需求分析

通过设计访问频度提高查找性能的方案是可行的，接下来先分析这种方案，然后提出几个其他可能的方案。

用户要从数据集中查找/取出一个元素，我们想让这个元素靠前进而更容易被用户找到，因此直观的思考就是对元素按频率进行排序，用户使用多的元素放在前面，这样用户每次新建查询时就会话费很少的开销进行查找。每次查找都动态的对查找元素的频率加一，动态改变其位置。

实现这种操作我最先想到了使用大顶堆进行元素的动态过滤。

推荐作业的同学有的提出了类似的方案，不过我直接使用大顶堆，每次查询就把频率加一，然后在堆的结构上更进一步讨论查询者如何能按照严格排序的顺序从堆中拿出元素，毕竟单纯的大顶堆只是一个能够部分排序的容器，无法严格按照频率从大到小排序，需要我们做出一些调整。

除了线性表之外，还可以用其他类型的数据结构来实现链表，比如树状结构，在这里不实现具体的，只是思考几种可能：

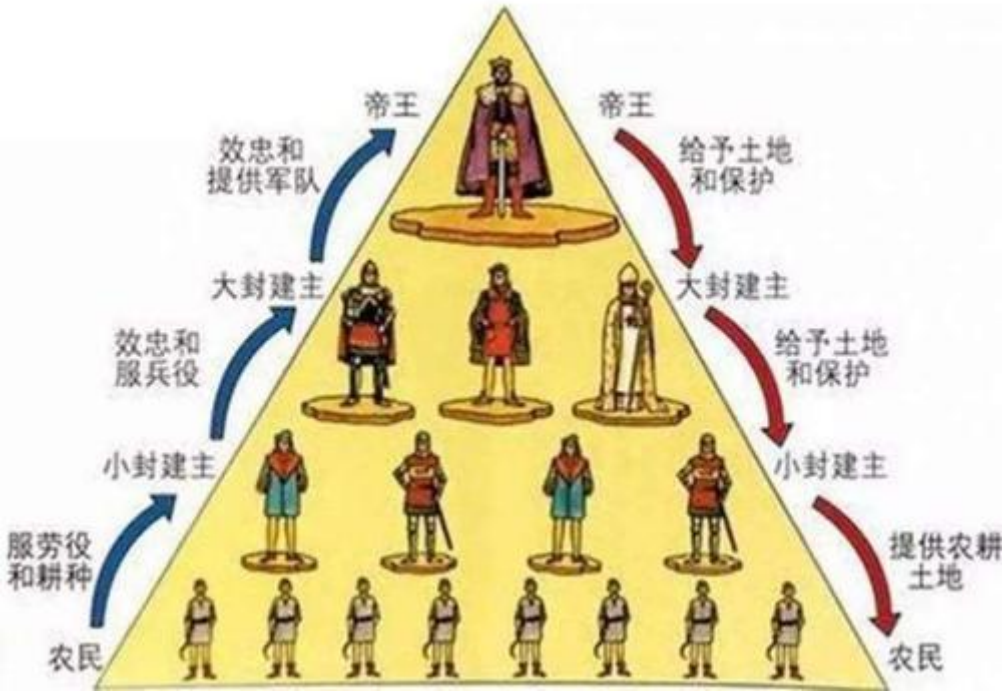
- 最小堆：最小堆非常符合我们的要求，唯一需要改变的，就是将频度作为优先级。为了适应最小堆，我们其实可以改成每次搜索、查找之后将频度减一，看上去是一个双向链表的优化，但是相对难维护一些。

