· **北大信息学院《数据结构与算法****A》期末考试**

1. 考试时间和地点

考试时间：第1周 周日(2025.01.05) 上午 8:30 –10:30，

考试地点: 理教303

2. 考试题型

填空或者选择、简答与辨析、算法填空和设计分析与证明

注意：

（1）数据结构/算法设计与分析题只要写明基本思想、无歧义即可，必要时加上足够的注释。

（2）对于算法中直接使用的类和函数（例如栈、队列的函数），应该先写ADT，并简单说明算法中用到的重要函数的功能、入口参数、出口参数。

3. 考试范围和重点

7-12章，以本文最后的内容为复习重点，尤其是★标出部分为重中之重。

考试时如果涉及到本大纲没有列出的内容，那么试卷中会给出足够的定义和性质。

4. 考场安排和注意事项

1. 没有正式选课的旁听同学，请不要来考场。
2. 请随身带好您的学生证(或其他可以证明身份的证件)，笔和涂改工具参加考试。
3. 考试形式为闭卷，可以使用计算器。

4）考前10分钟，请大家把书包、课本、讲义、作业本、自带的草稿纸等放在教室前面的讲台和窗台上，只需要留下学生证(或其他可以证明身份的证件)，笔和涂改工具。教室清理干净后，可以提前5分钟发放试卷(带有草稿纸和答题纸，可以撕下来)，从前排向后排发放。注意在试卷纸和有效答题纸上写上姓名和学号，并且一定要在试卷纸的诚实答题宣言旁边的姓名和学号栏签名（否则，试卷计零分）。

5. 我们统一发草稿纸，不够可以随时举手要。
6. 请大家注意考场纪律，不要交头接耳，私下讨论。考试时对试题有疑问，可以举手，待监考老师来到旁边时，再请向监考老师询问。

7）考试时间为120分钟，中间不休息。提前15分钟提醒大家整理试卷，注意写好姓名和学号。

8. 监考老师宣布“考试时间到”以后，请大家停笔（不停笔的同学，监考老师有权没收试卷并宣布作废），把草稿纸和答题纸放在试卷上面，使姓名和学号朝外（诚实答题宣言旁边的姓名和学号朝向最外面），对折以后放在桌面上。监考老师收卷清点无误，并宣布“全班同学都可以离开了”以后方可集体离开。注意，不要把试卷题带出考场，否则将计零分。
9. 提前交卷的同学，把试卷交到讲台上，并收拾好自己的东西，迅速离开考场。

5. 答疑安排

随时联系助教或者老师进行答疑！

复习大纲

从第7章图考到第12章高级数据结构。各章节以下面的内容为复习重点。尤其是绿颜色文字或★标出部分为重中之重。期中考过的内容，期末不直接考察，但可能在内容上有所涉及。

第**7章** **图**

一. 概念

1. 图的深度周游
2. 图的宽度周游
3. 图的生成树、生成树林、最小生成树

二. 方法及算法 ★

1. 图的存储方法★ 相邻矩阵和邻接表
2. 图的周游 （1） 深度优先 （2） 宽度优先
3. 图的生成树与最小生成树

² 从某一点出发，按深度优先或宽度优先周游的生成树

² 最小生成树 ① Prim算法 ② Kruskal算法(避圈法)

4. ★ 拓扑排序 : 对于给定图，找出若干个或所有拓扑序列。任何有向无环图，都可以拓扑排序。
5. ★ 最短路径算法：Dijkstra算法、Floyd算法(属于动态规划法) ★
6. ★ 最小生成树：Prim算法、Kruskal算法都是典型的贪心法（退化的动态规划法）

★**第8章** **内排序**

1. 重点排序算法：直接插入法、★Shell排序、★快速排序、★基数排序、归并排序
2. 算法分析

1）基于比较次数和移位次数分析最好、最坏的时间、空间：直接插入法、二分法插入排序、起泡排序、直接选择、快速排序、基数排序、归并排序

2） 记住各种排序方法的平均时间

3. 各种排序方法的局部修改和混合应用

第9章** 文件管理和外排序**

二. 方法及算法

1. ★ 置换选择排序
2. ★ 多路归并 (败者树，最佳归并树，多路归并的读盘和写盘次数)

第10章** 检索**

一. 概念

1. 平均检索长度
2. 二分法检索

★3. 散列表、同义词、碰撞、堆积

二. 方法

1. 二分法检索的判定树、查找某个结点的比较次数
2. 散列表: 1) 散列函数的选择 (除余法、平方取中法、折叠法)

2) 冲突处理方法 (分离同义词子表、线性探测、双散列函数)

★ 三. 散列算法（查找、插入、删除，对墓碑的处理）

第11章** 索引技术**

一. 概念

1. 顺序文件 2. 散列文件 3. 倒排文件 4. 静态索引结构 5.动态索引结构(B树) 6. 红黑树

二. 方法（不考算法代码）

★1. B树、B+树的插入 (注意保持性质，特别是等高；以及子结点和关键码个数的上下限制)

★2. B树/B+树的读盘和写盘次数分析

3. B树/B+树的效率分析

B树中关键码没有重复，父结点中的关键码是其子结点的分界；B+中最底层是关键码的一个全集，往根的方向一层层复写。

B树插入 : 插入 ------- 分裂

B+树插入 : 插入 -------- 分裂

★4. 红黑树的插入方法

插入算法首先是采用BST的方法把结点插入到位，然后注意调整。尤其是“红红”冲突的解决，注意有换色、重构。

第12章** 高级数据结构**

一. 概念

1 . 多维数组和稀疏矩阵 2. 广义表 3. Trie树 4. Patricia 5. AVL树 6. 伸展树

二. 方法

1. 特殊矩阵和稀疏矩阵的计算，重点在于理清楚索引值的规律。

★ 2. 广义表的结构和周游

3. 字符树：Trie树和Patricia树 （只做了解）

4. 最佳二叉搜索树，需要理解平均检索长度最优的特点

★ 5. AVL平衡二叉树的插入方法：注意首先找到失衡结点，注意LL、LR、RL、RR的四种旋转调整。不考删除算法，但可能考相关性质

★6. 伸展树及其简单应用：伸展树在搜索过程中旋转调整结构，使访问最频繁的结点靠近树结构的根。伸展树的旋转分为：单旋转、一字形旋转和之字形旋转。注意伸展树的变种，例如半伸展树。

三. 算法

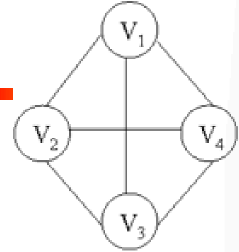
Splay树的插入及区间操作。

图

图的表示

- 点用V
- 边用E

图的集合表示

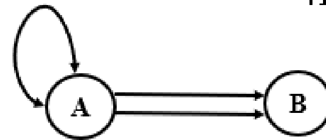


无向图G1

- 无向图G1的集合表示
  - G1=(V,E)
  - V(G1)={V_1，V_2，V_3，V_4}
  - E(G1)={(V_1,V_2),(V_1,V_3) ,(V_1,V_4) ,(V_2,V_3) ,(V_2,V_4) ,(V_3,V_4)} **//圆括弧**

有向图G2

- 有向图G2的集合表示
  - G2=(V,E)
  - V(G2)={V_1，V_2，V_3}
  - E(G2)={<V_1,V_2>,<V_2,V_1> ,<V_2,V_3>} **//尖括弧**
- 在本章后续图的表示中，请注意
  - **不考虑顶点到自身的边**
  - **不允许一条边在图中重复出现**



京大学信息科学技术学院 数据结构与算法 1

概念

- 邻接点：相邻顶点
  - 一条边所连接的两个顶点，成为`邻接点`
- 顶点的度
  - 入度
  - 出度
  - 终端节点（叶子）

$$e = \frac{1}{2}\sum_{i=1}^{n} d_i \quad d_i\text{是}V_i\text{的度数}$$

有根图

有根图

- 一个有向图中，若存在一个顶点V_0，从此顶点有路径可以到达图中其它所有顶点，则称此有向图为有根图，V_0称作图的根

连通图

- 对无向图G=（V，E）而言，如果从V_1到V_2有一条路径（从V_2到V_1也一定有一条路径），则称V_1和V_2是连通的
- 若图G中任意两个顶点都是连通的，则无向图G是连通的

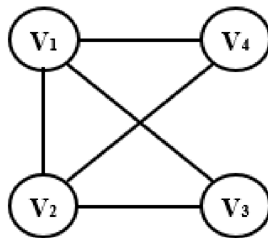
连通分量（连通分支）

- 指无向图的最大连通子图

图的相邻矩阵表示法

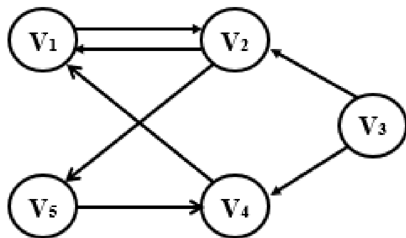
$$A[i,j]=\begin{cases}1, 若(V_i,V_j)或<V_i,V_j>是图G的边\\0, 若(V_i,V_j)或<V_i,V_j>不是图G的边\end{cases}$$

图示



G6

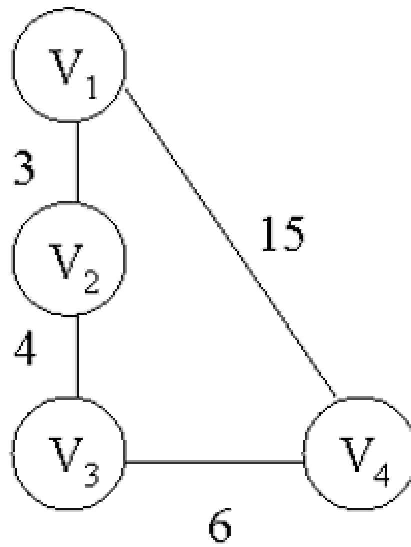
$$A_6=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}$$



G7

$$A_7=\begin{bmatrix}0&1&0&0&0\\1&0&0&0&1\\0&1&0&1&0\\1&0&0&0&0\\0&0&0&1&0\end{bmatrix}$$

加权矩阵



G4

$$A_4 = \begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$

图的邻接表表示法

邻接矩阵表示法的特点

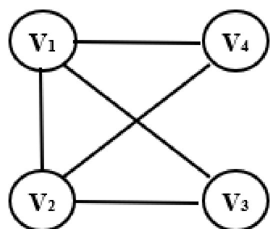
- 矩阵的规模只与顶点的个数n有关n^2
- 与边无关
- 由于大量的边不存在，造成空间浪费

邻接表表示法

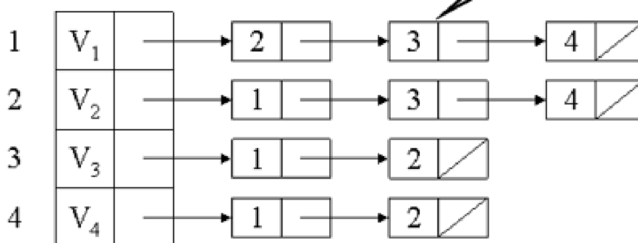
- 即与顶点有关，又与边有关
- 顶点表：对应n个顶点，包括顶点数据和指向边表的指针
- 边链表：对应m条边，包括顶点序号和指向边表下一表目指针

图示1

边链表



无向图G6



顶点表

无向图G6的邻接表表示

需要$|V|+2|E|$个存储单元

十字链表（不考）

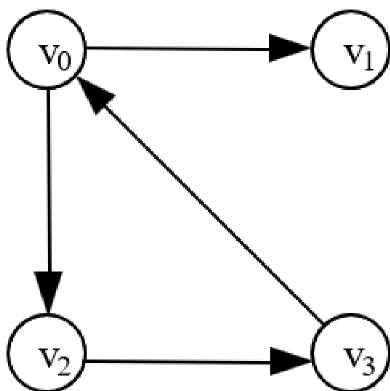
顶点表：对应图的顶点，三个域，分别是顶点的数据，(`firstinarc`)入边表的头指针和(`firstoutarc`)出边表的头指针

边链表：对应图的边，5个域，起点(`fromvex`)和终点(`tovex`)的顶点序号；边权值的info值；

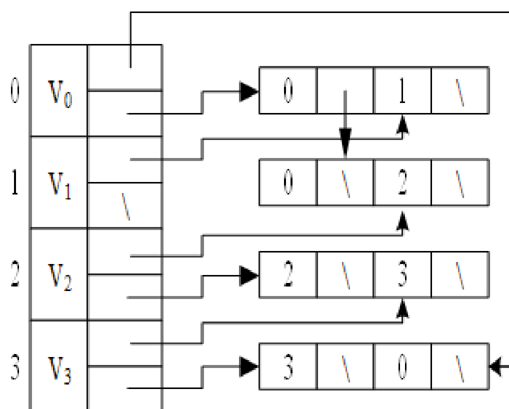
`fromnextarc`指针指向下一个以`fromvex`为起点的边

`tonextarc`指针指向下一个以`tovex`为终点的边

图示



有向图



顶点表

边表

图的周游

深度优先搜索(DFS)

- 从一个顶点出发，访问它的所有邻接点，再依次从这些邻接点出发，访问它们的邻接点，直到所有顶点都被访问过为止

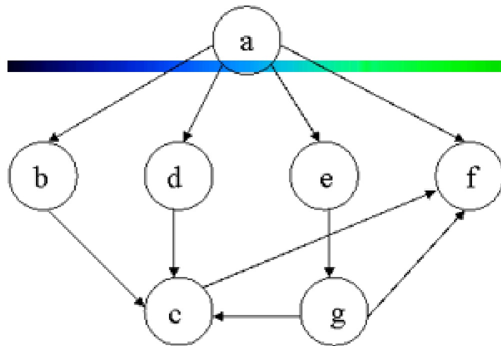
```
void DFS(Graph& G, int V)
{
    Visit(G, V);                    //访问V
    G.Mark[V]= VISITED;             //访问顶点V，并标记其标志位
    for(Edge e=G. FirstEdge(V); G.IsEdge(e); e=G. NextEdge(e))
        //递归地按照深度优先的方式访问V邻接的未被访问的顶点
        if(G.Mark[G. ToVertices(e)]== UNVISITED)
            DFS(G, G. ToVertices(e));
}
```

时间复杂度分析

- `dfs`对每一条边处理一次（无向图每条边从两个方向处理），每个顶点访问一次
- 邻接表表示法：有向图为$O(|V|+|E|)$，无向图为$O(|V|+2|E|)$
- 邻接矩阵表示法：有向图为$O(|V|^2)$，无向图为$O(|V|^2)$

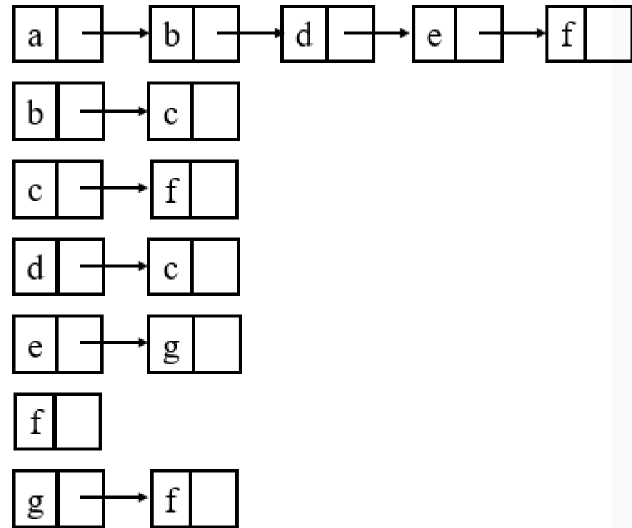
深度优先搜索树

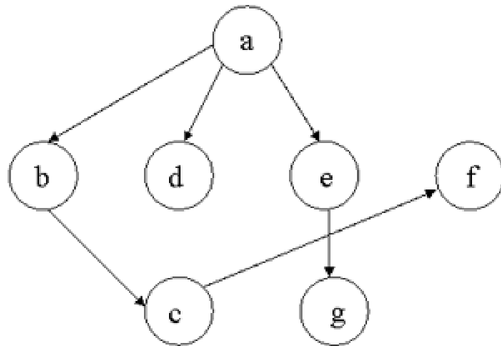
图示



(a)有向图

(b)邻接表



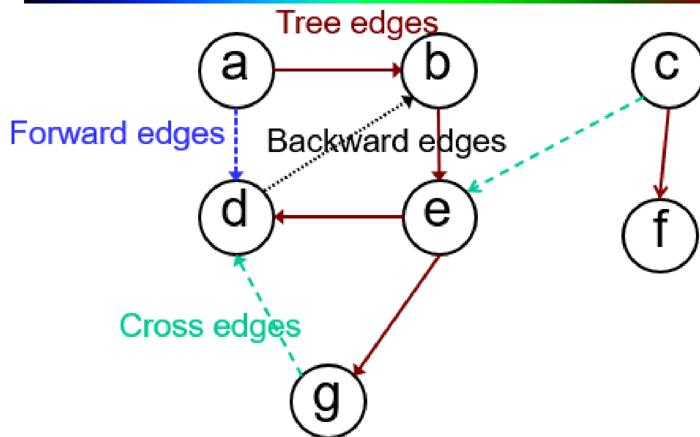


(c)深度优先搜索树

深度优先搜索的顺序：a, b, c, f, d, e, g

边的分类

基于DFS的边的分类

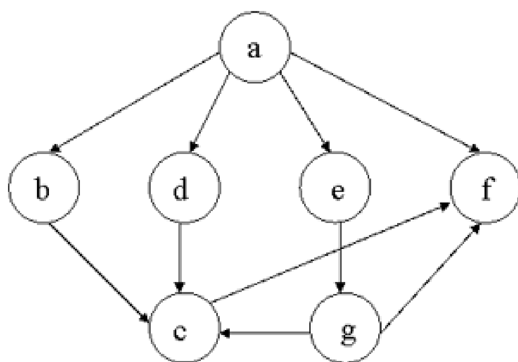


Forward Edges: 从DFS树的祖先到子孙。
Backward Edges：从DFS树的子孙到祖先。
Cross Edges: DFS树上两个不是“祖先-子孙”关系的两个点之间的边。

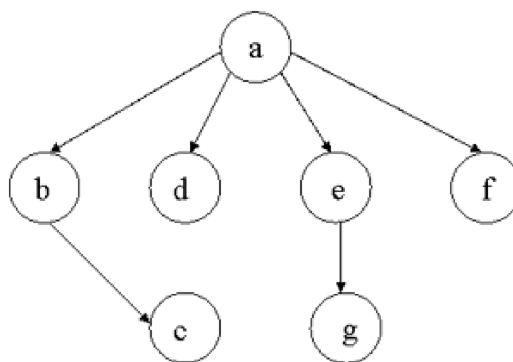
广度优先搜索(BFS)

- 基本思想
  - 访问顶点V_0
  - 然后访问V_0邻接到的所有未被访问过的邻居顶点V_{01}，V_{02}，…V_{0i}
  - 再依次访问V_{01}，V_{02}，…V_{0i}邻接到的所有未被访问的邻居顶点
  - 如此进行下去，直到访问遍所有的顶点。
- 广度优先搜索树（breadth-first search tree）

广度优先搜索树



(a)有向图



(b)广度优先搜索树

广度优先搜索的顺序：a, b, d, e, f, c, g

```
void BFS(Graph& G, int V) {
    using std::queue; // 初始化广度优先遍历要用到的队列
    queue<int> Q;
    G.Mark[V] = VISITED; // 访问顶点V，并标记其标志位，V入队
    Visit(G, V);
    Q.push(V);

    while (!Q.empty()) { // 如果队列仍然有元素
        int V = Q.front();
```

```
        Q.pop(); // 取顶部元素，并出队

        // 将与该点相邻的每一个未访问点都入队
        for (Edge e = G.FirstEdge(V); G.IsEdge(e); e = G.NextEdge(e)) {
            if (G.Mark[G.ToVertex(e)] == UNVISITED) {
                G.Mark[G.ToVertex(e)] = VISITED;
                Visit(G, G.ToVertex(e));
                Q.push(G.ToVertex(e)); // 入队
            }
        }
    }
}
```

拓扑排序（重点）

- 是指以某种线性顺序来组织多项任务，以便能在满足先决条件的情况下逐个完成各项任务
    - 有向无环图(DAG)进行拓扑排序
    - 从图中选择一个没有前驱的顶点，输出它
    - 从图中删除该顶点和所有以它为起点的边
    - 重复上述两步，直到所有顶点都已输出，或者图中不存在无前驱的顶点为止
- 若环路存在，则无法进行拓扑排序（最后仍然有顶点没有输出，但没有入度为0的顶点）

BFS-TopSort算法

```
void TopsortbyQueue(Graph& G) {                    //队列方式实现的拓扑排序
    for(int i=0;i<G.VerticesNum();i++)
        G.Mark[i]=UNVISITED;                       //初始化标记数组
    using std::queue;
    queue<int> Q;                                  //初始化队列
    for(i=0; i<G.VerticesNum(); i++) {             //图中入度为0的顶点入队
        if(G.Indegree[i]==0)
            Q.enqueue(i);
    }
    while(!Q.empty()){                             //如果队列中还有图的顶点
        V=Q.dequeue();                             //一个顶点出队
        Visit(G, V);                               //访问该顶点
        G.Mark[V]=VISITED;
```

```
        //边e的终点的入度值减1
        for(Edge e= G.FirstEdge(V); G.IsEdge(e);e=G.NextEdge(e)) {
            G.Indegree[G.ToVertex(e)]--;
            if(G.Indegree[G.ToVertex(e)]==0)
                Q.enqueue(G.ToVertex(e)); //入度为0的顶点入队
        }//end for
    }//end while
    for(i=0; i<G.VerticesNum(); i++) {
        if(G.Mark[i]==UNVISITED){
            Print("图有环");                //图有环
            break;
        }
    }
}   //注:在有环的情况下会提前退出, 从而可能没处理完所有的边和顶点
```

广度优先排序可以判定有环存在~~

复杂度

- 拓扑排序的时间复杂度
  - 采用相邻矩阵时，每次算法需要找所有入度为0的顶点，需要 Θ (|V|²)的时间，那么对|V|个顶点而言，总代价为Θ (|V|³)
  - 当采用邻接表时，因为在顶点表的每个顶点中可以有一个字段来存储入度，所以只需Θ(|V|)的时间，加上处理边、顶点的时间，总代价为Θ(2|V|+|E|)

