数据结构

主讲：项若曦　助教：申智铭、黄毅
rxxiang@blcu.edu.cn
主楼南329

回顾

- **图的定义、概念、性质**
- **图的存储：**
  - 邻接矩阵、邻接表
  - 操作：创建图等
- **图的遍历**
  - BFS、DFS
  - 基于遍历的算法设计
  - 最小生成树（MST）、最短路径（SP）
- **图的应用**
  - 有向无环图（DAG）、拓扑排序、AOE/AOV网络

回顾

- **图的遍历——基于遍历的算法设计**
  - 基于遍历的算法设计
    ——哈密顿路径、环（DFS，注意回溯方法）
    ——最短路径最长的顶点（BFS）
  - 最小生成树（MST）、最短路径（SP）
- **图的应用**
  - 有向无环图（DAG）、拓扑排序、AOE/AOV网络

图的遍历算法的应用

- **2. 求距v_0最短路径长度最长的一个顶点 （BFS的应用）**

```
1. int  MaxDistance(MGraph G, int v0){
2.          for(i=0; i<G.vexnum; i++)
3.                  visited[i]=FALSE;
4.          InitQueue(Q);
5.          EnQueue(Q, v0);
6.          visited[v0] = TRUE;
7.          while( !QueueEmpty(Q)){
8.                  DeQueue(Q, v);
9.                  for(w = 0; w < G.vexnum; w++)
10.                         if(G.arcs[v][w].adj && !visited[w] ){
11.                                 visited[w]=TRUE;
12.                                 EnQueue(Q, w);
13.                         } // ~for_if
14.         } //~while
15.         return(v);
16.}
```

回顾

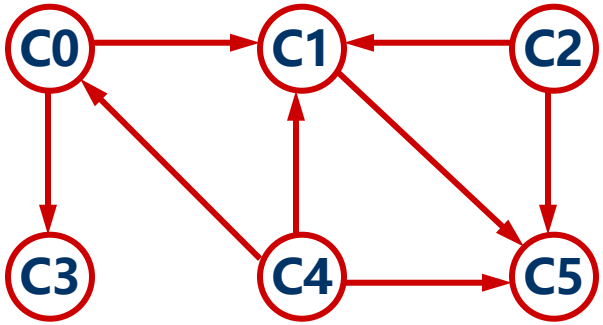
- 有向无环图(DAG)
- 拓扑排序：AOV网络、 AOE网络

拓扑排序(Topological Sort)

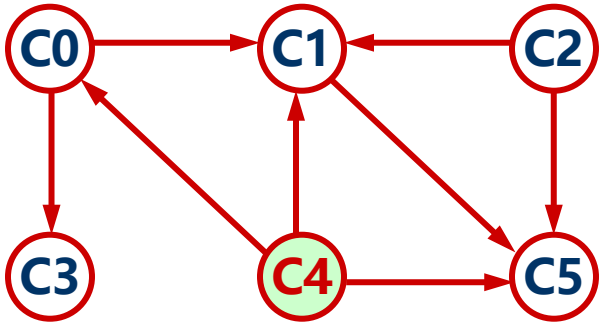
拓扑排序方法

1. 输入AOV网络。令 n 为顶点个数。
2. 在AOV网络中选一个没有直接前驱的顶点，并输出之;
3. 从图中删去该顶点，同时删去所有它发出的有向边;
4. 重复以上 2、3步, 直到
   - 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
   - 图中还有未输出的顶点, 但已跳出处理循环。说明图中还剩下一些顶点, 它们都有直接前驱。这时网络中必存在有向环。

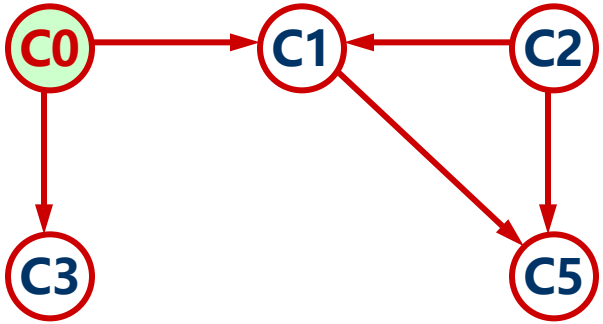
拓扑排序(Topological Sort)过程(1)



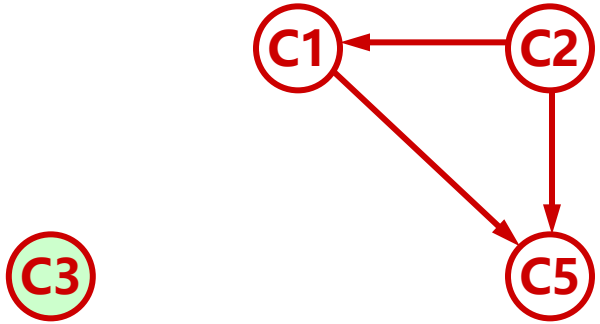
(a) 有向无环图



(b) 输出顶点C4

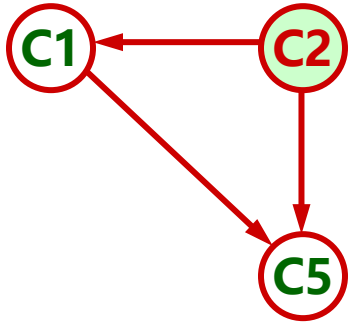


(c) 输出顶点C0

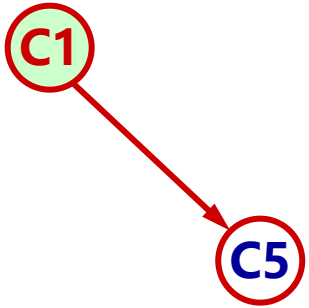


(d) 输出顶点C3

拓扑排序(Topological Sort)过程(2)



(e) 输出顶点C2



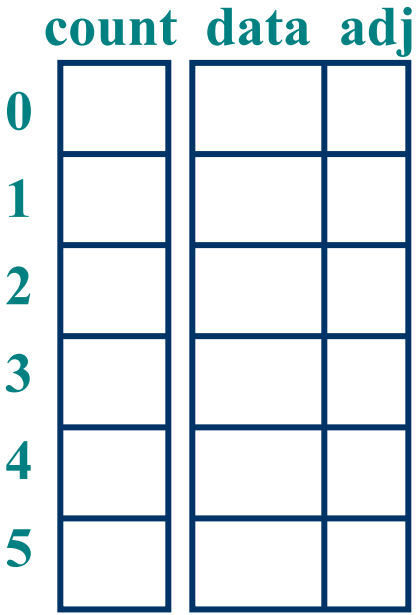
(f) 输出顶点C1

(g) 输出顶点C5

(h) 拓扑排序完成

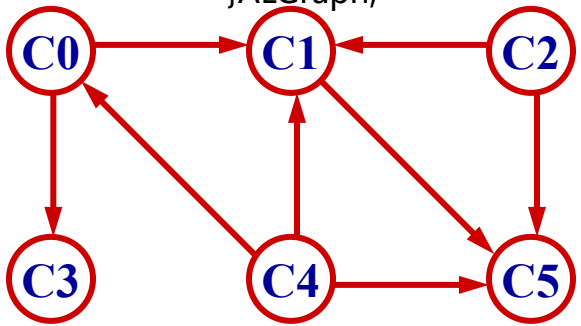
- 最后得到的拓扑有序序列为 C4, C0, C3, C2, C1, C5。它满足图中给出的所有前驱和后继关系，对于本来没有这种关系的顶点，如C4和C2，也排出了先后次序关系。

AOV网络及其邻接表表示

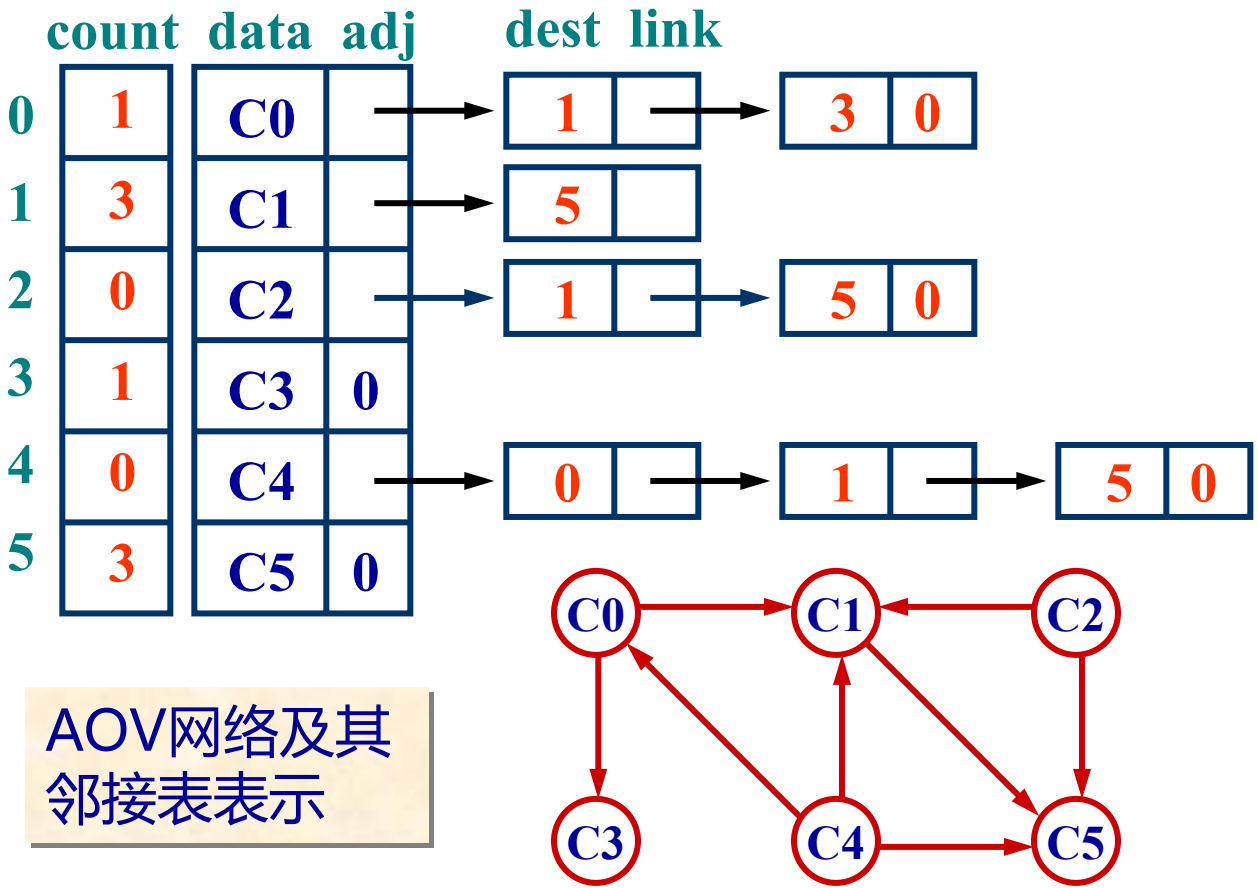


AOV网络及其邻接表表示

```
#define MVNum 100                    //最大顶点数
typedef struct ArcNode{              //边结点
    int adjvex;                      //该边所指向的顶
    struct ArcNode * nextarc;        //指向下一条边的
    EdgeData cost;                   //改造结点，type
}ArcNode;
typedef struct VNode{
    VerTexType data;                 //顶点信息
    ArcNode * firstarc;              //指向第一条依附
}VNode, AdjList[MVNum];              //AdjList表示邻接
typedef struct{
    AdjList vertices;                //邻接表
    int vexnum, arcnum;              //图的当前顶点数
}ALGraph;
```



