北京航空航天大学 BEIHANG UNIVERSITY


数据结构与程序设计（信息类）

Data Structure & Programming

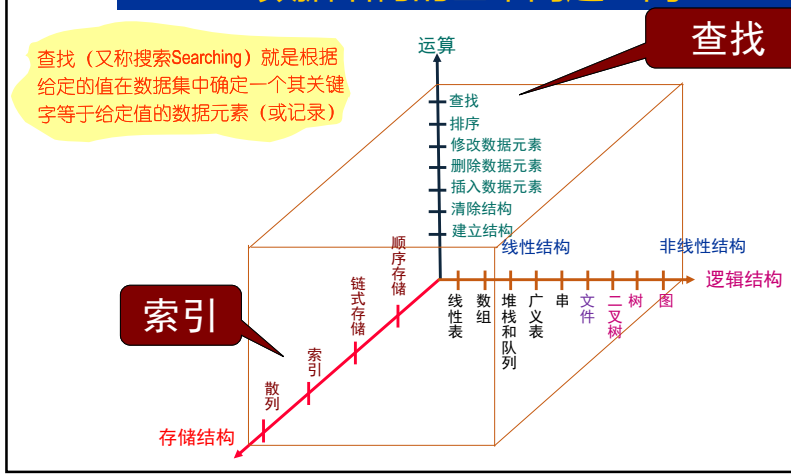
北京航空航天大学 数据结构课程组
软件学院 林广艳
2023年春

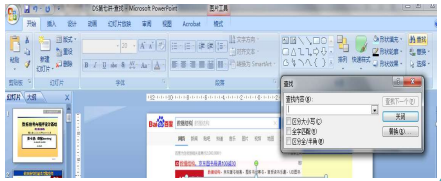
内容提要

1. 查找的基本概念
2. 顺序表的查找
3. 索引基础
4. 二叉查找树(BST)
5. B-树和B+树
6. 散列(Hash)

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）





互联网时代，几乎我们每个人都会要用到搜索。

1.1 查找的基本概念

例

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	… …

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	…
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…

5

名词术语

字段、数据项

- **属性** 描述一个**客体**某一方面特征的数据信息

属性的集合

- **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

次关键字

- **关键字** 区分不同记录的属性或属性组
  （主关键字、次关键字）

主关键字

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	… …

6

查找表的逻辑结构与物理结构

- **逻辑结构**
  - ✓ 记录呈现在用户眼前时排列的先后次序关系。 (线性结构)
- **物理结构**
  - ✓ 查找表（文件）在存储介质上的组织方式。
    - ① 连续组织方式(顺序组织方式) — 顺序查找
    - ② 链接组织方式 — 索引查找
    - ③ 索引组织方式
    - ④ 随机组织方式(散列组织方式) — 散列查找

7

查找表的基本操作

- **查找**是在查找表中确定某个特定记录存在与否的过程
  - ✓ 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
  - ✓ 具体操作，如
    - 查找表的第i个记录
    - 查找当前位置的下一个记录
    - 按关键字值查找记录
    - ……
- **插入、删除、修改，查找操作**均以查找操作为基础
- **排序**
  - ✓ **使记录按关键字值有序排列的过程**

8

静态查找表与动态查找表

- **静态查找表**(Static Search Table)
  - ✓ 如果只在查找表中确定某个特定记录是否存在或检索某个特定记录的属性，此类查找表为静态查找表
- **动态查找表**(Dynamic Search Table)
  - ✓ 如果在查找表中需要插入不存在的数据元素（记录）或需要删除检索到的数据元素（记录），此类查找表为动态查找表

显然查找效率与表的组织方式（结构）和类型有关！

1.2 顺序表的查找

- 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** 。

有序连续顺序表

关键字

有序顺序表!

学 号	姓 名	性别	年龄	其他
06001	张 三	女	20	…
06002	李 四	男	17	…
06003	王 五	男	19	…
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
06050	刘 末	女	16	…

关键字

一般顺序表!

内容提要

1. **查找的基本概念**
2. **顺序表的查找**
3. **索引基础**
4. **二叉查找树(BST)**
5. **B-树和B+树**
6. **散列(Hash)**

2.1连续顺序表的查找

◆ 查找思想

从表的第一个记录开始,将用户给出的关键字值与当前被查找记录的关键字值进行比较，若匹配，则查找成功，给出被查到的记录在表中的位置，查找结束。若所有n个记录的关键字值已比较，不存在与用户要查的关键字值匹配的记录，则查找失败，给出信息0。

$(key_1, key_2, key_3, \ldots, key_n)$

k

被查找记录的关键字值

关键字集合

如何计算查找效率?

