jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

1. 查找树
2. 满树、拟满树和丰满树
3. 堆和堆排序
4. 平衡树
5. 最佳查找树
6. Huffman算法
7. B-树
8. Trie结构
9. 解答树

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

1 查找树

是否可以找到一种结构和相应的方法，对于一个有序的list，同时满足：

可以快速的查找（类似顺序存储中的二分查找）
可以快速的插入和删除（类似链表）

jyang
@cs.ecnu.edu.cn

1 查找树

查找树——结合链接存储法的优点和二分查找法的效率
（A Binary Search Tree,BST）

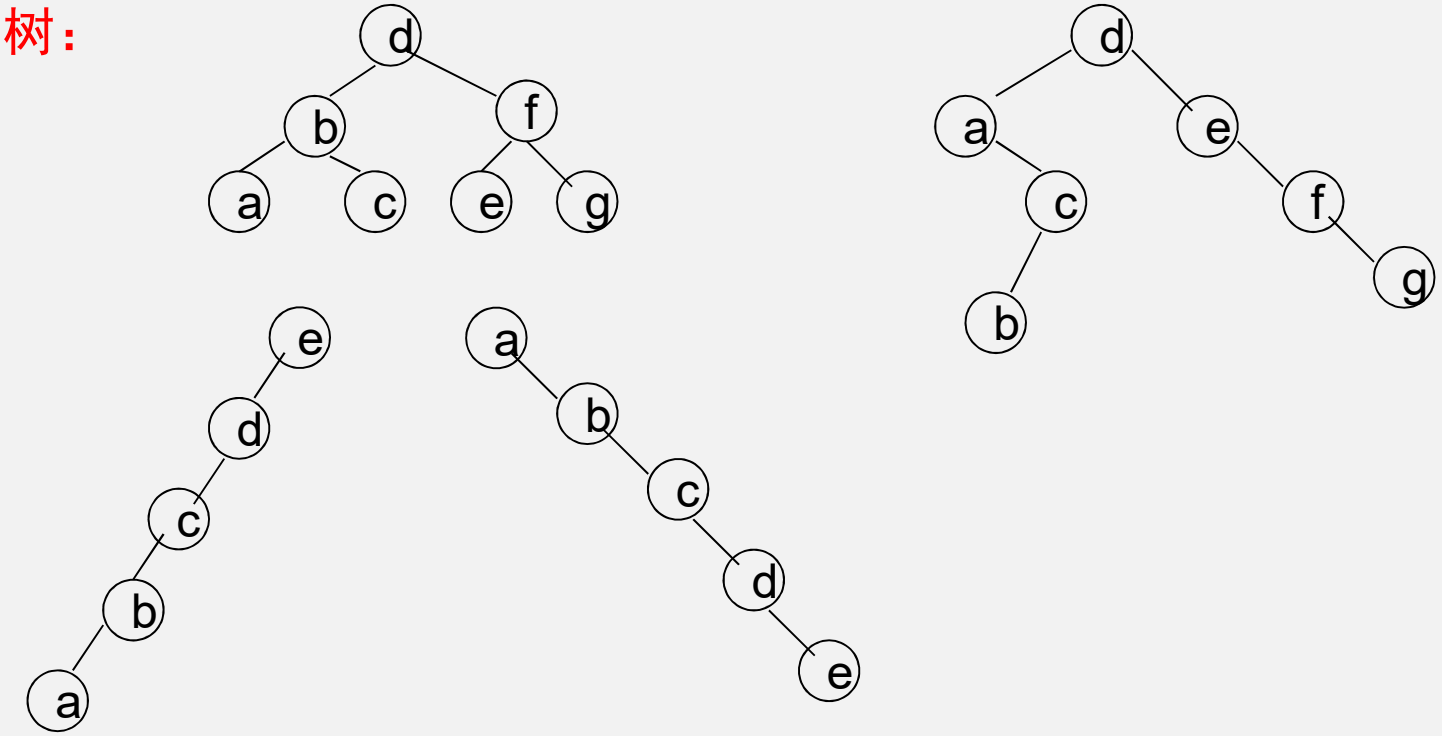
定义：查找树T是一棵二叉树T，它或空，或满足下面三个条件：
（1）如果树T的根节点的左子树非空，那么左子树中的所有结点的键值都小于T的根结点的键值。
（2）如果树T的根节点的右子树非空，那么右子树中的所有结点的键值都大于T的根结点的键值。
（3）树T的根结点的左、右子树也都是查找树。
（no equal keys）
定义方法二：对于一棵给定的二叉树T,如果树T中的结点的中序是排好序的，那么我们称树T是一棵查找树。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

1 查找树

查找树：



基本操作：查找、插入、删除（为讨论方便，只考虑结点关键字）

1 查找树

```
typedef struct BiSearchNode{

        char data; //存放可以比较大小的key, 在BST中唯一
        struct  BiSearchNode *lchild;
        struct  BiSearchNode * rchild;
}NODE;
```

jyang
@cs.ecnu.edu.cn

查找

在给定的查找树t中，如何查找具有给定键值target的结点？

```
——recursive
NODE* search(NODE* sub_root, char target)
{
  if sub_root == NULL || subroot->data == target)
      return sub_root;
  else if (sub_root->data < target)
         return search (sub_root->lchild, target);
  else   return search (sub_root->rchild, target);
}
```

jyang
@cs.ecnu.edu.cn

查找

在给定的查找树t中，如何查找具有给定键值target的结点。
——nonrecursive

（1）如果t空，查找失败，算法结束；否则转（2）

（2）如果t->data等于target，查找成功，算法结束。否则转（3）

（3）如果a小于t->data，那么t=t->lchild,转（1）；
　　否则，　　t=t->rchild，转（1）；

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

1 查找树

查找

在给定的查找树t中，如何查找具有给定键值a的结点。

——nonrecursive

```
NODE* search (NODE* sub_root, char target)
{
  while (sub_root != NULL && subroot->data != target)
      if (sub_root->data < target)
                  sub_root=sub_root->rchild;
      else
                  sub_root=sub_root->lchild;

  return sub_root;
}
```

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

1 查找树

```
void search(NODE *t,char target,NODE **p_p,NODE **p_q)
//p_p,p_q指向指针变量的指针。  *p_q是指针，指向当前结点；  *p_p指向当前结点的父结点
{      *p_p=NULL;
       *p_q=t;
       while (*p_q != NULL)
       { if (*p_q)->data == target) return; //找到结点“a
          *p_p= * p_q; //准备“下移”
          if (target<(*p_q)->data)
              *p_q=(*p_q)->lchild;
          else  * p_q=(*p_q)->rchild;
       }//  算法执行到结束，若*p_q不空，则*p_q所指的结点就是a结点（找到了），而*p_p指向它的父结点。若此时*p_p为空，则a结点就是根结点。
    假若*p_q为空，则表示找不到target结点。此时*p_p有两种情况：
    （1）若为空，则查找树t为空树；
    （2） *p_p不为空，*p_p指向查找路径的最后一个结点。
}
```

jyang
@cs.ecnu.edu.cn

查找

函数调用方法： Search(t,target,&p,&q)

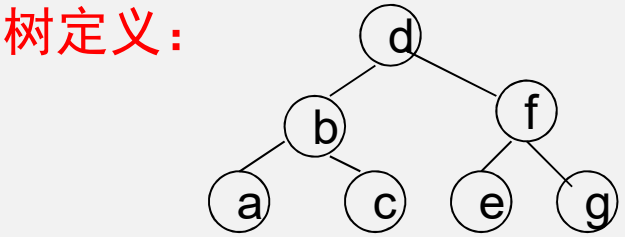
	函数执行后的结果
1.树空	p为空，q为空
2.树不空 找不到值为target的结点	p指向查找路径中最后一个结点，q为空
3.树不空 找到值为target的结点 这个结点就是树根	q指向键值为target的根节点 p为空
4. 树不空 找到值为target的结点， 这个结点不在树根上	q指向键值为target的结点 p指向此结点的父结点

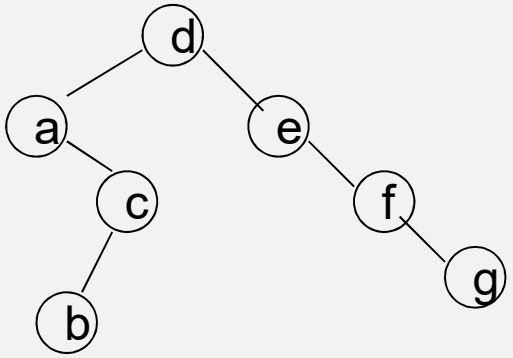
jyang
@cs.ecnu.edu.cn

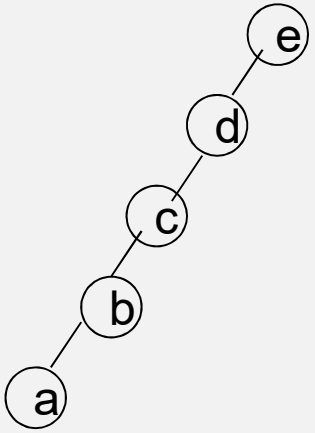
第六章
树的查找和树的应用

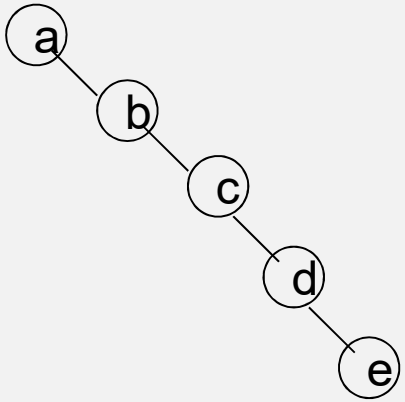
1 查找树

查找树定义：









jyang
@cs.ecnu.edu.cn

插入——在给定的查找树T中插入一个键值为target的结点。（插入之后要保证查找树的特性）

（1）已有键值为target的结点，不执行插入操作；
（2）树空，插入的结点成为树根；
（3）树不空， 寻找合适的位置插入

问题：插入在哪里？

插入的结点作为原来树中某个结点（在查找target时，查找路径中的最后一个结点）的子结点（插入之前先进行了一次未命中的查找，插入后的新结点是叶子结点）

jyang
@cs.ecnu.edu.cn

插入
——recursive

```
int search_and_insert (NODE **sub_root, char new_data)
{
   if  ( ( *sub_root ) == NULL)
       {*sub_root = ……; return (0);//构造结点成为树根
        }
   else if (new_data <(*sub_root)->data)
       return search_and_insert(&(*sub_root)->lchild,new_data);
   else if (new_data >(*sub_root)->data)
       return search_and_insert(&(*sub_root)->rchild,new_data);
   else  return(1);// 要插入的节点值重复则插入失败
}
```

jyang
@cs.ecnu.edu.cn

1 查找树

插入

——nonrecursive

```
int insert (NODE **p_t, char a)
{
      NODE *p,*q,*r;
      Search (*p_t,a,&p;&q);
      if  (q!=NULL) return (1);
      r=·····  //构造结点r
      if  (p==NULL)  *p_t=r;
      else  if (p->data>a)  p->lchild=r;
                 else p-rchild=r;
      return (0);
}
```

如果t指向查找树T，插入

i=insert（&t，a）；

如果T为空，则插入操作后，t指向新的根节点

i=0，插入成功；i=1，a已存在，插入失败

删除

要求：

找到要删除的结点，将它删去，要求删除后的二叉树仍是一棵查找树。

原则：

若被删结点有左子树，那么被删结点的位置由它的左子树的根结点来充当，它的右子树被链接在左子树的最右下方；否则，被删结点的位置由它的右子树的根结点来充当。

删除——删除键值为a的结点

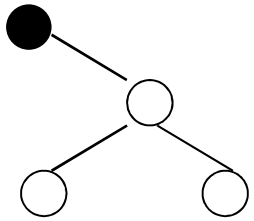
（1）先search，确定被删结点在树中的位置。
（2）如果被删结点不在树中，算法结束；
（3）如果被删结点在树中，则：
（i）被删结点是根结点，
(a)若无左子结点，则用被删结点的右子树作为删除后的树
(b)若有左子结点，则用被删结点的左子结点作为根，同时把被删结点的右子树作为被删结点的左子树按中序最后一个结点的右子树。
（ii）被删结点不是根结点，……（可以看作是子树的根）

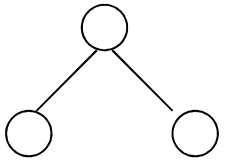
（iii）回收被删结点的存储单元，算法结束

jyang
@cs.ecnu.edu.cn

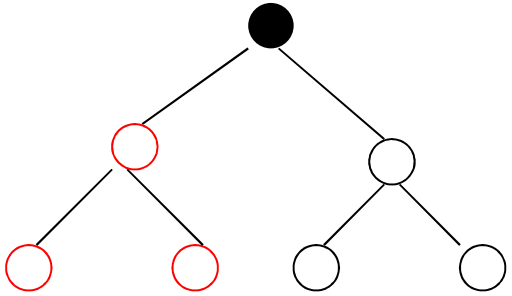
1 查找树

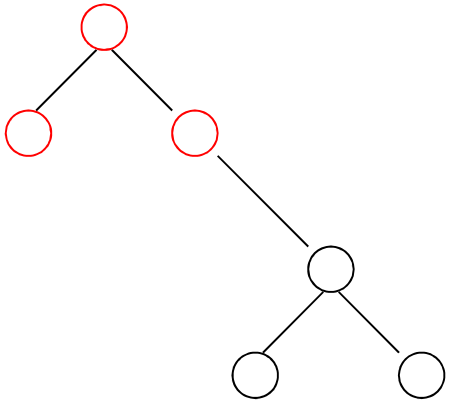
①它没有左子结点





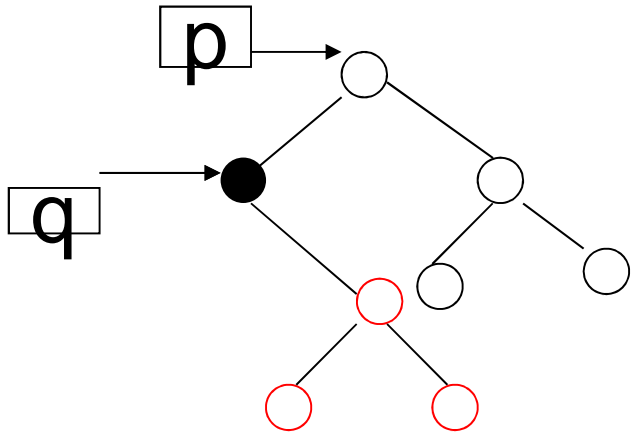
②它有左子结点



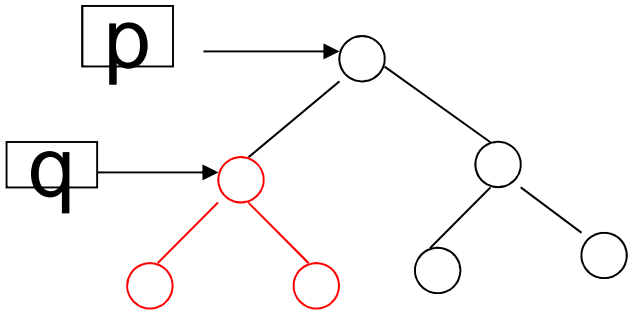


jyang
@cs.ecnu.edu.cn

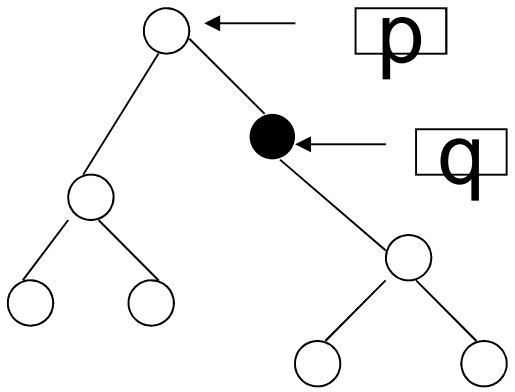
③被删结点是父结点的左子结点



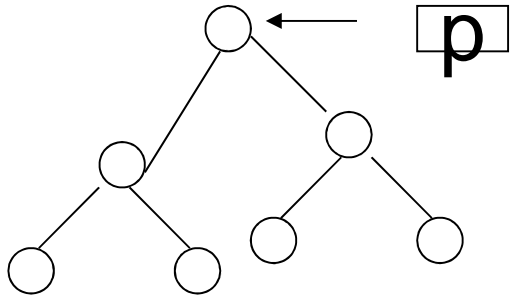
把被删除结点的右子树作为被删除结点的父亲的左子树



④被删结点是父结点的右子结点

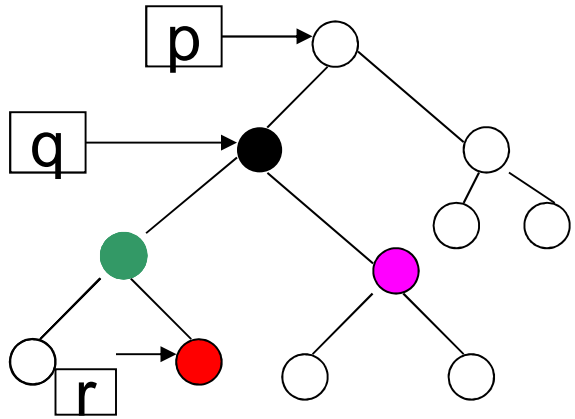


改造:把被删除结点右子树作为被删除结点父亲的右子树



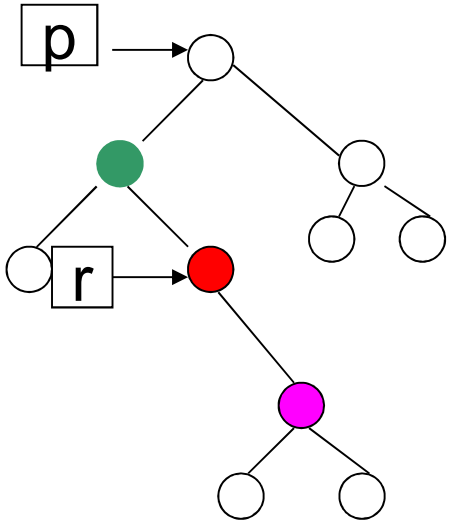
jyang
@cs.ecnu.edu.cn

⑤被删结点有左子树，是父结点的左子结点，



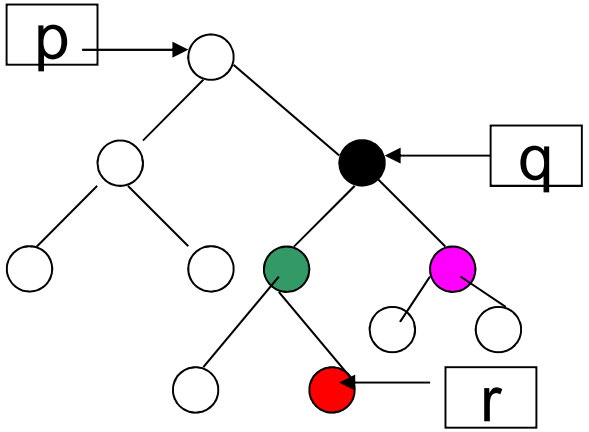
改造(1)把被删除结点右子树作为被删除结点左子树按中序最后一个结点的右子树;

(2)把被删除结点左子树作为被删除结点父亲的左子树



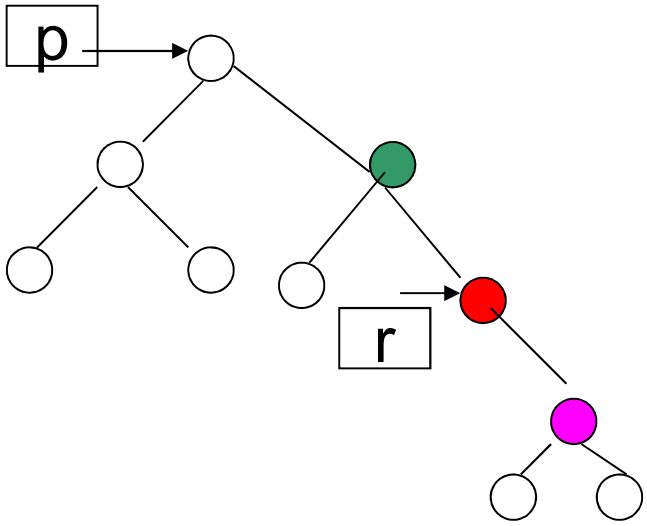
jyang
@cs.ecnu.edu.cn

⑥被删结点有左子树，是父结点的右子结点



改造(1)把被删除结点右子树作为被删除结点左子树按中序最后一个结点的右子树;

(2)把被删除结点左子树作为被删除结点父亲的右子树



jyang
@cs.ecnu.edu.cn

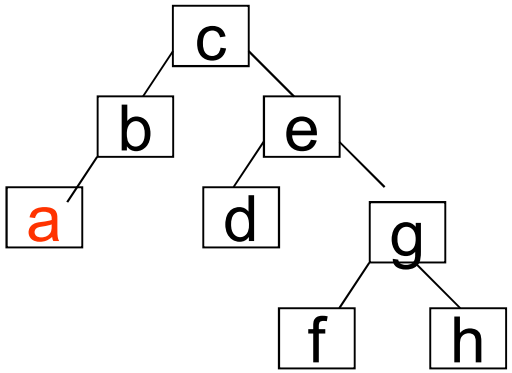
删除

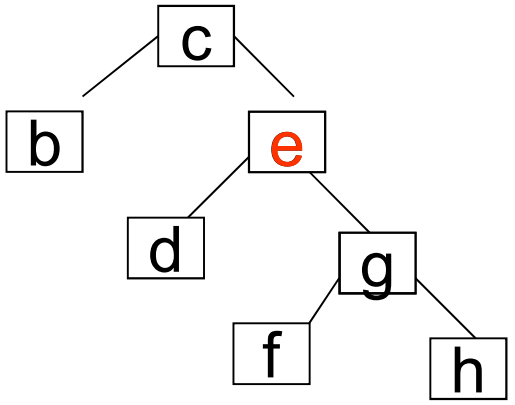
```
int delete (NODE  **p_t, char to_delete)
{
        NODE *p,*q,*r;
        search (*p_t, to_delete,&p;&q);
                //执行后q如果不空，指向找到的结点
                //待删除结点如果不是根，p指向q的父结点

        If  (q==NULL) return (1);  //结点“to_delete” 不在树中
        If  (p==NULL)   //p为NULL，找到的结点为根，被删结点为根结点
              if  (q->lchild==NULL)  *p_t=q->rchild;
              else { r=q->lchild;
                     while (r->rchid!=NULL) r=r->rchild; //找“最右”
                      r->rchild=q->rchild;
                      *p_t =q->lchild;
              }
        else ···········
```

