中国科学技术大学

University of Science and Technology of China

数据结构

查找表（9 学时）

January 21, 2021

目录

查找表的概念

问题描述

- 给定一个由同一类型的数据元素 (或记录) 构成的集合
- 从中查找指定数据项（或数据元素某个特征）的数据元素或记录
- 这是计算机操作中最基本的一类操作，其算法实现的时空性能直接影响许多实际应用
- 集合元素间的关系松散；我们在集合元素间添加“关系”，形成新的“逻辑结构”，得到“查找表”，让查找效率得到提高，这就是“查找表设计”

查找表的主要操作

- 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

查找表的基本概念和术语

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：在查找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，此类表为动态查找表

关键字/key

- 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）
- 若此关键字可以识别唯一的一个记录，则称之谓“主关键字”
- 若此关键字能识别若干记录，则称之谓“次关键字”

查找/searching

- 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数据元素或（记录）
- 若查找表中存在这样一个记录，则称“查找成功”，查找结果：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”，查找结果：给出“空记录”或“空指针”

查找的方法与评价

查找方法

- 查找的方法取决于查找表的结构，所谓查找表的结构来自于对集合中的元素间人为添加的“关系”

查找性能的评价

- 查找速度、占用存储空间多少、算法本身复杂程度
- 平均查找长度 ASL(Average Search Length)：
  - 为确定记录在表中的位置，需和给定值进行比较的关键字的个数的期望值叫查找算法的 ASL
  - 对于包括 n 个记录的表，查找成功时 $ASL = \sum_{i=1}^{n} p_i c_i$
  - 其中 p_i 为查找表中第 i 个记录的概率，且 $\sum_{i=1}^{n} p_i = 1$，c_i 为找到查找表中第 i 个记录需要比较关键字的次数

静态查找表的 ADT

```
1  ADT StaticSearchTable{//不支持待查记录集合的增删
2    数据对象D: D是具有相同特性的数据元素的集合。每个数
3                据元素含有类型相同的关键字，可唯一标识数据元素。
4    数据关系R: 数据元素同属一个集合。
5    基本操作 :
6      Create(&ST,n);//构造一个含 n 个数据元素的静态查找表ST
7      Destroy(&ST);//销毁表ST
8      Search(ST,key);//查找 ST 中其关键字等于key的数据元素
9      Traverse(ST,Visit());//按某种次序对ST的每个元素
10                          //调用函数Visit()一次且仅一次
11 }ADT StaticSearchTable
```

静态查找表的类型

- 顺序查找表：集合元素间添加序偶关系，构成顺序表
- 有序查找表：集合元素间依据大小添加序偶关系，构成有序表
- 索引顺序表：集合元素分块，块间有序，块内无序

顺序查找表

存储结构

```
1  typedef struct{
2    ElemType *elem; //数据元素存储空间基址，建表时
3         //按实际长度分配，0号单元留空
4    int length; // 表长
5  }SSTable;
```

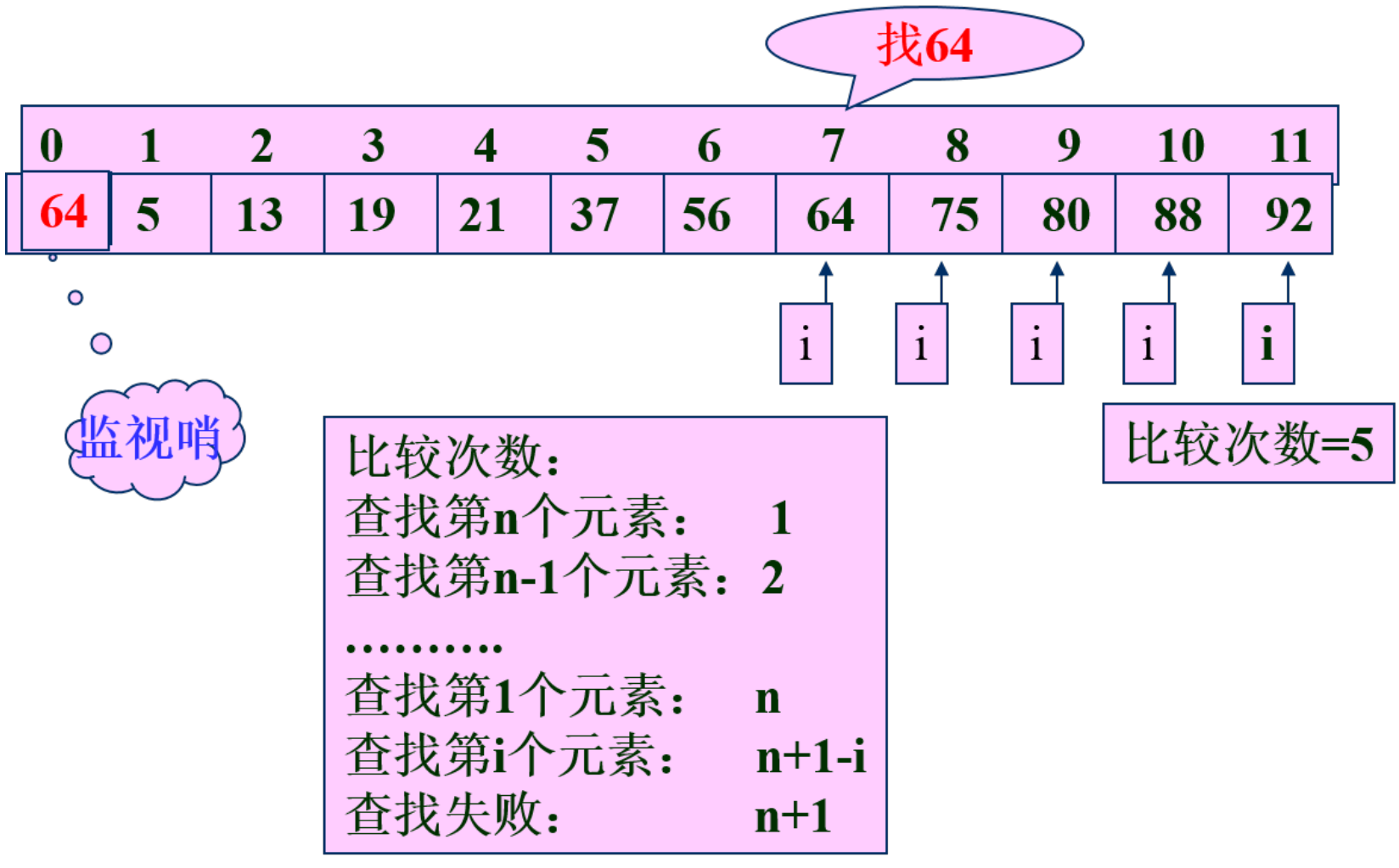
查找操作：search()

```
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
5     ST.elem[0].key = key;  // 设置“哨兵”
6     for (i=ST.length; ST.elem[i].key!=key; --i); // 从后往前找
7     return i;              // 找不到时，i为0
8  } // Search_Seq
```

顺序查找表的例子

查找过程

- 如图，从表的一端开始逐个进行记录的关键字和给定值的比较



顺序查找表的性能分析

平均查找长度:

- $ASL = \sum_{i=1}^{n} p_i c_i$，顺序表查找 $c_i = n - i + 1$
- 故每个元素被等概查找时 $p_i = 1/n$，$ASL = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2}$
- 在不等概率查找的情况下，ASL 在查找概率满足 $p_n \geq p_{n-1} \geq \ldots \geq p_1$ 时取最小值，即顺序查找表中记录按被查概率增序存储，以提高查询效率
- 若查找概率无法事先测定，则查找过程采取的改进办法是将访问频度大的记录后移。或每次查找之后，将刚刚查找到的记录直接移至表尾的位置上

查找不成功时

- 查找算法的 ASL 应是查找成功时的 ASL 和查找不成功时的 ASL 的期望
- 对顺序查找，查找不成功时和给定值进行比较的次数都是 $n+1$; 若设查找成功和不成功的可能性相同，对每个纪录的查找概率也相等，则 $p_i = \frac{1}{2n}$，此时 $ASL = \frac{1}{2n}\sum_{i=1}^{n}(n-i+1) + \frac{1}{2}(n+1) = \frac{3}{4}(n+1)$

有序查找表

有序表

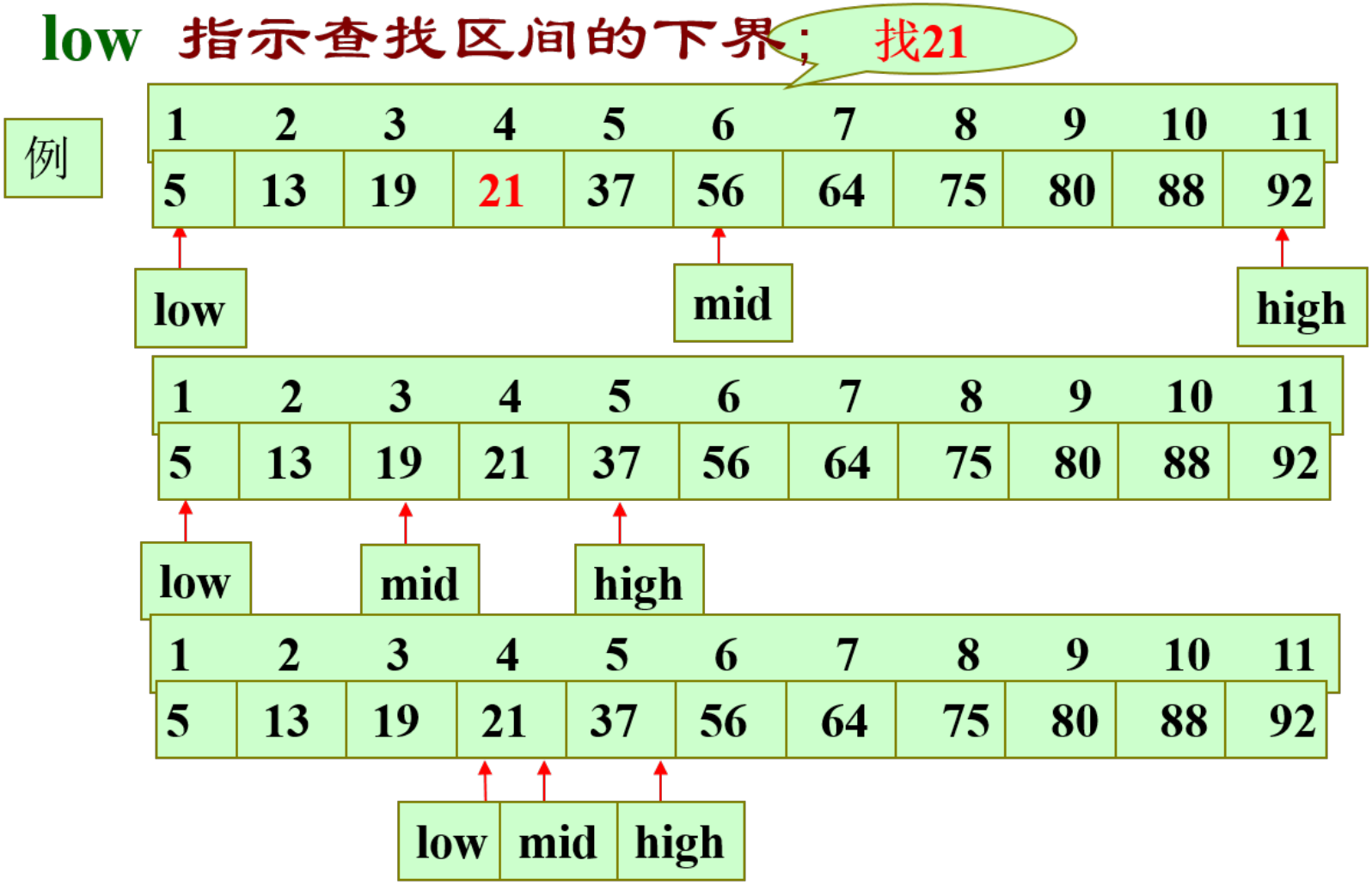
- 顺序表的相邻元素的关键字大小不需要有序，只能顺序查找，平均查找长度大，不适合大的查找表
- 将逻辑结构相邻的元素按关键字排好序，则得到有序查找表，可采用“折半”查找

折半查找/Binary Search

- 查找过程：每次将待查记录所在区间缩小一半
- 适用条件：采用顺序存储结构的有序表

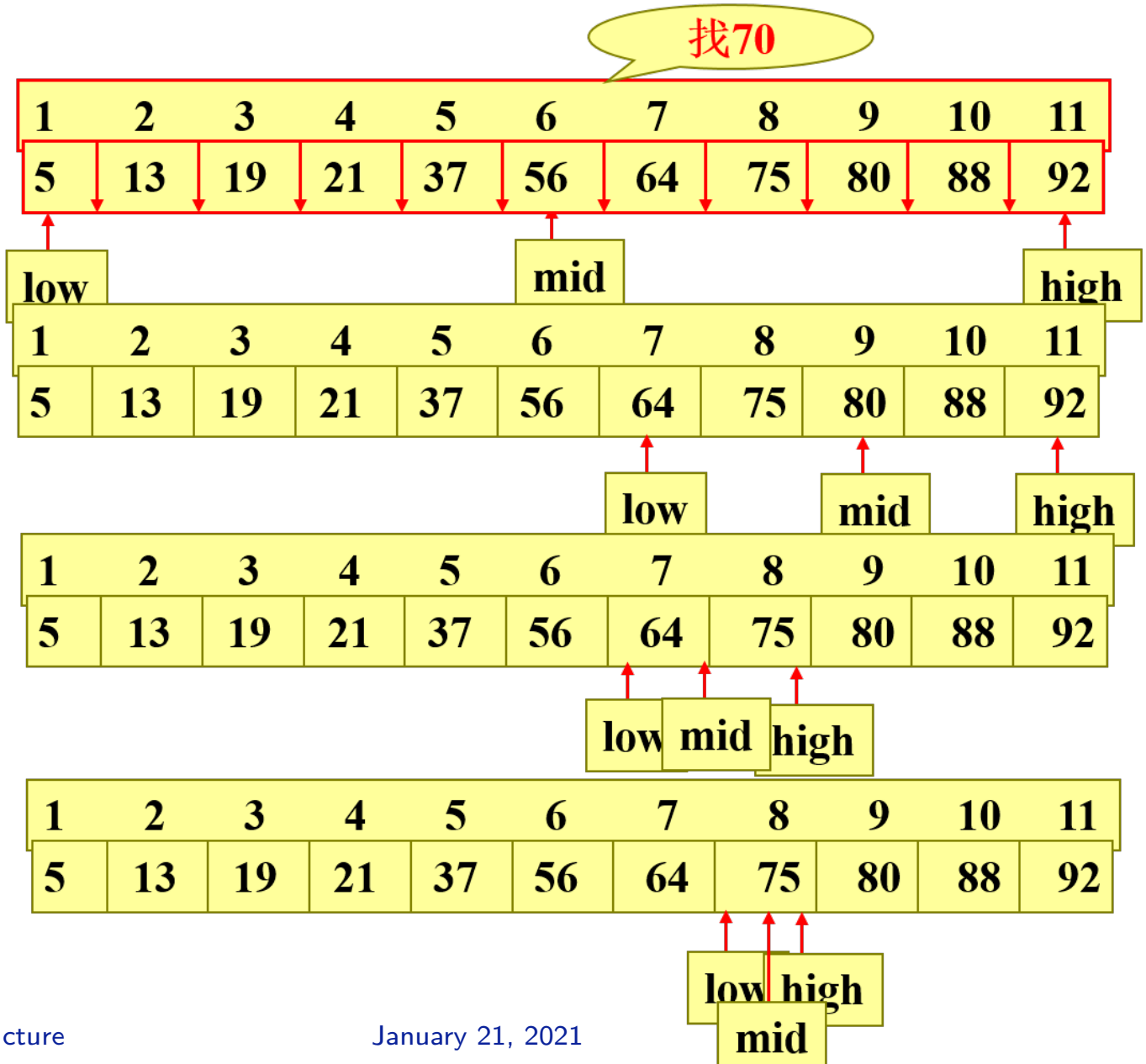
折半查找的例子

查找 21 的过程（查找成功）



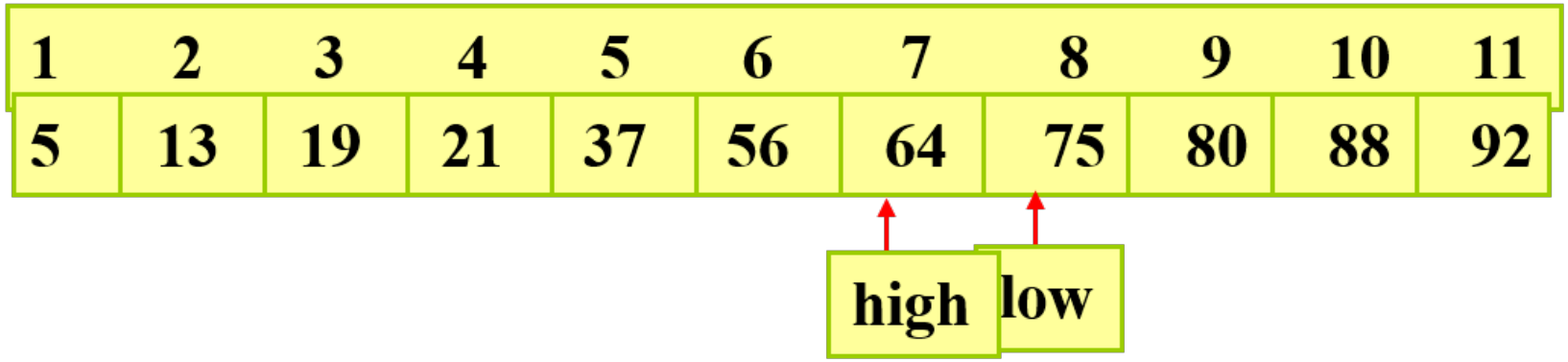
折半查找的例子

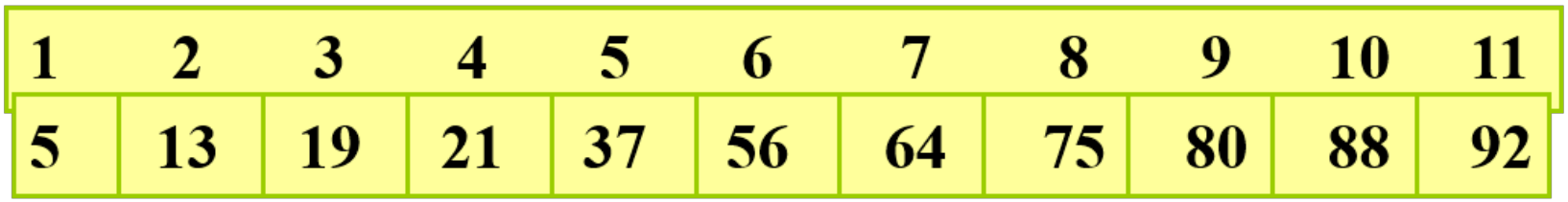
查找 70 的过程（查找失败）



折半查找的例子

查找 70 的过程（查找失败）





当下界 low 大于上界 high 时，则说明表中没有关键字等于 Key 的元素，查找不成功

顺序查找表的折半查找算法

实现代码

```
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
```

性能分析

- 关键字比较次数不超过 $\lfloor \log_2 n \rfloor + 1$，时间复杂度为 $O(\log n)$
- 特别的，当 $n > 50$，查找成功时的 $ASL \approx \log_2(n+1) - 1$
- 折半查找适用条件：有序的顺序表

