数据结构大作业报告

大作业 小蓝鲸的财务统计任务

计算机科学与技术系

181860055 刘国涛

邮箱:181860055@smail.nju.edu.cn

2020年1月1日

任务一：统计多个收入区间内的平均收入

任务描述

> 给定n个市民的收入和m个询问区间$[l_i,r_i]$
>
> 对于每个询问区间$[l_i,r_i]$，你的程序需要计算在l_i到r_i范围的收入的平均值

复杂度要求

时间复杂度$O((n+m)log(n+2m)$以内

算法描述

1. 将n个市民的收入进行排序得到数组 `income`
2. 计算收入的前序和数组 `presum`
3. 在每次询问区间$[l_i,r_i]$时，通过二分查找分别找到区间左右端点在数组 `income` 中对应的下标 `leftAddr`，`rightAddr`
4. 用前缀和数组计算出该区间的平均数$\frac{presum_{rightAddr}-presum_{leftAddr}}{rightAddr-leftAddr}$

二分查找：

- 二分查找时，数据中可能存在重复元素，在查找 `leftAddr` 时，若结果为重复元素，则需要返回重复元素所在区间的最左端；在查找 `rightAddr` 时，若结果存在重复元素，则需要返回重复元素所在区间的最右端
- 二分查找得到的区间是[`leftAddr`，`rightAddr`)
- 二分查找可能出现待查找元素 `target` 不是 `income` 数组中的元素，则返回 `income[i+1],income[i]<target<income[i+1]`

注意：如果$rightAddr - leftAddr$则直接输出0

复杂度分析

- 输入：$O(n+m)$
- 对 `income` 排序：堆排序平均时间复杂度$O(nlog(n))$
- 计算前缀和：$O(n)$
- m次询问：一次二分查找$O(log(n))$，询问复杂度为$O(mlog(n))$
- 计算平均数：$O(1)$

总时间复杂度为$O((n+m)log(n))$

代码实现

堆排序实现

```
void siftDown(int *a, int start, int m) {
    int i = start;
    int j = 2 * i + 1; //j是i的左子女位置
    int tmp = a[i];
    while (j <= m) {   //检查是否到最后位置
        if (j<m&&a[j]<a[j + 1])j++;      //让j指向两子女中的大者
        if (tmp >= a[j])break;  //大则不做调整
        else { a[i] = a[j]; i = j; j = 2 * j + 1; }//否则大者上移, i, j下降
    }
    a[i] = tmp;//回放tmp中暂存的元素
}
void HeapSort(int *a, int n) {
    int i;
    for (i = (n - 2) / 2; i >= 0; i--)  //将表转换为堆
        siftDown(a, i, n - 1);
    for (i = n - 1; i >= 0; i--) {      //对表排序
        int tmp = a[0];
        a[0] = a[i];
```

```
            a[i] = tmp;
            siftDown(a, 0, i - 1);
        }
    };
```

计算前缀和

```
for(int i=0;i<n;i++){           // get presum
    presum[i+1]=presum[i]+income[i];
}
```

区间询问

```
int check(int n,int m,int *income,ull *presum){
    int l,r;
    int leftAddr,rightAddr;
    for(int i=0;i<m;i++){
        scanf("%d%d",&l,&r);
        //所有的区间都是[l,r)
        leftAddr=binarySearch(income,0,n,l,0);
        rightAddr=binarySearch(income,0,n,r,1);

        int num=rightAddr-leftAddr;
        if(num==0)
            printf("0\n");
        else{
            printf("%llu\n",(presum[rightAddr]-
presum[leftAddr])/num);
        }

    }
}
```