◆ 平均查找长度ASL(Average Search Length)

- ✓ 确定一个记录在查找表中的位置所需要进行的关键字值的比较次数的期望值(平均值)。
- ✓ 对于具有n个记录的查找表，有 $ASL=\sum_{i=1}^{n} p_i c_i$

其中，p_i为查找第i个记录的概率，c_i为查找第i个记录所进行过的关键字的比较次数

- ✓ 对于具有n个记录的顺序表，若查找概率相等，则有

$$ASL=\sum_{i=1}^{n} p_i c_i = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

算法的时间复杂度为O(n)

算法实现

```
int search(keytype key[], int n, keytype k)
{
    int i;
    for (i = 0; i < n; i++)
        if (key[i] == k)
            return i;
    return -1;
}
```

例

key[0..9]	38	75	19	57	100	48	50	7	62	11
i	0	1	2	3	4	5	6	7	8	9

若查找 k=48　经过6次比较, 查找成功, 返回 i=5

若查找 k=35　查找失败，返回信息 -1

顺序查找法

◆ 优点

- ✓ 查找原理和过程简单，易于理解
- ✓ 对于被查找对象的排列次序没有限制

◆ 缺点

- ✓ 查找的时间效率低

思考： 插入对象的位置对查询效率是否有影响？

- 随机插入
- 在头部插入
- 在尾部插入
- 按顺序插入

2.2有序连续顺序表的折半查找法（二分查找法、 对半查找法）

- 查找思想

将要查找的关键字值与当前查找范围内位置居中的记录的关键字的值进行比较。

若匹配，则查找成功，给出被查到记录在文件中的位置，查找结束。

若要查找的关键字值小于位置居中的记录的关键字值，则到当前查找范围的前半部分重复上述查找过程，否则，到当前查找范围的后半部分重复上述查找过程，直到查找成功或者失败。

若查找失败，则给出错误信息（0）。

2.2 有序连续顺序表的折半查找法（二分查找法、 对半查找法）

- 几个变量
  - ✓ n 排序连续顺序文件中记录的个数
  - ✓ low 当前查找范围内第一个记录在文件中的位置， 初值 low=0
  - ✓ high 当前查找范围内最后那个记录在文件中的位置， 初值 high=n-1
  - ✓ mid 当前查找范围内位置居中的那个记录在文件中的位置。

$$mid = \left\lfloor \frac{low+high}{2} \right\rfloor$$

算法实现：非递归算法

```
int binsearch(keytype key[], int n, keytype k){
    int low = 0, high = n - 1, mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (k == key[mid])
            return mid; /*查找成功*/
        if (k > key[mid])
            low = mid + 1; /*准备查找后半部分 */
        else
            high = mid-1; /*准备查找前半部分 */
    }
    return -1; /*查找失败  */
}
```


算法实现：递归算法

```
int binsearch2(keytype key[], int low, int high, keytype k){
    int mid;
    if (low > high)
        return -1;
    else
    {
        mid = (low + high) / 2;
        if (k == key[mid])
            return mid;
        else if (k < key[mid])
            return binsearch2(key, low, mid-1, k);
        else
            return binsearch2(key, mid + 1, high, k);
    }
}
```

在第1次调用的算法中

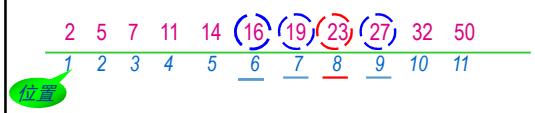
```
low=0;
high=n-1;
pos=binsearch2(KEY, low, high, k);
```

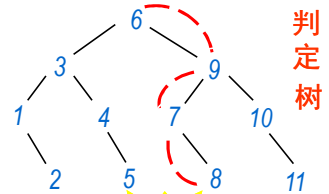
查找效率如何？

如何计算查找效率?

- 判定树

若把当前查找范围内居中的记录的**位置**作为根结点，前半部分与后半部分的记录的**位置**分别构成根结点的左子树与右子树，则由此得到一棵称为**“判定树”**的二叉树，利用它来描述折半查找的过程。





判定树

查找23

成功！

- 成功的查找过程正好等于走了一条从根结点到被查找结点的路径
- 经历的比较次数恰好是被查找结点在二叉树中所处的层次数！

21

2.2 有序连续顺序表的折半查找法

- 优点
  - ✓ 查找原理和过程简单，易于理解
  - ✓ 查找的时间效率较高
- 缺点
  - ✓ 要求查找表中的记录按照关键字值有序排列
  - ✓ 对于查找表，只适用于有序连续顺序表

为了保持数据集为排序顺序数据集，在数据集中插入和删除记录 时需要移动大量的其它记录

折半查找方法适用于一经建立就很少改动、而又经常需要查找的查找表

23

如何计算查找效率?

- 平均查找长度ASL(Average Search Length)
  - ✓ 对于具有n个记录的排序连续顺序文件，若查找概率相等，则有

$$ASL=\sum_{i=1}^{n} p_i c_i = \frac{1}{n}\sum_{j=1}^{h} j\times(2^{j-1}) = \frac{n+1}{n}\log_2(n+1)-1$$

第j层结点数的最大值

第j层每个结点的比较次数

  - ✓ 当n足够大时，有

$$ASL \approx \log_2(n+1)-1$$

算法的时间复杂度为$O(\log_2 n)$

22

思考

有序连续顺序存储（数组）适合于静态查找表。

在线性表中采用折半查找方法查找数据元素，该线性表应该满足什么条件？

数据元素按值有序排列

必须采用顺序存储结构

24

基于折半查找的元素定位

◆对于动态表，通常元素没有查找到时要进行插入操作，基于折半查找算法，如何获取元素的插入位置?

```
int insertElem(ElemType list[], ElemType item){
    int i = 0, j;

    if (N == MAXSIZE) return -1;
    //折半查找寻找item的合适位置
    i = searchElem(list, item);

    for (j = N - 1; j >= i; j--)
        list[j + 1] = list[j];

    list[i] = item; //将item插入表中
    N++;
    return 1;
}
```

