第七章 高级字典结构

1

首先讨论字典与索引的关系;

然后讨论字典的其它实现:

- 以字符为结点的字符树表示
- 以关键码为结点的二叉排序树（包括静态的最佳二叉排序树和保持动态 平衡的二叉排序树）
- 多级索引结构（包括静态的多分树和动态的B树、B+树）

本章是第6章关于字典实现的继续

包括关于索引技术和方法的系统讨论

还可看作第5章介绍的树型结构的具体应用

2

7.1 字典与索引

7.1.1字典的索引

不等长结点的问题

上一章关于字典的讨论中，将元素关键码类型 KeyType 和值类型 DataType 都为 int 类型

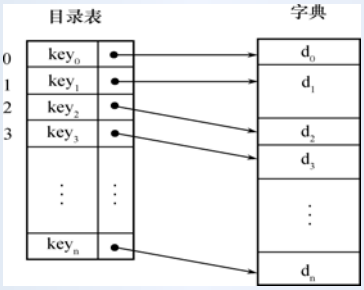
实际应用时，关键码可为其它类型，值也可出现为多种形式。不同的值需要的空间大小也可能不同，这样的字典难以简单地采用上一章介绍的顺序存储或散列存储实现

索引是从关键码到地址的映射。引入索引可把对不等长元素的字典的处理，转换成对仅包含关键码到地址对应关系（简单类型，而且元素等长）的索引结构的处理

3

举例

如果顺序表示字典中，元素值需要的空间长度不等，可另建立一个字典索引——常称为目录表。增加了目录表后，字典可以采用顺序存储，也可采用其它方式存储



4

索引的作用

检索元素时，只需在目录表中找对应关键码，就可以得到对应结点的存储位置

如果想做二分法检索，只需要对目录表中的元素（索引项）排序，而不需要移动字典本身的任何数据

通常索引项比字典数据小得多，操作起来更加方便快捷

索引与散列

索引与散列一样，都是从关键码到存储地址的映射方法

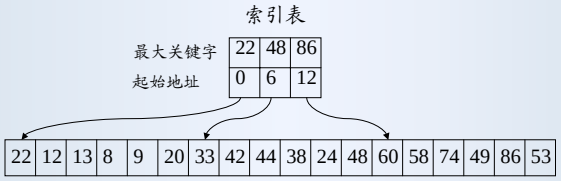
但散列法的映射是通过散列函数定义；而索引法是通过建立辅助的索引表解决（直接保存）

完全可能结合两者，例如先通过散列得到索引项

5

密集索引与稀疏索引

- 每个索引项对应字典中一个元素，称为密集索引
- 每个索引项对应字典中一组元素，称为稀疏索引



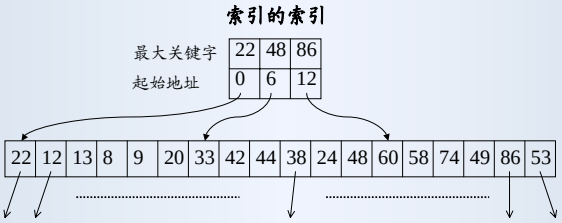
这里的数据分段有序（一段中最小的元素比前一段最大的元素大），段内无序。排序索引表可提高检索速度

6

索引的索引

大型字典的索引也很大

所谓索引的索引就是给庞大的（通常是密集的）索引，建立另外的辅助（通常是稀疏）索引结构，以达到加快查找字典中特定结点的目的



索引与字典

- 如果一个结构能支持基于关键码的字典信息检索，那么在这个结构里必定要保存关键码的信息
- 如果需要获得的与关键码关联的信息也存在这一结构里，那么这个结构就是一个字典
- 如果与关键码关联的信息存在这一结构之外，结构里只保存对相关信息的索引，那么该结构就是一个字典的索引
- 易见：同样组织方式既可以用于实现字典，也可以用于实现字典的索引。下面讨论中将不区分这两种情况
- 下面要讨论的各种树形结构都是如此，它们既可以用于实现字典，也可以用于实现字典的索引

7.2 字符树

字符树与树目录

关键码为字符串时，目录可以用称为字符树的结构表示

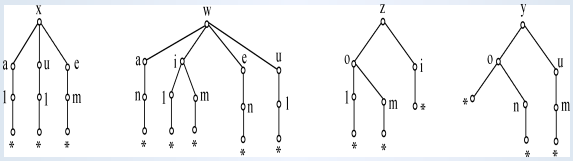
字符树:

- 每个树结点表示关键码中的一个字符，从树根出发的每条路径上的字符连起来得到的字符串是一个关键码
- 一个字典的所有关键码，可用一个字符树（林）中从根到其它结点路径对应字符串的集合表示
- 在每个构成关键码的结点增加一个指向对应元素的指针，这个字符树（林）就成为相应字典的一个树目录

举例

某字典的关键码由1至3个字符组成：第一个字符为w，x，y，z，第二个字符可为a，e，i，o，u，第三个字符可为l，m，n

关键码集合：K = {xal，wan，wil，zol，yo，xul，yum，wen，wim，zi，yon，xem，wul，zom}.

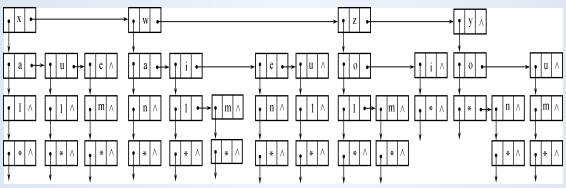


字符树（林）表示的树目录。标*的结点为其父结点的关键码对应的字典元素。这种结点的个数是字典的元素个数

双链树表示

把字符树（林）转换为对应的二叉树，并用llink-link法进行存储，通常称作双链树。

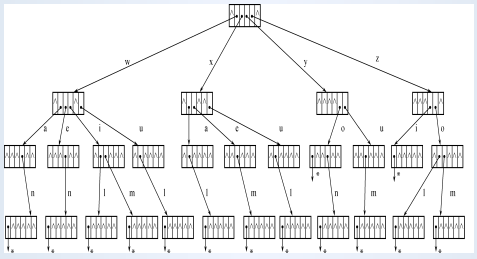
前例的字符树林转换成的双链树如下图，其中用*标记的结点的左指针给出对应元素的位置



多链表示

将字符树（林）的字符信息隐藏在边里，用代表字符的指针指向不同子字符树，整个字符树（林）变成一棵以指针数组为结点的树。多链表示常被称为trie结构。

下图是前面字符树林转换成的trie结构



7.3 二叉树排序树

采用（链接实现的）二叉树作为字典的存储结构，可能

- 得到较高的检索效率
- 链式存储方式，元素插入、删除操作比较灵活方便

基本想法:

- 在二叉树结点里存储信息，设法组织好存储方式
- 利用树的平均高度远小于树中结点个数的性质，希望沿着路径检索，可能大大提高检索效率
- 二叉排序树 (Binary Sort Tree) 是保存有序数据的二叉树，也是一种基于二叉树的字典实现方法

13

7.3.1 二叉排序树的概念

定义：二叉排序树或为空；或是具有下列性质的二叉树:

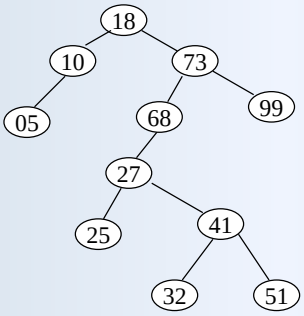
1. 如果其左子树不空，则左子树里所有结点的（关键码）值均小于它的根结点的（关键码）值;
2. 若其右子树不空，则右子树上所有结点的（关键码）值均大于它的根结点的（关键码）值;
3. 其左右子树（如果存在）也是二叉排序树。

二分法检索的判定树就是一棵二叉排序树

对二叉排序树做中序周游，得到的序列是按关键码排序的“上升”序列

14

K= {18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}



二叉排序树示例

将集中研究二叉排序树本身的结构

讨论中将忽略关键码以外的属性或指针

讨论中不区分是字典的二叉排序树索引还是字典的二叉排序树表示。下面对其他结构的讨论也如此

15

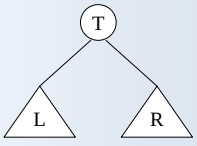
二叉排序树的存储结构（llink_rlink):

```
typedef  struct  BinTNode  * pBinTNode;

struct BinTNode {
    KeyType   key;    /* 结点的关键码字段 */
    DataType  value; /* 结点的属性字段 */
    PBinTNode  llink,  rlink;   /* 左、右指针 */
} BinTNode;

typedef  struct   BinTNode  *BinTree;
typedef   BinTree  *PBinTree;
```