AOV网络及其邻接表表示



AOV网络及其邻接表表示

7.2.2 图的存储结构-邻接表

- **邻接表：通过把顶点的所有邻接点组织成一个单链表来描述边**

```
#define MVNum 100                    //最大顶点数
typedef struct ArcNode{              //边结点
    int adjvex;                      //该边所指向的顶点的位置
    struct ArcNode * nextarc;        //指向下一条边的指针
    EdgeData cost;                   //改造结点，typedef int EdgeData;
}ArcNode;
typedef struct VNode{
    VerTexType data;                 //顶点信息
    ArcNode * firstarc;              //指向第一条依附该顶点的边的指针
}VNode, AdjList[MVNum];              //AdjList表示邻接表类型
typedef struct{
    AdjList vertices;                //邻接表
    int vexnum, arcnum;              //图的当前顶点数和边数
}ALGraph;
```

AOV网络及其邻接表表示

- 在邻接表中增设一个数组count[]，记录各顶点入度。入度为零的顶点即无前驱顶点。
- 在输入数据前，顶点表VexList[]和入度数组count[]全部初始化。在输入数据时，每输入一条边<j,k>，就需要建立一个边结点，并将它链入相应边链表中，统计入度信息：

```
ArcNode *p = new ArcNode;
p->adjvex = k;                    //建立边结点
p->nextarc = G. AdjList[j]. firstarc;
G. AdjList[j]. firstarc = p;      //链入顶点 j 的边链表的前端
count[k]++;                       //顶点 k 入度加一
```

拓扑排序算法

- 在算法中，使用一个存放入度为零的顶点的链式栈，供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下：

a) 建立入度为零的顶点栈;
b) 当入度为零的顶点栈不空时，重复执行
   - 从顶点栈中退出一个顶点，并输出之;
   - 从AOV网络中删去这个顶点和它发出的边，边的终顶点入度减一;
   - 如果边的终顶点入度减至0，则该顶点进入度为零的顶点栈;
- 如果输出顶点个数少于AOV网络的顶点个数，则报告网络中存在有向环。

拓扑排序算法


```
1.  void TopologicalSort(AdjGraph& G) {
2.      int i, j, k;   stack S;   InitStack(S);
3.      //入度为零的顶点栈初始化
4.      for (i = 0; i < G.n; i++)
5.          if (count[i] == 0)  Push(S, i);
6.          //入度为零顶点进栈
7.      for (i = 0; i < n; i++)                    //期望输出 n 个顶点
8.          if (StackEmpty(S)) {                   //中途栈空,转出
9.              cout << "网络中有回路!\n"; return;
10.         }
11.         else {                                 //继续拓扑排序
```

拓扑排序算法

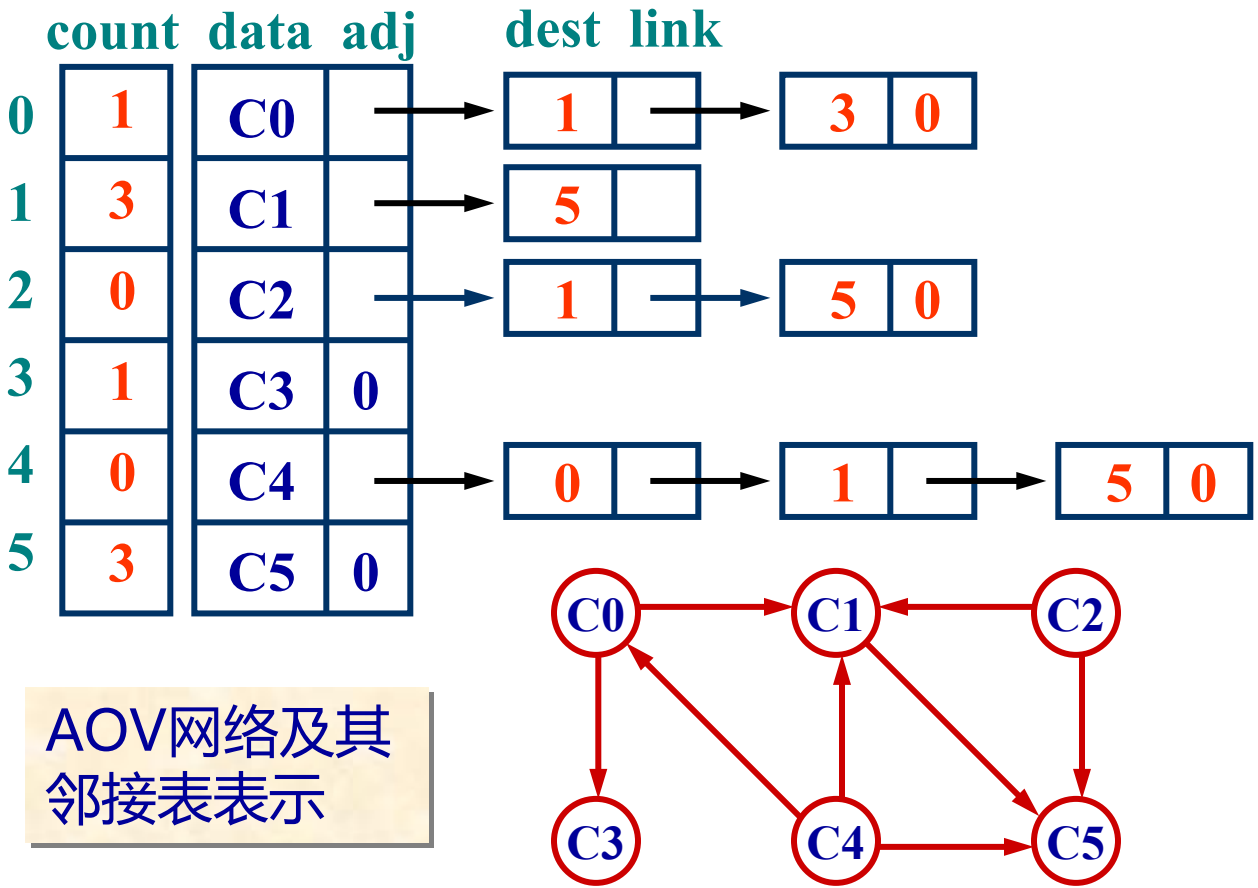

```
1.          Pop(S, j);                                  //退栈
2.          cout << j << endl;                          //输出拓扑序列
3.           ArcNode * p = G. AdjList[j]. firstarc;
4.          while (p != NULL) {                         //扫描出边表
5.              k = p-> nextarc;                        //另一顶点
6.              if (--count[k] == 0)                    //顶点入度减一
7.                  Push(S, k);                         //入度减至零, 进栈
8.              p = p->nextarc;
9.          }
10.     }
11.}
```

2.5.1 单链表-定义

- 链表定义
  - 结点-数据元素的存储映像，包括数据域和指针域（其信息成为指针或链）
  - 头指针-链表存储的开始
  - 空（NULL）-最后一个结点的指针
  - 单链表由表头唯一确定，因此单链表可以用头指针的名字来命名。

data：数据域，存放结点的值。
next：指针域，存放结点的直接后继的地址。

如何改造count数组，同时实现栈、入度计算两个功能?



AOV网络及其邻接表表示

AOE网络

关键路径与AOE网络

AOE——结点表示事件，边代表活动

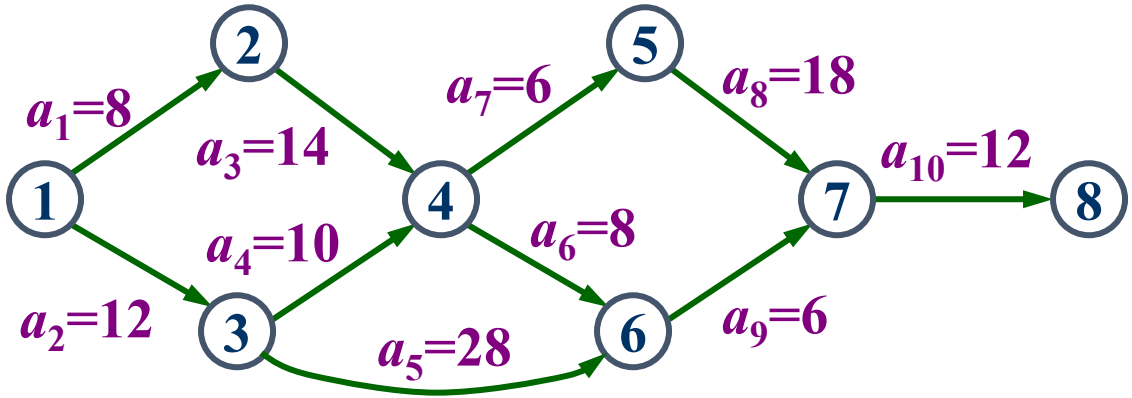
- 如果在无有向环的带权有向图中，用有向边表示一个工程中的**活动**(Activity)，用边上权值表示活动**持续时间** (Duration)，用顶点表示**事件**(Event)，则这样的有向图叫做用边表示活动的网络，简称 **AOE (Activity On Edges) 网络。**
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
  - 完成整个工程至少需要多少时间（假设网络中没有环）？
  - 为缩短完成工程所需的时间，应当加快哪些活动?

关键路径与AOE网络

- 从开始点到各个顶点，以至从开始点到结束点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。
- 因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做**关键路径**(**Critical Path**)。
- 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。

关键路径与AOE网络

- 关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。例如，下图就是一个AOE网。



- 顶点 1 是整个工程的开始点，顶点 8 是整个工程的结束点。a_i = …是各边上的活动及持续时间。

几个与计算关键活动有关的量

1. 事件V_i的最早可能开始时间Ve(i)

   它是从开始点V_1到顶点V_i的最长路径长度。

2. 事件V_i的最迟允许开始时间Vl[i]

   它是在保证结束点V_n在Ve[n]时刻完成的前提下，事件V_i的允许的最迟开始时间。

3. 活动a_k的最早可能开始时间 e[k]

   设活动a_k在边<V_i，V_j>上，则e[k]是从开始点V_1到顶点V_i的最长路径长度。因此

   $$e[k] = Ve[i]。$$

4. 活动a_k的最迟允许开始时间 l[k]

   它是在不会引起时间延误的前提下，该活动允许的最迟开始时间。

   l[k] = Vl[j] - dur(<i, j>)。

   其中，dur(<i, j>)是完成 a_k 所需的时间。

5. 时间余量 l[k] - e[k]

   表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。l[k] == e[k] 表示活动 a_k 是没有时间余量的关键活动。

