assign 3

求是科学班（计算机） 朱淞伯

3160103828

问题概述

> *一个文本(recorder.txt)记录了大量商品价格，商品价格范围0.00~999999.99元，试找出第n高的价格值，n=自己的学号后4位+10000 (3160103828)。*
>
> *Reader.txt中，每个价格后跟一个空格，价格没有重复，数目可能超过1000万个*
>
> *选做部分：假设数据规模是reader.txt的10倍，应如何做？给出思路，进一步可以编写程序实现*

设计思路

先考虑总体框架，这里我利用了oop的思想，首先 `class Prices` ，构造函数为从文件读取操作，拥有写出的单独函数，同时有单独的排序函数。private的内容为number（排序数字数量），以及一个double数组

首先关于读入写入，供参考的文件读写代码已经够用，加入略微修改

其次，由于题目要求是输出第10000+学号后四位高的价格，这里就有两种思路，其一是直接排序，然后直接读排序好的数组中的值；其二是直接从0开始遍历，找出从0到13828的最高值。我们知道，排序的时间复杂度是$O(nlg(n))$，而如果用第二种，时间复杂度大概为$O(10000*n)$，考虑到样例输入的n大概为800万，这里用第一种方法应该能快一点。

关于快速排序，和课上介绍的内容大致相同，由于以前上过计算机学院开设的数据结构与算法基础课程，加入了pivot的优化，从数组中的第一个，中间，以及最后一个元素，找到中间值作为确定位置的值，这样可以更好的保证快排$O(nlg(n))$的时间复杂度。同时当数据个数小于16时，可以直接使用插入排序

具体实现

首先对于class的定义如下：

```
class Prices
```

```
{
private:
    int number;     // the total number
    double *prices;  // the array to save the data
public:
    Prices();
    int getnumber();   // const
    void qsortP(int left, int right);    // quick sort
    double Median(double *a, int left, int right);    // get the pivot
    void swap(double *a, double *b);    // change two number
    void insertsortP(double *a, int x);  // insertion sort
    void output_id(int student_id = 3828);  // output the required number
    void outputP();   // for test
};
```

其次，对于文件读写，读入的代码：

```
Prices::Prices()
{
    this->prices = new double[MAX_ELEMENTS];   // malloc a place
    ifstream reader;
    reader.open("recorder.txt", ios::in);   // open the file
    int i = 0;
    double x;
    reader >> x;
    this->prices[i++] = x;
    reader >> x;
    this->prices[i++] = x;   // get the first two data

    while (prices[i-1] != prices[i-2]){    // 由于没有相同的数，所以直接这样判断结束
        reader >> x;
        prices[i++] = x;
        if (i % 1000000 == 0)
            printf("getting the %d's number.\n", i);   // 分段输出进程信息
    }
    i -= 1;
    this->number = i;
    if (i == MAX_ELEMENTS)    // 如果达到了最大值，需要修改程序MAX_ELEMENT宏
    {
        printf("Please make sure the number of unsorted data is less than %d\n", MAX_ELEMENTS);
        printf("Or you can change the code line 5, adjust the MAX_ELEMENTS MARCO.\n");
        exit(1);
```

```
    }
    printf("The total number of unsorted data is %d.\n", i);  // 输出一共读入个数
    printf("Now begin to sort: \n");

    reader.close();
}
```

然后关于快排序的代码，参考了算法导论的部分代码，修改为了OOP的形式，首先是寻找pivot的代码：

```
double Prices::Median(double *a, int left, int right)
{
    int center = (left + right) / 2;
    if (a[left] > a[center])
        swap(&a[left], &a[center]);
    if (a[left] > a[right])
        swap(&a[left], &a[right]);
    if (a[center] > a[right])
        swap(&a[center], &a[right]);   // 首先对于三个数进行排序

    swap(&a[center], &a[right-1]);    // 将中间值藏入最后
    return a[right-1];            // 返回中间值
}
```

