶点为尾的弧*/
          {  vex_no=p->adjvex ;
             G->adjlist[vex_no].indegree-- ;
             if  (G->adjlist[vex_no].indegree==0)
                 stack[++top]=vex_no  ;
             p=p->nextarc ;
          }
       }
    } while(boolean==0) ;
   if (count<G->vexnum)  return(-1) ;
   else  return(1) ;
}
```

# 有向无环图及其应用

AOE网

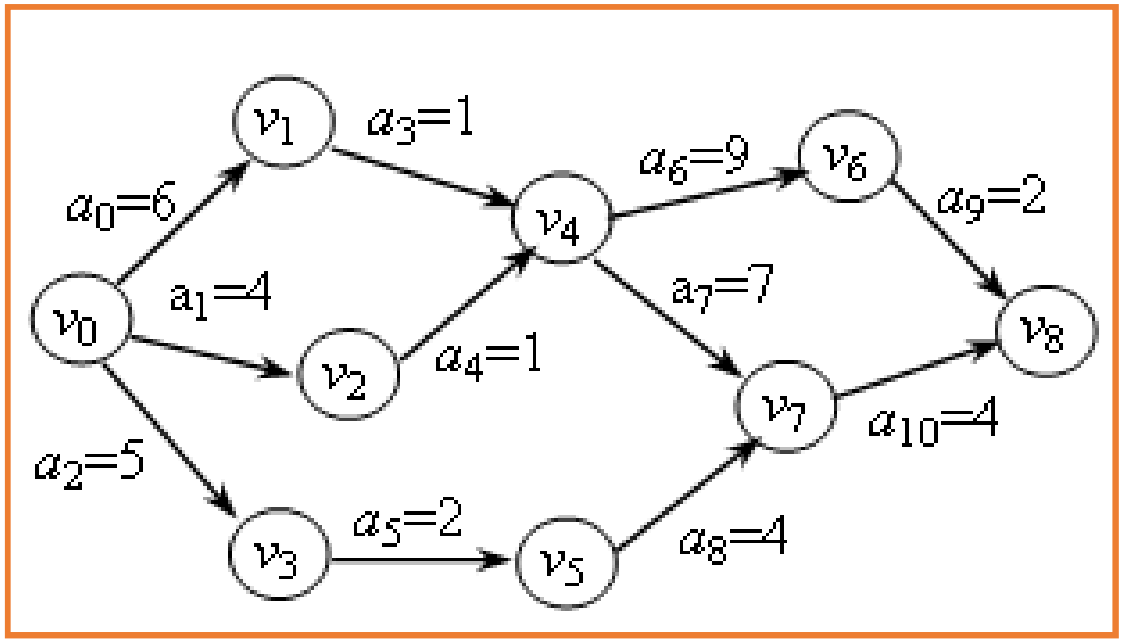
**AOE网：**在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，边上的权值表示活动的持续时间，称这样的有向图叫做边表示活动的网，简称AOE网。AOE网中没有入边的顶点称为始点（或源点），没有出边的顶点称为终点（或汇点）。

**AOE网的性质：**

(1) 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始；

(2) 只有在进入某顶点的各活动都结束，该顶点所代表的事件才能发生。

# AOE网




例如，事件$v_4$表示活动$a_3$和$a_4$已经结束，活动$a_6$和$a_7$可以开始。

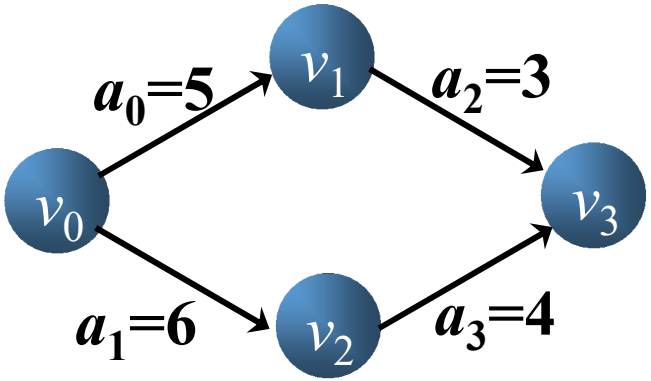
# AOE网

**AOE网可以回答下列问题：**

**1. 完成整个工程至少需要多少时间?**

**2. 为缩短完成工程所需的时间, 应当加快哪些活动?**

**以下AOE网的最短工期是多少？**




最短工期应该是最长路径

# 关键路径

**关键路径：**在AOE网中，从始点到终点具有最大路径长度（该路径上的各个活动所持续的时间之和）的路径称为关键路径。

**关键活动：**关键路径上的活动称为关键活动。

关键路径可能不只一条，重要的是找到关键活动

## 首先计算以下与关键活动有关的量：

关键路径

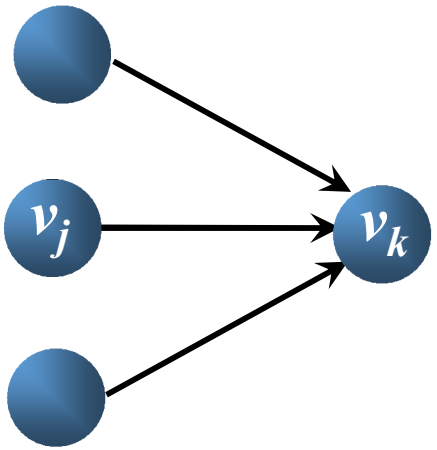
要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。

(1) 事件的最早发生时间**ve[k]**
(2) 事件的最迟发生时间**vl[k]**
(3) 活动的最早开始时间**e[i]**
(4) 活动的最晚开始时间**l[i]**

最后计算各个活动的时间余量 **l[k] - e[k]**，时间余量为**0**者即为关键活动。

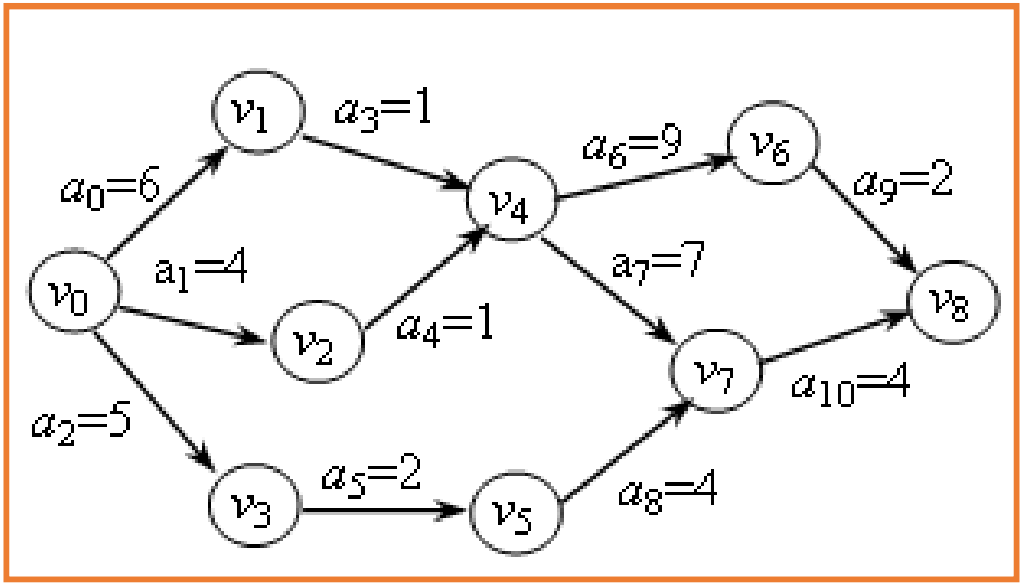
## (1) 事件的最早发生时间ve[k]

ve[k]是指从始点开始到顶点$v_k$的最大路径长度。这个长度决定了所有从顶点$v_k$发出的活动能够开工的最早时间。




$$\begin{cases} ve[1]=0 \\ ve[k]=\max\{ve[j]+len<v_j, v_k>\} \ (<v_j, v_k> \in p[k]) \end{cases}$$

p[k]表示所有到达$v_k$的有向边的集合

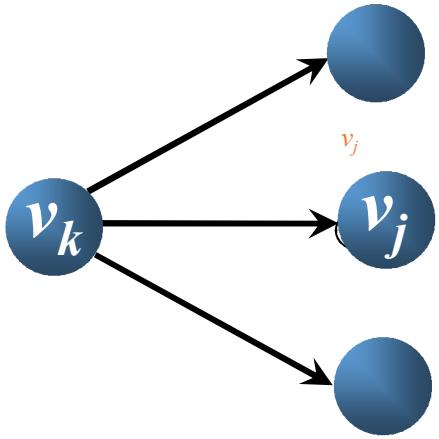



$$ve[k]=\max\{ve[j]+len<v_j, v_k>\}$$

| | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ |
|---|---|---|---|---|---|---|---|---|---|
| ve[k] | 0 | 6 | 4 | 5 | 7 | 7 | 16 | 14 | 18 |

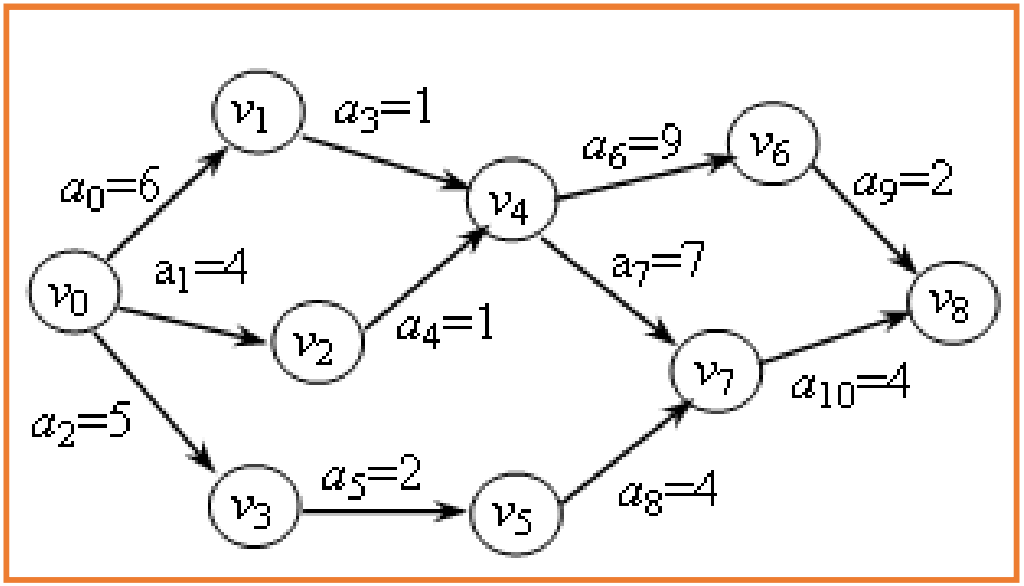