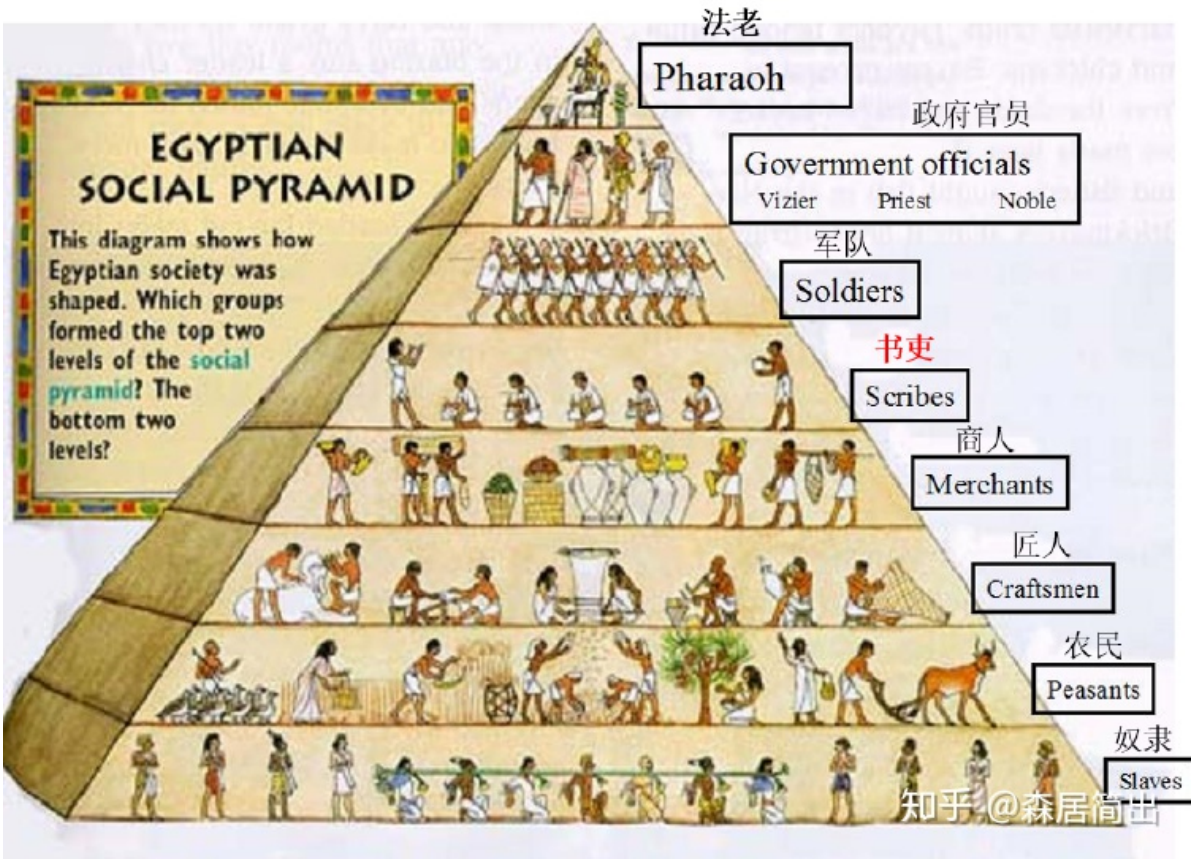
实现思路-大顶堆

堆可以通过动态的上滤和下滤操作实现元素的移动，我们只需要每次搜索把元素的频率加一，然后把这个元素进行上滤操作即可。

堆结构

我们这里讨论基于完全二叉树的二叉堆，为什么选择树形结构作为堆结构呢？因为堆就像一个“金字塔”结构，而树状结构和这种结构很相似，厉害人的在顶部，向下延伸出多个下属。二叉堆的堆顶永远是最大元素/最小元素。





二叉堆结构

基于二叉堆独特的索引规则，我们就可以用数组实现二叉堆了。

■ 二叉堆的逻辑结构就是一棵完全二叉树，所以也叫完全二叉堆

■ 鉴于完全二叉树的一些特性，二叉堆的底层（物理结构）一般用数组实现即可

■ 索引 i 的规律（ n 是元素数量）

□ 如果 i = 0 ，它是根节点

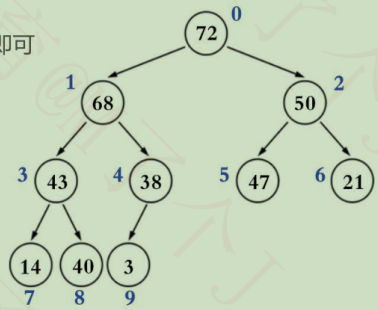
□ 如果 i > 0 ，它的父节点的索引为 floor((i – 1) / 2)