这里有一个细节，就是讲center的中间值藏入right-1的位置，由于right的位置的值一定已经是三个数中最大的，所以center的值一定小于right内的值，所以藏进right-1的位置，回头找准pivot应该放入的位置，直接swap一下即可，然后就是快速排序的主逻辑：

```
void Prices::qsortP(int left, int right)
{
    int i, j;
    double pivot;
    if (left + 16 < right)
    {
        pivot = Median(prices, left, right);
        i = left;
        j = right - 1;
        while(1)
        {
            while(prices[++i] < pivot);   // 从左至右
            while(prices[--j] > pivot);   // 从右至左
            if (i < j)
                swap(&prices[i], &prices[j]);  // 交换两个元素
```

```
        else
            break;
    }
    swap(&prices[i], &prices[right-1]);   // 交换原来的pivot
    qsortP(left, i - 1);
    qsortP(i + 1, right);   // 递归排序两个子序列
  }
  else
    insertsortP(prices + left, right - left + 1);
}
```

也是根据算法导论的标准代码修改而成，可以看到，这里我们i从左向右，j从右至左，直到发现一对不和谐的数据，交换之后继续，继续下一次迭代。当扫完一遍之后，将藏在 `prices[right-1]` 的pivot交换，得到最终的序列。当序列小于16时，直接使用插入排序，代码如下：

```
void Prices::insertsortP(double *a, int x)
{
  int min_num;
  for (int i = 0; i < x; i++)
  {
    min_num = i;
    for (int j = i; j < x; j++)
    {
      if (a[j] < a[min_num]){
        min_num = j;      // 交换
      }
    }
    swap(&a[i], &a[min_num]);
  }
}
```

在中间，还有一段测试代码，在函数 `void Prices::outputP()` 中，直接输出来查看是否符合从小到大的规律，可以直接调用。

```
void Prices::outputP()
{
  for (int i = 5333330; i <= 5333433; i++)
  {
    printf("%lf\n", this->prices[i]);    // 遍历输出
  }
}
```

测试

这里我尝试测试了一下当前的效率，在调用之前记录时间，调用之后记录时间，计算一下排序的用时，代码如下：

```
#define TIME_TEST_FOR(TEST_TIMES, FUN_CALL) do \
{\
    clock_t start, stop;\
    start = clock();\
    for (int i = 0; i < TEST_TIMES; i++)\
    {\
        FUN_CALL;\
    }\
    stop = clock();\
    printf("Sorting cost %0.8lf(s)\n", TEST_TIMES , stop-start, (double)(stop-start)/(CLK_TCK*10000));\
}\
while (0)
```

这样直接在main函数中调用 `TIME_TEST_FOR(1, Price.qsortP(0, Price.getnumber()));` 即可获得排序所用时间，这里我们用时大约为：

```
The total number of unsorted data is 8333333.
Now begin to sort:
Sorting cost 1.55780300(s)
The 13828's highest price is: 994495.300000.
```

1.5s左右，看起来800w的数据量，还可以接受。同时可以看出，**问题的答案为994495.30**

之后我尝试了一下另一个版本，在最开始也有提及，即直接计算13828次最大值，代码很好写，类似于插入排序，如下：

```
void Prices::get_id_high(int number)
{
    int i, j, max;
    for (i = 0; i < number; i++)
    {
        max = i;
        for (j = i; j < this->number; j++)
        {
            if (prices[j] > prices[max])
                max = j;
        }
        swap(&prices[i], &prices[max]);
    }
```

```
    printf("After traversal based on query method, the %d'th highest number is %lf.\n", number, prices[i-1]);
}
```

同样的，在main函数直接宏展开调用即可

```
Prices Price1;
TIME_TEST_FOR(1, Price1.get_id_high(13828));
```

然后我跑了一下，等了1分钟左右，电脑风扇转的飞快，就是不显示结果，还以为自己写出了死循环，debug之后感觉有没有可能时间太长了跑不出来，于是把最大值调整成为了1000，这次经过了10秒之后出了结果

这里我把前面的排序输出也改为了输出第1000大的数，结果：

```
/Users/zuhxs/CLionProjects/opt2/cmake-build-debug/opt2
The total number of unsorted data is 8333333.
Now begin to sort:
Sorting cost 1.41335600(s)
The 1000's highest price is: 996026.100000.
The total number of unsorted data is 8333333.
Now begin to sort:
After traversal based on query method, the 1000'th highest number is 996026.100000.
Sorting cost 19.21490200(s)
```

可以看出输出了两个时间结果，第一个是我们先排序的结果，用时**1.4133*s***，而第二种方法在输出第1000大的数字的时候就已经需要**20*s***左右，如果是13828，可以预见大概需要**200*s***左右，显然不可取。当然从这里可以看出，无论是哪个方法，最后的结果都是996026.10，从侧面印证了算法的正确性

