# 查找

## 定义

根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。

## 分类

1）静态查找和动态查找；

注：静态或者动态都是针对查找表而言的。动态表指查找表中有删除和插入操作的表。

　　2）无序查找和有序查找。

无序查找：被查找数列有序无序均可；

有序查找：被查找数列必须为有序数列。

## 性能

平均查找长度（Average Search Length，ASL）：需和指定key进行比较的关键字的个数的期望值，称为查找算法在查找成功时的平均查找长度。

　　对于含有n个数据元素的查找表，查找成功的平均查找长度为：ASL = Pi\*Ci的和。

　　Pi：查找表中第i个数据元素的概率。

Ci：找到第i个数据元素时已经比较过的次数。

# 顺序表查找

## 基本思想

顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值k相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于k的结点，表示查找失败。

## 具体步骤

## 代码实现

## 性能分析

查找成功时的平均查找长度为：（假设每个数据元素的概率相等） ASL = 1/n(1+2+3+…+n) = (n+1)/2 ;

　　当查找不成功时，需要n+1次比较，时间复杂度为O(n);

　　所以，顺序查找的时间复杂度为O(n)。

# 有序表查找

## 折半查找

### 基本思想

也称为是折半查找/二分查找，属于**有序查找**算法。

用给定值k先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

前提条件：有序数组

二分查找寻找中间索引点需要注意：

### 具体步骤

### 代码实现

#### 递归



#### 循环



#include <stdio.h>

#include <stdlib.h>

//二分查找

/\*

在一个有序序列中查找某一个元素是否存在

如果存在，返回该元素在该序列中的索引位置

如果不存在，返回-1

\*/

/\*开区间实现：

如果high = n-1,while(high >= low) high = middle -1;

如果high = n while(high > low) high = middle;

\*/

int binary\_search(int\* array,int n,int key)

{

/\*闭区间实现\*/

int low = 0;

int high = n-1;

int mid;

if(high < low)

return -1;

while(high >= low)

{

mid = low+((high-low)>>1);

/\*二分查找中间值获取的标准写法，常见不规范的写法：

(high-low)/2或low+(high-low)/2

\*/

if(array[mid] == key)

return mid;

else if(array[mid] > key)

high = mid-1;

else

low = mid+1;

}

return -1;

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10};

printf("%d\n",binary\_search(array,10,4));

return 0;

}

### 性能分析

### 应用

**二分搜索常见的应用场景：**

1. 在有序序列中查找一个数，时间复杂度为O(logN)；
2. 并不一定非要在有序序列中才能得到应用。

**二分搜索常见考察点：**

1. 对于边界条件的考察（中间划分点的设计和循环终止条件）以及代码实现的能力；
2. 二分搜索常见题目的变化：

2.1 给定处理或查找的对象不同；

2.2 判断条件不同；

2.3 要求返回的内容不同。

1. 在有序循环数组中进行二分查找（如循环数组）

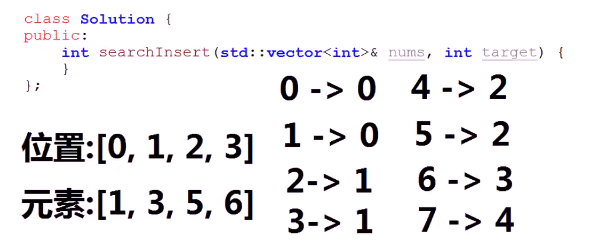
**二分查找重要提醒：**

经典的写法：mid=(left+right)/2（left+right可能溢出）

更安全的写法：mid=left+(right-left)/2

#### 插入位置

**题目：**给定一个排序数组nums（无重复元素）与目标target，如果target在nums里出现，则返回target所在下标，如果target在nums里未出现，则返回target应该插入位置的数组下标，使得target插入数组nums后，数组仍然有序。