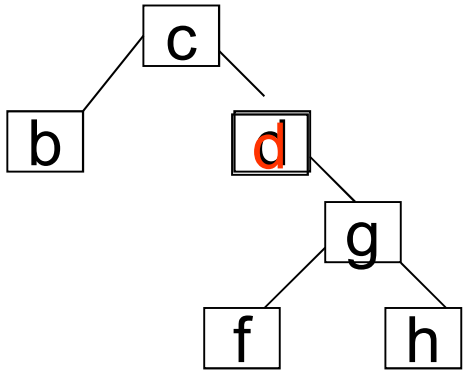
删除

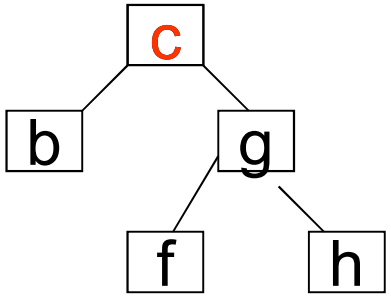
```
int delete (NODE ** p_t,char a)
{……
//将要被删除的结点q不是根，作为子树的根来处理，同时完成父结点p的指针变化
       else if  (q->lchild == NULL )
                        if  (q==p->lchild)  p->lchild=q->rchild;
                        else p->rchild = q->rchild;
              else{     r=q-lchild;

                        while (r->rchid!=NULL) r=r->rchild;

                        r->rchild=q->rchild;
                        if  (q==p->lchild)  p-lchild=q->lchild;
                             else p->rchild = q->lchild;
                   }
free  (q) ;
return (0);
```

删除——示例









查找算法的效率分析

假定k在查找树中，查找k所需要的比较次数：从根结点到结点k的树枝长度（λk）加1，因此，可以得到：

MAX（二叉查找法）= $\max\{1+\lambda k | \text{树T中的k}\}$

AVG（二叉查找法）= $\sum_{k\ of\ \mathrm{T}} P(k)(1+\lambda k)$

其中， p（k）是结点k的相对使用概率。如果考虑所有结点的相对使用概率相等，则有

$AVG（二叉查找法）= \frac{1}{n}\sum_{k\ of\ T}(1+\lambda k)$

显然，当树中结点尽量靠近树根时，AVG（二叉查找法）的值达到最小。当查找树退化成链接表时，AVG的值达到最大。

jyang
@cs.ecnu.edu.cn

2 满树、拟满树和丰满树

在查找树中，查找结点的最大查找时间：

最坏情况：树退化成链，O(n)
最好情况：除了最后一层，树满 O（$\log_2 n$）

为了使最大查找时间、平均查找时间达到最小，我们应使得从根到其他结点的路径尽量短。为此，给出若干定义：

满二叉树 完全二叉树（拟满二叉树 ） 丰满二叉树

2 满树、拟满树和丰满树

满二叉树 拟满二叉树 丰满二叉树

设二叉树T有n个结点，$i=\lfloor \log_2(n+1) \rfloor$，$r=n-(2^i-1)$。如果其中$(2^i-1)$个结点放满第0至第$(i-1)$层。

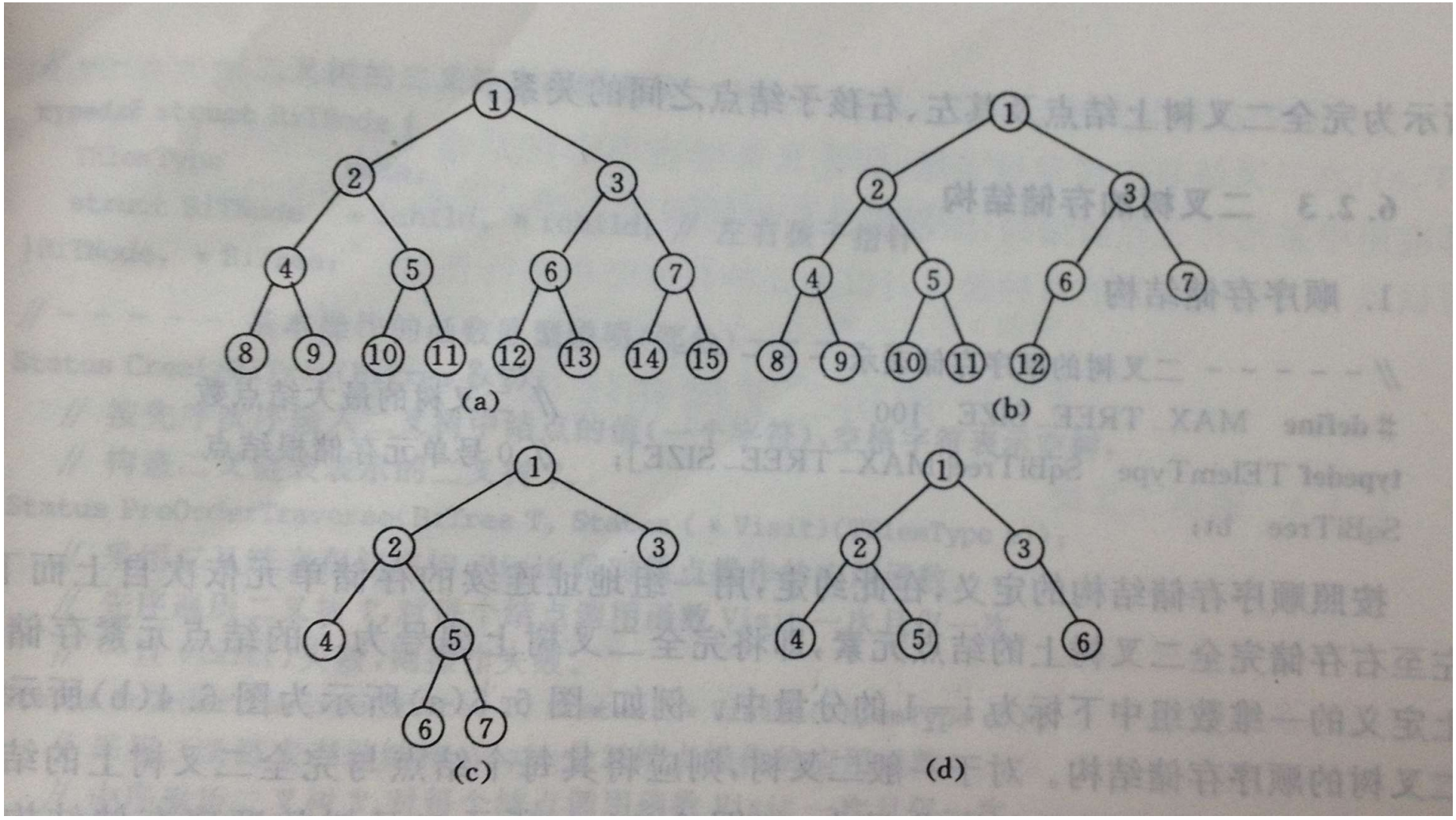
（1）若r=0，则称树T是一棵满二叉树，简称满树。
（2）若r>0，且剩下的r个结点尽量靠左地排列在第i层上，则称树T是一棵拟满二叉树（完全二叉树）。
（3）若r>0，且剩下的r个结点随意分布在第i层，则称树T是一棵丰满二叉树。

若T同时也是查找树，则称树T是满查找树、拟满查找树、丰满查找树。

如果树T是满树，那么树T一定是拟满树（完全二叉树）和丰满树。
如果树T是拟满树（完全二叉树），那么树T一定是丰满树。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用



2 满树、拟满树和丰满树

对于n个结点的任意序列，我们可以用平分法构造出该结点序列的丰满树，其构造算法如下：

（1）如果结点序列为空，那么得到一棵空的二叉树。
（2）如果序列中有n>=1个结点，$k_0, k_1, \cdots\cdots, k_{n-1}$，那么m=$\lfloor (n-1)/2 \rfloor$，所求的树由根结点$k_m$，以及它的左子树$T_l$和右子树$T_r$所组成。其中，$T_l$和$T_r$分别是用平分法由两个结点序列得到的丰满树。

如果原来的结点序列是排好序的，那么得到的树是丰满查找树。

若丰满树T中有n个结点，则树T有$\lfloor \log_2 n \rfloor$+1层。当$n=2^t-1$时，丰满树刚好是一棵满树。

jyang
@cs.ecnu.edu.cn

2 满树、拟满树和丰满树

对于n个结点的丰满查找树，如果树中所有结点都具有相同的使用概率1/n，那么AVG（二叉查找法） ≈ $\log_2 n$
由于具有n个结点的丰满查找树具有$\lfloor \log_2 n \rfloor$+1层，
所以 MAX（二叉查找法）≈$\log_2 n$

用二分查找法查找已排序的结点序列 与用二叉查找法查找由F构造的丰满查找树 所需要的比较次数是一样的。

当查找树退化成线性链表时，则
AVG（二叉查找法）=（n+1）/2
MAX（二叉查找法）=n

查找树与顺序存储的线性表相比，它优点是向查找树插入和删除结点比较容易。（如何保证插入和删除后不破坏树的形态？）

平衡树——定义

对于二叉查找树来说，丰满查找树最为有利。但对于丰满查找树进行结点的插入或删除后，很容易变成非丰满查找树。

二叉树的高度：……

平衡度（平衡因子）：设k是二叉树T的结点，T_{kl}和T_{kr}分别是结点k的左子树和右子树，我们称T_{kr}和T_{kl}的高度之差为结点k的平衡度。

平衡树：如果二叉树T中每个结点k的平衡度的绝对值都小于等于1（即左子树和右子树的最多差1），那么称T是一棵平衡树。

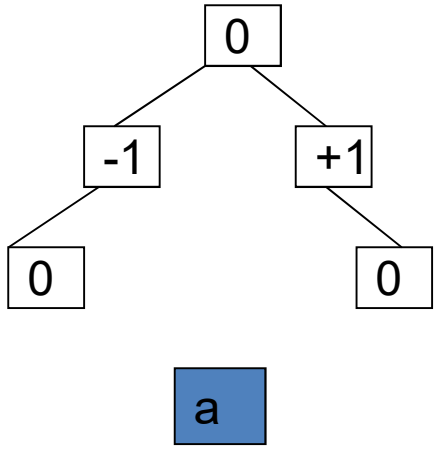
jyang
@cs.ecnu.edu.cn

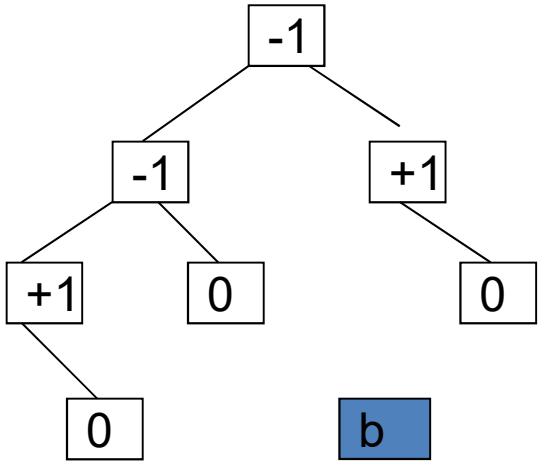
4 平衡树

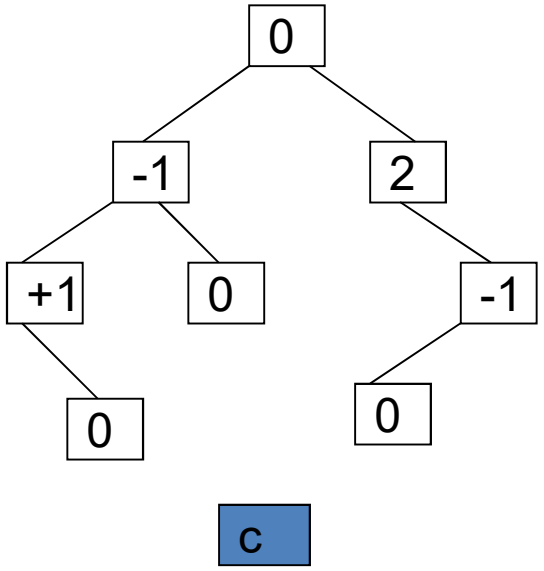
平衡树——定义

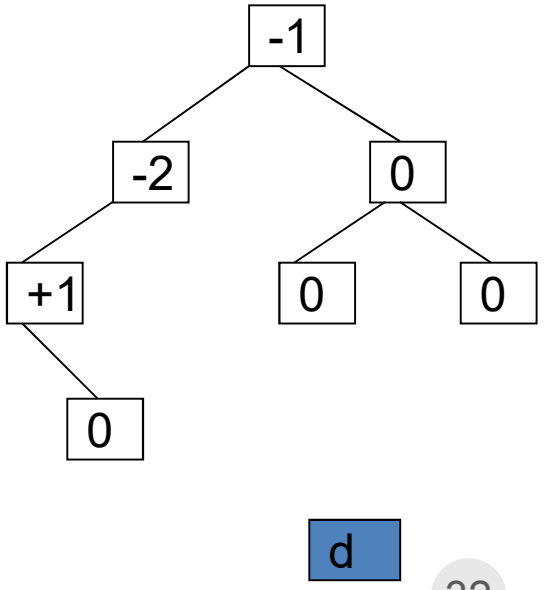
丰满树一定是平衡树，
但平衡树不一定是丰满树。

高度为4的AVL树可能具有的
最少结点数？
最多结点数？









4 平衡树

如果T即是平衡树，又是查找树，那么称树T是平衡查找树（即平衡二叉查找树。AVL（Adelson-Velskii and Landis）树，BBT（Balanced Binary Tree））。

可以证明，n个结点的平衡树的树枝的最大长度小于$3/2\log_2 n$
因此，平衡查找树几乎与丰满树一样适用于二叉查找法。

平衡查找树：查找效率和丰满查找树接近。

在插入和删除结点时，如何保证该树仍然是一棵平衡查找树。

平衡树——插入算法Adelson

插入，平衡，改组

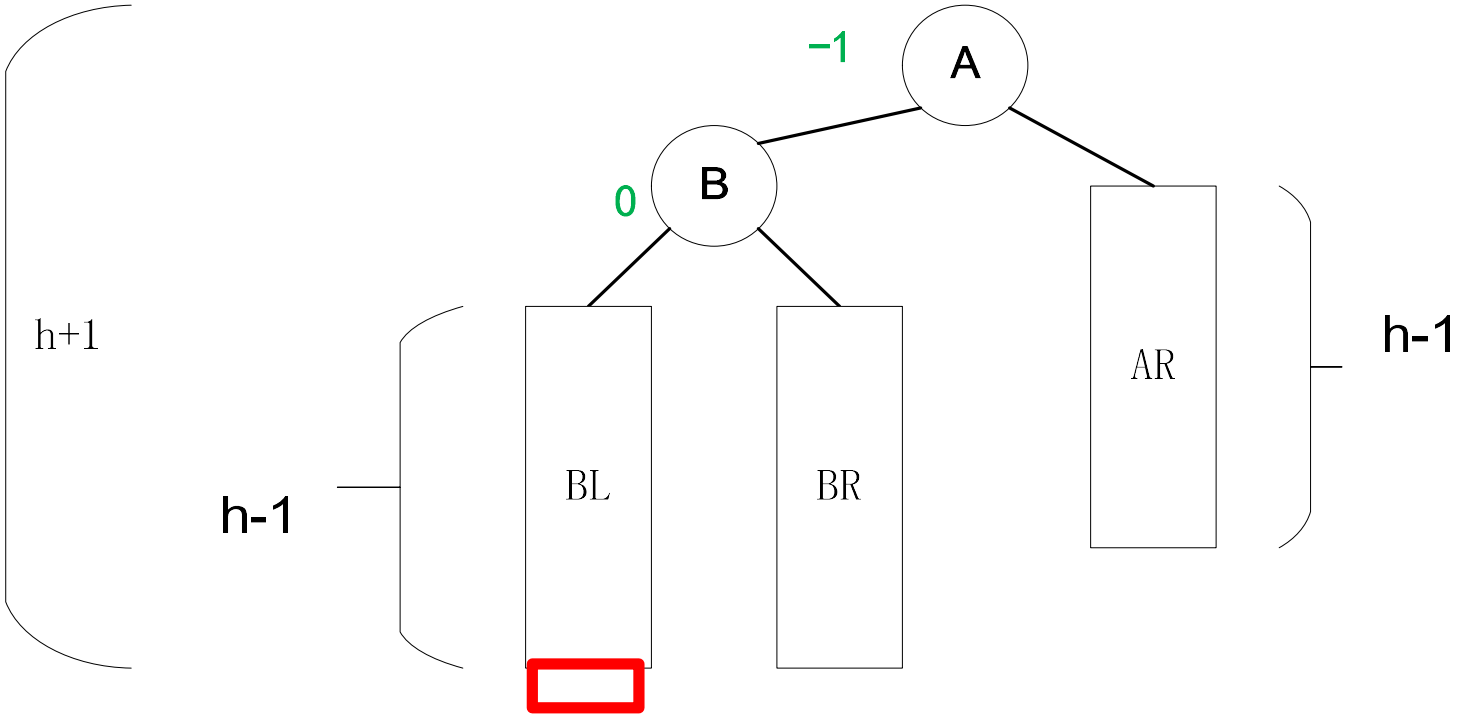
插入：不考虑结点的平衡度，使用在查找树中插入新结点的算法，把新结点k插入树中，同时置新结点平衡度 $\beta(k) = 0$ ；

