数据结构与程序设计
(Data Structure and Programming)

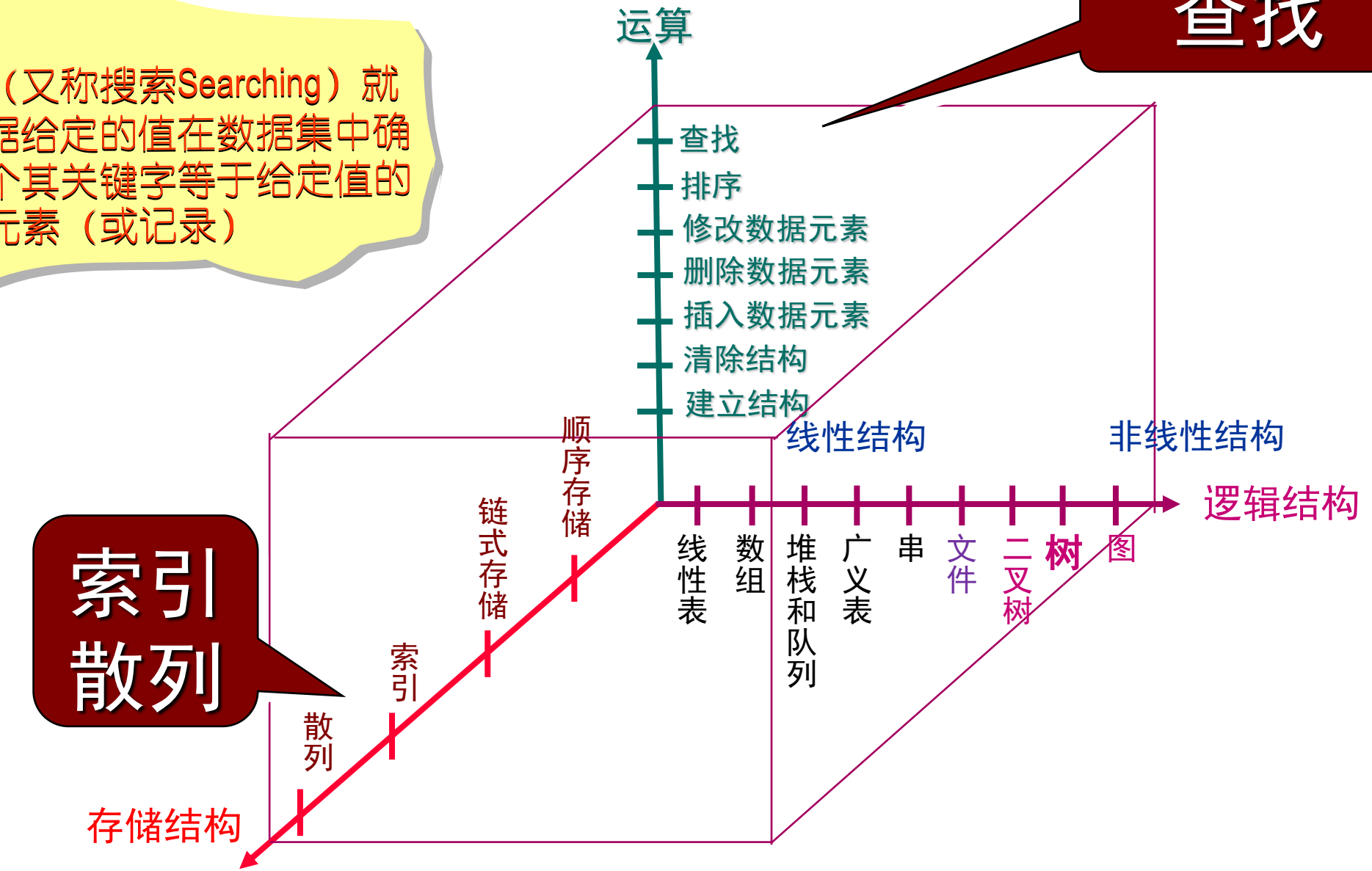
查找
(Searching)

北航计算机学院 林学练

数据结构的基本问题空间

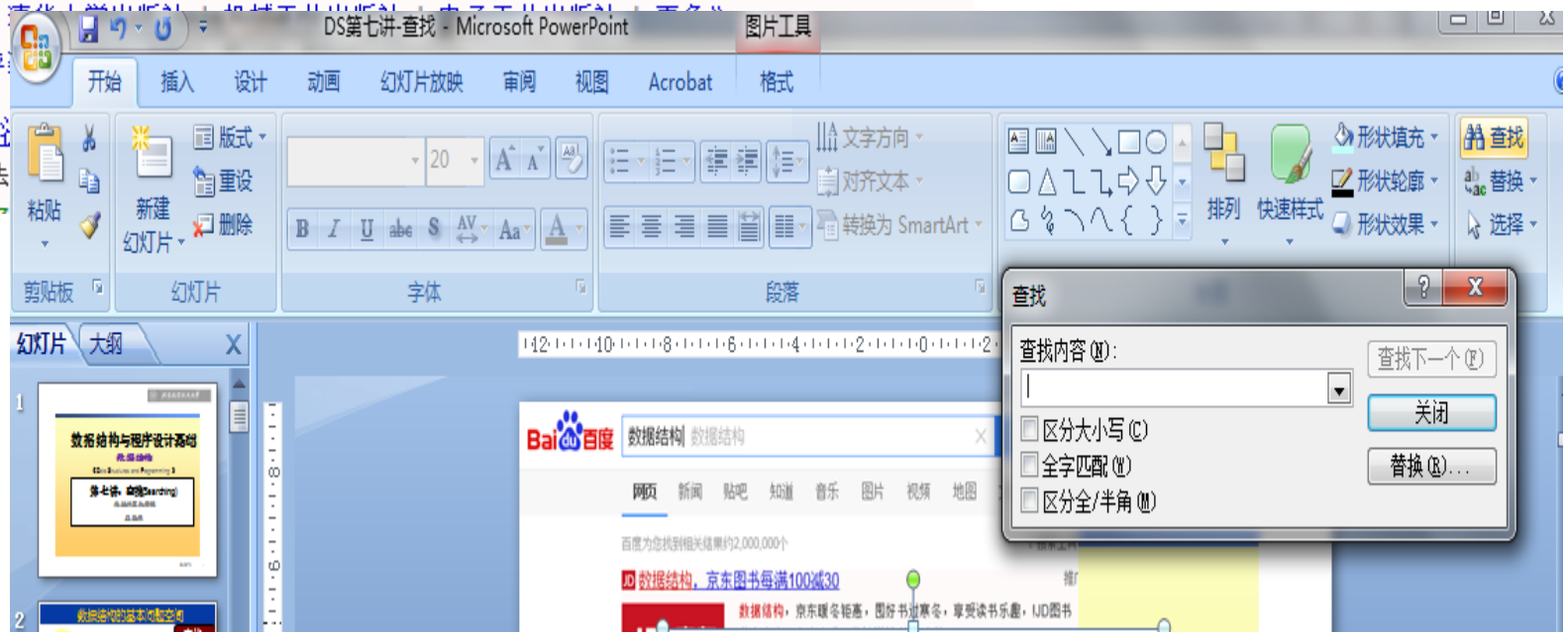
查找（又称搜索Searching）就是根据给定的值在数据集中确定一个其关键字等于给定值的数据元素（或记录）

查找



索引
散列

互联网时代，几乎我们每个人都会要用到搜索。



7.1 查找的基本概念

例1 花名册

学号	姓名	性别	年龄	其他
99001	张三	女	20	……
99002	李四	男	18	……
99003	王五	男	17	……
…	…	…	…	……
…	…	…	…	……
…	…	…	…	……
99030	刘末	女	19	……

例2 商品清单

编号	名称	库存数量	入库时间	其他
010020	电视机	300	2005.7	…
010021	洗衣机	100	2006.1	…
010023	空调机	50	2006.5	…
010025	电冰箱	30	2006.9	…
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

一、查找表（Search Table）

学 号	姓 名	性别	年龄	其 他
99001	张 三	女	20	……
99002	李 四	男	17	……
99003	王 五	男	18	……
99030	刘 末	女	19	……

主关键字

主关键字（Primary Key）：可以唯一的标识一个记录。

字段、数据项

属性：描述一个客体某一方面特征的数据信息。

记录：反映一个客体数据信息的集合。

查找表：具有相同**属性定义**的记录的集合。

关键字：区分不同记录的属性或属性组。

（主关键字、次关键字）

二、数据特点及访问要求

1. 数据有“增删改查”的要求，查找是基础

2. 查询可分为随机查询和范围查询

随机查询：确定某个特定记录是否存在或返回记录信息。

输入：(1) 查找表的第i个记录；
(2) 查找当前位置的下一个记录；
(3) 按关键字值查找记录。

返回：查找成功,给出被查到记录的位置；
查找失败,给出相应的信息。

3. 若数据规模小，可以把全量数据加载到内存，采用顺序查找或折半查找法；亦可构造二叉查找树

4. 若数据规模大，大部分数据需持久存储在外存……

数据在存储介质上的组织方式

1. 连续组织方式(顺序组织方式)

2. 链接组织方式

在物理结构中记录排列的先后次序与在逻辑结构中记录排列的先后次序一致的查找表称为 *顺序表* 。

记录的排列按关键字值有序的顺序表称为有序顺序表，否则，称为一般顺序文件。

逻辑上划分

在存储介质上采用连续组织方式的顺序表称为连续顺序表；采用链接组织方式的顺序表称为链接顺序表。

物理上划分

若排序顺序文件在存储介质上采用连续组织方式，称之为 *有序连续顺序表* 。

数据在存储介质上的组织方式

1. 连续组织方式(顺序组织方式)
2. 链接组织方式
3. 索引组织方式
4. 散列组织方式

顺序查找　　索引查找　　散列查找

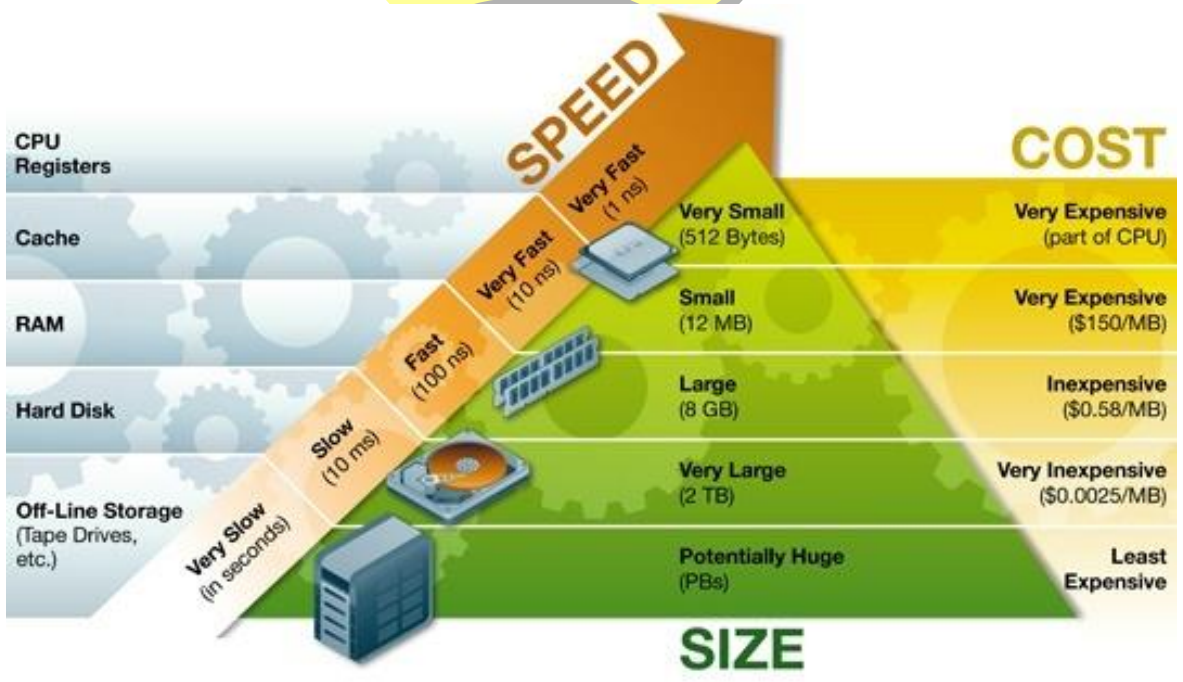
到目前为止，我们都假设数据足够小，可以全量加载到内存的情况，未考虑数据规模太大以至于无法或不适合将全量数据加载到内存的情形。