顺序表和有序表的比较

二者当作查找表时：

	顺序表	有序表
表的特性	无序	有序
存储结构	顺序 或 链式	顺序
插删操作	易于进行	需移动元素
ASL的值	大	小

斐波拉契查找

基本思想

- 适用条件：有序静态查找表，顺序表的实现方式
- 将折半查找的“砍一半”搜索空间，改成“不平衡”地近似“砍一半”，用斐波那契数做分割点：$F_{n+1} = F_n + F_{n-1}$
- 假设有序查找表长 $m = F_{n+1} - 1$，则比较查找关键字 key 和 $ST.elem[F_n].key$, 前一半有 $F_n - 1$ 个记录，后一半有 $F_{n-1} - 1$ 个记录，采用类似折半查找的方法，递归调用
- 若有序查找表的长度 $m \neq F_n - 1$，则取表的前 $F_n - 1$ 项，使得 n 尽可能大；查找表后面的项采用递归调用斐波拉契诃查找

性能分析

- 平均性能优于折半查找，但是最坏性能比折半差（但依然保证是 $O(\log n)$)
- 另一个优点是：分割时只需要进行加减运算

插值查找

思想

- 直接依据给定的 key 值，来估计记录应该在的位置，估计位置的公式 $i = \frac{key - ST.elem[l].key}{ST.elem[h].key - ST.elem[l].key}(h - l + 1)$
- 适用条件：关键字在查找表中“均匀分布”

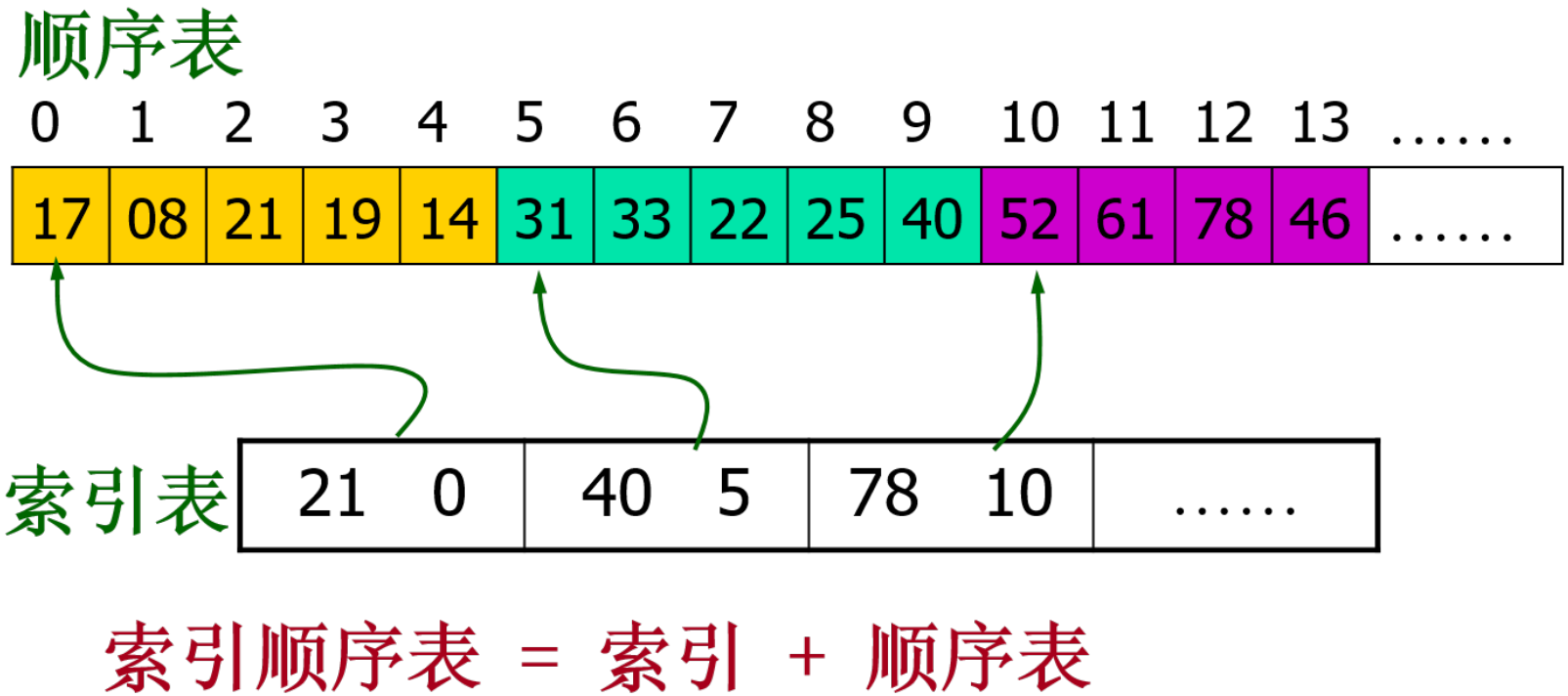
性能

- 表大时，平均性能优于折半查找

索引顺序表

定义

- 在建立顺序表的同时，建立一个索引项，包括两项：关键字项（块内最大关键字）和指针项（块在顺序表的起始位置）
- 索引表按关键字有序，表则为分块有序，如图所示



索引顺序表的查询

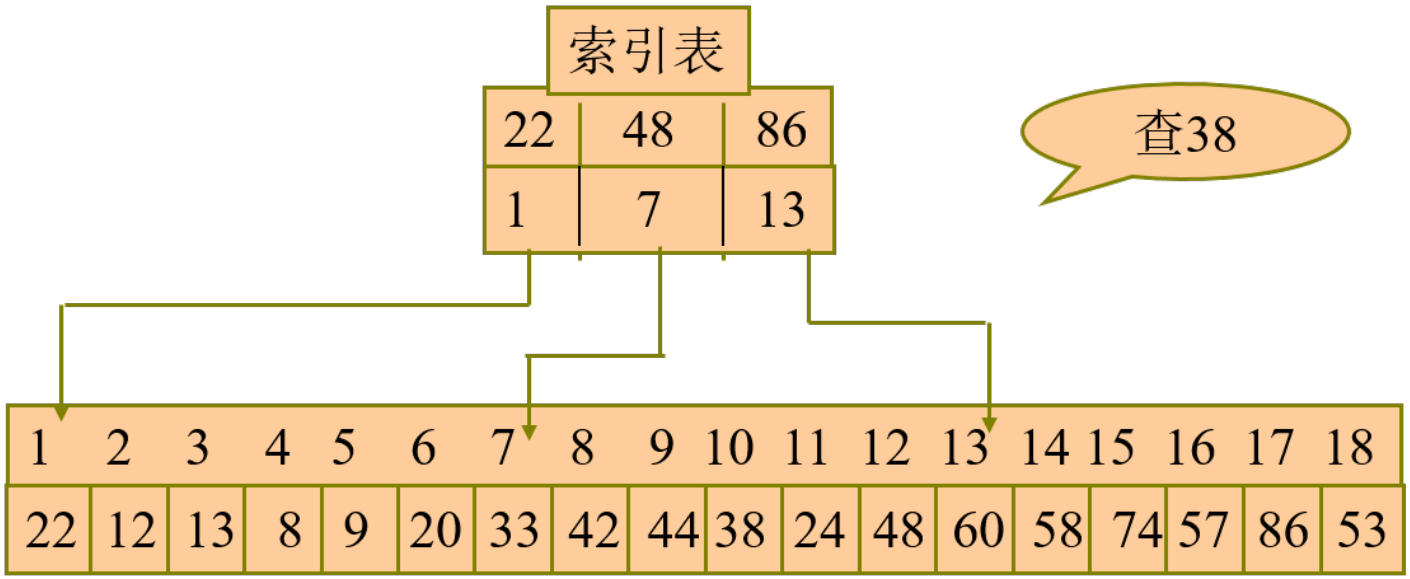
又称分块查找

- 查找过程：将表分成几块，块内无序，块间有序；先确定待查记录所在块，再在块内查找
- 适用条件：分块有序表

算法实现

- 用数组存放待查记录，每个数据元素至少含有关键字域
- 建立索引表，每个索引表结点含有最大关键字域和指向本块第一个结点的指针
- 利用索引表，确定块，然后在块内顺序查找

分块查找的例子



查找过程

- 先找到块（第 2 块）
- 然后在块内顺序查找

分块查找的实现

```
1  typedef struct IndexType{
2    keyType maxkey;//块中最大的关键字
3    int startpos;//块的起始位置指针
4  }Index;
5  int Block_search(RecType ST[],Index ind[],KeyType key,int n,int b){
6    //在分块索引表中查找关键字为key的记录,表长为n,块数为b
7    int i=0,j;
8    while((i<b)&&LT(ind[i].maxkey,key)) i++;
9    if (i>b) {
10     printf("\nNot found"); return(0);}
11   j=ind[i].startpos;
12   while((j<n)&&LQ(ST[j].key,ind[i].maxkey)){
13     if ( EQ(ST[j].key, key) ) break;
14     j++;
15   } //在块内查找
16   if (j>=n||!EQ(ST[j].key,key)){
17     j=0;printf("\nNot found");}
18   return(j);
19 }
```

分块查找的性能分析

$ALS_{bs} = L_b + L_w$

- L_b : 查找索引表，确定所在块的平均查找长度
- L_w: 在块中查找元素的平均查找长度

例子

- 若长度为 n 的表被平均分为 b 块，每块包含 $s = \frac{n}{b}$ 个记录，假设每个记录的查找概率相等
  - 若用顺序查找确定块位置:
    $ASL_{bs} = \frac{1}{b}\sum_{i=1}^{b} i + \frac{1}{s}\sum_{j=1}^{s} j = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}(\frac{n}{s} + s) + 1$
  - 若用折半查找确定块位置: $ASL_{bs} \approx \log_2(\frac{n}{s} + 1) + \frac{s}{2}$

几种查找方法的对比

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

几种查找表的对比

	查找	插入	删除
无序顺序表	O(n)	O(1)	O(n)
无序线性链表	O(n)	O(1)	O(1)
有序顺序表	O(logn)	O(n)	O(n)
有序线性链表	O(n)	O(n)	O(n)
静态查找树表	O(logn)	O(nlogn)	O(nlogn)

结论

- 从查找性能看，最好情况能达 $O(\log n)$，要求表有序
- 从插入和删除的性能看，最好情况能达 $O(1)$，要求存储结构是链表

动态查找表的 ADT

```
1  ADT DynamicSearchTable{//支持待查记录集合的改变
2    数据对象D: D是具有相同特性的数据元素的集合。每个数
3              据元素含有类型相同的关键字，可唯一标识数据元素。
4    数据关系R: 数据元素同属一个集合。
5    基本操作 :
6      InitDSTable(&DT);//构造一个空的动态查找表DT
7      DestroyDSTable(&DT);//销毁动态查找表DT
8      SearchDSTable(DT,key);//查找DT中与关键字key等值的元素
9      InsertDSTable(&DT,e);//若DT中不存在其关键字等于e.key
10                          //的数据元素，则插入e到DT
11     DeleteDSTable(&T,key);//删除DT中关键字等于key的数据元素
12     TraverseDSTable(DT,Visit());//按某种次序对DT的每个结点
13                                 //调用函数Visit()一次且至多一次
14 }ADT DynamicSearchTable
```

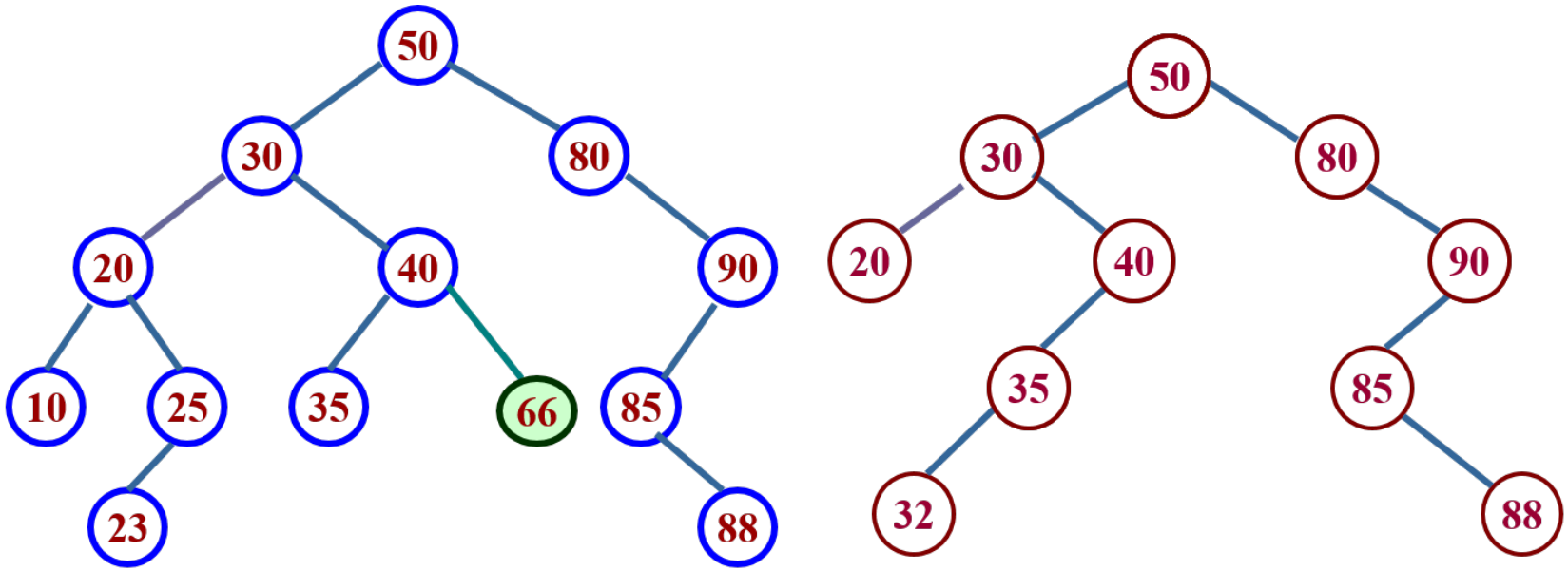
动态查找表的分类

- 二叉排序树：待查记录集合的元素构建二叉树式的逻辑结构
- $B-$ 树和 B^+ 树：待查记录集合的元素构成多叉树的逻辑结构
- 键树

二叉排序树

定义

- 二叉排序树或者是一棵空树；或者是具有如下特性的二叉树：
- 若它的左子树不空，则左子树上所有结点的值均小于根结点的值
- 若它的右子树不空，则右子树上所有结点的值均大于根结点的值
- 它的左、右子树也都分别是二叉排序树
- 如下图所示例子：左图不是二叉排序树，右图是二叉排序树



二叉排序树的存储结构与查找

存储结构: 二叉链表

```
1  typedef struct BiTNode{//结点结构
2      TElemType data;
3      struct BiTNode *lchild,*rchild; //左右指针
4   } BiTNode, *BiTree;
```

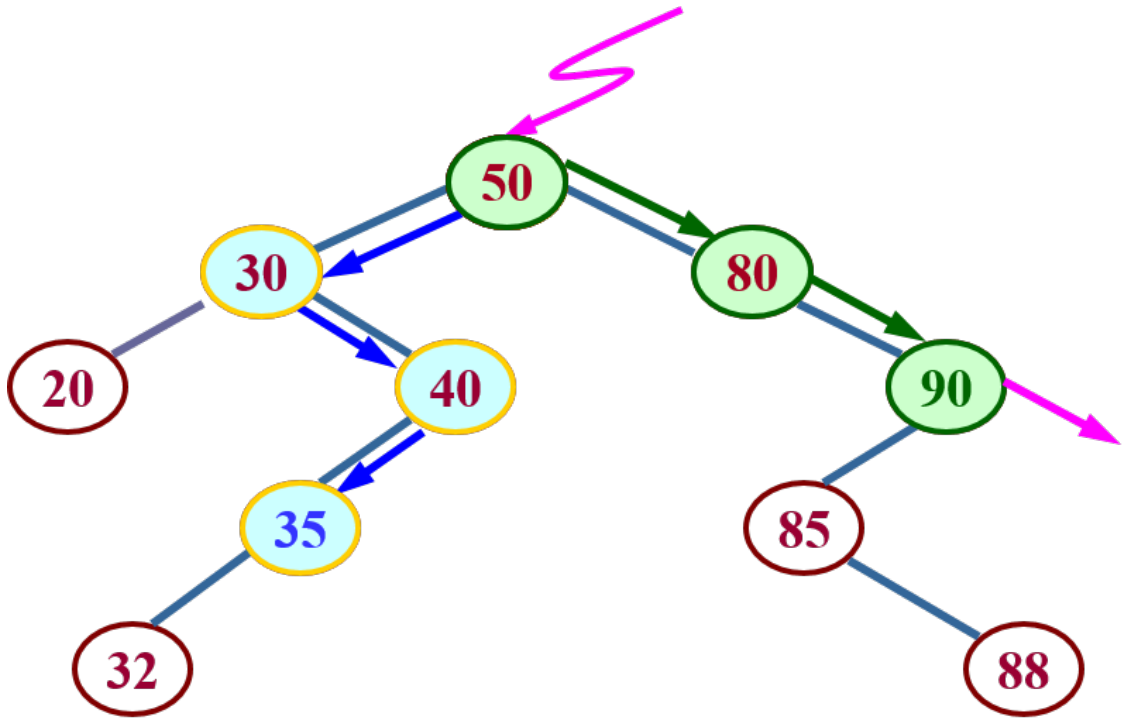
查找算法思想

- 若二叉排序树为空，则查找不成功；
- 否则
  - 若给定值等于根结点的关键字，则查找成功
  - 若给定值小于根结点的关键字，则继续在左子树上进行查找
  - 若给定值大于根结点的关键字，则继续在右子树上进行查找