平衡：调整平衡度，确定需要调整的范围——**最小不平衡子树**。

改组：改变以k_i为根的子树的形态，使得新子树的高度和插入以前以k_i为根的子树的高度相同，同时新子树是一棵平衡查找树。

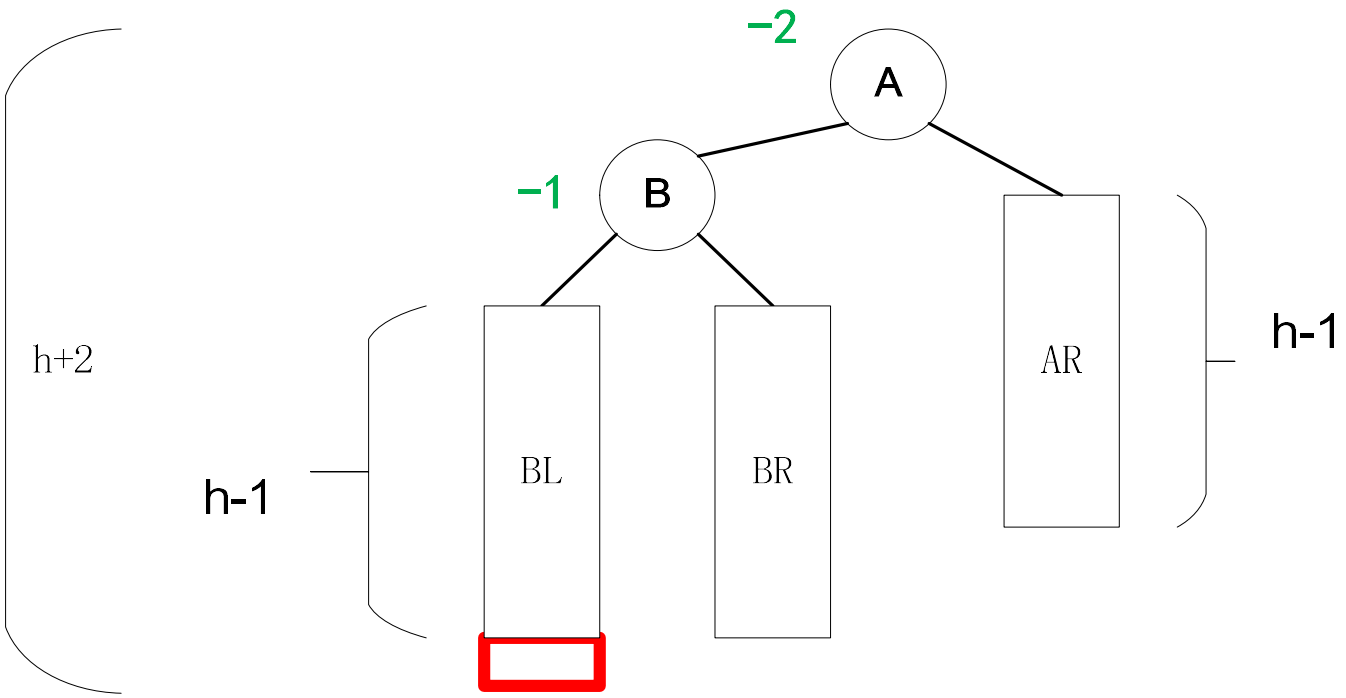
平衡树

改组 LL型：在结点A的左子树上插入新结点导致失去平衡



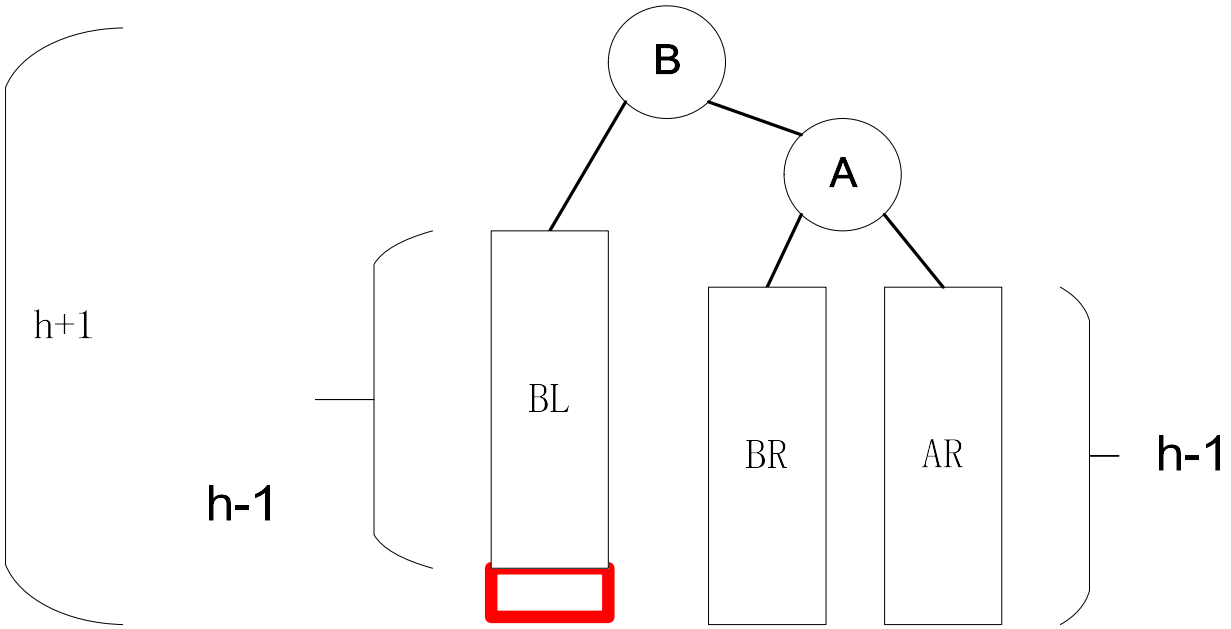
平衡树

改组 LL型：在结点A的左子树上插入新结点导致失去平衡



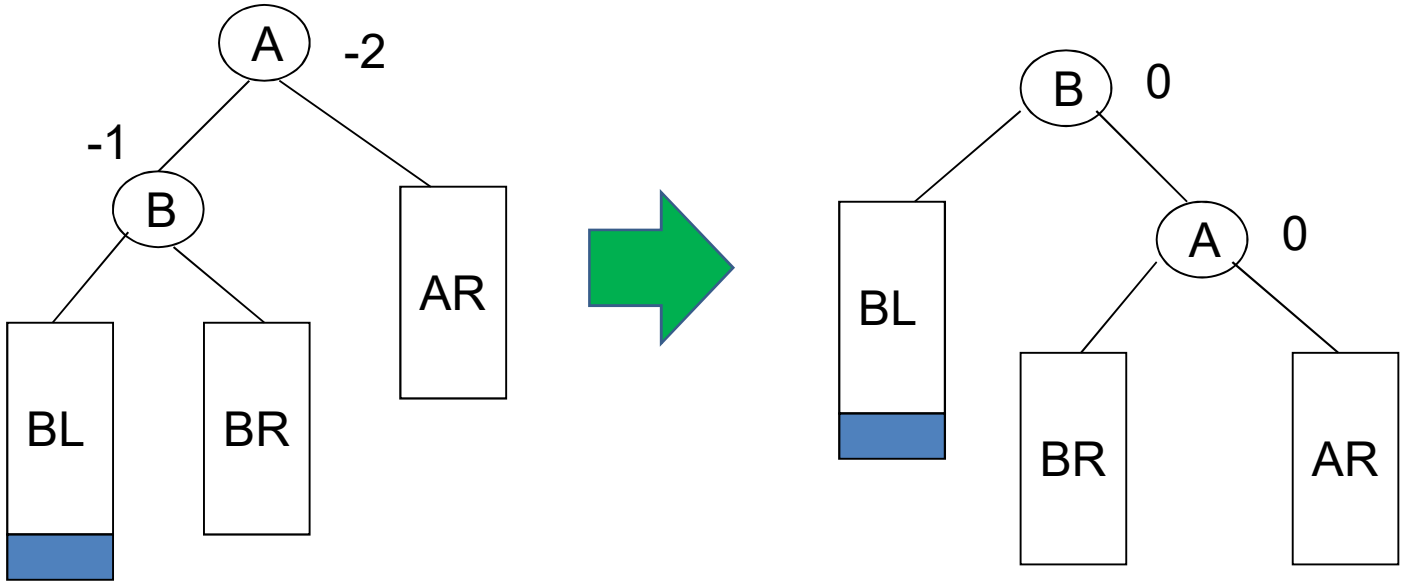
平衡树

改组 LL型：在结点A的左子树上插入新结点导致失去平衡



平衡树

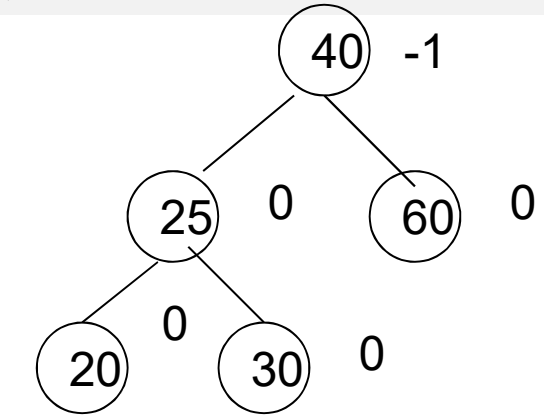
改组 LL型：在结点A的左子树上插入新结点导致失去平衡



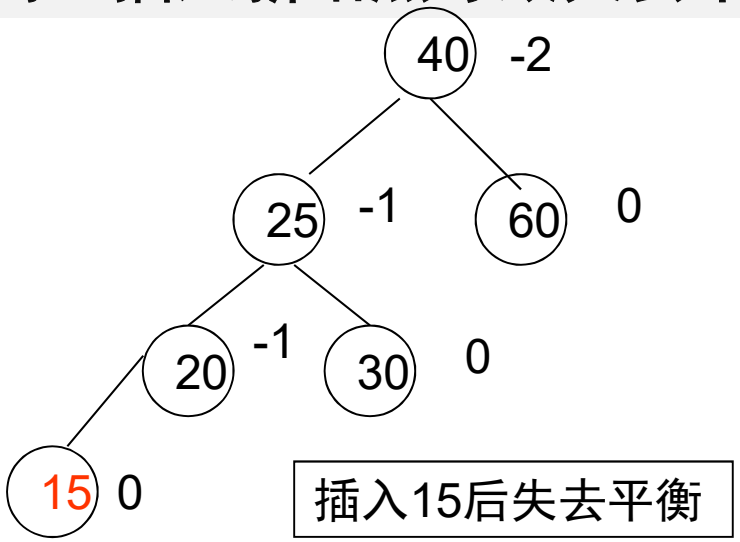
改组方法：将A改为B右子树，B原来的右子树BR改为A的左子树。

平衡树——

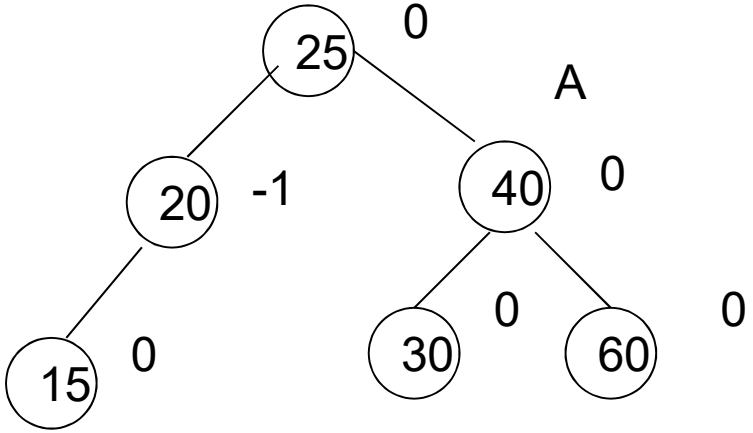
改组 LL型：在结点A的左子树上插入新结点导致失去平衡



一棵平衡二叉树



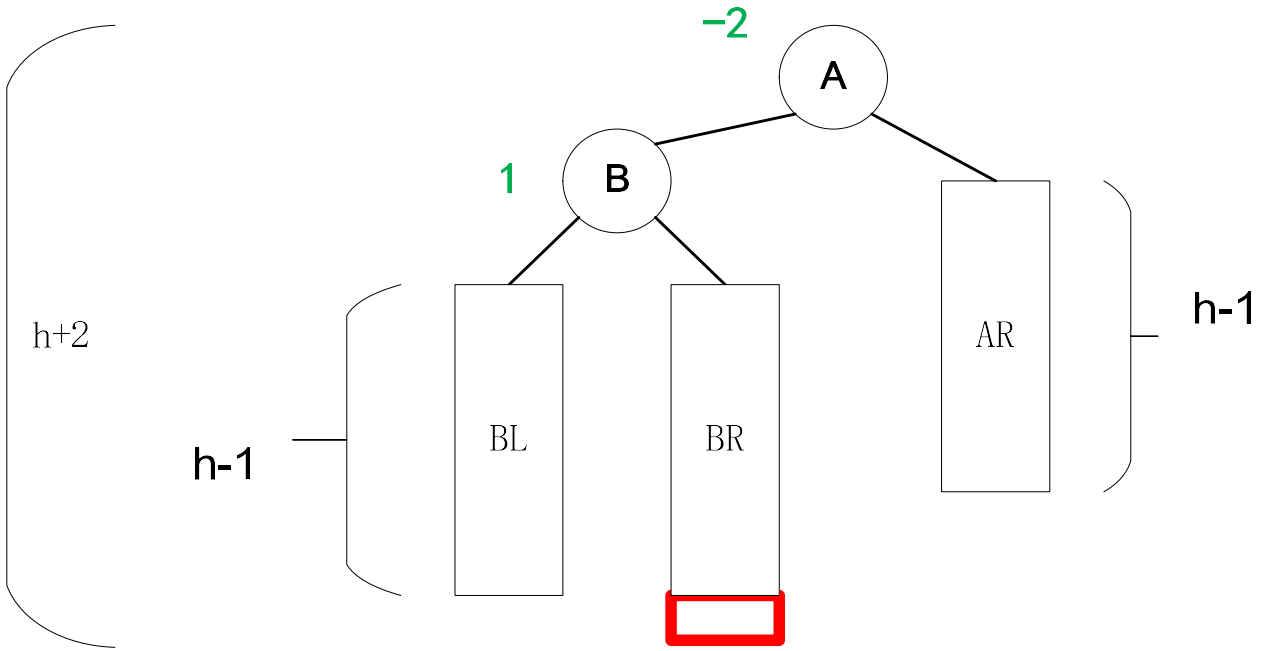
插入15后失去平衡



进行左改组：LL，右旋转

平衡树——

LR型：在结点A左子树的右子树上插入新结点失去平衡

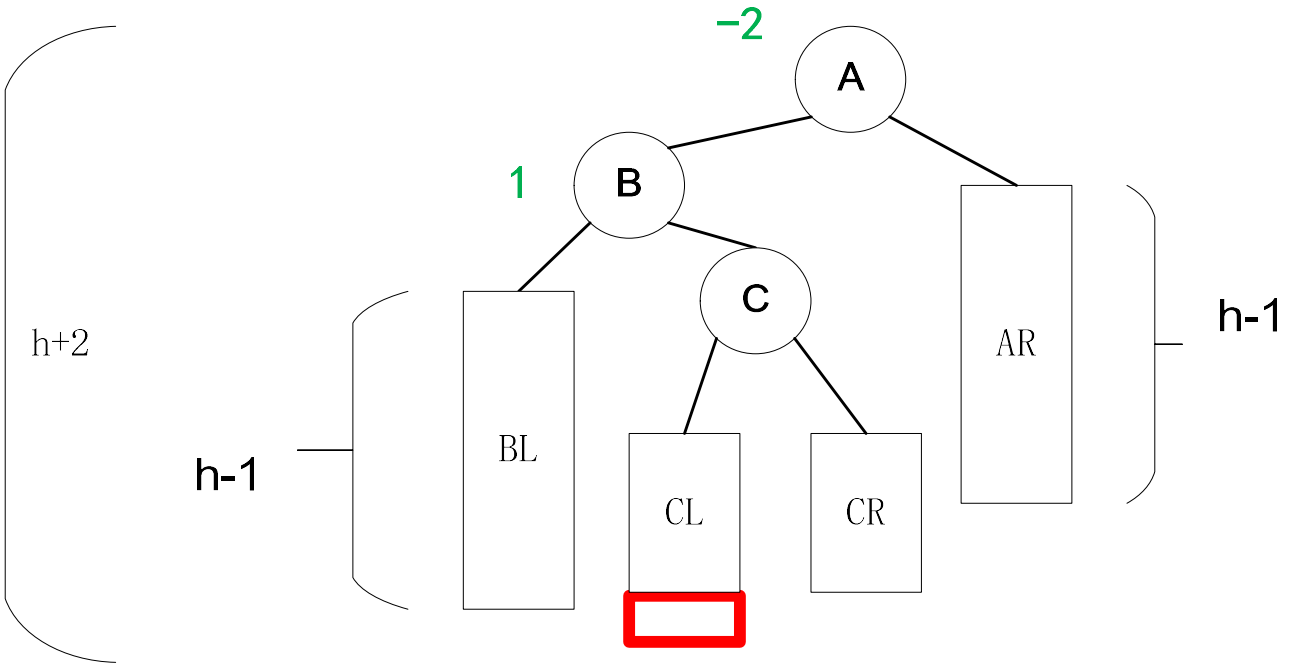


jyang
@cs.ecnu.edu.cn

4 平衡树

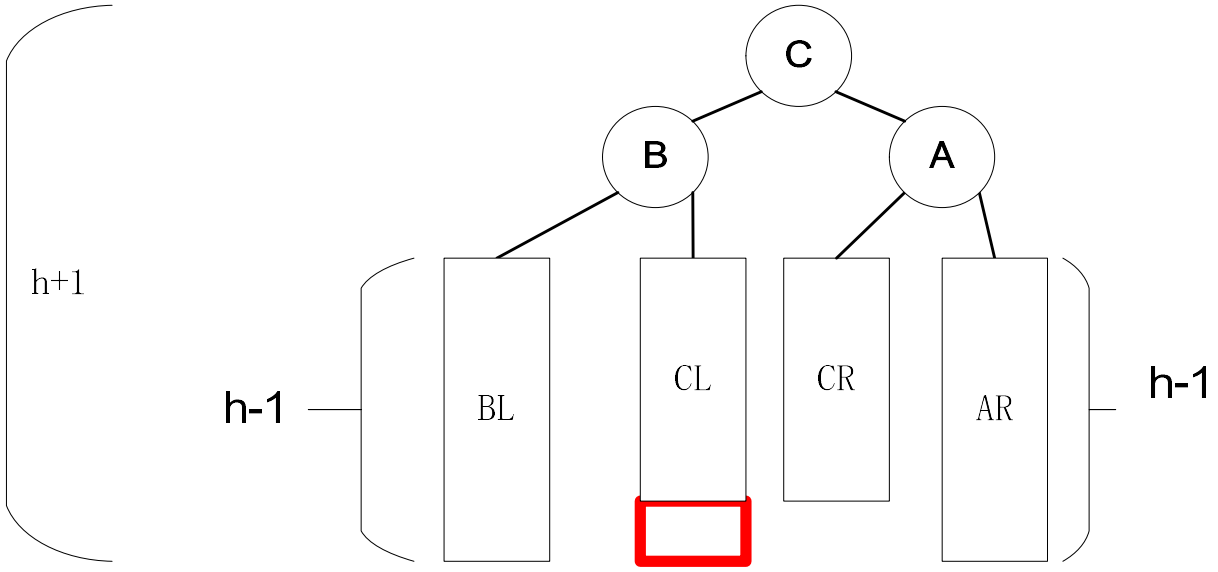
平衡树——

LR型：在结点A左子树的右子树上插入新结点失去平衡



平衡树——

LR型：在结点A左子树的右子树上插入新结点失去平衡

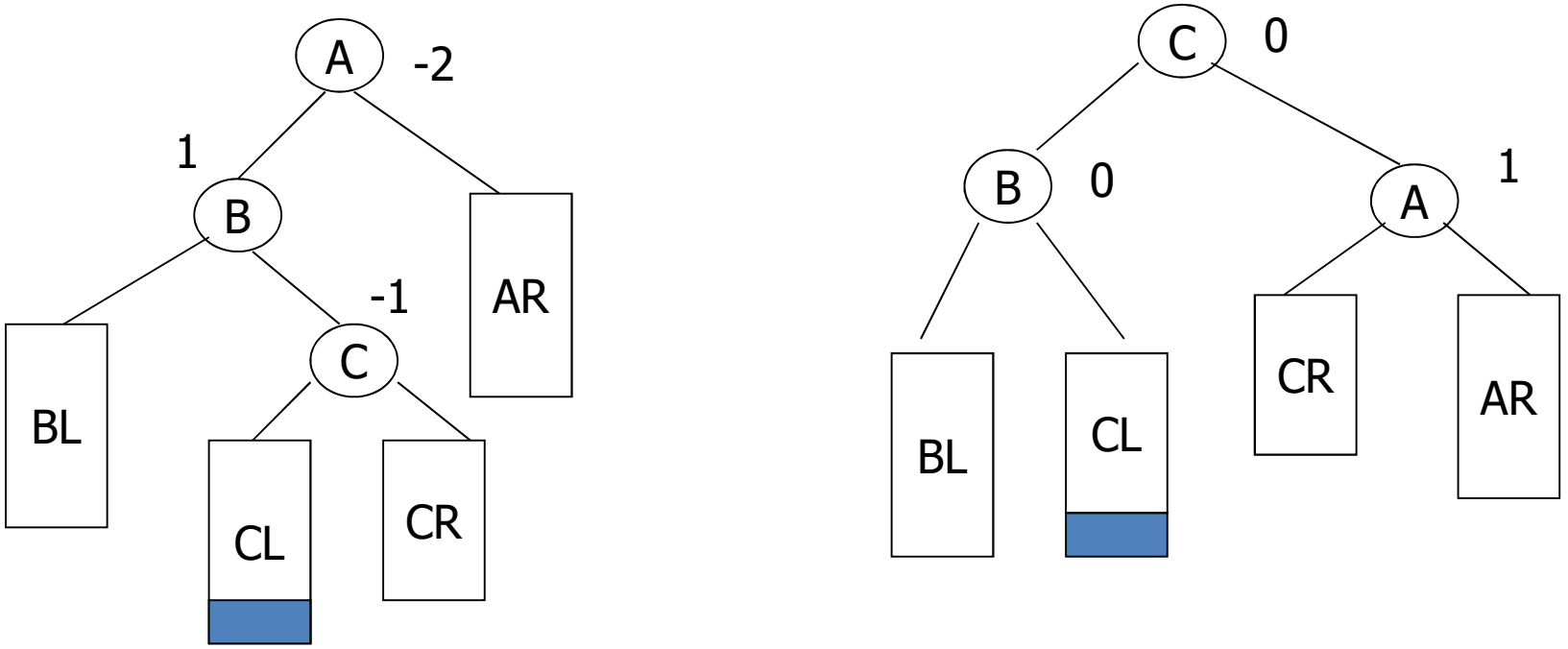


jyang
@cs.ecnu.edu.cn

4 平衡树

平衡树——

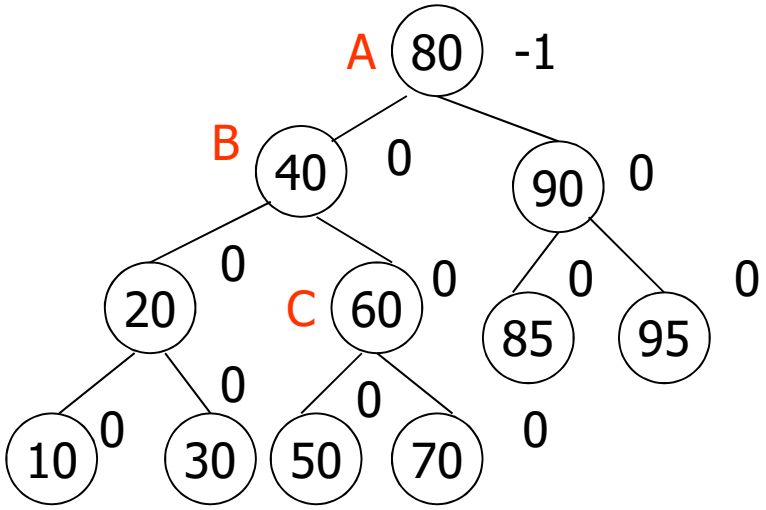
LR型：在结点A左子树的右子树上插入新结点失去平衡



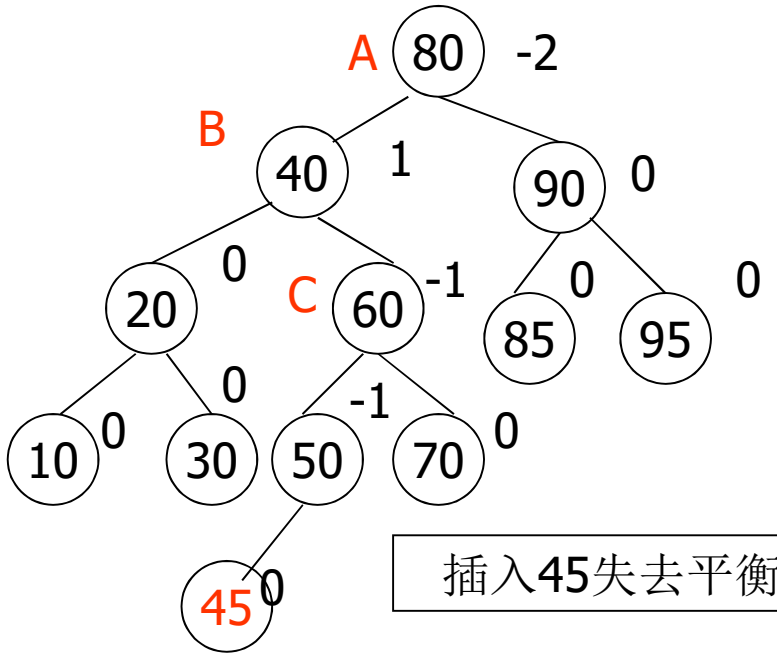
改组方案:首先将B改为C的左子树，而C原来的左子树CL改为B的右子树，然后将A改为C的右子树,C原来的右子树CR改为A的左子树。(相当于对B先左旋转,再对A执行右旋转)

jyang
@cs.ecnu.edu.cn

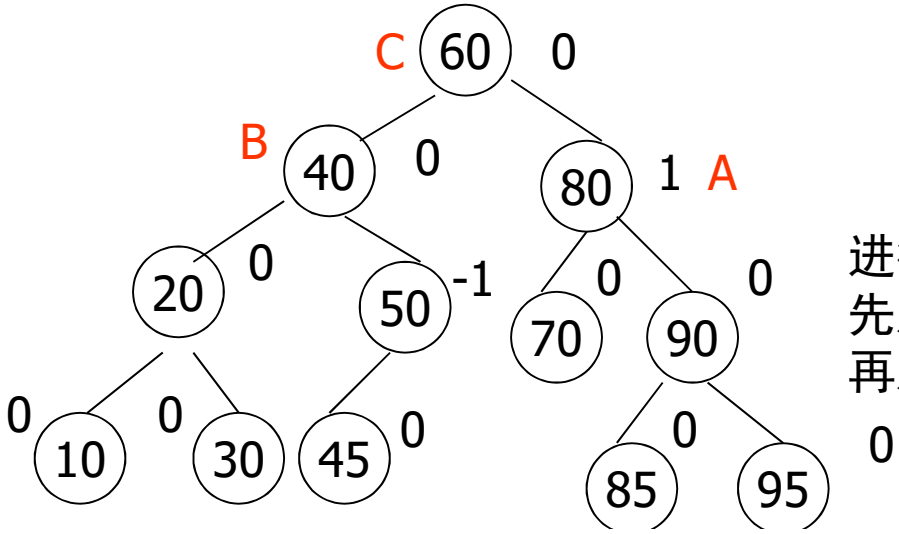
4 平衡树



一棵平衡二叉树



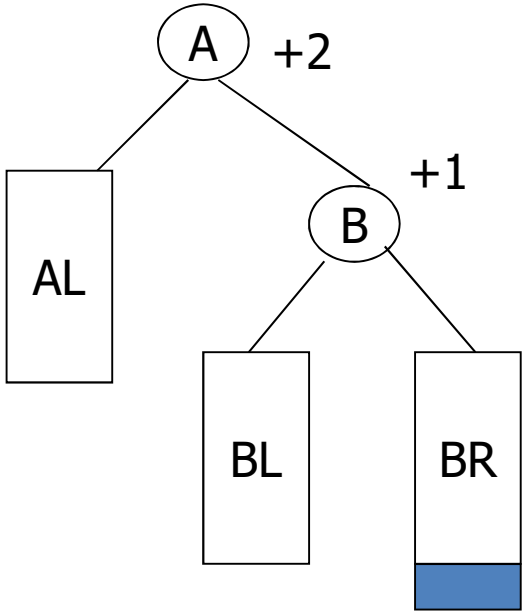
插入45失去平衡

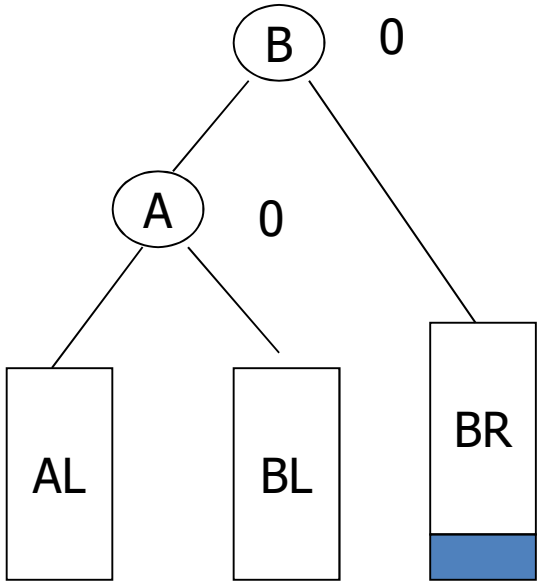


进行左改组LR：
先对B左旋转，
再对A右旋转

平衡树——

RR型：在结点A右子树的右子树上插入新结点失去平衡

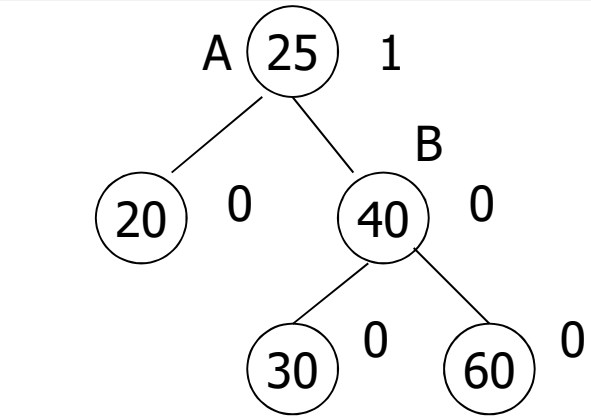




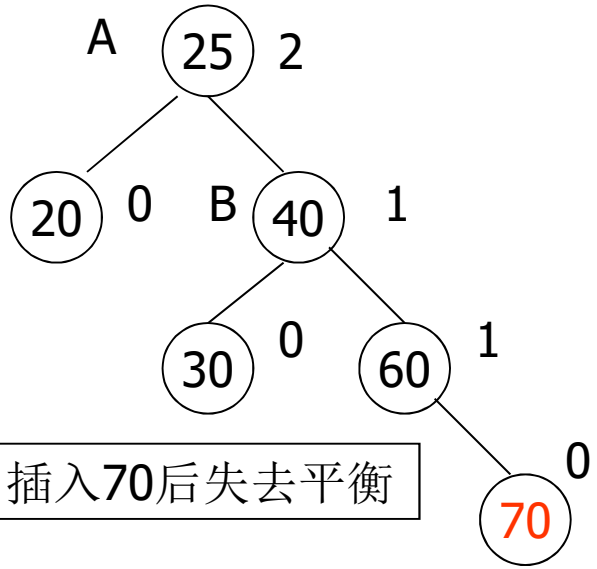
改组方案：将A改为B左子树，B原来的右子树BL改为A的右子树，实现单向左旋转操作。

