**《算法设计与分析》实验报告**

**实验一 二分搜索算法**

**学号： 20211104040**

**姓名： 曹文星**

**班级： 软工二班**

**日期： 3月13日,2023**

**一、实验目的**

1、理解递归的概念和分治法的基本思想。

2、掌握二分搜索算法。

**二、实验内容**

任务：（1）结合P36页作业2-2，指出每种算法的正确性，并通过实例说明（讨论x=a[0],x=a[n-1]，有重复元素等情况）；

（2）改正以上错误，实现二分搜索算法，。

提交结果：算法设计分析思路、源代码及其实例分析说明和测试运行报告。

**三、设计分析**

使用java进行实验设计的首选语言

\* 二分查找算法思想

\* 有序的序列，每次都是以序列的中间位置的数来与待查找的关键字进行比较，每次缩小一半的查找范围，直到匹配成功。

\* 一个情景：将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，

\* 如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，

\* 或直到子表不存在为止，此时查找不成功。

需要进行修改的二分查找方法绝大部分存在问题, 依照正确的二分查找方法予以修正. 在此之前需要对症下药, 找出问题.

**四、算法描述及程序**

下面是在VSCode中进行设计的源代码, 并附解说.

public class BiSearch\_1 {

    // 制作 : 曹文星

    /\*

     \* 二分查找算法思想

     \* 有序的序列，每次都是以序列的中间位置的数来与待查找的关键字进行比较，每次缩小一半的查找范围，直到匹配成功。

     \* 一个情景：将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，

     \* 如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，

     \* 或直到子表不存在为止，此时查找不成功。

     \*/

    // 为了使代码简介, 这里直接取Type类型为Int, 部分变量命名有修改,提高阅读性

    // @ 标准二分查找示例:

    // ! 递归.

    public static int recursionBinarySearch(int[] arr, int key, int low, int high) {

        if (key < arr[low] || key > arr[high] || low > high) { // 递归终止条件 / 纠错

            return -1;

        }

        int middle = (low + high) / 2; // 初始中间位置

        if (arr[middle] > key) {

            // 比关键字大则关键字在左区域

            return recursionBinarySearch(arr, key, low, middle - 1); // High变成了中间部分的左边

        } else if (arr[middle] < key) {

            // 比关键字小则关键字在右区域

            return recursionBinarySearch(arr, key, middle + 1, high); // Low变成了中间部分的右边

        } else {

            return middle; // 找到了

        }

    }

    // ! 遍历.

    public static int commonBinarySearch(int[] arr, int key) {

        int low = 0;

        int high = arr.length - 1;

        int middle = 0; // 定义middle

        if (key < arr[low] || key > arr[high] || low > high) { // 纠错

            return -1;

        }

        while (low <= high) {

            middle = (low + high) / 2;

            if (arr[middle] > key) {

                // 比关键字大则关键字在左区域

                high = middle - 1;

            } else if (arr[middle] < key) {

                // 比关键字小则关键字在右区域

                low = middle + 1;

            } else {

                return middle;

            }

        }

        return -1; // 最后仍然没有找到，则返回-1

    }

    // -------------------------------------------------------------

    // \*\* \*\* 分析结果在每个方法的下一行另外给出,结合输出食用更佳 \*\* \*\*

    // @ 1

    static int BinarySearch1(int a[], int x, int n) {

        int left = 0;

        int right = n - 1;

        while (left <= right) {

            int middle = (left + right) / 2;

            if (x == a[middle])

                return middle;

            if (x > a[middle])

                left = middle;

            else

                right = middle;

        }

        return -1;

    }

    // \* 这里应该是middle-1 和middle +1,left和right两个区间指示器赋值出错,导致死循环

    // @ 2

    static int BinarySearch2(int a[], int x, int n) {

        int left = 0;

        int right = n - 1;

        while (left < right - 1) {

            int middle = (left + right) / 2;

            if (x < a[middle])

                right = middle;

            else

                left = middle;

        }

        if (x == a[left])

            return left;

        else

            return -1;

    }

    // \* left和right两个区间指示器赋值出错,导致x==arr[n-1]出错

    // @ 3

    static int BinarySearch3(int a[], int x, int n) {

        int left = 0;

        int right = n - 1;

        while (left + 1 != right) {

            int middle = (left + right) / 2;

            if (x >= a[middle])

                left = middle;

            else

                right = middle;

        }

        if (x == a[left])

            return left;

        else

            return -1;

