02-D2-7 查找长度 & Fibonacci查找

#数据结构邓神

如何更为精细的评估查找算法的性能

考察关键码的比较次数，即查找长度(Search Length)

通常，需要针对成功和失败查找，从最好最坏平均等角度评估

比如成功和失败平均查找长度大致为O(1.50logn)

还有改进的余地!

Fibonacci 查找

前面查找的算法的复杂度为 O(1.5logn)

这个1.5是可以减少的！！！

就要使用 Fibnacci查找

改进余地在于：转向左右分支的关键码比较次数不想等，但是递归深度却相等

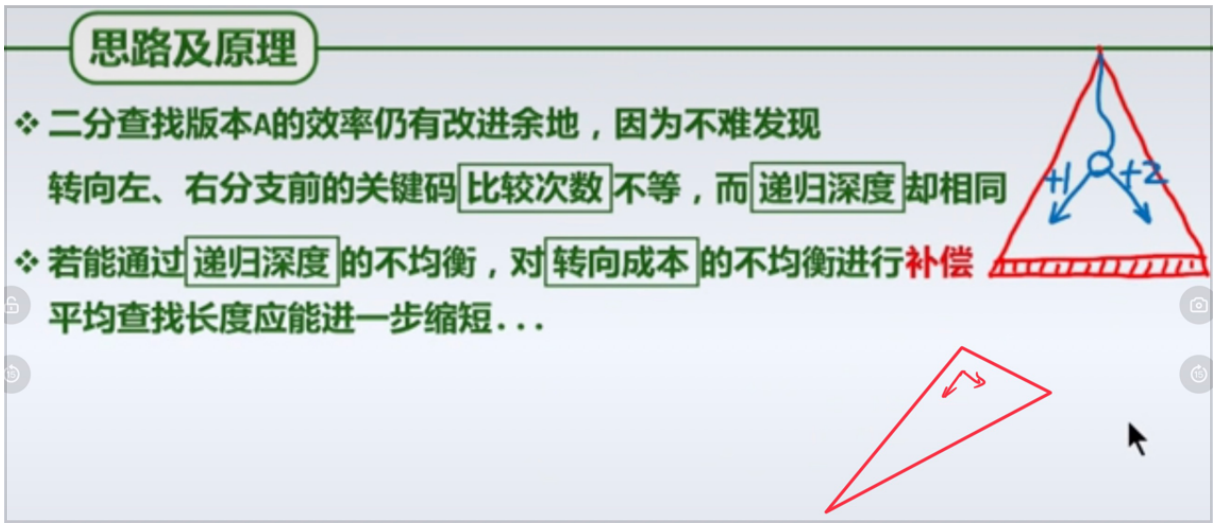
像左侧需要一次比较但是像右侧却需要两次比较?

算法效率可能还没有达到最优

我们发现像左侧的成本要低于像右侧的成本

那么我们能不能让这颗树能够让左侧深度更深，让右侧深度更浅

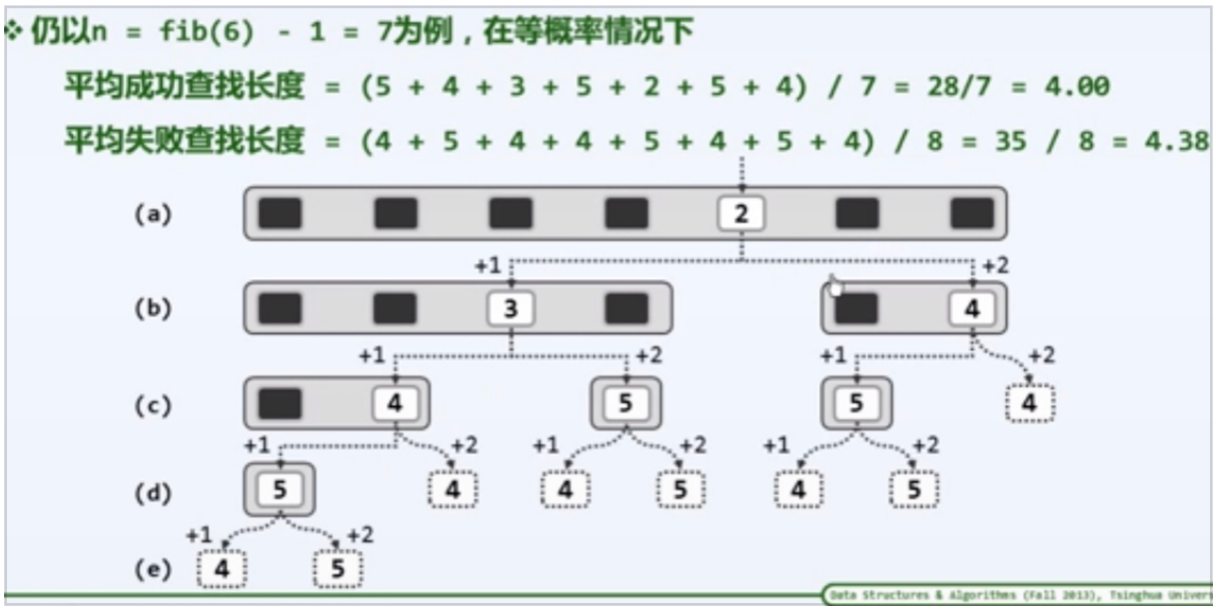
简单来说：我们希望做更多次成本低的转向，成本高的转向少做



Fibonacci 查找

如果长度N刚好为一个斐波那契数列减去1，那么我们可以去中间的点的值为这个斐波那契数列的前一个树减去1，这样就把数列分成三段

Fibonacci 查找的ASL(在常系数的意义上)优于二分查找



实现

```
template <typename T>
static Rank fibSearch(T* A,T const& e,Rank lo,Rank hi){
    Fib fib(hi - lo); // 用 O(logn) = O(log(hi - lo))时间来创建Fibonacci数列
    while(lo < hi){
        while(hi - lo < fib.get()){
            fib.prev();
        }
        Rank mi = lo + fib.get() -1;
        if (e < A[mi]){
            hi = mi;
        }
```

```
        else if (A[mi] < e){
            lo = mi + 1;
        }
        else {
            return mi;
        }
    }
    return -1;
}
```

具体的差别在于中点选择!

最关键就是确定mi

每次都取Fibonacci数列点下一项

我们通过 Fib 类取出他的下一项或者前面一项

整个Fibnacci关键点就是在于mi如何确定