二分查找

```
int binarySearch(int *arr,int l,int r,int target,bool
mode){
    register int left=l,right=r;
    register int mid;
    while(left<right){
        mid = (left+right)>>1;
```

```
        if(arr[mid]<target){
            left=mid+1;
        }
        else if(arr[mid]>target){
            right=mid;
        }
        else{
            //处理重复数据的方式
            if(mode){                   //区间最右
                right=r;
                while(mid<right&&arr[mid]==target)
                    ++mid;
            }
            else{                       //区间最左
                left=l;
                while(mid>=left&&arr[mid-1]==target)
                    --mid;
            }
            return mid;
        }
    }
    //target不在arr中
    //选择区间最右
    if(arr[mid]<target&&mid<r)
        ++mid;

    return mid;
}
```

任务二：实时统计收入的中位数

任务描述

> 对于n个市民，逐一获取每个市民的年收入。当获取第k个年收入时，统计并输出前k个年收入的中位数
>
> 注：当k为技术时，直接输出第$\frac{k+1}{2}$大的收入；当k为偶数时，输出第$\frac{k}{2}$和$\frac{k}{2}+1$大的收入的平均值(k>=1)

复杂度要求

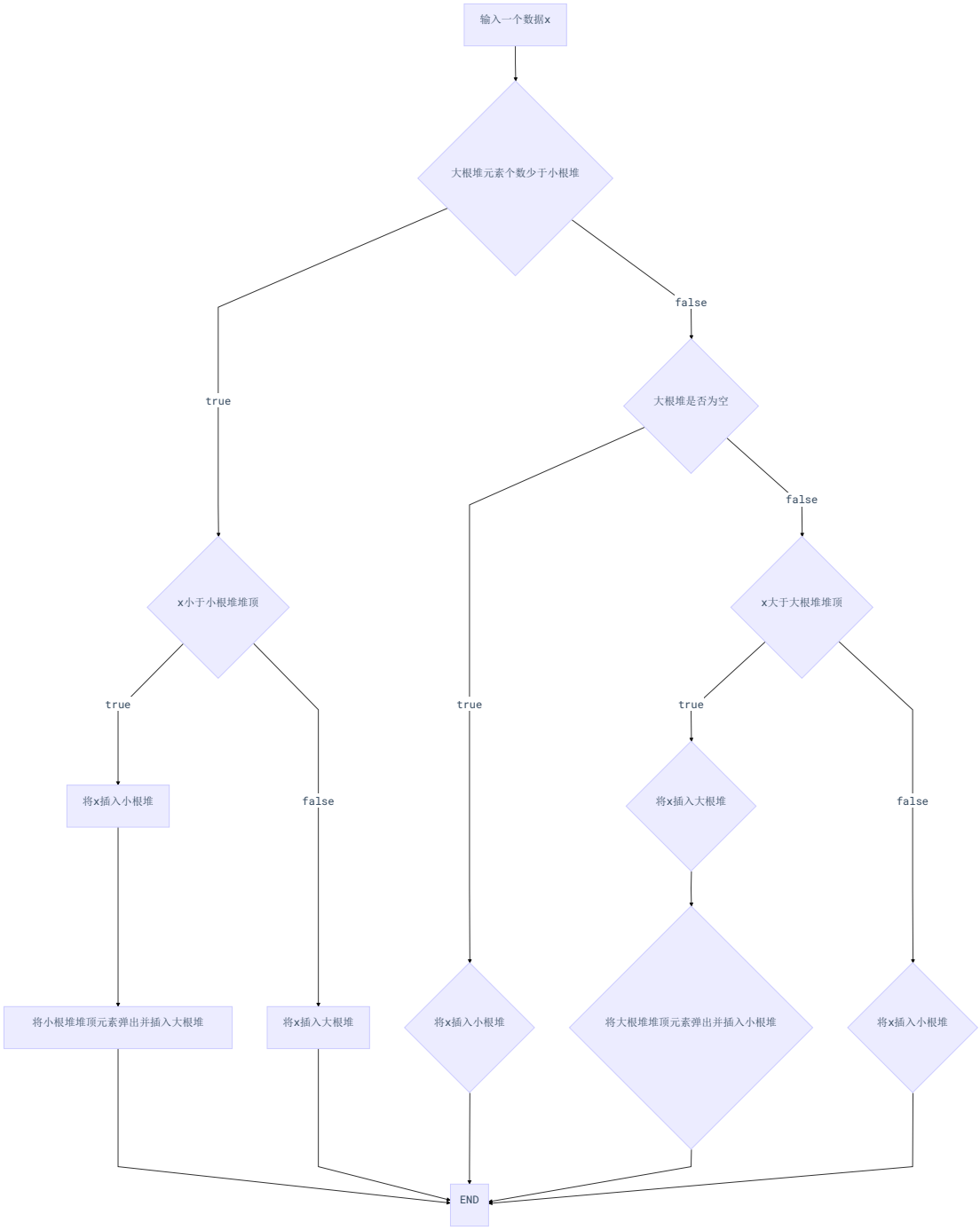
时间复杂度$O(nlog(n))$以内

算法描述

整体思路

1. 大根堆存放小k/2个数据，小根堆存放大k/2个数据
2. 这样大根堆的堆顶便是当前k个数据中第k/2小的数据，小根堆堆顶便是第k/2大的数据
3. 通过对两个堆顶元素的计算即可得到中位数

对两个堆之间的维护



复杂度分析

在输入第k个数据时，如果对一个堆直接插入需要 `log k` ，对两个堆之间进行数据调整需要 `3log k` ，因此维护两个堆的开销是$O(logk)$级别的

因此总时间复杂度是$O(nlogn)$

代码实现

小根堆模板实现

```
template<class T>
class MinHeap{
public:
    MinHeap(int size);
    MinHeap(T arr[],int n);
    bool insert(T ele);
    bool pop();
    bool isFull();
    bool isEmpty();
    void print();
    int Size();
    T top();
private:
    T *heap;
    int currentSize;
    int maxHeapSize;
    void siftDown(int start,int end);
    void siftUp(int start);
};

template<class T>
MinHeap<T>::MinHeap(int size){
    maxHeapSize=size;
    heap=new T[size];
    currentSize=0;