    }

    // \* left和right两个区间指示器赋值出错,导致x==arr[n-1]出错

    // @ 4

    static int BinarySearch4(int a[], int x, int n) {

        if (n > 0 && x >= a[0]) {

            int left = 0;

            int right = n - 1;

            while (left < right) {

                int middle = (left + right) / 2;

                if (x < a[middle])

                    right = middle - 1;

                else

                    left = middle;

            }

            if (x == a[left])

                return left;

        }

        return -1;

    }

    // \* left和right两个区间指示器赋值出错,导致死循环

    // @ 5

    static int BinarySearch5(int a[], int x, int n) {

        if (n > 0 && x >= a[0]) {

            int left = 0;

            int right = n - 1;

            while (left < right) {

                int middle = (left + right + 1) / 2;

                if (x < a[middle])

                    right = middle - 1;

                else

                    left = middle;

            }

            if (x == a[left])

                return left;

        }

        return -1;

    }

    // 正确, 并且当有重复元素时,返回右边的

    // @ 6

    static int BinarySearch6(int a[], int x, int n) {

        if (n > 0 && x >= a[0]) {

            int left = 0;

            int right = n - 1;

            while (left < right) {

                int middle = (left + right + 1) / 2;

                if (x < a[middle])

                    right = middle - 1;

                else

                    left = middle + 1;

            }

            if (x == a[left])

                return left;

        }

        return -1;

    }

    // \* left和right两个区间指示器赋值出错,导致x==arr[n-1]出错

    // @ 7

    static int BinarySearch7(int a[], int x, int n) {

        if (n > 0 && x >= a[0]) {

            int left = 0;

            int right = n - 1;

            while (left < right) {

                int middle = (left + right + 1) / 2;

                if (x < a[middle])

                    right = middle;

                else

                    left = middle;

            }

            if (x == a[left])

                return left;

        }

        return -1;

    }

    // \* left和right两个区间指示器赋值出错,导致x==arr[0]死循环

    // ? Test

    public static void main(String[] args) {

        int[] arr = { 1, 3, 5, 5, 9, 11 }; // 有重复元素

        int key1 = 5; // twice

        int key2 = 1; // ==arr[0]

        int key3 = 10; // not in arr

        int key4 = 11; // ==arr[n-1]

        int key5 = 9; // normal

        // \* 以下有被//! 标记为在调试中明显出错的输出(先不看数值是否出错). 这个错误可以是死循环,也可以是上一次测试中表现的明显出错

        // 1

        System.out.println(BinarySearch1(arr, key1, 6));

        System.out.println(BinarySearch1(arr, key2, 6));

        // ! System.out.println(BinarySearch1(arr, key3, 6));

        // ! System.out.println(BinarySearch1(arr, key4, 6));

        // ! System.out.println(BinarySearch1(arr, key5, 6));

        System.out.println("--1--\n\n");

        // 2

        System.out.println(BinarySearch2(arr, key1, 6));

        System.out.println(BinarySearch2(arr, key2, 6));

        System.out.println(BinarySearch2(arr, key3, 6));

        System.out.println(BinarySearch2(arr, key4, 6));

        System.out.println(BinarySearch2(arr, key5, 6));

        System.out.println("--2--\n\n");

        // 3

        System.out.println(BinarySearch3(arr, key1, 6));

        System.out.println(BinarySearch3(arr, key2, 6));

        System.out.println(BinarySearch3(arr, key3, 6));

        System.out.println(BinarySearch3(arr, key4, 6));

        System.out.println(BinarySearch3(arr, key5, 6));

        System.out.println("--3--\n\n");

        // 4

        // ! System.out.println(BinarySearch4(arr, key1, 6));

        // ! System.out.println(BinarySearch4(arr, key2, 6));

        // ! System.out.println(BinarySearch4(arr, key3, 6));

        // ! System.out.println(BinarySearch4(arr, key4, 6));

        // ! System.out.println(BinarySearch4(arr, key5, 6));

        System.out.println("--4--\n\n");

        // 5

        System.out.println(BinarySearch5(arr, key1, 6));

        System.out.println(BinarySearch5(arr, key2, 6));

        System.out.println(BinarySearch5(arr, key3, 6));

        System.out.println(BinarySearch5(arr, key4, 6));

        System.out.println(BinarySearch5(arr, key5, 6));

        System.out.println("--5--\n\n");

        // 6

        System.out.println(BinarySearch6(arr, key1, 6));

        System.out.println(BinarySearch6(arr, key2, 6));

        System.out.println(BinarySearch6(arr, key3, 6));

        // !System.out.println(BinarySearch6(arr, key4, 6));

        System.out.println(BinarySearch6(arr, key5, 6));