DFS——拓扑排序

假设在DAG中有一条有向路径从vi到vj，根据拓扑排序的规则，vi一定排在vj之前。在DFS中利用类似后序访问的规则，当vi所有可以达到的节点被访问完以后，vi才会被访问，这样节点被访问的顺序，恰好是拓扑排序的逆序。

DFS-TopSort算法

```
void TopsortbyDFS(Graph& G){                     //深度优先拓扑排序,结果逆序
    for(int i=0; i<G.VerticesNum(); i++)         //初始化标志位
        G.Mark[i]=UNVISITED;
    int *result=new int[G.VerticesNum()];        //最终输出的逆序结果
    int tag=0;
    for(i=0; i<G.VerticesNum(); i++)             //对图的所有顶点进行处理
        if(G.Mark[i]== UNVISITED)
            Do_topsort(G,i,result,tag);          //调用递归函数
    for(i=G.VerticesNum()-1;i>=0;i--) {          //逆序输出
        Visit(G, result[i]);
    }
}
```

```
//深度优先搜索实现的拓扑排序
void Do_topsort(Graph& G, int V, int *result, int& tag){
    G.Mark[V]= VISITED;
    //访问V邻接到的所有未被访问过的顶点
    for(Edge e= G.FirstEdge(V); G.IsEdge(e);e=G.NextEdge(e))
        if(G.Mark[G.ToVertex(e)]== UNVISITED)
            Do_topsort(G, G.ToVertex(e), result, tag); //递归调用
    result[tag++]=V;
}
```

注意：深度优先搜索无法判断环的存在

最短路径算法

单源最短路径（`Dijkstra`算法）

- 从一个顶点出发，到其他所有顶点的最短路径

1、Dijkstra算法

- **Dijkstra**算法是**E.W. Dijkstra**于**1959**年提出的，是目前公认的对**边权非负**情况下的最好算法。
- 基本思想
  - 每次从距离**已生成最短路径的节点集**“**一步之遥**”的节点中，选择距离原点V_0最近的边进行延伸
  - 结果由近及远生成以起始点V_0为根的有向树。
- 是一类贪心算法

实现策略

- 把图中顶点分成两组
  - 第一组：已确定最短路径的顶点
  - 第二组：尚未确定最短路径的顶点
- 按最短路径长度递增顺序逐个把第二组的顶点加到第一组中
  - 直至从s出发可以到达的所有顶点都包括进第一组
- 在合并过程中，保持s到第一组各顶点的最短路径长度都不大于从s到第二组各顶点的最短路径长度
  - 第一组顶点对应的距离值：从s到该顶点的最短路径长度
  - 第二组顶点对应的距离值：从s到该顶点的值包括第一组的顶点为中间顶点的最短路径长度

具体过程

- **初始化**：第一组只包括源点s，第二组包括其它所有顶点
  - s距离值为0，第二组顶点的距离值确定如下：
    - 若有边<s，V_i>或（s，V_i），则Vi的距离值为边所带的权，否则为∞
- **过程**：每次从第二组的顶点中选一个其距离值为最小的顶点V_m加入到第一组中
  - 每往第一组加入顶点V_m，要对第二组各顶点的距离值进行一次修正
    - 若加进V_m做中间顶点，使从s到V_i的最短路径比不加V_m的短，则需要修改V_i的距离值
  - 修改后再选距离值最小的顶点加入到第一组中，重复上述过程
  - **结束条件**：直到图的所有顶点都包括在第一组中或者再也没有可加入到第一组的顶点存在

```cpp
class Dist {
    // Dist类，Dijkstra和Floyd算法用于保存最短路径信息
public:
    int index;   // 顶点的索引值，仅Dijkstra算法用到
    int length;  // 当前最短路径长度
    int pre;     // 路径最后经过的顶点
};

void Dijkstra(Graph& G, int s, Dist* &D) {
    D = new Dist[G.VerticesNum()]; // 初始化Mark数组、D数组
    for (int i = 0; i < G.VerticesNum(); i++) {
        G.Mark[i] = UNVISITED;      // 标记为未访问
        D[i].length = INFINITY;    // 初始化最短路径长度为无穷大
        D[i].index = i;            // 设置顶点索引
        D[i].pre = -1;             // 路径的前驱初始化为 -1
    }
    D[s].length = 0;               // 源点为s，路径长度设为0
    MinHeap<Dist> H(G.EdgesNum()); // 声明一个最小值堆
    H.Insert(D[s]);                 // 将源点初始化进堆

    for (int i = 0; i < G.VerticesNum(); i++) {
        bool FOUND = false;
        Dist d;

        while (!H.empty()) {        // 找下一最短路径的顶点
            d = H.RemoveMin();      // 从堆中取出最小值
            if (G.Mark[d.index] == UNVISITED) { // 如果是未访问的点
                FOUND = true;
                break;              // 该点加入已访问组
            }
        }
        if (!FOUND) break;          // 如果没有找到，退出
```

```
        int v = d.index;
        G.Mark[v] = VISITED;          // 标记顶点v为已访问
        Visit(v);                     // 打印输出（可选操作）

        for (Edge e = G.FirstEdge(v); G.IsEdge(e); e = G.NextEdge(e)) {
            // 遍历v的所有邻接边
            if (D[G.ToVertex(e)].length > D[v].length + G.Weight(e)) {
                // 如果发现更短路径，更新权值
                D[G.ToVertex(e)].length = D[v].length + G.Weight(e);
                D[G.ToVertex(e)].pre = v;     // 更新路径的前驱
                H.Insert(D[G.ToVertex(e)]); // 将更新后的节点重新加入堆
            }
        }
    }
}
```

- 如果不采用最小堆的方式，而是通过两两比较来扫描D数组
  - 每次寻找权值最小结点，需要进行|V|次扫描，每次扫描|V|个顶点（$|V|^2$），而在更新D值处总共扫描|E|次
  - 总时间代价为$\Theta(|V|^2 + |E|) = \Theta(|V|^2)$
- 如果采用最小堆的方式
  - 每次改变D[i].length，通过先删除再重新插入的方法来改变顶点i在堆中的位置
  - 或者仅为某个顶点添加一个新值(更小的)，作为堆中新元素(而不作删除旧值的操作，因为旧值被找到时，该顶点一定被标记为VISITED，从而被忽略)。
  - 不作删除旧值的缺点是，在最差情况下，它将使堆中元素数目由$\Theta(|V|)$增加到$\Theta(|E|)$，此时总的时间代价为$\Theta((|V|+|E|)\log|E|)$，因为处理每条边时都必须对堆进行一次重排

北京大学信息科学技术学院 数据结构与算法 106

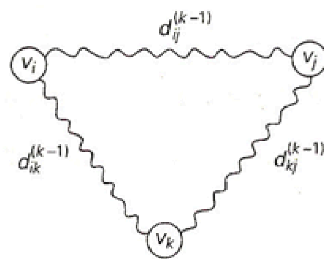
多源最短路径（`Floyd`算法)

- 方法一：反复执行Dijkstra算法，时间复杂度为$O(n^3)$
- 方法二：Floyd算法，时间复杂度为$O(n^3)$

- 假设用相邻矩阵adj表示图
  - 任意两点间距离是边的权，如果两点间没有边直接相连，则权为无穷大（∞）
- 在原图的相邻矩阵$adj^{(0)}$上做n次迭代，递归地产生一个矩阵序列$adj^{(1)}$, $adj^{(2)}$,…, $adj^{(n)}$
  - $adj^{(k)}$[i，j]等于从顶点V_i到顶点V_j中间顶点序号不大于k的最短路径长度
- $adj^{(n)}$包括了所有最终的最短路径

递推公式

- 假设已求得矩阵$adj^{(k-1)}$，那么从顶点V_i到顶点V_j中间顶点的序号不大于k的最短路径有两种情况：
  - 中间不经过顶点V_k，那么就有$adj^{(k)}[i，j]=adj^{(k-1)}[i，j]$
  - 中间经过顶点V_k，那么$adj^{(k)}[i,j] < adj^{(k-1)}[i,j]$，且$adj^{(k)}[i,j]= adj^{(k-1)}[i,k] + adj^{(k-1)}[k,j]$



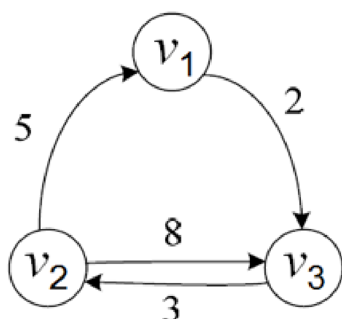
最短路径确定

- 确定最短路径
  - 设置一个n×n的矩阵path，path[i，j]是由顶点v_i到顶点v_j的最短路径上排在顶点v_j前面的那个顶点，即当*k*是使得$adj^{(k)}[i,j]$达到最小值，那么就置path[i,j]=path[k,j]
  - 如果当前没有最短路径时，就将path[i，j]置为-1。

最短路径确定

- 确定最短路径
  - 设置一个n×n的矩阵path，path[i，j]是由顶点v_i到顶点v_j的最短路径上排在顶点v_j前面的那个顶点，即当*k*是使得$adj^{(k)}[i,j]$达到最小值，那么就置path[i,j]=path[k,j]
  - 如果当前没有最短路径时，就将path[i，j]置为-1。

示例



$$adj = \begin{bmatrix} 0 & \infty & 2 \\ 5 & 0 & 8 \\ \infty & 3 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 1 & -1 & 1 \\ 2 & 2 & 2 \\ -1 & 3 & 3 \end{bmatrix}$$

$$adj^{(1)} = \begin{bmatrix} 0 & \infty & 2 \\ 5 & 0 & 7 \\ \infty & 3 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 1 & -1 & 1 \\ 2 & 2 & 1 \\ -1 & 3 & 3 \end{bmatrix}$$

$$adj^{(2)} = \begin{bmatrix} 0 & \infty & 2 \\ 5 & 0 & 7 \\ 8 & 3 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 1 & -1 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \end{bmatrix}$$

$$adj^{(3)} = \begin{bmatrix} 0 & 5 & 2 \\ 5 & 0 & 7 \\ 8 & 3 & 0 \end{bmatrix} \quad path = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \end{bmatrix}$$

Floyd算法实现

```
void Floyd(Graph& G, Dist** &D){
    int i,j,v;                                   //i, j, v作为计数器
    D=new Dist*[G.VerticesNum()];
    for(i=0; ;i<G.VerticesNum();i++)             //申请数据D的空间
        D[i]=new Dist[G.VerticesNum()];
    for(i=0;i<G.VerticesNum();i++)               //初始化D数组
        for(j=0;j<G.VerticesNum();j++)
            if(i==j){
                D[i][j].length=0;                //权值矩阵
                D[i][j].pre=i;                   //path矩阵
            }else {
                D[i][j]=INFINITY;
                D[i][j].pre=-1;
    }
```

初始化

```
for(v=0;v<G.VerticesNum();v++)          //矩阵初始化，仅初始化邻接顶点
    for(Edge e=G.FirstEdge(v); G.IsEdge(e); e=G.NextEdge(e)){
        D[v][G.ToVertex(e)].length=G.Weight(e);
        D[v][G.ToVertex(e)].pre=v;
    }
//如果两顶点间最短路径经过顶点v，则有权值进行更新！
for(v=0;v<G.VerticesNum();v++)
    for(i=0;i<G.VerticesNum();i++)
        for(j=0;j<G.VerticesNum();j++)
            if((D[i][v].length +D[v][j].length) < D[i][j].length){
                D[i][j].length =D[i][v].length +D[v][j].length;
                D[i][j].pre=D[v][j].pre;
            }
}
```

权值、路径更新

- **时间复杂性：三重for循环，复杂度是O(n^3)，适合稠密图**

最小生成（支撑）树

最小支撑树

- 最小支撑树（Minimum-cost Spanning Tree，MST）
  - 对于带权的**连通无向图G**，其最小支撑树是一个包括G的所有顶点和部分边的图，这部分的边满足下列条件：
    - 保证图的连通性
    - 边权值总和最小
- 代表算法
  - Prim算法
  - Kruskal算法

- 具体操作
  - 从图中任意一个顶点开始，把这个顶点包括在MST里
  - 然后，在那些其一个端点已在MST里，另一个端点还未在MST里的边中，找权最小的一条边（相同边存在，任选择其一），并把这条边和其不在MST里的那个端点包括进MST里
  - 如此进行下去，每次往MST里加一个顶点和一条权最小的边，直到把所有的顶点都包括进MST里
- **MST不唯一，但是最小权值是确定的**

Prim算法实现

```
void Prim(Graph& G, int s, Edge* &MST ) {
    int MSTtag=0;                              //最小支撑树边的标号
    Edge *MST=new Edge[G.VerticesNum()-1];
    MinHeap<Edge> H(G.EdgesNum());
    for(int i=0;i<G.VerticesNum();i++)         //初始化Mark数组、距离数组
        G.Mark[i]=UNVISITED;
    int v=s;                                   //开始顶点
    G.Mark[v]=VISITED;                         //开始顶点首先被标记
    do{  //将以v为顶点，另一顶点未被标记的边插入最小值堆H
        for(Edge e= G. FirstEdge(v);G.IsEdge(e);e=G. NextEdge(e))
            if(G. Mark[G. ToVertex(e)]==UNVISITED)
                H.Insert(e);
        bool Found=false;
        while(!H.empty()) {                    //寻找下一条权最小的边
            e=H.RemoveMin();
```

```
            if(G. Mark[G. ToVertex(e)]==UNVISITED){
                Found=true;
                break;
            }
        } //end while
        if(!Found){
            Print("不存在最小支撑树。");
            delete [] MST;              //释放空间
            MST=NULL;                   //MST是空数组
            return;
        }
        v= G. ToVertex(e);
        G.Mark[v]=VISITED;     //在顶点v的标志位上做已访问的标记
        AddEdgetoMST(e,MST,MSTtag++); //将边e加到MST中
    }  while(MSTtag < (G. VerticesNum()-1))
}
```

Kruskal算法

Kruskal算法的基本思想

- 对于图G=（V，E），开始时，将顶点集分为|V|个等价类，每个等价类包括一个顶点
- 然后，以权的大小为顺序处理各条边，如果某条边连接两个不同等价类的顶点，则这条边被添加到MST，两个等价类被合并为一个；
- 反复执行此过程，直到只剩下一个等价类

Kruskal算法描述

```
void Kruskal(Graph& G, Edge* &MST{
    Partree A(G.VerticesNum());                    //等价类
    MinHeap<Edge> H(G.EdgesNum());                 //声明一个最小堆
    MST=new Edge[G.VerticesNum()-1];               //最小支撑树
    int MSTtag=0;                             //最小支撑树边的标号
    for(int i=0; i<G.VerticesNum(); i++) {   //将图的所有边插入最小值堆H中
        for(Edge e= G.FirstEdge(i); G.IsEdge(e);e=G. NextEdge(e))
            if(G.FromVertex(e)< G.ToVertex(e))
                H.Insert(e);
    int EquNum=G.VerticesNum();               //开始时有|V|个等价类
    while(EquNum>1) {                         //合并等价类
        Edge e=H.RemoveMin();                 //获得下一条权最小的边
```

边集入堆


```
        int from=G.FromVertex(e);            //记录该条边的信息
        int to= G.ToVertex(e);
        if(A.differ(from,to)) {      //如果边e的两个顶点不在一个等价类
            //将边e的两个顶点所在的两个等价类合并为一个
            A.UNION(from,to);
            AddEdgetoMST(e,MST,MSTtag++); //将边e加到MST
            EquNum--;                              //将等价类的个数减1
        }
    }//end while
}
```

- 算法代价：$O(E \log E)$，即堆排序的时间复杂度

内排序

插入排序

直接插入排序

```
void stringInsertSort(int A[], int n) {
    for(int i=1;i<n;i++){
        for(int j=i;j>0;j--){
            if(Array[j]<Array[j-1]){
                swap(Array[j],Array[j-1]);
            }
            else{
                break;
            }
        }
    }
}
```

稳定

时间复杂度：$O(n^2)$

空间复杂度：$O(1)$

优化的插入排序算法

```
void ImprovedInsertSort(int A[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = A[i];
        int j = i - 1;
        while (j >= 0 && A[j] > temp) {
            A[j + 1] = A[j];
            j--;
        }
        A[j + 1] = temp;
    }
}
```

基于二分查找的插入排序

```
void BinaryInsertSort(int A[], int n) {
    for (int i = 1; i < n; i++) {
        int temp = A[i];
        int low = 0, high = i - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (A[mid] > temp) high = mid - 1;
            else low = mid + 1;
        }
        for (int j = i - 1; j >= low ; j--) {
            A[j + 1] = A[j];
        }
        A[low] = temp;//插入left
    }
}
```

- 最佳情况下代价为 $O(n \log n)$

- 平均情况下代价为 $O(n^2)$

冒泡排序

冒泡排序算法

```
void BubbleSorter (Record Array[], int n){
    for (int i=1; i<n; i++)                //向前冒泡
        //依次比较相邻记录，如果发现逆置，就交换
        for (int j=n-1; j>=i; j--)
            if (Array[j] < Array[j-1])
                swap(Array, j, j-1);
}
```

算法分析

- 算法是稳定的
- 空间代价：Θ(1)的临时空间
- 时间代价
  - 比较次数

$$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2=\Theta(n^2)$$

  - 交换次数最多为 $\Theta(n^2)$，最少为0，平均为 $\Theta(n^2)$。
  - 最大，最小，平均时间代价均为 $\Theta(n^2)$。

直接选择排序

- 基本思想
  - 每一趟在后面n-i个待排记录中选取**最小记录和第i个记录互换**。
- 具体过程
  - 首先，在n个记录中选择最小者与r[0]互换；
  - 然后，从剩余的n-1个记录中选择最小者与r[1]互换；
  - …如此下去，直到全部有序为止。
- 优点：实现简单
- 缺点：每趟只能确定一个元素，表长为n时需要n-1趟

不稳定

直接选择排序算法

```
void StraightSelectSorter (Record Array[], int n) {
    // 依次选出第i小的记录，即剩余记录中最小的那个
    for (int i=0; i<n-1; i++){
        int Smallest = i;              // 首先假设记录i就是最小的
        for (int j=i+1;j<n; j++)       //开始向后扫描所有剩余记录
            if ( Array[j] < Array[Smallest])
                Smallest = j;          // 如发现更小记录，记录其位置
        swap(Array, i, Smallest);      //将第i小的记录放在第i个位置
    }
}
```

与冒泡排序的关系: (1) 冒泡排序从后面两两交换冒出最小的；而直接选择排序是直接找到最小的和第一个进行交换！（2）前者是稳定的，后者是不稳定的！

shell排序

直接插入排序的2个性质

- 当n较小时，直接插入排序较快
- 当序列基本有序时，直接插入排序较快

思想：

- 先将整个待排序的记录序列分割成若干子序列分别进行直接插入排序
- 逐渐扩大子序列的长度，直到整个序列被分为一个子序列

具体实现方法

1) 选定一个**间隔增量序列**（$n>d_1>d_2>\ldots>d_t=1$）

   - n：文件长度，d_i：间隔增量，t：排序趟数

2) 将文件按 d_1 分组（彼此相距 d_1 的记录划为一组），在各组内采用直接插入法进行排序。

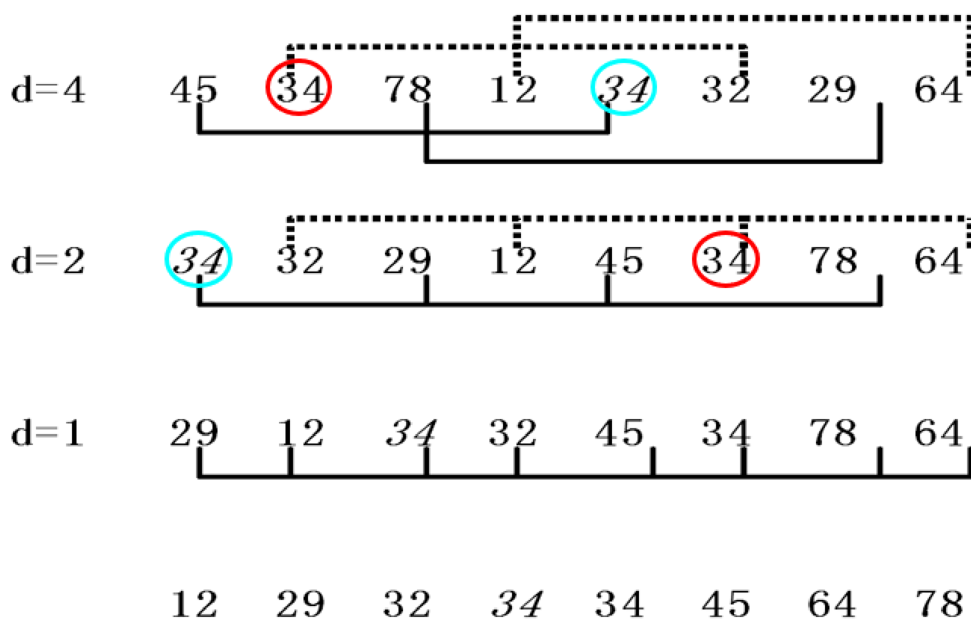
3) 分别按 d_2, ...d_t 重复上述分组和排序工作。

- Shell 最初提出的增量序列是

$$d_1=\lfloor n/2\rfloor,\ d_{i+1}=\lfloor d_i/2\rfloor$$

北京大学信息科学技术学院 数据结构与算法

Shell排序过程



北京大学信息科学技术学院 数据结构与算法 30

```
void shellsort(int A[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for(int j=0;j<gap;j++){
```

```
                ModifiedInsertSort(A, n-j, gap);
            }
        }
    }
void ModifiedInsertSort(int A[], int n, int gap) {
    for (int i = gap; i < n; i++) {
        int temp = A[i];
        int j = i - gap;
        while (j >= 0 && A[j] > temp) {
            A[j + gap] = A[j];
            j -= gap;
        }
        A[j + gap] = temp;
    }
}
```