注：Leetcode 35

**分析：**

1. 当target在nums中出现时，二分查找的流程无变化；
2. 当target在nums没有出现时：

2.1如果target<nums[mid]，且target>nums[mid+1]；

2.2如果target>nums[mid]，且target<nums[mid+1]；

1. 当mid=0或者mid=nums.size()-1时，边界条件应该如何处理？

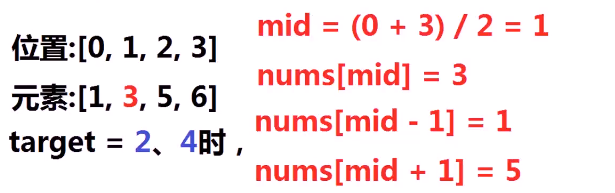
**算法思路：**

设元素所在的位置（或最终需要插入的位置）为index，在二分查找的过程中：

如果target==nums[mid]：index=mid；

如果target<nums[mid]，且（mid==0或target>nums[mid-1]）：index=mid；

如果target>nums[mid]，且（mid==nums.size()-1或target<nums[mid+1]）：index=mid+1；

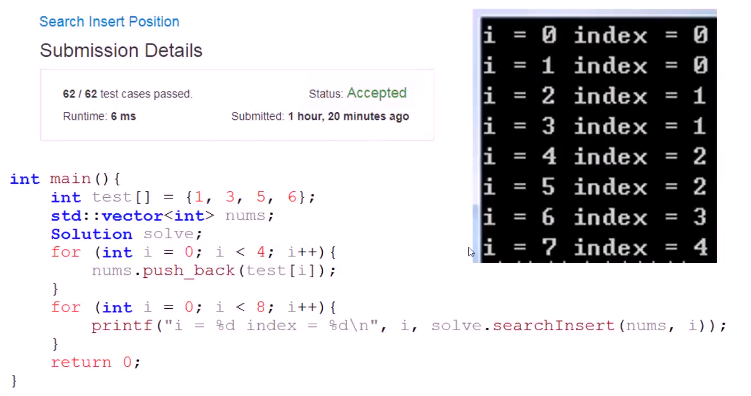




**代码：**

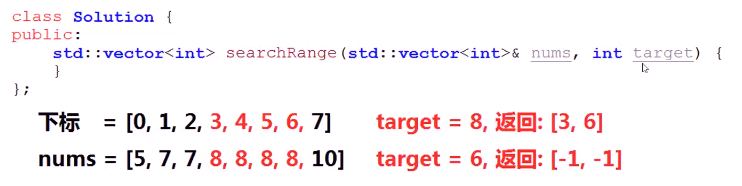


**测试：**



#### 区间查找

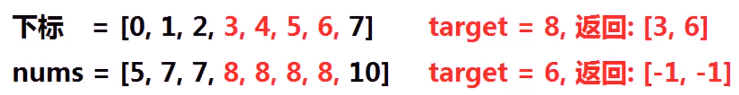
**题目：**给定一个排序数组nums（nums中有重复元素）与目标值target，如果target在nums中出现，则返回target所在区间的左右端点下标，[左端点，右端点]，如果target在nums里未出现，则返回[-1,-1]。



注：Leetcode 34

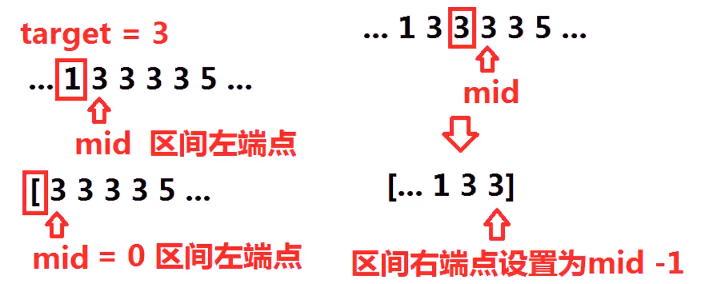
**分析：**

1. 可否直接通过二分查找，很容易同时求出目标target所在区间的左右端点？
2. 若无法同时求出区间左右端点，将对目标target的二分查找增加怎样的限制条件，就可分别求出目标target所在区间的左端点与右端点？