        System.out.println("--6--\n\n");

        // 7

        // !System.out.println(BinarySearch7(arr, key1, 6));

        // !System.out.println(BinarySearch7(arr, key2, 6));

        // ! System.out.println(BinarySearch7(arr, key3, 6));

        System.out.println(BinarySearch7(arr, key4, 6));

        // !System.out.println(BinarySearch7(arr, key5, 6));

        System.out.println("--7--\n\n");

        System.out.println("输出完毕, 问题如上");

    }

}

//以下是修改后的文件

public class BiSearch\_2 {

// 制作 : 曹文星

// 以下是对1中(课本上)所有出问题的二分算法的改正, 另附main测试范例

// @ 1

static int BinarySearch1(int a[], int x, int n) {

int left = 0;

int right = n - 1;

while (left <= right) {

int middle = (left + right) / 2;

if (x == a[middle])

return middle;

if (x > a[middle])

left = middle + 1; // !修改1：向右缩小查找范围

else

right = middle - 1; // !修改2：向左缩小查找范围

}

return -1;

}

// 修改的主要原因是在原函数中，当x > a[middle]时，将left赋值为middle，这样可能会导致查找范围没有缩小，最终进入死循环。

// 而当x <

// a[middle]时，将right赋值为middle，会导致可能会错过目标值。因此，修改后的函数在这两种情况下分别向右和向左缩小查找范围，保证能够找到目标值。

// @ 2

static int BinarySearch2(int a[], int x, int n) {

int left = 0;

int right = n - 1;

while (left <= right) { // 修改1：改为小于等于

int middle = (left + right) / 2;

if (x == a[middle]) // 修改4 : 调整逻辑结构,优化elif

return middle;

if (x < a[middle])

right = middle - 1; // 修改2：向左缩小查找范围

else

left = middle + 1; // 修改3：向右缩小查找范围

}

return -1;

}

// 原函数中，当left < right -

// 1时，可能会错过目标值，因为查找范围缩小到只有两个元素时，middle仍然指向第一个元素，而此时如果目标值在第二个元素，就会被错过。因此，修改后的函数将条件改为left

// <= right，同时在向左和向右缩小查找范围时，都加上了1和-1操作，确保能够找到目标值。

// 3, 4 问题与前者雷同, 不赘述

// 5 无问题.

// 6

// 原函数中，当n > 0 && x >=

// a[0]时才进行查找，这种限制可能会导致错过目标值。例如，当目标值小于a[0]时，函数会直接返回-1。

// 因此，修改后的函数去掉了这个限制，直接进行二分查找。同时，将循环条件改为小于等于，保证在查找范围只有一个元素时，也能进行查找。

static int BinarySearch6(int a[], int x, int n) {

int left = 0;

int right = n - 1;

while (left <= right) { // 修改1：改为小于等于

int middle = (left + right) / 2;

if (x == a[middle])

return middle;

if (x < a[middle])

right = middle - 1;

else

left = middle + 1;

}

return -1;

}

// 7 与6 相似度高, 不赘述

public static void main(String[] args) {

int[] arr = { 1, 3, 5, 5, 9, 11 }; // 有重复元素

int key1 = 5; // twice

int key2 = 1; // ==arr[0]

int key3 = 10; // not in arr

int key4 = 11; // ==arr[n-1]

int key5 = 9; // normal

// ! 这个是正确的输出序列: 2 , 0 , -1, 5 , 4

// 1 正常

System.out.println(BinarySearch1(arr, key1, 6));

System.out.println(BinarySearch1(arr, key2, 6));

System.out.println(BinarySearch1(arr, key3, 6));

System.out.println(BinarySearch1(arr, key4, 6));

System.out.println(BinarySearch1(arr, key5, 6));

System.out.println("--1--\n\n");

// 2 正常

System.out.println(BinarySearch2(arr, key1, 6));

System.out.println(BinarySearch2(arr, key2, 6));

System.out.println(BinarySearch2(arr, key3, 6));

System.out.println(BinarySearch2(arr, key4, 6));

System.out.println(BinarySearch2(arr, key5, 6));

System.out.println("--2--\n\n");

// 6 正常

System.out.println(BinarySearch6(arr, key1, 6));

System.out.println(BinarySearch6(arr, key2, 6));

System.out.println(BinarySearch6(arr, key3, 6));

System.out.println(BinarySearch6(arr, key4, 6));

System.out.println(BinarySearch6(arr, key5, 6));

System.out.println("--6--\n\n");