二叉排序树查找的例子

在二叉排序树中查找关键字值分别等于 50,35,90,95

- 不同颜色代表不同的查询路径：蓝色（35），绿色（90），紫红（95）



二叉排序树查找算法的实现

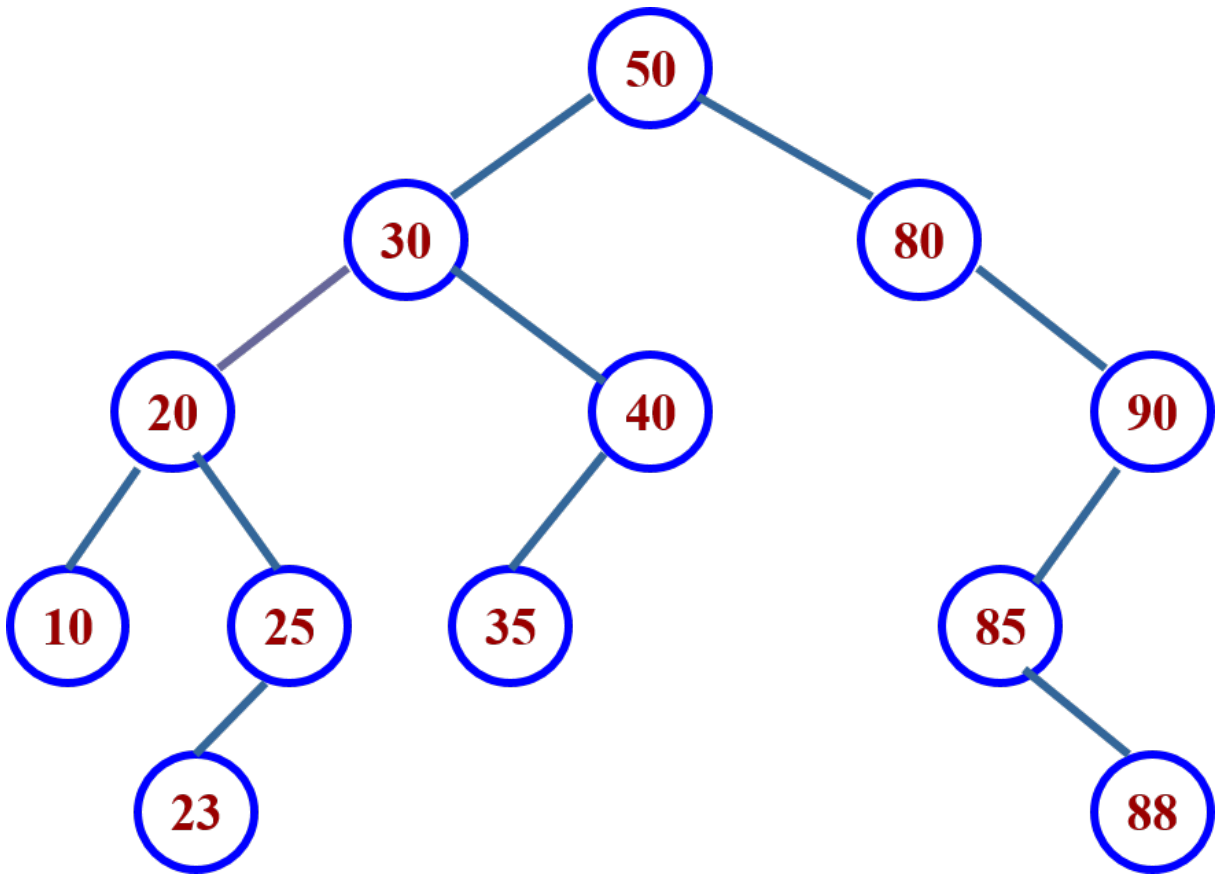
实现代码

```
1  Status SearchBST(BiTree T, KeyType key,BitNode* &f, BitNode* &p){
2    if (!T){
3      p=f; //p指向查找路径上访问的最后一个结点并返回FALSE
4      return FALSE;
5    }
6    if EQ(key, T->data.key){
7      p=T; //若查找成功，p指向该结点，并返回TRUE
8      return TRUE;
9    }
10   if LT(key,T->data.key) {
11     f=T;
12     return(SearchBST(T->lchild,key,f,p));
13     // 在左子树中继续查找
14   }
15   else {
16     f=T;
17     return(SearchBST(T->rchild, key,f,p));
18     // 在右子树中继续查找
19   }
20 }//SearchBST
```

二叉排序树的中序遍历

中序遍历二叉排序树可得到一个关键字的有序序列

- 如图所示的二叉排序树，中序遍历输出序列为：10,20,23,25,30,...,50,80,85,88,90



二叉排序树的插入算法

插入新数据

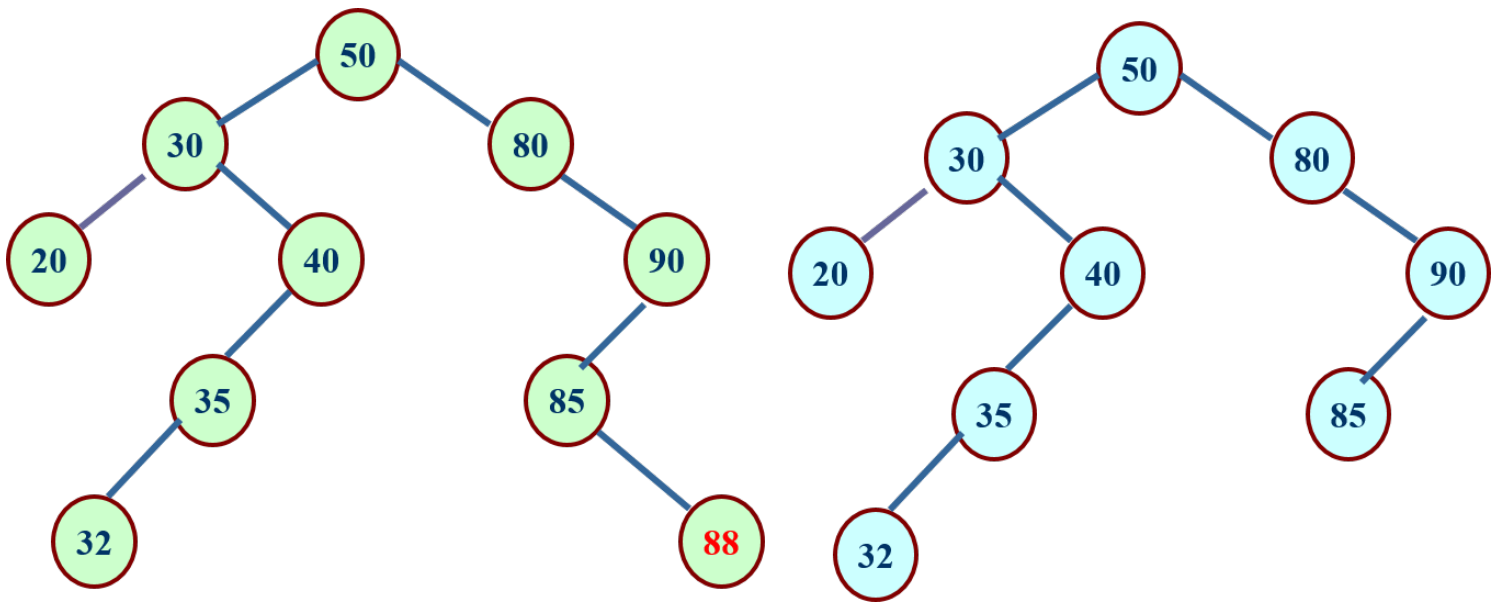
- “插入”操作在查找不成功时才进行
- 若二叉排序树为空树，则新插入的结点为新的根结点；否则，新插入的结点必为一个新的叶子结点，其插入位置由查找过程得到。

```
1  Status InsertBST(BiTree &T, ElemType e){
2    if (!SearchBST(T,e.key,NULL,p)){ //查找不成功
3      BiTree* s = (BiTree)malloc(sizeof(BiTNode));
4      s->data=e;
5      s->lchild=s->rchild=NULL;
6      if (!p) T=s; //插入s为新的根结点
7      else if LT(e.key,p->data.key)
8             p->lchild=s;//插入*s为*p的左孩子
9          else p->rchild=s; //插入*s为*p的右孩子
10     return TRUE; //插入成功
11   }
12   else return FALSE;
13 }//Insert BST
```

二叉排序树的删除算法

删除节点

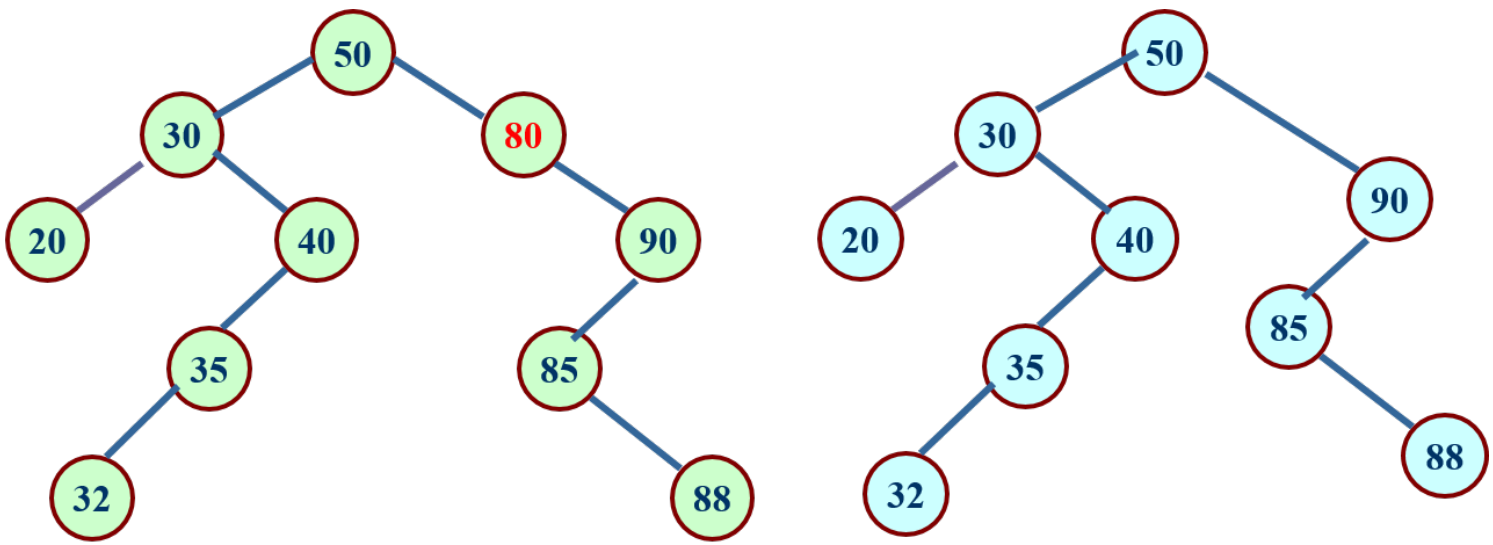
- 删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性
- 可分三种情况讨论
  - 被删除的结点只有左子树或者只有右子树
  - 被删除的结点既有左子树，也有右子树
  - 被删除的结点是叶子，如下图所示，删除叶子节点 key=88, 将其双亲节点的指针域指向空



二叉排序树的删除算法

被删除的节点只有左子树或右子树时

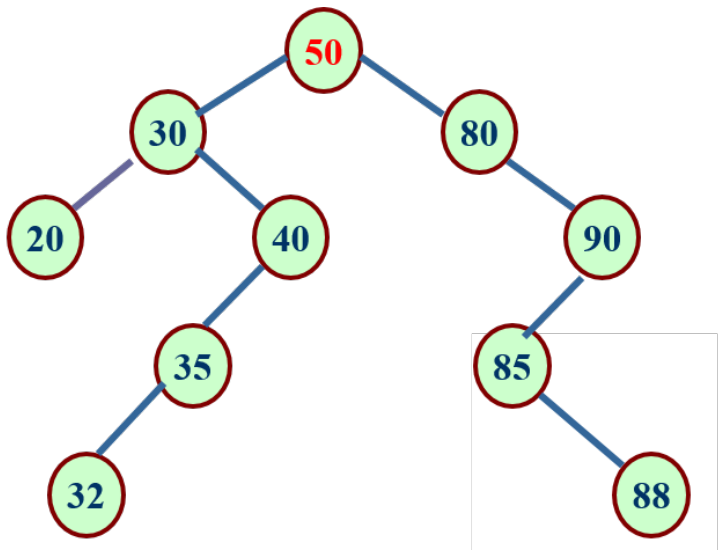
- 如图，删除 key=80 的节点
- 结果：其双亲结点的相应指针域的值改为“指向被删除结点的左子树或右子树”

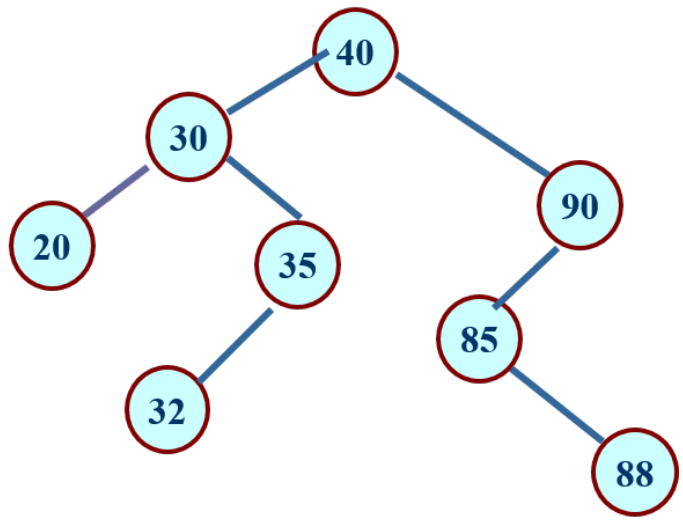


二叉排序树的删除算法

被删除的节点既有左子树也有右子树时

- 如图，删除 key=50 的节点
- 结果：用中序遍历二叉树输出结果序列中被删除节点的前驱替代之，然后再删除该前驱结点





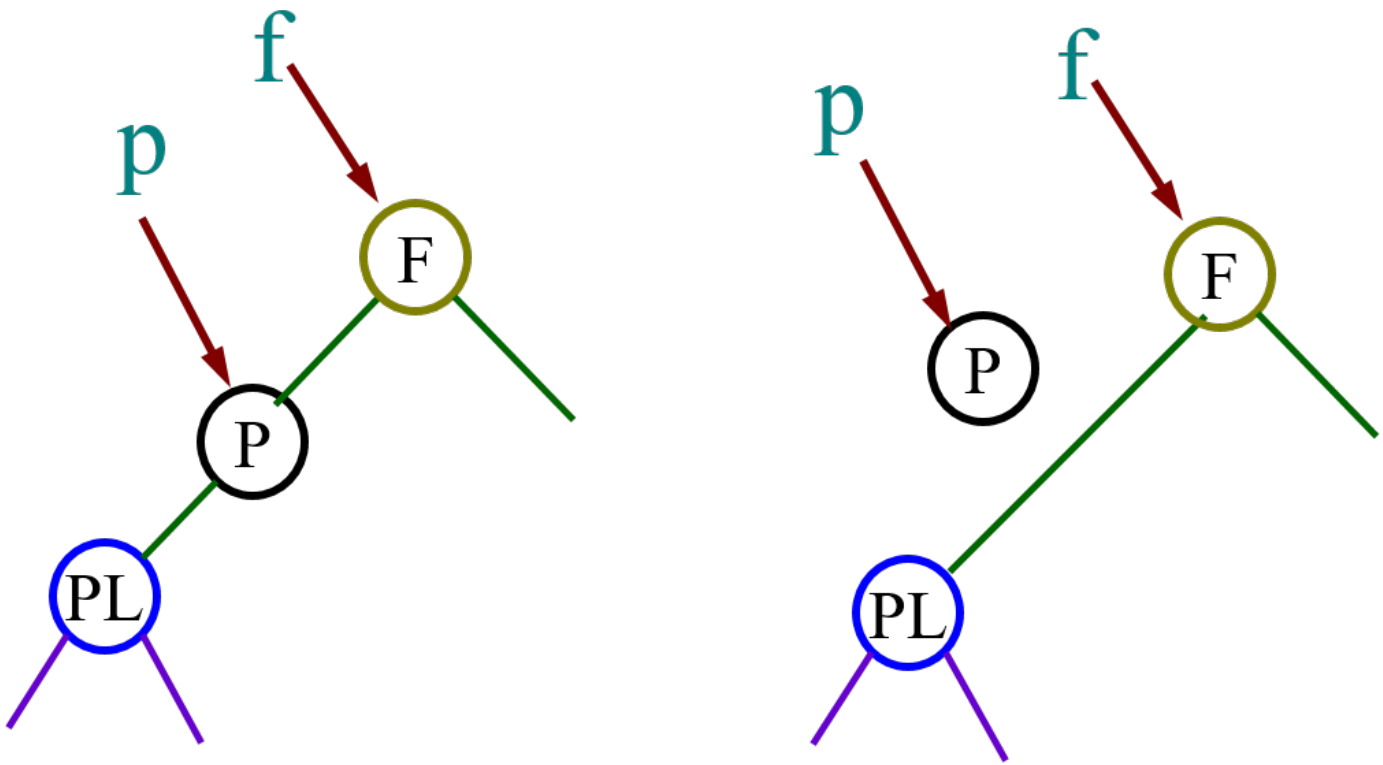
二叉排序树的删除算法

```
1  Status DeleteBST(BiTree &T, KeyType key){
2    BitNode *f=NULL, *p;
3    if (!SearchBST(T,key,f,p)) //查找不成功
4      return FALSE; //不存在关键字等于key的数据元素
5    Delete(p,f);
6  }//DeleteBST
7
8  void Delete(BiTree &p, BiTree f){//删除过程
9    //从二叉排序树中删除结点p，并重接它的左子树或右子树
10   if (!p->rchild && !p->lchild){.....;return ..}//删除叶子节点
11   if (!p->rchild) {.....} //左孩子为空
12   else if (!p->lchild) {.....} //右孩子为空
13       else {.....}//左右孩子都非空
14 }//Delete
```

二叉排序树的删除算法

右子树为空树则只需重接它的左子树

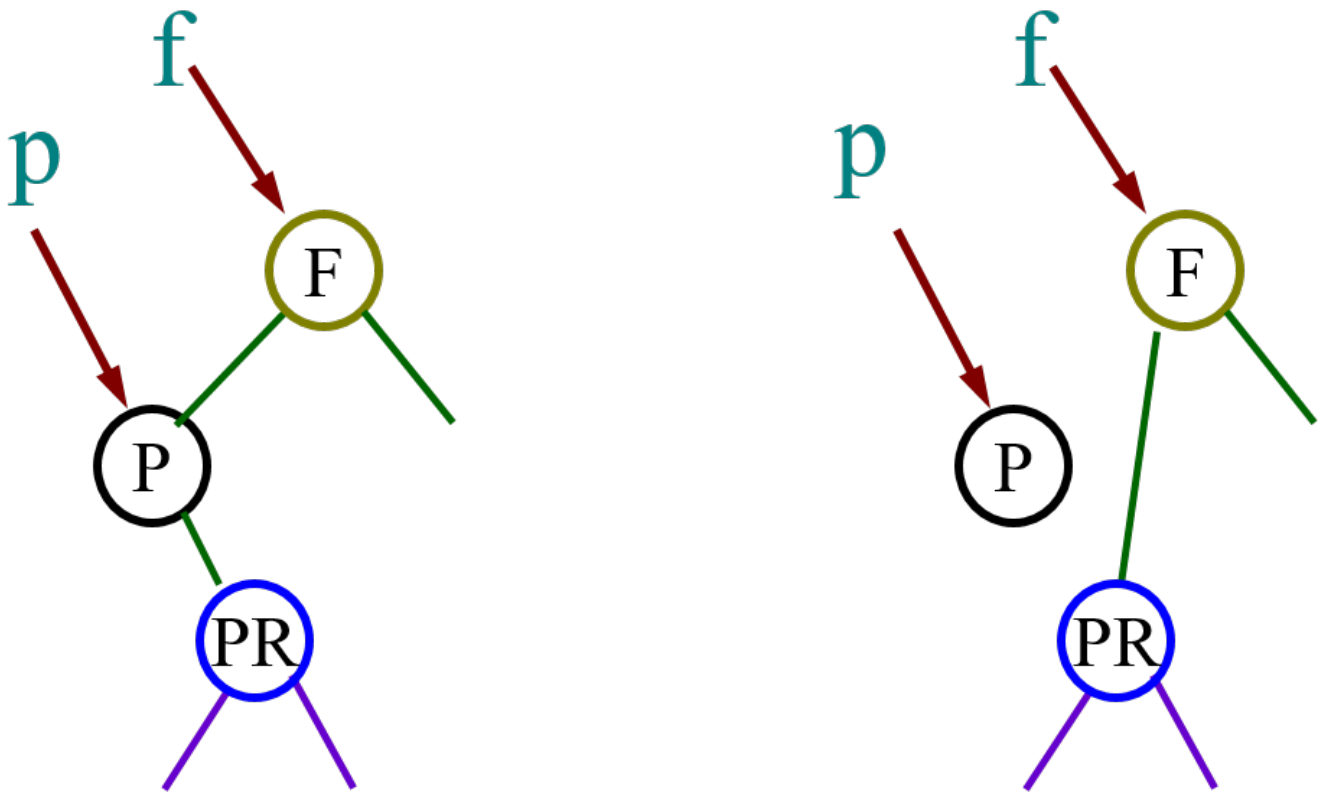
- f->lchild=p-lchild; free(p);
- 需要判定被删除节点是其双亲的左孩子还是右孩子