## (2) 事件的最迟发生时间vl[k]

vl[k]是指在不推迟整个工期的前提下,事件$v_k$允许的最晚发生时间。




$$\begin{cases} vl[n]=ve[n] \\ vl[k]=\min\{vl[j]-len<v_k, v_j>\}\ (<v_k, v_j>\in s[k]) \end{cases}$$

s[k]为所有从$v_k$发出的有向边的集合




$$vl[k]=\min\{vl[j]-len<v_k, v_j>\}$$

| | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ |
|---|---|---|---|---|---|---|---|---|---|
| ve[k] | 0 | 6 | 4 | 5 | 7 | 7 | 16 | 14 | 18 |
| vl[k] | 0 | 6 | 6 | 8 | 7 | 10 | 16 | 14 | 18 |

### (3) 活动的最早开始时间e[i]

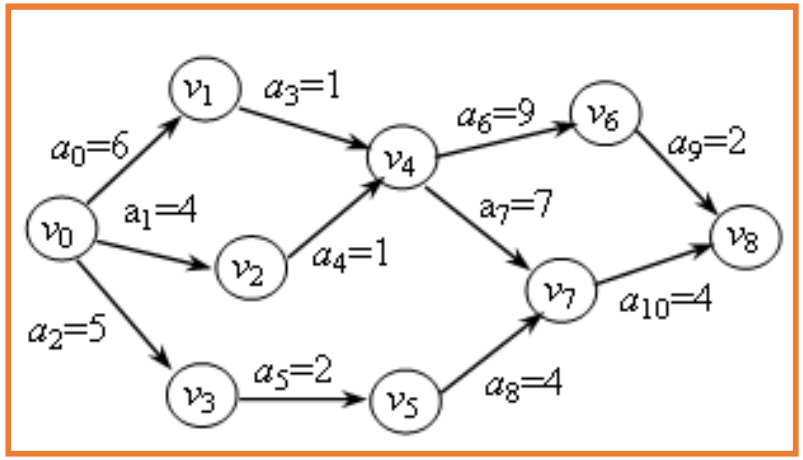
若活动$a_i$是由弧$<v_k, v_j>$表示，则活动$a_i$的最早开始时间应等于事件$v_k$的最早发生时间。因此，有：

**e[i]=ve[k]**

### (4) 活动的最晚开始时间l[i]

若$a_i$由弧$<v_k, v_j>$表示，则$a_i$的最晚开始时间要保证事件$v_j$的最迟发生时间不拖后。因此，有：

**l[i]=vl[j]-len$<v_k, v_j>$**

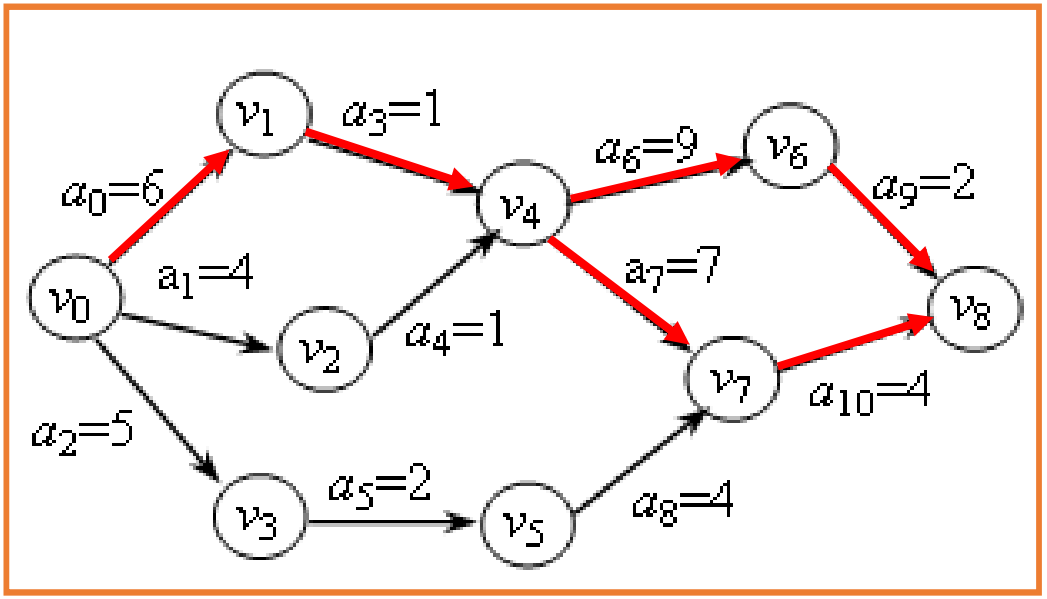



e[i]=ve[k]

l[i]=vl[j]-len<$v_k$, $v_j$>

| | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ |
|---|---|---|---|---|---|---|---|---|---|
| ve[k] | 0 | 6 | 4 | 5 | 7 | 7 | 16 | 14 | 18 |
| vl[k] | 0 | 6 | 6 | 8 | 7 | 10 | 16 | 14 | 18 |

| | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| e[i] | 0 | 0 | 0 | 6 | 4 | 5 | 7 | 7 | 7 | 16 | 14 |
| l[i] | 0 | 2 | 3 | 6 | 6 | 8 | 7 | 7 | 10 | 16 | 14 |



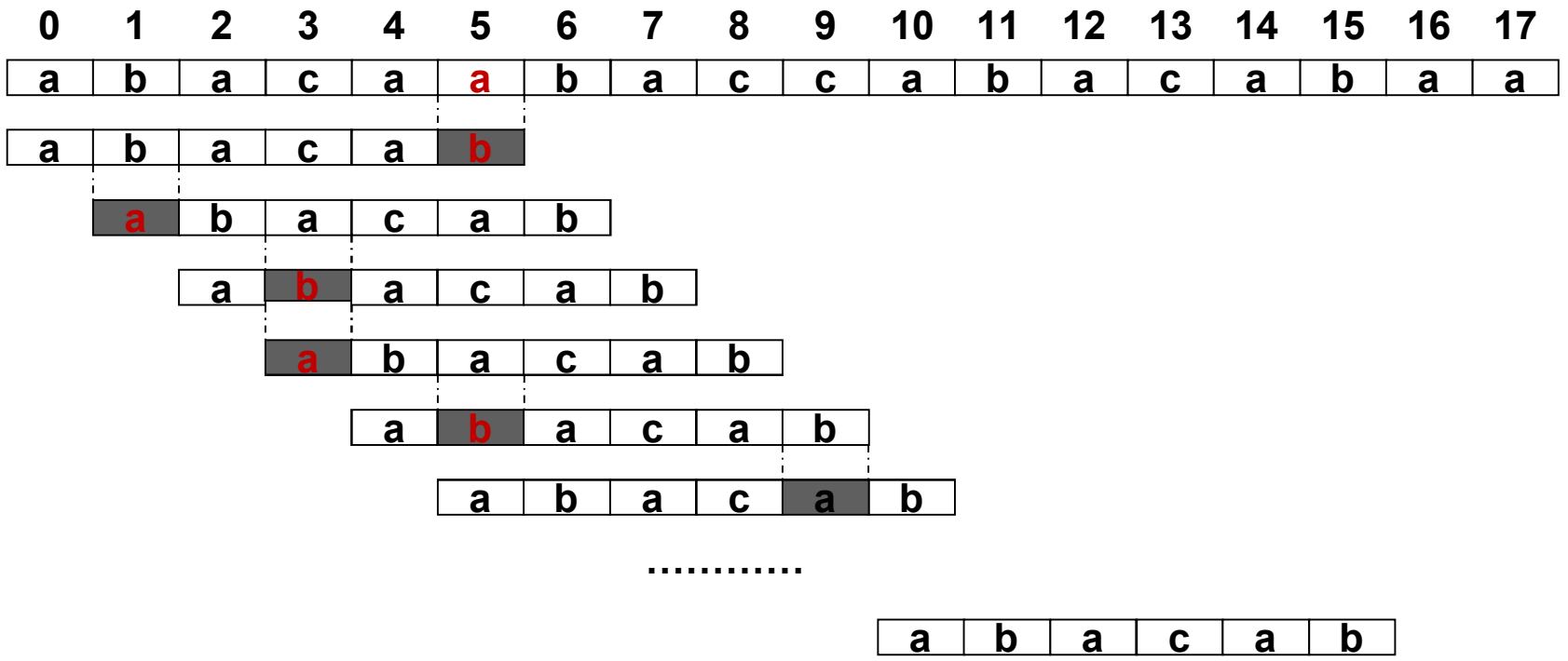

| | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| e[i] | 0 | 0 | 0 | 6 | 4 | 5 | 7 | 7 | 7 | 16 | 14 |
| l[i] | 0 | 2 | 3 | 6 | 6 | 8 | 7 | 7 | 10 | 16 | 14 |

# Search (查找，搜索)

- 静态查找结构
- 动态查找结构
- 散列
- 可扩充散列

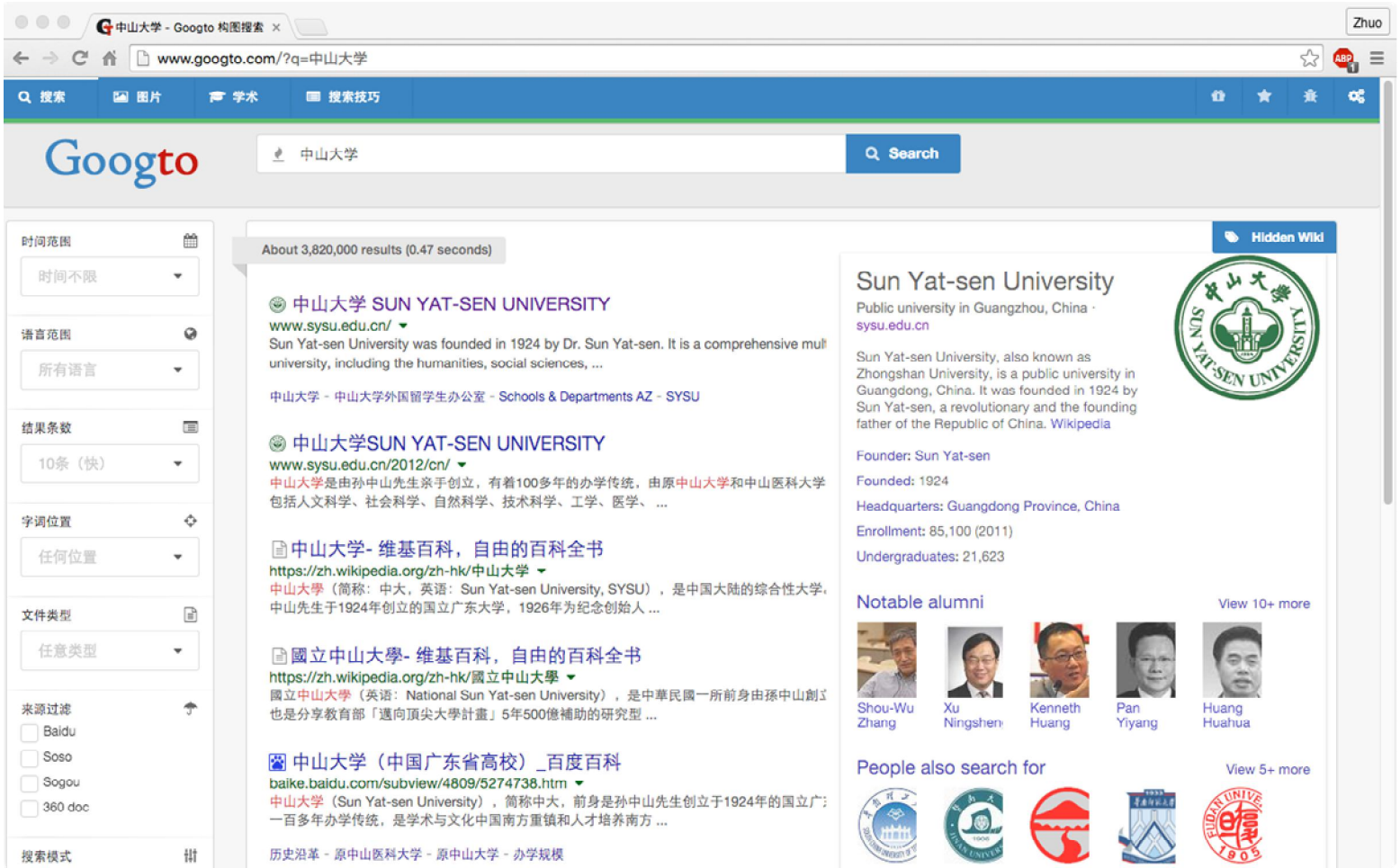
# 回顾字符串匹配

- 子串在主串中的定位称为模式匹配(Pattern Matching)或串匹配(String Matching)。模式匹配成功是指在主串S中能够找到模式串T，否则，称模式串T在主串S中不存在。



# 搜索引擎

- 通过关键词检索词条



# Search

- 数据的组织和查找是大多数应用程序的核心，而查找是所有数据处理中最基本、最常用的操作。特别当查找的对象是一个庞大数量的数据集合中的元素时，查找的方法和效率就显得格外重要。
- 主要讨论顺序表、有序表、树表和哈希表查找的各种实现方法，以及相应查找方法在等概率情况下的平均查找长度。

# 查找的概念

查找表(Search Table)：相同类型的数据元素(对象)组成的集合，每个元素通常由若干数据项构成。