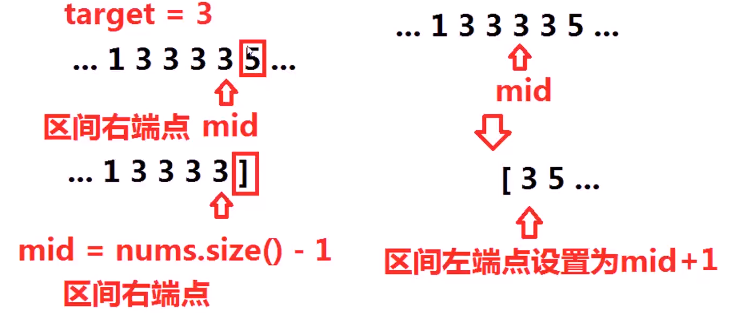
查找区间左端点时，增加如下限制条件：

当target==nums[mid]时，若此时mid==0或nums[mid-1]<target，则说明mid即为区间左端点，返回；否则设置区间右端点为mid-1。

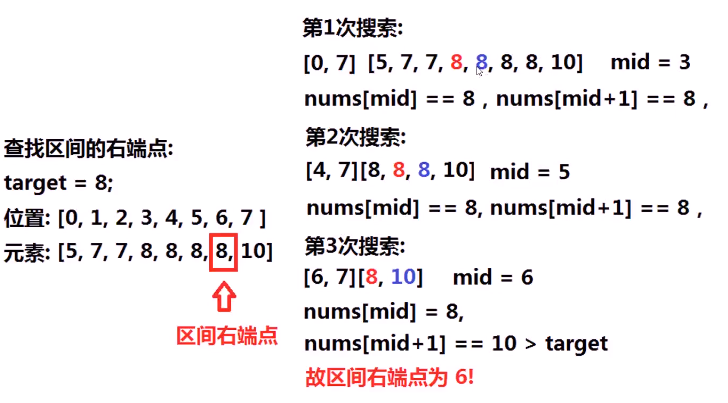


查找区间右端点时，增加如下限制条件：

当target==nums[mid]时，若此时mid==nums.size()-1或nums[mid+1]>target，则说明mid即为区间右端点；否则设置区间左端点为mid+1