综上，在该数据范围下，使用快速排序排序的结果大概可以将结果压缩至1.4s左右，属于可以接受的范围

选做部分

当数据量大于一亿时，按每个double用8个byte来计算，大概需要8G左右的内存，显然全部读入内存再快速排序不太现实，这里我们可以采用另一种方法。

设计思路

其实这里最大的限制就是不能拿进来排序了，而我们根据上面第二种方法的结果可知，肯定不可能在文件中遍历，毕竟读取文件也十分费工夫，所以我们的思路还是，能不能依旧只读一遍文件，分着存储，或者说，只存储一部分？

这就自然而然的引出了我们第一个思路，把整个文件按照1000万个数据点为一组分组，每读取1000万个数据点，得到13282个最大值，然后将所有的结果用merge sort的思路合并，但显然当数据量再大的时候，这个方法就不适用了，因为需要归并的数据量也会增长。

之后我就想到了第二种思路，利用的是堆排序的做法。我们都知道，堆排序是一种基于最小堆的排序，所以其实这里我们只需要维护一个大小为13828的最小堆即可。总的来看，就是先读取前13828个数据，先建立一个最小堆，之后迭代。读取一个数据，将其与最小堆堆顶的值进行比较，如果小于，说明序列中有13828个数大于该值，于是该值被直接跳过。而如果大于该值，则直接弹出最小堆堆顶的值，将该值放入最小堆中，重新排序。于是，总的来看，这个的时间复杂度就是$O(log_2 13828 * N)$的时间复杂度，远远小于之前的$O(13828 * N)$。看似可行：

具体实现

这里的具体实现还是参考了算法导论中的内容：首先我们需要一个insert的操作，将一个数插入到最小堆中（最小堆实际上就是一个数组），代码如下：

```
void insert(double x, double *result, int size)
{
    int i;
    for (i = ++size; result[i/2] > x; i /= 2)
        result[i] = result[i/2];   // 从底向上找合适的节点
    result[i] = x;
}
```

实际上就是自底而上找到一个合适的位置，进行交换，把插入的数放到最小堆内

之后是deleteMin操作，用于删除最小堆上面的元素，首先是直接扔掉最上面的元素，然后把最后的元素放到上面，再进行调整，代码：

```
double deleteMin(double *result)
{
    int i, child;
    double minE, lastE;
    minE = result[1];    // 头结点
    lastE = result[ID];   // 尾节点
    for (i = 1; i * 2 <= ID-1; i = child)
    {
        child = i * 2;    // 首先child是i*2
        if (child != ID - 1 && result[child+1] < result[child])
            child++;      // 找到最小的child
        if (lastE > result[child])
            result[i] = result[child];   // 如果合适，进行交换
        else break;
    }
    result[i] = lastE;    // 把最后一个node放到合适的位置去
    return minE;
}
```

需要注意的是，我们的 `double *result` 就是用来存储我们最小堆的结果， `result[0]` 是一个哨兵节点，不起任何作用

有了这两个操作后，最小堆的生成代码就非常简单了，如下：

```
double *init_heap()
{
    double *result = new double[ID+1];

    ifstream reader;
    reader.open("recorder.txt", ios::in);
    int i = 0;
    double x;
    for (int i = 0; i < ID; i++)
    {
        reader >> x;
        insert(x, result, i);
    }    // 首先遍历ID个节点，粗略的建一个最小堆

    while (!reader.eof())    // 当最后一个元素之前
    {
        reader >> x;      // 读入一个数据
        if (x > result[1])    // 如果数据大于当前堆中的最小值，插入改数据
        {
            double y = deleteMin(result);   // 首先删除最小值
            insert(x, result, ID-1);       // 插入x
        }
    }
    printf("The %d's highest price is: %lf.\n", ID, result[1]);   // 排序完成，直接输出
}
```

在最开始宏定义 `#define ID 13828` ，之后直接用ID即可

测试

其实首先我想的是先测试原先800万的数据，结果发现居然很快，其实可以横向对比一下，在最开始的快排算法，我们的测量都是不算文件读取的操作的。我重新测量了一下，那里的文件读取需要花费3.45826600(s)的时间，加上快排序的1.5(s)，一共需要大概5(s)左右的样子，而我们利用这个方法之后，测试结果：

```
#define TIME_TEST_FOR(TEST_TIMES, FUN_CALL) do \
{\
    clock_t start, stop;\
    start = clock();\
    for (int i = 0; i < TEST_TIMES; i++)\
    {\
        FUN_CALL;\
    }\
    stop = clock();\
    printf("getting the number cost %0.8lf(s)\n", TEST_TIMES , stop-start,
(double)(stop-start)/(CLK_TCK*10000));\
}\
while (0)
TIME_TEST_FOR(1, init_heap());
```

```
/Users/zuhxs/CLionProjects/opt3/cmake-build-debug/opt3
994495.300000
getting the number cost 2.98333300(s)
```