折半查找确定插入位置

```
//折半查找算法，返回插入位置
int searchElem(ElemType list[], ElemType item){
    int low = 0, high = n - 1, mid;
    while (low <= high)    {
        mid = (high + low) / 2;
        if(( item < list[mid])
            high = mid - 1;
        else if ( item > list[mid])
            low = mid + 1;
        else
            return (mid);
    }
    return low;
}
```

使用标准库中的查找函数

◆类似于qsort函数，标准库中提供了一个查找函数，实现按照关键字的查找（stdlib.h）

```
void* bsearch( const void *key, const void *ptr,
               size_t count, size_t size,
               int (*comp)(const void*, const void*) );
```

- ✓ key 指向要查找的元素的指针
- ✓ ptr 指向要检验的数组的指针
- ✓ count 数组的元素数目
- ✓ size 数组每个元素的字节数
- ✓ comp 比较函数。若首个参数小于第二个，则返回负整数值，若首个参数大于第二个，则返回正整数值，若两参数相等，则返回零。 将 key 传给首个参数，数组中的元素传给第二个

27

插值查找（Interpolation Search）*

◆对于有序顺序表，折半查找时：mid = low + (high-low)/2

◆对于有序顺序表，插值查找时：

$$mid = low + (high-low)*(k-a[low])/(a[high]-a[low])$$

延伸阅读*：

折半查找算法效率非常高（时间复杂度仅为$O(\log_2 n)$），针对一些特定的有序集，有没有更快的查找算法呢?

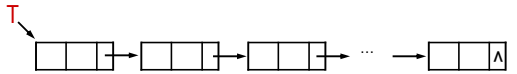
请同学自学有关插值查找(Interpolation Search)及斐波那契查找（Fibonacci Search）算法原理及C实现。

26

2.3 链接顺序表的查找

链结点：

链接顺序表（链表）适合于动态查找表，但查找效率低。




```
struct node *search(struct node *p, keytype k)
{
    for (; p != NULL; p = p->link)
        if (p->key == k)
            return p; /* 查找成功 */

    return NULL; /* 查找失败 */
}
```

```
struct node{
    keytype key;
    rectype rec;
    struct node *link;
};
```

内容提要

1. 查找的基本概念
2. 顺序表的查找
3. 索引基础
4. 二叉查找树(BST)
5. B-树和B+树
6. 散列(Hash)

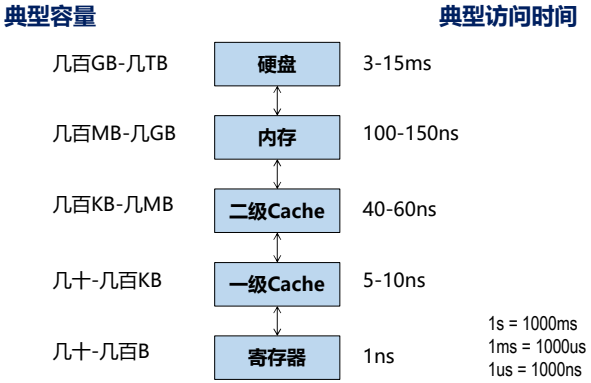
3. 索引(Index)基础

如何在大规模数据集中快速查找？

如何根据不同属性查找？

如何利用不同存储介质的性能特性实现快速查找？

Memory Hierarchy

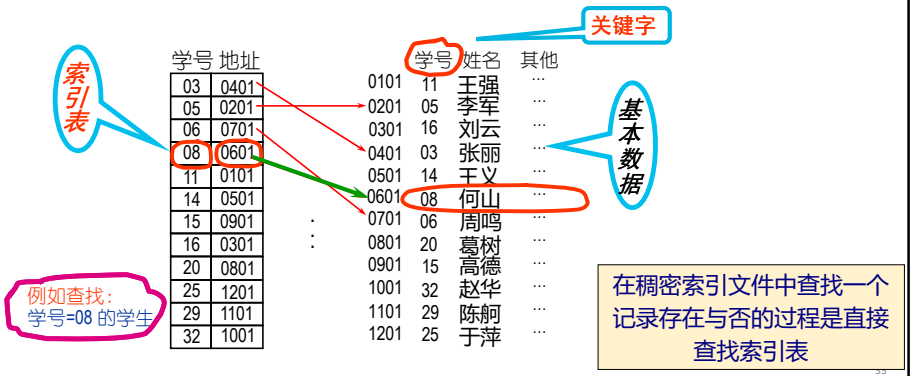


基本概念

- 索引
  - ✓ 记录关键字值与记录的存储位置之间的对应关系
- 索引文件
  - ✓ 由基本数据与索引表两部分组成的数据集称为索引文件
- 索引表的特点
  - ✓ 索引表是由系统自动产生的
  - ✓ 索引表中表项按关键字值有序排列

3.1 稠密索引

- 特点是基本数据中的每一个记录在索引表中都占有一项。



3.2 非稠密索引-分块索引

- 在非稠密索引(分块)文件中查找一个记录存在与否的过程是：
  - ✓ 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

3.2 非稠密索引-分块索引

- 特点是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

3.3 多级索引

- 特点是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。

二叉排序树多级索引结构、
多分树索引结构

树形结构的多级索引

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。

动态查找表

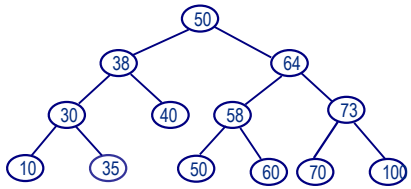
- 若表无序（无论是顺序存储还是链式存储）
  - 查找采用顺序查找方法，元素的插入和删除操作简单，但查找效率低；
- 若表有序
  - 如果采用顺序存储，可用折半查找方法，查找效率高，但插入和删除操作效率低；
  - 若采用链式存储，插入和删除操作效率高，但查找效率低（只能用顺序查找方法）