算法思路：

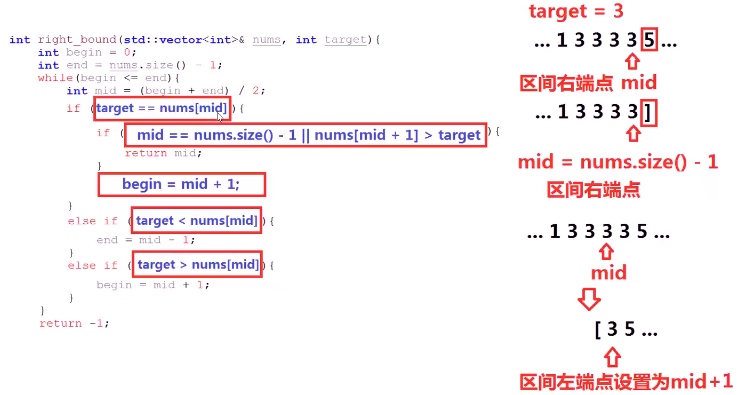


**代码：**

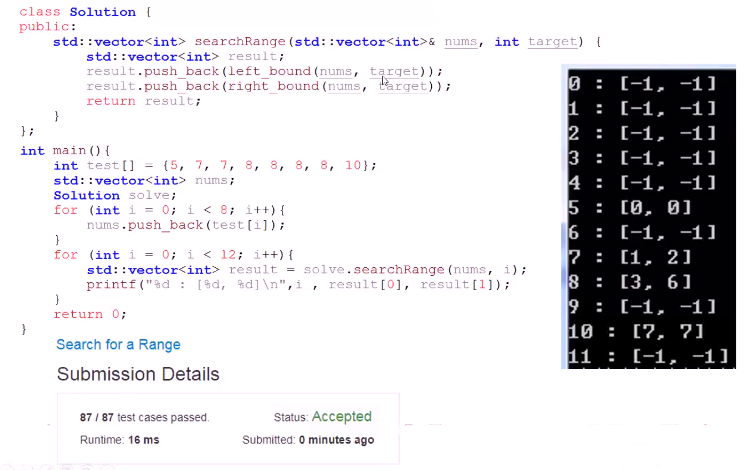
设置区间左端点：



设置区间右端点：



**测试：**



#### 查找缺失元素

题目：一个有N个元素的数组，每个元素都是[1,N+1]序列中的某一个，且序列中的每个数至多被包含一次，那么该序列中有一个数没被包含，如何查找缺失的那个数？

解题思路：将1…N+1这些数加起来，得到一个和sum，然后遍历数组，sum减去数组中的每一个值，最后的值就是缺失的元素。

**拓展：**

一个有N个元素的数组，每个元素都是[1,N+2]序列中的某一个，且序列中的每个数至多被包含一次，那么该序列中有2个数没被包含，如何查找缺失的那2个数？

思路：使用中位数，将数组分为2个子数组，然后按照前面的解题思路递归就可以解决问题。如果前后两个子数组各有一个缺失的数，则正好解决；如果都在前面的子数组中，则继续按照中位数划分前面的子数组，递归下去求解；如果两个缺失的数都在后面的子数组中，则继续按照中位数递归下去求解。

#### 查找元素的上下限

题目要求：在一个有序序列中查找某一元素的上下限。

代码：

#include <iostream>

#include <vector>

using namespace std;

//在一个有序序列中，查找某一个元素在该序列中的下限索引

//找到第一个不小于该元素的下标索引

int lower\_bound(int\* array,int low,int high,int key)

{

int mid;

if(high < low)

return -1;

//利用二分查找的策略

while(high >= low)

{

mid = low+((high-low)>>1);

if(array[mid] < key)

low = mid+1;

else

high = mid-1;

}

return low;

}

/\*

在一个有序序列中，查找某一个元素在该序列中的上限索引

找到第一个大于该元素的下标索引

\*/

int upper\_bound(int\* array,int low,int high,int key)

{

int mid;

if(high < low)

return -1;

while(high >= low)

{

mid = low+((high-low)>>1);

if(array[mid]>key)

high = mid-1;

else

low = mid+1;

}

return low;

}

/\*上述代码就是查找元素上下限的代码实现\*/

/\*

查找某一个元素在一个有序序列总的范围

包含该缘故的上下限 该区间为两端闭区间

如果没有找到就返回-1

\*/

pair<int,int> Findvalue(vector<int>& vec,int key)

{

pair<int,int> pos(-1,-1);

int mid,begin = 0,end = vec.size();

//首先判断该元素是否存在

while(begin <= end)

{

mid = begin + ((end-begin)>>1);

if(vec[mid] == key)

{

pos.first = mid;

pos.second = mid;

break;

}

else if(vec[mid] > key)

end = mid-1;

else

begin =mid+1;

}

if(vec[mid] != key)

return pos;

int low = mid-1;

int high = mid+1;

//找低地址

while(begin <= low)

{

mid = begin +(low-begin)/2;

if(vec[mid] < key)

begin = mid+1;

else

low = mid-1;

}

pos.first = begin;

//找高地址

while(high <= end)

{

mid = high +(end-high)/2;

if(vec[mid] > key)

end = mid-1;

else

high = mid+1;

}

pos.second = high -1;

return pos;

}

int main()

