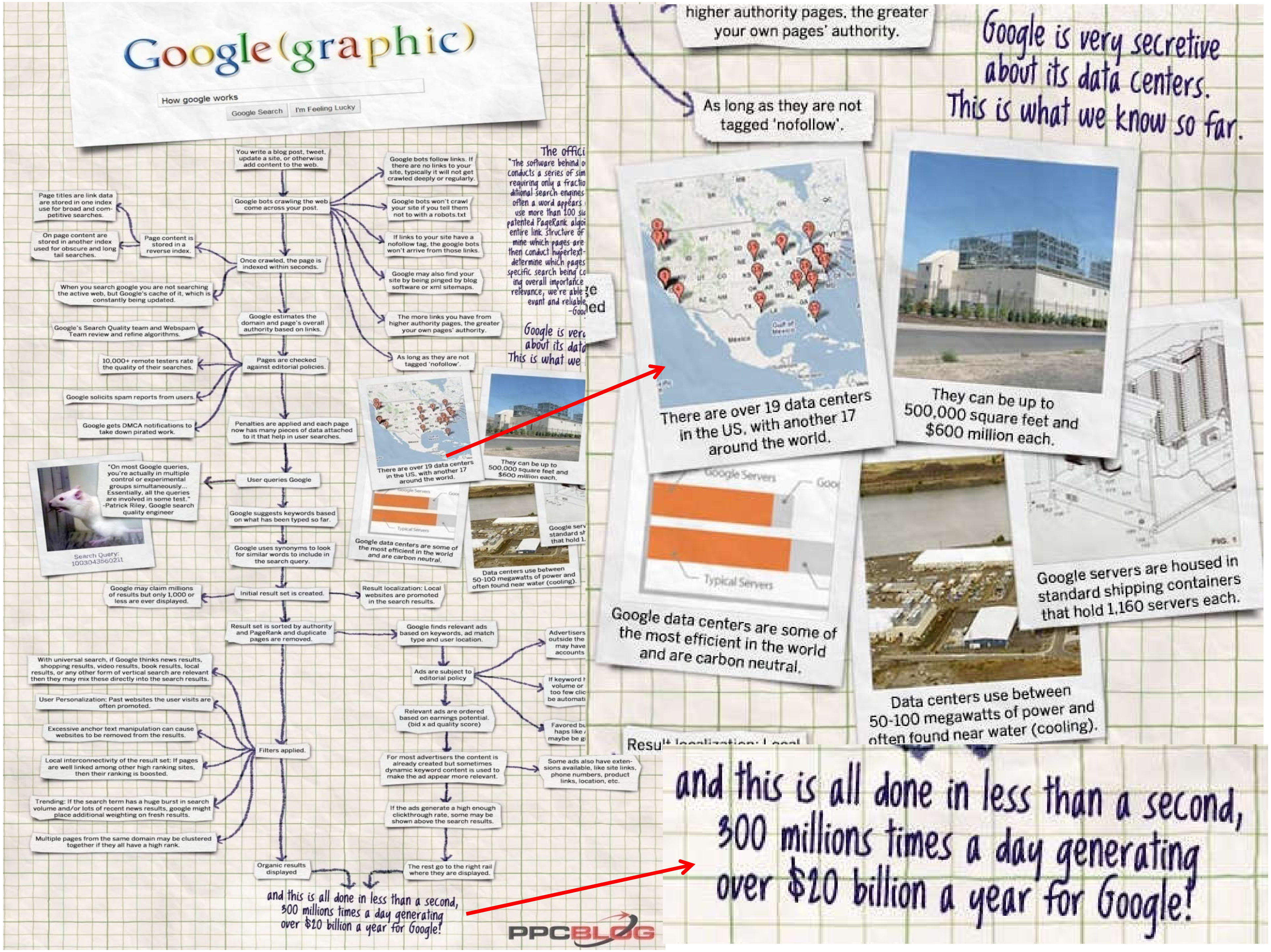
数据结构与算法
查找

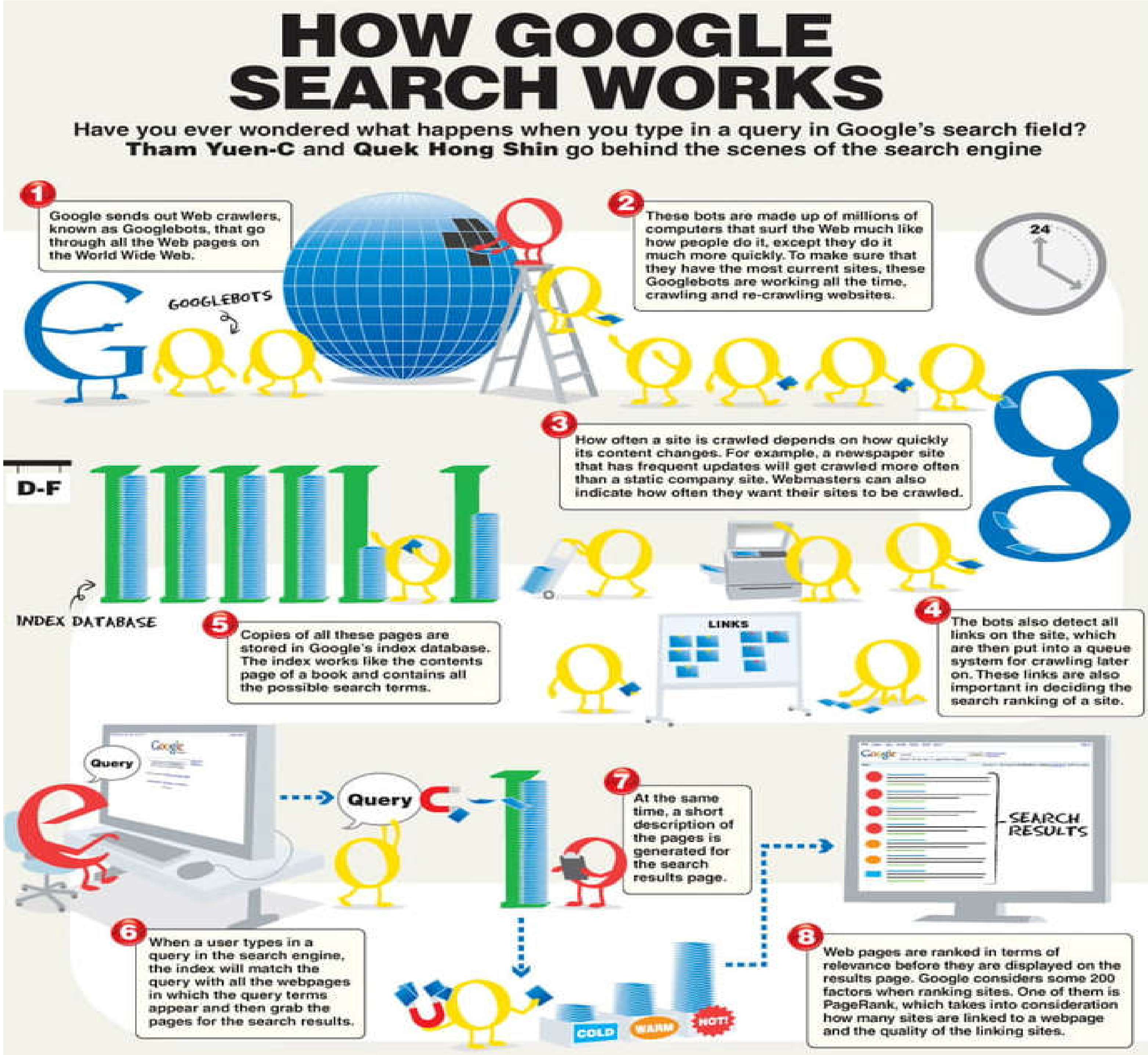
臧天仪 教授

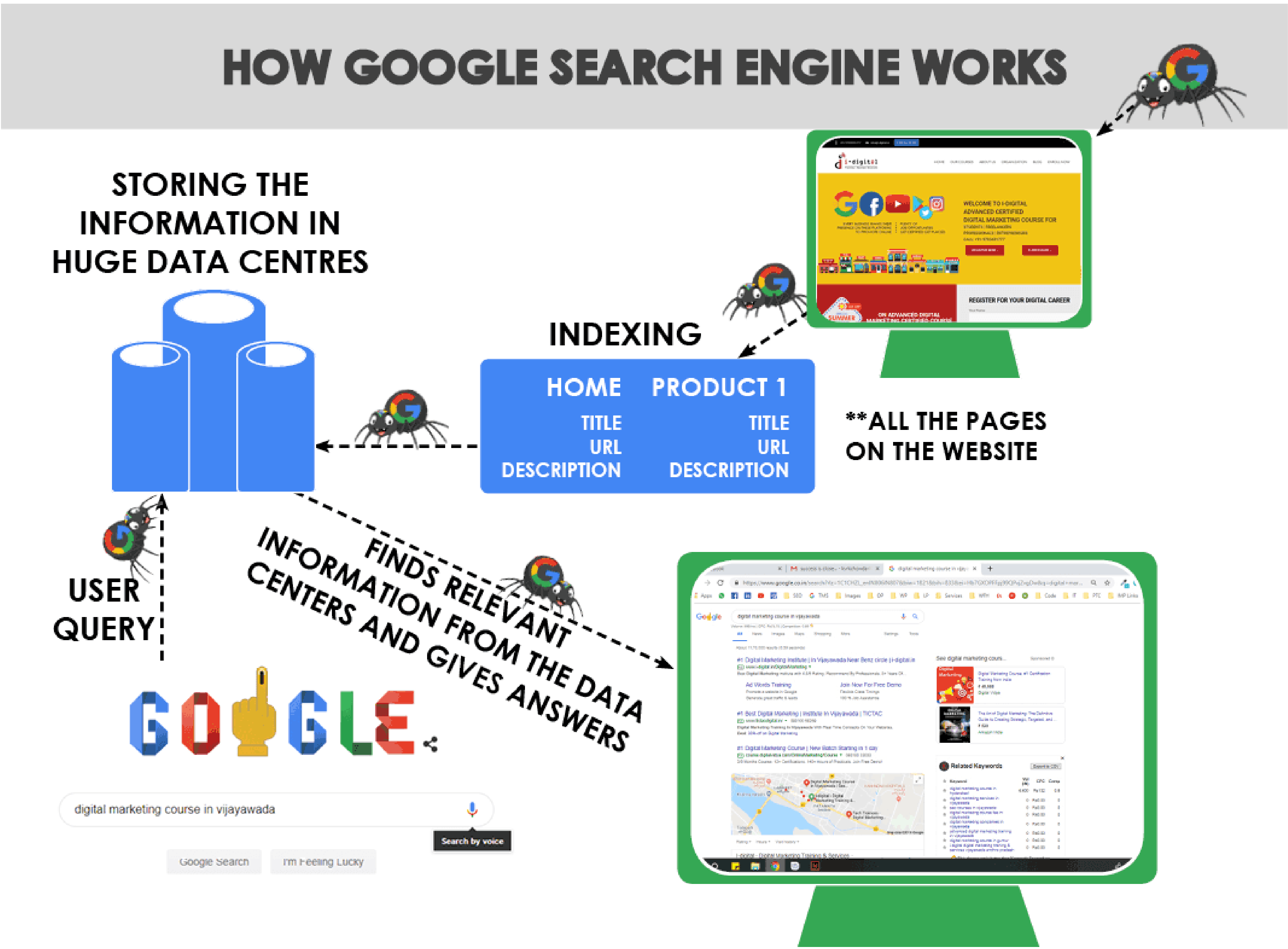
tianyi.zang@hit.edu.cn

哈尔滨工业大学计算学部



Google(graphic)
How google works
Google Search
I'm Feeling Lucky
You write a blog post, tweet, update a site, or otherwise add content to the web.
Google bots follow links. If there are no links to your site, typically it will not get crawled deeply or regularly.
Google bots crawling the web come across your post.
Google bots won't crawl your site if you tell them not to with a robots.txt
If links to your site have a nofollow tag, the google bots won't arrive from those links.
Google may also find your site by being pinged by blog software or xml sitemaps.
The more links you have from higher authority pages, the greater your own pages' authority.
As long as they are not tagged 'nofollow'.
Page titles are link data are stored in one index use for broad and competitive searches.
On page content are stored in another index used for obscure and long tail searches.
Page content is stored in a reverse index.
Once crawled, the page is indexed within seconds.
When you search google you are not searching the active web, but Google's cache of it, which is constantly being updated.
Google estimates the domain and page's overall authority based on links.
Google's Search Quality team and Webspam Team review and refine algorithms.
10,000+ remote testers rate the quality of their searches.
Pages are checked against editorial policies.
Google solicits spam reports from users.
Google gets DMCA notifications to take down pirated work.
Penalties are applied and each page now has many pieces of data attached to it that help in user searches.
"On most Google queries, you're actually in multiple control or experimental groups simultaneously... Essentially, all the queries are involved in some test." -Patrick Riley, Google search quality engineer
Search Query: 1003043560211
User queries Google
Google suggests keywords based on what has been typed so far.
Google uses synonyms to look for similar words to include in the search query.
Google may claim millions of results but only 1,000 or less are ever displayed.
Initial result set is created.
Result localization: Local websites are promoted in the search results.
Result set is sorted by authority and PageRank and duplicate pages are removed.
Google finds relevant ads based on keywords, ad match type and user location.
Ads are subject to editorial policy
Relevant ads are ordered based on earnings potential. (bid x ad quality score)
With universal search, if Google thinks news results, shopping results, video results, book results, local results, or any other form of vertical search are relevant then they may mix these directly into the search results.
User Personalization: Past websites the user visits are often promoted.
Excessive anchor text manipulation can cause websites to be removed from the results.
Local interconnectivity of the result set: If pages are well linked among other high ranking sites, then their ranking is boosted.
Trending: If the search term has a huge burst in search volume and/or lots of recent news results, google might place additional weighting on fresh results.
Multiple pages from the same domain may be clustered together if they all have a high rank.
Filters applied.
For most advertisers the content is already created but sometimes dynamic keyword content is used to make the ad appear more relevant.
Some ads also have extensions available, like site links, phone numbers, product links, location, etc.
If the ads generate a high enough clickthrough rate, some may be shown above the search results.
Organic results displayed
The rest go to the right rail where they are displayed.
higher authority pages, the greater your own pages' authority.
As long as they are not tagged 'nofollow'.
Google is very secretive about its data centers. This is what we know so far.
There are over 19 data centers in the US, with another 17 around the world.
They can be up to 500,000 square feet and $600 million each.
Google Servers
Typical Servers
Google data centers are some of the most efficient in the world and are carbon neutral.
Data centers use between 50-100 megawatts of power and often found near water (cooling).
Google servers are housed in standard shipping containers that hold 1,160 servers each.
and this is all done in less than a second, 300 millions times a day generating over $20 billion a year for Google!
PPCBLOG


HOW GOOGLE SEARCH WORKS
Have you ever wondered what happens when you type in a query in Google's search field?
Tham Yuen-C and Quek Hong Shin go behind the scenes of the search engine
1
Google sends out Web crawlers, known as Googlebots, that go through all the Web pages on the World Wide Web.
GOOGLEBOTS
2
These bots are made up of millions of computers that surf the Web much like how people do it, except they do it much more quickly. To make sure that they have the most current sites, these Googlebots are working all the time, crawling and re-crawling websites.
24
3
How often a site is crawled depends on how quickly its content changes. For example, a newspaper site that has frequent updates will get crawled more often than a static company site. Webmasters can also indicate how often they want their sites to be crawled.
D-F
INDEX DATABASE
5
Copies of all these pages are stored in Google's index database. The index works like the contents page of a book and contains all the possible search terms.
LINKS
4
The bots also detect all links on the site, which are then put into a queue system for crawling later on. These links are also important in deciding the search ranking of a site.
Query
Query
7
At the same time, a short description of the pages is generated for the search results page.
SEARCH RESULTS
6
When a user types in a query in the search engine, the index will match the query with all the webpages in which the query terms appear and then grab the pages for the search results.
COLD
WARM
HOT!
8
Web pages are ranked in terms of relevance before they are displayed on the results page. Google considers some 200 factors when ranking sites. One of them is PageRank, which takes into consideration how many sites are linked to a webpage and the quality of the linking sites.


HOW GOOGLE SEARCH ENGINE WORKS
STORING THE INFORMATION IN HUGE DATA CENTRES
INDEXING
HOME
TITLE
URL
DESCRIPTION
PRODUCT 1
TITLE
URL
DESCRIPTION
**ALL THE PAGES ON THE WEBSITE
USER QUERY
FINDS RELEVANT INFORMATION FROM THE DATA CENTERS AND GIVES ANSWERS
digital marketing course in vijayawada
Search by voice
Google Search
I'm Feeling Lucky

学习目标

- 查找是在某种数据结构上，找出满足给定条件数据元素的操作，又称检索，是数据处理中的重要操作。
- 了解不同数据结构上的查找方法。
- 掌握各种查找结构的性质、查找算法的设计思想与实现方法。
- 掌握各种查找方法的时间性能(平均查找长度)的分析方法。
- 能够根据实际情况选择适合的数据结构与方法解决问题。

本章主要内容

5.1 基本概念和术语

- 查找表：由同一类型数据元素（记录）构成的集合(文件)。
- 关键字：可以标识一个记录的某个数据项或数据项组合。
- 键值：关键字的取值。
- 主关键字：可以唯一地标识一个记录的关键字。
- 次关键字：不能唯一地标识一个记录的关键字。
- 查找：在查找表中找出（确定）其关键字值等于给定值的数据元素（记录）。
- 查找结果：若在查找集合中找到了与给定值相匹配的数据元素则称查找成功；否则，称查找失败。

学号	姓名	性别	年龄	入学成绩
0001	张亮	男	19	625
0002	张亮	女	28	617
0003	刘楠	女	19	623
…	…	…	…	…

5.1 基本概念和术语(Cont.)

查找的分类：

- 根据记录键值和记录存储位置
  - 基于关键字比较的查找
    - 顺序查找、折半查找、分块查找
    - BST&AVL、B-树和B^+树
  - 基于关键字存储位置的查找
    - 散列法
- 根据查找数据集合的存储位置
  - 内查找：查找过程都在内存进行
  - 外查找：查找过程需要访问外存，如B树和B^+树。

查找的分类：

- 根据是否改变数据集合
  - 静态查找
    - 查找+提取数据元素属性信息
    - 查找后并不改变被查找数据集合，既不插入新记录，也不删除原有记录。
  - 动态查找
    - 查找+（插入或删除元素）
    - 查找后可能改变被查找数据集合，可插入新记录，也可删除原有记录。

5.1 基本概念和术语(Cont.)

查找结构（表）：

- 面向查找操作的数据结构，查找所使用的数据结构。
- 查找结构决定查找方法
- 主要查找结构：
  - 线性表：适用于静态查找，主要采用线性（顺序）查找、折半查找技术。
  - 树表：静态和动态查找均适用，主要采用BST、AVL和B树等查找技术。
  - 散列表：静态和动态查找均适用，主要采用散列技术。

5.1 基本概念和术语(Cont.)

查找结构（表）的操作

- SEARCH (k，F)：
  – 在数据集合(查找表、文件) F 中查找关键字值等于k 的数据元素（记录）。
  – 若查找成功，则返回包含 k 的记录位置；否则，返回一个特定值。
- INSERT (R，F)：
  – 在F 中查找记录 R；
  – 若查找不成功，则插入R；否则，不插入。
- DELETE(k，F)：
  – 在 F 中查找关键字值等于k 的数据元素（记录）；
  – 若查找成功，则删除关键字值等于k 的记录；否则，不删除任何记录。

5.1 基本概念和术语(Cont.)

- 查找表结点（数据元素或记录）类型定义：

```
struct  records{
        keytype  key;
        fields other;
};
```

- 查找性能
  - 查找算法时间性能由关键字的比较次数来度量。
  - 同一查找集合、同一查找算法，关键字的比较次数与哪些因素有关呢？
  - 查找算法的时间复杂度是问题规模n和待查关键字在查找集合中位置k的函数，记为：$T(n，k)$。

5.1 基本概念和术语(Cont.)

查找性能

- 平均查找长度：
  - 给定值与关键字比较次数的期望值，被称为查找算法在查找成功时的平均查找长度 (Average Search Length, ASL)。
  - 计算公式：假设待查的记录是查找表中第i 个记录，其在第i 个位置的概率为p_i，$\sum p_i=1$，c_i为查找第i 个记录所进行的比较次数，则在等概率情况下，即$p_i = 1/n$ 时，

$$ASL = \sum_{i=1}^{n} p_i c_i$$

$$ASL = \frac{1}{n} \sum_{i=1}^{n} c_i$$

5.2 线性查找

- 线性（顺序）查找基本思想：
  - 从线性表的一端开始，顺序扫描线性表，依次将扫描的结点关键字与给定值K比较。
  - 若当前扫描的结点关键字与k相等，则查找成功；
  - 若扫描结束后，仍未找到关键字等于k的结点，则查找失败。
- 线性（顺序）查找对存储结构要求
  - 适用于线性表的顺序存储结构：适用于静态查找
  - 适用于线性表的链式存储结构：也适用于动态查找

5.2 线性查找(Cont.)

顺序线性表上查找：适合于静态查找

- 顺序线性表的类型定义

  typedef records LIST[MaxSize] ;

  LIST F ;

- SEARCH操作：k=35

	0	1	2	3	4	5	6	7	8	9
F	35	10	15	24	6	12	35	40	98	55

哨兵

i *i* *i* *i* *i*

- 不适合顺序线性表的操作
  – INSERT操作和DELETE操作

5.2 基本概念和术语(Cont.)