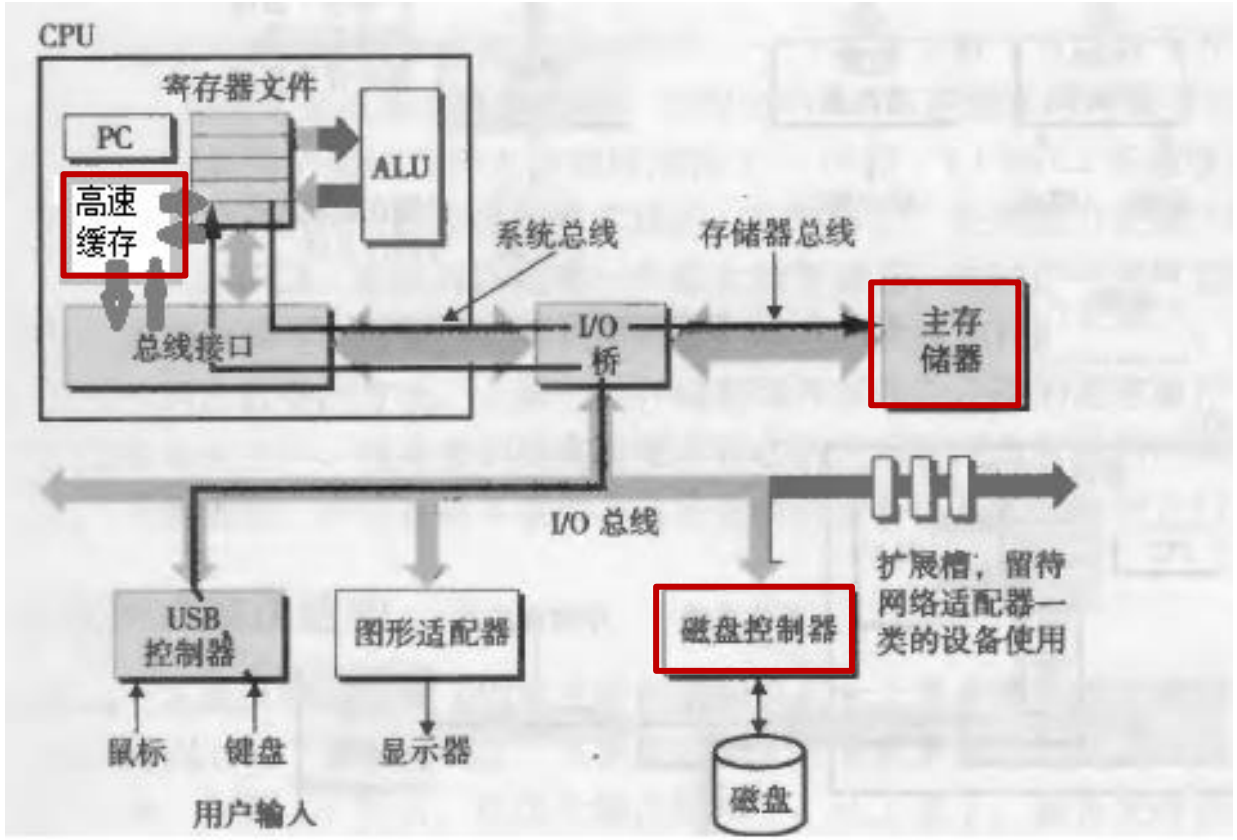
当数据规模大，查询需求多样，需平衡内外存访问的特性以获得较好的综合性能时，问题将变得更为复杂一些。

大数据时代，真实的应用场景往往就是如此！

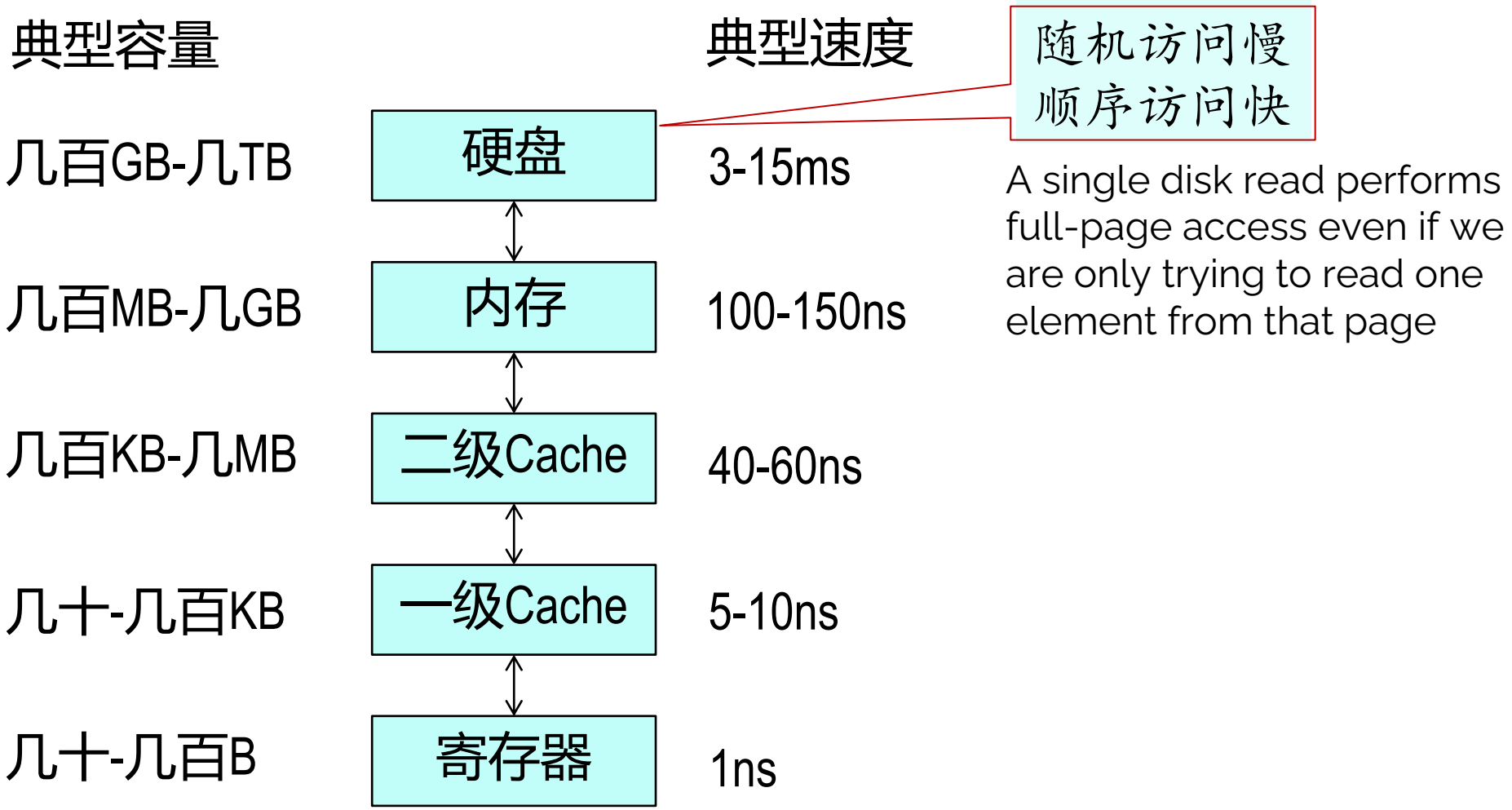
扩展知识*

计算机结构
与存储性能





扩展知识*



1s = 1000ms　　1ms = 1000us　　1us = 1000ns

扩展知识*

如何在大规模数据集中快速查找?

对于一次不能加载至内存中的大数据（如数据库、文件系统，实际存储在硬盘上，访问速度慢），如何以尽可能少的硬盘访问次数找到所要的数据?

如何利用不同存储介质的性能特性实现快速查找?

如何根据不同属性查找？

7.2 索引（Index）

一、索引的基本概念

1.索引(index)

记录关键字值与记录的存储位置之间的对应关系。

2.索引表(index table)——一种索引方式

(1) 索引表一般由数据管理系统自动产生的；

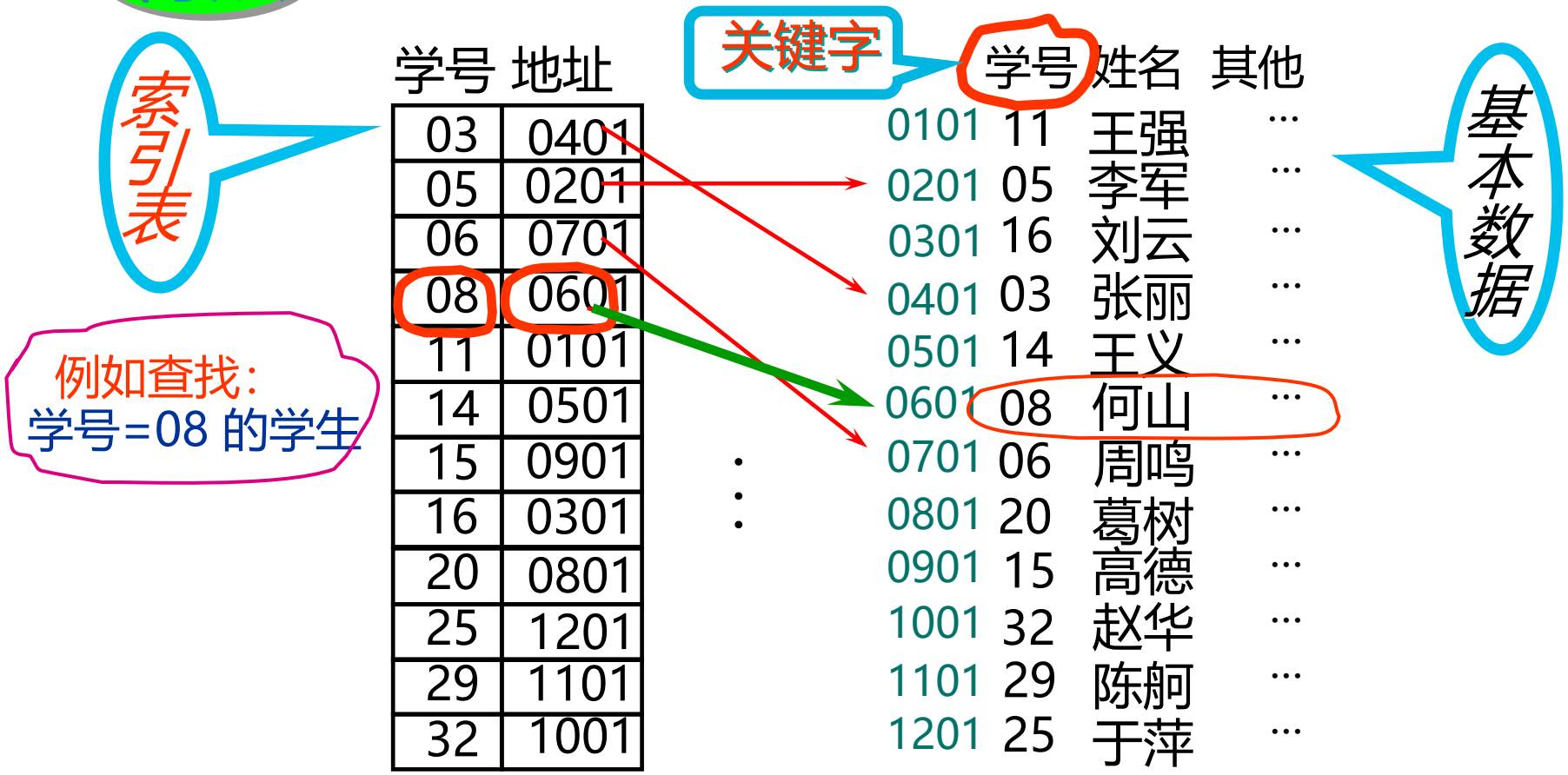
(2) 索引表中表项按关键字值有序组织。

3.索引文件(indexed file)