普通的二叉树链接实现结构。但结点里包括关键码和数据

16

二叉排序树的检索算法



基本算法框架（检索 key 值）:

```
while （T 非空）{
    if (T.key==key) 成功结束;
    else  if (key < T.key ) 将 T 改为其左子树;
    else  将 T 改为其右子树;
}
失败结束; /* 子树为空 */
```

17

```
int searchNode (PBinTree ptree, KeyType key,
                PBinTNode *position) {
  PBinTreeNode p = *ptree, q = NULL;
  while (p != NULL) {
    q = p;
    if (p->key == key) {/* 成功，position指向找到的结点 */
      *position = p;
      return(TRUE);
    }
    else if (key < p->key) p = p->llink;/* 进入左子树继续 */
    else p = p->rlink;        /* 进入右子树继续 */
  }
  *position = q;
  return FALSE;/* 失败，position指向应插入处的父结点 */
}
```

18

二叉排序树中插入结点的算法

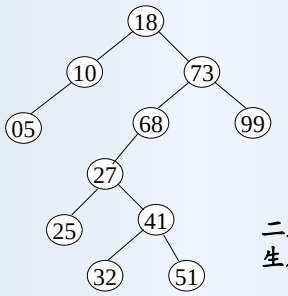
```
在二叉排序树里检索结点
    （找到相应结点或者插入位置的父结点）;
if ( 没有相应结点) {
    建立新结点;
    if (原二叉排序树是空树) 新结点作为根结点;
    else if (新结点 < 父结点) 插入左子树;
    else  插入右子树;
}
```



```
void insertNode (PBinTree ptree, KeyType key, DataType  v) {
   PBinTNode p, position;
   if ( searchNode(ptree,key,&position) == TRUE )
      return; /* 存在关键码为key的结点。根据需要处理 */
   p = (PBinTNode)malloc( sizeof(BinTNode) );/* 申请空间 */
   if (p == NULL) {   /* 空间耗尽 */
      printf("Out of Space!\n");
      return;
   }
   p->key = key; p->value = v; /* 结点数据填充 */
   p->llink = p->rlink = NULL;
   if (position == NULL)          /* 原树为空树 */
      *ptree = p;
   else if (key < position->key)
      position->llink = p;        /* 插入position的左子树 */
   else position->rlink = p;      /* 插入position的右子树 */
}
```


从空树出发经过一系列插入，可生成一棵二叉排序树。

例如，K={18, 73, 10, 05, 68, 99, 27, 41, 51, 32, 25}



二叉排序树的生成过程示例

二叉排序树结点的删除

从二叉排序树中删除结点的操作比较复杂

分两步：先找到被删除结点（假设用 p 指向），而后删除

关键：被删除结点可能在树中任何地方。要删除指定结点，还要维护二叉排序树的完整性（树结构完整，树中结点有序。整个树仍然是二叉排序树）

注意：保持有序 iff 二叉树中序周游序列不变！（少了 *p）

分3种情况讨论（设 *p 是被删除结点）。

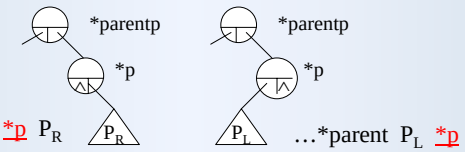
情况一：*p是叶结点

由于删除叶结点不破坏整棵树的结构和结点之间的序关系，因此只需要修改其父结点的指针

情况二：*p只有左子树 P_L 或只有右子树 P_R

只要用*p 的唯一子结点P_L或P_R的根结点代替 *p 即可.

假设指针 parentp 指向 *p 的父结点。



删除后：…*parent P_R

删除后：…*parent P_L

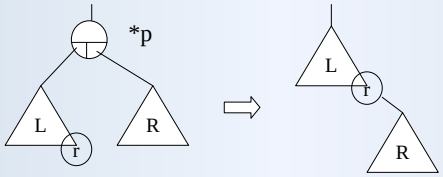
显然删除 *p 后中序周游序列没有变（仅仅是少了 *p）。

3. 若*p的左右子树均不空，删除 *p 前中序周游序列为

{…L r *p R …}

删除*p 后要保持其它元素的中序顺序不变，有两种做法：

第一种方法：用*p的左子树的根结点代替*p，*p的右子树作为 r (*p 的左子树中的最大结点) 的右子树;



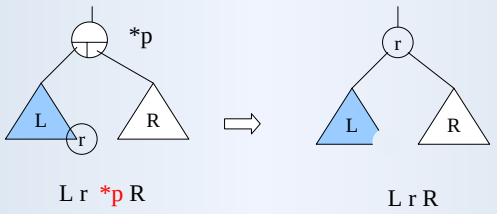
结果形态“可能”不好，“可能”使树变高，检索效率“可能”恶化

但也未必，依赖于实际的树状态

第二种方法:

(a) 按对称序周游p的左子树，找到关键码最大的结点r，把r从树中取下（叶结点简单，非叶用其左子女代替）

(b) 用结点 r 代替被删除结点p（可以把结点 *r 里的信息复制到 *p 去）



L r *p R　　L r R

算法　从二叉排序树上删除结点

```
检索被删除结点*p;
if (*p无左子女)
    用*p的右子女代替*p;
else { /* 第一种方式 */
    找*p的左子树中最右下结点*r;
    用*r的右指针指向*p的右子女;
    用*p的左子女代替*p;
}
```



```
void deleteNode(PBinTree ptree, KeyType key) {
  PBinTNode pp = NULL, p = *ptree, r;
  while (p != NULL && p->key != key) { /* 检索结点 */
    pp = p; /* 保证 *pp 总是 *p 的父结点 */
    if (p->key > key) p = p->llink;
    else p = p->rlink;
  }
  if (p == NULL) return;    /* 树中无关键码为key的结点 */
  if (p->llink == NULL) {    /* 结点*p无左子树 */
    if (pp == NULL) *ptree = p->rlink;/* 删除的是原树根*/
    else if (pp->llink == p)   /* *p是其*pp的左子女 */
      pp->llink = p->rlink;  /* 将*p右子树链到*pp左链 */
    else  pp->rlink = p->rlink; /* 将*p右子树链到*pp右链 */
  } /* 接下页 */
```



```
 /* 实际上，还可以考虑没有右子树的情况 */
 else {  /* 结点*p有左子树 */
    /* 用第一种方式删除 */
    for (r = p->llink; r->rlink != NULL; r = r->rlink)
        ;  /* 找到 p 的左子树的最右结点 */
    r->rlink = p->rlink;       /* 令*r右指针指向 *p 右子女 */
    if (pp == NULL) *ptree = p->llink;
    else if (pp->llink == p)       /* 用*p的左子女代替*p */
      pp->llink = p->llink;
    else pp->rlink = p->llink;
  }
  free p;           /* 释放被删除结点的存储 */
}
```



```
  /* 用第二种方式删除 */
 else {  /* 结点*p有左子树 */
    PBinTNode rr = p;
    for (r = p->llink; r->rlink != NULL; r = r->rlink)
        rr = r;   /* 找到 p 的左子树的最右结点 */
    p->key = r->key;       /* 复制结点信息 */
    p->value = r->value;
    if (rr == p) p -> llink = r->llink; /* *r 的父结点就是 *p */
    else rr->rlink = r->llink; /* 用*r 的左子女代替 *r */
    p = r;        /* 为统一用下面删除 */
  }
  free p;           /* 释放被删除结点的存储 */
}
```


性能分析

对不同形态的二叉排序树，平均检索长度也不同

最坏情况下，二叉树蜕变为某种单支树，此时的平均检索长度为(n+1)/2，复杂性是 O(n)。

在随机的情况下，平均性能为:

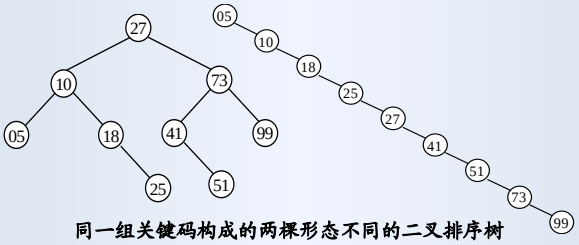
$$P(n) = O(\log n)$$

这里的随机假设：n 个结点的关键码不同；n个结点集的任何随机选择序列可以生成一棵二叉排序树。计算在每棵二叉排序树上的所有检索的平均长度。求出平均值

算法分析可参考：“Introduction to Algorithms”，高教出版社影印本，265页。其他许多关于算法的书籍上也都有

7.4 最佳二叉排序树

n 个不同关键码按不同顺序插入一棵空二叉排序树，可以形成 n! 棵二叉排序树。



同一组关键码构成的两棵形态不同的二叉排序树

最佳二叉排序树：检索效率最高的二叉排序树（直观考虑）

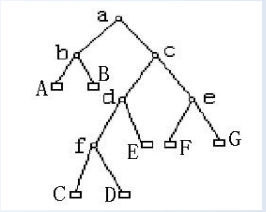
应该用平均检索长度评价二叉排序树的优劣。需要考虑各种关键码出现的概率，不仅要考虑对二叉树里有的关键码的检索，还要考虑从对其他关键码的检索（失败的检索）