关键字(Key，码)：数据元素中某个(或几个)数据项的值，它可以标识一个数据元素。若关键字能唯一标识一个数据元素，则关键字称为主关键字；将能标识若干个数据元素的关键字称为次关键字。

查找/检索(Searching)：根据给定的K值，在查找表中确定一个关键字等于给定值的记录或数据元素。

- 查找表中存在满足条件的记录：查找成功；结果：所查到的记录信息或记录在查找表中的位置。
- 查找表中不存在满足条件的记录：查找失败。

# 查找的概念

查找有两种基本形式：静态查找和动态查找。

静态查找(Static Search)：在查找时只对数据元素进行查询或检索，查找表称为静态查找表。

动态查找(Dynamic Search)：在实施查找的同时，插入查找表中不存在的记录，或从查找表中删除已存在的某个记录，查找表称为动态查找表。

查找的对象是查找表，采用何种查找方法，首先取决于查找表的组织。查找表是记录的集合，而集合中的元素之间是一种完全松散的关系，因此，查找表是一种非常灵活的数据结构，可以用多种方式来存储。

# 查找的概念

- 根据存储结构的不同，查找方法可分为三大类：

① 顺序表和链表的查找：将给定的K值与查找表中记录的关键字逐个进行比较， 找到要查找的记录；

② 散列表的查找：根据给定的K值直接访问查找表， 从而找到要查找的记录；

③ 索引查找表的查找：首先根据索引确定待查找记录所在的块 ，然后再从块中找到要查找的记录。

# 查找方法评价指标

- 查找过程中主要操作是关键字的比较，查找过程中关键字的平均比较次数(平均查找长度ASL：Average Search Length)作为衡量一个查找算法效率高低的标准。ASL定义为：

$$ASL = \sum_{i=1}^{n} P_i C_i$$

$ASL$是存储结构中对象总数n的函数

$P_i$为检索第$i$个元素的概率

$C_i$为找到第$i$个元素所需的关键码值与给定值的比较次数

# 查找方法评价指标

- 平均检索长度的例子

假设线性表为（a, b, c）检索a、b、c的概率分别为0.4、0.1、0.5

- 顺序检索算法的平均检索长度为$0.4\times1+0.1\times2+0.5\times3=2.1$
- 即平均需要2.1次给定值与表中关键码值的比较才能找到待查元素

# 静态查找

静态查找表的抽象数据类型定义如下：

ADT Static_SearchTable{

数据对象D：D是具有相同特性的数据元素的集合，

各个数据元素有唯一标识的关键字。

数据关系R：数据元素同属于一个集合。

基本操作P：

⋮

} ADT Static_SearchTable

线性表是查找表最简单的一种组织方式，本节介绍几种主要的关于顺序存储结构的查找方法。

# 静态查找--顺序查找(Sequential Search)

## 查找思想

从表的一端开始逐个将记录的关键字和给定K值进行比较，若某个记录的关键字和给定K值相等，查找成功；否则，若扫描完整个表，仍然没有找到相应的记录，则查找失败。顺序表的类型定义如下：

```
#define MAX_SIZE  100
typedef  struct  SSTable
    {  RecType  elem[MAX_SIZE] ;   /*  顺序表  */
        int  length ;    /*  实际元素个数  */
    }SSTable ;
```

# 静态查找--顺序查找(Sequential Search)

```
int  Seq_Search(SSTable  ST , KeyType key)
{   int p ;
    ST. elem[0].key=key ;    /* 设置监视哨兵,失败返回0 */
    for (p=ST.length; !EQ(ST. elem[p].key, key); p--)
    return(p) ;
 }
```

比较次数：

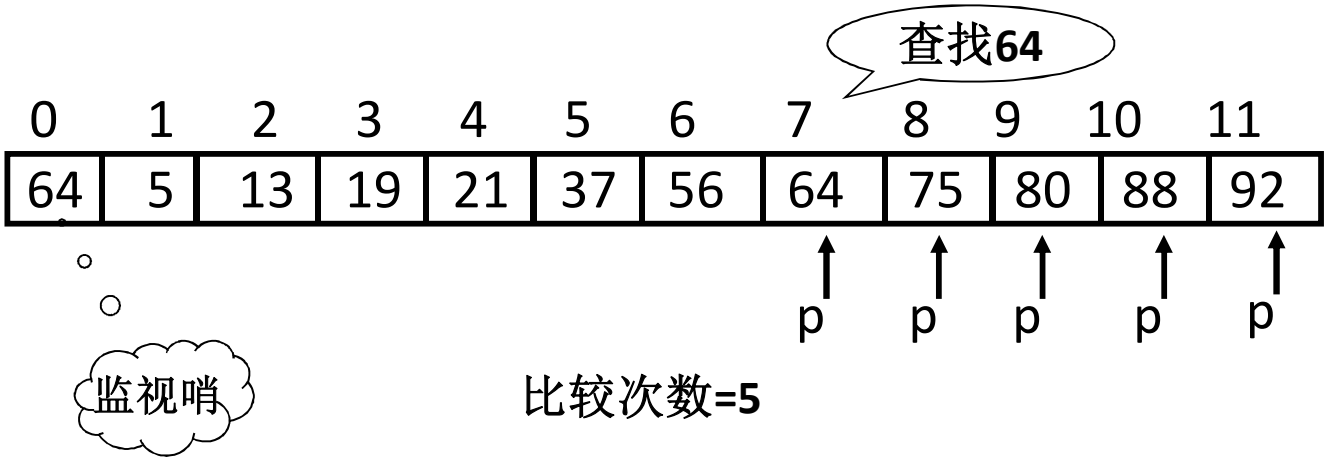
查找第n个元素：　1

…….