有没有一种针对动态查找表的数据的组织方式，能够兼顾查找和插入、删除操作的效率？

二叉树排序树（二叉搜索树，Binary Search Tree,BST）

4. 二叉查找（排序）树（BST）

- 采用链式存储，元素插入与删除效率高，同时查找效率通常较高（平衡二叉排序树AVL的查找算法时间复杂度为$O(\log_2 n)$）
- BST特别适合动态查找表的数据组织（如单词词频统计中单词表的构造）



39

内容提要

1. 查找的基本概念
2. 顺序表的查找
3. 索引基础
4. 二叉查找树(BST)
5. B-树和B+树
6. 散列(Hash)

38

二叉查找树的查找和插入算法

功能：在一个二叉查找树中查找某个元素。若该元素不存在，则将节点插入到二叉查找树中的相应位置上。（特别适合动态查找表的构造和查找）

```
BTNodeptr searchBST(BTNodeptr t, Datatype key)
{
    BTNodeptr p = t;
    while (p != NULL) {
        if (key == p->data)
            return p;
        if (key > p->data)
            p = p->rchild;
        else
            p = p->lchild;
    }
    return NULL;
}
```

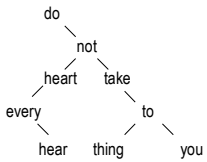
```
BTNodeptr insertBST(BTNodeptr p, Datatype item)
{
    if (p == NULL){
        p = (BTNodeptr)malloc(sizeof(BTNode));
        p->data = item;
        p->lchild = p->rchild = NULL;
    }
    else if (item < p->data)
        p->lchild = insertBST(p->lchild, item);
    else if (item > p->data)
        p->rchild = insertBST(p->rchild, item);
    else
        do-something; //找到该元素
    return p;
}
```

40

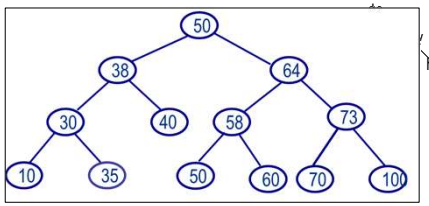
4. 二叉查找（排序）树（BST）

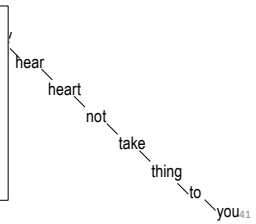
- BST通常不是一棵平衡树，它的树结构与输入数据的顺序有很大的关系，它**很难达到**理想的$O(\log_2 n)$查找性能

输入：do not take to heart every thing you hear



输入：do every hear heart not take thing to you





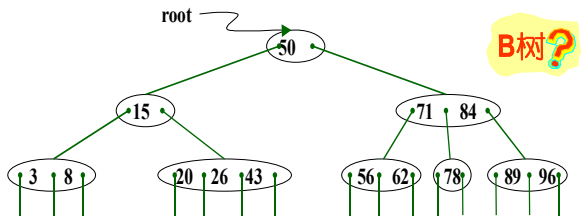
问题

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），如何构造索引，使得以尽可能少的硬盘访问次数，找到所要的数据?

内容提要

1. 查找的基本概念
2. 顺序表的查找
3. 索引基础
4. 二叉查找树(BST)
5. B-树和B+树
6. 散列(Hash)

5. B-树和B+树（多路查找树）



5.1 B-树的定义

◆ **一个m阶的B-树为满足下列条件的m叉树**

(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⌈m/2⌉棵子树

(3) 根结点最少有两棵子树(除非根为叶结点，此时B-树只有一个结点)

(4) 所有"叶结点"都在同一层上，叶结点不包含任何关键字信息（可以把叶结点视为实际上不存在的外部结点，指向这些"叶结点"的指针为空）

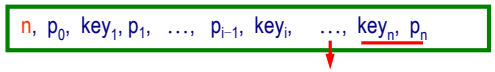
(5) 所有分支结点中包含下列信息:

5. 2 B-树的查找

从根结点出发，沿指针搜索结点和在结点内进行顺序（或折半）查找 两个过程交叉进行。

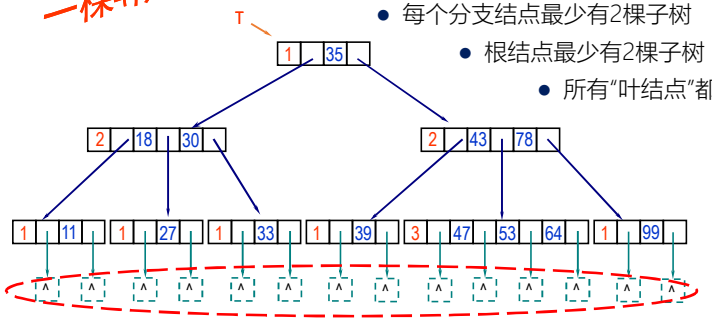
首先将给定的关键字k在B-树的根结点的关键字集合中采用**顺序查找法**或者**折半查找法**进行查找，若有$k=key_i$，则查找成功，根据相应的指针取得记录。否则，若$k<key_i$,则在指针p_{i-1}所指的结点中重复上述查找过程，直到在某结点中查找成功，或者有p_{i-1}=NULL，查找失败。