扩充二叉排序树的对称序列（中序周游序列）：按中序周游扩充二叉树得到的序列。其中内外部结点交叉排列，第i个内部结点位于第i个外部结点和第i+1个外部结点之间。

用扩充二叉排序树表示字典的可能关键码集合，内部结点代表一个元素的关键码，外部结点代表与其相邻的两个内部结点关键码之间的那些关键码。

右图的中序周游（对称序列）：

AbBaCfDdEcFeG



在扩充二叉排序树里，关键码的平均检索长度

$$E(n)=\frac{1}{w}\left[\sum_{i=1}^{n}p_i(l_i+1)+\sum_{i=0}^{n}q_i l'_i\right]$$

其中 l_i 是内部结点i的层数，l'_i 是外部结点 i 的层数，p_i 是检索内部结点 i 的关键码的频率，q_i 是被检索的关键码属于外部结点i的关键码集合的频率，p_i，q_i 也叫结点的权。

令

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

p_i/w 是检索内部结点 i 的关键码的概率，q_i/w 是被检索的关键码属于外部结点 i 的关键码集合的概率。

最佳二叉排序树：检索中平均比较次数最小，即E(n)最小的二叉排序树。

如何构造最佳二叉排序树？

简单情况：所有结点的检索概率相等

$$\frac{p_1}{w}=\frac{p_2}{w}\cdots\frac{p_n}{w}=\frac{q_0}{w}=\cdots=\frac{q_n}{w}=\frac{1}{2n+1}$$

$$E(n)=\frac{1}{2n+1}\left[\sum_{i=1}^{n}(l_i+1)+\sum_{i=0}^{n}l_i'\right]$$

$$=(\ IPL+n+EPL)/(2n+1)$$

$$=(2IPL+3n)/(2n+1)\quad /*\ EPL=IPL+2n\ */$$

其中 $IPL=\sum_{k=1}^{n}\lfloor\log_2 k\rfloor=(n+1)\lfloor\log_2 n\rfloor-2^{\lfloor\log_2 n\rfloor+1}+2$

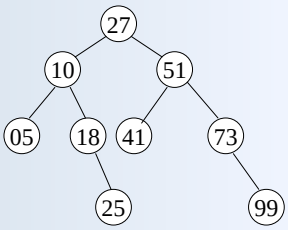
结论：平均检索长度 E(n) 为 $O(\log_2 n)$

构造方法一：按二分检索遇到的顺序逐个插入

1）将结点按关键码排序

2）按二分法检索依次关键码，检索中遇到的尚未放入二叉排序树中的结点，就把它插入树中。

例：K={05, 10, 18, 25, 27, 41, 51,73, 99}
0 1 2 3 4 5 6 7 8



检索5，依次遇到27, 10, 5，把它们逐个插入；

检索10, 都在；检索18, ...

构造算法的时间代价（不计排序）为 O(n log n)

检索：O(log n)

构造方法二：递归构造方式

设数组 a[low .. high] 里是按照关键码排序的一组字典项。

1，m = (high + low)/2;

2，把 a[m] 存入被构造的二叉排序树的根结点 t，递归地：

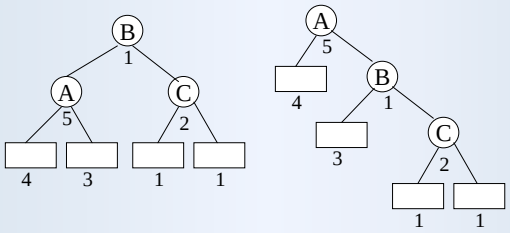
- 从 a[low .. m-1] 出发构造二叉排序树作为 t 的左子树
- 从 a[m+1 .. high] 出发构造二叉排序树作为 t 的右子树

数组片段为空时直接返回 NULL，表示空树

构造最佳二叉排序树的算法的这一部分的复杂性为 O(n)

从第7章可知，n个元素的排序算法的复杂性为 O(n log n)。因此整个构造过程的复杂性也为 O(n log n)。

二、结点的检索概率（频率）不等的情况



E(n)=33/17　　E（n)=29/17

结点处标的是频率，除以17得到“概率”

问题：给定排序的关键码集合 {key_1, key_2,…,key_n} 和两个权集合{p_1,…, p_n}和{ q_0, q_1 …,q_n}

其中p_j是检索内部结点i的概率，q_j是被检索关键码属于外部结点i 的关键码集合的概率

要求构造一棵最佳二叉排序树，即构造出一棵二叉排序树，使下面的代价函数达到最小

E(0, n) 表示包含 n 个内部结点和 n+1个外部结点的树的代价:

$$E(0,n) = \sum_{i=1}^{n} p_i(l_i+1) + \sum_{i=0}^{n} q_i l_i'$$

易证明：最佳二叉排序树里的任何子树都最佳

设对应 key_1, key_2,…,key_n 的内部结点为 v_1, …,v_n，外部结点为 e_0, e_1, …,e_n

用T(i, j) 代表包含内部结点 v_{i+1}, …, v_j 和外部结点 e_i, e_{i+1}…, e_j 的最佳二叉排序树

例：T(0, 1) 表示包含内部结点 v_1，以及外部结点 e_0, e_1 的最佳二叉排序树

T(2, 5) 表示包含内部结点v_3, v_4 , v_5 ，以及外部结点 e_2, e_3, e_4, e_5 的最佳二叉排序树

下面讨论中用一个例子说明构造过程:

设给定关键码集合 {A, B, C}，权集合 {5, 1, 2} 和 {4, 3, 1, 1}

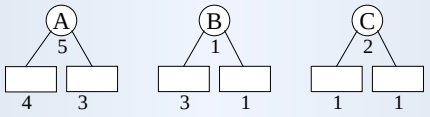
三个内部结点，4个外部结点，要求做出最佳二叉排序树

用 T(0, 1) 表示只含e0, v1, e1 的二叉排序树，它最佳（这种二叉排序树只有一棵），代价是$q_0+p_1+q_1$

T(1, 2), …, T(n-1, n) 自然也都是最佳二叉排序树（都唯一），相应的代价 E(i, i+1) 很容易算

1）对实例构造 3 棵包含一个内部结点的最佳二叉排序树:

T(0, 1)　　T(1, 2)　　T(2, 3)



E(0, 1) = 12/17　　E(1, 2) = 5/17　　E(2, 3) = 4/17

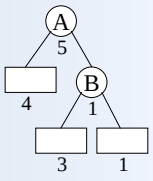
2）构造 T(0, 2) 和 T(1, 3)，它们各包含两个内部结点

T(0, 2) 有两种可能:

- 以 v_1 为 根，e_1为左子树，T(1, 2) 为右子树
- 以 v_2 为 根，T(0, 1) 为左子树，e_2为右子树

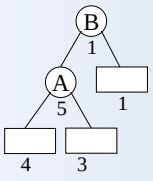
比较哪种构造方式得到的树最佳，选出它作为 T(0, 2)

候选$_1$



$E(0, 2)_1$ = 19/17

候选$_2$

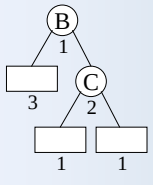


$E(0, 2)_2$ = 26/17

可以类似的构造出 T(1, 3)

两种可能性:

- 以 v2 为根，e1 为左子树，T(2, 3) 为右子树
- 以 v3 为根，T(1, 2) 为左子树，e3 为右子树



前一个是最优

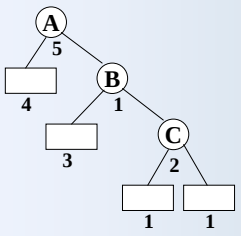
可算出 E(1, 3) = 12/17

每次都要重新计算 E 值吗?