快速排序

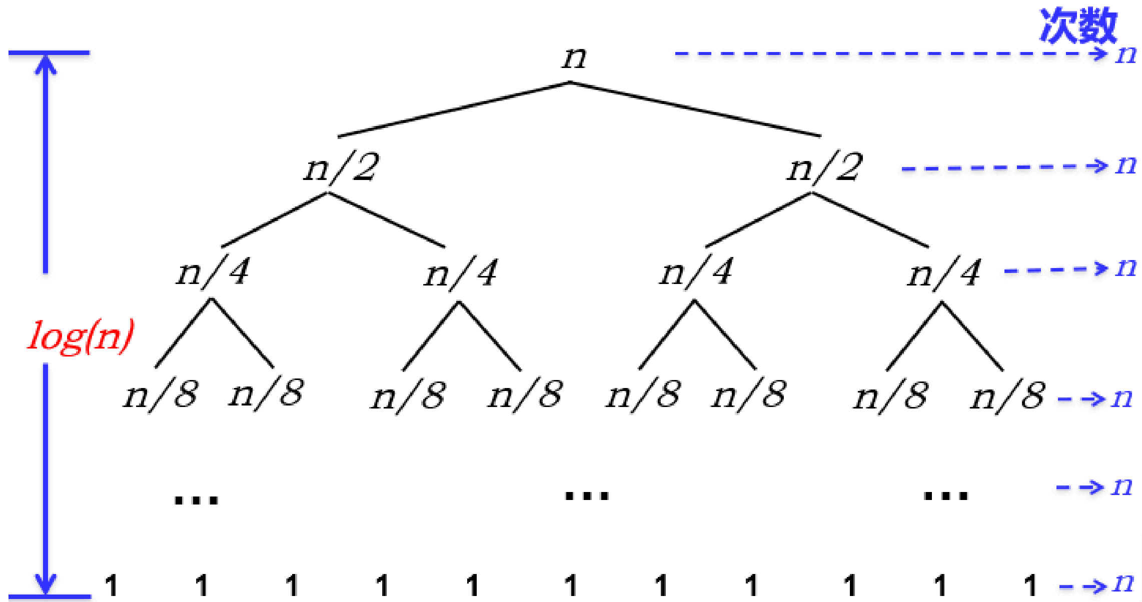
- 轴值选择：从待排序列中任选一个元素k作为轴值
- 序列划分：划分为子序列L和R，使得L中的元素都小于k，R中的元素都大于k
- 递归排序：对L和R分别进行快速排序

```
void Quicksort(int A[], int low, int high) {
    if (low < high) {
        int pivot=SelectPivot(left,right);
        int pivot = Partition(A, low, high);
        Quicksort(A, low, pivot - 1);
        Quicksort(A, pivot + 1, high);
    }
}
int Partition(int Array[], int left, int right) {
    int i=left,j=right;
    int pivotValue=Array[left];     //将轴值放在临时变量中
    while(i!=j){
        while((Array[j]>pivotValue)&&(i<j))j--; //左移
        if(i<j){Array[i]=Array[j];i++;}                    //交换
        while((Array[i]<=pivotValue)&&(i<j))i++;
        if(i<j){Array[j]=Array[i];j--;}
    }
    Array[i]=pivotValue; //轴值归位
    return i;
}
```

算法分析

算法分析(1)

➢ **最佳性能：** $T(n)=O(n\lg(n))$

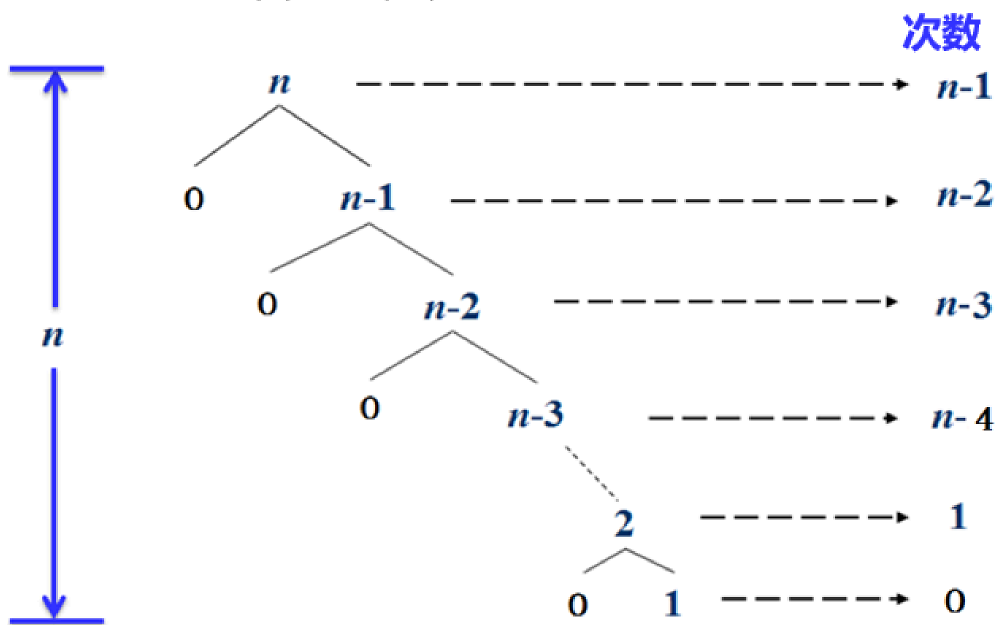


算法分析(2)

➢ **最差性能：** $T(n)=O(n^2)$

每层比较

次数



平均性能推导：

$$T(n) = O(nlogn)$$

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)) + cn$$

$$nT(n) = 2\sum_{i=0}^{n-1}T(i) + cn^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

$$nT(n) = (n+1)T(n-1) + 2cn$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

算法分析(3)(续)

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$

…

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

公式两侧分别相加求和

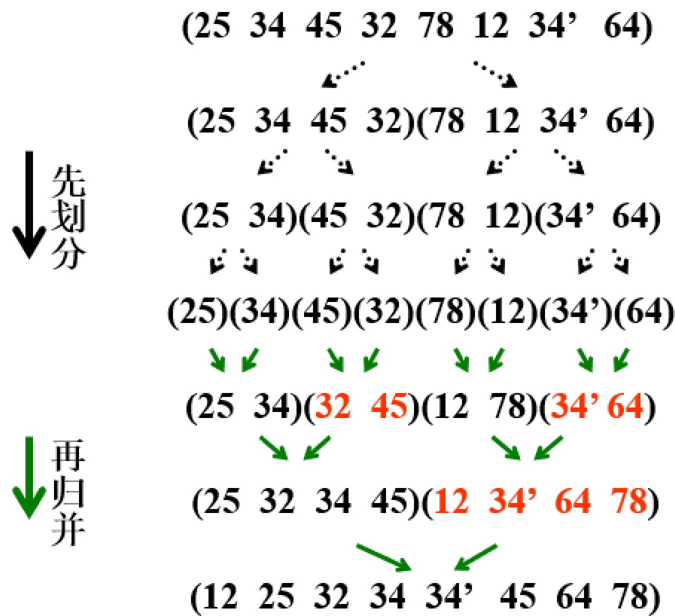
$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\left(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{3}\right)$$

$$\sum_{i=3}^{n+1}\frac{1}{i} = log(n+1) + \gamma + \frac{3}{2}$$

$$\frac{T(n)}{n+1} \approx \frac{T(1)}{2} + 2c * log(n+1)$$

$$T(n) = O(nlogn)$$

归并排序



两路归并排序算法

```
template <class Record>
void MergeSort(Record Array[], Record TempArray[], int left, int right)  {
    // Array为待排序数组，left，right两端
    int middle;
    if (left < right)  {                    //如果序列中只有0或1个记录，就不用排序
        middle = (left + right) / 2;        //从中间划分为两个子序列
        MergeSort(Array,TempArray,left,middle);          //对左边一半进行递归
        MergeSort(Array, TempArray,middle+1,right);      //对右边一半进行递归
        Merge(Array, TempArray,left,right,middle);       //进行归并
    }
}
```

```
template <class Record>
void Merge(Record Array[], Record TempArray[], int left, int right, int middle)  {
    int i, j, index1, index2;
    for (j = left; j <= right; j++)                        // 将数组暂存入临时数组
        TempArray[j] = Array[j];
    index1 = left;                                         // 左边子序列的起始位置
    index2 = middle+1;                                     // 右边子序列的起始位置
    i = left;                                              // 从左开始归并
    while (index1 <= middle && index2 <= right)  {
        // 取较小者插入合并数组中
        if (TempArray[index1] <= TempArray[index2]) //为保稳定，相等时左边优先
            Array[i++] = TempArray[index1++];
        else  Array[i++] = TempArray[index2++];
    }
```



```
    while (index1 <= middle)                  // 只剩左序列，可以直接复制
        Array[i++] = TempArray[index1++];
    while (index2 <= right)           // 与上个循环互斥，直接复制剩余的右序列
        Array[i++] = TempArray[index2++];
}
```

归并排序性能分析

- 容易看出，对 n 个记录进行归并排序的时间复杂度为 O(nlogn)。即：
  - **每一趟归并的时间复杂度为 O(n)，**
  - **总共需进行⌈logn⌉趟。**
  - **归并排序需要附加一倍的存储量O(n)**
    - **是辅助存储量最多的一种排序方法**
- 是稳定排序算法

堆排序

略

算法分析

- 非稳定性排序
- 建堆：Θ(n)
- 删除一次堆顶重新建堆：Θ(log *n*)
- 一次建堆 ,n次删除堆顶，总时间代价为Θ(*n*log *n*)
- 理论上，堆排序最佳、最差、平均情况下的时间代价均为Θ(*n*log *n*)
- 辅助空间代价：Θ(1)

桶排序

- 桶排序（Bucket Sorting）
  - 序列中记录取值范围：[0，m)，标记为m个桶
  - 将相同值的记录**分配**到对应桶中
  - 按桶号依次**收集**记录，组成有序序列

如何将序列中的记录分配到相应的桶中?

后继起始下标

待排数组： 7 3 8 9 6 1 8’ 1’ 2

Count数组：

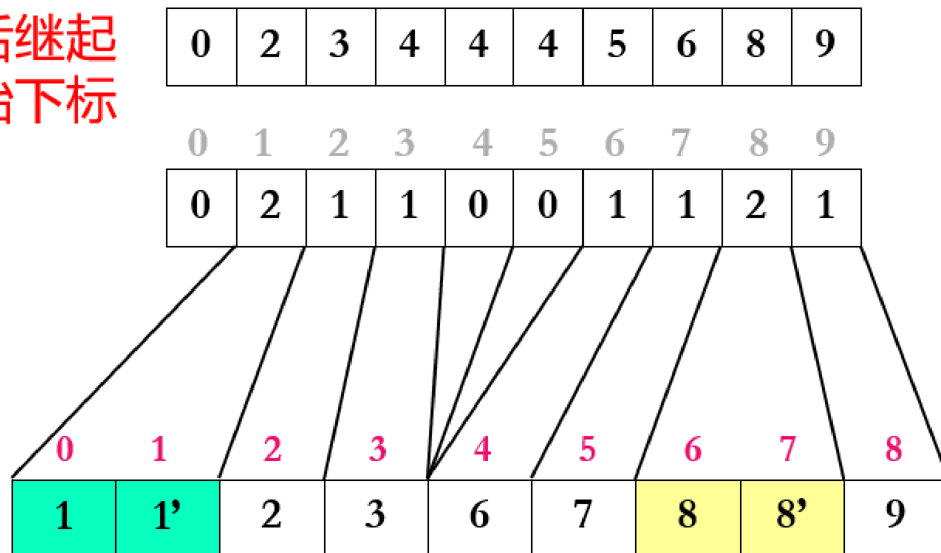
0	1	2	3	4	5	6	7	8	9
0	2	1	1	0	0	1	1	2	1

count[i]=count[i-1]+count [i]

后继起始下标：

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

桶计数与排序序列的映射关系



桶排序过程示例

收集自后向前，保持稳定！

待排数组： 7 3 8 9 6 1 8’ 1’ 2

	0	1	2	3	4	5	6	7	8	9
第一趟count	0	2	1	1	0	0	1	1	2	1
后继起始下标：	0	0	2	4	4	4	5	6	7	9

	0	1	2	3	4	5	6	7	8
收集：	1	1’	2	3	6	7	8	8’	9

count[i]记录了(i+1)开始的位置；元素i应该从 Arrary[count[i]-1]往前追溯


```
template <class Record>
void BucketSort(Record Array[], int n, int m) {
    int *TempArray = new Record[n];        // 临时数组
    int *count = new int[m];               // 小于或等于i的元素个数
    int i;
    for (i = 0; i < n; i++)                // 把序列复制到临时数组
        TempArray[i] = Array[i];
    for (i = 0; i < m; i++)                // 所有计数器初始都为0
        count[i] = 0;
```

```
for (i = 0; i < n; i++)                    // 统计每个取值出现的次数
      count[Array[i]]++;
   for (i = 1; i < m; i++)                 // 后继起始地址计算
      count[i] = count[i-1]+count [i];
   // 从尾部开始按顺序输出，保证排序的稳定性
   for (i = n-1; i >= 0; i--)
      Array[--count[TempArray[i]]] = TempArray[i];
}
```

算法分析

- 时间代价
  - **统计计数：Θ(m+n)**
  - **总的时间代价：Θ(m+n)**
- 空间代价
  - **需要m个计数器，n个临时空间**
  - **总的空间代价：Θ(m+n)**
  - **适用于m相对于n很小的情况**
- 稳定算法

- 只适用于m较小的情况

基数排序

2、基数排序

- 排序码由多个部分组成
  - 序列$R=\{r_0,r_1,\ldots,r_{n-1}\}$，每个排序码K由d位子排序码组成
    - $K=(k_{d-1},k_{d-2},\ldots,k_1,k_0)$，
  - R有序就是对于任意两记录R_i, R_j，满足

    $(k_{i,d-1}, k_{i,d-2}, \ldots, k_{i,1}, k_{i,0}) \le (k_{j,d-1}, k_{j,d-2}, \ldots, k_{j,1}, k_{j,0})$
    - 其中k_{d-1}称为最高排序码，k_0称为最低排序码。
- K_i的取值范围称为**基数**，记做r

- 0到9999之间整数排序
  - 将4位数看作是由4个排序码决定，即千、百、十、个位，其中千位为最高排序码，个位为最低排序码。基数r=10
  - 可以按千、百、十、个位数字依次进行4次桶式排序
  - 4趟分配排序后，整个序列就排好序

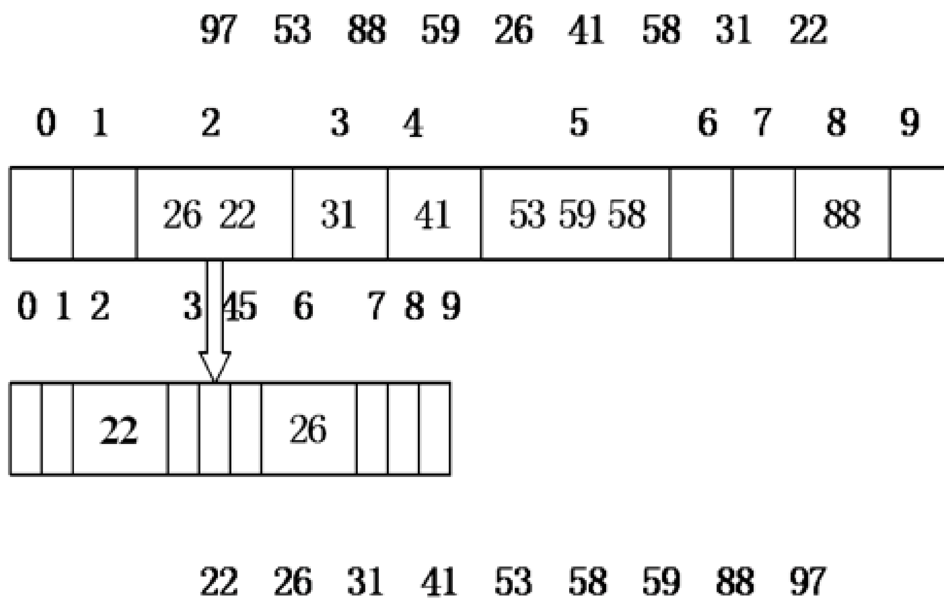
- 扑克牌
  - 若规定花色和面值的顺序关系为：
  - 花色：♦ < ♣ < ♠ < ♥
  - 面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A
  - 则可以先按花色排序，花色相同者再按面值排序；
  - 也可以先按面值排序，面值相同者再按花色排序。

基数排序分类

- 高位优先法（MSD，Most Significant Digit first）
- 低位优先法（LSD，Least Significant Digit first）
- 扑克牌排序
  - **MSD方法思路：先设立4个花色“箱”，将全部牌按花色分别归入4个箱内（每个箱中有13张牌）；然后对每个箱中的牌按面值进行插入排序。**
  - **LSD方法思路：先按面值分成13堆（每堆4张牌），然后对每堆牌按花色进行排序（用插入排序等稳定的算法）**

高位优先法

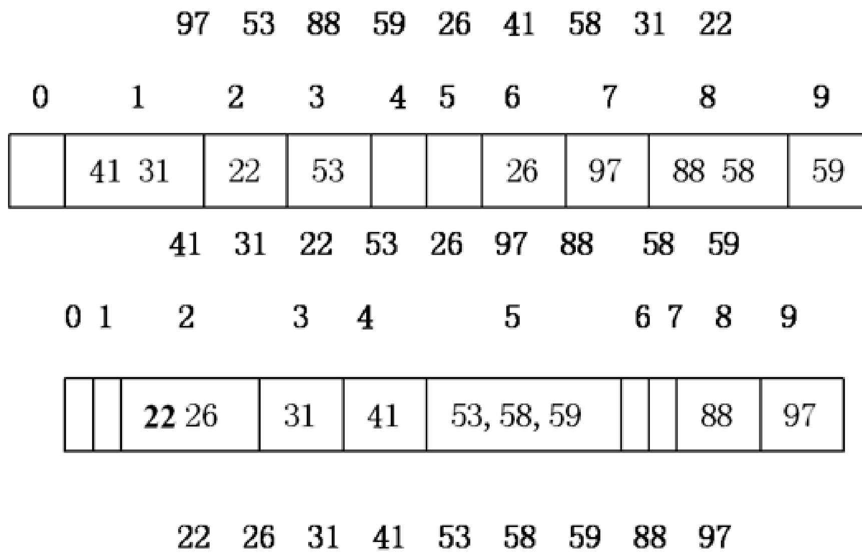
- 先按最高位k_{d-1}进行桶式排序，将序列分成若干个桶中
- 再按次高位k_{d-2}进行桶式排序，分成更小的桶
- 依次重复，直到对k_0排序后，分成最小的桶，每个桶内含有相同的排序码(k_{d-1}，k_{d-2}，…，k_1，k_0)；
- 最后将所有的桶依次连接在一起，成为一个有序序列
- 这是一个分、分、…、分、收的过程
- 是一个递归分治问题



(a) 高位优先

低位优先法

- 从最低位k_0开始对整个序列进行排序
- 对于排好序的整个序列再用次低位k_1排序
- 依次重复，直至对最高位k_{d-1}排好序后，整个序列成为有序的
- 这是一个**分、收；分、收；…；分、收**的过程。
- 比较简单，计算机常用



(b)低位优先

如何实现

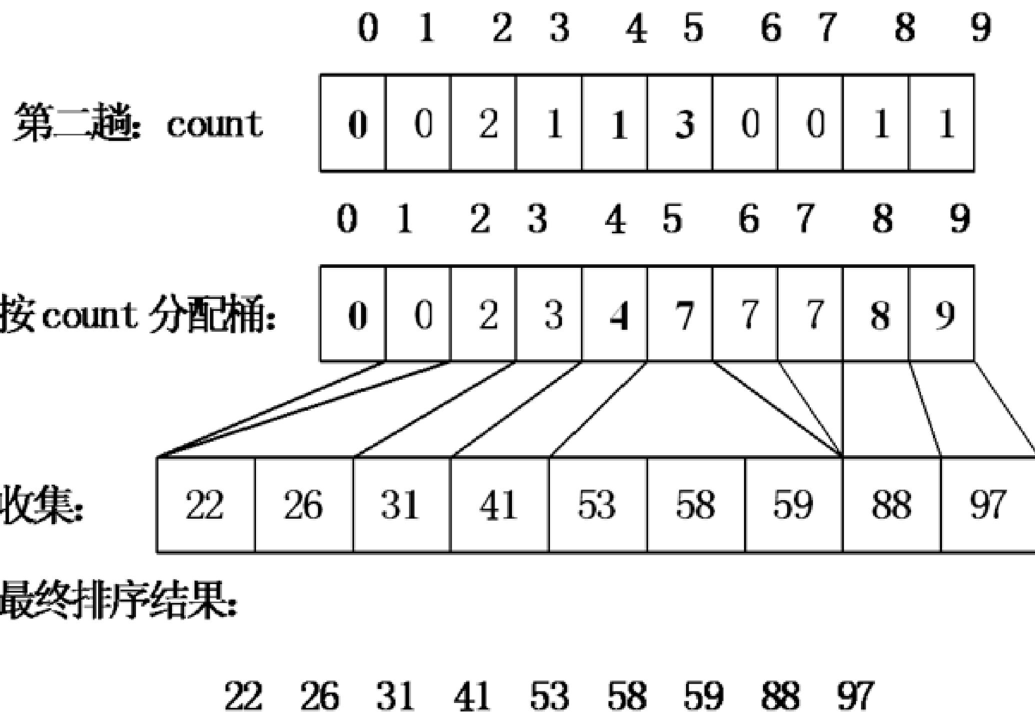
- 数组R长度n
- 基数r
- 排序码位数：d

初始数组内容: 97 53 88 59 26 41 58 31 22

	0	1	2	3	4	5	6	7	8	9
第一趟: count	0	2	1	1	0	0	1	1	2	1
按count分配桶:	0	2	3	4	4	4	5	6	8	9

收集:

41	31	22	53	26	97	88	58	59



基于数组的基数排序算法

```
void RadixSorter<Record>::Sort(Record Array[], int n,int d, int r){
    //n为数组长度，d为排序码数，r为基数
    Record *TempArray =new Record[n];          //临时数组
    int* count= new int[r];                    //计数器
    int i,j,k;
    int Radix=1;
    for (i=1; i<=d; i++) {                     //取Array[j]的第i位排序码
        for (j=0; j<r; j++)                    //分别对第i个排序码分配
            count[j] = 0;                      // 初始计数器均为0
```

```
for (j=0; j<n; j++){    //统计每个桶中的记录数
        //取Array[j]的第i位排序码
        k=(Array[j] /Radix)%r;
        count[k]++;          //相应计数器加1
}
// 每个元素的后继起始下标地址
for (j=1; j<r; j++)
        count[j] = count[j-1] + count[j];
```

```
    //将所有桶中的记录依次收集到TempArray中
    for (j=n-1; j>=0; j--) {
        k=(Array[j] /Radix)%r;//取Array[j]的第i位排序码
        TempArray[--count[k]] = Array[j];
    }
    for (j=0; j<n; j++)// 将临时数组中的内容复制到Array中
        Array[j] = TempArray[j];
    Radix*=r;
  }
}
```