平衡树——

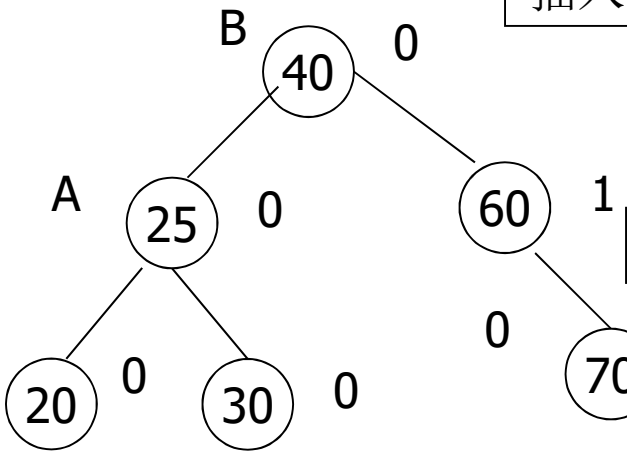
RR型：在结点A右子树的右子树上插入新结点失去平衡



一棵平衡二叉树



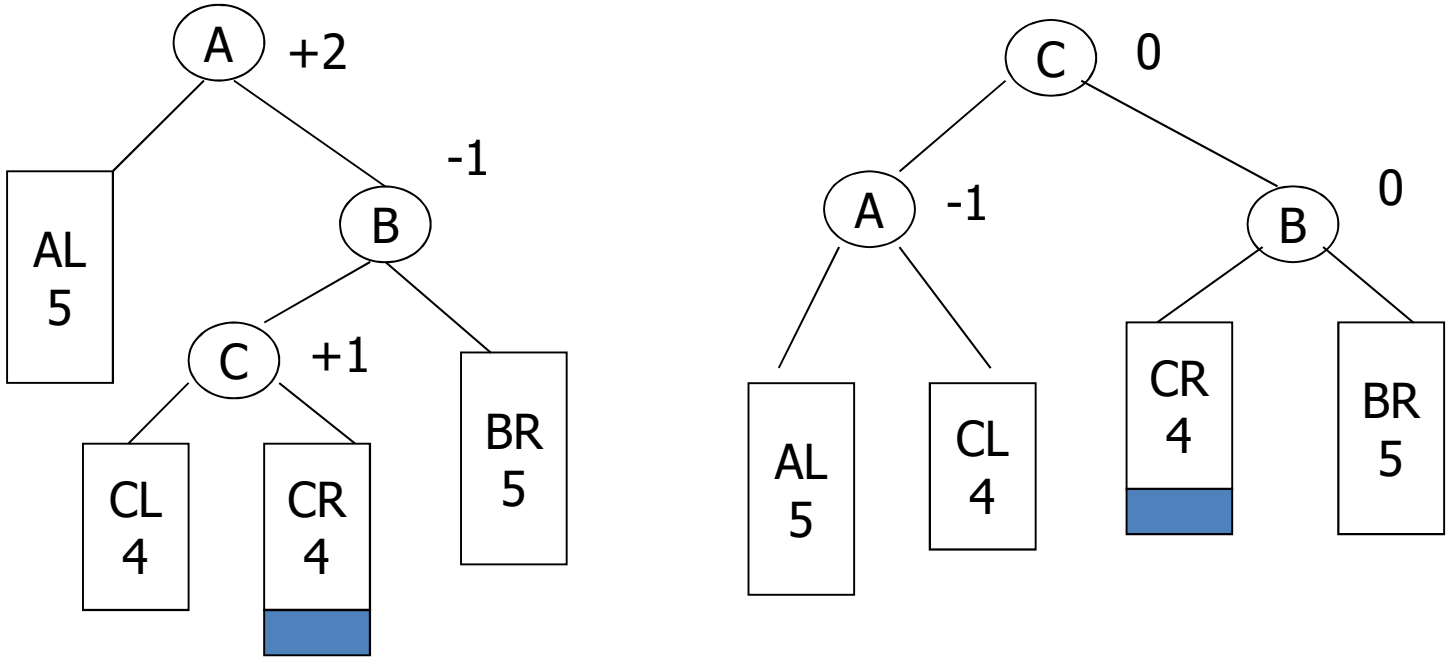
插入70后失去平衡



进行左改组：RR，左旋转

平衡树——

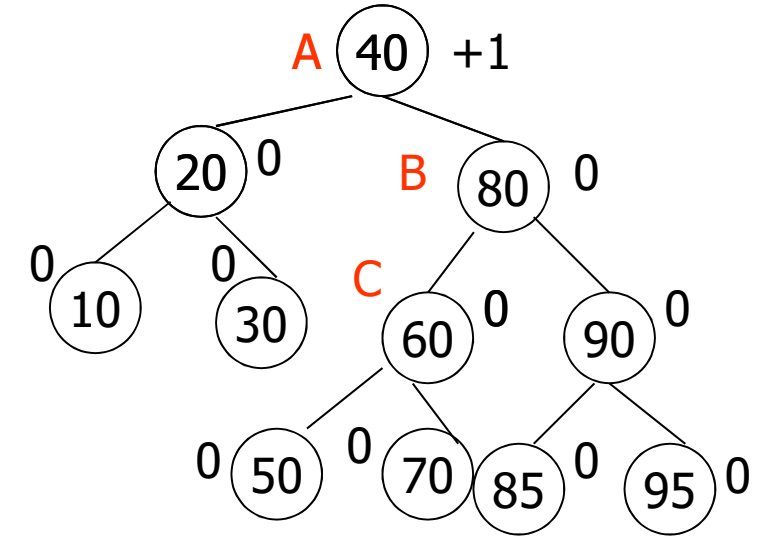
RL型：在结点A右子树的左子树上插在新结点失去平衡



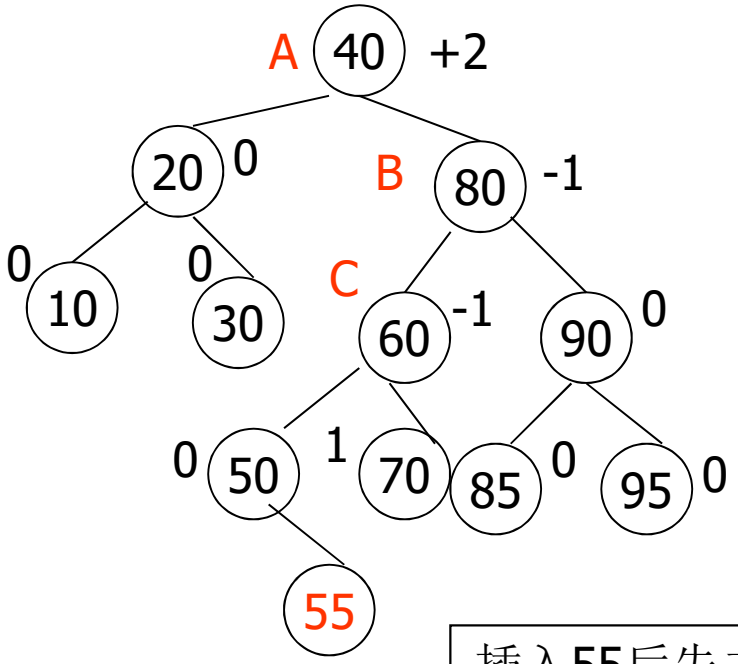
改组方案:首先将B改为C的右子树，而C原来的右子树CR改为B的左子树，然后将A改为C的左子树,C原来的左子树CL改为A的右子树。(相当于对B先右旋转,再对A执行左旋转)

平衡树——

RL型：在结点A右子树的左子树上插在新结点失去平衡



一棵平衡二叉树



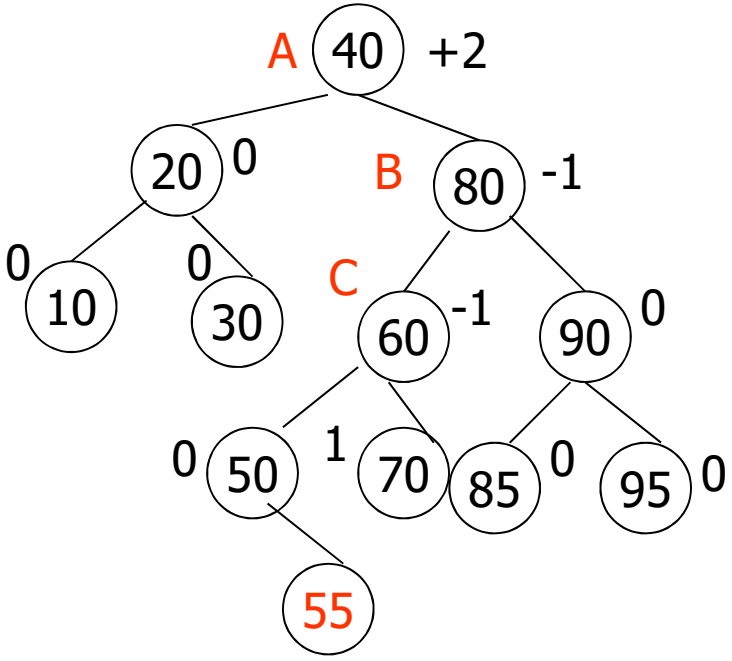
插入55后失去平衡

jyang
@cs.ecnu.edu.cn

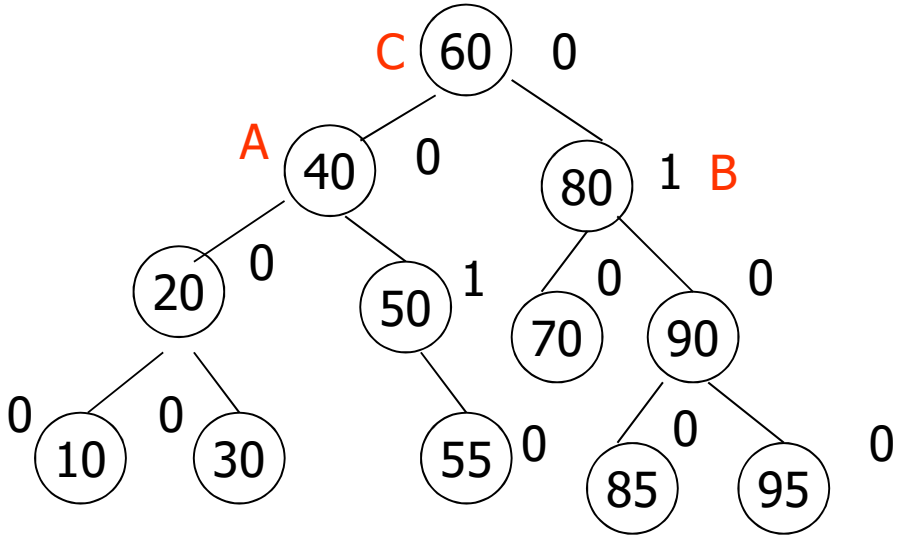
4 平衡树

平衡树——

RL型：在结点A右子树的左子树上插在新结点失去平衡



插入55后失去平衡



进行右改组RL ： 先对B右旋转，再对A左旋转

平衡树的插入

例：图6. 4. 4

思考：删除结点？

删除节点操作比插入复杂。插入只需要一次平衡操作即可恢复树的平衡性。而删除节点在删除后，可能需要多次操作才能恢复树的平衡。
要先找到最底部不平衡的子树，将其恢复平衡。再根据情况判断需不需要继续沿着子树向上恢复平衡。
为了找到最底部不平衡的子树，将删除操作迁移到最底部的子树上进行。

5 最佳查找树

在具有n个结点的查找树中，假设被查找的结点k都在查找树中。用二叉查找法进行查找时：

$$\text{AVG（二叉查找法）} = \sum_{\text{树中的}k} p(k)(1+\lambda k)$$

如果考虑所有结点的相对使用概率相等，则有

$$AVG\text{（二叉查找法）} = \frac{1}{n}\sum_{k\ of\ T}(1+\lambda k)$$

考虑更一般的情况：被查找的结点可能在树中或不在树中，并且使用概率不相同。

5 最佳查找树

树中的空指针：每个空指针上挂上外部附加结点结点。具有n个结点的二叉树，经过扩充后，产生（n+1）个外部结点。

在查找时，如果查找的是不在查找树中的结点，则会终止于外部结点，故外部结点也称为失败结点。

外部路径长度：由根结点到二叉树所有外部结点的路径长度的总和为二叉树的外部路径长度E_n。
内部路径长度：由根结点到二叉树的所有内部结点的路径长度的总和为二叉树的内部路径长度I_n。

一棵具有n个内部结点的二叉树的内部路径长度I_n和外部路径长度E_n之间的关系：$E_n=I_n+2n$

5 最佳查找树

对于具有n个结点的所有二叉树，具有最大（或最小）内部路径长度的二叉树T，也一定是具有最大（或最小）外部路径长度的二叉树；反之亦然。

为了得到具有最小内部路径长度的二叉树，必须使内部结点尽量靠近根结点。当二叉树退化成线性链表时，其内部路径长度最大，其长度为：$I_n=n(n-1)/2$

对于由结点a_1，……，a_n构成的所有可能的查找树中，使

$$\sum_{i=1}^{n} p_i(1+\lambda a_i)+\sum_{i=0}^{n} q_i(\lambda b_i)$$

取最小值的查找树就是a_1，……，a_n的最佳查找树。

5 最佳查找树

当成功和不成功的查找都具有相同的查找概率时，

AVG=$\frac{1}{2n+1}(2I_n+3n)$

用平分法构造出来的丰满查找树就能使I_n达到最小。

AVG=$\frac{1}{2n+1}(2I_n+3n)=O(log_2 n)$

查找概率不同时，如何构造最佳二叉查找树（查找代价最小）？

性质：一棵最佳查找树的所有子树都是最佳查找树。
构造方法：P168~170

5 最佳查找树

构造方法：P168~170

用权（和树的形态无关）代替使用概率，以方便计算。

$w_{ii}=q_i$

$w_{ij}=q_i+\sum_{k=i+1}^{j}(p_k+q_k)$ 表示T_{ij}的权。（w_{ij}和树的形态无关）

$$w_{ij}=w_{ij-1}+p_j+q_j$$

用C_{ij} 表示查找树T_{ij}的代价。定义：$C_{ii}=0$.

最佳查找树的性质：一棵最佳查找树的所有子树都是最佳查找树。

$$C_{ij}=p_k+cost(L)+w(L)+cost(R)+w(R)$$

$$C_{ij}=w_{ij}+C_{ik-1}+C_{kj}$$

$$C_{ik-1}+C_{kj}=\min_{i<t\le j}\{C_{it-1}+C_{tj}\}$$

jyang
@cs.ecnu.edu.cn

5 最佳查找树

构造方法：P168~170

最佳查找树的性质：一棵最佳查找树的所有子树都是最佳查找树。

例　(a1, a2, a3, a4)=(a, b, c, d)
概率：……

最佳查找树的可能：以 a1为根，a2为根，a3为根，a4为根。

jyang
@cs.ecnu.edu.cn

6 Huffman算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。
“A Method for the Construction of Minimum-Redundancy Codes”

例：Huffman编码——一种压缩算法

如果需要对由字母A B C D E组成的序列进行二进制编码。已知这些字母的出现次数不相同，分别为 10，5，20，10，18。

编码方案：

A:000　　B:001　　C:010　　D:011　　E:100

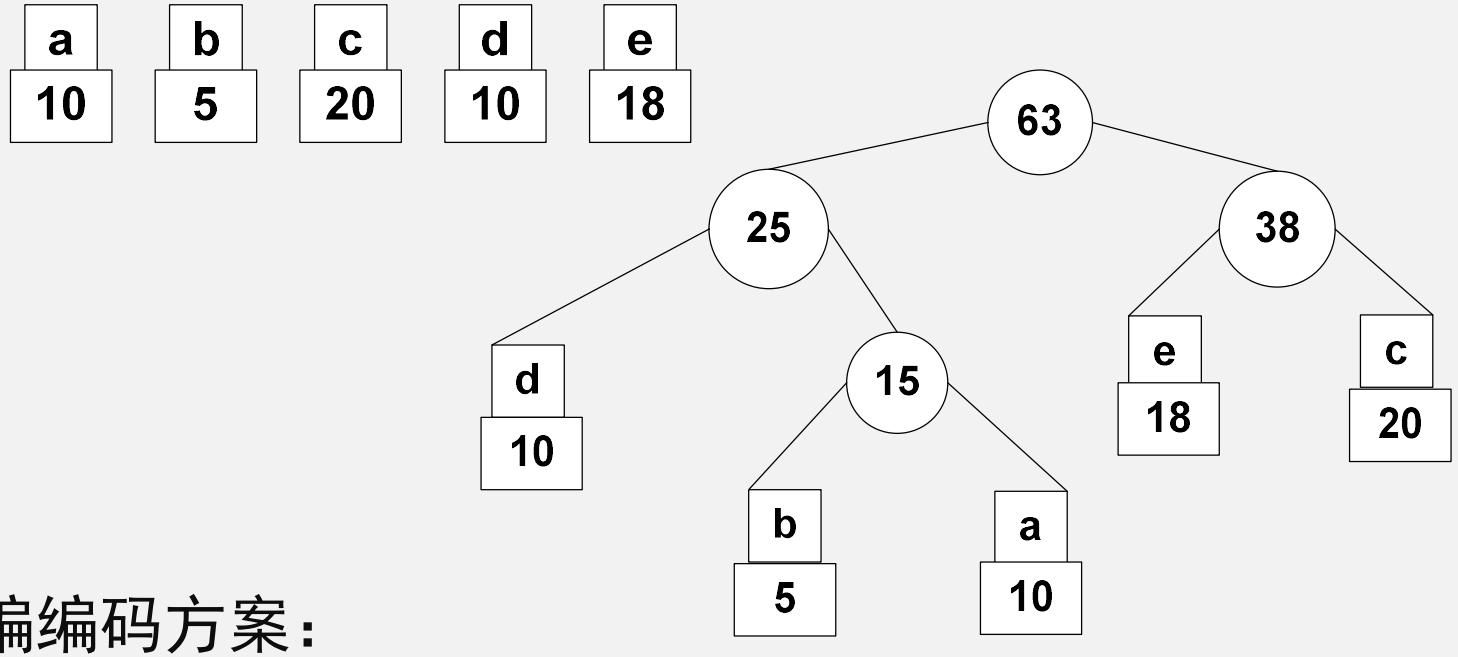
编码完以后的长度：63×3。（定长：不需要额外的分隔符即可唯一译码，有错误发现能力）

是否还可以有更“短”的编码方法（前提：不需要额外的分隔符可以唯一解码）？——Huffman编码

jyang
@cs.ecnu.edu.cn

6 Huffman算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。



Huffman编编码方案：

A:011　　B:010　　C:11　　D:00　　E:10

观察可得：出现次数多的结点，编码不会比出现次数少的结点长。

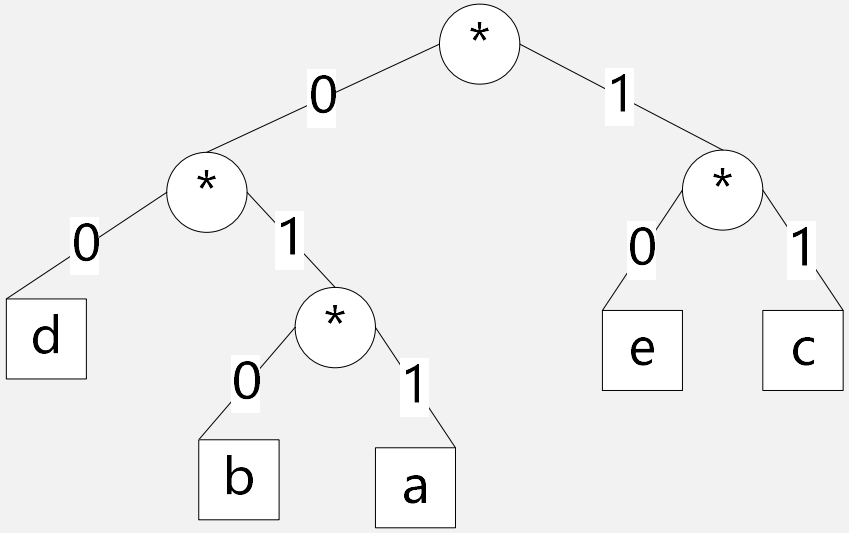
“字符”只出现在叶子，所以，任意编码，不会是其他编码的前缀

jyang
@cs.ecnu.edu.cn

6 Huffman算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。

也可以把这棵树称为译码树
每种合格的编码都能用这棵译码树来进行译码，
并且译码的结果是唯一的。



例如，序列0100110010的相应译码
b a d e

注：并不是所有的二进制串都能得到译码，
例：不合格的序列001110111 就不能得到译码

编码效率？

6 Huffman算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。

已知每个字符的概率，为每个字符单独分配编码，为了达到编码效率最高，

$$\bar{\mathrm{L}} = \sum_{\mathrm{i}=0}^{n-1} p_i \lambda k_i$$

Huffman编码，已经达到这类编码的最优。

jyang
@cs.ecnu.edu.cn

6 Huffman算法
Hu-Tucher算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。

设给定一个具有n个结点的序列F=$k_0, k_1, \cdots\cdots, k_{n-1}$，它们的权都是正整数，且分别是w（ k_0）w（k_1）……，w（k_{n-1}）。我们要构造一棵以$k_0, k_1, \cdots\cdots, k_{n-1}$作为叶子结点的二叉树T，使得：

$$\sum_{i=0}^{n-1} w(k_i)\,\lambda\,k_i$$

达到最小。

其中，λk是从根结点到达叶子节点的树枝长度（外部路径长度）。我们称T 是一棵具有最小加权外部路径长度的二叉树(n个结点看做外部结点)。

jyang
@cs.ecnu.edu.cn

6 Huffman算法
Hu-Tucher算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。

初始时，A=F，m=n，对t=1，2，……，n-1 执行：

设A= a_0，a_1，……，a_{m-1}，A中的结点都是已形成的子树的根。

如果a_i和a_j分别是A中权最小的两个结点，那么用具有权$w(b_t)=w(a_i)+w(a_j)$的新结点b_t和a_i，a_j形成新的子树(其中b_t是新子树的根结点)。