不必！可以递推计算。递推公式见后面幻灯片和书

3）构造包含三个内部结点的最佳二叉排序树 T(0, 3)，即本问题的最终结果。候选二叉排序树：

- 以 v_1 为根，其左子树为 e_0，右子树为 T(1, 3)
- 以 v_2 为根，其左子树为 T(0, 1) 右子树为 T(2, 3)
- 以 v_3 为根，其左子树为 T(0, 2)，右子树为 e_3



计算这 3 棵二叉排序树的 E 值，从中选出最佳二叉排序树（右图）

对一般问题，都可以这样逐步构造出一组最佳二叉排序树，最终构造出所需要的最佳二叉排序树

一般步骤（第 m 步）：

对所有 0 < i < m，下面的最佳二叉排序树都已构造：

T(0, 0+i), T(1, 1+i), T(2, 2+i), …

借助于它们，可以构造出 n - m +1 棵最佳二叉排序树：

T(0, 0+m), T(1, 1+m), …

因为，可以从所有可能的排序树构造方式中选出最佳

下面算法的基本想法就是逐步构造最佳二叉排序树的子树，最终构造出包含所有结点的最佳二叉排序树。构造过程：

步骤1：构造包含一个内部结点 T(0,1), T(1,2),…,T(n-1,n)

步骤2：构造包含二个内部结点 T(0,2), T(1,3),…,T(n-2,n)

……

步骤n： 构造T(0,n) 。

构造和代价计算：

考虑key_{i+1}, key_{i+2}, …, key_j的内部结点，相应内/外部结点的权为 q_i, p_{i+1}, q_{i+1}, …, p_j, q_j。用 R(i, j) 表示 T(i, j) 的根。

权和：$W(i, j) = p_{i+1}+\ldots+ p_j + q_i + q_{i+1} +\ldots+ q_j \ (0 \leqslant i \leqslant j \leqslant n)$

在构造 T(i, j) 时，对所有 $i < k \leqslant j$，T(i, k-1) 和 T(k, j) 都已存在，而且已知相应代价 C(i, k) 和 C(k, j)。

对每个 k，以key_k 为根，T(i, k-1) 和 T(k, j) 为左右子树的二叉树的权为 $C_k(i, j) = W(i, j) + C(i, k-1)+C(k, j)$ (结点增加一层)

从所有可能 k 中选择使 $C_k(i, j)$ 达到最小的那个 k，与之对应树就是 T(i, j)，其根 R(i, j) = k。

新树的代价可以由构造它的已知最佳二叉排序树的代价算出：

$$C(i, j) = W(i,j) + \min\{ C(i,k-1)+C(k,j) \mid i < k \leqslant j \}$$

关键码 {A, B, C}

p = {5, 1, 2}　q = {4, 3, 1, 1}

w, r, c 为4×4数组

交替的权值序列 4 5 3 1 1 2 1

直接算出

w :				
	4	12	14	17
	0	3	5	8
	0	0	1	4
	0	0	0	1

第一步

r :				
	0	1	0	0
	0	0	2	0
	0	0	0	3
	0	0	0	0

c :				
	0	12	0	0
	0	0	5	0
	0	0	0	4
	0	0	0	0

第二步

r :				
	0	1	1	0
	0	0	2	2
	0	0	0	3
	0	0	0	0

c :				
	0	12	19	0
	0	0	5	12
	0	0	0	4
	0	0	0	0

第三步

r :				
	0	1	1	1
	0	0	2	2
	0	0	0	3
	0	0	0	0

c :				
	0	12	19	29
	0	0	5	12
	0	0	0	4
	0	0	0	0

参数数组 p[0..n-1] 和 q[0..n] 存内部结点和外部结点的权

c, w, r 为 (n+1)*(n+1) 个元素的二维数组，只用上三角部分

w[i][j] 记录 W(i, j) 值（一段权之和），可直接算出

c[i][j] 和 r[i][j] 分别记录最佳二叉排序树 T(i,j) 的代价和根

构造好的最佳二叉排序树的总代价存放在 c[0][n]

r[0][n] 是根结点的编号。根据 r[0][n] 的值，就可以确定两棵子树的根在那里

例：假定内部结点为 v_1 到 v_8，r[0][8] 的值是 5，可知

- 其左子树的根结点的编号保存在 r[0][4]
- 其右子树的根结点的编号保存在 r[5][8]

按这种方式可以通过追溯得到构造出的树的结构


```
for (i = 0; i <= n; i++)      /* 数组上三角清零 */
      for (j = 0; j <= i; j++)
        c[i][j] = w[i][j] = r[i][j] = 0;

for (i = 0; i <= n; i++) {    /* 计算所有的w[i][j] */
      w[i][i] = q[i];
      for (j = i+1; j <= n; j++)
        w[i][j] = w[i][j-1] + p[j] + q[j];
}

for (j = 1; j <= n; j++) {
      /* 计算只包含一个内部结点的n棵最佳二叉排序树 */
      c[j-1][j] = w[j-1][j]; r[j-1][j] = j;
}
```

```
for (m = 2; m <= n; m++) {
   /* 计算所有包含m个内部结点的最佳二叉排序树
      共计 n – m + 1 棵 */
   for (i = 0; i <= n-m; i++) {
      j = i+m; k0 = i+1;
      min = MAXVALUE;
      for (k = i+1; k <= j; k++)
         /* 在 (i, j] 内找使 C(i,k-1)+C(k,j) 最小的 k */
         if ( c[i][k-1] + c[k][j] < min) {
            min = c[i][k-1] + c[k][j]; k0 = k;
         }
      c[i][j] = w[i][j] + min;
      r[i][j] = k0;
   }
}
```


算法的复杂性为 $O(n^3)$，对较大的 n 非常耗时

任意权值的最佳二叉排序树有一定意义，但更多是理论价值，说明即使问题如此复杂，还是存在一般性的构造方法

实际中使用很少。一是创建这种树的代价比较高，二是实际中很难得到有价值的访问分布情况

这个算法很典型，采用的算法模式称为“动态规划”。这种方法在复杂的计算中很有用

动态规划算法的特点：在计算过程中维持一大批子问题的解（在这里维护的是最优解），基于对小的子问题的解，逐步构造更大子问题的解，最终构造出整个问题的解

动态规划法在算法设计中使用广泛

7.5 平衡二叉排序树（AVL树）

最佳二叉排序树很好，但这种结构：

1）只能根据元素全面情况静态构造

2）结点权值不同时构造过程很耗时

3）适合作为静态字典表示，不能很好支持动态变化

最佳是全局性质，难以在局部把握

如果静态构造好一棵最佳二叉排序树，在使用中经过一系列动态插入删除后不进行维护，树结构就可能不断恶化，导致性能下降。最坏检索性能可能趋向 O(n)

在变动中维护最佳性质很困难，因为一次插入删除可能造成大范围的影响，维护代价很高（一定可以做）

为有效地实现动态字典，人们研究并提出了许多种支持动态字典操作的树形结构

这些结构的共同性质都是放弃最佳，但设法提供接近最佳的性质，换取在动态操作中较易通过局部调整进行维护

这些树结构的基本想法，都是通过树的局部性质反应某种近似最佳的性质，支持通过局部的调整维护树的较好结构（树的高度与结点个数成对数关系）

对各种操作都有最坏性能保证（无论是检索还是插入删除），保证各种操作都能在一条路径上完成

不同结构所采用的思想不同，下面要介绍的结构都属此列

- AVL树
- B树等

平衡二叉排序树（ AVL树）

平衡二叉排序树用局部的“平衡”概念代替最佳概念。

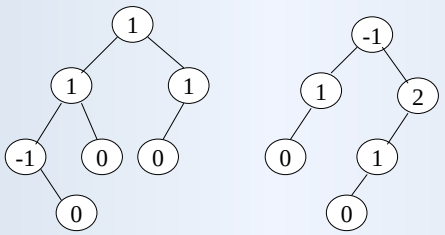
基本想法：如果二叉树中每个结点的左右子树高度差不多，整个二叉树的结构也会比较好，不会出现特别长的路径

定义：平衡二叉排序树或是空树，或其左右子树都是平衡二叉排序树，而且左右子树的深度之差的绝对值不超过1

这种树的性质可用各结点的平衡刻画。结点平衡可用平衡因子 BF（Balance Factor）描述。BF定义为该结点左右子树的深度之差，可能取值只有 -1, 0, 1。（局部性质，局部描述）

可证：n 结点平衡二叉排序树的深度小于 c log n（c为常量）

平衡二叉排序树又称AVL树，以其发明者（前苏联的G. M. Adel'son-Vel'skii 和 E. M. Landis）的名字命名。



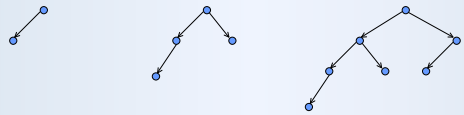
平衡和不平衡二叉排序树示例（结点数值为BF）

在AVL树上检索的时间代价（最坏情况）与最佳二叉排序树中最长路径的长度仅差一个常量因子

插入删除可能要调整树结构。只要调整能在一条路径上完成，这些操作的时间代价也会是 O(log n)

AVL树的形态分析

考虑高度为h的结点最少的（最接近不平衡的）AVL树:



结点个数: t(1) = 2, t(2) = 4, t(h+2) = t(h+1) + t(h) + 1

很像斐波纳契序列 F(0) = 0, F(1) = 1, F(h+2) = F(h+1) + F(h)

易证（数学归纳法） t(h) = F(h+3) – 1

由渐进公式 $F_h \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h$，可得:

n个结点AVL树的高度 h = $O(\log_2 n)$，实际不超过 3/2 $\log_2(n+1)$

可参考: 《数据结构》，许卓群等，高教出版社1984

AVL树检索与普通二叉排序树相同。问题是在动态变化中维持平衡? 插入删除不但要保持树结构和结点序，还要维持平衡

AVL树动态操作（插入/删除）的方式:

1. 按关键码顺序要求确定位置插入结点（或删除结点）;
2. 如果树失衡，则进行尽可能局部的调整，恢复树的平衡

插入和删除操作及其调整都可以在一条路径上完成。由于AVL树的性质，插入删除操作的时间开销为 O(log n)。

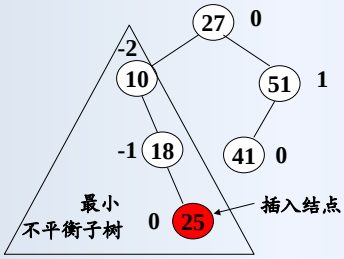
为实现AVL树，每个结点需增加一个平衡因子记录。

下面先讨论插入与其后可能的树调整。

7.5.1 调整平衡的模式

插入后可能导致树中某个局部失衡，必须调整。

最小不平衡子树: 离插入结点最近，且根结点的平衡因子绝对值大于1的子树。



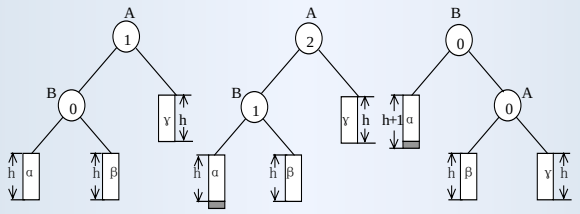
若调整后最小不平衡子树恢复平衡，且恢复到插入前的高度，那么整棵树也就恢复平衡了

若恢复可局部完成，插入操作的复杂性就不会超过 O(log n)

设最小不平衡子树的根为A

根据当时的具体情况，调整动作分为四种:

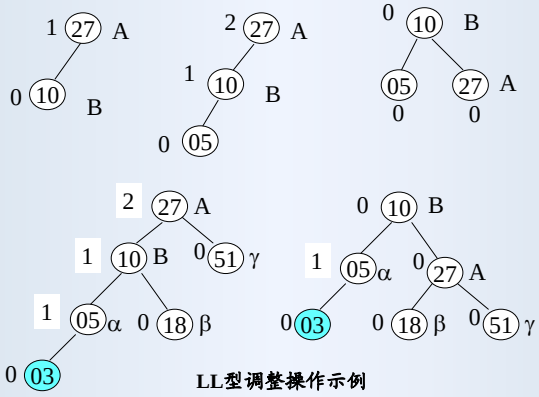
1. LL型调整;
2. RR型调整;
3. LR型调整;
4. RL型调整;



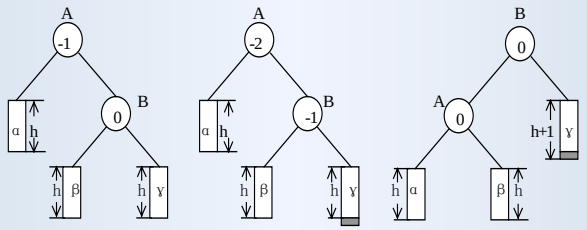
LL型调整操作示意图 (α B β)A(γ)= (α)B(β A γ)

情况: 原树左子树高，新元素插入左子树左侧（外侧）。

调整规则: A的左子女B提升为新二叉树的根; A（连同其右子树γ）向右下旋转作为B的右子树; B原来的右子树β作为A的左子树。



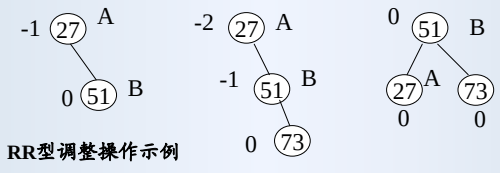
LL型调整操作示例



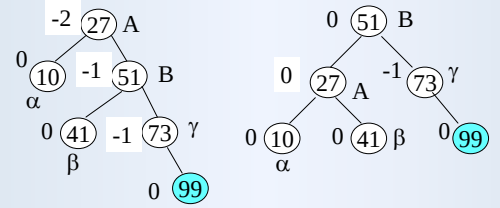
RR型调整操作示意图 （α)A(β B γ)= (α A β)B(γ)

情况：原树右子树高，新结点插入右子树右侧（外侧）

调整规则：A的右子女B提升为新二叉树的根；A（连同其左子树α）向左下旋转作为B的左子树；B原来的左子树作为A的右子树。

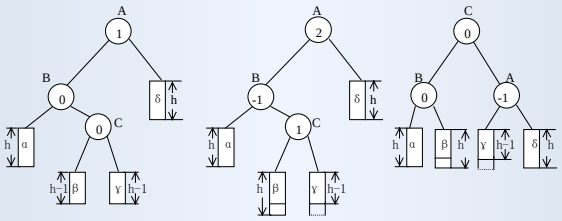


RR型调整操作示例



下面操作都返回调整结构后的新子树的根

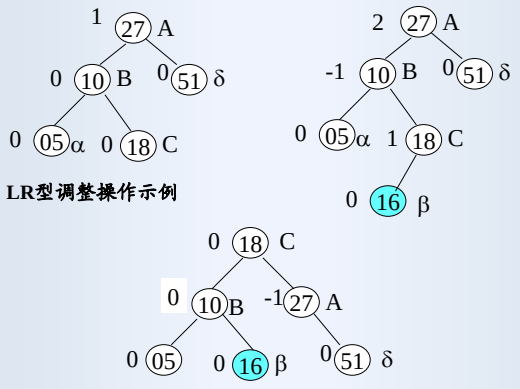
```
PAVLNode lL(PAVLNode a, PAVLNode b) {
   a->llink = b->rlink; b->rlink = a; a->bf = b->bf = 0;
   return b; /* b指向调整后子树的根 */
}

PAVLNode rR(PAVLNode a, PAVLNode b) {
   a->rlink = b->llink; b->llink = a; a->bf = b->bf = 0;
   return b;
}
```

LR型调整操作示意图 ((α)B(β C γ))A(δ)= (α B β)C(γ A δ)

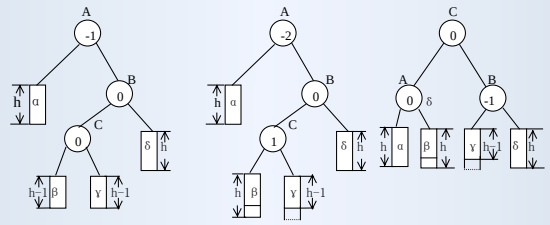
情况：原树左子树高，新结点插入左子树右侧（内侧）。

调整规则:设C为A的左子女B的右子女，将C提升为新二叉树的根；C的原父结点B（连同其左子树α）向左下旋转作为C的左子树，C的原左子树β成为B的右子树；A（连同其右子树δ）向右下旋转作为C的右子树，C的原右子树γ作为A的左子树。



LR型调整操作示例

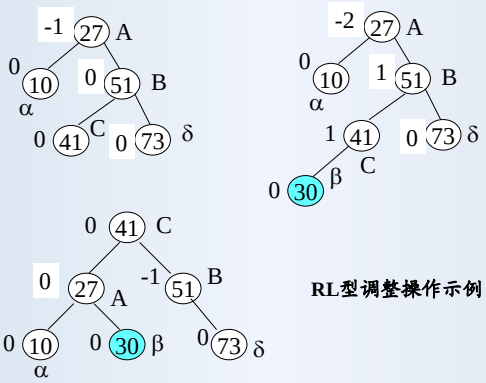

```
PAVLNode lR(PAVLNode a, PAVLNode b) {
   PAVLNode c = b->rlink;
   a->llink = c->rlink; b->rlink = c->llink;
   c->llink = b;  c->rlink = a;
   switch (c->bf) {
   case 0:   /* LR(0)型调整 */
      a->bf = b->bf = 0;  break;
   case 1:   /* 新结点在*c左子树，LR(L)型调整 */
      a->bf = -1;  b->bf = 0;  break;
   case -1:  /* 新结点在*c右子树，LR(R)型调整 */
      a->bf = 0;  b->bf = 1; break;
   }
   c->bf = 0;
   return c;
}
```



RL型调整操作示意图（α)A((β C γ)B(δ))=(α A β)C(γ B δ)