算法分析

- 临时数组,n
- r个计数器
- 空间代价$O((n+r))$
- d次桶排序，每次代价$O(n+r)$
- 时间代价$O(d(n+r))$

重点

8.7 各种排序算法的理论和实验时间代价

算法	最大时间	平均时间	最小时间	辅助空间代价	稳定性
直接插入排序	Θ(n^2)	Θ(n^2)	Θ(n)	Θ(1)	稳定
二分法插入排序	Θ(n^2)	Θ(n^2)	Θ(nlog^n)	Θ(1)	稳定
冒泡排序	Θ(n^2)	Θ(n^2)	Θ(n^2)	Θ(1)	稳定
改进的冒泡排序	Θ(n^2)	Θ(n^2)	Θ(n)	Θ(1)	稳定
选择排序	Θ(n^2)	Θ(n^2)	Θ(n^2)	Θ(1)	不稳定
Shell排序(3)	Θ(n3/2)	Θ(n3/2)	Θ(n3/2)	Θ(1)	不稳定
快速排序	Θ(n^2)	Θ(nlog^n)	Θ(nlog^n)	Θ(log^n)	不稳定
归并排序	Θ(nlog^n)	Θ(nlog^n)	Θ(nlog^n)	Θ(n)	稳定
堆排序	Θ(nlog^n)	Θ(nlog^n)	Θ(nlog^n)	Θ(1)	不稳定
桶式排序	Θ(n+m)	Θ(n+m)	Θ(n+m)	Θ(n+m)	稳定
基数排序	Θ(d·(n+r))	Θ(d·(n+r))	Θ(d·(n+r))	Θ(n+r)	稳定

外排序

- 文件流是以外存文件为输入输出对象的数据流
- 文件流与文件不是同一个概念，文件流不是由若干个文件组成的流
- 文件流本身不是文件，而只是以文件为输入输出对象的流

9.3 外排序

- 根据内存的大小，将外存中的数据文件划分成若干段，每次把其中一段读入内存并用内排序方法进行排序。
- 这些已排序的段或子文件称为**顺串或归并段**。
- 顺串写回外存等待进一步处理，让出内存空间处理文件的其它未排序的段。

外排序的基本过程

I. 置换选择排序

- **目的**：把外存文件初始化为尽可能长的有序顺串集；

II.归并排序

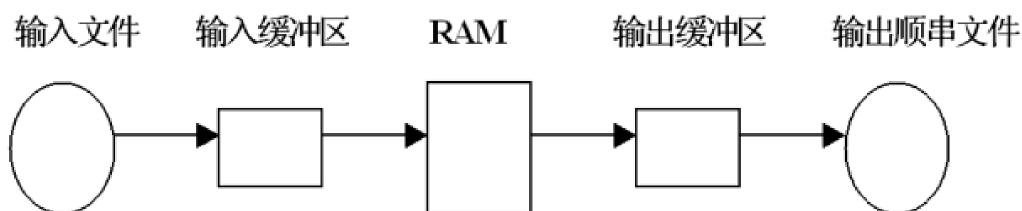
- **目的**：把顺串集合逐趟归并排序，形成全局有序的外存文件

外排序的时间组成

- 产生初始顺串的内排序所需时间
- 初始化顺串和归并过程所需的读写(I/O)时间
- 内部归并所需要的时间

减少外存信息的读写次数是提高外部排序效率的关键

9.3.1 置换选择排序



- **目的：** 将文件生成若干初始顺串（顺串越长越好，个数越少越好）
- **实现：** 借助在RAM中的堆来完成

置换选择算法

1. 初始化最小堆：目的是提高RAM中排序的效率

   (a) 从缓冲区读M个记录放到数组RAM中

   (b) 设置堆尾标志：LAST = M - 1

   (c) 建立一个最小值堆

置换选择算法(续)

2. 重复以下步骤，直至堆空（结束条件）（即LAST < 0）

(a) 把具有最小关键码值的记录(根结点)送到输出缓冲区

(b) 设R是输入缓冲区中的下一条记录

i. 如果R的关键码不小于刚输出的关键码值，则把R放到根结点

ii. 否则，使用数组中LAST位置的记录代替根结点，然后把R放到LAST位置（等待下一顺串处理），设置LAST = LAST-1

(c)重新排列堆，筛出根结点

算法分析

- 算法结束后，RAM中也填满了不能处理的数据，直接建成堆，留待下一顺串来处理
- 大小是M的堆，最小顺串的长度为M的记录
  - 至少原来堆中的那些记录将成为顺串的一部分
  - 最好情况下，有可能一次就把整个文件生成为一个顺串（何种情况下？）
  - 平均长度2M

```
//A是从外存读入n个元素后所存放的数组
template <class Elem>
void ReplacementSelection(Elem * A, int n, const char * in,
    const char * out){
    Elem mval;              //存放最小堆的最小值
    Elem r;                 //存放从输入缓冲区中读入的元素
    FILE * iptF;            //输入文件句柄
    FILE * optF;            //输出文件句柄
    Buffer<Elem> input;             //输入 buffer
    Buffer<Elem> output;            // 输出buffer
```

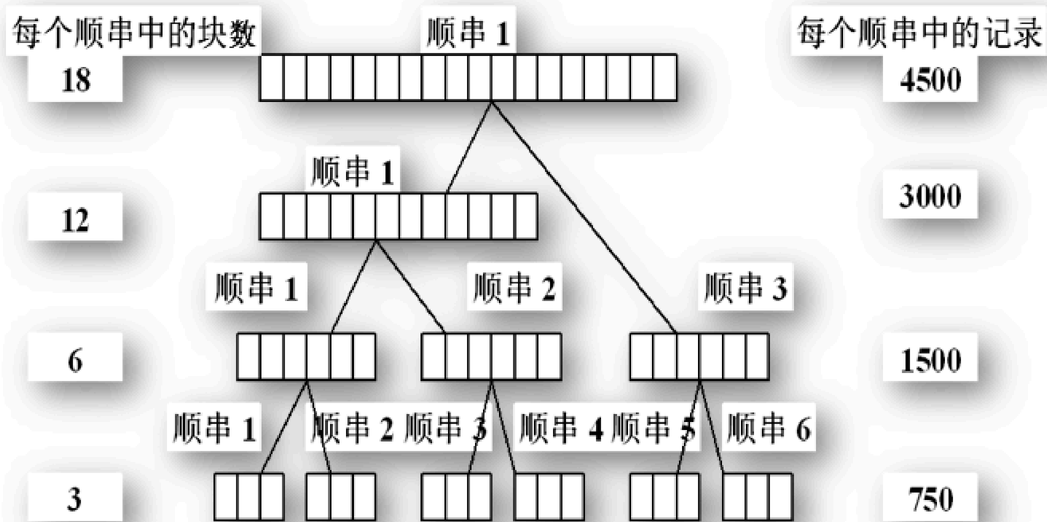
```
    //初始化输入输出文件
    initFiles(inputFile, outputFile, in, out);
    //初始化堆的数据，读入n个数据
    initMinHeapArry(inputFile, n, A);
    //建立最小值堆
    MinHeap<Elem> H(A, n, n);
     //初始化inputbuffer，读入部分数据
    initInputBuffer(input, inputFile);
```

```
for(int last =n-1; last >= 0;){
    mval = H.heapArray[0];    //堆的最小值
    sendToOutputBuffer(input,output,iptF,optF, mval);
    input.read(r); //从输入缓冲区读入一个记录
    if(!less(r, mval))      H.heapArray[0] = r;
    else  {//否则用last位置记录代替根结点，把r放到last
        H.heapArray[0] = H.heapArray[last];
        H.heapArray[last] = r;
```


归并排序

9.3.2 归并排序

➤产生顺串→归并排序



- 二路归并
  - m为顺串的个数
  - 合并树高$\lceil \log_2 m \rceil+1$，进行$\lceil \log_2 m \rceil$遍扫描
  - 2个输入缓冲区，1个输出缓冲区

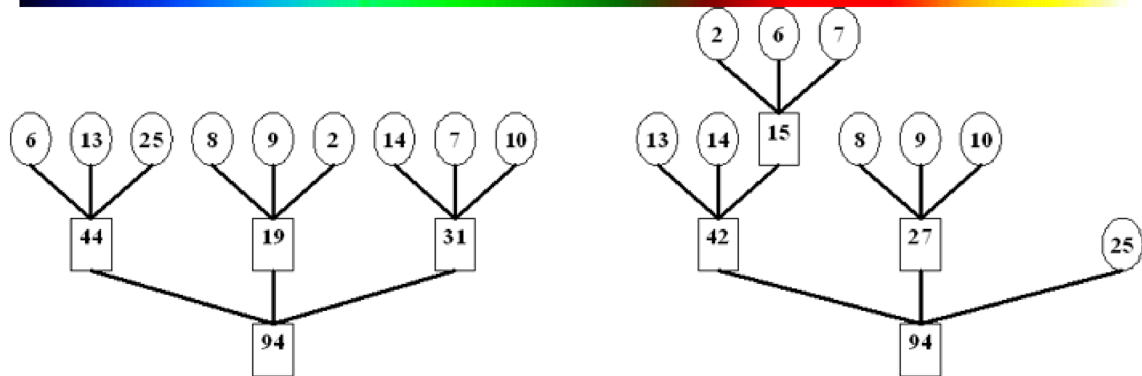
归并排序的趟数

- 所需读写外存次数与归并趟数有关系
- 假设有m个初始顺串，每次对k个顺串进行归并，归并趟数为$\lceil \log_k m \rceil$
- 为了减少归并趟数，可以从两个方面着手
  - 减少初始顺串的个数m
  - 增加同时归并的顺串数量k

最佳归并树

- 进行多路归并时，各初始顺串的长度不同，对外存扫描的次数，即执行时间会产生影响
- 把所有初始顺串的块数作为树的叶结点，如果是K路归并则建立起一棵K-叉Huffman树。这样的一棵Huffman树就是最佳归并树。
- 通过最佳归并树进行多路归并可以使对外存的I/O降到最少，提高归并执行效率

最佳归并树



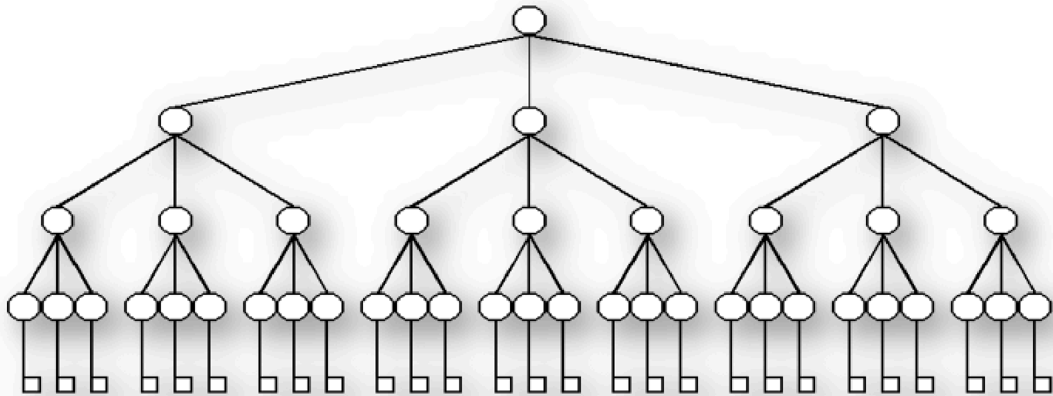
(a)一棵普通的归并树　　　　(b)最佳归并树

- （a）访外总次数为(6+13+25+8+9+2+14+7+10)×2×2=376
- （b）访外总次数为

(2+6+7)×3×2+(13+14)×2×2+(8+9+10)×2×2+25×2=356

多路归并树

➢ k路归并指每次将k个顺串合并成一个顺串



多(3)路归并外排序

如何减小k路比较时的比较次数

- 选择树
  - 赢者树
  - 败者树

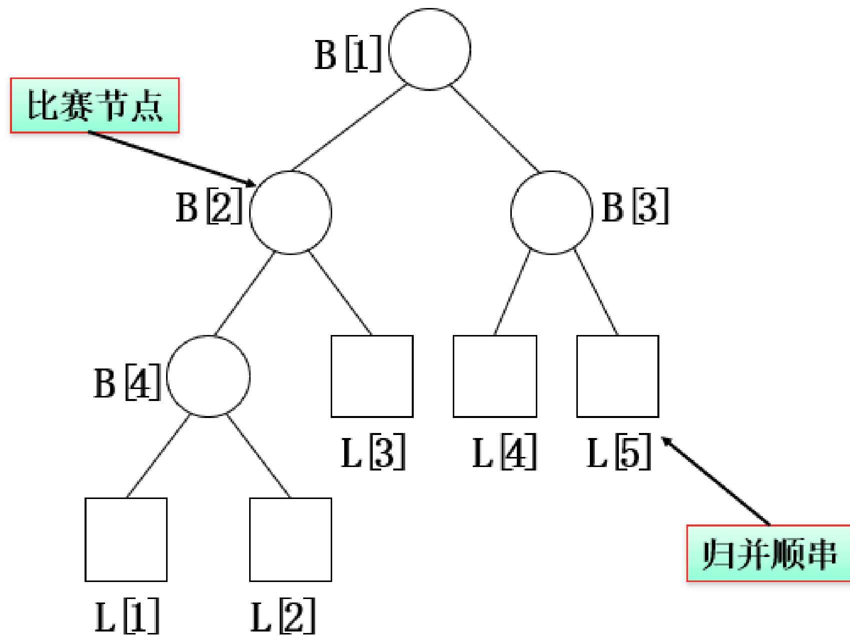
赢者树

➢ 用**完全二叉树**作为存储结构

- 叶结点用数组L[1…n]表示，内部结点用数组B[1…n-1]表示
- 数组B中实际存放的是数组L的索引

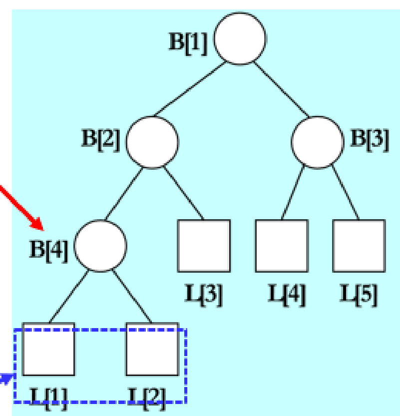
赢者树的结构

n路归并，赢者树具有2n-1个节点



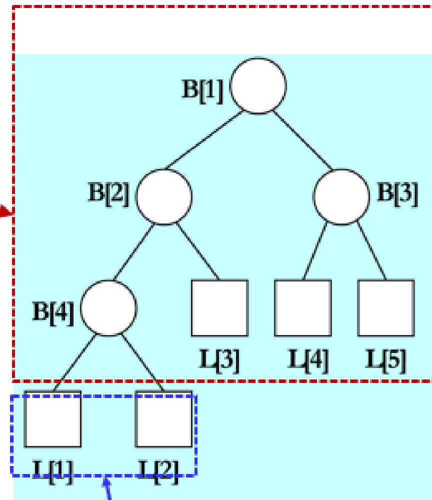
外部结点L[i]与内部父结点B[p]关系

- 外部结点数为n，内部节点数为n-1，比赛树深度s=$\lceil \log_2 n \rceil - 1$
  - 如右图，比赛树（只考虑内部节点）深度为 s=$\lceil \log_2 5 \rceil - 1 = 2$
- 最底层最左端内部结点编号为2^s
  - 如右图，B[4]= 2^2=4
- 最底层的内部结点数为n-2^s
  - 如右图，n-2^s=5-4=1
- 最底层的外部结点数（ LowExt ）为最底层内部结点数的2倍
  - 即：LowExt = 2*(n-2^s)=2*1=2



外部结点L[i]与内部父结点B[p]关系

- 最底层外部结点之上的所有(内+外)结点数目（offset），offset = 2^(s+1)-1
  - 如右图，offset=2^(2+1)-1=7
- 外部结点L[i]与内部父结点B[p]关系可以表示如下：



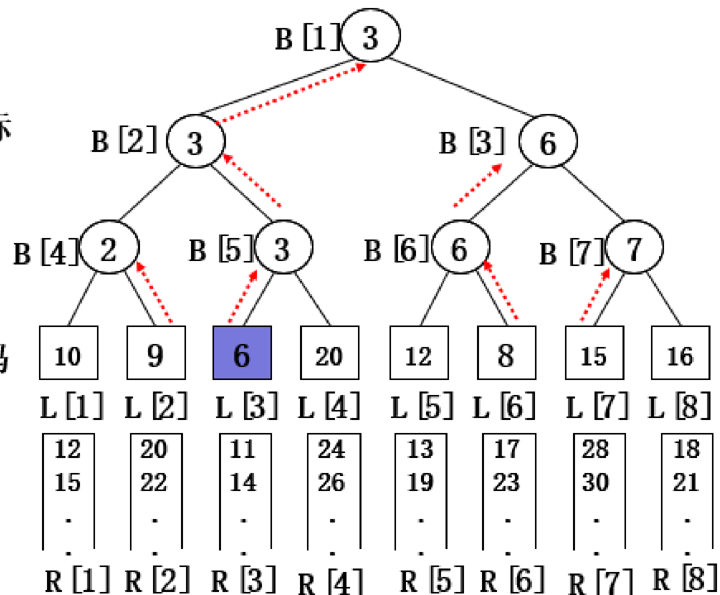
$$p=\begin{cases}(i+offset)/2, & i\le LowExt\\ (i-LowExt+n-1)/2, & i>LowExt\end{cases}$$

赢者树的示例

数组B中存储的是L的下标

当前进行比较的8个关键码

等待进行比较的8个顺串

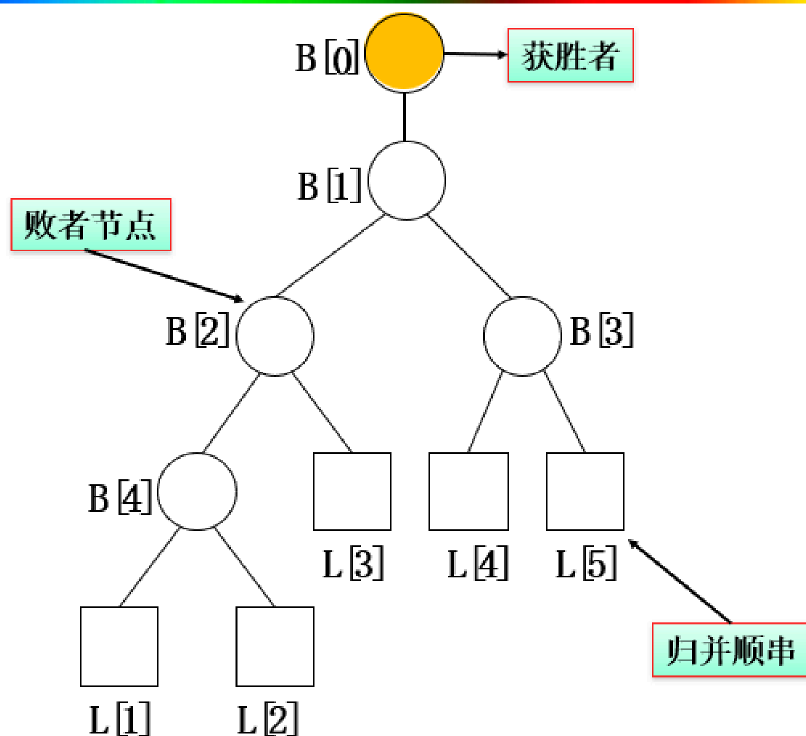


OfficePLUS 字体 保存

赢者树的特点

- 通过比较两个选手的分数确定一场比赛的赢家
  - 从树最底层开始，每两个树叶之间进行比赛，输的选手被淘汰，赢的继续向上进行竞赛，树根记录了整个比赛的胜者。
- 如果选手L[i]的分值改变，可以修改这棵赢者树
  - 沿着从L[i]到根结点的路径，和兄弟节点的值进行比较，根据比赛结果修改二叉树节点的值（赢着上），而不必改变树其他部分的比赛结果

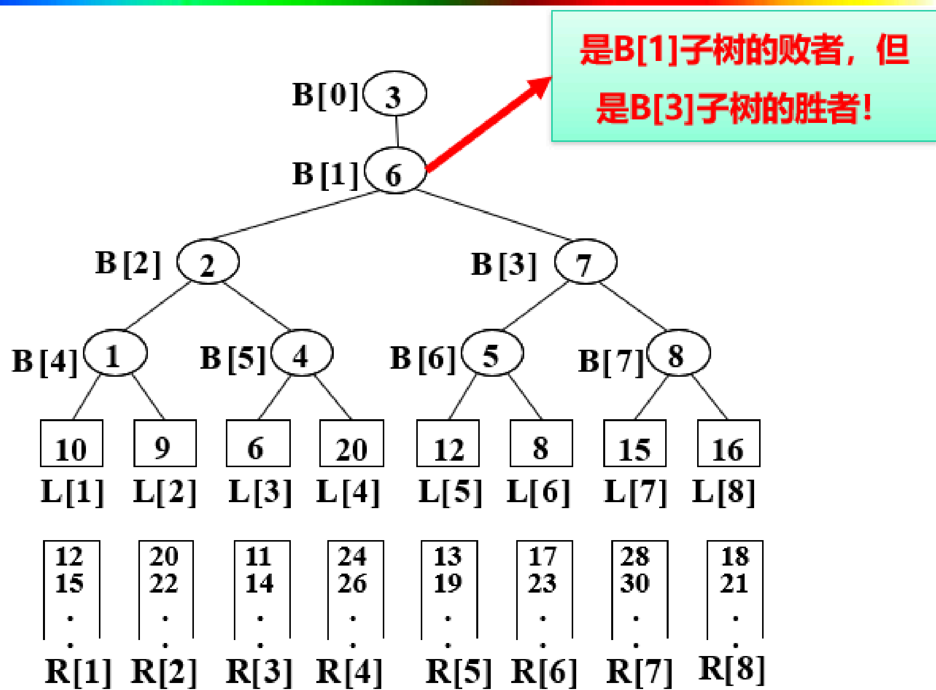
败者树



比赛过程

- 将新进入树的结点与其父结点进行比赛
  - 把败者存放在父结点中
  - 而把胜者再与上一级的父结点进行比赛
- 这样的比赛不断进行，直到结点B[1]
  - 把败者的索引放在结点B[1]
  - 把胜者的索引放到结点B[0]