SEARCH操作实现

//在F[1]…F[last]中查找关键字为k的记录，

//若找到，则返回该记录所在的下标，否则返回 0 。

```
int   SEARCH (keytype k, int last, LIST  F )
{   int  i ;
    F[0].key = k ;   /* F[0]为伪记录或哨兵 */
    i = last ;
    while ( F[i].key != k )
            i = i –1 ;
    return i ;
}
```

- 时间复杂度： O(n) ;
- $ASL_{成功}=(n+1)/2$，$ASL_{失败}=n+1$

5.2 线性查找(Cont.)

单向链式线性表上的查找：也适合于动态查找

- 单向表的类型定义

```
struct  celltype  {
      records  data  ;
      celltype  * next ;
};
typedef  celltype *LIST ;
```

- INSERT操作
- DELETE操作

/*在不带表头的单向链表中查找关键字为k 的记录，返回其指针*/

```
LIST  SEARCH(keytype k, LIST  F)
{     LIST p =F ;
      while ( p! = NULL )
          if ( p->data.key == k )
              return p ;
          else
              p = p->next ;
      return p ;
}
```

- 时间复杂度：O(n)
- $ASL_{成功}$=(n+1)/2， $ASL_{失败}$=n+1

5.3 折半查找（二分查找）

折半查找（也称二分查找）的要求(适用条件)：

- 查找表（被查找的数据集合）必须采用顺序式存储结构；
- 查找表中的数据元素（记录）必须按关键字有序。

	1	2	3	4	5	6	7	8	9	10	11
F	**05**	**13**	**19**	**21**	**37**	**56**	**64**	**75**	**80**	**88**	**92**

$$F[1].key \le F[2].key \le F[3].key \le \ldots \le F[last].key$$

或 $$F[1].key \ge F[2].key \ge F[3].key \ge \ldots \ge F[last].key$$

- 注意：折半查找只适合于静态查找。

5.3 折半查找(Cont.)

折半查找的基本思想：

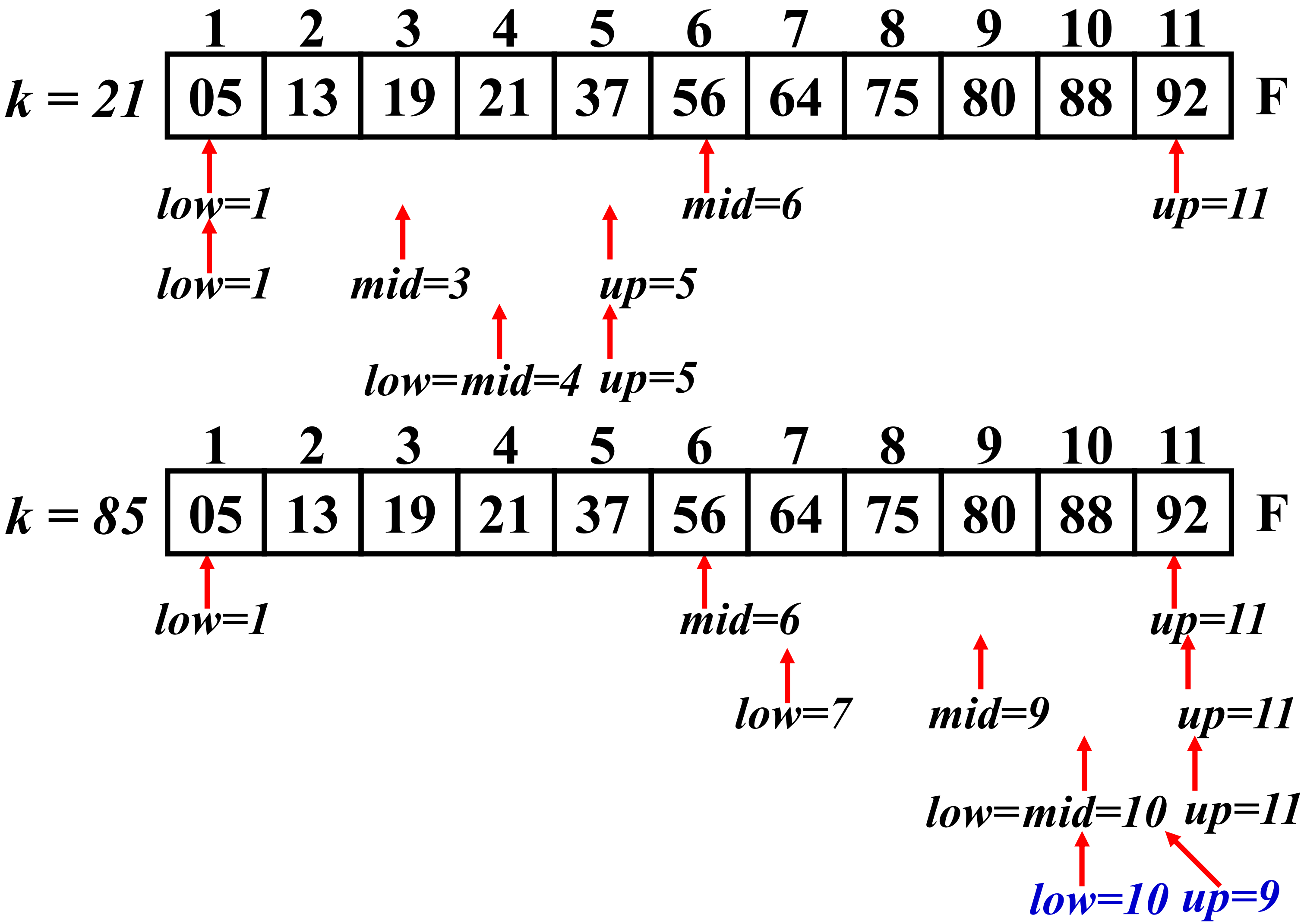
- 在有序表中，取中间记录作为比较对象，若给定值与中间记录的键值相等，则查找成功；
- 若给定值小于中间记录的键值，则在中间记录的左半区继续查找；
- 若给定值大于中间记录的键值，则在中间记录的右半区继续查找；
- 不断重复上述过程，直到查找成功；若查找区域无记录，查找失败。

（$mid=(1+n)/2$） k

$$\underbrace{[\; k_1 \;\dots\;\dots\;\dots\; k_{mid-1} \;]}_{\substack{\text{如果}k<k_{mid}\\ \text{查找左半区}}}\; \underset{\displaystyle k_{mid}}{\overset{\displaystyle k}{\updownarrow}} \;\underbrace{[\; k_{mid+1} \;\dots\;\dots\;\dots\; k_n \;]}_{\substack{\text{如果}k>k_{mid}\\ \text{查找右半区}}}$$

5.3 折半查找(Cont.)

折半查找示例：



5.3 折半查找(Cont.)

- 折半查找的非递归算法：实现步骤

1. 初态化：令low、up分别表示查找范围的上、下界，初始时low = 1, up = last;

2. 折半：令mid = (low+up)/2，取查找范围中间位置元素下标;

3. 比较：k与F[mid].key

3.1 若F[mid].key == k，查找成功，返回 mid;

3.2 若F[mid].key ＞ k，low不变，调整up = mid - 1，查找范围缩小一半；

3.3 若F[mid].key ＜ k，调整low = mid + 1，up不变，查找范围缩小一半；

4. 重复2 ~ 3 步。当low > up 时，查找失败，返回 0。

5.3 折半查找(Cont.)

折半查找的非递归算法：实现

```
int  BinSearch1(keytype  k, LIST F )
{    int  low , up , mid ;
     low = 1 ; up = last ;
     while ( low <= up ) {
          mid = ( low + up ) / 2 ;
           if ( F[mid].key = = k )      return   mid ;
           else if (F[mid].key > k)    up = mid – 1 ;
                  else                 low = mid + 1 ;
      }
      return  0;
}
```

- F必须是有序表（此处为增序）
- 时间复杂度：$O(\log_2 n)$

5.3 折半查找(Cont.)

折半查找的递归算法：实现步骤

1. 初态化：设置查找范围的上界up和下界low;

2. 测试查找范围：如果low > up，则查找失败；否则，

3.取查找范围中间位置元素下标：mid = (low+up)/2；
比较k与F[mid].key：

   3.1 若F[mid].key == k，查找成功，返回 mid;

   3.2 若F[mid].key ＞ k，，递归地在左半部分查找（low不变，调整up = mid – 1）；

   3.3 若F[mid].key ＜ k，递归地在右半部分查找（调整 low = mid + 1，up不变）。

5.3 折半查找(Cont.)

折半查找的递归算法：实现

```
int  BinSearch2(LIST F, int low, int up, keytype k )
{   if (low>up) return 0;
   else {
      mid=(low+up)/2;
      if (k < F[mid].key )
          return BinSearch2(F, low, mid-1, k);
      else  if (k>F[mid].key)
          return BinSearch2(F, mid+1, up, k);
      else return mid;
   }
}
```

- F必须是有序表（此处为增序）
- 时间复杂度：$O(\log_2 n)$

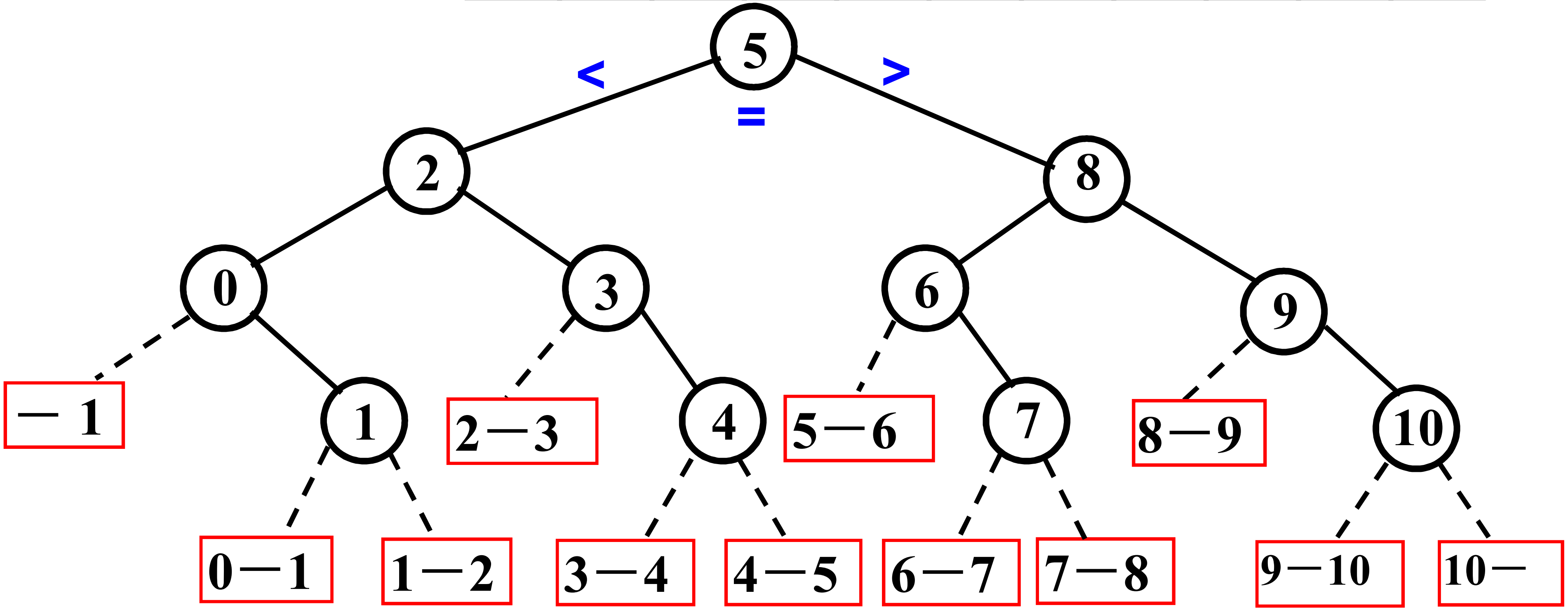
5.3 折半查找(Cont.)

- 折半查找判定树
  - 折半查找过程可以用二叉树来描述，树中每个结点对应有序表中的一个记录，结点值为该记录在表中的位置。
  - 描述折半查找过程的二叉树称为折半查找判定树，简称判定树。
- 折半查找判定树构造
  - 当n=0时，折半查找判定树为空；
  - 当n＞0时，折半查找判定树的根结点是有序表中序号为mid=(n-1)/2的记录；
  - 根结点的左子树为与F[0]～F[mid-1]相对应的折半查找判定树；
  - 根结点的右子树为与F[mid+1]～F[n-1]相对应的折半查找判定树。

5.3 折半查找(Cont.)

判定树构造

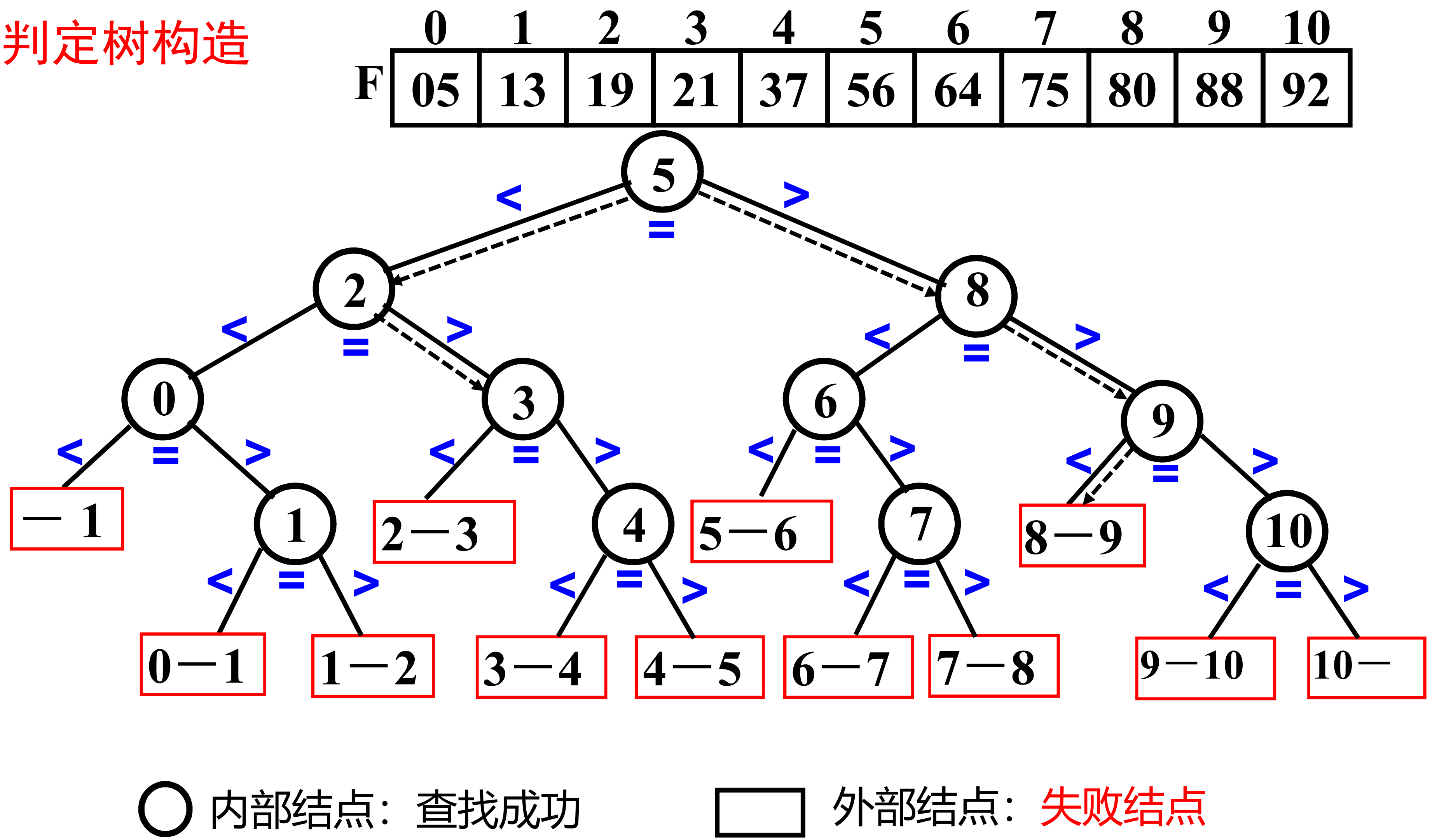
	0	1	2	3	4	5	6	7	8	9	10
F	05	13	19	21	37	56	64	75	80	88	92



内部结点：查找成功　　外部结点：失败结点

5.3 折半查找(Cont.)

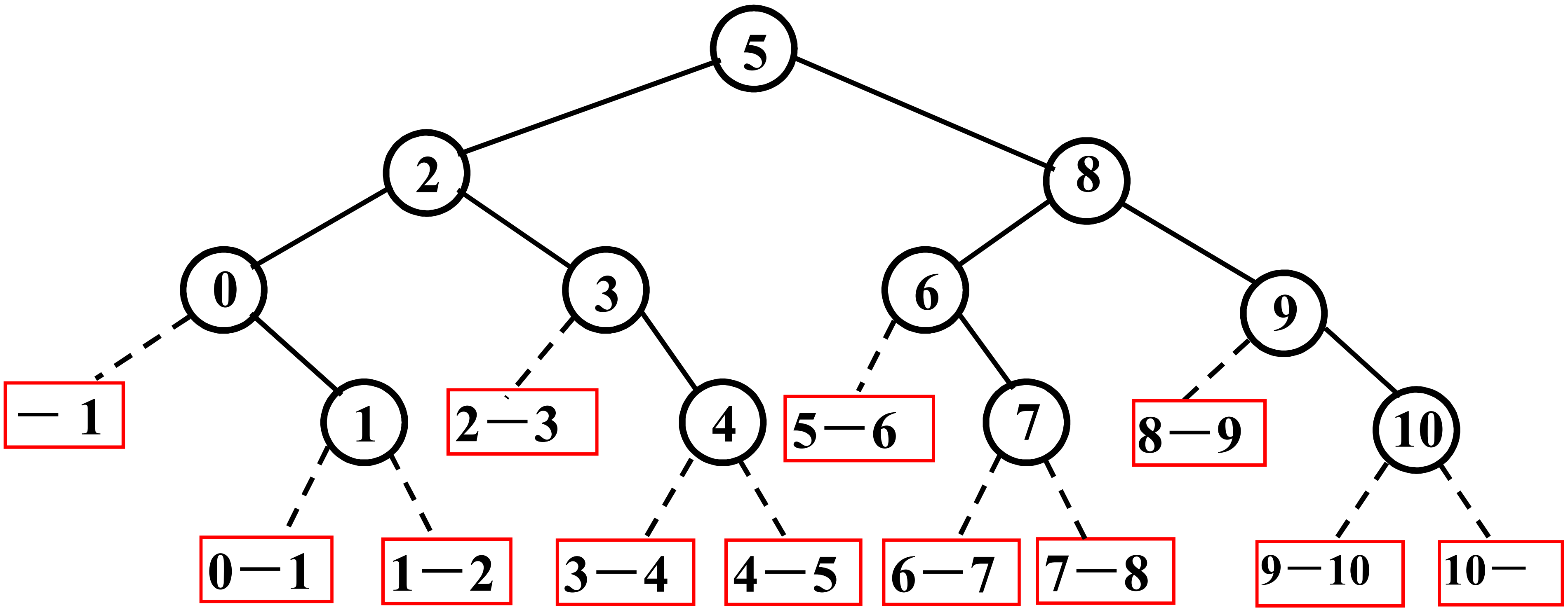
判定树构造



5.3 折半查找(Cont.)

折半查找的ASL

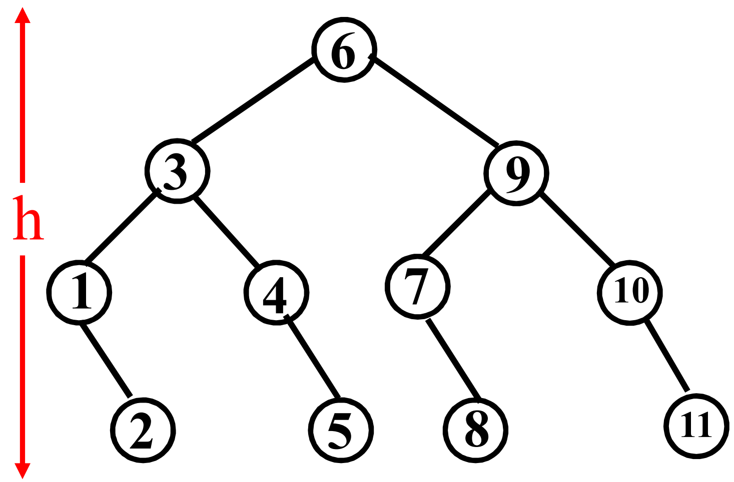
- 若有n个关键字，则判定树的失败结点数为n+1个
- $ASL_{成功}$=（1*1+2*2+3*4+4*4）/11 = 33/11=3
- $ASL_{失败}$=（3*4+4*8）/12 = 44/12



5.3 折半查找(Cont.)

