查找算法

**基本概念：**

1. **关键字：**假如有结构

struct Node //一个结点，存储数据和指针

{

DATA data; //数据属性，用于存储数据

int key; //假设key为int值，其在整个表里是唯一的

//指针域，具体略，指向其他结点，或者是数组的下标

};

key值便是关键字，对于每一个结点而言，其key值都是不一样的（不一定必须是int值）。因此，当我们查找数据时，只要知道其key值，然后对比key值和我们要查找的key值是否相同，便能判断是否是我们要查找的数据了。

优点：①数据属性的被更改，不影响查找，而且key值通常是不会被更改的；

缺点：①需要更多的空间用于存储key值；

**（2）查找时间复杂度：**

查找时（查找也是一个算法），需要记录执行代码需要的时间（可以理解为执行多少行代码），而这个执行需要的时间，就是算法的时间复杂度。当数据量为n时，记作T(n)=O（f(n)），f(n)表示数据量为n时的某个函数。

而O（f(n)）指的是，这个函数在数据量为n时，算法的时间复杂度。

他表示随问题规模n的增大，算法执行的时间的增长率。这样用大写O（）来体现算法复杂度的记法，称为大O记法。

一般关心的是大O记法的平均时间，和最坏结果的时间。

其常见的几种情况是：

①假如无论数据多少，其查找时间都是一个常数，那么记为O（1），表示是常数；

②假如是对数型增长（数据每增长一倍，次数增加1次），记为O（log n），表示对数；

③假如是线性增长，记为O（n）。

④假如是乘方增长（和数据量的关系是乘方关系），记为O（n2）

**（3）查找表和查找**

查找表：同一个类型的数据的集合（例如树中的结点是同一个类型，树中结点的集合就是一个查找表）。

查找：根据某个值（key），在查找表中确定一个项（比如树中的一个结点，比如说得到指向这个结点的指针）。

**（4）命中**

可以理解为查找到自己要查找的项了。

**顺序查找（线性查找）：**

1. 适用情况：

绝大多数情况。

（2）原理：

从第一个开始查找，并依次尝试进行匹配，一直到查找到符合要求的值，或者是最后一个项为止。

1. 算法最佳适用情况：

①数据库较小；

②对时间没有苛刻要求。

（4）算法时间复杂度：

O（n）

**二分法查找：**

（1）适用情况：

前提：key是有序的，并且表中顺序由key规定，且一般不变（如果变的话需要重新对表排序）。

1. 原理：

先找查找表中最中间的结点m，然后比较m的key值和要查找的key值的关系，如果比m.key值大，则找m右边的（范围比之前缩小一半）。如果比m.key值小，则找m左边的（范围依然比之前缩小一半）。如果和m.key值一样大，则命中。

如代码：

Node\* find(int last, int key,Node \*a) //这里适用的是数组型，a指的是指向结点数组的指针

{

int f, l, m;

f = 0;

l = last;

while (f <= l) //只要范围左限的下标比右限的下标小即可

{

m = (f + l) / 2;

if (key < a[m].key) //如果比中间结点的key值小

l = m - 1; //比中间值小1（也是出现新范围）

else if (key>a[m].key) //比中间结点的key值大

f = m + 1;

else

return a; //命中

}

return NULL; //说明没命中，返回空指针

}

1. 算法最佳适用情况：

有序的表、有序的二叉树，数据较多时。

1. 算法时间复杂度：

O（log n）

**插值查找：**

1. 算法原理：

根据key值和最大、最小下标之间的关系，来优化的。（具体我也没看懂）

1. 适用情况：

key值比较均匀的表。例如1、2、3、4、5、6……这样。不适合分布极端的（如1、100、500、1000……）

1. 算法：

Node\* find(int last, int key,Node \*a) //这里适用的是数组型，a指的是指向结点数组的指针

{

int f, l, m;

f = 0;

l = last;

while (f <= l) //只要范围左限的下标比右限的下标小即可

{

m = l + (key - a[f].key) / (a[l].key - a[f].key)\*(l - f); //\*\*\*\*修改的是这一行\*\*\*\*

if (key < a[m].key) //如果比中间结点的key值小

l = m - 1; //比中间值小1（也是出现新范围）

else if (key>a[m].key) //比中间结点的key值大

f = m + 1;