{

int array[]={1,2,3,3,3,3,3,3,4,5,7};

cout<<lower\_bound(array,0,10,3)<<endl;

cout<<upper\_bound(array,0,10,3)<<endl;

vector<int> vec(array,array+sizeof(array)/sizeof(int));

pair<int,int> pos = Findvalue(vec,3);

cout<<pos.first<<endl<<pos.second<<endl;

return 0;

}

拓展：熟悉STL模板库中查找元素上下限的函数，并阅读其代码实现。

#### 查找元素下限值位置

题目：给定一个有序数组arr，其中不含有重复元素，请找到满足arr[i]==i条件的最左的位置。如果所有位置上的数都不满足条件，返回-1。

#### 局部最小值

题目：给定一个无序数组arr，已知任意相邻的两个元素，值都不重复，请返回任意一个局部最小的位置。

所谓局部最小的位置是指，如果arr[0]<arr[1]，那么位置0就是一个局部最小的位置。如果arr[N-1]（也就是arr最右的数）小于arr[N-2]，那么位置N-1也是局部最小的位置。如果位置i既不是最左位置也不是最右位置。那么只要满足arr[i]同时小于它左右两侧的值即（arr[i-1]和arr[i+1]），那么位置i也是一个局部最小的位置。

#### 寻找多个集合的交集

题目：有两个无序序列，寻找两个序列的交集。

思考：如果拓展为多个无序序列呢？

代码：

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

/\*

查找两个**无序**数组的交集

\*/

void FindElements(vector<int>& first,vector<int>& second)

{

if(first.size() <=0 || second.size() <= 0)

return;

//先排序

sort(first.begin(),first.end());

/\*使用标准STL中的sort，底层实现是快排\*/

sort(second.begin(),second.end());

//查找元素个数少的数组在另一个数组中是否出现

if(first.size() >second.size())

first.swap(second);

/\*这个判断是保证first始终是元素少的数组\*/

int i,key;

for(i =0;i<first.size();i++)

{

key = first[i];

int low = 0;

int high = second.size()-1;

//以first数组中的元素为基准 在second数组中进行二分查找

while(low<= high)

{

int mid = low +(high-low)/2;

//如果找到说明是交集中的元素，那么打印出来，再判断下一个元素是否可以找到

if(key == second[mid])

{

cout<<second[mid]<<endl;

break;//相等则跳出，否则死循环

}

else if(key > second[mid])

low = mid+1;

else

high = mid-1;

}

}

}

int main()

{

int array[]={5,4,3,8,9,7,0};

int array1[]={11,8,34,3,4,8,9,2};

vector<int> first(array,array+sizeof(array)/sizeof(int));

vector<int> second(array1,array1+sizeof(array1)/sizeof(int));

FindElements(first,second);

return 0;

}

拓展：

1、如果是多个无序序列，最优的时间复杂度是多少？

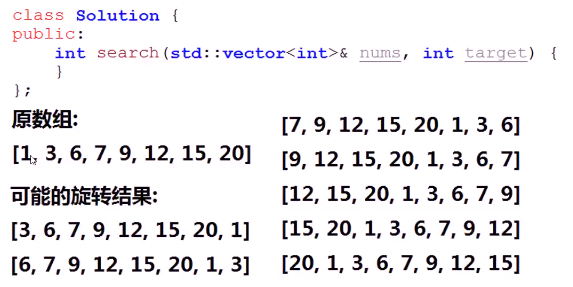
2、一个有N个元素的数组，每个元素都是[1,N+2]序列中的某一个，且序列中每个数至多被包含一次，那么该序列中有两个数没被包含，如何查找没被包含的两个数？

比如有100个元素，以50为基准，左右看缺少多少个，然后递归。

#### 旋转数组中的查找

**题目：**在一个无重复元素的旋转数组中查找某一个元素是否存在（这里的旋转数组是指原来一个升序的无重复元素的数组经过向右旋转得到）。

给定一个排序数组nums（nums中有无重复元素），且nums可能以某个未知下标旋转，给定目标值target，求target是否在nums中出现，若出现返回所在下标，未出现返回-1。