- 折半查找的判定树高度

$$ASL_{bs}=\sum p_i.c_i$$
$$=1/n\sum j\cdot 2^{j-1}$$
$$=(n+1)/n[log_2(n+1)-n]$$

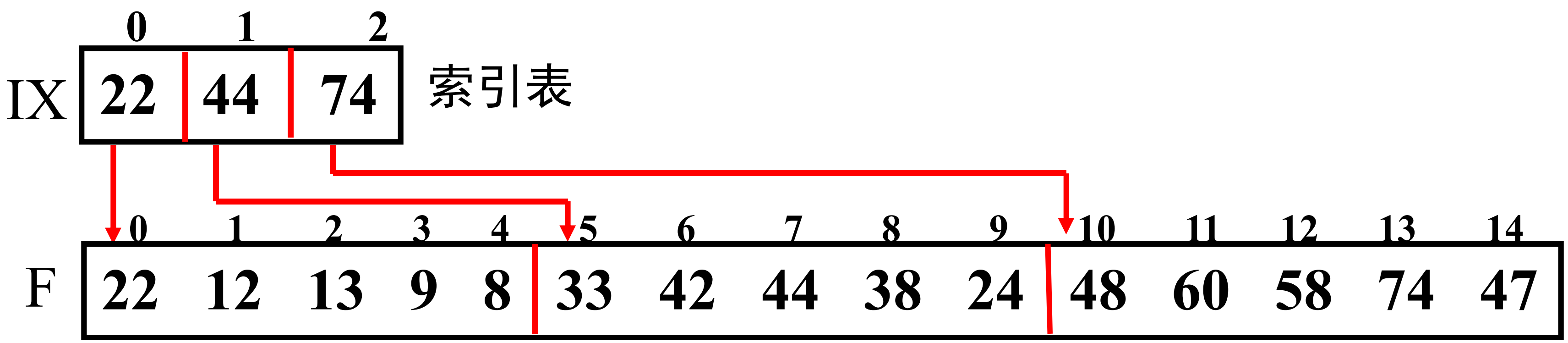


- 当n 很大时，$ASL_{bs} \approx \log_2(n+1)-1$作为查找成功时的平均查找长度。
- 在查找不成功和最坏情况下查找成功所需关键字的比较次数都不超过判定树的高度。
- 判定树中度小于2 的结点只能出现在下面两层上，所以 n 个结点的判定树高度和 n 个结点的完全二叉树的高度相同，即 $\lceil \log_2(n+1) \rceil$。
- 可见，折半查找的最坏性能与平均性能相当接近。

5.4 分块查找：线性查找+折半查找

分块查找的基本思想

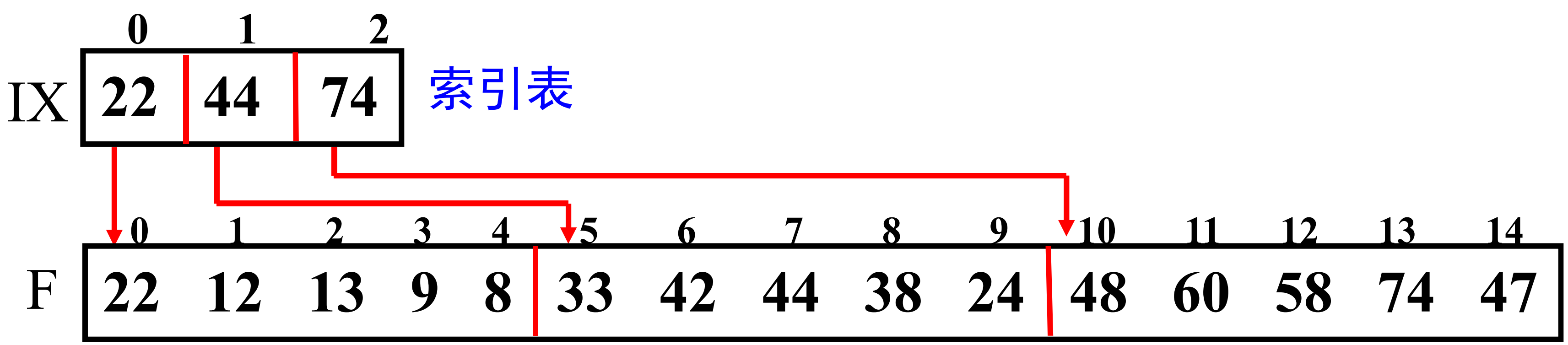
- 均匀分块，块间有序，块内无序：首先将表中的元素均匀地分成若干块，每一块中的元素任意排列，而块间有序；
  - 块间有序：若按从小到大的顺序排列，则第一块中所有元素的关键字都小于第二块中所有元素的关键字，第二块中所有元素的关键字都小于第三块中所有元素的关键字，如此排列。
- 建块索引：建立一个线性表，存放每块中最大（或最小）的关键字，此线性表称为索引表，它是一个有序表.



5.4 分块查找(Cont.)

- 分块查找算法要点：若在带索引的线性表中查找已知关键字为k 的记录，则
  - 首先查找索引表，确定 k 可能出现的块号；
  - 然后到此块中进行进行顺序查找。
- 算法实现

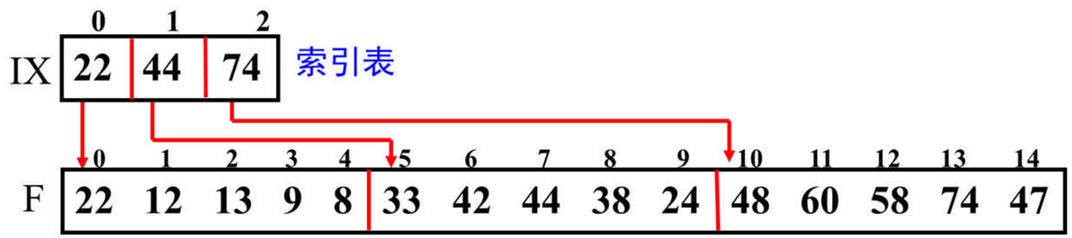
```
typedef   records  LIST[maxsize] ;           // 线性表：主表
typedef   keytype  INDEX[maxblock] ;  // 线性表：索引表
```



5.4 分块查找(Cont.)

分块查找算法实现

```
int   index_search (keytype k, int last,  int blocks, INDEX ix, LIST F, int L)
{   int  i =0, j ;
    while (( k > ix[i])&&( i < blocks))  //查索引表，确定k 所在块i
        i++  ;
    if( i<blocks ) {
        j = i*L;                                  // 第i 块的起始下标
        while(( k != F[j].key )&&( j <= (i+1)*L-1 )&&( j < last ))
            j = j + 1 ;
        if ( k == F[ j ].key )  return   j ;   // 查找成功
     }
     return   –1 ; //查找失败
}
```



5.4 分块查找(Cont.)

分块查找性能分析

- 设长度为n的表分成b块，每块长度为L，则 $b=\lceil n/L \rceil$；表中每个元素的查找概率相等，则每块查找概率为$1/b$，块中每个元素的查找概率为$1/L$。
- 索引表的平均查找长度：

$$ASL_{ix}=\sum_{i=1}^{b} p_j \cdot c_j = 1/\mathrm{b} \sum_{i=1}^{b} i$$

- 块内的平均查找长度：

$$ASL_{blk}=\sum_{j=1}^{L} p_j \cdot c_j = 1/L \sum_{i=1}^{L} j$$

- 分块查找平均长度：

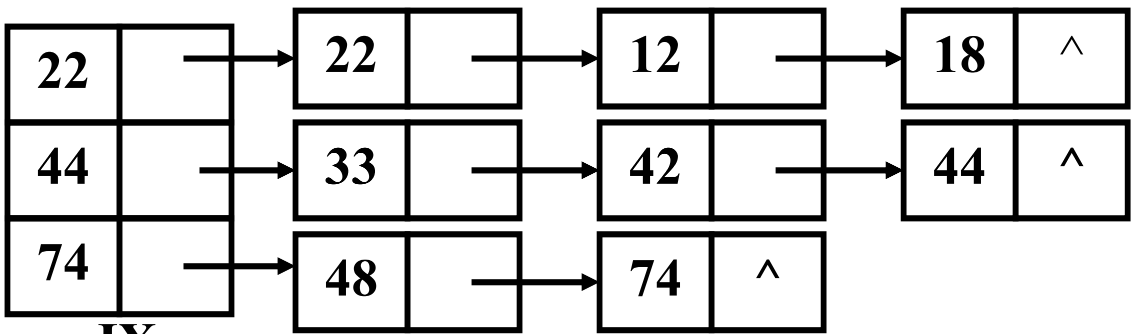
$$ASL(L) = ASL_{ix} + ASL_{blk} = \frac{b+1}{2} + \frac{L+1}{2} = \frac{\frac{n}{L}+L}{2} + 1$$

当 $L=\sqrt{n}$ 时，$ASL(L)=\sqrt{n}+1$（最小值)。

- 索引表为有序表，可采用折半查找

5.4 分块查找(Cont.)

- 分块查找局限性和改进
  - 只适合静态查找；
  - 如何改进：
    - 在索引表中保存各块的下标最大/小值，此时不必均匀分块
    - 各块存放在一个不同向量（一维数组）中；
    - 把同一块中的元素组织成一个链表。
- 动态环境的分块查找
  - 带索引表的链表
  - 算法实现
    - 数据结构定义
    - 三个操作算法：Search, Insert, Delete

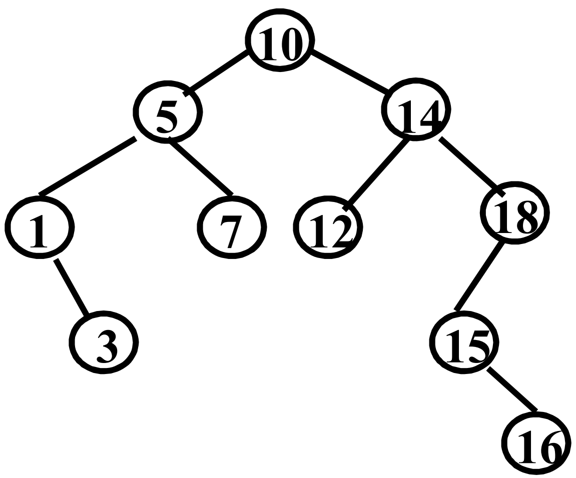


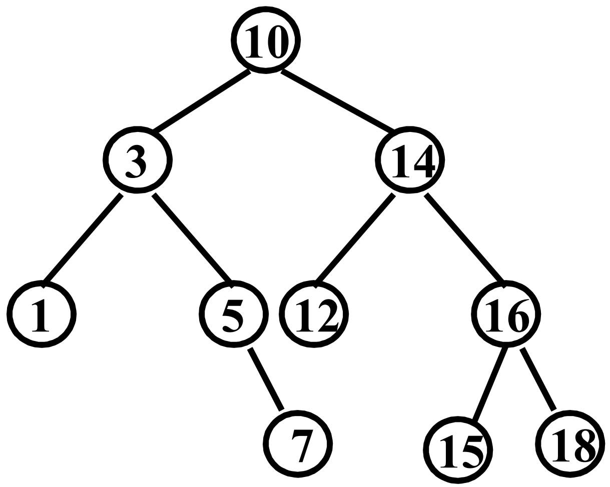
IX

5.5 二叉查找树BST

二叉查找树：二叉搜索树、二叉分类（排序）树

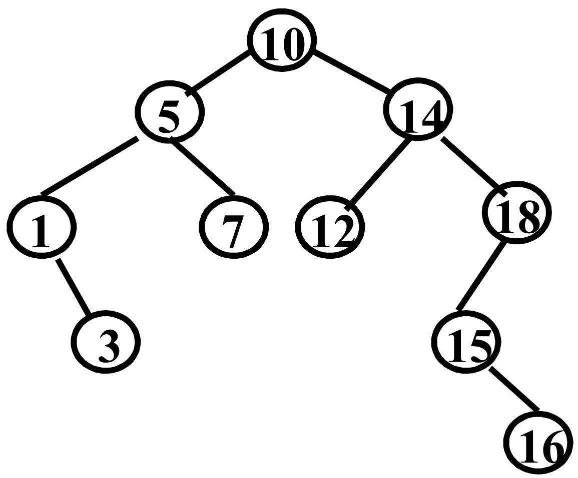
- 二叉查找树或者是空树，或者是满足下列性质的二叉树：
  - 若其左子树不空，则左子树上所有结点的键值都小于根结点的键值；
  - 若其右子树不空，则右子树上所有结点的键值都大于根结点的键值；
  - 其左、右子树本身都是一个二叉查找树。

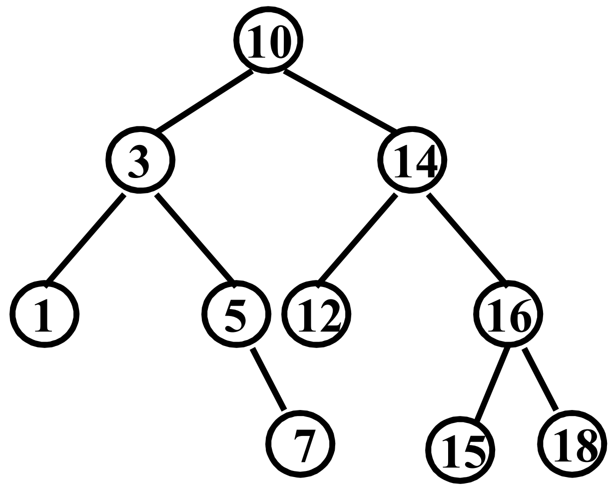




5.5 二叉查找树BST (Cont.)

- 二叉查找树结构特点：
  - 任意一个结点的关键字键值，都大于(小于)其左(右)子树中任意结点的键值，因此各结点键值互不相同
  - 按中序遍历二叉查找树所得的中序序列，是一个递增有序序列，因此，二叉查找树可以把无序序列变为有序序列。
  - 同一数据集合的二叉查找树不唯一，同一个数据集合，可按键值表示成不同二叉查找树，但中序序列相同。





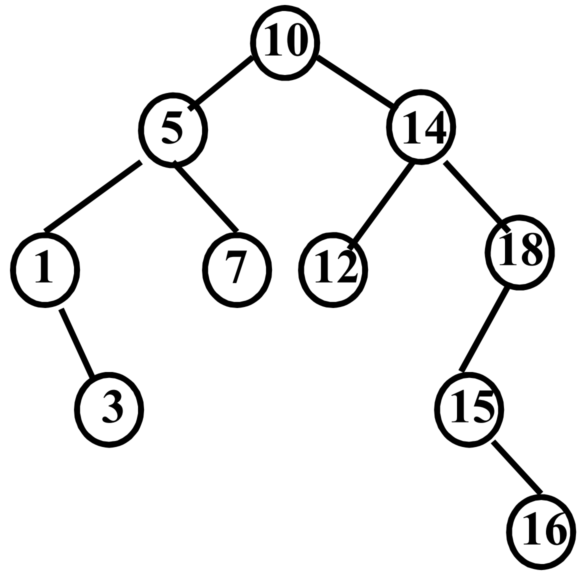
5.5 二叉查找树BST (Cont.)

- 二叉查找树结构特点：
  - 每个结点X的右子树的最左结点Y，称为X的继承结点，其有如下性质：

    (1)在此右子树中，其键值最小，但大于X的键值；

    (2)最多有一个右子树，即没有左子树。
- 二叉查找树的存储结构：

```
typedef struct  celltype  {
        records   data ;
        struct celltype   *lchild,*rchild ;
} BSTNode;
typedef   BSTNode   * BST ;
```

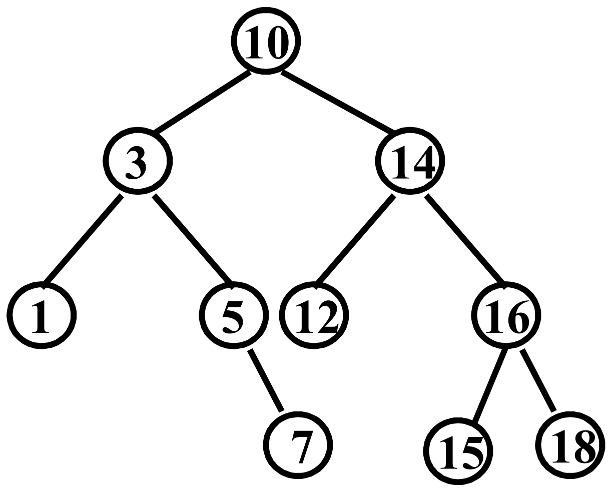


5.5 二叉查找树BST (Cont.)

二叉查找树的查找操作

在F 中查找关键字为k 的记录，步骤如下：

- 若F = NULL，则查找失败；否则，
- k ==F->data.key，则查找成功；否则，
- k < F->data.key，则递归地在F 的左子树查找k；否则
- k > F->data.key，则递归地在F 的右子树查找k。



5.5 二叉查找树BST(Cont.)

二叉查找树查找操作：递归实现

```
BSTNode * SearchBST( keytype  k, BST F )
{   BSTNode * p = F ;
    if ( p == NULL || k == p->data.key ) /* 递归终止条件 */
        return   p;
    if ( k < p->data.key )  /* 查找左子树 */
        return ( SearchBST ( k,  p->lchild ) );
    else                    /* 查找右子树 */
        return ( SearchBST ( k,  p->rchild ) );
}
```

5.5 二叉查找树BST(Cont.)

二叉查找树的查找操作：非递归实现

```
BSTNode *SearchBST ( keytype k, BST F)
{
   BSTNode *p = F;
   while( p!= NULL && k != p->data.key)
   {
       if(k > p->data.key)
            p = p->rchild;
       else
            p = p->lchild;
   }
   return p;
}
```

5.5 二叉查找树BST(Cont.)

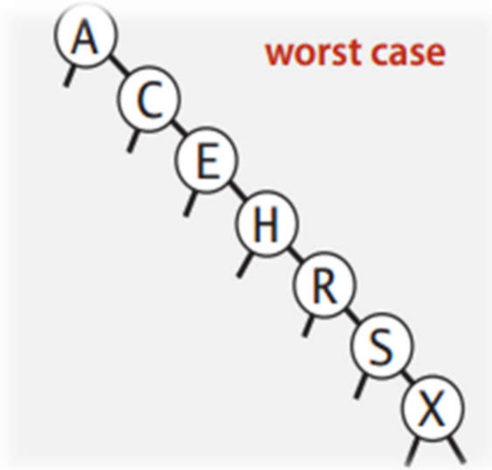
- 二叉查找树的插入操作
  - 若二叉排序树为空树，则新插入的结点为根结点；
  - 否则，新插入的结点必为一个新叶结点。
- 新插入的结点一定是查找不成功时，查找路径上最后一个结点的左儿子或右儿子。

```
void  InsertBST(records R, BST F)
{   if ( F ==NULL ) {
        F = new  BSTNode ;
        F->data = R ;
        F->lchild =  NULL ；
        F->rchild = NULL ;
    }else  if ( R.key < F->data.key )
        InsertBST( R , F->lchild )；
    else  if ( R.key > F->data.key )
        InsertBST( R , F->rchild );
} //若R.key==F->data.key，则返回
```

5.5 二叉查找树BST(Cont.)

二叉查找树的建立

```
BST  CreateBST ( void )
{  BST  F = NULL;  //初始F为空
   keytype  key;
   cin>>key>>其他字段;      //读入一个记录
   while( key ){ //假设key=0是输入结束标志
      InsertBST( R , F );          //插入记录R
      cin>>key>>其他字段 ; //读入下个记录
   }
   return   F;  //返回建立的二叉查找树的根
}
```



注意：

在建立二叉查找树时，若按键值有序顺序输入各记录，则产生退化的二叉查找树：单链表

- 如何防止退化？
  ① 随机输入各结点
  ② 在建立、插入和删除各结点过程中，平衡相关结点的左、右子树高度，AVL树

5.5 二叉查找树BST(Cont.)

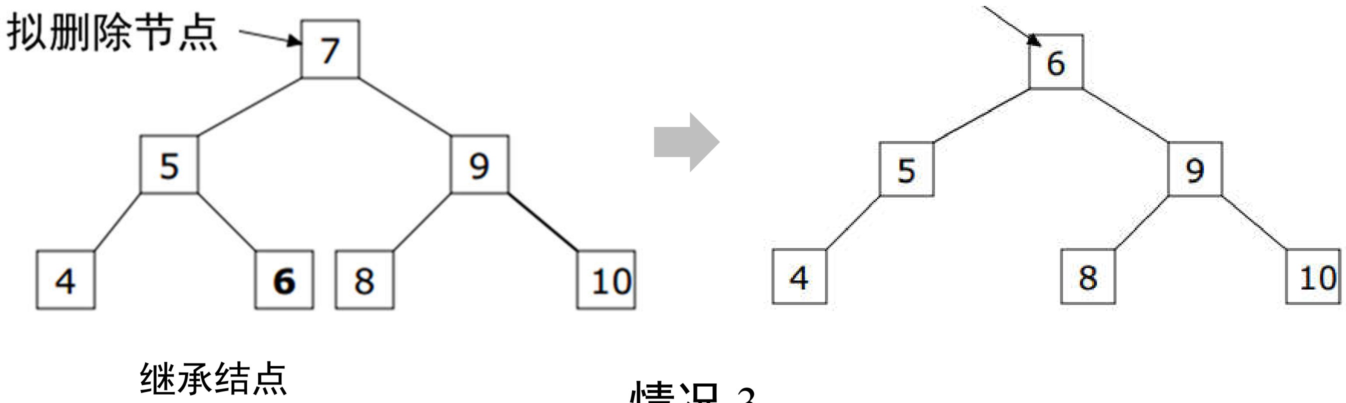
二叉查找树的删除操作

删除某结点，并保持二叉排序树特性，分三种情况处理：

1）如果删除的是叶结点，则直接删除；

2）如果删除的结点只有一株左子树或右子树，则直接继承：将该子树移到被删结点位置；

3）如果删除的结点有两株子树，则用继承结点键值代替被删结点键值，然后删除继承结点（按 1) 或 2) 处理继承结点）。



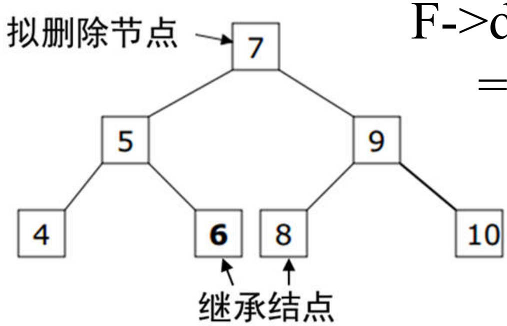
情况 3

5.5 二叉查找树BST(Cont.)