败者树的示例



败者树算法

```
template<class T>
class LoserTree{
   private:
       int MaxSize;    // 最大选手数
       int n;          // 当前选手数
       int LowExt;     // 最底层外部结点数
       int offset;     // 最底层外部结点之上的结点总数
       int *B;         // 败者树数组，实际存放的是下标
       T *L;           // 元素数组
       // 在内部结点从右分支向上比赛
       void Play(int p, int lc, int rc, int(*winner)(T A[], int
       b, int c), int(*loser)(T A[], int b, int c));

   public:
       LoserTree(int Treesize = MAX);
       ~LoserTree(){delete [] B;}
       // 初始化败者树
       void Initialize(T A[], int size,int (*winner)(T A[], int b,
       int c), int(*loser)(T A[], int b, int c));
       int Winner();                  // 返回最终胜者索引
       // 重构败者树
       void RePlay(int i, int(*winner)(T A[], int b, int c), int
       (*loser)(T A[], int b, int c));
};
```

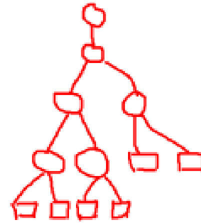
```
template<class T>
void LoserTree<T>::Initialize(T A[], int size, int(*winner)(T A[],
   int b, int c), int(*loser)(T A[], int b, int c))  {
        int i,s;
        n = size;                              // 初始化成员变量
        L = A;
        for (s = 1; 2*s <= n-1; s += s);       //计算倒数第2层节点数
        LowExt = 2*(n-s);
        offset = 2*s-1;
```

$$p = \begin{cases} (i + offset)/2, & i \le LowExt \\ (i - LowExt + n - 1)/2, & i > LowExt \end{cases}$$

```
        for (i = 2; i <= LowExt; i += 2) // 最底层外部结点比赛
                Play((offset+i)/2, i-1, i, winner, loser);
        // 处理其余外部结点
        if (n%2) {  //n为奇数，内部结点和外部结点比一次
                 // 暂存在父中的左胜者与外部右子结点比
                Play(n/2, B[n], LowExt+1, winner, loser);
                i = LowExt+3;
        }
        else  i = LowExt+2;
        for (; i <= n; i += 2)      // 剩余外部结点的比赛
                Play((i-LowExt+n-1)/2, i-1, i, winner, loser);
}
```

```
for (i = 2; i <= LowExt; i += 2) // 最底层外部结点比赛
        Play((offset+i)/2, i-1, i, winner, loser);
// 处理其余外部结点
if (n%2) {  //n为奇数，内部结点和外部结点比一次
        // 暂存在父中的左胜者与外部右子结点比
        Play(n/2, B[n], LowExt+1, winner, loser);
        i = LowExt+3;
}
else  i = LowExt+2;
for (; i <= n; i += 2)     // 剩余外部结点的比赛
        Play((i-LowExt+n-1)/2, i-1, i, winner, loser);
```

$$p = \begin{cases} (i + offset)/2, & i \le LowExt \\ (i - LowExt + n - 1)/2, & i > LowExt \end{cases}$$



```
template<class T>
void LoserTree<T>::PLAY(int p, int lc, int rc, int(*winner)(T A[], int b, int c),
    int(*loser)(T A[], int b, int c))  {
        B[p] = loser(L, lc, rc);                        //败者索引放在B[p]中
        int temp1, temp2;
        temp1 = winner(L, lc, rc);
        while (p>1 && p%2) {                            //p为奇数向上比赛
                temp2 = winner(L, temp1, B[p/2]);       //胜者与父比较
                B[p/2] = loser(L, temp1, B[p/2]);       //败者留下
                temp1 = temp2;                          //胜者放入temp1
                p/=2;                                   //p向上指向父结点
        }// while
        B[p/2] = temp1;                                 //B[p]是左孩子或者p=1
}
```

(3) 重构败者树

```
void LoserTree<T>::RePlay(int i, int (*winner)(T A[], int b, int c),
    int (*loser)(T A[], int b, int c)) {
        int p;                      //用于计算父结点索引的临时变量
        if (i <= LowExt)        //确定父结点的位置
                p = (i+offset)/2;
        else  p = (i-LowExt+n-1)/2;
        B[0] = winner(L, i, B[p]);  // B[0]中保存胜者的索引
        B[p] = loser(L, i, B[p]);     // B[p]中保存败者的索引
```

```
      for (; (p/2) >= 1; p/=2) {  // 沿路径向上比赛
            int temp;     // 临时存放赢者的索引
            temp = winner(L,B[p/2], B[0]);
            B[p/2] = loser(L,B[p/2], B[0]);
            B[0] = temp;
      }
}
```

多路归并的效率

假设对k个顺串进行归并

- **原始方法：找到每一个最小值的时间是Θ(k)，产生一个大小为n的顺串的总时间是Θ (k·n)**
- **败方树方法**
  - **初始化包含k个选手的败方树需要Θ (k)的时间**
  - **读入一个新值并重构败方树的时间为Θ (*log* k)**
  - **故产生一大小为n的顺串的总时间为Θ (k+n·*log* k)**

检索

平均检索长度

$$ASL=\sum_{i=1}^{n}P_iC_i(P_i\text{为检索第}i\text{个元素的概率，}C_i\text{为找到第}i\text{个元素所需的比较次数})$$

顺序检索

- 检索成功
  - 假设检索每个关键码是等概率的：$P_i = 1/n$

$$ASL_S=\sum_{i=1}^{n}P_i*(n-i+1)=\frac{1}{n}*\sum_{i=1}^{n}(n-i+1)$$

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

- 检索失败：设置了一个监视哨

$$ASL_F=n+1$$

二分检索

前提条件：待检索序列有序

```
template  <class Type> int BinSearch (vector<Item<Type>*>&
    dataList, int length, Type k){
    int low=1, high=length, mid;
    while (low<=high)  {                       //结束条件！！
        mid = (low+high)/2;
        if (k<dataList[mid]->getKey())
              high = mid-1;                        //右缩检索区间
        else if (k>dataList[mid]->getKey())
              low = mid+1;                         //左缩检索区间
        else return mid;                           //成功返回位置
     }
    return 0; //检索失败，返回0
}   //为与顺序检索保持一致，位置0不存放实际元素；
```

散列表的检索

重要概念

- 负载（或者装填）因子 α=n/M
  - n：散列表中已有结点数
  - M：散列表空间大小
- 冲突
  - 将不同的关键码映射到相同的散列地址
  - 实际应用中，不产生冲突的散列函数极少存在
- 同义词
  - 发生冲突的两个关键码

散列函数

- 散列函数：把关键码映射到存储位置的函数，通常用 h 来表示

$$Address = Hash(key)$$

除余法

- 用关键码除以M(往往取散列表长度)，并取余数作为散列地址。散列函数为：

$$h(x) = x \bmod M$$

- 通常选择质数作为M值
  - 函数值依赖于变量x的所有位，而不是某些位，增大了均匀分布的可能性

注意：M不能取偶数

- 若把M设置为偶数
  - x是偶数，h(x)也是偶数
  - x是奇数，h(x)也是奇数
- 缺点：分布不均匀
  - 如果偶数关键码比奇数关键码出现的概率大，那么函数值就不能均匀分布
  - 反之亦然

平方取中法

- 先通过求关键码的平方来扩大差别，再取其中的几位或其组合作为散列地址
- 例如，
  - 一组二进制关键码：(00000100，00000110，000001010，000001001，000000111)
  - 平方结果为：(00010000，00100100，01100010，01010001，00110001)
  - 若表长为4个二进制位，则可取中间四位作为散列地址：(0100，1001，1000，0100，1100)

折叠法

- 基本思想
  - 将关键码分割成位数相同的几部分
    - ✓ 最后一部分的位数可以不同
  - 然后取这几部分的叠加和（舍去进位）作为散列地址

- 两种叠加方法
  - 移位叠加：把各部分的最后一位对齐相加
  - 分界叠加：沿各部分的分界来回折叠，然后对齐相加，将相加的结果当做散列地址

冲突解决

一．开散列方法（也称拉链法）

- 所有同义词链接在同一链表
- α可以大于1，但一般取α≤1

二．闭散列方法（也称开地址法）

- 把发生冲突的关键码存储在散列表中另一个空地址内

开散列方法

拉链法

➢ 把散列表中的每个槽定义为一个链表的表头，散列到特定槽的所有记录都放到这个槽的链表中

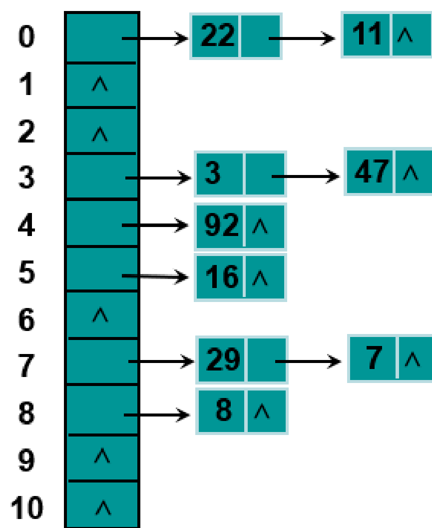
例子：关键码集合：

{47, 7, 29, 11, 16, 92, 22, 8, 3}

散列函数：

$H(key) = key \bmod 11$

查找长度分析



- **成功**

$$ASL_{succ}=\frac{1*6+2*3}{9}=\frac{12}{9}=\frac{4}{3}$$

- **失败**

$$ASL_{unsucc}=\frac{1}{11}(3+1+1+3+2+2+1+3+2+1+1)=\frac{20}{11}$$

桶式散列

检索访问

- 计算H(i)的值，然后调桶目录表中包含第i个桶目录的页块进入内存，查到第i个存储桶的第一个页块的地址，然后根据该地址调入相应页块

磁盘访问性能

- 调存储桶目录表进入内存（设不在内存）需进行一次访外
- 逐个检查桶内各页块，则平均访外次数为桶内页块数一半
- 对于修改、插入等其他运算尚需另1次访外写外存。

闭散列方法

- $d_0=h(K)$称为K的基地址
- 当冲突发生时，使用某种方法为关键码K生成一个候选的散列地址序列，称为探查序列

  $d_1, d_2, \ldots d_i, \ldots d_{M-1}$
- $d_i = d_0 + p(K, i)$ $(0<i<M)$是后继散列地址，p(K，i)是探查函数

- 检索要遵循插入时同样的探查序列
  - 重复冲突解决过程
  - 找出在基位置没有找到的记录
- 插入和检索函数都假定每个关键码的探查序列中至少有一个存储位置是空的
  - 否则可能会进入一个无限循环中
  - 也可以限制探查序列长度

线性探测

- $d_i = d_0 + p(K, i)$ (0

- **基本思想**
  - **如果记录的基位置存储位置被占用，那么就在表中下移，直到找到一个空存储位置**
    - ✓**探查序列：d+1, d+2, ……, M-1, 0, 1, ……, d-1**
  - **用于简单线性探查的探查函数是：p(K，i) = i**
- **优点**
  - **表中所有的存储位置都可作为插入记录的候选**

聚集示例

- 在理想情况下，表中的每个空槽都应该有相同的机会接收下一个要插入的记录。
  - 下一条记录放在第11个槽中的概率是2/13
  - 放到第7个槽中的概率是9/13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
26	25	41	15	68	44	6				36		38	12	51

平均查找性能分析

- **成功查找的ASLsucc**

$$ASL_{succ} = \frac{1}{11}\sum_{i=1}^{11} C_i = \frac{1}{11}(1*6+2+2+2+3+5) = \frac{20}{11}$$

- **失败查找的ASLunsucc**

$$ASL_{unsucc} = \frac{8+7+6+5+4+3+2+1+1+1+2+1+11}{13} = \frac{52}{13} = 4$$

改进线性探查

- 每次跳过常数c个而不是1个槽
  - 探查序列中的第i个槽是(h(K) + ic) mod M
  - 基位置相邻的记录就不会进入同一个探查序列了
- 探查函数是p(K，i) = i*c

二次探查

- 探查增量序列依次为：1^2，-1^2，2^2，-2^2，…，即地址公式是

$$d_{2i-1} = (d + i^2) \% M$$

$$d_{2i} = (d - i^2) \% M$$

- 用于简单线性探查的探查函数是

$$p(K, 2i-1) = i*i$$

$$p(K, 2i) = - i*i$$

二级聚集

- **基本聚集**
  - 基地址不同的关键码，其探查序列的某些段重叠在一起
  - 伪随机探查和二次探查**可以消除基本聚集**
- **二级聚集** (secondary clustering)
  - 如果两个关键码散列到同一个基地址，还是得到同样的探查序列，所产生的聚集
  - 原因
    - ✓ 探查序列只是基地址的函数，与关键码值无关
    - ✓ 例子：伪随机探查和二次探查

双散列探查法

避免二级聚集

- 双散列探查法使用两个散列函数h_1和h_2

- 若在地址$h_1(key)=d$发生冲突，则计算$h_2(key)$，得到的探查序列为：

  $(d+h_2(key))$ % M，

  $(d+2h_2 (key))$ %M ，

  $(d+3h_2 (key))$ % M ，

  …

- 双散列函数探查法序列公式：

$$d_i=(d + i* h_2 (key)) \% M$$

- 探查函数：

$$p(K, i) = i * h_2 (key)$$

双散列函数法特征

- h_2 (key) 必须与M互素
  - 使发生冲突的同义词地址均匀地分布在整个表中
  - 否则可能造成同义词地址的循环计算
- 双散列的优点：不易产生“聚集”
- 缺点：计算量增大

散列表的插入

散列函数h，假设给定的值为K

- 若表中基地址空间未被占用，则插入记录
- 若表中基地址的值与K相等，则报告“已有此记录”
  - 不允许重复记录存在！
- 否则，按设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
  - 直到某个地址空间未被占用（可以插入）
  - 或者关键码比较相等（不需要插入）为止

插入算法代码

```
bool HashInsert(const Elem& e) {
    int home= h(getkey(e));                 //home存储基位置
    int i=0;
    int pos = home;                         //探查序列的初始位置
    while (!eq(EMPTY, HT[pos])) {
        if (eq(e, HT[pos])) return false;   //若插入值e存在
        i++;
        pos = (home+p(getkey(e), i)) % M;   //下一探查地址
    }
    HT[pos] = e;                            // 插入元素e
    return true;
}
```

散列表的检索

2、散列表的检索

假设散列函数h，给定的值为K

- 若基地址空间未被占用，则检索失败
- 否则将该地址中的值与K比较，若相等则检索成功
- 否则，按建表时设定的处理冲突方法查找探查序列的下一个地址，如此反复下去
  - 关键码比较相等，检索成功
  - 地址空间未被占用，检索失败

```
bool HashSearch(const Key& K, Elem& e) const{
  int i=0, pos= home= h(K);              // 初始位置
  while (! eq(EMPTY, HT[pos])) {
    if (eq(K, HT[pos])) {                // 找到
        e = HT[pos];
        return true;
    }
    i++;
    pos = (home + p(K, i)) % M;          // 探查序列中的下一地址
  }
  return false;
}
```

散列表的删除

- 删除记录的时候，有两点需要重点考虑：
  - (1) 删除记录不能影响后续检索
  - (2) 释放存储位置能为将来所用
- 只有开散列方法可以真正删除，空间重用
- 闭散列方法都只能作标记，**不能真正删除，空间未再次分配之前不可用**

墓碑

- 设置一特殊的标记位，来记录散列表中的单元状态
  - **单元被占用、空单元、已删除**
- 是否可以把**空单元、已删除**这两种状态，用统一的标记，以区别于“单元被占用”状态？
  - 不可以！
  - **检索时遇到已删除标记还需要继续检索下去**
  - 增加了平均检索长度
- 被删除标记值称为**墓碑**(tombstone)
  - 标志一个记录曾经占用这个槽，现在已经不再占用了

```
Elem hashDelete(const Key& K){
    int i=0, pos = home= h(K);                    //初始位置
     while (!eq(EMPTY, HT[pos])) {
         if (eq(K, HT[pos])){
                 temp = HT[pos];
                 HT[pos] = TOMB;                  //设置墓碑
                 Return temp;                     //返回目标
         }
         i++;
         pos = (home + p(K, i)) % M;
     }
   Return EMPTY;
}
```

- 在插入时，如果遇到标志为墓碑的槽，可以把新记录存储在该槽中吗?
  - 避免插入两个相同的关键码
  - 检索过程仍然需要沿着探查序列下去，直到找到一个真正的空位置

带墓碑的插入操作改进

```
bool HashInsert(const Elem &e){
    int insplace, i=0, pos=home= h(getkey(e)); bool tomb_pos=false;
    while (!eq(EMPTY, HT[pos])) {
        if (eq(e, HT[pos])) return false;              //出现相同值元素!
        if (eq(TOMB, HT[pos]) &&  !tomb_pos) {
            insplace=pos;  tomb_pos=true;
        }                                              //记下第1个墓碑!
        pos = (home + p(getkey(e), ++ i)) % M;
    }
    if (!tomb_pos) insplace=pos;                       //没有墓碑
    HT[insplace]=e;    return true;
}
```

10.3.5 散列方法的效率分析

- 衡量标准：插入、删除和检索操作所需的ASL
- 散列表的插入和删除操作都是基于检索进行的
  - 删除：必须先找到该记录
  - 插入：必须找到探查序列的尾部，即对这条记录进行一次不成功的检索
    - ✓不考虑墓碑的情况，是尾部的空槽
    - ✓考虑墓碑的情况，也要找到尾部，才能确定是否有重复记录

影响检索的效率的重要因素

- 散列效率与负载因子 $\alpha = N/M$ 有关
  - α 较小时，散列表比较空，所插入的记录比较容易插入到其空闲的基地址
  - α 较大时，插入记录很可能要靠冲突解决策略来寻找探查序列中合适的另一个槽
- 随着 α 增加，越来越多的记录有可能放到离其基地址更远的地方

结论1

- 散列方法代价一般接近于访问一个记录的时间，效率高，比需要log n次记录访问的二分检索好得多
  - 不依赖于n，只依赖于负载因子$\alpha=n/M$
  - 随着α增加，预期的代价也会增加
  - $\alpha \le 0.5$时，大部分操作的分析预期代价都小于2
- 经验表明，负载因子的临界值是0.5（将近半满）
  - 大于这个临界值，性能就会急剧下降

结论2

- 散列表的插入和删除操作如果很频繁，将降低散列表的检索效率
  - 大量的插入操作，将使得负载因子增加
    - ✓从而增加了同义词子表的长度
    - ✓也就是增加了平均检索长度
  - 大量的删除操作，也将增加墓碑的数量
    - ✓这将增加记录本身到其基地址的平均长度

结论3

- 实际应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重新散列
  - 把所有记录重新插入到一个新的表中
    - ✓清除墓碑
    - ✓把最频繁访问的记录放到其基地址

108

索引

主码

- 主码(primary key)是数据库中的每条记录的唯一标识
  - 例如，身份证号码
  - 如果只有主码，不便于各种灵活检索

辅码

- 辅码：数据库中可出现重复值的码（属性）
  - 如姓名，性别等
- 辅码索引把一个辅码值与具有这个辅码值的多条记录的主码值关联起来
  - 多数检索都是利用辅码索引完成的

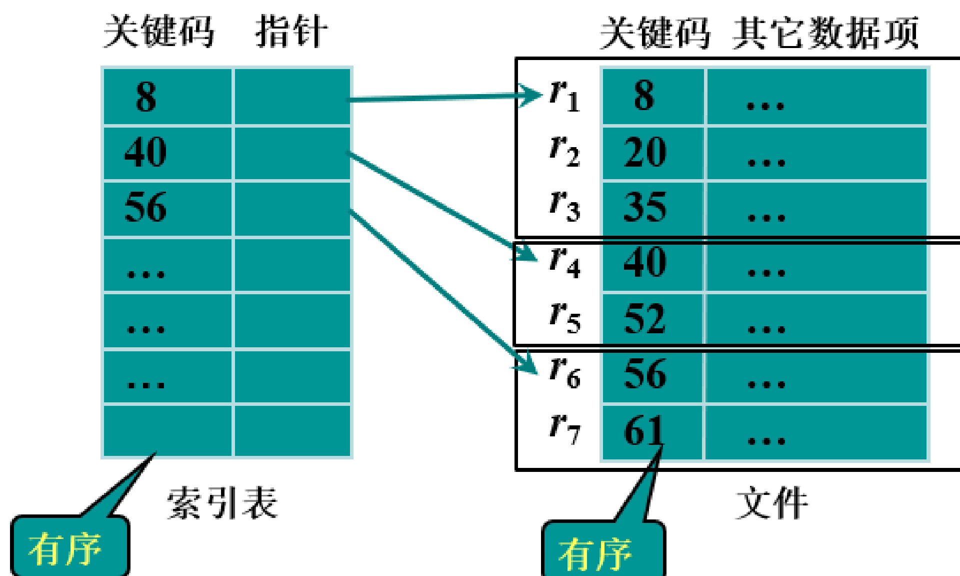
- 索引
  - 把关键码与它对应的记录位置关联起来的过程
  - (关键码，指针)对，即(key, pointer)
  - 指针指向主数据库文件（或“主文件”）中的完整记录
- 索引文件：用于记录这种联系的文件
- 索引技术是组织大型数据库的一项重要技术
  - 高效率的检索
  - 支持动态的数据插入、删除和更新

索引文件

- 原始数据记录组成的文件称为**主文件**
- 索引数据组成的文件称为**索引文件**
- 一个主文件可能有多个相关索引文件
  - 每个索引文件往往支持一个索引字段
  - 不需要重新排列主文件
- 通过索引文件高效访问记录中的关键码