由基本数据与索引表两部分组成的数据集称为索引文件。

二、稠密索引

特点 **基本数据中的每一个记录在索引表中都占有一项。**



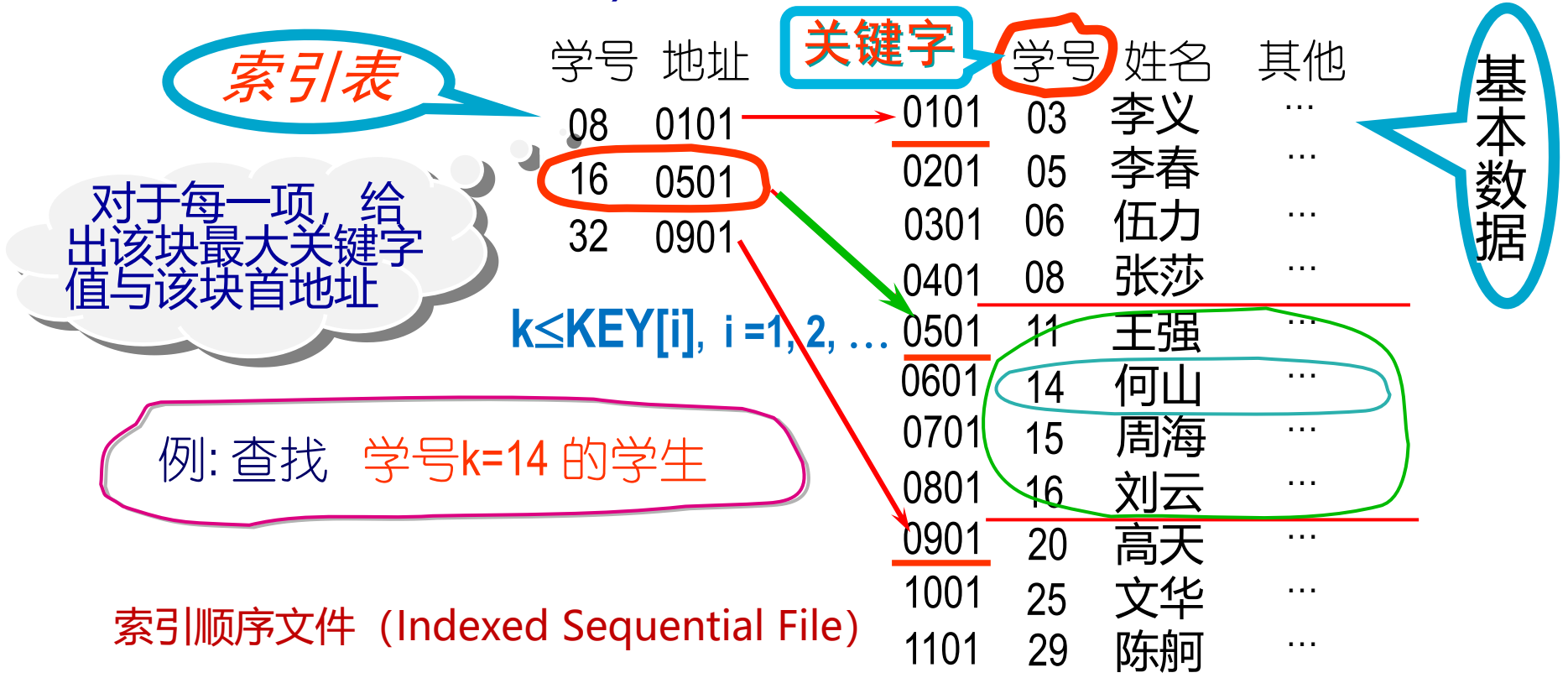
索引非顺序文件（Indexed NonSequential File）

查找一个记录存在与否的过程是直接查找索引表。

三、非稠密索引-分块索引

特点

将文件的基本数据中记录分成若干块(块与块之间记录按关键字值有序, 块内记录是否按关键字值有序无所谓),索引表中为每一块建立一项。



查找一个记录存在与否的过程：先查找索引表(确定被查找记录所在块)，然后在相应块中查找被查记录存在与否。

四.多重索引

有时不仅需要对主关键码进行查找，还可能需要对次关键码进行查找，此时还需建立一系列的次关键码索引。

五.多级索引

当索引文件的索引本身非常庞大时,可以把索引分块,建立索引的索引,形成*树形结构的多级索引*。

二叉树/AVL树：当数据存储在外存时，AVL树由于深度过大，造成磁盘IO次数多，访问效率低

多路(平衡)树：B-Tree，B+Tree

7.2 B-树——多路(平衡)查找树

B-树(B-Tree)由R.Bayer和E.MacCreight于1970年提出，是一种平衡的多路树。为什么叫B-树，有人认为是由“平衡(Balanced)”而来，而更多认为是因为他们是在Boeing科学研究实验发明的此概念并以此命名的。

B-树针对大块数据的读写操作做了优化。

B-树多用于文件系统或数据库系统常用的索引结构。

注：网络上经常混用B树，B-树，B_树。其中“B_”是没有的，“B减树”的叫法是错误的。

7.2 B-树——多路(平衡)查找树

二叉查找树的问题：

- 二叉查找树：当数据规模比较大时，二叉查找树深度高（查找树的深度主要由数据规模决定），查找效率随之降低；极端情况会退化成线性表，查找效率更差。
- AVL树降低了树的高度，一定程度上提高了查找效率，但是由于数据规模大，树的深度仍然较高。
- 当数据存储在外存时并且以二叉树/AVL树管理数据，由于深度过大，造成磁盘IO次数多，访问效率显著降低。

问题：如果你来设计这棵多路查找树，你认为它相比二叉查找树该有哪些变化？

7.2.1 B-树的定义

B-树是平衡二叉查找树的泛化!

一个m阶的B-树为满足下列条件的m叉树:

降低树的高度

(1) 每个分支结点最多有m棵子树;

(2) 除根结点外，每个分支结点最少有$\lceil m/2 \rceil$棵子树;

(3) 根结点最少有两棵子树(除非根为叶结点);

无需频繁平衡

平衡

(4) 所有“叶结点”都在同一层上;

(5) 每个节点都存有关键字索引和数据，包含下列信息:

$$n,\ p_0,\ key_1,\ p_1,\ key_2,\ p_2,\ \ldots,\ key_n,\ p_n$$

关键字大小关系……

其中,n为结点中关键字值的个数, $n \le m-1$

最多m−1个关键字

key_i为关键字，且满足 $key_i < key_{i+1}$, $1 \le i < n$

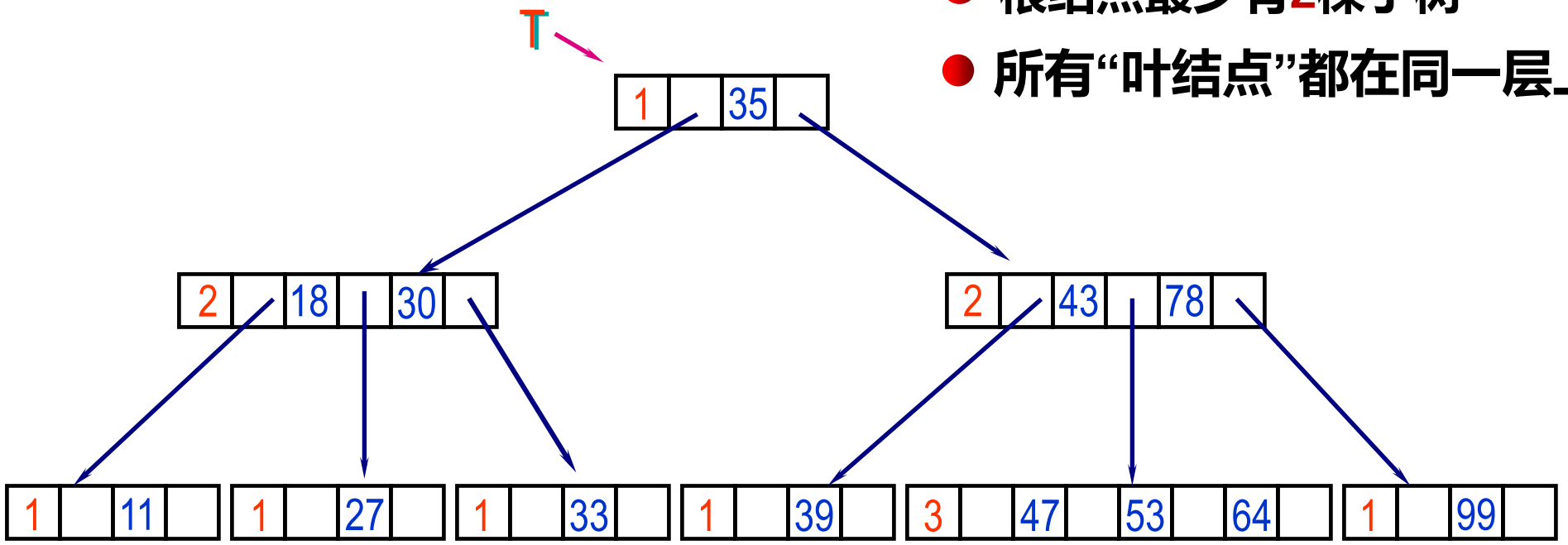
p_i为指向该结点的第i+1棵子树根结点的指针.