□ 如果 2i + 1 ≤ n – 1，它的左子节点的索引为 2i + 1

□ 如果 2i + 1 > n – 1 ，它无左子节点

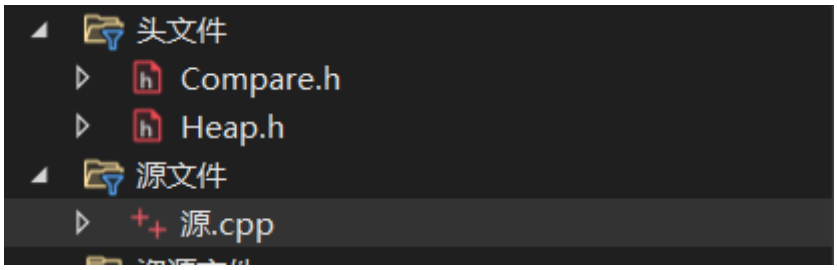
□ 如果 2i + 2 ≤ n – 1 ，它的右子节点的索引为 2i + 2

□ 如果 2i + 2 > n – 1 ，它无右子节点



0	1	2	3	4	5	6	7	8	9
72	68	50	43	38	47	21	14	40	3

具体实现



- Heap.h ---------- 实现二叉堆逻辑
- Compare.h----------实现堆内部的比较逻辑，为了兼容更多的数据类型因此抽象出一个仿函数比较器，便于后续添加新的比较规则。
- 源.cpp ----------测试文件,时间测试

Heap.h的实现

- 注： 代码中其他除了搜索外的冗余函数并不是真的冗余，而是为了方便我后续进一步开发

```
#pragma once
#include<cmath>
#include<iostream>
#include<vector>
#include"Compare.h"
#include<cassert>



template<class T>
struct Node
{
    T ele;
    int frequency;
    Node() {}
    Node(T e, int f = 0) :ele(e), frequency(f) {};
};