二叉查找树的删除操作：递归实现

```
records  deletemin(BST  &F)
/*删除并返回F的最小结点*/
{  records tmp ; BST  p ;
   if (F->lchild==NULL){//最左最小
     p = F ;
     tmp = F->data ;
     F = F->rchild ; //右链继承
     delete p ;
      return tmp ;
   }
   else//左子树不空，最小在左子树
   return  //在左子树上递归删除
         (deletemin( F->lchild) ;
}
```

```
void  DeleteB( keytype  k, BST &F )
{ if ( F != NULL )
     if ( k < F->data.key )
        DeleteB( k, F->lchild ) ;
     else if  ( k > F->data.key )
        DeleteB( k, F->rchild );
     else //  k==F->data.key
       if ( F->rchild == NULL )
          F = F->lchild;//左链继承
       else if( F->lchild == NULL )
          F = F->rchild;//右链继承
       else //有两棵子树
           F->data
                =deletemin(F->rchild);}
```



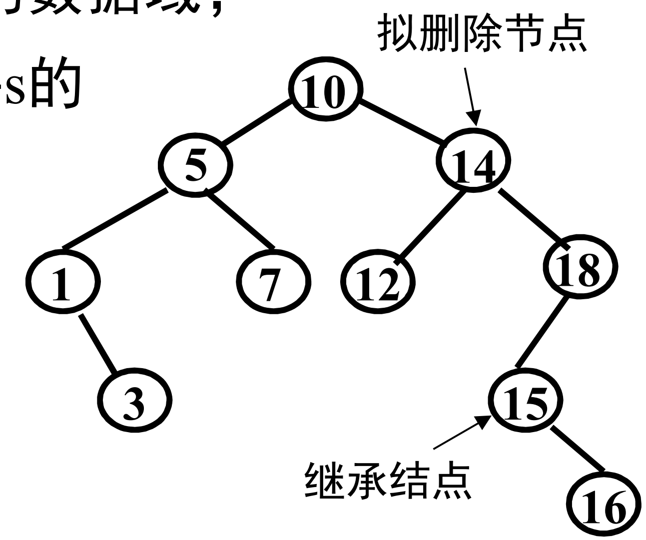
5.5 二叉查找树BST(Cont.)

二叉查找树删除操作：非递归实现步骤

1. 若结点p是叶子，则直接删除结点p；
2. 若结点p只有左子树，则只需重接p的左子树；

   若结点p只有右子树，则只需重接p的右子树；
3. 若结点p的左右子树均不空，则
   - 3.1 查找结点p的右子树上的最左下结点s及其双亲结点par；
   - 3.2 将结点s数据域替换到被删结点p的数据域；
   - 3.3 若结点p的右孩子无左子树，则将s的右子树接到par的右子树上；

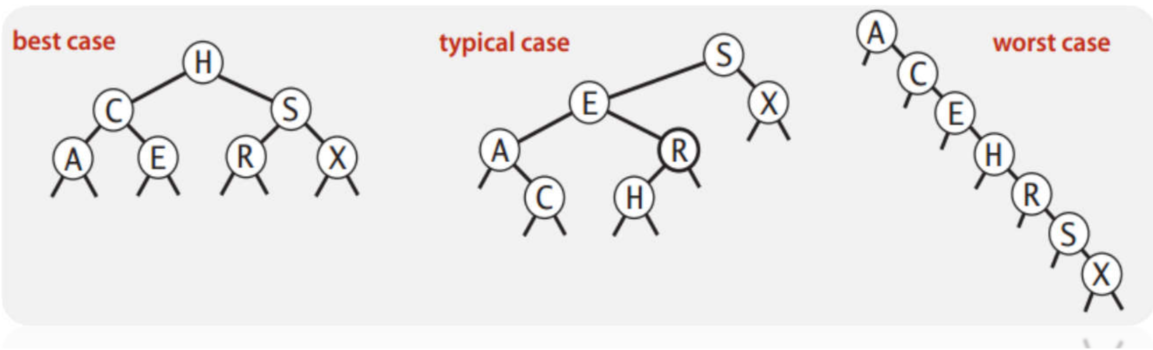
     否则，将s的右子树接到结点par的左子树上；//例如结点14
   - 3.4 删除结点s；



5.5 二叉查找树BST(Cont.)

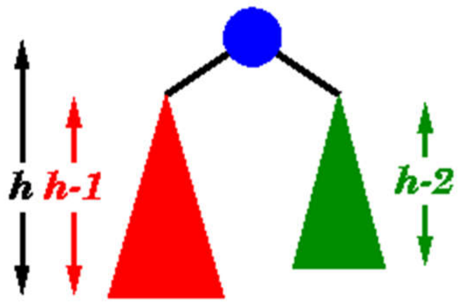
二叉查找树的查找性能

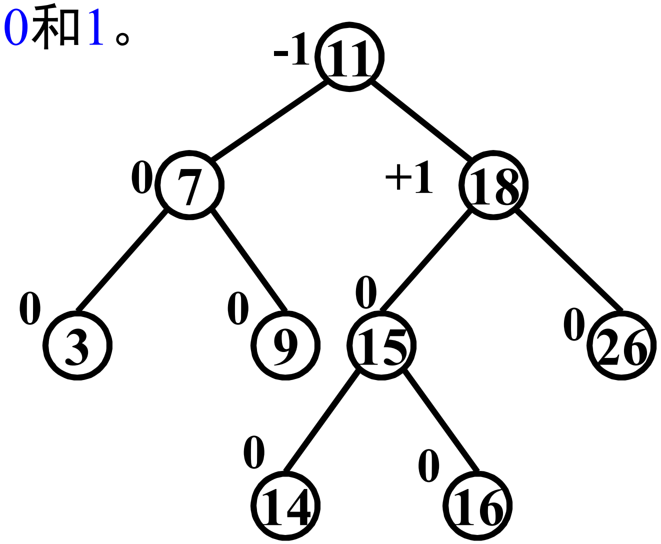
- 二叉排序树的查找性能取决于二叉排序树的形态，在$O(\log_2 n)$和$O(n)$之间。
- 在最坏情况下，二叉查找树是把有序表的 n 个结点依次插入而生成的，所得的二叉查找树退化为一株高度为 n 的单支树，它的平均查找长度和单链表顺序查找相同 (n+1)/2。
- 在最好情况下，二叉查找树的形态比较均匀，最终得到的形态与折半查找的判定树相似，此时平均查找长度为$\log_2 n$。
- 二叉查找树平均高度为$O(\log_2 n)$。因此，平均情况下，三种操作的平均时间复杂性为$O(\log_2 n)$。
- 就平均性能而言，二叉查找树查找与折半（二分）查找差不多；
- 就维护表的有序性而言，二叉查找树更有效。



5.6 AVL树

- AVL树(Adelson-Velskii & Landis, 1962)
  (Balanced Binary Tree or Height-Balanced Tree)
  AVL树或者是空二叉树，或者是具有如下性质的BST：
  - 根结点的左、右子树高度之差的绝对值不超过1；
  - 且根结点左子树和右子树仍然是AVL树。
- 结点的平衡因子BF（Balanced Factor）
  - 一个结点的左子树与右子树的高度之差。
  - AVL树中的任意结点的BF只可能是-1，0和1。
  - AVL树的ASL可保持在O(log *n*)
- AVL树的查找操作
  - 与BST的相同

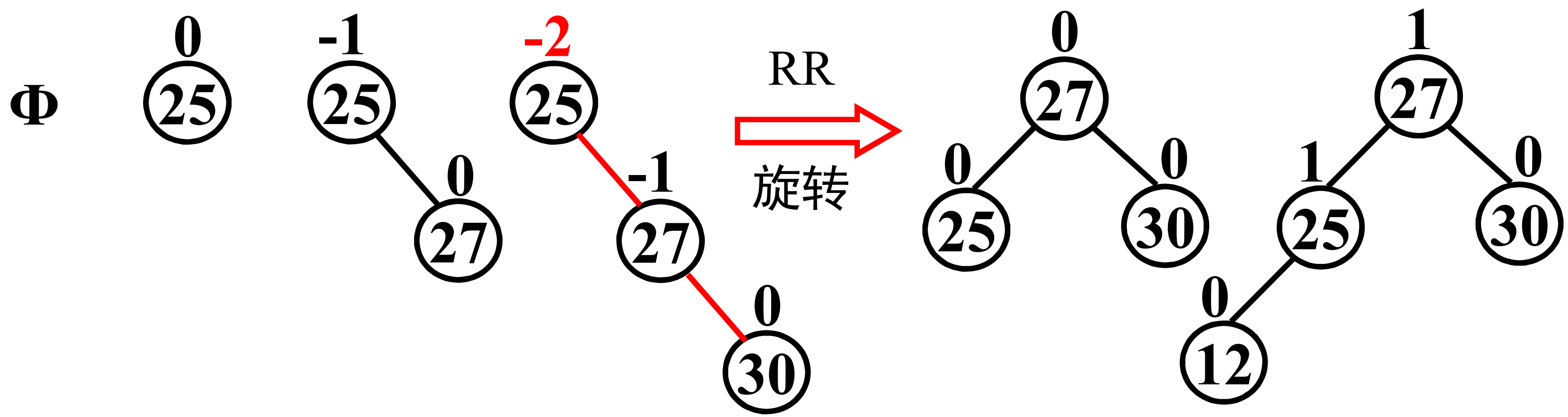




5.6 AVL树(Cont.)

AVL树的平衡化处理

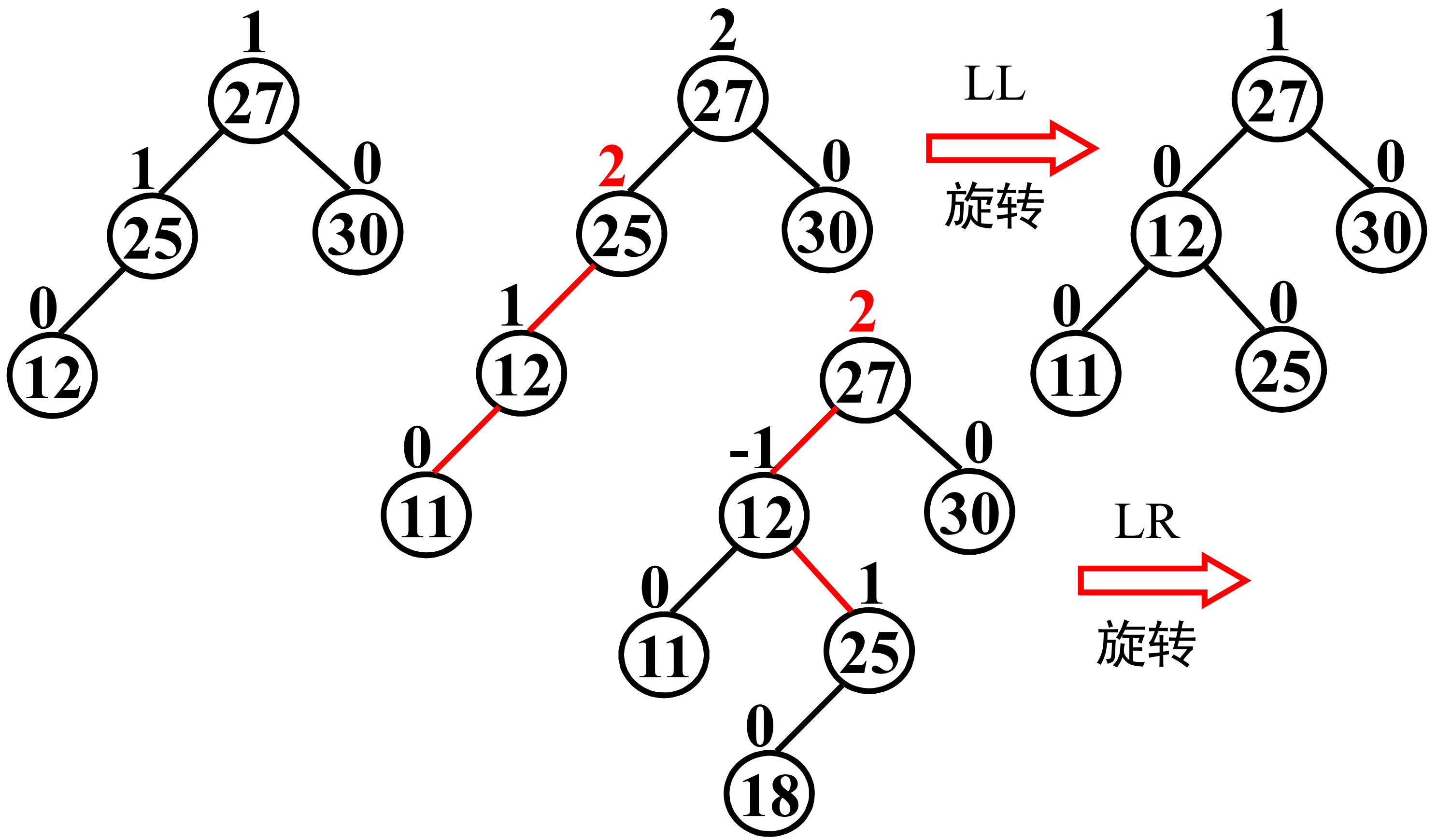
- 向AVL树插入结点可能造成不平衡，此时要调整树结构，使之重新达到平衡
- 任何初始序列构成的二叉树都应是AVL树，即与输入序列的次序无关。
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



5.6 AVL树(Cont.)

AVL树的平衡化处理

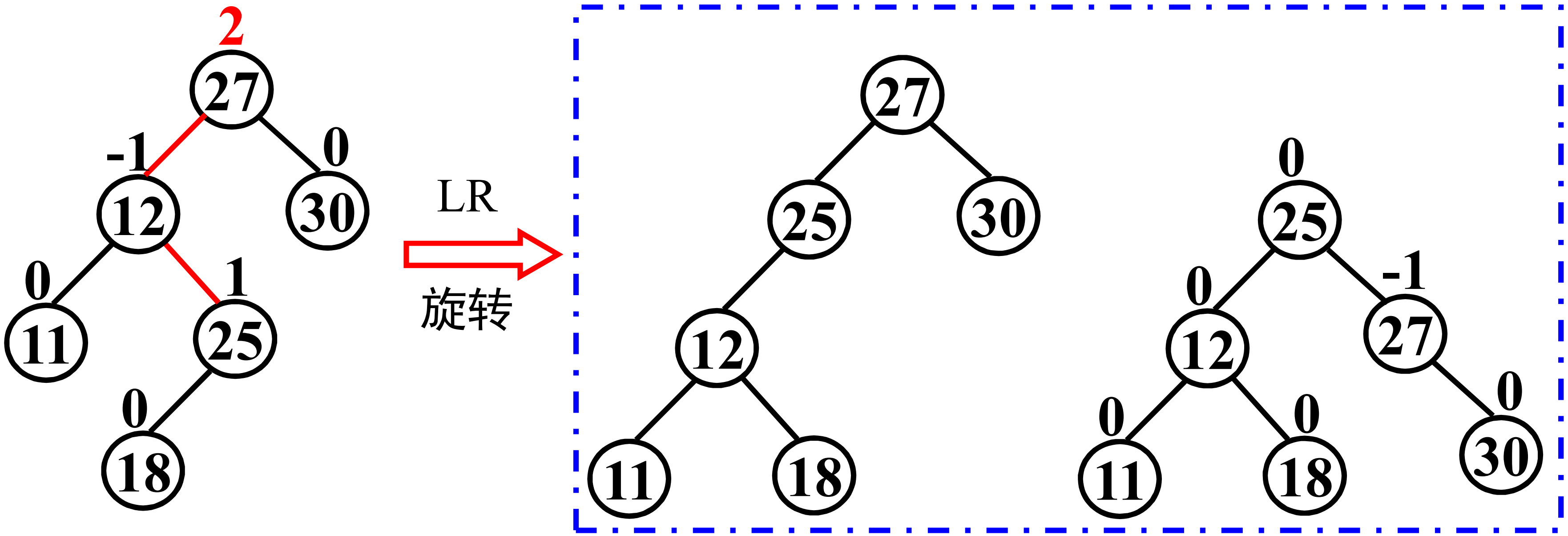
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



5.6 AVL树(Cont.)

AVL树的平衡化处理

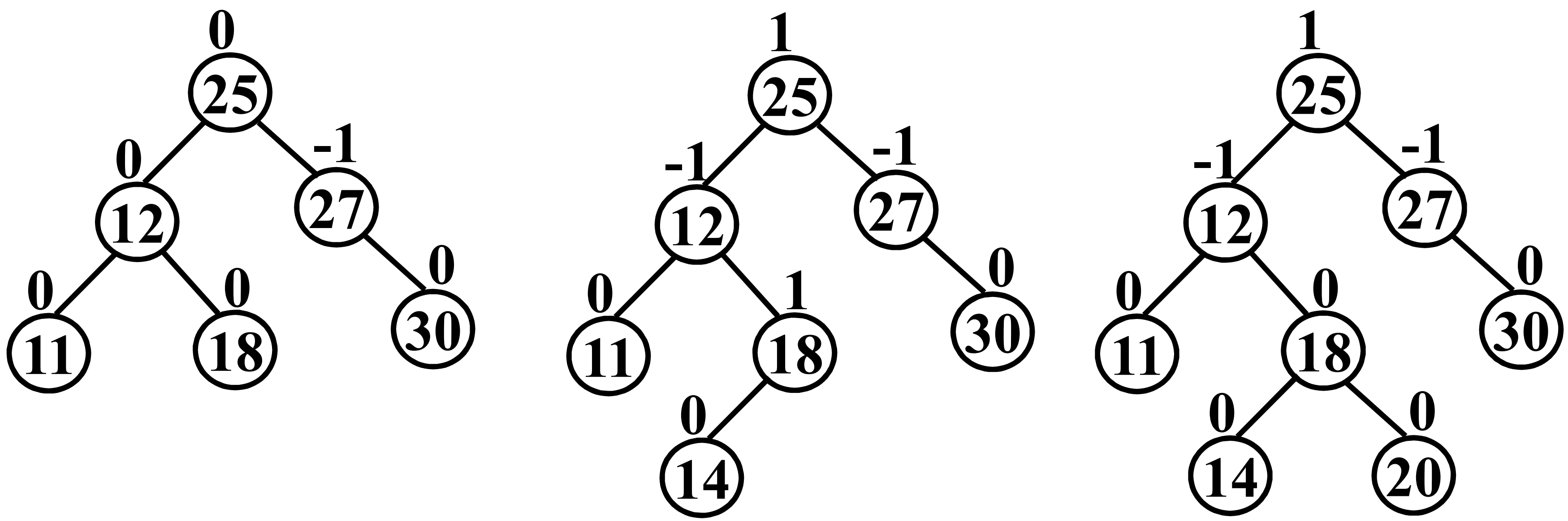
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



5.6 AVL树(Cont.)

AVL树的平衡化处理

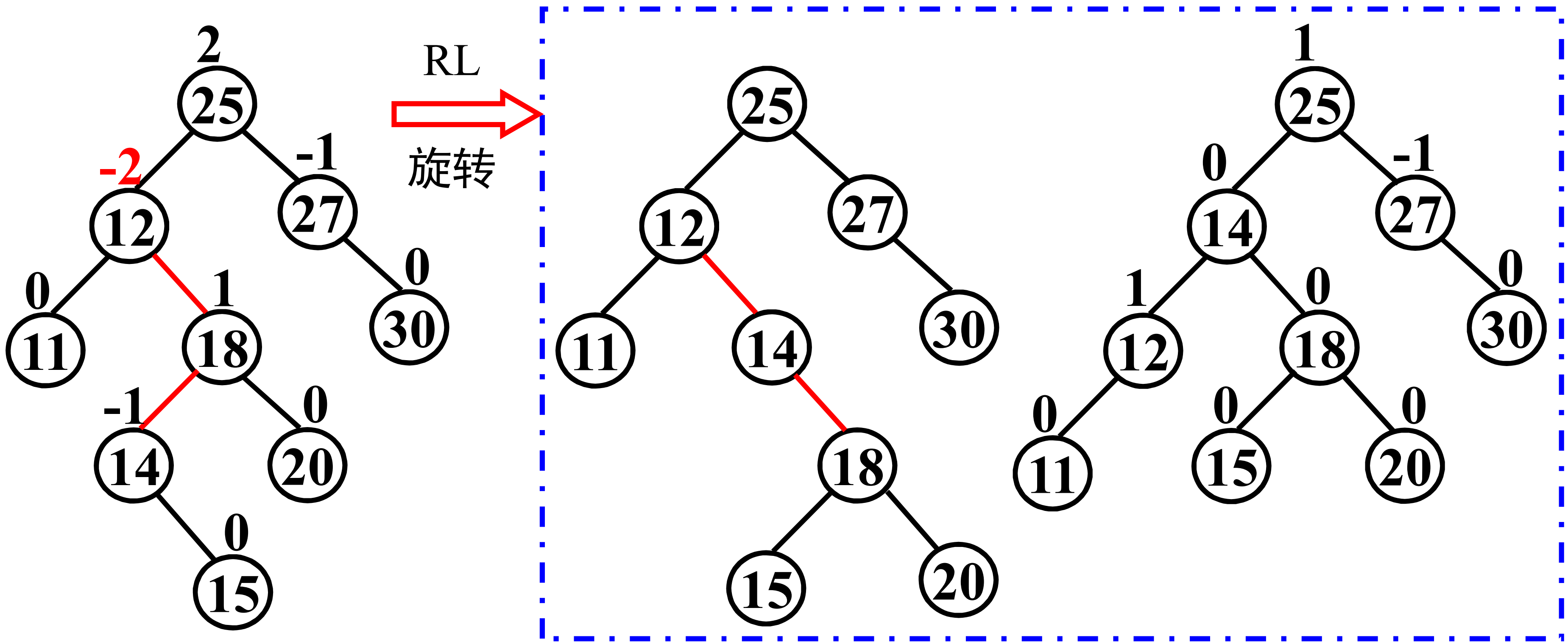
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