情况：原树右子树高，新结点插入右子树的左侧（内侧）。

调整规则：设C为A的右子女B的左子女，将C提升为新二叉树的根，C的原父结点B（连同其右子树δ）向右下旋转作为C的右子树，C的原右子树γ作为B的左子树；A连同其左子树α向左下旋转作为C的左子树，C原来的左子树β作为A的右子树。



RL型调整操作示例


```
PAVLNode  rL(PAVLNode a, PAVLNode b) {
   PAVLNode c = b->llink;
   a->rlink = c->llink;  b->llink = c->rlink;
   c->llink = a;  c->rlink = b;
   switch (c->bf) {
   case 0:  /* *c本身就是插入结点，RL(0)型调整 */
      a->bf = 0;  b->bf = 0;  break;
   case 1:  /* 插在*c的左子树中，RL(L)型调整 */
      a->bf = 0;  b->bf = -1;  break;
   case -1: /* 插在*c的右子树中，RL(R)型调整 */
      a->bf = 1;  b->bf = 0;  break;
   }
   c->bf = 0;
   return c;
}
```


上面的几种情况在经过平衡旋转处理后：

- 原来的最小不平衡子树变成了平衡二叉排序树，
- 其高度与插入之前这棵子树的高度相同。
- 因此：当平衡二叉排序树因插入结点而失衡时，仅需对最小不平衡子树进行旋转处理，就可以恢复整棵树的平衡。

调整完全是局部的，可以高效实现。

7.5.2 插入元素的算法

插入关键码key的结点，平衡二叉排序树用链接表示。梗概：

1）找插入位置的过程中记录遇到的最小不平衡子树的根：

令 a 指向离插入位置最近的平衡因子不为0的结点。不存在这种结点时 a 指向树根；

若插入后失衡，*a 就是失衡位置。

2）修改从 *a 的子结点到新结点的路径中各结点的平衡因子：

插入前这段路径里的结点的平衡因子均为0；

插入后从 *a 的子结点开始扫描路径上的结点*p，若新结点插入 *p 左子树，则 *p 平衡因子变为1；否则变为 -1

3）检查以 *a 为根的子树是否失衡。

若 *a 平衡因子为0，则插入后不会失衡，修改平衡因子；

若 *a 平衡因子为1且新结点插入 *a 的左子树则出现失衡：

- 若新结点插入*a的左子女的左子树则采用LL型调整；
- 若插入到*a的左子女的右子树则采用LR型调整；

若 *a 平衡因子为 –1且新结点插入 *a 的右子树则出现失衡：

- 若新结点插入 *a 的右子女的右子树则采用RR型调整；
- 若插入到*a的右子女的左子树则采用RL型调整。

```
PAVLNode createNode(KeyType key) { /* 可加value域赋值 */
   PAVLNode node  = (PAVLNode)malloc(sizeof(AVLNode));
   if (node != NULL) {
      node->key = key; node->bf = 0; node->llink = node->rlink = NULL;
   }
   return node;
}
```

AVL树插入算法

```
void avlSearchInsert(PAVLTree ptree, KeyType key){
   PAVLNode a, b, parent_a, p, q, node;      int d;
   if(*ptree == NULL) { /* 原树为空 */
      *ptree = createNode(key); return;
   }

   parent_a = q = NULL; a = p = *ptree;
   while ( p != NULL ) {   /* 确定插入位置及最小不平衡子树 */
      if (key == p->key) return;  /* 关键码key存在，酌情处理 */
      if(p->bf != 0) {         /* 记录最小不平衡子树 *a */
         parent_a = q;  a = p;
      }
      q = p;
      if(key < p->key) p = p->llink;
      else p = p->rlink;
   }
   /* *q是插入点的父结点，parent_a 和a确定最小不平衡子树 */
```



```
   node = createNode(key);
   if (key < q->key) q->llink = node;  /* 作为*q的左子树 */
   else q->rlink = node;                /* 作为*q的右子树 */

   /* 新结点已插入，a 指向最小不平衡子树 */
   if (key < a->key) { /* 新结点在 *a 的左子树 */
      p = b = a->llink; d = 1;
   }
   else {                /* 新结点在*a 的右子树 */
      p = b = a->rlink; d = -1;
   } /* d 记录了新结点在 *a 的哪棵子树 */

   /* 修改 *b 到新结点路上各结点的BF值，*b 为 *a 的子女 */
   while ( p != node ) { /* node 一定存在，不用判断 p 空 */
      if (key < p->key) {   /* *p的左子树增高 */
         p->bf = 1;  p = p->llink;
      }
      else {                /* *p的右子树增高 */
         p->bf = -1;  p = p->rlink;
      }
   }
```



```
   if (a->bf == 0) {  /* *a原BF为0，插入后不会失衡 */
      a->bf = d;  return;
   }
   if (a->bf == -d) { /* 新结点插在较低的子树中 */
      a->bf = 0;  return;
   }
   /* 新结点插在较高子树中，失衡，需对子树调整 */
   if (d == 1)                /* 新结点插在*a的左子树*/
      if(b->bf == 1) b = lL(a, b);   /* LL型调整*/
      else b = lR(a, b);  /* LR型调整*/
   else                       /* 新结点插在*a的右子树*/
      if(b->bf == -1) b = rR(a, b);   /* RR型调整 */
      else b = rL(a, b);  /* RL 型调整 */

   if (parent_a == NULL) *ptree = b; /* 原来的 *a 为树根 */
   else {
      if (parent_a->llink == a) parent_a->llink = b;
      else parent_a->rlink = b;
   }
}
```


AVL树的删除算法

基本思想与插入一样，先删除而后调整。其中：

- 把删除任意结点的问题变成删除某各子树的最右结点
- 进行删除（就是二叉排序树的删除）
- 调整平衡

可以参考许多书籍，例如：

《数据结构与程序设计：C语言描述》，清华大学影印。

插入关键码为key的结点的最大时间耗费为树的深度O(log n)。算法在检索插入结点的同时找到最小不平衡子树。对最小不平衡子树中平衡因子的最大调整时间与深度成正比。四种子树调整时间为常数O(1)。整个算法的时间复杂度为 O(log n)

AVL树的结点删除操作的复杂性也是O(log n)。因此 AVL 树能较好地支持动态字典

最佳二叉排序树的检索效率高，但若需要进行结点的插入或删除操作，操作后维持其最佳性的代价太大。

平衡二叉排序树的检索效率与最佳二叉排序树在同一数量级，优点是维护可以在局部完成。因此，内存的动态字典通常采用平衡二叉排序树（或其他类似结构）作为存储结构。

人们还提出了一些与性质与 AVL 树类似的树型结构（以及其他结构），可用于支持动态字典。

7.6 索引文件和B树/B+树

内存存储容量有限，且属于易失性存储器，关机使所存数据全部丢失。需要长期保存的字典不能留驻内存。由于容量有限，大型字典必须存放在外存（磁盘，光盘，磁带等）。

外存特点：

- 非易失性（持久性），容量大。
- 速度慢（磁盘的一次访问耗时为毫秒级，是内存访问纳秒级时间的10^6倍或者更多；磁带访问速度是秒级）。
- 成块传送，采用缓冲技术，缓和外存与程序使用间的差异。

外存访问通常包括两步：存储块定位，数据传输。

支持外存字典实现的数据结构必须考虑外存特点。

7.6.1 文件

外存数据的逻辑单位是文件，一个文件有一个文件明，其中保存着一系列数据记录。每个记录里保存的一组信息。

外存（磁盘、磁带等）的存储空间被划分为一大批固定大小的存储块（页块），每个存储块有一个唯一的“外存地址”。

顺序文件

基本实现方式是把一个文件的记录安排在一系列页块里。在一个页块里保存几个数据记录。一个具体记录的存储位置：所在页块（页块地址），块内的存储位置。

从物理上看，这样的一个文件就是一系列页块，这种结构适合顺序访问。访问一个记录，需要首先找到相关的页块，而后在那个页块里访问所需记录。

为加快文件内容的访问速度，需要更复杂的文件组织方式

散列文件

根据关键码把数据记录散列到确定的外存位置，或者在一个子文件里保存一批散列值相同的记录

这样的组织方式适合随机的数据访问（随机地根据关键码访问记录），但可能不适合顺序访问

索引文件

- 为加快字典访问速度，常需要为外存字典建立索引
- 文件数据存在顺序的一系列页块里，可支持顺序访问
- 另外为文件建立一个索引结构，支持基于关键码的高效的随机访问

文件很大时，可以采用多级索引结构，以提高检索效率:

