数据结构

课程设计报告

广州大学 计算机科学与网络工程学院
计算机系 17 级软件工程专业 1 班

谢金宏

(学号：1706300001)

(班内序号：02)

指导教师： 杜娇

2019 年 6 月 18 日

题目

随时找到数据流中的中位数

有一个源源不断地吐出整数的数据流，假设你有足够的空间来保存吐出的数。请设计一个名叫 *MedianHolder* 的结构，*MedianHolder* 可以随时取得之前吐出所有数的中位数。
要求：
1）如果 *MedianHolder* 已经保存了吐出的 N 个数，那么任意时刻将一个新数加入到 *MedianHolder* 的过程，其时间复杂度是O(logN)。
2）取得已经吐出的N个数整体的中位数的过程，时间复杂度为O(1)。

算法设计

寻找中位数是很常见的需求。朴素的做法是维护一个有序的序列 A，在新数加入时，需要为待插入的新数寻找到正确的位置，以维护数列有序的性质。查询中位数 *Median* 的操作：

$$\text{Median(A)} = \begin{cases} \frac{1}{2}(A_{\frac{len(A)}{2}} + A_{\frac{len(A)}{2}+1}), & \text{len(A)为偶数} \\ A_{\frac{len(A)+1}{2}}, & \text{len(A)为奇数} \end{cases}$$

在序列采用线性结构的前提下。若序列采用顺序储存，则加入新数时需要进行O(logN)二分查找，并使用O(N)时间使调整序列元素的位置；完成插入操作的时间复杂度为O(N)。易知取中位数操作的时间复杂度为O(1)。插入新数的时间复杂度不能满足题目要求。

若序列采用链式储存，则加入新数时寻找的合适的位置需要O(N)时间，但可以在O(1)时间内调整元素位置以维护序列有序性质。易知取中位数操作的时间为O(N)。取中位数的时间复杂度不满足题目要求。

经过上述分析可知，采用线性结构不能满足题目要求。下面考虑树状的结构。容易想到，使用二分搜索树，并在树的结点中维护子孙结点的个数可以确保在O(logN)内完成插入操作，但寻找中位数的时间提升至O(logN)。不是一种可行的解决方案。

回顾题目的要求，注意到题目只是要求找出中位数，而前述算法中很大一部分时间花费在维护“序列有序”上。实际上并不需要整个序列有序，而只是需要确保找出序列中的一个数（或两个数），使这个数不小于序列中一半的数字，不大于序列中一半的数字。这说明使用待定的数据结构内部元素可以不是有序的。根据这个特点，容易想到使用堆这种数据结构。

具体的算法是，维护一个大顶堆和小顶堆，使得它们满足下列性质：

1. 大顶堆的堆顶不大于小顶堆的堆顶。
2. 大顶堆的元素数量最多比小顶堆的元素数量多 1。

在插入操作时，若新插入数字比小顶堆堆顶小，则压入大顶堆，否则，从小顶堆中取出堆顶压入大顶堆，并将新的数字压入小顶堆。注意在插入操作后要维护前面提到的堆的性质。

如果将序列中的数字标记在数轴上，则大顶堆实际上包含了数轴上序列中位数左边的所有数字，小顶堆则包含了序列中位数右边的所有数字。当序列中元素的个数为奇数个时，大顶堆堆顶是序列的中位数；元素个数为偶数个时，大顶堆和小顶堆的两个堆顶的算术平均数

就是序列的中位数。这样，对于取中位数的操作：

$$\text{Median(A)} = \begin{cases} \frac{1}{2}(\text{top(MaxHeap)} + \text{top(MinHeap)}), & \text{size(MaxHeap)} + \text{size(MinHeap)为偶数} \\ \text{top(MaxHeap)}, & \text{size(MaxHeap)} + \text{size(MinHeap)为奇数} \end{cases}$$