```

```cpp

/*  堆：在不排序的基础上得到最值，原理是通过添加时的上溢操作*/
constexpr int DEFAULT_CAPACITY = 8;


template<class T>
class BinaryHeap : Node<T>
{
private:

    int capacity;
    int m_size;
    Node<T>* danamicArray;
    Compare cmp;

public:
    BinaryHeap();

    void clear();

    int size();

    bool empty();
// 初始时添加元素
    void add(T Ele);
// 添加节点
    void addNode(Node<T> node);

    void printArray();
// 查找
    bool Search(T Ele);


private:

    bool FindEle(T Ele);


    int Indexof(T Ele);


    // 根据查找添加元素频率
    void SearchAdd(T ele);

    // 动态建堆
    void heapify();

    // 删除堆顶元素
    Node<T> remove();

    // 下滤操作
    void siftDown(int index);


    /// <summary>
    /// 根据某节点的索引计算其父亲节点索引
    /// <返回父节点索引></returns>
    int Pindex(int index);
```

```
79
80      /// <summary>
81      /// 检测索引是否合法
82      /// </summary>
83      /// <param name="index"></param>
84      void CheckIndex(int index);
85
86      /// <summary>
87      /// 上滤操作
88      /// 将添加的大元素向上传递
89      /// 采用非交换形式，减少赋值次数
90      /// <param name="index"></param>
91      void siftUp(int index);
92
93
94      /// <summary>
95      /// 动态扩容
96      /// </summary>
97      /// <param name="capa"></假定容量>
98      void ensureCapacity(int capa);
99
100 };
101
102
103
104
105 template<class T>
106 BinaryHeap<T>::BinaryHeap() {
107     m_size = 0;
108     capacity = DEFAULT_CAPACITY;
109     danamicArray = new Node<T>[capacity];
110 }
111
112 template<class T>
113 void BinaryHeap<T>::clear()
114 {
115     delete[] danamicArray;
116     m_size = 0;
117     //danamicArray = new Node<T>[capacity];
118 }
119
120
121 template<class T>
122 int BinaryHeap<T>::size()
123 {
124     return m_size;
125 }
126
127
128 template<class T>
129 bool BinaryHeap<T>::empty() {
130     return m_size == 0;
131 }
132
133
134 template<class T>
135 void BinaryHeap<T>::add(T Ele) {
136     if (this->FindEle(Ele))
```

```
    {
        danamicArray[Indexof(Ele)].frequency++;
        return;
    }

    /* 先添加一个新元素到尾部 */
    ensureCapacity(m_size + 1);
    /* 添加新元素到堆的最后一个位置 */
    danamicArray[m_size] = Node<T>(Ele, 0);
    m_size++;
    /* 向上过滤 */
    int index = m_size - 1; // 针对最后一个元素进行
}



template<class T>
void BinaryHeap<T>::addNode(Node<T> node) {

    /* 先添加一个新元素到尾部 */
    ensureCapacity(m_size + 1);
    /* 添加新元素到堆的最后一个位置 */
    //Node<T>* newNode = new Node<T>(Ele, 0);
    danamicArray[m_size] = node;
    m_size++;
    /* 向上过滤 */
    int index = m_size - 1; // 针对最后一个元素进行
    siftUp(index);
}


template<class T>
void BinaryHeap<T>::printArray() {
    if (m_size == 0) {
        std::cout << "Heap is empty!" << std::endl;
        return;
    }

    for (int i = 0; i < m_size; i++) {
        std::cout << this->danamicArray[i].ele << "(" << i << ")" << ' ';
    }
    std::cout << std::endl;
}



template<class T>
bool BinaryHeap<T>::Search(T Ele)
{
    SearchAdd(Ele);  // 增加频率

    bool flag = false;
    int count = 0;
    for (int i = 0; i < m_size; i++)
    {
        if (Ele == this->danamicArray[0].ele)  // 比较堆顶元素
        {
            flag = true;
```

```
        }
        else
        {
            count++;
            remove();  // 删除堆顶元素（其实就是把堆顶元素和最后一个元素做了交换）
        }
    }

    Node<T>* oldNode = this->danamicArray + m_size;

    for (int i = 0; i < count; i++)
    {
        addNode(oldNode[i]);
    }
    return flag;
}


template<class T>
    bool BinaryHeap<T>::FindEle(T Ele)
    {
        return Indexof(Ele) == -1 ? false : true;
    }


    template<class T>
    int BinaryHeap<T>::Indexof(T Ele)
    {
        int index = -1;
        for (int i = 0; i < m_size; i++)
        {
            if (this->danamicArray[i].ele == Ele)
                index = i;
        }
        return index;
    }


    // 根据查找添加元素频率
    template<class T>
    void BinaryHeap<T>::SearchAdd(T ele)
    {
        if (FindEle(ele))
        {
            danamicArray[Indexof(ele)].frequency++;
            siftUp(Indexof(ele));
        }
    }



    // 动态建堆
    template<class T>
    void BinaryHeap<T>::heapify()
    {
        // 自下而上的下滤
        // 从最后一个非叶子节点开始
        for (int index = (m_size >> 1) - 1; index >= 0; index--)
```

```
        {
            siftDown(index);
        }
    }


    // 删除堆顶元素
    template<class T>
    Node<T> BinaryHeap<T>::remove() {

        // 得到堆顶，用于返回
        Node<T> deleEle = this->danamicArray[0];

        /* size减一，为了避免最后一个元素丢失，把其复制到堆顶 */
        Node<T> temp = danamicArray[m_size - 1];
        danamicArray[m_size - 1] = deleEle;
        this->danamicArray[0] = temp;

        m_size--;

        /* 现在可能造成堆顶小、两边大的格局，所以对堆顶进行下移操作*/
        int index = 0;
        // 只有非叶子节点才能下溢
        siftDown(index);

        return deleEle;
    }


    // 下滤操作
    template<class T>
    void BinaryHeap<T>::siftDown(int index)
    {

        Node<T> element = this->danamicArray[index];

        int half = (this->m_size) >> 1;
        while (index < half) { // index必须是非叶子节点
            // 默认是左边跟父节点比
            int childIndex = (index << 1) + 1;
            Node<T> child = this->danamicArray[childIndex];

            int rightIndex = childIndex + 1;
            // 右子节点比左子节点大
            if (rightIndex < m_size &&
                cmp(child.frequency, danamicArray[rightIndex].frequency)) {
                child = this->danamicArray[childIndex = rightIndex];
            }

            // 大于等于子节点
            if (this->cmp(child.frequency, element.frequency)) break;

            this->danamicArray[index] = child;
            index = childIndex;
        }
        this->danamicArray[index] = element;
    }