然后，从A中除去a_i和a_j，并把b_t作为A的最后一个结点，m减去1。

Huffman算法：构造的二叉树称为为Huffman树.

jyang
@cs.ecnu.edu.cn

6 Huffman算法 Hu-Tucher算法

Huffman算法：寻找具有最小加权外部路径长度的二叉树的方法。

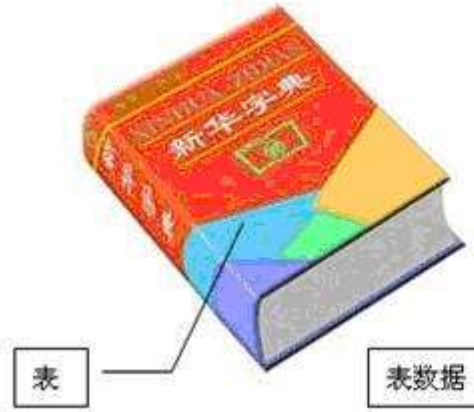
P172~173

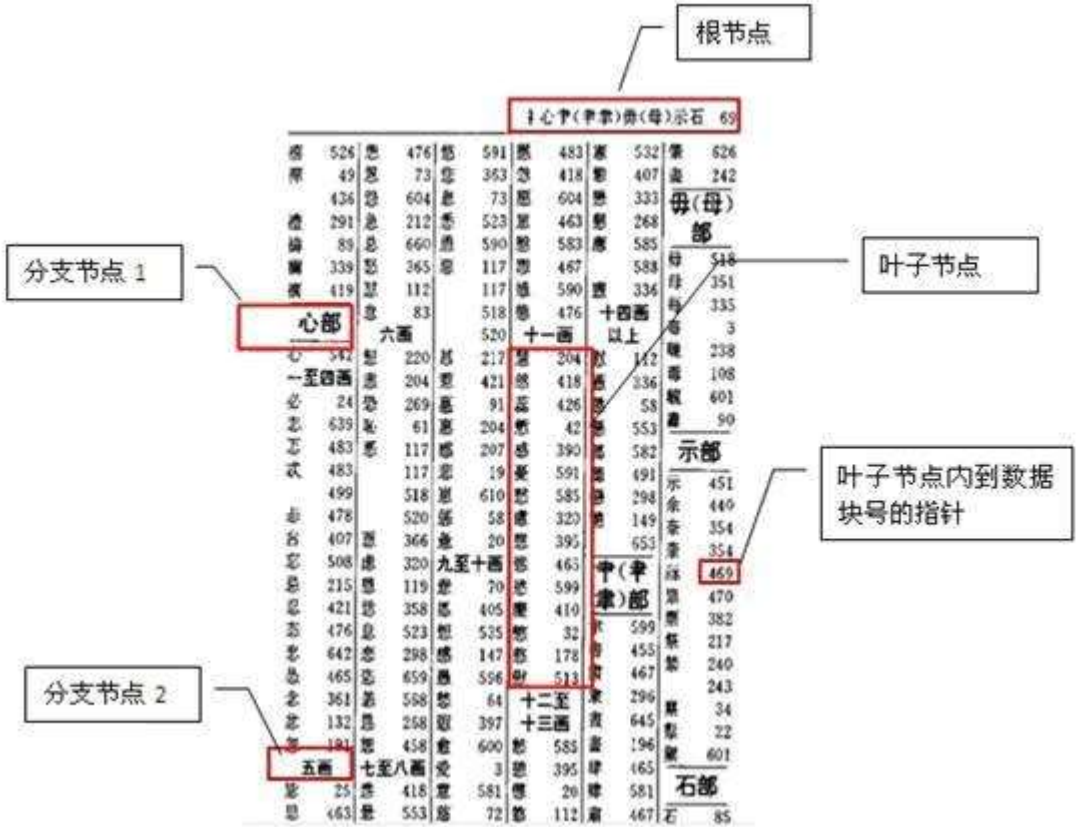
Huffman树：
1. Huffman树非叶子结点的次树（度）是多少？

2. 结点序列具有n个结点，构造完的Huffman树一共有多少个结点（包括叶子结点和非叶子结点）？

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用





7 B-树

B-树是一种平衡的多叉树。

一棵m阶B-树是具备下列性质的树：

（1）每个结点的子结点个数≤m；
（2）除根和叶子之外，每个结点的子结点个数≥$\lceil m/2 \rceil$；
（3）如果根结点不是叶子结点，则至少有两棵子树；
（4）所有叶子结点都出现在同一层上，而且不带有信息。
（5）具有k个子结点的非叶子结点含有k-1个键值。

叶子结点是终端结点，没有子结点，不带有信息。（可以看作实际上不在树中的外部结点）。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

jyang
@cs.ecnu.edu.cn

7 B-树

在m阶B-树中，每个结点具有如下信息：

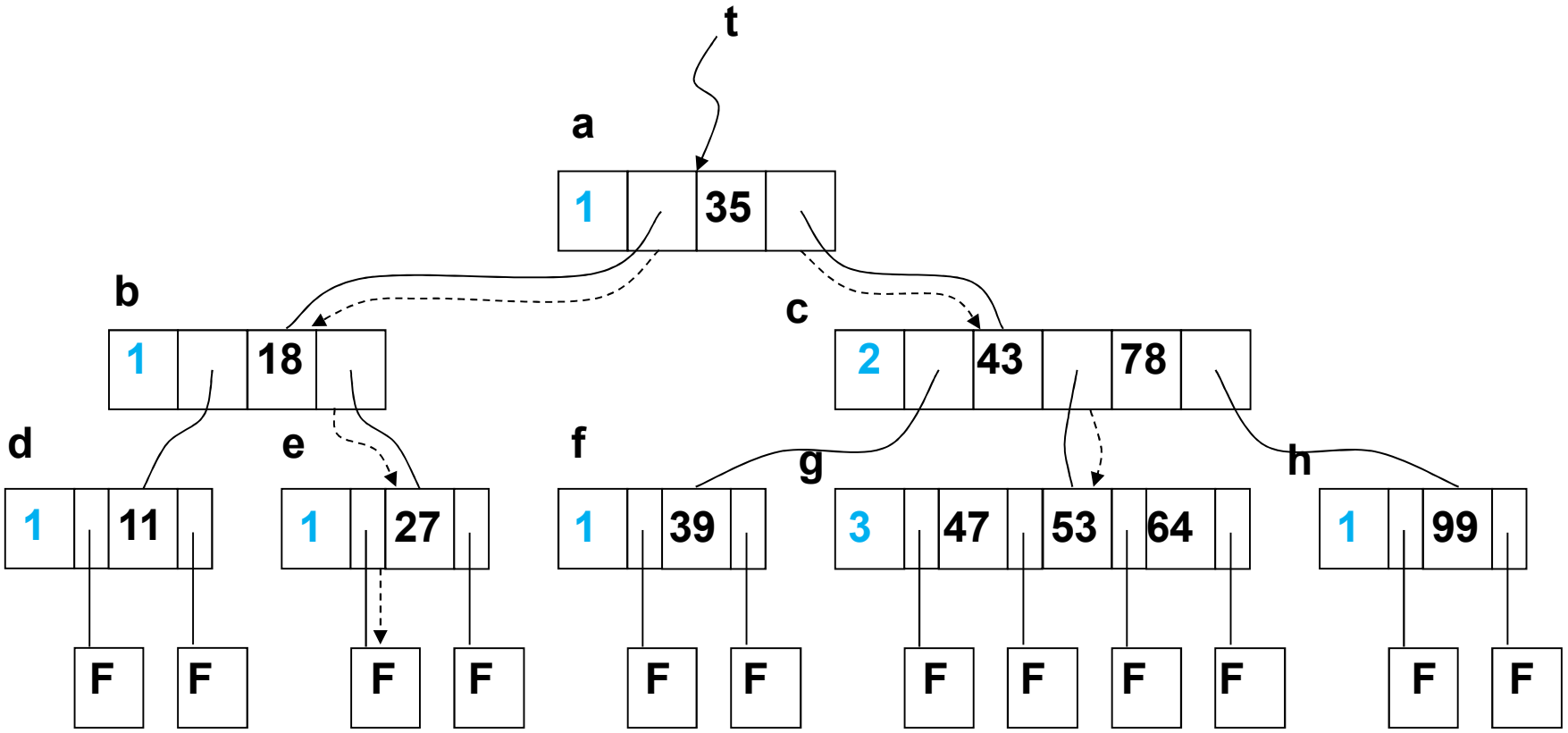
$(p_0, K_1, p_1, K_2, p_2, \ldots, K_j, p_j)$其中：

$j \leqslant m-1$；
K_i （$1 \leqslant i \leqslant j$）是键值，所有的键值都是唯一的。
Pi （$0 \leqslant i \leqslant j$）是指向该结点的子结点的指针。
结点中的键值是排好序的（约定升序或降序）。
如果按升序，有$k_1<k_2<\cdots<k_j$，那么p_0指向一棵键值都小于k_1的子树的根结点。P_i（$0<i<j$）指向一棵键值都在k_i和k_{i+1}之间的子树的根结点；P_j指向一棵键值都大于k_j的子树的根结点。

为便于查找、插入和删除，也可以在结点上附加一个信息，比如指明结点当前有多少个键值。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用



jyang
@cs.ecnu.edu.cn

7 B-树

在m阶B-树上进行查找、插入和删除的算法

（一）查找：从根结点出发，沿指针搜索结点 和 在结点内进行顺序（或二分）查找 两个过程交叉进行。

对于给定的一棵m阶B-树，在最坏情况下，查找一个键值最多要取几个结点？

取结点的次数，依赖于树的层数。如果叶子结点在第t层，非叶子结点的层次是（0～t-1），则需要取t个结点。

$O(\log_m n)$

jyang
@cs.ecnu.edu.cn

7 B-树

（一）查找

对于给定的一棵m阶B-树，具有N个关键字.

在最坏情况下，查找一个键值最多要取几个结点？

若m阶B-树中具有N个关键字，则叶子结点即查找不成功的结点个数？

N＋1。

层次	最少的结点个数
0	1
1	2
2	$2\lceil m/2 \rceil$
3	$2\lceil m/2 \rceil^2$
……	
t	$2\lceil m/2 \rceil^{t-1}$

$$N+1 \geq 2\left\lceil \frac{m}{2} \right\rceil^{t-1}$$

$$t \leq 1 + \log_{\lceil m/2 \rceil} \frac{N+1}{2}$$

（二）插入
先查找。
要插入的键值不在树中，可以通过查找得到被插入结点的位置。
被插入结点总是位于叶子层的上面一层。

插入时考虑如下限制条件：

（1）每个结点的子结点个数≤m；
（2）除根和叶子之外，每个结点的子结点个数≥$\lceil m/2 \rceil$；

达到m，分裂。
分裂以后，父节点增加键值和指针，也可能达到m，分裂。……

（二）插入

当我们把一个新键值插入到一棵m阶B-树，第t层是叶子，新键值插入在第（t-1）层。

如果被插入的结点在插入前键值个数少于（m-1），则插入即可。

如果被插入的结点在插入前键值个数为（m-1），则插入后键值共m个。那么我们把这m个键值分裂为两个结点，

$(k_1, k_2, \cdots\cdots, k_{\lceil m/2 \rceil - 1})$　　$(k_{\lceil m/2 \rceil + 1} \cdots\cdots, k_m)$

$k_{\lceil m/2 \rceil}$插入到父结点中，父结点指向被插入结点的指针p改成p, $k_{\lceil m/2 \rceil}$, p’ . p指向分裂后的左面的结点，p’ 指向右面。

如果一直分裂到根结点，则把根结点分裂成两个结点，中间的键值往上推，做为一个新的根，B-树长高一层。

（三）删除

找到所要删除的键值k_i的位置。

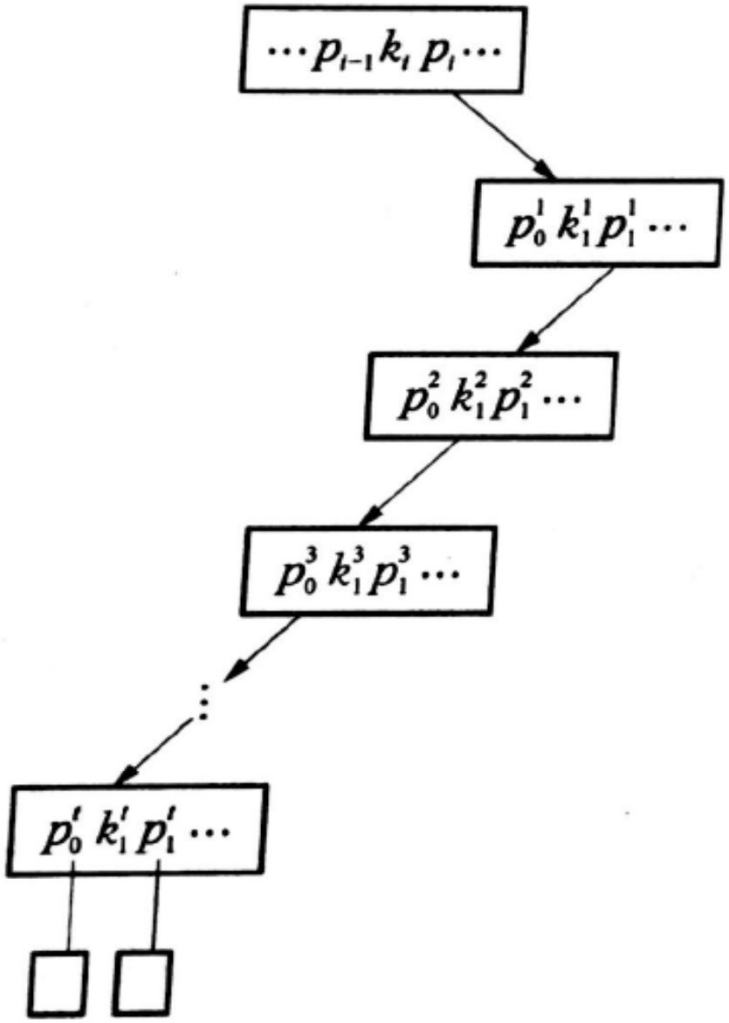
（1）若k_i在叶子层的上面一层，则删除ki和它右边的指针p_i
并检查结点中的键值个数，看是否需调整（ “借” ）。

（2）若ki不在最靠近叶子结点的一层，则用pi指向的子树中的最小键值k_{1t}替代k_i，再删除p_t和k_{it}。
即真正的删除是从叶子层的上面一层开始。

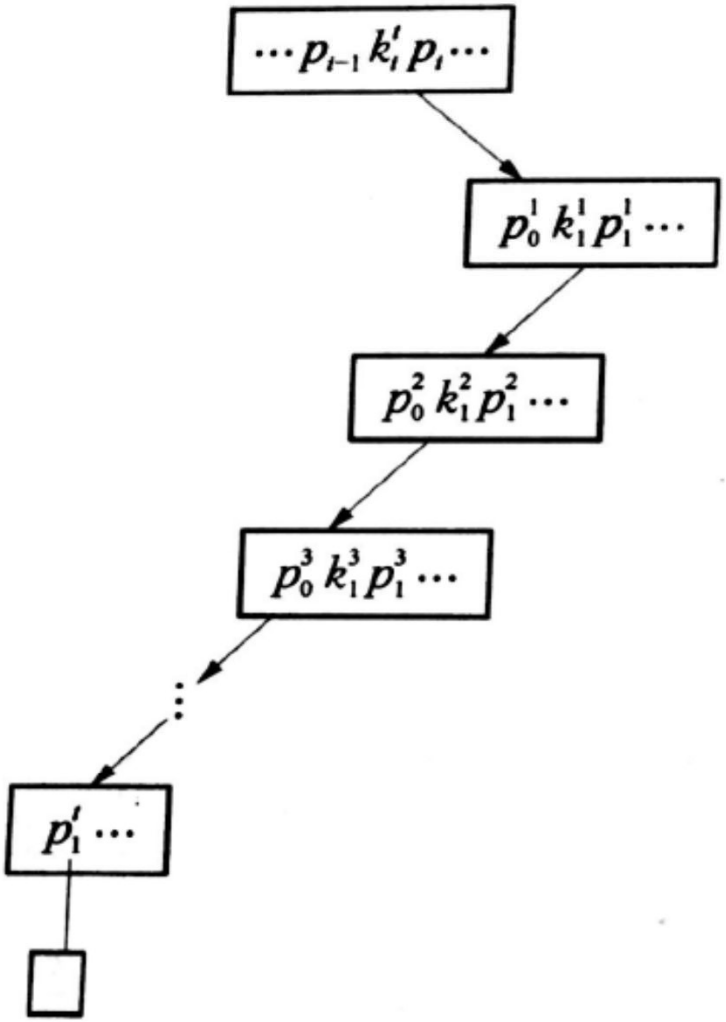
在执行了真正的删除之后，检查被删的结点键值的个数是否符合要求。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用



(a) 删k_i之前



(a) 删除k_i之后

（三）删除

若删除后小于 $\lceil m/2 \rceil - 1$（即删除前子结点的个数已经是最小值），要“借”一个键值。

与该结点相邻的右兄弟（或左兄弟）结点中的键值的个数大于 $\lceil m/2 \rceil - 1$，将其兄弟结点中的最小（或最大）的键值k_j上移至父结点中，而父结点中介于这两个子结点之间的键值下移至被删键值所在结点中。

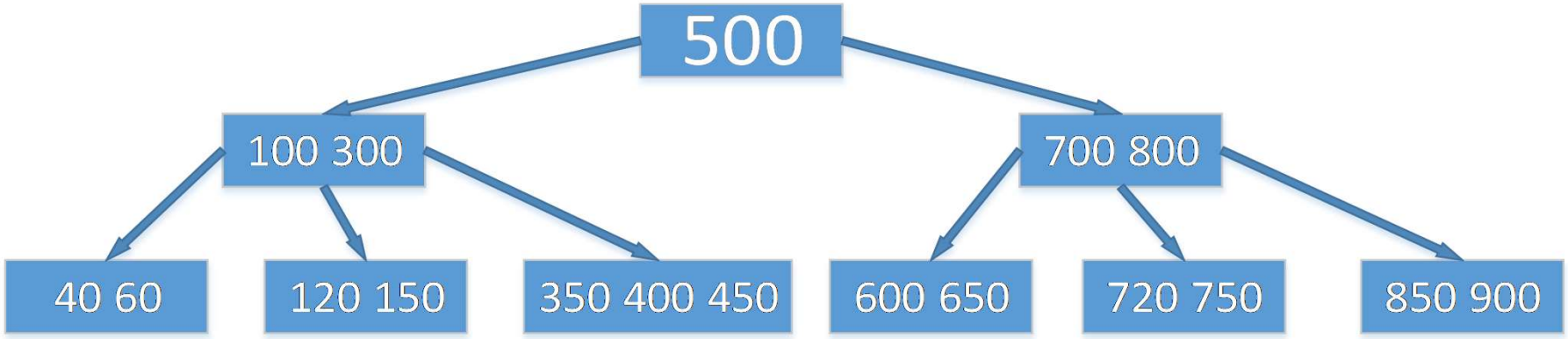
如果相邻的兄弟结点的键值个数都是$\lceil m/2 \rceil - 1$，那么就“借”不到键值。联结——把两个结点中的键值连同介于其中间的父结点的键值组成一个新结点。此时父结点就少了一个键值。

父结点有可能还需要进行“借”或“联结”。如一直影响到“根”，而此时根结点中只有一个 键值，那么联结的结果就会使整棵B-树减少一层。

（三）删除

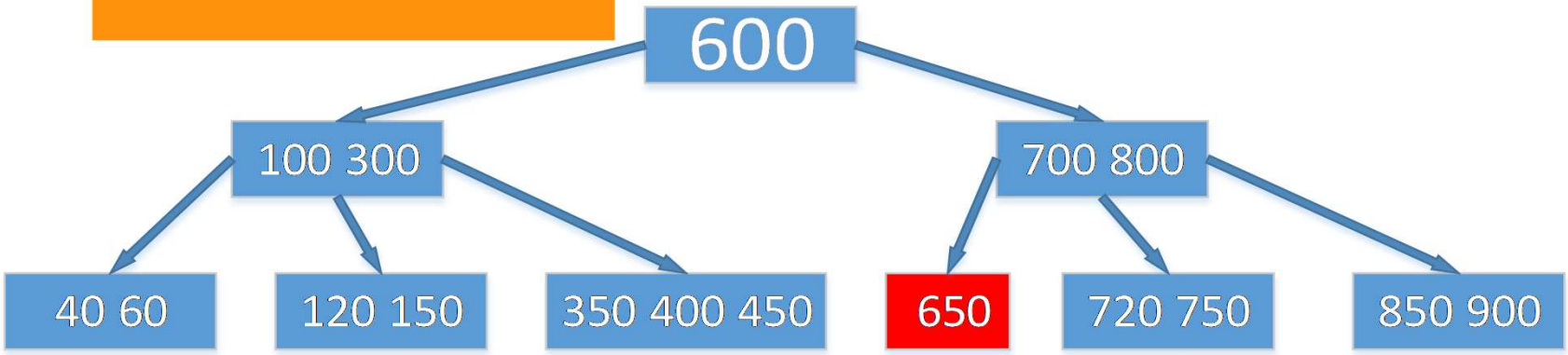
P181-182 插入和删除实例。

（三）删除

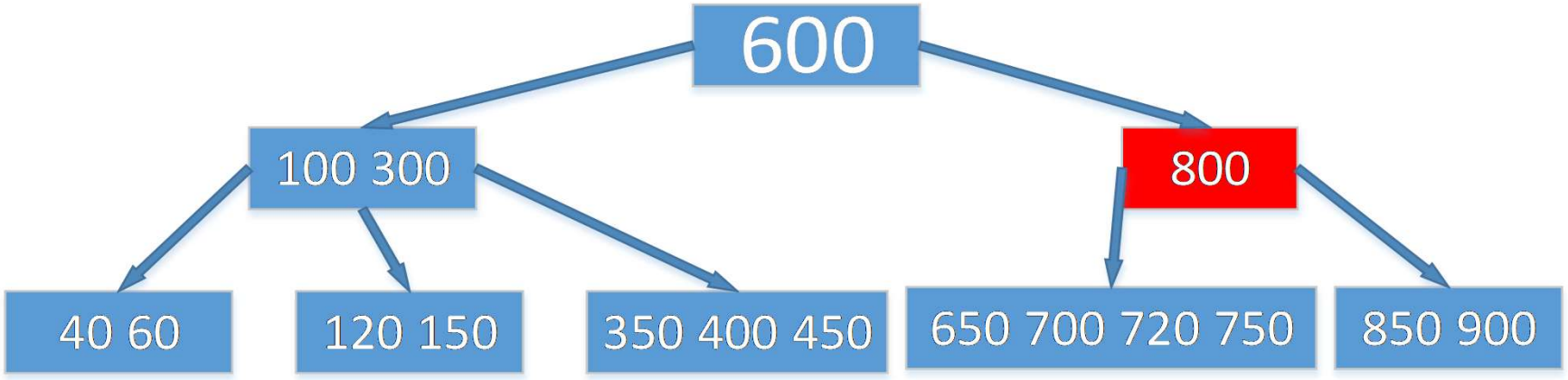


删除 键值 k 500:用k’（600)替代 k，删除k’