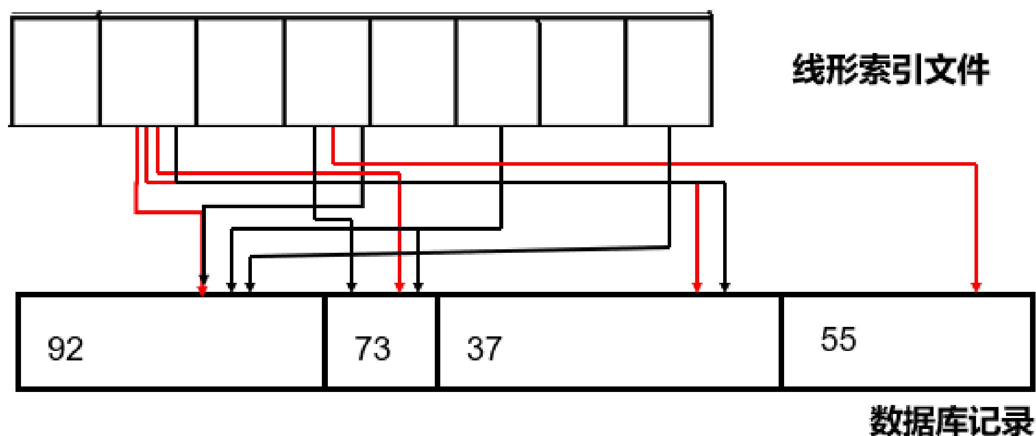
- 稀疏索引
- 稠密索引

稀疏索引示例



线性索引

- 按照索引码值的顺序进行排序的文件
- 索引文件中的指针指向存储在磁盘上的文件记录起始位置或者主索引中主码的起始位置



线性索引优缺点

- 可对变长的数据库记录访问
- 支持对数据的高效检索
  - 二分检索
- 线性索引太大，存储在磁盘中
  - 一次检索可能多次访问磁盘，影响检索的效率
  - 使用二级线性索引

二级线性索引

- 设磁盘块大小是1024字节，每对(关键码，指针)索引对需要8个字节
  - 1024 / 8 = 128
  - 每磁盘块可以存储128条索引对
- 假设数据文件包含10000条记录
  - 稠密一级线性索引中包含10000条记录
    - ✓10000/128 = 78.1
    - ✓那么一级线性索引占用79个磁盘块
  - 相应地，二级线性索引文件中有79项索引对
  - 这个二级线性索引文件可以在一个磁盘块

二级线性索引的例子

- 关键码与相应磁盘块中第一条记录的关键码值相同
- 指针指向相应磁盘块的起始位置

二级索引 1 2003 5744 10723 ……

一级索引 1…… 2002 2003 5583 5744 9297 10723 13293
……
磁盘块

例如：检索关键码为2555的记录

关键字2555的记录指针

二级索引

1	2003	5744	10723	……

线性索引

1　2002	2003　5583	5744　9297	10723　13293
……			

磁盘块

关键码为2555的记录

1. 二级线性索引文件读入内存
2. 二分法找关键码的值小于等于2555的最大关键码所在一级索引磁盘块地址——关键码为2003的记录
3. 根据记录2003中的地址指针找到其对应的一级线性索引文件的磁盘块，并把该块读入内存
4. 按照二分法对该块进行检索，找到所记录在磁盘上的位置
5. 最后把所需记录读入，完成检索操作

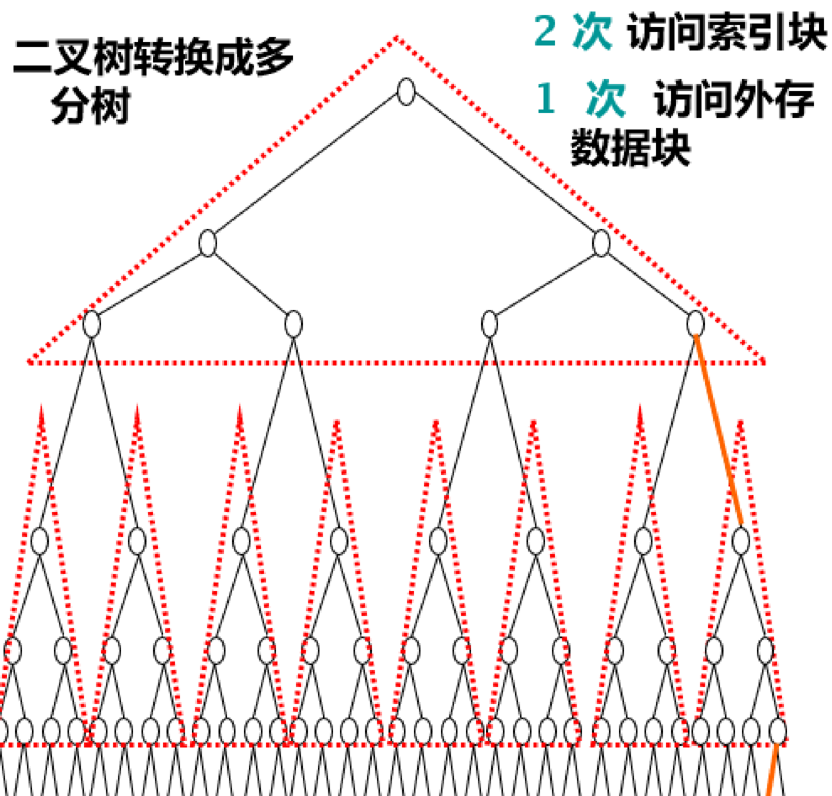
静态索引

11.2 静态索引

- 索引结构在文件创建时生成
- 一旦生成就固定下来，在系统运行(例如插入和删除记录)过程中，索引结构不做改变
- 只有当文件再组织时才允许改变索引结构

多分树

- **组织索引一般不用二叉树而采用多分树**
- **大大减少访问外存的次数**



倒排索引

- 按属性值建立索引，索引表中的每一项包括
  - 一个属性值和具有该属性值的各关键码或者记录地址
    - ✓关键码对应的指针地址
- 由属性值来确定记录的位置，称之为**倒排索引**
- 带有倒排索引的文件称为**倒排文件**
- 分类
  - 基于属性的倒排
  - 对正文文件的倒排

- 按属性值建立起来索引表，称**倒排表**
  - (attr，ptrList)
    - ✓**属性值**
    - ✓**记录指针**：可以是关键码，或该记录的主文件地址
- 倒排文件：带有倒排索引的文件

教师数据库主表

EMP#	NAME	Department	Profession	Specialty	Address
0155	李宇	数学	教授	代数	C105
0421	刘阳	外语	助教	英语	E310
0208	赵亮	物理	助教	力学	C211
0211	张伟	物理	讲师	原子物理	D508
0132	王亮	数学	助教	几何	E220
0119	王卓	数学	讲师	代数	B102
0330	孙丽	计算机	教授	软件	A108
0455	刘珍	外语	讲师	法语	A225
0310	周兵	计算机	讲师	英语	B423
0341	何江	计算机	助教	计算机	F406
……					

教师数据库倒排表

Department list	EMP#
数学 物理 计算机 外语	0155, 0132,0119 0208, 0211 0330, 0310,0341 0421, 0455
Profession list	**EMP#**
教授 讲师 助教	0155, 0330 0211, 0119, 0455, 0310 0421, 0208, 0132, 0341
Specialty list	**EMP#**
代数 几何 力学 原子物理 软件 英语 法语	0155, 0119 0132 0208 0211 0330, 0341 0421, 0310 0455

优缺点

- 优点
  - 支持基于属性的高效检索
- 缺点
  - 花费了保存倒排表的存储代价
  - 降低了更新运算的效率

11.3.2 对正文文件的倒排

- 正文索引(Text Indexing)
  - 支持对文本内容的快速检索
- 方法
  - 词索引(word index)
  - 全文索引(full-text index)

词索引

基本思想

- 从正文中抽取出关键词，然后用这些关键词组成适合快速检索的数据结构

支持多种文本类型

- 适用于英文
- 中文等东方文字要经过“切词”处理

全文索引

- **基本思想**
  - 正文看作一个长的字符串，记录每个字符串的开始位置，查询可以针对正文中的任何字符串进行
- 可以对每个字符串建立索引，从而使查询词不再限于关键词
- 索引结构将耗费更大的存储空间

词索引应用广泛

- **一个已经排过序的关键词列表**
  - **其中每个关键词指向一个倒排表（posting list）**
    - ✓**指向该关键词出现文档集合**
    - ✓**在文档中的位置**

文档编号	文本内容
1	Pease porridge hot, please porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

文档编号	文本内容
1	Pease porridge hot, please porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

倒排索引

类似处理1中的其他词语

同理，处理2中的词语

依次处理所有文档

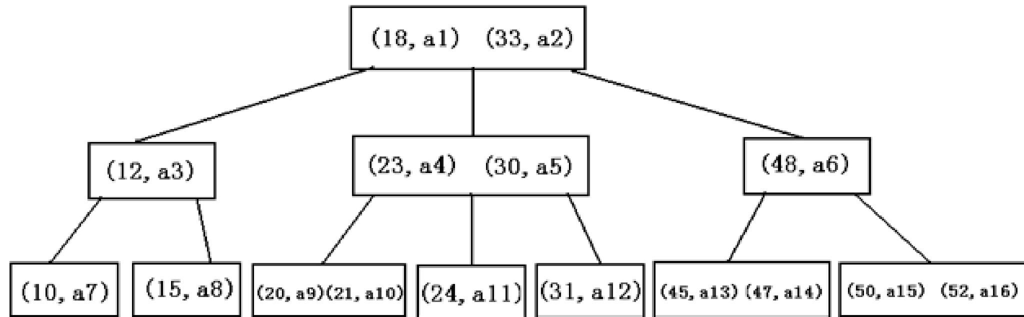
编号	词语	（文档编号，位置）
1	cold	(1,6)
2	days	
3	hot	(1,3)
4	in	
5	it	
6	like	
7	nine	
8	old	
9	pease	(1,1) (1,4) （2,1）
10	porridge	(1,2) (1,5)
11	pot	
12	some	
13	the	

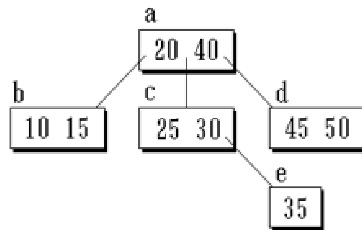
B树

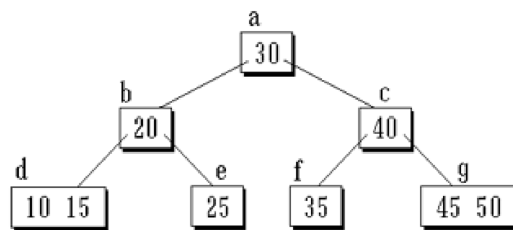
11.4.1 m阶B树的结构定义

- *m* 阶B-树是一棵 *m* 路查找树，或者为空，或者：
  - 上界：树中每个结点至多有*m*个子结点
  - 下界：根结点至少有2棵子树，其它非叶结点至少有$\lceil m/2 \rceil$棵子树
  - 有k棵子树的结点有k-1个关键码
  - 叶结点都位于同一层，有$\lceil m/2 \rceil$-1到m-1个关键码

示例



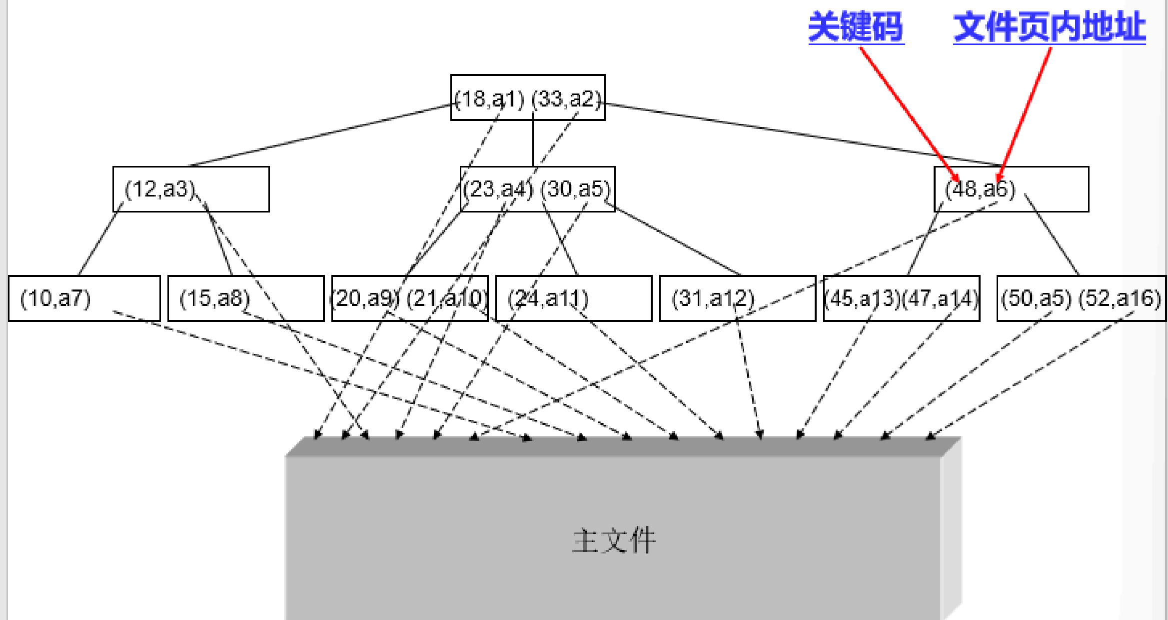




非B树

B树

每个关键码对应一个记录指针域



B树的查找

- 交替的两步过程
  1. 读取根结点，在所包含的关键码K_1，…，K_j中查找给定的关键码
     - ✓ 找到则检索成功
  2. 否则，确定要查的关键码值是在某个K_i和K_{i+1}之间，于是取p_i所指向的结点继续查找
- 如果p_i指向外部空结点，表示检索失败

B树的检索长度

- 设B树的高度为h
  - 独根树高为1
- 在自顶向下检索到叶结点的过程中可能需要进行 h 次读盘
- 最多需要h+1次访外

B树的插入过程

- 结构要调整，性质要保持（等高和阶）
  - 1) 找到最底层插入
  - 2) 若溢出，则结点分裂，中间关键码连同新指针插入父结点
  - 3) 若父结点也溢出，则继续分裂
    - ✓ 分裂过程可能传达到根结点(则树升高一层)

具体操作

- B树是从空树起，逐个插入关键码而生成的
- B树的每个内部结点的关键码个数都在$[\lceil m/2 \rceil -1, m-1]$之间
- 插入在某个叶结点开始
  - 如果在关键码插入后结点中关键码数超出上界 m-1，则结点“分裂”
  - 否则可以直接插入
- 实现结点“分裂”的原则
  - 设结点 A 中已有m-1 个关键码，当再插入一个关键码后结点的状态为

$$(m, A_0, K_1, A_1, K_2, A_2, \ldots\ldots, K_m, A_m)$$

其中 $K_i < K_{i+1}, 1 \le i < m$

结点分裂方法

- 分裂时从叶结点开始，往树根方向生长
- 结点p分裂成两个结点 p 和 q，它们包含的信息分别为：
  - 结点 p

$$(\lceil m/2 \rceil -1, A_0, K_1, A_1, \ldots\ldots, K_{\lceil m/2 \rceil -1}, A_{\lceil m/2 \rceil -1})$$

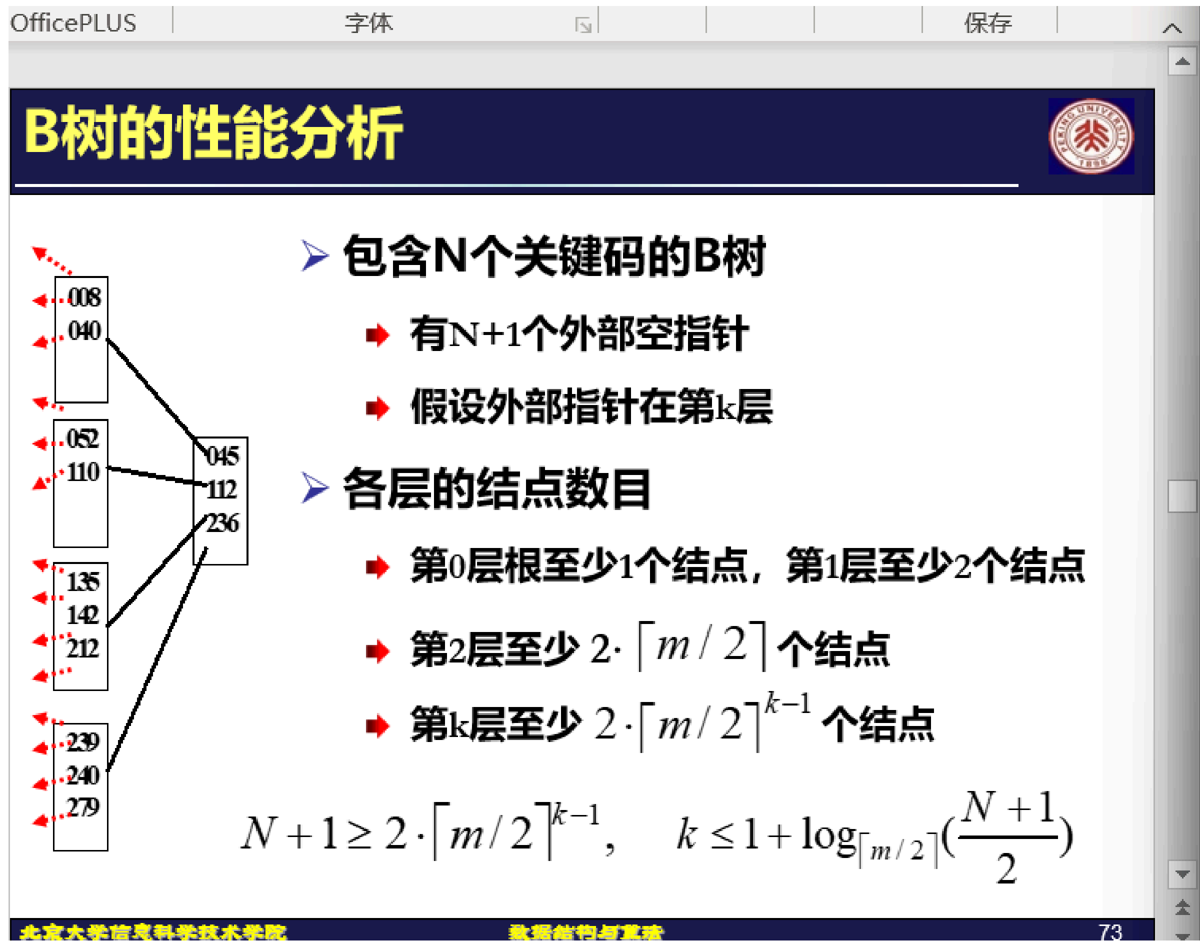
  - 结点 q

$$(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil +1}, A_{\lceil m/2 \rceil +1}, \ldots\ldots, K_m, A_m)$$

- 位于中间的关键码 $K_{\lceil m/2 \rceil}$ 与指向新结点 q 的指针形成一个二元组 ($K_{\lceil m/2 \rceil}$, q)，插入到这两个结点的双亲结点中去

插入看ppt

效率分析

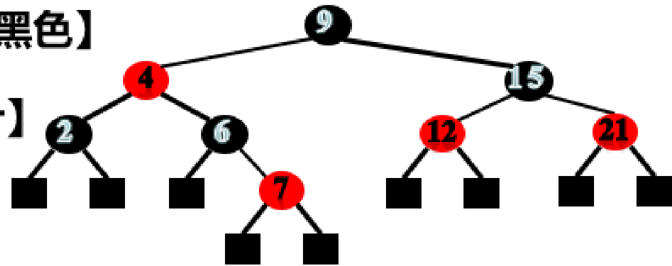


- 检索效率
  - 存取次数

$$k \leq 1+\log_{\lceil \frac{m}{2} \rceil}\left(\frac{N+1}{2}\right)$$

- N=1,999,998，m=199时，一次检索最多4层。

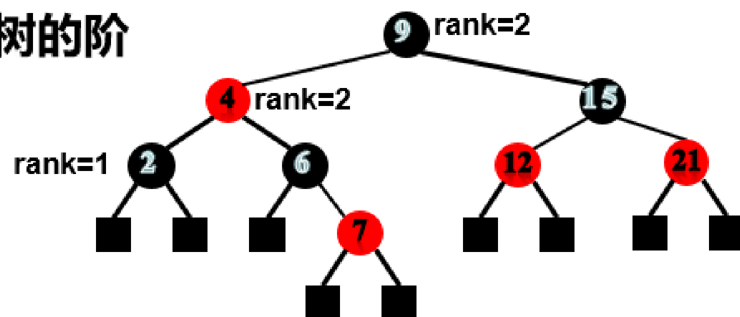
- 满足下列条件的BST树（要满足BST的约束条件）
  1. 颜色两态：【红色 or 黑色】
  2. 树根为黑：【根和树叶】
  3. 树叶也为黑
  4. 红红限制：父子节点不允许红红连续
  5. 路径上黑结点数目相同：任意结点到其每个叶结点包含相同数目的黑结点。



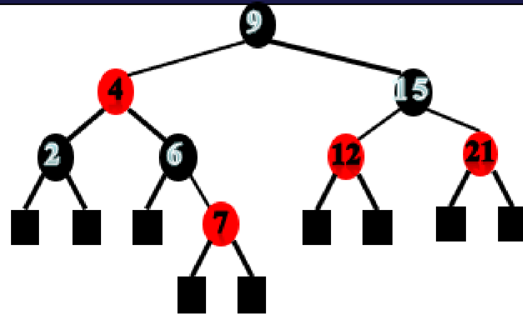
是一种扩充的BST树

红黑树的阶

- 结点X的阶（Rank，也称“黑色高度”）
  - 从该结点到外部结点的黑色结点数量
  - 不包括X结点本身，包括叶结点
- 叶结点的阶是0
- 根的阶称为该树的阶