```

```
311
312
313     /// <summary>
314     /// 根据某节点的索引计算其父亲节点索引
315     /// <返回父节点索引></returns>
316     template<class T>
317     int BinaryHeap<T>::Pindex(int index) {
318         if (index == 0) return -1;
319         if (index % 2 == 0) return (index - 2) / 2;
320         return (index - 1) / 2;
321     }
322
323
324     /// <summary>
325     /// 检测索引是否合法
326     /// </summary>
327     /// <param name="index"></param>
328     template<class T>
329     void BinaryHeap<T>::CheckIndex(int index)
330     {
331         if (index < 0 || index >= m_size)
332             throw std::exception("index is inviled!");
333     }
334
335
336     /// <summary>
337     /// 上滤操作
338     /// 将添加的大元素向上传递
339     /// 采用非交换形式，减少赋值次数
340     /// <param name="index"></param>
341     template<class T>
342     void BinaryHeap<T>::siftUp(int index) {
343         CheckIndex(index);
344
345         Node<T> e = danamicArray[index]; // 保留新添加的元素值
346
347         while (index > 0)
348         {
349             int pindex = Pindex(index); // 计算父节点索引
350             Node<T> oldE = danamicArray[pindex]; // 得到父节点的数值
351
352             if (cmp(e.frequency, oldE.frequency)) break; // 比较父节点和新添加
    节点的大小
353                                                 // 如果新添加的元素值 < 父节点 ，break
354             danamicArray[index] = danamicArray[pindex]; // 否则把父节点的值向
    下赋值
355             index = pindex; // 更新索引为原来父节点的索引
356         }
357         // 最后index的位置就是新添元素最终上滤后的位置，进行赋值即可。
358         danamicArray[index] = e;
359     }
360
361
362     /// <summary>
363     /// 动态扩容
364     /// </summary>
365     /// <param name="capa"></假定容量>
366     template<class T>
```

```
367     void BinaryHeap<T>::ensureCapacity(int capa)
368     {
369         int oldCapacity = capa;
370         if (oldCapacity <= capacity) return;
371 
372         // 更新capacity
373         capacity = capacity + (capacity >> 1);
374         Node<T>* newArray = new Node<T>[capacity];
375 
376         for (int i = 0; i < capa; i++)
377         {
378             newArray[i] = danamicArray[i];
379         }
380 
381         delete[] danamicArray;
382         danamicArray = newArray;
383     }
384 
```

compare.h

```
1  #pragma once
2  
3  class Compare
4  {
5  public:
6  
7      template<class T>
8      bool operator()(T a, T b)
9      {
10         return (a < b) ? true : false;
11     }
12 };
```

源.cpp

```
1  #include<iostream>
2  #include<Windows.h>
3  #include <chrono>
4  
5  #include"Heap.h"
6  using namespace std;
7  
8  
9  int main() {
10 
11     BinaryHeap<int>* heap = new BinaryHeap<int>;
12     heap->add(1);
13     heap->add(2);
14     heap->add(3);
15     heap->add(4);
16     heap->add(5);
17     heap->add(6);
18     heap->add(7);
19     heap->add(2);
```

```
    heap->add(11);
    heap->add(12);
    heap->add(45);
    heap->add(61);
    heap->add(72);
    heap->add(22);
    heap->add(32);
    heap->add(24);
    heap->add(52);
    heap->add(62);
    heap->add(72);
    heap->add(22);
    heap->add(32);
    heap->add(42);
    heap->add(33);
    heap->add(63);
    heap->add(73);
    heap->add(23);
    heap->add(33);
    heap->add(43);
    heap->add(53);
    heap->add(63);
    heap->add(73);
    heap->add(27);
    heap->add(37);
    heap->add(47);
    heap->add(57);
    heap->add(67);
    heap->add(77);
    heap->add(27);
    heap->add(39);
    heap->add(40);
    heap->add(50);
    heap->add(60);
    heap->add(70);
    heap->printArray();

    // 比较耗时
    double fp_ms1 = 0.0;
    double fp_ms2 = 0.0;

    //auto startTime = std::chrono::high_resolution_clock::now();
    //auto endTime = std::chrono::high_resolution_clock::now();


    auto startTime1 = std::chrono::high_resolution_clock::now();
    cout << heap->Search(70) << endl;
    auto endTime1 = std::chrono::high_resolution_clock::now();
    fp_ms1 = (chrono::duration<double, std::milli>(endTime1 -
startTime1)).count();

    heap->Search(6);
    heap->Search(6);
    heap->Search(6);
    heap->Search(3);
    heap->Search(3);
    heap->Search(3);
    heap->Search(70);
```

```
    heap->Search(70);
    heap->Search(70);
    heap->Search(3);


    auto startTime2 = std::chrono::high_resolution_clock::now();
    cout << heap->Search(70) << endl;
    auto endTime2 = std::chrono::high_resolution_clock::now();
    fp_ms2 = (chrono::duration<double, std::milli>(endTime2 -
startTime2)).count();

    cout << "第一次查找: " << fp_ms1 << endl;
    cout << "第二次查找: " << fp_ms2 << endl;

    heap->printArray();

    system("pause");
    return 0;
}
```