数据结构

使用的主要的数据结构为堆。堆中元素采用顺序储存方案。下面，Heap.hpp 是堆的源代码，HeapTest.cpp 是堆的测试代码：


Heap.hpp


HeapTest.cpp

```
// Heap.hpp
// 最大堆和最小堆的实现。

#include <functional>
#include <cassert>

// 用于生成大顶堆的比较函数。
const std::function<bool(double, double)> MAXIMUMHEAP_CMP = [](double
lhs, double rhs) {
    return lhs > rhs;
};

// 用于生成小顶堆的比较函数。
const std::function<bool(double, double)> MINIMUMHEAP_CMP = [](double
lhs, double rhs) {
    return lhs < rhs;
};

// 堆
// 自动维护堆的性质的数据结构。
class Heap {
private:
    double *heap;     // 指向保存堆中元素的数组
    int capacity;     // 保存堆中元素的数组的容量
    int size;         // 堆中元素的个数

    // 用于比较堆中元素大小的函数，重载此函数可以方便地生成大顶堆和小顶堆。
    std::function<bool(double , double)> compare;
```

```
private:
    // 交换索引位置 idx1 和 idx2 的元素值。
    void swapNodeAt(int idx1, int idx2) {
        double tmp = heap[idx1];
        heap[idx1] = heap[idx2];
        heap[idx2] = tmp;
    }

    // 返回 idx 处的结点是否为空。
    bool isEmpty(int idx) {
        return idx <= 0 || idx > size;
    }

    // 返回 idx 处的结点是否为根节点。
    bool isRoot(int idx) {
        return idx == 1;
    }

    // 获取父亲结点的索引。
    int getParentIndex(int idx) {
        return idx / 2;
    }

    // 获取结点的左子结点的索引。
    int getLeftChildIndex(int idx) {
        return idx * 2;
    }

    // 获取结点的右子结点的索引。
    int getRightChildIndex(int idx) {
        return idx * 2 + 1;
    }

    // 使 idx 处的结点上浮。
    void shiftUp(int idx) {
        if (isRoot(idx)) {  // 如果当前结点已经是根节点，则不再上浮。
            return;
        }
        int parent_idx = getParentIndex(idx);
        if (compare(heap[idx], heap[parent_idx])) {  // 如果比较函数返回积
极结果。
            swapNodeAt(idx, parent_idx);  // 将当前结点上浮到父亲结点。
            shiftUp(parent_idx);  // 递归上浮。
        }
```

```
    }

    // 使 idx 处的结点下沉。
    void shiftDown(int idx) {
        int left_child_idx = getLeftChildIndex(idx);
        int right_child_idx = getRightChildIndex(idx);
        if (isEmpty(left_child_idx)) {
            return;  // 左子树为空则不再下沉。
        }
        // 两个子树中更优先与 idx 结点比较的结点。在大顶堆中优先结点为左右结点
中较大的一个。
        int prior_child_idx;
        if (isEmpty(right_child_idx)) {
            prior_child_idx = left_child_idx;  // 若右子节点为空，则当前结
点只能与左子节点进行比较。
        } else {
            prior_child_idx = (compare(heap[left_child_idx],
heap[right_child_idx]) ?
                left_child_idx
                : right_child_idx);  // 当前结点应该与左右结点中较大的一个进
行比较。
        }
        if (!compare(heap[idx], heap[prior_child_idx])) {
            swapNodeAt(idx, prior_child_idx);  // 将 idx 结点下沉到
prior_child_idx 结点位置。
            shiftDown(prior_child_idx);  // 递归下沉。
        }
    }

    // 必要时将堆的容量扩大。
    void resize() {
        if (size + 8 > capacity) {
            int new_capacity = capacity * 2;
            double *new_heap = new double[new_capacity];
            memcpy(new_heap + 1, heap + 1, sizeof(double) * size);

            delete[] heap;
            heap = new_heap;
            capacity = new_capacity;
        }
    }


public:
    // 堆的构造函数
```

```
// 需要传入一个比较函数，可以方便地根据比较函数的不同生成大顶堆或小顶堆。
Heap(std::function<bool(double , double)> cmp) {
    capacity = 1024;
    heap = new double[capacity];
    size = 0;
    compare = cmp;
}

// 堆的复制构造函数
Heap(const Heap& oth) {
    capacity = oth.capacity;
    heap = new double[capacity];
    size = oth.size;
    memcpy(heap + 1, oth.heap + 1, sizeof(double) * size);
    compare = oth.compare;
}

~Heap() {
    delete[] heap;
}

// 向堆中插入值为 val 的结点。
void Push(double val) {
    heap[++size] = val;  // 将新加入的结点放置在堆尾。
    shiftUp(size);       // 将新加入的结点上浮。
    resize();            // 必要时将堆的容量扩大。
}

// 返回堆中元素的个数。
int Size() {
    return size;
}

// 清空堆。
void Clear() {
    size = 0;
}

// 返回堆顶的元素。
double Top() {
    assert(size >= 1);  // 求堆顶操作需要堆中至少存在一个元素。
    return heap[1];
}
```

```
    // 将堆顶出堆。
    void Pop() {
        assert(size >= 1);  // 出堆操作要求堆中至少存在一个元素。
        swapNodeAt(1, size);
        --size;
        if (size > 0) shiftDown(1);  // 使新的堆顶下沉。
    }
};

// 返回一个大顶堆。
Heap GetMaximumHeap() {
    return Heap(MAXIMUMHEAP_CMP);
}

// 返回一个小顶堆。
Heap GetMinimumHeap() {
    return Heap(MINIMUMHEAP_CMP);
}
```

