广州大学学生实验报告

开课实验室： 电子楼 416B　　2018 年 6 月 3 日

学院	计算机科学与网络工程学院	年级、专业、班	软件 171	姓名	谢金宏	学号	1706300001
实验课程名称	数据结构					成绩	
实验项目名称	实验六 综合实验					指导老师	杜娇

一、实验目的

综合利用课程所学知识完成实验。

二、使用仪器、器材

1. 使用 Windows 操作系统的微型计算机一台。
2. 符合 C++11 标准的 C++编程软件。

三、实验内容及原理

1. 排队系统模拟

设计一个程序模拟银行叫号系统。假设某银行有 5 个服务窗口，一个统一的叫号系统。在服务时段内，每个时间单位有随机数目的顾客到来，每个新来的顾客得到一个顺序号，排入等待服务队列，等待叫号。

每个服务窗口的职员处理完一个顾客的业务后，顾客即刻离开，职员呼叫下一个顾客，窗口显示下一个顾客的服务号。被叫号的顾客随即得到服务。假定每个顾客需要的服务时间也是随机的。试设计程序模拟上述场景。

要求在服务时段内显示如下实时信息和统计信息：

1）每个窗口正在办理业务的顾客的顺序号

2）目前等待顾客人数

3）每个顾客得到服务之前已等待的时间单位数

并在所有顾客离开后，显示如下统计信息：

1）顾客等待的平均时间单位数

2）每个窗口服务的顾客数

2. 基于字符串模式匹配算法的病毒感染检测问题

设计一个程序模拟病毒感染检测问题

医学研究者最近发现了某些新病毒，通过对这些病毒的分析，得知他们的 DNA 序列都是环状的。现在研究者已收集了大量的**病毒 DNA** 和**人的 DNA** 数据，想快速检测出这些人是否感染了相应的病毒。为了方便研究，研究者将人的 DNA 和病毒 DNA 均表示成由一些字母组成的字符串序列，然后**检测某种病毒 DNA 序列**是否在**患者的 DNA 序列中出现过**，如果出现过，这此人感染了该病毒，否则没有感染。例如，假设病毒的 DNA 序列为 baa，患者 1 的 DNA 序列为 aaabbba，则感染。患者 2 的 DNA 序列为 babbba，则未感染。（注意：人的 DNA 序列是线性的，而病毒的 DNA 序列是环状的）。

输入：多组数据，每组数据有 1 行，为序列 A 和 B，A 对应病毒的 DNA 序列，B 对应人的 DNA 序列。

输出：对于每组数据输出 1 行，若患者感染了病毒则输出“YES”；否则输出“NO”。

四、实验过程原始数据记录

本次实验中使用的源代码和输入、输出文件：

队列模拟

基本结构

```
// 客户
struct Client {
    int minutes_left_to_finish;  // 业务剩余需要花费的时间
    Moment arrived_moment;       // 到达的时刻
    Moment served_moment = 0;    // 开始办理业务的时刻
};

// 银行窗口
struct Window {
    int serving_client_seqnum = 0;    // 正在被服务的客户的 ID；为 0 表示窗口空闲。
    int served_client_count   = 0;    // 已完成服务的顾客数量
};
```

辅助函数

```
// 显示 moment 时刻。
void ShowMoment(Moment moment) {
    int hh = moment / 100 % 100;
    int mm = moment % 100;
    cout << "== ";
    cout << (hh < 10 ? to_string(0) + to_string(hh) : to_string(hh));
    cout << ":";
    cout << (mm < 10 ? to_string(0) + to_string(mm) : to_string(mm));
    cout << " ==";
}

// 显示顺序号为 seqnum 的顾客的信息。
void ShowClient(int seqnum) {
    Client &client = wait_queue[seqnum - 1];
    cout << "[顾客" << seqnum << "] ";
    cout << "(剩余" << client.minutes_left_to_finish << "min) ";
    if (client.served_moment == 0) {  // 尚未轮到该客户办理业务。
        cout << "(等待" << current_moment - client.arrived_moment << "min)";
    } else {  // 客户正在办理业务或已经完成业务。
        cout << "(等待" << client.served_moment - client.arrived_moment << "min)";
    }
}


// 显示第 winnum 号窗口的情况。
void ShowWindow(int winnum) {
    Window & window = windows[winnum - 1];
    cout << "{窗口" << winnum <<  "} ";
    if (window.serving_client_seqnum == 0) {  // 窗口空闲
        cout << "空闲";
    } else {
        ShowClient(window.serving_client_seqnum);
    }
    cout << endl;
}

// 显示等待队列
void ShowWaitQueue() {
    for (int i = next_client_seqnum - 1; i < wait_queue.size(); ++i) {
        ShowClient(i + 1); cout << endl;
    }
}
```

主函数

```
int main() {
    Introduction();

    cout << "--------------------------------" << endl;
    OpenBank();
    cout << "银行上班。" << endl;

    cout << "--------------------------------" << endl;
    while (BankIsOpen() || !AllClientsAreDone())
    {
        if (BankIsOpen()) NewClientsArrive();
        WindowsDealWithClients();
        ShowRealtimeStatus();
        ProceedToNextMoment();
        cout << endl;
    }
    cout << "银行下班。" << endl;

    cout << "--------------------------------" << endl;
    ShowOverallStatistics();
}
```