- 为找出关键活动，需要求各个活动的 e[k] 与 l[k]，以判别是否 l[k] == e[k]。

关键路径与AOE网络

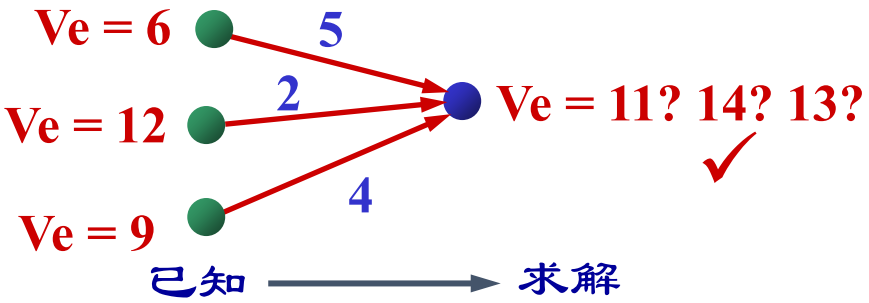
- 为求得e[k]与l[k]，需要先求得从开始点V1 到各个顶点Vi 的Ve[i] 和Vl[i]。
- 求Ve[i]的递推公式

a) 从Ve[1] = 0开始，向前递推

$$Ve[i] = \max_{j}\{Ve[j] + dur(<V_j, V_i>)\},$$

$<V_j, V_i> \in S2$，　i = 2, 3, …, n

S2 是所有指向V_i的有向边$< V_j , V_i >$的集合。

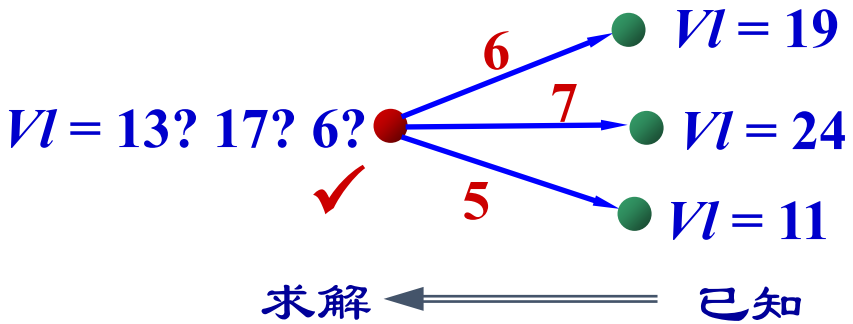


b) 从Vl[n] = Ve[n]开始，反向递推

$<V_i, V_j> \in S1, i = n-1, n-2, \ldots, 1$

$$Vl[i] = \min_j \{Vl[j] - dur(<V_i, V_j>)\},$$

S1是所有源自V_i的有向边$< V_i , V_j >$的集合。



- 这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。

关键路径与AOE网络

- 设活动a_k (k = 1, 2, ..., e) 在带权有向边<V_i, V_j> 上，其持续时间用dur (<V_i , V_j >) 表示，则有

  e[k] = Ve[i];

  l[k] = Vl[j] - dur(<V_i , V_j >)； k = 1, 2, ..., e。

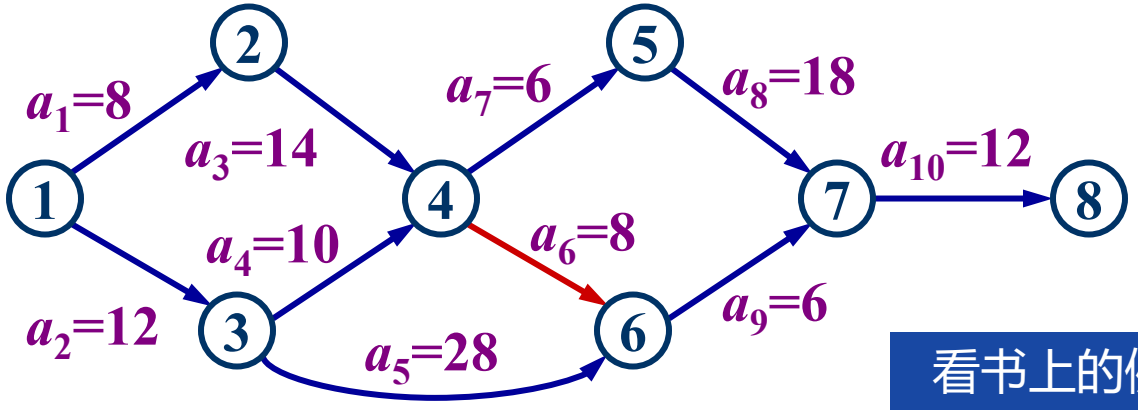
这样就得到计算关键路径的算法。

- 为了简化算法，假定在求关键路径之前已经对各顶点实现了拓扑排序，并按拓扑有序的顺序对各顶点重新进行了编号。

关键路径与AOE网络

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	0	8	12	22	28	40	46	58

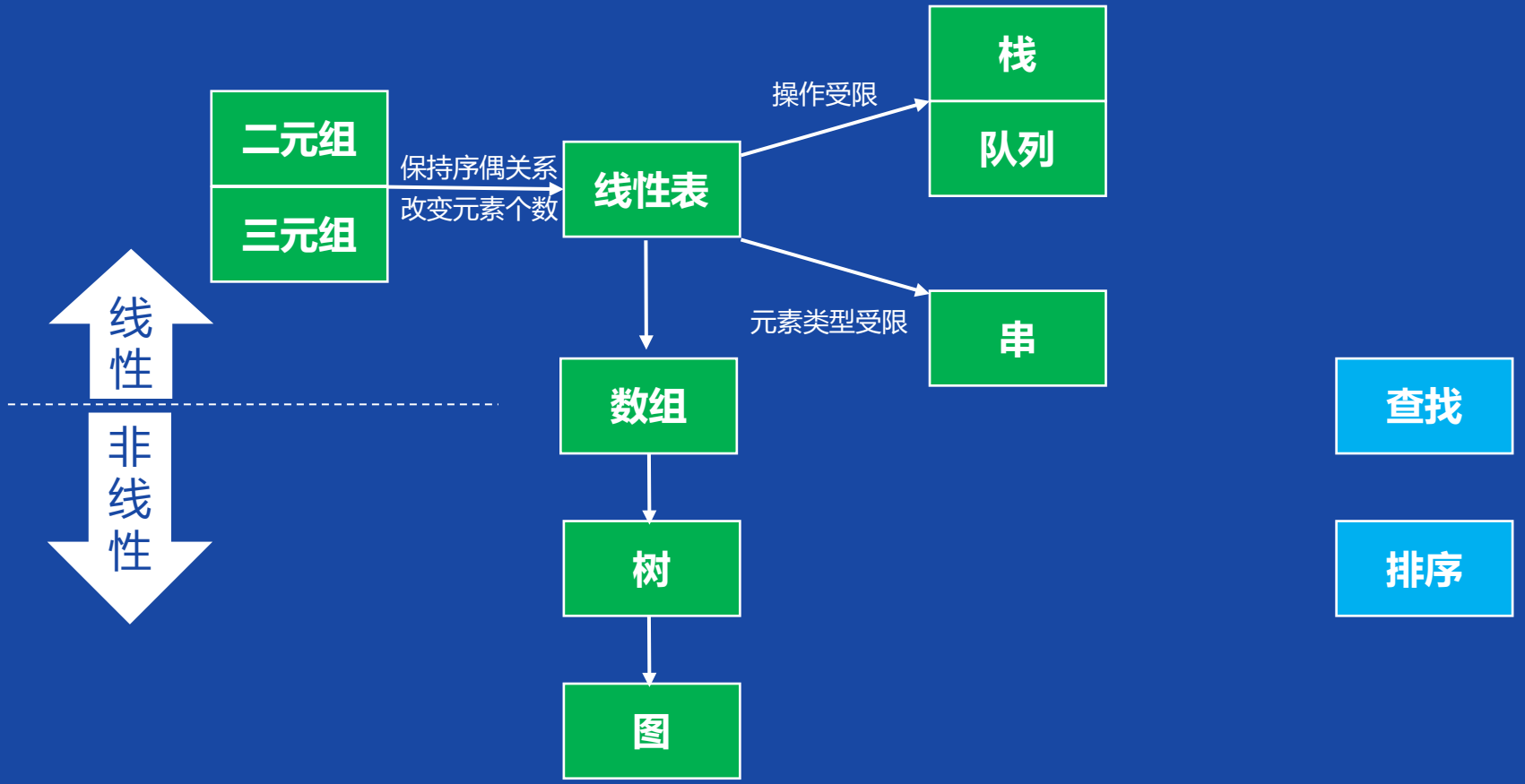
	1	2	3	4	5	6	7	8	9	10
e	0	0	8	12	12	22	22	28	40	46
l	0	0	8	12	12	32	22	28	40	46



看书上的例子更清晰

注意

- 所有顶点按拓扑有序的次序编号
- 仅计算 Ve[i] 和 Vl[i] 是不够的，还须计算 e[k] 和 l[k]。
- 不是任一关键活动加速一定能使整个工程提前。
- 想使整个工程提前，要考虑各个关键路径上所有关键活动。



- 程序 = 数据结构 + 算法

本节内容

第七章 查找

- 7.1 查找的基本概念
- 7.2 线性表的查找
- 7.3 树表的查找
- 7.4 散列表的查找

重点：掌握顺序查找、折半查找、二叉排序树上查找以及散列表上查找的基本思想和算法实现。

难点：查找的效率分析。

本节内容

- **基本概念**
- **线性表的查找**
  - 顺序表查找（监视哨、递归）、有序顺序表的折半查找、索引和分块查找（选修）
- **树表的查找（二叉排序树部分已学，其余选修）**
  - 平衡化
- **查找效率**

  平均查找长度

  (ASL_{succ}/ASL_{unsucc})
- **散列表（哈希表）的查找**

一些定义-查找表

- **查找表(Search Table)**
  - 定义：由同一类型的数据元素(或记录)构成的集合。
    “集合”中的数据元素之间存在着完全松散的关系->查找表是一种非常灵便的数据结构
  - 操作
    - 查询某个“特定的”数据元素是否在查找表中；
    - 检索某个“特定的”数据元素的各种属性；
    - 在查找表中插入一个数据元素；
    - 从查找表中删去某个数据元素。

一些定义-关键字

- **关键字**
  - 定义：是数据元素（或记录）中某个数据项的值，用以标识（识别）一个数据元素（或记录）。
  - 若此关键字可以唯一地标识一个记录，则称之为**主关键字**；
  - 若此关键字能识别若干记录，则称之为**次关键字**。

一些定义-查找

- **查找**
  - 定义：根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素。
  - 若查找表中存在这样一个记录，则称**查找**是**成功**的，此时查找结果为给出整个记录的信息，或指示该记录在查找表中的位置；
  - 否则称**查找不成功**，此时查找结果可给出一个**空记录**或**空指针**。

一些定义-查找表的分类

- **查找表的分类**
  - 静态查找表：仅作查询和检索操作的查找表。
  - 动态查找表：在查询过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素。

一些定义-查找表的分类

- **性能评价：平均查找长度**
  - 关键字的平均比较次数，也称平均搜索长度ASL(Average Search Length)

一些定义-类型定义


```
typedef struct {
        KeyType key; //关键字域
        ..... //其它域
} ElemType;
```

根据具体的关键字类型, 定义用于比较的、带参的宏

```
#define EQ(a, b) ...
#define LT(a, b) ...
#define LQ(a, b) ...
```

本节内容

- 线性表的查找——顺序查找

静态查找表-顺序查找

- 查找表结构：以顺序表或线性链表表示
- 基本思想：从一端开始向另一端，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功；反之，若直至另一端，其关键字和给定值比较都不等，则表明表中没有所查记录，查找不成功。
- 查找成功示例：

  (34, 44, 43, 12, 53, 55,73, 64, 77)，key = 64

  查找不成功示例：

  (34, 44, 43, 12, 53, 55,73, 64, 77)，key = 88

静态查找表-顺序查找

- 对顺序表的顺序查找（注意和书上数据结构不一样，但是和线性表一样）

```
typedef struct{
    ElemType *elem;          //数据元素存储空间基址，0号单元留空
    int length;              //表长度
}SSTable;
```