居然只需要3(s)左右的时间，减少了将近一半，看到这个结果还是很惊喜的，毕竟如果没有进一步的数据量扩大的思考，也不会有这样的结果。

生成器

如果想测试上亿的数据，首先我们要有一个生成器，根据题目要求，生成的数据要在0-999999.99之间，首先想到了自带的 `rand()` 函数，但 `rand()` 函数是满足正态函数的生成器，而我们想要的是随机生成的满足平均分布的数据，本来想模拟一下内部实现，但后来觉得有点麻烦，直接 `rand()*rand()` 基本就可以满足需求了，之后，由于 `rand()*rand()` 生成的是一个long int，我们可以先取模，取 `(rand() * rand()) % 100000000` ，然后除以100，就可以生成出满足0-999999.99的数据了。

之后关于IO的问题，流输出是有格式化输出的，比如我们最后生成的是y，我们在输出的时候只要 `writer << setprecision(8) << y;` 就可以保证输出的是两位小数了，当然要在最开始引入 `iomanip` ，include进来。

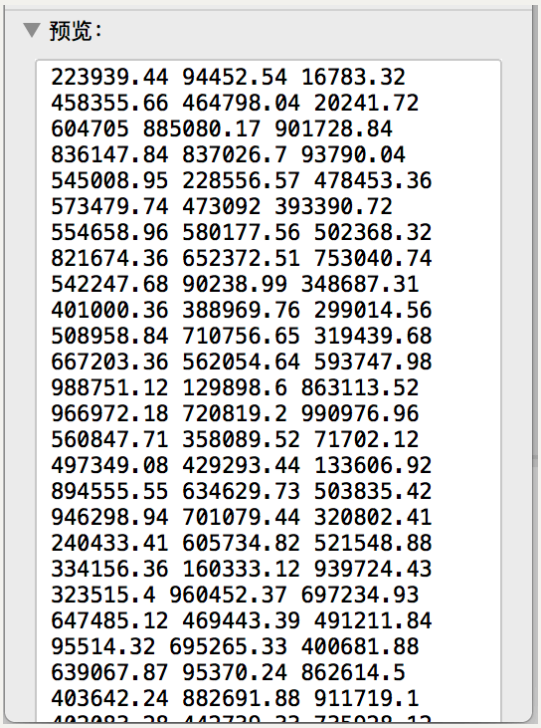
由于代码不长，这里直接贴一下源码，源码如下：

```
#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

int main()
```

```
{
    ofstream writer;
    writer.open("recorder.txt", ios::trunc);
    srand(time(0));
    unsigned int x = rand() * rand();
    double y = (x % 100000000) / 100.0;
    printf("%d, %lf", x, y);
    writer << setprecision(8) << y;
    writer << " ";


    for (int i = 0; i < 200000000; i++)
    {
        x = rand() * rand();
        y = (x % 100000000) / 100.0;
        writer << setprecision(8) << y;
        writer << " ";
    }
}
```

生成的结果，由于生成了2亿的数据，这里的文件大小有1.4G，如下：

然后大概看一下数据，



预览：

```
223939.44 94452.54 16783.32
458355.66 464798.04 20241.72
604705 885080.17 901728.84
836147.84 837026.7 93790.04
545008.95 228556.57 478453.36
573479.74 473092 393390.72
554658.96 580177.56 502368.32
821674.36 652372.51 753040.74
542247.68 90238.99 348687.31
401000.36 388969.76 299014.56
508958.84 710756.65 319439.68
667203.36 562054.64 593747.98
988751.12 129898.6 863113.52
966972.18 720819.2 990976.96
560847.71 358089.52 71702.12
497349.08 429293.44 133606.92
894555.55 634629.73 503835.42
946298.94 701079.44 320802.41
240433.41 605734.82 521548.88
334156.36 160333.12 939724.43
323515.4 960452.37 697234.93
647485.12 469443.39 491211.84
95514.32 695265.33 400681.88
639067.87 95370.24 862614.5
403642.24 882691.88 911719.1
```

