二分查找算法BinarySearch

# 二分查找及其要求

**二分查找**，又叫**折半查找；是一种效率较高的查找算法。**

## 二分查找的要求：

线性表是有序表，即**表中结点按关键字有序**，并且要用向量作为表的存储结构。不妨设有序表是递增有序的。

**存储结构：**二分查找只适用于顺序存储结构且要求元素按关键字有序排列。

## 适用情况

二分查找只适用**顺序存储结构**。为保持表的有序性，在顺序结构里插入和删除都必须移动大量的结点。因此，**二分查找特别适用于那种一经建立就很少改动、而又经常需要查找的线性表**。

对那些查找少而又经常需要改动的线性表，可采用**链表作存储结构**，进行顺序查找。**链表上无法实现二分查找**。

# 二分查找的基本思想

二分查找的基本思想是：

设R[low…..high]是当前的查找区间

1. 首先确定该区间的**中点位置**：midIndex = (low+high)/2;
2. 然后将待查的K值与arr[mid]比较：若相等，则查找成功并返回此位置，否则须确定新的查找区间，继续二分查找，具体方法如下：

### 若arr[mid]>K，则由表的有序性可知arr[mid….high]均大于K，因此若表中存在关键字等于K的结点，则该结点必定是在位置mid左边的子表R[low…mid-1]中，故新的查找区间是左子表arr[low…...mid-1]。

### 若arr[mid]<K，则要查找的K必在mid的右子表arr[mid+1……high]中，即新的查找区间是右子表arr[mid+1……high]。下一次查找是针对新的查找区间进行的。

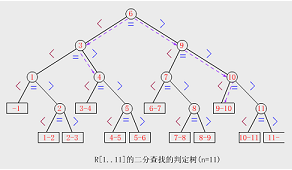
因此，从初始的查找区间R[1..n]开始，每经过一次与当前查找区间的中点位置上的结点关键字的比较，就可确定查找是否成功，不成功则当前的查找区间就缩小一半。这一过程重复直至找到关键字为K的结点，或者直至当前的查找区间为空(即查找失败)时为止。

* 成功的情况下：查找过程走了一条**从判定树的根到被查记录的路径**，比较的次数恰为记录在树中的层数。
* 失败的情况下：查找过程经过了从**判定树的根**到**某个外部节点**的路径，所需比较次数不超过判定树的高度。

二分查找的时间复杂度：时间复杂度O(log2N)。

# 二分查找判定树：

二分查找过程可用二叉树来描述：把当前查找区间的中间位置上的数据作为**根**，左子表和右子表分别作为根的左子树和右子树。由此得到的二叉树，称为**描述二分查找的判定树(Decision Tree)或比较树(Comparison Tree)**。



注意：判定树的形态只与表结点个数n相关，而与输入实例中arr[low…..high]的取值无关。

（1）二分查找判定树的组成

①圆结点即树中的内部结点。树中圆结点内的数字表示该结点在有序表中的位置。

②外部结点：圆结点中的所有空指针均用一个虚拟的方形结点来取代，即外部结点。

③树中某结点i与其左(右)孩子连接的左(右)分支上的标记"<"、"("、">"、")"表示：当待查关键字K<R[i].key(K>R[i].key)时，应走左(右)分支到达i的左(右)孩子，将该孩子的关键字进一步和K比较。若相等，则查找过程结束返回，否则继续将K与树中更下一层的结点比较。

（2）二分查找判定树的查找

二分查找就是将给定值K与二分查找判定树的根结点的关键字进行比较。若相等，成功。否则若小于根结点的关键字，到左子树中查找。若大于根结点的关键字，则到右子树中查找。

【例】对于有11个结点的表，若查找的结点是表中第6个结点，则只需进行一次比较；若查找的结点是表中第3或第9个结点，则需进行二次比较；找第1，4，7，10个结点需要比较三次；找到第2，5，8，11个结点需要比较四次。

由此可见，成功的二分查找过程恰好是走了一条**从判定树的根到被查结点的路径**，经历**比较**的关键字次数恰为**该结点在树中的层数**。若查找失败，则其比较过程是经历了一条从判定树根到某个外部结点的路径，所需的关键字比较次数是该路径上内部结点的总数。

【例】待查表的关键字序列为：(05，13，19，21，37，56，64，75，80，88，92)，若要查找K=85的记录，所经过的内部结点为6、9、10，最后到达方形结点"9-10"，其比较次数为3。

实际上方形结点中"i-i+1"的含意为被查找值K是介于R[i].key和R[i+1].key之间的，即R[i].key<K<R[i+1].key。

② 二分查找的平均查找长度

设内部结点的总数为**n=2h-1**，则判定树是深度为**h=lg(n+1)**的满二叉树(深度h不计外部结点)。树中第k层上的结点个数为2k-1，查找它们所需的比较次数是k。因此在等概率假设下，二分查找成功时的平均查找长度为：

ASLbn≈lg(n+1)-1

二分查找在查找失败时所需比较的关键字个数不超过判定树的深度，在最坏情况下查找成功的比较次数也不超过判定树的深度。即为：



二分查找的最坏性能和平均性能相当接近。

# 二分查找的优缺点

### 二分查找的优点

折半查找的时间复杂度为O(logn)，远远好于顺序查找的O(n)。

### 二分查找的缺点

虽然二分查找的效率高，但是**要将表按关键字排序**。而排序本身是一种很费时的运算。即使采用高效率的排序方法也要**花费O(nlgn)**的时间。

# 二分查找的Java代码

递归实现与while循环实现，一般使用**while循环**比较常用。

## BinarySearch

public int binarySearch(int[] arr,int low,int high,int value){

while(low <= high){

int middleIndex = (low+high)/2;

**if(value < arr[middleIndex]){**

**high = middleIndex - 1;**

**}else if(value > arr[middleIndex]){**

**low = middleIndex + 1;**

**}else{**

**return middleIndex;//返回查找的索引**

**}**

}

return -1;//没有找到

}

测试：

@Test

public void test(){

int[] arr = {-1,2,8,32,54,89,100,443};

int index = binarySearch(arr,0,arr.length-1,100);

System.out.println("index:"+index);

}

## 变种1：查找第一个与value相等的元素

查找第一个相等的元素，也就是说等于查找key值的元素有好多个，返回这些元素最左边的元素下标。

public int binarySearchFirst(int[] arr,int low,int high,int value){

while(low <= high){

int middleIndex = (low+high)/2;

if(value < arr[middleIndex]){

high = middleIndex - 1;

}else if(value > arr[middleIndex]){

low = middleIndex + 1;

}else{

//**//注意防止索引越界**

**while(--middleIndex >= low &&arr[middleIndex] == value);**

return middleIndex+1;//返回查找的索引

}

}

return -1;//没有找到

}

## 变种2：查找最后一个与value相等的元素

查找第一个相等的元素，也就是说等于查找key值的元素有好多个，返回这些元素最右边的元素下标。

public int binarySearchFirst(int[] arr,int low,int high,int value){

while(low <= high){

int middleIndex = (low+high)/2;

if(value < arr[middleIndex]){

high = middleIndex - 1;

}else if(value > arr[middleIndex]){

low = middleIndex + 1;

}else{

//**//注意防止索引越界**

**while(++middleIndex <= high &&arr[middleIndex] == value);**

return middleIndex-1;//返回查找的索引

}

}

return -1;//没有找到

}