else

return a; //命中

}

return NULL; //说明没命中，返回空指针

}

1. 最佳适用情况：

key值分布均匀的。

**斐波那契查找：**

1. 原理：

没看懂。。。。好吧，是没耐心看。

（2）适用情况：

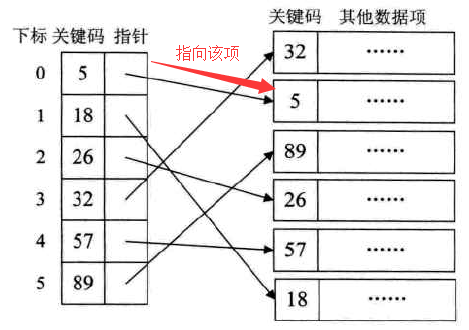
对被查找值靠近**右半侧**的，效率比二分查找更高；

但对很靠近**最左边**的，效率比二分查找低。

**索引：**

所谓的索引，指有一个专门的索引表，用于存储每一个key值和指向该key值所在的项的指针。这里的索引，指的是 **线性索引**。

例如：



**索引的特点：**

①有序的。数据项可能是无序的（如图中的右半部分），但索引是有序的。因此无论数据项是否有序，只要找到key值符合的索引项，自然能根据索引项的指针，找到指向的数据项。

②因为有序（这里并非指索引表中全部有序），所以可以使用二分查找法，或者其他查找法，用于查找符合要求的key值。

③占用空间很小。可能只需要一个int值和一个指针，相比较数据项而言，空间小很多。

④但提升效率很高。在无序数据项中查找，基本只有线性查找了，然而利用索引，将线性索引（O（n）），变为二分法查找（O（log n）），因此提升效率很高。

**稠密索引：**

1. 定义：

指在线性索引中，数据集的每一个对应一个索引项。

1. 特点：

①一定是按有序（按key值）排列的；

②查找效率高；

1. 缺点：

①当数据量增长极快时，是没办法（或很难）进行有序排列的；

②当数据量极大时，读取是比较困难的（因为每一个对应一个索引项，因此索引表也很大）。

**分块索引：**

1. 定义：

将数据集分为了几块（几部分），然后每一块对应了一个索引项。

1. 特点：

①块内是无序的；

②块间是有序的（因此索引是有序的）；

（3）原理：

①块内虽然无序，但符合一定的要求，例如key值在一定范围之内；

②块内有两个值，用于记录块内的最小key值和最大key值；（因此其他块某项的值，必然比这个块所有项的key值大，或者小）；

③索引项会有一个值，记录当前块内的项数；

④指针指向块首（无需知道块尾的，因为有项数，查找完项数的数目之后，自然结束）；

（4）优点：

①快速划定所在块，然后可以用逐个查找（此时块内项数并不多）也不会很慢。

**倒排索引：**

（1）简单概念：

①设置关键词（这个关键词是我们要查找的内容，例如单词），然后产生一个关键词表。

②每个关键词项，有一个数组，记录该该关键词所在的项的编号（例如记录它是数据库里面的第几项）；

③查找一个关键词时，可以先找关键词表中，该关键词所在的项。然后便找到该项记录的编号表（记录着有哪些数据库的项，有这个关键词）；

④于是便得到一张表，里面每个项，都包含我们要找的关键词；

⑤显示出来，搜索结束。

（2）优点：

①适合查找单词，原理简单，存储空间小，响应速度快。

②关键词表可以按首字母排列，然后同一个字母的所有单词甚至可以放到同一个块内（分块索引），因此效率很高；

③实际应用中，不需要一次显示出所有的，因此可以一次读取若干项（例如数组的0#~9#项，下一次再读取10#~19#项）；

**二叉排序树：**

（1）特点

利用二叉树的形式，进行搜索。

与二分搜索不同的是，二分搜索主要面对的是数组（有下标），而二叉排序树是没有数组的（用的是链表）。因此，在设计代码的时候，是不能用middle=（first+last）/2这样的办法的。

（2）算法：（这里的二叉树换个思路重新写，**新增删除结点**）

看的时候，建议自己画个二叉树，然后跟着代码思路走一遍，理解的会比较深刻。

结点：

struct Tree

{

data m;

int key;

Tree\* Lchild = NULL, \*Rchild = NULL;

};

查找：

bool SearchTree(Tree\*T, int key, Tree\*p, Tree\*\*n) //指向当前结点的指针T，key值key，指向父结点p（默认为NULL），指向搜索路径上最后一个非空结点的指针的地址n