查找第i个元素：　n-i+1

查找第1个元素：　n

查找失败：　n+1




顺序查找示例图

# 静态查找--顺序查找(Sequential Search)

- 算法分析
- 检索成功

  假设检索每个关键码是等概率的：$P_i = 1/n$

$$\sum_{i=0}^{n-1} P_i \bullet (n-i) = \frac{1}{n} \sum_{i=0}^{n-1} (n-i)$$

$$= \frac{n+1}{2}$$

- 检索失败

  假设检索失败时都需要比较$n+1$次（设置了一个监视哨）

# 静态查找--顺序查找(Sequential Search)

- 假设检索成功的概率为$p$，检索失败的概率为$q=(1-p)$，则平均查找长度ASL：

$$
\begin{aligned}
ASL &= p \cdot \left(\frac{n+1}{2}\right) + q \cdot (n+1) \\
&= p \cdot \left(\frac{n+1}{2}\right) + (1-p)(n+1) \\
&= (n+1)(1-p/2)
\end{aligned}
$$

- $(n+1)/2 < ASL < (n+1)$

# 静态查找--顺序查找(Sequential Search)

- 顺序查找优缺点
- 优点：插入元素可以直接加在表尾O(1)
- 缺点：检索时间太长O(n)

# 静态查找--折半查找(Binary Search)

折半查找又称为二分查找，是一种效率较高的查找方法。

**前提条件**：查找表中的所有记录是按关键字有序(升序或降序)。

查找过程中，先确定待查找记录在表中的范围，然后逐步缩小范围(**每次将待查记录所在区间缩小一半**)，直到找到或找不到记录为止。

# 静态查找--折半查找(Binary Search)

- **查找思想**

用Low、High和Mid表示待查找区间的下界、上界和中间位置指针，初值为Low=1，High=n。

(1) 取中间位置Mid：Mid=⌊(Low+High)/2⌋；

(2) 比较中间位置记录的关键字与给定的K值：

① 相等： 查找成功；

② 大于：待查记录在区间的前半段，修改上界指针：High=Mid-1，转(1)；

③ 小于：待查记录在区间的后半段，修改下界指针：Low=Mid+1，转(1)；

直到越界(Low>High)，查找失败。

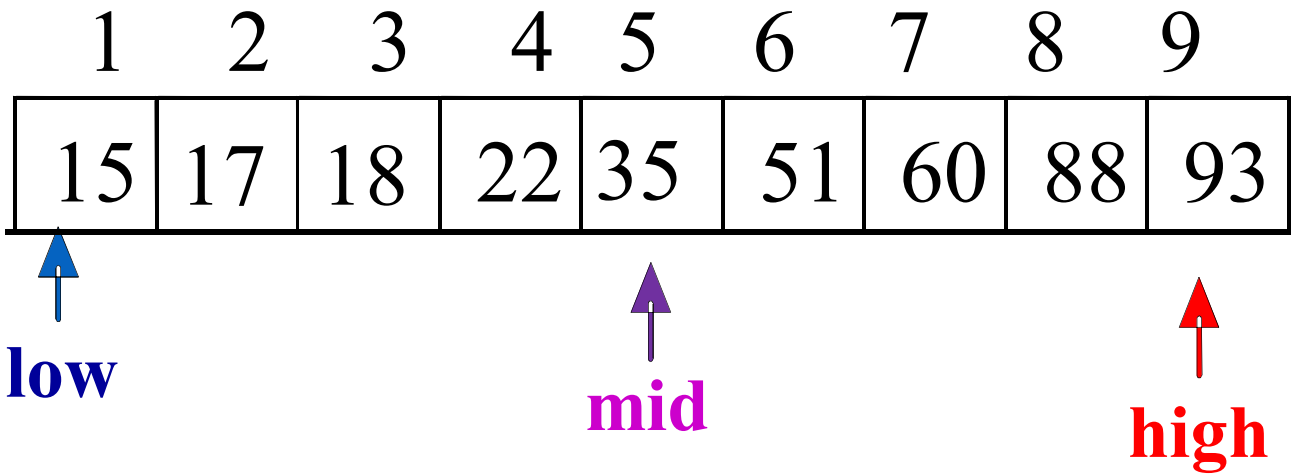
# 静态查找--折半查找(Binary Search)

- **算法实现**

```
int  Bin_Search(SSTable  ST , KeyType  key)
    {   int  Low=1 , High=ST.length, Mid ;
        while (Low<High)
            {    Mid=(Low+High)/2 ;      % 确定中点位置
                 if  (EQ(ST. elem[Mid].key, key))
                     return(Mid) ;
                 else if (LT(ST. elem[Mid].key, key))
                             Low=Mid+1 ;       % 从低位进行查找
                         else   High=Mid-1 ;  % 从高位进行查找
            }
        return(0) ;      /*   查找失败  */
    }
```

# 静态查找--折半查找(Binary Search)

- **关键码18 low=1 high=9**



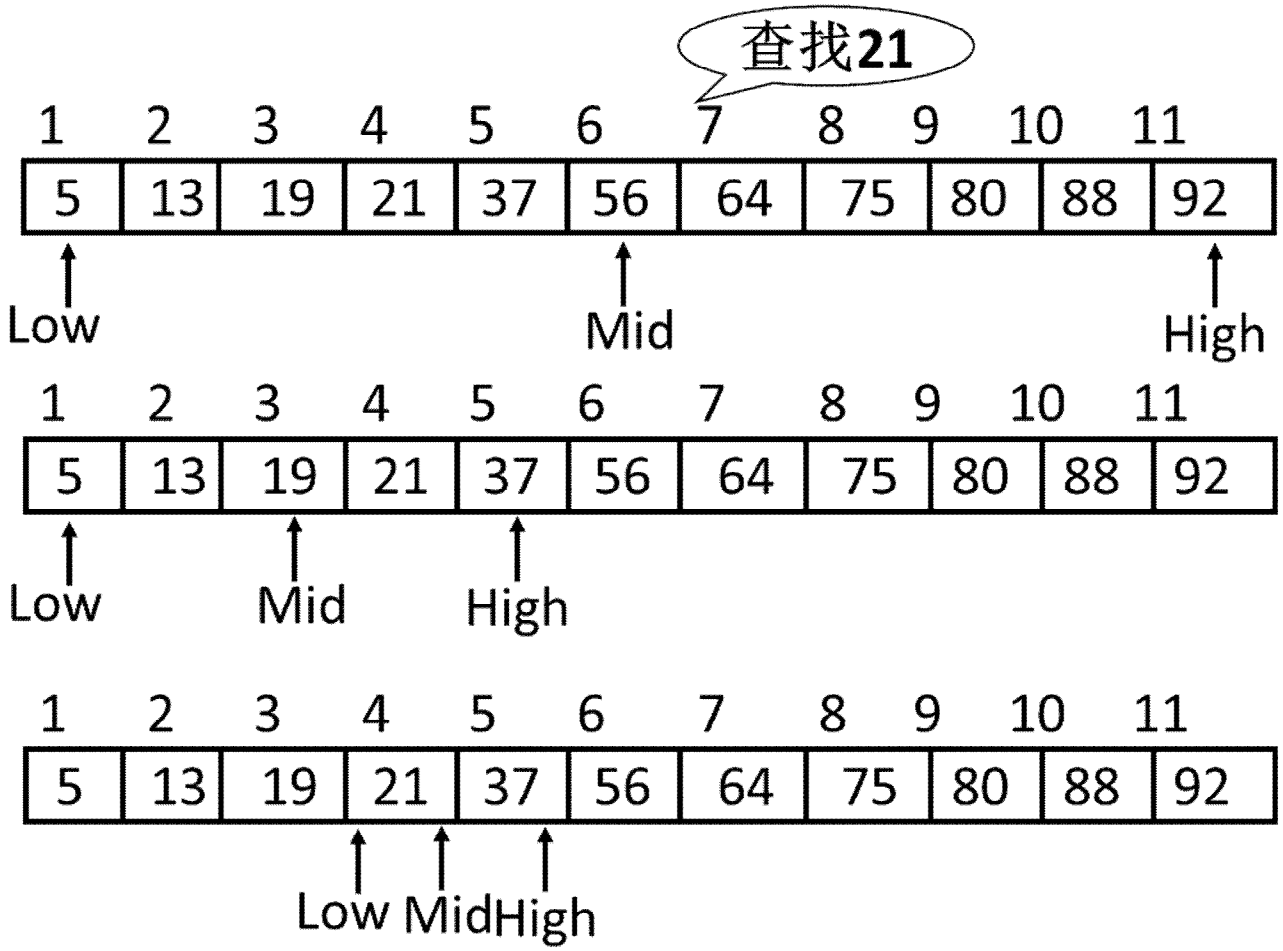

**第一次：l=1, h=9, mid=5; array[5]=35>18**

**第二次：l=1, h=4, mid=2; array[2]=17<18**

**第三次：l=3, h=4, mid=3; array[3]=18＝18**

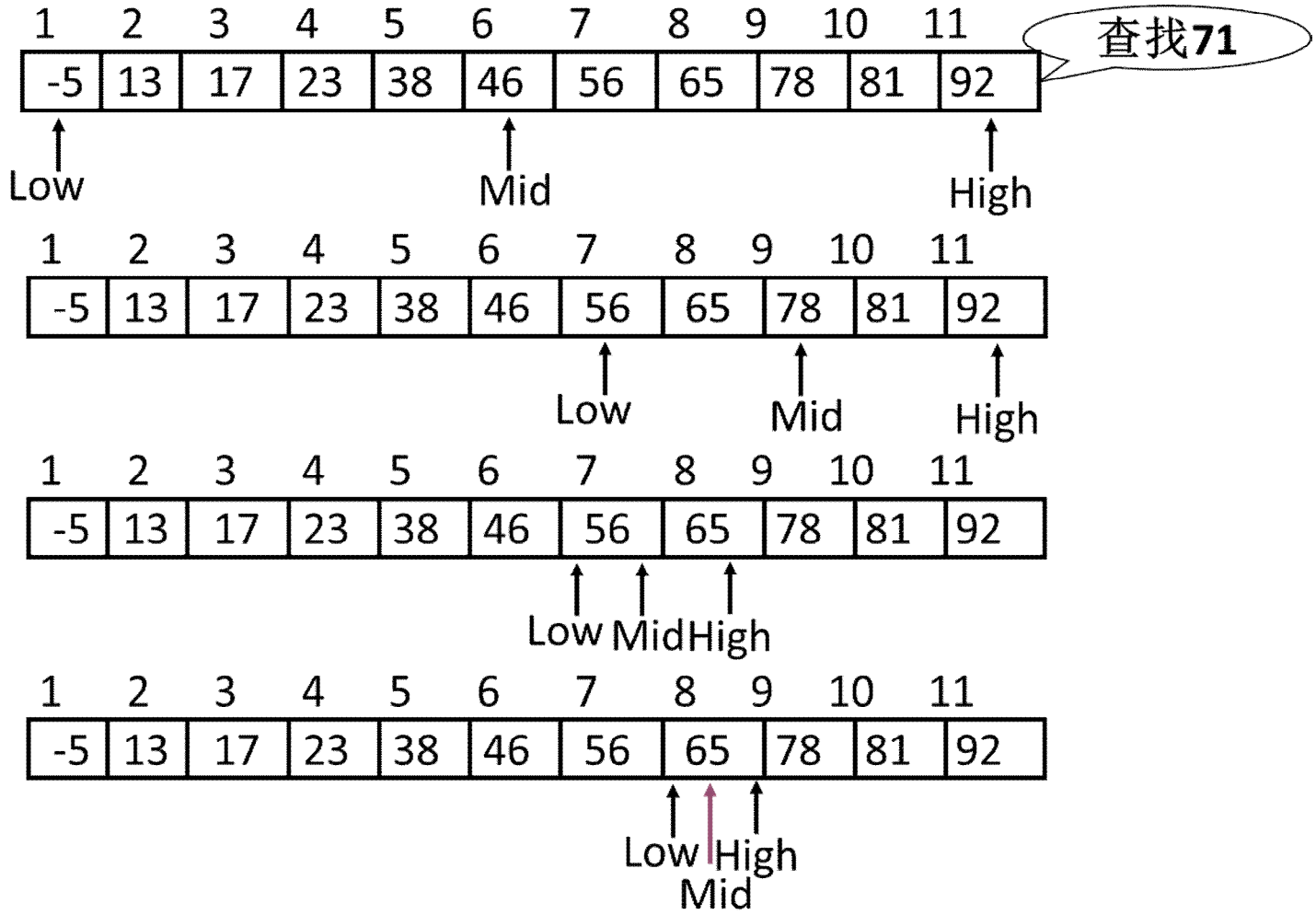
# 静态查找--折半查找(Binary Search)

- 查找成功示例



# 静态查找--折半查找(Binary Search)

- 查找不成功示例



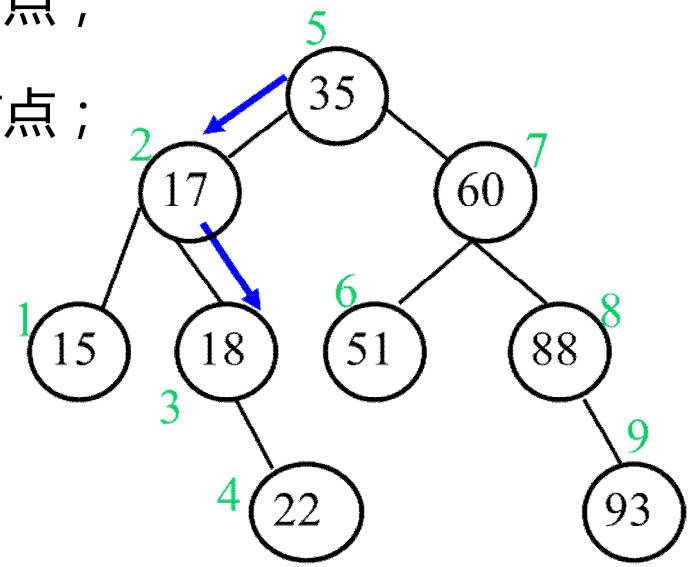
# 静态查找--折半查找(Binary Search)

- 算法分析

① 查找时每经过一次比较，查找范围就缩小一半，该过程可用一棵二叉树表示：

- 根结点就是第一次进行比较的中间位置的记录；
- 排在中间位置前面的作为左子树的结点；
- 排在中间位置后面的作为右子树的结点；