2.4 基本操作实现（查找 LocateElem）

- 查找（P22. 算法2.3 按值查找）
  - 求表长L.length
  - 取出第i个元素L.elem[i-1](0<i<L.length+1)
  - 按值查找

```
1.  int LocateElem( SqList L, ElemType e) {
2.          //在顺序表中查找值为e的数据元素，返回其序号
3.          for(int i=0; i < L.length; i++)
4.                  if(L.elem[i]==e)return i+1;          //查找成功，返回序号
5.          return 0;                                    //查找失败，返回0
6.  }
```

基本操作实现（查找 LocateElem）

- 查找（P22. 算法2.3 按值查找 改造）
  - 求表长L.length
  - 取出第i个元素L.elem[i-1](0<i<L.length+1)
  - 按值查找

```
1.  int LocateElem( SqList L, ElemType e) {
2.          //在顺序表中查找值为e的数据元素，返回其序号
3.          for(int i=1; i <= L.length; i++)
4.                  if(L.elem[i]==e)return i;          //查找成功，返回序号
5.          return 0;                                  //查找失败，返回0
6.  }
```

基本操作实现（查找 LocateElem）

- 查找（P22. 算法2.3 按值查找 再改造）
  - 求表长L.length
  - 取出第i个元素L.elem[i-1](0<i<L.length+1)
  - 按值查找

```
1.  int LocateElem( SqList L, ElemType e) {
2.          //在顺序表中查找值为e的数据元素，返回其序号
3.          for(int i=L.length; i >=1; --i)
4.                  if(L.elem[i]==e)return i;          //查找成功，返回序号
5.          return 0;                                  //查找失败，返回0
6.  }
```

基本操作实现（查找 LocateElem）

- 查找（P22. 算法2.3 按值查找 设监视哨）
  - 求表长L.length
  - 取出第i个元素L.elem[i-1](0<i<L.length+1)
  - 按值查找

```
1.  int LocateElem( SqList L, ElemType e) {
2.          L.elem[0]=e;
3.          for(int i=L.length; L.elem[i]!=e; --i);
4.          return i;
5.  }
```

静态查找表-顺序查找

- 对顺序表的顺序查找

```
typedef struct {
    KeyType key;            //关键字域
    InfoType otherinfo;     //其他域
) ElemType;

typedef struct{
    ElemType *elem;         //数据元素存储空间基址，0号单元不留白
    int length;             //表长度
}SSTable;
```

静态查找表-顺序查找

- 对顺序表的顺序查找

```
1.  int Search_Seq(SSTable ST, KeyType key) {
2.      // 在顺序表ST中顺序查找其关键字等于key的数据元素。
3.      // 若找到则函数值为该元素在表中的位置否则为0
4.      ST.elem[0].key = key; //  “哨兵”
5.      // 从后往前找
6.      for (i=ST.length; ST.elem[i].key!=key; --i);
7.      return i; //找不到时，i为0
8.  } // Search_Seq
```

哨兵的作用：免去查找过程中每一步都要检测整个表是否查找完毕。

静态查找表-顺序查找，监视哨在高下标处

- 对顺序表的顺序查找

```
1.  int Search_Seq(SSTable ST, KeyType key) {
2.      // 在顺序表ST中顺序查找其关键字等于key的数据元素。
3.      // 若找到则函数值为该元素在表中的位置否则为0
4.      ST.elem[ST.length].key = key;            //  “哨兵”
5.      // 从前往后找
6.      for (int i=0; ST.elem[i].key!=key; i++);
7.      return i; //找不到时，i为ST.length
8.  } // Search_Seq
```

哨兵的作用：免去查找过程中每一步都要检测整个表是否查找完毕。

9.1.1 静态查找表-顺序查找

- 性能分析
  - 平均查找长度（ASL）：为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值。
  - 查找成功时

$$ASL = nP_1 + (n-1)P_2 + \ldots + 2P_{n-1} + P_n, \quad n = ST.length$$

假定$P_i = 1/n$

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

  - 查找不成功时

$$ASL = n + 1$$

顺序查找的递归算法

顺序查找可以用递归方法实现。当查找表中第一个元素即为所求，查找成功；否则对除第一个元素外的后续表元素构成的查找表使用相同方法递归查找。当后续查找表为空，则查找失败。

```
1.  int SeqSearch (SSTable L, KeyType x, int loc) {
2.  //在数据表L.elem[1]..L.elem[n]中查找其关键字值
3.  //与给定值x匹配的元素, 函数返回其表中位置。
4.  //参数 loc 是在表中开始查找位置
5.        if (loc >L.length) return 0;
6.                                          //查找失败
7.        else if (L.elem[loc].key == x) return loc;
8.                                          //查找成功
9.        else return SeqSearch(L, x, loc+1);
10.                                         //递归查找
11.}
```

本节内容

- 有序表的查找

基于有序顺序表的顺序查找算法


```
1. int SeqSearch (SSTable L, KeyType x) {
2. //在数据表L.elem[1]..L.elem[n] 中顺序查找关键字值为 x 的数据元素
3.     for (int i = 1; i <= L.length; i++)
4.         if (L.elem[i].key == x) return i;        //成功成功
5.         else if (L.elem[i].key > x) break;       //查找失败
6.     return 0;                                    //顺序查找失败, 返回失败信息
7. }
```

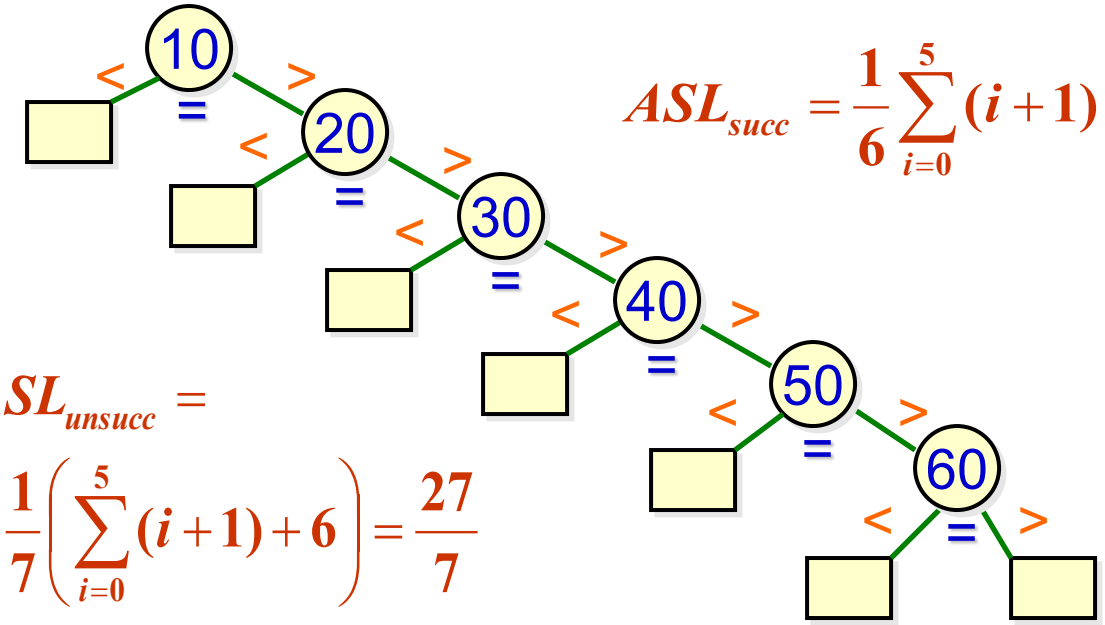
基于有序顺序表的顺序查找

- 有序顺序表限定表中的元素按照其关键字值从小到大或从大到小依次排列。在这类表中进行查找比表中元素任意存放，查找速度会有所提高。
- 在有序顺序表中做顺序查找时，若查找不成功，不必检测到表尾才停止，只要发现有比它的关键字值大的即可停止查找。
- 若设表中有 n 个元素，则查找成功的平均查找长度依旧，而查找不成功的平均查找长度为：

$$ASL_{unsucc} = \frac{1}{n+1}\left(n + \sum_{i=0}^{n-1}(i+1)\right) = \frac{1}{n+1}\left(n + \frac{n(n+1)}{2}\right)$$

- 有序顺序表的顺序查找示例及分析其查找效率的判定树：

(10, 20, 30, 40, 50, 60)



$$ASL_{succ} = \frac{1}{6}\sum_{i=0}^{5}(i+1) = \frac{7}{2}$$

$$ASL_{unsucc} = \frac{1}{7}\left(\sum_{i=0}^{5}(i+1)+6\right) = \frac{27}{7}$$

9.1.2 有序表的查找-折半查找

- 折半查找（二分查找）
  - 查找表结构：有序表
  - 基本思想：查找区间逐步缩小（折半）

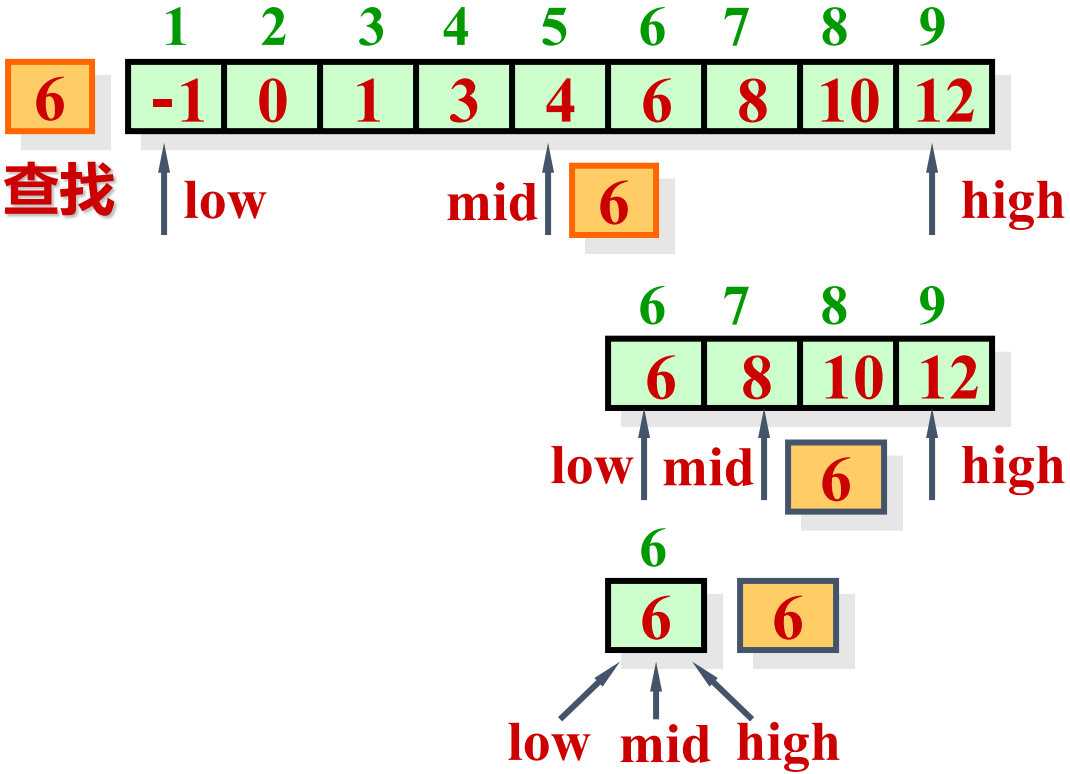
做折半查找时，先求位于查找区间正中的元素的下标mid，用其关键字值与给定值x比较:

A.elem[mid].key == x, 查找成功；

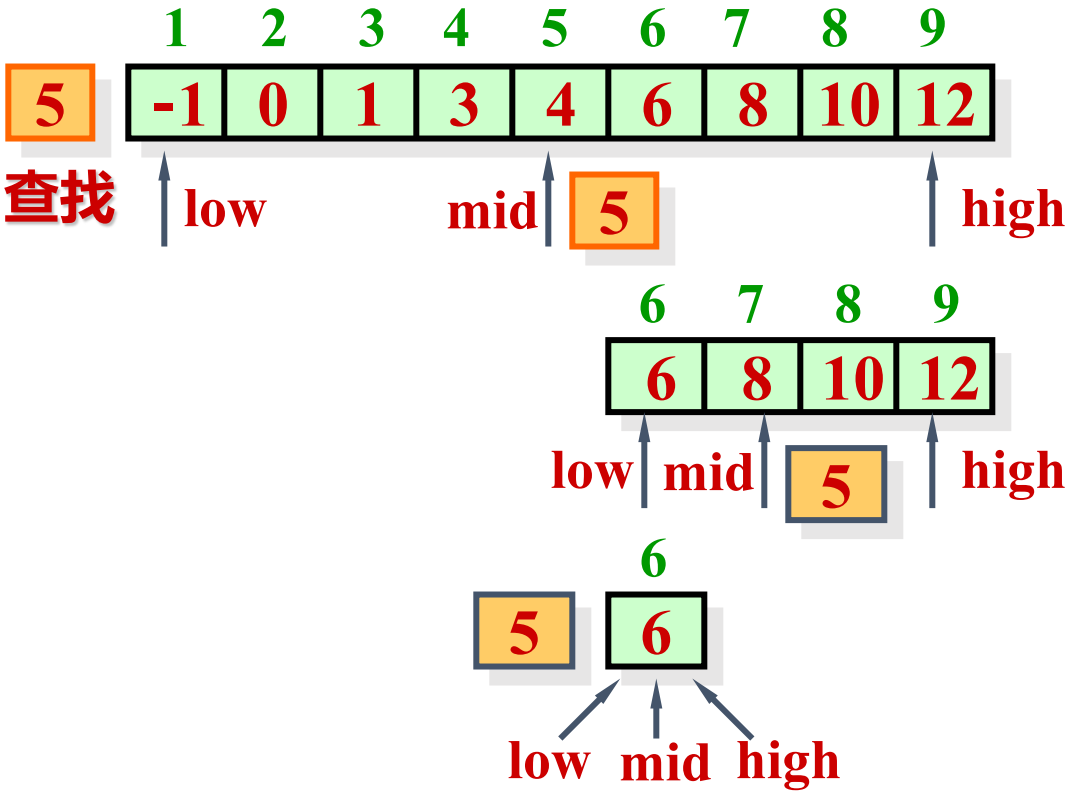
A.elem[mid].key > x, 把查找区间缩小到 表的前半部分，继续折半查找；

A.elem[mid].key < x，把查找区间缩小到表的后半部分，继续折半查找。

如果查找区间已缩小到一个元素，仍未找到想要查找的元素，则查找失败。


1 2 3 4 5 6 7 8 9
6
-1 0 1 3 4 6 8 10 12
查找
low
mid
6
high
6 7 8 9
6 8 10 12
low
mid
6
high
6
6
6
low mid high

查找成功的例子


1 2 3 4 5 6 7 8 9
5
-1 0 1 3 4 6 8 10 12
查找
low
mid
5
high
6 7 8 9
6 8 10 12
low
mid
5
high
6
5
6
low mid high

查找失败的例子

9.1.2 有序表的查找-折半查找

```
1.  int Search_Bin ( SSTable ST, KeyType key ) {
2.      low = 1; high = ST.length;                    // 置区间初值
3.      while (low <= high) {
4.          mid = (low + high) / 2;
5.          if (key == ST.elem[mid].key)
6.              return mid;                           // 找到待查元素
7.          else if ( key < ST.elem[mid].key )
8.              high = mid - 1;                       // 继续在前半区间进行查找
9.          else low = mid + 1;                       // 继续在后半区间进行查找
10.     }
11.     return 0;                                     // 顺序表中不存在待查元素
12.} // Search_Bin
```

递归的折半查找算法——递归

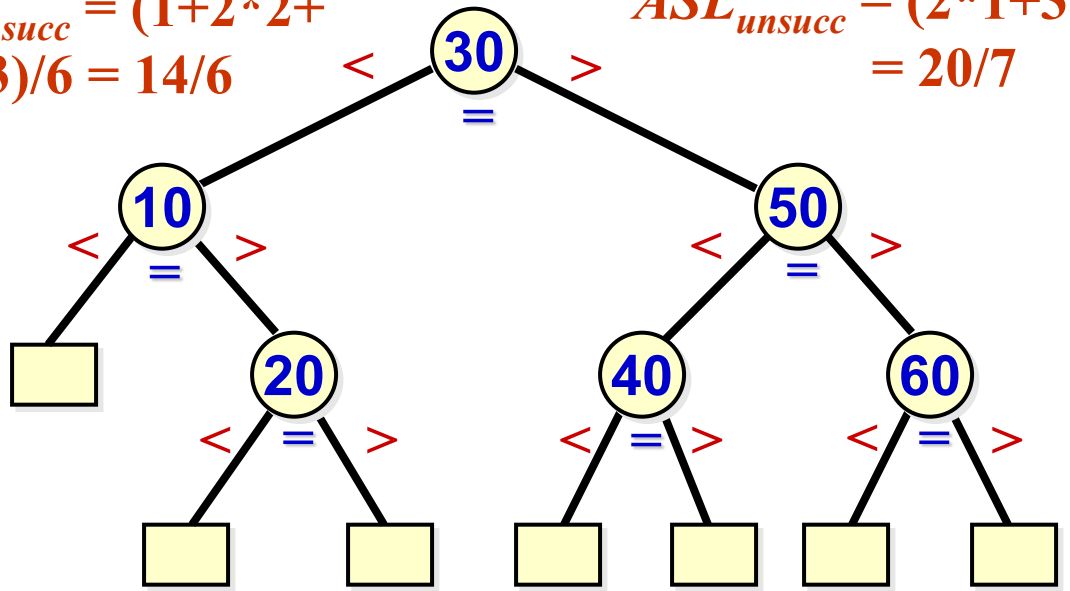

```
1.  int BinSearch(SSTable L, KeyType x, int low, int high) {
2.      int mid = 0;
3.      if (low <= high) {
4.          mid = (low + high) / 2;
5.          if (L.elem[mid].key < x)
6.              mid = BinSearch(L, x, mid +1, high);
7.          else if (L.elem[mid].key > x )
8.              mid = BinSearch(L, x, low, mid-1);
9.      }
10.     return mid;
11. }
12. //调用方式 BinSearch(L, x, 1, L.length);
```

有序顺序表的折半查找的判定树

(10, 20, 30, 40, 50, 60)

ASL_{succ} = (1+2*2+3*3)/6 = 14/6

ASL_{unsucc} = (2*1+3*6)/7 = 20/7



折半查找性能分析

- 若设 $n = 2^h-1$，则描述折半查找的判定树是高度为 h 的满二叉树。
- $2^h = n+1,\ h = \log_2(n+1)$
- 第 1 层结点有1个，查找第 1 层结点要比较 1 次; 第 2 层结点有2个，查找第 2 层结点要比较 2 次；…, 第 i ($1 \le i \le h$) 层结点有 2^{i-1} 个, 查找第 i 层结点要比较 i 次，….
- 假定每个结点的查找概率相等，即 $p_i = 1/n$，则查找成功的平均查找长度为

$$ASL_{succ}=\sum_{i=1}^{n}p_i\cdot c_i=\frac{1}{n}\sum_{i=1}^{n}c_i=\frac{1}{n}(1*2^0+2*2^1+$$

$$+3*2^2+\cdots+(h-1)*2^{h-2}+h*2^{h-1})$$

可以用归纳法证明：

$$1\times 1+2\times 2^1+3\times 2^2+\cdots+(h-1)\times 2^{h-2}+h\times 2^{h-1}=$$

$$=(h-1)\times 2^h+1$$

这样

$$ASL_{succ}=\frac{1}{n}((h-1)\times 2^h+1)=\frac{1}{n}((n+1)\log_2(n+1)-n)$$

$$=\frac{n+1}{n}\log_2(n+1)-1\approx\log_2(n+1)-1$$

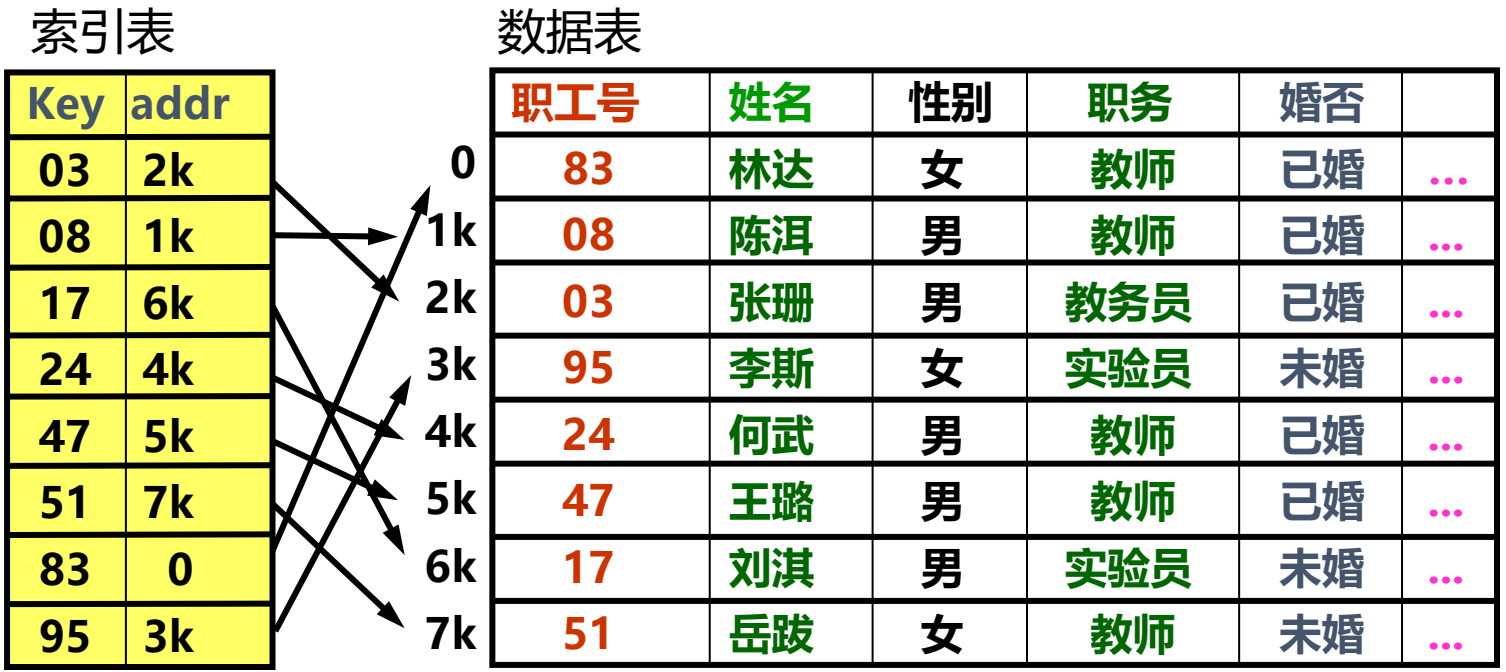
总结

- 查找效率与数据集有关！也与判定树的形态有关！
- 有序表的顺序查找、折半查找
- 其他查找：
  - 斐波那契查找 （通过改变切割点改变树的形态，平均性能比折半好，最差情况比折半差；分割只要进行加减计算）
  - 插值查找（如果表中数据均匀）