分析：

```

    bool Search(T Ele)
    {
        SearchAdd(Ele);  // 增加频率

        int oldsize = this->m_size; // 保留size
        bool flag = false;

        for (int i = 0; i < m_size; i++)
        {
            if (Ele == this->danamicArray[0].ele)  // 比较堆顶元素
            {
                flag = true;
            }
            else
            {
                remove();  // 删除堆顶元素（其实就是把堆顶元素和最后一个元素做了交换）
            }
        }
        this->m_size = oldsize;
        heapify();  // 重新建堆

        return flag;
    }
```

- 初始添加元素，每个元素的频率都是0
- 每次查询，都对查询元素的频率+1，然后找到这个元素进行上滤操作，使其向堆顶靠近

- 每次查询，从堆顶比较元素，如果和堆顶元素相同就返回true代表查询成功；如果不是当前堆顶元素就把当前堆顶元素删除（后面会恢复），再比较新的堆顶元素和待查询元素是都相同.....因为每次堆顶的元素都是最大值，因此能够保证每次比较的元素都是当前频率最大的那个元素。
- 比较完成后恢复删除操作造成的影响，首先恢复堆的size，然后对非叶子节点进行自上而下的下滤操作O(n)（比自上而下的上滤更高效O(nlogn)），使得堆的所有元素按照最大堆的规则进行排序，等待下一次查询。

复杂度分析

- 上滤和下滤操作的复杂度都是O（logn）级别，删除后的恢复操作是O（n），查询时最坏情况是查询频率最低的那个元素，时间复杂度为O(nlogn)。最好情况为O（1）。
- 使用堆最大的开销是构建堆O(n)，构建好堆后实际的查询开销其实很小。
- 删除堆顶的复杂度O(logn)

优化

每次查询堆顶元素不是所需元素时删除堆顶，后面再恢复整个堆的堆结构，这个过程太复杂了，因此考虑在这里进行优化。

尝试了使用遍历操作，但是二叉堆不具备像AVL树那样的排序规则，因此难以实现高效比较搜索。

尝试不删除堆顶而是直接把其频率赋值为0，但是后期发现难以恢复

受到TOP-K问题启发想从堆中取出前K个最大值，后发现和本问题还是不匹配

受到优先队列的启发：不用删除后恢复，而是删除后再次添加！！堆恢复的时间复杂度为O(n)，而再次添加的时间复杂度为O(logn)。 而且仅仅添加被删除的那些点并不需要对所有点进行下滤操作，只需要对删除的那些点进行上滤操作即可,大大减少了没有必要的操作。

代码改进如下：

```
    bool Search(T Ele)
    {
        SearchAdd(Ele);  // 增加频率

        bool flag = false;
        int count = 0;
        for (int i = 0; i < m_size; i++)
        {
            if (Ele == this->danamicArray[0].ele)  // 比较堆顶元素
            {
                flag = true;
            }
            else
            {
                count++;
                remove();  // 删除堆顶元素（其实就是把堆顶元素和最后一个元素做了交换）
            }
        }

        // 很重要的一步： 找到原来被移除的那些堆顶元素
        Node<T>* oldNode = this->danamicArray + m_size;

        for (int i = 0; i < count; i++)
```

```
        {
            addNode(oldNode[i]); // 把那些被移除的对顶元素添回去
        }
        return flag;
    }
```