类似于二叉排序树的查找



一棵4阶B-树

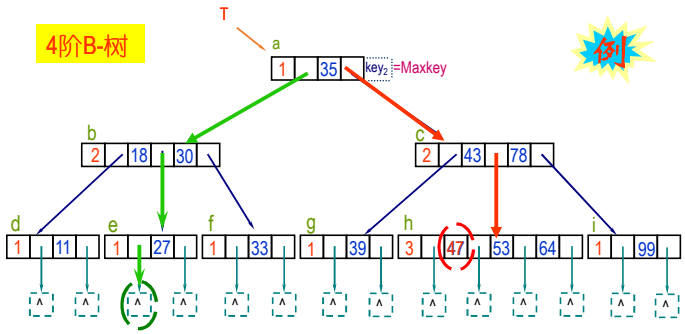
- 每个分支结点最多有4棵子树(即最多有m-1个关键字值)
- 每个分支结点最少有2棵子树
- 根结点最少有2棵子树
- 所有"叶结点"都在同一层上



多叉树、排序性、平衡性

4阶B-树

例



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

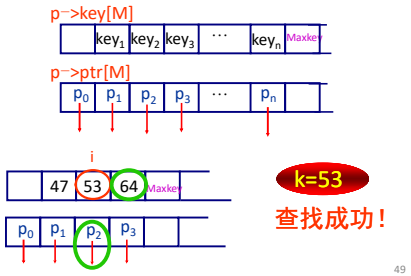
(2) $k<key_i$ 在p_{i-1}所指的结点中查找

B-树的查找过程

```
keytype searchBTree(BNode *t, keytype k){
    int i, n;
    BNode *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