11.6.2 红黑树的性质



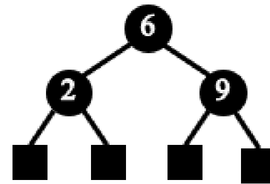
性质1：红黑树是满二叉树

- 空树叶也看作结点

性质2：k阶红黑树路径长度

- 从根到叶的简单路径长度，即介于【k，2k】
- 或者说树高介于【k+1，2k+1】之间

性质3：k阶红黑树内部结点数



- 最少时是一棵完全满二叉树
- 内部结点数最少是2^k-1

性质4：n个内部结点红黑树最大高度：$2*\log_2(n+1)+1$

证明：

设红黑树的阶为k，高为h。

由性质（2）得$h <= 2k+1$

✓ 则 $k >= (h-1) / 2$

由性质（3）得$n >= 2^k - 1$

✓ 即 $n >= 2^{(h-1)/2} - 1$

可得出 $h <= 2 \log_2 (n+1)+1$

红黑树---插入

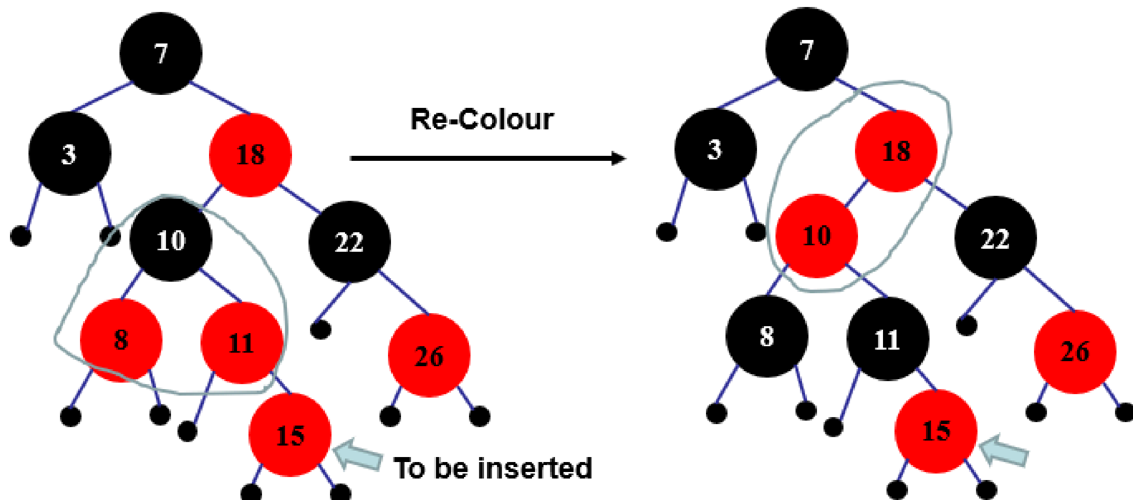
RBTree-Insert(x)

基本思路：------- Tree_Insert(X) // 利用普通二叉搜索树的方法

------- 将插入的X设置为“红色”

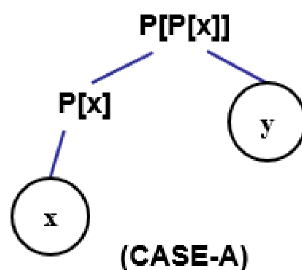
带来的问题： X的父亲也可能为红色，造成红-红冲突（违反了红黑树的第4条性质）

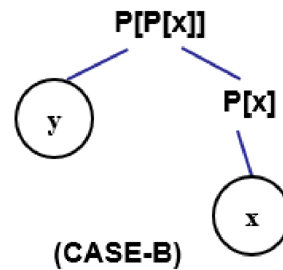
-------- 通过Rotation和Re-colour来调整



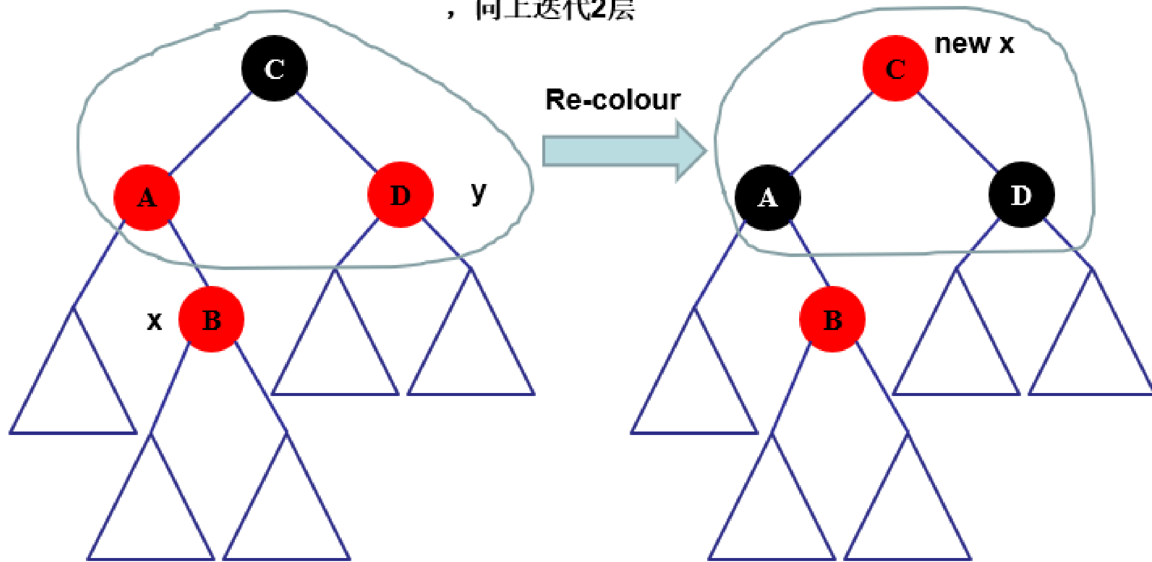
红黑树---插入

```
RB_Insert(T,X)
    Tree_Insert(T,X) // BST 插入算法
    Colour(x)← RED
    While X != root ∧ Colour[X]==RED ∧ Colour[P[X]]==RED
        Do If  P[X]==left[P[P[x]]]  // (CASE-A)
            then  y ← Right[P[P[x]]]
                  if Colour[y] == RED
                  then <Case 1>
                  else if X==right[P[X]]
                       then <Case 2>
                  else <Case 3>
            Else  // (CASE-B)
    Colour[root] ←BLACK
```

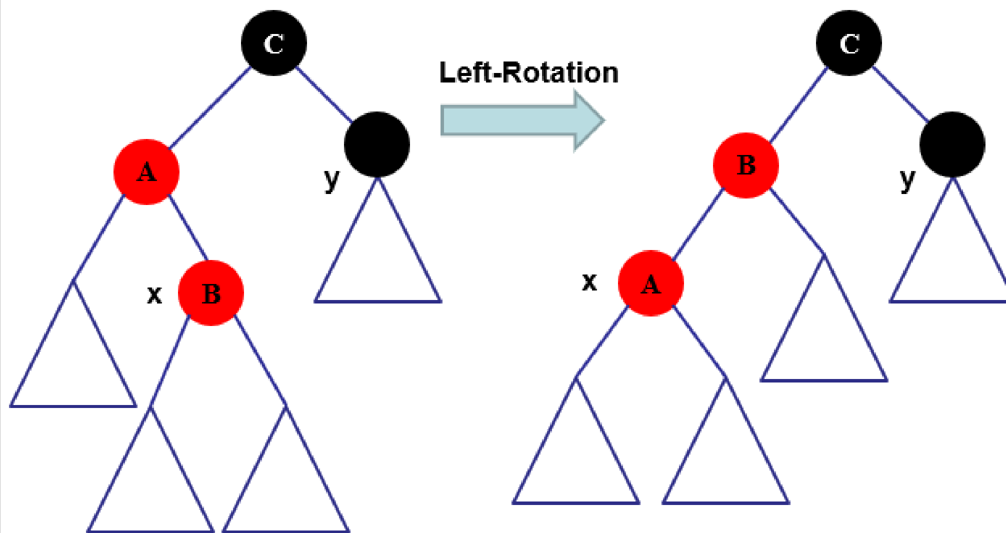




红黑树---插入



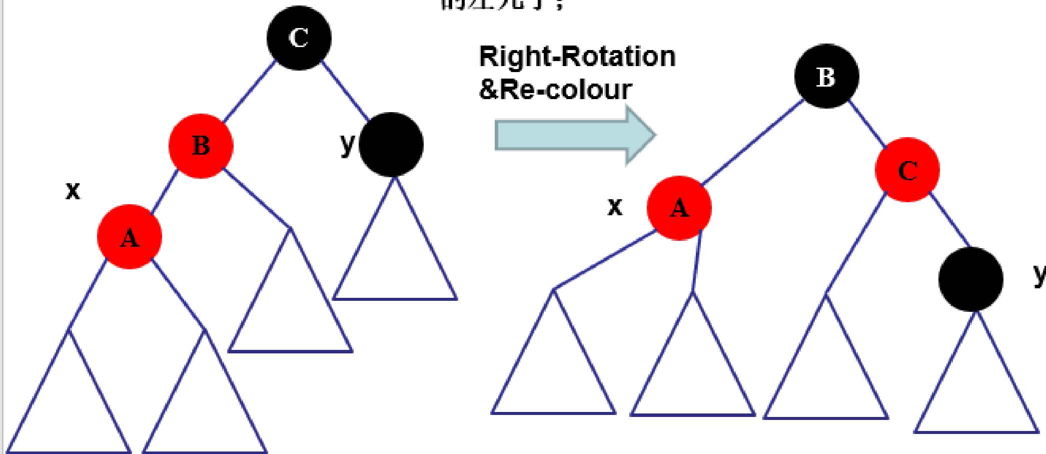
红黑树---插入



红黑树---插入

(CASE-3)

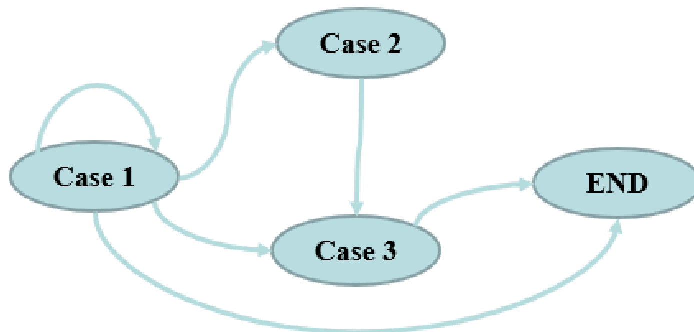
x的叔父是黑，x是P[x]的左儿子；



插入算法停止在此步

红黑树---插入

为什么红黑树的插入算法是O(Log(n))复杂度，n为树中节点的数目。



每次Case 1可以将待处理的x向上推2层，所以Case 1最多执行O(Log(n))次。

基本概念

- 数组（Array）是数量和元素类型固定的有序序列
  - 静态数组必须在定义它的时候指定其大小和类型
  - 动态数组可以在程序运行时才分配空间
- 多维数组（Multi-array）是向量的扩充
  - 向量的向量就组成了多维数组
  - 可以表示为：ELEM A[c_1..d_1][c_2..d_2]…[c_n..d_n]
    - ✓ c_i和d_i是各维下标的下界和上界。
    - ✓ 所以其元素个数为：

$$\prod_{i=1}^{n}(d_i - c_i + 1)$$

- 二维数组A_{mn}， a_{ij}的地址计算函数为：

$$LOC(a_{ij})=LOC(a_{00})+[\ i*n + j\ ]*d$$

- 三维数组A_{mnp}，a_{ijk}的地址计算函数为：

$$LOC(a_{ijk})=LOC(a_{000})+[\ i*n*p + j*p + k\ ] * d$$

特殊矩阵

- 矩阵描述为二维数组，其元素可以随机访问
- 特殊矩阵
  - 非零元素呈某种规律分布或者矩阵中有大量的零元素
  - 仍用数组存放，会造成极大浪费，尤其是高阶矩阵时
- 为节省空间，可对这类矩阵进行压缩存储

对称矩阵

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

在一个n阶方阵A中，若元素满足下述性质：$a_{ij}=a_{ji}$ $0\leq i,j\leq n-1$，则称A为对称矩阵。

特征：元素关于主对角线对称

压缩存储方法：只存矩阵中上三角或下三角中的元素。

所需空间：$$N=\sum_{i=0}^{n-1}(i+1)=\frac{n(n+1)}{2}$$

三角矩阵

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

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

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

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

- 分布特征
  - 上三角矩阵中，主对角线的下三角中的元素均为常数。常为零
  - 下三角矩阵正好相反
- 压缩方法
  - 只存上(下)三角阵中上(下)三角中的元素
  - 常数c共享一个存储空间
- 所需空间

$$N = \frac{n(n+1)}{2} + 1$$

稀疏矩阵

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

- 分布特征
  - 只有少量非零元素，且非零元素的分布没有规律
- 压缩方法
  - 只存非零元素
- 所需空间
  - 与非零元素的个数和存储方式有关

1．对称矩阵

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

如何确定一维数组的大小？

$$N=\frac{n(n+1)}{2}$$

设：存放下三角阵中的元素，
则：如何确定元素Aij在一维数组中的位置？

$$Loc(A_{ij}) = \begin{cases} \frac{i\times(i+1)}{2}+j & \text{当 } i>=j, A_{ij}\text{在下三角阵中} \\ \frac{j\times(j+1)}{2}+i & \text{当 } i<j, A_{ij}\text{在上三角阵中（根据 } A_{ij}=A_{ji}\text{）} \end{cases}$$

1
2
3
3
4
5
4
5
6
7
…

2. 三角矩阵

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

如何确定一维数组的大小？

$$N=\frac{n(n+1)}{2}+1$$

设：在下三角阵中，
则：如何确定元素Aij在一维数组中的位置？

$$Loc(A_{ij}) = \begin{cases} \frac{i\times(i+1)}{2}+j, \text{当} i>=j, \text{即下三角阵中的元素} \\ \frac{n\times(n+1)}{2}, \quad \text{当} i<j, \quad \text{即下三角阵中的常数} \end{cases}$$

1
2
3
3
6
5
…
…
4

1.三元组表存稀疏矩阵

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

M=5
N=5
T=7

i	j	Aij
0	0	1
0	1	2
0	4	5
1	1	3
2	1	4
3	2	6
4	3	8

- 考虑：
  - 只存非零元素
  - 一个非零元素的必需信息有：（i，j，a_{ij}）
  - 记录一个稀疏矩阵的必需信息有：行数M，列数N，非零元素个数T

广义表

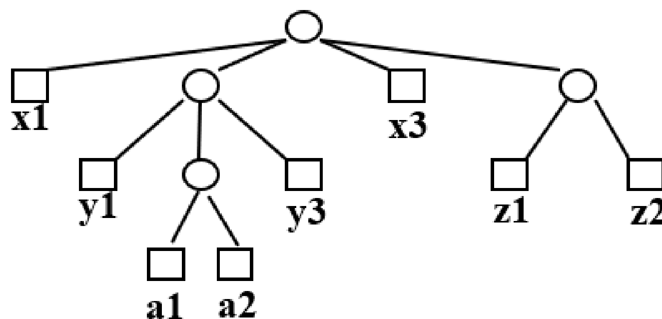
1、基本概念

- 回顾：线性表
  - 由n（n≥0）个数据元素组成的有限有序序列
  - 线性表的每个元素都具有相同的数据类型
- 广义表（Generalized Lists，也称Multi-list）
  - 如果一个线性表中还包括一个或者多个子表
  - 一般记作：L＝（x_0，x_1，…，x_i，…，x_{n-1}）

2、广义表的类型

➢ 纯表（pure list）

- 从根结点到任何叶结点只有一条路径
- 任何元素（原子、子表）只能在广义表中出现一次



(x1, (y1 ,(a1 ,a2), y3), x3 ,(z1 ,z2))

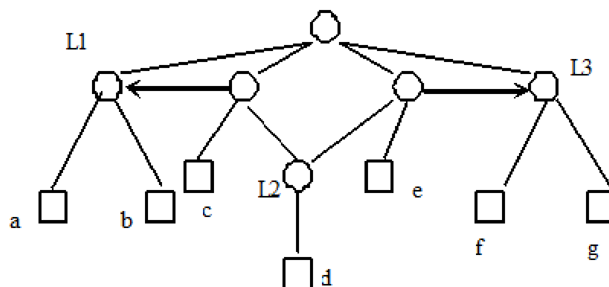
(L1:(a, b), (L1, c, L2: (d)), (L2, e, L3: (f, g)), L3)

➢ 可重入表（再入表）

- 其元素(包括原子和子表)可能在表中多次出现
- 对应一个DAG

➢ 对子表和原子标号

((a, b), ((a,b),c,(d)), ((d), e, (f, g)) , (f ,g))

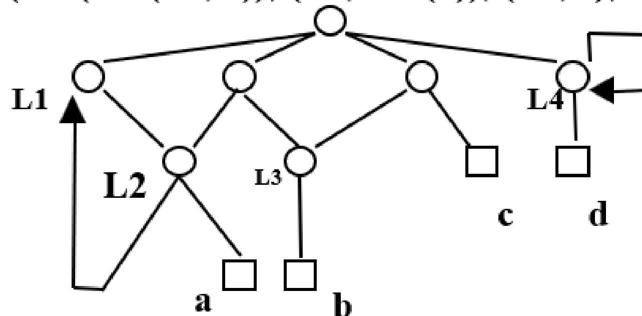


没有标志箭头的边都指向下方

➢ 循环表

➧ 包含回路 ，循环表深度为无穷大

(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d,L4))



➢ 图⊇再入表⊇纯表(树)⊇线性表

➧ 广义表是线性与树形结构的推广

为了分配一个大小为N的空闲块，找到长度为K的，并且K大于等于N的空闲块。

第一种分配：将上述整个空闲分配，则会造成“内部碎片”；
第二种分配：将该空闲块中的N个字节分配出去，同时将剩下的K-N个字节作为新的空闲块，则会造成“外部碎片”

最佳BST树

- 一个拥有n个关键码的集合，关键码可以有n!种不同的排列法
  - 是否正好可以构成n!棵二叉搜索树？
- 其实不然，不同排列所构成的BST树有可能相同
  - 例如，｛2，1，3｝和｛2，3，1｝
- 可以证明，这n!种排列中，只有 $C_{2n}^{n}-C_{2n}^{n-1}=\frac{1}{n+1}C_{2n}^{n}$（组合数学中称之为Catalan数）种前序排列能够构成二叉搜索树。
- 如何评价上述BST树的效率呢？

卡特兰数 $\frac{1}{n+1}C_{2n}^{n}$

BST树效率的度量

- 成功检索
  - 比较的次数就是关键码所在的层数加1（根为第0层）
- 不成功检索
  - 比较次数就等于被检索关键码所属的那个外部结点的层数
- 平均比较次数

$$ASL(n)=\frac{1}{W}\left[\sum_{i=1}^{n}p_i(l_i+1)+\sum_{i=0}^{n}q_i l_i'\right]$$

p_i：检索第i个内部结点的的频率

q_i：检索关键码处于第i和第i+1内部结点间的频率

$W=\sum_{i=1}^{n}p_i+\sum_{i=0}^{n}q_i$

l_i：第i个内部结点的层数

l_i'：第i个外部结点的层数

动态规划构造法

- 动态规划过程
  - 第1步：构造包含1个关键码的最佳二叉搜索树
    - ✓找t（0，1），t（1，2），…，t（n-1，n）
  - 第2步：构造包含2个关键码的最佳二叉搜索树
    - ✓找t（0，2），t（1，3），…，t（n-2，n）
  - 再构造包含3，4，…个关键码的最佳二叉搜索树
  - 最后构造包含n个关键码的t（0，n）

若干定义

- 树根定义：r(i，j)
  - 包含的内部结点关键码为：$(key_{i+1}，key_{i+2}，\cdots，key_j)$
  - 内部结点和空树叶的权为：$(q_i，p_{i+1}，q_{i+1}，\cdots，p_j，q_j)$
- 查询代价定义：C(i，j)

$$\sum_{x=i+1}^{j} p_x(l_x+1)+\sum_{x=i}^{j} q_x l'_x$$

- 权的总和

$$W(i，j) = p_{i+1} + \cdots + p_j + q_i + q_{i+1} + \cdots + q_j$$

对于子树 T(i, j)

- 以key_k为根
  - 左子树包含key_{i+1}，…，key_{k-1}
    - ✓C(i, k-1)已求出
  - 右子树包含key_{k+1}，key_{k+2}，…，key_j
    - ✓C(k, j)已求出
- $C(i, j)=W(i, j) + \min_{i<k\le j}(C(i, k-1) + C(k, j))$

AVL

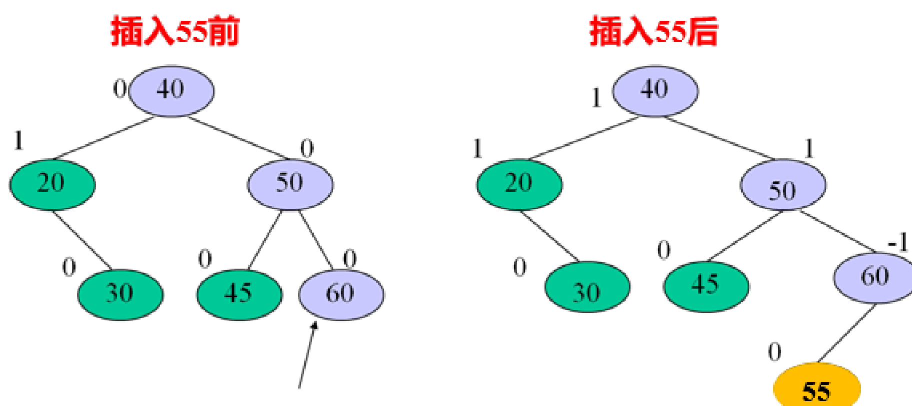
- AVL树的性质
  - 空二叉树是一个AVL树；
  - 如果T是一棵AVL树，那么其左右子树T_L、T_R也是AVL树，并且$|h_R-h_L|\leq 1$，h_L、h_R 是其左右子树的高度；
  - 如果T是一棵具有n个结点的AVL树，其高为$O(\log n)$。