5.6 AVL树(Cont.)

AVL树的平衡化处理

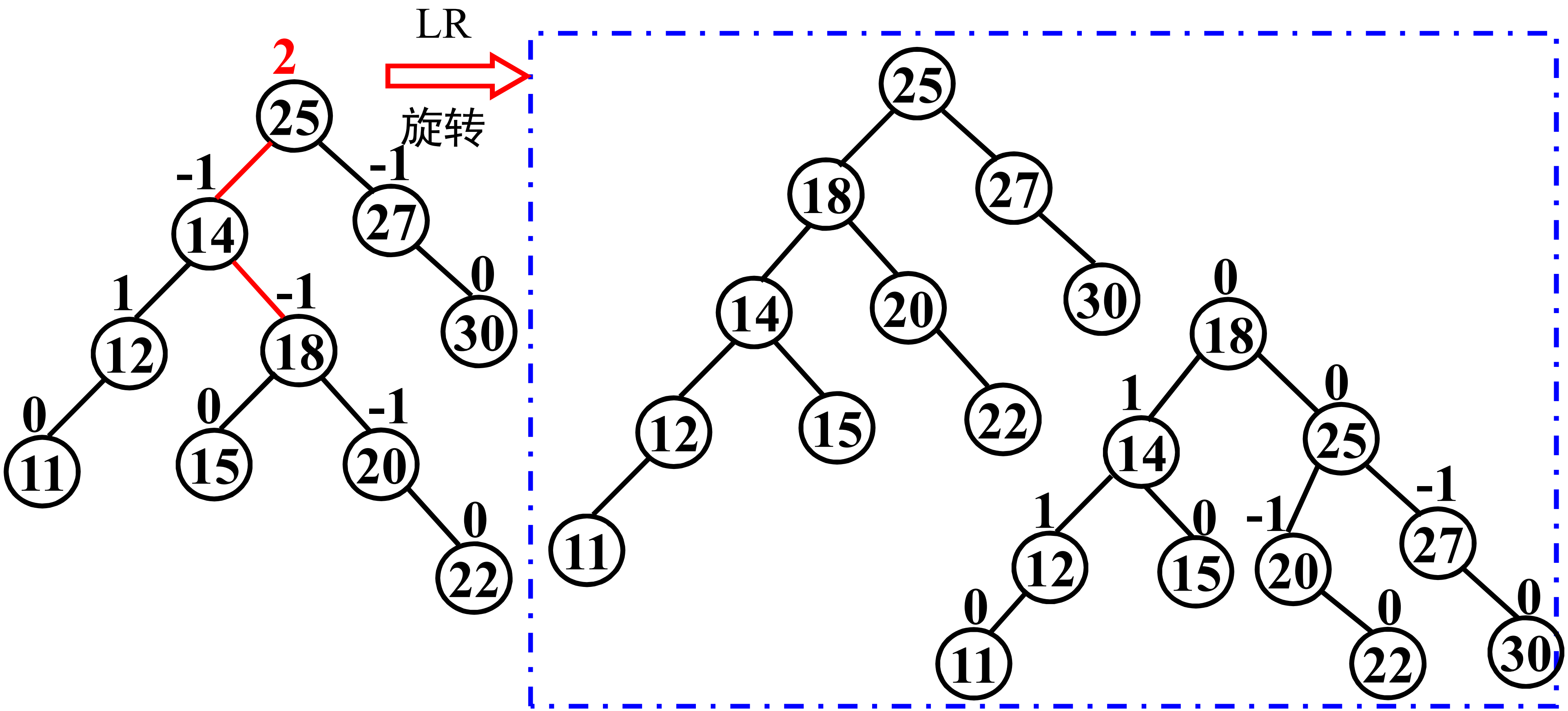
- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



5.6 AVL树(Cont.)

AVL树的平衡化处理

- 示例：假设25，27，30，12，11，18，14，20，15，22是一关键字序列，并以上述顺序建立AVL树。



5.6 AVL树(Cont.)

AVL树的平衡化处理

- 在一棵AVL树上插入结点可能会破坏树的平衡性，需要平衡化处理恢复平衡，且保持BST的结构性质。
- 若用Y表示新插入的结点，A表示离新插入结点Y最近的，且平衡因子变为±2的祖先结点。
- 有4种旋转进行平衡化处理:

  ① LL型：新结点Y 被插入到 A 的左子树的左子树上（顺）

  ② RR型：新结点Y 被插入到 A 的右子树的右子树上（逆）

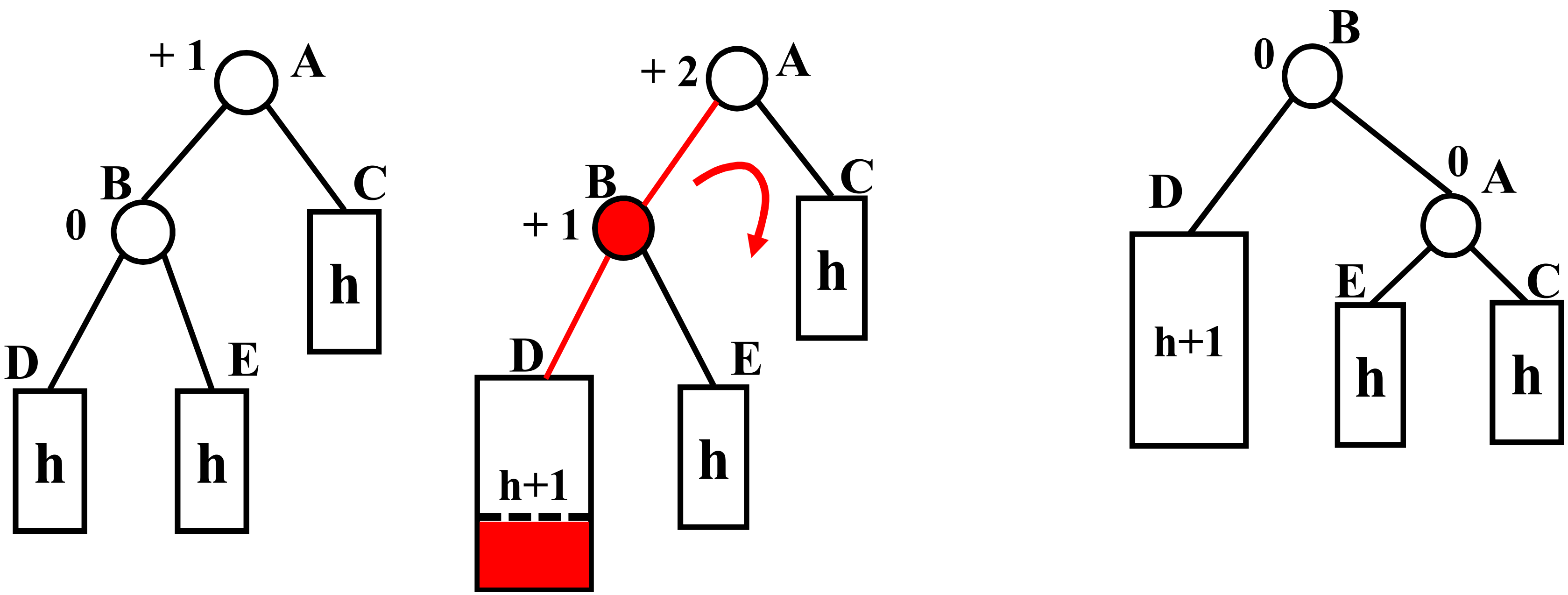
  ③ LR型：新结点Y 被插入到 A 的左子树的右子树上（逆、顺）

  ④ RL型：新结点Y 被插入到 A 的右子树的左子树上（顺、逆）

5.6 AVL树(Cont.)

AVL树的平衡化处理

- LL型：新结点Y 被插入到 A 的左子树的左子树上（顺）

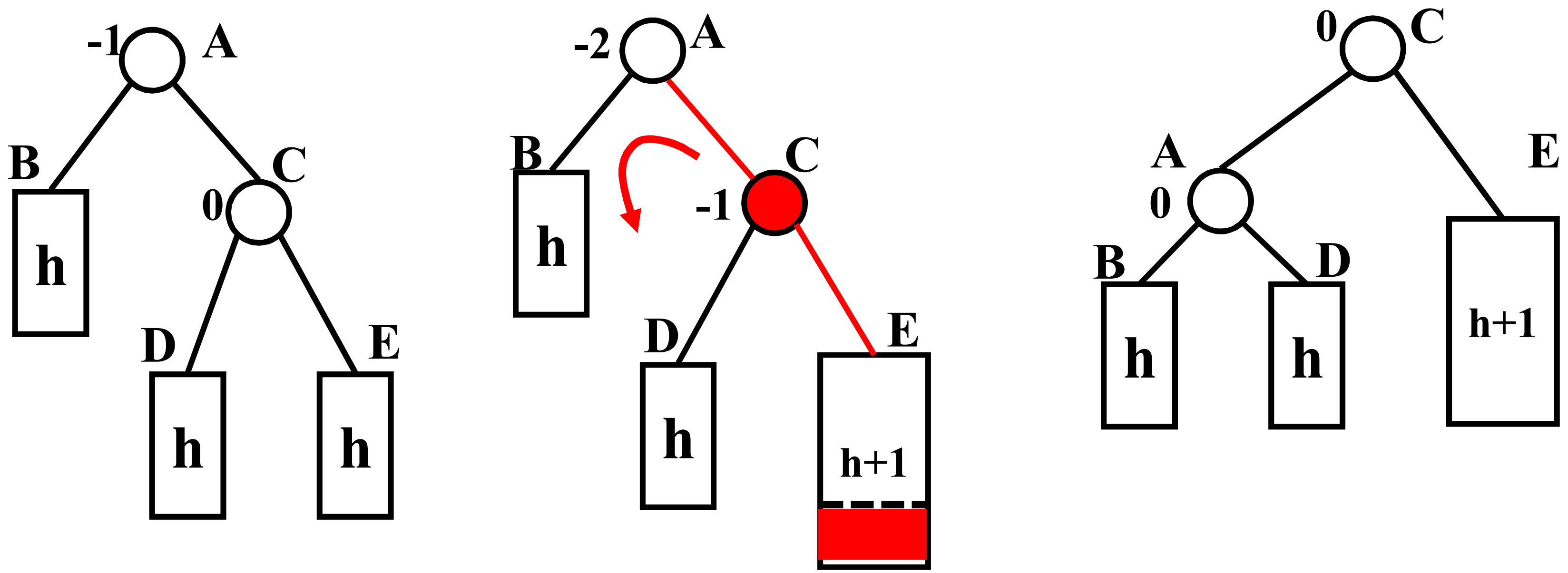


(a)AVL树　(b) D子树中插入结点　(c) 右向旋转后的AVL树

5.6 AVL树(Cont.)

AVL树的平衡化处理

- RR型：新结点Y 被插入到 A 的右子树的右子树上（逆）

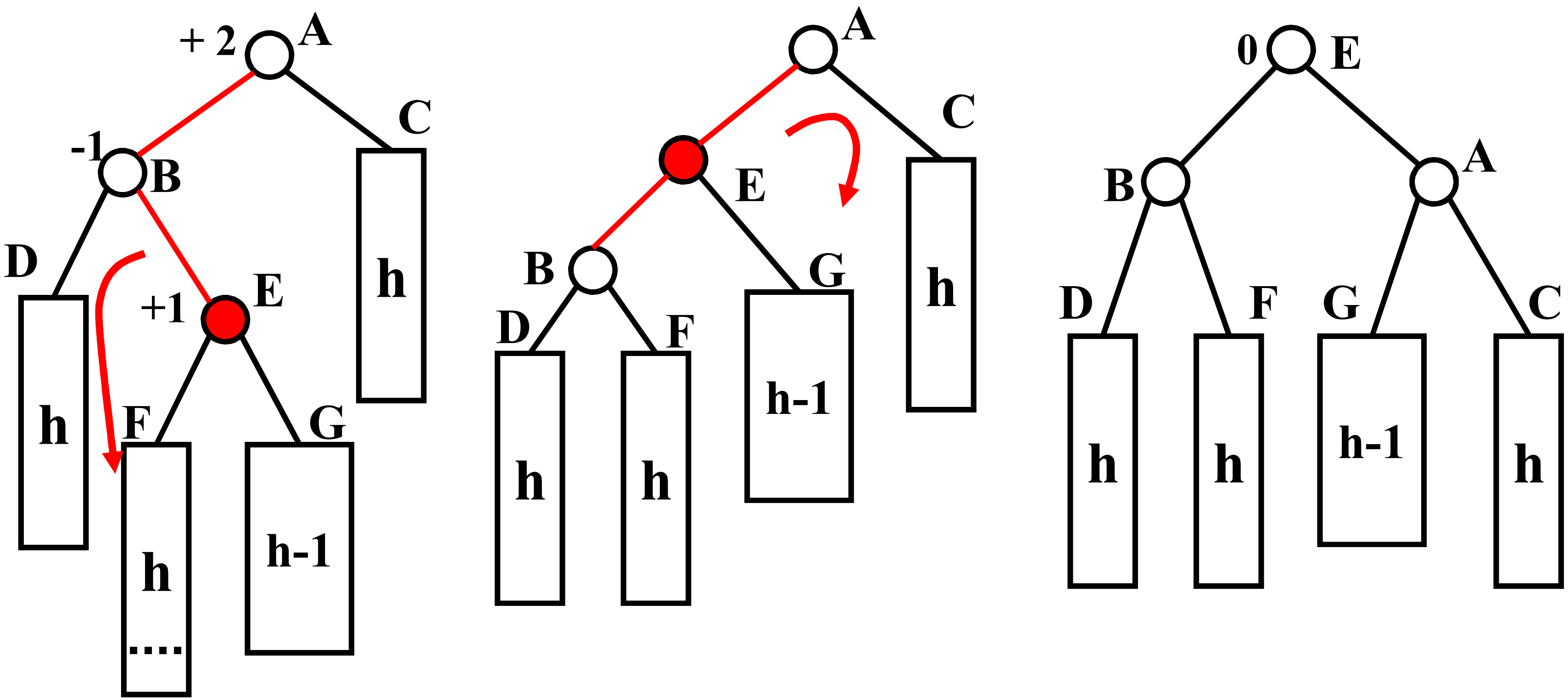


(a)AVL树　　(b)E子树中插入结点　　(c)左向旋转后的AVL树

5.6 AVL树(Cont.)

AVL树的平衡化处理

- LR型：新结点Y 被插入到 A 的左子树的右子树上（逆，顺）



(a)F子树插入结点高度变为h

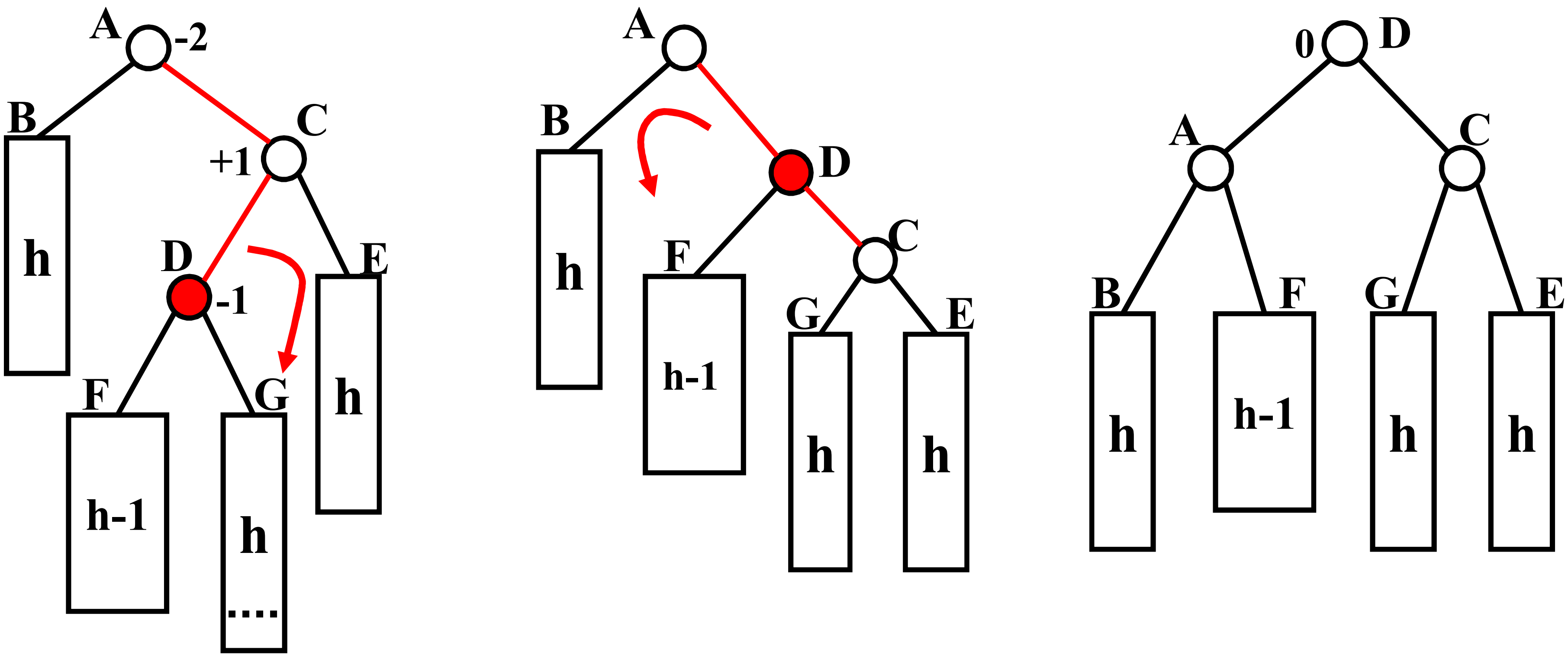
(b)绕E，将B逆时针转后

(c)绕E，将A顺时针转后

5.6 AVL树(Cont.)

AVL树的平衡化处理

- RL型：新结点Y 被插入到 A 的右子树的左子树上(顺, 逆)



(a)G子树插入结点高度变为h

(b)绕D，C顺时针转之后

(c)绕D，A逆时针转之后

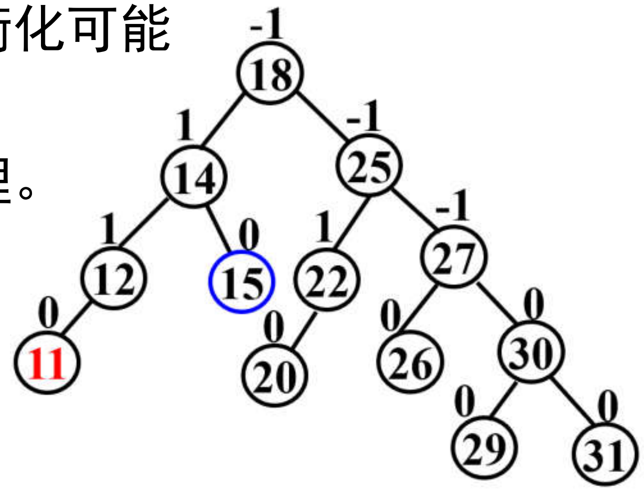
5.6 AVL树(Cont.)