第六章
树的查找和树的应用

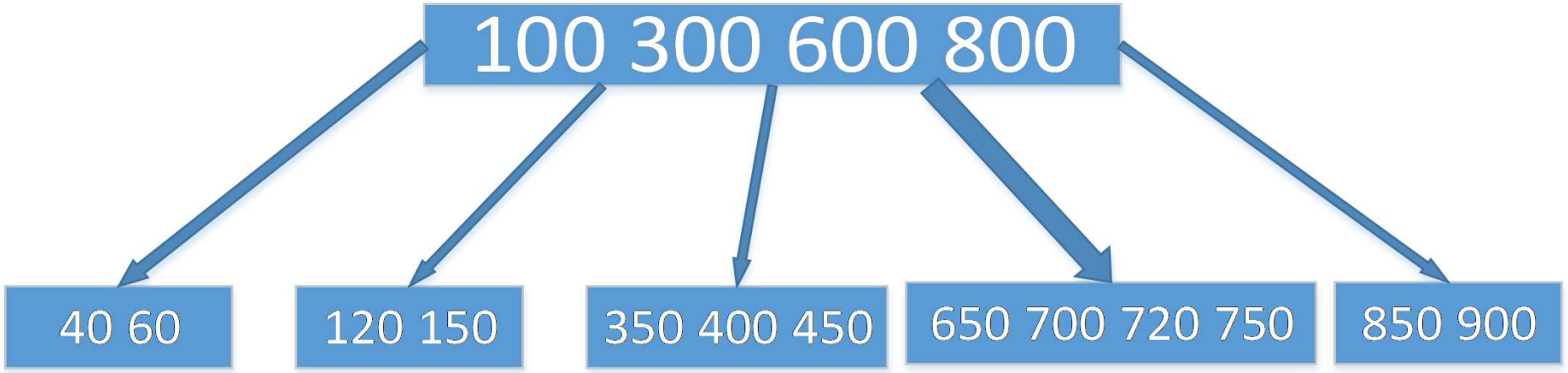


“借”不到键值。则 “联结”：把两个结点中的键值连同介于其中间的父结点的键值组成一个新节点。



jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用



jyang
@cs.ecnu.edu.cn

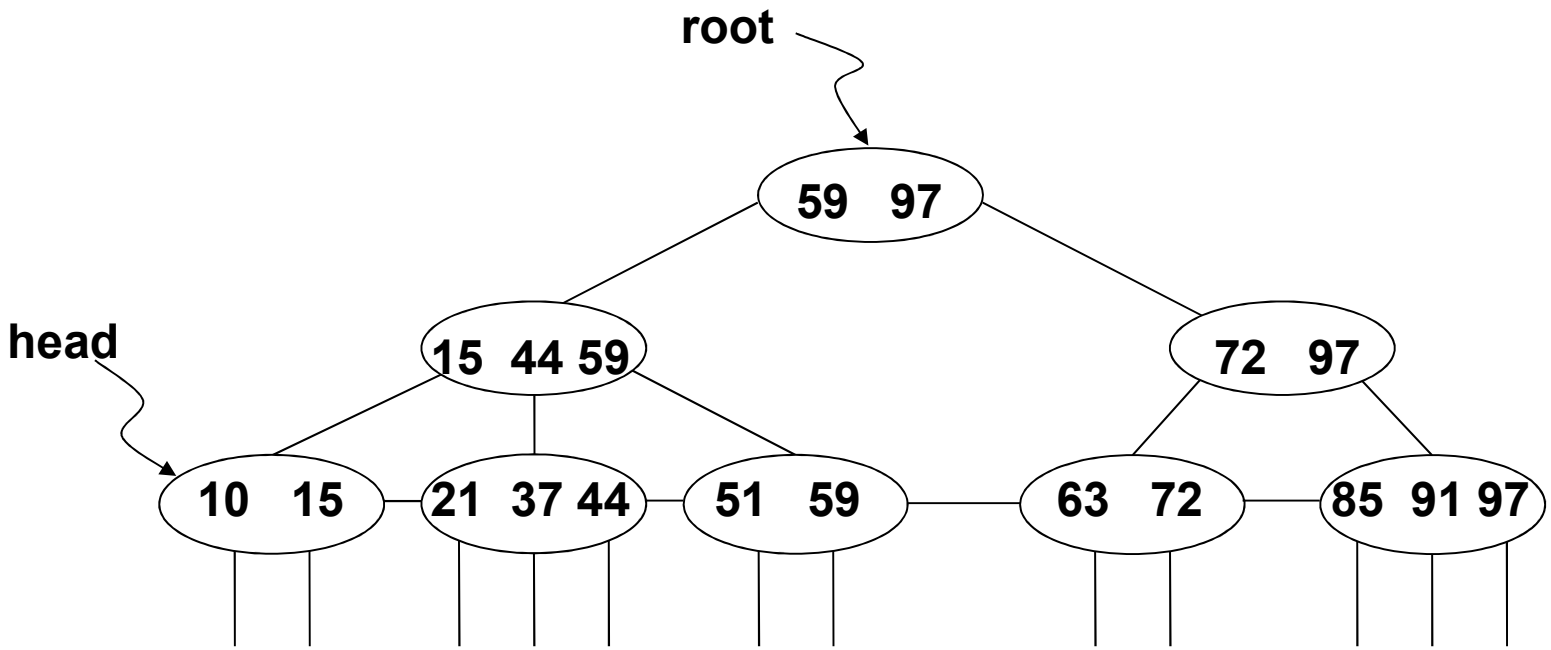
7 B-树

B+树——B-树的变形

（1）每个结点的子结点个数≤m；
（2）除根和叶子之外，每个结点的子结点个数≥$\lceil (m+1)/2 \rceil$
（3）根结点至少有2个子结点，最多有m个子结点。
（4）有k个子结点的结点必有k个键值。
（5）所有叶子结点包含全部键值及指向相应记录的指针，而且叶子结点按照键值从小到大顺序链接。
（6）所有非叶子结点仅包含其子树中的最大键值。

所有的非叶子结点可以看成是索引部分，结点中仅含其子树（根结点）中的最大（或最小）关键字。
通常在B^+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。

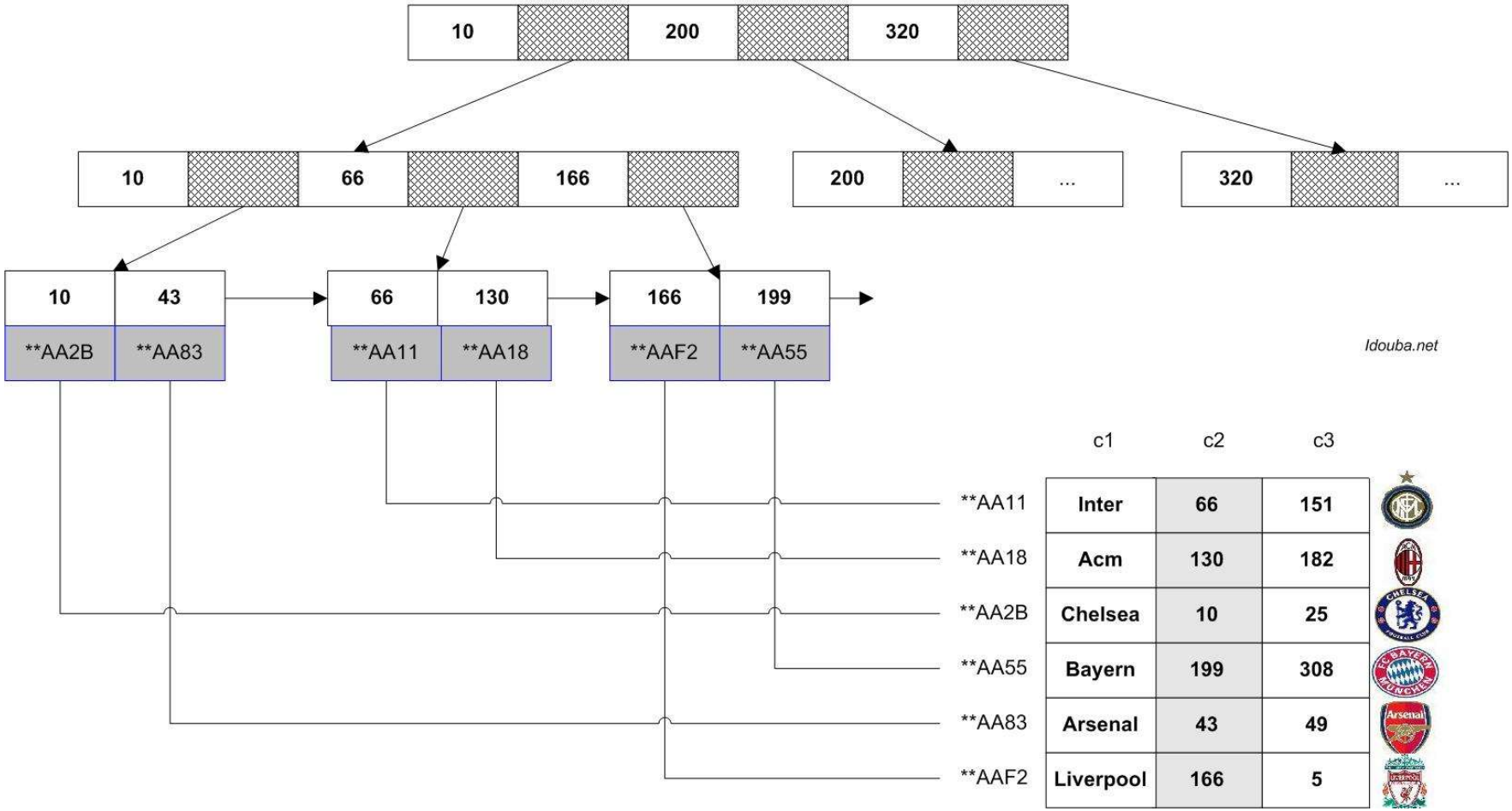
B+树



jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用

按照C2列上创建(非聚集)索引



B+树——查找，插入和删除

对B+树可以进行两种查找运算：

1.从最小关键字起顺序查找；

2.从根结点开始，进行随机查找

在查找时，如果非叶子结点上的键值等于给定值，并不终止，而是继续向下直到叶子结点。

因此，在B＋树中，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。每次查找的路径长度相同。
其余同B-树的查找类似。

jyang
@cs.ecnu.edu.cn

B+树——查找，插入和删除

插入：

B$^+$树的插入仅在叶子结点上进行，当结点中的关键字个数大于m时要分裂成两个结点，它们所含关键字的个数分别为

$$\left\lceil \frac{m+1}{2} \right\rceil \qquad \left\lfloor \frac{m+1}{2} \right\rfloor$$

并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。

其余同B-树的插入类似。

B+树——查找，插入和删除

删除：
B$^+$树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界键值”存在。
若因删除而使结点中的键值的个数少于$\lceil m/2 \rceil$时，需要进行联结，过程与B-树类似。

jyang
@cs.ecnu.edu.cn

7 Trie结构

例：一个信息检索的例子
《Introduction to Information Retrieval》

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

图 1-1　词项-文档关联矩阵，其中每行表示一个词，每列表示一个剧本。当词 t 在剧本 d 中存在时，矩阵元素 (t, d) 的值为 1，否则为 0[3]

例：一个信息检索的例子
《Introduction to Information Retrieval》

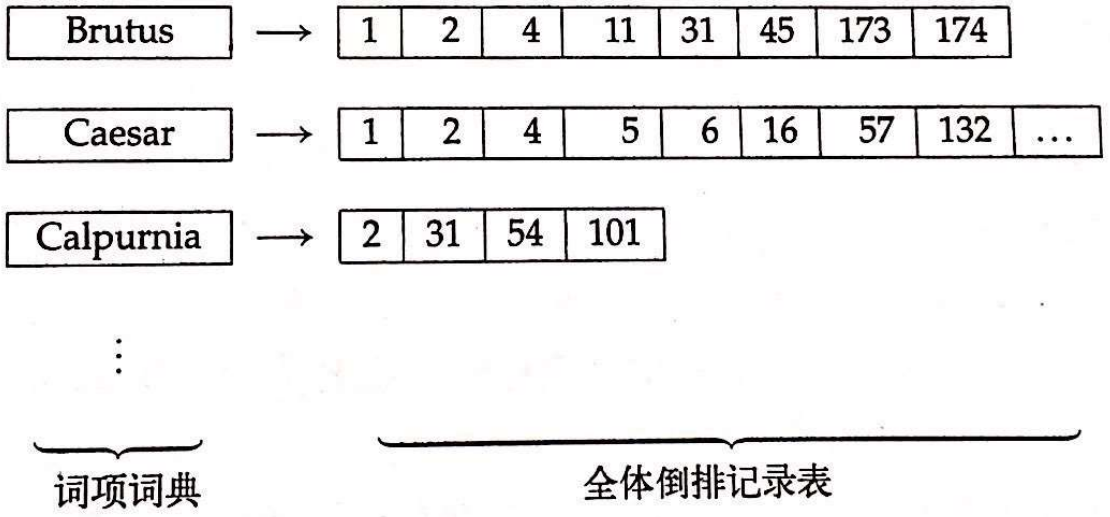


图 1-3 倒排索引的两个部分。词典部分往往放在内存中，而指针指向的每个倒排记录表则往往存放在磁盘上

7 Trie结构

Trie：由retrieve（检索）的中间四个字母组成，也称为“单词查找树”，它利用字符串的公共前缀来减少查询时间.

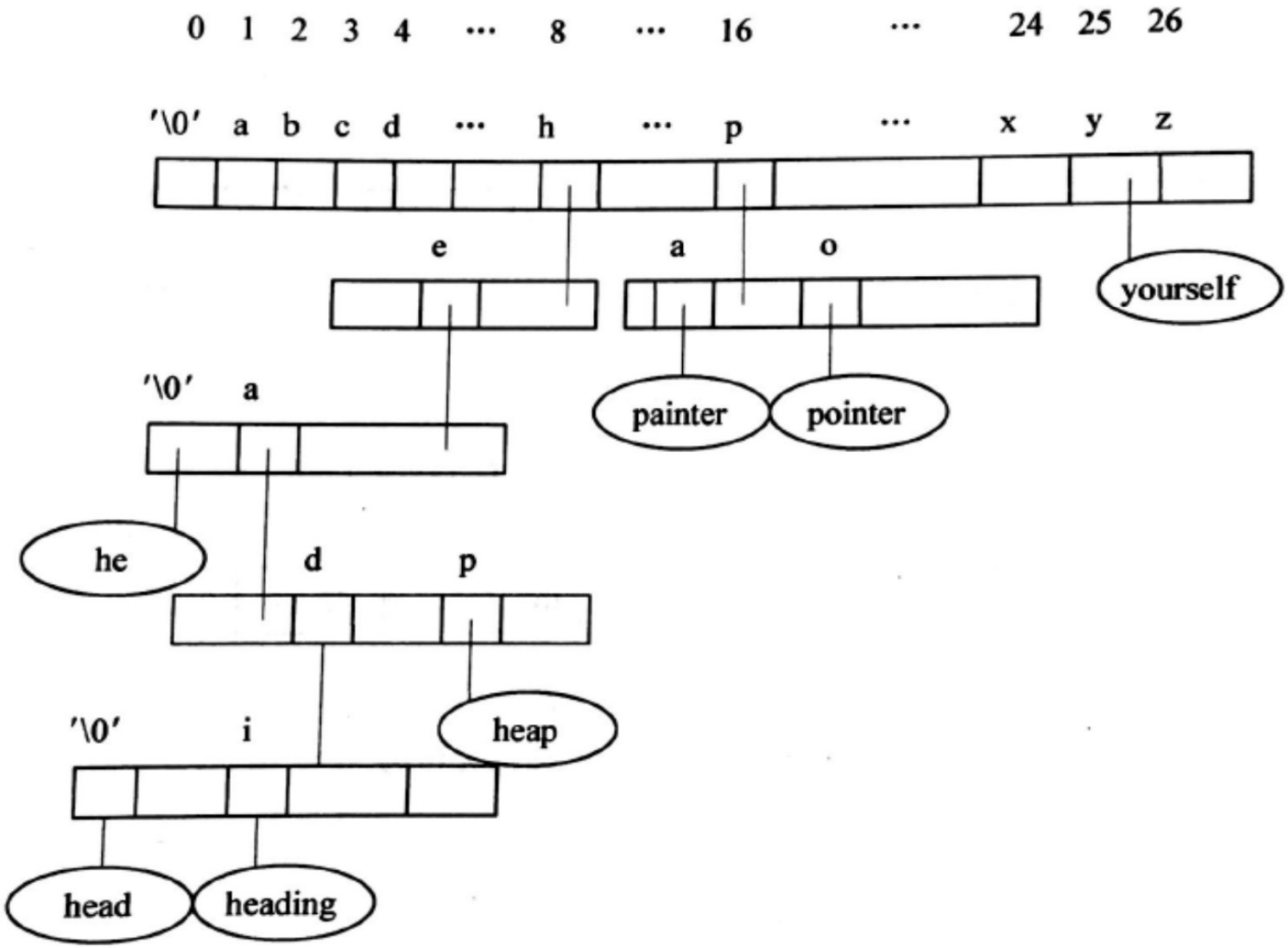
Trie树是一棵m次树（m>=2）.

例. 由 键值 head,heap,he,heading,point Er,painter,yourself 构成一棵Trie树。

树中有两种结点，一种是分支结点（可以看做内部结点），另一种是信息结点（可以把信息结点看做是叶子结点）。信息结点，除了存放键值，还存放包含键值有关的信息，例如该键值所在的记录的存放地址。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用



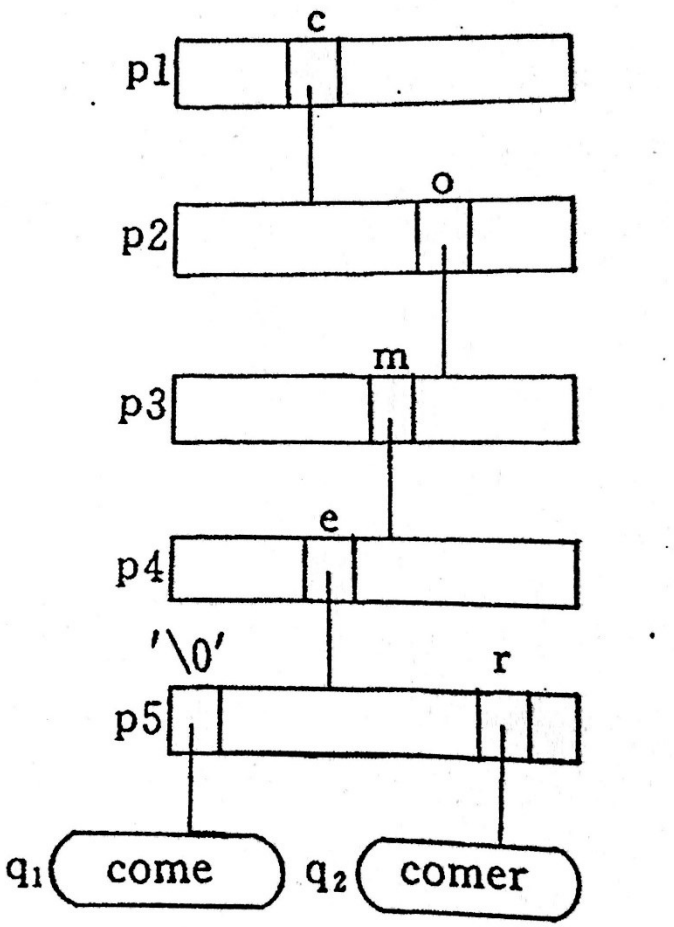
为描述方便，假设t是Trie树的根结点，那么t->link[i]指向一个Trie子树，这棵子树所包含的键值都是以英文字母表第i个字母开头。（内部结点通常有一个指针数组）
在Trie树的第j层，结点的分支情况由键值中序号为j的字符所确定。
如果一棵Trie子树只有一个键值，那么用一个信息结点（叶子结点）代替这棵Trie子树。