注：不同文献对B-树的定义可能有微小的差异

一棵4阶B-树

也称为2-3-4 树

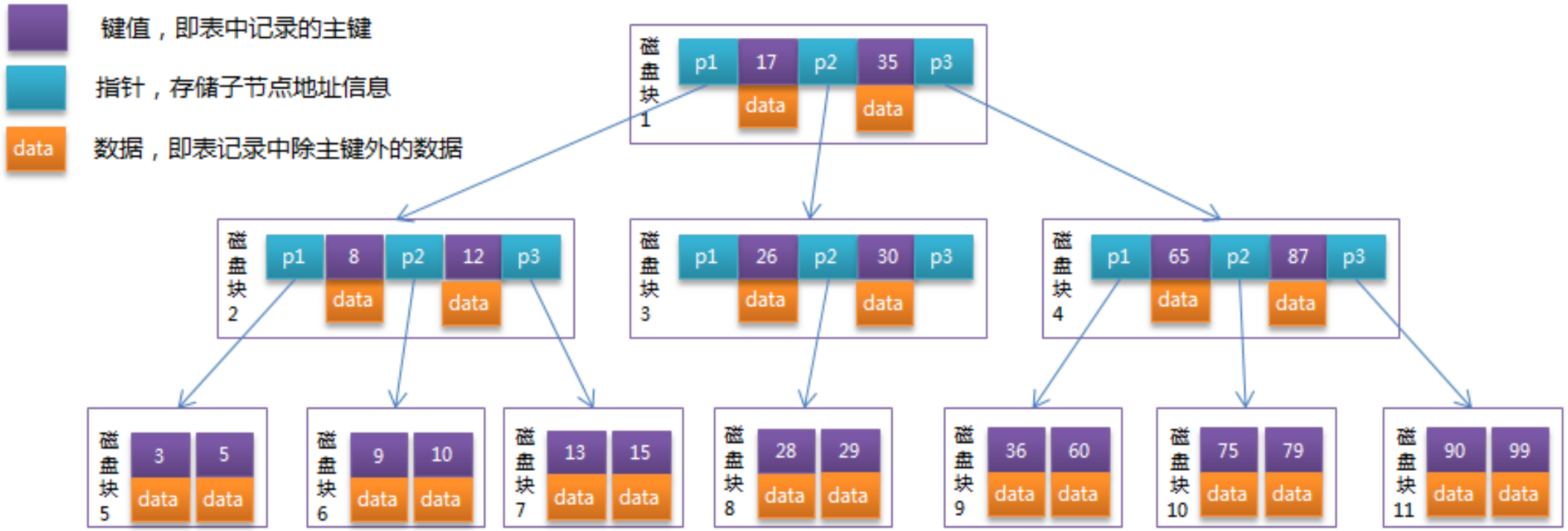
- 每个分支结点最多有4棵子树(即最多有3个关键字值)
- 每个分支结点最少有2棵子树
- 根结点最少有2棵子树
- 所有“叶结点”都在同一层上



B-树用于管理外存数据*

B-Tree是为磁盘等外存储设备设计的一种多路平衡查找树

文件系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位，大小一般为4K, 8K或16K

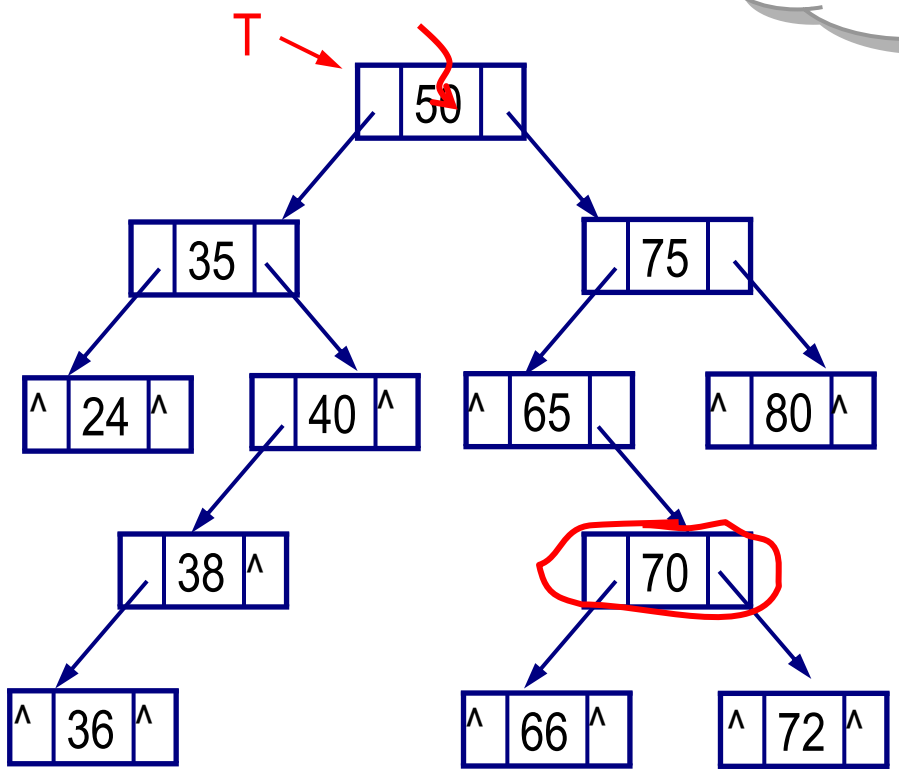


图片来源：BTree和B+Tree详解_b+tree 结构_-CSDN博客
(https://blog.csdn.net/yin767833376/article/details/81511377)

7.2.2 B-树的查找

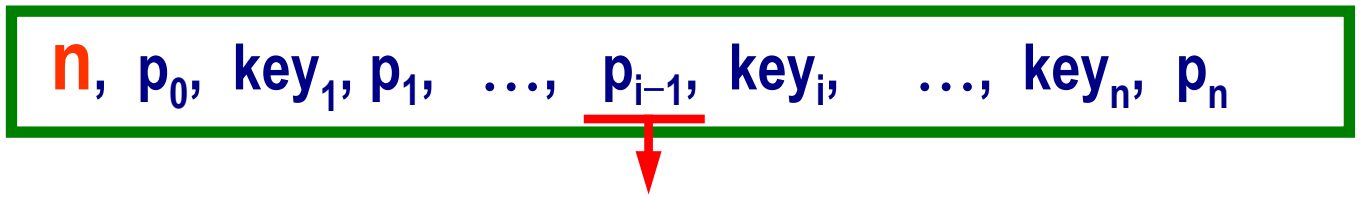
查找 key=70

类似于二叉排序树的查找

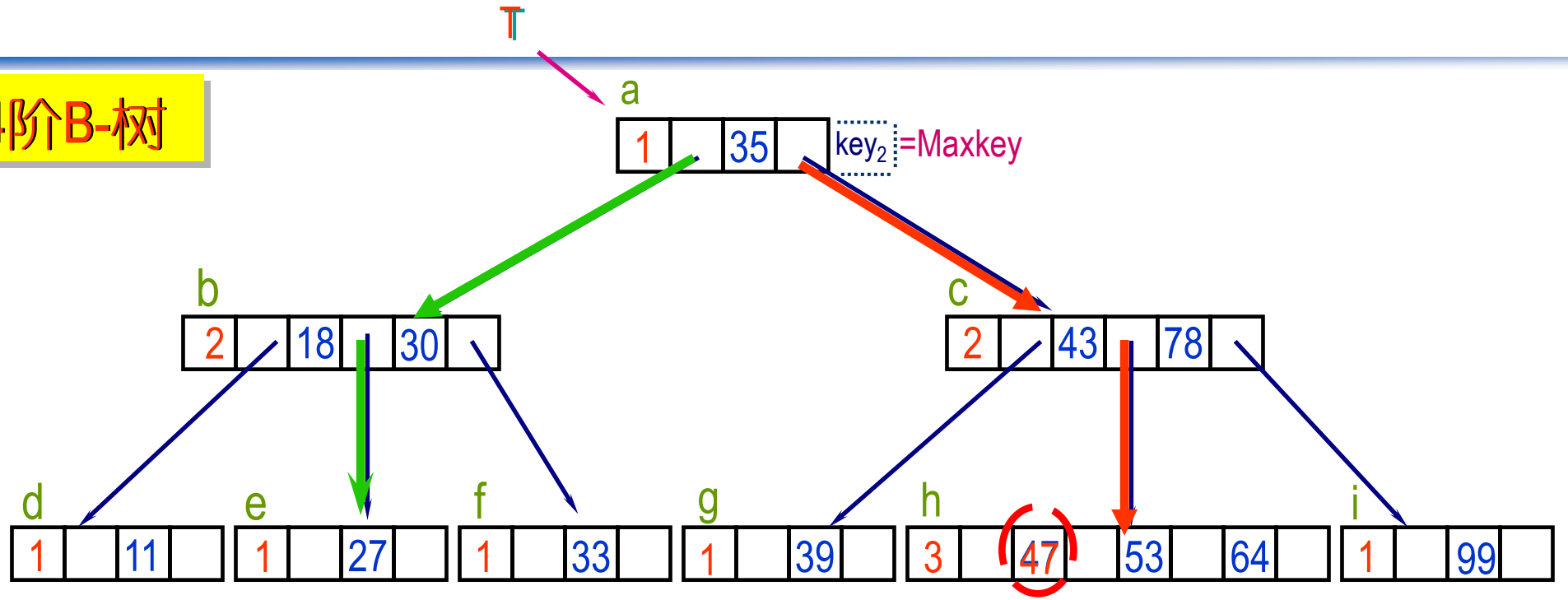


首先将给定的关键字k在B-树根结点的关键字集合中采用顺序查找法 或者折半查找法 进行查找，

- 若$k=key_i$, 则查找成功，根据相应的指针取得记录.否则，
- 若$k<key_i$,则在指针p_{i-1}所指的结点中重复上述查找过程,直到查找成功，或者有$p_{i-1}=NULL$，查找失败。



4阶B-树



例如，查找关键字值k=47　　查找成功 ！

例如，查找关键字值k=23　　查找失败 ！

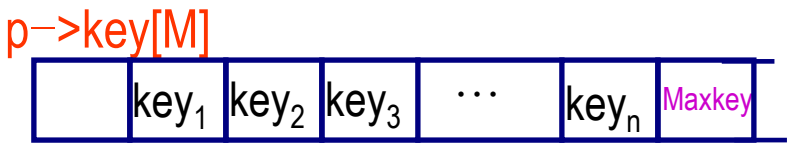
原则
(1) $k=key_i$　查找成功
(2) $k<key_i$　在p_{i-1}所指的结点中查找

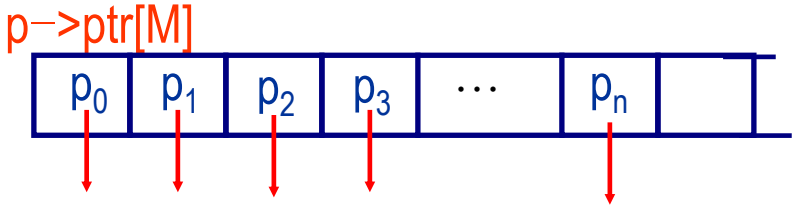
类型定义

```
#define M     1000
typedef struct node {
      int  keynum;
      keytype  key[M+1];        //多一个Key用来记录Maxkey
      struct node  *ptr[M+1];
      rectype  *recptr[M+1];  //指向数据记录
} BTNode;
```

```
keytype  searchBTree(BTNode *t,keytype k){
    int i,n;
    BTNode *p=t;

    while(p!=NULL){
        n=p->keynum;
        p->key[n+1]=Maxkey;

        i=1;
        while(k>p->key[i])
            i++;

        if(p->key[i]==k)
            return  p->key[i];
        else
            p=p->ptr[i-1];
    }
    return  -1;
}
```

在p指结点的关键字集合中查找k



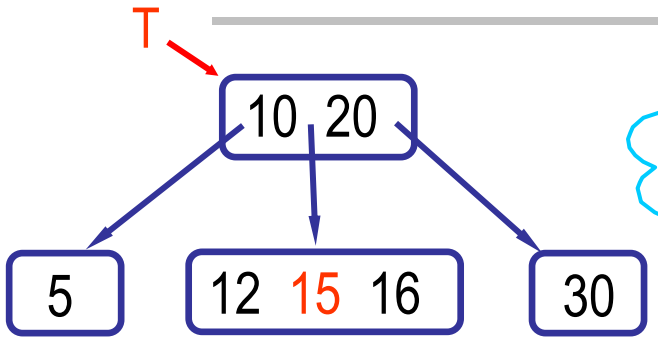


7.2.3 B-树的插入

一棵3阶B-树

B-树的生成从空树开始，即逐个在叶结点中插入结点(关键字)而得到

插入 k=15



一棵m阶B-树的结点中最多有m-1个关键字值

基本思想

若将k插入到某结点后使得该结点中关键字值数目超过m-1时，则要以该结点**位置居中**的那个关键字值为界将该结点**一分为二**，产生一个新结点，并把位置居中的那个关键字值插入到双亲结点中；

如双亲结点也出现上述情况，则需要再次进行分裂．最坏情况下，需要一直分裂到根结点，以致于使得B-树的深度加1.