二叉排序树的删除算法

左子树为空树则只需重接它的右子树

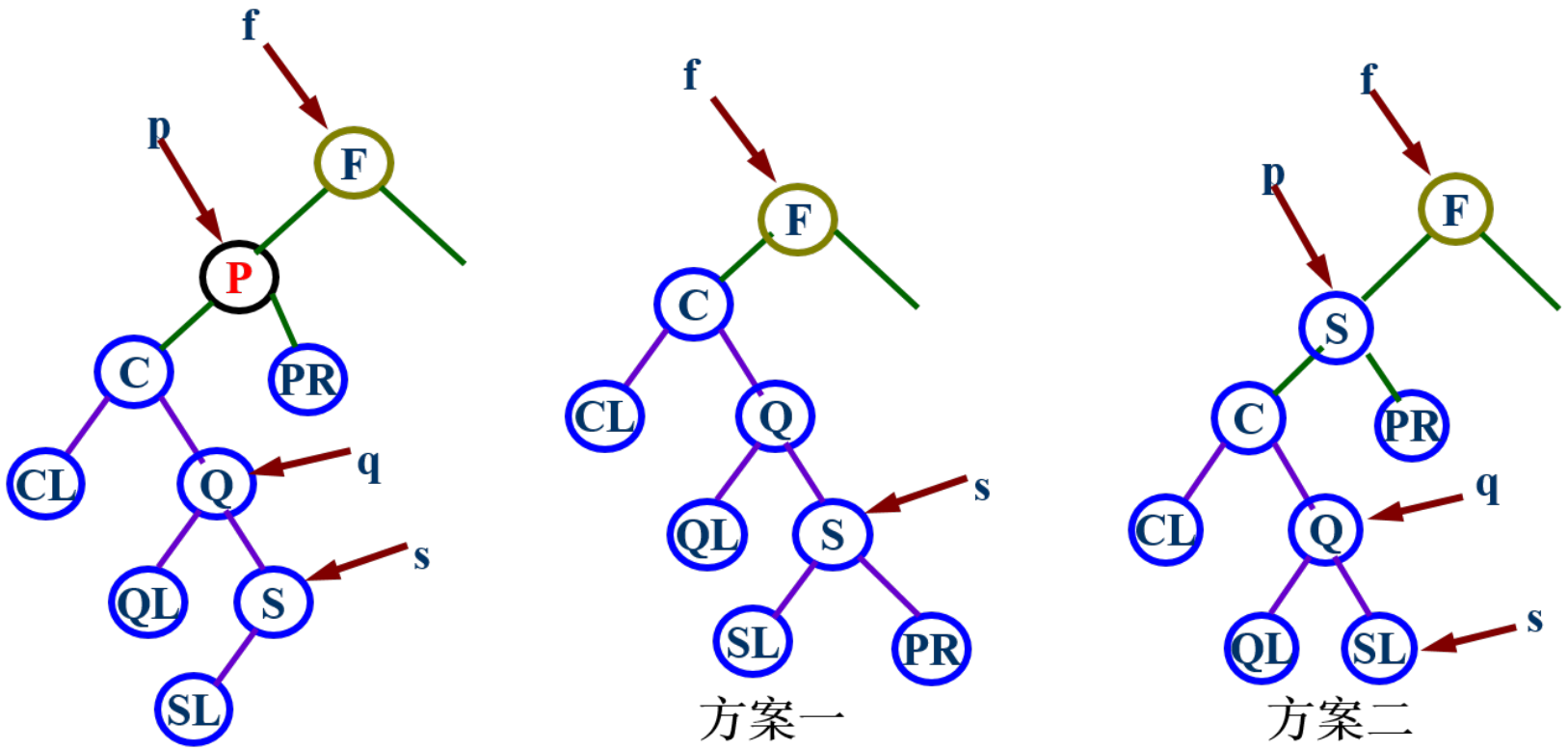
- f->lchild=p->rchild; free(p);
- 需要判定被删除节点是其双亲的左孩子还是右孩子



二叉排序树删除算法

左右子树均不为空

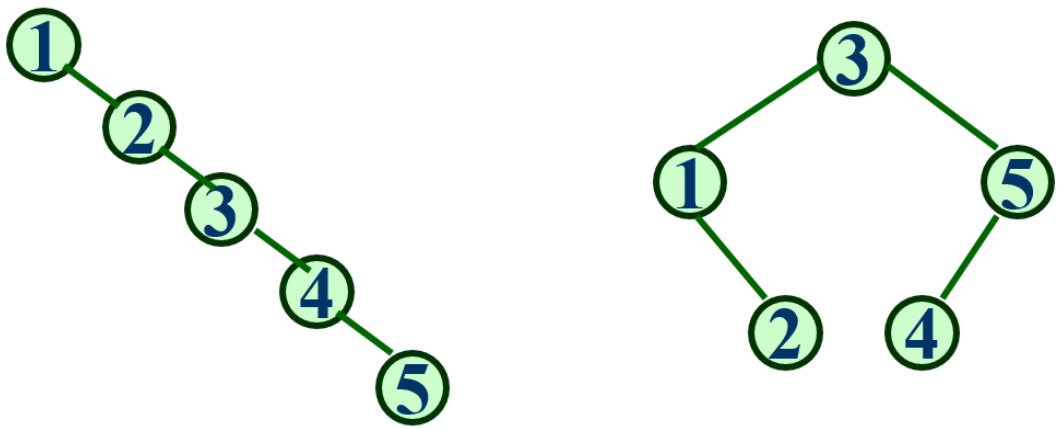
- 方案一：用被删除节点的左孩子替代它，然后合并左孩子的右孩子和左孩子的右兄弟（将右兄弟挂成被删除节点的前驱的右孩子）
- 方案二：用被删除节点的左子树的最右/最大节点（中序遍历序列中被删除节点的前驱）来替换被删除节点
- 两个方案的核心都在于找到被删除节点的前驱节点，如图中的节点 S
  q=p; s=p->lchild; while(s->rchild) {q=s; s=s->rchild;}



二叉排序树的构造

给定一个关键字输入序列，求二叉排序树

- 假定关键字输入序列的关键字被按给定次序，从空树开始，逐一添加到二叉排序树中，生成的二叉排序树是唯一的（为什么?）
- 由关键字序列 1，2，3，4，5 构造而得的二叉排序树，如左图，等概查找时，其 $ASL = 3$
- 有关键字序列 3，1，2，5，4 构造而得的二叉排序树，如右图，等概查找时，其 $ASL = 2.2$



二叉排序树操作的性能

增加和删除操作的关键在于查找的性能

- 插入和删除都的开销都集中在查找操作
- n 个关键字，可构造不同形态的二叉排序树，其平均查找长度可能会不同，甚至可能差别很大
- 共有 $n!$ 种不同的输入序列，每个输入序列对应一棵二叉排序树，假设每棵二叉排序树中每个关键字被等概查找，那么得到平均查找长度为：$2\frac{n+1}{n}\log_2 n + C = O(\log n)$

二叉排序树练习题

题目一：设二叉排序树中的关键字互不相同，则

- 最小元素无左孩子，最大元素无右孩子，此命题是否正确?
- 最大和最小元素一定是叶子结点吗?
- 一个新结点总是插入在叶子结点上吗?

题目二：构建二叉排序树并遍历

- 将关键字序列 (10, 2, 26, 4, 18, 24, 21, 15, 8, 23, 5, 12, 14) 依次插入到初态为空的二叉排序树中，请画出所得到的树 T
- 然后画出删除 10 之后的二叉排序树 T1
- 若再将 10 插入到 T1 中得到的二叉排序树 T2 是否与 T 相同?
- 请给出 T2 的先序、中序和后序序列

平衡二叉排序树

引入的原因

- 二叉排序树是一种查找效率比较高的组织形式，但其平均查找长度受树的形态影响较大，形态比较均匀时查找效率很好，形态明显偏向某一方向时其效率就大大降低
- 因此，希望有更好的二叉排序树，其形态总是均衡的，查找时能得到最好的效率，这就是平衡二叉排序树

平衡二叉树的定义

- Balanced Binary Tree 或 Height-Balanced Tree 是在 1962 年由 Adelson-Velskii 和 Landis 提出的，又称 AVL 树
- 平衡二叉树或者是空树，或者是满足下列性质的二叉树：
  1. 左子树和右子树深度之差的绝对值不大于 1
  2. 左子树和右子树也都是平衡二叉树

平衡二叉排序树

- 如果一棵二叉树既是二叉排序树又是平衡二叉树，称为平衡二叉排序树 (Balanced Binary Sort Tree)

平衡二叉排序树的术语与性质

平衡因子/Balance Factor

- 称二叉树上结点左子树的深度减去右子树深度为结点的平衡因子
- 平衡二叉树上每个结点的平衡因子只可能是-1、0 和 1，否则，只要一个结点的平衡因子的绝对值大于 1，该二叉树就不是平衡二叉树

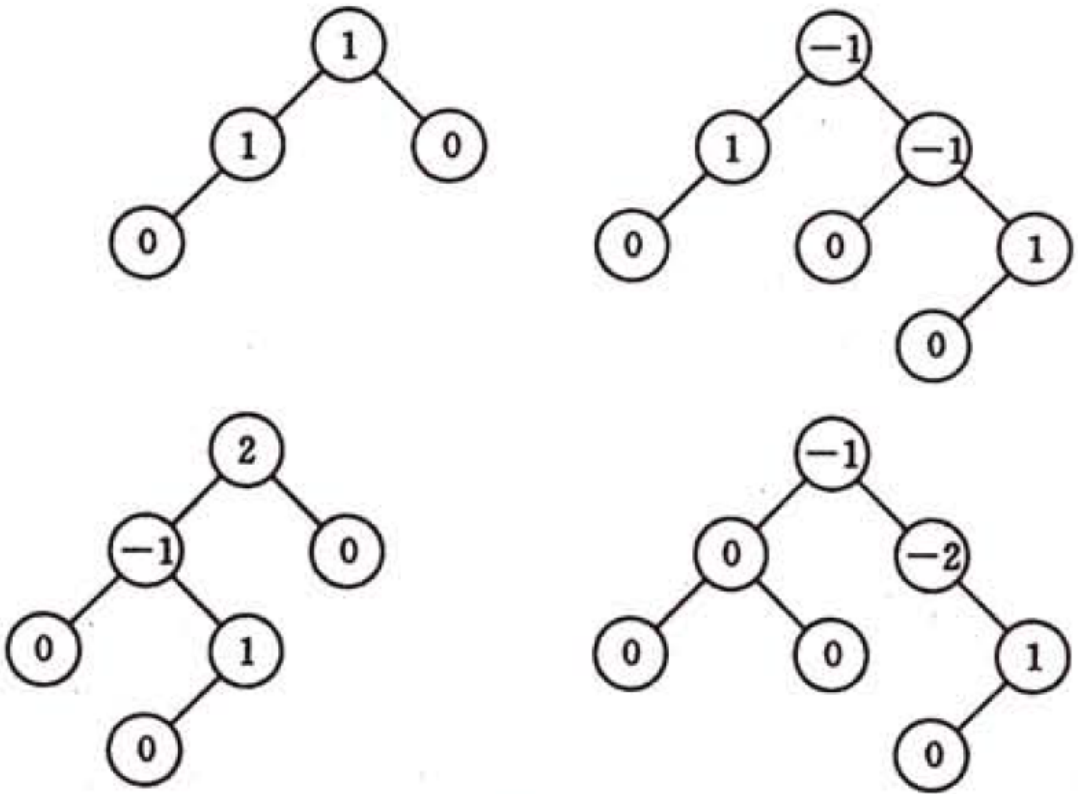
性质

- 在平衡二叉排序树上执行查找的过程与二叉排序树上的查找过程完全一样，和给定的 Key 值比较的次数不超过树的深度
- 设深度为 h 的平衡二叉排序树所具有的最少结点数 N_h，则 N_h 满足递推关系：$N_0 = 0, N_1 = 1, N_2 = 2, \ldots, N_h = N_{h-1} + N_{h-2}$，类似斐波那契数（有关系式：$N_h = F_{h+2} - 1$），可解得 $h \approx \log_{\Phi}(\sqrt{5}(n+1)) - 2$，其中 $\Phi = \frac{\sqrt{5}+1}{2}$，故
- 在平衡二叉排序树上的平均查找长度和 $\log_2 n$ 是一个量级的，也是 $O(\log n)$

平衡二叉树的例子

通过平衡因子来判定是否是平衡二叉树

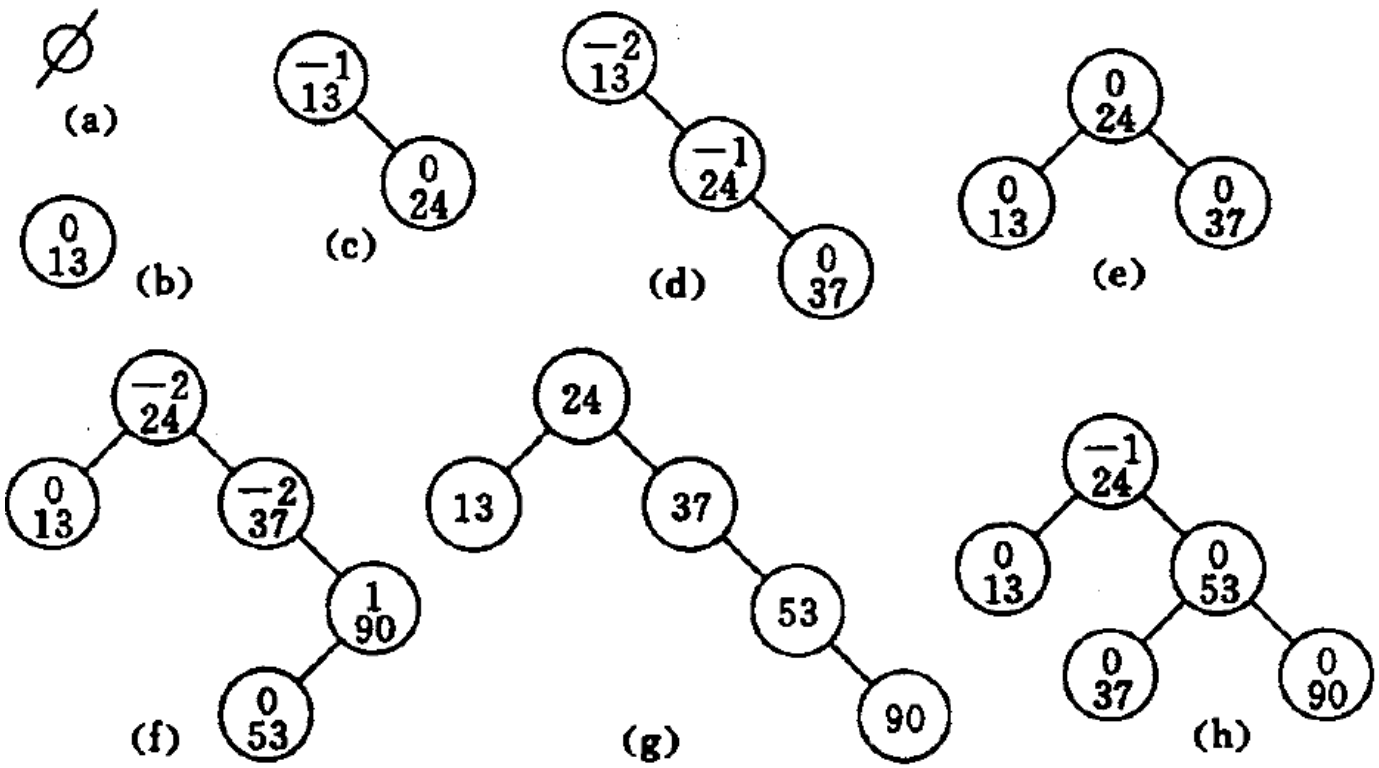
- 如图所示，每个节点的平衡因子都标在节点上



构造平衡的二叉排序树

思路：适当修改二叉排序树的构造过程

- 给定输入序列，当新增一个节点后，检查二叉树的平衡性；若二叉树不平衡了，那么执行一个“调整”过程，将二叉树修改成平衡的
- 从插入的叶节点开始，往上到根节点，找到不平衡的节点就调整成平衡的
- 如图所示的例子：在不平衡的节点上应用逆时针和顺时针旋转



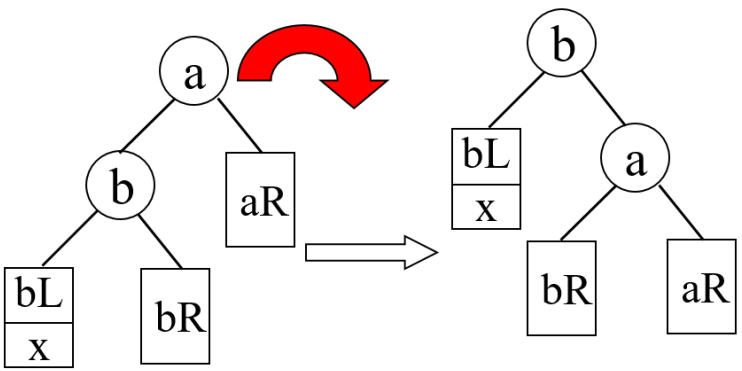
平衡化旋转（LL）

失衡节点

- 沿着插入或删除结点上行到根结点就能找到受到影响的结点，这些结点的平衡因子和子树深度都可能会发生变化，其中平衡因子绝对值大于 1 的结点称为失衡结点

情形一：LL 型平衡化旋转

- 失衡原因：失衡节点 a 初始时刻平衡因子为 1，在结点 a 的左孩子 b 的左子树上进行插入 x，使得 a 的平衡因子变成了 2。如图所示
- 旋转方法：顺时针旋转，用 a 的左孩子 b 取代 a 的位置，a 成为 b 的右子树的根结点，b 原来的右子树作为 a 的左子树