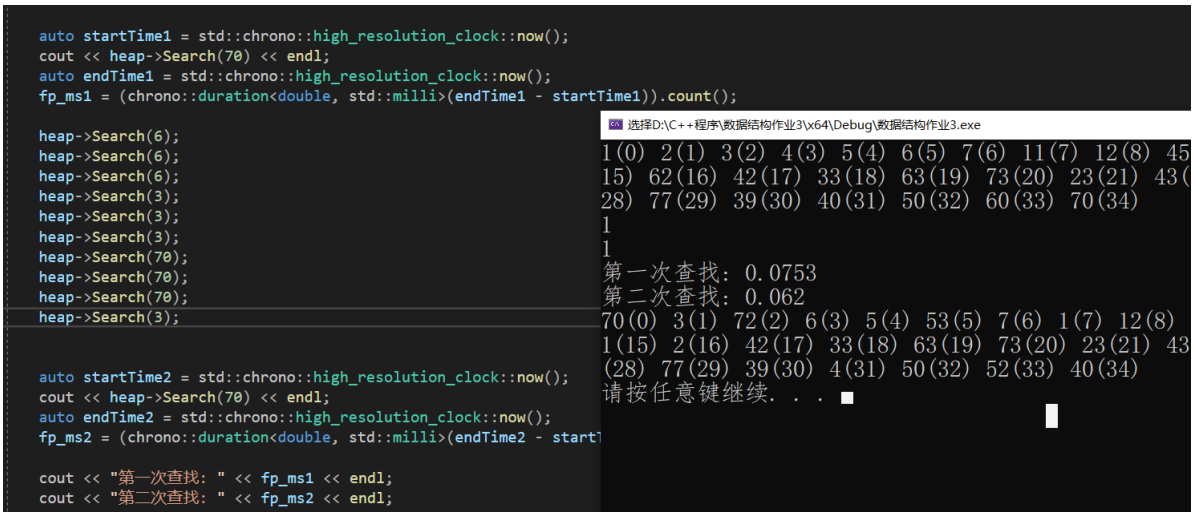
多了一个addNode()函数，用来添加原来删除的那些点：

```
    void addNode(Node<T> node) {
        /* 先添加一个新元素到尾部 */
        ensureCapacity(m_size + 1);
        /* 添加新元素到堆的最后一个位置 */
        danamicArray[m_size] = node;
        m_size++;
        /* 向上过滤 */
        siftUp(m_size - 1);
    }
```

优化后复杂度分析

- 最坏情况是把堆顶元素全都删除、然后再全都添加进来，时间复杂度为O(nlogn)。原先不论什么情况最后都要对所有非叶子节点进行下滤操作，现在除了最坏情况之外只需要处理被删除的节点即可。
- 最好情况O(1)

如下图，查找时间被优化了：



其他思路分析

1. 基于全排序的双向链表

每次插入时寻找合适的插入位置进行插入。优势在于查询时可以向前查询，也可以向后查询，效率更高。 **我个人认为这个比堆更稳定。**

2.基于排序的动态数组

优势在于结构简单，易于维护

上面的方法存在一个问题： 每次排序的复杂度至少为O(nlogn)，除非采用哈希等非比较排序。

3. 优先队列

实际上c++的优先队列是用大顶堆实现的，因此也可以直接用优先队列实现这个问题。

- 优先队列队头元素出队后再次添加到队列里，这一步不好实现
- 优先队列的比较逻辑：如果使用c++自带的优先队列，需要使用运算符重载。

开饭

其实针对这个问题我更喜欢另一个课上回答问题同学的答案，**那就是每次把当前访问的元素直接放到最前面**，这个方式类似于cache缓存机制，cache中用替换算法解决这个问题（FIFO、LRU、随机替换）；此外鸿蒙系统的智能应用推荐也采用了这个方法，直接把用户上一次点击的应用放在推荐位的首位。

此外，如果非要针对频率进行排序，我倒是认为不如不排，直接使用哈希映射不香吗？ 用户不管查什么都是O(1)的复杂度，实则是太方便了，根本不需要考虑频率问题，所有元素通通高速访问（数据量比较小的时候）。

此外，如果元素是单词（比如“abandon”），最好使用trie树进行储存，其对于单词元素的查询效率更高。