一般情况下

若某结点已有m−1个关键字值，在该结点中插入一个新的关键字值，使得该结点内容为

q

m	key_1	key_2	key_3	…	key_i	key_{i+1}	…	key_{m-1}	key_m

则需要将该结点分解为两个结点q与q′，即

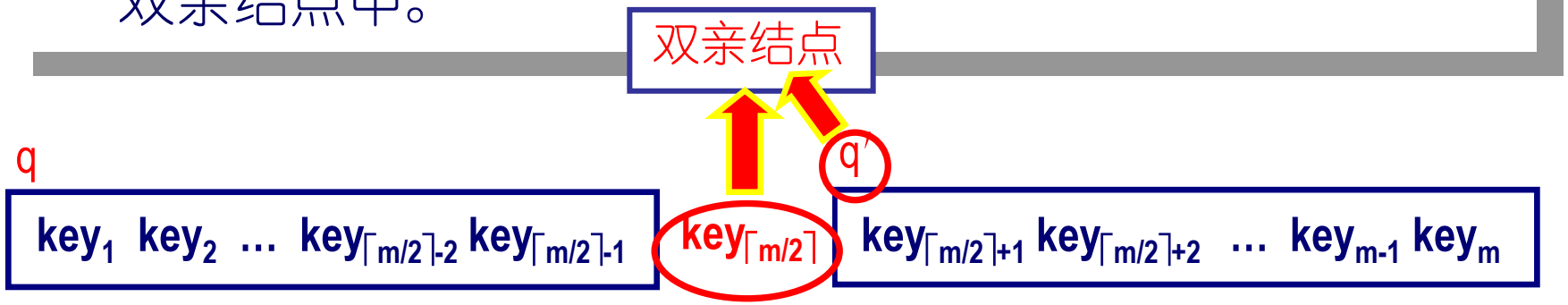
q

$\lceil m/2 \rceil - 1$	key_1	key_2	…	$key_{\lceil m/2 \rceil - 2}$	$key_{\lceil m/2 \rceil - 1}$

q′

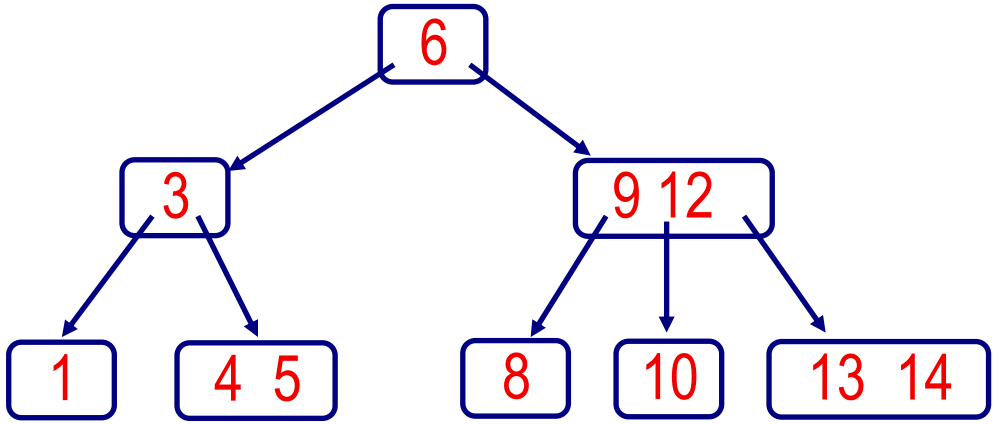
$m - \lceil m/2 \rceil$	$key_{\lceil m/2 \rceil + 1}$	$key_{\lceil m/2 \rceil + 2}$	…	key_{m-1}	key_m

并且将关键字值$key_{\lceil m/2 \rceil}$与一个指向q′的指针插入到q的双亲结点中。



练习

请画出依次插入关键字序列(5, 6, 9, 13, 8, 1, 12, 14, 10, 4, 3)中各关键字值以后的4阶B-树。



原则

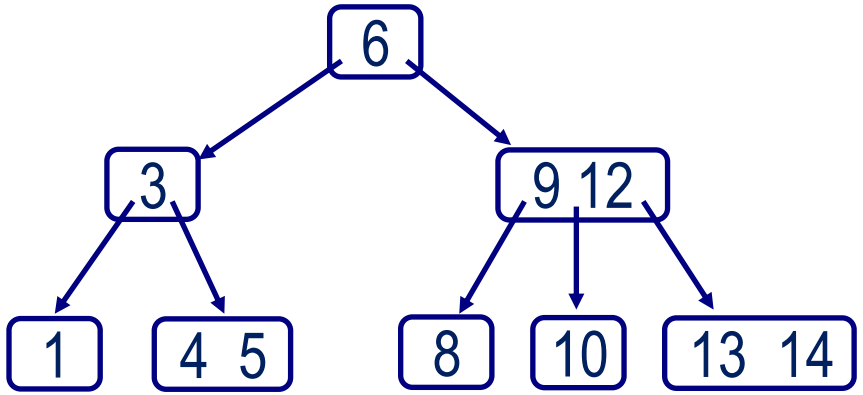
1. 4阶B-树的每个分支结点中关键字个数不能超过3；
2. 生成B-树从空树开始，逐个插入关键字而得到的；
3. 每次在最下面一层的某个分支结点中添加一个关键字;若添加后该分支结点中关键字个数不超过3,则本次插入成功，否则，进行结点分裂。

B-树与平衡二叉树

- **B-树是平衡二叉树的泛化。**当数据规模大需要存储在外存时，平衡二叉树由于深度过大，造成磁盘IO次数多，效率低；而B树通过降低树的高度，可以减少IO次数
- 设计上可以让**B-树结点的大小**和磁盘扇区/块的大小相适应，一次IO读取整个块，提高访问效率
- B-树所有叶子结点在同一层
- B-树节点的关键字数量在一定范围之内，因此不需要象平衡二叉树那样经常的重新平衡
- 插入结点时，B-树通过分裂保持树的平衡，而平衡二叉树通过和旋转(4种情况)保持平衡

B-树的不足

- Deletion process is complicated.
  - Eg., delete 3 or 9



- The search process may take a longer time because all the keys are not obtainable at the leaf.
  - Eg., search keys in [4, 9]

7.3 B+树

一个m阶的B+树为满足下列条件的m叉树：

同B-树

(1) 每个分支结点最多有m 棵子树；

(2) 除根结点外，每个分支结点最少有$\lceil m/2 \rceil$棵子树；

(3) 根结点最少有2棵子树 (除非根为叶结点结点,此时B+树只有一个结点)；

结点内容(和功能)有不同

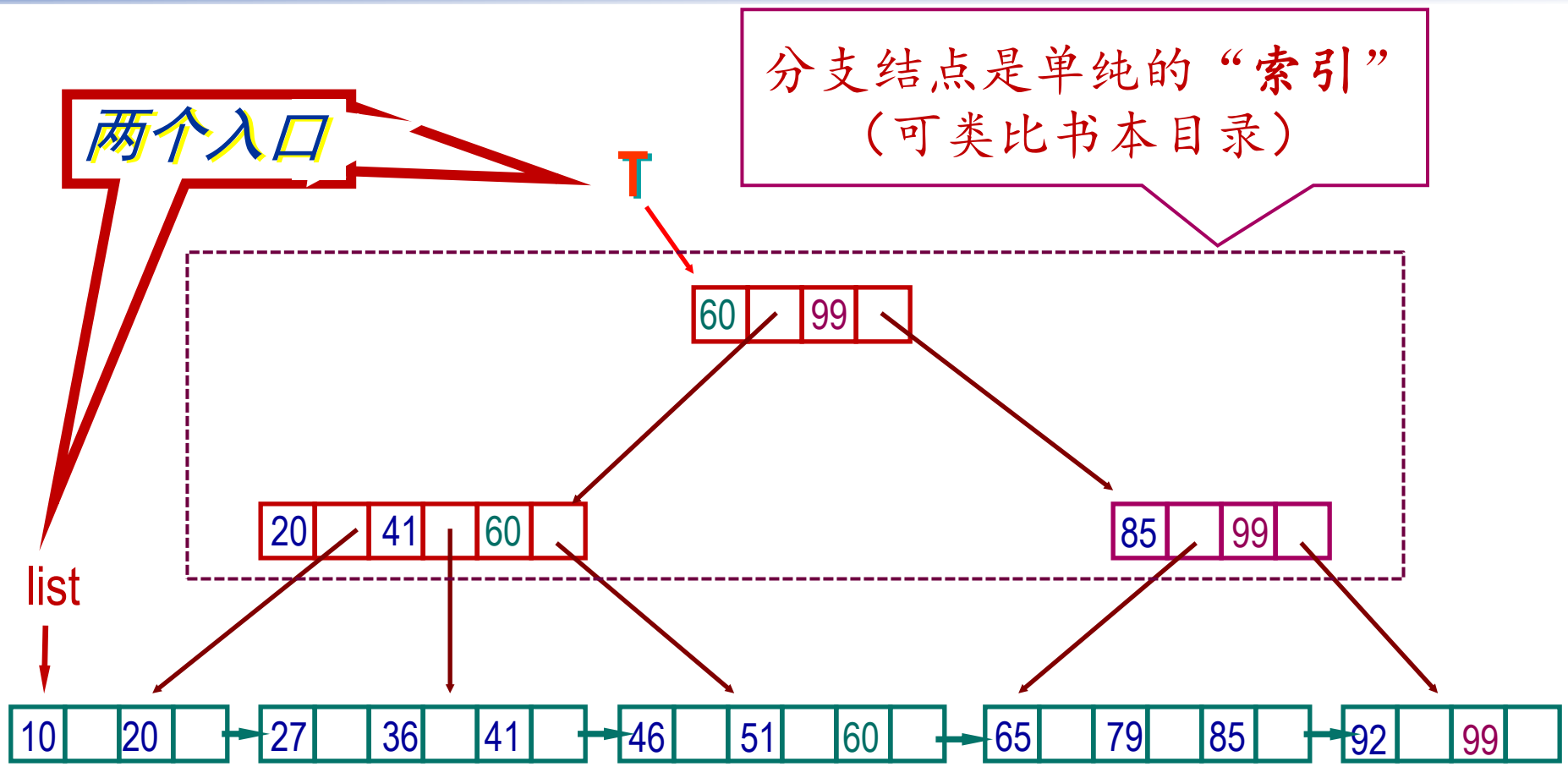
(4) 叶结点中存放记录的关键字以及指向记录的指针,或数据分块后每块的最大关键字值及指向该块的指针，并且叶结点按关键字值的大小顺序链接成线性链表。