    maxHeapSize=size;
}

template<class T>
MinHeap<T>::MinHeap(T arr[],int n){
    maxHeapSize=n;
    heap=new T[maxHeapSize];
    currentSize=n;
    for(int i=0;i<n;i++){
        heap[i]=arr[i];
    }
    int currentPos=(n-2)/2;
    while(currentPos>=0){
        siftDown(currentPos,currentSize-1);
```

```
        currentPos--;
    }
}

template<class T>
bool MinHeap<T>::isFull(){
    return currentSize==maxHeapSize;
}

template<class T>
bool MinHeap<T>::isEmpty(){
    return currentSize==0;
}

template<class T>
void MinHeap<T>::siftDown(int start,int end){
    T tmp=heap[start];
    int i=start;
    int j=i*2+1;
    while(j<=end){
        if(j<end&&heap[j]>heap[j+1])j++;
        if(tmp>heap[j]){
           heap[i]=heap[j];
           i=j;
           j=i*2+1;
        }
        else{
            break;
        }
    }
    heap[i]=tmp;
}

template<class T>
void MinHeap<T>::siftUp(int start){
    T tmp=heap[start];
    int j=start,i=(j-1)/2;
    while(j>0){
        if(heap[i]>tmp){
            heap[j]=heap[i];
            j=i;
```

```
            i=(j-1)/2;
        }
        else{
            break;
        }
    }
    heap[j]=tmp;
}

template<class T>
bool MinHeap<T>::pop(){
    if(this->isEmpty())return false;
    heap[0]=heap[currentSize-1];
    siftDown(0,--currentSize-1);
    return true;
}

template<class T>
bool MinHeap<T>::insert(T ele){
    if(this->isFull())return false;
    heap[currentSize]=ele;
    siftUp(currentSize++);
    return true;
}

template<class T>
void MinHeap<T>::print(){
    for(int i=0;i<currentSize;i++){
        cout<<heap[i]<<' ';
    }
    cout<<endl;
}

template<class T>
int MinHeap<T>::Size(){
    return currentSize;
}

template<class T>
T MinHeap<T>::top(){
    return heap[0];
```

```
122  }
```