- 对各子树来说都是相同的。这样所得到的二叉树称为**判定树**(Decision Tree)。



# 静态查找--折半查找(Binary Search)

- 算法分析

② 将二叉判定树的第$\lfloor \log_2 n \rfloor+1$层上的结点补齐就成为一棵满二叉树，深度不变，$h=\lfloor \log_2(m+1) \rfloor$。(m：补齐之后的节点数)

③ 由满二叉树性质知，第 h 层上的结点数为$2^{h-1}$，设表中每个记录的查找概率相等，即$P_i=1/n$，查找成功时的平均查找长度ASL：

$$ASL=\sum_{i=1}^{n} P_i \times C_i = \frac{1}{n}\sum_{j=1}^{h} j \times 2^{j-1} = \frac{n+1}{n}\log_2(n+1)-1$$

当n很大 (n>50)时， $ASL \approx \log_2(n+1)-1$。

# 静态查找--分块查找

- **分块查找**(Blocking Search)又称索引顺序查找，是前面两种查找方法的综合。
- **查找表的组织**

  ① 将查找表分成几块。块间有序，即第i+1块的所有记录关键字均大于(或小于)第i块记录关键字；块内无序。

  ② 在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：

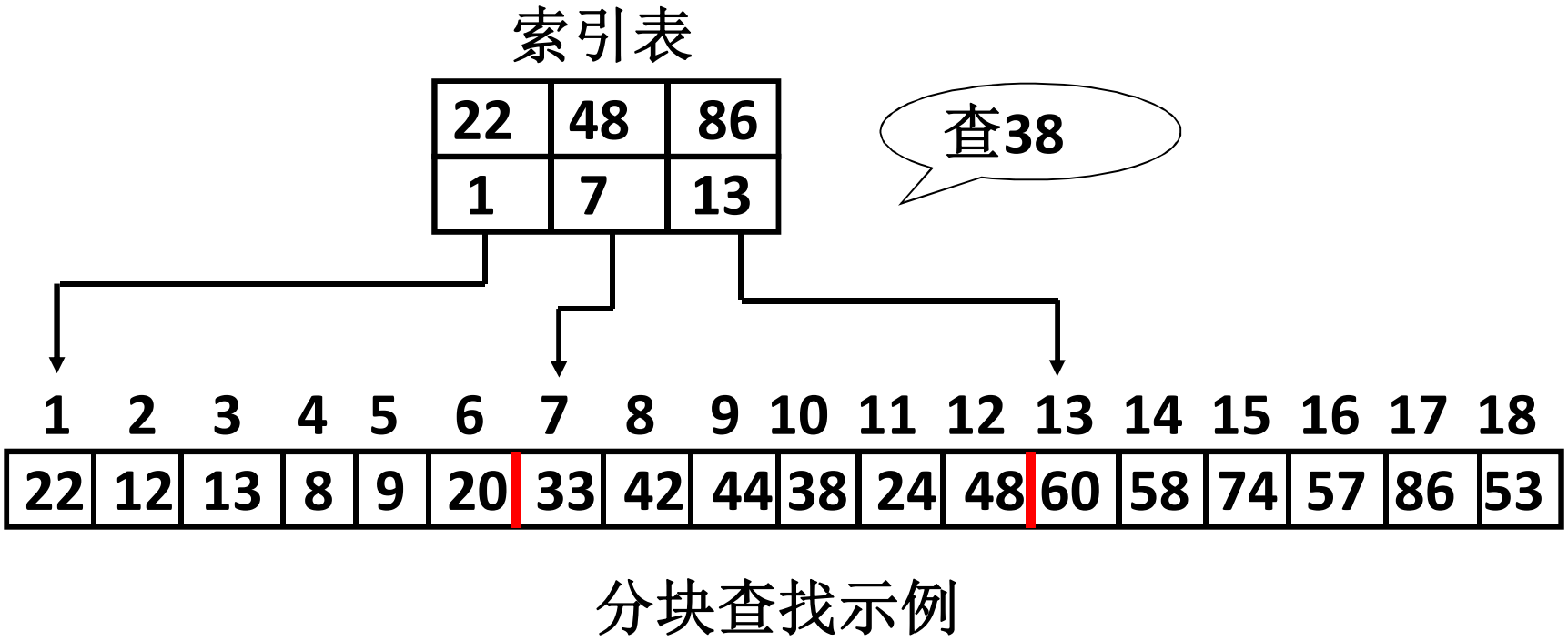
| 最大关键字 |
| --- |
| 起始指针 |

- **查找思想**

  先确定待查记录所在块，再在块内查找(顺序查找)。

# 静态查找--分块查找

- **算法示例**




分块查找示例

# 静态查找--分块查找

```
int Block_search(RecType ST[] , Index ind[] , KeyType key , int n , int b)
    /*  在分块索引表中查找关键字为key的记录 */
    /*表长为n ， 块数为b */
    {   int i=0 , j , k ;
        while ((i<b)&&LT(ind[i].maxkey, key) )  i++  ;
            if (i>b) {  printf("\nNot found");   return(0);  }
        j=ind[i].startpos ;
        while ((j<n)&&LQ(ST[j].key, ind[i].maxkey) )
            {   if ( EQ(ST[j].key, key) )  break ;
                j++ ;
            }    /*  在块内查找  */
        if (j>n||!EQ(ST[j].key, key) )
            {  j=0; printf("\nNot found");  }
        return(j);
    }
```

# 静态查找--分块查找

- 性能分析
- 分块检索为两级检索
  - 先在索引表中确定待查元素所在的块；
    - 设在索引表中确定块号的时间开销是$\mathrm{ASL}_b$
  - 然后在块内检索待查的元素。
    - 在块中查找记录的时间开销为$\mathrm{ASL}_w$
- $\mathrm{ASL}(n) = \mathrm{ASL}_b + \mathrm{ASL}_w$

# 静态查找--分块查找

- 假设在索引表中用顺序检索，在块内也用顺序检索，设表长为n个记录，均分为b块，每块记录数为s，则b=[n/s]。设记录的查找概率相等，每块的查找概率为1/b，块中记录的查找概率为1/s，则平均查找长度ASL：

$$ASL_b = \frac{b+1}{2} \qquad ASL_w = \frac{s+1}{2}$$

$$ASL = \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1$$

$$= \frac{n+s^2}{2s} + 1$$

- 当$s = \sqrt{n}$ 时，ASL取最小值，

$$ASL = \sqrt{n} + 1 \approx \sqrt{n}$$

# 静态查找--分块查找

- 分块检索的优缺点
- 优点：
  - 插入、删除相对较易
  - 没有大量记录移动
- 缺点：
  - 增加一个辅助数组的存储空间
  - 初始线性表分块排序
  - 当大量插入/删除时，或结点分布不均匀时，速度下降

# 静态查找

- 查找方法比较

| | 顺序查找 | 折半查找 | 分块查找 |
|---|---|---|---|
| ASL | 最大 | 最小 | 两者之间 |
| 表结构 | 有序表、无序表 | 有序表 | 分块有序表 |
| 存储结构 | 顺序存储结构<br>线性链表 | 顺序存储结构 | 顺序存储结构<br>线性链表 |