本节内容

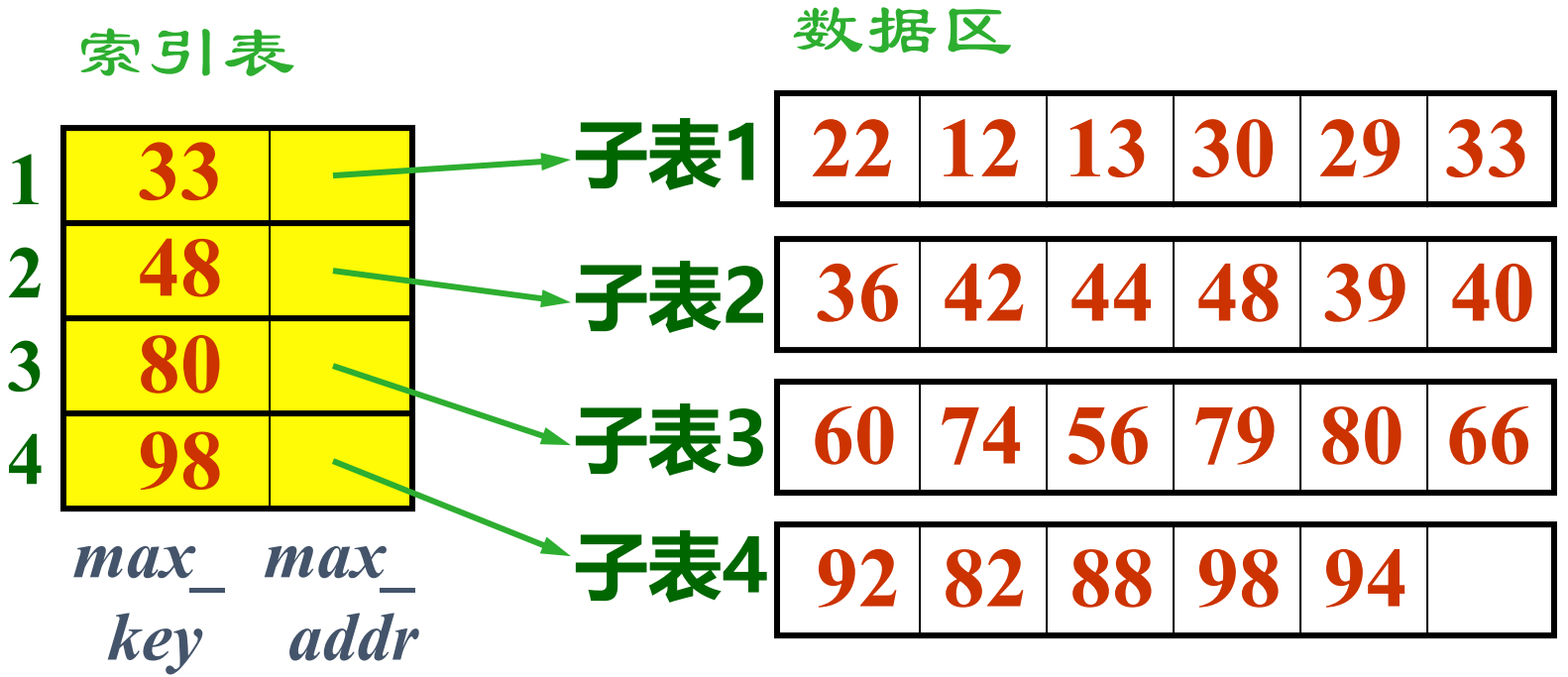
- 索引(*)

索引


索引表
Key
addr
03
2k
08
1k
17
6k
24
4k
47
5k
51
7k
83
0
95
3k
数据表
0
1k
2k
3k
4k
5k
6k
7k
职工号
姓名
性别
职务
婚否
83
林达
女
教师
已婚
...
08
陈洱
男
教师
已婚
...
03
张珊
男
教务员
已婚
...
95
李斯
女
实验员
未婚
...
24
何武
男
教师
已婚
...
47
王璐
男
教师
已婚
...
17
刘淇
男
实验员
未婚
...
51
岳跋
女
教师
未婚
...

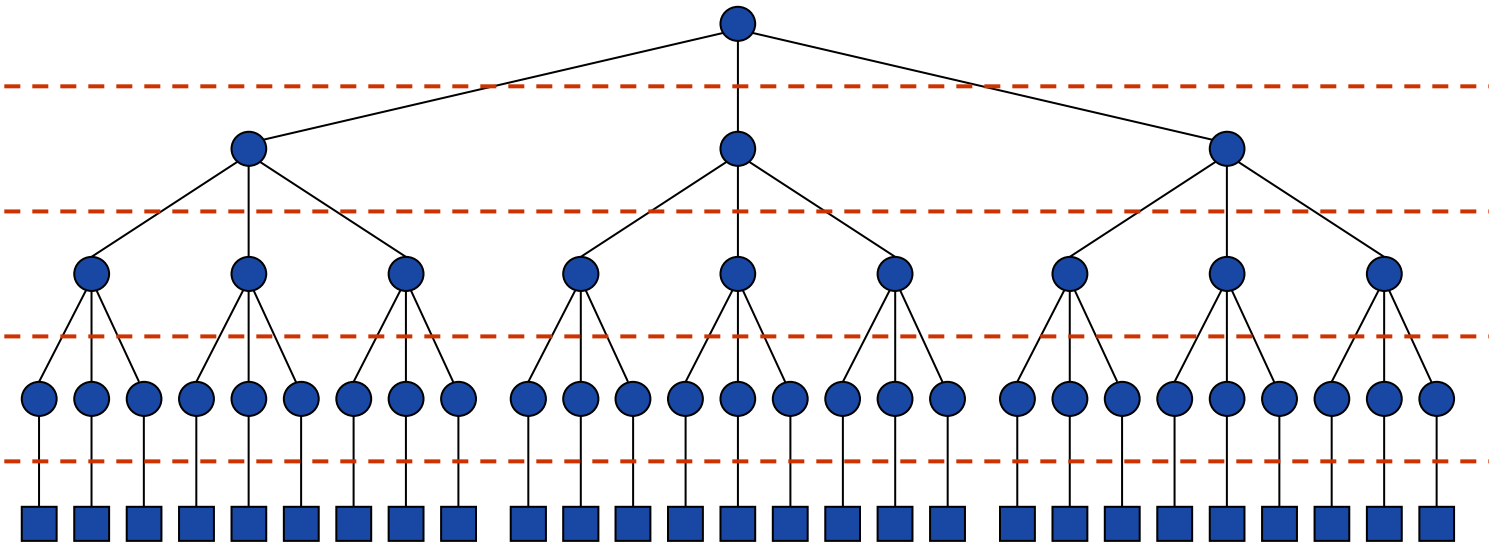
本节内容

- 分块查找（索引顺序查找）



- 分块查找/索引顺序查找 P225（分块有序，块内无序；相当于做一次有序查找一次无序表顺序查找）

索引



多级索引结构形成 m 路查找树

本节内容

- **树表查找**
  - 动态查找：二叉排序树及平衡化
  - B树(*)

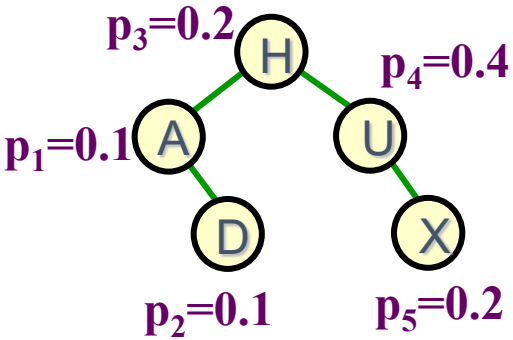
总结

- 到目前为止，平均查找长度计算基于一个前提：“等概率”！
- 然而，讲哈夫曼树的时候，电报的例子告诉我们，关键字的查找概率往往不是等概率的……

一个例子

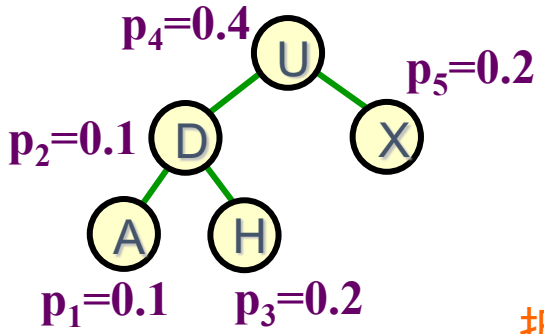
- 设含有 5 个关键字的有序表 (A, D, H, U, X)，其查找概率分别为

A	D	H	U	X
$p_1 = 0.1$	$p_2 = 0.1$	$p_3 = 0.2$	$p_4 = 0.4$	$p_5 = 0.2$



折半查找的判定树

ASL_{succ}= 0.2*1+(0.1+0.4)*2
+(0.1+0.2)*3 = 2.1



另一查找树

ASL_{succ}= 0.4*1+(0.1+0.2)*2
+(0.1+0.2)*3 = 1.9

折半查找并非最优

最/次优查找树

- 设有序顺序表中关键字为 $k_1, k_2, \ldots, k_n$, 它们的查找概率分别为 $p_1, p_2, \ldots, p_n$, 构成查找树后，它们在树中的层次为 $l_1, l_2, \ldots, l_n$。

- 基于该查找树的查找算法的平均查找长度为

$$ASL_{succ} = \sum_{i=0}^{n-1} p_i \cdot l_i$$

- 使得 ASL_{succ} 达到最小的静态查找树为静态最优二叉排序树。然而，求最优查找树的算法效率较低，达$O(n^3)$，可求次优查找树，其时间代价减低为$O(n\log_2 n)$。构造方法见（旧版）教材 P223

本节内容

- 散列表(哈希表)——重要!

