链滴

摸一摸数据结构（查找 1.0）

作者：stillwarter

原文链接：https://ld246.com/article/1631005824457

来源网站：链滴

这一块猫猫在学生时代也只是知道一点简单的查找方式，更高级更难的猫猫也不懂，所以会以概念与找的思想为主，不会再去测试与实现。

猫猫接受了折磨，还是先把树的坑磨平了再来……

查找的基本概念

查找：在数据集合中寻找满足某种条件的数据元素称为查找。

查找表：用于查找的数据集合称为查找表。对查找表的操作一遍有四种

1. 查询元素是否在表中
2. 检索满足某个特定的数据元素的各种属性
3. 在查找表中插入一个数据元素
4. 从查找表中删除某个数据元素

静态查找表：在查找表的操作只涉及到 1,2

关键字：数据元素中唯一标识该元素的某个数据项的值（有点像数据库的主键）

平均查找长度：在查找过程中，一次查找的长度是指需要比较的关键字次数。

查找算法分类

根据概念分有静态和动态查找，从其他方面，也有不同的归类方式

1. 线性结构
    1. 顺序查找
    2. 折半查找
    3. 分块查找（索引顺序查找）
2. 树形结构
    1. 二叉排序树
    2. 二叉平衡树
    3. B 树，B+ 树
3. 散列结构：散列表
4. 效率指标：查找成功或失败

查找算法实现

在此之前我们先声明表结构，有了表的结构才能在表上进行查找。

表结构

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include "Status.h"
#include "Scanf.c"

/* 查找表类型定义 */
typedef int KeyType;            //关键字类型
typedef struct
{
    int key;                    //关键字域
    float weight;               //其他域（此处可设为权重）
}ElemType_Search;                //有序表元素类型

//0号单元弃用
typedef struct
{
    ElemType_Search *elem;       //数据元素存储空间基址，0号单元为空
    int length;                  //表长度
}Table;
```

表结构中有表长度和指针（指向表中的结点信息）信息，而表中的结点结构有关键字和权值信息。

所以我们可以通过 key 来查找表中的结点信息。

```c
#include "Base.h"

Status Create(FILE *fp, Table *T, int n)
{
    int i;
    int a;
    float b;

    T->elem = (ElemType_Search *)malloc((n+1)*sizeof(ElemType_Search));
    if(!(T->elem))
        exit(OVERFLOW);

    //此表0号单元是弃用的
    for((*T).length=0,i=1; i<=n; i++)
    {
        if(Scanf(fp, "%d%f", &a, &b)==2)
        {
            (*T).elem[i].key = a;
            (*T).elem[i].weight = b;
            (*T).length++;
        }
    }

    return OK;
}

void Destory(Table *T)
{
    if(T->elem)
        free(T->elem);
```

```
    T->elem = NULL;
    T->length = 0;
}

void Traverse(Table T, void(Visit)(ElemType_Search))
{
    int i;

    for(i=0; i<T.length; i++)
    {
        if(i && !(i%10))
            printf("\n");

        Visit(T.elem[i+1]);
    }

    printf("\n");
}

/* 输出查找表中的关键字 */
void PrintKey(ElemType_Search e)
{
    printf("%3d ", e.key);
}
```

顺序查找（无序表）

```
int Search_Seq(Table T, KeyType key)
{
    int i;

    T.elem[0].key = key;

    for(i=T.length; !EQ(T.elem[i].key, key); --i)
        ;

    return i;
}
```

传入表 T 和关键字 key，将传入的 key 值赋予表头元素的 key 值（注意创建表是 0 号位次的结点是有信息的，而传入这个信息是为了在查找不到的时候最后输出 0 代表没有该关键字）。

折半查找

```
int zheban(Table T,KeyType key){
    int low,high,mid;
    low=1;
    high=T.len;
    while(low<=high){
        mid=(low+high)/2;
        if(EQ(key,T.elem[mid].key)) return mid;
        else if(LT(key,T.elem[mid].key)) high=mid-1;
        else low=mid+1;
```

```
    }
    return 0;

}
```

折半查找的思想很容易理解，以查找数组为例，我们定义 mid，low，high 来代表数组的上下限和中位置的下标值。

从数组的中间开始查找（偶数也随便在中间的那两个随便选一个），最好的情况是等于（但一般不是；若大于中间数据，则我们更新中间值 mid 为（mid+high）/2，low 更新为 mid；若小于，则更新 mid 和 high。

如此循环，直至找到。若最后更新到 2 个相邻位次的值不等，则数组内没有该数据。

斐波那契查找

知道这两个的先知道斐波那契数列（1,1,2,3,5,8,13,21……），在这样的数列中，随着其递增，前后 2 数的比值会越来越接近 0.618（黄金比例），利用这个特性，我们可以对折半进行提升。（通过运用个比例来选择查找点进行查找，所以斐波那契查找也是一种有序查找算法）

步骤：

- 构建斐波那契数列；
- 计算数组长度对应的斐波那契数列元素个数；
- 对数组进行填充；
- 循环进行区间分割，查找中间值；
- 判断中间值和目标值的关系，确定更新策略；