- 为主文件建立了索引后，如果索引仍然很大，需要用许多页块实现，我们可以再针对一级索引的每个页块建立一个索引项，用这些索引项构成索引的索引
- 如果新一级索引仍占了多个页块，则可以再为它建索引。这样下去，就得到了一种多级索引结构——多分树
- 如果每个内部结点（根除外）有m个子女，则称为m分树
- 每个多分树结点占一个页块，结点里是二元组（k, p）的序列，在结点内按关键码 k 的值排序。
- 第 i 个二元组的 p 是本结点第 i 个子结点的（页块）地址，k 是该子结点上的最小（或最大）关键码。
- 多分树的叶结点就是主文件的最底层索引。主文件的每个页块可以看成是多分树的外部结点。

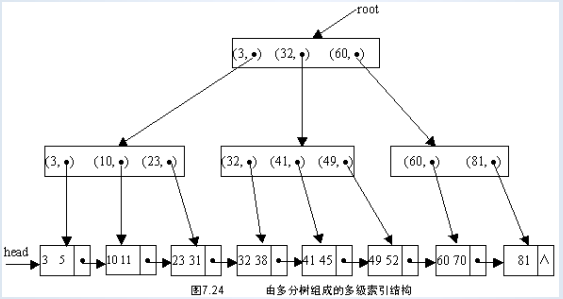


图7.24 由多分树组成的多级索引结构

检索关键码为 K 的记录，先读入根结点页块，从中找到满足 k⩽K的最后索引项 (k, p) 后读入 p 所指页块（下一级索引）。这样直到读入主文件页块，在其上查找关键码为 k 的记录

为支持动态的插入和删除操作，需要使用特殊的多分树。使用最广泛的就是B树和B+树

B/B+树的结构特征:

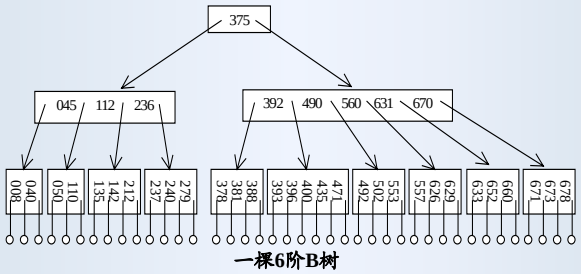
- 都是特殊形式的多分树
- 从树根到叶结点的每个分支严格等长
- 通过限制每个结点的最小分支数，保证树的深度较低
- 能在插入/删除中比较方便地维持较好的结构，基本技术是系统地定义了一套结点分裂和合并规则

B 树/B+ 树结点通常与页块对应（一个页块对应一个树结点）。分支数可达几十或者几百（如256或更大），1000000 文件记录，索引树只有 3 到 4 层结点，高度很低

下面详细讨论B树和B+树的结构和操作

7.6.2 B树

一、B树的定义（m 阶B树）



一棵6阶B树

（根结点2到6个子树，一般分支结点3（6/2）到6棵子树）

一棵m阶的B树或为空，或满足下面特性:

1. 树中每个结点至多有m棵子树;
2. 除根之外的所有分支结点至少有⌈m/2⌉棵子树;
3. 若根结点不是叶，则至少有两棵子树;
4. 叶结点都在同一层，实际上它们不存在（不属索引树）。
5. 有j个子女的非叶结点中包含 j-1 个关键码，且按递增顺序排列，结点中的信息为 $(p_0, k_1, p_1, k_2, \ldots, k_{j-1}, p_{j-1})$

说明: k_i (i = 1, .., j-1) 为关键码，$k_i < k_{i+1}$(i = 1,…,j-2)。

p_i (i = 0,…,j-1) 是指向子树的指针:

- p_i(i = 0,…,j-2)所指子树中的关键码都小于k_{i+1}
- p_i(i = 1,…,j-1)所指子树中的关键码都大于k_{i-1}

与关键码k_i一起还有相应的实际记录，存储相关数据。

实际应用中，B 树结点大小 m 的选择和内外存交换的缓冲区或者外存存储块的大小有关（例如m = 256）。

结点类型可以如下说明:

```
#define m 256

typedef struct BTNode {
    int          keyNum;          /* 实际关键字个数 */
    struct BTNode  *parent;       /* 指向父结点*/
    struct BTNode  *ptr[m];       /* 子树指针向量 */
    KeyType      key[m-1];        /* 关键字向量 */
    Record       *recPtr[m-1];/* 指向具体记录 */
} BTNode, *BTree;
```


二. B树的运算

1. 在B树中的检索关键码key

在根结点的关键码集合中做（二分）检索，若有 key== k_i 则成功；否则若存在 i 使key在k_i 和 k_{i+1} 之间，则沿 p_i 继续检索。重复这个过程，直到检索成功，或找不到 p_i 时检索失败。

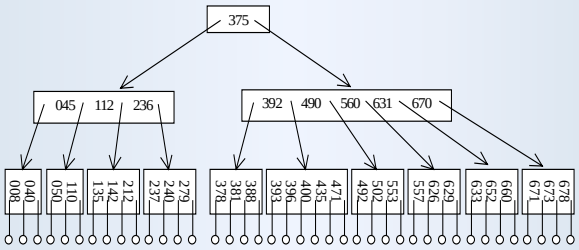
性能分析：在B树里检索包含两种基本操作：1）在B树中找对应结点；2）在结点中找相关的关键字。

B树通常存在磁盘，操作1需访问磁盘（把p所指结点读入内存），后一操作在内存进行。磁盘比内存慢得多，磁盘检索次数（待查关键字的结点在B树的层数）是决定检索效率的关键。

最坏情况：n个关键字的m阶B树的最大深度为$\log_{\lceil m/2 \rceil}(n/2) + 1$

若m = 256，n = 1000000，B树最大深度为 4。

例：在下面B树里检索关键码211，537

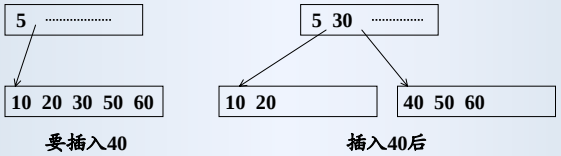


2. 插入结点

深度为 h 的 m 阶B树，新结点一般插入到h层:

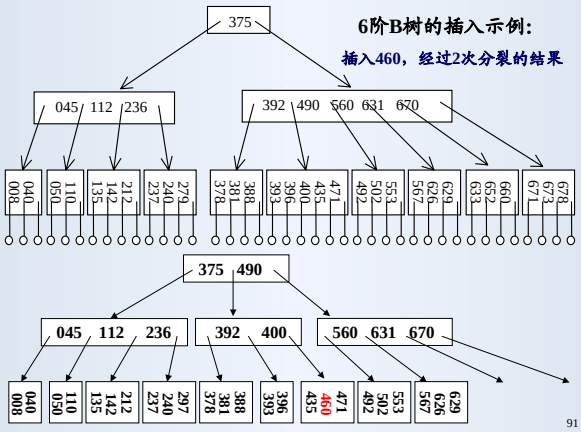
1. 首先检索正确插入位置，用堆栈记录经过的结点。
2. 若应该插入的结点中关键码个数小于m-1，直接插入。
3. 若被插入结点中关键码个数等于m-1，则引起结点“分裂”:
   1. 将结点关键码从小到大分为3部分，中间 1 个关键码，左右两部分关键码个数相同（或相差1）；
   2. 建一个新结点，把关键码大的那一部分移入新结点;
   3. 把中间关键码和新结点指针一起插入原结点的父结点。
4. 插入父结点的情况与上面一样，如果父结点已满，它也会分裂并把一个关键码向上传。
5. 如果这种分裂传播到根，导致根结点分裂，树增高一层。

例：在下面 6 阶B树中插入



要插入40　　插入40后

如果父结点已满，不能直接插入，就会继续分裂，并把作为分界的关键码上传。



3. 删除结点

1. 首先找到该关键码所在结点 N，然后根据不同情况删除。
2. 如果 N 在第h层，根据情况分别处理。该结点：
   1) 关键码数大于⌈m/2⌉-1，则简单删去该关键码 k_i;
   2) 关键码数等于⌈m/2⌉-1，其左（或右）兄弟结点中的关键码数多于⌈m/2⌉－1，删除关键码后把结点N的左（或右）兄弟结点和父结点的信息调整过来，维持树结构;
   3) 兄弟结点中的关键码数均等于⌈m/2⌉－1，需合并。设右兄弟结点由父结点中指针p_j所指。删去关键码后把结点N剩余关键码和指针连同父结点中的关键码k_j，一起合并到p_j所指结点中（没有右兄弟时合并到左兄弟结点）。
3. 若结点层数小于h，设删除的是结点中第 i 个关键码 k_i，则用p_i所指的子树中的最小关键码k与k_i互换，然后像前面一样处理h层的删除关键码k_i 。

