二分查找算法学习札记

说明

作者:那谁
blog: http://www.cppblog.com/converse

二分查找算法基本思想

二分查找算法的前置条件是,一个已经排序好的序列(在本篇文章中为了说明问题的方便,假设这个序列是升序排列的),这样在查找所要查找的元素时,首先与序列中间的元素进行比较,如果大于这个元素,就在当前序列的后半部分继续查找,如果小于这个元素,就在当前序列的前半部分继续查找,直到找到相同的元素,或者所查找的序列范围为空为止.

用伪代码来表示, 二分查找算法大致是这个样子的:

```
left = 0, right = n -1
while (left <= right)
   mid = (left + right) / 2
   case
      x[mid] < t:    left = mid + 1;
      x[mid] = t:    p = mid; break;
      x[mid] > t:    right = mid -1;

return -1;
```

第一个正确的程序

根据前面给出的算法思想和伪代码, 我们给出第一个正确的程序,但是,它还有一些小的问题,后面会讲到

```
int search(int array[], int n, int v)
{
   int left, right, middle;

   left = 0, right = n - 1;

   while (left <= right)
   {
      middle = (left + right) / 2;
      if (array[middle] > v)
      {
         right = middle;
      }
      else if (array[middle] < v)
      {
         left = middle;
      }
      else
      {
         return middle;
      }
   }
```

```
    return -1;
}
```

下面,讲讲在编写二分查找算法时可能出现的一些问题.

边界错误造成的问题

二分查找算法的边界,一般来说分两种情况,一种是左闭右开区间,类似于[left, right),一种是左闭右闭区间,类似于[left, right].需要注意的是, 循环体外的初始化条件,与循环体内的迭代步骤, 都必须遵守一致的区间规则,也就是说,如果循环体初始化时,是以左闭右开区间为边界的,那么循环体内部的迭代也应该如此.如果两者不一致,会造成程序的错误.比如下面就是错误的二分查找算法:

```
int search_bad(int array[], int n, int v)
{
    int left, right, middle;

    left = 0, right = n;

    while (left < right)
    {
        middle = (left + right) / 2;
        if (array[middle] > v)
        {
            right = middle - 1;
        }
        else if (array[middle] < v)
        {
            left = middle + 1;
        }
        else
        {
            return middle;
        }
    }

    return -1;
}
```

这个算法的错误在于, 在循环初始化的时候,初始化right=n,也就是采用的是左闭右开区间,而当满足array[middle] > v的条件是, v如果存在的话应该在[left, middle)区间中,但是这里却把right赋值为middle - 1了,这样,如果恰巧middle-1就是查找的元素,那么就会找不到这个元素.

下面给出两个算法, 分别是正确的左闭右闭和左闭右开区间算法,可以与上面的进行比较:

```
int search2(int array[], int n, int v)
{
    int left, right, middle;

    left = 0, right = n - 1;

    while (left <= right)
    {
        middle = (left + right) / 2;
        if (array[middle] > v)
        {
            right = middle - 1;
        }
        else if (array[middle] < v)
        {
```

```
            left = middle + 1;
        }
        else
        {
            return middle;
        }
    }

    return -1;
}

int search3(int array[], int n, int v)
{
    int left, right, middle;

    left = 0, right = n;

    while (left < right)
    {
        middle = (left + right) / 2;

        if (array[middle] > v)
        {
            right = middle;
        }
        else if (array[middle] < v)
        {
            left = middle + 1;
        }
        else
        {
            return middle;
        }
    }

    return -1;
}
```

死循环

上面的情况还只是把边界的其中一个写错, 也就是右边的边界值写错, 如果两者同时都写错的话,可能会造成死循环,比如下面的这个程序:

```
int search_bad2(int array[], int n, int v)
{
    int left, right, middle;

    left = 0, right = n - 1;

    while (left <= right)
    {
        middle = (left + right) / 2;
        if (array[middle] > v)
        {
            right = middle;
        }
        else if (array[middle] < v)
```

```
        {
            left = middle;
        }
        else
        {
            return middle;
        }
    }

    return -1;
}
```

这个程序采用的是左闭右闭的区间.但是,当array[middle] > v的时候,那么下一次查找的区间应该为[middle + 1, right], 而这里变成了[middle, right];当array[middle] < v的时候,那么下一次查找的区间应该为[left, middle - 1], 而这里变成了[left, middle].两个边界的选择都出现了问题, 因此,有可能出现某次查找时始终在这两个范围中轮换,造成了程序的死循环.

溢出

前面解决了边界选择时可能出现的问题, 下面来解决另一个问题,其实这个问题严格的说不属于算法问题,不过我注意到很多地方都没有提到,我觉得还是提一下比较好.
在循环体内,计算中间位置的时候,使用的是这个表达式:
middle = (left + right) / 2;
假如,left与right之和超过了所在类型的表示范围的话,那么middle就不会得到正确的值.
所以,更稳妥的做法应该是这样的:
middle = left + (right - left) / 2;

更完善的算法

前面我们说了,给出的第一个算法是一个"正确"的程序, 但是还有一些小的问题.
首先, 如果序列中有多个相同的元素时,查找的时候不见得每次都会返回第一个元素的位置, 比如考虑一种极端情况:序列中都只有一个相同的元素,那么去查找这个元素时,显然返回的是中间元素的位置.
其次, 前面给出的算法中,每次循环体中都有三次情况,两次比较,有没有办法减少比较的数量进一步的优化程序?
<<编程珠玑>>中给出了解决这两个问题的算法,结合前面提到溢出问题我对middle的计算也做了修改:

```
int search4(int array[], int n, int v)
{
    int left, right, middle;

    left = -1, right = n;

    while (left + 1 != right)
    {
        middle = left + (right - left) / 2;

        if (array[middle] < v)
        {
            left = middle;
        }
        else
        {
            right = middle;
        }
    }
```

```
    if (right >= n || array[right] != v)
    {
        right = -1;
    }

    return right;
}
```

这个算法是所有这里给出的算法中最完善的一个,正确,精确且效率高.

参考资料

1.<<编程珠玑>>
2.wiki上关于二分查找的说明:http://en.wikipedia.org/wiki/Binary_search