```
#define M 1000
typedef struct node {
      int  keynum;
      keytype  key[M+1];
      struct node  *ptr[M+1];
      rectype  *recptr[M+1];
} BNode;
```



k=62

沿着新的指针p_2继续查找！

k=53

查找成功！

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

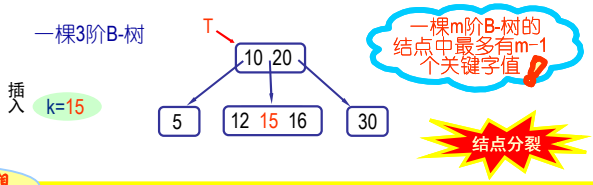
q | $\lceil m/2 \rceil-1$ key_1 key_2 … $key_{\lceil m/2 \rceil-2}$ $key_{\lceil m/2 \rceil-1}$

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。

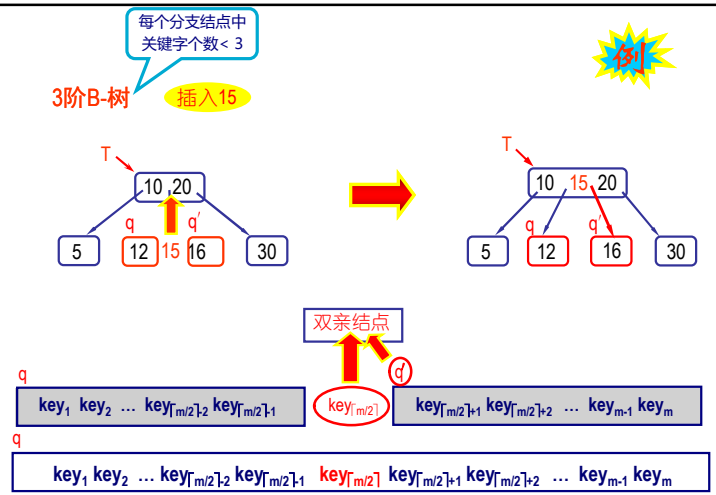
5.3 B-树的插入

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



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。



练习

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

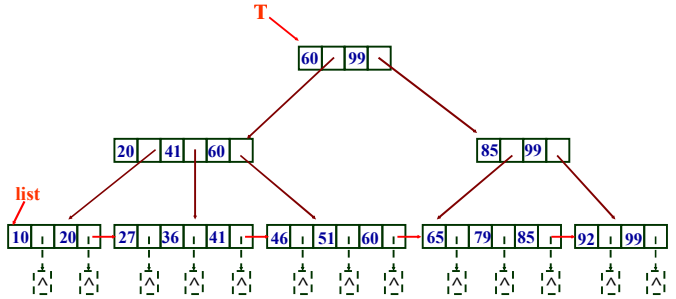
B-树的生成从空树开始，即逐个在叶结点中插入结点(关键字)而得到

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。

5.4 B+树的定义

◆ 在索引文件组织中，经常使用B-树的一些变形，其中B+树是一种应用广泛的变形

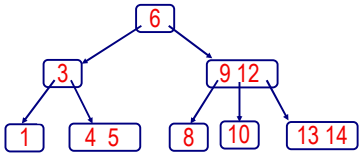


55

（5, 6, 9, 13, 8, 1, 12, 14, 10, 4, 3）

4阶B-树的建立过程

结点分裂



5.4 B+树的定义

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⌈m/2⌉棵子树；

(3) 根结点最少有两棵子树(除非根为叶结点结点,此时B+树只有一个结点)；

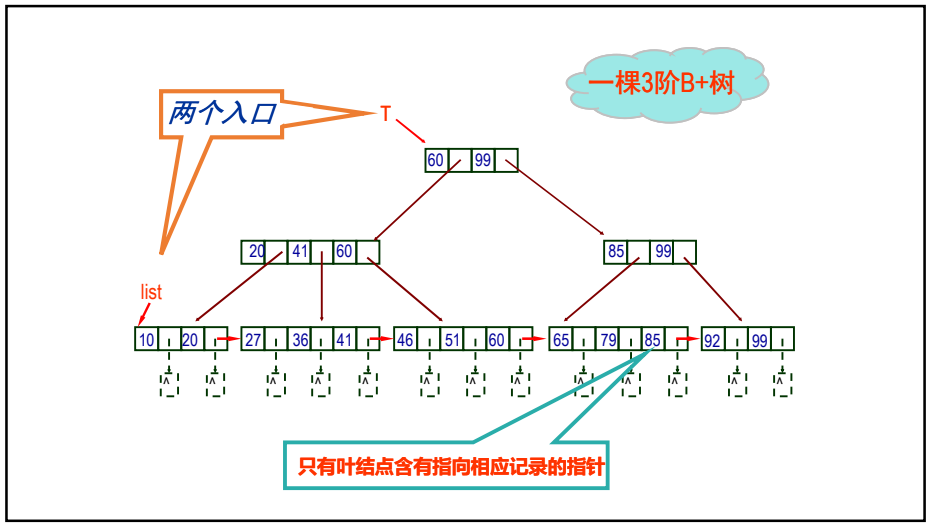
(4) 具有n 棵子树的结点中一定有n 个关键字；

(5) 叶结点中存放记录的关键字以及指向记录的指针,或者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。

key_1 p_1 key_2 p_2 ..... key_n p_n

(6) 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。

56

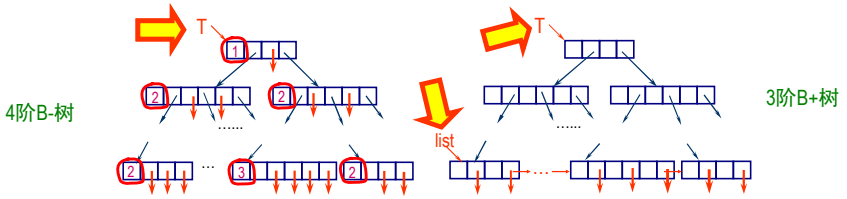


B-树和B+树的应用

- B-树是1970年由R.Bayer和E.MacCreight提出的，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-树多用于文件系统或数据库系统的索引结构
- B*树
  - ✓ B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针

59

5.5 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)。**

58

5.6 Trie结构及查找*

- 在二叉树遍历中通常是通过比较整个键值来进行的，即每个结点包含一个键值，该键值与要查找的键值进行比较然后在树中寻找正确的路径。而用**键值的一部分**来确定查找路径的树称为trie树（它来源于retrieval）（为了在发音上区别tree，可读作try）
- 主要应用
  - ✓ 信息检索（information retrieval）
  - ✓ 用来存储英文字符串，特别是大规模的英文词典（在自然语言理解软件中经常用到，如词频统计、拼写检查）

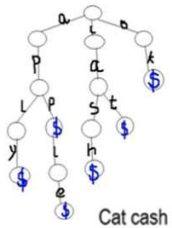
60

Trie结构的适用情况*

- **Trie结构主要基于两个原则**
  - ✓ 键值由固定的字符序列组成（如数字或字母），如Huffman码(只由0,1组成)、英文单词（只由26个字母组成）；
  - ✓ 对应结点的分层标记；
- **Trie结构典型应用“字典树”：** 英文单词仅由26个字母组成（不考虑大小写）
  - ✓ 字典树每个内部结点都有26个子结点 – **多叉树**
  - ✓ 树的高度为最长单词长度

Trie的核心思想是以空间换时间。利用字符串的公共前缀来降低查找时间的开销，以达到提高效率的目的。

Trie树缺点，假定对字母与数字进行处理，每个结点有52+10个子结点



Cat cash app apple aply ok

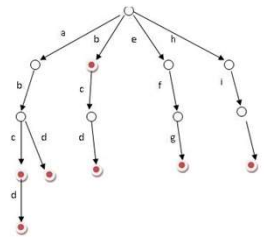
Trie结构性能分析*

- 采用Trie结构，对英文单词来说，树的高度取决于最长的单词长度。绝大多数常用单词通常都不是很长，一般访问几个结点（很可能是5~7个）就可以解决问题。
- 而采用（最理想的）平衡二叉查找树，假设有10000个单词，则树的高度为14（lg10000）。由于大多数的单词都存储在树的最低层，因此平均查找单词需要访问13个结点，是trie树的两倍。
- 此外，在BST树中，查找过程需要比较整个单词（串比较），而在trie结构中，每次比较只需要比较一个字母。
- **在访问速度要求很高的系统中，如拼写检查、词频统计中，trie结构是一个非常好的选择。**

Trie树示例

一种用于描述单词的trie结构定义

```
struct tnode {  // word tree
   char isword;  // is or not a word
   char isleaf;     // is or not a leaf node
   struct tnode *ptr[26];
};
```