LL型平衡化旋转示意图

平衡因子的变化

- 插入前，a 为 1，b 为 0，bL,aR 和 bR 深度相同
- 插入后，b 为 1，a 为 2
- 旋转后，a,b 都为 0

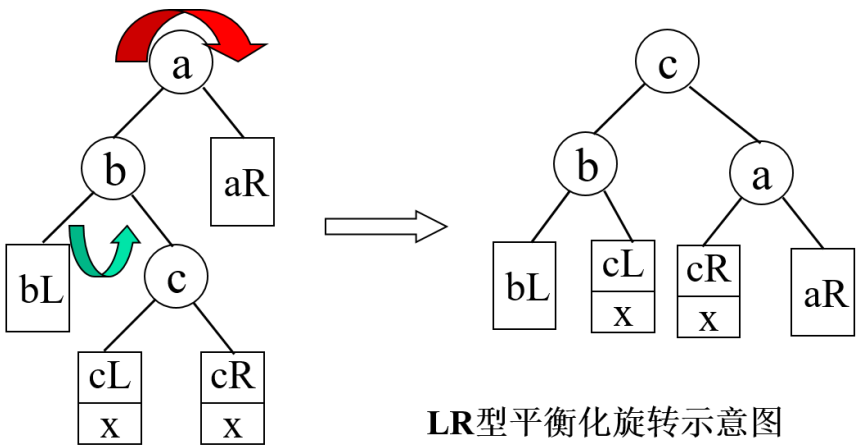
平衡化旋转的实现（LL）

```
1   typedef struct BNode{
2    KeyType key; //关键字域
3    int Bfactor; //平衡因子域
4    …    //其它数据域
5    struct BNode *Lchild, *Rchild;
6  }BSTNode;
7
8  void LL_rotate(BSTNode *a){
9    BSTNode *b;
10   b=a->Lchild;
11   a->Lchild=b->Rchild;
12   b->Rchild=a;
13   a->Bfactor=b->Bfactor=0;
14   a=b;
15 }
```

平衡化旋转（LR）

情形二：LR 型平衡化旋转

- 失衡原因：失衡节点 a 初始时刻平衡因子为 1，在节点 a 的左孩子 b 的右子树上插入 x，使得 a 的平衡因子变成 2，如图所示
- 旋转方法：先逆时针旋转 a 的左子树，再顺时针旋转 a 为根的子树



LR型平衡化旋转示意图

平衡因子的变化，如图所示

- 插入前，a 为 1，b,c 都为 0，aR,bL 深度相同，cL,cR 的深度相同且比 bL 小 1
- 插入后，c 为 0(c 就是插入的叶子节点),1 或-1，b 为-1，a 为 2
- 旋转后：(a,b,c) 的平衡因子可能值为 (-1,0,0),(0,1,0),(0,0,0)

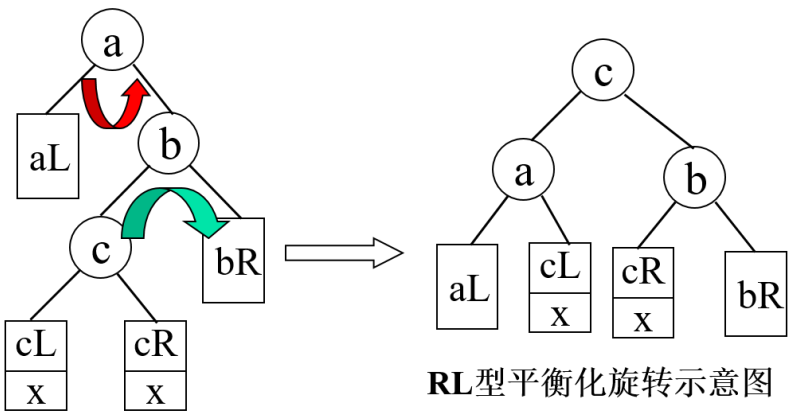
平衡化旋转的实现（LR）

```
1  void LR_rotate(BSTNode *a){
2    BSTNode *b,*c;
3    b=a->Lchild;
4    c=b->Rchild; //初始化
5    a->Lchild=c->Rchild;
6    b->Rchild=c->Lchild;
7    c->Lchild=b;
8    c->Rchild=a;
9    if (c->Bfactor==1){
10     a->Bfactor=-1;
11     b->Bfactor=0;
12     c->Bfactor=0;
13   }
14   else if (c->Bfactor==0)
15         a->Bfactor=b->Bfactor=0;
16       else {
17         a->Bfactor=0;
18         b->Bfactor=1;
19         c->Bfactor=0;
20       }
21 }
```

平衡化旋转（RL）

情形三：RL 平衡化旋转

- 失衡原因：失衡节点 a 初始时刻平衡因子为-1，在结点 a 的右孩子 b 的左子树上插入 x，使得 a 的平衡因子变成-2，如图所示
- 旋转方法：先顺时针旋转 a 的右子树，再逆时针旋转 a 为根的子树



RL型平衡化旋转示意图

平衡因子的变化，如图所示

- 插入前：a 为-1，b,c 都为 0；aI,bR 深度相同，cL,cR 的深度相同且比 bR 小 1
- 插入后：c 为 0(c 就是插入的叶子节点),-1 或 1,b 为 1，a 为-2
- 旋转后：(a,b,c) 的平衡因子可能值为 (1,0,0),(0,-1,0),(0,0,0)

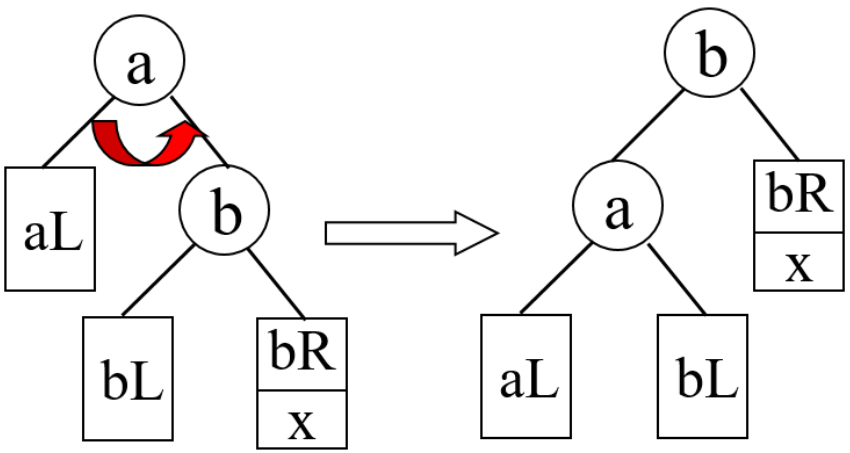
平衡化旋转的实现（RL）

```
1  Void RL_rotate(BSTNode *a){
2    BSTNode *b,*c;
3    b=a->Rchild;
4    c=b->Lchild; //初始化
5    a->Rchild=c->Lchild;
6    b->Lchild=c->Rchild;
7    c->Lchild=a;
8    c->Rchild=b;
9    if (c->Bfactor==1){
10     a->Bfactor=0;
11     b->Bfactor=-1;
12     c->Bfactor=0;
13   }else if (c->Bfactor==0)
14           a->Bfactor=b->Bfactor=0;
15         else {
16           a->Bfactor=1;
17           b->Bfactor=0;
18           c->Bfactor=0;
19         }
20 }
```

平衡化旋转（RR）

情形三：RL 平衡化旋转

- 失衡原因：失衡节点 a 初始时刻平衡因子为-1，在结点 a 的右孩子 b 的右子树上进行插入 x，使得 a 的平衡因子变成了-2。如图所示
- 旋转方法：逆时针旋转，用 a 的右孩子 b 取代 a 的位置，a 成为 b 的左子树的根结点，b 原来的左子树作为 a 的右子树



RR型平衡化旋转示意图

平衡因子的变化，如图所示

- 插入前，a 为-1，b 为 0，aL,bR 和 bL 深度相同
- 插入后，b 为 -1，a 为 -2
- 旋转后，a,b 都为 0

平衡化旋转的实现（RR）

```
1  BSTNode *RR_rotate(BSTNode *a){
2    BSTNode *b;
3    b=a->Rchild;
4    a->Rchild=b->Lchild;
5    b->Lchild=a;
6    a->Bfactor=b->Bfactor=0;
7    a=b;
8  }
```

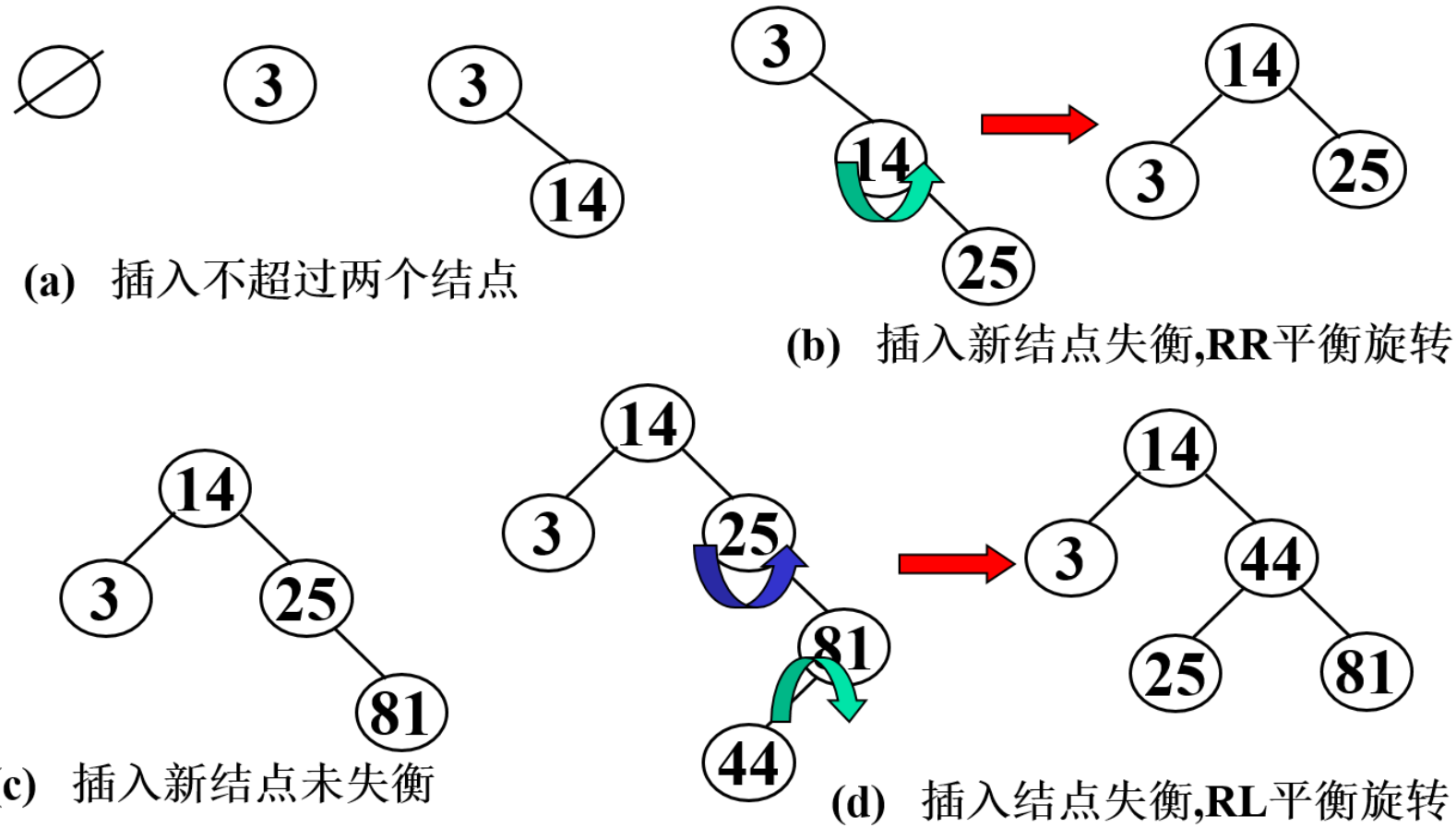
平衡化旋转算法的评述

- 上述四种平衡化旋转，其正确性容易由“遍历所得中序序列不变”来证明
- 无论是哪种情况，平衡化旋转处理完成后，形成的新子树仍然是平衡二叉排序树，且其深度和插入前以 a 为根结点的平衡二叉排序树的深度相同
- 所以，在平衡二叉排序树上因插入结点而失衡，仅需对失衡子树做平衡化旋转处理

平衡二叉排序树的构造

基于二叉排序树的构造

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



平衡二叉树的构造过程

二叉排序树练习题

题目说明

- 设有关键字序列为：(Dec, Feb, Nov, Oct, June, Sept, Aug, Apr, May, July, Jan, Mar)
- 请手工构造一棵二叉排序树
- 该树是平衡二叉排序树？若不是，请为其构造一棵平衡二叉排序树

改进索引技术

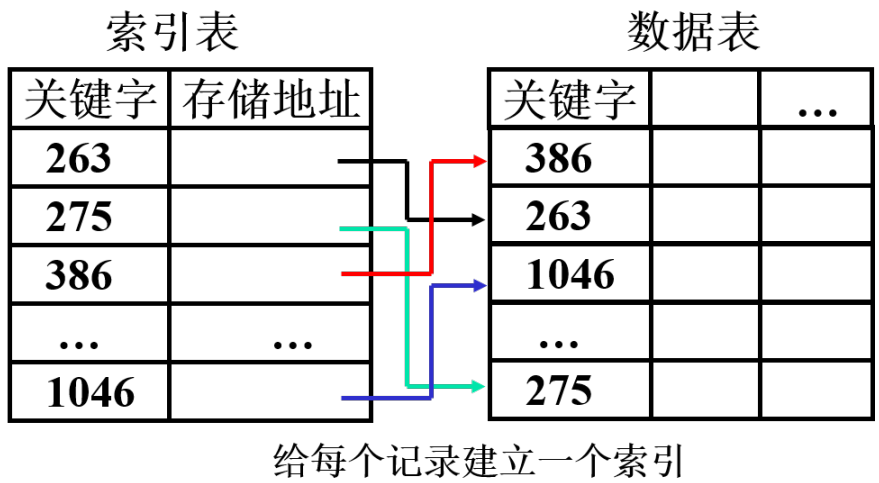
索引顺序表：将数据表分块，块间有序，块内无序，块有索引

- 如果每个记录都建立索引，即块的大小为 1，这就给数据建立了“稠密的”索引表
- 索引表有序，可以用折半查找或平衡二叉树来构建索引表

如右图，为每个记录建立索引

适用于以下应用场景：

- 记录大小不固定，或者每个记录都很大，比如 10MB
- 记录存储在外存，内存放不下，比如数据库文件
- 增删记录的操作非常频繁时，不移动数据，仅更新索引表



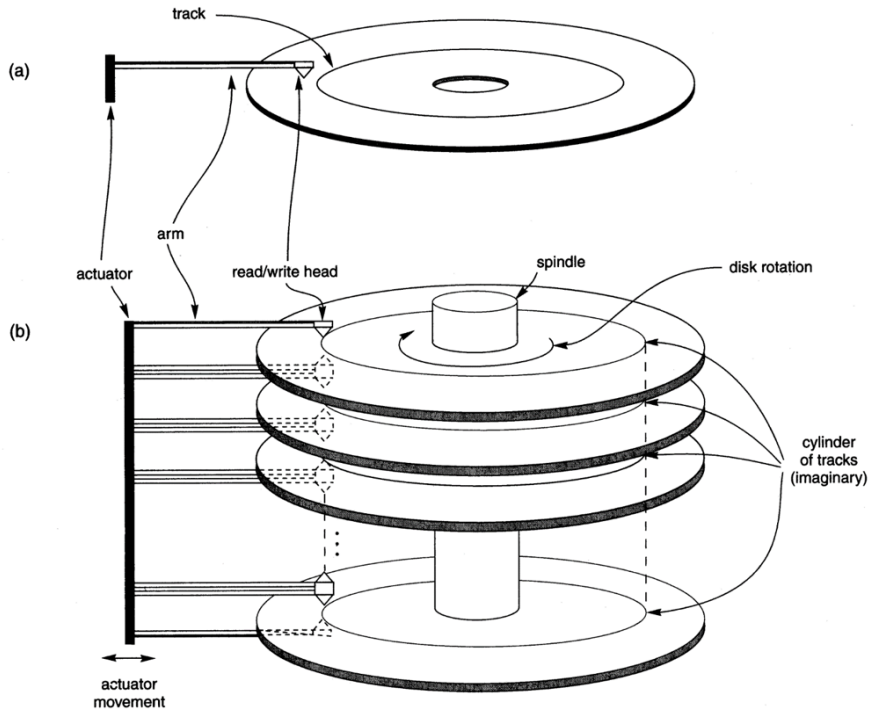
存在问题：索引频繁更新，索引在内存中也放不下，怎么办？

树形索引表

问题的提出

- 平衡二叉排序树便于动态查找，因此用平衡二叉排序树来组织索引表是一种可行的选择
- 当用于大型数据库时，所有数据及索引都存储在外存，因此，涉及到内、外存之间频繁的数据交换，这种交换速度的快慢成为制约动态查找的瓶颈
- 若以二叉树的结点作为内、外存之间数据交换单位，则查找给定关键字时对磁盘平均进行 $O(\log_2 n)$ 次访问是不能容忍的
- 因此，必须选择一种能尽可能降低磁盘 I/O 次数的索引组织方式
- 树结点的大小尽可能地接近页的大小
- R.Bayer 和 E.Mc Creight 在 1972 年提出了一种多路平衡查找树，称为 B-树 (其变型体是 B+ 树)

磁盘结构



磁盘的简化模型

- 磁盘是很多块 (block) 构成的集合
- 每个块有自己的“地址”，用于查找给定地址的块
- 每个磁盘块的大小都是固定的，格式化的时候设定，比如 512Bytes，1024Bytes，...
- 磁盘块是磁盘读取的基本单位