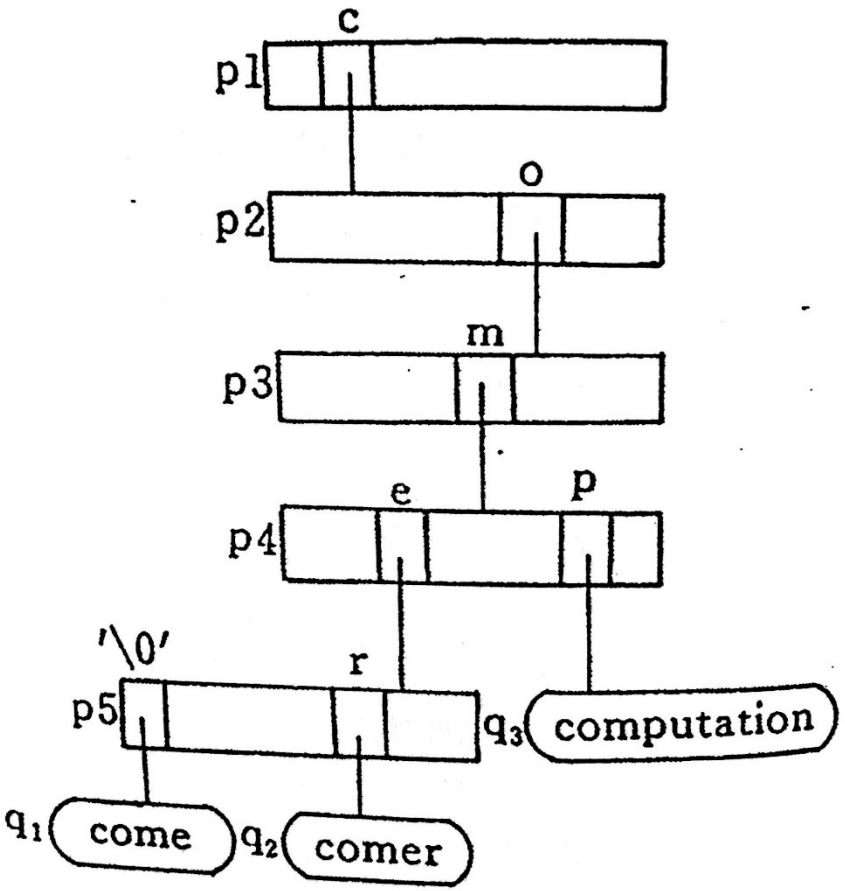
查找

在一棵Trie树中查找键值x，必须把x分解出一个一个的字符，按照这些字符所确定的分支进行查找。

jyang
@cs.ecnu.edu.cn

第六章
树的查找和树的应用





插入

查找，不在树中则插入。两种情况：

对应的link为空，直接在相应的位置插入新的信息结点。

查找停止在信息节点上，但信息结点的值不是要插入的键值，则需要构造新的子树。

例 186，图 6.8.3，图6.8.4

第六章
树的查找和

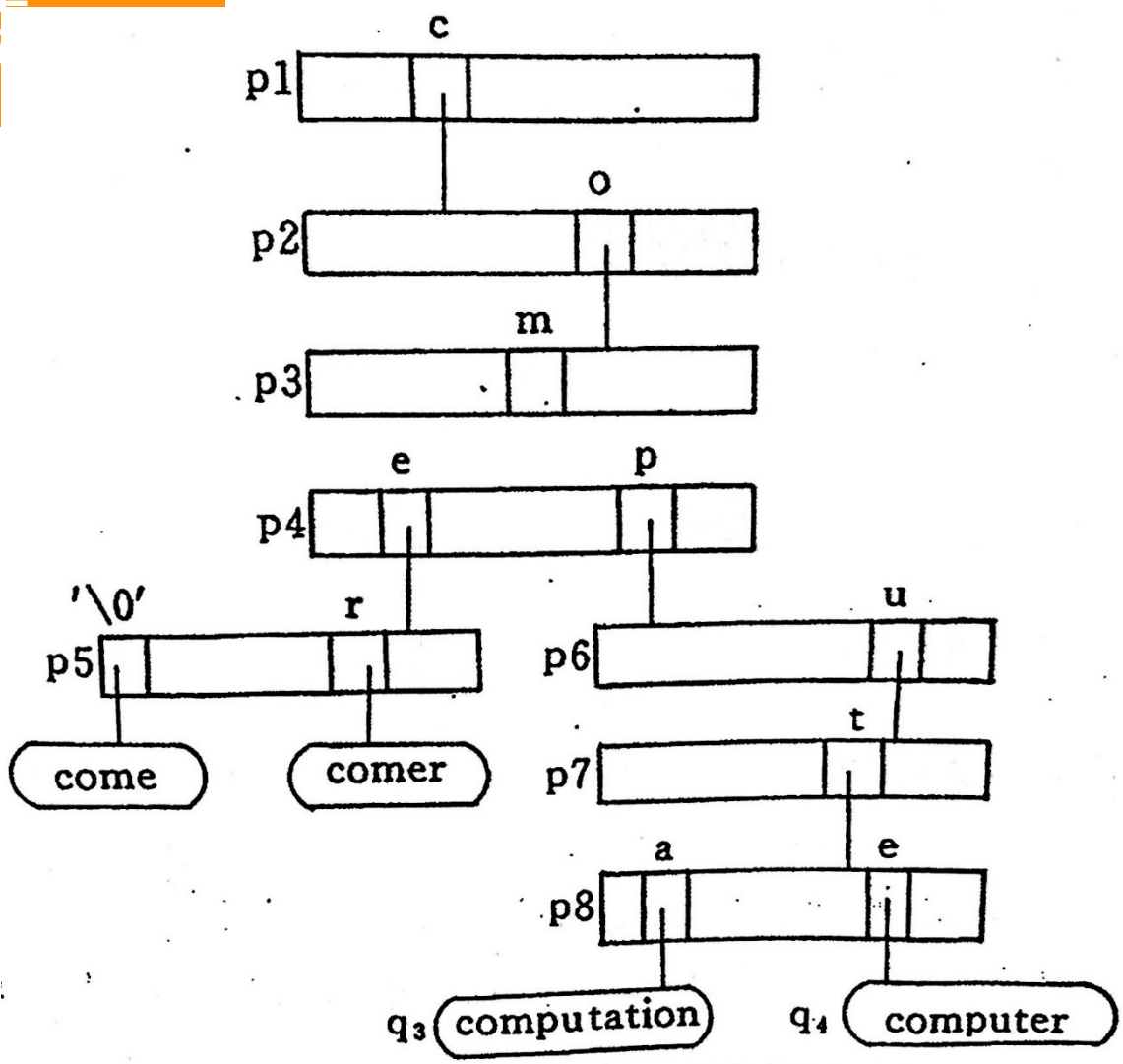


图6.8.4 插入 computer 后的“Trie 树”

删除

删除信息结点后，检查是否分支结点为根的子树只有一个信息结点，则可以将这个信息结点“向上”，删除冗余的分支结点。

可添加附加信息：以该分支结点为根的子树当前有多少个信息结点。

7 Trie结构

应用实例：中文分词，拼法检查算法，……

例如，正向最大匹配策略的中文分词

华东师范大学计算机系有很多学生。

词表：华东，师范，大学，师范大学，华东师范大学，计算机，计算，……

用 Trie结构表示这个词表，匹配的过程类似查询的过程。

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆的定义：

假设二叉树T是一棵完全二叉树(拟满树)，如果树T中的任一结点的值不小于它的左子结点的值（如果左子结点存在的话），且不小于它的右子结点的值（如果右子结点存在的话），那么我们称树T是一个堆（heap）.

存放：对于具有n个结点的完全二叉树T，我们可以按层次序把树中的所有结点存放在数组a[n]中。这样，我们可以用数组a[n]表示完全二叉树T。

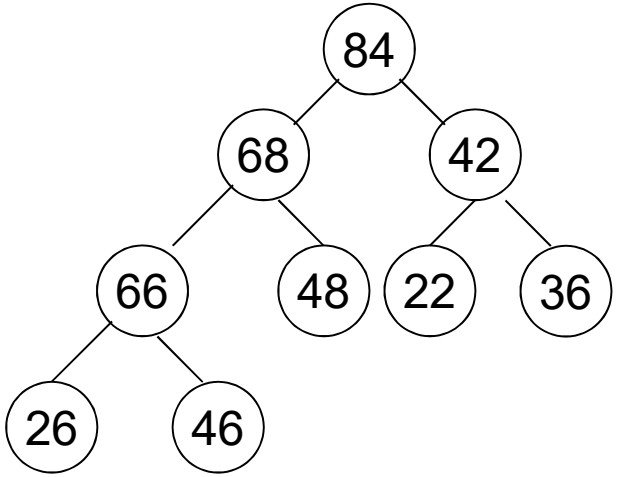
提问：完全二叉树按层次序存放，有哪些性质？
最后一个非叶子结点的下标？

3 堆和堆排序

堆的性质：如果具有n个结点的完全二叉树T是一个堆，按层次序存放树T的数组a[n]。数组a[n]中的元素具有如下的性质：

（1）若2i+1<n, 则 a[i]≥a[2i+1]
（2）若2i+2<n, 则 a[i]≥a[2i+2]

例：键值序列{84，68，42，66，48，22 ，36，26，46}



根据堆的定义，
最大的结点？根

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序

主要工作：调整堆

a[n]:待排序的全部结点组成的序列。
堆排序：借助“堆”的逻辑结构，在a[n]中交换，完成排序。

过程：

（1）建堆，a[0]..a[n-1]表示一个堆
（2）i从n-1到1(共进行n-1次)，
a)交换a[0]与a[i],
b)将a[0]到a[i-1]之间的结点调整成堆。
重复执行，直到调整范围内只有一个结点a[0]为止。

```
void siftdown(int a[], int i, int n)
//调整以a[i]为根的二叉树，使其成为堆。前提条件:树中各子树都是堆。
{      int j,t;
       t=a[i];
       while ( (j=2*i+1)< n  )  //当i有子结点的时候才需要调整
        {     if (j<n-1  &&  a[j]<a[j+1] ) j++;
              //j指向a[i]左子结点和右子结点（如果有）中较大的一个
              if  (t < a[j]) //如果t比子结点小，不符合堆的定义
               {    a[i]=a[j];  //将较大的值放入根a[i]。
                    i=j;  //  结点j作为新的将要调整的根
                }
               else break;//t大于左右子结点，符合定义，终止循环
        }
        a[i]=t;     //循环终止后， t填入当前停止的结点
}
```

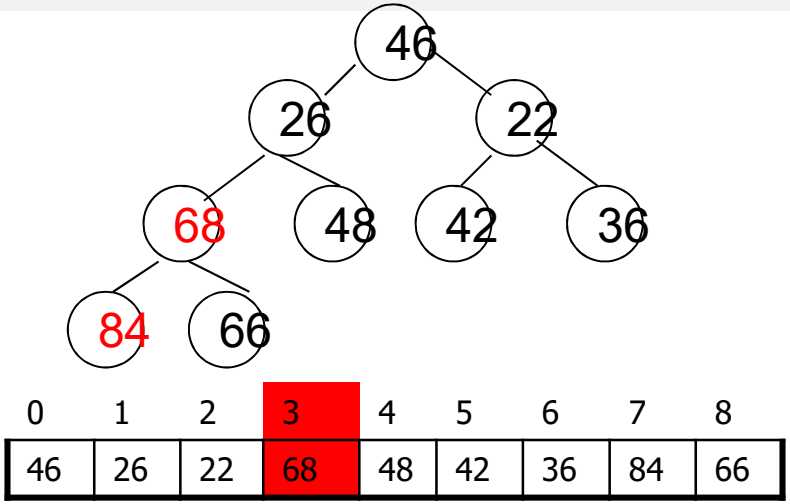
```
void siftdown(int a[], int i, int n)
//调整以a[i]为根的二叉树，使其成为堆。前提条件:树中各子树都是堆。
{     int j; //j是i的子结点
      while ( (j=2*i+1)< n  ) //当i有子结点的时候才需要调整
       { if (j<n-1  &&  a[j]<a[j+1] ) j++;
         //j指向a[i]左子结点和右子结点（如果有）中较大的一个
         if  (a[i] < a[j]) //如果t比子结点小，不符合堆的定义
             {    swap(a[i],a[j]); //将较大的值和根a[i]交换
                  i=j;  //  结点j作为新的将要调整的根
              }
             else break;//t大于左右子结点，符合定义，终止循环
       }
}
```

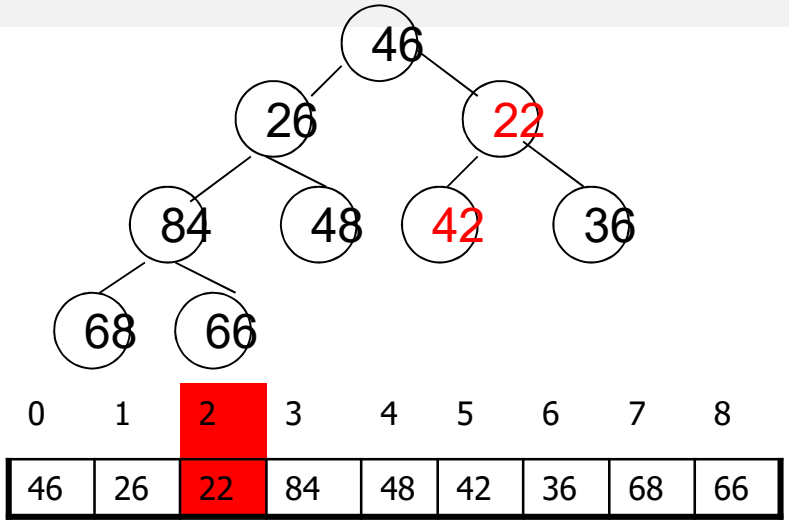
jyang
@cs.ecnu.edu.cn

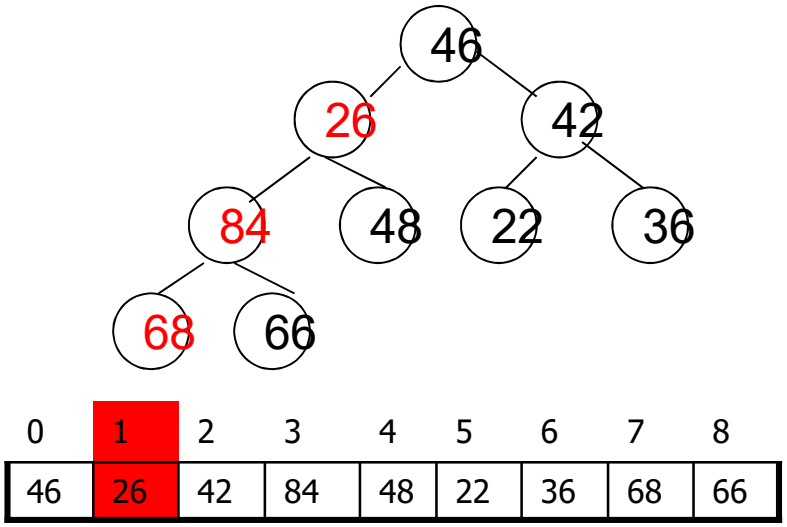
堆排序：

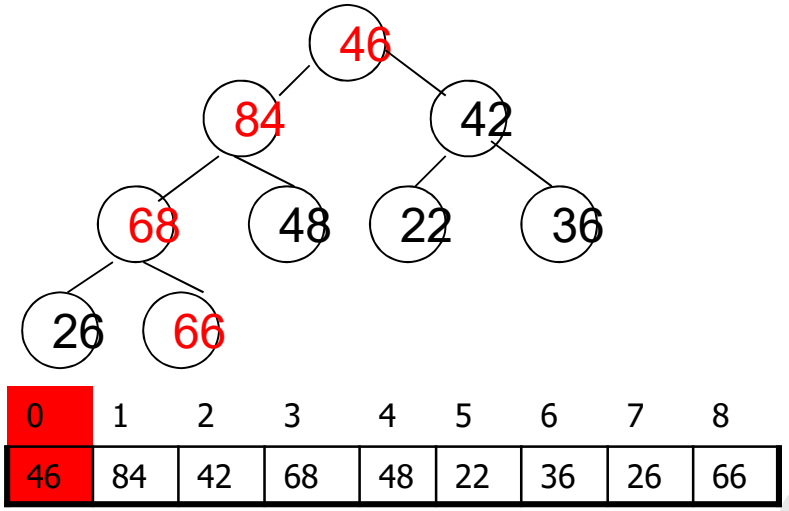
```
Void heap_sort(int a[],int n)
{       int i,m,t;
        for (i=(n-2)/2;i>=0;i--)    siftdown(a,i,n); //从最后一个非叶子结点开始，逐渐调整，向上一直到根a[0],构成初始堆

        for (m=n-1;m>0;m--)   //最大值在a[0],与正在处理的序列的末尾a[i]对换，然后调整以a[0]为根的树使其还是堆(调整以前，子树都是堆)
        {       t=a[0];
                a[0]=a[m];
                a[m]=t;
                siftdown(a,0,m); //调整以a[0]为根的二叉树，调整涉及的结点有m个，
        }
}
```

堆排序——例：建立初始堆


46
26
22
68
48
42
36
84
66
0 1 2 3 4 5 6 7 8
46 26 22 68 48 42 36 84 66


46
26
22
84
48
42
36
68
66
0 1 2 3 4 5 6 7 8
46 26 22 84 48 42 36 68 66

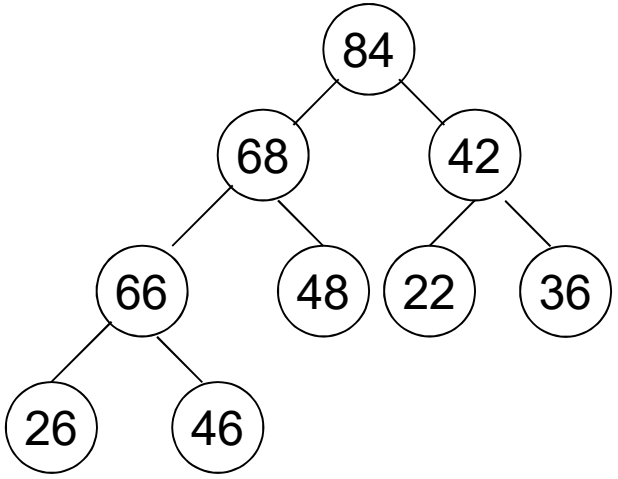

46
26
42
84
48
22
36
68
66
0 1 2 3 4 5 6 7 8
46 26 42 84 48 22 36 68 66


46
84
42
68
48
22
36
26
66
0 1 2 3 4 5 6 7 8
46 84 42 68 48 22 36 26 66

堆排序——例：建立初始堆

建立完成的初始堆

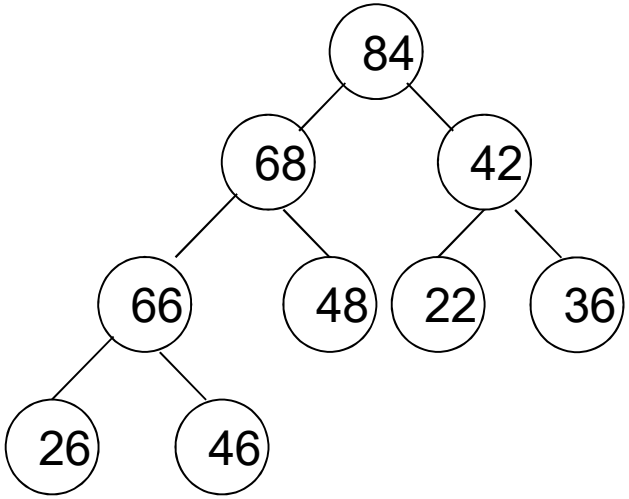
0	1	2	3	4	5	6	7	8
84	68	42	66	48	22	36	26	46

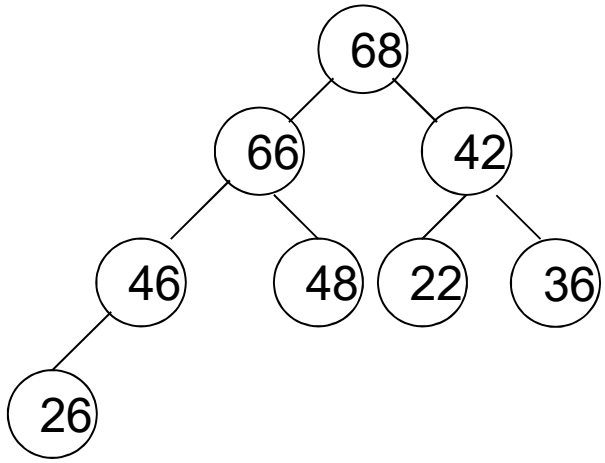


jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序——交换，调整





交换，调整

84

0	1	2	3	4	5	6	7	8
84	68	42	66	48	22	36	26	46

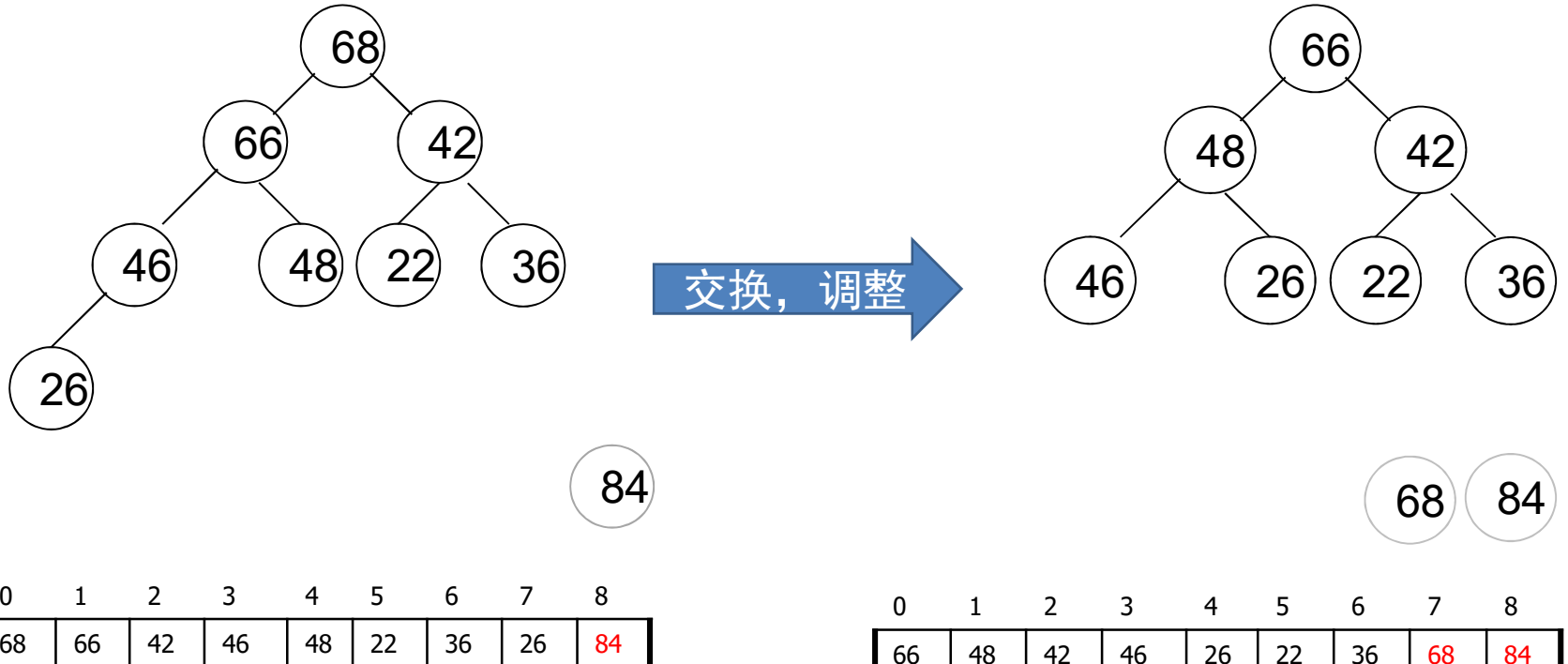
0	1	2	3	4	5	6	7	8
68	66	42	46	48	22	36	26	84

交换：a[0],a[8]
调整：siftdown(a,0,7),
在调整过程中发生的交换为：（0,1）（1,3）

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序——交换，调整



84

0	1	2	3	4	5	6	7	8
68	66	42	46	48	22	36	26	84

68 84

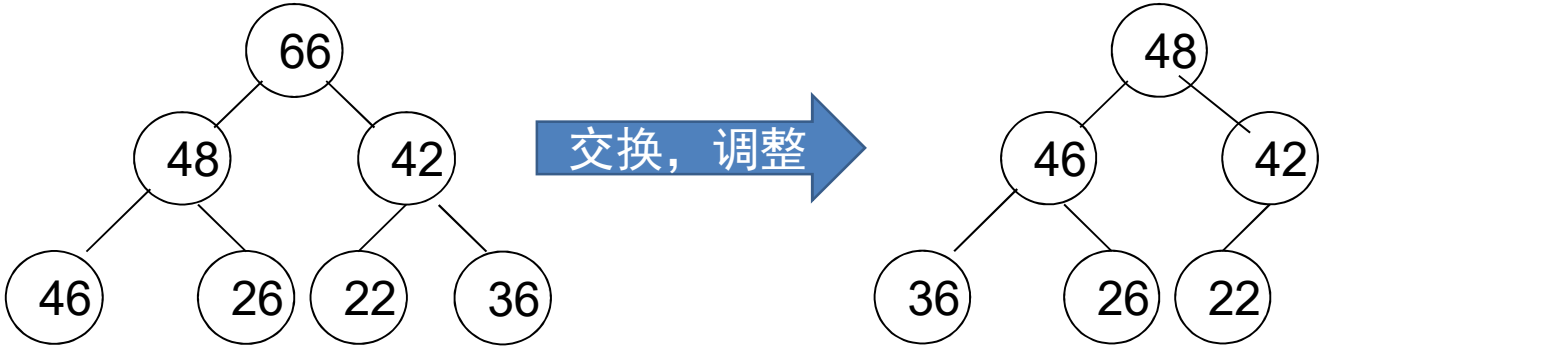
0	1	2	3	4	5	6	7	8
66	48	42	46	26	22	36	68	84