注：Leetcode 33

说明：旋转数组是指{1,2,3,4,5,6,7,8}🡪{4,5,6,7,8,1,2,3}

**分析：**

在旋转数组[7,9,12,15,20,1,3,6]中，若使用未加修改的二分查找，查找target=12或target=3，则出现如下情况：

当前mid=3，nums[mid]=15：

查找target=12：target(12)<nums[mid](15)，则在子区间[7,9,12]中继续查找，可以找到12，返回正确结果。

查找target=3：target(3)<nums[mid](15)，则在子区间[7,9,12]中继续查找，不可找到3，返回错误结果。

思考：二分查找是否可以继续使用，如希望得到正确的结果，应如何修改？

对于旋转数组：[7,9,12,15,20,1,3,6]，nums[begin]>nums[end]

查找target=12，由于target(12)<nums[mid](15)，查找正确的原因：

1. nums[begin](7)<nums[mid](15)，区间[7,9,12,15]顺序递增；
2. target(12)>nums[begin](7)，故target(12)只可能在顺序递增区间[7,9,12,15]中。

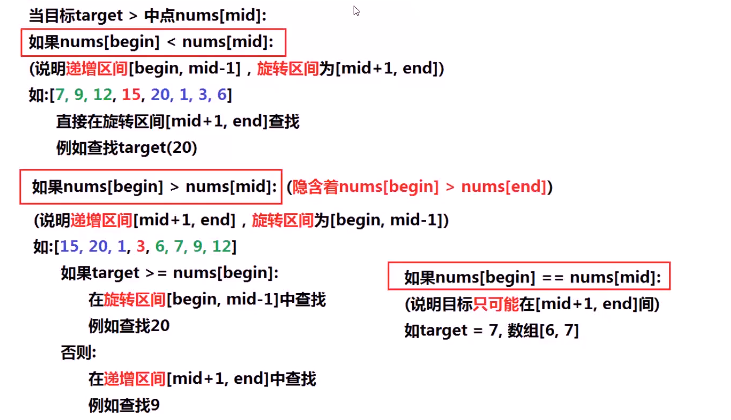
在查找target=3时，由于target(3)<nums[mid](15)，查找错误的原因：

1. nums[begin](7)<nums[mid](15)，区间[20,1,3,6]包括旋转点，为旋转区间；
2. target(3)<nums[begin](7)，故target(3)可能在旋转区间[20,1,3,6]中，此时忽略了该情况。

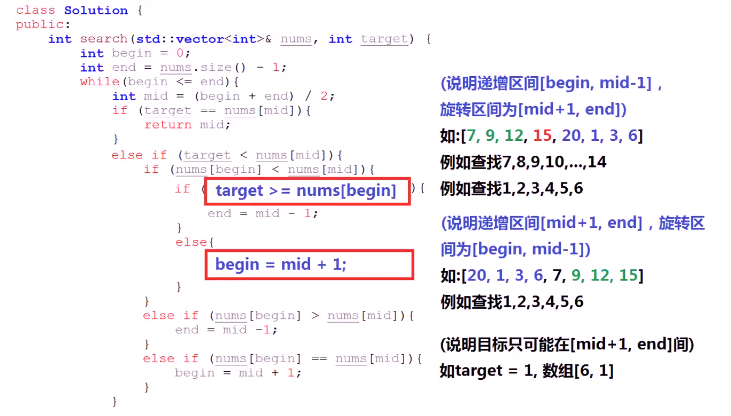
结论：若希望使用二分查找，需要修改二分查找，将可能在旋转区间[20,1,3,6]的情况考虑进去。