92

情况2)的调整方法:

设父结点中的信息为：（$p_0, k_1,p_1,k_2,p_2,\ldots,k_x p_x$),

由p_i指向被删除关键码的结点，其信息为:

（$p'_0, k'_1,p'_1,k'_2,p'_2,\ldots,k'_y p'_y$),

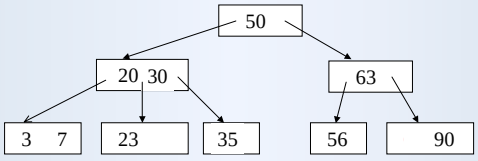
左兄弟结点中的信息为:

（$p''_0, k''_1,p''_1,k''_2,p''_2,\ldots,k''_z p''_z$)。z>⌈m/2⌉

将$k''_{\lceil (z+y)/2+1 \rceil}$上移到父结点中，并将左兄第中大于$k''_{\lceil (z+y)/2+1 \rceil}$及父结点中的$k_i$移到被删除关键码的结点中

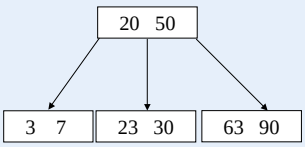
与右兄弟结点调整的情况与此类似

93



(a) 一棵三阶B树
(b) 从(a)中删除关键码66
(c) 从(b)中删除关键码42

94



(d) 从 (c)中删除关键码35
(e) 从 (d)中删除关键码56

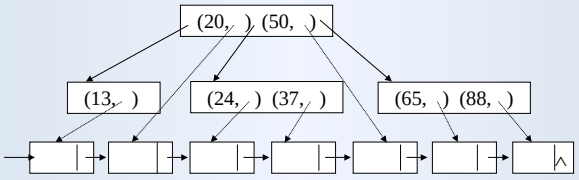
95

三. B树在索引文件中的应用

实现索引顺序文件时用B树作为文件索引（下图实线箭头）

B树里的每个关键码关联一个文件记录索引（下图虚线箭头）

主文件页块顺序链接，支持内容的顺序访问



96

B 树总结:

1. B树是一种多分树，分支数可在一定范围内变动，所有叶结点维持在同一层，维持每个分支结点的最小分支数;
2. 由于结点有最小分支数的保证，因此树高度与叶结点的个数（以及整个树中的结点个数）成对数关系，保证了树检索和插入、删除操作的效率;
3. B 树检索分为结点内检索（可用二分法）和顺着指针访问结点两种动作;
4. 关键码插入可能导致结点分裂，最终导致树增高；结点删除可能导致结点合并，最终导致树高度下降。
5. B 树的分支数通常很大（100 或者更多），一个结点存入一个外存块，作为外存文件的索引。B 树通常深度很小，使通过B树索引的文件访问能高效进行。

7.6.3 B+树

B+树是一种与B树类似的结构。

定义：一棵m阶的B+树或为空树，或是满足下列条件的树:

1. 树中每个结点至多有m棵子树;
2. 除根之外的所有分支结点至少有⌈m/2⌉棵子树;
3. 若根结点不是叶子结点，则至少有两棵子树;
4. 有 n 棵子树的结点有 n 个关键码;
5. 叶结点存数据文件记录的关键码及指向它的指针（或存数据文件中每块的最大关键码及指向指针），在块里按关键码值顺 序排列。每个叶结点可看成一个基本索引块。
6. 分支结点可看成是索引的索引，结点中仅包含它的各子结点中最大关键码及指向子结点的指针。

m阶的B+树和B树的差异在于:

1. 有n棵子树的结点中含有n个关键码（B树有m-1个）
2. 叶结点包含了全部关键码信息及指向含这些关键码的记录的指针。叶结点本身依关键码顺序排列（B树的分支结点和叶结点的全体包含了所有关键码信息）
3. 所有非终端结点可以看成索引，其中仅包含其子树（的根结点）中的最大关键码

B+树通常有两个头指针，一个指向根结点，另一个指向关键码最小的叶子结点。

B+树的设计中也可统一用子树的最小关键码

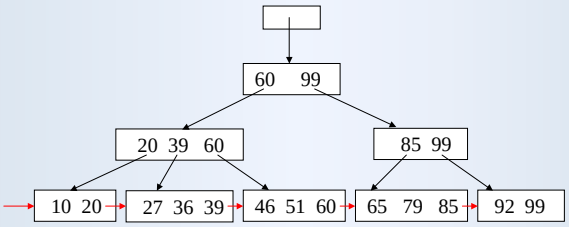


图6.26 一棵三阶的B+树

二. B+树的运算

1, B+树的检索（两种方式）:

1. 从根结点开始检索。如果存在 i 使 key 位于 k_i 和 k_{i+1} 之间，则沿p_{i+1}继续检索，重复上述过程，直到检索到叶结点，若key == k_i 则检索成功；否则检索失败。
2. 从最小关键码开始沿叶结点链顺序检索。

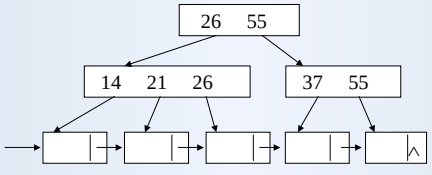
2, B+ 树的插入和B树的插入相似，但插入一定在叶结点进行，当结点中已有m个关键码时，结点分裂

3, B+ 树删除仅在叶结点进行，在与兄弟结点合并时，父结点中作为分界的关键码不放入合并后的结点，直接删除

B+ 树的结构比较规整，插入/删除操作的实现比B树简单。

三. B+树在索引顺序文件中的应用

主文件的每个页块作B+树的一个外部结点，并且这些页块之间连成单链。 B+树的树叶层是主文件的稀疏索引，整个B+树构成多级索引。索引项就是B+树中一个关键码和它对应的指针所构成的二元组。



用B+树组织索引顺序文件

B+ 树总结：

1. B+ 树也是多分树，通过允许结点分支数在一定范围内变动的方式维持叶结点在同一层，所有索引关键码都放在叶。
2. 通过结点的最小分支数，保证检索和插入、删除的效率。检索也分为结点内检索（可以用二分法）和顺着指针访问结点两种动作。插入可能导致结点分裂，最终导致树增高；结点删除可能导致结点合并，最终导致树高度下降。
3. 分支结点中关键码只起区分检索路径的作用，处理较灵活。且B+ 树结构比较规整，因此实际使用比B树更多。

许多数据库管理系统都以B/B+树作为基本索引结构，而B+树由于其操作的实现比较简单等优点，使用更为广泛。

7.7 字典的各种表示的比较

1. 静态字典

1）顺序表示

- 顺序检索：简单，常用于未排序字典，检索效率不高
- 二分法检索：只能用于元素排序的字典，检索效率高

2）散列表示：通过合适的散列函数可以得到满意的存取速度。散列表示经常出现碰撞与堆积现象，可能增加检索长度。

- 负载因子较小时检索效率很高（接近常量时间），负载因子较大时接近线性
- 内部冲突消解技术不易支持元素删除操作

3）二叉树表示：用二叉排序树表示。一般二叉排序树不能保证高效检索。静态字典可以考虑用最佳二叉排序树，平均检索长度为$O(\log_2 n)$。

2. 动态字典

1）顺序表示：元素不排序时检索效率低；元素排序时插入删除需要移动大量元素，效率低。只适合元素个数较少的字典。

2）散列表示：采用冲突内消解技术，元素删除不易处理。采用外溢出区或散列桶，元素数量很大时接近线性。

3）平衡二叉排序树（AVL树）表示：可以始终维持较高的检索效率，但操作的实现比较复杂。

4）人们也常常采用混合技术，例如用二叉排序树或AVL树作为散列字典的存储桶。

3. 较大的必须存放在外存贮器上的字典

通常采用B树或B+树表示。B+树是B树的变种。B树和B+树都能动态调整，保持均衡，保持较高的检索效率。

小　　结

字典是一种特殊的集合，最主要操作为字典中元素的检索。

本章介绍字典的各种存储表示及相应检索方法，包括各种线性表示（例如：顺序表表示，链表表示和目录表表示等）、各种散列表示（例如：开地址法、拉链法和桶散列等）、各种二叉树表示（例如：一般的二叉排序树、最佳二叉排序树和平衡的二叉排序树等）以及各种多叉树表示（例如B树和B+树）。

其中顺序表示的线性表、开地址法处理碰撞的散列表和最佳二叉排序树主要使用于静态或基本静态的字典；链表表示的线性表、拉链法处理碰撞的散列表、桶散列结构、平衡的二叉排序树、B树和B+树等较多地考虑到元素的动态处理，适合于组织各种动态的字典；其中B树、B+树和桶散列结构主要使用于外存的大型字典表示。