平衡因子bf（balance factor）

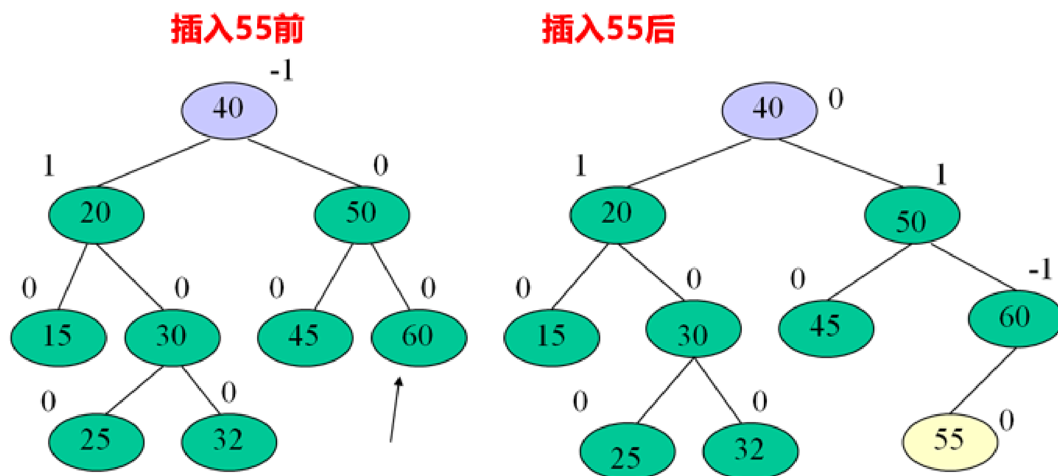
- *bf*(x)：表示结点x的平衡因子
  - 定义：*bf(x)* = *x*的右子树高度 － x的左子树高度。
  - *bf*(*x*) 取值：{0, 1, −1}
  - AVL树的ASL：$O(\log_2 n)$
- AVL树的平衡调整
  - 结点插入
  - 结点删除

1. AVL树结点的插入

- 与BST树类似，执行失败的查找，确定插入位置插入
- 根据平衡因子决定是否调整
  - 第一情况：原来树平衡，插入新结点后发生偏重但未失衡



➢ 第二情况：路径上原来各结点是有偏重的，插入新结点后，反而使这些结点恢复平衡。



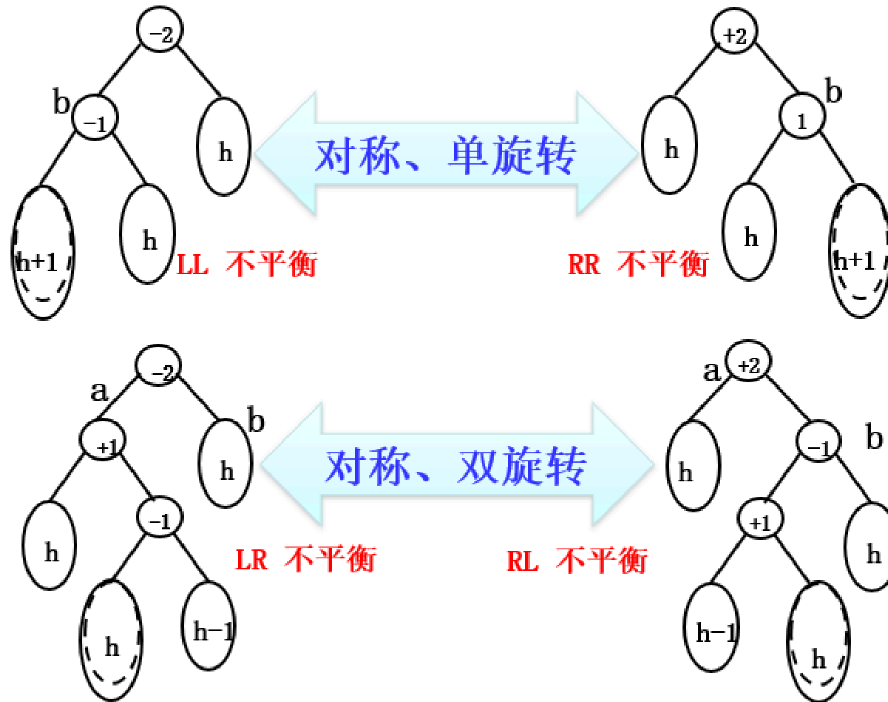
➢ 第三种情况：插入结点后失衡，执行平衡操作

➢ 分四种情况（设A为失衡节点）

- LL型：A的左子树的左子树导致失衡，A的平衡因子为-2
- LR型：A的左子树的右子树导致失衡，A的平衡因子为-2
- RL型：A的右子树的左子树导致失衡，A的平衡因子为2
- RR型：A的右子树的右子树导致失衡，A的平衡因子为2

➢ 失衡结点一定在根结点与新加入结点间的路径上，并且其平衡因子只能是2或者-2（之前是1或者-1）

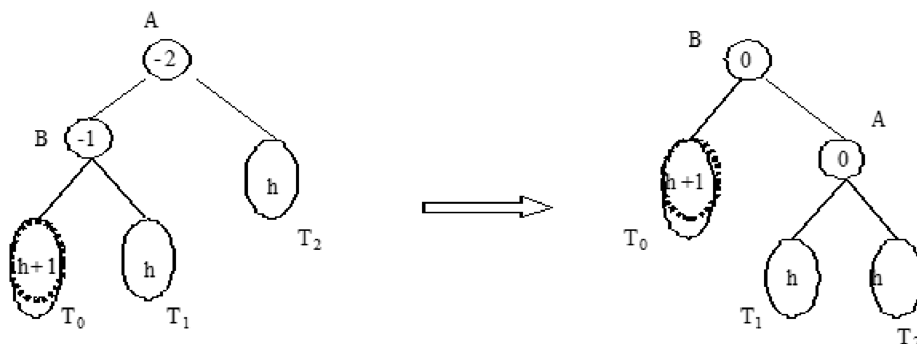
四种不平衡情况



1、单旋转

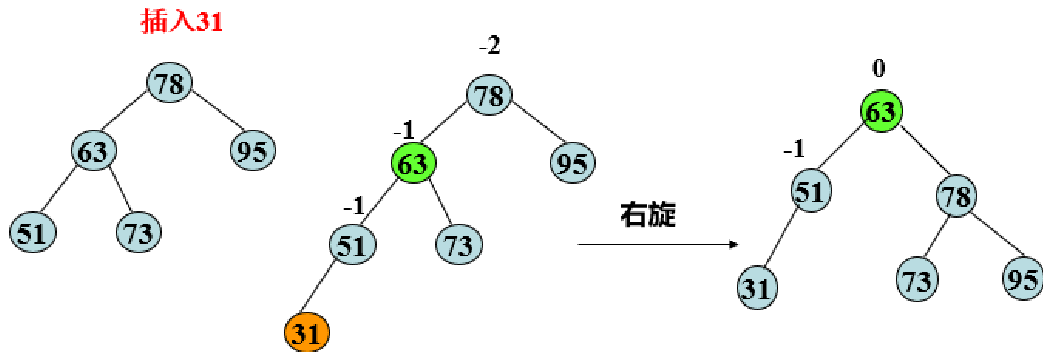
- 调整过程仅涉及A和B两个结点（以LL型调整为例）
  - 令B代替A成为新根，A作为B的右子结点，A和B的子树根据BST的性质分别挂接到B和A结点下，保持中序周游序列为

T_0 B T_1 A T_2



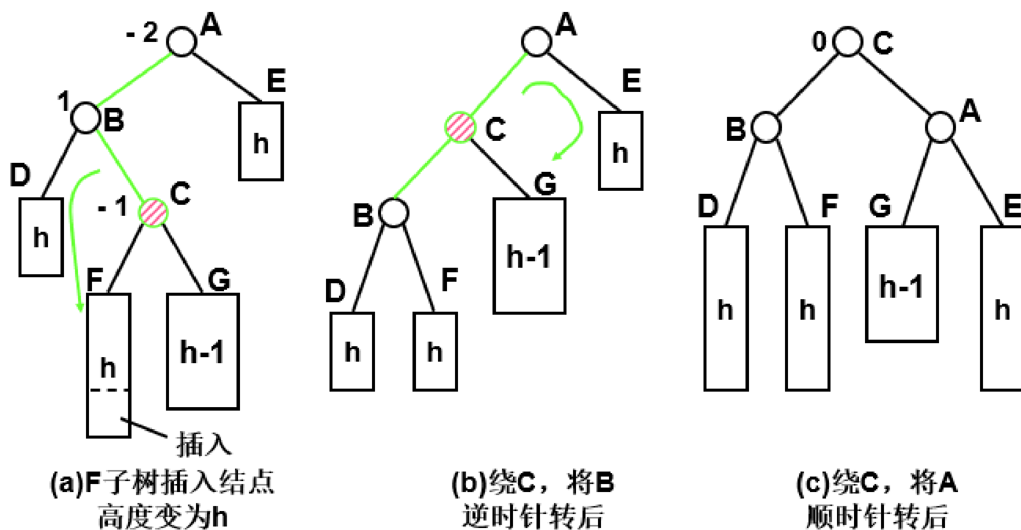
单旋转：LL型示例

➤ 导致不平衡的结点为A的左子树的左子树，这时A的平衡因子为-2



2、双旋转（提升中间节点）

➤ 调整过程涉及ABC三个结点（以LR型调整为例）



(a)F子树插入结点高度变为h

(b)绕C，将B逆时针转后

(c)绕C，将A顺时针转后

2. AVL树结点的删除

- 删除操作同BST删除：与后继交换再删除
- 删除会导致树高及平衡因子变化，需要沿着被删除结点到根结点的路径来调整这种变化。但删除比插入复杂
  - 删除后的再平衡可能需要在相应的路径上的不止一次实施单旋或双旋

AVL删除后的调整

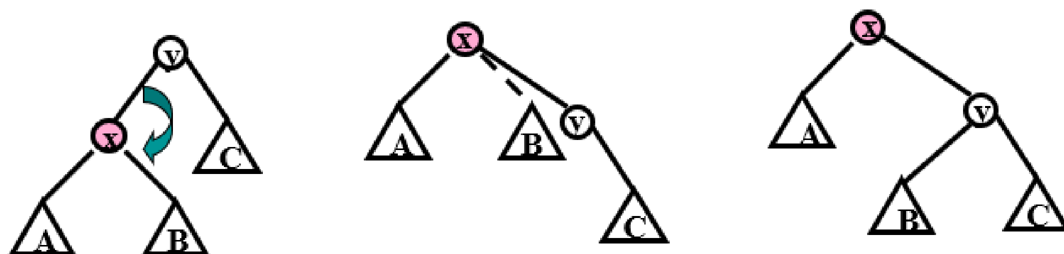
- 删除结点后，如果导致AVL树失衡，需要执行类似LL、RR、LR和RL操作进行平衡调整
- 如平衡调整后不导致新的失衡，则不必继续向上回溯
  - 用布尔变量*modified*来标记（初值为TRUE ），当为FALSE时，停止回溯
- 分情况处理，假定
  - 最下层不平衡发生在结点root开始的子树中
  - 删除操作发生在root的左子树中

展开(splaying)

- 访问一次结点 *x*，完成一次展开过程
  - 插入检索*x*时：把结点*x*移到BST的根结点
  - 删除结点*x*时：把结点*x*的父结点移到根结点
- 结点*x*的展开包括一组旋转
  - 调整结点*x*、父结点、祖父结点，把*x*旋转到更高层
  - 旋转分：单旋转和双旋转

单旋转

- 当被访问结点*x*是根结点的子结点时，单旋转
  - 伸展树的单旋转与AVL树的单旋转是一样的
  - 结点提升一层

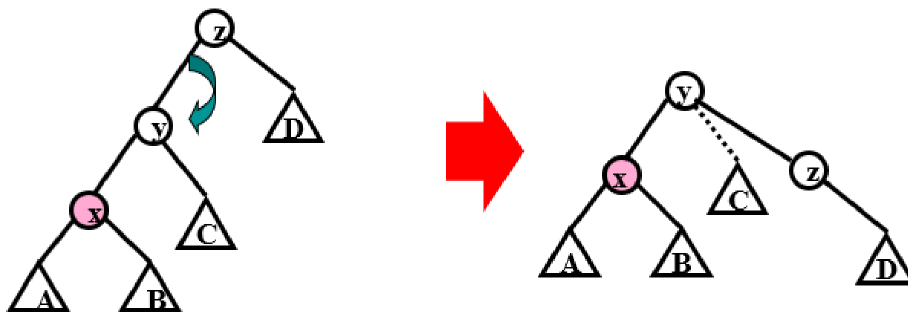


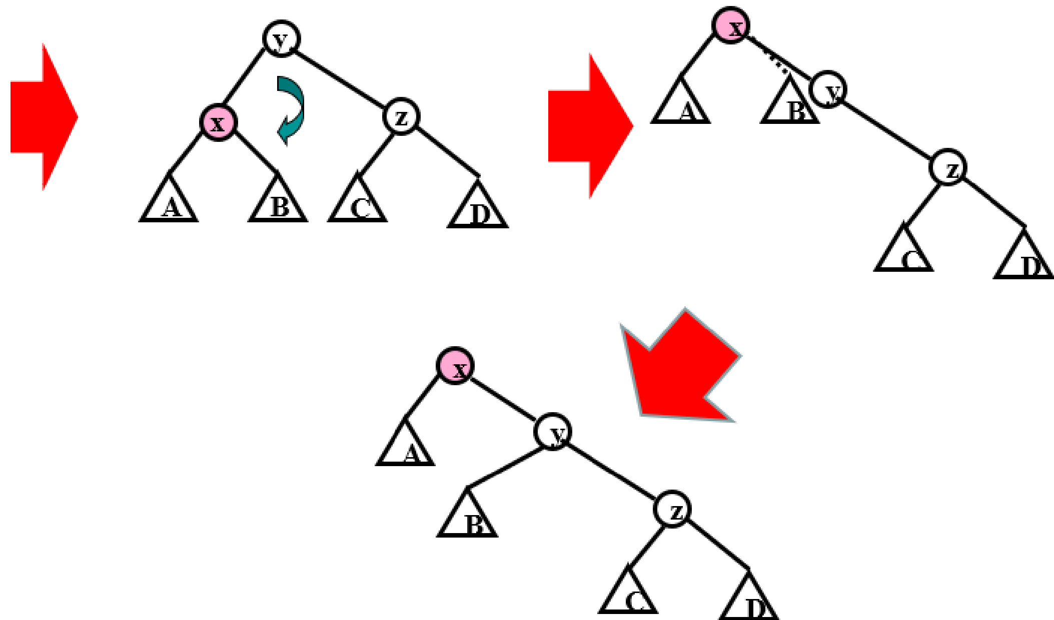
双旋转

- 伸展树需要两种类型的双旋转
  - 一字形旋转，也称为同构调整
  - 之字形旋转，也称为异构调整
- 双旋转涉及
  - 结点*x*、*x*的父结点 *y*、*x*的祖父结点*z*
  - 把结点*x*在树结构中向上移两层

一字形旋转

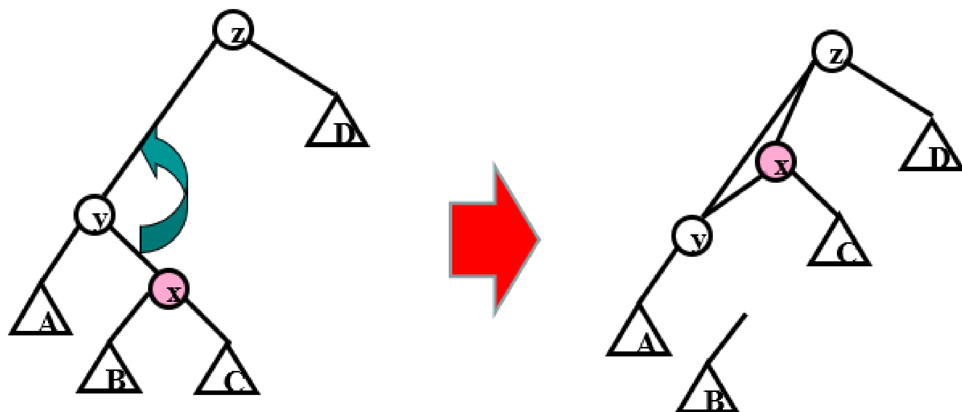
- *x*、*y*、*z*是一顺的
  - 左顺：*x*是*y*的左子结点，*y*是z的左子结点
  - 右顺：*x*是*y*的右子结点，*y*是z的右子结点

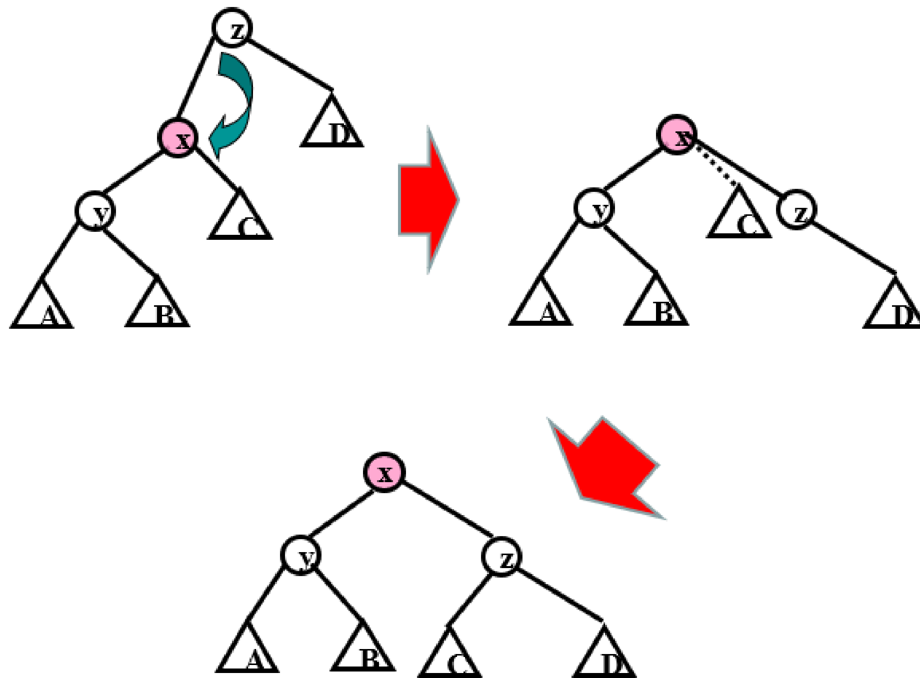




之字形旋转

- 出现以下两种情况之一时，会发生之字形旋转：
  - x是y的左子结点，y是z的右子结点
  - x是y的右子结点，y是z的左子结点





两种旋转的不同作用

- 之字形旋转
  - 把新访问的记录向根结点移动
  - 使子树结构的高度减1
  - 趋向于使树结构更加平衡
- 一字形提升
  - 一般不会降低树结构的高度
  - 只是把新访问的记录向根结点移动

- 一系列双旋转
  - 直到结点x到达根结点或者根结点的子结点
- 如果结点x到达根结点的子结点
  - 进行一次单旋转使结点x成为根结点
- 这个过程趋向于使树结构重新平衡
  - 使访问最频繁的结点靠近树结构的根层
  - 从而减少访问代价

几种平衡机制比较

- AVL树要求完全平衡
  - AVL树结构与访问频率无关，只与插入、删除的顺序有关
- 伸展树与操作频率相关
  - 根据插入、删除、检索等动态地调整
- RB-Tree局部平衡
  - 统计性能好于AVL树
  - 增删记录算法性能好

基于伸展树的区间操作

（1） 区间提取

给定一个数列$[n_1, n_2, \cdots, n_m]$，假设我们需要提取区间$[n_a, n_b]$的子序列。

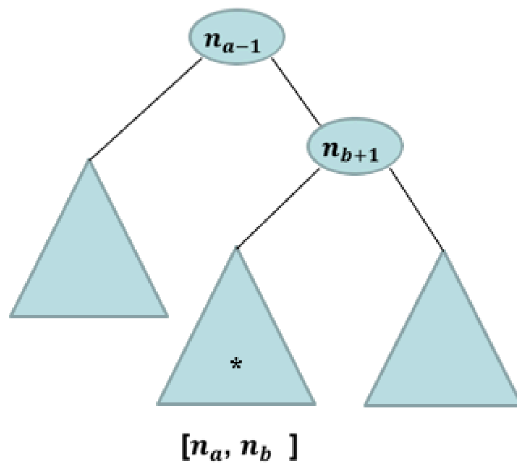
步骤：

Step 1：将位置n_a前面的元素转到树根；

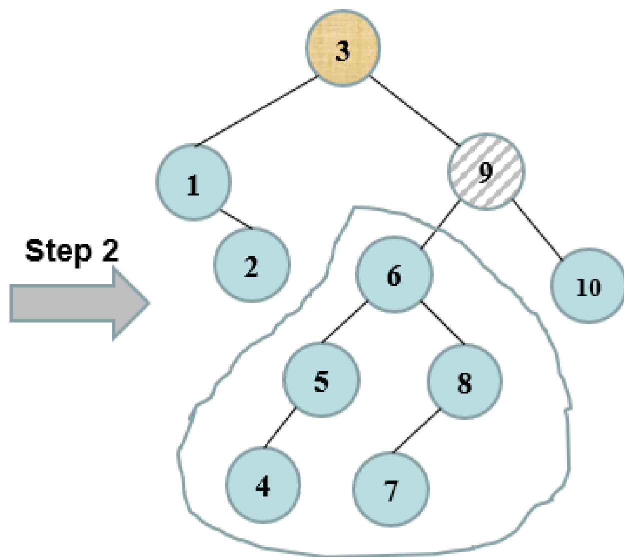
Step 2：将位置n_b后面一个元素转到树根的右儿子；

Step 3：则树根右儿子的左子树就是需要提取的区间。

基于伸展树的区间操作



例如： A=[1,2,3,4,5,6,7,8,9,10]
提取 [4---8]



基于伸展树的区间操作

(2) 区间删除

给定一个数列$[n_1, n_2, \cdots, n_m]$，假设我们删除区间$[n_a, n_b]$子序列。

步骤：

Step 1：利用区间提取方法找到区间所对应的子树；

Step 2：删除对应的子树

基于伸展树的区间操作

(3) 区间插入

给定一个数列A=[n_1，n_2，⋯，n_m]和对应的伸展树，假设我们在n_a和n_{a+1}之间插入一个新序列B。

Step 1：把n_a转到树根；

Step 2：把n_{a+1}转到树根的右儿子；

Step 3：将待插入序列B 构建成一颗伸展树。

Step 4：把新构建的伸展树插入到树根的右儿子的左子树。

(3) 区间翻转

给定一个数列A=[n_1，n_2，⋯，n_m]和对应的伸展树，假设我们在n_a和n_b之间插入一个子序列做一个前后翻转。

Step 1：按照区间提取算法，定位[n_a，n_b]所对应的子树；

Step 2：对此区间中的每个结点，交换它的左右儿子。