交换：a[0],a[7]
调整：siftdown(a,0,7),
在调整过程中发生的交换为：（0,1）（1,4）

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序——交换，调整

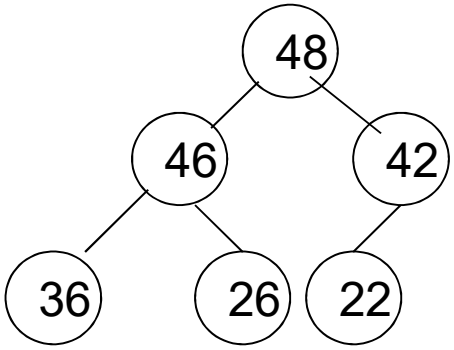


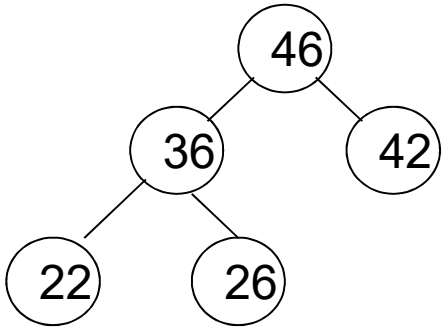
交换：a[0],a[6]
调整：siftdown(a,0,5),
在调整过程中发生的交换为：（0,1）（1,3）

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序——交换，调整





0	1	2	3	4	5	6	7	8
48	46	42	36	26	22	66	68	84

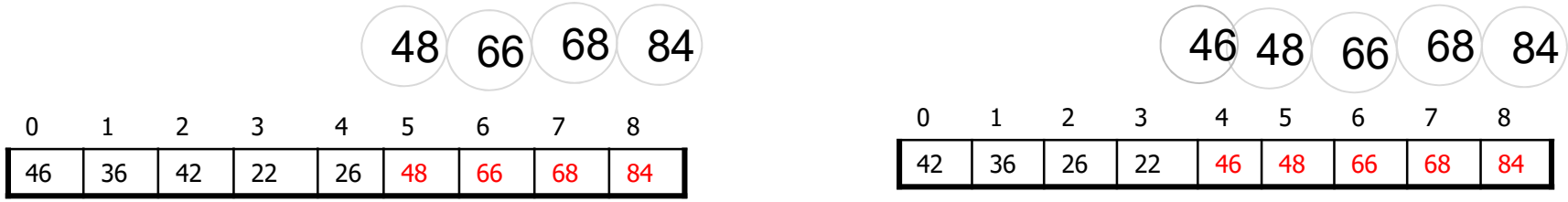
0	1	2	3	4	5	6	7	8
46	36	42	22	26	48	66	68	84

交换：a[0],a[5]
调整：siftdown(a,0,4),
在调整过程中发生的交换为：（0,1）（1,3）

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序——交换，调整

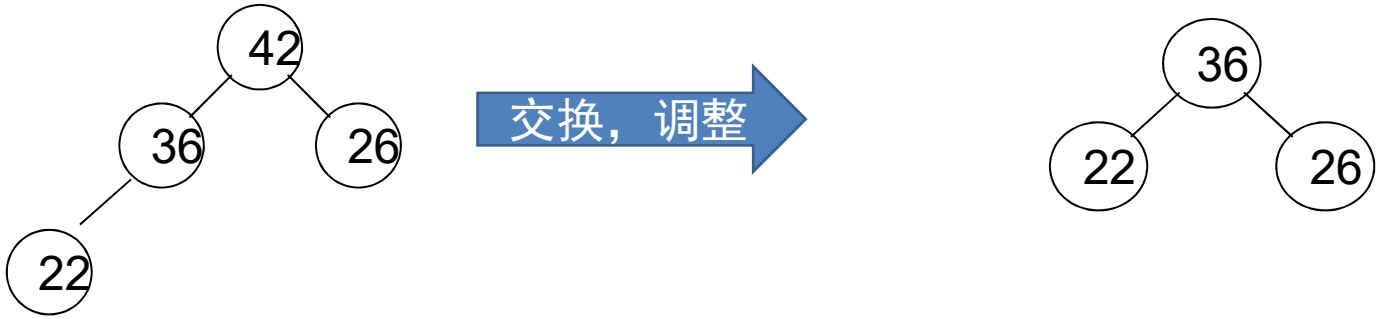


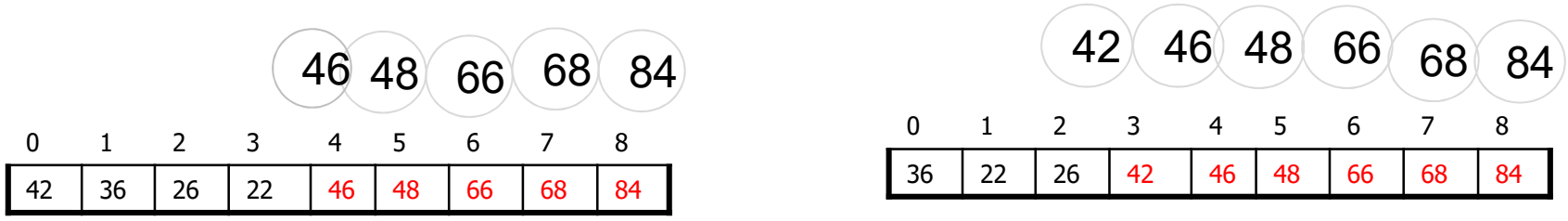
交换：a[0],a[4]
调整：siftdown(a,0,3),
在调整过程中发生的交换为：（0,2）

jyang
@cs.ecnu.edu.cn

3 堆和堆排序

堆排序——交换，调整



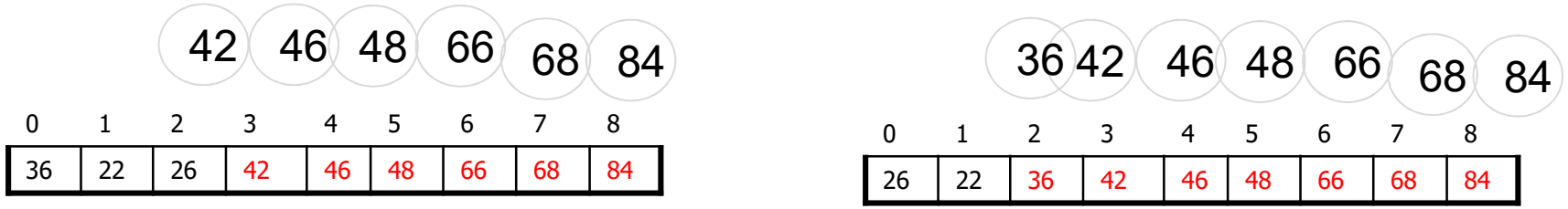


交换：a[0],a[3]
调整：siftdown(a,0,2),
在调整过程中发生的交换为：（0,1）

jyang
@cs.ecnu.edu.cn

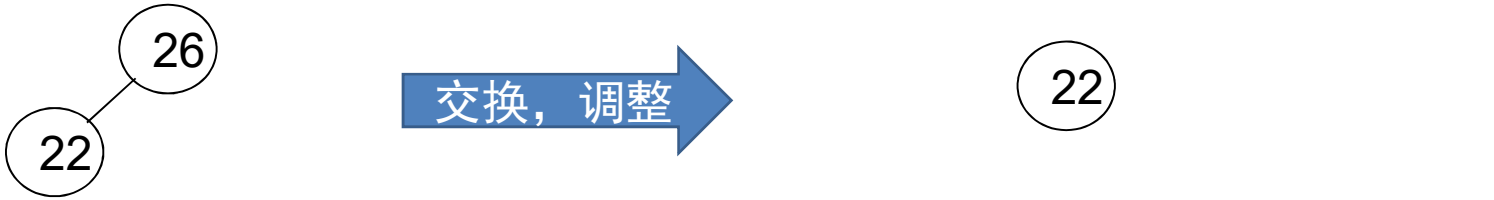
3 堆和堆排序

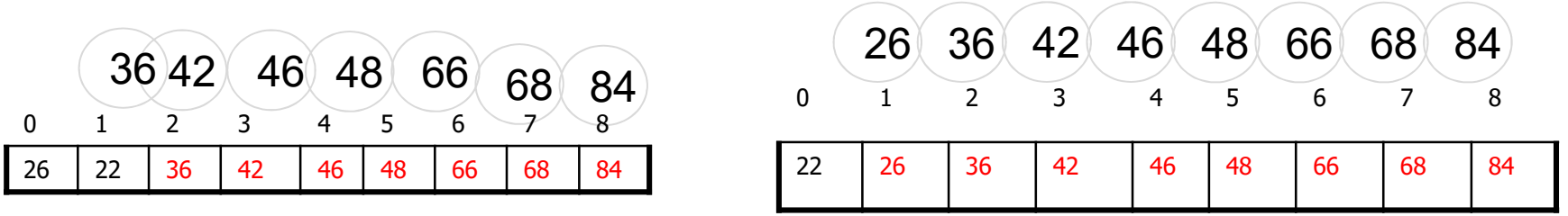
堆排序——交换，调整



交换：a[0],a[2]
调整：siftdown(a,0,1),
在调整过程中发生的交换为：不需要交换

堆排序——交换，调整





交换：a[0],a[1]
调整：siftdown(a,0,0)，函数中的循环不执行

堆排序——执行时间分析

$O(n\log_2 n)$

堆排序是不稳定的。

9 解答树

用回溯法查找解答树

某类问题，只需要检查有限多个可能性就可以得到关于它的答案。通常把这些可能性看成树的结点，这种树称为解答树。

为了找出解答位置，可以分为两步：
（1）构造解答树。
（2）检查解答树的结点，看它们是否对应于解答位置。

为了加快检查结点的速度，通常约定：任何一棵子树，只要它的根结点不满足条件，它就不会含有解答树位置。因此，在第一步构造解答树时可以不构造这种子树。

jyang
@cs.ecnu.edu.cn

9 解答树

用回溯法查找解答树

如果逐步构造和检查解答树的工作是由左到右逐个树枝进行的，那么一般来说，为检查长度为n的树枝T，只要保留n个结点。（在T的左边的结点已经检查过，而T的右边的一切结点还没有产生）。

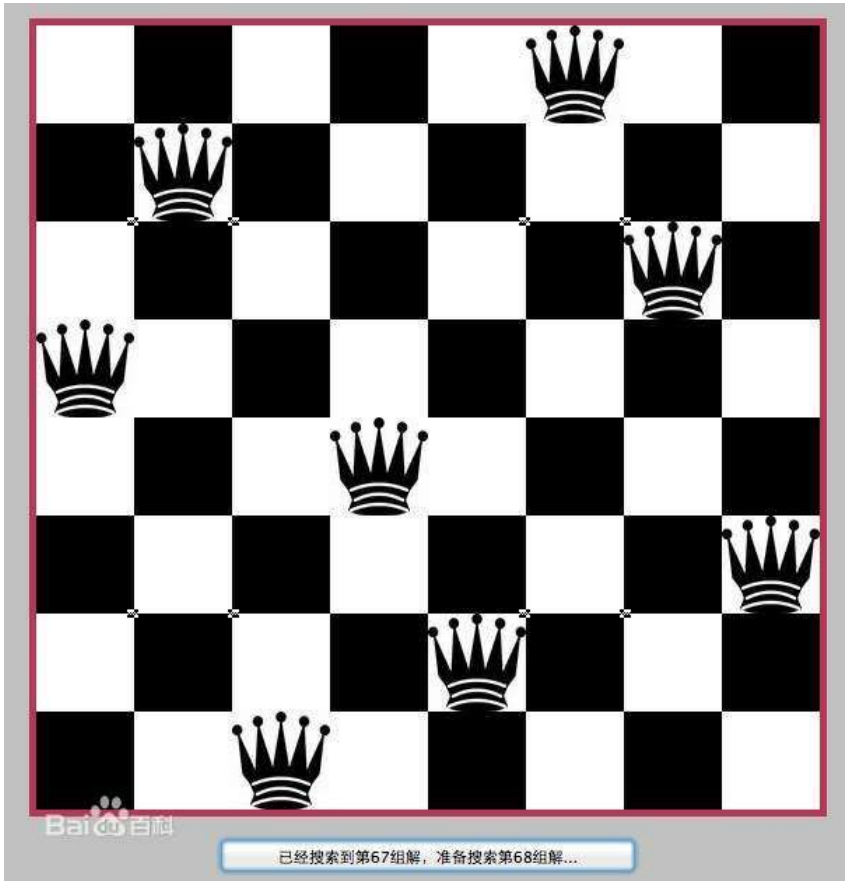
在按这种方式得到一条从树根到达叶子的树枝T后，要得到T右边的下一条树枝的方法是：沿着树枝T回溯到第一个尚未用过的向右分支点。由此向右走一步，随后仍然向叶子走，直到到达叶子为止。

称这种查找解答树的方法为回溯法。

9 解答树

皇后问题：

八皇后问题：该问题是国际西洋棋棋手马克斯·贝瑟尔于1848年提出：在8×8格的国际象棋上摆放八个皇后，使其不能互相攻击——即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。



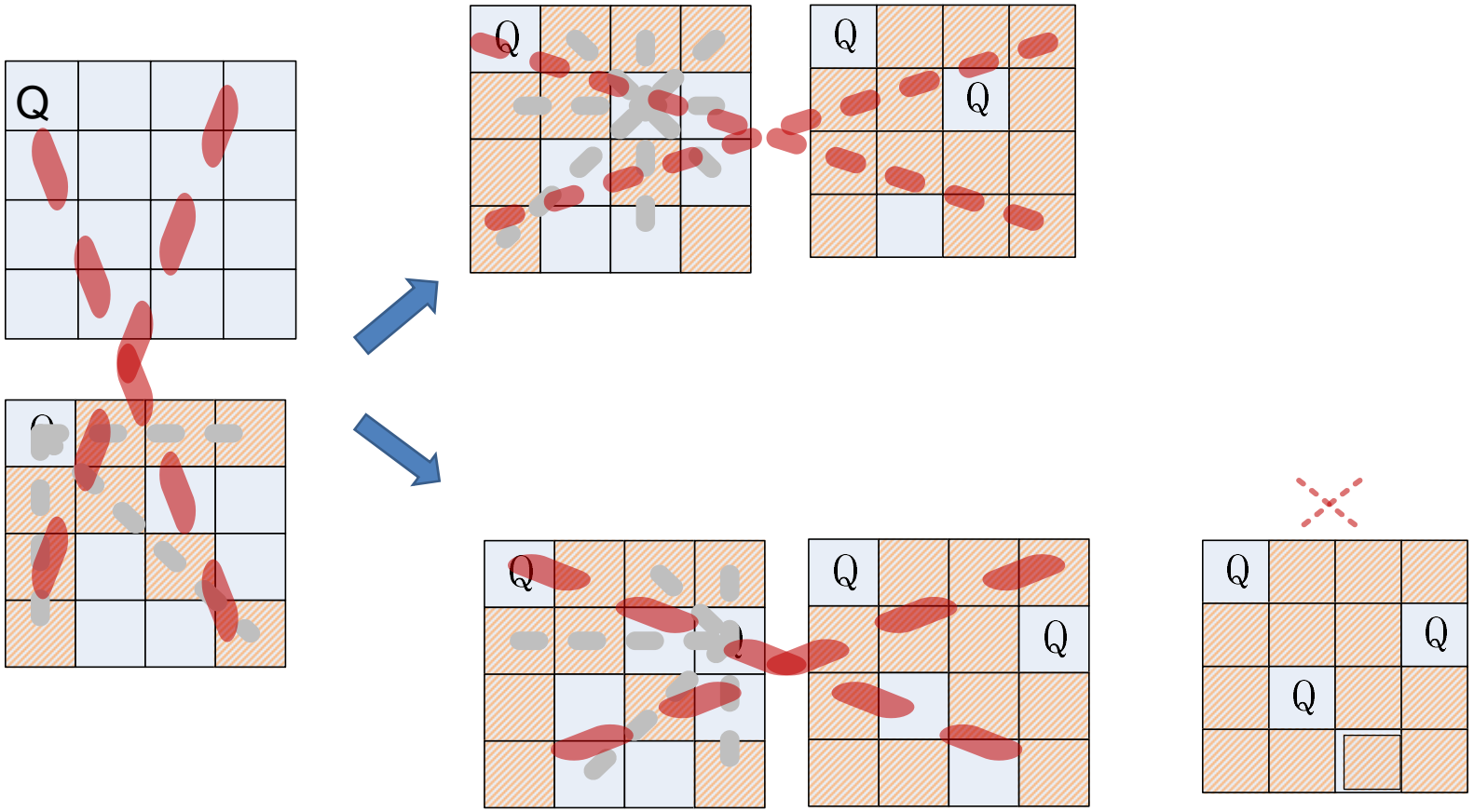
9 解答树

皇后问题：

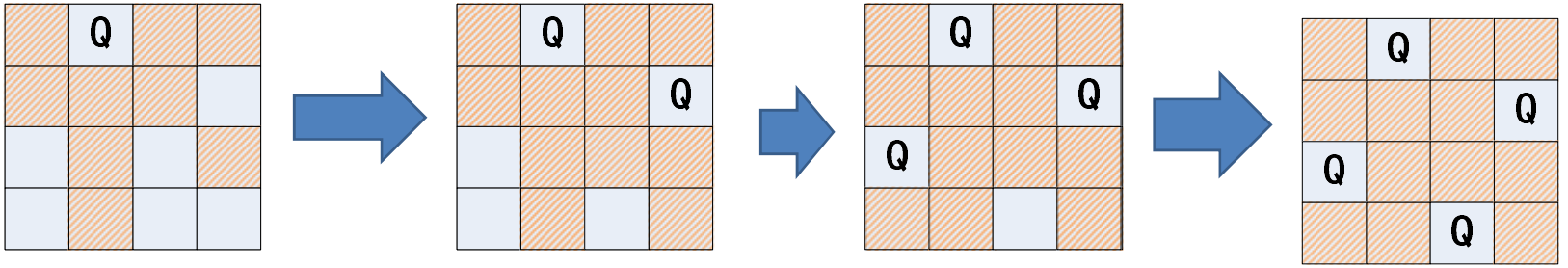
由n^2个方块排成n行n列的正方形，叫做n元棋盘。如果两个皇后位于n元棋盘的同一行、同一列或同一条对角线上，那么称它们在互相攻击。

N元棋盘上的皇后问题：找出使棋盘上的n个皇后互不攻击的布局。

皇后问题



皇后问题



N=4

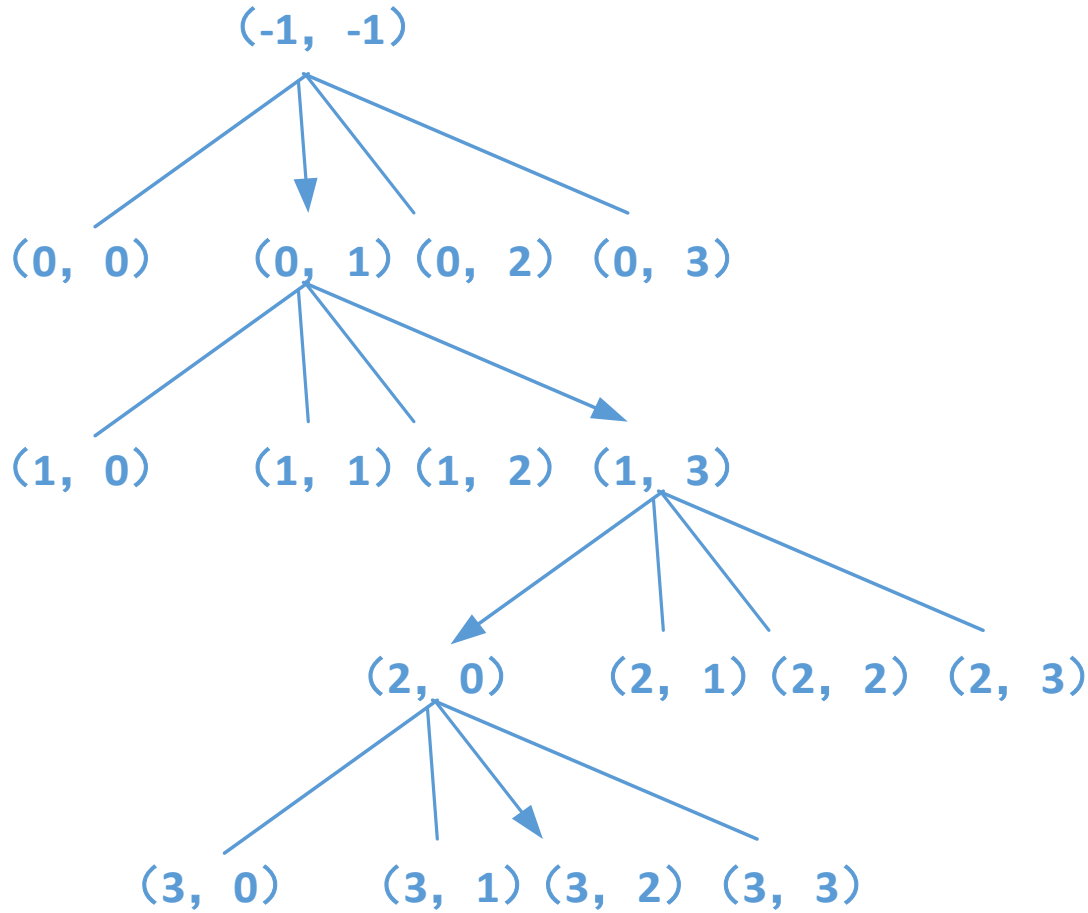
(0, 1) (1, 3) (2, 0) (3, 2)
(0, 2) (1, 0) (2, 3) (3, 1)

9 解答树

皇后问题

可以用解答树来描述皇后问题的解答。
解答树中的结点，写成（r, c），
表示r行，c列位置上放置了一个皇后。
取（-1, -1）为初态结点，

（逐行）
（n-1, c）是终止结点。



9 解答树

皇后问题

N元棋盘，n个行由上到下编号，n个列由左到右编号。
（2n-1）条主对角线由右上角到左下角编号。
（2n-1）条次对角线由左上角到右下角编号。

皇后问题

从左到右逐条检查解答树的树枝。

在检查的过程中，使用一个栈，存放（r，c，tag）
tag取值为0或1，当新结点进栈时，它的tag置为0；
当栈顶结点k被判断为可能成为布局中的一个位置时（k所代表的皇后没有在攻击别的皇后，即：C列、通过（r，c）位置的主对角线、次对角线上没有皇后），把栈顶结点的tag置1，并产生它的n个子结点，让这n个子结点进栈，同时置这n个进栈的子结点的tag为0。

9 解答树

皇后问题

P197。