# 动态查找

- 当查找表以线性表的形式组织时，若对查找表进行插入、删除或排序操作，就必须移动大量的记录，当记录数很多时，这种移动的代价很大。
- 利用树的形式组织查找表，可以对查找表进行动态高效的查找。

# 动态查找--二叉排序树(BST)

**二叉排序树**(Binary Sort Tree或Binary Search Tree) 的定义为：二叉排序树或者是空树，或者是满足下列性质的二叉树。

(1) 若左子树不为空，则左子树上所有结点的值(关键字)都小于根结点的值；

(2) 若右子树不为空，则右子树上所有结点的值(关键字)都大于根结点的值；

(3) 左、右子树都分别是二叉排序树。

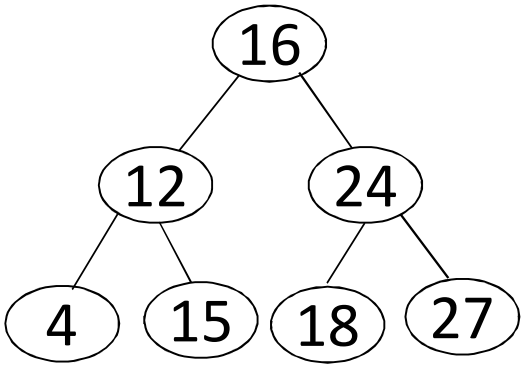
**结论**：若按中序遍历一棵二叉排序树，所得到的结点序列是一个递增序列。

# 动态查找--二叉排序树(BST)

- BST仍然可以用二叉链表来存储，如下图所示。

结点类型定义如下：

```
typedef  struct  Node
    {  KeyType  key ;    /*  关键字域  */
        …      /*  其它数据域  */
        struct  Node  *Lchild , *Rchild ;
    }BSTNode ;
```




二叉排序树

# 动态查找--二叉排序树(BST)

**BST树的查找--思想**

首先将给定的K值与二叉排序树的根结点的关键字进行比较：若相等： 则查找成功；

① 给定的K值小于BST的根结点的关键字：继续在该结点的左子树上进行查找；

② 给定的K值大于BST的根结点的关键字：继续在该结点的右子树上进行查找。

# 动态查找--二叉排序树(BST)

- 算法实现

**(1) 递归算法**

```
BSTNode *BST_Serach(BSTNode *T , KeyType key)
    {  if (T==NULL)  return(NULL) ;
        else
            {  if  (EQ(T->key, key) )  return(T) ;
                else if ( LT(key, T->key) )
                        return(BST_Serach(T->Lchild, key)) ;
                    else  return(BST_Serach(T->Rchild, key)) ;
            }
    }
```

# 动态查找--二叉排序树(BST)

**(2) 非递归算法**

```
BSTNode *BST_Serach(BSTNode *T , KeyType key)
    {  BSTNode p=T ;
       while (p!=NULL && !EQ(p->key, key) )
            {  if ( LT(key, p->key) )  p = p->Lchild ;
                 else p = p->Rchild ;
            }
       if  (EQ(p->key, key) )  return(p) ;
       else return(NULL) ;
    }
```

在随机情况下，二叉排序树的**平均查找长度**ASL和Log(n)(树的深度)是等数量级的。

# 二叉排序树(BST)--插入结点

- **BST树的插入--思想**
- 在BST树中插入一个新结点x时，若BST树为空，则令新结点x为插入后BST树的根结点；否则，将结点x的关键字与根结点T的关键字进行比较：

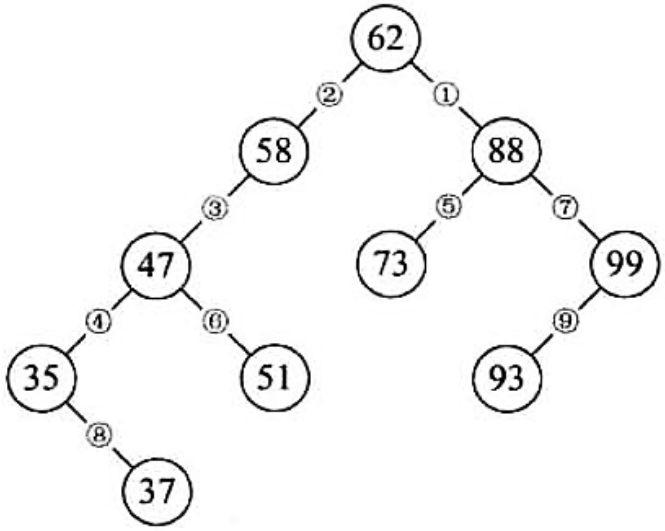
  ① 若相等： 不需要插入；

  ② 若x.key < T->key：结点x插入到T的左子树中；

  ③ 若x.key > T->key：结点x插入到T的右子树中。

# 二叉排序树(BST)--插入结点

- 建立二叉排序树的例子
- 有集合 { 62, 88, 58, 47, 35, 73, 51, 99, 37, 93 }




- 对上述二叉树进行中序遍历时，就可以得到一个有序的序列 { 35, 37, 47, 51, 58, 62, 73, 88, 93, 99 }

# 二叉排序树(BST)--插入结点

- 算法实现

(1) 递归算法

```
void  Insert_BST (BSTNode *T , KeyType  key)
    {  BSTNode *x ;
        x =(BSTNode *) malloc(sizeof(BSTNode)) ;
        x->key=key; x->Lchild=x->Rchild=NULL ;