```
#include "FibonacciSearch.h"            //**▲09 查找**//

int Search_Fib(Table T, KeyType key)
{
    int low, high, mid;
    int Fib[MaxSize+1];
    int i, k;
    int *table;

    Fib[1] = Fib[2] = 1;
    for(i=3; i<=MaxSize; i++)
        Fib[i] = Fib[i-1] + Fib[i-2];

    table = (int *)malloc((Fib[MaxSize])*sizeof(int));

    for(i=1; i<=T.length; i++)
        table[i] = T.elem[i].key;

    for(i=T.length+1; i<=Fib[MaxSize]-1; i++)
        table[i] = T.elem[T.length].key;

    k = MaxSize;
    low = 1;
    high = Fib[k];
```

```
    while(low<=high)
    {
        mid = low + Fib[k-1] - 1;    //中点的确定

        if(key<table[mid])
        {
            high = mid - 1;
            k = k - 1;
        }
        else if(key>table[mid])
        {
            if(mid<T.length)
            {
                low = mid + 1;
                k = k - 2;
            }
            else
                return 0;           //未找到
        }
        else
        {
            if(mid<T.length)
                return mid;
            else
                return T.length;    //在最后一个位置
        }
    }

    return 0;
}

#endif
```

首先创建斐波那契数列，在根据表的长度将斐波那契数列的值替换为表中的关键字（注意这里是将斐那契的值添加入表值，没加一次表长度加一）。

之后我们就可以确定表中的最值和中点值。这里中点值的算法不是像折半一样用最值和除以二，而是低于高值之间的关系来确定。

因为斐波那契数列 f[k]=f[k-1]+f[k-2]；所以数列可以分为 2 段，（后补）

插值查找

```
#include "InterpolationSearch.h"
int Search_Int(Table T, KeyType key)
{
    int low, high, m;
    float j;

    low = 1;
    high = T.length;
```

```
    while(low<=high)
    {
        m = (1.0*(key-T.elem[low].key)/(T.elem[high].key-T.elem[low].key))*(high-low) + low;    /
注意浮点数转整型

        if(key<T.elem[m].key)
            high = m - 1;
        else if(key>T.elem[m].key)
            low = m + 1;
        else
            return m;
    }

}

#endif
```

差值查找是一种有序表的操查找方式，根据关键字与查找表中最大最小记录关键字比较后的查找方法同时也是基于二分查找，将查找点的选择改为自适应选择来提高效率。

什么是差值查找

次优查找树

二叉排序树

平衡二叉树

B 树查找与相关操作

什么是 B 树

我们可以先知道它是怎么实现了，再去思考他的思想。

B 树，又叫多路平衡二叉树，B 树中所有结点的孩子个数的最大值是 B 树的阶，通常用 m 表示。一棵 m 阶 b 树或为空树，或为满足以下特性的 m 叉树

1. 树中每个结点最多有 m 棵子树，至多含有 m-1 个关键字。
2. 若根结点不是终端节点（不是叶子结点），则至少有 2 课子树。
3. 除根结点外的所有非叶子结点至少有 m/2 棵子树。
4. 所有非叶子结点的结构：

**n p_0
 k_1
 p_1 k_2
 _2 ..... k_n
 p_n**

其中 k_i 是关键字，满足从小到大的顺序，P_i 是指向子树根结点的指针，且指针

5. 所有叶子结点都出现在同一层次上，并且不带信息，实际上这些结点不存在（指向这些结点的针为空）

```c
#include <stdlib.h>
#include <math.h>                       //提供ceil原型
#include "../00 Base/Base.c"              //**▲09 查找**//

/* 宏定义 */
#define M 3                             //B树的阶，暂设为3
#define EQ(a,b) ((a)==(b))
#define LT(a,b) ((a) < (b))

/* 类型定义 */
typedef ElemType_Search BTElemType;         //B树元素类型
typedef struct BTNode                       //B树存储表示
{
    int keynum;                         //结点中关键字个数
    struct BTNode* parent;                  //指向双亲结点

    KeyType key[M+1];                   //关键字向量，0号单元未用
    struct BTNode* ptr[M+1];                //子树指针向量
}BTNode;                                //B树结点
typedef BTNode* BTree;                      //指向B树结点的指针

/* B树查找结果类型 */
typedef struct
{
    BTree pt;                           //指向找到的结点
    int i;                              //1...n，关键字在结点中的序号（插入位置）
    int tag;                            //1:查找成功，0:查找失败
}Result;
```

B+ 树

双链树

trie 树

哈希表

ps: 摸一摸数据结构(目录)