(5) 所有分支结点可以看成是索引的索引，结点中仅包含它的各个孩子结点中最大(或最小)关键字值和指向孩子结点的指针。

(6) 结点中有 n 个关键字：

key_1	p_1	key_2	p_2	……	key_n	p_n

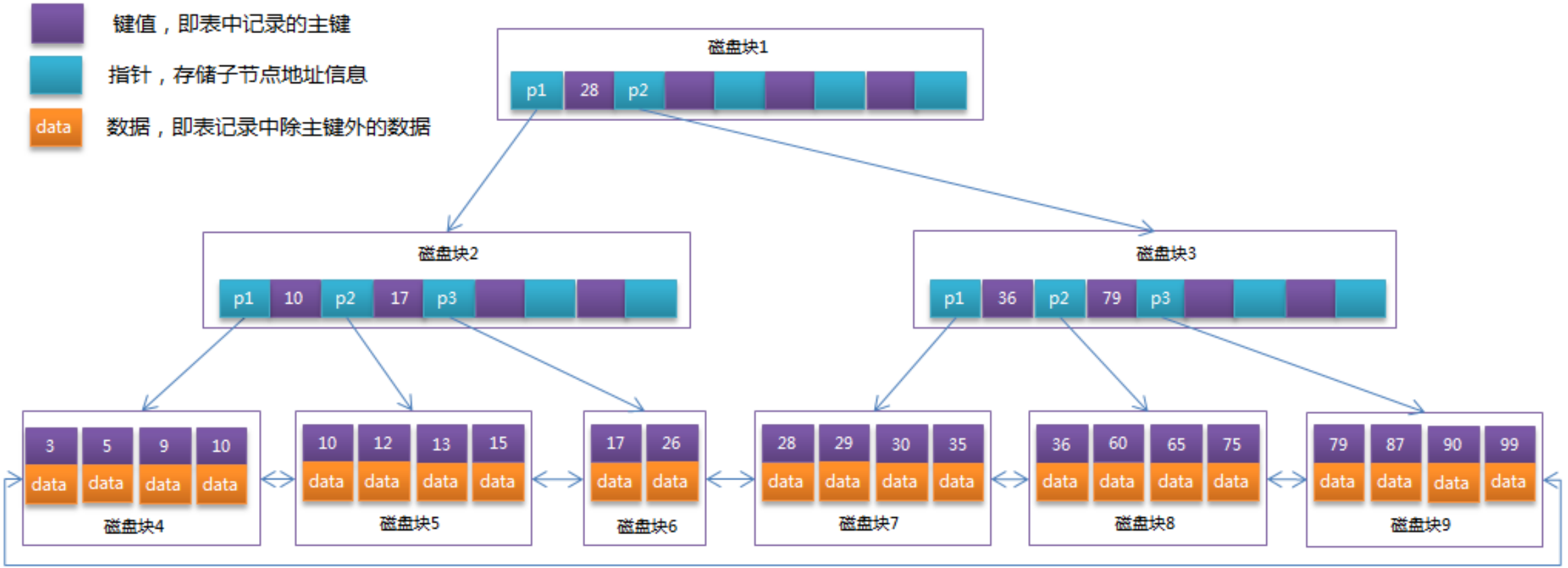
一棵3阶B+树



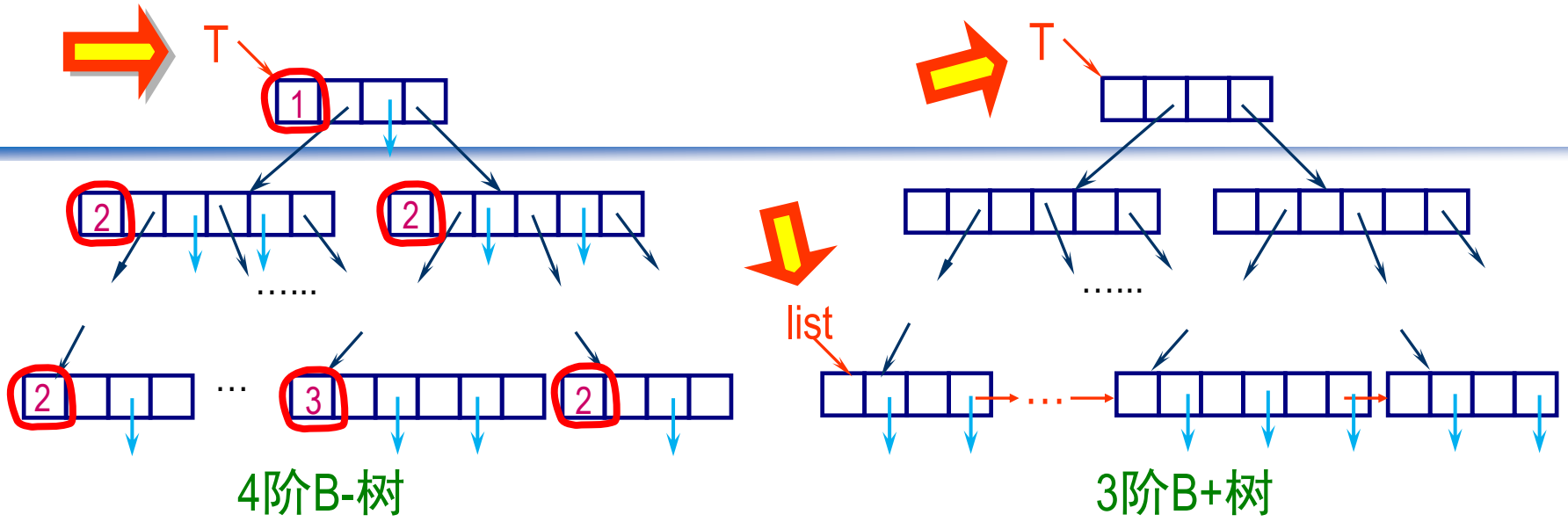
只有叶结点包含数据记录或指向相应记录的指针

B+树用于磁盘数据管理*

文件系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位，大小一般为4K, 8K或16K



图片来源：BTree和B+Tree详解_b+tree 结构_-CSDN博客
(https://blog.csdn.net/yin767833376/article/details/81511377)



-分支可以存更多Key；
-删除发生在叶子结点

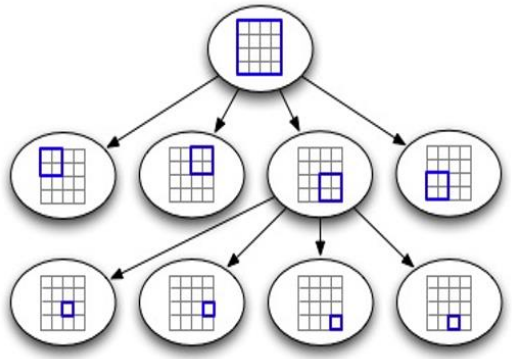
B-树与B+树的区别（从结构上看）

1. B-树的每个分支结点中含有该结点中关键字值的个数,B+树没有；
2. B-树的每个分支结点中含有指向关键字值对应记录的指针,而B+树只有叶结点有指向关键字值对应记录的指针；
3. B-树只有一个指向根结点的入口，而B+树的叶结点被链接成为一个不等长的链表，因此，B+树有两个入口，一个指向根结点，另一个指向最左边的叶结点(即最小关键字所在的叶结点)。

容易实现高效的范围查询(Range Query)

扩展知识*：空间划分树

- ***k*-d树**：每个节点都为k维点的二叉树。所有非叶子节点可以视作用一个超平面把空间分割成两个半空间。
- **四叉树和八叉树**：针对二（三）维空间，在每一个节点上有四（八）个子区块
- **R树(R-Tree，空间索引树)**：是B树向多维空间发展的另一种形式，也是平衡树。它将对象空间按范围划分，每个结点都对应一个区域和一个磁盘页，非叶结点的磁盘页中存储其所有子结点的区域范围，非叶结点的 所有子结点的区域都落在它的区域范围之内；叶结点的磁盘页中存储其区域范围之内的所有空间对象的外接矩形。R树的扩展有R+, R*, QR, RT树等



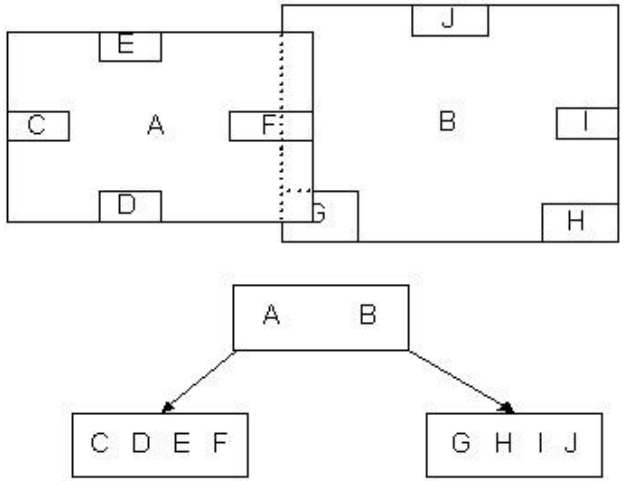


图1：一个R树实例

扩展知识：

索引是现代数据库系统不可缺少的组成部分，索引技术是数据库领域的一个重要分支。

索引的代价：额外的空间和时间开销，以及索引信息和数据项信息的一致性问题等。

延伸阅读*：

倒排索引（inverted index）是目前搜索引擎中常用的搜索技术。

请同学自学有关倒排索引的基本原理。

数据在存储介质上的组织方式

1. 连续组织方式(顺序组织方式)
2. 链接组织方式
3. 索引组织方式
4. **散列**组织方式

   散列函数
   冲突消解：开放地址、链地址、再散列

顺序查找　　*索引查找*　　*散列查找*

7.3 散列(Hash)查找

一、散列查找的基本概念

顺序查找、折半查找以及多数基于树的查找，都是基于(关键字)比较的查找方法

学 号	姓 名	年龄	…
99001	王 亮	17	…
99002	张 云	18	…
99003	李海民	20	…
99004	刘志军	19	…
…	…		
99049	周 颖	18	…
99050	罗 杰	16	…

查找的时间效率主要取决于查找过程中进行的关键字比较次数

能否有一种不经过任何关键字值的比较或者经过很少次的关键字值的比较就能够达到目的方法？

答案是肯定的。

但是需要建立记录的关键字与记录的存储位置之间的映射关系。

1. 散列函数

$$A = H(k)$$

其中，

k　为记录的关键字，

H(k) 称为散列函数，或哈希(Hash)函数，或杂凑函数。

A　为k对应的记录在查找表中位置。

例1

关键字

学号	姓名	性别	…
99001	张 云	女	…
99002	王 民	男	…
99003	李 军	男	…
99004	汪 敏	女	…
……	…	…	…
99030	刘小春	男	…

1	张 云 …
2	王 民 …
3	李 军 …
4	汪 敏 …
:	……
:	
30	刘小春 …

“相对”地址范围：[1..30]