{

if (T == NULL) //如果是空指针（查找失败）

{

\*n = p; //指向当前结点指针的指针，指向父指针（实质上p是指向访问路径上的最后一个非空结点指针的地址）

return false;

}

else if (T->key == key) //当前结点的key值符合要求

{

\*n = T; //指向当前结点指针的指针，指向当前结点

return true;

}

else if (T->key > key) //在左子树

SearchTree(T->Lchild, key, T, n); //其参数分别为指向左孩子的指针，key值，指向左孩子的父结点的指针，指向

else

SearchTree(T->Rchild, key, T, n);

}

效果是，找到对象返回指向对象的指针，没有找到就返回空指针。

插入：

bool InsertTree(Tree\*T, int key) //先查找，key值重复插入失败返回false，key值不重复插入成功返回true

{

Tree\*temp = NULL, \*p;

if (!SearchTree(T, key, NULL, &temp)) //如果查找失败（说明不重复），此时temp的值是路径上最后一个非空结点（一定是一个叶结点）

{

p = new Tree;

p->key = key;

if (T == NULL) //如果插入的位置是根结点（由于预先设置，因此T是存在的，直接赋值key给根结点的key

T->key = key;

if (p->key > temp->key) //如果要插入的结点的key值比其父节点大

temp->Rchild = p;

else //否则小（不可能相等）

temp->Lchild = p;

return true; //插入成功，返回true

}

else

return false; //查找到，插入失败返回false

}

效果：将一个key值插入二叉树之中（不涉及数据域的操作），若二叉树里无该key值则成功，返回true，否则返回false

删除一个结点：

分为四种情况：

（1）该结点为空结点（不用删）；

（2）该结点左子树为空，将指向自己的指针，指向自己的右子树；

（3）该结点右子树为空，将指向自己的指针，指向自己的左子树；

（4）该结点A左右子树都存在，在左子树（根结点为B）里找最右边的（或者在右子树找最左边的）结点C，然后替换自己。然后A的父结点指向A的指针，指向C，C的左子结点指针指向B，B的右子结点指向A的左子结点。

思路很简单，但是写起来很别扭。

我尝试写了一个，不确定是否正确。以后有机会的话在验证吧，如果有人验证出错，欢迎留言提醒。

bool DeleteTree(Tree\*\*T, int key) //先查找，然后删除

{

Tree\*l,\*r;

if ((\*T)->key = key) //第一个就是（说明是根）

{

delete \*T;

return true;

}

else

{

while ((\*T)->key != key&&(\*T)!=NULL) //如果当前key不同，并且不是空指针

{

l = (\*T)->Lchild;

r = (\*T)->Rchild;

if ((\*T)->key > key) //如果key更小

\*T = (\*T)->Lchild; //指向其左子

else \*T = (\*T)->Rchild; //如果key更大，指向其右子

}

if(\*T==nullptr) //如果是空指针，说明没找到

return false;

//于是此时找到key项了，并且此时l和r是其父结点的左右子

Tree\*\* temp;

if (l->key == key) //如果左子是key值

temp = &l; //temp是左指针的地址

else temp = &r; //否则temp是右指针的地址

//于是\*temp是指向该项的指针（其父节点指向其的指针）

if ((\*T)->Lchild == NULL) //如果左子树为空

\*temp = (\*T)->Rchild;

else if ((\*T)->Rchild == NULL)

\*temp = (\*T)->Lchild;

else //否则左右子树都不空

{

//查找左子树的最右结点，及其父结点（因为要将修改他的右指针指向）

Tree\*A, \*B, \*C;

C = (\*temp)->Lchild; //C将为其左子树的最右结点

while (C->Rchild != NULL)

C = C->Rchild; //指向成功

//4）该结点A左右子树都存在，在左子树（根结点为B）里找最右边的（或者在右子树找最左边的）结点C，然后替换自己。然后A的父结点指向A的指针，指向C，C的左子结点指针指向B，B的右子结点指向A的左子结点。

A = \*temp; //A此时是指向被删除结点的指针

\*temp = C; //被删除结点的父结点目前指向替换的结点

B = A->Lchild; //B是A的左子

if (B != C) //要排除B和C是同一个结点

{

B->Rchild = C->Lchild; //B的左子是C的左子（左子树保持不变）

C->Lchild = B; //然后C的左子是B

}

C->Rchild = A->Rchild; //C的右子是A的右子

delete A;

}

}

}