- AVL树的插入与建立操作
  - 对于一组关键字的输入序列，从空开始不断地插入结点，最后构成AVL树
  - 每插入一个结点后就应判断从该结点到根的路径上有无结点发生不平衡
  - 如有不平衡问题，利用旋转方法进行树调整，使之平衡化
  - 建AVL树过程是不断插入结点和必要时进行平衡化的过程
- 过程：
  - 插入>>判断平衡>>判断失衡类型>>旋转调整

5.6 AVL树(Cont.)

AVL树的删除操作

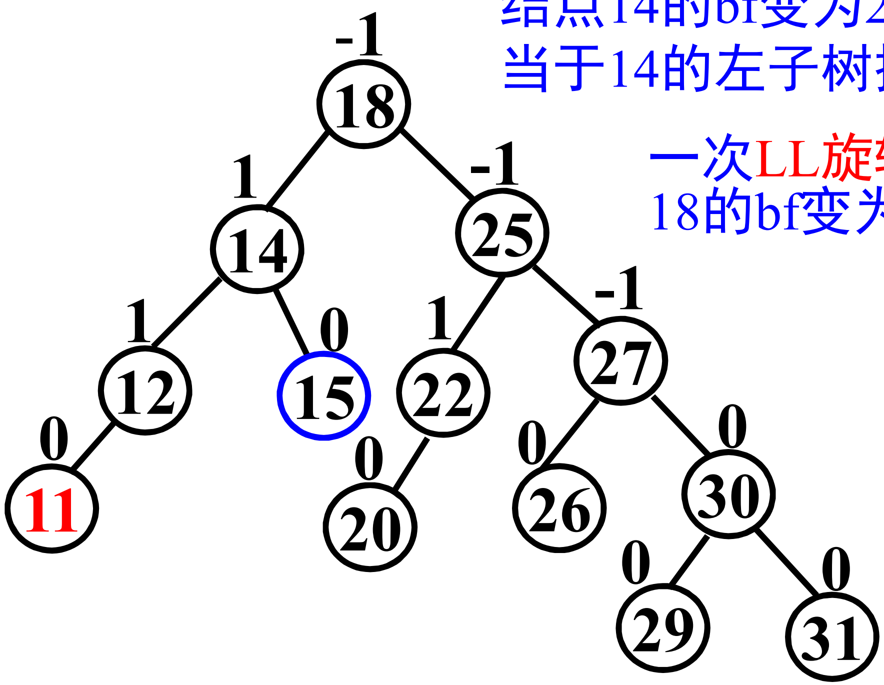
- 删除与插入操作是对称的（镜像，互逆的）：
  - 删除右子树结点导致失衡时，相当于在左子树插入导致失衡，即LL或LR；
  - 删除左子树结点导致失衡时，相当于在右子树插入导致失衡，即RR或RL；
- 但是，删除操作可能需要多次平衡化处理
  - 平衡化不会增加子树的高度，但可能会减少子树的高度。
  - 在有可能使树减低的删除操作中，平衡化可能会带来祖先结点的不平衡。
  - 故，删除操作可能需要多次平衡化处理。



5.6 AVL树(Cont.)

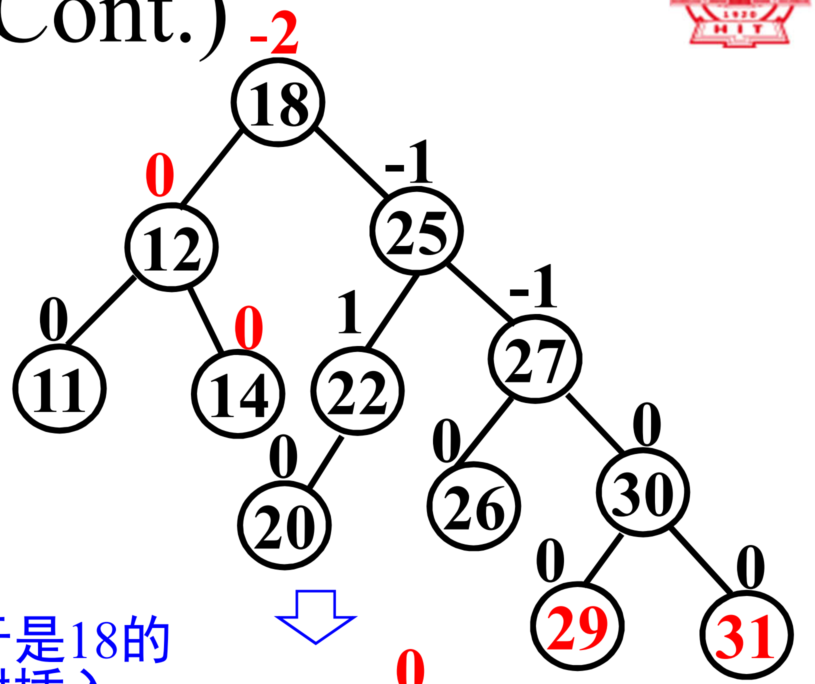
AVL树的删除操作

- 示例：删除15，2次旋转到根

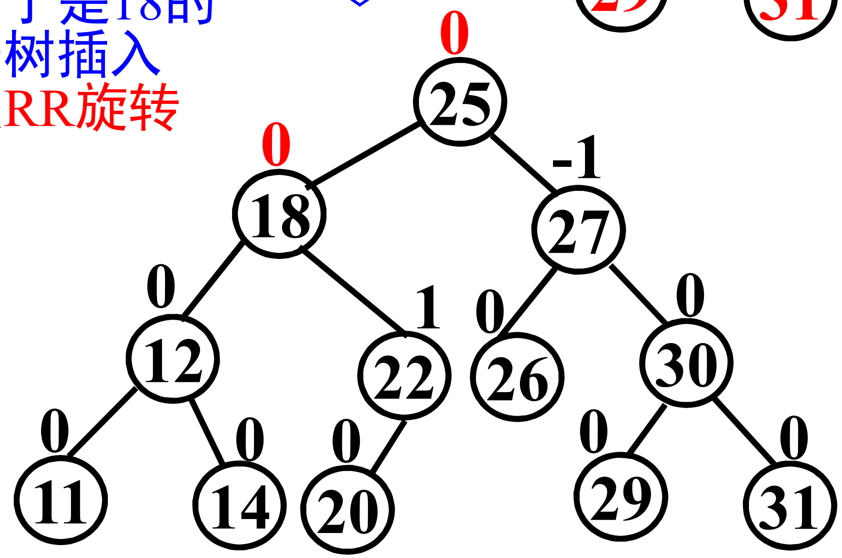


结点14的bf变为2相
当于14的左子树插入

一次LL旋转
18的bf变为-2



相当于是18的
右子树插入
一次RR旋转



5.6 AVL树(Cont.)

AVL树的性能分析

- 令N_h是高为h的AVL树中结点个数的最小值，在最稀疏情况下，这棵AVL树的一棵子树高度为h–1，而另一棵子树高度为h–2，这两棵子树也都是AVL树。

  因此， $N_h = N_{h-1} + N_{h-2} + 1$，其中，$N_0=0$，$N_1=1$，$N_2=2$，$N_3=4$
- 可以发现， N_h的递归定义与Fibonacci数的定义$F_n = F_{n-1} + F_{n-2}$（其中，$F_0 = 0$，$F_1 = 1$）相似
- 用数学归纳法证得：$N_h = F_{h+2} - 1$ （$h \geq 0$）

  $F_h \approx \varphi^h/\sqrt{5}$，其中 $\varphi = (1+\sqrt{5})/2$，所以， $N_h \approx \varphi^{h+2}/\sqrt{5}-1$

  所以，一棵包含n个结点的AVL树，其高度h至多为$\log_\varphi(n+1)-2$
- 因此，对于包含n个结点的AVL树，其最坏情况下的查找、插入和删除操作时间复杂度均为$O(\log n)$

5.7 B-树和B+树

- 在AVL树在结点高度上采用相对平衡的策略，使其平均性能接近于BST的最好情况性能。
- AVL树能力受限：每个结点只有一个关键字，结点宽度（结点包含的记录数量）太小。
- 如果保持查找树在高度上绝对平衡，而允许查找树结点的子树个数（分支数）在一定范围内变化（增加结点宽度），能否获得很好的查找性能？
- 已有许多在高度上保持绝对平衡，而在宽度上保持相对平衡的查找结构。
  - 这些查找结构不再是二叉结构，而是*m*-路查找树（*m-way search tree*）
  - 例如，B-树及其各种变形结构，以子树保持等高为其基本性质。
  - 实际应用广泛

5.7 B-树和B+树(Cont.)

m-路(叉)查找树：

一棵m-路查找树或者是一棵空树, 或者是满足如下性质的树：

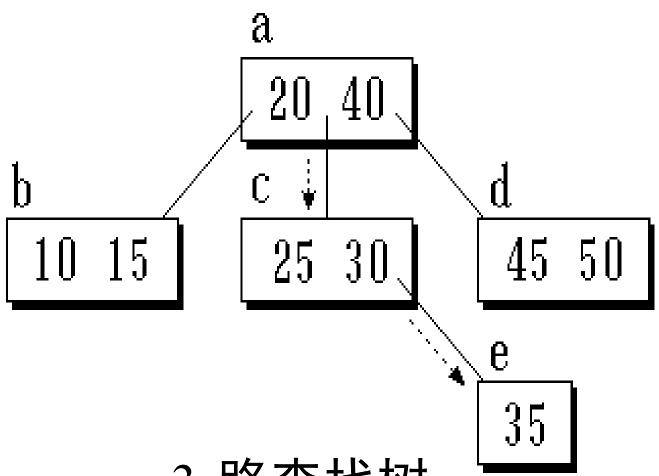
- 根结点最多有 m 棵子树, 结点结构包含如下信息：

  $n, A_0, (K_1, A_1), (K_2, A_2), \ldots, (K_i, A_i), \ldots, (K_n, A_n)$

  其中，A_i 是指向子树的指针，$0 \le i \le n < m$

  K_i 是键值，$1 \le i \le n < m$， $K_i < K_{i+1}$, $1 \le i < n$

- 子树 A_i 中所有的键值都小于K_{i+1}而大于K_i，$0 < i < n$。
- 子树 A_0 中所有的键值都小于K_0；
- 子树 A_n 中所有的键值都小于 K_n。
- 每棵子树 A_i 也是 m -路查找树，$0 \le i \le n$。



3-路查找树

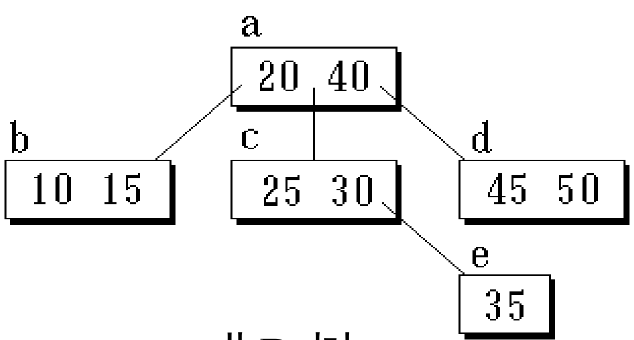
5.7 B-树和B+树(Cont.)

B-树：

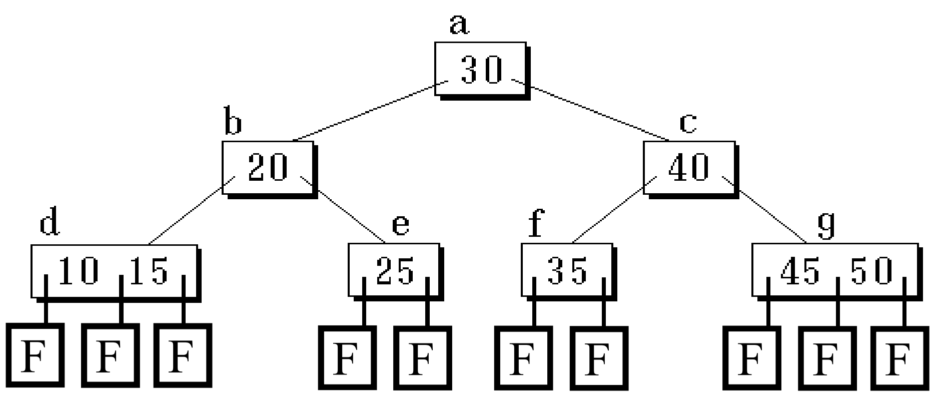
一棵 m 阶B-树是一棵 m-路查找树，或是空树，或满足下列性质：

- 树中每个结点至多有m棵子树；
- 根结点至少有 2 棵子树(2~m)；
- 除根结点和失败结点外，所有结点至少有$\lceil m/2 \rceil$棵子树($\lceil m/2 \rceil$~m)；
- 所有终端结点和失败结点都位于同一层。

B的阶	非根结点子树个数	
	最少	最多
3	2	3
4	2	4
5	3	5
6	3	6
7	4	7
…	…	…
m	$\lceil m/2 \rceil$	m



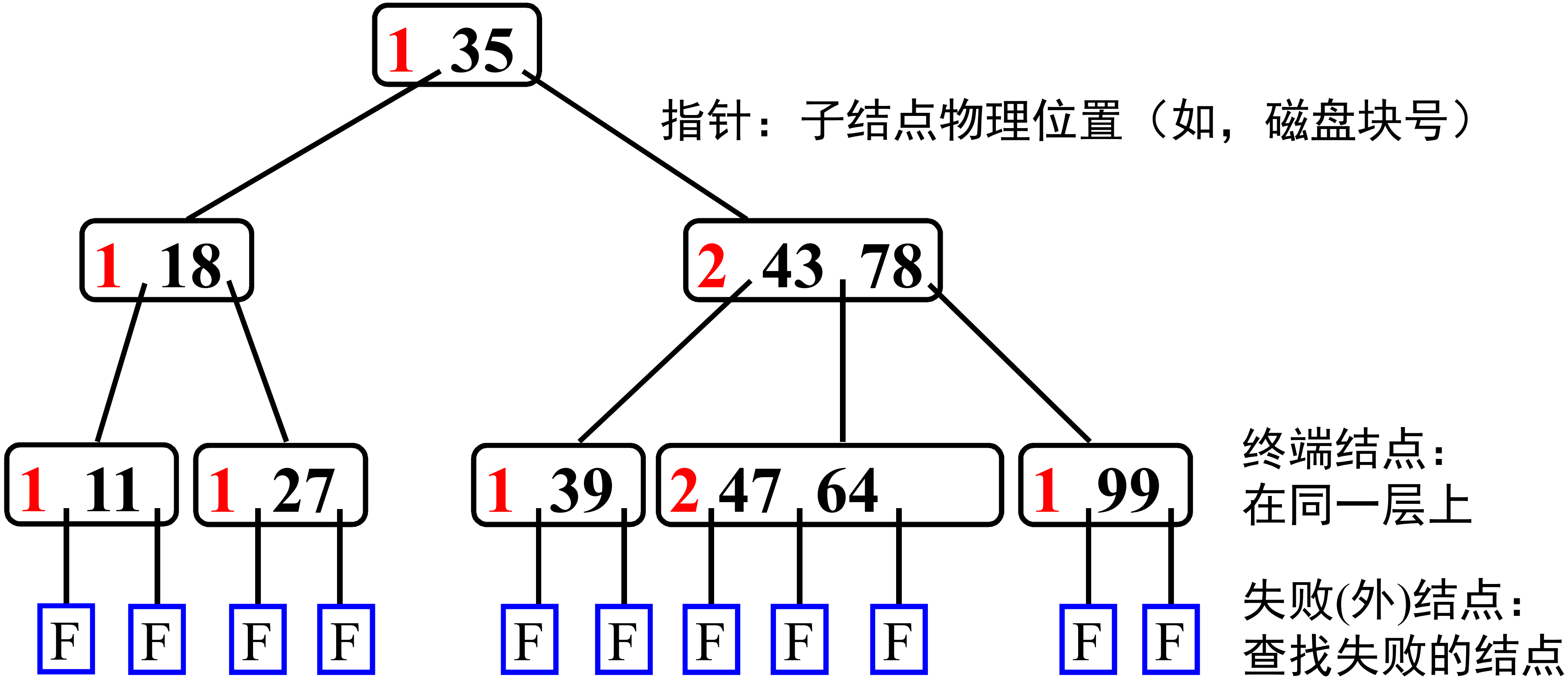
非B-树



3阶B-树

5.7 B-树和B+树(Cont.)

B-树示例



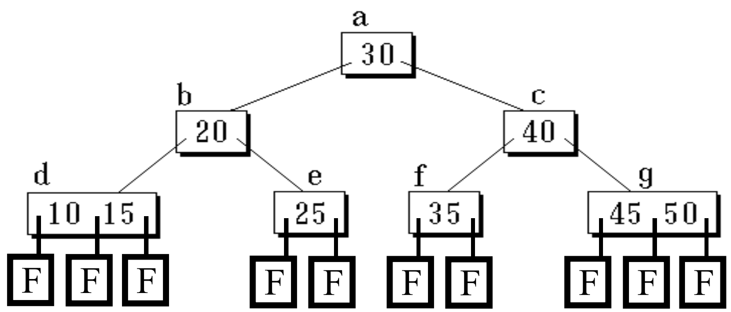
5.7 B-树和B+树(Cont.)

B-树高度h与关键字个数 N 之间的关系

- 设 m 阶B-树的高为h，失败结点位于第 h+1层，这棵B-树的关键字个数 N 最小能达到多少？
- 由B-树的定义可知：
  - 1层　1 个结点
  - 2层　至少 2 个结点
  - 3层　至少 $2\lceil m/2 \rceil$个结点
  - 4层　至少 $2\lceil m/2 \rceil^2$个结点
  - 以此类推，……
  - h 层　至少有$2\lceil m/2 \rceil^{h-2}$个结点
  - 失败结点在$h+1$ 层

    至少失败结点数：$2\lceil m/2 \rceil^{h-1}$ = 关键字个数 N+1
  - 高为h的B-树最多的结点数和最多关键字数分别为：

$$\sum_{i=1}^{h} m^{i-1} = (m^h - 1)/(m - 1) \text{ 和 } m^h - 1$$



3阶B-树

5.7 B-树和B+树(Cont.)

- 设 m阶B-树中关键字为 N 个, 则失败结点数为 $N+1$，即

  $N+1$ = 失败结点数

  = 位于第 h+1 层的结点数 $\geq 2\,\lceil m/2 \rceil^{h-1}$

  $N \geq 2\,\lceil m/2 \rceil^{h-1}-1$ 为最少关键字的个数。（N最多：m^h-1）

- 反之，如果在一棵 m 阶B-树中有 N 个关键字，则

  $h \leq 1 + \log_{\lceil m/2 \rceil}[(N+1)/2]$ (最大高度)

- 例，若B-树的阶数 $m = 199$，关键字总数 $N = 1999999$，则B-树的高度 h （不超过 ）：

  $h \leq 1 + \log_{\lceil 199/2 \rceil}[(1999999+1)/2] = \log_{100}1000000 + 1 = 4$

- 例，若B-树阶数 $m = 3$，高度$h = 4$，则关键字总数至少为

  $N \geq 2\,\lceil m/2 \rceil^{h-1}-1 = 2\,\lceil 3/2 \rceil^{4-1} - 1 = 15$

5.7 B-树和B+树(Cont.)

B-树的查找操作

- B-树的查找过程是一个顺指针（纵向）查找结点和在结点中（横向）查找（可采用折半查找）交替进行的过程。
- 因此，B-树查找时间直接相关:
  - B-树的高度*h*(纵向查找)
  - B-树的阶数*m*(横向查找)
  - 必须对二者进行权衡
- 在B-树上进行查找的时间开销：
  - 查找成功所需的时间取决于键值所在的层次；
  - 查找不成功所需的时间取决于树的高度。

5.7 B-树和B+树(Cont.)

B-树的插入与建立操作

- B-树从空树建起，逐个插入关键字而生成。
- 在m阶B-树中，每个非失败结点的关键字个数都在

  [⌈m/2⌉-1, m-1]之间（根结点：[1, m-1]）

B-树插入操作：

- 首先执行查找操作以确定可以插入新关键字的终端结点 p；
- 若在关键字插入后，结点中的关键字个数没有超出上界 m-1, 直接插入s；否则(=m)，结点需要“分裂”。
- 分裂结点的原则：设插入前，终端结点 p 中已经有 m-1 个关键字，当再插入一个关键字后结点的状态为：

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

其中 $K_i < K_{i+1}, 1 \le i < m$

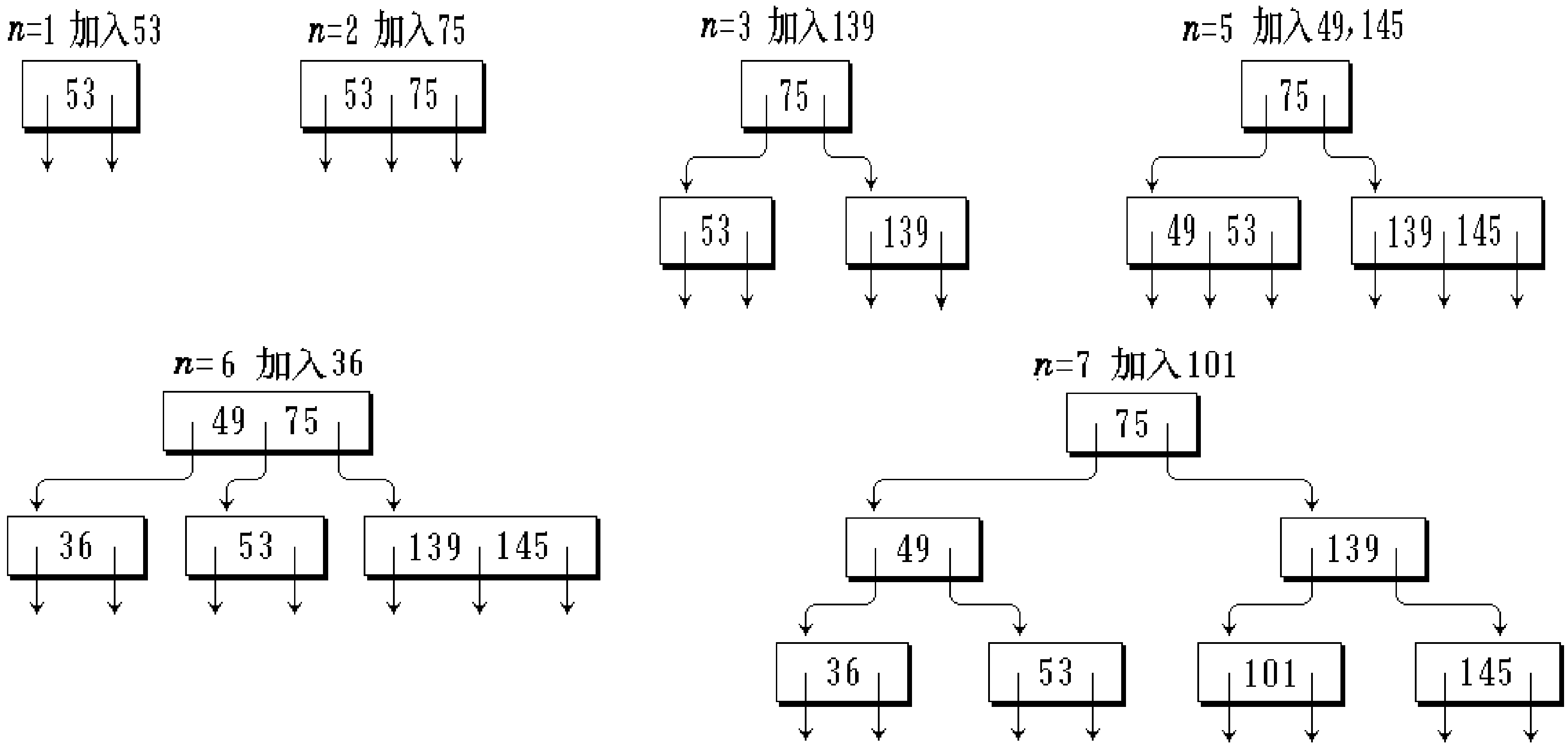
5.7 B-树和B+树(Cont.)