散列函数： H(k) = k–99000

注意：散列函数可能会造成地址冲突！

例2

学 号	姓名	性别	…
99001	张云	女	…
99002	王 民	男	…
99003	李 军	男	…
99004	汪 敏	女	…
……	… …		..
99030	刘小春	男	…

1	李 军 …
2	张 云 …
3	
4	王汪民敏…
:	……
:	
30	

地址冲突

需选择一种处理冲突的方法

地址范围：[1..30]

散列函数：

H(k) = “将组成关键字k的串转换为一个1-30之间的代码值”

H(张云)=2
H(王民)=4
H(李军)=1
H(汪敏)=4

2.散列冲突

对于不同的关键字k_i与k_j，经过散列得到相同的散列地址，即有

$$H(k_i) = H(k_j)$$

这种现象称为散列冲突。

称k_i与k_j为“同义词”

3.什么是散列表

根据构造的散列函数与处理冲突的方法将一组关键字映射到一个有限的连续地址集合上，按照这种方法组织起来表称为散列表,或 哈希表，或杂凑表；建立表的过程称为哈希造表或者散列，得到的存储位置称为散列地址或者杂凑地址。

二、散列函数的构造

1. 原 则

- 散列函数的定义域必须包括将要存储的全部关键字；若散列表允许有m个位置时，则函数的值域为[0 .. m–1](地址空间)。
- 利用散列函数计算出来的地址应能尽可能均匀分布在整个地址空间中。
- 散列函数应该尽可能简单，应该在较短的时间内计算出结果。

一个“好”的散列函数

2. 建立散列表的步骤

- 确定散列的地址空间(地址范围)；
- 构造合适的散列函数；
- 选择处理冲突的方法。

3. 散列函数的构造方法

1. 直接定址法
2. 数字分析法
3. 平方取中法
4. 叠加法
5. 基数转换法
6. 除留余数法

一般形式

$H(k)=ak+b$

如：$H(k)=k-99000$

缺陷：对Key的范围和分布的要求高

除留余数法

$$H(k) = k \ MOD \ p$$

其中，若 m 为地址范围大小，或称表长，
则 p 可为小于等于m的素数。

例：某散列表的长度为100，散列函数H(k)=k mod P, 则通常情况下P最好选择下列【】哪个?

A、91 B、93 C、97 D、99

实践证明，当P取小于哈希表长的最大质数时，产生的哈希函数较好，即冲突较少。以上的97是离长度值最近的最大质数。

为便于讨论，后续内容将H简化为：$H(k) = k \ MOD \ m$

三、冲突的处理方法

所谓处理冲突,是在发生冲突时,为冲突的元素找到另一个散列地址以存放该元素。如果找到的地址仍然发生冲突，则继续寻找，直到不再发生冲突。

1.开放地址法

闭散列方法

所谓开放地址法是在散列表中的“空”地址向处理冲突开放。即当散列表未满时，处理冲突需要的“下一个”地址在该散列表中解决。给定关键字k及其第i次寻址：

$$D_i = (H(k) + d_i) \text{ MOD } m \qquad i=1, 2, 3, \ldots$$

其中，H(k)为哈希函数，m为表长，d_i为地址增量，有：

(1) $d_i=1, 2, 3, \ldots, m-1$　　称为线性探测再散列

(2) $d_i=1^2, -1^2, 2^2, -2^2, \ldots,$　　称为二次探测再散列

(3) d_i=伪随机数序列　　称为伪随机再散列

例3

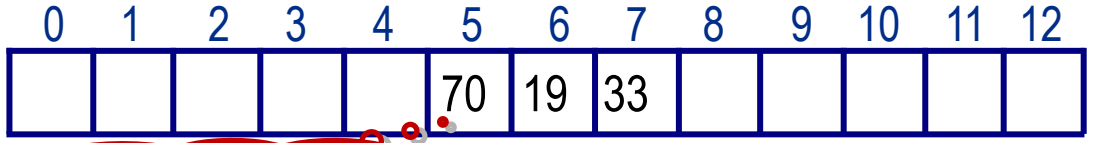
设散列函数为

除留余数法

$$H(k) = k \ \ MOD \ \ 13$$

散列表为[0..12],表中已分别有关键字为19,70,33的记录，现将第四个记录(关键字值为18)插入散列表中。

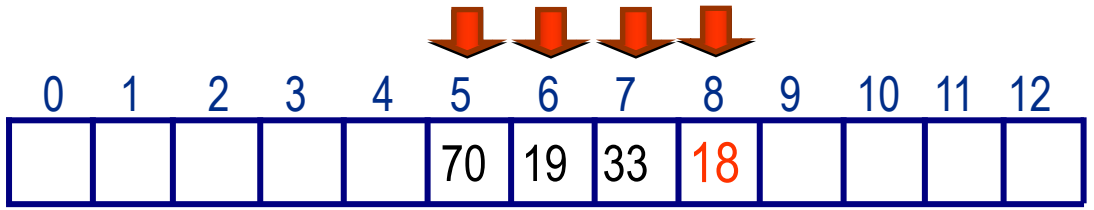
插入前



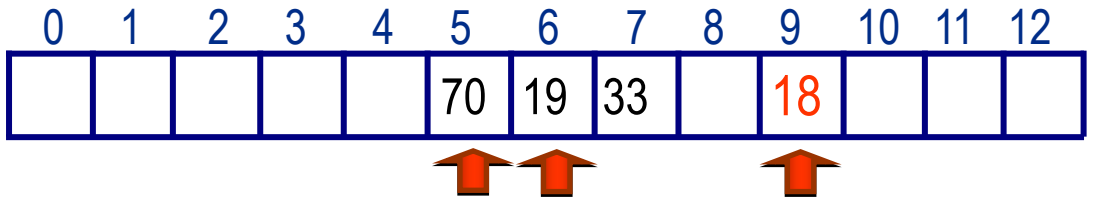
18的散列地址为5

开放地址法： $D_i = (k \ \ MOD \ \ 13 + d_i) \ \ MOD \ \ 13$

d_i采用线性再散列



d_i采用二次再散列



练习

散列函数： $H(k)=k \ MOD \ 13$

采用线性探测再散列方法处理冲突：

$$D_i = (k \ MOD \ 13 + d_i) \ MOD \ 13$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
HT:	13					70	19	33					25

依次插入 key=18 key=38 key=20

	0	1	2	3	4	5	6	7	8	9	10	11	12
HT:	13	38				70	19	33	18	20			25

问题：可否直接删除元素？如19.

聚集 —— 散列地址不同的元素争夺同一个后继散列地址的现象。

产生聚集的主要原因

1. 散列函数选择得不合适；
2. 负载因子（装填因子）过大。

负载因子 —— 衡量散列表的 饱满程度 。

$$\alpha = \frac{\text{散列表中实际存入的元素数}}{\text{散列表中基本区的最大容量}}$$

一般情况下，$\alpha<1$，
α越大，散列表越满

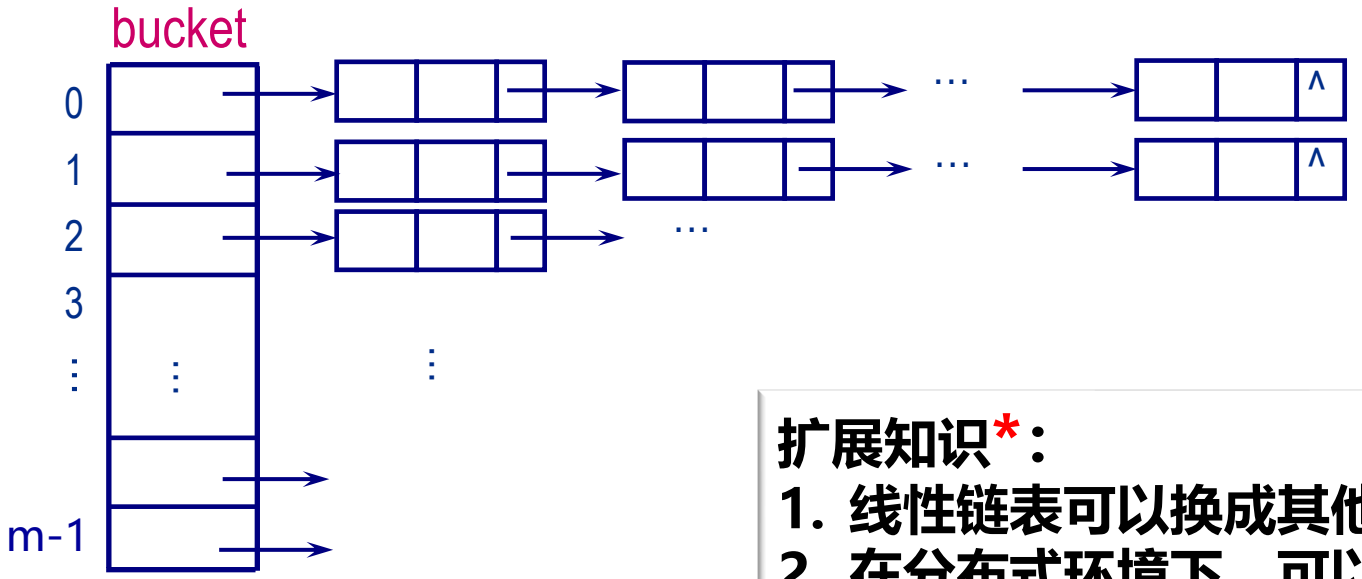
冲突及冲突处理小结

- 冲突会降低哈希表的查找性能。
- “线性探测法”容易产生元素“聚集”的问题。
- “二次探测法”可以较好地避免元素“聚集”的问题，但不能探测到表中的所有元素(至少可以探测到表中的一半元素)。
- 只能对表项进行逻辑删除(如做删除标记)，而不能进行物理删除。使得表面上看起来很满的散列表实际上存在许多未用位置。

