DSRW Task 3 任务描述

任务描述

具体来说，你需要维护一个集合 S，初始为空。有 n 个操作，每个操作是以下两种之一：

- `insert [key] [time]`，其中 `[key]` 是由大小写英文字母和数字组成的长度为 16 的字符串。表示将 `[key]` 插入集合 S。`[time]` 表示该操作发生的时间，其作用将在下面解释。
- `query [key] [time]`，其中 `[key]` 的格式同上。表示询问 `[key]` 是否在 S 中。

False Positive Rate (FPR) 和 False Negative Rate (FNR) 是评价你的程序正确性的指标。我们要求 $\mathrm{FPR} \leq 1\%, \mathrm{FNR} = 0$。

你还需要支持多线程，令 k 表示线程数量，则你需要支持 $k = 2, 4, 8, 16, 32$ 的情形。在计算开始之前，你可以将所有操作载入到内存中，并划分为 k 个子操作序列，供每个线程使用；或者进行其他任意形式的预分配和调度。这一部分的时间和空间开销不计。

为了降低难度，我们为你提供了一种调度方式，可供参考。程序运行时，当前目录下一个文件 `data.in`，描述整个问题的操作序列 Q。同时，有 k 个独立的文件 `data1.in, data2.in, ..., data[k].in`，第 i 个文件描述一个子操作序列 Q_i，且这 k 个子操作序列的并等于完整的操作序列 Q。这个调度方式满足：

- 在满足各个子操作序列 $Q_1, \ldots, Q_k$ 各自顺序不变的前提下，将所有操作以任意顺序排列，合并成为一个完整的操作序列 Q'，则操作序列 Q 与 Q' 的正确答案完全相同。

换言之，若你直接令第 i 个线程执行第 i 个操作序列，则线程之间操作执行的顺序并不会影响你的答案。

你也可以不使用我们的调度方式，而是从 `data.in` 读取整个操作序列，自行完成调度。

`[time]` 描述了一个操作发生的时间。若 N 表示整个任务中的操作数量，则 `[time]` 是一个 $[1, N]$ 之间的正整数，且不同操作的 `[time]` 不同。且对于单个文件描述的操作序列，`[time]` 的值是单调递增的。换言之，完整操作序列 Q 中，第 i 个操作的 `[time]` 值等于 i。当你选择从 k 个不同的文件中读入数据时，你可能需要靠 `[time]` 来判断操作之间的相对顺序，尤其是输出各个询问结果的顺序。

输入格式

你有两种输入方式可供选择：

1. 从 k 个独立的文件 `data1.in` 至 `data[k].in` 中读入，每个文件描述一个子操作序列 Q_i。
2. 从 `data.in` 中读入完整的操作序列 Q。

当我们说某文件描述一个操作序列时，这个文件的第一行含有一个整数 $n \geq 0$，表示操作的数量。接下来 n 行，每行为以下两种格式之一：

1. `insert [key] [time]`。其中 `[key]` 是长度为 16 的、由大小写英文字母及数字组成的字符串；`[time]` 是一个正整数。
2. `query [key] [time]`。`[key]` 和 `[time]` 的格式同上。

此外，线程数量 k 是通过命令行传参的方式输入的。假设你的可执行程序名为 `main`，则我们运行时会使用命令 `./main [k]`，其中 `[k]` 会被替换为 k 的值；例如以 `./main 4` 运行程序，表示你需要使用 4 个线程。下面示例代码可能对你有帮助：

```
#include <cstdlib>

int main(int argc, char **argv) {
  int k = atoi(argv[1]);  // gets k.
}
```

输出格式

将询问的答案输出到 `result.out`。假设完整操作序列 Q 中包含 N_q 个 `query` 操作，则输出 N_q 行，按照 `[time]` 规定的顺序回答各个询问。每行仅包含一个整数 0 或 1，其中 1 表示认为查询的 `[key]` 在集合 S 中；0 认为不在。

你需要将程序的用时输出到 `time.out`。你仅需要计算多线程运行 Bloom Filter 的用时，而不需要将其他部分囊括进来。仅输出一行，包含一个浮点数，表示你的用时，以毫秒为单位，保留三位小数。下面示例代码可能对你有帮助：

```
// Head
#include <chrono>
#include <fstream>
#include <iomanip>
using namespace std;
using namespace chrono;

// Start
auto starting_time = system_clock::now();

// Run Bloom Filter Here.

// End & Output
double time_used = duration_cast<microseconds>(system_clock::now() -
                                               starting_time).count() / 1e3;
ofstream f_time("time.out");
f_time << fixed << setprecision(3) << time_used << endl;
f_time.close();
```

我们将手动检查你的代码，来确保你正确地完成了计时。请不要试图缩小计时范围，或故意输出伪造的计时结果。

样例输入

```
[data.in]
5
query a123456789ABCDEF 1
insert a123456789ABCDEF 2
insert c123456789ABCDEF 3
query a123456789ABCDEF 4
query b123456789ABCDEF 5

[data1.in]
3
query a123456789ABCDEF 1
insert a123456789ABCDEF 2
query a123456789ABCDEF 4
```

```
[data2.in]
2
insert c123456789ABCDEF 3
query b123456789ABCDEF 5
```

运行 `./main 2`。

样例输出

```
[result.out]
0
1
0

[time.out]
12.345
```

上面的输出表示，我们的程序在 Bloom Filter 部分共使用了 12.345 毫秒时间。

请注意，在评测时，你的输出不需要与正确答案完全相同。我们允许至多 1% 的 FPR，即正确答案是 0 的询问中，你可以有至多 1% 误判为 1。但正确答案是 1 的询问，你不能误判成 0。

数据范围

对于完整操作序列 Q，保证 $n \le 3 \times 10^7$，且 `insert` 操作不超过 10^7 个。