算法思路：



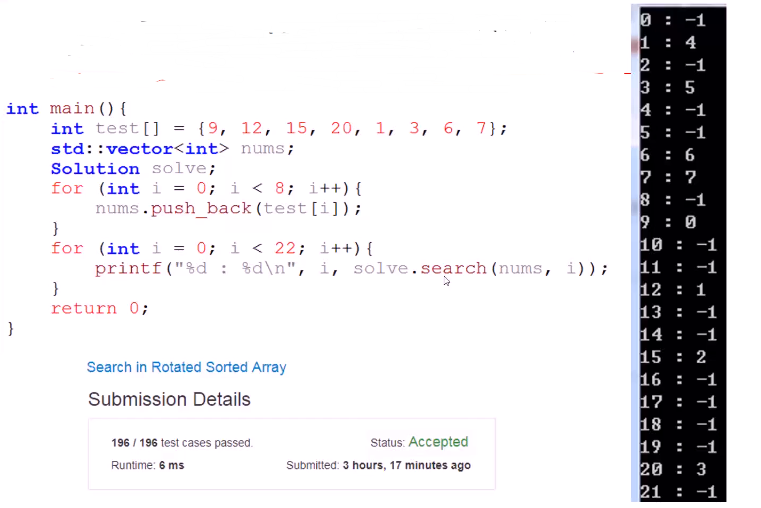


**代码：**





**测试：**



代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

在一个无重复元素的已经被旋转过的数组中查找某一个元素

如果该元素存在该旋转数组中，返回1

如果该元素不存在该旋转数组中，返回0

\*/

bool RotateArray(vector<int>& vec,int key)

{

int low =0,high = vec.size()-1;

int mid;

while(low<=high)

{

mid = low + (high-low)/2;

if(vec[mid] == key)

return 1;

//根据两个子数组的性质来划分情况

if(vec[mid]>vec[low]) //前半部分一定是升序

{

if(key >= vec[low]&& key<vec[mid])

//说明待查找的元素在升序子序列中

high =mid-1;

else//剩下的情况继续在旋转数组中查找

low = mid+1;

}

else //前半部分不是升序，包含了经过旋转的部分

{

if(key > vec[mid]&& key<= vec[high])

low= mid+1;

else

high = mid-1;

}

}

return 0;

}

/\*

在一个有重复元素的旋转数组中查找某一个元素

判断该元素是否存在

\*/

bool RotateArray(vector<int>& vec,int key)

{

int low = 0,high=vec.size()-1;

int mid;

//同样使用二分查找的思想

while(low <= high)

{

mid = low+(high-low)/2;

if(vec[mid] == key)

return 1;

if(vec[mid] > vec[low])//前半部分是升序

{

if(key >= vec[low] && key < vec[mid])

high = mid-1;

else

low = mid+1;

}

else if(vec[mid] < vec[low]) //前半部分是降序

{

if(key > vec[mid] && key <= vec[high])

low = mid+1;

else

high = mid-1;

}

else

low++;

}

return 0;

}

int main()

{

int array[]={4,5,6,7,0,1,2};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

cout<<RotateArray(vec,9)<<endl;

}

#### 完全二叉树结点个数

题目：给定一颗完全二叉树的头结点head，返回这棵树的节点个数。如果完全二叉树的节点数为N，请实现时间复杂度低于O(N)的解法。

#### 求K的N次方

题目：如果更快的求一个整数k的N次方。如果两个整数相乘并得到结果的时间复杂度为O(1)，得到整数k的N次方的过程请实现时间复杂度为O(logN)的方法。

## 差值查找

### 基本思想

### 具体步骤

### 代码实现

### 性能分析

## 斐波那契查找

### 基本思想

### 具体步骤

### 代码实现

### 性能分析

# 线性索引查找

## 稠密索引

## 分块索引

## 倒排索引

# 二叉排序树

## 平衡二叉树（AVL树）

## 多路查找树（B树）

# 散列表查找（哈希表）

# 应用

## 岛屿数量

## 词语阶梯

## 火柴棍摆正方形

## 收集雨水