        if (T==NULL)  T=x ;
        else
            {  if (EQ(T->key, x->key) ) return  ; /*  已有结点  */
                else if (LT(x->key, T->key) )
                        Insert_BST(T->Lchild, key) ;
                    else   Insert_BST(T->Rchild, key) ;   }
    }
```

# 二叉排序树(BST)--插入结点

(2) 非递归算法

```
void Insert_BST (BSTNode *T , KeyType key)
    {  BSTNode *x, *p , *q ;
        x=(BSTNode *)malloc(sizeof(BSTNode)) ;
        X->key=key; x->Lchild=x->Rchild=NULL ;
        if (T==NULL)  T=x ;
        else
            {  p=T ;
               while (p!=NULL)
                  {  if  (EQ(p->key, x->key) )  return  ;
                     q=p ;   /*q作为p的父结点  */
                      if (LT(x->key, p->key) )  p=p->Lchild ;
                     else p=p->Rchild ;
                  }
               if (LT(x->key, q->key) )  q->Lchild=x ;
               else q->Rchild=x ;
            }
    }
```

# 二叉排序树(BST)--插入结点

- 特点分析：
- 由结论知，对于一个无序序列可以通过构造一棵BST树而变成一个有序序列。
- 由算法知，每次插入的新结点都是BST树的叶子结点，即在插入时不必移动其它结点，仅需修改某个结点的指针。

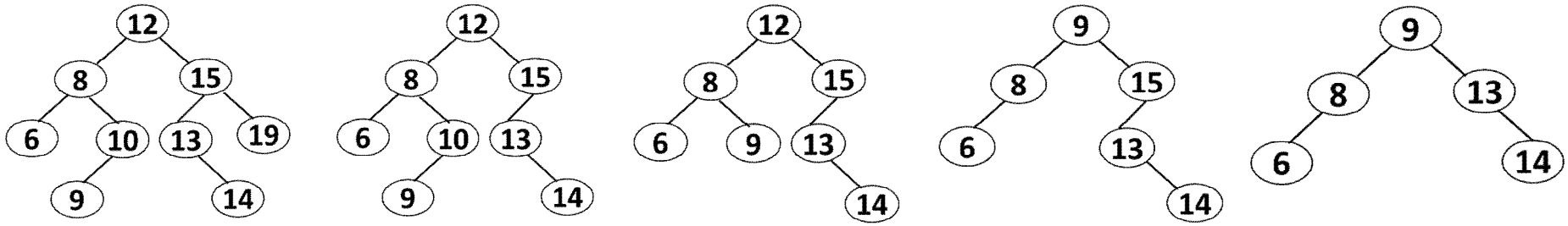
# 二叉排序树(BST)--删除结点

- 删除操作过程分析

从BST树上删除一个结点，仍然要保证删除后满足BST的性质。设被删除结点为p，其父结点为f，删除情况如下：

① 若p是叶子结点： 直接删除p，如图下图(b)所示。

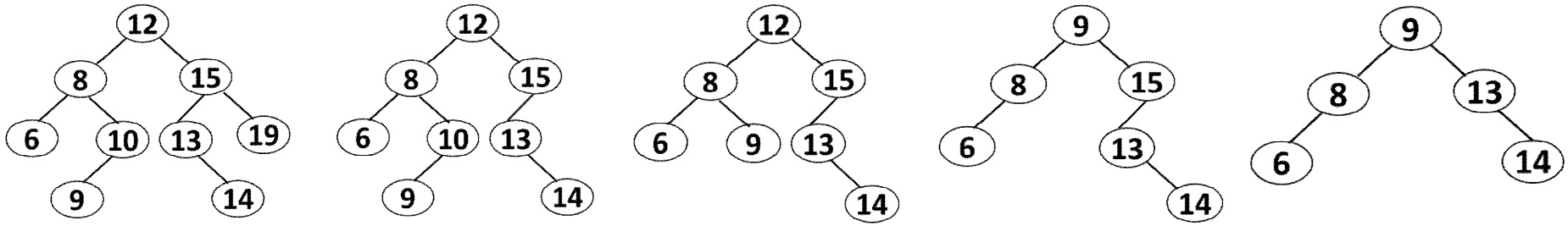
② 若p只有一棵子树(左子树或右子树)：直接用p的左子树(或右子树)取代p的位置而成为f的一棵子树。即原来p是f的左子树，则p的子树成为f的左子树；原来p是f的右子树，则p的子树成为f的右子树，如图 (c)、 (e)所示。



# 二叉排序树(BST)--删除结点

③ 若p既有左子树又有右子树 ：处理方法有以下两种，可以任选其中一种。

- 用p的直接前驱结点代替p。即从p的左子树中选择值最大的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的左子树中的最右边的结点且没有右子树，对s的删除同②，如图 (d)所示。
- 用p的直接后继结点代替p。即从p的右子树中选择值最小的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的右子树中的最左边的结点且没有左子树，对s的删除同②。



# 二叉排序树(BST)--删除结点

```
void Delete_BST (BSTNode *T , KeyType  key )
   /*  在以T为根结点的BST树中删除关键字为key的结点   */
       {  BSTNode *p=T , *f=NULL , *q , *s ;
            while ( p!=NULL && !EQ(p->key, key) )
                  {  f=p ;
                        if (LT(key, p->key) ) p=p->Lchild ;  /*  搜索左子树 */
                        else p=p->Rchild ;  /*  搜索右子树 */
                  }
            if  (p==NULL)  return ;     /*  没有要删除的结点   */
            s=p ;     /*  找到了要删除的结点为p   */
            if (p->Lchild != NULL && p->Rchild != NULL)