小根堆的数据的结构

```
struct lessInCome{
    int val;
    bool operator>(lessInCome& x){
        return val<x.val;
    }
    friend bool operator<(const int x,const lessInCome& y)
{
        return x<y.val;
    }
    lessInCome(int v):val(v){}
    lessInCome(){}
    friend ostream& operator<<(ostream& out,const
lessInCome& c){
        out<<c.val;
        return out;
    }
};
```

大根堆的数据的结构

```
struct moreInCome{
    int val;
    bool operator>(moreInCome& x){
        return val>x.val;
    }

    friend bool operator>(const int x,const moreInCome& y)
{
        return x>y.val;
    }
    moreInCome(int v):val(v){}
    moreInCome(){}
    friend ostream& operator<<(ostream& out,const
moreInCome& c){
        out<<c.val;
        return out;
    }
```

```
};
```

堆之间的维护

```
    MinHeap<lessInCome> minHeap(n);//小根堆存放大 k/2 个数据

    MinHeap<moreInCome> maxHeap(n);//大根堆存放小 k/2 个数据

    for(int i=0,x;i<n;i++){
        cin>>x;
        //维护两个堆
        if(maxHeap.Size()<minHeap.Size()){
            if(x<minHeap.top()){
                minHeap.insert(x);
                x=minHeap.top().val;
                minHeap.pop();
                maxHeap.insert(x);
            }
            else
                maxHeap.insert(x);
        }
        else{
            if(maxHeap.Size()==0){
                minHeap.insert(x);
            }
            else if(x>maxHeap.top()){
                maxHeap.insert(x);
                x=maxHeap.top().val;
                maxHeap.pop();
                minHeap.insert(x);
            }
            else
                minHeap.insert(x);
        }
        if(i&1)  //计算中位数
            ans=(minHeap.top().val+maxHeap.top().val)/2;
        else
            ans=minHeap.top().val;
        cout<<ans<<endl;
    }
```

任务三：实时统计当前最高的k个收入

任务描述

> 给定 n 个市民的年收入，以及查询参数 k，要求实时反馈出所有收入中最高的 k 个收入。收入信息按行给出，每行表示一个市民的年收入。 当该行的内容为“Check”字符串时，要求输出到上一条收入为止的所有收入中，最高的 k 个收入。如果 k 值超过了当前已经存储的收入数目，则仅需输出已有的数据。 输出在一行完成。

复杂度要求

时间复杂度$O(nklog(k))$以内

算法描述

数据存储

使用容量为k的小根堆MinHeap存储最高的k个收入

在每次插入时，如果堆满，则比较待插入数据和堆顶的大小

若堆顶更小，则将堆顶赋值为待插入数据，并对堆进行调整

数据输出

输出需要按照收入由大到小的顺序输出（根据样例数据推测）

因此实现一友元函数 `orderPrint(MinHeap<int> mp)` 实现堆数据的顺序输出

在 `mp` 非空时不断将其栈顶元素弹出，并存储在一数组中，然后将数组所有元素倒序输出即可得到由大到小的数据

> 为避免调用 `orderPrint` 时不断申请 `mp` 的内存空间，这里重载了 `MinHeap` 的拷贝构造函数，并用一全局数组作为拷贝构造的 `MinHeap` 对象的数据数组，具体实现可见**代码实现**部分