散列 (Hashing)

- 散列方法在表项存储位置与其关键字之间建立一个确定的对应函数关系Hash( )，使每个关键字与结构中一个唯一存储位置相对应：

  Address = Hash (Rec.key)

- 在查找时，先对表项的关键字进行函数计算,把函数值当做表项的存储位置，在结构中按此位置取表项比较。若关键字相等，则查找成功。在存放表项时，依相同函数计算存储位置，并按此位置存放。这种方法就是散列方法。
- 在散列方法中使用的转换函数叫做散列函数。按此方法构造出来的表或结构就叫做散列表。

- 使用散列方法进行查找不必进行多次关键字的比较，查找速度比较快，可以直接到达或逼近具有此关键字的表项的实际存放地址。
- 散列函数是一个压缩映象函数。关键字集合比散列表地址集合大得多。因此有可能经过散列函数的计算，把不同的关键字映射到同一个散列地址上，这就产生了冲突（碰撞）。
- 示例：有一组表项，其关键字分别是

  12361，07251，03309，30976

  采用的散列函数是 hash(x) = x % 73 + 13420，其中，“%”是除法取余操作。

则有：hash(12361) = hash(07250) = hash(03309) = hash(30976) = 13444。

- 就是说，对不同的关键字，通过散列函数的计算，得到了同一散列地址。我们称这些产生冲突的散列地址相同的不同关键字为同义词。
- 由于关键字集合比地址集合大得多，冲突很难避免。所以对于散列方法，需要讨论以下两个问题：
  - 对于给定的一个关键字集合，选择一个计算简单且地址分布比较均匀的散列函数， 避免或尽量减少冲突；
  - 拟订解决冲突的方案。

散列函数

- **构造散列函数时的几点要求：**
  - 散列函数应是简单的，能在较短的时间内计算出结果。
  - 散列函数的定义域必须包括需要存储的全部关键字，如果散列表允许有 m 个地址时，其值域必须在 0 到 m-1 之间。
  - 散列函数计算出来的地址应能均匀分布在整个地址空间中 。
- **下面讨论几个常用的散列函数。**

直接定址法

- **此类函数取关键字的某个线性函数值作为散列地址：**

  Hash(key) = a*key+b　{ a, b为常数 }

- **这类散列函数是一对一的映射，一般不会产生冲突。但是，它要求散列地址空间的大小与关键字集合的大小相同。**

直接定址法

- 示例：有一组关键字如下：

{ 942148，941269, 940527，941630，941805，941558，942047, 940001 }。散列函数为

Hash (key) = key – 940000

Hash(942148) = 942148 - 940000 = 2148

Hash(941269) = 941269 - 940000 = 1269

Hash(940527) = 940527 - 940000 = 527

Hash(941630) = 941630 - 940000 = 1630

Hash(941805) = 941805 - 940000 = 1805

Hash(941558) = 941558 - 940000 = 1558

Hash(940047) = 942047 - 940000 = 2047

Hash(940001) = 940001 - 940000 = 1

可以按计算出的地址存放记录。

数字分析法

- **设有 n 个 d 位数，每一位可能有 r 种不同的符号，它们在各位上出现的频率不一定相同。可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。**
- **示例：**
  - 有一组关键字如下：{ 942148，941269, 940527，941630，941805, 941558，942047, 940001 }。
  - 先把所有关键字按个位对齐，从高位到低位对每一位加以编号。假设散列表空间大小是H[400]，有 3 位，比较各位数字看哪 3 位数字分布均匀。

数字分析法

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

- 可取各关键字的 ④⑤⑥ 位做为记录的散列地址。也可以把第①，②，③和第⑤位相加，舍去进位位，变成一位数，与第④，⑥位合起来作为散列地址。

数字分析法

- 这样选择的地址范围是0～999，但实际的地址范围是0～399，因此还需对得到的结果乘一个压缩因子0.4，把地址压缩到0～399之内。
- 数字分析法仅适用于事先明确知道表中所有关键字每一位数值的分布情况，它完全依赖于关键字集合。如果换一个关键字集合，选择哪几位要重新决定。

除留余数法

- **设散列表中允许地址数为 m，取一个不大于 m，但最接近于或等于 m 的质数 p 作为除数，利用以下函数把关键字转换成散列地址：**

$$hash(key) = key \% p \qquad p \le m$$

其中，“%”是整数除法取余的运算，要求这时的质数 p 不是接近2的幂。

- **示例：**
  - 关键字 key = 962148, 散列表大小 m = 25, 即 HT[25]。取质数 p= 23。散列函数 hash (key) = key % p。则散列地址为

    hash (962148) = 962148 % 23 = 12。
  - 注意除数 p 的取法，一定要求得到的地址在允许的地址范围之内，所以不能取大于 m 的数字，也不能取偶数。

处理冲突的开地址方法

- 因为任一种散列函数也不能避免产生冲突，因此选择好的解决冲突的方法十分重要。
- 若设散列表中的编址为 0 到 m-1，当要加入一个表项 R2时，用它的关键字 R2.key，通过散列函数 hash(R2.key) 的计算，得到它的存放地址号 j。
- 但在存放时发现此地址已被另一个表项 R1 占据，发生了冲突。为此，需把 R2 存放到表中“下一个”空位中。 如果表未被装满，则在允许的范围内必定还有空位。

(1) 线性探查再散列 (Linear Probing)

- **需要查找或加入一个表项时, 使用散列函数计算地址：$H_0 = hash(key)$；一旦发生冲突，在表中顺次向后寻找“下一个”空位 H_i 的公式为：**

$$H_i = (H_{i-1} + 1) \% m，\ i = 1, 2, \ldots, m-1$$

- **示例：**
  - 设给出一组表项，它们的关键字为 Burke, Ekers, Broad, Blum, Attlee, Hecht, Alton, Ederly。采用的散列函数是：取其第一个字母在字母表中的位置再除以2。

    Hash (x) = ⌊(ord (x) - ord (‘A’) +1)/2⌋

    //ord ( ) 是求字符内码的函数

可得 Hash(Burke) = 1　　Hash(Ekers) = 2

Hash(Broad) = 1　　Hash(Blum) = 1

Hash(Attlee) = 0　　Hash(Hecht) = 4

Hash(Alton) = 0　　Hash(Ederly) = 2

设散列表 HT[0..16]，m = 17。采用线性探查再散列处理冲突，则散列结果如图所示。

0	1	2	3	4	5	6
Attlee	Burke	Ekers	Broad	Blum	Hecht	Alton
(1)	(1)	(1)	(3)	(4)	(2)	(7)

7	8	9	10	11	12	13
Ederly						
(6)						

- 寻找“下一个”空位 H_i 的公式还可以改写为：

$$H_i = (H_0 + i) \% m，\ i = 1, 2, \ldots, m-1$$

即用以下的线性探查序列在表中寻找“下一 个”空位的地址：

$$H_0 +1, H_0 +2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$$

- 当发生冲突时，探查下一个地址。当循环 m-1 次后就会回到开始探查时的位置，说明待查关键字不在表内，而且表已满，不能再插入新关键字。

性能分析

- **用平均查找长度*ASL* 衡量散列方法的性能。**
  - 查找成功的平均查找长度 ASL_{succ} 是指查找到表中已有表项的平均探查次数。它是找到表中各个已有表项的探查次数的平均值。
  - 查找不成功的平均查找长度 ASL_{unsucc} 是指在表中查找不到待查的表项，但找到插入位置的平均探查次数。它是表中所有可能散列到的位置上要插入新元素时为找到空位的探查次数的平均值。

    (上例中通过散列函数可计算出的地址为14)
  - 在使用线性探查法对示例进行查找时，查找成功的平均查找长度 ASL_{succ}为：

$$ASL_{succ} = \frac{1}{8}\sum_{i=1}^{8} C_i = \frac{1}{8}(1+1+1+3+4+2+7+6) = \frac{25}{8}.$$

- 查找不成功的平均查找长度ASLunsucc为：

$$ASL_{unsucc} = \frac{9+8+7+6+5+4+3+2+1*6}{14} = \frac{50}{14}.$$

- 散列表存放的表项不应有重复的关键字。在插入新表项时，如果发现表中已经有关键字相同的表项，则不再插入。
- 线性探查再散列方法容易产生“堆积（聚集）”，即不同探查序列互相交织，导致某一探查序列的存储位置被其他探查序列的关键字占据。这样，为寻找某一关键字需要经历不同的探查序列，导致查找时间增加。

- 在开地址情形下不能真正删除表中已有表项。删除表项会影响其他表项的查找。例如，若把关键字值为Broad的表项真正删除，以后在查找关键字值为Blum和Alton的表项时就查不下去，会错误地判断表中没有关键字值为Blum和Alton的表项。
- 若想删除一个表项，只能给它做一个删除标记，进行逻辑删除，不能把它真正删去。
- 逻辑删除的副作用是：在执行多次删除后，表面上看起来散列表很满，实际上有许多位置没有利用。

(2) 二次探查再散列 (quadratic probing)

- 首先通过某一个散列函数对表项的关键码 key 进行计算，得到散列地址，它是一个非负整数。

$$H_0 = hash(key)$$

- 然后使用二次探查再散列在表中寻找“下一个”空位，其公式为：

$$H_i = (H_0+i^2) \% m, \quad H_i = (H_0-i^2) \% m,$$

$$i = 1, 2, 3, \ldots, (m-1)/2$$

- m 是表的大小，它应是一个值为 4k+3 的质数，其中 k 是一个整数。这样的质数如3, 7, 11, 19, 23, 31, 43, 59, 127, 251, 503, ….

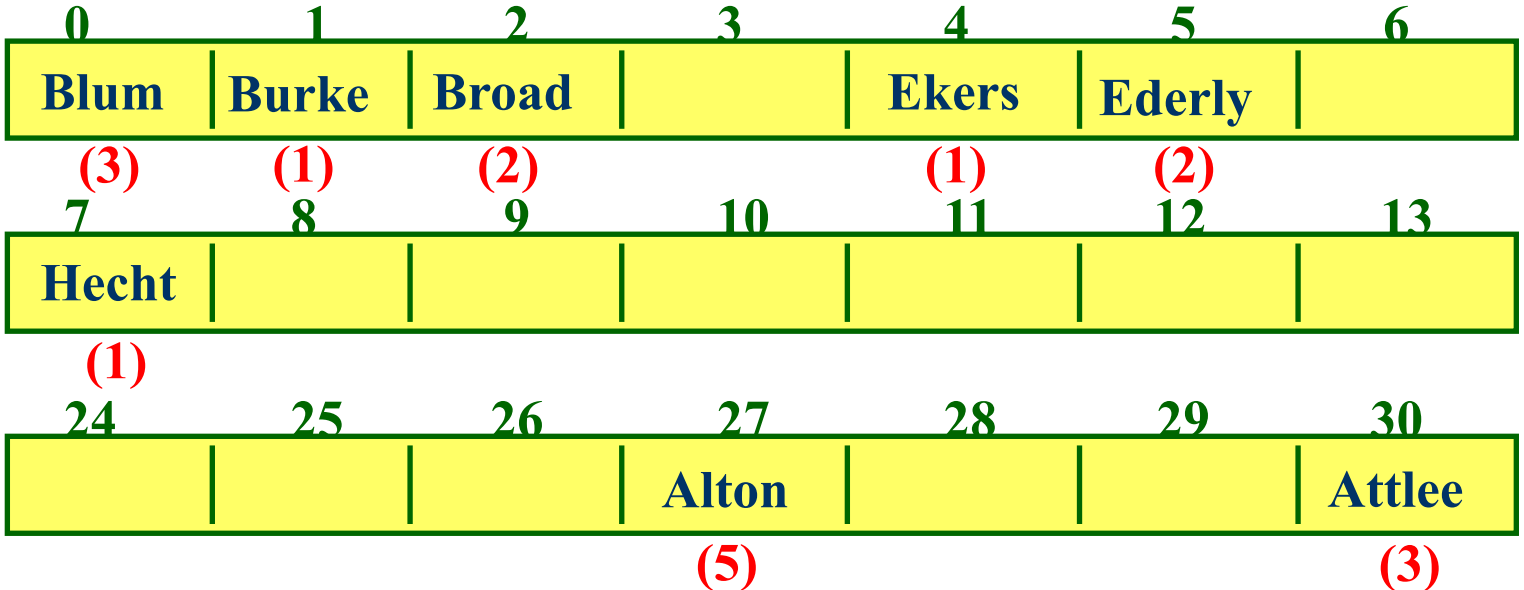
- 二次探查再散列的探查序列形如 H_0, H_0+1, H_0-1, H_0+4, H_0-4, …。
- 在做 (H_0-i^2) % m 的运算时，当 H_0-i^2 < 0 时，可补充一个修正语句：

  if ((j = (H_0-i^2)% m) < 0 = j += m
- 示例:给出一组关键字{Burke, Ekers, Broad, Blum, Attlee, Hecht, Alton, Ederly }。散列函数为：