可以看出这里的数据还是基本符合要求的，只是这里无法满足没有重复的要求了，但从0-999999.99最多也只有一亿的数据，而且不重复也确实在实现的时候没有什么意义，这里就不考虑这点了

最终测试

于是把这个生成的数据放进原工程文件夹进行测试：

```
/Users/zuhxs/CLionProjects/opt3/cmake-build-debug/opt3
999928.090000
getting the number cost 71.12932400(s)
```

花了71秒左右，毕竟时间复杂度一直是$O(log(13828)*N)$，在13828常数不变的情况下，时间复杂度随着N的变化是线性的，当时800万的数据量是3秒，这里2亿数据量正好是20倍左右，也就是60-70秒，非常合理

小结

这次作业陆陆续续做了两天，一是ddl很近，作业量确实挺大，二来这件事确实非常有意思，看到2亿的数据量在自己的优化之后缩短到70秒左右，非常有成就感。非常感谢老师的这次开放性作业，看似要求不明确，但自己做完回想的时候，要求还是很清晰的，看似模糊的地方都是需要我们自己经过尝试找到最优解的。就比如最开始的任务，我一直认为直接扫描出13828个最大值是线性时间复杂度，应该很快，但自己操作之后才发现真的很慢，相比而言，我认为很慢的快排反而有非常出色的表现。

然后就是之后的选做作业，做完之后同样很有收获，为了解决数据量大的问题想出的堆排序居然用时更短，如果没有进一步的思考，也不会有这样的结果。最后总而言之，这次作业真的很有意思，感谢老师。在最后贴一下所有的源代码。

源代码

快排版本：

```
#include <iostream>
#include <fstream>
using namespace std;

#define MAX_ELEMENTS 50000000

#define TIME_TEST_FOR(TEST_TIMES, FUN_CALL) do \
{\
   clock_t start, stop;\
   start = clock();\
   for (int i = 0; i < TEST_TIMES; i++)\
   {\
      FUN_CALL;\
   }\
   stop = clock();\
   printf("Sorting cost %0.8lf(s)\n", TEST_TIMES , stop-start, (double)(stop-start)/(CLK_TCK*10000));\
}\
while (0)

//template <class T>
class Prices
{
private:
   int number;     // the total number
   double *prices;  // the array to save the data
public:
   Prices();
   int getnumber();   // const
   void qsortP(int left, int right);    // quick sort
   double Median(double *a, int left, int right);    // get the pivot
   void swap(double *a, double *b);    // change two number
   void insertsortP(double *a, int x);  // insertion sort
   void output_id(int student_id = 3828);  // output the required number
   void outputP();   // for test
   void output_file();
   void get_id_high(int number);
```

```
};


int Prices::getnumber()
{
   return this->number;
}

void Prices::output_id(int student_id)
{
   printf("The %d's highest price is: %lf.\n", student_id + 10000, this-
>prices[this->number - (student_id + 10000) + 1]);
   //printf("The %d's highest price is: %lf.\n", student_id, this->prices[this-
>number - (student_id) + 1]);
}

//template <class T>
Prices::Prices()
{
   this->prices = new double[MAX_ELEMENTS];
   ifstream reader;
   reader.open("recorder.txt", ios::in);
   int i = 0;
   double x;
   reader >> x;
   this->prices[i++] = x;
   reader >> x;
   this->prices[i++] = x;

   while (prices[i-1] != prices[i-2]){
      reader >> x;
      prices[i++] = x;
      //printf("%lf", prices[i-1]);
      //i++;
      //if (i % 1000000 == 0)
         //printf("getting the %d's number.\n", i);
   }
   i -= 1;
   this->number = i;
   if (i == MAX_ELEMENTS)
   {
      printf("Please make sure the number of unsorted data is less than
%d\n", MAX_ELEMENTS);
      printf("Or you can change the code line 5, adjust the MAX_ELEMENTS
MARCO.\n");
      exit(1);
   }
```

```
    printf("The total number of unsorted data is %d.\n", i);
    printf("Now begin to sort: \n");

    reader.close();
}

void Prices::swap(double *a, double *b)
{
    double temp = *a;
    *a = *b;
    *b = temp;
}

//void Prices::qsortP(double *a, int left, int right)
void Prices::qsortP(int left, int right)
{
    int i, j;
    double pivot;
    if (left + 16 < right)
    {
        pivot = Median(prices, left, right);
        i = left;
        j = right - 1;
        while(1)
        {
            while(prices[++i] < pivot);
            while(prices[--j] > pivot);
            if (i < j)
                swap(&prices[i], &prices[j]);
            else
                break;
        }
        swap(&prices[i], &prices[right-1]);
        qsortP(left, i - 1);
        qsortP(i + 1, right);
    }
    else
        insertsortP(prices + left, right - left + 1);
}

double Prices::Median(double *a, int left, int right)
{
    int center = (left + right) / 2;
    if (a[left] > a[center])
        swap(&a[left], &a[center]);
    if (a[left] > a[right])
        swap(&a[left], &a[right]);
```

```
    if (a[center] > a[right])
        swap(&a[center], &a[right]);

    swap(&a[center], &a[right-1]);
    return a[right-1];
}

void Prices::insertsortP(double *a, int x)
{
    int min_num;
    for (int i = 0; i < x; i++)
    {
        min_num = i;
        for (int j = i; j < x; j++)
        {
            if (a[j] < a[min_num]){
                min_num = j;
            }
        }
        swap(&a[i], &a[min_num]);
    }
}


void Prices::outputP()
{
    for (int i = 8333330; i <= 8333333; i++)
    {
        printf("%lf\n", this->prices[i]);
    }
}


void Prices::output_file()
{

}

void Prices::get_id_high(int number)
{
    int i, j, max;
    for (i = 0; i < number; i++)
    {
        max = i;
        for (j = i; j < this->number; j++)
        {
            if (prices[j] > prices[max])
```

```
            max = j;
        }
        swap(&prices[i], &prices[max]);
        //printf("%lf\n", prices[i]);
    }
    printf("After traversal based on query method, the %d'th highest number
is %lf.\n", number, prices[i-1]);
}


int main()
{
    //Prices<double> Price;
    Prices Price;
    TIME_TEST_FOR(1, Price.qsortP(0, Price.getnumber()));
    //Price.qsortP(0, Price.getnumber());
    Price.output_id(3828);
    //Prices Price1;
    //TIME_TEST_FOR(1, Price1.get_id_high(1000));
    //Price1.outputP();
}
```