B-树

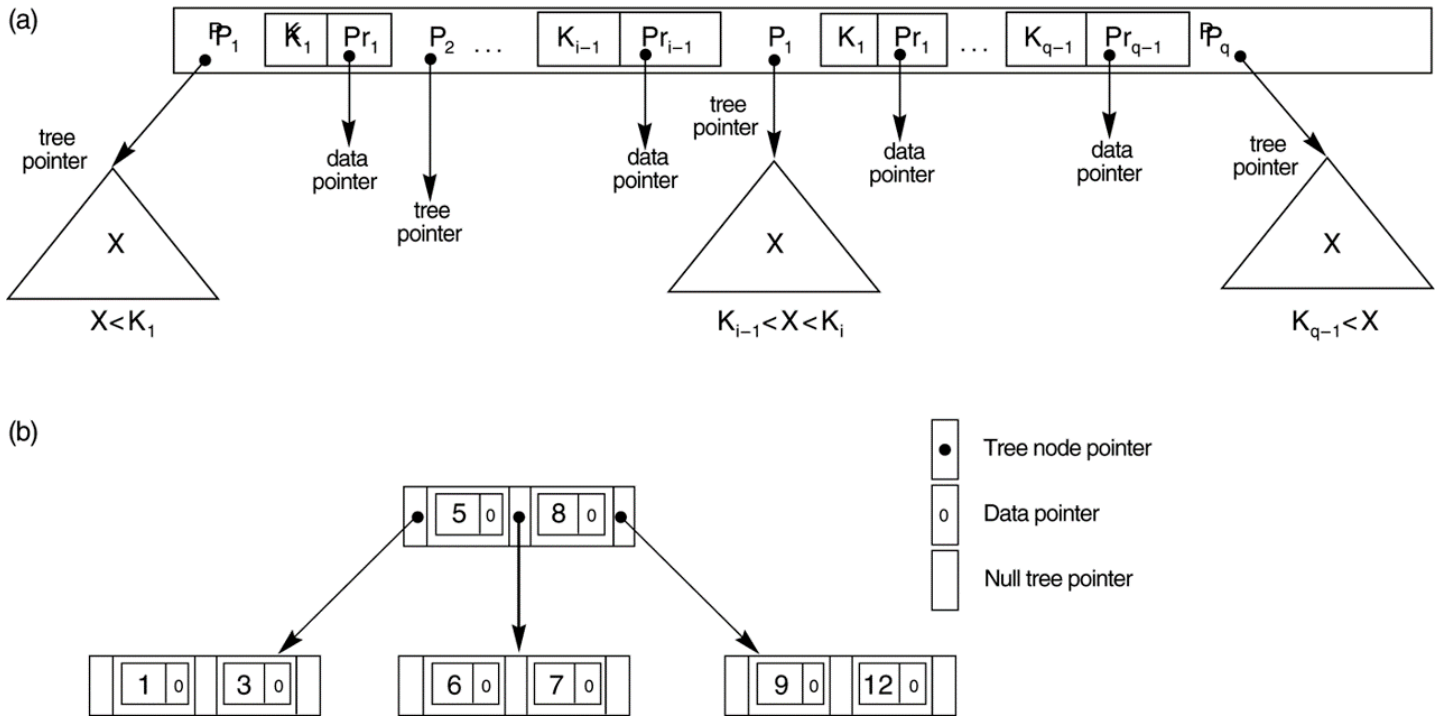
B-树应用场景

- B-树主要用于文件系统中，在 B-树中，每个结点的大小为一个磁盘页，结点中所包含的关键字及其孩子的数目取决于页的大小

B-树的定义

- 一棵 m 阶 B-树，或者是空树，或者是满足以下性质的 m 叉树：
  - 根结点或者是叶子，或者至少有两棵子树，至多有 m 棵子树
  - 除根结点外，所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，至多 m 棵子树
  - 所有叶子结点都在树的同一层上
  - 每个结点应包含如下信息：$(n, A_0, K_1, A_1, K_2, A_2, \ldots, K_n, A_n)$，其中 $K_i, 1 \leq i \leq n$ 是关键字，且 $K_i < K_{i+1}, 1 \leq i \leq n-1$，$A_i, i = 0, 1, \ldots, n$ 为指向孩子结点的指针，且 A_{i-1} 所指向的子树中所有结点的关键字都小于 K_i，A_i 所指向的子树中所有结点的关键字都大于 K_i；n 是结点中关键字的个数，且 $\lceil m/2 \rceil - 1 \leq n \leq m-1, n+1$ 为子树的棵数
  - 当然，在实际应用中每个结点中还应包含 n 个指向每个关键字的记录指针

B-树的图解



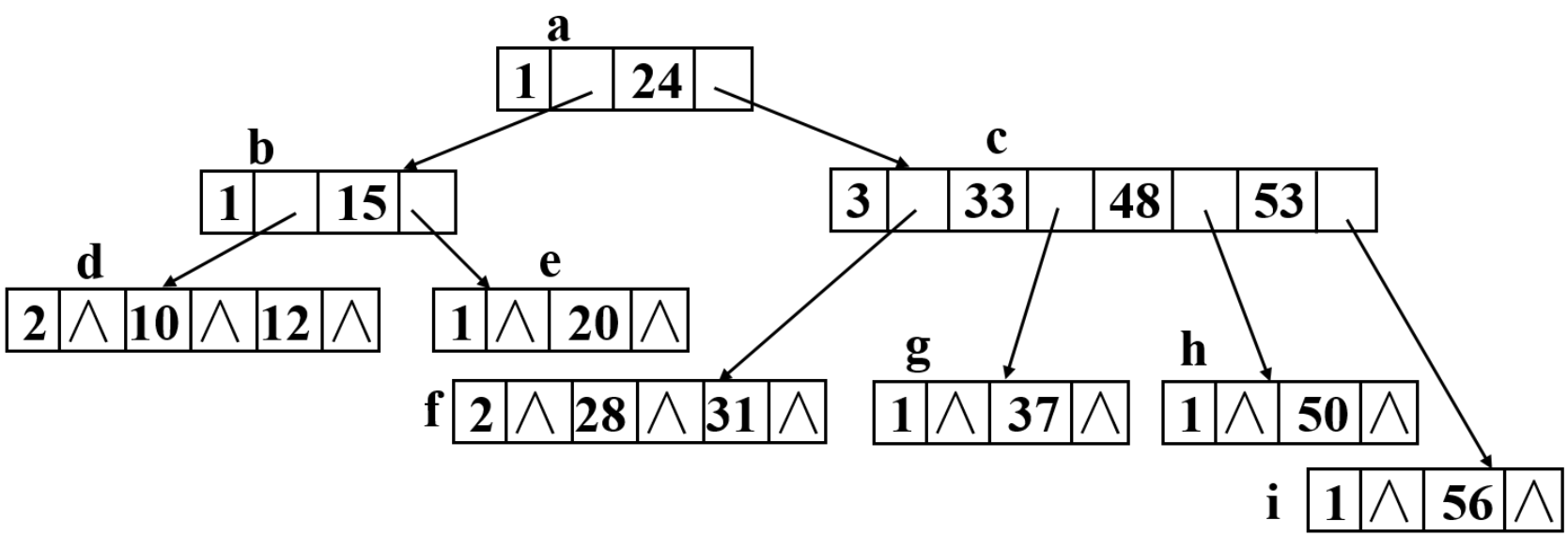
说明

- (a) 是一般性的 q 阶 B-树，(b) 是一个 3 阶 B-树的例子
- q 个指向下一层的指针，$q-1$ 个索引项
- 每个索引项都指向一个数据记录 (结构体，集合中待查找的元素)
- q 阶 B-树是二叉排序树简单的扩展为 q 叉排序树（根节点，叶节点，每个节点的非空指针数另有要求)

B-树的例子

一棵包含 13 个关键字的 4 阶 B-树

- 每个节点至少有 $\lceil m/2 \rceil = 2$ 棵子树，至多 $m = 4$ 棵子树
- 叶节点都在第三层
- 指针指向的子树中所有节点的关键字在指针左右的的关键字给出的范围内



B-树的存储结构

根据 m 阶 B-树的定义，结点的类型定义如下

```
1  #define M 5 //根据实际需要定义B-树的阶数
2  typedef struct BTNode{
3    int keynum;//结点中关键字的个数
4    struct BTNode *parent;//指向父结点的指针
5    KeyType key[M];//关键字向量,key[0]未用
6    struct BTNode *ptr[M];//子树指针向量
7    RecType *recptr[M];
8  /*记录指针向量,recptr[0]未用*/
9  }BTNode;
```

B-树的查找

算法思想:B-树上的查找过程和二叉排序树的查找相似

1. 从树的根结点 T 开始，在 T 所指向的结点的关键字向量 $key[1 \ldots keynum]$ 中查找给定值 K(用折半查找)：
   - 若 $key[i] = K(1 \leq i \leq keynum)$，则查找成功，返回结点及关键字位置
   - 否则，转步骤 2
2. 将 K 与向量 $key[1 \ldots keynum]$ 中的各个分量的值进行比较，以选定查找的子树：
   - 若 $K < key[1] : T = T- > ptr[0]$
   - 若 $key[i] < K < key[i+1](i = 1, 2, \ldots, keynum - 1)$，那么 $T = T- > ptr[i]$;
   - 若 $K > key[keynum] : T = T- > ptr[keynum]$
3. 转步骤 1，直到 T 是叶子结点且未找到相等的关键字，则查找失败

B-树查找的实现

代码实现

```
1  int BT_search(BTNode *T, KeyType K, BTNode *p){
2   //在B-树中查找关键字K, 查找成功返回在结点中的位置
3   //及结点指针p;否则返回0及最后一个结点指针
4    BTNode *q; int n;
5    p=q=T;
6    while(q!=NULL){
7      p=q;
8      q->key[0]=K;//设置查找哨兵
9      for(n=q->keynum;K<=q->key[n];n--)
10       if (n>0&&EQ(q->key[n],K))
11         return n;
12     q=q->ptr[n];
13   }
14   return 0;
15 }
```

B-树查询性能分析

影响 B-树查找性能的关键因素

- B-树的查找分为找节点和在节点中找关键字；而 B-树通常存储在磁盘上，故找节点是对外存数据处理；在节点中找关键字是对内存操作，故找节点是影响性能的主要因素
- 一个节点通常是一个磁盘块/一次磁盘访问，访问节点数越少越好
- 而访问节点数就是从根开始，关键字所在节点的深度

节点数目和关键字所在节点的深度的关系

- 设被查询关键字在 B-树的深度为 h
- 根据 m 阶 B-树的定义，第一层上至少有 1 个结点，第二层上至少有 2 个结点；除根结点外，所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树，...，第 h 层上至少有 $\lceil m/2 \rceil^{h-2}$ 个结点
- 这些节点中，根节点至少包括 1 个关键字，其它节点至少包括 $\lceil m/2 \rceil - 1$ 个关键字，则总关键字数目 n 满足式子：
  $n \geq 1 + (\lceil m/2 \rceil - 1) \sum_{i=2}^{h} 2(\lceil m/2 \rceil - 1)^i = 2(\lceil m/2 \rceil - 1)^{h-1} - 1$
- 即：$h \leq \log_{\lceil m/2 \rceil - 1}(\frac{n+1}{2}) + 1$

B-树的插入

B-树的构造和插入

- B-树的生成也是从空树起，逐个插入关键字
- 插入时不是每插入一个关键字就添加一个叶子结点，而是首先在最低层的某个叶子结点中添加一个关键字，然后有可能“分裂”

插入算法思想

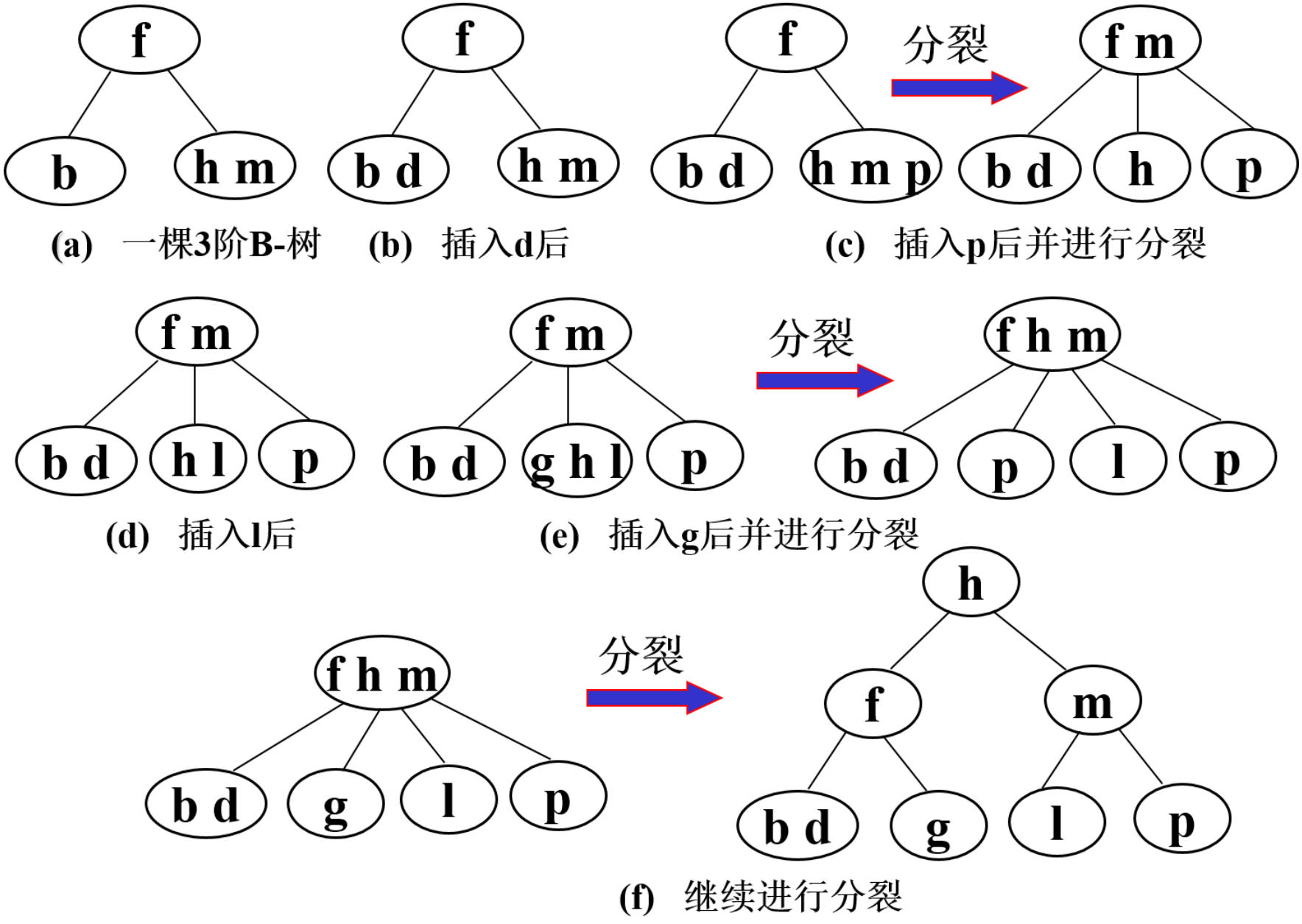
1. B-树的中查找关键字 K，若找到，表明关键字已存在，返回；否则，K 的查找操作失败于某个叶子结点，转步骤 2
2. 将 K 插入到该叶子结点中，插入时，若：
   - 叶子结点的关键字数 $<$m-1：直接插入
   - 叶子结点的关键字数 $=$m-1：将结点“分裂”

B-树的插入

节点分裂方法

- 设待“分裂”结点包含信息为 $(m, A_0, K_1, A_1, K_2, A_2, \ldots, K_m, A_m)$，从其中间位置分为两个结点
  - $(\lceil m/2 \rceil - 1, A_0, K_1, \ldots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$
  - $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, \ldots, K_m, A_m)$
- 将中间关键字 $K_{\lceil m/2 \rceil}$，并以分裂后两个节点作为其左右子节点
- 若父节点被插入新节点后，不满足 m 阶 B-树的定义，那么也需要继续分裂，一直进行下去，直到没有父结点或分裂后的父结点满足 m 阶 B-树的要求
- 当根结点分裂时，因无父结点，故建立一个新的根，B-树增高一层

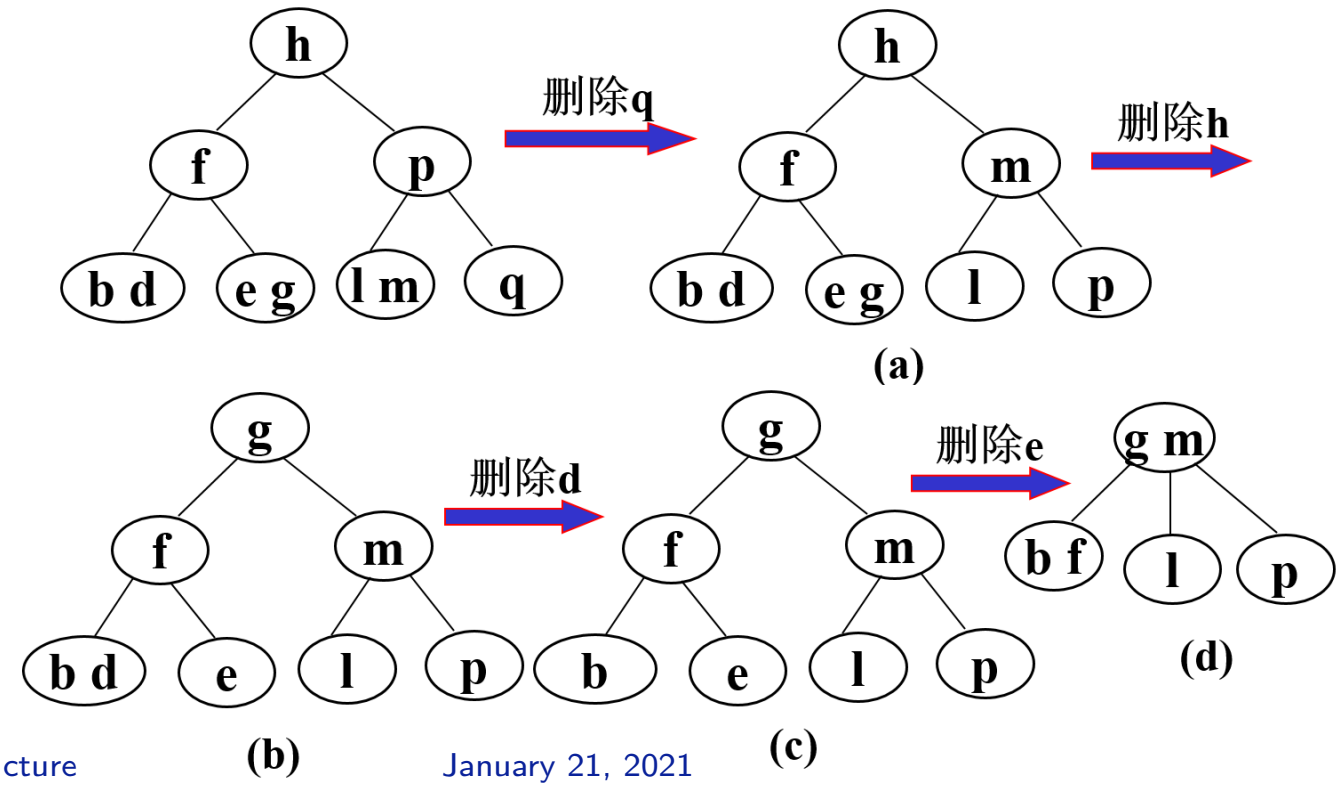
B-树插入的例子


f
b
h m
(a) 一棵3阶B-树
f
b d
h m
(b) 插入d后
f
b d
h m p
分裂
f m
b d
h
p
(c) 插入p后并进行分裂
f m
b d
h l
p
(d) 插入l后
f m
b d
g h l
p
分裂
f h m
b d
p
l
p
(e) 插入g后并进行分裂
f h m
b d
g
l
p
分裂
h
f
m
b d
g
l
p
(f) 继续进行分裂

B-树的删除

删除算法的思想

- 在 B-树上删除一个关键字 K，首先找到关键字所在的结点 N，然后在 N 中进行关键字 K 的删除操作
- 若 N 不是叶子结点，设 K 是 N 中的第 i 个关键字，则将指针 A_{i-1} 所指子树中的最大关键字 (或最小关键字)K' 放在 (K) 的位置，然后删除 K'，而 K' 一定在叶子结点上
- 如图所示，删除关键字 h，用关键字 g 代替 h 的位置，然后再从叶子结点中删除关键字 g



B-树的删除

从叶子节点删除一个关键字

- 若结点 N 中的关键字个数 $> \lceil m/2 \rceil - 1$，则直接删除关键字
- 若结点 N 中的关键字个数 $= \lceil m/2 \rceil - 1$，若结点 N 的左 (右) 兄弟结点中的关键字个数 $> \lceil m/2 \rceil - 1$，则将结点 N 的左 (或右) 兄弟结点中的最大 (或最小) 关键字上移到其父结点中，而父结点中大于 (或小于) 且紧靠上移关键字的关键字下移到结点 N
- 若结点 N 和其兄弟结点中的关键字数 $= \lceil m/2 \rceil - 1$，删除结点 N 中的关键字，再将结点 N 中的关键字、指针与其兄弟结点以及分割二者的父结点中的某个关键字 K_i，合并为一个结点，若因此使父结点中的关键字个数 $< \lceil m/2 \rceil - 1$，则依此类推