复杂度分析

对小根堆的维护的时间复杂度$O(nlog(k))$

一次数据输出的时间开销为$O(klog(k))$

因此假设最坏情况为进行了n次Check，则数据输出的总时间开销为$O(nklog(k))$

代码实现

小根堆实现

```
int *HeapData;
int *reverseHelper;
template<class T>
class MinHeap{
public:
    MinHeap(int size);
    MinHeap(const MinHeap<T>& mp){
        currentSize=mp.currentSize;
        maxHeapSize=mp.maxHeapSize;
        heap=(T*)HeapData;
        for(int i=0;i<currentSize;++i){
            heap[i]=mp.heap[i];
        }
    }
    MinHeap(T arr[],int n);
    bool insert(T ele);
    T pop();
    bool isFull();
    bool isEmpty();
    void print();
    int Size();
    T top();
    friend void orderPrint(MinHeap<int> mp){
        int i=0;
        while(!mp.isEmpty())
            reverseHelper[i++]=mp.pop();
        while(i-->0)
            cout<<reverseHelper[i]<<' ';
        cout<<endl;
```

```
30      }
31
32  private:
33      T *heap;
34      int currentSize;
35      int maxHeapSize;
36      void siftDown(int start,int end);
37      void siftUp(int start);
38  };
39
40  template<class T>
41  MinHeap<T>::MinHeap(int size){
42      maxHeapSize=size;
43      heap=new T[size];
44      currentSize=0;
45      maxHeapSize=size;
46  }
47
48  template<class T>
49  MinHeap<T>::MinHeap(T arr[],int n){
50      maxHeapSize=n;
51      heap=new T[maxHeapSize];
52      currentSize=n;
53      for(int i=0;i<n;i++){
54          heap[i]=arr[i];
55      }
56      int currentPos=(n-2)/2;
57      while(currentPos>=0){
58          siftDown(currentPos,currentSize-1);
59          currentPos--;
60      }
61  }
62
63  template<class T>
64  bool MinHeap<T>::isFull(){
65      return currentSize==maxHeapSize;
66  }
67
68  template<class T>
69  bool MinHeap<T>::isEmpty(){
70      return currentSize==0;
```

```
71  }
72
73  template<class T>
74  void MinHeap<T>::siftDown(int start,int end){
75      T tmp=heap[start];
76      int i=start;
77      int j=i*2+1;
78      while(j<=end){
79          if(j<end&&heap[j]>heap[j+1])j++;
80          if(tmp>heap[j]){
81             heap[i]=heap[j];
82             i=j;
83             j=i*2+1;
84          }
85          else{
86              break;
87          }
88      }
89      heap[i]=tmp;
90  }
91
92  template<class T>
93  void MinHeap<T>::siftUp(int start){
94      T tmp=heap[start];
95      int j=start,i=(j-1)/2;
96      while(j>0){
97          if(heap[i]>tmp){
98              heap[j]=heap[i];
99              j=i;
100             i=(j-1)/2;
101         }
102         else{
103             break;
104         }
105     }
106     heap[j]=tmp;
107 }
108
109 template<class T>
110 T MinHeap<T>::pop(){
111     T tmp=heap[0];
```

```
112     heap[0]=heap[currentSize-1];
113     siftDown(0,--currentSize-1);
114     return tmp;
115 }
116 
117 template<class T>
118 bool MinHeap<T>::insert(T ele){
119     if(this->isFull()){
120         if(heap[0]<ele){
121             heap[0]=ele;
122             siftDown(0,currentSize-1);
123             return true;
124         }
125         else return false;
126     }
127     heap[currentSize]=ele;
128     siftUp(currentSize++);
129     return true;
130 }
131 
132 template<class T>
133 void MinHeap<T>::print(){
134     for(int i=0;i<currentSize;i++){
135         cout<<heap[i]<<' ';
136     }
137     cout<<endl;
138 }
139 
140 template<class T>
141 int MinHeap<T>::Size(){
142     return currentSize;
143 }
144 
145 template<class T>
146 T MinHeap<T>::top(){
147     return heap[0];
148 }
```

主要过程

```
    while(scanf("%s",s)==1){
        if(s[0]=='C'){  //s=="Check"
            orderPrint(mp);
        }
        else{
            sscanf(s,"%d",&x);
            mp.insert(x);
        }
    }
```

任务四：查找蓝鲸市某市民

任务描述

> 给定所有蓝鲸市民的身份证号，共n个。市民与身份证号一一对应。身份证号总共有18位，由0~9的数字和大写字母组成。
>
> 再给一个市民的身份证号，需要判断这个市民是否属于蓝鲸市