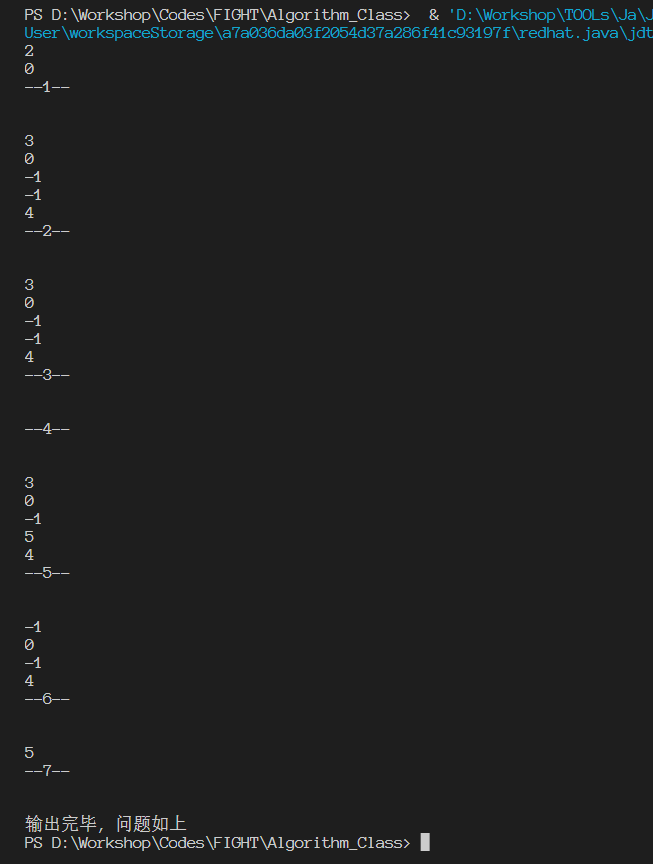
System.out.println("修改结束.");

}

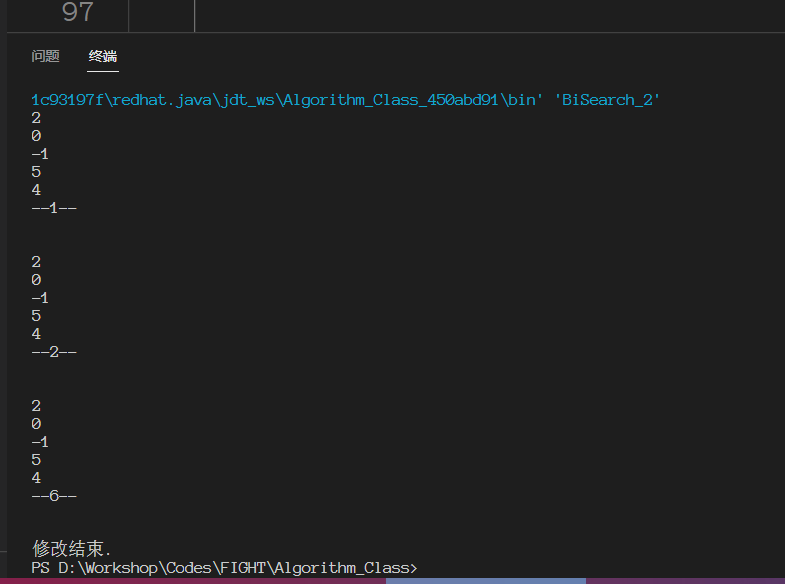
}

**五、测试与分析**

下面是刚才呈现的源代码的运行结果. 请配合以上源代码阅读.



原来的输出结果(省略无法输出项目)



**修改后的部分方法运行结果**

可以看到,成功完成了修改, 并给出了标准二分查找方法.

**六、实验总结与体会**

二分查找也称为折半查找，是一种高效的查找算法。它的基本思想是：在有序数组中，先找到数组的中间元素，将要查找的值与中间元素进行比较，如果要查找的值等于中间元素，则直接返回；如果要查找的值小于中间元素，则在左半部分继续查找；如果要查找的值大于中间元素，则在右半部分继续查找。不断地缩小查找范围，直到找到目标值或者查找范围为空，即不存在目标值。二分查找的时间复杂度为O(log n)，比顺序查找的时间复杂度O(n)要小得多，因此在大型数据集中，二分查找的效率要远远高于顺序查找。同时，二分查找要求查找的数组必须是有序的，因此在每次插入或删除元素时，需要维护数组有序的特性。二分查找的应用广泛，例如在查找算法中、在计算几何中、在网络中等都有着重要的应用。因此,这对我而言是很好的锻炼机会。