B-树插入和删除的算法实现

实现代码考试不做要求

- 课后阅读，提升编程能力

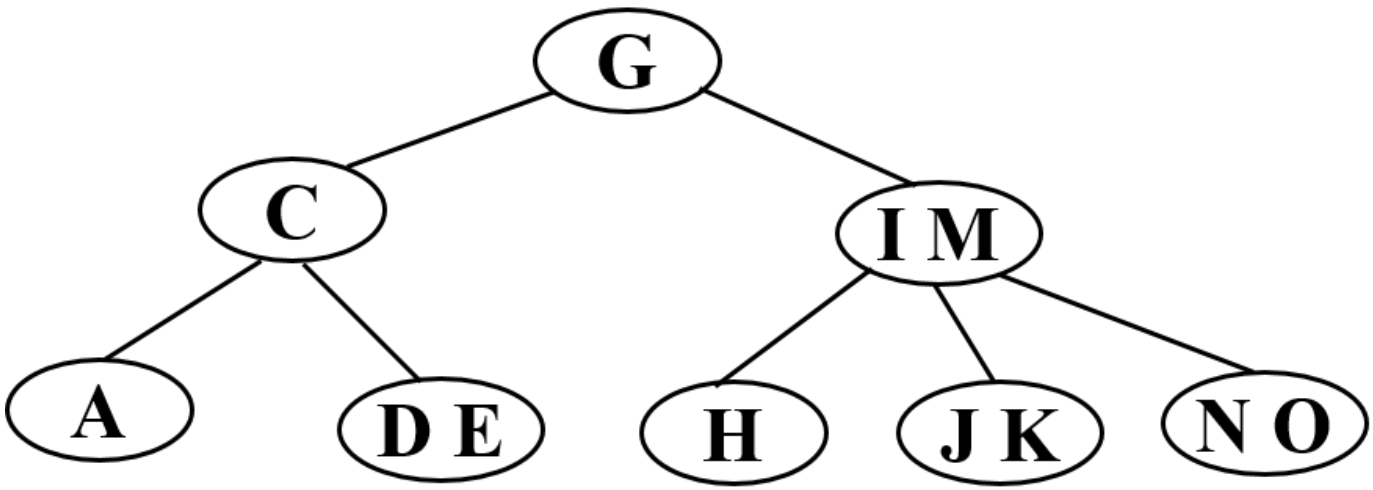
插入和删除的思想

- 先插入或删除
- 再检查是否满足 B-树的定义（额外的要求，根、叶子和非空指针数目）
- 然后在节点和节点的“直系亲属”（双亲，孩子，兄弟）间进行合并和分裂

B-树练习题

题目说明

- 下图是一棵 3 阶 B-树，请画出插入关键字 B,L,P,Q 后的树形



B+ 树

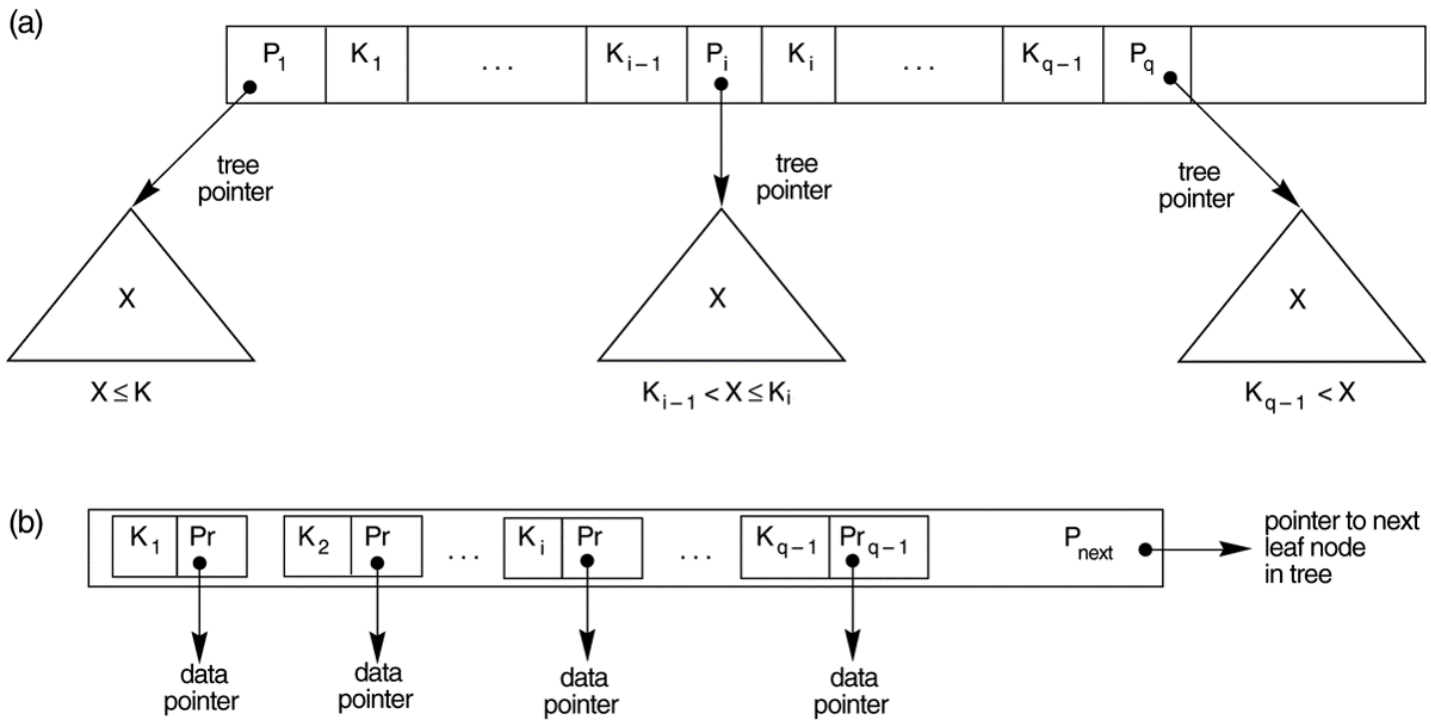
应用场景与 B-树的区别

- 在实际的文件系统中，基本上不使用 B-树，而是使用 B-树的一种变体，称为 m 阶 B+ 树
- 它与 B-树的主要不同是叶子结点中存储记录；而 B-树非叶节点也存储记录关键字及指向记录的指针
- 在 B+ 树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树

B+ 树的定义

- 一棵 m 阶 B+ 树:
  - 若一个结点有 m 棵子树，则必含有 m 个关键字
  - 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接
  - 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大 (或最小) 关键字

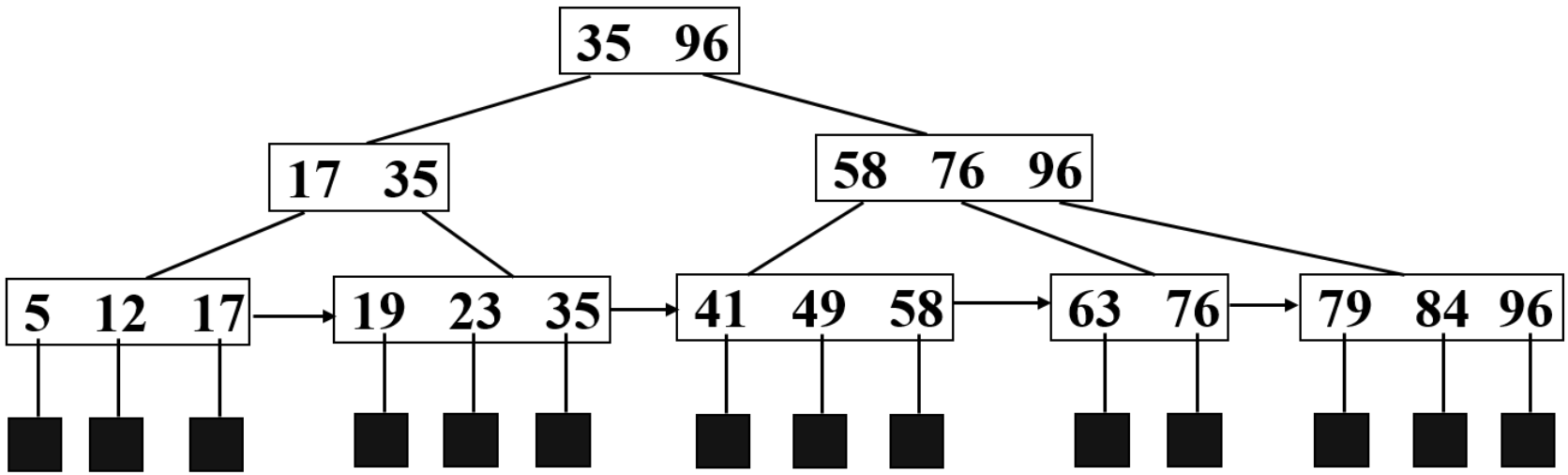
B+ 树图解



说明

- 注意 (a) 和 B-树的图的差异：B+ 树是关键字，不是 B-树的“关键字 + 地址/指针” 的索引项

B+ 树的例子：3 阶 B+ 树



B+ 树的存储结构

叶子节点和非叶节点有明显的不同

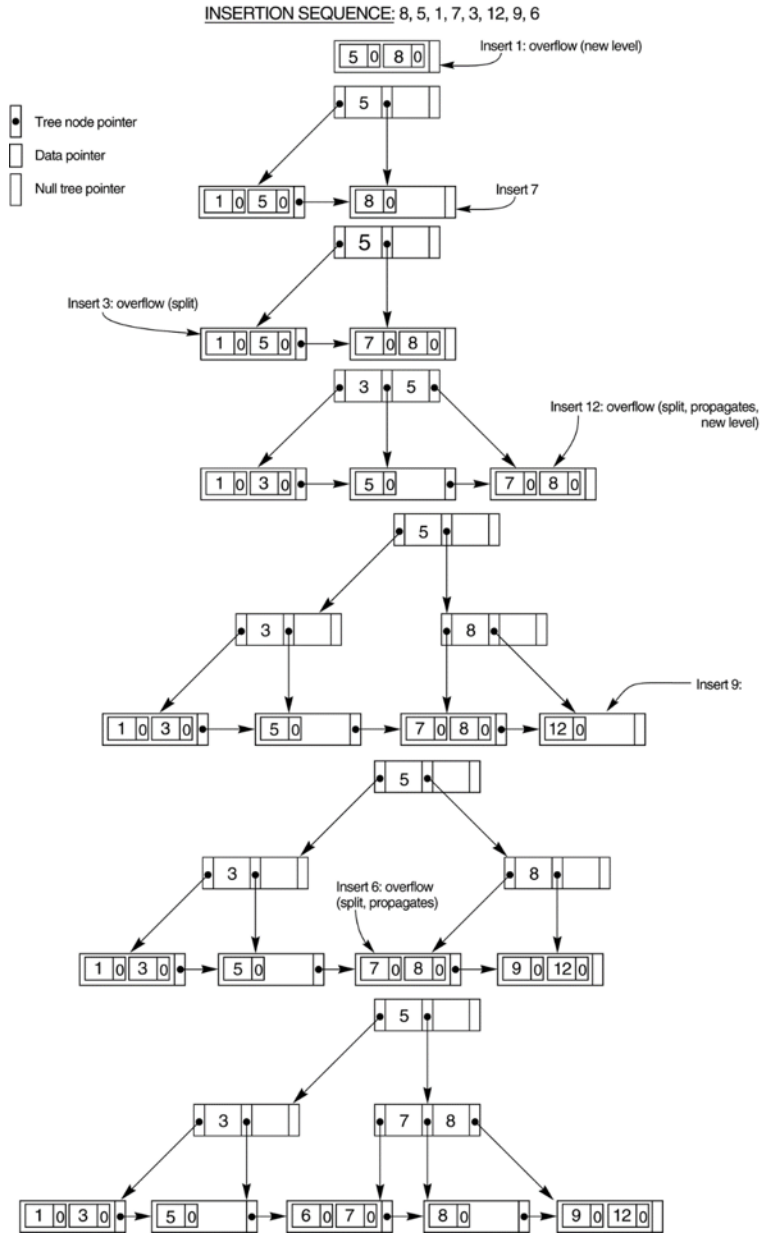
```
1  typedef enum{branch,leave} NodeType;
2  typedef struct BPNode{
3    NodeType tag;//结点标志
4    int keynum;//结点中关键字的个数
5    struct BPNode *parent;//指向父结点的指针
6    KeyType key[M+1];//组关键字向量,key[0]未用
7    union pointer{
8      struct BPNode *ptr[M+1];//子树指针向量
9      RecType *recptr[M+1];//recptr[0]未用
10 }ptrType;//用联合体定义子树指针和记录指针
11 }BPNode;
```

B+ 树的操作

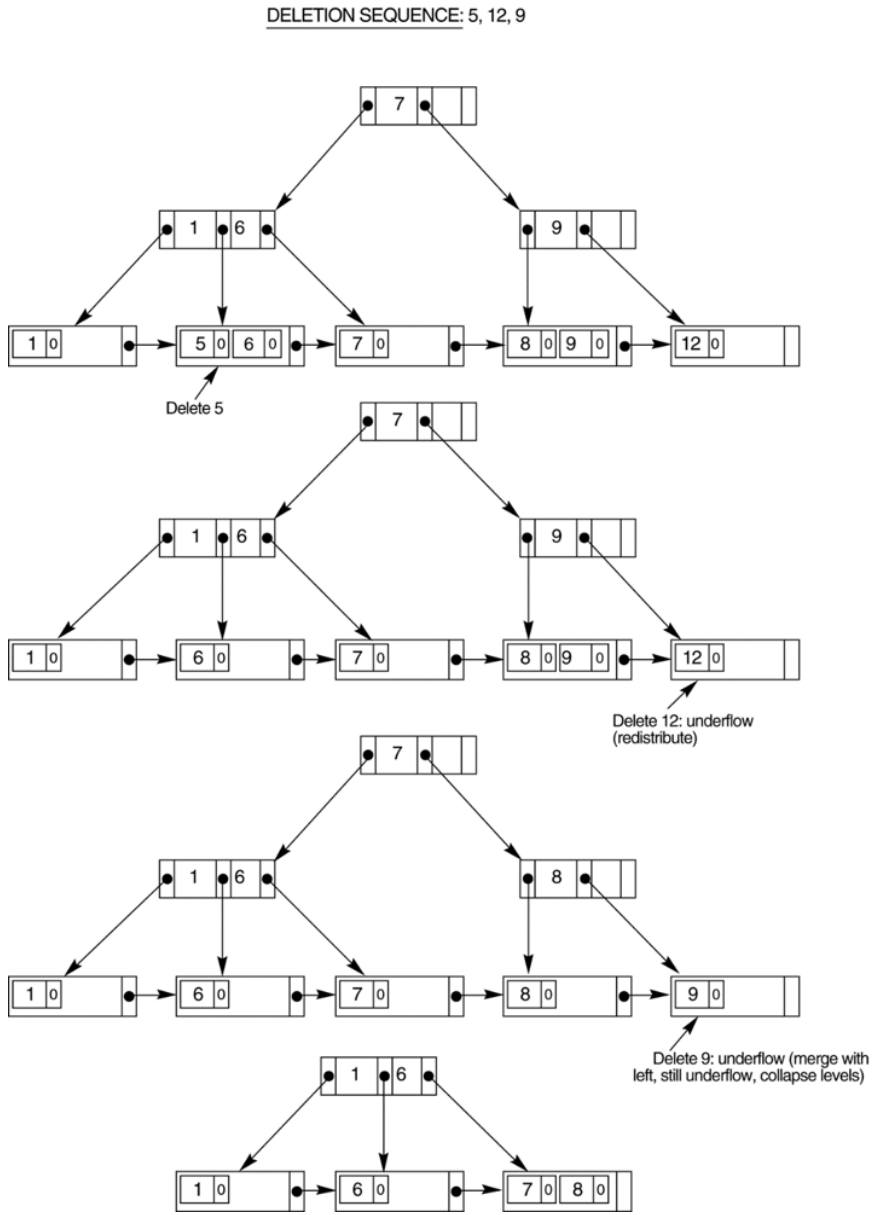
操作都类似 B-树

- 查询的区别：在非叶节点上找到关键字，但是并不停止，而是沿着路径找到叶节点，故查询的耗时都是从根到叶的定长路径
- 插入的区别：仅在叶节点上进行，当叶节点的包含超过 m 个关键字，则分裂，并在双亲节点中包含分裂后两个节点的最大关键字及其指针
- 删除的区别：仅在叶节点进行，当叶节点的最大关键字被删除，其双亲的最大关键字可当成是“分界关键字”依然存在；叶节点过小时 $(< \lceil m/2 \rceil)$, 需要类似 B-树进行节点合并

B+ 树的插入操作例子


INSERTION SEQUENCE: 8, 5, 1, 7, 3, 12, 9, 6
Insert 1: overflow (new level)
Tree node pointer
Data pointer
Null tree pointer
Insert 7
Insert 3: overflow (split)
Insert 12: overflow (split, propagates, new level)
Insert 9:
Insert 6: overflow (split, propagates)

B+ 树的删除操作例子


DELETION SEQUENCE: 5, 12, 9
Delete 5
Delete 12: underflow
(redistribute)
Delete 9: underflow (merge with
left, still underflow, collapse levels)

什么是哈希表与哈希查找

哈希函数/Hash

- 哈希函数是一种从关键字空间到存储地址空间的一种映射，即：
- $Addr(a) = H(key)$: 在记录的关键字与记录的存储地址之间建立的一种对应关系，其中 a 是一个记录，$Addr(a)$ 是 a 的存储地址，key 是 a 的关键字，$H(\ \cdot\ )$ 是哈希函数

哈希表

- 根据设定的哈希函数 $H(key)$，将一组关键字映射到一个有限的、地址连续的地址集 (区间) 上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”

哈希查找

- 也叫散列查找，依据给定的关键字 key 和哈希函数，直接算出记录存储的地址，不经过比较，一次存取就能得到要查找的记录

哈希表的例子

30 个地区的各民族人口统计表

编号	地区别	总人口	汉族	回族.....
1	北京			
2	上海			
⋮	⋮			

以编号作关键字，
构造哈希函数：H(key)=key
H(1)=1
H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19
H(Shenyang)=19

哈希函数/哈希表存在的问题

两个不同关键字计算出相同的哈希地址

- 这称之为“冲突”，需要设计一个函数，将所有不同 key 映射到不同的地址
- 若设计不出这样的函数，那么就需要设计冲突解决办法

哈希表存在很多空白的空间没有存放记录

- 存放在内存种的哈希表需要多大的空间?
- 空间太大，可能很多没存储记录，浪费了；空间太小，可能产生大量的冲突

两个问题是相关的，彼此影响

常见哈希函数

从 key 计算存储地址

- 直接定址法
- 平方取中法
- 随机数法
- 数字分析法
- 折叠法
- 除留余数法

直接定址法、平方取中法和随机数法

直接定址法：哈希函数为关键字的线性函数

- $H(key) = key$ 或者 $H(key) = a \times key + b$
- 此法仅适合于：地址集合的大小等于关键字集合的大小，其中 a, b 为常数