堆排序版本:

```
#include <iostream>
#include <fstream>
using namespace std;
#define ID 13828

#define TIME_TEST_FOR(TEST_TIMES, FUN_CALL) do \
{\
    clock_t start, stop;\
    start = clock();\
    for (int i = 0; i < TEST_TIMES; i++)\
    {\
        FUN_CALL;\
    }\
    stop = clock();\
    printf("getting the number cost %0.8lf(s)\n", TEST_TIMES , stop-start,
(double)(stop-start)/(CLK_TCK*10000));\
}\
while (0)

double *init_heap();
void insert(double x, double *result, int size);
double deleteMin(double *result);
```

```
int main()
{
    TIME_TEST_FOR(1, init_heap());
    //double *result = init_heap();
}


double *init_heap()
{
    double *result = new double[ID+1];

    ifstream reader;
    reader.open("recorder1.txt", ios::in);
    int i = 0;
    double x;
    for (int i = 0; i < ID; i++)
    {
        reader >> x;
        insert(x, result, i);
    }

    while (!reader.eof())
    {
        reader >> x;
        if (x > result[1])
        {
            double y = deleteMin(result);
            insert(x, result, ID-1);
        }
    }

    printf("%lf\n", result[1]);
}

void insert(double x, double *result, int size)
{
    int i;
    for (i = ++size; result[i/2] > x; i /= 2)
        result[i] = result[i/2];     // from bottom to the top, find the optimized
place
    result[i] = x;
}

double deleteMin(double *result)
{
    int i, child;
```

```
    double minE, lastE;
    minE = result[1];   // the first element
    lastE = result[ID];   // the last element
    for (i = 1; i * 2 <= ID-1; i = child)
    {
        child = i * 2;     // initialize child with i * 2
        if (child != ID - 1 && result[child+1] < result[child])
            child++;     // find the maxinum child
        if (lastE > result[child])
            result[i] = result[child];    // if proper, change the child with their
parent
        else break;
    }
    result[i] = lastE;    // optimize the last node
    return minE;
}
```

由于关键函数在之前都有分析了，这里就不补充上之前的注释了。