                  {  f=p ; s=p->Lchild ;        /*  从左子树开始找   */
                        while (s->Rchild != NULL)
                           {   f=s ; s=s->Rchild ;   }
                        /*   左、右子树都不空，找左子树中最右边的结点  */
                        p->key = s->key ; p->otherinfo = s->otherinfo ;
                              /*  用结点s的内容替换结点p内容  */
                  }  /*  将第3种情况转换为第2种情况*/
            if  (s->Lchild != NULL)   /*  若s有左子树，右子树为空 */
                  q=s->Lchild ;
              else q=s->Rchild ;
            if  (f==NULL)  T=q ;
                  else if (f->Lchild==s)  f->Lchild=q ;
                           else f->Rchild=q ;
            free(s) ;
       }
```

# 二叉排序树的问题

- 对于一般的二叉排序树，其期望高度（即为一棵平衡树时）为$\log_2 n$，其各操作的时间复杂度（$O(\log_2 n)$）同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的），二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即$O(n)$。
- 针对这种情况，可以通过随机化建立二叉搜索树来尽量避免。但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，使得操作的时间复杂度提高。

# 平衡二叉树（AVL树）的定义

- 平衡二叉树或者是空树，或者是满足下列性质的二叉树

  (1)：左子树和右子树深度之差的绝对值不大于1；

  (2)：左子树和右子树也都是平衡二叉树。

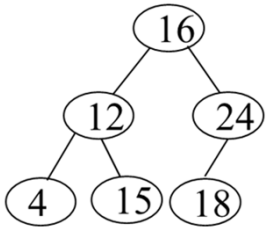


图9-6 平衡二叉树

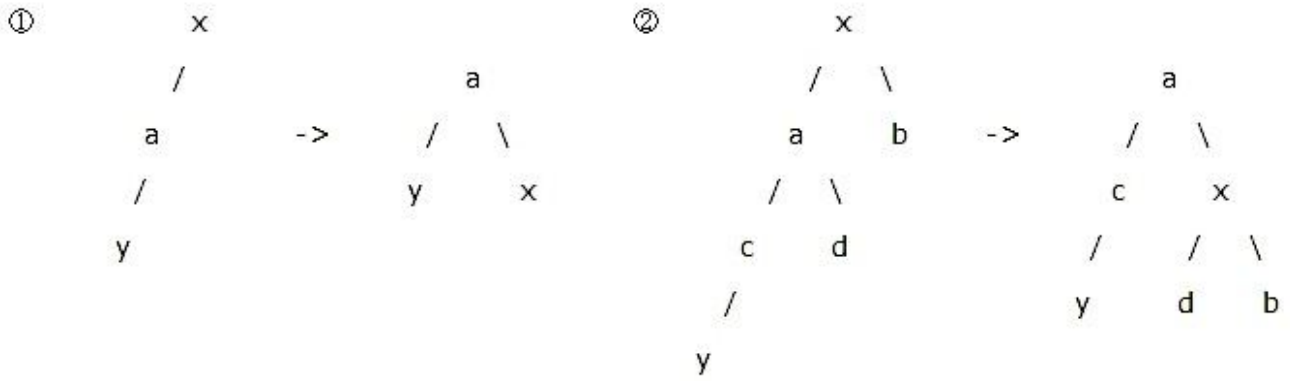
- 平衡因子(Balance Factor) ：二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子。
- 因此，平衡二叉树上每个结点的平衡因子只可能是-1、0和1，否则，只要有一个结点的平衡因子的绝对值大于1， 该二叉树就不是平衡二叉树。

# 平衡化旋转

- 一般的二叉排序树是不平衡的，若能通过某种方法使其既保持有序性，又具有平衡性，就找到了构造平衡二叉排序树的方法，该方法称为平衡化旋转。
- 在对AVL树进行插入或删除一个结点后，通常会影响到从根结点到插入(或删除)结点的路径上的某些结点，这些结点的子树可能发生变化。以插入结点为例，影响有以下几种可能性
  - 以某些结点为根的子树的深度发生了变化；
  - 某些结点的平衡因子发生了变化；
  - 某些结点失去平衡。

# 平衡化旋转

- 1　LL型平衡化旋转



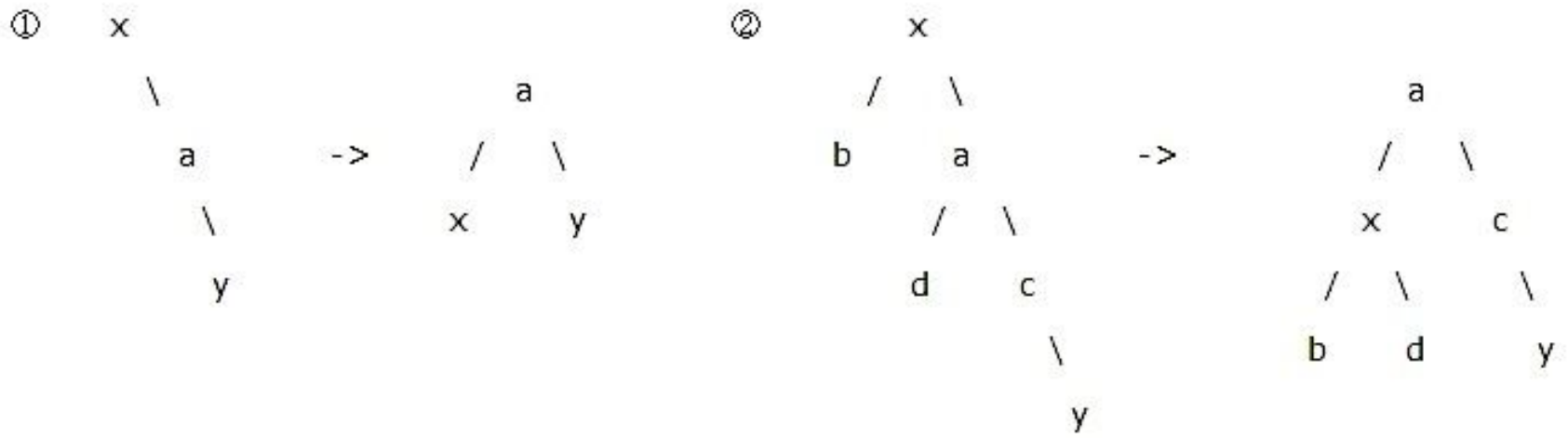

图①：结点x和结点a变换，则树平衡了；

图②是树中的一般情况，a结点有右孩子d，那要进行x和a变换，那么a的右孩子放哪？

分析：x>d, d>a,所以d可作为x的左孩子，且可作为a的右孩子中的孩子。

# 平衡化旋转

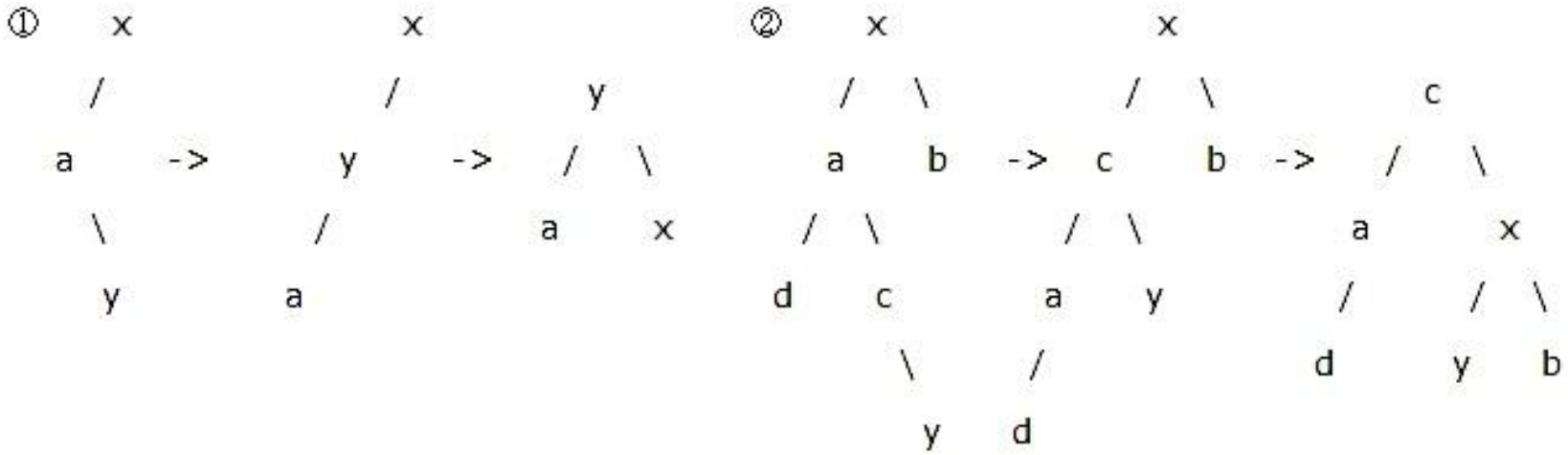
- 2　RR型平衡化旋转




实现：找到根结点x，与它的右孩子a进行交换即可使二叉树树再次平衡。

# 平衡化旋转

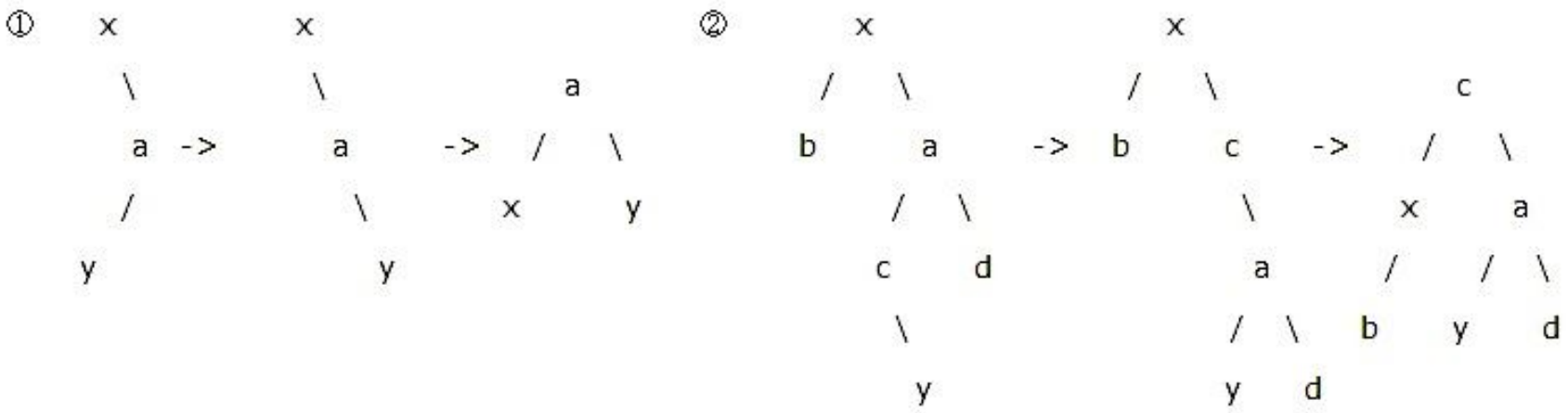
- 3　LR型平衡化旋转




需要进行两次交换，才能达到平衡，注意这时y是c的右孩子，最终y作为x的左孩子；若y是c的左孩子，最终y作为a的右孩子。

# 平衡化旋转

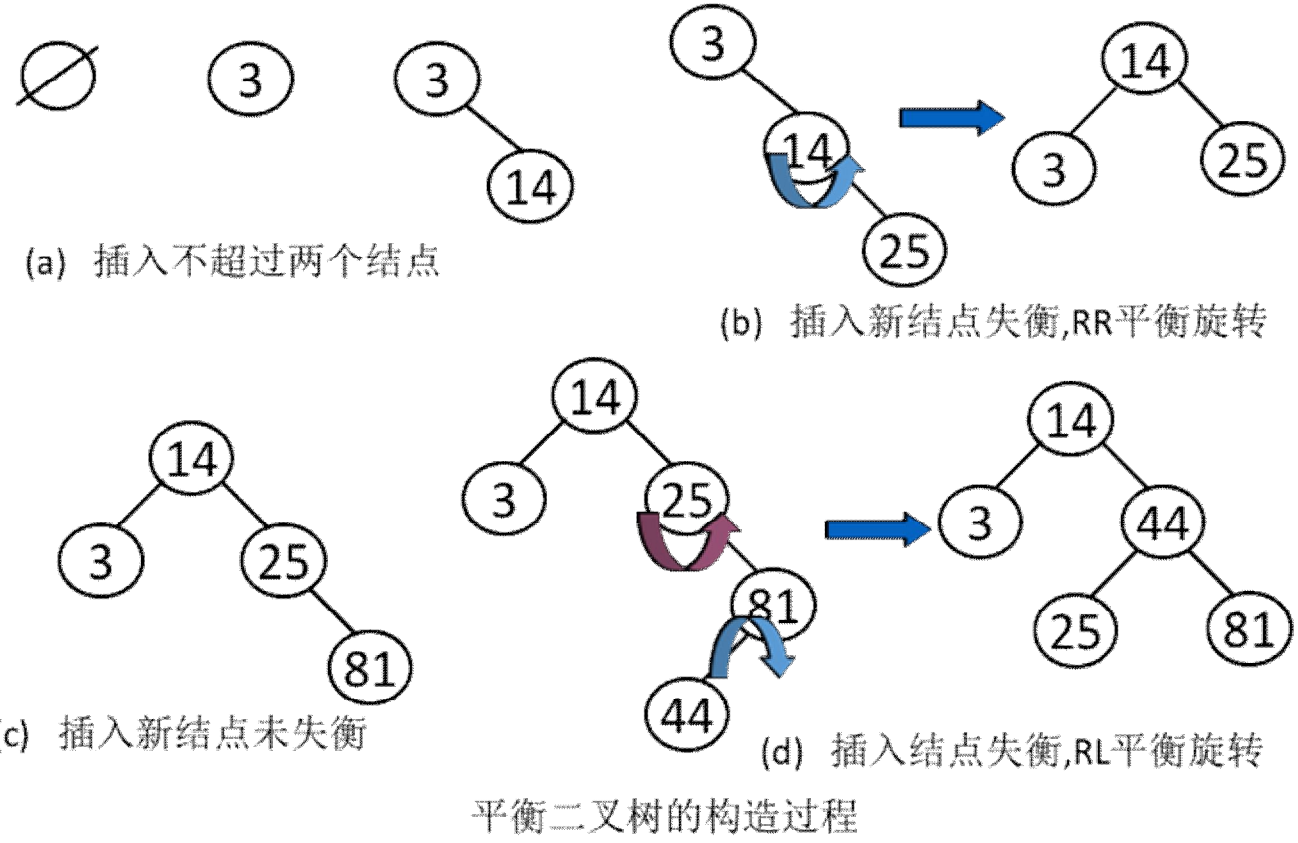
- 4　RL型平衡化旋转




实现：找到根结点x，让x的右孩子a与x的右孩子a的左孩子c进行交换，然后再让x与x此时的右孩子c进行交换，最终达到平衡。

# Case

- 例： 设要构造的平衡二叉树中各结点的值分别是(3, 14, 25, 81, 44)，平衡二叉树的构造过程如图所示。




平衡二叉树的构造过程