平方取中法

- 以关键字的平方值的中间几位作为存储地址
- 求“关键字的平方值”的目的是“扩大差别”，同时平方值的中间各位又能受到整个关键字中各位的影响
- 此方法适合于：关键字中的每一位都有某些数字重复出现频度很高的现象，或不知到关键字频率分布情况时

随机数法

- 设定哈希函数为：$H(key) = Random(key)$，其中，$Random$ 为以 key 为随机数种子的伪随机函数
- 此方法适合于：对长度不等的关键字构造哈希函数

数字分析法

数字分析法: 寻找关键字的数字特征

- 假设关键字集合中的每个关键字都是由 s 位数字组成 $(u_1, u_2, \ldots, u_s)$
- 分析所有关键字，从中提取分布均匀的若干位或它们的组合作为地址
- 此法适于能预先估计出全体关键字的每一位上各种数字出现的频度

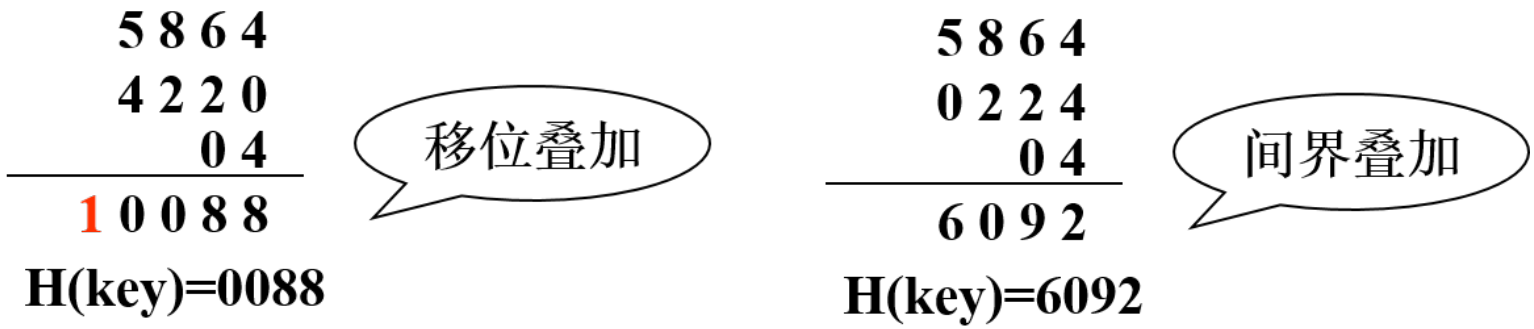
例子：有 80 个记录，关键字为 8 位十进制数，哈希地址为 2 位十进制数

折叠法

折叠法：叠加关键字的不同部分

- 将关键字分割成若干部分，然后取它们的叠加和为哈希地址
- 两种叠加处理的方法：
  - 移位叠加: 将分割后的几部分低位对齐相加
  - 间界叠加: 从一端沿分割界来回折送，然后对齐相加此法适于关键字的数字位数特别多

例子：关键字为：0442205864，哈希地址位数为 4



除留余数法

除留余数法：利用除法的余数

- 设定哈希函数为：$H(key) = key$ MOD $p(p \leq m)$
- 其中，m 为表长，p 为不大于 m 的素数或是不含 20 以下的质因子

为什么要对 p 加限制?

- 例如：给定一组关键字为：12, 39, 18, 24, 33, 21
- 若取 $p = 9$, 则他们对应的哈希函数值将为：3, 3, 0, 6, 6, 3
- 可见，若 p 中含质因子 3, 则所有含质因子 3 的关键字均映射到“3 的倍数”的地址上，从而增加了“冲突”的可能

选取哈希函数

需要考虑的因素

- 计算哈希函数所需时间：不能太复杂，现在通常被忽略
- 关键字长度：不是所有的哈希函数都关注关键字的长度
- 哈希表长度（哈希地址范围)：长度大，冲突少；
- 关键字分布情况：指计算得到的哈希地址分布情况
- 记录的查找频率：频率高的最好一次查到

冲突处理方法

冲突产生的原因

- 哈希函数无法保证不同的关键字得到不同的哈希地址

冲突处理：为产生冲突的地址寻找下一个哈希地址

- 开放定址法
- 再哈希法
- 链地址法
- 建立一个公共溢出区

开放定址法

为冲突的地址 $H(key)$ 准备好一系列备选地址

- 设备选地址序列为：$H_1, H_2, \ldots, H_k, 1 \leq k \leq m-1, m$ 为哈希表长度
- 备选地址的计算公式为：$H_i = (H(key) + d_i)$ MOD $m, i = 1, 2, \ldots, k$
- 其中 d_i 为增量序列，有以下三种常见取法：
  - 线性探测再散列：$d_i = c \times i, c = 1$ 是最简单的情形
  - 平方探测再散列：$d_i = 1^2, -1^2, 2^2, -2^2, \ldots$
  - 伪随机探测再散列：d_i 是一组伪随机数列

对备选地址的要求

- 增量 d_i 应具有“完备性”，即：产生的 H_i 均不相同，且所产生的 $k(\leq m-1)$ 个 H_i 值能覆盖哈希表中所有地址
- 具体来说：
  - 平方探测时的表长 m 必为形如 $4j+3$ 的素数 (如：7, 11, 19, 23, ...等)
  - 随机探测时取决于伪随机数列 (m, d_i 没有公因子)

开放定址法处理冲突的例子 1

设定哈希函数 $H(key) = key$ MOD 11 (表长为 11)

- 给定关键字集合 {19, 01, 23, 14, 55, 68, 11, 82, 36} 构造哈希表

采用线性探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	68	11	82	36	19		

采用平方探测再散列处理冲突

0	1	2	3	4	5	6	7	8	9	10
55	01	23	14	36	82	68		19		11

开放定址法处理冲突的例子 2

设定哈希函数 $H(key) = key$ MOD 11 (表长为 11)

- 哈希表中已填有关键字为 17，60，29 的记录，现有第 4 个记录，其关键字为 38，按三种处理冲突的方法，将它填入表中

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

(1) H(38)=38 MOD 11=5 冲突
H1=(5+1) MOD 11=6 冲突
H2=(5+2) MOD 11=7 冲突
H3=(5+3) MOD 11=8 不冲突

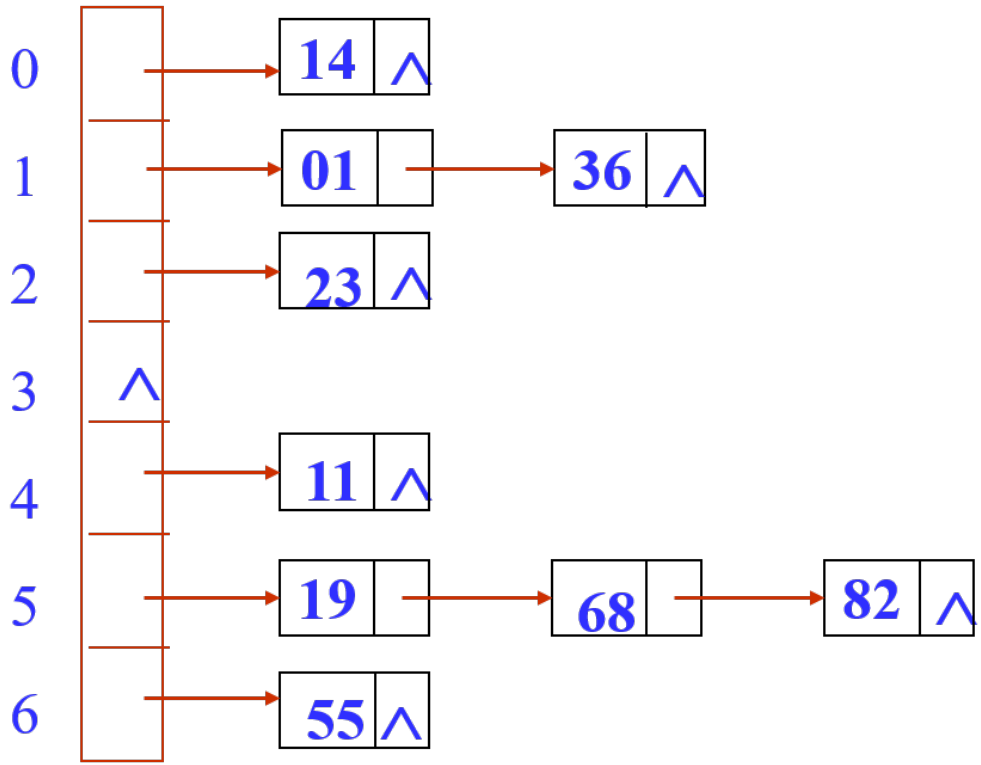
(2) H(38)=38 MOD 11=5 冲突
H1=(5+1²) MOD 11=6 冲突
H2=(5-1²) MOD 11=4 不冲突

(3) H(38)=38 MOD 11=5 冲突, 伪随机数序列为9，则：
H1=(5+9) MOD 11=3 不冲突

链地址法

链地址法：将所有哈希地址相同的记录都链接在同一链表中

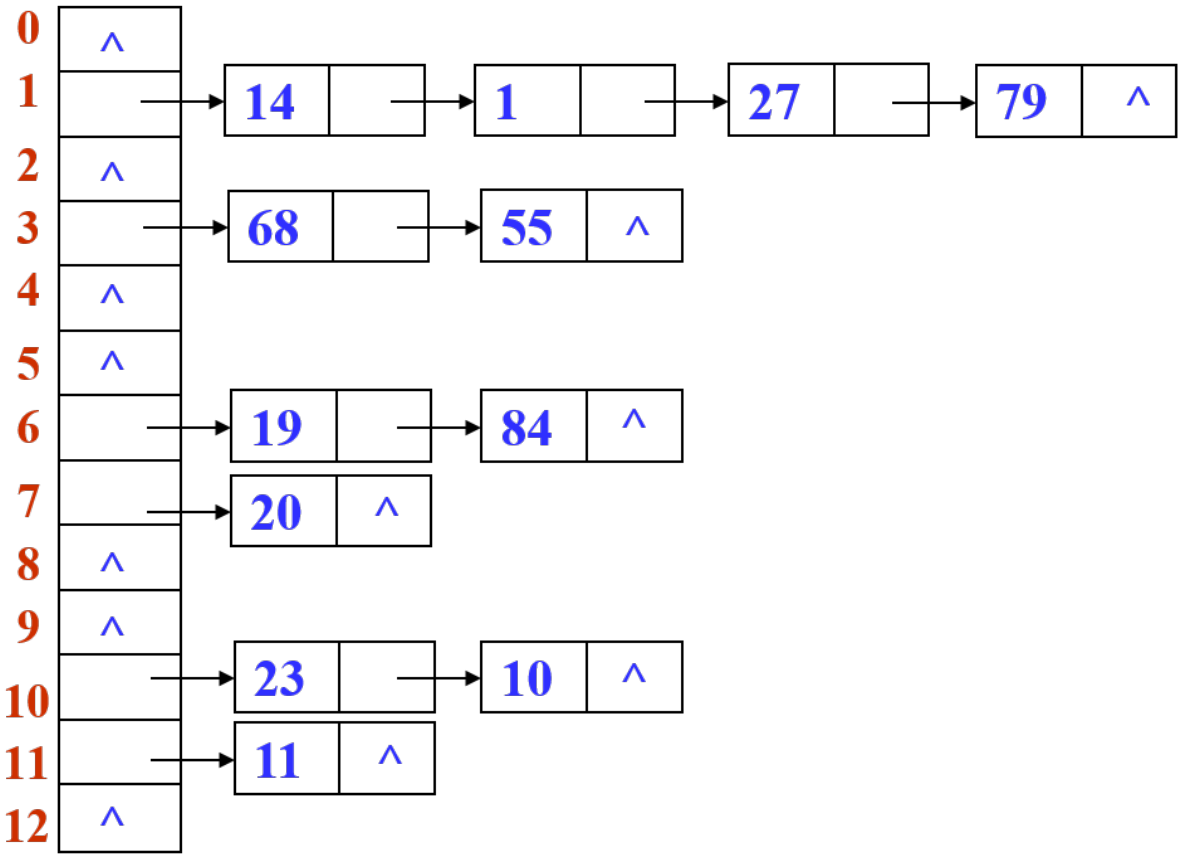
- 也称“拉链法”
- 例子：设给定关键字 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }，哈希函数为 $H(key) = key$ MOD 7



链地址法处理冲突的例子

哈希函数为 $H(key) = key$ MOD 13

- 给定关键字关键字 (19,14,23,1,68,20,84,27,55,11,10,79)，表长为 13，用链地址法处理冲突



再哈希法

用一系列哈希函数为冲突关键字计算地址

- 当发生冲突时，计算下一个哈希地址，即：
  $H_i = RH_i(key), i = 1, 2, \ldots, k$, 其中 RH_i 是不同的哈希函数
- 额外代价：计算哈希地址时间增加

建立公共溢出区

方法

- 在基本散列表之外，另设一个溢出表保存与基本表中记录冲突的所有记录
- 设散列表长为 m，设立基本散列表 $hashtable[m]$，每个分量保存一个记录；溢出表 $overtable[m]$，一旦某个记录的散列地址有冲突，都填入溢出表中

例子：哈希函数为 $H(key) = key$ MOD 7

- 给定一组关键字 (15, 4, 18, 7, 37, 47) ，哈希表长为 7，采用建立公共溢出区来处理冲突
- 得到基本散列表和溢出表为：

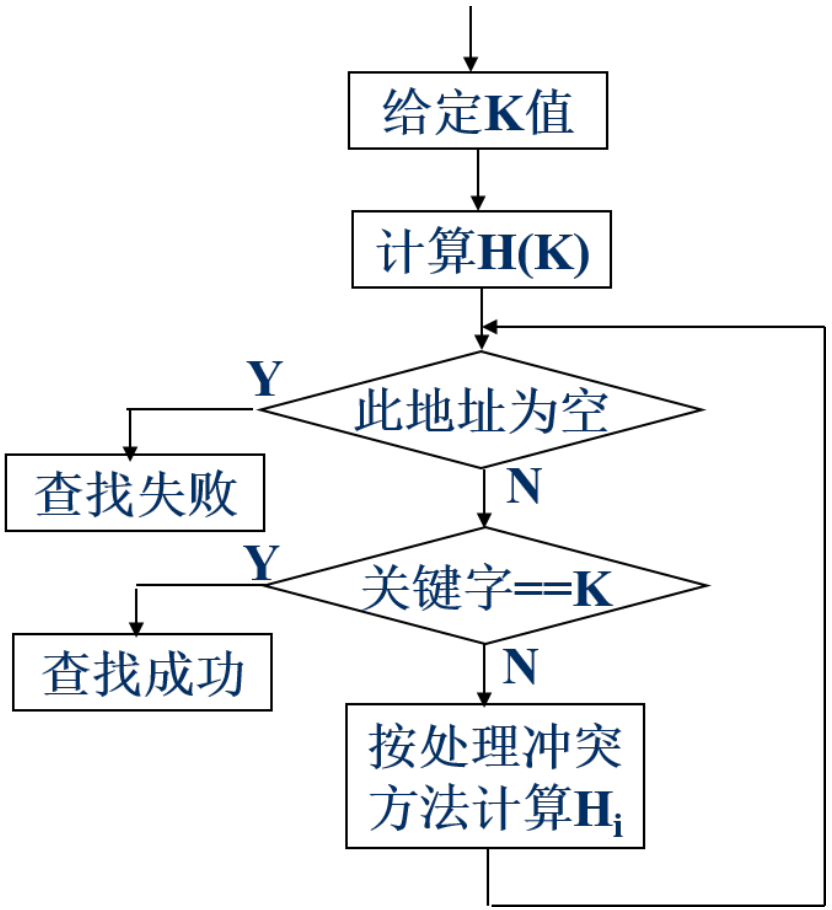
Hashtable表：

散列地址	0	1	2	3	4	5	6
关键字	7	15	37		4	47	

overtable表：

溢出地址	0	1	2	3	4	5	6
关键字	18						

哈希表的查找



哈希查找过程

- 对于给定值 K, 计算哈希地址 $i = H(K)$
  - 若 $r[i] = NULL$ 则查找不成功
  - 若 $r[i].key = K$ 则查找成功
- 否则“求下一地址 H_i”，直至
  - $r[H_i] = NULL$ (查找不成功)
  - 或 $r[H_i].key = K$ (查找成功)

哈希查找算法的实现

```
1  #define M 15
2  typedef struct node{
3    KeyType key;
4    struct node *link;
5  }HNode;
6
7  HNode *hash_search(HNode *t[], KeyType k){
8    HNode *p; int i;
9    i=h(k);
10   if (t[i]==NULL)
11     return(NULL);
12   p=t[i];
13   while(p!=NULL)
14     if (EQ(p->key,k))
15       return(p);
16     else p=p->link;
17   return(NULL);
18 }//查找散列表HT中的关键字K,用链地址法解决冲突
```

哈希查找的例子

哈希函数为 $H(key) = key$ MOD 13

- 给定一组关键字 (19,14,23,1,68,20,84,27,55,11,10,79)，哈希表长度为 16
- 用线性探测再散列处理冲突，得到哈希表为：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	1	68	27	55	19	20	84	79	23	11	10			

关键字 84 的查找过程

- $H(84) = 6$ 不空且不等于 84, 冲突
- $H1 = (6 + 1)$ MOD $13 = 7$ 不空且不等于 84, 冲突
- $H2 = (6 + 2)$ MOD $13 = 8$ 不空且等于 84, 查找成功, 返回记录的地址 8

关键字 38 的查找过程

- $H(38) = 12$ 不空且不等于 38, 冲突
- $H1 = (12 + 1)$ MOD $13 = 0$ 空记录，表中不存在关键字等于 38 的记录, 查找不成功

哈希表查找的分析

决定哈希查找 ASL 的因素

- 选用的哈希函数;
- 选用的处理冲突的方法;
- 哈希表饱和的程度：装载因子 $\alpha = n/m$ 值的大小，其中 m, n 分别表示表长和记录个数，α 越小，冲突的可能性越小

简化分析方法

- 一般情况下，可以认为选用的哈希函数是“均匀”的，则在讨论 ASL 时，可以不考虑它的因素
- 因此，哈希表的 ASL 是处理冲突方法和装载因子的函数
- 上一个例子中，采用线性探测再散列的冲突处理方法，其 ASL=2.5 (每个关键字查找一次的比较次数的均值)

哈希表查找的分析

当哈希表的冲突处理方法相同时，ASL 是装填因子 α 的函数，可证：

- 采用线性探测再散列的哈希查找：$ASL \approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$
- 采用平方探测再散列、随机探测再散列和再哈希的哈希查找：$ASL \approx -\frac{1}{\alpha}\ln(1-\alpha)$
- 采用链地址法的哈希查找：$ASL \approx 1 + \frac{\alpha}{2}$

进一步理解

- 哈希表的平均查找长度是 α 的函数，而不是 n 的函数
- 这说明，用哈希表构造查找表时，可选择一个适当的装填因子 α ，使得平均查找长度限定在某个范围内
- 这是哈希表所特有的特点

哈希表的练习题

题目说明

- 给定关键字序列是 (19, 14, 23, 01, 68, 84, 27, 55, 11, 34, 79)，哈希表长度是 11，哈希函数是 $H(key) = key$ MOD 11
  1. 采用开放地址法的线性探测方法解决冲突，请构造该关键字序列的哈希表
  2. 采用开放地址法的二次探测方法解决冲突，请构造该关键字序列的哈希表