源程序

MedianHolder.cpp 为前述算法的实现。

MedianHolder.cpp

```
#include "Heap.hpp"

#include <iostream>
#include <sstream>

using namespace std;


// 中位数寻找器
// 能在对数时间存入数据，并在常数时间内取出所存数据的中位数。
class MedianHodler {
private:
    Heap max_heap = GetMaximumHeap();
    Heap min_heap = GetMinimumHeap();
public:
    // 清空已保存的数字。
    void Clear() {
```

```
        max_heap.Clear();
        min_heap.Clear();
    }

    // 压入新的数字。
    void Push(double number) {
        if (max_heap.Size() == 0 || min_heap.Size() == 0) {
            max_heap.Push(number);
        } else {
            if (number > min_heap.Top()) {
                double top = min_heap.Top();
                min_heap.Pop();
                max_heap.Push(top);
                min_heap.Push(number);
            } else {
                max_heap.Push(number);
            }
        }

        // 确保左边的大顶堆元素数量始终不大于右边的小顶堆的元素数量加 1，
        // 也就是左边的大顶堆最多比右边的小顶堆多一个元素。
        while (max_heap.Size() > min_heap.Size() + 1) {
            double top = max_heap.Top();
            max_heap.Pop();
            min_heap.Push(top);
        }
    }

    // 取中位数。
    double GetMedian() {
        assert(max_heap.Size() + min_heap.Size() >= 1);
        if ((max_heap.Size() + min_heap.Size()) % 2 == 0) {
            // 序列长度为偶数的情况下，取中间两位数字的算术平均数作为中位数。
            return double(max_heap.Top() + min_heap.Top()) / 2;
        }
        // 序列长度为奇数的情况下，取中间一位数字（大顶堆的堆顶）作为中位数。
        return max_heap.Top();
    }
};

int main()
{
    MedianHodler holder;
```

```
    cout << "请在一行之内输入待寻找中位数的整数序列，按 Ctrl+Z 结束输入。" <<
endl;
    string line;
    while (getline(cin, line)) {
        holder.Clear();

        stringstream ss(line);

        double input;
        while (ss >> input) {
            holder.Push(input);
        }
        cout << "中位数：" << holder.GetMedian() << endl;
    }
}
```

测试数据和结果

小数据

每输入一行数字，执行一次寻找中位数的操作：

输入序列	输出结果
1 2 3 4 5 6 7 8	4.5
8 7 6 5 4 3 2 1	4.5
1 3 5 2 9 8 6 4	4.5

以上测试结果符合预期。

大数据

算法通过了 Open Judge #2388 上的大数据测试。

2388: 寻找中位数最近的提交

提交人	班级	结果	内存	时间	代码长度	语言	提交时间
lightyears		Accepted	408kB	15ms	6662 B	G++	23小时前

基本信息
#: 19421279
题目: 2388
提交人: lightyears
内存: 408kB
时间: 15ms
语言: G++
提交时间: 2019-06-18 09:33:17

结束语

在初次看到这份课程设计的题目时，我误以为只需要维护一个序列并使用二分搜索就能

解决问题。但没想到题目“暗藏玄机”，经过思考后我发现单纯地使用线性的数据结构不能满足题目所要求的时间复杂度。经过与同学的讨论交流，以及上网搜集资料，后我发现原来这个需求可以通过使用堆结构解决来轻松地解决。原来问题的解决方案如此巧妙，这令我大吃一惊。这里还要特别感谢在实验课程中指导我的杜娇老师，她考虑到的一些问题是原本我没有考虑到的。

经过本次数据结构课程设计，我认为自己的思维能力得到了提升，动手能力得到了锻炼。