B-树插入操作：分裂与提升

- 这时结点 p 必须分裂成两个结点 p 和 q，它们包含的信息分别为：
  - 结点 p：
    $(\lceil m/2 \rceil -1, A_0, K_1, A_1, \ldots\ldots, K_{\lceil m/2 \rceil -1}, A_{\lceil m/2 \rceil -1})$
  - 结点 q：
    $(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil +1}, A_{\lceil m/2 \rceil +2}, \ldots\ldots, K_m, A_m)$
  - 位于中间的关键字$K_{\lceil m/2 \rceil}$与指向新结点 q 的指针形成一个二元组 $(K_{\lceil m/2 \rceil}, q)$，插入到这两个结点的双亲结点中，没有增加树的高度。
- 若双亲结点也“满”了呢？
  - 自底向上分裂结点
  - 最坏情况下从被插关键字的终端结点到根的路径上的所有结点都要分裂，树增高1。

5.7 B-树和B+树(Cont.)

- B-树的插入操作与建立
  - 例：从空树开始逐个加入关键字建立3阶B-树

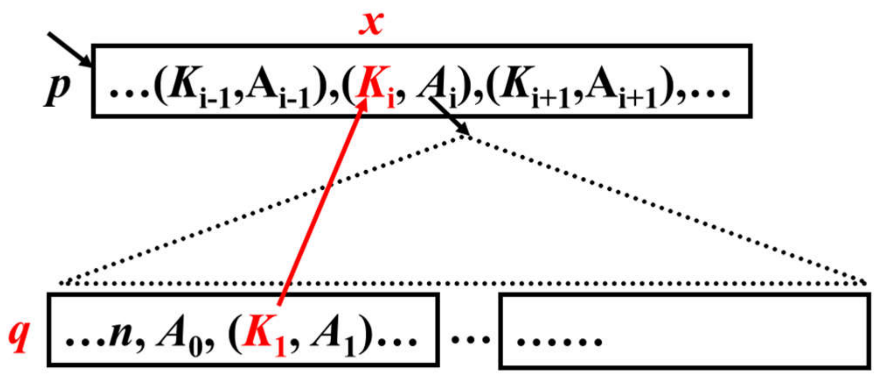


思考：为什么根结点可以只有2个子树（而不必至少半满⌈m/2⌉）？

5.7 B-树和B+树(Cont.)

B-树的删除操作：转换成在终端结点上的删除

- 首先，找到被删除关键字 x 所在结点p。若该结点不是终端结点，且$x=K_i$（$1\le i \le n$），则有两种选择来代替被删关键字K_i：
  - 该结点p中A_i所指子树的最小关键字（比如在结点q）；或者，
  - A_{i-1} 所指子树中的最大关键字，类似BST或AVL；
- 然后，在q所在的终端结点中删除x。把删除操作转换为终端结点上的删除操作。



5.7 B-树和B+树(Cont.)

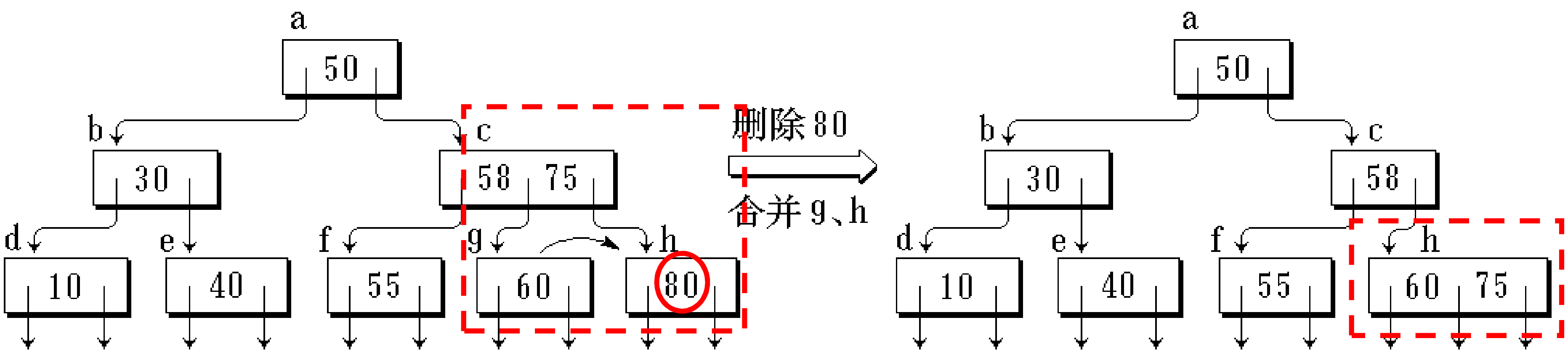
B-树的删除操作：终端结点上删除可分以下4 种情况：

情况 1：被删关键字所在终端结点p，是根结点，直接删除。
以下3种情况，p 都不是根结点。

情况 2：被删关键字所在终端结点p不是根结点，且删除前该结点中关键字个数大于最少限制，直接删除。

情况 3：删除前该结点中关键字个数为最少边界：兄弟够借，向兄弟借，然后调整，维持结构性质，以达到新的平衡。

情况 4：删除前该结点中关键字个数为最少边界：兄弟不够借，合并与调整，维持结构性质。

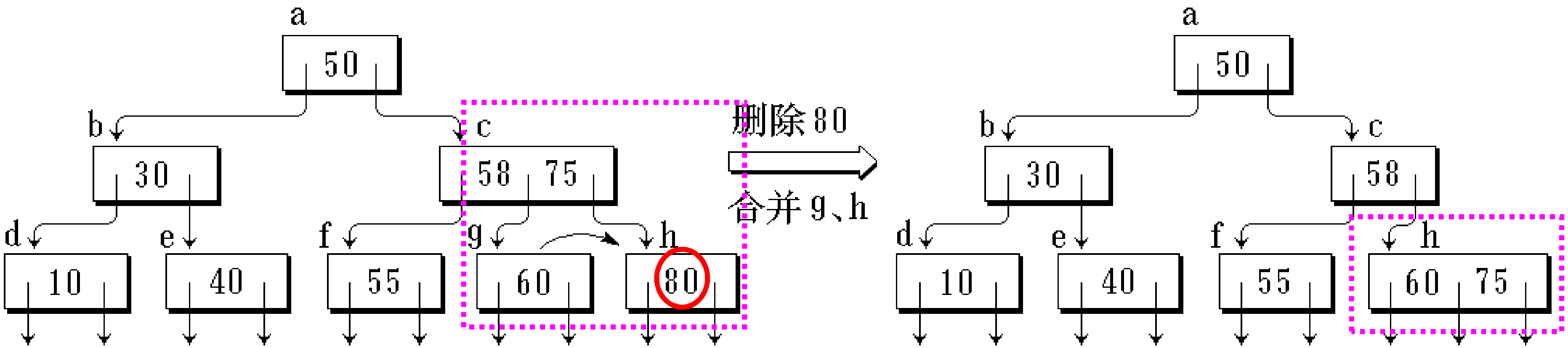


3阶B-树

5.7 B-树和B+树(Cont.)

B-树的删除操作：在终端结点上的删除：

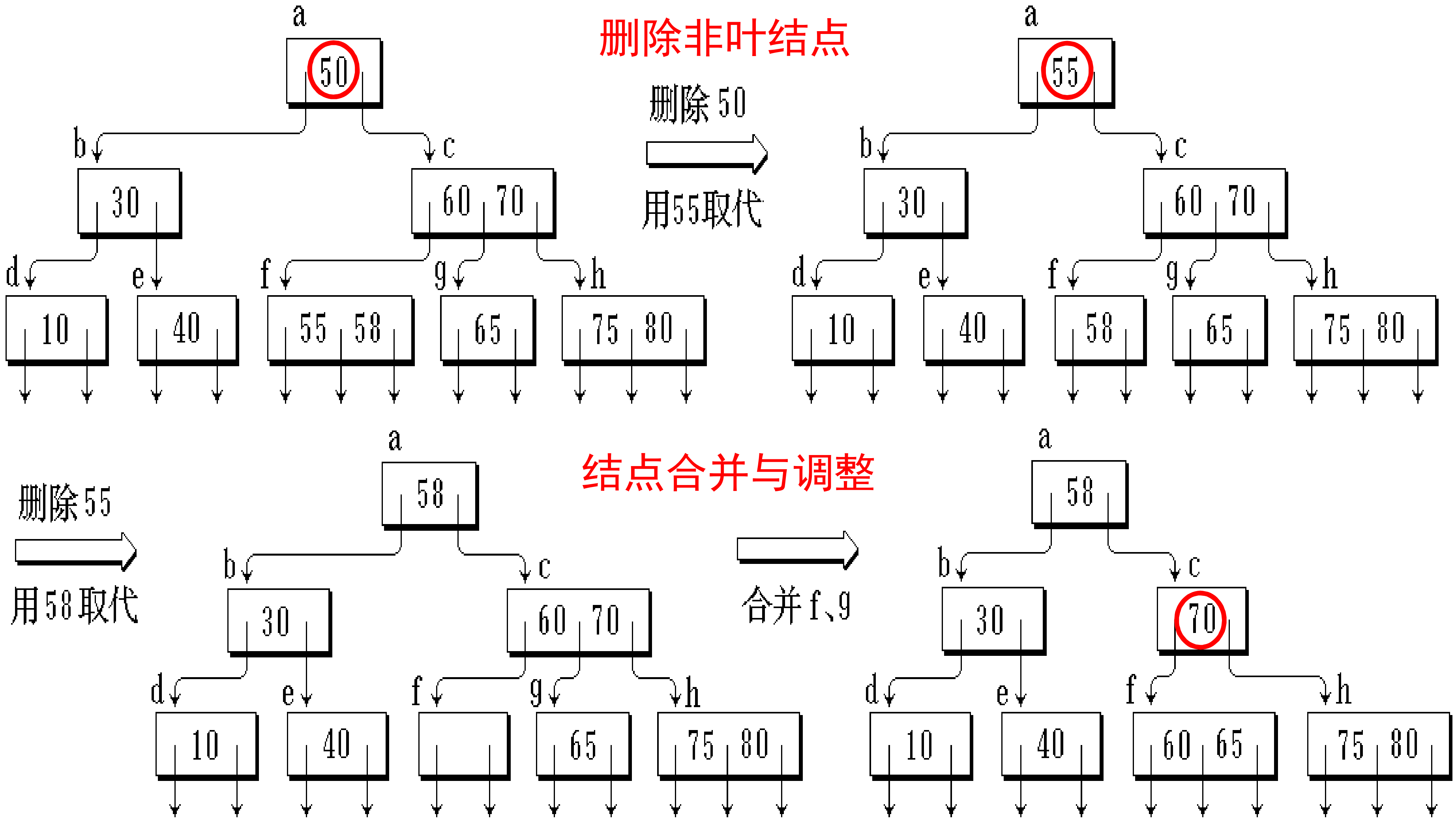
- 在合并结点的过程中，双亲结点中的关键字个数减少了。
- 若双亲结点是根结点且结点关键字个数减到 0，则该双亲结点应从树上删去，合并后保留的结点成为新的根结点（树高减1）；否则（根结点关键字个数未减到 0 ）将双亲结点与合并后保留（结点都写回磁盘），删除处理结束。
- 若双亲结点不是根结点，且关键字个数减到$\lceil m/2 \rceil$-2，则又要与它自己的兄弟结点合并，重复上面的合并步骤。最坏情况下这种结点合并处理要自下向上直到根结点。



3阶B-树

5.7 B-树和B+树(Cont.)

B-树的删除操作：在终端结点上删除

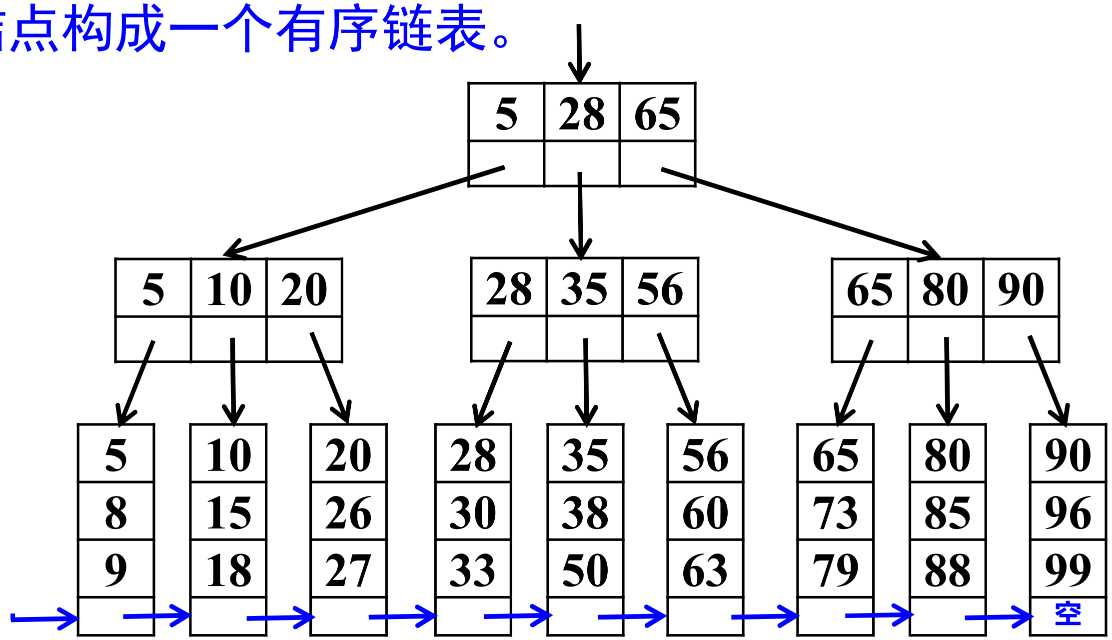


3阶B-树

5.7 B-树和B+树(Cont.)

B⁺树：B-树的一种变体，作为文件索引比B-树更普遍

- 与B-树的主要差异：
  - (1)非终端结点的子树个数与关键字个数相同
  - (2)与记录有关的信息均存放在终端结点中，尽管非终端结点具有与之相同的关键字，非终端结点仅具有索引作用
  - (3)终端结点附加一个指向下一个终端结点的指针，把所有终端结点构成一个有序链表。



3阶B+树

5.7 B-树和B+树(Cont.)

- B^+树同时支持随机检索和顺序检索：
  - B-树只适合随机检索
  - B^+树终端结点中存放的是对实际数据对象的索引
  - 在B^+树中有两个头指针：一个指向B^+树的根结点，一个指向关键字最小的终端结点。
  - 可对B^+树进行两种查找操作：一种是沿终端结点链顺序查找，另一种是从根结点开始，进行自顶向下，直至终端结点的随机查找。
- B^+树的查找与B-树的不同
  - 由于与记录有关的信息存放在终端结点中，查找时若在上层已找到待查的键值，并不停止，而是继续沿指针向下一直查到终端结点层的键值。
  - 由于B^+树的所有终端结点构成一个有序链表，可以按照键值排序的次序遍历全部记录。
  - 上面两种方式结合，使得B^+树非常适合范围检索。

5.7 B-树和B+树(Cont.)

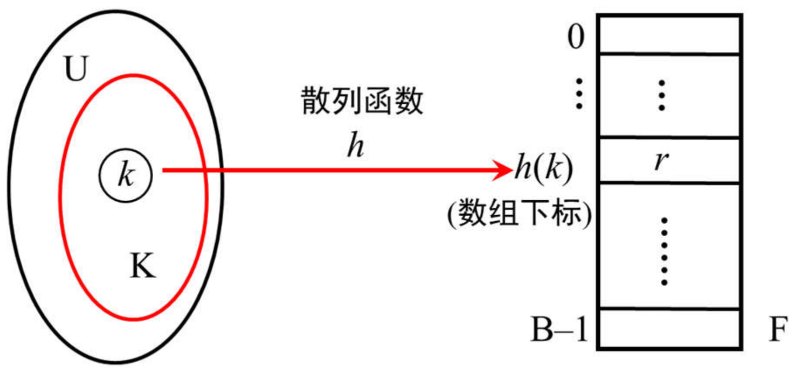
- B$^+$树的插入操作：与B-树的插入过程类似
  - 不同的是B$^+$树在终端结点上进行；
  - 如果终端结点中的键值个数超过m，就必须分裂成键值数目大致相同的两个结点，并保证上层结点中有这两个结点的最大键值。
- B$^+$树的删除操作
  - B$^+$树中的键值在终端结点层删除后，其在上层的副本可以保留，作为一个“分解关键字”存在；
  - 如果因为删除而造成结点中键值数小于[$m/2$]，其处理过程与B-树的处理一样。
- B*树：
  - 是B+树的变体，非根和非终端结点增加指向兄弟的指针
  - 非终端结点关键字个数至少为[$2m/3$]，即块的最低利用率为2/3（B+树是1/2）

5.8 散列技术

- 基于关键字比较的查找的时间性能分析
  - 查找操作：对于待查值k，通过比较，确定k在存储结构中的位置。
  - 其时间性能为$O(\log n)$~$O(n)$
  - 用判定树可以证明，基于关键字比较的查找平均和最坏情况下比较次数的下界是$\log n+O(1)$，即$\Omega(\log n)$
  - 要想突破此下界，就不能仅依赖于基于比较来进行查找。
- 能否不用比较，通过关键字键值直接确定存储位置？
  - 在键值和存储位置之间建立一个确定的对应关系

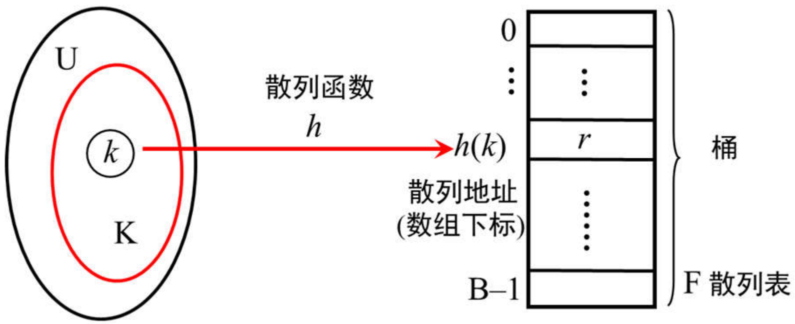
5.8 散列技术(Cont.)

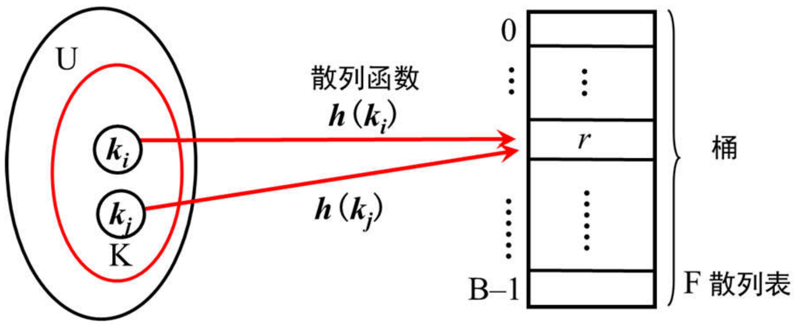
- 散列技术基本思想
  - 把记录（元素）的存储位置和该记录的关键字键值建立一种映射关系。键值的这种映射关系的像，就是相应记录在表中的存储位置。
  - 在理想情况下，散列技术无需任何比较就可以找到待查的关键字，其查找期望时间为O(1)。
- 散列技术相关概念
  - 散列函数(哈希函数，杂凑函数)：设 U 表示所有可能出现的关键字集合，K表示实际出现（实际存储）的关键字集合，即K⊆U，F[B – 1]是一个数组，其中B=O(|K|)。则，从 U 到数组F下标集合的一个映射 h：U→{0, 1, 2, …,B –1}称为散列函数。



5.8 散列技术(Cont.)

- 散列技术相关概念
  - 散列表（Hash表，杂凑表）：数组 F 称为散列表。
  - 桶(bucket)：数组 F 中的每个单元称为桶。
  - 散列地址（Hash地址，散列值，存储地址，桶号）：对于任意关键字$k \in U$，函数值h (k)称为 k 的散列地址。
  - 散列：将结点（记录）按其关键字的散列地址存储到散列表中的过程称为散列
  - 散列冲突(collision)：不同的关键字具有相同散列地址的现象称为散列冲突 。
  - 同义词(synonym)：发生冲突的两个关键字称为同义词。





5.8 散列技术(Cont.)

- 散列技术仅仅是一种查找技术吗？
  - 散列既是一种查找技术，也是一种存储技术。
- 散列是一种完整的存储结构吗？
  - 散列只通过记录的键值定位该记录，没有表达记录之间的逻辑关系，所以散列是面向查找的存储结构
- 散列技术适用于何种场合？
  - 常用于实际出现的关键字的数目远小于关键字所有可能取值的数量。
- 散列技术不适合于哪种类型的查找？
  - 不适用于允许多个记录有同样关键字值的情况；
  - 不适用于范围查找和最大或最小键值查找。
    - 如：在散列表中，不可能找到在某一范围内的记录，以及最大或最小键值的记录。
- 散列技术最适合回答：哪个记录的键值等于待查值。

5.8 散列技术(Cont.)