银行模拟

```
// 银行开门（初始化全局变量）。
void OpenBank() {
    current_moment = bank_opens_moment;
    wait_queue.clear();
    windows.clear();
    next_client_seqnum = 1;
    for (int i = 0; i < 5; ++i) {  // 开启 5 个窗口。
        windows.push_back(Window());
    }
}

// 判断银行是否在营业时间内。
bool BankIsOpen() {
    return current_moment <= bank_close_moment;
}
```

```
// 推进一个单位时间。
void ProceedToNextMoment() {
    ++current_moment;
}

// 新客户到达。
void NewClientsArrive() {
    int new_client_count = rand() % 3;  // 限定单位时间新到达的客户不超过 2 人。
    while (new_client_count--) {
        int minute_to_finish = rand() % 8 + 1;  // 限定每位顾客的业务时间不超过 8 分钟。
        int arrived_moment = current_moment;
        wait_queue.push_back(Client{minute_to_finish, arrived_moment, 0});  // 将新到达
的客户加入等待队列。
    }
}

// 窗口处理客户信息
void WindowsDealWithClients() {
    for (int i = 0; i < 5; ++i) {  // 遍历窗口
        Window & window = windows[i];
        if (window.serving_client_seqnum != 0) {  // 窗口忙
            Client & serving_client = wait_queue[window.serving_client_seqnum - 1];
            serving_client.minutes_left_to_finish -= 1;  // 剩余需要花费的业务办理时间减 1

            if (serving_client.minutes_left_to_finish == 0) {  // 客户已办理完业务
                window.serving_client_seqnum = 0;
                window.served_client_count += 1;
            }
        }
        if (window.serving_client_seqnum == 0) {  // 窗口空闲
            if (next_client_seqnum - 1 < wait_queue.size()) { // 若有顾客正在等待办理业务
                window.serving_client_seqnum = next_client_seqnum;
                wait_queue[next_client_seqnum - 1].served_moment = current_moment;
                next_client_seqnum += 1;
            }
        }
    }
}
```

```
// 显示实时信息，
// 包括：显示每个窗口正在办理业务的顾客的顺序号、
// 目前等待的顾客人数以及每个顾客在得到服务之前已经等待的分钟数。
void ShowRealtimeStatus() {
    ShowMoment(current_moment); cout << endl;;

    for (int i = 1; i <= 5; ++i) {
        ShowWindow(i);
    }

    cout << "正在等待的客户数量：" << (int(wait_queue.size()) - next_client_seqnum + 1) <<
endl;
    ShowWaitQueue();
}

// 显示最终统计数据，
// 包括：顾客的平均等待时间和每个窗口服务的顾客数。
void ShowOverallStatistics() {
    // 统计顾客的平均等待时间
    double total_wait_time = 0;
    for (int i = 0; i < wait_queue.size(); ++i) {
        Client & client = wait_queue[i];
        total_wait_time += (
            client.served_moment != 0 ?
                client.served_moment - client.arrived_moment
                : current_moment - client.arrived_moment
        );
    }
    double average_wait_time = total_wait_time / wait_queue.size();
    cout << "顾客平均等待时间：" << average_wait_time << "min" << endl;

    for (int i = 1; i <= 5; ++i) {  // 遍历 5 个窗口，显示每个窗口服务的顾客数量
        cout << i << "号窗口服务的顾客数量：" << windows[i-1].served_client_count << endl;
    }
}
```

```cpp
// 返回所有的客户是否都办理完了业务。
bool AllClientsAreDone() {
    for (int i = 0; i < 5; ++i) {
        if (windows[i].serving_client_seqnum != 0) return false;
    }
    return true;
}
```

病毒检测

主函数

```cpp
int main() {
    int kase;
    while (cin >> kase) {
        while (kase--) {
            string virus, human; cin >> virus >> human;
            cout << (HumanDNAContainsVirusDNA(human, virus) ? "YES" : "NO") << endl;
        }
    }
}
```

检测函数

```cpp
// 判断人类 DNA 是否包含病毒 DNA。
bool HumanDNAContainsVirusDNA(string human, string virus) {
    vector<string> possible_virus_represent(GetPossibleVirusDNARepresent(virus));

    bool matched = false;
    for (auto virus : possible_virus_represent) {
        matched = StringAContainsB(human, virus);
        if (matched) break;
    }
    return matched;
}
```

```cpp
// 获取环状的病毒 DNA 可能的链状表示。
vector<string> GetPossibleVirusDNARepresent(string virus) {
    vector<string> possible_represent;

    size_t length = virus.length();
    virus = virus + virus;
    for (size_t i = 0; i < length; ++i) {
        possible_represent.push_back(virus.substr(i, length));
    }
    return possible_represent;
}
```

KMP 算法

```cpp
// 返回 string a 是否包含 b。（b 为模板串）
bool StringAContainsB(string a, string b) {
    int *next = new int[b.length() + 1];

    function<void(string)> get_next = [&next](string tmp) {
        next[0] = -1;
        int i = -1, j = 0;
        while (j < int(tmp.length())) {
            if (i == -1 || tmp[i] == tmp[j]) ++i, ++j, next[j] = i;
            else i = next[i];
        }
    };
    get_next(b);

    function<bool(string, string)> kmp = [&next](string str, string tmp) {
        int i = 0, j = 0;

        while (i < int(str.length()) && j < int(tmp.length())) {
            if (j == -1 || str[i] == tmp[j]) ++i, ++j;
            else j = next[j];
        }
        return j == tmp.length();
    };
    bool matched = kmp(a, b);

    delete[] next;
    return matched;
}
```

五、实验结果及分析

完成了本次实验的全部要求。基本掌握了数据结构的一般使用和 KMP 算法。
程序的输出请见“实验过程原始数据记录”小节中的输出文本文件。