```
//基于trie结构的单词树的构造
void wordTree(struct tnode *root, char *w){
    struct tnode *p;
    for (p = root; *w != '\0'; w++){
        if (p->ptr[*w - 'a'] == NULL) {
            p->ptr[*w - 'a'] = talloc();
            p->isleaf = 0;
        }
        p = p->ptr[*w - 'a'];
    }
    p->isword = 1;
}
struct tnode *talloc(){
    int i;
    struct tnode *p;
    p = (struct tnode *)
         malloc(sizeof(struct tnode));
    isword = 0;   isleaf = 1;
    for (i = 0; i < 26; i++)
        ptr[i] = NULL;
    return p;
}
```

内容提要

1. **查找的基本概念**
2. **顺序表的查找**
3. **索引**
4. **二叉查找树(BST)**
5. **B-树和B+树**
6. **散列(Hash)**

顺序表查找法：

- 顺序查找法
- 折半查找法
- 索引查找法
- 在B-树与B+树中进行的查找方法

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	…

基于关键字值比较的查找方法

查找的时间效率主要取决于查找过程中进行的比较次数

能否有一种不经过或经过很少次的关键字值的比较就能够达到目的的方法?

6.1 散列查找的基本概念

◆ **散列函数定义**

$A = H(k)$

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

函数值A为k对应的记录在表中的位置

例1

地址范围: [1..30]

散列函数： H(k)=k–99000

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	…

6. 散列(Hash)查找

◆ 对于频繁使用的查找表，希望ASL = 0

◆ 只有一个办法：预先知道所查关键字在表中的位置

◆ 即要求：记录在表中位置和其关键字之间存在一种确定的关系

◆ 但是，对于动态查找表而言

1) 表长不确定

2) 在设计查找表时，只知道关键字所属范围，而不知道确切的关键字

◆ 因此在一般情况下，需在关键字与记录在表中的存储位置之间建立一个函数关系，以 H(key) 作为关键字为 key 的记录在表中的位置，通常称这个函数 H(key) 为哈希函数

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

例2

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	…

地址范围: [1..30]

1	李 军 …
2	张 云 …
3	
4	王民…
:	…
:	
30	

地址冲突

H(张云)=2
H(王民)=4
H(李军)=1
H(汪敏)=4

一个处理过程

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”

选择一种处理冲突的方法

6.1 散列查找的基本概念

◆ **散列冲突定义**

- ✓ 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”

◆ **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，并以关键字在该集合中的“象”作为记录的存储位置，按照这种方法组织起来的文件称为**散列表**,或 **哈希表** ,或称 **杂凑表。**
- ✓ 建立文件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。

6.2 散列函数的构造

◆ **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。

详见相关参考书

◆ **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 $H(k)=ak+b$

$H(k)=k-99000$

6.2 散列函数的构造

◆ **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

除留余数法

构造：取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，

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

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。

特点：简单、常用，可与上述几种方法结合使用，p的选取很重要；
p选的不好，容易产生同义词

6.3 开放地址法处理冲突

- **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，则继续为发生冲突的这个元素寻找另一个地址，直到不再发生冲突。
- **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。

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

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, …, m–1 称为线性探测再散列

(2) d_i=1^2, -1^2, 2^2, -2^2, …, 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

散列表查找过程

例4 已知有长度为M的散列表HT[0..M–1]，散列函数为H(k)，并且采用线性探测再散列方法处理冲突。请写出在该散列表中查找关键字值为key的元素存在与否的算法。若存在,则给出它在表中的位置，否则，给出相应信息。

例

采用线性探测再散列方法处理冲突

H(k)=k MOD 13

$D_i = (k \text{ MOD } 13 + d_i) \text{ MOD } 13$

HT:

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

key=70 key=18 key=38 key=20

散列表插入过程

除留余数法

例3 设散列函数为 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

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

$D_i = (k \text{ MOD } 13 + d_i) \text{ MOD } 13$

散列地址为5

线性再散列

0	1	2	3	4	5	6	7	8	9	10	11	12
					70	19	33	18				

二次再散列

0	1	2	3	4	5	6	7	8	9	10	11	12
					70	19	33		18			

聚集

- **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) 负载因子过大

装填因子

- **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，α<1，α越大，散列表越满

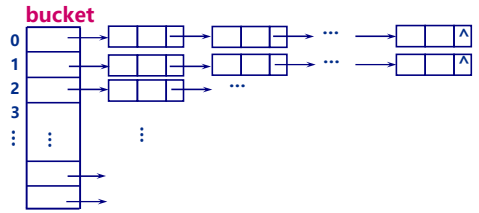
6.4 其他处理冲突的方法

- **再散列法**

  $D_i = H_i(k)$ i=1, 2, 3, ... 其中，D_i为散列地址，$H_i(k)$为不同的散列函数

- **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。



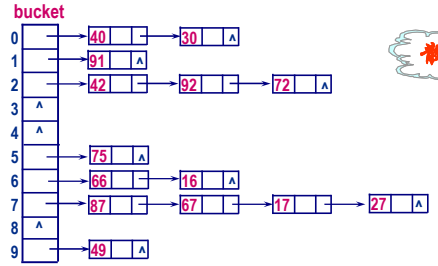
开放地址法处理冲突的特点

- “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

链地址法处理冲突

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

例5

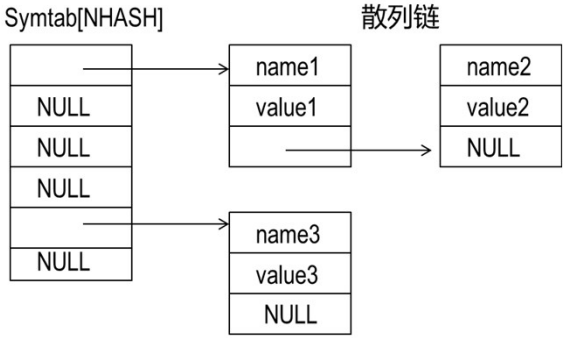


链地址法处理冲突的特点

- 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

在各种查找方法中，
只有*散列查找法*的平均查找长度ASL与元素的个数n无关!

6.5 散列表的典型应用*

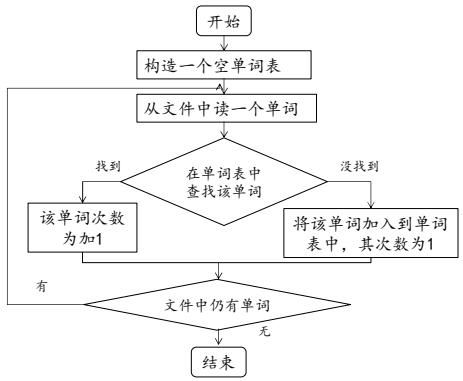


6.5 散列表的典型应用*

- 散列表的一个典型应用是**符号表（symbol）**，用于在数据值和动态符号（如变量名，关键码）集的成员间建立一种关联。
  - ✓ 符号表是**编译系统**中主要的数据结构，用于管理用户程序中各个变量的信息，通常编译系统使用散列表来组织符号表。
  - ✓ **散列表**的思想就是把关键码送给一个散列函数，以产生一个散列值，这种值通常平均分布在一个适当的整数区间中，用作存储信息的表的下标。
  - ✓ 常见做法是为每一个散列值关联一个数据项的链表，这些项共用同一个散列值（散列冲突）。
- **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。**

问题2.1：词频统计 – 链表

- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数
- 算法分析：本问题算法很简单，基本上只有查找和插入操作



84

问题2.1：词频统计 – 链表

本问题有如下特点：

(1) 问题规模不知（即需要统计的单词数量未知），有可能很大，如对一本小说进行词频统计

(2) 单词表在查找时需要频繁的执行插入操作，是一种典型的动态查找表

针对上述问题，在“线性表”一章采用了链表来实现

在“树”一章中采用了二叉排序树（BST）来实现

链表实现方式插入算法简单效率高，但查找效率低，

有没有方法能提高链表方式的查找效率？ **散列（Hash）查找！**

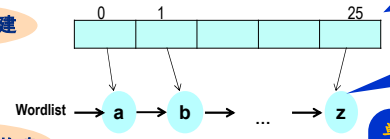
词频统计-利用散列查找提高链表实现查找效率

- **要求**
  - ✓ 读取文章中的单词，统计每个单词出现的频率，请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：**采用链地址法处理冲突，哈希值相同的单词存放在一个链表中

问题2.1：Hash表的设计与实现

1. Hash表的构建



然后构造一个长度为26的指针数组，数组内容分别为指向链表中每个结点的指针struct node *Hashtab[26];

首先构造一个仅由26个字母为单词的单词链表 struct node *Wordlist;

2. Hash函数的构建