- 散列技术需解决的关键问题：
  - 散列函数的构造。
    - 如何设计一个简单、均匀，高存储利用率的散列函数
  - 冲突处理
    - 如何采取合适的冲突处理方法来解决冲突？
  - 散列结构上的查找、插入和删除操作

散列函数构造

- 散列函数构造的原则：
  - 计算简单：散列函数不应该有很大的计算量，否则会降低查找效率。
  - 分布均匀：散列函数值即散列地址，要尽量均匀分布在地址空间，以保证存储空间的有效利用，并减少冲突。

5.8 散列技术(Cont.)

散列函数构造

- 散列函数构造方法：1）直接定址法
  - 散列函数是关键字值的线性函数，即：

    $h(key) = a \times key + b$ (其中，a, b为常数)
  - 示例：关键字的取值集合为{10, 30, 50, 70, 80, 90}，选取的散列函数为$h(key)=key/10$，则散列表为：

0	1	2	3	4	5	6	7	8	9
	10		30		50		70	80	90

  - 适用情况：事先知道关键字的值，关键字取值集合不是很大且连续性较好。

5.8 散列技术(Cont.)

散列函数的构造

- 散列函数的构造方法：2）质数除余法
  - 散列函数为：$h(key)=key \ \% \ m$
  - 通常选m为小于或等于表长B的最大质数。
  - 示例：关键字的取值集合为{14, 21, 28, 35, 42, 49, 56, 63}，表长B=12。则选取m=11，散列函数为$h(key)=key \% 11$，则散列表如下。
  - 适用情况：不要求事先知道键值的分布。质数除余法是一种最简单、也是最常用的构造散列函数的方法。

0	1	2	3	4	5	6	7	8	9	10
	56	35	14		49	28		63	42	21

5.8 散列技术(Cont.)

散列函数构造

- 散列函数构造方法：
  3）平方取中法
  - 取 key^2 的中间的几位数作为散列地址，扩大相近数的差别。
  - 然后根据表长取中间几位作为散列值，使地址值与关键字的每一位都相关。
  - 散列地址的位数要根据B来确定，有时要设一个比例因子，限制地址越界。
  - 适用情况：事先不知道键值的分布，且键值位数不是很大

记录	*key*	*key*^{2}	Hash
A	0100	0 010 000	010
I	1100	1 210 000	210
J	1200	1 440 000	440
I0	1160	1 370 400	370
P1	2061	4 310 541	310
P2	2062	4 314 704	314
Q1	2161	4 734 741	734
Q2	2162	4 741 304	741
Q3	2163	4 745 651	745

5.8 散列技术(Cont.)

散列函数构造

- 散列函数构造方法： 4）折叠法
  - 若关键字位数较多，可根据B的大小，将关键字分割成位数相同的若干段（最后一段位数可以不同），然后将各段叠加和(最高位舍去进位)作为散列地址。
  - 示例：图书编号: 0-442-20586-4

$$
\begin{array}{rr}
 & 5864 \\
 & 4220 \\
+ & 04 \\
\hline
 & 10088
\end{array}
\qquad
\begin{array}{rr}
 & 5864 \\
 & 0224 \\
+ & 04 \\
\hline
 & 6092
\end{array}
$$

左：移位叠加

$\text{Hash}(\,key\,) = 0088$

右：间界叠加

$\text{Hash}(\,key\,) = 6092$

  - 适用情况：键值位数很多，事先不知其分布。

5.8 散列技术(Cont.)

散列函数构造

- 散列函数构造方法：
  5）数字分析法
  - 根据关键字值各位数的分布情况，选取分布比较均匀的若干位组成散列地址。
  - 示例：散列表长为10000，可取中间四位中的两位为散列地址。
  - 适用情况：若事先知道关键字集合，且关键字的位数比散列表地址位数多。

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

5.8 散列技术(Cont.)

散列函数构造

- 散列函数构造方法：6）随机数法
  - 选择一个随机函数，取关键字的随机函数值作为散列地址，即：Hash (key) = random(key)
    其中，random是某伪随机函数，且函数值在0, …, B-1之间。
  - 适用情况：关键字长度不等时采用此法较恰当

- 构造散列函数重点解决如下问题：
  - 计算简单:计算Hash函数所需时间
  - 分布均匀:
    - 关键字长度
    - 散列表大小
    - 关键字分布情况
    - 记录查找频率

5.8 散列技术(Cont.)

冲突处理方法：开放定址法

- 基本思想：
  - 冲突发生时，使用某些探测技术，在散列表中形成一个探测序列，沿此序列逐个单元查找，直到：
    - 找到给定的关键字，或者
    - 碰到一个开放地址（即空地址单元、空桶），或者
    - 既未找到给定的关键字也没碰到一个开放地址。
- 常用探测技术方法：如何寻找下一个空散列地址?
  - 线性探测法
  - 线性补偿探测法
  - 二次探测法
  - 随机探测法
- 闭散列表：
  - 开放定址法处理冲突得到的散列表叫闭散列表。
  - 表内寻址。

5.8 散列技术(Cont.)

冲突处理方法：开放定址法：

线性探测法

- **基本思想**：发生冲突时，从冲突位置的下一个位置起，依次寻找空的散列地址。
- **探测序列**：设关键字值*key*的散列地址为$h(key)$，闭散列表的长度为B，则发生冲突时，寻找下一个散列地址的公式为：

$$h_i=(h(key)+d_i)\ \%\ B \quad （d_i=1，2，\ldots，m-1）$$

- 示例：关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为$h(key)=key\ \%11$，用线性探测法处理冲突，则散列表为：

0	1	2	3	4	5	6	7	8	9	
11	22		47	92	16	3	7	29	8	
22			3	3	3		29	8		

- **堆积现象**：处理冲突过程中，非同义词之间对同一个散列地址的争夺现象。

5.8 散列技术(Cont.)

冲突处理方法：开放定址法

- 线性补偿探测法：发生冲突时，寻找下一个散列地址的公式为：

  $h_i=(h(key)+d_i)\ \%\ B$　（d_i =1c，2c，…）

- 二次探测法：发生冲突时，寻找下一个散列地址的公式为：

  $h_i=(h(key)+d_i)\ \%\ B$

  其中，$d_i=1^2$，-1^2，2^2，-2^2，…，q^2，$-q^2$，且$q \le B/2$

- 随机探测法：发生冲突时，寻找下一个散列地址的公式为：

  $h_i=(h(key)+d_i)\ \%\ B$

  其中，下一个散列地址的位移量是一个随机数列， 即 d_1, d_2, …,d_{B-1}是1, 2, …, B-1的随机序列。

  注意：插入、删除和查找时，要使用同一个随机序列。

5.8 散列技术(Cont.)

冲突处理方法：开放定址法：线性探测法的实现

- 查找算法实现
  - 存储结构定义

```
struct records{
        keytype  key;
        fields other;
};
typedef records  HASH[B];
```

  - Search算法实现
  - Insert算法实现
  - Delete算法实现

Search算法实现

```
int  Search(keytype k, HASH F)
{   int locate=first= h(k), rehash=0;
    while((rehash<B)&&
            (F[locate].key!=empty)){
        if(F[locate].key==k)
                return locate;
        else
                rehash=rehash+1;
        locate=(first+rehash)%B
    }
    return -1;
}/*Search*/
```

5.8 散列技术(Cont.)

冲突处理方法：开放定址法—线性探测法的实现

- 相关操作实现

Insert算法实现

```
void Insert(records R, HASH F)
{   int locate = first=h(k), rehash= 0;
    while ((rehash<B)&&
              (F[locate].key!=R.key)) {
       locate=(first+rehash)%B;
       if ((F[locate].key==empty)||
               (F[locate].key==deleted))
          F[locate]=R;
      else
             rehash+=1; }
      if (rehash>=B)
             cout<<"hash table is full!";
} /*Insert */
```

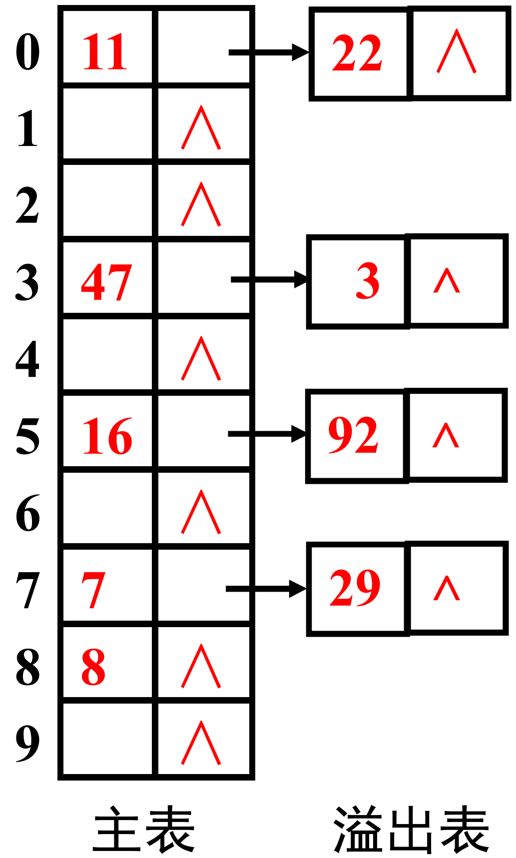
Delete算法实现

```
void Delete(keytype k,HASH F)
{      int locate;
       locate = Search(k,F);
       if( locate != -1)
        F[locate].key = deleted;
}/*Delete*/
```

5.8 散列技术(Cont.)

冲突处理方法：带溢出表的内散列法

- **基本思想**：扩充散列表每个桶，形成带溢出表的散列表。
  - 每个桶包括两部分：
    - 一部分是主表元素；
    - 另一部分或者为空或者由一个链表组成溢出表，其首结点指针存入主表链域。
  - 主表元素的类型与溢出表的类型相同。
- **示例**：
  - 关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为h(key)=key %11，用带溢出表的内散列法处理冲突。
- **特点**：
  - 主表及其溢出表元素的散列地址相同
  - 空间利用率不高



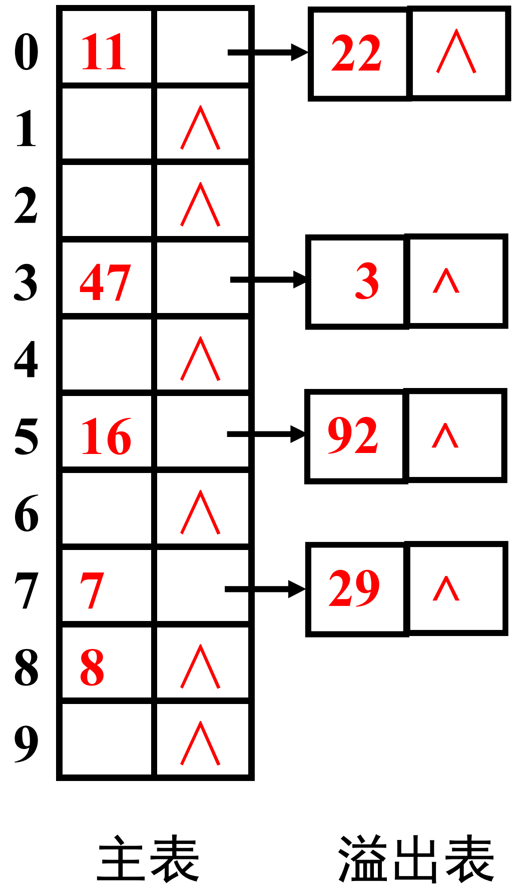
5.8 散列技术(Cont.)

冲突处理方法：带溢出表的内散列法

- 查找算法实现：
  - 存储结构的定义

```
typedef  struct  celltype{
        records   data;
        celltype   next;
} HASH[ B ];
```

  - 查找算法：
    - 插入：当发生冲突时，把新元素插入溢出表；
    - 查找：要查看同一散列地址的主表和溢出表；
    - 删除：若被删除的是主表元素，则要溢出表的首结点移入主表，删除首结点。



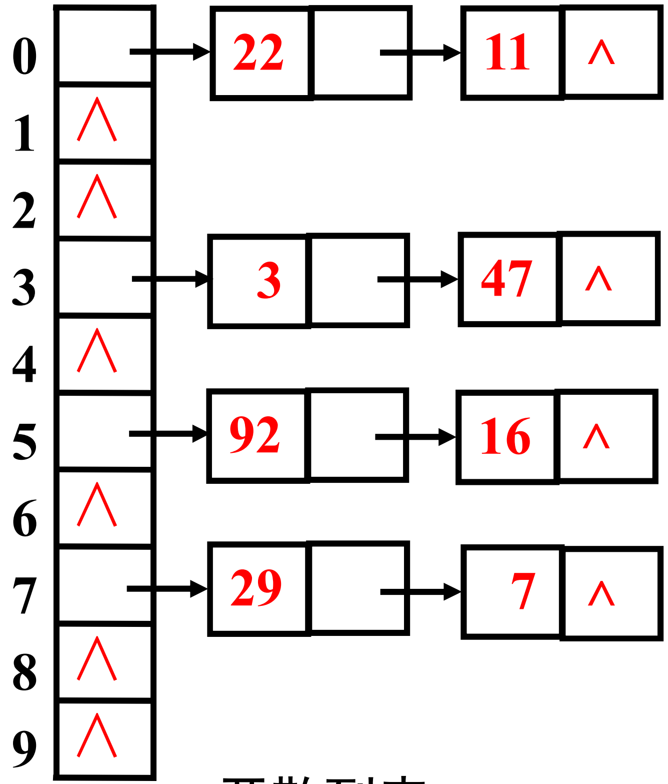
5.8 散列技术(Cont.)

冲突处理的方法：拉链法（链地址法）

- 基本思想：将所有散列地址相同的记录，即所有同义词的记录，存储在一个单链表中（称为同义词子表），在散列主表中存储所有同义词子表的首指针（桶的链表首指针）。
- 开散列表：用拉链法处理冲突构造的散列表叫做开散列表。
- 示例：关键字取值集合为 {47, 7, 29, 11, 16, 92, 22, 8, 3}，散列函数为：

  $$h(key)=key \ \%11$$

  采用链地址法处理冲突。
- 设n个记录存储在长度为B的散列表中，则同义词子表的平均长度为n / B。



开散列表

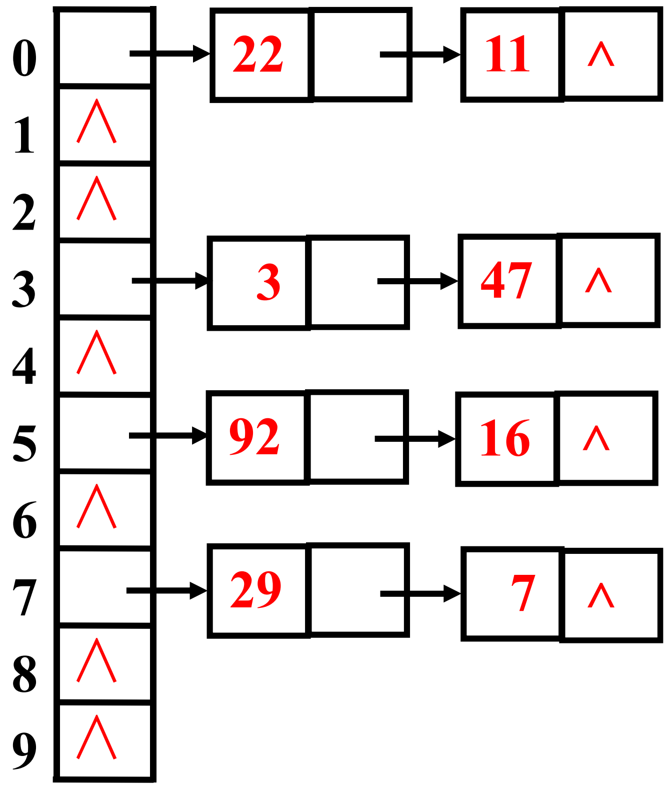
5.8 散列技术(Cont.)

冲突处理的方法：拉链法（链地址法）

- 开散列表的实现
  - 存储结构定义

```
struct celltype{
        records data;
        celltype *next;
};/*链表结点类型*/
typedef  celltype  *cellptr;

/*开散列表类型，B为桶数*/
typedef cellptr  HASH[B]；
```

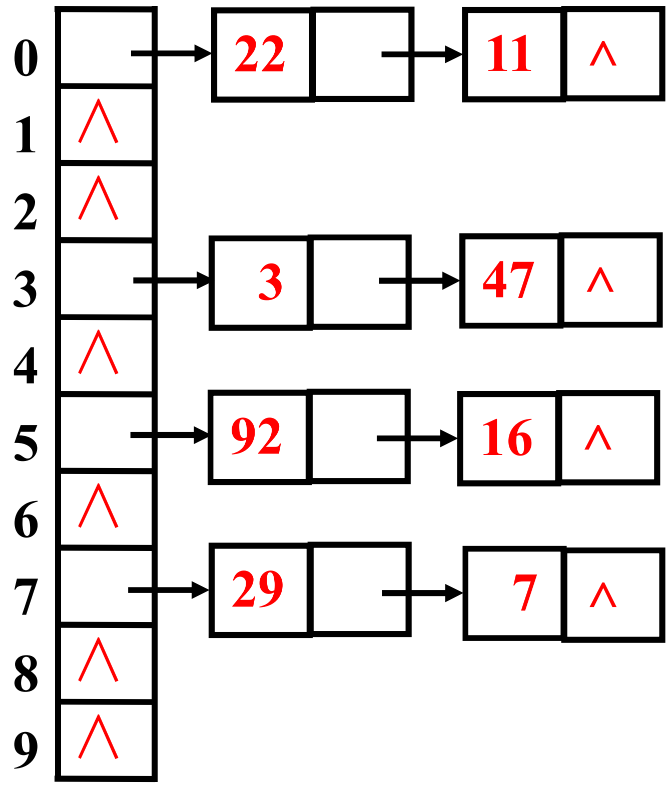


5.8 散列技术(Cont.)

冲突处理的方法：拉链法（链地址法）

- 开散列表的实现
  - 查找算法

```
cellptr Search(keytype k, HASH F)
{   cellptr bptr;
    bptr=F[h(k)];
    while(bptr!=NULL)
        if(bptr->data.key==k)
            return bptr;
        else
            bptr=bptr->next;
    return bptr; //没找到
}/*Search*/
```



5.8 散列技术(Cont.)

冲突处理方法：拉链法

- 开散列表的实现

插入算法

```
void Insert(records R, HASH  F)
{  int bucket;
   cellptr oldheader;
    if( SEARCH(R.key,F)==NULL){
         bucket=h(R.key);
         oldheader=F[bucket];
         F[bucket]=new celltype;
         F[bucket]->data=R;
         F[bucket]->next=oldheader;
   }
}/*Insert
```

删除算法

```
void Delete(keytype k, HASH F)
{ int bucket=h(k); celltype bptr, p;
  if(F[bucket] != NULL)//可能在表中
    if(F[bucket]->data.key==k){//首元素
      bptr= F[bucket];
      F[bucket]=F[bucket]->next;
      free (bptr);
    }else{//可能在中间或不存在
      bptr=F[bucket];
      while(bptr->next!=NULL)
         if(bptr->next->data.key==k){
            p=bptr->next;
            bptr->next=p->next;
             free( p);
         }else
             bptr=bptr->next;
    }}/* Delete */
```

5.8 散列技术(Cont.)

散列查找性能分析

- 由于冲突的存在，产生冲突后的查找过程仍需进行给定值与键值比较。
- 键值比较次数取决于产生冲突的概率
- 影响冲突产生的因素：
  - 散列函数是否均匀
  - 处理冲突的方法
  - 散列表的装载因子：α=表中填入的记录数/表的长度
    - 装载因子α反映了散列表的装满程度
    - α越大，冲突的概率越高，反之，冲突概率越小。
- 开散列表与闭散列表性能比较

	堆积现象	结构开销	插入/删除	查找效率	估计容量
开散列表	不产生	有	效率高	效率高	不需要
闭散列表	产生	没有	效率低	效率低	需要

5.8 散列技术(Cont.)

散列查找性能分析

- 不同处理冲突方法的平均查找长度

ASL / 处理冲突方法	查找成功时	查找不成功时
线性探测法	$\frac{1}{2}(1+\frac{1}{1-\alpha^2})$	$\frac{1}{2}(1+\frac{1}{1-\alpha^2})$
二次探测法	$-\frac{1}{\alpha}ln(1+\alpha)$	$\frac{1}{1-\alpha}$
拉链法	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$

应用例题

- 将关键字序列（7，8，30，11，18，9，14）散列存储到散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为：H(key)=（key*3）%7，处理冲突采用线性再散列法，要求装填（装载）因子为0.7。
  （1）请画出所构造的散列表。
  （2）分别计算等概率情况下查找成功和查找不成功的平均长度
- 解答：
  （1）由于序列长度为7，装填因子为0.7，所以表长为10。
  （2）$ASL_{成功}$ =（1+2+1+1+1+3+3）/7=12/7
  $ASL_{不成功}$ =（3+2+1+2+1+5+4）/7=18/7
  查找不成功：不是表中一元素，散列地址分别是0-6

数组下标	0	1	2	3	4	5	6	7	8	9
数组元素	7	14		8		11	30	18	9	
元素插入次序	1	7		2		4	3	5	6	
成功的比较次数	1	2		1		1	1	3	3	
不成功比较次数	3	2	1	2	1	5	4	不可能的散列地址		

本章小结

- 查找结构
  - 线性结构
    - 线性无序表
    - 线性有序表
    - 有序表+无序表
  - 树型结构
    - BST和AVL
    - B-树和B+树
  - 散列结构
    - 内散列结构
    - 外散列结构
    - 带溢出表的散列结构
- 查找方法
  - 基于关键字比较的查找
  - 基于关键字存储位置的查找
- 查找操作
  - Search
  - Insert(Create)
  - Delete
- 判定树
- 查找的性能指标
  - 查找成功的ASL
  - 查找不成功的ASL