复杂度要求

算法的存储空间尽量小，每次查询的平均时间复杂度尽量控制在$O(1)$

算法描述

使用 `Bloom Filter` 实现存储和查询：

存储实现

1. 每输入一个字符串 `str` （身份证号）
2. 使用k个哈希函数对字符串 `str` 哈希得到k个哈希值$\{x_1,x_2,\ldots\ldots,x_k\}$
3. 将m bit的二进制串 `bloom_filter` 的第x_i位置为1

查询实现

1. 输入一个待查询的字符串 `str`
2. 使用k个哈希函数对字符串 `str` 哈希得到k个哈希值$\{x_1,x_2,\ldots\ldots,x_k\}$
3. 在 `bloom_filter` 中判断第x_i位是否为1，如果出现第x_i位不为1，则认为 `str` 不在 `bloom_filter` 中，即该市民不属于蓝鲸市

Bloom Filter的参数

> 如何根据身份证号的个数n来确定
>
> - 哈希函数的个数k
> - 二进制串的长度m

假设我们允许的假阳性概率(FPR)$f = 0.5\%$

当集合S={x_1,x_2,…,x_n}的所有元素都被k个哈希函数映射到m位的位数组中时，这个位数组中某一位还是0的概率是

$$p = (1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$$

$(1-p)^k$表示k次哈希都刚好选中1的区域，即FPR：

$$f = (1 - e^{-kn/m})^k = (1 - p)^k$$

即

$$f = exp(kln(1 - e^{-kn/m}))$$

令

$$\begin{aligned} g &= kln(1 - e^{-kn/m}) \\ &= -\frac{m}{n} ln(p) ln(1 - p) \end{aligned}$$

所以当p=1/2时错误率最小，也就是让一半的位空着

$$k = ln2 * \frac{m}{n}$$

$$\begin{aligned} m &= n * log_2 e * log_2(1/f) \\ &\approx n * 1.44 * log_2(1/f) \end{aligned}$$

代入$f = 0.5\%$，得$m \approx 11n, k \approx 8$

哈希函数

在哈希类中，共实现了6种哈希函数JSHash、RSHash、BKDRHash、SDBMHash、DJBHash、DEKHash，其中BKDHash可以根据不同的`seed`得到不同的Hash值