  Hash (x) = ord (x) - ord ('A')
- 可能计算出的散列地址是0～25，取满足4k+3的质数，表的长度为 m = 31。
- 计算出的散列地址为：

Hash (Burke) = 1
Hash (Broad) = 1
Hash (Attlee) = 0
Hash (Alton) = 0

Hash (Ekers) = 4
Hash (Blum) = 1
Hash (Hecht) = 7
Hash (Ederly) = 4



性能分析

- 使用二次探查再散列处理冲突时的查找成功的平均查找长度ASL_{succ}为：

$$ASL_{succ}=\frac{1}{8}\sum_{i=1}^{8}C_i=\frac{1}{8}(3+1+2+1+2+1+5+3)=\frac{18}{8}.$$

- 查找不成功的平均查找长度ASL_{unsucc}为：

$$ASL_{unsucc}=\frac{1}{26}(6+5+2+3+2+2+20)=\frac{40}{26}$$

- 计算ASL_{unsucc}时除数取 26 的原因是散列函数可计算出的散列地址范围为0～25，总共 26 个位置，与散列表大小 m 无关。

- 可以证明，当表的长度 m 为质数且表的装载因子 α（表明表的装满程度）不超过 0.5 时，新的表项一定能够插入，而且任何一个位置不会被探查两次。
- 在查找时可以不考虑表满情况；但在插入时必须确保表的装填因子α不超过0.5。如果超出必须将表长度扩充一倍，进行表的分裂。
- 在删除一个表项时，为确保探查链不致中断，也只能做表项的逻辑删除，即给被删表项加一个逻辑删除的标记 Deleted。

(3) 双散列法

- **使用双散列法解决冲突时，需要两个散列函数。**
  - 第一个散列函数 Hash( ) 按表项关键字 key计算表项所在的地址号 H_0 = Hash(key)。
  - 一旦冲突，用第二个散列函数 ReHash( ) 计算该表项到达“下一个”地址的间隔。
- **要求ReHash(key) 的取值与key的值有关，它的取值应是小于地址空间大小m且与m互质的正整数。**
- **设表的长度为m，在表中寻找“下一个”地址的迭代公式为：**

$$H_0 = Hash(key),$$

$$H_i = (H_{i-1} + ReHash(key)) \% m.$$

- **如果写成通项公式为**

$$H_i = (H_0 + i * ReHash(key)) \% m,$$

$$i = 1, 2, \ldots, m-1$$

- **示例：给出一组表项关键字{ 22, 41, 53, 46, 30, 13, 01, 67 }。散列函数为：**

$$H_0(x) = (3x) \% 11。$$

- **散列表为 HT[0..10]，m = 11。因此再散列函数为**

$$ReHash(x) = (7x) \% 10 + 1。$$

$$H_i = (H_{i-1} + (7x) \% 10 + 1) \% 11,\ i = 1, 2, \ldots$$

$H_0(22) = 0$　　$H_0(41) = 2$　　$H_0(53) = 5$

$H_0(46) = 6$　　$H_0(30) = 2$ 冲突 $H_1 = (2+1) = 3$

$H_0(13) = 6$ 冲突 $H_1 = (6+2) = 8$

$H_0(01) = 3$ 冲突 $H_1 = (3+8) = 0$ 冲突

$H_2 = (0+8) = 8$ 冲突 $H_3 = (8+8) = 5$ 冲突

$H_4 = (5+8) = 2$ 冲突 $H_5 = (2+8) = 10$

$H_0(67) = 3$ 冲突 $H_1 = (3+10) = 2$ 冲突

$H_2 = (2+10) = 1$

0	1	2	3	4	5	6	7	8	9	10
22	67	41	30		53	46		13		01
(1)	(3)	(1)	(2)		(1)	(1)		(2)		(6)

- 查找成功的平均查找长度

$$ASL_{succ} = \frac{1}{8}(1+3+1+2+1+1+2+6) = \frac{17}{8}$$

0	1	2	3	4	5	6	7	8	9	10
22	67	41	30		53	46		13		01
(1)	(3)	(1)	(2)		(1)	(1)		(2)		(6)

- 查找不成功的平均查找长度
  - 每一散列位置的跨步间隔有10种：1～10。先计算每一散列位置各种跨步间隔下找到下一个空位的比较次数，求出平均值；

- 再计算各个位置的平均比较次数的总平均值。

- Rehash()的取法很多。例如, 当 m 是质数时, 可定义
  - ReHash(key) = key % (m-2) +1
  - ReHash(key) = ⌊key / m⌋ % (m-2)+1
- 当 m 是 2 的方幂时，ReHash(key) 可取从 0 到 m-1 中的任意一个奇数。

处理冲突的链地址法

- **链地址方法首先对关键字集合用某一个散列函数计算它们的散列地址。**
- **通常把各地址相同的表项通过一个单链表链接起来，称之为同义词子表，各链表的表头结点组成一个向量。**
- **向量的元素个数与散列地址数一致。地址为 i 的同义词子表的表头结点是向量中第 i 个元素。**
- **示例：**
  - 给出一组表项关键字 { Burke, Ekers, Broad, Blum, Attlee, Alton, Hecht, Ederly }。散列函数为：

$$Hash(x) = \lfloor (ord(x) - ord('A') + 1) / 2 \rfloor$$

用它计算可得：

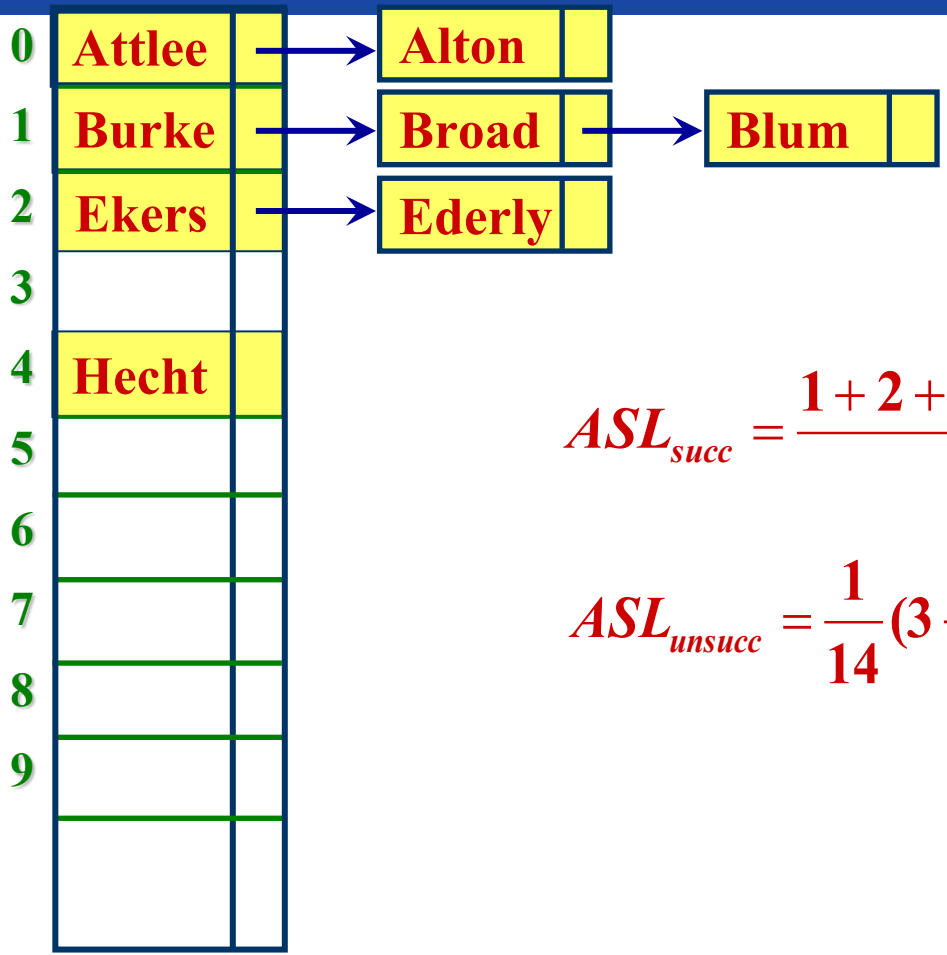
Hash(Burke) = 1　　Hash(Ekers) = 2

Hash(Broad) = 1　　Hash(Blum) = 1

Hash(Attlee) = 0　　Hash(Hecht) = 4

Hash(Alton) = 0　　Hash(Ederly) = 2

散列表为 HT[0..13]，m = 14。



$$ASL_{succ} = \frac{1+2+1+2+3+1+2+1}{8} = \frac{13}{8}$$

$$ASL_{unsucc} = \frac{1}{14}(3+4+3+1+2+1*9) = \frac{22}{14}$$

性能分析

- 通常，每个地址中的同义词子表都很短，查找速度快得多。
- 应用链地址法处理冲突，需要增设链接指针，似乎增加了存储开销。但由于开地址法必须保持大量的空闲空间以确保查找效率，而表项所占空间又比指针大得多，所以使用链地址法反而比开地址法节省存储空间。

散列法分析

- 散列表是一种直接计算记录存放地址的方法，它在关键字与存储位置之间直接建立了映象。
- 当选择的散列函数能够得到均匀的地址分布时，在查找过程中可以不做多次探查。
- 由于很难避免冲突，增加了查找时间。冲突的出现，与散列函数的选取 (地址分布是否均匀)，处理冲突的方法 (是否产生堆积) 有关。
- 实验结果表明，链地址法优于开地址法；在散列函数中，用除留余数法作散列函数优于其它类型的散列函数。

- Knuth对不同的冲突处理方法进行了分析。若设 α 是散列表的装填因子：

$$\alpha = \frac{\text{表中已装有记录数}}{\text{表中预设的最大记录数}} = \frac{n}{m}$$

- S_n 是查找一个随机选择的关键字 x_i $(1 \le i \le n)$ 所需的关键字比较次数的期望值
- U_n 是在长度为 m 的散列表中 n 个地址已装入表项的情况下，装入第 n+1 项所需执行的关键字比较次数期望值。
- 前者称为在 $\alpha = n / m$ 时的查找成功的平均查找长度，后者称为在$\alpha = n / m$ 时的查找不成功的平均查找长度。

- 用不同的方法处理冲突时散列表的平均查找长度如图所示。

处理冲突的方法		平均查找长度 ASL	
		查找成功 Sn	查找不成功 Un（登入新记录）
开地址法	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
开地址法	二次探查法 双散列法	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法（同义词子表法）		$1+\frac{\alpha}{2}$	α

- 散列表的装填因子α表明了表中的装满程度。越大，说明表越满，再插入新元素时发生冲突的可能性就越大。
- 散列表的查找性能，即平均查找长度依赖于散列表的装填因子，不直接依赖于n或m。
- 当装填因子α较高时，选择的散列函数不同，散列表的查找性能差别很大。在一般情况下多选用除留余数法。
- 不论表的长度有多大，我们总能选择一个合适的装填因子，以把平均查找长度限制在一定范围内。