2.链地址法

将所有散列地址相同的记录链接成一个线性链表。

若散列范围为[0..m-1],则定义指针数组bucket[0..m-1]分别存放m个链表的头指针。



扩展知识*：

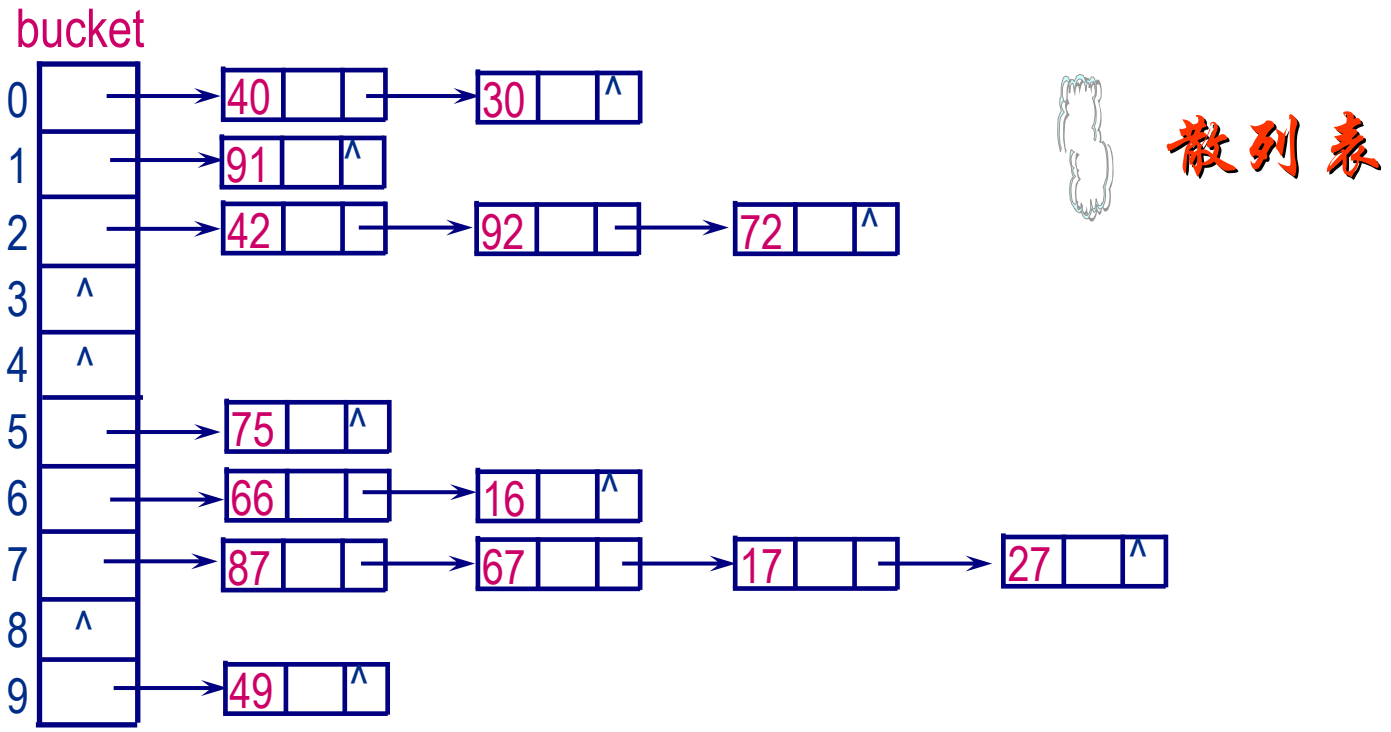
1. **线性链表可以换成其他数据结构；**
2. **在分布式环境下，可以首先利用哈希函数划分数据，然后在存储结点保存数据**

例5

设散列函数为

$$H(k) = k \ \ MOD \ \ 10$$

散列表为[0..9],采用链地址法处理冲突，画出关键字序列{75,66,42,192,91,40,49,87,67,16,17,30,72,27}对应的记录插入散列表后的散列文件。



散列表

特点

- 处理冲突简单，不会产生元素“聚集”现象，平均查找长度较小 。
- 适合建立散列表之前难以确定表长的情况 。
- 建立的散列表中进行删除操作简单 。
- 由于指针域需占用额外空间，当规模较小时，不如“开放地址法”节省空间。

3.再散列法

$$D_i = H_i(k) \qquad i=1, 2, 3, \ldots$$

其中，$H_i(k)$为第i个散列函数，D_i为相应的散列地址

延伸阅读*：Bloom Filter (BF)

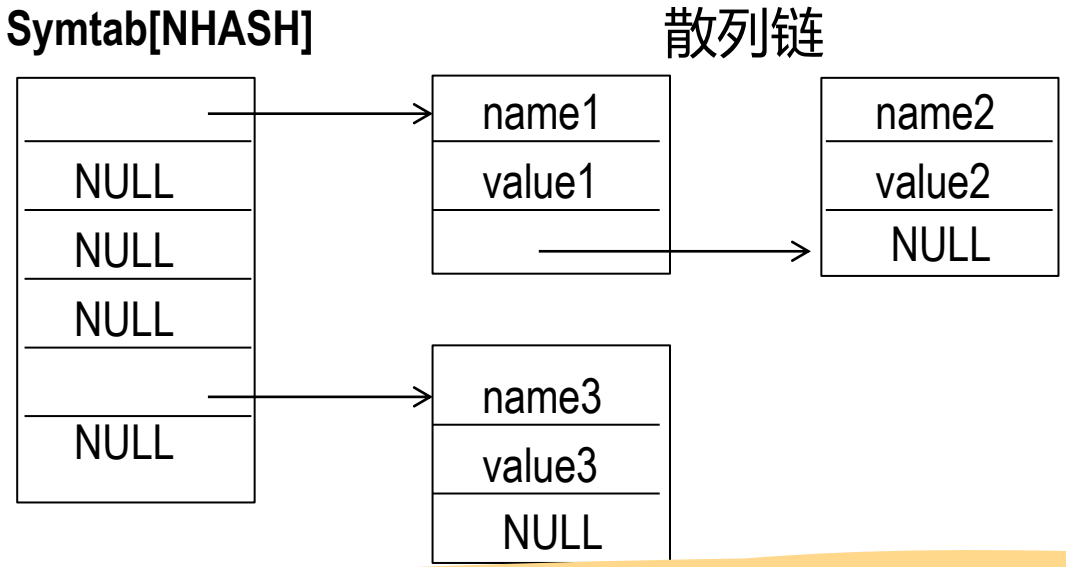
BF利用多个哈希函数，将一个元素映射到多个比特位中，用于快速判断数据元素是否存在。

延伸阅读*：

除了散列表和BF，Hash方法还应用于数据匿名化、数据摘要和数字签名等

散列表的典型应用*

散列表的一个典型应用是**符号表（symbol）**，它是**编译系统**中主要的数据结构，用于管理用户程序中各个标识符的信息，如其名字、数据类型、作用域、存储分配信息等。



此外，散列表还常用于浏览器中维持最近使用的页面踪迹、缓存最近使用过的域名及它们的IP地址。

散列表是计算机科学里的一个伟大发明，它是由数组、链表和一些数学方法相结合，构造起来的一种能够高效支持动态数据的存储和查找的结构，在程序设计中经常使用。

词频统计-利用散列查找提高链表实现查找效率

- **要求：读取文章中的单词，统计每个单词出现的频率，利用哈希表存储单词，输出哈希表项中前5项存储的单词**
- **解题要点：**
  - **哈希函数：**使用字符串哈希算法，根据字符串的内容计算哈希值
    - 几个字符串散列函数：https://www.cnblogs.com/dongsheng/articles/2637025.html
  - **冲突处理：**采用链地址法，哈希值相同的单词存放在一个链表中

词频统计：哈希表实现（头部信息）

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define WORDLEN 100 //单词长度
#define TABLELEN 100005 //哈希表长度
// 哈希表项结构，采用链地址法处理冲突
typedef struct _WordTable{
    char word[WORDLEN];
    int count;
    struct _WordTable* link; // 如果冲突，则放入链表后面
} WordTable, * PWordTable;
// 初始化哈希表为空
PWordTable wordTable[TABLELEN] = { NULL };
// 字符串哈希函数，限定哈希值范围0~size内
unsigned int SDBMHash(char* str, unsigned int size);
// 从文件中读入单词，返回1表示正确读入一个单词，返回0表示结束
int getWord(FILE* fp, char word[]);
// 哈希表中查找单词，如果不存在则插入，返回单词的位置
PWordTable searchAndInsertWord(char word[]);
```

词频统计：哈希表实现-字符串哈希函数

```
// 字符串哈希函数，哈希值映射到0~size之间
unsigned int SDBMHash(char* str, unsigned int size)
{
    unsigned int hash = 0;
    while (*str)
    {
        hash = (*str++) + (hash << 6) + (hash << 16) - hash;
    }
    return (hash % size);
}
```

词频统计：哈希表实现-哈希表的更新和插入

```
PWordTable  searchAndInsertWord(char word[]){
    int hash = SDBMHash(word, TABLELEN); // 计算将字符串哈希值
    PWordTable p = wordTable[hash];      // 根据哈希值得到位置
    PWordTable r = p; // 指向哈希表项冲突链表的表尾元素
    while (p != NULL) {
        if (strcmp(p->word, word) == 0)
        { // 找到位置，如果单词相等，则次数+1，再返回该位置
            p->count++; return p;
        }
        else { r = p;  p = p->link; } // 下一个位置
    }
    // 没找到，则将单词插入到当前哈希表的最后位置
    p = (PWordTable)malloc(sizeof(WordTable));
    strcpy(p->word, word);
    p->count = 1;  p->link = NULL;
    if (r == NULL) // 当前哈希表项没有元素，则直接插入首位置
        wordTable[hash] = p;
    else r->link = p; // 否则有冲突，接到表尾
    return r;
}
```

词频统计：哈希表实现-主函数

```
int main() {
    char filename[WORDLEN], word[WORDLEN];
    FILE* fp; int i, count;
    PWordTable p;
    scanf("%s", filename);
    // 打开文件，如果失败，则退出
    fp = fopen(filename, "r");
    if (fp == NULL){ perror("Can't open the file!\n");  return -1;}
    // 读取文件中的单词
    while (getWord(fp, word)) //读取一个单词
        searchAndInsertWord(word); //查询或者插入新单词，已有单词数量加1
    // 输出哈表的前5个非空表项中的单词，同一个表项中的单词在同一行输出
    for (i = 0, count = 0; i < TABLELEN && count < 5; i++) {
        p = wordTable[i];
        if (p != NULL) {//输出表项数+1
            if (count > 0) printf("\n\n");
            count++;     }
        while (p != NULL){//输出某个非空项中存储的单词
            printf("%s %d ", p->word, p->count); p = p->link; }
    } return 0;
}
```

词频统计：哈希表实现-读单词函数

```
int getWord(FILE* fp, char word[]) {
    int c, i = 0;
    // 忽略前面的非英文字符
    while (!isalpha(c = fgetc(fp)))
        if (c == EOF)  return 0;
    // 将出现的第一个英文字母存入word中
    word[i++] = tolower(c);
    // 继续读取后续字符
    while ((c = fgetc(fp)) != EOF) {
        if (isalpha(c)) // 是字母，继续存入word中
            word[i++] = tolower(c);
        else // 如果不是英文字母，则退出
            break;
    }
    // word最后补\0，确保字符完整
    word[i] = '\0';
    return 1;
}
```

问题：词频统计 –查找性能分析（不同实现）

查找与存储方式	比较次数	平均比较次数	运行时间	说明
顺序查找 + 顺序表（无序）	1,604,647,193	2962.5	7.114s	不需要移动数据，但查找效率低，查找性能为O(N)
顺序查找+链表（有序）	4,151,966,169	7,665.5	97.4s	不需要移动数据，但查找效率低，查找性能为O(N)
顺序查找+链表（无序）	1,604,647,193	2962.5	26.5s	不需要移动数据，但查找效率低，查找性能为O(N)
索引结构 + 链表(有序)	208,620,575	385.1	4.517s	建立26字母开头的单词索引，有效改进了链表查找性能
折半查找 + 顺序表（有序）	6,923,725	12.8	1.103s	需要移动数据，查找性能为$O(\log_2 N)$
二叉查找树	6,768,565	12.5	0.543s	理想情况下（平衡树）查找性能为$O(\log_2 N)$,无数据移动
字典树（Trie）	3,031,958	5.6	0.49s	查找性能与单词规模无关，只与单词平均长度有关
Hash查找（30000大小）	**569,410**	**1.05**	**0.456**	**查找性能与单词规模无关，只与Hash冲突数有关**

数据说明：文本单词总数541,639，不同单词总数22,086

思考

在各种查找方法中，哪种查找法的平均查找长度ASL与元素的个数n无关?

在散列函数与散列地址范围都分别相同的前提下，为什么说采用链地址法处理冲突比采用开放地址法的时间效率高?

顺序查找、折半查找、树型查找和散列查找四种查找方法中，只能在顺序存储结构上进行的查找方法是哪一种?

本章结束！