复杂度分析

空间复杂度

`Bloom Filter`需要m bit的空间，由$m = 11n$可知，空间复杂度为$O(n)$

时间复杂度

每个哈希函数计算的开销为$O(1)$，共有$k = 8$个哈希函数

因此每次查询的时间复杂度为$O(1)$

而每次存储的时间复杂度也是$O(1)$

代码实现

哈希类的实现

```
class Hash;
typedef unsigned long (Hash::*HashFunc)(const string&);
class Hash {
private:
    unsigned long JSHash(const string& str) {
        unsigned long hash = 1315423911;
        for (int i = 0; i < str.length(); ++i) {
            hash ^= ((hash << 5) + str.at(i) + (hash >> 2));
        }
        return hash;
    }
    unsigned long RSHash(const string& str) {
        int b = 378551;
        int a = 63689;
        unsigned long hash = 0;
        for (int i = 0; i < str.length(); i++) {
            hash = hash * a + str.at(i);
            a = a * b;
        }
```

```
20          return hash;
21      }
22      unsigned long BKDRHash0(const string& str) {
23          unsigned long seed = 31;//31 131 1313 13131
    131313 1313131
24          unsigned long hash = 0;
25          for (int i = 0; i < str.length(); ++i) {
26              hash = (hash*seed) + str.at(i);
27          }
28          return hash;
29      }
30      unsigned long BKDRHash1(const string& str) {
31          unsigned long seed = 131;//31 131 1313 13131
    131313 1313131
32          unsigned long hash = 0;
33          for (int i = 0; i < str.length(); ++i) {
34              hash = (hash*seed) + str.at(i);
35          }
36          return hash;
37      }
38      unsigned long BKDRHash2(const string& str) {
39          unsigned long seed = 1313;//31 131 1313 13131
    131313 1313131
40          unsigned long hash = 0;
41          for (int i = 0; i < str.length(); ++i) {
42              hash = (hash*seed) + str.at(i);
43          }
44          return hash;
45      }
46      unsigned long BKDRHash3(const string& str) {
47          unsigned long seed = 13131;//31 131 1313 13131
    131313 1313131
48          unsigned long hash = 0;
49          for (int i = 0; i < str.length(); ++i) {
50              hash = (hash*seed) + str.at(i);
51          }
52          return hash;
53      }
54      unsigned long BKDRHash4(const string& str) {
55          unsigned long seed = 131313;//31 131 1313 13131
    131313 1313131
```

```
56          unsigned long hash = 0;
57          for (int i = 0; i < str.length(); ++i) {
58              hash = (hash*seed) + str.at(i);
59          }
60          return hash;
61      }
62      unsigned long BKDRHash5(const string& str) {
63          unsigned long seed = 1313131;//31 131 1313 13131
    131313 1313131
64          unsigned long hash = 0;
65          for (int i = 0; i < str.length(); ++i) {
66              hash = (hash*seed) + str.at(i);
67          }
68          return hash;
69      }
70      unsigned long BKDRHash6(const string& str) {
71          unsigned long seed = 13131313;//31 131 1313 13131
    131313 1313131
72          unsigned long hash = 0;
73          for (int i = 0; i < str.length(); ++i) {
74              hash = (hash*seed) + str.at(i);
75          }
76          return hash;
77      }
78      unsigned long SDBMHash(const string& str) {
79          unsigned long hash = 0;
80          for (int i = 0; i < str.length(); ++i) {
81              hash = str.at(i) + (hash << 6) + (hash << 16)
    - hash;
82          }
83          return hash;
84      }
85      unsigned long DJBHash(const string& str) {
86          unsigned long hash = 5381;
87          for (int i = 0; i < str.length(); ++i) {
88              hash = ((hash << 5) + hash) + str.at(i);
89          }
90          return hash;
91      }
92      unsigned long DEKHash(const string& str) {
93          unsigned long hash = str.length();
```

```
94          for (int i = 0; i < str.length(); ++i) {
95              hash = ((hash << 5) ^ (hash >> 27)) ^
    str.at(i);
96          }
97          return hash;
98      }
99      int k;
100     HashFunc *hashFunc;
101 public:
102     Hash(int k) {
103         this->k = k;
104         hashFunc = new HashFunc[12];
105         hashFunc[0] = &Hash::JSHash;
106         hashFunc[1] = &Hash::RSHash;
107         hashFunc[2] = &Hash::SDBMHash;
108         hashFunc[3] = &Hash::DJBHash;
109         hashFunc[4] = &Hash::DEKHash;
110         hashFunc[5] = &Hash::BKDRHash0;
111         hashFunc[6] = &Hash::BKDRHash1;
112         hashFunc[7] = &Hash::BKDRHash2;
113         hashFunc[8] = &Hash::BKDRHash3;
114         hashFunc[9] = &Hash::BKDRHash4;
115         hashFunc[10] = &Hash::BKDRHash5;
116         hashFunc[11] = &Hash::BKDRHash6;
117     }
118     int* HashNums(const string& str, int m) {
119         int *res = new int[k];
120         for (int i = 0; i < k; i++) {
121             res[i] = (this->*hashFunc[i])(str) %
    (unsigned)m;
122         }
123         return res;
124     }
125 };
126
```