```
int  hash(char  *word) {
    return *word - 'a' ;
}
```

单词的查找不再每次由单词表的头开始，而是由查找单词的头字母开始的单词区域开始查找。在没有增加任何算法复杂度的情况下利用Hash查找方法大大提高了单词的查找效率。 searchWord(word);

3. 单词的查找和插入

```
int searchWord(char *w){
    …
    h = hash(w);
    for(p=Hashtab[h]; p != NULL; q=p,p=p->link)
    …
    return insertWord(q, w);
}
```


词频统计：哈希表实现（头部信息）

```
#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，范围0~size范围内的哈希值
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

```c
// 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);
}
```

89

词频统计：哈希表实现-主函数

```c
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-哈希表的更新和插入

```c
PWordTable  searchAndInsertWord(char word[]){
    int hash = SDBMHash(word, TABLELEN); // 计算字符串的哈希值
    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;
}
```

90

词频统计：哈希表实现-读单词函数

```c
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;
}
```

92

单选题 1分

在建立散列表时，若散列函数为H(k)，a与b分别为关键字值，下列哪种情况称之为散列冲突：

A **a = b**

B **a <> b**

C **a = b 且 H(a) = H(b)**

D **a <> b 且 H(a) = H(b)**

93

查找总结

- 查找的基本概念
- 顺序表及其查找
  - ✓ 顺序文件
    - 一般顺序表、排序顺序表
    - 连续顺序表、链接顺序表
    - 排序连续顺序表
  - 连续顺序表查找
    - 顺序查找和折半查找（递归和非递归）
    - 复杂度分析（判定树）
  - 链接顺序表查找
- 索引表及其查找
  - ✓ 索引与索引表
  - ✓ 稠密索引和非稠密索引
- 二叉查找树(BST)
- B-树与B+树
  - ✓ B-树的结构，B-树的查找
  - ✓ B-树的插入（结点分解规则）
  - ✓ B+树的结构
  - ✓ B-树与B+树的异同
- 散列（Hash）表及其查找
  - ✓ 散列的基本概念
    - 散列函数及其构造方法
    - 散列冲突
  - ✓ 散列冲突处理方法
    - 开放地址法，再散列法，链地址法

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)
BST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

94