Bloom Filter实现

```
1   class BloomFilter {
2   private:
3       int n, m;//n表示元素个数,m表示位串长度
4       int k;//哈希函数个数
```

```
5       Hash *hash;
6       int *bits;
7       void setBit(int x) {
8           int pos = x >> 5;   //x / 32
9           int addr = x - (pos << 5);//x % 32
10          int &cell = bits[pos];
11          cell |= 0x1 << addr;
12      }
13      bool getBit(int x) {
14          int pos = x >> 5;  //x / 32
15          int addr = x - (pos << 5);//x % 32
16          int &cell = bits[pos];
17          return cell & (0x1 << addr);
18      }
19  public:
20      BloomFilter(int n){
21          int size;
22          m = (double)n*1.44*7.643856;
23          size = (m >> 5) + 1;
24          k = log(2)*(double)m / (double)n;
25          this->n = n;
26          bits = new int[size];
27          for (int i = 0; i < size; i++)bits[i] = 0;
28          hash = new Hash(k);
29      }
30      void store(const string& s) {
31          int *hashNum = hash->HashNums(s, m);
32          for (int i = 0; i < k; ++i) {
33              setBit(hashNum[i]);
34          }
35          delete hashNum;
36      }
37
38      bool in(const string& s) {
39          int *hashNum = hash->HashNums(s, m);
40          for (int i = 0; i < k; i++) {
41              if (!getBit(hashNum[i]))
42                  return false;
43          }
44          return true;
45      }
```

```
46  };
```

存储过程

```
1   void inputHandle(BloomFilter *&bf) {
2       int n;
3       cin >> n;
4       bf = new BloomFilter(n);
5       string id;
6       for (int i = 0; i < n; i++)
7       {
8           cin >> id;
9           bf->store(id);
10      }
11  }
```

查询过程

```
1   void searchInFilter(BloomFilter &bf) {
2       string id;
3       while (cin >> id)
4           cout << (bf.in(id) ? "true" : "false") << endl;
5   }
```

主函数

```
1   int main()
2   {
3       BloomFilter *bf;
4       inputHandle(bf);
5       searchInFilter(*bf);
6       return 0;
7   }
```

任务五 计算收入的最大断档

任务描述

> 给定N个市民的年收入，要求计算工资最大断档，即对工资排序后相邻两数的差值的最大值。

复杂度要求

$O(N)$

算法描述

数据存储

> 采用桶的存储方式

1. 计算出年收入的最大值`max`和最小值`min`
2. 将区间`[min,max]`划分为大小为$size = \frac{max-min}{N-1}$的N个小区间
3. 将市民的年收入根据公式$pos = \frac{income-min}{size}$装入到区间第pos个区间中
    - 一个区间只存储该区间内的最大值和最小值
    - 在装入时对区间内的最大最小值更新即可完成装入

计算最大断档

根据公式

$$maxBreak = Max\{min_i - max_j\}$$

$0 <= j < i < N$, 且第j个区间是第i个区间的上一个有效区间

复杂度分析

- 计算最大值、最小值：$O(N)$
- 将年收入装入区间：$O(N)$
- 计算最大断档：$O(N)$

所以总复杂度也是$O(N)$

代码实现

桶的构建

```
inline int buildEles(int n,int *arr,Element *&eles){
    int min=0xffffffff,max=0;
    for(int i=0;i<n;++i){
        min=Min(min,arr[i]);
```

```
        max=Max(max,arr[i]);
    }
    int size=(max-min)/(n-1);
    size=size>0?size:1;
    int num=(max-min)/size+1;
    eles=new Element[num];

    int pos;
    for(int i=0;i<n;++i){
        pos=(arr[i]-min)/size;
        if(eles[pos].filled){
            eles[pos].min=Min(eles[pos].min,arr[i]);
            eles[pos].max=Max(eles[pos].max,arr[i]);
        }
        else{
            eles[pos].min=arr[i];
            eles[pos].max=arr[i];
        }
        eles[pos].filled=true;
    }
    return num;
}
```

最大断档的计算

```
int maxBreak(Element *eles,int n){
    int res=0;
    int lastMax=eles[0].max;
    for(int i=0;i<n;++i){
        if(eles[i].filled){
            res=Max(eles[i].min-lastMax,res);
            lastMax=eles[i].max;
        }
    }
    return res;
}
```